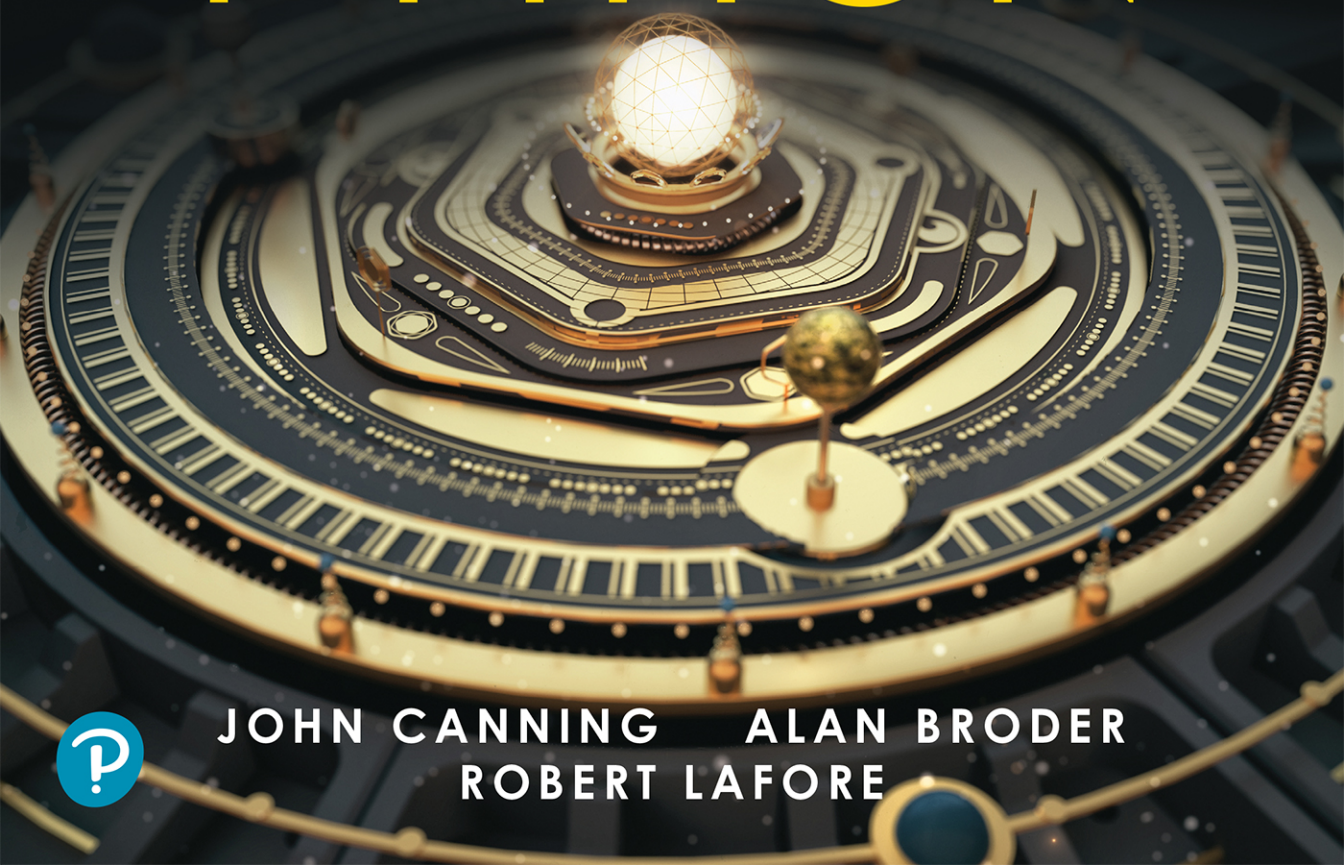
DATA STRUCTURES & ALGORITHMS in PYTHON



JOHN CANNING ALAN BRODER
ROBERT LAFORE

John Canning
Alan Broder
Robert Lafore

Data Structures & Algorithms in Python

Addison-Wesley

Boston · Columbus · New York · San Francisco · Amsterdam · Cape Town
Dubai · London · Madrid · Milan · Munich · Paris · Montreal · Toronto · Delhi · Mexico City
São Paulo · Sydney · Hong Kong · Seoul · Singapore · Taipei · Tokyo

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2022910068

Copyright © 2023 Pearson Education, Inc.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms, and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearson.com/permissions.

No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 978-0-13-485568-4

ISBN-10: 0-13-485568-X

ScoutAutomatedPrintCode

Editor-in-Chief
Mark Taub

Director, ITP Product Management
Brett Bartow

Acquisitions Editor
Kim Spenceley

Development Editor
Chris Zahn

Managing Editor
Sandra Schroeder

Project Editor
Mandie Frank

Copy Editor
Chuck Hutchinson

Indexer
Cheryl Lenser

Proofreader
Barbara Mack

Editorial Assistant
Cindy Teeters

Designer
Chuti Prasertsith

Compositor
codeMantra

Contents at a Glance

1	Overview	1
2	Arrays	29
3	Simple Sorting	75
4	Stacks and Queues	103
5	Linked Lists	157
6	Recursion	229
7	Advanced Sorting	285
8	Binary Trees	335
9	2-3-4 Trees and External Storage	401
10	AVL and Red-Black Trees	463
11	Hash Tables	525
12	Spatial Data Structures	597
13	Heaps	665
14	Graphs	705
15	Weighted Graphs	767
16	What to Use and Why	813

Appendixes

A	Running the Visualizations	833
B	Further Reading	841
C	Answers to Questions	845

Index 859

Table of Contents

1 Overview **1**

What Are Data Structures and Algorithms? 1
Overview of Data Structures 4
Overview of Algorithms 6
Some Definitions 6
Database 6
Record 6
Field 7
Key 7
Databases vs. Data Structures 7
Programming in Python 8
Interpreter 8
Dynamic Typing 12
Sequences 13
Looping and Iteration 15
Multivalued Assignment 17
Importing Modules 18
Functions and Subroutines 19
List Comprehensions 20
Exceptions 22
Object-Oriented Programming 23
Summary 27
Questions 27
Experiments 28

2 Arrays **29**

The Array Visualization Tool 30
Searching 31
The Duplicates Issue 35
Using Python Lists to Implement the Array Class 37
Creating an Array 37
Accessing List Elements 38
A Better Array Class Implementation 43
The OrderedArray Visualization Tool 47
Linear Search 48
Binary Search 48

Python Code for an OrderedArray Class ... 52
Binary Search with the find() Method ... 52
The OrderedArray Class ... 53
Advantages of Ordered Arrays ... 57
Logarithms ... 58
The Equation ... 59
The Opposite of Raising 2 to a Power ... 60
Storing Objects ... 60
The OrderedRecordArray Class ... 61
Big O Notation ... 65
Insertion in an Unordered Array: Constant ... 66
Linear Search: Proportional to N ... 66
Binary Search: Proportional to log(N) ... 67
Don't Need the Constant ... 67
Why Not Use Arrays for Everything? ... 69
Summary ... 69
Questions ... 70
Experiments ... 72
Programming Projects ... 73

3 Simple Sorting 75

How Would You Do It? ... 76
Bubble Sort ... 77
Bubble Sort on the Football Players ... 77
The SimpleSorting Visualization Tool ... 79
Python Code for a Bubble Sort ... 81
Invariants ... 82
Efficiency of the Bubble Sort ... 82
Selection Sort ... 83
Selection Sort on the Football Players ... 83
A Brief Description ... 83
A More Detailed Description ... 83
The Selection Sort in the SimpleSorting Visualization Tool ... 85
Python Code for Selection Sort ... 85
Invariant ... 86
Efficiency of the Selection Sort ... 86
Insertion Sort ... 87
Insertion Sort on the Football Players ... 87
Partial Sorting ... 87
The Marked Player ... 87
The Insertion Sort in the SimpleSorting Visualization Tool ... 89
Python Code for Insertion Sort ... 90

Invariants in the Insertion Sort 91
Efficiency of the Insertion Sort 91
Python Code for Sorting Arrays 91
Stability 96
Comparing the Simple Sorts 96
Summary 98
Questions 98
Experiments 100
Programming Projects 101

4 Stacks and Queues 103

Different Structures for Different Use Cases 103
Storage and Retrieval Pattern 103
Restricted Access 104
More Abstract 104
Stacks 104
The Postal Analogy 105
The Stack Visualization Tool 106
Python Code for a Stack 108
Stack Example 1: Reversing a Word 112
Stack Example 2: Delimiter Matching 113
Efficiency of Stacks 116
Queues 116
A Shifty Problem 117
A Circular Queue 118
The Queue Visualization Tool 119
Python Code for a Queue 120
Efficiency of Queues 125
Deques 125
Priority Queues 126
The PriorityQueue Visualization Tool 127
Python Code for a Priority Queue 129
Efficiency of Priority Queues 132
What About Search and Traversal? 132
Parsing Arithmetic Expressions 132
Postfix Notation 133
Translating Infix to Postfix 134
The InfixCalculator Tool 142
Evaluating Postfix Expressions 148
Summary 151
Questions 152
Experiments 154
Programming Projects 155

5 Linked Lists 157

Links 158
References and Basic Types 160
Relationship, Not Position 164
The LinkedList Visualization Tool 164
The Search Button 166
The Delete Button 166
The New Button 167
The Other Buttons 167
A Simple Linked List 167
The Basic Linked List Methods 168
Traversing Linked Lists 169
Insertion and Search in Linked Lists 170
Deletion in Linked Lists 174
Double-Ended Lists 177
Linked List Efficiency 183
Abstract Data Types and Objects 184
A Stack Implemented by a Linked List 184
A Queue Implemented by a Linked List 187
Data Types and Abstraction 189
ADT Lists 191
ADTs as a Design Tool 191
Ordered Lists 192
Python Code for Ordered Lists 193
Efficiency of Ordered Linked Lists 198
List Insertion Sort 198
Doubly Linked Lists 198
Insertion and Deletion at the Ends 201
Insertion and Deletion in the Middle 204
Doubly Linked List as Basis for Deques 208
Circular Lists 209
Iterators 211
Basic Iterator Methods 212
Other Iterator Methods 216
Iterators in Python 217
Summary 222
Questions 224
Experiments 226
Programming Projects 227

6 Recursion 229

Triangular Numbers 230
Finding the *n*th Term Using a Loop 231

Finding the *n*th Term Using Recursion 232
What's Really Happening? 234
Characteristics of Recursive Routines 235
Is Recursion Efficient? 236
Mathematical Induction 237
Factorials 237
Anagrams 239
A Recursive Binary Search 242
Recursion Replaces the Loop 243
Divide-and-Conquer Algorithms 245
The Tower of Hanoi 245
The TowerofHanoi Visualization Tool 246
Moving Pyramids 247
The Recursive Implementation 250
Sorting with mergesort 255
Merging Two Sorted Arrays 255
Sorting by Merging 257
Merging Subranges 260
Testing the Code 262
The Mergesort Visualization Tool 263
Efficiency of the mergesort 264
Eliminating Recursion 267
Recursion and Stacks 267
Simulating a Recursive Function: Triangular 268
Rewriting a Recursive Procedure: mergesort 270
Some Interesting Recursive Applications 275
Raising a Number to a Power 275
The Knapsack Problem 277
Combinations: Picking a Team 278
Summary 280
Questions 281
Experiments 283
Programming Projects 283

7 Advanced Sorting 285

Shellsort 285
Insertion Sort: Too Many Copies 286
N-Sorting 286
Diminishing Gaps 288
The AdvancedSorting Visualization Tool 289

Python Code for the Shellsort 291
Other Interval Sequences 293
Efficiency of the Shellsort 294
Partitioning 294
The Partition Process 295
The General Partitioning Algorithm 297
Efficiency of the Partition Algorithm 301
Quicksort 302
The Basic Quicksort Algorithm 302
Choosing a Pivot Value 304
A First Quicksort Implementation 306
Running Quicksort in the AdvancedSorting Visualization Tool 309
The Details 310
Degenerates to $O(N^2)$ Performance 312
Median-of-Three Partitioning 313
Handling Small Partitions 315
The Full Quicksort Implementation 315
Removing Recursion 318
Efficiency of Quicksort 318
Radix Sort 320
Algorithm for the Radix Sort 321
Designing a Radix Sort Program 321
Efficiency of the Radix Sort 322
Generalizing the Radix Sort 322
Using a Counting Sort 323
Timsort 324
Efficiency of Timsort 327
Summary 327
Questions 329
Experiments 331
Programming Projects 332

8 Binary Trees 335

Why Use Binary Trees? 335
Slow Insertion in an Ordered Array 335
Slow Searching in a Linked List 336
Trees to the Rescue 336
What Is a Tree? 336
Tree Terminology 337
Root 338
Path 338
Parent 338

Child 338
Sibling 339
Leaf 339
Subtree 339
Visiting 339
Traversing 339
Levels 339
Keys 339
Binary Trees 339
Binary Search Trees 340
An Analogy 340
How Do Binary Search Trees Work? 341
The Binary Search Tree Visualization Tool 341
Representing the Tree in Python Code 344
Finding a Node 346
Using the Visualization Tool to Find a Node 346
Python Code for Finding a Node 348
Tree Efficiency 350
Inserting a Node 350
Using the Visualization Tool to Insert a Node 351
Python Code for Inserting a Node 352
Traversing the Tree 353
In-order Traversal 353
Pre-order and Post-order Traversals 355
Python Code for Traversing 356
Traversing with the Visualization Tool 361
Traversal Order 363
Finding Minimum and Maximum Key Values 365
Deleting a Node 366
Case 1: The Node to Be Deleted Has No Children 367
Case 2: The Node to Be Deleted Has One Child 367
Case 3: The Node to Be Deleted Has Two Children 370
The Efficiency of Binary Search Trees 375
Trees Represented as Arrays 377
Tree Levels and Size 378
Printing Trees 379
Duplicate Keys 381
The BinarySearchTreeTester.py Program 382
The Huffman Code 386
Character Codes 386
Decoding with the Huffman Tree 388
Creating the Huffman Tree 389

Coding the Message 391
Summary 393
Questions 394
Experiments 396
Programming Projects 397

9 2-3-4 Trees and External Storage 401

Introduction to 2-3-4 Trees 401
What's in a Name? 402
2-3-4 Tree Terminology 403
2-3-4 Tree Organization 403
Searching a 2-3-4 Tree 404
Insertion 404
Node Splits 405
Splitting the Root 406
Splitting on the Way Down 407
The Tree234 Visualization Tool 408
The Random Fill and New Tree Buttons 409
The Search Button 409
The Insert Button 409
Zooming and Scrolling 410
Experiments 411
Python Code for a 2-3-4 Tree 412
The __Node Class 412
The Tree234 Class 415
Traversal 421
Deletion 423
Efficiency of 2-3-4 Trees 430
Speed 431
Storage Requirements 432
2-3 Trees 432
Node Splits 433
Promoting Splits to Internal Nodes 435
Implementation 437
Efficiency of 2-3 Trees 438
External Storage 438
Accessing External Data 439
Sequential Ordering 442
B-Trees 444
Indexing 450
Complex Search Criteria 452
Sorting External Files 453

Summary 456
Questions 458
Experiments 459
Programming Projects 460

10 AVL and Red-Black Trees 463

Our Approach to the Discussion 463
Balanced and Unbalanced Trees 464
Degenerates to O(N) 464
Measuring Tree Balance 465
How Much Is Unbalanced? 468
AVL Trees 470
The AVLTree Visualization Tool 471
Inserting Items with the AVLTree Visualization Tool 472
Python Code for the AVL Tree 474
The Efficiency of AVL Trees 486
Red-Black Trees 487
Conceptual 487
Top-Down Insertion 487
Bottom-Up Insertion 488
Red-Black Tree Characteristics 488
Using the Red-Black Tree Visualization Tool 489
Flipping a Node's Color 490
Rotating Nodes 491
The Insert Button 492
The Search Button 492
The Delete Button 492
The Erase & Random Fill Button 492
Experimenting with the Visualization Tool 492
Experiment 1: Inserting Two Red Nodes 493
Experiment 2: Rotations 493
Experiment 3: Color Swaps 494
Experiment 4: An Unbalanced Tree 495
More Experiments 496
The Red-Black Rules and Balanced Trees 496
Null Children 496
Rotations in Red-Black Trees 497
Subtrees on the Move 497
Inserting a New Node 498
Preview of the Insertion Process 499
Color Swaps on the Way Down 499
Rotations After the Node Is Inserted 501

Rotations on the Way Down 505
Deletion 508
The Efficiency of Red-Black Trees 509
2-3-4 Trees and Red-Black Trees 510
Transformation from 2-3-4 to Red-Black 510
Operational Equivalence 512
Red-Black Tree Implementation 514
Summary 515
Questions 517
Experiments 520
Programming Projects 521

11 Hash Tables 525

Introduction to Hashing 526
Bank Account Numbers as Keys 526
A Dictionary 527
Hashing 530
Collisions 533
Open Addressing 536
Linear Probing 536
Python Code for Open Addressing Hash Tables 544
Quadratic Probing 554
Double Hashing 559
Separate Chaining 565
The HashTableChaining Visualization Tool 566
Python Code for Separate Chaining 569
Hash Functions 575
Quick Computation 575
Random Keys 575
Nonrandom Keys 576
Hashing Strings 578
Folding 580
Hashing Efficiency 581
Open Addressing 581
Separate Chaining 583
Open Addressing Versus Separate Chaining 587
Hashing and External Storage 588
Table of File Pointers 588
Nonfull Blocks 588
Full Blocks 589

Summary ... 590
Questions ... 592
Experiments ... 594
Programming Projects ... 595

12 Spatial Data Structures 597

Spatial Data ... 597
Cartesian Coordinates ... 597
Geographic Coordinates ... 598
Computing Distances Between Points ... 599
Distance Between Cartesian Coordinates ... 599
Circles and Bounding Boxes ... 601
Clarifying Distances and Circles ... 601
Bounding Boxes ... 603
The Bounding Box of a Query Circle in Cartesian Coordinates ... 603
The Bounding Box of a Query Circle in Geographic Coordinates ... 604
Implementing Bounding Boxes in Python ... 605
The CircleBounds Subclass ... 607
Determining Whether Two Bounds Objects Intersect ... 609
Determining Whether One Bounds Object Lies Entirely Within Another ... 610
Searching Spatial Data ... 611
Lists of Points ... 612
Creating an Instance of the PointList Class ... 612
Inserting Points ... 613
Finding an Exact Match ... 614
Deleting a Point ... 614
Traversing the Points ... 615
Finding the Nearest Match ... 615
Grids ... 617
Implementing a Grid in Python ... 618
Creating an Instance of the Grid Class ... 619
Inserting Points ... 620
Finding an Exact Match ... 621
Big O and Practical Considerations ... 622
Deleting and Traversing ... 623
Finding the Nearest Match ... 624
Does the Query Circle Fall Within a Layer? ... 625
Does the Query Circle Intersect a Grid Cell? ... 628
Generating the Sequence of Neighboring Cells to Visit ... 629

Pulling It All Together: Implementing Grid's findNearest() 630
Quadtrees 633
Creating an Instance of the QuadTree Class 635
Inserting Points: A Conceptual Overview 636
Avoiding Ambiguity 638
The QuadTree Visualization Tool 639
Implementing Quadtrees: The Node Class 640
The insert Method 641
Efficiency of Insertion 644
Finding an Exact Match 644
Efficiency of Exact Search 645
Traversing the Points 645
Deleting a Point 646
Finding the Nearest Match 647
Finding a Candidate Node 648
Finding the Closest Node 649
Pulling It All Together: Implementing QuadTree's findNearest() 652
Efficiency of findNearest() 655
Theoretical Performance and Optimizations 656
Practical Considerations 656
Further Extensions 658
Other Operations 658
Higher Dimensions 658
Summary 659
Questions 661
Experiments 662
Programming Projects 663

13 Heaps 665

Introduction to Heaps 666
Priority Queues, Heaps, and ADTs 667
Partially Ordered 668
Insertion 669
Removal 670
Other Operations 674
The Heap Visualization Tool 674
The Insert Button 675
The Make Random Heap Button 676
The Erase and Random Fill Button 676
The Peek Button 677
The Remove Max Button 677
The Heapify Button 677

The Traverse Button 677
Python Code for Heaps 677
Insertion 679
Removal 680
Traversal 682
Efficiency of Heap Operations 683
A Tree-Based Heap 684
Heapsort 686
Sifting Down Instead of Up 686
Using the Same Array 688
The heapsort() Subroutine 691
The Efficiency of Heapsort 693
Order Statistics 694
Partial Ordering Assists in Finding the Extreme Values 695
The Efficiency of K Highest 696
Summary 700
Questions 701
Experiments 703
Programming Projects 703

14 Graphs 705

Introduction to Graphs 705
Definitions 706
The First Uses of Graphs 708
Representing a Graph in a Program 709
Adding Vertices and Edges to a Graph 713
The Graph Class 715
Traversal and Search 718
Depth-First 719
Breadth-First 729
Minimum Spanning Trees 734
Minimum Spanning Trees in the Graph Visualization Tool 735
Trees Within a Graph 736
Python Code for the Minimum Spanning Tree 737
Topological Sorting 740
Dependency Relationships 741
Directed Graphs 742
Sorting Directed Graphs 742
The Graph Visualization Tool 743
The Topological Sorting Algorithm 744
Cycles and Trees 745
Python Code for the Basic Topological Sort 746

Improving the Topological Sort 748
Connectivity in Directed Graphs 753
The Connectivity Matrix 753
Transitive Closure and Warshall's Algorithm 753
Implementation of Warshall's Algorithm 758
Summary 759
Questions 760
Experiments 762
Programming Projects 763

15 Weighted Graphs 767

Minimum Spanning Tree with Weighted Graphs 767
An Example: Networking in the Jungle 768
The WeightedGraph Visualization Tool 768
Building the Minimum Spanning Tree: Send Out the Surveyors 770
Creating the Algorithm 775
The Shortest-Path Problem 783
Travel by Rail 783
Dijkstra's Algorithm 784
Agents and Train Rides 784
Finding Shortest Paths Using the Visualization Tool 790
Implementing the Algorithm 794
Python Code 795
The All-Pairs Shortest-Path Problem 797
Efficiency 800
Intractable Problems 801
The Knight's Tour 802
The Traveling Salesperson Problem 802
Hamiltonian Paths and Cycles 803
Summary 805
Questions 806
Experiments 808
Programming Projects 809

16 What to Use and Why 813

Analyzing the Problem 814
What Kind of Data? 814
How Much Data? 815
What Operations and How Frequent? 816
Who Will Maintain the Software? 817
Foundational Data Structures 818

Speed and Algorithms 819
Libraries 820
Arrays 820
Linked Lists 821
Binary Search Trees 822
Balanced Search Trees 822
Hash Tables 823
Comparing the General-Purpose Storage Structures 824
Special-Ordering Data Structures 824
Stack 825
Queue 825
Priority Queue 826
Comparison of Special-Ordering Structures 826
Sorting 826
Specialty Data Structures 828
Quadtrees and Grids 828
Graphs 828
External Storage 829
Sequential Storage 829
Indexed Files 829
B-trees 830
Hashing 830
Choosing Among External Storage Types 830
Virtual Memory 830
Onward 831

Appendixes

A Running the Visualizations 833

For Developers: Running and Changing the Visualizations 834
Getting Python 834
Getting Git 835
Getting the Visualizations 835
For Managers: Downloading and Running the Visualizations 836
For Others: Viewing the Visualizations on the Internet 837
Using the Visualizations 838

B Further Reading 841

Data Structures and Algorithms 841

Object-Oriented Programming Languages 842
Object-Oriented Design (OOD) and Software Engineering 842

C Answers to Questions 845

Chapter 1, "Overview" 845
Chapter 2, "Arrays" 846
Chapter 3, "Simple Sorting" 847
Chapter 4, "Stacks and Queues" 848
Chapter 5, "Linked Lists" 848
Chapter 6, "Recursion" 849
Chapter 7, "Advanced Sorting" 850
Chapter 8, "Binary Trees" 851
Chapter 9, "2-3-4 Trees and External Storage" 851
Chapter 10, "AVL and Red-Black Trees" 852
Chapter 11, "Hash Tables" 853
Chapter 12, "Spatial Data Structures" 853
Chapter 13, "Heaps" 854
Chapter 14, "Graphs" 855
Chapter 15, "Weighted Graphs" 856

Index 859

Pearson's Commitment to Diversity, Equity, and Inclusion

Pearson is dedicated to creating bias-free content that reflects the diversity of all learners. We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, socioeconomic status, ability, age, sexual orientation, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world's leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where

- Everyone has an equitable and lifelong opportunity to succeed through learning
- Our educational products and services are inclusive and represent the rich diversity of learners
- Our educational content accurately reflects the histories and experiences of the learners we serve
- Our educational content prompts deeper discussions with learners and motivates them to expand their own learning (and worldview)

While we work hard to present unbiased content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

Please contact us with concerns about any potential bias at https://www.pearson.com/report-bias.html.

To my mother, who gave me a thirst for knowledge,
to my father, who taught me the joys of engineering,
and to June, who made it possible to pursue both.

John Canning

For my father Sol Broder, computer science pioneer,
for leading the way.
To my mother Marilyn Broder, master educator,
for inspiring me to teach.
To Fran, for making my life complete.

Alan Broder

Register your copy of *Data Structures & Algorithms in Python* at informit.com for convenient access to downloads, updates, and corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN 9780134855684 and click Submit. Once the process is complete, you will find any available bonus content under "Registered Products."

Visit the book's website, https://datastructures.live, to join the conversation with other readers about the book, and learn more about the visualizations that bring data structures to life.

Acknowledgments

From John Canning and Alan Broder

Robert Lafore's Java-based version of this book has been a mainstay in Data Structures courses and professionals' reference shelves around the world for many years. When Alan's Data Structures course at Stern College for Women of Yeshiva University moved on to Python, the inability to use Lafore's book in the course was a real loss. We're thus especially happy to bring this new and revised edition to the world of Python programmers and students.

We'd like to thank the many students at Stern who contributed to this book either directly or indirectly over the past several years. Initial Python versions of Lafore's Java implementations were central to Alan's Python-based courses, and Stern student feedback helped improve the code's clarity, enhanced its performance, and sometimes even identified and fixed bugs!

For their valuable feedback and recommendations on early drafts of this new edition, we are grateful to many students in Alan's Data Structures courses, including Estee Brooks, Adina Bruce, Julia Chase, Hanna Fischer, Limor Kohanim, Elisheva Kohn, Shira Orlian, Shira Pahmer, Jennie Peled, Alexandra Roffe, Avigail Royzenberg, Batia Segal, Penina Waghalter, and Esther Werblowsky. Our apologies if we've omitted anyone's name.

An open-source package of data structure visualizations is available to enhance your study of this book, and Stern students played an active role in the development of the visualization software. John and Alan extend many thanks to the Stern student pioneers and leaders of this project, including Ilana Radinsky, Elana Apfelbaum, Ayliana Teitelbaum, and Lily Polonetsky, as well as the following past and present Stern student contributors and mentors: Zoe Abboudi, Ayelet Aharon, Lara Amar, Natania Birnbaum, Adina Bruce, Chani Dubin, Sarah Engel, Sarah Graff, Avigayil Helman, Michal Kaufman, Sarina Kofman, Rachel Leiser, Talia Leitner, Shani Lewis, Rina Melincoff, Atara Neugroschl, Shira Pahmer, Miriam Rabinovich, Etta Rapp, Shira Sassoon, Shira Schneider, Mazal Schoenwald, Shira Smith, Riva Tropp, Alexandra Volchek, and Esther Werblowsky. Also, many thanks to the Stern faculty who mentored student participants: Professor Ari Shamash, Professor Lawrence Teitelman, and Professor Joshua Waxman. Our apologies if we have left anyone off this list.

Many thanks go to Professor David Matuszek of the University of Pennsylvania for his early contributions of ideas and PowerPoint slides when Alan first started teaching Data Structures at Stern. Many of the slides available in the Instructors Resources section have their origin in his clear and well-designed slides. Also, we are grateful to Professor Marian Gidea of the Department of Mathematics of Yeshiva University for his insights into spherical trigonometry.

Finally, we owe a great debt to the talented editors at Pearson who made this book a reality: Mark Taber, Kim Spenceley, Mandie Frank, Chuti Prasertsith, and Chris Zahn. Without their many talents and patient help, this project would just be an odd collection of text files, drawings, and source code.

From Robert Lafore for the Java-based versions of the book

Acknowledgments to the First Edition, Data Structures and Algorithms in Java

My gratitude for the following people (and many others) cannot be fully expressed in this short acknowledgment. As always, Mitch Waite had the Java thing figured out before anyone else. He also let me bounce the applets off him until they did the job, and extracted the overall form of the project from a miasma of speculation. My editor, Kurt Stephan, found great reviewers, made sure everyone was on the same page, kept the ball rolling, and gently but firmly ensured that I did what I was supposed to do. Harry Henderson provided a skilled appraisal of the first draft, along with many valuable suggestions. Richard S. Wright, Jr., as technical editor, corrected numerous problems with his keen eye for detail. Jaime Niño, Ph.D., of the University of New Orleans, attempted to save me from myself and occasionally succeeded, but should bear no responsibility for my approach or coding details. Susan Walton has been a staunch and much-appreciated supporter in helping to convey the essence of the project to the nontechnical. Carmela Carvajal was invaluable in extending our contacts with the academic world. Dan Scherf not only put the CD-ROM together, but was tireless in keeping me up to date on rapidly evolving software changes. Finally, Cecile Kaufman ably shepherded the book through its transition from the editing to the production process.

Acknowledgments to the Second Edition

My thanks to the following people at Sams Publishing for their competence, effort, and patience in the development of this second edition. Acquisitions Editor Carol Ackerman and Development Editor Songlin Qiu ably guided this edition through the complex production process. Project Editor Matt Purcell corrected a semi-infinite number of grammatical errors and made sure everything made sense. Tech Editor Mike Kopak reviewed the programs and saved me from several problems. Last but not least, Dan Scherf, an old friend from a previous era, provides skilled management of my code and applets on the Sams website.

About the Authors

Dr. John Canning is an engineer, computer scientist, and researcher. He earned an S.B. degree in electrical engineering from the Massachusetts Institute of Technology and a Ph.D. in Computer Science from the University of Maryland at College Park. His varied professions include being a professor of computer science, a researcher and software engineer in industry, and a company vice president. He now is president of Shakumant Software.

Alan Broder is clinical professor and chair of the Department of Computer Science at Stern College for Women of Yeshiva University in New York City. He teaches introductory and advanced courses in Python programming, data structures, and data science. Before joining Stern College, he was a software engineer, designing and building large-scale data analysis systems. He founded and led White Oak Technologies, Inc. as its CEO, and later served as the chairman and fellow of its successor company, Novetta, in Fairfax, Virginia.

Robert Lafore has degrees in Electrical Engineering and Mathematics, has worked as a systems analyst for the Lawrence Berkeley Laboratory, founded his own software company, and is a best-selling writer in the field of computer programming. Some of his titles are *Object-Oriented Programming in C++* and *Data Structures and Algorithms in Java*.

Introduction

What's in this book? This book is designed to be a practical introduction to data structures and algorithms for students who have just begun to write computer programs. This introduction will tell you more about the book, how it is organized, what experience we expect readers will have before starting the book, and what knowledge you will get by reading it and doing the exercises.

Who This Book Is For

Data structures and algorithms are the core of computer science. If you've ever wanted to understand what computers can do, how they do it, and what they can't do, then you need a deep understanding of both (it's probably better to say "what computers have difficulty doing" instead of what they can't do). This book may be used as a text in a data structures and/or algorithms course, frequently taught in the second year of a university computer science curriculum. The text, however, is also designed for professional programmers, for high school students, and for anyone else who needs to take the next step up from merely knowing a programming language. Because it's easy to understand, it is also appropriate as a supplemental text to a more formal course. It is loaded with examples, exercises, and supplemental materials, so it can be used for self-study outside of a classroom setting.

Our approach in writing this book is to make it easy for readers to understand how data structures operate and how to apply them in practice. That's different from some other texts that emphasize the mathematical theory, or how those structures are implemented in a particular language or software library. We've selected examples with real-world applications and avoid using math-only or obscure examples. We use figures and visualization programs to help communicate key ideas. We still cover the complexity of the algorithms and the math needed to show how complexity impacts performance.

What You Need to Know Before You Read This Book

The prerequisites for using this book are: knowledge of some programming language and some mathematics. Although the sample code is written in Python, you don't need to know Python to follow what's happening. Python is not hard to understand, if you've done some procedural and/or object-oriented programming. We've kept the syntax in the examples as general as possible,

More specifically, we use Python version 3 syntax. This version differs somewhat from Python 2, but not greatly. Python is a rich language with many built-in data types and libraries that extend its capabilities. Our examples, however, use the more basic constructs for two reasons: it makes them easier to understand for programmers familiar with other languages, and it illustrates the details of the data structures more explicitly. In later chapters, we do make use of some Python features not found in other languages such as generators and list comprehensions. We explain what these are and how they benefit the programmer.

Of course, it will help if you're already familiar with Python (version 2 or 3). Perhaps you've used some of Python's many data structures and are curious about how they are implemented. We review Python syntax in Chapter 1, "Overview," for those who need an introduction or refresher. If you've programmed in languages like Java, C++, C#, JavaScript, or Perl, many of the constructs should be familiar. If you've only programmed using functional or domain-specific languages, you may need to spend more time becoming familiar with basic elements of Python. Beyond this text, there are many resources available for novice Python programmers, including many tutorials on the Internet.

Besides a programming language, what should every programmer know? A good knowledge of math from arithmetic through algebra is essential. Computer programming is symbol manipulation. Just like algebra, there are ways of transforming expressions to rearrange terms, put them in different forms, and make certain parts more prominent, all while preserving the same meaning. It's also critical to understand exponentials in math. Much of computer science is based on knowing what raising one number to a power of another means. Beyond math, a good sense of organization is also beneficial for all programming. Knowing how to organize items in different ways (by time, by function, by size, by complexity, and so on) is crucial to making programs efficient and maintainable. When we talk about efficiency and maintainability, they have particular meanings in computer science. *Efficiency* is mostly about how much time it takes to compute things but can also be about the amount of space it takes. *Maintainability* refers to the ease of understanding and modifying your programs by other programmers as well as yourself.

You'll also need knowledge of how to find things on the Internet, download and install software, and run them on a computer. The instructions for downloading and running the visualization programs can be found in Appendix A of this book. The Internet has made it very easy to access a cornucopia of tools, including tools for learning programming and computer science. We expect readers to already know how to find useful resources and avoid sources that might provide malicious software.

What You Can Learn from This Book

As you might expect from its title, this book can teach you about how data structures make programs (and programmers) more efficient in their work. You can learn how data organization and its coupling with appropriate algorithms greatly affect what can be computed with a given amount of computing resources. This book can give you a thorough understanding of how to implement the data structures, and that should enable you to implement them in any programming language. You can learn the process of deciding what data structure(s) and algorithms are the most appropriate to meet a particular programming request. Perhaps most importantly, you can learn when an algorithm and/or data structure will *fail* in a given use case. Understanding data structures and algorithms is the core of computer science, which is different from being a Python (or other language) programmer.

The book teaches the fundamental data structures that every programmer should know. Readers should understand that there are many more. These basic data structures work in a wide variety of situations. With the skills you develop in this book, you should be able

to read a description of another data structure or algorithm and begin to analyze whether or not it will outperform or perform worse than the ones you've already learned in particular use cases.

This book explains some Python syntax and structure, but it will not teach you all its capabilities. The book uses a subset of Python's full capabilities to illustrate how more complex data structures are built from the simpler constructs. It is not designed to teach the basics of programming to someone who has never programmed. Python is a very high-level language with many built-in data structures. Using some of the more primitive types such as arrays of integers or record structures, as you might find in C or C++, is somewhat more difficult in Python. Because the book's focus is the implementation and analysis of data structures, our examples use approximations to these primitive types. Some Python programmers may find these examples unnecessarily complex, knowing about the more elegant constructs provided with the language in standard libraries. If you want to understand computer science, and in particular, the complexity of algorithms, you must understand the underlying operations on the primitives. When you use a data structure provided in a programming language or from one of its add-on modules, you will often have to know its complexity to know whether it will work well for your use case. Understanding the core data structures, their complexities, and trade-offs will help you understand the ones built on top of them.

All the data structures are developed using ***object-oriented programming*** (OOP). If that's a new concept for you, the review in Chapter 1 of how classes are defined and used in Python provides a basic introduction to OOP. You should not expect to learn the full power and benefits of OOP from this text. Instead, you will learn to implement each data structure as a ***class***. These classes are the *types of objects* in OOP and make it easier to develop software that can be reused by many different applications in a reliable way.

The book uses many examples, but this is not a book about a particular application area of computer science such as databases, user interfaces, or artificial intelligence. The examples are chosen to illustrate typical applications of programs, but all programs are written in a particular context, and that changes over time. A database program written in 1970 may have appeared very advanced at that time, but it might seem very trivial today. The examples presented in this text are designed to teach how data structures are implemented, how they perform, and how to compare them when designing a new program. The examples should not be taken as the most comprehensive or best implementation possible of each data structure, nor as a thorough review of all the potential data structures that could be appropriate for a particular application area.

Structure

Each chapter presents a particular group of data structures and associated algorithms. At the end of the chapters, we provide review questions covering the key points in the chapter and sometimes relationships to previous chapters. The answers for these can be found in Appendix C, "Answers to Questions." These questions are intended as a self-test for readers, to ensure you understood all the material.

Many chapters suggest *experiments* for readers to try. These can be individual thought experiments, team assignments, or exercises with the software tools provided with the book. These are designed to apply the knowledge just learned to some other area and help deepen your understanding.

Programming projects are longer, more challenging programming exercises. We provide a range of projects of different levels of difficulty. These projects might be used in classroom settings as homework assignments. Sample solutions to the programming projects are available to qualified instructors from the publisher and the website, https://datastructures.live.

History

Mitchell Waite and Robert Lafore developed the first version of this book and titled it *Data Structures and Algorithms in Java*. The first edition was published in 1998, and the second edition, by Robert, came out in 2002. John Canning and Alan Broder developed this version using Python due to its popularity in education and commercial and noncommercial software development. Java is widely used and an important language for computer scientists to know. With many schools adopting Python as a first programming language, the need for textbooks that introduce new concepts in an already familiar language drove the development of this book. We expanded the coverage of data structures and updated many of the examples.

We've tried to make the learning process as painless as possible. We hope this text makes the core, and frankly, the beauty of computer science accessible to all. Beyond just understanding, we hope you find learning these ideas fun. Enjoy yourself!

Figure Credits

Figures	Credit/Attribution
Figures 2.1–2.4, 2.6, 3.5, 3.7, 3.9, 4.2, 4.6, 4.10, 4.11, 5.3, 5.4, 5.9, 6.10, 6.18, 7.3, 7.4, 7.6, 7.10, 7.11, 8.5, 8.8, 8.9, 8.11, 8.14, 8.15, 9.7–9.9, 10.09-10.11(a-c), 10.14–10.16, 10.29(a-b), 11.06–11.10, 11.14, 11.15, 11.18, 11.19, 12.24, 13.7, 13.8, 13.9, 13.13, 14.6, 14.12, 14.17, 15.02, 15.9, 15.14–15.18, A.01-A.04	Shakumant Software
Figure 6.1	Courtesy of Droste B.V.
Figure 13.15	word cloud produced by www.wordclouds.com.
Cover	archy13/Shutterstock

CHAPTER 1

Overview

IN THIS CHAPTER

- What Are Data Structures and Algorithms?
- Overview of Data Structures
- Overview of Algorithms
- Some Definitions
- Programming in Python
- Object-Oriented Programming

You have written some programs and learned enough to think that programming is fun, or at least interesting. Some parts are easy, and some parts are hard. You'd like to know more about how to make the process easier, get past the hard parts, and conquer more complex tasks. You are starting to study the heart of computer science, and that brings up many questions. This chapter sets the stage for learning how to make programs that work properly and fast. It explains a bunch of new terms and fills in background about the programming language that we use in the examples.

What Are Data Structures and Algorithms?

Data organizations are ways data is arranged in the computer using its various storage media (such as random-access memory, or RAM, and disk) and how that data is interpreted to represent something. **Algorithms** are the procedures used to manipulate the data in these structures. The way data is arranged can simplify the algorithms and make algorithms run faster or slower. Together, the data organization and the algorithm form a **data structure**. The data structures act like building blocks, with more complex data structures using other data structures as components with appropriate algorithms.

Does the way data is arranged and the algorithm used to manipulate it make a difference? The answer is a definite yes. From the perspective of nonprogrammers, it often seems as though computers can do anything and do it very fast. That's not really true. To see why, let's look at a non-programming example.

When you cook a meal, a collection of ingredients needs to be combined and manipulated in specific ways. There is a huge variety of ways that you could go about the individual steps needed to complete the meal. Some of those methods are going to be more efficient than others. Let's assume that you have a written recipe, but are working in an unfamiliar kitchen, perhaps while visiting a friend. One method for making the meal would be *Method A*:

1. Read through the full recipe noting all the ingredients it mentions, their quantities, and any equipment needed to process them.
2. Find the ingredients, measure out the quantity needed, and store them.
3. Get out all the equipment needed to complete the steps in the recipe.
4. Go through the steps of the recipe in the order specified.

Let's compare that to *Method B*:

1. Read the recipe until you identify the first set of ingredients or equipment that is needed to complete the first step.
2. Find the identified ingredients or equipment.
3. Measure any ingredients found.
4. Perform the first step with the equipment and ingredients already found.
5. Return to the beginning of this method and repeat the instructions replacing the word *first* with *next*. If there is no next step, then quit.

Both methods are *complete* in that that they should finish the complete recipe if all the ingredients and equipment are available. For simple recipes, they should take about the same amount of time too. The methods differ as the recipes get more complex. For example, what if you can't find the fifth ingredient? In method A, that issue is identified at the beginning before any other ingredients are combined. While neither method really explains what to do about exceptions like a missing ingredient, you can still compare them under the assumption that you handle the exceptions the same way.

A missing ingredient could be handled in several ways: find a substitute ingredient, broaden the search for the ingredient (look in other rooms, ask a neighbor, go to the market), or ignore the ingredient (an optional garnish). Each of those remedies takes some time. If there is one missing ingredient and two cooks using the different methods handle it in the same way, both are delayed the same amount of time. If there are multiple missing ingredients, however, Method A should identify those earlier and allow for the possibility of getting all the missing ingredients in one visit to a neighbor or a market. The time savings of combining the tasks of replacing missing ingredients could be significant (imagine the market being far away or neighbors who want to talk for hours on every visit).

The order of performing the operations could have significant impact on the time needed to complete the meal. Another difference could be in the quality of the meal. For example, in Method B the cook would perform the first "step" and then move on to the next step. Let's assume those steps use two different groups of ingredients or equipment. If finding or measuring the ingredients, or getting the equipment for the later steps takes significant time, then the results of the first step sit around for a significant time. That can have a bad effect on the quality of the meal. The cook might be able to overcome that effect in some circumstances by, say, putting the results of the first step in a freezer or refrigerator and then bringing them back to room temperature later. The cook would be preserving the quality of the food at the expense of the time needed to prepare it.

Would Method B ever be desirable if it takes longer or risks degrading the quality of the food? Perhaps. Imagine that the cook is preparing this meal in the unfamiliar kitchen of a family relative. The kitchen is full of family members, and each one is trying to make part of the meal. In this crowded situation, it could be difficult for each cook to get out all of their ingredients and equipment at once. There might not be enough counter space, or mixing bowls, or knives, for example, for each cook to have all their items assembled at the beginning. The cooks could be constrained to work on individual steps while waiting for equipment, space, or ingredients to become available. In this case, Method B could have advantages over asking all the cooks to work one at a time using Method A.

Coming back to programming, the algorithm specifies the sequence of operations that are to be performed, much like the steps in the recipe. The data organizations are somewhat analogous to how the ingredients are stored, laid out in the kitchen, and their proximity to other ingredients and equipment. For example, having the ingredient in the kitchen makes the process much faster than if the ingredient needs to be retrieved from a neighbor or the market. You can think of the amount of space taken up by spreading out the ingredients in various locations as the amount of space needed for the algorithm. Even if all the ingredients are in the kitchen where the cook is, there are ways of setting up the ingredients to make the cooking tasks go faster. Having them compactly arranged in the order they are needed minimizes the amount of moving around the cook must do. The organization of the ingredients can also help if a cook must be replaced by another cook in the middle of the preparation; understanding where each of the ingredients fits in to the recipe is faster if the layout is organized. This is another reason why good data organization is important. It also reinforces that concept that the algorithm and the data organization work together to make the data structure.

Data structures are important not just for speed but also to properly model the meaning of the data. Let's say there's an event that many people want to attend and they need to submit their phone number to have a chance to get tickets. Each person can request multiple tickets. If there are fewer tickets than the number of people who want to get them, some method needs to be applied to decide which phone numbers to contact first and determine the number of tickets they will receive. One method would be to go through the phone numbers one at a time and total up the number of tickets until all are given out, then go through the remaining numbers to let them know the tickets are gone. That might be a fair method if the numbers were put in an ordered list defined in a way that

potential recipients understood— for example, a chronological list of phone numbers submitted by people interested in the tickets. If the tickets are to be awarded as a sort of lottery, then going through them sequentially means following any bias that is implicit in the order of the list. Randomly choosing a number from the list, contacting the buyer, and then removing the number would be fairer in a lottery system.

Data structures *model systems by assigning specific meanings to each of the pieces and how they interact*. The "systems" are real-world things like first-come, first-served ticket sales, or a lottery giveaway, or how roads connect cities. For the list of phone numbers with ticket requests, a first-come, first-served system needs the list in chronological order, some pointer to the next number that should be taken, and a pointer to where any newly arriving number should be added (after all previous list entries). The lottery system would need a different organization, and modeling the map of roads and cities needs another. In this book we examine a lot of different data structures. Each one has its strengths and weaknesses and is applicable to different kinds of real-world problems. It's important to understand how each one operates, whether it correctly models the behavior needed by a particular problem area, whether it will operate efficiently to perform the operations, and whether it can "scale" well. We say a data structure or algorithm scales well if it will perform as efficiently as possible as the amount of data grows.

Overview of Data Structures

As we've discussed, not every data structure models every type of problem. Or perhaps a better way to put it is that the structures model the problem awkwardly or inefficiently. You can generalize the data structures somewhat by looking at the common operations that you are likely to do across all of them. For example, to manage the requests for tickets, you need to

- **Add** a new phone number (for someone who wants one or more tickets)
- **Remove** a phone number (for someone who later decides they don't want tickets)
- **Find** a particular phone number (the next one to get a ticket by some method, or to look up one by its characteristics)
- **List** all the phone numbers (show all the phone numbers exactly once, that is, without repeats except, perhaps, for cases where multiple identical entries were made)

These four operations are needed for almost every data structure that manages a large collection of similar items. We call them **insertion**, **deletion**, **search**, and **traversal**.

Here's a list of the data structures covered in this book and some of their advantages and disadvantages with respect to the four operations. Table 1-1 shows a very high-level view of the structures; we look at them in much more detail in the following chapters. One aspect mentioned in this table is the data structure's **complexity**. In this context, we're referring to the structure's ability to be understood easily by programmers, not how quickly it can be manipulated by the computer.

Table 1-1 Comparison of Different Data Types

Data Structure	Advantages	Disadvantages
Array	Simple. Fast insertion as long as index is known. Fast traversal.	Slow search, size must be known at beginning. Slow to grow.
Ordered array	Same as array but quicker search for items if looking by sort key.	Same as array.
Stack	Simple. Fast for last-in, first-out (LIFO) insertion and deletion. Fast traversal.	Slow search for items other than last-in, first-out.
Queue	Moderately complex. Fast for first-in, first-out (FIFO) insertion and deletion. Fast traversal.	Slow search for items other than first-in, first-out.
Linked list	Moderately complex. Fast insertion and deletion for a known position in list. Easy to grow and shrink. Fast traversal.	Slow search.
Binary tree	Moderately complex. Quick search, insertion, and deletion when tree is balanced. Easy to grow and shrink. Fast traversal.	Deletion algorithm is complex, and balancing can be time-consuming.
Red-Black tree	Quick search, insertion, and deletion. Tree always balanced. Fast traversal.	Complex.
2-3-4 tree	Quick search, insertion, and deletion. Tree always balanced. Similar trees are good for disk storage. Easy to grow and shrink. Fast traversal.	Complex.
Quadtree	Quick search, insertion, and deletion by 2 dimensional coordinates. Easy to grow and shrink. Fast traversal.	Complex.
Hash table	Fast search. Fast insertion and quick deletion in most cases.	Complex. Can take more space than other data structures. Traversal may be slightly slower than others.
Heap	Moderately complex. Quick insertion and quick deletion of items in sorted order. Fast traversal.	Slow search for items other than minimum or maximum.
Graph	Fast insertion and unordered traversal of nodes and edges	Complex. Slow search (by label) and deletion. Traversal can be slow depending on path constraints. Can take more space than other data structures.

Overview of Algorithms

Algorithms are ways to implement an operation using a data structure or group of structures. A single algorithm can sometimes be applied to multiple data structures with each data structure needing some variations in the algorithm. For example, a depth first search algorithm applies to all the tree data structures, perhaps the graph, and maybe even stacks and queues (consider a stack as a tree with branching factor of 1). In most cases, however, algorithms are intimately tied to particular data structures and don't generalize easily to others. For example, the way to insert new items or search for the presence of an item is very specific to each data structure. We examine the algorithms for insertion, search, deletion, and traversal for all the data structures. That illustrates how much they vary by data structure and the complexity involved in those structures.

Another core algorithm is **sorting**, where a collection of items is put in a particular order. Ordering the items makes searching for items faster. There are many ways to perform sort operations, and we devote Chapter 3, "Simple Sorting," to this topic, and revisit the problem in Chapter 7, "Advanced Sorting."

Algorithms are often defined **recursively**, where part of the algorithm refers to executing the algorithm again on some subset of the data. This very important concept can simplify the definition of algorithms and make it very easy to prove the correctness of an implementation. We study that topic in more detail in Chapter 6, "Recursion."

Some Definitions

This section provides some definitions of key terms.

Database

We use the term **database** to refer to the complete collection of data that's being processed in a particular situation. Using the example of people interested in tickets, the database could contain the phone numbers, the names, the desired number of tickets, and the tickets awarded. This is a broader definition than what's meant by a relational database or object-oriented database.

Record

Records group related data and are the units into which a database is divided. They provide a format for storing information. In the ticket distribution example, a record could contain a person's name, a person's phone number, a desired number of tickets, and a number of awarded tickets. A record typically includes all the information about some entity, in a situation in which there are many such entities. A record might correspond to a user of a banking application, a car part in an auto supply inventory, or a stored video in a collection of videos.

Field

Records are usually divided into several **fields**. Each field holds a particular kind of data. In the ticket distribution example, the fields could be as shown in Figure 1-1.

Name	
Phone number	
Desired tickets	
Awarded tickets	

FIGURE 1-1 A record definition for ticket distribution

The fields are named and have values. Figure 1-1 shows an empty box representing the storage space for the value. In many systems, the type of value is restricted to a single or small range of data types, just like variables are in many programming languages. For example, the desired number of tickets could be restricted to only integers, or only non-negative integers. In object-oriented systems, objects often represent records, and each object's **attributes** are the fields of that record. The terminology can be different in the various programming languages. Object attributes might be called members, fields, or variables.

Key

When searching for records or sorting them, one of the fields is called the **key** (or **search key** or **sort key**). Search algorithms look for an exact match of the key value to some target value and return the record containing it. The program calling the search routine can then access all the fields in the record. For example, in the ticket distribution system, you might search for a record by a particular phone number and then look at the number of desired tickets in that record. Another kind of search could use a different key. For example, you could search for a record using the desired tickets as search key and look for people who want three tickets. Note in this case that you could define the search to return the first such record it finds or a collection of all records where the desired number of tickets is three.

Databases vs. Data Structures

The collection of records representing a database is going to require a data structure to implement it. Each record within the database may also be considered a data structure with its own data organization and algorithms. This decomposition of the data into smaller and smaller units goes on until you get to primitive data structures like integers, floating-point numbers, characters, and Boolean values. Not all data structures can be considered databases; they must support insertion, search, deletion, and traversal of records to implement a database.

Programming in Python

Python is a programming language that debuted in 1991. It embraces object-oriented programming and introduced syntax that made many common operations very concise and elegant. One of the first things that programmers new to Python notice is that certain whitespace is significant to the meaning of the program. That means that when you edit Python programs, you should use an editor that recognizes its syntax and helps you create the program as you intend it to work. Many editors do this, and even editors that don't recognize the syntax by filename extension or the first few lines of text can often be configured to use Python syntax for a particular file.

Interpreter

Python is an **interpreted language**, which means that even though there is a compiler, you can execute programs and individual expressions and statements by passing the text to an interpreter program. The compiler works by translating the source code of a program into **bytecode** that is more easily read by the machine and more efficient to process. Many Python programmers never have to think about the compiler because the Python interpreter runs it automatically, when appropriate.

Interpreted languages have the great benefit of allowing you to try out parts of your code using an interactive command-line interpreter. There are often multiple ways to start a Python interpreter, depending on how Python was installed on the computer. If you use an Integrated Development Environment (IDE) such as IDLE, which comes with most Python distributions, there is a window that runs the command-line interpreter. The method for starting the interpreter differs between IDEs. When IDLE is launched, it automatically starts the command-line interpreter and calls it the *Shell*.

On computers that don't have a Python IDE installed, you can still launch the Python interpreter from a command-line interface (sometimes called a terminal window, or shell, or console). In that command-line interface, type `python` and then press the Return or Enter key. It should display the version of Python you are using along with some other information, and then wait for you to type some expression in Python. After reading the expression, the interpreter decides if it's complete, and if it is, computes the value of the expression and prints it. The example in Listing 1-1 shows using the Python interpreter to compute some math results.

LISTING 1-1 Using the Python Interpreter to Do Math

```
$ python
Python 3.6.0 (default, Dec 23 2016, 13:19:00)
Type "help", "copyright", "credits" or "license" for more information.
>>> 2019 - 1991
28
>>> 2**32 - 1
4294967295
>>> 10**27 + 1
1000000000000000000000000001
```

```
>>> 10**27 + 1.001
1e+27
>>>
```

In Listing 1-1, we've colored the text that you type in *blue italics*. The first dollar sign ($) is the prompt from the command-line interpreter. The Python interpreter prints out the rest of the text. The Python we use in this book is version 3. If you see `Python 2…` on the first line, then you have an older version of the Python interpreter. Try running `python3` in the command-line interface to see if Python version 3 is already installed on the computer. If not, either upgrade the version of Python or find a different computer that has `python3`. The differences between Python 2 and 3 can be subtle and difficult to understand for new programmers, so it's important to get the right version. There are also differences between every minor release version of Python, for example, between versions 3.8 and 3.9. Check the online documentation at https://docs.python.org to find the changes.

The interpreter continues prompting for Python expressions, evaluating them, and printing their values until you ask it to stop. The Python interpreter prompts for expressions using >>>. If you want to terminate the interpreter and you're using an IDE, you typically quit the IDE application. For interpreters launched in a command-line interface, you can press Ctrl-D or sometimes Ctrl-C to exit the Python interpreter. In this book, we show all of the Python examples being launched from a command line, with a command that starts with `$ python3`.

In Listing 1-1, you can see that simple arithmetic expressions produce results like other programming languages. What might be less obvious is that small integers and very large integers (bigger than what fits in 32 or 64 bits of data) can be calculated and used just like smaller integers. For example, look at the result of the expression 10**27 + 1. Note that these big integers are not the same as floating-point numbers. When adding integers and floating-point numbers as in 10**27 + 1.0001, the big integer is converted to floating-point representation. Because floating-point numbers only have enough precision for a fixed number of decimal places, the result is rounded to 1e+27 or 1×10^{27}.

Whitespace syntax is important even when using the Python interpreter interactively. Nested expressions *use indentation instead of a visible character* to enclose the expressions that are evaluated conditionally. For example, Python `if` statements demarcate the `then`-expression and the `else`-expression by indentation. In C++ and JavaScript, you could write

```
if (x / 2 == 1) {do_two(x)}
else {do_other(x)}
```

The curly braces enclose the two expressions. In Python, you would write

```
if x / 2 == 1:
   do_two(x)
else:
   do_other(x)
```

You must indent the two procedure call lines for the interpreter to recognize their relation to the line before it. You must be consistent in the indentation, using the same tabs or spaces, on each indented line for the interpreter to know the nested expressions are at the same level. Think of the indentation changes as replacements for the open curly brace and the close curly brace. When the indent increases, it's a left brace. When it decreases, it is a right brace.

When you enter the preceding expression interactively, the Python interpreter prompts for additional lines with the ellipsis prompt (...). These prompts continue until you enter an empty line to signal the end of the top-level expression. The transcript looks like this, assuming that x is 3 and the `do_other()` procedure prints a message:

```
>>> if x / 2 == 1:
...     do_two(x)
... else:
...     do_other(x)
...
Processing other value
>>>
```

Note, if you've only used Python 2 before, the preceding result might surprise you, and you should read the details of the differences between the two versions at https://docs.python.org. To get integer division in Python 3, use the double slash (`//`) operator.

Python requires that the indentation of **logical lines** be the same if they are at the same level of nesting. Logical lines are complete statements or expressions. A logical line might span multiple lines of text, such as the previous `if` statement. The next logical line to be executed after the `if` statement's `then` or `else` clause should start at the same indentation as the `if` statement does. The deeper indentation indicates statements that are to be executed later (as in a function definition), conditionally (as in an `else` clause), repeatedly (as in a loop), or as parts of larger construct (as in a class definition). If you have long expressions that you would prefer to split across multiple lines, they either

- Need to be inside parentheses or one of the other bracketed expression types (lists, tuples, sets, or dictionaries), or
- Need to terminate with the backslash character (\) in all but the last line of the expression

Inside of parentheses/brackets, the indentation can be whatever you like because the closing parenthesis/bracket determines where the expression ends. When the logical line containing the expression ends, the next logical line should be at the same level of indentation as the one just finished. The following example shows some unusual indentation to illustrate the idea:

```
>>> x = 9
>>> if (x %
...             2 == 0):
...    if (x %
... 3 == 0):
...       'Divisible by 6'
...    else:
...       'Divisible by 2'
... else:
...    if (x %
...    3 == 0):
...       'Divisible by 3'
...    else:
...       'Not divisble by 2 or 3'
...
'Divisible by 3'
```

The tests of divisibility in the example occur within parentheses and are split across lines in an arbitrary way. Because the parentheses are balanced, the Python interpreter knows where the `if` test expressions end and doesn't complain about the odd indentation. The nested `if` statements, however, must have the same indentation to be recognized as being at equal levels within the conditional tests. The `else` clauses must be at the same indentation as the corresponding `if` statement for the interpreter to recognize their relationship. If the first `else` clause is omitted as in the following example,

```
>>> if (x %
...             2 == 0):
...    if (x %
... 3 == 0):
...       'Divisible by 6'
... else:
...    if (x %
...    3 == 0):
...       'Divisible by 3'
...    else:
...       'Not divisble by 2 or 3'
...
'Divisible by 3'
```

then the indentation makes clear that the first `else` clause now belongs to the `if (x % 2 == 0)` and not the nested `if (x % 3 == 0)`. If `x` is 4, then the statement would evaluate to `None` because the `else` clause was omitted. The mandatory indentation makes the structure clearer, and mixing in unconventional indentation makes the program very hard to read!

Whitespace inside of strings is important and is preserved. Simple strings are enclosed in single (') or double (") quote characters. They cannot span lines but may contain escaped whitespace such as newline (\n) or tab (\t) characters, e.g.,

```
>>> "Planck's constant:\n quantum of action:\t6.6e-34"
"Planck's constant:\n quantum of action:\t6.6e-34"
>>> print("Planck's constant:\n quantum of action:\t6.6e-34")
Planck's constant:
 quantum of action:     6.6e-34
```

The interpreter reads the double-quoted string from the input and shows it in *printed representation* form, essentially the same as the way it would be entered in source code with the backslashes used to escape the special whitespace. If that same double-quoted string is given to the `print` function, it prints the embedded whitespace in output form. To create long strings with many embedded newlines, you can enclose the string in triple quote characters (either single or double quotes).

```
>>> """Python
... enforces readability
... using structured
... indentation.
... """
'Python\nenforces readability\nusing structured\nindentation.\n'
```

Long, multiline strings are especially useful as documentation strings in function definitions.

You can add comments to the code by starting them with the pound symbol (#) and continuing to the end of the line. Multiline comments must each have their own pound symbol on the left. For example:

```
def within(x, lo, hi):   # Check if x is within the [lo, hi] range
   return lo <= x and x <= hi  # Include hi in the range
```

We've added some color highlights to the comments and reserved words used by Python like `def`, `return`, and `and`, to improve readability. We discuss the meaning of those terms shortly. Note that comments are visible in the source code files but not available in the runtime environment. The documentation strings mentioned previously are attached to objects in the code, like function definitions, and are available at runtime.

Dynamic Typing

The next most noticeable difference between Python and some other languages is that it uses **dynamic typing**. That means that the data types of variables are determined at runtime, not declared at compile time. In fact, Python doesn't require any variable declarations at all; simply assigning a value to a variable identifier creates the variable. You can change the value and type in later assignments. For example,

```
>>> x = 2
>>> x
2
>>> x = 2.71828
>>> x
2.71828
>>> x = 'two'
>>> x
'two'
```

The assignment statement itself doesn't return a value, so nothing is printed after each assignment statement (more precisely, Python's `None` value is returned, and the interpreter does not print anything when the typed expression evaluates to `None`). In Python 3.8, a new operator, `:=`, was introduced that allows assignment in an expression that returns a value. For example, evaluating

```
(x := 2) ** 2 + (y := 3) ** 2
```

sets x to be 2 and y to be 3 and returns 13 as a value.

The type of a variable's value can be determined by the `type()` function or tested using the `isinstance()` function.

Sequences

Arrays are different in Python than in other languages. The built-in data type that "looks like" an array is called a **list**. Python's `list` type is a hybrid of the arrays and linked lists found in other languages. As with variables, the elements of Python lists are dynamically typed, so they do not all have to be of the same type. The maximum number of elements does not need to be declared when creating a `list`, and `lists` can grow and shrink at runtime. What's quite different between Python `lists` and other linked list structures is that they can be indexed like arrays to retrieve any element at any position. There is no data type called array in the core of Python, but there is an `array` module that can be imported. The `array` module allows for construction of arrays of fixed-typed elements.

In this book, we make extensive use of the built-in `list` data type as if it were an array. This is for convenience of syntax, and because the underlying implementation of the `list` acts like arrays do in terms of indexed access to elements. Please note, however, that we do not use all of the features that the Python `list` type provides. The reason is that we want to show how fixed-type, fixed-length arrays behave in all computer languages. For new programmers, it's better to use the simpler syntax that comes with the built-in `list` for constructing arrays while learning how to manipulate their contents in an algorithm.

Python's built-in lists are indexed using 0-relative indexes. For example:

```
>>> a = [1, 'a', False]
>>> a
[1, 'a', False]
```

```
>>> len(a)
3
>>> a[2]
False
>>> a[2] = True
>>> a
[1, 'a', True]
>>>
```

The three-element list in the example contains an integer, a string, and a Boolean value. The square brackets are used either to create a new list (as in the first assignment to `a`) or to enclose the index of an existing list (as in `a[2]`). The built-in `len()` function is used to determine the current size of its argument. Individual values in the list can be changed using the assignment statement. Strings can be treated like lists or arrays of characters, except that unlike lists, strings are immutable, meaning that after a string is created, the characters of the string cannot be changed. In Python 3, strings always contain Unicode characters, which means there can be multiple bytes per character.

```
>>> s = 'π = 3.14159'
>>> s
'π = 3.14159'
>>> len(s)
11
>>> π = 3.14159
>>> π
3.14159
```

In the preceding example, the string, `s`, contains the Greek letter π, which is counted as one character by the `len()` function, whereas the Unicode character takes two bytes of space. Unicode characters can also be used in variable names in Python 3 as shown by using π as a variable name.

Python treats all the data types that can be indexed, such as lists, arrays, and strings, as **sequence data types**. The sequence data types can be **sliced** to form new sequences. Slicing means creating a subsequence of the data, which is equivalent to getting a substring for strings. Slices are specified by a start and end index, separated by a colon (:) character. Every element from the start index up to, but not including, the end index is copied into the new sequence. The start index defaults to 0, the beginning of the sequence, and the end index defaults to the length of the sequence. You can use negative numbers for both array and slice indexes. Negative indices count backwards from the end of the sequence; –1 means the last element, –2 means the second to last element, and so on. Here are some examples with a string:

```
>>> digits = '0123456789'
>>> digits[3]
'3'
```

```
>>> digits[-1]
'9'
>>> digits[-2]
'8'
>>> digits[3:6]
'345'
>>> digits[:-2]
'01234567'
```

Sequence data types can be **concatenated**, **multiplied**, **searched**, and **enumerated**. These operations sometimes require more function calls in other languages, but Python provides simple syntax to perform them. For example:

```
>>> [1, 2, 3] + ['a', 'b']
[1, 2, 3, 'a', 'b']
>>> '011' * 7
'011011011011011011011'
>>> '011' * 0
''
>>> 3 in [1, 2, 3]
True
>>> 'elm' in ['Elm', 'Asp', 'Oak']
False
```

The preceding example shows two lists **concatenated** with the plus (+) operator to form a longer list. Multiplying a string by an integer produces that many copies of the string, concatenated together. The `in` operator is a Boolean test that **searches** for an element in a sequence. It uses the == equality test to determine whether the element matches. These operations work with all sequence data types. This compact syntax hides some of the complexity of stepping through each sequence element and doing some operation such as equality testing or copying the value over to a new sequence.

Looping and Iteration

Frequently we want to implement algorithms that process each of the elements of a sequence in order. For that, Python has several ways to **iterate** and **enumerate** sequences. For example:

```
>>> total = 0
>>> for x in [5, 4, 3, 2, 1]:
...   total += x
...
>>> total
15
```

The `for` *variable* `in` *sequence* syntax is the basic loop construct (or **iteration**) in Python. The nested expression is evaluated once for each value in the sequence with the variable bound to the value. There is no need to explicitly manipulate an index variable that points to the current element of the sequence; that's handled by the Python interpreter. One common mistake when trying to enter this expression in the interactive interpreter is to forget the empty line after the nested expression.

```
>>> total = 0
>>> for x in [5, 4, 3, 2, 1]:
...   total += x
... total
  File "<stdin>", line 3
    total
        ^
SyntaxError: invalid syntax
```

The reason this is so common is that the empty line is needed only for the interactive interpreter; the same Python expressions written in a file would not report this as an error. The interactive interpreter, however, waits for the empty line to signal the end of the `for` loop and begin evaluation of that full expression. When the interpreter finds a new expression starting at the same indent level as the `for` loop, it is dealing with two consecutive expressions and does not allow it. The interpreter expects to read one expression, evaluate it, and print the value, before starting to determine where the next expression begins and ends.

In some circumstances, having an explicit index variable is important. In those cases, there are a couple of convenient ways to perform the work. For example:

```
>>> height = [5, 4, 7, 2, 3]
>>> weightedsum = 0
>>> for i in range(len(height)):
...   weightedsum += i * height[i]
...
>>> weightedsum
36
>>> for i, h in enumerate(height):
...   weightedsum += i * h
...
>>> weightedsum
72
```

The example calculates a weighted sum where we multiply each value in the `height` list by the index of that value. The `range()` function can be thought of as a function that produces a list of integers starting at 0 and going up to, but not equal to, its argument. By passing `len(height)` as an argument, `range()` produces the list `[0, 1, 2, 3, 4]`. In the

body of the first `for` loop, the `weightedsum` variable is incremented by the product of the index, `i`, and the value that `i` indexes in the `height` list. The second `for` loop repeats the same calculation using a slightly more concise form called **enumeration**. The `enumerate()` function can be thought of as taking a sequence as input and producing a sequence of pairs. The first element of each pair is an index, and the second is the corresponding value from its sequence argument. The second `for` loop has two variables separated by a comma, `i` and `h`, instead of just one as in the previous loops. On each iteration of the enumerate loop, `i` is bound to the index and `h` is bound to the corresponding value from `height`. Python makes the common pattern of looping over a sequence very easy to write, both with or without an index variable.

The `range()` and `enumerate()` functions actually create *iterators*, which are complex data types that get called in each loop iteration to get the next value of the sequence. It doesn't actually produce a list in memory for the full sequence. We discuss how iterators can be used to represent very long sequences without taking up much memory in Chapter 5, "Linked Lists."

Multivalued Assignment

The comma-separated list of variables can also be used in assignment statements to perform multiple assignments with a single equal sign (`=`) operator. This makes the most sense when all the values being assigned are closely related, like coordinates of a vector. To illustrate:

```
>>> x, y, z = 3, 4, 5
>>> y
4
>>> (x, y, z) = [7, 8, 9]
>>> y
8
```

The sequences on both sides of the assignment operator must have the same length; otherwise, an error occurs. The sequences can be of different types. The second assignment here uses a **tuple** on the left and a list on the right side. Python's tuple data type is also a sequence data type very similar to a list, with the distinction that its elements cannot be modified. In this case, the tuple on the left must be a tuple of variable names. The comma is the operator that creates the tuple of the `x`, `y`, and `z` variables. Each of them is bound to the corresponding value in the list on the right. Looking back at the enumerate loop in the previous example, the iterations of the loop are performing the equivalent of

```
>>> i, h = (0, 5)
>>> weightedsum += i * h
>>> i, h = (1, 4)
>>> weightedsum += i * h
>>> i, h = (2, 7)
```

```
>>> weightedsum += i * h
>>> i, h = (3, 2)
>>> weightedsum += i * h
>>> i, h = (4, 3)
>>> weightedsum += i * h
```

Python's multivalued assignments can be used to swap or rotate values. That can be a surprise to programmers of other languages where an explicit temporary variable must be used to hold one or more of the values. For example:

```
>>> left, middle, right = 'Elm', 'Asp', 'Oak'
>>> left, middle, right
('Elm', 'Asp', 'Oak')
>>> left, middle, right = middle, right, left
>>> left, middle, right
('Asp', 'Oak', 'Elm')
```

The Python interpreter evaluates all the expressions on the right of the equal sign, puts the results in a tuple, and then makes the assignments to the variables on the left side from the tuple. The tuple holding the results is something like a hidden temporary variable. The parentheses surrounding the tuples on both sides of the equal sign operator are optional.

There is another kind of assignment statement that looks like the multivalued assignment but is different. You use it to assign the same value to multiple variables. For example:

```
>>> left = middle = right = 'Gum'
>>> left, middle, right
('Gum', 'Gum', 'Gum')
```

Several variables are all assigned the same value. The overall assignment statement, however, still evaluates as `None`.

Importing Modules

Functions can return multiple values in the form of tuples, too. You can decide whether to store all these values as a single tuple, or in several variables, one for each of the tuple's elements, just like the multivalued assignment does. A good example of this is splitting pathname components. The `os` module is in Python's standard library and provides many tools to operate on the underlying operating system where Python is running. To access a module, you import it. After it is imported, you refer to its contents via the module name, followed by a period, and then the name of one of its functions or other definitions. For example:

```
>>> import math
>>> math.pi
3.141592653589793
```

```
>>> import os
>>> os.path.splitext('myfile.ext')
('myfile', '.ext')
>>> filename, extension = os.path.splitext('myfile.ext')
>>> extension
'.ext'
```

The import statement creates a **namespace** for all the objects it defines. That's important to avoid conflicts between a variable or function that you define and the one in the module. In the preceding example, the value for `pi` is stored in the `math` namespace. If you have a separate definition, say an approximation like `pi = 3.1`, your program can refer to each one without confusion. The `os` module has submodules, one of which is the `path` module. Nested submodules create nested namespaces with corresponding names separated by periods. The `os.path.splitext()` function splits a filename at the last period and returns the two parts as a tuple. The preceding example shows binding that result to two variables, `filename` and `extension`, and then shows the value of the extension. Occasionally, you may call a function that returns multiple values without needing to retain all of them. Some Python programmers like to use the simplest variable name, a single underscore (`_`), as a throwaway variable. For instance, if only the filename extension was needed, you could write

```
_, extension = os.path.splitext('myfile.ext')
```

Functions and Subroutines

Of course, the core of all programming languages is the ability to define functions and subroutines. In Python, they are defined with the `def` statement. The parameters the function accepts are provided in a list of variables. Parameters can be mandatory or optional; the optional ones must have a default value to use when they are missing. The `def` statement has the name of the function, the parameter list in parentheses, a colon, followed by a nested expression that is the body of the routine. If it is a function, the body should contain one or more `return` statements for the value to be returned. Subroutines in other programming languages do not return values. In Python, all functions and subroutines return something, but if no explicit return value is provided in the body of a `def` statement, then the value returned is `None`.

Here's an implementation of a weighted sum function. It takes a sequence of weights and a sequence of values. If the `weights` sequence is shorter than that of the `values`, it uses the `missing` value as the default weight.

```
>>> def weightedsum(values, weights, missing=0):
...     sum = 0
...     for i, val in enumerate(values):
...         sum += val * (weights[i] if i < len(weights) else missing)
...     return sum
...
```

```
>>> weightedsum([4, 9, 16, 25], [2, 2])
26
>>> weightedsum([4, 9, 16, 25], [2, 2], 1)
67
```

The preceding example also illustrates Python's **conditional expression**, which has the following form:

expression `if` *test_expression* `else` *expression*

First the *test_expression* is evaluated. If it is true, then the leftmost *expression* is evaluated and returned. Otherwise, the rightmost *expression* is evaluated and returned.

We refer to the variables in the function's `def` statement as its **parameters**. The **arguments** are the values that the caller submits to the function. Most of the time, arguments are matched to their parameters by their position in the argument list. Keyword arguments, in contrast, can be in any order but must follow any positional arguments in the call. A function call can specify a parameter's value by preceding the value with the parameter's name and an equal sign (`=`). Function definitions can also treat a sequence of parameters as a list. These are marked in the function definition using an asterisk (`*`) before the parameter name. As an example, Python's `print` function accepts any number of objects to print along with multiple keyword parameters that control what separators it uses in printing and where and how it produces the output. Its definition looks like this:

```
def print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False):
```

Inside the body of the `print` subroutine, the `objects` parameter holds the list of things to print. The printed output is the string representation of each object value separated by a space character and terminated by a newline, by default. Normally, the output string is sent to the standard output stream and can be buffered. If, instead, you wanted to send the output to the standard error stream separated by tab characters, you could write

```
>>> import sys
>>> print(print, 'is a', type(print), file=sys.stderr, sep='\t')
<built-in function print>     is a  <class 'builtin_function_or_method'>
```

In this example, the call to `print` has three positional parameters that all get bound to the `objects` list. Note that the `file` and `sep` arguments are not in the same position as their corresponding parameters.

List Comprehensions

Applying a calculation or function on each element of a list to produce a new list is such a useful concept that Python has a special syntax for it. The concept is called a **list comprehension**, and it is written as a little loop inside the brackets used to create lists:

[*expression* `for` *variable* `in` *sequence*]

The `for` *variable* `in` *sequence* part is the same syntax as for procedural loops. The *expression* at the beginning is the same as the loop body (except that it can't contain multiple statements; only a single expression is allowed). The Python interpreter goes through the *sequence* getting each element and binding the *variable* to that element. It evaluates the *expression* in the context of the bound *variable* to compute a value. That computed value is put in the output sequence in the same position as the element from the input *sequence*. Here are two equivalent ways to produce a list of squares of the values in another list:

```
>>> values = [7, 11, 13, 17, 19]
>>> squares = []
>>> for val in values:
...     squares.append(val * val)
...
>>> squares
[49, 121, 169, 289, 361]
>>> [ x * x for x in values ]
[49, 121, 169, 289, 361]
```

The first few lines in the example show a simple loop that appends the squares to an initially empty list. The last line in the example collapses all those lines (other than defining the values) into a single expression. The whitespace just inside the square brackets is optional, but many programmers put it in to make it clearer that this is a list comprehension instead of a list created by evaluating a series of comma-separated expressions.

The list comprehension is a very compact syntax for describing the incredibly useful concept of **mapping**. It hides all the implementation details of indices and loop exit conditions while still making the essence of the operation very clear. Mapping is used everywhere in programming. In many applications the same operation needs to be applied over huge amounts of data and collected into a similar structure to the input.

Let's look at a couple more comprehensions to get the idea. To get the cube of all the integers between 10 and 20, you can write

```
>>> [ x ** 3 for x in range(10, 21) ]
[1000, 1331, 1728, 2197, 2744, 3375, 4096, 4913, 5832, 6859, 8000]
```

To split hyphenated words, you could write

```
>>> [ w.split('-') for w in ['ape', 'ape-man', 'hand-me-down'] ]
[['ape'], ['ape', 'man'], ['hand', 'me', 'down']]
```

The loop can also have a filter condition. Only those elements satisfying the condition are put in the output sequence. The filter expression goes at the end, that is,

[*expression* `for` *variable* `in` *sequence* `if` *filter_expression*]

The filter expression should evaluate to `True` for those sequence elements you wish to keep in the output. For example, let's get the cube of all the integers between 10 and 20 that are not multiples of 3:

```
>>> [ x ** 3 for x in range(10, 21) if x % 3 != 0 ]
[1000, 1331, 2197, 2744, 4096, 4913, 6859, 8000]
```

And find all the characters in a string that are alphabetic:

```
>>> [ c for c in 'We, the People...' if c.isalpha() ]
['W', 'e', 't', 'h', 'e', 'P', 'e', 'o', 'p', 'l', 'e']
```

These compact forms make it easy to describe the core operation while hiding many of the looping details needed to implement basic operations.

Exceptions

Python allows programs to define what happens when particular exception conditions occur. These exceptions aren't just for errors; Python uses exceptions to handle things like the end of iteration for loops, keyboard interrupts, and timer expiration, which are all expected events. Your programs can define new kinds of exceptions and ways to handle them if they occur. Exception handling can be defined within `try except` statements. The basic form is

```
try:
    <statements>
except:
    <statements>
```

The set of *statements* after the `try:` are executed from top to bottom, and if and only if an exception occurs during that execution, the *statements* after the `except:` are executed. If no exception occurs, then the next logical line after the `try except` statement is executed.

Programs can define different handlers for different kinds of exceptions. For example:

```
try:
    long_running_function()
except KeyboardInterrupt:
    print('Keyboard interrupted long running function')
except IndexError:
    print('Index out of range during long running function')
```

Each of the `except` clauses can specify a type (class) of exception. Exceptions of that type (or a subclass of the declared class) trigger the corresponding statements to be executed. When no exception class is specified in an `except` clause, as in the basic form example, any type of exception triggers the execution of the clause.

When a program doesn't specify an exception handler for the type of exception that occurs, Python's default exception handlers print out information about the exception and where it occurred. There are also optional `else` and `finally` clauses for `try except` statements, but we don't use them in this book.

You can specify exceptions in your programs by using the `raise` statement, which expects a single argument, an exception object, describing the condition. The statement

```
raise Exception('No more tickets')
```

raises a general exception with a short description. If the program has defined an exception handler for the general `Exception` class at the time this exception is raised, then it is used in place of Python's default handler. Otherwise, Python prints a stack trace and exits the program if you're not running in the interactive interpreter. We look more at exception objects in the next section.

Object-Oriented Programming

Object-oriented programming developed as a way to organize code for data structures. The data managed in the structure is kept in an **object**. The object is also called an **instance** of a **class of objects**. For example, in distributing tickets for an event, you would want an object to hold the list of phone numbers for people desiring tickets. The class of the object could be a queue that makes it easier to implement a first-come, first-served distribution system (as described in Chapter 4, "Stacks and Queues"). Object classes define specific **methods** that implement operations on the object instances. For the ticketing system, there needs to be a method to add a new phone number for someone desiring tickets. There also needs to be methods to get the next phone number to call and to record how many tickets are assigned to each person (which might be stored in the same or different object). The methods are common to all objects in the class and operate on the data specific to each instance. If there were several events where tickets are being distributed, each event would need its own queue instance. The method for adding a phone number is common to each one of them and is **inherited** from the object class.

Python defines classes of objects with the `class` statement. These are organized in a hierarchy so that classes can inherit the definitions in other classes. The top of that hierarchy is Python's base class, `object`. The nested statements inside a `class` statement define the class; the `def` statements define methods and assignment statements define class attributes. The first parameter of each defined method should be `self`. The `self` parameter holds the object instance, allowing methods to call other instance methods and reference instance attributes. You reference an object instance's methods or attributes by appending the method or attribute name to the variable holding the object, separated by a period as in `object.method()`.

To define the constructor for object instances, there should be a `def __init__()` statement inside the class definition. Like the other methods, `__init__()` should accept `self` as its first parameter. The `__init__()` method takes the empty instance and performs any

needed initialization like creating instance variables and setting their values (and doesn't need to return `self` like constructors do in other languages). The `__init__()` method is unusual in that you will rarely see it called explicitly in a program (that is, in the form `variable.__init()`). Python has several special methods and other constructs that all have names beginning and ending with double underscores. We point out a few of them as we use them in examples in the text.

The short example in Listing 1-2 illustrates the basic elements of object-oriented programming in Python, along with some of its ability to handle fancy math concepts.

LISTING 1-2 Object-Oriented Program Example, Object_Oriented_Client.py

```
class Power(object):
    """A class that computes a specific power of other numbers.
    In other words, it raises numbers by a constant exponent.
    """

    default_exponent = 2

    def __init__(self, exponent=default_exponent):
        self.exponent = exponent

    def of(self, x):
        return x ** self.exponent

class RealPower(Power):  # A subclass of Power for real numbers

    def of(self, x):
        if isinstance(self.exponent, int) or x >= 0:
            return x ** self.exponent
        raise ValueError(
            'Fractional powers of negative numbers are imaginary')

print('Power:', Power)
print('Power.default_exponent:', Power.default_exponent)
square = Power()
root = Power(0.5)
print('square: ', square)
print('square.of(3) =', square.of(3))
print('root.of(3) =', root.of(3))
print('root.of(-3) =', root.of(-3))
real_root = RealPower(0.5)
print('real_root.of(3) =', real_root.of(3))
print('real_root.of(-3) =', real_root.of(-3))
print('Done.')
```

Listing 1-2 shows a file with two class definitions, three object instances, and some print statements to show how the objects behave. The purpose of the first class, `Power`, is to make objects that can be used to raise numbers to various exponents. That is explained in the optional document string that follows the `class` definition. Note that this is not a Python comment, which would have to start with a pound symbol (`#`) on the left.

Each object instance is created with its own exponent, so you can create `Power` objects to perform different power functions. The `Power` class has one class attribute, `default_exponent`. This is used by the constructor for the class to define what exponent to use if none is provided when an instance is created.

The constructor for the `Power` class simply stores the desired exponent as an **instance attribute**. This is somewhat subtle because there are three distinct kinds of storage: **class attributes**, **instance attributes**, and **local variables** in the methods. The `default_exponent` attribute is defined by the assignment in the top level of the `class` statement, so it is a **class attribute** and is shared among all instances. The `exponent` parameter in the `__init__()` method of `Power` is a **local variable** available only during the evaluation of the constructor. It has a default value that is supplied from the class attribute. When `__init__()` assigns a value to `self.exponent`, it creates an **instance attribute** for `self`. The instance attribute is unique to the object instance being created and is not shared with other objects of the class.

The `Power` class has one method called `of` that returns the result of raising a number to the exponent defined at instance creation. We use it after creating two instances, `square` and `root`, that can be called using `square.of(3)`, for example. To create the first instance, the program calls `Power(2)` and binds the result to the variable `square`. This behavior might be somewhat unexpected because there is no mention of `__init__()`. This is an example of Python's use of reserved names like `__init__()` to fill special roles such as object constructors. When a class like `Power` is referenced using a function call syntax—the name followed by a parenthesized list of arguments—Python builds a new instance of the class and then invokes the `__init__()` method on it. The program in Listing 1-2 makes two calls to `Power()` to make one object that produces squares and one that produces square roots.

Underscores in Python names mean that the item being defined is either special to the interpreter or to be treated as private, visible only in the block where it's defined. Python uses several special names like `__init__` for customizing the behavior of objects and programs. The special names start and end with double underscores and enable the use of tools like using the class name as constructor call.

In the example of Listing 1-2, both the `default_exponent` class attribute and the `exponent` instance attribute are public because their names don't start with an underscore. If the name were changed to begin with an underscore, they are expected to be accessed only by the class and its methods and not to be accessed outside of the class or object. The restriction on access, however, is only a convention; the Python interpreter does not enforce the privacy of any attributes regardless of their name, nor does it provide other mechanisms to enforce privacy. Python does have a mechanism to mangle the names of class and object attributes that begin with a double underscore and end with at most one

underscore. The name mangling is designed to make class-specific attributes that are not shared with subclasses, but they are still publicly accessible through the mangled names.

You can run the program by giving the filename as an argument to the Python interpreter program as follows (the colored text in *blue italics* is the text that you type):

```
$ python3 Object_Oriented_Client.py
Power: <class '__main__.Power'>
Power.default_exponent: 2
square:  <__main__.Power object at 0x10715ad50>
square.of(3) = 9
root.of(3) = 1.7320508075688772
root.of(-3) = (1.0605752387249068e-16+1.7320508075688772j)
real_root.of(3) = 1.7320508075688772
Traceback (most recent call last):
  File "01_code/Object_Oriented_Client.py", line 32, in <module>
    print('real_root.of(-3) =', real_root.of(-3))
  File "01_code/Object_Oriented_Client.py", line 20, in of
    'Fractional powers of negative numbers are imaginary')
ValueError: Fractional powers of negative numbers are imaginary
```

The transcript shows the output of the `print` statements. `Power` is a class, and `square` is an object instance of that class. The `square` instance is created by the `call` to `Power()` with no arguments provided, so the exponent defaults to 2. To create a similar way to compute square roots, the object created by `Power(0.5)` is assigned to `root`. The square of 3 printed by the program is the expected 9, and the square root of 3 is the expected 1.732.

The next `print` statement is for the square root of –3, which may be a less expected result. It returns a **complex number** where `j` stands for the square root of –1. While that may be interesting for some engineering applications, you might want a different behavior for other programs. The `RealPower` class is a subclass of the `Power` class that raises an exception when raising negative numbers to fractional powers.

In the `class` statement for `RealPower`, it is defined to inherit from the `Power` class. That means that `RealPower` will have the same `default_exponent` attribute and `__init__` constructor as `Power` does. In the class definition for `RealPower`, it replaces the `of` method with a new one that tests the values of the exponent and the numeric argument. If they fall in the category that produces imaginary numbers, it raises a `ValueError` exception.

The transcript shows how the Python interpreter handles exceptions. It prints a traceback showing the function and method calls that were being evaluated when the exception occurred. The traceback includes the line numbers and copies of the lines from the input file. After encountering the exception, the interpreter prints the traceback, quits, and does not evaluate the final `print` statement in the file.

Summary

- Data can be arranged in the computer in different ways and using various storage media. The data is organized and interpreted to represent something.
- Algorithms are the procedures used to manipulate data.
- By coupling good data organization with appropriate algorithms, data structures provide the fundamental building blocks of all programs.
- Examples of data structures are stacks, lists, queues, trees, and graphs.
- Data structures are often compared by how efficiently they perform common operations.
- A database is a collection of many similar records, each of which describes some entity.
- The records of a database are composed of fields, each of which has a name and a value.
- A key field is used to search for and sort records.
- Data structures that act like databases support four core operations: insertion, search, deletion, and traversal.
- Data structures are implemented as classes using object-oriented programming.
- Python has rich support for object-oriented programming with classes to implement data structures.

Questions

These questions are intended as a self-test for readers. The answers can be found in Appendix C.

1. Data structures that allow programs to _____ a record, _____ for a record, _____ a record, and _____ all the records are considered to be databases in this book.
2. Data structures are

   a. composed of names and values.

   b. built with fields that have methods attached to them.

   c. implemented with object classes.

   d. records that don't change in a database.
3. How can you tell if a `def` statement in Python defines a function or class method?
4. What algorithms can help make searching more efficient?
5. What are constructors used for? What special name does Python use for them?

6. What are some of the reasons for choosing one data structure over another?
7. What is a key field used for?
8. Good data organization can help with the speed of an algorithm, but what other benefits does it have?
9. For what purpose was object-oriented programming developed?
10. Which of the following are data structures used in programming?
    a. traceback
    b. heap
    c. list comprehension
    d. hash table
    e. recipe
    f. slices
    g. binary tree

Experiments

Try the following experiments:

1-A Write a Python list comprehension that returns the individual characters of a string that are not whitespace characters. Apply it to the string `"4 and 20 blackbirds.\n"`

1-B Take a deck of playing cards, pull out the 13 spade cards, set aside the rest, and shuffle the spade cards. Devise an algorithm to sort them by number under the constraints:

    a. All the cards must be held in one hand. This is the "first" hand.
    b. Initially, the shuffled cards are all stacked with faces in one direction so that only one card is visible.
    c. Initially, all the cards are held between the thumb and forefinger of the first hand.
    d. The visible card in the stack can be pulled out using the other hand and placed in between any of the fingers of the first hand. It can only be placed at the front or the back of the cards in the stack of cards between those fingers.
    e. The other hand can hold one card at a time and must place it somewhere in the first hand before picking out another visible card from one of the stacks.
    f. The algorithm is done when all the cards are in sorted order in one stack in the hand.

Compare the efficiency of your algorithm with that of classmates or friends.

CHAPTER 2
Arrays

IN THIS CHAPTER

- The Array Visualization Tool
- Using Python Lists to Implement the Array Class
- The OrderedArray Visualization Tool
- Binary Search
- Python Code for an Ordered Array Class
- Logarithms
- Storing Objects
- Big O Notation
- Why Not Use Arrays for Everything?

Arrays are the most commonly used data structure for many reasons. They are straightforward to understand and match closely the underlying computer hardware. Almost all CPUs make it very fast to access data at known offsets from a base address. Almost every programming language supports them as part of the core data structures. We study them first for their simplicity and because many of the more complex data structures are built using them.

First, we look at the basics of how data is inserted, searched, and deleted from arrays. Then, we look at how we can improve it by examining a special kind of array, the orderedarray, in which the data is stored in ascending (or descending) key order. This arrangement makes possible a fast way of searching for a data item: the binary search.

To improve a data structure's performance requires a way of measuring performance beyond just running it on sample data. Looking at examples of how it handles particular kinds of data makes it easier to understand the operations. We also take the first step to generalize the performance measure by looking at linear and binary searches, and introducing Big O notation, the most widely used measure of algorithm efficiency.

Suppose you're coaching kids-league soccer, and you want to keep track of which players are present at the practice field. What you need is an attendance-monitoring program for your computer—a program that maintains a database of the players who have shown up for practice. You can use a

simple data structure to hold this data. There are several actions you would like to be able to perform:

- Insert a player into the data structure when the player arrives at the field.
- Check to see whether a particular player is present, by searching for the player's number in the structure.
- Delete a player from the data structure when that player leaves.
- List all the players present.

These four operations—insertion, searching, deletion, and enumeration (traversal)—are the fundamental ones in most of the data storage structures described in this book.

The Array Visualization Tool

We often begin the discussion of a particular data structure by demonstrating it with a visualization tool—a program that animates the operations. This approach gives you a feeling for what the structure and its algorithms do, before we launch into a detailed explanation and demonstrate sample code. The visualization tool called Array shows how an array can be used to implement insertion, searching, and deletion.

Now start up the Array Visualization tool, as described in Appendix A, "Running the Visualizations." There are several ways to do this, as described in the appendix. You can start it up separately or in conjunction with all the visualizations. If you've downloaded Python and the source code to your computer, you can launch it from the command line using

```
python3 Array.py
```

Figure 2-1 shows the initial array with 10 elements, 9 of which have data items in them. You can think of these items as representing your players. Imagine that each player has been issued a team shirt with the player's number on the back. That's really helpful because you have just met most of these people and haven't learned all their names yet. To make things visually interesting, the shirts come in a variety of colors. You can see each player's number and shirt color in the array. The height of the colored rectangle is proportional to the number.

This visualization demonstrates the four fundamental operations mentioned earlier:

- The Insert button inserts a new data item.
- The Search button searches for a specified data item.
- The Delete button deletes a specified data item.
- The Traverse button lists all the items in the array.

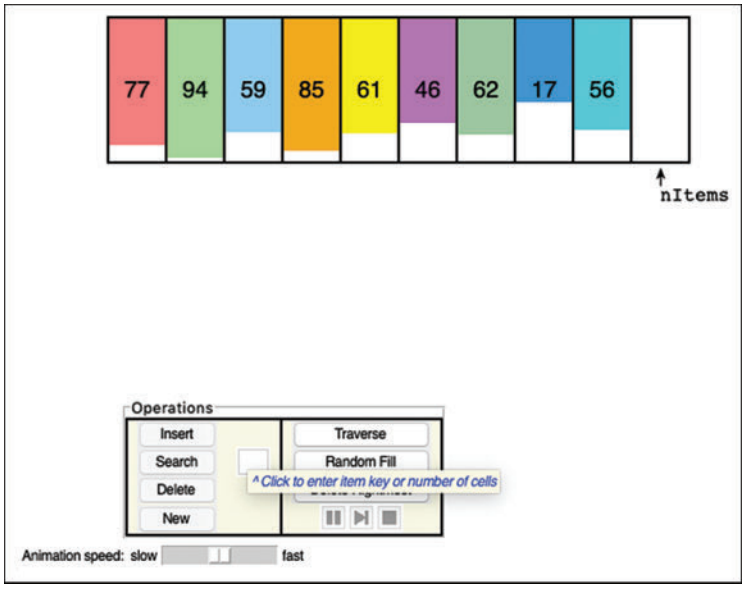


FIGURE 2-1 The Array Visualization tool

The buttons for the first three of those operations are on the left, grayed out and disabled. The reason is that you haven't entered a number in the small box to their right. The hint below the box suggests that when there's a number to search, insert, or delete, the buttons will be enabled.

The Traverse button is on the right and enabled. That's because it doesn't require an argument to start traversing. We explore that and the other buttons on the right shortly.

On the left, there is also a button labeled New. It's used to create a new array of a given size. The size is taken from the text entry box like the other arguments. (The initial hint shown in Figure 2-1 also mentioned that you can enter the number of cells here.) Arrays must be created with a known size because they place the contents in adjacent memory cells, and that memory must be allocated for exclusive use by the array. We look at each of these operations in turn.

Searching

Imagine that you just arrived at the playing field to start coaching, your assistant hands you the computer that's tracking attendance, and a player's parent asks if the goalies can start their special drills. You know that players 2, 5, and 17 are the ones who play as goalies, but are they all here? Although answering this question is trivial for a real coach with paper and pencil, let's look at the details needed to do it with the computer.

You want to search to see whether all the players are present. The array operations let you search for one item at a time. In the visualization tool, you can select the text entry box near the Search button, the hint disappears, and you enter the number 2. The button

becomes enabled, and you can select it to start the search. The tool animates the search process, which starts off looking like Figure 2-2.

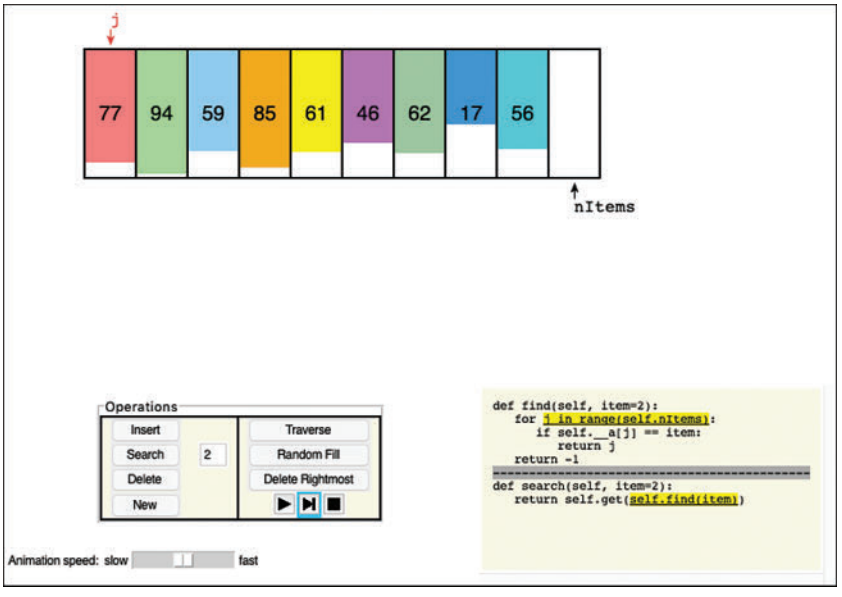


FIGURE 2-2 Starting a search for player 2

The visualization tool now shows several new things. A box in the lower right shows a program being executed. Next to the array drawing, an arrow labeled with a `j` points to the first cell. The `j` advances to each cell in turn, checking to see whether it holds the number 2. When it reaches where the `nItems` arrow points on the right, all the cells have been searched. The `j` arrow disappears, and a message appears at the bottom: `Value 2 not found`.

This process mimics what humans would do—scan the list, perhaps with a finger dragged along the numbers, confirming whether any of them match the number being sought. The visualization tool shows how the computer represents those same activities. The tool allows you to pause the animation of the process if you want to look at the details. The three buttons at the bottom right of the operations area, ▶⏭■, control the animation. The leftmost one plays or pauses the animation. The middle button plays until the next code step, and the square shape on the right stops the operation. At the bottom, you can slide the animation speed control left or right to slow down or speed up the animation.

Try searching for the value 17 (or some other number present in the array). The visualization tool starts with the `j` arrow pointing at the leftmost array cell. After checking the value there, it moves to the next cell to the right. That process repeats seven times in the array shown in Figure 2-2, and then the value is circled. After a few more changes in the

code box, the message `Value 17 found` appears at the bottom (we return to the code in a moment).

What's important to notice here is that the search went through seven steps before finding the value. When you searched for 2, it went through all nine items before stopping. Let's call the number of items N (which is just a shorter version of the `nItems` shown in the visualization). When you search for an item that is in the array, it could take 1 to N steps before finding it. If there's an equal chance of being asked to search for each item, the average number of search steps is (1 + 2 + 3 + … + N–1 + N) / N which works out to (N + 1) / 2. Unsuccessful searches take N steps every time.

There's another simple but still important thing to notice. What if there were duplicate values in the array? That's not supposed to happen on the team, of course. Every player should have a distinct number. If one player lent one of their shirts to a teammate who forgot theirs, however, it would not be surprising to see the same number twice.

The search strategy must know whether to expect multiple copies of a number to exist. The visualization tool stops when it finds the first matching number. If multiple copies are allowed, *and* it's important to find all of them, then the search couldn't stop after finding the first one. It would have to go through all N items and identify how many matched. In that case, both successful and unsuccessful searches would take N steps.

Insertion

We didn't find player 2 when asked before, but now that player has just arrived. You need to record that player 2 is at practice, so it's time to insert them in the array. Type `2` in the text entry box and select Insert. A new colored rectangle with 2 in it appears at the bottom and moves into position at the empty cell indicated by the `nItems` pointer. When it's in position, the `nItems` pointer moves to the right by one. It may now point beyond the last cell of the array.

The animation of the arrival of the new item takes a little time, but in terms of what the computer has to do, only two steps were needed: writing the new value in the array at the position indicated by `nItems` and then incrementing `nItems` by 1. It doesn't matter if there were two, three, or a hundred items already in the array; inserting the value always takes two steps. That makes the insertion operation quite different from the search operation and almost always faster.

More precisely, the number of steps for insertion doesn't depend on how many items are in the array as long as it is not full. If all the cells are filled, putting a value outside of the array is an error. The visualization tool won't let that happen and will produce an error message if you try. (In the history of programming, however, quite a few programmers did not put that check in in the code, leading to buffer overflows and security problems.)

The visualization tool lets you insert duplicate values (if there are available cells). It's up to you to avoid them, possibly by using the Search operation, if you don't want to allow it.

Deletion

Player 17 has to leave (he wants to start on the homework assignment due tomorrow). To delete an item in the array, you must first find it. After you type in the number of the item to be deleted, a process like the search operation begins in the visualization tool. A `j` arrow appears starting at the leftmost cell and steps to the right as it checks values in the cells. When it finds and circles the value as shown in the top left of Figure 2-3, however, it does something different.

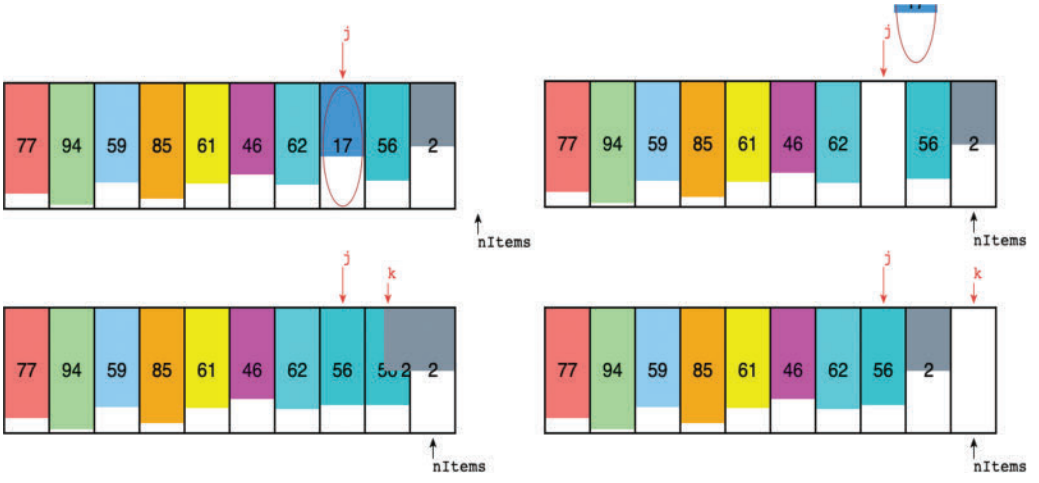


FIGURE 2-3 Deleting an item

The visualization tool starts by reducing `nItems` by 1, moving it to point at the last item in the array. That behavior might seem odd at first, but the reason will become clear when we look at the code. The next thing it does is move the deleted value out of the array, as shown in the top right of Figure 2-3. That doesn't actually happen in the computer, but the empty space helps distinguish that cell visually during what happens next.

A new arrow labeled `k` appears pointing to the same cell as `j`. Each of the items to the right of the deleted item is copied into the cell to its left, and `k` is advanced. So, for example, item 56 is copied into cell `j`, and then item 2 is copied into where 56 was, as shown in the bottom left of the figure. When they are all moved, the `nItems` arrow (and `k`) points at the empty cell just after the last filled cell, as shown at the bottom right. That makes the array ready to accept the next item to insert in just two steps. (The visualization tool clears the last cell after copying its contents to the cell to its left. This behavior is not strictly necessary but helps the visualization.)

Implicit in the deletion algorithm is the assumption that holes are not allowed in the array. A **hole** is one or more empty cells that have filled cells above them (at higher index numbers). If holes are allowed, the algorithms for all the operations become more complicated because they must check to see whether a cell is empty before doing something with its contents. Also, the algorithms become less efficient because they waste time looking

at unoccupied cells. For these reasons, occupied cells must be arranged contiguously: no holes allowed.

Try deleting another item while carefully watching the changes to the different arrows. You can slow down and pause the animation to see the details.

How many steps does each deletion take? Well, the `j` arrow had to move over a certain number of times to find the item being deleted; let's call that J. Then you had to shift the items to the right of `j`. There were N – J of those items. In total there were J + N – J steps, or simply N steps. The steps were different in character: checking values versus copying values (we consider the difference between making value comparisons and shifting items in memory later).

Traversal

Arrays are simple to traverse. The data is already in a linear order as specified by the index to the array elements. The index is set to 0, and if it's less than the current number of items in the array, then the array item at index 0 is processed. In the Array visualization tool, the item is copied to an output box, which is similar to printing it. The index is incremented until it equals the current number of items in the array, at which point the traversal is complete. Each item with an index less than `nItems` is processed exactly once. It's very easy to traverse the array in reverse order by decrementing the index too.

The Duplicates Issue

When you design a data storage structure, you need to decide whether items with duplicate keys will be allowed. If you're working with a personnel file and the key is an employee number, duplicates don't make much sense; there's no point in assigning the same number to two employees. On the other hand, a list of contacts might have several entries of people who have the same family name. Two entries might even have the same given and family names. Assuming the names are the key for looking up the contact, duplicate keys should be allowed in a simple contacts list. Another example would be a data structure designed to keep track of the food items in a pantry. There are likely to be several identical items, such as cans of beans or bottles of milk. The key could be the name of the item or the label code on its package. In this context, it's likely the program will not only want to search for the presence for an item but also count how many identical items are in the store.

If you're writing a data storage program in which duplicates are not allowed, you may need to guard against human error during an insertion by checking all the data items in the array to ensure that not one of them already has the same key value as the item being inserted. This check reduces the efficiency, however, by increasing the number of steps required for an insertion from one to N. For this reason, the visualization tool does not perform this check.

Searching with Duplicates

Allowing duplicates complicates the search algorithm, as we noted. Even if the search finds a match, it must continue looking for possible additional matches until the last occupied cell. At least, this is one approach; you could also stop after the first match and

perform subsequent searches after that. How you proceed depends on whether the question is "Find me everyone with the family name of Smith," "Find me someone with the family name of Smith," or the similar question "Find how many entries have the family name Smith."

Finding all items matching a search key is an **exhaustive search**. Exhaustive searches require N steps because the algorithm must go all the way to the last occupied cell, regardless of what is being sought.

Insertion with Duplicates

Insertion is the same with duplicates allowed as when they're not: a single step inserts the new item. Remember, however, that if duplicates are prohibited, and there's a possibility the user will attempt to input the same key twice, the algorithm must check every existing item before doing an insertion.

Deletion with Duplicates

Deletion may be more complicated when duplicates are allowed, depending on exactly how "deletion" is defined. If it means to delete only the first item with a specified value, then, on the average, only N/2 comparisons and N/2 moves are necessary. This is the same as when no duplicates are allowed. This would be the desired way to handle deleting an item such as a can of beans from a kitchen pantry when it gets used. Any items with duplicate keys remain in the pantry.

If, however, deletion means to delete *every* item with a specified key value, the same operation may require multiple deletions. Such an operation requires checking N cells and (probably) moving more than N/2 cells. The average depends on how the duplicates are distributed throughout the array.

Traversal with Duplicates

Traversal means processing each of the stored items exactly once. If there are duplicates, then each duplicate item is processed once. That means that the algorithm doesn't change if duplicates are present. Processing all the *unique* keys in the data store exactly once is a different operation.

Table 2-1 shows the average number of comparisons and moves for the four operations, first where no duplicates are allowed and then where they are allowed. N is the number of items in the array. Inserting a new item counts as one move.

Table 2-1 Duplicates OK Versus No Duplicates

	No Duplicates	Duplicates OK
Search	N/2 comparisons	N comparisons
Insertion	No comparisons, one move	No comparisons, one move
Deletion	N/2 comparisons, N/2 moves	N comparisons, more than N/2 moves
Traversal	N processing steps	N processing steps

The difference between N and N/2 is not usually considered very significant, except when you're fine-tuning a program. Of more importance, as we discuss toward the end of this chapter, is whether an operation takes one step, N steps, or N^2 steps, which would be the case if you wanted to enumerate all the pairs of items stored in a list. When there are only a handful of items, the differences are small, but as N gets bigger, the differences can become huge.

Not Too Swift

One of the significant things to notice when you're using the Array Visualization tool is the slow and methodical nature of the algorithms. Apart from insertion, the algorithms involve stepping through some or all of the cells in the array performing comparisons, moves, or other operations. Different data structures offer much faster but slightly more complex algorithms. We examine one, the search on an ordered array, later in this chapter, and others throughout this book.

Deleting the Rightmost Item

Deletion is the slowest of the four core operations in an array. If you don't care what item is deleted, however, it's easy to delete the last (rightmost) item in the array. All that task requires is reducing the count of the number of items, `nItems`, by one. It doesn't matter whether duplicates are allowed or not; deleting the last item doesn't affect whether duplicates are present. The Array Visualization tool provides a Delete Rightmost operation to see the effect of this fast—one "move" or assignment no comparison—operation.

Using Python Lists to Implement the Array Class

The preceding section showed the primary algorithms used for arrays. Now let's look at how to write a Python class called `Array` that implements the array abstraction and its algorithms. But first we want to cover a few of the fundamentals of arrays in Python.

As mentioned in Chapter 1, "Overview," Python has a built-in data structure called `list` that has many of the characteristics of arrays in other languages. In the first few chapters of this book, we stick with the simple, built-in Python constructs for lists and use them as if they were arrays, while avoiding the use of more advanced features of Python lists that might obscure the details of what's really happening in the code. In Chapter 5, "Linked Lists," we introduce linked lists and describe the differences. For those programmers who started with Python lists, it may seem odd to initialize the size of the `Array`'s list at the beginning, but this is a necessary step for true arrays in all programming languages. The memory to hold all the array elements must be allocated at the beginning so that the sequence of elements can be stored in a contiguous range of memory and any array element can be accessed in any order.

Creating an Array

As we noted in Chapter 1, Python lists are constructed by enclosing either a list of values or by using a list comprehension (loop) in square brackets. The list comprehension really builds a new `list` based on an existing list or sequence. To allocate lists with large

numbers of values, you use either an iterator like `range()` inside a list comprehension or the multiplication operator. Here are some examples:

```
integerArray = [1, 1, 2, 3, 5]          # A list of 5 integers
charArray = ['a' for j in range(1000)]  # 1,000 letter 'a' characters
booleArray = [False] * 32768            # 32,768 binary False values
```

Each of these assignment statements creates a list with specific initial values. Python is dynamically typed, and the items of a list do not all have to be of the same type. This is one of the core differences between Python lists and the arrays in statically typed languages, where all items must be of the same type. Knowing the type of the items means that the amount of memory needed to represent each one is known, and the memory for the entire array can be allocated. In the case of `charArray` in the preceding example, Python runs a small loop 1,000 times to create the list. Although all three sample lists start with items of the same type, they could later be changed to hold values of any type, and their names would no longer be accurate descriptions of their contents.

Data structures are typically created empty, and then later insertions, updates, and deletions determine the exact contents. Primitive arrays are allocated with a given maximum size, but their contents could be anything initially. When you're using an array, some other variables must track which of the array elements have been initialized properly. This is typically managed by an integer that stores the current number of initialized elements.

In this book, we refer to the storage location in the array as an **element** or as a **cell**, and the value that is stored inside it as an **item**. In Python, you could write the initialization of an array like this:

```
maxSize = 10000
myArray = [None] * maxSize
myArraySize = 0
```

This code allocates a list of 10,000 elements each initialized to the special `None` value. The `myArraySize` variable is intended to hold the current number of inserted items, which is 0 at first. You might think that Python's built-in `len()` function would be useful to determine the current size, but this is where the implementation of an array using a Python list breaks down. The `len()` function returns the allocated size, not how many values have been inserted in `myArray`. In other words, `len(myArray)` would always be 10,000 (if you only change individual element values). We will track the number of items in our data structures using other variables such as `nItems` or `myArraySize`, in the same manner that must be done with other programming languages.

Accessing List Elements

Elements of a list are accessed using an **integer index** in square brackets. This is similar to how other languages work:

```
temp = myArray[3]   # get contents of fourth element of array
myArray[7] = 66     # insert 66 into the eighth cell
```

Remember that in Python—as in Java, C, and C++—the first element is numbered 0, so the indices in an array of 10 elements run from 0 to 9. What's different in Python is that you can use negative indices to specify the count from the end of the list. You can think of that as if the current size of the array were added to the negative index. In the previous example, you could get the last item by writing `myArray[maxSize - 1]` or, more simply, `myArray[-1]`.

Python also supports **slicing** of lists, but that seemingly simple operation hides many details that are important to the understanding of an `Array` class's behavior. As a result, we don't use slices in showing how to implement an Array class using lists.

Initialization

All the methods we've explored for creating lists in Python involve specifying the initial value for the elements. In other languages, it's easy to create or **allocate** an array without specifying any initial value. As long as the array elements have a known size and there is a known quantity of them, the memory needed to hold the whole array can be allocated. Because computers and their operating systems reuse memory that was released by other programs, the element values in a newly allocated array could be anything. Programmers must be careful not to write programs that use the array values before setting them to some desired value because the uninitialized values can cause errors and other unwanted behavior. Initializing Python list values to `None` or some other known constant avoids that problem.

An Array Class Example

Let's look at some sample programs that show how a list can be used. We start with a basic, object-oriented implementation of an `Array` class that uses a Python list as its underlying storage. As we experiment with it, we will make changes to improve its performance and add features. Listing 2-1 shows the class definition, called `Array`. It's stored in a file called `BadArray.py`.

LISTING 2-1 The `BadArray.py` Module

```
# Implement an Array data structure as a simplified type of list.

class Array(object):

   def __init__(self, initialSize):          # Constructor
      self.__a = [None] * initialSize        # The array stored as a list
      self.nItems = 0                        # No items in array initially

   def insert(self, item):                   # Insert item at end
      self.__a[self.__nItems] = item         # Item goes at current end
      self.__nItems += 1                     # Increment number of items
```

```
   def search(self, item):
      for j in range(self.nItems):           # Search among current
         if self.__a[j] == item:             # If found,
            return self.__a[j]               # then return item

      return None                            # Not found -> None

   def delete(self, item):                   # Delete first occurrence
      for j in range(self.nItems):           # of an item
         if self.__a[j] == item:             # Found item
            for k in range(j, self.nItems):  # Move items from
               self.__a[k] = self.__a[k+1]   # right over 1
            self.nItems -= 1                 # One fewer in array now
            return True                      # Return success flag

      return False     # Made it here, so couldn't find the item

   def traverse(self, function=print):       # Traverse all items
      for j in range(self.nItems):           # and apply a function
         function(self.__a[j])
```

The `Array` class has a constructor that initializes a fixed length list to hold the array of items. The array items are stored in a private instance attribute, `__a`, and the number of items stored in the array is kept in the public instance attribute, `nItems`. The four methods define the four core operations.

Before we look at the implementation details, let's use a program to test each operation. A separate file, `BadArrayClient.py`, shown in Listing 2-2 uses the `Array` class in the `BadArray.py` module. This program imports the class definition, creates an `Array` called `arr` with a `maxSize` of 10, inserts 10 data items (integers, strings, and floating-point numbers) in it, displays the contents by traversing the `Array`, searches for a couple of items in it, tries to remove the items with values 0 and 17, and then displays the remaining items.

LISTING 2-2 The `BadArrayClient.py` Program

```
import BadArray
maxSize = 10                               # Max size of the Array
arr = BadArray.Array(maxSize)              # Create an Array object

arr.insert(77)                             # Insert 10 items
arr.insert(99)
arr.insert("foo")
arr.insert("bar")
arr.insert(44)
arr.insert(55)
arr.insert(12.34)
arr.insert(0)
arr.insert("baz")
```

```
arr.insert(-17)

print("Array containing", arr.nItems, "items")
arr.traverse()

print("Search for 12 returns", arr.search(12))

print("Search for 12.34 returns", arr.search(12.34))

print("Deleting 0 returns", arr.delete(0))
print("Deleting 17 returns", arr.delete(17))

print("Array after deletions has", arr.nItems, "items")
arr.traverse()
```

To run the program, you can use a command-line interpreter to navigate to a folder where both files are present and run the following:

```
$ python3 BadArrayClient.py
Array containing 10 items
77
99
foo
bar
44
55
12.34
0
baz
-17
Search for 12 returns None
Search for 12.34 returns 12.34
Traceback (most recent call last):
  File "BadArrayClient.py", line 23, in <module>
    print("Deleting 0 returns", arr.delete(0))
  File "/Users/canning/chapters/02 code/BadArray.py", line 24, in delete
    self.__a[k] = self.__a[k+1]   # right over 1
IndexError: list index out of range
```

The results show that most of the methods work properly; this example illustrates the use of the public instance attribute, `nItems`, to provide the number of items in the `Array`. The Python traceback shows that there's a problem with the `delete()` method. The error is that the list index was out of range. That means either `k` or `k+1` was out of range in the

line displayed in the traceback. Going back to the code in `BadArray.py`, you can see that `k` lies in the range of `j` up to but not including `self.nItems`. The index `j` can't be out of bounds because the method already accessed `__a[j]` and found that it matched `item` in the line preceding the `k` loop. When `k` gets to be `self.nItems - 1`, then `k + 1` is `self.nItems`, and that *is outside of the bounds* of the maximum size of the `list` initially allocated. So, we need to adjust the range that `k` takes in the loop to move array items. Before fixing that, let's look more at the details of all the algorithms used.

Insertion

Inserting an item into the `Array` is easy; we already know the position where the insertion should go because we have the number of current items that are stored. Listing 2-1 shows that the item is placed in the internal `list` at the `self.nItems` position. Afterward, the number of items attribute is increased so that subsequent operations will know that that element is now filled. Note that the method does not check whether the allocated space for the `list` is enough to accommodate the new item.

Searching

The `item` variable holds the value being sought. The `search` method steps through only those indices of the internal `list` within the current number of items, comparing the `item` argument with each array item. If the loop variable `j` passes the last occupied element with no match being found, the value isn't in the `Array`. Appropriate messages are displayed by the `BadArrayClient.py` program: `Search for 12 returns None` or `Search for 12.34 returns 12.34.`

Deletion

Deletion begins with a search for the specified item. If found, all the items with higher index values are moved down one element to fill in the hole in the `list` left by the deleted item. The method decrements the instance's `nItems` attribute, but you've already seen an error happen before that point. Another possibility to consider is what happens if the item isn't found. In the implementation of Listing 2-1, it returns `False`. Another approach would be to raise an exception in that case.

Traversal

Traversing all the items is straightforward: We step through the `Array`, accessing each one via the private instance variable `__a[j]` and apply the `(print)` function to it.

Observations

In addition to the bug discovered, the `BadArray.py` module does not provide methods for accessing or changing arbitrary items in the `Array`. That's a fundamental operation of arrays, so we need to include that. We will keep the code simple and focus attention on operations that will be common across many data structures.

The `Array` class demonstrates **encapsulation** (another aspect of object-oriented programming) by providing the four methods and keeping the underlying data stored in the `__a list` as private. Programs that use `Array` objects are able to access the data only through

those methods. The `nItems` attribute is public, which makes it convenient for access by `Array` users. Being public, however, opens that attribute to being manipulated by `Array` users, which could cause errors to occur. We address these issues in the next version of the program.

A Better Array Class Implementation

The next sample program shows an improved interface for the array storage structure class. The class is still called `Array`, and it's in a file called `Array.py`, as shown in Listing 2-3.

The constructor is almost the same except that the variable holding the number of items in the `Array` is renamed `__nItems`. The double underscore prefix marks this instance attribute as private. To make it easy to get the current number of items in `Array` instances, this example introduces a new method named `__len__()`. This is a special name to Python because objects that implement a `__len__()` method can be passed as arguments to the built-in Python `len()` function, like all other sequence types. By calling this function, programs that use the `Array` class can get the value stored in `__nItems` but are not allowed to set its value.

LISTING 2-3 The `Array.py` Module

```
# Implement an Array data structure as a simplified type of list.

class Array(object):
   def __init__(self, initialSize):    # Constructor
      self.__a = [None] * initialSize  # The array stored as a list
      self.__nItems = 0                # No items in array initially

   def __len__(self):                  # Special def for len() func
      return self.__nItems             # Return number of items

   def get(self, n):                   # Return the value at index n
      if 0 <= n and n < self.__nItems: # Check if n is in bounds, and
         return self.__a[n]            # only return item if in bounds

   def set(self, n, value):            # Set the value at index n
      if 0 <= n and n < self.__nItems: # Check if n is in bounds, and
         self.__a[n] = value           # only set item if in bounds

   def insert(self, item):             # Insert item at end
      self.__a[self.__nItems] = item   # Item goes at current end
      self.__nItems += 1               # Increment number of items
```

```
   def find(self, item):                # Find index for item
      for j in range(self.__nItems):    # Among current items
         if self.__a[j] == item:        # If found,
            return j                    # then return index to item
      return -1                         # Not found -> return -1

   def search(self, item):              # Search for item
      return self.get(self.find(item)) # and return item if found

   def delete(self, item):              # Delete first occurrence
      for j in range(self.__nItems):    # of an item
         if self.__a[j] == item:        # Found item
            self.__nItems -= 1          # One fewer at end
            for k in range(j, self.__nItems):  # Move items from
               self.__a[k] = self.__a[k+1]     # right over 1
            return True                 # Return success flag

      return False     # Made it here, so couldn't find the item

   def traverse(self, function=print): # Traverse all items
      for j in range(self.__nItems):    # and apply a function
         function(self.__a[j])
```

It's important to know about Python's mechanisms to manage "private" attributes. The underscore prefix in a name indicates that the attribute *should be treated as private but does not guarantee it*. Using a double underscore prefix in an attribute like `__nItems` causes Python to use **name mangling**, making it harder but not impossible to access the attribute. It also makes accessing that attribute in subclasses more complex. To make private attributes that can be easily accessed by the same name in subclasses, use a single underscore prefix. For this example, we choose to keep the double underscore name to illustrate how to control public access, such as using the `__len__()` method.

The example in Listing 2-3 also introduces `get()` and `set()` methods that allow `Array` users to read and write the values of individual elements based on an index. This is the basic functionality of arrays in all languages. The `get()` method checks that the desired index is within the current bounds and returns the value if it is. Note that if the index is out of bounds, there is no explicit return value. Python functions and methods return `None` if execution reaches the end of the function body. The `set()` method checks that the index is in bounds and sets that cell's value if so, returning `None`.

The `insert()` method remains the same, but we change the `search()` method by breaking it up into two methods. We define a new `find()` method that finds the index to the item being sought. This method loops through the current items and returns the index of the item if it's found, or –1 if it isn't. We choose –1 for the return value because it cannot possibly be confused with a valid index value into the `list` and to guarantee the output of `find()` is always an integer (not `None`). The output of the `find()` method can thus

be passed to the `get()` method to get the item after finding its index. If the item is not found, `find()` returns –1, and `get()` and `search()` still return `None`.

We alter the `delete()` method to fix the index out-of-bounds error in `BadArray.py` by moving the decrement of `__nItems` to occur *before* the loop that moves items to the left in the `list` (see Listing 2-3). The `traverse()` method remains the same.

LISTING 2-4 The `ArrayClient.py` Program

```
import Array
maxSize = 10                    # Max size of the array
arr = Array.Array(maxSize)      # Create an array object

arr.insert(77)                  # Insert 10 items
arr.insert(99)
arr.insert("foo")
arr.insert("bar")
arr.insert(44)
arr.insert(55)
arr.insert(12.34)
arr.insert(0)
arr.insert("baz")
arr.insert(-17)

print("Array containing", len(arr), "items")
arr.traverse()

print("Search for 12 returns", arr.search(12))

print("Search for 12.34 returns", arr.search(12.34))

print("Deleting 0 returns", arr.delete(0))
print("Deleting 17 returns", arr.delete(17))

print("Setting item at index 3 to 33")
arr.set(3, 33)

print("Array after deletions has", len(arr), "items")
arr.traverse()
```

A new client program, `ArrayClient.py`, exercises the `Array` class, as shown in Listing 2-4. This program is almost identical to `BadArrayClient.py` but uses the new module name, `Array`, and tests the new features of the interface by calling the `len()` function on the `Array` and the `set()` method. The tests that call `search()` are also testing the new `find()` and `get()` methods.

You can confirm that the bug is fixed and no new bugs have shown up by running `ArrayClient.py` to see the following:

```
$ python3 ArrayClient.py
Array containing 10 items
77
99
foo
bar
44
55
12.34
0
baz
-17
Search for 12 returns None
Search for 12.34 returns 12.34
Deleting 0 returns True
Deleting 17 returns False
Setting item at index 3 to 33
Array after deletions has 9 items
77
99
foo
33
44
55
12.34
baz
-17
```

We now have a functional `Array` class that implements the four core methods for a data storage object. The code shown in the Array visualization tool is that of the `Array` class in Listing 2-3. Try using the visualization tool to search for an item and follow the highlights in the code. You will see that it calls the `search()` method, which calls the `find()` method. Those both show up separated by a gray line. When the `find()` method finishes, its local variables are erased, and its source code disappears, leaving the `search()` method to use its result and try to get the item. The visualization does not show the execution of the `get()` method but does show a message indicating whether the item was found or not.

You can also try out the allocation of a new array. The "New" operation allocates an array of a size you provide (if the cells fit on the screen). If you ask for a large number of cells, it makes them smaller, and the numbers may be hidden. The code shown for the New operation is that of the `__init__()` constructor for the `Array` class. The Random Fill operation fills any empty cells of the current array with random keys. The Delete Rightmost removes

the last item from the array. These aren't methods in the basic `Array` class, but they are helpful for the visualization.

The OrderedArray Visualization Tool

Imagine an array in which the data items are arranged in order of ascending values—that is, with the smallest value at index 0, and each cell holding a value larger than the cell below. Such an array is called an **ordered array**.

When you insert an item into this array, the correct location must be found for the insertion: just above a smaller value and just below a larger one. Then all the larger values must be moved up to make room.

Why would you want to arrange data in order? One advantage is that you can speed up search times dramatically using a **binary search**. At the same time, you are making the insert operation more complex because it must find the proper location for each new item.

To get a feel for what these changes bring, start the OrderedArray Visualization tool, using the procedure described in Appendix A. You see an array; it's similar to the one in the Array Visualization tool, but the data is ordered. Figure 2-4 shows what this tool looks like when it starts.

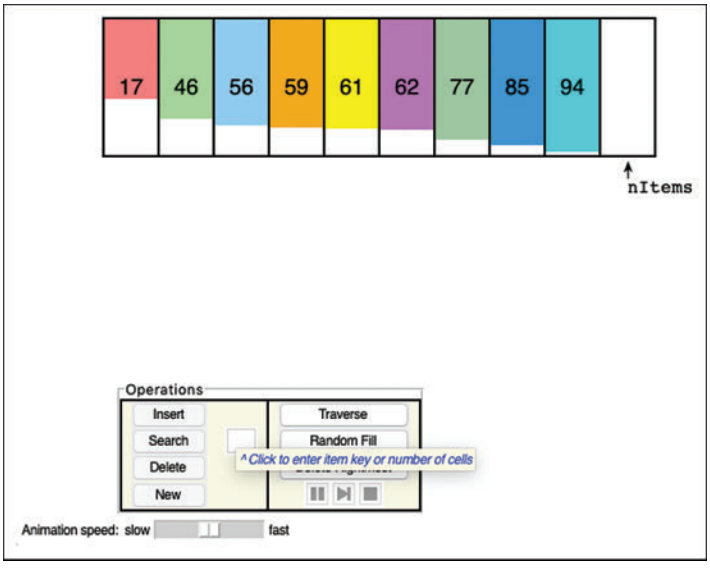


FIGURE 2-4 The OrderedArray Visualization tool

Linear Search

Before we describe how ordering the array helps, we need to elaborate on the kinds of searches we're discussing. The search algorithm used in the (unordered) Array Visualization tool is called a **linear search**. A linear search operates just like someone running a finger over a list of items to find a match. In the visualization, a brown arrow steps along, until it finds a match or reaches the `nItems` limit.

Binary Search

The payoff for using an ordered array comes when you use a binary search. You use this for the Search operation because it is much faster than a linear search, especially for large arrays.

The Guess-a-Number Game

Binary search is a classic approach to guessing games. In the Guess-a-Number game, a friend asks you to guess a number between 1 and 100 (Figure 2-5). When you guess a number, she tells you one of three things:

- Your guess is larger than the number she's thinking of, or
- It's smaller, or
- You guessed correctly.

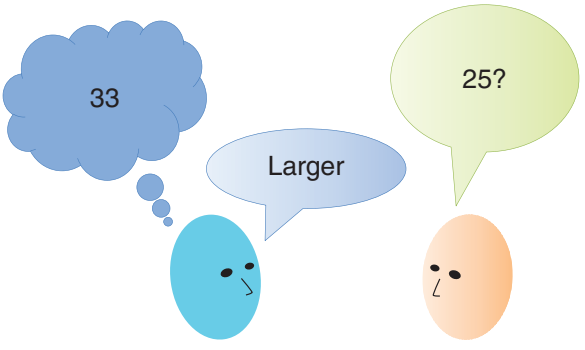


FIGURE 2-5 The Guess-a-Number Game

To find the number in the fewest guesses, you should always start by guessing 50. If your friend says your guess is too low, you deduce the number is between 51 and 100, so your next guess should be 75 (halfway between 51 and 100). If she says it's too high, you deduce the number is between 1 and 49, so your next guess should be 25.

Each guess allows you to divide the range of possible values in half. Finally, if you haven't already found it on an early guess, the range is only one number long, and that's the answer.

Notice how few guesses are required to find the number. If you used a linear search, guessing first 1, then 2, then 3, and so on, finding the number would take you, on the average, 50 guesses. In a binary search, each guess divides the range of possible values in half, so the number of guesses required is far fewer. Table 2-2 shows a game session when the number to be guessed is 33.

Table 2-2 Guessing a Number

Step Number	Number Guessed	Result	Range of Possible Values
0			1–100
1	50	Too high	1–49
2	25	Too low	26–49
3	37	Too high	26–36
4	31	Too low	32–36
5	34	Too high	32–33
6	32	Too low	33–33
7	33	Correct	

The correct number is identified in only 7 guesses. This is the maximum number of guesses, regardless of the number chosen by your friend. You might get lucky and guess the number before you've worked your way all the way down to a range of one. This would happen if the number to be guessed was 50, for example, or 34. The most important thing to remember is that you will always find the number in 7 or fewer guesses, which compares well with the maximum of 100 guesses and average of 50 guesses if you search linearly, ignoring the too high and too low clues.

Binary Search in the OrderedArray Visualization Tool

If you change the Guess-a-Number game into a Where-is-a-Number game, you can use the same strategy in searching the ordered array. It's a subtle shift, but the question is now asking, "What is the index of the cell holding number X?" The indices range from 0 to N–1, so there is the same kind of range to be searched. You can start with the middle array cell and then narrow the range based on what you find there.

Let's see that process in action on a 10-element array. First insert another value, say 55, into the ordered array by typing `55` in the text entry box and selecting Insert. Then try searching for the newly inserted value by typing it again and selecting Search. The OrderedArray Visualization tool shows the array and adds three arrows labeled `lo`, `mid`, and `hi`. The `lo` and `hi` arrows point to the first and last cells of the array, respectively. The `mid` arrow is placed at the midpoint between them, as shown in the top part in Figure 2-6.

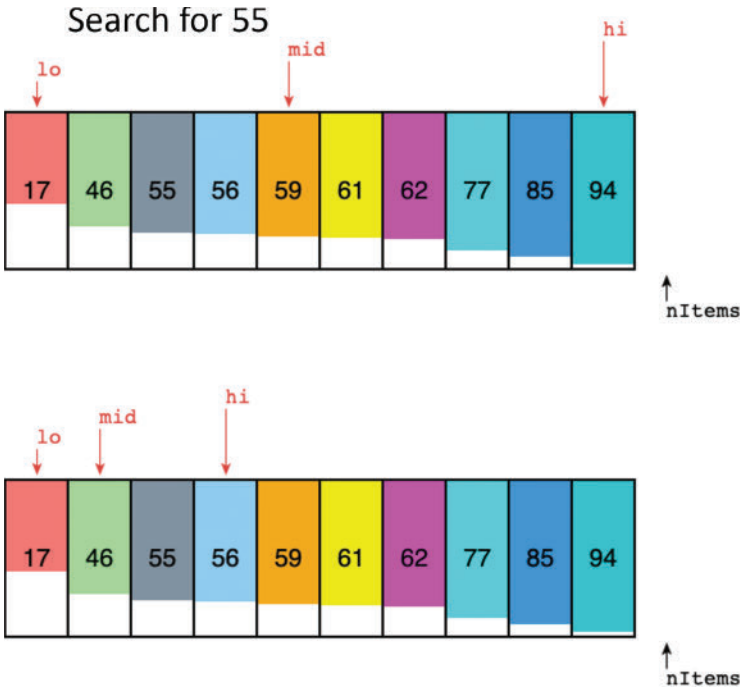


FIGURE 2-6 Initial ranges in a binary search

When the range of `lo` to `hi` spans an odd number of cells, the midpoint is the same number of cells from either end, but when it's even, it must be closer to one or the other end. The visualization tool always chooses `mid` to be closer to `lo`, and you'll see why when looking at the code.

After comparing the value at `mid` (59) with the value you're trying to find, the algorithm determines that 55 must lie in the range to the left of `mid`. It moves the `hi` arrow to be one less than `mid` to narrow the range and leaves `lo` unchanged. Then it updates `mid` to be at the midpoint of the narrowed range. The bottom of Figure 2-6 shows this second step of the process.

Each step reduces the range by about half. With the initial 10-element array, the ranges go from 10 to 5, to 2, to 1 at the very most (in Figure 2-6, it goes from 10 to 4 to 2, before finding 55 at index 2). If `mid` happens to point at the goal item, the search can stop. Otherwise, it will continue until the range collapses to nothing.

Try a few searches to see how quickly the visualization tool finds values. Try a search for a value not in the array to see what happens. With a 10-element array, it will take at most four values for `mid` to determine whether the value is present in the array.

What about larger arrays? Use the New operation to find out. Select the text entry box, enter `35`, and then select New. If there's enough room in the tool window, it will draw 35 empty cells of a new array. Fill them with random values by selecting Random Fill. The

cells show colored rectangles, but the numbers disappear when the cells are too skinny. You can try a variation of the Guess-a-Number game here by typing in a value, selecting Search, and seeing whether it ended up in the array. If you succeed, the tool will add an oval to the cell and provide a success message at the end. You can also select a colored rectangle with your pointer, and it will fill in its value in the text entry area.

Can you figure out how many steps the binary search algorithm will take to find a number based on the size of the array you're searching? We return to this question in the last section of this chapter.

Duplicates in Ordered Arrays

We saw that the presence of duplicate values affected the number of comparisons and shifts needed in searches and deletes on unordered arrays. Does that change for ordered arrays? Yes, a little bit.

Let's look at searching first because it affects insertion and deletion. If finding only a single matching item is sufficient, then there's no difference whether the array has duplicates or not. When you find the first one, you're done. If searches must return *all* matching items, the binary search algorithm finds only the first of them. After that, you would need to find any duplicates to the left or right of the one discovered by binary search. That could be done with linear searching, and it would need to search only the last `lo` to `hi` range explored by the binary search. That would add extra steps, possibly up to visiting all N items because the entire array could be duplicates of the same value. On average, however, it would be much less.

Note that a successful binary search does not guarantee finding the item with the lowest or highest index among those with duplicate keys. It guarantees finding one of them, but finding the relative position of that item to the others requires the extra linear searching.

Insertion remains the same for ordered arrays when duplicates are permitted. If a duplicate key exists, the binary search will find one of the duplicates and insert the new item beside it. If the new item has a unique key, it will be placed in order as before. You still need to shift values over to correctly insert any new value. The presence of duplicates might mean shifting fewer values, if the item to insert matches the value of two or more existing items. The presence of a few duplicates, however, does not significantly change the average number of comparisons and shifts needed, which remain about N/2.

For deletions, the effect of duplicates is also a bit complicated. If deleting only one of the matching items is sufficient, you can use binary search to find the item and shift items to the right of it to fill the hole caused by its removal. If there are many duplicates, you could save some shifts by shifting only the rightmost duplicate to fill the hole, but finding the rightmost takes almost as much time as shifting all the duplicates in between.

If deletion requires deleting *all* matching items, the complexity that we discussed for the search operation applies to the deletion process too. There would be fewer shifts needed after finding all the duplicates because you can shift values over D cells as fast as shifting over 1 cell, where D is the number of matching duplicate items. Overall, however, there is not much of a difference compared to the no-duplicates case because the number of operations is still proportional to N.

Python Code for an OrderedArray Class

Let's examine some Python code that implements an ordered array. This example uses the `OrderedArray` class to encapsulate the underlying `list` and its algorithms. The heart of this class is the `find()` method, which uses a binary search to locate a specified data item. We examine this method in detail before showing the complete program.

Binary Search with the find() Method

The `find()` method searches for the index to a specified item by repeatedly dividing in half the range of list items to be considered. The method looks like this:

```
def find(self, item):            # Find index at or just below
   lo = 0                        # item in ordered list
   hi = self.__nItems-1          # Look between lo and hi

   while lo <= hi:
      mid = (lo + hi) // 2       # Select the midpoint
      if self.__a[mid] == item:  # Did we find it at midpoint?
         return mid              # Return location of item
      elif self.__a[mid] < item: # Is item in upper half?
         lo = mid + 1            # Yes, raise the lo boundary
      else:
         hi = mid - 1            # No, but could be in lower half

   return lo   # Item not found, return insertion point instead
```

The method begins by setting the `lo` and `hi` variables to the first and last indices in the array. Setting these variables specifies the range for where the `item` may be found. Then, within the `while` loop, the index, `mid`, is set to the middle of this range.

If you're lucky, `mid` may already be pointing to the desired item, so you first check if `self.__a[mid] == item` is true. If it is, you've found the item, and you return with its index, `mid`.

If `mid` does not point to the `item` being sought, then you need to figure out which half of the range it falls in. You check whether the `item` is bigger than the one at the midpoint by testing `self.__a[mid] < item`. If the `item` is bigger, then you can shrink the search range by setting the `lo` boundary to be 1 above the midpoint. Note that setting `lo` to be the same as `mid` would mean including that midpoint item in the remaining search. You don't want to include it because the comparison with the item at `mid` already showed that its value is too low. Finally, if the midpoint item is neither equal to nor less than the `item` being sought, it must be bigger. In this case, you can shrink the search range by setting `hi` to be 1 below the midpoint. Figure 2-7 shows how the range is altered in these two situations.

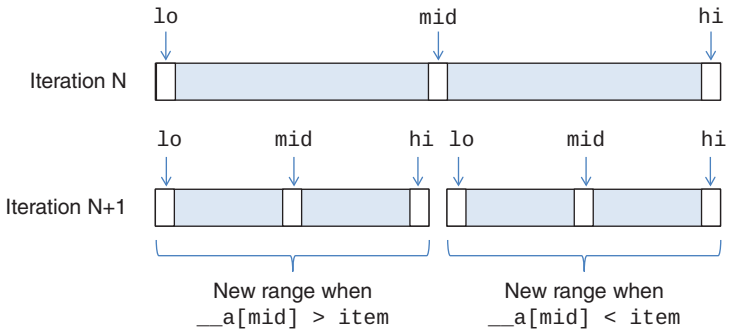


FIGURE 2-7 Dividing the range in a binary search

Each time through the loop you divide the range in half. Eventually, the range becomes so small that it can't be divided any more. You check for this in the loop condition: if `lo` is greater than `hi`, the range has ceased to exist. (When `lo` equals `hi`, the range is one and you need one more pass through the loop.) You can't continue the search without a valid range, but you haven't found the desired item, so you return `lo`, the last lower bound of the search range. This might seem odd because you're returning an index that doesn't point to the item being sought. It still could be useful, however, because it specifies where an item with that value would be placed in the ordered array.

The OrderedArray Class

In general, the `OrderedArray.py` program is similar to `Array.py` (refer to Listing 2-3). The main difference is that the `find()` method is changed to do a binary search, as we've discussed. Here, we show the class in two parts. Listing 2-5 shows the basic class infrastructure including the constructor, utility methods, and the traverse operation. Listing 2-6 shows the other three core operations, including the `find()` method.

The `OrderedArray` constructor is identical to that of the `Array` class; it allocates a list of the specified `initialSize` and sets the item count to 0. The `__len__()` and `traverse()` methods are identical too. In the `get()` method, one change has been added: it raises an exception if called on an index outside the range of active cells. The `IndexError` is a standard Python exception type used for this condition. A customized string message explains the problem.

The `OrderedArray` class includes a new `__str__()` method in Listing 2-5, which builds a string representation of the data items currently in the array. This is more than just a convenient utility for these tests; a Python object's `__str__()` method is invoked by the built-in `str()` function to create a string in contexts where one is needed, such as when the object is passed to the `print()` function. It uses the same syntax that Python uses for a string version of lists: a list of comma-separated values enclosed in square brackets.

LISTING 2-5 The Basic `OrderedArray` Class Definition

```
# Implement an Ordered Array data structure

class OrderedArray(object):
   def __init__(self, initialSize):    # Constructor
      self.__a = [None] * initialSize  # The array stored as a list
      self.__nItems = 0                # No items in array initially

   def __len__(self):                  # Special def for len() func
      return self.__nItems             # Return number of items

   def get(self, n):                   # Return the value at index n
      if 0 <= n and n < self.__nItems: # Check if n is in bounds, and
         return self.__a[n]            # only return item if in bounds
      raise IndexError("Index " + str(n) + " is out of range")

   def traverse(self, function=print): # Traverse all items
      for j in range(self.__nItems):   # and apply a function
         function(self.__a[j])

   def __str__(self):                  # Special def for str() func
      ans = "["                        # Surround with square brackets
      for i in range(self.__nItems):   # Loop through items
         if len(ans) > 1:              # Except next to left bracket,
            ans += ", "                # separate items with comma
         ans += str(self.__a[i])       # Add string form of item
      ans += "]"                       # Close with right bracket
      return ans
```

Note that we intentionally leave out the `set()` method because that would allow callers to change values in ways that might not keep the items in order.

Listing 2-6 starts with the `find()` method that implements the binary search algorithm. The `search()` method changes a little from that of the `Array`. It first calls `find()` and verifies that the index returned is in bounds. If not, or if the indexed item doesn't match the sought item, it returns `None`. That means searching for an item not in the array will return `None` without raising an exception.

A check at the beginning of the `insert()` method determines whether the array is full. This is done by comparing the length of the Python `list`, `__a`, to the number of items currently in the array, `__nItems`. If `__nItems` is equal to (or somehow, larger than) the size of the `list`, inserting another item will overflow it, so the method raises an exception.

LISTING 2-6 The Core Operations of the `OrderedArray` Class

```
class OrderedArray(object):
…
   def find(self, item):                # Find index at or just below
      lo = 0                            # item in ordered list
      hi = self.__nItems-1              # Look between lo and hi

      while lo <= hi:
         mid = (lo + hi) // 2           # Select the midpoint
         if self.__a[mid] == item:      # Did we find it at midpoint?
            return mid                  # Return location of item
         elif self.__a[mid] < item:     # Is item in upper half?
            lo = mid + 1                # Yes, raise the lo boundary
         else:
            hi = mid - 1                # No, but could be in lower half

      return lo   # Item not found, return insertion point instead

   def search(self, item):
      index = self.find(item)           # Search for item
      if index < self.__nItems and self.__a[index] == item:
         return self.__a[index]         # and return item if found

   def insert(self, item):              # Insert item into correct position
      if self.__nItems >= len(self.__a): # If array is full,
         raise Exception("Array overflow") # raise exception

      index = self.find(item)           # Find index where item should go
      for j in range(self.__nItems, index, -1): # Move bigger items
         self.__a[j] = self.__a[j-1]            # to the right

      self.__a[index] = item            # Insert the item
      self.__nItems += 1                # Increment the number of items

   def delete(self, item):              # Delete any occurrence
      j = self.find(item)               # Try to find the item
      if j < self.__nItems and self.__a[j] == item:  # If found,
         self.__nItems -= 1             # One fewer at end
         for k in range(j, self.__nItems): # Move bigger items left
            self.__a[k] = self.__a[k+1]
         return True                    # Return success flag

      return False                      # Made it here; item not found
```

Otherwise, the `insert()` method calls `find()` to locate where the new item goes. Then it uses a loop over the indices to the right of the insertion `index` to move those items one cell to the right. The loop uses `range(self.__nItems, index, -1)` to go backward through the indices from `__nItems` to `index + 1`. The number of items to be moved could be all N of them if the new item is the smallest. On average, it will move half the current items.

The `delete()` method calls `find()` to figure out the location of the item to be deleted and whether it is in the array. If it does find the item, it also must move half the current items to the left on average. If not, it can return `False` without moving anything.

Like before, we use a separate client program to test the operations of the class and the utility methods. The `OrderedArrayClient.py` program appears in Listing 2-7.

LISTING 2-7 The `OrderedArrayClient.py` Program

```
from OrderedArray import *

maxSize = 1000               # Max size of the array
arr = OrderedArray(maxSize)  # Create the array object

arr.insert(77)               # Insert 11 items
arr.insert(99)
arr.insert(44)               # Inserts not in order
arr.insert(55)
arr.insert(0)
arr.insert(12)
arr.insert(44)
arr.insert(99)
arr.insert(77)
arr.insert(0)
arr.insert(3)

print("Array containing", len(arr), "items:", arr)

arr.delete(0)                # Delete a few items
arr.delete(99)
arr.delete(0)                # Duplicate deletes
arr.delete(0)
arr.delete(3)

print("Array after deletions has", len(arr), "items:", arr)

print("find(44) returns", arr.find(44))
print("find(46) returns", arr.find(46))
print("find(77) returns", arr.find(77))
```

Note that you can pass the `arr` variable directly to `print` and expect a reasonable output because of the `__str__()` method. We also use a different form of the `import` statement in `OrderedArrayClient.py`. By importing the module using the "`from` *module* `import *`" syntax, the definitions it contains are added in the same namespace as the client program, not in a new namespace for the module. That means you can create the object using the expression `OrderedArray(maxSize)` instead of `OrderedArray.OrderedArray(maxSize)`. The output of the program looks like this:

```
$ python3 OrderedArrayClient.py
Array containing 11 items: [0, 0, 3, 12, 44, 44, 55, 77, 77, 99, 99]
Array after deletions has 7 items: [12, 44, 44, 55, 77, 77, 99]
find(44) returns 1
find(46) returns 3
find(77) returns 5
```

The last three print statements illustrate some particular cases of the binary search with duplicate entries. The result of `find(46)` shows that even though 46 is not in `arr`, it should be inserted after the first three items to preserve ordering. The `find(44)` finds the first occurrence of 44 at position 1. If `delete(44)` were called at this point, it would delete the first of the 44s currently in the array. By contrast, `find(77)` points at the second of the two 77s in the array. The binary search stops after it finds the first matching item, which could be any instance of an item that appears multiple times.

Advantages of Ordered Arrays

What have we gained by using an ordered array? The major advantage is that search times are much faster than in an unordered array. The disadvantage is that insertion takes longer because all the data items with a higher key value must be moved up to make room. Even though it uses binary search to find where the key belongs, it still must move most of the items in the array.

Deletions are slow in both ordered and unordered arrays because items must be moved down to fill the hole left by the deleted item. There is a bit of speed-up for requests to delete items not in the array. By using `find()`, you can quickly discover whether any items need to be moved in the array. That benefit, however, is reduced when the requested item is in the array. Finding the item with a binary search replaces a linear search over the left side of the array. Both linear and binary searching require shifting items to right of the deleted item.

Going back to insertion for a moment, would it be simpler to skip the binary search of `find()` and just move items to the right until you reach an item smaller than the item being inserted? That saves a function call to `find()` but requires more comparisons. The binary search algorithm uses way fewer than N comparisons to find the insertion point, and if it doesn't call `find()`, the insert code must compare values for all N items being shifted.

Ordered arrays are therefore useful in situations in which searches are frequent, but insertions and deletions are not. An ordered array might be appropriate for a database of a

transport company that tracks the location of its vehicles, for example. The need to add and delete the names of a fleet of vehicles happens much less frequently than the events that require updates to their position, so having fast search find the right vehicle for each position update would be important and worth the increased time needed to add each new vehicle. On the other hand, if the transport company keeps a log of the tasks assigned to each vehicle or their actions in picking up and delivering cargo, there would be frequent additions to the log, but perhaps little need to find a particular log entry. The task log could benefit from the data structure with fast insertion time at the expense of a longer search.

Logarithms

In this section we explain how you can use logarithms to calculate the number of steps necessary in a binary search. If you're a math fan, you can probably skip this section. If thinking about math makes you nervous, give it a try, and make sure to take a long, hard look at Table 2-3.

A binary search provides a significant speed increase over a linear search. In the number-guessing game, with a range from 1 to 100, a maximum of seven guesses is needed to identify any number using a binary search; just as in an array of 100 records, a maximum of seven comparisons is needed to find a record with a specified key value. How about other ranges? Table 2-3 shows some representative ranges and the number of comparisons needed for a binary search.

Table 2-3 Comparisons Needed in a Binary Search

Range	Comparisons Needed
10	4
100	7
1,000	10
10,000	14
100,000	17
1,000,000	20
10,000,000	24
100,000,000	27
1,000,000,000	30

Notice the differences between binary search times and linear search times. For very small numbers of items, the difference isn't dramatic. Searching 10 items would take an average of 5 comparisons with a linear search (N/2) and a maximum of 4 comparisons with a binary search. But the more items there are, the bigger the difference. With 100 items, there are 50 comparisons in a linear search, but only 7 in a binary search. For 1,000 items,

the numbers are 500 versus 10, and for 1,000,000 items, they're 500,000 versus 20. You can conclude that for all but very small arrays, the binary search is greatly superior.

The Equation

You can verify the results of Table 2-3 by repeatedly dividing a range (from the first column) in half until it's too small to divide further. The number of divisions this process requires is the number of comparisons shown in the second column.

Repeatedly dividing the range by two is an algorithmic approach to finding the number of comparisons. You might wonder if you could also find the number using a simple equation. Of course, there is such an equation, and it's worth exploring here because it pops up from time to time in the study of data structures. This formula involves logarithms. (Don't panic yet.)

You have probably already experienced logarithms, without having recognized them. Have you ever heard someone say "a six figure salary" or read about "a deal worth eight figures"? Those simplified expressions tell you the approximate amount of the salary or deal by telling you how many digits are needed to write the number. The number of digits could be found by repeatedly dividing the number by 10. When it's less than 1, the number of divisions is the number of digits.

The numbers in Table 2-3 leave out some interesting data. They don't answer such questions as "What is the exact size of the maximum range that can be searched in five steps?" To solve this problem, you can create a similar table, but one that starts at the beginning, with a range of one, and works up from there by multiplying the range by two each time. Table 2-4 shows how this looks for the first seven steps.

Table 2-4 Powers of Two

Step s, same as $\log_2(r)$	Range r	Range Expressed as Power of 2 (2^s)
0	1	2^0
1	2	2^1
2	4	2^2
3	8	2^3
4	16	2^4
5	32	2^5
6	64	2^6
7	128	2^7
8	256	2^8
9	512	2^9
10	1,024	2^{10}

For the original problem with a range of 100, you can see that 6 steps don't produce a range quite big enough (64), whereas 7 steps cover it handily (128). Thus, the 7 steps that are shown for 100 items in Table 2-3 are correct, as are the 10 steps for a range of 1,000.

Doubling the range each time creates a series that's the same as raising 2 to a power, as shown in the third column of Table 2-4. We can express this power as a formula. If *s* represents steps (the number of times you multiply by 2—that is, the power to which 2 is raised) and *r* represents the range, then the equation is

$$r = 2^s$$

If you know *s*, the number of steps, this tells you *r*, the range. For example, if *s* is 6, the range is 2^6, or 64.

The Opposite of Raising 2 to a Power

The original question was the opposite of the one just described: Given the range, you want to know how many comparisons are required to complete a search. That is, given *r*, you want an equation that gives you *s*.

The inverse of raising something to a power is called a **logarithm**. Here's the formula you want, expressed with a logarithm:

$$s = \log_2(r)$$

This equation says that the number of steps (comparisons) is equal to the logarithm to the base 2 of the range. What's a logarithm? The base 2 logarithm of a number *r* is the number of times you must multiply 2 by itself to get *r*. In Table 2-4, the step numbers in the first column, *s*, are equal to $\log_2(r)$.

How do you find the logarithm of a number without doing a lot of dividing? Most calculators and computer languages have a log function. For those that don't, sometimes it can be added as an option, such as with Python's `math` module. It might only provide a function for log to the base 10, but you can convert easily to base 2 by multiplying by 3.322. For example, $\log_{10}(100) = 2$, so $\log_2(100) = 2$ times 3.322, or 6.644. Rounded up to the whole number 7, this is what appears in the column to the right of 100 in Table 2-3.

In any case, the point here isn't to calculate logarithms. It's more important to understand the relationship between a number and its logarithm. Look again at Table 2-3, which compares the number of items and the number of steps needed to find a particular item. Every time you multiply the number of items (the range) by a factor of 10, you add only three or four steps (actually 3.322, before rounding off to whole numbers) to the number needed to find a particular item. This is true because, as a number grows larger, its logarithm doesn't grow nearly as fast. We compare this logarithmic growth rate with that of other mathematical functions when we talk about Big O notation later in this chapter.

Storing Objects

In the examples we've shown so far, we've stored single values in array data structures such as integers, floating-point numbers, and strings. Storing such simple values simplifies

the program examples, but it's not representative of how you use data storage structures in the real world. Usually, the data you want to store comprises many values or fields, usually called a **record**. For a personnel record, you might store the family name, given name, birth date, first working date, identification number, and so forth. For a fleet of vehicles, you might store the type of vehicle, the name, the date it entered service, a license tag, and so forth. In object-oriented programs, you want to store the objects themselves in data structures. The objects can represent records.

When storing objects or records in ordered data structures, like the `OrderedArray` class, you need to define the way the records are ordered by specifying a **key** that can be used on all of them. Let's look at how that changes the implementation.

The OrderedRecordArray Class

As shown in the previous examples, you can insert any data type into Python arrays. It's very convenient to allow complex data types to be inserted, deleted, and managed in the arrays, along with the benefits of storing them in order, which makes search faster. To distinguish them (and order them), you need a key for each record. The best way to do that is to define a function that extracts the key from a record and then use that function when comparing keys. By using a function, the array data structure doesn't need to know anything about format or organization of the record. All it must do is pass one of the records, let's say record R, as the argument to the function, F, to get that record's key, F(R).

The function for the key could be provided to the array data structure in several ways. The program needing the array could define a function with a known name like `array_key` that fetches the key. That approach wouldn't be very portable, and it would make it impossible to have different arrays using different key functions. The key function could also be passed as an argument to the array's methods like `find` and `insert`. That would allow different arrays to use different key functions, but it has a potential problem. If the program using the arrays accidentally passes the wrong key function to an array that ordered its records by a different key, then the records could be out of order using the new key. Instead, it's better to define the key function *when the array is created* and not allow it to change. That's how we implement the `OrderedRecordArray` class, as shown in Listing 2-8.

LISTING 2-8 The Basic `OrderedRecordArray` Class

```
# Implement an Ordered Array of Records structure

def identity(x):                        # The identity function
   return x

class OrderedRecordArray(object):
   def __init__(self, initialSize, key=identity):    # Constructor
      self.__a = [None] * initialSize  # The array stored as a list
      self.__nItems = 0                # No items in array initially
      self.__key = key                 # Key function gets record key
```

```
   def __len__(self):                  # Special def for len() func
      return self.__nItems             # Return number of items

   def get(self, n):                   # Return the value at index n
      if n >= 0 and n < self.__nItems: # Check if n is in bounds, and
         return self.__a[n]            # only return item if in bounds
      raise IndexError("Index " + str(n) + " is out of range")

   def traverse(self, function=print): # Traverse all items
      for j in range(self.__nItems):   # and apply a function
         function(self.__a[j])

   def __str__(self):                  # Special def for str() func
      ans = "["                        # Surround with square brackets
      for i in range(self.__nItems):   # Loop through items
         if len(ans) > 1:              # Except next to left bracket,
            ans += ", "                # separate items with comma
         ans += str(self.__a[i])       # Add string form of item
      ans += "]"                       # Close with right bracket
      return ans
```

The constructor for `OrderedRecordArray` takes a new argument, `key`, which is the key function. That function defaults to being the `identity` function, which is defined in the module and simply returns the first argument as the result. This makes the default behavior the same as the `OrderedArray` class shown in Listings 2-5 and 2-6. The key function is stored in the private instance variable `__key` so it should not be modified by the clients using `OrderedRecordArrays`.

Listing 2-9 shows that the `find()` and `search()` methods change to take a `key` as an argument, instead of the item or record used in the `OrderedArray` class. This key is a value, not a function, and is used to compare with the keys extracted from the records in the array. The `find` and `search` methods use the `internal __key()` function on the records to get the right value to compare with the `key` being sought. The `insert()` and `delete()` method signatures don't change—they still operate on `item` records—but internally they change the way they pass the appropriate key to `find()`.

LISTING 2-9 The Item Operations of the `OrderedRecordArray` Class

```
class OrderedRecordArray(object):
…
   def find(self, key):             # Find index at or just below key
      lo = 0                        # in ordered list
      hi = self.__nItems-1          # Look between lo and hi
```

```
        while lo <= hi:
            mid = (lo + hi) // 2           # Select the midpoint

            if self.__key(self.__a[mid]) == key:  # Did we find it?
                return mid                 # Return location of item

            elif self.__key(self.__a[mid]) < key: # Is key in upper half?
                lo = mid + 1               # Yes, raise the lo boundary

            else:
                hi = mid - 1               # No, but could be in lower half

        return lo   # Item not found, return insertion point instead

    def search(self, key):
        idx = self.find(key)               # Search for a record by its key
        if idx < self.__nItems and self.__key(self.__a[idx]) == key:
            return self.__a[idx]           # and return item if found

    def insert(self, item):                # Insert item into the correct position
        if self.__nItems >= len(self.__a): # If array is full,
            raise Exception("Array overflow") # raise exception

        j = self.find(self.__key(item))     # Find where item should go

        for k in range(self.__nItems, j, -1): # Move bigger items right
            self.__a[k] = self.__a[k-1]

        self.__a[j] = item                 # Insert the item
        self.__nItems += 1                 # Increment the number of items

    def delete(self, item):                # Delete any occurrence
        j = self.find(self.__key(item))    # Try to find the item
        if j < self.__nItems and self.__a[j] == item:  # If found,
            self.__nItems -= 1             # One fewer at end
            for k in range(j, self.__nItems): # Move bigger items left
                self.__a[k] = self.__a[k+1]
            return True                    # Return success flag

        return False                       # Made it here; item not found
```

The test program for this new class, `OrderedRecordArrayClient.py` shown in Listing 2-10, uses records with two elements or fields. The key for the records is set to be the second

element of each record. Loops in this version perform the insertions, deletions, and searches.

LISTING 2-10 The `OrderedRecordArrayClient.py` Program

```
from OrderedRecordArray import *

def second(x):                # Key on second element of record
    return x[1]

maxSize = 1000                            # Max size of the array
arr = OrderedRecordArray(maxSize, second)  # Create the array object

# Insert 10 items
for rec in [('a', 3.1), ('b', 7.5), ('c', 6.0), ('d', 3.1),
            ('e', 1.4), ('f', -1.2), ('g', 0.0), ('h', 7.5),
            ('i', 7.5), ('j', 6.0)]:
    arr.insert(rec)

print("Array containing", len(arr), "items:\n", arr)

# Delete a few items, including some duplicates
for rec in [('c', 6.0), ('g', 0.0), ('g', 0.0),
            ('b', 7.5), ('i', 7.5)]:
    print("Deleting", rec, "returns", arr.delete(rec))

print("Array after deletions has", len(arr), "items:\n", arr)

for key in [4.4, 6.0, 7.5]:
    print("find(", key, ") returns", arr.find(key),
          "and get(", arr.find(key), ") returns",
          arr.get(arr.find(key)))
```

After putting 10 records in the array including some with duplicate keys, the test program deletes a few records, showing the result of the deletion. It then tries to find a few keys in the reduced array. The result of running the program is

```
$ python3 OrderedRecordArrayClient.py
Array containing 10 items:
 [('f', -1.2), ('g', 0.0), ('e', 1.4), ('d', 3.1), ('a', 3.1), ('j', 6.0),
('c', 6.0), ('i', 7.5), ('h', 7.5), ('b', 7.5)]
Deleting ('c', 6.0) returns False
Deleting ('g', 0.0) returns True
Deleting ('g', 0.0) returns False
Deleting ('b', 7.5) returns False
Deleting ('i', 7.5) returns True
```

```
Array after deletions has 8 items:
 [('f', -1.2), ('e', 1.4), ('d', 3.1), ('a', 3.1), ('j', 6.0), ('c', 6.0), ('h',
7.5), ('b', 7.5)]
find( 4.4 ) returns 4 and get( 4 ) returns ('j', 6.0)
find( 6.0 ) returns 5 and get( 5 ) returns ('c', 6.0)
find( 7.5 ) returns 6 and get( 6 ) returns ('h', 7.5)
```

The program output shows that deleting the record `('c', 6.0)` fails. Why? The next two deletions show that deleting `('g', 0.0)` succeeds the first time but fails the second time because only one record has that key, 0.0. That's what is expected, but the next deletions are unexpected. The deletion of the record `('b', 7.5)` fails, but the deletion of `('i', 7.5)` succeeds. What is going on?

The issue comes up because of the *duplicate keys*. The program inserts three records that have the key 7.5. When the `find()` method runs, it uses binary search to get the index to one of those records. The exact one it finds depends on the sequence of values for the `mid` variable. You can see which one it finds in the output of the `find` tests. Note that `find(4.4)` returns a valid index, 4, and that points to the location where a record with that key should go. The record at index 4 has the next higher key value, 6.0. When you call `find(7.5)` on the final `Array`, it returns 6, which points to the `('h', 7.5)` record. That isn't equal to the`('b', 7.5)` record using Python's == test. The `delete()` method removes only items that pass the == test. You can also deduce that `find(7.5)` did find the `('i', 7.5)` record on the earlier delete operation. This example illustrates an important issue when duplicate keys are allowed in a sorted data structure like `OrderedRecordArray`. One of the end-of-chapter programming projects asks you to change the behavior of this class to correctly delete records with duplicate keys.

Big O Notation

Which algorithms are faster than others? Everyone wants their results as soon as possible, so you need to be able compare the different approaches. You can certainly run experiments with each program on a particular computer and with a particular set of data to see which is fastest. That capability is useful, but when the computer changes or the data changes, you could get different results. Computers generally get faster as better technologies are invented, and that makes all algorithms run faster. The changes with the data, however, are harder to predict. You've already seen that a binary search takes far fewer steps than a linear search because the number of items to search increases. We'd like to be able to extend that reasoning to help predict what will happen with other algorithms.

People like to categorize things, especially by what they are capable of doing. If you think about cutting grass, there are push lawn mowers, powered lawn mowers, riding lawn mowers, and towed grass cutters. Each one of them is good for different size jobs of grass cutting. Similarly for refrigeration, there are personal refrigerators, household refrigerators, restaurant kitchen refrigerators, walk-in refrigerators, and refrigerated warehouses for different quantities of perishable items. In each case, choosing something too big or too small for the job would be costly in time or money.

In computer science, a rough measure of performance called **Big O** notation is used to describe algorithms. It's primarily used to describe the speed of algorithms but is also used to describe how much storage they need. Algorithms with the same Big O speed are in the same category. The category gives a rough idea of what amount of data they can process (or storage they need). For example, the linear search `Array` class in Listing 2-3 should be plenty fast for small jobs like keeping a list of contacts in a personal computer or phone or watch. It would probably not be acceptable for the list of contacts of a large corporation with tens of thousands of employees, and it certainly would be too slow to manage all the contact information for a nation of tens of millions of people. By using the `OrderedRecordArray` class in Listing 2-8, you can get the benefit of binary search and drastically reduce the search time by making it proportional to the logarithm of the number of items. Are there even better algorithms? Big O notation helps answer that question.

Insertion in an Unordered Array: Constant

Insertion into an unordered array is the only algorithm we've discussed that doesn't depend on how many items are in the array. The new item is always placed in the next available position, at `__a[self.__nItems]`, and `self.__nItems` is then incremented. Instead of using the variable names in a particular program, Big O notation uses N to stand for the number of items being managed. Insertion into an unordered array requires the same amount of time no matter how big N is. You can say that the time, T, to insert an item into an unsorted array is a constant, K:

$$T = K$$

In a real situation, the actual time required by the insertion is related to the speed of the processor, how efficiently the compiler has generated the program code, how much data is in the item being copied into the array, and other factors. The constant K in the preceding equation is used to account for all such factors. To find out what K is in a real situation, you need to measure how long an insertion took. (Software exists for this very purpose.) K would then be equal to that time.

Linear Search: Proportional to N

You've seen that, in a linear search of items in an array, the number of comparisons that must be made to find a specified item is, on the average, half of the total number of items. Thus, if N is the total number of items, the search time T is proportional to half of N:

$$T = K \times N / 2$$

As with insertions, discovering the value of K in this equation would require timing searches for some (probably large) values of N and then using the resulting values of T to calculate K. There is probably some time spent launching the program and cleaning up when it's done that would add a small constant factor to the total time. By measuring the time taken for several searches, you can account for the variations for where the item falls in the array and for the extra constant factors added by the measurement. When you know K and any additional constants, you can calculate T for any other value of N.

For a handier formula, you could lump the 2 into the K. The new K is equal to the old K divided by 2. Now you have

$$T = K \times N$$

This equation says that average linear search times are proportional to the size of the array. If an array is twice as big, searching it will take twice as long. In this case, we are less concerned with getting a precise estimate of the time it will take as we are knowing how fast it will grow as N gets bigger.

Binary Search: Proportional to log(N)

Similarly, you can concoct a formula relating T and N for a binary search:

$$T = K \times \log_2(N)$$

As you saw earlier, the search time is proportional to the base 2 logarithm of N. Actually, because any logarithm is related to any other logarithm by a constant (for example, multiplying by 3.322 to go from base 2 to base 10), you can lump this constant into K as well. Then you don't need to specify the base:

$$T = K \times \log(N)$$

Don't Need the Constant

Big O notation looks like the formulas just described, but it dispenses with the constant K. When comparing algorithms, you don't really care about the particular processor or compiler; all you want to compare is how T changes for different values of N, not what the actual numbers are. Although the K might be important for getting a precise estimate for small values of N, when N is really large, it "dominates" the time calculation. Therefore, we drop the constant in Big O notation.

Big O notation uses the uppercase letter *O*, which you can think of as meaning "order of." In Big O notation, you would say that a linear search takes O(N) or "Order of N" or simply "Order N" time, and a binary search takes O(log(N)) time. A further simplification of the notation gets rid of the parentheses for the log function, and you simply write O(log N). Insertion into an unordered array takes O(1), or constant time. (That's the numeral 1 in the parentheses.)

Table 2-5 summarizes the running times of the algorithms we've discussed so far.

Table 2-5 Running Times in Big O Notation

Algorithm	Unordered Array	Ordered Array
Linear search	O(N)	
Binary search	Not possible	O(log N)
Insertion	O(1)	O(N)
Deletion	O(N)	O(N)

You might ask why deletion in ordered arrays isn't shown as O(log N) + O(N) or maybe O(log N + N) because it uses binary search to find the location of the item to delete. The reason is that the O(N) part needed for shifting the items of the array is so much larger than the O(log N) part that it really doesn't matter when N gets big. Big O notation is intended to describe how the algorithm behaves for very large numbers of items.

Figure 2-8 graphs some Big O relationships between time (in number of steps) and number of items, N. Based on this graph, you might rate the various Big O values (very subjectively) like this:

- O(1) is excellent,
- O(log N) is good,
- O(N) is fair,
- O(N × log N) is poor, and
- O(N^2) is bad.

O(N × log N) occurs in many kinds of sorting. O(N^2) occurs in simple sorting and in certain graph algorithms, all of which we look at later in this book.

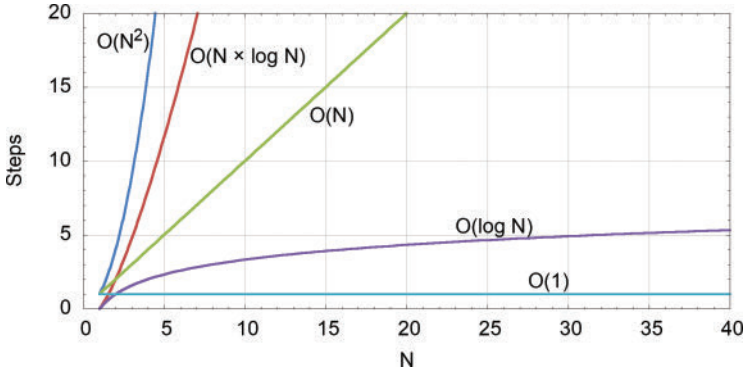


FIGURE 2-8 Graph of Big O times

The idea in Big O notation isn't to give actual figures for running times but to convey how the running time grows as the number of items increases. This is the most meaningful way to compare algorithms, without actually measuring running times in a real installation. This is often called the **computational complexity** of algorithms, or the **order of the function** that characterizes the running time (that's where the "O" comes from). Algorithms that are more complex for the computer to process have a higher order and are usually avoided.

Why Not Use Arrays for Everything?

Arrays seem to get the job done, so why not use them for all data storage? You've already seen some of their disadvantages. In an unordered array, you can insert items quickly, in O(1) time, but searching takes slow O(N) time. In an ordered array, you can search quickly, in O(log N) time, but insertion takes O(N) time. For both kinds of arrays, deletion takes O(N) time because half the items (on the average) must be moved to fill in the hole.

It would be nice if there were data structures that could do everything—insertion, deletion, and searching—quickly, ideally in O(1) time, but if not that, then in O(log N) time. Traversal, by definition, needs O(N) time, but in more complex data structures, it could be larger. In the chapters ahead, we examine how closely these ideals can be approached and the price that must be paid in complexity.

Another problem with arrays is that their size is fixed when they are first created. The reason for that is the compiler needs to know how much space to set aside for the whole array and keep it separate from all the other data. Usually, when the program first starts, you don't know exactly how many items will be placed in the array later, so you guess how big it should be. If the guess is too large, you'll waste memory by having cells in the array that are never filled. If the guess is too small, you'll overflow the array, causing at best a message to the program's user, and at worst a program crash.

Other data structures are more flexible and can expand to hold the number of items inserted in them. The linked list, discussed in Chapter 5, is such a structure. One of the programming projects in this chapter asks you to make an expanding array data structure.

Summary

- Arrays are sequential groupings of data elements. Each element can store a value called an item.
- Each element of the array can be accessed by knowing the start of the array and an integer index to the element.
- Object-oriented programs are used to implement data structures to encapsulate the algorithms that manipulate the data.
- Data structures use private instance variables to restrict access to important values of the structure that could cause errors if changed by the calling program.
- Unordered arrays offer fast insertion but slow searching and deletion.
- A binary search can be applied to an ordered array.
- The logarithm to the base B of a number A is (roughly) the number of times you can divide A by B before the result is less than 1.
- Linear searches require time proportional to the number of items in an array.
- Binary searches require time proportional to the logarithm of the number of items.
- Data structures usually store complex data types like records.

- A key must be defined to order complex data types.
- If duplicate items or keys are allowed in a data structure, the algorithms should have a predictable behavior for how they are managed.
- Big O notation provides a convenient way to compare the speed of algorithms.
- An algorithm that runs in O(1) time is the best, O(log N) is good, O(N) is fair, and $O(N^2)$ is bad.

Questions

These questions are intended as a self-test for readers. Answers may be found in Appendix C.

1. Aside from the insert, delete, search, and traverse methods common to all "database" data structures, array data structures should have _____ method(s).
2. When constructing a new instance of an `Array` (Listing 2-3):
   a. the initial value for at least one of the array cells must be set.
   b. the data type of all the array cells must be set.
   c. the key for each data item must be set.
   d. the maximum number of cells the array can hold must be set.
   e. none of the above.
3. Why is it important to use private instance attributes like `__nItems` in the definition of the array data structure?
4. Inserting an item into an unordered array
   a. takes time proportional to the size of the array.
   b. requires multiple comparisons.
   c. requires shifting other items to make room.
   d. takes the same time no matter how many items there are.
5. True or False: When you delete an item from an unordered array, in most cases you shift other items to fill in the gap.
6. In an unordered array, allowing duplicates
   a. increases times for all operations.
   b. increases search times in some situations.
   c. always increases insertion times.
   d. sometimes decreases insertion times.

7. True or False: In an unordered array, it's generally faster to find out an item is not in the array than to find out it is.
8. Ordered arrays, compared with unordered arrays, are
   a. much quicker at deletion.
   b. quicker at insertion.
   c. quicker to create.
   d. quicker at searching.
9. Keys are used with arrays
   a. to provide a single value for each array item that can be used to order the items.
   b. to decrypt the values stored in the array cell.
   c. to decrease the insertion time in unordered arrays.
   d. to allow complex data types to be stored as a single key value in the array.
10. The `OrderedArray.py` (Listing 2-6) and `OrderedRecordArray.py` (Listing 2-9) modules have both a `find()` and a `search()` method. How are the two methods the same and how do they differ?
11. A logarithm is the inverse of _____.
12. The base 10 logarithm of 1,000 is _____.
13. The maximum number of items that must be examined to complete a binary search in an array of 200 items is
    a. 200.
    b. 8.
    c. 1.
    d. 13.
14. The base 2 logarithm of 64 is _____.
15. True or False: The base 2 logarithm of 100 is 2.
16. Big O notation tells
    a. how the speed of an algorithm relates to the number of items.
    b. the running time of an algorithm for a given size data structure.
    c. the running time of an algorithm for a given number of items.
    d. how the size of a data structure relates to the speed of one of its algorithms.

17. O(1) means a process operates in _____ time.
18. Advantages of using arrays include
    a. the variable size of array cells.
    b. the variable length of the array over the lifetime of the data structure.
    c. the O(1) access time to read or write an array cell.
    d. the O(1) time to traverse all the items in the array.
    e. all of the above.
19. A colleague asks for your comments on a data structure using an unordered array without duplicates and binary search. Which of the following comments makes sense?
    a. Because the array can store any data type, a binary search won't be efficient.
    b. Because the array is unordered, a binary search cannot guarantee finding the item being sought.
    c. Because binary search takes O(N) time, it would be better to use an ordered array.
    d. Because the array doesn't have duplicates, binary search doesn't really have an advantage over the simpler linear search.
20. You've been asked to adapt some code that maintains a record about each planet and their moons in a solar system like ours into a system that will store a record about every planet and moon in every known galaxy. The record structure will be a little larger for each planet to hold some new attributes. It's likely that the records will be added and updated "randomly" as telescopes and other sensors point at different parts of the universe over time, filling in some initial attributes of the records, and then updating others during frequent observations. The current code uses an unordered array for the records. Would you recommend any changes? If so, why?

Experiments

Carrying out these experiments will help to provide insights into the topics covered in the chapter. No programming is involved.

2-A Use the Array Visualization tool to insert, search for, and delete items. Make sure you can predict what it's going to do. Do this both with duplicate values present and without.

2-B Make sure you can predict in advance what indices the OrderedArray Visualization tool will select at each step for `lo`, `mid`, and `hi` when you search for the lowest, second lowest, one above middle, and highest values in the array.

2-C In the OrderedArray Visualization tool, create an array of 12 cells and then use the Random Fill button to fill them with values. Use the Delete Rightmost button to

remove the five highest values. Then insert five of the same value, somewhere in the middle of the array. Note the colors that are assigned to inserted values and the order they were inserted. Can you predict the order they will be deleted? Try deleting the value you chose several times to see if your prediction is right.

Programming Projects

Writing programs to solve the Programming Projects helps to solidify your understanding of the material and demonstrates how the chapter's concepts are applied. (As noted in the Introduction, qualified instructors may obtain completed solutions to the Programming Projects on the publisher's website.)

2.1 To the `Array` class in the `Array.py` program (Listing 2-3), add a method called `getMaxNum()` that returns the value of the highest number in the array, or `None` if the array has no numbers. You can use the expression `isinstance(x, (int, float))` to test for numbers. Add some code to `ArrayClient.py` (Listing 2-4) to exercise this method. You should try it on arrays containing a variety of data types and some that contain zeros and some that contain no numbers.

2.2 Modify the method in Programming Project 2.1 so that the item with the highest numeric value is not only returned by the method but also removed from the array. Call the method `deleteMaxNum()`.

2.3 The `deleteMaxNum()` method in Programming Project 2.2 suggests a way to create an array of numbers sorted by numeric value. Implement a sorting scheme that does not require modifying the `Array` class from Project 2.2, but only the code in `ArrayClient.py` (Listing 2-4).

2.4 Write a `removeDupes()` method for the `Array.py` program (Listing 2-3) that removes any duplicate entries in the array. That is, if three items with the value `'bar'` appear in the array, `removeDupes()` should remove two of them. Don't worry about maintaining the order of the items. One approach is to make a new, empty `list`, move items one at a time into it after first checking that they are not already in the new list, and then set the array to be the new list. Of course, the array size will be reduced if any duplicate entries exist. Write some tests to show it works on arrays with and without duplicate values.

2.5 Add a `merge()` method to the `OrderedRecordArray` class (Listing 2-8 and Listing 2-9) so that you can merge one ordered source array into that object's existing OrderedArray. The merge should occur only if both objects' key functions are identical. Your solution should create a new list big enough to hold the contents of the current (`self`) list and the merging array list. Write tests for your class implementation that creates two arrays, inserts some random numbers into them, invokes `merge()` to add the contents of one to the other, and displays the contents of the resulting array. The source arrays may hold different numbers of data items. Your

algorithm needs to compare the keys of the source arrays, picking the smallest one to copy to the destination. You also need to handle the situation when one source array exhausts its contents before the other. Note that, in Python, you can access a parameter's private attributes in a manner similar to using `self`. If the parameter `arr` is an `OrderedRecordArray` object, you can access its number of items as `arr.__nItems`.

2.6 Modify the `OrderedRecordArray` class (Listing 2-8 and Listing 2-9) so that requests to delete records that have duplicate keys correctly find the target records and delete them if present. Make sure you test the program thoroughly so that regardless of the number of records with duplicate keys or their order within the internal `list`, your modified version finds the matching record if it exists and leaves the `list` unchanged if it is not present.

2.7 Modify the `OrderedRecordArray` class (Listing 2-8 and Listing 2-9) so that it stores the maximum size of the array. When an insertion would go beyond the current maximum size, create a new list capable of holding more data and copy the existing list contents into it. The new size can be a fixed increment or a multiple of the current size. Test your new class by inserting data in a way that forces the object to expand the list several times and determine which is the best strategy—growing the list by adding a fixed amount of storage each time it fills up, or multiplying the list's storage by a fixed multiple each time it fills up.

CHAPTER 3
Simple Sorting

IN THIS CHAPTER

- How Would You Do It?
- Bubble Sort
- Selection Sort
- Insertion Sort
- Comparing the Simple Sorts

As soon as you create a significant database, you'll probably think of reasons to sort the records in various ways. You need to arrange names in alphabetical order, students by grade, customers by postal code, home sales by price, cities in order of population, countries by land mass, stars by magnitude, and so on.

Sorting data may also be a preliminary step to searching it. As you saw in Chapter 2, "Arrays," a binary search, which can be applied only to sorted data, is much faster than a linear search.

Because sorting is so important and potentially so time-consuming, it has been the subject of extensive research in computer science, and some very sophisticated methods have been developed. In this chapter we look at three of the simpler algorithms: the bubble sort, the selection sort, and the insertion sort. Each is demonstrated in a Visualization tool. In Chapter 6, "Recursion," and Chapter 7, "Advanced Sorting," we return to look at more sophisticated approaches including Shellsort and quicksort.

The techniques described in this chapter, while unsophisticated and comparatively slow, are nevertheless worth examining. Besides being easier to understand, they are actually better in some circumstances than the more sophisticated algorithms. The insertion sort, for example, is preferable to quicksort for small arrays and for almost-sorted arrays. In fact, an insertion sort is commonly used in the last stage of a quicksort implementation.

The sample programs in this chapter build on the array classes developed in the preceding chapter. The sorting algorithms are implemented as methods of the `Array` class.

Be sure to try out the algorithm visualizations provided with this chapter. They are very effective in explaining how the sorting algorithms work in combination with the descriptions and static pictures from the text.

How Would You Do It?

Imagine that a kids-league football team is lined up on the field, as shown in Figure 3-1. You want to arrange the players in order of increasing height (with the shortest player on the left) for the team picture. How would you go about this sorting process?

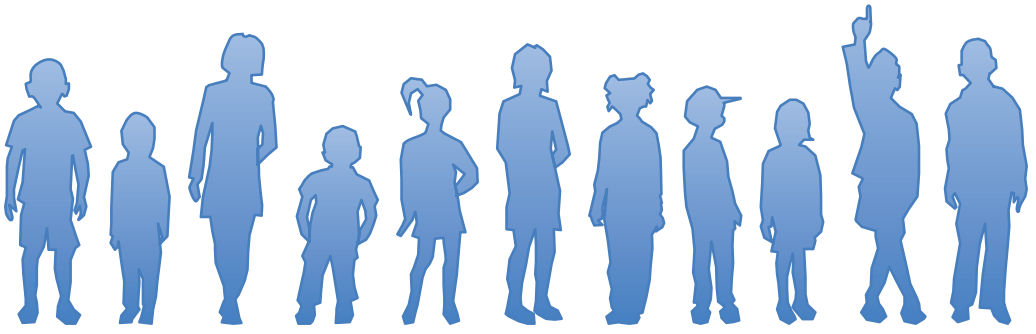


FIGURE 3-1 The unordered football team

As a human being, you have advantages over a computer program. You can see all the players at once, and you can pick out the tallest player almost instantly. You don't need to laboriously measure and compare everyone. The players are intelligent too and can self-sort pretty well. Also, the players don't need to occupy particular places. They can jostle each other, push each other a little to make room, and stand behind or in front of each other. After some ad hoc rearranging, you would have no trouble in lining up all the players, as shown in Figure 3-2.

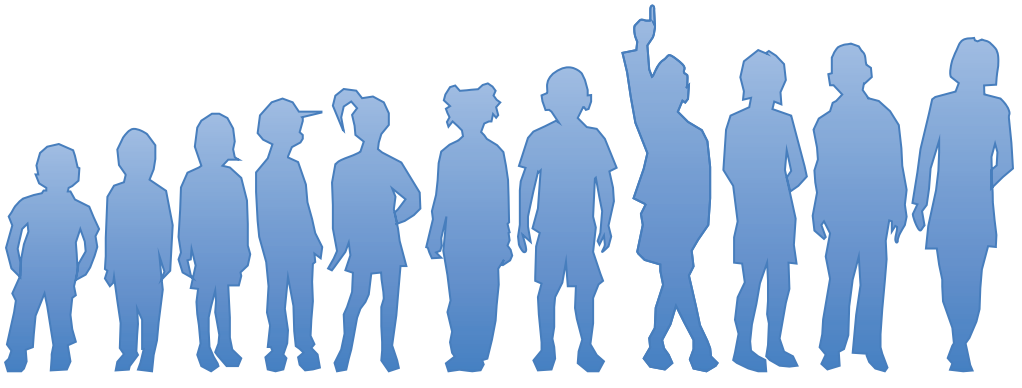


FIGURE 3-2 The football team ordered by height

A computer program isn't able to glance over the data in this way. It can compare only two players at one time because that's how the comparison operators work. These operators expect players to be in exact positions, moving only one or two players at a time. This tunnel vision on the part of algorithms is a recurring theme. Things may seem simple to us humans, but the computer can't see the big picture and must, therefore, concentrate on the details and follow strict rules.

The three algorithms in this chapter all involve two operations, executed over and over until the data is sorted:

A. Compare two items.

B. Swap two items, or copy one item.

Each algorithm, of course, handles the details in a different way.

Bubble Sort

The bubble sort is notoriously slow, but it's conceptually the simplest of the sorting algorithms and for that reason is a good beginning for our exploration of sorting techniques.

Bubble Sort on the Football Players

Imagine that you are wearing blinders or are near-sighted so that you can see only two of the football players at the same time, if they're next to each other and if you stand very close to them. Given these constraints (like the computer algorithm faces), how would you sort them? Let's assume there are N players, and the positions they're standing in are numbered from 0 on the left to N–1 on the right.

The bubble sort routine works like this: you start at the left end of the line and compare the two players in positions 0 and 1. If the one on the left (in 0) is taller, you swap them. If the one on the right is taller, you don't do anything. Then you move over one position and compare the players in positions 1 and 2. Again, if the one on the left is taller, you swap them. This sorting process is shown in Figure 3-3.

Here are the rules you're following:

1. Compare two players.
2. If the one on the left is taller, swap them.
3. Move one position right.

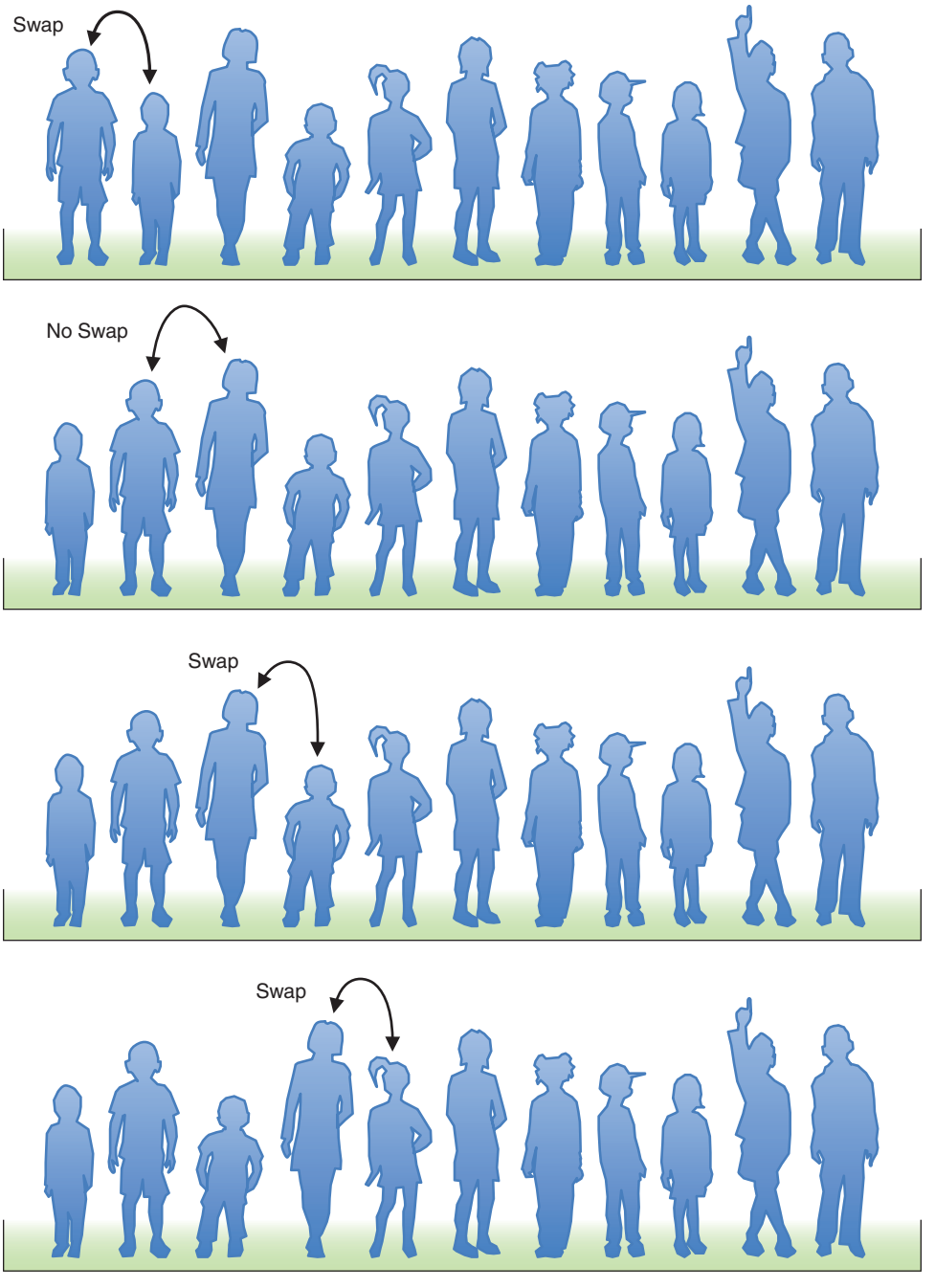


FIGURE 3-3 Bubble sort: swaps made during the first pass

You continue down the line this way until you reach the right end. You have by no means finished sorting the players, but you do know that the tallest player now is on the right. This must be true because, as soon as you encounter the tallest player, you end up swapping them every time you compare two players, until eventually the tallest reaches the right end of the line. This is why it's called the bubble sort: as the algorithm progresses, the biggest items "bubble up" to the top end of the array. Figure 3-4 shows the players at the end of the first pass.

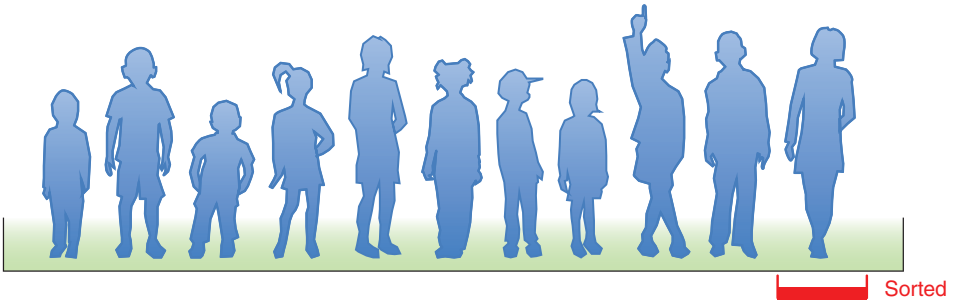


FIGURE 3-4 Bubble sort: the end of the first pass

After this first pass through all the players, you've made N–1 comparisons and somewhere between 0 and N–1 swaps, depending on the initial arrangement of the players. The player at the end of the line is sorted and won't be moved again.

Now you go back and start another pass from the left end of the line. Again, you go toward the right, comparing and swapping when appropriate. This time, however, you can stop one player short of the end of the line, at position N–2, because you know the last position, at N–1, already contains the tallest player. This rule could be stated as:

> When you reach the first sorted player, start over at the left end of the line.

You continue this process until all the players are in order. Describing this process is much harder than demonstrating it, so let's watch its work in the SimpleSorting Visualization tool.

The SimpleSorting Visualization Tool

Start the SimpleSorting Visualization tool (run `python3 SimpleSorting.py` as described in Appendix A). This program shows an array of values and provides multiple ways of sorting and manipulating it. Figure 3-5 shows the initial display.

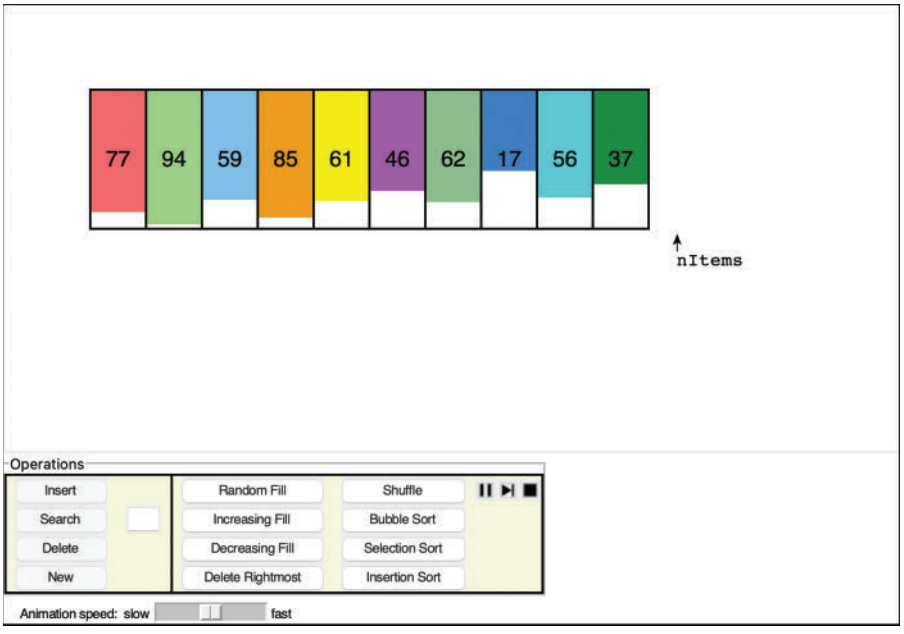


FIGURE 3-5 The SimpleSorting Visualization tool

The Insert, Search, Delete, New, Random Fill, and Delete Rightmost buttons function like those we saw in the Array Visualization tool used in Chapter 2. There are new buttons on the right relevant to sorting. Select the Bubble Sort button to start sorting the items. As with the other visualization tools, you can slow down or speed up the animation using the slider on the bottom. You can pause and resume playing the animation with the play, pause, and skip buttons (▶, ❚❚, ▶|).

When the bubble sort begins, it adds two arrows labeled "inner" and "last" next to the array. These point to where the algorithm is doing its work. The inner arrow walks from the leftmost cell to the right, swapping item pairs when it finds a taller one to the left of a shorter one, just like the players on the team. It stops when it reaches the last arrow, which marks the first item that is in the final sorted order.

As each maximum item "bubbles" up to the far right, the last arrow moves to the left by one cell, to keep track of which items are sorted. When the last arrow moves all the way to the first cell on the left, the array is fully sorted. To run a new sort operation, select Shuffle, and then Bubble Sort again.

Try going through the bubble sort process using the step (▶|) or play/pause (▶/❚❚) buttons. At each step, think about what should happen next. Will a swap occur? How will the two arrows change?

Try making a new array with many cells (30+) and filling it with random items. The cells will become too narrow to show the numbers of each item. The heights of the colored

rectangles indicate their number, but it's harder to see which one is taller when they are nearly equal in height. Can you still predict what will happen at each step?

The visualization tool also has buttons to fill empty cells with keys in increasing order and decreasing order. It's interesting to see what happens when sorting algorithms encounter data that is already sorted or reverse sorted. You can also scramble the data with the Shuffle button to try running a new sort.

NOTE

The Stop (■) button stops an animation and allows you to start other operations. The array contents, however, may be different than what they were at the start of the animation. There may be missing items or extra copies of items depending on when the operation was interrupted. In this way, the visualization mimics what happens in computer memory if something stops the execution.

Python Code for a Bubble Sort

The bubble sort algorithm is pretty straightforward to explain, but we need to look at how to write the program. Listing 3-1 shows the `bubbleSort()` method of an `Array` class. It is almost the same as the `Array` class introduced in Chapter 2, and the full `SortArray` module is shown later in this chapter in Listing 3-4.

LISTING 3-1 The `bubbleSort()` method of the `Array` class

```
def bubbleSort(self):                          # Sort comparing adjacent vals
   for last in range(self.__nItems-1, 0, -1):  # and bubble up
      for inner in range(last):                # inner loop goes up to last
         if self.__a[inner] > self.__a[inner+1]:  # If item less
            self.swap(inner, inner+1)          # than adjacent item, swap
```

In the `Array` class, each element of the array is assumed to be a simple value that can be compared with any of the other values for the purposes of ordering them. We reintroduce a key function to handle ordering records later.

The `bubbleSort()` method has two loops. The inner loop handles stepping through the array and swapping elements that are out of order. The outer loop handles the decreasing length of the unsorted part of the array. The outer loop sets the `last` variable to point initially at the last element of the array. That's done by using Python's `range` function to start at `__nItems-1` and step down to 1 in increments of –1. The inner loop starts each time with the `inner` variable set to 0 and increments by 1 to reach `last-1`. The elements at indices greater than `last` are always completely sorted. After each pass of the inner loop, the `last` variable can be reduced by 1 because the maximum element between 0 and `last` bubbled up to the `last` position.

The inner loop body performs the test to compare the elements at the `inner` and `inner+1` indices. If the value at `inner` is larger than the one to its right, you have to swap the two array elements (we'll see how `swap()` works when we look at all the `SortArray` code in a later section).

Invariants

In many algorithms, there are conditions that remain unchanged as the algorithm proceeds. These conditions are called **invariants**. Recognizing invariants can be useful in understanding the algorithm. In certain situations, they may help in debugging; you can repeatedly check that the invariant is true, and signal an error if it isn't.

In the `bubbleSort()` method, the invariant is that the array elements to the right of `last` are sorted. This remains true throughout the running of the algorithm. On the first pass, nothing has been sorted yet, and there are no items to the right of `last` because it starts on the rightmost element.

Efficiency of the Bubble Sort

If you use bubble sort on an array with 11 cells, the `inner` arrow makes 10 comparisons on the first pass, nine on the second, and so on, down to one comparison on the last pass. For 11 items, this is

$$10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = 55$$

In general, where N is the number of items in the array, there are N–1 comparisons on the first pass, N–2 on the second, and so on. The formula for the sum of such a series is

$$(N–1) + (N–2) + (N–3) + ... + 1 = N\times(N–1)/2$$

$N\times(N–1)/2$ is 55 ($11\times10/2$) when N is 11.

Thus, the algorithm makes about $N^2/2$ comparisons (ignoring the –1, which doesn't make much difference, especially if N is large).

There are fewer swaps than there are comparisons because two bars are swapped only if they need to be. If the data is random, a swap is necessary about half the time, so there would be about $N^2/4$ swaps. (In the worst case, with the initial data inversely sorted, a swap is necessary with every comparison.)

Both swaps and comparisons are proportional to N^2. Because constants don't count in Big O notation, you can ignore the 2 and the 4 in the divisor and say that the bubble sort runs in $O(N^2)$ time. This is slow, as you can verify by running the Bubble Sort in the SimpleSorting Visualization tool with arrays of 20+ cells.

Whenever you see one loop nested within another, such as those in the bubble sort and the other sorting algorithms in this chapter, you can suspect that an algorithm runs in $O(N^2)$ time. The outer loop executes N times, and the inner loop executes N (or perhaps N divided by some constant) times for each cycle of the outer loop. This means it's doing something approximately $N\times N$ or N^2 times.

Selection Sort

How can you improve the efficiency of the sorting operation? You know that $O(N^2)$ is pretty bad. Can you get to O(N) or maybe even O(log N)? We look next at a method called selection sort, which reduces the number of swaps. That could be significant when sorting large records by a key. Copying entire records rather than just integers could take much more time than comparing two keys. In Python as in other languages, the computer probably just copies pointers or references to record objects rather than the entire record, but it's important to understand which operations are happening the most times.

Selection Sort on the Football Players

Let's consider the football players again. In the selection sort, you don't compare players standing next to each other, but you still can look at only two players and compare their heights. In this algorithm, you need to remember a certain player's height—perhaps by using a measuring stick and writing the number in a notebook. A ball also comes in handy as a marker.

A Brief Description

What's involved in the selection sort is making a pass through all the players and picking (or *selecting*, hence the name of the sort) the shortest one. This shortest player is then swapped with the player on the left end of the line, at position 0. Now the leftmost player is sorted and doesn't need to be moved again. Notice that in this algorithm the sorted players accumulate on the left (lower indices), whereas in the bubble sort they accumulated on the right.

The next time you pass down the row of players, you start at position 1, and, finding the minimum, swap with position 1. This process continues until all the players are sorted.

A More Detailed Description

Let's start at the left end of the line of players. Record the leftmost player's height in your notebook and put a ball on the ground in front of this person. Then compare the height of the next player to the right with the height in your notebook. If this player is shorter, cross out the height of the first player and record the second player's height instead. Also move the ball, placing it in front of this new "shortest" (for the time being) player. Continue down the row, comparing each player with the minimum. Change the minimum value in your notebook and move the ball whenever you find a shorter player. When you reach the end of the line, the ball will be in front of the shortest player. For example, the top of Figure 3-6 shows the ball being placed in front of the fourth player from the left.

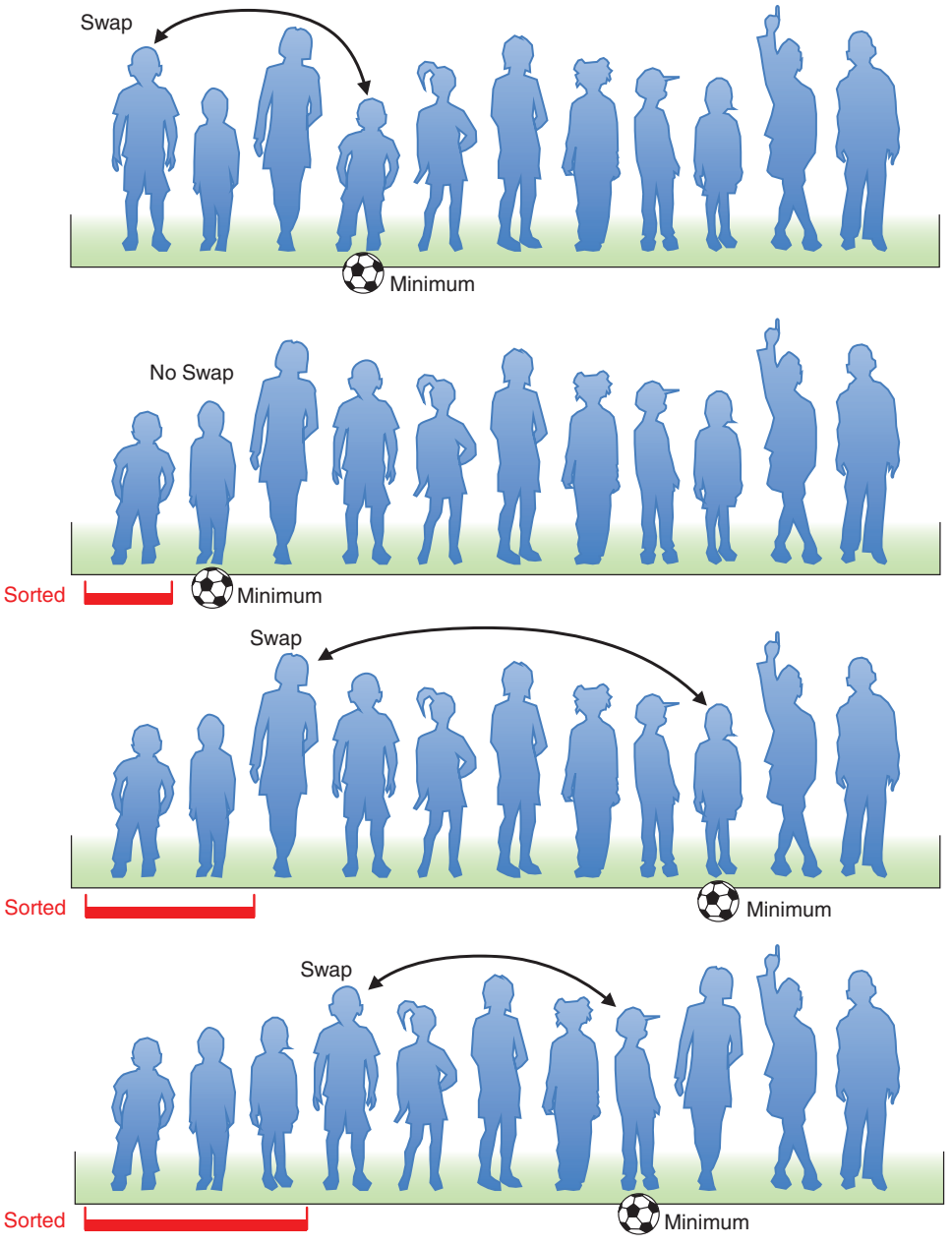


FIGURE 3-6 Selection sort on football players

Swap this shortest player with the player on the left end of the line. You've now sorted one player. You've made N–1 comparisons, but only one swap.

On the next pass, you do exactly the same thing, except that you can skip the player on the left because this player has already been sorted. Thus, the algorithm starts the second

pass at position 1, instead of 0. In the case of Figure 3-6, this second pass finds the shortest player at position 1, so no swap is needed. With each succeeding pass, one more player is sorted and placed on the left, and one fewer player needs to be considered when finding the new minimum. The bottom of Figure 3-6 shows how this sort looks after the first three passes.

The Selection Sort in the SimpleSorting Visualization Tool

To see how the selection sort looks in action, try out the Selection Sort button in the SimpleSorting Visualization tool. Use the New and Random Fill buttons to create a new array of 11 randomly arranged items. When you select the Selection Sort button, it adds arrows to the display that look something like those shown in Figure 3-7. The arrow labeled "outer" starts on the left along with a second arrow labeled "min." The "inner" arrow starts one cell to the right and marks where the first comparison happens.

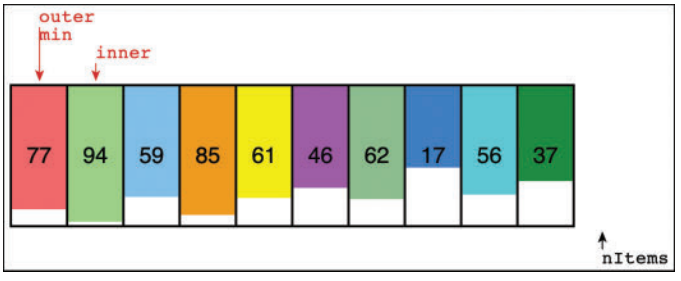


FIGURE 3-7 The start of a selection sort in the SimpleSorting Visualization tool

As inner moves right, each cell it points at is compared with the one at min. If the value at min is lower, then the min arrow is moved to where inner is, just as you moved the ball to lie in front of the shortest player.

When inner reaches the right end, the items at the outer and min arrows are swapped. That means that one more cell has been sorted, so outer can be moved one cell to the right. The next pass starts with min pointing to the new position of outer and inner one cell to their right.

The outer arrow marks the position of the first unsorted item. All items from outer to the right end are unsorted. Cells to the left of outer are fully sorted. The sorting continues until outer reaches the last cell on the right. At that point, the next possible comparison would have to be past the last cell, so the sorting must stop.

Python Code for Selection Sort

As with the bubble sort, implementing a selection sort requires two nested loops. The Python method is shown in Listing 3-2.

LISTING 3-2 The `selectionSort()` Method of the `Array` Class

```
def selectionSort(self):                    # Sort by selecting min and
   for outer in range(self.__nItems-1):     # swapping min to leftmost
      min = outer                           # Assume min is leftmost
      for inner in range(outer+1, self.__nItems): # Hunt to right
         if self.__a[inner] < self.__a[min]: # If we find new min,
            min = inner                     # update the min index

      # __a[min] is smallest among __a[outer]...__a[__nItems-1]
      self.swap(outer, min)                 # Swap leftmost and min
```

The outer loop starts by setting the `outer` variable to point at the beginning of the array (index 0) and proceeds right toward higher indices. The minimum valued item is assumed to be the first one by setting `min` to be the same as `outer`. The inner loop, with loop variable `inner`, begins at `outer+1` and likewise proceeds to the right.

At each new position of `inner`, the elements `__a[inner]` and `__a[min]` are compared. If `__a[inner]` is smaller, then `min` is given the value of `inner`. At the end of the inner loop, `min` points to the minimum value, and the array elements pointed to by `outer` and `min` are swapped.

Invariant

In the `selectionSort()` method, the array elements with indices less than `outer` are always sorted. This condition holds when `outer` reaches `__nItems-1`, which means all but one item are sorted. At that point, you could try to run the inner loop one more time, but it couldn't change the `min` index from `outer` because there is only one item left in the unsorted range. After doing nothing in the inner loop, it would then swap the item at `outer` with itself—another do nothing operation—and then `outer` would be incremented to `__nItems`. Because that entire last pass does nothing, you can end when `outer` reaches `__nItems-1`.

Efficiency of the Selection Sort

The comparisons performed by the selection sort are N×(N–1)/2, the same as in the bubble sort. For 11 data items, this is 55 comparisons. That process requires exactly 10 swaps. The number of swaps is always N–1 because the outer loop does one for each iteration. That compares favorably with the approximately $N^2/4$ swaps needed in bubble sort, which works out to about 30 for 11 items. With 100 items, the selection sort makes 4,950 comparisons and 99 swaps. For large values of N, the comparison times dominate, so you would have to say that the selection sort runs in $O(N^2)$ time, just as the bubble sort did. It is, however, unquestionably faster because there are O(N) instead of $O(N^2)$ swaps. Note that the number of swaps in bubble sort is dependent on the initial ordering of the array

elements, so in some special cases, they could number fewer than those made by the selection sort.

Insertion Sort

In most cases, the insertion sort is the best of the elementary sorts described in this chapter. It still executes in $O(N^2)$ time, but it's about twice as fast as the bubble sort and somewhat faster than the selection sort in normal situations. It's also not too complex, although it's slightly more involved than the bubble and selection sorts. It's often used as the final stage of more sophisticated sorts, such as quicksort.

Insertion Sort on the Football Players

To begin the insertion sort, start with the football players lined up in random order. (They wanted to play a game, but clearly they've got to wait until the picture can be taken.) It's easier to think about the insertion sort if you begin in the middle of the process, when part of the team is sorted.

Partial Sorting

You can use your handy ball to mark a place in the middle of the line. The players to the left of this marker are **partially sorted**. This means that they are sorted among themselves; each one is taller than the person to their left. The players, however, aren't necessarily in their final positions because they may still need to be moved when previously unsorted players are inserted between them.

Note that partial sorting did not take place in the bubble sort and selection sort. In these algorithms, a group of data items was completely sorted into their final positions at any given time; in the insertion sort, one group of items is only partially sorted.

The Marked Player

The player where the marker ball is, whom we call the "marked" player, and all the players to the right are as yet unsorted. Figure 3-8 shows the process. At the top, the players are unsorted. The next row in the figure shows the situation three steps later. The marker ball is put in front of a player who has stepped forward.

What you're going to do is insert the marked player in the appropriate place in the (partially) sorted group. To do this, you need to shift some of the sorted players to the right to make room. To provide a space for this shift, the marked player is taken out of line. (In the program, this data item is stored in a temporary variable.) This step is shown in the second row of Figure 3-8.

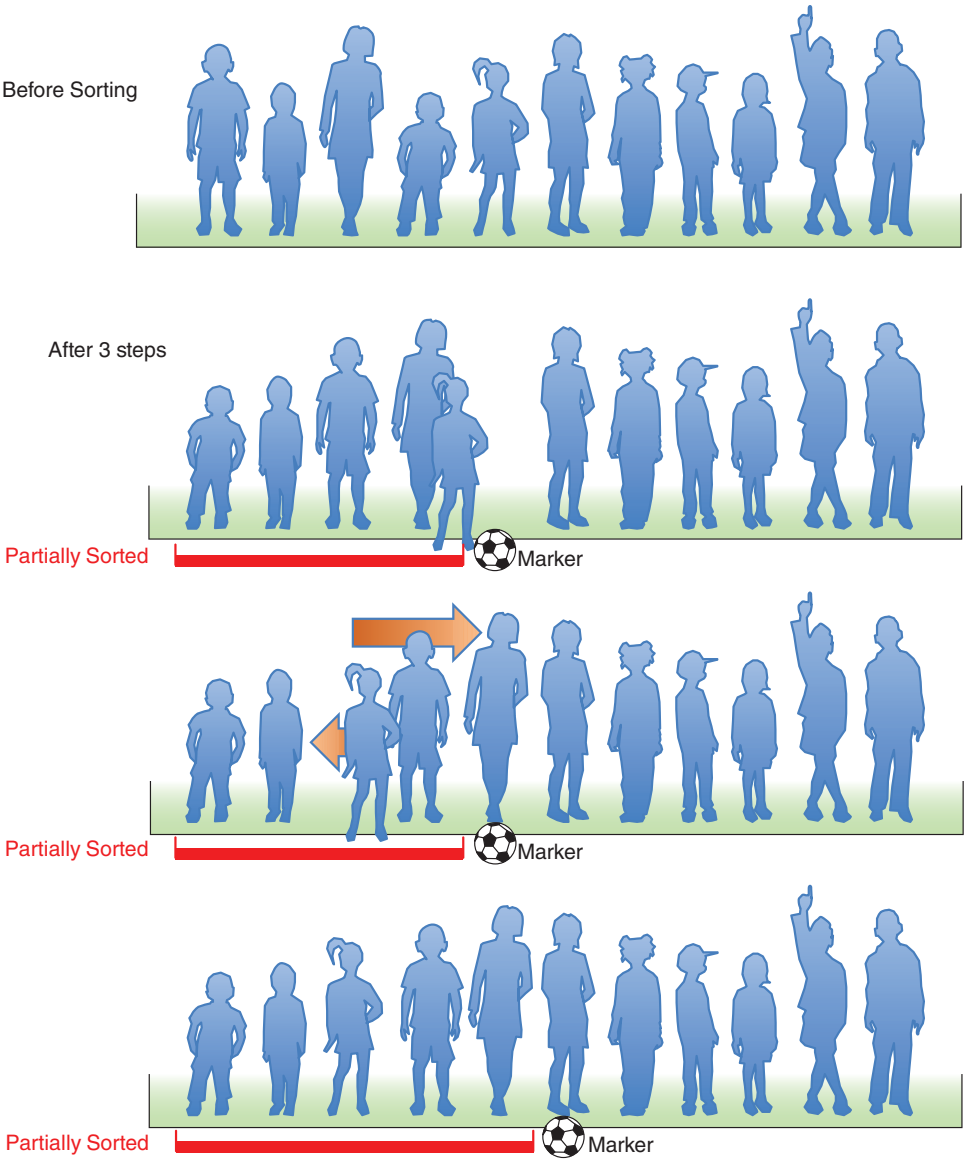


FIGURE 3-8 The insertion sort on football players

Now you shift the sorted players to make room. The tallest sorted player moves into the marked player's spot, the next-tallest player into the tallest player's spot, and so on as shown by the arrows in the third row of the figure.

When does this shifting process stop? Imagine that you and the marked player are walking down the line to the left. At each position you shift another player to the right, but you also compare the marked player with the player about to be shifted. The shifting process stops when you've shifted the last player that's taller than the marked player. The last shift opens up the space where the marked player, when inserted, will be in sorted order. This step is shown in the bottom row of Figure 3-8.

Now the partially sorted group is one player bigger, and the unsorted group is one player smaller. The marker ball is moved one space to the right, so it's again in front of the leftmost unsorted player. This process is repeated until all the unsorted players have been inserted (hence the name *insertion* sort) into the appropriate place in the partially sorted group.

The Insertion Sort in the SimpleSorting Visualization Tool

Return to the SimpleSorting Visualization tool and create another 11-cell array of random values. Then select the Insertion Sort button to start the sorting process. As in the other sort operations, the algorithm puts up two arrows: one for the outer and one for the inner loops. The arrow labeled "outer" is the equivalent of the marker ball in sorting the players.

In the example shown in Figure 3-9, the outer arrow is pointing at the fifth cell in the array. It already copied item 61 to the `temp` variable below the array. It has also copied item 94 into the fifth cell and is starting to copy item 85 into the fourth. The four cells to the left of the outer arrow are partially sorted; the cells to its right are unsorted. Even while it decides where the marked player should go and there are extra copies of one of them, the left-hand cells remain partially sorted. This matches what happens in the computer memory when you copy the array item to a temporary variable.

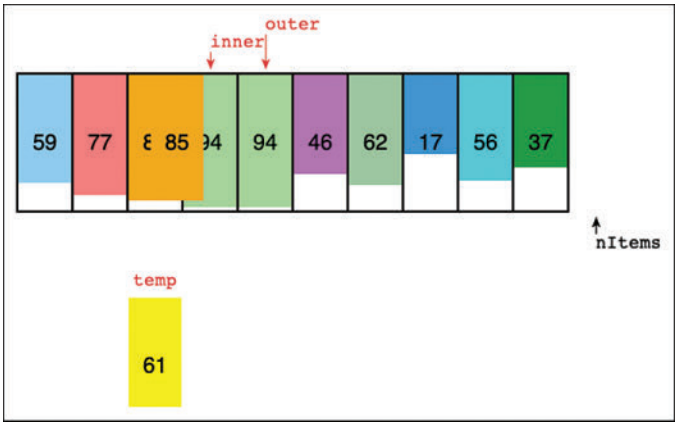


FIGURE 3-9 In the middle of an insertion sort in the SimpleSorting Visualization tool

The inner arrow starts at the same cell as the outer arrow and moves to the left, shifting items taller than the marked `temp` item to the right. In Figure 3-9, item 85 is being copied to where inner points because it is larger than item 61. After that copy is completed, inner moves one cell to the left and finds that item 77 needs to be moved too. On the next step for inner, it finds item 59 to its left. That item is smaller than 61, and the marked item is copied from temp back to the inner cell, where item 77 used to be.

Try creating a larger array filled with random values. Run the insertion sort and watch the updates to the arrows and items. Everything to the left of the outer arrow remains sorted by height. The inner arrow moves to the left, shifting items until it finds the right cell to place the item that was marked (copied from the outer cell).

Eventually, the outer arrow arrives at the right end of the array. The inner arrow moves to the left from that right edge and stops wherever it is appropriate to insert the last item. It might finish anywhere in the array. That's a little different than the bubble and selection sorts where the outer and inner loop variables finish together.

Notice that, occasionally, the item copied from outer to temp is copied right back to where it started. That occurs when outer happens to point at an item larger than every item to its left. A similar thing happens in the selection sort when the outer arrow points at an item smaller than every item to its right; it ends up doing a swap with itself. Doing these extra data moves might seem inefficient, but adding tests for these conditions could add its own inefficiencies. An experiment at the end of the chapter asks you to explore when it might make sense to do so.

Python Code for Insertion Sort

Let's now look at the `insertionSort()` method for the `Array` class as shown in Listing 3-3. You still need two loops, but the inner loop hunts for the insertion point using a `while` loop this time.

LISTING 3-3 The `insertionSort()` method of the `Array` class

```
   def insertionSort(self):                    # Sort by repeated inserts
      for outer in range(1, self.__nItems):    # Mark one element
         temp = self.__a[outer]                # Store marked elem in temp
         inner = outer                         # Inner loop starts at mark
         while inner > 0 and temp < self.__a[inner-1]: # If marked
            self.__a[inner] = self.__a[inner-1] # elem smaller, then
            inner -= 1                         # shift elem to right
         self.__a[inner] = temp                # Move marked elem to 'hole'
```

In the outer `for` loop, `outer` starts at 1 and moves right. It marks the leftmost unsorted array element. In the inner `while` loop, `inner` starts at `outer` and moves left to `inner-1`, until either `temp` is smaller than the array element there, or it can't go left any further. Each pass through the `while` loop shifts another sorted item one space right.

Invariants in the Insertion Sort

The data items with smaller indices than `outer` are partially sorted. This is true even at the beginning when `outer` is 1 because the single item in cell 0 forms a one-element sequence that is always sorted. As items shift, there are multiple copies of one of them, but they all remain in partially sorted order.

Efficiency of the Insertion Sort

How many comparisons and copies does this algorithm require? On the first pass, it compares a maximum of one item. On the second pass, it's a maximum of two items, and so on, up to a maximum of N–1 comparisons on the last pass. (We ignore the check for `inner > 0` and count only the intra-element comparisons.) This adds up to

$$1 + 2 + 3 + \ldots + N–1 = N\times(N–1)/2$$

On each pass, however, the number of items actually compared before the insertion point is found is, on average, half of the partially sorted items, so you can divide by 2, which gives

$$N\times(N–1)/4$$

The number of copies is approximately the same as the number of comparisons because it makes one copy for each comparison but the last. Copying one item is less time-consuming than a swap of two items, so for random data this algorithm runs twice as fast as the bubble sort and faster than the selection sort, on average.

In any case, like the other sort routines in this chapter, the insertion sort runs in $O(N^2)$ time for random data.

For data that is already sorted or almost sorted, the insertion sort does much better. When data is in order, the condition in the `while` loop is never true, so it becomes a simple statement in the outer loop, which executes N–1 times. In this case, the algorithm runs in O(N) time. If the data is almost sorted, the insertion sort runs in almost O(N) time, which makes it a simple and efficient way to order an array that is only slightly out of order.

For data arranged in inverse sorted order, however, every possible comparison and shift is carried out, so the insertion sort runs no faster than the bubble sort. Try some experiments with the SimpleSorting Visualization tool. Sort an 8+ element array with the selection sort and then try sorting the result with the selection sort again. Even though the second attempt doesn't make any "real" swaps, it takes the same number of steps to go through the process. Then shuffle the array and sort it twice using the insertion sort. The second attempt at the insertion sort is significantly shorter because the inner loop ends after one comparison.

Python Code for Sorting Arrays

Now let's combine the three sorting algorithms into a single object class to compare them. Starting from the `Array` class introduced in Chapter 2, we add the new methods for

swapping and sorting plus the `__str__()` method for displaying the contents of arrays as a string and put them in a module called `SortArray.py`, as shown in Listing 3-4.

LISTING 3-4 The `SortArray.py` module

```
# Implement a sortable Array data structure

class Array(object):
   def __init__(self, initialSize):    # Constructor
      self.__a = [None] * initialSize  # The array stored as a list
      self.__nItems = 0                # No items in array initially

   def __len__(self):                  # Special def for len() func
      return self.__nItems             # Return number of items

   def get(self, n):                   # Return the value at index n
      if 0 <= n and n < self.__nItems: # Check if n is in bounds, and
         return self.__a[n]            # only return item if in bounds

   def set(self, n, value):            # Set the value at index n
      if 0 <= n and n < self.__nItems: # Check if n is in bounds, and
         self.__a[n] = value           # only set item if in bounds

   def swap(self, j, k):               # Swap the values at 2 indices
      if (0 <= j and j < self.__nItems and # Check if indices are in
          0 <= k and k < self.__nItems): # bounds, before processing
         self.__a[j], self.__a[k] = self.__a[k], self.__a[j]

   def insert(self, item):             # Insert item at end
      if self.__nItems >= len(self.__a):     # If array is full,
         raise Exception("Array overflow")      # raise exception
      self.__a[self.__nItems] = item   # Item goes at current end
      self.__nItems += 1               # Increment number of items

   def find(self, item):               # Find index for item
      for j in range(self.__nItems):   # Among current items
         if self.__a[j] == item:       # If found,
            return j                   # then return index to element
      return -1                        # Not found -> return -1

   def search(self, item):             # Search for item
      return self.get(self.find(item)) # and return item if found

   def delete(self, item):             # Delete first occurrence
      for j in range(self.__nItems):   # of an item
         if self.__a[j] == item:       # Found item
```

```
            self.__nItems -= 1          # One fewer at end
            for k in range(j, self.__nItems):  # Move items from
               self.__a[k] = self.__a[k+1]     # right over 1
            return True                 # Return success flag

      return False     # Made it here, so couldn't find the item

   def traverse(self, function=print): # Traverse all items
      for j in range(self.__nItems):   # and apply a function
         function(self.__a[j])

   def __str__(self):                  # Special def for str() func
      ans = "["                        # Surround with square brackets
      for i in range(self.__nItems):   # Loop through items
         if len(ans) > 1:              # Except next to left bracket,
            ans += ", "                # separate items with comma
         ans += str(self.__a[i])       # Add string form of item
      ans += "]"                       # Close with right bracket
      return ans

   def bubbleSort(self):               # Sort comparing adjacent vals
      for last in range(self.__nItems-1, 0, -1):  # and bubble up
         for inner in range(last):     # inner loop goes up to last
            if self.__a[inner] > self.__a[inner+1]:  # If elem less
               self.swap(inner, inner+1) # than adjacent value, swap

   def selectionSort(self):            # Sort by selecting min and
      for outer in range(self.__nItems-1): # swapping min to leftmost
         min = outer                   # Assume min is leftmost
         for inner in range(outer+1, self.__nItems): # Hunt to right
            if self.__a[inner] < self.__a[min]: # If we find new min,
               min = inner             # update the min index

         # __a[min] is smallest among __a[outer]...__a[__nItems-1]
         self.swap(outer, min)         # Swap leftmost and min

   def insertionSort(self):            # Sort by repeated inserts
      for outer in range(1, self.__nItems): # Mark one element
         temp = self.__a[outer]        # Store marked elem in temp
         inner = outer                 # Inner loop starts at mark
         while inner > 0 and temp < self.__a[inner-1]: # If marked
            self.__a[inner] = self.__a[inner-1] # elem smaller, then
            inner -= 1                 # shift elem to right
         self.__a[inner] = temp        # Move marked elem to 'hole'
```

The `swap()` method swaps the values in two cells of the array. It ensures that swaps happen only with items that have been inserted in the array and not with allocated but uninitialized cells. Those tests may not be necessary with the sorting methods in this module that already ensure proper indices, but they are a good idea for a general-purpose routine.

We use a separate client program, `SortArrayClient.py`, to test this new module and compare the performance of the different sorting methods. This program uses some other Python modules and features to help in this process and is shown in Listing 3-5.

LISTING 3-5 The `SortArrayClient.py` program

```
from SortArray import *
import random
import timeit

def initArray(size=100, maxValue=100, seed=3.14159):
    """Create an Array of the specified size with a fixed sequence of
       'random' elements"""
    arr = Array(size)                          # Create the Array object
    random.seed(seed)                          # Set random number generator
    for i in range(size):                      # to known state, then loop
        arr.insert(random.randrange(maxValue)) # Insert random numbers
    return arr                                 # Return the filled Array

arr = initArray()
print("Array containing", len(arr), "items:\n", arr)

for test in ['initArray().bubbleSort()',
             'initArray().selectionSort()',
             'initArray().insertionSort()']:
    elapsed = timeit.timeit(test, number=100, globals=globals())
    print(test, "took", elapsed, "seconds", flush=True)

arr.insertionSort()
print('Sorted array contains:\n', arr)
```

The `SortArrayClient.py` program starts by importing the `SortArray` module to test the sorting methods, and the `random` and `timeit` modules to assist. The `random` module provides pseudo-random number generators that can be used to make test data. The `timeit` module provides a convenient way to measure the execution times of Python code.

The test program first defines a function, `initArray()`, that generates an unsorted array for testing. Right after the parameter list, it has a documentation string that helps explain its purpose, creating arrays of a fixed size filled with a sequence of random numbers. Because we'd like to test the different sort operations on the exact same sequence of

elements, we want `initArray()` to return a fresh copy of the same array every time it runs, but conforming to the parameters it is given. The function first creates an `Array` of the desired `size`. Next, it initializes the `random` module to a known state by setting the `seed` value. This means that pseudo-random functions produce the same sequence of numbers when called in the same order.

Inside the following `for` loop, the `random.randrange(maxValue)` function generates integers in the range [0, `maxValue`) that are inserted into the array (the math notation *[a, b)* represents a range of numbers that includes *a* but excludes *b*). Then the filled array is returned.

The test program uses the `initArray()` function to generate an array that is stored in the `arr` variable. After printing the initial contents of `arr`, a `for` loop is used to step through the three sort operations to be timed. The loop variable, `test`, is bound to a string on each pass of the loop. The string has the Python expression whose execution time is to be measured. Inside the loop body, the `timeit.timeit()` function call measures the elapsed time of running the `test` expression. It runs the expression a designated `number` of times and returns the total elapsed time in seconds. Running it many times helps deal with the variations in running times due to other operations happening on the computer executing the Python interpreter. The last argument to `timeit.timeit()` is `globals`, which is needed so that the interpreter has all the definitions that have been loaded so far, including `SortArray` and `initArray`. The interpreter uses those definitions when evaluating the Python expression in the `test` variable.

The result of `timeit.timeit()` is stored in the `elapsed` variable and then printed out with `flush=True` so that it doesn't wait until the output buffer is full or input needs to be read before printing. After all three test times are printed, it performs one more sort on the original `arr` array and prints its contents. The results look like this:

```
$ python3 SortArrayClient.py
Array containing 100 items:
 [77, 94, 59, 85, 61, 46, 62, 17, 56, 37, 18, 45, 76, 21, 91, 7, 96, 50, 31, 69, 80,
69, 56, 60, 26, 25, 1, 2, 67, 46, 99, 57, 32, 26, 98, 51, 77, 34, 20, 81, 22, 40,
28, 23, 69, 39, 23, 6, 46, 1, 96, 51, 71, 61, 2, 34, 1, 55, 78, 91, 69, 23, 2, 8,
3, 78, 31, 25, 26, 73, 28, 88, 88, 38, 22, 97, 9, 18, 18, 66, 47, 16, 82, 9, 56, 45,
15, 76, 85, 52, 86, 5, 28, 67, 34, 20, 6, 33, 83, 68]
initArray().bubbleSort() took 0.25465869531035423 seconds
initArray().selectionSort() took 0.11350362841039896 seconds
initArray().insertionSort() took 0.12348053976893425 seconds
Sorted array contains:
 [1, 1, 1, 2, 2, 2, 3, 5, 6, 6, 7, 8, 9, 9, 15, 16, 17, 18, 18, 18, 20, 20, 21, 22,
22, 23, 23, 23, 25, 25, 26, 26, 26, 28, 28, 28, 31, 31, 32, 33, 34, 34, 34, 37, 38,
39, 40, 45, 45, 46, 46, 46, 47, 50, 51, 51, 52, 55, 56, 56, 56, 57, 59, 60, 61, 61,
62, 66, 67, 67, 68, 69, 69, 69, 69, 71, 73, 76, 76, 77, 77, 78, 78, 80, 81, 82, 83,
85, 85, 86, 88, 88, 91, 91, 94, 96, 96, 97, 98, 99]
```

Looking at the output, you can see that the `randrange()` function provided a broad variety of values for the array in a random order. The results of the timing of the sort tests show the bubble sort taking at least twice as much time as the selection and insertion sorts do. The final sorted version of the array confirms that sorting works and shows that there are many duplicate elements in the array.

Stability

Sometimes it matters what happens to data items that have equal keys when sorting. The `SortArrayClient.py` test stored only integers in the array, and equal integers are pretty much indistinguishable. If the array contained complex records, however, it could be very important how records with the same key are sorted. For example, you may have employee records arranged alphabetically by family names. (That is, the family names were used as key values in the sort.) Let's say you want to sort the data by postal code too, but you want all the items with the same postal code to continue to be sorted by family names. This is called a **secondary sort key**. You want the sorting algorithm to shift only what needs to be sorted by the current key and leave everything else in its order after previous sorts using other keys. Some sorting algorithms retain this secondary ordering; they're said to be **stable**. Stable sorting methods also minimize the number of swaps or copy operations.

Some of the algorithms in this chapter are stable. We have included an exercise at the end of the chapter for you to decide which ones are. The answer is not obvious from the output of their test programs, especially on simple integers. You need to review the algorithms to see that swaps and copies only happen on items that need to be moved to be in the right position for the final order. They should also move items with equal keys the minimum number of array cells necessary so that they remain in the same relative order in the final arrangement.

Comparing the Simple Sorts

There's probably no point in using the bubble sort, unless you don't have your algorithm book handy. The bubble sort is so simple that you can write it from memory. Even so, it's practical only if the amount of data is small. (For a discussion of what "small" means and what trade-offs such decisions entail, see Chapter 16, "What to Use and Why.")

Table 3-1 summarizes the time efficiency of each of the sorting methods. We looked at what the algorithms would do when presented with elements whose keys are in random order to determine what they would do in the average case. We also considered what would happen in the worst case when the keys were in some order that would maximize the number of operations. The table includes both the Big O notation followed by the detailed value in square brackets [ ]. The Big O notation is what matters most; it gives the broad classification of the algorithms and tells what will happen for very large N. Typically, we are most interested in the average case behavior because we cannot anticipate what data they will encounter, but sometimes the worst case is the driving factor. For

example, if the sort must be able to complete within a specific amount of time for a particular size of data, we can use the best of the worst-case values to choose the algorithm that will guarantee that completion time.

Table 3-1 Time Order of the Sort Methods

Algorithm	Comparisons		Swaps or Copies	
	Average case	Worst case	Average case	Worst case
Bubble sort	$O(N^2)$ $[N^2/2]$	$O(N^2)$ $[N^2/2]$	$O(N^2)$ $[N^2/4]$	$O(N^2)$ $[N^2/2]$
Selection sort	$O(N^2)$ $[N^2/2]$	$O(N^2)$ $[N^2/2]$	$O(N)$ $[N–1]$	$O(N)$ $[N–1]$
Insertion sort	$O(N^2)$ $[N^2/4]$	$O(N^2)$ $[N^2/2]$	$O(N^2)$ $[N^2/4]$	$O(N^2)$ $[N^2/2]$

The detailed values shown in the table can be somewhat useful in narrow circumstances. For example, the relative number of comparisons in the average case between the bubble sort and the insertion sort corresponds well with the relative time measurements in the test program. That would also predict that the selection sort average case should be closer to the bubble sort, but the one measurement above doesn't agree with that. Perhaps that means that the swaps or copies were a bigger factor on running time with N around 100 on the test platform.

Overall, the selection sort minimizes the number of swaps, but the number of comparisons is still high. This sort might be useful when the amount of data is small and swapping data items is very time-consuming relative to comparing them. In real-world scenarios, there can be situations where keeping the swaps to a minimum is more important than the number of comparisons. For example, in moving patients between hospital rooms (in response to some other disaster), it could be critical to displace as few patients as possible.

The insertion sort is the most versatile of the three sorting methods and is the best bet in most situations, assuming the amount of data is small, or the data is almost sorted. For larger amounts of data, quicksort and Timsort are generally considered the fastest approach; we examine them in Chapter 7.

We've compared the sorting algorithms in terms of speed. Another consideration for any algorithm is how much memory space it needs. All three of the algorithms in this chapter carry out their sort **in place**, meaning that they use the input array as the output array and very little extra memory is required. All the sorts require an extra variable to store an item temporarily while it's being swapped or marked. Because that item can be quite large, you need to consider it, but the local variables for indices into the array are typically ignored.

Summary

- The sorting algorithms in this chapter all assume an array as a data storage structure.
- Sorting involves comparing the keys of data items in the array and moving the items (usually references to the items) around until they're in sorted order.
- All the algorithms in this chapter execute in $O(N^2)$ time. Nevertheless, some can be substantially faster than others.
- An invariant is a condition that remains unchanged while an algorithm runs.
- The bubble sort is the least efficient but the simplest sort.
- The insertion sort is the most commonly used of the $O(N^2)$ sorts described in this chapter.
- The selection sort performs $O(N^2)$ comparisons and only O(N) swaps, which can be important when swap time is much more significant than comparison time.
- A sort is stable if the order of elements with the same key is retained.
- None of the sorts in this chapter require more than a single temporary record variable, in addition to the original array.

Questions

These questions are intended as a self-test for readers. Answers may be found in Appendix C.

1. Computer sorting algorithms are more limited than humans in that
   a. the amount of data that computers can sort is much less than what humans can.
   b. humans can invent new sorting algorithms, whereas computers cannot.
   c. humans know what to sort, whereas computers need to be told.
   d. computers can compare only two things at a time, whereas humans can compare small groups.
2. The two basic operations in simplesorting are _____ items and _____ them (or sometimes _____ them).
3. True or False: The bubble sort always ends up comparing every possible pair of items in the initial array.
4. The bubble sort algorithm alternates between
   a. comparing and swapping.
   b. moving and copying.

c. moving and comparing.

d. copying and comparing.

5. True or False: If there are N items, the bubble sort makes exactly N×N comparisons.

6. In the selection sort,

   a. the largest keys accumulate on the left (low indices).

   b. a minimum key is repeatedly discovered.

   c. a number of items must be shifted to insert each item in its correctly sorted position.

   d. the sorted items accumulate on the right.

7. True or False: If, on a particular computing platform, swaps take much longer than comparisons, the selection sort is about twice as fast as the bubble sort for all values of N.

8. Ignoring the details of where the computer stores each piece of data, what is a reasonable assumption about the ratio of the amounts of time taken for a copy operation versus a swap operation?

9. What is the invariant in the selection sort?

10. In the insertion sort, the "marked player" described in the text corresponds to which variable in the `insertionSort()` method?

    a. `inner`

    b. `outer`

    c. `temp`

    d. `__a[outer]`

11. In the insertion sort, the "partially sorted" group members are

    a. the items that are already sorted but still need to be moved as a block.

    b. the items that are in their final block position but may still need to be sorted.

    c. only partially sorted in order by their keys.

    d. the items that are sorted among themselves, but items outside the group may need to be inserted in the group.

12. Shifting a group of items left or right requires repeated _____.

13. In the insertion sort, after an item is inserted in the partially sorted group, it

    a. is never moved again.

    b. is never shifted to the left.

c. is often moved out of this group.

d. finds that its group is steadily shrinking.

14. The invariant in the insertion sort is that _____.

15. Stability might refer to

a. items with secondary keys being excluded from a sort.

b. keeping cities sorted by increasing population within each state, in a sort by state.

c. keeping the same given names matched with the same family names.

d. items keeping the same order of secondary keys without regard to primary keys.

Experiments

Carrying out these experiments will help to provide insights into the topics covered in the chapter. No programming is involved.

3-A The "Stability" section explains that stable sorting algorithms don't change the *relative* position within the array of items having equal-valued keys. For example, when you're sorting an array containing tuples by their first element such as

(elm, 1), (asp, 1), (elm, 2), (oak, 1)

the algorithm always produces

(asp, 1), (elm, 1), (elm, 2), (oak, 1)

and never

(asp, 1), (elm, 2), (elm, 1), (oak, 1)

Review each of the simplesorting algorithms in this chapter and determine if they are always stable.

Determining whether a sorting algorithm is stable or not can be difficult. Try walking through some sample inputs that include some equal-valued keys to see when those items are moved. If you can think of even one example where two items with equal keys are reordered from their original relative order, then you have proven that the algorithm is unstable. If you can't think of an example where equal-valued keys are moved out of order, then the algorithm *might* be stable. To be sure, you need to provide more proof. That's usually done by finding invariant conditions about a partial sequence of items as the algorithm progresses. For example, if you can show that the output array after index `i` contains only items in stable order, that condition stays valid at iteration *j* if it held in iteration *j* – 1, and that `i` always ends at index 0, then the algorithm must be stable.

Note that if you get stuck, we provide an answer for this exercise in Appendix C.

3-B Sometimes the items in an array might have just a few distinct keys with many copies. That creates a situation where there are many duplicates of the same key. Use the SimpleSorting Visualization tool to create an array with 15 cells and filled with only two distinct values, say 10 and 90. Try shuffling and sorting that kind of array with the three sorting algorithms. Do any of them show advantages or disadvantages over the others for this kind of data?

3-C In the `selectionSort()` method shown in Listing 3-2, the inner loop makes a swap on every pass. It doesn't need to swap them if the value of the `min` and `outer` indices are the same. Would it make sense to add an additional comparison of those indices and perform the swap only if they are different? If so, under what conditions would that improve the performance? If not, why not?

Programming Projects

Writing programs to solve the Programming Projects helps to solidify your understanding of the material and demonstrates how the chapter's concepts are applied. (As noted in the Introduction, qualified instructors may obtain completed solutions to the Programming Projects on the publisher's website.)

3.1 In the `bubbleSort()` method (Listing 3-1) and the Visualization tool, the `inner` index always goes from left to right, finding the largest item and carrying it out on the right. Modify the `bubbleSort()` method so that it's bidirectional. This means the `inner` index will first carry the largest item from left to right as before, but when it reaches `last`, it will reverse and carry the smallest item from right to left. You need two outer indexes, one on the right (the old `last`) and another on the left.

3.2 Add a method called `median()` to the `Array` class in the `SortArray.py` module (Listing 3-4). This method should return the median value in the array. (Recall that in a group of numbers, half are larger than the median and half are smaller.) Do it the easy way.

3.3 Add a method called `deduplicate()` to the `Array` class in the `SortArray.py` module (Listing 3-4) that removes duplicates from a previously sorted array without disrupting the order. You can use any of the sort methods in your test program to sort the data. You can imagine schemes in which all the items from the place where a duplicate was discovered to the end of the array would be shifted down one space every time a duplicate was discovered, but this would lead to slow $O(N^2)$ time, at least when there were a lot of duplicates. In your algorithm, make sure no item is moved more than once, no matter how many duplicates there are. This will give you an algorithm with O(N) time.

3.4 Another simple sort is the odd-even sort. The idea is to repeatedly make two passes through the array. On the first pass, you look at all the pairs of items, a[j] and a[j + 1], where j is odd (j = 1, 3, 5, …). If their key values are out of order, you swap them. On the second pass, you do the same for all the even values (j = 0, 2, 4, …). You do

these two passes repeatedly until the array is sorted. Add an `oddEvenSort()` method to the `Array` class in the `SortArray.py` module (Listing 3-4). Perform the outer loop until no swaps occur to see how many passes are needed; one pass includes both odd-pair and even-pair swapping. Make sure it works for varying amounts of data and on good and bad initial orderings. After testing how many passes are needed before no more swaps occur, determine the maximum number of passes of the outer loop based on the length of the input array.

The odd-even sort is actually useful in a multiprocessing environment, where a separate processor can operate on each odd pair simultaneously and then on each even pair. Because the odd pairs are independent of each other, each pair can be checked—and swapped, if necessary—by a different processor. This makes for a very fast sort.

3.5 Modify the `insertionSort()` method in `SortArray.py` (Listing 3-4) so that it counts the number of copies and the number of item comparisons it makes during a sort and displays the totals. You need to look at the loop condition in the inner `while` loop and carefully count item comparisons. Use this program to measure the number of copies and comparisons for different amounts of inversely sorted data. Do the results verify $O(N^2)$ efficiency? Do the same for almost-sorted data (only a few items out of place). What can you deduce about the efficiency of this algorithm for almost-sorted data?

3.6 Here's an interesting way to remove items with duplicate keys from an array. The insertion sort uses a loop-within-a-loop algorithm that compares every item in the array with the partially sorted items so far. One way to remove items with duplicate keys would be to modify the algorithm in the `Array` class in the `SortArray.py` module (Listing 3-4) so that it removes the duplicates as it sorts. Here's one approach: When a duplicate key is found, instead of copying the duplicate back into the array cell at its sorted position, change the key for the marked item to be a special value that is treated as lower than any other possible key. With that low key value, it is automatically moved into place at the beginning of the array. By keeping track of how many duplicates are found, you know the end of duplicates and beginning of the remaining elements in the array. When the outer loop ends, the algorithm would have to make one more pass to shift the unique keys into the cells occupied by the duplicates. Write an `insertionSortAndDedupe()` method that performs this operation. Make sure to test that it works with all different kinds of input array data.

CHAPTER 4

Stacks and Queues

IN THIS CHAPTER

- Different Structures for Different Use Cases
- Stacks
- Queues
- Priority Queues
- Parsing Arithmetic Expressions

This chapter examines three data storage structures widely used in all kinds of applications: the stack, the queue, and the priority queue. We describe how these structures differ from arrays, examining each one in turn. In the last section, we look at an operation in which the stack plays a significant role: parsing arithmetic expressions.

Different Structures for Different Use Cases

You choose data structures to use in programs based on their suitability for particular tasks. The structures in this chapter differ from what you've already seen in previous chapters in several ways. Those differences guide the decision on when to apply them to a new task.

Storage and Retrieval Pattern

Arrays—the data storage structure examined thus far—as well as many other structures you encounter later in this book (linked lists, trees, and so on) are appropriate for all kinds of data storage. They are sometimes referred to as low-level data structures because they are used in implementing many other data structures. The organization of the data dictates what it takes to retrieve a particular record or object that is stored inside them. Arrays are especially good for applications where the items can be indexed by an integer because the retrieval is fast and independent of the number of items stored. If the index isn't known, the array still can be searched to find the object. As you've seen, sorting the objects made the search faster at the expense of other operations. Insertion and deletion operations can either be constant time or have the same Big O behavior as the search operation.

The structures and algorithms we examine in this chapter, on the other hand, are used in situations where you are unlikely to need to search for a particular object or item that is stored but rather where you want to process those objects in a particular order.

Restricted Access

In an array, any item can be accessed, either immediately—if its index number is known—or by searching through a sequence of cells until it's found. In the data structures in this chapter, however, access is restricted: only one item can be read or removed at a time.

The interface of these structures is designed to enforce this restricted access. Access to other items is (in theory) not allowed.

More Abstract

Stacks, queues, and priority queues are more abstract structures than arrays and many other data storage structures. They're defined primarily by their interface—the permissible operations that can be carried out on them. The underlying mechanism used to implement them is typically not visible to their user.

The underlying mechanism for a stack, for example, can be an array, as shown in this chapter, or it can be a linked list. The underlying mechanism for a priority queue can be an array or a special kind of tree called a **heap**. When one data structure is used to implement a more abstract one, we often say that it is a *lower-level data structure* by picturing the more abstract structure being layered on top of it. We return to the topic of one data structure being implemented by another when we discuss abstract data types (ADTs) in Chapter 5, "Linked Lists."

Stacks

A **stack** data structure allows access to only one data item in the collection: the last item inserted. If you remove this item, you can access the next-to-last item inserted, and so on. Although that constraint seems to be quite limiting, this behavior occurs in many programming situations. In this section we show how a stack can be used to check whether parentheses, braces, and brackets are balanced in a computer program source file. At the end of this chapter, a stack plays a vital role in parsing (analyzing) arithmetic expressions such as 3×(4+5).

A stack is also a handy aid for algorithms applied to certain complex data structures. In Chapter 8, "Binary Trees," you will see it used to help traverse the nodes of a tree. In Chapter 14, "Graphs," you will apply it to searching the vertices of a graph (a technique that can be used to find your way out of a maze).

Nearly all computers use a stack-based architecture. When a function or method is called, the return address for where to resume execution and the function arguments are **pushed** (inserted) onto a stack in a particular order. When the function returns, they're **popped** off. The stack operations are often accelerated by the computer hardware.

The idea of having all the function arguments pushed on a stack makes very clear what the operands are for a particular function or operator. This scheme is used in some older pocket calculators and formal languages such as PostScript and PDF content streams. Instead of entering arithmetic expressions using parentheses to group operands, you insert (or push) those values onto a stack. The operator comes after the operands and pops them off, leaving the (intermediate) result on the stack. You will learn more about this approach when we discuss parsing arithmetic expressions in the last section in this chapter.

The Postal Analogy

To understand the idea of a stack, consider an analogy provided by postal, and to a lesser extent, some email and messaging systems, especially ones where you can see only one message at a time. Many people, when they get their mail, toss it onto a stack on a table. If they've been away awhile, the stack might be quite large. If they don't open the mail, the following day, they may add more messages on top of the stack. In messaging systems, a large number of messages can accumulate since the last time people viewed them. Then, when they have a spare moment, they start to process the accumulated mail or messages from the top down, one at a time. First, they open the letter on the top of the stack and take appropriate action—paying the bill, reading the holiday newsletter, considering an advertisement, throwing it away, or whatever. After disposing of the first message, they examine the next one down, which is now the top of the stack, and deal with that. The idea is to start with the most recently delivered letter or message first. Eventually, they work their way down to the message on the bottom of the stack (which is now the top-most one). Figure 4-1 shows a stack of mail.

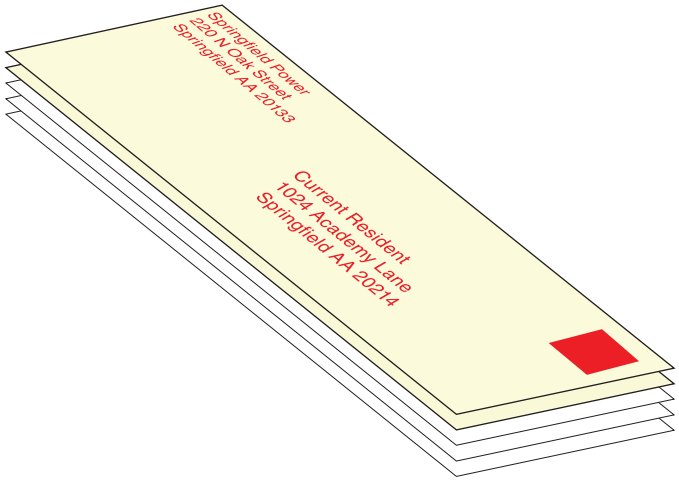


FIGURE 4-1 A stack of letters

This "do the top one first" approach works all right as long as people can easily process all the items in a reasonable time. If they can't, there's the danger that messages on the

bottom of the stack won't be examined for months, and the bills or other important requests for action are not handled soon enough. By reviewing the most recent one first, however, they're at least guaranteed to find any updates to bills or messages that occurred after the earlier deliveries.

