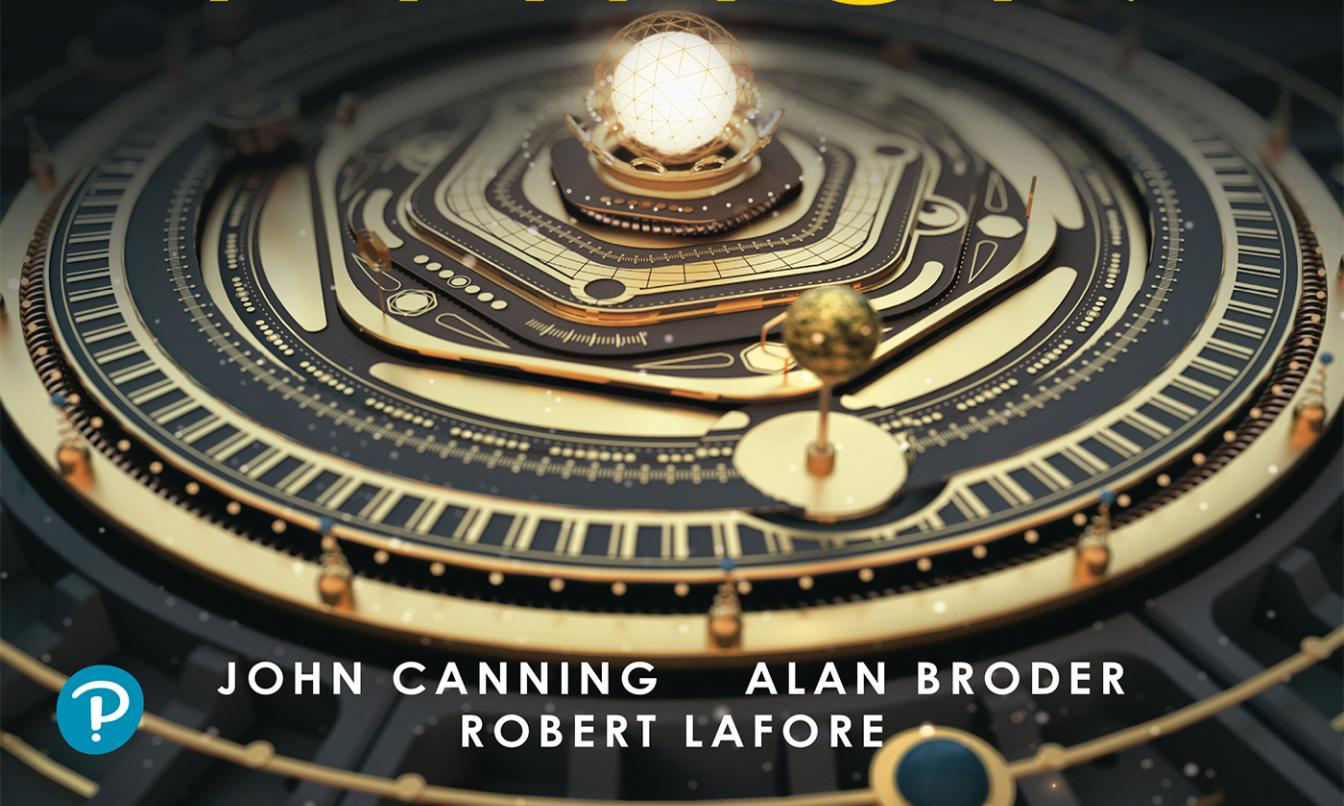




DATA STRUCTURES & ALGORITHMS *in* PYTHON



JOHN CANNING ALAN BRODER
ROBERT LAFORE

FREE SAMPLE CHAPTER



John Canning
Alan Broder
Robert Lafore

Data Structures & Algorithms in Python



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
 Appendices		
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 <i>n</i> 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 <code>BinarySearchTreeTester.py</code> 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
Traversals	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

Appendices

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 Orlan, 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 Melinoff, Atara Neugroschl, Shira Pahmer, Miriam Rabinovich, Etta Rapp, Shira Sassoona, 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

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

Figure 6.1

Credit/Attribution

Shakumant Software

Figure 13.15

Courtesy of Droste B.V.

word cloud produced by
www.wordclouds.com.

Cover

archy13/Shutterstock

CHAPTER 8

Binary Trees

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

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

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.

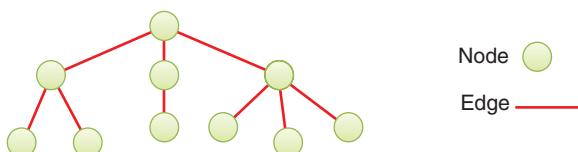


FIGURE 8-1 A general (nonbinary) tree

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.

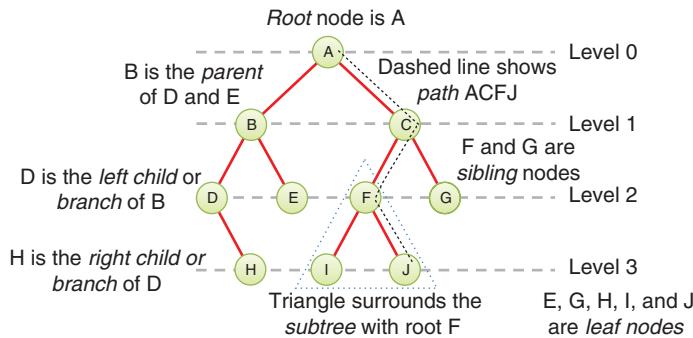


FIGURE 8-2 Tree terms

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.

