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.

and place that item in Si+1 if the coin comes up ·heads.· Thus, we expect S1 to have about n/2 items, S2 to have about n/4 items, and, in general, Si to have about n/2i items. In other words, we expect the height h of S to be about log n. The halving of the number of items from one list to the next is not enforced as an explicit property of skip lists, however. Instead, randomization is used.

Functions that generate numbers that can be viewed as random numbers are built into most modern computers, because they are used extensively in computer games, cryptography, and computer simulations, Some functions, called pseudo-random number generators, generate random-like numbers, starting with an initial seed. (See discusion of random module in Section 1.11.1.) Other methods use hardware devices to extract ·true· random numbers from nature. In any case, we will assume that our computer has access to numbers that are suf·ciently random for our analysis.

The main advantage of using randomization in data structure and algorithm design is that the structures and functions that result are usually simple and ef·cient. The skip list has the same logarithmic time bounds for searching as is achieved by the binary search algorithm, yet it extends that performance to update methods when inserting or deleting items. Nevertheless, the bounds are expected for the skip list, while binary search has a worst-case bound with a sorted table.

A skip list makes random choices in arranging its structure in such a way that search and update times are O(log n) on average, where n is the number of items in the map. Interestingly, the notion of average time complexity used here does not depend on the probability distribution of the keys in the input. Instead, it depends on the use of a random-number generator in the implementation of the insertions to help decide where to place the new item. The running time is averaged over all possible outcomes of the random numbers used when inserting entries.

Using the position abstraction used for lists and trees, we view a skip list as a two-dimensional collection of positions arranged horizontally into levels and vertically into towers. Each level is a list Si and each tower contains positions storing the same item across consecutive lists. The positions in a skip list can be traversed using the following operations:

next(p): Return the position following p on the same level.
prev(p): Return the position preceding p on the same level.
below(p): Return the position below p in the same tower.
above(p): Return the position above p in the same tower.

We conventionally assume that the above operations return None if the position requested does not exist. Without going into the details, we note that we can easily implement a skip list by means of a linked structure such that the individual traversal methods each take O(1) time, given a skip-list position p. Such a linked structure is essentially a collection of h doubly linked lists aligned at towers, which are also doubly linked lists.

10.4 Skip Lists

An interesting data structure for realizing the sorted map ADT is the skip list. In Section 10.3.1, we saw that a sorted array will allow O(log n)-time searches via the binary search algorithm. Unfortunately, update operations on a sorted array have O(n) worst-case running time because of the need to shift elements. In Chapter 7 we demonstrated that linked lists support very ef·cient update operations, as long as the position within the list is identi·ed. Unfortunately, we cannot perform fast searches on a standard linked list; for example, the binary search algorithm requires an ef·cient means for direct accessing an element of a sequence by index.

Skip lists provide a clever compromise to ef·ciently support search and update operations. A skip list S for a map M consists of a series of lists {S0 , S1 , . . . , Sh }. Each list Si stores a subset of the items of M sorted by increasing keys, plus items with two sentinel keys denoted ·· and +·, where ·· is smaller than every possible key that can be inserted in M and +· is larger than every possible key that can be inserted in M. In addition, the lists in S satisfy the following:

- List S0 contains every item of the map M (plus sentinels ·· and +·).
- For i = 1, . . . , h · 1, list Si contains (in addition to ·· and +·) a randomly generated subset of the items in list Si·1 .
- List Sh contains only ·· and +·.

An example of a skip list is shown in Figure 10.10. It is customary to visualize a skip list S with list S0 at the bottom and lists S1 , . . . , Sh above it. Also, we refer to h as the height of skip list S.

Intuitively, the lists are set up so that Si+1 contains more or less alternate items of Si . As we shall see in the details of the insertion method, the items in Si+1 are chosen at random from the items in Si by picking each item from Si to also be in Si+1 with probability 1/2. That is, in essence, we ··ip a coin· for each item in Si

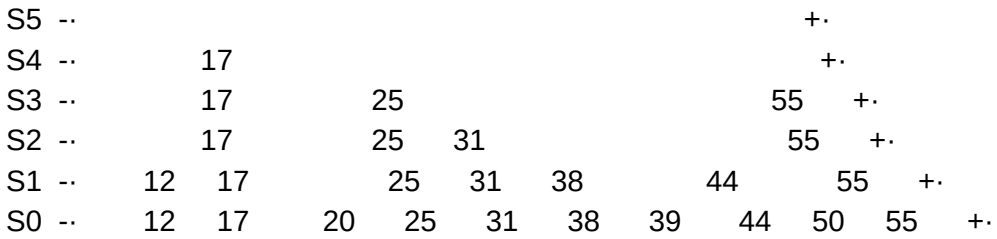

Figure 10.10: Example of a skip list storing 10 items. For simplicity, we show only the items· keys, not their associated values.