Of course, many people don't rigorously follow this top-to-bottom approach. They may, for example, take the mail off the bottom of the stack, to process the oldest letter first. That way, they see the messages in chronological order of arrival. Or they might shuffle through the messages before they begin processing them and put higher-priority messages on top. In these cases, their processing system is no longer a stack in the computer-science sense of the word. If they take mail off the bottom, it's a **queue**; and if they prioritize it, it's a **priority queue**. We'll look at these possibilities later.

The tasks you perform every day can be seen as a stack. You typically have a set of goals, often with different priorities. They might include things like these: be happy, be with my family, or live a long life. Each of those goals can have sub-goals. For instance, to be happy, you may have a sub-goal: to be financially stable. A sub-goal of that could be: to have a job that pays enough. The tasks that you perform each day are organized around these goals. So, if "be with my family" is a main goal, you go to work to satisfy your other goal of being financially stable. When work is done, you return to your main goal by returning home. While at work, you might start the day by working on a long-term project. When the manager asks you to handle some new task that just popped up, you set aside the project and work the pop-up task. If a more critical problem comes up, you set aside the pop-up task and solve the problem. When each higher-priority activity is finished, you return to the next lower-priority activity on your *stack* of tasks.

In data structures, placing a data item on the top of the stack is called *pushing* it. Removing it from the top of the stack is called *popping* it. Looking at the top item without popping it is called **peeking**. These are the primary stack operations. A stack is said to be a last-in, first-out (LIFO) storage mechanism because the last item inserted is the first one to be removed.

The Stack Visualization Tool

Let's use the Stack Visualization tool to get an idea how stacks work. When you launch this tool by following the guide in Appendix A, "Running the Visualizations," you see an empty stack with operations for Push, New, Pop, Peek, and the animation controls shown in Figure 4-2.

The Stack Visualization tool is based on an array, so you see an array of data items, this time arranged in a column. Although it's based on an array, a stack restricts access, so you can't access elements using an index (even though they appear next to the cells in the visualization). In fact, the concept of a stack and the underlying data structure used to implement it are quite separate. As we noted earlier, stacks can also be implemented by other kinds of storage structures, such as linked lists.

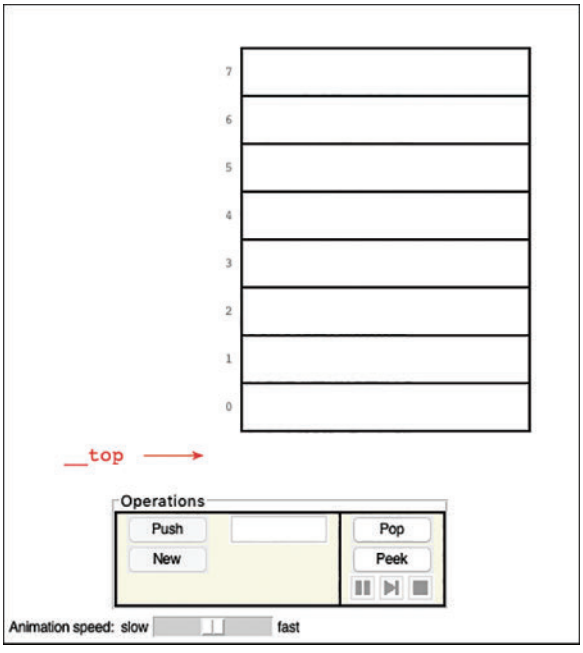


FIGURE 4-2 The Stack Visualization tool

The Push Button

To insert a data item on the stack, use the button labeled Push. The item can be any text string, and you simply type it in the text entry box next to Push. The tool limits the number of characters in the string, so they fit in the array cells.

To the left of the stack is a reddish-brown arrow labeled "`__top`." This is part of the data structure, an attribute that tracks the last item inserted. Initially, nothing has been inserted, so it points below the first array cell, the one labeled with a small 0. When you push an item, it moves up to cell 0, and the item is inserted in that cell. Notice that the `top` arrow is incremented before the item is inserted. That's important because copying the data before incrementing could either cause an error by writing outside of the array bounds or overwriting the last element in the array when there are multiple items.

Try pushing several items on the stack. The tool notes the last operation you chose when typing an argument, so if you press Return or Enter while typing the next one, it will run the push operation again. If the stack is full and you try to push another item, you'll get the `Error! Stack is already full` message. (Theoretically, an ADT stack doesn't become full, but any practical implementation does.)

The New Button

As with all data structures, you create (and remove) stacks as needed for an algorithm. Because this implementation is based on an array, it needs to know how big the array should be. The Stack Visualization tool starts off with eight cells allocated and none of

them filled. To make a new, empty stack, type the number of cells it should have in the text entry box and then select New. The tool adjusts the height of the cells to fit them within the display window. If you ask for a size that won't fit, you'll get an error message.

The Pop Button

To remove a data item from the top of the stack, use the Pop button. The value popped is moved to the right as it is being copied to a variable named `top`. It is stored there temporarily as the `pop()` method finishes its work.

Again, notice the steps involved. First, the item is copied from the cell pointed to by `__top`. Then the cell is cleared, and then `__top` is decremented to point to the highest occupied cell. The order of copying and changing the index pointer is the reverse of the sequence used in the push operation.

The tool's Pop operation shows the item actually being removed from the array and the cell becoming empty. This is not strictly necessary because the value could be left in the array and simply ignored. In programming languages like Python, however, references to items left in an array like this can consume memory. It's important to remove those references so that the memory can be reclaimed and reused. We show this in detail when looking at the code. Note that you still have a reference to the item in the `top` variable so that it won't be lost.

After the topmost item is removed, the `__top` index decrements to the next lower cell. After you pop the last item off the stack, it points to –1, below the lowest cell. This position indicates that the stack is empty. If the stack is empty and you try to pop an item, you'll get the `Error! Stack is empty` message.

The Peek Button

Push and pop are the two primary stack operations. It's sometimes useful, however, to be able to read the value from the top of the stack without removing it. The peek operation does this. By selecting the Peek button a few times, you see the value of the item at the `__top` index copied to the output box on the right, but the item is not removed from the stack, which remains unchanged.

Notice that you can peek only at the top item. By design, all the other items are invisible to the stack user. If the stack is empty in the visualization tool when you select Peek, the output box is not created. We show how to implement the empty test later in the code.

Stack Size

Stacks come in all sizes and are typically capped in size so that they can be allocated in a single block of memory. The application allocates some initial size based on the maximum number of items expected to be stacked. If the application tries to push items beyond what the stack can hold, then either the stack needs to increase in size, or an exception must occur. The visualization tool limits the size to what fits conveniently on the screen, but stacks in many applications can have thousands or millions of cells.

Python Code for a Stack

The Python `list` type is a natural choice for implementing stacks. As you saw in Chapter 2, "Arrays," the `list` type can act like an array (as opposed to a linked list, as

shown in Chapter 5, "Linked Lists"). Instead of using Python slicing to get the last element in the list, we continue to use the list as a basic array and keep a separate `__top` pointer for the topmost item, as shown in Listing 4-1.

LISTING 4-1 The `SimpleStack.py` Module

```
# Implement a Stack data structure using a Python list

class Stack(object):
   def __init__(self, max):                # Constructor
      self.__stackList = [None] * max      # The stack stored as a list
      self.__top = -1                      # No items initially

   def push(self, item):                   # Insert item at top of stack
      self.__top += 1                      # Advance the pointer
      self.__stackList[self.__top] = item  # Store item

   def pop(self):                          # Remove top item from stack
      top = self.__stackList[self.__top]   # Top item
      self.__stackList[self.__top] = None  # Remove item reference
      self.__top -= 1                      # Decrease the pointer
      return top                           # Return top item

   def peek(self):                         # Return top item
      if not self.isEmpty():               # If stack is not empty
         return self.__stackList[self.__top] # Return the top item

   def isEmpty(self):                      # Check if stack is empty
      return self.__top < 0

   def isFull(self):                       # Check if stack is full
      return self.__top >= len(self.__stackList) - 1

   def __len__(self):                      # Return # of items on stack
      return self.__top + 1

   def __str__(self):                      # Convert stack to string
      ans = "["                            # Start with left bracket
      for i in range(self.__top + 1):      # Loop through current items
         if len(ans) > 1:                  # Except next to left bracket,
            ans += ", "                    # separate items with comma
         ans += str(self.__stackList[i])   # Add string form of item
      ans += "]"                           # Close with right bracket
      return ans
```

The `SimpleStack.py` implementation has just the basic features needed for a stack. Like the `Array` class you saw in Chapter 2, the constructor allocates an array of known size, called `__stackList`, to hold the items with the `__top` pointer as an index to the topmost item in the stack. Unlike the `Array` class, `__top` points to the topmost item and not the next array cell to be filled. Instead of `insert()`, it has a `push()` method that puts a new item on top of the stack. The `pop()` method returns the top item on the stack, clears the array cell that held it, and decreases the stack size. The `peek()` method returns the top item without decreasing the stack size.

In this simple implementation, there is very little error checking. It does include `isEmpty()` and `isFull()` methods that return Boolean values indicating whether the stack has no items or is at capacity. The `peek()` method checks for an empty stack and returns the top value only if there is one. To avoid errors, a client program would need to use `isEmpty()` before calling `pop()`. The class also includes methods to measure the stack depth and a string conversion method for convenience in displaying stack contents.

To exercise this class, you can use the `SimpleStackClient.py` program shown in Listing 4-2.

LISTING 4-2 The `SimpleStackClient.py` Program

```
from SimpleStack import *

stack = Stack(10)

for word in ['May', 'the', 'force', 'be', 'with', 'you']:
   stack.push(word)

print('After pushing', len(stack),
      'words on the stack, it contains:\n', stack)
print('Is stack full?', stack.isFull())

print('Popping items off the stack:')
while not stack.isEmpty():
   print(stack.pop(), end=' ')
print()
```

The client creates a small stack, pushes some strings onto the stack, displays the contents, then pops them off, printing them left to right separated by spaces. The transcript of running the program shows the following:

```
$ python3 SimpleStackClient.py
After pushing 6 words on the stack, it contains:
 [May, the, force, be, with, you]
Is stack full? False
Popping items off the stack:
you with be force the May
```

Notice how the program reverses the order of the data items. Because the last item pushed is the first one popped, the you appears first in the output. Figure 4-3 shows how the data gets placed in the array cells of the stack and then returned for push and pop operations. The array cells shown as empty in the illustration still have the value `None` in them, but because only the values from the bottom up through the `__top` pointer are occupied, the ones beyond `__top` can be considered unfilled. The visualization tool uses the same `pop()` method as in Listing 4-1, setting the popped cell to `None`.

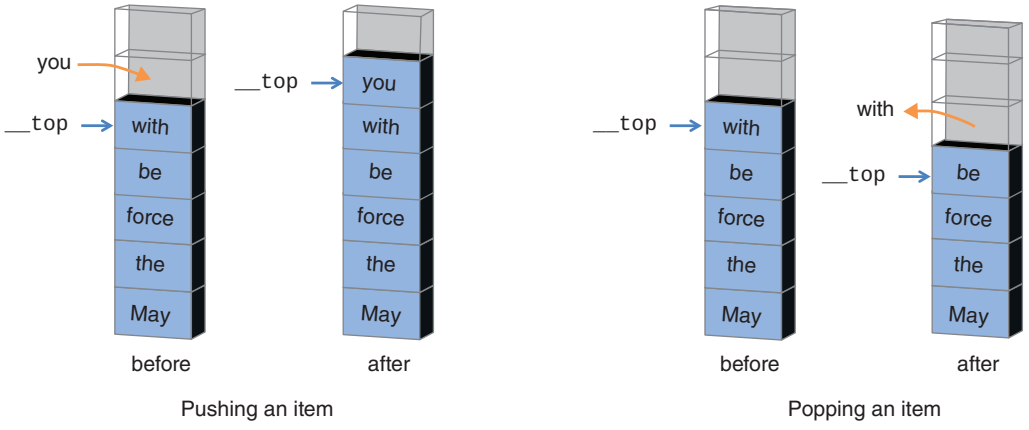


FIGURE 4-3 Operation of the `Stack` class `push()` and `pop()` methods

Error Handling

There are different philosophies about how to handle stack errors. What should happen if you try to push an item onto a stack that's already full or pop an item from a stack that's empty?

In this implementation, the responsibility for avoiding or handling such errors has been left up to the program using the class. That program should always check to be sure the stack is not full before inserting an item, as in

```
if not stack.isFull():
   stack.push(item)
else:
   print("Can't insert, stack is full")
```

The client program in Listing 4-2 checks for an empty stack before it calls `pop()`. It does not, however, check for a full stack before calling `push()`. Alternatively, many stack classes check for these conditions internally, in the `push()` and `pop()` methods. Typically, a stack class that discovers such conditions either throws an exception, which can then be caught and processed by the program using the class, or takes some predefined action, such as returning `None`. By providing both the ability for users to query the internal state and the

possibility of raising exceptions when constraints are violated, the data structure enables both **proactive** and **reactive** approaches.

Stack Example 1: Reversing a Word

For this first example of using a stack, we examine a simple task: reversing a word. When you run the program, you type in a word or phrase, and the program displays the reversed version.

A stack is used to reverse the letters. First, the characters are extracted one by one from the input string and pushed onto the stack. Then they're popped off the stack and displayed. Because of its last-in, first-out characteristic, the stack reverses the order of the characters. Listing 4-3 shows the code for the `ReverseWord.py` program.

LISTING 4-3 The `ReverseWord.py` Program

```
                          # A program to reverse the letters of a word

from SimpleStack import *

stack = Stack(100)        # Create a stack to hold letters

word = input("Word to reverse: ")

for letter in word:       # Loop over letters in word
   if not stack.isFull(): # Push letters on stack if not full
      stack.push(letter)

reverse = ''              # Build the reversed version
while not stack.isEmpty():# by popping the stack until empty
   reverse += stack.pop()

print('The reverse of', word, 'is', reverse)
```

The program makes use of Python's `input()` function, which prints a prompt string and then waits for a user to type a response string. The program constructs a stack instance, gets the word to be reversed from the user, loops over the letters in the word, and pushes them on the stack. Here the program avoids pushing letters if the stack is already full. After all the letters are pushed on the stack, the program creates the reversed version, starting with an empty string and appending each character popped from the stack. The results look like this:

```
$ python3 ReverseWord.py
Word to reverse: draw
The reverse of draw is ward
$ python3 ReverseWord.py
Word to reverse: racecar
The reverse of racecar is racecar
```

```
$ python3 ReverseWord.py
Word to reverse: bolton
The reverse of bolton is notlob
$ python3 ReverseWord.py
Word to reverse: A man a plan a canal Panama
The reverse of A man a plan a canal Panama is amanaP lanac a nalp a nam A
```

The results show that single-word palindromes come out the same as the input. If the input "word" has spaces, as in the last example, the spaces are treated just as any other letter in the string.

Stack Example 2: Delimiter Matching

One common use for stacks is to parse certain kinds of text strings. Typically, the strings are lines of code in a computer language, and the programs parsing them are compilers or interpreters.

To give the flavor of what's involved, let's look at a program that checks the delimiters in an expression. The text expression doesn't need to be a line of real code (although it could be), but it should use delimiters the same way most programming languages do. The delimiters are the curly braces { and }, square brackets [and], and parentheses (and). Each opening, or left, delimiter should be matched by a closing, or right, delimiter; that is, every { should be followed by a matching } and so on. Also, opening delimiters that occur later in the string should be closed before those occurring earlier. Here are some examples:

```
c[d]        # correct
a{b[c]d}e   # correct
a{b(c]d}e   # not correct; ] doesn't match (
a[b{c}d]e}  # not correct; nothing matches final }
a{b(c)      # not correct; nothing matches opening {
```

Opening Delimiters on the Stack

This delimiter-matching program works by reading characters from the string one at a time and placing opening delimiters, when it finds them, on a stack. When it reads a closing delimiter from the input, it pops the opening delimiter from the top of the stack and attempts to match it with the closing delimiter. If they're not the same type (there's an opening brace but a closing parenthesis, for example), an error occurs. Also, if there is no opening delimiter on the stack to match a closing one, or if a delimiter has not been matched, an error occurs. It discovers unmatched delimiters because they remain on the stack after all the characters in the string have been read.

Let's see what happens on the stack for a typical correct string:

```
a{b(c[d]e)f}
```

Table 4-1 shows how the stack looks as each character is read from this string. The entries in the second column show the stack contents, reading from the bottom of the stack on the left to the top on the right.

Table 4-1 Stack Contents in Delimiter Matching the Expression `a{b(c[d]e)f}`

Character Read	Stack Contents
a	
{	{
b	{
(	{(
c	{(
[	{([
d	{([
]	{(
e	{(
)	{
f	{
}	

As the string is read, each opening delimiter is placed on the stack. Each closing delimiter read from the input is matched with the opening delimiter popped from the top of the stack. If they form a pair, all is well. Nondelimiter characters are not inserted on the stack; they're ignored.

This approach works because pairs of delimiters that are opened last should be closed first. This matches the last-in, first-out property of the stack.

Python Code for DelimiterChecker.py

Listing 4-4 shows the code for the parsing program, `DelimiterChecker.py`. Like `ReverseWord.py`, the program accepts an expression after prompting the user using `input()` and prints out any errors found or a message saying the delimiters are balanced.

LISTING 4-4 The `DelimiterChecker.py` Program

```
# A program to check that delimiters are balanced in an expression

from SimpleStack import *

stack = Stack(100)          # Create a stack to hold delimiter tokens

expr = input("Expression to check: ")
```

```
errors = 0                  # Assume no errors in expression

for pos, letter in enumerate(expr): # Loop over letters in expression
   if letter in "{[(":      # Look for starting delimiter
      if stack.isFull():    # A full stack means we can't continue
         raise Exception('Stack overflow on expression')
      else:
         stack.push(letter) # Put left delimiter on stack

   elif letter in "}])":    # Look for closing delimiter
      if stack.isEmpty():   # If stack is empty, no left delimiter
         print("Error:", letter, "at position", pos,
               "has no matching left delimiter")
         errors += 1
      else:
         left = stack.pop() # Get left delimiter from stack
         if not (left == "{" and letter == "}" or  # Do delimiters
                 left == "[" and letter == "]" or  # match?
                 left == "(" and letter == ")"):
            print("Error:", letter, "at position", pos,
                  "does not match left delimiter", left)
            errors += 1

# After going through entire expression, check if stack empty
if stack.isEmpty() and errors == 0:
   print("Delimiters balance in expression", expr)

elif not stack.isEmpty():   # Any delimiters on stack weren't matched
   print("Expression missing right delimiters for", stack)
```

The program doesn't define any functions; it processes one expression from the user and exits. It creates a `Stack` object to hold the delimiters as they are found. The `errors` variable is used to track the number of errors found in parsing the expression. It loops over the characters in the expression using Python's `enumerate()` sequencer, which gets both the index and the value of the string at that index. As it finds starting (or left) delimiters, it pushes them on the stack, avoiding overflowing the stack. When it finds ending (or closing or right) delimiters, it checks whether there is a matching delimiter on the top of the stack. If not, it prints the error and continues. Some sample outputs are

```
$ python3 DelimiterChecker.py
Expression to check: a()
Delimiters balance in expression a()
$ python3 DelimiterChecker.py
Expression to check: a( b[4]d )
Delimiters balance in expression a( b[4]d )
$ python3 DelimiterChecker.py
```

```
Expression to check: a( b]4[d )
Error: ] at position 4 does not match left delimiter (
Error: ) at position 9 does not match left delimiter [
$ python3 DelimiterChecker.py
Expression to check: {{a( b]4[d )
Error: ] at position 6 does not match left delimiter (
Error: ) at position 11 does not match left delimiter [
Expression missing right delimiters for [{, {]
```

The messages printed by `DelimiterChecker.py` show the position and value of significant characters (delimiters) in the expression. After processing all the characters, any remaining delimiters on the stack are unmatched. The program prints an error for that using the `Stack's` string conversion method, which surrounds the list in square brackets, possibly leading to some confusion with the input string delimiters.

The Stack as a Conceptual Aid

Notice how convenient the stack is in the `DelimiterChecker.py` program. You could have set up an array to do what the stack does, but you would have had to worry about keeping track of an index to the most recently added character, as well as other bookkeeping tasks. The stack is conceptually easier to use. By providing limited access to its contents, using the `push()` and `pop()` methods, the stack has made the delimiter-checking program easier to understand and less error prone. (As carpenters will tell you, it's safer and faster to use the right tool for the job.)

Efficiency of Stacks

Items can be both pushed and popped from the stack implemented in the `Stack` class in constant O(1) time. That is, the time is not dependent on how many items are in the stack and is therefore very quick. No comparisons or moves within the stack are necessary.

Queues

In computer science, a **queue** is a data structure that is somewhat like a stack, except that in a queue the first item inserted is the first to be removed (first-in, first-out, or FIFO). In stacks, as you've seen, the last item inserted is the first to be removed (LIFO). A queue models the way people wait for something, such as tickets being sold at a window or a chance to greet the bride and groom at a big wedding. The first person to arrive goes first, the next goes second, and so forth. Americans call it a waiting line, whereas the British call it a queue. The key aspect is that the first items to arrive are the first to be processed.

Queues are used as a programmer's tool just like stacks are. They are found everywhere in computer systems: the jobs waiting to run, the messages to be passed over a network, the sequence of characters waiting to be printed on a terminal. They're used to model real-world situations such as people waiting in line for tickets, airplanes waiting to take off, or students waiting to see whether they get into a particular course. This ordering is

sometimes called **arrival ordering** because the time of arrival in the queue determines the order.

Various queues are quietly doing their job in your computer's (or the network's) operating system. There's a printer queue where print jobs wait for the printer to be available. Queues also store user input events like keystrokes, mouse clicks, touchscreen touches, and microphone inputs. They are really important in multiprocessing systems so that each event can be processed in the correct order even when the processor is busy doing something else when the event occurs.

The two basic operations on the queue are called **insert** and **remove**. Insert corresponds to a person inserting themself at the rear of a ticket line. When that person makes their purchase, they remove themself from the front of the line.

The terms for insertion and removal in a stack are fairly standard; everyone says *push* and *pop*. The terminology for queues is not quite as standardized. *Insert* is also called *put* or *add* or *enqueue*, whereas *remove* may be called *delete* or *get* or *dequeue*. The rear of the queue, where items are inserted, is also called the *back* or *tail* or *end*. The front, where items are removed, may also be called the *head*. In this book, we use the terms *insert*, *remove*, *front*, and *rear*.

A Shifty Problem

In thinking about how to implement a queue using arrays, the first option would be to handle inserts like a push on a stack. The first item goes at the last position (first empty cell) of the array. Then when it's time to remove an item from the queue, you would take the first filled cell of the array. To avoid hunting for the position of those cells, you could keep two indices, `front` and `rear`, to track where the filled cells begin and end, as shown in Figure 4-4. When Ken arrives, he's placed in the cell indexed by `rear`. When you remove the first item in the queue, you get Raj from the cell indexed by `front`.

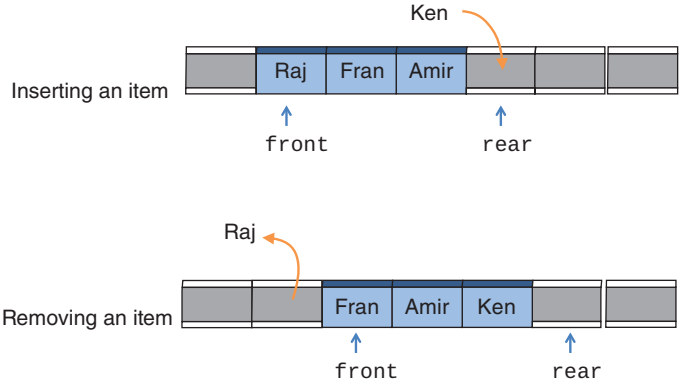


FIGURE 4-4 Queue operations in a linear array

This operation works nicely because both insert and remove simply copy an item and update an index pointer. It's just as fast as the push and pop operations of the stack and takes only one more variable to manage.

What happens when you get to the end of the array? If the `rear` index reaches the end of the array, there's no space to insert new items. That might be acceptable because it's no worse than when a stack runs out of room. If the `front` index has moved past the beginning of the array, however, free cells could be used for item storage. It seems wasteful not to take advantage of them.

One way to reclaim that unused storage space would be to shift all the items in the array when an insertion would go past the end. Shifting is similar to what happens with people standing in a line/queue; they all step forward as people leave the front of the queue. In the array, you would move the item indexed by `front` to cell 0 and move all the items up to `rear` the same number of cells; then you would set `front` to 0 and `rear` to `rear – front`. Shifting items, however, takes time, and doing so would make some insert operations take O(N) time instead of O(1). Is there a way to avoid the shifts?

A Circular Queue

To avoid the problem of not being able to insert more items into a queue when it's not full, you let the `front` and `rear` pointers **wrap around** to the beginning of the array. The result is a **circular queue** (sometimes called a **ring buffer**). This is easy to visualize if you take a row of cells and bend them around in the form of a circle so that the last cell and first cell are adjacent, as shown in Figure 4-5. The array has N cells in it, and they are numbered 0, 1, 2, …, N–2, N–1. When one of the pointers is at N–1 and needs to be incremented, you simply set it to 0. You still need to be careful not to let the wraparound go too far and start writing over cells that have valid items in them. To see how, let's look first at the Queue Visualization tool and then the code that implements it.

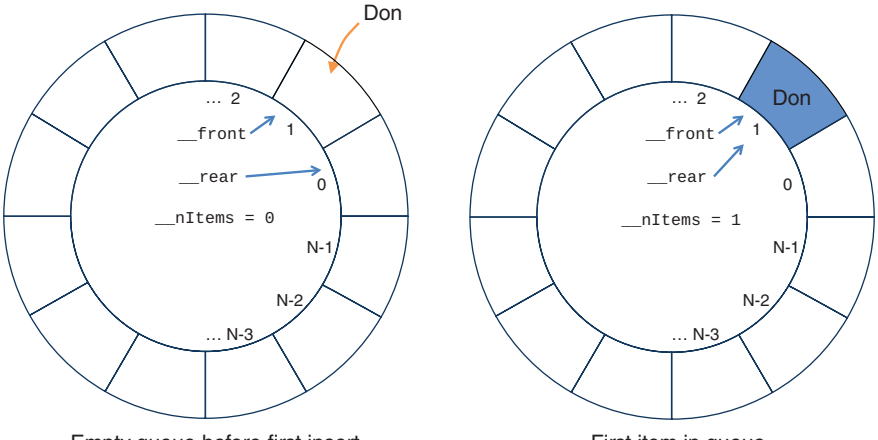


FIGURE 4-5 Operation of the `Queue.insert()` method on an empty queue

The Queue Visualization Tool

Let's use the Queue Visualization tool to get an idea how queues and circular arrays work. When you start the Queue tool, you see an empty queue that can hold 10 items, as shown in Figure 4-6. The array cells are numbered from 0 to 9 starting from the right edge. (The indices increase in the counterclockwise direction to match the convention in trigonometry where angles increase from the horizontal, X-axis.)

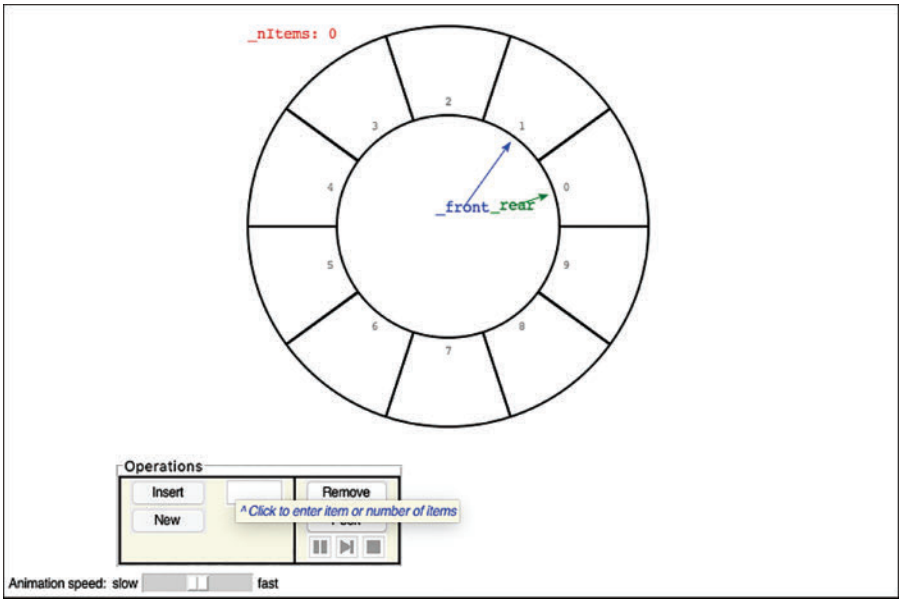


FIGURE 4-6 The Queue Visualization tool

The `_front` and `_rear` indices are shown in the center (with underscore prefixes to be somewhat like the attributes named in the code). There's also an `_nItems` counter at the top left. It might seem a little odd to have `front` point to 1 and `rear` point to 0, but the reason will become clearer shortly.

The Insert Button

After typing some text in the text entry box, select the Insert button. The tool responds by incrementing the `_rear` index and copying the value to cell 1. As with the stack, you can insert string values of limited length. Typing another string followed by pressing Return advances `_rear` to cell 2 and copies the value into it.

The Remove Button

When the queue has one or more items in it, selecting the Remove button copies the item at the `_front` cell to a variable named `front`. The `_front` cell is cleared and the index advances by one. The `front` variable holds the value returned by the operation.

Note that the `_front` and `_rear` indices can appear in any order. In the initial, empty queue, `_rear` was one less than `_front`. When the first item was inserted, both `_rear`

and_front pointed to the same cell. Additional inserts advance _rear past _front. The remove operations advance _front past _rear.

Keep inserting values until all the cells are filled. Note how the _rear index wraps around from 9 to 0. When all the cells are filled, _rear is one less than _front. That's the same relationship as when the queue was empty, but now the _nItems counter is 10, not 0.

The Peek Button

The peek operation returns the value of the item at the front of the queue without removing the item. (Like insert and remove, peek, when applied to a queue, is also called by a variety of other names.) If you select the Peek button, you see the value at _front copied to an output box. The queue remains unchanged.

Some queue implementations have a rearPeek() and a frontPeek() method, but usually you want to know what you're about to remove, not what you just inserted.

The New Button

If you want to start with an empty queue, you can use the New button to create one. Because it's based on an array, the size of the array is the argument that's required. The Queue Visualization tool lets you choose a range of queue sizes up to a limit that allows for values to be easily displayed. The animation shows the steps taken in the call to the object constructor.

Empty and Full

If you try to remove an item when there are no items in the queue, you'll get the Queue is empty! message. You'll also see that the code is highlighted in a different color because this operation has raised an exception. Similarly, if you try to insert an item when all the cells are already occupied, you'll get the Queue is full! message from a Queue overflow exception. These operations are shown in detail in the next section.

Python Code for a Queue

Let's look at how to implement a queue in Python using a circular array. Listing 4-5 shows the code for the Queue class. The constructor is a bit more complex than that of the stack because it must manage two index pointers for the front and rear of the queue. We choose to maintain an explicit count of the number of items, as explained later.

LISTING 4-5 The Queue.py Module

```
# Implement a Queue data structure using a Python list

class Queue(object):
   def __init__(self, size):             # Constructor
      self.__maxSize = size              # Size of [circular] array
      self.__que = [None] * size         # Queue stored as a list
      self.__front = 1                   # Empty Queue has front 1
      self.__rear = 0                    # after rear and
      self.__nItems = 0                  # No items in queue
```

```
def insert(self, item):                 # Insert item at rear of queue
   if self.isFull():                    # if not full
      raise Exception("Queue overflow")
   self.__rear += 1                     # Rear moves one to the right
   if self.__rear == self.__maxSize:    # Wrap around circular array
      self.__rear = 0
   self.__que[self.__rear] = item       # Store item at rear
   self.__nItems += 1
   return True

def remove(self):                       # Remove front item of queue
   if self.isEmpty():                   # and return it, if not empty
      raise Exception("Queue underflow")
   front = self.__que[self.__front]     # get the value at front
   self.__que[self.__front] = None      # Remove item reference
   self.__front += 1                    # front moves one to the right
   if self.__front == self.__maxSize: # Wrap around circular arr.
      self.__front = 0
   self.__nItems -= 1
   return front

def peek(self):                         # Return frontmost item
   return None if self.isEmpty() else self.__que[self.__front]

def isEmpty(self): return self.__nItems == 0

def isFull(self): return self.__nItems == self.__maxSize

def __len__(self): return self.__nItems

def __str__(self):                      # Convert queue to string
   ans = "["                            # Start with left bracket
   for i in range(self.__nItems):       # Loop through current items
      if len(ans) > 1:                  # Except next to left bracket,
         ans += ", "                    # separate items with comma
      j = i + self.__front              # Offset from front
      if j >= self.__maxSize:           # Wrap around circular array
         j -= self.__maxSize
      ans += str(self.__que[j])         # Add string form of item
   ans += "]"                           # Close with right bracket
   return ans
```

The `__front` and `__rear` pointers point at the first and last items in the queue, respectively. These and other attributes are named with double underscore prefixes to indicate they are private. They should not be changed directly by the object user.

When the queue is empty, where should `__front` and `__rear` point? We typically set one of them to 0, and we choose to do that for `__rear`. If we also set `__front` to be 0, we will have a problem inserting the first element. We set `__front` to 1 initially, as shown in the empty queue of Figure 4-5, so that when the first element is inserted and `__rear` is incremented, they both are 1. That's desirable because the first and last items in the queue are one and the same. So `__rear` and `__front` are 1 for the first item, and `__rear` is increased for the insertions that follow. That means the frontmost items in the queue are at lower indices, and the rearmost are at higher indices, in general.

The `insert()` method adds a new item to the rear of the queue. It first checks whether the queue is full. If it is, `insert()` raises an exception. This is the preferred way to implement data structures: provide tests so that callers can check the status in advance, but if they don't, raise an exception for invalid operations. Python's most general-purpose `Exception` class is used here with a custom reason string, `"Queue overflow"`. Many data structures define their own exception classes so that they can be easily distinguished from exception conditions like `ValueError` and `IndexError`.

You can avoid shifting items in the array during inserts by verifying that space is available before incrementing the `__rear` pointer and placing the new item at that empty cell of the array. The increment takes an extra step to handle the circular array logic when the pointer would go beyond the maximum size of the array by setting `__rear` back to 0. Finally, `insert()` increases the item count to reflect the inserted item at the rear.

The `remove()` method is similar in operation but acts on the `__front` of the queue. First, it checks whether the queue is empty and raises an "underflow" exception if it is. Then it makes a copy of the first item, clears the array cell, and increments the `__front` pointer to point at the next cell. The `__front` pointer must wrap around, just like `__rear` did, returning to 0 when it gets to the end of the array. The item count decreases because the front item was removed from the array, and the copy of the item is returned.

The `peek()` method looks at the frontmost item of the queue. You could create `peekfront()` and `peekrear()` methods to look at either end of the queue, but it's rare to need both in practice. The `peek()` method returns `None` when the queue is empty, although it might be more consistent to use the same underflow exception produced by `remove()`.

The `isEmpty()` and `isFull()` methods are simple tests on the number of items in the queue. Note that a slightly different Python syntax is used here. The whole body of the method is a single return statement. In that case, Python allows the statement to be placed after the colon ending the method signature. The `__len__()` method also uses the shortened syntax. Note also that these tests look at the `__nItems` value of the attribute rather than the `__front` and `__rear` indices. That's needed to distinguish the empty and full queues. We look at how wrapping the indices around makes that choice harder in Figure 4-9.

The last method for Queue is __str__(), which creates a string showing the contents of the queue enclosed in brackets and separated by commas for display. This method illustrates how circular array indices work. The beginning of the string has the front of the queue, and the end of the string is the rearmost item. The for loop uses the variable i to index all current items in the queue. A separate variable, j, starts at the __front and increments toward the __rear wrapping around if it passes the maximum size of the array.

Some simple tests of the Queue class are shown in Listing 4-6 and demonstrate the basic operations. The program creates a queue and inserts some names in it. The initial queue is empty, and Figure 4-5 shows how the first name, 'Don', is inserted. After that first insertion, both __front and __rear point at array cell 1 and the number of items is 1.

LISTING 4-6 The QueueClient.py Program

```
from Queue import *

queue = Queue(10)

for person in ['Don', 'Ken', 'Ivan', 'Raj', 'Amir', 'Adi']:
   queue.insert(person)

print('After inserting', len(queue),
      'persons on the queue, it contains:\n', queue)
print('Is queue full?', queue.isFull())

print('Removing items from the queue:')
while not queue.isEmpty():
   print(queue.remove(), end=' ')
print()
```

The person names inserted into the queue keep advancing the __rear pointer and increasing the __nItems count, as shown in Figure 4-7. After inserting all the names, the QueueClient.py program uses the __str__() method (implicitly) and prints the contents of the queue along with the status of whether the queue is full or not. After that's complete, the program removes items one at a time from the queue until the queue is empty. The items are printed separated by spaces (which is different from the way the __str__() method displays them). The result looks like this:

```
$ python3 QueueClient.py
After inserting 6 persons on the queue, it contains:
 [Don, Ken, Ivan, Raj, Amir, Adi]
Is queue full? False
Removing items from the queue:
Don Ken Ivan Raj Amir Adi
```

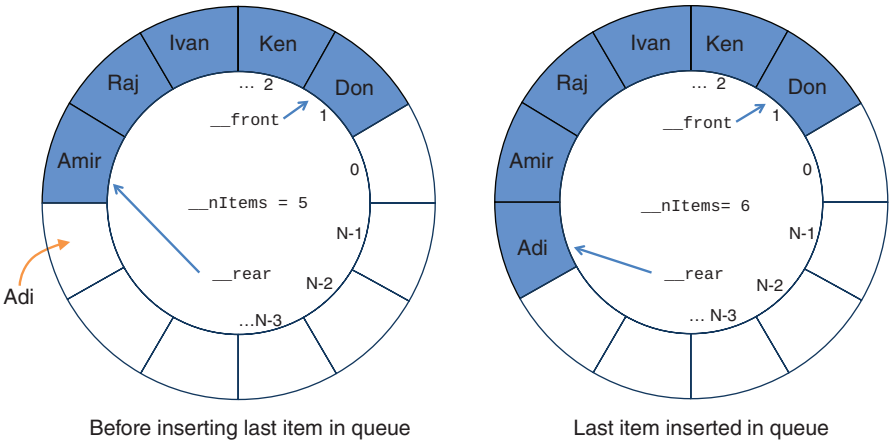


FIGURE 4-7 Inserting an item into a partially full queue

The printed result shows that items are deleted from the queue in the same order they were inserted in the queue. The first step of the deletion process is shown in Figure 4-8. When the first item, `'Don'`, is deleted from the front of the queue, the `__front` pointer is advanced to 2, and the number of items decreases to 5. The `__rear` pointer stays the same.

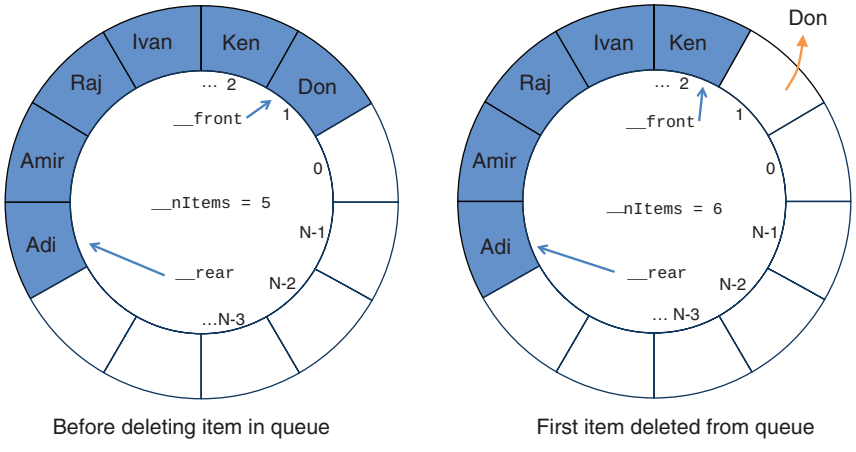


FIGURE 4-8 Deleting an item from the queue

Let's look at what happens when the queue wraps around the circular array. If you remove only one name and then insert more names into the queue, you'll eventually get to the situation shown in Figure 4-9. The `__rear` pointer keeps increasing with each insertion. After it gets to `__maxSize - 1`, the next insertion forces `__rear` to point at cell 0.

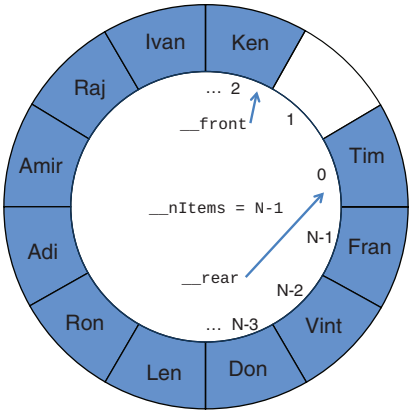


FIGURE 4-9 The `__rear` pointer wraps around

The last item inserted, `'Tim'`, is really placed at the beginning of the underlying array. This behavior can be somewhat counterintuitive because the sequence of items in the queue is no longer in one continuous array sequence [`__front`, `__rear`]. The front of the queue starts at index 2 and goes up to the max, and the rear is the single cell sequence [0, 0]. We say that these items are in *broken sequences*, or *noncontiguous sequences*. It may seem backward that `__rear` is now less than `__front`, but that's how circular arrays work. The design is to always start at `__front` and increment until you reach `__rear`, wrapping around to 0 as you pass the maximum size.

Now consider what happens when one more item is inserted in the queue of Figure 4-9. The item would go at cell 1, and `__rear` would point at an index 1 less than `__front` (which is 2). That is almost the same condition in which you started the empty queue, except that you've increased both `__rear` and `__front` by 1. To distinguish that condition from an empty queue, you need to know the count of items it contains. If `__rear` were always greater than `__front`, you could get the number of items from `__rear - __front + 1`. In the case of broken sequences, you could simply add the length of the two broken sequences, but you still need some information to distinguish an empty from a full queue when `__rear + 1 == __front`. That's why an explicit count of the items in the queue in the data structure simplifies the implementation.

Efficiency of Queues

As with a stack, items can be inserted and removed from a queue in O(1) time. This is based on avoiding shifts using a circular array.

Deques

There are several variations of a basic queue that you might find useful. A **deque** is a *double-ended queue*. You can insert items at either end and delete them from either end. The methods might be called `insertLeft()` and `insertRight()`, and `removeLeft()` and `removeRight()`.

You may also find this structure called a *dequeue*, but that can be a problem because some implementations like to call `insert()` as `enqueue()` and `remove()` as `dequeue()`. The shorter *deque* is preferred and is pronounced "deck" to help distinguish it from "dee cue."

If you restrict yourself to `insertLeft()` and `removeLeft()` (or their equivalents on the right), the deque acts like a stack. If you restrict yourself to `insertLeft()` and `removeRight()` (or the opposite pair), it acts like a queue.

A deque provides a more versatile data structure than either a stack or a queue and is sometimes used in container class libraries to serve both purposes. It's not used as often as stacks and queues, however, so we don't explore it further here.

Priority Queues

A **priority queue** is a more specialized data structure than a stack or a queue. Despite being more specialized, it's a useful tool in a surprising number of situations. Like an ordinary queue, a priority queue has a front and a rear, and items are removed from the front. In a priority queue, however, items are ordered by a priority value so that the item with the highest priority (which in many implementations is the lowest value of a key) is always at the front. When multiple items have the same priority, they follow the queue ordering, FIFO, so that the oldest inserted item comes first. As with both stacks and queues, no access is provided to arbitrary items in the middle of a priority queue.

Priority queues are useful for processing mail or messages. Faced with a big inbox, a reader might choose to put the messages into piles based on priority. All the messages requiring some kind of immediate action come first. There might be piles for immediate response, near-term response, long-term response, and no response. It might make sense to process some or all those piles in the order they arrived. If the messages are correspondence from a friend, going through the communications in chronological order would make the most sense, especially if they refer to things the friend said in previous messages. If the messages are purchase requests for tickets or some other scarce item, typically the first requests are processed before later requests, possibly with some different priority piles (for example, season ticket holders getting higher priority than others for playoff tickets).

Like stacks and queues, priority queues are often used as programmer's tools. In nearly every computer operating system, the programs are placed in a priority queue to be run. Many low-level operating system tasks get highest priority, especially those that need to determine what job needs to run next and how to respond to important input events. The operating system then takes the frontmost job from the priority queue to run next for a certain slice of time. When the time expires, the system puts that item back in the priority queue, waiting behind other jobs of the same priority. It might also put the job back in the priority queue after some delay in order to leave time open for other, lower-priority jobs to run. Jobs that complete normally—not because their time slice expired—do not get put back in the queue. We show another example of the use of a priority queue in programming techniques when we describe how to build a structure called a minimum spanning tree for a graph in Chapter 15, "Weighted Graphs."

Priority queues share aspects related to other data structures. They have the FIFO ordering behavior of queues, along with the sorting behavior needed for the different priority levels. As you saw in Chapter 3, "Simple Sorting," you can implement the sort algorithms in various ways that lead to different time performance. The stacks and queues you've seen have O(1) time performance for insert and removal. We will see whether that can be preserved as we add the prioritization behavior.

The PriorityQueue Visualization Tool

The PriorityQueue Visualization tool implements a priority queue with an array, in which the items are kept in sorted order. It's an *ascending-priority* queue, in which the item with the smallest key has the highest priority and is accessed with `remove()`. (If the highest-key item has highest priority, then the sequence of keys during removals decreases, and it would be a *descending-priority* queue.)

The minimum-key item is always at the right (highest index) in the array, and the largest item is always at index 0. Figure 4-10 shows the arrangement when the tool is started. Initially, there are no items in the 12-element queue, as indicated by the `nItems` index pointing to cell 0.

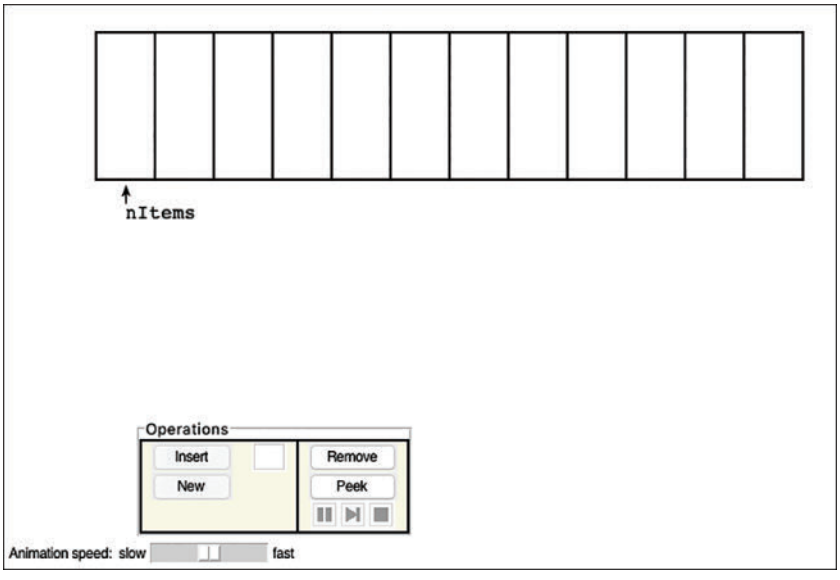


FIGURE 4-10 The PriorityQueue Visualization tool

The Insert Button

Try inserting an item by typing a number in the text entry box and selecting Insert. You use numbers for the item key because priorities are often numeric, and this approach makes it easy to draw the height of the item as proportional to the key. The first item is inserted in the leftmost cell, and the `nItems` index is advanced to 1.

Now try inserting another item. Because the priority queue must keep the items ordered, it behaves like the OrderedArray you saw in Chapter 2. If the item you insert has a lower key than the first, it will go to the right. If you insert a higher key, the items with lower keys will be shifted to the right to make room for the new item. The new item's key is compared with each item in the priority queue, starting with the one on the right. Items are shifted right until a cell becomes available where the new item can be inserted while preserving the key order.

Notice that there's no wraparound in this implementation of the priority queue. Insertion is slow out of necessity because the proper in-order position must be found, but deletion is fast. A wraparound implementation wouldn't improve the situation. Because there's no wraparound, all the items are in the leftmost, lower index, cells of the array. There's no need for two indices; all that's needed is an `nItems` index.

Try inserting the same key multiple times and observe the behavior. The visualization tool gives the new item a different color to make it easier to distinguish them. You can think of the colors as different data associated with the numeric keys. Where does it insert the new key? It always goes to the left of the other equal-valued keys. Putting it to the left preserves the FIFO queue order of those items.

The Remove Button

When you use the Remove button, the item to be removed is always at the right of the array, in the cell to the left of the `nItems` index, so removal is quick and easy. The item is copied to a `front` variable, and the `nItems` index is decremented, just like in a stack. No shifting or comparisons are necessary.

The Peek Button

You can peek at the minimum item (find its value without removing it) with the Peek button. The value is copied to an output box like the stack and queue.

The New Button

You can create a new, empty, priority queue with the New button. Like the other data structures, the size of the underlying array is the required argument and is limited to what fits easily on the display. The visualization tool animates the steps of recording the size, allocating the array, and setting the `nItems` index to 0. There's also a `__pri` attribute that we discuss later with the code.

Implementation Possibilities

The visualization program uses a particular implementation of priority queue that uses a single array for all the queued items. There are other possibilities, such as using separate queues for each priority or using a heap instead of an array (as discussed in Chapter 13, "Heaps"). Each one of these choices has different advantages.

The single array implementation has the advantage of simplicity at the cost of slowing down the insert or remove time. If you keep the array of items sorted by priority and arrival order, then time is needed to move items in the array during insertions to preserve

that order. If you leave the array of items unsorted (but maintain the arrival order), then insertion can still be fast, but removal is slow as you search for the highest-priority item to remove. After you remove an item from the unsorted list, you must shift items within the array to close the gap created by the removal. The movement of items in the array is expected to move half of the items, on average. That happens for either insertion or deletion, depending on whether you keep the array contents sorted. Knowing the number of moves means that either insertion or removal will be an O(N) operation. We examine a specific implementation to see how that works out.

Python Code for a Priority Queue

Listing 4-7 shows an implementation of a `PriorityQueue` class in Python that uses a single, sorted array (Python list). The choice of maintaining the internal array as sorted means that it won't need to manage the front and rear pointers of a circular array for the `Queue` as shown in Figure 4-5. Instead, it keeps a single array where the front of the queue is the last element, in order to quickly remove it. That means the rear of the queue is always at index 0, and it only manages an `__nItems` attribute somewhat like the `Stack` class.

LISTING 4-7 The `PriorityQueue.py` Program

```
# Implement a Priority Queue data structure using a Python list

def identity(x): return x                  # Identity function

class PriorityQueue(object):
   def __init__(self, size, pri=identity): # Constructor
      self.__maxSize = size                # Size of array
      self.__que = [None] * size           # Queue stored as a list
      self.__pri = pri                     # Func. to get item priority
      self.__nItems = 0                    # no items in queue

   def insert(self, item):                 # Insert item at rear of
      if self.isFull():                    # priority queue if not full
         raise Exception("Queue overflow")
      j = self.__nItems - 1                # Start at front
      while j >= 0 and (                   # Look for place by priority
            self.__pri(item) >= self.__pri(self.__que[j])):
         self.__que[j+1] = self.__que[j]   # Shift items to front
         j -= 1                            # Step towards rear
      self.__que[j+1] = item               # Insert new item at rear
      self.__nItems += 1
      return True

   def remove(self):                       # Return front item of priority
      if self.isEmpty():                   # queue, if not empty, & remove
         raise Exception("Queue underflow")
```

```
        self.__nItems -= 1                  # One fewer item in queue
        front = self.__que[self.__nItems]   # Store frontmost item
        self.__que[self.__nItems] = None    # Remove item reference
        return front

    def peek(self):                         # Return frontmost item
        return None if self.isEmpty() else self.__que[self.__nItems-1]

    def isEmpty(self): return self.__nItems == 0

    def isFull(self): return self.__nItems == self.__maxSize

    def __len__(self): return self.__nItems

    def __str__(self):                      # Convert pri. queue to string
        ans = "["                           # Start with left bracket
        for i in range(self.__nItems - 1, -1, -1): # Loop from front
            if len(ans) > 1:                # Except next to left bracket,
                ans += ", "                 # separate items with comma
            ans += str(self.__que[i])       # Add string form of item
        ans += "]"                          # Close with right bracket
        return ans
```

This class keeps another attribute, `__pri`, that stores a function, which can be called on any item in the queue to determine its priority. In the default case, `__pri` is the `identity()` function, which merely returns the record element itself. This is analogous to the key function used to get the keys of records stored in the `OrderedRecordArray` structure of Chapter 2. Typically, the client uses a particular field or set of fields to determine the priority among record structures, as explained in Chapter 2.

The `insert()` method of the `PriorityQueue` verifies that the queue isn't full and then searches where the new item should go within the queue based on its priority. It starts at the front of the queue and works toward the rear, by setting index `j` to point at the last filled cell and decrementing it. The loop compares the priority of the new item with that of item `j`. If item `j` has a higher or equal priority (lower value) than that of the new `item`, it shifts item `j` one cell to the right, like the `InsertionSort` algorithm in Chapter 3. After it finds an item that's lower in priority (a higher key) or the end of the queue, it can place the new item in the gap created by the shifts.

You might wonder why the `PriorityQueue` doesn't use binary search, especially after seeing how much it sped up finding items in ordered arrays in Chapter 2. We have chosen not to make that optimization here because it still requires a linear search among equal priority items in the priority queue to preserve FIFO order. The binary search could end on any of the equal priority items. The resulting code for `insert()` is shorter at the expense of having to do more key comparisons.

The `remove()` method verifies that the queue isn't empty, decrements the count of items in the queue, stores the frontmost item, clears the cell in the array, and then returns the noted item. The `peek()` method is even simpler because it doesn't have to remove the item before returning it.

The empty and full tests and length methods are based on the `__nItems` attribute (just like the `Queue` class). The `__str__()` method for converting the queue contents to a string also walks through the array from the front (index `__nItems - 1`) to the rear (index 0) to show the items in the same order used for the `Queue` class, which had the front of the queue on the left.

The `PriorityQueueClient.py` program in Listing 4-8 performs some basic tests on the priority queue implementation. It inserts tuples of the form (*priority, name*) into the `PriorityQueue` object, defining the first element of those tuples to be the priority.

LISTING 4-8 The `PriorityQueueClient.py` Program

```
from PriorityQueue import *

def first(x): return x[0]  # Use first element of item as priority

queue = PriorityQueue(10, first)

for record in [(0, 'Ada'), (1, 'Don'), (2, 'Tim'),
               (0, 'Joe'), (1, 'Len'), (2, 'Sam'),
               (0, 'Meg'), (0, 'Eva'), (1, 'Kai')]:
   queue.insert(record)

print('After inserting', len(queue),
      'persons on the queue, it contains:\n', queue)
print('Is queue full?', queue.isFull())

print('Removing items from the queue:')
while not queue.isEmpty():
   print(queue.remove(), end=' ')
print()
```

Like the client program for queues, after inserting the items, it shows the queue contents and then removes them one at a time, printing the results separated by spaces. The output follows:

```
$ python3 PriorityQueueClient.py
After inserting 9 persons on the queue, it contains:
 [(0, 'Ada'), (0, 'Joe'), (0, 'Meg'), (0, 'Eva'), (1, 'Don'),
  (1, 'Len'), (1, 'Kai'), (2, 'Tim'), (2, 'Sam')]
Is queue full? False
```

```
Removing items from the queue:
(0, 'Ada') (0, 'Joe') (0, 'Meg') (0, 'Eva') (1, 'Don') (1, 'Len')
(1, 'Kai') (2, 'Tim') (2, 'Sam')
```

Note that the `PriorityQueueClient.py` program inserted the items in a different order than the way they came out. All the items with priority 0 came out first, followed by the priority 1 and priority 2 items. Within each priority, the items are ordered by their insertion/arrival order (for example, `'Joe'` moved ahead of `'Don'` and `'Tim'` in the queue, because its priority was 0 as compared to 1 and 2, but remained after `'Ada'` because `'Joe'` arrived later).

Efficiency of Priority Queues

In the priority queue implementation we show here, insertion runs in O(N) time, whereas deletion takes O(1) time. That's a big performance difference. Can you think of ways to implement it that would take only O(log N) or O(1) time? We show how to improve insertion time with heaps in Chapter 13. In the special case where the set of priority values is known in advance, separate queues could be built for each one. Then the complexity of the overall priority queue would depend only on the time it takes to decide on which queue to apply the insert or remove method.

What About Search and Traversal?

We've discussed how to insert, remove, and look at one item in stacks, queues, and priority queues. For arrays and many of the data structures described later, the main concern is how they perform for the search operation. Shouldn't we examine how search and traversal work on stacks, queues, and priority queues?

The answer is: these structures are specifically designed for insertion and removal only. If an application needs to search for an item within the structure, then it's likely that some other data structure is a better choice. Stacks and (priority) queues serve to make specific orderings of items possible. At a minimum, you could use an array or an ordered array if you needed search and traversal operations. When you do use a stack or a queue and remove items until it is empty, then the algorithm effectively performs a destructive traversal, and the order of those items typically matches some requirement for the program. We look at such algorithms in the next section.

Parsing Arithmetic Expressions

We've introduced three different data storage structures and examined their characteristics. Let's shift gears now and focus on an important application that uses them. The application is **parsing** (that is, analyzing) arithmetic expressions such as 2+3 or 2×(3+4) or (2+4)×7+3×(9–5). The main storage structure you use is the stack. The `DelimiterChecker.py` program (Listing 4-4) shows how a stack could be used to check whether delimiters were formatted correctly. Stacks are used in a similar, although more complicated, way for parsing arithmetic expressions.

In some sense, this section should be considered optional. It's not a prerequisite to the rest of the book, and writing code to parse arithmetic expressions is probably not something you

need to do every day, unless you are a compiler writer or are designing domain-specific languages. Also, the coding details are more complex than any you've seen so far. Seeing this important use of stacks is educational, however, and raises interesting issues.

It's fairly difficult, at least for a computer algorithm, to evaluate an arithmetic expression in one pass through the string. It's easier for the algorithm to use a two-step process:

1. Transform the arithmetic expression into a different format, called *postfix* notation.
2. Evaluate the postfix expression.

Step 1 is a bit involved, but step 2 is easy. In any case, this two-step approach results in a simpler algorithm than trying to parse the arithmetic expression directly. For most humans, of course, it's easier to parse the ordinary arithmetic expression. We return to the difference between the human and computer approaches in a moment.

Before we delve into the details of steps 1 and 2, we introduce the new notation.

Postfix Notation

Everyday arithmetic expressions are written with an **operator** (+, –, ×, or /) placed between two **operands** (numbers, or symbols that stand for numbers). This is called **infix** notation because the operator is written in-between the operands. Typical arithmetic expressions are things like 2+2 and 4/7, or, using letters to stand for numbers, A+B and A/B.

In **postfix** notation (which is also called Reverse Polish Notation, or RPN, because it was invented by a Polish mathematician), the operator *follows* the two operands. Thus, A+B becomes AB+, and A/B becomes AB/. More complex infix expressions can likewise be translated into postfix notation, as shown in Table 4-2. One thing that might stand out in these translations is that there are no parentheses in the postfix expressions. We explain how the postfix expressions are generated in a moment.

Table 4-2 Infix and Postfix Expressions

Infix	Postfix
A+B–C	AB+C–
A×B/C	AB×C/
A+B×C	ABC×+
A×B+C	AB×C+
A×(B+C)	ABC+×
A×B+C×D	AB×CD×+
(A+B)×(C–D)	AB+CD–×
((A+B)×C)–D	AB+C×D–
A+B×(C–D/(E+F))	ABCDEF+/–×+

Besides infix and postfix, there's also a **prefix** notation, in which the operator is written before the operands: +AB instead of AB+. Also known as Polish notation, it is functionally like postfix but seldom used. The lambda calculus (λ calculus) and the Lisp programming language use prefix notation and are very important historically. For functional programming languages, prefix notation is very convenient. The lambda calculus is also the origin of the `lambda` keyword used for anonymous functions in Python.

Translating Infix to Postfix

The next several pages are devoted to explaining how to translate an expression from infix notation into postfix. This algorithm is fairly involved, so don't worry if every detail isn't clear at first. If you get bogged down, you may want to skip ahead to the sections "The InfixCalculator Tool" and "Evaluating Postfix Expressions"

To understand how to create a postfix expression, you might find it helpful to see how a postfix expression is evaluated; for example, how the value 14 is computed from the expression 234+×, which is the postfix equivalent of 2×(3+4). Notice that in this discussion, for ease of writing, we restrict ourselves to expressions with single-digit numbers (and sometimes variable names) for inputs. Later, we look at code to handle expressions with multidigit numbers and multicharacter variable names.

How Humans Evaluate Infix

How do you translate infix to postfix? Let's examine a slightly easier question first: How does a human evaluate a normal infix expression? Although, as we stated earlier, such evaluation is difficult for a computer, we humans do it fairly easily because of countless hours looking at similar expressions in math class. It's not hard to find the answer to 3+4+5, or 3×(4+5). By analyzing how you evaluate this expression, you can achieve some insight into the translation of such expressions into postfix.

Roughly speaking, when you "solve" an arithmetic expression, you follow rules something like this:

1. You read from left to right. (At least, we assume this is true. Sometimes people skip ahead, but for purposes of this discussion, you should assume you must read methodically, starting at the left.)
2. When you've read enough to evaluate two operands and an operator, you do the calculation and substitute the answer for these two operands and operator. (You may also need to solve other pending operations on the left, as we see later.)
3. You continue this process—going from left to right and evaluating when possible—until the end of the expression.

Table 4-3, Table 4-4, and Table 4-5 show three examples of how simple infix expressions are evaluated. Later, Tables 4-6, 4-7, and 4-8 show how closely these evaluations mirror the process of translating infix to postfix.

To evaluate 3+4–5, you would carry out the steps shown in Table 4-3.

Table 4-3 Evaluating 3+4–5

Item Read	Expression Parsed So Far	Comments
3	3	
+	3+	
4	3+4	
–	7	When you see the –, you can evaluate 3+4.
	7–	
5	7–5	
End	2	When you reach the end of the expression, you can evaluate 7–5.

You can't evaluate the 3+4 until you see whether an operator follows the 4 and what operator it is. If it's × or /, you need to wait before applying the + sign until you've evaluated the × or /. In this example, however, the operator following the 4 is a –, which has the same **precedence** (priority among operators) as a +, so when you see the –, you know you can evaluate 3+4, which is 7. The 7 then replaces the 3+4. You can evaluate the 7–5 when you arrive at the end of the expression, knowing that there are no more operators.

Because of precedence relationships, evaluating 3+4×5 is a bit more complicated, as shown in Table 4-4.

Table 4-4 Evaluating 3+4×5

Item Read	Expression Parsed So Far	Comments
3	3	
+	3+	
4	3+4	
×	3+4×	You can't evaluate 3+4 because × is higher precedence than +.
5	3+4×5	Because you don't know if there is some higher-precedence expression that follows, you simply put the 5 at the end.
End	3+20	When you see there are no more expressions, you can evaluate 4×5.
	23	Because there are still operators, you now evaluate 3+20.

Here, you can't add the 3 until you know the result of 4×5. Why not? Because multiplication has a higher precedence than addition. In fact, both × and / have a higher precedence than + and –, so all multiplications and divisions must be carried out before any additions or subtractions (unless parentheses dictate otherwise; see the next example).

Often, you can evaluate as you go from left to right, as in the preceding example. You need to be sure, when you come to an operand-operator-operand combination such as A+B, however, that the operator on the right side of the B isn't one with a higher precedence than the +. If it does have a higher precedence, as in this example, you can't do the addition yet. After you've read the 5 and found nothing after it, however, you know the multiplication can be carried out. Note that because multiplication has the highest priority among the four operators in this sample language, it doesn't matter whether a × or / follows the 5, and the multiplication could proceed without looking at the next input. You can't do the addition, however, until you've found out what's beyond the 5. When you find there's nothing more to read, you can go ahead and perform any remaining operations, knowing the highest precedence operators are on the right.

Parentheses are used to override the normal precedence of operators. Table 4-5 shows how you would evaluate 3×(4+5). Without the parentheses, you would do the multiplication first; with them, you do the addition first.

Table 4-5 Evaluating 3×(4+5)

Item Read	Expression Parsed So Far	Comments
3	3	
×	3×	
(	3×(	
4	3×(4	You can't evaluate 3×4 because of the parenthesis.
+	3×(4+	
5	3×(4+5	You can't evaluate 4+5 yet.
)	3×(4+5)	When you see the), you can evaluate 4+5.
	3×9	After replacing the parenthesized expression, you need to know if there's more to come with a higher precedence.
End	27	There isn't, so now you evaluate 3×9.

Here, you can't evaluate anything until you've reached the closing parenthesis. Multiplication has a higher or equal precedence compared to the other operators, so ordinarily you could carry out 3×4 as soon as you see the 4. Parentheses, however, have an even higher precedence than × and /. Accordingly, you must evaluate anything in parentheses before using the result as an operand in any other calculation. The closing parenthesis tells you that you can go ahead and do the addition. You find that 4+5 is 9 and substitute it. Finding that there are no more higher-precedence operators afterward, you can evaluate 3×9 to obtain 27.

As you've seen in evaluating an infix arithmetic expression, you go both forward and backward through the expression. You go forward (left to right) reading operands and operators. When you have enough information to apply an operator, you go backward, recalling two operands and an operator and carrying out the arithmetic.

Sometimes, you must defer applying operators if they're followed by higher-precedence operators or by parentheses. When this happens, you must apply the later, higher-precedence, operator first; then go backward (to the left) and apply earlier operators.

You could write an algorithm to carry out this kind of evaluation directly. As we noted, however, it's easier to translate into postfix notation first.

How Humans Translate Infix to Postfix

To translate infix to postfix notation, you follow a similar set of rules to those for evaluating infix. There are, however, a few small changes. You don't do any arithmetic. The idea is not to evaluate the infix expression, but to rearrange the operators and operands into a different format: postfix notation. The resulting postfix expression will be evaluated later.

As before, you read the infix from left to right, looking at each character in turn. As you go along, you copy these operands and operators to the postfix output string. The trick is knowing when to copy what.

If the character in the infix string is an operand, you copy it immediately to the postfix string. That is, if you see an A in the infix, you write an A to the postfix. There's never any delay: you copy the operands as you get to them, no matter how long you must wait to copy their associated operators.

Knowing when to copy an operator is more complicated, but it's the same as the rule for evaluating infix expressions. Whenever you could have used the operator to evaluate part of the infix expression (if you were evaluating instead of translating to postfix), you instead copy it to the postfix string.

Table 4-6 shows how A+B–C is translated into postfix notation.

Table 4-6 Translating A+B–C into Postfix

Character Read from Infix Expression	Infix Expression Parsed So Far	Postfix Expression Written So Far	Comments
A	A	A	
+	A+	A	
B	A+B	AB	
–	A+B-	AB+	When you see the –, you can copy the + to the postfix string.
C	A+B–C	AB+C	
End	A+B–C	AB+C–	When you reach the end of the expression, you can copy the –.

Notice the similarity of this table to Table 4-3, which showed the evaluation of the infix expression 3+4–5. At each point where you would have done an evaluation in the earlier table, you instead simply write an operator to the postfix output.

Table 4-7 shows the translation of A+B×C to postfix. This translation parallels that of Table 4-4, which covered the evaluation of 3+4×5.

Table 4-7 Translating A+B×C into Postfix

Character Read from Infix Expression	Infix Expression Parsed So Far	Postfix Expression Written So Far	Comments
A	A	A	
+	A+	A	
B	A+B	AB	
C	A+B×C	ABC	When you see the C, you can copy the ×.
	A+B×C	ABC×	
End	A+B×C	ABC×+	When you see the end of the expression, you can copy the +.

As the final example, Table 4-8 shows how A×(B+C) is translated to postfix. This process is similar to evaluating 3×(4+5) in Table 4-5. You can't write any postfix operators until you see the closing parenthesis in the input.

Table 4-8 Translating A×(B+C) into Postfix

Character Read from Infix Expression	Infix Expression Parsed so Far	Postfix Expression Written So Far	Comments
A	A	A	
×	A×	A	
(	A×(	A	
B	A×(B	AB	You can't copy × because of the parenthesis.
+	A×(B+	AB	
C	A×(B+C	ABC	You can't copy the + yet.
)	A×(B+C)	ABC+	When you see the), you can copy the +.
	A×(B+C)	ABC+×	After you've copied the +, you can copy the ×.
End	A×(B+C)	ABC+×	Nothing left to copy.

As in the numerical evaluation process, you go both forward and backward through the infix expression to complete the translation to postfix. You can't write an operator to the output (postfix) string if it's followed by a higher-precedence operator or a left parenthesis. If it is, the higher-precedence operator or the operator in parentheses must be written to the postfix before the lower-priority operator.

Saving Operators on a Stack

You'll notice in both Table 4-7 and Table 4-8 that the order of the operators is reversed going from infix to postfix. Because the first operator can't be copied to the output until the second one has been copied, the operators were output to the postfix string in the opposite order they were read from the infix string. That suggests a stack might be useful for handling the operators. A longer example helps illustrate how that could work. Table 4-9 shows the translation to postfix of the infix expression A+B×(C–D).

Table 4-9 Translating A+B×(C–D) into Postfix

Character Read from Infix Expression	Infix Expression Parsed So Far	Postfix Expression Written So Far	Stack Contents
A	A	A	
+	A+	A	+
B	A+B	AB	+
×	A+B×	AB	+×
(	A+B×(	AB	+×(
C	A+B×(C	ABC	+×(
–	A+B×(C–	ABC	+×(–
D	A+B×(C–D	ABCD	+×(–
)	A+B×(C–D)	ABCD–	+×(
	A+B×(C–D)	ABCD–	+×(
	A+B×(C–D)	ABCD–	+×
	A+B×(C–D)	ABCD–×	+
	A+B×(C–D)	ABCD–×+	

Here you see the order of the operands is +×– in the original infix expression, but the reverse order, –×+, in the final postfix expression. This happens because × has higher precedence than +, and –, because it's in parentheses, has higher precedence than ×. We need to track operators with lower precedence on the stack to be able to process them later. The last column in Table 4-9 shows the stack contents at various stages in the translation process.

Popping items from the stack allows you to, in a sense, go backward (right to left) through the input string. You're not really examining the entire input string, only the operators

and parentheses. They were pushed on the stack when reading the input, so now you can recall them in reverse order by popping them off the stack.

The operands (A, B, and so on) appear in the same order in infix and postfix, so you can write each one to the output as soon as you encounter it; they don't need to be stored on a stack or reversed. Changing their order would also cause issues with operators that lack the commutative property like subtraction and division; A – B ≠ B – A.

Translation Rules

Let's make the rules for infix-to-postfix translation more explicit. You read items from the infix input string and take the actions shown in Table 4-10. These actions are described in **pseudocode**, a blend of programming syntax and English.

In this table, the < and >= symbols are applied to the operator precedence, not numerical values. The *inputOp* operator has just been read from the infix input, while the *top* operator is popped off the stack. We use prec(*inputOp*) and prec(*top*) to mean the operator precedence of *inputOp* and *top*, respectively.

Table 4-10 Infix to Postfix Translation Rules

Item Read from Input	Action (Infix)
Operand	Write operand to postfix output string.
Open parenthesis (	Push parenthesis on stack.
Close parenthesis)	While stack is not empty: *top* = pop item from stack. If *top* is (, then break out of loop. Else write *top* to postfix output.
Operator (*inputOp*)	While stack is not empty: *top* = pop item from stack. If *top* is (, then push (back on stack and break. Else if *top* is an operator: If prec(*top*) >= prec(*inputOp*), output *top*. Else push top and break loop. Push *inputOp* on stack.
End of input	While stack is not empty: Pop stack and output item.

Convincing yourself that these rules work may take some effort. Tables 4-11, 4-12, and 4-13 show how the rules apply to the three sample infix expressions, similar to Tables 4-6, 4-7, and 4-8, except that the relevant rules for each step have been added. Try creating similar tables by starting with other simple infix expressions and using the rules to translate some of them to postfix.

Table 4-11 Translation Rules Applied to A+B–C

Character Read from Infix	Infix Parsed So Far	Postfix Written So Far	Stack Contents	Rule
A	A	A		Write operand to output.
+	A+	A	+	While stack not empty: (null loop) Push *inputOp* on stack.
B	A+B	AB	+	Write operand to output.
–	A+B–	AB		Stack not empty, so pop item +.
	A+B–	AB+		*inputOp* is –, *top* is +, prec(*top*) >= prec(*inputOp*), so output *top*.
	A+B–	AB+	–	Then push *inputOp*.
C	A+B–C	AB+C	–	Write operand to output.
End	A+B–C	AB+C–		Pop leftover item, output it.

Table 4-12 Translation Rules Applied to A+B×C

Character Read from Infix	Infix Parsed So Far	Postfix Written So Far	Stack Contents	Rule
A	A	A		Write operand to postfix.
+	A+	A	+	If stack empty, push *inputOp*.
B	A+B	AB	+	Write operand to output.
×	A+B×	AB		Stack not empty, so pop *top* +.
	A+B×	AB	+	*inputOp* is ×, *top* is +, prec(*top*) < prec(*inputOp*), so push *top*.
	A+B×	AB	+×	Then push *inputOp*.
C	A+B×C	ABC	+×	Write operand to output.
End	A+B×C	ABC×	+	Pop leftover item, output it.
	A+B×C	ABC×+		Pop leftover item, output it.

Table 4-13 Translation Rules Applied to A×(B+C)

Character Read from Infix	Infix Parsed So Far	Postfix Written So Far	Stack Contents	Rule
A	A	A		Write operand to postfix.
×	A×	A	×	If stack empty, push *inputOp*.
(	A×(	A	×(	Push (on stack.
B	A×(B	AB	×(	Write operand to postfix.
+	A×(B+	AB	×	Stack not empty, so pop item (.
	A×(B+	AB	×(	*top* is (, so push it and break.
	A×(B+	AB	×(+	Then push *inputOp*.
C	A×(B+C	ABC	×(+	Write operand to postfix.
)	A×(B+C)	ABC+	×(	Pop item, write to output.
	A×(B+C)	ABC+	×	Quit popping if (.
End	A×(B+C)	ABC+×		Pop leftover item, output it.

The InfixCalculator Tool

Before we look at the code, let's put these words into action and watch the process of converting infix expressions to postfix using a visualization tool. Follow the instructions in Appendix A to launch the InfixCalculator tool. The interface is simple: select the text entry box in the Operations area, type a numeric expression using infix, and select the Evaluate button. The animation of parsing the expression begins and looks something like Figure 4-11.

The expression being parsed in the figure, 2 * (3 + 4), is copied in the box at the top of the tool and then reduced as the characters are processed. (The full expression still appears in the text entry box below.) The figure shows that the first operand, 2, the multiplication operator, *, and the open parenthesis have already been pulled out of the expression and placed in the stack or queue. Each one of those strings is called a `token`. Looking at the "3" token corresponds to the fourth step detailed in Table 4-13 when the B operand is examined.

Because the current token "3" is a number, it's not an operator or a delimiter like those shown in the Operator Precedence table at the upper right. When the tool looks in that table, it finds that numbers have no precedence and writes `prec = None` and `delim = False` to indicate its status as an operand. The next (nonblank) character in the expression is a plus (+). It does have a precedence in the table, 4, and that means it's an operator.

As the animation runs, the stack contents build up in the stack on the left. Tokens that can be output as the postfix expression go in the queue to its right. The contents of this queue are used to make the postfix string later in the animation. Watch the stack grow as each operator is taken from the front of the expression. Pause the infix calculator and see whether you can predict what will happen next.

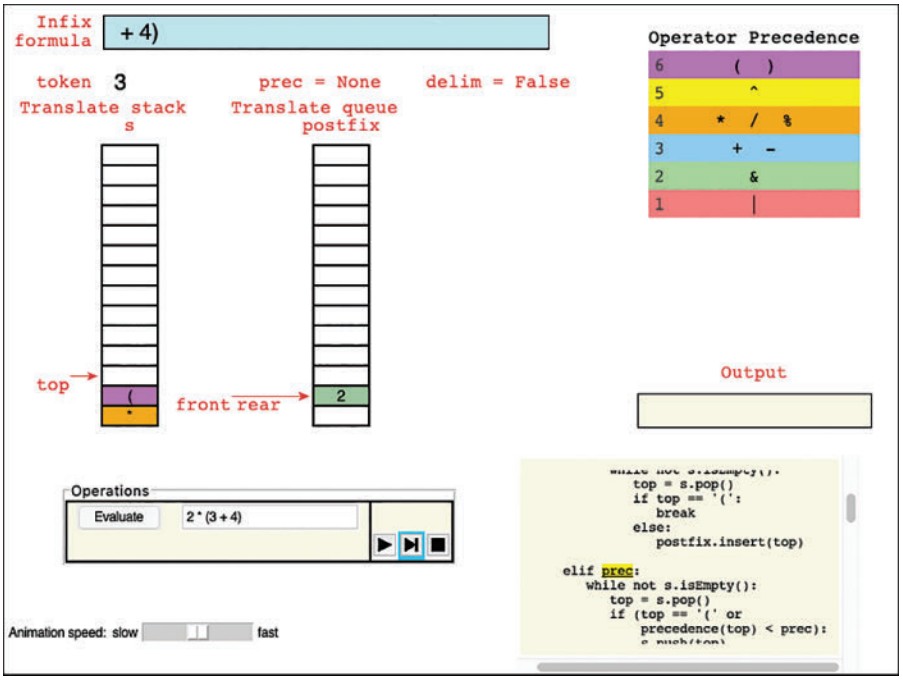


FIGURE 4-11 The InfixCalculator tool

The InfixCalculator shows the translation into posftfix, followed by the evaluation of the postfix. Before we look at how the evaluation works, let's look at the code for the parsing and translation. The visualization tool shows this code during the processing.

Python Code to Convert Infix to Postfix

Listing 4-9 shows the beginning of the `PostfixTranslate.py` program, which uses the rules from Table 4-10 to translate an infix expression to a postfix expression. This is a long program file, so we show it in two parts. The first part provides the processing of the input string, finding the individual tokens in the expression and determining the precedence of operators.

LISTING 4-9 Processing Tokens in `PostfixTranslate.py`

```
from SimpleStack import Stack
from Queue import Queue

# Define operators and their precedence
# We group single character operators of equal precedence in strings
# Lowest precedence is on the left; highest on the right
# Parentheses are treated as high precedence operators
operators = ["|", "&", "+-", "*/%", "^", "()"]
def precedence(operator): # Get the precedence of an operator
   for p, ops in enumerate(operators):  # Loop through operators
      if operator in ops:               # If found,
         return p + 1                   # return precedence (low = 1)
      # else not an operator, return None

def delimiter(character):               # Determine if character is delimiter
   return precedence(character) == len(operators)

def nextToken(s):                       # Parse next token from input string
   token = ""                           # Token is operator or operand
   s = s.strip()                        # Remove any leading & trailing space
   if len(s) > 0:                       # If not end of input
      if precedence(s[0]):              # Check if first char. is operator
         token = s[0]                   # Token is a single char. operator
         s = s[1:]
      else:                             # its an operand, so take characters up
         while len(s) > 0 and not (     # to next operator or space
               precedence(s[0]) or s[0].isspace()):
            token += s[0]
            s = s[1:]
   return token, s                      # Return the token, and remaining input
string
```

After importing the previous definitions of stacks and queues, the first statement of `PostfixTranslate.py` defines the `operators` that the program can process. We have expanded the number of operators that can be used. So far, we've only shown examples using the four most common operators, +–×/. This program adds several others available in Python. By putting these operator characters into an array of strings, we have grouped together operators with the same precedence and ordered the groups. Each array cell corresponds to a precedence level. You can look up the precedence of an operator by stepping through the array until you find the character in the string and returning the string's position in the array. Note that we used the asterisk * for the multiplication symbol ×, to be consistent with Python and other programming languages.

The `precedence()` function does the lookup, returning a number for the precedence or `None` if the character is not an operator. The operators are all the single-character operators that Python allows. Parentheses are included in the array of `operators` for convenience. They act like operators when breaking an expression up into tokens. A separate `delimiter()` function tests whether its argument is one of these highest-precedence characters.

A function breaks the input string into **tokens**. A token is a substring that corresponds to exactly one operator, operand, or delimiter in the input expression.

The `nextToken()` function goes through the input string, skipping over any whitespace, and finding the first token at the beginning of the string. It returns that token along with the rest of the string after the token, which is used for the subsequent call to `nextToken()`. The function checks whether a character is an operator or a delimiter by checking whether the `precedence` function returns a number for it (or `None` for operands). All the operators and delimiters in this example are single characters.

For operands, this program allows multicharacter (multidigit) tokens. The `while` loop steps through the input string, `s`, checking whether the first character is an operator or whitespace. If it is not, the first character is added to the output `token` and removed from the input string. After the loop finds either an operator, delimiter, whitespace, or the end of the input string, the token is complete.

The `PostfixTranslate()` function appears in Listing 4-10 and does the main work by allocating a stack to hold the operators (and left parentheses) and a queue to hold the postfix output. As you saw with the InfixCalculator Visualization tool, the operators build up in the stack and are then popped and moved to the queue. The queue holds the output postfix string with each operand or operator in its own cell. That enables the program to add some whitespace between elements in the output to make it more legible, especially for operands with more than one digit or character.

`PostfixTranslate()` loops over the input expression/formula using a **fencepost loop**—a loop that performs some work on the initial item before entering the main loop body to perform work on all the remaining loop items. In this case, it extracts the first token and then loops until there are no more tokens in the input expression/formula.

After determining whether the current token is an operand, operator, or delimiter, the `PostfixTranslate()` function applies the rules in Table 4-10. The `if delim … elif prec … else` statement separates the processing of the three types. The Python statements are very close to the pseudocode used to describe the rules. This is the heart of the algorithm, so study it closely. The `if delim: …` block handles the parentheses, pushing open parentheses on the stack and then popping off operators between the opening and closing parentheses after they are found.

LISTING 4-10 The `PostfixTranslate()` Function

```
def PostfixTranslate(formula):      # Translate infix to Postfix
   postfix = Queue(100)             # Store postfix in queue temporarily
   s = Stack(100)                   # Parser stack for operators
```

```
   # For each token in the formula  (fencepost loop)
   token, formula = nextToken(formula)
   while token:
      prec = precedence(token)      # Is it an operator?
      delim = delimiter(token)      # Is it a delimiter?
      if delim:
         if token == '(':           # Open parenthesis
            s.push(token)           # Push parenthesis on stack
         else:                      # Closing parenthesis
            while not s.isEmpty():  # Pop items off stack
                top = s.pop()
               if top == '(':       # Until open paren found
                  break
               else:                # and put rest in output
                   postfix.insert(top)

      elif prec:                    # Input token is an operator
         while not s.isEmpty():     # Check top of stack
             top = s.pop()
            if (top == '(' or       # If open parenthesis, or a lower
                precedence(top) < prec): # precedence operator
                s.push(top)         # push it back on stack and
                break               # stop loop
            else:                   # Else top is higher precedence
                postfix.insert(top) # operator, so output it
         s.push(token)              # Push input token (op) on stack

      else:                         # Input token is an operand
         postfix.insert(token)      # and goes straight to output

      token, formula = nextToken(formula) # Fencepost loop

   while not s.isEmpty():           # At end of input, pop stack
      postfix.insert(s.pop())       # operators and move to output

   ans = ""
   while not postfix.isEmpty():     # Move postfix items to string
      if len(ans) > 0:
         ans += " "                 # Separate tokens with space
      ans += postfix.remove()
   return ans

if __name__ == '__main__':
   infix_expr = input("Infix expression to translate: ")
   print("The postfix representation of", infix_expr, "is:",
         PostfixTranslate(infix_expr))
```

In the `elif prec: …` block, new operators from the input string are processed. It uses a loop to walk back through the stack looking for open parentheses or lower-precedence operators. The code shows a bit of an optimization from the loop's pseudocode for new operators:

While stack is not empty:

    top = pop item from stack

    If *top* is (, then push (back on stack and break

    Else if *top* is an operator:

        If prec(*top*) >= prec(*inputOp*), output *top*

        Else push *top* and break loop

The two highlighted phrases in the pseudocode perform the same operation because the open parenthesis is stored in the *top* variable. In the program, the two conditions are checked in the `if (top == '(' or precedence(top) < prec): …` statement. When `top` does not satisfy that condition, it must be an operator whose precedence is greater than or equal to that of the input operator in `token`. That's the only condition where the *top* operator is output by inserting it into the `postfix` queue.

After all the tokens have been processed in the first `while` loop, any remaining operators on the stack are popped and output to the postfix queue, reversing their order. That work happens in the final `while not s.isEmpty()` loop.

At the end, the `while not postfix.isEmpty()` loop creates the `ans` string, by concatenating the operands and operators in the `postfix` output queue separated by spaces. The spaces don't change the value of the expression but do make the string more readable.

After the function body of `PostfixTranslate()` is defined, the last section of the program is an `if` statement that is used to detect whether this file is being used as a program or as a module inside a bigger program. By testing whether the special variable, `__name__`, is set to the string '`__main__`', the Python interpreter can determine whether this file is being loaded as the main file to be executed or as part of an `import` statement inside another file. When `__name__` is '`__main__`' it's the main file to be executed, and this program prompts for an input infix expression to translate. It then prints that expression along with its postfix translation. The output looks like this:

```
$ python3 PostfixTranslate.py
Infix expression to translate: A+B-C
The postfix representation of A+B-C is: A B + C -
$ python3 PostfixTranslate.py
Infix expression to translate: A+B*C
The postfix representation of A+B*C is: A B C * +
```

```
$ python3 PostfixTranslate.py
Infix expression to translate: A*(B+C)
The postfix representation of A*(B+C) is: A B C + *
$ python3 PostfixTranslate.py
Infix expression to translate: A | B*C^D % E - F/G + H & J
The postfix representation of A | B*C^D % E - F/G + H & J is: A B C D ^ * E % F G /
- H + J & |
```

The last example uses all the operators the program knows about and shows how it interprets their precedence and ignores whitespace. The `PostfixTranslate.py` program doesn't check the input for errors. If you type an incorrect infix expression, such as one with consecutive operators or unbalanced parentheses, the program will provide erroneous output.

Experiment with this program. Start with some simple infix expressions and see whether you can predict what the postfix will be. Then run the program to verify your answer. Pretty soon, "a postfix Jedi, you will become." Note that you can also use the InfixCalculator tool to practice these transformations, but you won't be able to use letters as variable names.

Evaluating Postfix Expressions

As you can see, converting infix expressions to postfix expressions is not trivial. Is all this trouble really necessary? Yes, the payoff comes when you evaluate a postfix expression. Before we show how simple the algorithm is, let's examine how a human might carry out such an evaluation.

How Humans Evaluate Postfix

Figure 4-12 shows how a human can evaluate a postfix expression using visual inspection along with a pencil and paper. The postfix expression, 345+×612+/–, is shown in the gray rectangle. In this simplified example, operands can be only single-digit integers.

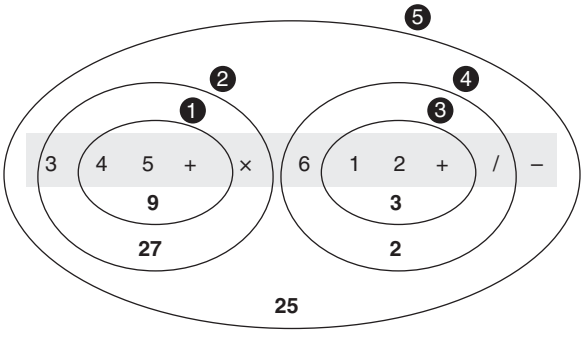


FIGURE 4-12 Visual approach to postfix evaluation of 345+×612+/–

Start with the first operator on the left and draw an oval around it plus the two operands to its immediate left. This is marked as step ❶ in the figure. Then apply the operator to these two operands—performing the actual arithmetic—and write down the result inside the oval. In the figure, evaluating 4+5 gives 9, which will be used in step ❷.

Now go to the next operator to the right and draw an oval around it, the oval you already drew, and the operand to the left of that. Apply the operator to the previous oval and the new operand and write the result in the new oval, labeled ❷. Here 3×9 gives 27. Continue this process until all the operators have been applied: 1+2 evaluates to 3 in step ❸, and 6/3 is 2, in step ❹. The answer is the result in the largest oval at step ❺: 27–2 is 25.

Rules for Postfix Evaluation

How do you write a program to reproduce this evaluation process? As you can see, each time you come to an operator, you apply it to the last two operands you've seen. Remembering what the `PostfixTranslate.py` program does (Listing 4-10) suggests that it might be appropriate to store the operands on a stack. The approach for postfix evaluation, however, differs from the infix-to-postfix translation algorithm, where *operators* were stored on the stack. You can use the rules shown in Table 4-14 to evaluate postfix expressions.

Table 4-14 Evaluating a Postfix Expression

Item Read from Postfix	Action Expression
Operand	Push it onto the stack.
Operator	Pop the top two operands from the stack and apply the operator to them. Push the result.

When you're done, pop the stack to obtain the answer. That's all there is to it. This process is the computer equivalent of the human oval-drawing approach of Figure 4-12.

Python Code to Evaluate Postfix Expressions

In the infix-to-postfix translation, symbols (A, B, and so on) were allowed to stand for numbers. This approach worked because you weren't performing arithmetic operations on the operands but merely rewriting them in a different format. Now say you want to evaluate a postfix expression, which means carrying out the arithmetic and obtaining an answer. The input must consist of actual numbers mixed with operators.

As shown in Listing 4-11, the `PostfixEvaluate.py` program imports the `PostfixTranslate` module and uses its functions to translate an infix expression to postfix before evaluating it. It makes use of the same `nextToken()` function from that module, but this time it's applied to the translated postfix string. The fencepost loop goes through each of the tokens in the postfix string. For operators, the left and right operands are popped off the stack, the operation is performed, and the result is pushed back on the stack. For operands, the string version of the operand is converted to an integer and pushed on the stack.

LISTING 4-11 The `PostfixEvaluate.py` Program

```
from PostfixTranslate import *
from SimpleStack import *
```

```
def PostfixEvaluate(formula):          # Translate infix to Postfix and
                                       # evaluate the result

   postfix = PostfixTranslate(formula) # Postfix string
   s = Stack(100)                      # Operand stack

   token, postfix = nextToken(postfix)
   while token:
      prec = precedence(token)         # Is it an operator?

      if prec:                         # If input token is an operator
         right = s.pop()               # Get left and right operands
         left = s.pop()                # from stack
         if token == '|':              # Perform operation and push
            s.push(left | right)
         elif token == '&':
            s.push(left & right)
         elif token == '+':
            s.push(left + right)
         elif token == '-':
            s.push(left - right)
         elif token == '*':
            s.push(left * right)
         elif token == '/':
            s.push(left / right)
         elif token == '%':
            s.push(left % right)
         elif token == '^':
            s.push(left ^ right)

      else:                            # Else token is operand
         s.push(int(token))            # Convert to integer and push

      print('After processing', token, 'stack holds:', s)

      token, postfix = nextToken(postfix) # Fencepost loop

   print('Final result =', s.pop())    # At end of input, print result

if __name__ == '__main__':
   infix_expr = input("Infix expression to evaluate: ")
   print("The postfix representation of", infix_expr, "is",
         PostfixTranslate(infix_expr))
   PostfixEvaluate(infix_expr)
```

The `PostfixEvaluate()` method includes a print statement that shows the stack contents after each token is processed, plus a final print statement showing the answer. The "main" part of the program (when `__name__ == '__main__'`) prompts the user for an infix expression, prints its postfix representation, and then evaluates it. Running the program produces a result like this:

```
$ python3 PostfixEvaluate.py
Infix expression to evaluate: 3*(4+5)-6/(1+2)
The postfix representation of 3*(4+5)-6/(1+2) is 3 4 5 + * 6 1 2 + / -
After processing 3 stack holds: [3]
After processing 4 stack holds: [3, 4]
After processing 5 stack holds: [3, 4, 5]
After processing + stack holds: [3, 9]
After processing * stack holds: [27]
After processing 6 stack holds: [27, 6]
After processing 1 stack holds: [27, 6, 1]
After processing 2 stack holds: [27, 6, 1, 2]
After processing + stack holds: [27, 6, 3]
After processing / stack holds: [27, 2.0]
After processing - stack holds: [25.0]
Final result = 25.0
```

Note this is the same calculation as shown in Figure 4-12, although the Python interpreter produced a floating-point number when it performed the division operation. The `PostfixEvaluate.py` program has not added any error checking on the input, so if you give it invalid expressions or expressions that contain names instead of numbers, the results will be wrong.

Experiment with the program and with the InfixCalculator tool. Try different expressions and check to see that they translate into postfix as you expect and verify the evaluation process. Use the animation controls to slow down or pause the calculator to see how the steps are carried out. Trying out different experiments can give you an understanding of the process faster than reading about it. One difference between the code and the InfixCalculator tool is that the tool creates floating-point numbers when division is used and pushes it on the stack. If that floating-point number creates an error, however, the tool will convert it to an integer and try again.

Summary

- Stacks, queues, and priority queues are data structures usually used to simplify common programming operations.
- In these data structures, only one data item can be accessed. They are designed to work for specific patterns in the order of access to the items.
- A stack allows access to the last item inserted: last-in, first-out (LIFO).

- The important stack operations are pushing (inserting) an item onto the top of the stack and popping (removing) the item that's on the top.
- A queue allows access to the first (oldest) item that was inserted: first-in, first-out (FIFO).
- The important queue operations are inserting an item at the rear of the queue and removing the item from the front of the queue.
- Stacks and queues typically do not have search or traverse operations because they are not databases.
- Stacks and queues typically do support a peek operation to examine the next item to be removed.
- A queue can be implemented using a circular array, which is based on a linear array in which the indices wrap around from the end of the array to the beginning.
- A deque is a two-ended queue that allows insertion and removal operations at both ends.
- The frontmost item in a priority queue is always the highest in priority and is the oldest in the queue of that priority.
- The important priority queue operations are inserting an item in sorted order based on priority and removing the oldest item within the highest-priority items.
- These data structures can be implemented with arrays or with other mechanisms such as linked lists.
- Ordinary arithmetic expressions are written in infix notation, so-called because the operator is written between the two operands.
- In postfix notation, the operator follows the two operands.
- Arithmetic expressions can be evaluated by translating them to postfix notation and then evaluating the postfix expression.
- A stack is a useful tool both for translating an infix to a postfix expression and for evaluating a postfix expression.

Questions

These questions are intended as a self-test for readers. Answers may be found in Appendix C.

1. Suppose you push the numbers 10, 20, 30, and 40 onto the stack in that order. Then you pop three items. Which one is left on the stack?
2. Which of the following is true?
   a. The Pop operation on a stack is considerably simpler than the Remove operation on a queue.

b. The contents of a queue can wrap around, while those of a stack cannot.

c. The top of a stack corresponds to the rear of a queue.

d. In both the stack and the queue, items removed in sequence are taken from increasingly higher index cells in the array.

3. What do LIFO and FIFO mean?

4. True or False: A stack or a queue often serves as the underlying mechanism on which an array data type is based.

5. As other items are inserted and removed, does a particular item in a queue move along the array from lower to higher indices, or higher to lower?

6. Suppose you insert 15, 25, 35, and 45 into a queue. Then you remove three items. Which one is left?

7. True or False: Pushing and popping items on a stack and inserting and removing items in a queue all take O(N) time.

8. A queue might be used appropriately to hold

a. the items to be sorted in an insertion sort.

b. reports of a variety of imminent attacks on the star ship Enterprise.

c. keystrokes made by a computer user writing a message.

d. delimiters in an algebraic expression being matched.

9. Inserting an item into a priority queue takes what Big O time, on average?

10. The term *priority* in a priority queue means that

a. the highest-priority items are inserted first.

b. the programmer must prioritize access to the underlying array.

c. the underlying array is ordered by the priority of the items.

d. the lowest-priority items are deleted first.

11. True or False: At least one of the methods in the `PriorityQueue.py` program in Listing 4-7 uses a linear search.

12. One difference between a priority queue and an ordered array is that

a. the lowest-keyed item cannot be removed as easily from the array as the lowest-priority item can from the priority queue.

b. the array must be ordered while the priority queue need not be.

c. the highest-priority item can be removed easily from the priority queue but the highest-keyed item in the array takes much more work to remove.

d. All the above.

13. Suppose a priority queue class is based on the `OrderedRecordArray` class from Chapter 2. This allows treating the priority as the key and provides a binary search capability. Would you need to modify the `OrderedRecordArray` class to maintain removals as an O(1) operation?

14. A priority queue might be used appropriately to hold

    a. patients to be picked up by an ambulance from different parts of the city.

    b. keystrokes made at a computer keyboard.

    c. squares on a chessboard in a game program.

    d. planets in a solar system simulation.

15. When parsing arithmetic expressions, it's convenient to use

    a. a stack for operators and a queue for operands and to evaluate postfix using another stack.

    b. a priority queue for the operators and operands and to evaluate postfix using a stack.

    c. a stack to transform infix expressions into prefix expressions that are then evaluated with a queue.

    d. a queue to transform the operators into postfix and then evaluate them using a priority queue.

Experiments

Carrying out these experiments will help to provide insights into the topics covered in the chapter. No programming is involved.

4-A Start with the initial configuration of the Queue Visualization tool. Alternately insert and remove items. Notice how the front and rear indices rotate around in the queue. Does the inserted item ever get stored in the same cell twice?

4-B Think about how you remember the events in your life. Are there times when they seem to be stored in your brain like a stack? Like a queue? Like a priority queue?

4-C Consider various processing activities and decide which of the data structures discussed in this chapter would be best to represent the process. Some activities are

- Handling order requests on an e-commerce website
- Dealing with interruptions to the task you're working on (like this homework)
- Folding and putting away clothes to be worn next week
- Folding rags to be used in cleaning
- Triaging patients that arrive at hospitals or clinics during a disaster with a limited number of treatment rooms and doctors
- Following the different paths in a maze at each decision point

4-D Using the InfixCalculator Visualization tool, try entering some valid numeric expressions and some with errors. Try putting in unbalanced parentheses and operators with missing operands. The tool doesn't try to find your errors, but if you were going to change it to report errors, how would you do it? At what point in the processing are the errors discoverable?

Programming Projects

Writing programs to solve the Programming Projects helps to solidify your understanding of the material and demonstrates how the chapter's concepts are applied. (As noted in the Introduction, qualified instructors may obtain completed solutions to the Programming Projects on the publisher's website.)

4.1 Revise the `Stack` class in `SimpleStack.py` shown in Listing 4-1 to throw exceptions if something is pushed on to a full stack, or popped off an empty stack. Write a test program that demonstrates that the revised class properly accepts items up to the original stack size and then throws an exception when another item is pushed.

4.2 Create a program that determines whether an input string is a palindrome or not, ignoring whitespace, digits, punctuation, and the case of letters. Palindromes are words or phrases that have the same letter sequence forward and backward. Show the output of your program on "A man, a plan, a canal, Panama." You should use a `Stack` as part of the implementation as was shown in the `ReverseWord.py` program in Listing 4-3.

4.3 Create a `Deque` class based on the discussion of deques (double-ended queues) in this chapter. It should include `insertLeft()`, `insertRight()`, `removeLeft()`, `removeRight()`, `peekLeft()`, `peekRight()`, `isEmpty()`, and `isFull()` methods. It needs to support wraparound at the end of the array, as queues do.

4.4 Write a program that implements a stack class that is based on the `Deque` class in Programming Project 4.3. This stack class should have the same methods and capabilities as the `Stack` class in the `SimpleStack.py` module (Listing 4-1).

4.5 The priority queue shown in Listing 4-7 features fast removal of the highest-priority item but slow insertion of new items. Write a program with a revised `PriorityQueue` class that has fast, O(1), insertion time but slower removal of the highest-priority item. Write a test program that exercises all the methods of the `PriorityQueue` class. Include the method that shows the `PriorityQueue` contents as a string and use it to show the contents of some test examples that include cases where the insertions do not occur in priority order.

4.6 Queues are often used to simulate the flow of people, cars, airplanes, transactions, and so on. Write a program that models checkout lines/queues at a store, using the `Queue` class (Listing 4-5). The system should model four checkout lines that initially start empty, labeled A, B, C, and D. Use a string to model the arrival and checkout completion events by using a lowercase letter to indicate a new customer arriving in one of the lines, and an uppercase letter to indicate a customer completing checkout

from the named line. For example, the string `"aababbAbA"` shows three people going into the A checkout line with two being processed, while four people enter the B checkout line and none are processed. When a customer is added, put a "person" in the queue by adding a string like `"C1"` to the queue where the number increases for each new person. Any nonalphabetic character in the string (such as a space or comma) signals that the current content of each of the queues should be printed. Use an `OrderedRecordArray` from Chapter 2 to store the `Queues` and their labels. Print error messages for queues that overflow or underflow. Show the output of your program on these strings:

```
aaaa,AAbcd,
abababcabc,Adb,Adb,Ca,
dcbadcbaDCBA-dddAcccBbbbCaaaD-
```

4.7 Extend the capabilities of `PostfixTranslate.py` in Listing 4-9 and `PostfixEvaluate.py` in Listing 4-11 to include the infix assignment operator, $A = B$. When evaluating expressions on the stack that reference variables, look up the assigned variable values before performing numeric operations. The assignment operator has the lowest precedence of any of the other operators. Unlike Python, the assignment operator itself should return the right-hand side value as its result. In other words, `A=3*2` should return a value of 6 (in Python, it returns `None`) with the side effect of binding A to 6. In this extended evaluator, references to variables must occur after they have been set (in some higher-precedence expression to the left of the reference). Your program should print an error message if the expression references variables that are not set. Variable values should be retrieved when an operator is trying to use the value for a calculation (not when they are pushed on the stack). Your program should display the contents of the stack as it processes each token. For example:

```
$ python3 project_4_7_solution.py
Infix expression to evaluate: (A = 3 * 2) * A
The postfix representation of (A = 3 * 2) * A is: A 3 2 * = A *
After processing A stack holds: [A]
After processing 3 stack holds: [A, 3]
After processing 2 stack holds: [A, 3, 2]
After processing * stack holds: [A, 6]
After processing = stack holds: [6]
After processing A stack holds: [6, A]
After processing * stack holds: [36]
Final result = 36
```

The first appearance of A defines the value for A as 6, and the second appearance references that value. Use the `OrderedRecordArray` class from Chapter 2 to store and retrieve records containing a variable name and value to implement this. Show the result of running your program on `'(A = 3 + 4 * 5) + (B = 7 * 6) + B/A'`.

CHAPTER 5

Linked Lists

IN THIS CHAPTER

- Links
- The LinkedList Visualization Tool
- A Simple Linked List
- Double-Ended Lists
- Linked List Efficiency
- Abstract Data Types and Objects
- Ordered Lists
- Doubly Linked Lists
- Circular Lists
- Iterators

In Chapter 2, "Arrays," you saw that arrays had certain disadvantages as data storage structures. In an unordered array, searching is slow, whereas in an ordered array, insertion is slow. In both kinds of arrays, deletion is slow. Also, the size of an array can't be changed after it's created. If a larger or smaller array is needed, a new array can be created, but all the items it contains need to be copied into the new structure, which slows things down.

In this chapter we look at a data storage structure that solves some of these problems: the **linked list**. Linked lists are probably the second most commonly used general-purpose storage structures after arrays.

The versatile linked list mechanism suits many kinds of general-purpose databases. It can also replace an array as the basis for other storage structures such as stacks and queues. In fact, you can use a linked list in many cases in which you use an array, unless you need frequent random access to individual items using an index.

Linked lists aren't the solution to all data storage problems, but they are surprisingly useful and conceptually simpler than some other popular structures such as trees. We investigate their strengths and weaknesses as we go along.

In this chapter we look at simple linked lists, double-ended lists, ordered lists, doubly linked lists, and circular lists. We also examine the idea of abstract data types (ADTs), see how stacks and queues can be viewed as ADTs, discuss how they can be implemented as linked lists instead of arrays, and introduce iterators as data structures for traversing other data structures.

Links

In a linked list, each data item is embedded in a **link**. A link represents one element of the overall list. Each link holds some data and a way to get to the next link in the list. This can be done in a couple of ways. Figure 5-1 shows the two most common methods with a sample list of ingredients (used perhaps as a shopping list). In both cases, the data for each link is a record. The example has records with fields that could be named: *ingredient type*, *subtype*, *amount*, and *unit*. The difference between the two styles of lists is in how the path to the next link is stored. In both cases, the path is stored as a **reference**—a pointer leading to another record. That reference can either be kept in a field in the data record itself or a field in a two-field structure designed as a general-purpose list.

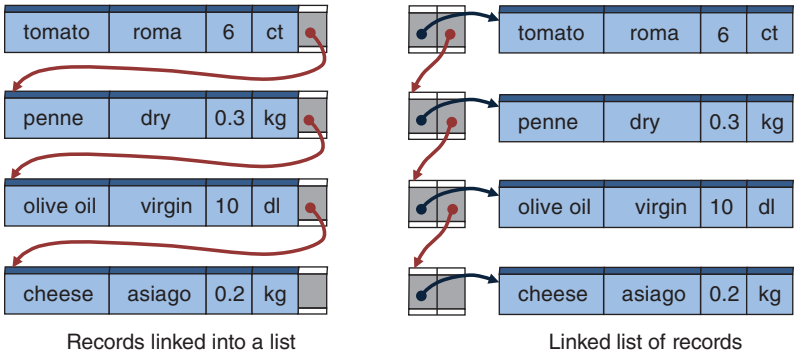


FIGURE 5-1 Different methods of representing links in a list

In the first method where records are linked into a list as shown on the left of Figure 5-1, each record has a field at the right called something like *next*. The last record in the list doesn't have a *next* link, so that field must get some special value that cannot be confused with a reference to another record. The figure shows it as an empty box. In Python, you typically use `None` to represent no link. In Java, you use `null`. For all but the last record, the next field has a reference to another link record, represented by a curving arrow in the figure.

The second method uses a separate, two-field record to represent each link. The first field of each link record is a reference to the ingredient record, and the second field is a reference to the *next* link record. Those different kinds of references are shown as different colored arrows in Figure 5-1. One advantage of this method is that the records used in the list don't need to have their own *next* link; that information is stored in the two-field link record. There are disadvantages too: you must follow a reference from the link record to get to the ingredient (data) record, and this method uses a little more memory to store the full list. Those disadvantages usually are not significant. Another advantage of the linked list of records shown on the right is that a single record could be part of more than one list; in the records linked into a list method (at the left of the figure), they can be part of only one list at any point in time.

With either method of representing the links, it's a good idea to have a separate object class for the linked list itself. Doing so allows the creation of empty lists—a data structure that can be modified to expand and contract to become empty—and enables utility functions like `len()` and `str()` in Python to be applied to object instances. We choose to use a `LinkedList` class to represent the overall list. The only thing to be stored in the `LinkedList` object is the reference to the first `Link` object. We could also store other attributes of the overall list, like its length, and we examine that topic later.

Let's look at the Python definitions of the classes `Link` and `LinkedList` that implement a linked list of records. Each `Link` has two fields: one for the data and one for the next link. The `LinkedList` needs only one attribute, and we name it `__first` because it points to the first `Link` object of the list, if there is one. We use the double underscore prefix, as usual, to indicate these are private fields. Listing 5-1 shows the constructor part of the definitions along with basic tests to see whether they are empty or the last link in the list.

LISTING 5-1 Constructors and Tests for Linked List Classes

```
class Link(object):                    # One datum in a linked list
   def __init__(self, data, next=None): # Constructor
      self.__data = data               # The datum for this link
      self.__next = next               # Reference to next Link

   def isLast():                       # Test if link is last in the chain
      return self.__next is None       # True if & only if no next Link

class LinkedList(object):              # A linked list of data elements
   def __init__(self):                 # Constructor
      self.__first = None              # Reference to first Link

   def isEmpty():                      # Test for empty list
      return self.__first is None      # True if & only if no 1st Link
```

The tests for `isLast()` and `isEmpty()` are nearly identical. The difference is their meaning in the context of the thing they represent—a link in the chain or the overall chain, respectively.

Because they are defined in Python, you don't have to specify the data types of the attributes in the classes, but they are important. In statically typed languages like Java, each attribute would need a declaration of its type. That's straightforward for the `__first` attribute of the `LinkedList` class because it is a `Link` object or `None`. Inside the `Link` class, the type to use for the `__next` attribute is more complicated. Here you need to refer to the very class that you're in the process of defining. This kind of class definition is sometimes called **self-referential** because it contains an attribute of the same type as itself.

More precisely, you don't want to store another entire `Link` object in the `__next` attribute of the `Link` class. If you did, then the self-referential definition would really become an endless nesting doll. We revisit that concept in the next chapter, "Recursion," but before we get there, we need to examine how references are stored and managed.

One of the reasons for specifying data types is so that the overall memory size of an object can be computed. The type information tells the compiler how many bytes to allocate when creating a new instance of the object. Self-referential class definitions would seem to need an infinite amount of memory if they were to contain themselves. Instead, they store a **reference** to the object. A reference is typically implemented by *storing the address of the object in memory*. That way, the amount of memory needed for the attribute is a fixed size because there is a maximum memory address.

The reference acts like the arrows in Figure 5-1, a path to find the referenced object. In some languages, these are called **pointers**, and following a reference or pointer is called **dereferencing**. There are some distinctions between memory addresses, pointers, and references, but we don't discuss those subtleties here. The important concept is that the `Link` records are connected into a chain by references that must be followed to traverse the whole structure. Compare that to the arrays where all the elements of the array were placed next to each other in memory so that you could find any of them by using an integer index.

References and Basic Types

You can easily get confused about references in the context of linked lists. The confusion can be compounded by dynamic typing in Python, which means you don't know the type of value that a variable references until runtime. Let's review how references work.

Every programming language has a set of **basic data types**, sometimes called **primitive** data types or *primitives*. These data types all have known, fixed sizes and usually take up one or two "machine words" at most. Integers and Boolean values are basic data types in almost every programming language. The Booleans, of course, have only two possible values, and the integer's range depends on the number of bits in a machine word (and some languages provide ways of specifying smaller or larger ranges than the standard machine word). Floating-point numbers are also normally stored with a fixed number of bits, based on the machine word size.

More complex data types like arrays, strings, records, and user-defined classes usually take more than one or two machine words of storage. For these structures, the programming language typically allocates memory for the size of the structure in a section of memory set aside for such items. When the structures are passed between functions, instead of copying the entire structure, only references to the structure are passed. Even when the object will not be passed to other functions, the programming language may still implement the object with a reference and a separate data storage location. Data types that are passed between functions as references are often called **reference data types**. They are distinct from the basic data types.

In Python, it's not obvious which data is stored as a primitive type and which is stored as a reference type. In other languages, programmers must be explicit about declaring whether a variable should be stored as primitive or as a reference to some other data type. That's one of the reasons why Python is popular as both a first programming language and an easy-to-use language; programmers don't need to manage references explicitly. When you write code that implements data structures, referencing and dereferencing become more of an issue because you are concerned with how long it will take to perform operations, how much space will be needed, and sometimes where the data will be stored.

One quick test to see whether data is being passed as a reference instead of by copying its value is to pass it to a function that modifies the value. For example, consider this small Python program:

```
def increment(a, b):
   a += 1
   b[0] += 1

x = 1
y = [1]
increment(x, y)
print(x, y)
```

What will the program print? The value of the variable `x` is initialized to 1, and the value of the variable `y` is a one-element list/array containing 1. When `increment()` is called, it increments the value of `a` and the first element of `b`. Will that change the values of the variables `x` and `y`? Do you think it will print `1 [1]`? Or maybe `2 [2]`? The answer is not obvious.

In the case of `x`, the answer is no; its value is not changed. The `print()` function will print the value 1 for `x`, because when `increment()` was running, the `a` variable stored *a copy of* 1, not the exact same 1 that was stored in `x`. When 1 was added to the value of `a`, the resulting 2 was stored back in `a` and did not change the copy stored in `x`.

In the case of `y`, however, the answer is different. The value of `y` is a Python list. After `increment()` runs, the value printed for `y` is `[2]`. Python passes lists as references. Any changes to the list by the `increment()` function affect the value in the caller's environment because both the function and the calling environment *share* the object being referenced.

This explanation overly simplifies what Python really does behind the scenes. In fact *all* Python variables are references to pieces of memory, and Python passes *all* arguments by reference. It creates new references for assignments to a plain parameter target, such as the `a` variable of `increment()`, without changing the old reference. The result is that you can think of the mechanism as if parameters had received their own copies of values of simple immutable types such as int, float, and bool.

Why is passing arguments so complicated? The short answer is that there are different situations where you'll want the different behaviors. Operating on copies of the data is quick when the data is small. It allows you to have different contexts in the different functions that won't interfere with one another. It makes it easier to understand what will happen in a function because nothing can change the values of its variables except

the instructions in the function itself. In other cases, it's faster to just overwrite the existing data, especially if it is very large and copying it would take a lot of time. Sharing the data structure does, however, lead to many bugs (errors) when programmers forget that some function can modify the data via a reference. As a general rule, you want to avoid designs that change (or **mutate**) the values in data that is shared by references. When you do design code to change values, make sure that the changes that can happen are well documented.

Back to the linked list, we need to use references to link together the elements. This approach solves the self-referential data structure paradox and enables the use of data structures that grow and shrink as the program runs. You can add `Link` objects as you need them, instead of trying to estimate how many are needed at the beginning of the program and allocating space for all of them in an array or some other structure.

When using references, you can think of them as their own data type. They are not integers even though most languages implement them internally as a kind of number. You can't do operations on them such as add two references together or check to see which one is larger. In general, there are no predefined values for them (like `False` and `True` for the Boolean type). There are a few special cases like the constant `null` in Java, which indicates the absence of a reference. The one operation you can do is check whether two references point to the exact same object. In Python, you can compare references with the `is` operator. Let's look at some examples:

```
$ python3
>>> x = 10
>>> lx = [10]
>>> ly = [5 + 5]
>>> print(x, lx, ly)
10 [10] [10]
>>> x is lx
False
>>> lx is lx
True
>>> lx is ly
False
>>> lx == ly
True
>>> lx[0] is ly[0]
True
>>>
```

The preceding transcript shows storing the integer 10 in the variable `x` and in two distinct one-element lists, `lx` and `ly`. In the case of `ly`, the 10 is the result of adding 5 and 5. The `print` function confirms these values. The program then tries comparing the variables using the `is` operator. As you might expect, `x` and `lx` do not reference the same object; one is an integer and the other is a list after all. It's unsurprising that `lx` refers to the same

object as `lx` because they are one and the same. Where it gets more interesting is where `lx` and `ly` are shown to be different objects because `lx is ly` evaluates to `False`. When you compare them using the equality operator, you find `lx == ly` to be `True`. Python's equality operator *follows the references* and compares the values they refer to. Because it finds a one-element list stored in both, and the elements in both lists are equal (both are 10), the result is `True`. The `is` operator, by contrast, *does not follow the references*; it just checks whether the references are the same.

If you look at how that data is stored in memory using references, the behavior of `is` and == becomes clearer. Figure 5-2 shows a simplified representation of the Python interpreter's memory after the assignment statements finish. Each of the three variables is shown with an arrow pointing to its value stored in the computer's memory. It appears that there are three copies of the value 10 in the diagram—though as we explain below, there is really just one copy of the 10. There are two copies of the list structure. The `is` operator compares where the arrows point and returns `True` if and only if they are identical. The == operator walks along the arrows and compares each pair of elements, returning `False` on the first unequal pair or `True` otherwise.

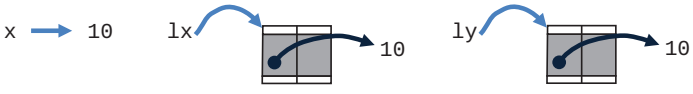


FIGURE 5-2 Comparing values in reference data types

The last line of the preceding transcript compares the first elements of the `lx` and `ly` lists using the `is` operator. They are both 10, so it may not seem surprising that the operator says they are the same object. It seems reasonable that two primitive data types with the same value compare as being the same using the `is` operator. Be careful, however, because this is a special case in Python. Small integers are stored with the exact same reference in Python, but larger integers might not be. For example:

```
>>> A = 650
>>> A == A
True
>>> A is A
True
>>> 2**A == 2**A
True
>>> 2**A is 2**A
False
>>>
```

To be safe, integers of all sizes should be compared for numerical equality using the equality operator, ==. When the `is` operator is used, however, the comparison is on the objects created to represent those big integers, and Python might create those big integer objects individually as needed. Big integers don't fit in a machine word or two. This means that it's tricky using arrows to point to integers or other types like in Figure 5-2. What may seem to be a simple primitive might be a more complex structure.

Relationship, Not Position

Let's examine one of the major ways in which linked lists differ from arrays. In an array, each item occupies a particular position. This position can be directly accessed using an index number. It's like a row of houses with sequential addresses: you can find a particular house using its numeric address. If the houses are equally spaced apart and the numbering sequence is equally spaced, you could know exactly how far to travel from the first house in the row to get to a particular numbered house.

In a list, the only way to find a particular element is to follow along the chain of elements. You must visit each of the first elements to get to the one you want. There are two ways you might know which element you want: by its position or by some key. An example of knowing its position is like saying, "It's the fifth house on the left." You don't know exactly how far down the street it is, but you can count the houses as you go and turn left after reaching a count of five. An example of knowing a house by a key would be by giving a description that distinguishes it from the other houses, for example, the stone house with a green door.

Another example of knowing something by its key is describing someone by their face or name. Imagine that you are at a huge party with hundreds of people. You are looking for an acquaintance, Zev, that you think is attending. Maybe you ask Raj where Zev is. Raj doesn't know, but he thinks Anna might know, so you go and ask Anna. Anna saw Zev go outside with Dimitri. You happen to have Dimitri's phone number, so you call him. He answers but says that Zev went back inside a few minutes ago, saying he was looking for Aoife. So, you start to look for, … you get the idea. When you start down the chain, you don't even know whether the item you seek is going to be there.

These kinds of chains happen in computer applications too. If you're trying to trace the path a message followed through the network, you start at the node where the message ended (or started) and find a record indicating where the message came from (or went to). You then visit that node and look for the next record showing the message's origin (destination). You must follow all the individual links to reconstruct the full path of the message. These kinds of systems share the property that you can't access a data item directly; you must use relationships between the items to locate a specific item. You start with the first item, go to the second, then the third, until you find what you're looking for, or the end of the relationships.

The LinkedList Visualization Tool

Let's look in more detail at how linkages are made in a list data structure. The LinkedList Visualization tool shows all the basic operations. You can launch it with the command `python3 LinkedList.py`, following the instructions in Appendix A, "Running the Visualizations." Figure 5-3 shows how the LinkedList Visualization tool looks when it starts. Initially, the list is empty with only a small red circle reserved for where the first link reference will go. The box represents an instance of the `LinkedList` object with its one attribute, as described in Listing 5-1.

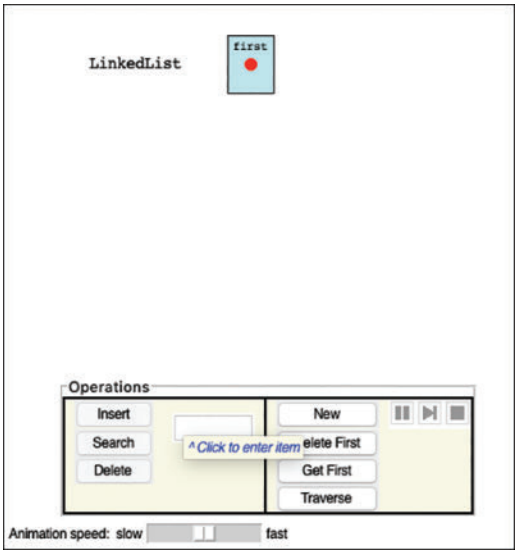


FIGURE 5-3 The LinkedList Visualization tool with an empty list

To make the first `Link` object, you type a word in the text entry area and select the Insert button. That action creates a colored rectangle holding the word, which represents the `Link` object and its data. The link is the only one in the list, so its pointer is `None`, represented by the red dot. The `LinkedList` object now has a reference to the `Link` stored in its `first` attribute.

Inserting a couple of words using the Insert button could make the start of a shopping list like the one in Figure 5-4. Each `Link` except the last one has a reference to the next `Link`. (The small `link` arrow at the top is left over from the insert operation that just finished.)

As you insert items, watch the animation closely. The new `Link` begins outside of the list. Its next pointer points to the first item in the list (or nothing when the `LinkedList` is empty). The insertion happens by changing the `first` pointer of the `LinkedList` to point at the new `Link`. The visualization tool then rearranges the linked items to make the familiar chain-like structure. There is no change happening to the data structure in memory when the tool rearranges the items; that just makes the structure easier to see in the visualization.

In this version of a linked list, new links are always inserted at the beginning of the list. This is the simplest approach, although sometimes there are operations that need to place items elsewhere in variations on the basic list, as we show later.

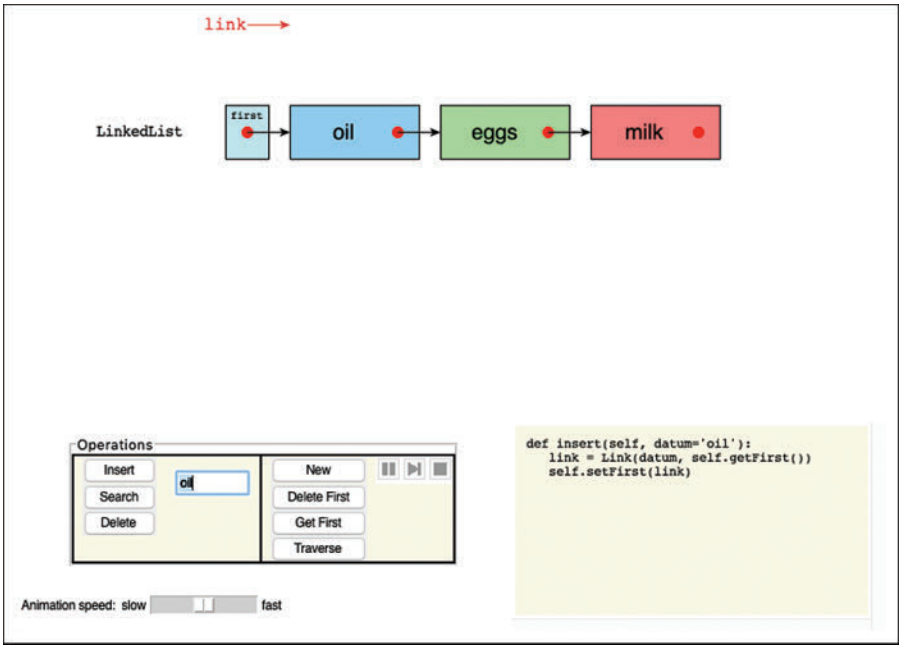


FIGURE 5-4 A linked list after inserting some items

The Search Button

The Search button allows you to find a link with a specified key value. Type the value of an existing link (or click it) and select Search. The operation begins by placing a `link` pointer at the first `Link`. The `link` pointer advances down the links until it points at a link whose key matches the `goal` key you entered. The object is highlighted, and the data is copied to an output box. If you search for a key that's not in the list, the `link` pointer advances past the last `Link` in the list and ends up pointing at nothing (`None`). The message at the bottom informs you whether the key was found.

The Delete Button

You can also delete a `Link` with a specified key. Type in the value of an existing link and then select Delete. This time, two arrows appear and move along the list, somewhat like an inchworm, looking for the goal link. When the `link` arrow arrives at the link with the `goal` key, the advancing stops, and the `previous` `Link`'s pointer is reconnected to the `goal` link's successor. After the `previous` pointer is updated, the `goal` link is no longer in the list and is moved away. If the `goal` link isn't found, the animation stops with an exception.

The visualization tool moves the `Links` around on the display, sometimes stretching and bending the arrows for the pointers. The length and shape of the visualization arrows don't correspond to anything in the program. They only serve to show what objects reference other objects.

The New Button

As with the other data structures, you create a new, empty linked list by selecting the New button. Unlike those other structures, you don't need to specify a size for the structure. The linked list is not based on an array; links are allocated and discarded as needed.

The Other Buttons

The Delete First button deletes the first link in the list, regardless of its key. This is somewhat like the Delete Rightmost operation that you saw with arrays. A Get First button gets the data value stored in the first link, and a Traverse button walks through all the links in the list, copying their values to an output box.

A Simple Linked List

Let's look at the methods needed to implement the basic operations in the `Link` and `LinkedList` classes in Python.

We expand on the constructors and simple tests shown in Listing 5-1, add methods to access the components, and add a helper method to display a `Link` as a string in Listing 5-2. This listing introduces `getData()` and `setData()` methods to access the data stored in the `Link`. Another alternative would be to make the `__data` attribute (or field) a public attribute of the `Link` class that calling programs could set. That would allow for direct access to the instance attribute, which might not be desired in some cases, such as enforcing some constraint on the kinds of data that can be added. The `setNext()` method illustrates the idea of a constraint by enforcing that the `__next` attribute can be set to only a reference to a `Link` object or the special "no next" value, `None`. This listing retains the `isLast()` method that defines the convention that a value of `None` in the `__next` field means there are no subsequent links. Lastly, the listing defines a `__str__()` method for showing links as strings. It simply applies the string conversion process to whatever datum is stored in the `Link`.

LISTING 5-2 Basic Methods for `Link` Objects

```
class Link(object):            # One datum in a linked list
   def __init__(self, datum, next=None): # Constructor
      self.__data = datum      # The datum for this link
      self.__next = next       # Reference to next Link
```

```
   def getData(self):          # Return the datum stored in this link
      return self.__data

   def setData(self, datum):   # Change the datum in this Link
      self.__data = datum

   def getNext(self): return self.__next # Return the next link

   def setNext(self, link):    # Change the next link to a new Link
      if link is None or isinstance(link, Link): #Must be Link or None
         self.__next = link
      else:
         raise Exception("Next link must be Link or None")

   def isLast(self):           # Test if link is last in the chain
      return self.getNext() is None  # True if & only if no next Link

   def __str__(self):          # Make a string representation of link
      return str(self.getData())
```

The Basic Linked List Methods

In Listing 5-3 the `LinkedList` class is updated with the same kind of accessor methods for its one attribute, `__first`. Along with the get and set methods for the first link, it defines `getNext()` and `setNext()` methods as synonyms whose utility will show up a little later. A new method, `first()`, returns the first item in the linked list. It's analogous to the `peek()` method for queues. It raises an exception if the list is empty. Note that to get to the data, it dereferences the `__next` attribute to get a `Link` and then uses that to get that link's data. This is a common operation on linked lists.

LISTING 5-3 Basic Methods for `LinkedList` Objects

```
class LinkedList(object):      # A linked list of data elements
   def __init__(self):         # Constructor
      self.__first = None      # Reference to first Link

   def getFirst(self): return self.__first # Return the first link

   def setFirst(self, link):   # Change the first link to a new Link
      if link is None or isinstance(link, Link): # It must be None or
         self.__first = link   # a Link object
      else:
         raise Exception("First link must be Link or None")
```

```
   def getNext(self): return self.getFirst()    # First link is next
   def setNext(self, link): self.setFirst(link) # First link is next

   def isEmpty(self):          # Test for empty list
      return self.getFirst() is None  # True iff no first Link

   def first(self):            # Return the first item in the list
      if self.isEmpty():       # as long as it is not empty
         raise Exception('No first item in empty list')
      return self.getFirst().getData() # Return data item (not Link)
```

Traversing Linked Lists

To make the linked list useful, you need to be able to insert, delete, search, and traverse the elements it holds. Traversal is perhaps the easiest and is the essence of helper methods, like getting the length of the list or creating a string representation of its contents. Listing 5-4 shows the traversal methods.

LISTING 5-4 Traversal Methods of `LinkedList`

```
class LinkedList(object):       # A linked list of data elements

… (basic definitions shown before) …

   def traverse(self,           # Apply a function to all items in list
                func=print):    # with the default being to print
      link = self.getFirst()    # Start with first link
      while link is not None:   # Keep going until no more links
         func(link.getData())   # Apply the function to the item
         link = link.getNext()  # Move on to next link

   def __len__(self):           # Get length of list
      l = 0
      link = self.getFirst()    # Start with first link
      while link is not None:   # Keep going until no more links
         l += 1                 # Count Link in chain
         link = link.getNext()  # Move on to next link
      return l

   def __str__(self):           # Build a string representation
      result = "["              # Enclose list in square brackets
      link = self.getFirst()    # Start with first link
```

```
    while link is not None:   # Keep going until no more links
       if len(result) > 1:    # After first link,
          result += " > "     # separate links with right arrowhead
       result += str(link)    # Append string version of link
       link = link.getNext()  # Move on to next link
    return result + "]"       # Close with square bracket
```

The `traverse()` method applies a function to each element of the list. It starts by setting a `link` variable to point at the first `Link` in the `LinkedList`. This is a reference to an object. The `while` loop then steps through the list as long as that next `link` is not `None`, which indicates the end of the list. If there is a `Link` object, it applies the desired function (`print` by default) to the data in it and then moves on to that `Link`'s next reference. Note the loop could make use of the `isLast()` method to detect the end of the list. Because the method still needs to apply the function to the last `Link`, it's shorter to write the loop by testing the `link` variable.

The `__len__()` method (which enables Python's `len()` function to work on `LinkedList` objects) follows the same outline as `traverse`. It keeps the length of the list in a variable, `l`, and increments that value in the body of the `while` loop. As it sets the `link` to each subsequent `Link` object, the count of the number of objects goes up by 1. When there are no more `Link` objects, it returns the count stored in `l`.

Similarly, the `__str__()` method initializes a string variable called `result` with a left bracket. The `while` loop is the same. On each pass through the loop body, a separator string, `" > "`, is added if the `Link` object is not the first one. Then the string representation of the `Link` is added to the `result`. At the end of the list, a final closing right bracket is added to the `result`, and it is returned.

Insertion and Search in Linked Lists

The insertion and searching methods use similar loops when looking for an item with a matching key in the list, as shown in Listing 5-5. To handle complex data within the linked list, it's convenient to use a function to extract the key from each link's data item. The code uses `identity()` as the default function to get the key for each element of the list.

LISTING 5-5 Insertion and Searching Methods for `LinkedList`

```
def identity(x): return x          # Identity function

class LinkedList(object):          # A linked list of data elements

… (other definitions shown before) …
```

```
def insert(self, datum):        # Insert a new datum at start of list
   link = Link(datum,           # Make a new Link for the datum
               self.getFirst()) # What follows is the current list
   self.setFirst(link)          # Update list to include new Link

def find(                       # Find the 1st Link whose key matches
      self, goal, key=identity): # the goal
   link = self.getFirst()       # Start at first link
   while link is not None:      # Search until the end of the list
      if key(link.getData()) == goal:  # Does this Link match?
         return link            # If so, return the Link itself
      link = link.getNext()     # Else, continue on along list

def search(                     # Find 1st item whose key matches goal
      self, goal, key=identity):
   link = self.find(goal, key)  # Look for Link object that matches
   if link is not None:         # If found,
      return link.getData()     # return its datum

def insertAfter(                # Insert a new datum after the first
      self, goal, newDatum,     # Link with a matching key
      key=identity):
   link = self.find(goal, key)  # Find matching Link object
   if link is None:             # If not found,
      return False              # return failure
   newLink = Link(              # Else build a new Link node with
      newDatum, link.getNext()) # new datum and remainder of list
   link.setNext(newLink)        # and insert after matching link
   return True
```

For inserting values in the list, the simplest thing to do is insert them at the front of the list. Like pushing a value on a stack, the value will become the first item (in the first `Link`) of the list. The `insert()` method does that by creating a new `Link` object whose data is set to the `datum` being inserted. The interesting, and perhaps confusing, step is the initialization of the new `Link` object's `__next` pointer. It gets set to the value of the `LinkedList`'s first pointer. The reason is that the original `LinkedList` has all the elements that are going to come after the newly created `Link`. The final step is to overwrite the `LinkedList`'s `__first` pointer so that it points to the newly created `Link`. The processing steps are illustrated in Figure 5-5.

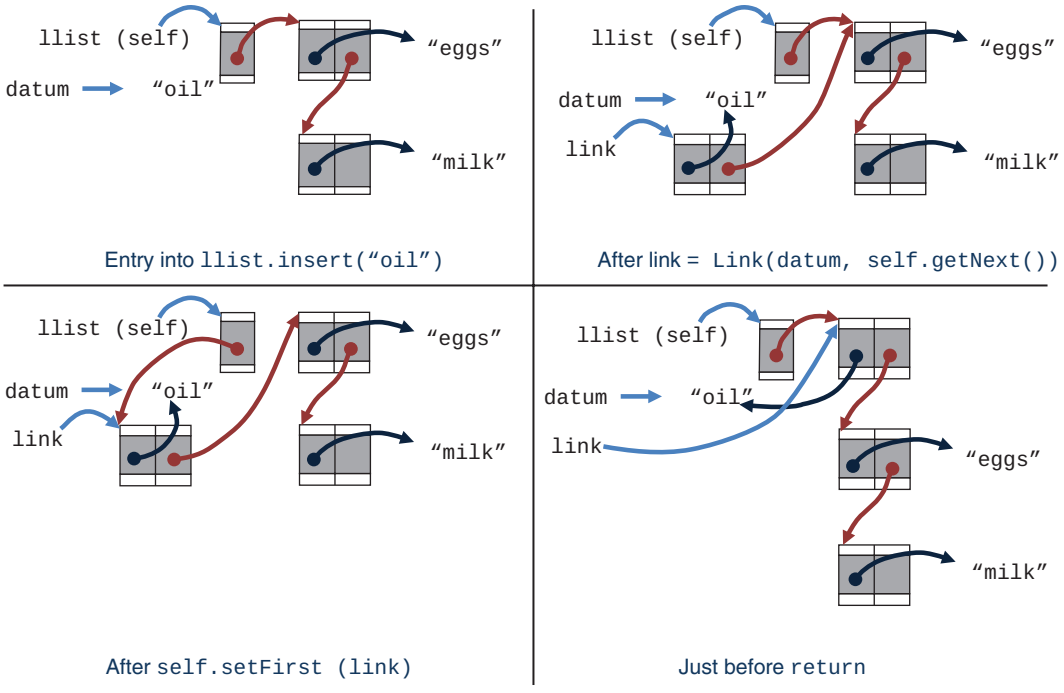


FIGURE 5-5 Execution of `insert()`

The example in Figure 5-5 shows the ingredient "oil" being added to a list that already contains "eggs" and "milk". The first panel shows the state of the arguments to the method on entry into `insert("oil")`. The `self` variable is the `LinkedList` to be modified, `llist` in this case. The `LinkedList` object has one field, `__first`, that points to the first `Link` in the list. The first `Link` is the one for "eggs" and it's followed by the `Link` for "milk". The second panel shows the creation of the new `Link` object stored in the `link` variable. The data is set to the input `datum`, and the `__next` pointer is set to the original `LinkedList`'s `__first` value. The third panel shows how the original `LinkedList`'s `__first` pointer is set to the newly created `Link`. The last panel shows the state right before returning from the call to `insert()` with the `Link` for "oil" clearly positioned at the front of the list, a simple rearrangement of the `Link` structures in the third panel.

The example in Figure 5-5 can also be animated in the LinkedList Visualization tool. The figure is a bit more detailed about the state of all the variables than the visualization is. The figure also shows the string data items being referenced through a pointer from each `Link`. The string reference is used in the computer but is difficult to see in the code. Try inserting the same items in a list and using the step button to step through each line of the code for one of the insertions.

If insertion and deletion to the linked list could only occur at the first position in the list, the data structure would behave like a stack where the last element put in is always the first element taken out. To be able to do things like insert an element at a particular position within an ordered list, you need to be able to specify and find positions within the list where the change should be made. For that, you need to find elements with a particular key, which is what the `find()` method does.

Just like the traversal operations, the `find()` method steps through the `Link`s of the list. For each link, the `find()` method shown in Listing 5-5 applies the `key()` function to the data stored there. If the value of the extracted key matches the `goal` key, then the `Link` containing that key is returned. If the `while` loop ends, then none of the `Link`s had a matching key, and the result is `None` (because Python returns an implicit `None` when the body of the function finishes). It's important to note that `find()` always returns a `Link` (really a reference to a `Link`) or `None`. It doesn't return an integer index for where the `Link` was found in the list like the `find()` method for arrays. Why not? If it did, then a subsequent operation to insert or delete a `Link` at that position would have to repeat stepping through the links to find the right spot. Returning the reference saves many steps.

To get the data stored at the link with a particular key, you use the `search()` method. This method takes the same arguments as the `find()` method but returns the *data in the* `Link` rather than a *reference to the* `Link`. This subtle difference is very important. Some operations need a position, and some need the data at the position. The `search()` method simply calls the `find()` method, and if it returns a pointer to a `Link`, it dereferences the pointer to get the value. Otherwise, it too returns `None`.

The next method in Listing 5-5, `insertAfter()`, provides a way to insert a new datum immediately after the first `Link` in the list that has a key matching a particular `goal` key. It makes use of the `find()` method to discover whether there's a `Link` with a matching key. If none is found, it returns `False` to indicate that. Otherwise, it creates a new `Link` to splice into the list right after the one with the matching key.

Just like the `insert()` method, the new `Link` for `insertAfter()` contains the provided datum, and its `__next` pointer is set to what should follow. In this case, it's the linked list that follows the `Link` with the matching key, as illustrated in Figure 5-6. The figure shows the same datum, "oil", being inserted in a list, but this time it is to be inserted after "eggs". The `link` variable stores the output of the `find()` call. The `newLink` is created in the third panel with "oil" for its data and the list after the "eggs" `Link` as the list that follows. After the `__next` pointer for the insertion `link` is updated to point at the `newLink`, the linked list is altered into its final form shown in the fourth panel. The routine returns `True` to indicate that it found the goal key and inserted the new `Link`. Note that the links have been rearranged in the bottom right of the figure to better show the new list structure.

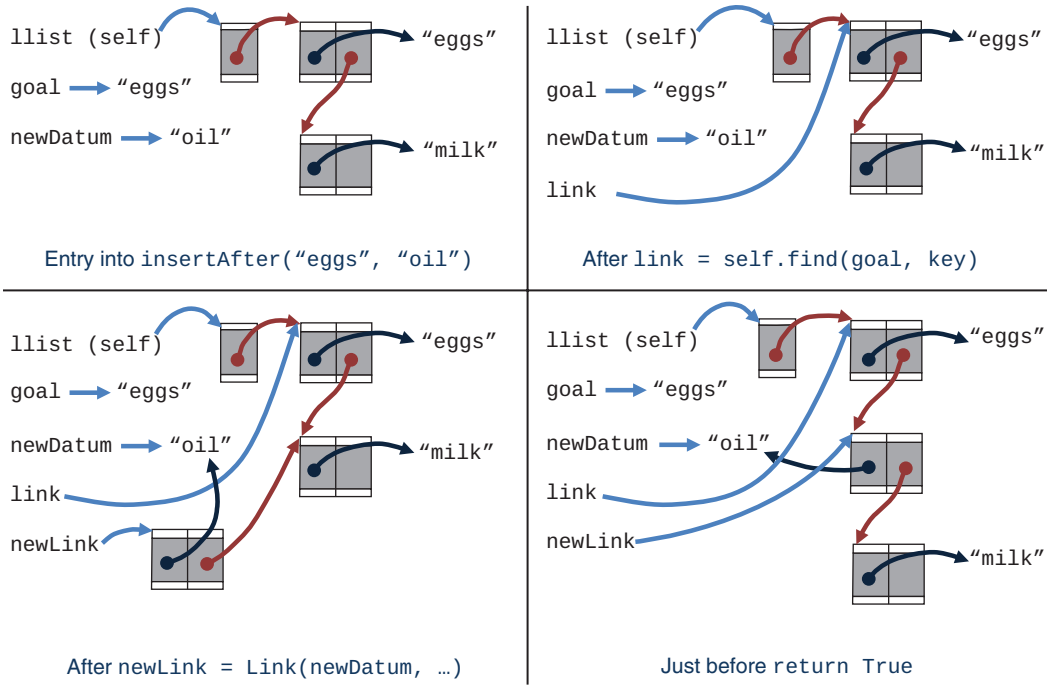


FIGURE 5-6 Inserting "oil" after "eggs" in a linked list

Deletion in Linked Lists

The final methods needed for a basic `LinkedList` structure are the ones for deletion. Deleting the first item is another quick operation that simply changes a couple of pointers. Deleting from the middle means finding an item by its key.

The `deleteFirst()` method shown in Listing 5-6 deletes the first link in the list, after checking that the list is not empty. The steps in the code are almost as simple as those for the `insert()` method. It stores a pointer to the first `Link` in the `first` variable. This is necessary to retain access to the data stored there after taking that link out of the list. The second step performs the actual deletion by changing the `LinkedList`'s `__first` attribute to point at the link after the `first` one. The `first` link is now disconnected even though its `__next` pointer points to the remainder of the list. The final step is to return the data from that `first` link.

When you need to delete from somewhere else in the list, the `delete()` method deletes an item with a matching `goal` key. Like the `insertAfter()` and `search()` methods, you first identify the `Link` where the delete should happen and then change the pointers. Although you could use the `find()` method to identify the `Link` to delete, the code shown in Listing 5-6 doesn't. Why do you think that is?

LISTING 5-6 The `deleteFirst()` and `delete()` Methods for `LinkedLists`

```
class LinkedList(object):           # A linked list of data elements

… (other definitions shown before) …

   def deleteFirst(self):           # Delete first Link
      if self.isEmpty():            # Empty list? Raise an exception
         raise Exception("Cannot delete first of empty list")

      first = self.getFirst()       # Store first Link
      self.setFirst(first.getNext()) # Remove first link from list
      return first.getData()        # Return first Link's data

   def delete(self, goal,           # Delete the first Link from the
              key=identity):        # list whose key matches the goal
      if self.isEmpty():            # Empty list? Raise an exception
         raise Exception("Cannot delete from empty linked list")

      previous = self               # Link or LinkedList before Link
      while previous.getNext() is not None: # to be deleted
         link = previous.getNext()  # Next link after previous
         if goal == key(link.getData()): # If next Link matches,
            previous.setNext(       # change the previous' next
               link.getNext())      # to be Link's next and return
            return link.getData()   # data since match was found
         previous = link            # Advance previous to next Link

      # Since loop ended without finding item, raise exception
      raise Exception("No item with matching key found in list")
```

The reason for not using the `find()` method is that `delete()` must identify the `Link` that *precedes* the one to be deleted, not the `Link` with the matching key itself. By finding the preceding `Link`, it can modify that link's `__next` pointer to shorten the list and eliminate the desired `Link`. That means the method uses a different kind of `while` loop where it updates a pointer called `previous` that points at the `Link` preceding the one with the goal key, if one exists. After checking for an empty list condition and raising an exception if it finds one, the `delete()` method sets `previous` to point at the input `LinkedList` (`self`), *not* the first `Link` in the chain. This is necessary if the `Link` to be deleted is the first `Link` in the list. Let's start by looking at that case, in particular.

The `previous` variable initially points to `self`, which is the `LinkedList` to be modified. On a call to the `delete()` method where the `goal` key matches the first `Link` in the linked list, the `while` loop is entered by first checking that the next `Link` after `previous` exists. It makes that check by calling `getNext()`. This explains why `getNext()`was defined as a

synonym for `getFirst()`; you want to treat the first pointer of the `LinkedList` just like all the other next pointers in the `Links` that follow.

The `delete()` method's loop sets the `link` variable to point at the `Link` that follows `previous`, which is the first `Link` of the list. When the loop body checks that first `Link`'s key, it matches the goal in the first case, so now the method can perform the deletion. The `previous` variable remains pointing at `self`, the `LinkedList` to be modified. By setting the next link of `previous` using `setNext()`, the `delete()` method modifies the input `LinkedList`'s `__first` link. This is the other synonym method defined in Listing 5-3 so that you can apply `setNext()` to modify either the `__first` attribute or the `__next` attribute as appropriate. It sets `__first` to be the `Link` following the one whose key matched the goal. That means the `Link` with the matching key has been removed from the list. Then the method can return the link's data because it found the `Link` to delete.

The second case to examine is what happens when the first `Link` in the list doesn't match the `goal` key to be deleted. Figure 5-7 illustrates the two cases, showing where `previous` is pointing right before the call to `previous.setNext()`. The `previous` and `link` variables start off the same in the second case, but this time, the first `Link`'s key doesn't match the `goal`. It skips the body of the inner `if` statement and sets `previous` to the value of `link`, which is the next `Link` in the list. Now `previous` is the first `Link` in the list, and the `while` loop checks whether there are any more `Links` after it. If there are more, the `link` variable is assigned to the next one, and its key is checked against the `goal`. When the key of `link` matches, the `previous` variable will point to the `Link` that immediately precedes `link`. The body of the `if` statement removes the goal `Link` in the same way as if it were the first `Link` in the list, thanks to the `setNext()` method working on both `LinkedLists` and `Links`.

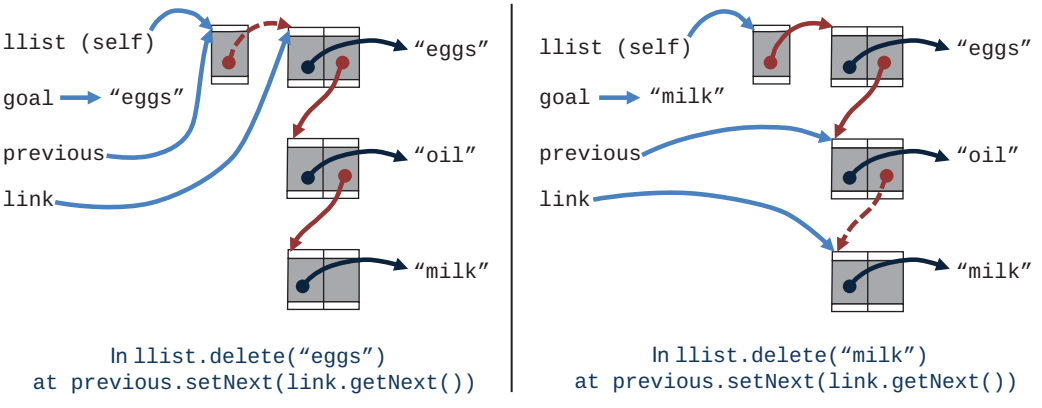


FIGURE 5-7 Two examples of deleting a `Link` from a `LinkedList`

In the first case we discussed, `previous` points to an object of type `LinkedList`, and in the second case it points to a `Link` object. We designed both classes to have `getNext()` and `setNext()` methods to make this kind of `while` loop in `delete()` easier to write. This is an example of polymorphism in object-oriented programming—where the same operation

can be performed on different object types with each type (class) having its own implementation. The idea is to treat both the `__first` and `__next` pointers in the same way across both classes. The reference that is about to be set is shown as a dashed arrow in Figure 5-7.

Double-Ended Lists

A double-ended list is like an ordinary linked list, but it has one additional feature: a reference to the `__last` link as well as to the first. Figure 5-8 shows such a list.

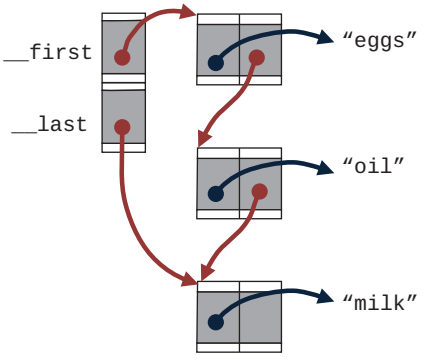


FIGURE 5-8 A double-ended list

The reference to the `__last` link permits inserting new links directly at the end of the list as well as at the beginning. Of course, you can insert a new link at the end of an ordinary single-ended list by iterating through the entire list until you reach the end. That approach, however, becomes quite inefficient when the list is long.

Quick access to the end of the list as well as the beginning makes the double-ended list suitable for certain situations that a single-ended list can't handle efficiently. One such situation is implementing a queue; we show how this technique works in the next section. There is a small cost for that benefit—maintaining the pointer to the other end—as shown in Listing 5-7.

LISTING 5-7 The `DoubleEndedList` Class

```
from LinkedList import *

class DoubleEndedList(LinkedList): # A linked list with access to both
   def __init__(self):             # ends of the list
      self.__first = None          # Reference to first Link, if any
      self.__last = None           # Reference to last link, if any

   def getFirst(self): return self.__first # Return the first link
```

```
   def setFirst(self, link):       # Change the first link to a new Link
      if link is None or isinstance(link, Link): #Must be Link or None
         self.__first = link       # Update first link
         if (link is None or       # When removing the first Link or
             self.getLast() is None): # the last Link is not set,
            self.__last = link     # then update the last link, too.
      else:
         raise Exception("First link must be Link or None")

   def getLast(self): return self.__last # Return the last link

   def last(self):                 # Return the last item in the list
      if self.isEmpty():           # as long as it is not empty
         raise Exception('No last element in empty list')
      return self.__last.getData()

   def insertLast(self, datum):    # Insert a new datum at end of list
      if self.isEmpty():           # For empty lists, end is the front,
         return self.insert(datum) # so insert there
      link = Link(datum, None)     # Else make a new end Link with datum
      self.__last.setNext(link)    # Add new Link after current last
      self.__last = link           # Change last to new end Link

   def insertAfter(                # Insert a new datum after the 1st
         self, goal, newDatum,     # Link with a matching key
         key=identity):
      link = self.find(goal, key)  # Find matching Link object
      if link is None:             # If not found,
         return False              # return failure
      newLink = Link(              # Else build a new Link node with
         newDatum, link.getNext()) # new datum and remainder of list
      link.setNext(newLink)        # and insert after matching link
      if link is self.__last:      # If the update was after the last,
         self.__last = newLink     # then update reference to last
      return True

   def delete(self, goal,          # Delete the first Link from the
              key=identity):       # list whose key matches the goal
      if self.isEmpty():           # Empty list? Raise an exception
         raise Exception("Cannot delete from empty linked list")
```

```
        previous = self              # Link or LinkedList before Link
        while previous.getNext() is not None: # to be deleted
           link = previous.getNext()  # Next link after previous
           if goal == key(link.getData()): # If next Link matches,
              if link is self.__last:   # and if it was the last Link,
                 self.__last = previous # then move last back 1
              previous.setNext(      # Change the previous' next
                 link.getNext())     # to be Link's next and return
              return link.getData()  # data since match was found
           previous = link           # Advance previous to next Link

        # Since loop ended without finding item exception
        raise Exception("No item with matching key found in list")
```

We choose to implement a double-ended list by making a subclass of the `LinkedList` that we've been studying. That's done in the `class` statement, which shows the `DoubleEndedList` inheriting the definition of `LinkedList`. That means we only need to redefine methods that change from those that were used in the parent, or "super," class. The constructor method, `__init__()`, differs only in that it sets the `__last` pointer to initially be empty, like the `__first` pointer.

The next definitions in Listing 5-7 are the accessor methods for the first link of the list. If you look carefully, you'll notice that the `getFirst()` method is identical to the definition in the parent class, `LinkedList`. That seems strange because subclasses only need to redefine methods that have changed from the definition in their super classes. The reason for redefining it is that Python has a special way of handling private class fields like `__first` and `__last`. Python treats those variables as being completely private to the class where they are defined. If subclasses use private fields with the same name, they are distinct fields from those of the parent class. So, the definition of the `getFirst()` method is really making an accessor for the `__first` attribute of the `DoubleEndedList` class in Listing 5-7, and that is different from the `LinkedList.__first attribute`. The new definition replaces that of the superclass so that all calls to `getFirst()` will get the `DoubleEndedList` `__first` attribute value.

The differences between the `DoubleEndedList` class and its parent begin to show up in the `setFirst()` method. Like the definition in the super class, it verifies that the new value is either an instance of the `Link` class or `None`. After setting the `__first` field, it goes on to see whether an update to the `__last` field is needed. The last `Link` needs to be updated when the first link is added to an empty list, or when the last `Link` is being removed from the list. The `if` statement checks for the removal of the last link by testing whether the new `link` is `None`. It checks for the addition of a potential new `Link` by testing whether the `__last` field is `None`. Note that it checks the status of the `__last` field by using the accessor method, `getLast()`, which is defined later. Avoiding references to the private attributes enables future subclassing of this class.

The `DoubleEndedList` class defines a `getLast()` method but not a `setLast()` method. That's intentional because it is not wise to allow callers to change the internal, private

field in a way that could point that field somewhere other than the last Link of the list. All changes should be managed by the code in the class definition. (To further protect the DoubleEndedList class from corruption, the setFirst() method would need to be rewritten to ensure that the __first and __last attributes always point to the beginning and end of the same linked list.)

The definition also includes a last() method, which, like the first() method, gets the indicated data rather than a reference to the Link object holding the data. The first() method is inherited from the LinkedList class, as are the synonym definitions for getNext() and setNext().

Continuing in Listing 5-7, you find the new insertLast() method, which exploits the presence of the __last field to make adding a Link at the end of the list very efficient. If the list is empty, then inserting at the end is the same as inserting at the front of the list. Otherwise, a new terminal Link object is created. The logic to insert it is the same as that used in insert(), except that the __next field of the last Link is updated rather than the __first field of the DoubleEndedList. Because a new Link is added, the __last field must also be updated too.

The last two methods of the DoubleEndedList class are slight rewrites of the same methods in LinkedList. The __last field must be updated appropriately after insertions and deletions. In the insertAfter() method, a new if statement is added at the end to check if the link that it modifies is the last Link of the list. If it is the last one, then the __last field must be advanced to point at the newly added Link. In the delete() method, another new if statement is added just before modifying the previous Link's __next (or LinkedList's __first) pointer. If the link to delete is the same as the last link, then the __last field must be updated to point at the previous link.

There is one special case to consider during deletion: when previous points to the DoubleEndedList and not a Link object, the final link is being deleted. The delete() method initially sets __last to previous which is the DoubleEndedList object, but then, the test in setNext()/setFirst() will find that the new value for __first is None and set both __first and __last of the DoubleEndedList to be None.

Interestingly, the DoubleEndedList does not need to redefine the deleteFirst() method inherited from LinkedList. When deleting the first Link, the deleteFirst() method calls setFirst(), which has already been customized in the DoubleEndedList to update its private __last pointer.

You can exercise the features of the DoubleEndedList class using the program shown in Listing 5-8. The client program is a little fancier by using records for the items instead of just strings. It builds an empty double-ended list, dlist, and then inserts pairs of numbers with names into the list. It uses the second element of each pair, the name, as the key for the data. The numbers in the record help indicate the order that they were inserted. It puts several people into the list, inserting the first one at the beginning, setting a variable called after to the person in that first datum, and then inserting all the other pairs after the first.

LISTING 5-8 The DoubleEndedListClient.py Program

```
from DoubleEndedList import *

def second(x): return x[1]

dlist = DoubleEndedList()

print('Initial list has', len(dlist), 'element(s) and empty =',
      dlist.isEmpty())
after = None
people = ['Raj', 'Amir', 'Adi', 'Don', 'Ken', 'Ivan']
for i, person in enumerate(people):
   if after:
      dlist.insertAfter(after, (i * i, person), key=second)
   else:
      dlist.insert((i * i, person))
      after = person

print('After inserting', len(dlist) - 1,
      'persons into the linked list after', after, 'it contains:')
dlist.traverse()
print('First:', dlist.first(), 'and Last:', dlist.last())

next = (404, 'Tim')
dlist.insertLast(next)
print('After inserting', next, 'at the end, the double-ended list',
      'contains:\n', dlist)

dlist.insert(next)
print('After inserting', next, 'at the front, the double-ended list',
      'contains:\n', dlist)
print('Deleting the first item returns', dlist.deleteFirst(),
      'and leaves the double-ended list containing:\n', dlist,
      'with first:', dlist.first(), 'and Last:', dlist.last())
print('Deleting the last item returns',
      dlist.delete(second(dlist.last()), key=second),
      'and leaves the double-ended list containing:\n', dlist,
      'with first:', dlist.first(), 'and Last:', dlist.last())

print('Removing some items from the linked list by key:')
for person in people[0:5:2]:
   dlist.delete(person, key=second)
   print('After deleting', person, 'the list is', dlist)
```

```
   if not dlist.isEmpty():
      print('The last item is', dlist.last())

print('Removing remaining items from the front of the linked list:')
while not dlist.isEmpty():
   print('After deleting', dlist.deleteFirst(), 'the list is', dlist)
   if not dlist.isEmpty():
      print('The last item is', dlist.last())
```

After the initial data is inserted, it displays the contents of the list using the `traverse()` method. The specific first and last data are also printed. The program adds another pair, `next`, at the end and beginning of the list, printing the contents of the list using the string conversion capability after each insertion. The next test deletes those first and last items, exercising the `deleteFirst()` and `delete()` methods.

Finally, it removes some items based on their keys, using a few of the `people` names. Then it removes the rest by deleting the first item in the list. After each deletion, it shows the content of the list and the last data item. The output of the program is

```
$ python3 DoubleEndedListClient.py
Initial list has 0 element(s) and empty = True
After inserting 5 persons into the linked list after Raj it contains:
(0, 'Raj')
(25, 'Ivan')
(16, 'Ken')
(9, 'Don')
(4, 'Adi')
(1, 'Amir')
First: (0, 'Raj') and Last: (1, 'Amir')
After inserting (404, 'Tim') at the end, the double-ended list contains:
 [(0, 'Raj') > (25, 'Ivan') > (16, 'Ken') > (9, 'Don') > (4, 'Adi') > (1, 'Amir') >
(404, 'Tim')]
After inserting (404, 'Tim') at the front, the double-ended list contains:
 [(404, 'Tim') > (0, 'Raj') > (25, 'Ivan') > (16, 'Ken') > (9, 'Don') > (4, 'Adi') >
(1, 'Amir') > (404, 'Tim')]
Deleting the first item returns (404, 'Tim') and leaves the double-ended list
containing:
 [(0, 'Raj') > (25, 'Ivan') > (16, 'Ken') > (9, 'Don') > (4, 'Adi') > (1, 'Amir') >
(404, 'Tim')] with first: (0, 'Raj') and Last: (404, 'Tim')
Deleting the last item returns (404, 'Tim') and leaves the double-ended list
containing:
 [(0, 'Raj') > (25, 'Ivan') > (16, 'Ken') > (9, 'Don') > (4, 'Adi') > (1, 'Amir')]
with first: (0, 'Raj') and Last: (1, 'Amir')
```

```
Removing some items from the linked list by key:
After deleting Raj the list is [(25, 'Ivan') > (16, 'Ken') > (9, 'Don') > (4, 'Adi') >
(1, 'Amir')]
The last item is (1, 'Amir')
After deleting Adi the list is [(25, 'Ivan') > (16, 'Ken') > (9, 'Don') > (1,
'Amir')]
The last item is (1, 'Amir')
After deleting Ken the list is [(25, 'Ivan') > (9, 'Don') > (1, 'Amir')]
The last item is (1, 'Amir')
Removing remaining items from the front of the linked list:
After deleting (25, 'Ivan') the list is [(9, 'Don') > (1, 'Amir')]
The last item is (1, 'Amir')
After deleting (9, 'Don') the list is [(1, 'Amir')]
The last item is (1, 'Amir')
After deleting (1, 'Amir') the list is []
```

The initial pair, `(0, 'Raj')`, stays at the beginning of the list while the other pairs are inserted after it. Their numeric values decrease, which shows that their order in the completed list is the *reverse* of the order they were inserted. This is not an ordered list. The insertion position is defined by the goal key passed to `insertAfter()`. The insertion of `(404, 'Tim')` is done with both `insertLast()` and `insert()`, so it doesn't even look at the keys of the list items. As the items are deleted, the program prints the list and the last element, just to verify that the `__last` pointer is being updated correctly.

The `DoubleEndedList` makes it easy to insert at both the front and end of the list as well as after a `Link` having a particular key. The `delete()` method always looks for a particular key, and it's easy to delete the first and last keys by using the `first()` and `last()` methods. Unfortunately, deleting the last link is still time-consuming because the program still must walk through the `Link`s to find the last key. To make it fast, you could add a reference to the next-to-last link, whose `__next` field would need to be changed to `None` when the last link is deleted. You already saw what it takes to modify the `LinkedList` to become the `DoubleEndedList`. Adding more and more fields like `second_to_last`, `third_to_last`, and so on, would be a lot of work, and it's unclear how much more efficiency that would achieve because you could have lists of arbitrary length. To conveniently delete the last link, a better approach is a *doubly linked* list, which we look at soon.

Linked List Efficiency

Insertion and deletion at the beginning of a linked list are very fast. They involve changing only one or two references, which takes O(1) time.

Finding, deleting, or inserting next to a specific item requires searching through, on average, half the items in the list. This operation requires O(N) comparisons. Arrays have the same complexity for these operations, O(N), but the linked list is faster because nothing needs to be copied when an item is inserted or deleted. The increased efficiency can be significant, especially if a copy takes much longer than a comparison.

Of course, another important advantage of linked lists over arrays is that a linked list uses exactly as much memory as it needs and can expand to fill all available memory. The size of an array is fixed when it's created; this could lead to inefficiency when the initial array size is too large, or it could lead to running out of room because the array is too small. Expandable arrays may solve this problem to some extent, but they usually expand in fixed-sized increments (such as doubling the size of the array whenever it's about to overflow). Typically, the data must be copied from the smaller to larger array when they expand. This adaptive solution is still not as efficient a use of memory as a linked list.

Abstract Data Types and Objects

In this section we shift gears and discuss a topic that's more general than linked lists: abstract data types (ADTs). What is an ADT? Is it the same as an object? Roughly speaking, an ADT is a way of looking at a data structure—focusing on what it does and ignoring how it does its job. Objects are a way of making concrete implementations of ADTs.

Stacks and queues are examples of ADTs. You've already seen that both stacks and queues can be implemented using arrays. Before we return to a discussion of ADTs, let's see how stacks and queues can be implemented using linked lists. This discussion demonstrates the "abstract" nature of stacks and queues—how they can be considered separately from their implementation.

A Stack Implemented by a Linked List

The implementation of the stack in Chapter 4, "Stacks and Queues," used an ordinary array to hold the stack's data. The stack's `push()` and `pop()` operations were carried out by array operations such as

```
a[top] = item
```

and

```
top = a[j]
```

which put data into and returned it from an array.

You can also use a linked list to hold a stack's data. In this case, the `push()` and `pop()` operations could be carried out by operations like

```
theList.insert(data)
```

and

```
data = theList.deleteFirst()
```

The user of the stack class calls `push()` and `pop()` to insert and delete items without knowing, or needing to know, whether the stack is implemented as an array or as a linked list. Listing 5-9 shows two ways of how a stack class can be implemented using the `LinkedList` class instead of an array.

LISTING 5-9 The LinkStack.py Module

```
from LinkedList import *

class LinkStack(object):
   def __init__(self):                  # Constructor for a
      self.__sList = LinkedList()       # stack stored as a linked list

   def push(self, item):                # Insert item at top of stack
      self.__sList.insert(item)         # Store item

   def pop(self):                       # Remove top item from stack
      return self.__sList.deleteFirst() # Return first and delete it

   def peek(self):                      # Return top item
      if not self.__sList.isEmpty():    # If stack is not empty
         return self.__sList.first()    # Return the top item

   def isEmpty(self):                   # Check if stack is empty
      return self.__sList.isEmpty()

   def __len__(self):                   # Return # of items on stack
      return len(self.__sList)

   def __str__(self):                   # Convert stack to string
      return str(self.__sList)

class Stack(LinkedList):                # Define stack by renaming
   push = LinkedList.insert             # Push is done by insert
   pop = LinkedList.deleteFirst         # Pop is done by deleteFirst
   peek = LinkedList.first              # Peek is done by first
```

The first thing to note about the `LinkStack` and `Stack` class definitions is how short they are. The implementation of a stack using a linked list is straightforward because they share so many common features. In `LinkStack`, the constructor allocates an empty linked list, `__sList`, and the other methods are basically translations of the common list methods. In the case of a stack, the key function for the `LinkedList` find, search, and delete functions is never used because stacks don't need keyed access to elements. Another difference is that the constructor doesn't require a maximum size parameter like the `SimpleStack` class does (in Chapter 4). The reason is that the implementation is not using an array that needs an initial size.

The `Stack` class definition is only four lines in Python. Instead of creating a `LinkedList` object as an internal attribute, it is defined as a subclass of `LinkedList`, inheriting all its attributes and methods. All that is needed is to provide the translations for those method names that differ between a stack and a list.

The shorthand definition of the `Stack` class might seem incomplete to new Python programmers. Normally, there would be some `def` statements for its methods. Here you simply define some class attributes: `push`, `pop`, and `peek`. This approach takes advantage of the way that Python represents class methods as class attributes that are bound to functions instead of some other data type. The statement `push = LinkedList.insert` takes the `insert` attribute (method) of the `LinkedList` class and assigns it to the `push` attribute of the `Stack` class. Because that attribute is executable (Python treats it as *callable*), the attribute becomes a method. The three assignment statements take care of all the changed method names. The remaining methods like `isEmpty()` and the constructor are inherited from `LinkedList`.

To see how both stack implementations work, it's a good idea to test them with something like the `LinkStackClient.py` program shown in Listing 5-10. The outer loop creates two empty stacks, one of each kind, and sets the `stack` variable to each one in turn. The first statement in the loop body prints the data type, shows the stack's string representation, and tests its `isEmpty()` method. The inner `for` loop pushes some squares on the stack. That's followed by printing the full stack contents and tests of the `len()` and `peek()` methods. The final `while` loop then pops the items off, one by one, showing the stack contents and length after each operation.

LISTING 5-10 The `LinkStackClient.py` Program

```
from LinkStack import *

for stack in (LinkStack(), Stack()):
   print('\nInitial stack of type', type(stack),
         'holds:', stack, 'is empty =', stack.isEmpty())

   for i in range(5):
      stack.push(i ** 2)

   print('After pushing', len(stack),
         'squares on to the stack, it contains', stack)
   print('The top of the stack is', stack.peek())

   while not stack.isEmpty():
      print('Popping', stack.pop(), 'off of the stack leaves',
            len(stack), 'item(s):', stack)
```

The results of running `LinkStackClient.py` are

```
$ python3 LinkStackClient.py

Initial stack of type <class 'LinkStack.LinkStack'> holds: [] is empty = True
After pushing 5 squares on to the stack, it contains [16 > 9 > 4 > 1 > 0]
```

```
The top of the stack is 16
Popping 16 off of the stack leaves 4 item(s): [9 > 4 > 1 > 0]
Popping 9 off of the stack leaves 3 item(s): [4 > 1 > 0]
Popping 4 off of the stack leaves 2 item(s): [1 > 0]
Popping 1 off of the stack leaves 1 item(s): [0]
Popping 0 off of the stack leaves 0 item(s): []

Initial stack of type <class 'LinkStack.Stack'> holds: [] is empty = True
After pushing 5 squares on to the stack, it contains [16 > 9 > 4 > 1 > 0]
The top of the stack is 16
Popping 16 off of the stack leaves 4 item(s): [9 > 4 > 1 > 0]
Popping 9 off of the stack leaves 3 item(s): [4 > 1 > 0]
Popping 4 off of the stack leaves 2 item(s): [1 > 0]
Popping 1 off of the stack leaves 1 item(s): [0]
Popping 0 off of the stack leaves 0 item(s): []
```

Note that the `LinkStackClient.py` program doesn't need to know how the `LinkStack` or `Stack` classes are implemented. If you also imported the `SimpleStack` module of Chapter 4 and added a call to the `SimpleStack.Stack(10)` class constructor (where a maximum stack size is required) to the values for `stack` in the outer loop, the rest of the `LinkStackClient.py` program would work identically on that structure. To the programmer writing `LinkStackClient.py`, there is no difference between using the list-based `LinkStack` class and using the array-based `Stack` class from the `SimpleStack` module of Chapter 4, except for providing a maximum stack size and seeing a different string format for the stack contents.

A Queue Implemented by a Linked List

Here's a similar example of an ADT implemented with a linked list. Listing 5-11 shows a queue implemented as a double-ended linked list.

LISTING 5-11 The `LinkQueue.py` Module

```
from DoubleEndedList import *

class Queue(DoubleEndedList):            # Define queue by renaming
   enqueue = DoubleEndedList.insertLast  # Enqueue/insert at end
   dequeue = DoubleEndedList.deleteFirst # Dequeue/remove at first
   peek = DoubleEndedList.first          # Front of queue is first
```

Similar to the stack, the queue ADT doesn't need to access elements of the queue based on a key, only by their position. It defines the method names needed for a queue using the counterpart methods of the `DoubleEndedList`. The very short 3-line definition looks very simple, but it somewhat hides a fundamental name conflict.

The `Queue` class you saw in Chapter 4 used the `insert()` and `remove()` methods for insertion and removal. If those same method names are used for the `Queue` class based on the `DoubleEndedList` shown in Listing 5-11, a problem would arise although it's hard to see. If you review the definition of `insertLast()` in Listing 5-7, it calls `insert()` when the list is empty to invoke the `LinkedList.insert()` method. If the new `Queue` class definition redefines the `insert()` method, however, then those calls would be redirected to the `DoubleEndedList.insertLast()` method. That method would call the `insert()` method again for an empty list and thus cause an infinite loop. A simple way to avoid that problem is to use the `enqueue()` and `dequeue()` method names for the queue's primary operations.

You can use a similar client program to test the new implementation of the queue with its `enqueue()` and `dequeue()` methods, as shown in Listing 5-12.

LISTING 5-12 The `LinkQueueClient.py` Program

```
from LinkQueue import *

queue = Queue()

print('Initial queue:', queue, 'is empty =', queue.isEmpty())

for i in range(5):
   queue.enqueue(i ** 2)

print('After inserting', len(queue),
      'squares on to the queue, it contains', queue)
print('The front of the queue is', queue.peek())

while not queue.isEmpty():
   print('Removing', queue.dequeue(), 'off of the queue leaves',
         len(queue), 'item(s):', queue)
```

The program creates a queue using the `Queue()` constructor; shows its initial state, which is empty; inserts five squares; displays the queue contents; peeks at the front, and then removes items from the queue, displaying the contents after each removal. Here's the output:

```
$ python3 LinkQueueClient.py
Initial queue: [] is empty = True
After inserting 5 squares on to the queue, it contains [0 > 1 > 4 > 9 > 16]
The front of the queue is 0
Removing 0 off of the queue leaves 4 item(s): [1 > 4 > 9 > 16]
Removing 1 off of the queue leaves 3 item(s): [4 > 9 > 16]
Removing 4 off of the queue leaves 2 item(s): [9 > 16]
Removing 9 off of the queue leaves 1 item(s): [16]
Removing 16 off of the queue leaves 0 item(s): []
```

Here, the methods `enqueue()` and `dequeue()` in the `Queue` class are implemented by the `insertLast()` and `deleteFirst()` methods of the `DoubleEndedList` class. Compare this to the array used to implement the queue in the `Queue.py` module of Chapter 4 and see how much shorter the definition is.

The `LinkStack.py` and `LinkQueue.py` modules emphasize that stacks and queues are conceptual entities, separate from their implementations. A stack can be implemented equally well by an array or by a linked list. What's important about a stack is its `push()`, `pop()`, and `peek()` operations and how they're used, not the underlying mechanism used to implement them.

When would you use a linked list as opposed to an array as the implementation of a stack or queue? One consideration is how accurately you can predict the amount of data the stack or queue will need to hold. If this isn't clear, the linked list gives you more flexibility than an array. Both are fast, so speed is probably not a major consideration.

Data Types and Abstraction

Where does the term *abstract data type* come from? Let's look at the *data type* part of it first and then return to *abstract*.

Data Types

The term *data type* can be used in many ways. It is often used to describe built-in types such as `int`, `float`, and `str` in Python or equivalent types in other programming languages. This might be what you first think of when you hear the term.

When you talk about a primitive type, you're actually referring to two things: a data item represented as a collection of bits with certain characteristics, and the permissible operations on that data. For example, a Boolean type variable in Python can have two values—`True` or `False`–and the operators `not`, `and`, and `or` can be applied to them. An `int` variable has whole-number values, and operators like +, –, *, and / can be applied to them. The data type's permissible operations are an inseparable part of its identity; understanding the type means understanding what operations can be performed on it.

Object-oriented programming allows programmers to create new data types using classes. Some of these data types represent numerical quantities that are used in ways similar to primitive types. You can, for example, define a class for time duration (with fields for hours, minutes, seconds), and a class for fractions (with numerator and denominator fields). All these classes can be added, subtracted, multiplied, and divided like `int` and `float`. Typically, the new object methods are defined with functional notation like `add()`, `sub()`, `mul()`, and `div()`. That leads to expressions like `a.sub(b).mul(c)` that don't look like the math expressions we teach and use, but they still produce valid results. Python and some other languages have ways that programmers can define methods so that built-in operators like + and – can work with them. For example, if you define a method named `__add__(a, b)` for a class in Python, that class can be used with the built-in + operator. These mechanisms give programmers control over defining all the permissible operations of a data type.

The phrase *data type* seems to fit naturally with quantity-oriented classes. It can, however, apply to classes that don't have this quantitative aspect. In fact, *any* class represents a data type, in the sense that a class is made up of data (fields) and permissible operations on that data (methods). They are not, however, one and the same. A class goes beyond defining a data type or types and the permissible operations that can be performed on them because it also supplies the implementation details.

When a data storage structure like a stack or queue is represented by a class, it too can be referred to as a "data type." That might sound odd at first, to someone just learning about data storage structures. A stack is different in many ways from an `int`, but they are both defined as a certain arrangement of data and a set of operations on that data. When that new structure is not part of the language or its standard libraries, it's usually not called a "primitive" or a "built-in" data type. Instead, it might be called a "user-defined" or "program-defined" data type, but it is still a data type.

Abstraction

The word *abstract* means "considered apart from detailed specifications or implementation." An abstraction is the essence or important characteristics of something. For example, refrigeration is an abstraction. Its important characteristic is keeping something or someplace colder than it would normally be. It is distinct from a refrigerator, which is a device that makes things colder. The legislature of a state or country is another example. The legislature can be considered apart from the individual legislators who happen to perform that duty. The powers and responsibilities of the office remain roughly the same over time while individual officeholders come and go much more frequently.

In object-oriented programming, an **abstract data type** is a class considered without regard to its implementation. It's a description of the data in the class (fields or attributes), the relationships of the fields, a list of operations (methods) that can be carried out on that data, and instructions on how to use these operations. Specifically excluded are the details of how the methods carry out their tasks. As a class user, you're told what the fields mean, what values they can take, what methods to call, how to call them, and the results you can expect, but not how they work.

The meaning of *abstract data type* is further extended when it's applied to data structures such as stacks and queues. As with any class, it means the data and the operations that can be performed on it, but in this context even the fundamentals of how the data is stored become invisible to the user. Users not only don't know how the methods work; they also don't know what structure(s) store the data. Although the exact mechanism isn't known, users usually do know the complexity of the methods, that is, whether they are O(1), O(log N), O(N), and so on. When the insertion into a stack or a queue is O(1), you expect all implementations regardless of their implementation to maintain that efficiency.

For the stack, the user (programmer) knows that `push()`, `pop()`, and `peek()` (and perhaps a few other methods like `isEmpty()`) exist and what they are supposed to do. The user doesn't (at least not usually) need to know how `push()` and `pop()` work, or whether data is stored in an array, a linked list, or some other data structure like a tree. In this book, we usually refer to the abstract data type, like an array, without capitalization or a special

font. For a class like `LinkStack`, its methods, or a particular instance or object of that class, we use capitalization and the font to distinguish it from the abstract data type.

The Interface

An ADT specification is often called an **interface**. It's what the class user sees—usually its public methods and fields. In a stack class, `push()` and `pop()` and similar methods form the interface. Public attributes like `nItems` are also part of the interface.

ADT Lists

Now that you know what an abstract data type is, here's another one: the *list*. A list (sometimes called a linear list) is a group of items arranged in a linear order. That is, they're lined up in a certain way, like beads on a string or links in a chain. Lists support certain fundamental operations. You can insert an item, delete an item, and usually read an item from a specified location (the first, the last, and perhaps an intermediate item specified by a key or an index). The exact choice of which of those operations is possible is part of the definition of the ADT.

Don't confuse the ADT list with the linked list classes discussed in this chapter, `LinkedList` and `DoubleEndedList`, or Python's `list` data type. An ADT list is defined by its interface—the specific methods and attributes used to interact with it. This interface can be implemented by various structures, including arrays and linked lists. The list is an abstraction of such data structures. The classes are objects, and objects are a way of implementing abstract data types. When detailed information is added about storage and algorithms, it becomes a **concrete data type**.

ADTs as a Design Tool

The ADT concept is a useful aid in the software design process. If you need to store data, start by considering the operations that need to be performed on that data. Do you need access to the last item inserted? The first one? An item with a specified key? An item in a certain position? How frequently do you expect each kind of access to occur? Answering such questions leads to the choice of an appropriate ADT or the definition of a new one. Only after the ADT is completely defined should you worry about the details of how to represent the data and how to code the methods that access the data.

By decoupling the specification of the ADT from the implementation details, you can simplify the design process. You also make it easier to change the implementation at some future time. If a programmer writes a design or even develops some code using only the ADT interface, a second programmer should be able to change the implementation of the ADT without "breaking" the first programmer's code.

Of course, after designing the ADT, the underlying data structure and algorithm implementation must be carefully chosen to make the specified operations as efficient as possible. If you need random access to the Nth element, for example, the linked-list representation isn't so good because random access isn't an efficient operation for a linked list. You'd be better off with an array. If random access is needed only very rarely, and the extra time needed to find the item is small compared to the time it could take to copy the

array contents several times when growing (and maybe shrinking) an expandable array, then maybe the linked list representation is better.

NOTE

Remember that the ADT concept is only a conceptual tool. Data storage structures are not divided cleanly into some that are ADTs and some that are used to implement ADTs. A linked list, for example, doesn't need to be wrapped in a list interface to be useful; it can act as an ADT on its own, or it can be used to implement another data type such as a queue. A linked list can be implemented using an array, and an array-type structure can be implemented using a linked list. In fact, Python's `list` data type is implemented using arrays and acts like both an array and a list. This implementation is "invisible" to most programmers, although it's important to know that adding an element to the end of Python `list` is usually an O(1) operation, not O(N).

Ordered Lists

In the linked lists you've seen thus far, there was no requirement that data be stored in order. For many applications, however, it's useful to maintain the data in a particular order within the list. A list with this characteristic is called an **ordered list**.

In an ordered list, the items are always arranged in order by a key value. Deletion is often limited to the smallest (or the largest) item in the list, which is at the start of the list, although sometimes `find()` and `delete()` methods, which search through the list for specified links, are used as well.

In general, you can use an ordered list in most situations in which you use an ordered array. The advantages of an ordered list over an ordered array are speed of insertion (because elements don't need to be moved) and the fact that a list can expand to meet the minimum memory required, whereas an array is limited to a fixed size that must be declared before using it. An ordered list may, however, be more difficult to implement than an ordered array.

Later we look at one application for ordered lists: sorting data. An ordered list can also be used to implement a priority queue, although a heap (see Chapter 13, "Heaps") is a more common implementation. Note that it is very rare to start with an unordered linked list and sort its contents. Doing so efficiently takes extra work. We use the term *ordered list*, as opposed to *sorted list*, to reinforce that the data is always kept in order, not mixed up and later rearranged.

The LinkedList Visualization tool introduced at the beginning of this chapter demonstrated unordered lists. To see how ordered lists work, use the OrderedList Visualization tool, launched with a command like `python3 OrderedList.py`. The operations on the two lists have the same names but have different behaviors, of course, to maintain the ordering. Figure 5-9 shows the tool with a list of ingredients already entered, ordered alphabetically.

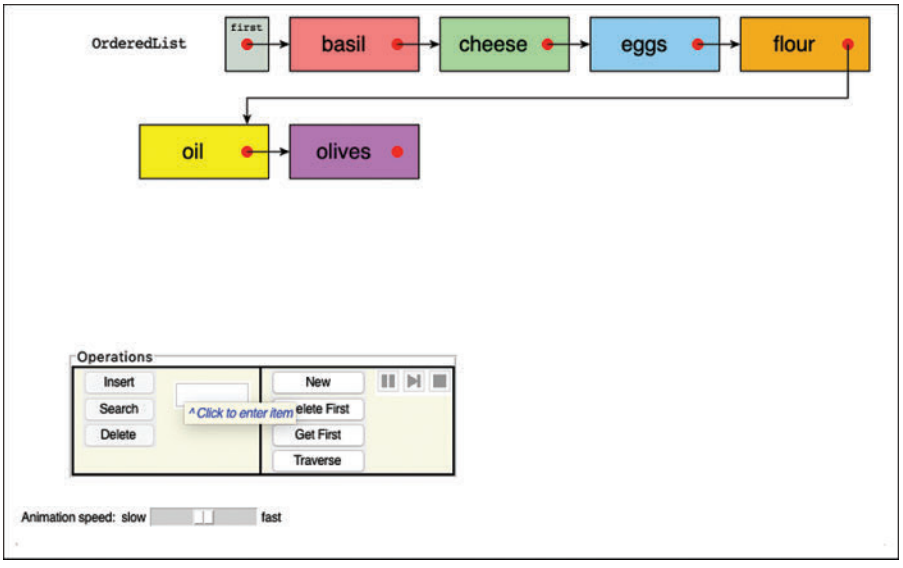


FIGURE 5-9 The OrderedList Visualization tool

Type some words and insert them using the text entry box and the Insert button. The first item always goes in the first position, but subsequent items get placed after any items already in the list by their alphabetical order. Watch the insertion animation after it finds the right place in the list. It's nearly the same as for the unordered list.

With several items in your list, try a search for an item. Search for both an existing key and one that's not in the list. The search stops as soon as it finds an item with a key larger than that of the goal key. Next, try deleting a key that's early in the list. The deletion process is the same as the unordered one when the key is in the list, but the search process for missing keys ends after it finds higher valued keys.

Python Code for Ordered Lists

An ordered list is a special kind of linked list, so the `OrderedList.py` module shown in Listing 5-13 defines the `OrderedList` class as a subclass of `LinkedList`. It has one extra attribute, `__key`, which is a function that extracts the sorting key from items in the list. The sort key is stored when the list is created and cannot be changed because that would allow a list of existing items to become unordered with respect to the key.

Because `OrderedList` is a subclass of `LinkedList`, it inherits all the methods defined in Listing 5-3 and Listing 5-4. The `getFirst()` and `setFirst()` methods must be redefined here for the same reason we noted in the `DoubleEndedList` subclass: Python's name mangling of attributes that start with double underscores prevents sharing the parent class's attributes. The other methods like `getNext()` and `isEmpty()` work without rewriting them. Note that making `OrderedList` a subclass of `LinkedList` leaves `setFirst()` as a public method, which is not advisable because callers could use it to place items in the list out of order.

LISTING 5-13 The Beginning of the `OrderedList.py` Module

```
from LinkedList import *

class OrderedList(LinkedList):      # An ordered linked list where items
   def __init__(self,               # are in increasing order by key, which
                key=identity):      # is retrieved by a func. on each item
      self.__first = None           # Reference to first Link, if any
      self.__key = key              # Function to retrieve key

   def getFirst(self): return self.__first # Return the first link

   def setFirst(self, link):        # Change the first link to a new Link
      if link is None or isinstance(link, Link): #Must be Link or None
         self.__first = link
      else:
         raise Exception("First link must be Link or None")

   def find(self, goal):            # Find the 1st Link whose key matches
                                    # or is after the goal
      link = self.getFirst()        # Start at first link, and search
      while (link is not None and         # while not end of the list
             self.__key(link.getData()) < goal): # and before goal
         link = link.getNext() # Advance to next in list
      return link # Return Link at or just after goal or None for end

   def search(self, goal):          # Find 1st datum whose key matches goal
      link = self.find(goal)        # Look for Link object that matches
      if (link is not None and      # If Link found, and its key
          self.__key(link.getData()) == goal): # matches the goal
         return link.getData()      # return its datum
```

The `find()` method resembles what was used for the `LinkedList`, but there are important changes. Because the list items are ordered, the methods that step through the list looking for a particular key take advantage of the ascending ordering. The method has a test in the `while` loop condition that checks that the key of the current link is less than that of the goal. That means that the loop can end when it

- Finds the link with the `goal` key
- Finds the next link immediately after where the link with the goal key would go or
- Hits the end of the list

Because the loop can end with `link` pointing at either the desired `Link` or the one with the next higher key value, there's a question on what value to return when the `goal` is not found. This implementation returns either kind of `Link` because the capability of finding the one immediately following a particular key can be used for some operations (for example, splitting the list). This is like the behavior of the `find()` method for ordered record arrays in Chapter 4, which returned the index where a new item should be inserted. In other words, the `find()` method returns a pointer to where a `Link` having the `goal` key *should be inserted in the list*. If it goes before the first item in the list, the pointer to the first `Link` is returned. If it goes somewhere before or at one of the other items, the pointer to that item is returned. Only if it goes after the last item is `None` returned.

The `search()` method calls `find()` and adds a test to its `if` statement checking whether the `Link` returned from `find()` is valid, and it returns the item stored in that `Link` only if its key matches.

The `insert()` method of the `OrderedList.py` module shown in Listing 5-14 does something similar. Ideally, it would just use `find()` to locate the insertion point and modify the list there. In this case, however, it needs to modify the `__next` pointer of the `Link` preceding the one returned by `find()`. So, like the `delete()` method of `LinkedList`, it uses a loop to search for the previous `Link` to the insertion point. More precisely, it looks for either the `OrderedList` or `Link` that precedes the insertion point. The `while` loop condition changes to also verify that the next `Link`'s key is less than the `goal` key; otherwise, the loop can end.

LISTING 5-14 The `insert()` and `delete()` Methods of `OrderedList`

```
class OrderedList(LinkedList):
…

   def insert(self, newDatum):      # Insert a new datum based on key order
      goal = self.__key(newDatum)   # Get target key
      previous = self               # Link or OrderedList before goal Link
      while (previous.getNext() is not None and  # Has next link and
             self.__key(previous.getNext().getData())
             < goal):               # next link's key is before the goal
         previous = previous.getNext() # Advance to next link
      newLink = Link(               # Build a new Link node with new
         newDatum, previous.getNext()) # datum and remainder of list
      previous.setNext(newLink)     # Update previous' first/next pointer

   def delete(self, goal):          # Delete first Link with matching key
      if self.isEmpty():            # Empty list? Raise an exception
         raise Exception("Cannot delete from empty linked list")

      previous = self               # Link or OrderedList before Link
```

```
        while (previous.getNext() is not None and  # Has next link and
               self.__key(previous.getNext().getData())
               < goal):               # next link's key is before the goal
           previous = previous.getNext()  # Advance to next link
        if (previous.getNext() is None or # If goal key not in next
            goal !=                       # Link after previous
            self.__key(previous.getNext().getData())):
           raise Exception("No datum with matching key found in list")

        toDelete = previous.getNext() # Store Link to delete
        previous.setNext(toDelete.getNext()) # Remove it from list

        return toDelete.getData()     # Return data in deleted Link
```

After the loop ends, `insert()` constructs the new `Link` with the provided `newDatum` and the rest of the list after the `previous` pointer. The `__next` pointer of `previous` is updated similarly to the other linked list classes. By inheriting the definition of `setNext()` as a synonym for the `setFirst()` method, the `OrderedList.insert()` method updates either the `__first` pointer or the `__next` pointer depending on the data type of `previous`.

The `delete()` method uses a similar structure as the `insert()` method, after first checking for an empty list and raising an exception if one is found. It uses the same style of `while` loop to get `previous` to point at the `Link` immediately before the link where the `goal` key is or should be. To avoid having the loop end with `previous` pointing to a position after where the `goal` key would go, the loop condition looks ahead at the next link's key, if there is one. When the loop is done, it checks if the `Link` just after `previous` has the `goal` key and raises an exception if it doesn't. Having verified that the `goal` key was found, it can note the link `toDelete` and remove it by modifying the appropriate `__next` or `__first` field by calling `setNext()` on the previous link. It replaces the old value by calling `getNext()` on the `Link` to be deleted. The data stored in the `toDelete` link is returned.

The movement of the `previous` pointer and the addition/removal of the next link is animated in the Insert/Delete operations of the Ordered List Visualization tool. Look at the individual steps to see the operations in detail.

Listing 5-15 shows a client program to test the `OrderedList` class. It starts with an empty ordered list using the default sorting key function, `identity()`. It inserts an alternating sequence of positive and negative integers. It displays the ordered list and then searches it for certain values to see what the `find()` and `search()` methods return. Finally, it removes the numbers in the order they were inserted, which means it first deletes items from the middle of the list and finishes by deleting the ones on the ends.

LISTING 5-15 The `OrderedListClient.py` Program

```
from OrderedList import *

olist = OrderedList()
print('Initial list has', len(olist), 'element(s) and empty =',
      olist.isEmpty())

for i in range(5):
   olist.insert((-1 - i) ** i)
print('After inserting', len(olist),
      'numbers into the ordered list, it contains:\n', olist,
      'and empty =', olist.isEmpty())

for value in [9, 999]:
   for sign in [-1, 1]:
      val = sign * value
      print('Trying to find', val, 'in ordered list returns',
            olist.find(val), ', search returns', olist.search(val))

print('Deleting items from the ordered list:')
for i in range(5):
   number = (-1 - i) ** i
   print('After deleting', olist.delete(number),
         'the list is', olist)
```

The result of running this program is

```
$ python3 OrderedListClient.py
Initial list has 0 element(s) and empty = True
After inserting 5 numbers into the ordered list, it contains:
 [-64 > -2 > 1 > 9 > 625] and empty = False
Trying to find -9 in ordered list returns -2 , search returns None
Trying to find 9 in ordered list returns 9 , search returns 9
Trying to find -999 in ordered list returns -64 , search returns None
Trying to find 999 in ordered list returns None , search returns None
Deleting items from the ordered list:
After deleting 1 the list is [-64 > -2 > 9 > 625]
After deleting -2 the list is [-64 > 9 > 625]
After deleting 9 the list is [-64 > 625]
After deleting -64 the list is [625]
After deleting 625 the list is []
```

Notice the result of the call to `find(–9)`. It returns a pointer to the `Link` holding –2. That might seem odd, but –9 falls between –64 and –2. The `find()` method returns a pointer

to a link immediately after where the given key should be inserted. The call to `find(9)` does find the `Link` holding 9 in it, and that is the only call to the `search()` method that returns something other than `None`. The calls to find –999 and 999 show the results when the goal key would land before or after all the list items, respectively.

Efficiency of Ordered Linked Lists

Insertion and deletion of arbitrary items in the ordered linked list require O(N) comparisons (N/2 on the average) because the appropriate location must be found by stepping through the list. The minimum value, however, can be found, or deleted, in O(1) time because it's at the beginning of the list. If an application frequently accesses the minimum item, and fast insertion isn't critical, then an ordered linked list is an effective choice. Similarly, if the maximum item is needed much more frequently than the minimum and the same O(N) average insertion time is acceptable, the items could be ordered in descending order. If both the minimum and maximum were needed, a double-ended ordered list would be good. A priority queue might be implemented by a double-ended ordered linked list, for example.

List Insertion Sort

An ordered list can be used as a fairly efficient sorting mechanism for data in an array. Suppose you have an array of unordered data items. If you take the items from the array and insert them one by one into the ordered list, they'll be placed in order automatically. If you then remove them from the list, always deleting from the front and putting them back in the array by increasing indices, the array will be ordered.

This type of sort turns out to be substantially more efficient than the more usual insertion sort within an array, described in Chapter 3, "Simple Sorting," because fewer copies are necessary. It's still an $O(N^2)$ process because inserting each item into the ordered list involves comparing a new item with an average of half the items already in the list. There are N items to insert, which results in about $N^2/4$ comparisons. Each item is copied only twice, however—once from the array to the list and once from the list to the array. The $2\times N$ copies compare favorably with the insertion sort within an array, where there are about N^2 copies.

The downside of the list insertion sort, compared with an array-based insertion sort, is that it takes somewhat more than twice as much memory: the array and linked list must be in memory at the same time. If you have an ordered linked list class handy, however, the list insertion sort is a convenient way to sort arrays that aren't too large.

Doubly Linked Lists

Let's examine another variation on the linked list: the **doubly linked list** (not to be confused with the *double-ended list*). What's the advantage of a doubly linked list? A potential problem with ordinary linked lists is that it's difficult to traverse backward along the list. A statement like

```
current = current.getNext()
```

steps conveniently to the next link, but there's no corresponding way to return to the previous link. Depending on the application, this limitation could pose problems.

For example, imagine a text editor in which a linked list is used to store the text. Each text line on the screen is stored as a `String` object referenced from a link. When the user moves the cursor downward on the screen, the program steps to the next link to manipulate or display the new line. What happens if the user moves the cursor upward? In an ordinary linked list, you would need to return `current` (or its equivalent) to the start of the list and then step all the way down again to the new current link. This isn't very efficient. You want to make a single step upward.

The doubly linked list provides this capability. It allows programs to traverse backward as well as forward through the list. The secret is that each link keeps two references to other links instead of one. The first is to the next link, as in ordinary lists. The second is to the previous link. Figure 5-10 shows a double-ended version of this type of list.

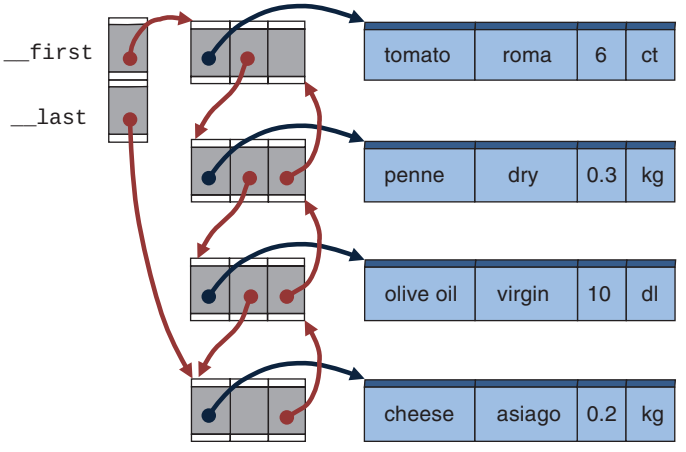


FIGURE 5-10 A doubly linked (and double-ended) list

The chain of links ends with a null pointer (an empty box in the figure and a value of `None` in Python) for both the forward and reverse chains. The downside of doubly linked lists is that every time you insert or delete a link, you must deal with four links instead of two—two attachments to the previous link and two attachments to the following one. That will become clearer when we look at the implementation a little later. It should be no surprise that each link record is a little bigger because of the extra reference needed for the reverse link.

Let's look at how to manage the extra linkages. The specification for the `Link` class in a doubly linked list (as a subclass of singly linked list) is shown in Listing 5-16.

LISTING 5-16 The `Link` Class of the `DoublyLinkedList.py` Module

```
import LinkedList

class Link(LinkedList.Link):            # One datum in a linked list
   def __init__(self, datum,            # Constructor with datum
                next=None,              # and optional next and
                previous=None):         # previous pointers
      self.__data = datum
      self.__next = next                # reference to next item in list
      self.__previous = previous        # reference to previous item

   def getData(self): return self.__data   # Accessors
   def getNext(self): return self.__next
   def getPrevious(self): return self.__previous
   def setData(self, d): self.__data = d
   def setNext(self, link):             # Accessor that enforces type
      if link is None or isinstance(link, Link):
         self.__next = link
      else:
         raise Exception("Next link must be Link or None")
   def setPrevious(self, link):         # Accessor that enforces type
      if link is None or isinstance(link, Link):
         self.__previous = link
      else:
         raise Exception("Previous link must be Link or None")

   def isFirst(self): return self.__previous is None
```

The definition for this `Link` class is only a little more complex than that for the singly linked list shown in Listing 5-2. The class definition makes the doubly linked version a subclass of the singly linked version and adds a private `__previous` field, which means that new accessor methods need to be defined for getting and setting its value. Just like for the `__next` pointer, the `setPrevious()` method enforces the constraint about the type of reference stored in that field. Most of the methods need to be rewritten in the subclass to correctly access the private fields in Python. Only the `isLast()` and `__str__()` methods remain the same in both `Link` classes. A new method for testing whether this is the first `Link` in a list, `isFirst()`, is now easy with the addition of the `__previous` field.

A doubly linked list doesn't necessarily need to be a double-ended list (keeping a reference to the last element on the list), but creating it this way is useful. The implementation of `DoublyLinkedList` shown in Listing 5-17 uses both a `__first` and `__last` pointer to make it easy to traverse the list in reverse order. It has the usual accessor methods for getting and setting the fields. The methods for setting the pointers enforce the type constraint on those fields and perform special checks to update both ends of the list when adding the first or deleting the last link.

LISTING 5-17 The `DoublyLinkedList` Class Constructor and Accessors

```
class DoublyLinkedList(LinkedList.LinkedList):
   def __init__(self):                  # Constructor
       self.__first, self.__last = None, None

   def getFirst(self): return self.__first # Accessors
   def getLast(self): return self.__last

   def setFirst(self, link):            # Set first link
      if link is None or isinstance(link, Link): # Check type
         self.__first = link
         if (self.__last is None or     # If list was empty or
             link is None):             # list is being truncated
            self.__last = link          # update both ends
      else:
         raise Exception("First link must be Link or None")

   def setLast(self, link):             # Set last link
      if link is None or isinstance(link, Link): # Check type
         self.__last = link
         if (self.__first is None or    # If list was empty or
             link is None):             # list is being truncated
            self.__first = link         # update both ends
      else:
         raise Exception("Last link must be Link or None")

   def traverseBackwards(               # Apply a function to all Links in list
         self, func=print):             # backwards from last to first
      link = self.getLast()             # Start with last link
      while link is not None:           # Keep going until no more links
         func(link)                     # Apply the function to the link
         link = link.getPrevious()      # Move on to previous link
```

The `DoublyLinkedList` class inherits the methods defined for the `LinkedList` class to traverse the list in the forward direction. More specifically, the `isEmpty()`, `first()`, `traverse()`, `__len__()`, and `_str()` methods of `LinkedList` all work without modification. A new `traverseBackwards()` method applies a function to each list item in the reverse direction.

Insertion and Deletion at the Ends

The next methods to look at are the ones for inserting and deleting links at the beginning and end of the doubly linked list. The `insertFirst()` method inserts at the beginning of the list, and `insertLast()` inserts at the end. Similarly, the `deleteFirst()` and `deleteLast()`

methods delete the first and last `Links` in the list. They are shown in Listing 5-18. Later we look at methods to insert and delete links in the middle.

LISTING 5-18 Methods to Insert and Delete the Ends of `DoublyLinkedLists`

```
class DoublyLinkedList(LinkedList.LinkedList):
… (other definitions shown before) …

   def insertFirst(self, datum):      # Insert a new datum at start of list
      link = Link(datum,              # New link has datum
                  next=self.getFirst()) # and precedes current first
      if self.isEmpty():              # If list is empty,
         self.setLast(link)           # insert link as last (and first)
      else:                           # Otherwise, first Link in list
         self.getFirst().setPrevious(link) # now has new Link before
         self.setFirst(link)          # Update first link

   insert = insertFirst               # Override parent class insert

   def insertLast(self, datum):       # Insert a new datum at end of list
      link = Link(datum,              # New link has datum
                  previous=self.getLast()) # and follows current last
      if self.isEmpty():              # If list is empty,
         self.setFirst(link)          # insert link as first (and last)
      else:                           # Otherwise, last Link in list
         self.getLast().setNext(link) # now has new Link after
         self.setLast(link)           # Update last link

   def deleteFirst(self):             # Delete and return first link's data
      if self.isEmpty():              # If list is empty, raise exception
         raise Exception("Cannot delete first of empty list")
      first = self.getFirst()         # Store the first link
      self.setFirst(first.getNext())  # Remove first, advance to next
      if self.getFirst():             # If that leaves a link in the list,
         self.getFirst().setPrevious(None) # Update its predecessor
      return first.getData()          # Return data from first link

   def deleteLast(self):              # Delete and return last link's data
      if self.isEmpty():              # If list is empty, raise exception
         raise Exception("Cannot delete last of empty list")
      last = self.getLast()           # Store the last link
      self.setLast(last.getPrevious()) # Remove last, advance to prev
      if self.getLast():              # If that leaves a link in the list,
         self.getLast().setNext(None) # Update its successor
      return last.getData()           # Return data from last link
```

The `insertFirst()`method first creates the new `Link` holding the new `datum`. The current first `Link` of the list is passed as the next link to follow the new `link`. The `__previous` field of the new `link` is left empty because this will be the first `Link` of the list. The next step is to update the `__first` (and possibly the `__last` field) to be the new `link`. There are two cases:

- If the list is empty, the method updates both with a single call to `setLast()`.
- If there are already some links, then it must change the reverse pointer from the current first link to point back at the new `link`. Then it replaces the first link with the newly created `Link`.

The process for inserting the string `"a"` into a doubly linked list containing `["A", "B"]` is shown in Figure 5-11.

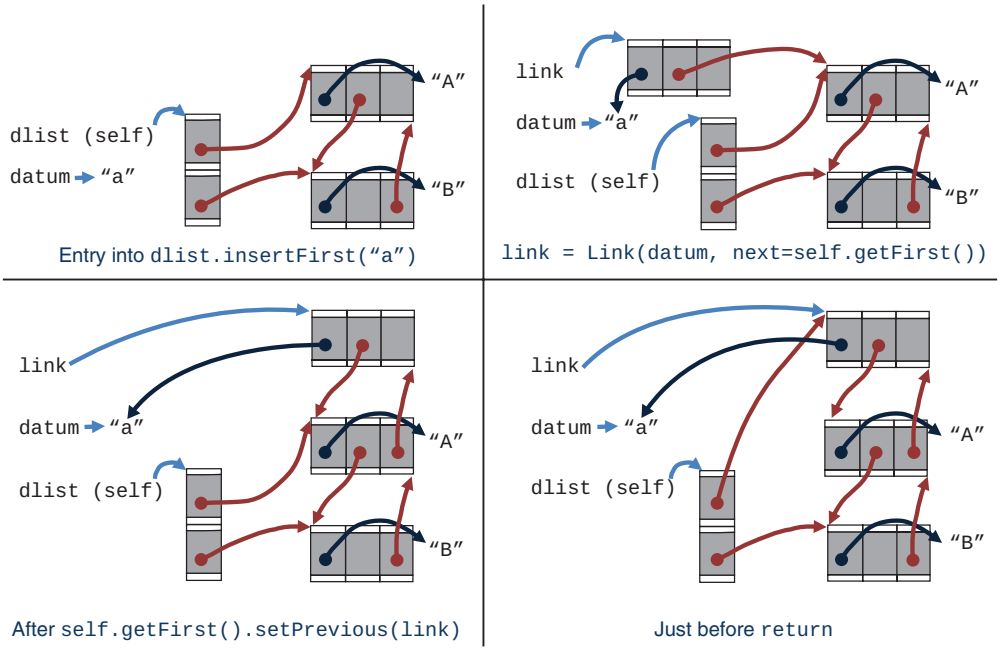


FIGURE 5-11 Insertion at the beginning of a doubly linked list

After defining the `insertFirst()` method, Listing 5-18 also redefines `insert()` to be synonymous with it. That redefinition is needed so that the simpler name in the parent class, `LinkedList`, refers to the same operation. The `insertLast()` follows the same structure but updates the `__last` field. It's something like a mirror image of `insertFirst()`, swapping first ⇄ last, and previous ⇄ next.

Deleting a `Link` from one of the ends of the list also involves carefully updating the pointers. The order of the updates is important so that you avoid removing a reference that is needed later. The implementation of `deleteFirst()` checks whether the list is empty and throws an exception if it is. If not, it stores a pointer to the first `Link`. That `first` pointer is used to get the data in that `Link` after it's been removed from the list. Then it advances the `__first` field to point at the second `Link`. The pointer to that second `Link` might be `None`. If it is, the `setFirst()` method also updates the `__last` field to be `None`. Otherwise, there is at least one more link in the list, and its `__previous` field must be updated to be `None` because it just became the first `Link`, as shown in Figure 5-12. Finally, the data from the deleted `Link` is returned.

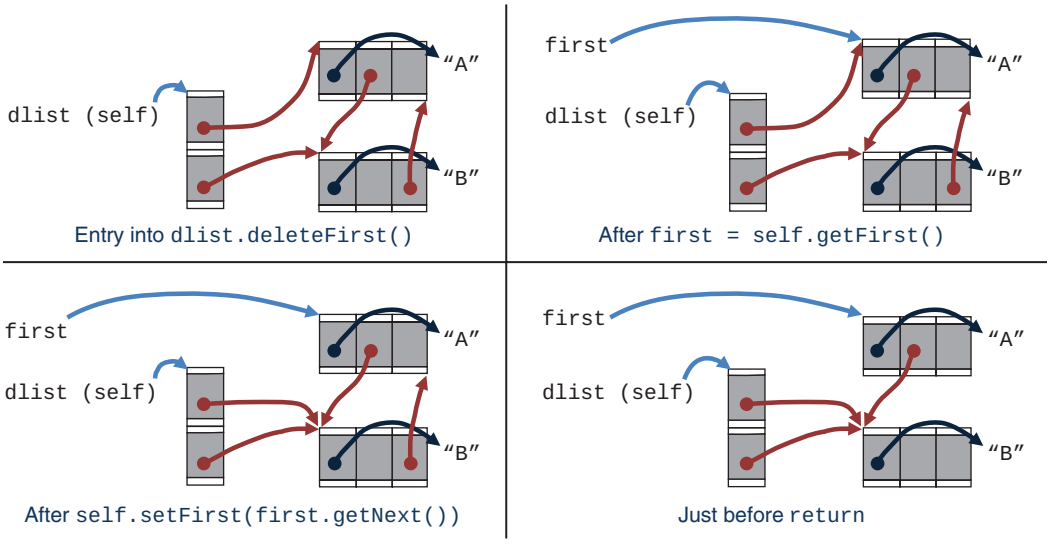


FIGURE 5-12 Deleting the first item of a doubly linked list

Insertion and Deletion in the Middle

Modifying `Links` in the middle of a doubly linked list is slightly more complicated because all four pointers need to be considered (the forward-reverse pointers both before and after). To specify where the operation should happen, typically you identify a `Link` with a matching key. For that, you need a `find()` method that returns a pointer to the link. Because the `DoublyLinkedList` class is a subclass of the `LinkedList` and that has a `find()` method, as shown in Listing 5-5, there's no need to rewrite one for `DoublyLinkedList`. The methods for inserting after and deleting target `Links` are shown in Listing 5-19.

LISTING 5-19 Methods to Insert and Delete Middle Links of `DoublyLinkedLists`

```
def identity(x): return x           # Identity function
…
class DoublyLinkedList(LinkedList.LinkedList):
… (other definitions shown before) …

   def insertAfter(                 # Insert a new datum after the
         self, goal, newDatum,      # first Link with a matching key
         key=identity):
      link = self.find(goal, key)   # Find matching Link object
      if link is None:              # If not found,
         return False               # return failure
      if link.isLast():             # If matching Link is last,
         self.insertLast(newDatum)  # then insert at end
      else:
         newLink = Link(            # Else build a new Link node with
            newDatum,               # the new datum that comes just
            previous=link,          # after the matching link and
            next=link.getNext())    # before the remaining list
         link.getNext().setPrevious( # Splice in reverse link
            newLink)                # from link after matching link
         link.setNext(newLink)      # Add newLink to list
      return True

   def delete(self, goal,           # Delete the first Link from the
              key=identity):        # list whose key matches the goal
      link = self.find(goal, key)   # Find matching Link object
      if link is None:              # If not found, raise exception
         raise Exception("Cannot find link to delete in list")
      if link.isLast():             # If matching Link is last,
         return self.deleteLast()   # then delete from end
      elif link.isFirst():          # If matching Link is first,
         return self.deleteFirst()  # then delete from front
      else:                         # Otherwise it's a middle link
         link.getNext().setPrevious( # Set next link's previous
            link.getPrevious())     # to link preceding the match
         link.getPrevious().setNext( # Set previous link's next
            link.getNext())         # to link following the match
         return link.getData()      # Return deleted data item
```

The `insertAfter()` method starts by using the `find()` method to find a `Link` whose key matches the given `goal`. If such a `Link` isn't found, then the method quits, returning `False` to indicate it failed. Alternatively, it could raise an exception or insert the data at the first or last position, depending on what's needed. After a `Link` with a `goal` key is

found, `insertAfter()` checks whether it is the last link, and if so, uses `insertLast()` to place it after the last link.

After handling the special cases, `insertAfter()` can deal with the case of inserting between two consecutive links of the list. It builds a new `Link` holding the new data with the previous pointer set to the goal `Link`, and the next pointer set to the following `Link`. This is shown in the third panel of Figure 5-13. The fourth panel shows how it alters the `__previous` pointer of the following `Link` and the `__next` pointer of the `goal Link` to point at the `newLink`. The order of these steps is very important.

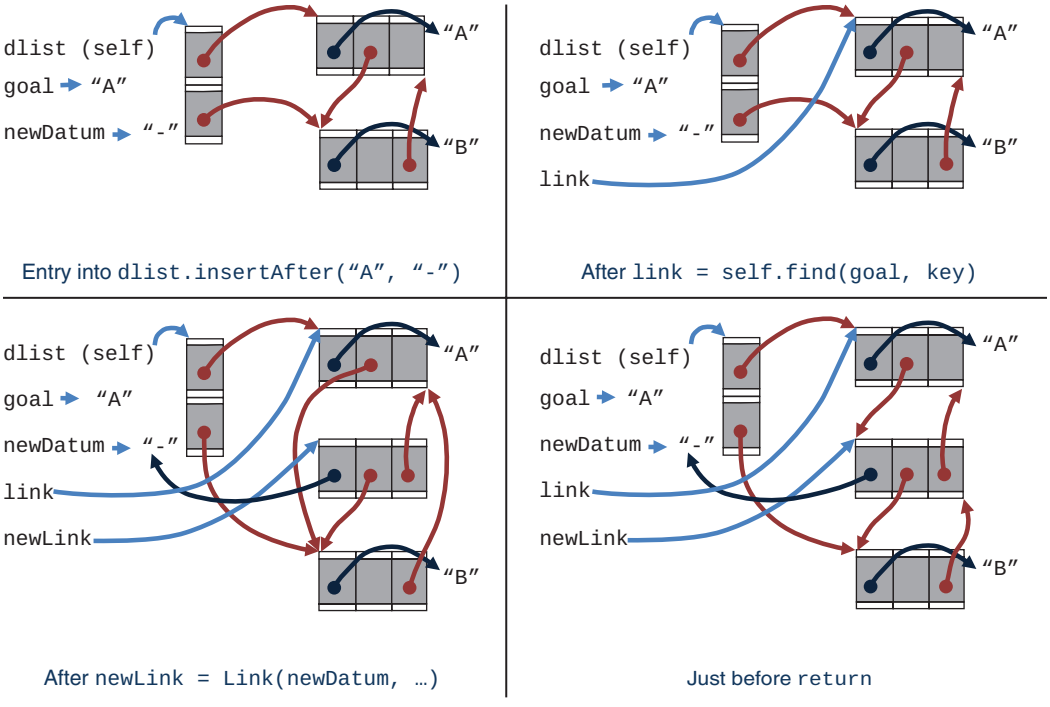


FIGURE 5-13 Insertion in the middle of a doubly linked list

The `delete()` method also starts by using the `find()` method to find a `Link` whose key matches the given `goal`. If such a `Link` isn't found, then it raises an exception. Alternatively, it could return `False` to indicate it failed. Having found the `Link` to delete, it checks whether it's the last or first link of the list. If it is, it uses the deletion methods for the ends. Otherwise, it needs to "snip" out the `goal Link` by altering the `__previous` and `__next` pointers of the `Links` after and before it, respectively. With the pointers modified, it returns the data stored in the `goal link`.

Note that the doubly linked list avoids the need to search for the `Link` preceding the `goal` for both insertion and deletion. The data structure stores that pointer for every `Link`, unlike singly linked lists. Do you think that's a simplification in terms of how complicated it is to implement the insertion and deletion methods? Opinions differ on that, but remember that this kind of complexity is very different from the computational

complexity of the algorithm. Complexity for the programmer is a completely separate concept from how efficiently the computer can perform the operations. Both are important for different reasons.

To verify the implementation of the doubly linked list, you can use a program like the `DoublyLinkedListClient.py` program shown in Listing 5-20. Note that programs like this exercise the basic operations of a data structure but are not truly comprehensive test programs.

LISTING 5-20 The `DoublyLinkedListClient.py` Program

```
from DoublyLinkedList import *

dlist = DoublyLinkedList()

for data in [(1968, 'Richard'), (1967, 'Maurice'), (1966, 'Alan')]:
   dlist.insertFirst(data)
for data in [(2015, 'Whitfield'), (2015, 'Martin'),
             (2016, 'Tim'),
             (2017, 'David'), (2017, 'John')]:
   dlist.insertLast(data)
print('After inserting', len(dlist),
      'entries into the doubly linked list, it contains:\n', dlist,
      'and empty =', dlist.isEmpty())

print('Traversing backwards through the list:')
dlist.traverseBackwards()

print('Deleting first entry returns:', dlist.deleteFirst())
print('Deleting last entry returns:', dlist.deleteLast())
def year(x): return x[0]
for date in [1967, 2015]:
   print('Deleting entry with key', date, 'returns',
         dlist.delete(date, key=year))
print('List after deletions contains:', dlist)

for date in [1968, 2015]:
   data = (date + 1, '?')
   print('Inserting', data, 'after', date, 'returns',
         dlist.insertAfter(date, data, key=year))
print('List after insertions contains:', dlist)

print('Traversing backwards through the list:')
dlist.traverseBackwards()
```

The program in Listing 5-20 builds a doubly linked list holding pairs of a year and a name. It shows the contents of the list both in the string format, which steps through the list in

the forward direction, and by calling the `traverseBackwards()` method to print each data item in reverse order. It tests deletion at both ends of the list, followed by deleting some entries by using the year as a key (it defines a `year()` function that returns the first entry of each pair). The items are printed as they are deleted, and the remaining list is printed after that. Then it inserts some new pairs after items with specific dates and prints the list both forward and backward to show that the linkages are preserved after all the changes. The output is

```
$ python3 DoublyLinkedListClient.py
After inserting 8 entries into the doubly linked list, it contains:
 [(1966, 'Alan') > (1967, 'Maurice') > (1968, 'Richard') > (2015, 'Whitfield') >
(2015, 'Martin') > (2016, 'Tim') > (2017, 'David') > (2017, 'John')] and empty =
False
Traversing backwards through the list:
(2017, 'John')
(2017, 'David')
(2016, 'Tim')
(2015, 'Martin')
(2015, 'Whitfield')
(1968, 'Richard')
(1967, 'Maurice')
(1966, 'Alan')
Deleting first entry returns: (1966, 'Alan')
Deleting last entry returns: (2017, 'John')
Deleting entry with key 1967 returns (1967, 'Maurice')
Deleting entry with key 2015 returns (2015, 'Whitfield')
List after deletions contains: [(1968, 'Richard') > (2015, 'Martin') > (2016, 'Tim')
> (2017, 'David')]
Inserting (1969, '?') after 1968 returns True
Inserting (2016, '?') after 2015 returns True
List after insertions contains: [(1968, 'Richard') > (1969, '?') > (2015, 'Martin')
> (2016, '?') > (2016, 'Tim') > (2017, 'David')]
Traversing backwards through the list:
(2017, 'David')
(2016, 'Tim')
(2016, '?')
(2015, 'Martin')
(1969, '?')
(1968, 'Richard')
```

Doubly Linked List as Basis for Deques

A doubly linked list can be used as the basis for a deque, mentioned in the preceding chapter. In a deque, you can insert and delete at either end, and the doubly linked list provides this capability. A programming project at the end of this chapter has you implement this ADT.

Circular Lists

All the lists we've looked at so far have a defined beginning and end. Occasionally, it's useful to create a chain of links that forms a circle or loop. For example, a chain of links could represent the players in a game where the turn passes from one player to the next and returns to the first when all players have had their turn. In some sports the players are organized in a rotation, taking turns at serving or batting. In chemistry there are chains of molecules that form rings. In biology there are organisms that depend on other organisms, and the dependencies sometimes form loops. The obvious question then becomes, "Where does the circular list end?" In each of these examples, there really isn't an end, but there's a need for a marker, or sometimes multiple markers, to designate a specific member of the list for various purposes.

Using a "circular" array to implement a queue is a convenient way to reuse array cells for storage after they had been inserted at one end and removed from the other. If the queue length was less than the size of the array, some cells of the array were not used. Circular lists that are implemented using references between objects avoid having unused elements. That design saves space but introduces complications like determining the size of the list and traversing each element of the list exactly once. The double-ended list is a better design for a queue because it doesn't have unused elements and is simpler than circular lists. There can be other uses, however, where a circular list might help.

Figure 5-14 shows a typical way to represent circular lists. The list is singly linked, but it could be doubly linked if reversing the direction of traversal is important. It needs a marker for one of the elements to designate it as the last (or current or first), just like the `LinkedList` points at an individual `Link` object. The figure doesn't show any data stored in the links. Any data would be stored using references, the same way as the other linked lists we've discussed.

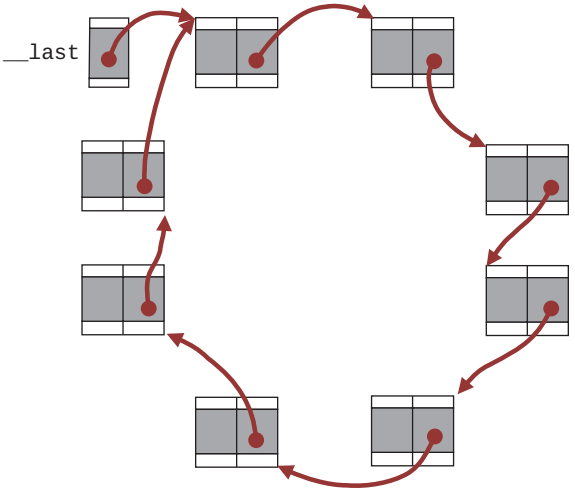


FIGURE 5-14 A circular linked list (without data)

If the only thing stored for the marker is a pointer to one of the `Links`, then traversal requires following the pointers until the loop revisits the marked `Link`. That's no more complicated than following the pointers until `None` is found. By choosing to mark the last `Link` in the chain, you can find the first `Link` in two steps, making both `insert()` (or `insertFirst`) and `insertLast()` methods constant time operations.

The biggest problem with circular lists is ensuring they remain circular. The individual `Link` objects would typically offer a `setNext()` method that would be used by the `CircularList` class to insert and delete links from the structure. If that `setNext()` method is public and callable by the code using the `CircularList` class, then the caller could set a link pointer to some other `Link`, or possibly set it to `None`, to form a noncircular, or linear, list. These can take unusual shapes, like the ones shown in Figure 5-15.

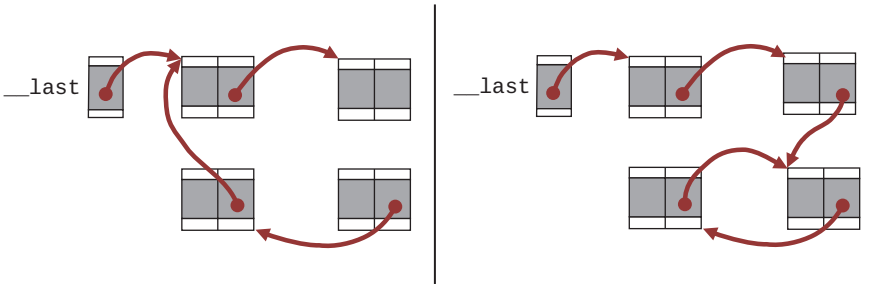


FIGURE 5-15 Broken circular lists (without data)

Coping with anomalies in circular lists can be difficult. If `None` is allowed as a value for the next pointer, then the traversal loop conditions must look for that value in addition to finding the first visited link when determining the end of a traversal loop. Even that, however, won't help when the `__last` pointer is not pointing to a `Link` in a loop, as shown on the right in Figure 5-15 where the `__last` field points at a "spur" off the loop. Traversal would start on the spur and progress onto the circular part. That means the loop condition must compare the current pointer to not only the first pointer but to *all* `Links` visited so far. That's not much of a concern for a small circular list, but as the number of links grows large, it becomes very time-consuming. Each of the *N* steps in the traversal must compare the current link with up to *N* other links, so the total number of comparisons appears to be $O(N^2)$.

To avoid such problems, circular list implementations must avoid making the `setNext()` method public and offer other methods for altering the list that guarantee it remains a single loop with no spurs or breaks. Similarly, linked lists of all kinds (simply linked, doubly linked, double-ended) must not allow public `setNext()` methods to be used to create what is essentially a circularly linked list. If they do, then the traversal methods that terminate when `None` is found will never find it.

Iterators

All the linked lists and array structures you've studied have included a `traverse()` method. That method applies a function on every item stored in the data structure, in a known order. Visiting every item is very useful, for instance, when you need to update all the personnel records to assign a new identification code to every one of them or to print a mailing label for every record. The function passed to `traverse()` takes a single argument, an item stored in the list, and performs some operation.

Calling a function on every item works well when the operation to be performed on each item is independent of the operations performed on all the other items. On the other hand, operations that combine or have some interaction between the items are not as easy to handle with a single function on one item. For example, if you needed the average of some numeric field in each record, the function would need to collect the sum somewhere and then divide it by the number of items at the end. Another difficult example would be to collect pairs of records where some fields in each pair don't match one another.

For combining data across all the items in a data structure, it's more convenient to have some way to traverse the items in a loop inside the calling program. That allows the caller to keep track of sums, minimums, maximums, the last value processed, and so on. What's needed is a way to iterate over the items in the context of another program. Iterating this way is straightforward for arrays where the items are indexed by an integer and any item can be fetched in constant time. Data structures like lists and some of the more complex ones we study later don't have a way to access arbitrary items in constant time but can follow a defined order to step through all the items with constant time access for each step.

An **iterator** is a data structure that provides methods to step through all the items contained in another data structure, visiting each item exactly once. The order of the items is fixed; each iterator for a particular object goes through the sequence of items in the same order as long as the contents of the object don't change. The iterator must provide at least one method, usually called `next()`, `getNext()`, or something similar, that is used to get the next item in the sequence. The other methods it provides are somewhat dependent on the data structure it steps through and the language used to implement it.

The object that the iterator steps through is sometimes called a **collection**, a **container**, a **sequence**, or an **iterable**. The choice of which term to use is somewhat dependent on the kind of data structure being sequenced. Python has sequence data types, such as `list` and `str`. There is a natural order of the items in those structures, so they are appropriately named. Other data structures like hash tables (dictionaries), trees, and sets have less obvious orderings. The items they contain, however, can still be put into a sequence that the iterator steps through.

There's another benefit to iterators: they can represent infinite sequences and sequences of indefinite length. For interactive programs, the various inputs coming in, like clicks, keystrokes, touches, sounds, camera images, and network packets, can be handled by iterators. It's usually unclear how many of these inputs will happen. The program that

consumes them only needs to know that they come in order and that it should continue processing them until some termination code is received. In math, there are infinite sequences, such as all the integers or all the prime numbers. It's impractical to make a data structure that holds an infinite number of things, but in many cases, you can make one that steps through that infinite sequence in a particular order.

Python uses the term *iterable* for any class that can produce an iterator. We use that term in this book to describe any object being sequenced, although that term has a more specific meaning in Python. Let's examine how to construct iterators in general and then look at the specifics of how to implement them in Python.

Basic Iterator Methods

An iterator is a data structure in its own right that references another data structure. The iterator, however, is strongly dependent on the iterable, so they are frequently implemented together in the same source code. Because the iterator is a data structure, it is an object like other objects, and needs a constructor. Instead of defining a public class like `LinkedListIterator`, you often define the constructor for the iterator as a method of the iterable data structure. That constructor method could be called `iterator()`. For example, you could construct a linked list, fill it with some data, and create an iterator for it by executing

```
llist = LinkedList()
llist.insert(<some data>)
  …
it = llist.iterator()
```

If you make a public class for `LinkedListIterator`, then the iterator would be constructed using something like

```
it = LinkedListIterator(llist)
```

The choice is mostly stylistic. The advantage of using a method in the iterable class is that it can be the same method name for every iterable class. Programmers who use the data structure don't need to hunt for what the iterator class name is; they can just call `iterator()`.

After it is constructed, the caller gets the items by calling the iterator's `next()` method. The caller typically does this in a loop until all the items have been processed. Sometimes the iterator interface also has a method like `current()`. This is somewhat like the `peek()` method used in queues to look at the value at the front of the queue without removing it from the queue. Calling `next()`is like calling `pop()` from a stack; it changes the iterator to move on to the next item and returns an item. If the iterator has a `current()` method, then it must return the first item when the iterator is constructed (before any calls to `next`).

Finishing the Iteration: Using a Marker or Sentinel Value

Because the iterator is likely to be used in a loop, it needs a good interface for handling the start and end of loops. If the iterable is empty, then the loop shouldn't be entered at all. If there are some items, then the loop must end after the iterator reaches the last one. If there is a known value that could never be an item stored inside the iterable, then the loop condition could test for that. For example:

```
while it.current() is not None:
   <loop body>
   it.next()
```

This loop looks at the current value, and if it is not the marker for the end of the iteration (`None` in this example), it executes the loop body and advances to the next item. The question then becomes how to handle the end of a sequence where current could take on any value, including `None`.

Finishing the Iteration: Using a Termination Test

Another strategy for loop termination is to define a method like `hasMore()` that provides a Boolean value to say whether another call to `next()` will produce a value. That means the loop would look like this:

```
while it.hasMore():
   current = it.next()
   <loop body>
```

This termination test strategy eliminates the need for a known value as a marker for the end of the sequence. That allows storing `None`—or any other value—as one of the items in the list.

Finishing the Iteration: Using Exception Handling

Yet another strategy is to use exception handling. If the iterator has a `current()` method, then calling it or `next()` when the iterable is empty could cause an exception. If there is no `current()` method, then calling the `next()` method will cause an exception when it reaches the end of the sequence. We haven't discussed file data structures yet, but this is a common way to handle the end of a file or user input. In these cases, you need some exception handling around the iteration loop to detect the end of the sequence. For example, if the end of the sequence causes a `StopIteration` exception to be thrown, then the iteration loop takes the form:

```
try:
    while True:
       current = it.next()
       <loop body>

except StopIteration:
    pass
```

This alternative is definitely longer in terms of number of lines of code. It's also a bit harder to follow the flow because the loop body is now two levels deep and there is an exception handler at the end. Before dismissing it based on that, however, note that it also shares the advantage that there is no special known value for the end of the sequence, and it doesn't require a `hasMore()` method to check for more items. The program can try to get another item from the iterable, perhaps pausing execution while waiting for new input to be added to the iterable, and either return the new item or throw an exception to say no more could be found.

Normally, `next()` returns the item stored in the object it sequences, and not a reference to the internal structure that holds a reference to the item. For example, in a linked list, `next()` would return the item stored in the `__data` field of a `Link` object, and not a reference to the `Link` object itself. That's important for preserving the integrity of the references between links.

Listing 5-21 shows a simple implementation for an iterator that handles two of these alternatives for termination. It iterates over the singly linked list (whose other definitions appeared in Listings 5-3, 5-4, 5-5, and 5-6). The iterator is a class of its own. In this case, we've chosen to make it a private class called `__ListIterator` that is *defined within* the `LinkedList` class. The rationale for keeping it private is that there should be no need to make subclasses or perform other operations on the iterator class.

LISTING 5-21 Simple Iterator for a `LinkedList`

```
class LinkedList(object):

… (other definitions shown before) …

   class __ListIterator(object):          # Private iterator class
      def __init__(self, llist):          # Construct an iterator over a
         self._llist = llist              # linked list
         self._next = llist.getNext() # Start at first Link

      def next(self):                     # Iterator's next() method
         if self._next is None:           # Check for end of list
            raise StopIteration           # At end, raise exception
         item = self._next.getData()      # Store next data item
         self._next = self._next.getNext() # Advance to following
         return item

      def hasMore(self):                  # Is there more to iterate?
          return self._next is not None  # Check for end of list

   def iterator(self):
      return LinkedList.__ListIterator(self)
```

The implementation of the `__ListIterator` class has a constructor that records the linked list iterable in a field named `_llist`. It initializes a second field called `_next` to point at

the first `Link` of the linked list. If the list is empty, then `_next` will be `None`. The constructor is used by the `LinkedList`'s `iterator()` method at the end of Listing 5-21 to create the iterator object. It's a little unusual in that it passes the `self` variable explicitly to the constructor; that approach is needed to distinguish the `LinkedList` object from the `__ListIterator` object.

The iterator class's `next()` method steps through the list and throws an exception at the end. The `next()` method uses the information stored in the iterator object to step through each of the items of the list. It first checks whether the internal `_next` pointer is `None`. If so, then either the iterator started on an empty list, or past calls to `next()` have advanced to the end of the list, and it throws the `StopIteration` exception. This is a predefined exception in Python, but it could just as easily be a user-defined exception class. If you choose this particular exception, it operates like Python's implementation of iterators, as discussed in the later "Iterators in Python" section.

When `_next` points to a `Link` object, then the `next()` method stores the item from that link; advances to the next link, which could be `None`; and returns the stored item. To use this iterator, a calling program would use a loop like this:

```
it = llist.iterator()
print('Created an iterator', it)
try:
   while True:
      print('The next item is:', it.next())
except StopIteration:
   print('End of iterator')
```

Executing that program on a list of squares would produce something like this:

```
Created an iterator <LinkedList.__ListIterator object at 0x1100d5668>
The next item is: 16
The next item is: 9
The next item is: 4
The next item is: 1
The next item is: 0
End of iterator
```

The implementation of `__ListIterator` in Listing 5-21 also defines a `hasMore()` method. Thus, the termination test strategy can be used by writing something like

```
it = llist.iterator()
print('Created an iterator', it)
while it.hasMore():
   print('The next item is:', it.next())
print('End of iterator')
```

This kind of multiple strategy iterator supports different programming styles.

Other Iterator Methods

The basic function of the iterator is to step through the sequence of items stored in another data structure, but it's sometimes convenient to add more capabilities. As mentioned earlier, examining the next item without calling `next()` (which advances the iterator to the following item) can be useful, such as when merging two ordered lists. You could build iterators for both lists, examine the first element from each one, and advance the iterator whose next item comes first in the sort order. A method named `getCurrent()` or `current()` or `peek()` is typically used for this purpose.

Another useful function is to point the iterator back at the beginning of the sequence. This might be used when identifying items that should get awarded labels in some priority order. Imagine handing out some delicious treats to the top five finishers of a race. One iterator could loop through the treats while another loops through the racers in the order they finished. Each racer is asked if they would like the treat currently being offered. If they accept, the treat iterator advances to the next treat, and the racer iterator resets back to the beginning. That way, any racer who finished faster gets priority for the next treat. (Of course, racers who already selected treats don't get to select again.)

For this kind of reset operation to work, the iterator must maintain a reference to the original iterable data structure. In the linked list example, that reference is stored in the `_llist` field, so a definition like

```
def reset(self):            # Reset iterator to first link
   self._next = self._llist.getNext()
```

could be used to return to the beginning.

Altering Structures During Iteration

Some iterators provide methods to alter the iterable data structure. This operation is very tricky because changing the data structure could change the items it contains and/or the sequence those items should be visited. For example, if the last item from a list were to be deleted, presumably the iterator would skip over that item after iterating up to the second-to-last item. In the example of assigning treats to racers, it would be convenient to remove a racer from the iterator after they chose a treat. Taking them out of those subsequent iterations, however, is difficult to do.

It's somewhat easier to plan for such deletions by restricting them to be done only by the iterator so that any references to the iterable data structure can be updated appropriately. If alterations to the iterable happen somewhere else in the code, however, it is impossible for the iterator to properly handle all the conditions that might arise. For example, if, after creating an iterator on a linked list using the `iterator()` method in Listing 5-21, some other part of the code deletes the first item in the list, what should happen? The internal `_next` pointer of the iterator points to a `Link` that is no longer part of the iterable `_llist`, and the subsequent call to `next()` would return an item that's not in the list. It's not clear where `next()` would advance the `_next` pointer to because the deleted `Link` could have had its pointer altered by the outside code.

When changes to the data structure are allowed while stepping through the iterable sequence, it's best that all changes to the sequence of items are performed through the iterator. Methods like `insertAfter()`, `insertBefore()`, and `deleteNext()` can be defined to change the list in the vicinity of the next item while still allowing for the rest of the sequence to be traversed. The `insertBefore()` method can be efficiently implemented if the list is doubly linked or the iterator maintains a reference to the previously visited `Link`. The `insertAfter()` method can be implemented with a singly linked list. There are two alternatives to handling the `_next` pointer after the insertion: include the newly inserted item or skip over the newly inserted item in the remaining sequence of the iterator. If the `_next` pointer is not advanced, then multiple insertions could add several items that would then be visited in the reverse order of their insertion. If the `_next` pointer is advanced, then multiple items could be inserted and would retain the order of insertion (and be skipped by the iterator sequence). The `deleteNext()` method would remove an item from the iterable and from the iterator sequence.

As you can see, mixing iteration over an iterable sequence with alterations to the iterable can cause horrendous problems. It's best to prevent alterations while iterating.

Iterators in Python

Python makes traversing data structures very easy with special support for iterators. In fact, all the `for x in [a, b, c]:` style loops you've been using are implemented using iterators. Much of the detail described previously for implementing iterators is performed by Python without any visible (user) source code. It's important to learn about how that works as you implement data structures in other languages with different syntaxes.

As you might expect, the built-in `list` type is iterable. Strings and tuples are also iterable. The general syntax for a loop using an iterator is

```
for var in <iterable>:
    <body>
```

Looking under the hood, the way that Python executes a loop in this form is to evaluate the *<iterable>* expression on entry into the loop. The *<iterable>* expression produces an iterator object that is not directly available to the program but is maintained by the Python interpreter. The interpreter then creates a new local variable, *var*, and assigns it a value by calling the `next()` method of the iterator. More precisely, the iterator is created by calling the `__iter__()` method of the object returned by evaluating the *<iterable>* expression.

With the iterator created, the interpreter then calls its `__next__()` method to get the next item in the sequence. After the variable is assigned, it evaluates the loop body. When the body finishes, the interpreter calls the `__next__()` method to obtain the next item in the sequence. When the iterator goes past the last item in the data structure, the `__next__()` method raises the `StopIteration` exception. That exception could happen on the first attempt to call the `__next__()` method, in which case the variable is not assigned, and the loop body is not evaluated at all.

To be a little more concrete, let's look at an example where *var* is `v` and the <*iterable*> expression is `expr`. The Python interpreter essentially executes the following:

```
it = expr.__iter__()
try:
    while True:
       v = it.next()
       <body>
except StopIteration:
    pass
```

This isn't exactly the translation of the `for` *var* `in` <*iterable*>`:` loop; the `it` variable is hidden, and the `v` variable is accessible only in the loop <*body*>, just like local variables are accessible only within a function definition. The rewritten version shows, however, the basic steps the interpreter takes. The `__iter__()` method creates the iterator object, `it`, and its `__next__()` method is called in each pass through the loop. The loop creates a hidden exception handler for `StopIteration`. When that exception occurs, the "infinite" `while` loop stops, and execution passes on to whatever comes after the loop.

If any other exceptions happen in the loop body, they are ignored by the hidden exception handler and passed on to the next containing level of the program. If there are nested loops in the <*body*>, they create their own exception handlers that catch the end of the iteration for each of those loops without affecting the end of this outer loop. Because the iterator and the exception handler that catches its termination are not directly accessible in the source code, simple Python loops provide an *implicit iterator* rather than an *explicit* one like in Listing 5-21.

Generators

Any Python object class that implements an `__iter__()` method becomes an iterable object. The iterator object it returns must implement both a `__next__()` and an `__iter__()` method. These methods could be written using a hidden class like the one shown in Listing 5-21, but there's an easier way: **generators**. Generators are functions that create iterators. Python has a special statement, the `yield` statement, that takes care of creating the iterator class and its required methods. Let's look at an example using a mathematical sequence.

One of the most famous infinite sequences is the Fibonacci sequence. It occurs in nature in the fruit sprouts of a pineapple, flowers of an artichoke, and bracts (petals) of pinecones. It starts with the number 1, repeated twice. Each subsequent number in the sequence is the sum of the previous two numbers. The sequence begins

1, 1, 2, 3, 5, 8, 13, 21, …

You can write a function that prints out all the Fibonacci numbers like this:

```
def Fibonacci():
   previous = 0
   current = 1
```

```
   while True:
      print(current)
      next = previous + current
      previous = current
      current = next
```

This function never ends because it prints out each number in the sequence (so be careful when executing it). You can convert this function into a generator with a simple change, replacing the `print` statement with a `yield` statement, as shown in Listing 5-22.

LISTING 5-22 A Python Generator for the Fibonacci Sequence

```
def Fibonacci():
   previous = 0
   current = 1
   while True:
      yield current
      next = previous + current
      previous = current
      current = next
```

The `yield` statement is a little bit like the `return` statement of a function. It causes the execution of the function to stop and return the value of the expression to its caller. What's very different from the `return` statement is that the Python interpreter preserves the state of execution of the function so that the next time it is called, execution resumes just after the `yield` statement and continues the processing. Continuing the processing makes the `yield` statement act more like the `print` statement it replaced. You might think of `yield` as sending a message back to the caller without losing track of what happens next.

When the Python interpreter finds the `yield` statement inside the definition of `Fibonacci()` in Listing 5-22, it changes the defined function to a special form called a *generator function*. The presence of `yield` means that this function shouldn't execute like most functions. When the generator function is called, it returns a generator object, which is a kind of iterator. That means you can use it to step through the sequence using `__next__()`. You can try it out in the Python interpreter using the definition in Listing 5-22:

```
>>> gen = Fibonacci()
>>> gen
<generator object Fibonacci at 0x1012386d0>
>>> gen.__next__()
1
>>> gen.__next__()
1
>>> gen.__next__()
2
>>> gen.__next__()
3
```

The call to `Fibonacci()` produces an object that is stored in the `gen` variable. That object is a generator object. Because it's an iterator as well, it must have a `__next__()` method. The first call to `__next__()` produces (yields) the first number in the sequence, 1. Inside the interpreter, that first call starts the execution of the body of the `Fibonacci()` definition up to the first `yield` statement. That operation yields the value of `current`, which is initially 1. The second call to `__next__()` yields 1 again after the execution of the body resumes just after the `yield` statement. Let's look more carefully at how that happens.

After yielding the first 1, the second call to `__next__()` causes execution to resume in `Fibonacci()` after the `yield` statement. The `current` and `previous` variables have the same values they had during the first call, 0 and 1, because they are stored in the generator object. The next step in executing the loop body stores a new value, 1, in the `next` variable. Then it updates the `previous` and `current` values, returns to the top of the loop, and hits the `yield` statement again. The third call to `__next__()` yields 2 as the process repeats. The generator object always keeps the values of `next`, `current`, and `previous` stored as part of the state of execution of the body so that it can resume right after the `yield` statement.

Having defined `Fibonacci()` as a generator, you can use it where iterable objects are expected like loops. The following example demonstrates that with a loop and a counter to print the first 15 numbers in the Fibonacci sequence:

```
count = 15
print('The first', count, 'numbers in the Fibonacci series are:')
for x in Fibonacci():
   if count < 1:
      break
   print(x)
   count -= 1
```

Note that the variable `x` is not bound to the value of the call to `Fibonacci()`. That generator object gets stored in some inaccessible location. Instead, the variable `x` gets bound to the values yielded by the successive calls to `__next__()`. The program outputs

```
The first 15 numbers in the Fibonacci series are:
1
1
2
3
5
8
13
21
34
55
```

```
89
144
233
377
610
```

This generator object could keep on producing Fibonacci sequence numbers indefinitely, so it essentially represents the infinite series in a finite data structure! The definition of `Fibonacci()` has a `while True` loop in it. That's what makes it go on forever. Using the `break` statement and decrementing the `count` prevent an infinite loop. This can be a powerful combination.

Python accomplishes this feat by creating an iterator data structure that holds the body of the function definition, the environment within that function (the state of all its parameters and local variables), and a pointer to where execution should resume in that function when `__next__()` is called. These combined "housekeeping" features make it easy to write generators that will implement the iterators for data structures.

Listing 5-23 shows one way to create the iterator for the singly linked list data structure using the `yield` statement in Python. It defines an `__iter__()` method that looks a lot like the `traverse()` method (Listing 5-4). Compare this definition of `__iter__()` to the `iterator()` method in Listing 5-21, which involved defining an internal class for the iterator.

LISTING 5-23 Using a Python Generator to Make the `LinkedList` Iterator

```
class LinkedList(object):

… (other definitions shown before) …

   def __iter__(self):          # Define an iterator for the list
      next = self.getFirst()    # Start with first Link
      while next is not None:   # As long as the link is not None,
         yield next.getData()   # yield data for the link
         next = next.getNext() # then move on to next link
```

Defining the `__iter__()` method as a generator function instead of a regular function takes care of building the extra class needed for an iterator. The iterator still needs to start with a pointer to the first `Link`, stored in its local `next` variable, and follows the chain of references until a `None` is found. For valid links, the iterator yields the data and moves on to the next link. When there are no more links, the loop ends, and the generator exits, raising the `StopIteration` exception. Then when the generator is invoked by a loop like

```
for item in myList:
   print(item)
```

Python catches the `StopIteration` exception with an implicit handler to break out of the `for` loop after printing each `item`.

It might seem a little unsettling that changing `return` statements to `yield` statements in a Python function has such a big effect on the behavior of the code. When you use generators, a lot of work is done by the compiler without any explicit variables or control statements in the source code. When you're reading code that someone else wrote, it won't be clear whether something is a subroutine, function, or generator until you locate a specific `return` or `yield` statement. If you find a `return`, then it's a function, possibly with explicit values to be returned. If you find `yield`, then it's a generator that will first return an iterator.

Every iterator can produce a sequence of values. When the iterator finishes the sequence (assuming it has an end), it raises a `StopIteration` exception. Note the source code of the generator doesn't have an explicit `raise StopIteration` statement. The exception is raised sort of like the implicit `return None` when execution reaches the end of the body of a function. In fact, in Python 3.7 and later, a generator that explicitly raises a `StopIteration` exception causes a `RuntimeError`. Instead, you should use the implicit end of a function or explicit `return` with no argument to signal the end of the sequence.

Using `yield` statements is a compact way to define iterators, although the special behavior of the generator function can be a little hard to understand initially. The Programming Projects in this chapter give you a chance to practice writing some.

Summary

- A linked list consists of one `LinkedList` object and a number of `Link` objects.
- The `LinkedList` object contains a reference to the first link in the list.
- Each `Link` object contains two things: data and a reference to the next link in the list. The data could be the data itself (like integers, characters, or bytes) or some reference to another, possibly larger, structure.
- A special value (typically `None` in Python) in the reference to the first or next `Link` signals the end of the list.
- Inserting an item at the beginning of a linked list involves creating a new `Link` with its data field holding or pointing to the item, changing the new `Link`'s next field to point to the old first link, and changing the `LinkedList`'s first field to point to the new `Link`.
- Deleting an item at the beginning of a nonempty list involves setting the first field to point to the first `Link`'s `next` field.
- To traverse a linked list, you start at first `Link` and then go from link to link, using each link's `next` field to find the next link.
- Some implementations use the same named field or accessor method for the first reference of the `LinkedList` as is used for the next reference of each `Link`. This makes the traversal code simpler.

- A link with a specified key value can be found by traversing the list.
- A new link can be inserted before or after a link with a specified key value, following a linear search to find this link.
- A double-ended list maintains a pointer to the last link in the list as well as to the first.
- A double-ended list allows insertion at the end of the list in constant time rather than O(N) time for a singly linked list.
- An abstract data type (ADT) is a data storage class considered without reference to its implementation.
- Stacks and queues are ADTs. They can be implemented using either arrays or linked lists.
- In an ordered linked list, the links are arranged in order of ascending (or sometimes descending) key value.
- Insertion in an ordered list takes O(N) time because the correct insertion point must be found. Deletion of the smallest (or sometimes largest) link takes O(1) time.
- In a doubly linked list, each link contains a reference to the previous link as well as the next link.
- A doubly linked list permits backward traversal and deletion from the end of the list.
- Circular lists are similar to singly linked lists, but there is no last item. Instead of having a next pointer set to `None` for the last item, that item points back to the first `Link`.
- Circular lists have one item designated as a marked item. It can be called first, last, head, tail, mark, and so on. This is the `Link` pointed to by the circular list object.
- Singly linked circular lists have the same efficiency as linear linked lists. By marking the last item instead of the first, the `insert()` and `insertLast()` methods on a circular list take O(1) time instead of O(N).
- Circular lists must be carefully updated to preserve the single loop structure to avoid making it difficult to terminate loops through all the items.
- An iterator is a kind of reference, encapsulated in its own class, that points to one item in another data structure. When the data structure is a linked list, the iterator references a particular link in that list.
- Iterator methods allow the calling program to move the iterator sequentially through all the items of the data structure. For linked lists, this follows the next pointers in each link.
- The data structure being sequenced by the iterator is called an iterable or a container.

- Iterators allow access to the data stored in the iterable without direct access to the iterable's structures and references (`Links` in a linked list) to avoid allowing the calling program to alter the internal structure.
- An iterator can be used to traverse through a list, performing some operation on selected links (or all links).
- If the iterable changes while an iterator is stepping through its items, the iteration sequence can be altered, sometimes to the point of causing bad errors.

Questions

These questions are intended as a self-test for readers. Answers may be found in Appendix C.

