Page 7

Book Features

This book is based upon the book Data Structures and Algorithms in Java by Goodrich and Tamassia, and the related Data Structures and Algorithms in C++ by Goodrich, Tamassia, and Mount. However, this book is not simply a translation of those other books to Python. In adapting the material for this book, we have signi■cantly redesigned the organization and content of the book as follows:

- The code base has been entirely redesigned to take advantage of the features of Python, such as use of generators for iterating elements of a collection.
- Many algorithms that were presented as pseudo-code in the Java and C++ versions are directly presented as complete Python code.
- In general, ADTs are de■ned to have consistent interface with Python's built-in data types and those in Python's collections module.
- Chapter 5 provides an in-depth exploration of the dynamic array-based underpinnings of Python's built-in list, tuple, and str classes. New Appendix A serves as an additional reference regarding the functionality of the str class.
- Over 450 illustrations have been created or revised.
- New and revised exercises bring the overall total number to 750.

Online Resources

This book is accompanied by an extensive set of online resources, which can be found at the following Web site:

www.wiley.com/college/goodrich

Students are encouraged to use this site along with the book, to help with exercises and increase understanding of the subject. Instructors are likewise welcome to use the site to help plan, organize, and present their course materials. Included on this Web site is a collection of educational aids that augment the topics of this book, for both students and instructors. Because of their added value, some of these online resources are password protected.

For all readers, and especially for students, we include the following resources:

- All the Python source code presented in this book.
- PDF handouts of Powerpoint slides (four-per-page) provided to instructors.
- A database of hints to all exercises, indexed by problem number.

For instructors using this book, we include the following additional teaching aids:

- Solutions to hundreds of the book's exercises.
- Color versions of all ■gures and illustrations from the book.
- Slides in Powerpoint and PDF (one-per-page) format.

The slides are fully editable, so as to allow an instructor using this book full freedom in customizing his or her presentations. All the online resources are provided at no extra charge to any instructor adopting this book for his or her course.

Page 639

Chapter Notes

The KMP algorithm is described by Knuth, Morris, and Pratt in their journal article [66], and Boyer and Moore describe their algorithm in a journal article published the same year [18]. In their article, however, Knuth et al. [66] also prove that the Boyer-Moore algorithm runs in linear time. More recently, Cole [27] shows that the Boyer-Moore algorithm makes at most 3n character comparisons in the worst case, and this bound is tight. All of the algorithms discussed above are also discussed in the book chapter by Aho [4], albeit in a more theoretical framework, including the methods for regular-expression pattern matching. The reader interested in further study of string pattern-matching algorithms is referred to the book by Stephen [90] and the book chapters by Aho [4], and Crochemore and Lecroq [30].

Dynamic programming was developed in the operations research community and formalized by Bellman [13].

The trie was invented by Morrison [79] and is discussed extensively in the classic Sorting and Searching book by Knuth [65]. The name “Patricia” is short for “Practical Algorithm to Retrieve Information Coded in Alphanumeric” [79]. McCreight [73] shows how to construct suf■x tries in linear time. An introduction to the ■eld of information retrieval, which includes a discussion of search engines for the Web, is provided in the book by Baeza-Yates and Ribeiro-Neto [8].

Page 717

P-14.81 Write a program that builds the routing tables for the nodes in a computer network, based on shortest-path routing, where path distance is measured by hop count, that is, the number of edges in a path. The input for this problem is the connectivity information for all the nodes in the network, as in the following example:

241.12.31.14 : 241.12.31.15 241.12.31.18 241.12.31.19

which indicates three network nodes that are connected to 241.12.31.14, that is, three nodes that are one hop away. The routing table for the node at address A is a set of pairs (B,C), which indicates that, to route a message from A to B, the next node to send to (on the shortest path from A to B) is C. Your program should output the routing table for each node in the network, given an input list of node connectivity lists, each of which is input in the syntax as shown above, one per line.