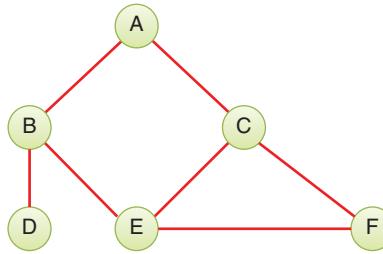


FIGURE 8-3 A nontree

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.

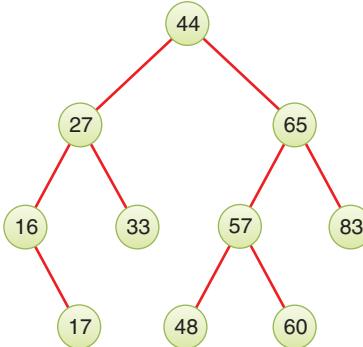


FIGURE 8-4 A binary search tree

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.

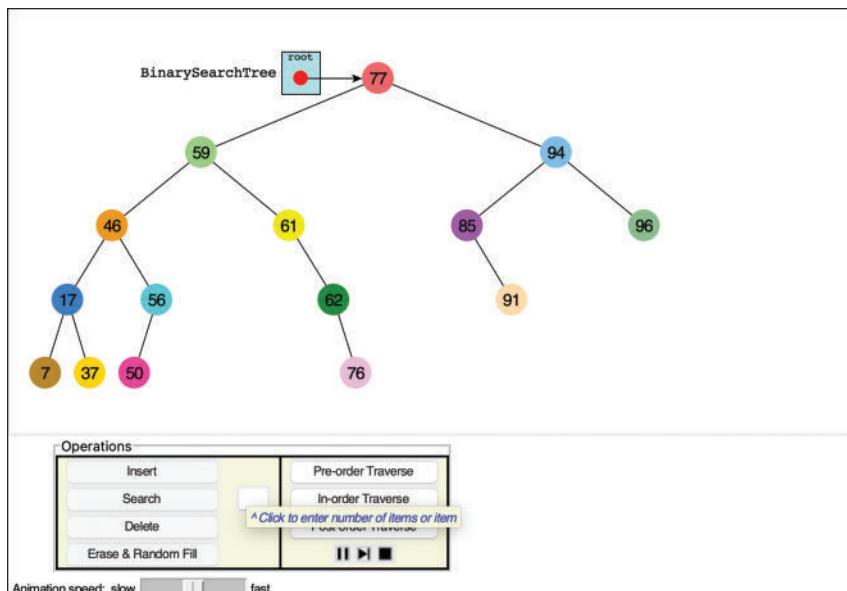


FIGURE 8-5 The Binary Search Tree Visualization tool

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.

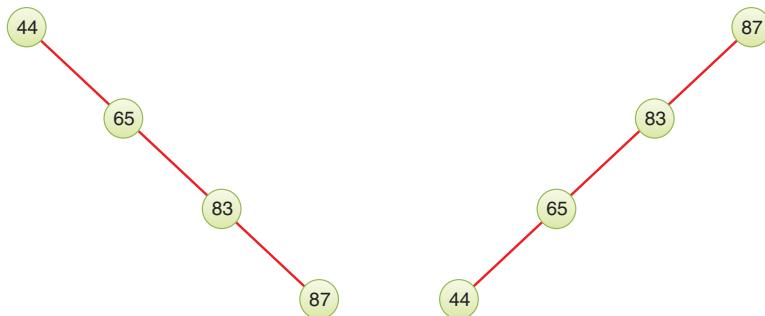


FIGURE 8-6 Trees made by inserting nodes in sorted order

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.

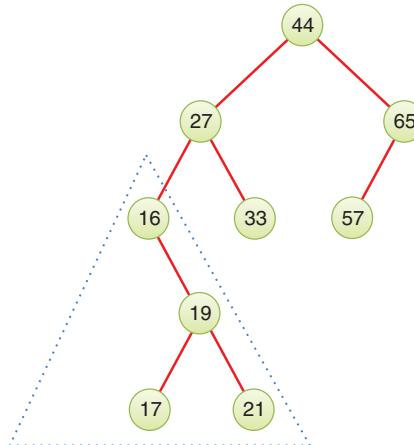


FIGURE 8-7 An unbalanced tree (with an unbalanced subtree)

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):
    def __init__(self, key, data, left=None, right=None):
        self.key = key
        self.data = data
        self.leftChild = left
        self.rightChild = right

        # A node in a binary search tree
        # Constructor takes a key that is
        # used to determine the position
        # inside the search tree,
        # the data associated with the key
        # and a left and right child node
        # if known
        # Copy parameters to instance
        # attributes of the object

    def __str__(self):          # Represent a node as a string
        return "{" + str(self.key) + ", " + str(self.data) + "}"

    def __init__(self):
        self.__root = None          # The tree organizes nodes by their
                                    # 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 (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.