1. Which of the following is **not** true? A reference to a class object
   a. can be stored in the data field of a singly linked list.
   b. can be used to access public methods in the object.
   c. has a size dependent on its class.
   d. does not hold the object itself.
2. Access to the links in a linked list is usually through the _____ link.
3. Lists differ from arrays in that
   a. lists have a fixed size per item, whereas arrays don't.
   b. the relationships between items are explicit in lists but not in arrays.
   c. position is only explicit in arrays; the key of a list item determines its position.
   d. to get the Nth item in a list, a program must follow N links, whereas in an array, the program can compute the position of the item from N.
4. How many references must be set or changed to insert a link in the middle of a singly linked list?
5. How many references must be set or changed to insert a link at the end of a singly linked list?
6. In the `insert()` method in the `LinkedList.py` program, Listing 5-5, the statement `link = Link(datum, self.getNext())` means that
   a. the datum will be assigned to the `__data` field in the first `Link` of the `LinkedList`.
   b. the `__next` field of the new link will refer to the `LinkedList`'s first `Link`.
   c. `link` will be set to the first link whose data matches the given `datum`.
   d. a new, two-item linked list with the datum and the next link will be stored in `link`.

7. Assume you are writing a method of `LinkedList` and the variable `x` holds the next-to-last `Link`. What statement will delete the last link from the `LinkedList`?
8. When one function calls another function using a variable, as in `do_something(x)`, when can the value of `x` in the calling function be changed by the called function?
9. A double-ended list
   a. allows inserts to performed at either end in constant time.
   b. has its last link connected to its first link.
   c. is another name for a doubly linked list.
   d. has pointers running both forward and backward between links.
10. An abstract data type
    a. specifies the fields of the data type without defining any methods.
    b. allows references to data to be stored instead of the data itself.
    c. defines the kinds of data that are represented and the operations that can be performed on them.
    d. is used as a placeholder for a data collection inside a large program while other modules are written.
11. Consider storing an unordered collection of records in an array or in a linked list. Assuming copying a record takes longer than comparing keys to find a record, is it faster to delete a record with a certain key from a linked list or from an array?
12. How many times would you need to traverse an unordered doubly linked list to delete the item with the largest key?
13. Of the lists discussed in this chapter, which one would be best for implementing a queue?
14. Of the lists discussed in this chapter, which one would be best for implementing a priority queue?
15. Which of the following is **not** true? Iterators could be useful if you wanted to
    a. do an insertion sort on a linked list.
    b. delete all links with a certain key value.
    c. swap two items with the keys A and B in a list.
    d. insert a new link at the beginning of a list.
16. Which do you think would be a better choice to implement a stack: a singly linked list or an array?
17. Which do you think would be a better choice to implement a collection of objects where it must be fast to locate an object by a key: an ordered doubly linked list or an ordered array?

18. Circular lists

    a. can implement ordered lists more efficiently than doubly linked lists.

    b. can more efficiently represent loop-like orderings such as turns in a game than doubly linked lists.

    c. can make finding an item with a matching key less efficient than singly linked lists.

    d. simplify the representation of lists by eliminating the need for a special value indicating the end of a list.

19. What are the consequences of making the `setNext()` method of a `Link` object public, assuming that it performs only the assignment if the argument is a reference to a `Link` object or `None`?

20. Python generators

    a. are useful for creating iterators.

    b. can be used to make classes like circular lists from simpler classes.

    c. are the only way to represent infinite sequences like the Fibonacci sequence.

    d. require the use of the `yield` statement and cannot contain loops.

Experiments

Carrying out these experiments will help to provide insights into the topics covered in the chapter. No programming is involved.

5-A The `OrderedList` class shown in Listing 5-13 has a `find()` method that steps through the list in increasing order to find the goal. Would it be better to do a binary search instead of a linear one? Why or why not?

5-B Imagine that you gave a friend a gift last year, say a nice candle. This year a different friend gives you a nice candle as a gift. You wonder if the exact same candle, not just a similar one, has been returned to you. How would you go about determining whether it's the same one? How does that process relate to the concepts in this chapter?

5-C Imagine a gift exchange among a group of people. Each person will give a small gift to one other person in the group, and the receiver won't know who brought their gift. To set up this exchange, you put the names of all the people in a box, and everyone takes turns drawing a name from the box. If the name they draw is their own, they draw another name from the box and put theirs back in. The name is the person who will receive their gift. This selection process continues until everyone has drawn a name. These are sort of like the links in lists.

Here's how the exchange runs:

- One gift is given each day.
- Each person gives a gift the day after they receive their gift.
- The first person to draw a name gives their gift on day 1.

How long will the gift giving take if there were N uniquely named people? Can you think of any problems that could affect the answer? Can you think of ways to eliminate the problems?

5-D Iterators are great for stepping through data structures to find an item. Would you use one if you want to delete that item? Specifically, if you used one of the iterators shown in Listing 5-21 or Listing 5-23 to find the item to delete, what would be the next step? How efficient would that step be?

5-E Linked lists use memory more efficiently than expandable arrays, but by how much? Imagine an application that starts with an array of 10 elements. It inserts items into the array and doubles the size of the array every time an insertion would go past the end of the current array size. When it doubles the size of the array, it allocates a new segment of memory that is twice as long and copies all the contents of the first array into the beginning half of the second array. How much more memory is used by the expandable arrays if the number of items inserted is 1,000? How about for 1,000,000? Assume that the smaller arrays still take up space even when they are no longer used, and that storing 1 item takes as much memory as storing 1 reference.

Programming Projects

Writing programs to solve the Programming Projects helps to solidify your understanding of the material and demonstrates how the chapter's concepts are applied. (As noted in the Introduction, qualified instructors may obtain completed solutions to the Programming Projects on the publisher's website.)

5.1 Rewrite the `traverse()`, `__str__()`, and `__len__()` methods of the `LinkedList` class shown in Listing 5-4 to use the iterator (created by the generator) shown in Listing 5-23.

5.2 Implement a priority queue based on an ordered linked list. The items stored in the queue can be passed to a function that extracts their priority value (or key) as in the `PriorityQueue.py` module in Chapter 4. The remove operation on the priority queue should remove the item with the smallest key.

5.3 Implement a deque based on a doubly linked list. (See Programming Project 4.3 in the preceding chapter.) It should include `insertLeft()`, `insertRight()`, `removeLeft()`, `removeRight()`, `peekLeft()`, `peekRight()`, and `isEmpty()` methods.

5.4 Make a class for a singly linked circular list that has no end and no beginning like the one shown in Figure 5-14. The only access to the list is a single reference, `__last`, that can point to any link on the list. This reference can move around the list as needed. Your data structure should have methods to check if the list is empty and inspect the first item. It should provide methods for `insertFirst()`, `insertLast()`, `deleteFirst()`, and `search()` (but not `deleteLast()`), ensuring that the list always remains circular. You should provide a `__str__()` method to display the list from first to last (and you need to break the loop before repeating any items). A `step()` method that moves `__last` along to the next link and a `seek()` method that advances it to the next link that matches a particular goal key might come in handy too.

5.5 Implement stack and queue classes based on the circular list of Programming Project 5.4. The `Stack` class must have `push()`, `pop()`, and `peek()` methods. The `Queue` class must have `insert()`, `remove()`, and `peek()` methods. This exercise is not too difficult, although you should be careful about maintaining the LIFO and FIFO orders. (Implementing a deque can be harder, unless you make the circular list doubly linked or allow deleting from one end to be O(N).)

5.6 Implement another priority queue class as a two-level structure. The top-level list is a list of queues, one for each of the different priority values stored so far. The second level queues hold all the items of the same priority in a way that makes it easy to enforce the queue's first-in, first-out order. This time, the priority of an item is a separate argument to the `insert()` method, not something that is computed by a function on the item being inserted. This allows items to be reprioritized and placed at the end of their new priority queue. The `insert()` method should search the top-level list to find a queue with the matching priority or create a new queue if none of them match. The `remove()` method should take an optional argument, `priority`, which selects a priority queue from which to remove the next item. If `priority` is `None`, then `remove()` should find the highest priority queue that has at least one item. The class should have an iterator to step through all items in all the queues and a second iterator, `priorities()`, to iterate over the priority keys that have at least one item in their queue. Use the full iterator to get the item count for all priorities in the `__len__()` method of the class you define.

5.7 Create an iterator with both a `next()` and a `previous()` method to step through the Fibonacci sequence both forward and backward. Because the series is not infinite in both directions, the `previous()` method should raise `StopIteration` instead of returning 0 or a negative number. Calling `previous()` right after `next()` should repeat the same number in the sequence because the iterator has already passed that number. Similarly, calling `next()` right after `previous()` should produce a repeated number. Hint: You can't implement this project with a Python generator because it creates only a single directional iterator.

CHAPTER 6

Recursion

IN THIS CHAPTER

- Triangular Numbers
- Factorials
- Anagrams
- A Recursive Binary Search
- The Tower of Hanoi
- Sorting with mergesort
- Eliminating Recursion
- Some Interesting Recursive Applications

Recursion is a programming technique in which a function calls itself. This behavior may sound strange, or even catastrophic. Recursion is, however, one of the most interesting, and one of the most surprisingly effective, techniques in programming. It's been said,"The definition of insanity is doing the same thing over and over again, but expecting different results," so how could a function calling itself ever get a better result? It not only works but also provides a unique conceptual framework for solving many problems.

The concept of recursion appears in the natural world and in art, such as the label used on the early twentieth century Droste Cacao tin shown in Figure 6-1. The image of the tin appears on the tray carried by the nurse. Presumably, that smaller image would contain the entire image of the tin, and that tin's label would contain the entire image of the tin, and so on. This illustration was a popular example of recursion in art, leading to the name *Droste effect*. Mirrors placed facing each other and aligned to be parallel produce infinite recursive images. Many patterns in nature repeat at smaller scales within their structures.

In this chapter we examine numerous examples to show the wide variety of situations to which recursion can be applied in programming. We show you how to calculate triangular numbers and factorials, generate anagrams, perform a recursive binary search, solve the Tower of Hanoi puzzle, and investigate a sorting technique called mergesort. Visualization tools are provided to demonstrate the Tower of Hanoi and mergesort.

We also discuss the strengths and weaknesses of recursion and show how recursive approaches can be transformed into stack-based approaches.

FIGURE 6-1 The Droste Effect—recursion in art

Triangular Numbers

It's said that the Pythagoreans, a band of mathematicians in ancient Greece who worked under Pythagoras (of Pythagorean theorem fame), felt a mystical connection with the series of numbers 1, 3, 6, 10, 15, 21, 28, … (where the … means the series continues indefinitely). Can you find the next member of this series?

The *n*th term in the series is obtained by adding *n* to the previous term. For example, the second term is found by adding 2 to the first term (which is 1), giving 3. The third term is 3 added to the second term (which is 3), giving 6, and so on. You can also consider the 0^{th} term to be 0, because adding 1 to it gives the first term. The numbers in this series are called **triangular numbers** because they can be visualized as a triangular arrangement

of objects, shown as blue squares in Figure 6-2. The length of the shorter (equilateral) sides is *n*, the term number.

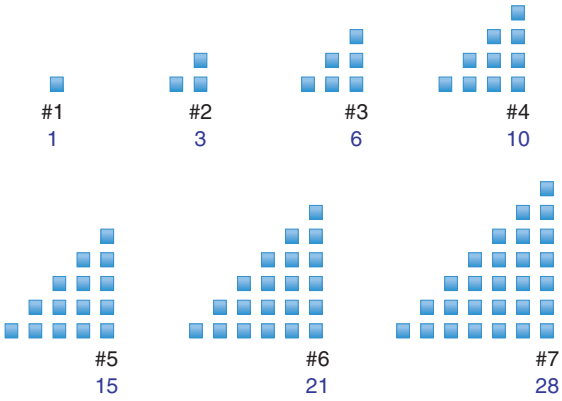


FIGURE 6-2 The triangular numbers

Finding the *n*th Term Using a Loop

Suppose you wanted to find the value of some arbitrary *n*th term in the series—say the fourth term (whose value is 10). How would you calculate it? Looking at Figure 6-3, you might decide to determine the value of any term by adding up the number of squares in the vertical columns.

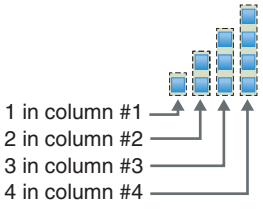


FIGURE 6-3 Triangular number as columns

There are four squares in the fourth column. The value of the fourth term is 4 plus the value of the third term. The third term adds the three squares in its column to the second term, and so on down to column #1 with one square. Adding 4+3+2+1 gives 10.

The following `triangular_loop()` function uses this column-based technique to find the *n*th triangular number. It sums all the columns, from a height of *n* to a height of 1:

```
def triangular_loop(nth):       # Get the nth triangular number using a loop
   total = 0                    # Keep a total of all the columns
   for n in range(nth, 0, -1): # Start at nth and go back to 1
      total += n                # add n (column height) to total
   return total                 # Return the total of all the columns
```

Of course, the values could be added up in any order such as 1+2+3+4, which would change the range iterator to `range(1, nth + 1)`.

Finding the *n*th Term Using Recursion

The loop approach is straightforward, but there's another way to look at this problem. The value of the *n*th term can be thought of as the sum of only two things, instead of a whole series. They are

1. The *n*th (tallest) column, which has the value *n*
2. The sum of all the remaining columns

This concept is shown in Figure 6-4.

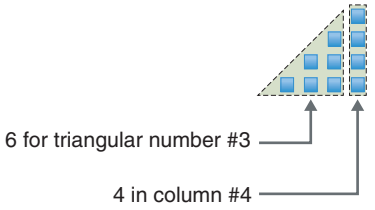


FIGURE 6-4 Triangular number as column plus triangle

Finding the Remaining Columns

If we knew a method that found the sum of all the preceding columns, we could write a `triangular()` method, which returns the value of the *n*th triangular number, like this:

```
def triangular(nth):        # Get the nth triangular number (incomplete)
   return (nth +            # Add this column to the preceding
           sum_before(nth)) # column sum
```

But what have we gained here? It looks like writing the `sum_before()` method is just as hard as writing the `triangular()` method in the first place.

Notice in Figure 6-4, however, that the sum of all the columns preceding term *n* is the same as the sum of *all* the columns for term *n*–1. Thus, if we knew about a method that summed all the columns up to term *n*, we could call it with an argument of *n*–1 to find the sum of all the remaining columns for term *n*:

```
def triangular(nth):           # Get the nth triangular number (incomplete)
   return (nth +               # Add this column to the preceding
           sum_up_to(nth - 1)) # column sum
```

When you think about it, the `sum_up_to()` method does exactly the same thing that the `triangular()` method does: sum all the columns for some number *n* passed as an

argument. So why not use the `triangular()` method itself, instead of some other method? That would look like this:

```
def triangular(nth):            # Get the nth triangular number (incomplete)
   return (nth +                # Otherwise add this column to the preceding
           triangular(nth - 1)) # triangular number
```

The fact that a function can call itself might seem odd, but why shouldn't it be able to? A function call is, among other things, a transfer of control to the start of the function. This transfer of control can take place from within the function as well as from outside. The same is true, of course, for object methods.

Passing the Buck

All these approaches may seem like passing the buck. Someone asks you to find the 9th triangular number. You know this is 9 plus the 8th triangular number, so you call Harry and ask him to find the 8th triangular number. When you hear back from him, you can add 9 to whatever he tells you, and that will be the answer.

Harry knows the 8th triangular number is 8 plus the 7th triangular number, so he calls Sally and asks her to find the 7th triangular number. This process continues with each person passing the buck to another one.

Where does this buck-passing end? Someone at some point must be able to figure out an answer that doesn't involve asking another person to help. If this didn't happen, there would be an infinite chain of people asking other people questions—a sort of arithmetic Ponzi scheme that would never end. In the case of `triangular()`, this would mean the function calls itself over and over in an infinite series that would eventually crash the program.

The Buck Stops Here

To prevent an infinite regress, the person who is asked to find the first triangular number of the series, when *n* is 1 (or 0), must know, without asking anyone else, that the answer is 1 (or 0). There are no smaller numbers to ask anyone about, there's nothing left to add to anything else, so the "buck" stops there. In fact, we certainly don't want anyone to make a mistake and ask about negative numbers, which might throw off the total count. We can express this by adding a condition to the `triangular()` method, as shown in Listing 6-1.

LISTING 6-1 The Recursive `triangular()` Function

```
def triangular(nth):            # Get the nth triangular number recursively
   if nth < 1: return 0         # For anything less than 1, it's 0
   return (nth +                # Otherwise add this column to the preceding
           triangular(nth - 1)) # triangular number
```

The condition that leads to a recursive method returning without making another recursive call is referred to as the **base case**. It's critical that every recursive method have a base case to prevent infinite recursion and the consequent demise of the program.

Using the recursive definition described earlier, you can ask for large triangular numbers like this:

```
>>> triangular(100)
5050
>>> triangular(200)
20100
>>> triangular(500)
125250
```

What's Really Happening?

Let's modify the `triangular()` function to provide an insight into what's happening when it executes. We'll insert some print statements to keep track of the arguments and return values:

```
def show_triangular(nth): # Print the recursive execution steps of
   print('Computing triangular number #', nth) # computing the nth
   if nth < 1:            # triangular number. Base case
      print('Base case. Returning 0') # Print the return information
      return 0
   value = nth + show_triangular(nth - 1) # Non-base case, get value
   print('Returning', value, 'for #', nth) # Print the return info
   return value
```

The print statements show the entry and exit information for each call to `show_triangular()`. Here's the result of calling `show_triangular(5)`:

```
>>> show_triangular(5)
Computing triangular number # 5
Computing triangular number # 4
Computing triangular number # 3
Computing triangular number # 2
Computing triangular number # 1
Computing triangular number # 0
Base case. Returning 0
Returning 1 for # 1
Returning 3 for # 2
Returning 6 for # 3
Returning 10 for # 4
Returning 15 for # 5
15
```

Each time the `show_triangular()` function calls itself, its argument is reduced by 1. The function plunges down calling itself again and again until its argument is reduced to 0. Then it returns. This process triggers an entire series of returns. The control passes back up

out of all the versions that were waiting for a value. Each time the function returns, the caller adds its value of `nth` to the return value of the function it called.

The return values reconstruct the series of triangular numbers, until the answer is returned to the Python interpreter. Figure 6-5 shows how each invocation of the `triangular()` function can be imagined as being "inside" the previous one.

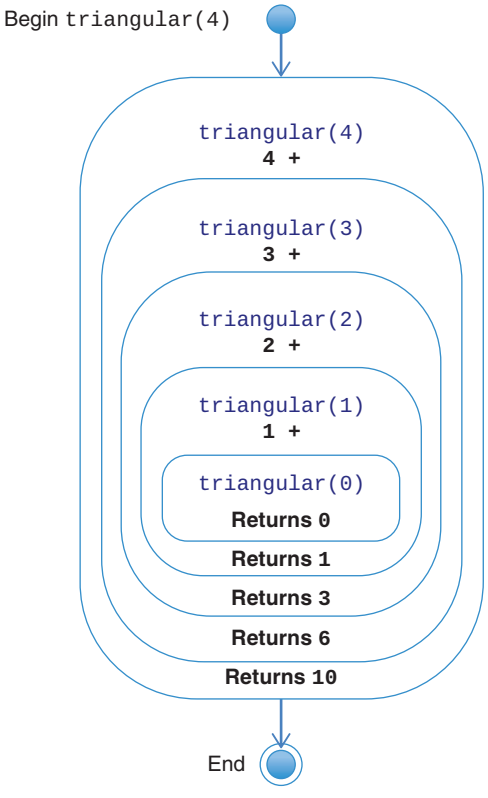


FIGURE 6-5 Execution of the recursive `triangular(4)` function

Notice that, just before the innermost version returns a 0, there are actually five different incarnations of `triangular()` in existence at the same time. The outer one was passed the argument 4; the inner one was passed the argument 0. The top and bottom edges of each invocation correspond to the print statements in `show_triangular()`.

Characteristics of Recursive Routines

Although it's short, the `triangular()` function possesses the key features common to all recursive routines:

- It calls itself.

- When it calls itself, it does so to solve a smaller version of the same problem, also called a subproblem.
- There's some version of the problem that is simple enough that the routine can solve it, and return, without calling itself.

In each successive call of a recursive method to itself, the argument becomes smaller (or perhaps a range described by multiple arguments becomes smaller), reflecting the fact that the problem has become "smaller" or easier. Here we explore how data structures holding fewer data elements can be the smaller problem. When the argument or range reaches a certain minimum size, a condition triggers and the method returns without calling itself.

Is Recursion Efficient?

Calling a function or method involves certain overhead. Control must be transferred from the location of the call to the beginning of the function. In addition, the arguments to the function and the address to which the function should return must be pushed onto an internal stack so that the function can access the argument values and know where to return.

In the case of the `triangular()` function, it's probable that, as a result of this overhead, the `while` loop approach executes more quickly than the recursive approach. The penalty may not be significant, but if there are a large number of function calls as a result of a recursive function, it might be desirable to eliminate the recursion. We discuss this issue more at the end of this chapter.

Another inefficiency is that memory is used to store all the intermediate arguments and return values on the system's internal stack. This behavior may cause problems if there is a large amount of data, leading to a stack overflow.

These efficiency concerns are especially relevant to triangular numbers and other numeric computations. There are usually mathematical simplifications that can avoid loops or recursion. As you might have guessed, the number of little squares in the triangle is about equal to the area of the triangle, which is half the area of the square formed by putting two of the triangles together, sharing their longest edge. For a square of width n, its area is n^2, so the area of the triangle would be $n^2/2$. The exact formula for the sum of all the integers from 0 to a positive integer, n, is

$$\frac{n(n+1)}{2}$$

This formula shows up again and again in the study of data structures. The most efficient way to calculate it would be to implement the computation of this formula because the addition, multiplication, and division are all O(1) operations.

Recursion is usually used because it simplifies a problem *conceptually*, not because it's inherently more efficient.

Mathematical Induction

For those programmers with an interest in math, it's important to see the relationship between recursion and induction. **Mathematical induction** is a way of defining something in terms of itself. (The term is also used to describe a related approach to proving theorems, which is also a technique for proving the correctness of programs.) Using induction, you could define the triangular numbers mathematically by saying

$$\text{tri}(n) = 0 \qquad \text{if } n <= 0$$

$$\text{tri}(n) = n + \text{tri}(n-1) \quad \text{if } n >= 1$$

Defining something in terms of itself may seem circular, but in fact it's perfectly valid (provided there's a base case).

Factorials

Factorials are similar in concept to triangular numbers, except that multiplication is used instead of addition. The triangular number corresponding to *n* is found by adding *n* to the triangular number of *n*–1, while the factorial of *n* is found by multiplying *n* by the factorial of *n*–1. In other words, the fifth triangular number is 5+4+3+2+1, while the factorial of 5 is 5×4×3×2×1, which equals 120.

Factorials are useful for finding the number of possible orderings or combinations of things. Think of a race between three people. How many different ways can the race end assuming no ties occur? The first person can finish in first, second, or third position. The second person can finish in any position other than the first, so two positions. The third person takes the final position. So, there are 3×2×1, which equals 6, orderings. That same logic extends to higher numbers. Table 6-1 shows the factorials of the first 10 numbers.

Table 6-1 Table Factorials

Number	Calculation	Factorial
0	by definition	1
1	1 × 1	1
2	2 × 1	2
3	3 × 2	6
4	4 × 6	24
5	5 × 24	120
6	6 × 120	720
7	7 × 720	5,040
8	8 × 5,040	40,320
9	9 × 40,320	362,880

The factorial of 0 is defined to be 1. If you think about the example of the orderings of a group of racers, there is exactly 1 ordering of an empty group of racers. Factorial numbers grow large very rapidly, as you can see. In math, the factorial of 5 is written with an exclamation mark, 5!, perhaps as a way to remind us how big they are.

A recursive method similar to `triangular()` can be used to calculate factorials. It looks like this:

```
def factorial(n):            # Get factorial of n
   if n < 1: return 1        # It's 1 for anything < 1
   return (n *               # Otherwise, multiply n
           factorial(n - 1)) # by preceding factorial
```

There are only a few differences between `factorial()` and `triangular()`. First, we changed the variable to `n` from `nth` because factorial is treated as a function rather than a sequence. Second, `factorial()` uses a * instead of a + in the expression

```
n * factorial(n - 1)
```

Third, the base case returns 1, not 0. It's possible to eliminate one or more recursive calls by adding more base cases for $n = 1$, $n = 2$, and so on, but doing so only saves a little work. Again, the purpose of recursion is its conceptual simplicity.

Here are some samples of what the function produces. Figure 6-6 shows how the recursion creates a similar set of invocations of the function to produce the last value.

```
>>> factorial(9)
362880
>>> factorial(40)
815915283247897734345611269596115894272000000000
 >>> factorial(4)
24
```

Various other numeric entities lend themselves to calculation using recursion in a similar way, such as finding the greatest common denominator of two numbers (which is used to reduce a fraction to lowest terms), raising a number to a power, and so on. The factorial function is somewhat more interesting because it can't be easily simplified into a formula using other math primitives that can be computed in constant time. Even though this calculation is interesting for demonstrating recursion, it probably wouldn't be used in practice because a loop-based approach is more efficient. Recursion can be used to prototype an algorithm, which is later converted to a more efficient implementation.

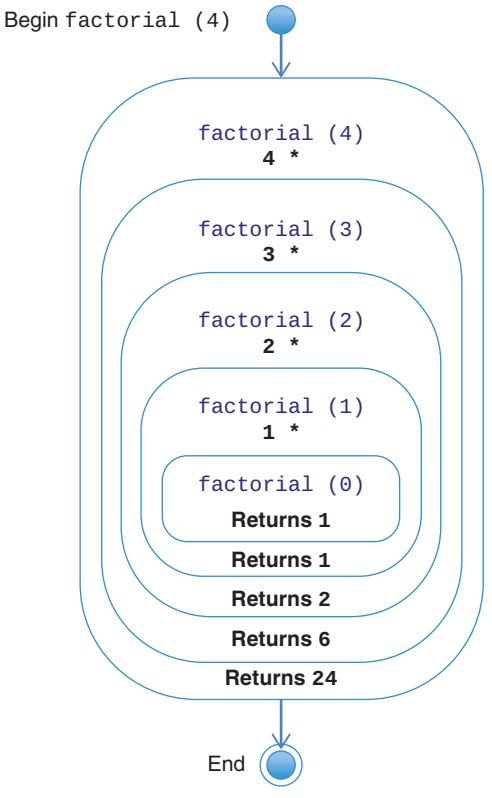


FIGURE 6-6 Execution of the recursive `factorial(4)` function

Anagrams

Here's a different kind of situation in which recursion provides a neat solution to a problem. A **permutation** is an arrangement of things in a definite order (sometimes called an *ordering*). Suppose you want to list all the **anagrams** of a specified word—that is, all possible permutations that can be made from the letters of the original word (whether they make a real word or not). We call this *anagramming* a word. Anagramming `cat`, for example, would produce

```
cat
cta
atc
act
tca
tac
```

Try anagramming some words yourself. Because this process is analogous to ordering of the racers, the number of possibilities is the factorial of the number of letters. For 3 letters, there are 6 possible words (permutations); for 4 letters there are 24 words; for 5 letters, 120; and so on. These numbers assume that all letters are distinct; if there are multiple instances of the same letter, there will be fewer distinct words.

How would you write a program to find all the anagrams of a word? Think about it with this new technique, recursion. Here's where the simplifying concept helps. Assume you already have a function for anagramming a string of *n* letters. Now all you need is a way to handle adding another letter. If there are *n*+1 letters in the original word, where can that extra letter go in each of the short anagrams of length *n*? Well, it could go in between any of the other letters or at the beginning or at the end. Is that all the anagrams with *n*+1 letters?

Think about the example with `cat`. The two anagrams of `at` are `at` and `ta`. If you place `c` between the two characters, you get `act` and `tca`. If you add `c` to the beginning, you get `cat` and `cta`. Adding `c` to the end produces `atc` and `tac`. Altogether you have

```
act, tca, cat, cta, atc, tac
```

You have found all six orderings and haven't introduced any duplicates (as long as there were no duplicates in the list of anagrams at the start). So, this looks correct for producing all the possible anagrams when adding a single character to a set of distinct anagrams. All that's left to do is define the base case (or cases).

If a string has one character in it, then there's only one ordering, the string itself. What about an empty string? Well, you should still return the sole ordering of that too, the empty string itself. That corresponds to `factorial(0)` returning 1 as discussed previously. So, you can set up a recursive `anagrams()` function as follows:

```
def anagrams(word):                  # Return a list of anagrams for word
   if len(word) <= 1:                # Empty words and single letters
      return [word]                  # have a single anagram, themselves
   result = []                       # Start with an empty list
   for part in anagrams(word[1:]):   # Loop over smaller anagrams
      for i in range(len(part) + 1): # For each index in smaller word
         result.append(              # Add a new anagram with
            part[:i] +               # the smaller word up to the index
            word[0] +                # plus the 1st character of this word
            part[i:])                # plus the rest of the smaller word
   return result                     # Return the list of bigger anagrams
```

The function starts with the base case. Short words of 0 or 1 character are the only possible anagram, so the function immediately returns a list of the `word` itself. If the word is longer than 1 character, it gets all the anagrams of the last part of `word` (all but the first character) by using the Python slice operator (`word[1:]`). It then loops over each one of

those shorter anagrams, storing it in the `part` variable. Before entering the loop, it initializes a `result` list to be the empty list.

Within each `part`, there are `n`, or `len(part)`, characters. The first character of the `word` to be anagrammed can go between any of them or at the beginning or the end. That's the same as inserting the first character before any character of `part` plus inserting it at position `n`. The function uses another loop to set `i` to each of those insertion indices, 0 up to `n`, inclusive. The body of the inner loop appends a new `n+1` character anagram to the `result` list. (In this case, Python's `list` type is being treated as a linked list rather than an array.) The longer anagram starts with the (possibly empty) slice of the shorter anagram, `part`, up to `i`, then the first letter of the `word`, and then the (possibly empty) remaining slice of the shorter anagram starting at `i`.

Testing this implementation, you get

```
>>> anagrams('')
['']
>>> anagrams('c')
['c']
>>> anagrams('cat')
['cat', 'act', 'atc', 'cta', 'tca', 'tac']
```

The invocations of the anagram function on `"cat"` are shown in Figure 6-7. The innermost invocation is the base case, which returns the list of one string, the last letter, `"t"`. The invocation surrounding that innermost one inserts `"a"` into the `"t"` string at the two possible locations, the beginning and the end, to produce `["at", "ta"]`. The outermost invocation inserts `"c"` into each of the two strings at one of three possible locations. The combination of two strings and three possible locations produces six anagrams.

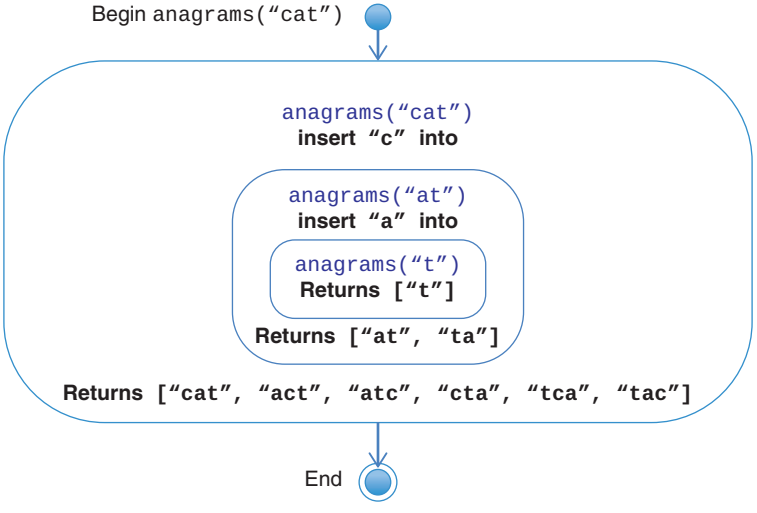


FIGURE 6-7 Anagramming a word

Anagramming short words is fine. For example:

```
>>> anagrams('tire')
['tire', 'itre', 'irte', 'iret', 'trie', 'rtie', 'rite', 'riet', 'trei', 'rtei',
'reti', 'reit', 'tier', 'iter', 'ietr', 'iert', 'teir', 'etir', 'eitr', 'eirt',
'teri', 'etri', 'erti', 'erit']
```

Anagramming longer words, however, is likely to become more of a nuisance. The factorial of 6 is 720, and generating such long sequences may produce more words than you want to know about.

A Recursive Binary Search

Remember the binary search we discussed in Chapter 2, "Arrays"? The search finds the index to a cell with a matching key in an ordered array using the fewest number of comparisons. The solution kept dividing the array in half, seeing which half contained the desired cell, dividing that half in half again, and so on. Here's the `OrderedRecordArray.find()` method:

```
   def find(self, key):             # Find index at or just below key
      lo = 0                        # in ordered list
      hi = self.__nItems-1          # Look between lo and hi
      while lo <= hi:
         mid = (lo + hi) // 2       # Select the midpoint

         if self.__key(self.__a[mid]) == key:  # Did we find it?
            return mid              # Return location of item

         elif self.__key(self.__a[mid]) < key: # Is key in upper half?
            lo = mid + 1            # Yes, raise the lo boundary

         else:
            hi = mid - 1            # No, but could be in lower half

      return lo   # Item not found, return insertion point instead
```

You might want to reread the section on binary searches in ordered arrays in Chapter 2, which describes how this method works, or review the algorithm visualization.

You can transform this loop-based method into a recursive method quite easily. The loop-based method changes either `lo` or `hi` to specify a new range on each iteration. Each time through the loop it divides the range (roughly) in half.

Recursion Replaces the Loop

In the recursive approach, instead of changing `lo` or `hi` inside a loop, you call `find()` again with the new values of `lo` or `hi` as arguments. The loop disappears and is replaced by the recursive calls. Here's how that looks:

```
class OrderedRecordArray(object):
… # other definitions as shown in Chapter 2 …
   def find(self, key,              # Find index at or just below key
            lo = 0,                 # in ordered list between lo
            hi = None):             # and hi using recursion
      if hi is None:                # If hi was not provided,
         hi = self.__nItems - 1     # use upper bound of array
      if lo > hi:                   # If range is empty,
         return lo                  # return lo for base case
      mid = (lo + hi) // 2          # Select the midpoint
      if self.__key(self.__a[mid]) == key:  # Did we find it?
         return mid                 # Return location of item

      if self.__key(self.__a[mid]) < key: # Is key in upper half?
         return self.find(          # then recursively search
            key, mid + 1, hi)       # in upper half
      else:                         # Otherwise, it must be in
         return self.find(          # lower half so recursively
            key, lo, mid - 1)       # search below mid
```

This recursive version defines default values for the `lo` and `hi` parameters to `find()`. That allows callers to call the method without specifying the range of indices to search. The first `if` statement fills in the value of `hi` as the last valid index if the caller did not provide a value (or passed `None` as an argument). The default value for `lo` is a constant, 0, so it can be specified in the parameter definition.

Like the other recursive methods, the next test is for the base case. The second `if` statement looks for an empty range of indices where `lo > hi`. That will happen for empty arrays or when the key is not found among the items in the array. In this case, `find()` returns `lo` because an item with that key would be inserted at `lo` to maintain the ordering. Why not check if `lo == hi` as the base case? If you did so, then you would still need to compare the key being sought with the key of the item at `lo`. Depending on their relationship, the value to return could be `lo`, `lo - 1`, or `lo + 1`. It's easier to skip those checks and let the recursion proceed until `lo > hi`. We examine each of those cases in the rest of the routine.

The next statement sets `mid` to be the midpoint of the range between `lo` and `hi`. The third `if` statement compares the key at that index of the array with the `key` being sought. If they are equal, the method can return `mid` as the result; no more recursion is needed. If there were previous recursive calls to get to this point, they all will pass the `mid` value as the result back to their callers.

After the third `if` statement compares the search `key` with the one at `mid` and fails to find a match, the only thing left to try is searching the upper or lower search ranges. By checking whether the search `key` is above or below the key at `mid`, the method can determine the proper range and search it using the recursive call to `self.find()`. These two calls handle the other two conditions related to `lo == hi == mid`. The recursive call goes to `lo + 1` if the key at `mid` is less than the search `key`. Otherwise, it searches the `lo` to `mid - 1` range, which is empty because `lo == mid`, and returns `lo`.

Here's a small test of the recursive `find()` method. In this example, the program places a small set of integers in an array and tries finding some that are present and some that are not.

```
arr = OrderedRecordArray(10)
for item in [3, 27, 14, 10, 88, 41, 67, 51, 95]:
   arr.insert(item)

print("Array containing", len(arr), "items:\n", arr)

for goal in [0, 10, 11, 99]:
   print("find(", goal, ") returns", arr.find(goal))
```

The output from this test is

```
Array containing 9 items:
 [3, 10, 14, 27, 41, 51, 67, 88, 95]
find( 0 ) returns 0
find( 10 ) returns 1
find( 11 ) returns 2
find( 99 ) returns 9
```

Searching for 0 returns 0 because the value would go in the first cell of the array if inserted, so it would be in front of 3. The call to `find(10)` returns 1, which is the index of the cell holding 10. The next call to `find(11)` shows it returning an index one before the next larger integer in the array, and the last one is beyond the current end of the array because it is bigger than all the items.

Figure 6-8 shows the recursive calls needed to find the index for 11 in the preceding array. In the initial call, `lo` and `hi` are not specified, so their defaults (shown in green inside the parentheses) are used. The internal recursive calls don't use the defaults.

The range of indices to search is initially all 9. In the first recursive call, the range shrinks to 4, indices 0 through 3. The second recursive call reduces the range to 2, indices 2 through 3. The last recursive call reduces the range to 0 after finding that the key at index 2, 14, is higher than the goal key of 11. Because the range is empty, the base case causes 2 to be returned up the chain of recursive calls.

The recursive binary search has the same Big O efficiency as the nonrecursive version: O(log N). It is more elegant but may be slightly slower because function calls and returns can take more time than updating loop variables.

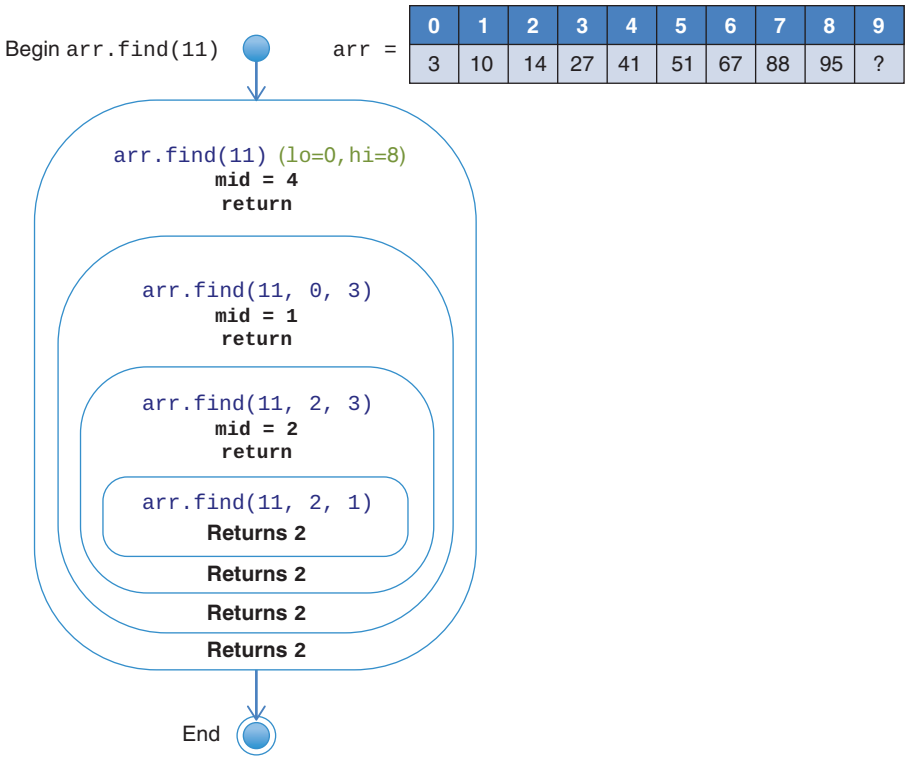


FIGURE 6-8 Performing recursive binary search

Divide-and-Conquer Algorithms

The recursive binary search is an example of the *divide-and-conquer* approach. You divide the big problem into smaller problems and solve each one separately. The solution to each smaller problem is the same: you divide it into even smaller problems and solve them. The process continues until you get to the base case, which can be solved easily, with no further division.

The divide-and-conquer approach is commonly used with recursion, although, as you saw in the binary search in Chapter 2, you can also use a nonrecursive approach. It usually involves a method that contains two recursive calls to itself, one for each half of the problem. In the binary search, there are two such calls, but only one of them is actually executed. (Which one depends on the value of the keys.) The mergesort, which is described later in this chapter, actually executes both recursive calls (to sort two halves of an array).

The Tower of Hanoi

The Tower of Hanoi is a historic puzzle consisting of several disks placed on three spindles or columns, as shown in Figure 6-9. The puzzle is also known as the Tower of Brahma or Lucas's Tower for Édouard Lucas, its inventor. Lucas was a mathematician who studied the

Fibonacci sequence and other recursively defined sequences. The goal of the puzzle is to move all of the disks from one spindle to another, following a set of specific rules.

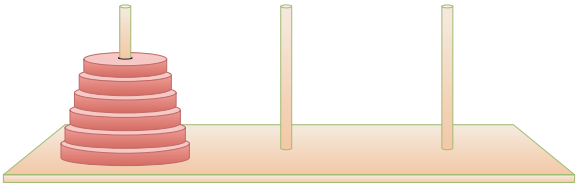


FIGURE 6-9 The Tower of Hanoi puzzle

The disks all have different diameters. Each one has a hole in the middle that fits over the spindles. All the disks start out on one spindle, stacked in order of diameter, which gives the appearance of a tower. The object of the puzzle is to transfer all the disks from the starting spindle, say the one on the left, to another spindle, say the one on the right. Only one disk can be moved at a time, and no disk may be placed on a disk that's smaller than itself.

The legend that goes along with the puzzle is that in a distant temple, a group of monks labor day and night to transfer 64 golden disks from one of three diamond-studded towers to another. When they are finished, the world will end. If that alarms you, wait until you see how long it takes to solve the puzzle with 64 disks.

The TowerofHanoi Visualization Tool

Start up the TowerOfHanoi Visualization tool. Create a new three-disk puzzle by typing `3` in the text entry box and selecting New. You can attempt to solve the puzzle yourself by using the mouse to drag the topmost disk to another tower. The star by the spindle on the right indicates the goal tower. Figure 6-10 shows how the towers look after several moves have been made.

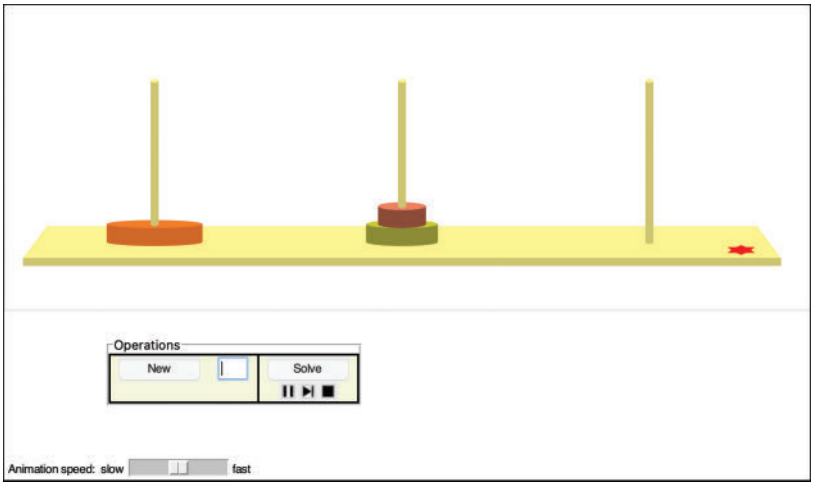


FIGURE 6-10 The TowerOfHanoi Visualization tool

There are two ways to use the visualization tool:

- You can attempt to solve the puzzle manually, by dragging the disks from tower to tower.
- You can select the Solve button and watch the algorithm solve the puzzle with no intervention on your part. The disks zip back and forth between the posts. Using the Pause/Play button, you can stop and resume the animation to examine what happens at each step.

To restart the puzzle, type in the number of disks you want to use, from 1 to 6, and select New. The specified number of disks will be arranged on the left spindle. You can drag the top disk to either of the other two spindles. If you pick up the next larger disk, the tool will allow you to place it only on a spindle whose topmost disk has a larger diameter. If you release it away from a spindle that can accept it, the disk returns to where you picked it up. When all the disks form a tower on the leftmost spindle, the Solve button is enabled. If you stop the automated solver, you can resume manually solving the puzzle.

Try solving the puzzle manually with a small number of disks, say three or four. Work up to higher numbers. The tool gives you the opportunity to learn intuitively how the problem is solved.

Moving Pyramids

Let's call the arrangement of disks on a spindle a "pyramid" because they are ordered by diameter. You could call them a "stack," but that would be somewhat confusing with the stack data structure you've studied (although that data structure will be useful to model the disks). Using this terminology, the goal of the puzzle is to move the pyramid from the left spindle to the right. For convenience, the spindles can be labeled L, M, and R for left, middle, and right.

How do you solve the puzzle? Let's start with the easy cases. If there's only one disk, the solution is trivial. Move the one disk from spindle L to spindle R. In fact, even easier, if there are no disks, then there is nothing to move and the puzzle is solved. Then what about two disks? That's pretty easy, too. Move disk 1 from L to M, move disk 2 from L to R, and then move disk 1 from M to R. These moves are shown in the four panels of Figure 6-11.

How do you solve problems with more disks? You've seen the base cases; can recursion help with this? You might be surprised to know that you've already figured out all the steps of the recursive algorithm by enumerating the cases up to two disks.

To see how to get to a recursive solution, remember that you need to make smaller versions of the same problem. The two strategies you've seen are dividing the problem in half or reducing the number of things by one. Dividing it in half doesn't seem to fit, so let's look at reducing the number of disks by one. In particular, let's look at the case of three disks.

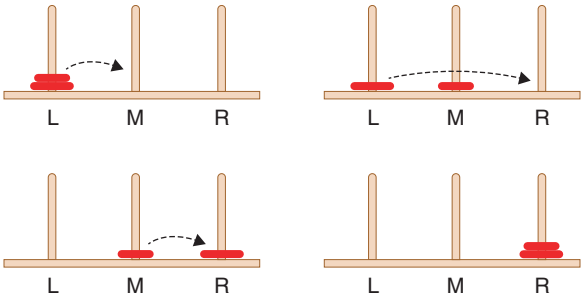


FIGURE 6-11 Solution to the Tower of Hanoi with two disks

If you perform the same steps that you took to solve the two-disk puzzle, then the three-disk puzzle would end up with disk 3 still on the left spindle and disks 1 and 2 on the right spindle, as shown in Figure 6-12. From here, it's easy to move disk 3 onto the middle spindle, but not easy to move it to the right-hand one.

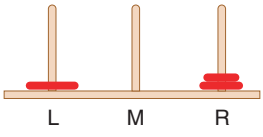


FIGURE 6-12 A Tower of Hanoi with three disks after moving two of them

If you changed the first moves to swap the middle and right spindles, then disks 1 and 2 would end up on the middle spindle and leave the right one open, as shown in Figure 6-13.

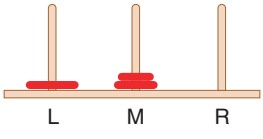


FIGURE 6-13 A Tower of Hanoi after moving two disks to the middle

With the right spindle empty, you can easily move disk 3 from the left to the right. Now all that's left is moving the two disks on the middle spindle over to the right. And that's a problem you've already solved. This example provides the basic outline for the solution of the three-disk problem, namely:

1. Move the top two disks to the middle spindle (using the right one as the spare).
2. Move disk 3 to the right spindle.
3. Move the top two disks on the middle spindle to the right spindle (using the left one as the spare).

Steps 1 and 3 are solutions that have the same form as the three moves shown in Figure 6-11. They are solutions to the two-disk problem, but with different starting and ending positions. Now it's time to recognize that you've solved the three-disk problem by reducing it to two calls to the two-disk solution plus one disk move in between. If you rephrased the preceding outline to solve an *N* disk problem, it would be

1. Solve the *N*–1 disk problem by moving the *N*–1 disk pyramid from the start spindle to the nongoal spindle (using the goal spindle as a spare).
2. Move the *N*th disk to the goal spindle.
3. Solve the *N*–1 disk problem by moving the *N*–1 disk pyramid from step 1 on to the goal spindle (using the starting spindle as a spare).

Steps 1 and 3 look like recursive calls to the same solution routine. That solution routine would need to know how many disks to move and the role of each of the three spindles: start spindle, goal spindle, and spare spindle. In the various recursive calls, the roles of the spindles change. If you think about the difference between Figure 6-12 and Figure 6-13, they perform the same solution but with the swapped roles of the right two spindles.

Is that it? Have you figured out all the steps? What about all the other, bigger disks? Wouldn't they prevent the solution from working because they were in the way of one of the steps?

If you think about it, it doesn't matter that there are more disks in the puzzle because they are all larger than the topmost disks. The outlined solution applies only to the top *N* disks, and disks *N*+1, *N*+2, and so on all must be larger. For example, applying the two-disk solution to move from the left to right spindles works equally well in all the situations shown in Figure 6-14, which has five disks. You count on that fact in the outlined solution. In fact, you hope the situation looks like the rightmost panel of Figure 6-14 at some point during the $N = 5$ puzzle. If all five disks started on the left spindle, performing all the outline steps for $N = 5$ and $N = 4$ and then steps 1 and 2 when $N = 3$ should leave disk 3 on the right (goal) spindle with disks 4 and 5 underneath it. All that remains is performing step 3 on the remaining *N*–1 (two) disks.

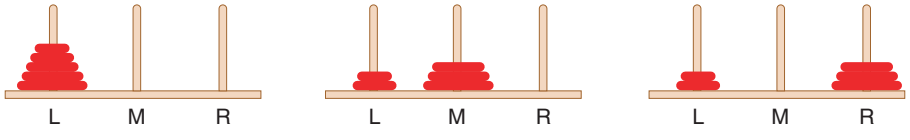


FIGURE 6-14 Possible states in a five-disk Tower of Hanoi

It still seems as though the algorithm might ask to move a larger disk on top of a smaller one after all the swapping of roles of the spindles. How can you tell that it won't try to move a disk from the right spindle to the left in the rightmost condition shown in Figure 6-14? There are two ways: write a program to test it and use mathematical induction to prove it. We tackle that program next.

The Recursive Implementation

Before diving into the code, let's look at what base cases are needed. As we mentioned, if there are no disks, no moves are needed, and the solution is done. That's definitely a base case. Do you need a base case for a single disk? You can check by seeing what happens if you apply the outline recursive solution to the case where *N* is 1. Step 1 is to solve the movement of the *N*–1 disk problem. Because *N*–1 is 0, you know that nothing will be moved. That's followed by moving the *N*th disk to the goal spindle, which means the one disk is moved to the final position. Then step 3 also does nothing (sometimes called a "no-op"), because *N*–1 is 0. Hence, there doesn't seem to be a need to write anything special for the case where *N* is 1. You only need the base case for 0 and the recursive outline.

Listing 6-2 shows the first part of a class, `TowerOfHanoi`, that solves the puzzle for any number of disks. Each instance of the puzzle will have a specific number of disks that is provided to the constructor. The puzzle needs to keep track of what disks are on what spindles. For that, you could use the `SimpleStack` class that was implemented in Chapter 4, "Stacks and Queues." A stack is perfect for modeling each spindle because the only allowed disk movements involve the top of the stack/spindle. You create a stack for each of the three spindles and push integers on it to represent the diameters of the disks. The diameters can be the numbers from 1 up to *N*, the number of disks. The bigger numbers are the larger diameter disks.

LISTING 6-2 The `TowerOfHanoi.py` Module—Puzzle Object

```
from SimpleStack import *

class TowerOfHanoi(object):      # Model the tower on 3 spindles using
                                 # 3 stacks
   def __init__(self, nDisks=3): # Constructor w/ starting number of
      self.__stacks = [None] * 3 # Stacks of disks
      self.__labels = ['L', 'M', 'R'] # Labels for stacks/spindles
      self.__nDisks = nDisks     # Total number of disks
      self.reset()

   def reset(self):              # Initialize state of puzzle
      for spindle in range(3):   # Set up each of 3 spindles
         self.__stacks[spindle] = Stack( # Start w/ empty stack
            self.__nDisks)       # that can hold all the disks
         if spindle == 0:        # On the first spindle,
            for disk in range(   # push the disks on the stack
                  self.__nDisks, 0, -1): # in descending order of size
               self.__stacks[spindle].push(disk)

   def label(self, spindle):     # Get the label of spindle
      return self.__labels[spindle]
```

```
   def height(self, spindle):    # Get the number of disks on a spindle
      return len(self.__stacks[spindle])

   def topDisk(self, spindle):   # Get top disk number on a spindle or
      if not self.__stacks[spindle].isEmpty(): # None if no disks
         return self.__stacks[spindle].peek() # Peek at top disk

   def __str__(self):            # Show puzzle state as a string
      result = ""                # Start with empty string
      for spindle in range(3):   # Loop over spindles
         if len(result) > 0:     # After first spindle,
            result += "\n"       # separate stacks on new lines
         result += (
            self.label(spindle) + ': ' + # Add spindle label
            str(self.__stacks[spindle])) # and spindle contents
      return result
```

The constructor for `TowerOfHanoi` initializes the private class attributes. The stacks modeling the spindle contents go in a three-element array called `__stacks`. A separate three-element array, `__labels`, holds the names for the spindles—L, M, and R—as in the earlier figures. The total number of disks is stored in `__nDisks`. A default value of 3 is provided for `nDisks`.

The constructor uses a separate method to set up the spindle contents. This method allows the calling program to reset a puzzle object back to the beginning state. The `reset()` method loops through all three spindles, creating a stack that can hold all of the disks on each one (the `SimpleStack` implementation uses an array that requires a maximum size). For the first spindle (at index 0), it pushes the integer diameters of the disks on to the stack, starting with the biggest disk first.

The next methods are accessor functions for key values of the spindles. The `label()` method takes a spindle index and returns the label for the spindle in the puzzle. The spindle indices are 0, 1, and 2 for the three spindles. Similarly, the `height()` method takes a spindle index and returns the number of disks on that spindle by returning the length of the stack representing its contents. The `topDisk()` method also takes a spindle index and returns the diameter of the topmost disk on it. If the spindle is empty, it returns `None`.

The `__str__()` method produces a string to show the puzzle state. It puts each spindle on a different line of text showing the spindle label and the stack of disk diameters. For example, the starting state of the three-disk puzzle looks like this:

```
>>> print(TowerOfHanoi(3))
L: [3, 2, 1]
M: []
R: []
```

The bottom of the stack/spindle is on the left of each line. This uses the `SimpleStack` object's `__str__()` method to show the spindle contents.

The `move()` method in Listing 6-3 handles the movement of disks. It enforces the rules of the puzzle and throws exceptions if they are violated. The parameters are the source and destination spindle numbers (`source` and `to`) and a flag (`show`) indicating whether to print the movement information. The first rule to check is that the source spindle is not empty. If the source spindle is empty, it throws an exception. The second rule checks that the destination spindle is either empty or has a disk larger than the topmost one on the source spindle. In other words, if the destination spindle is not empty and has a top disk smaller than that of the source spindle, it throws a different exception. If both rule checks are satisfied, it performs the move by popping the top disk from the source and pushing it onto the destination stack. Finally, if the `show` flag is set, it prints the move information showing the source and destination spindle labels and the disk diameter.

LISTING 6-3 The `TowerOfHanoi.py` Module—Puzzle Movement and Solution

```
class TowerOfHanoi(object):

… (other definitions shown before) …

   def move(self, source, to,  # Move a single disk from source
            show=False):       # spindle to another, possibly printing
      if self.__stacks[source].isEmpty(): # Source spindle must have
         raise Exception(      # a disk, or it's an error
            "Cannot move from empty spindle " + self.label(source))
      if (not self.__stacks[to].isEmpty() and # Destination cannot
          self.topDisk(source) > # have a disk smaller than that of
          self.topDisk(to)):    # source
         raise Exception(
            "Cannot move disk " + str(self.topDisk(source)) +
            "on top of disk " + str(self.topDisk(to)))
      self.__stacks[to].push(  # Push top disk of source spindle
         self.__stacks[source].pop()) # on to the 'to' spindle
      if show:
         print('Move disk', self.topDisk(to),
               'from spindle', self.label(source),
               'to', self.label(to))

   def solve(self,             # Solve the puzzle to move
             nDisks=None,      # N disks from
             start=0,          # starting spindle
             goal=2,           # to goal spindle
             spare=1,          # with spare spindle
             show=False):      # and possibly showing steps
```

```
if nDisks is None:       # Default number of disks to move
   nDisks = self.height(start) # is all the disks on start
if nDisks <= 0:          # If no request to move disks
   return                # there's nothing to do
if self.height(start) < nDisks: # Check if there are fewer
   raise Exception(             # disks to move than requested
      "Not enough disks (" + str(nDisks) +
      ") on starting spindle " + self.label(start))

self.solve(nDisks - 1,    # Move n - 1 from start to spare with
           start, spare, goal, show) # goal as spare
self.move(start, goal, show)  # Move nth from start to goal
if show: print(self)          # Show puzzle state after move
self.solve(nDisks - 1,    # Then move n - 1 from spare to goal
           spare, goal, start, show) # with start as spare
if (nDisks == self.__nDisks and   # Were all disks moved?
    show):                # then puzzle is solved and can show
   print("Puzzle complete") # conclusion if requested
```

The recursive `solve()` method in Listing 6-3 handles all of the solutions to the puzzle. The solve method can take all default arguments to solve the full puzzle, or it can take all the parameters needed to solve the "subpuzzles" in the recursive steps. In particular, it needs an `nDisks` parameter specifying how many disks are in the pyramid to be moved. This parameter defaults to `None` and is filled in by the first `if` statement with the number of disks on the starting spindle if no value is provided by the caller. The `start`, `goal`, and `spare` parameters are the indices for the starting, goal, and spare spindles. Because the `reset()` method puts all the disks on the first spindle, index 0, label `L`, they default to 0, 2, and 1, respectively. The final parameter is `show`, a flag to indicate whether to display the moves and intermediate states of the puzzle.

The second `if` statement in the `solve()` method checks for the base case. If the number of disks to move is zero (or somehow negative), there is nothing to be done so it simply returns. The third `if` statement verifies that the number of disks to be moved does not exceed the ones stacked on the starting spindle.

The remaining statements in the `solve()` method execute the recursive solution algorithm outlined previously. The first step is to solve the problem of moving *N*–1 disks from the starting spindle to the spare (nongoal) spindle. This is done by reducing the value of `nDisks` and swapping the roles of the spare and goal spindles. The next statement calls the `move()` method to move the *N*th disk, which is now on top of the starting spindle, to the goal spindle. If the `show` flag is set, it then prints the state of the puzzle after the move. The next step is the recursive solution to moving *N*–1 disks from the spare spindle on to the goal spindle using the original starting spindle as the spare. The final `if` statement checks whether the puzzle has been solved and prints a message if the `show` flag is set.

You can test the solution by creating and solving puzzles of various sizes, as shown here:

```
>>> TowerOfHanoi(3).solve(show=True)
Move disk 1 from spindle L to R
L: [3, 2]
M: []
R: [1]
Move disk 2 from spindle L to M
L: [3]
M: [2]
R: [1]
Move disk 1 from spindle R to M
L: [3]
M: [2, 1]
R: []
Move disk 3 from spindle L to R
L: []
M: [2, 1]
R: [3]
Move disk 1 from spindle M to L
L: [1]
M: [2]
R: [3]
Move disk 2 from spindle M to R
L: [1]
M: []
R: [3, 2]
Move disk 1 from spindle L to R
L: []
M: []
R: [3, 2, 1]
Puzzle complete
```

If you run the program with more disks, you should find it dependably enumerating all the steps needed to solve bigger problems. You can use the visualization tool as well. It's quite amazing to see what a few recursive lines of code can produce.

Be careful, however, because the number of moves required to solve a puzzle grows quite fast with the number of disks. How fast?

You can easily find the number of moves needed for the puzzles of size 1 to 10. They're shown in Table 6-2.

Table 6-2 Moves Required to Solve the Tower of Hanoi with N Disks

N	1	2	3	4	5	6	7	8	9	10
Moves	1	3	7	15	31	63	127	255	511	1,023

If you've worked with binary numbers, it should be clear that it takes $2^N - 1$ moves to solve the *N*-disk puzzle. This $O(2^N)$ complexity grows even faster than the sorting algorithms that you saw in Chapter 3, "Simple Sorting," that were $O(N^2)$. For example, 10^2 is 100, but 2^{10} is 1,024. So, if the legend of the monks solving the puzzle with 64 disks is true, they will have to make $2^{64} - 1$ moves to complete it. If they could complete 1 move every 10 seconds on average, that would take over 5.8×10^{12} years. This is not likely to cause the end of the world any time soon.

It's also important to note that in order to find the $2^N - 1$ moves that solve the puzzle, there will be a chain of only $N + 1$ recursive calls at any time during the computation. This is called the **recursive depth** of the algorithm. It corresponds to the number of nested calls like those shown in Figures 6-5, 6-6, 6-7, and 6-8. There are other algorithms where this recursive depth can grow very large and cause an exception when the execution stack runs out of memory.

Sorting with mergesort

Our final example of recursion is the mergesort. It is a much more efficient sorting technique than those you saw in Chapter 3, at least in terms of speed. The mergesort is also fairly easy to implement. It's conceptually easier than quicksort and the Shellshort, which are described in the next chapter.

The downside of the mergesort is that it requires an additional array in memory, equal in size to the one being sorted. If your original array barely fits in memory, the mergesort won't work. If you have enough space, however, it's a good choice.

Merging Two Sorted Arrays

One of computing science's famous pioneers, John von Neumann, invented the mergesort algorithm by considering merging two already-sorted arrays, A and B. Merging them creates a third array, C, that contains all the elements of A and B, arranged in sorted order. We examine this merging process first.

The two sorted arrays don't need to be the same size. Let's say array A has 4 elements and array B has 6. They will be merged into an array C that starts with 10 empty cells. Figure 6-15 shows these arrays just before the last value is copied into array C.

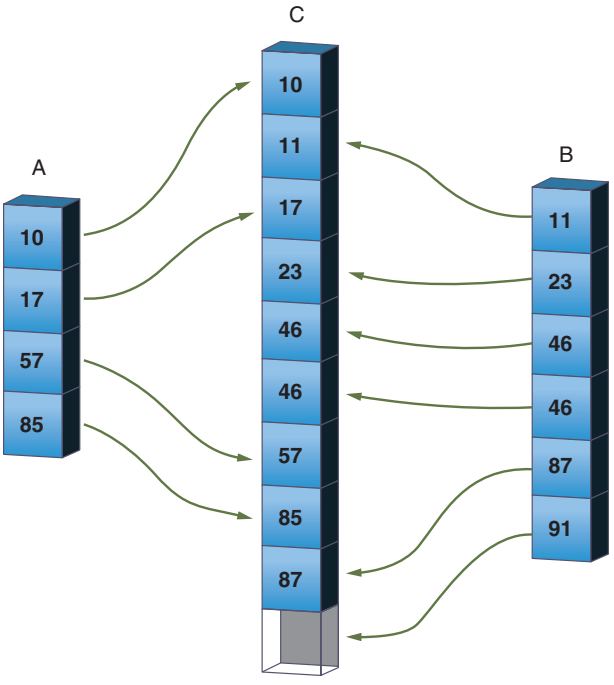


FIGURE 6-15 Merging two arrays

In the figure, the values from arrays A and B are copied to C from top to bottom. Choosing whether to copy from A or B is based on the lowest value remaining to be copied. Table 6-3 shows the comparisons necessary to determine which element will be copied. The steps in the table correspond to the cells of array C in the figure. Following each comparison, the smaller element is copied to C.

Table 6-3 Merging Comparisons and Copies

Step	Comparison (If Any)	Copy
1	Compare 10 and 11	Copy 10 from A to C
2	Compare 17 and 11	Copy 11 from B to C
3	Compare 17 and 23	Copy 17 from A to C
4	Compare 57 and 23	Copy 23 from B to C
5	Compare 57 and 46	Copy 46 from B to C
6	Compare 57 and 46	Copy 46 from B to C
7	Compare 57 and 87	Copy 57 from A to C
8	Compare 85 and 87	Copy 85 from A to C
9		Copy 87 from B to C
10		Copy 91 from B to C

Notice that, because A is empty following step 8, no more comparisons are necessary; all the remaining elements are simply copied from B into C.

Sorting by Merging

The idea in the mergesort is to divide the unsorted, input array in half, sort each half, and then use the merge algorithm just outlined to merge the two halves into a single sorted array. How do you sort each half? This chapter is about recursion, so you probably already know the answer: you divide the half into two quarters, sort each of the quarters, and merge them to make a sorted half.

Similarly, each pair of 8^{ths} is merged to make a sorted quarter, each pair of 16^{ths} is merged to make a sorted 8th, and so on. You divide the array again and again until you reach a subarray with only one element. This is the base case; an array with one element is already sorted.

You've seen that something is reduced in size each time a recursive method calls itself and built back up again each time the method returns. In mergesort, the range of cells is divided in half each time this method calls itself, and each time it returns, it merges two smaller ranges into a larger one.

As the mergesort algorithm returns from processing two arrays of one element each, it merges them into a sorted array of two elements. Each pair of resulting two-element arrays is then merged into a four-element array. This process continues with larger and larger arrays until the entire array is sorted. This sort is easiest to see when the original array size is a power of 2, as shown in Figure 6-16. The input, unsorted array is shown at the left, with the first cell (index 0) at the top. Time elapses moving toward the right. The recursive calls split the eight-element array into halves of four, then quarters of two, and finally into the base case call on a single element.

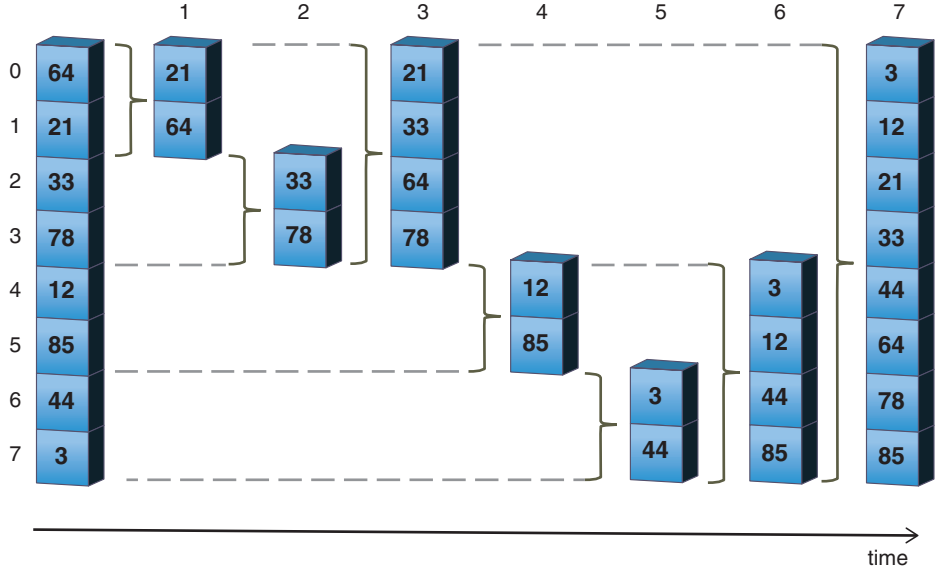


FIGURE 6-16 Mergesorting an eight-element array

Each recursive call works on a range of the array cells. You can use Python slice notation to indicate the ranges being worked in each call. The full array is [0:8], and the first quarter is [0:2]. Note the second index of the slice is one past the last index of the range. Mathematicians might prefer the notation [0,8) for the range of integers 0, 1, 2, 3, 4, 5, 6, and 7, but Python uses [0:8]. We use both in different parts of this book.

Let's assume you have a recursive implementation called `mergesort()` that works on a range in the array. The first call to it is with the full array, [0:8]. Because it's not the base case of a single cell, it makes a recursive call to `mergesort()` the first half of the array, [0:4]. That too is not a base case, so it makes a recursive call to `mergesort()` the first quarter, [0:2], and finally the first eighth, [0:1]. That is a single-cell range, which is already sorted, so that call returns immediately. The same is true of the second eighth, the range [1:2].

The second column (numbered 1) in Figure 6-16 shows the merge step where the first two eighths are merged into a sorted quarter, range [0:2], moving 21 to precede 64. This quarter range is passed back to the recursive call to `mergesort()` on the first half, [0:4]. That call now goes on to `mergesort()` the second quarter, [2:4]. The third column (numbered 2) in Figure 6-16 shows the merge step on the eighths that make up that quarter. There is no change in the values because they were already in sorted order.

The fourth column in Figure 6-16 shows the merging of the first two sorted quarters. The lowest values are chosen first in zipper-like order to make the sorted half. The same process then repeats to dive into the second half of the array, breaking it into quarters, then eighths, and merging the sorted results. In the far-right column of Figure 6-16, the full array is sorted by merging the two sorted halves.

When the array size is not a power of 2, arrays of different sizes must be merged. For example, Figure 6-17 shows the situation when the array size is 10. Here an array of size 2 must be merged with an array of size 1 to form an array of size 3.

As `mergesort()` makes its recursive calls to subdivide the range, it first divides the 10 cells into two ranges of 5 cells each. Those are then subdivided into arrays of size 2 and 3. The two-cell arrays are processed as above, but in the case of the three-cell array, it must be split into unequal subranges. For example, the first half, [0:5], is split into subranges [0:2] and [2:5]. The [2:5] subrange is split into [2:3] and [3:5]. The [2:3] subrange has only one cell, so it is handled as the base case where no sorting needs to be done. The fourth column of Figure 6-17 shows the merge of the one- and two-cell arrays into the three-cell array.

You may wonder how much memory is required to hold all these smaller arrays during the mergesort. To merge two arrays of size M and N together, the algorithm needs another M+N size array to hold the result. Reviewing the basic algorithm:

1. Sort the first half of the array
2. Sort the second half of the array
3. Merge the two sorted halves together

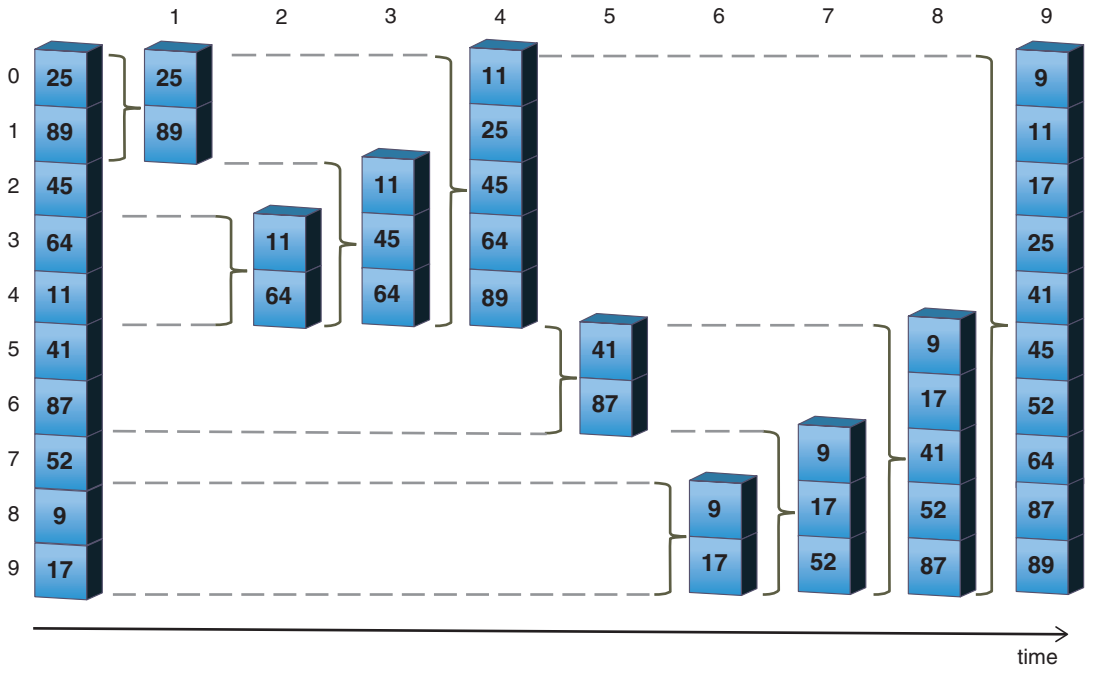


FIGURE 6-17 Mergesort when the array size not a power of 2

only the third step requires temporary storage. The first two steps can be done by putting the result back in the input array cells. So, if you expand step 3 to be

a. Merge the two sorted halves of the input array into a temporary array

b. Copy the temporary array back into the input array

you will need a full copy of the input array, but nothing more, for all the recursive steps. The implementation shown in Listing 6-4 uses a work array to hold the temporary results.

The `Mergesort` class is defined with a single purpose—to sort array objects. This example reuses the `Array` class defined in Chapter 2 for the input and output array. Each `Mergesort` object will be constructed from a single `Array` object. The constructor method takes the unsorted array and a `key` function to get the sort key from each item in the array. The default key function is the identity function that returns the item itself, which can be used to sort an array of primitive types like integers or characters.

The constructor adds private attributes for the `__arr` array and the sort `__key` function. It allocates a `__work` array that is the same size as the input array. All of the recursive calls will be able to use the same work array because they each copy their result back into the input array. The work array is filled with the `None` value to ensure all the cells are allocated. Finally, the constructor calls `mergesort()` on the entire array range, `[0:n]`. There is no return statement because the results will be stored in the input/output array by `mergesort()`.

LISTING 6-4 The `Mergesort.py` Module

```
def identity(x): return x          # Identity function

from Array import *

class Mergesort(object):           # An object to mergesort Arrays
   def __init__(self,              # Constructor takes the unordered
                unordered,         # array and orders its items by using
                key=identity):     # mergesort on their keys
      self.__arr = unordered       # Array starts unordered
      self.__key = key             # Key func. returns sort key of item
      n = len(unordered)           # Get number of items
      self.__work = Array(n)       # A work array of the same length
      for i in range(n):           # is needed to rearrange the items
         self.__work.insert(None) # Work array is filled with None
      self.mergesort(0, n)         # Call recursive sort on full array

   def mergesort(self, lo, hi):    # Perform mergesort on subrange
      if lo + 1 >= hi:             # If subrange has 1 or fewer items,
         return                    # then it is already sorted
      mid = (lo + hi) // 2         # Otherwise, find middle index
      self.mergesort(lo, mid)      # Sort the lower half of subrange,
      self.mergesort(mid, hi)      # Sort the upper half of subrange,
      self.merge(lo, mid, hi)      # Merge the 2 sorted halves
```

The recursive `mergesort()` method is defined to work on subranges of the array between a `lo` and `hi` index (`hi` is one past the highest index). As usual, the first step is to check for the base case, a 1-cell array subrange. It also looks for a 0-cell or empty range just in case the input array was empty. In either of those cases, the input subrange is already sorted, and the method can return without any further action.

For subranges larger than 1, `mergesort()` computes a midpoint index between `lo` and `hi` to divide the subrange into two approximately equal halves, `[lo:mid]` and `[mid:hi]`. Recursive calls sort these subranges. Finally, the two sorted subranges are merged together in the call to `merge()`.

Merging Subranges

The `mergesort()` method calls `merge()` to merge the items from the sorted subranges, making use of the work array as shown in Listing 6-5.

The `merge()` method works with two, adjacent subranges of the array stored in the `__arr` attribute of the object. It steps through the subranges, copying the lowest value in each one to the `__work` array that was created by the constructor. It uses `n` to count the number values that have been copied and `idxLo` and `idxHi` to index the two subranges. The two indices start at the lowest index of their respective ranges and work up.

LISTING 6-5 The `Mergesort.py` Module's `merge()` Method

```
def merge(self, lo, mid, hi): # Merge 2 sorted subranges of input
   n = 0                      # into work array which starts empty
   idxLo = lo                 # Use indices into lo and hi
   idxHi = mid                # subranges to track next items
   while (idxLo < mid and     # Loop until one of the subranges
          idxHi < hi):        # is empty
      itemLo = self.__arr.get(idxLo) # Get next items from the
      itemHi = self.__arr.get(idxHi) # two subranges
      if (self.__key(itemLo) <=   # Compare keys of those items
          self.__key(itemHi)):
         self.__work.set(n, itemLo) # Lo subrange is first so
         idxLo += 1           # copy item and advance to next
      else:
         self.__work.set(n, itemHi) # Hi subrange is first so
         idxHi += 1           # copy item and advance to next
      n += 1                  # One more item now in work array

   while idxLo < mid:         # Loop to copy remaining lo
      self.__work.set(        # subrange items to work array
         n, self.__arr.get(idxLo))
      idxLo += 1
      n += 1

   while n > 0:               # Copy sorted work array contents
      n -= 1                  # back to input/output array in
      self.__arr.set(         # reverse order
         lo + n, self.__work.get(n))
```

The first `while` loop in `merge()` handles the case when there are values to compare from both subranges. It checks that both indices are still within their valid ranges and gets the items at those indices, `itemLo` and `itemHi`. It compares the sort keys by applying the sort `key` function to each one. Whichever item has the lower sort key is copied to the work array. The count of copied items is advanced along with the index for the corresponding subrange.

After one of the subranges becomes empty, no more comparisons are needed, just copying values to their proper positions. The `merge()` method copies any remaining items in the low subrange in the second `while` loop. Finally, the entire `__work` array can be copied back into the subrange of the `__arr` array starting at `lo`. The last `while` loop decrements `n` back to 0 instead of using another index variable for this copying.

Does something look strange in `merge()`? In particular, what about copying any remaining items in the high subrange to the `__work` array? In the previous examples and in Figure 6-15, it shows copying all the items to the work array. Why is that missing in `merge()`?

Because the input array is also the output array, copying the highest part of a range into the work array and then back again accomplishes nothing. When the first `while` loop is done, one of the two ranges has been fully copied to the work array. If it's the lower range, then the second `while` loop has nothing to do. That leaves some or all of the higher range in the input array, `idxHi < hi`, but all those items must have higher key values than what has already been copied. You could copy them to the work array and increment `n`, but they would just be copied back to the same cells by the third `while` loop. Thus, the method skips the unneeded copying and leaves them unchanged.

Testing the Code

The `MergesortClient()` shown in Listing 6-6 tests the basic operation of `Mergesort()`.

LISTING 6-6 The `MergesortClient()` Module

```
from Mergesort import *
from Array import *

values = [19, 49, 70, 72, 43, 80, 95, 46, 19, 18, 45, 6, 56, 85]
array = Array(len(values))
for value in values:
   array.insert(value)

print('Initial array contains', len(array), 'items')
array.traverse()

Mergesort(array)

print('After applying Mergesort, array contains', len(array), 'items')
array.traverse()
```

Because the `Array` class introduced in Chapter 2 doesn't have a `__str__()` method, the test client uses the `traverse()` method to show its contents. Running the test shows

```
$ python3 MergesortClient.py
Initial array contains 14 items
```

```
19
49
70
72
43
80
95
46
19
18
45
6
56
85
After applying Mergesort, array contains 14 items
6
18
19
19
43
45
46
49
56
70
72
80
85
95
```

The Mergesort Visualization Tool

This sorting process is easier to appreciate when you see it happening before your very eyes. Start up the Mergesort Visualization tool. The tool starts with a small array of random numbers. You can create a new array of a particular size using the now-familiar New button and fill it with more random integers using the Random Fill button. You start the mergesort process by selecting the Mergesort button. The first thing it does is create the work array near the bottom to hold the merged ranges. Then the mergesort algorithm starts dividing the array into halves, quarters, and so on. The range of cells that it is working on is moved down for each recursive level. When it reaches a merge step, it copies items into the work array. Figure 6-18 shows an example as it merges a two-cell range of the array.

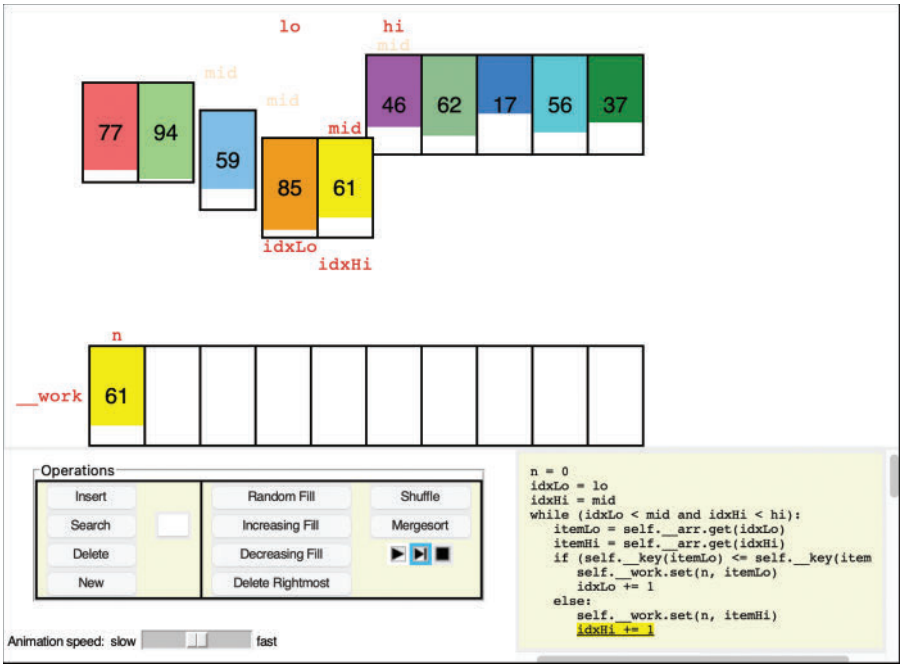


FIGURE 6-18 The Mergesort Visualization tool

The range of cells is split at the `mid` index that's calculated by `mergesort()`. As it descends a level, the range from `lo` to `mid` is moved a little to the left, and the range from `mid` to `hi` is moved a little to the right. The `lo`, `mid`, and `hi` indices change at each level, and the visualization shows only the `mid` index to reduce clutter. It leaves the `mid` index of higher levels in a fainter color and restores the index when the recursion returns to that level.

The `idxLo` and `idxHi` indices show which cells the `merge()` method is examining. The lower keyed item is copied to the work array. After one of the ranges is fully copied, it copies anything remaining in the low range and then copies all `n` items back into the input array cells. The cells move back up as the recursive call ends.

As the algorithm runs, watch the code window. The recursive calls to `mergesort()` stack up, each with different ranges to sort. At the end of each call to `mergesort()`, it pushes the call to `merge()` on the stack to merge the two subranges. You can step through or pause the animation to look at the details of the call stack to see what the different loops do.

Efficiency of the mergesort

Although the bubble, insertion, and selection sorts described in Chapter 3 take $O(N^2)$ time, the mergesort is $O(N \times \log N)$. The graph at the end of Chapter 2 shows how much faster this is. For example, if N (the number of items to be sorted) is 1,048,576 (2^{20}), then N^2 is 1,099,511,627,776, while $N \times \log N$ is only 20,971,520. If sorting this many items required 1 second with the mergesort, it would take over 14 hours for the insertion sort.

How do you know the mergesort takes O(N×log N) time? That's not easy to see just by looking at the algorithm or the implementation, and it's one of the most challenging and interesting parts of computer science. To figure it out, you can count the number of times a data item must be copied and the number of times it must be compared with another data item during the course of the algorithm. We assume that copying and comparing are the most time-consuming operations and that the recursive calls and returns don't add much overhead.

Number of Copies

Consider Figure 6-16, which showed the sorting of an eight-cell array. Each cell to the right of the left-hand column represents an element copied from the input array into the work array. More accurately, they are cells that *could have been copied* into the work array because, as we discussed previously, the items left in the high range didn't need to be copied. Let's assume the worst case, however, and count all the cells as being copied. (We'll handle the copying back to the input/output array shortly.)

Adding up all the cells in Figure 6-16 (the seven columns to the right) shows there are 24 copies necessary to sort 8 items. $Log_2(8)$ is 3, so $8 \times log_2(8)$ equals 24. This shows that, for the case of 8 items, the number of copies is proportional to $N \times log_2 N$.

Another way to look at this calculation is that to sort 8 items requires 3 *levels*, each of which involves 8 copies. A level means all copies go into the same size subrange of the array (or subarray). In the smallest subrange level, there are four 2-cell subarrays; in the next larger level, there are two 4-cell subarrays; and in the third level, there is one 8-cell subarray. Each level has 8 elements, so again there are 3×8, or 24, copies. The levels correspond to the recursive calls to `mergesort()` that divide the range in half.

You can use Figure 6-16 to figure out the number of copies needed for a four-cell array by only considering the top four cells of the array. To sort those cells, eight copies are needed. Checking the formula for that size shows $4 \times log_2(4) = 4 \times 2 = 8$. Going one level lower, two copies are necessary for two items. Similar calculations provide the number of copies necessary for larger arrays. Table 6-4 summarizes this information.

The items are not only copied into the work array but are also copied back into the original array. In the worst case, this process doubles the number of copies, as shown in the Total Copies column. The final column of Table 6-4 shows comparisons, which we cover in a moment.

It's harder to calculate the number of copies and comparisons when N is not a multiple of 2, but these numbers fall between those that are a power of 2. For 10 items, there are 34 cells in Figure 6-17, which means there are at most 34×2 = 68 total copy operations. That count lies between the counts for N = 8 and N = 16, that is, between 48 and 128.

Table 6-4 Number of Operations When N Is a Power of 2

N	$\log_2 N$	Number of Copies into Work Array ($N \times \log_2 N$)	Total Copies	Comparisons Max (Min)
2	1	2	4	1 (1)
4	2	8	16	5 (4)
8	3	24	48	17 (12)
16	4	64	128	49 (32)
32	5	160	320	129 (80)
64	6	384	768	321 (192)
128	7	896	1,792	769 (448)

Number of Comparisons

In the mergesort algorithm, the number of comparisons is always somewhat less than the number of copies depending on the amount of sorting that's needed. How much less? Assuming the number of items is a power of 2, for each individual merging operation, the maximum number of comparisons is always one fewer than the number of items being merged, and the minimum is half the number of items being merged. That minimum is called the **best case**, and it happens when the input array is already in sorted order. The merge copies all the items from the lower half of the array while doing the comparisons. The second loop does nothing because all the lower items are in the work array, and then the final loop copies the lower half back to the input array. The maximum number is the **worst case** and happens when the item's sort keys interleave. The items are copied zipper-like into the work array, and only the last item requires no comparisons. In the example of Figure 6-15, eight comparisons were needed before the last two items could be merged into the result.

Those cases tell us the range of possible comparisons. To get a specific count for a specific input array, you have to add up all the comparisons at all the levels. Referring to Figure 6-16, you can see that seven merge operations are required to sort eight items. The number of items being merged and the resulting number of comparisons are detailed in Table 6-5.

For each merge, the maximum number of comparisons is one fewer than the number of items. Adding these figures for all the merges gives a total of 17. The minimum number of comparisons is always half the number of items being merged, and adding these figures for all the merges results in 12 comparisons. Similar arithmetic results in the values shown in the Comparisons column of Table 6-4.

Table 6-5 Comparisons Involved in Sorting Eight Items

Step/Column Number	1	2	3	4	5	6	7	Totals
Number of items being merged (N)	2	2	4	2	2	4	8	24
Maximum comparisons (N–1)	1	1	3	1	1	3	7	17
Minimum comparisons (N/2)	1	1	2	1	1	2	4	12
Example in Figure 6-16	1	1	3	1	1	3	7	17

The actual number of comparisons to sort a specific array depends on how the input items are arranged, but it will be somewhere between the minimum and maximum values. Because both the best and worst cases depend on N, the average number does too. That means that at every recursion level, there will be O(N) comparisons. Because there are O(log N) levels of recursion, there will be O(N×log N) comparisons overall.

Eliminating Recursion

Some algorithms lend themselves to a recursive approach; some don't. As you've seen, the recursive binary search and `factorial()` functions can be implemented more efficiently using simple loops. Various divide-and-conquer algorithms, however, work very well as recursive routines.

Often an algorithm is easy to conceptualize as a recursive method, but in practice the recursive approach proves to be inefficient. In such cases, it's useful to transform the recursive approach into a nonrecursive approach. Such transformations often make use of a stack.

Recursion and Stacks

There is a close relationship between recursion and stacks. In fact, most compilers implement recursion by using stacks. As we noted, when a method is called, the compiler pushes the arguments to the method and the return address (where control will go when the method returns) on the stack, and then transfers control to the method. When the method returns, it pops these values off the stack. The arguments disappear, and control returns to the return address.

Simulating a Recursive Function: Triangular

In this section we demonstrate how any recursive solution can be transformed into a stack-based solution. Remember the recursive `triangular()` function from the first section in this chapter? Here it is again renamed `triangular_recursive()`:

```
def triangular_recursive(nth): # Get the nth triangular number
   if nth < 1: return 0        # For anything less than 1, it's 0
   return (nth +               # Otherwise add this column to the preceding
           triangular_recursive(nth - 1)) # triangular number
```

We use this function to show the basic idea, even though we know that we could really transform this function into one that doesn't require recursion or looping; for example,

```
def triangular(nth):           # Get the nth triangular number
   return 0 if nth < 1 else nth * (nth + 1) / 2
```

If we didn't know about that closed form solution, we could use a stack to model the recursive calls. Let's start with a stack that has one item on it that defines the problem to be solved. Then we can iterate until the stack is either empty or has one item with a solution to return. Here's the basic idea:

```
def triangular_via_stack(nth): # Get the nth triangular number using
   todo = LinkStack()          # a stack of problem descriptions
   todo.push([nth, None])      # Description: nth and recursive result
   while not todo.isEmpty():   # Loop until no more problems to solve
      top = todo.peek()        # Look at topmost problem
      if top[1] is None:       # If recursive result is not solved,
         if top[0] < 1:        # check if top is base case
            top[1] = 0         # If so, then no recursion needed
         else:                 # Otherwise, solve smaller problem
            todo.push([top[0] - 1, None])
      else:                    # Topmost is solved
         top = todo.pop()      # Pop it off the stack
         if todo.isEmpty():    # If it was the last one,
            return top[1]      # then return the solution
         else:                 # Else add recursive call result to
            caller = todo.peek() # caller's nth which is next on stack
            caller[1] = caller[0] + top[1]
   raise Exception("Stack empty without finding solution")
```

The `todo` variable holds a stack. We choose the `LinkStack` class defined in Chapter 5, "Linked Lists," which does not require a maximum size for the stack contents. The problem descriptions placed on the stack are simple lists of two items: a number that is the value of `nth` from the recursive definition and `None`, which is a placeholder for the result being sought. The algorithm will fill in that value eventually.

After initializing the `todo` stack, the loop iterates until the stack is empty. In the loop body, there must be at least one problem to solve, and that is copied to the `top` variable. The remainder of the loop body performs one of the calls from the recursive version of the function. In the recursive body, it looked first at `nth`, but here we need to look to see if the result value has been filled in for the topmost problem. The reason is that we need to know whether to execute the part of the code before or after the recursive call. It's kind of like the instruction pointer that was pushed onto a call stack telling the interpreter where to resume execution after a recursive call finishes.

When the topmost result is not filled in, which is the way the stack is initialized, the loop body executes the part of the recursive function that comes before the recursive call. That is where it checks whether the problem description is a base case or requires recursion. The base case occurs when the problem is to find the 0^{th} or lower triangular number. That is 0 by definition, so it puts the result in the topmost problem description by setting `top[1]`. If the `top` problem description is for a number 1 or higher, then the recursive case pushes a new, smaller problem description to be solved. That's done by pushing `[top[0] - 1, None]` on the `todo` stack. That's effectively going to perform the recursive call to `triangular_recursive(nth - 1)`. The second component being `None` leaves another placeholder on the stack for the result of that call.

The `else` clause of the outer `if` statement handles when the topmost problem has a solution. That's the point after the recursive call on the smaller problem has returned, so it pops off the smaller result on top of the stack. This is where the recursive version adds the value `nth` to the recursive result and returns that to its caller. In order to "return" a result, the function needs to store a value in the placeholder created by the caller to hold it. First, it checks if there is a "caller" by seeing whether the stack is empty or not. An empty stack means that it has already popped off the original problem description and can simply return the result that was computed for it—the value in `top[1]`.

When the stack is not empty, `triangular_via_stack()` function "returns" the result by adding the result of the recursive call, `top[1]`, to the value of `nth` in the caller. The caller's problem description is at the top of the stack, so it assigns that pair to the `caller` variable by calling `todo.peek()`. The value of the caller's `nth` variable is stored in `caller[0]`. The result of the caller, `caller[1]`, is updated to be the sum of its `nth`, (stored in `caller[0]`), and the smaller recursive call result, `top[1]`.

The code also has an ending that checks whether the stack is ever completely emptied. That should never happen for triangular numbers, so an exception is thrown.

To summarize, writing a stack-based version of a recursive function involves

- Creating a stack to hold problem descriptions
- Pushing the original problem description on the stack
- Including all the parameters that would be passed to a recursive call in each problem description
- Including one or more placeholders for the results of the recursive call (if any are returned)

- Looping until the problem description stack is emptied
- Checking the topmost problem description and handling
  - When the problem has not yet been solved, that is, the part before recursive calls where the base cases are handled
  - Adding any recursive calls to the stack
  - When the recursive calls have been solved, combining their results with other problem description parameters, and "returning" results by putting them in placeholders on the stack, if any
- Storing any local variables from the recursive function as items in the problem description. (There aren't any of these in the preceding `triangular_recursive()` function, so you need only the `nth` function parameter and the result placeholder.)

This entire process of translating recursive functions into nonrecursive versions using stacks can be automated. Some compilers do it for you, but it's important to understand how this process happens. One of the trickiest parts is remembering where all the data gets stored. All the arguments to the recursive function and any local variables must be kept in the problem description on the stack so that all the nested versions of the problem can keep track of their data separately (see Figures 6-5, 6-6, 6-7, and 6-8 for examples). The return values for each recursive call must also be stored there.

Rewriting a Recursive Procedure: mergesort

The process of converting a recursive procedure is almost identical to that of converting a recursive function. Let's convert the `mergesort()` method of Listing 6-4 to look at the differences and similarities.

The `mergesort()` method is the only recursive method of the `Mergesort` class. Because it is part of a class, you can allocate a stack when the object is created to hold the recursive subproblem descriptions. Thus, the class constructor takes care of the first two bullets of the conversion process.

The problem descriptions need to have all the parameters that are passed to the recursive method—namely `lo` and `hi`, which define the subrange of array cells to sort. There is also the `mid` variable that is a local variable of the `mergesort()` method. You can include that in the problem description and put it in the middle so that each description is a list in the form `[lo, mid, hi]`. Each iteration of the loop will examine the topmost problem description on the stack to perform part of the algorithm.

Each loop iteration must handle part of the original routine's processing between recursive calls. Listing 6-7 shows the different parts of the processing using different colors.

The first part of the routine includes the processing steps up to the first recursive call. The routine checks the base case and returns if it's found or creates the local variable, `mid`, in the initial part. The second part is the processing done between the recursive calls, which is nothing (but is still important because the subrange has been sorted). The third and last part follows the second recursive call and involves merging the two sorted subranges of the array.

LISTING 6-7 Parts of the Recursive `mergesort()` Method

```
   def mergesort(self, lo, hi):  # Perform mergesort on subrange
      if lo + 1 >= hi:           # Initial processing
         return
      mid = (lo + hi) // 2
      self.mergesort(lo, mid)    # Processing after 1st recursive call
      self.mergesort(mid, hi)    # Processing after 2nd recursive call
      self.merge(lo, mid, hi)
```

Each iteration of the stack-based processing performs one of these three parts of the routine. It has to decide which part to perform based on the problem description record. You could put a field in the record called `step` or something similar to keep track of what step to perform, but you can also look at the value of the local variable `mid`. The `mid` variable starts off undefined and then is later filled in with a value that is the average of the `lo` and `hi` variables. If you initially create problem descriptions with `mid` as `None` and then set it to the average or some third value, you can use this field of the problem description to determine what step to perform. An easy test would be to check whether `mid` is equal to `lo`. Having that value could indicate that the first recursive call has completed on the [`lo`, `mid`) range, while a higher value indicates that the higher subrange is done, and the last part of the processing is next. Thus, when `mid` is `None` the initial base case processing is done, when it equals `lo`, the first recursive call is done, and when it exceeds `lo`, the second recursive call is done.

The implementation of the `MergesortViaStack.py` module shown in Listing 6-8 uses this approach of altering the `mid` value. It defines the `Mergesort` class in the same fashion as that of Listing 6-4. Inside the constructor, it creates a linked-list stack to hold the problem descriptions to be processed. The overall problem description is pushed onto the `__todo` stack before calling the `mergesort()` method to sort the unordered array. That initial problem description is the list `[0, None, n]` to indicate that it should sort the whole range `[0, n)` of the array, and the `mid` variable is initially undefined.

LISTING 6-8 The `MergesortViaStack.py` Module

```
def identity(x): return x        # Identity function

from Array import *
from LinkStack import *

class Mergesort(object):         # An object to mergesort Arrays
   def __init__(self,            # Constructor, takes the unordered
                unordered,       # array and orders its items by using
                key=identity):   # mergesort on their keys
      self.__arr = unordered     # Array starts unordered
      self.__key = key           # Key func. returns sort key of item
      n = len(unordered)         # Get number of items
```

```
        self.__work = Array(n)                # A work array of the same length
        for i in range(n):                    # is needed to rearrange the items
            self.__work.insert(None)          # Work array is filled with None
        self.__todo = LinkStack()             # Stack to manage subproblems
        self.__todo.push([0, None, n])        # Add overall problem description
        self.mergesort()                      # Call mergesort on problem

    def mergesort(self):                      # Perform mergesort on subrange
        while not self.__todo.isEmpty():      # Loop until no problems remain
            lo, mid, hi = self.__todo.peek()  # Get [lo, mid, hi] values
            print('Mergesort working on [lo, mid, hi] =', # Show progress
                  self.__todo.peek(), 'at depth', len(self.__todo))
            if lo + 1 >= hi:                  # If subrange has 1 or fewer items,
                self.__todo.pop()             # then done, and remove problem
                if self.__todo.isEmpty(): # If that was 1st problem
                    return                    # then everything is done
                self.__todo.peek()[1] = lo # Otherwise, store lo index in
                                              # caller's problem description for
                                              # 'mid' to signal completion
            elif mid is None:                 # If mid is None, need to compute it
                mid = (lo + hi) // 2          # Find middle index, and add subtask
                self.__todo.push(             # for the lower half of subrange
                    [lo, None, mid])
            elif (mid == lo):                 # If mid is lo, lower half is done
                self.__todo.push(             # Add subtask for upper half of
                    [(lo + hi) // 2, None, hi])  # subrange
            else:                             # Both lower half and upper half done
                print('Merging ranges [', lo, ',', mid, ') with [',
                      mid, ',', hi, ')')
                self.merge(lo, mid, hi)       # Merge the 2 sorted halves
                self.__todo.pop()             # Remove completed problem
                if self.__todo.isEmpty():     # If that was the 1st problem,
                    return                    # then everything is done
                self.__todo.peek()[1] = lo    # Otherwise, signal caller
        raise Exception('Empty stack in mergesort')
```

As with the `triangular_via_stack()` function, the `mergesort()` method is a loop that processes problem descriptions on the `__todo` stack until it is empty. It takes the top-most problem description and copies the fields into the same variable names used in the recursive version, `lo`, `mid`, and `hi`. After printing the description record, it goes to the first part of the recursive process, checking the base case. If it is a one-cell or shorter subrange, nothing needs sorting, so it pops off the problem description. If the stack is now empty, then completing this recursive call means everything is done and the stack-based procedure can return. Otherwise, it must signal that it has completed the recursive call by updating the `mid` variable of the caller's problem description. It sets the middle element of the caller's problem description (stored on the top of the stack) to the `lo` value of the

subrange just processed. The "caller" had originally put `None` for that field, and it is now updated to an integer within the subrange the caller was processing.

Setting the `mid` value of the caller to the `lo` value of the problem just solved may seem as though it's going to cause an error in the caller due to that unexpected change. Remember, however, that the stack-based approach is going to use that value to determine what step is being performed, and the caller can easily reconstruct `mid` from its `lo` and `hi` values, which do not change. Note also that by setting this field to `lo`, the caller will be able to tell if the first subrange or second subrange has just been processed.

The next step after handling the base case depends on the value of `mid`. If it's not set, then this is the first part of the processing and the range is bigger than 1 cell. It calculates what `mid` should be and pushes the lower half subrange on the `__todo` stack. Note that `mid` must be at least 1 larger than `lo` because the base case handled when `lo` and `hi` are separated by less than 2.

The next test looks to see if `mid` is the same as `lo`. That indicates the problem description on the top of the stack just finished the recursive call on the lower half, so it pushes a description of the upper half subrange onto the stack. Note that `mid` was set to `lo` by the first recursive call, so the method must recompute the higher subrange starting index as `(lo + hi) // 2`.

The final `else` clause handles the case after returning from the second recursive call, the one that sorted the upper half subrange. The value of `mid` is not `None` nor is it the same as `lo`, so it must have been set to something above `lo` by the recursive call. Now the two subranges can be merged using the exact same `merge()` method shown in Listing 6-5. When they're merged, the problem description can be popped off the stack. If the stack is empty, everything is done. Otherwise, it updates the "caller" problem description on top of the stack to mark the beginning of the range that was just finished. If the "caller" shared the same value for `lo`, then this pass through the loop finished its lower half. If the "caller" value for `lo` is different (lower) than this pass's value, then this pass mergesorted the upper half of the `lo` to `hi` range.

The final line of the `mergesort()` method raises an exception if the stack is emptied unexpectedly. Normally, the return statements in the loop body handle the end of processing.

The stack-based implementation of `mergesort()` in Listing 6-8 includes two `print` statements to show what's going on. Here's a partial transcript of the end of the execution on a 14-cell array.

```
Mergesort working on [lo, mid, hi] = [0, 0, 14] at depth 1
Mergesort working on [lo, mid, hi] = [7, None, 14] at depth 2
Mergesort working on [lo, mid, hi] = [7, None, 10] at depth 3
Mergesort working on [lo, mid, hi] = [7, None, 8] at depth 4
Mergesort working on [lo, mid, hi] = [7, 7, 10] at depth 3
Mergesort working on [lo, mid, hi] = [8, None, 10] at depth 4
Mergesort working on [lo, mid, hi] = [8, None, 9] at depth 5
Mergesort working on [lo, mid, hi] = [8, 8, 10] at depth 4
Mergesort working on [lo, mid, hi] = [9, None, 10] at depth 5
```

```
Mergesort working on [lo, mid, hi] = [8, 9, 10] at depth 4
Merging ranges [ 8 , 9 ) with [ 9 , 10 )
Mergesort working on [lo, mid, hi] = [7, 8, 10] at depth 3
Merging ranges [ 7 , 8 ) with [ 8 , 10 )
Mergesort working on [lo, mid, hi] = [7, 7, 14] at depth 2
Mergesort working on [lo, mid, hi] = [10, None, 14] at depth 3
Mergesort working on [lo, mid, hi] = [10, None, 12] at depth 4
Mergesort working on [lo, mid, hi] = [10, None, 11] at depth 5
Mergesort working on [lo, mid, hi] = [10, 10, 12] at depth 4
Mergesort working on [lo, mid, hi] = [11, None, 12] at depth 5
Mergesort working on [lo, mid, hi] = [10, 11, 12] at depth 4
Merging ranges [ 10 , 11 ) with [ 11 , 12 )
Mergesort working on [lo, mid, hi] = [10, 10, 14] at depth 3
Mergesort working on [lo, mid, hi] = [12, None, 14] at depth 4
Mergesort working on [lo, mid, hi] = [12, None, 13] at depth 5
Mergesort working on [lo, mid, hi] = [12, 12, 14] at depth 4
Mergesort working on [lo, mid, hi] = [13, None, 14] at depth 5
Mergesort working on [lo, mid, hi] = [12, 13, 14] at depth 4
Merging ranges [ 12 , 13 ) with [ 13 , 14 )
Mergesort working on [lo, mid, hi] = [10, 12, 14] at depth 3
Merging ranges [ 10 , 12 ) with [ 12 , 14 )
Mergesort working on [lo, mid, hi] = [7, 10, 14] at depth 2
Merging ranges [ 7 , 10 ) with [ 10 , 14 )
Mergesort working on [lo, mid, hi] = [0, 7, 14] at depth 1
Merging ranges [ 0 , 7 ) with [ 7 , 14 )
After applying Mergesort, array contains 14 items
6
18
19
19
43
45
46
49
56
70
72
80
85
95
```

The transcript starts right after the lower half of the array has been sorted. The top of the stack is `[0, 0, 14]`. That indicates that it should be sorting the range `[0, 14]` and `mid` has been set to 0. That was done by processing the "recursive" subproblem for the lower half, which isn't shown but must have processed the [0, 7) range. It now pushes on the next subproblem to solve `[7, None, 14]`, the upper half of the array. Because the size of

that subrange is larger than 1, it continues pushing on smaller subranges until it gets to `[7, None, 8]` at a stack depth of 4. Because that is a subrange of one cell, it is handled by the base case.

The single-cell base case causes `mergesort()` to return to the problem at depth 3, but with the `mid` value of its caller now set to 7, so the full description is `[7, 7, 10]`. That is a three-cell subrange that must be recursively divided as before. After several more subdivisions to get to the base cases, the transcript has its first merges, [8, 9) with [9, 10). Those are both single-cell subranges. After being merged, the subrange [8, 10) is now sorted. The next merge combines [7, 8) with [8, 10), to complete the three-cell subrange [7, 10).

The rest of the transcript continues handling the various subranges and merging their results. The second-to-last merge merges [7, 10) with [10, 14), the upper half of the entire array. The final merge combines the upper half with the lower half (whose transcript is not shown) to produce the fully sorted array.

The key point to remember is that you can prototype a new function, procedure, or method using recursion and later convert it to a stack-based or simple loop form. The recursive approach is conceptually simpler and can be easier to write, whereas the stack or loop forms gain efficiency.

Some Interesting Recursive Applications

Let's look briefly at some other situations in which recursion is useful. You will see from the diversity of these examples that recursion can pop up in unexpected places. Here, we examine three problems: raising a number to a power, fitting items into a knapsack, and choosing members of a team. We explain the concepts and leave the implementations as exercises.

Raising a Number to a Power

Most scientific calculators, programming languages, and math libraries allow you to raise a number to an arbitrary power. They usually have a function like `power(x, y)` or a key labeled something like x^y or *x*^*y*. How would you do this calculation if your programming language didn't have this function or a math library? You might assume you would need to multiply x by itself y times. That is, if x was 2 and y was 8 (2^8), you would carry out the arithmetic for 2×2×2×2×2×2×2×2. For large values of y, however, this approach might prove longer than necessary. Is there a quicker way?

One solution is to rearrange the problem so that you multiply by multiples of 2 whenever possible, instead of by the number itself. Take 3^8 as an example. Eventually, you must involve eight 3s in the multiplication process. Let's say you start with 3×3=9. You've used up two of the 3s, but there are still six to go. You now have a new number to work with: 9. So you try 9×9=81. This uses four 3s (because each 9 is two 3s multiplied together). You need to use up four more 3s, but now you have 81 to work with, and 81×81=6,561 uses exactly eight 3s (because each 81 used four 3s).

So, you've found the answer to 3^8 with only three multiplications instead of seven. That's O(log N) time instead O(N).

Can you make this process into an algorithm that a computer can execute? The scheme is based on the mathematical equality $x^y = (x^2)^{y/2}$. In this example, $3^8 = (3^2)^{8/2}$, or $3^8 = (3^2)^4$. This is true because raising a power to another power is the same as multiplying the powers.

Remember the assumption, however, that the computer can't raise a number to a power, so it can't handle $(3^2)^4$ directly. Let's see if you can transform this into an expression that involves only multiplication. The trick is to start by substituting a new variable for 3^2.

Let's say that $3^2=a$. Then 3^8 equals $(3^2)^4$, which is a^4. According to the original equality, however, a^4 can be written $(a^2)^2$, so $3^8 = (a^2)^2$.

Again, you can substitute a new variable for a^2, say $a^2=c$, then $(c)^2$ can be written $(c^2)^1$, which also equals 3^8, if you apply all the substitutions.

Now you have a problem you can handle with simple multiplication: c times c.

You can imbed this scheme in a recursive method—let's call it `power()`—for calculating powers. The arguments are b and p, and the method returns b^p. You don't need to worry about variables like a and c anymore because b and p get new values each time the method calls itself. The recursive call arguments are $b \times b$ and $p/2$. For b=3 and p=8, the sequence of arguments and return values would be

```
b=3, p=8
b=9, p=4
b=81, p=2
b=6561, p=1
Returning 6561, b=6561, p=1
Returning 6561, b=81, p=2
Returning 6561, b=9, p=4
Returning 6561, b=3, p=8
```

When p is 1, you return b. The answer, 6561, is passed unchanged back up the sequence of methods.

We've shown an example in which p is an even number throughout the entire sequence of divisions. This will not always be the case. Here's how to revise the algorithm to deal with the situation where p is odd. Use integer division on the way down and don't worry about a remainder when dividing p by 2. However, during the return process, whenever p is an odd number, do an additional multiplication by b. Here's the sequence for 3^{18}:

```
b=3, p=18
b=9, p=9
b=81, p=4
b=6561, p=2
b=43046721, p=1
Returning 43046721, b=43046721, p=1
Returning 43046721, b=6561, p=2
Returning 43046721, b=81, p=4
Returning 387420489, b=9, p=9  # p is odd; so multiply by b
Returning 387420489, b=3, p=18
```

The Knapsack Problem

The knapsack problem is a classic in computer science. In its simplest form, it involves trying to fit items of different weights into a knapsack so that the knapsack ends up with a specified total weight. You don't need to fit in all the items. It's equivalent to the problem of coming up with exact change for a particular value given a specific set of coins.

For example, suppose you want the knapsack to weigh exactly 20 kilograms, and you have five items, with weights of 11, 8, 7, 6, and 5 kilograms. For small numbers of items, humans are pretty good at solving this problem by inspection. So, you can probably figure out that only the 8, 7, and 5 combination of items adds up to 20.

If you want a computer to solve this problem, you need to give it more detailed instructions. Here's the algorithm:

1. If at any point in this process the sum of the selected items adds up to the target, you're done.
2. Start by selecting the first item. The remaining items must add up to the knapsack's target weight minus the first item; this is a new target weight.
3. Try, one by one, each of the possible combinations of the remaining items. Notice, however, that it doesn't really need to try all the combinations, because whenever the sum of the items is more than the target weight, it can stop adding items.
4. If none of the combinations work, discard the first item, and start the whole process again with the second item.
5. Continue this process with the third item and so on until all the combinations have been tried, at which point it knows there is no solution.

In the preceding example, the algorithm starts with the 11 kg weight. Now you want the remaining items to add up to 9 (20 minus 11). Of these, you start with 8, which is too small. Now you want the remaining items to add up to 1 (9 minus 8). You start with 7, but that's bigger than 1, so you try 6 and then 5, which are also too big. You've run out of items, so you know that any combination that includes 8 won't add up to 9. Next, you try 7, so now you're looking for a target of 2 (9 minus 7). You continue in the same way, as summarized here:

```
Items: 11, 8, 7, 6, 5
=======================================
11         // Target = 20, 11 is too small
11, 8      // Target = 9, 8 is too small
11, 8, 7   // Target = 1, 7 is too big
11, 8, 6   // Target = 1, 6 is too big
11, 8, 5   // Target = 1, 5 is too big. No more items
11, 7      // Target = 9, 7 is too small
11, 7, 6   // Target = 2, 6 is too big
11, 7, 5   // Target = 2, 5 is too big. No more items
11, 6      // Target = 9, 6 is too small
```

```
11, 6, 5   // Target = 3, 5 is too big. No more items
11, 5      // Target = 9, 5 is too small. No more items
8,         // Target = 20, 8 is too small
8, 7       // Target = 12, 7 is too small
8, 7, 6    // Target = 5, 6 is too big
8, 7, 5    // Target = 5, 5 is just right. Success!
```

As you may recognize, a recursive routine can pick the first item, and, if the item is smaller than the target, the routine can call itself with a new target to investigate the sums of all the remaining items.

Combinations: Picking a Team

In mathematics, a **combination** is a selection of things in which their order doesn't matter. For example, suppose there is a group of five candidate astronauts from which you want to select a team of three to go on a multiyear journey through the solar system. There is concern, however, about how the team members will get along and handle all the important tasks, so you decide to list all the possible teams; that is, all the possible combinations of three people. You can name the candidates A, B, C, D, and E. You want a program that would show the 10 possible combinations:

ABC, ABD, ABE, ACD, ACE, ADE, BCD, BCE, BDE, CDE

How would you write such a program? It turns out there's an elegant recursive solution. It involves dividing these combinations into two groups: those that begin with A and those that don't. Suppose you abbreviate the idea of 3 people selected from a group of 5 as C(5, 3). The C() function produces a set of teams. Let's say n is the size of the group of candidates and k is the size of a team, that is C(n, k). A theorem says that if A is the first of the n candidates, then

$$C(n, k) = A \otimes C(n - 1, k - 1) + C(n - 1, k)$$

where A ⊗ C(e) means add (cross) A to all the combinations produced by C(e), and C(n – 1, j) means find all combinations of size j from the candidates other than A, which is one of the n candidates. For the example of 3 people selected from a group of 5, you have

$$C(5, 3) = A \otimes C(4, 2) + C(4, 3)$$

This example breaks a large problem into two smaller ones. Instead of selecting from a group of 5, you're selecting twice from a group of 4, the candidates other than A. First, all the ways to select 2 people from 4 candidates, then all the ways to select 3 people from 4 candidates.

There are 6 ways to select 2 people from a group of 4. In the C(4, 2) term—which you can call the left term—these 6 combinations are

BC, BD, BE, CD, CE, DE

The theorem tells you to add (cross) A to all of these 2-person combinations:

ABC, ABD, ABE, ACD, ACE, ADE

There are four ways to select 3 people from a group of 4. In the C(4, 3) term—the right term—you have

BCD, BCE, BDE, CDE

When these 4 combinations from the right term are added to the 6 from the left term, you get the 10 combinations for C(5, 3).

You can apply the same decomposition process to each of the groups of 4. For example, C(4, 2) is B × C(3, 1) added to C(3, 2), where both A and B have been taken out of the candidates. As you can see, this is a natural place to apply recursion.

You can think of this problem visually with C(5, 3) as a single node on the top row, C(4, 2) and (4, 3) as nodes on the next row, and so on, where the nodes correspond to recursive function calls. Figure 6-19 shows what this looks like for the C(5, 3) example.

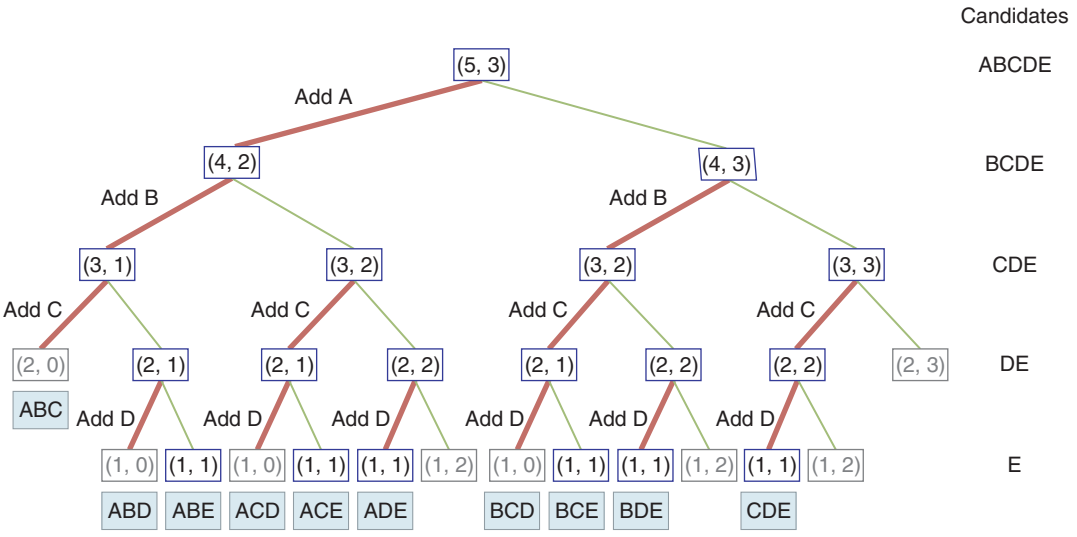


FIGURE 6-19 Picking a team of 3 from a group of 5

The base cases are combinations that make no sense: those with a 0 for the team size and those where the team size is greater than the number of candidates. The combination C(1, 1) is valid, but there's no point trying to break it down further. In the figure, grayed-out boxes show the base cases where recursion ends.

The recursion depth corresponds to the candidates: the node on the top row represents candidate A with its two choices of whether to include A or not in the team. The two nodes on the next row represent candidate B, and so on. If there are 5 candidates, you'll have 5 levels.

As the algorithm descends the rows, it needs to remember the sequence of candidates visited. Here's how to do that: Whenever it makes a call to a left term, it records the node it's leaving by adding its letter to a sequence. These left calls and the letters to add to the sequence are shown by the darker and thicker lines in the figure. The sequence needs to be passed back up on the returns.

To record all the combinations, the algorithm can add them to a list or display them as they are found. After a complete team is assembled, it's either added to the list or printed (or both).

Summary

- A recursive method calls itself repeatedly, with different argument values each time.
- Some value(s) of its arguments causes a recursive method to return without calling itself. This is called a base case.
- When the innermost instance of a recursive method returns, the process "unwinds" by completing pending instances of the method, going from the latest back to the original call.
- The *n*th triangular number is the sum of all the numbers from 1 to *n*. (*Number* means *integer* in this context.) For example, the triangular number of 4 is 10, because 4+3+2+1 = 10.
- The factorial of an integer number is the product of itself and all numbers smaller than itself. For example, the factorial of 4 is 4×3×2×1 = 24.
- Both triangular numbers and factorials can be calculated using either a recursive method or a simple loop.
- The anagrams of a word (all possible permutations of its N letters) can be found recursively by placing the leftmost letter at every position within all the anagrams of the rightmost N–1 letters.
- A binary search can be carried out recursively by checking which half of a sorted range the search key is in, and then doing the same kind of search within that half.
- The Tower of Hanoi puzzle consists of three spindles and an arbitrary number of disks stacked in a tower on one of the spindles.
- Solving the Tower of Hanoi involves moving disks one at a time between spindles until they are all stacked on the goal spindle, without ever placing a larger disk on top of a smaller disk.
- The Tower of Hanoi puzzle can be solved recursively by moving all but the bottom disk of a pyramid to an intermediate tower, moving the bottom disk to the destination tower, and finally moving the remaining pyramid to the destination.
- The number of steps needed to solve a Tower of Hanoi puzzle with N disks is $O(2^N)$.
- $O(2^N)$ is significantly worse than $O(N^2)$ complexity.
- Merging two sorted arrays means creating a third array that contains all the elements from both arrays in sorted order.
- In mergesort, one-element subarrays of a larger array are merged into two-element subarrays, two-element subarrays are merged into four-element subarrays, and so on until the entire array is sorted.
- Mergesort requires O(N×log N) time for both comparisons and copies.

- Mergesort requires a workspace equal in size to the original array.
- For triangular numbers, factorials, anagrams, and the binary search, the recursive method contains only one call to itself. (There are two shown in the code for the binary search, but only one is used on any given pass through the method's code.)
- For the Tower of Hanoi and mergesort, the recursive method contains two calls to itself.
- Any operation that can be carried out with recursion can be carried out with a stack.
- There is a process to convert recursive functions and procedures into stack-based algorithms.
- A recursive approach may be inefficient. If so, it can sometimes be replaced with a simple loop, a stack-based approach, and very occasionally without iteration.
- Prototyping in recursive form is simpler conceptually and can be useful to test an algorithm's accuracy and correctness.

Questions

These questions are intended as a self-test for readers. Answers may be found in Appendix C.

1. Which of the following is **not** a characteristic of recursive routines?

   a. They call themselves.

   b. Each call performs its work on a smaller version of the same problem.

   c. When a smaller version of the problem is too complex, control passes back to the caller to try a different approach.

   d. Some versions of the problem don't require calling the recursive routine.

2. If a program calls `triangular(100)` using the definition in Listing 6-1, what is the maximum number of "copies" of the `triangular()` function in execution that exist at any one time?

3. Where are the copies of the argument passed to the `triangular()` function, mentioned in question 2, stored?

   a. in a variable in the `triangular()` function

   b. in a field of the `Triangular` class

   c. in a placeholder variable of the problem description record

   d. on a stack

4. Assume a call to `triangular(100)` as in question 2. What is the value of `nth` right after the `triangular()` function first returns a nonbase case value?

5. True or False: In the `triangular()` function of Listing 6-1, the return values are stored on the stack.

6. In the `anagrams()` function, at a certain depth of recursion, assume a version of the function is working with the string `"que"`. When this method calls a new version of itself, what letters will the new version be working with?
7. In the section "A Recursive Binary Search," the original, loop-based form was compared with a recursive form of the `find()` method. Which of the following is **not** true?
   a. The search range starts with the whole array, and only the recursive version can work on a subrange passed through arguments.
   b. Both forms of the program divide the search range repeatedly in half.
   c. If the key is not found, the loop version returns when the range bounds cross, but the recursive version finishes when the recursive depth is more than half the initial search range.
   d. If the key is found, the loop version returns from the entire method, whereas the recursive version returns from one level of recursion.
8. What kind of subproblem is solved in the recursive calls of the `TowerOfHanoi.solve()` method (Listing 6-3) as compared to the overall problem?
9. The algorithm in the `TowerOfHanoi.solve()` method involves
   a. dividing the number of disks on the source spindle in half.
   b. changing which spindles are the source and destination.
   c. removing the small disks from all the spindles to move the large disks.
   d. moving one small disk and then a stack of larger disks.
10. Which is **not** true about the `Mergesort.merge()` method in Listing 6-5?
    a. Its algorithm can handle arrays of different sizes.
    b. It must search the target array to find where to put the next item.
    c. It is not recursive.
    d. It continuously copies the smallest item irrespective of what input array it's in.
11. The disadvantage of mergesort is that
    a. it is not recursive.
    b. it uses more memory.
    c. although faster than the insertion sort, it is much slower than bubble sort.
    d. it is complicated to implement.
12. Using the recursive version of `mergesort()` in Listing 6-4, what recursive depth will be reached in a call to sort an array of 1,024 cells?
13. In addition to a loop, a _____ can often be used instead of recursion.

14. In the procedure outlined for converting a recursive function to a nonrecursive algorithm, how are function arguments and local variables of the recursive version stored?

15. In the procedure outlined for converting a recursive function to a nonrecursive function, what test is used to decide when the function should return?

Experiments

Carrying out these experiments will help to provide insights into the topics covered in the chapter. No programming is involved.

6-A Use a Python interpreter to load the definitions for `triangular_loop()` and `triangular()` shown in the "Triangular Numbers" section. Run both routines to get the results for the 100th, 1,000th, and 10,000th triangular number. What's different about the results? Does one routine take noticeably longer?

6-B This chapter discussed triangular numbers and factorials. It showed recursive algorithms to generate them and a recursive algorithm to generate anagrams. How are anagrams related to one or both of those numeric sequences?

6-C The mergesort algorithm is another sorting algorithm like the ones introduced in Chapter 3. Do you think it is stable? In other words, will any two items in the input array with equal keys remain in the same relative ordering in the resulting array? Why or why not? Thinking about the simple base cases first can be helpful. The recursive nature of the algorithm determines the rest.

6-D In Chapter 3, you saw that in the best case, an insertion sort would take O(N) time. Can you think of what the best case input array would be for mergesort? Would sorting that best case array take less than its worst case time, O(N×log N)?

Programming Projects

Writing programs to solve the Programming Projects helps to solidify your understanding of the material and demonstrates how the chapter's concepts are applied. (As noted in the Introduction, qualified instructors may obtain completed solutions to the Programming Projects on the publisher's website.)

6.1 Suppose you are asked to write a program on an embedded processor that has very limited processing capability. This particular processor's Python interpreter can only do addition and subtraction, not multiplication, and that's needed for this project. You program your way out of this quandary by writing a recursive function, `mult()`, that performs multiplication of x and y by adding x to itself y times. Its arguments are integers, and its return value is the product of x and y. Write such a method and a test program to call it. Does the addition take place when the function calls itself or when it returns?

6.2 Every positive integer can be divided into a set of integer factors. The minimum set of factors must be a collection of prime numbers, where a prime number is one that is only evenly divisible by 1 and itself. Write a recursive function, `factor()`, that returns the list of integer factors of *x*. If you haven't worked with factoring before, it is helpful to know that only the factors between 2 and the square root of *x* need to be tested. Instead of doing those with a loop, add a second, optional parameter to `factor(x, lowest)` that is the lowest possible integer factor of *x*. The recursive function should check for the base case(s), and make recursive calls based on whether `lowest` evenly divides `x` (`x % lowest == 0`). If it does, then add `lowest` to the factors of *x* divided by `lowest`. If `lowest` doesn't evenly divide *x*, then look for factors with the next higher possible factor. Ideally, the next higher factor would always be the next higher prime number. There are algorithms for that, but for this exercise, it's sufficient to try the next higher integer. You may use Python's built-in `list` structure to assemble the factors or the `LinkedList` structure of Chapter 5. Note that a factor can appear more than once in the final list. For extra utility, handle requests for the factors of negative integers by returning the factors of the positive version but with the factor 1 replaced by –1. Test your function on some compound (nonprime) and prime numbers and the special cases of 0 and 1.

6.3 Implement the recursive approach to raising a number to a power, as described in the "Raising a Number to a Power" section near the end of this chapter. Write the recursive `power()` function. For extra utility, make use of this transformation to handle negative integer exponents:

$$x^y = \frac{1}{x^{-y}} \qquad when\ y < 0$$

Test your function on several combinations including positive and negative integers and the special cases where the exponent is 0 and 1 ($x^0 = 1$ and $x^1 = x$).

6.4 Write a program that solves the knapsack problem for an arbitrary knapsack capacity and series of weights. Assume the weights are stored in an array. Hint: The arguments to the recursive `knapsack()` function are the target weight, the array or weights, and the index into that array where the remaining items start. The `knapsack()` function should either return a list of weights or print each list of weights that add up to the target weight. You may use Python's built-in `list` structure to assemble the lists or the `LinkedList` structure of Chapter 5.

6.5 Implement a recursive approach to showing all the teams of size *k* that can be created from a group of *n* people. For this exercise, represent the *n* people with a single character, for example, A, B, T, Z. The list of all possible candidates can be represented as a string, for example, 'ABTZ'. The recursive `teams()` function should take the candidates string and *k* as parameters and generate a list of strings that are the different teams. Teams are formed by concatenating character strings.

CHAPTER 7

Advanced Sorting

IN THIS CHAPTER

- Shellsort
- Partitioning
- Quicksort
- Radix Sort
- Timsort

We started discussing sorting in the aptly titled Chapter 3, "Simple Sorting." The sorts described there—the bubble, selection, and insertion sorts—are easy to implement but rather slow. In Chapter 6, "Recursion," we described the mergesort. It runs much faster than the simple sorts but requires twice as much space as the original array; this is often a serious drawback.

This chapter covers two advanced approaches to sorting in detail: Shellsort and quicksort. These sorts both operate much faster than the simple sorts: the Shellsort in about $O(N \times (\log N)^2)$ time, and quicksort in $O(N \times \log N)$ time. Neither of these sorts requires a large amount of extra space, as mergesort does. The Shellsort is almost as easy to implement as mergesort although quicksort is the fastest of all the general-purpose sorts that don't require a lot of memory.

We examine the Shellsort first. Quicksort is based on the idea of partitioning, so we examine that separately, before examining quicksort itself. We conclude the chapter with short descriptions of radix sort and Timsort, which are quite different approaches to fast sorting.

Shellsort

The Shellsort is named for Donald L. Shell, the computer scientist who discovered it in 1959. It's based on the insertion sort but adds a new feature that dramatically improves the insertion sort's performance.

The Shellsort is good for medium-sized arrays, depending on the particular implementation. It's not quite as fast as quicksort and other $O(N \times \log N)$ sorts, so it's not optimum

for very large collections. Shellsort, however, is much faster than the $O(N^2)$ sorts like the selection sort and the insertion sort, and it's very easy to implement. The code is short and simple.

Shellsort's worst-case performance is not significantly worse than its average performance. (Later in this chapter we look into that the worst-case performance for quicksort, which can be much worse unless precautions are taken.) Some experts (see Sedgewick in Appendix B, "Further Reading") recommend starting with a Shellsort for almost any sorting project and changing to a more advanced sort, like quicksort, only if Shellsort proves too slow in practice.

Insertion Sort: Too Many Copies

Because Shellsort is based on the insertion sort, you might want to review the "Insertion Sort" section in Chapter 3. Recall that partway through the insertion sort the items to the left of a marker are partially sorted (sorted among themselves), and the items to the right are not. The algorithm removes the item at the marker and stores it in a temporary variable. Then, beginning with the item to the left of the newly vacated cell, it shifts the sorted items right one cell at a time, until the item in the temporary variable can be reinserted in sorted order.

Here's the problem with the insertion sort. Suppose a small item is on the far right, where the large items should be. To move this small item to its proper place on the left, all the intervening items (between the place where it is and where it should be) must be shifted one space right. This step takes close to N copies, just for one item. Not all the items must be moved a full N spaces, but the average item must be moved N/2 spaces, which means the full process takes N times N/2 shifts for a total of $N^2/2$ copies. Thus, the performance of insertion sort is $O(N^2)$.

This performance could be improved if you could somehow move a smaller item many spaces to the left without shifting all the intermediate items individually.

N-Sorting

The Shellsort achieves these large shifts by insertion-sorting widely spaced elements. After they are sorted, Shellsort sorts somewhat less widely spaced elements, and so on. The spacing between elements for these sorts is called the *increment* and is traditionally represented by the letter *h*. Figure 7-1 shows the first step in the process of sorting a 10-element peg board (array) with an increment of 4. Here the elements at indices 0, 4, and 8 are sorted.

After the pegs at 0, 4, and 8 are sorted, the algorithm shifts over one cell and sorts the pegs at locations 1, 5, and 9. This process continues until all the pegs have been *4-sorted*, which means that all items spaced four cells apart are sorted among themselves. Figure 7-2 illustrates the process with different panels showing the state after each subset is sorted.

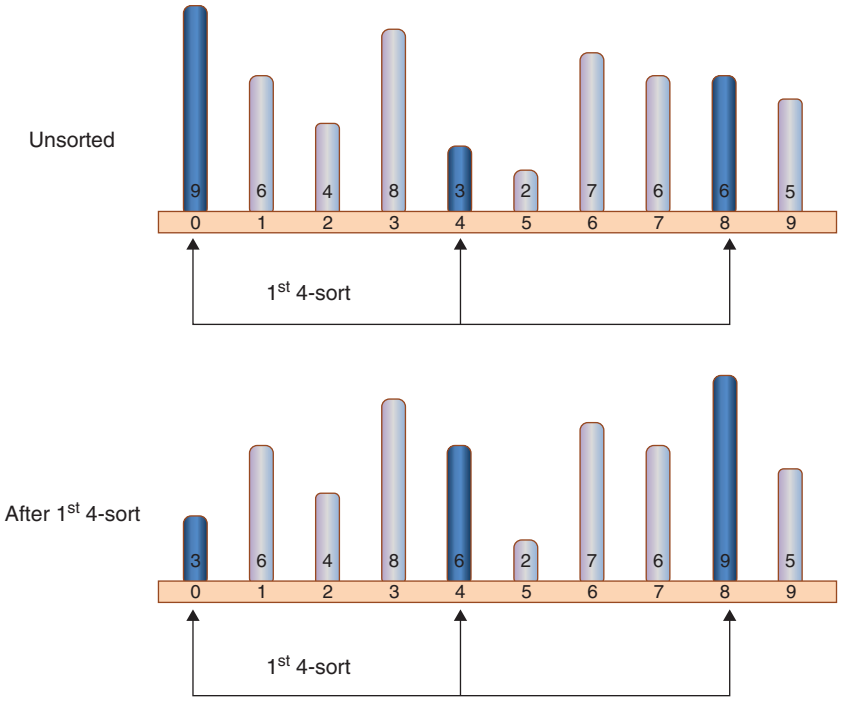


FIGURE 7-1 4-sorting at 0, 4, and 8

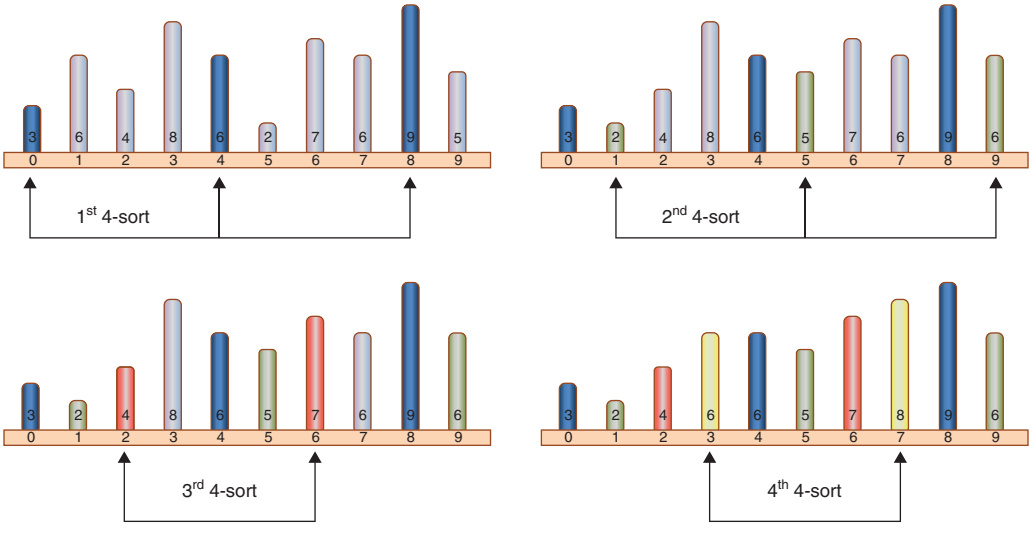


FIGURE 7-2 A complete 4-sort

After the complete 4-sort, the array can be thought of as comprising four subarrays with heights: [3,6,9], [2,5,6], [4,7], and [6,8], each of which is completely sorted. These subarrays are interleaved at locations [0,4,8], [1,5,9], [2,6], and [3,7] respectively—as shown by the colors of the pegs in the figure—but otherwise independent.

What has been gained? Well, the *h*-sorts have reduced the size of the number of items to be sorted in a single group. Instead of sorting N items, you've sorted N/*h* items *h* times. Because the insertion sort is an algorithm that is $O(N^2)$, that makes *h*-sorting $O((N/h)^2 \times h) = O(N^2/h)$, so that's an improvement. If *h* were big, say N/2, then the overall complexity would get down to O(N). Of course, the array isn't fully sorted yet, but note that *the items are closer to their final positions*.

In the example of sorting 10 items in Figure 7-2, at the end of the 4-sort, no item is more than three cells from where it would be if the array were completely sorted. This is what is meant by an array being "almost" sorted and is the secret of the Shellsort. By creating interleaved, internally sorted sets of items using "inexpensive" *h*-sorts, you minimize the amount of work that must be done to complete the sort.

Now, as we noted in Chapter 3, the insertion sort is very efficient when operating on an array that's almost sorted. In fact, for an already-sorted array, it's O(N). If it needs to move items only one or two cells to sort the array, it can operate in almost O(N) time. Thus, after the array has been 4-sorted, you can 1-sort it using the ordinary insertion sort. The combination of the 4-sort and the 1-sort is much faster than simply applying the ordinary insertion sort without the preliminary 4-sort.

Diminishing Gaps

We've shown an initial interval—or gap—of four cells for sorting a 10-cell array. For larger arrays, the interval should start out much larger. The interval is then repeatedly reduced until it becomes 1.

For instance, an array of 1,000 items might be 364-sorted, then 121-sorted, then 40-sorted, then 13-sorted, then 4-sorted, and finally 1-sorted. The sequence of numbers used to generate the intervals (in this example, 364, 121, 40, 13, 4, 1) is called the **interval sequence** or **gap sequence**. The particular interval sequence shown here, attributed to Knuth (see Appendix B), is a popular one. In reversed form, starting from 1, it's generated by the expression

$$h_i = 3 \times h_{i-1} + 1$$

where the initial value of h_0 is 1. Table 7-1 shows how this formula generates the sequence.

There are other approaches to generating the interval sequence; we return to this issue later. First, let's explore how the Shellsort works using Knuth's sequence.

Table 7-1 Knuth's Interval Sequence

i	h_i	$3 \times h_i + 1$	$(h_i - 1) / 3$
0	1	4	
1	4	13	1
2	13	40	4
3	40	121	13
4	121	364	40
5	364	1,093	121
6	1,093	3,280	364

In the sorting algorithm, the sequence-generating formula is first used in a short loop to figure out the initial gap. A value of 1 is used for the first value of *h*, and the $h_i = 3 \times h_{i-1} + 1$ formula is applied to generate the sequence 1, 4, 13, 40, 121, 364, and so on. This process ends when the gap is larger than the array. For a 1,000-element array, the seventh number in the sequence, 1,093, is too large. Thus, Shellsort begin the sorting process with the sixth-largest number, creating a 364-sort. Then, each time through the outer loop of the sorting routine, it reduces the interval using the inverse of the formula previously given

$$h_{i-1} = (h_i - 1) / 3$$

This sequence is shown in the last column of Table 7-1. This inverse formula generates the reverse sequence 364, 121, 40, 13, 4, 1. Starting with 364, each of these numbers is used to *h*-sort the array. When the array has been 1-sorted, the algorithm is done.

The AdvancedSorting Visualization Tool

You can use the AdvancedSorting Visualization tool to see how Shellsort works. Figure 7-3 shows the tool as the Shellsort begins and determines what *h* value to use.

Like the Simple Sorting Visualization tool introduced in Chapter 3, this tool provides the common operations for inserting, searching, and deleting single items. It also allows you to create new arrays of empty cells and fill empty cells with random keys or keys that increase or decrease.

As you watch the Shellsort, you'll notice that the explanation we gave in the preceding discussion is slightly simplified. The sequence of indices processed by the 4-sort is not actually [0, 4, 8], [1, 5, 9], [2, 6], and [3, 7]. Instead, the first two elements of the first group of three are sorted, then the first two elements of the second group, and so on. After the first two elements of all the groups are sorted, the algorithm returns and sorts the last element of the three-element groups. This is what happens in the insertion sort; the actual sequence is [0, 4], [1, 5], [2, 6], [3, 7], [0, 4, 8], [1, 5, 9].

It might seem more obvious for the algorithm to 4-sort each complete subarray first, that is, [0, 4], [0, 4, 8], [1, 5], [1, 5, 9], [2, 6], [3, 7], but the algorithm handles the array indices more efficiently using the first scheme.

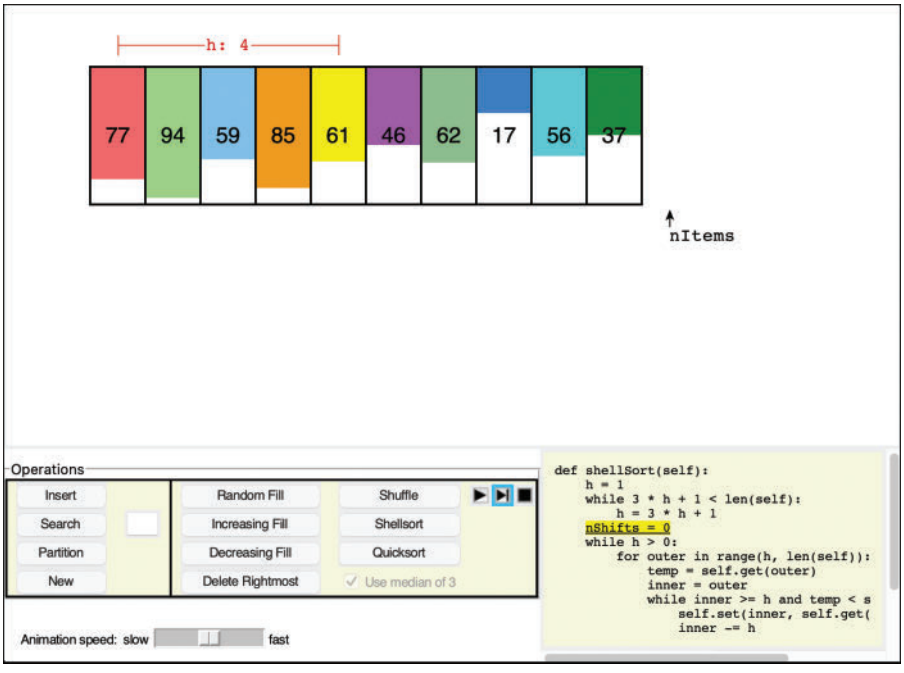


FIGURE 7-3 The AdvancedSorting Visualization tool

The Shellsort is actually not very efficient with only 10 items, making almost as many swaps and comparisons as the insertion sort. With larger arrays, however, the improvement becomes significant.

Figure 7-4 shows the AdvancedSorting Visualization tool starting with a 90-cell array of inversely sorted bars. (Creating such a large array in the visualization tool requires making the window much wider before using the New button.) The Decreasing Fill button fills the empty cells with the values in the top row of the figure. The second row shows the array after it has been completely 40-sorted. On the left, the outer and inner indices are set to their values for the beginning of the 13-sort. The first two cells to be compared in the 13-sort are lowered from their normal position to show what cells are being checked. The third row shows the beginning of the 4-sort.

With each new value of *h*, the array becomes more nearly sorted. The groups of 40 are clear in the second row of Figure 7-4. The first 10 items are the shortest of the whole array. They came from the top 10 items of the initial array during the 40-sort. The next groups of 40 cells have runs of 30 and 10 decreasing keys each. The runs of 30 came from the 40-sorts on subarrays like indices [11, 51]. They keep the overall trend of decreasing keys, but each subarray is now in increasing order. When you get to the 4-sort starting in the last row of the figure, the overall trend now looks to be generally increasing. Every item is now within 13 cells of its final position. After the 4-sort, every item will be within four cells of its final position.

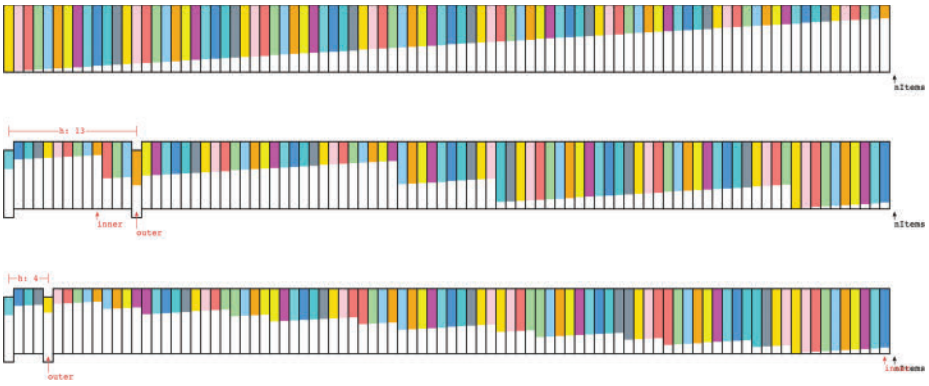


FIGURE 7-4 After the 40-sort and 13-sort in a 90-cell array

Why is the Shellsort so much faster than the insertion sort, on which it's based? As mentioned previously, when *h* is large, the number of items per pass is small, and items move long distances. This sort is very efficient. As *h* grows smaller, the number of items per pass increases, but the items are already closer to their final sorted positions, which is more efficient for the insertion sort. It's the combination of these trends that makes the Shellsort so effective.

Notice that later sorts (small values of *h*) don't undo the work of earlier sorts (large values of *h*). An array that has been 40-sorted remains 40-sorted after a 13-sort, for example. If this wasn't so, the Shellsort couldn't work.

Python Code for the Shellsort

The Python code for the Shellsort is scarcely more complicated than for the insertion sort shown in Chapter 3. The key difference, of course, is the addition of the *h* interval and ensuring that the program takes steps of *h* elements instead of 1. Here, we add a `ShellSort()` method to the `Array` class (that was defined in the `SortArray.py` module) and put it into the `ShellSort.py` module shown in Listing 7-1.

The first part sets the initial `h` value. Starting at 1, it follows Knuth's sequence to increment `h`, as long as it stays below about a third of the array size. A variable, `nShifts`, has been added to count the number of shift operations performed during the sorting. It's not really necessary but helps to illustrate and instrument the actual complexity on specific arrays.

The main loop iterates over the `h` value, decreasing it through Knuth's sequence until it gets below 1. The revised insertion sort algorithm appears inside the main loop. This time, `outer` starts at `h`. That is the index of the first cell that is not yet sorted (remember every cell below `outer`—stepping in increments of `h`—is already sorted). The value at `outer` is stored in `temp` as the marked item. The inner loop then sets its index, `inner`, to start at `outer` and work backward to lower indices stepping by `h`. After the inner loop finds that `inner` has reached the last possible index for this *h*-sort or the marked item, `temp`, is bigger than or equal to what is in the cell at `inner - h`, it stops. As the inner loop steps backward, it shifts the contents of the cells it passes (where the marked item is less than the value at `inner - h`). It increments the `nShifts` counter accordingly and does so once more at the end of the inner loop, if moving the marked item from `temp` back to the array is warranted.

LISTING 7-1 The ShellSort.py Module

```
import SortArray

class Array(SortArray.Array):            # Base new Array class on SortArray

   def ShellSort(self):                 # Sort using Shell's method:
      h = 1                             # Choose h to sort every h items
      while 3 * h + 1 < len(self):      # Use Knuth's h sequence, find
         h = 3 * h + 1                  # largest h less than array length
      nShifts = 0                       # Count number of shifts
      while h > 0:                      # Loop over decreasing h values
         for outer in range(h, len(self)): # Mark one item
            temp = self.get(outer)      # Store marked item in temp
            inner = outer               # Inner loop starts at mark
            while inner >= h and temp < self.get(inner-h): # If marked
               self.set(inner, self.get(inner-h)) # item smaller, then
               inner -= h               # shift item to right
               nShifts += 1             # Count shift
            if inner < outer:           # If inner loop advanced a step, then
               self.set(inner, temp)    # Move marked item to 'hole'
               nShifts += 1             # and count the shift
         h = (h - 1) // 3               # Reduce h to sort smaller intervals
      return nShifts                    # Return number of shifts
```

After completing the first pass, the `outer` variable is incremented by 1 to move on to the next *h*-sort group. In terms of the first example, this means going from the [0, 4] subsequence to the [1, 5] subsequence. Incrementing `outer` by 1 all the way up to the length of the array eventually covers the longer subsequences like [0, 4, 8], [1, 5, 9], and so on. The outermost `while` loop decreases `h` following Knuth's formula to perform the *h*-sort on smaller increments.

The Shellsort algorithm, although it's implemented in just a few lines, is not simple to follow. Here's a sample output from running it with some print statements:

```
$ python3 ShellSortClient.py
Array containing 100 items:
 [77, 94, 59, 85, 61, 46, 62, 17, 56, 37, 18, 45, 76, 21, 91, 7, 96, 50, 31, 69, 80,
69, 56, 60, 26, 25, 1, 2, 67, 46, 99, 57, 32, 26, 98, 51, 77, 34, 20, 81, 22, 40,
28, 23, 69, 39, 23, 6, 46, 1, 96, 51, 71, 61, 2, 34, 1, 55, 78, 91, 69, 23, 2, 8,
3, 78, 31, 25, 26, 73, 28, 88, 88, 38, 22, 97, 9, 18, 18, 66, 47, 16, 82, 9, 56, 45,
15, 76, 85, 52, 86, 5, 28, 67, 34, 20, 6, 33, 83, 68]
initArray().ShellSort() took 0.1190280788578093 seconds
```

```
Sorted array contains:
 [1, 1, 1, 2, 2, 2, 3, 5, 6, 6, 7, 8, 9, 9, 15, 16, 17, 18, 18, 18, 20, 20, 21, 22,
22, 23, 23, 23, 25, 25, 26, 26, 26, 28, 28, 28, 31, 31, 32, 33, 34, 34, 34, 37, 38,
39, 40, 45, 45, 46, 46, 46, 47, 50, 51, 51, 52, 55, 56, 56, 56, 57, 59, 60, 61, 61,
62, 66, 67, 67, 68, 69, 69, 69, 69, 71, 73, 76, 76, 77, 77, 78, 78, 80, 81, 82, 83,
85, 85, 86, 88, 88, 91, 91, 94, 96, 96, 97, 98, 99] and took 565 cell shifts
```

Now that you've looked at the code, go back to the AdvancedSorting Visualization tool and follow the details of its operation, stepping through the process of Shellsorting a 10-item array. The visualization makes it easier to see the insertion sort happening on the subarrays.

Other Interval Sequences

Picking an interval sequence is a bit of a black art. Our discussion so far has used the formula $h_{i+1} = h_i \times 3 + 1$ to generate the interval sequence, but other interval sequences have been used with varying degrees of success. The only absolute requirement is that the diminishing sequence ends with 1, so the last pass is a normal insertion sort.

Shell originally suggested an initial gap of N/2, which was simply divided in half for each pass. Thus, the descending sequence for N=100 is 50, 25, 12, 6, 3, 1. This approach has the advantage of simplicity, where both the initial gap and subsequent gaps can be found by dividing by 2. This sequence turns out *not* to be the best one. Although it's still better than the insertion sort for most data, it sometimes degenerates to $O(N^2)$ running time, which is no better than the insertion sort.

A variation of this approach is to divide each interval by 2.2 instead of 2 and truncate the result. For n=100, this leads to 45, 20, 9, 4, 1. This approach is considerably better than dividing by 2 because it avoids some worst-case circumstances that lead to $O(N^2)$ behavior. Some extra code is needed to ensure that the last value in the sequence is 1, no matter what N is. This variation gives results comparable to Knuth's sequence shown in the code.

Another possibility for a descending sequence suggested by Bryan Flamig is

```
if h < 5:
   h = 1
else:
   h = (5*h-1) // 11
```

It's generally considered important that the numbers in the interval sequence are relatively prime; that is, they have no common divisors except 1. This constraint makes it more likely that each pass will intermingle all the items sorted on the previous pass. The inefficiency of Shell's original N/2 sequence is due to its failure to adhere to this rule.

You may be able to invent a gap sequence of your own that does just as well (or possibly even better) than those shown. Whatever it is, it should be quick to calculate so as not to slow down the algorithm.

Efficiency of the Shellsort

No one so far has been able to analyze the Shellsort's average efficiency theoretically, except in special cases. Based on experiments, there are various estimates, which range from $O(N^{3/2})$ down to $O(N^{7/6})$. For Knuth's gap sequence, the worst-case performance is $O(N^{3/2})$.

Table 7-2 shows some of these estimated O() values for the Shellsort in the shaded rows, compared with the slower insertion sort and the faster quicksort. The theoretical times corresponding to various values of N are shown. Note that $N^{x/y}$ means the y^{th} root of N raised to the x power. Thus, if N is 100, $N^{3/2}$ is the square root of 100^3, which is 1,000. We used $\log_{10}$ instead of $\log_2$ because it's a little easier to show how it applies to values like 1,000 and 10,000. Also, $(\log N)^2$ means the log of N, squared. This is often written $\log^2 N$, but that's easy to confuse with $\log_2 N$, the logarithm to the base 2 of N.

Table 7-2 Estimates of Shellsort Running Time

O() Value	Type of Sort	10 Items	100 Items	1,000 Items	10,000 Items
N^2	Insertion, etc.	100	10,000	1,000,000	100,000,000
$N^{3/2}$	Shellsort	32	1,000	31,623	1,000,000
$N \times (\log N)^2$	Shellsort	10	400	9,000	160,000
$N^{5/4}$	Shellsort	18	316	5,623	100,000
$N^{7/6}$	Shellsort	14	215	3,162	46,416
$N \times \log N$	Quicksort, Timsort, etc.	10	200	3,000	40,000

For most applications, planning for the worst-case of $N^{3/2}$ is the best approach.

You might also wonder about the efficiency of the loop that finds the initial gap value, *h*. If that loop took a long time, it would affect the overall complexity of Shellsort. Note that the loop goes faster than O(N) because it has to try fewer than N values to find the initial *h*. Because O(N) is lower in complexity than $O(N \times (\log N)^2)$, $O(N^{3/2})$, and $O(N^{7/6})$, it doesn't affect the overall complexity. Because the trial *h* values grow approximately as 3^S, where S is the step in the initial loop, the total number of steps is approximately $\log_3 S$. The initial value can also be computed from the formula $h_S = (3^S - 1) / 2$ and solving it where $h_S <= N/3$.

Partitioning

Partitioning is the underlying mechanism of quicksort, which we explore next, but it's also a useful operation on its own, so we cover it here in its own section.

To **partition** data is to divide it into groups. In the case of sorting, the result is two groups, so that all the items with a key value higher than a specified amount are in one group, and all the items with a lower key value are in another.

You can easily imagine situations in which you would want to partition data. Maybe you want to divide your team members into two groups: those who are taller than 1.75 meters and those who are shorter. A medical test might need to divide cell colonies into two groups by their size from data like that shown in Figure 7-5. The relative number of colonies could indicate the presence or absence of a disorder or disease.

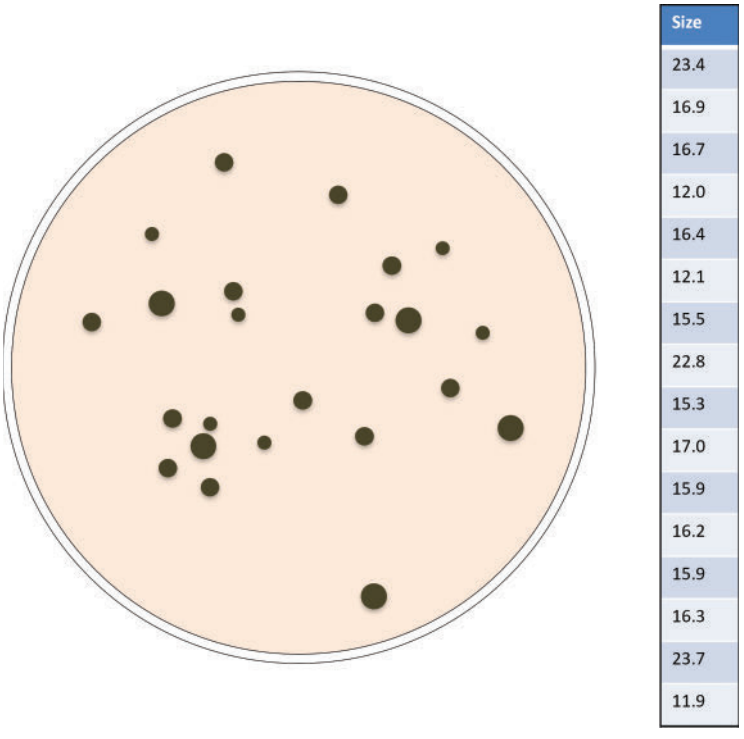


FIGURE 7-5 Partitioning cell colonies by size

The Partition Process

The AdvancedSorting Visualization tool has a Partition button that rearranges the array items into two groups. You provide the value that distinguishes the groups.

Figure 7-6 shows an example of partitioning a 10-element array. The top row shows the initial array contents. After `50` is entered in the text entry box and Partition is selected, the tool animates the process of partitioning the items. The key that was entered, 50, is called the **pivot value**. The tool illustrates that value by showing a dashed line across the

rectangles in the array, as in the second row of the figure. The pivot line is also labeled with [0, 9] to indicate that it is partitioning all the cells from cell 0 through cell 9.

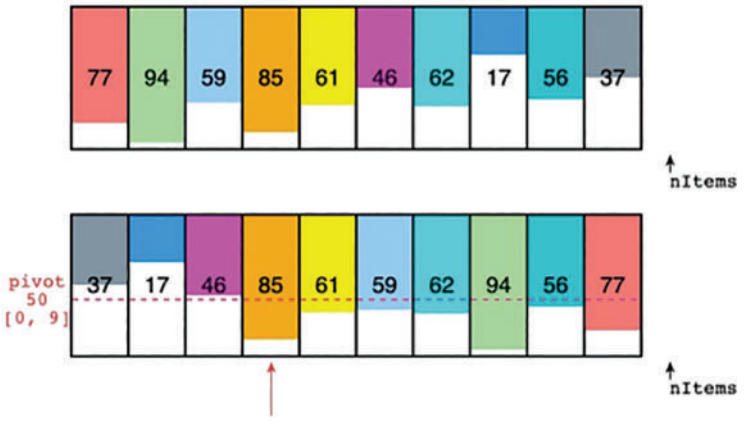


FIGURE 7-6 Partitioning a 10-element array in the AdvancedSorting Visualization tool

The animation starts with a couple of index pointers at the two ends of the range. These indices are moved toward the center, searching for array items that are not in partition order—that is, with a value below the pivot to the left and a value above the pivot to the right. When it finds such a pair, it swaps them. When the two indices eventually meet, the process is done.

In Figure 7-6, items 77 and 37 at the ends were swapped first because they are both on the wrong side of the pivot. Then the `lo` index advanced to item 94 and the `hi` index moved left, skipping 56 because it's above the pivot, and stopped at 17. Items 94 and 17 swap positions, leaving 56 in its original location. The indices move one closer to the middle and continue looking for a pair of items in the wrong partitions. The `lo` index stops at item 59, and the `hi` index stops at 46. This pair is swapped and the hunt resumes.

The 59-46 pair turns out to be the last needed to partition them all. Now the first three items of the array are all below the pivot (50), and the last seven items are above it. The tool draws an arrow pointing to item 85 (index 3) to show where the higher partition begins.

It's not easy to see where the partition will land when you look at the initial array. It depends, of course, on what pivot was chosen and only becomes obvious by going through the swap process. For example, if 90 were the pivot, only item 94 would need to be in the high partition, and the process would be done after 37 and 94 were swapped. Indeed, if you happened to choose certain pivot values, there might not be any items that need to be swapped at all. Even when that happens, however, you still need to find the lowest index of the higher partition. That is an output of this process.

If the pivot were some extreme value, like –10 or 212, then the entire array would be in the same partition. If it were too low, then the result index would be at 0; if it were too

high, it would end up at `nItems`. Let's look at how to choose the pivot value because it drastically affects the outcome.

After being partitioned, the data is by no means sorted; it has simply been divided into two groups. It's more sorted, however, than it was before. You can use that improvement to completely sort it with different partitionings, in a manner similar to the way Shellsort used *h*-sorting. Try running a couple more partitionings using different pivots in the AdvancedSorting Visualization tool. Eventually, the entire array will be sorted. When the data is sorted, partitioning can still be run, but it doesn't swap any items (except under some rare circumstances we discuss shortly).

Notice that partitioning is *not stable*. That is, each group is not in the same order it was originally. In fact, partitioning tends to reverse the order of some of the data with equal valued keys in each group. Try inserting some items with duplicate keys into the array and shuffling their positions. Can you find a pivot value that changes their ordering after a partitioning?

The General Partitioning Algorithm

How is the partitioning process carried out? We discuss it in general because it's useful outside of quicksort. The example shown in the AdvancedSorting Visualization tool is just one kind. The example, however, helps illustrate some important characteristics.

First, note that the partitioning was done within an array by rearranging the items. This means you don't need to allocate memory for another copy of all the data, which is very useful for large collections.

There must be some test that can be carried out on an item that determines the partition to which it belongs. In the general case, it could be any test such as whether the item's name contains the letter *e*. For sorting, you use a pivot value and put everything with a key value less than the pivot in one partition and everything else in the other partition. The pivot doesn't have to be numeric; it could be any value as long as two values can be compared to see if one is less than the other. The general case of using a partition test function that returns 0 for one partition and 1 for the other is sometimes useful.

As with the insertion sort, which became O(N) when applied to an array that's already sorted, it's advantageous to design the algorithm to be fast if the array is already partitioned. For that, we can have a pointer into the array where all the cells to the left are known to be in the lower partition. It would start at index 0 to indicate that none are initially known to be in that partition. Similarly, we set up a second pointer where all the cells to the right are in the higher partition. We initialize it to the last element of the array because none are known to be in that partition either. The idea is to *shrink the range* between the left and right pointers while moving items into their proper partition.

The core of the algorithm is as follows:

- Increment the left pointer until either we find a cell belonging to the upper partition, or it meets the right index.
- Decrement the right pointer until either we find a cell belonging to the lower partition, or it meets the left index.

- If the left index is at or above the right, the array is already partitioned, so we can return the left index as the end of the lower partition.
- Otherwise, swap items at the left and right indices and partition the range between left and right.

As discussed in Chapter 6, this algorithm can be set up as a recursive algorithm that works on smaller and smaller subranges of the array. Listing 7-2 shows the implementation as a recursive method of the `Array` class. Like the other sorting algorithms that work on records, you can use a `key` function to extract the key from the items in the array and partition based on that value's relationship to the `pivot` value. Unlike some other algorithms, it allows items whose key exactly matches the pivot to be in either partition. We explain the advantages of that choice after the listing.

LISTING 7-2 The Recursive Partitioning Method of the `Quicksort.py` Module

```
def identity(x): return x           # Identity function
import SortArray

class Array(SortArray.Array):       # Base new Array class on SortArray

   def partition_rec(               # Recursively partition array moving
         self,                      # items whose keys are below or equal
         pivot,                     # a pivot value to the left/low side
         lo=0,                      # the rest to the right/high side
         hi=None,                   # within the [lo, hi] range (inclusive)
         key=identity):             # Use key function to extract keys
      if hi is None:                # Default hi value is last index
         hi = len(self) - 1         # Everything above hi is in upper part
      while (lo <= hi and           # Increment lo until it goes past hi
             key(self.get(lo)) < pivot): # or we find a key that's not
         lo += 1                    # in the lower partition
      while (lo < hi and            # Decrement hi until it matches lo
             pivot < key(self.get(hi))): # or we find the pivot or
         hi -= 1                    # a key not in the upper partition
      if lo >= hi:                  # If lo is at or above hi, then the
         return lo                  # lower partition ends at lo
      self.swap(lo, hi)             # Otherwise, swap the items at lo & hi
      return self.partition_rec(    # and recursively partition remaining
         pivot, lo + 1, hi - 1, key) # items in the array
```

The `partition_rec()` method starts by filling in the default value for `hi`, the index into the array that's just before the upper partition. At the start, that's the last index into the array. The lower index, `lo`, starts at 0, indicating that no cells of the array have been found to have keys below the pivot. The cells between `lo` and `hi` need to be partitioned and form the range to be shrunk.

The first `while` loop implements the first step of the core algorithm. It increments `lo` until either `lo` > `hi` or the cell at `lo` has a key that is greater than or equal to the `pivot`, which means it belongs in the upper partition. The second `while` loop implements the second step of decrementing `hi` until it either goes past `lo` or the cell at `hi` has a key indicating it should be in the lower partition.

Note that the comparisons are slightly different in the two `while` loops. The first loop must use `lo <= hi` rather than `lo < hi` to ensure that the value at `lo` is checked against the `pivot`, and `lo` is incremented if there is only one cell in the range, that is, `lo == hi`. It's important to increment `lo` to point past `hi` if the entire array belongs in the lower partition. Using `lo < hi` in the first loop condition would mean the loop body is not entered, and `lo` could end up pointing to a cell with a key below the `pivot`. The second loop, however, uses `lo < hi` in its loop condition because when `lo == hi`, you know that the cell at that index must belong in the upper partition.

After `lo` and `hi` have been moved to narrow the range of unchecked cells, the `if` statement checks whether the range is one or fewer cells in length. If so, there's nothing to swap, and it can return `lo` as the index that is just above the lower partition. This is the base case for this recursive algorithm. If the range is bigger than one, then the values at `lo` and `hi` are swapped (using the `swap()` method defined in the `SortArray` class of Chapter 3). The recursive call partitions the range between the two swapped values.

Converting the recursive version of the partitioning algorithm to a loop version is easier than the general case. The reasons that it's simpler are

- The program doesn't do anything with the returned value of the recursive call other than return it to its caller, and
- The values of the parameters—`pivot`, `lo`, and `hi`—don't need to be saved and restored between calls.

With these two conditions, you don't need a stack of problem descriptions; you only need to update the `lo` and `hi` variables as the range of cells to check is reduced. Listing 7-3 shows the loop version of the method, `partition()`.

LISTING 7-3 The Loop-Based Partitioning Method of the `Quicksort.py` Module

```
def identity(x): return x       # Identity function
import SortArray

class Array(SortArray.Array):   # Base new Array class on SortArray

   def partition(               # Loop to partition array, moving
         self,                  # items whose keys are below or equal
         pivot,                 # a pivot value to the left/low side
         lo=0,                  # the rest to the right/high side
         hi=None,               # within the [lo, hi] range (inclusive)
         key=identity):         # Use key function to extract keys
```

```
    if hi is None:              # Default hi value is last index
       hi = len(self) - 1       # Everything above hi is in upper part
    while lo <= hi:             # Loop until no more items to inspect
       while (lo <= hi and      # Increment lo until it goes past hi
              key(self.get(lo)) < pivot): # or we find a key that's
          lo += 1               # not in the lower partition
       while (lo < hi and       # Decrement hi until it matches lo
              pivot < key(self.get(hi))): # or we find the pivot or
          hi -= 1               # a key not in the upper partition
       if lo >= hi:             # If lo is at or above hi, then the
          return lo             # lower partition ends at lo
       self.swap(lo, hi)        # Otherwise, swap the items at lo & hi
       lo, hi = lo + 1, hi - 1 # Continue partitioning in between
    return lo                   # Range to partition is now empty
```

In the loop-based method, the `lo` and `hi` parameters become local variables that are initialized at the beginning. Instead of recursive calls, a new outer `while` loop is added that iterates until the `[lo, hi]` range becomes empty. Inside that loop, the logic is nearly identical to the recursive version. The `lo` and `hi` indices move up and down like before. The next three lines of code—checking on the size of the remaining range, returning `lo` if the range is too small, and swapping the values at `lo` and `hi`—are identical to the recursive version. The difference comes at the end where `lo` and `hi` are updated (the way they would have been changed in the recursive call), and there is a return `lo` statement if the outer loop completes with an empty range.

Testing the partitioning method on a 10-element array with two different pivots shows:

```
Initialized array contains [77, 94, 59, 85, 61, 46, 62, 17, 56, 37]
Partitioning an array of size 10 around 61 returns 5
                                                V
Partitioned array contains [37, 56, 59, 17, 46, 61, 62, 85, 94, 77]

Initialized array contains [37, 56, 59, 17, 46, 61, 62, 85, 94, 77]
Partitioning an array of size 10 around 46 returns 2
                                    V
Partitioned array contains [37, 17, 59, 56, 46, 61, 62, 85, 94, 77]
```

The `V` in the output is pointing to the index of the first cell in the upper partition, that is, the index returned by the `partition()` method. You can see that the partitions are successful: the numbers to the left of the returned index are all smaller than the pivot values of 61 and 46, respectively. Note that the sizes of the partitions depend on the choice of pivot. Only those array cells that had to be swapped were affected, for example, only the values of 17 and 56 in the second example that pivoted around 46. The first partition had already swapped 77 and 37, 94 and 56, 85 and 17, and 61 and 46.

Equal Keys

Somewhat surprisingly, it doesn't matter if items whose key matches the pivot go in the lower or higher partition. The pivot value is the dividing line between the two partitions. Items that have a key equal to the pivot could be considered to belong to either one.

In the `partition()` method in Listing 7-3, both loops that increment the `lo` and decrement the `hi` indices stop when they find an item with the pivot as its key (in addition to finding an item belonging in the opposite partition). If the indices differ after going through both loops, then the items they index are swapped. If both items have a key equal to the pivot, there is no need to swap them. So, shouldn't the algorithm skip the swap if it finds equal keys? The answer is not so straightforward.

To add such a test would put another comparison in the loop that would be run once for every iteration of the outer loop. It would save the expense of a swap, but only if the data has keys of equal value, *and both* are equal to the pivot. That's not likely to happen, in general. Furthermore, it would save significant time only if the cost of swapping two items in the array were much more than comparing the keys.

There's another, more subtle reason to swap the items even when one or both has a key equal to the pivot. If the algorithm always puts items whose keys match the pivot in one partition, say the higher one, then it can decrement the `hi` index past those items as it looks for items to swap. Doing so moves the eventual partition index to the lowest possible value and minimizes swaps. As discussed later in the section on quicksort, it's good for the partition indices to end up in the middle of the array and very bad for them to end up at the ends. The idea is somewhat similar to binary search: dividing the range in half is most efficient because it limits the maximum size of the remaining ranges to search.

Efficiency of the Partition Algorithm

The partition algorithm runs in O(N) time. It's easy to see why this is so when running the Partition operation in the Visualization tool: the two pointers start at opposite ends of the array and move toward each other, stopping and swapping as they go. When they meet, the partition is complete. Each cell of the array is visited at most one time, either by the `lo` or the `hi` pointer. If there were twice as many items to partition, the pointers would move at the same rate, but they would have twice as many items to compare and swap, so the process would take twice as long. Thus, the running time is proportional to N.

More specifically, partitioning an N-cell array makes exactly N comparisons between keys and the pivot value. You can see that by looking at the code where the `key()` function is called and its value compared with the pivot. There's one test with the item at the `lo` pointer and one with the `hi` pointer. Because `lo` and `hi` are checked prior to those comparisons, you know that either `lo` < `hi` or that the comparison with the pivot and the key at `hi` doesn't happen. The `lo` and `hi` values range over all N cells.

The `lo` and `hi` pointers are compared with each other N + 2 times because that comparison must succeed for each of the pivot comparisons to happen, and they must each fail when they find either a pair to swap or find each other. The number of comparisons is independent of how the data is arranged (except for the uncertainty between one or two extra comparisons at the end of the process).

The number of swaps, however, does depend on how the input data is arranged. If it's inversely ordered, and the pivot value divides the items in half, then every pair of values must be swapped, which is N/2 swaps.

For random data, there are fewer than N/2 swaps in a partition, even if the pivot value is such that half the items are shorter and half are taller. The reason is that some items are already in the right place (short bars on the left, tall bars on the right in the visualization). If the pivot value is higher (or lower) than most of the items, there will be even fewer swaps because only those few items that are higher (or lower) than the pivot will need to be swapped. On average, for random data with random pivots, about half the maximum number of swaps take place.

Although there are fewer swaps than comparisons, they are both proportional to N. Thus, the whole partitioning process runs in O(N) time.

Quicksort

Quicksort is a popular sorting algorithm, and for good reason: in the majority of situations, it's the fastest, operating in O(N×log N) time, and only needs O(log N) extra memory. C. A. R. (Tony) Hoare discovered quicksort in 1962.

To understand quicksort, you should be familiar with the partitioning algorithm described in the preceding section. Basically, the quicksort algorithm operates by partitioning an array into two subarrays and then calling itself recursively to quicksort each of these subarrays. There are some embellishments, however, to be made to this basic scheme. They have to do with the selection of the pivot and the sorting of small partitions. We examine these refinements after we look at a simple version of the main algorithm.

It's difficult to understand *what* quicksort is doing before you understand *how* it does it, so we reverse our usual presentation and show the Python code for quicksort before presenting the visualization tool.

The Basic Quicksort Algorithm

The code for a basic recursive quicksort method is fairly simple. Listing 7-4 shows a sketch of the algorithm.

LISTING 7-4 A Sketch of the Quicksort Algorithm in Python

```
def quicksort_sketch(       # Sort items in an array between lo
      self, lo=0, hi=None,  # and hi indices using Hoare's
      key=identity):        # quicksort algorithm on the keys
   if hi is None:           # Fill in hi value if not specified
      hi = len(self) - 1
   if lo >= hi:             # If range has 1 or fewer cells,
      return                # then no sorting is needed

   pivot = self.choosePivot(lo, hi) # Choose a pivot

   hipart = self.partition(  # Partition array around the key
      key(pivot),            # of the item at the pivot index and
      lo, hi, key)           # record where high part starts

   self.quicksort_sketch(lo, hipart - 1, key) # Sort lower part
   self.quicksort_sketch(hipart, hi, key) # Sort higher part
```

As you can see, there are five basic steps:

1. Check the base case and return if the [sub]array is small enough.
2. Choose a pivot.
3. Partition the subarray into lower and higher parts around the pivot.
4. Make a recursive call to sort the lower part.
5. Make another recursive call to sort the higher part.

The first lines should look familiar from other recursive routines. Like the `partition_rec()` method in Listing 7-2, the `lo` and `hi` indices are the leftmost and rightmost cells of the array to be sorted. If not provided by the caller, they are set to be the first and last cells of the full array. Testing for the base case involves seeing whether there are two or more cells to be sorted. If there are one or zero, then no sorting is needed, and the method returns immediately.

The sketch shows a call to `choosePivot()` to select a `pivot` item in the subarray. We return to that choice in a moment, but for now, assume that it randomly chooses one of the items between the `lo` and `hi` cells. After the partition, all the items in the left subarray, below `hipart`, are smaller than or equal to all those on the right. If you then sort the left subarray and the right subarray, the entire array will be sorted. How do you sort these subarrays? By calling this very method recursively.

When `quicksort_sketch()` is called on something other than the base case, the algorithm chooses a pivot value and calls the `partition()` method, described in the preceding section, to partition it. This method returns the index number of the first cell in the higher partition. That index gets stored in the `hipart` variable and is used to determine the subarrays to be recursively sorted. The situation is shown in Figure 7-7 with a "randomly" chosen pivot of 38.

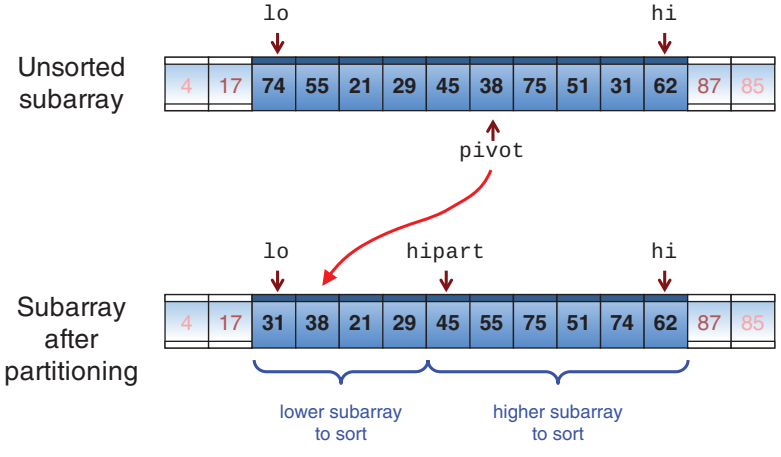


FIGURE 7-7 Partitioning in one call to `quicksort_sketch()`

Note that the pivot doesn't necessarily end up being moved to the beginning or end of its partition. Nor does it always stay where it started. After the subarray is partitioned, `quicksort_sketch()` calls itself recursively, once for the lower part of its array, from `lo` to `hipart - 1`, and once for the higher part, from `hipart` to `hi`. These calls move the pivot item (and all other items) into their sorted positions.

Choosing a Pivot Value

What pivot value should the quicksort algorithm use? Here are some relevant ideas:

- The pivot value should be the key value of an actual data item; this item is also called the *pivot*. At a minimum, avoiding an extreme value outside the range of keys prevents making empty partitions, but it also allows for some other optimizations.
- You can pick a data item to be the pivot somewhat arbitrarily. For simplicity, you could always pick the item on the right end of the subarray being partitioned.
- After the partition, if the pivot is inserted at the boundary between the lower and upper subarrays, it will be in its final sorted position.

This last point may sound unlikely, but remember that, because the pivot's key value is used to partition the array, following the call to `partition()` the lower subarray holds items with keys equal to or smaller than the pivot, and the right subarray holds items with keys equal to or larger. If the pivot could somehow be placed between these two subarrays, it would be in the correct place—that is, in its final sorted position. Figure 7-8 shows how this looks with the pivot example value of 38.

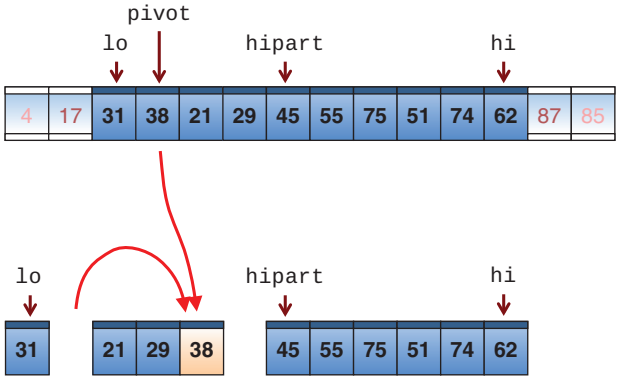


FIGURE 7-8 Moving the pivot between the subarrays

This figure is somewhat fanciful because you can't actually slice up the array as shown, at least not without copying a lot of cells. So how do you move the pivot to its proper place?

You could shift all the items in the left subarray to the left one cell to make room for the pivot. This approach is inefficient, however, and unnecessary. You can select the pivot to be any item from the array. If you select the rightmost one and partition everything to the

left of it, then you can simply swap the pivot with the item indexed by `hipart`—the leftmost of the higher part as shown in Figure 7-9. In fact, doing so brings the array one step closer to being fully sorted. This swap places the pivot in its proper position between the left and right groups. The 75 is switched to where the pivot was, and because it remains in the right (higher) group, the partitioning is undisturbed. Note that in choosing the rightmost cell of the subarray—the one at the `hi` index—it must be excluded from the range that the `partition()` method reorganizes because it shouldn't be swapped for any other item.

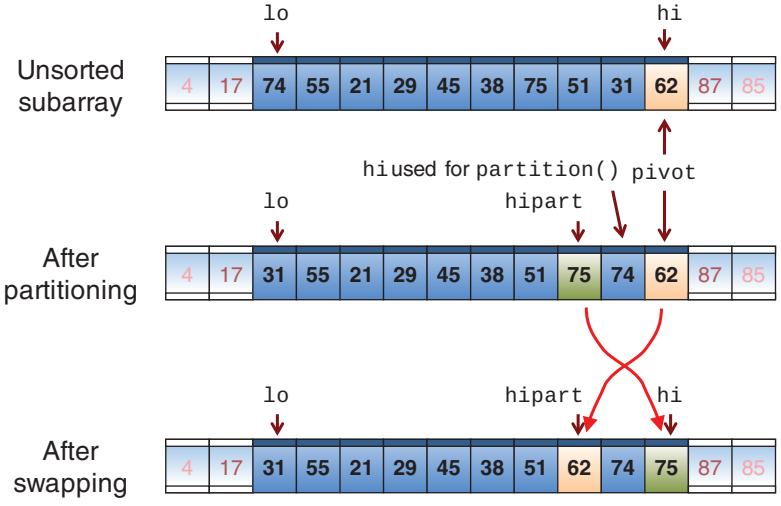


FIGURE 7-9 Swapping the pivot from the right

Similarly, the pivot could be chosen as the leftmost cell and then swapped into position just to the left of `hipart`. In either case, the swap might not be needed if the partitioning left no cells in the higher or lower partition, respectively. Because the pivot can be placed in either partition, these methods can be used to guarantee that there will be at least one cell in the higher or lower partition, respectively.

Swapping the pivot with the value at the start of the higher partition ensures the pivot ends up in its final resting place. All subsequent activity will take place on one side of it or on the other, but the pivot itself won't be moved again. In fact, you can exclude it from the subrange in the recursive call to sort the higher range—for example:

```
self.quicksort_sketch(lo, hipart - 1, key) # Sort lower part
self.quicksort_sketch(hipart + 1, hi, key) # Sort higher part
```

By removing that one cell from subsequent processing, the algorithm guarantees that the subranges are always diminishing. To see that, think about the range for `hipart`. It could have any value from `lo` to `hi` (but not `hi + 1` because you chose the pivot to be equal to the key at `hi`). At those extreme values, one of the recursive subranges will be empty while

the other will have one fewer cell than the original `lo` to `hi` range—either [`lo, hi - 1`] or [`lo + 1, hi`].

We can make an optimization to the partitioning algorithm by taking advantage of the constraint that there will always be at least one item that belongs in the high partition. That constraint means the test for the end of the first loop—the one that advances `lo` to find a higher partition item—can be simplified. It no longer needs to check that `lo <= hi` because the second part of the test checking the key at `lo` versus the pivot must fail when `lo == hi`. Although this comparison test might seem trivial, it occurs in the innermost loop of what could be a very long calculation, so saving a few operations here can have a large impact. There are a few more improvements in the full implementation in the following sections.

A First Quicksort Implementation

To flesh out the embellishments to the sketch shown earlier, we provide a working version, called the `qsort()` method, in Listing 7-5. This method captures the algorithm just described, along with its revised partitioning method.

LISTING 7-5 The `qsort()` Method with Improved Partitioning

```
def identity(x): return x          # Identity function
import SortArray

class Array(SortArray.Array):      # Base new Array class on SortArray
   def __partition(                # Private function partitions array by
         self,                     # items whose keys are below or equal
         pivot,                    # a pivot value to the left/low side
         lo,                       # the rest to the right/high side
         hi,                       # within the [lo, hi] range (inclusive)
         key=identity):            # knowing at least one key == pivot
      while lo <= hi:              # Loop until no more items to inspect
         while (key(self.get(lo)) # Increment lo until we find a key
                < pivot):          # that's not in the lower partition
            lo += 1                # Knowing pivot == one key in [lo,hi+1]
         while (lo < hi and        # Decrement hi until it matches lo
                pivot < key(self.get(hi))): # or we find the pivot or
            hi -= 1                # a key not in the upper partition
         if lo >= hi:              # If lo is at or above hi, then the
            return lo              # lower partition ends at lo
         self.swap(lo, hi)         # Otherwise, swap the items at lo & hi
         lo, hi = lo + 1, hi - 1   # Continue partitioning in between
      return lo                    # Range to partition is now empty

   def qsort(                      # Sort items in an array between lo
         self, lo=0, hi=None,      # and hi indices using Hoare's
         key=identity):            # quicksort algorithm on the keys
```

```
if hi is None:              # Fill in hi value if not specified
   hi = len(self) - 1
if lo >= hi:                # If range has 1 or fewer cells,
   return                   # then no sorting is needed
pivot_i = hi                # Choose pivot index to be rightmost
pivotItem = self.get(pivot_i) # Get item at pivot index
hipart = self.__partition(  # Partition array around the key
   key(pivotItem),          # of the item at the pivot index and
   lo, hi - 1, key)         # record where high part starts
if hipart < pivot_i:        # If pivot index is above high
   self.swap(               # part start, then swap pivot item
      hipart, pivot_i)      # with high part start
self.qsort(lo, hipart - 1, key) # Sort lower part
self.qsort(hipart + 1, hi, key) # Sort higher part
```

Because the partitioning method now relies on the pivot value always being a key of one of the cells between `lo` and `hi`, we have made it a private method, `__partition()`, so that it cannot be called by the users of this `Array` class using some other pivot value. The enhancement comes in the second `while` loop of `__partition()`, by getting rid of the test for `lo <= hi`. It still needs the test for `lo < hi` in the third `while` loop because there is no guarantee that it will find an item in the lower partition before running in to `lo`. If the pivot value was the lowest key in the whole array and distinct, all the items would satisfy the `pivot < key(self.get(hi))` test. Because this is a private method that is only called by `qsort()`, we can also remove the default values for the `lo` and `hi` parameters since the `qsort()` method always provides them.

The `qsort()` method remains recursive (although it could be converted to a loop-based algorithm as you've seen). It tracks both the index to the pivot item and the item itself in different variables, `pivot_i` and `pivotItem`. It replaces the call to `choosePivot()` with a simple assignment to improve efficiency slightly. Because the pivot is the rightmost item, it calls the private `__partition()` method with the range `[lo, hi - 1]` to prevent the pivot from being swapped. After partitioning, it swaps the pivot item and the item at the start of the higher partition. That swap needs to be done only if the pivot index is higher than the first index of that part, `hipart`. The method ends with the recursive calls on the lower and higher parts, excluding the pivot item that was moved to the cell at `hipart`.

You can add a `print` statement to the `qsort()` method right after the partitioning to see what's happening. By adding the line

```
print('Partitioning', lo, 'to', hi, 'leaves', self)
```

you can see the ranges being partitioned and the changes in the values of the array. Placing this `print` statement after the partitioning and swapping of the pivot are done (just before the recursive calls to sort the lower and higher parts), along with an initial printing of the array contents, produces the output in Listing 7-6.

LISTING 7-6 Output of `qsort()` with a `print` Statement

```
Quicksorting [77, 94, 59, 85, 61, 46, 62, 17, 56, 37] :
Partitioning 0 to 9 leaves [17, 37, 59, 85, 61, 46, 62, 77, 56, 94]
Partitioning 2 to 9 leaves [17, 37, 59, 85, 61, 46, 62, 77, 56, 94]
Partitioning 2 to 8 leaves [17, 37, 46, 56, 61, 59, 62, 77, 85, 94]
Partitioning 4 to 8 leaves [17, 37, 46, 56, 61, 59, 62, 77, 85, 94]
Partitioning 4 to 7 leaves [17, 37, 46, 56, 61, 59, 62, 77, 85, 94]
Partitioning 4 to 6 leaves [17, 37, 46, 56, 61, 59, 62, 77, 85, 94]
Partitioning 4 to 5 leaves [17, 37, 46, 56, 59, 61, 62, 77, 85, 94]
```

The first call to the `qsort()` method operates on the full array, the range [0, 9]. The pivot is the rightmost item, whose value is 37. The only item lower than that is item 17 at index 7. The partitioning swaps the 77 from the leftmost cell with the 17 and then swaps the pivot value 37 with the item at index 1 (the value of `hipart` after partitioning), which is item 94. Now the array has items equal to and below 37 in the range [0, 0] and the items equal to and higher in the range [1, 9] (the `hipart` starts at index 1 and includes the pivot).

The recursive call to sort the lower range ends up being a base case requiring no action. The call to sort the higher range [2, 9] is the next output line. Note that the pivot from the first call, 37, has been removed from the range because it's known to be in the correct position, index 1. In the range [2, 9], the pivot is 94. Because that is the maximum value of the range, the partitioning makes no changes. The pivot remains at the right end, and the lower partition is sorted next.

The third `Partitioning` output line shows sorting the range below the pivot, 94, which spans [2, 8]. This time the pivot is 56. There is one item less than the pivot, and that is 46 at index 5. That value gets swapped with the leftmost value, 59, at index 2. Then the pivot is swapped with the leftmost item of the higher partition, 85. That leaves the lower partition as [2, 2] and the higher partition as [3, 8].

The fourth `Partitioning` output line in Listing 7-6 shows sorting the higher range [4, 8], ignoring the pivot of the previous call. The pivot at index 8 is 85; the value that was swapped with the pivot of the previous call. Because 85 is the maximum value of that range, no swaps are made, and only the lower partition is a non-base case.

The fifth `Partitioning` output line sorts the lower partition range, [4, 7]. The pivot is 77 for that range, which again is a maximum, so no swaps are made. The sixth output line sorts the next lower partition range [4, 6]. Here the pivot is 62, and again it's the maximum of that range, so it's on to sort the lower partition.

The seventh call works on the lower partition of the sixth call, the range [4, 5]. The pivot for that range is 59, which is less than the only other item, 61. That pivot gets swapped with the 61 because that is where `hipart` starts. Because the range has only two cells, the recursive calls are base cases that make no more changes.

Was that approach efficient? The particular array processed in Listing 7-6 generated seven calls to the `__partition()` method. That's quite a few when compared to the 10 elements

in the array. This is close to the worst-case behavior, which we analyze a little later. In general, however, the pivots are likely to split long ranges in approximately equal subranges, making the algorithm very fast.

Running Quicksort in the AdvancedSorting Visualization Tool

At this point you know enough about the quicksort algorithm to run some examples in the Visualization tool. The code in the Visualization tool runs a version of the algorithm with some optimizations that we haven't discussed yet. The first thing to do is to ensure the check box labeled "Use median of 3" below the Quicksort button is unchecked. We get to that optimization a little later, along with some others.

With the Median of 3 option turned off, run the quicksort on an array. Figure 7-10 shows the final result after the quicksort finishes. As the quicksort process runs, you see the partition lines appear as it processes the subarrays. The first call to quicksort runs on the full range of the array, cells 0 through 9 in the 10-element array of Figure 7-10. If you look carefully, you can see that it chooses the rightmost item in each subarray as the pivot. The pivot line for each partition shows during the partitioning but then disappears.

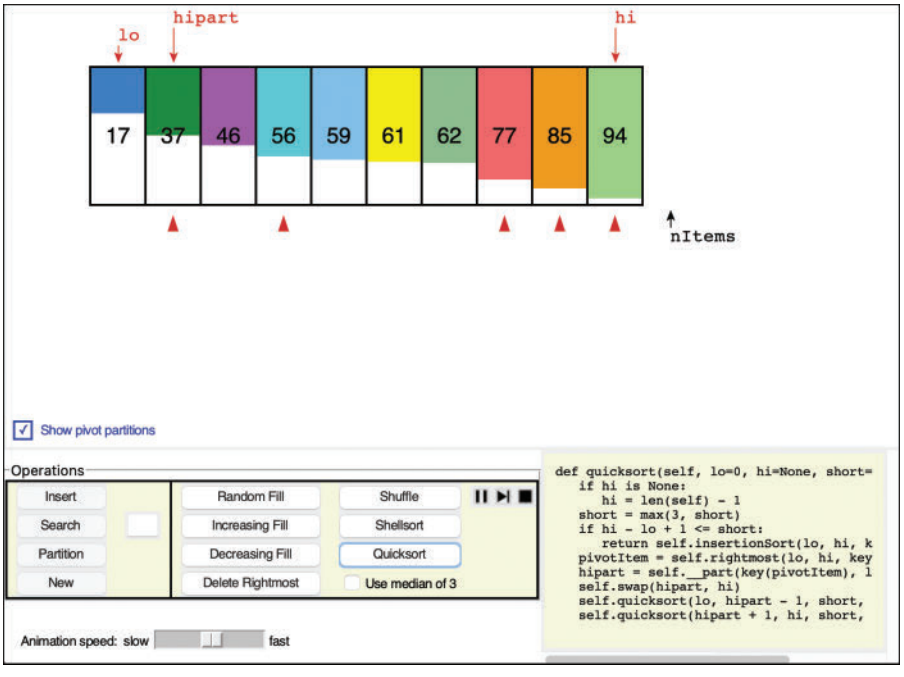


FIGURE 7-10 The result of quicksort in the AdvancedSorting Visualization tool

At the end of each partitioning, a triangle is placed below the first cell of the higher partition, similar to the arrow that the Partition button left. These triangles remain after the sort is complete to show where each partition landed.

In Figure 7-10, the last item of the initial array, 37, was the first pivot value, and led to placing the triangle below cell 1. Only item 17 is lower, and the other nine items were placed in the higher partition. The quicksort runs the partitioning four more times, placing the other triangles to show where it split the subarrays. You can hide or show the triangles by selecting the check box labeled "Show pivot partitions."

As you watch the animation of the quicksort process, you can easily see how the ranges are split at the `hipart` index, followed by processing of the left and right partitions. When the size of partitions grows small, it does some special processing that we describe a little later.

One of the key things to watch is how many partitions are made. The number of triangles showing at the end provides the count. You want to make as few partitions as possible to make the sorting process efficient. Try making a larger array, filled with random values, and quicksorting it (with the Median of 3 option still turned off). Figure 7-11 shows an example with a 77-element array that led to 29 partitions.

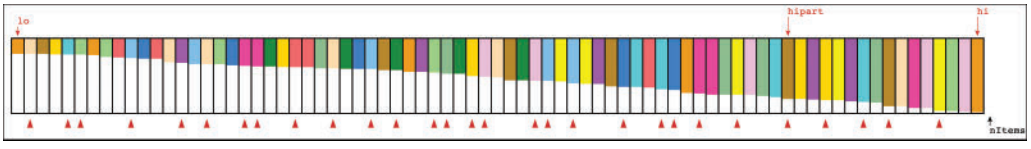


FIGURE 7-11 Running quicksort on a 77-element array choosing the rightmost as pivot

While you watch the sorting animation, compare it to what you saw with the Shellsort. Does it appear to be generally "more sorted" with each partitioning?

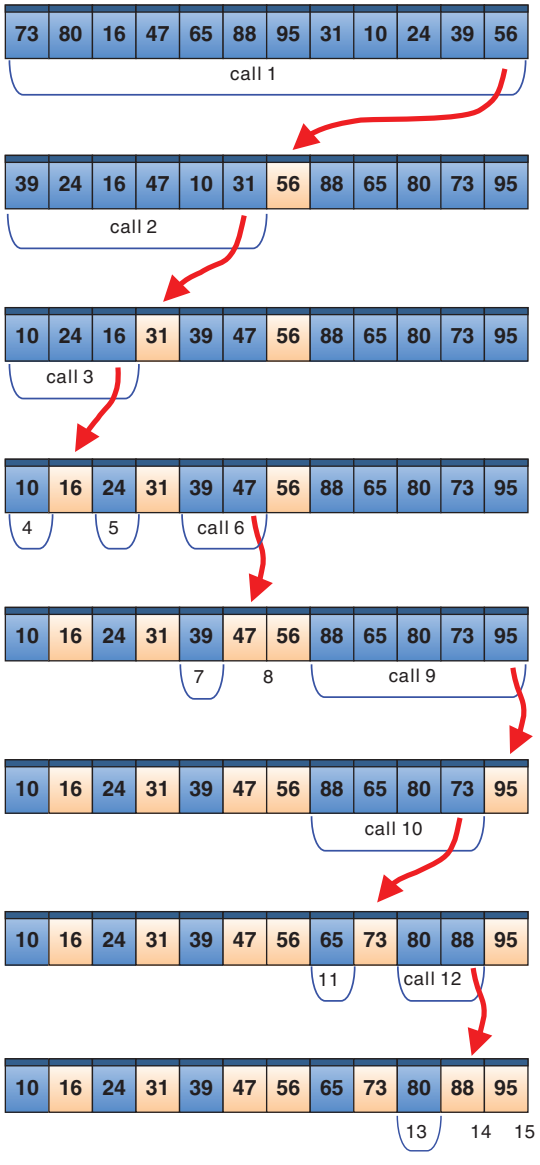
The Details

Figure 7-12 shows all the steps involved in sorting a 12-element array using `qsort()`. The horizontal brackets under the arrays show which subarray is being partitioned at each recursive call, along with a call number. Some calls are on single element or empty ranges; these are the base cases that return immediately and have no arrow indicating a pivot was chosen. The pivots are highlighted as they are placed in their final positions.

Sometimes, as in calls 6, 9, and 12, the pivot ends up in its original position on the right side of the subarray being sorted. In this situation, after sorting the subarray to the left of the pivot, the subarray to the right doesn't need to be sorted because it is empty. There still is a call to the `qsort()` method, as indicated by the call numbers in the figure that have no bracketed range—calls 8, 14, and 15.

The different calls in Figure 7-12 occur at different levels (or depths) of recursion, as shown in the table on the right. The first call to `qsort()` is at the first level. It makes two recursive calls to sort the subarrays to the left and the right of its pivot, 56. Those calls are at level 2, and they make calls for the lower and higher partitions that they create.

The level 1 call to sort its lower partition is call 2 and covers indices 0 through 5. The call to sort its higher partition ends up being call 9 on indices 7 through 11. The intervening calls are all the deeper recursive calls to finish sorting the lower partition.



Call	Recursion Level
1	1
2, 9	2
3, 6, 10, 15	3
4, 5, 7, 8, 11, 12	4
13, 14	5

FIGURE 7-12 The qsort process

The order in which the partitions are created, corresponding to the call numbers, does not correspond with depth. It's not the case that all the first-level partitions are done first, then all the second-level ones, and so on. Instead, the left group at every level is handled before any of the right groups.

The number of levels in the table shows that with 12 data items, the machine stack needs enough space for five sets of arguments and return values—one for each recursion level. This is, as you see later, somewhat greater than the logarithm to the base 2 of the number of items: $\log_2 N$. The size limit of the machine stack varies between systems. Sorting very large numbers of data items using recursive procedures may cause this stack to overflow.

Degenerates to O(N²) Performance

Try the following example using the AdvancedSorting Visualization tool. Make an empty array of 10 to 20 cells and use the Increasing Fill button to fill them with a sequence of increasing keys. Then run the quicksort with the Median of 3 option turned off. You'll see that it seems to create more triangles for the partitions. A 10-item array creates 7 triangles, and a 20-item array creates 17. What's happening here?

The problem is in the selection of the pivot. Ideally, the pivot should be the median of the items being sorted. That is, half the items should be larger than the pivot, and half smaller. That choice would result in the array being partitioned into two subarrays of equal size. Having two equal subarrays is the optimum situation for the quicksort algorithm. If it sorts one large and one small array, the quicksort is less efficient because the larger subarray must be subdivided more times. That also requires more recursive levels.

The worst situation results when a subarray with N cells is divided into one subarray with 1 cell and the other with N–1 cells. This division into 1 cell and N–1 cells can be seen in calls 6, 9, and 12 of Figure 7-12. In the visualization tool, it happens in every partition (except that three cells don't get partitioned, which we explain in a moment).

If this 1 and N–1 division happens with every partition, then every cell requires a separate partition step. This is, in fact, what takes place when the input data is already sorted (or inversely sorted). In all the subarrays, the pivot is the largest (or smallest) item. If it's the largest item, the partitions are of size N–1 and 1, the pivot (assuming no duplicate keys). If the pivot is the smallest item, all the cells go in the larger partition, and the recursive calls are made on an empty subarray and one of size N–1, and the pivot must be swapped into final position (leading to the largest remaining item being the pivot on the next call).

As you can see in the Visualization tool, the triangles for the pivot points are next to one another, which means one partition ended up being a single cell. In this situation, the advantage gained by the partitioning process is lost, and the performance of the algorithm degenerates to $O(N^2)$.

Besides being slow, there's another potential problem when quicksort operates in $O(N^2)$ time. When the number of partitions increases, the number of recursive method calls also increases. In the ideal case, the number of method calls is $O(\log_2 N)$, but in this worst

situation, it becomes O(N). Every function or method call takes up room on the machine stack. If there are too many calls, the machine stack may overflow and paralyze the system. Even if you convert the recursive procedure to an explicit stack approach, O(N) memory will be consumed to hold the information needed to correctly process the subranges.

To summarize: the `qsort()` method chooses the rightmost item's key as the pivot. If the data is truly random, this choice isn't too bad because usually the pivot isn't too close to either extreme value of the keys. When the input data is (inversely) sorted, however, choosing the pivot from one end or the other is a bad idea. How can we improve on this approach to selecting the pivot?

Median-of-Three Partitioning

Many schemes have been devised for picking a better pivot. The method of choosing an item at random is simple, but—as you've seen—doesn't always result in a good selection. Choosing items with keys at or near the extreme ends of the range of keys leads to unbalanced partitions. Alternatively, you could examine all the items and actually calculate which of their keys was the median. This pivot choice would be ideal, but doing so isn't practical because it could take more time than the sort itself.

A compromise solution is to examine the first, last, and middle items of the subarray, and use the median of their keys for the pivot. Picking the median of the first, last, and middle elements is called the **median-of-three** approach and is shown in Figure 7-13.

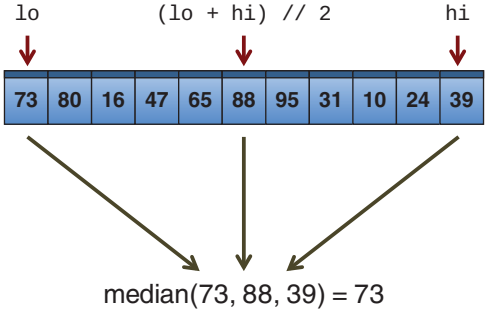


FIGURE 7-13 The median of three

Finding the median of three items is obviously much faster than finding the median of all the items, and yet it successfully avoids picking the largest or smallest item in cases where the input data is already sorted or inversely sorted. There are probably some pathological arrangements of data where the median-of-three scheme works poorly, but normally it's a fast and effective technique for finding the pivot.

Besides picking the pivot more effectively, the median-of-three approach has an additional benefit: you can dispense with the `lo < hi` test in the second inner `while` loop, leading to a small increase in the algorithm's speed. How is this possible?

You can eliminate this test because you can use the median-of-three approach to not only select the pivot but also to sort the three elements used in the selection process. Figure 7-14 shows this operation.

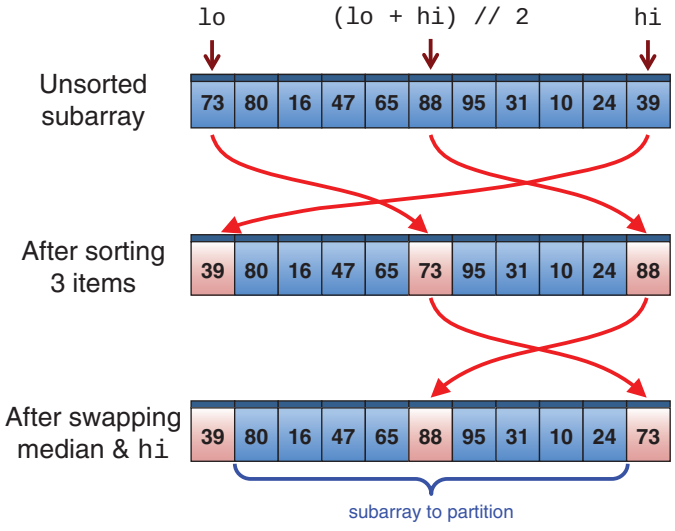


FIGURE 7-14 Sorting the left, center, and right elements

When these three elements are sorted, and the median item is selected as the pivot, you are guaranteed that the element at the left end of the subarray is less than (or equal to) the pivot, and the element at the right end is greater than (or equal to) the pivot. This means that the `lo` and `hi` indices can't step beyond the right or left ends of the array, respectively, even if you remove the `lo < hi` test (and the `lo <= hi` test that was previously removed from the first inner loop).

Removing that test may not seem like a wise idea, especially with the swaps that happen during the partitioning algorithm. On closer inspection, however, those swaps only move lower items to the left and higher items to the right, so there will always be a key that causes the two loops to stop. These items are called **sentinels** because they guard against going out of bounds. The `hi` index will stop decrementing, thinking it needs to swap the item, only to find that it may have already crossed the `lo` index and the partition is complete.

Another small benefit to median-of-three partitioning is that after the left, middle, and right elements are sorted, and the median is swapped with the rightmost item, the partition process doesn't need to examine the lowest and median elements again. The partition can begin at `lo+1` and `hi-1` as shown in the last row of Figure 7-14 because `lo` and `hi` have, in effect, already been partitioned. You know that `lo` is in the correct partition because it's on the left and it's less than or equal to the pivot, and `hi` is in the correct place because it's on the right and it is the pivot. Neither one may be in its final, sorted position, but all that matters is that they are partitioned correctly at this point.

Thus, median-of-three partitioning not only avoids $O(N^2)$ performance for already-sorted data but also allows you to speed up the inner loops of the partitioning algorithm and slightly reduces the number of items that must be partitioned.

Handling Small Partitions

Using the median-of-three pivot selection method, it follows that the quicksort algorithm won't work for partitions with fewer than three items. With exactly three items, the median-of-three would fully sort the items and then perform two unnecessary swaps of the median and high items. Performing these extra swaps seems wasteful, so finding another method for processing three or fewer items seems appropriate. The number 3 in this case is called a **cutoff** point. What is the best way to process these small subarrays? Would it be wise to apply the nonmedian choice of pivot as we did before in `qsort()`? Are there other sorting methods that might do better with small subarrays?

Using an Insertion Sort for Small Partitions

One option for dealing with small partitions is to use the insertion sort. That approach works for any size subarray and has the added benefit of being O(N) for already-sorted subarrays. In fact, you aren't restricted to a cutoff of 3. You could set the cutoff to 10, 20, or any other number 3 or higher, expecting to find some optimum value. It's interesting to experiment with different values of the cutoff to see where the best performance lies. Knuth (see Appendix B) recommended a cutoff of 9. The optimum number is usually more than 3 and depends on a variety of factors: the kinds of input data distributions that happen frequently, the computer, the operating system, the compiler (or interpreter), and so on.

The Full Quicksort Implementation

Combining all these improvements into the implementation results in the full `quicksort()` method shown in Listing 7-7. It has its own private partitioning method, `__part()`, that includes the optimizations in the loop conditions based on the sentinel values from the median-of-three algorithm.

LISTING 7-7 The full `quicksort()` Method for Sorting Arrays

```
def identity(x): return x      # Identity function
import SortArray

class Array(SortArray.Array):  # Base new Array class on SortArray
   def __part(                 # Private function partitions array by
         self,                 # items whose keys are below or equal
         pivot,                # a pivot value to the left/low side
         lo,                   # the rest to the right/high side
         hi,                   # within [lo, hi] knowing there is 1
         key=identity):        # key below pivot & pivot at hi+1
```

```
        while lo <= hi:             # Loop until no more items to inspect
            while (key(self.get(lo)) # Increment lo until we find a key
                   < pivot):         # that's not in the lower partition
                lo += 1               # Knowing pivot == one key in [lo,hi+1]
            while (pivot <            # Decrement hi until it points to key
                   key(self.get(hi))): # in lower partition
                hi -= 1               # a key not in the upper partition
            if lo >= hi:              # If lo is at or above hi, then the
                return lo             # lower partition ends at lo
            self.swap(lo, hi)         # Otherwise, swap the items at lo & hi
            lo, hi = lo + 1, hi - 1   # Continue partitioning in between
        return lo                     # Range to partition is now empty

    def quicksort(                  # Sort items in an array between lo
            self,                   # and hi indices using Hoare's
            lo=0,                   # quicksort algorithm. For short
            hi=None,                # subarrays, use insertion sort.
            short=3,                # Short must be 3 or more to enable
            key=identity):          # median of three choice of pivot
        if hi is None:              # Fill in hi value if not specified
            hi = len(self) - 1      # as last item in array
        short = max(3, short)       # Enforce short limit >= 3
        if hi - lo + 1 <= short:    # If subarray is short, then use
            return self.insertionSort(lo, hi, key) # insertion sort
        pivotItem = self.medianOfThree( # Else find median key of lo,
            lo, hi, key)            # mid, hi and place item at hi index
        hipart = self.__part(       # Partition array around the key of
            key(pivotItem),         # the pivot item and
            lo + 1, hi - 1, key)    # record where high part starts
        self.swap(hipart, hi)       # Swap pivot with high part start
        self.quicksort(lo, hipart - 1, short, key) # Sort lower part
        self.quicksort(hipart + 1, hi, short, key) # Sort higher part
```

The `quicksort()` method has a new, optional parameter, `short`, which determines the cutoff for using insertion sort on short subarrays. It defaults to 3 but can be set higher. (The AdvancedSorting Visualization tool uses the default value of 3.) The base case test is modified to look for subarrays that have `short` or fewer cells in them. They are processed by the `insertionSort()` method shown in Listing 7-8.

If you compare these two methods to their counterparts in the earlier implementation shown in Listing 7-5, they are similar. The small changes do not reduce (or significantly increase) the complexity of the program. They do, however, result in a big impact on the performance.

Listing 7-8 shows the helper methods used by `quicksort()`. The `medianOfThree()` method performs a three-element sort at the low, middle, and high indices of the subarray. Because the number of items to sort is known and small, it's better to write a few `if`

statements to swap values than to make a loop-based sorting routine. After computing the index of the middle item, `mid`, the first two `if` statements swap items to place the item with the lowest key at the `lo` position. Normally for a sort, the next step would be to compare the other two items and ensure the item with the highest key is placed at `hi`. Because this method is preparing the data for partitioning, however, the item with the middle key is placed at `hi`. It returns that item because it is the median of the three keys.

LISTING 7-8 The Helper Methods for `quicksort()`

```
  def medianOfThree(           # Find median of lo, middle, and hi
        self, lo, hi,          # keys in subarray and put median
        key=identity):         # in hi position for partition
     mid = (lo + hi) // 2      # Compute middle index
     if key(self.get(lo)) > key(self.get(mid)): # Compare 1st pair
        self.swap(lo, mid)     # of keys and swap if lo > mid
     if key(self.get(lo)) > key(self.get(hi)): # Compare 2nd pair
        self.swap(lo, hi)      # of keys and swap if hi is lowest
     # At this point lo has the minimum of the 3 keys
     if key(self.get(hi)) > key(self.get(mid)): # Compare 3rd pair
        self.swap(hi, mid)     # of keys again and swap if hi > mid
     return self.get(hi)       # Return item with median key (@ hi)

  def insertionSort(           # Sort subarray by repeated inserts
        self,                  # This insertion sort will be used
        lo=0,                  # on small subarrays by quicksort
        hi=None,
        key=identity):
     if hi is None:            # Fill in hi value if not specified
        hi = len(self) - 1     # as last item in array
     for outer in range(lo + 1, hi + 1): # Mark one item
        temp = self.get(outer) # Store marked item in temp
        temp_key = key(temp)
        inner = outer          # Inner loop starts at mark at right
        while (inner > lo and  # If inner hasn't reached lo and next
               temp_key < key(self.get(inner-1))): # item's key is
           self.set(inner, self.get(inner-1)) # smaller, then shift
           inner -= 1          # next item to right & move inner left
        self.set(inner, temp)  # Move marked item to 'hole'
```

For subarrays longer than `short`, the median-of-three algorithm is used to select the pivot item. As before, the pivot value (with the median key) is placed in the array at the `hi` index so that the subarray below it can be partitioned. The call to `__part()` inside `quicksort()` (refer to Listing 7-7) runs on the subarray from `lo + 1` to `hi - 1` because the item with the lowest key from the median-of-three algorithm is stored at `lo` and, hence, does not need to be swapped with any other items during partitioning.

After partitioning, the pivot is swapped into its final position at the lowest index of the higher partition. The recursive calls sort the lower and higher partitions. These include the cells at both ends, `lo` and `hi`, because they now contain items that are in the correct partition but may not be in their final sorted position.

The other helper method is `insertionSort()`. This is the same as the method presented in Chapter 3 but has been adapted to work on subarrays. The now-familiar `lo` and `hi` indices become new parameters and default to the beginning and ending cells of the array. There is a small change in handling the `hi` limit because it indexes the last cell of the subarray instead of one past the last cell.

Quicksort in the AdvancedSorting Visualization Tool

The AdvancedSorting Visualization tool demonstrates the quicksort algorithm using median-of-three pivot selection when the check box is selected. Before, you ran quicksort without checking that box, and it selected the rightmost key as the pivot value. For three or fewer cells, the tool simply sorts the subarray using the insertion sort, regardless of whether pivots are chosen by the median-of-three.

Repeat the experiment of sorting a new 10- to 20-element array filled with increasing keys, but this time check the "Use median of 3" box. Using the rightmost cell's key caused quicksort to make 17 calls to `__part()` for the 20-element array, but when you use the median-of-three, that drops to 7. No longer is every subarray partitioned into 1 cell and N–1 cells; instead, the subarrays are partitioned roughly in half until they reach three or fewer cells.

Other than this improvement for ordered data, the difference in choosing the pivot produces similar results. It is no faster when sorting random data; its advantages become evident only when sorting ordered data.

Removing Recursion

Another embellishment recommended by many writers is removing recursion from the quicksort algorithm. This task involves rewriting the algorithm to store deferred subarray bounds (`lo` and `hi`) on a stack and using a loop instead of recursion to oversee the partitioning of smaller and smaller subarrays. The idea in doing this is to speed up the program by removing method calls. This idea, however, arose with older compilers and computer architectures, which imposed a large time penalty for each method call. It's not clear that removing recursion is much of an improvement for modern systems, which handle method calls more efficiently. The depth of recursion could be an issue on systems that limit the size of the call stack. Those systems might allow a larger data stack than the call stack, so the loop-based approach managing the subarray bounds could be able to handle larger arrays without running out of space.

Efficiency of Quicksort

We've said that quicksort operates in O(N×log N) time. As you saw in the discussion of mergesort in Chapter 6, this is generally true of the divide-and-conquer algorithms, in which a recursive method divides a range of items into two groups and then calls itself

to handle each group. In this situation, the logarithm has a base of 2: the running time is proportional to $N \times \log_2 N$.

To see why this running time is a good model, let's look at the sequence of subarrays (or partitions) processed by `quicksort()` as visualized in Figure 7-15. The first call is shown as the thickest middle line of the figure and spans all N items in the array. That middle line is numbered 1 at the left to show it is the first call. After partitioning, that subarray is split into two partitions. The index of the partition's pivot is represented by the dotted vertical line. That line connects to two thinner horizontal lines in a different color representing the work done on the lower and higher partitions. The lower one is processed first in call #2 (lower in terms of array indices and key values; higher in the figure). The higher partition is processed after all the recursive calls needed to handle the lower partition are completed; that is call #9. The process repeats for each recursive level (or depth), splitting the subarray into two smaller subarrays. The partitions at a particular depth all have the same color, while the different depths have different colors in the figure.

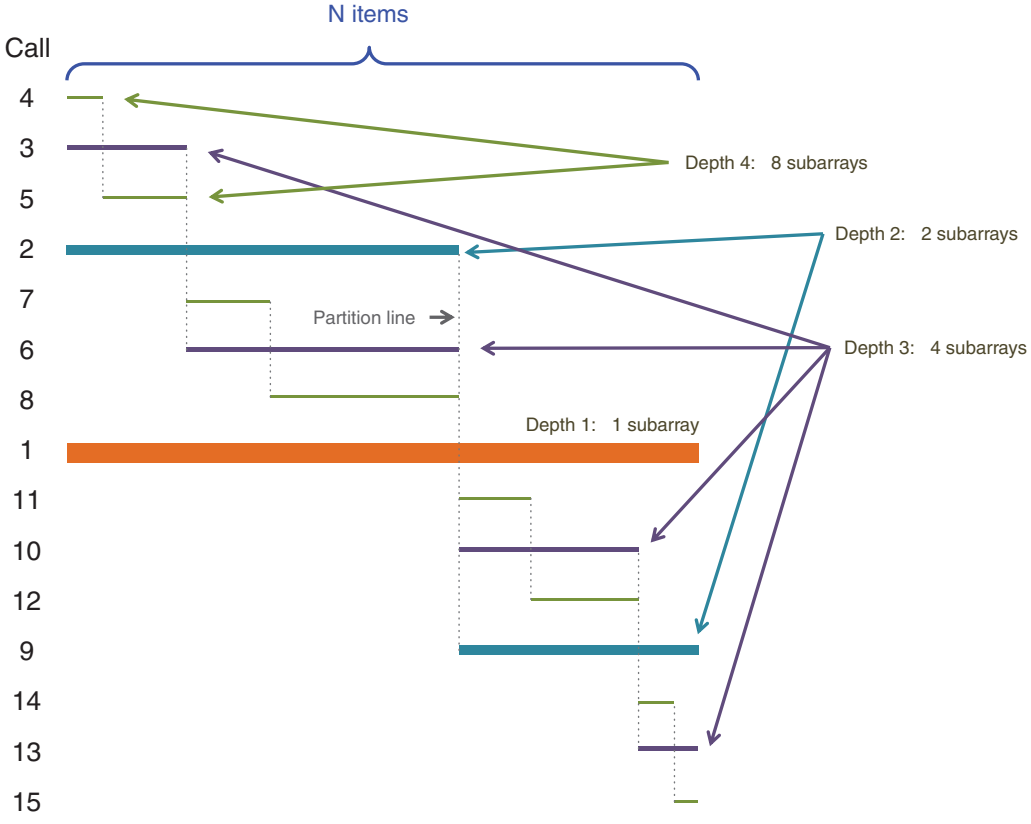


FIGURE 7-15 Lines corresponding to partitions in quicksort

The first call to `quicksort()` can be considered depth 1 of the recursive calls. It operates on the full N-item array. The two calls at depth 2 also process a total of N items; dealing with

them in two separate subarrays. Technically, they only process N – 1 items because the pivot of call #1 is not included in either of the depth 2 call's subarrays. Determining the pivot, however, requires three comparisons and zero to three swaps, which is close to the amount of work done if the pivot item were included (one more comparison and perhaps a swap). For determining the total amount of work performed at a particular depth, it's a simplification to say it processes N items.

At depth 3 there are four calls: two from each of the two subarrays at depth 2. Again, the combined items from all four calls total N (technically N – 3 items due to the removed pivots). Similarly, depth 4 has eight subarrays that cover all N items (technically N – 7). The pattern is that at depth D, 2^{D-1} calls cover N items (technically $N - (2^{D-1} - 1)$).

How deep is the recursion? If the choice of pivot were perfect every time, it would split each subarray into equal-sized partitions. That would mean there would be $\log_2 N$ depths (rounded up to the next integer). As we've pointed out, the median-of-three algorithm is only a guess at the true median key value. In the worst case, the median-of-three could end up putting N – 2 items in one partition and 1 item in the other, excluding the pivot item. If all the pivot choices were that bad, the recursive depth would go up to N/2 calls, because only 2 items would be excluded at each depth. It's the recursive depth that drives how much memory quicksort uses. It needs to record the subarray ranges for each recursive call on the stack. That means it will need O(log N) space in the best case and O(N) in the worst.

At each recursive depth, N items are processed, and you saw in the discussion of the partitioning algorithm that it performs O(N) comparisons and a smaller number of swaps (at most N/2). Multiplying the work done at each depth with total number of depths shows that the quicksort is O(N×log N) in the best case and $O(N \times N) = O(N^2)$ in the worst case.

To keep the quicksort fast, near the optimal O(N×log N) complexity, the choice of pivot and the other optimizations are essential. Going back to the partitioning algorithm, this includes the choice of how to handle items with keys equal to the pivot. By choosing to stop advancing `lo` and `hi` when they reach an item with a key equal to the pivot and then swapping those items, you increase the chance of some unnecessary swaps but improve the chance of balancing the sizes of the two partitions. In the special case (also called *degenerate case*) of an array of items all having equal keys, this algorithm divides the subarray in half. Keeping the partition sizes close to the optimum of half the subarray lowers the chance of getting the worst-case $O(N^2)$ performance.

Radix Sort

We close this chapter by briefly mentioning a few other approaches to sorting. The sorts we've looked at so far treat the key as a simple value that is compared with other values to sort the data. The comparison works with multiple data types for the keys such as numbers and strings. The radix sort decomposes the key into digits or characters and arranges the data items according to the value of the digits. Amazingly, no comparisons are necessary. All that's needed is a finite set of possible values for each digit or character. Note that the radix sort is different than the similar but somewhat more complex *radix-exchange sort*.

Algorithm for the Radix Sort

We introduce the radix sort in terms of normal base-10 arithmetic on numeric keys, which is easy to visualize. Efficient implementations of the radix sort using integer keys, however, would use base-2 arithmetic to take advantage of the computer's speed in bit manipulation.

The word **radix** means the base of a system of numbers. Ten is the radix of the decimal system, and two is the radix of the binary system. The **radix sort** involves examining each digit of the key separately, starting with the 1s (least significant) digit.

1. All the data items are divided into 10 groups, according to the value of the 1s digit of their key.
2. These 10 groups are then reassembled into a single group: All the items with keys ending with 0 go first, followed by all the items with keys ending in 1, and so on, up to 9. These reassembled groups are called a *sub-sort*.
3. In the second sub-sort, all data is divided into 10 groups again, but this time according to the value of the 10s digit of their keys. This must be done without changing the order of the previous sort. That is, within each of the 10 groups, the ordering of the items remains the same as it was after step 2. In other words, the sub-sorts must be *stable*.
4. Again the 10 groups are recombined, those with a key whose 10s digit is 0 first, then those with a 10s digit of 1, and so on up to 9.
5. This process is repeated for the remaining digits of the keys. If some keys have fewer digits than others, their higher-order digits are considered to be 0.

Here's an example, using seven data items, each with three digits. Leading zeros are shown for clarity.

```
421 240 035 532 305 430 124                 // unsorted array
(240 430) (421) (532) (124) (035 305)       // sorted on 1s digit
(305) (421 124) (430 532 035) (240)         // sorted on 10s digit
(035) (124) (240) (305) (421 430) (532)     // sorted on 100s digit
035 124 240 305 421 430 532                 // sorted array
```

The parentheses delineate the groups. Within each group, the digits in the appropriate position are the same. To convince yourself that this approach really works, try it on a piece of paper with some numbers you make up.

Designing a Radix Sort Program

In practice, the original data probably starts out in an ordinary array. Where should the 10 groups go? There's a problem with using another array or an array of 10 arrays. It's not likely there will be exactly the same number of items that have 0 for the first digit as the items having 1 for the first digit, 2 for the first digit, and so on. That makes it hard to know how big to make the arrays for each radix value. The brute-force approach would be

to allocate 10 arrays of size N so that all the items would fit in one of them. That could be a very large amount of memory, so another way to solve the problem is to use 10 linked lists instead of 10 arrays. Linked lists expand and contract as needed.

Regardless of what structure is used to hold the 10 groups, the groups can be reused for each sub-sort. The data copied into them will be copied back to the original array after each digit of the keys is processed. There's no need to allocate more memory for each pass.

An outer loop looks at each digit of the keys in turn. There are two inner loops: the first takes the data from the array and puts it on the lists; the second copies it from the lists back to the array. Choosing the right kind of linked list will make this efficient. To keep the sub-sorts stable, you need the data to come out of each list in the same order it went in. Which kind of linked list makes this easy? We leave the coding details as an exercise.

Efficiency of the Radix Sort

At first glance, the efficiency of the radix sort seems too good to be true. All you do is copy the original data from the array to the lists and back again. That's two copies per item. If there are 10 data items, this is 20 copies for the first digit. You repeat this procedure once for each digit in the keys. If you assume, say, 5-digit keys, then you'll have 20×5 equals 100 copies. If you have 100 data items, there are 200×5 equals 1,000 copies. The number of copies is proportional to the number of data items, which is O(N), the most efficient sorting algorithm we've seen in this book.

Unfortunately, it's generally true that the more data items needing to be sorted, the longer the keys need to be to distinguish them. If you have 10 times as much data, you may need to add another digit to the key. The number of copies is proportional to the number of data items times *the number of digits in the key*. The number of digits is proportional to the log of the range of key values, so in most situations you're back to O(N×log N) efficiency, the same as quicksort.

There are no comparisons, although it takes time to extract each digit from the number. This must be done once for every two copies. It may be, however, that a given computer can do the digit-extraction in binary more quickly than it can do a comparison. Of course, like mergesort, the radix sort uses O(N) memory to hold all the items in the different linked lists. Compare this to quicksort, which performs all of its swapping in the input array, and needs O(log N) memory for the stack of calls (and O(N) in the degenerate cases).

Generalizing the Radix Sort

If the keys are strings or *sequences* of values (for example tuples in Python), the radix sort can still be applied under some conditions:

- Each element in the key sequence has a known, limited number of possible values.
- There is a fast way to convert each possible value in the sequence to an integer.

For the decimal radix sort, the sequence is of digits and the number of possible values is 10. Call the radix *R* for short. For characters in a string key, the radix depends on the character set being used. For Latin alphabets, this can be limited to 256 possible characters or even 128 ignoring accented characters (for example, limiting the possibilities to just the ASCII characters). For Unicode character sets, R can grow quite large. If only a subset of the full character set can appear in a key, then R is the size of that subset, but the second condition becomes important. There must be a fast way to convert each character to an integer in the range 0 to R–1. That's required to be able to find the right linked list in which to place the items based on a character in their key.

As you might guess, the efficiency of a generalized radix sort depends heavily on the size of R and the length of the keys. Having lots of possible values for each element of a key means maintaining lots of storage structures to hold the items during each pass through the N array elements. There are some techniques to minimize the number of storage structures to just those that appear in the input keys, and we describe some in later chapters. Having long keys adds a lot of passes through N array elements. Suffice it to say that the radix sort is only really appropriate in a narrow set of circumstances.

Using a Counting Sort

One variation on implementing the radix sort avoids the use of linked list structures. The **counting sort** uses an array of integer values to count how many item keys match a particular digit of the radix. The counting array starts with all its R elements set to 0. A loop through the N input items increments the appropriate cell in the counting array for a particular digit/element of the sort key. The first two rows of Figure 7-16 show this process for the 10s digit of a 15-element array that has already been sorted on the 1s digit.

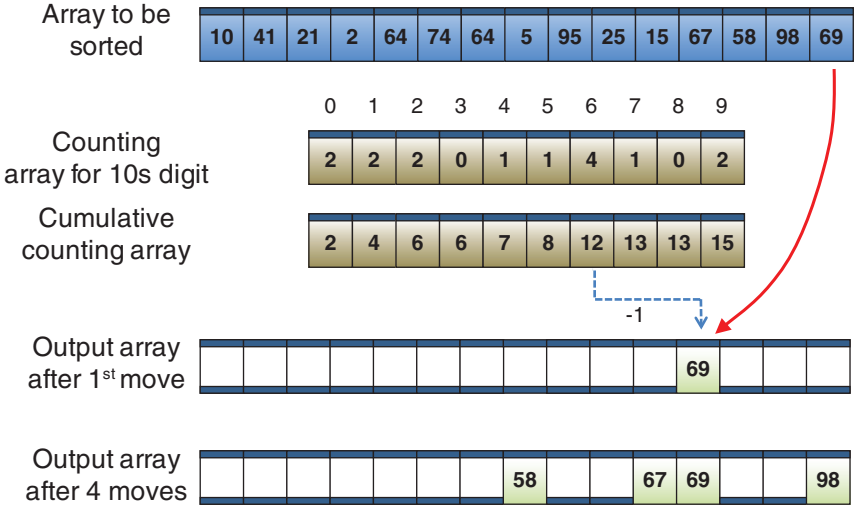


FIGURE 7-16 Applying the counting sort on the 10s digit

At the end of the loop, the counting array holds the number of matching keys for the different values of that particular digit. Some elements of the counting array may be zero, meaning that value of the digit did not appear in any key. A second loop through all but the first element of the counting array adds the count to the count on its left. That makes the r^{th} element of the counting array contain the count of all keys that match r or less. In other words, it has the *cumulative count* of matching keys. The third row of Figure 7-16 shows the result of adding the counts. Note the final value in the array is always the total count of all the keys in the input.

The next step in the counting sort makes use of the cumulative counting array to move items from the input array to an output array. The process starts from the right (highest index) cell. In the data of Figure 7-16, the rightmost key is 69 and its 10s digit is a 6. The algorithm looks at the cumulative counting array at index 6, which is 12. That means there were 12 keys whose 10s digit was 6 or less. All those items must go in the first 12 cells of the output array. It decrements the count of the cumulative array at index 6 by 1. That is 11, and it moves the item from the input array, 69, into the output array at index 11.

The process repeats for each remaining item in the input array. The next one in Figure 7-16 is an item with a key of 98. It looks up 9 in the cumulative counting array, finds 15 stored there, decrements it to 14, and places the item at index 14 in the output, which is the final position. The third item to move is one with a key of 58, whose 10s digit is 5, so it is moved to index 7. When it gets to the fourth item to move, the key is 67. The 10s digit is 6, so it looks at index 6 in the cumulative counting array. This time it finds 11 there because it was decremented once before, so it decrements it again and places item 67 at index 10 of the output array. The last row of Figure 7-16 shows the state of the output array after moving the four items.

The counting sort "knows" where to put the items in the output array based on cumulative counting array values. These items are updated with each move so that every item goes to a unique position. After processing the 10s digit, the entire process would need to be repeated on the 100s digit of all the keys. That process would use the output array from the 10s digit process and put its results back in the input array. Of course, in this particular example, none of the keys has a nonzero hundreds digit, so it requrires only sorting by the 1s followed by the 10s digits. When all of the digits of all of the keys have been processed, the output array holds the sorted items. If there were an odd number of digits in the keys, one more copy operation would be needed to copy the result back to the input array.

The counting sort has the same complexity as the radix sort. Both run in $O(N \times \log N)$ time and take O(N) memory. More precisely, both algorithms require O(N + R) memory and run in $O(N \times L)$, where R is the radix and L is the length of the keys, and L must generally be log N or greater. The difference is in what kind of structure is used for the intermediate results.

Timsort

In Chapter 3, you saw that the simple sorting algorithms could get $O(N^2)$ performance using O(1) memory. In Chapter 6, you saw that mergesort could get $O(N \times \log N)$ performance using O(N) memory. Quicksort can get $O(N \times \log N)$ performance using

O(log N) memory in most cases but can degrade to $O(N^2)$ in some cases. Timsort is a balance between some of these extremes.

Tim Peters implemented Timsort for the Python `sort()` method in 2002 based on the mergesort, insertion sort, and some clever ideas to control the memory usage and guarantee $O(N \times \log N)$ performance. Timsort uses half the memory that mergesort needs and has the side benefit of running in O(N) time on sorted input data, like the insertion sort. It is widely used outside of Python as well because of these characteristics.

Timsort is based around finding runs of items in the original array where the keys are either nondecreasing or strictly decreasing. The idea is to iterate over the items in the array until it finds a key that is lower than the key of its predecessor. That marks the end of a nondecreasing run and the beginning of a new decreasing run. Then it can look for the next key that is equal to or higher than its predecessor to find the end of the decreasing run. If it finds a decreasing run of two or more items, it swaps pairs of items from the ends like the quicksort partitioning algorithm to make it a nondecreasing run.

Figure 7-17 illustrates the runs that Timsort identifies. The height of each rectangle represents the key for that item. The array starts with an increasing sequence of four items. This group forms run 1, which is a nondecreasing run. The next four items form run 2, where the keys strictly decrease. After that, the next two items form run 3, which is nondecreasing. Adjacent keys with equal values are treated as nondecreasing, which leads to the next three runs being only two items each. Run 11 is one example where adjacent equal-valued keys are grouped together in the same run.

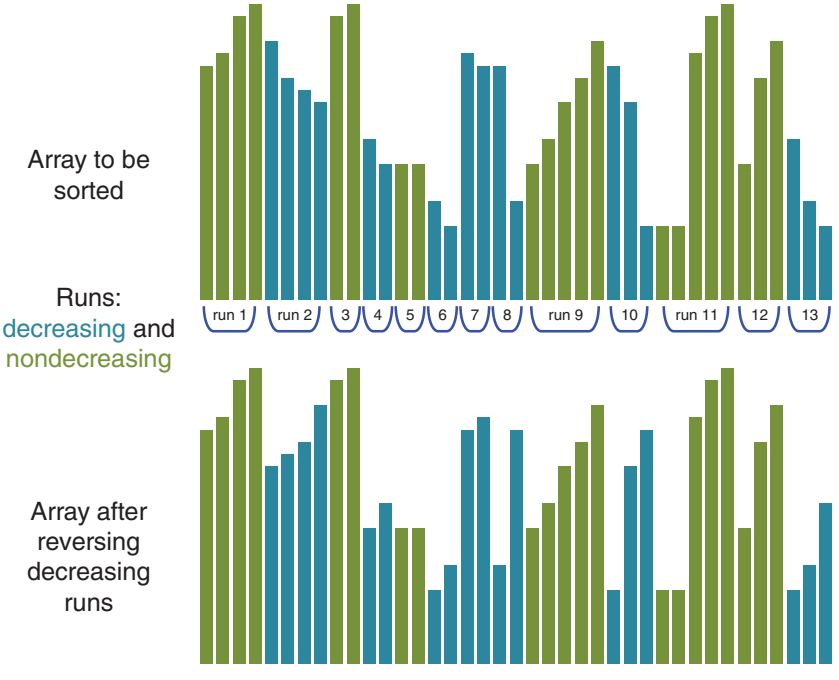


FIGURE 7-17 Runs in Timsort

The bottom panel of Figure 7-17 shows how Timsort reverses all the decreasing runs into nondecreasing ones. Now every run has nondecreasing key values.

It's straightforward to create a generator that yields these runs as it walks along the sequence, converting any decreasing runs into nondecreasing runs, all in O(N) time. A loop in Timsort uses the generator to push the runs on a stack as they are produced. After adding a new run, it checks the conditions of the top three or four runs on the stack. Under certain conditions, it starts to combine the runs using mergesort. These collapse the small runs into bigger ones. Because the runs have keys that are nondecreasing, they are already sorted. Because it runs mergesort only on adjacent runs, the sorting can be done in place in the input array, leaving a larger sorted run where the two smaller runs were.

By mergesorting the top two runs on the stack repeatedly, the entire stack of runs could be sorted into a single run. Timsort uses several techniques to reduce the amount of extra space needed for those mergesorts. First, it can reduce subarray lengths by finding where in the first run the second run begins and only mergesorting the remainder of the first run. Likewise, items with keys higher than the last key of the first run can be excluded from the second run. When you use binary search, finding where these keys land in the sorted runs is quick, O(log N).

To keep memory down and maximize the efficiency, Timsort needs to balance the sizes of the runs that it merges. The merge algorithm needs temporary space for a copy of the smaller run, but not for the larger run. Figure 7-18 illustrates a case where runs K and K+1 within the larger input array have been selected for merging. First, Timsort identifies the `left` index, which is the first key higher than the lowest key in run K+1 (20). Similarly, the `right` index is the last key lower than the highest key in run K (59). Only the subarray between `left` and `right` needs to be merged because the other items in the full runs are already in their sorted positions.

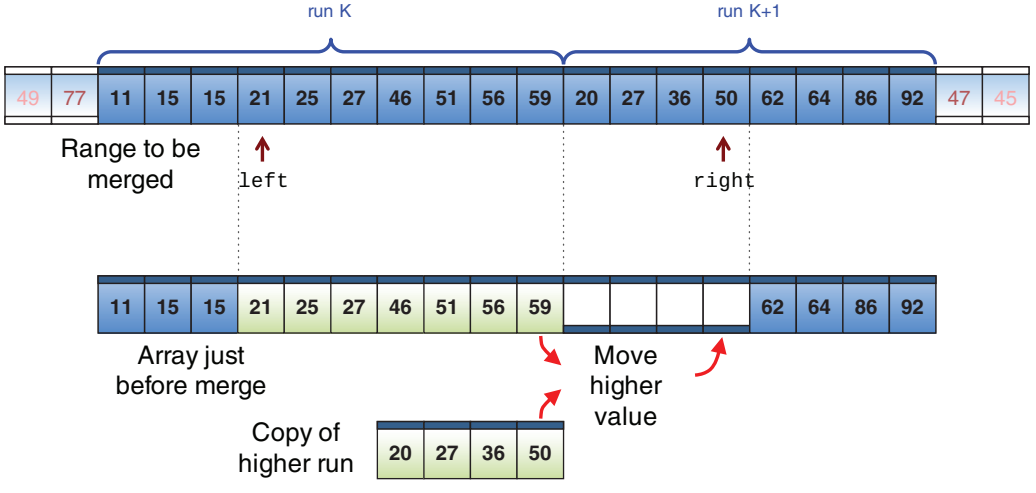


FIGURE 7-18 Merging adjacent runs in Timsort

Knowing `left`, `right`, and the boundary index of the two runs, Timsort finds the length of the sub-runs that need to be merged. In the case shown in the figure, the higher sub-run is shorter, with only four cells. Those items must be copied to a temporary array, as shown in the lower part of the figure. That "empties" the cells in the input array to allow for a merge taking the highest key from the highlighted sub-runs and moving that item to the rightmost empty cell. If the lower sub-run were shorter, then the merge process would move the lowest keyed item into the empty cells at the left (lower) side. The merge process ends when all the items in the copied sub-run have been copied back to the input array.

Efficiency of Timsort

The Timsort algorithm is quite a bit more complex and intricate (perhaps even more beautiful) than quicksort and mergesort. We've left out some details, such as using the insertion sort to combine and sort short, adjacent runs and "galloping mode." The benefit of all these techniques is that, overall, it has been shown to operate in O(N×log N) time, even in the worst case. In the best case of a forward- or reverse-sorted array, it runs in O(N) time. That extra speed comes at the cost of needing temporary storage for at most N/2 cells plus O(log N) storage for the stack of runs. In certain conditions, such as when the input data is already sorted, no merging of runs is needed, so the memory usage is O(1) in the best case. It ends up needing about half the memory of mergesort to cap the worst-case runtime, delivering the benefit of O(N) time for sorted data.

Summary

- The Shellsort applies the insertion sort to widely spaced elements of an array, then less widely spaced elements, and so on.
- The expression *n-sorting* means sorting every *n*th array element.
- A sequence of numbers, called the interval sequence, or gap sequence, is used to determine the sorting intervals in the Shellsort.
- A widely used interval sequence is generated by the recursive expression $h_i = 3 \times h_{i-1} + 1$, where the initial value, h_0, is 1.
- If an array holding 1,000 items were first to be 364-sorted, then 121-sorted, then 40-sorted, then 13-sorted, then 4-sorted, and finally 1-sorted, it would be sorted more efficiently than by simply 1-sorting it.
- The Shellsort is hard to analyze, but runs in approximately $O(N \times (\log N)^2)$ time. This is much faster than the $O(N^2)$ algorithms like the insertion sort, but slower than the O(N×log N) algorithms like the quicksort and Timsort.
- To partition an array is to divide it into two subarrays, based on a key value of each item.
- The pivot value is the value that divides the other items during partitioning. Items with keys smaller than the pivot value go in the lower group; larger keyed items in the higher group.

- Items with keys equal to the pivot may go in either partition.
- Items in the lower partition have lower indices than those of the higher partition.
- In the partitioning algorithm, two array indices, each in its own `while` loop, start at opposite ends of the array and step toward each other, looking for items that need to be swapped.
- When an array index points to an item that needs to be swapped, its `while` loop exits.
- When both `while` loops exit and the indices have met or passed each other, the partition is complete.
- When both `while` loops exit and the indices have not yet met, the items are swapped.
- Partitioning operates in linear O(N) time, making N plus 1 or 2 comparisons and at most N/2 swaps.
- The partitioning algorithm may require extra tests in its inner `while` loops to prevent the indices running off the ends of the array.
- Quicksort partitions an array and then calls itself twice recursively to sort the two resulting partitions (stored in subarrays).
- Subarrays of one or fewer elements are already sorted; this is the base case for quicksort.
- Quicksort uses the key value of a specific item in the subarray, called the pivot, to partition the subarry.
- In a simple version of quicksort, the pivot can be the item at the right end of the subarray.
- During the partitioning, the pivot is placed out of the way on the right and is not moved while the rest of the subarray is partitioned.
- Later the pivot is swapped into the space between the two partitions. This is its final sorted position.
- In the simple version of quicksort, performance is $O(N^2)$ for already sorted (or inversely sorted) data.
- An enhanced version of quicksort selects the pivot as the median of the first, middle, and last items in the subarray. This is called median-of-three partitioning.
- Median-of-three partitioning effectively eliminates the problem of $O(N^2)$ performance for already sorted data.
- In median-of-three partitioning, the first, middle, and last items are sorted at the same time the median is determined.

- This three-element sort eliminates the need for the end-of-array tests in the inner `while` loops of the partitioning algorithm.
- Quicksort operates in O(N×log N) time (except in degenerate cases such as when the simpler version is applied to already-sorted data).
- Subarrays smaller than a certain size (the cutoff) can be sorted by a method other than quicksort.
- The insertion sort is commonly used to sort subarrays smaller than the cutoff.
- The radix sort is about as fast as quicksort but uses at least O(N) memory to store copies of the data.
- Generalized radix sort can be used to sort items whose keys are sequences of values, such as strings or tuples.
- The efficiency of a radix sort depends on how many values each sequence element can take, which is the radix value, and the maximum length of the key sequences.
- Timsort is a hybrid sorting technique that finds and merges runs in the input data.
- The runs in Timsort are adjacent items in the input array with either nondecreasing or decreasing key values.
- Timsort merges adjacent runs using an enhanced version of mergesort.
- Timsort runs in O(N×log N) time in the worst case and O(N) time for forward- or reverse-sorted input data.
- Timsort uses O(N) memory in the worst case.

Questions

These questions are intended as a self-test for readers. Answers may be found in Appendix C.

1. The Shellsort works by
   a. partitioning the array into two parts.
   b. swapping adjacent elements.
   c. dealing with widely separated elements.
   d. forming increasingly larger shells of elements that are already sorted.
2. If an array has 100 elements, then Knuth's interval sequence algorithm would start with an interval of _____.

3. To transform the insertion sort into the Shellsort, which of the following do you ***not*** do?

   a. Substitute *h* for 1.

   b. Insert an algorithm for creating gaps of decreasing width.

   c. Enclose the normal insertion sort in a loop.

   d. Change the direction of the indices in the inner loop.

4. What could happen when using the Shellsort with an interval sequence created by repeatedly dividing the array size in half?

5. Fill in the Big O values: The speed of the Shellsort is more than _____ but less than _____.

6. Partitioning is

   a. putting all elements larger than a certain value on one end of the array.

   b. dividing an array into (almost) equal sized halves.

   c. partially sorting parts of an array.

   d. sorting each half of an array separately.

7. During partitioning, each array element is compared to the _____.

8. In partitioning, if an array element is equal to the answer to question 7,

   a. it is passed over.

   b. it is passed over or not, depending on the other array element.

   c. it is placed next to the array element at the right end.

   d. it is swapped.

9. True or False: In quicksort, the pivot can be an arbitrary element of the array.

10. Assuming larger keys on the right, the value returned by the `__partition()` method is

    a. the element between the left and right subarrays.

    b. the key value of the element between the left and right subarrays.

    c. the index of the leftmost element in the right subarray.

    d. the key value of the leftmost element in the right subarray.

11. Quicksort involves partitioning the original input array and then _____.

12. In the simple version of quicksort, after partitioning, the pivot may be

    a. used to find the median of the array.

    b. exchanged with an element of the right subarray.

c. used as the starting point of the next partition.

d. discarded.

13. Median-of-three partitioning is a way of choosing the _____.

14. In quicksort, for an array of N elements, the `__partition()` method will examine each element approximately _____ times.

15. What unfortunate outcome could happen when using quicksort and always choosing the rightmost element of subarrays as the input to the `__partition()` method?

16. True or False: You may speed up quicksort if you stop partitioning when the partition size is 5 or less and finish by using a different sort.

17. Radix sort

    a. uses linked lists to hold elements whose keys have certain digits.

    b. divides each element by 10 recursively to find its final sorted position.

    c. first partitions the array in 10 parts, then rearranges elements in each partition by digits in their keys.

    d. runs faster than quicksort because it doesn't have to compare entire keys.

18. The memory needed for decimal radix sort of an array of N elements is O(_____).

19. The Timsort algorithm starts by

    a. *h*-sorting the original array.

    b. finding the median key of the input array.

    c. merging the first and second halves of the array until keys start descending.

    d. finding runs of elements with nondecreasing and decreasing keys.

20. What other sorting methods are used as part of Timsort?

    a. Shellsort

    b. mergesort

    c. quicksort

    d. *h*-sort

Experiments

Carrying out these experiments will help to provide insights into the topics covered in the chapter. No programming is involved.

7-A In the `ShellSort()` implementation of Listing 7-1, the number of shifts is counted and returned. Would adding a count of the number of key comparisons be helpful in measuring performance? If so, what would it show? If not, why not?

7-B Moving the pivot in quicksort by slicing the array and reinserting it between the subarrays as shown in Figure 7-8 would require a lot of copying of cells. Is there another data structure to use in place of the array that would allow slicing and splicing in constant—O(1)—time? If so, which one and why?

7-C The partitioning algorithm can put items with keys equal to that of the pivot in either partition. That means that using partitioning in a sort algorithm like quicksort cannot be stable. Try experimenting with partitioning arrays that have many duplicate keys. Using the AdvancedSorting Visualization tool, make a new array of 10 cells and fill it with random values. Delete the rightmost item five times to clear space. Then choose a value somewhere between the minimum and maximum, insert it five times, and shuffle the array. Before running the partitioning using the repeated key value, predict which partition each copy will end up in. Now run the partitioning to check your prediction. Reshuffle the array and try again. Can you generate a situation that demonstrates that the sort will not be stable? Can you generate an initial situation where the repeated key values are stable after the sort?

7-D Judicious choice of the pivot value speeds up the quicksort. Using the AdvancedSorting Visualization tool, make a new array of 10 cells and fill it with random values. Try to sort it using only a sequence of Partition operations where *you* choose the pivot. A very poor choice of the pivot might require 10 partitions to sort 10 items. How few partitions does it take you? Can you beat the quicksort? Of course, the initial ordering affects the number of partitions that will be needed. If the initial order happens to be increasing order, you don't need any partitioning. You can always get the same order of 10 items by restarting the Visualization tool.

7-E Consider using the quicksort algorithm to rank order teams in a tournament. One team is chosen at random as the "pivot" team. It plays each of the other N–1 teams. The teams that beat the pivot team go into bracket A and those that lose go into bracket B. Teams that tie the pivot team would go into alternating brackets (first A, then B, then A, then B). The pivot team is then placed into the bracket with the fewest teams. Then each bracket conducts its own rank ordering using the same algorithm except that the choice of the pivot team for a bracket must be a team with the fewest number of previous selections as pivot within that bracket. The brackets would get smaller with each iteration until they reach size 1, and the sequence of bracket placements would determine the overall ranking of a team with A-A-A-A being highest, B-B-B-B being lowest, A-B-B-B being in the middle, and so on.

How many games would take place for a whole tournament of 16 teams? How many games would each team play? How does that compare with other tournament competition ladders that you're familiar with? If the exact numbers of games is variable or hard to determine, find the best and worst case numbers.

Programming Projects

Writing programs to solve the Programming Projects helps to solidify your understanding of the material and demonstrates how the chapter's concepts are applied. (As noted in the

Introduction, qualified instructors may obtain completed solutions to the Programming Projects on the publisher's website.)

7.1 Extend the `ShellSort.py` module of Listing 7-1 to include the two other interval sequences described in the text, namely dividing the interval by 2.2 and truncating, and Flamig's formula. For both methods the first interval, *h*, should be calculated as if the preceding value of *h* was the number of array elements. Set up a program that creates an input array and passes copies of that array to the three Shellsort methods (Knuth's sequence, divide by 2.2, and Flamig's sequence). Compare the number of shifts performed on (a) an array that is already sorted, (b) an array that is inversely sorted, and (c) a random ordering. Use a loop to perform the test with arrays of length 95 to 100 elements. Is there a method that clearly performs fewer shifts?

7.2 Modify the `quicksort()` and helper methods in Listings 7-7 and 7-8 to count the number of item copies and key comparisons they make during a sort and return the counts to the caller (using Python's ability to return multiple values). For the copies, a swap counts as 3 copies. For the comparisons, only count comparisons of key values (not array indices). Use the modified method to show how many copies and comparisons are made when sorting (a) a forward-sorted array of 50 items, (b) a reverse-sorted array of 50 items, (c) a constant value for all 50 items, and (d) an array of 50 random items. For each of those array styles, show the counts when `short` is 3, 7, and 11 to see the effect of using insertion sort for the shorter subarrays. Use the same random array for all variations of the `short` parameter to make a fair comparison.