Chapter Notes

The depth-■rst search method is a part of the “folklore” of computer science, but Hopcroft and Tarjan [52, 94] are the ones who showed how useful this algorithm is for solving several different graph problems. Knuth [64] discusses the topological sorting problem. The simple linear-time algorithm that we describe for determining if a directed graph is strongly connected is due to Kosaraju. The Floyd-Warshall algorithm appears in a paper by Floyd [38] and is based upon a theorem of Warshall [102].

The ■rst known minimum spanning tree algorithm is due to Baru■vka [9], and was published in 1926. The Prim-Jarn■■k algorithm was ■rst published in Czech by Jarn■■k [55] in 1930 and in English in 1957 by Prim [85]. Kruskal published his minimum spanning tree algorithm in 1956 [67]. The reader interested in further study of the history of the minimum spanning tree problem is referred to the paper by Graham and Hell [47]. The current asymptotically fastest minimum spanning tree algorithm is a randomized method of Karger, Klein, and Tarjan [57] that runs in O(m) expected time. Dijkstra [35] published his single-source, shortest-path algorithm in 1959. The running time for the Prim-Jarn■■k algorithm, and also that of Dijkstra’s algorithm, can actually be improved to be O(n log n + m) by implementing the queue Q with either of two more sophisticated data structures, the “Fibonacci Heap” [40] or the “Relaxed Heap” [37].

To learn about different algorithms for drawing graphs, please see the book chapter by Tamassia and Liotta [92] and the book by Di Battista, Eades, Tamassia and Tollis [34]. The reader interested in further study of graph algorithms is referred to the books by Ahuja, Magnanti, and Orlin [7], Cormen, Leiserson, Rivest and Stein [29], Mehlhorn [77], and Tarjan [95], and the book chapter by van Leeuwen [98].

C-15.20 Consider the page caching strategy based on the least frequently used (LFU) rule, where the page in the cache that has been accessed the least often is the one that is evicted when a new page is requested. If there are ties, LFU evicts the least frequently used page that has been in the cache the longest. Show that there is a sequence P of n requests that causes LFU to miss $\Omega(n)$ times for a cache of m pages, whereas the optimal algorithm will miss only O(m) times.

C-15.21 Suppose that instead of having the node-search function f (d) = 1 in an order-d B-tree T , we have f (d) = log d. What does the asymptotic running time of performing a search in T now become?

Projects

P-15.22 Write a Python class that simulates the best-■t, worst-■t, ■rst-■t, and next-■t algorithms for memory management. Determine experimentally which method is the best under various sequences of memory requests.

P-15.23 Write a Python class that implements all the methods of the ordered map ADT by means of an (a, b) tree, where a and b are integer constants passed as parameters to a constructor.

P-15.24 Implement the B-tree data structure, assuming a block size of 1024 and integer keys. Test the number of “disk transfers” needed to process a sequence of map operations.

Chapter Notes

The reader interested in the study of the architecture of hierarchical memory systems is referred to the book chapter by Burger et al. [21] or the book by Hennessy and Patterson [50]. The mark-sweep garbage collection method we describe is one of many different algorithms for performing garbage collection. We encourage the reader interested in further study of garbage collection to examine the book by Jones and Lins [56]. Knuth [62] has very nice discussions about external-memory sorting and searching, and Ullman [97] discusses external memory structures for database systems. The handbook by Gonnet and Baeza-Yates [44] compares the performance of a number of different sorting algorithms, many of which are external-memory algorithms. B-trees were invented by Bayer and McCreight [11] and Comer [28] provides a very nice overview of this data structure. The books by Mehlhorn [76] and Samet [87] also have nice discussions about B-trees and their variants. Aggarwal and Vitter [3] study the I/O complexity of sorting and related problems, establishing upper and lower bounds. Goodrich et al. [46] study the I/O complexity of several computational geometry problems. The reader interested in further study of I/O-ef■cient algorithms is encouraged to examine the survey paper of Vitter [99].