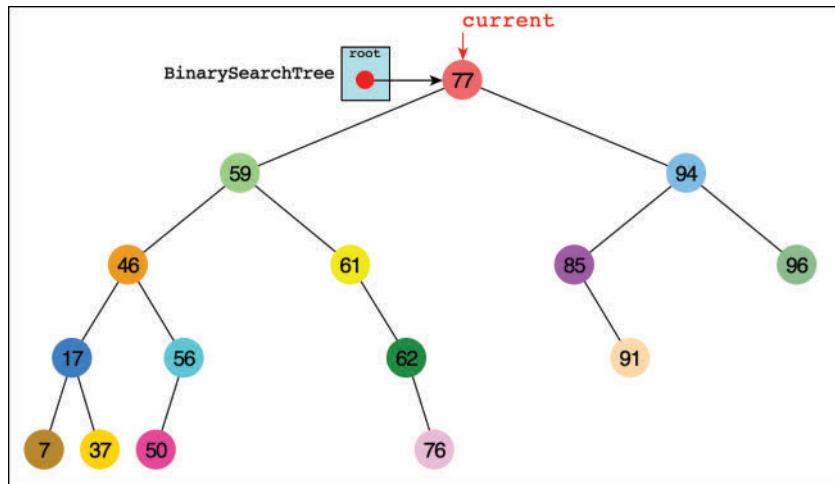


FIGURE 8-8 Finding the node with key 50

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.

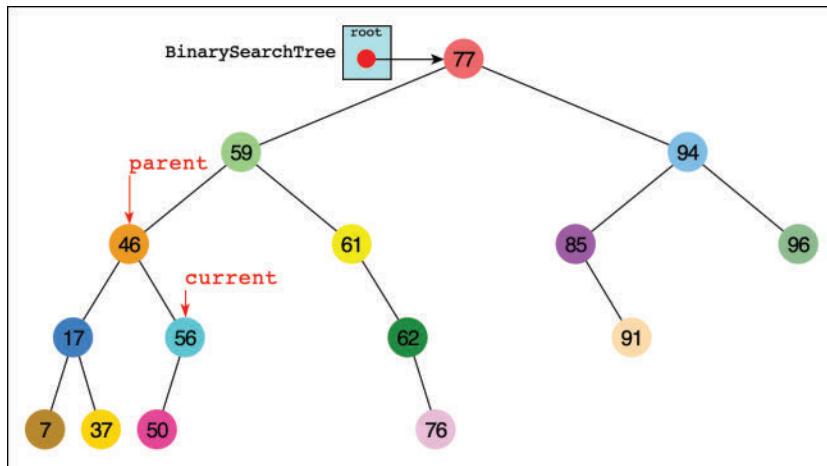


FIGURE 8-9 The second to last step in finding key 50

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):
        current = self.__root            # Find an internal node whose key
        parent = self                     # matches goal and its parent. Start at
        while (current and             # root and track parent of current node
               goal != current.key):     # While there is a tree left to explore
            parent = current           # and current key isn't the goal
            current = (                 # Prepare to move one level down
                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.

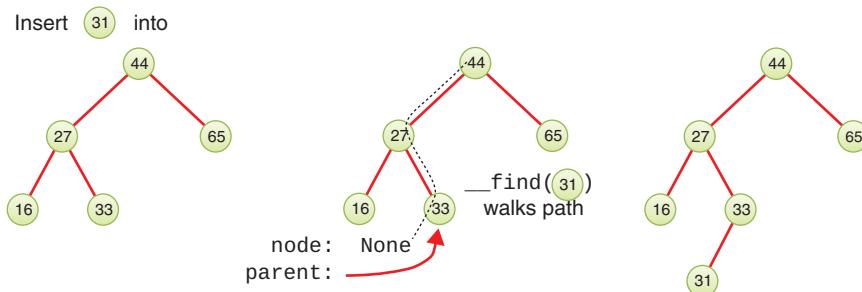


FIGURE 8-10 Inserting a node in binary search tree

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.