7.3 In Exercise 3.2 in Chapter 3, we suggested finding the median of a set of data by sorting the data and picking the middle element. You might think using quicksort and picking the middle element would be the fastest way to find the median, but there's an even faster way. The quickselect algorithm uses partitioning to find the median without completely sorting the data.

To see how this works, imagine that you partition the data choosing the rightmost item as the pivot, and, by chance, the pivot happens to end up at the middle element. You're done! All the items to the right of the pivot are larger (or equal), and all the items to the left are smaller (or equal), so if the pivot falls in the exact center of the array, then it's the median. The pivot won't end up in the center very often, but you can end up getting the median by solving a smaller problem: repartitioning the partition that contains the middle element.

Suppose your array has seven elements numbered from 0 to 6. The middle is the element at index 3. If you partition this array and the pivot ends up at 4, then you need to partition again from 0 to 4 (the partition that contains index 3), not 5 to 6. If the next pivot ends up at 2, you need to partition from 2 to 6, not 0 to 1. You continue partitioning the appropriate parts recursively, always checking if the pivot falls on the original middle element at index 3. Eventually, it will, and you're done. Because you need fewer partitions than in quicksort, this algorithm is faster.

The quickselect algorithm generalizes the goal a little more by letting the caller ask for any index in the subarray. So, for example, if you wanted the 90th item among an array of 100 numbers, you would call `quickselect()` with a goal index of 90 (or maybe 89, depending on what exactly is needed). If you choose 90, `quickselect()` would return whatever item would get sorted into the cell at that index by `quicksort()`, while only partially sorting the array. To get the median, the goal index would be 50 for the 100-element array. Extend your solution to Programming Project 7.2 to implement the `quickselect()` method and make another method, `median()`, to get the middle element of an array. Your implementation should make recursive calls somewhat like those in quicksort, but they will only partition each subarray, not completely sort it. Use the rightmost element of the subarray as the pivot, as was done in `qsort()`. The process stops when the goal index is found, not when the array is sorted. As an extension of Programming Project 7.2, your method should also return the number of copies and comparisons made. Use your method to show the median found, the number of copies, and the number of comparisons for the same kinds of inputs as the previous project, namely, (a) a forward-sorted array of 50 items, (b) a reverse-sorted array of 50 items, (c) a constant value for all 50 items, and (d) an array of 50 random items.

7.4 Implement a decimal radix sort for arrays as described in the "Designing a Radix Sort Program" section. It should handle variable amounts of data and variable numbers of digits in the key. You could make the radix a parameter as well (so it can be something other than 10), but it will be hard to see what's happening unless you develop a routine to print values in different bases. Count the number of item copies made by the program. Show the results for arrays of 50 items whose maximum key is 99 and 999 to see the differences key length makes. Your program will need to determine how many digits are in the keys by examining the inputs and the radix (typically as the first pass through all the items is completed).

7.5 Implement a simplified version of the Timsort algorithm described in the "Timsort" section. Specifically, make a generator that looks for runs in an input array that are either nondecreasing or decreasing. The generator should reverse decreasing runs into increasing runs before it yields the `lo, hi` index pair that represents the run. Write a `mergerun()` method that takes two adjacent runs in the input array, uses a binary search to find the positions within the runs that need to be merged, copies the shorter section to a temporary array or other structure, and then merges the items back into the input array. The `Timsort()` method should use the generator to identify runs, one at a time, merging the second run into the first until the entire array is sorted. Show the program working on (a) a forward-sorted array of 50 items, (b) a reverse-sorted array of 50 items, (c) a constant value for all 50 items, and (d) an array of 50 random items.

CHAPTER 8

Binary Trees

IN THIS CHAPTER

- Why Use Binary Trees?
- Tree Terminology
- An Analogy
- How Do Binary Search Trees Work?
- Finding a Node
- Inserting a Node
- Traversing the Tree
- Finding Minimum and Maximum Key Values
- Deleting a Node
- The Efficiency of Binary Search Trees
- Trees Represented as Arrays
- Printing Trees
- Duplicate Keys
- The BinarySearchTreeTester.py Program
- The Huffman Code

In this chapter we switch from algorithms, the focus of Chapter 7, "Advanced Sorting," to data structures. Binary trees are one of the fundamental data storage structures used in programming. They provide advantages that the data structures you've seen so far cannot. In this chapter you learn why you would want to use trees, how they work, and how to go about creating them.

Why Use Binary Trees?

Why might you want to use a tree? Usually, because it combines the advantages of two other structures: an ordered array and a linked list. You can search a tree quickly, as you can an ordered array, and you can also insert and delete items quickly, as you can with a linked list. Let's explore these topics a bit before delving into the details of trees.

Slow Insertion in an Ordered Array

Imagine an array in which all the elements are arranged in order—that is, an ordered array—such as you saw in Chapter 2, "Arrays." As you learned, you can quickly search such an array for a particular value, using a binary search. You check in the center of the array; if the object you're looking for is greater than what you find there, you narrow your search to the top half of the array; if it's less, you narrow your search to the bottom half. Applying this process repeatedly finds the object in O(log N) time. You can also quickly traverse an ordered array, visiting each object in sorted order.

On the other hand, if you want to insert a new object into an ordered array, you first need to find where the object will go and then move all the objects with greater keys up one space in the array to make room for it. These multiple

moves are time-consuming, requiring, on average, moving half the items (N/2 moves). Deletion involves the same multiple moves and is thus equally slow.

If you're going to be doing a lot of insertions and deletions, an ordered array is a bad choice.

Slow Searching in a Linked List

As you saw in Chapter 5, "Linked Lists," you can quickly perform insertions and deletions on a linked list. You can accomplish these operations simply by changing a few references. These two operations require O(1) time (the fastest Big O time).

Unfortunately, *finding* a specified element in a linked list is not as fast. You must start at the beginning of the list and visit each element until you find the one you're looking for. Thus, you need to visit an average of N/2 objects, comparing each one's key with the desired value. This process is slow, requiring O(N) time. (Notice that times considered fast for a sort are slow for the basic data structure operations of insertion, deletion, and search.)

You might think you could speed things up by using an ordered linked list, in which the elements are arranged in order, but this doesn't help. You still must start at the beginning and visit the elements in order because there's no way to access a given element without following the chain of references to it. You could abandon the search for an element after finding a gap in the ordered sequence where it should have been, so it would save a little time in identifying missing items. Using an ordered list only helps make traversing the nodes in order quicker and doesn't help in finding an arbitrary object.

Trees to the Rescue

It would be nice if there were a data structure with the quick insertion and deletion of a linked list, along with the quick searching of an ordered array. Trees provide both of these characteristics and are also one of the most interesting data structures.

What Is a Tree?

A tree consists of **nodes** connected by **edges**. Figure 8-1 shows a tree. In such a picture of a tree the nodes are represented as circles, and the edges as lines connecting the circles.

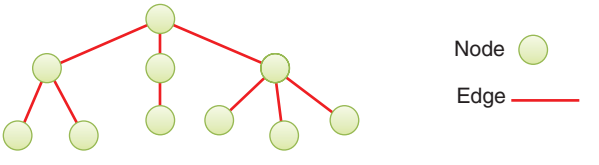


FIGURE 8-1 A general (nonbinary) tree

Trees have been studied extensively as abstract mathematical entities, so there's a large amount of theoretical knowledge about them. A tree is actually an instance of a more general category called a **graph**. The types and arrangement of edges connecting the nodes distinguish trees and graphs, but you don't need to worry about the extra issues graphs present. We discuss graphs in Chapter 14, "Graphs," and Chapter 15, "Weighted Graphs."

In computer programs, nodes often represent entities such as file folders, files, departments, people, and so on—in other words, the typical records and items stored in any kind of data structure. In an object-oriented programming language, the nodes are objects that represent entities, sometimes in the real world.

The lines (edges) between the nodes represent the way the nodes are related. Roughly speaking, the lines represent convenience: it's easy (and fast) for a program to get from one node to another if a line connects them. In fact, the *only* way to get from node to node is to follow a path along the lines. These are essentially the same as the references you saw in linked lists; each node can have some references to other nodes. Algorithms are restricted to going in one direction along edges: from the node with the reference to some other node. Doubly linked nodes are sometimes used to go both directions.

Typically, one node is designated as the **root** of the tree. Just like the head of a linked list, all the other nodes are reached by following edges from the root. The root node is typically drawn at the top of a diagram, like the one in Figure 8-1. The other nodes are shown below it, and the further down in the diagram, the more edges need to be followed to get to another node. Thus, tree diagrams are small on the top and large on the bottom. This configuration may seem upside-down compared with real trees, at least compared to the parts of real trees above ground; the diagrams are more like tree root systems in a visual sense. This arrangement makes them more like charts used to show family trees with ancestors at the top and descendants below. Generally, programs start an operation at the small part of the tree, the root, and follow the edges out to the broader fringe. It's (arguably) more natural to think about going from top to bottom, as in reading text, so having the other nodes below the root helps indicate the relative order of the nodes.

There are different kinds of trees, distinguished by the number and type of edges. The tree shown in Figure 8-1 has more than two children per node. (We explain what "children" means in a moment.) In this chapter we discuss a specialized form of tree called a **binary tree**. Each node in a binary tree has a maximum of two children. More general trees, in which nodes can have more than two children, are called **multiway trees**. We show examples of multiway trees in Chapter 9, "2-3-4 Trees and External Storage."

Tree Terminology

Many terms are used to describe particular aspects of trees. You need to know them so that this discussion is comprehensible. Fortunately, most of these terms are related to real-world trees or to family relationships, so they're not hard to remember. Figure 8-2 shows many of these terms applied to a binary tree.

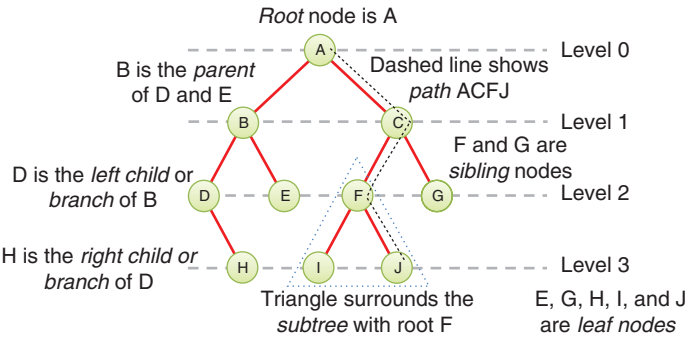


FIGURE 8-2 Tree terms

Root

The node at the top of the tree is called the **root**. There is only one root in a tree, labeled A in the figure.

Path

Think of someone walking from node to node along the edges that connect them. The resulting sequence of nodes is called a **path**. For a collection of nodes and edges to be defined as a tree, there must be one (and only one!) path from the root to any other node. Figure 8-3 shows a nontree. You can see that it violates this rule because there are multiple paths from A to nodes E and F. This is an example of a *graph* that is not a tree.

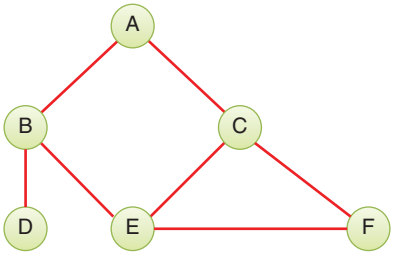


FIGURE 8-3 A nontree

Parent

Any node (except the root) has exactly one edge connecting it to a node above it. The node above it is called the **parent** of the node. The root node must not have a parent.

Child

Any node may have one or more edges connecting it to nodes below. These nodes below a given node are called its **children**, or sometimes, **branches**.

Sibling

Any node other than the root node may have **sibling** nodes. These nodes have a common parent node.

Leaf

A node that has no children is called a **leaf node** or simply a **leaf**. There can be only one root in a tree, but there can be many leaves. In contrast, a node that has children is an **internal node**.

Subtree

Any node (other than the root) may be considered to be the root of a **subtree**, which also includes its children, and its children's children, and so on. If you think in terms of families, a node's subtree contains all its **descendants**.

Visiting

A node is **visited** when program control arrives at the node, usually for the purpose of carrying out some operation on the node, such as checking the value of one of its data fields or displaying it. Merely passing over a node on the path from one node to another is not considered to be visiting the node.

Traversing

To **traverse** a tree means to visit all the nodes in some specified order. For example, you might visit all the nodes in order of ascending key value. There are other ways to traverse a tree, as we'll describe later.

Levels

The **level** of a particular node refers to how many generations the node is from the root. If you assume the root is Level 0, then its children are at Level 1, its grandchildren are at Level 2, and so on. This is also sometimes called the **depth** of a node.

Keys

You've seen that one data field in an object is usually designated as a **key value**, or simply a **key**. This value is used to search for the item or perform other operations on it. In tree diagrams, when a circle represents a node holding a data item, the key value of the item is typically shown in the circle.

Binary Trees

If every node in a tree has at most two children, the tree is called a **binary tree**. In this chapter we focus on binary trees because they are the simplest and the most common.

The two children of each node in a binary tree are called the **left child** and the **right child**, corresponding to their positions when you draw a picture of a tree, as shown in Figure 8-2. A node in a binary tree doesn't necessarily have the maximum of two

children; it may have only a left child or only a right child, or it can have no children at all (in which case it's a leaf).

Binary Search Trees

The kind of binary tree we discuss at the beginning of this chapter is technically called a **binary search tree**. The keys of the nodes have a particular ordering in search trees. Figure 8-4 shows a binary search tree.

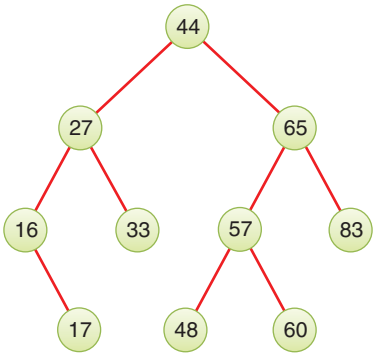


FIGURE 8-4 A binary search tree

NOTE

The defining characteristic of a binary search tree is this: a node's left child must have a key less than its parent's key, and a node's right child must have a key greater than or equal to that of its parent.

An Analogy

One commonly encountered tree is the hierarchical file system on desktop computers. This system was modeled on the prevailing document storage technology used by businesses in the twentieth century: filing cabinets containing folders that in turn contained subfolders, down to individual documents. Computer operating systems mimic that by having files stored in a hierarchy. At the top of the hierarchy is the root directory. That directory contains "folders," which are subdirectories, and files, which are like the paper documents. Each subdirectory can have subdirectories of its own and more files. These all have analogies in the tree: the root directory is the root node, subdirectories are nodes with children, and files are leaf nodes.

To specify a particular file in a file system, you use the full path from the root directory down to the file. This is the same as the path to a node of a tree. Uniform resource locators (URLs) use a similar construction to show a path to a resource on the Internet. Both file system pathnames and URLs allow for many levels of subdirectories. The last name in a file system path is either a subdirectory or a file. Files represent leaves; they have no children of their own.

Clearly, a hierarchical file system is not a binary tree because a directory may have many children. A hierarchical file system differs in another significant way from the trees that we discuss here. In the file system, subdirectories contain no data other than attributes like their name; they contain only references to other subdirectories or to files. Only files contain data. In a tree, every node contains data. The exact type of data depends on what's being represented: records about personnel, records about components used to construct a vehicle, and so forth. In addition to the data, all nodes except leaves contain references to other nodes.

Hierarchical file systems differ from binary search trees in other aspects, too. The purpose of the file system is to organize files; the purpose of a binary search tree is more general and abstract. It's a data structure that provides the common operations of insertion, deletion, search, and traversal on a collection of items, organizing them by their keys to speed up the operations. The analogy between the two is meant to show another familiar system that shares some important characteristics, but not all.

How Do Binary Search Trees Work?

Let's see how to carry out the common binary tree operations of finding a node with a given key, inserting a new node, traversing the tree, and deleting a node. For each of these operations, we first show how to use the Binary Search Tree Visualization tool to carry it out; then we look at the corresponding Python code.

The Binary Search Tree Visualization Tool

For this example, start the Binary Search Tree Visualization tool (the program is called `BinaryTree.py`). You should see a screen something like that shown in Figure 8-5.

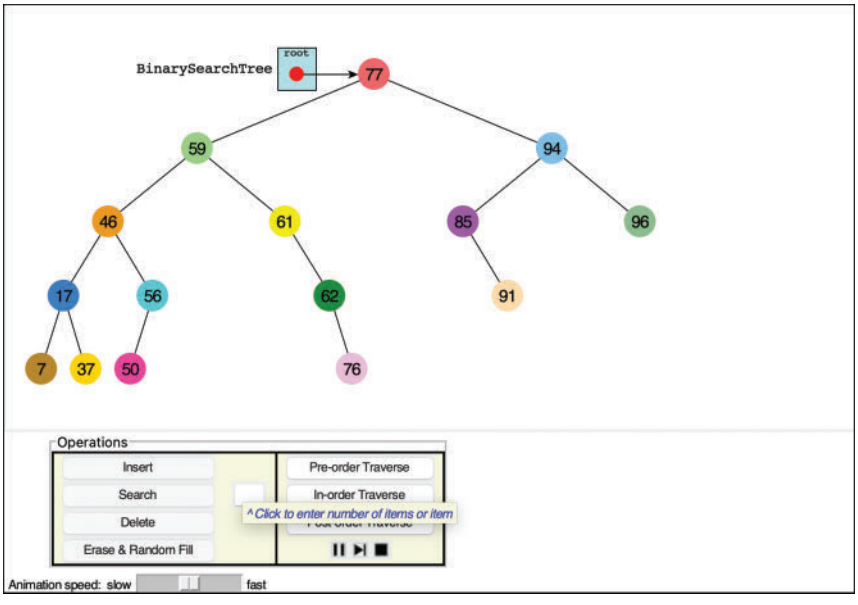


FIGURE 8-5 The Binary Search Tree Visualization tool

Using the Visualization Tool

The key values shown in the nodes range from 0 to 99. Of course, in a real tree, there would probably be a larger range of key values. For example, if telephone numbers were used for key values, they could range up to 999,999,999,999,999 (15 digits including country codes in the International Telecommunication Union standard). We focus on a simpler set of possible keys.

Another difference between the Visualization tool and a real tree is that the tool limits its tree to a depth of five; that is, there can be no more than five levels from the root to the bottom (level 0 through level 4). This restriction ensures that all the nodes in the tree will be visible on the screen. In a real tree the number of levels is unlimited (until the computer runs out of memory).

Using the Visualization tool, you can create a new tree whenever you want. To do this, enter a number of items and click the Erase & Random Fill button. You can ask to fill with 0 to 99 items. If you choose 0, you will create an empty tree. Using larger numbers will fill in more nodes, but some of the requested nodes may not appear. That's due to the limit on the depth of the tree and the random order the items are inserted. You can experiment by creating trees with different numbers of nodes to see the variety of trees that come out of the random sequencing.

The nodes are created with different colors. The color represents the data stored with the key. We show a little later how that data is updated in some operations.

Constructing Trees

As shown in the Visualization tool, the tree's shape depends both on the items it contains as well as the order the items are inserted into the tree. That might seem strange at first. If items are inserted into a sorted array, they always end up in the same order, regardless of their sequencing. Why are binary search trees different?

One of the key features of the binary search tree is that it does not have to fully order the items as they are inserted. When it adds a new item to an existing tree, it decides where to place the new leaf node by comparing its key with that of the nodes already stored in the tree. It follows a path from the root down to a missing child where the new node "belongs." By choosing the left child when the new node's key is less than the key of an internal node and the right child for other values, there will always be a unique path for the new node. That unique path means you can easily find that node by its key later, but it also means that the previously inserted items affect the path to any new item.

For example, if you start with an empty binary search tree and insert nodes in increasing key order, the unique path for each one will always be the rightmost path. Each insertion adds one more node at the bottom right. If you reverse the order of the nodes and insert them into an empty tree, each insertion will add the node at the bottom left because the key is lower than any other in the tree so far. Figure 8-6 shows what happens if you insert nodes with keys 44, 65, 83, and 87 in forward or reverse order.

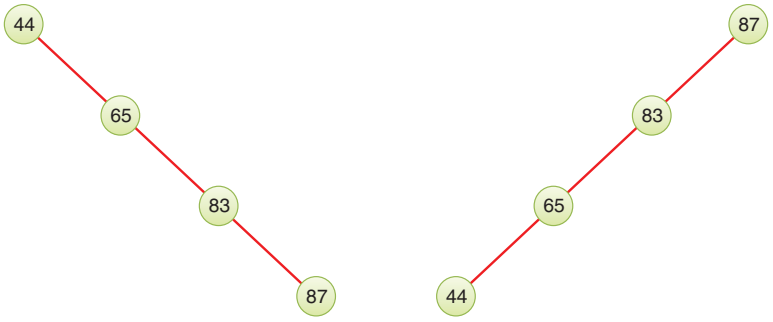


FIGURE 8-6 Trees made by inserting nodes in sorted order

Unbalanced Trees

The trees shown in Figure 8-6, don't look like trees. In fact, they look more like linked lists. One of the goals for a binary search tree is to speed up the search for a particular node, so having to step through a linked list to find the node would not be an improvement. To get the benefit of the tree, the nodes should be roughly balanced on both sides of the root. Ideally, each step along the path to find a node should cut the number of nodes to search in half, just like in binary searches of arrays and the guess-a-number game described in Chapter 2.

We can call some trees **unbalanced**; that is, they have most of their nodes on one side of the root or the other, as shown in Figure 8-7. Any subtree may also be unbalanced like the outlined one on the lower left of the figure. Of course, only a fully balanced tree will have equal numbers of nodes on the left and right subtrees (and being fully balanced, every node's subtree will be fully balanced too). Inserting nodes one at a time on randomly chosen items means most trees will be unbalanced by one or more nodes as they are constructed, so you typically cannot expect to find fully balanced trees. In the following chapters, we look more carefully at ways to balance them as nodes are inserted and deleted.

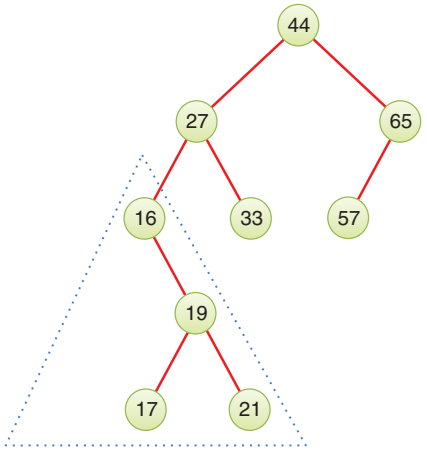


FIGURE 8-7 An unbalanced tree (with an unbalanced subtree)

Trees become unbalanced because of the order in which the data items are inserted. If these key values are inserted randomly, the tree will be more or less balanced. When an ascending sequence (like 11, 18, 33, 42, 65) or a descending sequence is encountered, all the values will be right children (if ascending) or left children (if descending), and the tree will be unbalanced. The key values in the Visualization tool are generated randomly, but of course some short ascending or descending sequences will be created anyway, which will lead to local imbalances.

If a tree is created by data items whose key values arrive in random order, the problem of unbalanced trees may not be too much of a problem for larger trees because the chances of a long run of numbers in a sequence is small. Sometimes, however, key values will arrive in strict sequence; for example, when someone doing data entry arranges a stack of forms into alphabetical order by name before entering the data. When this happens, tree efficiency can be seriously degraded. We discuss unbalanced trees and what to do about them in Chapters 9 and 10.

Representing the Tree in Python Code

Let's start implementing a binary search tree in Python. As with other data structures, there are several approaches to representing a tree in the computer's memory. The most common is to store the nodes at (unrelated) locations in memory and connect them using references in each node that point to its children.

You can also represent a tree in memory as an array, with nodes in specific positions stored in corresponding positions in the array. We return to this possibility at the end of this chapter. For our sample Python code we'll use the approach of connecting the nodes using references, similar to the way linked lists were implemented in Chapter 5.

The BinarySearchTree Class

We need a class for the overall tree object: the object that holds, or at least leads to, all the nodes. We'll call this class `BinarySearchTree`. It has only one field, `__root`, that holds the reference to the root node, as shown in Listing 8-1. This is very similar to the `LinkedList` class that was used in Chapter 5 to represent linked lists. The `BinarySearchTree` class doesn't need fields for the other nodes because they are all accessed from the root node by following other references.

LISTING 8-1 The Constructor for the `BinarySearchTree` Class

```
class BinarySearchTree(object): # A binary search tree class

   def __init__(self):          # The tree organizes nodes by their
      self.__root = None        # keys. Initially, it is empty.
```

The constructor initializes the reference to the root node as `None` to start with an empty tree. When the first node is inserted, `__root` will point to it as shown in the Visualization tool example of Figure 8-5. There are, of course, many methods that operate on `BinarySearchTree` objects, but first, you need to define the nodes inside them.

The __Node Class

The nodes of the tree contain the data representing the objects being stored (contact information in an address book, for example), a key to identify those objects (and to order them), and the references to each of the node's two children. To remind us that callers that create `BinarySearchTree` objects should not be allowed to directly alter the nodes, we make a private `__Node` class inside that class. Listing 8-2 shows how an internal class can be defined inside the `BinarySearchTree` class.

LISTING 8-2 The Constructors for the `__Node` and `BinarySearchTree` Classes

```
class BinarySearchTree(object):  # A binary search tree class
…

   class __Node(object):         # A node in a binary search tree
      def __init__(              # Constructor takes a key that is
            self,                # used to determine the position
            key,                 # inside the search tree,
            data,                # the data associated with the key
            left=None,           # and a left and right child node
            right=None):         # if known
         self.key  = key         # Copy parameters to instance
         self.data = data        # attributes of the object
         self.leftChild = left
         self.rightChild = right

      def __str__(self):         # Represent a node as a string
         return "{" + str(self.key) + ", " + str(self.data) + "}"

   def __init__(self):           # The tree organizes nodes by their
      self.__root = None         # keys. Initially, it is empty.

   def isEmpty(self):            # Check for empty tree
      return self.__root is None

   def root(self):               # Get the data and key of the root node
      if self.isEmpty():         # If the tree is empty, raise exception
         raise Exception("No root node in empty tree")
      return (                   # Otherwise return root data and its key
         self.__root.data, self.__root.key)
```

The `__Node` objects are created and manipulated by the `BinarySearchTree`'s methods. The fields inside `__Node` can be initialized as public attributes because the `BinarySearchTree`'s methods take care never to return a `__Node` object. This declaration allows for direct reading and writing without making accessor methods like `getKey()` or `setData()`. The `__Node` constructor simply populates the fields from the arguments provided. If the child nodes are not provided, the fields for their references are filled with `None`.

We add a `__str__()` method for `__Node` objects to aid in displaying the contents while debugging. Notably, it does not show the child nodes. We discuss how to display full trees a little later. That's all the methods needed for `__Node` objects; all the rest of the methods you define are for `BinarySearchTree` objects.

Listing 8-2 shows an `isEmpty()` method for `BinarySearchTree` objects that checks whether the tree has any nodes in it. The `root()` method extracts the root node's data and key. It's like `peek()` for a queue and raises an exception if the tree is empty.

Some programmers also include a reference to a node's parent in the `__Node` class. Doing so simplifies some operations but complicates others, so we don't include it here. Adding a parent reference achieves something similar to the `DoublyLinkedList` class described in Chapter 5, "Linked Lists"; it's useful in certain contexts but adds complexity.

We've made another design choice by storing the key for each node in its own field. For the data structures based on arrays, we chose to use a key function that extracts the key from each array item. That approach was more convenient for arrays because storing the keys separately from the data would require the equivalent of a key array along with the data array. In the case of node class with named fields, adding a key field makes the code perhaps more readable and somewhat more efficient by avoiding some function calls. It also makes the key more independent of the data, which adds flexibility and can be used to enforce constraints like immutable keys even when data changes.

The `BinarySearchTree` class has several methods. They are used for finding, inserting, deleting, and traversing nodes; and for displaying the tree. We investigate them each separately.

Finding a Node

Finding a node with a specific key is the simplest of the major tree operations. It's also the most important because it is essential to the binary search tree's purpose.

The Visualization tool shows only the key for each node and a color for its data. Keep in mind that the purpose of the data structure is to store a collection of records, not just the key or a simple color. The keys can be more than simple integers; any data type that can be ordered could be used. The Visualization and examples shown here use integers for brevity. After a node is discovered by its key, it's the data that's returned to the caller, not the node itself.

Using the Visualization Tool to Find a Node

Look at the Visualization tool and pick a node, preferably one near the bottom of the tree (as far from the root as possible). The number shown in this node is its *key value*. We're going to demonstrate how the Visualization tool finds the node, given the key value.

For purposes of this discussion, we choose to find the node holding the item with key value 50, as shown in Figure 8-8. Of course, when you run the Visualization tool, you may get a different tree and may need to pick a different key value.

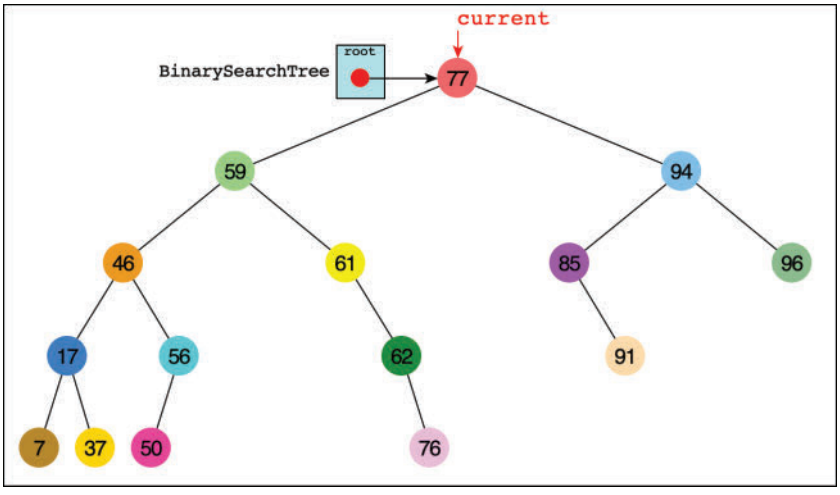


FIGURE 8-8 Finding the node with key 50

Enter the key value in the text entry box, hold down the Shift key, and select the Search button, and then the Step button, ▶|. By repeatedly pressing the Step button, you can see all the individual steps taken to find key 50. On the second press, the `current` pointer shows up at the root of the tree, as seen in Figure 8-8. On the next click, a `parent` pointer shows up that will follow the `current` pointer. Ignore that pointer and the code display for a moment; we describe them in detail shortly.

As the Visualization tool looks for the specified node, it makes a decision at the `current` node. It compares the desired key with the one found at the current node. If it's the same, it's found the desired node and can quit. If not, it must decide where to look next.

In Figure 8-8 the `current` arrow starts at the root. The program compares the goal key value 50 with the value at the root, which is 77. The goal key is less, so the program knows the desired node must be on the left side of the tree—either the root's left child or one of that child's descendants. The left child of the root has the value 59, so the comparison of 50 and 59 will show that the desired node is in the left subtree of 59. The `current` arrow goes to 46, the root of that subtree. This time, 50 is greater than the 46 node, so it goes to the right, to node 56, as shown in Figure 8-9. A few steps later, comparing 50 with 56 leads the program to the left child. The comparison at that leaf node shows that 50 equals the node's key value, so it has found the node we sought.

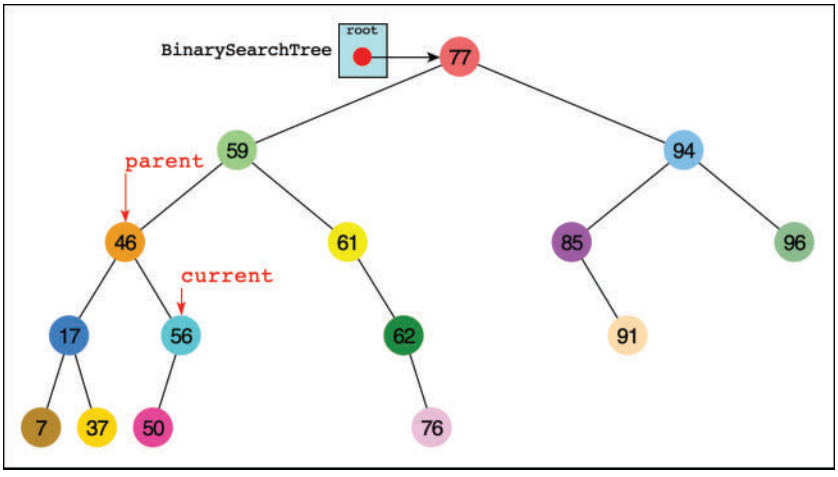


FIGURE 8-9 The second to last step in finding key 50

The Visualization tool changes a little after it finds the desired node. The `current` arrow changes into the `node` arrow (and `parent` changes into `p`). That's because of the way variables are named in the code, which we show in the next section. The tool doesn't do anything with the node after finding it, except to encircle it and display a message saying it has been found. A serious program would perform some operation on the found node, such as displaying its contents or changing one of its fields.

Python Code for Finding a Node

Listing 8-3 shows the code for the `__find()` and `search()` methods. The `__find()` method is private because it can return a node object. Callers of the `BinarySearchTree` class use the `search()` method to get the data stored in a node.

LISTING 8-3 The Methods to Find a Binary Search Tree Node Based on Its Key

```
class BinarySearchTree(object):          # A binary search tree class
…
   def __find(self, goal):               # Find an internal node whose key
      current = self.__root              # matches goal and its parent. Start at
      parent = self                      # root and track parent of current node
      while (current and                 # While there is a tree left to explore
             goal != current.key):       # and current key isn't the goal
         parent = current                # Prepare to move one level down
         current = (                     # Advance current to left subtree when
            current.leftChild if goal < current.key else # goal is
            current.rightChild)          # less than current key, else right

      # If the loop ended on a node, it must have the goal key
      return (current, parent)           # Return the node or None and parent
```

```
def search(self, goal):                # Public method to get data associated
   node, p = self.__find(goal)         # with a goal key. First, find node
   return node.data if node else None # w/ goal & return any data
```

The only argument to `__find()` is `goal`, the key value to be found. This routine creates the variable `current` to hold the node currently being examined. The routine starts at the root – the only node it can access directly. That is, it sets `current` to the root. It also sets a `parent` variable to `self`, which is the tree object. In the Visualization tool, `parent` starts off pointing at the tree object. Because parent links are not stored in the nodes, the `__find()` method tracks the parent node of `current` so that it can return it to the caller along with the goal node. This capability will be very useful in other methods. The `parent` variable is always either the `BinarySearchTree` being searched or one of its `__Node` objects.

In the `while` loop, `__find()` first confirms that current is not `None` and references some existing node. If it doesn't, the search has gone beyond a leaf node (or started with an empty tree), and the goal node isn't in the tree. The second part of the `while` test compares the value to be found, `goal`, with the value of the `current` node's `key` field. If the key matches, then the loop is done. If it doesn't, then `current` needs to advance to the appropriate subtree. First, it updates `parent` to be the `current` node and then updates `current`. If `goal` is less than `current`'s key, `current` advances to its left child. If `goal` is greater than `current`'s key, `current` advances to its right child.

Can't Find the Node

If `current` becomes equal to `None`, you've reached the end of the line without finding the node you were looking for, so it can't be in the tree. That could happen if the root node was `None` or if following the child links led to a node without a child (on the side where the goal key would go). Both the current node (`None`) and its parent are returned to the caller to indicate the result. In the Visualization tool, try entering a key that doesn't appear in the tree and select Search. You then see the `current` pointer descend through the existing nodes and land on a spot where the key should be found but no node exists. Pointing to "empty space" indicates that the variable's value is `None`.

Found the Node

If the condition of the `while` loop is not satisfied while `current` references some node in the tree, then the loop exits, and the `current` key must be the goal. That is, it has found the node being sought and `current` references it. It returns the node reference along with the parent reference so that the routine that called `__find()` can access any of the node's (or its parent's) data. Note that it returns the value of `current` for both success and failure of finding the key; it is `None` when the goal isn't found.

The `search()` method calls the `__find()` method to set its `node` and parent (`p`) variables. That's what you see in the Visualization tool after the `__find()` method returns. If a non-`None` reference was found, `search()` returns the data for that node. In this case, the method assumes that data items stored in the nodes can never be `None`; otherwise, the caller would not be able to distinguish them.

Tree Efficiency

As you can see, the time required to find a node depends on its depth in the tree, the number of levels below the root. If the tree is balanced, this is O(log N) time, or more specifically O($\log_2$ N) time, the logarithm to base 2, where N is the number of nodes. It's just like the binary search done in arrays where half the nodes were eliminated after each comparison. A fully balanced tree is the best case. In the worst case, the tree is completely unbalanced, like the examples shown in Figure 8-6, and the time required is O(N). We discuss the efficiency of `__find()` and other operations toward the end of this chapter.

Inserting a Node

To insert a node, you must first find the place to insert it. This is the same process as trying to find a node that turns out not to exist, as described in the earlier "Can't Find the Node" section. You follow the path from the root toward the appropriate node. This is either a node with the same key as the node to be inserted or `None`, if this is a new key. If it's the same key, you could try to insert it in the right subtree, but doing so adds some complexity. Another option is to replace the data for that node with the new data. For now, we allow only unique keys to be inserted; we discuss duplicate keys later.

If the key to insert is not in the tree, then `__find()` returns `None` for the reference to the node along with a parent reference. The new node is connected as the parent's left or right child, depending on whether the new node's key is less or greater than that of the parent. If the parent reference returned by `__find()` is `self`, the `BinarySearchTree` itself, then the node becomes the root node.

Figure 8-10 illustrates the process of inserting a node, with key 31, into a tree. The `__find(31)` method starts walking the path from the root node. Because 31 is less than the root node key, 44, it follows the left child link. That child's key is 27, so it follows that child's right child link. There it encounters key 33, so it again follows the left child link. That is `None`, so `__find(31)` stops with the parent pointing at the leaf node with key 33. The new leaf node with key 31 is created, and the parent's left child link is set to reference it.

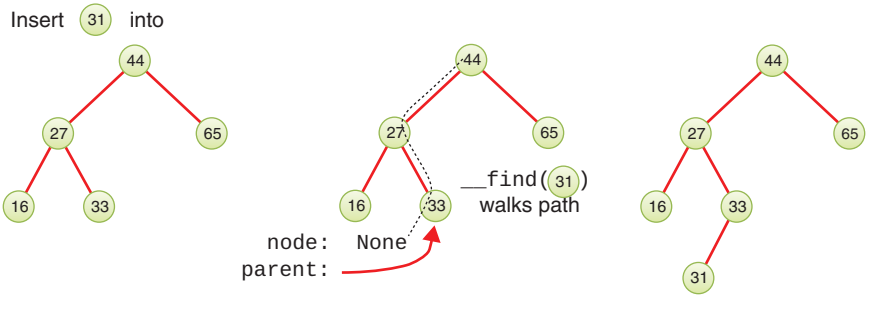


FIGURE 8-10 Inserting a node in binary search tree

Using the Visualization Tool to Insert a Node

To insert a new node with the Visualization tool, enter a key value that's not in the tree and select the Insert button. The first step for the program is to find where it should be inserted. For example, inserting 81 into the tree from an earlier example calls the `__find()` method of Listing 8-3, which causes the search to follow the path shown in Figure 8-11.

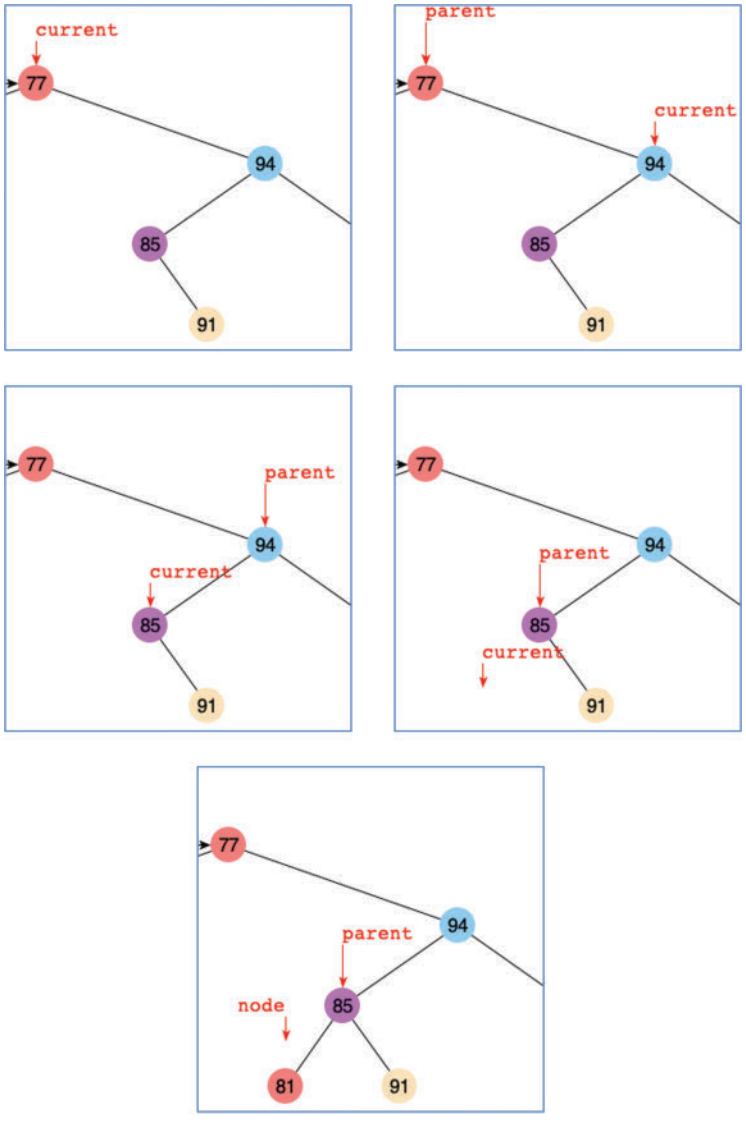


FIGURE 8-11 Steps for inserting a node with key 81 using the Visualization tool

The `current` pointer starts at the root node with key 77. Finding 81 to be larger, it goes to the right child, node 94. Now the key to insert is less than the `current` key, so it descends

to node 85. The `parent` pointer follows the `current` pointer at each of these steps. When `current` reaches node 85, it goes to its left child and finds it missing. The call to `__find()` returns `None` and the `parent` pointer.

After locating the `parent` node with the empty child where the new key should go, the Visualization tool creates a new node with the key 81, some data represented by a color, and sets the left child pointer of node 85, the parent, to point at it. The `node` pointer returned by `__find()` is moved away because it still is `None`.

Python Code for Inserting a Node

The `insert()` method takes parameters for the key and data to insert, as shown in Listing 8-4. It calls the `__find()` method with the new node's key to determine whether that key already exists and where its parent node should be. This implementation allows only unique keys in the tree, so if it finds a node with the same key, `insert()` updates the data for that key and returns `False` to indicate that no new node was created.

LISTING 8-4 The `insert()` Method of `BinarySearchTree`

```
class BinarySearchTree(object):                # A binary search tree class
…
   def insert(self,                            # Insert a new node in a binary
              key,                             # search tree finding where its key
              data):                           # places it and storing its data
      node, parent = self.__find(              # Try finding the key in the tree
         key)                                  # and getting its parent node
      if node:                                 # If we find a node with this key,
         node.data = data                      # then update the node's data
         return False                          # and return flag for no insertion

      if parent is self:                       # For empty trees, insert new node at
         self.__root = self.__Node(key, data) # root of tree
      elif key < parent.key:                 # If new key is less than parent's key,
         parent.leftChild = self.__Node(       # insert new node as left
            key, data, right=node)             # child of parent
      else:                                    # Otherwise insert new node as right
         parent.rightChild = self.__Node(      # child of parent
            key, data, right=node)
      return True                              # Return flag for valid insertion
```

If a matching node was not found, then insertion depends on what kind of parent reference was returned from `__find()`. If it's `self`, the `BinarySearchTree` must be empty, so the new node becomes the root node of the tree. Otherwise, the parent is a node, so `insert()` decides which child will get the new node by comparing the new node's `key` with that of the parent. If the new key is lower, then the new node becomes the left child; otherwise, it becomes the right child. Finally, `insert()` returns `True` to indicate the insertion succeeded.

When `insert()` creates the new node, it sets the new node's right child to the `node` returned from `__find()`. You might wonder why that's there, especially because `node` can only be `None` at that point (if it were not `None`, `insert()` would have returned `False` before reaching that point). The reason goes back to what to do with duplicate keys. If you allow nodes with duplicate keys, then you must put them somewhere. The binary search tree definition says that a node's key is less than or equal to that of its right child. So, if you allow duplicate keys, the duplicate node cannot go in the left child. By specifying something other than `None` as the right child of the new node, other nodes with the same key can be retained. We leave as an exercise how to insert (and delete) nodes with duplicate keys.

Traversing the Tree

Traversing a tree means visiting each node in a specified order. Traversing a tree is useful in some circumstances such as going through all the records to look for ones that need some action (for example, parts of a vehicle that are sourced from a particular country). This process may not be as commonly used as finding, inserting, and deleting nodes but it is important nevertheless.

You can traverse a tree in three simple ways. They're called **pre-order**, **in-order**, and **post-order**. The most commonly used order for binary search trees is in-order, so let's look at that first and then return briefly to the other two.

In-order Traversal

An in-order traversal of a binary search tree causes all the nodes to be visited in *ascending order* of their key values. If you want to create a list of the data in a binary tree sorted by their keys, this is one way to do it.

The simplest way to carry out a traversal is the use of recursion (discussed in Chapter 6). A recursive method to traverse the entire tree is called with a node as an argument. Initially, this node is the root. The method needs to do only three things:

1. Call itself to traverse the node's left subtree.
2. Visit the node.
3. Call itself to traverse the node's right subtree.

Remember that *visiting* a node means doing something to it: displaying it, updating a field, adding it to a queue, writing it to a file, or whatever.

The three traversal orders work with any binary tree, not just with binary search trees. The traversal mechanism doesn't pay any attention to the key values of the nodes; it only concerns itself with the node's children and data. In other words, in-order traversal means "in order of increasing key values" only when the binary search tree criteria are used to place the nodes in the tree. The *in* of *in-order* refers to a node being visited in between the left and right subtrees. The *pre* of *pre-order* means visiting the node before visiting its children, and *post-order* visits the node after visiting the children. This distinction is like

the differences between *infix* and *postfix* notation for arithmetic expressions described in Chapter 4, "Stacks and Queues."

To see how this recursive traversal works, Figure 8-12 shows the calls that happen during an in-order traversal of a small binary tree. The `tree` variable references a four-node binary search tree. The figure shows the invocation of an `inOrderTraverse()` method on the tree that will call the `print` function on each of its nodes.

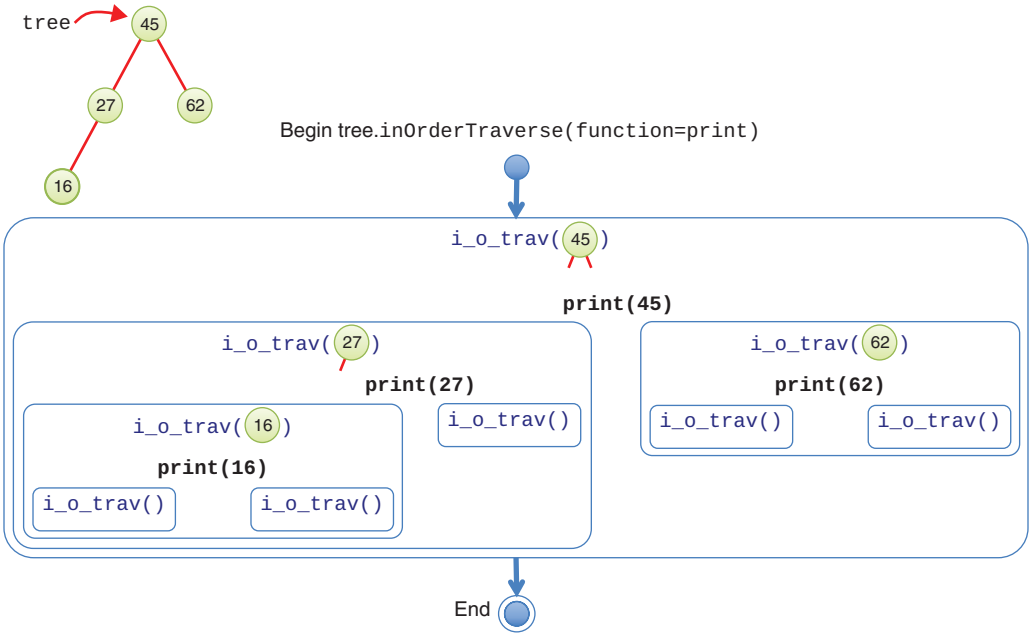


FIGURE 8-12 In-order traversal of a small tree

The blue rounded rectangles show the recursive calls on each subtree. The name of the recursive method has been abbreviated as `i_o_trav()` to fit all the calls in the figure. The first (outermost) call is on the root node (key 45). Each recursive call performs the three steps outlined previously. First, it makes a recursive call on the left subtree, rooted at key 27. That shows up as another blue rounded rectangle on the left of the figure.

Processing the subtree rooted at key 27 starts by making a recursive call on its left subtree, rooted at key 16. Another rectangle shows that call in the lower left. As before, its first task is to make a recursive call on its left subtree. That subtree is empty because it is a leaf node and is shown in the figure as a call to `i_o_trav()` with no arguments. Because the subtree is empty, nothing happens and the recursive call returns.

Back in the call to `i_o_trav(16)`, it now reaches step 2 and "visits" the node by executing the function, `print`, on the node itself. This is shown in the figure as `print(16)` in black. In general, visiting a node would do more than just print the node's key; it would take some action on the data stored at the node. The figure doesn't show that action, but it would occur when the `print(16)` is executed.

After visiting the node with key 16, it's time for step 3: call itself on the right subtree. The node with key 16 has no right child, which shows up as the smallest-sized rectangle because it is a call on an empty subtree. That completes the execution for the subtree rooted at key 16. Control passes back to the caller, the call on the subtree rooted at key 27.

The rest of the processing progresses similarly. The visit to the node with key 27 executes `print(27)` and then makes a call on its empty right subtree. That finishes the call on node 27 and control passes back to the call on the root of the tree, node 45. After executing `print(45)`, it makes a call to traverse its right (nonempty) subtree. This is the fourth and final node in the tree, node 62. It makes a call on its empty left subtree, executes `print(62)`, and finishes with a call on its empty right subtree. Control passes back up through the call on the root node, 45, and that ends the full tree traversal.

Pre-order and Post-order Traversals

The other two traversal orders are similar: only the sequence of visiting the node changes. For pre-order traversal, the node is visited first, and for post-order, it's visited last. The two subtrees are always visited in the same order: left and then right. Figure 8-13 shows the execution of a pre-order traversal on the same four-node tree as in Figure 8-12. The execution of the `print()` function happens before visiting the two subtrees. That means that the pre-order traversal would print 45, 27, 16, 62 compared to the in-order traversal's 16, 27, 45, 62. As the figures show, the differences between the orders are small, but the overall effect is large.

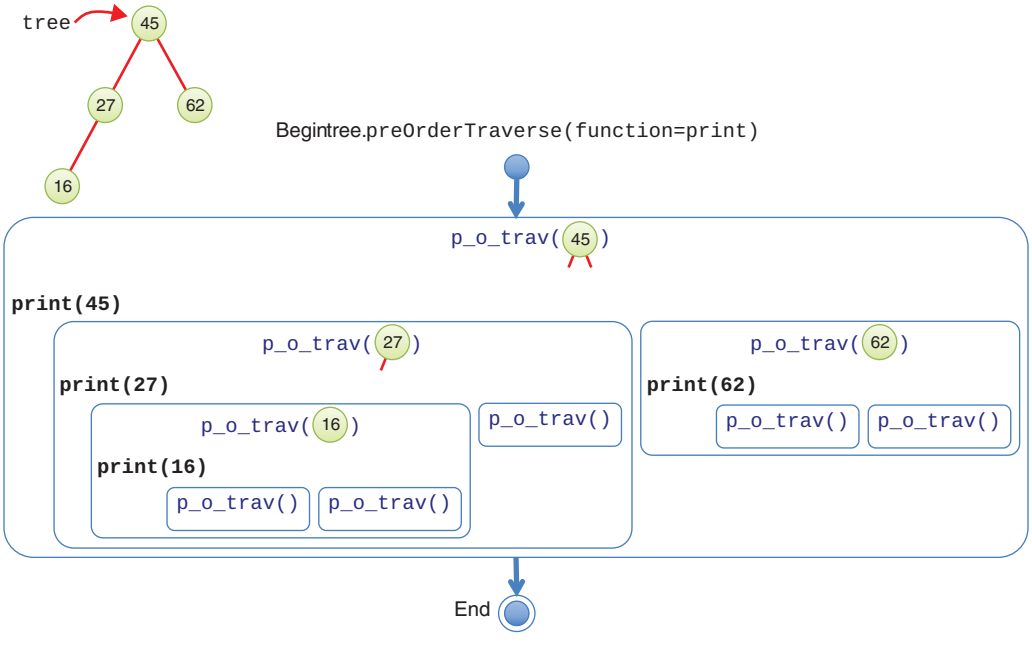


FIGURE 8-13 Pre-order traversal of a small tree

Python Code for Traversing

Let's look at a simple way of implementing the in-order traversal now. As you saw in stacks, queues, linked lists, and other data structures, it's straightforward to define the traversal using a function passed as an argument that gets applied to each item stored in the structure. The interesting difference with trees is that recursion makes it very compact.

Because these trees are represented using two classes, `BinarySearchTree` and `__Node`, you need methods that can operate on both types of objects. In Listing 8-5, the `inOrderTraverse()` method handles the traversal on `BinarySearchTree` objects. It serves as the public interface to the traversal and calls a private method `__inOrderTraverse()` to do the actual work on subtrees. It passes the root node to the private method and returns.

LISTING 8-5 Recursive Implementation of `inOrderTraverse()`

```
class BinarySearchTree(object):          # A binary search tree class
…
   def inOrderTraverse(                  # Visit all nodes of the tree in-order
         self, function=print):          # and apply a function to each node
      self.__inOrderTraverse(            # Call recursive version starting at
         self.__root, function=function) # root node

   def __inOrderTraverse(                # Visit a subtree in-order, recursively
         self, node, function):          # The subtree's root is node
      if node:                           # Check that this is real subtree
         self.__inOrderTraverse(         # Traverse the left subtree
            node.leftChild, function)
         function(node)                  # Visit this node
         self.__inOrderTraverse(         # Traverse the right subtree
            node.rightChild, function)
```

The private method expects a `__Node` object (or `None`) for its `node` parameter and performs the three steps on the subtree rooted at the node. First, it checks `node` and returns immediately if it is `None` because that signifies an empty subtree. For legitimate nodes, it first makes a recursive call to itself to process the left child of the node. Second, it visits the `node` by invoking the `function` on it. Third, it makes a recursive call to process the node's right child. That's all there is to it.

Using a Generator for Traversal

The `inOrderTraverse()` method is straightforward, but it has at least three shortcomings. First, to implement the other orderings, you would either need to write more methods or add a parameter that specifies the ordering to perform.

Second, the `function` passed as an argument to "visit" each node needs to take a `__Node` object as argument. That's a private class inside the `BinarySearchTree` that protects the nodes from being manipulated by the caller. One alternative that avoids passing a reference to a `__Node` object would be to pass in only the data field (and maybe the key field)

of each node as arguments to the visit `function`. That approach would minimize what the caller could do to the node and prevent it from altering the other node references.

Third, using a function to describe the action to perform on each node has its limitations. Typically, functions perform the same operation each time they are invoked and don't know about the history of previous calls. During the traversal of a data structure like a tree, being able to make use of the results of previous node visits dramatically expands the possible operations. Here are some examples that you might want to do:

- Add up all the values in a particular field at every node.
- Get a list of all the unique strings in a field from every node.
- Add the node's key to some list if none of the previously visited nodes have a bigger value in some field.

In all these traversals, it's very convenient to be able to accumulate results somewhere during the traversal. That's possible to do with functions, but generators make it easier. We introduced generators in Chapter 5, and because trees share many similarities with those structures, they are very useful for traversing trees.

We address these shortcomings in a recursive generator version of the traverse method, `traverse_rec()`, shown in Listing 8-6. This version adds some complexity to the code but makes using traversal much easier. First, we add a parameter, `traverseType`, to the `traverse_rec()` method so that we don't need three separate traverse routines. The first `if` statement verifies that this parameter is one of the three supported orderings: `pre`, `in`, and `post`. If not, it raises an exception. Otherwise, it launches the recursive private method, `__traverse()`, starting with the root node, just like `inOrderTraverse()` does.

There is an important but subtle point to note in calling the `__traverse()` method. The public `traverse_rec()` method returns the result of calling the private `__traverse()` method and does not just simply call it as a subroutine. The reason is that the `traverse()` method itself is not the generator; it has no `yield` statements. It must return the iterator produced by the call to `__traverse()`, which will be used by the `traverse_rec()` caller to iterate over the nodes.

Inside the `__traverse()` method, there are a series of `if` statements. The first one tests the base case. If `node` is `None`, then this is an empty tree (or subtree). It returns to indicate the iterator has hit the end (which Python converts to a `StopIteration` exception). The next `if` statement checks whether the traversal type is pre-order, and if it is, it yields the node's key and data. Remember that the iterator will be paused at this point while control passes back to its caller. That is where the node will be "visited." After the processing is done, the caller's loop invokes this iterator to get the next node. The iterator resumes processing right after the `yield` statement, remembering all the context.

When the iterator resumes (or if the order was something other than pre-order), the next step is a `for` loop. This is a recursive generator to perform the traversal of the left subtree. It calls the `__traverse()` method on the node's `leftChild` using the same `traverseType`. That creates its own iterator to process the nodes in that subtree. As nodes are yielded

back as `key`, `data` pairs, this higher-level iterator yields them back to its caller. This loop construction produces a nested stack of iterators, similar to the nested invocations of `i_o_trav()` shown in Figure 8-12. When each iterator returns at the end of its work, it raises a `StopIteration`. The enclosing iterator catches each exception, so the various levels don't interfere with one another.

LISTING 8-6 The Recursive Generator for Traversal

```
class BinarySearchTree(object):            # A binary search tree class
…
   def traverse_rec(self,                  # Traverse the tree recursively in
                    traverseType="in"):    # pre, in, or post order
      if traverseType in [                 # Verify type is an accepted value and
            'pre', 'in', 'post']:          # use generator to walk the tree
         return self.__traverse(           # yielding (key, data) pairs
            self.__root, traverseType)     # starting at root

      raise ValueError("Unknown traversal type: " + str(traverseType))

   def __traverse(self,                    # Recursive generator to traverse
                  node,                    # subtree rooted at node in pre, in, or
                  traverseType):           # post order
      if node is None:                     # If subtree is empty,
         return                            # traversal is done
      if traverseType == "pre":            # For pre-order, yield the current
         yield (node.key, node.data)       # node before all the others
      for childKey, childData in self.__traverse( # Recursively
            node.leftChild, traverseType): # traverse the left subtree
         yield (childKey, childData)       # yielding its nodes
      if traverseType == "in":             # If in-order, now yield the current
         yield (node.key, node.data)       # node
      for childKey, childData in self.__traverse( # Recursively
            node.rightChild, traverseType):# traverse right subtree
         yield (childKey, childData)       # yielding its nodes
      if traverseType == "post":           # If post-order, yield the current
         yield (node.key, node.data)       # node after all the others
```

The rest of the `__traverse()` method is straightforward. After finishing the loop over all the nodes in the left subtree, the next `if` statement checks for the in-order traversal type and yields the node's key and data, if that's the ordering. The node gets processed between the left and right subtrees for an in-order traversal. After that, the right subtree is processed in its own loop, yielding each of the visited nodes back to its caller. After the right subtree is done, a check for post-order traversal determines whether the node should be yielded at this stage or not. After that, the `__traverse()` generator is done, ending its caller's loop.

Making the Generator Efficient

The recursive generator has the advantage of structural simplicity. The base cases and recursive calls follow the node and child structure of the tree. Developing the prototype and proving its correct behavior flow naturally from this structure.

The generator does, however, suffer some inefficiency in execution. Each invocation of the `__traverse()` method invokes two loops: one for the left and one for the right child. Each of those loops creates a new iterator to yield the items from their subtrees back through the iterator created by this invocation of the `__traverse()` method itself. That layering of iterators extends from the root down to each leaf node.

Traversing the N items in the tree should take O(N) time, but creating a stack of iterators from the root down to each leaf adds complexity that's proportional to the depth of the leaves. The leaves are at O(log N) depth, in the best case. That means the overall traversal of N items will take O(N×log N) time.

To achieve O(N) time, you need to apply the method discussed at the end of Chapter 6 and use a stack to hold the items being processed. The items include both `Node` structures and the (key, data) pairs stored at the nodes to be traversed in a particular order. Listing 8-7 shows the code.

The nonrecursive method combines the two parts of the recursive approach into a single `traverse()` method. The same check for the validity of the traversal type happens at the beginning. The next step creates a stack, using the `Stack` class built on a linked list from Chapter 5 (defined in the `LinkStack` module).

Initially, the method pushes the root node of the tree on the stack. That means the remaining work to do is the entire tree starting at the root. The `while` loop that follows works its way through the remaining work until the stack is empty.

At each pass through the `while` loop, the top item of the stack is popped off. Three kinds of items could be on the stack: a `Node` item, a (key, data) tuple, or `None`. The latter happens if the tree is empty and when it processes the leaf nodes (and finds their children are `None`).

If the top of the stack is a `Node` item, the `traverse()` method determines how to process the node's data and its children based on the requested traversal order. It pushes items onto the stack to be processed on subsequent passes through the `while` loop. Because the items will be popped off the stack in the reverse order from the way they were pushed onto it, it starts by handling the case for post-order traversal.

In post-order, the first item pushed is the node's (key, data) tuple. Because it is pushed first, it will be processed last overall. The next item pushed is the node's right child. In post-order, this is traversed just before processing the node's data. For the other orders, the right child is always the last node processed.

After pushing on the right child, the next `if` statement checks whether the in-order traversal was requested. If so, it pushes the node's (key, data) tuple on the stack to be processed in-between the two child nodes. That's followed by pushing the left child on the stack for processing.

LISTING 8-7 The Nonrecursive `traverse()` Generator

```
from LinkStack import *

class BinarySearchTree(object):          # A binary search tree class
…
   def traverse(self,                    # Non-recursive generator to traverse
                traverseType='in'):      # tree in pre, in, or post order
      if traverseType not in [           # Verify traversal type is an
            'pre', 'in', 'post']:        # accepted value
         raise ValueError(
            "Unknown traversal type: " + str(traverseType))

      stack = Stack()                    # Create a stack
      stack.push(self.__root)            # Put root node in stack

      while not stack.isEmpty():         # While there is work in the stack
         item = stack.pop()              # Get next item
         if isinstance(item, self.__Node): # If it's a tree node
            if traverseType == 'post':   # For post-order, put it last
               stack.push((item.key, item.data))
            stack.push(item.rightChild) # Traverse right child
            if traverseType == 'in':     # For pre-order, put item 2nd
               stack.push((item.key, item.data))
            stack.push(item.leftChild)  # Traverse left child
            if traverseType == 'pre':    # For pre-order, put item 1st
               stack.push((item.key, item.data))
         elif item:                      # Every other non-None item is a
            yield item                   # (key, value) pair to be yielded
```

Finally, the last `if` statement checks whether the pre-order traversal was requested and then pushes the node's data on the stack for processing before the left and right children. It will be popped off during the next pass through the `while` loop. That completes all the work for a `Node` item.

The final `elif` statement checks for a non-`None` item on the stack, which must be a (key, data) tuple. When the loop finds such a tuple, it yields it back to the caller. The `yield` statement ensures that the `traverse()` method becomes a generator, not a function.

The loop doesn't have any explicit handling of the `None` values that get pushed on the stack for empty root and child links. The reason is that there's nothing to do for them: just pop them off the stack and continue on to the remaining work.

Using the stack, you have now made an O(N) generator. Each node of the tree is visited exactly once, pushed on the stack, and later popped off. Its key-data pairs and child links are also pushed on and popped off exactly once. The ordering of the node visits and child links follows the requested traversal ordering. Using the stack and carefully reversing the

items pushed onto it make the code slightly more complex to understand but improve the performance.

Using the Generator for Traversing

The generator approach (both recursive and stack-based) makes the caller's loops easy. For example, if you want to collect all the items in a tree whose data is below the average data value, you could use two loops:

```
total, count = 0, 0
for key, data in random_tree.traverse('pre'):
   total += data
   count += 1
average = total / count
below_average = []
for key, data in random_tree.traverse('in'):
   if data <= average:
      below_average.append((key, data))
```

The first loop counts the number of items in `random_tree` and sums up their data values. The second loop finds all the items whose data is below the average and appends the key and data pair to the `below_average` list. Because the second loop is done in in-order, the keys in `below_average` are in ascending order. Being able to reference the variables that accumulate results—`total`, `count`, and `below_average`—without defining some global (or nonlocal) variables outside a function body, makes using the generator very convenient for traversal.

Traversing with the Visualization Tool

The Binary Search Tree Visualization tool allows you to explore the details of traversal using generators. You can launch any of the three kinds of traversals by selecting the Pre-order Traverse, In-order Traverse, or Post-order Traverse buttons. In each case, the tool executes a simple loop of the form

```
for key, data in tree.traverse("pre"):
   print(key)
```

To see the details, use the Step button (you can launch an operation in step mode by holding down the Shift key when selecting the button). In the code window, you first see the short traversal loop. The example calls the `traverse()` method to visit all the keys and data in a loop using one of the orders such as `pre`.

Figure 8-14 shows a snapshot near the beginning of a pre-order traversal. The code for the `traverse()` method appears at the lower right. To the right of the tree above the code, the `stack` is shown. The nodes containing keys 59 and 94 are on the stack. The top of the stack was already popped off and moved to the top right under the `item` label. It shows the key, 77, with a comma separating it from its colored rectangle to represent the (key, data) tuple that was pushed on the stack. The `yield` statement is highlighted, showing that the `traverse()` iterator is about to yield the key and data back to caller. The loop that called `traverse()` has scrolled off the code display but will be shown on the next step.

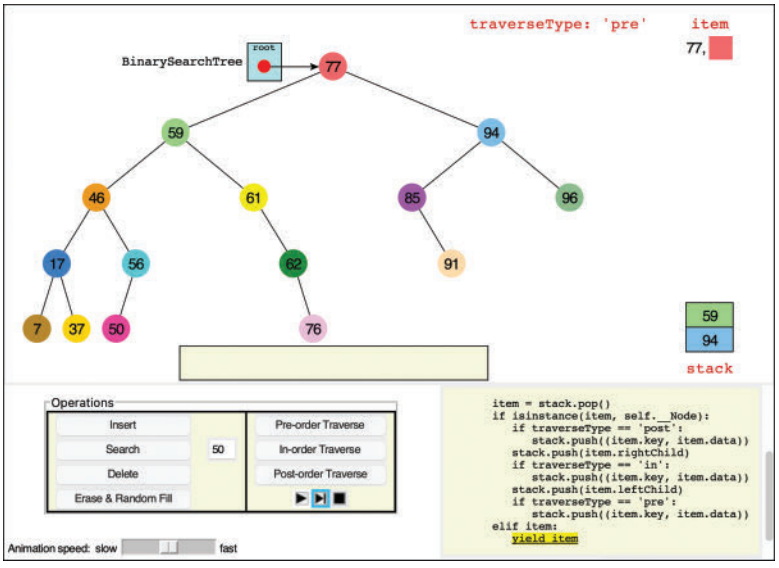


FIGURE 8-14 Traversing a tree in pre-order using the `traverse()` iterator

When control returns to the calling loop, the `traverse()` iterator disappears from the code window and so does the stack, as shown in Figure 8-15. The `key` and `data` variables are now bound to 77 and the root node's data. The `print` statement is highlighted because the program is about to print the key in the output box along the bottom of the tree. The next step shows key 77 being copied to the output box.

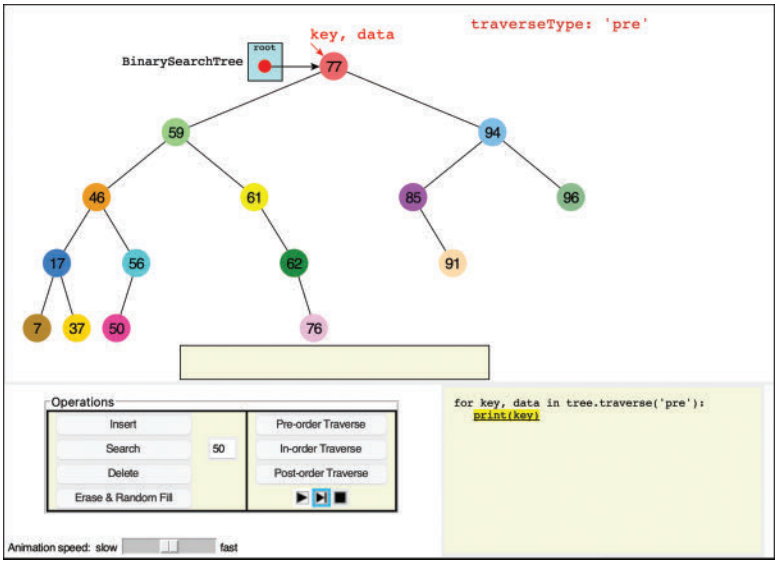


FIGURE 8-15 The loop calling the `traverse()` iterator

After printing, control returns to the `for key, data in tree.traverse('pre')` loop. That pushes the `traverse()` iterator back on the code display, along with its stack similar to Figure 8-14. The `while` loop in the iterator finds that the stack is not empty, so it pops off the top item. That item is node 59, the left child of node 77. The process repeats by pushing on node 59's children and the node's key, data pair on the stack. On the next loop iteration, that tuple is popped off the stack, and it is yielded back to the print loop.

The processing of iterators is complex to describe, and the Visualization tool makes it easier to follow the different levels and steps than reading a written description. Try stepping through the processing of several nodes, including when the iterator reaches a leaf node and pushes `None` on the stack. The stack guides the iterator to return to nodes that remain to be processed.

Traversal Order

What's the point of having three traversal orders? One advantage is that in-order traversal guarantees an ascending order of the keys in binary search trees. There's a separate motivation for pre- and post-order traversals. They are very useful if you're writing programs that *parse* or analyze algebraic expressions. Let's see why that is the case.

A binary tree (not a binary search tree) can be used to represent an algebraic expression that involves binary arithmetic operators such as `+`, `–`, `/`, and `*`. The root node and every nonleaf node hold an operator. The leaf nodes hold either a variable name (like A, B, or C) or a number. Each subtree is a valid algebraic expression.

For example, the binary tree shown in Figure 8-16 represents the algebraic expression

(A+B) * C – D / E

This is called **infix notation**; it's the notation normally used in algebra. (For more on infix and postfix, see the section "Parsing Arithmetic Expressions" in Chapter 4.) Traversing the tree in order generates the correct in-order sequence A+B*C–D/E, but you need to insert the parentheses yourself to get the expected order of operations. Note that subtrees form their own subexpressions like the (A+B) * C outlined in the figure.

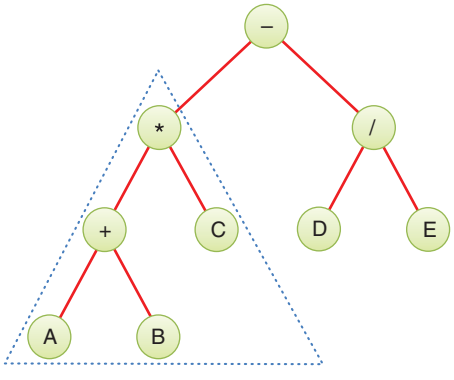


FIGURE 8-16 Binary tree representing an algebraic expression

What does all this have to do with pre-order and post-order traversals? Let's see what's involved in performing a pre-order traversal. The steps are

1. Visit the node.
2. Call itself to traverse the node's left subtree.
3. Call itself to traverse the node's right subtree.

Traversing the tree shown in Figure 8-16 using pre-order and printing the node's value would generate the expression

–*+ABC/DE

This is called **prefix notation**. It may look strange the first time you encounter it, but one of its nice features is that parentheses are never required; the expression is unambiguous without them. Starting on the left, each operator is applied to the next two things to its right in the expression, called the **operands**. For the first operator, –, these two things are a product expression, *+ABC, and a division expression, /DE. For the second operator, *, the two things are a sum expression, +AB, and a single variable, C. For the third operator, +, the two things it operates on are the variables, A and B, so this last expression would be A+B in in-order notation. Finally, the fourth operator, /, operates on the two variables D and E.

The third kind of traversal, post-order, contains the three steps arranged in yet another way:

1. Call itself to traverse the node's left subtree.
2. Call itself to traverse the node's right subtree.
3. Visit the node.

For the tree in Figure 8-16, visiting the nodes with a post-order traversal generates the expression

AB+C*DE/–

This is called **postfix notation**. It means "apply the last operator in the expression, –, to the two things immediately to the left of it." The first thing is AB+C*, and the second thing is DE/. Analyzing the first thing, AB+C*, shows its meaning to be "apply the * operator to the two things immediately to the left of it, AB+ and C." Analyzing the first thing of that expression, AB+, shows its meaning to be "apply the + operator to the two things immediately to the left of it, A and B." It's hard to see initially, but the "things" are always one of three kinds: a single variable, a single number, or an expression ending in a binary operator.

To process the meaning of a postfix expression, you start from the last character on the right and interpret it as follows. If it's a binary operator, then you repeat the process to

interpret two subexpressions on its left, which become the operands of the operator. If it's a letter, then it's a simple variable, and if it's a number, then it's a constant. For both variables and numbers, you "pop" them off the right side of the expression and return them to the process of the enclosing expression.

We don't show the details here, but you can easily construct a tree like that in Figure 8-16 by using a postfix expression as input. The approach is analogous to that of evaluating a postfix expression, which you saw in the `PostfixTranslate.py` program in Chapter 4 and its corresponding InfixCalculator Visualization tool. Instead of storing operands on the stack, however, you store entire subtrees. You read along the postfix string from left to right as you did in the `PostfixEvaluate()` method. Here are the steps when you encounter an operand (a variable or a number):

1. Make a tree with one node that holds the operand.
2. Push this tree onto the stack.

Here are the steps when you encounter an operator, O:

1. Pop two operand trees R and L off the stack (the top of the stack has the rightmost operand, R).
2. Create a new tree T with the operator, O, in its root.
3. Attach R as the right child of T.
4. Attach L as the left child of T.
5. Push the resulting tree, T, back on the stack.

When you're done evaluating the postfix string, you pop the one remaining item off the stack. Somewhat amazingly, this item is a complete tree depicting the algebraic expression. You can then see the prefix and infix representations of the original postfix notation (and recover the postfix expression) by traversing the tree in one of the three orderings we described. We leave an implementation of this process as an exercise.

Finding Minimum and Maximum Key Values

Incidentally, you should note how easy it is to find the minimum and maximum key values in a binary search tree. In fact, this process is so easy that we don't include it as an option in the Visualization tool. Still, understanding how it works is important.

For the minimum, go to the left child of the root; then go to the left child of that child, and so on, until you come to a node that has no left child. This node is the minimum. Similarly, for the maximum, start at the root and follow the right child links until they end. That will be the maximum key in the tree, as shown in Figure 8-17.

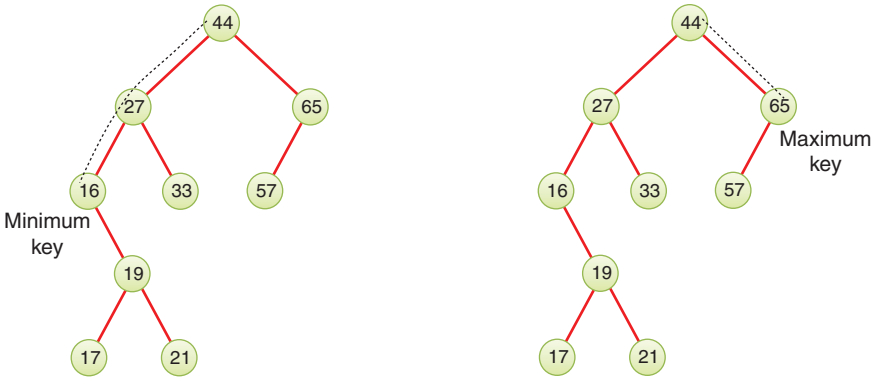


FIGURE 8-17 Minimum and maximum key values of a binary search tree

Here's some code that returns the minimum node's data and key values:

```
def minNode(self):                  # Find and return node with minimum key
   if self.isEmpty():               # If the tree is empty, raise exception
      raise Exception("No minimum node in empty tree")
   node = self.__root               # Start at root
   while node.leftChild:            # While node has a left child,
      node = node.leftChild         # follow left child reference
   return (node.key, node.data) # return final node key and data
```

Finding the maximum is similar; just swap the right for the left child. You learn about an important use of finding the minimum value in the next section about deleting nodes.

Deleting a Node

Deleting a node is the most complicated common operation required for binary search trees. The fundamental operation of deletion can't be ignored, however, and studying the details builds character. If you're not in the mood for character building, feel free to skip to the Efficiency of Binary Search Trees section.

You start by verifying the tree isn't empty and then finding the node you want to delete, using the same approach you saw in `__find()` and `insert()`. If the node isn't found, then you're done. When you've found the node and its parent, there are three cases to consider:

1. The node to be deleted is a leaf (has no children).
2. The node to be deleted has one child.
3. The node to be deleted has two children.

Let's look at these three cases in turn. The first is easy; the second, almost as easy; and the third, quite complicated.

Case 1: The Node to Be Deleted Has No Children

To delete a leaf node, you simply change the appropriate child field in the node's parent to `None` instead of to the node. The node object still exists, but it is no longer part of the tree, as shown when deleting node 17 in Figure 8-18.

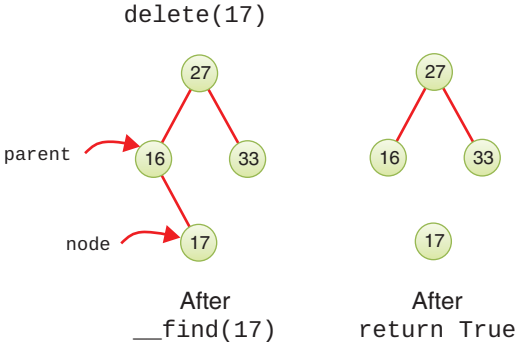


FIGURE 8-18 Deleting a node with no children

If you're using a language like Python or Java that has garbage collection, the deleted node's memory will eventually be reclaimed for other uses (if you eliminate all references to it in the program). In languages that require explicit allocation and deallocation of memory, the deleted node should be released for reuse.

Using the Visualization Tool to Delete a Node with No Children

Try deleting a leaf node using the Binary Search Tree Visualization tool. You can either type the key of a node in the text entry box or select a leaf with your pointer device and then select Delete. You see the program use `__find()` to locate the node by its key, copy it to a temporary variable, set the parent link to `None`, and then "return" the deleted key and data (in the form of its colored background).

Case 2: The Node to Be Deleted Has One Child

This second case isn't very difficult either. The node has only two edges: one to its parent and one to its only child. You want to "cut" the node out of this sequence by connecting its parent directly to its child. This process involves changing the appropriate reference in the parent (`leftChild` or `rightChild` or `__root`) to point to the deleted node's child. Figure 8-19 shows the deletion of node 16, which has only one child.

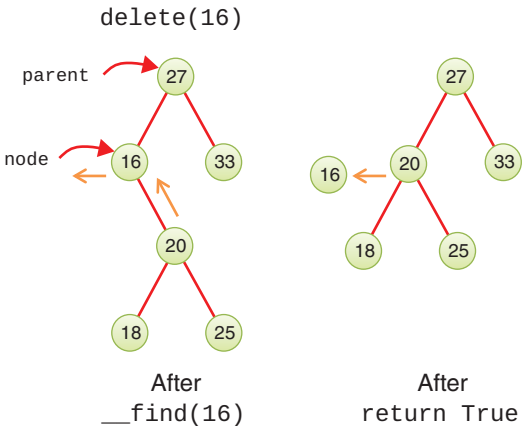


FIGURE 8-19 Deleting a node with one child

After finding the node and its parent, the delete method has to change only one reference. The deleted node, key 16 in the figure, becomes disconnected from the tree (although it may still have a child pointer to the node that was promoted up (key 20). Garbage collectors are sophisticated enough to know that they can reclaim the deleted node without following its links to other nodes that might still be needed.

Now let's go back to the case of deleting a node with no children. In that case, the delete method also made a single change to replace one of the parent's child pointers. That pointer was set to `None` because there was no replacement child node. That's a similar operation to Case 2, so you can treat Case 1 and Case 2 together by saying, "If the node to be deleted, D, has 0 or 1 children, replace the appropriate link in its parent with either the left child of D, if it isn't empty, or the right child of D." If both child links from D are `None`, then you've covered Case 1. If only one of D's child links is non-`None`, then the appropriate child will be selected as the parent's new child, covering Case 2. You promote either the single child or `None` into the parent's child (or possibly `__root`) reference.

Using the Visualization Tool to Delete a Node with One Child

Let's assume you're using the Visualization tool on the tree in Figure 8-5 and deleting node 61, which has a right child but no left child. Click node 61 and the key should appear in the text entry area, enabling the Delete button. Selecting the button starts another call to `__find()` that stops with `current` pointing to the node and `parent` pointing to node 59.

After making a copy of node 61, the animation shows the right child link from node 59 being set to node 61's right child, node 62. The original copy of node 61 goes away, and the tree is adjusted to put the subtree rooted at node 62 into its new position. Finally, the copy of node 61 is moved to the output box at the bottom.

Use the Visualization tool to generate new trees with single child nodes and see what happens when you delete them. Look for the subtree whose root is the deleted node's child. No matter how complicated this subtree is, it's simply moved up and plugged in as the new child of the deleted node's parent.

Python Code to Delete a Node

Let's now look at the code for at least Cases 1 and 2. Listing 8-8 shows the code for the `delete()` method, which takes one argument, the key of the node to delete. It returns either the data of the node that was deleted or `None`, to indicate the node was not found. That makes it behave somewhat like the methods for popping an item off a stack or deleting an item from a queue. The difference is that the node must be found inside the tree instead of being at a known position in the data structure.

LISTING 8-8 The `delete()` Method of `BinarySearchTree`

```
class BinarySearchTree(object):              # A binary search tree class
…
   def delete(self, goal):                   # Delete a node whose key matches goal
      node, parent = self.__find(goal) # Find goal and its parent
      if node is not None:                   # If node was found,
         return self.__delete(               # then perform deletion at node
            parent, node)                    # under the parent

   def __delete(self,                        # Delete the specified node in the tree
                parent, node):               # modifying the parent node/tree
      deleted = node.data                    # Save the data that's to be deleted
      if node.leftChild:                     # Determine number of subtrees
         if node.rightChild:                 # If both subtrees exist,
            self.__promote_successor(        # Then promote successor to
               node)                         # replace deleted node
         else:                     # If no right child, move left child up
            if parent is self:               # If parent is the whole tree,
               self.__root = node.leftChild # update root
            elif parent.leftChild is node:  # If node is parent's left,
               parent.leftChild = node.leftChild # child, update left
            else:                            # else update right child
               parent.rightChild = node.leftChild
      else:                                  # No left child; so promote right child
         if parent is self:                  # If parent is the whole tree,
            self.__root = node.rightChild    # update root
         elif parent.leftChild is node:      # If node is parent's left
            parent.leftChild = node.rightChild # child, then update
         else:                               # left child link else update
            parent.rightChild = node.rightChild # right child
      return deleted                         # Return the deleted node's data
```

Just like for insertion, the first step is to find the node to delete and its parent. If that search does not find the `goal` node, then there's nothing to delete from the tree, and `delete()` returns `None`. If the node to delete is found, the node and its parent are passed to the private `__delete()` method to modify the nodes in the tree.

Inside the `__delete()` method, the first step is to store a reference to the node data being `deleted`. This step enables retrieval of the node's data after the references to it are removed from the tree. The next step checks how many subtrees the node has. That determines what case is being processed. If both a left and a right child are present, that's Case 3, and it hands off the deletion to another private method, `__promote_successor()`, which we describe a little later.

If there is only a left subtree of the node to delete, then the next thing to look at is its parent node. If the parent is the `BinarySearchTree` object (`self`), then the node to delete must be the root node, so the left child is promoted into the root node slot. If the parent's left child is the node to delete, then the parent's left child link is replaced with the node's left child to remove the node. Otherwise, the parent's right child link is updated to remove the node.

Notice that working with references makes it easy to move an entire subtree. When the parent's reference to the `node` is updated, the child that gets promoted could be a single node or an immense subtree. Only one reference needs to change. Although there may be lots of nodes in the subtree, you don't need to worry about moving them individually. In fact, they "move" only in the sense of being conceptually in different positions relative to the other nodes. As far as the program is concerned, only the parent's reference to the root of the subtree has changed, and the rest of the contents in memory remain the same.

The final `else` clause of the `__delete()` method deals with the case when the `node` has no left child. Whether or not the `node` has a right child, `__delete()` only needs to update the parent's reference to point at the node's right child. That handles both Case 1 and Case 2. It still must determine which field of the parent object gets the reference to the node's right child, just as in the earlier lines when only the left child was present. It puts the `node.rightChild` in either the `__root`, `leftChild`, or `rightChild` field of the parent, accordingly. Finally, it returns the data of the node that was deleted.

Case 3: The Node to Be Deleted Has Two Children

Now the fun begins. If the deleted node has two children, you can't just replace it with one of these children, at least if the child has its own (grand) children. Why not? Examine Figure 8-20 and imagine deleting node 27 and replacing it with its right subtree, whose root is 33. You are promoting the right subtree, but it has its own children. Which left child would node 33 have in its new position, the deleted node's left child, 16, or node 33's left child, 28? And what do you do with the other left child? You can't just throw it away.

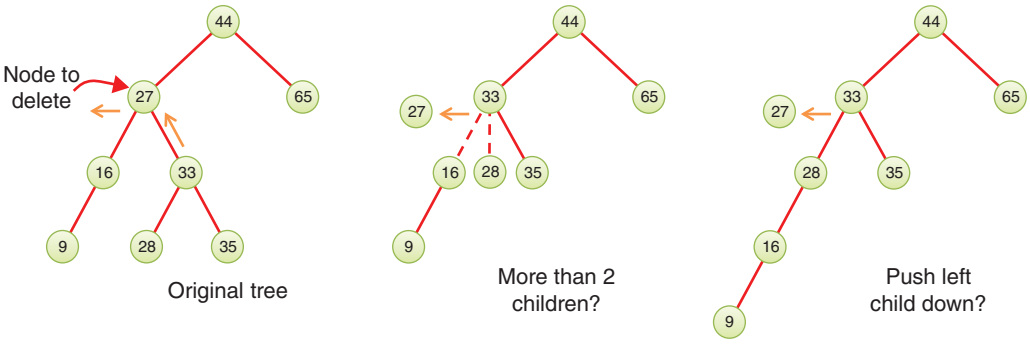


FIGURE 8-20 Options for deleting a node with two subtrees

The middle option in Figure 8-20 shows potentially allowing three children. That would bring a whole host of other problems because the tree is no longer binary (see Chapter 9 for more on that idea). The right-hand option in the figure shows pushing the deleted node's left child, 16, down and splicing in the new node's left child, 28, above it. That approach looks plausible. The tree is still a binary search tree, at least. The problem, however, is what to do if the promoted node's left child has a complicated subtree of its own (for example, if node 28 in the figure had a whole subtree below it). That could mean following a long path to figure out where to splice the left subtrees together.

We need another approach. The good news is that there's a trick. The bad news is that, even with the trick, there are special cases to consider. Remember that, in a binary search tree, the nodes are arranged in order of ascending keys. For each node, the node with the next-highest key is called its **in-order successor**, or simply its **successor**. In the original tree of Figure 8-20, node 28 is the in-order successor of node 27.

Here's the trick: To delete a node with two children, *replace the node with its in-order successor*. Figure 8-21 shows a deleted node being replaced by its successor. Notice that the nodes are still in order. All it took was a simple replacement. It's going to be a little more complicated if the successor itself has children; we look at that possibility in a moment.

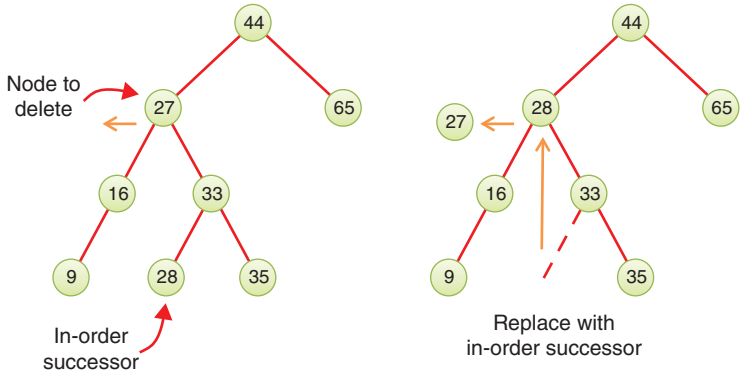


FIGURE 8-21 Node replaced by its successor

Finding the Successor

How do you find the successor of a node? Human beings can do this quickly (for small trees, anyway). Just take a quick glance at the tree and find the next-largest number following the key of the node to be deleted. In Figure 8-21 it doesn't take long to see that the successor of 27 is 28, or that the successor of 35 is 44. The computer, however, can't do things "at a glance"; it needs an algorithm.

Remember finding the node with the minimum or maximum key? In this case you're looking for the *minimum key larger than* the key to be deleted. The node to be deleted has both a left and right subtree because you're working on Case 3. So, you can just look for the minimum key in the right subtree, as illustrated in Figure 8-22. All you need to do is follow the left child links until you find a node with no left child.

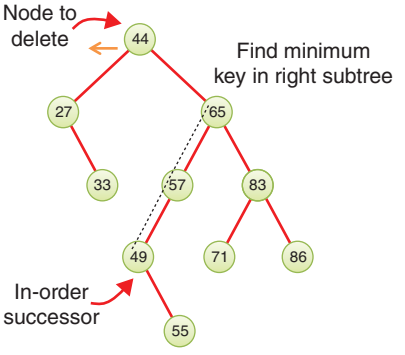


FIGURE 8-22 Finding the successor

What about potential nodes in the trees rooted above the node to be deleted? Couldn't the successor be somewhere in there? Let's think it through. Imagine you seek the successor of node 27 in Figure 8-22. The successor would have to be greater than 27 and less than 33, the key of its right child. Any node with a key between those two values would be inserted somewhere in the left subtree of node 33. Remember that you always search down the binary search tree choosing the path based on the key's relative order to the keys already in the tree. Furthermore, node 33 was placed as the right child of node 27 because it was less than the root node, 44. Any node's right child key must be less than its parent's key if it is the left child of that parent. So going up to parent, grandparent, or beyond (following left child links) only leads to larger keys, and those keys can't be the successor.

There are a couple of other things to note about the successor. If the right child of the original node to delete has no left children, this right child is itself the successor, as shown in the example of Figure 8-23. Because the successor always has an empty left child link, it has at most one child.

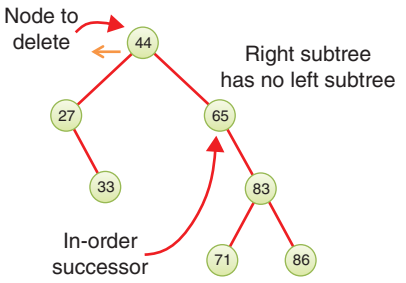


FIGURE 8-23 The right child is the successor

Replacing with the Successor

Having found the successor, you can easily copy its key and data values into the node to be deleted, but what do you do with the subtree rooted at the successor node? You can't leave a copy of the successor node in the tree there because the data would be stored in two places, create duplicate keys, and make deleting the successor a problem in the future. So, what's the easiest way to get it out of the tree?

Hopefully, reading Chapter 6 makes the answer jump right out. You can now delete the successor from the tree using a recursive call. You want to do the same operation on the successor that you've been doing on the original node to delete—the one with the `goal` key. What's different is that you only need to do the deletion in a smaller tree, the right subtree where you found the successor. If you tried to do it starting from the root of the tree after replacing the `goal` node, the `__find()` method would follow the same path and end at the node you just replaced. You could get around that problem by delaying the replacement of the key until after deleting the successor, but it's much easier—and more importantly, faster—if you start a new delete operation in the right subtree. There will be much less tree to search, and you can't accidentally end up at the previous `goal` node.

In fact, when you searched for the successor, you followed child links to determine the path, and that gave you both the successor and the successor's parent node. With those two references available, you now have everything needed to call the private `__delete()` method shown in Listing 8-8. You can now define the `__promote_successor()` method, as shown in Listing 8-9. Remember, this is the method used to handle Case 3—when the node to delete has two children.

The `__promote_successor()` method takes as its lone parameter the `node` to delete. Because it is going to replace that node's data and key and then delete the successor, it's easier to refer to it as the node to be replaced in this context. To start, it points a `successor` variable at the right child of the `node` to be replaced. Just like the `__find()` method, it tracks the `parent` of the successor node, which is initialized to be the node to be replaced. Then it acts like the `minNode()` method, using a `while` loop to update `successor` with its left child if there is a left child. When the loop exits, `successor` points at the successor node and `parent` to its parent node.

LISTING 8-9 The `__promote_successor()` Method of `BinarySearchTree`

```
class BinarySearchTree(object):        # A binary search tree class
…
   def __promote_successor(            # When deleting a node with both subtrees,
         self,                         # find successor on the right subtree, put
                                       # its data in this node, and delete the
         node):                        # successor from the right subtree
      successor = node.rightChild      # Start search for successor in
      parent = node                    # right subtree and track its parent
      while successor.leftChild:       # Descend left child links until
         parent = successor            # no more left links, tracking parent
         successor = successor.leftChild
      node.key = successor.key         # Replace node to delete with
      node.data = successor.data       # successor's key and data
      self.__delete(parent, successor) # Remove successor node
```

All that's left to do is update the key and data of the node to be replaced and delete the successor node using a recursive call to `__delete()`. Unlike previous recursive methods you've seen, this isn't a call to the same routine where the call occurs. In this case, the `__promote_successor()` method calls `__delete()`, which in turn, could call `__promote_successor()`. This is called **mutual recursion**—where two or more routines call each other.

Your senses should be tingling now. How do you know this mutual recursion will end? Where's the base case that you saw with the "simple" recursion routines? Could you get into an infinite loop of mutually recursive calls? That's a good thing to worry about, but it's not going to happen here. Remember that deleting a node broke down into three cases. Cases 1 and 2 were for deleting leaf nodes and nodes with one child. Those two cases did not lead to `__promote_successor()` calls, so they are the base cases. When you do call `__promote_successor()` for Case 3, it operates on the subtree rooted at the node to delete, so the only chance that the tree being processed recursively isn't smaller than the original is if the node to delete is the root node. The clincher, however, is that `__promote_successor()` calls `__delete()` only on *successor nodes*—nodes that are guaranteed to have at most one child and at least one level lower in the tree than the node they started on. Those always lead to a base case and never to infinite recursion.

Using the Visualization Tool to Delete a Node with Two Children

Generate a tree with the Visualization tool and pick a node with two children. Now mentally figure out which node is its successor, by going to its right child and then following down the line of this right child's left children (if it has any). For your first try, you may want to make sure the successor has no children of its own. On later attempts, try looking at the more complicated situation where entire subtrees of the successor are moved around, rather than a single node.

After you've chosen a node to delete, click the Delete button. You may want to use the Step or Pause/Play buttons to track the individual steps. Each of the methods we've described will appear in the code window, so you can see how it decides the node to delete has two children, locates the successor, copies the successor key and data, and then deletes the successor node.

Is Deletion Necessary?

If you've come this far, you can see that deletion is fairly involved. In fact, it's so complicated that some programmers try to sidestep it altogether. They add a new Boolean field to the `__Node` class, called something like `isDeleted`. To delete a node, they simply set this field to `True`. This is a sort of a "soft" delete, like moving a file to a trash folder without truly deleting it. Then other operations, like `__find()`, check this field to be sure the node isn't marked as deleted before working with it. This way, deleting a node doesn't change the structure of the tree. Of course, it also means that memory can fill up with previously "deleted" nodes.

This approach is a bit of a cop-out, but it may be appropriate where there won't be many deletions in a tree. Be very careful. Assumptions like that tend to come back to haunt you. For example, assuming that deletions might not be frequent for a company's personnel records might encourage a programmer to use the `isDeleted` field. If the company ends up lasting for hundreds of years, there are likely to be more deletions than active employees at some point in the future. The same is true if the company experiences high turnover rates, even over a short time frame. That will significantly affect the performance of the tree operations.

The Efficiency of Binary Search Trees

As you've seen, most operations with trees involve descending the tree from level to level to find a particular node. How long does this operation take? We mentioned earlier that the efficiency of finding a node could range from O(log N) to O(N), but let's look at the details.

In a full, balanced tree, about half the nodes are on the bottom level. More accurately, in a full, balanced tree, there's exactly one more node on the bottom row than in the rest of the tree. Thus, about half of all searches or insertions or deletions require finding a node on the lowest level. (About a quarter of all search operations require finding the node on the next-to-lowest level, and so on.)

During a search, you need to visit one node on each level. You can get a good idea how long it takes to carry out these operations by knowing how many levels there are. Assuming a full, balanced tree, Table 8-1 shows how many levels are necessary to hold a given number of nodes.

The numbers are very much like those for searching the ordered array discussed in Chapter 2. In that case, the number of comparisons for a binary search was approximately equal to the base 2 logarithm of the number of cells in the array. Here, if you call the number of nodes in the first column N, and the number of levels in the second column L, you can say that N is 1 less than 2 raised to the power L, or

$$N = 2^L - 1$$

Adding 1 to both sides of the equation, you have

$$N + 1 = 2^L$$

Using what you learned in Chapter 2 about logarithms being the inverse of raising a number to a power, you can take the logarithm of both sides and rearrange the terms to get

$$\log_2(N + 1) = \log_2(2^L) = L$$

$$L = \log_2(N + 1)$$

Thus, the time needed to carry out the common tree operations is proportional to the base 2 log of N. In Big O notation, you say such operations take O(log N) time.

Table 8-1 Number of Levels for Specified Number of Nodes

Number of Nodes	Number of Levels
1	1
3	2
7	3
15	4
31	5
…	…
1,023	10
…	…
32,767	15
…	…
1,048,575	20
…	…
33,554,431	25
…	…
1,073,741,823	30

If the tree isn't full or balanced, the analysis is difficult. You can say that for a tree with a given number of levels, average search times will be shorter for the nonfull tree than the full tree because fewer searches will proceed to lower levels.

Compare the tree to the other data storage structures we've discussed so far. In an unordered array or a linked list containing 1,000,000 items, finding the item you want takes, on average, 500,000 comparisons, basically O(N). In a balanced tree of 1,000,000 items, only 20 (or fewer) comparisons are required because it's O(log N).

In an ordered array, you can find an item equally quickly, but inserting an item requires, on average, moving 500,000 items. Inserting an item in a tree with 1,000,000 items requires 20 or fewer comparisons, plus a small amount of time to connect the item. The extra time is constant and doesn't depend on the number of items.

Similarly, deleting an item from a 1,000,000-item array requires moving an average of 500,000 items, while deleting an item from a 1,000,000-node tree requires 20 or fewer

comparisons to find the item, plus a few more comparisons to find its successor, plus a short time to disconnect the item and connect its successor. Because the successor is somewhere lower in the tree than the node to delete, the total number of comparisons to find both the node and its successor will be 20 or fewer.

Thus, a tree provides high efficiency for all the common data storage operations: searches, insertions, and deletions. Traversing is not as fast as the other operations, but it must be O(N) to cover all N items, by definition. In all the data structures you've seen, it has been O(N), but we show some other data structures later where it could be greater. There is a little more memory needed for traversing a tree compared to arrays or lists because you need to store the recursive calls or use a stack. That memory will be O(log N). That contrasts with the arrays and lists that need only O(1) memory during traversal.

Trees Represented as Arrays

Up to now, we've represented the binary tree nodes using objects with references for the left and right children. There's a completely different way to represent a tree: with an array.

In the array approach, the nodes are stored in an array and are not linked by references. The position of the node in the array corresponds to its position in the tree. We put the root node at index 0. The root's left child is placed at index 1, and its right child at index 2, and so on, progressing from left to right along each level of the tree. This approach is shown in Figure 8-24, which is a binary search tree with letters for the keys.

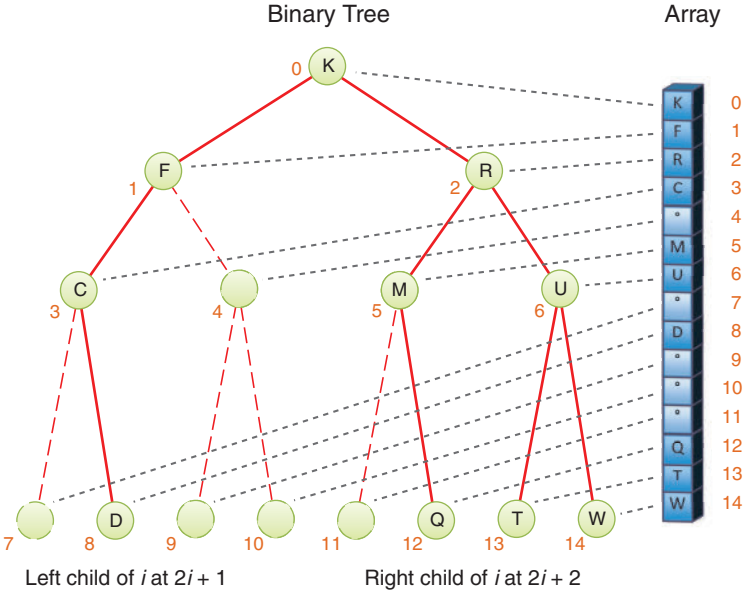


FIGURE 8-24 A binary tree represented by an array

Every position in the tree, whether it represents an existing node or not, corresponds to a cell in the array. Adding a node at a given position in the tree means inserting the node into the equivalent cell in the array. Cells representing tree positions with no nodes are filled with 0, `None`, or some other special value that cannot be confused with a node. In the figure, the ° symbol is used in the array for empty nodes.

With this scheme, a node's children and parent can be found by applying some simple arithmetic to the node's index number in the array. If a node's index number is `index`, this node's left child is

```
2 * index + 1
```

its right child is

```
2 * index + 2
```

and its parent is

```
(index  1) // 2
```

(where the `//` indicates integer division with no remainder). You can verify these formulas work by looking at the indices in Figure 8-24. Any algorithm that follows links between nodes can easily determine where to check for the next node. The scheme works for any binary tree, not just binary search trees. It has the nice feature that edges/links between nodes are just as easy to travel up as they are going down (without the double linking needed for lists). Even better, it can be generalized to any tree with a fixed number of children.

In most situations, however, representing a tree with an array isn't very efficient. Unfilled nodes leave holes in the array, wasting memory. Even worse, when deletion of a node involves moving subtrees, every node in the subtree must be moved to its new location in the array, which is time-consuming in large trees. For insertions that insert nodes beyond the current maximum depth of the tree, the array may need to be resized.

If deletions aren't allowed or are very rare and the maximum depth of the tree can be predicted, the array representation may be useful, especially if obtaining memory for each node dynamically is, for some reason, too time-consuming. That might be the case when programming in assembly language or a very limited operating system, or a system with no garbage collection.

Tree Levels and Size

When trees are represented as arrays, the maximum level and number of nodes is constrained by the size of the array. For linked trees, there's no specific maximum. For both representations, the current maximum level and number of nodes can be determined only by traversing the tree. If there will be frequent calls to request these metrics, the `BinarySearchTree` object can maintain values for them, but the `insert()` and `delete()` methods must be modified to update the values as nodes are added and removed.

To count nodes in a linked tree, you can use the `traverse()` method to iterate over all the nodes and increment a count, as shown earlier in the example to find the average key value and again in the `nodes()` method of Listing 8-10. To find the maximum level, you cannot use the same technique because the level of each node during the traversal is not provided (although it could be added by modifying the generator). Instead, the recursive definition shown in Listing 8-10 gets the job done in a few lines of code.

LISTING 8-10 The `levels()` and `nodes()` Methods of `BinarySearchTree`

```
class BinarySearchTree(object):          # A binary search tree class
…
   def levels(self):                     # Count the levels in the tree
      return self.__levels(self.__root) # Count starting at root

   def __levels(self, node):             # Recursively count levels in subtree
      if node:                           # If a node is provided, then level is 1
         return 1 + max(self.__levels(node.leftChild),  # more than
                        self.__levels(node.rightChild)) # max child
      else: return 0                     # Empty subtree has no levels

   def nodes(self):                      # Count the tree nodes, using iterator
      count = 0                          # Assume an empty tree
      for key, data in self.traverse(): # Iterate over all keys in any
         count += 1                      # order and increment count
      return count
```

Counting the levels of a subtree is somewhat different than what you've seen before in that each node takes the maximum level of each of its subtrees and adds one to it for the node itself. It might seem as if there should be a shortcut by looking at the depth of the minimum or maximum key so that you don't need to visit every node. If you think about it, however, even finding the minimum and maximum keys shows the depth only on the left and right "flanks" of the tree. There could be longer paths somewhere in the middle, and the only way to find them is to visit all the nodes.

Printing Trees

You've seen how to traverse trees in different orders. You could always use the traversal method to print all the nodes in the tree, as shown in the Visualization tool. Using the in-order traversal would show the items in increasing order of their keys. On a two-dimensional output, you could use the in-order sequence to position the nodes along the horizontal axis and the level of each node to determine its vertical position. That could produce tree diagrams like the ones shown in the previous figures.

On a simple command-line output, it's easier to print one node per line. The problem then becomes positioning the node on the line to indicate the shape of the tree. If you want the root node at the top, then you must compute the width of the full tree and place

that node in the middle of the full width. More accurately, you would have to compute the width of the left and right subtrees and use that to position the root in order to show balanced and unbalanced trees accurately.

On the other hand, if you place the root at the left side of an output line and show the level of nodes as indentation from the leftmost column, it's easy to print the tree on a terminal. Doing so essentially rotates the tree 90° to the left. Each node of the tree appears on its own line of the output. That allows you to forget about determining the width of subtrees and write a simple recursive method, as shown in Listing 8-11.

LISTING 8-11 Methods to Print Trees with One Node per Line

```
class BinarySearchTree(object):          # A binary search tree class
…
   def print(self,                       # Print the tree sideways with 1 node
             indentBy=4):                # on each line and indenting each level
      self.__pTree(self.__root,          # by some blanks. Start at root node
                   "ROOT:   ", "", indentBy) # with no indent

   def __pTree(self,                     # Recursively print a subtree, sideways
               node,                     # with the root node left justified
               nodeType,                 # nodeType shows the relation to its
               indent,                   # parent and the indent shows its level
               indentBy=4):              # Increase indent level for subtrees
      if node:                           # Only print if there is a node
         self.__pTree(node.rightChild, "RIGHT:  ", # Print the right
                      indent + " " * indentBy, indentBy) # subtree
         print(indent + nodeType, node) # Print this node
         self.__pTree(node.leftChild,  "LEFT:   ", # Print the left
                      indent + " " * indentBy, indentBy) # subtree
```

The public `print()` method calls the private `__pTree()` method to recursively print the nodes starting at the root node. It takes a parameter, `indentBy`, to control how many spaces are used to indent each level of the tree. It labels the nodes to show their relationship with their parent (if it wasn't already clear from their indentation and relative positions). The recursive method implementation starts by checking the base case, an empty `node`, in which case nothing needs to be printed. For every other node, it first recursively prints the right subtree because that is the top of the printed version. It adds spaces to the indent so that subtree is printed further to the right. Then it prints the current node prefixed with its indentation and `nodeType` label. Lastly, it prints the left subtree recursively with the extended indentation. This produces an output such as that shown in Figure 8-25. The nodes are printed as {*key*, *data*} pairs and the figure example has no data stored with it.

```
            RIGHT: {86, }
        RIGHT: {83, }
            LEFT: {71, }
    RIGHT: {65,}
        LEFT: {57, }
                RIGHT: {55, }
            LEFT: {49, }
ROOT: {44, }
        RIGHT: {33, }
    LEFT: {27, }
```

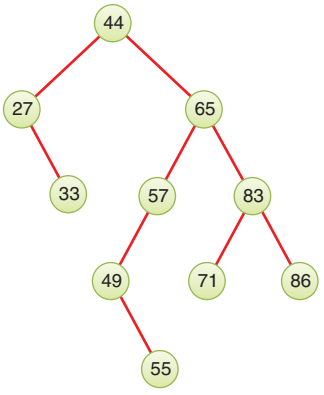


FIGURE 8-25 Tree printed with indentation for node depth

In printing the tree like this, you use a different traversal order from the three standard ones. The print order uses a *reverse* in-order traversal of the tree.

Duplicate Keys

As in other data structures, the problem of duplicate keys must be addressed. In the code shown for `insert()` and in the Visualization tool, a node with a duplicate key will not be inserted. The tool shows the data for the node being updated by moving a new colored circle to fill the node.

To allow for duplicate keys, you must make several choices. The duplicates go in the right subtree based on the fundamental binary search tree rule. They form a chain of nodes with only right child links, as shown in Figure 8-26. One of the design choices is where to put any left child link. It should go only at the first or last duplicate in the chain so that the algorithms know where to find it. The figure illustrates the two choices. New duplicate keys should be inserted at the opposite end of the chain.

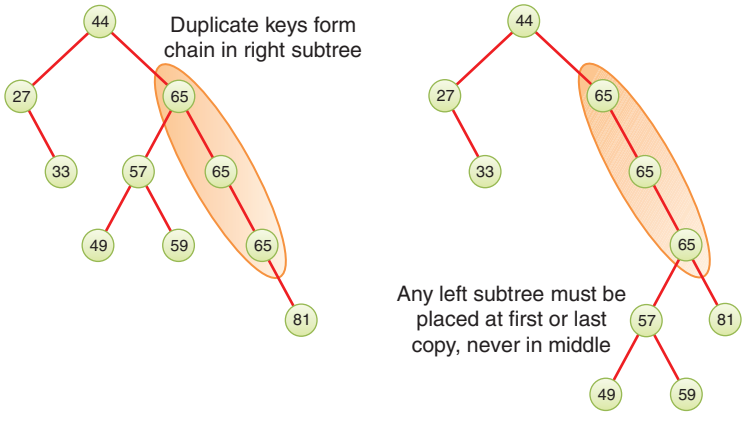


FIGURE 8-26 Duplicate keys in binary search trees

Another choice is what to return from the `__find()` and `search()` methods for a key that has duplicates. Should it return the first or the last? The choice should also be consistent with what node is deleted and returned by the `delete()` method. If they are inserted at the first and removed from the first, then `delete()` will act like a mini stack for the duplicate nodes.

The delete operation is complicated by the fact that different data values could be stored at each of the duplicate nodes. The caller may need to delete a node with specific data, rather than just any node with the duplicate key. Whichever scheme is selected, the deletion routine will need to ensure that the left subtree, if any, remains attached to the appropriate place.

With any kind of duplicate keys, balancing the tree becomes difficult or impossible. The chains of duplicates add extra levels that cannot be rearranged to help with balance. That means the efficiency of finding an item moves away from best case of O(log N) toward O(N).

As you can see, allowing duplicate keys is not a simple enhancement to the data structure. In other data structures, duplicate keys present challenges, but not all of them are as tricky as the binary search tree.

The BinarySearchTreeTester.py Program

It's always a good idea to test the functioning of a code module by writing tests that exercise each operation. Writing a comprehensive set of tests is an art in itself. Another useful strategy is to write an interactive test program that allows you to try a series of operations in different orders and with different arguments. To test all the `BinarySearchTree` class methods shown, you can use a program like `BinarySearchTreeTester.py` shown in Listing 8-12.

LISTING 8-12 The `BinarySearchTreeTester.py` Program

```
# Test the BinarySearchTree class interactively
from BinarySearchTree import *

theTree = BinarySearchTree()          # Start with an empty tree

theTree.insert("Don",  "1974 1")      # Insert some data
theTree.insert("Herb", "1975 2")
theTree.insert("Ken",  "1979 1")
theTree.insert("Ivan", "1988 1")
theTree.insert("Raj",  "1994 1")
theTree.insert("Amir", "1996 1")
theTree.insert("Adi",  "2002 3")
theTree.insert("Ron",  "2002 3")
theTree.insert("Fran", "2006 1")
theTree.insert("Vint", "2006 2")
theTree.insert("Tim",  "2016 1")

def print_commands(names):            # Print a list of possible commands
   print('The possible commands are', names)
```

```
def clearTree():                        # Remove all the nodes in the tree
   while not theTree.isEmpty():
      data, key = theTree.root()
      theTree.delete(key)

def traverseTree(traverseType="in"):  # Traverse & print all nodes
   for key, data in theTree.traverse(traverseType):
      print('{', str(key), ', ', str(data), '}', end=' ')
   print()

commands = [  # Command names, functions, and their parameters
   ['print', theTree.print, []],
   ['insert', theTree.insert, ('key', 'data')],
   ['delete', theTree.delete, ('key', )],
   ['search', theTree.search, ('key', )],
   ['traverse', traverseTree, ('type', )],
   ['clear', clearTree, []],
   ['help', print_commands, []],
   ['?', print_commands, []],
   ['quit', None, []],
]
                                        # Collect all the command names in a list
command_names = ", ".join(c[0] for c in commands)
for i in range(len(commands)):          # Put command names in argument list
   if commands[i][1] == print_commands: # of print_commands
      commands[i][2] = [command_names]
# Create a dictionary mapping first character of command name to
# command specification (name, function, parameters/args)
command_dict = dict((c[0][0], c) for c in commands)

                                        # Print information for interactive loop
theTree.print()
print_commands(command_names)
ans = ' '

# Loop to get a command from the user and execute it
while ans[0] != 'q':
   print('The tree has', theTree.nodes(), 'nodes across',
         theTree.levels(), 'levels')
   ans = input("Enter first letter of command: ").lower()
   if len(ans) == 0:
      ans = ' '
   if ans[0] in command_dict:
      name, function, parameters = command_dict[ans[0]]
      if function is not None:
```

```
            print(name)
            if isinstance(parameters, list):
               arguments = parameters
            else:
               arguments = []
               for param in parameters:
                  arg = input("Enter " + param + " for " + name + " " +
                              "command: ")
                  arguments.append(arg)
            try:
               result = function(*arguments)
               print('Result:', result)
            except Exception as e:
               print('Exception occurred')
               print(e)
      else:
         print("Invalid command: '", ans, "'")
```

This program allows users to enter commands by typing them in a terminal interface. It first imports the `BinarySearchTree` module and creates an empty tree with it. Then it puts some data to it, using `insert()` to associate names with some strings. The names are the keys used to place the nodes within the tree.

The tester defines several utility functions to print all the possible commands, clear all the nodes from the tree, and traverse the tree to print each node. These functions handle commands in the command loop below.

The next part of the tester program defines a list of commands. For each one, it has a name, a function to execute the command, and a list or tuple of arguments or parameters. This is more advanced Python code than we've shown so far, so it might look a little strange. The names are what the user will type (or at least their first letter), and the functions are either methods of the tree or the utility functions defined in the tester. The arguments and parameters will be processed after the user chooses a command.

To provide a little command-line help, the tester concatenates the list of command names into a string, separating them with commas. This operation is accomplished with the `join()` method of strings. The text to place between each command name is the string (a comma and a space), and the argument to `join()` is the list of names. The program uses a list comprehension to iterate through the command specifications in `commands` and pull out the first element, which is the command name: `", ".join(c[0] for c in commands)`. The result is stored in the `command_names` variable.

Then the concatenated string of command names needs to get inserted in the argument list for the `print_commands` function. That's done in the `for` loop. Two entries have the `print_commands` function: the `help` and `?` commands.

The last bit of preparation for the command loop creates a dictionary, `command_dict`, that maps the first character of each command to the command specification. You haven't

used this Python data structure yet. In Chapter 11, "Hash Tables," you see how they work, so if you're not familiar with them, think of them as an **associative array**—an array indexed by a string instead of integer. You can assign values in the array and then look them up quickly. In the tester program, evaluating `command_dict['p']` would return the specification for the print command, namely `['print', theTree.print, []]`. Those specifications get stored in the dictionary using the compact (but cryptic) comprehension: `dict((c[0][0], c) for c in commands)`.

The rest of the tester implements the command loop. It first prints the tree on the terminal, followed by the list of commands. The `ans` variable holds the input typed by the user. It gets initialized to a space so that the command loop starts and prompts for a new command.

The command loop continues until the user invokes the quit command, which starts with `q`. Inside the loop body, the number of nodes and levels in the tree is printed, and then the user is asked for a command. The string that is returned by `input()` is converted to lowercase to simplify the command lookup. If the user just pressed Return, there would be no first character in the string, so you would fill in a `?` to make the default response be to print all the command names again.

In the next statement—`if ans[0] in command_dict:`—the tester checks whether the first character in the user's response is one of the known commands. If the character is recognized, it extracts the `name`, `function`, and `parameters` from the specification stored in the `command_dict`. If there's a function to execute, then it will be processed. If not, then the user asked to quit, and the `while` loop will exit. When the first character of the user's response does not match a command, an error message is printed, and the loop prompts for a new command.

After the command specification is found, it either needs to prompt the user for the arguments to use when calling the `function` or get them from the specification. This choice is based on whether the parameters were specified as Python tuple or list. If it's a tuple, the elements of the tuple are the names of the `parameters`. If it's a list, then the list contains the `arguments` of the function. For tuples, the user is prompted to enter each argument by name, and the answers are stored in the `arguments` list. After the `arguments` are determined, the command loop tries calling the function with the `arguments` list using `result = function(*arguments)`. The asterisk (`*`) before the `arguments` is not a multiplication operator. It means that the `arguments` list should be used as the list of positional arguments for the function. If the function raises any exceptions, they are caught and displayed. Otherwise, the result of the function is printed before looping to get another command.

Try using the tester to run the four main operations: search, insert, traverse, and delete. For the deletion, try deleting nodes with 0, 1, and 2 child nodes to see the effect. When you delete a node with 2 children, predict which successor node will replace the deleted node and see whether you're right.

The Huffman Code

You shouldn't get the idea that binary trees are always search trees. Many binary trees are used in other ways. Figure 8-16 shows an example where a binary tree represents an algebraic expression. We now discuss an algorithm that uses a binary tree in a surprising way to compress data. It's called the Huffman code, after David Huffman, who discovered it in 1952. Data compression is important in many situations. An example is sending data over the Internet or via digital broadcasts, where it's important to send the information in its shortest form. Compressing the data means more data can be sent in the same time under the bandwidth limits.

Character Codes

Each character in an uncompressed text file is represented in the computer by one to four bytes, depending on the way characters are encoded. For the venerable ASCII code, only one byte is used, but that limits the range of characters that can be expressed to fewer than 128. To account for all the world's languages plus other symbols like emojis ☺, the various Unicode standards use up to four bytes per character. For this discussion, we assume that only the ASCII characters are needed, and each character takes one byte (or eight bits). Table 8-2 shows how some characters are represented in binary using the ASCII code.

Table 8-2 Some ASCII Codes

Character	Decimal	Binary
@	64	01000000
A	65	01000001
B	66	01000010
…	…	…
Y	89	01011001
Z	90	01011010
…	…	…
a	97	01100001
b	98	01100010

There are several approaches to compressing data. For text, the most common approach is to reduce the number of bits that represent the most-used characters. As a consequence, each character takes a variable number of bits in the "stream" of bits that represents the full text.

In English, *E* and *T* are very common letters, when examining prose and other person-to-person communication and ignoring things like spaces and punctuation. If you choose a scheme that uses only a few bits to write *E*, *T*, and other common letters, it should be more compact than if you use the same number of bits for every letter. On the other end of the spectrum, *Q* and *Z* seldom appear, so using a large number of bits occasionally for those letters is not so bad.

Suppose you use just two bits for *E*—say 01. You can't encode every letter of the English alphabet in two bits because there are only four 2-bit combinations: 00, 01, 10, and 11. Can you use these four combinations for the four most-used characters? Well, if you did, and you still wanted to have some encoding for the lesser-used characters, you would have trouble. The algorithm that interprets the bits would have to somehow guess whether a pair of bits is a single character or part of some longer character code.

One of the key ideas in encoding is that we must set aside some of the code values as indicators that a longer bit string follows to encode a lesser-used character. The algorithm needs a way to look at a bit string of a particular length and determine if that is the full code for one of the characters or just a prefix for a longer code value. You must be careful that no character is represented by the same bit combination that appears at the beginning of a longer code used for some other character. For example, if *E* is 01, and *Z* is 01011000, then an algorithm decoding 01011000 wouldn't know whether the initial 01 represented an *E* or the beginning of a *Z*. This leads to a rule: *No code can be the prefix of any other code*.

Consider also that in some messages, *E* might not be the most-used character. If the text is a program source file, for example, punctuation characters such as the colon (:), semicolon (;), and underscore (_) might appear more often than *E* does. Here's a solution to that problem: for each message, you make up a new code tailored to that particular message. Suppose you want to send the message `SPAM SPAM SPAM EGG + SPAM`. The letter *S* appears a lot, and so does the space character. You might want to make up a table showing how many times each letter appears. This is called a frequency table, as shown in Table 8-3.

Table 8-3 Frequency Table for the SPAM Message

Character	Count		Character	Count
A	4		P	4
E	1		S	4
G	2		Space	5
M	4		+	1

The characters with the highest counts should be coded with a small number of bits. Table 8-4 shows one way how you might encode the characters in the SPAM message.

You can use 01 for the space because it is the most frequent. The next most frequent characters are *S*, *P*, *A*, and *M*, each one appearing four times. You use the code 00 for the last one, *M*. The remaining codes can't start with 00 or 01 because that would break the rule

that no code can be a prefix of another code. That leaves 10 and 11 to use as prefixes for the other characters.

Table 8-4 Huffman Code for the SPAM Message

Character	Count	Code		Character	Count	Code
A	4	111		P	4	110
E	1	10000		S	4	101
G	2	1001		Space	5	01
M	4	00		+	1	10001

What about 3-bit code combinations? There are eight possibilities: 000, 001, 010, 011, 100, 101, 110, and 111, but you already know you can't use anything starting with 00 or 01. That eliminates four possibilities. You can assign some of those 3-bit codes to the next most frequent characters, *S* as 101, *P* as 110, and *A* as 111. That leaves the prefix 100 to use for the remaining characters. You use a 4-bit code, 1001, for the next most frequent character, *G*, which appears twice. There are two characters that appear only once, *E* and +. They are encoded with 5-bit codes, 10000 and 10001.

Thus, the entire message is coded as

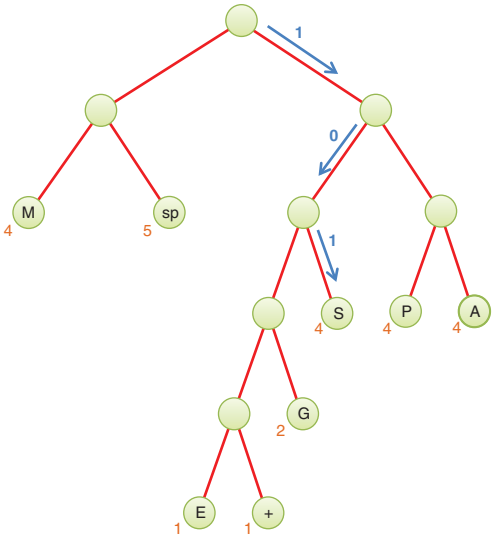
```
101 110 111 00 01 101 110 111 00 01 101 110 111 00 01 10000 1001 1001 01 10001 01
101 110 111 00
```

For legibility, we show this message broken into the codes for individual characters. Of course, all the bits would run together because there is no space character in a binary message, only 0s and 1s. That makes it more challenging to find which bits correspond to a character. The main point, however, is that the 25 characters in the input message, which would typically be stored in 200 bits in memory (8×25), require only 72 bits in the Huffman coding.

Decoding with the Huffman Tree

We show later how to create Huffman codes. First, let's examine the somewhat easier process of decoding. Suppose you received the string of bits shown in the preceding section. How would you transform it back into characters? You could use a kind of binary tree called a **Huffman tree**. Figure 8-27 shows the Huffman tree for the SPAM message just discussed.

The characters in the message appear in the tree as leaf nodes. The higher their frequency in the message, the higher up they appear in the tree. The number outside each leaf node is its frequency. That puts the space character (sp) at the second level, and the *S*, *P*, *A*, and *M* characters at the second or third level. The least frequent, *E* and +, are on the lowest level, 5.



Message: SPAM SPAM SPAM EGG + SPAM

Character	Count	Code
A	4	111
E	1	10000
G	2	1001
M	4	00
P	4	110
S	4	101
Space	5	01
+	1	10001

FIGURE 8-27 Huffman tree for the SPAM message

How do you use this tree to decode the message? You start by looking at the first bit of the message and set a pointer to the root node of the tree. If you see a 0 bit, you move the pointer to the left child of the node, and if you see a 1 bit, you move it right. If the identified node does not have an associated character, then you advance to the next bit in the message. Try it with the code for *S*, which is 101. You go right, left, then right again, and voila, you find yourself on the *S* node. This is shown by the blue arrows in Figure 8-27.

You can do the same with the other characters. After you've arrived at a leaf node, you can add its character to the decoded string and move the pointer back to the root node. If you have the patience, you can decode the entire bit string this way.

Creating the Huffman Tree

You've seen how to use a Huffman tree for decoding, but how do you create this tree? There are many ways to handle this problem. You need a Huffman tree object, and that is somewhat like the `BinarySearchTree` described previously in that it has nodes that have up to two child nodes. It's quite different, however, because routines that are specific to keys in search trees, like `find()`, `insert()`, and `delete()`, are not relevant. The constraint that a node's key be larger than any key of its left child and equal to or less than any key of its right child doesn't apply to a Huffman tree. Let's call the new class `HuffmanTree`, and like the search tree, store a key and a value at each node. The key will hold the decoded message character such as S or G. It could be the space character, as you've seen, and it needs a special value for "no character".

Here is the algorithm for constructing a Huffman tree from a message string:

Preparation

1. Count how many times each character appears in the message string.
2. Make a `HuffmanTree` object for each character used in the message. For the SPAM message example, that would be eight trees. Each tree has a single node whose key is a character and whose value is that character's frequency in the message. Those values can be found in Table 8-3 or Table 8-4 for the SPAM message.
3. Insert these trees in a priority queue (as described in Chapter 4). They are ordered by the frequency (stored as the value of each root node) and the number of levels in the tree. The tree with the smallest frequency has the highest priority. Among trees with equal frequency, the one with more levels is the highest priority. In other words, when you remove a tree from the priority queue, it's always the one with the deepest tree of the least-used character. (Breaking ties using the tree depth, improves the balance of the final Huffman tree.)

That completes the preparation, as shown in Step 0 of Figure 8-28. Each single node Huffman trees has a character shown in the center of the node and a frequency value shown below and to the left of the node.

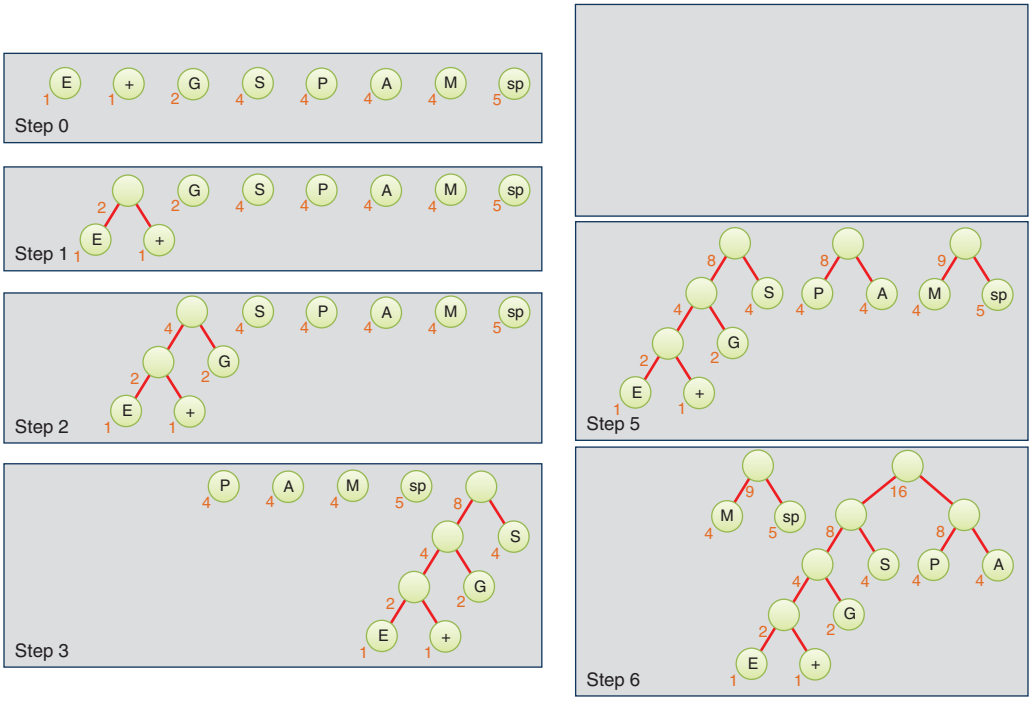


FIGURE 8-28 Growing the Huffman tree, first six steps

Then do the following:

Tree consolidation

1. Remove two trees from the priority queue and make them into children of a new node. The new node has a frequency value that is the sum of the children's frequencies; its character key can be left blank (the special value for no character, not the space character).
2. Insert this new, deeper tree back into the priority queue.
3. Keep repeating steps 1 and 2. The trees will get larger and larger, and there will be fewer and fewer of them. When there is only one tree left in the priority queue, it is the Huffman tree and you're done.

Figure 8-28 and Figure 8-29 show how the Huffman tree is constructed for the SPAM message.

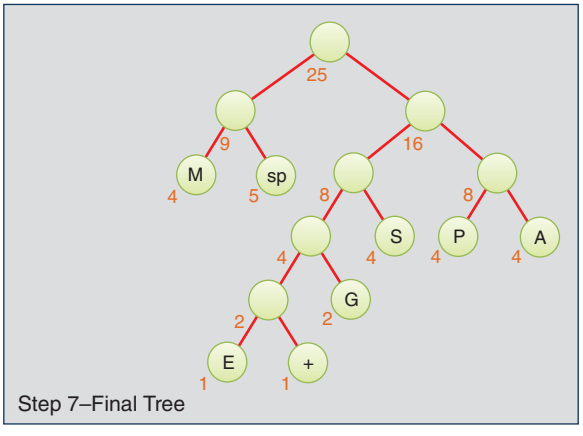


FIGURE 8-29 Growing the Huffman tree, final step

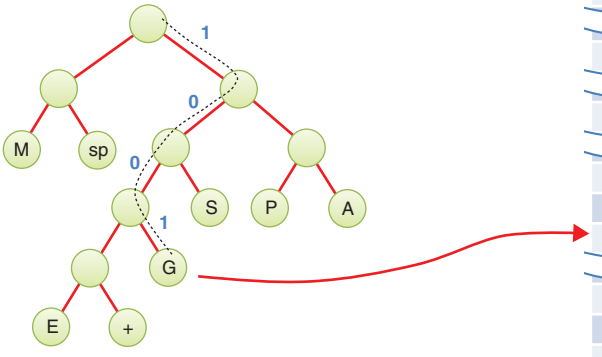
Coding the Message

Now that you have the Huffman tree, how do you encode a message? You start by creating a code table, which lists the Huffman code alongside each character. To simplify the discussion, we continue to assume that only ASCII characters are possible, so we need a table with 128 cells. The index of each cell would be the numerical value of the ASCII character: 65 for A, 66 for B, and so on. The contents of the cell would be the Huffman code for the corresponding character. Initially, you could fill in some special value for indicating "no code" like `None` or an empty string in Python to check for errors where you failed to make a code for some character.

Such a code table makes it easy to generate the coded message: for each character in the original message, you use its code as an index into the code table. You then repeatedly append the Huffman codes to the end of the coded message until it's complete.

To fill in the codes in the table, you traverse the Huffman tree, keeping track of the path to each node as it is visited. When you visit a leaf node, you use the key for that node as the index to the table and insert the path as a binary string into the cell's value. Not every cell contains a code—only those appearing in the message. Figure 8-30 shows how this looks for the SPAM message. The table is abbreviated to show only the significant rows. The path to the leaf node for character *G* is shown as the tree is being traversed.

The full code table can be built by calling a method that starts at the root and then calls itself recursively for each child. Eventually, the paths to all the leaf nodes will be explored, and the code table will be complete.



Index	Character	Code
32	Space	01
43	+	10001
65	A	111
69	E	10000
70	F	
71	G	1001
77	M	00
78	N	
79	O	
80	P	110
81	Q	
82	R	
83	S	101

FIGURE 8-30 Building the code table

One more thing to consider: if you receive a binary message that's been compressed with a Huffman code, how do you know what Huffman tree to use for decoding it? The answer is that the Huffman tree must be sent first, before the binary message, in some format that doesn't require knowledge of the message content. Remember that Huffman codes are for *compressing the data*, not encrypting it. Sending a short description of the Huffman tree followed by a compressed version of a long message saves many bits.

Summary

- Trees consist of nodes connected by edges.
- The root is the topmost node in a tree; it has no parent.
- All nodes but the root in a tree have exactly one parent.
- In a binary tree, a node has at most two children.
- Leaf nodes in a tree have no child nodes and exactly one path to the root.
- An unbalanced tree is one whose root has many more left descendants than right descendants, or vice versa.
- Each node of a tree stores some data. The data typically has a key value used to identify it.
- Edges are most commonly represented by references to a node's children; less common are references from a node to its parent.
- Traversing a tree means visiting all its nodes in some predefined order.
- The simplest traversals are pre-order, in-order, and post-order.
- Pre-order and post-order traversals are useful for parsing algebraic expressions.
- *Binary Search Trees*
  - In a binary search tree, all the nodes that are left descendants of node A have key values less than that of A; all the nodes that are A's right descendants have key values greater than (or equal to) that of A.
  - Binary search trees perform searches, insertions, and deletions in O(log N) time.
  - Searching for a node in a binary search tree involves comparing the goal key to be found with the key value of a node and going to that node's left child if the goal key is less or to the node's right child if the goal key is greater.
  - Insertion involves finding the place to insert the new node and then changing a child field in its new parent (or the root of the tree) to refer to it.
  - An in-order traversal visits nodes in order of ascending keys.
  - When a node has no children, you can delete it by clearing the child field in its parent (for example, setting it to `None` in Python).
  - When a node has one child, you can delete it by setting the child field in its parent to point to its child.
  - When a node has two children, you can delete it by replacing it with its successor and deleting the successor from the subtree.
  - You can find the successor to a node A by finding the minimum node in A's right subtree.

  - Nodes with duplicate key values require extra coding because typically only one of them (the first) is found in a search, and managing their children complicates insertions and deletions.

- Trees can be represented in the computer's memory as an array, although the reference-based approach is more common and memory efficient.
- A Huffman tree is a binary tree (but not a search tree) used in a data-compression algorithm called Huffman coding.
- In the Huffman code, the characters that appear most frequently are coded with the fewest bits, and those that appear rarely are coded with the most bits.
- The paths in the Huffman tree provide the codes for each of the leaf nodes.
- The level of a leaf node indicates the number of bits used in the code for its key.
- The characters appearing the least frequently in a Huffman coded message are placed in leaf nodes at the deepest levels of the Huffman tree.

Questions

These questions are intended as a self-test for readers. Answers may be found in Appendix C.

1. Insertion and deletion in a binary search tree require what Big O time?
2. A binary tree is a search tree if
   a. every nonleaf node has children whose key values are less than or equal to the parent.
   b. the key values of every nonleaf node are the sum or concatenation of the keys of its children
   c. every left child has a key less than its parent and every right child has a key greater than or equal to its parent.
   d. in the path from the root to every leaf node, the key of each node is greater than or equal to the key of its parent.
3. True or False: If you traverse a tree and print the path to each node as a series of the letters *L* and *R* for whether the path followed the left or right child at each step, there could be some duplicate paths.
4. When compared to storing data in an ordered array, the main benefit of storing it in a binary search tree is
   a. having the same search time as traversal time in Big O notation.
   b. not having to copy data when inserting or deleting items.
   c. being able to search for an item in O(log N) time.
   d. having a key that is separate from the value identified by the key.

5. In a complete, balanced binary tree with 20 nodes, and the root considered to be at level 0, how many nodes are there at level 4?
6. A subtree of a binary tree always has
   a. a root that is a child of the main tree's root.
   b. a root unconnected to the main tree's root.
   c. fewer nodes than the main tree.
   d. a sibling with an equal or larger number of nodes.
7. When implementing trees as objects, the _____ and the _____ are generally separate classes.
8. Finding a node in a binary search tree involves going from node to node, asking
   a. how big the node's key is in relation to the search key.
   b. how big the node's key is compared to its right or left child's key.
   c. what leaf node you want to reach.
   d. whether the level you are on is above or below the search key.
9. An unbalanced tree is one
   a. in which most of the keys have values greater than the average.
   b. where there are more nodes above the central node than below.
   c. where the leaf nodes appear much more frequently as the left child of their parents than as the right child, or vice versa.
   d. in which the root or some other node has many more left descendants than right descendants, or vice versa.
10. True or False: A hierarchical file system is essentially a binary search tree, although it can be unbalanced.
11. Inserting a node starts with the same steps as _____ a node.
12. Traversing tree data structures
    a. requires multiple methods to handle the different traversal orders.
    b. can be implemented using recursive functions or generators.
    c. is much faster than traversing array data structures.
    d. is a way to make soft deletion of items practical.
13. When a tree is extremely unbalanced, it begins to behave like the _____ data structure.

14. Suppose a node A has a successor node S in a binary search tree with no duplicate keys. Then S must have a key that is larger than _____ but smaller than or equal to _____.

15. Deleting nodes in a binary search tree is complex because

    a. copying subtrees below the successor requires another traversal.

    b. finding the successor is difficult to do, especially when the tree is unbalanced.

    c. the tree can split into multiple trees, a forest, if it's not done properly.

    d. the operation is very different for the different number of child nodes of the node to be deleted, 0, 1, or 2.

16. In a binary tree used to represent a mathematical expression,

    a. both children of an operator node must be operands.

    b. following a post-order traversal, parentheses must be added.

    c. following a pre-order traversal, parentheses must be added.

    d. in pre-order traversal, a node is visited before either of its children.

17. When a tree is represented by an array, the right child of a node at index *n* has an index of _____.

18. True or False: Deleting a node with one child from a binary search tree involves finding that node's successor.

19. A Huffman tree is typically used to _____ text data.

20. Which of the following is ***not*** true about a Huffman tree?

    a. The most frequently used characters always appear near the top of the tree.

    b. Normally, decoding a message involves repeatedly following a path from the root to a leaf.

    c. In coding a character, you typically start at a leaf and work upward.

    d. The tree can be generated by removal and insertion operations on a priority queue of small trees.

Experiments

Carrying out these experiments will help to provide insights into the topics covered in the chapter. No programming is involved.

8-A Use the Binary Search Tree Visualization tool to create 20 random trees using 20 as the requested number of items. What percentage would you say are seriously unbalanced?

8-B Use the `BinarySearchTreeTester.py` program shown in Listing 8-12 and provided with the code examples from the publisher's website to do the following experiments:

a. Delete a node that has no children.

b. Delete a node that has 1 child node.

c. Delete a node that has 2 child nodes.

d. Pick a key for a new node to insert. Determine where you think it will be inserted in the tree, and then insert it with the program. Is it easy to determine where it will go?

e. Repeat the previous step with another key but try to put it in the other child branch. For example, if your first node was inserted as the left child, try to put one as the right child or in the right subtree.

8-C The `BinarySearchTreeTester.py` program shown in Listing 8-12 prints an initial tree of 11 nodes across 7 levels, based on the insertion order of the items. A fully balanced version of the tree would have the same nodes stored on 4 levels. Use the program to clear the tree, and then determine what order to insert the same keys to make a balanced tree. Try your ordering and see whether the tree comes out balanced. If not, try another ordering. Can you describe in a few sentences the insertion ordering that will always create a balanced binary search tree from a particular set of keys?

8-D Use the Binary Search Tree Visualization tool to delete a node in every possible situation.

Programming Projects

Writing programs to solve the Programming Projects helps to solidify your understanding of the material and demonstrates how the chapter's concepts are applied. (As noted in the Introduction, qualified instructors may obtain completed solutions to the Programming Projects on the publisher's website.)

8.1 Alter the `BinarySearchTree` class described in this chapter to allow nodes with duplicate keys. Three methods are affected: `__find()`, `insert()`, and `delete()`. Choose to insert new left children at the shallowest level among equal keys, as shown on the left side of Figure 8-26, and always find and delete the deepest among equal keys. More specifically, the `__find()` and `search()` methods should return the deepest among equal keys that it encounters but should allow an optional parameter to specify finding the shallowest. The `insert()` method must handle the case when the item to be inserted duplicates an existing node, by inserting a new node with an empty left child below the deepest duplicate key. The `delete()` method must delete the deepest node among duplicate keys, thus providing a LIFO or stack-like behavior among duplicate keys. Think carefully about the deletion cases and whether the

choice of successor nodes changes. Demonstrate how your implementation works on a tree inserting several duplicate keys associated with different values. Then delete those keys and show their values to make it clear that the last duplicate inserted is the first duplicate deleted.

8.2 Write a program that takes a string containing a postfix expression and builds a binary tree to represent the algebraic expression like that shown in Figure 8-16. You need a `BinaryTree` class, like that of `BinarySearchTree`, but without any keys or ordering of the nodes. Instead of `find()`, `insert()`, and `delete()` methods, you need the ability to make single node `BinaryTrees` containing a single operand and a method to combine two binary trees to make a third with an operator as the root node. The syntax of the operators and operands is the same as what was used in the `PostfixTranslate.py` module from Chapter 4. You can use the `nextToken()` function in that module to parse the input string into operator and operand tokens. You don't need the parentheses as delimiters because postfix expressions don't use them. Verify that the input expression produces a single algebraic expression and raise an exception if it does not. For valid algebraic binary trees, use pre-, in-, and post-order traversals of the tree to translate the input into the output forms. Include parentheses for the in-order traversal to make the operator precedence clear in the output translation. Run your program on at least the following expressions:

a. 91 95 + 15 + 19 + 4 *

b. B B * A C 4 * * –

c. 42

d. A 57 # this should produce an exception

e. + / # this should produce an exception

8.3 Write a program to implement Huffman coding and decoding. It should do the following:

▶ Accept a text message (string).

▶ Create a Huffman tree for this message.

▶ Create a code table.

▶ Encode the text message into binary.

▶ Decode the binary message back to text.

▶ Show the number of bits in the binary message and the number of characters in the input message.

If the message is short, the program should be able to display the Huffman tree after creating it. You can use Python `string` variables to store binary messages as arrangements of the characters 1 and 0. Don't worry about doing actual bit manipulation using `bytearray` unless you really want to. The easiest way to create the code table

in Python is to use the dictionary (`dict`) data type. If that is unfamiliar, it's essentially an array that can be indexed by a string or a single character. It's used in the `BinarySearchTreeTester.py` module shown in Listing 8-12 to map command letters to command records. If you choose to use an integer indexed array, you can use Python's `ord()` function to convert a character to an integer but be aware that you will need a large array if you allow arbitrary Unicode characters such as emojis (☺) in the message.

8.4 Measuring tree balance can be tricky. You can apply two simple measures: node balance and level (or height) balance. As mentioned previously, balanced trees have an approximately equal number of nodes in their left and right subtrees. Similarly, the left and right subtrees must have an approximately equal number of levels (or height). Extend the `BinarySearchTree` class by writing the following methods:

a. `nodeBalance()`—Computes the number of nodes in the right subtree minus the number of nodes in the left subtree

b. `levelBalance()`—Computes the number of levels in the right subtree minus the number of levels in the left subtree

c. `unbalancedNodes(by=1)`— Returns a list of node keys where the absolute value of either of the balance metrics exceeds the `by` threshold, which defaults to 1

These three methods all require (recursive) helper methods that traverse subtrees rooted at nodes inside the tree. In a balanced tree, the list of unbalanced nodes would be empty. Try your measures by inserting the following four lists of keys into an empty `BinarySearchTree` (in order, left to right), printing the resulting 15-node tree, printing the node and level balance of the resulting root node, and then printing the list of unbalanced keys with `by=1` and `by=2`.

```
[7, 6, 5, 4, 3, 2, 1, 8, 12, 10, 9, 11, 14, 13, 15],
[8, 4, 5, 6, 7, 3, 2, 1, 12, 10, 9, 11, 14, 13, 15],
[8, 4, 2, 3, 1, 6, 5, 7, 12, 10, 9, 11, 14, 13, 15],
[8, 4, 2, 3, 1, 6, 5, 7, 12, 10, 9, 11, 14, 13, 8.5]
```

8.5 Every binary tree can be represented as an array, as described in the section titled "Trees Represented as Arrays." The reverse of representing an array as a tree, however, works only for some arrays. The missing nodes of the tree are represented in the array cells as some predefined value—such as `None`—that cannot be a value stored at a tree node. If the root node is missing in the array, then the corresponding tree cannot be built. Write a function that takes an array as input and tries to make a binary tree from its contents. Every cell that is not `None` is a value to store at a tree node. When you come across a node without a parent node (other than the root node), the function should raise an exception indicating that the tree cannot be built. Note that the result won't necessarily be a binary search tree, just a binary tree. Hint: It's easier to work from the leaf nodes to the root, building nodes for each cell that is not `None` and storing the resulting node back in the same cell of the input array for

retrieval when it is used as a subtree of a node on another level. Print the result of running the function on the following arrays where `n = None`. The values in the array can be stored as either the key or the value of the node because the tree won't be interpreted as a binary search tree.

```
[],
[n, n, n],
[55, 12, 71],
[55, 12, n, 4],
[55, 12, n, 4, n, n, n, n, 8, n, n, n, n, n, n, n, n, 6, n],
[55, 12, n, n, n, n, 4, n, 8, n, n, n, n, n, n, n, n, 6, n]
```

CHAPTER 9

2-3-4 Trees and External Storage

IN THIS CHAPTER

- Introduction to 2-3-4 Trees
- The Tree234 Visualization Tool
- Python Code for a 2-3-4 Tree
- Efficiency of 2-3-4 Trees
- 2-3 Trees
- External Storage

In a binary tree, each node has one data item and can have up to two children. If you allow more data items and children per node, the result is a *multiway tree*. A 2-3-4 tree, to which we devote the first part of this chapter, is a multiway tree that can have up to four children and three data items per node.

These 2-3-4 trees are interesting for several reasons. First, they're balanced trees designed to avoid the problems that come from having too many nodes along some of the paths in the tree. Second, there's an interesting way to convert them into binary trees that we describe in a later chapter. Third, and most important, they serve as an easy-to-understand introduction to B-trees.

A B-tree is another kind of multiway tree that's particularly useful for organizing data in external storage. In this case, *external* means external to main memory, usually a disk drive. A node in a B-tree can have dozens or hundreds of children. We discuss external storage and B-trees at the end of this chapter.

Introduction to 2-3-4 Trees

In this section we look at the characteristics of 2-3-4 trees. Later we show how a Visualization tool models a 2-3-4 tree and how to program a 2-3-4 tree in Python. Figure 9-1 shows a small 2-3-4 tree. Each lozenge-shaped node can hold one, two, or three data items, making it a kind of **multiway tree**.

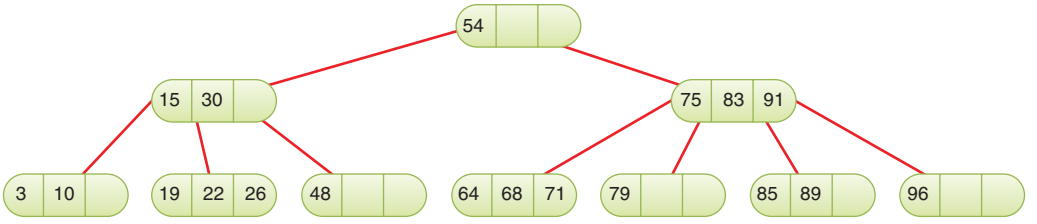


FIGURE 9-1 A 2-3-4 tree

Here, the top three nodes have children, and the seven nodes on the bottom row are all leaf nodes, which have no children, by definition. In a 2-3-4 tree, all the leaf nodes are always on the same level. Having all the leaves at the same level ensures the balance.

What's in a Name?

The 2, 3, and 4 in the name **2-3-4 tree** refer to how many links to child nodes can potentially be referenced by a given node. For nonleaf nodes, three arrangements are possible:

- A node with one data item always has two children.
- A node with two data items always has three children.
- A node with three data items always has four children.

In short, nonleaf nodes must always have one more child than they have data items. Or, to put it symbolically, if the number of child links is L and the number of data items is D, then

D = 1 or D = 2 or D = 3

L = 0 or L = D + 1

These critical relationships determine the structure of 2-3-4 trees. A leaf node has no children, but it can nevertheless contain one, two, or three data items. Empty nodes are not allowed. Figure 9-1 shows an example of each kind of node. There's a node with one data item and two children at the root. Below that, there is a node with two data items and three children on the left, and a node with three data items and four children on the right. The rest are leaf nodes with one, two, or three data items.

Because a 2-3-4 tree can have nodes with up to four children, it's called a *multiway tree of order 4*.

You may wonder why a 2-3-4 tree isn't called a 1-2-3-4 tree. Can't a node have only one child, as some nodes do in binary trees? A binary tree—described in Chapter 8, "Binary Trees"—might be thought of as a multiway tree of order 2 because each node can have up to two children. There are two important differences, however, between binary trees

and 2-3-4 trees. The obvious one is the maximum number of children—four in the case of 2-3-4 trees. In a binary tree, a node can have *up to* two child links. A single link, to its left or to its right child, is also perfectly permissible. The other link has an empty or `None` value.

In a 2-3-4 tree, on the other hand, there is a *minimum number* of two children for nonleaf nodes; internal nodes with a single link are not permitted. A node with one data item must either have two links or be a leaf, in which case it has no links.

2-3-4 Tree Terminology

The nodes in 2-3-4 trees are named for their number of child links. You might be tempted to call them by the number of data items they contain, but that could leave some doubt as to whether they are leaf nodes or not. Instead, you call the root node of Figure 9-1 a *2-node* because it has two children (and, by rule, one data item, 54). The child to the left of the root is a *3-node* because it has three children (and two data items, 15 and 30). The final internal node is a *4-node* with three data items: 75, 83, and 91. The bottom nodes are still called leaf nodes, not 0-nodes. In fact, even though they have no children, they are sometimes referred to as 2-nodes, 3-nodes, or 4-nodes based on the number of data items they contain.

For convenience, we number the data items in a node from 0 to 2, and the child links from 0 to 3. The data items in a node are arranged in ascending key order, by convention from left to right (lower to higher numbers). So, instead of left, middle, and right, we refer to data (or key) 0, data 1, data 2, child 0, child 1, child 2, and child 3.

2-3-4 Tree Organization

An important aspect of any tree's structure is the relationship of its links to the key values of its data items. In a binary search tree, all children with keys less than the node's key are in a subtree rooted in the node's left child, and all children with keys larger than or equal to the node's key are rooted in the node's right child. In a 2-3-4 tree, the principle is the same because they are always used for searching, but there's more to it:

- All children in the subtree rooted at child 0 have key values less than key 0.
- All children in the subtree rooted at child 1 have key values greater than key 0 but less than key 1.
- All children in the subtree rooted at child 2 have key values greater than key 1 but less than key 2.
- All children in the subtree rooted at child 3 have key values greater than key 2.

The keys are stored in ascending order within a node, usually inside an array. They are named by the indices to the array, not by labels such as "left," "middle," and "right." This relationship is shown in Figure 9-2. Duplicate values are not usually permitted in 2-3-4 trees, so you don't need to worry about comparing equal keys.

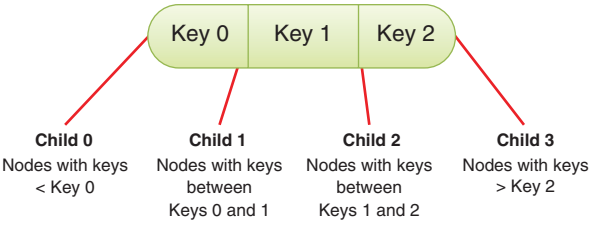


FIGURE 9-2 Keys and children

Look again at the tree in Figure 9-1. As in all 2-3-4 trees, the leaves are all on the same level (the bottom row). Upper-level nodes are often not full; that is, they may contain only one or two data items instead of three. Also, notice that the tree is balanced. It retains its balance even if you insert a sequence of data in ascending (or descending) order. The 2-3-4 tree's self-balancing capability results from the way new data items are inserted, as we'll describe in a moment.

Searching a 2-3-4 Tree

Finding a data item with a particular key is like the search routine in a binary tree. You start at the root and, unless the search key is found there, select the link that leads to the subtree with the appropriate range of values.

For example, to search for the data item with key 89 in the tree in Figure 9-1, you start at the root. You search the root but don't find 89 among its keys. Because 89 is larger than 54, you go to child 1, which is represented as 75-83-91. (Remember that child 1 is on the right because the numbering of children and links starts at 0 on the left.) You don't find the key in this node either, so you must go to the next child. Here, because 89 is greater than 83 but less than 91, you go to child 2. This time, you find the specified item in the 85-89 node.

Insertion

New data items are always inserted in leaves, which are on the bottom row of the tree. If items were inserted in nodes with children, the number of children would need to be changed to maintain the structure of the tree, which stipulates that there should be one more child than data items in a node.

Insertion into a 2-3-4 tree is sometimes quite easy and sometimes rather complicated. In any case, the process begins by searching for the appropriate leaf node.

If there is space in all the nodes encountered during the search, insertion is easy. When the appropriate leaf node is reached, the new data item is simply inserted into it. Figure 9-3 shows a data item with key 46 being inserted into the 2-3-4 tree from Figure 9-1. The path to the leaf node where 46 belongs is shown.

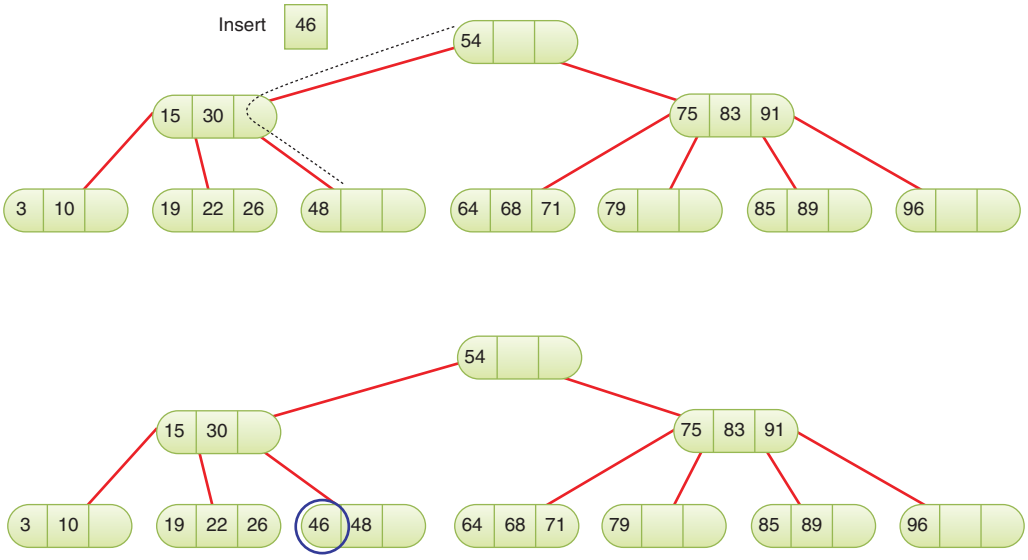


FIGURE 9-3 Insertion with no splits

Insertion may involve moving one or two other items in a node so that the keys will be in the correct order after the new item is inserted. In this example, the 48 had to be shifted right to make room for the 46.

Node Splits

Insertion becomes more complicated if a full node is encountered on the path down to the insertion point. A *full node* is one that contains three data items. When this happens, the node must be *split*. The splitting process keeps the tree balanced. The kind of 2-3-4 tree we're discussing here is often called a *top-down 2-3-4 tree* because nodes are split on the way down to the insertion point.

Here's what happens when the insertion process splits a nonroot node. (We examine splitting the root later.)

- A new, empty node is created. It's a sibling of the node being split and will be placed to its right.
- Data item 2 is moved into the new node.
- Data item 1 is moved into the parent of the node being split.
- Data item 0 remains where it is in the node being split.
- The rightmost two children are disconnected from the node being split and connected to the new node.

Figure 9-4 shows an example of a node split. Another way of describing a node split is to say that a 4-node has been transformed into two 2-nodes.

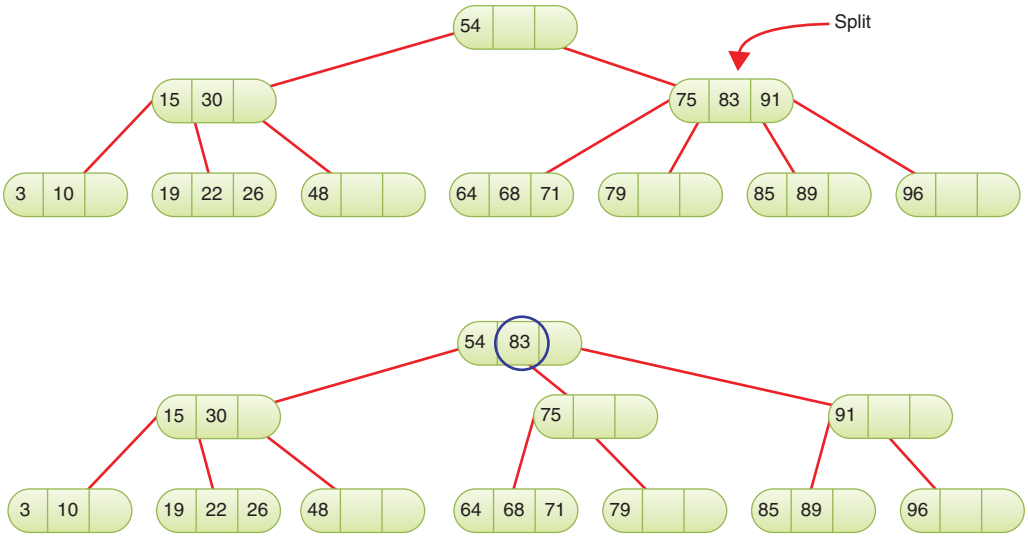


FIGURE 9-4 Splitting a nonroot node

Notice that the effect of the node split is to move data up or to the right. Key 1 from the split node moved up to the parent as circled in the figure. The 75 and 91 keys go into their own nodes as the children on either side of the promoted key. This rearrangement keeps the tree balanced.

The example shows only one node split, but more than one full node may be encountered along the path to the insertion point. When this is the case, there will be multiple splits.

Full nodes can occur anywhere in the tree. When the full node is a leaf, it has no children, such as nodes 19-22-26 and 64-68-71 in Figure 9-4. If it's an internal node, then it must have four children, as node 75-83-91 did before it was split.

Splitting the Root

When a full root node is encountered at the beginning of the search for the insertion point, the resulting split is slightly more complicated:

- A new root node is created. It holds data item 1 of the node being split and becomes the parent of the old root node.
- A second new node is created. It becomes a sibling of the node being split.
- Data item 2 of the split root is moved into the new sibling.
- The old root maintains data item 0 (discarding its references to data items 1 and 2).
- The two rightmost children of the old root node being split are disconnected from it and connected to the new sibling node.

Figure 9-5 shows the root being split. This process creates a new root that's at a higher level than the old one. Thus, the overall height of the tree is increased by one. Another way to describe splitting the root is to say that a 4-node is split into three 2-nodes.

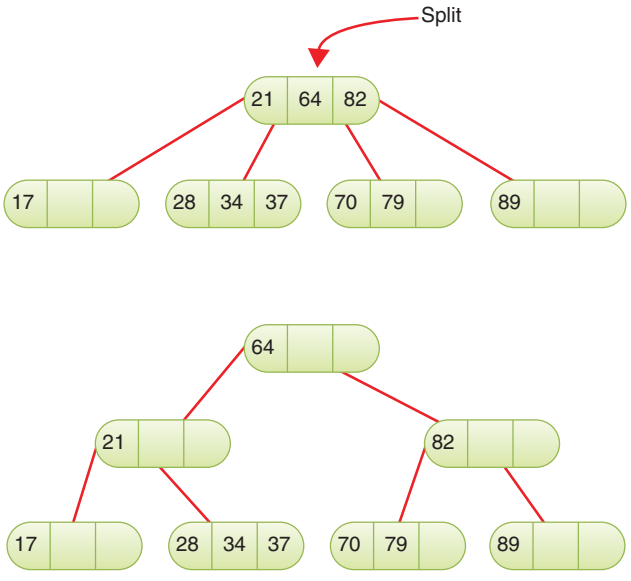


FIGURE 9-5 Splitting the root

As mentioned before, the node splits happen during the search for an insertion node. After each split, the search for the insertion point continues down the tree, possibly finding more full nodes that need to be split. In Figure 9-5, if the data item to be inserted had a key between 21 and 64, the search for the insertion node would lead to the node containing 28-34-37. That node would have to be split because it is full. Inserting in any other leaf node would not require another split.

Splitting on the Way Down

Notice that, because all full nodes are split on the way down, a split can't cause an effect that ripples back up through the tree. The parent of any node that's being split is guaranteed not to be full and can therefore accept data item 1 without itself needing to be split. Of course, if this parent already had two children when its child was split, it will become full with the addition of data item 1. Becoming full means that it will split when the next insertion search encounters it.

Figure 9-6 shows a series of insertions starting with an empty tree and building up to a three-level tree. There are five node splits: two of the root and three of leaves. After 79 is inserted, the tree is the same as in the top of Figure 9-5. Unlike that example, the next insertion of 49 in Figure 9-6 causes two splits: the one at the root and the one at the leaf node containing 28-34-37. Splitting the root adds the third level, and splitting the leaf node enables the insertion of 49 after 37. The tree remains balanced at every stage, and the number of levels increases whenever the root node is split.

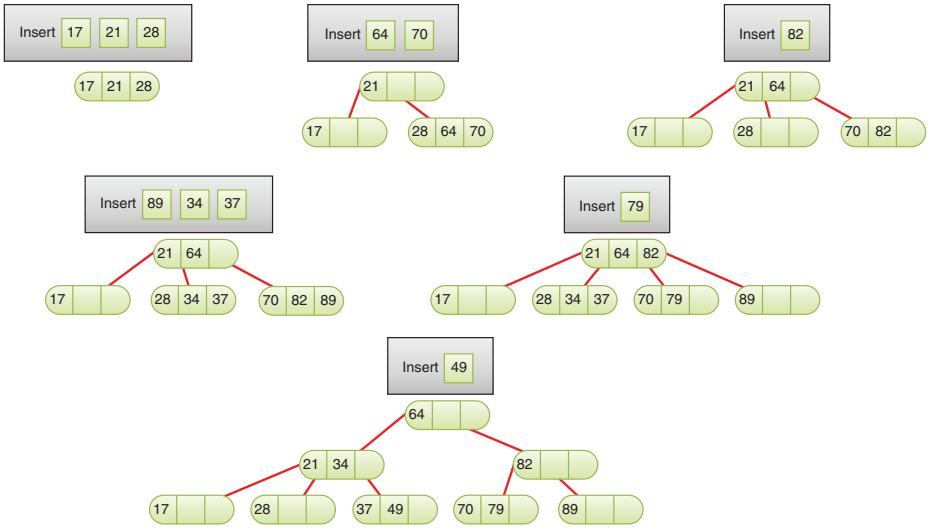


FIGURE 9-6 Insertions into a 2-3-4 tree

The Tree234 Visualization Tool

Operating the Tree234 Visualization tool provides a way to see the details of how 2-3-4 trees work. When you start the Visualization tool, you see a screen like Figure 9-7.

FIGURE 9-7 The Tree234 Visualization tool

The tree is initially empty when you start the tool. The `Tree234` label at the top is for the tree object itself, with an empty pointer to the root node. This matches the structure you saw for binary search trees in Chapter 8.

The Random Fill and New Tree Buttons

You can use the Random Fill button to add items to the tree. You type the number of data items to add in the text entry box and select Random Fill. Try adding a few items (up to three) to an empty tree to see the creation of the first node. As with the binary search tree, each item has a numeric key and a colored background to represent the data stored with the key.

Filling in many items (the maximum allowed is 99) will create a large 2-3-4 tree. Because 2-3-4 trees can be very "bushy" (wide), they are difficult to fit easily on the screen. The visualization provides tools for zooming and scrolling over the larger trees, but it is easier to learn how 2-3-4 trees operate by focusing on smaller trees. To keep all the nodes visible, fill empty trees with 10 or fewer data items.

If you create a large tree and want to return to smaller ones, use the New Tree button to return to the empty tree. You can then add new items either randomly or using the Insert button that we describe shortly.

The Search Button

You can watch the Visualization tool locate a data item by typing its key into the text entry box and selecting the Search button. Clicking an existing data item with your pointer device enters its key in the text entry box. You can use the Step and Pause/Play buttons to stop the animation to observe the individual steps.

A search involves examining one node on each level. Within each nonleaf node, the algorithm examines each data item, starting on the left, to see if it matches the search key or, if not, which child it should go to next. In a leaf node, it examines each data item to see whether it matches the search key. If it can't find the specified item in the leaf node, the search fails.

The Insert Button

The Insert button causes a new data item, with a key specified in the text box, to be inserted in the tree. The algorithm first searches for the appropriate leaf node. If it encounters a full node along the way, it splits that node before continuing. Splitting nodes widens the tree, so parts of the expanded tree may be placed outside the visible region. Splitting the root node makes the tree grow taller and narrower. For tall trees, the root node may appear far to the right, in anticipation of further splits at level 1 that broaden the tree.

Experiment with the insertion process. Watch what happens when there are no full nodes on the path to the insertion point. This process is straightforward. Then try inserting at the end of a path that includes a full node, either at the root, at the leaf, or somewhere in

between. Watch how new nodes are formed and the contents of the node being split are distributed among three different nodes.

Zooming and Scrolling

One of the problems with viewing 2-3-4 trees is that there are a great many nodes and data items just a few levels down. Keeping the number of levels to a minimum benefits the search speed, of course, but makes it hard to see all the items on a particular level when displayed horizontally. Levels 0, 1, and 2 are fairly easy to see, but when the tree grows to level 3 and deeper, the leaf nodes become spread over a long distance.

You can move around the tree using the scrollbars that appear when tree size is larger than what can be displayed in the window. You can also zoom in and out to focus on a particular part of the tree or try to see the overall shape. The Zoom In and Zoom Out buttons let you narrow or broaden the focus around what is in the center of the view.

You also can zoom by double-clicking the tree or its background. Each double-click zooms in a step while keeping the clicked point in the same place on the screen. Holding down the Shift key or using the second mouse button when double-clicking zooms out a step around the clicked point. When one or both scrollbars disappear, centering around the clicked point may no longer be possible. Scroll wheels should also work to change the zoom factor. Returning to an empty (New) tree restores the zoom factor and removes the scrollbars.

For example, if you start with an empty tree and insert 55 random data items, you produce a view something like Figure 9-8 where only 11 of the items are visible.

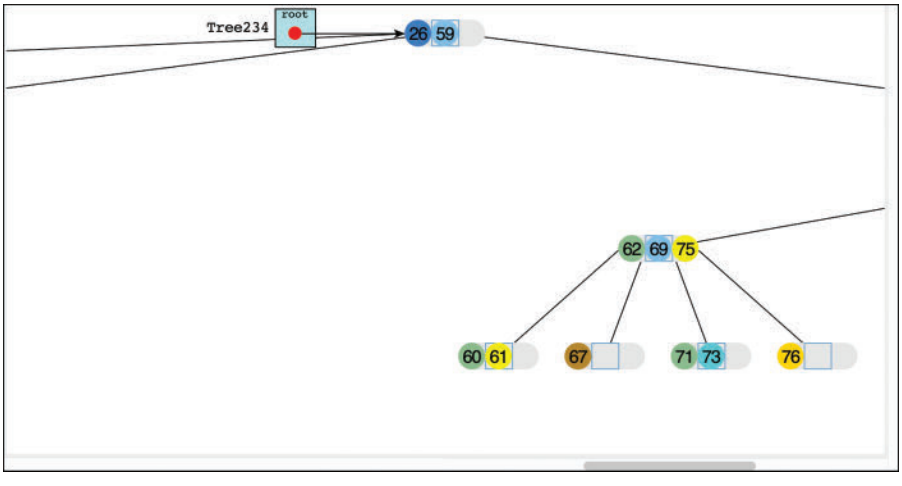


FIGURE 9-8 The zoomed-in view on a tree with 55 data items

If you zoom out five steps, the view becomes what's shown in Figure 9-9. You can see data items at all four levels now, but the individual items become illegible. The gaps exist to accommodate new nodes at the leaf level.

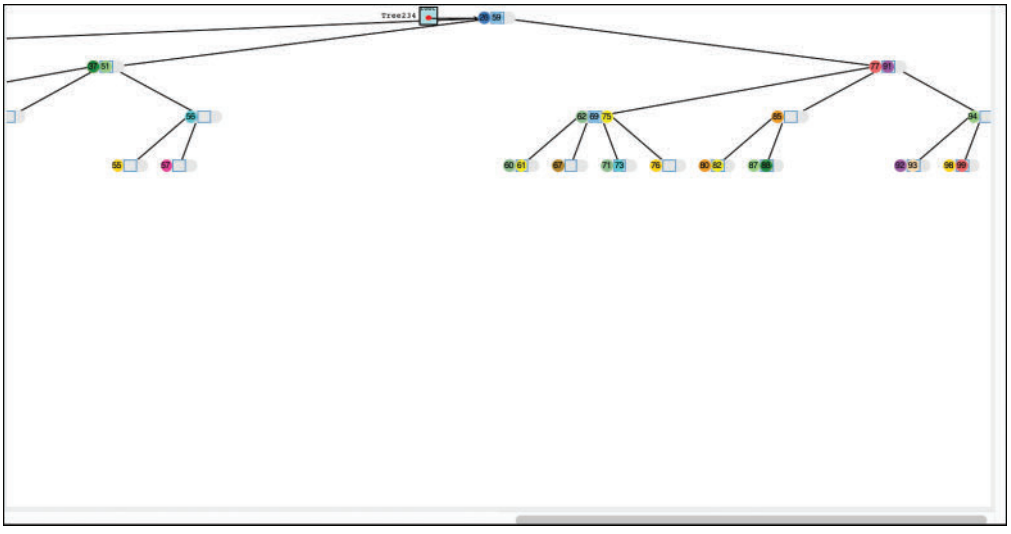


FIGURE 9-9 The zoomed-out view on the same 55 data item tree

Using the zoom and scrolling controls allows you to see both the big picture and the details and, we hope, put the two together in your mind.

During animated operations, the code keeps pointers to one or two nodes in the tree. The visualization tool attempts to keep those active pointers visible on the screen by changing the scrollbars. The program's manipulation of the scrollbars can lead to big jumps in viewpoint when you're zoomed in to a few nodes of large trees. The tool does not change the zoom factor, leaving that to you. Changing the zoom while the animation is paused is possible if you double-click but may leave some pointers in the wrong locations.

Experiments

The Tree234 Visualization tool offers a quick way to learn about 2-3-4 trees. Try inserting items into the tree. Watch for node splits. Stop before one is about to happen and figure out where the three data items from the split node are going to go. Then resume the animation to see whether you're right. As the tree gets larger, you'll need to move around it to see all the nodes.

How many data items can you insert in the tree? There's a limit because only keys with values 0 to 99 are allowed. Those hundred items could be packed into as few as 34 nodes because each node can hold three items.

How many levels can there be? At level 0, there is exactly 1 node. At level 1, there are 4, and at level 2, 16. A maximum-sized tree occupies at least 34 nodes, so it would have to have some nodes at level 3. Depending on the order the items are inserted, however, some splits could cause the tree to reach level 4. Are there cases when it could go to 5? to 6?

You can insert the most items in the fewest levels by deliberately inserting them into nodes that lie on paths with no full nodes, so that no splits are necessary. Of course, there is no way to guarantee such an ordering with real data.

Python Code for a 2-3-4 Tree

In this section we examine a Python program that implements a 2-3-4 tree. We introduce the code in sections. You can get the full `Tree234.py` program in the supplemental files with this text, and you can see most of it in the code window of the visualization tool. The program is relatively complex, although it's similar in some ways to the `BinarySearchTree` that was described in Chapter 8. It's often best to peruse the entire program in an editor to see how it works.

Like the `BinarySearchTree`, this data structure uses two classes: `Tree234` and `__Node`. `Tree234` is the public class, and `__Node` is a private class defined within it. Let's start with the `__Node` class.

The __Node Class

Objects of the `__Node` class represent the individual nodes of the tree. They manage the items stored at the node as well as the references to any child subtrees. We define the `__Node` class as a private class within the `Tree234` class to indicate that calling programs should not make changes to the tree's internal relationships.

As shown in Listing 9-1, the `Tree234` class defines relevant constants for the maximum number of child links and key-value pairs (data items) it can store. The `__Node` class uses these constants in allocating arrays for data. In this implementation, the keys, their corresponding values, and related children are kept in separate arrays. This arrangement contrasts with other data structures that store a single object for each data item and use a key function to extract the key value from the object. Both styles have advantages.

The constructor for `__Node` objects takes a single `key` and `data` value as input. The reason is that nodes in the 2-3-4 tree must always have at least one item stored in them; there's never an empty node. The node may be a leaf node or an internal node holding one item. Nodes are created when other nodes split (other than the first root). Depending on whether the node being split is the root, an internal node, or a leaf node, the node being constructed will have either two or no children. The asterisk before the `children` parameter of the constructor means that Python will interpret any arguments after the `data` argument as the references to the children nodes.

The `children` arguments that are the same type as the `__Node` object are placed in the `valid` list. That filtering is done by using a list comprehension—`[x for x in children if isinstance(x, type(self))]`—which uses `x` as a variable to go through all the `children` items and test that they are instances of this object's type. The first `if` statement after the list comprehension checks whether the caller passed an appropriate number of valid child nodes to the constructor. More than two is not allowed because the node will have only one item. A single child is never allowed in 2-3-4 trees. When the number of children isn't zero or two, the constructor raises an exception to report the problem.

After the number and type of children provided are verified, the constructor initializes the attributes of the object. The number of key-value pairs, `nKeys`, is set to 1. The `key` and `data` are put into arrays of size `maxKeys`. The constructor allocates the full array to enable shifting keys and values later when more items are added or removed. Note the arrays

are initialized to have three copies of the `key` and `data` in them, but the extra copies are ignored because `nKeys` is 1. Similarly, the child links are stored in the `children` array, and the quantity is kept in the `nChild` field. The children array is padded with `None` to ensure `maxLinks` cells are allocated.

LISTING 9-1 The `__Node` Class Within the `Tree234` Class

```
class Tree234(object):                          # A 2-3-4 multiway tree class

   maxLinks = 4                                 # Maximum number of child links
   maxKeys = maxLinks - 1                       # Maximum number of key-value pairs

   class __Node(object):                        # A node in a 2-3-4 tree
      def __init__(                             # Constructor takes a key-data pair
            self,                               # since every node must have 1 item
            key,                                # It also takes a list of children,
            data,                               # either empty or a pair that must
            *children):                         # both be of __Node type.
         valid = [x for x in children           # Extract valid child links
                  if isinstance(x, type(self))]
         if len(children) not in (0, 2):        # Check number of children
            raise ValueError(
               "2-3-4 tree nodes must be created with 0 or 2 children")
         self.nKeys = 1                         # Exactly 1 key-data pair is kept
         self.keys = [key] * Tree234.maxKeys    # Store array of keys
         self.data = [data] * Tree234.maxKeys # Store array of values
         self.nChild = len(valid)       # Store number of valid children
         self.children = (              # Store list of child links
            valid + [None] * (Tree234.maxLinks - len(valid)))

      def __str__(self):                # Represent a node as a string of keys
         return '<Node234 ' + '-'.join( # joined by hyphens w/ prefix
            str(k) for k in self.keys[:self.nKeys]) + '>'

      def isLeaf(self):                 # Test for leaf nodes
         return self.nChild == 0
```

We provide a `__str__()` method to enable inspection of nodes and printing trees. This function converts a node containing the keys 27 and 48 to the string `'<Node234 27-48>'` by joining the list of keys with the hyphen character. The keys put in the string are limited by the node's `nKeys` attribute. The `isLeaf()` method checks whether the node is a leaf with no children or an internal node.

After constructing a 2-3-4 node, you need to be able to insert other data items into it. In contrast to binary search trees, 2-3-4 nodes hold more than one key and datum. Insertion occurs after splitting any full 2-3-4 nodes and finding a leaf node where the new item's key belongs. Insertion can also occur when splitting a node and inserting key 1 in a parent

node. The `insertKeyValue()` method shown in Listing 9-2 takes a `key`, a `data` value, and an optional `subtree` to perform this action. The subtree is needed when inserting the item in an internal node during a split.

LISTING 9-2 The `insertKeyValue()` Method of `__Node` Objects

```
class Tree234(object):              # A 2-3-4 multiway tree class
…
   class __Node(object):            # A node in a binary search tree
…
      def insertKeyValue(           # Insert a key value pair into the
            self,                   # sorted list of keys. If key is already
            key,                    # in list, its value will be updated.
            data,                   # If key is not in list, add new subtree
            subtree=None):          # if provided, just after the new key
         i = 0                      # Start looking at lowest key
         while (i < self.nKeys and # Loop until i points to a key
                self.keys[i] < key): # equal or greater than goal
            i += 1                  # Advance to next key
         if i == Tree234.maxKeys: # Check if goal is beyond capacity
            raise Exception(
               'Cannot insert key into full 2-3-4 node')
         if self.keys[i] == key:  # If the key is already in keys
            self.data[i] = data   # then update value of this key
            return False          # Return flag: no new key added
         j = self.nKeys           # Otherwise point j at highest key
         if j == Tree234.maxKeys: # Before shifting keys,
            raise Exception(      # raise exception if keys are maxed out
               'Cannot insert key into full 2-3-4 node')
         while i < j:             # Loop over keys higher than key i
            self.keys[j] = self.keys[j-1] # and shift keys, values
            self.data[j] = self.data[j-1] # and children to right
            self.children[j+1] = self.children[j]
            j -= 1                # Advance to lower key
         self.keys[i] = key       # Put new key and value in hole created
         self.data[i] = data      # by shifting array contents
         self.nKeys += 1          # Increment number of keys
         if subtree:              # If a subtree was provided, store it
            self.children[i + 1] = subtree # in hole created
            self.nChild += 1      # This node now has one more child
         return True              # Return flag: a new key was added
```

Inserting a key in a sorted array is the same process the insertion sort used in Chapter 3, "Simple Sorting." Because we've chosen to store keys, values, and child trees in separate arrays with a related index, it's a little complex to reuse the `SortArray` class. Instead, we implement a simple `while` loop to find the insertion index of the new `key`. Because the

maximum array is only three elements long, we can use the simpler linear search rather than a binary search with little effect on performance. After `i` is set to the index where the `key` should be inserted, the loop exits, and the method raises an exception if the insertion would overflow the array.

The next `if` statement checks whether the `key` to insert duplicates an existing key. In that case, it simply updates the data value associated with that `key` and returns a Boolean flag to indicate that no new key was inserted. Alternatively, the method could raise an exception to disallow duplicate keys.

The next section of `insertKeyValue()` uses index `j` to index the highest key that must be stored in the node. When `j == Tree234.maxKeys`, the node must already be full, so an exception is raised, instead of overflowing the storage. The `j` variable points at the cell just after the last active key. The `while` loop shifts keys, data values, and child links to higher indices to make room for the insertion at index `i`. The references in the `children` array are offset by one from the corresponding keys and data arrays. After shifting all the cell values, the new key and data can be inserted at index `i`. The number of keys is increased and the new subtree is inserted, if provided.

The `insertKeyValue()` method finishes by returning `True` to indicate the insertion added a key and value. You see how that return value benefits the caller for different kinds of 2-3-4 tree inserts in the next section.

The Tree234 Class

An object of the `Tree234` class represents the entire tree. The class has only one field, `root`, which is either a `__Node` object or `None`. All public operations start at the root, but several methods can benefit from having private, recursive methods that operate on subtrees specified by a particular `__Node` object.

The constructor for `Tree234` objects shown in Listing 9-3 creates an empty tree by simply setting the `root` field to `None`. The `isEmpty()` method checks whether any items have been inserted in the tree by comparing `root` with `None`. The `root()` method uses it to raise an exception on empty trees. For trees containing nodes, it returns arrays of the data and keys that the root node contains. The Python array lengths (of the slices) tell whether the root node contains one, two, or three items.

Searching

Searching for a data item with a specified `goal` key, possibly for insertion, is carried out by the `__find()` routine. The one shown in Listing 9-3 is a private method that will return `__Node` objects for both the target node and its parent. This will be used for `search()` and `insert()` operations, which differ in whether full nodes should be split during the search process. The `prepare` parameter tells the method whether or not to split those nodes during the search.

By using the same routine to perform both searching and inserting, `__find()` must perform differently on duplicate keys depending on the operation. Searching for an existing key and inserting a duplicate key both call `__find()` with a `goal` that matches some key in the tree. For a search, `__find()` should locate the node with the key and return its associated data value. It doesn't need to split any full nodes because the tree isn't being altered.

LISTING 9-3 Constructor, Basic, and `__find()` Methods of the `Tree234` Class

```
class Tree234(object):        # A 2-3-4 multiway tree class
…
   def __init__(              # The 2-3-4 tree organizes items in
         self):               # nodes by their keys.
      self.__root = None      # Tree starts empty.

   def isEmpty(self):         # Check for empty tree
      return self.__root is None

   def root(self):            # Get the data and keys of the root node
      if self.isEmpty():      # If the tree is empty, raise exception
         raise Exception("No root node in empty tree")
      nKeys = self.__root.nKeys # Get active key count
      return (                # Otherwise return root data and key
         self.__root.data[:nKeys], # arrays shortened to current
         self.__root.keys[:nKeys]) # active keys

   def __find(self,                      # Find a node with a key that matches
              goal,                      # the goal and its parent node.
              current, parent,           # Start at current and track its parent
              prepare=True):             # Prepare nodes for insertion, if asked
      if current is None:                # If there is no tree left to explore,
         return (current, parent) # then return without finding node
      i = 0                                 # Index to keys of current node
      while (i < current.nKeys and          # Loop through keys in current
             current.keys[i] < goal):       # node to find goal
         i += 1
      if (i < current.nKeys and             # If key i is valid and matches goal
          goal == current.keys[i]):
         return (current, parent)           # return current node & parent
      if (prepare and              # If asked to prepare for insertion and
          current.nKeys == Tree234.maxKeys): # node is full,
         current, parent = self.__splitNode( # then split node, update
            current, parent, goal) # current and parent, and adjust i
         i = 0 if goal < current.keys[0] else 1 # for new current
      return ((prepare and current, parent) # Return current if
              if current.isLeaf() else      # it's a leaf being prepared
              self.__find(         # Otherwise continue search recursively
                 goal,                      # to find goal
                 current.children[i],       # in the ith child of current
                 current, prepare))         # and current as parent
```

For inserts, `__find()` should split the full nodes as the search descends the tree. As noted earlier, splitting full nodes while descending from the root maintains the tree's balance. If `__find()` starts an insert operation but ends up finding a duplicate of the `goal` key, it will update the data for that key. This makes the 2-3-4 tree behave as an *associative key store* for duplicate keys.

The `__find()` routine recursively descends the 2-3-4 nodes. It starts at the `current` node and tracks its `parent`. The caller normally passes the root node with `parent` pointing at the `Tree234` object itself. If `current` is `None`, `__find()` immediately returns it and the parent pointer, handling the base recursion case. When there is a `current` node, the first step is to determine where among the existing keys the `goal` key lies.

After setting index `i` to 0, the lowest key index, the `while` loop goes through all the valid keys, stopping when the i^{th} key is equal to or greater than the `goal`. Because the sorted array contains at most three keys, using a binary search for the goal key wouldn't save more than one comparison. After the loop, if the `goal` was found as a valid key, it can return the current node and its parent. Note that even if this node is full and `__find()` is preparing for an insert, there's no need to split it. Updating the data for a duplicate key doesn't require a split.

The next `if` statement handles splitting up the full nodes. When the `prepare` flag is true and the number of keys is the maximum allowed, it calls the `__splitNode()` method to redistribute those key-data pairs into separate nodes. Performing that split can change where the `current` and `parent` pointers should be, so `__find()` updates those variables from the multiple results of the call. Remember that splitting the root requires creating a new parent of the node being split, and that parent becomes the new root. Because the key-data pairs get moved, it also needs to update the `i` index in the `current` node. That `current` node will have only one key left in it (when split by the `__splitNode()` method, as you see shortly), so it only needs to check whether the goal is before or after the first key. When the goal is before, the `i` index is set to 0; otherwise, it is set to 1.

At this point, `__find()` hasn't found the goal, and it has performed any requested split. The search should continue in one of the child nodes if they exist. There are several cases here. If the `current` node is a leaf, then there are no children, and this is the node where the insertion should occur or the search should terminate. The complex `return` statement first checks if `current.isLeaf()` returns true. If so, it returns the `current` and `parent` nodes with one additional check. By returning `prepare and current` as the first result, `__find()` ends up returning `False` when the call is part of a search operation and returning `current` as part of an insert. Checking the `prepare` flag and conditionally returning `current` handles both kinds of operations on leaf nodes.

If the `current` node is not a leaf, then both searches and inserts must continue in a child node. The exact child depends on where the `while` loop finished. If the `goal` is less than the lowest key, `i` will be 0 and the lowest indexed child will be the next to explore. Every other key that was checked has incremented `i` and moved on to the next link in the `children` array. If all three keys were checked, `i` ends on the last child. The `__find()` method recursively calls itself with the i^{th} child. The `parent` pointer for that child is the `current` node as it descends the tree. If it had to split the current node, then it adjusted the `i` index to point at the appropriate child before making the recursive call.

Splitting Full Nodes

The method for splitting nodes containing three items, `__splitNode()`, is shown in Listing 9-4. The parameters are a reference to the node to be split, `toSplit`, its `parent` node, and the `goal` key that is being inserted in the tree. It handles splits for all three situations: the root node, an internal node, and a leaf node. All situations require making at least one new node that contains key (and data) 2. That `newNode` is a leaf node if the node `toSplit` is also a leaf node. The first `if` statement checks the node's position in the tree and passes the highest two child links to the `__Node()` constructor if they exist. These are the child links just below and above key 2.

LISTING 9-4 The `__splitNode()` method of `Tree234`

```
class Tree234(object):          # A 2-3-4 multiway tree class
…
   def __splitNode(             # Split a full node during top-down
         self,                  # find operation.
         toSplit,               # Node to split (current)
         parent,                # Parent of node to split (or tree)
         goal):                 # Goal key to find
      if toSplit.isLeaf():      # Make new node for Key 2, either as a
         newNode = self.__Node( # leaf node
            toSplit.keys[2],    # with key 2 and value 2 as its
            toSplit.data[2])    # sole key-value pair
      else:
         newNode = self.__Node( # or as an internal node
            toSplit.keys[2],    # with key 2 and value 2 as its
            toSplit.data[2],    # sole key-value pair and the highest
            *toSplit.children[2:toSplit.nChild]) # 2 child links
      toSplit.nKeys = 1         # Only key 0 and data 0 are kept in
      toSplit.nChild = max(     # node to split and child count is
         0, toSplit.nChild - 2) # either 0 or 2
      if parent is self:        # If parent is empty (top of 2-3-4 tree)
         self.__root = self.__Node( # make a new root node
            toSplit.keys[1],    # with key 1 and value 1
            toSplit.data[1],    # and the node to split plus new node
            toSplit, newNode) # as child nodes
         parent = self.__root # New root becomes parent
      else:                     # For existing parent node,
         parent.insertKeyValue( # insert key 1 in parent with
            toSplit.keys[1],    # new node as its higher subtree
            toSplit.data[1], newNode)
      return (toSplit           # Find resumes at node to split if goal
              if goal < toSplit.keys[1] # is less than key 1
              else newNode,     # else new node
              parent)           # Parent is either new root or same
```

After `__splitNode()` creates the `newNode`, the next statements update the number of keys and children of the node `toSplit`. That node will contain only key (and data) 0 after the split. The method leaves the references to the other keys, data, and child links in the arrays because it needs to reference key 1 shortly. Ideally, the array cells should be set to `None` to erase the extra references. Keeping only key 0 means the `toSplit` node now has either two or no child links, depending on whether it is a leaf node.

After `__splitNode()` alters the number of keys and child links, the next `if` statement checks whether the node `toSplit` is the root node by testing if its `parent` is the `Tree234` object itself. Splitting the root node requires making a second new node to serve as the root of the tree. The new root holds key (and data) 1 of the node `toSplit`. The new node becomes the parent of the node `toSplit`, which contains key 0, and the `newNode` containing key 2.

With the creation of the new root, the `parent` reference passed from the `__find()` method is set to that root. That `parent` pointer will be returned shortly, so the caller can recognize the new parent node.

If the node `toSplit` is not the root node, then it must be an internal or leaf node, and `__splitNode()` should insert its key (and data) 1 into the existing `parent` node by calling `insertKeyValue()`. Remember that the parent cannot be full because it would have been split on the previous call to the `__find()` method, which was the call on the `parent` node. You also know that the subtree holding keys just above key 1 is rooted at the `newNode` created earlier, holding only key 2.

At this point, after the second `if` statement, all the data items that were stored at the node `toSplit` have been placed in their new nodes, and those nodes have been linked to one another. It's time to return to the `__find()` method and update its `current` and `parent` pointers to where the search should continue. The `current` pointer should point to the subtree that contains the `goal`. That means the subtree to return is based on the relationship of the `goal` to the keys that were split. If the `goal` key is less than key 1 of the node `toSplit`, then the search resumes with that same node because it contains key 0 and any subtrees below it. Similarly, if the `goal` key is greater than key 1, the search should continue with the `newNode` that was created containing key 2 and its child links, if any. The `goal` key cannot match key 1 because that would mean it's a duplicate handled by the `__find()` method. The return value of `parent` is simple because it is always the variable, `parent`, although that could have been modified if the node `toSplit` was the root node.

You now have all the modules needed to define the public `search()` and `insert()` methods as shown in Listing 9-5. Both start by calling `__find()` on the root to get a pointer to the node that either holds or should hold the `goal` key. The `search()` method passes `False` for the `prepare` flag to avoid splitting nodes and allowing `__find()` to return `None` or `False` if the `goal` isn't found. If a `node` is returned, then one of its keys must match the `goal`. The `return` statement of `search()` selects one of the three `node.data` items by checking the quantity of keys in the `node` and the relationship of its key 1 to the `goal`. If no node was returned from `__find()`, then `search()` also returns `None`.

The `insert()` method is almost as simple. If no `node` was returned from `__find()`, then the tree should have no leaf (or other) nodes. That's different from the find methods

you've seen in other data structures, but remember, the `__find()` method will split nodes to make room for the key to insert. The second `if` statement checks the parent, `p`. If it is the `Tree234` object, then this is the first key being added to an empty tree. The root node of the tree is set to be a new `__Node` holding the `key` and `value` to insert with no children. It returns `True` to indicate that a new item was inserted. If the parent was something other than the `Tree234` object and `node` is `None`, then a bug has occurred, and it raises an exception. This could only happen if the `__find()` method started descending through some 2-3-4 nodes and ended up returning `None` for the insertion node.

LISTING 9-5 The `search()` and `insert()` Methods of `Tree234`

```
class Tree234(object):          # A 2-3-4 multiway tree class
…
   def search(self, goal):      # Public method to get data associated
      node, p = self.__find(    # with a goal key. First, find node
         goal, self.__root,     # starting at root with self as parent
         self, prepare=False) # without splitting any nodes
      if node:                  # If node was found, find key in node
         return node.data[      # data. It's the first data (index 0) if
            0 if node.nKeys < 2 or # there's only 1 key or
            goal < node.keys[1] else  # if the goal < key 1, else it's
            1 if goal == node.keys[1] # the 2nd data if goal == key 1
            else 2]                   # Otherwise it's the 3rd data

   def insert(self,             # Insert a new key-value pair in a
              key,              # 2-3-4 tree by finding the node where
              value):           # it belongs, possibly splitting nodes
      node, p = self.__find(    # First, find insertion node for key
         key, self.__root,      # starting at root with self as parent
         self, prepare=True)    # and splitting full nodes
      if node is None:          # If no node was found for insertion
         if p is self:          # Check if this the root
            self.__root = self.__Node( # Make a root node with just
               key, value)      # 1 key value pair
            return True         # and return True for node creation
         raise Exception(       # If not root, then something is wrong
            '__find did not find 2-3-4 node for insertion')
      return node.insertKeyValue( # Otherwise, insert key in node
         key, value)            # with no subtree, returning insert flag
```

When `__find()` returns a nonempty 2-3-4 node, the `insert()` method inserts the key-value pair in that node. The `insertKeyValue()` method returns `True` if a new key is inserted and `False` if an existing key is updated. The `insert()` method returns that flag to its caller.

As you can see, insertion in 2-3-4 trees is quite a bit more complicated than in binary search trees or ordered linked lists. If the code confuses you in spots, use the Visualization

tool and step slowly through the confusing spots. Seeing the code alongside the data structure as it is updated can help clarify the algorithm and purpose of each line of code.

Traversal

Traversing a 2-3-4 tree adds some new wrinkles to the possible tree traversal orders. The in-order traversal ought to visit each of the data items in order of ascending keys, similar to binary search trees. That's easy to do by first traversing child 0, and then alternating between the keys in the node and their corresponding "right" child. In terms of the child and key names for a node, that would be child 0, key 0, child 1, key 1, child 2, key 2, and child 3.

It's less clear what a pre-order or post-order traversal should do. One option for pre-order traversal would be to visit all the key-value pairs stored in a node before visiting any of its children. Another would be to visit the smallest key before visiting the first two children to mimic the behavior of a binary tree, and then visit key 1 before child 2 and key 2 before child 3. That approach somewhat preserves the in-order behavior for keys 1 and 2. A third pre-ordering would visit the first key and the first child, then the second key and the second child, followed by the third key and the third and fourth child. That last ordering is shown in Figure 9-10 along with the symmetrical ordering for post-order traversal.

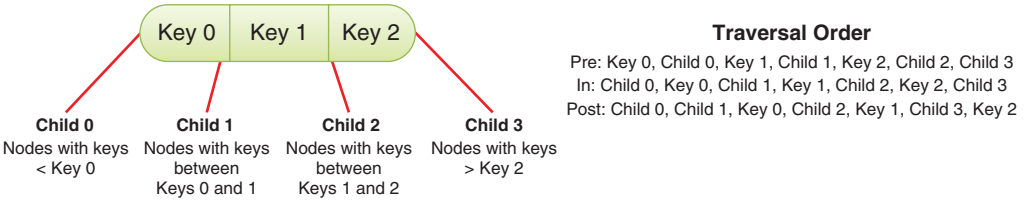


FIGURE 9-10 Traversing 2-3-4 trees

The exact ordering to implement depends on the planned uses of the data structure. An implementation of the traversal orders of Figure 9-10 in the `traverse()` generator is shown in Listing 9-6. Like with the binary search trees, implementing this using a nonrecursive generator is the most efficient way to yield keys paired with their data.

At the start, the `traverse()` method raises an exception if the requested `traverseType` is not one of the expected values: pre, in, or post. Next it creates a stack, using the linked list stack defined in `LinkStack` from Chapter 5, "Linked Lists." The stack is initialized to hold the 2-3-4 tree's root node. These are the same steps as in the `traverse()` method used in binary search trees.

The `while` loop processes items from the stack until it is empty. It pops the top `item` off the stack and looks at its type to determine what to do next. Like the `traverse()` method for binary search trees, there can be three types of items—a 2-3-4 node, a (key, data) tuple, or a `None` value—which occur for empty 2-3-4 trees or when leaf nodes push their child links on the stack.

LISTING 9-6 The `traverse()` Generator for `Tree234`

```
from LinkStack import *

class Tree234(object):        # A 2-3-4 multiway tree class
…

   def traverse(self,              # Traverse the tree in pre, in, or post
                traverseType="in"): # order based on type
      if traverseType not in [ # Verify traversal type is an
            'pre', 'in', 'post']:   # accepted value
         raise ValueError(
            "Unknown traversal type: " + str(traverseType))

      stack = Stack()              # Create a stack
      stack.push(self.__root)      # Put root node in stack

      while not stack.isEmpty():   # While there is work in the stack
         item = stack.pop()        # Get next item
         if isinstance(item, self.__Node): # If it's a tree node
            last = max(            # Find last child or last key index
               item.nChild,        # going 1 past last key for post order
               item.nKeys + (1 if traverseType == 'post' else 0))
            for c in range(last - 1, -1, -1): # Loop in reverse
               if (traverseType == 'post' and # For post-order, push
                   0 < c and c - 1 < item.nKeys): # last data item
                  stack.push((item.keys[c - 1], item.data[c - 1]))
               if (traverseType == 'in' and # For in-order, push
                   c < item.nKeys): # valid data items to yield
                  stack.push((item.keys[c], item.data[c]))
               if c < item.nChild:  # For valid child links,
                  stack.push(item.children[c]) # traverse child
               if (traverseType == 'pre' and # For pre-order, push
                   c < item.nKeys): # valid data items to yield
                  stack.push((item.keys[c], item.data[c]))
         elif item:                # Every other non-None item is a
            yield item             # (key, data) pair to be yielded
```

When the item to process is a 2-3-4 node, the various keys and child links must be pushed onto the stack in an order that depends on the traverse type. The number of keys and child links is different for each node, and leaf nodes differ from internal nodes in the number relationship. The method determines the `last` index of either a child node or key that it will need to process using the maximum of the number of children and the number of keys. For an internal node, the loop must cover all the child nodes. For a leaf node, the loop

must cover all the stored items. In the case of post-order traversal, the `last` index could be as high as the number of keys plus one, which is the same as the number of child links for internal nodes but not for leaf nodes. Adding one helps simplify the loop that comes next.

The `for` loop increments a variable, `c`, over the range 0 through `last - 1` in reverse order. This loop differs from what you've seen in other data structures, but it's needed to handle all the different types of nodes and traversal orders. The reverse ordering makes the items pushed on the stack come out in the desired order when they are popped by the outer `while` loop.

Inside the loop, four `if` statements control the order of visiting the keys and child links. The first one handles post-order traversal. This is the most complex of the three orders because you want to process the last key after the last child. In this case, you want to push the `c - 1` key on the stack so it is visited after child `c`. For instance, in a full 2-3-4 node, the last key, key 2, must be pushed on the stack before pushing on the last child, child 3. The stack items are processed in the reverse order, yielding key 2 after processing child 3. The `if` statement verifies that `c - 1` refers to a valid key before pushing a tuple with that key and its corresponding data onto the stack.

The second `if` statement handles the case of in-order traversal. In this case, the `c` key should be visited immediately after child `c`. If `c` refers to a valid key in the node, the `c` key and `c` data are pushed as tuple on the stack.

The third `if` statement handles the processing of child `c`. Depending on the traversal type, you could have already pushed on the (key, data) pair for either the `c - 1` key or the `c` key. When the method pushes child `c` on the stack now, that child is processed immediately before that key. This `if` statement checks that `c` refers to a valid child before pushing it on the stack (although it would be acceptable to push a `None` value). The type of this item is always a 2-3-4 node, which can be distinguished from the (key, data) tuples.

The fourth and final `if` statement handles the case of pre-order traversal. In this case, child `c` was just pushed on the stack, so the method pushes key `c` and its data to be processed before that child.

Traversal is usually the simplest operation of a data structure, but the various types of nodes and numbers of items within them complicate the process for 2-3-4 trees. The Visualization tool shows the execution of the three orderings along with the stack contents described here to help clarify the details.

Deletion

We showed how the insertion method can maintain a balance as items are inserted in the 2-3-4 tree. It would be nice if deleting a node did the same. Unfortunately, deleting while maintaining balance is quite a bit more complex than insertion. We review how you can accomplish this task but skip the detailed implementation for the sake of brevity.

Deleting at Leaf Nodes

Let's first consider the easy case. If you delete an item from a leaf node with more than one item in it, nothing needs to be done other than shifting items in the arrays of keys and data. There is still at least one item in the node, and its key remains in proper relation with the keys in the parent and sibling nodes. More importantly, the number of nodes and their levels are identical, so balance is maintained.

There's one more easy case for a leaf node. When you delete the only item in the root node and the root is also a leaf node, then it must be the last item in the tree. Deleting it means the tree is now empty. Deleting leaf nodes elsewhere in the tree might cause imbalances and we shall soon see.

Deleting at Internal Nodes

Now let's consider a deletion from an internal node. Deleting an item from the node would mean reducing the number of child nodes. That's possible in some circumstances but not always easy. You might be able to combine the two subtrees on either "side" of the key being deleted, but that could be messy. Is there another way? Remember deletion in binary search trees? We saw some simple cases and the more difficult case when the node had two children. Do you remember the "trick" for that case?

The idea is that you could replace the item to be deleted by *promoting its successor*. The successor item has the next higher key in the tree. There must be one because you're deleting from an internal node that has at least two child nodes. Finding the successor item is almost as simple as it was in the binary search tree. Start in the child node just to the "right" of the key being deleted. If you're deleting the i^{th} key of a node, you start looking for the successor in child i + 1. This is the subtree containing keys larger than the i^{th} key. Then you follow any child 0 links until you reach a leaf node. The lowest key in the leaf node identifies the successor.

Figure 9-11 shows an example of deleting the item with key 15 from a 2-3-4 tree. After locating the node holding keys 15 and 30, you find that 15 is key 0 and it's an internal node. That means the successor must be the smallest key in the subtree that is child 1 of that node. The child, node 19-22-26, is a leaf node so you don't have to descend any more levels (through the child 0 links), and the successor is the smallest key in that leaf node, 19. If you put item 19 in the node that was holding items 15 and 30, then you can delete item 19 from the leaf node. That's the simple case you already know how to handle. The bottom of Figure 9-11 shows the tree after deleting item 15. Item 19 was promoted to fill the hole created by deleting item 15, and items 22 and 26 were shifted left in the leaf node.

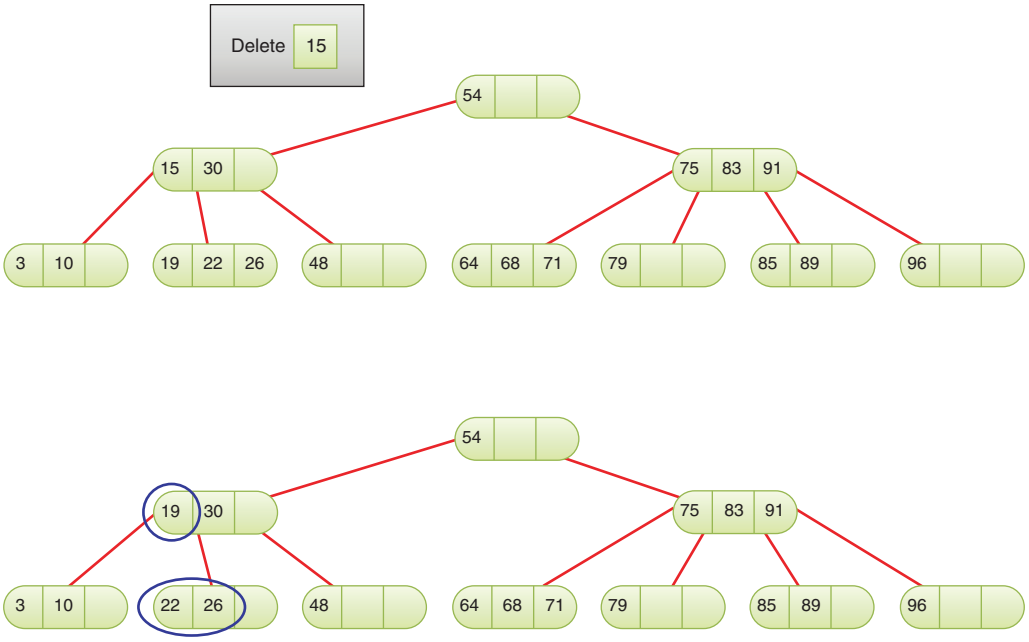


FIGURE 9-11 Deleting from an internal node by promoting the successor

To delete an item from an internal node whose successor lies in a leaf node with other items, you can now replace the item followed by the simple deletion of the item in the leaf. The tree maintains the same number of nodes and levels, so balance is maintained. Does this strategy work for all items in all internal nodes?

Consider deleting item 30 in the initial tree of Figure 9-11. This time you need to hunt for the successor in child 2 of node 15-30. The successor is 48, but there's a problem. Item 48 is the only item in the leaf node. If you promote it to replace 30, and then delete it from the child node, you will have an empty 2-3-4 node, and that's not allowed. Is there another way?

In this case, yes, there is. Just as there is a successor item, there is also a *predecessor* item for every item in an internal 2-3-4 node. Finding the predecessor is the mirror of finding the successor. The search starts in the i^{th} child and then follows the maximum child links in that subtree, if any, until it reaches a leaf node. In the leaf node, the maximum key identifies the predecessor item. If the predecessor is not the only item in the leaf node, you can promote it to replace the target item—key i—and delete the predecessor. In the example of deleting item 30, this ends up being a promotion of item 26 followed by the easy case of deleting 26 from leaf node 19-22-26.

Deleting from 2-Nodes

The technique of promoting a successor or predecessor is pretty powerful. All six of the items in internal nodes of the initial tree of Figure 9-11 have a successor or a predecessor that lies in a leaf node with more than one item. Unfortunately, you cannot count on the

predecessor or successor to always lie in a node with other items. You need some other techniques to deal with nodes with only one item, also called 2-nodes because they are either internal nodes with two children or leaf nodes with none.

There are two techniques to solve this problem. Consider deleting item 54 from the tree shown in Figure 9-12. By following the links, you can see that both the predecessor and successor contain only one item (and can be considered 2-nodes). Now you can't simply choose one of them to replace item 54 because deleting the single item in the leaf would break the rules of 2-3-4 trees.

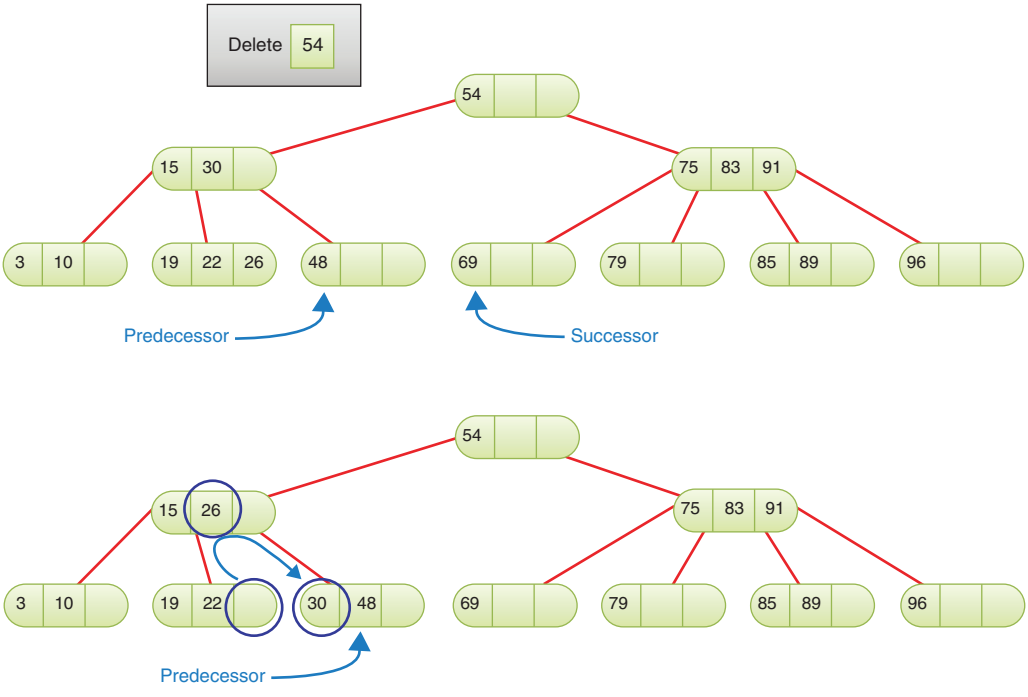


FIGURE 9-12 Deleting an item by rotating items into the predecessor node

To change the situation for the predecessor, we can look at its relationship to its sibling nodes. The predecessor in this case has one sibling holding three items (which can be considered a 4-node). Can we "borrow" one of them to make the predecessor have two? Well, not exactly, but we can shift the sibling's item up and move an item down from the predecessor's parent. This is shown in the bottom tree of Figure 9-12. You move item 26 from the sibling to the parent and item 30 from the parent to predecessor. The circles show where items changed.

Moving these two items is called a **rotation**. The rotation reconfigures the tree without breaking any of the rules of 2-3-4 trees. You still have the items stored in sorted order, and the tree is still balanced. By rotating item 26 up and item 30 down, you've created a tree

where you can do the simple deletion: remove predecessor item 48 from the leaf node and replace item 54 with item 48. That final step is not shown in Figure 9-12.

Rotations can be performed whenever the predecessor or successor has a sibling that is either a 3-node or a 4-node (when it has two or three items stored in it). Depending on which side the sibling lies, the rotation is either to the left or the right (shifting higher keys to the lower sibling or lower keys to the higher sibling). The parent can be any kind of node, 2-node, 3-node, or 4-node, because you are simply changing one item in it.

What if neither the predecessor nor the successor has a 3-node or 4-node sibling? That's what happens with the successor of item 54 in Figure 9-12; the successor item has key 69 and its only sibling is a 2-node containing item 79. Figure 9-13 shows the second technique that applies to this situation.

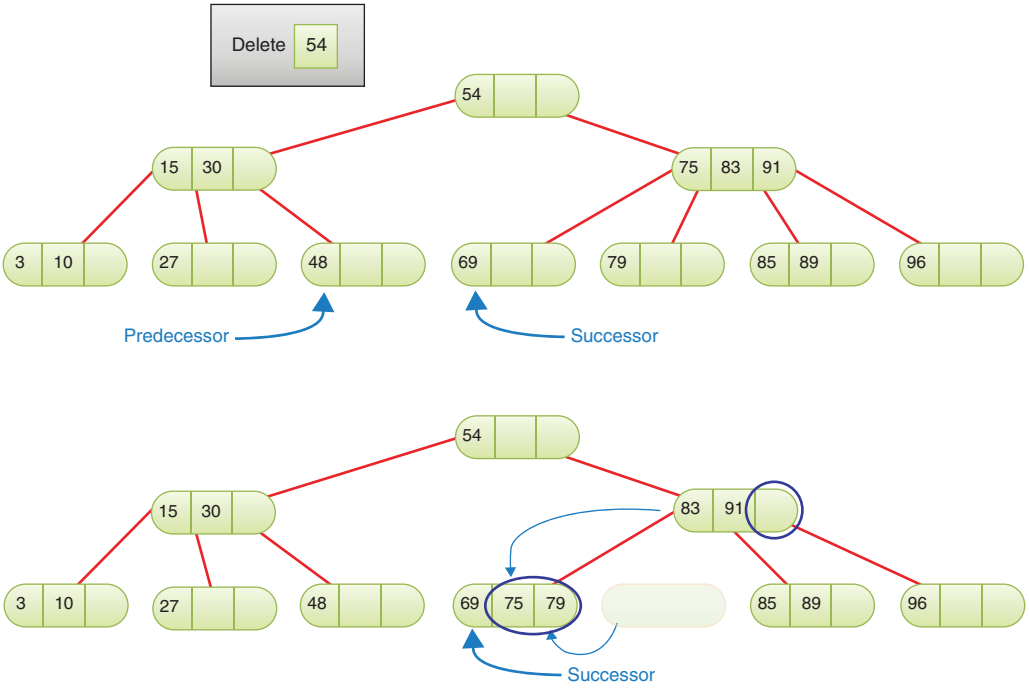


FIGURE 9-13 Deleting an item by fusing items into the successor node

When the 2-node that you're trying to fix only has siblings that are also 2-nodes, you can **fuse** the single item of the successor with that of a sibling and an item from their common parent, as long as the parent has two or three items (is a 3-node or 4-node). In the figure, the successor to item 54 is item 69, a 2-node. The only sibling to the successor is also a 2-node containing the single item 79. Because their common parent, node 75-83-91, is a 4-node, you can steal the item that separates the two siblings, item 75, and put it together with the single items to form a new 4-node, node 69-75-79, as shown in the lower tree. This is called a **fusion** operation.

As Figure 9-13 shows, fusing the two single item siblings with an item from their parent eliminates a node from the tree. That sounds like it might affect the balance, but if you look closely, it doesn't. The items remain in sorted order, and the number of levels hasn't changed on any path. The path to the deleted node went away, so it doesn't matter anymore. One item was removed from the parent of the successor, but that's fine because the corresponding child link was removed.

Using both rotation and fusion, you can make either a successor or a predecessor node into a multi-item node and then perform the deletion after replacing the item to delete higher in the tree. Although we've shown rotation on the predecessor and fusion on the successor, they really can be applied on either side. What matters is the number of items stored at the nodes and their parent.

Extending Fusion

Is that all the cases? Will rotation and fusion solve everything? The answer is: almost. We need to address a few more special cases, but they turn out to be variations on what you've already seen. An exception that doesn't fit any of the rules you've seen is a full binary tree. If every node in a 2-3-4 tree has exactly one item, then every node is a 2-node, like the top example shown in Figure 9-14.

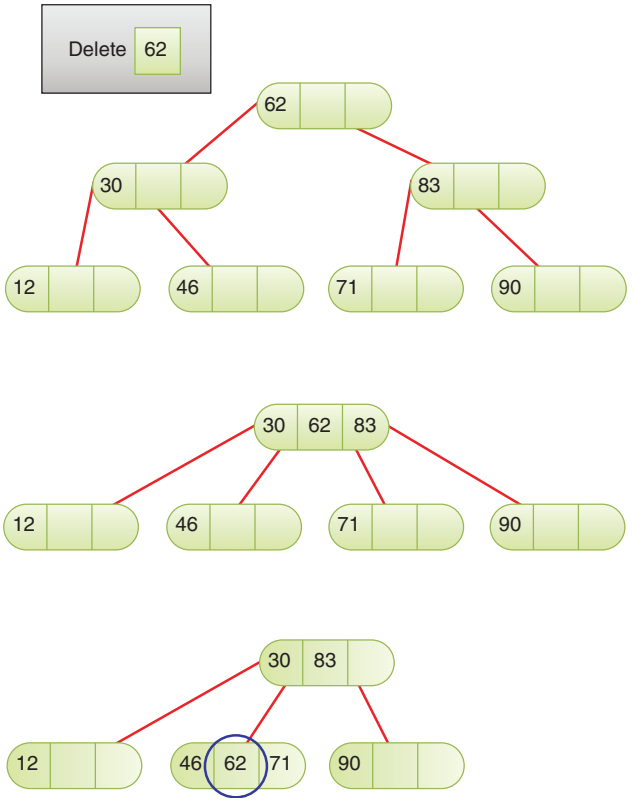


FIGURE 9-14 Deleting from a 2-3-4 tree with only 2-nodes

When every node is a 2-node, you can't use rotation or fusion as described earlier because both need a sibling or parent node to be either a 3-node or a 4-node. You obviously need some other technique to deal with this situation. There's another limitation of the operations described so far: the tree never shrinks. Each of the cases you've seen so far keeps the height of the tree (number of levels in a path to a leaf) the same. Replacing the item to delete with its successor and then deleting the successor from the leaf node leaves the same number of levels. Rotation and fusion also leave the same number of levels. Even deleting the last item from the root node that is also a leaf node technically changes the number of levels from zero to zero. How can the tree ever shrink from having two levels to one level with these techniques?

You can solve both limitations with a simple change to the fusion operation. If you allow fusion to occur when the parent is a *2-node that is also the root node* and replace the root node by the fused node, then the tree shrinks by one level. Fusion already eliminates a node; now you are extending it to eliminating a whole level. By limiting this fusion to only happen on the root node, it guarantees that all paths in the tree shrink by one level and preserves balance.

In the example in Figure 9-14, the top tree is transformed into the middle tree by fusing the top three nodes and their single items into one 4-node. The item at the root with key 62 becomes the middle item of the new 4-node. The items at the root's two children become item 0 and item 2 of the new 4-node. Child 0 and 1 of node 30 become child 0 and 1 of the 4-node. Child 0 and 1 of node 83 become child 2 and 3 of the 4-node. In other words, extended fusion collapses the top three 2-nodes into a single 4-node, removing one level of the tree.

From the middle tree in Figure 9-14, you can now delete item 62 using the operations you've already seen. Let's now focus on item 62's predecessor or successor. Both are 2-nodes, and both have only 2-nodes as siblings. Their parent, the new root, is a 4-node, so you can apply the fusion operation. The lowest tree in the figure shows the application of fusion to nodes 46 and 71 to make a new 4-node containing 46-62-71. This is one more special aspect of the fusion operation: it may move the item to be deleted into a lower position, possibly a leaf node to be deleted easily. In this case, item 62 started at level 0. The first fusion left it as item 1 of the new root, which is still level 0. The second fusion moved it to level 1, from which it can be deleted without reconfiguring nodes.

Applying Rotation and Fusion on Descent

It looks as though you now have all the rules to transform the tree and use simple deletion from leaf nodes. There is, however, one more thing to add in how to apply these operations. Consider what would happen if you asked to delete item 83 from the top tree in Figure 9-14. If you simply skip past the root, node 62, and then try to delete item 83 from the right subtree, you'll get stuck again because both the successor and predecessor are 2-nodes with a 2-node for a parent. Because the parent of the predecessor and successor isn't the root, you can't use the extended fusion operation (if you did, it would shorten the paths for only the leaves of this subtree without shortening the paths of all leaf nodes equally). Now what?

The problem here is that you didn't recognize at the root that you would need to apply a fusion operation below. If you had, you would end up with the middle tree in Figure 9-14, which makes it possible to delete 83 after another fusion operation. How can you know whether that fusion will be needed or not? The answer resembles what we did for insertion into the tree. Remember that as we followed the path to the insertion point, we split nodes that were full. That approach ensured that when we arrived at the insertion point, the parent node was not full and could accept another item. For insertion, we assumed we would need the full nodes to be split. What happens if you assume that you need 2-nodes to be collapsed as you descend the tree to hunt for the node to be deleted and its successor or predecessor?

When deletion is implemented for 2-3-4 trees, it requires an algorithm like the `__find()` method with the `prepare` flag that looks at every node encountered along the path and applies rotation or fusion to change any 2-node into a 3-node or a 4-node. In other words, you simply assume that nearly empty 2-nodes need to be collapsed. This algorithm also must track both the item to find and delete and the successor item as nodes are rearranged. All these extra details make it quite a bit more complicated than the insertion routine. There are many cases to examine with sibling nodes and the parent node, and a lot of rearranging of items and child links. In the rotation example we showed, the items being rotated were all at the leaf level and one above. When you apply rotation on internal nodes, the subtrees associated with the two deeper items being rearranged must follow those items. We look at that topic in more detail when we discuss rotation in red-black trees in Chapter 10, "AVL and Red-Black Trees."

The key points to remember about deletion in 2-3-4 trees are

- The tree can remain balanced as items are deleted.
- Deletion uses a modified version of the `__find()` method to locate the item to delete by descending through the tree comparing keys.
- After you find the item to delete, a similar descent of the tree is used to find the predecessor and successor of the item.
- Both descents apply rotation and fusion operations to 2-nodes encountered along the way to ensure that when the item to delete or the predecessor/successor is located, it's in a node with more than one item (or it's the root node with a single item).

Efficiency of 2-3-4 Trees

Fully analyzing the efficiency of 2-3-4 trees is hard, but they bear a lot of similarities to the binary trees. We can certainly determine the Big O complexity of operations and memory usage, and that's what's most important.

Speed

With the binary trees you saw in Chapter 8, one node on each level must be visited during a search, whether to find an existing node, insert a new one, or delete one. The number of levels in a balanced binary tree is about $\log_2(N)$, so search times are proportional to this.

One node must be visited at each level in a 2-3-4 tree as well, but the 2-3-4 tree is shorter (has fewer levels) than a binary tree with the same number of data items. To see this, compare the transformed trees in Figure 9-5 and Figure 9-14. In both cases, a 2-3-4 tree has been split or collapsed into a (partial) binary tree.

More specifically, in 2-3-4 trees there are up to four children per node. A full, single-node 2-3-4 tree has three items and height 0. A full 2-3-4 tree of height 1 has 5 nodes and 15 items. If every node were full, the height of the tree would be proportional to $\log_4(N)$, where N is the number of items (not nodes). Logarithms to the base 2 and to the base 4 differ by a constant factor, $2 \times \log_4(x) = \log_2(x)$ or $\log_4(x) = ½ \log_2(x)$. Thus, the height of a 2-3-4 tree would be about half that of a binary tree, provided that all the nodes were full. Because they aren't all full, the height of the 2-3-4 tree is somewhere between $\log_2(N)$ and $\log_2(N)/2$. The reduced height of the 2-3-4 tree somewhat decreases the path to search compared with binary trees.

On the other hand, there are more items to examine in each node along the path, which increases the search time. Examining the data items in the node using a linear search multiplies the search time by an amount proportional to M, the average number of items per node. Even if a binary search is used on the sorted keys in a node, you end up with a search time proportional to $M \times \log_4(N)$ because a binary search only saves one comparison compared to a linear search and only when there are three keys to compare.

Some nodes contain one item, some two, and some three. If you estimate that the average is two, search times will be proportional to $2 \times \log_4(N)$. If you convert that to logarithms of base 2, you get $2 \times \log_4(N) = 2 \times ½ \log_2(N) = \log_2(N)$. Thus, for 2-3-4 trees, the increased number of items per node tends to cancel out the decreased height of the tree. The search times for a 2-3-4 tree and for a balanced binary tree are approximately equal, and are both O(log N).

The story is similar for insertions and deletions. They each descend through the $\log_4(N)$ levels. Both insertions and deletions need to reach the leaf level to do their work. As they descend, the insertion process splits full 4-nodes and the deletion process collapses 2-nodes into 3-nodes or 4-nodes. These operations consume time, but the amount of time does not depend on the number of nodes in the tree; it's roughly constant for each node along the path. It does depend on the number of items per node, M. The node modification operations can be thought of as another constant factor, C, which ends up making the time to insert or delete $C \times M \times \log_4(N)$. That's still O(log N).

Traversal, of course, must visit every node, so its overall time is O(N).

Storage Requirements

Each node in a 2-3-4 tree contains storage for three keys and references to data items and four references to its children. This space may be in the form of arrays, as shown in `Tree234.py`, or of individual variables. Most 2-3-4 trees, however, do not use all this storage. A 2-node with only one data item will waste 2/3 of the space for data and 1/2 the space for children. A 3-node with two data items will waste 1/3 of the space for data and 1/4 of the space for children. If the spaces taken for a key, data, and child reference are identical and there are two data items per node as the average utilization, then 4 of the 6 cells for keys and data are used and 3 of 4 cells for child links are used, on average. That's 7 of 10 in use and 3 of 10 (30%) wasted.

You might consider using linked lists instead of arrays to hold the child and data references, but the time overhead of the linked list compared with an array, for only three or four items, would probably not make this a worthwhile approach.

The number of nodes is not the same as the number of items in a 2-3-4 tree. The number of nodes could be as low as 1/3 the number items if all the nodes are full (4-nodes). At the other extreme, there can be only one item per node (2-nodes), so the number of nodes ranges from N/3 to N, with the average expected to be N/2 items. In Big O notation, you treat all of those as O(N). That also means that the average amount of unused memory is O(N).

As traversals are performed, the algorithm needs to store information for each level of the descent of the tree. This storage can either take the form of recursive calls or an explicit stack of nodes that need to be visited. That stack will grow to the number of levels of the tree, so the memory needed is O(log N). Note that search, insertion, and deletion don't need recursion (although the implementation of `__find()` in `Tree234` did use it); they can be written to follow the chain of pointers without storing the full path back to the root. That means they need only O(1) space.

2-3 Trees

We discuss 2-3 trees here because they are historically important. Also, some of the techniques used with 2-3 trees are applicable to B-trees, which we examine in the next section. Finally, it's interesting to see how a small change in the number of children per node can cause a large change in the tree's algorithms.

There are many similarities between 2-3 trees and 2-3-4 trees except that, as you might have guessed from the name, they hold one fewer data item and have one fewer child. They were the first multiway tree, invented by J. E. Hopcroft in 1970. B-trees (of which the 2-3-4 tree is a special case) were not invented until 1972.

In many respects, the operation of 2-3 trees resembles that of 2-3-4 trees. Nodes can hold one or two data items and can have zero, two, or three children. Otherwise, the arrangement of the key values of the parent and its children is the same. The process

of inserting a data item into a node is potentially simplified because fewer comparisons and moves are potentially necessary. As in 2-3-4 trees, all insertions are made into leaf nodes, and all leaf nodes are on the same level, as in the sample 2-3 tree shown in Figure 9-15.

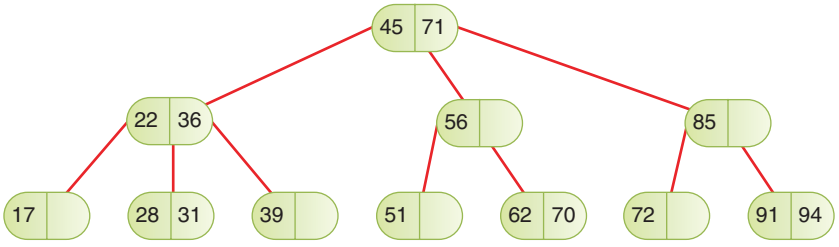


FIGURE 9-15 A 2-3 tree

Node Splits

You search for an existing data item in a 2-3 tree just as you do in a 2-3-4 tree, except for the number of data items and children. You might guess that insertion is also like that of a 2-3-4 tree, but there is a surprising difference in the way splits are handled.

Here's why the splits are so different. In either kind of tree, a node split requires three data items: one to be kept in the node being split, one to move right into the new node, and one to move up to the parent node. A full node in a 2-3-4 tree has three data items, which are moved to these three destinations. A full node in a 2-3 tree, however, has only two data items. Where can you get a third item? You must use the new item—the one being inserted in the tree.

In a 2-3-4 tree the new item is inserted after all the splits have taken place. In the 2-3 tree it must participate in the split. It must be inserted in a leaf, so no splits are possible on the way down. If the leaf node where the new item should be inserted is not full, the new item can be inserted immediately, but if the leaf node is full, it must be split. Its two items and the new item are distributed among these three nodes: the existing node, the new node, and the parent node. The three cases for how the items are distributed are shown in Figure 9-16.

In cases 0 and 2, the new item to insert goes into a leaf holding a single item. In case 1, when the new key falls between the two items that were in the leaf node being split, it must be passed to the parent node for insertion there. In all the cases, one item gets passed (or "promoted") to the parent node, as shown in the right part of the figure with the dashed black arrow pointing up. If the parent node has only a single item, the promoted item can be inserted in the node, a new child link attached, and the insertion is complete. If the parent node is full, the process must repeat, possibly continuing all the way up to the root node.

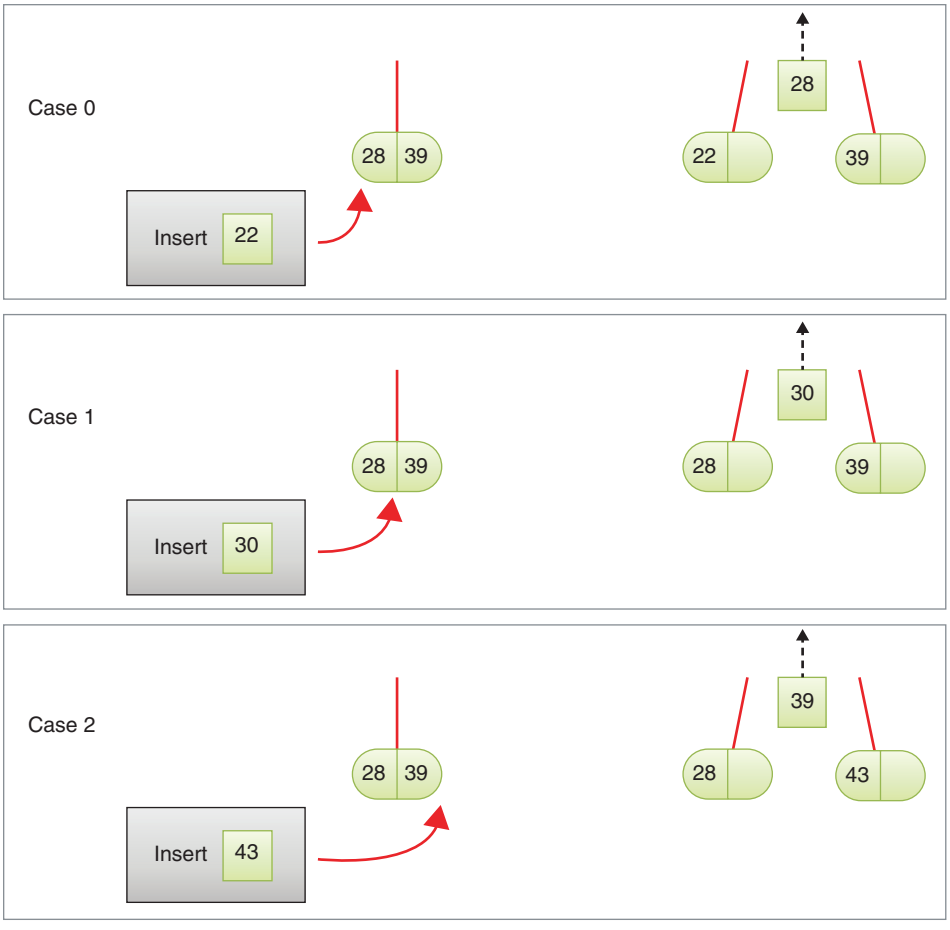


FIGURE 9-16 Insertion into leaf nodes that are full causing splits

For example, inserting an item with key 47 in the 2-3 tree of Figure 9-15 would add the item to leaf node 51, moving item 51 over to the right to keep the items in the node ordered. Because the leaf node could accept the new item, nothing is promoted to its parent.

If you try inserting item 67 into the same tree, you find leaf node 62-70 to be full. This is an example of case 1 because the new item's key is larger than one of the keys already stored there. Items 62 and 70 would be separated and item 67 would be promoted to the parent, node 56. Because the parent is not full, it can accept item 67 and the insertion is finished. Item 70 would be in the new split node while item 62 would remain in the leaf node that was split.

Promoting Splits to Internal Nodes

If you tried to insert a new item 27 in the 2-3 tree of Figure 9-15, the leaf node 28-31 would split and promote item 28 to its parent. That node 22-36 would split as well, and item 28 would be promoted again to its parent, the root node 45-71. The root node would also split, putting item 45 in the new root and making all paths to leaf nodes one level longer.

As you can see, the effects of promoting items up through internal nodes are complex. There's the item being promoted, and there's also the child links that need to be rearranged to keep the tree structure intact. The split cases shown in Figure 9-16 all show one item being promoted with two child links to the parent. The left-hand child link is the one that came from the parent when searching down for the leaf node to split. The right-hand link leads to the new node that was created by the split. It always contains a key that is larger than the key being promoted. That enables the parent node to insert the promoted item as item j and the promoted right-hand child link as child j+1. The original link to the node being split remains just to the left of the promoted item at child j.

Handling the child links gets a bit more confusing as the splits and promotions ripple up the tree. Figure 9-17 shows the cases that can occur when promotion finds an internal node that is already full. In each case, one item and one subtree are being promoted.

Just as for the leaf nodes, there are three cases, depending on which child link is the source of the split. What's tricky is that the child links and items all need different handling in the different cases. The examples in Figure 9-17 all show items 28 and 39 being split into different nodes but with different arrangements. Let's look at each case.

In case 0, the split came from child 0, the lowest subtree. The promoted key, 22, must be lower than the lowest key of the node being split. To maintain the key order, the promoted key must replace key 0, and that key, 28, will be promoted next to its parent. After replacing key 0, you also must replace child 1 of the node being split with the right subtree that was being promoted from below, the blue link in Figure 9-17. The blue link could lead to a simple leaf node or a large subtree, depending on the split operation that happened below. All that this part of the algorithm needs to know, however, is that the blue link belongs to the right of the item that just replaced item 0.

What about the old values of child links 1 and 2 of the node being split? Child link 1 of the node being split is shown in yellow in Figure 9-17. In case 0, child 1's keys are above that of the item being promoted (28) and below that of the item moved to the new split node, 39. Thus, the only place that child link 1 (yellow) can go is as child 0 of the new split node. Child link 2 of the node being split goes to child link 1 of the new split node.

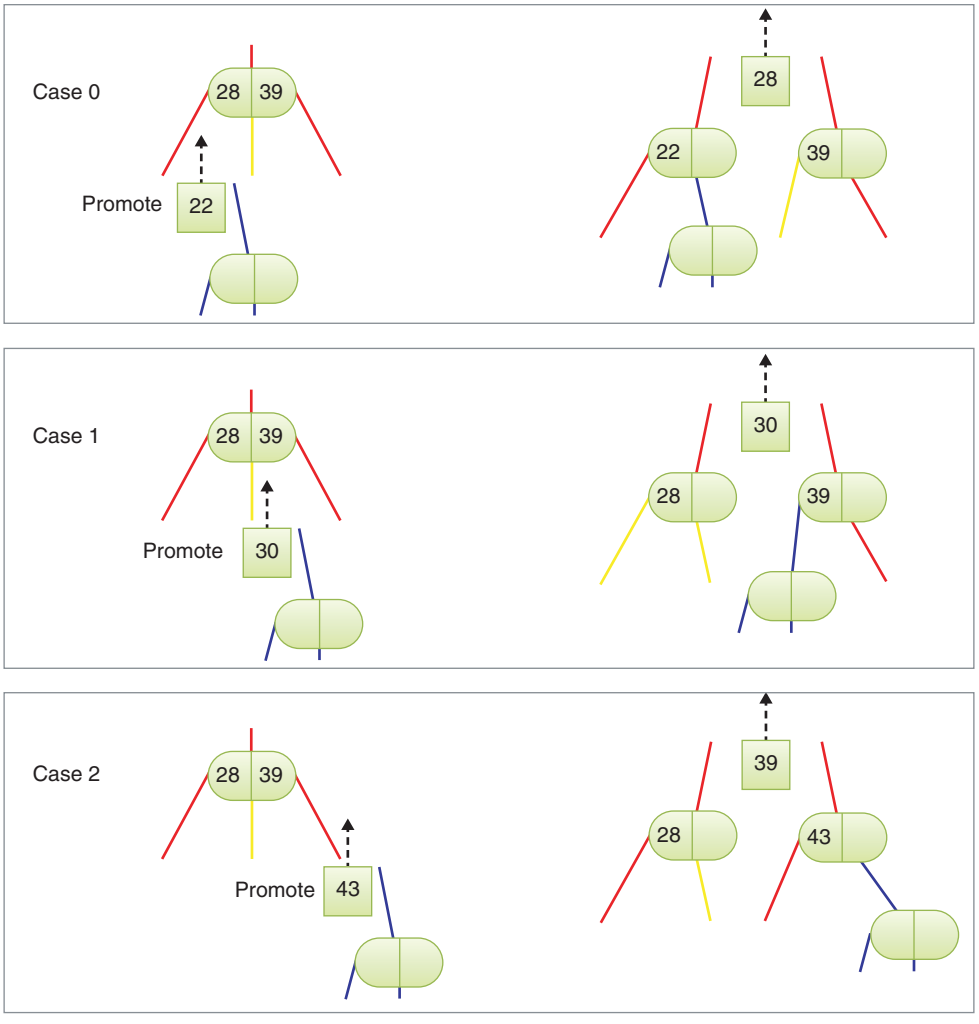


FIGURE 9-17 Splitting internal nodes

In case 1, the promoted item and subtree come from child link 1 of the node about to be split. In this case, the promoted item 30 continues to be promoted to the next parent. The promoted (blue) subtree, however, doesn't follow it up the tree. Instead, it becomes child 0 of the new split node because its keys are larger than item 30 (going up to the next parent) and less than the key of the new split node, 39. Child links 0 and 1 of the node being split remain as before, and child 2 moves over to become child 1 of the new split node because it holds all the keys above 39.

Case 2 is perhaps the easiest of the three. The promoted item key, 43, goes into the new split node, and the promoted (blue) subtree becomes child 1 of that split node. Item

39 gets promoted from the node being split to the next parent along with a link to the new split node, 43. Child 2 of the node being split moves over to become child 0 of the new split node because all its keys are greater than 39 and less than 43.

Implementation

We leave a complete Python implementation of insertion in a 2-3 tree as an exercise. We finish with some hints on how to handle splits.

On the way down, the insertion routine doesn't notice whether the nodes it encounters are full or not. It searches down through the tree until it finds the appropriate leaf. If the leaf is not full, it inserts the new value and the insertion is done. If the leaf is full, however, the routine must rearrange the tree to make room. To do this, there are a couple of options. It could call a `split()` method. Parameters to this method would include the full leaf node and the new item to insert. It would be the responsibility of `split()` to make the split and insert the new item in the new leaf, and then promote an item to the parent.

If `split()` finds that the leaf's parent is full, it would call itself recursively to split the parent. It keeps calling itself until a non-full node or the root is found. The return value of `split()` is the new right node, which can be used by the previous incarnation of `split()`.

How would `split()` know the parent node? If `split()` is called on a leaf node along with an item to insert, it doesn't have a direct link to the leaf node's parent, at least not if you follow the tree and node structure that we used for the 2-3-4 trees. There are several ways to address that. First, you could add a `parent` field to each `Node` object to have an explicit link in every node. The root node would have `None` or possibly the 2-3 tree object for its `parent` value. When creating nodes, the constructor would need a `parent` parameter. Then when you rearrange the child links while promoting items from splits, you would rearrange the parent links at the same time. This is about the same amount of effort needed to make doubly linked lists from singly linked ones, as discussed in Chapter 5.

The second option would be to keep a stack of node objects that represents the path to the node being split. The bottom of the stack would be the root node of the tree, and the top would initially be the leaf node to split. This stack could replace the node parameter to `split()`, and finding the parent would mean popping the top item off the stack. This process is fairly straightforward, although the `insert()` method would need to build the stack, or you could modify the `__find()` method to return it as a secondary return value.

The third option is to use the recursion that is naturally part of the `__find()` process in the `insert()` implementation. Each call to `insert()` would try to insert the new item if the node is a leaf and return an item to promote if the node was full. In other words, as `insert()` recursively calls itself on child nodes, it would check the return values to see whether it was done (no promoted item requiring more splits), or whether there is an item

to promote that must be inserted on the current node. If a promoted item is received from a recursive call and the node is not full, then the promoted item can be inserted, and no further splits or promotions are needed above that node in the tree. If the node is full, it makes the new split node, rearranges the items and child links, and returns the item to promote according to the cases described in Figure 9-17. The recursive changes ripple back up to the root, and if the root is full, it would be split too.

Coding the splitting process using any of these options is complicated by the case logic for the promoted items and their accompanying subtrees. We've mentioned the promoted item as a parameter for the `split()` method or as a return value for the recursive `insert()` method. There is both the key and data value for that item as well as the accompanying subtree that must be passed or returned.

Efficiency of 2-3 Trees

We haven't shown the deletion operation for 2-3 trees, but as you might imagine, it involves rotation and fusion operations like what was done for 2-3-4 trees. That means that 2-3 trees have largely the same efficiency as 2-3-4 trees. Search time is proportional to the height of the tree. If every node is full, each internal node has three children, and the height is $\log_3(N)$, where N is the number of items. With the same analysis as for 2-3-4 trees, you end up with a search being an O(log N) operation.

Insertion also descends the height of the tree but sometimes must go back up the path to the root, splitting full nodes to make room for the inserted item. That means insertion takes somewhere between $\log_3(N)$ and $2\times \log_3(N)$ steps. That's less efficient than the 2-3-4 tree, but only in terms of a constant multiplier. Overall both are still O(log N) operations.

In terms of memory usage, because it stores at most two items and three child links at each node, the amount of unused memory will be less for a 2-3 tree than for a 2-3-4 tree.

External Storage

Remember that 2-3-4 trees are examples of multiway trees, which can have more than two children and more than one data item. Another kind of multiway tree, the B-tree, is useful when data resides in external storage. **External storage** typically refers to some kind of disk system containing **files**, such as the hard disk found in many desktop computers and servers. For cloud computing, chunks of data are stored together and can be part of a **dataset** spread across many servers. In general, external storage could mean any storage system that is slower to access than main memory and addressable by some integer. The file systems used on external storage devices can be quite complex. For now, just think of them as a very large storage space.

In this section we begin by describing various aspects of external data handling. We talk about a simple approach to organizing external data: sequential ordering. Finally, we discuss B-trees and explain why they work so well with disk files. We finish with another

approach to external storage, indexing, which can be used alone or with a B-tree. We also touch on other aspects of external storage, such as searching techniques.

The details of external storage techniques are dependent on the operating system, language, and even the hardware used in a particular installation. Here, we stick with a general discussion to get the concepts across.

Accessing External Data

The data structures discussed so far are all based on the assumption that data is stored entirely in memory (sometimes called main memory or *RAM*, for random-access memory). Every element in RAM is accessible in the same amount of time if you have its address. In many situations, however, the amount of data to be processed is too large to fit in memory all at once. In this case, a different kind of storage is necessary. External storage generally has a much larger capacity than main memory—made possible by a lower cost per byte of storage.

Of course, disk files have another advantage over most RAM: their permanence. When you turn off your computer (or the power fails), the data in main memory is lost. Disk files and other **nonvolatile data storage** devices can retain data indefinitely with the power off. RAM is a volatile data store because its contents can be easily lost.

The disadvantage of external storage is its speed; it's *much slower* than main memory. This speed difference means that different techniques must be used to handle it efficiently. All important data must be kept on some kind of device where it won't be lost. Moving the data efficiently between slower permanent stores to faster volatile stores like RAM motivates the need for different kinds of data structures.

As an example of external storage, imagine that you're writing a database program to handle the basic contact data for everyone living in a particular state or country—perhaps 1 to 100 million entries. Each entry includes a surname, given name, phone number, address, and various other data. Let's say an entry is stored as a record requiring 1,024 bytes, a kilobyte. The result is a database size of at least $1{,}000{,}000 \times 1{,}024$, which is 1,024,000,000 bytes, or close to 1 gigabyte. There was a day when no computer on earth had a gigabyte of memory. Now gigabyte storage is commonplace, and machines can have terabytes of memory. We'll assume that there is some collection of data that is too large to fit in main memory but small enough to fit on the disk drive of some target machine. This could be for a larger collection of data, like all the credit card transactions in the world for a year.

Thus, we have a large amount of data on a disk or other slow drive. How do you organize it to provide the usual desirable characteristics of data structures: quick search, insertion, and deletion?

In investigating the answers, keep in mind two constraints. First, accessing external data is much slower than accessing it in main memory. Second, external data access typically reads or writes many records at once. Let's explore these points.

Very Slow Access

A computer's main memory works electronically. Any byte can be accessed just as fast as any other byte. The access time depends on the technology being used, but let's assume it takes something like 10 nanoseconds (10 billionths of a second). How much slower does it take to get a byte from external storage?

Let's first consider disk drives. Data is arranged in circular tracks on a spinning disk. To access a particular piece of data on a disk drive, the read-write head must first be moved to the correct track. This is done with a stepping motor or similar device; it's a mechanical activity that requires several milliseconds (thousandths of a second).

After the correct track is found, the read-write head must wait for the data to rotate into position. On average, it takes half a revolution for the head to be over the data. Even if the disk is spinning at 15,000 revolutions per minute, about 2 more milliseconds pass before the data can be read. After the read-write head is positioned, the actual reading (or writing) process begins; this might take a few more milliseconds.

Thus, disk access times of around 10 milliseconds are common. This is something like 1,000,000 times slower than a main memory that could access a cell in 10 nanoseconds. The ratio varies considerably with the devices, but it's always large, say above 100,000.

Next, let's consider flash memory, another form of external storage. Like disks, it retains its data even when power is turned off. Flash is faster to access than spinning disks but still slower than main memory. A flash memory device might take something like 10 to 100 microseconds (millionths of a second) to access a particular part of the data. Although that's much faster than the time to get the disk head positioned, it still is 1,000 or 10,000 times more than reading a cell from main memory.

In cloud computing environments, blocks of data are distributed over networked servers. Accessing a particular block stored on server A to be processed on server B means copying the data over the network. Network speeds grow faster every few years but can vary considerably with the traffic load. The network access speed compared to the speed of access to main memory is likely to be millions or billions of times slower.

The technology changes quickly, and speeds and costs of both main memory and external storage will change. What is likely not to change is that the fastest memory remains costly, either in terms of the money needed to buy it or the energy it consumes, compared with the slower external memory systems. The difference in speed is likely to remain on the order of thousands or millions to one. Hence, we will always need effective ways of accessing the higher-capacity, lower-cost external storage.

One Block at a Time

For disk storage, when the read-write head is correctly positioned and the reading (or writing) process begins, the drive can transfer a large amount of data to (or from) main memory quickly. For this reason, and to simplify the drive control mechanism, data is stored on the disk in chunks called *blocks*, *pages*, *allocation units*, *segments*, or some other name, depending on the system. Flash memory also provides access in terms of pages or blocks even though it doesn't involve a spinning disk. We call the chunks of data *blocks* in

this discussion. Notably, the chunks of storage do *not* correspond directly to files in external memory; files typically take many blocks.

The disk drive always reads or writes a minimum of one block of data at a time. Block size varies depending on the device and is usually a power of 2. For our contact database example, let's assume a block size of 8,192 bytes (2^{13}). That means a database of a million (1 kilobyte) records will require 1,024,000,000 bytes divided by 8,192 bytes per block, which is 125,000 blocks.

External memory accesses are most efficient when they read or write a multiple of the block size. If you ask to read 100 bytes, the system will read one block, 8,192 bytes, and ignore all but 100 of them. If you ask to read 8,200 bytes, it will read two blocks, or 16,384 bytes, and throw away almost half of them. The device must still spend time fetching the full block(s), regardless of the request size. By organizing your software so that it works with a full block or blocks of data at a time, you can optimize its performance.

Assuming the contact database record size is 1,024 bytes, you can store eight records in a block (8,192 divided by 1,024), as shown in Figure 9-18. Thus, for maximum efficiency, it's important to read eight records at a time (or multiples of this number).