Page 129

Chapter Notes

For a broad overview of developments in computer science and engineering, we refer the reader to The Computer Science and Engineering Handbook [96]. For more information about the Therac-25 incident, please see the paper by Leveson and Turner [69].

The reader interested in studying object-oriented programming further, is referred to the books by Booch [17], Budd [20], and Liskov and Guttag [71]. Liskov and Guttag also provide a nice discussion of abstract data types, as does the survey paper by Cardelli and Wegner [23] and the book chapter by Demurjian [33] in the The Computer Science and Engineering Handbook [96]. Design patterns are described in the book by Gamma et al. [41].

Books with speci■c focus on object-oriented programming in Python include those by Goldwasser and Letscher [43] at the introductory level, and by Phillips [83] at a more advanced level,

Page 479

Chapter Notes

Hashing is a well-studied technique. The reader interested in further study is encouraged to explore the book by Knuth [65], as well as the book by Vitter and Chen [100]. Skip lists were introduced by Pugh [86]. Our analysis of skip lists is a simpli■cation of a presentation given by Motwani and Raghavan [80]. For a more in-depth analysis of skip lists, please see the various research papers on skip lists that have appeared in the data structures literature [59, 81, 84]. Exercise C-10.36 was contributed by James Lee.

Page 168

P-3.57 Perform experimental analysis to test the hypothesis that Python's sorted method runs in O(n log n) time on average.

P-3.58 For each of the three algorithms, unique1, unique2, and unique3, which solve the element uniqueness problem, perform an experimental analysis to determine the largest value of n such that the given algorithm runs in one minute or less.

Chapter Notes

The big-Oh notation has prompted several comments about its proper use [19, 49, 63]. Knuth [64, 63] de■nes it using the notation f (n) = O(g(n)), but says this "equality" is only "one way." We have chosen to take a more standard view of equality and view the big-Oh notation as a set, following Brassard [19]. The reader interested in studying average-case analysis is referred to the book chapter by Vitter and Flajolet [101]. For some additional mathematical tools, please refer to Appendix B.

Page 325

Ordered Trees

A tree is ordered if there is a meaningful linear order among the children of each node; that is, we purposefully identify the children of a node as being the ■rst, second, third, and so on. Such an order is usually visualized by arranging siblings left to right, according to their order.

Example 8.3: The components of a structured document, such as a book, are hierarchically organized as a tree whose internal nodes are parts, chapters, and sections, and whose leaves are paragraphs, tables, ■gures, and so on. (See Figure 8.6.) The root of the tree corresponds to the book itself. We could, in fact, consider expanding the tree further to show paragraphs consisting of sentences, sentences consisting of words, and words consisting of characters. Such a tree is an example of an ordered tree, because there is a well-de■ned order among the children of each node.

Book

Preface Part A Part B References

¶ ... ¶ Ch. 1 ... Ch. 5 Ch. 6 ... Ch. 9 ¶ ... ¶

§ 1.1 ... § 1.4 § 5.1 ... § 5.7 § 6.1 ... § 6.5 § 9.1 ... § 9.6

¶ ... ¶ ... ¶ ... ¶

Figure 8.6: An ordered tree associated with a book.

Let’s look back at the other examples of trees that we have described thus far, and consider whether the order of children is signi■cant. A family tree that describes generational relationships, as in Figure 8.1, is often modeled as an ordered tree, with siblings ordered according to their birth.

In contrast, an organizational chart for a company, as in Figure 8.2, is typically considered an unordered tree. Likewise, when using a tree to describe an inheritance hierarchy, as in Figure 8.4, there is no particular signi■cance to the order among the subclasses of a parent class. Finally, we consider the use of a tree in modeling a computer’s ■le system, as in Figure 8.3. Although an operating system often displays entries of a directory in a particular order (e.g., alphabetical, chronological), such an order is not typically inherent to the ■le system’s representation.