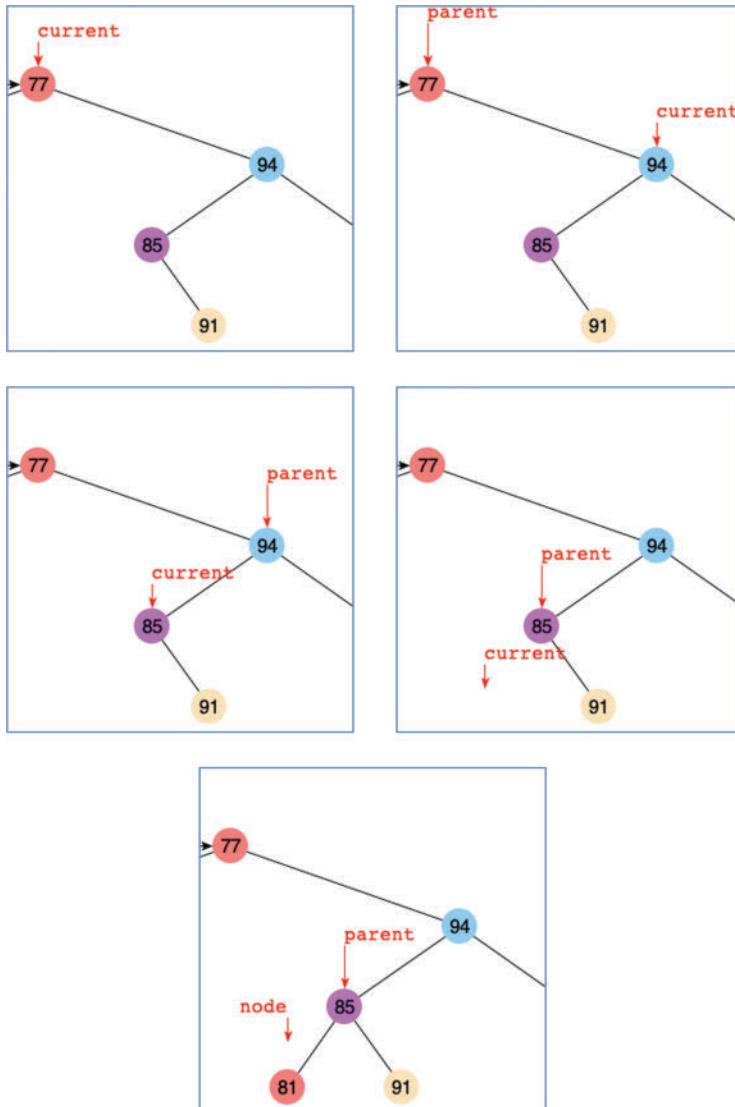


FIGURE 8-11 Steps for inserting a node with key 81 using the Visualization tool

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,
               key,
               data):
        node, parent = self.__find(
            key)
        if node:
            node.data = data
            return False
        if parent is self:
            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.

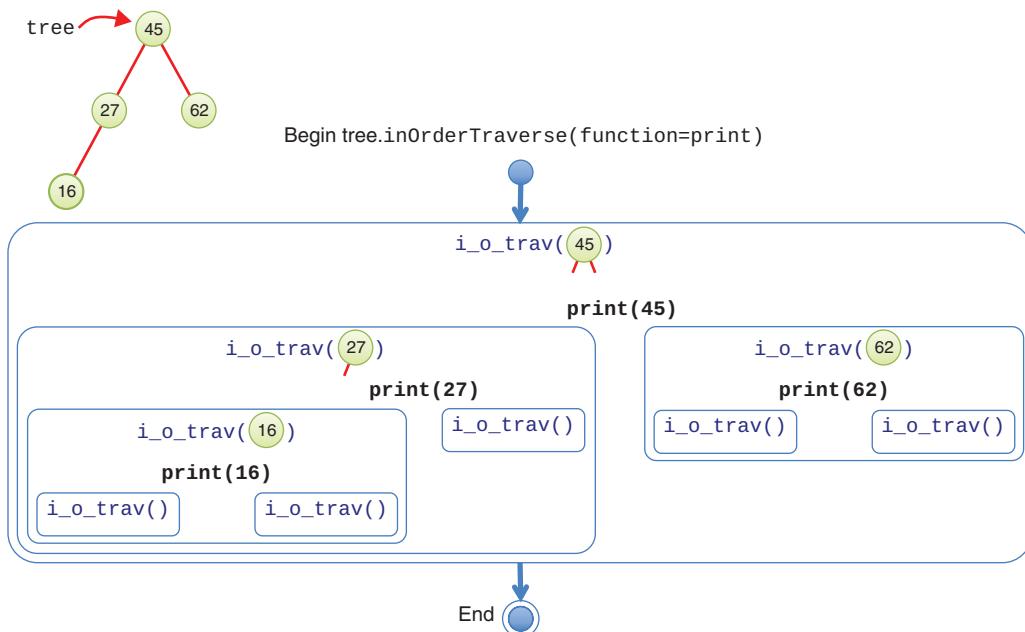


FIGURE 8-12 In-order traversal of a small tree

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 traversals 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.

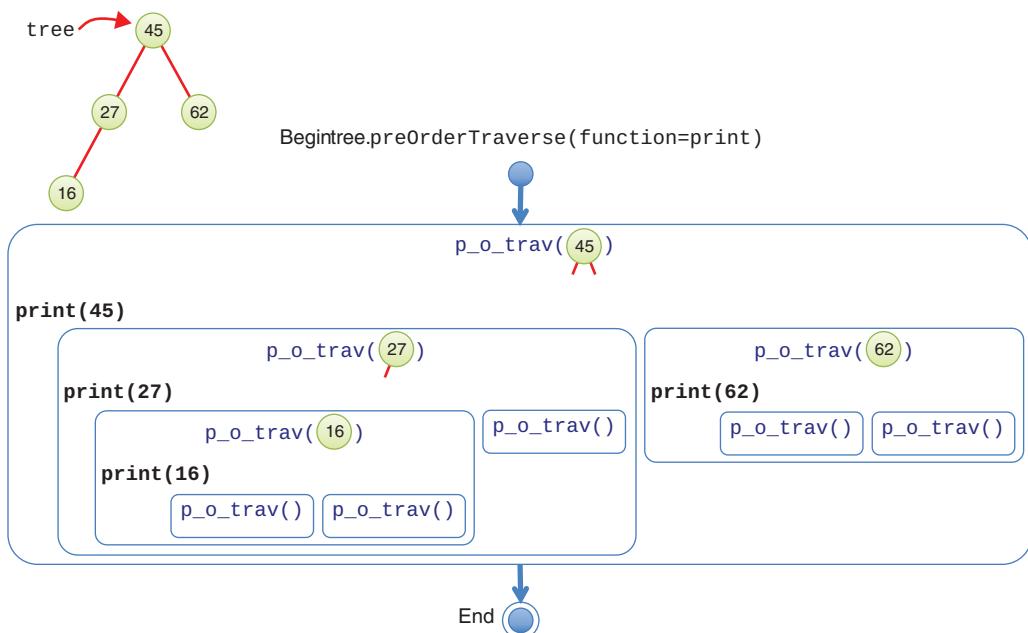


FIGURE 8-13 Pre-order traversal of a small tree

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 `__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,
                      traverseType="in"):      # Traverse the tree recursively 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,
                  node,                      # Recursive generator to traverse
                  traverseType):            # subtree rooted at node in pre, in, or
                                         # 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 $O(N \times \log N)$ 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,
                 traverseType='in'):
        if traverseType not in [
            'pre', 'in', 'post']:
            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.