Removal in a Skip List

Like the search and insertion algorithms, the removal algorithm for a skip list is quite simple. In fact, it is even easier than the insertion algorithm. That is, to perform the map operation del M[k] we begin by executing method SkipSearch(k). If the position p stores an entry with key different from k, we raise a KeyError. Otherwise, we remove p and all the positions above p, which are easily accessed by using above operations to climb up the tower of this entry in S starting at position p. While removing levels of the tower, we reestablish links between the horizontal neighbors of each removed position. The removal algorithm is illustrated in Figure 10.13 and a detailed description of it is left as an exercise (R-10.24). As we show in the next subsection, deletion operation in a skip list with n entries has O(log n) expected running time.

Before we give this analysis, however, there are some minor improvements to the skip-list data structure we would like to discuss. First, we do not actually need to store references to values at the levels of the skip list above the bottom level, because all that is needed at these levels are references to keys. In fact, we can more ef·ciently represent a tower as a single object, storing the key-value pair, and maintaining j previous references and j next references if the tower reaches level S j . Second, for the horizontal axes, it is possible to keep the list singly linked, storing only the next references. We can perform insertions and removals in strictly a top-down, scan-forward fashion. We explore the details of this optimization in Exercise C-10.44. Neither of these optimizations improve the asymptotic performance of skip lists by more than a constant factor, but these improvements can, nevertheless, be meaningful in practice. In fact, experimental evidence suggests that optimized skip lists are faster in practice than AVL trees and other balanced search trees, which are discussed in Chapter 11.

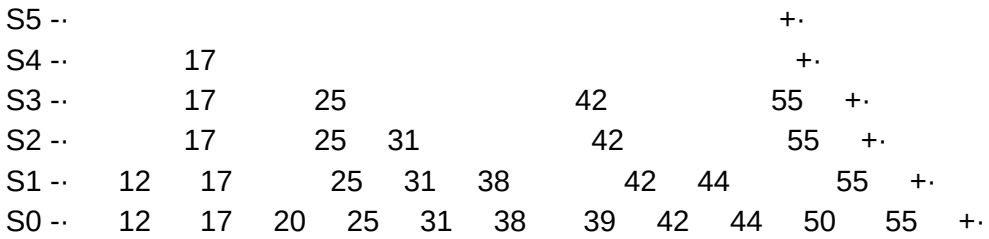

Figure 10.13: Removal of the entry with key 25 from the skip list of Figure 10.12. The positions visited after the search for the position of S0 holding the entry are highlighted. The positions removed are drawn with dashed lines.

```
Algorithm SkipSearch(k):
    Input: A search key k
    Output: Position p in the bottom list S0 with the largest key such that key(p) · k
     p = start                                  {begin at start position}
     while below(p)  = None do
       p = below(p)                                  {drop down}
       while k · key(next(p)) do
          p = next(p)                             {scan forward}
     return p.
```

Code Fragment 10.12: Algorthm to search a skip list S for key k.

As it turns out, the expected running time of algorithm SkipSearch on a skip list with n entries is O(log n). We postpone the justi·cation of this fact, however, until after we discuss the implementation of the update methods for skip lists. Navigation starting at the position identi·ed by SkipSearch(k) can be easily used to provide the additional forms of searches in the sorted map ADT (e.g., ·nd gt, ·nd range).

Insertion in a Skip List

The execution of the map operation M[k] = v begins with a call to SkipSearch(k). This gives us the position p of the bottom-level item with the largest key less than or equal to k (note that p may hold the special item with key ··). If key(p) = k, the associated value is overwritten with v. Otherwise, we need to create a new tower for item (k, v). We insert (k, v) immediately after position p within S0 . After inserting the new item at the bottom level, we use randomization to decide the height of the tower for the new item. We ··ip· a coin, and if the ·ip comes up tails, then we stop here. Else (the ·ip comes up heads), we backtrack to the previous (next higher) level and insert (k, v) in this level at the appropriate position. We again ·ip a coin; if it comes up heads, we go to the next higher level and repeat. Thus, we continue to insert the new item (k, v) in lists until we ·nally get a ·ip that comes up tails. We link together all the references to the new item (k, v) created in this process to create its tower. A coin ·ip can be simulated with Python·s built-in pseudo-random number generator from the random module by calling randrange(2), which returns 0 or 1, each with probability 1/2.