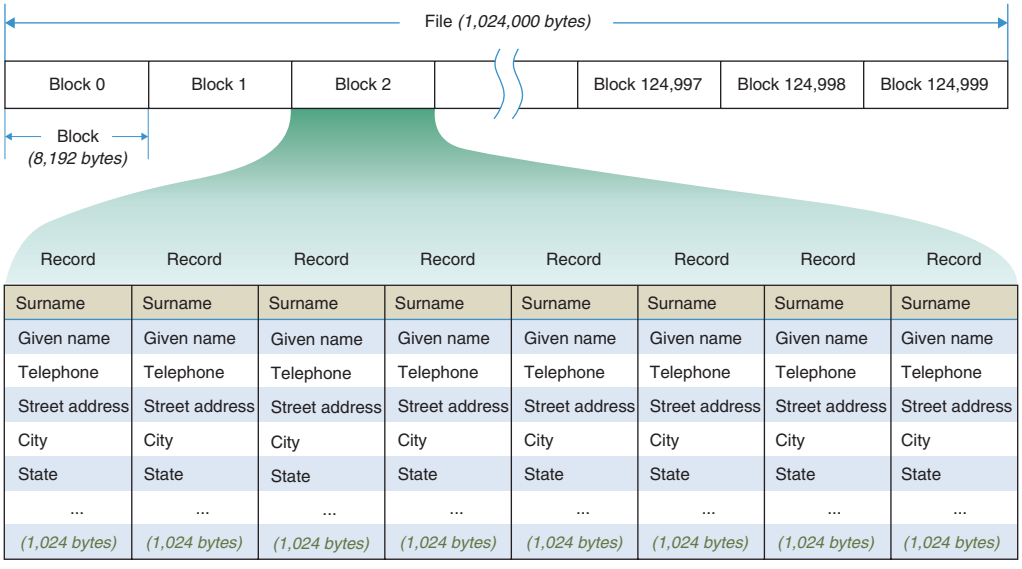


FIGURE 9-18 Blocks and records in a file

Notice that it's also useful to make your record size a power of 2 because that evenly divides the block size, which is typically a power of 2. In any case, it's best to have an integral number of records fit in a single block without any wasted space.

Of course, the sizes shown in the example for records, blocks, and so on are only illustrative; they will vary widely depending on the type, number, and size of records and other

software and hardware constraints. Blocks containing hundreds of records are common, and records may be much larger or smaller than 1,024 bytes.

After the read-write head of a disk is positioned as described earlier, reading a block is fast, requiring only milliseconds. Thus, a disk access to read or write a block is not very dependent on the size of the block. It follows that the larger the block, the more efficiently you can read or write a single record (assuming you use all the records in the block).

Sequential Ordering

One way to arrange the database in a file on the disk would be to order all the records according to some key, say alphabetically by surname. Maybe the record for a person named Aaron Aardvark would come first, and so on. This scenario is illustrated in Figure 9-19.

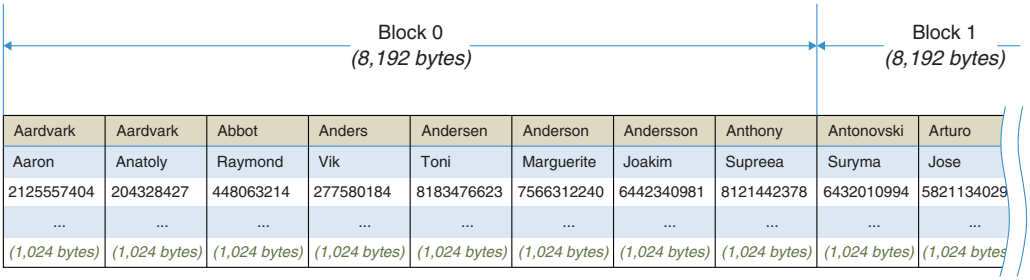


FIGURE 9-19 Sequential ordering of records

Searching

To search a sequentially ordered file for a particular surname such as Turing, you could use a binary search. You would start by reading a block of records from the middle of the file. Let's simplify the discussion and assume all the blocks of the file are in sequentially ordered blocks on the disk, so it's easy to find the middle block by knowing the first and last block numbers (this is rarely the case in actual file systems, but there are often ways to get the block numbers almost as quickly). The eight records in the block are all read at once into an 8,192-byte array (or **buffer**) in main memory.

If the keys of these middle records were too early in the alphabet (ending with Kahn, for example), you would go to the 3/4 point in the file and read a block there, perhaps finding Pearl; if the keys in the middle records came after the search key, you'd go to the 1/4 point, perhaps finding Englebart. By continually dividing the range in half, you would eventually find the target record.

As you saw in Chapter 3, a binary search in main memory takes $\log_2 N$ comparisons, which for 1,000,000 items would be about 20. If every comparison took, say 1 microsecond, this would be 20 microseconds, or about 1/50,000 of a second, much less than an eye blink.

In this section, however, we're dealing with data stored on a disk. Because each disk access is so time-consuming, it's more important to focus on how many disk accesses are

necessary than on how many individual records there are. The time to read a block of records will be very much larger than the time to search the eight records in the block once they're in memory.

Disk accesses are much slower than memory accesses, but on the other hand, you access the disk a block at a time, and there are far fewer blocks than records. In our example, there are 125,000 blocks. Log_2 of this number is about 17, so in theory, you need about 17 disk accesses to find the record you want.

In practice this number is reduced somewhat because you read eight records at once. In the beginning stages of a binary search, it doesn't help to have multiple records in memory because the next access will be in a distant part of the file. When you get close to the desired record, however, the next record you want may already be in memory because it's part of the same block of eight records. Having the next record in memory may reduce the number of disk accesses by two or so. Thus, you need about 15 disk accesses (17 – 2), which at 10 milliseconds per access requires about 150 milliseconds, or 0.15 seconds to complete the binary search of sequentially stored records. This is much slower than in-memory access of about 1/50,000 of a second, but still not too bad.

Insertion

Unfortunately, the picture is much worse if you want to insert (or delete) an item (a record) from a sequentially ordered file. Because the data is ordered, both operations require moving half the records on average and, therefore, about half the blocks.

Updating each block requires two disk accesses: one read and one write. When the insertion point is found, the block containing it is read into a memory buffer. The last record in the block is saved, and the appropriate number of records are shifted up to make room for the new one, which is inserted. Then the buffer contents are written back to the disk file as a block.

Next, the block following the insertion block is read into the buffer. Its last record is saved in main memory, all the other records are shifted up, and the last record from the previous block is inserted at the beginning of the buffer. Then the buffer contents are again written back to disk. This process continues until all the blocks beyond the insertion point have been rewritten.

Assuming there are 125,000 blocks, you must read and write (on average) 62,500 of them, which at 10 milliseconds per read and write requires more than 20 minutes to insert a single entry (20 ms × 62,500 = 1,250 seconds = 20.83 minutes). This isn't satisfactory by today's standards, especially if you have more than a few records to add.

Another problem with the sequential ordering is that it works quickly for only one key. Our example contact database is arranged by surnames. Suppose you wanted to search for a particular phone number or city. You can't use a binary search because the data is ordered by name. You would need to go through the entire file, block by block, using sequential access. This search would require reading an average of half the blocks, which would require about 10 minutes, which is very poor performance for a simple search. We obviously need a more efficient way to store large amounts of data in external memory.

B-Trees

How can the records kept in external memory be arranged to provide fast search, insertion, and deletion times? You've seen that trees are a good approach to organizing in-memory data. Do trees work with external memory such as files on a disk?

They do, but a different kind of tree must be used for external data than for in-memory data. The appropriate tree is a multiway tree somewhat like a 2-3-4 tree, but with many more data items per node; it's called a **B-tree**. B-trees were first conceived as appropriate structures for external storage by R. Bayer and E. M. McCreight in 1972. (Strictly speaking, 2-3 trees and 2-3-4 trees are B-trees of order 3 and 4, respectively, but the term *B-tree* is often taken to mean many more children per node.) Notably, B-trees are *not* binary trees, despite the similarity in their names.

One Block Per Node

Why would one need so many items per node? You've seen that disk access is most efficient when data is read or written one block at a time. In a tree, the structure containing data is the node (whether it be a node in a B-tree, a 2-3-4 tree, a binary search tree, or other). It makes sense then to store an entire block of data in each node of the tree. This way, reading a node accesses a maximum amount of data in the shortest time.

How much data can be put in a node? The answer depends on the size of the items, of course. In the contact database example using sequential records, you could fit eight 1-kilobyte data records into an 8,192-byte block.

In a tree, however, you also need to store the links to other nodes (which means links to other blocks because a node corresponds to a block). In an in-memory tree, such as those discussed in previous chapters, these links are references to nodes in other parts of memory. For a tree stored in a disk file, the links are to block numbers in the file (from 0 to 124,999, in the million record database example). For block numbers, you can use an integer field. If the integer takes 4 bytes (32 bits), it can reference a little more than 4 billion possible blocks, which might be enough. Bigger databases and external memory devices with even more addressable blocks would need even larger block numbers.

Organizing the data as tree nodes means that you can no longer squeeze eight 1,024-byte records into a block because you need room for the links to child nodes. You could reduce the number of records to seven to make room for the links that take 8×4-bytes, for example, but it's most efficient to have an even number of records per node and fill up the block. Perhaps the record size doesn't need to be the full 1,024 bytes; maybe a field or two can be reduced in size. Let's assume that the database designers would accept records of 1,010 bytes by shortening some of the fields. That means a node can still contain eight records along with nine child links (one more than the number of data items) requiring 36 bytes (9×4). The node would have $8 \times 1{,}010$-byte records plus 36 bytes of links and a couple more 4-byte integers that hold the number of keys and child links (`nKeys` and `nChild`), leaving 68 bytes unused (8,192 – 8,080 – 36 – 8). A block in such a tree (and the corresponding full node representation) is shown in Figure 9-20.

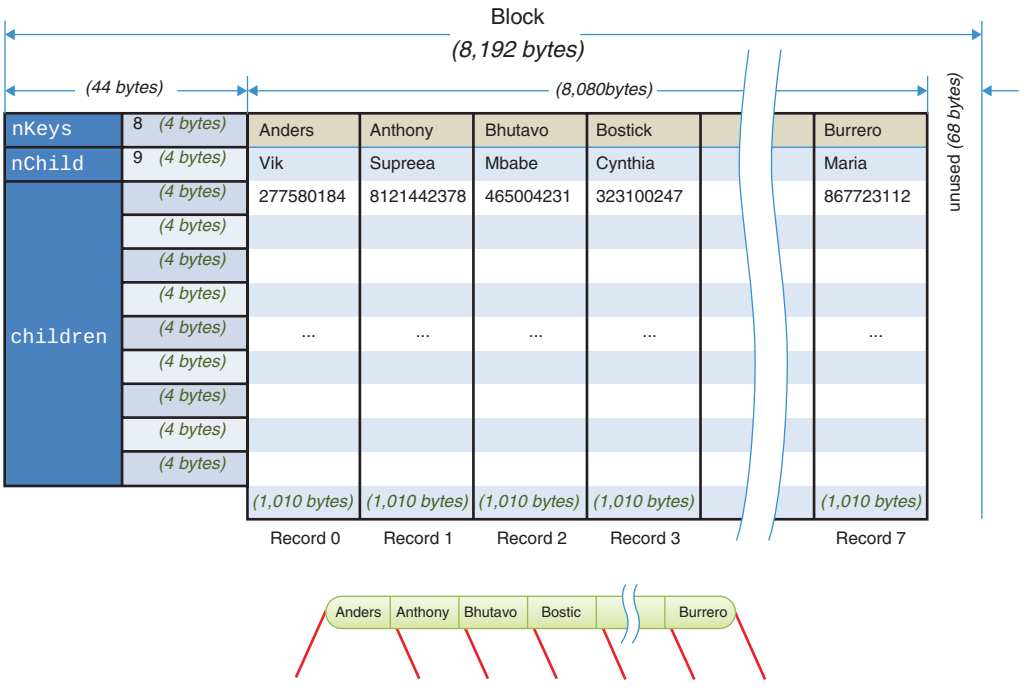


FIGURE 9-20 A node in a B-tree of order 9 with its block structure

Within each node the data is ordered sequentially by key, as in a 2-3-4 tree. In fact, the structure of a B-tree is like that of a 2-3-4 tree, except that there are more data items per node and more links to children. The order of a B-tree is the number of children each node can potentially have. In the example this is 9, so the tree is an order 9 B-tree.

If the full tree structure will be stored in external memory, which is what is typically done for a database, you also need to store the object that represents the tree. The `Tree234` object is used to hold that information in the in-memory implementation. All that must be stored in the corresponding B-tree object is the reference to the block of the root node. There probably would be other attributes stored for a large database that would go in this object. That information would probably be stored in a block all by itself, or perhaps in place of the first record of a block. Even though there would likely be a lot of unused bytes in the block, it's much more efficient in terms of disk accesses to keep it in a separate block and treat all tree nodes identically.

Searching

A search for a B-tree record with a specified key is carried out in much the same way it is in an in-memory 2-3-4 tree. Let's assume the tree object is already in memory with its reference to the block containing the root node. First, the block containing that root is read into memory. The search algorithm then starts examining each of the eight records (or, if it's not full, as many as the node holds), starting at 0. When it finds a record with a greater key, it knows to go to the child whose link lies between this record and the

preceding one. Note that you could use a binary search within the keys of a node, and it makes more sense to do so for large numbers of keys. Using binary search saves some in-memory comparisons, which is only a small savings compared to the time needed to read the external node/block.

This process continues until the correct node is found. If a leaf is reached without finding the specified key, the search is unsuccessful.

Insertion

The insertion process in a B-tree is more like an insertion in a 2-3 tree than in a 2-3-4 tree. Recall that in a 2-3-4 tree many nodes are not full and, in fact, contain only one data item. A 2-3-4 node split always produces two nodes with one item in each. This is not an optimum approach in a B-tree, even though splits don't have to create nodes with a single item.

In a B-tree it's important to keep the nodes as full as possible so that each disk access, which reads an entire node, can acquire the maximum amount of data. To help achieve this end, the insertion process differs from that of 2-3-4 trees in three ways:

- A node split divides the data items equally: half go to the newly created node, and half remain in the old one.
- Node splits are performed from the bottom up, as in a 2-3 tree, rather than from the top down.
- Again, as in a 2-3 tree, it's not the middle item in a node that's promoted upward, but the middle item in the sequence formed from the items in the node plus the new item.

We demonstrate these features of the insertion process by building a small B-tree, as shown in Figure 9-21. There isn't room to show a realistic number of records per node, so we show only eight; thus, the tree is an order 9 B-tree. We also switch to using integer keys to fit them all in the figure.

At the top left of Figure 9-21 is a panel labeled Insertion 1. It shows a root node after the first item, with key 60, has been inserted. The blue circle indicates the item that was just inserted. Below that is Insertion 8, where the eighth item, with key 40, has been inserted. This fills all the item slots in the root node. As usual, all the items are stored in sorted order.

In the next panel, Insertion 9, a new data item with a key of 75 is inserted, resulting in a node split. Here's how the split is accomplished. Because the root is being split, two new nodes are created (as in a 2-3-4 tree): a new root and a new node to the right of the one being split.

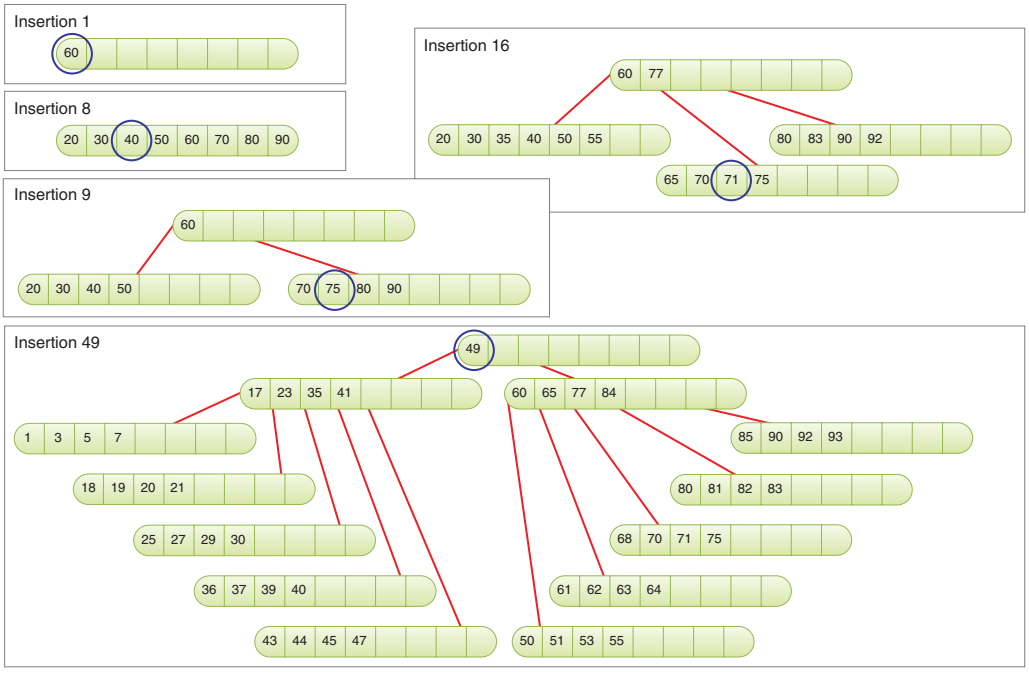


FIGURE 9-21 Building a B-tree

To decide where the data items go, the insertion algorithm arranges their nine keys in order, in an internal buffer. Eight of these keys are from the node being split, and the ninth is from the new item being inserted. This would go between item 70 and 80, leaving 60 as the middle item with four others on either side.

The middle item 60 is placed in the new root node. All the items to the left of it remain in the node being split, and all the items to the right go into the new right-hand node. In Insertion 9, the inserted item 75 ends up in the split leaf node, but that's not always the case, as you can see in later examples.

The number of items in a split node is always half that of a full node. So, if this were an order 31 B-tree, 15 items would go in the new split node. The single middle item would be promoted to the parent, and the original leaf node would be reduced to 15 items too.

It's clear after Insertion 9 that the B-tree has capacity to store many more items in the existing three nodes. Depending on their keys, newly inserted items go in one of the two leaf nodes. Let's insert six more items with keys 35, 65, 83, 55, 92, and 77. Two of those items, 35 and 55, go in the left leaf and the other four go in the right leaf. That brings the right leaf to full capacity with eight items, and the overall B-tree has 15 items in it.

On Insertion 16, item 71 arrives, and it must go in the right leaf. Because that leaf is full, it must be split. Item 71 lands before the middle of the eight items in that right leaf (65, 70, 75, 77, 80, 83, 90, 92). That means item 71 won't go into the new split node; it will stay with the "current" node being split. The four items with higher keys are split from the current node to form the new node. Item 77 is taken out of the current node and promoted to its parent, the root. The current node becomes child 1 of the root, and the new split node is child 2. In this case, the inserted item ends up in the left part of the split. The resulting B-tree is shown in the upper right panel of Figure 9-21. It has four nodes, and the 16th inserted item, 71, ended up in the central leaf.

Promoting item 77 from the leaf to the root doesn't cause more splits because the root only had a single item in it. Note that the three leaf nodes after Insertion 16 are all at the same level of the tree, level 1, even though we show them at different vertical positions in the figure.

The next insertions continue to add items to the leaves according to their keys. It takes at least five more inserts to split the newest leaves because they start off with four items each. As the splits occur, one item gets promoted to the parent, the root node. After at least 28 more inserts, the root can have nine child nodes each holding four items (44 items total = 9 $\times$ 4 + 8). After adding at least four more items, one of the child nodes can grow to holding eight items and become full. That brings the total to 48 items spread over the 10 nodes.

Insertion 49 of Figure 9-21 shows how a split could happen that expands the B-tree to two levels. For this to happen, the inserted item must go in a full leaf under a full root. Let's assume, somewhat coincidentally, that an item with key 49 is inserted. This item lands in a full leaf containing items 43, 44, 45, 47, 50, 51, 53, and 55. That node must split and, in this case, because 49 lands in the middle of the nine items, item 49 is promoted instead of one of the previously inserted items. That splits the full leaf node into two four-item leaf nodes shown at the bottom of the panel labeled Insertion 49 in Figure 9-21.

The promoted item 49 goes up to the root node, but because that is also full, the node must be split. Looking at the promoted item in relation to the items in the root, 17, 23, 35, 41, 60, 65, 77, and 84, shows that it again lands in the middle. Thus, item 49 gets promoted again, and the full root node is split in two. That is how you end up at Insertion 49 of Figure 9-21. Item 49 is promoted out of the root node so that it forms a new single-item node. The old root is split into nodes 17-23-35-41 and 60-65-77-84.

This (contrived) example shows how the inserted node could end up at an internal node or even in a new root. All the leaves remain at the same level in the tree, despite their positions in the figure.

Notice that throughout the insertion process no node (except the root) is ever less than half full, and many are more than half full. As we noted, keeping nodes as full as possible promotes efficiency because a file access that reads a node always acquires a substantial amount of data. The nodes are not split on the downward search to the node where the item should be inserted; they are only split as the insertion encounters a full leaf node and promotes the overflow back up toward the root. Thus, no unneeded splits cause expensive disk accesses.

Efficiency of B-Trees

Because there are so many records per node, and so many nodes per level, operations on B-trees are very fast, considering that the data is stored on slower storage like a disk. In the contact database example, there are a million records. All the nodes in the B-tree other than the root are at least half full, so they contain at least four records and five links to children. The height of the tree is thus somewhat less than $\log_5 N$ (logarithm to the base 5 of N), where N is 1,000,000. This is about 8.58, so there will be nine levels in the tree (height 8).

Thus, using a B-tree, only nine disk accesses are necessary to find any record in a file of 1,000,000 records. At 10 milliseconds per access, this takes about 90 milliseconds, or 9/100 of a second. This is dramatically faster than the binary search of a sequentially ordered file.

The more records there are in a node, the fewer levels there are in the tree. You've seen that there are nine levels in our example B-tree, with the nodes holding only eight records each. In contrast, a binary tree with 1,000,000 items would have about 20 levels, and a 2-3-4 tree would have 10. If you use blocks with hundreds of records, you can reduce the number of levels in the tree and further improve access times, although this is an option only for devices that support blocks large enough to hold all those records.

Although searching is faster in B-trees than in sequentially ordered disk files, it's insertion and deletion that demonstrate B-trees greatest advantages.

Let's first consider a B-tree insertion in which no nodes need to be split. This is the most likely scenario because of the large number of records per node. In our contact database example, at most nine block accesses are required to find the insertion point. Then one more access can write the block containing the newly inserted record back to the disk, for a total of 10 accesses.

Next let's see how things look if a node must be split. The node being split must be read, have half its records removed, and be written back to disk. That's the same as what was needed for an insertion with no split, 10 accesses. The newly created node must be written to the disk, and the parent must be read and, following the insertion of the promoted record, written back to disk. This adds three accesses to the 10 of the simpler case, for a total of 13 (although the parent node might remain in main memory after being read on the search for the insertion point, so the total could be 12). This number is a major improvement over the 500,000 accesses required for insertion in a sequential file.
As splits are promoted up the B-tree, they each add three more accesses per level, so at most $7 \times 3 = 21$ more accesses in an eight-level tree.

In some versions of the B-tree, only leaf nodes contain records. Nonleaf nodes contain only keys and block numbers. This strategy may result in faster operation because each block can hold many more block numbers. The resulting higher-order tree will have fewer levels, and access speed will be increased. On the other hand, programming may be complicated because there are two kinds of nodes, leaves with values and nonleaves without them, and the need to find a leaf node corresponding to an internal node's key to access the value.

Indexing

A different approach to speeding up external memory access is to store records in sequential order but use a **file index** along with the data itself. A file index is a list of key/block pairs, arranged with the keys in order. Recall that our original contact database example had 1,000,000 records of 1,024 bytes each, stored eight records to a block, in 125,000 blocks. To make a file index using surnames as the key, every entry in the index contains two items:

- The key, a string like `Jones`.
- The number of the block where the Jones record is located within the file. These numbers run from 0 to 124,999.

Let's say you use a string 59 bytes long for the key (big enough for most surnames including Unicode characters), 4 bytes for the block number (an integer), and a byte for the record number within the block. Each entry in such an index would require 64 bytes. This is only 1/16 the amount necessary for each record.

The entries in the index are arranged sequentially by surname. There could be multiple entries for common surnames. The original records on the disk can be arranged in any convenient order. This usually means that new records are simply appended to the end of the sequence, so the records are ordered by time of insertion. This arrangement is shown in Figure 9-22.

File Index in Memory

The file index is much smaller than the file containing actual records. It may even be small enough to fit entirely in main memory. In this example, there are 1,000,000 records. Each one has a 64-byte entry in the index, so the index will be $64 \times 1{,}000{,}000$, or 64,000,000, bytes long (a little less than 64 megabytes). In modern computers there's no problem fitting this index in memory (although it could be too big for a very tiny computer in some miniature device).

The file index can be stored in external memory and read into main memory whenever the database program is started up. From then on, operations on the index can take place in memory. At the end of each day (or whenever the database is shut down), the index can be written back to disk for permanent storage. The 64,000,000-byte index would take up 7,813 blocks on the disk. That might require several seconds to read or write, but only at startup and shutdown.

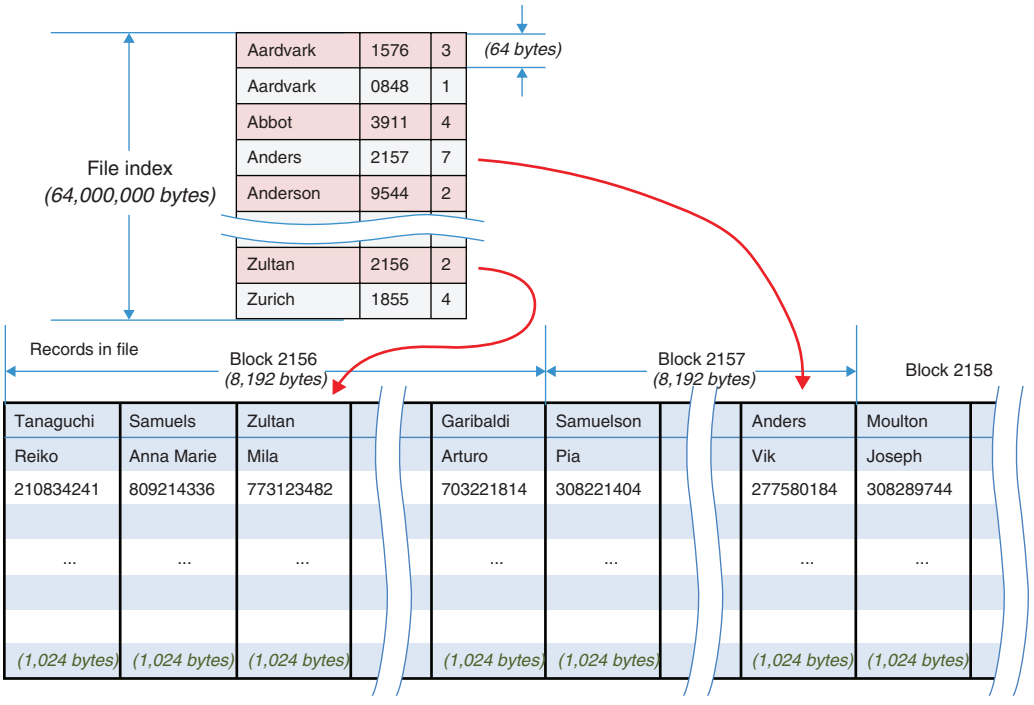


FIGURE 9-22 A file index example

Searching

The *index-in-memory* approach allows much faster operations on databases than are possible with records arranged sequentially in external memory. For example, a binary search could require 20 index accesses. Even at 0.1 microsecond (100 nanoseconds) per access, that's only about 2/1,000,000 of a second. The index must likely support duplicate keys, so the 20 index accesses would only identify the first of multiple entries to be checked. There's inevitably the time needed to read the actual records from external memory, after their block numbers have been found in the index. On a disk, the read operations might take one block access or about 10 milliseconds (1/100 of a second).

Insertion

To insert a new item in an indexed file, two steps are necessary. You first insert the item's full record into the main file; then you insert an entry, consisting of the key and the block number where the new record is stored, into the file index.

Because the file index is in sequential order, to insert a new item, you need to move half the index entries, on average. Figuring about 3 nanoseconds to move a byte in memory (one read and one write of 8 bytes takes about 20 nanoseconds), you have 500,000 entries times 64 bytes per entry times 3 nanoseconds per byte, or about .096 ($1/10^{th}$) seconds to

insert a new entry. This compares with 20 minutes for updating the unindexed sequential file. Note that you don't need to move any records in the main file; you simply append the new record at the end of the file. You would eventually need to write the file index to external memory, which would take about 72 seconds, writing another 7,182 (= 1,000,000 × 64 / 8,192) blocks at 1/100th of a second per block.

Of course, you can use a more sophisticated approach to storing the file index in memory. You could store it as a binary tree, 2-3-4 tree, or another multiway tree, for example. Any of these would significantly reduce insertion and deletion times. There's another structure that we discuss in Chapter 11, "Hash Tables," that is particularly fast. In any case the index-in-memory approach is much faster than the sequential-file approach. In some cases, it is also faster than a B-tree.

The only actual disk accesses necessary for an insertion into an indexed file involve the new record itself and eventually storing the file index. Usually, the last block in the file is read into memory, the new record is appended, and the block is written back out. This process involves only two file accesses. Storing the file index means writing thousands of blocks, but that is needed only when the database service is being shut down. The time spent preserving the file index is spread out over the hundreds, thousands, or perhaps millions of insertions and searches.

Multiple Indexes

An advantage of the indexed approach is that multiple indexes, each with a different key, can be created for the same set of records (database). In one index the keys can be last names; in another, telephone numbers; in another, addresses. Because the indexes are small compared with the data file, this doesn't increase the total data storage very much. Of course, it does present more of a challenge when items are inserted or deleted from the file because entries must be added to or deleted from all the indexes, but we don't get into that here.

Index Too Large for Memory

If the index is too large to fit in memory, it too must be broken into blocks and stored on the disk. For large files, storing the index itself as a B-tree may then be profitable. In the main file, the records are stored in any convenient order.

This arrangement can be very efficient. Appending records to the end of the main file is a fast operation, and inserting the index entry for the new record is also quick because the index is a tree. The result is very fast searching and insertion for large files.

Note that when a file index is arranged as a B-tree, each node contains n child pointers and n–1 data items. The child pointers are the block numbers of other nodes in the file index. The data items consist of a key value and a pointer to a block and record in the main file. Don't confuse these two kinds of block pointers.

Complex Search Criteria

In complex database searches, the only practical approach may be to read every block in a file sequentially. Suppose in our contact database example you wanted a list of all entries

with first name Kristen, who lived in Sao Paolo, and who had a phone number with the digits 284 in it. Imagine the name and phone number were found on a scrap of paper in a bag with a lot of money somewhere in Sao Paolo, and it's important to locate the full record for the person.

A file organized by last names would be no help at all. Even if there were index files ordered by first names and cities, there would be no convenient way to find which files contained both Kristen and Sao Paolo. Even less likely are indexes for three-digit subsequences of phone numbers (although they are sometimes created). In such cases the fastest approach is probably to use one of the indexes, if it exists, and read every record it references for the particular name, block by block, checking each record to see whether it meets the rest of the criteria.

Sorting External Files

You've seen how to use B-trees to manage a database stored in external memory. The data for each node, and hence each block, is sorted by a key. The sorting happens as items are inserted. What about when some other process wrote data to files, and you want to sort that data? Imagine a bunch of log files written by some process on a computer. Each record in the log probably has a date and time, along with other information that was relevant to the process such as the phase of the operations, the type of event that occurred, the size of some internal data structure, the account numbers for transactions, and so on. If you want to sort the log files by some field other than date and time (assuming they were recorded chronologically), and they don't all fit into internal memory, what's the best way to sort them?

Mergesort is the preferred algorithm for sorting large amounts of external data. The reason is that, more so than most sorting techniques, disk accesses tend to occur in adjacent records rather than random parts of the file.

Recall from Chapter 6, "Recursion," that mergesort works recursively by calling itself to sort smaller and smaller sequences. After two of the smallest sequences (one element each in the internal-memory version) have been sorted, they are then merged into a sorted sequence twice as long. Larger and larger sequences are merged, until eventually the entire file is sorted.

The approach for external storage is similar. However, the smallest sequence that can be read from the disk is a block of records. Thus, a two-stage process is necessary.

In the first phase, a block is read, its records are sorted internally, and the resulting sorted block is written back to disk. The next block is similarly sorted and written back to disk. This process continues until all the blocks are internally sorted.

In the second phase, two sorted blocks are read, merged into a two-block sequence, and written back to disk. This process continues until all pairs of blocks have been merged. Next, each pair of two-block sequences is merged into a four-block sequence. In each step, the size of the sorted sequences doubles, until the entire file is sorted.

Figure 9-23 shows the mergesort process on an external file. The file consists of eight blocks of 16 records each, for a total of 128 records. Instead of showing a record with a key in the figure, the records are shown as colored rectangles. The color and height of each rectangle are its key. Even though they are of different heights in the figure, the records on the disks have the same length in terms of bytes.

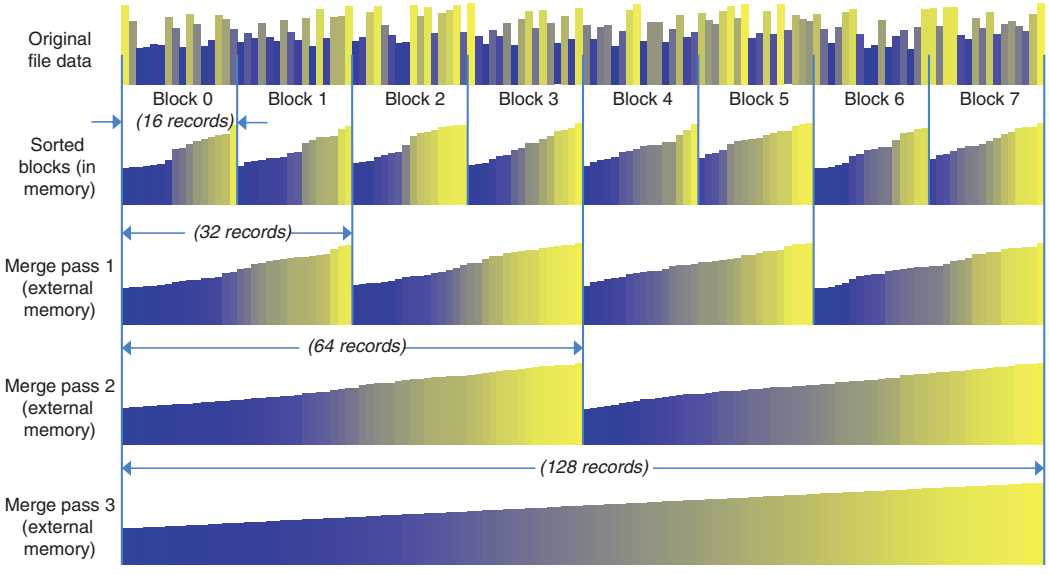


FIGURE 9-23 Mergesort on an external file

Let's assume that only three blocks can fit in internal memory, so you can't simply read all the records at once, sort them, and write them all out. Of course, all these sizes would be much larger in a real situation. The first row of Figure 9-23 shows the original, unsorted file data.

Internal Sort of Blocks

In the first phase, all the blocks in the file are sorted internally. Each block is read into memory and then its records are sorted with any appropriate internal sorting algorithm, such as quicksort (or, for smaller numbers of records, Shellsort or insertion sort). The internal sort could also be done with mergesort because the main disadvantage of mergesort is not an issue here. Remember that mergesort needs a second array the size of the original but that buffer must be available because mergesort needs it to merge two blocks later in memory. Whatever sorting algorithm is used, the sorted blocks of 16 records each are written back to external memory, as shown in row 2 of Figure 9-23.

A second file, File 2, may be used to hold the sorted blocks, under the assumption that the availability of external storage is not a problem. It's often desirable to avoid modifying the original file.

Merging

In the second phase, the sorted blocks are merged. The first pass merges every pair of 16-record blocks into a sorted two-block sequence. Thus, the leftmost two blocks, 0 and 1, are merged into the leftmost 32 records in merge pass 1. Also, blocks 2 and 3, blocks 4 and 5, and blocks 6 and 7 are merged into their respective sequences of 32 records each. The result is shown in the merge pass 1 row of Figure 9-23. Let's assume that the process writes a third file, file 3, to hold the result of this merge step (although that may not always be necessary).

In merge pass 2, two 32-record sequences are merged into a 64-record sequence, which can be written back to file 2. There are four of these sequences that get merged into two larger sequences. The last row of Figure 9-23 shows the result of merge pass 3 where the two 64-record sequences are merged into a 128-record file, file 3. Now the sort is complete. Of course, more merge steps would be required to sort larger files; the number of such steps is proportional to $\log_2 N$. The merge steps can alternate between two files, with one being discarded at the end.

Internal Arrays

Because the computer's internal memory has room for only three blocks, the merging process must take place in stages. Let's say there are three arrays, called `arr0`, `arr1`, and `arr2`, each of which can hold a block.

In merge pass 1, block 0 is read into `arr0`, and block 1 is read into `arr1`. These two arrays are then mergesorted into `arr2`. Because `arr2` holds only one block, it becomes full before the sort is completed. When it becomes full, its contents are written to disk in file 3. The sort then continues, filling up `arr2` again. This completes the sort, and `arr2` is again written to disk. The process moves on to blocks 2 and 3, reading them into `arr0` and `arr1` and merging their contents into `arr2`. As `arr2` becomes full, the results are written to the end of file 3.

In merge pass 2, you can continue using the three arrays, `arr0`, `arr1`, and `arr2`, even though the lengths of the sorted sequences have exceeded the array size. The input arrays, `arr0` and `arr1`, are initially filled with the first block of the sequence. After all the records from, say, `arr0` have been merged into the output array, `arr2`, the next block from that sequence can be read to refill `arr0`. You use arrays that are the size of a block (or perhaps a few blocks depending on how much memory is available) to efficiently read and write the external memory. Their size doesn't depend on the full amount of data to sort. This scheme works for all the merge passes.

We can make another important simplification to the external mergesort algorithm. In the first phase, it wrote the sorted blocks back to external memory. Because the first step of merge pass 1 is reading those blocks back into `arr0` and `arr1`, we can eliminate the first phase by performing the in-memory sort right after reading the unsorted block. The sort only happens on merge pass 1, and all the other merge passes only perform merges.

Efficiency of External Mergesort

The overall efficiency of mergesort remains O(N×log N), even when the data is in external memory. Due to the slow access to external data, however, the running time is much longer than if all the data could fit in internal memory. The number of external data accesses becomes the controlling factor. In the second phase, every block of the file is read, sorted if it's merge pass 1, merged appropriately, and then written back to the external store. Thus, there are two accesses (a read and write) for every block in every merge pass. That brings the total to two times the number of merge passes. The number of merge passes is proportional to $\log_2 N$ but is reduced by the fact that blocks contain multiple records, say B records per block. That makes the number of merge passes $\log_2 N/B$, or due to the way logarithms work, $\log_2 N - \log_2 B$. With the external memory accesses taking thousands or millions of times longer than the in-memory operations, they dominate the overall time.

Summary

- A multiway tree has more items and children per node than a binary tree.
- A 2-3-4 tree is a multiway tree with up to three items and four children per node.
- In a multiway tree, the items in a node are arranged in ascending order by their keys.
- In a 2-3-4 tree, all insertions are made in leaf nodes, and all leaf nodes are on the same level.
- Three kinds of internal nodes are possible in a 2-3-4 tree: A 2-node has one item and two children, a 3-node has two items and three children, and a 4-node has three items and four children. Each of the three kinds can be a leaf node too.
- There is no 1-node in a 2-3-4 tree.
- In a search of a 2-3-4 tree, the keys are examined at each node. If the search key is not found, the next node will be child 0 if the search key is less than key 0; child 1 if the search key is between key 0 and key 1; child 2 if the search key is between key 1 and key 2; and child 3 if the search key is greater than key 2.
- Insertion into a 2-3-4 tree requires that any full node be split on the way down the tree, during the search for the insertion point.
- Splitting the root creates two new nodes; splitting any other node creates one new node.
- The height of a 2-3-4 and a 2-3 tree can increase only when the root is split.
- The heights of 2-3-4 and 2-3 trees are less than $\log_2(N)$.
- Search times are proportional to the tree height.

- The 2-3-4 tree wastes space because many nodes might not even be half full.
- A 2-3 tree is like a 2-3-4 tree, except that it can have only one or two data items and zero, two, or three children.
- Insertion in a 2-3 tree involves finding the appropriate leaf and then performing splits from the leaf upward, until a non-full node is found.
- Both 2-3 and 2-3-4 trees maintain balance during inserts and deletes by keeping all leaf nodes on the same level.
- External storage means storing data outside of main memory, such as on a disk.
- External storage is larger, cheaper (per byte), and slower than main memory.
- Data in external storage is typically transferred to and from main memory a block at a time.
- Blocks can be different sizes but are typically hundreds to tens of thousands of bytes.
- Accessing an arbitrary block of external memory is thousands or millions of times slower than accessing arbitrary internal memory.
- Although accessing an arbitrary block is slow, accessing consecutive blocks is typically much faster.
- Data can be arranged in external storage in sequential key order. This gives fast search times but slow insertion (and deletion) times.
- A B-tree is a multiway tree in which each node may have dozens or hundreds of keys and children. The number of possible children is the order of the B-tree.
- There is always one more child than there are items in a B-tree, 2-3 tree, or 2-3-4 tree internal node. Leaf nodes have no child links.
- For the best performance with external memory, a B-tree is typically organized so that a node holds one block of data.
- Indexes to data stored in external memory provide fast search times and allow for multiple keys to be used in searches.
- If the search criteria involve many keys, a sequential search of all the records in a file may be the most practical approach.
- External mergesort is an efficient way to sort data stored in external memory.
- By using three arrays sized as (a small multiple of) the block size in external memory, mergesort can sort more data than can fit in internal memory.

Questions

These questions are intended as a self-test for readers. Answers may be found in Appendix C.

1. A 2-3-4 tree is so named because a node can have
   a. three children and four data items.
   b. zero, two, three, or four children.
   c. two parents, zero or three children, and four items.
   d. two parents, three items, and zero or four children.
2. A 2-3-4 tree is superior to a binary search tree in that it is _____.
3. Imagine a parent 2-3-4 node with item keys 25, 50, and 75. If one of its child nodes had items with values 60 and 70, it would be the child numbered _____.
4. True or False: Data items in a 2-3-4 tree are located exclusively in leaf nodes.
5. Which of the following is ***not*** true every time a node below the root in a 2-3-4 tree is split?
   a. Exactly one new node is created.
   b. Exactly one new data item is added to the tree.
   c. One data item moves from the split node to its parent.
   d. One data item moves from the split node to its new sibling.
6. A 2-3-4 tree increases its number of levels when _____.
7. Searching a 2-3-4 tree does ***not*** *involve*
   a. splitting nodes on the way down if necessary.
   b. picking the appropriate child to go to, based on the keys of items in a node.
   c. ending up at a leaf node if the search key is not found.
   d. examining at least one key in any node visited.
8. After a nonroot node of a 2-3-4 tree is split, which item does its parent's new right child contain, the item previously numbered 0, 1, or 2?
9. Which of the following statements about a node-splitting operation below the root of a 2-3 tree (not a 2-3-4 tree) is ***not*** true?
   a. The parent of a split node must also be split if it is full.
   b. The item with the smallest key in the node being split always stays in that node.
   c. The item being inserted at a leaf or the item promoted from a lower split must be compared with the other items of the node being split.
   d. The splitting process starts at a leaf and works upward.

10. What is the Big O efficiency of inserting and deleting an item in a 2-3 tree?
11. In accessing data on a disk drive,
    a. merging sorted records is not always possible because at least half the records must be in RAM to do so efficiently.
    b. moving data to make room to insert records is fast because so many items can be accessed at once.
    c. deleting a record is fast because it can be marked as available for use by other programs.
    d. reading two consecutive records could be 10,000 times faster than reading two random records in a large file.
12. In a B-tree for external storage, each node contains _____ data items.
13. Node splits in a B-tree are most like node splits in a _____ tree.
14. In external storage, indexing means keeping a file of
    a. keys and their corresponding blocks and records.
    b. records and their corresponding blocks.
    c. keys and their corresponding records.
    d. last names and their corresponding keys.
15. When sorting data in external storage that is too large to fit in memory, the most efficient way is
    a. creating a 2-3-4 tree for each block and traversing it in order.
    b. copying half the data in memory, sorting it, and then merging in the rest a block at a time.
    c. using block size arrays and mergesorting.
    d. quicksorting each block first and then using an insertion sort on blocks.

Experiments

Carrying out these experiments will help to provide insights into the topics covered in the chapter. No programming is involved.

9-A Draw by hand what a 2-3-4 tree looks like after inserting each of the following keys: elm, asp, oak, fig, bay, fir, gum, yew, and ash. Which item insertions cause splits? Don't use the Tree234 Visualization tool.

9-B Draw by hand what a 2-3 tree looks like after inserting the same sequence of values as in Experiment 9-A.

9-C Think about how you would remove a node from a 2-3 tree. What are all the cases that need to be handled?

Programming Projects

Writing programs to solve the Programming Projects helps to solidify your understanding of the material and demonstrates how the chapter's concepts are applied. (As noted in the Introduction, qualified instructors may obtain completed solutions to the Programming Projects on the publisher's website.)

9.1 Extend the definition of the `Tree234` class to include the following methods. Show how they operate on trees with 0, 1, and at least 10 items. Three of these methods don't need to explore all nodes, and at most only one of them should use the `traverse()` generator to be the most efficient.

a. `levels()`—Count the number of levels in the tree. An empty tree and a tree with a single node and no children have zero levels.

b. `nodes()`—Count the number of nodes in the tree. An empty tree has zero nodes. A single root node has one node.

c. `items()`—Count the number of items in the tree. An empty tree has zero items. A single root node can have one, two, or three items.

d. `minItem()`—Return the key and data of the item with the minimum key in the tree. Calling this on an empty tree should raise an exception.

e. `maxItem()`—Return the key and data of the item with the maximum key in the tree. Calling this on an empty tree should raise an exception.

9.2 Two things are needed to build an index file as shown in Figure 9-22: an assignment of records to blocks in external storage, and a sorted array of record keys. When you build a 2-3-4 tree, each of the records goes in a 2-3-4 node. You can assign to each 2-3-4 node a "block" number by writing a traversal method that returns each key and data along with a block number for the node it's in and the index number of the data within the block. Block numbers can start at 1 and increment by 1.

For the second part, write a `SortedArray` class that maintains a set of records in sorted order. The `SortedArray` constructor should take a parameter that is a function to extract the sorting key from each record. The function can be called on each record to get the sorting key. The `SortedArray` class should have

a. a `__len__()` method so that the number of records it holds can be found with the `len()` function.

b. a `get(i)` method that returns the i^{th} record in the array

c. a `find_index()` method that returns the index of a record containing a particular key, or if the key is not in the sorted array, the index where the new key would be inserted. This method should use binary search to find the index and stop when the first of multiple duplicate keys is found.

d. a `search()` method that returns a record associated with a goal key or `None` if no record has such a key.

e. an `insert()` method to add a new record to the sorted array (and should allow duplicate keys in records).

f. a `delete()` method to delete a record from the sorted array that takes a full record as a parameter so that the exact record can be deleted from the array even when there are other records with the same key present.

Use the `SortedArray` and the block traversal method to create an index of a 2-3-4 tree built using the `Tree234` class described earlier. Insert the following key-value pairs into the 2-3-4 tree and then build the sorted index using the value (a year) as the sorting key. The 2-3-4 tree uses the first element of the following tuples as the record key and the second element as the data. Your `SortedArray` should then sort them by the second element (a year).

```
("Fran", 2006), ("Amir", 1996), ("Herb", 1975), ("Ken", 1979),
("Ivan", 1988), ("Raj", 1994), ("Don", 1974), ("Ron", 2002),
("Adi", 2002), ("Len", 2002), ("Vint", 2004), ("Tim", 2016)
```

Show the contents of the 2-3-4 tree and all the records in the `SortedArray` in sorted order. Print a file index for the years in the records. Each entry in the file index has a key (year), block number (2-3-4 node number), and record index. After printing the full index, delete a couple of items with duplicate year keys and show what records remain in the `SortedArray`.

9.3 A 2-3-4 tree can be used as a sorting machine. Write a `sortarray()` function that's passed an array of values and writes them back to the array in sorted order.

This project is somewhat complicated by the issue of duplicate values because the 2-3-4 tree expects unique keys. You can handle them, however, by taking advantage of the value associated with each key in the 2-3-4 tree. By making the value be the count of the number of times the key appears in the array, the correct number of duplicate values can be placed in the output array (similar to the counting sort in Chapter 7).

The `sortarray()` function should start by creating an empty 2-3-4 tree, then loop through all the keys checking whether the key is already in the tree. If a key is already in the tree, increment its count and reinsert it in the tree with the correct count. Next, the function should traverse the tree in order to copy the keys back to the array, making any needed copies of duplicate keys. Show the input and output array contents, including some duplicate values.

An interesting "array" to sort is your source program. You can put each line of your source code into a Python array with an expression like

```
[line for line in open('my_source_code.py')]
```

A few blank lines in the source code will likely be duplicate values.

9.4 Modify the `Tree234.py` program to make a `Tree23` class so that it creates and works with 2-3 trees instead. It should display the tree and allow searches. It should also

allow items to be inserted, but only if the parent of the leaf node (which is being split) does not also need to be split. In other words, it will allow insertions only at level 0 and level 1 of the tree and only when one or the other node on the insertion path is not full. It must handle the split cases for leaf nodes shown in Figure 9-16. The next Programming Project explores how to insert items into deeper nodes. The `delete()` method is not required. If `insert()` is called on an existing key, the key's value should be updated (and the node should not be split). Show how the tree grows as you add items. Try adding 10 different items and then searching the tree for both items it contains and items it does not contain to show its performance.

If you plan to solve both this Programming Project and the next one, it helps to set up the `insert()` method as follows. The `insert()` method should call a recursive method, `__insert()`, that takes a node object as a parameter, along with the key and value to insert. The `insert()` method calls `__insert()` on the root node and looks to see if a split happened on the root. If it does, `insert()` should make a new root node to hold the promoted item with the old root node and the split node as children.

To make this work, the `__insert()` method should return three items: a key, the key's value, and a split node, all bundled as a Python tuple. When a split occurs, the key and value together are the item to promote, and the split is the new node that is the top of a subtree containing items having higher keys than the promoted item. If the key in the returned tuple is `None`, then no split occurred. The `__insert()` method is called recursively to descend the tree, but only one level down. If its node argument is `None`, that's the base case of an empty tree (no root node), so it returns a simulated split with a tuple containing the item being inserted and `None` for the split node. The `insert()` method turns that into a new root node that is also a leaf node.

If the `__insert()` method gets a valid leaf node, it should determine where the new key would go, insert it, and determine whether that causes a split, returning a tuple of three `None`s if no split is needed. If the `__insert()` method gets a valid internal node, it should check if either the internal node or the child where the insert should take place is not full and make the recursive call on the child. If the recursive call causes a split, the promoted item should be inserted in the internal node. If both nodes are full, it should raise an exception saying that kind of insertion is not allowed.

9.5 Extend the program in Programming Project 9.4 so that the `__insert()` routine is fully recursive and can handle situations with a full parent of a full child following the cases presented in Figure 9-17. This allows insertion of an unlimited number of items. The base cases of `__insert()` are the same as before, but without the check that either the current node or the child where the insertion will take place is full, the split can continue propagating up the tree in the recursive calls. As before, show how the tree grows as you add items. Try adding 10 different items and then searching the tree for both items it contains and items it does not contain to show its performance.

CHAPTER 10

AVL and Red-Black Trees

IN THIS CHAPTER

- Our Approach to the Discussion
- Balanced and Unbalanced Trees
- AVL Trees
- The Efficiency of AVL Trees
- Red-Black Trees
- Using the Red-Black Tree Visualization Tool
- Experimenting with the Visualization Tool
- Rotations in Red-Black Trees
- Inserting a New Node
- Deletion
- The Efficiency of Red-Black Trees
- 2-3-4 Trees and Red-Black Trees
- Red-Black Tree Implementation

As you learned in Chapter 8, "Binary Trees," ordinary binary search trees offer important advantages as data storage structures. They enable quick search for an item with a given key, and quick insertion or deletion of an item. Other data storage structures, such as arrays, sorted arrays, linked lists, and sorted linked lists, perform one or the other of these activities slowly. Thus, binary search trees might appear to be the ideal data storage structure.

In Chapter 9, "2-3-4 Trees and External Storage," you learned how storing multiple items per node and carefully controlling them during insertions could produce balanced trees. Maintaining balance in the trees keeps the time needed to find, insert, or delete an item consistent and quick. Unbalanced trees can be much slower by comparison. Ordinary binary search trees can become unbalanced depending on the order items are inserted.

This chapter explores ways to keep binary search trees balanced using AVL trees and red-black trees. Binary trees are simpler, in general, with a single item per node, and in some ways, easier to implement than multiway trees. They also are slightly easier to analyze because the number of items per node is a constant.

Surprisingly, there's a direct correspondence between the multiway 2-3-4 tree and binary red-black tree, as you will learn.

Our Approach to the Discussion

We're going to start with a simple approach to balancing binary trees: measure and correct. We add an extra attribute to every node to keep track of the height of the tree and keep updating it as items are inserted, moved, and removed.

From that, it will be easy to see when the tree's balance needs to be corrected. Later, we show a different approach that labels the nodes as either red or black. With just those two labels and some rules about how the labels are manipulated, it's also possible to determine when the tree is balanced or not.

The **AVL tree** is the earliest kind of balanced tree. It's named after its inventors: Adelson-Velskii and Landis. In AVL trees each node stores an additional piece of data: its height from the deepest leaf node in that subtree. This data is the easiest to understand conceptually and perhaps to implement too. The red-black tree is a bit more complicated conceptually, so we discuss how it works without showing an implementation. We also investigate its connection to the 2-3-4 tree of Chapter 9.

Balanced and Unbalanced Trees

Let's review how binary trees become unbalanced. When a group of keys is inserted in either ascending or descending order, the tree grows on only one side, like in the example shown in Figure 10-1.

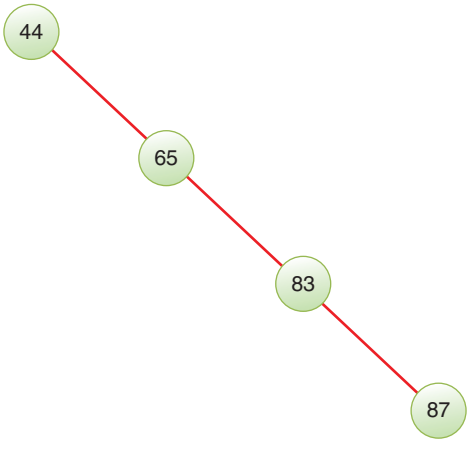


FIGURE 10-1 Unbalanced tree resulting from items inserted in ascending order

As mentioned in Chapter 8, the nodes arrange themselves in a line with no forked branches. Because each node is larger than the previously inserted one, every node is a right child, so all the nodes are on one side of the root. The tree is maximally unbalanced. If you were to insert items in descending order, every node would be the left child of its parent, and the tree would be unbalanced on the other side. When the insertion order is only partly increasing or decreasing, every run of increasing or decreasing keys can make a linear string of nodes like this.

Degenerates to O(N)

When there are no branches, the tree becomes effectively a sorted linked list. The arrangement of data is one-dimensional instead of two-dimensional. Unfortunately, as with

linked lists, you must now search through (on average) half the items to find the one with a particular key. In this **degenerate** situation, the speed of searching is reduced to O(N), compared with the O(log N) of a balanced tree. Searching through 1,000,000 items in such an unbalanced tree would require an average of 500,000 comparisons, whereas for a balanced tree it requires only 20.

Data that's only partly sorted generates trees that are only partly unbalanced. If you use the Binary Search Tree Visualization tool from Chapter 8 to attempt to generate trees with 31 nodes, some of them are more unbalanced than others. For example, the tree shown in Figure 10-2 has 14 items, even though the request was to fill it with 31 items (the maximum that can be shown). The Visualization tool prevents insertions below level 4, so the items that would have been inserted at level 5 or below were discarded.

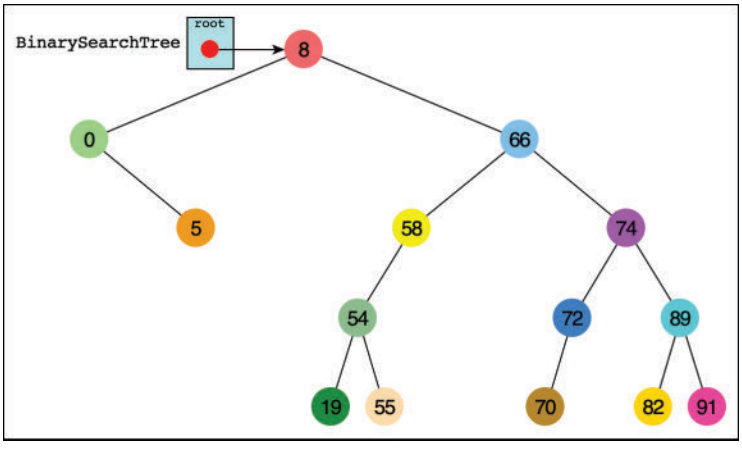


FIGURE 10-2 A partially unbalanced tree

Although not as bad as a maximally unbalanced tree, this situation is not optimal for searching. Searching partially unbalanced trees takes time somewhere between O(N) and O(log N), depending on how badly the tree is unbalanced.

Measuring Tree Balance

To know when a tree is unbalanced, we should have a way of measuring balance. With a metric, we can define some value or values as being "balanced," and everything else can be considered unbalanced.

What should we measure? As mentioned in Chapter 8, balanced trees should have an approximately equal number of nodes in their left and right sides. We could simply count the nodes on the left and the right. In Figure 10-1 the root, node 44, has zero nodes to the left and three nodes to the right. That's clearly unbalanced, but how about the root, node 49, in Figure 10-3? It has five nodes on the left and five nodes on the right. It's more balanced but still has long sequences of nodes with only a single child.

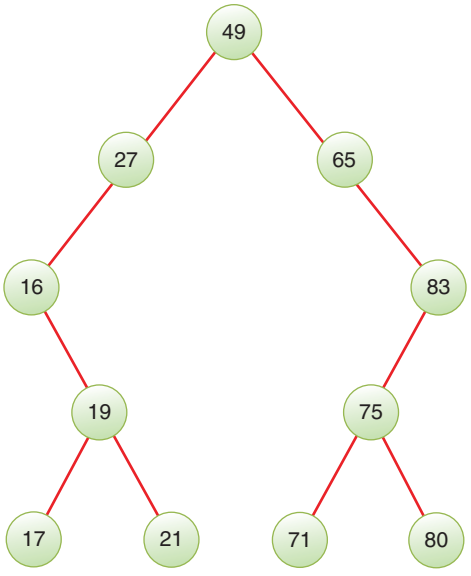


FIGURE 10-3 A symmetric binary tree

There are two difficulties with simply counting the nodes to the left and the right of the root. The first is that you're looking only at the root. If you only measure the tree at the top, you could miss problems further down. It's better to apply the metric to *every subtree*. That way, you can achieve two things: you get a measure over the entire tree, and you can measure each subtree separately. Being able to measure subtrees allows you to note things like nodes 19 and 75 in Figure 10-3 being fully balanced, whereas nodes 27 and 65 are quite unbalanced.

If you apply the node count technique to each subtree, you can certainly identify balanced and unbalanced nodes by subtracting the number of children on the right from the number on the left. Using those differences, zero means balanced, a positive value means there are more nodes on the left, and a negative value means there are more on the right.

To get a measure for the overall tree, you could try to combine the subtree measures into a single metric. If you were to add up all the left-right node count differences, you would get an overall metric, but that would still result in a value of zero for the tree in Figure 10-3 because every node with a negative value has a matching positive value on the other side. That might be an interesting measure of symmetry, but it doesn't help decide when the tree is balanced to minimize searching.

Adding up the absolute values of the differences would be one way to measure the full tree's imbalance. That would total 14 for the tree in Figure 10-3 (four each for nodes 27 and 65, three each for nodes 16 and 83, and zero for all the others because they are balanced). If you rearrange the nodes into the nearly symmetric and balanced tree shown in Figure 10-4, the overall metric goes down to 4 (one each for the nodes at level 2 with the keys 17, 21, 65, and 80).

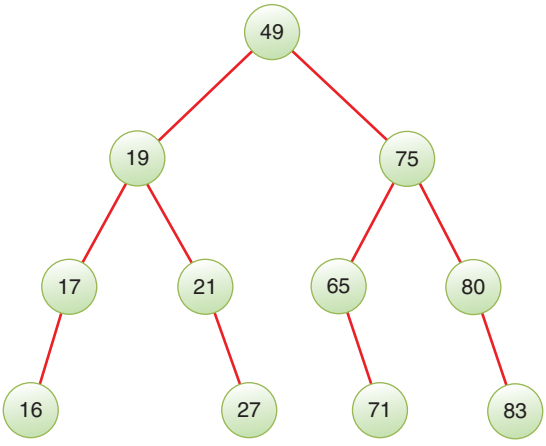


FIGURE 10-4 A nearly symmetric, balanced binary tree

That's definitely an improvement. The biggest impact to search performance, however, lies in the length of the path to the deepest leaf node. That becomes the longest path that a search must follow. In the degenerate case of insertion in ascending order of keys shown in Figure 10-1, the longest path has length N–1. Counting nodes on both subtrees sort of captures the imbalance, but summing the absolute difference in node counts produces a metric of four for the tree in Figure 10-4 where the length of the path to every leaf is the same. It would be better to measure the heights directly.

Say you'd like to measure the difference in depth of the two sides of each tree. Because you want to measure it at every subtree, it makes more sense to measure from the subtree root down to its leaf nodes. That way, you can ignore where the subtree lies within the overall tree; its position with respect to the overall tree's root doesn't change the metric.

The number of nodes on the longest path from a particular node X to a leaf node is called the **height** of the subtree rooted at node X, or more simply, the height of X. Figure 10-5 shows the distinction between the level (or depth) of a node and the height of the subtree rooted at a node. In the sample tree, each subtree's height is shown in orange next to the root node of the subtree. All the leaf nodes are at height 1 regardless of their depth. The height measures the longest path below the node (to an empty child), and the level measures the distance to the tree root.

For balance, you want to measure the difference in height between the two child branches of a node. You can subtract the right child's height from the left child's height to get the difference. When a node has only one child like nodes 17, 21, 65, and 80 do in Figure 10-4, the height of the empty child is considered zero. That way the height difference is either +1 or –1 for a node having a single leaf node as a child, like nodes 17 and 21. You want to ensure that the imbalance of those nodes is counted.

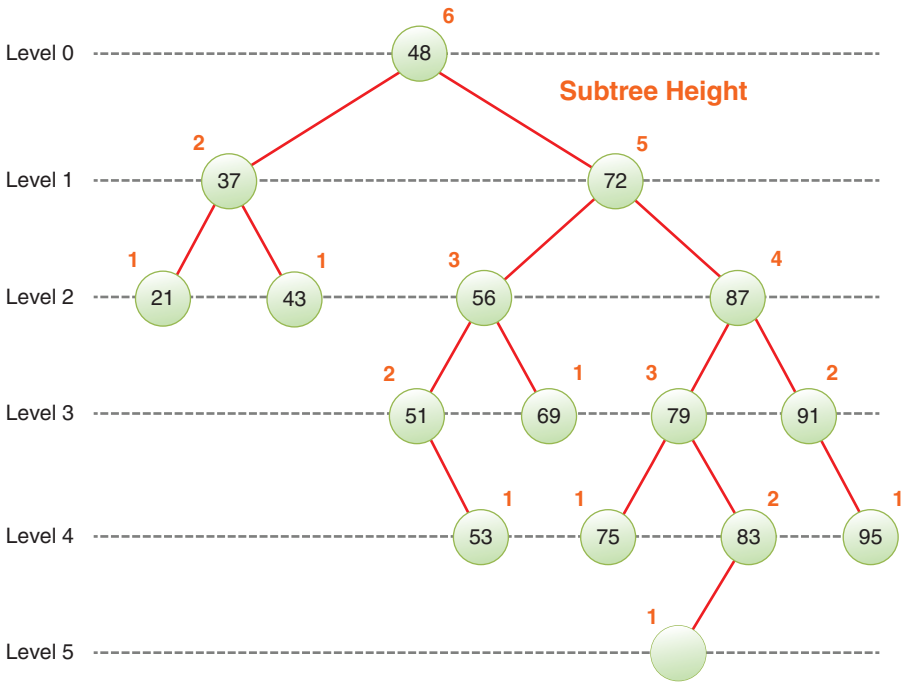


FIGURE 10-5 Node levels and subtree heights

Totaling up the absolute values of the height differences gives a different overall tree metric. For the unbalanced tree in Figure 10-3, the total is 10 (three each for nodes 27 and 65 at level 1, two each for nodes 16 and 93 at level 2, and zero for all the others). The tree in Figure 10-4 would have a total of 4 (one for each of nodes 17, 21, 65, and 80). This overall metric is the same as the total of node count differences. Both find that there are 4 nodes with imbalance of one between their left and right sides.

How Much Is Unbalanced?

With the overall metrics we've described so far, a tree with a metric of zero would certainly be balanced. If you demand that every tree with a nonzero metric be considered unbalanced, then you could only have balanced trees with node counts of 1, 3, 7, 15, …, $2^N - 1$. The reason is that the only way to be perfectly balanced is to have every node link to exactly two (or no) children and completely fill all the lowest levels of the tree (the ones nearest the root).

Defining balance to be metric = 0 is not a very practical definition because you'd like the binary trees to behave something like the 2-3-4 trees, which can contain any number of nodes and still be considered balanced. Balancing binary trees is about keeping the maximum height of any node to a minimum so that the longest search path is no longer than necessary. That's exactly the case in Figure 10-4, so that tree's metric should come out balanced. Let's see how far we need to extend the definition of balanced.

Consider the smallest binary trees. An empty tree and a tree with a single node are balanced. When a tree has two nodes, one node must be at a different height than the other. That means the root node must have at least a height difference of one. So, you must allow at least one node to have height difference of +/– 1 in a balanced tree.

When you add a third node, there are five possible tree shapes, as shown in Figure 10-6. The middle one is obviously balanced. In all the others, the root node has a height difference of two, and the midlevel node has a height difference of one.

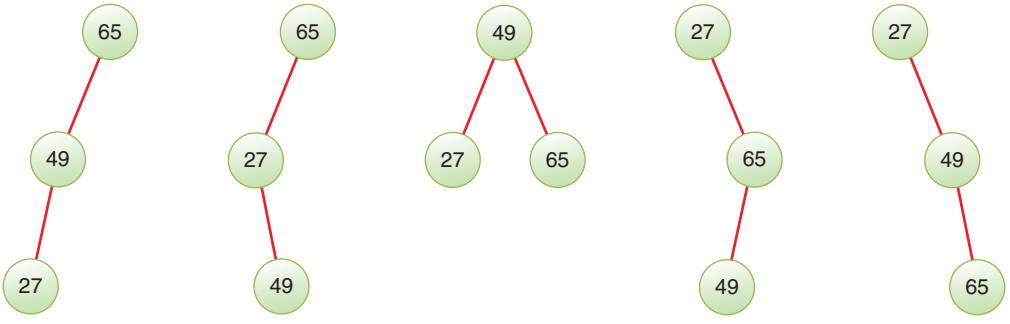


FIGURE 10-6 The possible shapes for 3-node binary search trees

Because a balanced 3-node configuration exists, you don't need to extend the definition for balanced to include anything but the middle tree in Figure 10-6. You saw that the 2-3-4 trees in Chapter 9 could be transformed into new configurations without violating any of the rules for constructing those trees. The same is true among these 3-node binary trees; they differ by simple rotations of the nodes. Ideally, the balancing algorithm could rotate the unbalanced versions into the balanced one. We come back to those rotations in a moment.

What happens when you add a fourth, fifth, sixth… node to the tree? When you add the fourth to a balanced 3-node tree, it becomes a leaf node at level 2, like the leftmost tree in Figure 10-7. Its parent has a height difference of one because it has only one child. So does the root node because it has one side with height 2 and the other with height 1. It doesn't matter which child at level 2 is the fourth node. The figure shows it being the leftmost child, but the height differences would remain for both the root and the leaf node at level 1, just with changes in sign.

As you add the fifth and sixth nodes, if they go at level 2 like the examples in Figure 10-7, the maximum absolute height difference of any node is one. The example shows keys that would make that happen, but even if a node were inserted at a lower level, there would be rotations that could transform the tree back into one of the shapes like those in Figure 10-7 as long as the node count was 4 through 6. Adding the seventh node means the tree can be put into the balanced shape on the right.

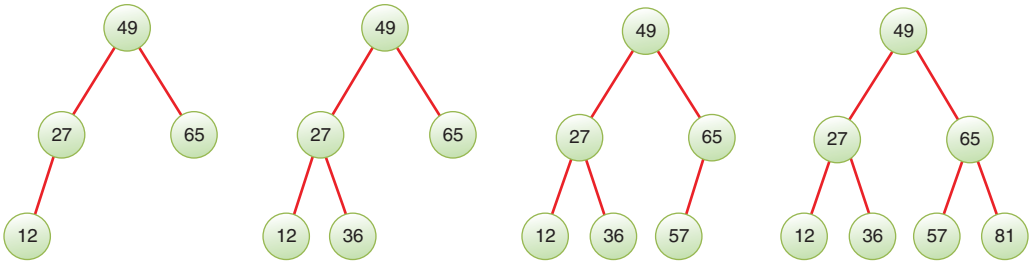


FIGURE 10-7 Balanced 4-, 5-, 6-, and 7-node binary search trees

This same pattern repeats at every level of the tree. Newly inserted nodes go at the leaf level. That either places them in a position where no node has a height difference of more than one, or it puts them at a level one lower than the lowest leaf, and rotations can raise them up to restore balance. This is the core idea of all self-balancing trees. We can now define balance as

> *A balanced, binary tree is one where all nodes have an absolute height difference of one or zero.*

AVL Trees

The AVL tree is a modified binary search tree that adds a height attribute to each node. The height differences between the left and right children of a node can be used to measure its balance. When the absolute height difference at any subtree becomes larger than one, rotations are used to correct the imbalance. Let's look first at those rotations.

You saw the five possible configurations of 3-node binary search trees in Figure 10-6. What may not have been clear is that each of those configurations is one rotation away from another. Figure 10-8 shows those configurations in a slightly different order. To transform the tree on the far left to the next tree on its right requires a rotation around the mid-level node—the one with key 27. That rotation moves node 27 down to the left and raises node 49 to the mid-level. The next transformation in the figure shows rotating around the root, node 65, in the opposite direction. That raises node 49 to the root and lowers node 65 to the right, producing the balanced tree in the middle.

The right side of Figure 10-8 shows the mirror image of the operations. Starting at the tree on the far right and rotating right around the mid-level node 65 produces the tree that's second from the right with node 49 at the mid-level. Rotating that tree's root to the left produces the balanced tree in the middle with node 49 at the root. You can also reverse the transform directions to go the other way in the figure. In other words, starting from the balanced tree in the middle, rotating right around the root, produces the tree that's second from the right in the figure.

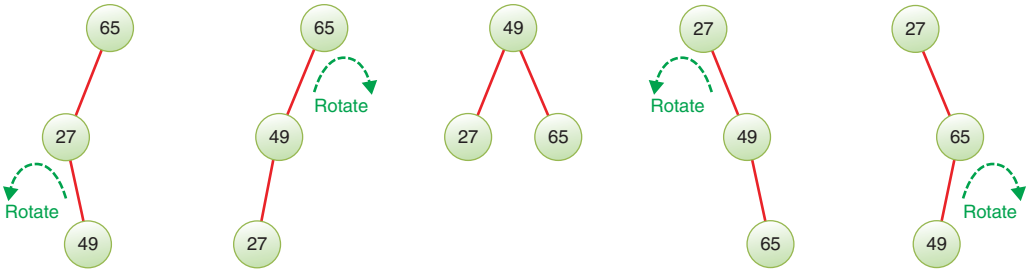


FIGURE 10-8 Rotations on 3-node binary search trees

The basic idea of the AVL tree is to use the additional height attribute in every node to determine the balance of its parent node. When an imbalance of more than one is created by an insertion or deletion, it will be corrected by using rotations.

The AVLTree Visualization Tool

Let's explore how AVL trees work using the AVLTree Visualization tool. When you launch the tool, it starts out like the Tree234 Visualization tool, except that the empty tree object at the top is labeled `AVLTree`. As with the other tree visualizations, only numeric keys from 0 to 99 are allowed. You can insert and search for individual keys using the Insert and Search buttons as before. There are also operation buttons for deleting, randomly filling, emptying, and traversing the nodes in order.

To start, try randomly filling the empty tree with 31 items. The tree that appears should look something like the one Figure 10-9. The heights appear above and to the right of each node in the tree.

The tree in Figure 10-9 is balanced; the height difference at all nodes is –1, +1, or 0. That's very different from what happened when you inserted 31 randomly chosen keys into the binary search tree of Figure 10-2. Was it just luck that it came out balanced? No, the AVL tree keeps the tree balanced with every insertion.

Try erasing the tree with the New Tree button and refilling it with 31 random nodes. The result is again a balanced tree. If you repeat the experiment many times and compare the results to the same experiment using binary search trees, you will also see that the resulting AVL trees have many more nodes, on average. Why is that?

The randomly generated AVL trees have more nodes than similarly generated binary search trees due to two factors. The first is the depth limit that the Visualization tools impose. Because insertions are not allowed at level 5 or below in either tool, some of the inserted keys are discarded. The second factor is that a balanced tree has more available nodes to fill within the depth limit. Because the AVL tree adjusts the balance on every insertion, the leaves of the tree are kept as close as possible to the root. That leaves room for later insertions. Eventually, even the AVLTree Visualization tool will run out of room at the lower levels, so it's very rare that all 31 randomly chosen values can be placed in the tree.

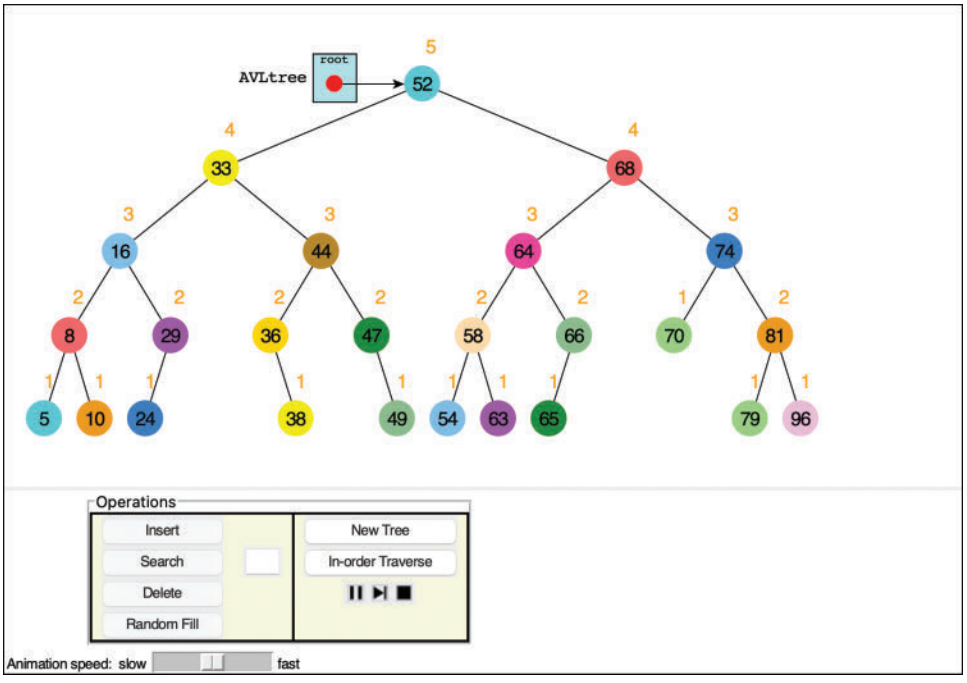


FIGURE 10-9 The AVLTree Visualization tool with a randomly filled tree

Inserting Items with the AVLTree Visualization Tool

Let's look at how the AVL tree maintains its balance. If you create a New Tree with the Visualization tool and then insert the keys 10 and 20 into it, you will see a tree like the one in Figure 10-10. The `flag = True` on the left indicates that the insert operation was successful (that is, it did not go past the depth limit and did not find an existing node with the same key). The 2 next to the root node indicates that it lies at height 2 in the tree.

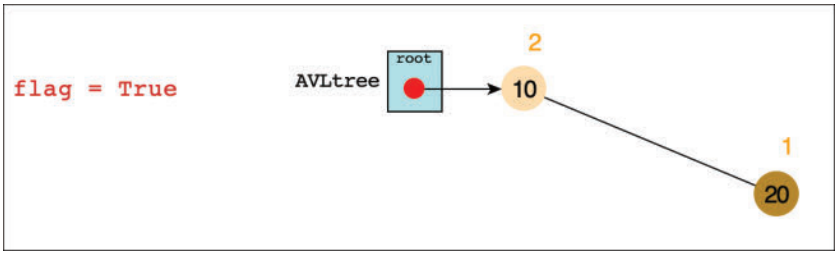


FIGURE 10-10 The first two items inserted in an AVL tree

If you try to insert a node with key 30 in the tree, it first attempts to insert the item as the right child of node 20, following the same procedure as binary search trees do. The top panel of Figure 10-11 shows the process just after completing the insertion, before updating the

height of node 20 and node 10. The procedure then returns up the path that led to the insertion point, checking the height difference of each internal node and updating its own height.

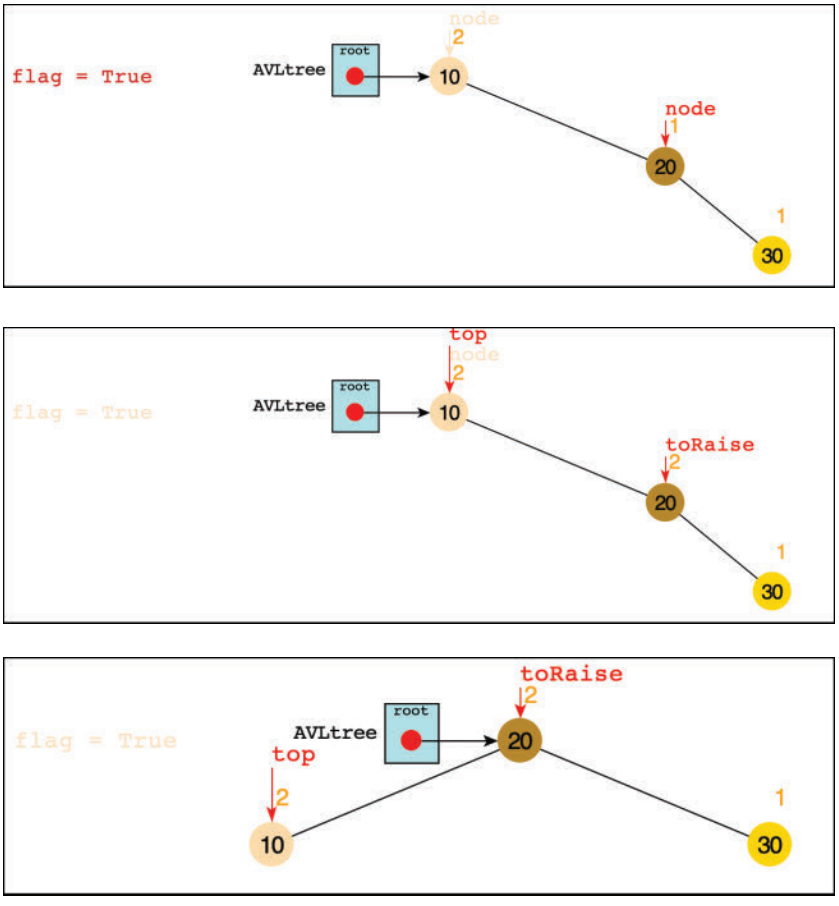


FIGURE 10-11 Steps in the insertion of node 30 into the AVL tree

In panel 2 of Figure 10-11, the process has returned to the root node (now with the `top` arrow pointing to it) and discovered that node 10 has a height difference of –2 (a height of 0 for its empty left child and 2 for its right, node 20). Note it has not yet updated the height of node 10 at this point. Finding the absolute value of the height difference to be larger than 1, it must rebalance the tree. For that, it chooses `toRaise` node 20 as indicated by the arrow.

The three nodes are rotated left, bringing node 20 to the root, as shown in the last panel of Figure 10-11. The heights of the `top` and `toRaise` nodes have been carried along in the rotation and need to be updated. Note, however, that node 30's height of 1 remains correct because height is measured relative to the leaf level.

This simplified example shows the basic operation of insertion but hides many details. In particular, this is the simplest form of rotation, where the node to raise has no children on

the side between it and the top node. Let's turn now to the code and investigate all that goes on within the structure.

Python Code for the AVL Tree

The `AVLtree` class shares much of its code with that of the `BinarySearchTree` of Chapter 8 and the `Tree234` of Chapter 9. We explain the key components that differ here and leave out some of the common implementation. You can refer to the code in the previous chapters or look at the accompanying source code for the rest of the implementation.

The `AVLtree` defines its `__Node` class as private for the same reasons as in the `BinarySearchTree` and `Tree234` classes. The main difference from the `__Node` constructor for the `BinarySearchTree` is the addition of the call to the `updateHeight()` method, which creates an instance attribute called `height`, if it doesn't exist, and determines its value from the height of the left and right child, as shown in Listing 10-1. Because AVL trees always create new nodes as leaf nodes, the constructor doesn't take a left and right parameter for the new node. The constructor could simply set `height` to 1 without calling `updateHeight()` because the new node's left and right child are always empty. We've left that call in the code to show how the calculation is made to get the initial value of 1.

LISTING 10-1 The `__Node` Class for `AVLtrees`

```
class AVLtree(object):

   class __Node(object):                # A node in an AVL tree
      def __init__(                     # Constructor takes a key-data pair
            self,                       # since every node must have 1 item
            key, data):
         self.key, self.data = key, data # Store item key & data
         self.left = self.right = None   # Empty child links
         self.updateHeight()             # Set initial height of node

      def updateHeight(self):            # Update height of node from children
         self.height = max(              # Get maximum child height using 0 for
            child.height if child else 0 # empty child links
            for child in (self.left, self.right)
         ) + 1                           # Add 1 for this node

      def heightDiff(self):              # Return difference in child heights
         left  = self.left.height  if self.left  else 0
         right = self.right.height if self.right else 0
         return left - right             # Return difference in heights
```

The `updateHeight()` method makes explicit the treatment of empty child links. It uses a Python list comprehension—`for child in (self.left, self.right)`—to examine both

children. For links that are `None`, the child height is treated as 0, otherwise it gets the height from the `child`'s `height` attribute. The child heights are passed to the `max()` function as arguments. Finally, it adds 1 to the maximum height of the children, so that leaf nodes get a height of 1.

The decision metric, `heightDiff()`, comes next. The height of a node's right subtree is subtracted from the height of its left subtree. Because those subtrees might be empty, it substitutes a default value of zero for the child's height if the child link is `None`. Thus, a node with only one child produces a height difference that is plus or minus the height of the child.

Inserting into AVL Trees

Inserting new items into an AVL tree starts off like insertion in binary search trees; the key of the item to insert is compared with the existing keys in the tree until the leaf node is found where it should be inserted. In the `BinarySearchTree` class, insertion needed to alter only one child link, in the parent of the new node, to do its work. Because AVL trees need to check the balance after each insertion and possibly make rotations all the way up to the root, it's going to need to follow the parent links back up the tree after inserting at the leaf level. Returning up the path that was followed is easy with a recursive implementation, so that's what's shown in Listing 10-2. Keeping an explicit stack of nodes visited on the path could also do the same thing.

The `insert()` method is simple. It calls the recursive, private `__insert()` method starting at the root node to do the work. Note that it sets the `__root` attribute to the result of that call. This is how the recursive algorithm updates the tree; it calls a method operating on a subtree and then replaces the subtree with whatever modifications occurred. In this way, for instance, the empty root attribute gets filled with a newly created node on the first insertion. The `insert()` method returns a Boolean `flag` indicating whether a new node was inserted in the tree (as opposed to updating an existing key). The `flag` is the second value returned by `__insert()`.

Inside the `__insert()` method, the first checks are for base case conditions. If `__insert()` was called on an empty tree (or subtree), the `node` parameter is `None`, and it simply returns a new node holding the key and data of the item to be inserted. The return of that new object handles inserting the first node of the tree. It also returns the insertion flag as `True` to tell the caller that another node was added.

If the key of the node to insert matches an existing key in the tree, the `__insert()` method faces the choice of what to do with duplicate keys. Like the 2-3-4 tree, this implementation updates the existing node's `data` attribute with the new data to insert. That means that duplicate keys are not allowed, and the AVL tree behaves like an associative array. The caller can know this by seeing the returned insertion flag is `False`. After the `data` attribute is changed, the modified `node` must be returned so its parent can store the same node back in the root or other child link.

LISTING 10-2 The `AVLtree insert()` Method

```
class AVLtree(object):
…
   def insert(self, key, data): # Insert an item into the AVL tree
      self.__root, flag = self.__insert( # Reset the root to be the
         self.__root, key, data) # modified tree and return the
      return flag             # the insert vs. update flag

   def __insert(self,         # Insert an item into an AVL subtree
                node,                      # rooted a particular node, returning
                key, data):                # the modified node & insertion flag
      if node is None:                     # For an empty subtree, return a new
         return self.__Node(key, data), True # node in the tree

      if key == node.key:                  # If node already has the insert key,
         node.data = data                  # then update it with the new data
         return node, False                # Return the node and False for flag

      elif key < node.key:                 # Does the key belong in left subtree?
         node.left, flag = self.__insert(  # If so, insert on left and
            node.left, key, data)          # update the left link
         if node.heightDiff() > 1:         # If insert made node left heavy

            if node.left.key < key:        # If inside grandchild inserted,
               node.left = self.rotateLeft( # then raise grandchild
                  node.left)

            node = self.rotateRight(       # Correct left heavy tree by
               node)                       # rotating right around this node

      else:                              # Otherwise key belongs in right subtree
         node.right, flag = self.__insert(  # Insert it on right and
            node.right, key, data)         # update the right link
         if node.heightDiff() < -1:        # If insert made node right heavy

            if key < node.right.key:       # If inside grandchild inserted,
               node.right = self.rotateRight( # then raise grandchild
                  node.right)

            node = self.rotateLeft(        # Correct right heavy tree by
               node)                       # rotating left around this node

      node.updateHeight()                  # Update this node's height
      return node, flag                    # Return the updated node & insert flag
```

After the base cases are handled, what remains are the recursive ones. (In the AVLTree Visualization tool, one other "base" case is checked: does the insertion go past the depth limit.) There are two options; the item to insert belongs in either the left or right subtree of the node. If the `key` to insert is less than this node's key, it belongs in the left side, and otherwise the right. These two choices are handled in the next two blocks of code.

For the left side, the first step is to update the `left` link with the result of recursively inserting the item in the left subtree (and record the insertion `flag` result). For programmers just getting used to recursion, that may seem like a silly thing to do because it looks like setting a variable to itself. As you saw in Chapter 6, "Recursion," it can be quite powerful. You've already seen that if the `node.left` link were `None`, `__insert()` would return a new leaf node to replace it, and that's exactly what's needed. If that left link points to some subtree, you can assume it's going to return the node on top of that subtree after any insertions and rotations that might happen inside it to maintain the subtree's balance. Having made that assumption, all that remains is completing the work by updating the link after the balanced subtree is returned. Of course, you have to return the revised subtree from this call to `__insert()` to ensure the assumption remains accurate.

What else must be done after the left subtree is updated? Well, it's possible that the insertion on the left caused the balance of this subtree to become **left heavy**; that is, the left subtree has a height that is now greater than that of the right subtree by more than one. It cannot have made this node **right heavy** because the insertion was made in the left subtree and you started off with a balanced subtree (because AVL trees are designed to always maintain self-balance). Another way of saying that is the height difference of `node` must have been either –1 , 0, or +1 when the `__insert()` method was called.

The `__insert()` method checks the node's balance by seeing whether `node.heightDiff()` now exceeds 1. As shown in Listing 10-1, `heightDiff()` computes the difference of the (newly modified) height of `node`'s left subtree with that of `node`'s right subtree. If the left node's height was updated properly in the recursive call, this will work. If you look ahead in the code, that update happens right before the `__insert()` method returns the modified node.

When `node` is left heavy after the insertion in its left child, the `__insert()` method needs to correct it by performing one or two rotations. To decide which ones are needed, the method looks to see where the key was inserted relative to the node. Back in Figure 10-8, you saw how to rotate 3-node trees until they were balanced. The same logic applies here, under the assumption that the item just inserted was the third node.

When the insertion goes in the left child of `node`, then the shape of the tree is either the leftmost tree in Figure 10-8 or the second to leftmost. The insertion must have produced a grandchild; otherwise, it couldn't have made `node` left heavy. The question then becomes, Was the item inserted in the *inside* or *outside grandchild* of the `node`? The leftmost tree of Figure 10-8 shows an insertion at the *inside grandchild* of the top node. That requires two rotations to get the balanced form in the middle tree of Figure 10-8. If the insertion was to the *outside grandchild*, the situation requires only one rotation.

The `__insert()` code in Listing 10-2 checks whether the `key` that was inserted is larger than `node`'s left child key. If it is, then the insertion was an inside grandchild because the

insertion path went left and then right on the way down. To correct the imbalance, it performs a left rotation around `node`'s left child. Otherwise, it performs only one rotation—a right rotation around the top of the subtree, `node`. That's what's needed if the insertion went to the outside grandchild, going left and left again on the way down. The right rotation corrects the left heaviness of the tree. The rotation operations are performed by methods we describe in a moment. They use the same style as the recursive `__insert()` method and set the node at the top of the rotation to whatever node is moved up.

The preceding steps handle the rotations for insertions in `node`'s left subtree. The next `else` clause handles the rotations for insertions in `node`'s right subtree. These are the mirror image of the steps in the left-hand rotations, swapping right and left and changing the sign of the height difference. The inside grandchild is the one following a right and then a left child link, and the outside follows a right-right link.

After either kind of subtree insertion, the `__insert()` method calls `updateHeight()` on the `node`, followed by returning the modified node and the insertion `flag` from the recursive call. The height could be different than the node's height on entry to the `__insert()` method because there is one more item in a subtree below. Updating the height at this point ensures that the node returned to the caller has proper information about its position relative to the leaves of the subtree.

Rotations are performed by changing the links between the nodes in the subtree. The `top` of the subtree is sometimes called the center of rotation or the pivot (although that latter term might be confused with the pivot used in quicksort). For `rotateRight()`, the `top`'s left child is raised up. The code of Listing 10-3 shows `top.left` being stored in the variable `toRaise`.

LISTING 10-3 The `rotateRight()` and `rotateLeft()` Methods of `AVLtree`

```
class AVLtree(object):
…
   def rotateRight(self, top): # Rotate a subtree to the right
      toRaise = top.left       # The node to raise is top's left child
      top.left = toRaise.right # The raised node's right crosses over
      toRaise.right = top      # to be the left subtree under the old
      top.updateHeight()       # top. Then the heights must be updated
      toRaise.updateHeight()
      return toRaise           # Return raised node to update parent

   def rotateLeft(self, top):  # Rotate a subtree to the left
      toRaise = top.right      # The node to raise is top's right child
      top.right = toRaise.left # The raised node's left crosses over
      toRaise.left = top       # to be the right subtree under the old
      top.updateHeight()       # top. Then the heights must be updated
      toRaise.updateHeight()
      return toRaise           # Return raised node to update parent
```

The next step is perhaps the most confusing. This is where the node being raised has its right child moved over to become `top`'s left child. The moving child is the **crossover subtree**. The right child is empty in the 3-node tree second from the left in Figure 10-8, so it appears to be just a way to initialize the left child of `top` to be empty in its new position. We discuss what happens when the crossover contains a subtree in a moment.

After the crossover link is moved, `top` becomes the right child of the node `toRaise`, completing the restructuring of the nodes. The heights of both changed nodes need to be updated because their relative positions changed. You change the height of `top` first because it moved lower in the tree, and the height of `toRaise` now depends on it. When both heights are updated, then the new top of the subtree, `toRaise`, is returned to be stored in the parent's link to the rotated subtree.

The code for `rotateLeft()` in Listing 10-3 is another mirror image of the code for `rotateRight()`, swapping the roles of left and right.

The Crossover Subtree in Rotations

What happens during rotations of more complex trees when rotations bubble up the tree after inserts at lower levels? There could be child links all around the nodes being rotated. Where do they go? To answer that question, let's look at a right rotation in the middle of a tree.

When you rotate right about some node, you are trying to raise its left child up. Let's call the node at the top, T, and the node to raise, R. In general, three subtrees appear below T and R, as shown in Figure 10-12. R's left child has all the nodes with keys less than R's key, and R's right child has all the keys greater than R's key. More specifically, it has all the keys between R's key and T's key. The reason is that you had to follow the left child link of T when inserting those keys. The third subtree that's involved is T's right child, which has nodes with keys greater than T's key.

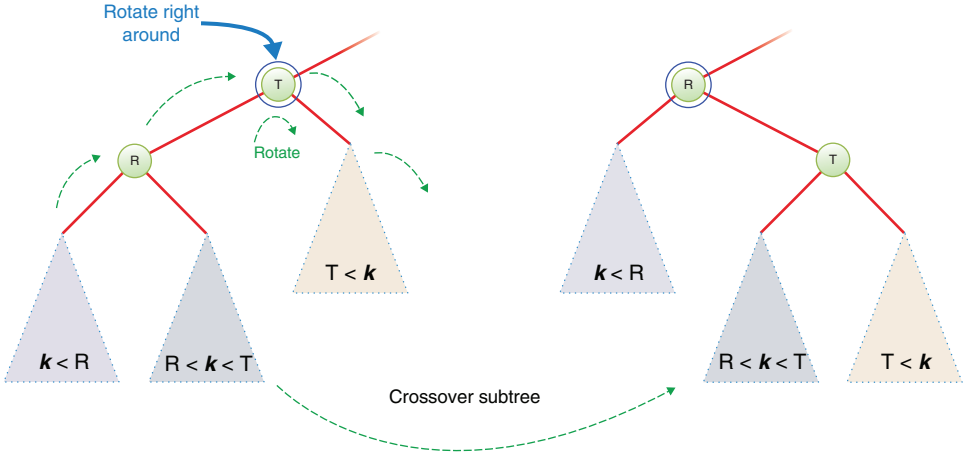


FIGURE 10-12 Right rotation and the crossover subtree

The left side of Figure 10-12 shows the relationships of T, R, and its three subtrees before the rotation. T sits at the top with R as its left child. T may or may not have a parent node, indicated by the fading line leading upward. Hanging below T and R are the three subtrees, containing the keys less than R (k < R), the keys between R and T (R < k < T), and the keys greater than T (T < k). The green arrows show the planned movement direction of the nodes. R moves up and to the right, while T moves down and to the right.

The outer two subtrees are obviously still in the correct position after the rotation produces the configuration on the right of the figure. What about the crossover subtree? It holds the keys higher than R's and less than T's key. Because it now is the left child of T, any search that gets to T will correctly choose the left link to find the keys it contains. Because T became the right child of R, any search for a key larger than R's will correctly choose its right link to T, and hence reach the crossover subtree.

The right rotation preserves the items in the correct subtrees relative to the keys being rotated. Also, the heights of the subtrees do not change during the rotation, so the call to `updateHeight()` in `rotateRight()` only needs to check the height of the topmost node of each subtree to correctly update the heights of T and R (`top` and `toRaise`). The `rotateLeft()` method performs the mirror image set of operations, preserving the subtree relationships.

The visualization tool animates the movements of the various subtrees during rotations. Figure 10-11 showed the simple example of a left rotation with an empty crossover subtree. Try creating situations where subtrees cross over from left to right (or vice versa). A simple example happens when you insert the sequence of keys: 10, 20, 30, 40, 50, 60. After node 60 is inserted, use the stepping capabilities to watch the updates to `top` and `toRaise` in the `rotateLeft` method when it is called to rotate the root node. This shows how the crossover subtree (rooted at node 30) gets put in place. Another example that shows the double rotation of an inside grandchild is to add keys 25 and 27 to the six just inserted. Then add keys 29, 28, and 22 to see a 3-node crossover subtree movement. Watch these animations several times until you can easily anticipate what will happen before the nodes are moved.

Deleting from AVL Trees

Deletion from an AVL tree is somewhat more complex and follows the strategy used for the binary search trees described in Chapter 8. Deleting leaf nodes is easy, although you must check the tree's balance afterward. Deleting an internal node is not much harder if it has a single child link; the child replaces the node in the tree. Only deletion of internal nodes with two child links poses significant challenges. Even then, you know that you can replace the item stored at that node with that of its successor, which always lies at one of the simpler deletion positions.

As was the case for insertions, each deletion in a subtree could cause the height to change enough that the tree is no longer balanced. Correcting the imbalance will involve rotations—in fact, the same kinds of rotations used for insertions. The implementation can use the same kind of recursive method to find the node to delete and then to hunt for the successor, updating the child links and heights after each recursive call completes.

The main `delete()` method shown in Listing 10-4 serves the same role as the `insert()` method; it calls the private recursive `__delete()` method on the root node, updates the root with the returned value, and returns a `flag` indicating whether the goal node was found and deleted.

LISTING 10-4 The `delete()` method of `AVLtree`

```
class AVLtree(object):
…
   def delete(self, goal):                  # Delete a node whose key matches goal
      self.__root, flag = self.__delete(    # Delete starting at root and
         self.__root, goal)                 # update root link
      return flag                          # Return flag indicating goal node found

   def __delete(self,                       # Delete matching goal key from subtree
                node, goal):                # rooted at node. Return modified node
      if node is None:                      # If subtree is empty,
         return None, False                 # then no matching goal key

      if goal < node.key:                   # Is node to delete in left subtree?
         node.left, flag = self.__delete(   # If so, delete from left
            node.left, goal)                # update the left link and store flag
         node = self.__balanceLeft(node)    # Correct any imbalance

      elif goal > node.key:                 # Is node to delete in right subtree?
         node.right, flag = self.__delete(  # If so, delete from right
            node.right, goal)               # update the right link and store flag
         node = self.__balanceRight(node)   # Correct any imbalance

      # Else node's key matches goal, so determine deletion case
      elif node.left is None:               # If no left child, return right child
         return node.right, True            # as remainder, flagging deletion
      elif node.right is None:              # If no right child, return left child
         return node.left, True             # as remainder, flagging deletion
      # Deleted node has two children so find successor in right
      else:                                 # subtree and replace this item
         node.key, node.data, node.right= self.__deleteMin(node.right)
         node = self.__balanceRight(node)   # Correct any imbalance
         flag = True                        # The goal was found and deleted

      node.updateHeight()                   # Update height of node after deletion
      return node, flag                     # Return modified node and delete flag
```

The `__delete()` method works on a subtree rooted at a `node` in the tree. First it checks the base cases. If `node` is `None`, the subtree is empty, so nothing can be deleted. The empty subtree is returned along with `False` to indicate no deletion occurred.

For subtrees with some items, the next checks determine where the `goal` lies relative to the root of the subtree, `node`. If the `goal` key is less than that of `node`, the item to delete must be in its left subtree. It updates that left subtree with the result of the recursive call to `__delete()` on `node.left`. Then it rebalances the node knowing that the left side has been reduced by calling `balanceLeft()`. Similarly, if the `goal` key is greater than that of `node`, the item to delete must be in its right subtree. It performs the delete on the right and rebalances accordingly. For both branches, it stores the `flag` indicating whether a deletion actually happened.

If neither of those `if` statement conditions applied, then `__delete()` knows the `goal` key matches the `node`'s key. Now it can look to see what deletion case it has. If the `node` has no left child, then deleting this node is simply a matter of promoting its right child to replace it in the tree. If both the left and right child links are empty, then `node` is a leaf, and returning `node.right`, which is `None`, will prune it from its parent. The next `elif` statement checks if the right child link is empty. If so, it simply promotes the left child to replace the node.

After all the simple deletion cases have been checked, you know you are deleting an item from a node that has two subtrees. You must find the successor for this node, store its item in this node's key and data attributes, delete the successor from the tree, correct any balance issues that causes, update this node's height, and return the modified node. That's quite a bit of work to do, and it's broken out into a couple of helper methods.

The `__delete()` method calls the `__deleteMin()` method to delete the item with the minimum key in the node's right subtree. That minimum is the successor to this node, as you learned in Chapter 8. Then it calls the `__balanceRight()` method to correct any imbalance caused by removing the successor from the right subtree. It also sets `flag` to `True` indicating a deletion happened.

After any of the different types of deletion are performed, the node's height could have changed, so it calls `node.updateHeight()`. Finally, it returns the modified `node` and the deletion `flag`. We look at each of those new methods to see how they do their work.

The `__deleteMin()` method returns three values using a Python tuple. The first two are the key and data of the successor node. They must replace the item being deleted in the node that `__delete()` found. The third returned value is the modified node for the right child of the node that `__delete()` found. It's possible that the right child is a leaf node (making it the successor), and that would mean returning `None` for the third value.

The `__deleteMin()` method shown in Listing 10-5 operates like `__delete()` in that it recursively descends the subtree to find and delete an item. The item to find is not known by a specific key; it's the item with the smallest key, so the recursion always follows the left link until a node with an empty left child is found. That's the base case, which is tested first. After that minimum node is found, its key and data can be returned as the successor. It returns the right child of the successor, which might be empty, as the subtree

to replace the minimum node. After that's returned to its caller, the minimum node has been removed from the tree. (There's no need for a recursive call to `__delete()` here like there was for deleting the successor in binary search trees because the assignment to `node.left` from the result of the recursive call eliminates the `node` from the tree.)

LISTING 10-5 The `__deleteMin()` and Rebalance Methods of `AVLtree`

```
class AVLtree(object):
…
   def __deleteMin(                    # Find minimum node of subtree, delete
         self,                         # it, return minimum key, data pair and
         node):                        # updated link to parent
      if node.left is None:            # If left child link is empty, then
         return (node.key, node.data, # this node is minimum and its
                 node.right)           # right subtree, if any, replaces it
      key, data, node.left = self.__deleteMin( # Else, delete minimum
         node.left)                    # from left subtree
      node = self.__balanceLeft(node)  # Correct any imbalance
      node.updateHeight()              # Update height of node
      return (key, data, node)

   def __balanceLeft(self, node):      # Rebalance after left deletion
      if node.heightDiff() < -1:       # If node is right heavy, then
         if node.right.heightDiff() > 0: # If the right child is left
            node.right = self.rotateRight( # heavy, then rotate
               node.right)             # it to the right first

         node = self.rotateLeft(       # Correct right heavy tree by
            node)                      # rotating left around this node
      return node                      # Return top node

   def __balanceRight(self, node):     # Rebalance after right deletion

      if node.heightDiff() > 1:        # If node is left heavy, then
         if node.left.heightDiff() < 0: # If the left child is right
            node.left = self.rotateLeft( # heavy, then rotate
               node.left)              # it to the left first

         node = self.rotateRight(      # Correct left heavy tree by
            node)                      # rotating right around this node
      return node                      # Return top node
```

The rest of the `__deleteMin()` method handles the recursive work of descending the left subtree, storing the successor key and data that were found, adjusting any balance problems the lower deletion in the left subtree causes, and updating the height of `node` after the deletion and possible rotations. These actions are like those for insertions, although

the call to the `__balanceLeft()` method is different. Before we describe how that works, let's first look carefully at which subtrees get rebalanced by `__deleteMin()`.

Because the call to `__balanceLeft()` occurs immediately after every recursive descent down the left child links, it is applied to every node visited that has a left child. The successor node, by definition, does not have a left child. Could not calling `__balanceLeft()` on the successor cause a problem?

To answer that question, think about two possibilities for the successor node: a leaf node, or a node with a right subtree and no left subtree. In the first case, the leaf node is being deleted, so there really isn't anything that could be balanced. In the second case, the successor is replaced by its right subtree. The right subtree, call it S, must have been balanced before the call to `__delete()` because AVL trees preserve balance for insertions and deletions. The balance measure of S depends solely on the height of S's subtrees, which don't depend on their position in the overall tree. So, you can safely skip rebalancing the successor node after updating it with an already-balanced subtree rooted at S. The successor's parent node, however, may need rebalancing because S will have a height that is one less than the successor that is being moved. The caller will take care of balancing the parent of the successor.

The `__balanceLeft()` method might not be called in one other case. That happens when the first call to `__deleteMin()` lands on a node with no left child. The `node` in that case is the right child of the node that matched the goal key (which will be updated with the successor key shortly). With `__deleteMin()` immediately returning its `node`'s item and right subtree in its base case, no call to the `__balanceLeft()` method occurs. Using the same reasoning as earlier, however, you know that the right subtree being returned must have been balanced before the call to `__delete()`. The already-balanced subtree is being promoted up to replace the `node` in the call to `__deleteMin()` and doesn't need rebalancing. When `__deleteMin()` returns, the `__delete()` method calls `__balanceRight()` to rebalance the shortened right subtree.

Rebalancing all the nodes that need it is a little tricky. The recursive structure in the delete routines follows every link and applies the `__balanceLeft()` or `__balanceRight()` method at every level. That should give you confidence that you can't have missed any nodes along the way.

Now it's time to look at the `__balanceLeft()` and `__balanceRight()` methods themselves in Listing 10-5. They use the `heightDiff()` method to determine if the subtree rooted at the node is right heavy or left heavy, respectively. If the subtree is not heavy on one side, it does nothing and returns the unmodified `node` (and hence its subtree).

In `__balanceRight()` when the subtree to rebalance is left heavy, it's almost the same situation as when the `__insert()` method had to deal with adding a new item in the left subtree. This time, however, you don't have an insert key to know where the extra height was added (because a node was deleted from the right subtree). Instead, you need to look at the balance—the height difference—of the left subtree, the one that needs to be raised to restore balance. If that left subtree is perfectly balanced or has a left height one greater than its right (a height difference of 0 or +1), then you will need only one right rotation to bring it up to the top. Note that the left subtree's height difference can take on only

three possible values: –1, 0, or +1. Any other value would mean the subtree was unbalanced, and you've balanced every subtree before passing it up the recursive chain.

What happens when the left subtree of the node to rebalance in `__balanceRight()` has a height difference of –1? That difference indicates that its right subtree's height is one more than its left. That situation is shown on the left of Figure 10-13. The `__balanceRight()` method was called on node T and found that it is left heavy due to the deletion in its right subtree. That means it found a height difference of +2 at T. Furthermore, T's left subtree, rooted at node R, has a height difference of –1. That means the right subtree has a height that is one more than that of the left. In the figure they are shown with heights of H and H + 1. It's that right side with height H + 1 that is going to cross over when the `__balanceRight()` method rotates T to the right.

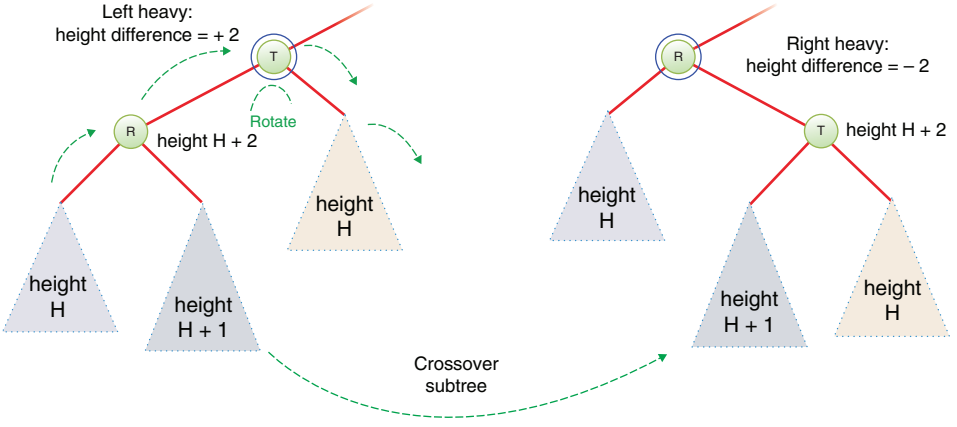


FIGURE 10-13 Balancing a left-heavy subtree

In this situation, the right subtree of T must have a height of H. You know that because the height difference at T is exactly +2, and R's height is H + 2. If you execute a right rotation around T, you will get the situation on the right of Figure 10-13. The right subtree of R with height H + 1 crosses over to become the left subtree of T. The height of T would then be H + 2 because it is one more than the height of its maximum child. Comparing the height difference at R, now the top node after the right rotation, you find –2, meaning it is a right-heavy tree. That's a problem, but you can solve it with another rotation.

The solution is to use the same operation used when handling the inside grandchild after an insertion on the left subtree (see Figure 10-8). If you first use a left rotation on the left subtree, R, that will reduce the height of the crossover subtree to H. By raising the inside grandchild of T up one (rotating left around R), you have evened up the left side prior to raising the left child of T up to be the new top of the tree (in the rotation right around T).

Thus, the `__balanceRight()` method performs one or two rotations to correct an unbalanced subtree. It checks the height differences at the root of the subtree and at its left child to determine which rotations are needed. The `__balanceLeft()` method performs

the mirror image operations to correct a left deletion that caused a right-heavy subtree. Note that it does not need to recursively descend into the subtrees. The reason is that you've ensured they all have an absolute height difference of one or less in previous rebalancing operations.

The visualization tool shows how all these methods operate. Try deleting nodes from a tree: first a few leaves, then some internal nodes with one and two children to see all the cases. You can click a node with your pointer device to enter its key in the text entry box or type the key number before selecting the Delete button. After several deletions on one side, the AVL tree will eventually perform a rotation using the `__balanceLeft()` or `__balanceRight()` method. Use the stepping controls to carefully follow any of the calls that cause confusion.

We've now covered insertion and deletion in AVL trees. The remaining methods of AVL trees such as `search()` and `traverse()` are the same as for binary search trees, as you can see if you look carefully at the visualization tool's code. The added height attribute of nodes might not be exposed to callers because it is only used to maintain the balance of the tree, but it can be a useful metric for some applications.

The Efficiency of AVL Trees

Because AVL trees are similar to binary search trees, their efficiency is similar. There is the added storage cost of keeping the height with each node and the added time it takes to update that attribute and rebalance subtrees during insertions and deletions. That extra work produces balanced, binary trees, which you know have a search time of O(log N). Best of all, the search time doesn't degrade to O(N) for degenerate cases.

How much do you have to pay to get the benefits of balance? On the insertion, there are the calls to `updateHeight()` at every level of recursion. There are $\log_2(N + 1)$ levels in the tree and the same number of recursion levels. There are also calls to `heightDiff()` at every level to check the tree's balance. Some of those lead to more key comparisons and rotations. In the worst case, every level would call `heightDiff()` once and make two rotations (which call `updateHeight()` twice each). That's quite a bit, but it's still a fixed amount of work that doesn't depend on the number of nodes in the tree. All the added manipulations increase the amount of work done per recursive level by a constant amount. In Big O notation, you can ignore the constant amount and conclude that insertion is O(log N) because the only thing that grows with N is the number of levels.

The analysis of deletion follows that of insertion. There are still $\log_2(N + 1)$ levels to visit, and each one involves calls to `updateHeight()` and `heightDiff()` inside of `__balanceLeft()` or `__balanceRight()`, and one or two rotations when imbalances are found. Those calls all add to the constant that multiplies the number of levels, and that constant doesn't change as the number of items grows larger. That leaves deletion as an O(log N) operation.

Thinking about the amount of memory used, it is O(N) because you need a fixed amount of memory for every node, and there is exactly one node for every item stored. The

constant that multiplies the number of items, however, has grown because you added the height attribute to the tree. Somewhat harder to see is that the height attribute needs to be big enough to accurately count the height of each subtree. If you used a single byte for the height, then it could only count up to a height of 255. That limitation could become a problem for a huge number of items. A full machine word could be used to store the height, which would be 64 (or possibly 32) bits, allowing for extremely deep trees.

There's also the O(log N) memory that will be used for the recursive stack because there is at least one call per level of the tree. There are, of course, many other procedure calls, but the total number that are active at any one time will be proportional to the number of levels. Because the O(log N) memory will be much smaller than the O(N) memory needed to store the nodes, it is usually ignored.

Red-Black Trees

Are there other ways to balance binary trees? Yes, of course there are, but are there any that are just as efficient but don't add as much extra work as the AVL tree does? That's a bigger challenge, but a structure called the **red-black tree** provides an interesting answer. Instead of relying on a measure of a node's height, all the red-black tree requires is 1 bit more of information about each node. The bit encodes whether the node is labeled as red or black. How can a single bit help balance the tree? Let's find out.

Red-black trees are not trivial to understand. Because of this, the actual code is more lengthy and complex than you might expect. It's therefore hard to learn about the algorithm by examining code. You've seen many of the implementations of key operations like rotations and determining inside and outside grandchildren in the binary search tree and AVL tree. So, we don't describe the red-black tree source code in this text. Instead, we discuss the structure and algorithms, and help you understand the operations using another visualization tool.

Conceptual

For a conceptual understanding of red-black trees, we are aided by the RedBlackTree Visualization tool. We describe how you can work in partnership with the tool to insert new nodes into a tree. Including a human into the insertion routine certainly slows it down but also makes it easier for the human to understand how the process works.

Searching works the same way in a red-black tree as it does in an AVL or ordinary binary search tree. As you might expect, insertion and deletion are where the differences emerge. Accordingly, in this chapter we concentrate on the insertion process.

Top-Down Insertion

Red-black trees use *top-down* insertion. This means that some structural changes may be made to the tree as the search routine descends the tree looking for the place to insert the node. This approach was used in the 2-3-4 trees in Chapter 9 when full nodes were split as the algorithm searched for an insertion point.

Bottom-Up Insertion

Another approach is *bottom-up* insertion. This type involves finding the place to insert the node and then working back up through the tree making structural changes. Bottom-up insertion is less efficient because two passes must be made through the levels of the tree. The 2-3 tree in Chapter 9 used a bottom-up approach, splitting full leaf nodes and promoting the overflow up toward the top (root). The AVL tree also rebalances the tree in a bottom-up fashion. In a red-black tree, balance is maintained during insertion and deletion, as was done for all the self-balancing trees. When an item is inserted, the insertion routine checks that certain characteristics of the tree are not violated. If they are, it takes corrective action, restructuring the tree as necessary. Because the routine maintains these characteristics, the tree is kept balanced.

Red-Black Tree Characteristics

In a red-black tree, every node is marked as either black or red. These are arbitrary colors; blue and yellow would do just as well. In fact, the whole concept of saying that nodes have colors is somewhat arbitrary. Some other analogy could have been used instead. You could say that every node is either heavy or light, or yin or yang. Colors, however, are convenient labels and help programmers use a consistent vocabulary. A Boolean data attribute, such as `isRed`, is added to the node class to embody this color information.

In the RedBlackTree Visualization tool, the red-black characteristic of a node is shown by its border color. The center color, as it is in all the tree visualization tools, is simply an indication of the data attribute of the node. When we speak of a node's color in this chapter, we are almost always referring to the solid red or black border color shown in the figures.

Red-Black Rules

When inserting (or deleting) a new node, you check for certain conditions, which are called the **red-black rules**. If they're all followed, the tree is called **red-black correct**, and it will be balanced.

Rule 1. Every node is either red or black.

Rule 2. The root is always black.

Rule 3. If a node is red, its child nodes must be black (although the converse isn't necessarily true).

Rule 4. Every path from the root to a leaf, or to a null child, must contain the same number of black nodes.

The "null child" referred to in Rule 4 is a place where a child could be attached to a non-leaf node. In other words, it's the potential left child of a node with only a right child or the potential right child of a node with only a left child. This description will make more sense as we go along.

The number of black nodes on a path from the root to a leaf is called the **black height**. Another way to state Rule 4 is that the black height must be the same for all paths from the root to a leaf or a null child. Note that the black height is defined for a path, while the height metric used for AVL trees is for a whole subtree (and doesn't care about node colors).

These rules might seem completely mysterious. It's not obvious how they will lead to a balanced tree, but they do; some very clever people invented them. Copy them down somewhere and keep them handy. You'll need to refer to them often as you learn about red-black trees.

Aside from the basic rules, red-black trees are as flexible as binary search trees. They allow any kind of key, as long as they can be compared to determine an ordering. They don't allow duplicate keys and act as an associative key store. People familiar with the red and black coloring of roulette wheels might infer those keys must be odd or even numbered in some way, but that is wrong. You won't always see alternating colors along every path; rule 3 only prevents red nodes from having child nodes that are also red. The black nodes can have black or red child nodes.

Fixing Rule Violations

Suppose you see (or are told by the visualization tool that we introduce shortly) that the color rules are violated. How can you fix your tree so that it complies? There are two, and only two, possible actions you can take:

- You can change the colors of nodes.
- You can perform rotations.

In the tool, changing the color of a node means changing its red-black border color (not the center color). Changing the color means setting the Boolean flag to a different value for one or more nodes. Because it's a Boolean, you can call this "flipping" the color between one of the two values, akin to flipping a coin. Rotations, as you have seen, rearrange some nodes in a way that, one hopes, leaves the tree more balanced. They always preserve the search relationships of the keys. The colors of the nodes stay with them as they rotate, and that "stickiness" can cause (or fix) rule violations.

At this point such concepts probably seem very abstract, so let's become familiar with the RedBlackTree Visualization tool, which can help to clarify things.

Using the Red-Black Tree Visualization Tool

To launch the Visualization tool, follow the instructions in Appendix A, "Running the Visualizations," and either run the `RedBlackTree.py` program or select it from the menu of visualizations. Figure 10-14 shows what the RedBlackTree Visualization tool looks like when it starts with two nodes inserted.

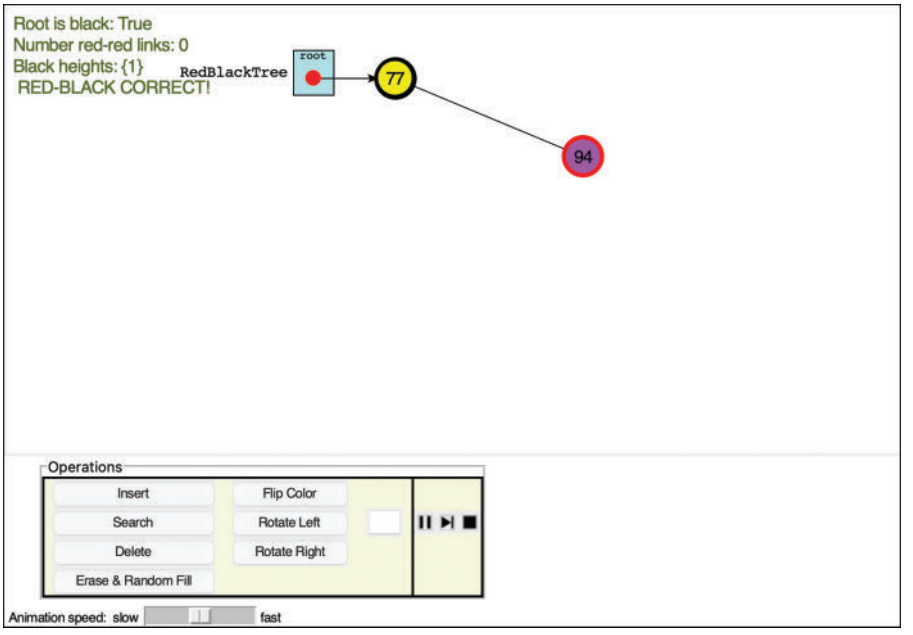


FIGURE 10-14 The RedBlackTree Visualization tool

Like the other trees you've studied, a `RedBlackTree` object has a single pointer to the root node of the tree. In the figure, node 77 is the root and is colored black. It has one child on the right, node 94, that is colored red.

In the upper left, three messages about the red-black rules appear (the first rule—that all nodes are colored red or black—is always enforced by the tool). Rule 2 is either true or false, depending on the color of the root. For Rule 3, the message counts the number of red nodes linked to other red nodes. For Rule 4, the message shows all the different black heights that can be found in the tree. In other words, it computes the black height for every leaf and null child in the tree and shows the set of unique heights inside of curly braces. In the case of Figure 10-14, node 94 is a leaf with a black height of 1, and the root node has a null left child that also has a black height of 1.

The red-black rules are all satisfied for this tree, so the messages are shown in green, and the last message says `"RED-BLACK CORRECT!"` That message disappears when any of the rules are violated.

Flipping a Node's Color

You can change a node's color by either clicking it with a pointer device or entering its key in the text entry box and selecting the Flip Color button. Try changing the color of node 94 to black. The ring color changes, and the messages change to show that one of the rules is violated, as shown in Figure 10-15. With exactly two black nodes in the tree, there are now two different black heights: 1 for the null left child of the root and 2 for leaf node 94. Flip node 94 back to red, and the rules are satisfied again.

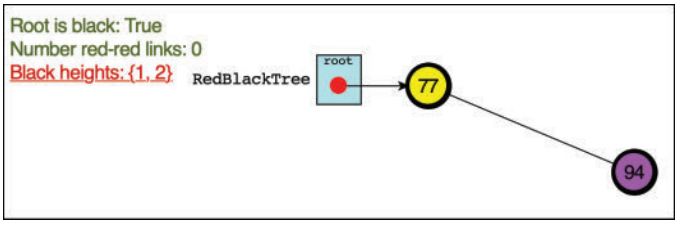


FIGURE 10-15 A tree with exactly two nodes, both black

One side effect of clicking a node to flip its color is that the node's key is entered into the text entry box for use in subsequent operations. This is a little different from other operations you've seen where the text entry box is cleared after an operation completes.

With node 94 red again, flip the color of root node 77 to red. Now two of the rules are violated, as shown in Figure 10-16. The root node is not black, violating Rule 2, and the one parent-child link is between two red nodes, violating Rule 3. The link is highlighted in red to indicate the problem. The two red nodes, however, satisfy Rule 4 because all the black heights are now 0.

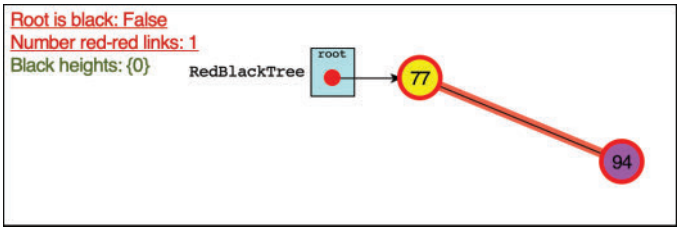


FIGURE 10-16 A tree with exactly two nodes, both red

Rotating Nodes

As you saw with AVL trees (and discussed for 2-3-4 trees), rotating around a node can change the balance of a tree. In the RedBlackTree Visualization tool, you select a top node of the subtree to rotate and then select either the Rotate Left or Rotate Right button to perform the rotation. You can select the top node by either typing its key in the text entry box or by clicking it with the pointer device to copy the key. The single click also changes its color, and if that is not desirable, you can click it a second time to revert the color.

Double-clicking a node also rotates that node to its right. The messages about the rules temporarily disappear. After the nodes are repositioned, the red-black rules are rechecked. If you hold down the Shift key or use the second mouse button when double-clicking, the rotation goes to the left. If you ask to rotate a node without a subtree to raise into its place, a message appears below the operations area explaining the problem.

The Insert Button

The Insert button causes a new node to be created, with the key that was typed into the text entry box (assuming there is space for it in the Visualization tool tree). Although the code isn't shown, it performs the familiar binary search tree algorithm to find where the key should be located. If the key already exists, it updates the data for that key by giving the data circle a new color. If the key doesn't exist but would go in a node at level 0 through 4, it inserts the new node with the key and data. Attempts to insert keys at level 5 or below fail with an error message. (Note also that rotations that move nodes to level 5 will delete those nodes.)

We discuss the choice of the red-black ring color of the new node in the next section. The color, of course, affects the red-black rules, so changes may be needed to keep the tree balanced.

The Search Button

The Search button acts like the one you've seen in the other tree visualization tools. It follows the binary search tree algorithm from the root (without animation) and reports whether the key was found or not. When the key is found, the node is encircled to highlight it.

The Delete Button

The Delete button acts like the one for the Binary Search Tree. First, it locates the requested key (without animation). If it's found, the number of children the node has determines how the deletion proceeds. With zero or one child, the parent's link to the node is adjusted. With two children, the successor node is found, copied to the node being deleted, and then the successor node is removed. The red-black rules are checked on the reduced tree to update the summary.

The Erase & Random Fill Button

When you want to start from an empty tree, type `0` in the text entry box and select the Erase & Random fill button. You can also create bigger (and more complex) trees by typing a larger number, up to 99. As the Binary Search Tree Visualization tool showed, you may not be able to insert keys in a sequence that fills all possible 31 nodes.

The red and black colors assigned to the nodes are not really random; their assignment follows the logic of individual insertions. Frequently, that will lead to at least one violation of red-black rules when the tree size is three or more. Occasionally, the random filling satisfies all the rules.

Experimenting with the Visualization Tool

Now that you're familiar with the RedBlackTree operations, let's do some simple experiments to get a feel for what the tool does. The idea here is to learn to manipulate the tool's controls. Later you use these skills to balance the tree.

Experiment 1: Inserting Two Red Nodes

If there's already a tree, clear it by typing `0` in the text entry box and selecting Erase & Random Fill. Next, type `50` and select Insert. It's convenient to experiment with a root key of 50 because that number provides maximum flexibility to insert keys on either side. Not surprisingly, all the red-black rules are satisfied for this single node tree because the insert operation chooses black for root nodes.

Insert a second node with a value smaller than the root, say 20. The insert operation makes this node red, preventing any rule violations.

Insert a third node that's larger than the root, say 71. The new node is also red, and the tree remains red-black correct. It's also balanced. The result is shown in Figure 10-17.

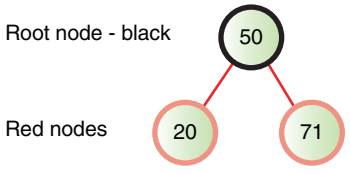


FIGURE 10-17 A balanced tree

Notice that newly inserted nodes are always colored red (except for the root). This color is not an accident. Inserting a red node is less likely to violate the red-black rules than inserting a black one. The reason is that, if the new red node is attached to a black one, no rule is broken. The first insertion doesn't create a situation in which there are two red nodes together (Rule 3), and it doesn't change the black height in any of the paths (Rule 4). Of course, if you insert a new red node below a red node, Rule 3 will be violated. With any luck, however, this will happen only half the time. On the other hand, adding a new black node always changes the black height for its path, violating Rule 4 if the tree already satisfied that rule.

Also, it's easier to fix violations of Rule 3 (parent and child are both red) than Rule 4 (black heights differ), as you see later.

Experiment 2: Rotations

Let's try some rotations. Start with the three nodes shown on the left in Figure 10-18. Select node 50 as the top node in the rotation (by either typing the key in the text entry box or clicking it twice to select it without changing its color). Now perform a right rotation by selecting the Rotate right button. The nodes all shift to new positions, as shown in the right of Figure 10-18.

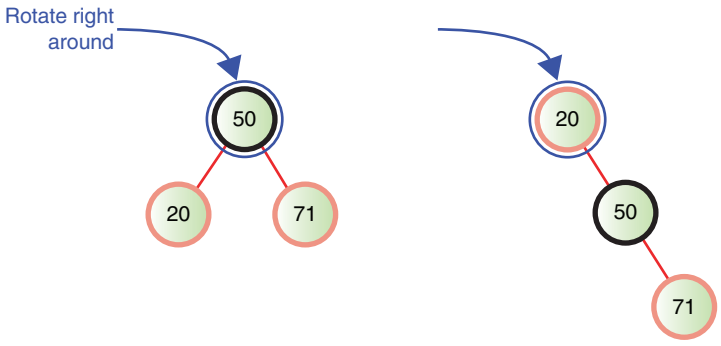


FIGURE 10-18 The right rotation operation

In this right rotation, the parent, or top, node moves into the place of its right child, the left child moves up and takes the place of the parent, and the right child moves down to become the grandchild of the new top node.

Notice that the tree is now unbalanced; the height of the right subtree is two more than that of the left. Also, the status messages indicate that the red-black rules are violated, specifically Rule 2 (the root is always black) and Rule 4 (black heights must be the same).

It's easy to get back to balance by rotating the other way. This time, you must select node 20 as the rotation top. Use the Rotate left button or hold the Shift key while double-clicking node 20 to rotate left. The nodes return to the position of Figure 10-17.

Experiment 3: Color Swaps

Start with the situation of Figure 10-17, with nodes 20 and 71 inserted in addition to 50 in the root position. Note that the parent (the root) is black, and both its children are red. Now try to insert another node, say with key value 12. No matter what value you use, you'll see a new kind of change. The animation shows the black color of the root being copied to the two red children. What's going on?

A color swap is necessary in the following situation: *when you're searching for the insertion point and encounter a black node with two red children*. If the two red children are leaves, it's clear why the colors must be flipped. Inserting a new red node as a child of a red node would violate Rule 3. Flipping the color of the parent and both of its children avoids that problem.

Or course, if you swap the black root's color with its two red children, the root becomes red and violates Rule 2. When performing color swaps, we make an exception for the root node and leave it black.

Figure 10-19 shows the steps that happen while inserting a node with key value 12. As the internal `__find()` method descends the tree to locate the insertion point, it checks the current node's color and that of its children. In this case, it finds the black-red-red pattern at root node 50 and performs a swap (but keeps the root black). After it descends to node 20, it doesn't find the black-red-red pattern. The insertion goes in node 20's left child slot.

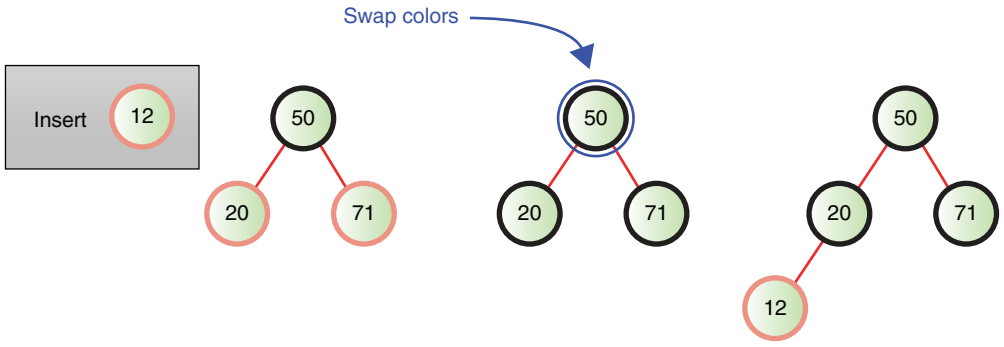


FIGURE 10-19 Inserting a node with a color swap

The tree remains red-black correct throughout the process. The root is black, there's no situation in which a parent and child are both red, and all the paths have the same number of black nodes (two). Adding the new red node doesn't change the red-black correctness.

Experiment 4: An Unbalanced Tree

Now let's see what happens when you try to do something that leads to an unbalanced tree. In the final tree of Figure 10-19, one path has one more node than the other. This example isn't very unbalanced and satisfies the definition of balance used with the AVL tree. No red-black rules are violated, so neither you nor the red-black algorithms need to worry about making changes.

Let's create an unbalanced tree. For this example, insert a node with key 6 into the final tree of Figure 10-19. You'll see the situation shown in Figure 10-20. Node 6 is red and lies beneath a red node 20. The red-red link is highlighted, and the status message shows the count of 1 for such links.

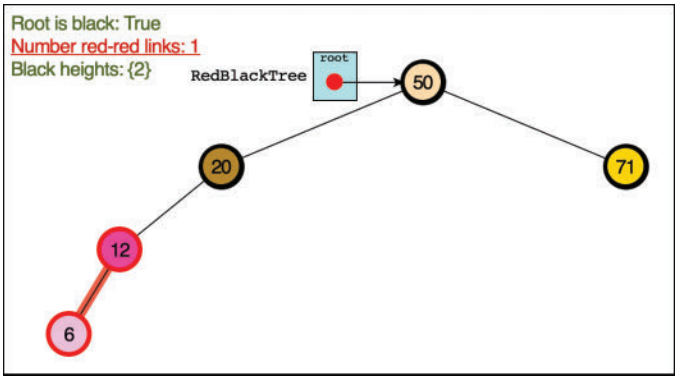


FIGURE 10-20 Parent and child are both red

How can you fix the tree so that Rule 3 isn't violated? An obvious approach is to change one of the indicated nodes to black. Try changing the child node 6 to black by clicking it.

The good news is you fixed the problem of both parent and child being red. The bad news is that now the status message for Rule 4 says Black heights: {2, 3}. The path from the root to node 6 has three black nodes in it, while the path from the root to node 71 has only two. It seems you can't win.

This problem, however, can be fixed with a rotation and some color changes. How to do this is the topic of later sections.

More Experiments

Experiment with the RedBlackTree Visualization tool on your own. Insert more nodes and see what happens. See whether you can use rotations and color changes to achieve a balanced tree. Does keeping the tree red-black correct seem to guarantee an (almost) balanced tree?

Try inserting five ascending keys (50, 60, 70, 80, 90) in an empty starting tree. Ignore the status messages until all the keys are inserted; then try flipping colors and rotating nodes. Can you balance the tree? If so, how many operations did it take? Restart with an empty tree and try five descending keys (50, 40, 30, 20, 10). Can you balance this tree with fewer operations?

The Red-Black Rules and Balanced Trees

Try to create a tree that is unbalanced by two or more levels but is red-black correct. In other words, try to create a tree where at least one node has a height difference of 3 or more. As it turns out, this is impossible. That's why the red-black rules keep the tree balanced. If one path is more than one node longer than another, it must either have more black nodes, violating Rule 4, or it must have two adjacent red nodes, violating Rule 3. Convince yourself that this is true by experimenting with the tool.

Null Children

Remember Rule 4, which specifies that all paths that go from the root to any leaf or to *any null children* must have the same number of black nodes. A null child is a child that a non-leaf (internal) node might have but doesn't. In other words, it's a missing left child if the internal node has a right child, or vice versa. In Figure 10-21 the path from 50 to 20 to the right child of 20 (its null child) has two black nodes in the tree on top. The path from 50 to 20 to 12 has three, which violates Rule 4. The paths to both leaf nodes have the same number of black nodes, but it's the null children higher up that cause the problem. Even if those internal nodes are red, as shown in the tree on the bottom, the number of black nodes in the four paths don't match.

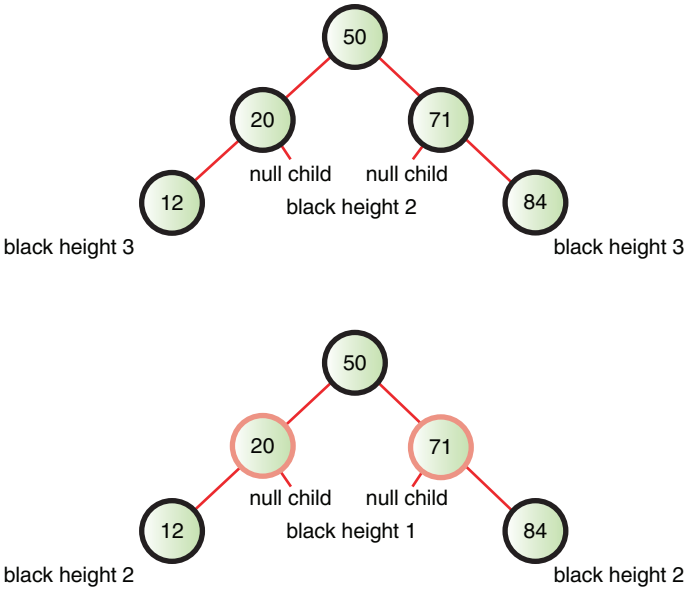


FIGURE 10-21 Paths to null children

Rotations in Red-Black Trees

To balance a tree, some algorithm must rearrange the nodes. If too many nodes are on the left of the root, as in Figure 10-20 for example, you need to move some of them over to the right side. As with the other binary trees, this is done using rotations. In this section you learn how rotations interact with the red-black rules.

Note that red-black rules and node colors are used only to help decide when to perform a rotation. Fiddling with the colors doesn't accomplish anything by itself; it's the rotation that's the heavy hitter. It's something like making improvements to your house or apartment. Changing the colors of walls and adding plants can change the mood of the room, while moving walls and adding doorways changes the whole structure.

Subtrees on the Move

You've seen individual nodes changing position during a rotation, but as you saw with AVL trees, entire subtrees can move as well. To see this movement, start with an empty tree (by entering `0` and selecting Erase & Random Fill), and then insert the following sequence of nodes in order: 50, 25, 75, 12, 37, 62, 87, 6, 18, 31, 43. You will see color swaps occur when you insert nodes 12 and 6. The resulting arrangement is shown on the left of Figure 10-22.

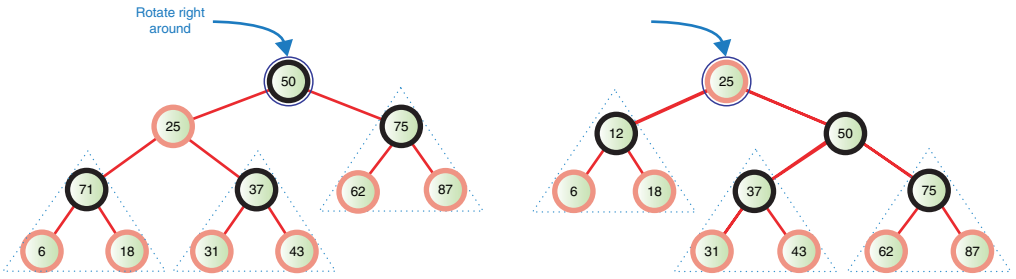


FIGURE 10-22 Subtree movement during rotation

Now rotate around root node 50 by double-clicking it or typing `50` and selecting the Rotate Right button. A lot of nodes have changed position. The result is shown on the right of Figure 10-22. Here's what happens:

- The top node (50) goes to its right child.
- The top node's left child (25) goes to the top (root).
- The entire subtree rooted at node 12 moves up.
- The entire subtree rooted at 37 crosses over to become the left child of 50.
- The entire subtree rooted at 75 moves down as the right child of node 50.

The status messages indicate that the root node isn't black and the black heights differ. You can ignore these messages for the moment. Try rotating back and forth around the root by double-clicking the node and holding down the Shift key or using the second mouse button when rotating left. Do this (perhaps with the animation speed slowed down) and watch what happens to the subtrees, especially the one with 37 as its root.

In Figure 10-22, the subtrees are enclosed in dotted triangles. Note that the rotation doesn't affect relations of the nodes within each subtree. The entire subtree moves as a unit. The subtrees can be larger (have more descendants) than the three nodes shown in this example. No matter how many nodes there are in a subtree, they will all move together during a rotation. Even when a subtree is empty, the movement is the same. The empty subtree goes to the same position it would have moved to if it had some nodes.

Inserting a New Node

Now you have enough background to see how a red-black tree's insertion routine could use rotations and the color rules to maintain the tree's balance. The remaining discussion describes the little details of exactly what conditions trigger what color changes and rotations to preserve balance.

Preview of the Insertion Process

Here, we briefly preview our approach to describing the insertion process. Don't worry if things aren't completely clear in the preview; we discuss this process in more detail in a moment.

The discussion that follows uses X, P, and G to designate a pattern of related nodes. X is a node that has caused a rule violation. Sometimes X refers to a newly inserted node, and sometimes to the child node when a parent and child have a red-red conflict.

- X is a particular node.
- P is the parent of X.
- G is the grandparent of X (the parent of P).

On the way down the tree to find the insertion point, you perform a color swap whenever you find a black node with two red children. Sometimes the swap causes a red-red conflict (a violation of Rule 3). Call the red child X and the red parent P. The conflict can be fixed with a single rotation or a double rotation, depending on whether X is an outside or inside grandchild of G. Following some color flips and rotations, you continue down to the insertion point and insert the new node.

After you've inserted the new node X, if P is black, you simply attach the new red node. If P is red, there are two possibilities: X is an outside or inside grandchild of G. You perform two color changes (we show what they are in a moment). If X is an outside grandchild, you perform one rotation, and if it's an inside grandchild, you perform two. This process restores the tree to a balanced state.

Now let's recapitulate this preview in more detail. We divide the discussion into three parts, arranged in order of complexity:

1. Color swaps on the way down
2. Rotations after the node is inserted
3. Rotations on the way down

If we were discussing these three parts in strict chronological order, we would examine part 3 before part 2. It's easier, however, to talk about rotations at the bottom of the tree than in the middle, and operations 1 and 2 are encountered more frequently than operation 3, so we discuss 2 before 3.

Color Swaps on the Way Down

The insertion routine in a red-black tree starts off doing essentially the same thing it does in an ordinary binary search tree: it follows a path from the root to the place where the node should be inserted, going left or right at each node, depending on the relative size of the node's key and the key to insert.

In a red-black tree, however, getting to the insertion point is complicated by color swaps and rotations. In the earlier "Experiment 3: Color Swaps" section, you learned the effects of changing colors; now let's look at them in more detail.

Imagine the insertion routine proceeding down the tree, going left or right at each node, searching for the place to insert a new node. To make sure the color rules aren't broken in the upcoming insertion, it needs to perform color swaps when necessary. Here's the rule: every time the insertion routine encounters a black node that has two red children, it changes the children to black and the parent to red (unless the parent is the root, which always remains black).

How does a color swap affect the red-black rules? For convenience, let's call the node at the top of the triangle, the one that's red before the swap, P for parent. You can call P's left and right children X1 and X2. This arrangement is shown on the left of Figure 10-23.

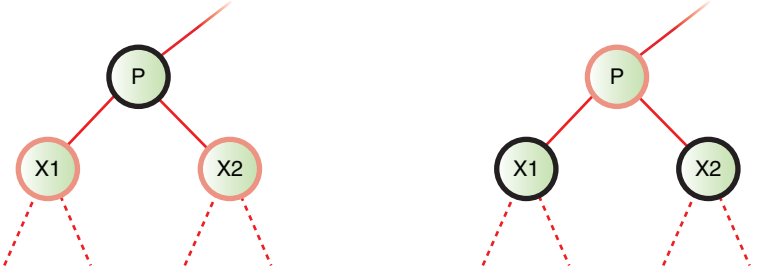


FIGURE 10-23 Color swap

Black Heights Unchanged

Figure 10-23 shows the nodes after the color swap on the right. The swap preserves the number of black nodes on the path from the root on down through P to any leaf or null node. All such paths go through P, and then through either X1 or X2. Before the swap (on the left of the figure), only P is black, so the triangle (consisting of P, X1, and X2) adds one black node to each of these paths.

After the swap, P is no longer black, but both X1 and X2 are, so again the triangle contributes one black node to every path that passes through it. In the exception where P is the root of the tree and the color swap leaves it black, one black node is added to every path in the tree. Thus, all the paths have the same black height, and a color swap can't cause Rule 4 to be violated.

Color swaps are helpful because they turn red leaf nodes into black leaf nodes. This design makes it easier to attach new red nodes without violating Rule 3.

Violation of Rule 3

Although Rule 4 is not violated by a color swap, Rule 3 (a node and its parent can't both be red) may be. If the parent of P is a black node, there's no problem when P is changed from black to red. If the parent of P is red, however, the color change creates two linked red nodes.

This problem must be fixed before you continue down the path to insert the new node. You can correct the situation with a rotation, as we'll soon see.

Inserting at a Leaf

After you've worked your way down to the appropriate place in the tree, performing color swaps (and rotations) if necessary, on the way down, you can then insert the new node as described in Chapter 8 for an ordinary binary search tree. That's not, however, the end of the story.

Rotations After the Node Is Inserted

The insertion of the new node may cause the red-black rules to be violated. Figure 10-20 showed the example of inserting a node with key 6 that violates Rule 3. Therefore, following the insertion, you must check for rule violations and take appropriate steps.

Remember that, as described earlier, the newly inserted node, which is called X, is always red. X may be located in exactly four positions relative to P and G, as shown in Figure 10-24.

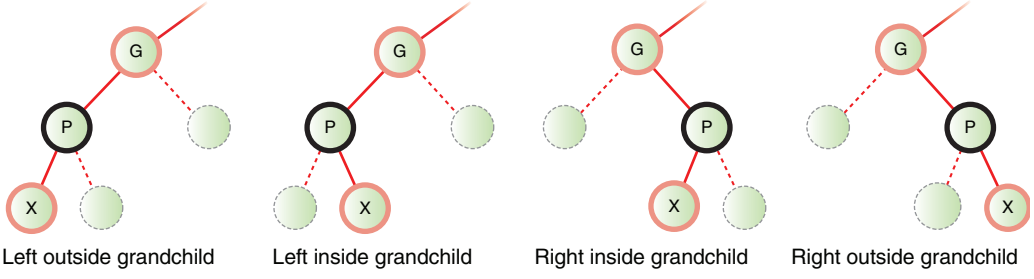


FIGURE 10-24 Handed variations of node being inserted

Remember that a node X is an outside grandchild if it's on the same side of its parent, P, that P is of its parent, G. That is, X is an outside grandchild if either it is a left child of P and P is a left child of G, or it is a right child of P and P is a right child of G (as in the two outermost relations of Figure 10-24). Conversely, X is an inside grandchild if it's on the opposite side of its parent, P, that P is of its parent, G (as in the inner relations of Figure 10-24).

The multiplicity of what might be called "handed" (left or right) relationships between the inserted node and its ancestors is one reason the red-black insertion routine is so complex to program.

The action needed to conform to the red-black rules is determined by the colors and configuration of X and its relatives. Perhaps surprisingly, nodes can be arranged in only three major ways (not counting the handed variations already mentioned). Each possibility must be dealt with in a different way to preserve red-black correctness and thereby lead to

a balanced tree. We list the three possibilities briefly and then discuss each one in detail in its own section. Figure 10-25 shows what they look like. Remember that X is always red.

1. P is black.
2. P is red and X is an outside grandchild of G.
3. P is red and X is an inside grandchild of G.

You might think that this list doesn't cover all the possibilities. We return to this question after we have explored these three.

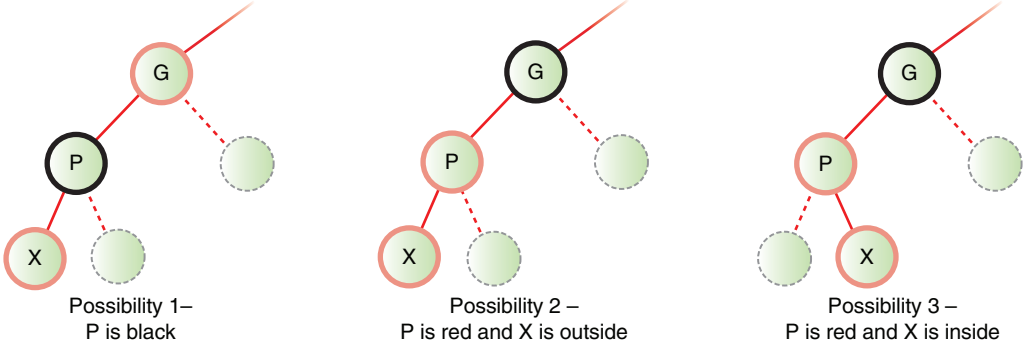


FIGURE 10-25 Three post-insertion possibilities

Possibility 1: P Is Black

If P is black, you get a free ride. The node you've just inserted is always red. If its parent is black, there's no red-to-red conflict (Rule 3) and no addition to the number of black nodes (Rule 4). Thus, no color rules are violated. You don't need to do anything else. The insertion is complete.

Possibility 2: P Is Red and X Is Outside

If P is red and X is an outside grandchild, you need a single rotation and some color changes. The single rotation is the same as what was needed in an AVL tree, but the color changes are new. Let's return to the example in Figure 10-20, which was set up in the RedBlackTree Visualization tool by inserting 50, 20, 71, 12, and 6 in an empty tree. Figure 10-26 shows the situation along with the X, P, and G labels. The status in the Visualization tool says there is one red-red link, so you know you need to take some action.

In this situation, you can take three steps to restore red-black correctness and thereby balance the tree. Here are the steps:

1. Switch the color of X's grandparent G (20 in this example).
2. Switch the color of X's parent, P (12).
3. Rotate with X's grandparent, G (20), at the top, in the direction that raises X (6). This is a right rotation in the example.

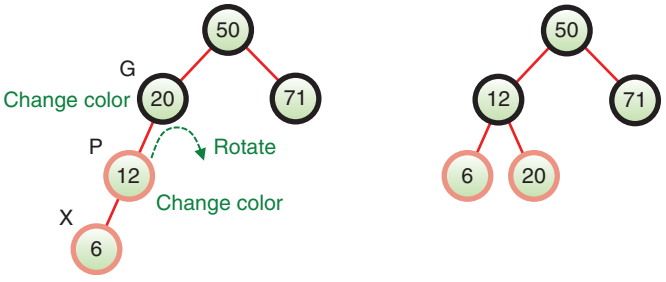


FIGURE 10-26 P is red and X is an outside grandchild

As you've learned, to flip colors in the Visualization tool, single-click the node with a pointer device. To rotate right, double-click the top node. (Alternatively, single-click the parent, 12, then single-click the grandparent, 20, and then select the Rotate Right button while 20 has already been filled in as the rotation point.) After you've completed the three steps, the Visualization tool will inform you that the tree is red-black correct. It's also balanced, as shown at the right of Figure 10-26.

In this example, X was an outside grandchild and a left child. There's a symmetrical situation when the X is an outside grandchild but a right child. Try this by creating the tree 50, 25, 75, 87, 93. Fix it by changing the colors of 87 and 75, and rotating left at node 75. Again, the tree becomes balanced.

Possibility 3: P Is Red and X Is Inside

If P is red and X is an inside grandchild, you need two rotations and some color changes. To see this one in action, use the Visualization tool to create the tree with keys 50, 20, 71, 12, 17. The result is the tree on the left of Figure 10-27.

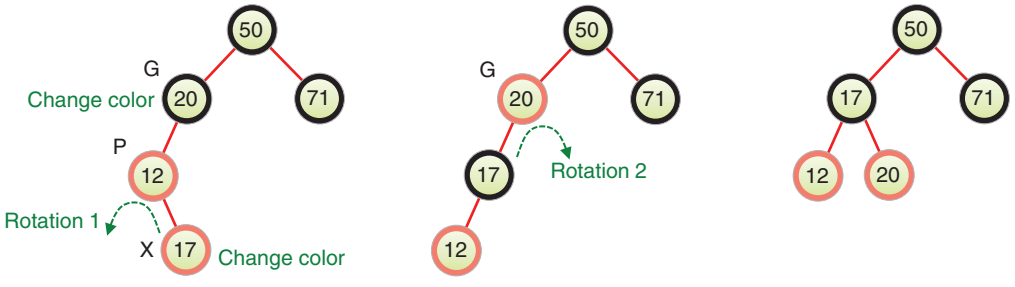


FIGURE 10-27 P is red and X is an inside grandchild

Note that node 17 is an inside grandchild. Both it and its parent are red, so again you see the status message that there is one red-red link.

Fixing this arrangement is slightly more complicated. If you try to rotate right with the grandparent node, G (20), at the top, as you did in Possibility 2, the inside grandchild, X (17), moves across rather than up, so the tree is no more balanced than before. (Try this; then rotate back left, with 12 at the top, to restore it.) A different solution is needed.

The trick when X is an inside grandchild is to perform *two* rotations rather than one, just as was needed for AVL trees. Rotation 1 changes X from an inside grandchild to an outside grandchild, as shown in the middle of Figure 10-27. Now the situation is like Possibility 1, and you can apply the same rotation, with the grandparent at the top, as you did before. The result is shown on the right.

You must also recolor the nodes. You do this before doing any rotations. The order of these operations doesn't really matter, but if you wait until after the rotations to recolor the nodes, it's hard to know what to call them. Here are the steps:

1. Switch the color of X's grandparent (20 in this example).
2. Switch the color of X (*not* its parent; X is 17 here).
3. Rotate with X's parent, P, at the top (*not* the grandparent; the parent is 12), in the direction that raises X (a left rotation in this example).
4. Rotate again with X's grandparent, G (20), at the top, in the direction that raises X (a right rotation).

The rotations and recoloring restore the tree to red-black correctness and balance it. As with Possibility 2, there is a mirror image case in which P is the right child of G rather than the left.

What About Other Possibilities?

Do the three Post-Insertion Possibilities just discussed really cover all situations?

Suppose, for example, that X has a sibling, S, the other child of P. This scenario might complicate the rotations necessary to insert X and would be as if the bottom dashed nodes in each of the possibilities of Figure 10-25 were filled in. If P is black, then there's still no problem inserting X, which is red (that's Possibility 1).

What about when P is red? Well, you know that the tree was balanced just before the insertion, so G must be black when P is red (to avoid violating Rule 3). If there were a sibling of X, it too would have to be black (to avoid violating Rule 3). That would mean, just before the insertion, P had only one child, the sibling, and it was black. That's not balanced because P's null child would have a different number of black nodes in its path, violating Rule 4. You can conclude that it's impossible for X to have a sibling when P is red.

Another case to explore is that G, the grandparent of P, has a child, U, the sibling of P and the uncle of X (the top dashed nodes in Figure 10-25). Again, this scenario might complicate any necessary rotations. If P were black, there would be no need for rotations when inserting X, as you've seen. Let's assume P is red and G is black (as in Possibilities 2 and 3 of Figure 10-25). Then U must also be red; otherwise, the black height going from G to P's null child would be different from that going from G to U. Because a black parent, G, with two red children, P and U, would be swapped on the way down, you conclude that this situation can't exist either.

Thus, the three possibilities just discussed are the only ones that can exist (except for the mirror cases of Possibilities 2 and 3, X can be a right or left child and P can be a right or left child).

What the Color Swaps Accomplished

Suppose that performing a rotation and appropriate color changes caused other violations of the red-black rules to appear further up the tree. You can imagine situations in which you would need to work all the way back up the tree, performing rotations and color switches, to remove rule violations. That's what was done in AVL trees.

Fortunately, this situation can't arise. Using color swaps on the way down eliminates the situations in which a rotation could introduce any rule violations further up the tree (we cover rotations on the way down in the next section). When you get to inserting at the leaf, it's guaranteed that one or two rotations plus two color flips will restore red-black correctness in the entire tree. Proving this guarantee is beyond the scope of this book, but such a proof is possible.

The color swaps on the way down make insertion in red-black trees more efficient than in other kinds of balanced trees, such as AVL trees. They ensure that you need to pass through the tree only once, on the way down. If the implementation is recursive, no work is needed after the recursive call returns.

Rotations on the Way Down

Now let's discuss the last of the three operations involved in inserting a node: making rotations on the way down to the insertion point. As we noted, although we discuss this operation last, it actually takes place before the node is inserted. We waited until now to discuss it only because it was easier to explain rotations for a just-installed node than for nodes in the middle of the tree.

In the discussion of color swaps during the insertion process, we noted that a color swap could cause a violation of Rule 3 (a parent and child can't both be red). The problem arises when you visit a black node with two red children and a red parent. We also noted that a rotation could fix this violation.

On the way down there are two possibilities, corresponding to Possibility 2 and Possibility 3 during the insertion phase described earlier. The offending node can be an outside grandchild, or it can be an inside grandchild. (In the situation corresponding to Possibility 1, no action is required.)

Outside Grandchild

First, let's examine an example in which the offending node is an outside grandchild. By "offending node," we mean the child in the parent-child pair that caused the red-red conflict.

For this example, start a new tree and insert the following keys: 50, 25, 75, 12, 37, 6, and 18. The insertion of 12 and 6 will cause color swaps.

Now try to insert a node with the value 3. The tree initially looks like the one on the left in Figure 10-28. As the search for the insertion point unfolds, it recognizes the need to swap colors at the subtree rooted at node 12. When the colors are swapped, node 12 becomes red, matching its parent, node 25. The Visualization tool proceeds to insert node 3, but let's look at the situation right after the color swap at node 12.

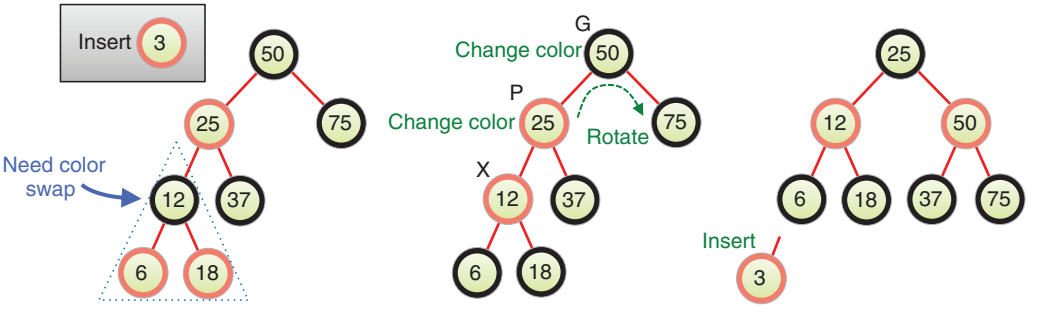


FIGURE 10-28 Correcting a red-red outside grandchild on the way down

The middle tree in Figure 10-28 shows the red-red conflict where node 12 is the problem, X; node 25 is its parent, P; and node 50 is its grandparent, G. Similar to the way you handle insertion at the leaf level, you must perform two color changes and one rotation. This example looks a little odd because X referred to the node being inserted in the earlier discussion, and here it's not even a leaf node. These on-the-way-down rotations, however, can take place anywhere within the tree.

You follow the same set of rules you did under Possibility 2: P Is Red and X Is Outside, discussed earlier:

1. Flip the color of X's grandparent G (50 in this example). Ignore the status message about the root not being black.
2. Flip the color of X's parent P (25).
3. Rotate with X's grandparent (50) at the top, in the direction that raises X (here a right rotation).

These steps result in the tree on the right of Figure 10-28. Suddenly, the tree is not only balanced but has also become pleasantly symmetrical! This outcome appears to be a bit of a miracle, but it's only the result of following the color rules.

Now the node with value 3 can be inserted in the usual way. Because the node it connects to, 6, is black, there's no complexity about the insertion.

When you try this in the Visualization tool, it doesn't perform the rotation automatically during the insertion of node 3; you will have to do it after the insertion. During the animation, when the `current` arrow reaches node 12, the tool shows the top panel in Figure 10-29. The bottom panel of the figure shows the display after the color swap and insertion of node 3.

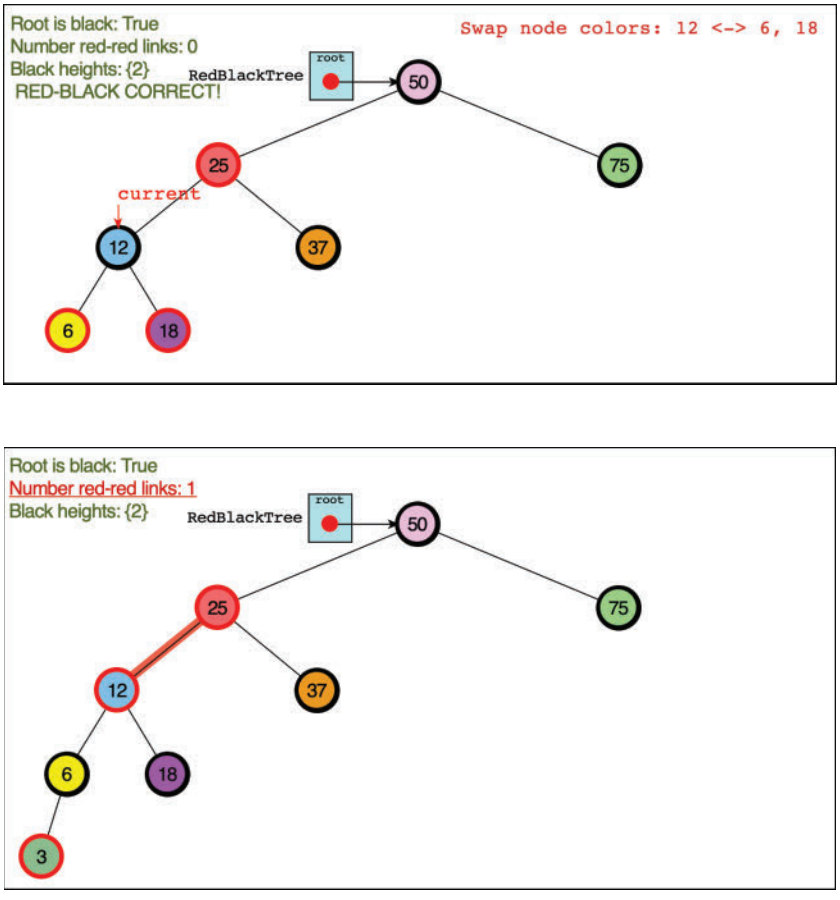


FIGURE 10-29 Correcting a red-red outside grandchild in the Visualization tool

The same color flips and rotation can be performed after the insertion to satisfy the red-black rules. In fact, if the algorithm were omniscient, it could perform the color flips and rotation on the original tree, even before an insertion was attempted. Sadly, omniscience is not an option. It's quite practical, however, to detect and fix the problem on the downward search.

Inside Grandchild

If X is an inside grandchild when a red-red conflict occurs on the way down, two rotations are required to set it right. This situation is like that of the inside grandchild in the post-insertion phase, which we called Possibility 3.

To see it in action, empty the tree in the Visualization tool, and insert 50, 25, 75, 12, 37, 31, and 43. Now try to insert a new node with the value 28. During the animation of the `current` arrow descending, it swaps node 37's color with that of its children. That leaves node 37 red and in conflict with its parent, node 25, as shown at the left of Figure 10-30. In this situation, G is 50, P is 25, and X is 37.

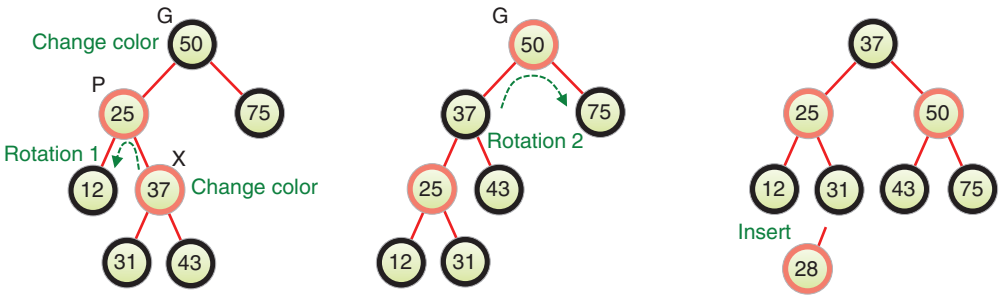


FIGURE 10-30 Correcting a red-red inside grandchild on the way down

To cure the red-red conflict, you must do the same two color flips and two rotations as described earlier in "Possibility 3: P Is Red and X Is Inside":

1. Change the color of G (node 50; ignore the message that the root is not black).
2. Change the color of X (37).
3. Rotate with P (25) as the top, in the direction that raises X (left in this example). That produces the tree in the middle of Figure 10-30.
4. Rotate with G (50) as the top, in the direction that raises X (right in this example).

Now the insertion of node 28 can proceed as shown at the right of Figure 10-30. Again, this operation poses no additional challenges because both nodes 31 and 43 were swapped to black when you first tried to insert 28.

When you use the Visualization tool to insert 28, it will perform the color swap of nodes 37, 31, and 43. You will have to perform the preceding four steps after the insertion of 28 is done.

We've concluded the description of how a tree is kept red-black correct during the insertion process. Adherence to the red-black rules guarantees balance because the level of leaves can only differ by at most two within any subtree (forgetting about the red and black labels). That's a different definition of balanced when compared to AVL trees, and we look at what that means when we discuss efficiency.

Deletion

As you may recall from Chapter 8 and saw in the implementation of AVL trees earlier, coding for deletion is harder than for insertion. Chapter 9 described how deletion could be done in 2-3-4 trees using rotation and fusion operations to maintain balance. With red-black trees, rotation and color change operations can be done to maintain red-black correctness (balance), and the deletion process remains, as you might expect, quite complex.

In fact, the deletion process is so complicated that many programmers sidestep it in various ways. One approach, as with ordinary binary trees, is to mark a node as deleted without actually deleting it—a "soft" delete. Any search routine that finds the node knows not

to report it to the caller. Deleted nodes must still maintain their red-black status, however, in order to balance the overall tree. The soft delete solution works in situations where deletions are not a common occurrence. That approach, however, has a way of ending up causing performance issues when someone takes the implementation and applies it to some problem where deletions happen more frequently. In any case, we forgo a discussion of the deletion process.

The Efficiency of Red-Black Trees

Like ordinary binary search trees, a red-black tree allows for searching, insertion, and deletion in $O(\log_2 N)$ time, which is the same as O(log N) time. Search times should be almost the same in the red-black tree as in the ordinary tree because the red-black characteristics of the tree aren't used during searches. The difference comes, however, in the height of the red-black trees. The rule that constrains the number of levels, Rule 4, only counts the black nodes on the path. That means that there can potentially be a path to a black leaf that has exactly two black nodes in it, while a path to a red leaf has two black nodes and two red nodes, as in Figure 10-31. The count of black nodes is the same, but the path followed by a search is twice as long. According to Sedgewick (see Appendix B), it can be shown that the search cannot require more than $2\times\log_2 N$ comparisons. That constant 2 in front comes from the extra path length, and it's the worst case. On average it takes about $\log_2 N$ comparisons. In big O notation, remember, you ignore the constant so it remains O(log N).

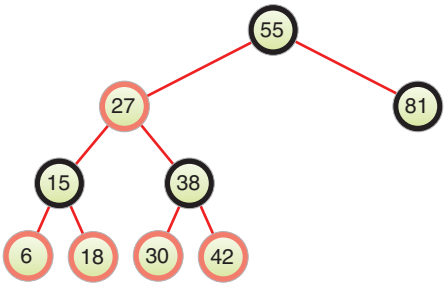


FIGURE 10-31 A red-black tree whose longest path is twice the shortest path

The times for insertion and deletion are increased by a constant factor because of having to perform color swaps and rotations on the way down and at the insertion point. On average, an insertion requires about one rotation. Therefore, insertion still takes O(log N) time but is slower than insertion in the ordinary binary tree. Compared with the AVL tree, which performs rotations bottom-up, the red-black tree is about twice as fast because it only needs to do work during the descent, although the other multipliers for color flips and rotations, plus the possibility of being deeper, change that a bit.

The storage penalty compared to ordinary binary search trees is the Boolean variable needed to label the color of each node. That's smaller than what the AVL tree required. It's also independent of the number of levels in the tree (you don't need more space per node when the number of levels gets huge).

Because in most applications there will be more searches than insertions and deletions, there is probably not much overall time penalty for using a red-black tree or an AVL tree instead of an ordinary binary search tree. Of course, the big advantage of red-black, AVL, and 2-3-4 trees is their insensitivity to the order items are inserted. None of them slow down to O(N) search performance, even if the data is inserted in the order of the keys.

2-3-4 Trees and Red-Black Trees

At this point 2-3-4 trees (described in Chapter 9) and red-black trees probably seem like entirely different entities. It turns out that in a certain sense they are completely equivalent. One can be transformed into the other by the application of a few simple rules, and even the operations needed to keep them balanced are equivalent. Mathematicians would say they are **isomorphic**.

You probably won't ever need to transform a 2-3-4 tree into a red-black tree, but the equivalence of these structures casts additional light on their operation and is useful in analyzing their efficiency.

Historically, the 2-3-4 tree was developed first; later the red-black tree evolved from it.

Transformation from 2-3-4 to Red-Black

As you might recall, the nodes of a 2-3-4 tree have either 2, 3, or 4 child links (possibly all empty for a leaf node). Each 2-3-4 node can be transformed into a red-black subtree by applying a rule based on its type, as shown in Figure 10-32. The rules are

- Transform any 2-node in the 2-3-4 tree into one black node in the red-black tree. Its two children, W and X, become the left and right children of the black node.
- Transform any 3-node into a child node, C, and a parent node, P. There are two possibilities for where to transform the three original child links—W, X, and Y—as shown in Figure 10-32. The child node, C, gets either W and X or X and Y as its child links. The parent, P, gets the one remaining child: either Y or W. It doesn't matter which item becomes the child and which the parent if the relationships between the keys are preserved. The parent is colored black, and the child is colored red.
- Transform any 4-node into a parent and two children. The first child gets children W and X; the second child gets children Y and Z. As before, the parent is black; the two children are colored red.

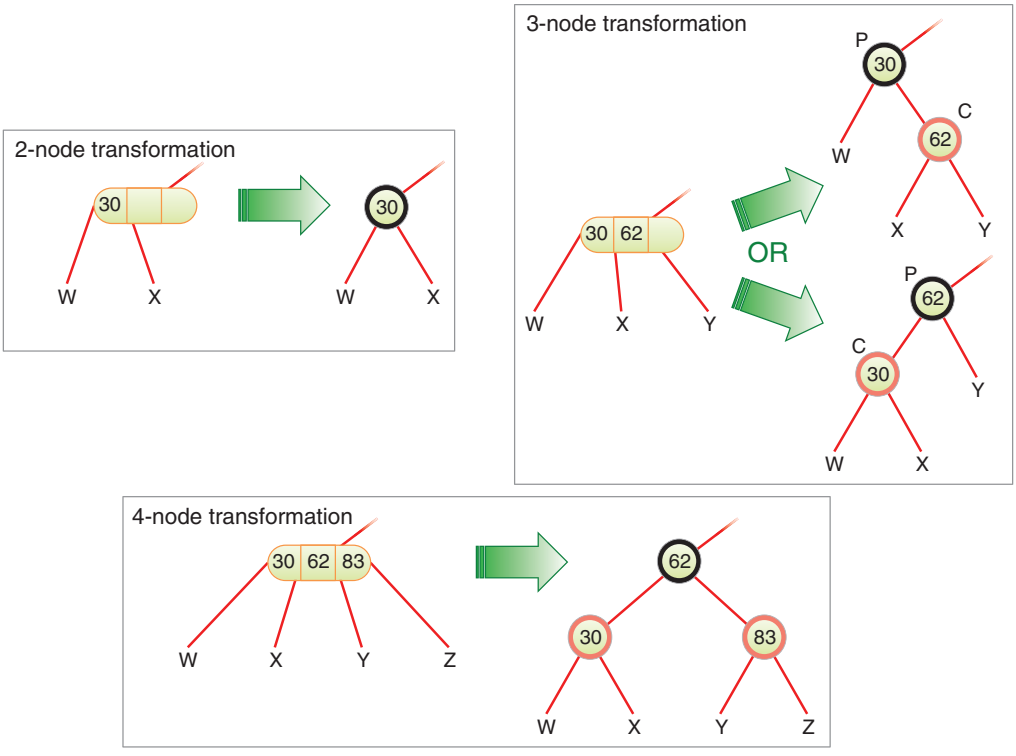


FIGURE 10-32 Transforming nodes: 2-3-4 to red-black

Remember that a leaf node storing a single item is considered a 2-node when applying these rules. In general, a leaf node storing L items transforms as an L+1 node does.

Figure 10-33 shows a 2-3-4 tree and its corresponding red-black tree obtained by applying these transformations. Dotted triangles surround the subtrees that were made from 4-nodes (one black and two red nodes). Dotted lozenges surround the subtrees that were made from 3-nodes (one black and one red node). The red-black rules are automatically satisfied by the transformation. Check that this is so: the root is black, two red nodes are never connected, and for every leaf and null child, the path to the root has the same number of black nodes, three.

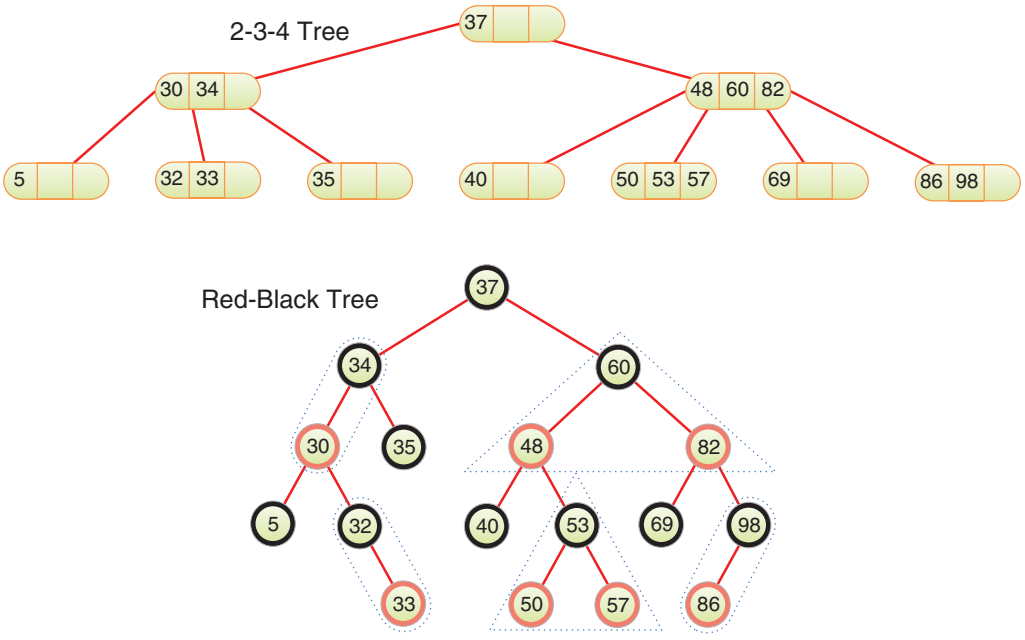


FIGURE 10-33 A 2-3-4 tree and its red-black equivalent

You can say that a 3-node in a 2-3-4 tree is equivalent to a parent with a red child in a red-black tree, and a 4-node is equivalent to a parent with two red children. It follows that a black parent with a black child in a red-black tree does *not* represent a 3-node in a 2-3-4 tree; it simply represents a 2-node with another 2-node child. Similarly, a black parent with two black children (like the root node of the red-black tree in Figure 10-33) does not represent a 4-node.

Operational Equivalence

Not only does the structure of a red-black tree correspond to a 2-3-4 tree, but the operations applied to these two kinds of trees are also equivalent. In a 2-3-4 tree, the tree is kept balanced during insertion using node splits. In a red-black tree, the balancing methods are color swaps, flips, and rotations.

4-Node Splits and Color Swaps

As the insertion algorithm descends a 2-3-4 tree searching for where a new node goes, it splits each 4-node into two 2-nodes. In a red-black tree, the algorithm calls for color swaps. Are these operations equivalent?

The top row of Figure 10-34 shows a 4-node (with keys 40-50-60) in a 2-3-4 tree being split. The 2-node holding item 70 that was the parent of the 4-node becomes a 3-node, holding 50 and 70.

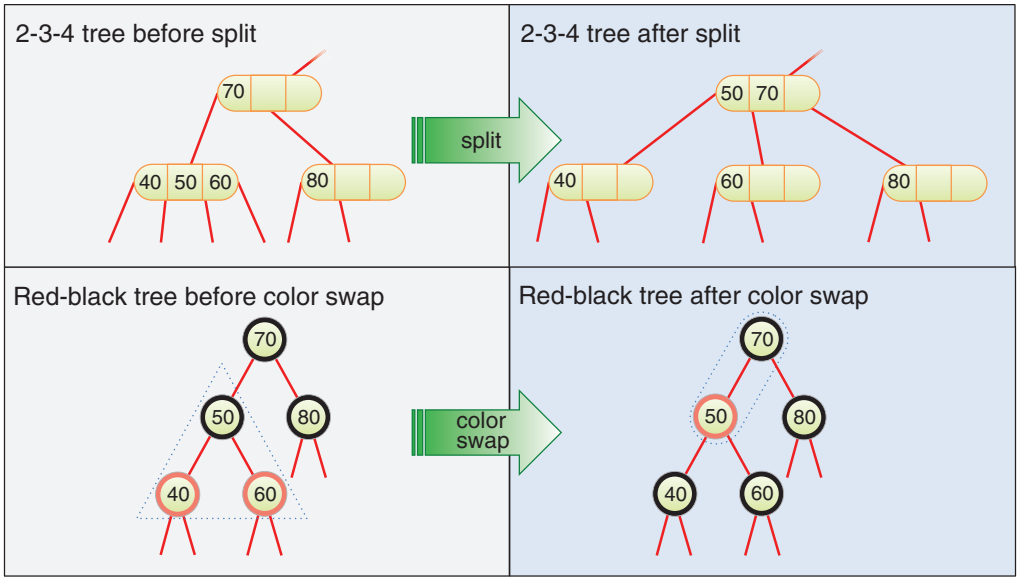


FIGURE 10-34 A 4-node split and color swap

The bottom row of Figure 10-34 shows the red-black equivalent to the 2-3-4 tree in the top row. The dotted triangle surrounds the equivalent of the 4-node, holding 40, 50, and 60. A color swap causes nodes 40 and 60 to become black and node 50 to become red. Thus, node 50 and its parent form the equivalent of a 3-node, as shown by the dotted lozenge shape surrounding nodes 50 and 70. This is equivalent to the 3-node holding 50 and 70 above.

Thus, it's clear that splitting a 4-node during the insertion process in a 2-3-4 tree is equivalent to performing color swaps during the insertion process in a red-black tree.

3-Node Splits and Rotations

When a 3-node in a 2-3-4 tree is transformed into its red-black equivalent, two arrangements are possible, as shown earlier in Figure 10-32. Either of the two data items can be placed as the parent. Depending on which one is chosen, the middle child of the 3-node is either a left child or a right child in the red-black tree, and the slant of the line connecting parent and child is either left or right. These are sometimes called *left-leaning* and *right-leaning red-black trees*.

Both arrangements are valid; however, they may not contribute equally to balancing the tree. Let's look at the situation in a slightly larger context.

The top panel in Figure 10-35 shows a 2-3-4 tree. The bottom panels show the left-leaning and right-leaning options for equivalent red-black trees derived from the transformation rules. The difference between them is the choice of which of the two data items in the 3-node at the top to make the parent. In the left-leaning option, node 80 becomes the parent, and in the right-leaning one, node 70 is chosen.

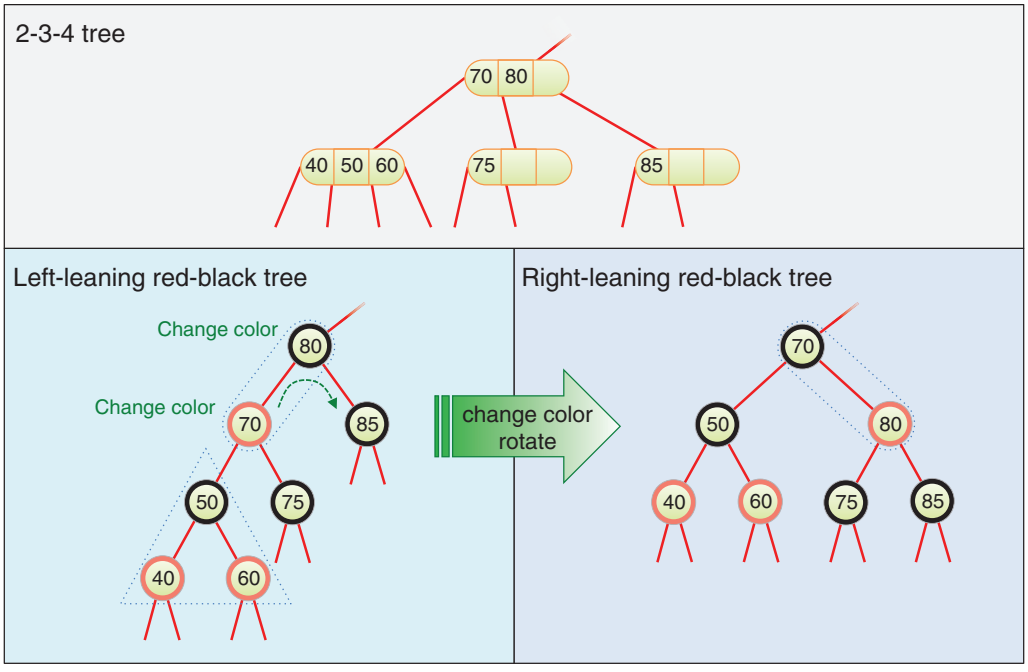


FIGURE 10-35 3-node transformation and equivalent rotation

Although these arrangements are equally valid, you can see that the tree on the left looks less balanced than the one on the right, even though both follow all the red-black rules. If you tried to insert an item with a key of 45 in the tree on the left, the insertion algorithm would descend the left branch and find that node 50 has two red children and perform a color swap. That would make node 50 red and violate Rule 3 with its parent, node 70. To fix that, you would apply the rule for Outside Grandchild conflicts. Amazingly, this rotation (performed on the tree without conflicts) results in the exact same tree as the right-leaning option!

Thus, you can see equivalence between rotations in red-black trees and the choice of which node to make the parent when transforming 2-3-4 trees to red-black trees. Although we don't show it, a similar equivalence can be seen for the double rotation necessary for inside grandchildren.

Red-Black Tree Implementation

If you're writing an insertion routine for red-black trees, all you would need to do (irony intended) is to write code to carry out the operations described in the preceding sections. As we noted, showing and describing such code are beyond the scope of this book. Nevertheless, here's what you would need to think about.

First, you'd need to add a red-black attribute to the `__Node` class. The constructor would create red nodes for insertion, by default.

You could adapt the insertion routine from the `BinarySearchTree` in Chapter 8, the `Tree234` class in Chapter 9, or the `AVLtree` of this chapter. On the way down to the insertion point, check if the current node is black and its two children are both red. If so, change the color of all three (unless the parent is the root, which must be kept black).

After a color swap, check that there are no violations of Rule 3. If so, perform the appropriate rotations: one for an outside grandchild, two for an inside grandchild.

When you reach a leaf node, insert the new node, making sure the node is colored red. Check again for red-red conflicts, and perform any necessary rotations.

Perhaps surprisingly, your software need not keep track of the height or black height of different parts of the tree (although you might want to check this during debugging). You need to check only for violations of Rule 3, a red parent with a red child, which can be done locally (unlike checks of black heights, Rule 4, which would require more complex bookkeeping like you did for the height in AVL trees).

If you perform the color swaps, color changes, and rotations described earlier, the black heights of the nodes take care of themselves, and the tree should remain balanced. The RedBlackTree Visualization tool reports black-height errors only to help you understand the rules; they don't need to be tracked by the algorithm. Although the concepts of the red-black tree are a bit harder to grasp, the implementation can be faster than that of an AVL tree, even though both remain O(log N).

Summary

- Keeping a binary search tree balanced ensures that the time to find a given node is as short as possible.
- Inserting data that has already been sorted can create a maximally unbalanced tree, which will have search times of O(N).
- You can measure tree balance by counting nodes or the length of paths in the left and right subtrees.
- You should measure balance at every node (every subtree), not just the root of the tree.
- In the AVL tree, each node keeps an attribute measuring its height.
- The height of a node is the number of nodes on the path to the deepest leaf node below it, including itself.
- The height of an empty subtree of a node is considered zero.
- A binary tree is fully balanced when the height difference of the left and right child of every node is at most one.
- AVL trees perform bottom-up correction of imbalances.
- Left and right rotations alter the positions and heights of subtrees.
- Rotations are implemented by changing the links between nodes of the tree.

- Rotations preserve the order of keys in the tree by moving the crossover subtree to the left or right child of the top of the rotation.
- When inserting a node causes an AVL subtree to become left or right heavy, it makes one rotation for an outside grandchild and two rotations for an inside grandchild to correct the imbalance.
- Deleting a node from an AVL subtree involves finding the node to delete, finding the successor of the node to delete, moving the successor item up to the node to delete, deleting the successor node, and correcting any imbalance created by the deletion of the successor node.
- Imbalances caused by deletion can also be corrected using one or two rotations by checking the height difference at a node and the child where the deletion occurred.
- Each AVL node's height must be updated after rotations, deletions, and insertions are performed on its subtrees.
- AVL trees take O(log N) for search, insertion, and deletion operations. Traversal is O(N).
- AVL trees consume O(N) memory.
- In the red-black balancing scheme, each node is assigned a characteristic—a color that is either red or black.
- A set of four rules, called the red-black rules, specifies permissible ways that nodes of different colors can be arranged.
- The red-black rules are maintained while inserting and deleting each node.
- A color swap changes a black node with two red children to a red node with two black children.
- Rotations alter the heights of subtrees and sometimes cause violations of the red-black rules.
- Color swaps, and sometimes rotations with color flips, are applied while searching down the tree to find where a new node should be inserted. These operations help return the tree to red-black correctness following an insertion.
- After a new node is inserted or an internal node undergoes a color swap, red-red conflicts are checked again. If a violation is found, appropriate rotations are carried out to make the tree red-black correct.
- These top-down adjustments result in the tree being balanced, or at least almost balanced.
- Red-black trees have the same big O performance as AVL trees do: O(log N) for search, insertion, and deletion operations. Traversal is O(N).
- In the worst case, the search path in a red-black tree is twice the length of that of an AVL tree holding the same items.

- Red-black trees consume O(N) memory.
- Red-black trees consume less memory per node than AVL trees because they need only one bit of storage for the node color, whereas AVL trees use an integer to track the height of the subtree.
- Adding balancing to a binary tree has only a small negative effect on average performance and avoids worst-case performance when the inserted data is ordered by key.
- A red-black tree can be derived from any 2-3-4 tree, maintaining balance.
- Insertion and deletion operations on 2-3-4 trees can be shown to be equivalent to the corresponding operations in red-black trees.

Questions

These questions are intended as a self-test for readers. Answers may be found in Appendix C.

1. A balanced binary search tree is desirable because it avoids slow performance when data is inserted _____.
2. In a balanced tree, there are roughly the same number of nodes in the left and right subtrees, and
   a. the tree may need to be restructured during searches.
   b. the paths from the root to all the leaf nodes are about the same length.
   c. all left subtrees are the same height as all right subtrees.
   d. the starting level of all subtrees is closely controlled.
3. Which of the following metrics guarantees a balanced binary tree if it is less than or equal to one?
   a. the sum of the differences between the node count of the left and right subtrees, at every node of the tree
   b. the difference in the number of nodes in the left and right subtrees of the root node
   c. the absolute difference between the heights of the left and right subtrees, at every node of the tree
   d. the height of the left subtree minus the height of the right subtree of the root node
4. AVL trees
   a. track the number of nodes in the subtree below each node.
   b. use average values of the keys in a subtree to speed up the search time.

c. insert each item with height 1 and then adjust that height after modifications to its subtrees in later operations.

d. use the difference in height of a node's subtrees to determine which subtree will store the next item inserted.

5. If an empty AVL tree has 100 items inserted into it, and every insertion causes either an even balance or a temporary left-heavy tree at the root, you can conclude that

   a. the number of right rotations will equal or exceed the number of left rotations.

   b. the tree will be partially unbalanced leading the average number of comparisons for a search to be around 50.

   c. the last 50 items inserted will end up in the right subtree of the root.

   d. only right rotations will be needed to correct imbalances.

6. Deleting an item from an AVL tree

   a. is easiest when the node containing the item has two subtrees.

   b. uses either one or two rotations to rebalance subtrees based on the height balance of (1) the subtree and (2) the child with the larger height,

   c. uses fusion operations like 2-3-4 trees do in some situations.

   d. uses the mirror image operations of inserting an item.

7. True or False: The red-black rules are a sequence of steps that rearrange the nodes in a tree to balance it.

8. A null child is

   a. a child that doesn't exist but will be created next as a placeholder to balance the tree.

   b. a child with no children of its own.

   c. a child whose key was deleted but was left in the tree as a soft deletion.

   d. a nonexistent left child of a node with a right child (or vice versa).

9. Which of the following is ***not*** a red-black rule?

   a. Every path from a root to a leaf, or to a null child, must contain the same number of black nodes.

   b. If a node is black, its children must be red.

   c. The root node is always black.

   d. Every node must be either black or red.

10. The two possible actions used to balance a red-black tree are _____ and _____.

11. Newly inserted nodes are always colored _____.

12. A crossover node or subtree starts as a _____ and becomes a _____, or vice versa.

13. Which of the following is ***not*** true of red-black trees? Rotations may need to be made

    a. before a node is inserted.

    b. after a node is inserted.

    c. during a search for the insertion point.

    d. when searching for a node with a given key.

14. A color swap involves changing the color of _____ and _____.

15. An outside grandchild is

    a. on the opposite side of its parent than its parent is of its sibling.

    b. on the same side of its parent than its parent is of its parent.

    c. one which is the left descendant of a right descendant (or vice versa).

    d. on the opposite side of its parent than its sibling is of their grandparents.

16. True or False: When one rotation immediately follows another as part of single insertion at the lowest level of a red-black tree, they are in opposite directions.

17. Two rotations are necessary when

    a. the node is an inside grandchild and the parent is red.

    b. the node is an inside grandchild and the parent is black.

    c. the node is an outside grandchild and the parent is red.

    d. the node is an outside grandchild and the parent is black.

18. Two color swaps are necessary when

    a. encountering a black node with two red children on deletion.

    b. encountering a red node with two black children on insertion.

    c. encountering a black node with two red children on insertion.

    d. none of these situations.

19. In comparing the efficiency of AVL trees and red-black trees

    a. all operations in AVL trees are somewhat more efficient than those of red-black trees because they operate bottom-up.

    b. AVL trees use more memory per node and have slower search than red-black trees.

    c. red-black trees use fewer rotations to balance trees, making them faster at insertion, deletion, and search.

    d. both are O(log N) but the red-black rules can result in balanced trees with path lengths up to twice as long as those in an AVL tree, leading to slower search times in red-black trees.

20. The 2-3-4 trees (Chapter 9)

    a. can be collapsed into 2-3 trees (Chapter 9), and in some cases, into balanced binary trees (Chapter 10).

    b. can be transformed into AVL trees, but their balance must be adjusted through additional rotations.

    c. can be restructured using rotations into a red-black tree, although the balance of the red-black tree may be left-heavy or right-heavy depending on how 3-nodes are handled.

    d. can be mapped into equivalent red-black trees and the 4-node split operation has the same effect as a red-black color swap.

Experiments

Carrying out these experiments will help to provide insights into the topics covered in the chapter. No programming is involved.

10-A Tree structures have been used by secret organizations such as resistance movements for a long time. The tree structure represents the hierarchy of the group. Group members know their leader—their parent node—and their direct subordinates—their child nodes. Knowing more than a few of their members, however, can be dangerous to the organization. An enemy who discovers one member may be able to discover the other parts of the group known to that member.

Which tree structure that you've studied (Chapters 9 and 10) would be best as the basis for organizing such a group? How would balancing the tree help or hurt the organization? If it helps, which method(s) of balancing that you've seen would be best in terms of preventing an enemy from knowing the size and structure of the organization if they captured one member?

10-B If an AVL tree uses a single byte to represent the height at each node, the height can range from 0 to 255. A tree with heights deeper than that would cause problems. How many items (nodes) could such an AVL tree hold without running in to that problem?

10-C If you haven't already, perform all of the experiments in the sections: "Experiment 1: Inserting Two Red Nodes," "Experiment 2: Rotations," "Experiment 3: Color Swaps," and "Experiment 4: An Unbalanced Tree."

10-D Use the RedBlackTree Visualization tool and fill an empty tree with 31 random nodes. Repeat this 50 times. How often does the resulting tree satisfy all the red-black rules? How often is the tree grossly unbalanced (that is, have some node with a height difference of 3 or more)? How does this compare with randomly constructing similar trees in the BinaryTree Visualization tool?

10-E Do enough insertions to convince yourself that if red-black rules 1, 2, and 3 are followed exactly along with the color swaps and rotations described in this chapter, Rule 4 will take care of itself.

Programming Projects

Writing programs to solve the Programming Projects helps to solidify your understanding of the material and demonstrates how the chapter's concepts are applied. (As noted in the Introduction, qualified instructors may obtain completed solutions to the Programming Projects on the publisher's website.)

10.1 Write a method `isBalanced()` for the `AVLtree` class that verifies the entire tree is balanced. It should return `True` when the height difference of every node is –1, 0, or +1, and `False` otherwise. Show that the tree starts balanced when it's empty and remains balanced as nodes are inserted and later deleted.

For extra thoroughness, use Python's `itertools` package to find every permutation of the order of six distinct keys to insert into the empty tree. Count how many of those insertion permutations produce balanced trees.

10.2 Write a method `howManyWithin()` for the `AVLtree` class that counts the number of items with keys within a low and a high value. Because the AVL tree is a binary search tree, not all nodes need to be investigated. For example, when visiting a node whose key is 20 when the low value of the range is 40, there's no need to visit its left child branch because it cannot possibly have any keys in the range. The method should count keys that match the low or high value, if the high value is equal to or greater than the low (a non-empty range). Test your method on a variety of ranges including empty ranges and ones that match exactly one key in the tree.

10.3 Perhaps you have used sets in math. They are used to group things like the set of all prime numbers or the set of all multiples of 3. Another similar concept is a *multiset*. In the standard set, an element is either in or out of the set and repeat instances of the element cannot be in the set. The multiset allows multiple instances of the same element in a set. The multiset tracks the count of each of its elements; if the count becomes zero, the element is no longer in the set. This same idea has been called *bag*, *aggregate*, *weighted set*, and other names in various places. It could be used, for example, to represent the coins in a bag or the cards in a player's hand in a game that has multiple copies of the same card. Python has a built-in `set` data type but not a specific multiset.

Use the `AVLtree` to implement a `Multiset` class and perform the basic set operations. The keys of the AVL tree are the elements of the multiset and their corresponding values store the count of that element. This allows for multisets of strings, integers, tuples, lists, and any other data type that can be ordered.

Your implementation of `Multiset` should include the following methods:

- `__len__()`—Returns the number of keys (distinct elements) stored in the multiset and allows the Python `len()` function to be used on multisets.
- `cardinality()`—Returns the total count of all elements stored in the multiset.
- `__str__()`—Returns a string representation of the multiset showing the keys and their counts, for example `"[(A: 4), (P: 1)]"`. This allows the multiset to be printed or passed to the Python `str()` function.

- `__contains__(key)` —Returns `True` if the given key is in the multiset and `False` otherwise, allowing the multiset to be used in an expression such as `key in multiset`.
- `count(key)` —Returns the count of the given key in the multiset or zero if the key is not in the multiset.
- `add(key, count=1)` —Add a key to the multiset with an optional `count` parameter that defaults to 1, allowing multiple copies to be added in one call. Another way to describe this method is that it increases the count for the given key by the `count` parameter.
- `remove(key, count=1)` —Remove a certain number of copies of key from the multiset. The number of copies is an optional parameter that defaults to 1. If the number of copies being removed is equal to or higher than the current count for the key, the key should be removed from the underlying AVL tree.

Show your implementation working on an empty multiset, adding some items, showing the string representing the multiset, showing the length and cardinality for the whole multiset, showing the count for individual keys, then deleting some elements, and then showing more counts after the deletions.

10.4 Extend the `Multiset` implementation from Programming Project 10.3 to add the `union()` and `intersection()` methods. The union of two multisets is like the union of a regular set except that the count for an element in the union is the maximum of its count in the input multisets. The intersection of two multisets has counts that are the minimum of the counts for the element in the input multisets. For both operations, elements that are not in a multiset have a count of zero.

The `union()` method of `Multiset` should take another multiset as input and return a new multiset containing the union. It's more difficult to write it as a method that modifies one of the input multisets because you must traverse the underlying AVL tree while modifying it. Similarly, the `intersection()` method should take another multiset as input and returns a new multiset containing the intersection.

To perform the union and intersection operations, you need to traverse the items in the underlying AVL trees. For this, you can look at the `traverse()` generator used for 2-3-4 trees in Chapter 9 or the code provided for the AVL tree with this textbook. To perform the operations efficiently, you can generate iterators for both multisets and visit all the elements in order. This operation is somewhat like mergesort, which takes two sorted sequences and produces a new combined sequence in sorted order. At each step, the algorithm looks at the element with the lowest key in the two sequences and determines what should go in the output multiset.

Show your implementation of union and intersection by adding a variety of elements to two multisets, ensuring some elements are in both sets while others are only in one but not the other. Also ensure that the counts of some of the common elements are different in the two multisets. Show the union and intersection multisets formed by combining those two along with their length and cardinality.

10.5 Measure the performance of the AVL tree by altering its structure to store and report statistics on the number of operations performed. In the constructor for an `AVLtree`, create storage for the number of calls to `updateheight()`, `heightdiff()`, `rotateRight()`, and `rotateLeft()` that are all initially zero. Add methods to get the values for these statistics and to clear them back to zero. Write a `__len__()` method that gets the count of the number of items in the `AVLtree` and a `height()` method that gets the height of the root node.

Use the new methods to get the absolute counts for each of the statistics as you insert 100 or more items and then delete them. Show the counts for insertions and deletions and the counts divided by the log of the size of the tree; that is, divide by `math.log(len(tree))`, when the size is four or above to always have a positive denominator and at least two levels in the tree. It's helpful to print the statistics in fixed width columns (see Python's `format` function for strings) so they look something like:

```
N       H updHgt hgtDif RotLft RotRgt updHgt hgtDif RotLft RotRgt
84      7      6      5      0      1  1.354  1.128  0.000  0.226
```

The first row shows the abbreviated titles of the statistics. It shows the number of nodes, N; the height of the tree, H; the four raw statistics; and the four ratios of those counts to the log N value. Print out about 15 samples of these statistics.

If the efficiency of the operations is O(log N), then the ratios should be bounded and relatively stable. They will be largest when there are the most rotations needed to rebalance the tree. Keep track of the maximum ratios and print them after all the items have been inserted and then again after all have been deleted. The ratios should remain bounded even if you increase the number of items to a thousand.

In fact, the worst ratios occur when the tree is small because the denominator is the smallest. For example, look at the subtree rooted at node 72 in Figure 10-5. If you forget about the rest of the tree and assume node 72 is the root, then it is a balanced AVL tree because all 12 of its nodes have a height difference of one or less. If you then delete node 69 from that tree, the left side becomes unbalanced. It requires two rotations to balance the left subtree because the inside grandchild must be promoted. The root is now unbalanced with a left side of height 2 and a right side of height 4. That requires a left rotation to fix, and it requires a further right rotation because the right subtree is left heavy. That's four rotations total, which each cause two calls to `updateHeight()`. There are two more calls to `updateHeight()` from the descent down to the node to delete, node 69, which leaves a total of 10 calls. The number of calls divided by the log of the number of nodes, 12 (before the deletion), is a little more that 4.0 when using the `math.log()` function, which is what's called the natural logarithm. Your results should show ratios less than 4 for larger trees.

This page intentionally left blank

CHAPTER 11

Hash Tables

IN THIS CHAPTER

- Introduction to Hashing
- Open Addressing
- Separate Chaining
- Hash Functions
- Hashing Efficiency
- Hashing and External Storage

A **hash table** is a data structure that offers very fast insertion and searching. When you first hear about them, hash tables sound almost too good to be true. No matter how many data items there are, insertion and searching (and sometimes deletion) can take close to constant time: O(1) in Big O notation. In practice this is just a few machine instructions.

For a human user of a hash table, this amount of time is essentially instantaneous. It's so fast that computer programs typically use hash tables when they need to look up hundreds of thousands of items in less than a second (as in spell checking or in auto-completion). Hash tables are significantly faster than trees, which, as you learned in the preceding chapters, operate in relatively fast O(log N) time. Not only are they fast, hash tables are relatively easy to program.

Despite these amazing features, hash tables have several disadvantages. They're based on arrays, and expanding arrays after they've been allocated can cause challenges. If there will be many deletions after inserting many items, there can be significant amounts of unused memory. For some kinds of hash tables, performance may degrade catastrophically when a table becomes too full, so programmers need to have a fairly accurate idea of how many data items will be stored (or be prepared to periodically transfer data to a larger hash table, a time-consuming process).

Also, there's no convenient way to visit the items in a hash table in any kind of order (such as from smallest to largest). If you need this kind of traversal, you'll need to look elsewhere.

However, if you don't need to visit items in order, and you can predict in advance the size of your database or accept

some extra memory usage and a tiny bit of slowness as the database is built up, hash tables are unparalleled in speed and convenience.

Introduction to Hashing

In this section we introduce hash tables and hashing. The most important concept is how a range of key values is transformed into a range of array index values. In a hash table, this transformation is accomplished with a **hash function**. For certain kinds of keys, however, no hash function is necessary; the key values can be used directly as array indices. Let's look at this simpler situation first and then go on to look at how hash functions can be used when keys aren't distributed in such an orderly fashion.

Bank Account Numbers as Keys

Suppose you're writing a program to access the bank accounts of a small bank. Let's say the bank is fairly new and has only 10,000 accounts. Each account record requires 1,000 bytes of storage. Thus, you can store the entire database in only 10 megabytes, which will easily fit in your computer's memory.

The bank director has specified that she wants the fastest possible access to any individual record. Also, every account has been given a number from 0 (for the first account created) to 9,999 (for the most recently created one). These account numbers can be used as keys to access the records; in fact, access by other keys is deemed unnecessary. Accounts are seldom closed, but even when they are, their records remain in the database for reference (to answer questions about past activity). What sort of data structure should you use in this situation?

Index Numbers as Keys

One possibility is a simple array. Each account record occupies one cell of the array, and the index number of the cell is the account number for that record. This type of array is shown in Figure 11-1.

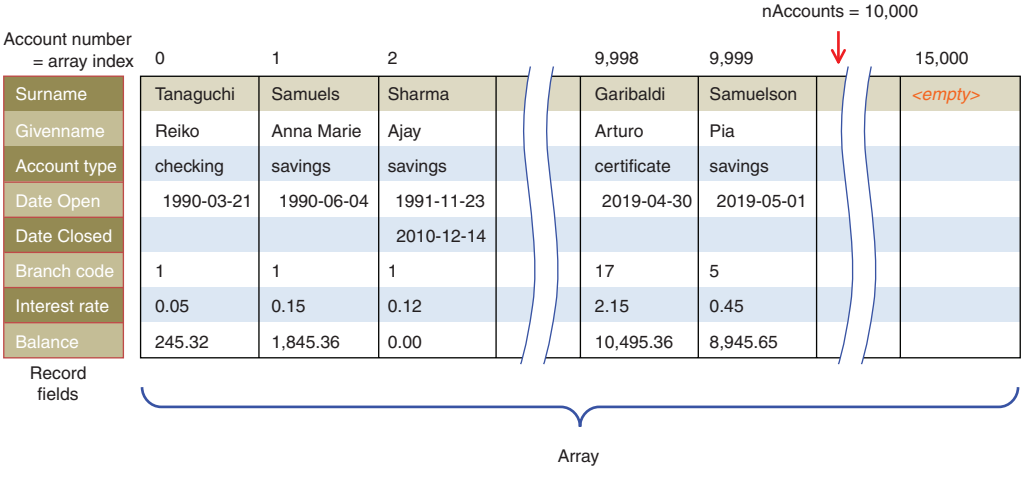


FIGURE 11-1 Account numbers as array indices

As you know, accessing a specified array element is very fast if you know its index number. The clerk looking up what account a check is drawn from knows that it comes from, say, number 72, so he enters that number, and the program goes instantly to index number 72 in the array. A single program statement is all that's necessary:

```
accountRec = databaseArray[72]
```

Adding a new account is also very quick: you insert it just past the last occupied element. If there are currently 9,300 accounts, the next new record would go in cell 9,300. Again, a single statement inserts the new record:

```
databaseArray[nAccounts] = newAccountRecord()
```

The count of the number of accounts would be incremented like this:

```
nAccounts += 1
```

Presumably, the array is made somewhat larger than the current number of accounts, to allow room for expansion, but not much expansion is anticipated, or at least it needs to be done only infrequently, such as once a month.

Not Always So Orderly

The speed and simplicity of data access using this array-based database make it very attractive. This example, however, works only because the keys are unusually well organized. They run sequentially from 0 to a known maximum, and this maximum is a reasonable size for an array. There are no deletions, so memory-wasting gaps don't develop in the sequence. New items can be added sequentially at the end of the array, and the array doesn't need to be much larger than the current number of items.

A Dictionary

In many situations the keys are not so well behaved, as in the bank account database just described. The classic example is a dictionary. If you want to put every word of an English-language dictionary, from *a* to *zyzzyva* (yes, it's a word), into your computer's memory so they can be accessed quickly, a hash table is a good choice.

A similar widely used application for hash tables is in computer-language compilers, which maintain a **symbol table** in a hash table (although balanced binary trees are sometimes used). The symbol table holds all the variable and function names made up by the programmers, along with the address (or register) where they can be found in memory. The program needs to access these names very quickly, so a hash table is the preferred data structure.

Coming back to natural languages, let's say you want to store a 50,000-word English-language dictionary in main memory. You would like every word to occupy its own cell in a 50,000-cell array, so you can access the word's record (with definitions, parts of speech, etymology, and so on) using an index number. This approach makes access very fast, but what's the relationship of these index numbers to the words? Given the word *ambiguous*, for example, how do you find its index number?

Converting Words to Numbers

What you need is a system for turning a word into an appropriate index number. To begin, you know that computers use various schemes for representing individual characters as numbers. One such scheme is the ASCII code, in which *a* is 97, *b* is 98, and so on, up to 122 for *z*.

The extended ASCII code runs from 0 to 255, to accommodate capitals, punctuation, accents, symbols, and so on. There are only 26 letters in English words, so let's devise our own code, a simpler one that can potentially save memory space. Let's say *a* is 1, *b* is 2, *c* is 3, and so on up to 26 for *z*. We'll also say a blank—the space character—is 0, so we have 27 characters. (Uppercase letters, digits, punctuation, and other characters aren't used in this dictionary.)

How could we combine the digits from individual letter codes into a number that represents an entire word? There are all sorts of approaches. We'll look at two representative ones, and their advantages and disadvantages.

Adding the Digits

A simple approach to converting a word to a number might be to simply add the code numbers for each character. Say you want to convert the word *elf* to a number. First, you convert the characters to digits using our homemade code:

$$e = 5 \quad l = 12 \quad f = 6$$

Then you add them:

$$5 + 12 + 6 = 23$$

Thus, in your dictionary the word *elf* would be stored in the array cell with index 23. All the other English words would likewise be assigned an array index calculated by this process.

How well would this approach work? For the sake of argument, let's restrict ourselves to 10-letter words. Then (remembering that a blank is 0), the first word in the dictionary, *a*, would be coded by

$$0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 1 = 1$$

The last potential word in the dictionary would be *zzzzzzzzzz* (10 letter *z*'s). The code obtained by adding its letters would be

$$26 + 26 + 26 + 26 + 26 + 26 + 26 + 26 + 26 + 26 = 260$$

Thus, the total range of word codes is from 1 to 260 (assuming a string of all spaces is not a word). Unfortunately, there are 50,000 words in the dictionary, so there aren't enough index numbers to go around. If each array element could hold about 192 words (50,000 divided by 260), then you might be able fit them all in, but how would you distinguish among the 192 words in one array element?

Clearly, this coding presents problems if you're thinking in terms of the one word-per-array element scheme. Maybe you could put a subarray or linked list of words at each array element. Unfortunately, such an approach would seriously degrade the access speed.

Accessing the array element would be quick, but searching through the 192 words to find the one you wanted could be very slow.

So this first attempt at converting words to numbers leaves something to be desired. Too many words have the same index. Certainly, any anagram of a word would have the same code because the order of the letters doesn't change the value. In addition, these words

acne *ago* *aim* *baked* *cable* *hack*

and dozens of other words have letters that add to 23, as *elf* does. For words with higher codes, there could be hundreds of other matching words. It is now clear that this approach doesn't discriminate enough, so the resulting array has too few elements. We need to spread out the range of possible indices.

Multiplying by Powers

Let's try a different way to map words to numbers. If the array was too small before, make sure it's big enough. What would happen if you created an array in which every word—in fact, every potential word, from *a* to *zzzzzzzzzz*—was guaranteed to occupy its own unique array element?

To do this, you need to be sure that every character in a word contributes in a unique way to the final number.

You can begin by thinking about an analogous situation with numbers instead of words. Recall that in an ordinary multidigit number, each digit-position represents a value 10 times as big as the position to its right. Thus 7,546 really means

$$\mathbf{7}*1{,}000 + \mathbf{5}*100 + \mathbf{4}*10 + \mathbf{6}*1$$

Or, writing the multipliers as powers of 10:

$$\mathbf{7}*10^3 + \mathbf{5}*10^2 + \mathbf{4}*10^1 + \mathbf{6}*10^0$$

In this system, you break a number into its digits, multiply them by appropriate powers of 10 (because there are 10 possible digits), and add the products. If this happened to be an octal number using the digits from 0 to 7, then you would get $\mathbf{7}*8^3 + \mathbf{5}*8^2 + \mathbf{4}*8^1 + \mathbf{6}*8^0$.

In a similar way, you can decompose a word into its letters, convert the letters to their numerical equivalents, multiply them by appropriate powers of 27 (because there are 27 possible characters, including the blank), and add the results. This approach gives a unique number for every word.

Let's return to the example of converting the word *elf* to a number. You convert the digits to numbers as shown earlier. Then you multiply each number by the appropriate power of 27 and add the results:

$$\mathbf{5}*27^2 + \mathbf{12}*27^1 + \mathbf{6}*27^0$$

Calculating the powers gives

$$\mathbf{5}*729 + \mathbf{12}*27 + \mathbf{6}*1$$

and multiplying the letter codes times the powers yields

$$3{,}645 + 324 + 6$$

which sums to 3,975.

This process does indeed generate a unique number for every potential word. You just calculated a 3-letter word. What happens with larger words? Unfortunately, the range of numbers becomes rather large. The largest 10-letter word, *zzzzzzzzzz*, translates into

$$26*27^9 + 26*27^8 + 26*27^7 + 26*27^6 + 26*27^5 + 26*27^4 + 26*27^3 + 26*27^2 + 26*27^1 + 26*27^0$$

Just by itself, 27^9 is more than 7,000,000,000,000, so you can see that the sum will be huge. An array stored in memory can't possibly have this many elements, except perhaps, in some huge supercomputer. Even if it could fit, it would be very wasteful to use all that memory to store a dictionary of just 50,000 words.

The problem is that this scheme assigns an array element to every potential word, whether it's an actual English word or not. Thus, there are cells reserved for *aaaaaaaaaa*, *aaaaaaaaab*, *aaaaaaaaac*, and so on, up to *zzzzzzzzzz*. Only a small fraction of these cells is necessary for real words, so most array cells are empty. This situation is illustrated in Figure 11-2. Near the word *elf*, there are several words that would be stored, such as *elk*, *eli* (for the given name *Eli*), and *elm*. The red arrows indicate a pointer to a record describing the word. At other places, such as around the word *bird*, there would be many unused cells, indicated by the cells without a pointer to some other structure.

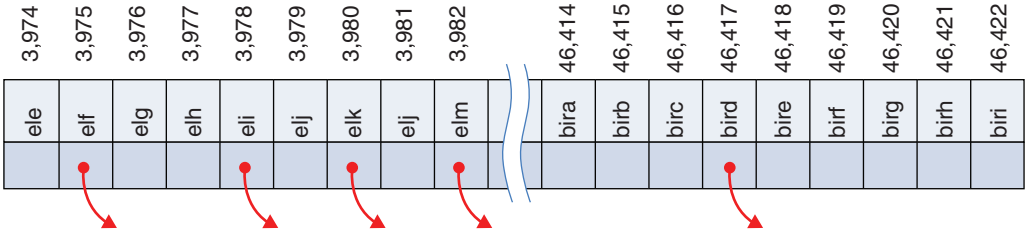


FIGURE 11-2 Index for every potential word

The first scheme—adding the numbers—generated too few indices. This latest scheme—adding the numbers times powers of 27—generates too many.

Hashing

What we need is a way to compress the huge range of numbers obtained from the numbers-multiplied-by-powers system into a range that matches a reasonably sized array.

How big an array are we talking about for this English dictionary? If you have only 50,000 words, you might assume the array should have approximately this many elements. It's preferable, however, to have extra cells, rather than too few, so you can shoot for an array

twice that size, 100,000 cells. We discuss the advantages to having twice the minimum amount needed a little later.

Thus, we seek a way to squeeze a range of 0 to more than 7 trillion into the range 0 to 100,000. A simple approach is to use the modulo operator (%), which finds the remainder when one number is divided by another.

To see how this approach works, let's look at a smaller and more comprehensible range. Suppose you are trying to squeeze numbers in the range 0 to 199 into the range 0 to 9. The range of the big numbers is 200, whereas the smaller range has only 10. If you want to convert a big number (stored in a variable called `largeNumber`) into the smaller range (and store it in the variable `smallNumber`), you could use the following assignment, where `smallRange` has the value 10:

```
smallNumber = largeNumber % smallRange
```

The remainders when any number is divided by 10 are always in the range 0 to 9; for example, 13 % 10 gives 3, and 157 % 10 is 7. With decimal numbers, it simply means getting the last digit. The modulo operation compresses (or **folds**) a large range into a smaller one, as shown in Figure 11-3. In our toy example, we're squeezing the range 0–199 into the range 0–9, which is a 20-to-1 compression ratio.

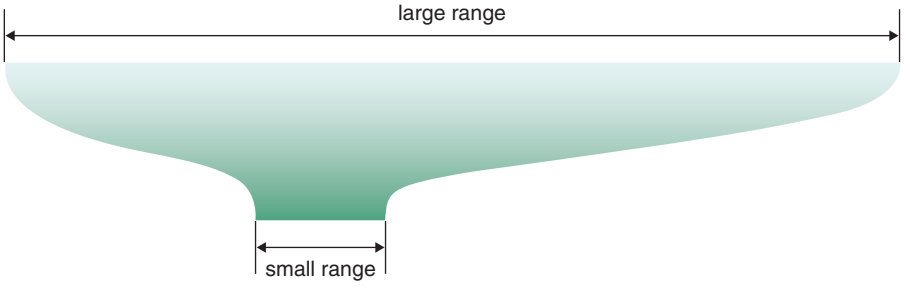


FIGURE 11-3 Range conversion using modulo

A similar expression can be used to compress the really huge numbers that uniquely represent every English word into index numbers that fit in the dictionary array:

```
arrayIndex = hugeNumber % arraySize
```

The function that computes the `hugeNumber` is an example of a **hash function**. It **hashes** (converts) a string (or some other piece of data) into a number in a large range. Taking the modulo with the `arraySize` maps the number into a smaller range. This smaller range is the range of index numbers in an array. The array into which the hashed data is inserted is called a **hash table**. The index to which a particular key maps is called the **hash address**. This terminology can be a little confusing. We use the term *hash table* to describe both the whole data structure and the array inside it that holds items.

To review: you can convert (or **encode**) a word into a huge number by converting each letter in the word into an integer and then multiplying them by an appropriate power of 27, based on their position in the word. Listing 11-1 shows some Python code that computes the huge number.

LISTING 11-1 Functions to Uniquely Encode Simple English Words as Integers

```
def encode_letter(letter):          # Encode letters a thru z as 1 thru 26
   letter = letter.lower()          # Treat uppercase as lowercase
   if 'a' <= letter and letter <= 'z':
      return ord(letter) - ord('a') + 1
   return 0                         # Spaces and everything else are 0

def unique_encode_word_loop(word): # Encode a word uniquely using
   total = 0                       # a loop to sum the letter codes times
   for i in range(len(word)):      # a power of 27 based on their position
      total += encode_letter(word[i]) * 27 ** (len(word) - 1 - i)
   return total

def unique_encode_word(word):       # Encode a word uniquely (abbreviated)
   return sum(encode_letter(word[i]) * 27 ** (len(word) - 1 - i)
              for i in range(len(word)))
```

The `encode_letter()` function takes a letter, gets its lowercase version, and checks whether it is in the range of `'a'` to `'z'`, inclusive. If it is, it converts the letter to an integer by using Python's built-in `ord()` function. This function returns the Unicode value (also called *point*) of the character, which is the same as the ASCII value for the English letters. It returns the value of the character relative to the `'a'` character ensuring that `'a'` returns a value of 1. For characters outside the range `'a'` to `'z'`, it returns 0. That means that space is encoded as 0, as well as every other Unicode character that's not in the range.

To get the unique numeric code for a word, you can use a loop to sum up the values for each letter. The `unique_encode_word_loop()` function uses an index, `i`, into the letters of its `word` parameter to extract each one, get its encoded value using `encode_letter()`, multiply that value with a power of 27 appropriate for its position, and add the product to the running `total`. The power of 27 should be 0 for the last character of the `word`, which has the index `len(word) - 1`. For the second-to-last character at `len(word) - 2`, the exponent expression would be 1. The third-to-last would be exponent 2, and so on, up to exponent `len(word) - 1` for the first character (leftmost) in the word. After the loop exits, the total is returned.

Listing 11-1 also shows a `unique_encode_word()` function that computes the exact same encoded value. It calculates it, however, using a more compact syntax with a list

comprehension. The `sum()` function returns the sum of its arguments. The list (tuple) comprehension provides the arguments to `sum()`. Comprehensions are in the form

expression `for` *variable* `in` *sequence*

and in the `unique_encode_word()` function, `i` is used as the index *variable* that comes from the comprehension *sequence* (which are the indices of letters in `word`). The *expression* is the same as what was used in the loop version.

The `unique_encode_word()` function is an example of a hash function. Using the modulo operator (%), you can squeeze the resulting huge range of numbers into a range about twice as big as the number of items you want to store. This computes a hash address:

```
arraySize = numberWords * 2
arrayIndex = unique_encode_word(word) % arraySize
```

In the huge range, each number represents a potential data item (an arrangement of letters), but few of these numbers represent actual data items (English words). A hash address is a mapping from these large numbers into the index numbers of a much smaller array. In this array, you can expect that, on the average, there will be one word for every two cells. Some cells will have no words, some will have one, and there can be others that have more than one. How should that be handled?

Collisions

We pay a price for squeezing a large range into a small one. There's no longer a guarantee that two words won't hash to the same array index.

This is similar to the problem you saw when the hashing scheme was the sum of the letter codes, but the situation is nowhere near as bad. When you added the letter codes, there were only 260 possible numeric values (for words up to 10 letters). Now you're spreading the codes over the 100,000 possible array cells.

It's impossible to avoid hashing several different words into the same array location, at least occasionally. The plan was to have one data item per index number, but this turns out not to be possible in most hash tables. The best you can do is configure things so that not too many words will hash to the same index.

Perhaps you want to insert the word *abductor* into the array. You hash the word to obtain its index number but find that the cell at that number is already occupied by the word *bring*, which happens to hash to the exact same number. This situation, shown in Figure 11-4, is called a **collision**. The word *bring* has a unique code of 1,424,122, which is converted to 24,122 by taking the modulo with 100,000. The word *abductor* has the unique code 11,303,824,122, and *missable* has 139,754,124,122. All three of them hash to index 24,122 of the hash table.

The slightly different words *brine* and *brink* hash to locations nearby the cell for *bring*. The reason is that they differ only in their last letter, and that letter's code is multiplied by 27^0, or 1. Other words could also hash to those same locations.

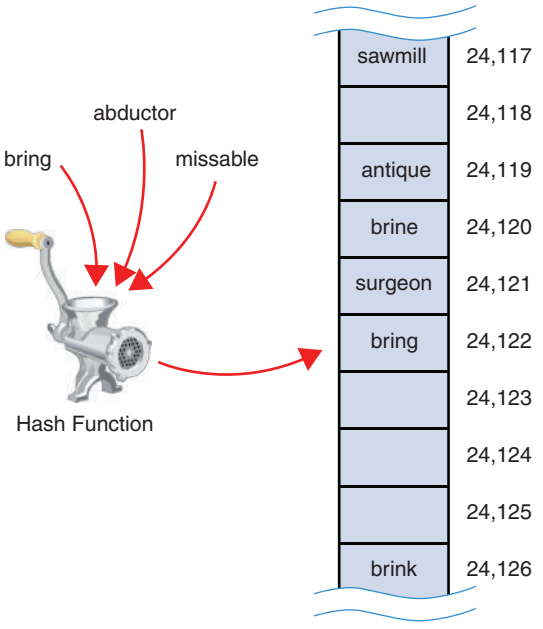


FIGURE 11-4 A collision

How Bad Are Collisions?

Is the hash function a good idea? Could running strings of letters through a "grinder" ever produce some kind of useful hash? It may appear that the possibility of collisions renders the scheme impractical. It would help to know how often they are likely to occur in designing strategies to deal with them.

One relevant measure can be seen by answering a classic question: when is it more likely than not that two people at a gathering share the same birth day and month? Figure 11-5 illustrates the concept. At first, that idea might not seem relevant to hash tables. On closer inspection, you can think of the days in a year as the cells of a hash table. The combined day and month form 366 uniique indices. Each birthday lands in exactly one of them.

If there are only a few people at the gathering, it's highly unlikely that they share a birth month and day. If there are 367 people, then it is certain that some have the same birth month and day. Somewhere between those extremes, there's a number of people where the likelihood of a shared month and day is greater than the likelihood that they are all different.

Maybe intuition tells you that if you have half as many people as there are days in the year, then the likelihood would be greater than 50 percent for a shared birth month and day. In other words, if there are 183 people, it is more likely than not to have a shared birth month and day. That intuition is correct, but the point at which it changes from

below 50 percent to above 50 percent is at 23 people. With 22 people, it's still more likely they all were born on different days. This calculation assumes that the birth days are distributed randomly throughout the year (which is not the case). At a gathering for people born under a particular sign of the zodiac, of course, the distribution would be very different!

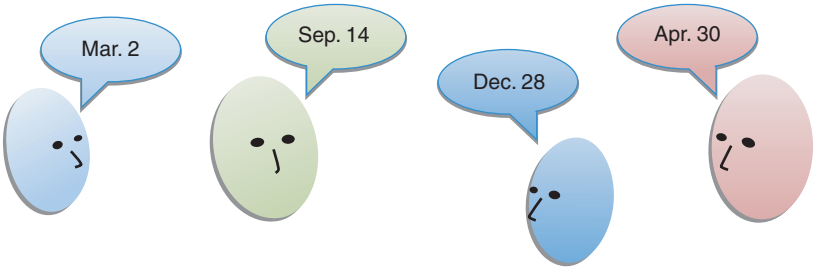


FIGURE 11-5 Finding shared birthdays at a gathering

So even when the ratio of items to hash table cells is less than 10 percent (23 out of 366), the chance of a collision is greater than 50 percent. That means you should plan your hash tables to always deal with collisions. You can work around the problem in a variety of ways.

We already mentioned the first technique: specifying an array with at least twice as many cells as data items. This means you expect half the cells to be empty. One approach, when a collision occurs, is to search the array in some systematic way for an empty cell and insert the new item there, instead of at the index specified by the hash address. This approach is called **open addressing**. It's somewhat like boarding a train or subway car; you enter at one point and take the nearest seat that's open. If the seats are full, you continue through the car until you find an empty seat. The seat closest to the door where you entered is analogous to the initial hash address.

Returning to hashing words into numbers, if *abductor* hashes to 24,122, but this location is already occupied by *bring*, then you might try to insert *abductor* in cell 24,123, for example. When the insert operation finds an empty cell, it stores *both the key and its associated value*. In that way, search operations using open addressing can compare the original keys to the keys stored in the table to determine how far the search should continue. It also enables easy checking for empty cells because they won't have a key-value structure.

A second approach (mentioned earlier) is to create an array that consists of references to another data structure (like linked lists of words) instead of the records for the individual words. Then, when a collision occurs, the new item is simply inserted in the list at that index. This is called **separate chaining**.

In the balance of this chapter, we discuss open addressing and separate chaining, and then return to the question of hash functions.

So far, we've focused on hashing strings. In practice, many hash tables are used for storing strings. Hashing by birthdays is certainly possible, but only useful in rare instances. Many other hash tables are keyed by numbers, as in the bank account number example, or in the case of credit card numbers. In the discussion that follows, we use numbers—rather than strings—as keys. This approach makes things easier to understand and simplifies the programming examples. Keep in mind, however, that in many situations these numbers would be derived from strings or byte sequences.

Open Addressing

In open addressing, when a data item can't be placed at the index calculated by the hash address, another location in the array is sought. We explore three methods of open addressing, which vary in the method used to find the next vacant cell. These methods are *linear probing*, *quadratic probing*, and *double hashing*.

Linear Probing

In **linear probing,** the algorithm searches sequentially for vacant cells. If cell 5,421 is occupied when it tries to insert a data item there, it goes to 5,422, then 5,423, and so on, incrementing the index until it finds an empty cell. This operation is called linear probing because it steps sequentially along the line of cells.

The HashTableOpenAddressing Visualization Tool

The HashTableOpenAddressing Visualization tool demonstrates linear probing. When you start this tool, you see a screen like that in Figure 11-6.

In this tool, keys can be numbers or strings up to 8 digits or characters. The initial size of the array is 2. The hash function has to squeeze the range of keys down to match the array size. It does this with the modulo operator (%), as you've seen before:

```
arrayIndex = key % arraySize
```

For the initial array size of 2, this is

```
arrayIndex = key % 2
```

This hash function is simple, so you can predict what cell will be indexed. If you provide a numeric key to one of the operations, the key hashes to itself, and the modulo 2 operation produces either array index 0 or 1. For string keys (anything that contains characters other than the decimal digits), it behaves similar to the `unique_encode_word()` function shown in Listing 11-1. For example, the key *cat* hashes to 7627107, which produces index 1. The key *bat* hashes to one less, 7627106, which produces index 0.

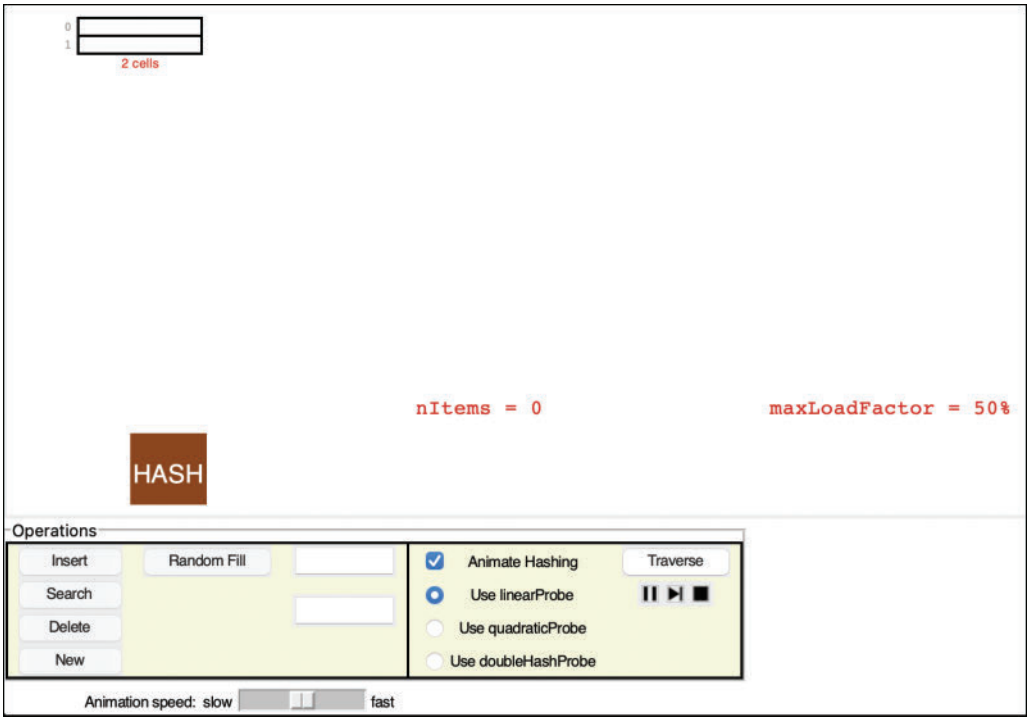


FIGURE 11-6 The HashTableOpenAddressing Visualization tool at startup

A two-cell hash table can't hold much data, obviously, and soon you'll see what happens as it begins to fill up. The number of cells is shown directly below the last cell of the table, and the cell indices appear to the left of each cell. The current number of items stored in the hash table, `nItems`, is shown in the center.

The box labeled "HASH" represents the hashing function. Let's see how new keys are processed by it and used to find array indices.

The Insert Button

To put some data in the hash table, type a key, *cat* for example, in the top text entry box and select Insert. The Visualization tool shows the process of passing the string `"cat"` through the hashing function to get a big integer. Using the modulo of the table size, it determines a cell index and draws an arrow connecting the hashed result to it like the one shown in Figure 11-7. The arrow points to where probing will begin to find an empty cell. Because the table is initially empty, the first probe finds an empty cell, and the insert operation finishes by copying the key into it—along with a colored background representing some associated data—and then incrementing the `nItems` value.

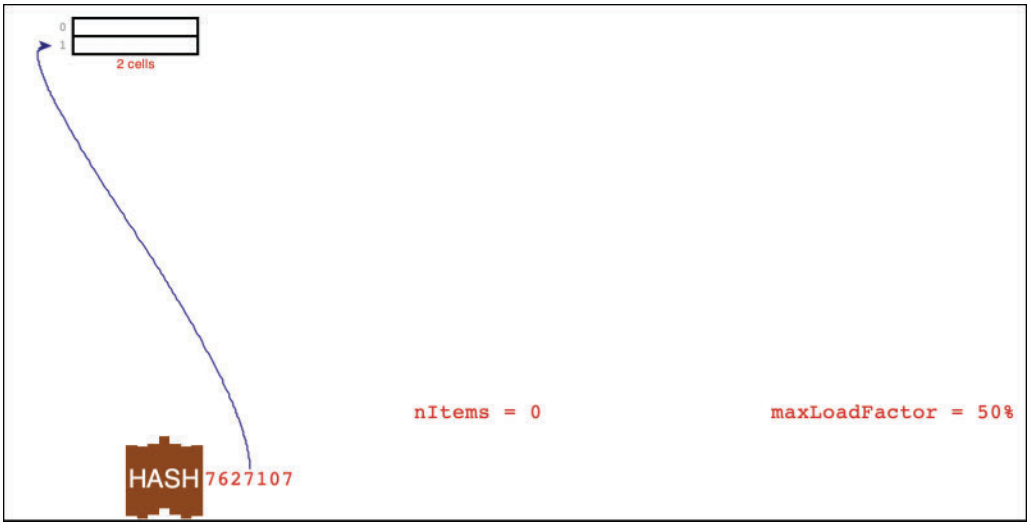


FIGURE 11-7 Probing to insert the key ***cat*** in an empty hash table

The next item inserted can show what happens in a collision. Inserting the key *eat* causes a hash address of 7627109, which probes cell 1, as shown in Figure 11-8.

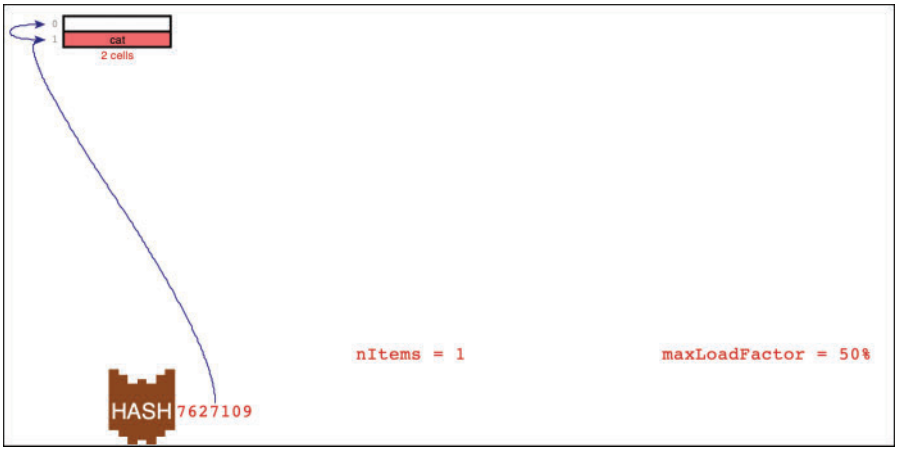


FIGURE 11-8 Probing to insert the key ***eat***

After finding cell 1 occupied, the insertion process begins probing each cell sequentially—linear probing—to find an empty cell, as shown with the additional curved arrow in Figure 11-8. The probing would normally start at index 2, but because that index lies beyond the end of the table, it wraps around to index 0. Because cell 0 is empty, the key *eat* can be stored there along with its associated data.

After incrementing the `nItems` value to 2, the table is now full. To be able to add more items in the future, the Visualization tool shows what happens next. A new table is allocated that is at least twice as big. The items from the old table are then reinserted in the new table by **rehashing** them. The hashing function hasn't changed, nor have the keys, so it might appear that the items would end up in their same relative positions. Because the size of the table grew, however, the modulo operator produces new cell indices. The *cat* and *eat* keys end up in cells 3 and 5 this time, as shown in Figure 11-9.

FIGURE 11-9 After inserting ***cat*** and ***eat*** in an empty hash table

We explore the details of this process in the "Growing Hash Tables" and "Rehashing" sections later. First, let's explore more about the Visualization tool and linear probing.

The Random Fill Button

Initially, the hash table starts empty and grows as needed. To explore what happens when larger tables become congested, you can fill them with a specified number of data items using the Random Fill button. Try entering `2` in the text entry box and selecting Random Fill. The Visualization tool generates two random strings of characters as keys and animates the process of inserting them.

The animation process takes some time, and when you understand how the insertions work, it may be preferable to jump right to the end result. If you uncheck the button labeled Animate Hashing, the Random Fill operation will perform all the insertions without animation. Similarly, single-item inserts will skip the animation of hashing the key (but not of the probing that happens afterward). Try disabling the animation and inserting 11 more items. You'll see that as the table grows, it divides into multiple columns, as shown in the example of Figure 11-10.

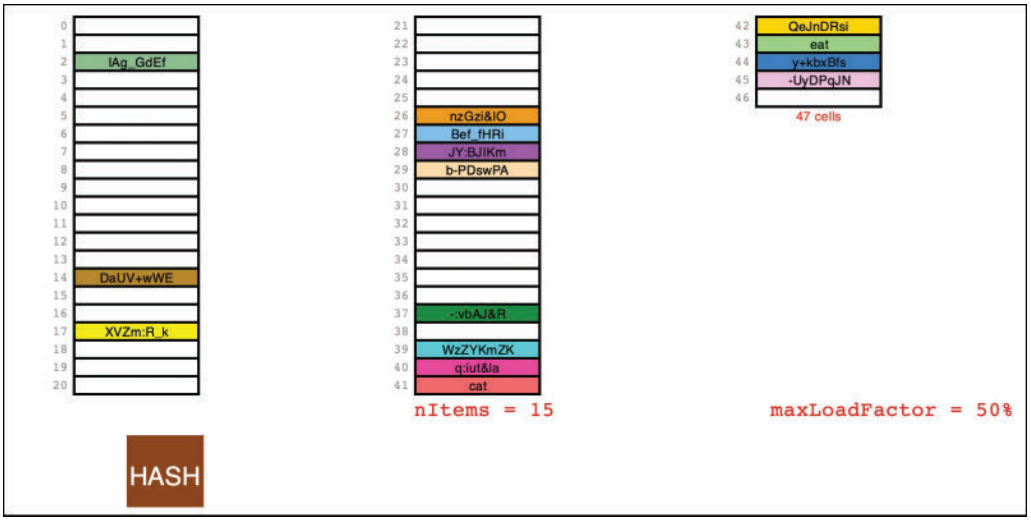


FIGURE 11-10 A hash table with 15 items

The Search Button

To locate items within the hash table, you enter the key of the item and select the Search button. If the Animate Hashing button is checked, the tool animates the conversion of the key string to a large number. The probing of the table begins with the index determined from the hashed key. If it finds the cell filled and the key matches, the key and the color representing its data are copied to an output box.

The Visualization tool simplifies searching for randomly generated and other existing keys by copying the key to the text entry box when a stored key is clicked. The search behavior gets a little more complex when the key isn't in the table. The tool uses a hashing function that treats numeric keys specially: they hash to their numeric value. Try typing `3` for the key (or clicking the index of another empty cell of a table like the one in Figure 11-10) and selecting Search. The initial probe lands on an empty cell, and the tool immediately discovers that the item is not in the table.

Now try entering the index of a filled cell, like `14` in Figure 11-10. You can also click the index number, but be sure that the key is the numeric index and not the string key stored in the cell. When you select Search, the Visualization tool shows the initial probe going to the selected index. Finding the cell full, but not containing the desired key, it starts linear probing to see whether a collision happened when the item was inserted. The next empty cell probed ends the search.

Filled Sequences and Clusters

As you might expect, some hash tables have items evenly distributed throughout the cells, and others don't. Sometimes there's a sequence of several empty cells and sometimes a sequence of filled cells. In the example of Figure 11-10, the filled sequences comprise four 1-item sequences, one 4-item sequence, and one 7-item sequence.

Let's call a sequence of filled cells in a hash table a **filled sequence**. As you add more and more items, the filled sequences become longer. This phenomenon is called **clustering** and is illustrated in Figure 11-11. Note that the order that items were inserted into the table determines how far away a key is placed relative to its default location.

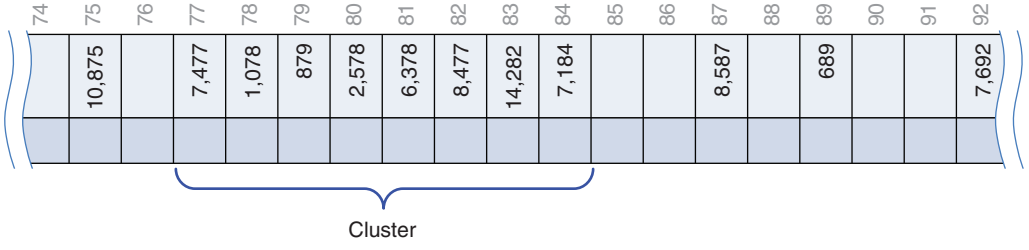


FIGURE 11-11 An example of clustering in linear addressing

When you're searching for a key, it's possible that the first indexed cell is already occupied by a data item with some other key. This is a collision; you see the visualization tool add another arrow pointing to the next cell. The process of finding an appropriate cell while handling collisions is called **probing**.

Following a collision, the hash table's search algorithm simply steps along the array looking at each cell in sequence. If it encounters an empty cell before finding the goal key, it knows the search has failed. There's no use looking further because the insertion algorithm would have inserted the item at this cell (if not earlier). Figure 11-12 shows successful and unsuccessful linear probes in a simplified hash table. By *simplified*, we mean that it uses the last two digits of the key as the table index, which is not a good idea in practice. (You see why a little later.) The initial probe for key 6,378 lands at cell 78. It probes the next adjacent cells until it finds the matching key in cell 81. The search for key 478 also starts at cell 78. After probing 7 cells in the filled sequence, it finds an empty cell at index 85, which ends the search.

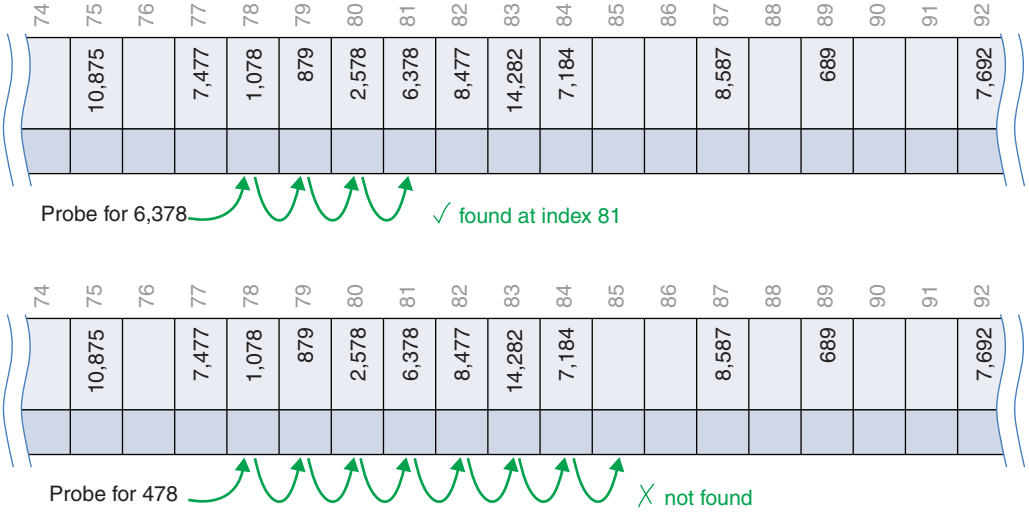


FIGURE 11-12 Linear probes in clusters

Try experimenting with filled sequences. Find the starting index of such a sequence like index 26 in Figure 11-10. After clicking that index to copy it into the text entry box, select Insert (not Search). The insertion algorithm must step through all of the filled cells to find the next empty one. After it's inserted, if you now search for that same index, the search must repeat that same process.

The Delete Button

The Delete button deletes an item whose key is typed by the user. Deletion isn't accomplished by simply removing a data item from a cell, leaving it empty. Why not? We look at the reason a little later. For now, you can see that the tool replaces the deleted item with a special key that appears as ~~*DELETED*~~ in the display.

The Insert button inserts a new item at the first available empty cell or in a deleted item. The Search button treats a deleted item as an existing item for the purposes of searching for another item further along.

If there are many deletions, the hash table fills up with these ersatz data items, which makes it less efficient. For this reason, some open addressing hash table implementations don't allow deletion. If it is implemented, it should be used sparingly to avoid large amounts of unused memory.

Duplicates Allowed?

Can you allow data items with duplicate keys to be used in hash tables? The visualization doesn't allow duplicates, which is typical behavior for hash tables. As mentioned in previous chapters, this approach implements the storage type as an associative array, where each key can have at most one value.

The alternative of allowing duplicate keys would complicate things. It would require rewriting the search algorithm to look for *all* items with the same key instead of just the first one, at least in some circumstances. That requires searching through all filled sequences of cells until an empty cell is found. Probing the entire filled sequence wastes time for all table accesses, even when no duplicates are present. Deleting an item would either try to delete the first instance or all instances of a particular key. Both cases require probing filled sequences to find the extent of the duplicates and then moving at least one of the items in the sequence to their default positions if the deletions opened up some cells. For these reasons, you probably want to forbid duplicates or use another data structure if they are required.

Avoiding Clusters

Try inserting more items in the HashTableOpenAddressing Visualization tool. The tool stops growing the table when it reaches 61 cells. As the table gets fuller, the clusters grow larger. Clustering can result in very long probe lengths. This means that accessing cells deeper in the sequence is very slow.

The fuller the array is, the worse clustering becomes. For an extreme example, use the Random Fill button to enter enough random keys so that the total number of keys is 60. Now try searching for a key that's not in the table. The initial probe lands somewhere in

the 61-cell array and then hunts for the single remaining empty (or possibly deleted) cell. If you are unfortunate enough for the initial probe to be the cell after the empty one, the search can go through all 61 cells.

Clustering is not usually a problem when the array is half full and still not too bad when it's two-thirds full. Beyond this, however, performance degrades seriously as the clusters grow larger and larger. For this reason, it's critical when designing a hash table to ensure that it never becomes more than half, or at the most two-thirds, full. We discuss the mathematical relationship between how full the hash table is and probe lengths at the end of this chapter.

Clusters are created during insertion but also affect deletion. When an item is deleted from a hash table, you would ideally mark its cell as empty so that it can be used again for a later insert and to break up potential clusters. That simple strategy, however, is a problem for open addressing because probes follow a sequence of indices in the table to locate items, stopping when they find empty cells. If you delete an item that happened to be in the middle of such a sequence, such as item 879 or item 2,578 in Figure 11-12, for example, items landing later in the probe sequence, such as item 6,378, would not be found by subsequent searches. Missing items are particularly problematic when the item being deleted is part of multiple probe sequences. In the example, item 879 happens to be in the probe sequence for item 6,378 even though their sequences start at different table indices.

You might think that there is some way to rearrange items to fill the hole created by a deletion. For instance, what if you follow the probe sequence to its end. Couldn't you move the last item in the sequence to fill the cell being deleted (somewhat like the successor node replacing a deleted node in a binary search tree)? Unfortunately, that last item in the sequence could be an item with a key that doesn't hash to the start of the probe sequence used to find the item to be deleted. For example, if you were deleting item 1,078 in Figure 11-12, following its probe sequence to the end would suggest that item 7,184 could replace it, but that would cause that item to be lost from its normal probe sequence that starts at cell 84.

Another idea: what if you find the last item in the deletion probe sequence that shares the same starting cell as the item being deleted? That ensures that you only shift an item on the same probe sequence. Unfortunately, that too causes problems because it could create a new hole in another probe sequence. If some other probe sequence happens to land on the item that was moved, then that sequence is broken. You could try to find any sequences that would visit the cell holding the item being moved as they skipped past collisions, but there could be many, many such sequences. Just like with automobiles, cleaning up after collisions is a big problem.

The approach for deleting items in open addressing is to simply mark cells as deleted, as the Visualization tool shows. The search algorithm can then step past deleted cells as it probes for a key. The insert algorithm, too, can hunt for either empty or deleted cells to use for new items. That helps a bit in keeping the size of clusters small, but only for insertion. You still must search past deleted cells when seeking an item (for search or delete). It also wastes memory, as we discuss later.

Python Code for Open Addressing Hash Tables

Let's look at the implementation of open addressing in hash tables. Aside from some of the fancier hashing functions, they are straightforward to implement. We'll make a class where it's easy to change the hash function and probing technique to resolve collisions. That design choice makes exploring different options more convenient, but it's not particularly good for performance.

The Core HashTable

The hash table object must maintain an array to hold the items it stores. That table should be private because callers should not be able to manipulate its entries. How big should that table be? We can choose to provide a parameter in the constructor to set the size, but like other data structures, it will be useful to allow it to expand later if more cells are needed.

Because we're making one class to handle hash tables with different open addressing probes, the constructor also needs a way to specify the method to search when collisions are found. The code in Listing 11-2 handles that feature by providing a `probe` parameter. The default value for the `probe` is `linearProbe`, which we describe shortly. There are also parameters for the initial `size` of the table, the `hash` function, and a `maxLoadFactor`, explained later.

LISTING 11-2 The Core `HashTable` Class

```
class HashTable(object):           # A hash table using open addressing
   def __init__(                   # The constructor takes the initial
         self, size=7,             # size of the table,
         hash=simpleHash,          # a hashing function,
         probe=linearProbe,        # the open address probe sequence, and
         maxLoadFactor=0.5):       # the max load factor before growing
      self.__table = [None] * size # Allocate empty hash table
      self.__nItems = 0            # Track the count of items in the table
      self.__hash = hash           # Store given hash function, probe
      self.__probe = probe         # sequence generator, and max load factor
      self.__maxLoadFactor = maxLoadFactor

   def __len__(self):              # The length of the hash table is the
      return self.__nItems         # number of cells that have items

   def cells(self):                # Get the size of the hash table in
      return len(self.__table)     # terms of the number of cells

   def hash(self, key):            # Use the hashing function to get the
      return self.__hash(key) % self.cells() # default cell index
```

The constructor creates a private `__table` of the specified size. The initial `None` value in the cells indicates that they are empty. The count of stored items, `__nItems`, is set to zero.