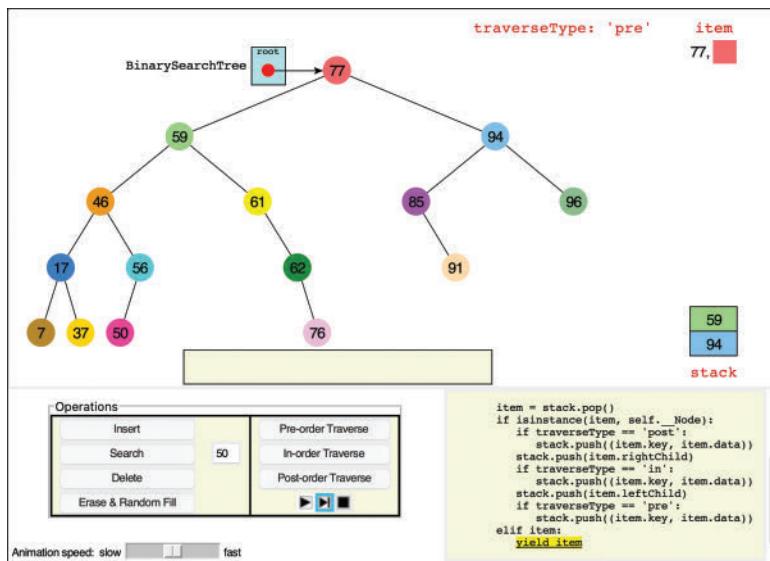
Traversing with the Visualization Tool

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

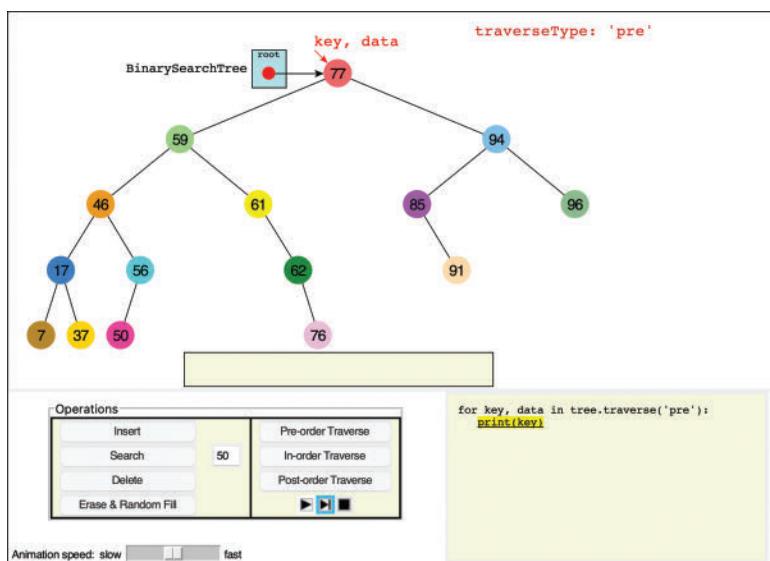
```
for key, data in tree.traverse("pre"):
    print(key)
```

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.

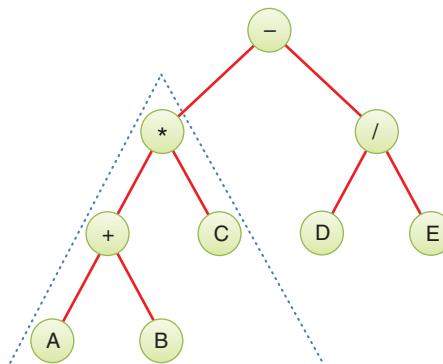


FIGURE 8-16 Binary tree representing an algebraic expression

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.

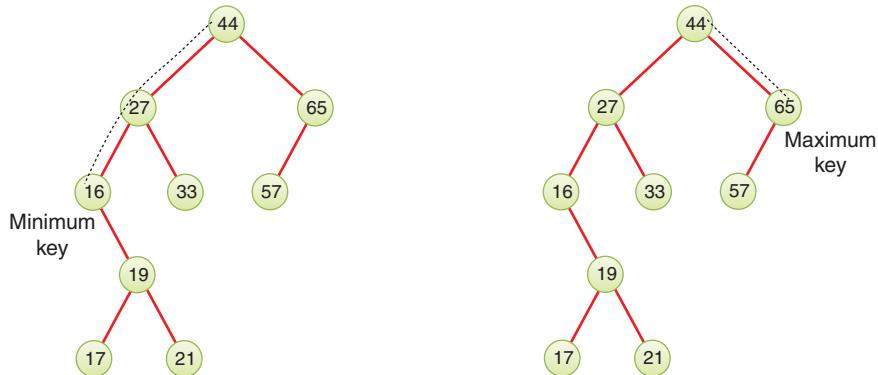


FIGURE 8-17 Minimum and maximum key values of a binary search tree

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.