P-8.68 Write a program that can play Tic-Tac-Toe effectively. (See Section 5.6.) To do this, you will need to create a game tree T , which is a tree where each position corresponds to a game con■guration, which, in this case, is a representation of the Tic-Tac-Toe board. (See Section 8.4.2.) The root corresponds to the initial con■guration. For each internal position p in T , the children of p correspond to the game states we can reach from p's game state in a single legal move for the appropriate player, A (the ■rst player) or B (the second player). Positions at even depths correspond to moves for A and positions at odd depths correspond to moves for B. Leaves are either ■nal game states or are at a depth beyond which we do not want to explore. We score each leaf with a value that indicates how good this state is for player A. In large games, like chess, we have to use a heuristic scoring function, but for small games, like Tic-Tac-Toe, we can construct the entire game tree and score leaves as +1, 0, –1, indicating whether player A has a win, draw, or lose in that con■guration. A good algorithm for choosing moves is minimax. In this algorithm, we assign a score to each internal position p in T , such that if p represents A's turn, we compute p's score as the maximum of the scores of p's children (which corresponds to A's optimal play from p). If an internal node p represents B's turn, then we compute p's score as the minimum of the scores of p's children (which corresponds to B's optimal play from p).

P-8.69 Implement the tree ADT using the binary tree representation described in Exercise C-8.43. You may adapt the LinkedBinaryTree implementation.

P-8.70 Write a program that takes as input a general tree T and a position p of T and converts T to another tree with the same set of position adjacencies, but now with p as its root.

Chapter Notes

Discussions of the classic preorder, inorder, and postorder tree traversal methods can be found in Knuth's Fundamental Algorithms book [64]. The Euler tour traversal technique comes from the parallel algorithms community; it is introduced by Tarjan and Vishkin [93] and is discussed by Ja■Ja■ [54] and by Karp and Ramachandran [58]. The algorithm for drawing a tree is generally considered to be a part of the "folklore" of graph-drawing algorithms. The reader interested in graph drawing is referred to the book by Di Battista, Eades, Tamassia, and Tollis [34] and the survey by Tamassia and Liotta [92]. The puzzle in Exercise R-8.12 was communicated by Micha Sharir.

Page 467

10.5 Sets, Multisets, and Multimaps

We conclude this chapter by examining several additional abstractions that are closely related to the map ADT, and that can be implemented using data structures similar to those for a map.

- A set is an unordered collection of elements, without duplicates, that typically supports ef■cient membership tests. In essence, elements of a set are like keys of a map, but without any auxiliary values.
- A multiset (also known as a bag) is a set-like container that allows duplicates.
- A multimap is similar to a traditional map, in that it associates values with keys; however, in a multimap the same key can be mapped to multiple values. For example, the index of this book maps a given term to one or more locations at which the term occurs elsewhere in the book.

10.5.1 The Set ADT

Python provides support for representing the mathematical notion of a set through the built-in classes frozenset and set, as originally discussed in Chapter 1, with frozenset being an immutable form. Both of those classes are implemented using hash tables in Python.

Python's collections module de■nes abstract base classes that essentially mirror these built-in classes. Although the choice of names is counterintuitive, the abstract base class collections.Set matches the concrete frozenset class, while the abstract base class collections.MutableSet is akin to the concrete set class.

In our own discussion, we equate the "set ADT" with the behavior of the built-in set class (and thus, the collections.MutableSet base class). We begin by listing what we consider to be the ■ve most fundamental behaviors for a set S:

S.add(e): Add element e to the set. This has no effect if the set already contains e.

S.discard(e): Remove element e from the set, if present. This has no effect if the set does not contain e.

e in S: Return True if the set contains element e. In Python, this is implemented with the special contains method.

len(S): Return the number of elements in set S. In Python, this is implemented with the special method len .

iter(S): Generate an iteration of all elements of the set. In Python, this is implemented with the special method iter .