We give the insertion algorithm for a skip list S in Code Fragment 10.13 and we illustrate it in Figure 10.12. The algorithm uses an insertAfterAbove(p, q, (k, v)) method that inserts a position storing the item (k, v) after position p (on the same level as p) and above position q, returning the new position r (and setting internal references so that next, prev, above, and below methods will work correctly for p, q, and r). The expected running time of the insertion algorithm on a skip list with n entries is O(log n), which we show in Section 10.4.2.

C-15.11 Describe an external-memory data structure to implement the queue ADT so that the total number of disk transfers needed to process a sequence of k enqueue and dequeue operations is O(k/B).

C-15.12 Describe an external-memory version of the PositionalList ADT (Section 7.4), with block size B, such that an iteration of a list of length n is completed using O(n/B) transfers in the worst case, and all other methods of the ADT require only O(1) transfers.

C-15.13 Change the rules that de·ne red-black trees so that each red-black tree T has a corresponding (4, 8) tree, and vice versa.

C-15.14 Describe a modi·ed version of the B-tree insertion algorithm so that each time we create an over·ow because of a split of a node w, we redistribute keys among all of w·s siblings, so that each sibling holds roughly the same number of keys (possibly cascading the split up to the parent of w). What is the minimum fraction of each block that will always be ·lled using this scheme?

C-15.15 Another possible external-memory map implementation is to use a skip list, but to collect consecutive groups of O(B) nodes, in individual blocks, on any level in the skip list. In particular, we de·ne an order-d B-skip list to be such a representation of a skip list structure, where each block contains at least d/2 list nodes and at most d list nodes. Let us also choose d in this case to be the maximum number of list nodes from a level of a skip list that can ·t into one block. Describe how we should modify the skip-list insertion and removal algorithms for a B-skip list so that the expected height of the structure is O(log n/ log B).

C-15.16 Describe how to use a B-tree to implement the partition (union-·nd) ADT (from Section 14.7.3) so that the union and ·nd operations each use at most O(log n/ log B) disk transfers.

C-15.17 Suppose we are given a sequence S of n elements with integer keys such that some elements in S are colored ·blue· and some elements in S are colored ·red.· In addition, say that a red element e pairs with a blue element f if they have the same key value. Describe an ef·cient external-memory algorithm for ·nding all the red-blue pairs in S. How many disk transfers does your algorithm perform?

C-15.18 Consider the page caching problem where the memory cache can hold m pages, and we are given a sequence P of n requests taken from a pool of m + 1 possible pages. Describe the optimal strategy for the of·ine algorithm and show that it causes at most m + n/m page misses in total, starting from an empty cache.

C-15.19 Describe an ef·cient external-memory algorithm that determines whether an array of n integers contains a value occurring more than n/2 times.

C-10.49 Python·s collections module provides an OrderedDict class that is unrelated to our sorted map abstraction. An OrderedDict is a subclass of the standard hash-based dict class that retains the expected O(1) performance for the primary map operations, but that also guarantees that the iter method reports items of the map according to ·rst-in, ·rst-out (FIFO) order. That is, the key that has been in the dictionary the longest is reported ·rst. (The order is unaffected when the value for an existing key is overwritten.) Describe an algorithmic approach for achieving such performance.

Projects

P-10.50 Perform a comparative analysis that studies the collision rates for various hash codes for character strings, such as various polynomial hash codes for different values of the parameter a. Use a hash table to determine collisions, but only count collisions where different strings map to the same hash code (not if they map to the same location in this hash table). Test these hash codes on text ·les found on the Internet.

P-10.51 Perform a comparative analysis as in the previous exercise, but for 10-digit telephone numbers instead of character strings.

P-10.52 Implement an OrderedDict class, as described in Exercise C-10.49, ensuring that the primary map operations run in O(1) expected time.

P-10.53 Design a Python class that implements the skip-list data structure. Use this class to create a complete implementation of the sorted map ADT.

P-10.54 Extend the previous project by providing a graphical animation of the skip-list operations. Visualize how entries move up the skip list during insertions and are linked out of the skip list during removals. Also, in a search operation, visualize the scan-forward and drop-down actions.