When items are inserted, they place (key, value) tuples in the table's cells, making empty and full cells easy to distinguish. All of the rest of the constructor parameters are stored in private fields for later use.

The `HashTable` defines a `__len__()` method so that Python's `len()` function can be used on instances to find the number of items they contain. A separate `cells()` method returns the number of cells in the table, so you can see how full the table is by comparing it to the number of items. As it fills, the likelihood of collisions increases.

The other core method shown in Listing 11-2 is the `hash()` method. This method is used to hash keys into table indices. We've allowed the caller to provide the hashing function in the constructor. This could be the `unique_encode_word()` function from Listing 11-1 or something similar. Whatever function is used, it should return an integer from a single key argument. The modulo of that integer with the number of cells in the table provides the initial table index for that key. This design allows callers to provide hashing functions that return very large integers, which are then mapped to the range of cells in the table.

The simpleHash() Function

The default hash function for the `HashTable` is `simpleHash()`, which is shown in Listing 11-3. This function accepts several of the common Python data types and produces an integer from their contents. It's not a sophisticated hashing function, but it serves to show how such functions are created to handle arbitrary data types.

LISTING 11-3 The `simpleHash()` Method

```
def simpleHash(key):                    # A simple hashing function
   if isinstance(key, int):             # Integers hash to themselves
      return key
   elif isinstance(key, str):           # Strings are hashed by letters
      return sum(                       # Multiply the code for each letter by
         256 ** i * ord(key[i])         # 256 to the power of its position
         for i in range(len(key)))      # in the string
   elif isinstance(key, (list, tuple)): # For sequences,
      return sum(                       # Multiply the simpleHash of each element
         256 ** i * simpleHash(key[i])  # by 256 to the power of its
         for i in range(len(key)))      # position in the sequence
   raise Exception(                     # Otherwise it's an unknown type
      'Unable to hash key of type ' + str(type(key)))
```

The `simpleHash()` function checks the type of its `key` argument using Python's `isinstance()` function. For integers, it simply returns the integer. Returning the unmodified key is, in general, a *very bad idea* because many applications use a hash table on integers in a small range. That small range (or distribution) of numbers will map directly to a small range of cell indices in the table and likely cause collisions. We choose to use it here to simplify the processing, experiment with collisions, and look at ways to avoid them.

If the key passed to `simpleHash()` is a string, the resulting integer it produces is something like the `unique_encode_word()` function you saw earlier. It takes the numeric value of each character in the key using `ord()` and multiplies that by a power of 256. The power is the position of the character in the string. The first character is power 0, which multiplies its numeric `ord` value by 1. The second character gets power 1, so it is multiplied by 256^1, the third character is multiplied by 256^2, and so on (which is the reverse of the `unique_encode_word()` function). The multiplication scheme ensures that anagram strings like `"ant"` and `"tan"` will map to different values, at least for simple strings. The products are all summed together using `sum()` and a tuple comprehension.

Note that the use of powers of 256 in the `simpleHash()` method is not sufficient to distinguish all string values. Because Python strings may contain any Unicode character whose numeric value can range up to 0x10FFFF = 1,114,111, the `simpleHash()` function can hash different strings to the same integer. Using 1,114,112 as the base instead of 256 avoids that problem, but we use 256 to keep the numbers smaller in our examples at the risk of causing more collisions.

The last `elif` clause in `simpleHash()` handles lists and tuples. These are the simple sequence types in Python. They are like strings, except that their elements could be any other kind of data, not just Unicode characters. It applies the same multiplication scheme by recursively calling `simpleHash()` on the elements individually. In this way, `simpleHash()` can recursively descend through a complex structure of sequences to find the integers and strings they contain, and build a number based on them and their relative positions.

Finally, if none of the data type tests succeed, `simpleHash()` gives up and raises an exception to signal that it doesn't have a method to convert it to an integer.

The search() Method

The `search()` method is used to find items in the hash table, navigating past any collisions. It does this by calling an internal `__find()` method to get the table index for the key as shown in Listing 11-4. It's best to keep that method private because callers shouldn't need to know which cell holds a particular item.

The `__find()` method returns an integer index to a cell or possibly `None` when it cannot find the `key` being sought. The `search()` method looks at the returned index and returns `None` in the cases where the key wasn't found. In other words, if the index returned by `__find()` is `None` or the table cell it indicates contains `None`, or the key stored in that table cell is not equal to the key being sought, the search for the key failed. The only other possibility is that the table cell's key matches the one being sought, so it returns the second item in the cell's tuple, the value associated with the key.

The definition of the constant, `__Deleted`, might seem a little unusual. This is the value stored in table cells that have been filled and later deleted. It's a **marker value**. By making it a tuple in Python, it has a unique reference address that can be compared using the `is` operator. The code must distinguish between empty cells containing `None`, deleted cells containing `__Deleted`, and full cells during open address searching. The comparison test in the `__find()` method (described shortly) uses the `is` operator instead of the == operator

to compare cell contents with __Deleted in case some application decided to store a copy of that same tuple. The search() method doesn't care whether the cell returned by __find() is empty or deleted, but the insert() method does, as you see shortly. Note also that the __Deleted marker's key, None, cannot be hashed by simpleHash(). If it could, then the search() method might return the deleted marker value as a result.

LISTING 11-4 The search() and __find() Methods of HashTable

```
class HashTable(object):                # A hash table using open addressing
…
   def search(self,                     # Get the value associated with a key
              key):                     # in the hash table, if any
      i = self.__find(key)              # Look for cell index matching key
      return (None if (i is None) or    # If index not found,
              self.__table[i] is None or # item at i is empty or
              self.__table[i][0] != key  # it has another key, return
              else self.__table[i][1])   # None, else return item value

   __Deleted = (None, 'Deletion marker') # Unique value for deletions

   def __find(self,                     # Find the hash table index for a key
              key,                      # using open addressing probes. Find
              deletedOK=False):         # deleted cells if asked
      for i in self.__probe(self.hash(key), key, self.cells()):
         if (self.__table[i] is None or  # If we find an empty cell or
             (self.__table[i] is HashTable.__Deleted and # a deleted
              deletedOK) or             # cell when one is sought or the
             self.__table[i][0] == key): # 1st of tuple matches key,
            return i                    # then return index
      return None                       # If probe ends, the key was not found
```

The __find() method takes the search key as a parameter with an optional flag parameter, deletedOK, that indicates whether it can stop after finding a deleted cell. This method implements the core of the open addressing scheme. The key is hashed using the hash function that was provided when the table was created. The hash() method (Listing 11-2) is called to map the large integer computed by simpleHash() or some other hash function to an integer in the range of the current size of the hash table. That hash address is the starting point for probing the cells of the table for the item.

The hash address returned by the call to hash() is passed to the probe function that was given when the hash table was constructed. The loop

```
for i in self.__probe(self.hash(key), key, self.cells()):
```

shows that the probe function is being used as a generator. In other words, it must create an iterator that iterates over a sequence. The elements of the sequence are the table cell

indices that should be probed for the item. The call to `self.hash(key)` returns the first index, and the key and number of cells arguments allow the generator to know how to create the rest of the sequence. We look at the `linearProbe()` generator definition shortly, but first let's look at the rest of the `__find()` method.

Inside the `for` loop, `__find()` checks the contents of cell `i` to see what's stored there. If it's `None`, the cell is empty and `i` can be returned to indicate the key is not in the table. If the cell isn't empty and has a matching key, then `__find()` can also return `i` as the result to indicate the item was found. The only tricky case is what to do if the cell has been marked as deleted. The default (`deletedOK=False`) is to treat it like another filled cell caused by a collision and continue the probe sequence. Only if the caller asked to stop on deleted cells, and the cell's value is the `__Deleted` marker, will `__find()` end the loop and return.

When some other item is found at cell `i`, the probe sequence continues. For linear probing, that is just index `i+1` or 0, after it reaches the number of cells in the table. If the whole probe sequence is completed without finding any empty cells, then the table must be full of nonmatching or deleted items. It that case, `__find()` returns `None`.

The insert() Method

The process of inserting items in the table follows the same scheme as searching and adds a few twists for handling the increasing number of items. Listing 11-5 shows the `insert()` method getting the index of a cell, `i`, by calling `__find()` on the key of the item to insert. The call is made with `deletedOK=True` to allow finding deleted cells, which `insert()` will fill.

The first test on `i` checks whether it is `None`, indicating that the probe sequence ended without finding the key, an empty cell, or a deleted cell. Either the table is full, or the probe sequence has failed to find any available cells in this case. The `insert()` method raises an exception for that. The method could try increasing the table size for this situation, but if there's a problem with the probe sequence, increasing the table size may only make matters worse.

The next test checks whether cell `i` is empty or deleted. In those cases, the contents can be replaced by a (key, value) tuple to store the item in the cell. Doing so adds a new item to the table, and the `insert()` method increments the number of items field. That increase could make the table full or nearly full. To reduce the problem of collisions, the method should increase the size of the table when the number of items exceeds some threshold.

What threshold should be used? An absolute number doesn't make sense because when it's surpassed, the table could become full again. Instead, it's better to look at the **load factor**, the ratio (or percentage) of the table cells that are full. The `load_factor()` method computes the value, which is always a number in the range 0.0 to 1.0. By comparing the load factor to the `maxLoadFactor` specified when the hash table was constructed, we can use a single threshold that's valid no matter how large the hash table grows. We examine the `__growTable()` method shortly.

LISTING 11-5 The `insert()` and `__growTable()` Methods

```
class HashTable(object):              # A hash table using open addressing
…
   def insert(self,                    # Insert or update the value associated
              key, value):             # with a given key
      i = self.__find(                 # Look for cell index matching key or an
         key, deletedOK=True)          # empty or deleted cell
      if i is None:                    # If the probe sequence fails,
         raise Exception(              # then the hash table is full
            'Hash table probe sequence failed on insert')
      if (self.__table[i] is None or  # If we found an empty cell, or
          self.__table[i] is HashTable.__Deleted): # a deleted cell
         self.__table[i] = (           # then insert the new item there
            key, value)                # as a key-value pair
         self.__nItems += 1            # and increment the item count
         if self.loadFactor() > self.__maxLoadFactor: # When load
            self.__growTable()         # factor exceeds limit, grow table
         return True                   # Return flag to indicate item inserted

      if self.__table[i][0] == key:    # If first of tuple matches key,
         self.__table[i] = (key, value) # then update item
         return False                  # Return flag to indicate update

   def loadFactor(self):               # Get the load factor for the hash table
      return self.__nItems / len(self.__table)

   def __growTable(self):              # Grow the table to accommodate more items
      oldTable = self.__table          # Save old table
      size = len(oldTable) * 2 + 1     # Make new table at least 2 times
      while not is_prime(size):        # bigger and a prime number of cells
         size += 2                     # Only consider odd sizes
      self.__table = [None] * size     # Allocate new table
      self.__nItems = 0                # Note that it is empty
      for i in range(len(oldTable)):   # Loop through old cells and
         if (oldTable[i] and           # insert non-deleted items by re-hashing
             oldTable[i] is not HashTable.__Deleted):
            self.insert(*oldTable[i]) # Call with (key, value) tuple
```

The `insert()` method finishes by returning `True` when an empty or deleted cell becomes filled. This value indicates to the caller that another cell became full. The alternative, when the hash table already has a value associated with the key to be inserted, is to replace or update the value with the new one. The final `if` clause of the `insert()` method returns `False` to indicate that no unused cells were filled by the insertion.

Growing Hash Tables

The `__growTable()` method in Listing 11-5 increases the size of the array holding the cells. We explored growing arrays in one of the Programming Projects from Chapter 2, "Arrays," and the process is a bit more complicated for hash tables. First, let's look at how much it should grow. We could add a fixed amount of cells or multiply the number of cells by some growth factor. Adding a small, fixed number of cells would keep the number of unused cells to a minimum. Multiplying by, say 2, creates a large number of unused cells initially, but means that the grow operation will be performed many fewer times.

To see the difference the growth method has, let's assume that the application using the hash table chooses to start with a small hash table of five cells and that it must store 100,000 key-value pairs. If the choices are to grow the table by five more cells or double its size for each growth step, how many steps will be needed? Table 11-1 shows the steps in growing the size of the table for the two methods.

Table 11-1 Growing Tables by a Fixed Increment and by Doubling

Fixed Size Growth	
Step	**Size**
0	5
1	10
2	15
3	20
4	25
…	
N	5 * (N +1)
…	
14	75
15	80
…	
19,998	99,995
19,999	100,000

Doubling Size Growth	
Step	**Size**
0	5
1	10
2	20
3	40
4	80
…	
N	$5 * 2^N$
…	
14	81,920
15	163,840

The fixed size growth takes 20,000 steps to reach the 100,000 cells needed. When the size doubles at every step, the 100,000 capacity is reached on the 16th step. As you've seen before, that is the difference between O(N) and O(log N) steps. Reducing the number of

growth steps is important because of what must be done after growing the array. Before we look at that, however, there's another factor in choosing the size of the new array.

The `__growTable()` method in Listing 11-5 first sets `oldTable` to reference the current hash table and estimates the `size` of the next table to be twice the old size, plus one. Then it starts a loop that finds the first prime number that equals or exceeds that size. Why? That's because prime numbers have special importance with algorithms that use the modulo operator. When you choose a prime number for the size, only multiples of that prime number hash to cell 0. Similarly, only multiples of that prime number plus one hash to cell 1. If the keys to be inserted in the hash table do not have that prime number as a factor, they tend to hash over the whole range of cells. That's very desirable behavior, as you will see later.

The test for prime numbers, `is_prime()`, is not a part of standard Python. There are many published algorithms for this (deceptively simple) test, so we don't show it here.

Rehashing

After deciding the new `size` of the hash table, the `__growTable()` method in Listing 11-5 creates the new array and sets the number of items back to zero. That might seem odd; why would we want an empty hash table at this point? The reason is that the key-value pairs in `oldTable` need to be stored in the new table, but in new positions, and none of them are in place yet. If you simply copy the contents of a cell in `oldTable` to the cell with the same index in the new array, the `__find()` method might not find it. The new array size affects where the algorithm starts its search because it is used in the modulo operation that computes the hash address. For example, in Figure 11-12, the linear probe for key 6,378 started at cell 78 and eventually found the item in cell 81 due to collisions. That was when the array size was 100. If the array size is 200, the linear probe would start at cell 178 (6,378 modulo 200). Storing that item at cell 78 in the new table would work only if there were a large cluster extending from 178 through cell 199 and then wrapping around from 0 to 78.

Instead of copying, *key-value pairs must be reinserted*, a process called **rehashing**, to ensure proper placement. The insertion process distributes them to their new cells, perhaps causing collisions, but probably fewer collisions than occurred in the smaller array. The `__growTable()` method in Listing 11-5 loops over all the cells in the smaller array and reinserts any filled cells that are not simply the deleted cell marker. This operation can be quite time-consuming, and it must scan all the cells using the `range(len(oldTable))` iterator, not just the `__nItems` known to be filled.

One implementation note: the asterisk in the `self.insert(*oldTable[i])` expression tells Python to take the tuple stored at `oldTable[i]` and use its two components as the two arguments in the call to `insert()`, which are `key` and `value`. The asterisk (`*`) means multiplication in most contexts but has a different meaning when it precedes the arguments of a function call or the elements of any tuple.

The linearProbe() Generator

Let's return to the part of the insert process that probes for empty cells. The `__find()` method in Listing 11-4 has a loop of the form

```
for i in self.__probe(self.hash(key), key, self.cells()):
```

This is the place where the `__probe` attribute of the object is called to generate the sequence of indices to check. The default value for the `__probe` attribute is `linearProbe()`, which is shown in Listing 11-6.

LISTING 11-6 The `linearProbe()` Generator for Open Addressing

```
def linearProbe(              # Generator to probe linearly from a
      start, key, size):      # starting cell through all other cells
   for i in range(size):      # Loop over all possible increments
      yield (start + i) % size # and wrap around after end of table
```

The `linearProbe()` is a straightforward generator that behaves similarly to Python's `range()` generator. In fact, it uses `range()` in an internal loop that steps a variable, `i`, through all `size` cells of the table. The `i` index is added to the starting index for the probe, so it will examine all the subsequent cells in the array. When that offset index goes past the end of the table, the iterator wraps the index back to zero by using the modulo of the offset index with `size`. The new index will always be between zero and one less than `size`.

The `yield` statement in the loop body sends the new index back to the `__find()` method to be checked (Listing 11-4). Remember that the `yield` statement returns a value and control to the caller. The caller then uses the value in its own loop until it's time to get the next value from the iterator. Control then passes back to the iterator right after the `yield` statement. In this case, `linearProbe()` goes on and increments its own `i` variable.

When `linearProbe()` finishes going through all the indices of the array, the generator ends (by raising a `StopIteration` exception). That signals to the caller, `__find()`, that all the cells have been probed. If the `insert()` method hasn't found an empty cell before the linear probe sequence finishes, then the table must be full.

Note that the `linearProbe()` iterator does not use the goal key to determine the sequence of indices. We will discuss that shortly.

The delete() Method

Deleting items is straightforward for hash tables because you only need to mark the deleted cells. Like with insertion, the `delete()` method starts by using the `__find()` method to find the cell containing the item to delete, as shown in Listing 11-7. After the cell index, `i`, is determined, the behavior depends on what is stored at the cell. If `__find()` could not discover that cell, or it already contains a deleted element or some other item whose key does not match, then a possible error has been found. This `delete()` method has an optional parameter, `ignoreMissing`, which determines whether an exception

should be raised. In general, data structures that store and retrieve data should raise exceptions when the caller tries to remove an item not in the store, but in some circumstances such errors can be safely ignored.

LISTING 11-7 The `delete()` Method of `HashTable`

```
class HashTable(object):                # A hash table using open addressing
…
   def delete(self,                     # Delete an item identified by its key
              key,                      # from the hash table. Raise an exception
              ignoreMissing=False):     # if not ignoring missing keys
      i = self.__find(key)              # Look for cell index matching key
      if (i is None or                  # If the probe sequence fails or
          self.__table[i] is None or    # cell i is empty or
          self.__table[i][0] != key):   # it's not the item to delete,
         if ignoreMissing:              # then item was not found. Ignore it
            return                      # if so directed
         raise Exception(               # Otherwise raise an exception
           'Cannot delete key {} not found in hash table'.format(key))

      self.__table[i] = HashTable.__Deleted # Mark table cell deleted
      self.__nItems -= 1                # Reduce count of items
```

When the `delete()` method finds a cell with a matching key, it marks the cell with the special `__Deleted` marker defined for the class and decrements the count of the number of stored items. Typically, no attempt to resize the hash table is made when many deletions cause the load factor to shrink below the threshold used to determine when to grow the table. Skipping the resize operation is based two assumptions: that deletion will occur much less often than insertion and search, and that the cost of having to rehash all the items stored in the table.

The traverse() Method

To traverse all the items in a hash table, all the table cells must be visited to determine which ones are filled. The process is easy to implement as a generator. The one special consideration is that deleted items should not be yielded. Listing 11-8 shows the implementation.

LISTING 11-8 The `traverse()` Method of `HashTable`

```
class HashTable(object):                # A hash table using open addressing
…
   def traverse(self):                  # Traverse the key, value pairs in table
      for i in range(len(self.__table)): # Loop through all cells
         if (self.__table[i] and        # For those that contain undeleted
             self.__table[i] is not HashTable.__Deleted): # items
            yield self.__table[i]       # yield them to caller
```

Because the implementation stores the key-value pairs as (immutable) Python tuples, they can be yielded directly to the caller, which can assign them to two variables in a loop such as

```
for key, value in hashTable.traverse():
```

Alternatively, callers can use a single loop variable holding the pair as a tuple.

The Traverse and New Buttons

Returning to the Visualization tool, the Traverse button launches the preceding loop. Each item's key is printed in a box (ignoring its data and adding a comma to separate the keys). It illustrates the `traverse()` iterator skipping over empty and deleted cells.

You can create new, empty hash tables with the New button. This button takes two arguments: the number of initial cells and the maximum load factor. You can specify starting sizes of 1 to 61 cells and maximum load factors from 0.2 to 1. When invalid arguments are provided, the default values of 2 and 0.5 are used.

If you create hash tables with nonprime sizes, they will grow using the `__growTable()` method of Listing 11-5, setting the new size to a prime number. Try stepping through the animation of the rehashing process. This animation shows how the items move to their new cells.

If you want to see the effects of using different table sizes, try using the New button to create a table of the desired size with a maximum load factor of 0.99. The table will not grow until it becomes completely full, so you can see the effects of different table sizes and clustering.

Quadratic Probing

Using open addressing, hash tables can find empty (and deleted) cells to fill with new values, but clusters can form. As clusters grow in size, it becomes more likely that new items will hash to cells within a cluster. The linear probe steps through the cluster and adds the new item to the end, making it even bigger, perhaps joining two clusters.

This behavior is somewhat like that of automobiles entering a highway. If only isolated vehicles make up the flow of highway traffic, the arriving vehicles have plenty of gaps to fit into. When the highway is crowded, longer chains of vehicles form clusters. Newly arriving vehicles wait for the cluster to pass and join at the end, increasing the cluster size. Hopefully, the arriving vehicles don't cause real-world collisions as they "probe" for an open spot on the highway.

The likelihood of forming clusters and the size of clusters depend on the ratio of the number of items in the hash table to its size—its load factor. Clusters can form even when the load factor isn't high, especially when the hashing function doesn't distribute keys uniformly over the table. Parts of the hash table may consist of big clusters, whereas others are sparsely populated. Clusters reduce performance.

Quadratic probing is an attempt to keep clusters from forming. The idea is to probe more widely separated cells instead of those adjacent to the primary hash site.

The Step Is the Square of the Step Number

In a linear probe, if the primary hash index is x, subsequent probes go to $x + 1$, $x + 2$, $x + 3$, and so on. In quadratic probing, subsequent probes go to $x + 1$, $x + 4$, $x + 9$, $x + 16$, $x + 25$, and so on. The distance from the initial probe is the square of the step number: $x + 1^2$, $x + 2^2$, $x + 3^2$, $x + 4^2$, $x + 5^2$, and so on. Figure 11-13 shows some quadratic probes.

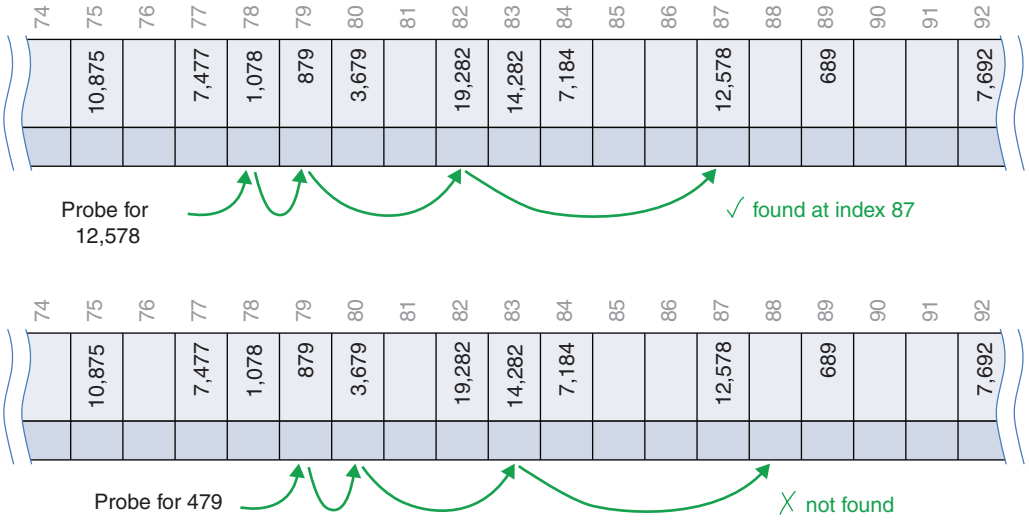


FIGURE 11-13 Quadratic probes

The quadratic probe starts the same as the linear probe. When the initial probe lands on a nonmatching key, it picks the adjacent cell. If that's occupied, it may be in a small cluster, so it tries something four cells away. If that's occupied, the cluster could be a little larger, so it tries nine cells away. If that's occupied, it really starts making long strides and jumps to 16 cells away. Pretty soon, it will go past the length of the array, although it always wraps around because of the modulo operator.

Using Quadratic Probing in the Open Addressing Visualization Tool

The HashTableOpenAddressing Visualization tool can demonstrate different kinds of collision handling—linear probing, quadratic probing, and double hashing. (We look at double hashing in the next section.) You can choose the probe method whenever the table is empty by selecting one of the three radio buttons. When the table has one or more items, the buttons are disabled to preserve the integrity of the data.

To see quadratic probing in action, try the following. Use the New button to create a hash table of 21 cells with a maximum load factor of 0.9. Select the Use quadraticProbe button to switch to quadratic probing. Then use the Random Fill button to insert 12 random keys in the table. This action produces a table with various filled sequences, like the one shown in Figure 11-14. There is a six-cell filled sequence along with several shorter ones.

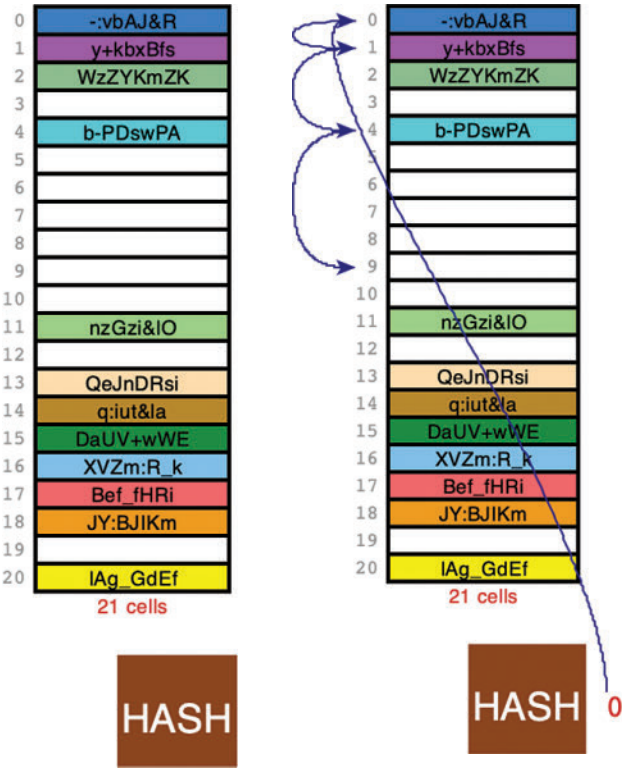


FIGURE 11-14 Quadratic probing in the HashTableOpenAddressing Visualization tool

In this example, try inserting the key *0*. Because it is a numeric key, the hashed value is 0 and the first probe is to cell 0. After finding cell 0 full, it tries the cell at 0 + 1. That cell is occupied, so it continues to cell 0 + 4, finding another stored item. When it reaches 0 + 9, it finds cell 9 empty and can insert key 0 there. It's easy to see how the quadratic probes spread out further and further.

If you next try to insert the key *21*, it will hash to cell 0 for its initial probe again because the table has 21 cells. The insertion now will repeat the same set of probes as for key 0 and then continue on to locate an empty cell. Perhaps surprisingly, it revisits some of the same cells during the probing, as shown in Figure 11-15.

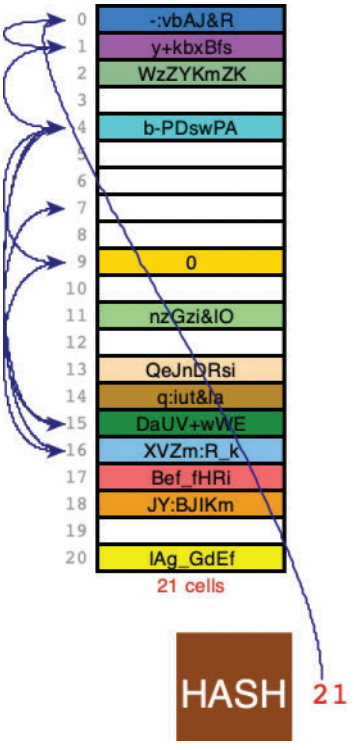


FIGURE 11-15 Inserting key 21 by quadratically probing a relatively full hash table

Specifically, the cells probed to insert key 21 are 0, 1, 4, 9, 16, (25) 4, (36) 15, (49) 7. The indices in parentheses are the values before taking the modulo with 21.

This example is particularly troublesome. Not only does it have to probe all the cells probed to insert key 1, but it also repeats the probe at cell 4 that wouldn't have occurred using linear probing. If you keep inserting more keys, you will see how this behavior becomes worse when the table is almost full. With the max load factor set to 0.9, it won't grow the table until the 19th key is inserted.

Incidentally, if you try to fill the hash table up to the maximum 61 items it supports or with a very high maximum load factor, the Visualization tool may not be able to insert an item, even if empty cells remain. The program tries only 61 probes before giving up (or whatever the size of the current table is). Because quadratic probes can revisit the same cells, the sequence may never land on one of the few remaining empty cells.

Also try some searches when the table is nearly full, both for existing keys and ones not in the table. The probe sequences get very long in some cases.

Implementing the quadratic probe is straightforward. Listing 11-9 shows the `quadraticProbe()` generator. Like the `linearProbe()` shown in Listing 11-6, it uses `range()` to

loop over all possible cells one time. The loop variable, `i`, is squared, added to the `start` index, and then mapped to the possible indices using the modulo operator. Because `i` starts at zero, the first yielded index is the `start` index.

LISTING 11-9 The `quadraticProbe()` Generator for Open Addressing

```
def quadraticProbe(                 # Generator to probe quadratically from a
      start, key, size):            # starting cell through all other cells
   for i in range(size):            # Loop over all possible cells
      yield (start + i ** 2) % size # Use quadratic increments
```

The Problem with Quadratic Probes

Quadratic probes reduce the clustering problem with the linear probe, which is called **primary clustering**. Quadratic probing, however, suffers from different and more subtle problems. These problems occur because all the probing sequences follow the same pattern in trying to find an available cell.

Let's say 184, 302, 420, and 544 all hash to address 7 and are inserted in this order. Then 302 will require a one-cell offset, 420 will require a four-cell offset, and 544 will require a nine-cell offset from the first probe. Each additional item with a key that hashes to 7 will require longer probes. Although the cells are not adjacent in the hash table, they still are causing collisions. This phenomenon is called **secondary clustering**.

Secondary clustering is not a serious problem. It occurs for any hashing function that places many keys at the same initial cell or at multiple cells that happen to be spaced by a square number away from other filled cells. There's another issue, however, and that's the coverage of cells visited by the probing sequence.

The quadratic probe keeps making larger and larger steps. There's an unexpected interaction between those steps and the modulo operator used to map the index onto the available cells. In the linear probe, the index is always incremented by one. That means that linear probing will eventually visit every cell in the hash table after wrapping around past the last index.

In quadratic probing, the increasing step sizes mean that it eventually visits only about half the cells. The example in Figure 11-15 illustrated part of the problem when it revisited cell 4. The reason for this behavior takes some mathematics to explain.

If you look at the spacing between the cells probed, you'll see that it increases by two at each step. The spacing between $x + 1$ and $x + 4$ is three. The spacing between $x + 4$ and $x + 9$ is five. The spacing between $x + 9$ and $x + 16$ is seven, and so on. It already looks as though it might skip every other cell because it will stay on the odd cells if the initial probe was to an even cell (and vice versa). That's actually not the case because the modulo operator will change between odd and even numbered cells when the index goes past the modulo value. That value is usually a prime number, so it is odd.

Even with a prime number of cells, however, the quadratic probe starts repeating the same sequence of cell indices fairly quickly. Here's a basic example. For simplicity, let's say the hash table has seven cells in it, and the key to store initially hashes to cell index 0.

Quadratic probing then visits indices 1, 4, 9, 16, 25, 36, 49, 64, and so on. After taking the modulo with seven, however, the full sequence is 0, 1, 4, 2, 2, 4, 1, 0, 1, 4, 2, 2, 4, 1, 0, and so on. That 0, 1, 4, 2, 2, 4, 1 sequence repeats forever, leaving out cell indices 3, 5, and 6.

Three cells may not seem like much, but they're three out of seven cells total. Even worse, the probe revisits indices 1, 2 and 4 twice. Because they've already been visited during the seven-probe sequence, they must already be occupied, so revisiting them just wastes time (much more time than the single cell revisited in the example in Figure 11-15). As the prime number of cells gets bigger, the repetitive behavior continues. After the quadratic term grows to be the square of the table size, the sequence returns to the starting index. Eventually, about half of the cells are visited, and half are not.

So linear probing ends up causing primary clustering, whereas quadratic probing ends up with secondary clustering and only half the coverage of the hash table. This approach is not used because there's a much better solution.

Double Hashing

To eliminate secondary clustering as well as primary clustering and to help with hash table coverage, there's another approach: **double hashing**. Secondary clustering occurs for any algorithm that generates the same sequence of probing for every key.

What we need are probe sequences that differ for each key instead of being the same for every key. Then numbers with different keys that hash to the same index will use different probe sequences.

The double hashing approach *hashes the key a second time, using a different method, and uses the result as the step size*. For a given key, the step size remains constant throughout a probe, but it's different for different keys. As long as the step size is not a multiple of the array size, it will eventually visit all the cells. That's one reason why prime numbers are good for the array size; they make it easier to avoid getting a step size that evenly divides the size of the array.

Experience has shown that this secondary hash function must have certain characteristics:

- It must not be the same as the primary hash function.
- It must never output a 0 (otherwise, there would be no step; every probe would land on the same cell).

Experts have discovered that functions of the following form work well:

```
stepSize = constant - (key % constant)
```

where `constant` is prime and smaller than the array size. For example,

```
stepSize = 5 - (key % 5)
```

The HashTableOpenAddressing Visualization tool uses this approach for its double hashing probe. Different keys may hash to the same index, but they will (most likely) generate

different step sizes. With this algorithm and the `constant = 5`, the step sizes are all in the range 1 to 5. Two examples are shown in Figure 11-16. The first search for key 4,678 starts at cell 78. The secondary hash function determines that the step size will be three for that key. After probing three filled cells, the function finds the desired key on the fourth step. The second search is for key 178 and starts at the same cell. For this key, however, the step size is determined to be four. After probing two filled cells, the function finds an empty cell on the third step.

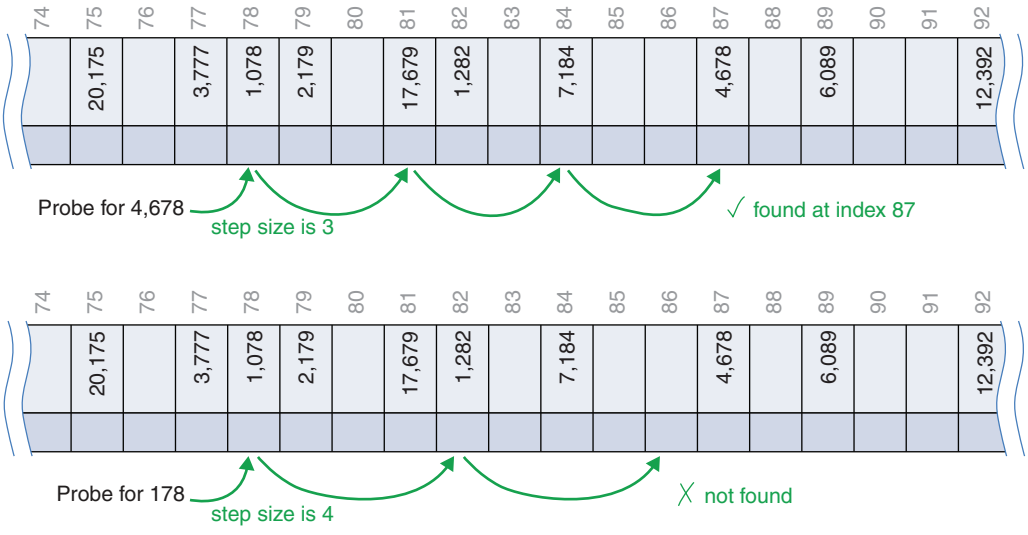


FIGURE 11-16 Double hashing

Implementing double hashing is only slightly more complicated than linear or quadratic probing. You still need a generator that loops over the possible indices, but this time you need to apply the secondary hash function to get the step size. That step should be less than or equal to a prime number below the size of the array. Listing 11-10 shows an implementation for that generator.

The `doubleHashProbe()` generator starts by immediately yielding the first cell index folded over the possible range of cells. This is a little different from the approach in the linear and quadratic probes. The reason: you don't have to determine the step size if the first cell ends up being the desired one. Like the other probes, it doesn't need to know whether the caller seeks an empty or filled cell. If the caller finds an acceptable cell, it will exit its loop through the generator sequence, skipping the calculation of the step size. If the loop continues, then the generator calls `doubleHashStep()` to compute the step size. That value is used in a loop similar to those of the linear and quadratic probes. The difference is that it starts with 1 times the step size added to the start index and continues through all possible cells.

The `doubleHashStep()` function computes the step size for the given `key`. First, it gets the largest prime number that is smaller than the array size by calling `primeBelow()`. Next, it

reapplies the hash function, `simpleHash()`, to the `key` (although it would be better to use a different hash function here). The large integer it produces gets mapped to the range of the smaller prime number using the modulo operator. The remainder is subtracted from the `prime` so that the step size falls in the range [1, `prime`].

LISTING 11-10 The `doubleHashProbe()` Generator for Open Addressing

```
def doubleHashProbe(                  # Generator to determine probe interval
      start, key, size):              # from a secondary hash of the key
   yield start % size                 # Yield the first cell index
   step = doubleHashStep(key, size)   # Get the step size for this key
   for i in range(1, size):           # Loop over all remaining cells using
      yield (start + i * step) % size # step from second hash of key

def doubleHashStep(key, size):        # Determine step size for a given key
   prime = primeBelow(size)           # Find largest prime below array size
   return prime - (                 # Step size is based on second hash and
      simpleHash(key) % prime)      # is in range [1, prime]

def primeBelow(n):                    # Find the largest prime below n
   n -= 1 if n % 2 == 0 else 2      # Start with an odd number below n
   while (3 < n and not is_prime(n)): # While n is bigger than 3 or
      n -= 2                         # is not prime, go to next odd number
   return n                           # Return prime number or 3
```

The `primeBelow()` function is a straightforward math calculation. It starts by finding an odd number below the parameter, `n`, by subtracting either 1 or 2 from it, depending on whether `n` is even or odd. The `while` loop decrements `n` by 2 until it either finds a prime or reaches 3.

To save some time, the `primeBelow` result could be stored in the `HashTable` object and recomputed only when the array size changes. The step size changes based on the key, so it cannot be stored as efficiently. You would essentially need a hash table to look up the step size for each key!

Using Double Hashing in the Open Addressing Visualization Tool

As with the other methods, you can set the probe type whenever the hash table is empty in the HashTableOpenAddressing Visualization tool. To see a good example of the probes at work, you need to fill the table rather full, say to about nine-tenths capacity or more. Try creating a hash table of 41 cells with the maximum load factor of 0.9. Set the probe type by selecting the Use doubleHashProbe button. Fill most of the table with 30 random keys (perhaps without animating the hashing to go faster).

With such high load factors, only about a quarter of new, random data items will be inserted at the cell specified by the first hash function; most will require extended probe sequences. Try inserting one or two more random keys by using the Random Fill button with animation of the hashing.

Try finding some existing keys in the crowded table. When a search needs a multistep probe sequence, you'll see how all the steps are the same size for a given key, but that the step size is different. Some step sizes can be large and, when wrapping around the table size, can take what looks like a random path among the cells.

The Visualization tool does not show the code for calculating the step size. In fact, the code is not shown for any of the probe sequence generators. You can still see the patterns they produce, however, by following the arrows indicating the cells being probed. The next section discusses the step-by-step execution of double hashing probes.

Double Hashing Example

The double hashing algorithm has many steps. If you haven't looked at the Visualization tool, here's an example of how a series of insertions works. We start with an empty `HashTable` created with the `doubleHashProbe()` as its probe sequence (see Listing 11-10). We create the hash table with an initial size of 7 (which is different than the Visualization tool default size of 2). As we insert keys, the hash function computes where to store them and how big the steps should be to probe for open addresses. When the load factor gets too large, the table grows to accommodate more key-value pairs.

Table 11-2 shows how each insertion probes to find the cell to modify. The first key inserted is 1. The `simpleHash()` function just returns the same integer. That makes it easy to see how insertion works (but it's a terrible hash function in general). Using the same integer also clarifies what the step size will be for double hashing. The first prime below 7 is 5. Subtracting 1 mod 5 from 5 leaves 4 as the step size. Because cell 1 is empty, however, the step size doesn't matter, and item 1 is inserted into cell 1.

Table 11-2 Filling a Hash Table Using Double Hashing

Item Number	Key	`simpleHash` Value	Step Size	Total Cells	Prime Below	Probe Sequence After `simpleHash`
1	1	1	4	7	5	
2	38	3	2	7	5	
3	37	2	3	7	5	
4	16	2	4	7	5	6
5	20	3	6	17	13	9
6	3	3	10	17	13	13
7	11	11	2	17	13	
8	24	7	2	17	13	
9	4	4	9	17	13	13, 5
10	16*	16	15	37	31	
11	10	10	21	37	31	
12	31	31	31	37	31	

Item Number	Key	simpleHash Value	Step Size	Total Cells	Prime Below	Probe Sequence After simpleHash
13	18	18	13	37	31	
14	12	12	19	37	31	
15	30	30	1	37	31	
16	1*	1	30	37	31	
17	19	19	12	37	31	
18	85	11	8	37	31	19, 27

The second key inserted, 38, follows the same pattern. After hashing and taking the modulo with 7, the probe starts at index 3. It would get a different step size of 2 based on 5 – (38 mod 5), but again, that doesn't matter because cell 3 is empty. The third key inserted, 37, maps to index 2, which is also empty.

On the fourth key, 16, we hit the first collision. The `simpleHash()` maps it to index 2, which holds key 37. The step size for that key is 5 – (16 mod 5) = 4. The second probe goes to cell 6 (2 + 4), which is empty, so key 16 gets stored there. The table shows cell 6 in the Probe Sequence After `simpleHash` column.

At this point, four items are stored in the hash table. The table has seven cells, so the load factor is now 4/7, which is larger than the default `maxLoadFactor` of 0.5. The `insert()` method calls the `__growTable()` method after the fourth item is inserted. The table grows to hold 17 cells, and the four items are rehashed into them as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	1		37	38												16

All of the keys rehash into cells of the expanded table at their initial hash addresses based on the new size.

The fifth item to insert has key 20. That hashes to cell 3 in the 17-cell table, which is occupied. The step size is 6, which is calculated using the largest prime below the table size, which is 13 (`step` = 13 – (20 mod 13)). Probing six cells away finds cell nine to be empty, so that's where item 20 lands.

The sixth item to insert has key 3. That also hashes to the occupied cell 3. The secondary hashing leads to a step size of 10, and the next probe finds cell 13 to be empty. Item 3 gets placed in cell 13, avoiding the enlargement of any of the clusters. These last insertions of keys 20 and 3 illustrate how two keys, whose primary hash addresses collide, avoid creating clusters by using different step sizes after the initial probe.

The seventh item with key 11 finds cell 11 empty and lands there. That pattern repeats for item 8, key 24, which is stored in the empty cell 7.

More collisions occur with the ninth item, which has a key of 4. The contents of the array just before the insertion of that item are

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	1		37	38			24		20		11		3			16

Cell 4 is occupied, so it computes the step size of 9 (13 – (4 mod 13)). The next cell checked is 13. That cell is full, too, so it probes at 5 (13 + 9 % 17 = 5), which is empty. Key 4 is stored there, forming a cluster of three filled cells.

Storing the ninth item in a table of 17 cells brings the load factor to 9/17, which exceeds 0.5. The table grows again to hold 37 cells, and the nine items are reinserted into it. One collision occurs during the reinsertions (key 38 would normally go at index 1 in a 37-cell table but gets stored at cell 25 (1 + (31 – (38 mod 31))).

As Table 11-2 shows, items 10 through 17 are inserted without any collisions. That behavior is typical when the load factor is reduced after growing the table. Looking at the details, you can see that item 10 is actually a duplicate key to item 4, as indicated by the asterisk (*). Both have a key of 16, so the second insertion becomes an update to the value associated with that key. Item 16 is another duplicate; this time it's key 1. It also immediately finds the key at cell 1 and doesn't need to probe other cells before updating the value.

The eighteenth item produces a collision. It's a new key, 85, which hashes to cell 11 (because the table is now of length 37). Because key 11 is stored in cell 11, the double hashing determines that it should step by eight cells. Probing cell 19 finds it occupied with key 19 from the previous insertion, so it moves on to store that key in the available cell 27.

As you can see from this example, collisions do cause some items to be placed in cells different than their original hash location. The chance of a collision increases as the load factor goes up. The 18 insertions filled 16 cells of the table because two of the keys were duplicates of previous keys. That puts the table load factor at 16/37. After inserting three new keys, the table must grow again, keeping the chance of collision low and the number of clusters small.

Table Size a Prime Number

Double hashing requires the size of the hash table to be a prime number. To see why, imagine a situation in which the table size is not a prime number. For example, suppose the array size is 15 (indices from 0 to 14), and that a particular key hashes to an initial index of 0 and a step size of 5. The probe sequence would be 0, 5, 10, 0, 5, 10, and so on, repeating endlessly. Only these three cells would ever be examined, so the algorithm will never find the empty cells that might be waiting at 1, 2, 3, and so on. The reduced coverage of the available cells means the algorithm will exhaust all its probes before quitting. In other words, it will crash and burn.

If the array size were instead 13, which is prime, the probe sequence would eventually visit every cell. It would be 0, 5, 10, 2, 7, 12, 4, 9, 1, 6, 11, 3, and so on. If there were even one empty cell, the probe would find it. Using a prime number as the array size makes it impossible for any number to divide it evenly (other than 1 and the prime itself), so the probe sequence will eventually check every cell.

Despite the added time needed to find prime numbers, double hashing wins overall when choosing the best probe sequence for open addressing.

Separate Chaining

In open addressing, collisions are resolved by looking for an open cell in the hash table. A different approach is to install a linked list or binary tree at each index in the hash table. A data item's key is hashed to the index in the usual way, and the item is inserted into the structure at that index. Other items that hash to the same index are simply added to the structure; there's no need to search for empty cells in the primary array. Figure 11-17 shows how separate chaining looks. The top version shows sorted linked lists in the table cells, and the bottom shows balanced, binary trees.

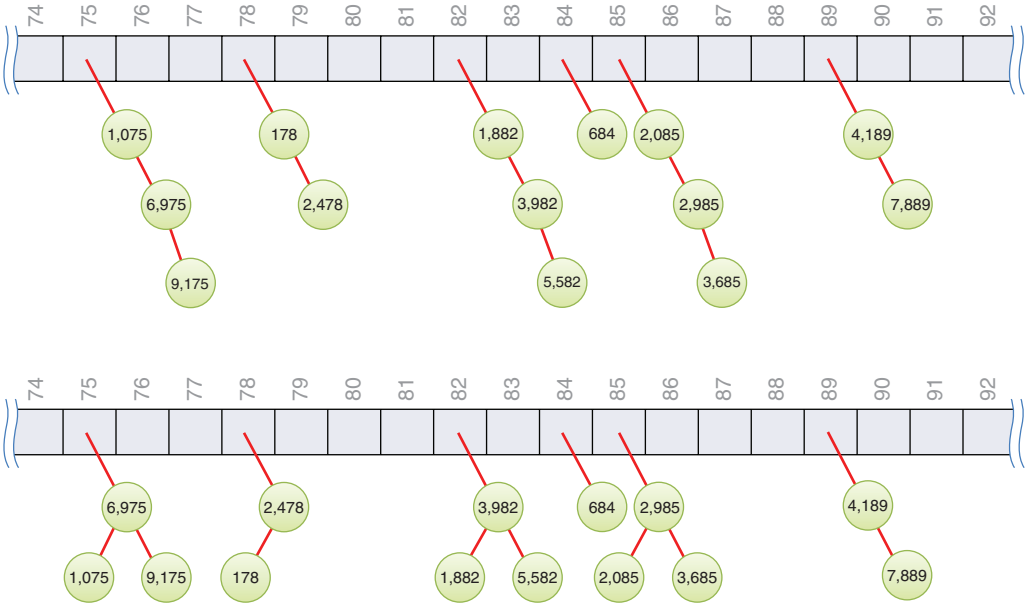


FIGURE 11-17 Examples of separate chaining

Separate chaining is conceptually somewhat simpler than the various probe schemes used in open addressing. The code, however, is longer because it must include the mechanism for the linked list or trees, usually in the form of an additional class.

The HashTableChaining Visualization Tool

To see how separate chaining works, start the HashTableChaining Visualization tool. It displays an empty array, as shown in Figure 11-18. Like the open addressing tool, it starts with two empty cells that will grow as new items are added. The linked lists start from these cells and grow downward.

FIGURE 11-18 Separate chaining in the HashTableChaining Visualization tool

The buttons in the HashTableChaining Visualization tool are nearly identical to those of the HashTableOpenAddressing tool, except that the probe choice buttons have been removed. Despite the similar appearance, the operations have quite a few differences.

Try inserting some items. As with the open addressing tool, the keys can be strings or integers, and the same `simpleHash()` function determines which cell of the array should contain the key. The hashing box, item count, and load factor limit are at the top to keep them out of the way as the linked lists grow downward.

The linked lists are unsorted (we discuss the option of keeping them sorted shortly). New items are appended to the end of the list. After the total number of items in all the lists divided by the number of cells exceeds the `maxLoadFactor` limit, a similar `__growTable()` function creates an array that's at least twice as big with a prime number of cells. The items are rehashed to find their location in the new array.

It might seem strange that the separate chaining tool starts with a `maxLoadFactor` of 1.0 (100%). You saw how congested hash tables caused problems in open addressing. Congestion can be a problem in separate chaining, too, but in a different way.

Try filling the array with 30 random keys (you can turn off the animation of the insertion by deselecting the Animate Hashing option before filling). You will likely get a table that looks similar to the one in Figure 11-19. Most (14) of the linked lists have one item, a few (5) have two items, and two have three items in the example (the lists in cells 26 and 43). The 30 items are stored in 21 cells of the 47-cell table. That's typical of separate chaining; the items are spread out over many, but not all, of the cells and the length of the chains is relatively short.

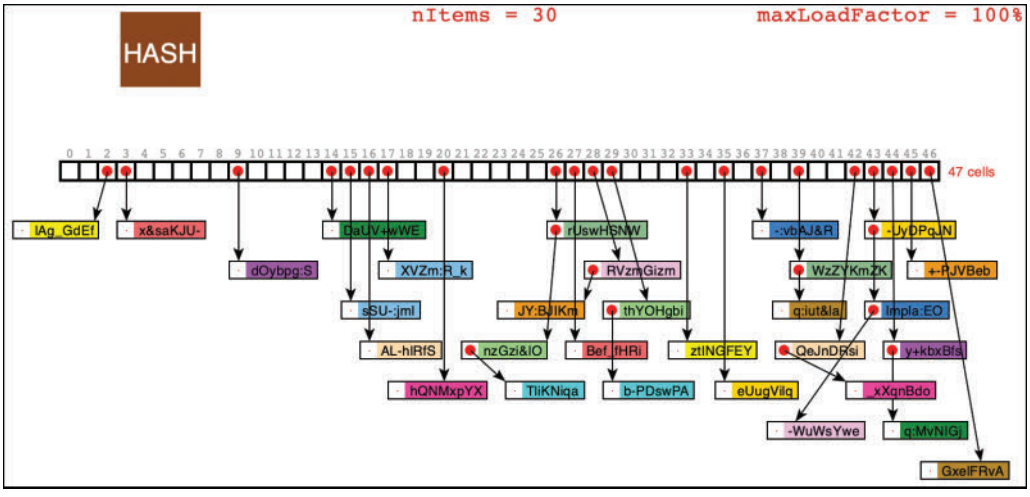


FIGURE 11-19 A separate chaining hash table with 30 random items

With items in the hash table, try a few searches. If you click the index number for a cell, it will enter that number in the text entry box. You can use that index as a search key. If you click index 26 of the hash table in Figure 11-19, enable the Animate Hashing option, and then select Search, you can see how the hashing finds cell 26 immediately and must step through each of the three items in the linked list to determine that no item with key 26 is in the hash table. Searches for existing keys also may step through a few items before finding the goal key.

Try deleting a few items (remembering that you can click a key to copy it to the text entry box). The search mechanism for the item to delete is the same, and the animation shows the steps it takes. Unlike open addressing, the item is deleted from the linked list, not replaced with a special deleted value.

In the Visualization tool, the items are placed roughly below the hash table cell in which they belong but are adjusted in position to avoid obscuring other items and to reduce some of the overlapping arrows. The arrows have different lengths, making the number of links in each list harder to see. For example, cell 16 in Figure 11-19 holds a linked list of one item, "AL-hlRfS," and that item was placed below all the others due to the presence of other items inserted before it.

Load Factors

The load factor is typically managed differently in separate chaining than in open addressing. In separate chaining it's normal to put N or more items into an N cell array; thus, the load factor can be 1 or greater. There's no problem with this; some locations simply contain two or more items in their lists.

Of course, if the lists have many items, access time takes longer because access to a specified item requires searching through an average of half the items on the list. Finding the initial cell is fast, taking O(1) time, but searching through a list takes time proportional to M, the average number of items on the list. This is O(M) time. Thus, you don't want the lists to become too full. If you use binary trees, the search time is O(log M). You can let the trees grow more than lists do, if they remain balanced.

A load factor of 1, as shown in the initial Visualization tool, is common. In open addressing, performance degrades badly as the load factor increases above one-half or two-thirds. In separate chaining the load factor can rise above 1 without hurting performance very much. This insensitivity to the load makes separate chaining a more robust mechanism and reduces unused memory, especially when it's hard to predict in advance how much data will be placed in the hash table. There is still a need to grow the hash table when the load factor (or the longest list or deepest tree) grows too big, but the decision metric is different.

You can experiment with load factor limits from 0.5 to almost 2.0 by creating new tables with the New button in the Visualization tool. You can also fill with up to 99 randomly keyed items, and the growth of the hash table size is limited to 61 cells. This limitation creates some very congested separate chaining hash tables. The scroll bars will be adjusted to allow you to view the whole structure. The visualization slows down for large tables as it attempts to adjust the links to avoid overlaps.

Duplicates

Duplicate keys could be allowed in separate chaining but typically are not for all the same reasons applied to open addressing, plus the reasons mentioned for linked lists and binary trees. If they are allowed, all items with the same key will be inserted in the same list or tree. Therefore, if you need to discover all of them, you must search more of the list or tree in both successful and unsuccessful searches. Deletion must also search further when deleting all items with the same key. These extended searches lower performance.

Deletion

In separate chaining, deletion is easier than in open addressing. There's no need for a marker for a deleted item. The algorithm hashes to the proper table cell and then deletes the item from the list or tree. If the item is the last item stored in that cell, the cell can be set back to empty, if minimum memory usage is important. Leaving an empty tree or list header in the cell does not harm garbage collection.

Table Size

With separate chaining, making the table size a prime number is not as important as it is with open addressing. There are no probe sequences in separate chaining, so you don't need to worry that a probe will go into an endless sequence or not cover all the cells because the step size divides evenly into the array size.

On the other hand, certain kinds of key distributions can cause data to cluster when the array size is not a prime number. We have more to say about this problem when we discuss hash functions.

Buckets

Another approach similar to separate chaining is to use an array at each cell in the hash table instead of a linked list. Such fixed-size arrays are sometimes called **buckets** (although some hash table descriptions use the term "*bucket*" to mean what we've been describing as a "*cell*" of the hash table). This approach is not as efficient as the linked list approach, however, because of the problem of choosing the size of the buckets. If they're too small, they may overflow, requiring growth of the hash table and rehashing all the items. If the buckets are too large, they waste memory. Linked lists and binary trees, which allocate memory dynamically, don't have this problem.

Python Code for Separate Chaining

There are many commonalities between the code for open addressing and separate chaining. Listing 11-11 shows the core class definition for a separate chaining `HashTable`. Compare that with the definition shown in Listing 11-2 for open addressing. There are three differences: there's an `import` statement to get the `KeyValueList` class, there's no `probe` parameter in the constructor because there is no probe sequence, and the default value for `maxLoadFactor` is higher. This class uses a slightly revised version of the `LinkedList` class from Chapter 5, "Linked Lists," for the separate chaining. You could also use the `AVLTree` from Chapter 10, "AVL and Red-Black Trees." It only needs to support the same interface of creation, insertion, search, deletion, and traversal. We discuss the `KeyValueList` class and the tree alternative later.

The constructor for the class could initialize all the cells of the hash table to be empty linked lists. Doing so, however, would increase the memory consumed by the structure and add time to the construction process. It's preferable to wait until items need to be inserted in cells before creating the chain (list or tree) to hold them. Note that the utility methods `__len__()`, `cells()`, and `hash()` remain identical. Now let's look at what changes for separate chaining.

Perhaps the most significant change is the lack of a `__find()` method. For other data structures you used a find method to locate where an item is stored. In the case of separate chaining, you can use the hash function to find the cell (or bucket), and the chain's search and find methods to locate the item within that cell.

LISTING 11-11 The Core `HashTable` Class for Separate Chaining

```
from KeyValueList import *

class HashTable(object):           # A hash table using separate chaining
   def __init__(                   # The constructor takes the initial
         self, size=7,             # size of the table,
         hash=simpleHash,          # a hashing function, and
         maxLoadFactor=1.0):       # the max load factor before growing
      self.__table = [None] * size # Allocate empty hash table
      self.__nItems = 0            # Track the count of items in the table
      self.__hash = hash           # Store given hash function, and max
      self.__maxLoadFactor = maxLoadFactor # load factor

   def __len__(self):              # The length of the hash table is the
      return self.__nItems         # number of cells that have items

   def cells(self):                # Get the size of the hash table in
      return len(self.__table)     # terms of the number of cells

   def hash(self, key):            # Use the hashing function to get the
      return self.__hash(key) % self.cells() # default cell index
```

The `search()` method for the separate chaining `HashTable` simply hashes the key to a cell index, `i`; checks whether cell `i` has been filled with a list object; returns `None` if it doesn't or the result of searching for the key in the list. Listing 11-12 shows the implementation. All the complexity of the probe sequence has gone away and is replaced by the, hopefully well-understood, operation of the chain structure (linked list or tree).

The `insert()` method is also simplified. After hashing the key to get the cell index, `i`, it checks whether the cell is filled with a list object. If not, a new, empty key-value list is created and stored there. The main work of the insertion happens using the `KeyValueList`'s `insert()` method, shown later. Like the tree implementations, the insert method returns a flag indicating whether a new node was created (as opposed to updating an existing key's value). When a new node is created, it increments the number of items and checks the load factor. If it has crossed the threshold, the hash table must be enlarged. It is important to count the number of items in all the lists stored in hash table cells, not the number of lists, as we discuss shortly. After managing the growth of the table, `insert()` can return the `flag` indicating whether a new node was added to the hash table.

The `__growTable()` method starts off the same as for open addressing. It keeps a local variable pointing at the old table while it allocates a new one with a size that's a prime number at least twice the size of the previous one. Then it rehashes each of the items in the old table into the new one. The traversal of the old table is a loop through all its cells. Filled cells must be traversed using the chain's `traverse()` method. The key-value tuples are stored in the `item` variable and then passed as the two arguments to the hash table's own `insert()` method.

LISTING 11-12 The Search and Insert Methods for a Separate Chaining `HashTable`

```
class HashTable(object):           # A hash table using separate chaining
…
   def search(self,                # Get the value associated with a key
              key):                # in the hash table, if any
      i = self.hash(key)           # Get cell index by hashing key
      return (None if self.__table[i] is None else # If list exists,
              self.__table[i].search(key)) # search it, else None

   def insert(self,                # Insert or update the value associated
              key, value):         # with a given key
      i = self.hash(key)           # Get cell index by hashing key
      if self.__table[i] is None:  # If the cell is empty,
         self.__table[i] = KeyValueList() # Create empty linked list
      flag = self.__table[i].insert(key, value) # Insert item in list
      if flag:                     # If a node was added,
         self.__nItems += 1        # increment item count
         if self.loadFactor() > self.__maxLoadFactor: # When load
            self.__growTable()     # factor exceeds limit, grow table
      return flag                  # Return flag to indicate update

   def __growTable(self):          # Grow the table to accommodate more items
      oldTable = self.__table      # Save old table
      size = len(oldTable) * 2 + 1 # Make new table at least 2 times
      while not is_prime(size):    # bigger and a prime number of cells
         size += 2                 # Only consider odd sizes
      self.__table = [None] * size # Allocate new table
      self.__nItems = 0            # Note that it is empty
      for i in range(len(oldTable)): # Loop through old cells and
         if oldTable[i]:           # if they contain a list, loop over
            for item in oldTable[i].traverse(): # all items
               self.insert(*item)  # Re-hash the (key, value) tuple
```

Let's look at the implementation of the `KeyValueList` class. It's a specialized version of the `LinkedList` class from Chapter 5 where every link item holds a (key, value) tuple, as shown in Listing 11-13. The definition starts off by importing the `LinkedList` class and defining some accessor functions for getting keys and values from the links.

The `insert()` method differs from its parent class in the way it handles insertion of duplicate keys. The simple `LinkedList` version always inserts new items at the beginning of the list. The `KeyValueList` must update an existing value if the key is already in the list. It first finds any `link` with a matching key (using the parent class's `find()` method with the `itemKey` function to extract the key from each link's tuple). If no such link is found, it uses the parent class's `insert()` method to put the (key, value) tuple at the start of the list. It returns `True` to indicate the addition of a new item. If a link with a matching key is found, it updates the data for that link with the (key, value) tuple and returns `False` to indicate no additions were made.

LISTING 11-13 The Definition of the KeyValueList Class

```
import LinkedList

def itemKey(item): return item[0]  # Key is first element of item
def itemValue(item): return item[1] # Value is second element of item

class KeyValueList(LinkedList.LinkedList): # Customize LinkedList

   def insert(self, key, value):   # Insert a key + value in list
      link = self.find(key, itemKey) # Find matching Link object
      if link is None:             # If not found,
         super().insert((key, value)) # insert item at front
         return True               # return success
      link.setData((key, value))   # Otherwise, update existing link's
      return False                 # datum and return no-insert flag

   def search(self, key):          # Search by matching item key
      item = super().search(key, key=itemKey) # Locate key + value
      return itemValue(item) if item else None # Return value if any

   def delete(self, key):          # Delete a key from the list
      try:                         # Try the LinkedList deletion by key
         item = super().delete(key, itemKey)
         return item               # If no exceptions, return deleted item
      except:                      # All exceptions mean key was not
         return False              # found, so return False

   def traverse(self):             # Linked list traverse generator
      link = self.getFirst()       # Start with first link
      while link is not None:      # Keep going until no more links
         yield link.getData()      # Yield the item
         link = link.getNext()     # Move on to next link
```

The `search()` method uses the parent class's `search()` method to get the first link with a matching key, if any. It returns the associated value if a link is found and `None` otherwise.

The `delete()` method differs the most from its parent. In the simple `LinkedList` version, `delete()` throws an error if the list is empty or the key is not found. The `KeyValueList` uses a `try except` clause to get the result from the parent `delete()` method. If an item is found and deleted, it returns that item. If an error occurs, it returns `False` to indicate the key was not found.

Finally, the `traverse()` method of `KeyValueList` is nearly the same code as its parent class but uses a `yield` statement to make it a generator that yields the (key, value) tuples in the list.

Returning to the separate chaining `HashTable` implementation, you can define its `traverse()` method, as shown in Listing 11-14, to use that of the `KeyValueList`. The loops are

similar to those used in `__growTable()`. The differences are that the `traverse()` method loops over the current hash table cells instead of the old copy, and it yields the items it finds instead of reinserting them.

LISTING 11-14 The `traverse()` and `delete()` Methods for Separate Chaining

```
class HashTable(object):                  # A hash table using separate chaining
…
   def traverse(self):                    # Traverse the key, value pairs in table
      for i in range(len(self.__table)): # Loop through all cells
         if self.__table[i]:              # For those cells containing trees,
            for item in self.__table[i].traverse(): # traverse
               yield item                 # the tree in-order yielding items

   def delete(self,                       # Delete an item identified by its key
              key,                        # from the hash table. Raise an exception
              ignoreMissing=False):       # if not ignoring missing keys
      i = self.hash(key)                  # Get cell index by hashing key
      if self.__table[i] is not None:     # If cell i is not empty, try
         if self.__table[i].delete(key): # deleting item in tree and
            self.__nItems -= 1            # if found, reduce count of items
            return True                   # Return flag showing item was deleted
      if ignoreMissing:                   # Otherwise, no deletion. If we ignore
         return False                     # missing items, return flag
      raise Exception(                    # Otherwise raise an exception
         'Cannot delete key {} not found in hash table'.format(key))
```

Before we discuss the `delete()` method, let's look at the order in which hash table items will be traversed.

Traversal Order in Hash Tables

What order should a hash table traverse its items? In binary search trees, you have the option to traverse in-order, pre-order, or post-order. Is there a way to do the same in hash tables? The orderings in trees are based on the structure of the tree with parent, left child, and right child nodes. There is no equivalent structure over all the items in the hash table (even though there can be such a structure in a single cell with separate chaining).

The order that the `traverse()` methods shown in Listing 11-14 and Listing 11-8 would yield items is based primarily on their hash address. That's a combination of both the hash function and the size of the hash table. If you wanted the keys to be yielded in ascending order, you would need to either reverse the hash function or collect all the keys and sort them. Reversing a hash function is very hard to do in most cases. Sorting the keys, as you've seen, could take O(N×log N) time. In general, hash tables return the keys in "arbitrary" (unpredictable) order. If the caller needs them in order, it can sort them (by key or value). Interestingly, Python's `dict` hash table returned keys in arbitrary order in early versions. In version 3.7 and beyond, it returns them in insertion order.

Note that in separate chaining, the items within each cell are traversed according to the traversal order of the list or tree that is used. In the `KeyValueList` implementation, the list keeps the items in reverse insertion order. When the hash table grows, however, the rehashing of the items goes through the linked lists and reverses the previous insertion order. That makes the ultimate traversal order very hard to predict. If you use an AVL tree for chaining, then the items in each tree could be traversed in key order, but due to rehashing, they still will be yielded by the hash table's traversal in arbitrary order.

The delete() Method for Separate Chaining

The last method of the separately chained hash table implementation shown in Listing 11-14 is `delete()`. Like `insert()`, the code for `delete()` is simpler than what was needed for open addressing. This method computes the hash address of the key and then uses the `KeyValueList` `delete()` method to remove the item, if the list is present. If that list reports that an item was deleted, it reduces the item count for the hash table and returns `True` to signal the deletion of the item. If minimal memory usage is important, the cell should be set to `None` before returning `True` when the deletion results in an empty list or tree.

If the cell is empty or the tree doesn't find the `key` to delete, the `ignoreMissing` parameter determines whether to return `False` or raise an exception.

Which Type of Chaining Should You Use?

Separate chaining can use a number of secondary structures to store the items in each cell. Lists and trees are very common, and sometimes small arrays (when dynamic allocation of new storage is being avoided). There is no single type that's best for all use cases. Some structures are more efficient than others depending on the hash function and the keys inserted.

The most important factors are the maximum number of items in a cell and the number of times they will be searched. If the hash function is very good, it will distribute the keys evenly among all the table cells. If N items are stored in M table cells, the average number of items per cell is N/M. Note that this is exactly the same as the load factor for the hash table. In the separate chaining implementation, we used a `maxLoadFactor` of 1.0 as the default value. With a good hashing function, the average number of items stored in a cell should be 1, at most. There will be some cells with two items, some with none, and very few with three or more items.

If no cell contains more than three items, it makes sense to use the simplest of structures for separate chaining, the linked list. The items don't need to be kept sorted because you only need to compare at most three keys. Inserting into an unsorted list is normally an O(1) operation. In the case of separate chaining, however, you must search the entire list to see if it is a duplicate key. Searching that list takes O(N)—or O(N/M) in this case—for the expected average length of the list. Maintaining the load factor, N/M, at 1.0 or below means that searching the unsorted list and inserting are both expected to be O(1).

Sorted lists don't speed up a successful search, but they do cut the time of an unsuccessful search in half. As soon as an item larger than the search key is reached, which on average is half the items in a list, the search can be declared a failure. That would become important the longer the lists grow. Deletion times are also cut in half in sorted lists.

If many unsuccessful searches are anticipated, it may be worthwhile to use the slightly more complicated sorted list rather than an unsorted list. An unsorted list, however, wins when insertion speed is more important. An example of that might be the use of a hash table to store all the entries for a "pick 6" lottery. Each lottery participant picks a sequence of six numbers. The hash table is used to store each participant's contact info, so the key is the sequence of numbers, and the data is a list of people who picked that sequence. There could be millions of these sequences, but because there will be only one search for the winning sequence chosen by the lottery managers, there will be very few searches of the hash table (both successful and unsuccessful). There's little point to spending time during the insertion to sort the keys.

The choice of the secondary structure to use in separate chaining can be affected by the choice of hash function too. Although there are many good hash functions, there are also some bad ones. If a particular application either chooses a bad hash function or somehow runs across a group of keys that hash to just one or two addresses in the hash table using that function, the number of items in one cell can grow as large as N. In that unlikely case, a balanced binary tree like the AVL tree could be best. That makes the insert and search operations within each cell O(log N) instead of O(N). This is a degenerate case where the hash function doesn't spread the data over a broad number of cells, so the more efficient tree structures are an improvement over lists.

We return to the question of when to use separate chaining versus open addressing when we discuss hash table efficiency later in this chapter.

Hash Functions

In this section we explore the issue of what makes a good hash function and see how we can improve the approach to hashing strings mentioned at the beginning of this chapter.

Quick Computation

A good hash function is simple, so it can be computed quickly. The major advantage of hash tables is their speed. If computing the hash function is slow, this speed will be degraded. A hash function with many iterations or levels of computation is not a good idea. Many are based on sophisticated math. If they involve a lot of multiplications and divisions, especially on a computing platform without hardware support for those kinds of operations, they could take quite a bit of time.

The purpose of a hash function is to take a range of key values and transform them into index values in such a way that the hash addresses are distributed randomly across all the indices of the hash table. Keys may be completely random or not so random.

Random Keys

A so-called **perfect hash function** maps every key into a different table location. This is rarely possible in practice. A special case happens when the keys are unusually well behaved and fall in a range small enough to be used directly as array indices. For example, a manufacturer gives numbers for each of the parts it creates. The numbers started at 1,000 and go up to the number of things they have ever produced, say 10,000 over the past 50 years. Because they were created to be unique and with no gaps, these could easily

be used directly as array indices without hashing, by simply having a hash function that subtracts 1,000 from the part number. These are unusually well-behaved keys.

If you need to store only a few of these part numbers, say the hundred parts currently kept in inventory, then it's possible to create a perfect hashing function that maps them to unique indices in a smaller array. The perfect hashing function needs to map each of the hundred part numbers to a unique index. You saw the Huffman coding algorithm in Chapter 8, "Binary Trees," which came up with a unique bit sequence for every letter used in a message. Similar techniques can be used to assign a unique index to every part number.

In most applications, however, it's impossible to forecast what keys will be inserted in the hash table. Without knowing the number and type of keys, it's impossible to build a perfect hashing function. So typically, you make assumptions. In this chapter we've assumed that the transformed keys were randomly distributed over a large numeric range. In this situation the hash function

```
index = key % arraySize
```

is satisfactory. It involves only one more mathematical operation, and if the keys are truly random, the resulting indices will be random too, and therefore well distributed. If the keys share some common divisor(s), you can reduce the chance that they cause collisions by choosing `arraySize` to be a prime number (and hoping that prime number is not the common divisor).

Nonrandom Keys

Data is often distributed nonrandomly. In fact, it's very rare to find truly randomly (mathematicians would call it uniformly) distributed data.

Let's consider some examples for keys: a timestamp key such as the milliseconds that have elapsed since a particular point in time and an IP address on the Internet. Typically, these kinds of keys are not uniformly distributed across all the possible values; they concentrate in ranges. The millisecond timestamps may be for events over a short duration, such as log messages on a computer server over the past week, or perhaps for some past events, such as the births of a group of people. Those births are all likely to be concentrated on dates in the past century, not uniformly spread out over tens of thousands of years. Even within the past century, the birth rate rises and falls, leaving an uneven distribution. For the IP addresses, it's rare to get a set of data that samples the addresses from all over the world. Typically, there will be many references to local IP addresses and smaller numbers sprinkled from whatever regions communicated with the computer or network appliance that collects the data.

Many keys that might be used have an internal structure. IP addresses are 32-bit or 128-bit numbers organized into four octets or eight 16-bit words. Various blocks of addresses are reserved for different purposes. Any particular set of keys is likely to have many of its keys from a few blocks, and none from most of them.

Part numbers for manufacturers typically have structure too. Let's look at an example of a system that uses car part numbers as keys and discuss how they can be hashed effectively. Perhaps these part numbers are of the form

033-400-03-94-05-0-535

This number, 033-400-03-94-05-0-535, might be interpreted as follows:

Digits 0–2: Supplier number (1 to 999, currently up to 70)

Digits 3–5: Category code (100, 150, 200, 250, up to 850)

Digits 6–7: Month of introduction (1 to 12)

Digits 8–9: Year of introduction (00 to 99)

Digits 10–11: Serial number (1 to 99, but never exceeds 100)

Digit 12: Toxic risk flag (0 or 1)

Digits 13–15: Checksum (sum of other fields, modulo 1000)

If you ignore the separating hyphens, the decimal key used for the preceding 16-digit part number would be 0,334,000,394,050,535. The keys for the parts are not randomly distributed over all possible numbers. The majority of numbers from 0 to 9,999,999,999,999,999 can't actually occur (for example, supplier numbers higher than 70, category codes that aren't multiples of 50, and months from 13 to 99). Also, the checksum is not independent of the other numbers. Some work should be done to these part numbers to help ensure that they form a range of more truly random numbers.

Don't Use Nondata

The key fields should be squeezed down until every bit counts. For example, the category codes in the car part numbers should be changed to run from 0 to 15 (corresponding to the values 100, 150, …, 850 that appear there). The checksum should be removed from the calculation of the hash because it doesn't add any additional information; it's deliberately redundant. Various other bit-twiddling techniques are appropriate for compressing the various fields in the key into the unique values they represent.

The address in memory of the key or the record containing the key should never be used in the hash function. In other words, if the key is accessed through a reference pointer, don't use the pointer when computing the hash. Use only the part number or other identifying elements referenced by the pointer. The location in memory where the data is stored changes from run to run. Using that location would mean that keys would only match on some runs of the program.

Use All the Data

Every part of the key (except nondata, as just described) should contribute to the hash function. Don't just use the first four digits, last four digits, or some such abbreviation. The more data that contributes to the key, the more likely it is that the keys will hash evenly into the entire range of indices.

Sometimes the range of keys is so large that it overflows the type of integer values that the programming language supports. Most computing platforms support 32-bit and 64-bit integers. Some embedded processors, however, might support only 16-bit or 8-bit. Regardless of the platform size limit, there will be keys with numeric values that go beyond what can be represented in a single machine word. We show how to handle that overflow when we talk about hashing strings in a moment.

To summarize: The trick is to find a hash function that's simple and fast, using all the available data, while excluding the nondata and redundant parts of the key.

Use a Prime Number for the Modulo Base

Often the hash function involves creating a number on a large range and using the modulo operator (%) with the table size to map it to the hash address. You've already seen that it's important for the table size to be a prime number when using a quadratic probe or double hashing. If the keys themselves are not randomly distributed, it's important for the table size to be a prime number no matter what hashing system is used.

To see why a prime number of cells is helpful, consider what happens if many hashed keys are separated in value by some number, X. If X is a divisor of the array size, say ¼ of the size, that large group of keys hash to the same four locations, causing primary clustering. Using a prime table size nearly eliminates this possibility. For example, if the table size were a multiple of 50 in the car part example, the category codes could all hash to index numbers that are multiples of 50 (assuming that code is multiplied into the hash value). With a prime number such as 53, however, you are guaranteed that only keys that hash to multiples of that prime (plus a constant offset) are hashed to the same address. Part numbering and other human-designed schemes rarely use prime numbers like that.

Returning to the example of timestamps as keys, the events represented by timestamps are often periodic. The times can represent things that usually happen at the top of the hour, or every 20 minutes, or every year. The periodic events will create timestamps bunched together around certain peak times. If those peaks are separated by a multiple of the array size, then many keys hash to the same address. Even when they don't land exactly on multiples of the array size, the bunches can create collisions of hash addresses causing clusters in open addressing or long chains in separate chaining.

The moral is to examine your keys carefully and tailor your hash algorithm to remove any regularities in the distribution of the keys.

Hashing Strings

At the beginning of this chapter, you saw how to convert short strings to key numbers by multiplying digit codes by powers of a constant. In particular, you saw that the three-letter word *elf* could turn into the number 3,975 by calculating

$$\text{key} = \mathbf{5}*27^2 + \mathbf{12}*27^1 + \mathbf{6}*27^0$$

This approach has the desirable attribute of involving all the characters in the input string. The calculated key value can then be hashed into an array index in the usual way:

```
index = key % arraySize
```

The `simpleHash()` method shown in Listing 11-3 used a similar calculation but with a base of 256 instead of 27. This calculation allows for many more possible characters in the string (but not all Unicode values).

The `simpleHash()` method is not as efficient as it might be. When hashing strings, it performs the character conversion, raises 256 to the power `i` (the position of character in the string), multiplies them, and adds all products in the `sum` expression:

```
sum(256 ** i * ord(key[i]) for i in range(len(key)))
```

This way of expressing the calculation is concise, but it does some extra work that can be avoided. You can eliminate computing the power of 256 by taking advantage of a mathematical identity called Horner's method. This method states that an expression like

$$var4*n^4 + var3*n^3 + var2*n^2 + var1*n^1 + var0*n^0$$

can be written as

$$(((var4*n + var3)*n + var2)*n + var1)*n + var0$$

The base, `n`, now appears without an exponent, multiplying each parenthesized expression (including `var4`). To convert this equation into loop form, you start inside the innermost parentheses and work outward. Translating this equation to a Python function results in the following:

```
def hashString1(key):                    # Use Horner's method to hash a string
   total = 0                             # Sum contribution of all characters
   for i in range(len(key) - 1, -1, -1): # Go in reverse order
      total = total * 256 + ord(key[i])  # Multiply by base, add char i
   return total                          # Return sum
```

The `hashString1()` function computes the same hash that `simpleHash()` does for a string, but with one multiply and one addition per character in the loop. Raising 256 to a power happens by repeating the multiplications.

This approach is a definite improvement because most processors can perform multiplications in a few clock cycles, whereas raising numbers to a power can take much longer. There are two more changes that can help a little more. Multiplying by a power of 2 is the same as shifting the bits of a binary number to the left by that power. In this case, 256 is 2^8, and you can use the bit shift operator, `<<`, instead of the multiplication:

```
def hashString2(key):                    # Use Horner's method to hash a string
   total = 0                             # Sum contribution of all characters
   for i in range(len(key)):             # Go in forward order
      total = (total << 8) + ord(key[i]) # Shift to mult., add char i
   return total                          # Return sum
```

Bit shifts are supported by every modern processor and are usually faster than multiplication. This example also changes the character index, `i`, to increase from 0 to the last character of the key. This approach saves a tiny bit of time (a subtraction) and makes the code simpler, although it will produce very different hash values than `hashString1()` for the same string (other than palindromes). The strings still get hashed to unique values, but the most significant characters—the ones that change the hash value the most—are on the left for the `hashString2()` function.

The `hashString2()` function provides a more optimized hash function that will compute a unique number very quickly from every string. There's another factor to consider, however, and that is the magnitude of the sum. As you shift bits (or multiply), you eventually create numbers bigger than what fits in a machine word. A 64-bit processor could shift the `total` by 8 bits seven times without overflowing its 64-bit registers. For strings longer than 8 characters (or character points greater than 255), an overflow is likely to occur.

Can we modify the basic approach so we don't overflow any variables? Notice that the hash address we eventually end up with is always less than the array size because we apply the modulo operator. It's not the final index that's too big; it's the intermediate `total` values.

With any arithmetic expression using `+`, `*`, and –, you can apply the modulo operator (`%`) at each step in the calculation. Using the operator this way gives the same result as applying the modulo operator once at the end but avoids overflow, at the cost of adding an operation inside the loop. The `hashString3()` method shows how this looks:

```
def hashString3(key, size):  # Use Horner's method to hash a string
   total = 0                 # Sum without overflowing
   for i in range(len(key)): # Go in forward order, shift, add char i
      total = ((total << 8) + ord(key[i])) % size # and use modulo
   return total              # Return sum
```

Most string hashing functions take this approach (or something like it). Various bit-manipulation tricks can be played as well, such as using a size that is a power of 2. That means the modulo operator can be replaced by AND-ing the total with a bit "mask" (for example, `total & 0xFFFFFFF`). On the other hand, using a power of 2 as the hash table size means that there could be hash table collisions for keys with patterns related to that power of 2, rather than a prime.

You can use similar approaches to convert any kind of string or byte sequence to a number suitable for hashing. Because all data is stored as sequences of bytes, this scheme handles nearly any kind of data.

Folding

Another reasonable hash function involves breaking the key into groups of digits and adding the groups. This approach ensures that all the digits influence the hash value. The number of digits in a group should correspond to the size of the array. That is, for an array of 1,000 items, use groups of three digits each. The **folding** technique is almost like writing the digit string on a strip of paper, folding the paper between every group of K digits, and then adding the numbers now piled on top of one another.

For example, suppose you want to hash 10-digit telephone numbers for linear probing. If the array size is 1,000, you would divide the 10-digit number into three groups of 3 digits, plus a final digit. If a particular telephone number was 123-456-7890, you would calculate a key value of 123+456+789+0 = 1368. The modulo operator can map those sums to the range of indices, 0–999. In this case, 1368 % 1000 = 368. If the array size is 100, you would need to break the 10-digit key into five 2-digit numbers: 12+34+56+78+90 = 270, and 270 % 100 = 70.

It's easier to imagine how this operation works when the array size is a multiple of 10. For best results, however, the array size should be a prime number, or perhaps a power of 2, as you've seen for other hash functions. We leave an implementation of this scheme as an exercise.

Hashing Efficiency

We've noted that insertion and searching in hash tables can approach O(1) time. If no collision occurs, or the separate chains contain one element at most, only a call to the hash function, an array reference, and maybe a link dereference are necessary to find an existing item or insert a new item. This is the minimum access time.

Note that the hash function takes some time to compute, and the amount of time depends on the length of the key and the hashing function. Keys are typically short, perhaps tens of bytes. Because the length of the keys is much shorter than a large N—the number of items stored—you treat the time spent hashing as O(1). When you're considering hashing a large sequence of bytes—say an entire video file—the time spent hashing could become significant, but it still could be small compared the number of items stored (for example, all the videos available on the Internet).

If collisions occur, access times become dependent on the resulting probe lengths or search of a chain. Each cell accessed during a probe or link in a chain adds another time increment to the search for a vacant cell (for insertion) or for an existing cell. During an access, a cell or link must be checked to see whether it's empty and whether it contains the desired item.

Thus, an individual search or insertion time is proportional to the length of the probe, length of the chain, or depth of the tree. This variable time must be added to the constant time for the hash function.

The average probe or chain length (and therefore the average access time) is dependent on the load factor (the ratio of items in the table to the size of the table). As the load factor increases, the lengths grow longer.

Let's look at the relationship between probe lengths and load factors for the various kinds of hash tables we've studied.

Open Addressing

The loss of efficiency with high load factors is more serious for the various open addressing schemes than for separate chaining.

In open addressing, unsuccessful searches generally take longer than successful searches. Remember that during a probe sequence, the algorithm stops as soon as it finds the desired item, which is, on average, halfway through the probe sequence. On the other hand, probing must go all the way to the end of the sequence before it's sure it can't find an item.

Linear Probing

The following equations show the relationship between probe length (P) and load factor (L) for linear probing. For a successful search, it's

$$P = (1 + 1 / (1 - L)) / 2$$

and for an unsuccessful search, it's

$$P = (1 + 1 / (1 - L)^2) / 2$$

These formulas are from Knuth (see Appendix B, "Further Reading"), and their derivation is quite complicated. Figure 11-20 shows the graphs of these equations in the blue (upper) curves. The upper graph shows the probe lengths for successful searches and the lower one shows the lengths for unsuccessful searches.

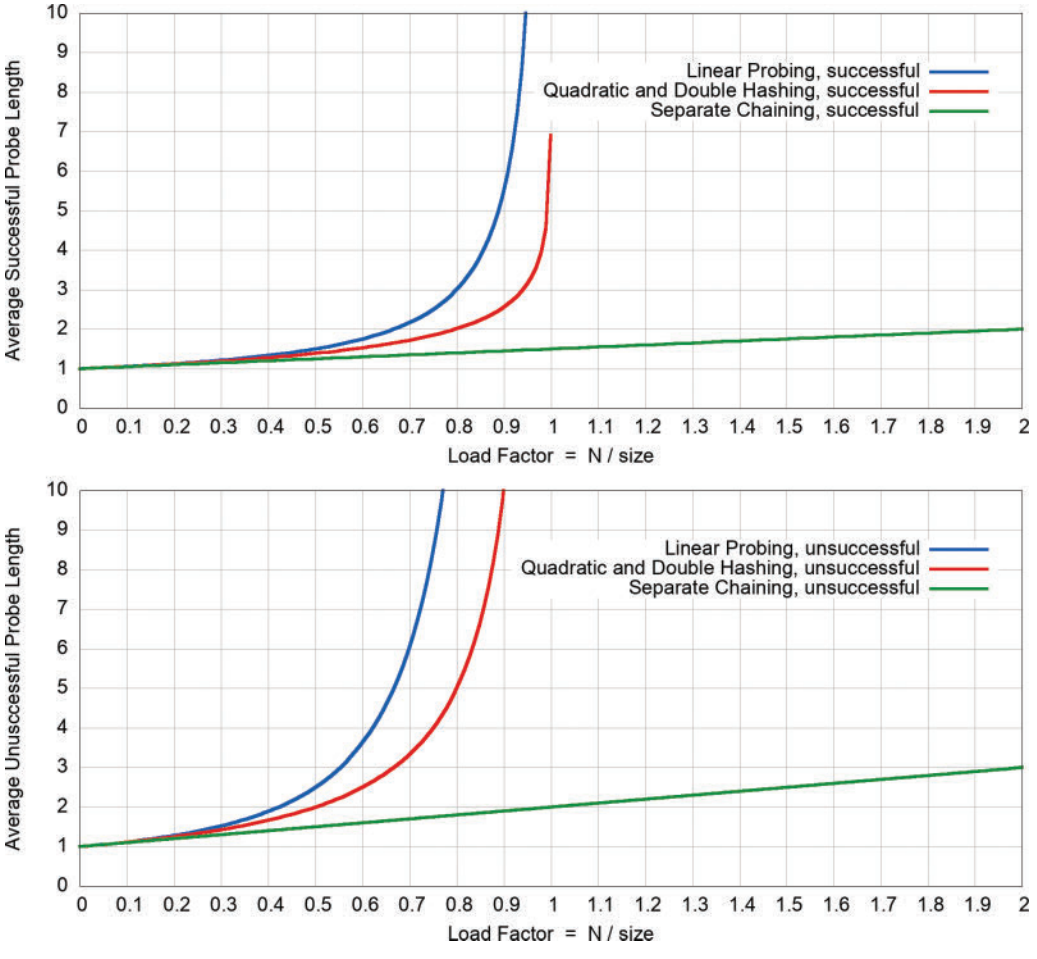


FIGURE 11-20 Successful and unsuccessful probe performance

At a load factor of 0.5, the average successful search takes 1.5 comparisons, and the average unsuccessful search takes 2.5. At a load factor of 2/3, the numbers are 2.0 and 5.0. At higher load factors, the numbers become very large—so high they go off the graph to infinity. We discuss the other lines in the graphs shortly.

The takeaway, as you can see, is that the load factor must be kept under 2/3 and preferably under 1/2. On the other hand, the lower the load factor, the more memory is needed for a given number of items. The optimum load factor in a particular situation depends on the trade-off between memory efficiency, which decreases with lower load factors, and speed, which increases.

Quadratic Probing and Double Hashing

Quadratic probing and double hashing share their performance equations. These equations indicate a modest superiority over linear probing. For a successful search, the formula (again from Knuth) is

$$P = -\ln(1 - L) / L$$

where ln() is the natural logarithm function. This is like $\log_2()$, except the base is the special constant, $e \cong 2.718$. For an unsuccessful search, it is

$$P = 1 / (1 - L)$$

Figure 11-20 shows the graphs of these formulas using red lines. At a load factor of 0.5, successful searches take about 1.4 probes, whereas unsuccessful ones average 2.0. At a 2/3 load factor, the numbers are about 1.6 and 3.0; and at 0.8, they're 2.0 and 5.0. Thus, somewhat higher load factors can be tolerated for quadratic probing and double hashing than for linear probing. That shows up in the graph as the red lines lying below the blue lines.

Note that both the red and blue lines climb steeply as the load factor approaches 1.0. That behavior is expected because it means the table is nearly full, and finding a key or an empty slot can take up to N probes.

Separate Chaining

The efficiency analysis for separate chaining is different, and generally easier, than for open addressing.

We want to know how long it takes to search for a key or to insert an item with a new key into a separate-chaining hash table. All of the methods must compute the hash function and determine a starting hash address. The time taken to compute that is a constant, so we focus on the number of key comparisons needed when searching the chain structure. For chaining, we assume that determining when the end of a list or tree has been reached is equivalent to one key comparison. Thus, all operations require `1 + nComps` time, where `nComps` is the number of key comparisons.

Say that the hash table contains `size` cells, each of which holds a list, and that N data items have been inserted in the table. Then, on average, each list holds N divided by `size` items:

$$\text{Average List Length} = N / \text{size}$$

This is the same as the definition of the load factor, L:

$$L = N / \text{size}$$

Therefore, the average list length equals the load factor.

Searching

In a successful search, the algorithm hashes to the appropriate list and then searches along the linked list for the item. On average, half the items must be examined before the correct one is located. Thus, the search time is

$$P = 1 + L / 2$$

This is true whether the lists are ordered or not. In an unsuccessful search, if the lists are unordered, all the items must be searched, so the time is

$$P = 1 + L$$

These formulas are graphed in Figure 11-20 using the green (lowest) lines. For an ordered list, only half the items must be examined in an unsuccessful search, so the time is the same as for a successful search.

In separate chaining it's typical to use a load factor of about 1.0 (the number of data items equals the array size). Smaller load factors don't improve performance significantly, but the time for all operations increases linearly with load factor, so going beyond 2 or so is generally a bad idea. Of course, the open addressing methods must keep the load factor well below 1.0.

Insertion

On the face of it, insertion of a new key is immediate, in the sense that no comparisons are necessary. Because we chose to not allow duplicate keys, however, any existing chain must be searched to determine whether the key is new or a duplicate. That means that insertion behaves exactly like search plus some constant work to either insert the new item at the end (or beginning) of the list or to update the existing data. The hash function must still be computed, and the data inserted or updated, so let's call the insertion time 1. To stay consistent with the other measures, you can call that a probe length, P, of 1. Finding that the key is new is equivalent to the time taken for an unsuccessful search of an unordered list:

$$P = 1 + L$$

If the lists are ordered or the key exists in the chain, then, as with an unsuccessful search, an average of half the items in each list must be examined, so the insertion time is

$$P = 1 + L / 2$$

Separate Chaining with Binary Search Trees

If binary search trees are used to organize the items in each cell, there are a few differences from separate chaining with lists. If you want to get the benefit of fast search of the binary tree, it makes the most sense to use one of the self-balancing binary search tree structures (for example, AVL, 2-3-4, or red-black trees). They are more complex to code, but the number of comparisons needed in both successful and unsuccessful searches is proportional to the depth of the tree. The average number of items stored in each tree is the load factor, like it was for lists. That means the average depth of the trees is $\log_2(L)$. When the load factor is zero, there still is one probe, so the probe lengths are approximately

$$P = 1 + \log_2(L + 1)$$

Inserting into the tree requires finding where the new key belongs, which takes $1 + \log_2(L + 1)$ steps. Finding an existing key also takes $1 + \log_2(L + 1)$ steps. Searching for a key that is not in the tree stops when there is no child node where the key would normally fit, so it too takes the same number of steps. There are small variations between the exact number of steps in each of these cases.

Compared with the other methods shown in Figure 11-20, the graph for binary search trees would be just below the green line for separate chaining. Both graphs would start at 1.0 and slowly rise as the load factor increases, but the binary search trees would rise more slowly after the load factor becomes greater than 1.0. The difference is so minor for low load factors that the simplicity of chaining with lists outweighs the probe performance. The faster search is a benefit, only if the load factor gets large, or a bad combination of the hash function with the keys being hashed puts large fractions of the N items in a single tree.

Growing Hash Tables

Along with the time spent probing for where to insert a new item, there is also the time spent growing the hash table and rehashing items already stored in it. Both open addressing and separate chaining benefit by keeping the load factor low, so they typically double the hash table size when the load factor exceeds a threshold (that differs for the two types).

How much extra work does reinserting the items cause? Consider the first insertions: when the load factor is low, insertion of a single item happens in O(1) or "constant" time. If you never had to rehash the items, then inserting N of them takes O(N) time. It may not seem intuitive, but allowing hash tables to grow exponentially by doubling in size maintains that O(N) performance.

To see why, let's assume that you start with a table of size 1 and that you double it every time an insertion makes the load factor exceed 0.5 (we are intentionally setting aside the complexity of choosing prime table sizes here). After the first insertion, the table is doubled to two cells, and the one item is reinserted. The second insertion pushes the load factor over the threshold again, and the two items must be reinserted into the four-cell table. The steps are detailed in Table 11-3.

Table 11-3 Reinsertions When Table Size Doubles

Item	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Table size	2	4	8	16	16	16	16	32	32	32	32	32	32	32	32	64
Reinsertions	1	2	3	4				8								16

Item	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Table size	64	64	64	64	64	64	64	64	64	64	64	64	64	64	64	128
Reinsertions																32

At the insertion of the fourth item, the table doubles to hold 16 items, and the 4 items must be reinserted. The fifth item, however, does not cause any doubling or reinsertion. The same conditions hold until you get to the eighth item, when another doubling happens, and the 8 items must be reinserted.

As Table 11-3 shows, the reinsertion work happens every time the number of items reaches another power of 2. There are longer and longer intervals between those expansions. The whole second section of Table 11-3 has no reinsertions until item 32 is inserted. By the time you insert some large number of items, say a million, the number of reinsertions will be

$$1 + 2 + 3 + 4 + 8 + 16 + \ldots + 262{,}144 + 524{,}288$$

The last number in the sum is the largest power of 2 below one million. Moving out the exception for 3 and writing these as powers of 2 makes the sum:

$$3 + 2^0 + 2^1 + 2^2 + 2^3 + 2^4 + \ldots + 2^{18} + 2^{19}$$

The sum of the powers of 2 should look familiar. It is the same as the count of the nodes in a binary tree going from the root to the leaves. As you saw when analyzing the efficiency of binary trees in Chapter 8, that sum of powers of 2 going from 0 to some level, K, can be written as a formula that depends on a power of 2 itself:

$$2^0 + 2^1 + \ldots + 2^K = \sum_{i=0}^{K} 2^i = 2^{K+1} - 1$$

So, the total number of reinsertions is $3 + 2^{K+1} - 1$ or $2 + 2^{K+1}$. That looks as though it could become a very large number, which means inserting N items would take a lot more work than O(N). What you have to remember is the relationship between K and N. When you insert a million items, K is 19; and 2^K is the largest power of 2 less than or equal to N. In other words, K is the integer just below $\log_2(N)$. Substituting $\log_2(N)$ in the equation for the number of reinsertions (and forgetting about getting the integer just below it) leaves

$$number\ of\ reinsertions = 2 + 2^{\log_2(N)+1} = 2 + 2 \times 2^{\log_2(N)} = 2 + 2 \times N = 2(N+1)$$

That means the number of reinsertions grows linearly as N grows. The original N items were inserted with O(N) work and the reinsertions just increase the constant multiplying N, so the overall complexity is still O(N). Even when you grow the table to be the next prime number larger than twice its current size, the same pattern holds. The number of items to reinsert is always less than half the new table size, and the overall sum of reinserted items does not grow faster than O(N).

Hash Table Memory Efficiency

We've noted that hash tables can contain many unused cells under various circumstances. Overall, they still consume O(N) memory to store N items. They need more memory than a simple array because they must store keys along with their associated values (arrays store only the values, and the key is implicit). Keeping the load factor below the thresholds means you need roughly twice as many cells as the number of items, N, but that is still O(N).

We have assumed that the table grows during insertions by doubling its size when needed so that it is never much more than twice N. When you look a bit closer, however, if you first insert P items and then delete some of them to reach N items, open addressing implementations will consume O(P) space. That amount could be significant when P is much larger than N.

For separately chained hash tables, the process of inserting P items and then deleting some to leave N items doesn't waste quite as much memory as open addressing does because deletions within a chain or tree free up the memory that was being consumed. Only deleting the last item in the chain or tree leaves an empty memory cell or empty chain object.

The other item to note is that *traversal time is proportional to the table size*. If many items were deleted from a hash table that once held P items, traversal time would still take O(P) to check all the cells. This is the first data structure we've seen where traversal could be a bit slower than O(N) where N is the current number of items stored. Deletions in hash tables cause the difference in the time efficiency of traversal. It remains O(N), but N is the maximum number of items inserted, not the current number stored.

Open Addressing Versus Separate Chaining

If open addressing is to be used, double hashing is the preferred system (certainly over quadratic probing). The exception is the situation in which plenty of memory is available and the data won't expand after the table is created; in this case, linear probing is somewhat simpler to implement and, if load factors below 0.5 are used, causes little performance penalty.

The number of items that will be inserted in a hash table generally isn't known when the data structure implementation is written, and sometimes not even when the table is created. Thus, in most cases, separate chaining is preferable to open addressing. Allowing the load factor to get large causes major performance penalties in open addressing, but performance degrades only linearly or logarithmically in separate chaining.

When in doubt, use separate chaining. Its drawback is the need for a linked list or binary search tree class, but the payoff is that adding more data than you anticipated won't cause performance to slow to a crawl.

Hashing and External Storage

At the end of Chapter 9, "2-3-4 Trees and External Storage," we discussed using B-trees as data structures for external (disk-based) storage. Let's look briefly at the use of hash tables for external storage.

Recall from Chapter 9 that a disk file is divided into blocks containing many records and that the time to access a block is much longer than any internal processing on data in main memory. For these reasons, the overriding consideration in devising an external storage strategy is minimizing the number of block accesses.

On the other hand, external storage is less expensive per byte, so it may be acceptable to use large amounts of it, more than is strictly required to hold the data, if by so doing you can speed up access time. Hash tables make this speed up possible.

Table of File Pointers

The central feature in external hashing is a hash table containing block numbers, which refer to blocks in external storage. This configuration is similar to the separately chained hash table, but the contents of the table cells point to external blocks rather than a list or a tree in memory. The hash table is sometimes called an **index** (in the sense of a book's index). It can be stored in main memory or, if it is too large, stored externally on disk, with only part of it being read into main memory at a time. Even if it fits entirely in main memory, a copy will probably be maintained on the disk and read into memory when the file is opened.

Nonfull Blocks

Let's assume the same characteristics as the contact database example from Chapter 9 in which the block size is 8,192 bytes, and a record is 1,024 bytes. Thus, a block can hold 8 records. Every entry in the hash table points to one of these blocks. Let's say there are 100 blocks in a particular file.

The index (hash table) in main memory holds pointers to the file blocks. The hash table has indices from 0 to 99. The contents of cell 11, for example, holds a block number in external storage where a group of records are stored. Those records' keys hash to 11.

In external hashing it's important that blocks don't become full. Thus, you might store an average that's about half the maximum capacity, so 4 records per block in this example. Some blocks would have more records, and some fewer. There would be about 400 records in the file. This arrangement is shown in Figure 11-21.

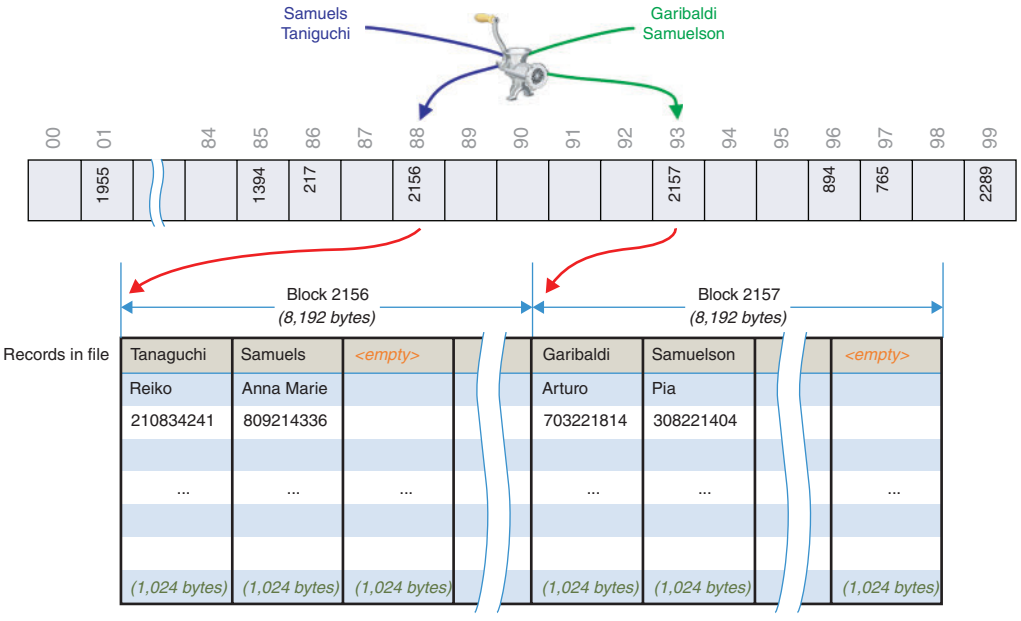


FIGURE 11-21 Hashing to external blocks

All records with keys that hash to the same value are located in the same block. To find a record with a particular key, the search algorithm hashes the key, uses the hash value as an index to the hash table, gets the block number at that index, and reads the block. In the Figure 11-21 example, the name *Samuels* hashes to address 88 in the 100-block index. That contains block number 2156. Reading that block from disk allows the algorithm to retrieve the full record for Samuels. A similar name like *Samuelson* could hash to a very different address, 93 in the example, and be stored in another block. Different names like *Taniguchi* could hash to the same table address, and block, as *Samuels*. A new key might hash to an empty cell that has no block number.

This process is efficient because only one block access is necessary to locate a given item. The downside is that considerable disk space is wasted because the blocks are, by design, not full. The example in Figure 11-21 shows two blocks, each holding only two records. For best performance, the load factor would need to be kept between 0.5 and 1.0.

To implement this scheme, you must choose the hash function and the size of the hash table with some care so that a limited number of keys hash to the same value. In this example, you want only four records per key, on the average.

Full Blocks

Even with a good hash function, a block will occasionally become full. This situation can be handled using variations of the collision-resolution schemes discussed for internal hash tables: open addressing and separate chaining.

In open addressing, if, during insertion, one block is found to be full, the algorithm inserts the new record in a neighboring block. In linear probing, this is the next block, but it could also be selected using double hashing. In separate chaining, special overflow blocks are made available; when a primary block is found to be full, the new record is inserted in the overflow block. Overflow blocks can be chained to allow large overflows, although these impact performance significantly.

Full blocks are undesirable because an additional disk access is necessary for the second and any subsequent overflow block(s); and the disk block access time can be many tens or thousands of times longer than a memory access. Such long access times may still be acceptable, however, if it happens rarely.

We've discussed only the simplest hash table implementation for external storage. One simple improvement on the example shown in Figure 11-21 is to add a record index to the hash table in addition to the block number. That would allow constant time retrieval of the record within the block after it's read for only a few bytes more storage per cell. There are many more complex approaches that are beyond the scope of this book.

Summary

- A hash table is based on an array.
- The range of possible key values is usually greater than the size of the array.
- A key value is hashed to a number by a hash function.
- The hashed number is mapped to an array index called the hash address, typically using a modulo operation.
- An English-language dictionary—where words are the keys and definitions are the values—is a typical example of a database that can be efficiently handled with a hash table.
- The hashing of a key to an array cell filled with a different key is called a collision.
- Collisions can be handled in two major ways: open addressing and separate chaining.
- In open addressing, data items that hash to a full array cell are placed in another cell in the array, chosen by following a prescribed probing sequence.
- In separate chaining, each array element consists of a linked list or binary tree. All data items hashing to a given array index are inserted in that chaining structure.
- We discussed three kinds of open addressing probe sequences: linear probing, quadratic probing, and double hashing.
- In linear probing the step size is always 1, so if x is the hash address calculated by the hash function, the probe goes to x, x+1, x+2, x+3, and so on.
- In linear probing, contiguous sequences of filled cells can appear. They are called primary clusters, and they reduce performance.

- In quadratic probing, the offset from the hash address, *x*, is the square of the step number, so the probe goes to *x*, *x*+1, *x*+4, *x*+9, *x*+16, and so on.
- Quadratic probing eliminates primary clustering but suffers from the less severe secondary clustering.
- Secondary clustering occurs because all the keys that hash to the same value follow the same sequence of steps during a probe.
- All keys that hash to the same value follow the same probe sequence in quadratic probing because the step size does not depend on the key.
- Quadratic probing only visits—or covers—about half the cells in the hash table.
- In double hashing, the step size depends on the key and is obtained from a secondary hash function.
- If the secondary hash function returns a value *s* in double hashing, the probe goes to *x*, *x*+*s*, *x*+2*s*, *x*+3*s*, *x*+4*s*, and so on, where *s* depends on the key but remains constant during the probe.
- The number of probed cells required to find a specified item is called the probe length.
- The load factor is the ratio of the number of data items stored in a hash table to the table size.
- The maximum load factor in open addressing should be around 0.5. For double hashing at this load factor, unsuccessful searches have an average probe length of 2.
- Search times go to infinity as load factors approach 1.0 in open addressing.
- It's crucial that an open-addressing hash table does not become too full.
- A load factor of 1.0 is appropriate for separate chaining.
- At load factor 1.0 a successful search in separate chaining has an average probe length of 1.5, and an unsuccessful search, 2.0.
- Probe lengths in separate chaining using lists increase linearly with load factor.
- Probe lengths in separate chaining using balanced binary trees increase logarithmically with load factor.
- By properly managing the load factor to limit probe lengths and collisions, hash tables have effectively O(1) performance for search, insertion, and deletion of a single item.
- Traversing a hash table takes O(N) time, where N is the maximum number of items inserted.
- Hash tables need O(N) storage and take more space than a simple array takes to store N items.

- A string can be hashed by multiplying the numeric value of each character by a different power of a constant and adding the products.
- To avoid overflow with large numbers, you can apply the modulo operator at each step in the process, if a polynomial function and Horner's method is used.
- Hash table sizes should generally be prime numbers. Using prime numbers helps minimize the chance of collisions without knowing anything about the distribution of keys.
- When hash tables grow exponentially, the cost of inserting N items remains O(N), even though many items must be reinserted at each doubling of the table.
- Hash tables can be used for external storage. One way to do this is to have the elements in the hash table contain disk-file block numbers. The blocks contain a limited number of records such that the load factor is kept low.

Questions

These questions are intended as a self-test for readers. Answers may be found in Appendix C.

1. Using Big O notation, how long does it take (ideally) to find an item in a hash table?
2. A(n) _____ transforms a range of key values into a (possibly large) number, which can be mapped to a range of index values.
3. The typical operation used to map large numeric ranges into small ones is _____.
4. When different keys map to the same index in a hash table, _____ occurs.
5. Open addressing refers to
   a. keeping many of the cells in the array unoccupied.
   b. using a parameter in the hashing function to expand the range of cells it can address.
   c. probing at cell *x*+1, *x*+2, and so on, until an empty cell is found.
   d. looking for another location in the array when the original one is occupied.
6. Searching for a key by testing adjacent cells in the hash table is called _____.
7. What are the first five offsets from the original address in quadratic probing?
8. Secondary clustering occurs because
   a. many keys hash to the same location.
   b. the sequence of step lengths is always the same.
   c. too many items with the same key are inserted.
   d. the hash function maps keys into periodic groups.

9. Double hashing

   a. should use a different hash function than that used for the hash address and compute a step size from the hashed value.

   b. applies the same hash function to the hash address, instead of the key, to get the next hash address.

   c. is more effective for separate chaining than for open addressing.

   d. decreases the search time by a factor of two.

10. Separate chaining involves the use of a(n) _____ or _____ at each hash table cell.

11. A reasonable load factor in separate chaining is _____.

12. True or False: A possible hash function for strings involves multiplying each character value by a number raised to a power that increases with the character's position.

13. The size of the hash table should _____ to minimize the number of collisions, in general.

14. If digit folding is used in a hash function, the number of digits in each group should reflect _____.

15. In which of the open address probing methods does an unsuccessful search take longer than a successful search?

16. In separate chaining with linked lists, the time to insert a new item

   a. increases as the logarithm of the load factor.

   b. is proportional to the ratio of items in the table to the number of table cells.

   c. is proportional to the number of lists.

   d. is proportional to the percentage of filled cells in the table.

17. When hash tables double or more than double in size when insertions exceed a threshold and the items must be rehashed into the array, the overall time taken to insert N items is

   a. O(log N) time.

   b. O(N) time.

   c. O(N×log N) time.

   d. $O(N^2)$ time.

18. Rank order these data structures for their "unused" memory in storing the exact same set of N items: a sorted linked list, an AVL tree, and an open addressing hash table using double hashing and a load factor of 0.6. Unused memory means cells or fields that are allocated but not filled (or filled with `None`) instead of a value or link to another structure.

19. True or False: In external hashing, it's important that the blocks never become full.
20. In external hashing, all records with keys that hash to the same value are located in _____.

Experiments

Carrying out these experiments will help to provide insights into the topics covered in the chapter. No programming is involved.

11-A A person boarding a long train and looking for an empty seat and cars entering a highway are pretty good analogies to the way open addressing looks for an empty cell to hold an item. Can you think of real-world processes that act like separate chaining? Think of cases in which there is some initial choice about where people or things go, followed by hunting through a list based on the first choice to find their final destination. When you think of one, how likely is it to have collisions? Can you think of ways to make the real-world process more efficient based on hash table structures?

11-B This one takes a little math. How many people need to be at a gathering to make it more likely than not that they share a birth month (not a birth *day*)? For this problem, assume that all twelve birth months are equally likely. If there is only one person at the gathering, then they must have a unique birth month. The second person will have a unique birth month with a likelihood of 11/12. The third person will have a unique birth month with a likelihood of 10/12, and so on. The likelihood that all the people have distinct birth months is the product of those likelihoods. Multiply them and find when the combined likelihood of having unique birth months becomes less than 50 percent.

11-C The idea of hashing can be used in sorting. In Chapter 3, "Simple Sorting," and Chapter 7, "Advanced Sorting," we introduced a number of sorting methods. Which ones use hashing or something like it to put the items in order?

11-D With the HashTableOpenAddressing Visualization tool, make a small quadratic hash table with a size that is *not* a prime number, say 24, and a maximum load factor of 0.9, so that it doesn't grow. Fill it very full with, say 20, random items. Now search for nonexistent key values. Try different keys until you find one that causes the quadratic probe to go into an unending sequence. This repetitive sequence happens because the quadratic step size modulo the array size forms a repeating series.

Repeat the experiment, but this time use a prime number for the array size, say 23. Can you find nonexistent keys that cause a similar unending sequence?

11-E With the HashTableChaining tool, create an array with 11 cells and a maximum load factor of 1.99 to allow high density. Next, fill it with 20 random items. Inspect the linked lists that are displayed. What is the longest list? Add the lengths of all these linked lists and divide by the number of lists to find the average list length. On average, you need to search this length in an unsuccessful search. (Actually, there's a quicker way to find this average length. What is it?)

Programming Projects

Writing programs to solve the Programming Projects helps to solidify your understanding of the material and demonstrates how the chapter's concepts are applied. (As noted in the Introduction, qualified instructors may obtain completed solutions to the Programming Projects on the publisher's website.)

11.1 Implement a new method for `HashTable` that finds all the keys that were not placed at their initial hash position within an open addressing hash table due to collisions. Show the counts of displaced keys for the linear, quadratic, and double hash probes in hash tables with maximum load factors of 0.5, 0.7, and 0.9 (in other words, under nine different conditions: three probe schemes times three load factors).

The number of collisions depends heavily on the distribution of keys inserted in the table. You should run tests several times using randomly generated keys because the results will vary with each set. Make sure you use the same set of keys to insert in each of the different hash table types to make a reasonable comparison. You can generate 200 `random` integers in the range [0, 999] by importing the `random` module and evaluating

```
random.sample(range(1000), 200)
```

Using the `random.sample()` function guarantees that there will be no duplicate keys in the sequence. Initialize your hash tables with a size of 103 to lessen the likelihood of some probe sequences being unable to find empty cells in a small table. Run your tests many times to see whether some probe algorithms are clearly better or worse than others.

11.2 Write two hash functions that implement the digit-folding approach described in the "Folding" section of this chapter. One of the functions should fold groups of three digits and the other groups of two digits. Use these functions to create two `HashTable` objects with linear probing and the displaced key counting method from Project 11.1. Write a program that fills these two hash tables with a thousand random 10-digit integers generated by `random.sample(range(10000000000), 1000)`. Show the counts of displaced keys for the two hashing functions and the same three maximum load factors: 0.5, 0.7, and 0.9.

Accessing a group of K digits in a positive number may be easier than you think. Can you generalize the folding hash function to work with any number of digits, or maybe even with any number for the folding range, not just 10^K?

11.3 Explore what happens when the hash table size is a power of 2 instead of a prime number. Rewrite the `HashTable.__growTable()` method so that it doubles the size of the table without finding the next prime number larger than that. Use the same conditions as in Project 11.1, except use a starting size of 128 for the hash tables. The 200 keys that are inserted will force the table to grow at least once, and it should remain a power of 2.

Using a table size that is not a prime number increases the chances of collisions, so much so that you are likely to run into the exception raised by the `insert()` method

when the probe sequence runs out of cells to try (see Listing 11-5). The exception will happen for only some distributions of keys, so you may need to seed the random number generator with different values to cause the exception. Make sure you catch the exception and record the problem for the particular probe sequence and load factor. The same set of keys may work with one probe sequence, but not in others.

As in Project 11.1, show the number of displaced keys for the nine different conditions: three probe schemes times three load factors. If the 200 keys cannot be inserted for a particular condition, show that too.

11.4 The double hashing step size calculation in Listing 11-10 uses the `simplehash()` function. Replace that function with a multiplicative hashing function that is a variation on Horner's method described in the "Hashing Strings" section, except that it's designed to work with integer hash keys. The integer keys can be treated as a sequence of bytes. You can get the lowest byte from a big integer N by using a bit mask in Python, `N & 0xFF`. The loop iterates over the bytes and computes a hash that starts at 0. On each iteration, the current hash is multiplied by a prime, and the low byte plus another prime are added to get the next value of the hash. Multiplying by a prime and adding another prime help spread the influence of each bit of the key across the hashed value.

Produce a table like Table 11-2 showing the insertion of 20 integer keys randomly selected from the range [0, 99999]. Show the multiplicative hashed address along with the modulo with the small prime to derive the step size. You need to write some code to produce the probe sequence and peek at the stored values before inserting the item in the hash table in order to show the last column of the table.

11.5 Hash tables are perfectly suited to the task of counting things like the number of times words are used in a text. By going through the text a word at a time, you can check a hash table to see whether the word has already been seen before. If it hasn't, the word is inserted as a key in the hash table with a value of 1. If it has been seen, the table is updated to hold the incremented count. Traversing the completed hash table gets the overall word counts.

Write a program that reads a text file, extracts the individual words, counts the number of times they occur using a hash table, and then prints out a list of all the distinct words and their counts. To get the lines of a text file in Python, you can use a loop like `for line in open('myfile.text', 'r')`. To get the words from the line, you can use a loop like `for word in line.split()`, which splits the string at whitespace characters. To trim off leading and trailing punctuation from a word, you can use the `strip()` method of strings, as in `word.strip('()<>[]{}-_,.?!:;"')`. This would convert `"(open-addressing!)"` to `"open-addressing"`, for example. For case-insensitive word counting, you can use the `lower()` method for strings to make all the characters lowercase. Show the output of your program running on a short text file.

CHAPTER 12

Spatial Data Structures

IN THIS CHAPTER

- Spatial Data
- Computing Distances Between Points
- Circles and Bounding Boxes
- Searching Spatial Data
- Lists of Points
- Grids
- Quadtrees
- Theoretical Performance and Optimizations
- Practical Considerations
- Further Extensions

So far, all our data structures have stored records containing a key to identify them. That's perfect when you want to find a record that exactly matches a key. What about when you want to identify items by position—for example, finding all the grocery stores within a certain distance of a particular location? For that task, you need a structure that uses two separate numeric keys to identify the location of records that correspond to points in space.

Although this pair of numbers may seem to be a simple extension from one numeric key, the data structures must be dramatically changed to make the kinds of desired operations happen efficiently. The nature of points in space and their distances drives much of the design. We look first at the ways to represent points in space and then at three ways of storing records associated with those points. We wrap up with a discussion of the performance that you can expect from these structures and some extensions needed for other domains.

Spatial Data

Data that has position attributes is called **spatial data**. Spatial data includes Cartesian coordinates and geographic coordinates.

Cartesian Coordinates

The simplest form of spatial data corresponds to points on a flat surface. Each point is represented by a pair of numbers that specify its X and Y coordinates on the surface, usually written as (x, y). These (x, y) coordinates on a flat surface are known as **Cartesian coordinates,** invented by French mathematician René Descartes. For example, if you were programming a simulation of a football game, you could use

Cartesian coordinates to represent the location of each player on the field. Because a football field is a very small piece of the surface of the Earth, you can conveniently pretend that the surface of the field is perfectly flat, so it is appropriate to use Cartesian coordinates.

Cartesian coordinates could be represented using any convenient unit of distance—for example, pixels on a screen, or feet or meters on a playing field. The exact units chosen are important to the application but do not affect the data structures and algorithms if the units are the same for both dimensions.

Geographic Coordinates

Spatial data points on the surface of the Earth are usually represented by a pair of numbers that specify the **latitude** (the angle north or south from the equator) and the **longitude** (the angle east or west from the prime meridian at Greenwich, England). Some applications also require a third value corresponding to the elevation on or above the surface of the Earth. In this chapter, we consider just points that are on the surface of the Earth, that is, at sea level.

Figure 12-1 shows how the globe is divided into hemispheres by the equator in blue and the prime meridian in red. For example, the location of Ottawa, Canada, is approximately (45.30096°, –75.81689°) as shown by the small green dot. The negative longitude indicates that the point is to the west of the prime meridian. The location of Johannesburg, South Africa, is about (–26.17643°, 28.04551°) marked by another dot. The negative latitude indicates that Johannesburg is south of the equator. Latitude, longitude coordinates are known as **geographic coordinates**.

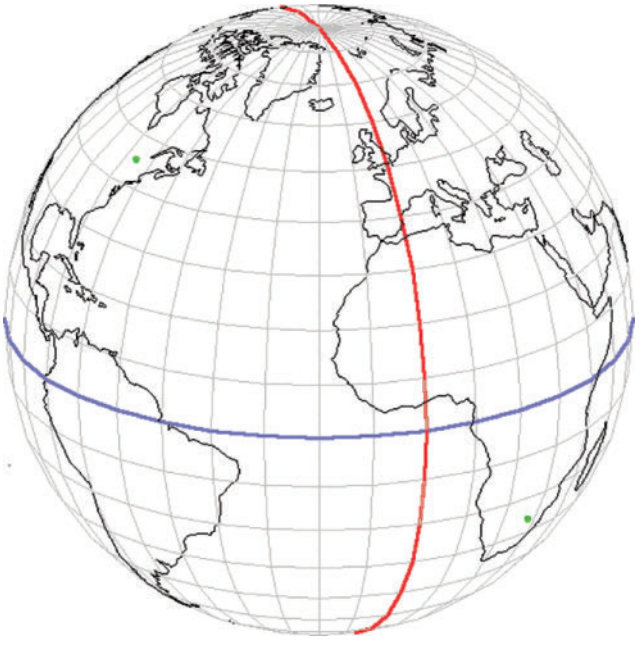


FIGURE 12-1 Geographic coordinates use longitude and latitude

Computing Distances Between Points

Later in this chapter, we look at data structures that support searching for records at or near specified Cartesian or geographic coordinates. To do these calculations, you need to be able to compute the distance between two spatial data points. This calculation is easy to implement for Cartesian coordinates and a little more complicated for geographic coordinates.

Distance Between Cartesian Coordinates

To compute the distance between two points specified by Cartesian coordinates, you make use of the familiar Pythagorean theorem that the length of the hypotenuse squared is the sum of the squares of the sides of a right triangle. Because the points are defined by their coordinates in the X and Y axes, you can align a right triangle with the axes to determine what is sometimes called the **Euclidean distance**. If the two points are at (x_1, y_1) and (x_2, y_2), then the Euclidean distance d between the two points is

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Implementing this distance measure is simple in Python, as shown in Listing 12-1. The result of this formula is in the same units of measurement as the input Cartesian coordinates.

LISTING 12-1 Euclidean Distance Between Cartesian Coordinates

```
# return the Euclidean distance between points (x1,y1) and (x2,y2)
def euclideanDistance(x1, y1, x2, y2):
    dx = x2 – x1
    dy = y2 – y1
    return (dx*dx + dy*dy) ** 0.5
```

Distance Between Geographic Coordinates

Calculating the distance between geographic coordinates is more complex. First, geographic coordinates are specified as angles of latitude and longitude on the Earth—which do not directly correspond to distances expressed in units of length. Second, the surface of the Earth is neither flat nor perfectly spherical, which further complicates the computation. There are many ways that distance between geographic coordinates can be calculated. The accuracy of the calculations depends on the simplifying assumptions and approximations used.

The shortest path between two points on a sphere lies along the curve of a **great circle**. To get the great circle, take the plane that contains the two points and the center of the sphere and intersect it with the sphere itself. Computing the distance between Ottawa and Johannesburg using the Euclidean distance would be useful only if you could travel in a straight line through the Earth. Even a straight line drawn on Figure 12-1 that seems to go directly between the two points is not necessarily the shortest path. To be sure that the

path is the shortest, it must lie on the great circle path that connects them. That is why a flight from New York to Spain flies in a path that curves toward the North Pole, rather than flying due east, as a typical map might imply.

The **haversine formula** computes the great circle distance between two geographic coordinates on a sphere. Although Earth is not a perfect sphere, the haversine formula will give correct answers to within 0.5 percent. That should be adequate for finding the nearest airport or coffee shop, but possibly not accurate enough for other applications. The haversine formula for the distance between two geographic coordinates is

$$d = 2\,R \arcsin\sqrt{\sin^2\left(\frac{\Delta\phi}{2}\right) + \cos\phi_1\ \cos\phi_2\ \sin^2\left(\frac{\Delta\lambda}{2}\right)}$$

where

- R is the radius of the Earth
- ϕ_1 and ϕ_2 are the latitudes of the two points
- $\Delta\phi$ is the difference between the latitudes of the two points
- $\Delta\lambda$ is the difference between the longitudes of the two points.

Although this formula looks rather daunting, the Python implementation is straightforward, as shown in Listing 12-2.

LISTING 12-2 Python Implementation of the Haversine Distance

```
from math import *
RADIUS_OF_EARTH = 6371      # radius in kilometers

# return the haversine distance between (lon1, lat1) and (lon2, lat2)
def haversineDistance(lon1, lat1, lon2, lat2):
    lat1 = radians(lat1)    # trig functions need radians
    lon1 = radians(lon1)
    lat2 = radians(lat2)
    lon2 = radians(lon2)

    dLon = lon2 - lon1      # difference of longitudes
    dLat = lat2 - lat1      # difference of latitudes

                            # Haversine formula:
    a = sin(dLat/2)**2
    if dLon != 0:           # save some trig for dLon == 0
        a += cos(lat1) * cos(lat2) * sin(dLon/2)**2

    # Numerical issues at antipodal points requires min, max:
    return 2 * RADIUS_OF_EARTH * asin(min(1, max(a, 0)**0.5))
```

It is worth noting that as commonly spoken in English, geographic coordinates are referred to as "latitude, longitude" pairs. When you're writing code to uniformly support either Cartesian or geographic coordinates, however, it makes more sense to refer to geographic coordinates as "longitude, latitude" pairs. Simply put, longitude angles correspond to values along the equator, analogous to points along the Cartesian X axis. Likewise, latitude angles correspond to values along the north-south meridians, analogous to the Cartesian Y axis. The methods in the class definitions we present for spatial data structures accept "*a*, *b*" pairs to specify spatial location, which are then interpreted appropriately to mean either *x*, *y* or longitude, latitude coordinates. Although the Euclidean distance will work with any units, this definition of the haversine distance always returns the distance in kilometers because the Earth's radius is specified that way.

Circles and Bounding Boxes

When working with spatial data, you frequently need to determine whether points lie within a particular area. For example, you might want to know whether a particular geographic coordinate lies within a particular country or within a property boundary. Many kinds of regions might be checked. The two simplest and most common ones are circular and box-shaped areas. You will need to use the distance calculations in many cases to determine whether a point is inside or outside one of these regions.

Clarifying Distances and Circles

When you begin the search for a data point nearest to a query point, and no data point has been considered yet, it is convenient to think of the initial distance from the query point to the nearest point as being at infinity—in other words, a number that is arbitrarily large. The Python `math` module conveniently provides a predefined constant `math.inf`, which is larger than any other int or float number. (In versions of Python before 3.5, the constant for infinity was produced by the expression `float('inf')`, which can still be used.) As the search proceeds, and subsequent points are encountered, that distance becomes a finite number, which gets successively smaller as points closer to the query point are identified.

In Cartesian coordinates, at any step of the search, you can think of the query point together with the distance to the nearest point so far as defining a circle around the query point, whose radius is equal to the distance to the current nearest point.

In geographic coordinates, a curve of constant distance is not a perfect circle when drawn on a grid whose latitude lines and longitude lines are spaced at uniform degrees of separation (known as the Mercator projection). Indeed, this is what causes the apparent distortion of landmasses when viewed on some maps. For example, near the poles, the meridians are much closer together than they are near the equator.

These changes in distance are illustrated in Figure 12-2 where three 1,000 km paths have been drawn in blue. The same paths are drawn on the globe as it appears from space and on a map using the Mercator projection with 10 degrees separating each of the longitude and latitude grid lines. The path around Iceland looks much larger than the path centered on the coast of Brazil in the Mercator projection, even though both are equal diameter. The path surrounding the tip of Tierra del Fuego, the southern tip of South America (only partially visible on the globe projection), is also the same diameter.

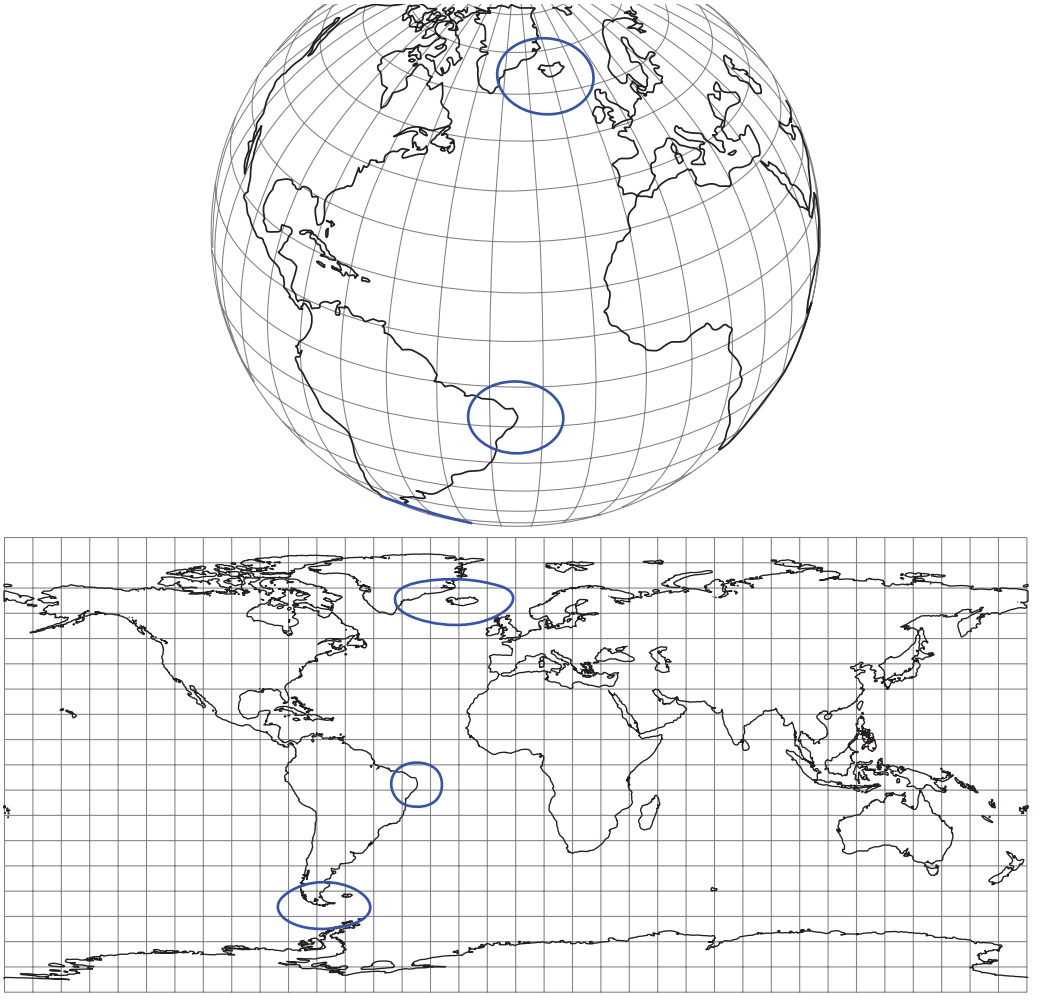


FIGURE 12-2 Comparing 1000 km circles at different points on the globe

In the following pages when we discuss a *query circle*, keep in mind that sometimes the circle is not truly circular. The term *query circle* is used to refer to a curve of constant distance from a query point, which may not always look perfectly circular depending on the map viewpoint. Our computations account for this apparent distortion.

Bounding Boxes

As we just mentioned, when using geographic coordinates, a query circle with a radius of constant distance does not look like a circle when projected on a map grid with uniform latitude and longitude spacing. For geographic coordinate query circles far from the equator, the circle begins to look more like an ellipsoid. In addition, the widest part of the query circle is not directly due-east or due-west of the center of the query circle as it is at the equator or in Cartesian coordinates. For example, as a search circle gets farther north from the equator, the circle gets increasingly flattened, and the widest part of the search circle progresses farther north of the center of the circle.

In the coming pages we examine data structures that partition space into rectangular pieces. To search for a particular point, the algorithms repeatedly check whether a query circle intersects a particular rectangular piece of space containing data points. It can be computationally complex to determine whether a circle and rectangle intersect, and, even more so, whether a distorted ellipsoid and a rectangle intersect.

One way to simplify these repeated query circle-rectangle intersection calculations is to approximate the query circle by a rectangular bounding box that snugly encloses the query circle in either Cartesian or geographic coordinates. As you will see, determining whether two rectangles intersect is a fast and simple computation. If the bounding box of the query circle doesn't intersect a rectangle of space, then you can be certain that there's no chance that the query circle itself intersects the rectangle of space. Only when the bounding box does intersect a rectangle of space do you need to do more detailed calculations to attempt to find a point within that rectangle that matches or is nearby the query point.

The Bounding Box of a Query Circle in Cartesian Coordinates

In Cartesian coordinates, you can easily identify the left, right, top, and bottom coordinates of the bounds of a query circle of radius r, with a center at a, b, as shown in Figure 12-3. The left boundary lies at $a - r$. The right boundary lies at $a + r$. The vertical boundaries are similar.

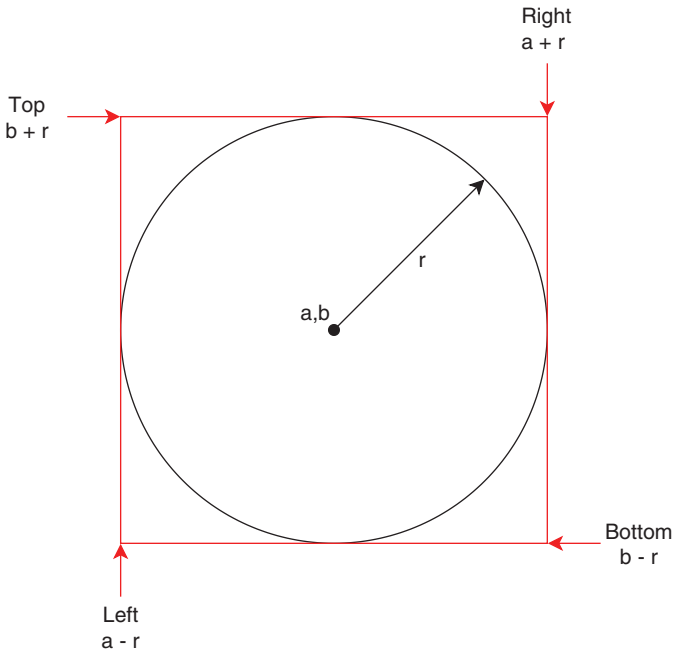


FIGURE 12-3 The bounding box of a circle in Cartesian coordinates

The Bounding Box of a Query Circle in Geographic Coordinates

How do you determine the bounding box of a query circle of radius *r* in geographic coordinates? It's clear that you need to calculate the bounds for longitude and latitude, but how? The formulas that solve this problem (shown here) employ methods in spherical geometry that are outside the scope of this text.

$$\Delta_{long} = \arcsin\left(\frac{\sin\left(\frac{r}{R}\right)}{\sin\left(\frac{\pi}{2} - B\right)}\right)$$

$$\Delta_{lat} = \frac{r}{R}$$

You do not need to understand the detailed math to create the methods that compute and use the bounding box of a search circle in geographic coordinates. The preceding equations

and the diagram in Figure 12-4 show how to compute the width in longitude and height in latitude from the center of the query circle to the edges of its bounding box, where

- r is the radius of the query circle.
- R is the radius of the Earth.
- B is the latitude b of the a, b center of the query circle, expressed in radians instead of degrees.
- (Δ_{long}) is the longitude width in radians from the center to the widest point of the query circle.
- (Δ_{lat}) is the latitude height in radians from the center to the top or bottom of the query circle.

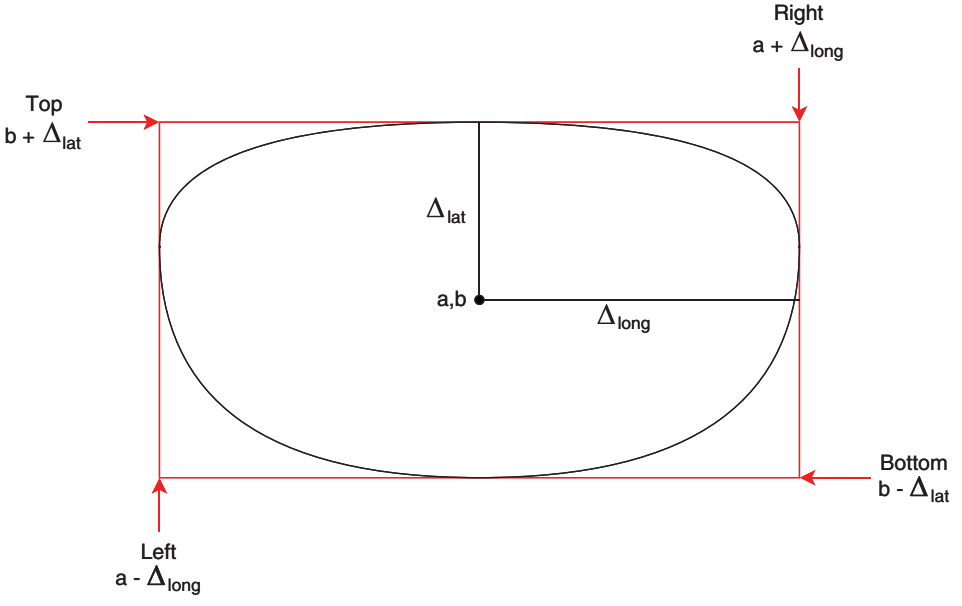


FIGURE 12-4 The bounding box of a query circle in geographic coordinates

Note that the widest point of the query circle may not be due east of the center. The example shown in the figure illustrates this distortion resulting in the widest part of the circle being slightly vertically offset from the latitude, b.

Implementing Bounding Boxes in Python

We represent bounding boxes of all types in Python using the `Bounds` class and its `CircleBounds` subclass, as shown in Listing 12-3.

LISTING 12-3 Creating and Adjusting `Bounds` Objects

```
class Bounds(object):
    def __init__(self, left = -inf, right  = inf,
                       top  =  inf, bottom = -inf):
        Bounds.adjust(self, left, right, top, bottom)

    # mutator to initialize/change bounds in existing object
    def adjust(self, left, right, top, bottom):
        self.__l = left
        self.__r = right
        self.__t = top
        self.__b = bottom

    # return a new Bounds, where some of the
    # current boundaries have been adjusted
    def adjusted(self, left = None, right = None,
                       top = None,  bottom = None):

        if left   == None: left   = self.__l
        if right  == None: right  = self.__r
        if top    == None: top    = self.__t
        if bottom == None: bottom = self.__b

        return Bounds(left, right, top, bottom)
```

Instances of the `Bounds` class shown in Listing 12-3 store the coordinates of the left, right, top, and bottom boundaries of a rectangular section of Cartesian or geographic space. You can create a new `Bounds` object by invoking the constructor as shown in this snippet:

```
b = Bounds(-125.6, -60.8, 49.72, 23.77)
```

The returned object represents the rough rectangular boundaries of the continental United States, with its left boundary at –125.6 degrees longitude, its right boundary at –60.8 degrees longitude, its top boundary at 49.72 degrees latitude, and its bottom boundary at 23.77 degrees latitude.

Invoking `Bounds()` with no arguments results in a rectangle with infinite extent. The mutator method, `adjust()`, enables the client code and the constructor to update the private attributes.

A separate method called `adjusted()` creates and returns a *new* `Bounds` object derived from the current object, and with some or all the boundaries replaced with new values.

The CircleBounds Subclass

The `CircleBounds` subclass of `Bounds`, shown in Listing 12-4, creates a box enclosing a circle.

LISTING 12-4 Initializing and Adjusting `CircleBounds` objects

```
# A bounding box that surrounds a search circle. Properly handles
# distorted circles resulting from geographic coordinates.
class CircleBounds(Bounds):
    # create the bounding box that surrounds the search
    # circle at a,b,r using the specified distance function
    def __init__(self, a, b, r, func):
        # remember the circle's parameters for later
        self.__a, self.__b, self.__r = a, b, r
        self.__func = func

        # get the width/height of the bounding box
        deltaA, deltaB = self.__deltas(a, b, r, func)

        # initialize the superclass's rectangular bounds
        super().__init__(a - deltaA,          # left
                         a + deltaA,          # right
                         b + deltaB,          # top
                         b - deltaB)          # bottom

    # if the circle's radius changed, mutate its bounds
    def adjust(self, r):
        if r != self.__r:
            self.__r = r

            # new dimensions of bounding box
            deltaA, deltaB = self.__deltas(
                self.__a, self.__b, r, self.__func)

            # update the box
            super().adjust(self.__a - deltaA,  # left
                           self.__a + deltaA,  # right
                           self.__b + deltaB,  # top
                           self.__b - deltaB)  # bottom
```

The user creates a new `CircleBounds` object by invoking the constructor with the *a*, *b* coordinates of a circle center, the circle's radius, and a reference to a distance function, as shown in this snippet:

```
c = CircleBounds(38.736, -9.142, 3.14, haversineDistance)
```

This snippet creates a `CircleBounds` object that represents the rectangular bounds surrounding a circle centered in Lisbon, Portugal, with a radius of 3.14 km.

As shown in Listing 12-4, the constructor saves the parameters in private attributes—to potentially be used later by the `adjust()` mutator method that adjusts the radius of the circle. Given the *a*, *b* coordinates and radius *r* of the circle, the constructor obtains the distances from the center of the circle to the widest and tallest points on the circle, called `deltaA` and `deltaB`. Finally, the constructor uses those two distances to invoke the superclass's constructor with the rectangular bounds of the circle.

The `adjust()` mutator can be invoked by the client with a new value for the circle's radius. If `adjust()` determines that the radius has changed, then `deltaA` and `deltaB` are recomputed, and then the superclass's `adjust()` method replaces the rectangular bounds with the newly computed boundaries.

All that remains to discuss is how the `deltaA` and `deltaB` distances are computed, as shown in the `__deltas()` method in Listing 12-5. The method implements the calculations for width and height of the two types of circles shown in Figure 12-3 and Figure 12-4. For infinite radius circles or Cartesian coordinates, `deltaA` and `deltaB` are simply the radius of the search circle. For geographic coordinates, `deltaA` and `deltaB` are computed using the preceding complex formulas and substituted for Δ_{long} and Δ_{lat} in Figure 12-4, respectively.

LISTING 12-5 Computing the Width and Height of a Bounding Box Around a Circle

```
class CircleBounds(Bounds):
…
    def __deltas(self, a, b, r, func):
                # When r is infinity, bounds are always infinite
                # For cartesian coordinates, the edges of the bounding
                # box are r away from the center of the search circle.
        if r == inf or func == euclideanDistance:
            deltaA = deltaB = r

        else:   # geographic coordinates
                # The width of the bounding box is determined by
                # the width of the distorted circle at its widest point.
            alpha = r / RADIUS_OF_EARTH
            gamma = radians(90 - abs(b)) # b is latitude

                # if the circle itself crosses the pole then
                # alpha >= gamma, so we set deltaA as inf,
                # effectively treating the width as if it were inf
            if alpha >= gamma:
                deltaA = inf
```

```
        else:
            # the circle doesn't cross a pole, so get
            # longitude angle from center of circle at
            # the circle's widest width calculated using
            # a spherical right triangle identity.
            deltaA = degrees(asin(sin(alpha) / sin(gamma)))

            # latitude angle directly above/below the
            # center of the circle. This works since above
            # or below is directly on a meridian.
        deltaB = degrees(alpha)

    return deltaA, deltaB
```

Recall that for geographic coordinates, a query circle gets progressively flattened as it approaches either pole. If the center of the circle gets within the query radius of a pole, the circle extends beyond the pole (in other words, the pole lies within the query circle). This condition is detected in the code when the `alpha` variable exceeds the `gamma` variable, which could result in an undefined value of the inverse sine function used to compute `deltaA`. In that case, instead of computing the inverse sine, it sets `deltaA` to be infinite because the search circle wraps all the way around the pole.

Determining Whether Two Bounds Objects Intersect

Later in this chapter in the discussion of the `Grid` and `QuadTree` classes, some methods need to determine whether two rectangles defined by their `Bounds` objects intersect. You can determine whether two rectangles intersect by making the simple observation that they do *not* intersect if one rectangle is fully to the left of the other or fully above the other.

As illustrated in Figure 12-5, rectangles 2, 3, and 4 do not intersect with rectangle 1 because they are each either directly above or to the side of rectangle 1. Rectangle 5, however, does intersect with rectangle 1.

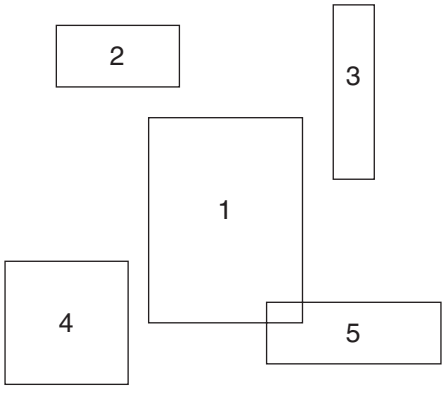


FIGURE 12-5 Intersecting and nonintersecting rectangles

For any two rectangles A and B, you can say that they *don't* intersect if the following pseudocode expression is true:

(A is to the left of B) **or** (B is to the left of A) **or**

(A is above B) **or** (B is above A)

So, to determine whether rectangles A and B *do* intersect, you simply negate that entire expression and simplify, resulting in

(A is NOT to the left of B) **and** (B is NOT to the left of A) **and**

(A is NOT above B) **and** (B is NOT above A)

Examining rectangles 1 and 5, you can see that this latter expression evaluates to True, and therefore that rectangles 1 and 5 *do* intersect.

The code in Listing 12-6 shows the simple implementation of the `intersects()` method. It uses the negation of the definition of *not* intersecting.

LISTING 12-6 Determining if `Bounds` Objects Intersect

```
class Bounds(object):
…
            # return True if bounds of the rectangles intersect.
            # Two rectangles don't intersect if one is totally to
            # the left of or above the other. So we negate to
            # determine if they do intersect.
    def intersects(self, other):
        return not (
            # check if self's box is to left or right of other
            self.__r < other.__l  or  self.__l > other.__r or

            # check if self's box is above or below other
            self.__b > other.__t  or  self.__t < other.__b)
```

Determining Whether One Bounds Object Lies Entirely Within Another

As you will soon see, some methods also need to determine whether the bounds of a circle fit entirely within the bounds of a square subset of cells on a grid.

Figure 12-6 illustrates a sample scenario, where you want to determine whether the circle drawn in black, with its bounding box drawn in red, lies entirely within the blue-shaded square of grid cells. If you have a `Bounds` object initialized with the boundaries of the blue shaded area, and another `CircleBounds` object initialized with the bounding box surrounding the circle, it is simple to determine whether the circle fits within the blue box, as shown in the `within()` method in Listing 12-7.

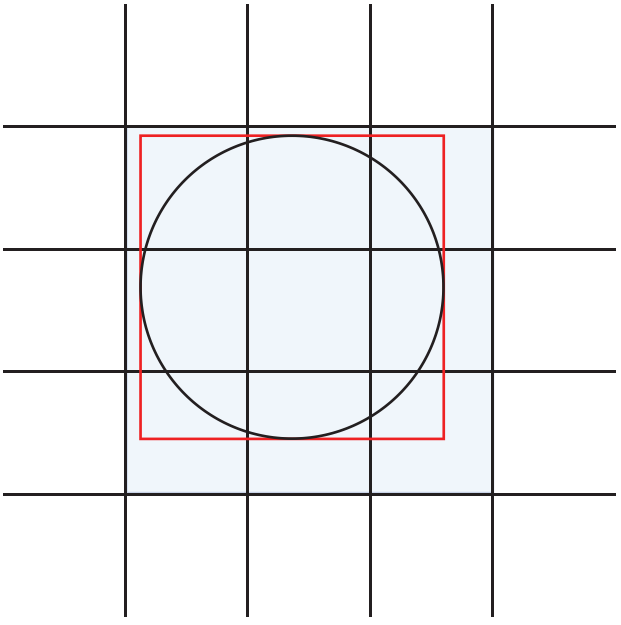


FIGURE 12-6 A `CircleBounds` within another `Bounds` object

LISTING 12-7 Determining Whether a `Bounds` Object Lies Within Another `Bounds` Object

```
class Bounds(object):
…
    # return True if the bounds of self fall within other
    def within(self, other):
        return (self.__r < other.__r and
                self.__l > other.__l and
                self.__t < other.__t and
                self.__b > other.__b)
```

Note that the same definition of `within()` is used for both `Bounds` and `CircleBounds` objects. This is good enough for the purpose of searching the data structures but would not correctly determine whether the *circle* of one `CircleBounds` object was truly within (or even overlapping) the *circle* of another `CircleBounds` object.

Searching Spatial Data

Now that we've handled these geometry basics, we can focus on the interesting challenge of searching a collection of spatial data points to find an exact match to a query point or to find the closest match to a query point. To find the closest match, we need to specify what it means to be close; in other words, which distance function should be used to compare a query point to a candidate data point? If the spatial data consists of Cartesian

coordinates, the `euclideanDistance()` function will be appropriate. If the spatial data consists of geographic coordinates, the `haversineDistance()` function will be appropriate.

In the following sections, we look at several classes that implement data structures and algorithms to support exact match and closest match spatial search. The classes vary in code complexity and Big O performance, so their applicability depends on the amount and distribution of data you are attempting to process. The important issues are how to deal with using coordinates as keys and how to deal with distance when searching. First, we start with a simple approach. Then we add more complex structures to improve performance. Finally, we look at the trade-offs between the different approaches.

Lists of Points

If you want to support spatial data search of a small number of points, it can be appropriate to just store the spatial data points in one of the structures you've already seen, such as an array or a list. Then the spatial data is queried by traversing over the items to find either an exact match or the closest match to the query point. This brute-force approach is the simplest possible solution, though because of that simplicity, the Big O performance is O(N) to search for an exact or nearest match. In other words, to do a spatial search using brute force, the code must compare the query point to every point stored in the structure.

Creating an Instance of the PointList Class

We'll use a simple Python list (array) to hold a collection of items in a `PointList` class. Client code creates an instance of the `PointList` class by invoking the constructor to create an object that contains Cartesian points:

```
b = PointList('cartesian')
```

or that contains geographic points:

```
b = PointList('geographic')
```

If the constructor is passed any string other than `'cartesian'` or `'geographic'`, an exception will be raised.

LISTING 12-8 The `PointList` Constructor

```
class PointList(object):
    def __init__(self, kind):     # must specify type of coordinate
        if kind == 'cartesian':
            self.__distance = euclideanDistance
        elif kind == 'geographic':
            self.__distance = haversineDistance
        else:
            raise Exception("Invalid kind of coordinate")

        self.__points = [ ]       # keep track of inserted points
```

Listing 12-8 shows the code for the `PointList` constructor. There are just two private attributes: a reference to the distance function appropriate to either Cartesian or geographic coordinates, and a reference to an initially empty list that will be used to store the points in the object.

Inserting Points

Points are inserted into the `PointList` object using the `insert()` method. For Cartesian points, the user passes to the `insert()` method an *x* coordinate, a *y* coordinate, and a reference to whatever data will be associated with the Cartesian *x*, *y* point:

```
b.insert(34, 56, "Player 1")  # insert a point at x == 34, y == 56
```

Likewise, for geographic points, the user passes to `insert` a longitude, a latitude, and a reference to the data associated with this geographic longitude, latitude point:

```
b.insert(-73.987853, 40.748440, "Empire State Building")
```

Most likely, the client code will use a loop to repeatedly invoke `insert()` so that the `PointList` object contains all the points to be searched.

The implementation of the `PointList.insert()` method is quite simple (see Listing 12-9). Each inserted point is stored as a tuple containing three elements appended to the end of the private `__points` list. Each tuple contains

- The *x*, or longitude, coordinate
- The *y*, or latitude, coordinate
- A reference to the data to be associated with the specified point

LISTING 12-9 Inserting a Point into a `PointList`

```
class PointList(object):
…
    def insert(self, a, b, data):
        # Loop through the points looking for an exact match
        for i in range(len(self.__points)):
            p = self.__points[i]
            if p[0] == a and p[1] == b:         # Replace data
                self.__points[i] = (a, b, data) # for a duplicate
                return

        self.__points.append((a, b, data))      # not there, so append
```

For all the data structures in this chapter, the `insert()` method ensures that there are no points stored with duplicate *a*, *b* coordinates. Thus, for `PointList`, `insert()` loops through all the points in the list, and if it finds an already-existing point with the same *a*, *b* coordinates, the data currently stored for that point is replaced with the newly inserted data.

If no duplicate is found, then the insertion is accomplished by appending a point to the end of the `__points` list. When the client later attempts to find an exact or nearest match, the `findExact()` or `findNearest()` methods will return the single point that is found to match the query point, and the returned data will be the data that had been most recently inserted for those coordinates. Because `insert()` loops over all the points in the list to avoid inserting a point with duplicate coordinates, it takes O(N) time for each insertion.

Finding an Exact Match

You call the `findExact()` method to search for the point contained in the `PointList` object that exactly matches the specified *x*, *y* or longitude, latitude coordinates. If an exact match is found, the method returns a reference to the data that was stored for the matching point:

```
>>> ans = b.findExact(-73.987853, 40.748440)
>>> print("Answer was:", ans) # Answer was: Empire State Building
```

Not surprisingly, the implementation of `findExact()` shown in Listing 12-10 is simple. The method loops through potentially every point stored in the `__points` list. If an exact match to the specified *a*, *b* coordinates is found, then the data for that point is returned to the client.

LISTING 12-10 Finding an Exact Match in a `PointList`

```
class PointList(object):
…
    def findExact(self, a, b):          # Return data for exact point
        for p in self.__points:         # Loop through all points
            if p[0] == a and p[1] == b: # found exact match
                return p[2]             # so return its data
        return None                     # Couldn't find a match!
```

Because `findExact` must potentially loop through every point of the `__points` list in the worst case, the complexity of the method must be O(N).

Deleting a Point

It is straightforward to implement a method to delete from the `PointList` a point specified by a pair of coordinates. As shown in Listing 12-11, the approach is very similar to that used by `findExact()`, except that after the matching point is identified, it is removed from the list using Python's `del` operator. If a matching point is found, then `delete()` returns the data associated with the deleted point. Otherwise, `delete()` returns `None`.

LISTING 12-11 Deleting a Point from a `PointList`

```
class PointList(object):
…
    # delete the point at a,b
    # Return the deleted point's data if found, or None
    def delete(self, a, b):
        # Loop through the points looking for an exact match
        for i in range(len(self.__points)):
            p = self.__points[i]
            if p[0] == a and p[1] == b: # found a match
                del self.__points[i]    # delete the point
                return p[2]             # return its data

        return None                     # point wasn't there
```

Just like `findExact()`, the Big O time for `delete()` is O(N).

Traversing the Points

Because `PointList` is implemented as a simple list of tuples, traversing the points is trivial. As with the other data structures you've seen, `traverse()` is implemented as a generator as shown in Listing 12-12.

LISTING 12-12 The `traverse()` method of `PointList`

```
class PointList(object):
…
    def traverse(self):
        for p in self.__points:         # yield each of the tuples
            yield p
```

The implementation loops through each of the elements of the list, yielding the tuples containing the *a*, *b* coordinates and the associated data. Because every point in the list is visited once, the complexity of `traverse` is O(N). The caller can use tuple-assignment in its `for` loop to assign the values to three variables, or just assign the returned tuple to a single loop variable.

Finding the Nearest Match

You call the `findNearest()` method to find the point stored in the `__points` list that lies closest to the query point. Depending on whether the `PointList` object stores Cartesian or geographic coordinates, `findNearest()` automatically uses the appropriate function to compute the distance from the query point to each of the object's points, as shown in Listing 12-13.

LISTING 12-13 Finding the Nearest Point in a `PointList`

```
class PointList(object):
…
    def findNearest(self, a, b):     # find closest point to (a,b)
        if len(self.__points) == 0:  # No points yet?
            return None

        distance = inf               # Assume no nearest point
        for p in self.__points:      # For each point
            newDist = self.__distance(a, b, p[0], p[1])
            if newDist == 0:         # Nothing could be closer!
                return p[0], p[1], p[2], 0
            if newDist < distance:   # If it is closest so far,
                distance = newDist   # remember the distance,
                answer   = p         # and the point

        return answer[0], answer[1], answer[2], distance
```

The method returns to the client a four-element tuple containing the *x*/longitude, *y*/latitude, the associated data, as well as the distance to the point that is closest to the query point. The distance function that was set when the object was constructed determines the units of the measurement.

For example, assuming that a geographic point for the location of the Eiffel Tower has already been inserted in the `PointList` object, this snippet shows what would be returned by an invocation of `findNearest()`:

```
>>> ans = b.findNearest(2.2, 48.8)
>>> print(ans)  # (2.294694, 48.858093, 'Eiffel Tower', 9.47495437535)
```

The implementation of `findNearest()` shown in Listing 12-13 starts by checking the base case of an empty list. If the `PointList` is empty, there is no nearest point, so `None` is returned. If there are some points, it initializes the `distance` variable to infinity and then loops through all the points. If the distance from *a*, *b* to one of the new points is less than the current best `distance`, it updates the `answer` and then records the new best `distance`.

This approach is not efficient because it is necessary to compute the distance between the query coordinates and every point stored in the `PointList` object's `__points` list. The only shortcut happens when the query point has the exact same coordinates as one of the points in the `__points` list, in which case there would be no reason to keep looping through the remaining points and it returns the current point as the answer. The complexity of `findNearest()` is also O(N) for each invocation.

Grids

The `PointList` class is appropriate to store and query only a relatively small number of spatial data points. If you have many points and they are mostly uniformly spread out in space, then you can superimpose a grid on top of the points to speed up the `insert()`, `findExact()`, and `findNearest()` methods. To see why this is so, consider Figure 12-7, which shows a collection of 2,000 spatial data points, with a query point shown as a red diamond.

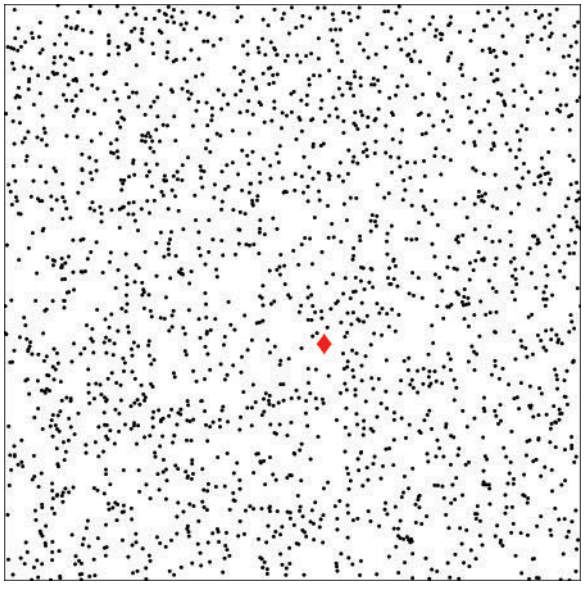


FIGURE 12-7 Two thousand data points and a query point

To find a point that exactly matches or is closest to the red query point, every one of the 2,000 points must be compared to the red query point. If you superimpose a grid on top of the points, you can break them up into 10 rows and 10 columns of cells, as in Figure 12-8.

To find whether there's an *exact* match to the red query point, you only need to check the points that fall within the same cell as the query point. Finding the *nearest* point to the red query point is more complicated because you might end up checking the points in some neighboring cells. Even in this case, it is only necessary to check a small fraction of the total number of points.

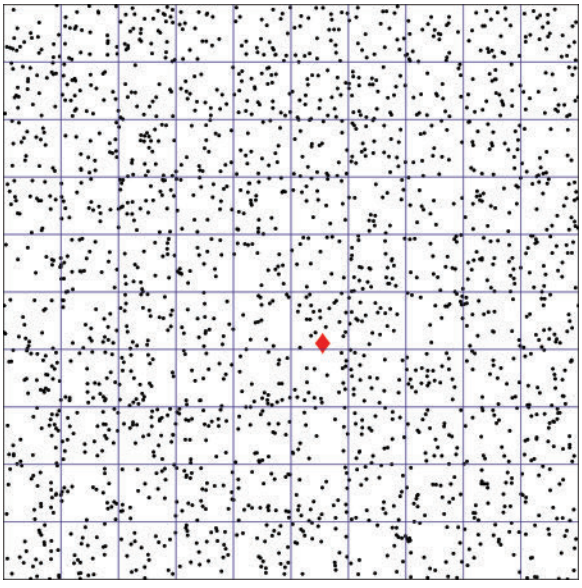


FIGURE 12-8 Breaking up space into a 10 × 10 grid

Implementing a Grid in Python

Transforming Cartesian or geographic coordinates to the row and column numbers of a grid only requires that the coordinates are divided by the cell size (which is specified in units appropriate to the type of coordinates). The grid cells can be represented as an array to quickly access the list of points associated with each cell.

A two-dimensional array seems like the natural choice to hold the separate point lists. The two coordinates can be computed separately and used to index a specific cell in the array. Some issues arise, however, in dealing with negative values and the range of possible coordinates. Most programming languages do not allow negative indices in arrays, so if you converted –23.75 to, say, –24, you couldn't use that to index a row or column.

To work around that limitation, you need to know the full range of possible coordinates that might be used. Knowing the bounds of the coordinates, you could subtract the minimum value of each coordinate from any given coordinate being processed to get a zero or positive value. The positive value can then be converted to an index in the array.

Knowing the coordinate bounds solves one problem but presents another. The bounds can be much larger than what's needed for a particular set of points. In geographic coordinates, the bounds are well known. Longitude ranges from –180 to +180, and latitude from –90 to +90. Let's assume that you want to track the position of every known ship on Earth in a data structure. If you organize them in a two-dimensional array, they will only be placed in some of the cells. Every cell that contains only land will be empty (except for

maybe a ship being transported over land on a trailer). Even among the cells representing oceans and seas, there could be many cells without a single ship. That means a two-dimensional array could have a significant fraction of unused cells.

As you saw in Chapter 11, hash tables can deal with both issues. The key provided to a hash table can include negative numbers or even a sequence of numbers like the *a*, *b* coordinates of a point. The hashing function transforms the coordinate pair into a large number and then maps the number to one of the cells in its table. As more items are stored in the hash table, the memory used to store it can grow while limiting the unused cells to a certain percentage of the total cells. Because the bounds on the coordinates are unimportant, a hash table can represent an infinite grid with positive and negative row and column numbers. This capability makes it well suited to managing the points in the grid data structure.

The next data structure for storing point data, `Grid`, uses a Python dictionary (hash table) whose key is a tuple containing the grid row number and column number (not the coordinates directly). The data stored in each grid cell is a list of points, just like the `list` used in `PointList`. When a dictionary is used to implement the storage of the grid, unused cells are eliminated, negative grid coordinates are handled well, and there is no need to determine the extreme bounds of the coordinates.

Creating an Instance of the Grid Class

You can create an instance of the `Grid` class by invoking the appropriate constructor to create an object that contains either Cartesian points or geographic points. The invocation must include an extra parameter, which is the width and height of each cell. For example, the following snippet creates a `Grid` instance that will contain points with Cartesian coordinates and with grid cells that are 10 units wide and high:

```
g = Grid('cartesian', 10)
```

Likewise, the following code creates a `Grid` instance that will contain points with geographic coordinates, where each grid cell is 1.2 degrees longitude wide and 1.2 degrees latitude high:

```
g = Grid('geographic', 1.2)
```

Note that the grid cell size is measured in degrees while the distances between points are measured in kilometers. The rationale for this difference may not be easy to see. Geographic point coordinates are provided in degrees. To find the appropriate grid cell for a single one of these points, the `Grid` instance will need to determine the cell from the two degree values. It is easier to define the grid size in those units. The distance measure remains the `haversineDistance()` function and `Grid` methods that calculate distances will return values in kilometers.

Like the `PointList`, if the `Grid` constructor is passed any string other than `'cartesian'` or `'geographic'`, an exception will be raised, as shown in Listing 12-14.

LISTING 12-14 The `Grid` Constructor

```
class Grid(object):
    def __init__(self, kind, cellSize):
        if cellSize <= 0:
            raise Exception("Cell size must be positive")
        self.__cellSize = cellSize

        if kind == 'cartesian':
            self.__distance = euclideanDistance

        elif kind == 'geographic':
            self.__distance = haversineDistance

        else:
            raise Exception("Invalid kind of coordinate")

        # dict key: (row, col) tuple
        # dict data: list of (a, b, data) tuples
        self.__cells = dict()
```

There are three private attributes: the size of each grid cell (`__cellSize`), a reference to the distance function appropriate to either Cartesian or geographic coordinates, and a reference to an initially empty hash table (dictionary) that will contain the cells of the grid (`__cells`). The constructor initializes all these attributes based on the arguments provided.

Inserting Points

Points are inserted into the `Grid` object using the `insert()` method. From the client's perspective, points are inserted in a fashion identical to the `PointList` insert. The `insert()` method is passed the *x*, *y* coordinates (or longitude, latitude coordinates) and an associated data item, as shown in Listing 12-15.

The private method `__getCell()` determines the row and column of the cell to contain the inserted point by dividing each of the coordinates by the cell size and then taking the floor (that is, the next lowest int) of the result. (The `floor()` function is provided by Python's math module.) The cell's row, column coordinates are stored as a tuple in the `cell` variable. Because this could be the first point in the cell, the `insert()` method first checks to see whether a list of points is already associated with the cell using the expression `cell in self.__cells`. That expression tests whether the `__cells` hash table has an entry for the key, `cell`. If nothing is stored yet at that cell, the `else:` clause just creates a new grid cell in the `__cells` dictionary and assigns to it a new list containing only the `(a, b, data)` tuple.

If the cell already exists, it is necessary to determine whether a point with the same *a*, *b* coordinates has already been inserted in the cell, by looping through all the `(a, b, data)` tuples stored in the cell's list. Like insertions in `PointList`, if a duplicate is found,

then the data already stored for that point in the list is replaced. But, if a duplicate is not found, then the new `(a, b, data)` point is appended to the cell's list.

Insertion is accomplished by possibly creating a new entry in the dictionary to contain a new list that is an O(1) operation or by checking an existing cell's list for a duplicate and possibly appending a point to the end of the list which is an O(N) operation. The grid cell contains only a portion of the N items stored in the full structure, but that number is proportional to the total number of items. The resulting complexity for `insert()` is therefore O(N). We discuss this further in the "Big O and Practical Considerations" and the "Theoretical Performance and Optimizations" section.

LISTING 12-15 The `Grid` Insert Method

```
class Grid(object):
…
    # inputs: x,y  or  long,lat
    # returns row,col tuple specifying grid cell
    def __getCell(self, a, b):
        col = floor(a / self.__cellSize)
        row = floor(b / self.__cellSize)
        return row, col

    # Insert either an x,y,data or longitude,latitude,data point
    def insert(self, a, b, data):
        cell = self.__getCell(a, b)           # which cell contains point?
        if cell in self.__cells:              # existing cell?
            c = self.__cells[cell]
            for i in range(len(c)):
                p = c[i]                      # For each point in cell
                if p[0] == a and p[1] == b:   # replace data for
                    c[i] = (a, b, data)       # a duplicate
                    return

            c.append((a,b,data))              # append new point to cell
        else:
            self.__cells[cell] = [(a,b,data)] # create new cell
```

Finding an Exact Match

The client uses the `findExact()` method to search for the point contained in the `Grid` object that exactly matches the specified *x*, *y* or longitude, latitude coordinates. If an exact match is found, the method returns a reference to the data that was stored for the matching point. The client uses `findExact()` in the same fashion as with the `PointList` class.

Two steps are needed to find an exact match in the grid. First, the *a*, *b* coordinates are transformed into the row, column of the corresponding cell by calling `__getCell()`, as shown in Listing 12-16. If a point with *a*, *b* coordinates exists in the `Grid`, it must be

found in that cell. The Python expression `__cells.get(cell, None)` looks to see whether the `__cells` dictionary has a value stored for the `cell` key and returns it, if so. It returns the second argument, `None`, when the key is not in the dictionary. The result is put in the `points` variable.

Next, if the cell contains any points, the `findExact()` method loops through each of them. If the exact match is found, then the method returns the associated data. If no match is found, then the loop ends and the method just returns `None`.

LISTING 12-16 Finding an Exact Match in a `Grid` Object

```
class Grid(object):
…
    # Find the data associated with exact match of coordinates
    def findExact(self, a, b):
        cell = self.__getCell(a, b) # which cell contains a,b
        points = self.__cells.get(cell, None) # get list of points
        if points:                  # there are points for the cell
            for p in points:        # check each point and seek a match
                if p[0] == a and p[1] == b:
                    return p[2]     # return the data for exact match
        return None
```

Big O and Practical Considerations

How efficient is `findExact()`? There is a theoretical answer and then a more practical answer that suggests that one must also consider a pragmatic, engineering perspective.

From a theoretical perspective, both `findExact()` and `insert()` are O(N). Let's review why. The first step in both methods transforms the *a*, *b* coordinates into a row, column pair that specifies the cell at which the *a*, *b* point might be stored. This can be done in O(1) time. Now that you know which cell might contain the search point, the code loops through all the points stored in the cell's list looking for an exact match. The Big O of this step depends on how the length of each list grows as the overall number of points N grows. Suppose that the grid had five rows and five columns, and suppose that the points are uniformly spread across the 25 grid cells. Then you can expect that there would be about N/25 points stored in each cell. Because you throw away the constant factor in analyzing complexity, looping through N/25 points to find a match takes O(N) time.

This scenario seems somewhat extreme. For example, if you have 1 million points, a five-by-five grid isn't appropriate. By taking a pragmatic approach, if the grid instead had 500 by 500 cells, then the million points would be spread out to about 4 points per each of the 250,000 cells, which surely seems like an O(1) operation to do an exact find! But from a theoretical perspective, the algorithm remains O(N) because Big O describes how the computation time increases *as N grows larger*. If you were to keep the

grid fixed at 500 × 500 cells, and then allow N to grow from 1 million points to 10 million, you would expect the computation time to likewise increase linearly by a factor of 10.

From a practical perspective, if you know that you have a fixed, or even very slowly growing, number of points, then it can be appropriate, if you have enough memory, to engineer your grid to contain enough cells so that each cell contains a tiny number of points, resulting in effectively constant time to process the cell's points. This approach offers the possibility of a classic engineering trade-off between time efficiency and space or memory efficiency for a specific scenario of known size. In the example of a 500 × 500 grid storing a fixed amount of 1 million points, you have effectively used O(N) memory for cells and lists to obtain an *approximation* of O(1) time performance in both `findExact()` and `insert()`. Even with that engineered solution, however, it is important to remember that the exact find operation is still O(N) for a fixed grid size as the number of points N grows.

In the next section, we consider quadtrees, which have a much more attractive O(log N) time complexity to find an exact match. Does that mean you will always want to use quadtrees? Not necessarily.

When coding actual applications, you often know the practical range of amounts of data points you may have to process. Toward the end of this chapter, we take a practical look at how to choose among spatial data structure solutions by timing them on different sized test data. As you will see, the most theoretically attractive solution doesn't always result in the shortest run time.

Deleting and Traversing

We leave it as a Programming Project for you to implement a method to delete from the `Grid` a point specified by a pair of coordinates. The approach should be similar to that used by `findExact()`, except that after the matching point is identified, it is removed from the cell's list. Just like `findExact()`, the Big O for `Grid`'s `delete` is also O(N).

Traversing the grid is straightforward as long as deletion correctly removes points from the cells and empty cells are skipped. Because the `Grid` is implemented as grid cells containing simple lists of tuples, you can implement the `traverse()` method as a generator using two nested loops. Listing 12-17 shows the implementation.

LISTING 12-17 The `traverse()` Method of `Grid`

```
class Grid(object):
…
    def traverse(self):
        for cell in self.__cells:        # for each cell in the grid
            for p in self.__cells[cell]: # for each point in the cell
                yield p                  # yield the point
```

The `traverse()` generator loops through each of the nonempty cells, getting the cell key from the `_cells` hash table. That key is the row, column tuple of the cell. The inner loop steps through all the points stored in that cell, yielding each point. The points are tuples that contain the *a*, *b* coordinates and the associated data. The caller can use tuple assignment in the `for` loop to assign the values to three variables or just assign the returned tuple to a single loop variable. This method has O(N) complexity (assuming the hash table holding the grid cells is also O(N) for traversal).

Finding the Nearest Match

Sometimes it is fast and easy to find the nearest match to a query point. The method can start just like `findExact()`, the coordinates of the query point are converted into a grid cell's row, column coordinates, and then all the points stored in that grid cell are checked to identify the point that is closest to the query point. Figure 12-9 shows a portion of the grid, the query point in red, and a circle drawn around the query point to show the distance to the nearest point in that grid cell. It is clear from the figure that there is no need to search any further because no point outside that grid cell could possibly be any closer than the current closest point.

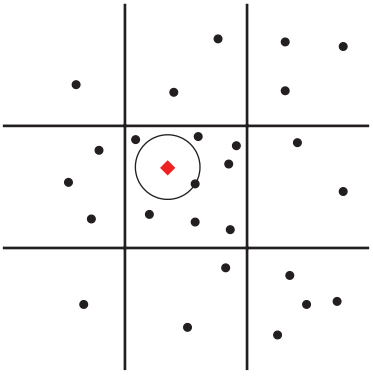


FIGURE 12-9 Finding the nearest match in a grid

Sometimes, though, you aren't so lucky. After checking the query point's grid cell, the closest point in that grid cell to the query point might be so far away that it is possible that a closer point could be in an adjacent grid cell. Figure 12-10 illustrates this scenario. In this case, it is necessary to check the points in adjacent grid cells (shown shaded in light blue) to be sure that the closest point has been found.

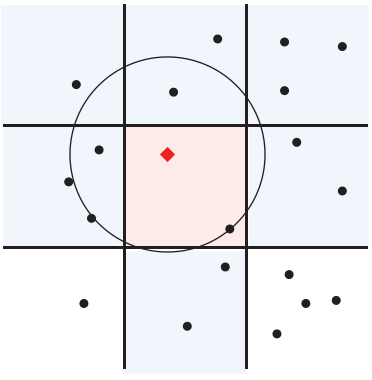


FIGURE 12-10 Other grid cells to be checked to find the nearest point

It is also entirely possible that there are no points at all in the query point's grid cell. In this case, it is necessary to check grid cells at distances farther and farther away from the query point until a point near (but possibly not nearest) the query point is found. This checking is done a *layer of cells* at a time, where each layer is one grid cell farther away from the query point's cell.

The search process can stop after the distance from the query point to the nearest point found so far is smaller than the distances from the query point to the edges of the entire layer whose points have all been compared to the query point.

Does the Query Circle Fall Within a Layer?

How can the algorithm be certain that `findNearest()` has found the point that is closest to the query point? After all the points within a layer have been checked, and if the query circle is enclosed by that layer, then it is certain that there are no points in farther layers that could possibly be closer to the query point. In the case of Figure 12-9, you know that there is no need to consider additional layers of cells. In the cases of Figure 12-10 and Figure 12-11, however, the query circle extends beyond the bounds of the current layer of cells, so it is necessary to consider the points in the next layer out. In Figure 12-11, the cell containing the query point is empty, forcing the search to at least include the first layer. Stopping the search at the first layer, however, would miss potentially closer points in the second layer.

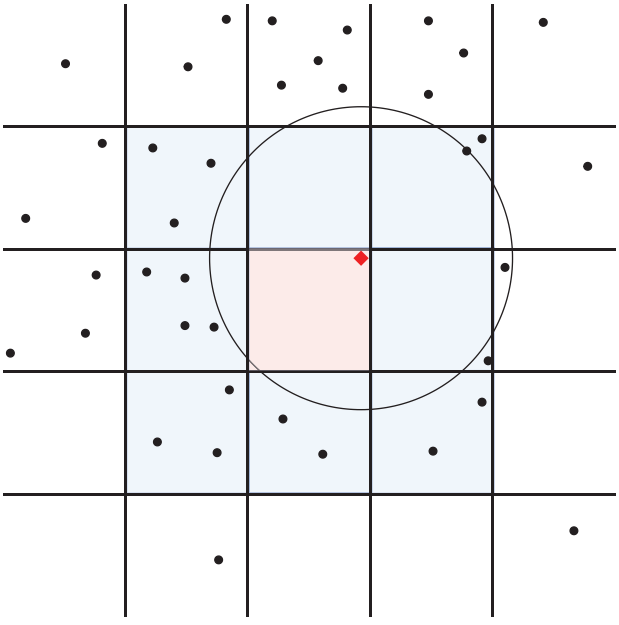


FIGURE 12-11 Checking additional layers to find the nearest point

The row and column number of a cell can be turned back into the coordinates of its lower-left corner by multiplying by the private `__cellSize` attribute. So, given the row and column of the cell containing the query point, the `__cellSize`, and the number of the layer that you're searching (assuming that the center cell containing the query point is at layer zero), it is easy to compute the left, right, top, and bottom bounds of the entire layer in either Cartesian or geographic coordinates.

For example, see Figure 12-12, where the row, col of the center query cell is at 1, 2; the current layer is 2; and the cell size is 50 units in Cartesian coordinates. Because you're at layer 2, the entire square layer encompasses five rows and five columns of cells, each of which is 50 × 50 in Cartesian coordinates. Then the row of the bottom of the layer is 1 – 2 or –1, the row of the top layer is 1 + 2 or 3, and in terms of Cartesian coordinates

bottom edge: y = (–1 * 50) = –50 top edge: y = ((3 + 1) * 50) = 200

Likewise, the left and right coordinates of the current layer can be calculated from the column number of the query cell.

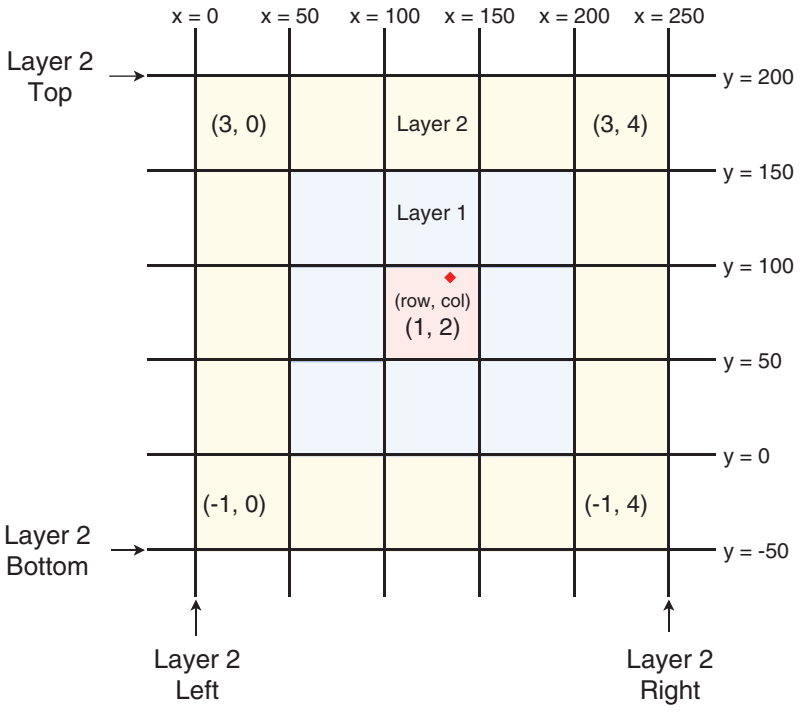


FIGURE 12-12 Determining the boundary coordinates of a layer

The `__getLayerBounds()` method in Listing 12-18 takes a row, column tuple specifying the `cell`, and a layer number as parameters, and does this computation, returning a `Bounds` object containing the Cartesian or geographic coordinates of the left, right, top, and bottom edges of the current layer of cells.

LISTING 12-18 Computing the Bounds of a Layer

```
class Grid(object):
…
    # return Bounds of the layer with cell at its center
    def __getLayerBounds(self, cell, layer):
        left = (cell[1]-layer) * self.__cellSize
        right = left + (self.__cellSize * (layer * 2 + 1))

        bottom = (cell[0]-layer) * self.__cellSize
        top = bottom + (self.__cellSize * (layer * 2 + 1))

        return Bounds(left, right, top, bottom)
```

It's now a simple matter to compute whether a query circle is entirely contained within the bounds of the rectangle centered at the query point and that extends out the specified number of layers, as shown in Listing 12-19. First, you compute the location of the layer's outer boundaries, and then you use the `Bounds.within()` method (refer to Listing 12-7) to determine whether the query circle's bounding box falls within the bounds of the current layer.

LISTING 12-19 Determining Whether a Query Circle Lies Within a Layer

```
class Grid(object):
…
    # Returns true if the query circle falls within the specified
    # layer surrounding the search coordinates.
    # a,b are the coordinates of the query circle center
    # cbounds is the current Bounds of the query circle
    def __withinLayer(self, cBounds, a, b, layer):
        # get the bounds of the layer
        cell = self.__getCell(a, b)     # row,col of cell
        layerBounds = self.__getLayerBounds(cell, layer)

        # check if the circle's bounds are within the layer
        return cBounds.within(layerBounds)
```

Does the Query Circle Intersect a Grid Cell?

The computation of distances between the query point and each point in a grid cell can be computationally expensive, especially for the computation of the haversine distance, which involves multiple trigonometric function calls. But you need to consider only the points contained in those cells that intersect with the query circle, and you can ignore the cells that don't intersect with the query circle.

Figure 12-13 illustrates five query circle sample scenarios in which might occur when determining whether any part of a query circle falls within the center grid cell. By creating a bounding box surrounding the query circles, you can then easily determine whether a query circle might intersect with the center grid cell. Of course, approximating query circles with their bounding boxes has limitations. The approximation may sometimes falsely decide that a query circle intersects a grid cell, when in reality it is the corner of the bounding box that intersects with the grid cell, but not the circle itself. We have chosen to use bounding boxes of the query circle, even at the expense of occasional additional visits to a cell, to simplify the intersection computations for highly distorted query circles that are far from the equator.

If the circle's bounding box and cell don't intersect, as is the case for query circles 3 and 5 in Figure 12-13, then you can move on to the next cell. The `intersects()` method of the `Bounds` class in Listing 12-6 performs this intersection test (with the bounding box approximation to the query circle).

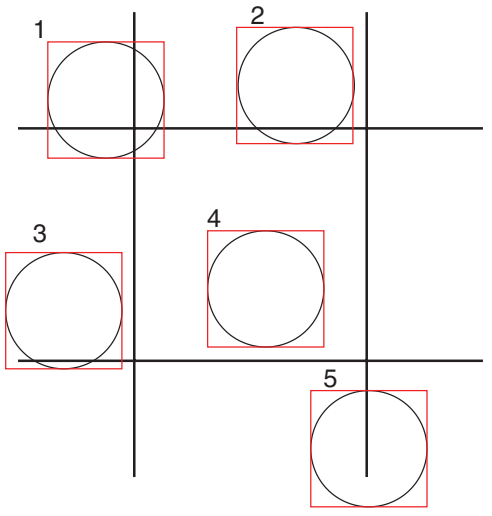


FIGURE 12-13 Determining the intersection between a query circle and a cell

Generating the Sequence of Neighboring Cells to Visit

The search for the point closest to the query point starts at the cell containing the query point and then works outward, considering the points in cells at successive layers of distance from the query point's cell. Figure 12-14 shows the query point in red and then the neighboring cells labeled with the row and column offset from the query cell. The layer number can be easily derived from the row, column offset by taking the maximum of the absolute values of the two offsets.

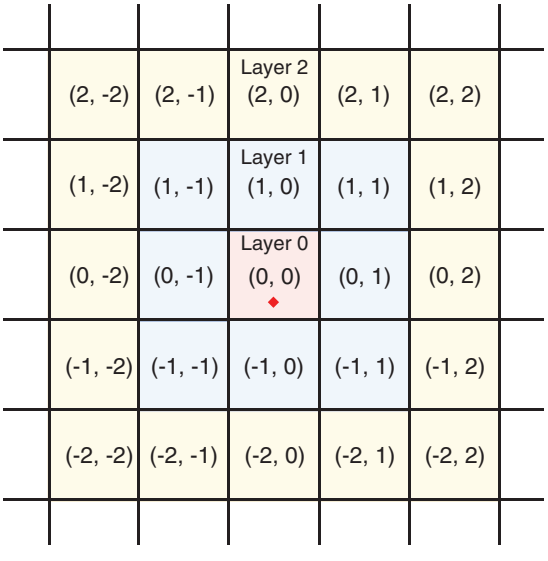


FIGURE 12-14 Offsets from the query center to each layer

The `Grid` class contains a static method called `offsets()`, written as a generator, and shown in Listing 12-20. The method yields tuples, one at a time, where each tuple is the row, column offset of the next grid cell to be visited. For each layer, `offsets()` first yields tuples for the cells along the sides of the layer, followed by the tuples for the corners of the layer. The corners are yielded last for each layer because the corner cells contain points that are potentially the farthest from the query point.

LISTING 12-20 A Generator to Produce Grid Offsets in Layer Order

```
class Grid(object):
    # Generator to yield grid offsets in concentric
    # layers, starting at the center 0,0
    @staticmethod
    def offsets():
        yield 0,0

        layer = 1
        while True:
            for num in range(-layer+1, layer):
                yield    num,  layer # yield offsets for the
                yield    num, -layer # cells along each side
                yield -layer, num
                yield  layer, num

            yield -layer,  layer # yield offsets for the
            yield  layer,  layer # corners of the layer
            yield  layer, -layer
            yield -layer, -layer

            layer += 1  # move on to the next layer
```

This method could hypothetically run forever, yielding an infinite sequence of successively farther layers of grid cell offsets. As you soon see, however, `findNearest()`stops consuming offsets from the generator after the query circle fits within a layer. As a practical matter, if there are sufficient points per grid cell, it is rare that more than one layer of neighboring cells needs to be visited.

Pulling It All Together: Implementing Grid's findNearest()

With these preliminaries out of the way, you can finally implement `findNearest()`. This method is a bit longer than usual, so the code comments in Listing 12-21 are annotated with lettered labels, which we expand on in the coming paragraphs.

> A: If nothing has been inserted yet into the grid, there's no need to proceed further, so just return `None`. This is the only scenario in which `None` is returned. Note that the `Grid` object does not keep track of the number of items it stores. The length of the Python dictionary (`__cells`) is the number of keys with values, so when there are no keys, no lists of points have been inserted.

B: The algorithm keeps track of the closest point seen so far, and the distance to that point. Initially, the closest point answer is `None`, and the distance to the closest point is `math.inf`. Because `inf` is the largest possible number, any subsequent point's distance will necessarily be closer.

C & D: The `__getCell()` method converts the query point's coordinates into the row, column of the query point's cell. That cell is considered to be layer zero.

E: Loop over each of the grid cell offsets returned by the `offsets` generator. The first offset is (0,0) so that causes the search process to start at the cell containing the query point.

F: At some point, the next offset in the sequence will be one layer farther out. To determine when the layer number changes, we calculate the layer of this next offset. As seen in Figure 12-14, the layer number is the larger of the absolute values of the row and column offset values.

G: If the next offset lies in a new layer of cells, this is the opportunity to check whether it is necessary at all to loop over the cells in the new layer. So, if an answer point has been found, and if the query circle fits entirely within the previous layer (`curLayer`), then there is no point in proceeding with the next layer. The method breaks out of the offset loop to return the best answer point so far. Otherwise, it records the next offset's layer as the current layer.

H: Having arrived at a new offset cell to search, `findNearest()` retrieves the list of points stored there. The cell's row and column are obtained by adding the offsets to the query point cell's row, column.

I: Now we're ready to decide whether to do the work to compare all the points in the new cell to the query point. Comparisons are only needed if there are some points in the cell, and if the query circle intersects the cell. Initially, when an answer hasn't yet been found, perhaps because all the grid cells so far have been empty, the distance is still infinity, so the query circle will intersect every cell. But if the query circle doesn't intersect the cell, then none of the points could possibly be closer to the closest point encountered so far.

J: Looping through each of the points in the current cell, compute the distance from the point to the query point. If the distance is exactly zero, this means that search has found an exact match to the query point and can return the point and its data. Otherwise, if the distance to the query point is less than the smallest distance found so far, remember the new candidate point and its distance. Because the distance to the nearest point has now decreased, the method invokes `CircleBounds.adjust()` to recompute the bounding box for the smaller query circle.

K: There's only one way to make it to this line of code. Namely, that the `break` statement was executed during step `G` because an answer was found and because the query circle was entirely contained within the previous layer of searched cells. The `findNearest()` method returns the coordinates of the point, the data stored with that point, and the distance from the query point to the answer (that is, nearest) point.

This process may seem complex. Yet, as we show at the end of the chapter, in certain situations this `Grid.findNearest()` can perform significantly better than the theoretically superior `QuadTree` code.

LISTING 12-21 The `findNearest()` Method for `Grid`

```
class Grid(object):
…
    def findNearest(self, a, b): # find the closest point to (a,b)
        if len(self.__cells) == 0:  # A: Nothing in grid yet
            return None

        answer = None  # B: remember the closest point so far

        # The current search circle and its bounds so far
        distance = inf
        cBounds = CircleBounds(a, b, distance, self.__distance)

        cell = self.__getCell(a, b) # C: cell containing a,b
        curLayer = 0                # D: the layer we're up to

        # E: for each offset
        for off in Grid.offsets():
            #F: what layer is this new offset?
            layer = max(abs(off[0]), abs(off[1]))

            # G: if we're about to consider a new layer,
            # but the search circle falls entirely within
            # the prior layer, then there's no need to continue.
            if layer != curLayer and answer and \
               self.__withinLayer(cBounds, a, b, curLayer):
                break
            curLayer = layer # remember what layer we're up to

            # H: get the points stored in the cell at that offset
            offsetCell = cell[0]+off[0], cell[1]+off[1]
            points = self.__cells.get(offsetCell, None)

            # I: if there are points in the cell, and
            # the search circle intersects the cell...
            if points and \
               cBounds.intersects(self.__getCellBounds(offsetCell)):

                for p in points: # J: for each point in the grid cell
                    # compute distance to that point from query point
                    newDist = self.__distance(a, b, p[0], p[1])
```

```
            if newDist == 0: # exact match?
                return p[0], p[1], p[2], 0

            if newDist < distance:  # new point closer?
                distance = newDist  # remember the distance
                answer = p          # and the point, and
                cBounds.adjust(distance) # adjust bounds

        # K: returns a, b, data, distance to query point
        return answer[0], answer[1], answer[2], distance
```

Quadtrees

In the previous sections, you saw that a grid is an excellent data structure for searching points when the data is somewhat uniformly spread out because each grid cell contains a relatively equal number of points for searching. But what happens if the data is not so uniformly spread out? Figure 12-15 shows about 30,000 points. Each of the clusters has about one-third of the points. This kind of clustered data is typical in the real world. For example, locations of restaurants, coffee shops, and other businesses tend to be clustered near each other.

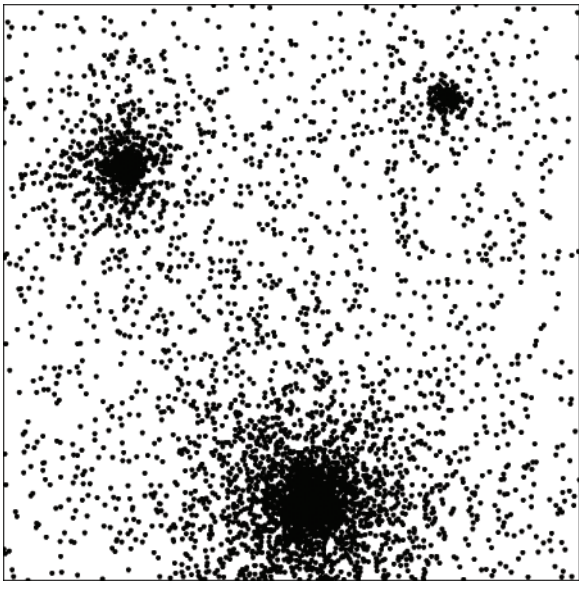


FIGURE 12-15 Thirty thousand highly clustered points

Overlaying a grid on top of this data, as shown in Figure 12-16, makes it clear that for clustered data a grid doesn't offer much benefit. Searching for a point inside the red-outlined grid cell near the bottom would involve processing nearly all the points in that cluster.

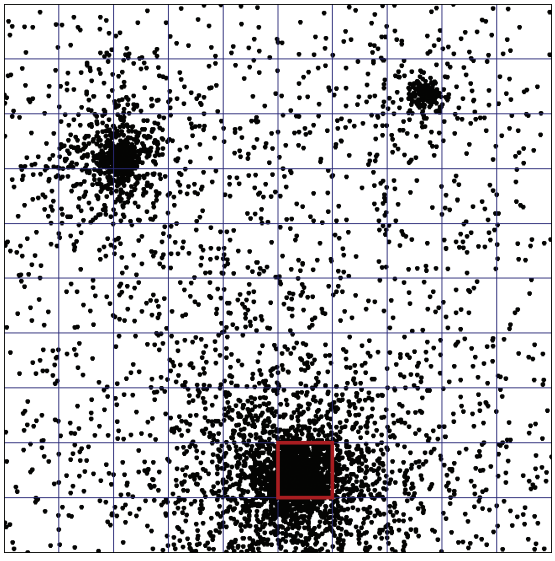


FIGURE 12-16 Uniform grid dividing the 30,000 points

Many of the grid cells are empty or contain just a few points, while most of the points are concentrated in a small number of the grid cells. As one solution, you could decrease the size of every one of the grid cells. So instead of a grid of 10×10 cells overall, you'd use a grid of 100×100 cells for a total of 10,000 grid cells to store the 30,000 points. Figure 12-17 zooms in to look at just one of the 10×10 cells, the red-outlined one, which contains several thousand points.

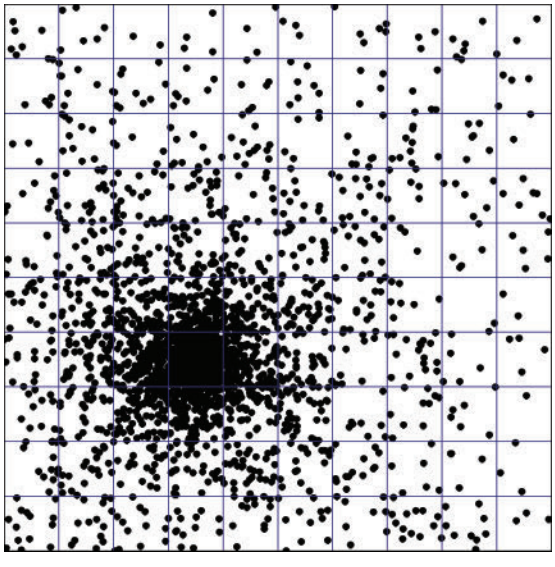


FIGURE 12-17 A finer grid on just one of the point clusters

At this scale, because you are now using a 100 × 100 grid, what used to be a single grid cell is now itself composed of 10 × 10 of the smaller cells. And again, most of the points are heavily concentrated within a small fraction of the cells, while most of the cells contain a relatively small number of points.

This is the dilemma with using fixed grids where the cell size is uniform across all the data. Real-life data tends to be very unevenly distributed. At whatever scale of magnification that you use to examine the data, clusters tend to be apparent. The implication is that no matter how fine the grid is made, some areas of space would benefit from a finer grid.

Quadtrees, also written as *quad trees*, were invented in the early 1970's by Raphael Finkel and Jon Bentley, and provide an elegant solution to this problem. A **quadtree** is a tree data structure that recursively divides up space into four quadrants. Parts of the quadtree that contain more points are further divided into finer-sized quadrants. Parts of the quadtree that contain fewer points divide space into larger quadrants. Adaptively subdividing space based on the density of spatial data makes it possible to insert or search for points in $O(\log_4 N)$ time, or as you saw in Chapter 2, "Arrays," more simply just O(log N) time.

This tree-like structure probably looks familiar. The different kinds of trees you studied in Chapters 8, 9, and 10 were all organized by dividing the items by their keys. Quadtrees do something similar but with two numeric values. A binary search tree with a single numeric key inserts items with lower-valued keys in the left subtree and higher-valued keys in the right subtree. With spatial data, you can do something similar, but this time with quadrants.

In the following pages, we look in more detail at a quadtree implementation using the same methods that were implemented for the `PointList` and `Grid` classes: a constructor, `insert()`, `findExact()`, `findNearest`(), `delete()`, and `traverse()`.

Creating an Instance of the QuadTree Class

Client code creates an instance of the `QuadTree` class by invoking the constructor to create an object that contains Cartesian points:

```
q = QuadTree('cartesian')
```

or that contains geographic points:

```
q = QuadTree('geographic')
```

If the constructor is passed any string other than `'cartesian'` or `'geographic'`, an exception is raised.

Listing 12-22 shows the code for the `QuadTree` constructor. It has just two private attributes: a reference to the distance function appropriate to either Cartesian or geographic coordinates, and an initially empty reference to the root node of the tree.

LISTING 12-22 The `QuadTree` Constructor

```
class QuadTree(object):

    def __init__(self, kind): # must specify type of coordinate
        if kind == 'cartesian':
            self.__distance = euclideanDistance
        elif kind == 'geographic':
            self.__distance = haversineDistance
        else:
            raise Exception("Invalid kind of coordinate")

        self.__root = None
```

Inserting Points: A Conceptual Overview

Let's look at the first few inserts of points into an initially empty quadtree, with each new point finding its place in the quadtree through a recursive descent. Each node in a quadtree potentially has four children, corresponding to the northeast, southeast, southwest, and northwest quadrants of space that surround that point. The first point to be inserted into the quadtree becomes the root of the tree (labeled point A in Figure 12-18), and the four children of that node all refer to `None`.

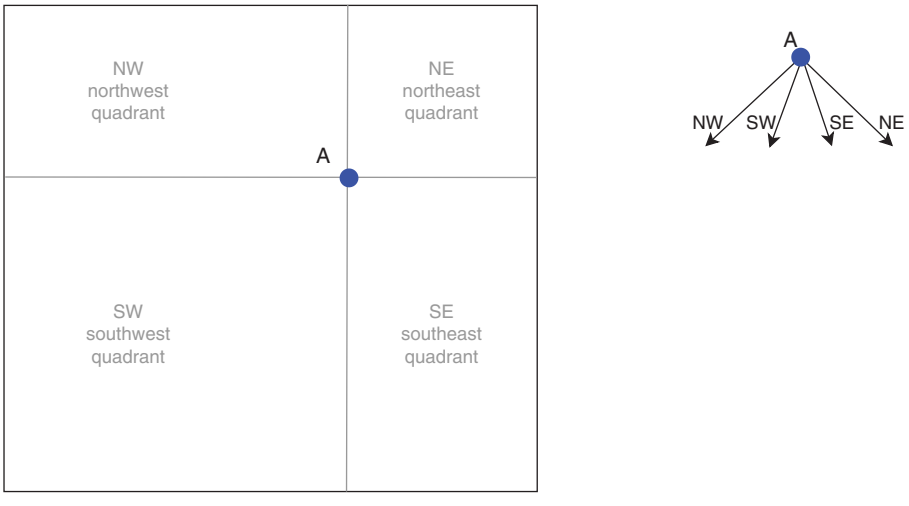


FIGURE 12-18 Inserting the root point in a `QuadTree`

Next, as shown in Figure 12-19, another point, labeled B, is inserted. Starting at the root, the coordinates of the new point are compared against the coordinates of the root node A. Point B lies to the southwest of point A, so you recursively descend to the corresponding child of point A. Because the southwest child currently refers to `None`, a new node is created to contain B, and upon return from the recursion, the southwest child of A is made to refer to B.

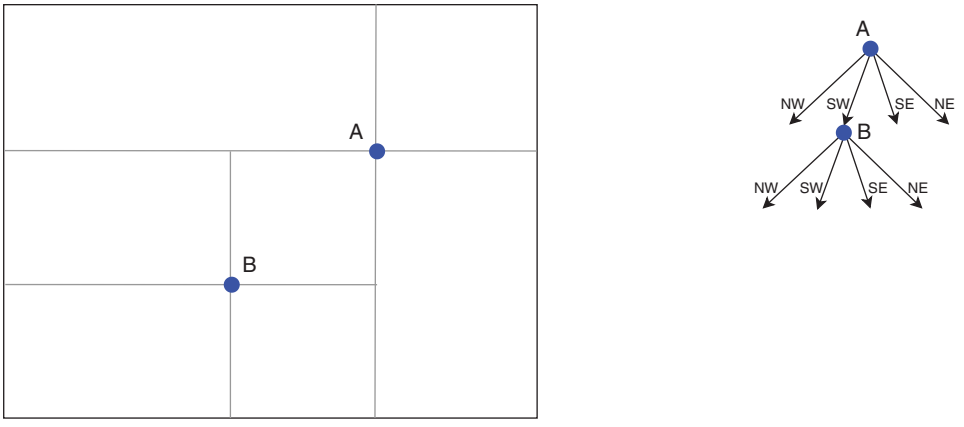


FIGURE 12-19 Inserting point B

The next point, labeled C, is inserted, as shown in Figure 12-20. Again, starting from the root node, C is compared to the coordinates of point A. Point C lies to the southeast of A, so the recursion proceeds to the southeast child, and because it is found to be `None`, a new node is created containing the coordinates of point C, which is then returned from the recursion and assigned to the southeast child of A.

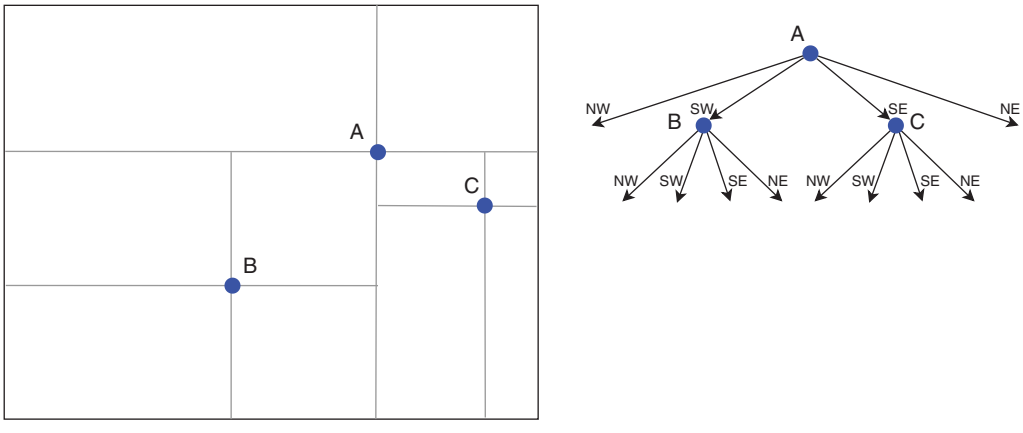


FIGURE 12-20 Inserting point C

Finally, as shown in Figure 12-21, a point, labeled D, is recursively inserted into the tree, starting at the root point A. Because D lies in the southwest quadrant of A, the recursion then proceeds to point B, which is the southwest child of point A. The coordinates of point D are northwest of the coordinates of point B, so the recursion then proceeds to the northwest child of B. Having reached `None`, a new node is created to hold point D, which is then returned from the recursion. Upon returning to point B, the northwest child of B is set to refer to the just-returned reference to the newly created point D.

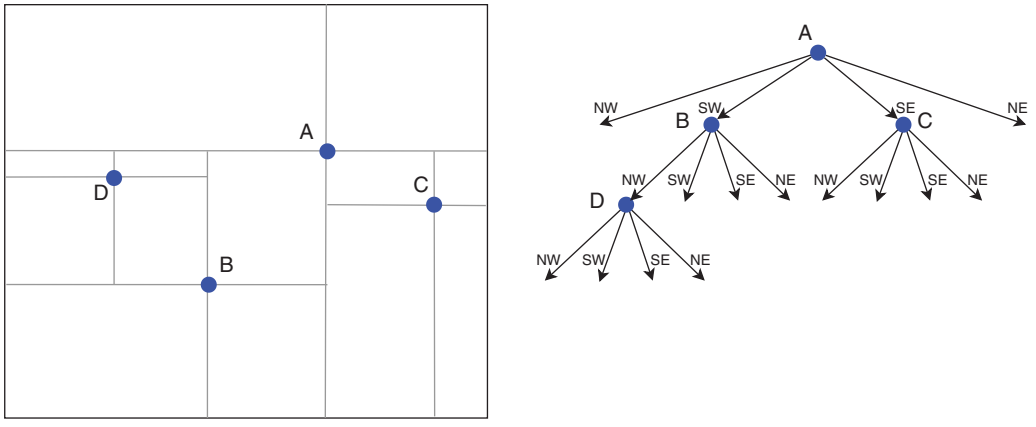


FIGURE 12-21 Inserting point D

Avoiding Ambiguity

When recursively descending a quadtree either for insertion or searching, it is important to make a consistent decision at each node regarding which child to descend into. If a point is properly within a quadrant, then there is no ambiguity. But which quadrant should be chosen if an inserted or searched point lies directly above or below, or directly to the right or left of a node? For example, in Figure 12-22, to which quadrant of A does point B belong?

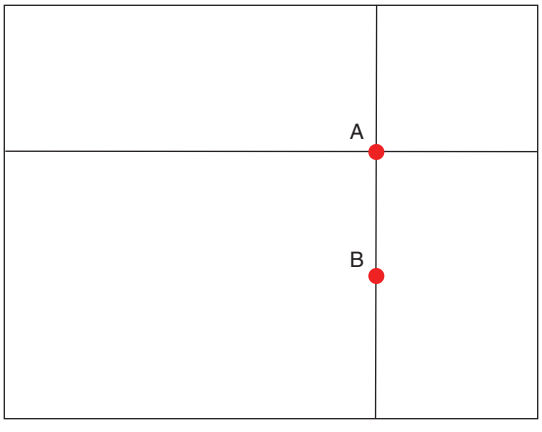


FIGURE 12-22 Quadrant ambiguity

Whatever solution we choose, we must make certain to apply it consistently. Our solution to this problem is to have the points on each of the four axis boundary lines belong to the quadrant that is clockwise of the axis line. So, in Figure 12-22, B would belong to the southwest quadrant of A.

More generally, as shown in Figure 12-23, the points that are exactly above the node's point all belong to the node's northeast quadrant. The points that are exactly to the right of the quadtree node belong to the node's southeast quadrant, and so on.

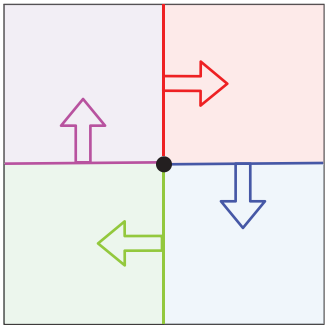


FIGURE 12-23 Assignment of axis boundary lines to quadrants

The QuadTree Visualization Tool

To see how quadtrees operate, you can run the QuadTree Visualization tool. The tool can be used to manage Cartesian point data in a rectangular region, as shown in Figure 12-24. The points are shown as dots with a label above them that represents the "data" associated with a point.

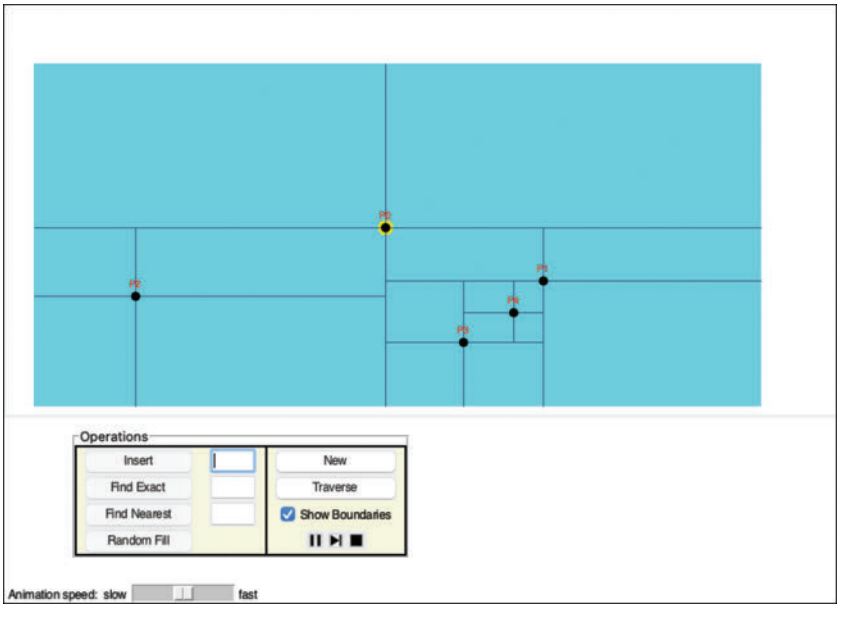


FIGURE 12-24 The QuadTree Visualization tool

You can insert new points in two ways: by double-clicking with a pointer (mouse) button in the light blue region, or by entering an X coordinate, Y coordinate, and data label in the three text boxes and selecting the Insert button.

The first point inserted becomes the root of the quadtree and is highlighted in yellow. When you insert points by double-clicking, the tool automatically chooses the data label in the form P0, P1, P2, and so on, based on the current number of points in the quadtree. When you type the coordinates, they are relative to the origin, point (0, 0), at the lower left of the region. You can provide any short string for the data (as long as the points have unique names).

The quadrants around each point are visible when the boundaries between them are displayed. If you uncheck the Show Boundaries checkbox, you will see only the points, making their tree relationship invisible. Finding the parent of a node is somewhat tricky. You must look at the boundaries, follow them to their endpoint, and then follow the crossing boundary to see whether it intersects a labeled node. For example, in Figure 12-24, P3's parent is P1, and two of P3's boundary lines end on the boundary lines of P1. The parent of P4 is harder to see because both P1 and P3 look equally likely based on the closest boundary lines. You must look at the extent of the boundaries of the two potential parents to resolve the ambiguity. P0, the root, has boundary lines that extend all the way to the blue region boundaries, and those have no boundary lines leading to another node.

Try inserting a few points by typing the coordinates for the points. By giving exact coordinates, you can insert new points on a quadrant boundary. Based on the rule in Figure 12-23, you should be able to predict the quadrant of the new point. The (single) boundary line drawn for it spans that quadrant. The other boundary line overlaps the boundary line where the point lies, so it is not visible. You can also click an existing point to copy its coordinates and label to the text input boxes. From there, you can edit the values to insert a new point that lands exactly on one of its quadrant boundaries. Clicking the blue region where there are no points simply enters the two coordinates into the text boxes.

Implementing Quadtrees: The Node Class

Just as with other types of trees, you can define a class to correspond to the nodes within a quadtree, as shown in Listing 12-23. The `Node` class and its attributes don't have to be made private because the `QuadTree` class will never directly return an object of the `Node` type, and therefore, encapsulation will not be violated.

LISTING 12-23 The `Node` Class

```
class Node(object):
    # a,b corresponds to either x,y or long,lat
    def __init__(self, a, b, data):
        self.a = a  # coordinates of the point
        self.b = b
        self.data = data # data associated with point

        # Four children of the Node
        self.NE = self.SE = self.SW = self.NW = None
```

Having seen the constructors you may wonder, where in the `Node` or `QuadTree` classes are the boundary lines for each quadrant stored? The answer is that the boundary lines are not explicitly stored anywhere! Rather, you can think about the *a*, *b* coordinates of each `Node` as establishing the boundaries that subdivide all of the space surrounding the point into four quadrants. For example, the southeast quadrant of a `Node` has its top boundary specified by the *b* coordinate, and its left boundary specified by its *a* coordinate. But what about the bottom and right boundaries? Those boundaries are not explicitly stored in the `Node` objects; rather, they are implicit. That is partially the reason why it is difficult to see the parent-child relationships from the boundaries shown in the visualization tool—the boundaries of a child extend to its parent's boundaries.

As you will see shortly, the implementation of `findNearest()` search recursively descends the quadtree, progressively refining the boundaries of the space of the quadrant being visited. For example, the boundaries of the root `Node` at (*a*, *b*) are, respectively:

left: `-math.inf`, right: `math.inf`, top: `math.inf`, bottom: `-math.inf`

In other words, the root node's space encompasses all points in the plane. Suppose you descend to the southeast corner of the root node, P0, which would be node P1 in Figure 12-24. Then you can refine the boundaries for the southeast quadrant of P0 at (*a*, *b*) to be

left: `a`, right: `math.inf`, top: `b`, bottom: `-math.inf`

In other words, the southeast quadrant of the root is thought of as being unbound to the right and downward. As you descend farther into the tree, the left, right, top, and bottom boundaries get updated to reflect the subset of the plane that is associated with a child `Node`. In many parts of the implementation, however, these boundaries don't need to be explicitly stored. For example, they are not needed for insertion. We discuss the explicit boundaries in more detail when we discuss `findNearest()`.

The insert Method

`QuadTree` clients insert point data using an `insert()` method with the same signature as those in `PointList` and `Grid`. Listing 12-24 shows the Python code for `insert()` and its companion method, `__insert()`.

The `insert()` method is a wrapper method whose sole function is to invoke the recursive `__insert()` method, passing to it the root of the quadtree. Whatever `__insert()` returns is then assigned back to the `__root` attribute. This unusual pattern is worthy of some discussion, as you may recall from when it was used in Chapter 10, "AVL and Red-Black Trees," for `AVLTree` objects.

Why should `insert()` need to reassign a value for `__root` upon every invocation? Consider the case when the quadtree does not yet contain any points, and therefore `__root` refers to `None`. When `insert()` invokes `__insert()`, a new `Node` object is created and then returned immediately to `insert()`, which then assigns it to `__root`. That makes sense! But what about subsequent invocations of `insert()`? Would its recursive call to `__insert()` ever return a reference to a different `Node`? No, it would not. But that is OK, because the assignment to `self.__root` just ends up assigning the same `Node` to it that is already there! As you look more carefully at `__insert()`, you'll see that you can ask the same question regarding the returns from all the layers of recursion.

LISTING 12-24 The `insert()` Method of `QuadTree`

```
class QuadTree(object):
…
    def insert(self, a, b, data): # Wrapper. Always succeeds.
        self.__root = self.__insert(self.__root, a, b, data)

    # Recursive private method that does the inserting
    def __insert(self, n, a, b, data):

        # return a new Node if we've reached None
        if not n: return Node(a, b, data)

        # if the point to be inserted is identical to current node,
        # overwrite the data, and don't recurse any further
        if n.a == a and n.b == b:
            n.data = data
            return n

        # recurse down into the appropriate quadrant
        if   a >= n.a and b >  n.b: n.NE=self.__insert(n.NE,a,b,data)
        elif a >  n.a and b <= n.b: n.SE=self.__insert(n.SE,a,b,data)
        elif a <= n.a and b <  n.b: n.SW=self.__insert(n.SW,a,b,data)
        else:                       n.NW=self.__insert(n.NW,a,b,data)
           # a <  n.a and b >= n.b

        return n
```

Moving on to `__insert()`, the method takes as parameters a reference to the current `Node` `n` in the `QuadTree`, along with the *a*, *b* coordinates and the associated data for the point to be inserted. In the first step, if `__insert()` reaches an empty child node (`None`), then it creates a new `Node` and returns it. The caller might have been the `insert()` wrapper, in which case the new `Node` will be assigned to `self.__root`. The caller could also have been one of the recursive invocations of `__insert()` later in the method. When those invocations return, the returned newly created `Node` gets assigned into the quadrant attribute that until now referred to `None`.

If `n` refers to anything other than `None`, then the next check is for the possibility that the newly inserted point *a*, *b* and the current node `n` both refer to the exact same coordinates. We made a design decision that `QuadTree` objects will not support multiple points with the duplicate coordinates. So, if there is an exact match, then the newly inserted data will overwrite the existing data stored for those coordinates, and then `__insert()` can return a reference to the current node `n`. We leave it as an exercise to explore how to insert points with duplicate coordinates.

Finally, the real work of the recursion happens. At this point of the `__insert()` method, *a*, *b* can't possibly be an exact match to `n`'s coordinates. The *a*, *b* coordinates of the inserted

point are compared to the coordinates of the current node `n` to decide which quadrant should receive the new point. Let's take a careful look at the first of these four similar recursion decisions:

```
if a >= n.a and b > n.b: n.NE = self.__insert(n.NE, a, b, data)
```

Notice that points will end up in the northeast quadrant if the `a` coordinate is greater than or equal to the current `Node`'s *a* coordinate and if the *b* coordinate is greater than the current `Node`'s *b* coordinate. But that is exactly the decision that we illustrated in Figure 12-23: ambiguous points are placed in the quadrant clockwise away from the boundary where they lie. The decisions for recursion into the other three quadrants are likewise consistent with the figure.

Also notice the peculiar form of the recursive invocation. It recursively passes `n.NE` into `__insert()`, and then whatever the invocation of `__insert` returns gets assigned right back into `n.NE`! This is just a variation of the first invocation of `__insert()` by the `insert()` wrapper method. Here, when `__insert()` is recursively invoked, `n.NE` might be `None`, or it might already refer to a child `Node`. If `n.NE` is `None`, then immediately upon the recursive invocation, `__insert()` will create and return a new `Node` containing the inserted *a*, *b* data point. The caller appropriately assigns the new node to `n.NE`. If `n.NE` is not `None` and already refers to a child `Node`, then `__insert()` will necessarily end up returning that very `Node`, which then gets harmlessly assigned back to `n.NE`.

Appropriately, the last line of `__insert()` causes a reference to the current `Node` `n` to be returned to the calling method.

In summary, there are three possible returns from the recursion:

1. A reference to a brand-new `Node` is returned if the recursion reached an empty node represented as `None`.
2. If a duplicate point was inserted, after overwriting the current node's data with the newly inserted data, a reference to the current `Node` is returned.
3. A reference to the current `Node` `n` is returned if the insertion recursively occurred somewhere below the current `Node`.

In all three cases, what is returned to the calling method is a reference to the `Node` that belongs at that point in the `QuadTree`. Most of the time, the returned value references a `Node` that has already been created and that is already referred to by one of the child attributes of the parent `Node`.

Another way to think about this is to imagine the process involved in inserting a point with new coordinates 10 levels down into the `QuadTree`. You recurse down 10 levels into the appropriate quadrants and reach a child attribute containing `None`. At that point, you create a new `Node`, and it is returned to be assigned to the parent's appropriate child attribute. As the recursion then "unwinds" all the way back to the root, each intermediate `Node` is just returned to the calling method, which then harmlessly assigns the `Node` back to the child attribute that already contains that reference.

Try stepping through the insertion of a point using the visualization tool. It shows how each recursive call to `__insert()` checks the relationship of the new point's coordinates with those of a node in the tree and then either descends to a child node, creates a new node, or updates the data of an existing one. As the recursion unwinds, the nodes visited along the path to the new or updated node are returned to their parent.

Efficiency of Insertion

If the quadtree is relatively well balanced, then inserts into the quadtree should take O(log N) time, so inserting N points would take $O(N \times \log N)$ time. Just as with binary search trees, degenerate sequences of inserts can cause the tree to be very unbalanced. For example, if points were inserted in order along a southwest diagonal line, the result would be the worst possible case, with each insertion taking O(N) time, resulting in $O(N^2)$ time to insert N points. You see shortly that such an unbalanced quadtree degrades the performance of searching, too.

Dynamic rebalancing of a `QuadTree` in a manner like an AVL tree is possible but very complex. If an application has all, or even most, of the points in advance that need to be stored in a `QuadTree`, then a bulk insert method can arrange for the tree to be better balanced. We leave this method for you to complete as an exercise at the end of the chapter.

Finding an Exact Match

The client uses the `findExact()` method to search for the point contained in the `QuadTree` object that exactly matches the specified *x*, *y* or longitude, latitude coordinates. If an exact match is found, the method returns a reference to the data stored for the matching point. The client uses `findExact()` in the same fashion as with the `PointList` and `Grid` classes.

Finding an exact match is a simple process, as shown in Listing 12-25. If the `QuadTree` contains an exact match, then a recursive descent that mirrors the descent used by `insert()` is guaranteed to find the matching point.

Like the `insert()` method, you can use a wrapper method called `findExact()` to start the recursion by passing the root node into the recursive `__find()` method and simply returning its result. The first base case occurs if the recursion has reached a `None` (empty leaf) value. If this happens, it is certain that there's no exact match for *a*, *b* and `None` should be returned to indicate that no point was found, and therefore no data returned.

Another base case occurs if the search has reached a node that exactly matches *a*, *b*. In this case, the associated data is returned and makes its way back to the wrapper method as the recursion unwinds.

Otherwise, if an exact match to *a*, *b* is anywhere in the quadtree, it must be in the quadrant that encloses *a*, *b*. So, just as was done for insertion, you recursively descend into the appropriate quadrant, and you return whatever is returned by the recursive call. If the recursion reaches `None`, then it is certain that there's no node that matches the specified coordinate.

LISTING 12-25 The `findExact()` Method of `QuadTree`

```
class QuadTree(object):
…
    # Wrapper method - returns the data object associated with a,b
    def findExact(self, a, b): return self.__find(self.__root, a, b)

    def __find(self, n, a, b):
        if not n: return None  # Couldn't find the exact point

        # Did we find the exact point? Return the data
        if n.a == a and n.b == b: return n.data

        # Recurse down into the appropriate quadrant.
        # If a,b is anywhere, it MUST be in that quadrant.
        if   a >= n.a and b >  n.b: return self.__find(n.NE, a, b)
        elif a >  n.a and b <= n.b: return self.__find(n.SE, a, b)
        elif a <= n.a and b <  n.b: return self.__find(n.SW, a, b)
        else:                       return self.__find(n.NW, a, b)
           # a <  n.a and b >= n.b
```

Try a few exact searches with the visualization tool to watch this process. Create a few points in the quadtree and then click a region with no points. Doing so fills in the coordinates of the clicked point in the arguments area and enables the Find Exact operation. Select the Find Exact button. It draws a cross at the clicked point and makes the recursive calls to `__find()` as it checks the relationship of *a*, *b* to each node. You can also click an existing point to verify that `findExact()` follows the nodes in the tree to that point.

Efficiency of Exact Search

If the quadtree is relatively well balanced, then finding an exact match for a point in a quadtree should take O(log N) time, though in the worst case with a completely unbalanced tree, `findExact()` could be as bad as O(N). This degenerate performance makes it no better than the operation in a simple `PointList`, although it is very unlikely to occur with data that is inserted in a random order with respect to spatial position.

Traversing the Points

Just as we did with `PointList` and `Grid`, `QuadTree`'s `traverse()` implementation should be done as a generator. The natural way to traverse a `QuadTree` is to do a pre-order recursion, starting at the root, and then recursively visiting and yielding each of the nodes in the tree, followed by each node's children. Mixing recursion with generators poses a challenge because it is difficult to write efficient recursive generators in Python.

As we discussed in Chapter 6, "Recursion," every recursive function can be transformed into a nonrecursive version by using a stack. We showed this transformation in Chapter 8,

"Binary Trees." Listing 12-26 shows a similar nonrecursive implementation of `traverse()`, where we use a Python `list` to implement the two required stack operations of push (using `list`'s `append()` and `pop()` methods). Alternately, it would be straightforward to use the `Stack` class from Chapter 4, "Stacks and Queues," if you know the number of points in the quadtree, or the `LinkStack` class from Chapter 5, "Linked Lists," if you don't.

LISTING 12-26 The `traverse()` Method of `QuadTree`

```
class QuadTree(object):
…
   def traverse(self):
        s = [ ]   # initialize stack with root
        if self.__root: s.append(self.__root)

        while len(s) > 0: # stack's not empty?

            # process Node at top of stack
            n = s.pop()
            yield n.a, n.b, n.data

            # push each child onto stack
            if n.NE: s.append(n.NE)
            if n.SE: s.append(n.SE)
            if n.SW: s.append(n.SW)
            if n.NW: s.append(n.NW)
```

The method starts by initializing the stack to contain just the root `Node`. Then, as long as the stack isn't empty, you pop the top `Node` from the stack and yield its coordinates and data. Finally, any children of the `Node` are pushed onto the stack.

This implementation seems more complex than the corresponding `traverse()` methods in the `PointList` and `Grid` classes. What is its Big O complexity?

With a bit of analysis, we can determine that the quadtree's `traverse` is also O(N). We determine this by focusing on the pushes and pops of points to and from the stack. Examining the code, we can see that each `Node` is pushed just once onto the stack, and that each `Node` is ultimately popped (and yielded) just once from the stack, corresponding to each pass through the `while` loop. Thus, for a `QuadTree` containing N nodes, the traversal can be accomplished with N passes through the `while` loop, and a total of N pushes and N pops. These roughly 3N operations correspond to O(N) complexity for the `traverse()` method.

Deleting a Point

To locate a point to delete, you should use the same approach as used in `findExact()`. If the matching point is identified, it needs to be deleted. If not, an exception should be raised.

Removing a leaf node from a quadtree should be relatively simple. What should the method do, however, if the matching node is an interior node of the quadtree? Because

a `Node` can't be easily removed from the quadtree without rebuilding the structure of its children, one reasonable approach would be to leave the `Node` in place, and instead set its data attribute to `None`. Thus, any `Node` with a data attribute containing `None` would not subsequently be treated as a valid data point stored in the quadtree. If the quadtree must support `None` as a legitimate data value for a point, then a special sentinel value must be inserted that can be distinguished from `None`. The `__Deleted` sentinel performs this role for open addressing hash tables (as described in Chapter 11, "Hash Tables").

Marking the data attribute as deleted means that small changes are needed to the `findExact()` method (in Listing 12-25) and the `findNearest()` method (described in the following sections) so that nodes without data are never considered as valid matching points. These data-less nodes take some extra memory but still provide useful information about where space is subdivided into quadrants.

Because `delete()` must first find the node like `findExact()`, the Big O must be at least O(log N). The rest of the work (removing a leaf node or inserting a no-data marker in the data attribute of an interior node) takes a few steps but does not depend on the number of items in the quadtree, so it adds nothing to the overall complexity. If many interior nodes are deleted, the tree does not shrink, so the N in O(log N) is the total number of inserted items, not the current number of items in the tree, for deletion, insertion, and search.

We leave it as a Programming Project for you to implement a method to delete a point specified by a pair of coordinates from the quadtree. The visualization tool does not have a delete operation, but you can erase all points with the New button and then insert new points.

Finding the Nearest Match

As usual, finding the nearest match via the `findNearest()` method is more complicated than `findExact()`. Just as with the `Grid` class, merely descending into the appropriate quadtree quadrant does not guarantee that you have found the nearest match to the search coordinates.

For example, in Figure 12-25, if you descend into the `QuadTree` to find the deepest node whose boundaries surround the red diamond query point (outlined in red), you can use that deepest point as a good *candidate* for the nearest match. However, if you draw a circle with the query point at its center and its radius matching the distance to the candidate, it becomes clear that any quadrant that is overlapped by the circle *might* conceivably contain a closer node and must therefore be considered. Those quadrants that you must visit are shaded in light blue. The example shows that there is a point in the query circle that was not visited on the path to the candidate (because the search from the root node—the one whose quadrant boundaries span the entire figure—followed the path to the NW, then NE, and then SW quadrants based on the query point). How do you ensure that those other quadrants shaded in blue are visited during the search?

Similar to how the `Grid` object had to intersect a query circle with the various cells, there's a way to compare quadtree boundaries with the query circle to minimize the quadrants to explore. Let's look at the implementation details to complete your understanding of how to find the nearest `Node` to a query point.

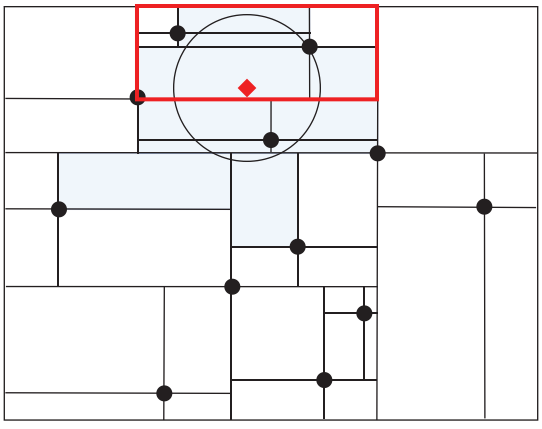


FIGURE 12-25 Additional quadrants to be visited based on query circle

Finding a Candidate Node

Let's start by following the logic used in `findExact()` to get a point somewhere in the vicinity of the query point. Listing 12-27 shows the private `__nearPoint` method. As you soon see, `findNearest()` invokes this method by passing in `__root` and the *a, b* search coordinates.

LISTING 12-27 The `__nearPoint()` Method

```
class QuadTree(object):
…
    # find a nearby (but not necessarily the nearest) point
    # to a,b by recursing as deep as possible into the tree.
    def __nearPoint(self, n, a, b):
        # Base cases:
        if not n: return None # reached None so return it

        if a == n.a and b == n.b: return n # found exact point

        # recurse down into the appropriate quadrant
        if   a >= n.a and b >  n.b: ans = self.__nearPoint(n.NE, a, b)
        elif a >  n.a and b <= n.b: ans = self.__nearPoint(n.SE, a, b)
        elif a <= n.a and b <  n.b: ans = self.__nearPoint(n.SW, a, b)
        else:                       ans = self.__nearPoint(n.NW, a, b)
           # a <  n.a and b >= n.b

        # if we found a lower Node near this point return it
        # otherwise return the current node
        return ans if ans else n
```

The `__nearPoint()` method returns the deepest `Node` in the tree along the path from the root toward the query point. It does so by recursively descending into the `QuadTree` in a fashion almost identical to `__find()`. Instead of returning the corresponding data if an exact match is found to *a*, *b*, `__nearPoint()` returns a reference to the `Node` that either matches *a*, *b* exactly, or is as deep as possible in the `QuadTree`.

Not surprisingly, the base cases are similar to `__find()`. If `__nearPoint()` reaches `None`, then `None` is returned. If it reaches a node whose coordinates exactly match *a*, *b*, then that `Node` (`n`) is returned.

Otherwise, it recursively descends into the quadrant containing the query point and stores in `ans` whatever is returned by the recursive invocation. There are only two possibilities after the recursive call. One possibility is that `ans` is `None` because `__nearPoint()` just descended into an empty child. If that happens, then it means that the current node `n` has no descendants in the query quadrant, and therefore it returns the current node, `n`. Otherwise, if `ans` is anything but `None`, it means that the recursion is returning a reference to the lowest possible `Node`, which is then returned to its caller.

Finding the Closest Node

After `__nearPoint()` has discovered a candidate for the `Node` nearest to *a*, *b*, the `__nearest()` method shown in Listing 12-28 can recursively descend the quadtree again, starting at the root, trying to find successively closer candidates. This method computes the distance from the query point to the coordinates of each `Node` visited. If the distance is smaller than the distance to the so-far best candidate `Node`, then the current node, `n`, becomes the new, closer candidate, `cand`.

LISTING 12-28 Recursively Refine Nearest Candidate

```
class QuadTree(object):
…
    # Returns the nearest Node and distance to query point a,b.
    def __nearest(self, n, # current Node being visit
                  a, b,    # query point
                  dist,    # distance to candidate
                  cand,    # best candidate so far
                  bounds): # Bounds of current quadrant

        # Reached a None node, or already found
        # a perfect match? Return our answer.
        if not n or dist == 0: return cand, dist

        # Is the current quad tree point closer than
        # the candidate? If so, update the candidate
        fn = self.__distance
        newDist  = fn(a, b, n.a, n.b)
        if newDist < dist:
            cand = n
            dist = newDist
```

```
        # bounding box surrounding the search circle
        cBounds = CircleBounds(a, b, dist, fn)

        # For each child node - update bounds for that quadrant
        # if the search circle's bounds intersects the child
        # quadrant's new bounds, descend to that child node.
        newB = bounds.adjusted(left = n.a, top = n.b)
        if cBounds.intersects(newB):
            cand, dist = self.__nearest(n.SE, a, b, dist, cand, newB)
            cBounds.adjust(dist) # adjust the circle's bounds

        # likewise for the other three quadrants
        newB = bounds.adjusted(left = n.a, bottom = n.b)
        if cBounds.intersects(newB):
            cand, dist = self.__nearest(n.NE, a, b, dist, cand, newB)
            cBounds.adjust(dist)

        newB = bounds.adjusted(right = n.a, top = n.b)
        if cBounds.intersects(newB):
            cand, dist = self.__nearest(n.SW, a, b, dist, cand, newB)
            cBounds.adjust(dist)

        newB = bounds.adjusted(right = n.a, bottom = n.b)
        if cBounds.intersects(newB):
            cand, dist = self.__nearest(n.NW, a, b, dist, cand, newB)
            # no need to update circle's bounds after last quadrant

        return cand, dist # best candidate seen so far
```

In the `Grid` class, you saw how the `intersects()` method determines whether the query circle's bounds intersect a grid cell and, therefore, whether to consider the points within the cell. `QuadTree`'s `__nearest()` method also uses the same `intersects()` method to determine whether the query circle's bounds intersect any of the quadrants below the current node and, therefore, whether to continue the recursion into that quadrant. To make that determination, `__nearest()` must manage `Bounds` objects for both the query circle and the quadrants associated with node `n`. The `bounds` parameter passed from the parent holds node `n`'s boundaries, and the `cBounds` variable is set to be the query circle's bounds using the new nearest distance to `n`. As the algorithm descends into each node's quadrants, these bounds will be updated, as described shortly.

Each return from the four possible recursive calls to `__nearest()`returns either the same candidate and distance that were just passed to the `__nearest()` recursion or a closer candidate and distance that the recursion discovered. Because it is possible that a recursive call may have decreased the distance to the nearest candidate, `__nearest` also invokes

`CircleBound`'s `adjust()` to adjust the bounding box around the current query circle. Having possibly checked all four quadrants, `__nearest()` then returns the best candidate and distance that was found.

There is some subtlety in how `__nearest()` uses `Bounds.intersects()`. As previously discussed, `Node` objects do not explicitly store the specific boundaries of the rectangle of space that they cover. Rather, each node stores just two of the boundaries, defined as appropriate, by the *a* coordinate (for either the left or right boundary) and by the *b* coordinate for either the top or bottom boundary.

How do you determine all four boundaries that define the rectangle surrounding each node? Consider Figure 12-26, which shows a `QuadTree` containing just three nodes: a root, G, drawn in green; its northeast child, P, drawn in purple; and P's southeast child C, drawn in (cobalt) blue. The nodes form a grandparent (G), parent (P), and child (C) relationship. The figure also shows the Cartesian coordinates for each of the three points.

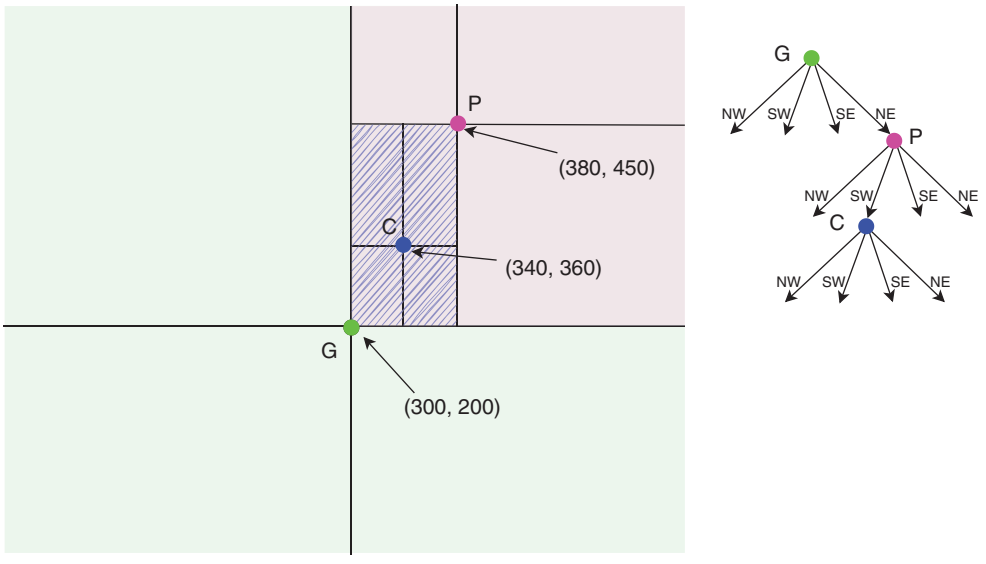


FIGURE 12-26 The intersection of ancestor quadrants forms a child's quadrant

What are the left, right, top, and bottom bounds of G's region of space? Because G is the root, its region of the plane is unbounded. In other words, its bounds are

left: `-math.inf`, right: `math.inf`, top: `math.inf`, bottom: `-math.inf`

When the search recursively descends to G's northeast child, node P, what are P's bounds? P's top and right bounds are still at infinity. However, P's left boundary is at G's *a* coordinate, 300, and P's bottom boundary is at G's *b* coordinate, 200. In other words, the bounds of P's region are

left: 300, right: `math.inf`, top: `math.inf`, bottom: 200

Taking one more step, what happens when the search recursively descends to P's southwest child, C. What are C's bounds (as shown in cross-hatched blue)? C's top boundary is defined by P's *b* coordinate 450, and C's right boundary is defined by P's *a* coordinate, 380. C's other boundaries are inherited from P. In other words, the bounds of C's region are

left: 300, right: 380, top: 450, bottom: 200

Each time you descend to a deeper node in the quadtree, the bounds for that child node are refined by replacing the appropriate two of the four bounds with the coordinates of the current node. In other words, the bounds of a node in the quadtree are formed by the intersection of the quadrant bounding boxes in the path from the root to that node.

Based on this understanding, take another look at the code for `__nearest()` in Listing 12-28. For each of the four child quadrants, the method uses `Bounds.adjusted()` to create a new `Bounds` object, `newB`, appropriately updated to reflect the bounds of that child based on the bounds of its parent. That `newB` object determines whether the search circle's bounds intersect the child's quadrant's bounds.

Pulling It All Together: Implementing QuadTree's findNearest()

As shown in Listing 12-29, the public `findNearest()` method proceeds as follows. First, using the `__nearPoint()` method, it finds a candidate node, `ans`, by recursively descending from the root as far as possible into the quadtree. Then it computes the distance from the query point to the candidate node. This is the best answer so far, even though it's a rough approximation (unless the query point matches one of the points exactly). The candidate node and the distance from it to the query point define a query circle centered at the query point, with a radius equal to the distance to the initial candidate node.

LISTING 12-29 The `findNearest()` Method of `QuadTree`

```
class QuadTree(object):
…
    # returns a,b,data,distance of point nearest to query a,b.
    def findNearest(self, a, b):
        if not self.__root: return None # No points yet!

        # Descend the tree and find the leaf node nearby
        # to this query point. This quickly sets an upper
        # bound on the nearest possible point
        ans  = self.__nearPoint(self.__root, a, b)
        dist = self.__distance(a, b, ans.a, ans.b)

        # Now we will descend the tree once more, refining
        # the candidate by progressively shrinking the radius.
        # The bounds of the root node are infinite.
        bounds =  Bounds()
        ans,dist = self.__nearest(self.__root, a,b, dist, ans, bounds)

        return ans.a, ans.b, ans.data, dist
```

Next, the `findNearest()` method constructs a `Bounds` object for the infinite extent of the root node. The root node, the query point, the current best candidate answer, and the infinite bounds are passed to the `__nearest()` method to begin the recursive search, which descends into any quadrant that is overlapped by the current query circle. Each time it encounters a `Node` that is closer to the query point than the current candidate, the candidate is replaced, and the radius of the query circle gets smaller. The result of the recursive call on the root node provides all the return values for the complete search.

The QuadTree Visualization tool makes it easier to see how the nodes are visited, how the distances change the query circle bounds, and how the bounds overlap. We show an example using the tool of a find nearest search starting, in Figure 12-27. It begins by descending the 7-node quadtree to find the leaf node closest to the *a*, *b* query point. Starting at the root, P0, it goes to the southeast quadrant to visit P2. The query point lies in P2's northwest quadrant, so `__nearPoint()` descends to P5. P5 is a leaf node, so returns it as the answer, `ans`.

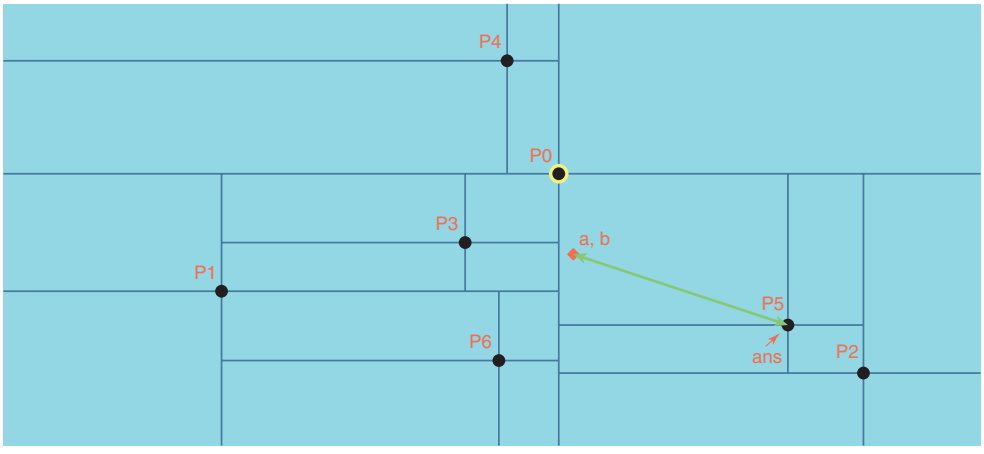


FIGURE 12-27 Finding the `nearPoint` during a find nearest search

Finding the near point and storing it in `ans` does not involve any distance calculations, only coordinate comparisons. The `findNearest()` method then computes the distance from the query point to the `ans`, as shown by the green line in Figure 12-27. The resulting `dist` measure is passed to the first call to the `__nearest()` method as the query circle radius (the distance from the current `cand` to the query point).

The first call to `__nearest()` examines the root node, P0. After checking the base cases of visiting an empty leaf node or finding an exact coordinate match where `dist` is 0, the method measures the distance from P0 to the query point. Figure 12-28 shows this measurement and makes it clear that P0 is closer than the candidate, `cand`, found by `__nearPoint()`, P5.

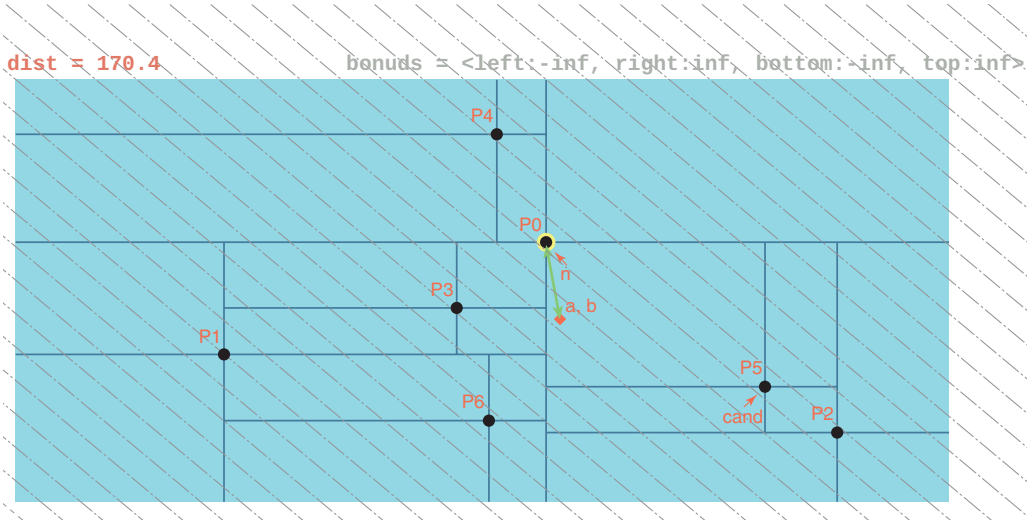


FIGURE 12-28 Visiting the root node during a find nearest search

The visualization tool also displays the various `Bounds` objects used by `__nearest()`. Initially, only the `bounds` value passed as an argument appears and has infinite extent. Figure 12-28 shows the `bounds` value both at the top right as a string and with diagonal hashing in the same color over the entire display.

The distance from the query point (*a*, *b*) to P0 is 62.2, which is less than the `dist` measure to the `cand` point shown in the upper left. That means the `__nearest()` method replaces `cand` with the current node, P0, and updates the `dist` measure, reducing the query circle. That new radius is what's used to compute the circle bounds, `cBounds`, which is used to determine what quadrants to explore.

Figure 12-29 shows the situation when `__nearest()` is being called on the southeast child of the root. The `n` pointer indicates the current node, P2, while `cand` still points at P0. At this point, three `Bounds` objects are displayed (in addition to a faded version of the infinite bounds of the root node). The `bounds` region was passed from the calling routine and covers the whole southeast quadrant of P0. The `cBounds` region shows the query circle bounding box around *a*, *b*. That was made just large enough to include P0, the root and the current best `cand` point. It's shown in a different color and with a different angle for the hash marks. Finally, the `newB` region shows the region currently being checked to see whether it should be searched, P2's southwest quadrant. It's displayed in a third color and different hash angle. It's bounded on three sides by its point's coordinates and the bounds inherited from its parent, P0. Because `newB` doesn't overlap the `cBounds` region, it's clear that the southwest quadrant of P2 could not contain any points closer than current best

cand point. Thus, the __nearest() method does not make a recursive call for the southwest quadrant of P2. It will, however, make a call for the northwest quadrant of P2 and visit node P5 because that quadrant does overlap the cBounds region.

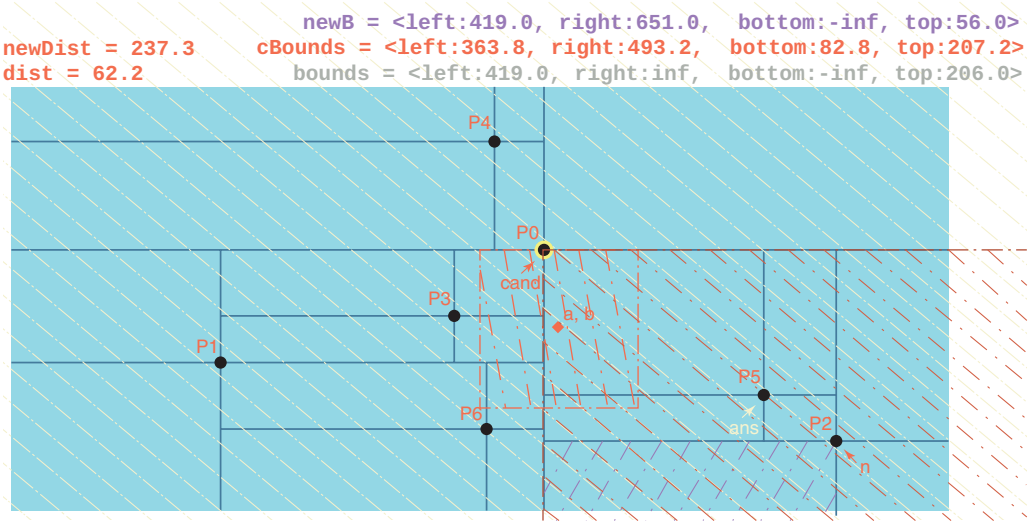


FIGURE 12-29 Eliminating quadrants during find nearest search

Although the overlapping information in the visualization tool may be confusing, the eliminated quadrants are somewhat easier to see. P2's SW, NE, and SE quadrants are eliminated, as are P5's NE and SE, P6's SW and SE, P1's NW and SW, P3's NW and SW, and P4's NW and NE quadrants. That's about half of the quadrants, which is similar to how binary search removes half of the range of cells to explore.