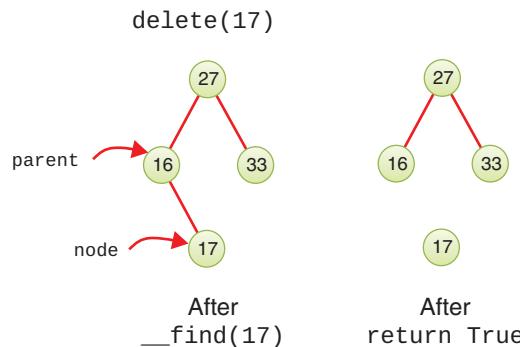


FIGURE 8-18 Deleting a node with no children

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.

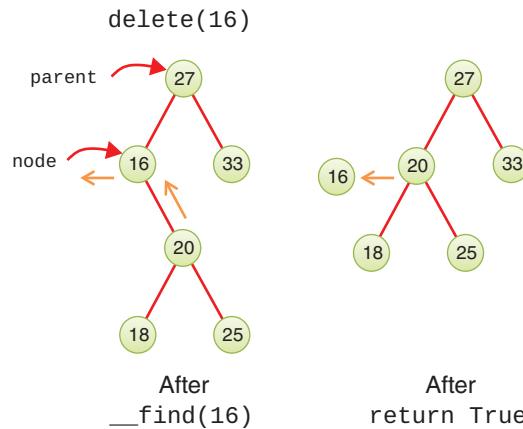


FIGURE 8-19 Deleting a node with one child

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.

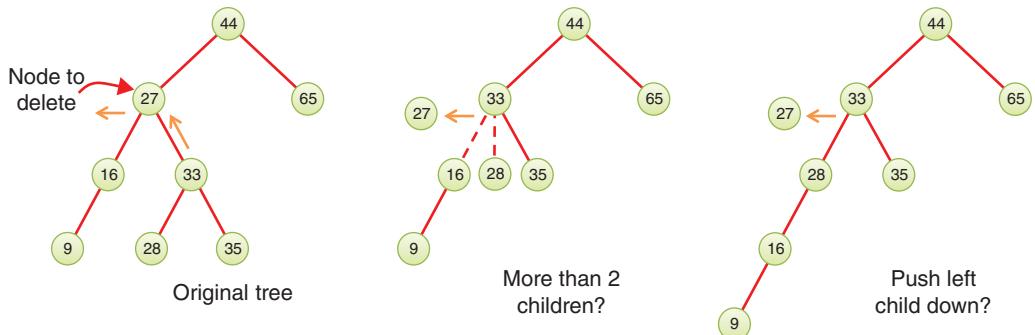


FIGURE 8-20 Options for deleting a node with two subtrees

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.

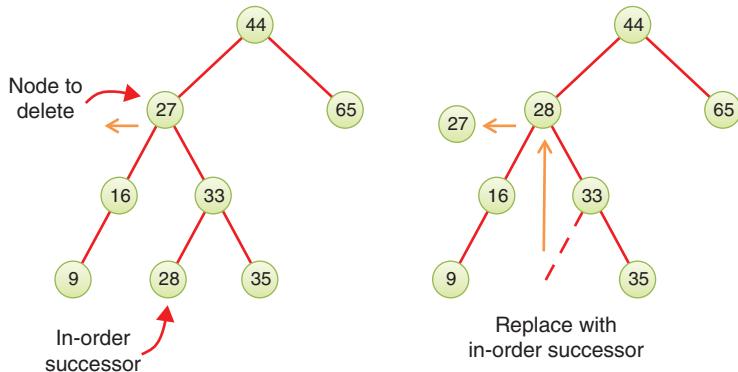


FIGURE 8-21 Node replaced by its successor

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.

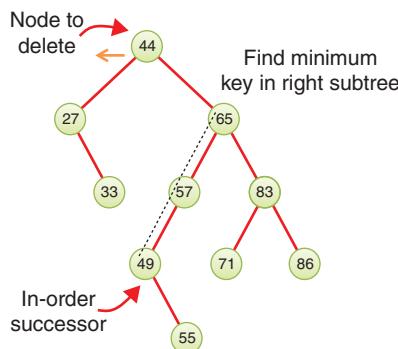


FIGURE 8-22 Finding the successor

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.

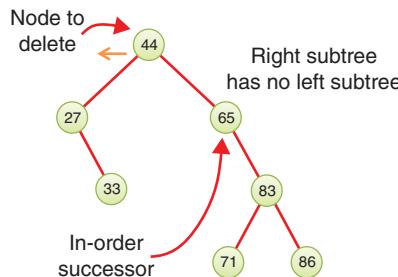


FIGURE 8-23 The right child is the successor

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
        node):                     # its data in this node, and delete the
            successor = node.rightChild # successor from the right subtree
            parent = node             # Start search for successor in
            while successor.leftChild: # right subtree and track its parent
                parent = successor   # Descend left child links until
                successor = successor.leftChild # no more left links, tracking parent
                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.

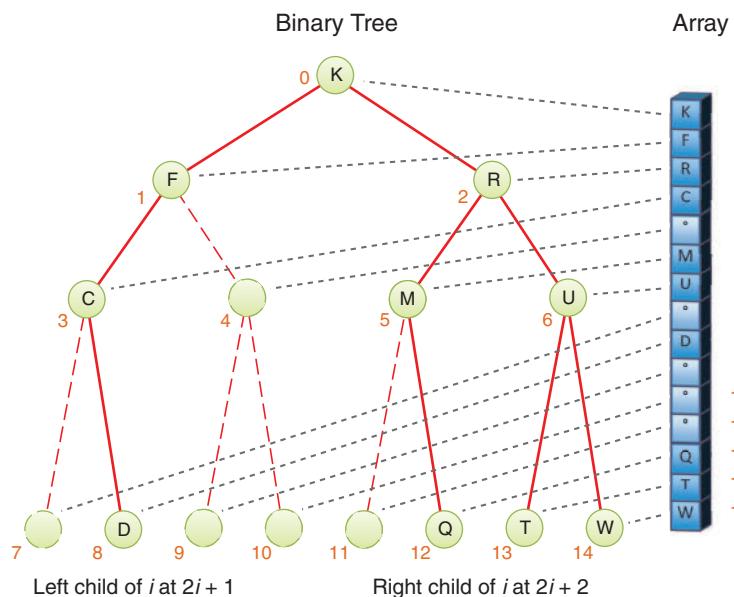


FIGURE 8-24 A binary tree represented by an array

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.