P-10.55 Write a spell-checker class that stores a lexicon of words, W , in a Python set, and implements a method, check(s), which performs a spell check on the string s with respect to the set of words, W . If s is in W , then the call to check(s) returns a list containing only s, as it is assumed to be spelled correctly in this case. If s is not in W , then the call to check(s) returns a list of every word in W that might be a correct spelling of s. Your program should be able to handle all the common ways that s might be a misspelling of a word in W , including swapping adjacent characters in a word, inserting a single character in between two adjacent characters in a word, deleting a single character from a word, and replacing a character in a word with another character. For an extra challenge, consider phonetic substitutions as well.

R-10.12 What is the result of Exercise R-10.9 when collisions are handled by double hashing using the secondary hash function h (k) = 7 · (k mod 7)?

R-10.13 What is the worst-case time for putting n entries in an initially empty hash table, with collisions resolved by chaining? What is the best case?

R-10.14 Show the result of rehashing the hash table shown in Figure 10.6 into a table of size 19 using the new hash function h(k) = 3k mod 17.

R-10.15 Our HashMapBase class maintains a load factor · · 0.5. Reimplement that class to allow the user to specify the maximum load, and adjust the concrete subclasses accordingly.

R-10.16 Give a pseudo-code description of an insertion into a hash table that uses quadratic probing to resolve collisions, assuming we also use the trick of replacing deleted entries with a special ·deactivated entry· object.

R-10.17 Modify our ProbeHashMap to use quadratic probing.

R-10.18 Explain why a hash table is not suited to implement a sorted map.

R-10.19 Describe how a sorted list implemented as a doubly linked list could be used to implement the sorted map ADT.

R-10.20 What is the worst-case asymptotic running time for performing n deletions from a SortedTableMap instance that initially contains 2n entries?

R-10.21 Consider the following variant of the ·nd index method from Code Fragment 10.8, in the context of the SortedTableMap class:

```
def ·nd index(self, k, low, high):
  if high < low:
    return high + 1
  else:
    mid = (low + high) // 2
    if self. table[mid]. key < k:
      return self. ·nd index(k, mid + 1, high)
    else:
      return self. ·nd index(k, low, mid · 1)
```

Does this always produce the same result as the original version? Justify your answer.

R-10.22 What is the expected running time of the methods for maintaining a maxima set if we insert n pairs such that each pair has lower cost and performance than one before it? What is contained in the sorted map at the end of this series of operations? What if each pair had a lower cost and higher performance than the one before it?

R-10.23 Draw an example skip list S that results from performing the following series of operations on the skip list shown in Figure 10.13: del S[38], S[48] = x , S[24] = y , del S[55]. Record your coin ·ips, as well.

Bounding the Height of a Skip List

Because the insertion step involves randomization, a more accurate analysis of skip lists involves a bit of probability. At ·rst, this might seem like a major undertaking, for a complete and thorough probabilistic analysis could require deep mathematics (and, indeed, there are several such deep analyses that have appeared in data structures research literature). Fortunately, such an analysis is not necessary to understand the expected asymptotic behavior of skip lists. The informal and intuitive probabilistic analysis we give below uses only basic concepts of probability theory.

Let us begin by determining the expected value of the height h of a skip list S with n entries (assuming that we do not terminate insertions early). The probability that a given entry has a tower of height i · 1 is equal to the probability of getting i consecutive heads when ·ipping a coin, that is, this probability is 1/2i . Hence, the probability Pi that level i has at least one position is at most

$$P_i \cdot \frac{n}{2^i},$$

for the probability that any one of n different events occurs is at most the sum of the probabilities that each occurs.

The probability that the height h of S is larger than i is equal to the probability that level i has at least one position, that is, it is no more than Pi . This means that h is larger than, say, 3 log n with probability at most

$$P_{3 \log n} \cdot \frac{n}{2^{3 \log n}} = \frac{n}{n^3} = \frac{1}{n^2}.$$

For example, if n = 1000, this probability is a one-in-a-million long shot. More generally, given a constant c > 1, h is larger than c log n with probability at most 1/nc·1 . That is, the probability that h is smaller than c log n is at least 1 · 1/nc·1 . Thus, with high probability, the height h of S is O(log n).