Efficiency of findNearest()

The time complexity of findNearest() is on average O(log N). The __nearPoint() method, which behaves similarly to findExact(), should be able to identify a candidate Node in O(log N) time, unless the tree is highly unbalanced. Likewise, a good initial candidate will result in a tight query circle, enabling __nearest() to efficiently descend back down into the tree considering just a small number of adjacent QuadTree cells, again in O(log N) time. An unbalanced tree or a degenerate distribution of data could result in O(N) time for each search. For example, if all the QuadTree nodes are distributed in a circle, and the query point is at or very near the center of the circle. Then, __nearest()has no alternative but to visit every Node in the QuadTree because the search circle necessarily overlaps each nonempty QuadTree cell.

Theoretical Performance and Optimizations

`PointList`, `Grid`, and `QuadTree` present three different approaches to storing and querying spatial data points. Table 12-1 summarizes the average Big O complexity of each operation implemented across spatial data structures.

Table 12-1 Big O Performance of the Spatial Data Structures

	PointList	Grid	QuadTree
`insert()`	O(N)	O(N)	O(log N)
`findExact()`	O(N)	O(N)	O(log N)
`findNearest()`	O(N)	O(N)	O(log N)
`delete()`	O(N)	O(N)	O(log N)
`traverse()`	O(N)	O(N)	O(N)

Could the Big O of `PointList` and `Grid` be improved to be competitive or superior to `QuadTree`?

One easy enhancement for `PointList` would be to use a hash table (such as a Python dictionary) instead of a list to store the entire collection of points in a `PointList`. Doing so would result in improved O(1) performance for `insert()`, `findExact()`, and `delete()`. The performance of `findNearest()` and `traverse()`, however, would not be improved because both require considering every point in the `PointList`.

The `Grid` implementation already uses a hash table to map grid cell coordinates to lists of points stored in each grid cell. Because the grid spacing is fixed, however, the list in each grid cell contains O(N) points, making the find and delete methods also O(N). By replacing the list in each grid cell with a dictionary, the performance of `findExact()` and `delete()` can also be improved to O(1). Here, too, `findNearest()` remains O(N) because the nearest neighbor to query point is found by examining all the O(N) points stored in one or more of the grid cells.

Even with these improvements, `QuadTree`'s theoretical O(log N) performance for `findNearest()` is difficult to surpass.

Practical Considerations

In this chapter we covered three different ways to organize spatial data: brute force using a list of points, uniform grids, and quadtrees. If you have dozens or even hundreds of spatial data points, then the brute-force approach may be sufficient.

But what should you use if you have many more points—say, tens of thousands, hundreds of thousands, or even millions? Understanding the Big O of a data structure provides some guidance. But remember that the Big O of an algorithm informs you only of the *shape* of the curve. It (intentionally) excludes an important constant factor that can make a difference in real-life applications.

Let's examine some practical time measurements of the performance of the `Grid` and `QuadTree` data structures. To do this, we ran experiments where we loaded points into a `Grid` or a `QuadTree` and then recorded the time to do 1,000 `findNearest()` queries. Geographic data points were randomly selected with a uniform distribution across a rectangle that roughly encompasses the continental United States. Then 1,000 uniformly distributed query points were randomly selected across that same rectangle, and the time to find the nearest data point was recorded.

Figure 12-30 shows the results of these informative experiments. The X axis shows the number of points loaded into a `Grid` or `QuadTree` data structure. The Y axis shows the total time in seconds to run 1,000 `findNearest()` queries against the data. For grids, we created a `Grid` object where the cells were sized at 0.5 degrees in latitude and longitude—resulting in about 6,500 grid cells covering the United States. A second grid object was created where the cells were of size 1.0 degrees latitude and longitude—resulting in about 1,600 grid cells. Finally, we also created a `QuadTree`, also loaded with varying numbers of points, but able to grow in depth as more points were inserted.

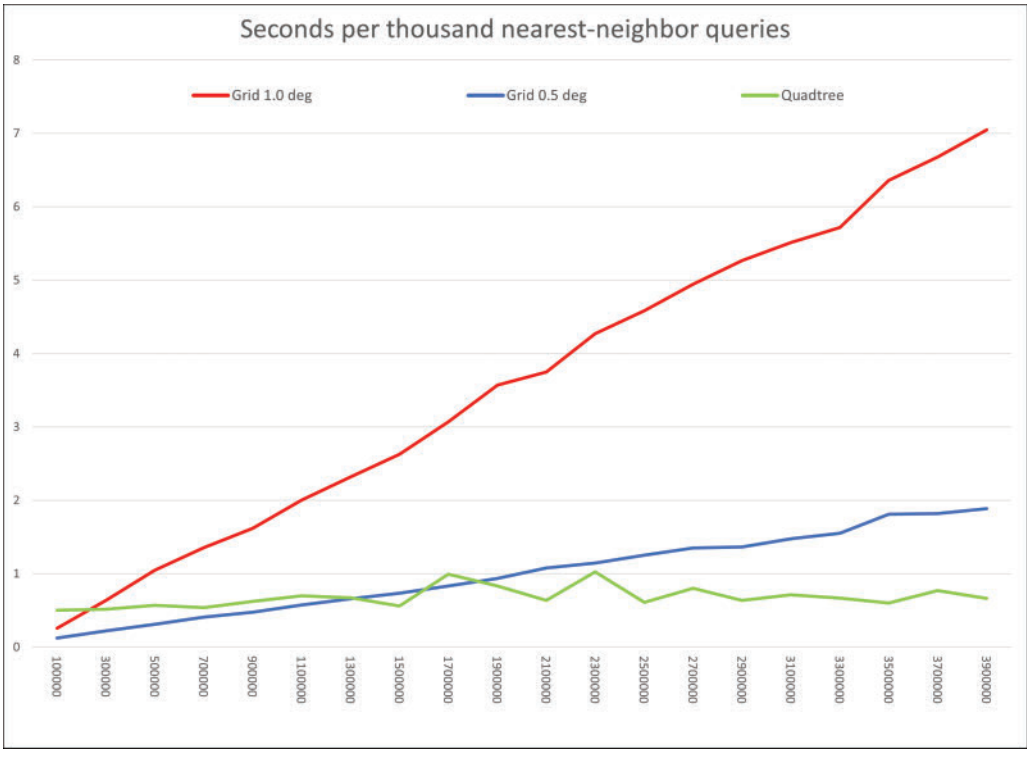


FIGURE 12-30 Performance of `findNearest()` on different-sized data sets

Not surprisingly, the red and blue plot lines are quite linear, confirming the analysis of their O(N) performance. Their slopes reflect the respective sizes of the grid cells and, therefore, how many points are stored on average per cell. In the long run, as the data

structures are loaded with millions of points, `QuadTrees` win, with their very slow O(log N) growth as the number of points increases.

Interestingly, if you have only 100,000 or fewer points, then grids of cell size 1 or 0.5 degrees can perform better than quadtrees. If you were willing to expend more memory on grid cell storage, then an even finer grid would result in better linear performance, by reducing the leading constant factor. Be careful, however, in ensuring that the data won't grow much larger in software that uses grids like these.

These experiments were somewhat artificial because real-life data is usually not uniformly distributed across the map. Businesses, people, vehicles, and other real-life objects tend to be very clustered in space. If that's what your data looks like, then you should be especially careful to measure the performance of your spatial data structure under realistic conditions.

Further Extensions

We have explored the basics of spatial data: what it is, how you might organize points in a data structure, and how those organizations impact the four basic operations. The search operation, common to all data structures, needed extensions for exact matches and nearest match. Spatial data, however, is used for many different purposes, and those applications often require more capabilities than what we've described here. Let's explore a few extensions.

Other Operations

Beyond the `insert()`, `findExact()`, `findNearest()`, `delete()`, and `traverse()` operations we covered, there are many other possible—and useful—operations on spatial data. Here are just a few:

- Instead of finding just the one nearest neighbor to a point, find the K nearest neighbors to the query point.
- Traverse the points in order of increasing distance from a query point.
- Find all points within a specified distance from a query point.
- Find all points within a rectangular region of space.
- Find all points that lie within a polygonal boundary such as the borders of a country.
- Find the closest pair of points.

Each of these operations needs special algorithms to identify candidate points, determine whether they affect the output, and determine how to incorporate them in the output. Making them efficient is a challenge.

Higher Dimensions

The data structures covered in this chapter all deal with two-dimensional data—points on the plane or on the surface of the globe, specified with just two coordinates. Point data

in three dimensions is also common. For example, a third dimension could correspond to the height of an object above the plane or globe. A natural extension to the `Grid` class could be used to store and query three-dimensional points. In effect, a three-dimensional grid would look like a regularly spaced collection of cubes (instead of squares).

Quadtrees also have a natural analog in the three-dimensional world, namely **octrees**. With quadtrees, each point recursively subdivides space into four quadrants. With octrees, each point subdivides space into eight *octants*. Analogous to map data, octrees are efficient at representing and searching three-dimensional point data that clusters in space.

It is possible to use spatial data structures to represent point data in even higher dimensions. You can use ***k*-d trees** (that is, *k*-dimensional) to represent points in arbitrarily high dimensions. These *k*-d trees are usually implemented by recursively subdividing space into half along carefully chosen dimensions.

These higher dimensional data structures can support the same types of operations, including insert, traverse, find exact, and find nearest.

Implementing and using data structures for high-dimensional data brings theoretical and engineering challenges. In a discussion of the distance between points in two-dimensional and three-dimensional space, you gain an intuitive understanding of what "distance" means. But when you consider additional dimensions, the notion of spatial distance is not meaningful and must be addressed carefully. For example, a fourth dimension of time could be very useful, but then what distance function would be useful in a space consisting of *x*, *y*, *z*, and time?

High-dimensional data also suffers from the "curse of dimensionality." If you measure the Euclidean distance between two points of very high dimension, the differences in distances among many pairs of points can be very small. Another way to say this is that if you have high-dimensional data, all points in your data set can seem equally or nearly equally distant from each other! Therefore, although quadtrees and even *k*-d trees for low-dimensional data can be searched in O(log N) time, as the dimension of your data increases, it becomes necessary to search a larger and larger proportion of all the tree's cells and data points to find a nearest neighbor!

Summary

- Spatial data structures store point data based on two-dimensional position.
- Positions can correspond to Cartesian *x*, *y* coordinates, or geographic latitude and longitude coordinates on the Earth.
- Distance between points is measured differently depending on the coordinate systems.
  - The Euclidean distance computes the straight-line distance between points on a plane.
  - The haversine distance computes the distance along the great circle connecting points on a globe.

- Point data can be stored in different kinds of spatial data structures.
- A brute-force approach stores points in a list or in an array such as the `PointList`.
- A `Grid` structure divides up the plane or globe into cells of equal size and stores point data as lists or arrays for each cell.
- Quadtrees organize point data as a tree.
- Each node in a quadtree stores data for a particular point in space.
- A quadtree node can have up to four child nodes corresponding to the four quadrants relative to its position.
- Grid structures do not adapt to point data that clusters spatially, but quadtrees do.
- All spatial data structures provide methods to insert new point data, search for an exact match, and search for the nearest point data to a query point.
- Grid searches begin in the query point grid cell and expand outward by layer.
- Quadtree searches begin at the root node and follow child quadrants containing the query point.
- Search performance is improved by eliminating grid cells or quadrants that lie outside of the current query circle.
- Query circles have a different shape when geographic coordinates are plotted with parallel lines for longitude meridians as in the Mercator projection.
- Rectangular bounding boxes simplify testing for the intersection or complete enclosure of one region and another but can only approximate the boundary of a query circle.
- Different spatial data structures yield different Big O performance.
- Simple point lists and grids take O(N) time for search and deletion (and insertion when duplicate points must be detected and updated).
- Quadtrees take O(log N) time for insertion, search, and deletion.
- The performance of spatial data structures can be improved by augmenting the structure with O(1) data structures such as hash tables.
- It is important to perform measurements with real-life data for your application to determine the best approach for your scale and type of data.
    - Uniformly distributed data can be organized well with grids, and the grid cells can be made small to improve performance.
    - Nonuniform data, or data of unknown size or distribution can be better organized with quadtrees because the quadtree adaptively subdivides space in regions that contain more data.
- Octrees can store and search three-dimensional data.
- There are a wide variety of other operations needed for spatial data applications.

Questions

These questions are intended as a self-test for readers. Answers may be found in Appendix C.

1. What distinguishes the keys for spatial data structures from those used for other structures like trees, arrays, and lists?
2. _____ coordinates are used to represent a point on the surface of the earth.
3. When you're making data structures for two-dimensional spatial data, what things must be initialized when creating the (empty) structure for all of them?
4. Bounding boxes are used in spatial calculations because
   a. rectangles are the only viable shapes that work with spatial coordinates.
   b. checking whether two rectangles overlap is computationally simpler than checking whether an ellipsoid overlaps with a rectangle.
   c. using bounding boxes is a convenient way to put a bound on the worst-case complexity of a spatial computation.
   d. the haversine distance works only with rectangular bounds.
5. True or False: Lists of points are never an acceptable data structure for searching spatial data.
6. What advantage does a grid have over a point list in storing spatial data?
7. What advantage does a quadtree have over a grid in storing spatial data?
8. To implement efficient access to the cells of a grid, you use
   a. a two-dimensional array or a list to store each cell of the grid containing data points.
   b. a quadtree to locate the quadrant storing the cell.
   c. a list of point coordinates sorted by each point's row and column number.
   d. a hash table that maps keys consisting of row, column tuples to the list of corresponding points in that grid cell.
9. Using Big O notation, how long does it take to find the nearest neighbor to a query point in each of the following structures holding N points?
   a. a point list
   b. a grid
   c. a quadtree

10. True or False: A quadtree supports data points with any number of dimensions.
11. Query circles
    a. are used in find exact searches to determine whether a point matches the query.
    b. are used by quadtrees to bound the size of each node's quadrants.
    c. are used by grids to limit which other grid cells should be traversed.
    d. start large and decrease in radius as find nearest searches points in grids and quadtrees.
12. True or False: A quadtree uses layers to gradually expand the search for the nearest point.
13. What are the quadrants of a node? How are points lying on the boundary between two quadrants placed into the appropriate quadrant?
14. The depth of a well-balanced quadtree of N points is proportional to
    a. N
    b. N/4
    c. $\sqrt[4]{N}$
    d. $\log_4 N$
15. True or False: Quadtrees are superior because their worst-case behavior is always better than point lists or grids.

Experiments

Carrying out these experiments will help to provide insights into the topics covered in the chapter. No programming is involved.

12-A Which of the three spatial data structures discussed in this chapter would use the least amount of memory per spatial data point? Which would use the most memory for the same number of points?

12-B Consider the problem of finding the item the farthest from a query point. How would you approach this problem, and which data structure would be more appropriate: a grid or a quadtree?

12-C The quadtree makes finding the nearest point very fast. Could the principles used in `findNearest()` be applied to finding the nearest key in other data structures? More specifically, how could you speed up the search for the nearest key in a binary search tree? Does the answer change if the keys are strings instead of integers?

12-D Consider the problem of finding the K nearest neighbors of a query point. How would you implement this method using your choice of spatial data structure, and what supplementary data structure might you use to make the implementation efficient?

Programming Projects

Writing programs to solve the Programming Projects helps to solidify your understanding of the material and demonstrates how the chapter's concepts are applied. (As noted in the Introduction, qualified instructors may obtain completed solutions to the Programming Projects on the publisher's website.)

12.1 Modify the `Grid` class to support a `delete()` method that is given the coordinates of a point to delete. If the point is found in the `Grid`, then the point is deleted from the appropriate cell, and the associated data is returned to the client. If the point is not found, then `None` is returned to the client.

12.2 Modify the `QuadTree` class to support a `delete()` method. Because a `Node` object can't easily be removed from a `QuadTree`, consider adopting the representation that a `Node` with `None` for its data denotes a "deleted" node. When implementing `delete` this way, you must modify the class's other methods to be consistent with this new representation. If the point is not found, then `None` is returned to the client.

12.3 Modify `Grid` and `QuadTree` to store more than one datum at a specific coordinate. In other words, change them to allow duplicate keys. Be sure to modify all methods that could be impacted by this change.

12.4 Create a subclass of `QuadTree` called `BulkQuadTree`. The new subclass has a constructor that accepts a sequence of points in bulk and then attempts to build a well-balanced quadtree. Add measurement methods to the quadtree to quantify its balance and measure the balance of quadtrees using large collections of random points and large collections of sorted points.

12.5 Add a method to `Grid` that moves the data associated with a point at a specified location to a new location.

This page intentionally left blank

CHAPTER 13
Heaps

IN THIS CHAPTER

- Introduction to Heaps
- The Heap Visualization Tool
- Python Code for Heaps
- A Tree-Based Heap
- Heapsort
- Order Statistics

Keeping items in priority order is useful in many contexts. At school and at work, we all deal with numerous tasks with various deadlines. Typically, the nearest deadlines get the highest priority. Sometimes, the importance of the task or the severity of not completing it outweighs a later deadline, so you move it up in priority and work it before tasks with shorter deadlines. When doctors and nurses triage patients arriving at a medical facility, they weigh decisions about the severity of the injury or illness and how long the patient can wait before treatment must start.

One characteristic of organizing tasks by priority is that you only need to know what the highest priority task is. That's the one that will be worked first. Perhaps you need to know what the first two or three tasks are because they can all be worked at the same time (or worked in smaller increments in rotation).

What's less obvious in prioritizing is that you don't need to know the precise order of the remaining tasks. You can set them aside and deal with them after higher-priority tasks are completed. That means you don't have to fully sort all the tasks; a partial sort allows you to focus on the important ones.

You learned in Chapter 4, "Stacks and Queues," that a priority queue is a data structure that offers convenient access to the data item with the smallest (or largest) key. In those implementations, all the items are stored in a sequence sorted by their priority.

A **heap** organizes items in a *partially sorted order* inside a binary tree. That might seem as if it wouldn't be as useful as the binary search and other trees you saw in Chapters 8, 9, and 10, but it is surprisingly powerful. The partial sorting means the heap is faster in some operations. That makes

it particularly well suited for implementing priority queues, where it is much faster than maintaining a fully sorted sequence.

Heaps are used for more than just priority queues. They can be implemented with arrays, enabling an algorithm called a **heapsort** that competes with the quicksort both in speed and its low memory requirements. They are perfect for getting order statistics, such as the top 1 percent of a population of items when the rest won't be used (for example, finding the most important key words in a billion text documents from the Internet to summarize their content).

NOTE

Don't confuse the term *heap*, used here for a special kind of binary tree, with the same term used to mean the portion of computer memory available to store some kinds of data as the program runs. The computer memory heap is a section of random-access memory used to hold dynamically allocated and deallocated objects.

Introduction to Heaps

A **heap** is a binary tree with these characteristics:

- It's **complete**. This means all its levels are completely filled, except, perhaps, the leaf level. If you read the items from left to right across each row, either the whole row is full, or all the items are on the left side of the bottom row. Figure 13-1 shows examples of complete and incomplete trees.
- Each node in a heap satisfies the **heap condition**: every node's key is larger than (or equal to) the keys of its children. This makes the tree partially ordered.
- It's (usually) implemented as an array. This isn't a requirement, but it is a desirable feature in most use cases. In Chapter 8, "Binary Trees," we described how binary trees can be stored in arrays, rather than using references to connect distinct node objects.

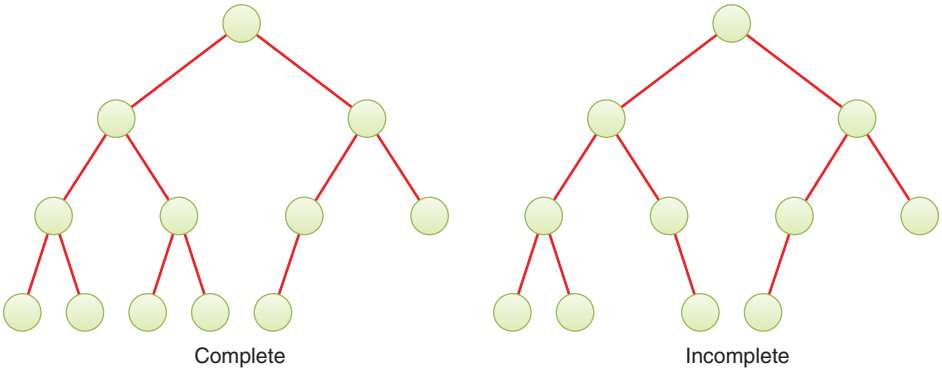


FIGURE 13-1 Complete and incomplete binary trees

Figure 13-2 shows a heap as both a tree and the array used to implement it. The array is what's stored in memory; the heap tree is only a conceptual representation. Notice that the tree is complete and that the heap condition is satisfied for all the nodes.

Another big difference from the other trees we've studied: the keys in the nodes are not in sorted order, neither in the array, nor in the binary tree, at least not the way they would be ordered in a binary search tree.

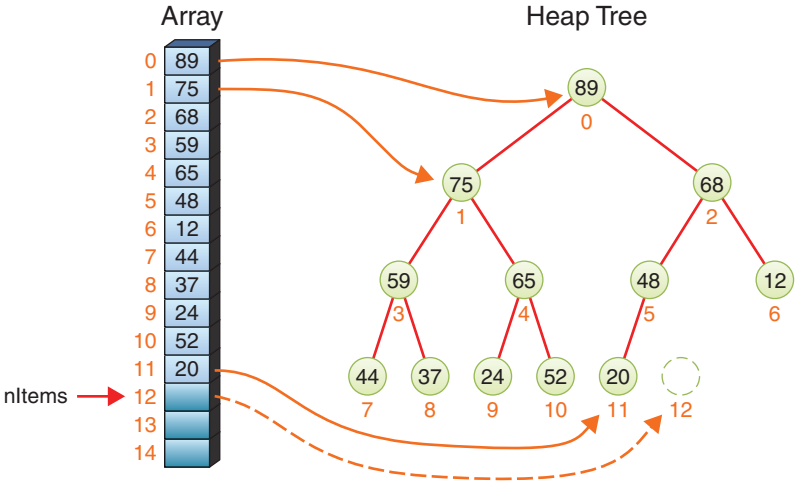


FIGURE 13-2 A heap array and its corresponding binary tree

The fact that a heap is a complete binary tree means there are no "holes" in the array used to represent it. Every cell is filled, from 0 to `nItems` – 1 (`nItems` is 12 in Figure 13-2). When the next item is inserted, the cell at index `nItems` is filled, in the way that items were inserted at the end of arrays in Chapter 2, "Arrays."

We assume in this chapter that the maximum key (rather than the minimum) is stored in the root. A priority queue based on such a heap is a *descending-priority* queue. (We discussed ascending-priority queues in Chapter 4.)

Priority Queues, Heaps, and ADTs

A priority queue is an abstract data type (ADT) offering methods that allow removal of the item with the maximum (or minimum) key value, insertion, and sometimes other operations such as traversal. We describe heaps in this chapter, which are frequently used to implement priority queues. There's a very close relationship between a priority queue and the heap used to implement it. The code fragment of Listing 13-1 shows the equivalence.

The three methods for the `PriorityQueue` class are identical to the methods for the underlying `Heap` class, so it can be implemented as a subclass. This example and the implementation in Chapter 4 make it conceptually clear that a priority queue is an ADT that can be implemented in a variety of ways, whereas a heap is a more fundamental kind of data

structure. In this chapter, for simplicity, we simply show the heap's methods without the priority-queue wrapping.

LISTING 13-1 Implementing Priority Queues Using a Heap

```
class Heap(object):
   def __init__(self, size=2):  # Heap constructor
      self._arr = [None] * size # Heap stored as a list/array
…
   def insert(self, item):      # Insert a new item in a heap
…
   def remove(self):            # Remove top item of heap and return it
…

class PriorityQueue(Heap):      # Create a priority queue, using a heap
```

Partially Ordered

A heap is *partially ordered* compared with a binary search tree, where all a node's left descendants have keys less than all its right descendants. As you saw in Chapter 8, the full ordering of binary search trees allows you to traverse the nodes in the order of their keys by following a simple algorithm.

In a heap, traversing the nodes in order is difficult because the organizing principle (the heap condition) is not as strong as the organizing principle in a binary search tree. All you can say about a heap is that, along every path from the root to a leaf, the nodes are arranged in descending order. As you can see in the tree of Figure 13-2, the nodes to the left or right of a given node, or on higher or lower levels—provided they're not on the same path—can have keys larger or smaller than the node's key. Except where they share the same nodes, paths are independent of each other.

Because heaps are partially ordered, some operations are difficult or impossible. Besides its failure to support traversal ordered by the keys, a heap also does not allow convenient searching for a specified key. The reason is that there's not enough information to decide which of a node's two children to pick in trying to descend to a lower level during the search. It follows that a node with a specified key can't be deleted, at least in O(log N) time, because there's no way to find it. These operations can be carried out by looking at every cell of the array in sequence, but this is only possible in slow, O(N), time.

Thus, the organization of a heap may seem dangerously close to randomness. Nevertheless, the ordering is just sufficient to allow fast removal of the maximum node and fast insertion of new nodes. These operations are all that's needed to use a heap as a priority queue, as a sorting mechanism, and for finding certain members of a distribution. We discuss briefly how the two core operations are carried out and then see them in action in a visualization tool.

Insertion

Inserting an item in a heap is straightforward. You know it means adding a node to the tree, and the only place new nodes can go is at the leaf level. Because the tree is complete, the new node is added just to the right of the last node if the bottom row is not full, or as the left child of the leftmost node of a full bottom row. In Figure 13-2, there are 12 nodes in the tree. Adding a thirteenth item means filling the array at index 12, which corresponds to the dashed node in the tree, just to the right of index 11.

What about the heap condition? If you're lucky, the inserted item has a key less than its parent (or it is the root node), and you are done. You can't count on being lucky, however, so you need to compare the keys. If the newly inserted leaf has a key greater than its parent, you can swap the two items. Figure 13-3 shows an example where item 80 is inserted at the bottom in cell 12. The blue link indicates which two nodes need to be compared.

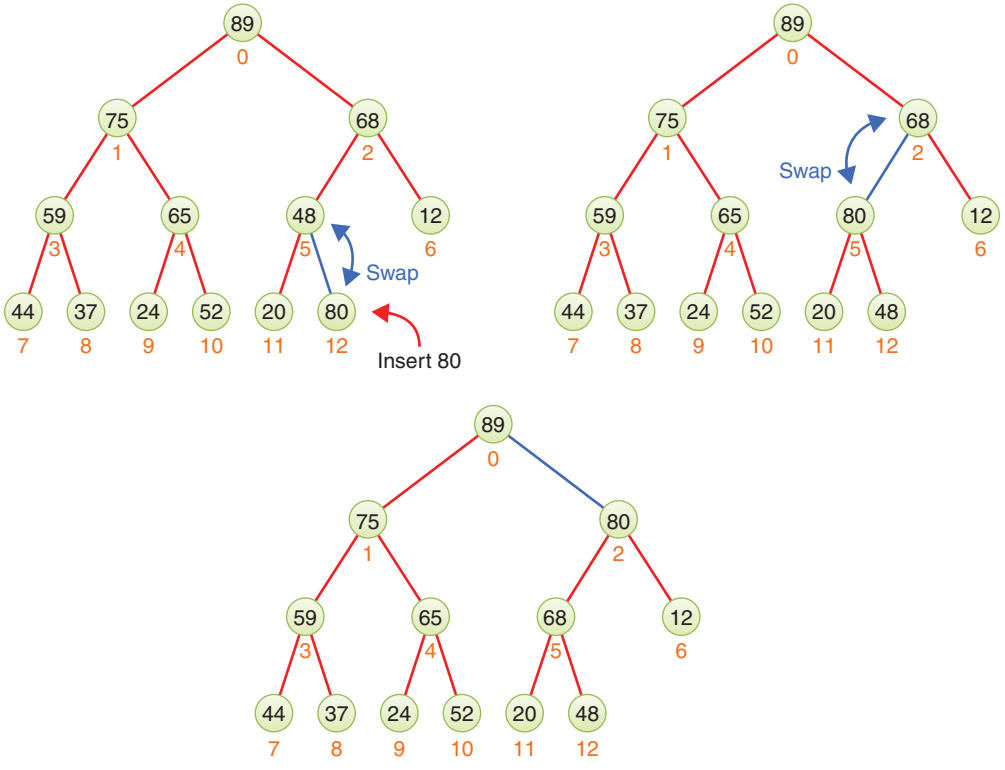


FIGURE 13-3 Inserting an item in a heap and sifting up

Because the inserted item's key, 80, is greater than 48, the key of its parent in cell 5, the items must be swapped. After the swap, item 80 now has items 20 and 48 as children, and both keys satisfy the heap condition because they are smaller. Whenever you swap a

parent and child key, you only increase the value of the parent key. That means that you can never break the heap condition with the other child—the one that wasn't swapped.

After moving the inserted item up to the second from bottom row, you still need to check whether it satisfies the heap condition in its new position. That's shown at the upper right of Figure 13-3. Comparing 80 with 68 reveals that the two nodes linked by the blue line need to be swapped too.

After the second swap, item 80 lands in cell 2, as shown at the bottom of the figure. Comparing item 80 with its new parent, item 89 in cell 0, reveals that it now satisfies the heap condition. No more swapping is needed.

Moving items up the tree until they satisfy the heap condition is called **sifting up**. This operation is very similar in concept to one iteration of the bubble sort. The operation has also been called by other names such as *bubble up*, *trickle up*, *percolate up*, *swim up*, *heapify up*, and *up heap*. Because you apply it to a single new item on a heap that already satisfies the heap condition, only a single pass toward the root node is needed.

Removal

Removal means removing the item with the maximum key. In other data structures, like trees and hash tables, you use a `delete()` method to remove an item with a specified key. Heaps typically support the removal of the maximum key only. This item is always kept at the root node, so finding it is easy. The root is always at index 0 of the heap array.

The problem is that after the root is gone, the tree is no longer complete. Alternatively, from the point of view of the heap array, there's an empty cell. This "hole" must be filled.

There are several possible approaches to filling the hole. What if you simply shifted every item in the array by one cell toward index 0? That would get rid of the hole, but it could introduce new problems. If the array contained the three keys: 50, 30, and 40, deleting 50 would leave 30, 40. That could put 30 at the root with 40 as its one child, and that would break the heap condition.

What about using a modified version of the sift up method that was used for insertion? You could sift up the child of the (empty) root with the larger key. You could then repeat the process of filling the hole created by moving up the child by moving down a level in the tree. You would continue until you move up a child node that is also a leaf node. That would certainly preserve the heap condition because you always move up the higher valued key. It doesn't appear to create any holes because it fills each vacancy it creates. Or does it?

In Figure 13-4, two very similar heap trees are shown at the left. Removing the maximum item from the top, item 75, leaves a hole. The blue arrow shows the next step is to promote item 65 in cell 1 to fill the hole. The next step promotes item 44 to fill the hole at cell 1. At the end of the top row, the final heap has four items and the heap condition is preserved. So, the approach works for this heap.

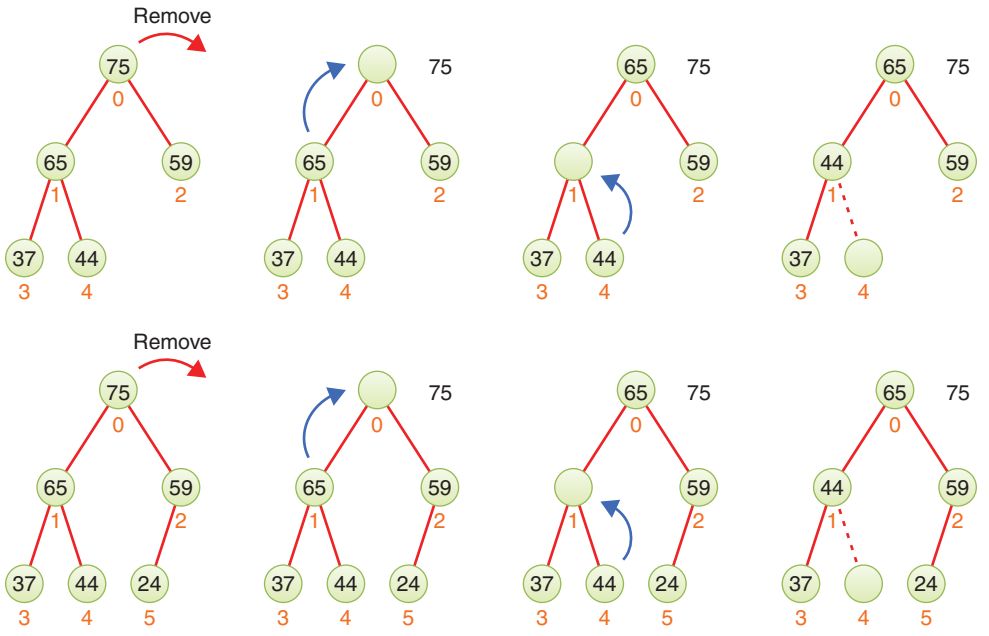


FIGURE 13-4 Two examples of filling holes with the maximum child item

In the bottom row of Figure 13-4, however, the heap has one more item. The same items are moved in the same sequence. After you move up item 44, there is an empty cell 4 but a full cell 5, item 24. That empty leaf node means the tree is no longer complete and you don't have a heap.

Simply filling with the maximum child does not work. It seems as though you would need to sift up a child item based on the path to the last node in the tree; that is the node in the highest numbered filled cell. But if you sift up a child that has a key less than its sibling, it would create a node where the parent has a key less than its child. That would mean you would have to do more swaps or moves or actions like the rotations that were used to balance AVL trees. That's getting very complicated.

Fortunately, there's a simpler solution: take the root item out of the tree to return later, replace it with the last item in the heap, shrink the heap size by one, and **sift down** the new root. This operation is very similar to the approach of filling holes and solves the problem of leaving a hole in the wrong place on the leaf layer.

To see how this approach works, let's look at an example. Figure 13-5 starts with the same initial heap as in the bottom row of Figure 13-4. The first step is the removal of item 75 from the root node. The second step takes the last item in the heap, item 24 at cell 5, and moves it to the root. The third step is shrinking the heap size by one to eliminate cell 5.

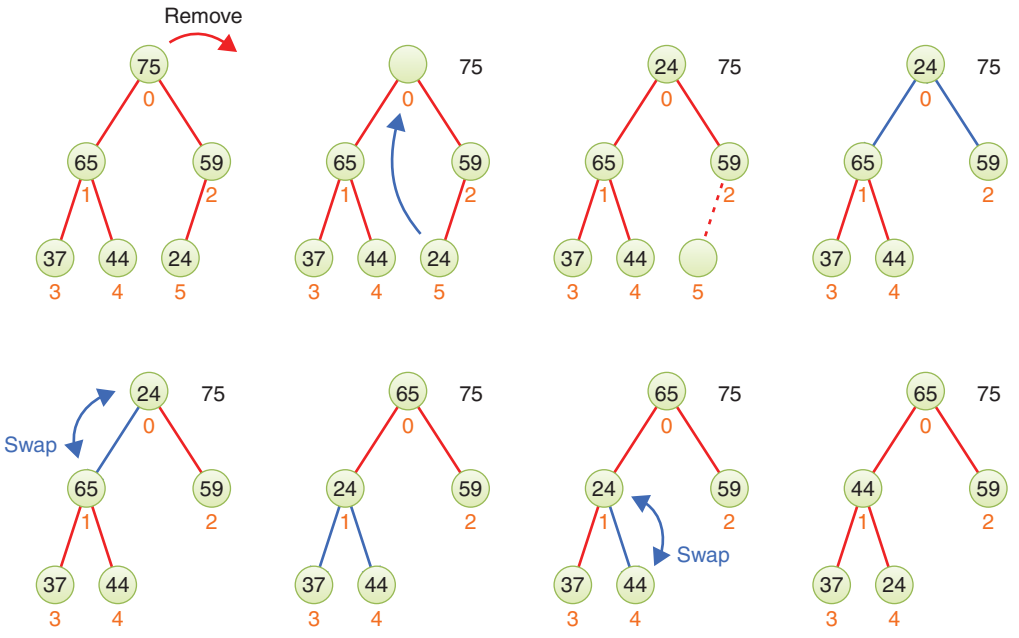


FIGURE 13-5 Replacing the root with the last item and sifting down

Now you can start the sift down process at the root node. In the upper right of the figure, item 24 sits at the root node. The blue links to its children indicate that we must look at their keys to determine which one is larger. The larger one (at cell 1) becomes the candidate for swapping.

In the next step—at the left of the second row in the figure—you compare the keys of cells 0 and 1 to find that a swap is needed. After swapping those two items, you begin the sift down process starting at cell 1. The two blue links below cell 1 show that you compare the keys of cells 3 and 4 to find which one is larger. After settling on cell 4, it's clear that another swap is needed. When that swap is made, you end up with the heap at the bottom right of the figure.

The final tree in Figure 13-5 satisfies the heap condition and is complete. If you try to sift down further, you will find that cell 4 has no children, so no more sifting can be done. In fact, you can stop the process when either a leaf node is found, or no swap is needed with the maximum child.

As you might expect, this whole process is called *sift down* (or bubble down, trickle down, percolate down). It's only slightly more complex than sift up. The difference being that there are up to two children at each node. It must choose the node with the maximum key if there are two children, and the left node if there is only one. (There can never be a node with a right node and no left node in a complete tree.) Then it compares the parent with the target child, swaps them if needed and continues, or finishes if they are already in heap order. Because it always chooses the maximum child, the swap cannot create a heap condition violation.

The sift down algorithm guarantees that the items in the nodes it visits are in heap order. It also guarantees the binary tree remains complete because removing the last node of a complete tree preserves completeness, and swapping parent and child nodes cannot create holes. Like sift up, this operation is similar to one iteration of a bubble sort. It also shares a characteristic with deletion in binary search trees where the successor node replaces the node to be deleted (without having to do rotations to rebalance the tree).

Not Really Swapped

Both the sift down and sift up algorithms are easy to understand when you visualize the item swaps between parent and child nodes like those shown in Figure 13-3 and Figure 13-5. Each swap involves two moves or copies, typically using a temporary variable. There's a more efficient way of implementing the changes, however, as shown in Figure 13-6.

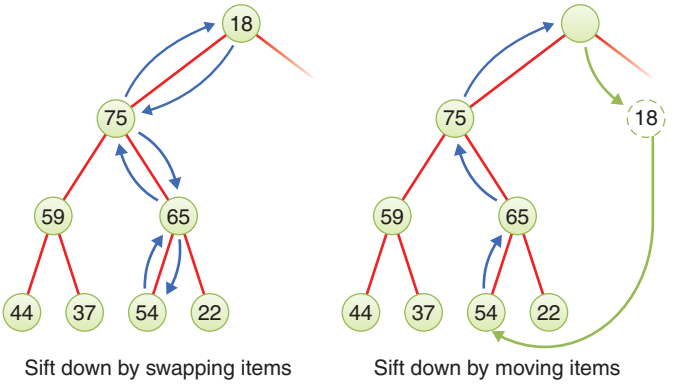


FIGURE 13-6 Sifting down with swaps versus with moves

When you sift down, the item that was inserted at the root is moved down by each swap. Every item lower in the tree is compared with the same key of the item from the root. That means you can copy the root item into a temporary storage location and simply move items up into the holes that are created by each move/copy. The process is shown on the right of Figure 13-6 and ends up being a rotation of the items.

Why is this process more efficient? In the example in Figure 13-6, six items are moved when you use swapping. Five items are copied when you move items instead of swapping; three copies up (the blue arrows), plus one copy to the temporary variable and one copy from the temporary variable to the final position (the green arrows). That's not much of a savings, but think about sifting an item down 30 levels in a large heap. In that case, swapping makes 60 copies, whereas moving makes 32. You save about half the copy operations (and close to two-thirds considering a typical swap makes three copies using a temporary variable). The same method works when sifting up a newly inserted item at a leaf node.

Another way to visualize the sift up and sift down processes being carried out with copies is to think of a "hole" or "blank"—the absence of an item in a node—moving down in a sift down and up in a sift up. In Figure 13-6, moving item 18 into the temporary storage at right leaves a hole at the root. When item 75 is copied up to the root, the hole moves

down one level. The next move up of item 65 moves the hole down another level. Eventually, the hole moves to a leaf node or a node whose child keys are smaller than the item sifting down from the root. That hole is filed with the content of the temporary storage. The holes, of course, are conceptual; copying items in memory leaves the original value in the source node.

Sifting data down the heap saves time in the way the insertion sort saved time over the bubble sort. You use many fewer copy operations to achieve the result.

Other Operations

Heaps' affinity to priority queues makes a few other operations very useful. In particular, priority queues used to manage the processes running on a computer can benefit by having methods for *peek*, *change-priority*, and *replace-max*. The peek operation is the same as for stacks and queues: it returns the key and data of the maximum keyed item without changing the heap contents.

The *change-priority* operation changes the priority of an item in the queue. This is needed when the priority of a process must be changed before it is next removed from the priority queue for running, perhaps because higher priority jobs are preventing it from being run.

The `change_priority()` method first must find the item in the heap, which could be quite slow, as mentioned in the "Partially Ordered" section. When it is found, increasing the priority means a sift up operation must follow. Similarly, decreasing the priority means sifting the item down. These changes can be performed just like insertion and removal, respectively, except that the sifting starts with an existing item anywhere in the tree after modifying its priority.

The *replace-max* operation is widely used. When operating systems take the next process item from the priority queue to run, they frequently have a process to put back in the queue to run later. The operating system usually limits the amount of time each process can run before it is suspended to allow other processes to run. The process that was running is placed back on the priority queue, perhaps with a different priority.

If you simply remove the maximum priority process/item from the heap and then insert the previously running process in the heap, there will be both a sift down for the remove operation followed by a sift up for the insert operation. A `replace_max()` operation eliminates the sift up operation by replacing the root item with the one to be inserted back in the priority queue. The new root item is sifted down into proper position, and the old root is returned, so the `replace_max()` can be used in place of `remove()`. This approach saves considerable time in switching between hundreds or thousands of processes that might run for just a few microseconds.

The Heap Visualization Tool

The Heap Visualization tool demonstrates the operations we discussed in the preceding section: it allows you to insert new items into a heap and to remove or peek at the largest item. It does not, however, implement the other operations mentioned: change priority and replace max.

When you start up the Heap Visualization tool, you see a display like Figure 13-7. The tool shows the heap both as an array and as a tree. Both are initially empty. The array is shown on the left side. It has two cells to start, and the `nItems` pointer points at cell 0 to show that no items are currently in the heap.

FIGURE 13-7 The Heap Visualization tool

The tree is shown to the right of the array. When the heap is empty, there are no nodes. Furthermore, because no tree object is needed when the tree is represented as an array, there's no box at the top like there was in the other tree visualizations.

The Insert Button

A new item is always inserted in the first available array cell—the one indicated by the `nItems` pointer. In the visualization, cell 0 is at the top of the column of array cells to align with the tree that has its root at the top. The first node goes in cell 0 and creates a root node in the tree. Every subsequent insert goes in the array as a leaf node in the tree just to the right of the last node on the bottom row of the heap. Try entering a key—an integer between 0 and 99—in the text entry box and selecting Insert. If you enter the values 33, 51, 44, 12, and 70, you should see a heap like the one in Figure 13-8.

During each insertion, you see the tool put the new item in the last array cell and last leaf of the tree, and then sift it up until it is correctly positioned according to the heap condition. You also see the array expand when needed, which we look at in the Python code.

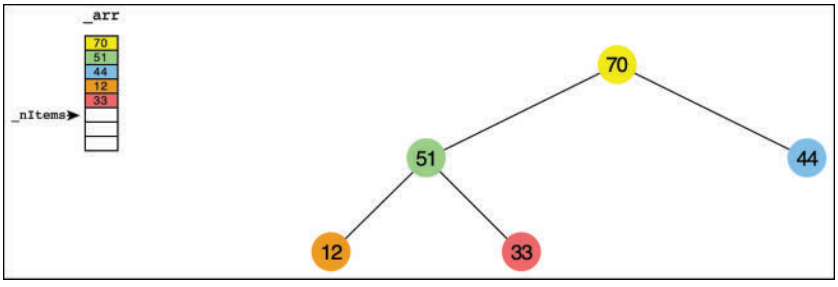


FIGURE 13-8 A small heap in the Heap Visualization tool

The Make Random Heap Button

You can create heaps of up to 31 random items for experiments. Type the number of desired items in the text entry box and then select the Make Random Heap button. The array and the tree are filled with items, some of which may have duplicate keys. Duplicates are less of an issue in heaps because they aren't used like a database where items are identified by unique keys.

The Make Random Heap clears any existing items before it places the new items to satisfy the heap condition. If you wish to simply clear the current heap, you can enter `0` for the number of items to empty the heap.

The Erase and Random Fill Button

The Erase and Random Fill button performs a similar operation to that of the Make Random Heap button, except that it *only fills the array* with a randomly ordered collection of items. Because the heap condition is almost never satisfied after inserting items with random keys, the number of items in the heap is set to one (or zero if you choose an empty heap). A single-item array always satisfies the heap condition because there are no child items with which to compare its key.

FIGURE 13-9 A random set of items in the array used for the heap

The Visualization tool shows the random items in the array with the `nItems` pointer at cell 1, as shown in Figure 13-9. With only one item satisfying the heap condition, only the

root node is shown in the tree. If you insert a new item in this condition, the new item will go in the cell indicated by the `nItems` pointer, overwriting whatever item had been stored there earlier. A separate operation can be used to organize the array items beyond the `nItems` pointer into a proper heap.

The Peek Button

Just as with stacks and queues, many algorithms need to see what the next item will be before they decide to remove it. The Peek button performs this common task, copying the key and data from the root node, the one with the maximum key, to return as an output.

The Remove Max Button

Selecting the Remove Max button takes the maximum item out of the heap, replaces it with the last item, and sifts that item down from the root of the tree. The removed key and data are returned as an output.

The Heapify Button

The Heapify button takes whatever data is stored in the array beyond the `nItems` pointer and rearranges it to form a heap. We study this operation in the "Heapsort" section later.

The Traverse Button

Occasionally, it's useful to perform some operation on all the items in a heap, such as collecting statistics about the current collection. The Traverse button provides an example by printing the keys of all the items. We explore the issue of traversal order later in this chapter.

Python Code for Heaps

As mentioned previously, heaps are almost always implemented as arrays. The reason is that the corresponding binary tree is complete and always fills the first N cells of the array, where N is the number of items in the heap. We use Python's `list` (array) to store the items in this implementation.

Listing 13-2 shows the beginning of the definition of the `Heap` class. As with other array-based structures, the constructor takes an initial `size` parameter to determine the size of the array to allocate. An internal `_nItems` field is set to 0 to indicate no items are stored in the private `_arr` yet. The constructor also takes a `key` parameter, which defines the function to extract the key used in ordering records or other complex structures stored in the array.

The first four methods are like those for stacks and queues. The `isEmpty()` method tests whether the heap has any items stored in it. The `isFull()` method checks whether the number of items fills the entire array that was allocated. The `__len__()` method allows

callers to know the number of items in the heap using Python's built-in `len()` function. Finally, the `peek()` method returns the maximum-keyed item in the heap if it is not empty.

LISTING 13-2 The Core `Heap` Class

```
def identity(x): return x   # Identity function

class Heap(object):
   def __init__(self, key=identity, size=2): # Heap constructor
      self._arr = [None] * size # Heap stored as a list/array
      self._nItems = 0      # No items in initial heap
      self._key = key       # Function to get key from heap item

   def isEmpty(self): return self._nItems == 0 # Test for empty heap

   def isFull(self):  return self._nItems == len(self._arr)

   def __len__(self): return self._nItems  # Number of items

   def peek(self):          # Return item with maximum key
      return None if self.isEmpty() else self._arr[0]
```

The next methods define the correspondence between the array and tree representations of a heap. Traversing a tree when sifting up or sifting down means following parent-child relationships, which are not explicit in the array. As you saw in Chapter 8, the node at index `i` in the array has its

- Parent at `(i - 1) // 2`,
- Left child at `i * 2 + 1`, and
- Right child at `i * 2 + 2`.

Those relationships can be seen in Figure 13-2. The formulas may return indices that are outside of the array bounds, `[0, _nItems)`, in which case there is either no parent or no child for that node. The methods in Listing 13-3 translate cell indices into those of their neighboring nodes but do not test whether there is an item there.

NOTE

Remember that Python's `//` operator performs integer division, in which the answer is rounded to the lowest integer. This is true even for negative numbers:

`-1 // 2 == -1`.

LISTING 13-3 Parent and Child Relationship Methods

```
class Heap(object):
…
   def parent(self, i):     # Get index of parent in heap tree
      return (i - 1) // 2   # Item i's parent index is half of i - 1

   def leftChild(self, i):  # Get index of left child in heap tree
      return i * 2 + 1      # Item i's left child is at twice i plus 1

   def rightChild(self, i): # Get index of right child in heap tree
      return i * 2 + 2      # Item i's right child -> twice i plus 2
```

Insertion

The `insert()` method is fairly short; the bulk of the work is handled by the sift up algorithm in a separate method, as shown in Listing 13-4.

If the internal array is full, the `insert()` method calls the private `_growHeap()` method to increase the size of the array. Let's look at growing the heap before the rest of the insert operation. Like the hash tables in Chapter 11, "Hash Tables," growing the heap doubles the size of the array. First, it saves the `current` array in a temporary variable, allocates a new array that's twice as large (including at least one cell), and then copies all the current items into it. Unlike the hash tables, there is no need to reinsert (rehash) the items in the heap; they can be copied to the new array in the same order. The Visualization tool animates the `_growHeap()` process, and seeing it in operation can help clarify the details.

After the array size has been checked and perhaps enlarged, the `insert()` method places the new item in the empty cell indexed by `_nItems`. After incrementing `_nItems` by one, it can now call `_siftUp()` on the new item at the end of the array (bottom of the tree).

The `_siftUp()` method starts at a particular index in the array, `i`, and works up the parent links to the root. If called on the root (or a negative index), it does nothing because the item cannot move upward in the tree. Otherwise, it prepares to move the `item` at `i` by storing it and its key in temporary variables. This creates the first "hole" at cell `i`.

The `while` loop "walks" up the parent links. When `i` gets to 0, it has reached the root, and no more moves are possible. In the loop body, `_siftUp()` finds the parent node and compares its key to that of the item being sifted up. If the parent's key is less than the `itemkey`, it moves the parent item down into cell `i`. That means the "hole" moves up, and it changes `i` to point at the parent cell. If the parent's key is greater than or equal to the `itemkey`, then the final position for the item has been found and the loop terminates.

The final step is moving the `item` to the "hole," where it either has a parent with a larger key or no parent node. The hole is now filled, and the heap condition is restored.

LISTING 13-4 The `insert()` Method for Heaps

```
class Heap(object):
…
   def insert(self, item):  # Insert a new item in a heap
      if self.isFull():     # If insertion would go beyond array
         self._growHeap()   # then expand heap array
      self._arr[self._nItems] = item # Store item at end of array
      self._nItems += 1     # Increase item count
      self._siftUp(self._nItems - 1) # Sift last item up

   def _growHeap(self):     # Grow the array for the heap
      current = self._arr   # Store the current array
      self._arr = [None] * max(1, 2 * len(self._arr)) # Double array
      for i in range(self._nItems): # Loop over all current items &
         self._arr[i] = current[i]  # copy them to the new array

   def _siftUp(self, i):    # Sift item i up toward root to preserve
      if i <= 0:            # heap condition. The root node, i = 0,
         return             # cannot go higher, so done.
      item = self._arr[i]   # Store item at cell i
      itemkey = self._key(item) # and its key
      while 0 < i:          # While i is below the root
         parent = self.parent(i) # Get the index of its parent node
         if (self._key(self._arr[parent]) < # If parent's key is
             itemkey):      # less than that of item i,
            self._arr[i] = self._arr[parent] # copy parent to i
            i = parent      # and continue up tree
         else:              # If parent's key is greater or equal,
            break           # then we have found where item i belongs
      self._arr[i] = item   # Move item i into final position
```

Removal

The removal algorithm is also not complicated. As with insertion, the method has two parts: 1) the initial work of removing the maximum keyed item and filling that hole, and 2) sifting down the new root item. Listing 13-5 shows the code. The first step is to raise an exception if the heap is empty. Next, it copies the item in the root node to a temporary variable. This original `root` is returned after reducing the heap.

The `remove()` method copies the last item in the heap—the one at `_nItems - 1`—to the root node. It decrements `_nItems` first because the last item in the array was just moved to index 0. After clearing the last cell for the garbage collector, it calls `_siftDown()` to move the new root item down the heap to where it belongs. When the sift down finishes, it can return the item removed from the root node earlier.

The `_siftDown()` method needs to know which nodes are leaf nodes. Because the tree is complete, this index can be calculated from the number of items in the heap. The

`firstleaf` index always occurs at index `_nItems // 2`. You can see examples of these index relationships in Figure 13-4. The first tree has five items, and the first leaf is at index 2. The second tree (the leftmost one in the second row) has six items, and its first leaf is at index 3. All nodes with an index equal to or larger than `firstleaf` are leaf nodes. If index `i` for the item to be sifted down is in that range, the item cannot be moved down because it has no child nodes, so the `_siftDown()` method returns immediately.

LISTING 13-5 The `remove()` Method for Heaps

```
class Heap(object):
…
   def remove(self):        # Remove top item of heap and return it
      if self.isEmpty():    # It's an error if the heap is empty
         raise Exception("Heap underflow")
      root = self._arr[0]   # Store the top item
      self._nItems -= 1     # Decrease item count
      self._arr[0] = self._arr[self._nItems] # Move last to root
      self._arr[self._nItems] = None # Clear for garbage collection
      self._siftDown(0)     # Move last item down into position
      return root           # Return top item

   def _siftDown(self, i):  # Sift item i down to preserve heap cond.
      firstleaf = len(self) // 2 # Get index of first leaf
      if i >= firstleaf:    # If item i is at or below leaf level,
         return             # it cannot be moved down
      item = self._arr[i]   # Store item at cell i
      itemkey = self._key(item) # and its key
      while i < firstleaf:  # While i above leaf level, find children
         left, right = self.leftChild(i), self.rightChild(i)
         maxi = left        # Assume left child has larger key
         if (right < len(self) and # If both children are present, and
             self._key(self._arr[left]) < # left child has smaller
             self._key(self._arr[right])): # key
            maxi = right    # then use right child
         if (itemkey <      # If item i's key is less
             self._key(self._arr[maxi])): # than max child's key,
            self._arr[i] = self._arr[maxi] # then move max child up
            i = maxi
         else:              # If item i's key is greater than or equal
            break           # to larger child, then found position
      self._arr[i] = item   # Move item to its final position
```

When `_siftDown()` is called on an internal (nonleaf) node, it stores the `item` at `i` and its key in temporary variables. Then it begins a loop that will descend the heap tree until it reaches a leaf node or finds a node where the stored item can be reinserted to preserve the heap condition.

Inside the loop, it gets the indices of node `i`'s children. If there is only a left child, it must have the maximum key; otherwise, the left and right child keys must be compared to determine the index of the child node with the maximum key, `maxi`. The test starts by assuming `maxi` is the left child. If the right child exists and has a larger key, `maxi` is set to the right child.

Now the key of the item being sifted down can be compared with the maximum child key. If the `itemkey` is smaller than the maximum child key, the maximum child is moved up. That creates a "hole" at the maximum child node, and `i` is updated to point there for the next loop. If the `itemkey` equals or exceeds the maximum child key, index `i` points to where the item belongs, and the loop terminates by using `break`.

The final step of `_siftDown()` moves the `item` to sift down into cell `i`. That fills the last "hole" and restores the heap condition.

Traversal

Traversing the items in a heap is somewhat common. This operation might be used, for example, to list all the processes waiting to run in a priority queue or to visit each item and collect statistics. Because the heap only partially orders the items, it is not easy to traverse the items in the order of their key values. Instead, the simplest traversal order is the same as traversing the underlying array.

Listing 13-6 shows the `traverse()` generator used to step through all the items. The completeness of the heap tree means it can simply step through every index for active items in the array. That's done with a `for` loop over the range of indices, yielding each array cell content back to the caller.

LISTING 13-6 The `traverse()` Generator and `print()` Method for Heaps

```
class Heap(object):
…
   def traverse(self):      # Generator to step through all heap items
      for i in range(len(self)): # Get each current item index
         yield self._arr[i] # and yield the item at the index

   def print(               # Print heap tree with root on left
         self, indentBy=2,  # indenting by a few spaces for each level
         indent='', i=0):   # starting with indent at node i
      if i >= len(self):    # If item i is not in tree
         return             # don't print it
      next = indent + ' ' * indentBy
      self.print(indentBy,  # Print right subtree of i at next indent
                 next, self.rightChild(i))
      print(indent, self._arr[i]) # Print item i after indent, then
      self.print(indentBy,  # Print left subtree of i at next indent
                 next, self.leftChild(i))
```

Although this simple traversal order doesn't follow the key ordering, it does follow a particular order for the heap's binary tree. The order is called **breadth first** because it works across the broad levels of the tree visiting all the nodes at a particular level in the tree before going to the next lower level. The nodes are visited from shallowest to deepest, and the breadth of each level is traversed in order, left to right. Try traversing a medium- or large-sized heap (15 or more items) using the Visualization tool to see this. This ordering is very useful for some tree algorithms.

Printing the heap tree is often useful when developing the code. The `print()` method in Listing 13-6 prints the tree on its "side," as was done with trees in Chapters 8, 9, and 10. The method is recursive and uses a reverse form of in-order traversal, where the right subtree of a node is printed, then the node itself, and then its left subtree. The indentation increases for each subtree.

Efficiency of Heap Operations

For a heap with a substantial number of items, the sift up and sift down algorithms are the most time-consuming part of the operations you've seen. These algorithms spend time in a loop, repeatedly moving nodes up or down along a path. The maximum number of copies necessary is bounded by the height of the heap; if there are five levels, four copies carry the "hole" from the top to the bottom. (We ignore the two moves used to transfer the end node to and from temporary storage; they're always necessary, so they require constant time.)

The `_siftUp()` method has up to four operations in its loop: (1) calculate the parent index of the current "hole," (2) compare the key of the node to insert with that of the parent, (3) copy the parent down to the "hole," and (4) move the "hole" up. The third and fourth operations are skipped on the final iteration of the loop. The `_siftDown()` method has up to six operations in its loop: (1) calculate the left and right child indices of the current "hole," (2) compare the keys of the left and right child items, (3) assign the largest child index, (4) compare the largest child key with that of the item sifting down, (5) copy the largest child item into the "hole," and (6) move the "hole" down. The fifth and sixth operations are skipped on the final iteration of the loop. Both methods must move the sifted node into the final "hole" after the loop exits.

A heap is a special kind of binary tree, and as you saw in Chapter 8, the number of levels L in a binary tree equals $\log_2(N+1)$, where N is the number of nodes. The `_siftUp()` and `_siftDown()` routines cycle through their loops L – 1 times, so the first takes time proportional to $\log_2 N$, and the second somewhat more because of the extra comparison and assignment. Thus, the insert and remove operations both operate in O(log N) time.

For insertion, you must also consider the time to grow the heap array up to the size needed to hold all the heap items. The analysis of the time needed to grow the array follows that of growing the array for hash tables. Remember that doubling the hash array

when the load factor was above a threshold allowed a hash table for N items to be built in O(N) time. In the case of heaps, the insert operation is O(log N) instead of O(1). That means inserting N items into a heap takes O(N × log N) time.

Traversing a heap of N items requires O(N) time. There are no empty cells in the array that must be visited, like in the hash table, so traversing a heap is a little faster than a hash table.

A Tree-Based Heap

In the figures of this chapter, we've shown heaps as if they were trees because it's easier to visualize them that way, but the implementation is array-based. It's possible, of course, to use an actual tree-based implementation. The tree will be a binary tree, but it won't be a search tree because, as you've seen, the ordering principle is not as strong. It will be a complete tree, with no missing nodes. Let's call such a tree a *tree-based heap*.

One problem with tree-based heaps is finding the last node. You need to find this node to remove the maximum item because it's the node that's inserted in place of the deleted root (and then sifted down). You also need to find the first "empty node"—the node just after the last node"—because that's where the insert method needs to place the new item before sifting it up. In general, you can't search for these nodes because you don't know their values, and anyway it's not a search tree. You could add some fields to a tree-based heap data structure that maintain pointers to the end, just like for double-ended lists, but there's another way that doesn't require the additional fields.

As you saw in the discussion of the Huffman tree in Chapter 8, you can represent the path from root to leaf as a binary number, with the binary digits indicating the path from each parent to its child: 0 for left and 1 for right. It turns out there's a simple relationship between the number of nodes in the tree and the binary number that codes the path to the last node. Assume the root is numbered 1; the next row has nodes 2 and 3; the third row has nodes 4, 5, 6, and 7; and so on. This is, of course, one more than the array index used in the array-based implementation.

To find the last filled node (or the first empty node), convert the number of nodes (or one more than the number) to binary. For example, say there are 29 nodes in the tree and you want to find the last node. The number 29 (decimal) is 11101 in binary. Remove the initial 1, leaving 1101. This is the path from the root to node 29: right, right, left, right. Each 1 in the binary representation means take the right child, and each 0 means the left. The first available null node will be found by following the path to 30, which (after removing the initial 1) is 1110 binary: right, right, right, left. Figure 13-10 shows the path through the tree.

You can see the binary representation of an integer in Python using the string format tool. For example, `'{:b}'.format(29)` evaluates to `'11101'`. It's also easy to repeatedly use the % operator to find the remainder (0 or 1) when the number `n` is divided by 2 and then use the `//` or `>>` operators to divide `n` by 2 and drop the last bit. When `n` is less than 1, you're done. The sequence of remainders, which you can save in an array, is the binary number. (Be careful which order you interpret them!)

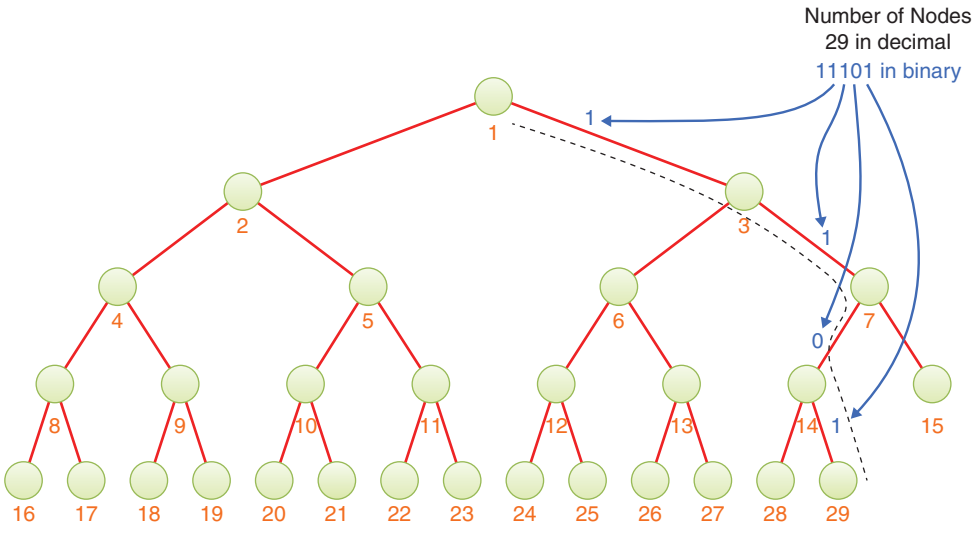


FIGURE 13-10 Finding the path to the last node in a 29-node tree.

Using the binary representation allows a tree-based heap to find the path to the last leaf or next empty leaf, but it doesn't speed up reaching that node. The algorithm still must follow all the references from the root to the node, taking O(log N) steps. For an array-based heap, that can be done in constant time, O(1).

After the appropriate node (or null child) is found, the heap operations are straightforward. When you're sifting up or down, the structure of the tree doesn't change, so you don't need to change linkages to move the actual nodes around. You can simply copy the data items from one node to the next, like was done in the array-based implementation. This way, you don't need to connect and disconnect all the children and parents for a simple move. The `Node` class, however, needs a field for the parent node in addition to the child nodes because you need to access the parent when you sift up. This is somewhat like the doubly linked lists you saw in Chapter 5, "Linked Lists." The root of the tree should also keep an `nItem` field to facilitate finding the path to the last leaf and first empty node.

In the tree heap, insertion and removal operations take O(log N) time. As in the array-based heap, the time is mostly spent doing the sift up and sift down operations, which take time proportional to the height of the tree. Traversal of a tree-based heap still takes O(N) time, but the traversal order can be any of the orders you saw for binary trees: pre-order, in-order, or post-order. While the name might be "in-order," the traversal is still not in the order of the item keys. The flexibility to perform those three orderings comes from the explicit child references in the structure. It's possible to do a breadth-first traversal by converting node numbers to paths from the root, and then following those paths to the node, but that would take significantly more than O(N) time.

Heapsort

The efficiency of the heap data structure lends itself to a surprisingly simple and very efficient sorting algorithm called a **heapsort**.

The basic idea is to insert all the unordered items from a source data structure into a heap using the normal `insert()` method. Repeated application of the `remove()` method then removes the items in sorted order. Here's how that might look to sort a Python sequence into a result array:

```
theHeap = Heap(size=len(aSequence)) # Create an empty heap
for item in aSequence:              # Loop over unsorted sequence
   theHeap.insert(item)             # Copy items to heap
result = []                         # Make a result array
while not theHeap.isEmpty():        # Loop over array indices
   result.append(theHeap.remove())  # Copy items back to array
```

This code would put the items in the result array in descending order because the first item removed from the heap is the maximum. To put the items in ascending order, you could construct the result array to have enough cells and then index it in reverse order.

Because `insert()` and `remove()` operate in O(log N) time, and each must be applied N times, the entire sort requires O(N × log N) time, which is the same as the quicksort. It's not quite as fast as quicksort overall, however, partly because there are more operations in the inner `while` loop in `_siftDown()` than in the inner loop in quicksort.

With a little cleverness, we can enhance this basic algorithm to make heapsort more efficient. The first enhancement saves time, and the second saves memory.

Sifting Down Instead of Up

If you insert N new items into a heap, you apply the `_siftUp()` method N times. You can take advantage, however, of a similar technique to what you did when removing items: sifting an out-of-sequence item down from the root into a correctly arranged heap. What's more, doing this only needs N/2 calls to `_siftDown()`, because you need to do it only for internal (nonleaf) nodes. This approach offers a small speed advantage, even though sifting down is slightly more time-consuming.

Two Correct Subheaps Make a Correct Heap

To see how this approach works, we need to look at smaller parts of the heaps: **subheaps**. What do we mean by subheap? They are just like subtrees inside of trees. Any node in a heap tree can be thought of as the root of a subheap. Figure 13-11 shows three subheaps outlined with dotted triangles.

Remember that the `remove()` algorithm extracts the root item, replaces it with the last item in the heap, decrements the item count, and sifts the root item down. In that case, an out-of-order item was placed in node 0, and both its subheaps were correctly ordered.

After it was sifted down, the heap condition was restored. That property generalizes to work at any node in the heap tree. To be more specific, if you apply `_siftDown()` to node J and both J's left and right subheaps satisfy the heap condition, then the subheap rooted at J will also satisfy the heap condition after the sift down operation completes. This property holds whether or not the item at node J satisfied the heap condition before sifting down.

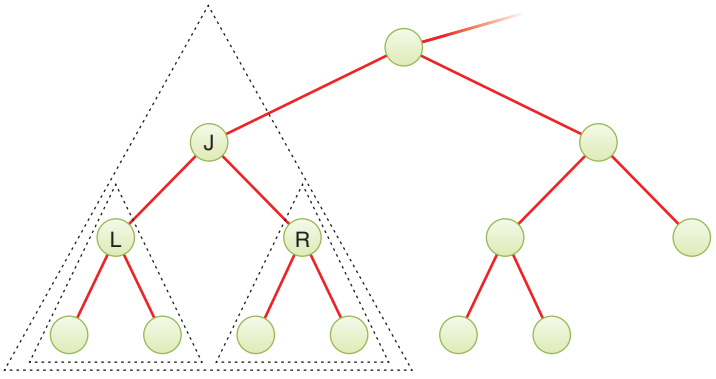


FIGURE 13-11 Sifting down into two correct subheaps makes a correct heap

In Figure 13-11, if an out-of-order item is placed in node J, the subheap rooted at node J does not meet the heap condition. If you then sift down that item, subheap J will correctly satisfy the heap condition if both its child subheaps—rooted at nodes L and R—are correct. When subheap J is correct, you can then consider what to do with the subheap at its parent, which may not be correct.

This example suggests a way to transform an unordered array into a heap. You can apply `_siftDown()` to the nodes on the bottom of the (potential) heap—that is, at the end of the array—and work your way upward to the root at index 0. At each step, the subheaps below you will already be correct heaps because you already applied `_siftDown()` to them. When you apply it to the root, the unordered array will have been transformed into a heap.

Notice also, that the nodes on the bottom row—those with no children—are already correct heaps because they are trees with only one node; they have no relationships that can be out of order. Therefore, you don't need to apply `_siftDown()` to these nodes. You can start at node N/2 – 1, the rightmost node with children, instead of N – 1, the last node. Thus, you need only half as many sift operations as you would using `insert()` N times. Figure 13-12 shows that in a 13-node heap, sifting down starts at node 5, then node 4, and so on until it reaches the root at 0.

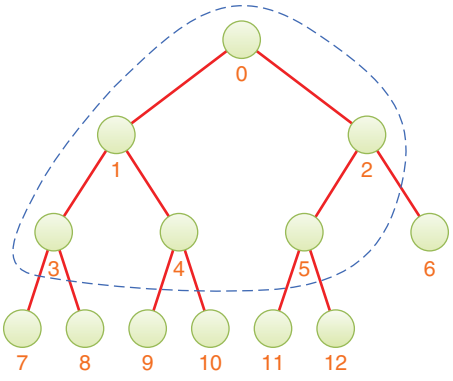


FIGURE 13-12 The `_siftDown()` method is applied to the internal nodes

In Python, the first internal node of an N node heap has index `N // 2 - 1`. So, to apply `_siftDown()` to the internal nodes, working back to the root, you could run

```
for j in range(N // 2 - 1, -1, -1):

   theHeap._siftDown(j);
```

Using the Same Array

The initial code fragment for the heapsort showed unordered data in a sequence. This data was then inserted into a heap and finally removed from the heap and written back to an array in sorted, descending order. In this procedure, three size-N structures are required: the initial sequence, the array used by the heap, and the result array.

In fact, the same array can be used for all three: the input, the heap, and the result array. Reusing the array cuts the amount of memory needed for heapsort by two-thirds; no memory beyond the initial array is necessary. In other words, you can create the heap *in place*. This approach only works, of course, if the source data is in an array to start. To use the heapsort algorithm on data stored in a hash table or linked list, the items would need to be inserted into a heap and then removed (or the underlying array returned as the result).

You've already seen how `_siftDown()` can be applied to half the elements of an array to transform them into a correct heap. If you transform the unordered array data into a heap in place, only one array is necessary for this task. Thus, the first step in heapsort requires only one array.

What about the second step when you apply `remove()` repeatedly to the heap? Where are you going to put the items that are removed?

Each time an item is removed from the heap, an element at the end of the heap array becomes empty because the heap shrinks by one. You can put the removed item in this newly freed cell. As more items are removed, the heap array becomes smaller and smaller, while the array of ordered data becomes larger and larger. In other words, part of the array

is holding the heap, and part is holding the sorted output. With a little planning, it's possible for the ordered array and the heap array to share the same space.

The fact that all the work can be done in one array means that you don't need a separate heap object to hold the intermediate results. You can simply operate on the input array and **heapify** it. The array starts off unsorted. All N items are part of the unsorted section. When you construct the heap, the leaf nodes—the second half the array—are automatically part of the heap. The first half of the array contains the unsorted section. As you sift items down, the unordered section of the array shrinks as the heap grows. When the array is fully heapified, then you can start removing the maximum item from index 0 and placing it at the end of the array. That way, you avoid disrupting the heap because the heap has shrunk by one.

The Heap Visualization tool can heapify an array of data items by their keys. To see an example, use the Erase and Random Fill button to erase any existing data and fill the array with a few dozen items. While the array is full, the heap tree shows only the root node, as in Figure 13-9, because the random arrangement of data is unlikely to satisfy the heap condition.

Start the *heapification* by selecting the Heapify button. Figure 13-13 shows a snapshot during the processing of a 30-item array. After creating the 15 leaf nodes, the algorithm steps backward through the internal nodes just above the leaves. In the figure, six of the internal nodes have been made into subheaps, with five of them satisfying the heap condition locally. The arrow labeled `j` indicates that it is about to sift down item 37 to make that subheap satisfy the heap condition. After it finishes making all the 3-node (and one 2-node) subheaps, it goes to the next higher level and joins them into 7-node (and one 6-node) subheaps. Note that the `j` arrow points at item 37 in both the array on the left and the partial tree on the right, indicating the boundary between the array cells that have been heapified so far.

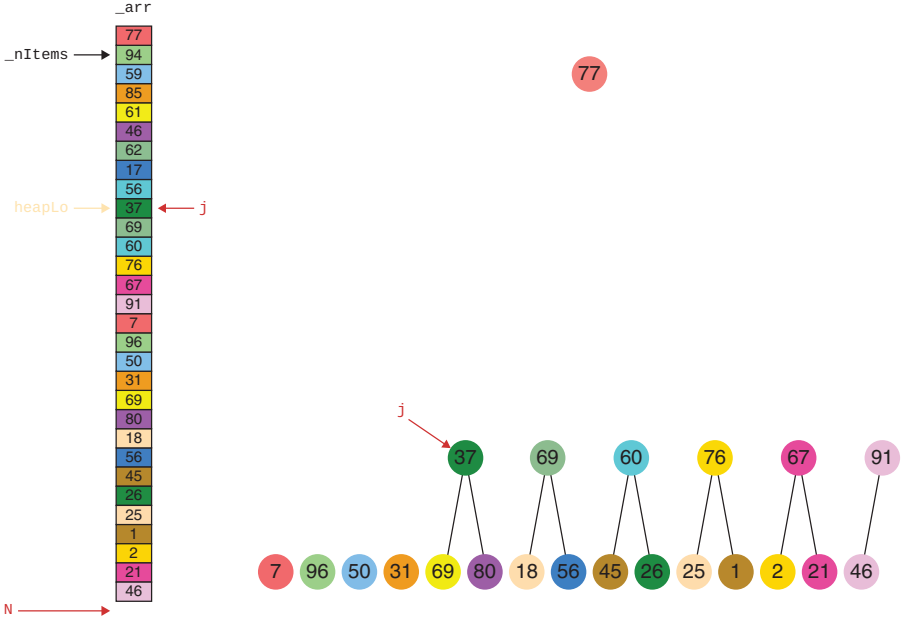


FIGURE 13-13 Heapifying an array of 30 random items in the Visualization tool

With the data now in the form of a heap, the heapsort can begin producing the final, sorted ordering. The Visualization tool doesn't show the sorting process, so let's follow the full process in a small example. Figure 13-14 shows the sorting of a six-item array in place using heapsort. The first row shows the heapification process, and the second row shows the construction of the sorted array.

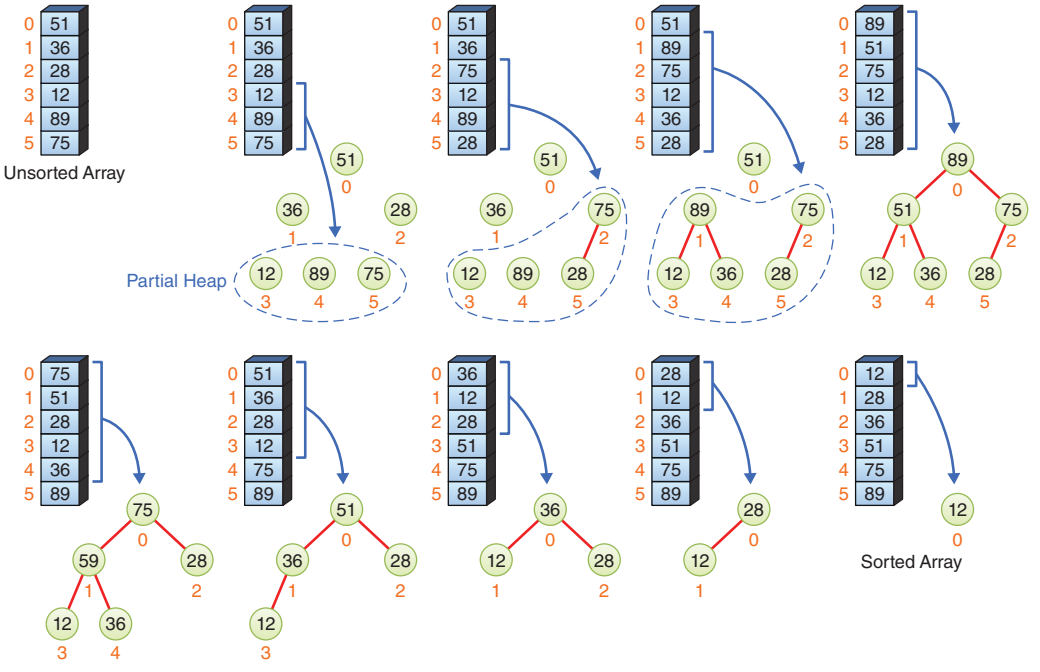


FIGURE 13-14 Heapsort in place

The array in the upper left of the figure is the original, unsorted array. The next step to the right shows the starting state in heapifying the array. The last three items of the array—indices 3 through 5—are treated as leaf nodes in the partial heap (shown by the dashed line encircling those nodes). You could also call these leaves "single node sub-heaps." Note that not one of the items in the array has changed; a range of those nodes/cells has simply been marked as the heap area.

The third step shows what happens after calling `siftDown()` on the node at index 2. The original item in node 2 has key 28. That gets sifted down to node 5 and item 75 moves up. The heap now has four items—indices 2 through 5. The figure shows the other items in nodes 0 and 1, but they are not part of the heap. (The Visualization tool shows a node at the root of the heap tree during heapification, but it is not connected to the other nodes and is not part of the collection of verified subheaps.)

The fourth step shows the outcome of sifting down item 36 from node 1. It swaps places with item 89 in node 4, and the heap now has five items. The red links show that the heap condition has been confirmed for the connected nodes.

In the fifth step, the final item at node 0, item 51, is sifted down. It swaps places with item 89 but doesn't need to sift down any further to the second level. Now that the six-item array is fully heapified; it's a complete, correct heap. That is shown at the right end of the top row of Figure 13-14.

The heapsort algorithm then removes items from the top of the heap and places them at the end of the array. Removing the maximum item to place at the end of the array effectively swaps the root item with the last item in the heap. In the example in the figure, item 89 at the root is swapped with item 28 at node 5. Then item 28 sifts down into node 2 as shown in the sixth step (first step of the second row) of Figure 13-14. Item 89 sits at the end of the array, which is outside of the heap range of 0 through 4.

The seventh step takes the maximum item at the root, item 75, swaps it with the last item of the heap, item 36, and sifts item 36 down. This time the item goes down the left branch to node 1. Item 51 moves up to the root. The extracted maximum, 75, ends up in node 4. Now the four-item heap spans indices 0 through 3, and indices 4 and 5 hold the partial result.

The eighth and ninth steps are similar. They swap the root and last items of the heap, and then sift down from the root. Each time, the maximum item from the root is placed just before the other sorted items from previous steps. The heap shrinks and ends up with just one element in the heap. At this point, the array is fully sorted. The one-item heap contains the item with the minimum key. There's no need to swap it with the last heap item, itself, and do any sifting.

Going through the details shows the very efficient use of the array. The heap starts as the second half of the array, and heapify grows it to include all the items up to the root. Then the heap shrinks while occupying the first part of the array as items are removed. Let's look at the code to do this.

The heapsort() Subroutine

Here, we combine the two techniques—converting the array into a heap ("heapifying" it) and then placing removed items at the end of the same array—together in a subroutine that performs heapsort on an input array.

The `heapsort()` subroutine shown in Listing 13-7 takes a Python array (`list`) and sorts its items by their keys as extracted by the `key` function, which defaults to the `identity()` function. The first statement sets a local variable, `heapHi`, to manage the range of indices where the heap is stored. We use the same mapping of array indices to binary tree nodes as in the `Heap` class. It calls the `heapify()` routine to organize the items into a heap. Let's first discuss `heapify()` and then return to `heapsort()`.

The `heapify()` subroutine determines the amount of the data within the array, `N`, in case the caller did not provide it. The lower bound of the heap, `heapLo`, is set to `N // 2` because all the leaf nodes are correct, single-item subheaps. Thus, the heap covers the cells from `heapLo` to `N - 1`.

LISTING 13-7 The `heapsort()`, `heapify()`, and `siftDown()` Subroutines

```
def heapsort(array,              # Sort an array in-place by keys extracted
             key=identity):      # from each item using the key function
   heapHi = len(array)           # Make entire array from 0 to heapHi
   heapify(array, heapHi, key)   # into a heap using heapify
   while heapHi > 1:             # While heap has more than 1 item
      heapHi -= 1                # Decrement heap's higher boundary & swap
      array[0], array[heapHi] = array[heapHi], array[0] # max and last
      siftDown(array, 0, heapHi, key) # & sift down item moved to top

def heapify(array,               # Organize an array of N items to satisfy
            N=None,              # the heap condition using keys extracted
            key=identity):       # from the items by the key function
   if N is None:                 # If N is not supplied,
      N = len(array)             # then use number of items in array
   heapLo = N // 2               # The heap lies in the range [heapLo, N)
   while heapLo > 0:             # Heapify until the entire array is a heap
      heapLo -= 1                # Decrement heap's lower boundary
      siftDown(array, heapLo, N, key) # Sift down item at heapLo

def siftDown(array,              # Sift item down in heap starting from
             j,                  # node j
             N=None,             # down to but not including node N
             key=identity):      # using key function to extract item's key
   if N is None:                 # If N is not supplied,
      N = len(array)             # then use number of items in array
   firstleaf = N // 2            # Get index of first leaf in heap
   if j >= firstleaf:            # If item j is at or below leaf level,
      return                     # it cannot be moved down
   item = array[j]               # Store item at cell j
   itemkey = key(item)           # and its key
   while j < firstleaf:          # While j above leaf level, find children
      left, right = j + j + 1, j + j + 2 # Get indices of children
      maxi = left                # Assume left child has larger key
      if (right < N and          # If both children are present, and
          key(array[left]) <     # left child has smaller
          key(array[right])):    # key
         maxi = right            # then use right child
      if (itemkey <              # If item j's key is less
          key(array[maxi])):     # than max child's key,
         array[j] = array[maxi] # then move max child up
         j = maxi                # and continue from new "hole"
      else:                      # If item j's key is greater than or equal
         break                   # to larger child, then found position
   array[j] = item               # Move item to its final position
```

The `while` loop of `heapify()` expands the heap by reducing `heapLo` to incorporate all the nonleaf nodes. It starts with the node immediately before the heap, at `heapLo - 1`. That's the rightmost internal node at the lowest level of the tree. The item there is sifted down by calling the `siftDown()` routine on the array and passing the heap bounds, `heapLo` to `N`. This loop continues until `heapLo` reaches 0, which means all the internal nodes have been sifted into the heap.

When `heapify()` is done, control returns to `heapsort()`. The `while` loop there removes items from the heap in descending order by key. Like the way the `Heap.remove()` method in Listing 13-5 operates, the number of items in the heap is reduced by decrementing `heapHi`. Instead of copying the maximum item to a temporary variable, however, it is swapped with the last item of the heap. That puts the maximum item at the end of the array, in its final position for the result. The swap also moves the last leaf item to the root, and the next call to the `siftDown()` routine sifts it down into the remaining heap.

The loop stops when `heapHi` is 1 because a single-item heap needs no sifting, and the remaining item must have the smallest key. When the loop finishes, the array's items are sorted in order of increasing key values.

The `siftDown()` routine is nearly the same as the private `Heap._siftDown()` method in Listing 13-5. The difference is that in the `heapsort()` context, it must pass the array and the range of indices where the heap lies. That changes as `heapsort()` runs. The operation only needs to know the index of the node to sift down, where the heap ends, and the `key` function because it does not move up in the heap tree.

If the end of the heap is not provided, the length of the array is used. The index of the first leaf node in the heap is calculated to determine when sifting down terminates. If the item to be sifted down is at or below the first leaf (in other words, has an index at or above `firstleaf`), nothing needs to be done, and the routine returns to the caller.

When node `j` is an internal node, `siftDown()` stores the item and its key in temporary variables. Then it starts a loop to perform the item moves needed to position the `item` in the correct position within its subheaps. It calculates the child node indices and determines the one with the maximum key. Then the `itemkey` is compared with the maximum child key. If the `itemkey` is smaller, the maximum child is moved up and the loop continues downward until a leaf node is reached. At the end of the loop, the `item` is moved into the cell last visited in the loop.

The Efficiency of Heapsort

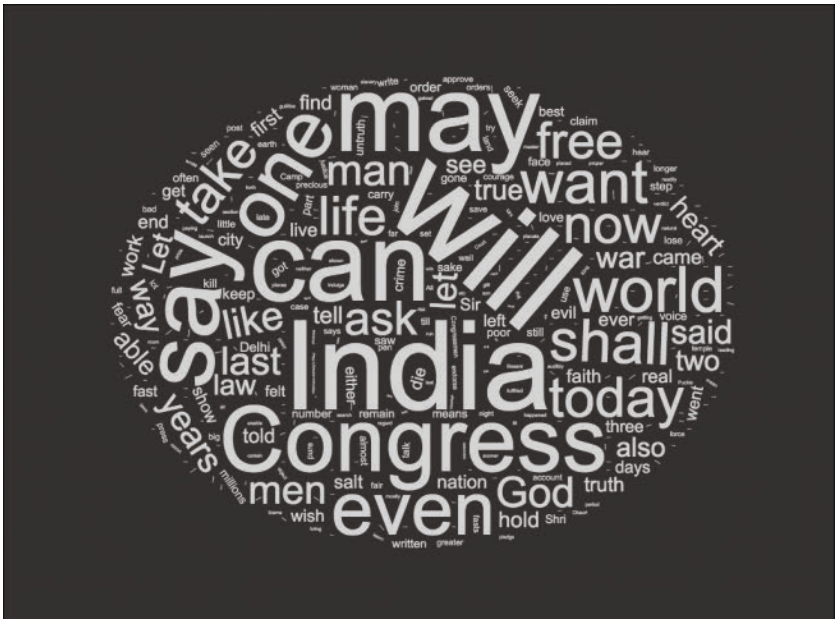
As we noted, heapsort runs in O(N×log N) time. Although it may be slightly slower than quicksort, an advantage over quicksort is that it is less sensitive to the initial distribution of data. Certain arrangements of key values can reduce quicksort to slow $O(N^2)$ time, whereas heapsort runs in O(N×log N) time no matter how the data is distributed. Both sort methods take O(N) memory, and both can sort arrays in place.

What might be somewhat surprising is that the first part of heapsort, the heapification of the array, takes only O(N) time. Extracting the full sorted sequence is what requires the O(N×log N) time. You can take advantage of that capability for other algorithms such as in statistics.

Order Statistics

Heaps have another special application: calculating **order statistics**. When you analyze large quantities of data, a variety of statistical measures provide information about the overall distribution of data. When the data can be ordered—from smallest to largest, darkest to brightest, earliest to latest, and so on—you typically want to know the minimum and maximum values. They are easy to compute by going through all the values once and updating variables that store the minimum and maximum values found.

Word clouds are a common example of the use of order statistics. In word (or text) clouds, the frequency of words in a collection such as books, chat messages, essays, or speeches is used to determine the size of the words in a graphic. Figure 13-15 shows a word cloud derived from several of Mahatma Gandhi's speeches. The largest words were the most common, and the word size indicates how frequently the other terms appeared. (Very common words such as *is*, *be*, *a*, and *to* are not shown. The speech transcripts are from www.mkgandhi.org/speeches/speechMain.htm, and the word cloud was produced by www.wordclouds.com.)


may
will
India
Congress
can
one
say
Let
take
life
want
free
man
now
world
shall
today
even
God
men
years
ask
like
said
two
also
truth
heart
war
came
let
Sir
tell
last
law
told
salt
nation
hold
wish
Shri
Delhi
felt
keep
kill
work
way
able
fast
show
end
live
city
true
see
gone
find
first
order
write
woman
approve
seek
best
claim
face
step
longer
carry
part
precious
Camp
often
get
bad
late
little
left
poor
still
evil
ever
voice
faith
real
three
days
number
remain
means
either
die
almost
written
pledge
greater
millions
crime
sake
case
Part
Pan
Ali
Al
Ali
Ali
Ali

To find these order statistics like the decile, quartile, or median, and perhaps go a step further to not just identify the value but also collect all the records that fall within, say, the highest decile, you could sort the data. With heapsort or quicksort, you can do that in O(N×log N) time. That's fast, but still quite a bit more than the O(N) time it takes to find the minimum, maximum, and linear statistics like the average. Is there a faster way?

Partial Ordering Assists in Finding the Extreme Values

As you've seen, the maximum (or minimum) keyed item goes to the root of a heap. If you want to find the 10 highest keyed items, you can simply remove 10 items from the heap. Removing these items could save quite a bit of work compared with using heapsort to fully sort, say, 10 million records, just to find the 10 highest. So, you need to put the data in the heap efficiently and remove only the desired items.

How much time does it take to make the heap? As you saw in heapsort, if the data is already in an array that can be modified, you can heapify the items by sifting down all the items stored in internal nodes. As shown in Listing 13-7, the `heapify()` subroutine consists of a single loop over the data calling `siftdown()` on each of the internal nodes.

Getting the highest keyed items from the array operates similarly to `heapsort()`, but removes only a fixed number, *K*, of items. The `highest()` function shown in Listing 13-8 starts off by determining the number of items in the array and then heapifying it. A `result` array of *K*-elements is allocated to hold the maximum-keyed items.

LISTING 13-8 Subroutine to Get the K Highest Keyed Items in an Array

```
def highest(K, array,                  # Get the highest K items from an array
            N=None,                    # of N items by heapifying the array based
            key=identity):             # on keys extracted by the key function
   if N is None:                       # If N is not supplied,
      N = len(array)                   # then use number of items in array
   heapify(array, N, key)              # Organize items into a heap
   result = [None] * K                 # Construct an output array
   heapHi = N                          # End of heap starts at last item
   while N - heapHi < K:               # While we have not yet removed K items,
      result[N - heapHi] = array[0]    # Put max from heap in result
      heapHi -= 1                      # Decrement heap's higher boundary & swap
      array[0], array[heapHi] = array[heapHi], array[0] # max and last
      siftDown(array, 0, heapHi, key) # & sift down item moved to top
   return result                       # Return K-item result
```

The `highest()` function uses a `while` loop to remove the K items from the heap. Each removal shrinks the heap, lowering the `heapHi` index by one, and the count of items removed is the difference between the end of the array, `N`, and `heapHi`. That difference is used as the index in the `result` array for storing the output items (in descending order by key). The `heapHi` index is then decremented to effectively remove the maximum-keyed

item from the heap. The first and last items on the heap are swapped, and the new item at the root is sifted down into the correct heap position.

When the `highest()` function returns, the result array contains the K highest-keyed items in descending order, and the input array has been rearranged to satisfy the heap condition in the first N – K cells, followed by K cells of the highest-keyed items in increasing order by key.

The Efficiency of K Highest

How much work is done in computing `highest()`? There are two phases to consider. The first phase is the `heapify()` operation on N items, and the second is the removal of the K highest-keyed items.

Analyzing the complexity of the second phase is straightforward. Removing an item causes one item to sift down through the heap. It's possible that the item sifts down to the leaf level of the heap. The full heap has $\log_2(N+1)$ levels, so the removal phase takes O(K×log N) time. The heap does shrink by one for each removal, but we can assume that N is much bigger than K, and K is 1 or larger, so log(N) doesn't change significantly as K items are removed.

The analysis of the first phase might be a surprise. As mentioned earlier, the `heapify()` operation takes O(N) time, even though it must go through half of the items in the array, sifting them down into the heap. That means the overall complexity of computing `highest()` is O(N + K×log N). When K is much smaller than N, that is quite an improvement over a heapsort that takes O(N×log N).

To understand how `heapify()` takes O(N) time, let's first imagine that the sift down operation does not depend on the depth of the heap. Then each internal node would take O(1) time, so processing all N/2 of them would take O(N) time. You can think of that as a lower bound on how fast `heapify()` computes.

The next step is to look at how much work is done for each internal node. All the leaf nodes and perhaps some internal nodes have been processed to form a group of subheaps, as shown in Figure 13-16. The dashed line surrounds the nodes that have been processed so far, and the red circle surrounds the next item to sift down.

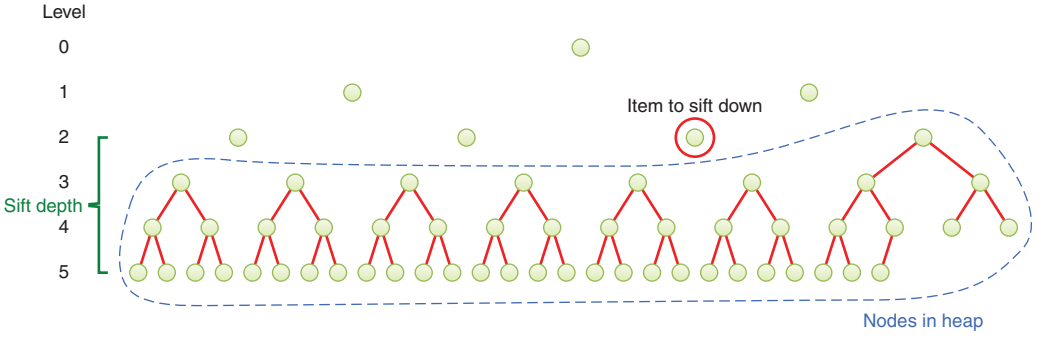


FIGURE 13-16 Sifting down during heapify

The item sifting down lies at a particular level, L, of the overall heap. It can sift down all the way to the leaf level, which lies at $\log_2(N+1)$. The figure shows the difference between L and the leaf level as the *sift depth*. For the leaves, the sift depth is zero. For the parents of the leaves, it's one. Each level higher in the tree adds one until you reach the root, where the sift depth is $\log_2(N+1)$.

The sift depth limits the amount of work that must be done for a particular node. If the sift depth is S, it will take at most $S \times 2$ comparisons and S copies to shift the stored items along the path to the leaf nodes. The reason is that you need one comparison to find the maximum child, one comparison of the maximum child with its parent, and one copy to move the maximum child up (in the worst case).

Now you have a maximum bound on the work at each node. Let's see what the whole tree could require. Assume that you have a complete binary tree where every level is full of nodes. Down at the leaf nodes, the sift depth is zero, so you don't even try to sift them down. Half of the tree's nodes are leaf nodes (technically, (N + 1) / 2). They don't add anything to the total work.

One level up from the leaves, you have S = 1, a sift depth of one. How many nodes are at that level? Exactly half as many as there are leaves, so (N + 1) / 4 need to sift down at most one level. You can start writing down the total work done on these first two levels by multiplying the sift depth with the number of nodes at that depth:

$$total\ work\ in\ first\ two\ levels = 0\ \frac{N+1}{2} + 1\ \frac{N+1}{4}$$

In this case, 1 is the amount of work when S = 1. If you count comparisons and copies equally, then it might actually take 3 units of work. Because that 3 is a constant, and constants are ignored later when looking at the Big O complexity, you leave that constant out.

Moving up a level, you reach S = 2 and the number of nodes is halved again:

$$total\ work\ in\ first\ three\ levels = 0\ \frac{N+1}{2} + 1\ \frac{N+1}{4} + 2\ \frac{N+1}{8}$$

Let's move the common (N + 1) part out of the sum for all the levels:

$$total\ work\ in\ first\ three\ levels = (N+1)\left(0\ \frac{1}{2} + 1\ \frac{1}{4} + 2\ \frac{1}{8}\right)$$

As you go up in sift depth, a pattern emerges. Each term in the sum multiplies the sift depth, S, with a fraction that is a power of ½. When you sum up K of those levels, you get

$$total\ work\ in\ first\ K\ levels = (N+1)\left(0\ \frac{1}{2} + 1\ \frac{1}{4} + 2\ \frac{1}{8} + \ldots + (K-1)\left(\frac{1}{2}\right)^K\right)$$

Notice that most of the internal nodes lie near the bottom of the tree and have a small sift depth (half of them have sift depth 0). The root of the tree has the largest sift depth, but there's only one root node. So, you must process many nodes with short distances to sift down, and fewer nodes with longer distances. In the summation, each term that's added is smaller than the term before it. That means that it might be going down so fast that it doesn't keep expanding the total for more levels.

Let's now write the summation compactly. You sum from sift depth 0 up to the maximum sift depth at the root, $\log_2(N+1)$. You use S to stand for the sift depth. That makes the total work for all levels:

$$total\ work = (N+1)\sum_{S=0}^{\log_2(N+1)} S\left(\frac{1}{2}\right)^{S+1}$$

The first term in the total work is (N + 1). If that fancy summation on the right doesn't grow with N, then you end up with the work taking O(N) time! Is that even possible?

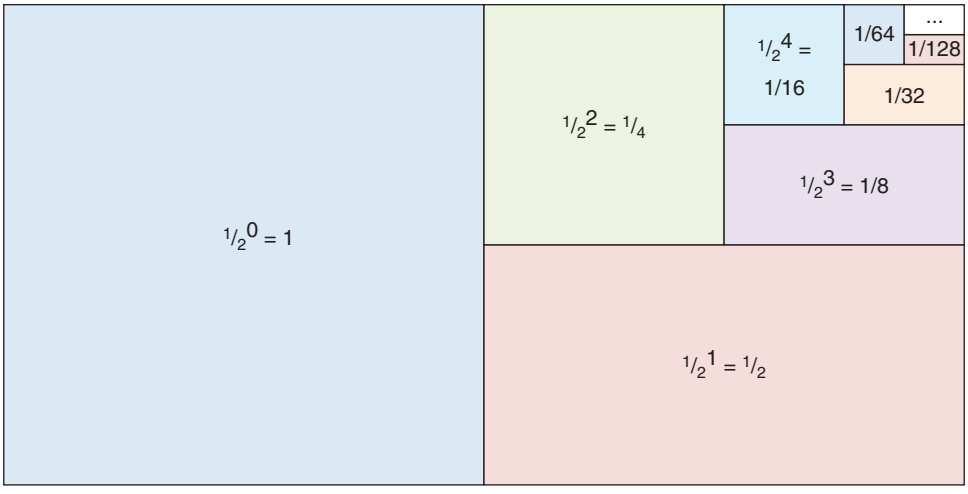
To answer that, we need to bring in some math knowledge of other summations.

$$\sum_{j=0}^{\infty} x^j = \frac{1}{1-x} \qquad when \quad -1 < x < 1$$

$$\sum_{j=0}^{\infty} jx^j = \frac{x}{(1-x)^2} \qquad when \quad -1 < x < 1$$

The first summation is easy to see visually. Figure 13-17 shows the example when *x* is ½. When ½ is raised to the power 0, it is one and is represented by the square on the left. When it's raised to the power 1, it is one-half that size and is represented by the rectangle on the lower right. With each successive power, you divide the last rectangle in half and place it in the space at the upper-right corner.

These ever-shrinking rectangles continue to fill up *half of the remaining space*. They never shrink to zero, but the space remaining continues to shrink. As the power goes to infinity, the total space of all the rectangles sums to 2. How do you know that? If you look at the rectangle that holds all the squares and little rectangles in the figure, it's exactly twice as big as the first square, which contributed exactly 1 unit to the total. Looking at the first formula and substituting ½ for *x*, you get 1 / (1 – ½), which is 2.



$$\sum_{j=0}^{\infty} \left(\frac{1}{2}\right)^{j} = \frac{1}{1-1/2} = \frac{1}{1/2} = 2$$

FIGURE 13-17 Infinite sum of the powers of ½

The second summation formula comes by taking the derivative of both sides of the first equation and then multiplying both by *x*. That's hard to see visually. If you haven't seen it before, it's part of calculus.

You can use the second summation to simplify the total work equation. By summing all the way to infinity, instead of just to $\log_2(N+1)$, you get an upper bound on the work done by `heapify()`.

$$total\ work = (N+1)\sum_{S=0}^{\log 2\,(N+1)} S\left(\frac{1}{2}\right)^{S+1} < (N+1)\sum_{S=0}^{\infty} S\left(\frac{1}{2}\right)^{S+1}$$

The summation over S doesn't quite match the formula with jx^j but you can modify it by factoring out one of the powers of ½:

$$total\ work < \frac{N+1}{2}\sum_{S=0}^{\infty} S\left(\frac{1}{2}\right)^{S}$$

Substituting ½ for *x* in the second summation formula above, you get

$$total\ work < \frac{N+1}{2}\sum_{S=0}^{\infty} S\left(\frac{1}{2}\right)^{S} = \frac{N+1}{2}\frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^{2}} = \frac{N+1}{2}\frac{\frac{1}{2}}{\frac{1}{4}} = N+1$$

So, the upper bound on the total work performed by `heapify()` is just a multiple of N, and that means heapify runs in O(N) time. Proving that took somewhat complicated math but was worth the effort. It means that `highest()` runs in O(N + K×log N) time to find the K highest keyed items. Compared with a `heapsort()` that takes O(N×log N) time, it saves O((N – K)×log N – N) time. The bigger the difference between N and K, the more important it is to use `highest()`.

Summary

- A heap is a fundamental data structure that organizes items in a partial ordering within a binary tree.
- The item with the largest key is stored at the root of the heap tree.
- The heap condition requires that the key of an item at a node of the tree is greater than or equal to those of its child nodes, if any.
- The binary tree for the heap is always complete. All levels of the tree are completely full, and the lowest level can only be missing nodes on the right.
- Priority queues are typically constructed using heaps either as descending-priority queues or ascending-priority queues (where the smallest keyed item is at the root of the heap tree and all children have keys greater than or equal to their parent's).
- Heaps and priority queues offer common methods: inserting a new item, removing the largest item, and peeking at the first (largest) item.
- A heap offers removal of the largest item, and insertion, in O(log N) time.
- Heaps support traversal but do *not* support *ordered* traversal of the data, locating an item with a specific key, nor deletion.
- Heaps are usually implemented as arrays, taking advantage of the completeness of the binary tree to uniquely map array cells to tree nodes.
- The root is at index 0 and the last item at index N – 1.
- The completeness of the tree ensures that all cells between 0 and N – 1 are used.
- Insertion operates by placing an item in the first vacant cell of the array and then sifting it up to its appropriate position.
- When an item is removed from the root, it's replaced by the last item in the array, which is then sifted down to its appropriate position.
- The sift up and sift down processes can be thought of as a sequence of swaps but are more efficiently implemented as a sequence of copies.
- If the heap supports changing the priority of an arbitrary item, it must first locate the node with the item. Next, its key is changed. Then, if the key was increased, the item is sifted up, and if the key was decreased, the item is sifted down.
- A heap can be implemented using nodes and references to form the binary tree (not a search tree). This is called a tree-based heap.

- Algorithms exist to find the last occupied node or the first free node in a tree-based heap using the number of items in the tree.
- Heapsort is an efficient sorting procedure that requires O(N × log N) time.
- Conceptually, heapsort consists of making N insertions into a heap, followed by N removals.
- Heapsort can be made to run faster by applying the sift down algorithm directly to N/2 items in the unsorted array, rather than inserting N items.
- The same array can be used for the initial unordered data, for the heap array, and for the final sorted data. Thus, heapsort requires no extra memory.
- The heapify operation organizes the items in an array to satisfy the heap condition.
- Heapifying an array takes O(N) time.
- Finding the K highest (or lowest) keys among N items in an array can be done efficiently using heapify followed by K removals.
- The complexity of finding these K order statistics is O(N + K×log N).

Questions

These questions are intended as a self-test for readers. Answers may be found in Appendix C.

1. What does the term *complete* mean when applied to binary trees?
   a. All the necessary data items have been inserted.
   b. All the rows are filled with nodes, except possibly the bottom one.
   c. All existing nodes contain data.
   d. The node arrangement satisfies the heap condition.
2. What does the term *partially ordered* mean when applied to heaps?
3. When an item is removed from a heap, it is always removed from the _____.
4. When an item is inserted into a heap,
   a. a hole is introduced at the root node and sifted down until it reaches the position the item should occupy.
   b. a search starts from the root to find the item with the key equal to or just above the key to insert, and the item is inserted as that node's child.
   c. the maximum item in the heap is moved to the first empty cell, and the new item is inserted and sifted down until it reaches the position it should occupy.
   d. the item is inserted in the first empty cell and then sifted up until it reaches the position it should occupy.

5. A heap can be represented by an array because a heap
   a. is a binary tree.
   b. is partially complete.
   c. is partially ordered.
   d. satisfies the heap condition.
6. The last node in a heap is
   a. always a left child.
   b. always a right child.
   c. always on the bottom row.
   d. never less than its sibling.
7. A heap is to a priority queue as a(n) _____ is to a stack.
8. Which operation is more complex, sifting up or down? Why?
9. The basic heapsort concept involves
   a. removing data items from a heap and then inserting them again.
   b. inserting data items into a heap and then removing them.
   c. copying data from two heaps into another empty one, merging their items.
   d. copying data from the array representing a heap to the heap tree.
10. How many arrays, each big enough to hold all the data, does it take to perform a heapsort?
11. The time complexity of running heapsort is O(_____).
12. Compared to quicksort, heapsort is _____.
13. To heapify an array:
    a. the items in the second half of the array are sifted up.
    b. the items in the array are sequentially inserted into a separate heap data structure.
    c. the array is reordered in place to establish the heap condition between nodes.
    d. the heapsort algorithm is applied to the leaf nodes of the partial heap tree.
14. The `heapify()` routine on an N-item array takes O(_____) time.
15. To get the K highest keyed items from an N-item array takes O(_____) time.

Experiments

Carrying out these experiments will help to provide insights into the topics covered in the chapter. No programming is involved.

13-A Imagine you have a heap, H, and you insert an item M on it that has a key larger than any item currently in the heap. If you now remove an item from the heap, you will get item M as the item returned. Will the heap after the two operations be identical to what it was before? By identical, we mean all the items are in the same nodes of the binary tree as they were before. Why or why not?

13-B Sorting algorithms are called stable when items with equal keys remain in the same relative order after an array of items is sorted. Does the `heapsort()` of Listing 13-7 implement a stable sort? Why or why not?

13-C While the Visualization tool does not perform a heapsort, it can shed light on Experiment 13-B. Insert some items with equal keys. Then remove them. The color of the nodes is the secondary data item. Carefully note the colors assigned to each item. Can you find examples where the items are removed in something other than the reverse of the order they were inserted?

13-D Does the order in which data is inserted in a heap affect the arrangement of the heap? Use the Heap Visualization tool to find out. Try taking a group of seven distinct keys and inserting them in different orders.

13-E Use the Visualization tool's Insert button to insert 10 items in ascending order into an empty heap. If you remove these items with the Remove Max button, will they come off in the reverse order? Does the answer change depending on which 10 items are inserted?

Programming Projects

Writing programs to solve the Programming Projects helps to solidify your understanding of the material and demonstrates how the chapter's concepts are applied. (As noted in the Introduction, qualified instructors may obtain completed solutions to the Programming Projects on the publisher's website.)

13.1 Add two `levels()` methods to the `Heap` class that return the number of levels in the heap. An empty heap has zero levels. A heap with one item has one level. The number of levels is one more than the level of the deepest leaf node. Implement `levels_loop()` by making a loop that descends the heap tree to find the deepest leaf, counting the levels along the way. Implement `levels()` by using Python's `math` package and its `log2()` function on the number of items in the heap. Test your program on heaps of sizes ranging from 0 to 33 items.

13.2 Make a version of `Heap` that accepts a flag in its constructor to select an ascending, rather than descending, heap. In other words, the item key at the root is the smallest rather than the largest, and child items have keys greater than or equal to their parent, when ascending is chosen. Make sure all operations work correctly for both ascending and descending heaps.

13.3 Implement a `merge()` method for the `Heap` class that takes a second heap and merges its items into the object. The method should check that the key functions are identical for two heaps and raise an exception if they differ. Is there a faster way to merge the heaps than removing items from the second one and inserting them in the first? Hint: Think about the analysis for heapsort and K highest. Demonstrate your program merging both empty and nonempty heaps, and heaps with different key functions.

13.4 Implement a `replaceItem()` method for the `Heap` class that behaves like the `change_priority()` method described in the "Other Operations" section. The `replaceItem()` method should take two arguments: an item already in the heap and a new item to replace it. Because the heap only has a function to extract keys from items, it cannot update the key of the existing item, so a new item must replace it. Your implementation must find the existing item in the heap and raise an exception if it is not found. After the item is found, it should replace the item and then sift it either up or down, depending on whether the new item key is larger or smaller than the old one.

The search for the item to replace can be made more efficient by not searching subheaps where the key of the existing item exceeds that of the top of the subheap. All the keys in the subheap must have equal or lower values. This filtering of subheaps can be done by starting with a queue containing just the root index. At each iteration, remove an index from the queue. If it indexes the goal item, stop and return the value. If not, put the child node indices in the queue if their keys equal or exceed the goal key. If the queue becomes empty, the item is not in the heap.

Demonstrate the results of your program for these cases:

- Item exists in heap; replacement item has a higher key
- Item exists in heap; replacement item has a lower key
- Item not in heap and has a key higher than all those in the heap
- Item not in heap and has a key lower than all those in the heap

13.5 Use the word frequency-counting program that was built for Programming Project 11.5 to analyze the 20 most frequently used and 20 least frequently used words in a text file. As a reminder, the word frequency program reads a text file, extracts the individual words, and counts the number of times they occur using a hash table. The program should traverse the items in the hash table and insert them into a plain array. The array can then be heapified first in descending order and then ascending order using the word count as the key. Use something like the `highest()` subroutine to extract and print the top 20 for each order. Be careful if the text file has fewer than 40 distinct words in it. Finding the top *K* words won't produce a good word cloud because the most frequently occurring words are likely to be the most common words in the source language of the text, but it still can yield interesting results.

CHAPTER 14

Graphs

IN THIS CHAPTER

- Introduction to Graphs
- Traversal and Search
- Minimum Spanning Trees
- Topological Sorting
- Connectivity in Directed Graphs

Graphs are among the most versatile structures used in computer programming. They appear in all kinds of problems that are generally quite different from those we've dealt with thus far in this book. If you're dealing with general kinds of data storage problems such as records in a database, you probably don't need a graph, but for some problems—and they tend to be interesting ones—a graph is indispensable.

Our discussion of graphs is divided into two chapters. In this chapter we cover the algorithms associated with unweighted graphs, show some problems that these graphs can represent, and present a visualization tool to explore them. In the next chapter we look at the more complicated algorithms associated with weighted graphs.

Introduction to Graphs

Graphs are data structures rather like trees. In a mathematical sense, a tree is a particular kind of graph. In computer programming, however, graphs are used in different ways than trees.

The data structures examined previously in this book have an architecture intertwined with the algorithms used on them. For example, a binary tree is structured the way it is because that "shape" makes it easy to search for data and insert new data. The links between nodes in a tree represent quick ways to get from parent to child.

Graphs, on the other hand, often have a shape dictated by a physical or abstract problem. For example, nodes in a graph may represent cities, whereas edges (links) may represent airline flight routes or roads or railways between the cities. Another more abstract example is a graph representing the individual tasks necessary to complete a project. In

the graph, nodes may represent tasks, whereas directed edges (with an arrow at one end) indicate which task must be completed before another. In both cases, the shape of the graph arises from the specific real-world situation.

If graphs represent real-world things, what can they be used for? Well, the ones that describe transportation links can be used to find all the possible ways of getting from one place to another. If you're only interested in the shortest path (or maybe the longest), there are algorithms that use graphs to find that path. When the graph represents communication between people, you can find clusters of people that form communities or organizations. Similarly, communication graphs can be used to find people or groups that are isolated from one another.

Before going further, we must mention that, when discussing graphs, nodes are traditionally called **vertices** (the singular is **vertex**). The links between vertices are called **edges**. The reason is probably that the nomenclature for graphs is older than that for trees, having arisen in mathematics centuries ago.

Definitions

Figure 14-1 shows a simplified map of the major freeways in the vicinity of Seattle, Washington. Next to the map is a graph that models these freeways.

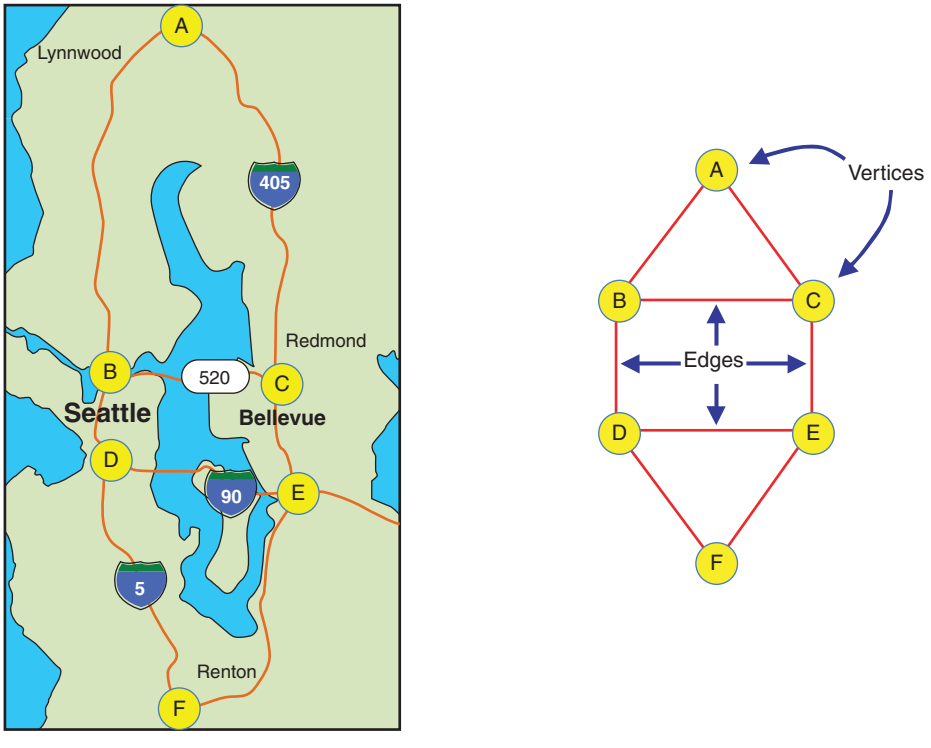


FIGURE 14-1 Roadmap and a corresponding graph

In the graph, circles represent freeway interchanges, and straight lines connecting the circles represent freeway segments. The circles are *vertices*, and the lines are *edges*. The vertices are usually labeled in some way—often, as shown here, with letters of the alphabet. Each edge connects and is bounded by the two vertices at its ends.

The graph doesn't reflect the exact geographical positions shown on the map; it shows only the relationships of the vertices and the edges—that is, which edges are connected to which vertex. It doesn't concern itself with physical distances or directions (even though the figure shows them in roughly the same orientation as the map). The primary information provided by the graph is the *connectedness* (or lack of it) of one intersection to another, not the actual routes.

Adjacency and Neighbors

Two vertices are said to be **adjacent** to one another if they are connected by a single edge. Thus, in Figure 14-1, vertices C and E are adjacent, but vertices C and F are not. The vertices adjacent to a given vertex are said to be its **neighbors**. For example, the neighbors of vertex D are B, E, and F.

Paths

A **path** is a sequence of edges. The graph in Figure 14-1 has a path from vertex A to vertex F that passes through vertices B and D. By convention, we call this path ABDF. There can be more than one path between two vertices; another path from A to F is ACEF. Because this graph came from a road network, you can easily see the correspondence of real-world routes to paths in the graph.

Connected Graphs

A graph is said to be **connected** if there is at least one path from every vertex to every other vertex, as in the graph in Figure 14-1. If "You can't get there from here" (as a rural farmer might tell city slickers who stop to ask for directions), the graph is not connected. For example, the road networks of North America are not connected to those of Japan. A **nonconnected graph** consists of several **connected components**. In Figure 14-2, two graphs with the same group of vertices are shown. In the graph on the left, all five vertices are connected, forming a single connected component. The graph on the right has two connected components: B-D and A-C-E.

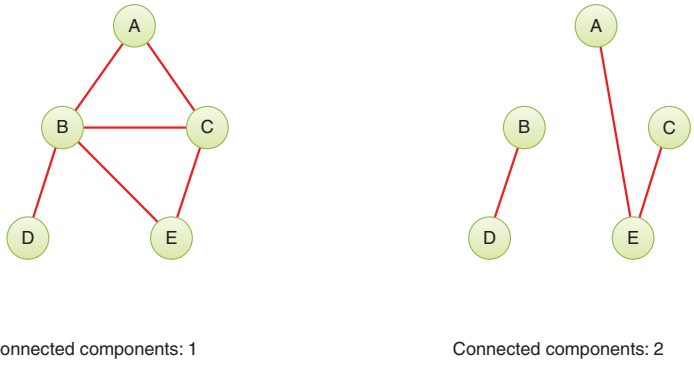


FIGURE 14-2 Connected and nonconnected graphs

Note that an edge always links two vertices in a graph. It would be incorrect to eliminate, for instance, vertex D in the right-hand graph of Figure 14-2 and leave a "dangling" edge connected to B.

For simplicity, the algorithms we discuss in this chapter are written to apply to connected graphs or to one connected component of a nonconnected graph. If appropriate, small modifications usually enable them to work with nonconnected graphs as well.

Directed and Weighted Graphs

Figure 14-1 and Figure 14-2 show **undirected graphs**. That means that the edges don't have a *direction*; you can go either way on them. Thus, you can go from vertex A to vertex B, or from vertex B to vertex A, with equal ease. Undirected graphs model rivers and roads appropriately because you can usually go either way on them (at least, slow-flowing rivers). Sometimes undirected graphs are called **bidirectional graphs**.

Graphs are often used to model situations in which you can go in only one direction along an edge—from A to B but not from B to A, as on a one-way street, the northbound or southbound lanes of a freeway, or downstream on a river with waterfalls and rapids. Such a graph is said to be **directed**. The allowed direction is typically shown with an arrowhead at the end of the edge. A valid path in a directed graph is a sequence of edges where the end vertex of edge J is the start vertex of edge J + 1.

In some graphs, edges are given a numeric **weight**. The weight is used to model something such as the physical distance between two vertices, or the time it takes to get from one vertex to another, or how much it costs to travel from vertex to vertex (on airline routes, for example). Such graphs are called **weighted graphs**. We explore them in the next chapter.

In this chapter we start the discussion on simple undirected, unweighted graphs; later we explore directed, unweighted graphs. We have by no means covered all the definitions and descriptions that apply to graphs; we introduce more as we go along.

The First Uses of Graphs

One of the first mathematicians to work with graphs was Leonhard Euler in the early eighteenth century. He solved a famous problem dealing with the bridges in the town of Königsberg, on the Baltic coast. This town on a river included an island and seven bridges, as shown in Figure 14-3.

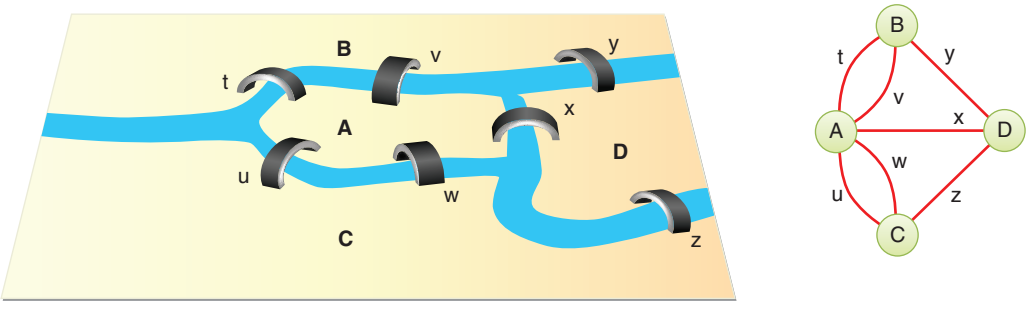


FIGURE 14-3 The bridges of Königsberg

The problem, much discussed by the townsfolk, was to find a way to walk across all seven bridges without recrossing any of them. We don't recount Euler's solution to the problem; it turns out that there is no such path. The key to his solution, however, was to represent the problem as a graph, with land areas as vertices and bridges as edges, as shown at the right of Figure 14-3. This is perhaps the first example of a graph being used to represent a problem in the real world.

Note that in the graph of the Königsberg bridges, multiple bridges connect the different land areas. For example, the island, A, connects to the lower bank of the river, C, by bridges u and w. The graph shows multiple edges, and the edges are labeled to distinguish them. The term for a graph that allows multiple edges to connect a single pair of vertices is a **multigraph**. In this the case, the edge labels aren't weights—just ways to distinguish the possible paths between vertices A and C.

Representing a Graph in a Program

It's all very well to think about graphs in the abstract, as Euler and other mathematicians did, but you want to represent graphs using a computer. What sort of software structures are appropriate to model graphs? Let's look at vertices first and then at edges.

Vertices

In an abstract graph program, you could simply number the vertices 0 to N–1 (where N is the number of vertices). You wouldn't need any sort of variable to hold the vertices because their usefulness would result from their relationships with other vertices.

In most situations, however, a vertex represents some real-world object, and the object must be described using data items. If a vertex represents a city in an airline route simulation, for example, it may need to store the name of the city, the name of the airport, its altitude, its location, runway orientations, and other such information. Thus, it's usually convenient to represent a vertex by an object of a vertex class. Our example programs store only a name string (like *A*), used as a label for identifying the vertex. Listing 14-1 shows how the basic `Vertex` class might look.

LISTING 14-1 The Basic `Vertex` Class

```
class Vertex(object):        # A vertex in a graph
   def __init__(self, name): # Constructor: stores a vertex name
      self.name = name       # Store the name

   def __str__(self):        # Summarize vertex in a string
      return '<Vertex {}>'.format(self.name)
```

Note that the name attribute is declared public here. The reason is that you can allow it to be manipulated by the caller during various operations without affecting the graph containing it.

Vertex objects can be placed in an array and referred to using their index number. The vertices might also be placed in a list or some other data structure. The unique vertex index or the object itself can identify this vertex within a graph.

For vertices with coordinates like latitude and longitude, storing them in a quadtree may make sense, as described in Chapter 12, "Spatial Data Structures." It is important to have them in some structure that preserves their unique identifiers so that even if the caller changes the name or other attribute, the vertex can be retrieved. If you use a quadtree, you could not allow the coordinates of the vertex to change without changing its placement in the quadtree. For graphs with simply labeled vertices, an array is fine for storage if the vertex always stays at its original index.

Whatever structure is used for vertices, this storage is for convenience only. It has no relevance to how they are connected by edges. For edges, you need another mechanism.

Edges

In Chapter 8, "Binary Trees," you saw that a computer program can represent trees in several ways. Mostly that chapter examined trees in which each node contained references to its children, but you also learned that an array could be used, with a node's position in the array indicating its relationship to other nodes. Chapter 13, "Heaps," described arrays used to represent a particular kind of tree called a *heap*.

A graph, however, doesn't usually have the same kind of fixed organization as a tree. In a binary tree, each node has a maximum of two children, but each vertex in a graph may be connected to an arbitrary number of other vertices. For example, in Figure 14-2, the left-hand graph's vertex B is connected to four other vertices, whereas D is connected to only one.

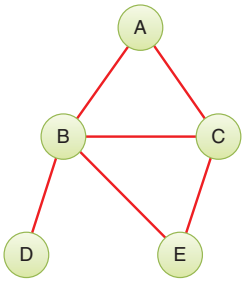
To model this sort of free-form organization, a different approach to representing edges is preferable to that used for trees. Several methods are commonly used for graphs. We examine two (although some might call it three): the *adjacency matrix* and the *adjacency list*. Remember that one vertex is said to be *adjacent* to another if they're connected by a single edge, not a path through several edges.

The Adjacency Matrix

An **adjacency matrix** can be represented as a two-dimensional array in which the elements indicate whether an edge is present between two vertices. If a graph has N vertices, the adjacency matrix is an N×N array. Table 14-1 shows an example of an adjacency matrix for the left-hand graph of Figure 14-2, repeated here.

Table 14-1 Adjacency Matrix

	A	B	C	D	E
A	0	1	1	0	0
B	1	0	1	1	1
C	1	1	0	0	1
D	0	1	0	0	0
E	0	1	1	0	0



The vertex labels are used as headings for both rows and columns. An edge between two vertices is indicated by a 1; the absence of an edge is a 0. (You could also use Boolean `True`/`False` values.) As you can see, vertex B is adjacent to all four other vertices; A is adjacent to B and C; C is adjacent to A, B, and E; D is adjacent only to B; and E is adjacent to B and C. In this example, the "connection" of a vertex to itself is indicated by 0, so the diagonal from upper left to lower right, A-A to E-E, which is called the **identity diagonal** and shaded gray, is all 0s. The entries on the identity diagonal don't convey any real information, so you can equally well put 1s along it, if that's more convenient to a program. (A kind of graph called a *pseudograph* allows edges that go from a vertex to itself. Such graphs can use the diagonal of the adjacency matrix to represent the presence or absence of such edges.)

Note that the triangular-shaped part of the matrix above the identity diagonal is a mirror image of the part below; both triangles contain the same information. This redundancy is a bit inefficient, but there's no simple way to create a triangular array in most computer languages, so it's simpler to accept the redundancy. Consequently, when you add an edge to the graph, you make two entries in the adjacency matrix rather than one.

Note, in this chapter, we focus on unweighted graphs, and the edges don't need separate labels like in the bridges of Königsberg. Graphs that allow multiple edges between a single pair of vertices are called *multigraphs*. They can be quite useful as in the case of the bridges of Königsberg but are beyond the scope of this text.

Using Hash Tables for the Adjacency Matrix

One way to improve storage efficiency is to keep the matrix as a hash table rather than a two-dimensional array. To do this, the hash table must accept keys with two parts—one for each vertex. That's straightforward, as discussed in Chapter 11, "Hash Tables." The individual characters in a string or the indices in a tuple can be hashed using different weights in the hashing function to produce a single hash index.

To represent an edge using a hash table adjacency matrix, you make an entry at a particular pair of vertices. For example, to add an edge between vertices 2 and 7, you would insert a value for the key `(2, 7)`. Using the data structures from Chapter 11, you could write

```
adjacencyMatrix = HashTable()
adjacencyMatrix.insert((2, 7), True)
```

After the edges are placed in the matrix, programs can determine whether two edges are adjacent by using the hash table's `search()` method. For example, to see whether two vertices are adjacent, you would evaluate `adjacencyMatrix.search((4, 12))`. If no entry had been made for (4, 12), the search would return `None`, which Python treats as False in a Boolean context. If an entry had been made for that key, its `True` value would be returned.

With either hash tables or two-dimensional arrays, you can avoid the duplicate storage of the two triangular halves of the matrix by always using the smaller vertex index in the first position. In other words, to check if vertex 19 is adjacent to vertex 8 by a bidirectional edge, you would check the matrix at `(8, 19)` instead of `(19, 8)`. Reordering the vertex indices would be necessary for all operations on the matrix—insertion, deletion,

and searching. That adds a little extra time to each of those operations. The alternative of adding time to the insertion and deletion operations by using both orderings and updating both matrix cells is usually preferable when searching will be much more frequent than inserting and deleting. It also is better for directional graphs, as you see later.

Hash tables have an extra benefit over two-dimensional arrays when there are few edges. If a graph has, for example, a thousand vertices and two thousand edges, the two-dimensional array needs storage for a million cells while the hash table needs only enough for the two thousand edges (so perhaps four thousand cells). The difference becomes significant for large graphs because the memory needed for two-dimensional arrays is $O(N^2)$, where N is the number of vertices, limiting what can be kept in the computer's memory.

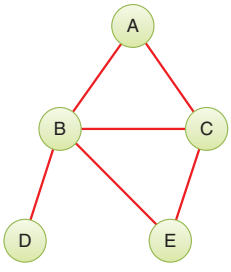
On the other hand, two-dimensional arrays are easier to decompose into rows and columns. Many graph algorithms were developed to take advantage of the capability to access rows and columns quickly. The choice of representation depends on what operations need to be done quickly on the graph.

The Adjacency List

The other common way to represent edges is with a list. The list in **adjacency list** refers to a linked list of the kind examined in Chapter 5, "Linked Lists." In actuality, an adjacency list is an array of lists, or sometimes a list of lists, or sometimes a list stored in each `Vertex`. Each individual list contains references to the adjacent vertices of that vertex. Table 14-2 shows the adjacency lists for the left-hand graph of Figure 14-2, repeated here.

Table 14-2 Adjacency Lists

Vertex	Adjacent Vertices List
A	B➔C
B	A➔C➔D➔E
C	A➔B➔E
D	B
E	B➔C



In this table, the ➔ symbol indicates a link in a linked list. Each link in the list is a vertex. Here, the vertices are arranged in alphabetical order in each list, although that's not strictly necessary. More likely, the order of the vertices would depend on the order the edges were added to the graph.

Don't confuse the contents of adjacency lists with paths. The adjacency list shows which vertices are adjacent to—one edge away from—a given vertex, not paths from vertex to vertex.

In the next chapter we discuss when to use one of the adjacency matrices as opposed to an adjacency list. The Visualization tool shown in this chapter shows the adjacency matrix approach, but in many cases, the list approach is more efficient.

Adding Vertices and Edges to a Graph

To add a vertex to a graph, you make a new `Vertex` object, insert it into a vertex array or list, and grow the adjacency structure, as we discuss shortly. In a real-world program, a vertex might contain many data elements, but for simplicity of the program examples, you can assume that it contains only a single attribute, its name. (The Visualization tool also associates different colors with the vertices in addition to their names to indicate the different data they reference.) Thus, the creation of a vertex looks something like this:

```
vertices.append(Vertex('F'))
```

This command inserts a vertex object named F at the end of the Python array called `vertices`.

How you add an edge to a graph depends on whether you're using an adjacency matrix or adjacency lists to represent the graph. Let's say that you're using an adjacency matrix and want to add an edge between vertices 1 and 3. These numbers correspond to the array indices in `vertices` where the vertices are stored.

When the adjacency matrix was created, it would have a particular size. If adding a new vertex means it needs an extra row and column, the array will need to grow. When the adjacency matrix, `adjMat`, is first created, it is filled it with 0s or perhaps a Boolean `False`. When it grows, all the new cells must be filled with that same value. Adjacency matrices represented as hash tables do not require growing when new vertices are added. They grow when edges are added as new keys as described in Chapter 11, "Hash Tables."

Two-Dimensional Arrays and Python

To insert the edge between vertices 1 and 3 in two-dimensional arrays in Java or C++, you could write

```
adjMat[1][3] = 1;
adjMat[3][1] = 1;
```

The core Python language supports only one-dimensional arrays, although extensions like NumPy provide multidimensional arrays. You can store an array in the cell of another array to approximate multidimensional arrays. For example, you can create an array of vertices plus an adjacency matrix, and insert an edge with code like the following:

```
vertices = []                   # A list of vertices
vertices.append(Vertex('A'))
vertices.append(Vertex('B'))
vertices.append(Vertex('C'))
vertices.append(Vertex('D'))

adjMat = [ [False for v in range(len(vertices))]
           for _ in range(len(vertices)) ]
adjMat[1][3] = True
adjMat[3][1] = True
```

To create the adjacency matrix, we've used nested list comprehensions that create an outer array filled with arrays. The inner comprehension, `[False for v in range(len(vertices))]`, creates a list/array of four cells filled with the value `False`. The outer comprehension repeats the inner creation four times to create four cells filled with lists/arrays. The result of the last two assignment statements leaves `adjMat` holding

```
[[False, False, False, False],
 [False, False, False, True],
 [False, False, False, False],
 [False, True,  False, False]]
```

The `True` values show up in the rows and columns indexed by (1, 3) and (3, 1). Note that Python's capability to expand a list/array using the multiplication operator doesn't work correctly here. If you try

```
badMat = [ [False] * len(vertices) ] * len(vertices)
badMat[1][3] = True
```

the resulting `badMat` array contains

```
[[False, False, False, True],
 [False, False, False, True],
 [False, False, False, True],
 [False, False, False, True]]
```

That's not what you want, but why did it come out that way? The reason is that the multiplication operator fills the cells of the expanded list with the *exact same value*. The inner multiply creates a four-cell array of `False` values. The outer multiply does the same kind of expansion, and every cell is filled with a reference to the *same* inner four-cell array. Because the same inner array is shared among all the cells of the outer array, assigning `True` to cell 3 in one row affects all the rows.

Storing Edges in Adjacency Lists

Returning to the concept of using adjacency lists, they have a similar nested list/array structure, but with a significant difference. Instead of each row having the same length, they are variable-length lists of vertex indices. For example,

```
adjList = [ [] for v in range(len(vertices)) ]
adjList[1].append(3)
adjList[3].append(1)
```

produces an array containing

```
[[], [3], [], [1]]
```

The cells of this array contain lists of vertex indices. Cell 0 has the (empty) list of vertices for vertex 0, and so on. The complete adjacency list shows that vertices 0 and 2 have no edges, whereas vertices 1 and 3 share an edge.

Note that the multiplication operator also fails to work for adjacency lists too. For example,

```
badList = [[]] * len(vertices)
badList[1].append(3)
```

produces

```
[[3], [3], [3], [3]]
```

The list comprehension method effectively executes a loop to create separate elements for its output array; that's how it builds separate lists on each iteration.

The Graph Class

Let's make this discussion more concrete with a Python `Graph` class that constructs a vertex list and an adjacency matrix and contains methods for adding vertices and edges. Listing 14-2 shows the code.

This implementation makes use of Python's `list` type to manage the list of vertices and the `dict` type to store the adjacency matrix. These two structures behave like the `Stack` and `HashTable` classes you saw in earlier chapters, but with some different syntax. The constructor for the `Graph` creates an empty `list` called `_vertices` and an empty `dict` called `_adjMat`. You saw hash tables used to store a grid of cells in Chapter 12, "Spatial Data Structures." In case you skipped that, we describe the use of Python's `dict` type as a dictionary (hash table) in more detail here.

The pair of curly braces, `{}`, in the constructor creates an empty hash table that can accept most Python data structures as a key. For this graph, we use tuples of vertex indices, like `(2, 7)`, as the keys to the adjacency matrix. When we need to store a 1 in the adjacency matrix for a particular tuple, we could write

```
self._adjMat[(2, 7)] = 1
```

The square brackets after `_adjMat` surround the key to the hash table. That tells Python that it should hash the key inside to find where to place the value in the hash table. Later we can retrieve the value from the hash table using the same syntax.

You can also use the slightly simpler syntax

```
self._adjMat[2, 7] = 1
```

to do the same thing. To programmers familiar with other languages, this syntax might look like a multidimensional array reference, but it is not. The comma in the expression within the brackets tells Python to construct a **tuple**. In this case, it constructs the tuple `(2, 7)` and uses that as the key for the hash table. The hashing function uses all the tuple elements in calculating the hash table index. This means that the value gets stored in a unique location for the key `(2, 7)`. In a two-dimensional array, the cell at row 2, column 7 would be addressed, but in the hash table, some location in its one-dimensional array is used. To the calling program, it doesn't matter which one, as long as that exact same cell is found when it references `(2, 7)` in the hash table later.

LISTING 14-2 The Basic `Graph` Class

```
class Graph(object):          # A graph containing vertices and edges
   def __init__(self):        # Constructor
      self._vertices = []     # A list/array of vertices
      self._adjMat = {}       # A hash table mapping vertex pairs to 1

   def nVertices(self):       # Get the number of graph vertices, i.e.
      return len(self._vertices) # the length of the vertices list

   def nEdges(self):          # Get the number of graph edges by
      return len(self._adjMat) // 2 # dividing the # of keys by 2

   def addVertex(self, vertex): # Add a new vertex to the graph
      self._vertices.append(vertex) # Place at end of vertex list

   def validIndex(self, n):  # Check that n is a valid vertex index
      if n < 0 or self.nVertices() <= n: # If it lies outside the
         raise IndexError    # valid range, raise an exception
      return True             # Otherwise it's valid

   def getVertex(self, n):   # Get the nth vertex in the graph
      if self.validIndex(n): # Check that n is a valid vertex index
         return self._vertices[n] # and return nth vertex

   def addEdge(self, A, B):  # Add an edge between two vertices A & B
      self.validIndex(A)     # Check that vertex A is valid
      self.validIndex(B)     # Check that vertex B is valid
      if A == B:             # If vertices are the same
         raise ValueError    # raise exception
      self._adjMat[A, B] = 1 # Add edge in one direction and
      self._adjMat[B, A] = 1 # the reverse direction

   def hasEdge(self, A, B):  # Check for edge between vertices A & B
      self.validIndex(A)     # Check that vertex A is valid
      self.validIndex(B)     # Check that vertex B is valid
      return self._adjMat.get( # Look in adjacency matrix hash table
         (A, B), False)      # Return either the edge count or False
```

Note that the Python syntax for addressing one-dimensional arrays and hash tables is identical. When Python sees `var[2]`, it looks at the type of the `var` variable to determine what to do next. If `var` is a `list`, then the expression in the brackets should be an integer index to one of its array cells. If `var` is a `dict`, then the contents of the brackets hold a key to be hashed. The tuples of vertex numbers such as `(2, 7)` are treated as a single hash key.

Moving on to the first method defined in the `Graph` class of Listing 14-2, you find that `nVertices()` makes use of Python's `len()` function to get the length of the list/array holding the vertices. A freshly constructed `Graph` has an empty `_vertices` list, so the length is zero.

The second method, `nEdges()`, is similar but uses the length of the hash table, `_adjMat`, in its calculation. Python uses the number of keys that have been stored in the hash table as its length. Because we plan to store vertex pairs along with their mirror image—for example `(2, 7)` and `(7, 2)`—as separate keys for each edge, the total number of edges is half the number of keys.

The third method, `addVertex()`, adds a vertex to the graph by using Python's built-in `append()` method on the `_vertices` list. This is just like pushing an element on a stack (assuming the top of the stack is the end of the list). We left out a check in this method that the `vertex` argument is one of the `Vertex` objects as defined in Listing 14-1, but that would be good to include.

Next, the program introduces a simple test for a valid vertex index, `validIndex()`. Because callers specify vertices by their index, they could specify indices outside the range of those already added to the graph. This *predicate*—a function with a Boolean result—checks whether the index lies outside of the range `[0, nVertices)`, raising the `IndexError` exception if it does. This is the same exception that Python uses for invalid array indices.

The `getVertex()` method gets a vertex from the graph based on an index, `n`. After the index is validated, the vertex object can be retrieved from the array.

The `addEdge()` method takes two vertex indices, `A` and `B`, as parameters. To create the edge, it first verifies that both `A` and `B` are valid vertex indices. Without these checks, the `Graph` could become internally inconsistent. It also checks whether `A` and `B` are the same index. That would create an edge from a vertex to itself. This simple `Graph` class doesn't allow the creation of pseudographs. Finally, the method updates the adjacency matrix to create the edge. It uses both orders of the vertices because the edge is bidirectional.

We now have a basic `Graph` object that can expand to accept any number of vertices and edges. The next important method is one that tests whether an edge exists between two vertices. The `hasEdge()` method takes two vertex indices, `A` and `B`, as parameters. They are checked for validity like before. With valid indices, the adjacency matrix can be checked to see whether an edge was defined. You might expect to use the same Python syntax, `_adjMat[A, B]`, to test for that edge. That, however, causes a problem if the edge does not exist. Python hash tables raise a `KeyError` exception when asked to access a key that has not been previously inserted.

There are a couple of ways to address missing keys in the adjacency matrix. One is to catch the `KeyError` exception that could occur when accessing `_adjMat[A, B]`. The other is to use Python's `get()` method for hash tables, which is what this implementation does. The `get()` method takes the key as the first parameter plus a second one for the value to return if the key is not in the hash table. We pass the tuple, `(A, B)`, for the key. The default value is set to `False` so that `hasEdge()` returns that when the edge does not exist. If the edge does exist, the call to `get()` will return 1, the value that was inserted by `addEdge()`, which is interpreted as `True` in Boolean contexts.

Another way to implement the `hasEdge(A, B)` check would be to return the value of this expression `(A, B) in self._adjMat`. Python uses the `in` operator to test whether a key has been inserted in a hash table. Because we only ever set the values in `_adjMat` to 1, we could ignore the value and only check for the presence of the key. When edges are deleted, however, we could not simply set the value to 0 if we use the `(A, B) in self._adjMat` test; we would need to remove the key from the hash table.

Traversal and Search

One of the most fundamental operations to perform on a graph is finding which vertices can be reached from a specified vertex. This operation is used to find the connected components of a graph and underlies many more complex operations. When we look at a graph diagrams, it's usually immediately obvious which vertices are connected, at least for simple graphs. To the computer, however, it must discover what vertices are connected by chaining together edges. Let's look at some examples of discovering what's connected.

Imagine that you are traveling to a foreign country by airplane or boat for a vacation. When there, you will be visiting the countryside by bicycle, and you want to know all the places you can go. Paved roads and some dirt roads would be good for the trip, but roads that are closed for repair or washed out with mud or floods are not worth traversing. The road conditions might be cataloged somewhere, but in some cases, you would have to go to a nearby location to find out whether the road was passable. Based on the road conditions, some towns could be reached, whereas others couldn't. You still want to have the best vacation possible, so you would determine what places are accessible via bicycle and decide your route among them. You might have to remake the plan as you travel and learn more about road conditions in each area.

Other situations where you need to find all the vertices reachable from a specified vertex are in designing circuits and plumbing networks. Electronic circuits are composed of components like transistors that are connected by conductors such as wires or metal paths. In plumbing networks various components such as water heaters, faucets, drains, and gas stoves connect via pipes. Figure 14-4 shows small examples of an electronic circuit and a plumbing network. In both cases, many of the components are connected, and others are not. It's critical to their function that the connections are complete. If not, hot water might not reach a particular faucet, or the signal from an antenna might not reach a decoder. Perhaps even more important, the disconnected vertices must remain that way—lest the power supply connect directly to a speaker or the gas supply connect to a faucet.

In these kinds of networks, it's important to find all the vertices connected to a given vertex. That defines the connected components of the graph. Each connected component must be realized in the form of electrical or plumbing connections. That's often easy for you to see in diagrams like Figure 14-4, especially when the colors of the edges in the plumbing diagram clearly separate the kinds of pipes. It's much less clear in the case of the road network for bicycling, especially when you must travel to an area to learn what's connected and what's not.

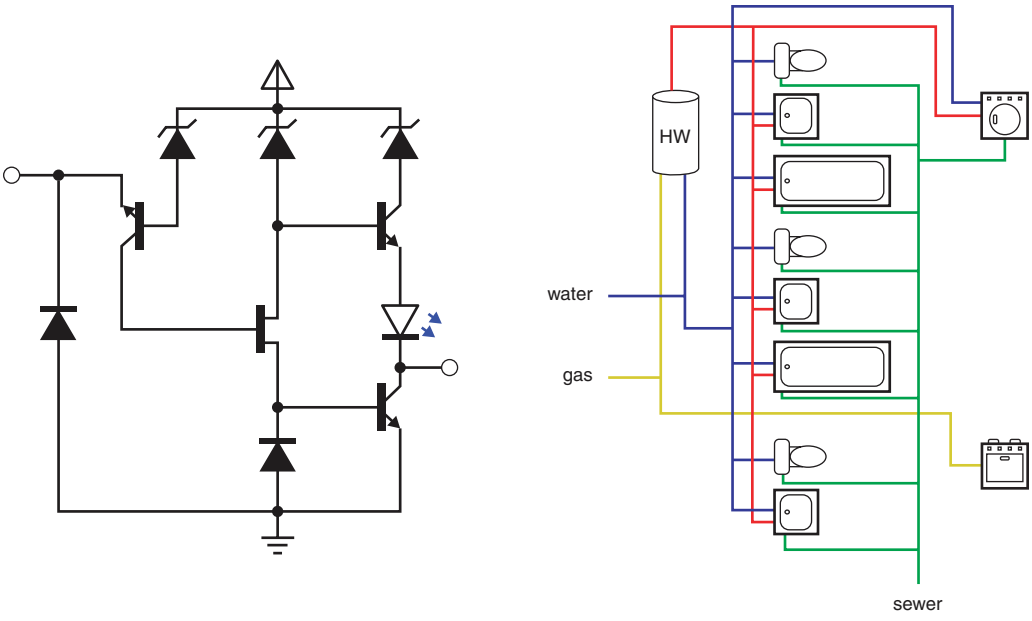


FIGURE 14-4 An electronic circuit and a plumbing network

Assume that you've been given a graph that describes a network. Now you need an algorithm that provides a systematic way to start at a specified vertex and move along edges to other vertices in such a way that, when it's done, you are guaranteed that it has *visited* every vertex that's connected to the starting vertex. Here, as it did in Chapter 8, where we discussed binary trees, *visit* means to perform some operation on the vertex, such as displaying it, adding it to a collection, or updating one of its attributes.

There are two common approaches to traversing a graph: **depth-first** (*DF*) and **breadth-first** (*BF*). Both eventually reach all connected vertices but differ in the order they visit them. The depth-first traversal is implemented with a stack, whereas breadth-first is implemented with a queue. You can traverse all the connected vertices or perhaps stop when you find a particular vertex. When the goal is to stop at a particular vertex, the operation is called **depth-first search** (*DFS*) or **breadth-first search** (*BFS*) instead of traversal.

You saw in Chapter 8 that the different traversal orders of binary trees—pre-order, in-order, and post-order—had different properties and uses. The same is true of graphs. The choice of depth-first or breadth-first depends on the goal of the operation.

Depth-First

The depth-first traversal uses a stack to remember where it should go when it reaches a dead end (a vertex with no adjacent, unvisited vertices). We show an example here, encourage you to try similar examples with the Graph Visualization tool, and then finally show some code that carries out the traverse.

An Example

Let's look at the idea behind the depth-first traversal in relation to the graph in Figure 14-5. The colored numbers and dashed arrows in this figure show the order in which the vertices are visited (which differ from the ID numbers used to identify vertices).

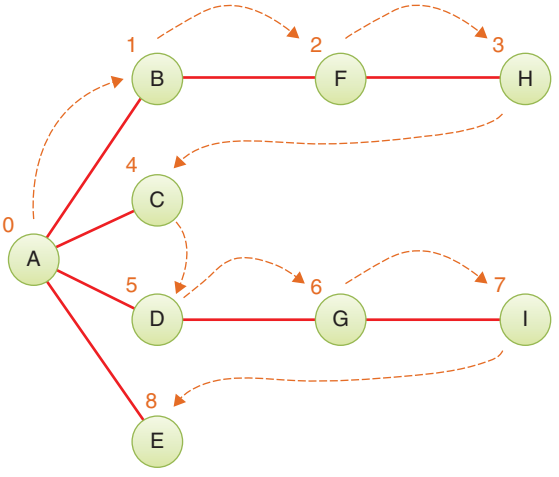


FIGURE 14-5 Depth-first traversal example

To carry out the depth-first traversal, you pick a starting point—in this case, vertex A. You then do three things: visit this vertex, push it onto a stack so that you can remember it, and mark it so that you won't visit it again.

Next, you go to any vertex adjacent to A that hasn't yet been visited. We'll assume the vertices are selected in alphabetical order, so that brings up B. You visit B, mark it, and push it on the stack.

Now what? You're at B, and you do the same thing as before: go to an adjacent vertex that hasn't been visited. This leads you to F. We can call this process Rule 1.

Rule 1

If possible, visit an adjacent unvisited vertex, mark it, and push it on the stack.

Applying Rule 1 again leads you to H. At this point, however, you need to do something else because there are no unvisited vertices adjacent to H. Here's where Rule 2 comes in.

Rule 2

If you can't follow Rule 1, then, if possible, pop a vertex off the stack.

Following Rule 2, you pop H off the stack, which brings you back to F. F has no unvisited adjacent vertices, so you pop it. The same is true of vertex B. Now only A is left on the stack.

A, however, does have unvisited adjacent vertices, so you visit the next one, C. The visit to C shows that it is the end of the line again, so you pop it, and you're back to A. You

visit D, G, and I, and then pop them all when you reach the dead end at I. Now you're back to A. You visit E, and again you're back to A.

This time, however, A has no unvisited neighbors, so you pop it off the stack. Now there's nothing left to pop, which brings up Rule 3.

Rule 3

If you can't follow Rule 1 or Rule 2, you're done.

Table 14-3 shows how the stack looks in the various stages of this process, as applied to Figure 14-5. The contents of the stack show the path you took from the starting vertex to get where you are (at the top of the stack). As you move away from the starting vertex, you push vertices as you go. As you move back toward the starting vertex, you pop them. The order in which you visit the vertices is ABFHCDGIE. Note that this is not a path, just a vertex list, because H is not adjacent to C, for example.

Table 14-3 Stack Contents During Depth-First Traversal

Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF
Visit H	ABFH
Pop H	ABF
Pop F	AB
Pop B	A
Visit C	AC
Pop C	A
Visit D	AD
Visit G	ADG
Visit I	ADGI
Pop I	ADG
Pop G	AD
Pop D	A
Visit E	AE
Pop E	A
Pop A	
Done	

You might say that the depth-first algorithm likes to get as far away from the starting point as quickly as possible and returns only when it reaches a dead end. If you use the term *depth* to mean the distance from the starting point, you can see where the name *depth-first* comes from.

An Analogy

One analogy to a depth-first search is solving a maze. The maze—perhaps one of the people-size ones made of hedges, popular in England, or corn stalks, popular in America—consists of narrow passages (think of edges) and intersections where passages meet (vertices).

Suppose that Minnie is lost in a maze. She knows there's an exit and plans to traverse the maze systematically to find it. Fortunately, she has a ball of string and a marker pen. She starts at some intersection and goes down a randomly chosen passage, unreeling the string. At the next intersection, she goes down another randomly chosen passage, and so on, until finally she reaches a dead end.

At the dead end, she retraces her path, reeling in the string, until she reaches the previous intersection. Here she marks the path she's been down, so she won't take it again, and tries another path. When she's marked all the paths leading from that intersection, she returns to the previous intersection and repeats the process.

The string represents the stack in depth-first: it "remembers" the path taken to reach a certain point. The pen represents marking vertices as visited. If Minnie didn't have a pen, she could always choose, say, the leftmost passage to visit when arriving at an intersection. When returning along the string and coming back to an intersection, she would choose the next passage to the right to visit and continue to follow the string back if there were no more passages to the right. The critical thing is that Minnie needs to remember having visited an intersection/vertex so that she doesn't just revisit them. The sense of left and right doesn't exist in most graphs, and computers don't use pens; they must use explicit marking.

The Graph Visualization Tool and Depth-First Traverse

You can try out the depth-first traversal with the Depth-First Traverse button in the Graph Visualization tool. Start the tool (as described in Appendix A, "Running the Visualizations"). At the beginning, there are no vertices or edges, just an empty shaded rectangle. You create vertices by double-clicking the desired location within the shaded box. The first vertex is automatically labeled A, the second one is B, and so on. They're each given a different color.

To make an edge, drag the pointer from one vertex to another. Figure 14-6 shows part of the graph of Figure 14-5 as it looks while being created using the tool. The adjacency matrix appears in the lower-right corner. When the Visualization tool first starts, there are no vertices, and the matrix is empty. The matrix and the `_vertices` table in the upper right grow as you add vertices.

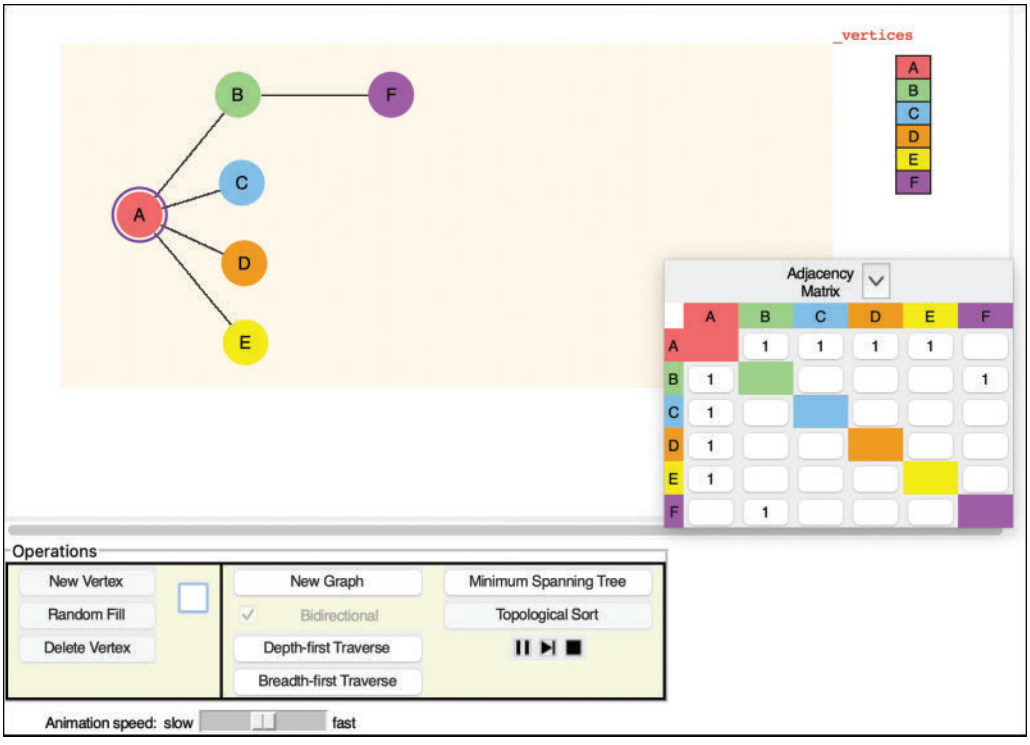


FIGURE 14-6 The Graph Visualization tool

You can edit the graph in many ways. Like the other Visualization tools, this one has buttons in the operations area that allow you to create and delete vertices by their label. Because the tool must find vertices by their label, all the names must be unique (and short enough to fit in the little circles and rectangles). When you add a vertex—say with the label V—the program will choose a random location for it within the shaded box and conveniently update the label in the text entry box to W in case you plan to add another. That allows you to click New Vertex N times to create N vertices with unique names. You can also put a number in the text entry box and select the Random Fill button to create that many more vertices.

If you want to rearrange the vertex positions, press and hold the Shift key while you click a vertex and drag it to its new position. The Visualization tool allows you to place the vertex only where it lies completely within the shaded box and does not overlap another vertex. If you want to get rid of a vertex or an edge, you can double-click it.

Another way to delete edges is by clicking the box corresponding to that edge in the adjacency matrix table. Because edges are either present or absent in this kind of graph, each click toggles the edge on or off. You can collapse the adjacency matrix by pressing the ∨ button and restore it with the > button. That capability is especially useful for large graphs.

To run the depth-first traversal algorithm, select a starting vertex by clicking it and then select the Depth-First Traverse button. When you click a vertex, the blue circle marks it

as the starting vertex for the traversal, as vertex A is marked in Figure 14-6. Try it on the graph shown in Figure 14-6 or one of your own design. The animation shows the creation of a stack below the shaded box and how vertices are pushed on and popped off to traverse the graph. Of course, the traversal covers only the connected component that includes the selected vertex.

Python Code

The depth-first traversal algorithm must find the vertices that are unvisited and adjacent to a specified vertex. How should this be done? The adjacency matrix holds the answer. By going to the row for the specified vertex and stepping across the columns, you can pick out the columns with a 1; the column number is the number of an adjacent vertex. You can then check whether this vertex is unvisited by examining the value for that vertex in a separate array called `visited`. If that value is `False`, you've found what you want—the next vertex to visit. If no vertices on the row are simultaneously 1 (adjacent) and unvisited, there are no unvisited vertices adjacent to the specified vertex. The code for this process is made up of several parts—a generator for all vertices, a generator for getting adjacent vertices, and a generator for getting unvisited vertices—as shown in Listing 14-3.

LISTING 14-3 Code for Traversing Adjacent Vertices

```
class Graph(object):
…
   def vertices(self):      # Generate sequence of all vertex indices
      return range(self.nVertices()) # Same as range up to nVertices
   def adjacentVertices(    # Generate a sequence of vertex indices
         self, n):          # that are adjacent to vertex n
      self.validIndex(n)    # Check that vertex n is valid
      for j in self.vertices(): # Loop over all other vertices
         if j != n and self.hasEdge(n, j): # If other vertex connects
            yield j         # via edge, yield other vertex index

   def adjacentUnvisitedVertices( # Generate a sequence of vertex
         self, n,           # indices adjacent to vertex n that do
         visited,           # not already show up in the visited list
         markVisits=True):  # and mark visits in list, if requested
      for j in self.adjacentVertices(n): # Loop through adjacent
         if not visited[j]: # vertices, check visited
            if markVisits:  # flag, and if unvisited, optionally
               visited[j] = True # mark the visit
            yield j         # and yield the vertex index
```

The generator for all vertex indices is the same as Python's `range()` generator for index values up to `nVertices - 1`. The `adjacentVertices()` generator takes a vertex index, `n`, as the starting vertex. It checks that `n` is within the bounds of the known vertices and then starts a loop over all the other vertex indices. For another vertex, `j`, that is not `n` but does have an edge in the adjacency matrix to `n`, the generator yields vertex `j` for processing by

the caller. When all `nVertices` have been checked, the generator is finished, and it raises the `StopIteration` exception.

One way to implement the marking of vertices as visited or unvisited is to share a data structure between the caller and the generator. The simplest approach uses an array of Boolean flags indicating which of the `nVertices` have been visited.

The `adjacentUnvisitedVertices()` generator of Listing 14-3 takes a starting vertex index, `n`, plus a `visited` array and a `markVisits` flag as parameters. The `visited` array should have at least one cell for all the vertices, initially with all false values. This array can be created in Python using an expression like `[False] * nVertices` or `[None] * nVertices`. When `markVisits` is true, the generator will set the flag for the cells it visits to `True`.

By using `adjacentVertices()` to enumerate the vertices adjacent to vertex `n`, all that remains is the check for whether they have been visited. If they have not, they are optionally marked as visited before yielding them. With this kind of generator, it's easy to define different traversal orderings like depth-first, as shown in Listing 14-4.

LISTING 14-4 Implementation of Depth-First Traversal of a `Graph`

```
class Stack(list):          # Use list to define Stack class
   def push(self, item): self.append(item) # push == append
   def peek(self): return self[-1] # Last element is top of stack
   def isEmpty(self): return len(self) == 0

class Graph(object):
…
   def depthFirst(          # Traverse the vertices in depth-first
         self, n):          # order starting at vertex n
      self.validIndex(n)    # Check that vertex n is valid
      visited = [False] * self.nVertices() # Nothing visited initially
      stack = Stack()       # Start with an empty stack
      stack.push(n)         # and push the starting vertex index on it
      visited[n] = True     # Mark vertex n as visited
      yield (n, stack)      # Yield initial vertex and initial path
      while not stack.isEmpty(): # Loop until nothing left on stack
         visit = stack.peek() # Top of stack is vertex being visited
         adj = None
         for j in self.adjacentUnvisitedVertices( # Loop over adjacent
            visit, visited): # vertices marking them as we visit them
            adj = j         # Next vertex is first adjacent unvisited
            break           # one, and the rest will be visited later
         if adj is not None: # If there's an adjacent unvisited vertex
            stack.push(adj) # Push it on stack and
            yield (adj, stack) # yield it with the path leading to it
         else:              # Otherwise we're visiting a dead end so
            stack.pop()     # pop the vertex off the stack
```

The depth-first algorithm needs a stack that keeps track of the path of vertices as it traverses the graph. Python's `list` data type acts like a stack, including having a `pop()` method. Because it does not have a corresponding `push()` or `peek()` method, we define those in terms of equivalent operations in the simple `Stack` subclass of `list`. The four lines of Python code at the top of Listing 14-4 show how little needs to be changed to implement a stack with a list.

The `depthFirst()` generator traverses the vertices using the rule-based approach outlined previously. After checking the index for the starting vertex, it makes a `visited` array filled with `False` for each of the `nVertices`. After the stack is created, the starting vertex, `n`, is pushed on it and marked as visited. The generator can now yield the first vertex, `n`, in the depth-first traversal.

The `depthFirst()` generator yields both the vertex index being visited and the stack (path) to the vertex because different callers need one or both of those. For example, to find the connected components of the graph you would need to collect only the vertices, whereas to solve a maze you would need the path that reaches the exit.

The main `while` loop of the `depthFirst()` generator applies the three rules. The top of the stack is the last vertex visited. It uses `peek()` to get its index and stores it in `visit`. The rules need to know if there are unvisited vertices adjacent to the `visit` vertex. The method assumes there are none by setting `adj` to `None` and calling `adjacentUnvisitedVertices()` generator to find such a vertex. If one is yielded, `adj` is updated with its index, and the inner `for` loop is exited. If none are found, `adj` remains `None`. The depth-first traversal needs only to find the first unvisited neighbor. The others will be found during a later pass through the loop.

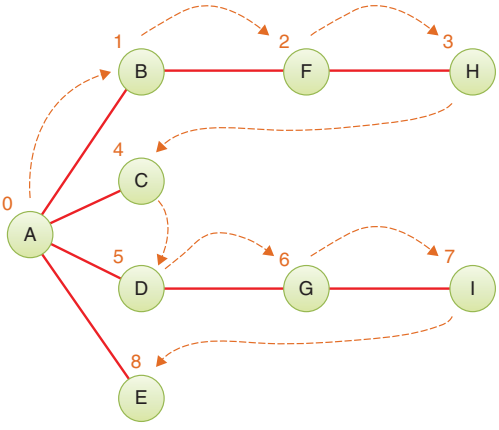
Now we can test for Rule 1: *if there is an adjacent unvisited vertex, mark it as visited and push it on the stack*. The marking is done by `adjacentUnvisitedVertices()`, and `depthFirst()` pushes the adjacent vertex on the stack. The new vertex and its path are yielded to the caller for processing, and the `while` loop continues.

If Rule 1 doesn't apply, it now checks Rule 2: *if possible, pop a vertex off the stack*. At this point, the stack can't be empty because the `while` loop test checked that, and no vertices have been popped off since then. A simple `pop()` operation accomplishes Rule 2.

If neither Rule 1 nor Rule 2 applies, then you reach Rule 3: *you're done*. When the outer `while` loop finishes, all the reachable vertices have been pushed on the stack and then popped off, following the depth-first ordering.

You can test this code on a simple graph, as shown in Figure 14-7.

```
graph = Graph()
graph.addVertex(Vertex('A'))
graph.addVertex(Vertex('B'))
graph.addVertex(Vertex('C'))
graph.addVertex(Vertex('D'))
graph.addVertex(Vertex('E'))
graph.addVertex(Vertex('F'))
graph.addVertex(Vertex('G'))
graph.addVertex(Vertex('H'))
graph.addVertex(Vertex('I'))
graph.addEdge(0, 1)
graph.addEdge(0, 2)
graph.addEdge(0, 3)
graph.addEdge(0, 4)
graph.addEdge(1, 5)
graph.addEdge(5, 7)
graph.addEdge(3, 6)
graph.addEdge(6, 8)
print('Depth-first traversal')
for vert, path in graph.depthFirst(0):
   print(graph.getVertex(vert).name,
         end='')
print()
```



produces

```
Depth-first traversal
ABFHCDGIE
```

FIGURE 14-7 A test of `depthFirst()` on a small graph

Looking back on the choice to yield both the vertex and the path to that vertex in the `depthFirst()` generator, note the redundancy because the vertex is always the last element in the path. You could rewrite the generator to return only the path, shifting the responsibility of extracting the vertex to the caller. That's not much of a change in Python where the caller can access the last vertex using an O(1) reference to `path[-1]`. If the path is returned as a linked list, however, getting the last element takes O(N) time and merits the extra return value.

Depth-First Traversal and Game Simulations

When you're implementing game programs, depth-first traversal can simulate the sequence of moves made by players. In two-player board games like chess, checkers, and backgammon, each player chooses from among a set of possible actions on their turn. The actions at each turn depend on the state of the game, sometimes called the *board state*. For example, in the initial state of chess, the actions are limited to movement of the pawns or knights. Subsequent board states allow actions involving the other pieces.

This behavior can be modeled with a graph where the possible actions form edges between board states, which are the vertices in a graph. Starting from a particular board state, depth-first traversal enumerates all possible board states that could be reached from the initial state.

Let's look at how that works for a simple game of tic-tac-toe. The first player, let's say it is the X-player, can make one of nine possible moves. The O-player can counter with one of eight possible moves, and so on. Each move leads to another group of choices by your opponent, which leads to another series of choices for you, until the last square is filled. Figure 14-8 shows a partial graph of the board states. Only the edges connecting the topmost board state for each move are included, and symmetric board states are not drawn. That leaves only three distinct possible choices for the X-player's first move.

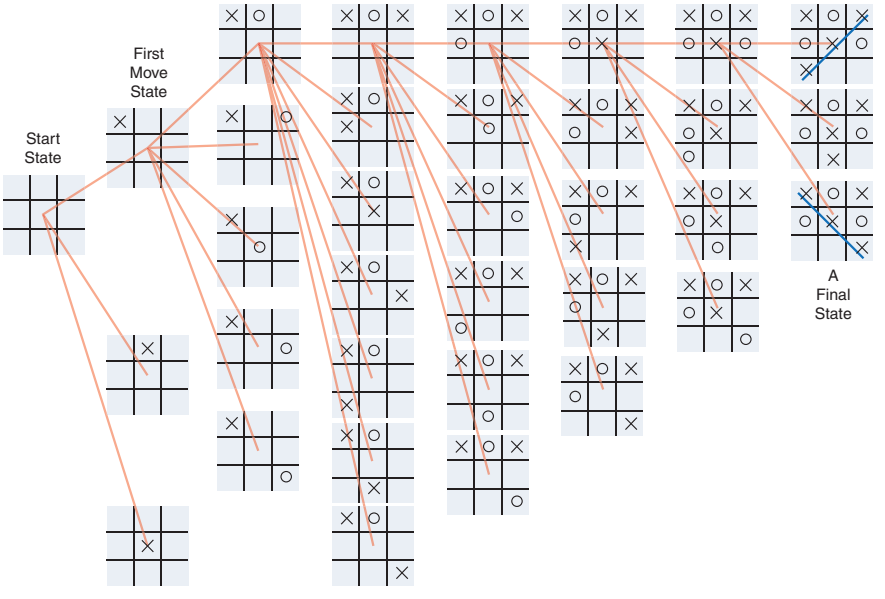


FIGURE 14-8 Tic-tac-toe board states as a graph

When you are deciding what move to make, one approach is to mentally imagine a move, then your opponent's possible responses, then your responses, and so on. You can decide what to do by seeing which move leads to the best outcome. In simple games like tic-tac-toe, the number of possible moves is sufficiently limited that it's possible to follow each path to the end of the game.

Analyzing these moves involves traversing the graph. As evident in Figure 14-8, even a "simple" game like tic-tac-toe can lead to a large graph. The number of edges for the first move, after eliminating symmetric moves, is three. The O-player's first move includes four

options. The next X-player move can have up to seven options after eliminating symmetry. The number of options diminishes after that.

Using depth-first traversal means the analysis drives toward the final states first. Any path that leads to an opponent's victory could be eliminated or, at least, set aside to be further explored as a last option. That would allow you to "prune" the graph significantly. How much could pruning save? If you don't pay attention to symmetry, there are nine possible first moves, followed by eight possible opponent moves, followed by seven possible first-player moves, and so on. That's 9×8×7×6×5×4×3×2×1 (9 factorial or 9! or 362,880) moves, ignoring the reduction due to games ending after one player completes three in a row. Those moves are edges in the graph, and that's a lot of edges to follow. Pruning could make a huge savings.

Although exploring 362,880 edges seems manageable for modern computers, other games have much larger graphs. Chess has 64 squares and 16 pieces for each player. The game of go has a 19-by-19 grid of points where players place either a black or white stone. Those 361 points mean there are something like 361! potential sequences of moves (and that number doesn't account for the removal and replacement of stones). Even though some of the sequences result in the same board state, exploring the full graph is quite daunting. Most game-playing algorithms only explore the graph to a particular depth and use many techniques to eliminate as many paths as possible in analyzing move options. They may never generate the complete graph.

Breadth-First

The depth-first traversal algorithm acts as though it wants to get as far away from the starting point as quickly as possible. In the breadth-first, on the other hand, the algorithm likes to stay as close as possible to the starting point. It first visits all the vertices adjacent to the starting vertex, and only then goes further afield. This kind of traversal is implemented using a queue instead of a stack.

An Example

Figure 14-9 shows the same graph as Figure 14-5, but this time we traverse the graph breadth-first. Again, the numbers indicate the order in which the vertices are visited. Like before, A is the starting vertex. You mark it as visited and place it in an empty queue. Then you follow these rules:

Rule 1

Take the first vertex in the queue (if there is one) and insert all its adjacent unvisited vertices into the queue, marking them as visited.

Rule 2

If you can't carry out Rule 1 because the queue is empty, you're done.

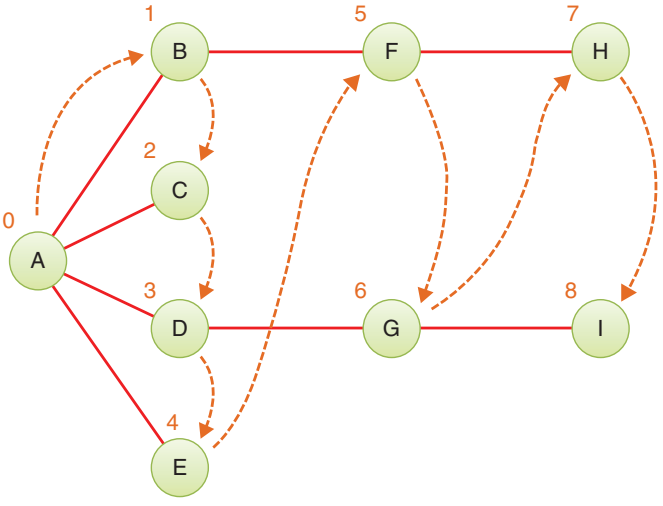


FIGURE 14-9 Breadth-first traversal example

The breadth-first traversal is slightly simpler than the depth-first traversal because there are only two rules. Walking through the example, you first visit A. Then you take all the vertices adjacent to A and insert each one into the queue as you visit and mark it. Now you've visited A, B, C, D, and E. At this point the queue (from front to rear) contains BCDE.

You apply Rule 1 again, removing B from the queue and looking for vertices adjacent to it. You find A and F, but A has been visited, so you visit and insert only F in the queue. Next, you remove C from the queue. It has no adjacent unvisited vertices, so nothing is visited or inserted in the queue. You remove D from the queue and find its neighbor G is unvisited, so you visit it and insert it in the queue. You remove E and find no unvisited neighbors.

At this point the queue contains FG, the only adjacent unvisited vertices found while previously visiting BCDE. You remove F and visit and insert H on the queue. Then you remove G and visit and insert I.

Now the queue contains HI. After you remove each of these and find no adjacent unvisited vertices, the queue is empty, so you're done. Table 14-4 shows this sequence.

At each moment, the queue contains the vertices that have been visited but whose neighbors have not yet been fully explored. (Contrast this breadth-first traversal with the depth-first traversal, where the contents of the stack hold the route you took from the starting point to the current vertex.) The vertices are visited by breadth-first in the order ABCDEFGHI.

Table 14-4 Queue Contents During Breadth-First Traversal

Event	Queue (Front to Rear)
Visit A	A
Remove A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Remove B	CDE
Visit F	CDEF
Remove C	DEF
Remove D	EF
Visit G	EFG
Remove E	FG
Remove F	G
Visit H	GH
Remove G	H
Visit I	HI
Remove H	I
Remove I	
Done	

The Graph Visualization Tool and Breadth-First Traversal

Use the Graph Visualization tool to try out a breadth-first traversal using the Breadth-First Traverse button. Again, you can experiment with the graph in Figure 14-9, or you can make up your own.

Notice the similarities and the differences of the breadth-first traversal compared with the depth-first traversal.

You can think of the breadth-first traversal as proceeding like ripples widening when you drop a stone in water or—for those of you who enjoy epidemiology—as the influenza virus carried by air travelers from city to city. First, all the vertices one edge away from the starting point (plane flight) are visited, then all the vertices two edges away are visited, and so on.

Python Code

The `breadthFirst()` method of the `Graph` class is like the `depthFirst()` method, except that it uses a queue instead of a stack and fully explores the sequence of adjacent unvisited vertices. The implementation for both the `Queue` class and the traversal generator is shown in Listing 14-5.

LISTING 14-5 The `breadthFirst()` Traversal Generator for a `Graph`

```
class Queue(list):          # Use list to define Queue class
   def insert(self, j): self.append(j) # insert == append
   def peek(self): return self[0] # First element is front of queue
   def remove(self): return self.pop(0) # Remove first element
   def isEmpty(self): return len(self) == 0

class Graph(object):
…
   def breadthFirst(        # Traverse the vertices in breadth-first
         self, n):          # order starting at vertex n
      self.validIndex(n)    # Check that vertex n is valid
      visited = [False] * self.nVertices() # Nothing visited initially
      queue = Queue()       # Start with an empty queue and
      queue.insert(n)       # insert the starting vertex index on it
      visited[n] = True     # and mark starting vertex as visited
      while not queue.isEmpty(): # Loop until nothing left on queue
         visit = queue.remove() # Visit vertex at front of queue
         yield visit        # Yield vertex to visit it
         for j in self.adjacentUnvisitedVertices( # Loop over adjacent
               visit, visited): # unvisited vertices
            queue.insert(j) # and insert them in the queue
```

Here, we define the `Queue` class as a subclass of `list`. Inserting an item in the queue uses the list's `append()` method. This means the back (or end) of the queue is at the highest index of the list. That means, `peek()` and `remove()` operate on the first element of the list at index 0. The Python `list.pop()` method takes an optional parameter for the index of the item to remove.

The beginning of the traversal starts off just as it did for depth-first, checking the validity of the starting vertex and creating a `visited` array for all the vertices. Then the `visited` array is created and seeded with the starting vertex index, and that index is marked as visited. The implementation in Listing 14-5 deviates slightly from the rules shown earlier in that the visit of the first vertex doesn't happen until inside the `while` loop where the rules are applied. Also note that marking vertices in the `visited` array happens before they are yielded to the caller. The overall `breadthFirst()` generator "visits" a vertex by yielding it to the caller for processing while keeping the internal `visited` array to track which vertices it has already put in its queue to process.

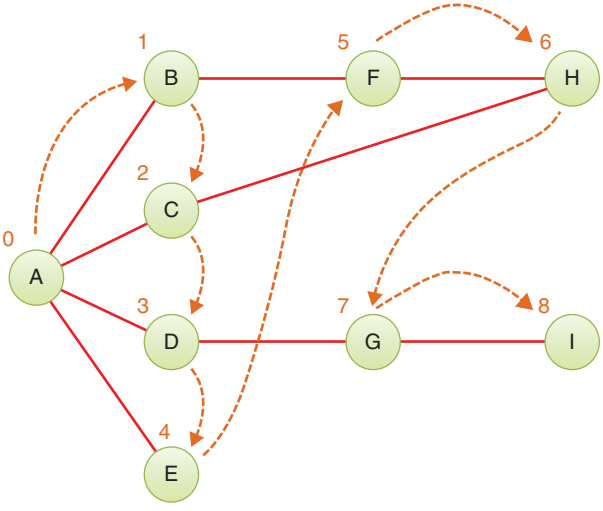
The `while` loop test determines whether Rule 1 or Rule 2 will apply. If it's Rule 1, the loop body removes a vertex from the front of the queue and immediately visits it by yielding

it. The method visits the starting vertex in the same way all the others are visited. Note that we no longer have the stack to provide the path taken to reach this vertex. As a consequence, the `breadthFirst()` generator yields only a vertex index. It's possible and useful to provide the path, and we leave that as a programming project.

After `breadthFirst()` removes a vertex from the queue and visits it, the next thing to do is insert all the adjacent unvisited vertices in the queue. That process is handled by looping through the `adjacentUnvisitedVertices()` generator (shown in Listing 14-3) and inserting the vertices in the `queue`. That completes the implementation of Rule 1. Rule 2 is also done because no more processing is needed when the queue is empty.

Let's test the `breadthFirst()` traversal on nearly the same graph used for depth-first, but let's add one edge between C and H to see what effect it has. Figure 14-10 shows the code used to construct the graph and traverse the vertices in breadth-first order. The extra edge is highlighted in the code. Note that the vertex ID numbers used to create the edges are different from the visit numbers next to the vertices in the figure.

```
graph = Graph()
graph.addVertex(Vertex('A'))
graph.addVertex(Vertex('B'))
graph.addVertex(Vertex('C'))
graph.addVertex(Vertex('D'))
graph.addVertex(Vertex('E'))
graph.addVertex(Vertex('F'))
graph.addVertex(Vertex('G'))
graph.addVertex(Vertex('H'))
graph.addVertex(Vertex('I'))
graph.addEdge(0, 1)
graph.addEdge(0, 2)
graph.addEdge(0, 3)
graph.addEdge(0, 4)
graph.addEdge(1, 5)
graph.addEdge(5, 7)
graph.addEdge(3, 6)
graph.addEdge(6, 8)
graph.addEdge(2, 7)
print('Breadth-first traversal')
for vert in graph.breadthFirst(0):
   print(graph.getVertex(vert).name,
         end='')
print()
```



produces

```
Breadth-first traversal
ABCDEFHGI
```

FIGURE 14-10 A test of `breadthFirst()` on a small graph

In the test example, after visiting ABCDE, the queue contains the unvisited adjacent vertices, FHG. Without the edge from C to H, the queue would have been FG, like in the graph of Figure 14-9. With the edge CH, H is marked as visited and inserted on the queue when C is removed. Then F is removed, and its neighbors searched for unvisited vertices. H has not yet been visited (yielded by the generator that starts at F), but it was marked as visited as it was added to the queue. So, H does not show up as a vertex to add to the queue. The next pass through the `while` loop removes H from the front of the queue and visits it. Because it, too, has no adjacent unvisited neighbors, nothing is put on the queue. The next pass removes G from the queue, finds I as unvisited, and inserts that final vertex.

This example illustrates an interesting property of breadth-first traversal and search: it first finds all the vertices that are one edge away from the starting point, then all the vertices that are two edges away, and so on. This capability is useful if you're trying to find the *shortest path* from the starting vertex to a given vertex (see Project 14.3). You start a breadth-first traversal, and when you visit a particular vertex, you know the path you've traced so far is the shortest path to it. If there were a shorter path, the breadth-first traversal would have visited it already. In Figure 14-10, there are two paths from A to H: ABFH and ACH. The fact that breadth-first visits H before G is due to having a two-edge path using the added edge. In Figure 14-9, the absence of edge CH means G is visited before H because both G and H can only be reached through three-edge paths.

Minimum Spanning Trees

Suppose you're designing a large apartment building and need to decide where the hot water pipes should go. The example shown in Figure 14-4 shows a completed plumbing network for a two-and-a-half-bathroom house, but let's imagine you have hundreds of sinks, bathtubs, showers, washing machines, and hot water heaters to connect. For the water supply, you could connect every fixture to every other one with a pipe. That would certainly provide the shortest possible path between every pair of fixtures. That would also cost a lot, and any plumber asked to do the installation would probably laugh.

After you get over your embarrassment, you realize that you only really need to connect every sink, bathtub, shower, and washing machine to a water heater. You want those pipes to be short so that the hot water can reach the faucets quickly and to minimize the length of pipe needing insulation. You can't run the pipes through the rooms of the building; they must be hidden inside the walls, floors, and ceilings. Not every wall will allow pipes to go through it, such as a wall made of glass. So, you must take some twisted paths to hook everything together. How do you find which paths to hook up so that all the fixtures are connected but the amount of pipe is the shortest?

It would be nice to have an algorithm that, for any connected set of fixtures and pipes (vertices and edges, in graph terminology), would find the minimum set of pipes needed to connect all the fixtures. Even if it didn't find the absolute minimum pipe length overall or the minimum total distance to the water heater, you wouldn't want any extra pipes connecting two fixtures if there was another path between them. For instance, you wouldn't want a loop of pipes (edges) like the path ABFHCA in Figure 14-10 or the fully

connected graph on the left of Figure 14-11 (although there are cases where plumbing loops are desirable). The result of such an algorithm would be a graph with the minimum number of edges necessary to connect the vertices. The graph on the left of Figure 14-11 has the maximum number of edges, 10, for a five-vertex graph. The graph on the right has the same five vertices but with the minimum number of edges necessary to connect them, four. This constitutes a **minimum spanning tree** *(MST)* for the graph.

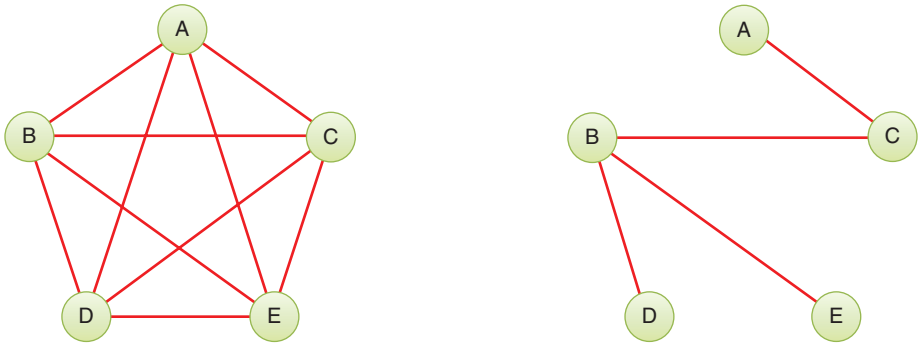


FIGURE 14-11 A fully connected graph and a minimum spanning tree

There are many possible minimum spanning trees for a given graph. The MST of Figure 14-11 shows edges AC, BC, BD, and BE, but edges AC, CE, ED, and DB would do just as well. The arithmetically inclined will note that the number of edges in a minimum spanning tree is always one fewer than the number of vertices. Removing any edge from a minimum spanning tree would create multiple connected components.

For now, don't worry about the length of the edges. You're not trying to find a minimum physical length, just the minimum number of edges. (Your plumber might have a different opinion.) We discuss more about this when we talk about weighted graphs in the next chapter.

The algorithm for creating the minimum spanning tree is almost identical to that used for traversing. It can be based on either the depth-first or the breadth-first traversal. This example uses the depth-first traversal.

Perhaps surprisingly, by executing the depth-first traversal and recording the edges you've traveled to reach a vertex, you automatically create a minimum spanning tree. It's also somewhat counterintuitive because breadth-first traversal finds the shortest path to each vertex. The only difference between the minimum spanning tree method, which you see in a moment, and the depth-first traversal, which you saw earlier, is that it must somehow record and return a tree-like structure.

Minimum Spanning Trees in the Graph Visualization Tool

The Graph Visualization tool runs the minimum spanning tree algorithm after you select a starting vertex and select the corresponding button. You see the `depthFirst()` generator run in the code window to yield results to a method we explore shortly.

As the depth-first traversal yields vertices and their paths from the starting vertex, the Visualization tool highlights them, as shown in Figure 14-12. In the example, the algorithm adds vertex E and its path from the starting vertex, C, to the tree it is building. The vertices that have been added have brown circles around them (B, C, D, and E) and the edges forming the path returned by `depthFirst()` are highlighted in blue. Edges added in earlier steps (for example, edge BD) are highlighted in brown.

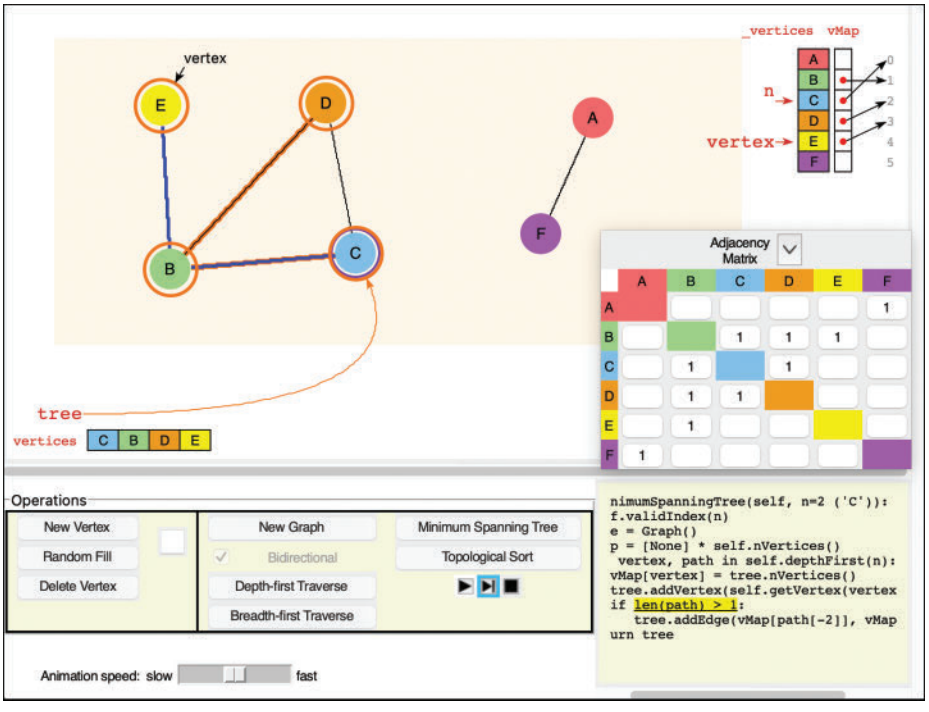


FIGURE 14-12 The Graph Visualization tool adding a path to a minimum spanning tree

As you can see, the minimum spanning tree implementation must track many things as it runs. We explore the detailed implementation in Python, but first we need to revisit a structure that we discussed in detail in several preceding chapters: trees.

Trees Within a Graph

What does a minimum spanning tree "look" like? Is it a binary tree? No, because the number of vertices connected to a given vertex could be one, two, three, and so on. For example, in the minimum spanning tree on the right of Figure 14-13, there is no way to arrange the vertices and edges into a binary tree form. Even though the trees for both Figure 14-11 and Figure 14-13 come from the same starting graph, it's not possible to know whether the resulting tree will have edges that could form a binary tree.

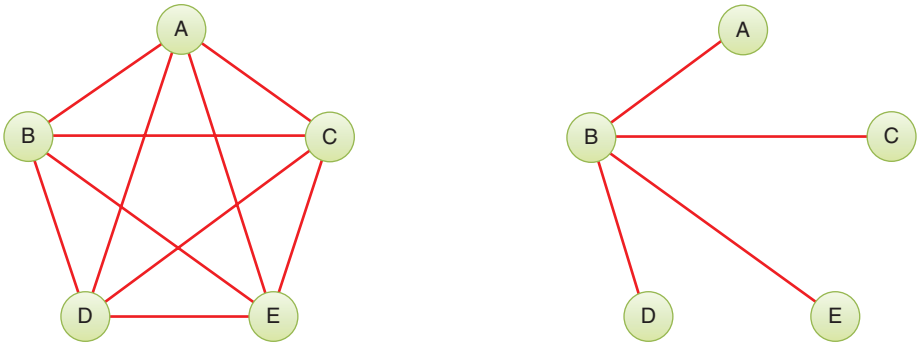


FIGURE 14-13 Another minimum spanning tree from a fully connected five-vertex graph

So, what structure should be used to represent the minimum spanning tree? You could try to make a tree structure with an arbitrary number of children for each node, something like the 2-3-4 tree but with any number of children. That could be a good solution, but there's an easier one: use the `Graph` class itself.

A minimum spanning tree is a **subgraph**—a subset of the original graph's vertices and edges. The *spanning* part means that all the vertices in a connected component are still connected, and the *tree* means there is a single, unique path between every pair of vertices. The trees you've seen so far all share those properties. There was a unique path from the root to every node in the tree, and all of them were reachable from the root.

It's possible for the minimum spanning tree to include every vertex and edge of the input graph. Typically, however, the tree is a proper subset of the vertices and edges forming a connected component.

Python Code for the Minimum Spanning Tree

Our `minimumSpanningTree()` method returns a new graph with a subset of the vertices and edges in the starting graph. We create the MST using a depth-first traversal from a starting vertex. All the vertices in the connected component that includes the starting vertex will go in the MST. Any nonconnected vertices, like A and F in Figure 14-12, are left out. Listing 14-6 shows the code.

Because the result contains a subset of the original vertices, the vertex indices could be different in the two graphs. For instance, a 10-vertex graph might be composed of two connected components: one with six vertices and the other with four. If you ask for an MST starting from a vertex in the six-vertex component, the MST will have exactly six vertices with indices 0 through 5. Those vertices could have indices up to 9 in the original graph, so you need a way to track the correspondence of vertices in the original graph to those in the MST.

LISTING 14-6 The `minimumSpanningTree()` Method of `Graph`

```
class Graph(object):
…
   def minimumSpanningTree( # Compute a minimum spanning tree
         self, n):          # starting at vertex n
      self.validIndex(n)    # Check that vertex n is valid
      tree = Graph()        # Initial MST is an empty graph
      vMap = [None] * self.nVertices() # Array to map vertex indices
      for vertex, path in self.depthFirst(n):
         vMap[vertex] = tree.nVertices() # DF visited vertex will be
         tree.addVertex(    # last vertex in MST as we add it
            self.getVertex(vertex))
         if len(path) > 1:  # If the path has more than one vertex,
            tree.addEdge(   # add last edge in path to MST, mapping
               vMap[path[-2]], vMap[path[-1]]) # vertex indices
      return tree
```

The `minimumSpanningTree()` method starts with an empty graph—called `tree`—to hold the MST. As we add vertices to it, we note the translation between old and new vertices in an array called `vMap`. The array needs a cell for every possible vertex. Just before we add a vertex to the MST, we know what index it will get because it's placed at the end of the existing vertices. That means the old vertex index maps to the number of the vertices added to the MST so far.

To see how the mapping works, look at the sample `vMap` in Figure 14-12. The `vMap` appears in the upper right of the display as an array aligned with input vertices whose values point to indices in the new tree (vertex indices). As the starting point, vertex C was visited first and becomes the first vertex in the output tree, so the `vMap` shows C mapping to index 0. The next vertex added to the tree is vertex B, so it maps to index 1, and so on. Only the vertices included in the minimum spanning tree will have entries in `vMap` mapping them to their new indices. In Figure 14-12, vertices A and F are not in `vMap` because they are not connected to C.

In the code after setting up the empty `tree` and `vMap`, `minimumSpanningTree()` uses the `depthFirst()` traversal over the graph to visit all the vertices in the connected component. The `depthFirst()` traversal yields both a vertex and the path to the vertex for each visit (in the form of a stack). The first vertex visited will always be the starting vertex, and the path will contain just that one vertex on the visit.

In the depth-first loop body, we first store the mapping from the `vertex` being visited to its index in the MST. The number of vertices already in the `tree` provides the index to store in `vMap`. The next call adds the `vertex` being visited to the MST. Then, if the path to the visited vertex has at least one edge in it, we add the last edge to the MST as well. We get the vertices forming that edge from the last two vertices in the path: `path[-2]` and `path[-1]`. We must translate those vertex indices using the `vMap` array. We're guaranteed that the indices in the path were set in `vMap` because the path contains only vertices that were previously visited by `depthFirst()`.

Does this code handle everything? What about the edges? We add one edge for each vertex added to the MST other than the starting vertex. Is that enough? Yes, adding one edge per vertex means we get exactly the N–1 edges from the N vertex connected component. Each added edge is the last edge in the unique path that leads to that vertex. So, the algorithm covers all the vertices and edges of the connected component that belongs in the MST.

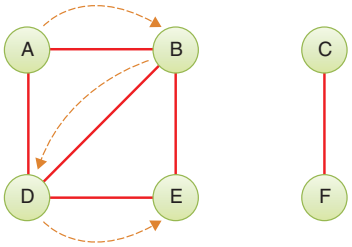
When the depth-first traversal is complete, `tree` contains the full minimum spanning tree, so it is returned to the caller. To help with development and see the detailed contents of the various arrays, you can define some print methods for the `Graph` class, as shown in Listing 14-7.

LISTING 14-7 Methods for Summarizing and Printing `Graphs`

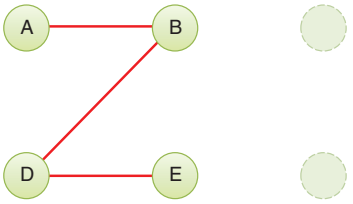
```
class Graph(object):
…
   def __str__(self):       # Summarize the graph in a string
      nVertices = self.nVertices()
      nEdges = self.nEdges()
      return '<Graph of {} vert{} and {} edge{}>'.format(
         nVertices, 'ex' if nVertices == 1 else 'ices',
         nEdges, '' if nEdges == 1 else 's')

   def print(self,          # Print all the graph's vertices and edges
             prefix=''):    # Prefix each line with the given string
      print('{}{}'.format(prefix, self)) # Print summary form of graph
      for vertex in self.vertices(): # Loop over all vertex indices
         print('{}{}:'.format(prefix, vertex), # Print vertex index
               self.getVertex(vertex)) # and string form of vertex
         for k in range(vertex + 1, self.nVertices()): # Loop over
            if self.hasEdge(vertex, k): # higher vertex indices, if
               print(prefix, # there's an edge to it, print edge
                     self._vertices[vertex].name,
                     '<->',
                     self._vertices[k].name)
```

These printing methods use some of Python's string formatting features, which we don't describe here. Instead, we show what they produce on a small example.

For this example, we use a small graph with six vertices, two of which are not connected to the other four. Figure 14-14 shows the original graph on the left along with the depth-first traversal starting at vertex A. The minimum spanning tree is on the right. Vertices C and F do not appear in the minimum spanning tree because they are not connected by an edge to the other vertices. The output of the `print()` method appears below each graph. Note that edges are printed once next to their "first" (lower index) vertex and not repeated next to their "second" (higher index) vertex.




```
Initial graph
<Graph of 6 vertices and 6 edges>
0: <Vertex A>
 A <-> B
 A <-> D
1: <Vertex B>
 B <-> D
 B <-> E
2: <Vertex C>
 C <-> F
3: <Vertex D>
 D <-> E
4: <Vertex E>
5: <Vertex F>
```




```
Minimum spanning tree starting at 0
<Graph of 4 vertices and 3 edges>
0: <Vertex A>
 A <-> B
1: <Vertex B>
 B <-> D
2: <Vertex D>
 D <-> E
3: <Vertex E>
```

FIGURE 14-14 Printed descriptions of a graph and a minimum spanning tree

The minimum spanning tree algorithm follows the depth-first traversal to get all the vertices in the connected component of the starting vertex. The vertices of other components are ignored. The resulting subgraph will always have N–1 edges for the N vertices in the connected component.

How much time does it take to build the minimum spanning tree? Well, it's at least O(N) to go through all the N vertices. The `depthFirst()` traversal method seems as though it should be O(N), as was the case for all the traversal algorithms you've seen for other data structures. A closer inspection, however, shows that for each vertex visited, the `depthFirst()` generator starts up its own internal loop over all vertices to find the first adjacent unvisited one. If there are few edges, that inner loop could go through most of the vertices before finding the next adjacent one. That means it could take O(N) just to find the next adjacent vertex, and `depthFirst()` traversal could take $O(N^2)$. The breadth-first traversal always completes its inner loop, so it's definitely $O(N^2)$. That's not always going to happen for depth-first, as we discuss in the next chapter, but for now you can assume that worst case and that means the minimum spanning tree takes $O(N^2)$ too.

Topological Sorting

Topological sorting is another operation that can be modeled with graphs. It's useful in situations where items or events must be arranged in a specific order. Let's look at an example.

Dependency Relationships

In school, students find (sometimes to their dismay) that they can't take just any course they want. Some courses have prerequisites—other courses that must be taken first. The course sequence usually models a hierarchy where some concepts cannot be mastered without first understanding another set of concepts. These are **dependency relationships**. They occur everywhere. Algebra depends on understanding arithmetic; linear algebra depends on understanding algebra. In programming, a source file can import or include other source files, which makes one file dependent on another. In cooking, each step depends on completing the previous preparations. For example, baking bread depends on mixing the dough and heating an oven.

Dependency relationships can be described with a graph—specifically a **directed graph**. Figure 14-15 shows a simplified recipe for baking bread.

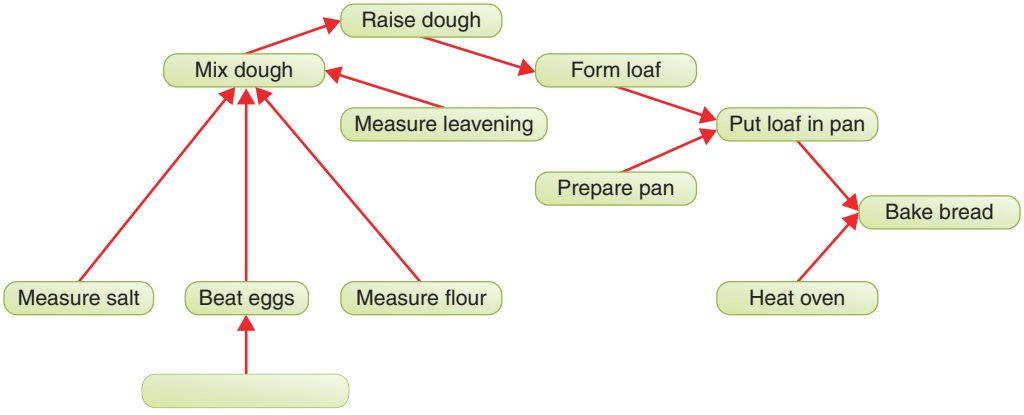


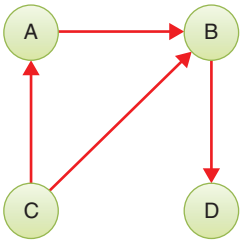
FIGURE 14-15 A dependency graph for baking bread

To bake bread, you need to have a heated oven and a loaf of dough in a pan. To have a loaf, it should be shaped from dough. That dough should be raised first, and so on. Most of the steps must be done in a particular order. Removing the eggshells after beating the eggs would be problematic (removing them after raising the dough would be even worse). Some pairs of steps, however, can be done in either order. It doesn't matter which order you measure the flour and the salt; and heating the oven and mixing the dough can be done in either order, even simultaneously. The dependency relations shown by the arrows capture this partial ordering.

Analyzing the different tasks and their relative order may seem trivial to anyone who has baked bread before, but they are anything but trivial to a robot. Imagine a robot that could a read a cookbook recipe and figure out all the steps. Assuming it could understand the meaning of the words, it would still need to determine the ordering of tasks. Recipes are usually written in chronological order of the steps, so it could simply follow that sequence. That, however, would probably lengthen the preparation time because steps that could be done at the same time would be done sequentially. Ideally, it would analyze all the possible orderings of the steps and find the most efficient one.

Directed Graphs

Dependency relationships expose a graph feature we haven't discussed yet: the edges need to have a *direction*. When this is the case, the graph is called a *directed graph* or a **digraph**. In a directed graph you can proceed only one way along an edge. The arrows in Figure 14-16 show the direction of the edges.



	A	B	C	D
A	0	1	0	0
B	0	0	0	1
C	1	1	0	0
D	0	0	0	0

FIGURE 14-16 A small directed graph and its adjacency matrix

In a program, the difference between an undirected graph and a directed graph is that an edge in a directed graph has only one entry in the adjacency matrix. Figure 14-16 shows a small directed graph and its adjacency matrix.

Each 1 in the matrix represents a single edge. The row labels show where the edge starts, and the column labels show where it ends. Thus, the edge from A to B is represented by a single 1 at row A column B. If the directed edge were reversed so that it went from B to A, there would be a 1 at row B column A instead.

For an undirected graph, as noted earlier, half of the adjacency matrix mirrors the other half, so half the cells are redundant. For directed graphs, however, every cell in the adjacency matrix conveys unique information. The halves are not mirror images.

For a directed graph, the method that adds an edge thus needs only a single statement

```
self._adjMat[A, B] = 1
```

instead of the two statements required in an undirected graph. If you use the adjacency-list approach to represent your graph, then A has B in its list, but—unlike an undirected graph—B does not have A in its list.

Sorting Directed Graphs

Imagine that you make a list of all the actions needed to bake bread, using Figure 14-15 as your input data. You then arrange the actions in the order you need to take them. The final Bake bread step (Bb) is the last item on the list, which might look like this using two initials for each vertex:

Mf,Ms,Ml,Re,Be,Md,Rd,Fl,Pp,Pl,Ho,Bb

Arranged this way, the graph vertices are said to be **topologically sorted**. Any action you must take before some other action occurs before it in the list.

Many possible orderings would satisfy the dependency relationships. You could heat the oven and prepare the pan first as in

Ho,Pp,Mf,Ms,Ml,Re,Be,Md,Rd,Fl,Pl,Bb

This approach also satisfies all the relationships (although the similarity to chemical symbols might be a bit unsettling when thinking about bread). There are many other possible orderings as well. When you use an algorithm to generate a topological sort, the approach you take determines which of various valid sortings are generated. The specific constraint that must be followed in graph terminology is as follows. For it to be a valid sorting of a directed graph, if there is a path from vertex A to vertex B, vertex A must precede vertex B in the topological sort.

Directed graphs can model other situations besides recipe steps and course prerequisites. Many industrial projects are managed by breaking down the overall project into smaller jobs or tasks. Each task might depend on outputs of other tasks and might require some time to elapse between the end of one task and the start of another. Take building a house as an example. The person who wants the house constructed must identify a site, find a builder, obtain permits, and get financing. The actual construction work depends on having all those tasks done. There could be a delay between completing all the preparation tasks and commencing construction as the builder finds crewmembers available to start the new work. These delays are somewhat predictable, but rarely with any precision.

Modeling job schedules with graphs is called **critical path analysis**. Although we don't show it here, a weighted, directed graph (discussed in the next chapter) can be used, which allows the graph to include the time necessary to complete different tasks in a project. The graph can then tell you such things as the minimum time necessary to complete the entire project and overall costs by looking at different possible topological sorts.

The Graph Visualization Tool

The Graph Visualization tool can model directed graphs too. When the graph has no edges, you can select or clear the Bidirectional checkbox. The checkbox is disabled when any edges exist. You can select New Graph to clear all the vertices and edges and uncheck the Bidirectional box.

The vertices and edges are edited in the same way for directed graphs. The differences are that edges are drawn as a curve with arrow heads; dragging the pointer from one vertex to another only creates the edge from the first one to the second, and clicking a cell in the adjacency matrix only creates the edge from the corresponding row to the corresponding column. Figure 14-17 shows the same directed graph as the bread baking example from Figure 14-15 using the two-letter abbreviations for the operations. The vertices have been rearranged but the edges connecting them remain the same. The adjacency matrix is large and obscures the table of vertices. Use the collapse and uncollapse button to switch the view of the matrix.

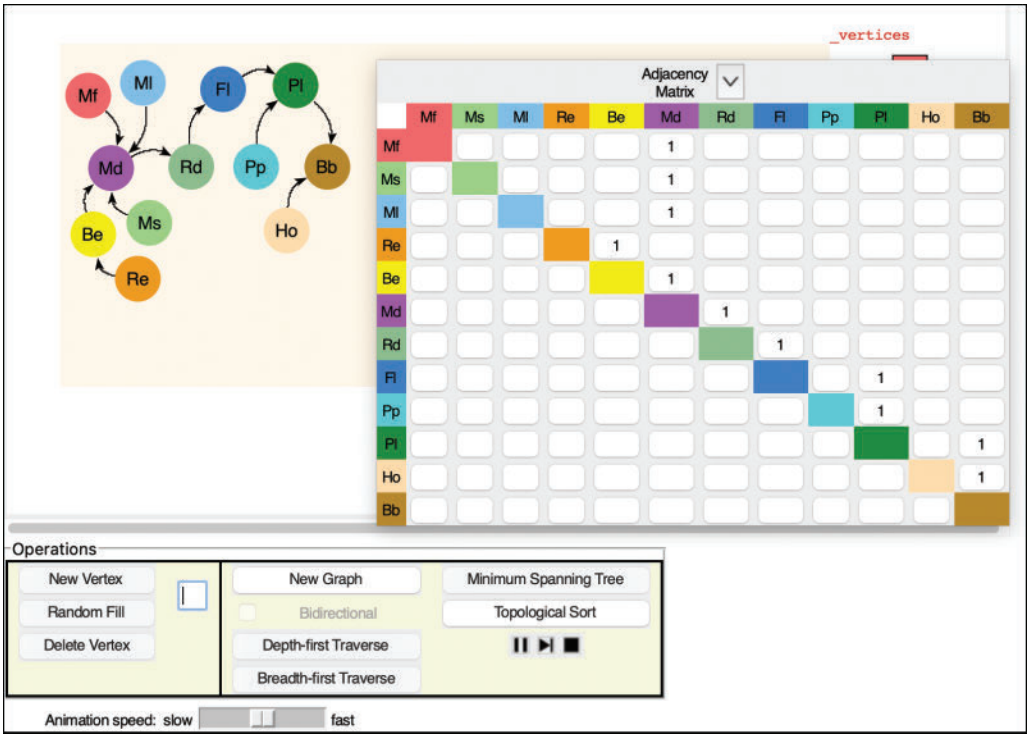


FIGURE 14-17 A directed graph in the Graph Visualization tool.

You can use the tool to run depth and breadth-first traversals and minimum spanning trees of directed graphs. The results are different than for the undirected graphs because the edges are now directional.

The Topological Sorting Algorithm

Let's look at how to sort the vertices topologically. Here, we start with a basic algorithm and then improve it. For the basic algorithm, we need at least two steps.

Step 1

Find an unvisited vertex that has no (unvisited) predecessors.

The *predecessors* to a vertex, *V*, are those vertices that are directly "upstream" from it—that is, connected to it by an edge that points to *V*. If an edge points from A to B, then A is a predecessor to B (and B is a *successor* to A). In Figure 14-16, the only vertex with no predecessors is C. In the bread baking example of Figure 14-17, six vertices have no predecessors: Mf, Ms, Ml, Re, Pp, and Ho. These vertices can be found by looking for columns in the adjacency matrix with no edges.

Step 2

Mark the vertex found in Step 1 as visited and add it at the end of the result list.

Steps 1 and 2 are repeated until all the vertices are gone. At this point, the result list holds the vertices arranged in topological order.

You can see the process at work by using the Graph Visualization tool. Construct the graph in Figure 14-16 (or any other graph, if you prefer). Then select the Topological Sort button. There's no need to select a starting vertex because the algorithm doesn't need one.

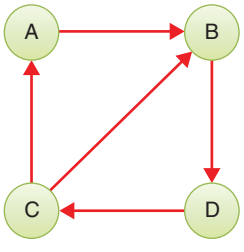
The animation in the Visualization tool proceeds in two phases. It's a bit fancier than the two-step algorithm shown earlier, but those same steps are used. We look at a couple of different implementations later to see how to sort the vertices efficiently.

Note that this topological sorting algorithm works on unconnected graphs as well as connected graphs. This models the situation in which you have two unrelated goals, such as getting a degree in mathematics and at the same time taking a hiking trip in the mountains. Some graphs, however, cause problems.

Cycles and Trees

One kind of graph that this basic algorithm cannot handle is a graph with **cycles**. We mentioned how the minimum spanning tree of a graph never creates a loop connecting a vertex to itself by a path of two or more edges. The formal name for such a loop is a *cycle*.

If a path ends up at the vertex where it started by following directed edges, the graph has at least one cycle. In Figure 14-18 the path BDCB forms a cycle. In addition, ABDCA forms a cycle. Notice that ABCA is not a cycle because you can't go from B to C in this directed graph.



	A	B	C	D
A	0	1	0	0
B	0	0	0	1
C	1	1	0	0
D	0	0	1	0

FIGURE 14-18 Directed graph with cycles.

A cycle models the paradox (which some students claim to have encountered at certain institutions), in which course B is a prerequisite for course D, D is a prerequisite for C, and C is a prerequisite for B. Fortunately, such paradoxes are rare.

Cycles can occur in undirected graphs too, but you must be more specific about them. You can typically ignore the cycle formed by following an edge and returning immediately back along the same edge. Cycles must be formed by paths of two or more distinct undirected edges. Multigraphs allow a pair of vertices to be connected by multiple edges,

so a cycle could form with as few as two edges. In non-multigraphs, you need at least three undirected edges to form a cycle.

It's easy to figure out whether some kinds of undirected graphs have cycles. If a graph with N vertices has more than N–1 edges, it must have cycles. You can make this point clear to yourself by trying to draw a graph with N vertices and N edges that does not have any cycles. On the other hand, detecting cycles in directed graphs and graphs with fewer edges can be hard to do. In Chapter 15, "Weighted Graphs," the section on efficiency discusses the complexity of finding certain kinds of cycles.

An undirected graph with no cycles is always a tree. It's the presence of cycles that allows two vertices to be connected by distinct paths.

The binary and multiway trees you saw in earlier chapters of this book are types of directed graphs because the edges always link a parent node to a child node. We never allowed those trees to create cycles. Doing so would cause problems like those described for circular lists in Chapter 5.

Topological sorts succeed only on directed graphs with no cycles. Such a graph is called a **directed acyclic graph**, often abbreviated *DAG*. The basic topological sorting algorithm doesn't have a step for what to do if Step 1 fails—when there is no vertex that has no unvisited predecessors. For example, if you ask it to process the graph in Figure 14-18, Step 1 will see that all vertices have at least one predecessor and fail. Thus, we need to add a final step:

> ***Step 3***
>
> After Step 1 fails, if the result list has fewer vertices than the graph, the graph must have a cycle.

This three-step algorithm can handle all directed graphs with and without cycles.

Python Code for the Basic Topological Sort

We introduce some new methods in the `Graph` class to work on predecessors and use them to implement the three steps of the basic topological sorting algorithm in Listing 14-8. The first method, `predecessorVertices()`, is a generator like `adjacentVertices()`, which yields all the predecessor vertices of a given vertex, `n`. It validates the vertex index and then loops over all possible indices, yielding only those with an edge pointing to `n`.

To implement Step 1, we need a test to find an unvisited vertex with no unvisited predecessors. In other words, the algorithm seeks a vertex where all its predecessors, if any, are among the visited vertices. The `onlyVisitedPredecessors()` method takes a vertex index, `n`, as a parameter and uses Python's `all()` function to test whether all of `n`'s predecessors satisfy the condition. The arguments to the `all()` function are the values of the `visited` array for each predecessor index. If one or more of those predecessors have a `visited` value of `None` or `False`, then vertex `n` fails the test. Notably, if `n` has no predecessors, the argument list to the `all()` function is empty and it returns `True`.

LISTING 14-8 Code for the Basic Topological Sort of Vertices

```
class Graph(object):
…
   def predecessorVertices( # Generate a sequence of vertex indices
         self, n):          # that are adjacent predecessors to n
      self.validIndex(n)    # Check that vertex index n is valid
      for j in self.vertices(): # Loop over all other vertices
         if j != n and self.hasEdge(j, n): # If other vertex connects
            yield j         # via edge, yield other vertex index

   def onlyVisitedPredecessors( # Test whether vertex n's predecessors
         self, n, visited): # have all been visited, if any
      return all(visited[j] # All predecessors must have been set in
                 for j in self.predecessorVertices(n)) # visited array

   def findUnvisitedWithoutPredecessor( # Find a vertex without
         self, visited):    # unvisited predecessor vertices, if any
      for vertex in self.vertices(): # Loop over all vertices
         if (not visited[vertex] and # If vertex is unvisited and has
             self.onlyVisitedPredecessors( # only visited
                vertex, visited)): # predecessors,
            return vertex   # then return it
      return None           # Otherwise there's a cycle or no vertices

   def sortVerticesTopologically( # Return a sequence of all vertex
         self):             # indices sorted topologically
      result = []           # Result list of vertices
      nVertices = self.nVertices() # Number of vertices
      visited = [None] * nVertices # Array to mark visited vertices
      while len(result) < nVertices: # Loop until all vertices handled
         vertex = self.findUnvisitedWithoutPredecessor( # Find an
               visited)     # unvisited vertex without predecessors
         if vertex is None: # If no such vertex exists, then raise an
            raise Exception('Cycle in graph, cannot sort') # exception
         result.append(vertex) # Append unvisited vertex and
         visited[vertex] = True # mark it as visited
      return result
```

The `findUnvisitedWithoutPredecessor()` method performs Step 1, looping over all possible vertices, skipping visited vertices, and returning the first unvisited one that satisfies `onlyVisitedPredecessors()`. If it completes the loop without finding an unvisited vertex without predecessors, it returns `None`, signaling that there must be a cycle.

The `sortVerticesTopologically()` method performs Steps 1, 2, and 3. It sets up an empty `result` list to hold the sorted vertices. Next it creates a `visited` array, like the ones used

in the traversal methods, marking all the vertices as unvisited. The main work begins with the `while` loop. On each pass of the loop, it tries Step 1 to find a `vertex` with no predecessors. If that comes back as `None`, it raises an exception to signal the detection of a cycle. Otherwise, it adds the vertex to the end of the `result` and marks it as `visited`, which implements Step 2. If the loop finishes, then there are `nVertices` in the `result` list, and it is returned, handling Step 3.

The basic topological sort algorithm is easy to understand, but let's look at its efficiency. Figure 14-19 shows the different levels of the algorithm. Each level represents one of the methods being called. Let's look at the innermost level first. Every loop through the `vertices()` sequence is, of course, O(N). Thus, one call to the `onlyVisitedPredecessors()` test for a particular vertex looks at all the other vertices, so it takes O(N) time too. At the next level, outward, the `findUnvisitedWithoutPredecessor()` method calls the inner test on every vertex until it finds one that matches. Although that middle level does quit when it finds such a vertex, it still has to scan some fraction of the vertices, so the combined worst-case complexity of the middle level is $O(N^2)$ (to be thorough, we would need to determine the average number of tries needed to find the vertex, but we can assume it's proportional to N without knowing anything about the graph's edges).

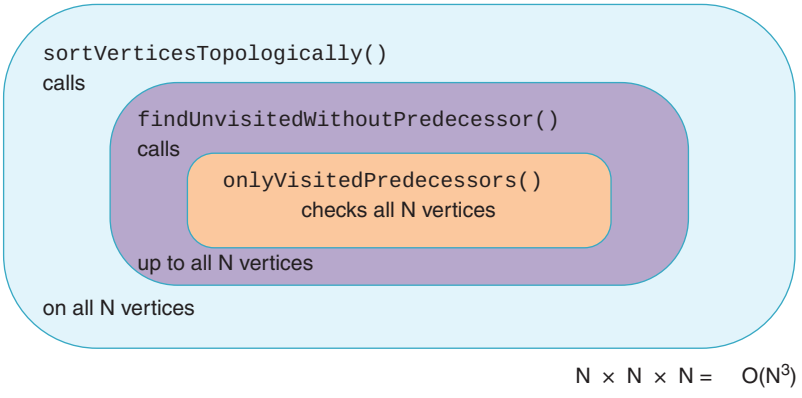


FIGURE 14-19 Complexity of the basic topological sort algorithm

The outermost routine, `sortVerticesTopologically()`, calls the middle level until all `nVerticies` have been copied. That's another O(N) factor, so each level of the algorithm contributes O(N) as shown. Overall, the algorithm is $O(N^3)$. Can we do better than that?

Improving the Topological Sort

Yes, we can reduce that complexity to at least $O(N^2)$, but it will take a more complicated organization of the data. While this transformation is complicated, learning from examples of how others have improved algorithm performance will help you analyze and improve your own programs. It's another case study in how to select and use the data structures most appropriate for a task.

The key concept that we're interested in is the number of edges for each vertex. If we organize the vertices by the number of inbound edges, we can quickly find the ones with no predecessors. In graphs, the number of inbound and outbound edges are called the **inbound degree** and the **outbound degree**, or **indegree** and **outdegree**, for short.

The first thing we need is a method to get the inbound and outbound degree of a vertex. The `degree()` method in Listing 14-9 takes a vertex as a parameter, counts both kinds of edges at that vertex, and returns the counts as a tuple.

After we know the inbound degree for all the vertices, we can put all the vertices with degree 0 in one place, so they are easy to find. In fact, if we keep an array indexed by degree, we can put each vertex in the cell of the array for its inbound degree. We need some structure in each array cell that can hold a group of vertices. Should that structure be a stack? A queue? A tree? The answer depends on how we process the vertices, so let's look at the remaining steps.

After we take a vertex out of the cell for degree 0, we can put it in the sorted result list. The successors of that vertex now have one fewer inbound edge, effectively. A successor that had only one inbound edge from the vertex that was put in the result list can now be considered to have degree 0. So instead of tracking which vertices have been visited, we can simply change the degree for each successor vertex, moving it to the proper cell in the array for its new degree. As we process vertices and their successors, every vertex will eventually be moved down to degree 0 (if there are no cycles).

Knowing that we want to move vertices between cells in the array, it becomes clear that we need a structure that enables fast insertion and deletion. Stacks and queues have O(1) insert and delete complexity, but only for specific orders (LIFO and FIFO). For this algorithm we would need to, say, remove vertex 27 from one cell and add it to another.

If we want to insert and delete items by a key, such as their vertex index, the best data structures would be an array or a hash table. Both have O(1) insert and delete times (assuming we don't need to shift items in the array to eliminate holes). The hash table has the added advantage of only needing memory proportional to the items in it; an array would need N cells to hold any of the N vertices.

Figure 14-20 illustrates the relations of structures we need. The left side shows a simple four-vertex graph. Next on its right is the `Graph` object that holds the `_vertices` array and `_adjMat` adjacency matrix to represent it. Next to that is a four-element array that shows the inbound degree for each of the four vertices. For example, vertex A has inbound degree 1 while vertex B has inbound degree 2.

The `vertsByDegree` array groups together the vertices by their inbound degree. Each cell in this array references a hash table that holds the keys for the vertices with a particular inbound degree. For example, `vertsByDegree[1]` references a hash table holding two keys. The keys are the vertex indices of vertices of inbound degree one: 0 (A) and 3 (D). The hash table for degree three is empty because none of the vertices have inbound degree three. The hash tables have extra cells to maintain a 50 percent load factor.

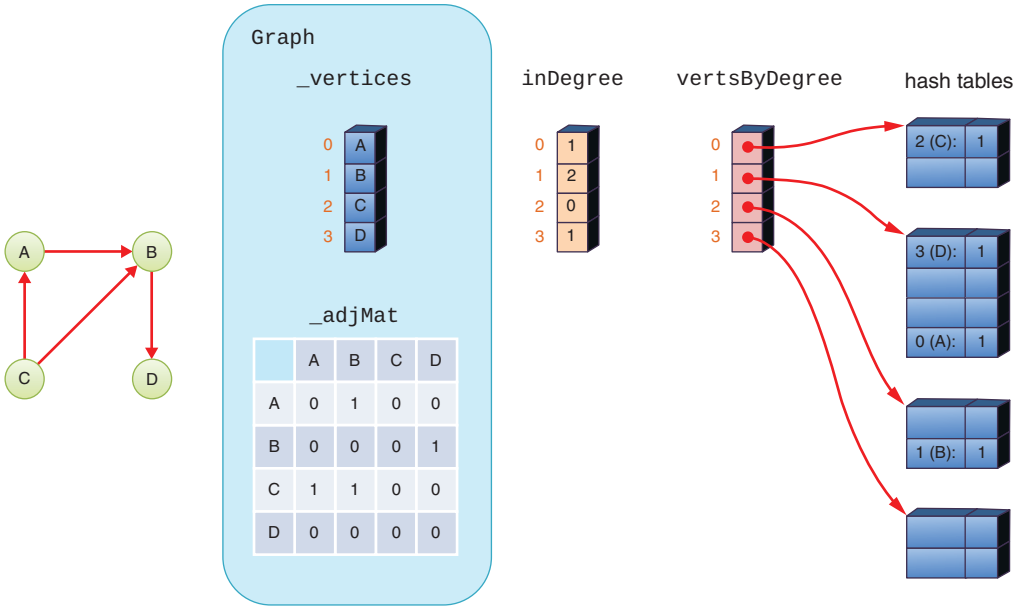


FIGURE 14-20 Structures used for topological sort

The `sortVertsTopologically()` method in Listing 14-9 starts by creating the `vertsByDegree` array. The maximum inbound degree any vertex could ever have is the number of vertices minus one. For a graph with E edges, the highest inbound edge count could be E. To account for vertices with indegree 0, we need to add 1 to E. The size of the array is the minimum of these values: `nVertices` and `nEdges+1`. The `vertsByDegree` array is built using a list comprehension `[{} for j in range(min(self.nVertices(), self.nEdges() + 1))]`. As you saw for building the adjacency matrix, using the list comprehension ensures that N distinct empty hash tables are made to fill the array cells.

The sorting algorithm creates an `inDegree` array to hold the inbound degree of every vertex. Initially, these are filled with 0s.

Now we can populate the new data structures. The next section is a `for` loop over all the vertices. It calculates the degree of the `vertex` by calling `self.degree()` and extracts the first element of the returned tuple using `[0]` to get the inbound degree.

The sorting algorithm now inserts the `vertex` in the `vertsByDegree` array based on its inbound degree. It finds the appropriate hash table by referencing `vertsByDegree[inDegree[vertex]]`. This kind of nested reference looks a little complicated, but it really is only two array lookups. The `vertex` variable indexes the `inDegree` array to get a numeric degree for the vertex. That numeric degree indexes the `vertsByDegree` array. The content of that array cell is a hash table. To enter a `key` in a hash table, `ht`, you could write `ht[key] = 1`. That's what's being done by writing `vertsByDegree[inDegree[vertex]][vertex] = 1`; it sets the `vertex` key in the hash table retrieved from `vertsByDegree`. These strings of references can be tricky to understand when you first look; try working from inner to outer references as well as outer to inner.

LISTING 14-9 The Improved Topological Sort of Vertices

```
class Graph(object):
…
   def degree(self, n):     # Get degree of vertex as (in, out) pair
      self.validIndex(n)    # Validate vertex index
      inb, outb = 0, 0      # Count inbound and outbound edges
      for j in self.vertices(): # Loop over all vertices
         if j != n:         # other than target vertex
            if self.hasEdge(j, n): # If other vertex precedes
               inb += 1            # increase inbound degree
            if self.hasEdge(n, j): # If other vertex succeeds n
               outb += 1           # increase outbound degree
      return (inb, outb)    # Return inbound and outbound degree

   def sortVertsTopologically( # Return sequence of all vertex indices
         self):             # sorted topologically more efficiently
      vertsByDegree = [     # Make an empty hash table for every
         {} for j in range( # possible degree, max = nVerts – 1 or
            min(self.nVertices(), self.nEdges() + 1))] # nEdges
      inDegree = [0] * self.nVertices() # Allocate indegree array
      for vertex in self.vertices(): # Loop over all vertices, record
         inDegree[vertex] = self.degree(vertex)[0] # inbound degree
         vertsByDegree[     # In hash table for this inbound degree
            inDegree[vertex]][vertex] = 1 # insert vertex
      result = []           # Result list is initially empty
      while len(            # While there are vertices with inbound
            vertsByDegree[0]) > 0: # degree of 0
         vertex, _ = vertsByDegree[0].popitem() # take vertex out of
         result.append(vertex) # hash table & add it to end of result
         for s in self.adjacentVertices( # Loop over vertex's
               vertex):     # successors; move them to lower degree
            vertsByDegree[ # In hash table holding successor vertex
               inDegree[s]].pop(s) # delete the successor
            inDegree[s] -= 1 # Decrease inbound degree of successor
            vertsByDegree[ # In hash table for lowered inbound degree
               inDegree[s]][s] = 1 # insert modified successor
      if len(result) == self.nVertices(): # All vertices in result?
         return result      # Yes, then return it, otherwise cycle
      raise Exception('Cycle in graph, cannot sort')
```

After the `vertsByDegree` array is populated with all the vertices and an empty `result` list/array created, the algorithm can now process the vertices in a `while` loop. The loop condition tests whether the hash table for inbound degree 0 vertices has any keys in it. If there are no degree 0 vertices, then we're either done, or the graph has at least one cycle in it.

Inside the `while` loop body, we pop one of the degree 0 vertices out of its hash table. Python has several methods for removing items from hash tables. The `popitem()` method removes and returns a key and its value from the hash table. In this case, the sorting algorithm doesn't care which key is removed, nor what value it has, so it puts the value in the underscore (_) variable. (Python's `popitem()` always removes the last key inserted, making it behave like a stack.) Another way to remove a key from a hash table is to use the `pop()` method, which requires a specific key.

Next, the algorithm appends the degree 0 vertex to the result list. Because the `vertex` has no predecessors, it can be the next vertex in the output sequence. That allows each of the `vertex`'s successors to reduce their inbound degree by one. The inner `for` loop goes through the adjacent vertices of `vertex`, which are the same as the successors for directed graphs. Each successor vertex, `s`, is removed from the hash table holding `s` based on its current degree. The `vertsByDegree[inDegree[s]]` looks up the hash table and the `.pop(s)` removes the successor's key. Next, the algorithm lowers the degree for `s` by one, and `s` is inserted in the hash table for that lower degree. After all the successors have been lowered by one degree, the algorithm continues the outer `while` loop, popping vertices out of the degree 0 hash table.

When the degree 0 hash table is empty, we check the length of the result list against the number of vertices. If the result contains all the vertices, it can be returned as the sorted vertex list. If not, a cycle prevented finding a vertex without predecessors.

Try looking at the details of a topological sort using the Graph Visualization tool. It implements this improved algorithm and shows how the `inDegree` and `vertsByDegree` arrays are constructed and updated. As the `sortVertsTopologically()` method executes, try to figure out what will happen next at each step. The hash tables are shown as simple lists rather than as arrays with empty cells that would illustrate their load factors, but that is not critical to understanding the overall algorithm.

Efficiency of the Topological Sort

Does this new structure—an array of hash tables—improve the efficiency? Let's review. Building the initial, empty `vertsByDegree` array takes O(N) time. The first `for` loop will also take at least O(N) time because it goes through all the vertices. Now we must look at what happens inside that loop.

The call to `self.degree(vertex)` takes O(N) time because it has to check all N vertices to see whether they have inbound or outbound edges to the given `vertex`. Thus, populating the `inDegree` array makes the overall `for` loop take $O(N^2)$ time because it covers the entire two-dimensional adjacency matrix to compute all of these.

We also must examine the statement that puts the vertices in their appropriate hash table inside the `vertsByDegree` array. It does two array lookups (one inside `inDegree` and the other inside `vertsByDegree`) and a hash table access for every vertex. The lookups and the hash table access are all O(1), so this doesn't add more complexity to the first `for` loop; overall it takes $O(N^2)$ time.

The main `while` loop of the `sortVertsTopologically()` method executes N times (if there are no cycles, and fewer if there are any). Inside the loop, the act of popping an item and inserting it at the end of the result list is O(1). The inner `for` loop must process each of the successors. Finding the (adjacent) successors takes O(N) time, so the inner `for` loop is at least O(N). Some subset of those N vertices will be successors, and each gets popped from a hash table, decremented in degree, and inserted in another hash table, which are all O(1) operations. The inner `for` loop takes O(N) time; thus, the outer, main `while` loop combined with it, takes $O(N^2)$ time.

Both preparing the array of hash tables in the first `for` loop and processing the result in the main `while` loop take $O(N^2)$ time. The extra memory needed for the `vertsByDegree` and `inDegree` arrays has made an improvement over the $O(N^3)$ time of the basic algorithm. That might seem small, but it can be huge when N grows large.

Connectivity in Directed Graphs

You've seen how, in a undirected graph, you can find all the vertices that are connected by doing a depth-first or breadth-first traversal. When you try to find all the connected vertices in a directed graph, things get more complicated. You can't just start from a randomly selected vertex and expect to reach all the other connected vertices.

Consider the small, directed graph in Figure 14-16 and Figure 14-20. If you start on A, you can get to B and D but not to C. If you start on B, you can get only to D, and if you start on D, you can't get anywhere. The meaningful question about connectivity is: What vertices can you reach if you start on a particular vertex?

The Connectivity Matrix

You can easily use the `depthFirst()` method (Listing 14-4) to traverse part of the graph starting from each vertex in turn. For the graph of Figure 14-16, the output will look something like this:

```
ABD
BD
CABD
D
```

This is the **connectivity table** for the directed graph. The first letter is the starting vertex, and subsequent letters show the vertices that can be reached (either directly or via other vertices) from the starting vertex.

Transitive Closure and Warshall's Algorithm

In some applications it's important to find out quickly whether one vertex is reachable from another vertex. This is the essential question in genealogy: Who are my ancestors? Another example is a celebrity connection game such as the "six degrees of Kevin

Bacon," where people try to find a path through acquaintances to reach a particular person. Some people hypothesize that you need at most a path length of six acquaintances to reach a celebrity. In many applications the path length doesn't matter. Perhaps you want to take a trip by train from Athens to Kamchatka, and you don't care how many intermediate stops you need to make. Is this trip possible? Graphs are ideal for answering these questions. The trip won't be possible unless the vertices are part of the same connected component.

You could examine the connectivity table, but then you would need to look through all the entries on a given row, which would take O(N) time (where N is the average number of vertices reachable from a given vertex). But you're in a hurry. Is there a faster way?

It's possible to construct a table that will tell you quickly (that is, in O(1) time) whether one vertex connects to another. Such a table can be obtained by systematically modifying a graph's adjacency matrix. The graph represented by this revised adjacency matrix is called the **transitive closure** of the original graph. Such a revised matrix can be called the **connectivity matrix**.

Remember that in an ordinary adjacency matrix the row number indicates where an edge starts, and the column number indicates where it ends. The connectivity matrix has a similar arrangement, except the path length between the two vertices may be more than one. In the adjacency matrix, a 1 or True at the intersection of row C and column D means there's an edge from vertex C to vertex D, and you can get from one vertex to the other in one step. Of course, in a directed graph it does not follow that you can go the other way, from D to C.

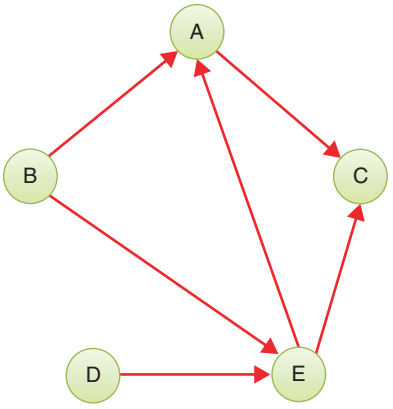
You can use **Warshall's algorithm** named for Stephen Warshall to find the transitive closure of the graph. It changes the adjacency matrix into a connectivity matrix. This algorithm does a lot in a few lines of code. It's based on a simple idea:

If you can get from vertex L to vertex M, and you can get from M to N, then you can get from L to N.

A two-step path is thus derived from two one-step paths. The adjacency matrix shows all possible one-step paths, so it's a good starting place to apply this rule.

You might wonder whether this algorithm can find paths of more than two edges. After all, the rule only talks about combining two one-edge paths into one two-edge path. As it turns out, the algorithm can build on previously discovered multiedge paths to create paths of arbitrary length. The basic idea is: if you can build the connectivity matrix for one and two edge paths, then you could apply the same algorithm to that table to build all the three-edge and fewer paths. If you keep reapplying the algorithm, you will eventually discover all the possible paths.

Here's how it works. Let's use the adjacency matrix of Figure 14-21 as an example. For this example, you examine every cell in the adjacency matrix, one row at a time.



_adjMat

	A	B	C	D	E
A	0	0	1	0	0
B	1	0	0	0	1
C	0	0	0	0	0
D	0	0	0	0	1
E	1	0	1	0	0

FIGURE 14-21 A five-vertex directed graph and its adjacency matrix

Row A

Let's start with row A. There's nothing (0) in columns A and B of that row, but there's a 1 at column C, so you can stop there.

Now the 1 at this location says there is a path from A to C. If you knew there was a path from some other vertex, X, to A, then you would know there was a path from X to C. Where are the edges (if any) that end at A? They're in column A. So, you examine all the cells in column A. In the `_adjMat` of Figure 14-21, there's only one 1 in column A: at row B. It says there's an edge from B to A. So, you know there's an edge from B to A, and another (the one you started with when examining row A) from A to C. From this, you infer that you can get from B to C in two steps. You can verify this is true by looking at the graph.

To record this result, you put a 1 at the intersection of row B and column C. The result is shown in the second matrix in Figure 14-22. The highlighted cell shows where the value changed to 1.

The remaining cells of row A are blank. You only need to copy the contents of column A to columns that have a 1 in row A. In other words, you perform an operation akin to a bitwise OR of column A and column C for this first "bit" that you find turned "on". Note that column A also has a 1 for vertex E. Column C already had a 1 for vertex E, so the bitwise OR operation doesn't change it.

Rows B, C, and D

Next, you go to row B. The first cell, at column A, has a 1, indicating an edge from B to A. Do any edges end at B? You look in column B, but it's empty, so you know that none of the 1s you find in row B will result in finding longer paths because no edges end at B. You could perform the bitwise OR of column B with the three columns that have a 1 in row B, but a bitwise OR with 0s doesn't change anything, as shown in the fourth panel of Figure 14-22.

Row C has no 1s at all, so you go to row D. Here, you find an edge from D to E. Column D is empty, however, so no edges end on D and the OR changes nothing.

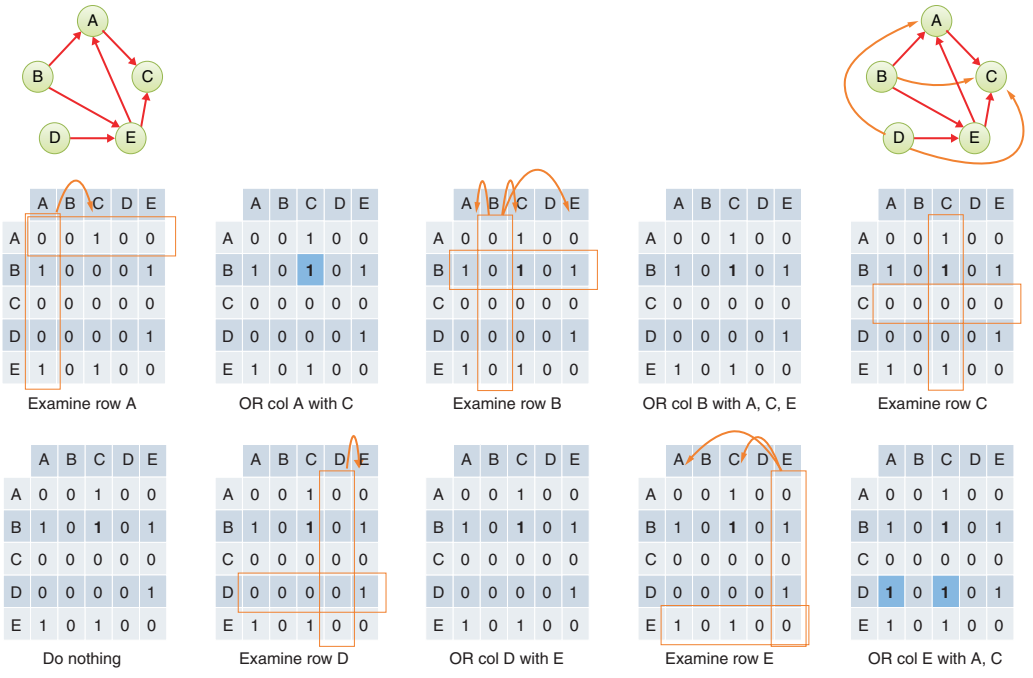


FIGURE 14-22 Steps in Warshall's algorithm

Row E

In row E you see there are edges from E to A and from E to C. Looking in column E, you see the first entry is for the edge B to E, so with B to E and E to A, you infer there's a path from B to A. That path, however, has already been discovered, as indicated by the 1 at that location.

There's another 1 in column E, at row D. This edge from D to E plus the ones from E to A and E to C imply paths from D to A and C, so you insert a 1 in both of those cells. The result is shown in last matrix of Figure 14-22.

Warshall's algorithm is now complete. You've added three 1s to the adjacency matrix, which now shows which vertices are reachable from another vertex in any number of steps. The graph at the top right of Figure 14-22 shows the edges added to the graph by the transitive closure as curved arrows in a different color.

Long Paths

Can Warshall's algorithm find long paths in the graph and build the full closure? It seems as though going row by row once through the matrix might not find long, complicated chains of edges. Figure 14-23 shows a longer example.

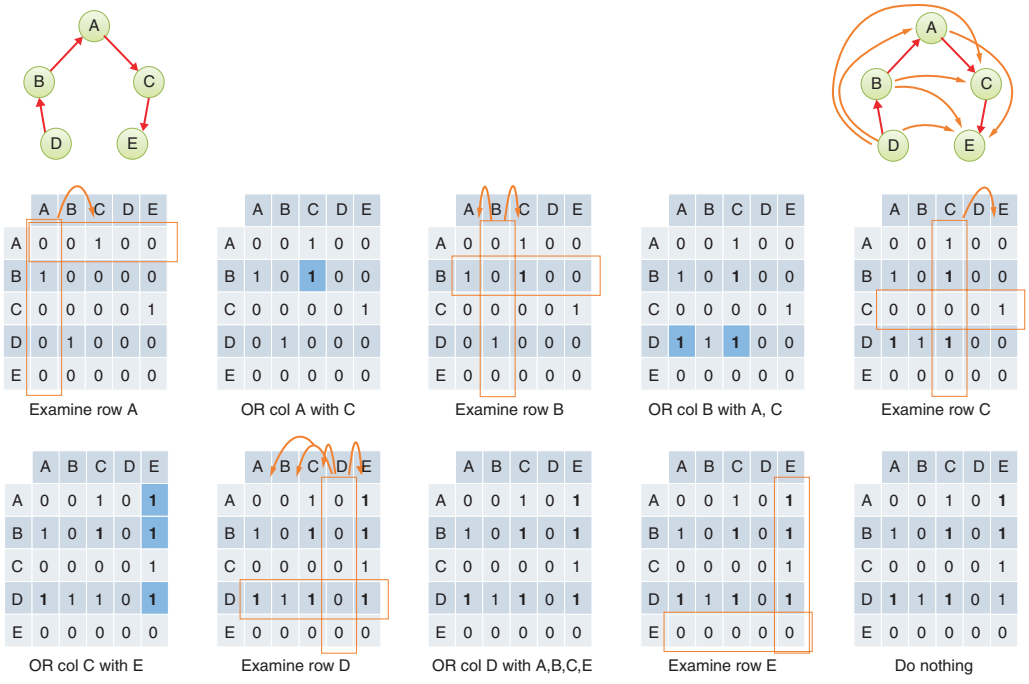


FIGURE 14-23 Warshall's algorithm on a long path

In this graph, the five vertices are connected by four edges to form a long chain. Each step of Warshall's algorithm adds new edges to the matrix. First, one edge is added for row A, then two edges when it examines row B, and then three edges for row C. The new entries sometimes line up in a column and sometimes in a row. The six edges are all that are needed to build the final connectivity matrix; rows D and E contribute nothing because either the row or its corresponding column contains all 0s.

Cycles

If D were connected to E in the graph of Figure 14-23, it could form a cycle, assuming the direction of the edge went from E to D. What happens in the connectivity matrix if there's a cycle? Does Warshall's algorithm still work?

In fact, Warshall's algorithm can be used to *detect cycles*. Consider the example shown in Figure 14-24. Vertices A, B, C, and D form a cycle with vertex E dangling from vertex C. In the initial adjacency matrix, which is the upper-left matrix in the figure, the identity diagonal where the row index is the same as the column index has all 0s. That's what you expect in adjacency matrices because only pseudographs to allow vertices to have edges to themselves.

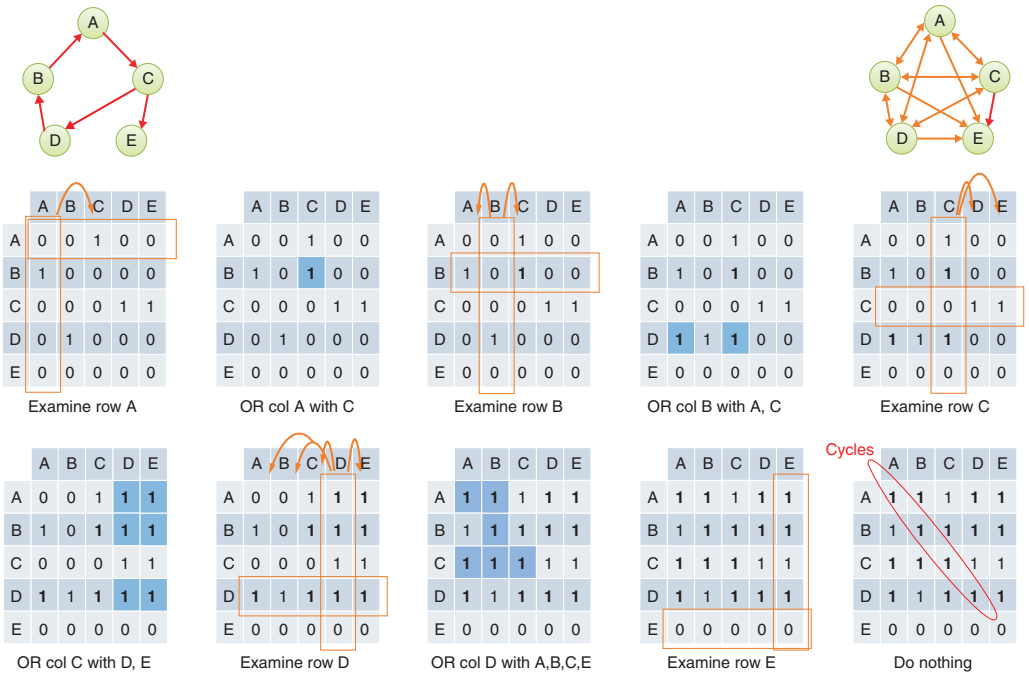


FIGURE 14-24 Warshall's algorithm applied to a graph with a cycle

If you go through the steps of Warshall's algorithm, the final connectivity matrix shows all 1s along the diagonal except for vertex E (as circled in the matrix at the bottom right of Figure 14-24). That means vertices A, B, C, and D must be part of at least one cycle. There is some path starting from each vertex that leads back to itself.

The presence of 1s along the diagonal tells you that cycles must exist, but not how many there are nor which vertices are in which cycles. Those are harder problems to solve. Nevertheless, it's useful to have methods that identify the presence (and absence) of cycles. The 0 in the diagonal entry for vertex E in Figure 14-24 tells you that it is not part of a cycle. (Perhaps schools should be required to check for 0s by applying this to the dependency relationship graph of all their course prerequisites.)

Implementation of Warshall's Algorithm

One way to implement Warshall's algorithm is with three nested loops (as suggested by Sedgewick; see Appendix B, "Further Reading"). The outer loop looks at each row; let's call its variable `R`. The loop inside that looks at each cell (column) in the row; it uses variable `C`. If a 1 is found in matrix cell (`R`, `C`), there's an edge from `R` to `C`, and then a third (innermost) loop is activated.

The third loop performs the OR operation between column R and column C. It has to loop over each of the cells (vertices) in those columns using its own variable and perform the OR between their values. We leave the details as an exercise.

Because there are three loops over all N vertices, the overall complexity is $O(N^3)$. That's a lot of computation to build the connectivity matrix. If you only want to find the answer to the problem "Is there a sequence of train trips that go from Athens to Kamchatka?" you could find the answer in $O(N^2)$ time with a depth-first or breadth-first search. Building the full connectivity matrix first, however, can make a huge difference in other, advanced graph algorithms. It could also be used to test for the presence of cycles, possibly returning as soon as any diagonal is set to one. That would still take $O(N^3)$ time (in the worst case and average case), so it's not very fast. We discuss the complexity of this and other graph algorithms in the next chapter.

Summary

- Graphs consist of vertices connected by edges.
- Graphs can represent many real-world entities such as transportation routes, electrical circuits, and job scheduling.
- Vertices are adjacent if a single edge connects them.
- The adjacency of vertices is usually represented by either an adjacency matrix or adjacency lists.
- Adjacency matrices can be represented using two-dimensional arrays or hash tables.
- Traversal algorithms allow you to visit each vertex in a graph in a systematic way and are the basis of several other activities such as searches.
- The two main traversal algorithms are depth-first (DF) and breadth-first (BF).
- The depth-first traversal can be based on a stack; the breadth-first traversal can be based on a queue.
- A breadth-first search finds the shortest path between two vertices (in terms of number of edges), if one exists.
- Depth-first search explores parts of the graph furthest away from the starting vertex early in the traversal, which can be useful in move analysis of games.
- A minimum spanning tree (MST) is a subgraph with the minimum number of edges necessary to connect all an undirected graph's vertices.
- Minimum spanning trees are useful in finding layouts of networks with the fewest number of interconnections.
- A minimum spanning tree can be determined using depth-first traversal on an unweighted, undirected graph.
- Trees are a type of undirected graph where a unique path connects any two vertices.
- In a directed graph, edges have a direction (often indicated by an arrow).
- Directed graphs can represent situations such as dependency relationships, river flows, and one-way road networks.

- The adjacency matrices of undirected graphs always have mirror image symmetry, but those of directed graphs do not.
- In a topological sort of directed graph vertices, if there is a path from vertex A to vertex B, vertex A precedes B in the result list. Vertex pairs not connected by a path can appear in either order.
- Topological sorting can be done only on directed acyclic graphs (DAG)—graphs without cycles.
- Topological sorting is typically used for scheduling complex projects that consist of tasks contingent on other tasks.
- Topological sorting can be done in $O(N^2)$ time, where N is the number of vertices, by computing the inbound degree of each vertex and holding the vertices in hash tables for each degree.
- Warshall's transitive closure algorithm finds whether there is a connection, of either one or multiple edges, from any vertex to any other vertex.
- Warshall's algorithm transforms an adjacency matrix into a connectivity matrix, and that matrix can be used to detect the existence of cycles.
- The basic implementation of Warshall's algorithm could take $O(N^3)$ time.

Questions

These questions are intended as a self-test for readers. Answers may be found in Appendix C.

1. In a graph, a(n) _____ connects two _____.
2. How do you tell, by looking at its adjacency matrix, how many edges there are in an undirected graph?
3. In a game simulation using a graph, a(n) _____ corresponds to a game board state and a(n) _____ corresponds to a player's move.
4. A directed graph is one in which
   a. you must follow the minimum spanning tree.
   b. you must go between vertices in topologically sorted order.
   c. you can go in only one direction from one given vertex to another.
   d. you can go in only one direction on any valid path.
5. If a graph's adjacency matrix has rows [0,1,0,0], [1,0,1,1], [0,1,0,0], and [0,1,1,0], what is the corresponding adjacency list for vertices A, B, C, and D?
6. A minimum spanning tree is a graph in which
   a. the number of edges connecting all the vertices is as small as possible.
   b. the number of edges is equal to the number of vertices.

c. all unnecessary vertices have been removed.

d. every combination of two vertices is connected by the minimum number of edges.