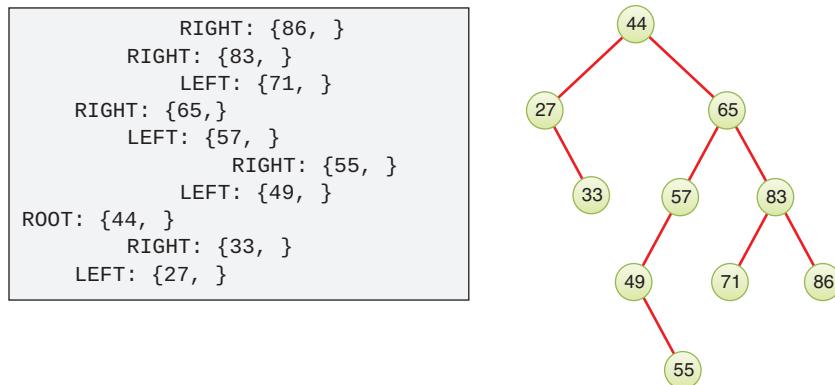


FIGURE 8-25 Tree printed with indentation for node depth

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.

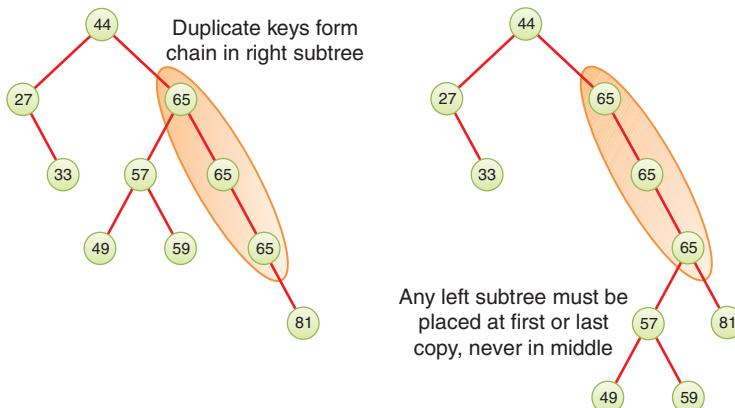


FIGURE 8-26 Duplicate keys in binary search trees

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.

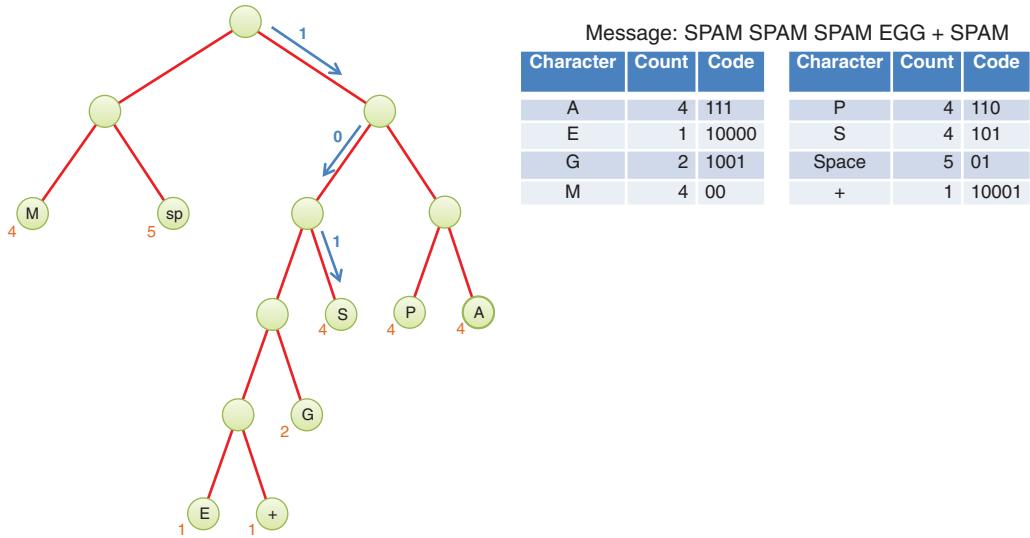


FIGURE 8-27 Huffman tree for the SPAM message

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.