Analyzing Search Time in a Skip List

Next, consider the running time of a search in skip list S, and recall that such a search involves two nested while loops. The inner loop performs a scan forward on a level of S as long as the next key is no greater than the search key k, and the outer loop drops down to the next level and repeats the scan forward iteration. Since the height h of S is O(log n) with high probability, the number of drop-down steps is O(log n) with high probability.

Maintaining the Topmost Level

A skip list S must maintain a reference to the start position (the topmost, left position in S) as an instance variable, and must have a policy for any insertion that wishes to continue inserting a new entry past the top level of S. There are two possible courses of action we can take, both of which have their merits.

One possibility is to restrict the top level, h, to be kept at some ·xed value that is a function of n, the number of entries currently in the map (from the analysis we will see that h = max{10, 2 log n } is a reasonable choice, and picking h = 3 log n is even safer). Implementing this choice means that we must modify the insertion algorithm to stop inserting a new position once we reach the topmost level (unless $\log n < \log(n+1)$, in which case we can now go at least one more level, since the bound on the height is increasing).

The other possibility is to let an insertion continue inserting a new position as long as heads keeps getting returned from the random number generator. This is the approach taken by algorithm SkipInsert of Code Fragment 10.13. As we show in the analysis of skip lists, the probability that an insertion will go to a level that is more than O(log n) is very low, so this design choice should also work.

Either choice will still result in the expected O(log n) time to perform search, insertion, and removal, however, which we show in the next section.

10.4.2 Probabilistic Analysis of Skip Lists

As we have shown above, skip lists provide a simple implementation of a sorted map. In terms of worst-case performance, however, skip lists are not a superior data structure. In fact, if we do not of·cially prevent an insertion from continuing significantly past the current highest level, then the insertion algorithm can go into what is almost an in·nite loop (it is not actually an in·nite loop, however, since the probability of having a fair coin repeatedly come up heads forever is 0). Moreover, we cannot in·nitely add positions to a list without eventually running out of memory. In any case, if we terminate position insertion at the highest level h, then the worst-case running time for performing the getitem , setitem , and delitem map operations in a skip list S with n entries and height h is O(n + h). This worst-case performance occurs when the tower of every entry reaches level h · 1, where h is the height of S. However, this event has very low probability. Judging from this worst case, we might conclude that the skip-list structure is strictly inferior to the other map implementations discussed earlier in this chapter. But this would not be a fair analysis, for this worst-case behavior is a gross overestimate.

So we have yet to bound the number of scan-forward steps we make. Let ni be the number of keys examined while scanning forward at level i. Observe that, after the key at the starting position, each additional key examined in a scan-forward at level i cannot also belong to level i + 1. If any of these keys were on the previous level, we would have encountered them in the previous scan-forward step. Thus, the probability that any key is counted in ni is 1/2. Therefore, the expected value of ni is exactly equal to the expected number of times we must ·ip a fair coin before it comes up heads. This expected value is 2. Hence, the expected amount of time spent scanning forward at any level i is O(1). Since S has O(log n) levels with high probability, a search in S takes expected time O(log n). By a similar analysis, we can show that the expected running time of an insertion or a removal is O(log n).

Space Usage in a Skip List

Finally, let us turn to the space requirement of a skip list S with n entries. As we observed above, the expected number of positions at level i is n/2i , which means that the expected total number of positions in S is

$$\sum_{i=0}^{h} n \cdot 2i = n \cdot \sum_{i=0}^{h} 2i .$$

Using Proposition 3.5 on geometric summations, we have

$$\sum_{i=0}^{h} 2 \cdot i = 1 \cdot \frac{1 - 2^{h+1}}{1 - 2} = 2 \cdot 1 \cdot 2^{h+1} < 2 \text{ for all } h \cdot 0.$$

Hence, the expected space requirement of S is O(n).

Table 10.4 summarizes the performance of a sorted map realized by a skip list.

Operation	Running Time
len(M)	O(1)
k in M	O(log n) expected
M[k] = v	O(log n) expected
del M[k]	O(log n) expected
M.·nd min( ), M.·nd max( )	O(1)
M.·nd lt(k), M.·nd gt(k)	O(log n) expected
M.·nd le(k), M.·nd ge(k)	O(log n) expected
M.·nd range(start, stop)	O(s + log n) expected, with s items reported
iter(M), reversed(M)	O(n)

Table 10.4: Performance of a sorted map implemented with a skip list. We use n to denote the number of entries in the dictionary at the time the operation is performed. The expected space requirement is O(n).