7. How many different minimum spanning trees are there in a undirected graph of three vertices and three edges?

8. Choose the fastest way to check whether a path exists from vertex A to vertex Z in a directed graph among these options.

a. Get the `minimumSpanningTree(A)`, and then find the path from the root of that tree to Z.

b. Loop over the vertices returned by calling `depthFirst(A)` until Z shows up.

c. Loop over the vertices returned by calling `breadthFirst(Z)` until A shows up.

d. Apply Warshall's algorithm to the graph and then check the connectivity matrix to see if A can reach Z.

9. An undirected graph must have a cycle if

a. any vertex can be reached from some other vertex.

b. the number of connected components is more than one.

c. the number of edges is equal to the number of vertices.

d. the number of paths is fewer than the number of edges.

10. A(n) _____ is a graph with no cycles.

11. The degree of a vertex

a. is the number of edges in the path linking it to a starting vertex.

b. is the number of edges that connect it to other vertices.

c. is the number of vertices in its connected component of the graph.

d. is half the number of edges in its row of the adjacency matrix.

12. Can a minimum spanning tree for an undirected graph have cycles?

13. True or False: There may be many correct topological sorts for a given directed graph.

14. Topological sorting results in

a. edges being directed so vertices are in ascending order.

b. vertices listed in order of increasing number of edges from the beginning vertex.

c. vertices arranged in ascending order, so F precedes G, which precedes H, and so on.

d. vertices listed so the ones later in the list are downstream from the ones earlier.

15. If a graph's adjacency matrix has rows [0,1,0,0], [0,0,0,1], [1,0,0,0], and [1,0,0,0] with vertices A, B, C, and D, could it be passed as an argument to `sortVertsTopologically()`? If so, what would the result be?
16. What's a DAG?
17. Warshall's algorithm
    a. finds the largest cycle in a graph, if there is one.
    b. changes the adjacency matrix into a connectivity matrix.
    c. sorts the vertices in ascending order but not topologically.
    d. finds the fewest number of edges needed to perform closure.
18. Under what conditions does it make sense to perform a topological sort on a undirected graph?
19. If graph G1 has 100 vertices and G2 has 10 vertices, what's the computing time ratio between calling `sortVertsTopologically()` on G1 and G2?
20. Which algorithm solves the bridges of Königsberg problem?
    a. Warshall's algorithm
    b. the minimum spanning tree algorithm
    c. the topological sort algorithm
    d. the breadth-first traversal algorithm

Experiments

Carrying out these experiments will help to provide insights into the topics covered in the chapter. No programming is involved.

14-A You saw how depth-first traversal is used to determine the minimum spanning tree of a particular graph. Could you use breadth-first traversal instead? If not, why not? If so, what would be different between the two methods? Experiment with the sample graphs shown in this chapter.

14-B Think about representing the bridges of Königsberg network (see Figure 14-3) in the computer. Would using an adjacency matrix work? How about an adjacency list? Does it matter if the bridges need distinct labels (like those in the figure) or if all that needs to be stored is the number of edges between two vertices? If either representation won't work, explain why not, and propose a way to make it work.

14-C Using the Graph Visualization tool, start with a new (empty graph) and collapse the Adjacency Matrix window. Then randomly fill in 5 vertices and add 7 edges. Without exposing the Adjacency Matrix, write down the adjacency matrix for the graph. When you're done, expand the matrix view to see if you got it right. Repeat this exercise with a directed graph of 5 vertices and 10 edges.

14-D On paper, create a five-by-five matrix. Put Xs along the diagonal. Then randomly fill some of the cells with 1s, leaving the rest blank (or filled with 0s) to make a five-vertex adjacency matrix. Don't worry about symmetry around the diagonal. Now, with the Adjacency Matrix of the Graph Visualization tool hidden, create the corresponding directed graph from your paper matrix. When you're done, show the matrix in the tool to see if the graph corresponds to your adjacency matrix.

14-E In the Graph Visualization tool, see whether you can create a directed graph with a cycle that the Topological Sort operation cannot identify.

Programming Projects

Writing programs to solve the Programming Projects helps to solidify your understanding of the material and demonstrates how the chapter's concepts are applied. (As noted in the Introduction, qualified instructors may obtain completed solutions to the Programming Projects on the publisher's website.)

14.1 Change the `Graph` class to be able to create both kinds of graphs: undirected or directed. Add an optional `directed` parameter to the constructor that is `False` by default. Change all the methods that have different behavior for directed graphs than undirected graphs based on whether the `directed` flag is set, including the `print()` method to show `->` (instead of `<->`) for directed edges. Construct both kinds of graphs using 10 vertices (A, B, C, D, E, F, G, H, I, and J) and the 12 edges (AG, AI, CF, DA, DI, HD, HE, HF, HG, IH, JC, and JH). Show the output of the `print()` method on both graphs and the `depthFirst()` traversal vertices and paths starting from vertex J.

14.2 Make a recursive traversal generator, `depthFirstR()`. The method shown in Listing 14-4 uses a stack to track the path to the current vertex. Your method should make use of the call stack and values passed in the recursive calls to determine the order of visiting the vertices and the paths to yield. Demonstrate the output of your generator on the graph in Project 14.1 and on the graph used in the second breadth-first example in Figure 14-10. Show the order of the vertices and the paths to each one.

14.3 Modify the `breadthFirst()` generator to return the path to each vertex as well as the vertex being visited. Use it to write a `shortestPath()` method that finds the shortest path between two vertices, if such a path exists. Demonstrate its output searching for the shortest path from vertex A to H in the second breadth-first example of Figure 14-10, and from vertex F to A in the initial graph of Figure 14-14. Include a full breadth-first traversal starting from vertices A and F, respectively, in those graphs.

14.4 Implement Warshall's algorithm to compute the connectivity matrix from the adjacency matrix of a graph. Instead of updating the adjacency matrix, write a new method, `connectivityMatrix()`, that returns a new matrix. The result can start out as a copy of the internal `_adjMat` (using the Python `dict`'s `copy()` method). Use your directed graph implementation from Project 14.1. Write a second method, `hasCycles()`, that tests for the presence of cycles in the graph. Demonstrate your methods on the graphs of Figure 14-22, Figure 14-23, and Figure 14-24.

14.5 A *clique* is a graph or subgraph of N vertices where every vertex is adjacent to all the N–1 other vertices. In graphs of communication patterns, cliques can indicate interconnected and influential groups. They represent groups of people or organizations that communicate with one another—more so than with others in the graph. That tendency to interact only with members of the group leads to the feeling of exclusion felt by people outside a social clique.

Add a method to the `Graph` class that returns all the subgraph cliques with N vertices. The parameter, `N`, should be between 2 and the total number of vertices in the graph to be meaningful. Every pair of adjacent vertices forms a clique of size 2 (in a undirected graph), so there is exactly one subgraph clique of size 2 per edge. Larger cliques can be formed by adding a single vertex to a smaller one, if that vertex has edges to all the vertices in the smaller clique.

Your method should return the cliques as subgraphs with N vertices. Demonstrate your method's output when seeking cliques of size 3, 4, and 5 in at least the following two graphs:

- A 10-vertex graph where five of the vertices are fully interconnected and the other five are not connected to more than one other vertex.
- A 10-vertex graph with three overlapping cliques of size 4. The cliques overlap by sharing one vertex. This pattern can be generated using the following expression for vertex index pairs:

```
[(a, b) for c in range(0, 9, 3)
for a in range(c, c + 4) for b in range(a + 1, c + 4)]
```

Note: Finding cliques in graphs can be quite computationally complex, especially when searching for all of them. Running your method on large graphs could take a long time to process, as we discuss in the next chapter.

14.6 The Knight's Tour is an ancient and famous chess puzzle. The object is to move a knight from one square to another on an otherwise empty chess board until it has visited every square exactly once. Write a program that solves this puzzle using a depth-first search. It's best to make the board size variable so that you can attempt solutions for smaller, square boards (K×K). The regular 8×8 board could take years to solve on small computers, but a 5×5 board should take less than a minute. We have more to say about the complexity of this problem in the next chapter.

Refer to the "Depth-First Traversal and Game Simulations" section in this chapter, keeping in mind that a puzzle is like a one-player game. It may be easier to think of a new knight being created and remaining on the new square when a move is made (rather than moving a single knight around). This way, a sequence of added knights represents the game board state, and the occupied squares can be deduced from the knights' positions. When the board is completely filled with knights (the sequence of knights equals the size of the board), you win.

When looking for its next move, a knight must not only make a legal knight's move (two spaces in one direction and one space in the other), but it must also not move

off the board or onto an already-occupied (visited) square. If you make the program display the board and wait for a keypress after every move, you can watch the progress of the algorithm as it places more and more knights on the board. When it gets boxed in, you can see it backtrack by removing some knights and trying a different series of moves.

This problem has some complexities that might not seem obvious. One of the most important is what the vertices in the graph represent. Looking at the tic-tac-toe boards in Figure 14-8 would suggest using vertices to represent the board states: where each of the nine squares is either blank, an X, or an O. To use the `depthFirst()` traversal method shown in Listing 14-4, you would need to first create a `Graph` with a vertex for every possible board state. In tic-tac-toe there are 3^9 = 19,683 possible ways of placing blank, X, or O in the nine squares. Even though many of those would be impossible in a real game (for example, where the number of Xs and Os differ by more than 1), creating all those vertices and then adding edges between them would be time-consuming. For an 8×8 chessboard where every square is either empty or occupied by a knight, there are 2^{64}, or over 4 billion, board states. That is not likely to be an efficient way to solve this problem.

For many game simulations, the graph is not fully created at the beginning. As moves are made and potential counter moves are explored, new vertices are added to the graph based on the legal moves from the last board state. Thus, the graph is only partially represented throughout the game. That means that the depth-first traversal methods we implemented won't work to solve the Knight's tour because neither the adjacency matrix nor the list of vertices is complete.

Another approach is to create *one vertex per square on the board*. Edges between vertices could then represent legal knight moves in chess. In this way the graph represents *legal board moves*, not board states. The state of the board is implicit in the path taken by the search, which is a sequence of squares where the knights are placed. The number of vertices for the graph would be K×K, with approximately 4×K×K edges. If you use the `depthFirst()` traversal method on this legal board move graph, would you solve the puzzle? This method would certainly provide a path of all legal moves and avoid revisiting vertices (squares) previously visited on the path. The problem, however, is that it is designed to visit every vertex exactly once.

In the case of the Knight's Tour, you need to explore every possible *path* to a vertex. To see why, imagine that after visiting 24 of the 25 squares in a 5×5 board, you find that the last empty square cannot be reached from the last square visited. So, you must backtrack in the depth-first search. Let's say you return to the 20th knight and try a new path through the remaining 5 squares. If you had marked 4 of those last 5 squares in the `visited` array used by the `depthFirst()` traversal method, they wouldn't be searched again. You need a different way of marking what has already been searched.

If you use a legal board move graph, you will need to write a depth-first traversal that explores all potential paths in the graph, not just all vertices. If you create game board state vertices as you search, then you must ensure that you visit those vertices in depth-first order.

This page intentionally left blank

CHAPTER 15

Weighted Graphs

IN THIS CHAPTER

- Minimum Spanning Tree with Weighted Graphs
- The Shortest-Path Problem
- The All-Pairs Shortest-Path Problem
- Efficiency
- Intractable Problems

In the preceding chapter you saw that a graph's edges can have direction. This chapter explores another edge feature: weight. For example, if vertices in a weighted graph represent cities, the weight of the edges might represent distances between the cities, or costs to fly between them, or the number of vehicle trips made annually between them (a figure of interest to highway engineers).

When you include weight as a feature of a graph's edges, some interesting and complex questions arise. What is the minimum spanning tree for a weighted graph? What is the shortest (or cheapest) distance from one vertex to another? Such questions have important applications in the real world.

We first examine a weighted but undirected graph and its minimum spanning tree. In the second half of this chapter, we examine graphs that are both directed and weighted, in connection with the famous Dijkstra's algorithm, used to find the shortest path from one vertex to another.

Minimum Spanning Tree with Weighted Graphs

To introduce weighted graphs, we return to the question of the minimum spanning tree. Creating such a tree is a bit more complicated with a weighted graph than with an unweighted one. When all edges are the same weight, it's fairly straightforward—as you saw in Chapter 14, "Graphs"—for the algorithm to choose one edge to add to the minimum spanning tree. When edges have different weights, however, you need to choose a bit more carefully.

An Example: Networking in the Jungle

Suppose you want to install high-speed network lines to connect six cities in the mythical country of Turala. Five links are all that is needed to connect the six cities, but which five links should they be? The cost of connecting each pair of cities varies, so you must pick the route carefully to minimize the overall cost. Figure 15-1 shows a weighted graph with six vertices. Each edge has a weight, shown by a number alongside the edge. This is the abstract form of a weighted graph. Notice that some links are not shown (for example, no direct link from A to D or A to F). When the graph represents a real-world problem, some links could be impractical because of distance, terrain, environmental, or other issues.

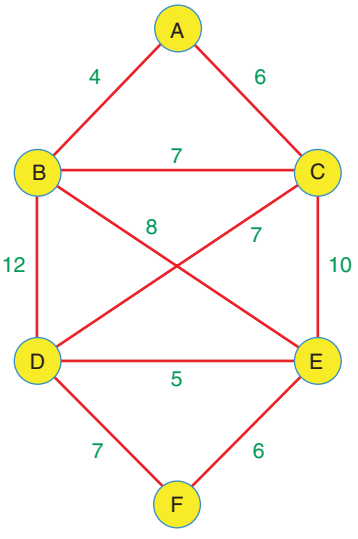


FIGURE 15-1 A weighted graph

How can you pick a subgraph that minimizes the cost of connecting vertices into a network? The answer is to calculate a minimum spanning tree. It will have five edges (one fewer than the number of vertices), it will connect all six cities, and it will minimize the total cost of the links. Can you figure out this route by looking at the graph in Figure 15-1? If not, you can solve the problem with the WeightedGraph Visualization tool.

The WeightedGraph Visualization Tool

The WeightedGraph Visualization tool is like the Graph tool, but it creates weighted, undirected graphs. You can create vertices, link them with edges, and show or hide the adjacency matrix as before. In the shaded graph region, you can double-click to create a new vertex, drag from one vertex to another to create an edge, and drag using the second mouse button or by holding the Shift key to move a vertex. Double-clicking an edge or vertex deletes it.

Edges created by dragging the pointer get an initial weight of 1. You can change the weight by selecting it—either in the shaded graph region or in the adjacency matrix—and changing the numeric value. As you change weights in one place, they update everywhere. If you erase the weight, or make it zero, the edge disappears. Weights are restricted to the values 1–99.

Try out this tool by creating some small graphs and finding their minimum spanning trees. (For some configurations, you'll need to be careful positioning the vertices so that the weight numbers don't fall on top of each other.)

Use the WeightedGraph Visualization tool to construct the graph of Figure 15-1. The result should look something like Figure 15-2. If you happen to have the Visualization tool running on a computer where you can launch it from the command line, you can initialize the graph (with random positioning of the vertices) using this command:

```
python3 WeightedGraph.py A B C D E F A-B:4 A-C:6 B-C:7 B-D:12 B-E:8 C-D:7 C-E:10
D-E:5 D-F:7 E-F:6
```

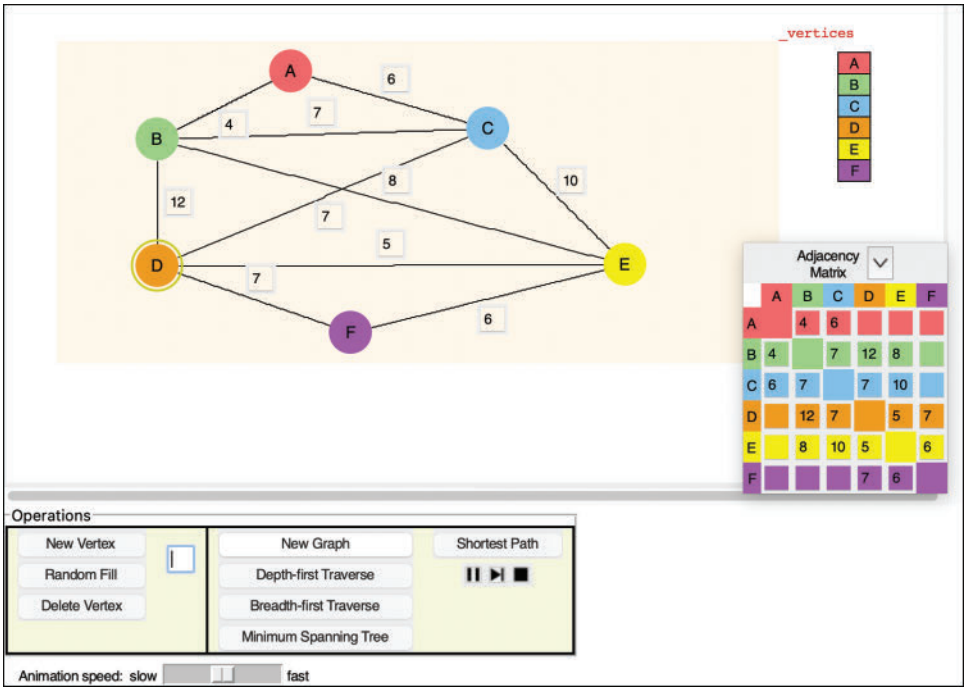


FIGURE 15-2 The WeightedGraph Visualization tool

Now find this graph's minimum spanning tree starting from vertex A by clicking that vertex to highlight it with the blue ring and then selecting the Minimum Spanning Tree button. The result should be the minimum spanning tree shown in Figure 15-3.

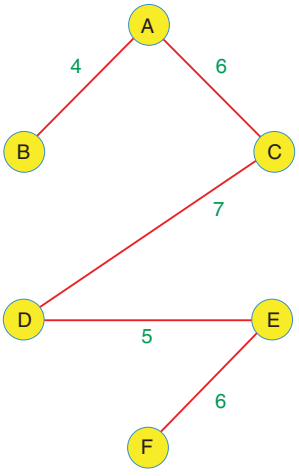


FIGURE 15-3 The minimum spanning tree of the previous graph

The tool discovers that the minimum spanning tree consists of the edges AB, AC, CD, DE, and EF, for a total edge weight of 28. The order in which the edges are specified is unimportant. If you start at a different vertex, you will create a tree with the same edges, but in a different order. For a graph with distinct connected components, the minimum spanning tree connects only one of them, as you saw in Chapter 14.

Building the Minimum Spanning Tree: Send Out the Surveyors

The algorithm for constructing the minimum spanning tree is a little involved, so we're going to introduce it with the analogy of building a high-speed network in Turala. Imagine that your company won the contract to bring high-speed networking to this jungle-covered country. You are the project manager, of course, and there are also various surveyors on your team.

The first challenge of this project is something that happens in many problems: you have to "discover" the graph. Although the location of the cities and the approximate cost of laying network cables or creating point-to-point microwave transmission systems are known, the exact details of the terrain between the cities is not. That means that the graph and its edge weights are unknown at the start. Part of your job is to discover that information.

Computer algorithms always focus on tiny parts of a problem. Unlike humans who like to look at the big picture, computers focus on a single vertex, edge, node, key, and so on, at a time, making a series of local decisions that combine to arrive at a particular goal. With graphs, algorithms tend to start at some vertex and work away from it, acquiring data about nearby vertices before finding out about vertices farther away. This is especially true in large graphs, which are difficult or impossible to fully represent at one time, such as all possible routes between the stars in a galaxy.

In a similar way, planning the route in Turala involves getting to some of the cities and finding out what lies on the roads and paths linking them. Acquiring this information takes time. That's where your surveyors come in.

Starting in Blum

You start by setting up an office in the city of Blum. (You could start in any city, but you've heard Blum has the best restaurants.) The other cities on the contract to be connected are Cerf, Dahl, Gray, Kay, and Naur. Blum is at one end of the country, and you learn that only two cities can be reached from Blum—Cerf and Dahl—as shown in Figure 15-4. The graph shows Blum with a thicker colored outline to indicate you have an office there.

Technically, you could try to reach the other cities by direct paths through the jungle. The network must reach those two cities, and your analysis shows that it will be less expensive to go through one or both cities than to cut through the jungle, mountains, and rivers on some new path.

FIGURE 15-4 First cities visited in Turala

You send two of your surveying teams out along the paths to these cities to see what conditions are like. Their job is to evaluate these routes for different networking options (buried cable, overhead cable, transmission towers, and so on). They take the actual distance and add extra cost factors for soil conditions, river crossings, elevation changes, and the like.

The first team arrives in Cerf and calls you with their report; they say the link from Blum to Cerf should cost 31 million Turala kopeks (and the scenery is gorgeous). The second team reports a little later from Dahl that the Blum–Dahl link, which crosses more level country, should cost 24 million. You make a list:

Blum–Dahl, 24

Blum–Cerf, 31

You always list the links in order of increasing cost; you'll see why this is a good idea soon.

Finalizing the Blum–Dahl Link

At this point you figure you can send out the construction crew to actually build the Blum–Dahl link. How can you be sure that link will eventually be part of the optimal solution (the minimum spanning tree)? So far, you know the cost of only two links in the system. Don't you need more information?

To get a feel for this situation, try to imagine some other route linking Blum to Dahl that would be better than the direct link. If it doesn't go directly to Dahl, this other route must go through Cerf and circle back to Dahl, possibly via one or more other cities indicated by the question marks in Figure 15-5. You already know the link from Blum to Cerf would be

more costly, 31 million, than the link from Blum to Dahl at 24. Your preplanning indicated that trying to reach any of the other cities directly from Blum would be even more costly. So even if the remaining links in this hypothetical circle route are cheap, with a cost of 1 million (or even 0), it will still cost more to get to Dahl from Blum by going through Cerf and the unknown cities (> 31 million). OK, then what if the costs are high on the unknown circle route, say 100 (or really anything bigger than 31)? If those costs are high, you'll probably keep both the Blum–Dahl and Blum–Cerf routes in the plan. In both cases for the unknown costs, the Blum–Dahl route stays. The Blum–Cerf link is not certain, so you won't build anything there yet.

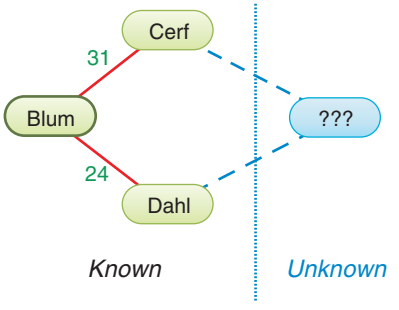


FIGURE 15-5 Hypothetical circle route to Dahl

You conclude that the Blum–Dahl route will be part of the minimum spanning tree. This isn't a formal proof (which is beyond the scope of this book), but it does suggest your best bet is to pick the cheapest link. You send some of your staff to set up an office in Dahl.

Why do you need another office? Due to a Turalan government regulation, you must install an office before you can send out surveyors from a town. In graph terms, you must add a vertex to the tree before you can learn the weight of the edges leading away from that vertex. All towns with offices are on links that will be in the final minimum spanning tree; towns with no offices are not yet connected.

Building the Blum–Cerf Link

After you've completed the Blum–Dahl link and built your new office, you can send out surveyors from Dahl to all the cities reachable from there. They learn that these are Cerf, Gray, and Kay. The survey teams reach their destinations and report back costs of 35, 41, and 52 million, respectively. Of course, you don't send a survey team back to Blum because you've already surveyed the route, installed the networking link, and have an office there (with a nice view of the coast).

Now you know the costs of four links (in addition to the one you've already built):

- Blum–Cerf, 31
- Dahl–Cerf, 35
- Dahl–Gray, 41
- Dahl–Kay, 52

At this point it may not be obvious what to do next. There are many potential links to choose from. What do you imagine is the best strategy now? Here's the rule:

Rule

From the list, always pick the lowest-cost edge.

Actually, you already followed this rule when you chose which route to follow from Blum; the Blum–Dahl edge had the lowest weight (cost). Here the lowest-cost edge is Blum–Cerf, so you can now finalize the Blum-to-Cerf route at a cost of 31 million. You can now open an office in Cerf (and enjoy the unique musical culture there).

Let's pause for a moment and make a general observation. At a given time in planning the best route, there are three kinds of cities:

1. Cities on routes that have been finalized. (In graph terms they're vertices in the minimum spanning tree.)
2. Cities that have been visited by the surveyors, so you know the cost to link them to at least one city in the first group of cities. You can call these "fringe" cities.
3. Cities that have not been visited by your team.

At this stage, Blum, Dahl, and Cerf are in category 1; Gray and Kay are in category 2; and Naur is in category 3, as shown in Figure 15-6. As you work your way through the algorithm, cities move from unknown to fringe, and from fringe to finalized.

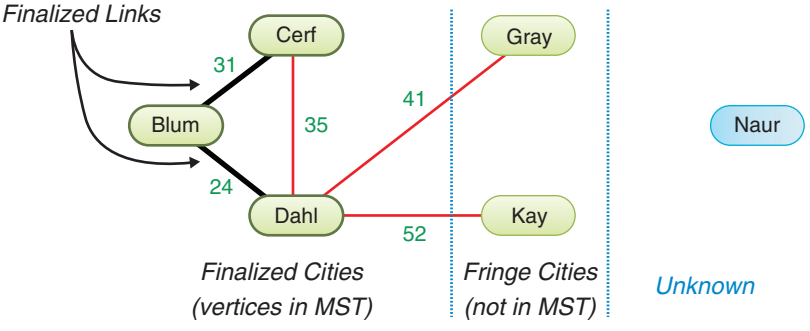


FIGURE 15-6 Partway through the minimum spanning tree algorithm

Building the Cerf–Kay Link

At this point, Blum, Dahl, and Cerf are connected to the network and have offices. You already know the costs from Blum and Dahl to cities in category 2, but you don't know the costs of connecting from Cerf. So, from Cerf you send out surveyors to Gray, Kay, and Naur. They report back costs of 49 million to Gray, 38 million to Kay, and 87 million to Naur. Here's the new list:

~~Dahl–Cerf, 35~~

Cerf–Kay, 38

Dahl–Gray, 41

Cerf–Gray, 49

Dahl–Kay, 52

Cerf–Naur, 87

The Dahl–Cerf link was on the previous list and is crossed out in this list. Why? Well, there's no point in considering links to cities that are already connected, even by an indirect route. Furthermore, minimum spanning trees must not contain cycles. You can now revise your rule slightly:

Rule

From the list, always pick the lowest-cost edge *to a fringe city (vertex).*

From this list, you can see that the next least costly route is Cerf–Kay, at 38 million. You send out the crew to install this link and set up an office in Kay, resulting in the graph of Figure 15-7.

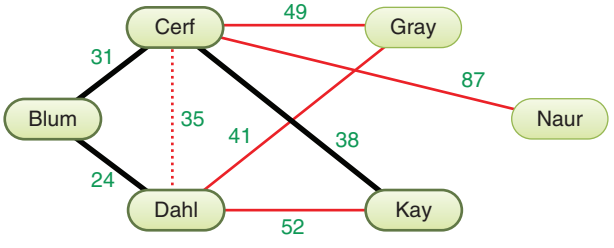


FIGURE 15-7 The minimum spanning tree after visiting all links from Cerf

Building the Kay–Gray Link

From Kay, you dispatch more survey teams, and they report back costs of 25 million to Gray and 40 to Naur. The Dahl–Kay link from the previous list must be removed because Kay is now a connected city. Your new list of edges to fringe cities (ignoring the crossed out links to finalized cities) is

Kay–Gray, 25

Dahl–Gray, 41

Kay–Naur, 43

Cerf–Gray, 49

Cerf–Naur, 87

The lowest-cost link is Kay–Gray, so you build this link and install an office in Gray.

And, Finally, the Link to Naur

The choices are narrowing. After you remove already-linked cities and send out surveyors from Gray, the situation appears like the left-hand graph in Figure 15-8. Your list now shows only:

Kay–Naur, 43

Gray–Naur, 46

Cerf–Naur, 87

You install the last link from Kay to Naur, build an office in Naur, and you're done. You know you're done because there's now an office in all six cities. You've constructed the links for Blum–Dahl, Blum–Cerf, Cerf–Kay, Kay–Gray, and Kay–Naur, as shown on the right of Figure 15-8. This is the lowest-cost network at 161 million (24 + 31 + 38 + 25 + 43) linking the six cities in Turala.

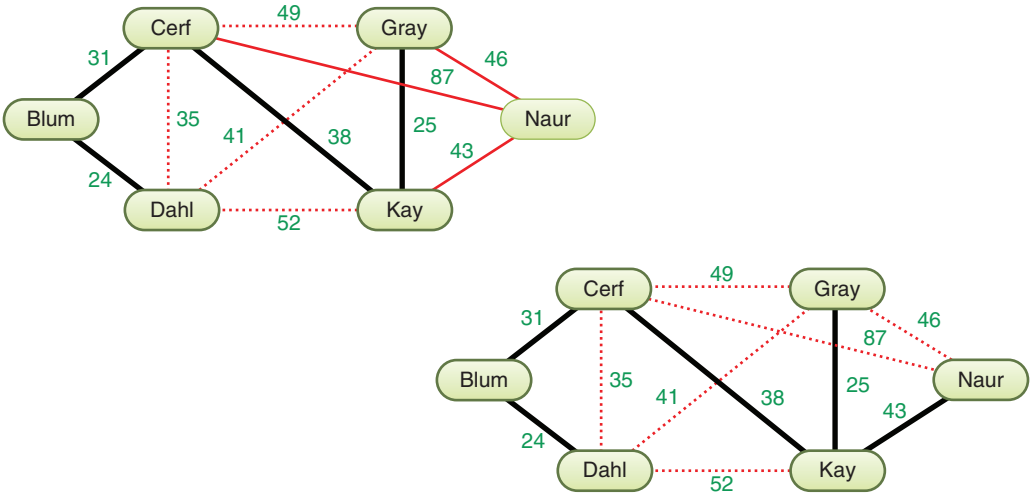


FIGURE 15-8 The final connections of the minimum spanning tree in Turala

Note that the final minimum spanning tree does not include all of the lowest-cost edges. The last link added, from Kay to Naur, had a cost of 43. That's higher than the cost of Dahl–Cerf at 35 and Dahl–Gray at 41. Could there be another minimum spanning tree that uses one of those links? The short answer is yes; there can be other spanning trees with an equal total cost (but none with a lower cost).

Creating the Algorithm

Using the somewhat fanciful idea of networking a small country, we've shown the main ideas behind the minimum spanning tree for weighted graphs. Now let's see how to go about creating the algorithm for this process.

Storing the Edges

The key activities in the example were tracking the category for each city and listing the costs of links between pairs of cities. You decided where to build the next link (edge) and add a new city (vertex) by selecting the minimum cost link.

A collection of weighted edges in which you repeatedly select the minimum value suggests a priority queue or sorted list or minimum heap as an appropriate data structure. These are all efficient ways to handle the selection of the minimum cost edge, but you should look at the other operations that need to be done, too. You certainly need to remove the minimum cost edge from the collection, but you also need to either remove nonminimum edges that connect vertices already in the MST or discard them when they are selected as the minimum cost edge. Of course, you also must insert all the edges in the structure as they are discovered.

Inserting into a priority queue based on a sorted array is an O(N) operation due to shifting the values in the array (or finding the insertion point in a list). Inserting into a heap is an O(log N) operation. For removing the minimum value, priority queues and sorted arrays need O(1) time (if the minimum is kept at the end of the array so that no shifting is required) while the heap takes O(log N).

Let's assume the graph has V vertices and E edges. You only discover those values as the graph is explored, but you can give them names for the analysis. During the process, you will insert E edges into the collection and also remove at least V – 1 of them, perhaps discarding those that link two vertices already in the MST. You know that E must be greater than or equal to V – 1, or the graph must have more than one connected component. Let's summarize the costs of the various structures in Table 15-1. Because we need to analyze the costs for edges and vertices, we use O(E) and O(V) instead of O(N).

Table 15-1 Efficiency of Different Edge Storage Schemes

Structure	E Inserts	V – 1 Removals	Overall
Priority queue	$E \times O(E)$	$(V - 1) \times O(1)$	$O(E^2)$
Sorted list	$E \times O(E)$	$(V - 1) \times O(1)$	$O(E^2)$
Minimum heap	$E \times O(\log E)$	$(V - 1) \times O(\log E)$	$O(E \times \log E)$

In the overall costs, the insertion costs dominate. The removals are much less costly for the priority queue and sorted list, but the inserts grow as the square of the number of edges. For the heap, both insertion and removal grow logarithmically. You can simplify and use E only in the overall cost because it is bigger than V (in all but the simplest graphs). That makes it clear that heaps are the most efficient because $E \times \log E < E^2$ as E grows large.

Instead of a list or array, you use a minimum heap, as described in Chapter 13, "Heaps." Because heaps don't offer an efficient way to delete items other than the minimum valued one, you simply remove the minimum edges until you find one that connects to a fringe vertex.

Outline of the Algorithm

Let's restate the algorithm in graph terms (as opposed to linking city terms).

Start with a vertex (any one will do) and put it in the tree (a subgraph). Then repeatedly do the following:

1. For each outgoing edge from the newest vertex, if it goes to a vertex already in the tree, "prune" (discard) it. Otherwise, put the edge in the minimum heap.
2. Pick the edge with the lowest weight from the heap until either
   a. the heap is empty, or
   b. you reach an edge with one vertex out of the tree.
3. If the heap was emptied in the previous step, then you've found the full tree for the connected component. Otherwise, add the edge to the tree, and note the destination vertex as the newest in the tree.

Repeat these steps until all the vertices are in the tree or the heap becomes empty. At that point, you're done.

In step 1, *newest* means most recently installed in the tree. The edges for this step can be found in the adjacency matrix (or perhaps computed if the graph edges have yet to be discovered). In step 2, edges with one vertex out of the tree are the edges from finalized vertices to fringe vertices.

Implementing Weights: To Infinity and Beyond

When we introduced graphs in Chapter 14, we described how the adjacent vertices could be represented either as an array or as lists. When edges have weights, you need to consider the implementation options again.

Typically, weights can be any number, perhaps including floating-point values. That provides maximum flexibility in representing the weights for a wide range of problems. If you use a list of adjacent vertices, you simply add a field to each link in that list to hold the weight associated with the edge going to the vertex. In adjacency matrices, you used numbers to represent adjacent versus nonadjacent vertices (1 versus 0). Could you just put the edge weight as the number in that array cell? In other words, could `adjMat[j, k]` hold the weight of the edge between vertex indices `j` and `k`? The answer is not quite as simple as it might appear.

You need to distinguish adjacent and nonadjacent vertices. If the weights are never allowed to be zero, you could continue using zeros to represent nonadjacent vertices. That might not seem like much of a constraint, but in practice, it can be quite limiting. There are many applications in which having an edge with zero weight is useful. Sometimes even having negative weights is useful and avoiding accidentally setting a weight to zero would be very inconvenient. In this chapter, however, we assume that edge weights are zero or positive.

If you can't use zeros to represent nonadjacent vertices, what can you do? One way to address this limitation is to store a value that represents positive infinity as the edge

weight for nonadjacent vertices. This allows zero weight edges and negative weights, if needed. Infinite weight is something like being infinitely distant—or unconnected. How do you represent infinity in a finite amount of memory? There are several ways, but like the way rectangular bounds and the query circle radius were managed in Chapter 12, "Spatial Data Structures," you can use the convenient mechanism Python offers. In the `math` module, a constant called `inf` is defined that behaves like positive infinity (+∞). If it is compared with any other integer or floating-point number, it will always be greater. The only number it is equal to is itself.

Using positive infinity to represent nonadjacent vertices means you would fill the initial adjacency matrix with that value for empty graphs. Sometimes this is called a **weighted adjacency matrix** because all cells have edge weights, but only the finite ones are considered adjacent. Alternatively, in a hash table representation of the adjacency matrix, any key not in the hash table would have a weight of positive infinity, and only the adjacent vertex pairs would have finite weights stored. In this book, we implement the weighted graph class using a hash table.

Python Code

We don't have to change much in the implementation of the `WeightedGraph` class compared to the `Graph` class described in Chapter 14. Only the edges have changed. Listing 15-1 shows the revised parts.

The weighted graph needs a `Heap` data structure, specifically the minimum (or ascending) heap from Programming Project 13.2. We also need the `math` module for the positive infinity constant. The `Vertex` class is the same because the changes apply only to the edges of the graph. The constructor is included in Listing 15-1 even though it is identical to that of the `Graph`.

The first method that needs a new definition is the one for adding edges. It must now take a third parameter for the weight of the new edge. As before, `addEdge()` starts by validating the vertex indices, `A` and `B`, passed by the caller and verifies that the edge links distinct vertices. Then it stores the weight, `w`, in the `_adjMat` cell for the vertex pair in both directions. In the unweighted `Graph`, it just stored a 1 in those cells.

The next method, `hasEdge()`, changes to first test for the presence of the vertex pair among the keys of the adjacency matrix (instead of using Python's `get()` method for hash tables to supply a default value when the key is missing). It also compares any value stored in the matrix with `math.inf`, positive infinity. That comparison might not be needed in many applications because it will make a difference only if the caller stores `math.inf` as the weight of some edge. That might be done, for example, to remove an edge from the graph. If the comparison were left out of `hasEdge()`, however, then the method would return `True` for edges with infinite weight.

The final method, `edgeWeight()`, is new. We need a method that returns the weight of a given edge. It checks for valid vertex indices and then returns the weight stored in the adjacency matrix if the vertex pair is a key in the adjacency matrix. Otherwise, it returns positive infinity.

That's all that needs to be changed in the basic representation. A few other changes like including the weights in the output of the `print()` method are helpful. Now we can focus on the algorithms that use weights.

LISTING 15-1 The Basic `WeightedGraph` Class

```
from project_13_2_solution import Heap       # Minimum heap
import math

class Vertex(object):                        # A vertex in a graph
…                                            # same as for Graph

class WeightedGraph(object):                 # A graph containing vertices and edges
   def __init__(self):                       # with weights.
      self._vertices = []                    # A list/array of vertices
      self._adjMat = {}                      # Hash table maps vertex pairs to weight

…                                            # skipping shared definitions with Graph

   def addEdge(                              # Add edge of weight w between two
         self, A, B, w):                     # vertices A & B
      self.validIndex(A)                     # Check that vertex A is valid
      self.validIndex(B)                     # Check that vertex B is valid
      if A == B:                             # If vertices are the same
         raise ValueError                    # raise exception
      self._adjMat[A, B] = w                 # Add edge in one direction and
      self._adjMat[B, A] = w                 # the reverse direction

   def hasEdge(self, A, B):                  # Check for edge between vertices A & B
      return ((A, B) in self._adjMat and     # If vertex tuple in adjMat
              self._adjMat[A, B] < math.inf) # and has finite weight

   def edgeWeight(self, A, B):               # Get edge weight between vertices
      self.validIndex(A)                     # Check that vertex A is valid
      self.validIndex(B)                     # Check that vertex B is valid
      return (                               # If vertex tuple in adjMat, return
         self._adjMat[A, B] if (A, B) in self._adjMat
         else math.inf)                      # the weight stored there otherwise +∞
```

The Weighted Minimum Spanning Tree Algorithm

To find the spanning tree that minimizes the total edge weight, we can set up data structures similar to those used for the unweighted graph. Listing 15-2 shows the code. After validating the starting index, `minimumSpanningTree()` creates an empty subgraph called `tree` to hold the result. It then builds an array to map vertex indices from the graph to

those in the tree, `vMap`. We need that because we will be adding vertices in a different order to the tree as the algorithm discovers the edges with lowest weight. It also serves to identify which vertices have been copied (mapped) into the `tree`.

LISTING 15-2 The `minimumSpanningTree()` Method of `WeightedGraph`

```
class WeightedGraph(object):     # A graph containing vertices and edges
…

   def minimumSpanningTree(      # Compute a spanning tree minimizing edge
         self, n):               # weight starting at vertex n
      self.validIndex(n)         # Check that vertex n is valid
      tree = WeightedGraph()     # Initial MST is an empty weighted graph
      nVerts = self.nVertices()  # Number of vertices
      vMap = [None] * nVerts     # Array to map vertex indices into MST
      edges = Heap(              # Use min heap for explored edges
         key=weight,             # Store (A, B) vertex pair & weight in
         descending=False)       # each heap item
      vMap[n] = 0                # Map start vertex into MST
      tree.addVertex(self.getVertex(n)) # Copy vertex n into MST
      while tree.nVertices() < nVerts: # Loop until all verts mapped
         for vertex in self.adjacentVertices(n): # For all adjacent
            if not vMap[vertex]: # vertices that are not mapped,
               edges.insert(     # put weighted edges in heap
                  ((n, vertex), self.edgeWeight(n, vertex)))
         edge, w = (             # Get first edge and weight, if one exists
            (None, 0) if edges.isEmpty() else edges.remove())
         while (not edges.isEmpty() and  # While there are more edges
                vMap[edge[1]] is not None): # and current edge in MST,
            edge, w = edges.remove()        # go on to next edge
         if (edge is None or     # If we didn't find an edge or it goes
             vMap[edge[1]] is not None): # to a mapped vertex
            break                # there are no more edges to be added
         n = edge[1]             # Otherwise get new vertex and
         vMap[n] = tree.nVertices() # map it into MST
         tree.addVertex(self.getVertex(n)) # copy it into MST
         tree.addEdge(           # Add weighted edge to MST mapping
            vMap[edge[0]], vMap[edge[1]], w) # vertex indices
      return tree                # Return the minimum spanning tree

def weight(edge): return edge[1] # Get weight from edge tuple in heap
```

Next, an empty heap is constructed to keep the edges in partially sorted order so that it's quick to get the lowest weight edge. The items we insert on the heap of `edges` are tuples of an edge—a vertex pair—and its weight. We use the `key` parameter of the `Heap` class to

specify the function that extracts the weight from each tuple. The `weight()` function is shown at the end of Listing 15-2. It returns the second element of the tuple.

With the output subgraph (tree), the vertex map, and the edges heap built, the main part of the algorithm begins. Initially we map the starting vertex in the input graph, `n`, to vertex 0 in the output tree and add that vertex object to the tree. If the input graph happened to be a single vertex graph, that's all that would be needed because there can be no edges in the output tree.

The main `while` loop iterates until the output `tree` has as many vertices as the input graph. You'll see what happens if the graph has more than one connected component a little later. Inside the loop, we begin with the start vertex, `n`, find all its adjacent vertices, and look to see if each adjacent vertex has been mapped into the `tree`. At the start, the only mapped vertex is the initial `n`, but on subsequent passes through the loop, `n` will be the last vertex added to the output tree. That's why we must check every adjacent `vertex` using the `vMap` array.

Adjacent vertices that have *not* been mapped mean that the corresponding edge leads to the fringe and should be inserted in the heap. The call to `edges.insert()` puts the tuple of the edge and its edge weight in the heap. The heap knows where to place it according to the `weight` key defined when it was constructed.

Next, the method takes the first edge out of the heap or sets `edge` to `None` if the `edges` heap is empty. The inner `while` loop checks whether this `edge` connects to a vertex that is already mapped into the output tree, by looking at the value of `vMap[edge[1]]`, the mapping for the adjacent vertex. How do we know to examine the vertex at `edge[1]` instead of the one at `edge[0]`? The reason is that we add edges to the `edges` heap only where the second vertex is on the fringe and the first vertex is already in the finalized tree.

If the `edge` goes to a vertex already in the tree and there are more edges in the heap, we take the next lowest weight edge from the heap by calling `edges.remove()` in the inner loop. After the inner loop exits, if no edge was found (`edge` is `None`) or we only found edges connecting within the `tree` (`vMap[edge[1]]` is not `None`), then we've run out of edges to follow in expanding the tree. In that case, it's time to break out of the outer `while` loop because we have built the minimum spanning tree of the connected component containing the start vertex. The returned `tree` will have fewer vertices than the input graph.

After verifying that we did find an edge leading to an unmapped vertex, we set `n` to be that new vertex, `edge[1]`, map it into the output tree, add the vertex to the tree, and add the edge that led to it. Each of the vertices in the `edge` must be mapped to their new indices in the output tree using `vMap`. The next pass through the main `while` loop will explore edges extending from `n`.

When control exits from the main `while` loop, the minimum spanning tree is complete (for the connected component that includes the starting vertex). As you can see, adding weights to the minimum spanning tree algorithm makes it more complicated than the depth-first traversal that was used for the unweighted graph. Having the heap and vertex mapping array available, however, reduces the complexity of the code.

The WeightedGraph Visualization tool can display each of the steps in this algorithm. Try creating a simple graph, selecting a starting vertex, and using the step button, ▶|, to see how each operation happens.

Figure 15-9 shows the WeightedGraph Visualization tool during the computation of the minimum spanning tree of the graph shown in Figures 15-1, 15-2, and 15-3. At this point in the processing, vertices A, B, C, and D have been added to the output tree. There are a lot of data structures on the display, including

- The array for the input graph's `_vertices` (at the upper right)
- The hash table for the input graph's `_adjMat`, adjacency matrix (which is collapsed in Figure 15-9 to show other variables)
- The `vMap` array mapping input vertices to their index in the output tree (at the upper right next to the `_vertices` array)
- The output minimum spanning `tree` (shown by a curved arrow starting at the lower left pointing to the first vertex, highlighted rings on the vertices that have been mapped into the tree, and thicker, highlighted edges that have been added to the tree)
- The `edges` heap shown as a simple, ordered array at the bottom instead of a heap tree (Showing the heap as a tree would take more room, and placing the items in standard heap ordering would make their relationships less clear.)

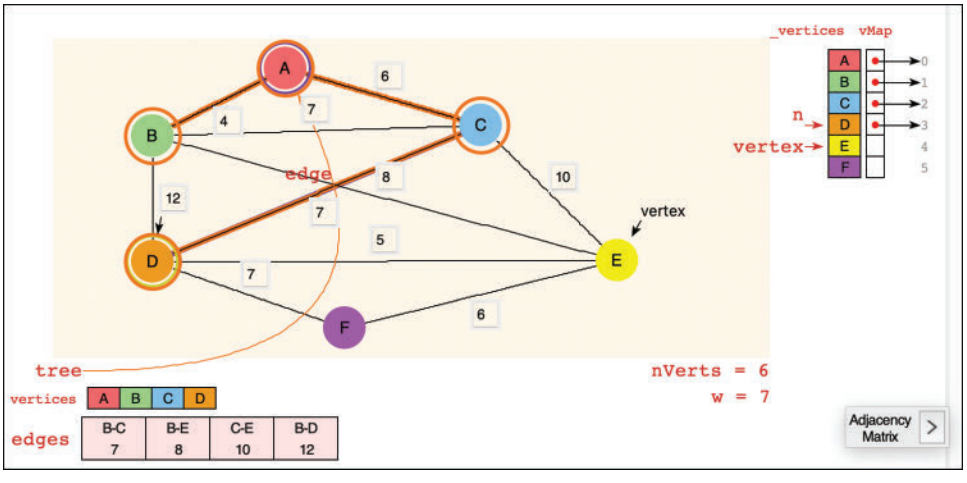


FIGURE 15-9 The Visualization tool computing a minimum spanning tree

The quantity of data and their relationships are complex. Try stepping through the algorithm, watching the incremental changes to the different structures, and predicting what will change at each step before selecting the button to see the outcome.

The Shortest-Path Problem

Perhaps the most commonly encountered problem associated with weighted graphs is that of finding the shortest (lowest-weight) path between two given vertices. The solution to this problem is applicable to a wide variety of real-world situations: planning travel routes, laying out integrated circuits, project scheduling, and more. It is a more complex problem than we've seen before, so let's start by looking at a (somewhat) real-world scenario in the same mythical country of Turala.

Travel by Rail

This time you're concerned with railroads rather than network connections. While Turala was a little behind the times in terms of communication infrastructure, the country has a wonderful rail system. You want to find the fastest route from one city to another to plan for a race across the country.

Finding the fastest route is one variant of the **shortest-path problem**. In this situation you're interested in the shortest travel times, but you could look for the shortest-distance or lowest-fare route. In weighted graphs, *shortest* doesn't necessarily mean shortest in terms of time or distance; it can also mean cheapest, lowest emission, or best route by some other measure.

The time to travel between any two cities by train can vary. That's due to the track conditions and changing weather; heavy rain, snow, or fog means the trains travel slower. The railroads post the travel times daily, but they only post them at the station for the trains leaving from there. Your team will have to get the conditions on the day of the race.

Possible Routes

To plan your route, you need to know what options are available. The first step is to outline the possible train rides your team might take. You would likely construct a map of the cities in graph form like the one at the left of Figure 15-10. There are several possible routes between any two cities. For example, to take the train from Blum to Kay, you could go through Dahl, or you could go through Cerf and Gray, or through Dahl and Gray, or you could take several other routes.

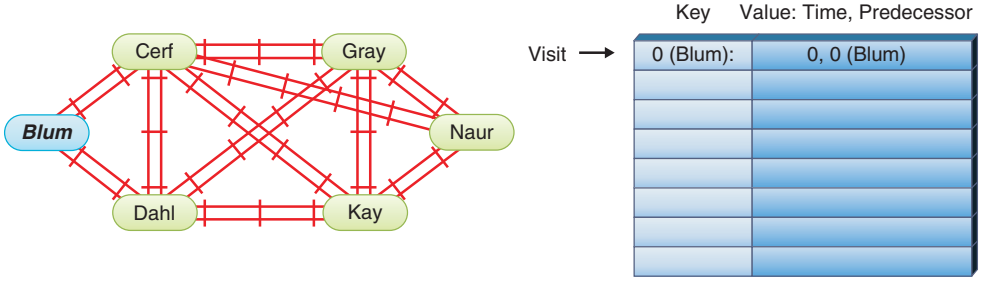


FIGURE 15-10 Train travel in Turala

Note the graph in Figure 15-10 doesn't have travel times (or fares or schedule). All you know at this point is that the race will start in Blum, so that's why it's highlighted.

A Directed, Weighted Graph

The travel times between cities can be different depending on the direction of travel. Trains going uphill might proceed more slowly than those going downhill. Planes flying against the jet stream have lower ground speed than those flying with it. To model differences like these, we use directed graphs where the weight for traveling from vertex J to vertex K is different than the weight from K to J. The Turalan railroad has only single-track lines, so you can go in only one direction between any two cities at any point in time. In your solution, you will only note the travel times going away from Blum because you don't need to plan a round trip. In the case of finding a lowest-fare route, you could use a undirected graph if the fare was the same in both directions.

Dijkstra's Algorithm

The solution we show for the shortest-path problem is called Dijkstra's algorithm, after Edsger Dijkstra, who first described it in 1959. Interestingly, the method not only solves for the shortest path to a destination city, but it can also solve for the shortest path to any other destination, as you shall see.

Agents and Train Rides

On the day of the race, you learn that the finish line is in Naur. To see how Dijkstra's algorithm works, your team is going to operate the way a computer does, looking at one piece of information at a time, so we assume that you are similarly unable to see the big picture (as in the preceding section). That means your team needs to send agents to stations like the surveying teams for the network routing.

At each city, the stationmasters can tell you how long it will take to travel to the other cities that you can reach directly (that is, in a single ride, without passing through another city). Alas, they cannot tell you the times to cities further than one ride away. You keep a notebook, like the one at the right of Figure 15-10, with a row for each city. You hope to end up with rows filled in with the shortest time from your starting point to that city (plus a little more information that you will need at the end).

The First Agent: In Blum

Eventually, you're going to place agents in every city (at least those needed to get to Naur). These agents must obtain information about travel times to other cities. You yourself are the agent in Blum.

All the stationmaster in Blum can tell you is that today it will take 22 minutes to get to Cerf and 16 minutes to get to Dahl. You write this information in your notebook, as shown in Table 15-2.

Table 15-2 Step 1: Notebook with Information from the Agent at Blum

To	Time	Via
Cerf	22	Blum
Dahl	16	Blum

The table lists all the cities for which you have some information about the travel time. The Via column records what city you came from to get that time. You'll see later why this is good to know. What do you do now? Here's the rule you follow:

> ***Rule***
>
> Always send an agent to the unvisited city whose overall route from the starting point (Blum) is the shortest.

Notice that this is not quite the same rule as that used in the minimum spanning tree problem (the network installation). There, you picked the least expensive *single link* (edge) from the connected cities to an unconnected city. Here, you pick the least expensive *total route* from Blum to a city with no agent. At this particular point in your planning, these two approaches amount to the same thing because all known routes from Blum consist of only one edge. As you send agents to more cities, however, the routes from Blum will become the sum of several direct edges.

The Second Agent: In Dahl

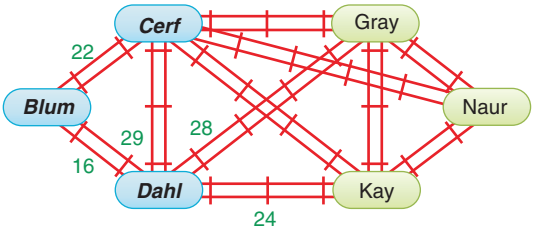
The shortest route from Blum is to Dahl, at 16 minutes. So, you send half your team to Dahl, where one will act as the agent and others will be ready for later tasks. When they arrive, they message you that the Dahl stationmaster says today's trains will take 29 minutes to reach Cerf, 28 minutes to reach Gray, 24 minutes to reach Kay, and 15 minutes to return to Blum. That last travel time confuses your agents, until they realize the stationmaster doesn't know where they came from and is just trying to be helpful.

Now you can update your notebook. Adding up the time you already spent getting to Dahl with the times to reach Gray and Kay, you can make two new entries in the notebook as shown in Table 15-3. You now know that you can reach Gray in 44 minutes and Kay in 40 minutes from Blum. You also received information about Cerf. If you travel to Cerf via Dahl, it takes 16 + 29 = 45 minutes. That's longer than the direct travel from Blum, so there's no need to change the entry for Cerf.

Table 15-3 Step 2: Notebook with Information from the Agents at Dahl

To	Time	Via
Cerf	22	Blum
Dahl	16	Blum
Gray	44	Dahl
Kay	40	Dahl

It helps to see how this information looks in graph form. Figure 15-11 shows the weights on the routes (edges) that have been learned so far. You have agents in Blum and Dahl, and a quick look among the other cities in your notebook shows that Cerf is the next shortest route. Following the rule, you and the other agents in Blum go to Cerf to learn what conditions are like there. That's what the "Visit" pointer is showing; it's time to visit Cerf.



Visit →

Key	Value: Time, Predecessor
0 (Blum):	0, 0 (Blum)
3 (Gray):	44, 2 (Dahl)
1 (Cerf):	22, 0 (Blum)
4 (Kay):	40, 2 (Dahl)
2 (Dahl):	16, 0 (Blum)

FIGURE 15-11 Following step 2 in the shortest-path algorithm.

The table on the right of Figure 15-11 shows the same information that's in your notebook but in a different format. The cities are in a different order, it includes the start city of Blum, and there are some extra numbers along with the city names. The reasons for those are explained shortly.

After you've placed agents in a city, you can be sure that the route taken by them to get to that city is the fastest route. Why? Consider the present case. If there were a faster route than the direct one from Blum to Cerf, it would need to go through some other city. The only other way out of Blum by train is to Dahl, which you explored first. You already found that the Blum–Dahl–Cerf route takes more time than the direct route to Cerf. If there were a route via Gray, Kay, or some other city back to Cerf, it would have to take at least 40 minutes (the time needed to reach Kay), so there's no faster way via Dahl. Hence, you conclude with certainty that you know the shortest time to all the visited cities (Blum, Dahl, and now Cerf).

Based on this analysis, you decide that from now on you won't need to update the entries for the time from Blum to Cerf or Blum to Dahl. You won't be able to find faster routes, so you can cross them off the list.

Three Kinds of Cities

As in the minimum spanning tree algorithm, the cities for the shortest-path algorithm are divided into three categories:

1. Cities in which you've installed an agent (for which you've already found the shortest path).
2. Cities with known travel times from cities with an agent; they're on the fringe.
3. Unknown cities.

At the beginning of step 2, Blum and Dahl are category 1 cities because they have agents there. Based on the travel times learned in Dahl, you sent agents to Cerf, moving that city to category 1. Those three cities form a tree consisting of paths that all begin at the starting vertex (Blum) and that each end on a different destination vertex. This is not the same

tree, of course, as a minimum spanning tree. It's the shortest-path tree because you've concluded you now know the fastest routes to each city.

Two other cities have no agents, but you know some of the times to reach them because you have agents in adjacent category 1 cities. You know the time to go from Blum to Gray is at most 44 minutes and to Kay is at most 40 minutes. The information in your notebook means that Gray and Kay are category 2 (fringe) cities.

You don't know anything yet about Naur; it's an "unknown" city (from the perspective of this algorithm). As in the minimum spanning tree algorithm, Dijkstra's shortest-path algorithm moves cities from the unknown category to the fringe category, and from the fringe category to the tree, as it goes along.

The Agents in Cerf

With you and your other agents now in Cerf, you can cross out Cerf and Dahl in your list and get information from the stationmaster there. Cerf is a busy station because five rail lines connect to it. Your agents learn today's trains take 34 minutes to Gray, 65 minutes to Naur, 26 minutes to Kay, 24 minutes to Dahl, and 18 minutes to Blum. You add the times to your notebook, for the unvisited cities to get Table 15-4.

Table 15-4 Step 3: Notebook with Information from the Agents at Cerf

To	Time	Via
~~Cerf~~	~~22~~	~~Blum~~
~~Dahl~~	~~16~~	~~Blum~~
Gray	44	Dahl
Kay	40	Dahl
Naur	87	Cerf

The Blum–Cerf–Gray route takes 22 + 34 = 56 minutes. That's more than the Blum–Dahl–Gray route of 44 minutes, so the entry from before remains. Similarly, Blum–Cerf–Kay takes 22 + 26 = 48 minutes, which is slower than Blum–Dahl–Kay at 40. You do make a new entry for Naur (so it is no longer in the unknown category).

The times to return to Blum or Dahl are less than what it took to go to Cerf from those cities. Shouldn't that affect the route plan? It might be that Cerf is higher in elevation so going back is faster, but that's not going to help you pick the fastest route *from* Blum *to* Naur. The crossed-out entries don't need updates.

You now have the situation shown in Figure 15-12. The information learned in Cerf shows that the next city to visit is Kay with the total time from Blum being 40 minutes. It's also clear that the fastest route to Kay is the Blum–Dahl–Kay route because you noted the predecessor of Kay was Dahl in the notes. You now have all of the cities in category 1 or category 2; none remain in category 3.

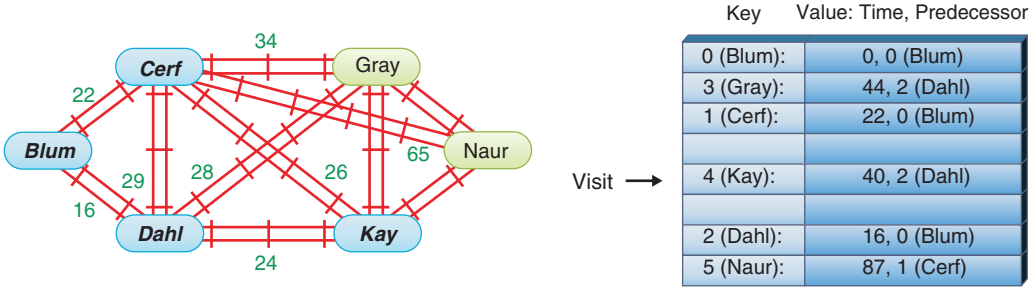


FIGURE 15-12 Following step 3 in the shortest-path algorithm.

Because you have Naur in your notebook, can you stop sending agents out? Some members of the team are eager to finish the surveying. You do have one route to Naur, but it's not certain yet that you know the very fastest route. You tell your team in Dahl to split up, move half to Kay, and press on.

The Agents in Kay

When you have agents in Kay, they quickly report the following travel times: Gray 25 minutes, Naur 36 minutes, Cerf 29 minutes, and Dahl 24 minutes. Those last two are cities you have already crossed off, so only the first two matter. Adding the times to the shortest time to get to Kay of 40 minutes (via Dahl) means you could get to Gray in 65 minutes and Naur in 76 minutes. You already know a route to Gray that takes 44 minutes (via Dahl), so the new route is no good. The route to Naur, however, is shorter. (It was worth pressing on!) You update your notebook to show what's in Table 15-5.

Table 15-5 Step 4: Information from the Agents at Kay

To	Time	Via
~~Cerf~~	~~22~~	~~Blum~~
~~Dahl~~	~~16~~	~~Blum~~
Gray	44	Dahl
~~Kay~~	~~40~~	~~Dahl~~
Naur	~~87~~ 76	~~Cerf~~ Kay

You are faced with another choice of ending the process with the newer, better route to Naur. You trust in the process, however, and decide to continue. The shortest route to an unvisited city is now to Gray (via Dahl) at 44 minutes. This corresponds to the upper graph of Figure 15-13.

You're closing in on the goal. You send the remaining team in Dahl to Gray because that was the fastest way to get to Gray based on the notes. (A sharp observer might note that you could get agents from Kay there faster, but that's not really part of the computer algorithm, which simply has to choose which vertex to explore next.)

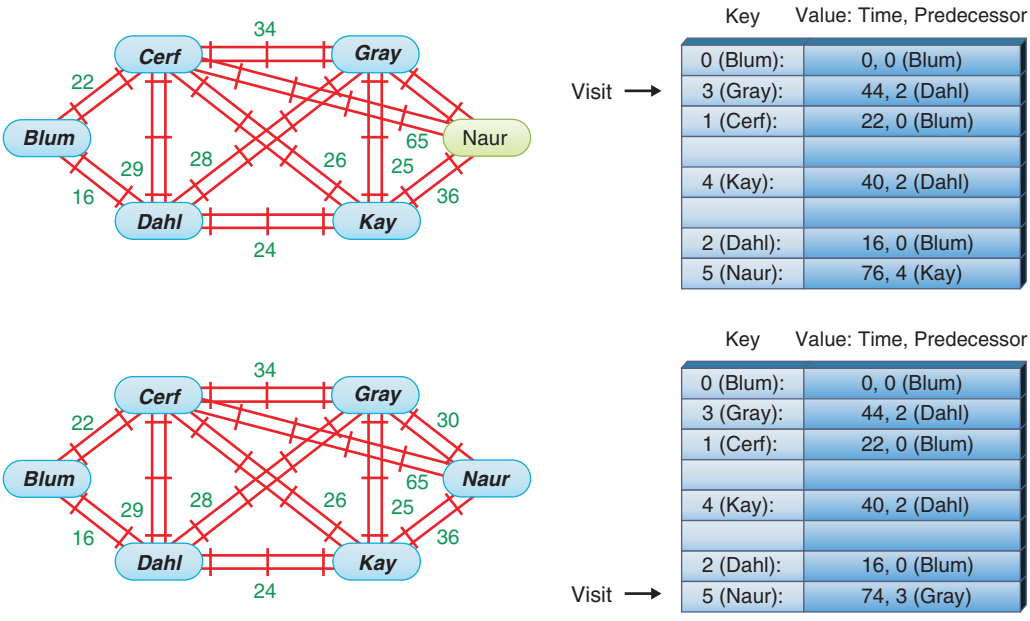


FIGURE 15-13 Following steps 4 and 5 in the shortest-path algorithm.

The Last Agents in Gray

Your (now very experienced) agents report the only route that matters from Gray: it will take 30 minutes to reach Naur from there. Because it takes 44 minutes to get to Gray (via Dahl), you now could get to Naur in 74 minutes. You make another revision to your notebook and end up with the graph and table shown at the bottom of Figure 15-13. You send an agent to Naur, just to confirm the time.

When there's an agent in every city, you know the best times from Blum to every other city. You're done. With no further calculations, the last entries in your notebook show today's fastest routes from Blum to all other visited cities. Your team now has the optimal route for the Blum to Naur race. That route plus the other fastest routes form a tree rooted at the starting point (not to be confused with the minimum spanning tree).

Note that you don't have the fastest route to any cities not yet visited when you stop updating the notebook. If Turala had a dozen more cities, you wouldn't know for sure how long it takes to get to them. To get all the fastest routes, you would need to visit every city. You have, however, found the fastest way to get from Blum to Naur and know that any other team who simply chose the direct train from Cerf to Naur would lose the race!

This narrative has demonstrated the essentials of Dijkstra's algorithm. On the surface, it seems more like the breadth-first search than the depth-first one, expanding the shortest-known path away from the starting point. The key points are

- Each time you send agents to a new city, you use the new information provided by them to revise your list of times. Only the shortest route (that you know about) from the starting point to a given city is retained, including the previously visited city that achieved that time.

- You always send your next agents to the unvisited city that is closest (in time) to the starting point. That is different from the fastest single train trip (edge) from any city with an agent, as was selected in the minimum spanning tree.

In the terms of graphs, Dijkstra's algorithm adds one vertex to its output subgraph on each iteration. The vertex added has the lowest total edge weight on the path back to the starting vertex. Because that path back is unique (each vertex has a unique preceding vertex), the subgraph is always a tree. If there are multiple, equal weight paths to a vertex, it will choose a path with fewer edges due to the breadth-first style exploration.

Finding Shortest Paths Using the Visualization Tool

Let's see how Dijkstra's algorithm looks using the WeightedGraph Visualization tool. If you happen to have the tool where you can launch it using the command line, you can run

```
python3 WeightedGraph.py -TuralaSP
```

This command creates the graph using two-letter prefixes for the city names and the weights shown in the previous examples. The vertex positions are random, so if you want the graph in the tool to look like the examples, you will probably have to move them around to look something like Figure 15-14.

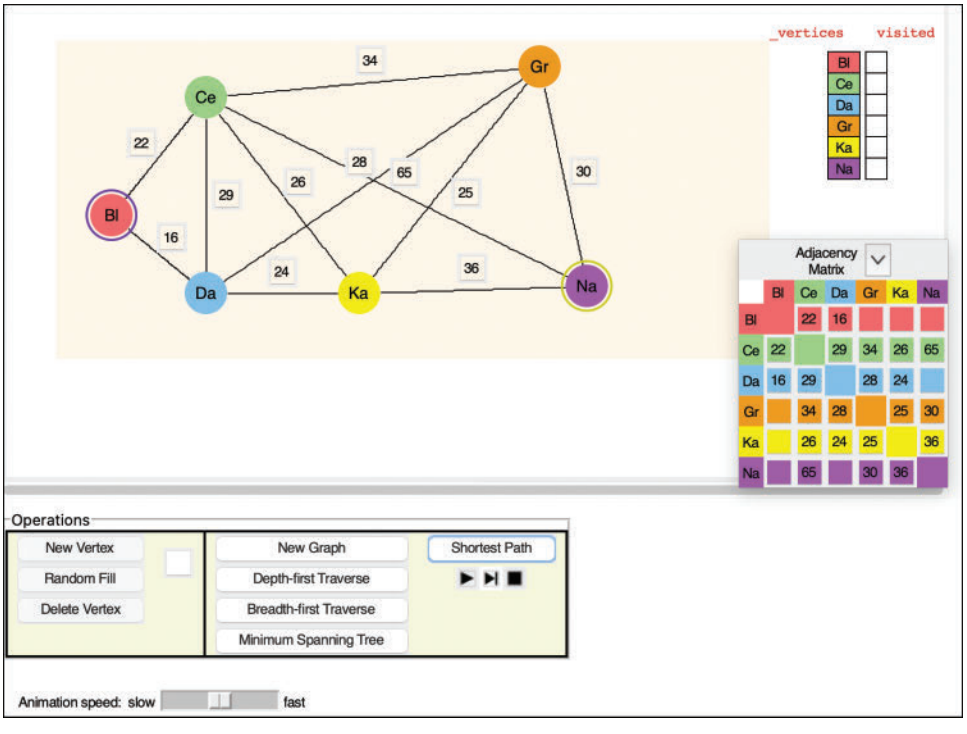


FIGURE 15-14 The railroad scenario in WeightedGraph Visualization tool

Two vertices must be selected to calculate a shortest path: a start and an end vertex. The Visualization tool lets you select the start vertex with the first mouse button, just as with the minimum spanning tree. You select the end vertex by holding the Shift key or pressing a different mouse button. The vertices are highlighted with rings like those in Figure 15-14 for Bl(um) and Na(ur).

Note that the Visualization tool shows the graph as undirected. Dijkstra's algorithm works on both kinds of graphs, and the results are the same (assuming only positive weights and the weights are those that are needed for the final direction of travel). Showing only one edge between a pair of vertices keeps the display less cluttered.

After the graph is configured and the endpoints selected, the Shortest Path operation is enabled. Try pressing the button and stepping through the process. We look at some key steps next, focusing on the "notebook" used for tracking the shortest paths and the visited vertices before looking at the details of the implementation.

The Costs Notebook

The Visualization tool draws the notebook entries a little differently than the preceding figures. At the start of the shortest-path algorithm, it creates a structure at the bottom of the display called `costs` with a single entry for Bl(um), as in Figure 15-15. The top of the entry is the vertex name along with its matching-colored background. The bottom of the entry is the total time (weight, distance, cost, and so on) to reach that vertex, along with the vertex visited just before. These are shown as tuple like `(0, 'Bl')`, where the first component is the total time and the second is the previous or *parent vertex*. For this starting vertex, there is no previous one, so it's marked as being Bl(um).

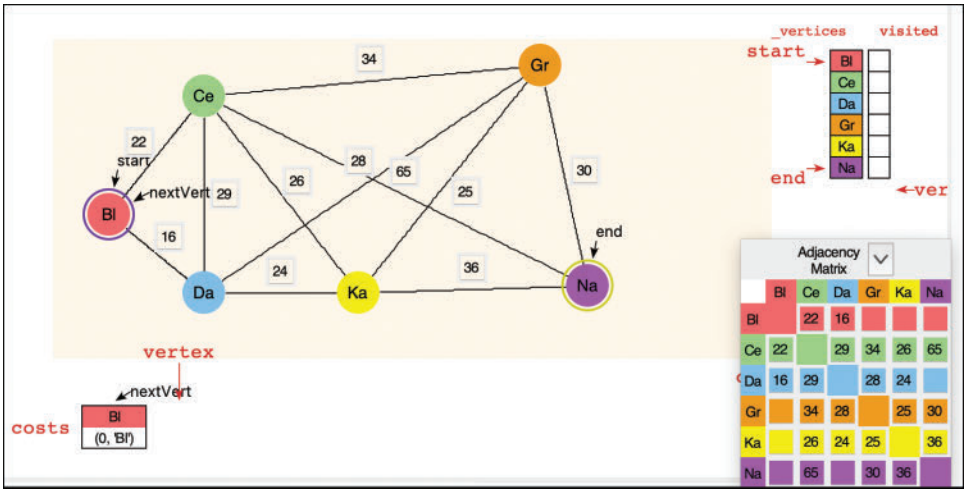


FIGURE 15-15 The first entry in `costs` for the shortest path to Naur

In the upper right of the display, a `visited` array tracks which vertices have been finalized with their shortest path (put in category 1). The next step will place a 1 in the cell next to

Bl(um) to indicate that it—and it alone—falls in this category at this stage. The total time for category 1 vertices in the `costs` structure is that for the final, shortest path.

The display has many different arrows indicating variables used in different parts of the program, which can be confusing. Several of them show the same information but in different ways. For example, the `start` and `end` variables indicate the start and end vertices in both the `_vertices` array and the graph layout. The `nextVert` arrow shows the next vertex to be processed in both the `costs` structure and the graph layout. Sometimes these arrows and other display elements overlap each other, making them harder to read.

When the algorithm chooses the second city to visit, Da(hl), the situation looks like Figure 15-16. The `costs` structure now has three entries, showing the shortest paths found so far for the three vertices, just like the notebook in Table 15-2. All of them have Bl(um) as the preceding vertex. The `visited` array shows that only Bl(um) and Da(hl) have been visited, and hence are in category 1 with their final shortest paths.

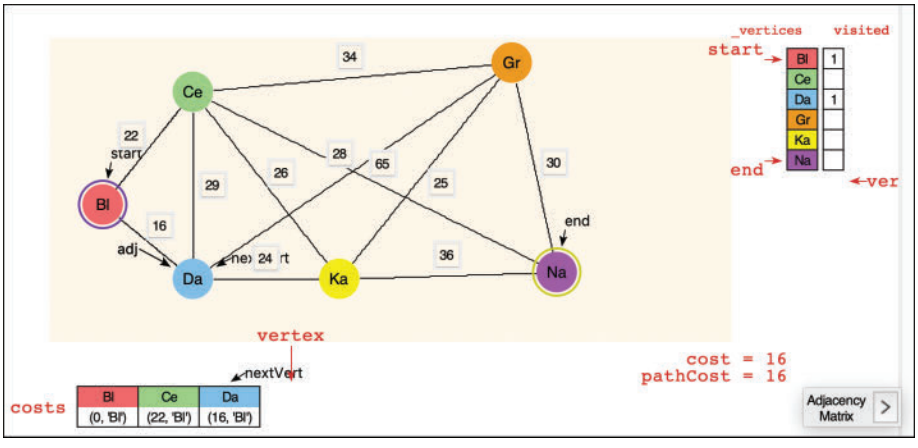


FIGURE 15-16 After selecting Da(hl) to visit in finding the shortest path to Naur

The figure also collapses the adjacency matrix to show the variables `cost` and `pathCost`, which might otherwise be hidden. The edge weights remain visible in the graph layout, so the adjacency matrix information is still available.

The process continues visiting cities and updating the structures. After skipping over the visit to Ce(rf), Figure 15-17 shows the contents of the `costs` and `visited` structures when Ka(y) is visited. It shows that both Gr(ay) and Ka(y) were reached from Da(hl). Ka(y) has a shorter total path cost of 40 compared to Gr(ay)'s cost of 44, so it is chosen as the next vertex to visit. The `visited` array shows that Ka(y) has now been put in category 1. Ce(rf) was visited in the previous stage, but the path through Da(hl) took less time.

Both Gr(ay) and Na(ur) are unvisited at this stage, but you do have estimated times to reach them of 44 and 87, respectively. Their parent vertices are different because they were reached along different paths.

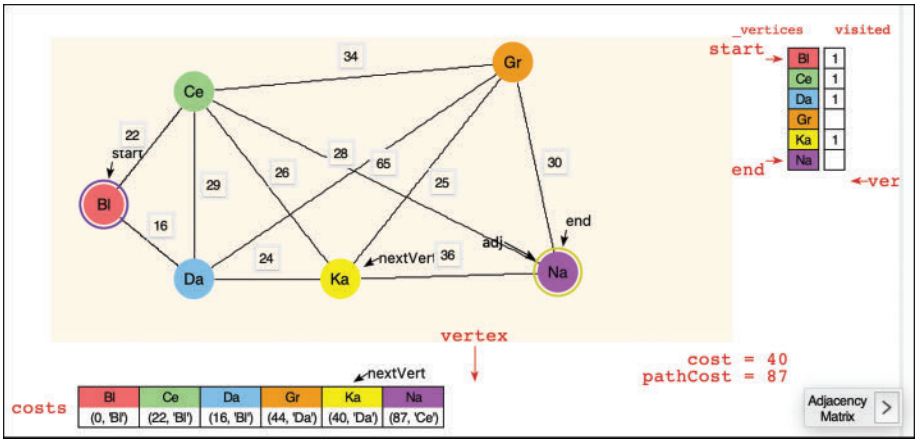


FIGURE 15-17 After selecting Ka(y) to visit in finding the shortest path to Naur

The algorithm selects Gr(ay) for the next visit as shown in Figure 15-18. No new vertices have been added to the `costs` structure, but one has been changed. The total time taken to reach Na(ur) has been updated from 87 to 76. That's due to the new information found when visiting Ka(y) that Na(ur) could be reached in 36 minutes. Unlike a paper notebook, where you might put a line through an old entry, the previous estimate is simply erased from the entry for Na(ur), and the new entry showing the path from Kay replaces it.

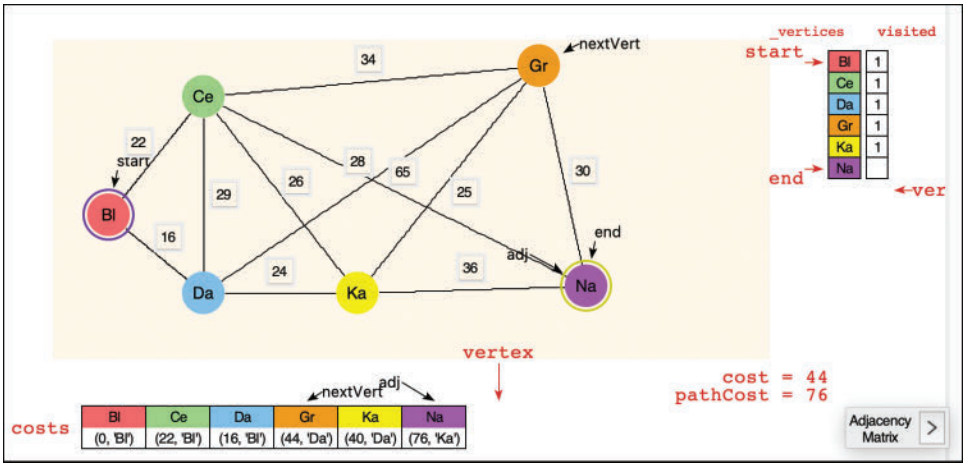


FIGURE 15-18 Visiting Gr(ay) in finding the shortest path to Naur

The final shortest path to Naur is updated at the next stage, changing the total time to 74 with Gr(ay) as the parent vertex. With Naur in the `visited` array, we're done updating the `costs` structure. What's left is following the "breadcrumbs" stored in that structure to determine the best path from Blum. We'll see how to do that and figure out what we

should use to represent the `costs` notebook and how to deal with an end vertex in a different connected component in the next section.

Implementing the Algorithm

What kind of data structure should we use for the notebook? Dijkstra's algorithm needs something that records total weights (times) and predecessors for individual vertices of the graph (cities). What options do we have?

Among the data structures you've seen so far, you could use nearly all of them to store a record holding a path weight and vertex index for the predecessor. These records would be identified by an integer key, the vertex index for the destination. Using an array would give you the fastest access by that key, O(1). Sorted lists, sorted arrays, trees, and heaps could be an option because that allows for quickly finding the lowest weight path. Using those structures, however, means finding the records to be updated for each newly discovered edge weight will take longer than O(1); probably O(log N) or O(N), where N is the number of vertices you're tracking. Like the analogy with the agents sent to the train stations, you must update several records for each vertex visited (all of its adjacent vertices minus the parent vertex), so you really need to keep the time to find each record as O(1).

If you use an array, you need an array that holds all the vertices. That's not much memory for small graphs but becomes more significant with huge ones. Finding the shortest path in Turala only needed an array that could hold six items. If you were finding the shortest route between two road intersections in North America, however, there are more than a million vertices to consider. Do you really want to create an array for millions of vertices when the final result route needs to visit only a couple of dozen?

If the purpose of applying Dijkstra's algorithm is to find the shortest path from a starting vertex to *all other vertices* in the connected component, then it makes sense to allocate an array for all the vertices. They will all be visited as part of the process. If the purpose is to find the shortest path between *two specific vertices*, however, it makes more sense to use a hash table. After you visit the destination vertex, the algorithm can stop and return the shortest (lowest weight) path. There will be entries for all the vertices that are adjacent to a vertex on the shortest path, but that could be millions fewer.

As you saw in Chapter 11, hash tables can grow as needed while still providing O(1) search time. Using Dijkstra's algorithm to search for the shortest path between two out of a two million vertex road network is likely to need to explore a tiny fraction of the total number of vertices, maybe tens or hundreds of vertices. A hash table allows you not only to use memory proportional to that tiny fraction of vertices, but it also means that enumerating (traversing) all the current vertices in the table to, say, find the one with lowest total path weight takes time proportional to the tiny fraction. That's not as efficient as keeping all the items in a priority queue or heap where the time to find the shortest takes O(1), but those structures would take more time to find each vertex when updating their total path cost.

You should also consider how to track vertices that have been visited by the algorithm. In the notebook, you crossed off cities (vertices) as they were visited. That suggests that

you could add a flag to each record indicating whether the vertex had been visited or not. With that representation, determining whether vertex K has been visited or not takes a hash table search, and if a record for K is found, checking if its flag is set. That is probably the most memory-efficient way, but you could also keep a separate hash table to store the visited vertices. Doing so means determining the visit status for K only requires seeing whether key K has been inserted in the hash table.

In the Visualization tool, the costs notebook and visited table appear as simple tables or arrays. That is only to simplify their appearance in the tool; we use hash tables in the code.

Python Code

You can implement the shortest-path algorithm in a single method as shown in Listing 15-3. This version will finish as soon as it finds the shortest path between the given `start` and `end` vertices, not continuing to find the shortest paths to all other vertices.

As explained in the previous section, the `shortestPath()` method starts off by creating two hash tables: `visited` and `costs`. The Python `dict` structures used for these hash tables simplify the syntax a little, but you could also use instances of the `HashTable` class introduced in Chapter 11. The `visited` table is initially empty because no vertices have yet been visited. As the algorithm visits vertices, it will insert a 1 in this hash table at the vertex's index.

The `costs` hash table maps vertex indices to the records containing the total path weight to that vertex and the predecessor vertex index to reach it. This holds the notebook contents and is the same as what is shown on the right side of Figures 15-10, 15-11, 15-12, and 15-13. Initially, it holds a record of `(0, start)`, as shown in Figure 15-10. That represents the zero path weight of the zero length path from the starting vertex. The first element of the record (tuple) is the path length, and the second element is the predecessor or parent vertex.

For the starting vertex, there is no predecessor, so we could store `None` in the tuple instead of `start`. The figure shows the record being placed in the first cell of the hash table, but the starting index could hash to any cell.

With these two hash tables set up, we now enter the main `while` loop of the algorithm. The loop condition, `end not in visited`, says it all. The loop continues until the `end` vertex is inserted in the visited hash table. (If the `start` and `end` vertices are not in the same connected component, we'll detect that later and break out of the loop.)

In the loop body, the first thing to do is find which vertex to visit next. The rule in Dijkstra's algorithm is to find the unvisited vertex with the lowest weight path from the start. We set up a `nextVert` and a `cost` variable to search for these. The `cost` variable is initialized to `math.inf` so that any (finite) cost found is considered as a lower-cost path. The first inner `for` loop goes over all the vertices with records in the `costs` hash table. Inside that `for` loop, when it finds a `vertex` that has not yet been visited and the path cost to reach that `vertex` is less than the minimum cost path found so far, it updates `nextVert` and `cost` to use that `vertex` and its path cost (stored in the first element `[0]` of the `costs` record).

LISTING 15-3 The `shortestPath()` Method for `WeightedGraph`

```
class WeightedGraph(object):      # A graph containing vertices and edges
…

  def shortestPath(               # Find shortest path between two vertices,
         self, start, end):       # if it exists, as list of vertex indices
      visited = {}                # Hash table of visited vertices
      costs = {                   # Hash of path costs to vertices including
         start: (0, start)}       # their predecessor vertex
      while end not in visited:   # Loop until we visit the end vertex
         nextVert, cost = None, math.inf # Look for next vertex
         for vertex in costs:     # among unvisited vertices whose cost
            if (vertex not in visited and # to reach is the lowest
                costs[vertex][0] <= cost):
               nextVert, cost = vertex, costs[vertex][0]
         if nextVert is None:     # If no unvisited vertex could be found
            break                 # we cannot get to the end, so exit loop
         visited[nextVert] = 1    # Visit vertex at end of lowest cost
         for adj in self.adjacentVertices(nextVert): # path and
            if adj not in visited: # adjacent, unvisited vertices
               pathCost = (       # Extended path costs weight of adj.
                  self.edgeWeight(nextVert, adj) + # edge plus cost of
                  costs[nextVert][0])  # path so far
               if (adj not in costs or # If reached adj for first time
                   costs[adj][0] > pathCost): # or old path costlier,
                  costs[adj] = (  # update cost to reach vertex
                     pathCost, nextVert) # via extended path

      path = []                   # Build output path from end
      while end in visited:       # Path only contains visited vertices
         path.append(end)         # Append end vertex
         if end == start:         # If we reached the start,
            break                 # we're done. Otherwise go to
         end = costs[end][1]      # predecessor via lowest cost path
      return list(reversed(path)) # Return path from start to end
```

We now know which vertex to visit. Because we put the `start` vertex in the `costs` table outside of the main loop, it will be found by the inner loop and become the first vertex visited. If the inner `for` loop ended without finding an unvisited vertex, that means that we've explored all the vertices connected to the starting vertex without discovering a path to the destination. In other words, if the `nextVert` is `None`, then we can break out of the main loop.

When the `nextVert` is not `None`, we mark it in the `visited` hash table and proceed to process the edges connected to it. The second inner `for` loop steps through all adjacent

vertices of `nextVert`, storing the index in the `adj` variable. Adjacent vertices that have not been visited are examined. The cost of the edge from `nextVert` to the `adj` vertex is added to the path cost of reaching `nextVert`. For the starting vertex, the path cost was set to zero, so no special handling is needed for the first edges. The next `if` statement checks whether `adj` is either a new vertex in the `costs` hash table or the `pathCost` to reach it is lower than the one recorded in `costs`. If it's new or lower, the `costs` hash table entry for the `adj` vertex is updated with the new lower cost and its predecessor vertex.

That takes care of the one rule and update strategy in Dijkstra's algorithm. When the main `while` loop ends, we'll either have the `end` vertex among those `visited` along with its path cost in the `costs` table, or no path could be found. We assume the latter, setting `path` to `[]`, an empty list. The next loop reconstructs the path from the `costs` table, if there is one.

The final `while` loop starts from the `end` vertex and works backward. It appends the `end` vertex to the output `list`. Next, it checks whether that `end` vertex was the `start` vertex. If so, the path is complete, and it breaks out of the loop. If not, it sets `end` to the predecessor vertex index that was stored as the second element of the `costs` table record and continues on. When the loop exits, `path` holds the vertex indices of the shortest path but in reverse order. We use the Python utilities `reversed()` and `list()` to put the list in forward order and return it as a list. When the `end` vertex is not reachable from the `start` vertex, it will not be in the `visited` table. In that case, the final `while` loop does nothing and the empty path is returned.

This implementation is straightforward and fairly efficient. To see how all the different data structures work together to achieve the goal, it helps to run through a few examples, either by hand or using the Visualization tool.

The All-Pairs Shortest-Path Problem

In discussing connectivity in Chapter 14, we wanted to know whether it was possible to travel from Athens to Kamchatka if we didn't care how many stops were made. With weighted graphs you can answer other questions that might occur to you as you wait at the ticket counter: How much will the journey cost? How long will the journey take?

To find whether a trip was possible, you created a connectivity matrix starting from the adjacency matrix. The analogous table for weighted graphs gives the minimum cost from any vertex to any other vertex using multiple edges. This is called the **all-pairs shortest-path** problem.

You might create such a matrix by

1. Making a `shortestPaths()` method based on the `shortestPath()` method that takes a single vertex as its parameter,
2. Changing its main `while` loop to end when all vertices have been visited, and
3. Returning paths and their costs for all visited vertices.

Then you could run `shortestPaths()` starting at each vertex in turn to populate the rows of the all-pairs shortest-paths matrix. With a nondirectional graph, the result matrix would be symmetric, and you could limit the amount of work done by only determining the cost of paths where the start index is less than the end index, but it would still be quite complex.

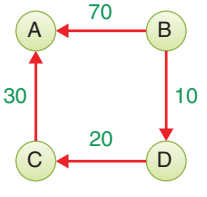
In the end, this could produce a travel timetable for Turala like the one in Table 15-6, if you assume the return travel time is the same as the forward travel times shown in Figure 15-13. The best time found for the trip from Blum to Naur, 74 minutes, is in the table, but it also would find that the Cerf to Naur trip could be done in 62 minutes compared to the 65-minute direct route.

Table 15-6 All-Pairs Shortest-Path Table

	Blum	Cerf	Dahl	Gray	Kay	Naur
Blum	—	22	16	44	40	74
Cerf	22	—	29	34	26	62
Dahl	16	29	—	28	24	58
Gray	44	34	28	—	25	30
Kay	40	26	24	25	—	36
Naur	74	62	58	30	36	—

In the preceding chapter, you saw that Warshall's algorithm was a way to create a table showing which vertices could be reached from a given vertex using one or many steps. Analogous approaches for weighted graphs were published by Robert Floyd and Stephen Warshall separately in 1962. The pattern of moving through the adjacency matrix is the same, but the update strategy is different because you need to sum the edge/path weights rather than doing inclusive OR operations.

Let's learn the Floyd-Warshall algorithm using a simple four-vertex graph. Figure 15-19 shows a weighted, directed graph and its weighted adjacency matrix in two formats.



	A	B	C	D
A				
B	70			10
C	30			
D			20	

Weighted Adjacency Matrix

	A	B	C	D
A	∞	∞	∞	∞
B	70	∞	∞	10
C	30	∞	∞	∞
D	∞	∞	20	∞

Weighted Adjacency Matrix with infinite weights for disconnected vertices

FIGURE 15-19 A weighted, directed graph and its adjacency matrix

The weighted adjacency matrix shows the cost of all the one-edge paths. The format on the right explicitly fills in an edge weight of positive infinity for nonadjacent vertices. The goal is to extend this matrix to show the cost of all paths regardless of length. Any path that doesn't exist will have infinite weight.

For humans, it's easy to see that you can go from B to C at a cost of 30 (10 from B to D plus 20 from D to C). The return trip, from C to B, is not possible (or at least would cost more than anyone with finite resources can afford).

As in Warshall's algorithm, you copy the adjacency matrix and then systematically modify it into a path weight matrix. You examine every cell in every row. If there's a finite weight (let's skip down to row C where there's a 30 in column A), then you then look in column C (because C is the row where the 30 is). If you find a finite entry in column C (like the 20 at row D), you know there is a path from C to A with a weight of 30 and a path from D to C with a weight of 20. From this, you can deduce that there's a two-edge path from D to A with a weight of 50. Figure 15-20 shows the steps when the Floyd-Warshall algorithm is applied to the graph in Figure 15-19.

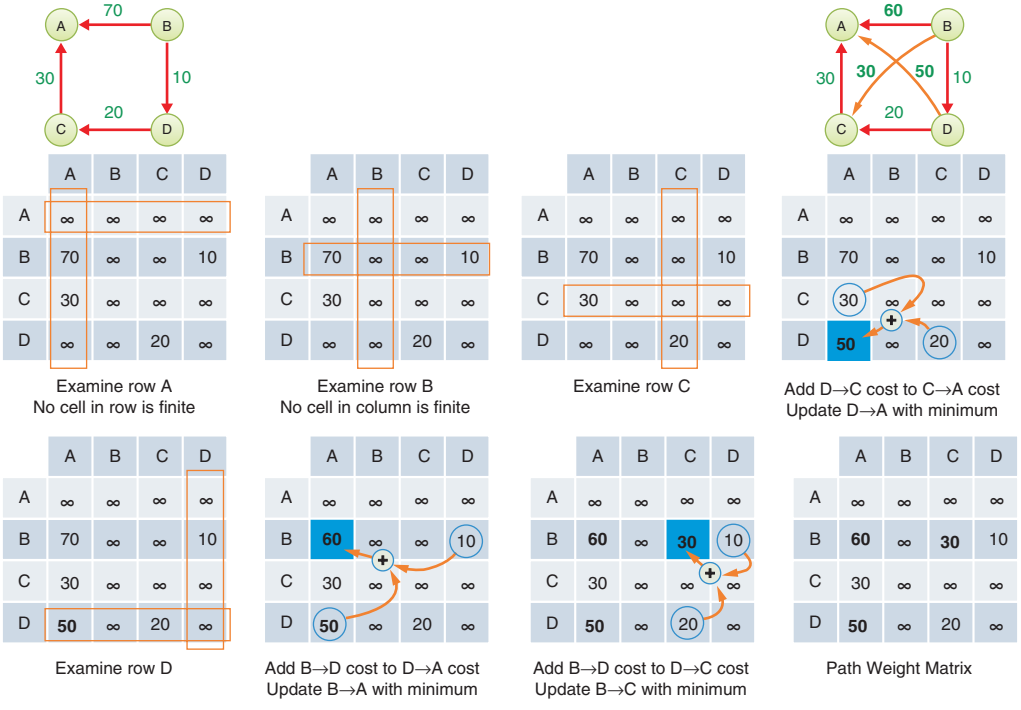


FIGURE 15-20 Steps in the Floyd-Warshall algorithm to create the path weight matrix

Row A has no finite weights, so there's nothing to do there. In row B there's a 70 in column A and a 10 in column D, but there are no finite weights in column B, so the entries in row B can't be combined with any edges ending on B. (Technically, you could add the 70 to infinity, but that gives infinity and you only make changes if it lowers a path weight.)

In row C, however, you find a 30 at column A. Looking in column C, you find a 20 at row D. Now you have C to A with a weight of 30 and D to C with a weight of 20, so you have D to A with a weight of 50.

Row D shows an interesting situation: where the algorithm updates an existing cost. There's a 50 in column A. There's also 10 in row B of column D, so you know there's a path from B to A with a cost of 60. There's already a cost of 70 in this cell. What do you do? Because you want the *shortest* path and 60 is less than 70, you replace the 70 with 60.

The implementation of the Floyd-Warshall algorithm is similar to that for Warshall's algorithm. Instead of simply inserting 1s into the table when a multiple-edge path is found, you add the costs of the two paths and insert the sum, as long as the sum is less than the cost already stored for the combined path. Similar to the Dijkstra algorithm, each cell contains the minimum path weight connecting two vertices that have been found so far. After going through all the rows and columns, the overall shortest-path weights are discovered. We leave the details as an exercise.

Efficiency

So far, we haven't discussed much about the efficiency of the various graph algorithms. The issue is complicated by the two ways of representing graphs: the adjacency matrix and adjacency lists, and the fact that there are two different measures of graph size.

If an adjacency matrix is used, the algorithms we've discussed mostly require $O(V^2)$ time, where V is the number of vertices. Why? If you analyze the algorithms, you'll see that they involve examining each vertex once, and for that vertex going across its row in the adjacency matrix, looking at each edge in turn. In other words, each cell of the adjacency matrix, which has V^2 cells, is examined.

For large matrices $O(V^2)$ isn't very good performance. If the graph is dense, there isn't much you can do about improving this performance. As we noted earlier, by *dense*, we mean a graph that has many edges—one in which many or most of the cells in the adjacency matrix are filled. It's worthwhile noting that the maximum density graphs, ones where every vertex is adjacent to every other vertex, have $V \times (V - 1) / 2$ or $O(V^2)$ edges.

Many graphs are *sparse*, the opposite of dense. There's no clear-cut definition of how many edges a graph must have to be described as sparse or dense, but if each vertex in a large graph is connected by only a few edges, the graph would normally be described as sparse.

Consider again finding the shortest route between two road intersections. There could be millions of intersections for a graph covering a large continent. The intersections are the vertices of the graph, and the roads connecting them are the edges. Maybe the busiest intersection has 12 roads connected to it (such as the l'Arc de Triomphe de l'Etoile in Paris, France). Twelve is much smaller than a million, and even if every vertex had 12 edges, a million-vertex graph would need to look at most 12 million edges. That's vastly

smaller than what a dense graph would have: $O(V^2)$ or millions squared, which are trillions. All but the smallest road network graphs are sparse.

In sparse graphs, running times can be improved by using the adjacency list representation rather than the adjacency matrix. This is easy to understand: you don't waste time examining adjacency matrix cells that don't hold edges. It also takes less memory because you don't need an array cell for every vertex pair—only enough to hold each edge. That's an advantage of the hash table representation of the adjacency matrix too.

For unweighted graphs, a depth-first search with adjacency lists requires O(V+E) time, where V is the number of vertices and E is the number of edges. The traversal visits each vertex and each vertex's edges exactly once. For weighted graphs, both the minimum spanning tree and the shortest-path algorithm require $O((E+V) \times \log V)$ time. They, too, must visit every vertex and edge and choose the lowest-cost path at each iteration. The lowest-cost edge is kept in a heap with O(log N) removal time for the MST. (For the shortest-path algorithm, we would need to introduce another data structure to get O(log N) time to find the shortest cumulative path so far.) In large, sparse graphs these $O((E+V) \times \log V)$ times can represent dramatic improvements over the $O(V^2)$ adjacency matrix approach.

There's another kind of complexity—one that's especially important for new programmers. That's the intellectual complexity of the code. We've used the adjacency matrix approach throughout this chapter to make the code easier to read and the algorithmic steps easier to visualize. You can consult Sedgewick (see Appendix B, "Further Reading") and other writers for examples of graph algorithms using the adjacency-list approach. The adjacency list isn't much more intellectually complex and definitely outperforms the matrix on sparse graphs. The hash tables we've used for some of the structures like the costs table improve the performance by making insertion and search take O(1) time, but they do add intellectual complexity to the code.

The Warshall and the Floyd-Warshall algorithms are slower than the other algorithms we've discussed so far in this book. They both operate in $O(V^3)$ time, making use of the row and column nature of the adjacency matrix. This is the result of the three nested loops used in their implementation. They can be implemented using adjacency lists to save some time and get to $O(V^2 \times E)$, but at some cost of increased code complexity.

Intractable Problems

In this book you've seen Big O values starting from O(1), through O(N), $O(N \times \log N)$, $O(N^2)$, up to (for the Warshall and the Floyd-Warshall algorithms) $O(N^3)$. Even $O(N^3)$ running times can be reasonable for values of N in the thousands. Algorithms with these Big O values can be used to find solutions to most practical problems.

Some algorithms, however, have Big O running times that are so large that they can be used only for relatively small values of N. Many real-world problems that require such algorithms simply cannot be solved in a reasonable length of time. Such problems are said

to be **intractable**. (Another term used for such problems is **NP complete**, where NP means nondeterministic polynomial. An explanation of what this means is beyond the scope of this book.) The very complexity of these problems is part of what makes them interesting. Let's look at a couple of them.

The Knight's Tour

The Knight's Tour (Programming Project 14.6 in Chapter 14) is an example of an intractable problem because the number of possible paths is so large. The total number of possible move sequences (tours) is difficult to calculate, but you can approximate it. A knight in the middle of a board can move to a maximum of eight squares. This number is reduced by moves that would be off the edge of the board and moves that would end on a square that was already visited. In the early stages of a tour, there will be closer to eight moves, but this number will gradually decrease as the board fills up.

Let's assume (conservatively) only two possible moves from each position averaged over the entire puzzle where it will vary quite a bit. After the initial square, the knight can visit 63 more squares. Thinking of this situation like a binary tree where the knight goes to either the left or right child of a node, there is a total of 2^{63} possible move sequences. This is about 10^{19}. Assume a computer can make a billion moves a second (10^9). There are roughly 10^7 seconds in a year, so the computer can try about 10^{16} moves in a year. Solving the puzzle by brute force—exploring every possible move sequence—can therefore be expected to take 10^3, more than a thousand years because this was a conservative estimate.

This particular problem can be made more tractable if strategies are used to "prune" the tree created by the move sequences. One is Warnsdorff's heuristic (H. C. von Warnsdorff, 1823), which specifies that you always move to the square that has the fewest possible exit moves. A **heuristic** (from the ancient Greek: εὑρίσκω, heurískō, "I find, discover") in computer science is a technique that helps solve a problem more quickly, although it is not always guaranteed to do so.

The Traveling Salesperson Problem

Here's another famous intractable problem. Suppose you're a salesperson, and you need to go to all the cities where you have potential clients. You would like to minimize the number of miles, and thus time, you travel. You know the distance from each city to every other city. You want to start in your home city, visit each client city only once, and return to your home city. In what sequence should you visit these cities to minimize the total miles traveled? In graph theory, this is called the **traveling salesperson problem**, often abbreviated TSP. (It was originally called the traveling salesman problem.)

Let's return to Turala for another business opportunity, selling networking equipment in each of the six major cities. Figure 15-21 shows the cities and their driving distances. What's the shortest way to travel from Blum through each of the other cities and back to Blum? As before, not all cities are connected by roads to all other cities, at least not without passing through another city. For example, there are many ways to get from Blum to Gray, but none of them include driving directly from Blum to Gray without going through Cerf or Dahl.

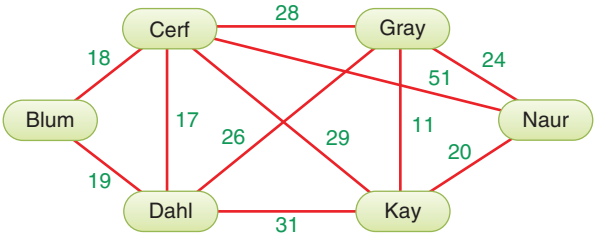


FIGURE 15-21 Driving distances in Turala

To find the shortest salesperson route, you could list all the possible permutations of the other cities (Dahl–Kay–Naur–Gray–Cerf, Dahl–Gray–Naur–Kay–Cerf, Dahl–Gray–Kay–Naur–Cerf, and so on) and calculate the total distance for each permutation. The route Blum–Dahl–Kay–Naur–Gray–Cerf–Blum has a total length of 140.

Unfortunately, the number of permutations can be very large: it's the factorial of the number of cities (not counting your home city). If there were 6 cities to visit, there are 6 choices for the first city, 5 for the second, 4 for the third, and so on, for a total of 6×5×4×3×2×1, or 720 possible routes. For the moment, you can count all possible permutations, even if no roads connect some pairs of cities.

The problem quickly becomes impractical to solve for even 50 cities (more than 10^{64}). Again, there are strategies to reduce the number of sequences that must be checked, but this helps only a little. You can apply the same logic that was used in the Knight's tour, noting that at each city you choose among a few edges, excluding those going to previously visited cities. If the average number of choices at each city is 2 or more, then there are at least 2^{50} paths to explore when trying to visit 50 cities. That's over 10^{15}, which is much smaller than 10^{64}, but still a huge number.

A weighted graph is used to implement the TSP problem, with weights representing distance and vertices representing the cities. The graph can be undirected if the distance is the same going from A to B as from B to A, as it usually is when driving. If the weights represent airfares or travel times, they may be different in different directions, in which case a directed graph is used.

Hamiltonian Paths and Cycles

A problem that's similar to the TSP but more abstract is that of finding a Hamiltonian cycle of a graph. As we noted earlier, a cycle is a path that starts and ends on the same vertex. A **Hamiltonian path** is one that visits every vertex in the graph exactly once but does not have a final edge that returns to the initial vertex. It's named for William Rowan Hamilton, who invented the *icosian game*, now also known as *Hamilton's puzzle*, which involves finding a Hamiltonian cycle in the edge graph of the dodecahedron. Finding a Hamiltonian path is somewhat like the bridges of Königsberg problem discussed in Chapter 14. The difference is between finding a path that visits all the vertices once, while the bridges of Königsberg problem seeks a path that visits all the edges once (and allows revisiting vertices).

Don't confuse finding a Hamiltonian cycle with detecting whether there are any cycles in a graph. A **Hamiltonian cycle** must include all the vertices, not just a subset, and be a cycle. Cycles in a graph can be detected with Warshall's algorithm (or Floyd-Warshall or some other more targeted algorithms) and detecting their existence is considerably easier than finding a Hamiltonian cycle.

Every solution to the TSP is a Hamiltonian cycle, but unlike the TSP, finding Hamiltonian paths and cycles doesn't care about distances; all you want to know is whether such a cycle exists. In Figure 15-21 the route Blum–Dahl–Kay–Gray–Naur–Cerf–Blum is a Hamiltonian cycle, while Blum–Dahl–Kay–Cerf–Gray–Naur–Blum is not because there is no Naur–Blum edge. The Knight's Tour problem is an example of a Hamiltonian cycle, if the knight returns to its starting square, and a Hamiltonian path, if not.

Finding a Hamiltonian cycle takes the same O(N!) time as the TSP. You'll see the term **exponential time** used for Big O values such as 2^N and N! The term *exponential* is used loosely in everyday parlance to mean anything that increases really fast. For computer science, however, we like to be a bit more precise.

To see how much worse O(N!) and $O(2^N)$ time are than $O(N^2)$, you can plot their growth on a graph (well, a different kind of graph…). Figure 15-22 puts them in perspective with the other growth rates that you saw in Chapter 2, "Arrays."

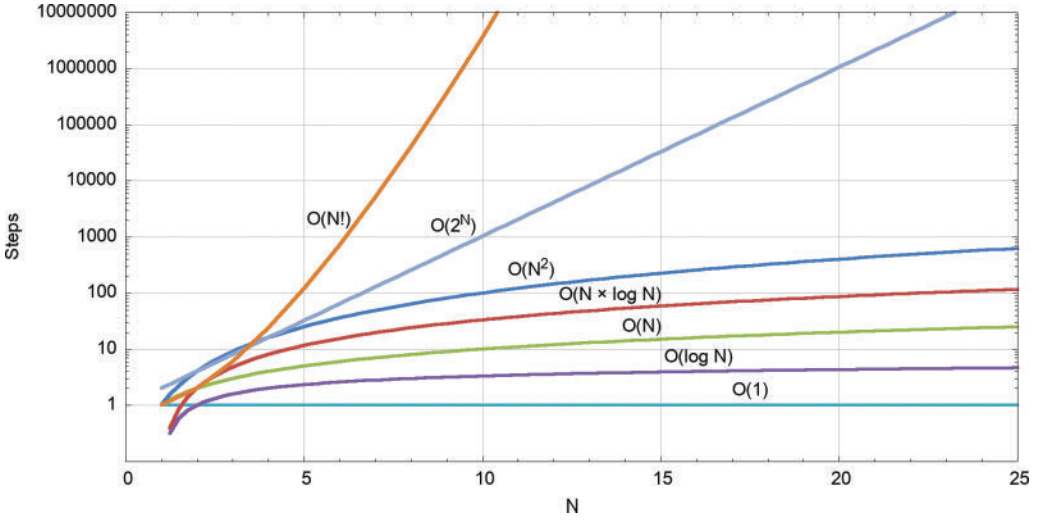


FIGURE 15-22 Exponential growth

The uppermost line shows the O(N!) growth rate. It starts off being less than $O(N^2)$ for small values of N but quickly dominates all the others. The next line below it is for $O(2^N)$, and it also dominates all of the other growth rates you've seen so far. It's clear that the number of steps grows far faster for these exponential categories than the others.

The $O(2^N)$ line appears straight in this graph, whereas all the others have some curvature. Is it clear to you why it is straight? In the graph of Figure 15-22, the vertical axis is *logarithmic*. Every step vertically in the graph means the number grows tenfold. You saw that logarithms are the inverse of exponential functions back in Chapter 2. When you plot an exponential function like 2^N on a logarithmic scale, they "cancel each other," and the curve flattens out into a line.

The growth rates $O(1)$, $O(\log N)$, $O(N)$, $O(N \times \log N)$, and $O(N^2)$ are all less than exponential, curving downward as N increases in the logarithmic graph of Figure 15-22. Note that while the $O(N^2)$ curve has an exponent in the formula, it is *not* called exponential because it grows significantly more slowly than the $O(2^N)$ curve. Even the more complex $O(N^3)$, $O(N^4)$, and so on growth rates will be smaller than $O(2^N)$ for large values of N.

The $O(N!)$ growth rate not only exceeds the $O(2^N)$ exponential, but it has a slight upward curve. That's on the logarithmic scale, so it grows even more aggressively than the others do. You must be very careful with algorithms that have this kind of performance. If you invent a new algorithm to solve a problem, you should analyze its performance to see what size inputs it can handle.

Would you ever want to implement a program that takes exponential time? Probably not, but there's another reason why they are so intriguing. Certain kinds of problems can be shown to *always* require exponential time to solve. It's not a question of choosing the right data structures to get a lower complexity solution; there are none. These kinds of algorithms are the basis of cryptography—scrambling information into sequences of symbols that should be accessible only to specific people or groups. Finding really hard problems to solve means the problem's solution can be used to encrypt the information. Only those people who know the solution will be able to decrypt the information in any kind of reasonable time.

Summary

- In a weighted graph, edges have an associated number called the weight, which might represent distances, costs, times, or other quantities.
- The minimum spanning tree in a weighted graph minimizes the weights of the edges necessary to connect all the vertices.
- An algorithm using a priority queue or heap can be used to find the minimum spanning tree of a weighted graph.
- At each step, the minimum spanning tree algorithm chooses the lowest weighted edge from a visited vertex to an unvisited one.
- Finding the minimum spanning tree of a weighted graph solves real-world challenges such as installing network cables between cities.
- Large graphs are often "discovered" as an algorithm focuses on one vertex, edge, or path at a time.

- Nonadjacent vertices can be used by assigning them an infinite weight edge.
- The shortest-path problem in a nonweighted graph involves finding the minimum number of edges between two vertices.
- Solving the shortest-path problem for weighted graphs yields the path with the minimum total edge weight.
- The shortest-path problem for weighted graphs can be solved with Dijkstra's algorithm.
- In each iteration of Dijkstra's algorithm, the lowest weight route to an unvisited vertex determines the next vertex to visit.
- Dijkstra's algorithm keeps records for the shortest path to all visited vertices including the total weight and parent vertex on the path to that visited vertex.
- The algorithms for large, sparse graphs generally run much faster if the adjacency-list representation of the graph is used rather than the adjacency matrix.
- Hash tables are a good representation for sparse adjacency matrices.
- Finding the total weight of the edges between every pair of vertices in a graph is called the all-pairs shortest-path problem. The Floyd-Warshall algorithm can be used to solve this problem.
- Some graph algorithms take exponential time and are therefore not practical for graphs with more than a few vertices.
- The exponential growth rates—O(N!) and $O(2^N)$—grow significantly faster than the $O(N^2)$ rate, which in turn, is much higher than other rates like $O(N \times \log N)$, O(N), and O(log N).
- Examples of problems that take exponential time are the traveling salesperson problem (TSP) and finding Hamiltonian cycles.

Questions

These questions are intended as a self-test for readers. Answers may be found in Appendix C.

1. The weights in a weighted graph are used to model things like _____, _____, and _____.
2. The minimum spanning tree (MST) of a weighted graph minimizes
   a. the number of edges from the starting vertex to a specified vertex.
   b. the number of edges used to span all the vertices.
   c. the total weight of edges connecting all the vertices.
   d. the total weight of the edges between two specified vertices in the tree.

3. The numerical weights in weighted graphs
   a. must be integers with zeros reserved for nonadjacent vertices.
   b. can be any finite value but must have a sum less than the number of vertices.
   c. cannot include negative values.
   d. can include any value including positive infinity for nonadjacent vertices.
4. True or False: The weight of the MST depends on the starting vertex.
5. In the MST algorithm, what is removed from the priority queue?
6. In the country network installation example, at the time each edge is added to the MST, the edge connects
   a. the starting city to an adjacent city.
   b. an already-connected city to an unconnected city.
   c. an unconnected city to a fringe city.
   d. two cities with offices.
7. True or False: After adding an edge to the MST, it could be removed or replaced later with a better edge.
8. When exploring edges at a newly visited vertex, the MST algorithm "prunes" edges that lead to a vertex that _____.
9. True or False: The shortest-path problem must be carried out on a directed graph.
10. Dijkstra's algorithm finds the shortest
    a. paths from one specified vertex to all the vertices visited while finding the shortest path to another specified vertex.
    b. paths from all vertices to all other vertices that can be reached along one edge.
    c. paths from all vertices to all other vertices that can be reached along multiple edges.
    d. path from one specified vertex to another specified vertex.
11. True or False: The rule in Dijkstra's algorithm when applied to a graph where the edge weights are distances is to always put in the subgraph the unvisited vertex that is closest to the starting vertex.
12. In the railroad route example, a fringe city is one
    a. to which the travel time is known, but from which no other travel times are known.
    b. which is in the tree.
    c. to which the travel time is known and which was just added to the tree.
    d. which is completely unknown.

13. The all-pairs shortest-path problem involves finding the shortest path
    a. from the starting vertex to every other vertex.
    b. from the starting vertex to every vertex that is one edge away.
    c. from every vertex to every other vertex that is more than one edge away.
    d. from every vertex to every other vertex.
14. Comparing the Floyd-Warshall algorithm with Warshall's algorithm (described in Chapter 14), how are the matrix cells updated differently?
15. Problems that take an exponential amount of time to solve are called _____.
16. Representing adjacency using a(n) _____ can reduce the complexity of several algorithms on sparse graphs compared to using a(n) _____.
17. A path weight matrix is the output of the _____ algorithm.
18. What is an approximate Big O time for an attempt to solve the knight's tour on a K×K board?
19. In Figure 15-21, is the route Blum–Cerf–Naur–Kay–Gray–Dahl–Blum the minimum solution for the traveling salesperson problem? Why or why not?
20. When should a directed graph be used to solve the traveling salesperson problem, as opposed to an undirected graph?

Experiments

Carrying out these experiments will help to provide insights into the topics covered in the chapter. No programming is involved.

15-A Consider an alternative proposal for finding the minimum spanning tree. For a connected, undirected graph with V vertices, take the V – 1 lowest weight edges and make a graph with them and the V vertices. If some of the edges have the same weight, randomly select among the edges of equal weight to get the V – 1 edges. Try it on the graphs of Figure 15-1 and Figure 15-8. Does it provide a better solution? Why or why not? Can you think of other graph examples where it would work or not work?

15-B The minimum spanning tree algorithm can start from any vertex. If it starts at different vertices, will it find the same tree? Can you find some small graphs that produce different trees and some graphs that produce the same trees? (Hint: Keep the graphs small.) See whether you can define under what graph conditions the algorithm will find the exact same minimum spanning tree.

15-C Dijkstra's algorithm finds the shortest path between two vertices and has the side benefit of finding the shortest path from the starting vertex to some of the other vertices. Can you define which of those other vertices get that benefit? Does it

matter what ending vertex was chosen? Try some examples with graphs in the WeightedGraph Visualization tool, where the graph has only one connected component.

15-D Traveling salespeople are much rarer now than they used to be. Think about what other kinds of business activity need to solve the TSP. How fast do they need to solve it? How many vertices would be in the graphs?

15-E Turala's government tells you they want to connect water supplies between their different regions using pipelines. They would like to pump water in either direction between about 50 water sources depending on changing rainfall and other conditions. Of course, they want to minimize the costs, and it's acceptable to pump water via intermediate points rather than connecting all sources to all other sources. What kind of problem is this, and what algorithm would you use? What's the difference between finding the minimum construction costs versus minimum operational costs?

Programming Projects

Writing programs to solve the Programming Projects helps to solidify your understanding of the material and demonstrates how the chapter's concepts are applied. (As noted in the Introduction, qualified instructors may obtain completed solutions to the Programming Projects on the publisher's website.)

15.1 Create an `allShortestPathsMatrix()` method for the `WeightedGraph` class using the Floyd-Warshall algorithm. It should return an array of the minimum costs to get from any vertex to any other vertex. The result matrix can be a Python `dict` that is indexed like the `WeightedGraph._adjMat` attribute. Demonstrate it running on the train times shown in the lower graph of Figure 15-13.

15.2 Create a *directed* weighted graph class based on the `WeightedGraph` described in the text. Use it to make a generator for all maximum length paths in the directed graph. A maximum length path from starting vertex K is one that cannot be extended to an adjacent vertex without revisiting a vertex already on the path. The `allMaxPaths()` generator should yield all such paths starting from all vertices. This can be done as a recursive generator. The generator will produce all the long paths but will also include shorter and even single vertex paths for vertices with no outgoing edges. Create another generator, `HamiltonianPaths()`, that calls `allMaxPaths()` and yields only those paths that contain all the vertices in the graph. Show the output of both methods operating on three graphs: the one shown in Figure 15-19, that same graph minus the outgoing edges from Vertex B, and that same graph with an additional edge from C to B with weight 15.

15.3 Implement a method that solves the traveling salesperson problem described in the "Intractable Problems" section in this chapter. In spite of its intractability, your method should have no trouble solving the problem for small N, say 10 cities or fewer. Use a directed graph as implemented in Programming Project 15.2.

You can use the brute-force approach of testing every possible sequence of cities. Your method should return either an empty list or `None` if there is no solution. Demonstrate your method running on the driving times shown in Figure 15-21 with three variations on the return times: (a) the same time, (b) the return takes 5 minutes longer, and (c) the return takes 10 minutes longer. Does the solution change among the variations?

15.4 When you're planning a project, it's important to know how long it will take to complete. You saw in Chapter 14 that directed graphs could be used to model dependency relationships and how to topologically sort the vertices to show the order of tasks needed to complete a project. With time as the weight in a graph, you can find two things: the total time needed to complete the project and the **critical path**—the path through the dependency graph that takes the longest to complete.

Use the directed graph as implemented in Programming Project 15.2 and create a `criticalPath()` method that takes a vertex index as parameter and returns the critical path leading to it along with the total time (weight) along that path. Vertices with no predecessors (inbound edges) have only themselves in the critical path and a total time of zero. All other vertices return the critical path among their predecessors that has the longest time. This method can be implemented recursively. Take care to prevent infinite recursion if the graph has a cycle; the `criticalPath()` method should return the path up to a vertex that would cause a cycle along with infinite time for that path. Show the output of the method operating on three graphs: the one shown in Figure 15-19, that same graph minus the outgoing edges from Vertex B, and that same graph with an additional edge from B to C with weight 35.

15.5 Many problems require finding the connected components of a graph and uniquely labeling the vertices that comprise them. For a road network in a mountainous, snowy country, the towns and cities are all part of one connected component in good weather. As snow falls in the mountains, roads become impassable, sometimes cutting off towns from the others. Weighted graphs can model that situation by having towns as vertices, roads as edges, and weighting the edges by their maximum elevation. When snow cuts off travel above a given elevation, the connected components change.

Write a `connectedComponents()` method for the `WeightedGraph` class that returns an array of labels for each of the vertices. Vertices with the same label in the array are part of the same component; different labels imply disconnected components. The `connectedComponents()` method takes a threshold argument that will be used to identify which roads are still passable. Its default value is infinity so that all roads (edges) with finite weight are included.

The algorithm for finding the labels starts off by assigning each vertex the label of its `name` attribute (you can assume all the cities have unique names). Then you make a sequence of update passes to change the labels. In each update pass, the algorithm

looks at every edge with weight below the threshold. If the vertices at the ends of the edge have different labels, it replaces the higher label with the lower one (by comparing their lexicographic order). This repeats until an update pass goes through all the edges without changing any labels. The minimum labels "spread" across all the vertices in their connected component until no more labels change.

Write a second method, `componentVertices()`, that takes the array of labels and builds a hash table that maps a label to a list of vertex indices that share the label. The number of keys in the hash table is the number of connected components. Demonstrate your methods running on the graph in Figure 15-21 using thresholds of 50, 21, and 15.

This page intentionally left blank

CHAPTER 16
What to Use and Why

IN THIS CHAPTER

- Analyzing the Problem
- Foundational Data Structures
- Special-Ordering Data Structures
- Sorting
- Specialty Data Structures
- External Storage
- Onward

We briefly summarize what we've learned so far, with an eye toward deciding what data structure or algorithm to use in a particular situation.

This chapter comes with some caveats. Of necessity, it's very general. Every real-world situation is unique, so what we say here may not be the right answer to your problem. You need to analyze the problem to determine its characteristics and see whether they match those of the data structure or algorithm.

This chapter is divided into these somewhat arbitrary sections:

- Analyzing the problem: Determine the characteristics that guide the choice of data structure and algorithm
- Foundational data structures: Arrays, linked lists, trees, hash tables
- Special-ordering data structures: Stacks, queues, priority queues, heaps
- Sorting: Insertion sort, Shellsort, quicksort, mergesort, heapsort
- Specialty data structures: Quadtrees, graphs
- External storage: Sequential storage, indexed files, B-trees, hashing

NOTE

For detailed information on these topics, refer to the individual chapters in this book.

Analyzing the Problem

As a developer, you should always review the plans for the software you intend to develop. There can be—and usually are—many goals for a particular software project. Commercial companies want to create products that sell. Scientific organizations want systems that accurately record, model, and analyze their data. Governments want software that helps educate, protect, and account for their citizens and jurisdictions. Analyzing these kinds of goals is important; it tells you what needs to be built or changed. The organization's goals are less informative about *how* to build it to achieve the speed, cost (in terms of memory and other computing resources), and ease of maintenance that everyone wants.

Let's assume someone has analyzed the overall goals and provided a set of desired features and requirements for the new system. To figure out what data structures and algorithms are appropriate, you need to look at the characteristics of the computations involved. What kind of data is being manipulated? How much will be processed? What kinds of results are needed and how fast? The answers to these kinds of questions help narrow your choices.

What Kind of Data?

The problem domain usually makes very clear what type of data will be manipulated. It could be records about company transactions, audio recordings of animal calls, measurements of experiments in a lab, or videos that will be streamed to viewers. Each kind of data is built up from the primitive data types: numbers, characters, and Booleans. The primitives are grouped into meaningful chunks—for example, text message strings, arrays of numbers measuring temperatures, records for personnel, centerlines of roadways, medical images, and volumetric shapes. The enormous variety of kinds of data can be quite daunting.

Despite that variety, some characteristics help pick the best approach for data structures. First among them is how will the different parts of the data be accessed? For example, if you're processing text message strings, you expect them to be short and to be requested as a whole; there likely is no need to extract the last word from each text nor to find the median-valued character in each one. Typically, the "chunks" are manipulated as a whole. If components within the chunks need separate processing, that fact should be noted. For example, videos typically contain a series of still images and a separate series of numeric values for one or more audio channels. The visual and audio content can be manipulated together—as an editor might carve up a video of an interview—or manipulated separately to find utterances of words or find the appearances of blue sky in the video frames.

With the different chunks of data identified, the next question to ask is how will those chunks be identified? In particular, is there a key or index that helps uniquely identify it? For example, personnel records usually have many fields, and several of them might be used as keys to identify the records. Text messages don't have an implicit key to identify them, but when they are part of records, they often have keys for sender, recipient, and time sent. The text content can, of course, be processed to find other keys like words, phrases, or languages used. Keys are often stored alongside the more numeric structures

like arrays or images. For example, an image database might have keywords associated with the image content—for example, outdoor/indoor, people, landscapes, animals.

The other way to identify chunks of data is by index or coordinates. To get the next message in a sequence means going to the next index. To get the image that appears 10 seconds after the start of a video, a program would find the video frame by its index. The array-like data chunks always have natural indices and are frequently accessed by those indices. It would make no sense to store an image, for example, with each of its pixels in a different node of binary search tree ordered by pixel value. Reconstructing the array form of the image from the tree would be time-consuming and likely to be needed frequently. Other indices can involve multiple numbers like map coordinates and volume/spatial coordinates.

How Much Data?

When you know the kinds of data, how they fit into chunks, and how they are going to be addressed, the next question is how many of those chunks are there? This characteristic often drives the choice of data structure significantly and is sometimes the hardest to determine. Projects often start with a small scope in mind and only later does someone try to run it with significant amounts of data. It's easy and tempting to focus on the smaller sizes, because that means the program is likely to run fast enough regardless of the data structure(s) and algorithm(s) used. With a little more analysis, you can often foresee how the amount of data might expand and be prepared for it.

In the analysis, you don't have to be very precise. You really only need to know how many data items in terms of **orders of magnitude**. Are there tens of items? Thousands? Millions? The exact number doesn't matter, but the number of decimal places does.

Look for External Constraints

If the data to be processed is constrained in some way, you may be able to conclude that the code will never need to process more than a certain amount. If you know, for example, that the system will need to store the known contacts of only one person, it's extremely unlikely to need to store more than 1,000 items and certainly fewer than 100,000. A system tracking one or two items per person for everyone alive today needs to handle at most tens of billions of items. By the time such a system will need to handle trillions, it's likely that some new system will have replaced it.

What Creates the Data?

The biggest difference in amounts of data comes from how it is produced. If the data is manually produced, it is likely to be far smaller than data produced by automation. Things that people produce manually like text messages, songs, or paintings are all limited by the amount of time spent doing the activity and the number of people doing it. There are good models about how many people are alive now and how many there will be for decades. It may be less clear how many of the people will perform the activity, but you really only need estimates of the order of magnitude: thousands, millions, and so on.

Data produced by automation can be many orders of magnitude larger. Imagine all of the video produced by an astronomical telescope operating continuously (or at least nightly)

for a year. Multiply that by all the telescopes in operation and the total time period to get an estimate for the quantity of data that might be collected for analysis. When making estimates, you must also consider potential growth. For example, when you're thinking about the number of sources of video that might produce content on a video sharing site, an estimate made in the year 2000 would probably have been one or two orders of magnitude smaller than one made in 2020 after the widespread acceptance of handheld smartphones with video recording capability.

Underestimates are common. Projects usually start by focusing on small goals and leave the challenge of scaling for later. A classic example was the common design decision to store two digits for the year in many database systems developed in the late twentieth century. Companies and governments made great efforts to find and fix all such limited systems before the year 2000. Similar problems will occur in the future too, because nearly all timekeeping systems use a fixed amount of storage for their timestamps. Although these examples are not underestimates of the quantity of data, they are indicative of the tendency to develop systems that can handle initial estimates without accounting for how the data will change over time.

What Operations and How Frequent?

The system to be built does something with these chunks of data. A good design lists all the operations, along with estimates of how frequently they will be done and what the requirements are for their performance.

For example, a bookkeeping system for a business has insertion of new data for every sales transaction, income, and payments. There are hundreds or thousands or (hopefully) hundreds of thousands of transactions daily. There are deletions for errors occasionally, maybe a hundred in a day. There are operations to report on overall activity, income and expenses, every day, week, month, quarter, and year. There are performance requirements for each of those, such as being able to insert a new transaction in a microsecond or get a quarterly report in less than a minute. All of those can be reasonably achieved, but on some new project, you might spot problems where it will become difficult to do.

It's often quite clear what kinds of operations must be done for inputs and outputs to humans. Operations that are done for maintenance or for sending to some other process might have time constraints that are less clear. For example, storm prediction models need to be rerun with the latest inputs at regular intervals, legal requirements can require that old data be removed from a system periodically, and financial transactions that streamed in too fast to be fully processed during business hours need to be reconciled and cataloged after trading stops. The term *background processing* is sometimes used to describe such maintenance tasks. They can affect all the system's data and hence could take significant computing resources.

When you're considering how frequently operations occur, it's not always possible to be specific. Unlike the bookkeeping system with daily, weekly, monthly, quarterly, and annual reports, most systems are run on demand, and the amount of demand is highly variable. What's most important to know, however, is the *relative frequency* of the operations. Are searches more frequent than insertions? Is traversal more frequent than search?

At a minimum, you should try to rank order the four basic operations of insertion, deletion, search, and traversal.

The processing order of data items strongly influences the choice of structures. For example, reconciling all the transactions that streamed in during a trading day likely must be done in the order they occurred. That means a queue is more appropriate than a stack. If the streaming order is not quite the same as the trading order, then a priority queue or heap would be better.

In another example, a warehouse inventory system would have updates for new stock added to the warehouse and for items removed to fill orders. When users want to place an order, the current item quantities determine whether or not it can be filled immediately. The sequence of what items appear in an order is not predictable. The orders contain seemingly random items. This means that the processing order is unpredictable, and using a data structure designed for a specific order like a queue or a stack might be ill advised.

Different algorithms can impose the processing order requirement as well. In Dijkstra's algorithm for the shortest path between two vertices in a graph (Chapter 15, "Weighted Graphs"), there is a specific order in visiting the vertices. In the Tower of Hanoi problem (Chapter 6, "Recursion"), both the constraints on moving disks and the method of finding the specific order of disk movements to solve the puzzle need the last-in, first-out (LIFO) processing order provided by a stack. The algorithmic constraints are often better defined than what the system requirements provide.

Most importantly, what operations must be fast? Are any of them special? If the purpose of the system is to produce the driving routes of delivery trucks or to find the locations of warehouses nearby, the general-purpose data structures are not likely to perform well enough. The operations that must be performed the fastest often force the choice of using specialized data structures and/or algorithms.

Who Will Maintain the Software?

Determining what data structures and algorithms to use depends not only on the data and operations on that data, but on the people who will maintain the software as (the inevitable) changes are made. Programmers have different skill levels, and using highly complicated algorithms or structures could become a barrier to being able to make changes easily.

Knowing who will work on a problem and what kinds of data structures are challenges for them is probably the hardest problem characteristic to gauge. If this is a program run by a large corporate or government agency, they are likely to be able to find highly skilled programmers to maintain and update the system. At the other end of the spectrum, a program written by volunteers for a social club might have trouble finding more volunteers to fix some future problem. Nontechnical organizations could have more trouble than ones that attract volunteers with software skills.

If you expect the people maintaining the software to be comfortable with advanced data structures, then the choices of structures and algorithms are wide open. Technical and scientific organizations are generally very receptive to using complex structures, if they will achieve better performance. Another way to put it is that technical organizations are less

averse to the risk of using advanced structures and algorithms that might be hard for software developers to master in later updates. That doesn't mean you should always choose the most advanced structure or algorithm for such an organization; choosing the least complex one that meets the performance requirements means a broader group of developers will be able to maintain it.

When the future development staff is less well-defined, or is unlikely to have familiarity with advanced structures, or would take a long time to learn complex structures, the design choices should be more limited. Using exotic algorithms that are not widely known adds to the maintenance burdens for the system. That said, it is always important to achieve the performance requirements of the system. It would be unacceptable, for example, to use an $O(N^2)$ algorithm that's easier to understand than a complex $O(N \times \log N)$ algorithm if the system is not able to process the expected amount of data in the required time.

Note that some complexity issues are mitigated by the use of standard libraries to implement the data structures. Hash tables use some rather complex concepts in their implementation, but using them inside another program is straightforward when a good library is available, or when there's already a highly tuned built-in implementation such as a Python dictionary or set. Data structures in standard libraries with well-documented programming interfaces and well-known performance characteristics should be fairly easy for other developers to use and maintain.

Foundational Data Structures

Every data structure is built on top of the primitive types—numbers, characters, and Booleans—and ways of putting them together. The two fundamental ways of arranging these primitives into bigger groups are by putting sequences of the same primitive type into arrays, and by linking different chunks together using references. You can also say that references are a primitive type (although you use them only for organizing the other data, not to hold the values to be manipulated).

With these basic constructs, you can form the foundational data structures: arrays, linked lists, and trees. You can also include hash tables in this category. Hash tables behave like arrays in many ways (even though the indexing is much more complicated than for arrays). All other structures can be built using combinations of these, but sometimes they are all that is needed.

The biggest differences among these foundational structures are

- How items they contain are addressed: by index, by reference, by key, by coordinates
- How memory is allocated and deallocated for them

Arrays, of course, allow items to be indexed by an integer and are allocated in blocks where the program specifies the number of items when the array is created. Linked lists and trees make use of references. The items they contain can be specified by a reference to a link or node, or perhaps by an integer that counts items from the beginning of the list or the root of the tree. The links and nodes are allocated as they are needed. Hash tables

address items by a key and typically only by a key. Of course, items in arrays, linked lists, and trees can also be addressed by a key, but it takes more time to do so, significantly more time as the number of items grows. Hash tables are also somewhat like lists and trees in the way that memory can be allocated as needed.

Which of these general-purpose data structures is appropriate for a given problem? We go through them individually and show how the characteristics of a problem help answer that.

Speed and Algorithms

The general-purpose data structures can be roughly arranged in terms of speed when items are specified by key: arrays and linked lists are slow, trees are fairly fast, and hash tables are very fast.

Don't draw the conclusion, however, that it's always best to use the fastest structures. There's a penalty for using them. First, they are—in varying degrees—more complex to program than the array and linked list. Also, hash tables use memory somewhat inefficiently. Ordinary binary trees revert to slow O(N) operation for ordered data; and balanced trees, which avoid this problem, are more complex to program.

Processing Speed: A Moving Target

The fast structures come with some drawbacks, and another development makes the slow structures more attractive. Every year there's an increase in the CPU and memory-access speed of the latest computers. Moore's Law (postulated by Gordon Moore in 1965) specifies that the number of transistors on microchips will double every 18 months. Over 50 years later, that trend has held up, even as the size of individual transistors approaches the size of molecules and atoms.

In a related trend, the clock speed of chips rose exponentially, although the rate of increase has begun to diminish. Coupled with the ability to run many programs (or threads) in parallel, the effective processing speed of computers has continued to rise exponentially. Doesn't this suggest that you don't need to worry about the big O efficiency of data structures? If the computer runs exponentially faster, won't it overcome the differences between the slower and faster structures?

The answer is not a simple yes or no. While the computers have been able to run faster, software developers have taken advantage of these gains to write software that uses more memory and processing cycles on the computer. This **software bloat** means that operating system size, random-access memory requirements, library size, and features have all been increasing as well. Ironically, those increases have reduced and occasionally reversed the speed gains from hardware. The instinct to implement "easier-to-use" but slower or more memory-intensive structures has counteracted the gains in processing speed.

Suppose a computer a few years ago handled an array of 100 objects in acceptable time. Now, computers are much faster, so an array with 10,000 objects might run at the same speed. Many writers provide estimates of the maximum size you can make a data structure before it becomes too slow. Don't trust these estimates (including those in this book). Today's estimate doesn't apply to tomorrow.

Instead, start by considering the simplest data structures that meet the requirements. Unless it's obvious they'll be too slow, code a simple version of an array or linked list and see what happens. If it runs in acceptable time on the biggest amount of data, look no further. Why toil away on a balanced tree when no one would ever notice if you used an array instead? Even if you must deal with millions or tens of millions of items, it's still worthwhile to see how well an array or linked list will handle them. Only when experimentation shows their performance to be too slow should you revert to more sophisticated data structures. Of course, you must plan properly to have time to make such changes if your initial design doesn't pan out.

Libraries

Libraries of data structures are available commercially in all major programming languages. Languages themselves may have some structures built in. We've explored Python's structures extensively throughout this book. Java, for example, includes `Vector`, `Stack`, and `Hashtable` classes. C++ includes the Standard Template Library (STL), which contains classes for many data structures and algorithms.

Using a commercial library may eliminate or at least reduce the programming necessary to create the data structures described in this book. It also eliminates some of the risk that future developers will need to dig into the details of their coding. When commercial or well-maintained public libraries are available, using a complex structure such as a balanced tree, or a delicate algorithm such as quicksort, becomes a more attractive possibility. You must, however, analyze their specific features, requirements, and limitations to ensure that the class, method, or algorithm can be adapted to your particular problem.

Arrays

In many situations the array is the first kind of structure you should consider when storing and manipulating data. Arrays are useful when

- The amount of data is reasonably small.
- The amount of data is predictable in advance.
- The items will be addressed by index for most of the time.
- The order the items will be accessed is variable.

What is "reasonably small"? The answer depends on your computing resources and how much you must store for each array cell. The basic idea is that it should fit in the computer's random-access memory (RAM). It can be as large as the virtual address space allowed on the computer, but parts of that space will be swapped in and out of the random-access memory, slowing it down. As an example, a single processor might have 8 gigabytes of RAM and a virtual address space of 2^{64} bytes. Allocating a billion-cell array (where each cell takes 8 bytes) might be fine in that environment. The larger the array, the more the speed of shuffling data in and out of RAM will slow down the computation. The shuffling becomes even more pronounced on systems where multiple processes must share the RAM. We discuss this issue more in the "Virtual Memory" section.

Predicting the size of the data in advance is hard in many applications. Arrays work best when the whole array can be allocated at once, or perhaps in a few iterations. When the array size grows, data must be copied from the existing array to the new one. The growth operation can sometimes lead to a significant amount of memory being allocated but unused.

The speed of arrays is contingent on being able to find a cell in constant time by using an integer index to find where it lies in memory relative to the array's starting address. If the algorithms that use the items in the array have that integer index available, arrays are as fast as it gets. If the items must be moved within the array, say within a sorted array or an array where the active data must always be the first N cells, then the indices change with deletions and/or insertions, and it becomes harder to take advantage of the constant time access. Arrays are best when the data will be inserted, randomly accessed by the index, and rarely deleted.

You shouldn't use an array

- If the data will be addressed by key (try a hash table).
- If there will be many insertions and/or deletions that require moving items within the array.
- If the data must be kept in sorted order and is large (try a heap).

Linked Lists

Consider a linked list when

- The amount of data to be stored is not easily predicted in advance and minimizing memory usage is important.
- Data will frequently be inserted and deleted, and the order of the items is important.
- The data will be addressed or processed in the order it was received (or the reverse).

The linked list obtains whatever storage it needs as new items are added, so it can expand to fill all available memory; and there is no need to fill "holes" during deletion, as there is in arrays.

Insertion is fast in an unordered list. Searching and deletion by key or by index are slow (although deletion is faster than in an array). Like arrays, linked lists addressed by key or index are best used when the amount of data is comparatively small.

You shouldn't use a linked list

- If the data will be addressed by a key (try a hash table).
- If the data will be addressed by an integer index in arbitrary order (try an array).

Binary Search Trees

A binary search tree is the next structure to consider when arrays and linked lists prove too slow. Consider binary search trees

- When data will be addressed and ordered by a key.
- When data will be frequently inserted and deleted.
- When minimizing memory usage is important.

A tree provides fast O(log N) insertion, searching, and deletion. Traversal in key order is O(N), which is the fastest for any data structure. [The fact that binary search trees order their items by a key means a traversal in sorted order takes O(N) time while an unsorted array or list would need to be sorted first.] You can also find the minimum and maximum quickly, O(log N), and traverse a range of items.

An unbalanced binary tree is much easier to program than one that balances itself, but unfortunately ordered data can reduce its performance to O(N) time, no better than a linked list. If you're certain the data will be inserted in random order with respect to the key, there's not much point in using a balanced tree.

You shouldn't use a binary search tree

- If an array or a linked list meets all the requirements.
- If the order of the items is unimportant.
- If the items are likely to be inserted in the forward or reverse order of the key.

Balanced Search Trees

Of the various kinds of balanced search trees, we discussed 2-3-4, red-black, and AVL trees. They are all balanced trees, and thus guarantee O(log N) performance whether insertions are ordered according to the key or not. These balanced trees are challenging to program. They may also impose slightly more memory overhead than plain binary search trees, which may or may not be significant.

Consider a balanced search tree

- When data will be addressed and ordered by a key.
- When long sequences of insertions in (reverse) order of the key are likely.
- When data will be frequently inserted and deleted.
- When minimizing memory usage is important.
- When the complexities of the balancing algorithms are handled by experienced developers.

The risk associated with developing and maintaining complex algorithms may be reduced by using libraries maintained by qualified companies or other organizations. In many

cases, hash tables may be a better choice than balanced search trees, but also may use complex hashing algorithms.

You shouldn't use a balanced search tree

- If an array or a linked list meets all the requirements.
- If the order of the items is unimportant.
- If the items are likely to be inserted in random order with respect to the key.

Hash Tables

Hash tables have the most desirable performance for almost all data items referenced by a key. The keys can be any data type that might be considered an index, including integers. The O(1) performance for searching, insertion, and deletion by key are the fastest possible.

Consider a hash table

- When data will be addressed by a key and the traversal order of the items by the key is unimportant.
- When minimizing memory usage is somewhat important and some unused memory is tolerable.
- When the complexities of the hashing and collision resolution algorithms are handled by experienced developers.

Hash tables are not sensitive to the order in which data is inserted and so can take the place of a balanced tree. Programming them is simpler than for balanced trees assuming a good hashing function for the keys is available.

Hash tables require more than the minimum amount of memory, especially for open addressing. For example, with a load factor limit of 50 percent, they use double the number of cells or more than the links needed for a linked list. Like the linked lists and trees, the memory can be allocated as items are inserted with only a modest amount of extra time. (Some insertions cause the array to grow and some deletions can cause the array to shrink, but over N operations, only time proportional to N is needed to copy items.)

Hash tables don't support any kind of ordered traversal, or easy access to the minimum or maximum keys and items. Even finding all the keys that have been inserted is more time-consuming than a tree or list. If these capabilities are important, the binary search tree is a better choice.

You shouldn't use a hash table

- If the order of the items by key is important (try a balanced search tree).
- The amount of memory used must be as small as possible (try a balanced search tree).
- The keys of items are not hashed uniformly by the hashing function (try another hashing function).

Comparing the General-Purpose Storage Structures

Table 16-1 summarizes the speeds of the various foundational data storage structures using Big O notation.

Table 16-1 Foundational Data Structure Speeds

Data Structure	Search	Insertion	Deletion	Traversal
Array	O(N)	O(1)	O(N)	O(N)*
Ordered array	O(log N)	O(N)	O(N)	O(N)
Linked list	O(N)	O(1)	O(N)	O(N)*
Ordered linked list	O(N)	O(N)	O(N)	O(N)
Binary tree (average)	O(log N)	O(log N)	O(log N)	O(N)
Binary tree (worst case)	O(N)	O(N)	O(N)	O(N)
Balanced tree (average and worst case)	O(log N)	O(log N)	O(log N)	O(N)
Hash table	O(1)	O(1)	O(1)	O(N)* (but higher C)

In unordered arrays and lists, insertion is made at the end and beginning of the structure, respectively. Deletions within arrays require filling the gap by shifting items. Deletion of the first item of a list is O(1), but the general delete by key takes O(N). The ordered array uses a binary search, which is fast, but insertion and deletion require moving half the items on the average, which is slow. Traversal means visiting all N items, but only the ordered structures make it easy to visit them in the order of ascending (or descending) keys; the asterisk (*) means the traversal by ordered keys is not supported.

Special-Ordering Data Structures

The special-ordering data structures discussed in this book are the stack, the queue, and the priority queue. These structures provide fast access to the data in a particular order. Instead of acting as a general-purpose data store, they are typically used by a computer program to aid in carrying out some algorithm. We've seen examples of this kind of usage throughout this book, such as in Chapters 14, "Graphs," and 15, "Weighted Graphs," where stacks, queues, priority queues, and hash tables are all used in graph algorithms.

Stacks, queues, and priority queues are implemented by a foundational structure such as an array, a linked list, or (in the case of the priority queue) a heap. These data structures present a simple interface to the user, typically allowing only insertion and the ability to access or delete only one particular data item. These accessible items are

- For stacks: The last item inserted
- For queues: The first item inserted
- For priority queues: The item with the highest priority (and the first item among items with the same priority)

These abstract data types (ADTs) can be seen as conceptual aids. Their functionality could be obtained using the underlying structure (such as an array) directly, but the well-defined interface makes it clear to any programmer that their purpose is to provide fast access in a specific order. For example, imagine two similar programs, one with an array (`list` in Python) and one with a queue object. They might perform the exact same operations on the underlying cells, but the very presence of the queue signals that the data will be handled in first-in, first-out (FIFO) order.

The main drawback of these ADTs is that they can't be conveniently searched for an item by key value. They can be traversed easily, but only the priority queue can be quickly traversed in key order (if it is not implemented as a heap).

Stack

A stack is used when you want access only to the last data item inserted; the other items are accessed only after doing something with the last data item. It's a last-in, first-out (LIFO) structure. It also allows fast peeking at the first item inserted on the stack but not deleting it.

A stack is often implemented as an array or a linked list. The array implementation is efficient because the most recently inserted item is placed at the end of the array, where it's also easy to delete. Stack overflow can occur but is not likely if the array is adequately sized.

If the stack will contain a lot of data and the amount can't be predicted accurately in advance (as when recursion is implemented as a stack), a linked list is a better choice than an array. A linked list is efficient because items can be inserted and deleted quickly from the head of the list. Stack overflow doesn't occur until the entire memory is full.

Implementing a stack with a linked list is slightly slower than with an array because memory allocation is necessary to create a new link for each insertion, instead of allocating all the space needed at the beginning. Deallocation of the links is necessary at some point following removal of an item from the list. This might be done explicitly by the program or through a process known as garbage collection (both of which take time). The linked list implementation can either be more or less memory efficient than the array implementation. Linked lists require storage of the pointers to the next link, while arrays can have many unused cells if the stack size is much smaller than what was allocated.

Queue

Use a queue when you want access only to the first data item inserted; it's a first-in, first-out (FIFO) structure.

Like stacks, queues can be implemented as arrays or linked lists. Both are efficient with O(1) insertion and deletion. The array requires additional programming to handle the situation in which the queue wraps around at the end of the circular array. A linked list must be double-ended to allow insertions at one end and deletions at the other. Both array and double-ended linked list implementations also allow O(1) access to peek at the last item inserted but not to delete it.

As with stacks, the choice between an array implementation and a linked list implementation is determined by how well the maximum number of items can be predicted. Use the array if you know about how much data there will be; otherwise, use a linked list.

Priority Queue

A priority queue is used when the only access desired is to the data item with the highest priority. This is the item with the largest (or sometimes the smallest) key.

Priority queues can be implemented as an ordered array or as a heap. Insertion into an ordered array is slow, but deletion is fast. With the heap implementations, both insertion and deletion take O(log N) time. The drawback of the heap implementation is that it does not preserve FIFO ordering among items with equal keys (without including insertion time as part of the priority key).

The heap implementation is almost always faster than implementing as an ordered array. Memory usage of these types varies but is always O(N). If the maximum number of items can be predicted in advance, the sorted array needs the least memory. If the size is hard to predict, both ordered arrays and heaps can be grown as needed without degrading overall insertion time but at the expense of having more unused memory.

Use an ordered array if FIFO ordering of equal keyed items is required, and it's difficult to reliably prioritize items by insertion time. For all other priority queues, the heap is the best choice.

Comparison of Special-Ordering Structures

Table 16-2 shows the Big O times for insertion and deletion on stacks, queues, and priority queues and whether the items can be traversed according to the order of their keys in O(N) time. These structures don't support fast searching.

Table 16-2 Special-Ordering Data Structure Speeds

Data Structure	Insertion	Deletion	Traversal in Key Order	Comment
Stack (array or linked list)	O(1)	O(1)	No	Deletes most recently inserted item (LIFO)
Queue (array or linked list)	O(1)	O(1)	No	Deletes least recently inserted item (FIFO)
Priority queue (ordered array)	O(N)	O(1)	Yes	Deletes highest-priority item
Priority queue (heap)	O(log N)	O(log N)	No	Deletes highest-priority item

Sorting

The widespread availability of libraries with good sorting capabilities has made writing sorting methods rare. As a first step, look for well-tested libraries and try them on the problem data. If you can't find one that performs well enough, you may need to implement your own.

For data that will fit in memory, it's worthwhile initially to try a slow but simple sort, such as the insertion sort. Your computing platform might have enough processing speed to sort your data in a reasonable time. Remember to try it with many different initial orderings of the data because the performance of insertion sort (and other algorithms) can vary significantly with the input ordering.

Insertion sort is the best of the simple sorting algorithms we saw in Chapter 3, "Simple Sorting," and is especially good for almost sorted data, operating in about O(N) time if not too many items are out of place. For example, when adding 10 items to an already-sorted file of 20 million, insertion sort should be very fast.

If the insertion sort proves too slow, then it's likely you'll need an O(N×log N) algorithm, like those we introduced in Chapter 7, "Advanced Sorting." These can be significantly more complex, so having a simple but slow algorithm like insertion sort is good for testing the correctness of your next method.

You could try the Shellsort next. It's fairly straightforward to implement and should get the performance to $O(N^{3/2})$. You need to implement an interval sequence, and Knuth's is likely to be fine.

If the Shellsort proves too slow, you should use one of the more complex but faster sorts: mergesort, heapsort, or quicksort. Mergesort requires extra memory and is the best choice if the data will not fit in memory. When there's not enough RAM, the merging can be done by reading items one at a time from two sorted files and outputting the lowest item to a third file. The file merging is often paired with an in-memory sorting method like Shellsort or heapsort to sort small sections of the initial file data and write them to disk

For in-memory sorting, the heapsort described in Chapter 13, "Heaps," has distinct advantages. The data can be sorted within an initial array without requiring extra memory. It sorts the data in O(N×log N) time regardless of the initial data ordering. A side benefit is that the heapify method used to make the initial array into a heap can be used for making heaps in O(N) time, enabling calculations like order statistics without a full sort. The disadvantage of heapsort is its somewhat more complicated algorithm, which ends up making more comparisons than quicksort, although both are O(N×log N) overall.

If complex algorithms are not a deterrent, Timsort is also an option. While the average performance of Timsort is also O(N×log N), it has the added benefit of running in O(N) time for initially sorted data.

Table 16-3 summarizes the running time for the sorting algorithms. The column labeled Algorithm Complexity attempts to categorize the level of difficulty in understanding and correctly implementing the algorithm. This, of course, depends greatly on the programmers doing the implementation. What's simple for some people can be complex for others.

Table 16-3 Comparison of Sorting Algorithm Speeds

Sort	Best	Average	Worst	Algorithm Complexity	Extra Memory
Bubble	$O(N^2)$	$O(N^2)$	$O(N^2)$	Simple	No
Selection	$O(N^2)$	$O(N^2)$	$O(N^2)$	Simple	No
Insertion	O(N)	$O(N^2)$	$O(N^2)$	Simple	No
Shellsort	$O(N^{3/2})$	$O(N^{3/2})$	$O(N^{3/2})$	Moderate	No
Quicksort	O(N×log N)	O(N×log N)	$O(N^2)$	Difficult	No
Mergesort	O(N×log N)	O(N×log N)	O(N×log N)	Moderate	Yes
Heapsort	O(N×log N)	O(N×log N)	O(N×log N)	Difficult	No
Timsort	O(N)	O(N×log N)	O(N×log N)	High	Yes

Specialty Data Structures

Many data structures are created to address particular problem areas. They work well for their designed goals but can't really serve as general-purpose data stores. Although they may support insertion, deletion, search, and traversal of items, they differ in their primary operations of interest. You can turn to these structures based on what operations are needed.

Quadtrees and Grids

When the data items have a spatial component, located in either a two-dimensional space such as a map or in three dimensions, you need structures that enable fast searches of those spaces. Quadtrees and grids offer a way of capturing the spatial information of points or regions so that items can be found by position in a two-dimensional space. Octrees offer a similar structure for items in three-dimensional space.

By organizing the items by their coordinates, quadtrees enable fast searches for the items nearest to or the items within a certain radius of a given point. Similar to binary trees, these kinds of searches (plus insertion and deletion) can be done in O(log N) time. This is much faster than a simple list of points or grids when there are large numbers of points that cluster in a few regions.

Grids can perform very well for smaller collections of points. They can outperform quadtrees when the points are close to a uniform distribution within known bounds.

Graphs

Graphs can be used to model real-world situations. The structure of the graph reflects the structure of the problem such as in transportation and communication networks. They are also used in other algorithms such as the PageRank algorithm—the initial ranking mechanism of Google's search results. The interlinked structure of websites can be modeled with a directed graph.

When you need a graph, nothing else will do, so there's no decision to be made about when to use one. The primary choice is how to represent the graph: using an adjacency matrix or adjacency lists. Your choice depends on whether the graph is dense, when the adjacency matrix is preferred, or sparse, when the adjacency list or adjacency hash table should be used.

The special operations that graphs handle include finding paths, trees, and subgraphs. The depth-first search and breadth-first search for a particular vertex or kind of vertex run in $O(V^2)$ time, where V is the number of vertices, for adjacency matrix representation. They run in O(V+E) time, where E is the number of edges, for adjacency list representation. Finding minimum spanning trees and shortest paths run in $O(V^2)$ time using an adjacency matrix and $O((E+V)\times\log V)$ time using adjacency lists. The number of edges can be on the order of V^2 in dense graphs, so you need to estimate V and E for your graph and do the arithmetic to see which representation is appropriate.

External Storage

For most of the data storage structures, we assumed that data was kept in main memory. When the data is too large to fit in memory, some or all of it must be stored in external storage, which often means disk files. We discussed external storage in the second parts of Chapter 9, "2-3-4 Trees and External Storage," and Chapter 11, "Hash Tables."

When data is stored in a disk file, we assumed that it was organized in fixed-size units called blocks, each of which holds a certain number of records. A record in a disk file holds the same sort of data as an object in main memory. Like an object, a record has at least one key value used to access it.

We also assumed that reading and writing operations always involve a single block, and these read and write operations are far more time-consuming than any processing of data in main memory. For fast operation, the number of disk accesses must be minimized.

Sequential Storage

The simplest approach to external storage is to store records randomly and read them sequentially when searching for one with a particular key. New records can simply be inserted at the end of the file. Deleted records can be marked as deleted, or records can be shifted down (as in an array) to fill in the gap.

On the average, searching and deletion involve reading half the blocks, so sequential storage is not very fast, operating in O(N) time, where N is the number of blocks. Still, it might be satisfactory for a small number of records.

Indexed Files

Speed increases dramatically when indexed files are used. In this scheme an index of keys and corresponding block numbers is kept in main memory. To access a record with a specified key, the index is consulted. It supplies the block number for the key, and only one block needs to be read, taking O(1) time.

Several indices with different kinds of keys can be used (one for last names, one for telephone numbers, and so on). This scheme works well until the index becomes too large to fit in memory. When that is the situation, the index files are themselves stored on disk and read into memory as needed.

The disadvantage of indexed files over sequential storage is that the index must be created and maintained. This process involves reading through all the records sequentially, so creating the index could be slow, if it's created after many external records have been stored. Also, the index needs to be updated when items are inserted into and deleted from the file.

B-trees

B-trees are multiway trees, commonly used in external storage, in which nodes correspond to blocks on the disk. As in other trees, the algorithms find their way down the tree, reading one block at each level. B-trees provide searching, insertion, and deletion of records in O(log N) time. This is quite fast and works even for very large files. The programming, however, is not trivial.

Hashing

If it's acceptable to use about twice as much external storage as a file would normally take, then external hashing might be a good choice. It has the same access time as indexed files, O(1), but can handle larger files.

Choosing Among External Storage Types

The choice of what scheme to use in managing external storage depends on the problem's characteristics. If multiple indices to the records must be maintained (for example, by name, by telephone, by address), using indexed files is really the only option. This option takes a bit longer for insertions and deletions because multiple indices must be updated for each operation, but it provides the fastest access for searches that use one or more of the indices.

If only a single index is needed, then B-trees or hashing become attractive. B-trees use less disk space, and hashing provides the fastest search access.

Virtual Memory

Sometimes you can let your operating system's virtual memory capabilities solve disk access problems with little programming effort on your part.

If you skipped the complications of B-trees or hashing and simply used sequential storage, you can read large disk files into an array. If that array is too big to fit in main (RAM) memory, the virtual memory system keeps part of that array in main memory and stores the rest on the disk. As you access different parts of the array, they are read from the disk automatically and placed in memory. This is typically done in chunks of memory called *pages*.

You can apply internal algorithms to the entire file contents by assuming the corresponding array was in memory at the same time, and let the operating system worry about reading the appropriate part of the file if it isn't in memory already. Operating systems have some sophisticated methods of predicting what pages of memory will be needed next and ensuring they are loaded quickly.

Of course, the operation on such a huge array is slower than when the entire array is in memory, but changing your algorithm to use B-trees or hashing to find records also takes time. It's unclear which will be faster, and it is quite possible for virtual memory paging to be faster or slower than one of the external-storage algorithms depending on the amount of data. Given that uncertainty, it may be worth simply ignoring the fact that a file doesn't fit in memory and seeing how well your algorithms work with the help of virtual memory. If the simple implementation doesn't perform well enough, then you can invest time in optimizing the external memory accesses.

Onward

We've come to the end of our survey of data structures and algorithms. The subject is large and complex, and no one book can make you an expert. We hope this book has given you a taste for them and taught you the fundamentals. If that taste intrigued you, many, many more data structures and variations on the ones described here await you. Some researchers spend their entire careers inventing, developing, and analyzing their performance. Those of us who simply use them as building blocks benefit by interpreting those analyses and comparing them to the requirements of our jobs.

Appendix B, "Further Reading," contains suggestions for further study.

This page intentionally left blank

APPENDIX A

Running the Visualizations

IN THIS APPENDIX A

- For Developers: Running and Changing the Visualizations
- For Managers: Downloading and Running the Visualizations
- For Others: Viewing the Visualizations on the Internet
- Using the Visualizations

We believe that active learning is the best way to master new material. Reading a book and attending a lecture provide the key concepts, but applying that knowledge by solving practice problems and performing experiments makes the abstract ideas more concrete.

To understand data structures and algorithms, it helps to see them in action. To that end, we have developed interactive visualizations of most of the data structures and algorithms described in this book. These programs show the step-by-step processing of data in each structure, allowing you to see all the key operations. You can try the structures with your own data to see how they handle it. The individual chapters suggest exercises for you to try with the visualizations to ensure that you see the important behaviors of each structure and algorithm.

There are several ways for you to run the visualization programs. The best way for each reader depends on how much programming you plan to do. For readers who plan to or already work as software developers, we encourage you to download the source code and run the programs on a computer with an editor or other integrated development environment. You can even help future students by contributing to the open-source software project for the visualizations. Your experience can help suggest new ways of making the complex ideas easier to understand by others. Follow the instructions in "For Developers: Running and Changing the Visualizations."

If you don't plan to do much programming but still want to try running the visualizations on your own data, you can download them and run them on a personal computer.

This approach would be a good choice for someone who manages or analyzes software development, without doing much coding on their own. If that suits your needs, follow the instructions in "For Managers: Downloading and Running the Visualizations."

If you don't plan on doing any programming and have a connection to the Internet, you can run the visualizations in a web browser. The programs were written in Python and display their visualization using Tk, which means they don't execute in a browser. That adds extra layers of interface that somewhat limits what you can do and see. For someone without access to a computer for software development, however, this may be the only option. If that's the case, follow the instructions in "For Others: Viewing the Visualizations on the Internet."

After you have been able to launch the visualizations, make sure to read the section "Using the Visualizations" at the end. This section explains some of the key features of all the visualizations and their user interfaces, including some limitations in the visualization that are not necessarily present in actual implementations of the data structures.

For Developers: Running and Changing the Visualizations

To maximize your ability to run the programs and modify them for your own experiments or to contribute to the open-source software project, you need at least three pieces of software on the computer where you plan to do the development:

- Python
- Git
- The visualization software

The first two come in different versions depending on your computer's operating system.

Getting Python

To get the Python programming language, you have many options. First, check whether it is already available on your system. If you're already familiar with command-line interfaces like the Windows Power Shell, the Windows Command interpreter, or macOS and Linux Terminals, try launching one of those interfaces and typing the command:

```
python3
```

If Python release 3 is installed, this should produce some lines that look something like:

```
Python 3.7.5 (default, Nov  4 2019, 10:26:32)
[Clang 9.0.0 (clang-900.0.39.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

If this code doesn't show up, then you probably need to install Python. There are many places on the Internet to get Python. The primary one, www.python.org/downloads, has all the releases of Python and many preconfigured downloads for the different operating systems. It's also the official source for all information about the programming language, the different releases, documentation, and proposed changes for the future. The visualizations were made using Python release 3. You need a version that starts with "3."; do not use one that starts with "2." or lower. If this is your first time using Python, you may want to follow the documentation links to read the guides for beginners.

Getting Git

The second tool, git, manages collections of software and their different versions. It's exceptionally well suited to managing source code because it provides many ways to track the changes in text files. Git became the tool of choice for the developers of the Linux operating system. One of its key features is the distributed nature of the software collections, called *repositories*, that it manages. While github.com maintains a copy of most of these repositories, every developer gets their own complete repository including all the history of changes, in their local copy of the repository.

The `git` command-line tool is incredibly powerful, and not surprisingly, incredibly complex. It was invented by software professionals for use by software professionals. Its complexity, however, can be a hindrance for newcomers. Many users just need the basics, and graphical user interfaces can make that much easier. The GitHub Desktop tool provides a straightforward graphical interface to working with repositories, their branches, and commits (changes) to those branches.

Learning how to use the `git` command-line tool is beyond the scope of this book. Instead, we describe how to use GitHub Desktop. If you're already familiar with the `git` command-line tool, however, go ahead and use that, because it can do everything GitHub Desktop does and more.

Use your web browser to visit https://desktop.github.com. It should auto-detect the operating system of your computer and propose a download option. At the time of this writing, only Windows and macOS are supported (on Linux, you need to use the `git` command-line tool, which is often bundled with the Linux distribution). Follow the download instructions to install GitHub Desktop.

Both GitHub Desktop and the `git` command-line tool allow you to review the history of changes in a repository and make new changes to the files it contains. In the next section, we discuss how to get the repository that contains the visualizations for this book. If you plan to make contributions to the open-source visualization project, you also need an account on the GitHub.com website. That is where the project's repository is stored, and your user account allows you to contribute to many different projects. Go to https://github.com and select the Sign up button to create a new account.

Getting the Visualizations

The third piece of software you need is the visualization code itself, which is bundled in a git repository available at https://github.com/JMCanning78/datastructures-visualization.

The process of downloading a git repository is called *cloning* the repository. If you are using Github Desktop, go to the File > Clone Repository dialog. Choose the GitHub.com tab and then start typing in the repository name, `JMCanning78/datastructures-visualization`. The repository should pop up as a choice that you can select. The dialog lets you specify a Local Path on your computer where it will store the clone. After selecting the repository on GitHub.com and specifying the local path, select the Clone button to make the clone.

If you're using the command-line version of `git`, the best way to get the visualization software is to put one of your command-line windows into the parent directory where you want to store the software and then run the `git clone` command. For example, if you wanted to store the repository in `/Users/Me/myrepositories`, the commands would look something like this:

```
cd /Users/Me/myrepositories
git clone https://github.com/JMCanning78/datastructures-visualization
```

The `cd` command is short for change directory. Your command-line tool might have a different name for the change directory command. The `git clone` command creates a subdirectory called `datastructures-visualization` that contains a clone of the git repository. You can use the `cd` command again to change the current working directory to be that subdirectory. From that directory, you can launch the visualizations by running

```
python3 DatastructureVisualizations.py
```

If you get a message about missing required modules, you may need to add some modules to your Python environment. Try running the Python installer program (`pip3`), as follows:

```
pip3 install -r requuirements.txt
```

If you find that your computer doesn't have `pip`, there is help on ways to install it available at https://pip.pypa.io/en/stable/. When `pip3` runs successfully, try the previous command to run the visualizations.

If you prefer to use an Integrated Development Environment (IDE) for your Python development work, you need to register the cloned repository with the IDE. From there, you should be able to run `DatastructureVisualizations.py` or any other Python programs.

For Managers: Downloading and Running the Visualizations

If you have a macOS or Windows desktop computer, you may be able to download a prepackaged collection of all the visualizations. It runs basically the same program as the DatastructureVisualizations.py described previously. Browse https://datastructures.live/ and look for the links related to downloads.

The software bundles are cryptographically signed to help ensure the integrity of the software. Your computer and its malware protection software may ask whether you trust the downloaded visualization software before allowing you to run it.

For Others: Viewing the Visualizations on the Internet

You can view the visualizations by visiting https://datastructures.live/view/ in a web browser. When you select the Run button, the website runs the visualizations on a server and copies the output to a pane in your web browser window. The initial screen looks something like Figure A-1.

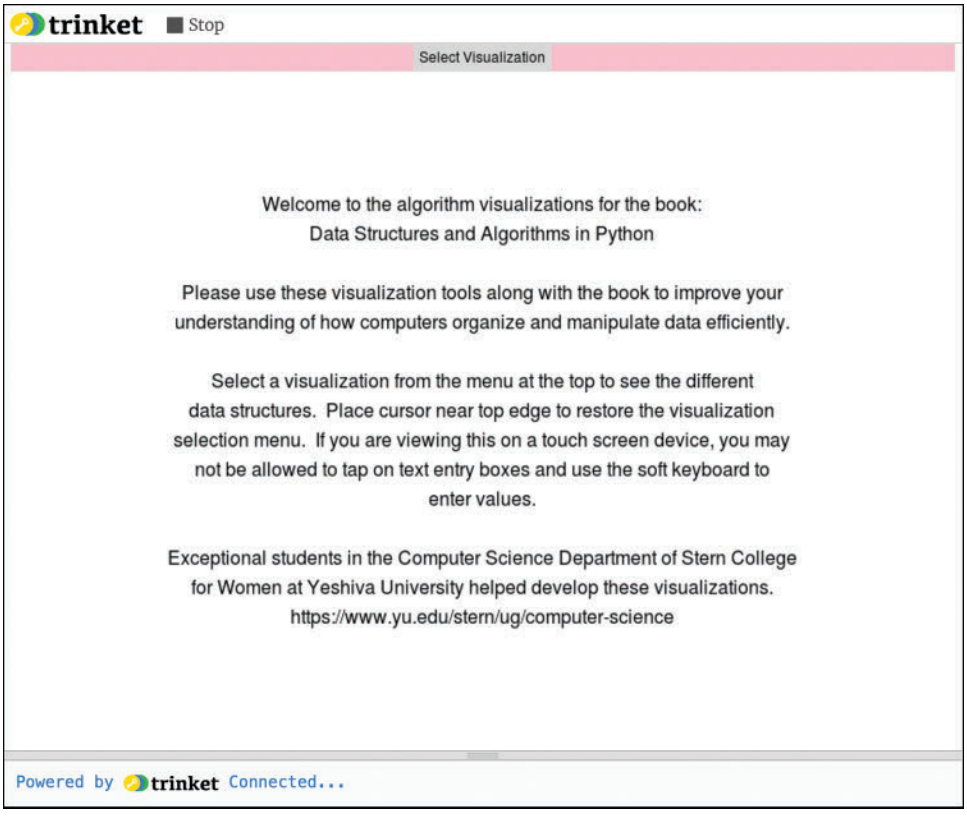


FIGURE A-1 Initial screen when viewing visualizations in a web browser

Select the Select Visualization button at the top to get a menu of all the visualizations that can be shown.

There are limitations on viewing the visualizations in a web browser. In particular:

- The window cannot be resized although you can increase the size of the upper pane by adjusting the size of the web browser window and dragging the separator bar just above the "Powered by trinket" text.
- The screen is being copied over the Internet, so there might be choppy or slow updates in response to interactions. Try using the visualization tool's lower animation speed or stepping mode when this occurs.
- The viewing works best on computers with a mouse and keyboard. Viewing the trinket.io powered website on a device with only a touchscreen might not allow you to enter text for the arguments to operations. Selecting a text entry area does not put the keyboard focus on that area.

Using the Visualizations

The visualizations are small programs that usually show one data structure at a time. They all have a similar structure with a drawing area at the top and a set of controls on the bottom (see Figure A-2). There are specific instructions for how to use the visualizations in each of the chapters, and some general guidelines common to all of them follow.

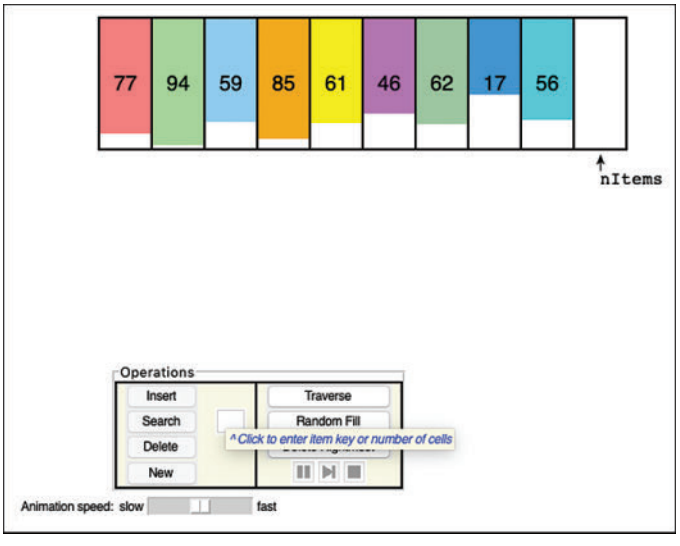


FIGURE A-2 Drawing area at the top and a set of controls on the bottom

The visualizations show the data in a drawing area at the top, such as in Figure A-2. The Operations box in the lower left is where you initiate operations on the data structure. The buttons are grayed out and disabled when they are not applicable. In particular, the operations that take an argument like a key to insert are on the left side of the Operations box. They are grouped with a text entry area that must have a value entered before they become enabled. The hint in blue text explains what can be typed in the text entry area

for the different operation arguments. Operations that take no arguments, like Traverse and Random Fill, are on the right.

Below the Operations box is a slider. This control can be adjusted to slow down or speed up the animation of the operations. During animations, the three small buttons on the lower right ▶ ▶| ■ are enabled to provide fine control. The leftmost of three buttons is either a triangle to resume playback, or two vertical bars ❚❚ ▶| ■ to pause the animation, depending on the current state. The second button (with a triangle pointing into a vertical bar) steps forward in the animation by one "step." That button is enabled only when code is displayed because the steps are defined by the code. The square button stops the current operation and enables other operations to be run. Stopping an operation before it finishes can leave the data structure in an odd state, such as leaving extra copies of data items or not sorting all the items.

During most operations, a code box appears in the lower right (see Figure A-3). It shows the same program described in the book, highlighting the part of the program being executed. In the figure, the items in an array are being sorted. The various local variables are shown along with the data structure in the drawing at the top. Only the three animation buttons are enabled while the sort operation runs. The highlighting around the single step button indicates that keyboard focus is on that button. Pressing the spacebar when a button has keyboard focus "presses" the button. Operations normally begin running in the "play" mode when a button is pressed. By holding down the Shift key when pressing the button, you can start the operation in stepping mode (if it displays code).

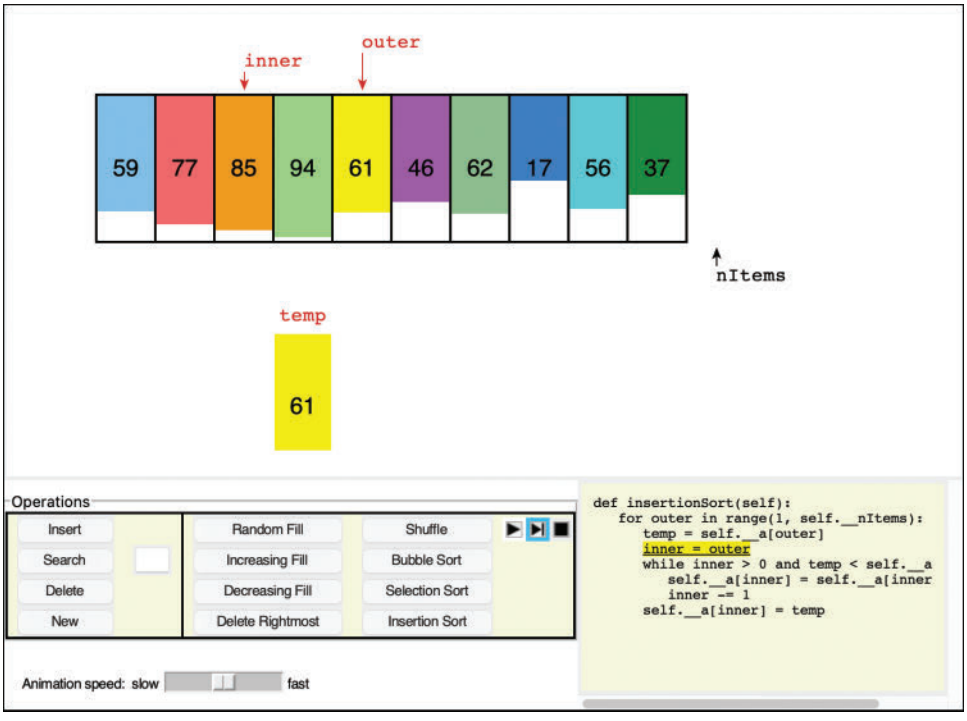


FIGURE A-3 The code box in the lower right

After each operation finishes, the code and local variables remain on the display until the next operation is started. Status or error messages can appear at the bottom of the window, between the sliding speed control and the code box. The messages usually appear when operations finish. Figure A-4 shows the display after the successful search for key 62 in the array.

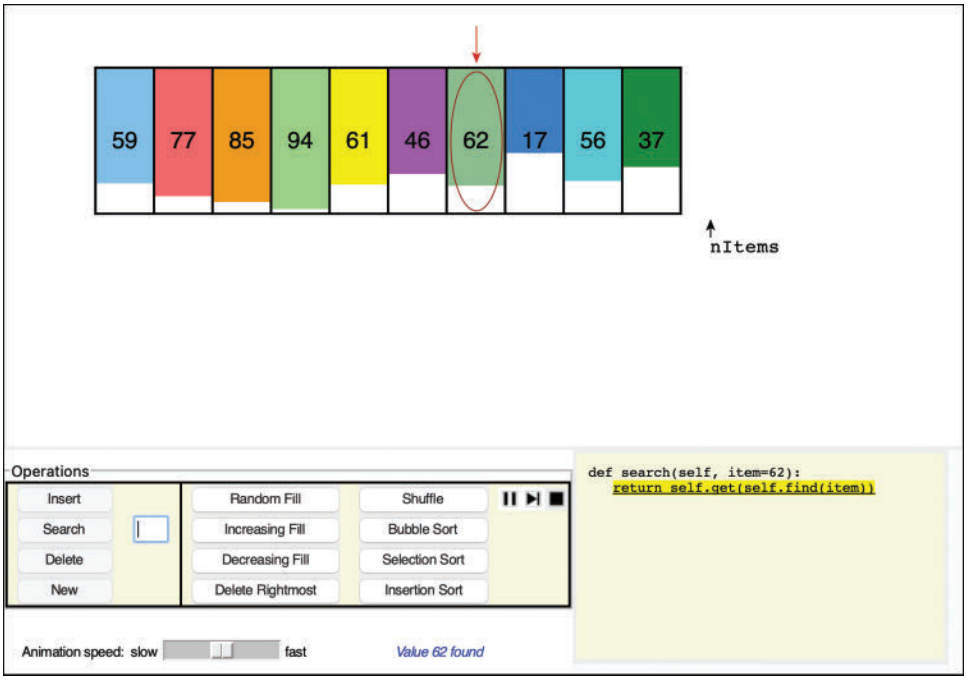


FIGURE A-4 Results of the search for key 62 in the array

In most visualizations, you can click a data item to select it and copy its value into the text entry area. This makes it less likely that a typographical error will prevent searching for or deleting an existing value.

Hint messages about the controls can appear next to the text entry box(es) and operations buttons. Hovering the mouse pointer over one of them displays a short hint about what values can be entered in that box or what the button does. When you first start one of the visualizations, it displays a message telling you all the kinds of things that can be entered in all the text entry box(es), like the message in Figure A-2.

Depending on how you run the visualization, you may be able to resize the window. This capability is sometimes useful when the amount of data grows too large or the lines of code are wide. The code text box does not grow vertically but can be scrolled for longer programs. Some visualizations require more display space and provide scrollbars to move around. At least one visualization also lets users zoom in and out on particular regions.

APPENDIX B

Further Reading

IN THIS APPENDIX B

- Data Structures and Algorithms
- Object-Oriented Programming Languages
- Object-Oriented Design (OOD) and Software Engineering

Because there are so many good books on computer science, data structures, and algorithms, it's hard to choose what's right to recommend. If we've sparked your interest to read further, then we've succeeded in one of our goals. There are many paths from which to choose. Here are a few that we've found to be especially good, organized into topic areas.

Data Structures and Algorithms

The definitive reference for any study of data structures and algorithms is *The Art of Computer Programming* by Donald E. Knuth, of Stanford University (Addison-Wesley, 2011). This seminal work was originally published in the 1970s and was cited as a major part of his Turing award in 1974. It currently consists of four volumes: *Volume 1: Fundamental Algorithms*, *Volume 2: Seminumerical Algorithms*, *Volume 3: Sorting and Searching,* and *Volume 4A: Combinatorial Algorithms, Part 1*. Of these, Volume 3 is the most relevant to the topics in this book. This work is highly mathematical and does not make for easy reading, but it is the bible for anyone contemplating serious research in the field. As of this writing, Prof. Knuth continues working on Volume 4B of the series, releasing fascicles from time to time.

A somewhat more accessible text is Robert Sedgewick and Kevin Wayne's *Algorithms* (Addison-Wesley, 2016). This fourth edition of the book contains code examples written in Java and is adapted from earlier editions in which the code examples were written in Pascal, C, and C++. It is comprehensive and authoritative. The text and code examples are quite compact and require close reading.

Every computer scientist needs to have and read the encyclopedic *Introduction to Algorithms*, by Cormen, Leiserson,

Rivest, and Stein (MIT Press, 2009). The book provides clear, concise coverage of a startling range of topics, and is language agnostic—usefully providing algorithms in a generic pseudocode.

Programming Pearls by Jon Louis Bentley (second edition, Addison-Wesley, 2000) was originally written in 1986 but is nevertheless stuffed full of great advice for the programmer. Much of the material deals with data structures and algorithms.

We've covered high-level data structures and algorithms, but there's a whole world of fascinating low-level bitwise data structures and algorithms that are still surprisingly useful in modern programming. For a great survey of such techniques, try *Hacker's Delight* by Henry S. Warren Jr. (Addison-Wesley, 2013).

Recursion is a recurring theme throughout our book. Readers who are new to the concept and practice of writing recursive code can benefit by practicing recursively coding diverse problems. While these books address recursive solutions in Java, they are unbeatable sources of challenging problems: *Thinking Recursively with Java*, by Eric Roberts (Wiley, 2006), and *Practicing Recursion in Java*, by Irena Pevac (CreateSpace Independent Publishing, 2016).

Object-Oriented Programming Languages

For a clear, beautifully structured, step-by-step introduction to the concepts of object-oriented programming in Java, it is worth seeking out the now out-of-print *On to Java* by Patrick Henry Winston and Sundar Narasimhan (Pearson, 2002). One can only hope for a Python version of this classic book one day!

To get started in Python, try the comprehensive *Building Python Programs* by Reges, Stepp, and Obourn (Pearson, 2019). The book and its companion website provide extensive opportunities to methodically learn and hone your Python programming skills.

When you're ready to master Python's clever features, *Python Tricks—The Book*, by Dan Bader (Dan Bader, 2017) is a great resource. The book is filled with many self-contained nuggets of Python wisdom that you can incrementally add to your repertoire, including how to best use Python's object-oriented features.

If you're already an experienced Python programmer, then *Python Distilled*, by David M. Beazley (Addison-Wesley Professional, 2021), should always be kept in reach on your desk for quick reference. This compact yet authoritative book contains almost everything you'll need to know about making the most effective and appropriate use of Python's most advanced features.

Object-Oriented Design (OOD) and Software Engineering

For an easy, nonacademic introduction to software engineering, try *The Object Primer: The Application Developer's Guide to Object-Orientation, Second Edition,* by Scott W. Ambler (Cambridge University Press, 2001). This short book explains in plain language how to

design a large software application. The title is a bit of a misnomer; it goes way beyond mere OO concepts.

Object-Oriented Design in Java by Stephen Gilbert and Bill McCarty (Waite Group Press, 1998) is an unusually accessible text.

A classic in the field of OOD is *Object-Oriented Analysis and Design with Applications* by Grady Booch et al. (Addison-Wesley, 2007). The author is one of the pioneers in this field and the creator of the Booch notation for depicting class relationships. This book isn't easy for beginners but is essential for more advanced readers.

An early book on software design is *The Mythical Man-Month* by Frederick P. Brooks, Jr. (Addison-Wesley, 1996), which explains in a very clear and literate way some of the reasons why good software design is necessary. Since its original publication in 1975, it is said to have sold more copies than any other computer book.

This page intentionally left blank

APPENDIX C

Answers to Questions

IN THIS APPENDIX C

- Chapter 1, "Overview"
- Chapter 2, "Arrays"
- Chapter 3, "Simple Sorting"
- Chapter 4, "Stacks and Queues"
- Chapter 5, "Linked Lists"
- Chapter 6, "Recursion"
- Chapter 7, "Advanced Sorting"
- Chapter 8, "Binary Trees"
- Chapter 9, "2-3-4 Trees and External Storage"
- Chapter 10, "AVL and Red-Black Trees"
- Chapter 11, "Hash Tables"
- Chapter 12, "Spatial Data Structures"
- Chapter 13, "Heaps"
- Chapter 14, "Graphs"
- Chapter 15, "Weighted Graphs"

Index

- Chapter 1, "Overview"
- Chapter 2, "Arrays"
- Chapter 3, "Simple Sorting"
- Chapter 4, "Stacks and Queues"
- Chapter 5, "Linked Lists"
- Chapter 6, "Recursion"
- Chapter 7, "Advanced Sorting"
- Chapter 8, "Binary Trees"
- Chapter 9, "2-3-4 Trees and External Storage"
- Chapter 10, "AVL and Red-Black Trees"
- Chapter 11, "Hash Tables"
- Chapter 12, "Spatial Data Structures"
- Chapter 13, "Heaps"
- Chapter 14, "Graphs"
- Chapter 15, "Weighted Graphs"

Chapter 1, "Overview"

1. insert, search, delete, traverse
2. c
3. If it is indented below a `class` definition with no intervening definition at the outer (unindented) level.
4. sorting

5. Constructors are functions that build an instance of an object class. Python uses the `__init__()` method of a class as the constructor.
6. You choose data structures that model the system and behavior being represented by the program. Among data structures that model the system equally well, you choose the ones that perform the most efficiently. Among those that perform equally fast, you choose ones that take the least amount of memory space.
7. A key field is used to find a larger record, typically by keeping a sorted structure of keys to accelerate the search.
8. It can reduce the amount of memory needed to store the data. It makes the program easier to understand, and hence, easier to maintain by other programmers.
9. To implement data structures and tightly organize algorithms with the data they manipulate.
10. b, d, & g

Chapter 2, "Arrays"

1. They should have `get()` and `set()` methods to access the value of a particular array cell. The ability to get and set cells indexed by an integer value is the basic functionality of arrays in all languages.
2. d. Some languages don't require specifying the maximum number of cells at construction time, but most do.
3. to prevent access to internal values of the object instance from being manipulated by code outside the object.
4. d
5. True
6. b
7. False
8. d
9. a
10. Both `find()` methods return a numeric index to a cell in the ordered array containing an item being sought. If the item being sought is not in the array, `find()` returns the index of the next item just after the sought item, if any. Both `search()` methods return the item stored at the array index discovered by `find()` or `None` if the item or key could not be found. The versions in `OrderedRecordArray.py` search by key instead of the item itself.
11. raising to a power
12. 3

13. b
14. 6
15. False
16. a
17. constant
18. c
19. b
20. The number of records in the current, solar-system-based implementation is good for tens, maybe hundreds of records. For potentially billions of records that will need to be accessed in "random" order, a search method that is faster than O(N) is needed. To get the O(log N) search speed, an ordered array is required, at a minimum. In fact, even more complex data structures are needed to get the search time down to something reasonable. The size of the individual records is not an important factor here.

Chapter 3, "Simple Sorting"

1. d
2. comparing and swapping (or copying/moving)
3. False
4. a
5. False
6. b
7. False
8. A swap would take three times longer than a copy operation because a swap involves three separate copy operations, assuming no concurrent copying.
9. Items with indices less than or equal to `outer` are sorted.
10. c
11. d
12. copies
13. b
14. Items with indices less than `outer` are partially sorted.
15. b

    Experiment 3-A: The bubble sort and insertion sort algorithms are stable. The selection sort is not stable. For example, the list `[('elm', 1), ('elm', 2),`

`('asp', 1)]` sorts to `[('asp', 1), ('elm', 2), ('elm', 1)]` when sorting by the first element.

Chapter 4, "Stacks and Queues"

1. 10
2. b
3. last-in, first-out and first-in, first-out
4. False
5. Neither; it doesn't move.
6. 45
7. False
8. c
9. O(N)
10. c
11. True
12. b
13. Yes. The `find()` method would need to be modified to find the lowest index into the array for the priority of the item being inserted (assuming high indices are at the front of the queue and highest priority). That's needed to maintain the FIFO ordering of the items with equal priority. The binary search algorithm in the `find()` method stops at the first index it finds with the matching priority key.
14. a (and maybe b if some keystrokes can have higher priority and need to be processed ahead of others in the queue)
15. a

Chapter 5, "Linked Lists"

1. c
2. first (called `__first` in `LinkedList`)
3. d
4. 2
5. 1 plus setting the `__next` field to `None` in the new link, which really is a nonreference.
6. b

7. `x.setNext(None)`
8. When the variable, `x`, is passed as a reference instead of a copy of its value
9. a
10. c
11. a linked list
12. once
13. a double-ended list
14. an ordered double-ended list
15. d
16. Usually, the list. They both do `push()` and `pop()` in O(1) time, but the list uses memory more efficiently.
17. The array. Binary search allows O(log N) time to find the item by key, but if you use a list, even a doubly linked one, each search step has to walk through all the intermediate links to get to the next item to test.
18. b
19. Calling programs can change the link's next pointer to create or break circular lists, regardless of whether the list type supports them.
20. a

Chapter 6, "Recursion"

1. c
2. 101
3. d
4. 2. The base case returns 0, and `nth` has a value of 1 in the caller to the base case. That caller returns the first non-base case value of 1. The value of `nth` right after 1 is returned is in the call where `nth` has the value 2.
5. True
6. `"ue"`
7. c
8. moving a smaller number of disks and reassigning the role of each spindle
9. b
10. b

11. b
12. 11. There are calls to `mergesort()` for array sizes 1,024, 512, 256, 128, 64, 32, 16, 8, 4, 2, and 1 on the stack at one point.
13. stack
14. Both function arguments and local variables of a recursive function must be stored for each call that happens in the recursive execution. These can be stored in the problem descriptions and kept on a stack.
15. when the stack of problem descriptions is empty

Chapter 7, "Advanced Sorting"

1. c
2. 40
3. d
4. The complexity could degenerate to $O(N^2)$.
5. $O(N \times \log N)$, $O(N^2)$
6. a
7. pivot
8. d
9. True
10. c
11. partitioning the resulting subarrays recursively (to fully sort the array)
12. b
13. pivot
14. $\log_2 N$
15. For data that is initially forward or reverse sorted, the complexity becomes $O(N^2)$.
16. True
17. a
18. O(N). It needs 10 linked lists and those lists must collectively store a full copy of the N elements.
19. d
20. b only, and it's a modified version of mergesort.

Chapter 8, "Binary Trees"

1. O(log N)
2. c
3. False
4. b
5. 5
6. c
7. node, (binary search) tree
8. a
9. d
10. False
11. finding
12. b
13. linked list
14. A's key, the key of A's right child
15. d
16. d
17. $2 \times n+2$
18. False
19. compress
20. c

Chapter 9, "2-3-4 Trees and External Storage"

1. b
2. lower in height; always balanced
3. 2
4. False
5. b
6. the root is split
7. a

8. 2
9. b
10. O(log N)
11. d
12. one block's worth of
13. 2-3
14. a
15. c

Chapter 10, "AVL and Red-Black Trees"

1. in order (or inverse order) of the keys
2. b
3. c
4. c
5. a
6. b
7. False
8. d
9. b
10. rotations, changing the colors of nodes (swaps and flips)
11. red
12. left child, right child
13. d
14. a node (black), its two children (red)
15. b
16. True
17. a
18. d. Only one color swap is needed for a black node with two red children on insertion.
19. d
20. d

Chapter 11, "Hash Tables"

1. O(1)
2. hash function
3. the modulo operation
4. a collision
5. d
6. linear probing
7. (0), 1, 4, 9, 16, 25
8. b
9. a
10. linked list or binary search tree
11. 1.0
12. True
13. be a prime number
14. the array size
15. all of them; for the same key, an unsuccessful search must go through the whole probe sequence, whereas a successful search stops with the found item.
16. b
17. b
18. A sorted linked list is most efficient with one empty link for all N items. Next most efficient is the hash table at 0.6 load factor, which has $0.4 \times N$ empty or deleted cells. The AVL tree has two empty links for all its leaf nodes, which is up to $(N + 1)/2$, and one empty link for some of the parents of the leaf nodes. For a complete, balanced binary tree, there would be exactly $(N + 1)/2$ leaf nodes and no empty links among the internal nodes. Overall, the total would be close to $2 \times N/2$ or N empty cells. The actual count in a specific tree could vary from that, but it is not going to get below $N/2$, which is still more than $0.4 \times N$.
19. False
20. the same block

Chapter 12, "Spatial Data Structures"

1. The keys are a combination of two numbers, rather than a single integer or string.
2. Geographic or latitude-longitude

3. The distance function (and the separate structures for storing the points)
4. b
5. False
6. Only a portion of the spatial data is stored in each grid cell, reducing search time
7. Quadtrees can adapt to clusters in spatial data while grids cells may end up having some cells very full and others be empty. They have O(log N) search time instead of O(N) for grids and point lists.
8. d
9. a. O(N) b. O(N) c. O(log N)
10. False
11. d
12. False
13. The quadrants are the northeast, northwest, southwest, and southeast regions around a node's center. Points lying on the boundary of two quadrants are placed in the next quadrant in clockwise order.
14. d
15. False

Chapter 13, "Heaps"

1. b
2. Both the right and left children have keys less than (or equal to) the parent, but no ordering is specified between the child items.
3. root
4. d
5. a
6. c
7. array or linked list
8. sifting down because it must determine the maximum child key
9. b
10. one
11. O(N×log N)

12. slower because it takes more time per item but faster in some cases because it does not degenerate to $O(N^2)$
13. c
14. N
15. N + K×log N

Experiment 13-B: The heapsort algorithm is not stable. Consider an array of all equal keys such as 2, 2, 2, 2, 2, 2. It already meets the heap condition, so heapify doesn't reorder any items. The max (root) item is removed from the heap and goes to the last position of the array. The last of the remaining equal keyed items in the heap is then moved to the root and sifted down. Because all keys in the heap are equal, the item does not sift down and becomes the next item moved to the end. This repeats for all the equal keyed items, swapping the last for the first and then swapping them back again for each removal. Using the original indices of the six-element array—0, 1, 2, 3, 4, and 5 to track the item order—the final arrangement will be 1, 2, 3, 4, 5, 0.

Chapter 14, "Graphs"

1. edge, vertices (or sometimes, nodes)
2. Count the number of 1s or edges and divide by 2 (assuming the identity diagonal is all 0s).
3. vertex, edge
4. d
5. A:B, B:A➔C➔D, C:B, D:B➔C
6. a
7. Three. If the vertices are A, B, and C, the minimum spanning trees are defined by the paths ABC, BCA, and CAB.
8. Option b, depth-first search, is the fastest because it takes $O(N^2)$ time. Option a first finds an MST using depth-first traversal over the entire connected component and then takes more time to find whether Z is in the MST. Option c runs in $O(N^2)$ time but doesn't work in all directed graphs because a breadth-first search would find only A and Z in the same component if there was a cycle including A and Z. Option d works but takes $O(N^3)$ time to compute the connectivity matrix.
9. c
10. directed acyclic graph (DAG) or tree (directed or undirected)
11. b
12. No
13. True

14. d
15. Yes. The adjacency matrix is asymmetric, so the graph is undirected. The result would be an exception because the graph contains a cycle, ABDA.
16. directed acyclic graph
17. b
18. Only if the goal is to find vertices with no edges. Every vertex with an undirected edge has a predecessor. The result would find all the vertices with no edges with the rest essentially forming a cycle (if cycles like ABA are allowed for an undirected edge AB).
19. Using the improved topological sort algorithm, computing time is $O(N^2)$. Because the first graph has 10 times as many vertices, the ratio would be $10^2 = 100$. The basic topological search took $O(N^3)$, so its ratio would be $10^3 = 1{,}000$.
20. None of the algorithms mentioned in this chapter solve the bridges of Königsberg problem.

Chapter 15, "Weighted Graphs"

1. distances, costs, times
2. c
3. d
4. False (for undirected graphs)
5. the lowest-weight edge
6. b
7. False
8. is already in the MST
9. False
10. a
11. True
12. a
13. d
14. In Warshall's algorithm, connectivity matrix cells are updated using bitwise-OR rules, whereas in the Floyd-Warshall algorithm, the cells are updated using the minimum of their current value and that of adding two other cell values.
15. intractable
16. adjacency list, adjacency matrix

17. Floyd-Warshall
18. 2^N, where N is $K^2 - 1$, the number of squares on the board minus 1
19. No. The route mentioned in the "Traveling Salesperson Problem" section of the text had total cost of 140, whereas the proposed route costs 145. Neither is the minimum solution of the TSP.
20. when the cost (weight) of traveling in one direction between two vertices is not the same as traveling in the opposite direction

Index

NUMBERS

2-3 trees
  described, 432–433
  node splits, 433–434
    insertion and, 437–438
    promoting to internal nodes, 435–437
  speed of, 438
2-3-4 trees
  advantages/disadvantages, 5
  defined, 401–403
  insertion, 404–405
  node splits, 405–406, 407–408
  organization of, 403–404
  red-black trees and, 508–510
    operational equivalence, 510–514
    transformation between, 509–510
  root splits, 406–407
  searching, 404
  speed of, 431
  storage requirements, 432
  terminology, 403
  Tree234 class, 415
    deleting nodes, 423–430
    __Node class, 412–415
    node splits, 418–421
    searching, 415–417
    traversing, 421–423
  Tree234 Visualization tool, 408–411

A

abstract data types. *See* ADTs (abstract data types)
abstraction, described, 190–191
adjacency, defined, 707
adjacency lists
  modeling, 712–713
  storing edges in, 714–716
adjacency matrix
  as hash table, 712
  as two-dimensional array, 710–711
ADT lists, 191
ADTs (abstract data types)
  defined, 184, 190–191
  as design tool, 191–192
  as interface, 191
  priority queues and, 667–668
  when to use, 824–826
AdvancedSorting Visualization tool
  partitioning with, 295–297
  quicksorts with, 309–310, 318
  Shellsorts with, 289–291
algorithms
  defined, 1
  invariants, 82
  purpose of, 40–47
  recipe analogy, 1–3
  speed of
    Big O notation, 65–68
    general-purpose data structures, 819–820, 824
    sorting algorithms, 828
    special-ordering data structures, 826
allocation of arrays, 39
all-pairs shortest-path problem, 796–798
altering data structures during iteration, 216–217
anagrams, described, 239–242
analyzing the problem, 814–818
  amount of data, 815–816
  frequency of operations, 816–817
  software maintenance responsibilities, 817–818
  types of data, 814–815
arguments, defined, 20
arithmetic expressions, parsing
  described, 132–133
  evaluating postfix expressions, 148–151
  Infix Calculator tool, 142–148
  postfix notation, 133–134
  translating infix to postfix, 134–142
Array class
  accessing elements, 38–39
  bubble sorts, 81–82
  creating arrays, 37–38
  deletion, 42
  encapsulation, 42–43
  example of, 39–42
  improved example of, 43–47
  initialization, 39
  insertion, 42
  insertion sorts, 90
  searches, 42
  selection sorts, 85–86
  traversal, 42
Array Visualization tool, 30–37
  deletion, 34–35, 37
  duplicates, 35–37
  insertion, 33
  searches, 31–33
  speed of algorithms, 37
  traversal, 35
arrays. *See also* hash tables; hashing
  advantages/disadvantages, 5, 69, 820–821

Array class
accessing elements, 38–39
creating, 37–38
deletion, 42
encapsulation, 42–43
example of, 39–42
improved example of, 43–47
initialization, 39
insertion, 42
searches, 42
traversal, 42
Array Visualization tool, 30–37
deletion, 34–35, 37
duplicates, 35–37
insertion, 33
searches, 31–33
speed of algorithms, 37
traversal, 35
binary search trees as
described, 377–378
levels and size, 378–379
insertion, speed of, 66
linked lists vs.164
lists as, 37
Ordered Array Visualization tool, 47–51
binary searches, 49–51
duplicates, 51
Guess-a-Number game example, 48–49
OrderedArray class
advantages of, 57–58
example of, 53–57
find() method, 52–53
OrderedRecordArray class, 61–65
reusing in heapsort, 688–691
as sequences, 13–15
sorting. *See* sorting
two-dimensional
adjacency matrix as, 710–711
in Python, 713–714
use case comparison with stacks/queues, 103–104
arrival ordering, defined, 116–117
ASCII codes, 386
assignment statements, multivalued assignment in Python, 17–18
attributes
defined, 7
Python name mangling, 44
AVL trees
AVLtree class
deleting nodes, 479–484
inserting nodes, 474–478
__Node class, 472–474
AVLTree Visualization tool, 470
inserting nodes, 470–472
crossover subtrees in rotations, 478–479
defined, 463, 469–471
speed of, 484–485
AVLtree class
deleting nodes, 479–484
inserting nodes, 474–478
__Node class, 472–474
AVLTree Visualization tool, 470
inserting nodes, 470–472

B

balanced trees. *See also* AVL trees; red-black trees
defined, 463
degenerates, 463–464
measuring balance, 464–469
red-black rules and, 495
when to use, 822–823

base case, defined, 233
best case, defined, 266
Big O notation, 65–68
  2–3 trees, 438
  2–3-4 trees, 431–432
  AVL trees, 484–485
  binary search trees, 350, 375–377
  binary searches, 67
  bubble sorts, 82
  comparison of sorting methods, 96–97
  constants in, 67–68
  counting sorts, 324
  degenerates, 463–464
  exact point matches, 622–623
  general-purpose data structures, 824
  graphs, 798
  hashing, 581
    open addressing, 581–583
    separate chaining, 583–587
  heaps, 683–684
  heapsort, 693
  insertion in unordered arrays, 66
  insertion sorts, 91
  K highest, 696–700
  linear searches, 66–67
  linked lists, 183–184
  mergesorts, 264–267, 456
  ordered lists, 198
  partitioning algorithm, 301–302
  priority queues, 132
  quadtrees
    exact matches, 645
    insertion, 644
    nearest matches, 655
  queues, 125
  quicksorts, 318–320
  radix sorts, 322
  recursion, 236
  red-black trees, 508
  selection sorts, 86–87
  Shellsorts, 294
  sorting algorithms, 828
  spatial data searches, 656–658
  special-ordering data structures, 826
  stacks, 116
  Timsorts, 327
  topological sorting, 751
Binary Search Tree Visualization tool, 341–344
  deleting double child nodes, 374
  deleting leaf nodes, 367
  deleting single child nodes, 368–369
  finding nodes, 346–348
  inserting nodes, 351–352
  traversal with, 361–363
binary search trees. *See also* AVL trees; nodes; red-black trees
  as arrays
    described, 377–378
    levels and size, 378–379
  Binary Search Tree Visualization tool, 341–344
    deleting double child nodes, 374
    deleting leaf nodes, 367
    deleting single child nodes, 368–369
    finding nodes, 346–348
    inserting nodes, 351–352
    traversal with, 361–363
  BinarySearchTree class, 344
    deleting single child nodes, 369–370
    finding nodes, 348–349
    inserting nodes, 352–353
    __Node class, 345–346
    testing code, 382–385
    traversal with, 356–361
  defined, 340
  duplicate keys in, 381–382

hierarchical file system analogy, 340–341
minimum/maximum key values, 365–366
printing, 379–381
separate chaining with, 585
speed of, 350, 375–377
when to use, 822
binary searches, 48–51
duplicates, 51
Guess-a-Number game example, 48–49
logarithms in, 58–60
Ordered Array Visualization tool, 49–51
OrderedArray class
advantages of, 57–58
example of, 53–57
find() method, 52–53
recursion in, 242–244
speed of, 67
binary trees. *See also* binary search trees
advantages/disadvantages, 5
balanced/unbalanced
defined, 463
degenerates, 463–464
measuring balance, 464–469
defined, 337, 339–340
heaps as, 666–667, 684–685
Huffman trees
creating, 389–391
decoding with, 388–389
defined, 388
encoding with, 391–392
representing arithmetic expressions, 363–365
BinarySearchTree class, 344
deleting single child nodes, 369–370
finding nodes, 348–349
inserting nodes, 352–353
__Node class, 345–346
testing code, 382–385
traversal with, 356–361
black height
in color swaps, 499–500
defined, 488
blocks per node in B-trees, 444–445
bottom-up insertion, 486–487
bounding boxes of query circles, 603
Bounds class, 605–606
bounds completely within other bounds, 610–611
Cartesian coordinates, 603–604
CircleBounds subclass, 607–609
geographic coordinates, 604–605
intersection of bounding boxes, 609–610
intersection with grid cells, 628–629
within layers, 625–628
Bounds class, 605–606
branches, defined, 338
breadth-first traversal
example of, 727–731
Graph class, 731–733
Graph Visualization tool, 731
B-trees
blocks per node, 444–445
defined, 444
inserting nodes, 446–449
searching, 445–446
speed of, 449–450
when to use, 830
bubble sorts, 77–82
in Array class, 81–82
comparison of sorting methods, 96–97
described, 77–79
invariants, 82
Simple Sorting Visualization tool, 79–81
speed of, 82
buckets, defined, 569
buffers, defined, 442
bytecode, defined, 8

C

Cartesian coordinates
  bounding boxes of query circles, 603–604
  defined, 597–598
  distance between, 599
cells, defined, 38
character codes, 386–388
children of tree nodes
  defined, 338
  double child nodes, deleting, 370–375
  null children in red-black trees, 495–496
  single child nodes, deleting, 367–370
CircleBounds subclass, 607–609
circles. *See* query circles
circular lists, 209–210
circular queues, defined, 118
class attributes, defined, 25
classes, defined, 23
close matches. *See* nearest matches
clustering in hash tables
  with HashTableOpenAddressing Visualization tool, 540–543
  primary and secondary clustering, 558–559
collisions
  defined, 533
  hashing and, 533–536
combinations, recursion and, 278–280
comments in Python, described, 12
complex numbers, defined, 26
compressing data. *See* Huffman code
computational complexity, defined, 68
concatenating sequences, 15
conditional expressions, defined, 20
connected graphs, defined, 708
connectivity in directed graphs, 751
  connectivity matrix, 753
  transitive closure, 751–756
constants in Big O notation, 67–68
counting sort, 323–324
crossover subtrees
  defined, 478
  in rotations, 478–479
cutoff points, defined, 315
cycles
  in directed graphs, 743–744
  Hamiltonian, 800–802
  Warshall's algorithm and, 755–758

D

data compression. *See* Huffman code
data organizations
  defined, 1
  recipe analogy, 1–3
data structures. *See also* types of specific data structures
  altering during iteration, 216–217
  choosing what to use
    foundational data structures, 818–824
    problem analysis, 814–818
    special-ordering data structures, 824–826
    specialty data structures, 828–829
  databases vs.7
  defined, 1
  list of, 4–5
  operations on, 4
  purpose of, 3–4
data types
  ADTs (abstract data types)
    defined, 184, 190–191
    as design tool, 191–192
    as interface, 191
  described, 189–190

dynamic typing, 12–13
reference types, 160–163
sequences in Python, 13–15
databases
data structures vs.7
defined, 6
datasets distributed in cloud, defined, 438
decoding with Huffman trees, 388–389
degenerate trees, defined, 463–464
deletion. *See also* removal
in arrays
Array class, 42
Array Visualization tool, 34–35, 37
defined, 4
in doubly linked lists
at ends, 201–204
in middle, 204–208
with duplicates, 36
in grids, 623
in hash tables
with HashTable class, 552–553
with HashTableChaining Visualization tool, 568
with HashTableOpenAddressing Visualization tool, 542
for separate chaining, 574
in linked lists, 166–167, 174–177
of nodes
in 2–3-4 trees, 423–430
in AVL trees, 479–484
double child nodes, 370–375
leaf nodes in binary trees, 367
process of, 366–367
in red-black trees, 491, 508
single child nodes, 367–370
in point lists, 614–615
in quadtrees, 646–647
delimiter matching example for stacks, 113–116
dependency relationships example (topological sorting), 739
depth-first traversal
example of, 720–722
game simulation, 727
Graph class, 724–727
Graph Visualization tool, 722–723
maze analogy, 722
deques
defined, 125–126
doubly linked lists for, 208
descendants, defined, 339
dictionary example (hashing), 527–530
Dijkstra's algorithm
implementation, 791–792
rail travel example, 782–788
WeightedGraph Visualization tool, 788–791
directed graphs
connectivity in, 751
connectivity matrix, 753
transitive closure, 751–756
cycles in, 743–744
defined, 708–709
described, 739–740
in Graph Visualization tool, 741–742
topological sorting in, 742–743
distance between points
Cartesian coordinates, 599
geographic coordinates, 599–601
query circles, 601–603
divide-and-conquer algorithms
defined, 245
mergesort as, 257–260
double child nodes, deleting, 370–375
double hashing, 559–565
example of, 562–564
HashTableOpenAddressing Visualization tool, 561–562

implementation, 559–561
speed of, 583
table size, 564–565
double-ended lists, 177–183
doubly linked lists, 198–201
for deques, 208
insertion/deletion at ends, 201–204
insertion/deletion in middle, 204–208
duplicate keys in binary search trees, 381–382
duplicates
in arrays
Array Visualization tool, 35–37
Ordered Array Visualization tool, 51
in hash tables
with HashTableChaining Visualization tool, 568
with HashTableOpenAddressing Visualization tool, 542
dynamic typing, described, 12–13

E

edges
adding to graphs, 713–716
defined, 336, 706
modeling, 710
storing
in adjacency lists, 714–716
in adjacency matrices, 710–712
for weighted graphs, 774–776
efficiency. *See* speed
elements (of lists)
accessing, 38–39
defined, 38
eliminating recursion, 267
in mergesorts, 270–275
in quicksorts, 318
with stacks, 267–270
encapsulation, defined, 42–43
encoding
defined, 532
with Huffman trees, 391–392
enumerating sequences, 15–17
error handling in stacks, 111–112
errors. *See* exceptions
Euclidean distance, defined, 599
Euler, Leonhard, 709
evaluating postfix expressions, 148–151
exact matches
in grids, 621–623
in point lists, 614
in quadtrees, 644–645
exceptions
described, 22–23
finishing iteration, 213–215
external storage
accessing, 439–442
B-trees
blocks per node, 444–445
defined, 444
inserting nodes, 446–449
searching, 445–446
speed of, 449–450
choosing what to use, 829–831
defined, 438
file indexes
complex search criteria, 452–453
defined, 450–451
hashing and, 588–590
inserting in, 451–452
in memory, 450, 452
multiple, 452
searching, 451
sequential ordering, 442–443
sorting with mergesort, 453–456
extreme values, finding

in binary search trees, 365–366
in heaps, 695–700

F

factorials, described, 237–238
fencepost loops, defined, 145
Fibonacci sequence, 218–222
fields, defined, 7
file indexes
complex search criteria, 452–453
defined, 450–451
hashing and, 588–590
inserting in, 451–452
in memory, 450, 452
multiple, 452
searching, 451
when to use, 829–830
files, defined, 438
filled sequences in hash tables, 540–542
find() method, binary searches with, 52–53
finding. *See also* searching
extreme values in heaps, 695–700
minimum/maximum key values, 365–366
nodes
with Binary Search Tree Visualization tool, 346–348
with BinarySearchTree class, 348–349
successors, 372–373
finishing iteration
exception handling, 213–215
markers/sentinel values, 213
termination tests, 213
Floyd, Robert, 798
Floyd-Warshall algorithm, 798
folding, defined, 580–581
folds, defined, 531
foundational data structures, when to use, 818–824
functions in Python, described, 19–20
fusion
applying on descent, 429–430
defined, 427
extending, 428–429

G

game simulation with depth-first traversal, 727
gap sequences
defined, 288–289
selecting, 293
general-purpose data structures, when to use, 818–824
generators
for 2–3-4 tree traversal, 421–423
for adjacent vertex traversal, 724–727
described, 218–222
doubleHashProbe(), 559–565
for graph traversal, 731–733
for grid offsets, 629–630
for grid traversal, 623–624
for hash table traversal, 553–554
for heap traversal, 682–683
linearProbe(), 552
for point traversal, 615
quadraticProbe(), 554–559
for quadtree traversal, 645–646
in Timsorts, 326
for tree traversal, 356–361
geographic coordinates
bounding boxes of query circles, 604–605
defined, 598
distance between, 599–601

Graph class, 715–718
breadth-first traversal, 731–733
depth-first traversal, 724–727
minimum spanning trees in, 735–739
topological sorting in, 746–747
Graph Visualization tool
breadth-first traversal, 731
depth-first traversal, 722–723
directed graphs in, 741–742
minimum spanning trees in, 733
topological sorting algorithm, 742
graphs. *See also* topological sorting; weighted graphs
adding vertices/edges, 713–716
advantages/disadvantages, 5
defined, 337
directed graphs
connectivity in, 751–756
cycles in, 743–744
described, 739–740
in Graph Visualization tool, 741–742
topological sorting in, 742–743
Graph class, 715–718
history of, 709
intractable problems
defined, 798
Hamiltonian paths/cycles, 800–802
Knight's Tour, 798
Traveling Salesperson, 799–800
minimum spanning trees
described, 733
in Graph class, 735–739
in Graph Visualization tool, 733
as subgraphs, 733–737
modeling
adjacency list, 712–713
adjacency matrix, 710–712
edges, 710
vertices, 709–710
purpose of, 706
speed of algorithms, 798
terminology, 707–709
traversal, 718–719
breadth-first, 727–733
depth-first, 719–727
when to use, 828–829
great circle, defined, 599–600
Grid class, 619–620
grids, 617
deleting points, 623
exact matches, 621–623
Grid class instances, 619–620
implementation in Python, 618–619
inserting points, 620–621
intersection with query circles, 628–629
nearest matches, 624–625, 630–633
neighboring cell sequence, 629–630
query circles within layers, 625–628
traversing points, 623–624
when to use, 828
growing hash tables
with HashTable class, 550–551
speed of, 585–587
Guess-a-Number game example, 48–49

H

Hamiltonian paths/cycles intractable problem, 800–802
hash addresses, defined, 531
hash functions
computation speed, 575
defined, 526, 531
folding, 580–581

nonrandom keys, 576–578
random keys, 575–576
simpleHash() function, 545–546
for strings, 578–580
hash tables
adjacency matrix as, 712
advantages/disadvantages, 5, 525
defined, 525, 531
for external storage, 588–590
HashTable class, 544–545
deleting data, 552–553
growing hash tables, 550–551
inserting data, 548–549
rehashing data, 551
searching data, 546–548
simpleHash() function, 545–546
traversal, 553–554
HashTableChaining Visualization tool, 566–569
buckets, 569
deleting data, 568
duplicates, 568
load factors, 568
table size, 569
HashTableOpenAddressing Visualization tool, 536–543
clustering, 540–543
deleting data, 542
double hashing, 561–562
duplicates, 542
inserting data, 537–540
quadratic probing, 555–558
searching data, 540
traversal, 554
when to use, 823
hashingf
collisions and, 533–536
external storage and, 588–590
keys
dictionary example, 527–530
nonrandom keys, 576–578
numbers as, 526–527
random keys, 575–576
open addressing
double hashing, 559–565
HashTable class, 544–554
linear probing, 536–543
quadratic probing, 554–559
separate chaining vs.587–588
process of, 530–533
separate chaining
defined, 565
HashTable class, 569–574
HashTableChaining Visualization tool, 566–569
KeyValueList class, 571–572
open addressing vs.587–588
types to use, 574–575
simpleHash() function, 545–546
speed of, 581
open addressing, 581–583
separate chaining, 583–587
strings, 578–580
when to use, 830
HashTable class
open addressing, 544–545
deleting data, 552–553
growing hash tables, 550–551
inserting data, 548–549
linearProbe() generator, 552
rehashing data, 551
searching data, 546–548
traversal, 553–554
separate chaining, 569–574
HashTableChaining Visualization tool, 566–569
buckets, 569

deleting data, 568
duplicates, 568
load factors, 568
table size, 569
HashTableOpenAddressing Visualization tool, 536–543
clustering, 540–543
deleting data, 542
double hashing, 561–562
duplicates, 542
inserting data, 537–540
quadratic probing, 555–558
searching data, 540
traversal, 554
haversine formula, defined, 600
Heap class, 677–683
Heap Visualization tool, 674–677
heapify() subroutine, 691–693
heaps
advantages/disadvantages, 5
as binary trees, 666–667, 684–685
changing priority, 674
defined, 104, 666
finding extreme values, 695–700
Heap class, 677–683
Heap Visualization tool, 674–677
erasing and randomly filling, 676
insertion in, 669–670
with Heap class, 679–680
with Heap Visualization tool, 675
for order statistics, 694–695
as partially ordered, 668
peeking, 674
with Heap Visualization tool, 677
priority queues and, 667–668
purpose of, 665
removal in, 670–674
with Heap class, 680–682
replacing maximum, 674
with Heap Visualization tool, 677
sifting up/down, 670–674
sorting. *See* heapsort
speed of, 683–684
traversal
with Heap class, 682–683
with Heap Visualization tool, 677
heapsort
heapify() subroutine, 691–693
heapsort() subroutine, 691–693
process of, 686
reusing array for, 688–691
sifting up/down, 686–688
speed of, 693
when to use, 827
heapsort() subroutine, 691–693
height of subtrees, defined, 467
hierarchical file system analogy, 340–341
holes, defined, 34–35
Huffman, David, 386
Huffman code
character codes, 386–388
defined, 386
Huffman trees
creating, 389–391
decoding with, 388–389
defined, 388
encoding with, 391–392

I

importing Python modules, 18–19
indentation in Python, described, 9–12
indexes. *See also* file indexes

hash tables as, 588
when to use, 829–830
induction, defined, 237
infix notation
comparison with postfix notation, 133
defined, 133, 363
InfixCalculator tool, 142–148
translating to postfix, 134–142
InfixCalculator tool, 142–148
inheritance, defined, 23
initialization of arrays, 39
in-order successors
defined, 371
finding, 372–373
replacing nodes with, 373–374
in-order traversal, 353–355
insertion
in 2–3 trees, 437–438
in 2–3-4 trees, 404–405
with Tree234 Visualization tool, 409–410
in arrays
Array class, 42
Array Visualization tool, 33
speed of, 66
bottom-up in trees, 486–487
defined, 4
in doubly linked lists
at ends, 201–204
in middle, 204–208
with duplicates, 36
in file indexes, 451–452
in grids, 620–621
in hash tables
with HashTable class, 548–549
with HashTableOpenAddressing Visualization tool, 537–540
speed of, 584
in heaps, 669–670
with Heap class, 679–680
with Heap Visualization tool, 675
in linked lists, 170–174
of nodes
with AVLtree class, 474–478
with AVLTree Visualization tool, 470–472
with Binary Search Tree Visualization tool, 351–352
with BinarySearchTree class, 352–353
in B-trees, 446–449
multiple red nodes, 492
process of, 350
in red-black trees, 492, 498–499
in point lists, 613–614
in priority queues, 127–128
in quadtrees, 636–638, 641–644
in queues, 117, 119
in sequentially ordered files, 443
in stacks. *See* push (stacks)
top-down in trees, 486
insertion sorts, 87–91
in Array class, 90
comparison of sorting methods, 96–97
described, 87–89
disadvantages, 286
invariants, 91
list insertion sorts, 198
Simple Sorting Visualization tool, 89–90
within small partitions, 315
speed of, 91
when to use, 827
instance attributes, defined, 25
instances, defined, 23
integer index, defined, 38

internal nodes
defined, 339
deleting in 2–3-4 trees, 424–425
promoting splits to, 435–437
interpreter (Python), described, 8–12
intersection
of bounding boxes, 609–610
of grid cells, 628–629
interval sequences
defined, 288–289
selecting, 293
intractable problems
defined, 798
Hamiltonian paths/cycles, 800–802
Knight's Tour, 798
Traveling Salesperson, 799–800
invariants
in bubble sorts, 82
defined, 82
in insertion sorts, 91
in selection sorts, 86
isomorphic, defined, 508
items, defined, 38
iteration
altering data structures during, 216–217
described, 15–17
finishing
exception handling, 213–215
markers/sentinel values, 213
termination tests, 213
iterators
described, 211–212
methods in, 212–217
in Python, 217–222

K

K highest
finding extreme values, 695–696
speed of, 696–700
k-d trees, defined, 659
key values
defined, 346
finding minimum/maximum in trees, 365–366
keys
defined, 7, 61, 339
duplicates in binary search trees, 381–382
for hashing
dictionary example, 527–530
nonrandom keys, 576–578
numbers as, 526–527
random keys, 575–576
secondary sort keys, 96
KeyValueList class, 571–572
knapsack problem, 277–278
Knight's Tour intractable problem, 798
Königsberg bridge problem, 709

L

latitude, defined, 598
layers, query circles within, 625–628
leaves
defined, 339
deleting
in 2–3-4 trees, 424
in binary search trees, 367

left child, defined, 339–340
left heavy, defined, 476
levels (of nodes)
  defined, 339
  in trees as arrays, 378–379
libraries, when to use, 820
linear probing, 536–543
  defined, 536
  HashTable class, 552
  HashTableOpenAddressing Visualization tool, 536–543
    clustering, 540–543
    deleting data, 542
    duplicates, 542
    inserting data, 537–540
    searching data, 540
    traversal, 554
  speed of, 582–583
linear searches
  defined, 48
  speed of, 66–67
Link class, 159
  deletion, 174–177
  insertion, 170–174
  methods in, 167–169
  searches, 170–174
  traversal, 169–170
linked lists
  adjacency list, modeling, 712–713
  advantages/disadvantages, 5, 336, 821
  arrays vs.164
  circular lists, 209–210
  double-ended lists, 177–183
  doubly linked lists, 198–201
    for deques, 208
    insertion/deletion at ends, 201–204
    insertion/deletion in middle, 204–208
  Link and LinkedList classes
    deletion, 174–177
    insertion, 170–174
    methods in, 167–169
    searches, 170–174
    traversal, 169–170
  LinkedList Visualization tool, 164–167
  links in, 158–160
  ordered lists
    described, 192
    list insertion sort with, 198
    OrderedList class, 193–198
    OrderedList Visualization class, 192–193
    speed of, 198
  queue implementation by, 187–189
  reference types and, 160–163
  speed of, 183–184
  stack implementation by, 184–187
linked trees, nodes in, 378–379
LinkedList class, 159
  deletion, 174–177
  insertion, 170–174
  methods in, 167–169
  searches, 170–174
  traversal, 169–170
LinkedList Visualization tool, 164–167
links in linked lists, 158–160
list comprehensions, described, 20–22
list of points, 612
  deleting points, 614–615
  exact matches, 614
  inserting points, 613–614
  nearest matches, 615–616
  PointList class, 612
  traversing points, 615

lists (data type in Python). *See also* ADT lists; linked lists
as arrays, 37
accessing elements, 38–39
creating, 37–38
deletion, 42
initialization, 39
insertion, 42
searches, 42
traversal, 42
as sequences, 13–15
slicing, 39
as stacks, 108–112
delimiter matching example, 113–116
error handling, 111–112
word reversal example, 112–113
load factors
defined, 548
in separate chaining, 568
local variables, defined, 25
logarithms in binary searches, 58–60
logical lines in Python, defined, 10
longitude, defined, 598
looping. *See also* iterators
described, 15–17
list comprehensions, 20–22

M

mapping, defined, 21
markers, finishing iteration, 213
matching delimiters example for stacks, 113–116
mathematical induction, defined, 237
maximum, replacing in heaps, 674, 677
maze analogy (depth-first traversal), 722
measuring tree balance, 464–469
median-of-three partitioning, 313–315
mergesort
advantages/disadvantages, 255, 827
as divide-and-conquer algorithm, 257–260
eliminating recursion in, 270–275
for external files, 453–456
Mergesort Visualization tool, 263–264
with sorted arrays, 255–257
speed of, 264–267
with subranges, 260–262
testing code, 262–263
Mergesort Visualization tool, 263–264
methods, defined, 23
minimum spanning trees
described, 733
in Graph class, 735–739
in Graph Visualization tool, 733
as subgraphs, 733–737
with weighted graphs
building, 770–774
creating algorithm, 774–780
networking example, 768
WeightedGraph class, 776–779
WeightedGraph Visualization tool, 768–770
modeling graphs
adjacency list, 712–713
adjacency matrix, 710–712
edges, 710
vertices, 709–710
modules, importing, 18–19
multiple file indexes, 452
multiple red nodes during insertion, 492
multiplying sequences, 15
multivalued assignment, described, 17–18
multiway trees, defined, 337. *See also* 2–3 trees; 2–3-4 trees; B-trees
mutual recursion, defined, 374

N

name mangling in Python, defined, 44
namespaces in Python, defined, 19
nearest matches
  in grids, 624–625, 630–633
  in point lists, 615–616
  in quadtrees, 647–655
neighbors, defined, 707
networking example (minimum spanning trees), 768
  algorithm, 774–780
  building minimum spanning tree, 770–774
__Node class
  AVLtree class, 472–474
  BinarySearchTree class, 345–346
  Tree234 class, 412–415
Node class (quadtrees), 640–641
nodes
  blocks per node in B-trees, 444–445
  defined, 336
  deleting
    in 2–3-4 trees, 423–430
    in AVL trees, 479–484
    double child nodes, 370–375
    leaf nodes, 367
    process of, 366–367
    in red-black trees, 491, 508
    single child nodes, 367–370
  finding
    with Binary Search Tree Visualization tool, 346–348
    with BinarySearchTree class, 348–349
  flipping colors, 489–490, 493–494, 499–500
  inserting
    with AVLtree class, 474–478
    with AVLTree Visualization tool, 470–472
    with Binary Search Tree Visualization tool, 351–352
    with BinarySearchTree class, 352–353
    in B-trees, 446–449
    process of, 350
    in red-black trees, 492–491, 498–499
  in linked trees, 378–379
  replacing with successors, 373–374
  rotating in red-black trees, 490–491, 492–493, 500–507
  splitting
    in 2–3 trees, 433–434, 437–438
    in 2–3-4 trees, 405–406, 407–408
    color swaps and, 512
    promoting to internal nodes, 435–437
    rotations and, 512–514
    with Tree234 class, 418–421
nonrandom keys for hashing, 576–578
nonvolatile data storage, defined, 439
null children in red-black trees, 495–496
numbers
  converting words to, 527–530
  as hash keys, 526–527
  raising to powers, 275–276

O

object-oriented programming, described, 23–26
objects
  defined, 23
  storing, 60–65
octrees, defined, 659
open addressing
  defined, 535
  double hashing, 559–565
    example of, 562–564

HashTableOpenAddressing Visualization tool, 561–562
implementation, 559–561
table size, 564–565
HashTable class, 544–545
deleting data, 552–553
growing hash tables, 550–551
inserting data, 548–549
linearProbe() generator, 552
rehashing data, 551
searching data, 546–548
traversal, 553–554
linear probing, 536–543
defined, 536
HashTableOpenAddressing Visualization tool, 536–543, 554
quadratic probing, 554–559
separate chaining vs.587–588
speed of, 581–583
operands, defined, 133, 364
operators
defined, 133
precedence, 135
saving on stacks, 139–140
order of the function, defined, 68
order statistics, heaps for, 694–695
ordered arrays
advantages/disadvantages, 5, 57–58, 335–336, 826
defined, 47
Guess-a-Number game example, 48–49
OrderedArray class
advantages of, 57–58
example of, 53–57
find() method, 52–53
OrderedArray Visualization tool, 47–51
binary searches, 49–51
duplicates, 51
OrderedRecordArray class, 61–65
ordered lists
described, 192
list insertion sort with, 198
OrderedList class, 193–198
OrderedList Visualization class, 192–193
speed of, 198
OrderedArray class
advantages of, 57–58
example of, 53–57
find() method, 52–53
OrderedArray Visualization tool, 47–51
binary searches, 49–51
duplicates, 51
Guess-a-Number game example, 48–49
OrderedList class, 193–198
OrderedList Visualization class, 192–193
OrderedRecordArray class, 61–65
orders of magnitude, defined, 815

P

parameters, defined, 20
parent nodes, defined, 338
parsing arithmetic expressions
described, 132–133
evaluating postfix expressions, 148–151
Infix Calculator tool, 142–148
postfix notation, 133–134
translating infix to postfix, 134–142
traversal order and, 363–365
partial sorting
defined, 87
finding extreme values, 695–700
heaps and, 668

partitioning
with AdvancedSorting Visualization tool, 295–297
algorithm for, 297–301
defined, 294–295
in quicksort algorithm, 302–304
detailed explanation, 310–313
eliminating recursion in, 318
full implementation, 315–318
initial implementation, 306–309
median-of-three partitioning, 313–315
small partitions, 315
speed of, 318–320
speed of, 301–302
paths
defined, 338, 707–708
Hamiltonian, 800–802
peeking
in heaps, 674
with Heap Visualization tool, 677
in priority queues, 128
in queues, 120
in stacks
defined, 106
in Stack Visualization tool, 108
perfect hash functions, defined, 575–576
permutations, defined, 239
Peters, Tim, 325
pivot values
defined, 295–296
equal keys, 300–301
median-of-three, 313–315
selecting, 304–306
PointList class, 612
points
distance between
Cartesian coordinates, 599
geographic coordinates, 599–601
query circles, 601–603
grids, 617
deleting points, 623
exact matches, 621–623
Grid class instances, 619–620
implementation in Python, 618–619
inserting points, 620–621
intersection with query circles, 628–629
nearest matches, 624–625, 630–633
neighboring cell sequence, 629–630
query circles within layers, 625–628
traversing points, 623–624
lists, 612
deleting points, 614–615
exact matches, 614
inserting points, 613–614
nearest matches, 615–616
PointList class, 612
traversing points, 615
quadtrees
ambiguity in, 638–639
deleting points, 646–647
described, 633–635
exact matches, 644–645
inserting points, 636–638, 641–644
nearest matches, 647–655
Node class, 640–641
QuadTree class, 635–636
QuadTree Visualization tool, 639–640
traversing points, 645–646
pop (stacks)
defined, 104
in Stack Visualization tool, 108
postfix notation
comparison with infix notation, 133
defined, 133, 364

described, 133–134
evaluating expressions, 148–151
Infix Calculator tool, 142–148
translating infix to, 134–142
post-order traversal, 355
powers, raising numbers to, 275–276
precedence (of operators), defined, 135
prefix notation, defined, 134, 364
pre-order traversal, 355
primary clustering, defined, 558
printing binary search trees, 379–381
priority, changing in heaps, 674
priority queues
defined, 106
described, 126–127
heaps and, 667–668
PriorityQueue class, 129–132
PriorityQueue Visualization tool, 127–129
search and traversal, 132
speed of, 132
use case comparison with arrays, 103–104
when to use, 826
PriorityQueue class, 129–132
PriorityQueue Visualization tool, 127–129
probing, defined, 541
problem analysis, 814–818
amount of data, 815–816
frequency of operations, 816–817
software maintenance responsibilities, 817–818
types of data, 814–815
pseudocode, defined, 140
push (stacks)
defined, 104
in Stack Visualization tool, 107
Python
comments, 12
dynamic typing, 12–13
exceptions, 22–23
functions/subroutines, 19–20
as interpreted language, 8–12
iteration, 15–17
list comprehensions, 20–22
modules, importing, 18–19
multivalued assignment, 17–18
as object-oriented programming, 23–26
sequences, 13–15
whitespace, 8, 9–12

Q

quadratic probing, 554–559
speed of, 583
QuadTree class, 635–636
QuadTree Visualization tool, 639–640
quadtrees
advantages/disadvantages, 5, 828
ambiguity in, 638–639
deleting points, 646–647
described, 633–635
exact matches, 644–645
inserting points, 636–638, 641–644
nearest matches, 647–655
Node class, 640–641
QuadTree class, 635–636
QuadTree Visualization tool, 639–640
traversing points, 645–646
query circles
bounding boxes of, 603
Bounds class, 605–606
bounds completely within other bounds, 610–611
Cartesian coordinates, 603–604
CircleBounds subclass, 607–609
geographic coordinates, 604–605

intersection of bounding boxes, 609–610
intersection with grid cells, 628–629
within layers, 625–628
distance between points, 601–603
Queue class, 120–125
Queue Visualization tool, 119–120
queues. *See also* priority queues
advantages/disadvantages, 5, 825
circular, 118
defined, 106
deques, 125–126
described, 116–117
linked list implementation of, 187–189
Queue class, 120–125
Queue Visualization tool, 119–120
search and traversal, 132
shifting, 117–118
speed of, 125
use case comparison with arrays, 103–104
quicksort
AdvancedSorting Visualization tool, 309–310, 318
algorithm for, 302–304
defined, 302
detailed explanation, 310–313
eliminating recursion in, 318
full implementation, 315–318
initial implementation, 306–309
median-of-three partitioning, 313–315
pivot values, 304–306
small partitions, 315
speed of, 318–320

R

radix, defined, 321
radix sort, 320–323
defined, 321
designing, 321–322
generalizing, 322–323
speed of, 322
rail travel example (shortest-path problem), 780–782
all-pairs shortest-path problem, 796–798
Dijkstra's algorithm, 782–788
implementation of algorithm, 791–792
WeightedGraph class, 792–796
WeightedGraph Visualization tool, 788–791
raising numbers to powers, 275–276
random heaps, 676
random keys for hashing, 575–576
records
defined, 6, 60–61
OrderedRecordArray class, 61–65
recursion
anagrams, 239–242
applications for
combinations, 278–280
knapsack problem, 277–278
raising numbers to powers, 275–276
binary searches, 242–244
characteristics of, 235–236
defined, 6, 229
divide-and-conquer algorithms, 245
eliminating, 267
in mergesorts, 270–275
with stacks, 267–270
factorials, 237–238
finding nth terms with, 232–235
mathematical induction and, 237
mergesort
advantages/disadvantages, 255
as divide-and-conquer algorithm, 257–260
eliminating recursion in, 270–275
Mergesort Visualization tool, 263–264

with sorted arrays, 255–257
speed of, 264–267
with subranges, 260–262
testing code, 262–263
in partitioning algorithm, 297–301
in quicksort algorithm, 302–304
detailed explanation, 310–313
eliminating recursion in, 318
full implementation, 315–318
initial implementation, 306–309
median-of-three partitioning, 313–315
small partitions, 315
speed of, 318–320
speed of, 236
Tower of Hanoi puzzle
described, 245–246
implementation of solution, 250–255
solution to, 247–249
TowerOfHanoi Visualization tool, 246–247
recursive depth, defined, 255
red-black correct, defined, 487
red-black rules
balanced trees and, 495
described, 487–488
fixing violations, 488
null children, 495–496
rotations and, 496–497
red-black trees, 485
2–3-4 trees and, 508–510
operational equivalence, 510–514
transformation between, 509–510
advantages/disadvantages, 5
bottom-up insertion, 486–487
characteristics of, 487
implementation, 513–515
red-black rules
balanced trees and, 495
described, 487–488
fixing violations, 488
null children, 495–496
rotations and, 496–497
RedBlackTree Visualization tool, 488–489
deleting nodes, 491, 508
erasing and randomly filling, 491
flipping node colors, 489–490, 493–494, 499–500
inserting nodes, 492–491, 498–499
multiple red nodes insertion experiment, 492
rotating nodes, 490–491, 492–493, 500–507
searching, 491
unbalanced tree experiment, 494–496
speed of, 508
top-down insertion, 486
RedBlackTree Visualization tool, 488–489
deleting nodes, 491, 508
erasing and randomly filling, 491
flipping node colors, 489–490, 493–494, 499–500
inserting nodes, 492–491, 498–499
multiple red nodes during insertion experiment, 492
rotating nodes, 490–491, 492–493, 500–507
searching, 491
unbalanced tree experiment, 494–496
reference types, 160–163
rehashing hash tables with HashTable class, 551
removal. *See also* deletion
in heaps, 670–674
with Heap class, 680–682
in priority queues, 128
in queues, 117, 119–120
in stacks. *See* pop (stacks)
reversing words example for stacks, 112–113
right child, defined, 339–340
right heavy, defined, 476

ring buffers, defined, 118
root (of trees)
  defined, 337, 338
  splitting, 406–407
rotation of tree nodes
  applying on descent, 429–430
  crossover subtrees in, 478–479
  defined, 426–427
  node splits and, 512–514
  red-black rules and, 496–497
  in red-black trees, 490–491, 492–493, 500–507

S

saving operators on stacks, 139–140
scrolling in Tree234 Visualization Tool, 410–411
search
  binary searches. *See* binary searches
  defined, 4
  linear searches, 48, 66–67
search keys. *See* keys
searching. *See also* finding
  2–3-4 trees, 404
    with Tree234 class, 415–417
    with Tree234 Visualization tool, 409
  arrays
    Array class, 42
    Array Visualization tool, 31–33
  B-trees, 445–446
  with duplicates, 35–36
  file indexes, 451, 452–453
  hash tables
    with HashTable class, 546–548
    with HashTableOpenAddressing Visualization tool, 540
    speed of, 584
  linked lists, 166, 170–174
  red-black trees, 491
  sequences, 15
  sequentially ordered files, 442–443
  spatial data, 611–612
    grids, 617–633
    list of points, 612–616
    performance optimization, 656–658
    quadtrees, 633–655
  stacks and queues, 132
secondary clustering, defined, 558
secondary sort keys, defined, 96
selection sorts, 83–87
  in Array class, 85–86
  comparison of sorting methods, 96–97
  described, 83–85
  invariants, 86
  Simple Sorting Visualization tool, 85
  speed of, 86–87
sentinel values
  finishing iteration, 213
  median-of-three partitioning, 314
separate chaining
  with buckets, 569
  defined, 535, 565
  HashTable class, 569–574
  HashTableChaining Visualization tool, 566–569
    deleting data, 568
    duplicates, 568
    load factors, 568
    table size, 569
  KeyValueList class, 571–572
  open addressing vs.587–588
  speed of, 583–587
  types to use, 574–575
sequences
  described, 13–15
  enumerating, 15–17
  multivalued assignment, 17–18

sequential ordering, 442–443
sequential storage, 829
Shell, Donald L.285
Shellsort
  AdvancedSorting Visualization tool, 289–291
  advantages/disadvantages, 285–286
  described, 286–288
  insertion sort disadvantages, 286
  interval sequences, 288–289, 293
  ShellSort class, 291–293
  speed of, 294
  when to use, 827
ShellSort class, 291–293
shifting queues, 117–118
shortest-path problem
  all-pairs shortest-path problem, 796–798
  Dijkstra's algorithm, 782–788
  implementation, 791–792
  rail travel example, 780–782
  WeightedGraph class, 792–796
  WeightedGraph Visualization tool, 788–791
siblings, defined, 339
sifting up/down
  in heaps, 670–674
  in heapsort, 686–688
Simple Sorting Visualization tool
  bubble sorts, 79–81
  insertion sorts, 89–90
  selection sorts, 85
simpleHash() function, 545–546
single child nodes, deleting, 367–370
slicing
  lists in Python, 39
  sequences, 14
software bloat, defined, 819
sort keys. *See* keys
SortArray class, 91–96
sorting
  with counting sort, 323–324
  defined, 6
  with heapsort
    heapify() subroutine, 691–693
    heapsort() subroutine, 691–693
    process of, 686
    reusing array for, 688–691
    sifting up/down, 686–688
    speed of, 693
  with mergesort
    advantages/disadvantages, 255
    as divide-and-conquer algorithm, 257–260
    eliminating recursion in, 270–275
    for external files, 453–456
    Mergesort Visualization tool, 263–264
    with sorted arrays, 255–257
    speed of, 264–267
    with subranges, 260–262
    testing code, 262–263
  partitioning. *See* partitioning
  with quicksort
    AdvancedSorting Visualization tool, 309–310, 318
    algorithm for, 302–304
    defined, 302
    detailed explanation, 310–313
    eliminating recursion in, 318
    full implementation, 315–318
    initial implementation, 306–309
    median-of-three partitioning, 313–315
    pivot values, 304–306
    small partitions, 315
    speed of, 318–320
  with radix sort, 320–323
  with Shellsort

AdvancedSorting Visualization tool, 289–291
advantages/disadvantages, 285–286
described, 286–288
insertion sort disadvantages, 286
interval sequences, 288–289, 293
ShellSort class, 291–293
speed of, 294
simple sorting algorithms
bubble sorts, 77–82
comparison of, 96–97
insertion sorts, 87–91
list insertion sorts, 198
selection sorts, 83–87
SortArray class, 91–96
stability of, 96
with Timsort, 324–327
topological sorting
algorithm for, 742
dependency relationships example, 739
in directed graphs, 742–743
in Graph class, 746–747
improving, 747–751
speed of, 751
when to use, 826–828
spatial data
Cartesian coordinates
bounding boxes of query circles, 603–604
defined, 597–598
distance between points, 599
defined, 597
geographic coordinates
bounding boxes of query circles, 604–605
defined, 598
distance between points, 599–601
higher dimensions for, 658–659
operations on, 658
query circles
bounding boxes of, 603
Bounds class, 605–606
bounds completely within other bounds, 610–611
CircleBounds subclass, 607–609
distance between points, 601–603
intersection of bounding boxes, 609–610
within layers, 625–628
searching, 611–612
grids, 617–633
list of points, 612–616
performance optimization, 656–658
quadtrees, 633–655
special-ordering data structures, when to use, 824–826, 828–829
speed
2–3 trees, 438
2–3-4 trees, 431
algorithms
Big O notation, 65–68
general-purpose data structures, 819–820, 824
sorting algorithms, 828
special-ordering data structures, 826
AVL trees, 484–485
binary search trees, 350, 375–377
B-trees, 449–450
bubble sorts, 82
counting sort, 324
graph algorithms, 798
hash function computation, 575
hashing, 581
open addressing, 581–583
separate chaining, 583–587
heaps, 683–684
heapsort, 693
insertion sorts, 91

K highest, 696–700
linked lists, 183–184
mergesort, 264–267, 456
ordered lists, 198
partitioning, 301–302
priority queues, 132
quadtrees
exact matches, 645
insertion, 644
nearest matches, 655
queues, 125
quicksorts, 318–320
radix sort, 322
recursion, 236
red-black trees, 508
selection sorts, 86–87
Shellsorts, 294
spatial data searches, 656–658
stacks, 116
Timsort, 327
topological sorting, 751
splitting
nodes
in 2–3 trees, 433–434, 437–438
in 2–3-4 trees, 405–406, 407–408
color swaps and, 512
promoting to internal nodes, 435–437
rotations and, 512–514
with Tree234 class, 418–421
root (of trees), 406–407
stability of sorting algorithms, 96
Stack class, 108–112
delimiter matching example, 113–116
error handling, 111–112
word reversal example, 112–113
Stack Visualization tool, 106–108
New button, 107–108
Peek button, 108
Pop button, 108
Push button, 107
size of stack, 108
stacks
advantages/disadvantages, 5, 825
defined, 104–105
eliminating recursion with, 267–270
linked list implementation of, 184–187
postal analogy, 105–106
saving operators on, 139–140
search and traversal, 132
speed of, 116
Stack class, 108–112
delimiter matching example, 113–116
error handling, 111–112
word reversal example, 112–113
Stack Visualization tool, 106–108
New button, 107–108
Peek button, 108
Pop button, 108
Push button, 107
size of stack, 108
use case comparison with arrays, 103–104
storage requirements. *See also* external storage
2–3 trees, 438
2–3-4 trees, 432
AVL trees, 484–485
binary search trees, 350, 375–377
B-trees, 449–450
bubble sorts, 82
counting sort, 324
graph algorithms, 798
hash tables, 581
open addressing, 581–583
separate chaining, 583–587
heaps, 683–684
heapsort, 693
insertion sorts, 91

K highest, 696–700
linked lists, 183–184
mergesort, 264–267, 456
ordered lists, 198
partitioning, 301–302
priority queues, 132
quadtrees
exact matches, 645
insertion, 644
nearest matches, 655
queues, 125
quicksorts, 318–320
radix sort, 322
recursion, 236
red-black trees, 508
selection sorts, 86–87
Shellsorts, 294
spatial data searches, 656–658
stacks, 116
Timsort, 327
topological sorting, 751
storing
edges
in adjacency lists, 714–716
in adjacency matrices, 710–712
for weighted graphs, 774–776
objects, OrderedRecordArray class, 61–65
strings
hashing, 578–580
whitespace in, 12
subgraphs, minimum spanning trees as, 733–737
subheaps, 686–688
subranges, merging, 260–262
subroutines, described, 19–20
subtrees
defined, 339
height, 467
rotation, 496–497
successors
defined, 371
finding, 372–373
replacing nodes with, 373–374
swapping node colors, 489–490, 493–494, 499–500
node splits and, 512
symbol tables, defined, 527

T

termination tests, finishing iteration, 213
testing
arrays, 40–47
BinarySearchTree class, 382–385
double-ended lists, 180–183
doubly linked lists, 207–208
mergesort, 262–263
ordered arrays, 56–57, 61–65
ordered lists, 196–198
priority queues, 131–132
queues, 123–125, 188–189
Shellsorts, 292–293
simple sorting algorithms, 94–96
stacks, 110–112, 186–187
Timsort, 324–327, 827
tokens, defined, 145
top-down insertion, 486
topological sorting
algorithm for, 742
dependency relationships example, 739
of directed graphs, 742–743
in Graph class, 746–747
improving, 747–751
speed of, 751
Tower of Hanoi puzzle
described, 245–246
implementation of solution, 250–255

solution to, 247–249
TowerOfHanoi Visualization tool, 246–247
TowerOfHanoi Visualization tool, 246–247
transitive closure, 751–756
translating infix to postfix notation, 134–142
Traveling Salesperson intractable problem, 799–800
traversal
in arrays
Array class, 42
Array Visualization tool, 35
defined, 4
with duplicates, 36–37
of graphs, 718–719
breadth-first, 727–733
depth-first, 719–727
in grids, 623–624
in hash tables
with HashTable class, 553–554
with HashTableOpenAddressing Visualization tool, 554
order of, 573–574
in heaps
with Heap class, 682–683
with Heap Visualization tool, 677
iterators
described, 211–212
methods in, 212–217
in Python, 217–222
in linked lists, 169–170
in point lists, 615
in quadtrees, 645–646
in stacks and queues, 132
of trees
2–3-4 trees, 421–423
with Binary Search Tree Visualization tool, 361–363
with BinarySearchTree class, 356–361
defined, 339, 353
order of, 363–365
in-order traversal, 353–355
post-order traversal, 355
pre-order traversal, 355
Tree234 class, 415
deleting nodes, 423–430
__Node class, 412–415
node splits, 418–421
searching, 415–417
traversing, 421–423
Tree234 Visualization tool, 408–411
tree-based heaps, 684–685
trees. *See also* types of trees
advantages/disadvantages, 335–336
balanced/unbalanced
defined, 463
degenerates, 463–464
measuring balance, 464–469
cycles and, 743–744
defined, 336–337
terminology, 337–340
traversal of
2–3-4 trees, 421–423
with Binary Search Tree Visualization tool, 361–363
with BinarySearchTree class, 356–361
defined, 339, 353
order of, 363–365
in-order traversal, 353–355
post-order traversal, 355
pre-order traversal, 355
triangular numbers
defined, 230–231
eliminating recursion in, 268–270
finding nth term
with loops, 231–232
with recursion, 232–235

tuples, defined, 17, 715
two-dimensional arrays
  adjacency matrix as, 710–711
  in Python, 713–714

U

unbalanced trees
  defined, 343–344
  degenerates, 463–464
  example of, 494–496
  formation of, 463
  measuring balance, 464–469
unweighted graphs. *See* graphs
use cases for data structures, 103–104

V

vertices
  adding to graphs, 713–716
  defined, 706
  modeling, 709–710
virtual memory, 830–831
visiting, defined, 339, 353
visualization tools
  2–3-4 trees, 408–411
  advanced sorting
    partitioning, 295–297
    quicksorts, 309–310, 318
    Shellsorts, 289–291
  arrays, 30–37
    deletion, 34–35, 37
    duplicates, 35–37
    insertion, 33
    searches, 31–33
  speed of algorithms, 37
  traversal, 35
AVL trees, 470
  inserting nodes, 470–472
binary search trees, 341–344
  deleting double child nodes, 374
  deleting leaf nodes, 367
  deleting single child nodes, 368–369
  finding nodes, 346–348
  inserting nodes, 351–352
  traversal with, 361–363
graphs
  breadth-first traversal, 731
  depth-first traversal, 722–723
  directed graphs in, 741–742
  minimum spanning trees in, 733
  topological sorting algorithm, 742
hash tables
  double hashing, 561–562
  open addressing, 536–543, 554
  quadratic probing, 555–558
  separate chaining, 566–569
heaps, 674–677
linked lists, 164–167
mergesorts, 263–264
ordered arrays, 47–51
  binary searches, 49–51
  duplicates, 51
  Guess-a-Number game example, 48–49
ordered lists, 192–193
priority queues, 127–129
quadtrees, 639–640
queues, 119–120
red-black trees, 488–489
  deleting nodes, 491, 508
  erasing and randomly filling, 491
  flipping node colors, 489–490, 493–494, 499–500

inserting nodes, 492–491, 498–499
multiple red nodes insertion experiment, 492
rotating nodes, 490–491, 492–493, 500–507
searching, 491
unbalanced tree experiment, 494–496
simple sorting
bubble sorts, 79–81
insertion sorts, 89–90
selection sorts, 85
stacks, 106–108
New button, 107–108
Peek button, 108
Pop button, 108
Push button, 107
size of stack, 108
Tower of Hanoi puzzle, 246–247
weighted graphs
minimum spanning trees, 768–770
shortest-path problem, 788–791

W

Warshall, Stephen, 752, 798
Warshall's algorithm
implementation, 756
transitive closure and, 751–758
weighted graphs
all-pairs shortest-path problem, 796–798
defined, 708–709
minimum spanning trees with
building, 770–774
creating algorithm, 774–780
networking example, 768
WeightedGraph class, 776–779
WeightedGraph Visualization tool, 768–770
shortest-path problem
Dijkstra's algorithm, 782–788
implementation, 791–792
rail travel example, 780–782
WeightedGraph class, 792–796
WeightedGraph Visualization tool, 788–791
WeightedGraph class
minimum spanning trees, 776–779
shortest-path problem, 792–796
WeightedGraph Visualization tool
minimum spanning trees, 768–770
shortest-path problem, 788–791
whitespace in Python, described, 8, 9–12
word clouds example (heaps), 694–695
word reversal example for stacks, 112–113
worst case, defined, 266

Z

zooming in 2–3-4 trees, 410–411

Exclusive Offer – 40% OFF

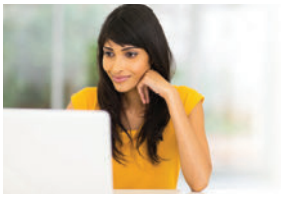
Pearson IT Certification Video Training

livelessons

pearsonitcertification.com/video

Use coupon code **PITCVIDEO40** during checkout.

Video Instruction from Technology Experts



Advance Your Skills

Get started with fundamentals, become an expert, or get certified.

Train Anywhere

Train anywhere, at your own pace, on any device.

Learn

Learn from trusted author trainers published by Pearson IT Certification.

Try Our Popular Video Training for FREE!

pearsonitcertification.com/video

Explore hundreds of **FREE** video lessons from our growing library of Complete Video Courses, LiveLessons, networking talks, and workshops.

PEARSON
IT CERTIFICATION

pearsonitcertification.com/video

ALWAYS LEARNING

PEARSON

Photo by Olena Yakobchuk/Shutterstock

Register Your Product at pearsonITcertification.com/register

Access additional benefits and **save 35%** on your next purchase

- Automatically receive a coupon for 35% off your next purchase, valid for 30 days. Look for your code in your Pearson IT Certification cart or the Manage Codes section of your account page.
- Download available product updates.
- Access bonus material if available.*
- Check the box to hear from us and receive exclusive offers on new editions and related products.

**Registration benefits vary by product. Benefits will be listed on your account page under Registered Products.*

Learning Solutions for Self-Paced Study, Enterprise, and the Classroom

Pearson IT Certification delivers training materials that address the learning, preparation, and practice needs of a new generation of certification candidates, including the official publishing programs of Adobe Press, Cisco Press, and Microsoft Press. At pearsonITcertification.com, you can:

- Shop our books, eBooks, practice tests, software, and video courses
- Sign up to receive special offers
- Access thousands of free chapters and video lessons

Visit **pearsonITcertification.com/community** to connect with Pearson IT Certification

Pearson

Addison-Wesley • Adobe Press • Cisco Press • Microsoft Press • Pearson IT Certification • Que • Sams • Peachpit Press