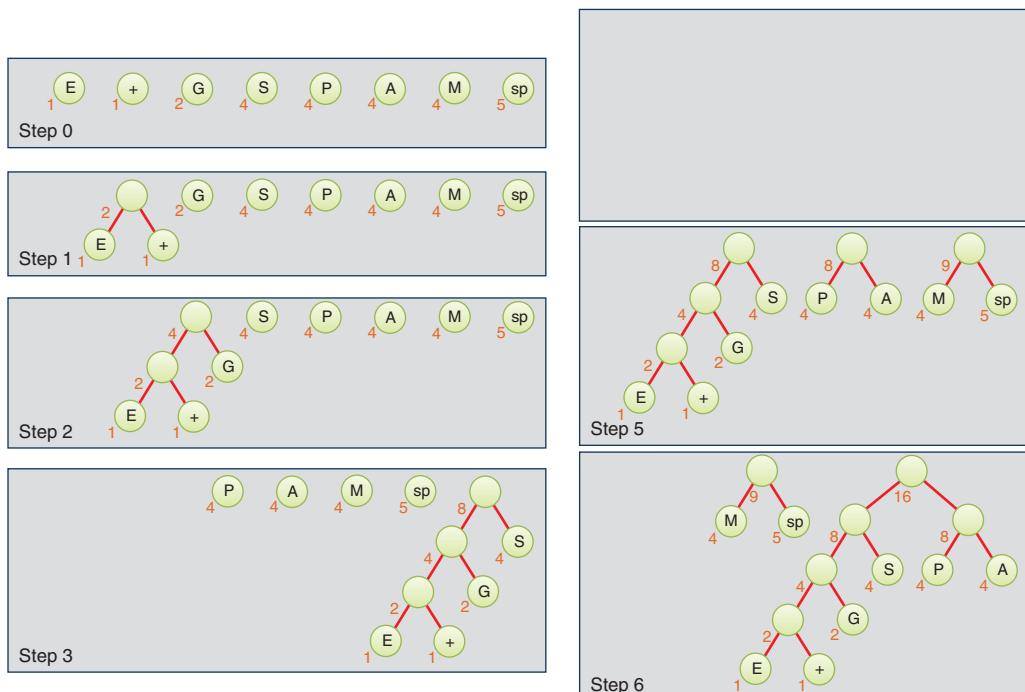


FIGURE 8-28 Growing the Huffman tree, first six steps

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.

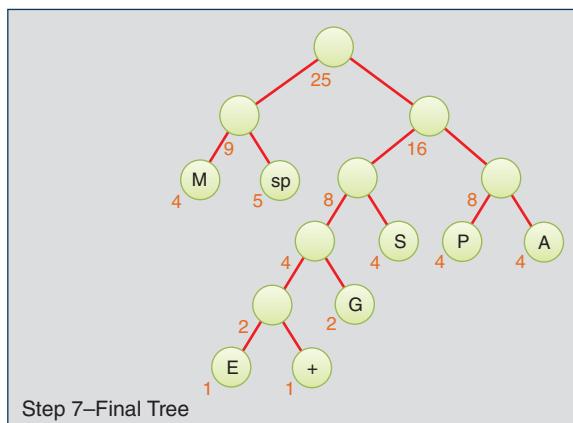


FIGURE 8-29 Growing the Huffman tree, final step

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.

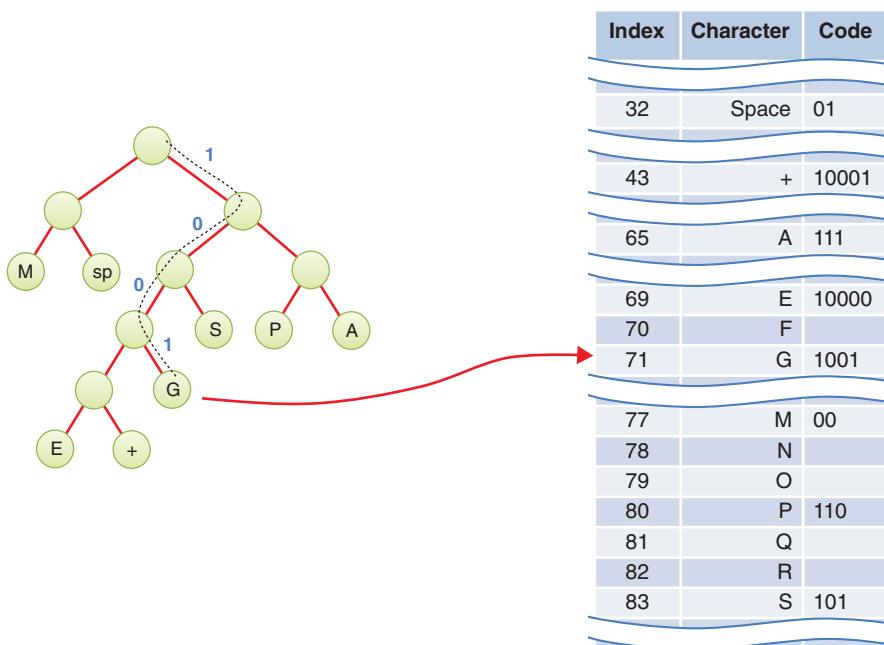


FIGURE 8-30 Building the code table

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:
- Delete a node that has no children.
 - Delete a node that has 1 child node.
 - Delete a node that has 2 child nodes.
 - 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?
 - 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 `bytarray` 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 $n = \text{None}$. 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, 6, n],
[55, 12, n, n, n, n, 4, n, 8, n, n, n, n, n, n, n, n, n, 6, n]
```

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

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
- 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
- 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

R

- radix, defined, 321
- radix sort, 320–323

- 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
- 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 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
 - Pop button, 108
 - Push button, 107
 - size of stack, 108
- 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
 - 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

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

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

- 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
- 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

W

- Warshall, Stephen, 752, 798
- Warshall's algorithm
 - implementation, 756
 - transitive closure and, 751–758

Z

- zooming in 2–3–4 trees, 410–411