if this list were maintained as a priority queue (Chapter 9). In each algorithm, the requested amount of memory is subtracted from the chosen memory hole and the leftover part of that hole is returned to the free list.

Although it might sound good at first, the best-fit algorithm tends to produce the worst external fragmentation, since the leftover parts of the chosen holes tend to be small. The first-fit algorithm is fast, but it tends to produce a lot of external fragmentation at the front of the free list, which slows down future searches. The next-fit algorithm spreads fragmentation more evenly throughout the memory heap, thus keeping search times low. This spreading also makes it more difficult to allocate large blocks, however. The worst-fit algorithm attempts to avoid this problem by keeping contiguous sections of free memory as large as possible.

15.1.2 Garbage Collection

In some languages, like C and C++, the memory space for objects must be explicitly deallocated by the programmer, which is a duty often overlooked by beginning programmers and is the source of frustrating programming errors even for experienced programmers. The designers of Python instead placed the burden of memory management entirely on the interpreter. The process of detecting "stale" objects, deallocating the space devoted to those objects, and returning the reclaimed space to the free list is known as ***garbage collection***.

To perform automated garbage collection, there must first be a way to detect those objects that are no longer necessary. Since the interpreter cannot feasibly analyze the semantics of an arbitrary Python program, it relies on the following conservative rule for reclaiming objects. In order for a program to access an object, it must have a direct or indirect reference to that object. We will define such objects to be ***live objects***. In defining a live object, a ***direct reference*** to an object is in the form of an identifier in an active namespace (i.e., the global namespace, or the local namespace for any active function). For example, immediately after the command `w = Widget()` is executed, identifier `w` will be defined in the current namespace as a reference to the new widget object. We refer to all such objects with direct references as ***root objects***. An ***indirect reference*** to a live object is a reference that occurs within the state of some other live object. For example, if the widget instance in our earlier example maintains a list as an attribute, that list is also a live object (as it can be reached indirectly through use of identifier `w`). The set of live objects are defined recursively; thus, any objects that are referenced within the list that is referenced by the widget are also classified as live objects.

The Python interpreter assumes that live objects are the active objects currently being used by the running program; these objects should ***not*** be deallocated. Other objects can be garbage collected. Python relies on the following two strategies for determining which objects are live.

15.1.1 Memory Allocation

With Python, all objects are stored in a pool of memory, known as the ***memory heap*** or ***Python heap*** (which should not be confused with the "heap" data structure presented in Chapter 9). When a command such as

```
w = Widget( )
```

is executed, assuming Widget is the name of a class, a new instance of the class is created and stored somewhere within the memory heap. The Python interpreter is responsible for negotiating the use of space with the operating system and for managing the use of the memory heap when executing a Python program.

The storage available in the memory heap is divided into ***blocks***, which are contiguous array-like "chunks" of memory that may be of variable or fixed sizes. The system must be implemented so that it can quickly allocate memory for new objects. One popular method is to keep contiguous "holes" of available free memory in a linked list, called the ***free list***. The links joining these holes are stored inside the holes themselves, since their memory is not being used. As memory is allocated and deallocated, the collection of holes in the free lists changes, with the unused memory being separated into disjoint holes divided by blocks of used memory. This separation of unused memory into separate holes is known as ***fragmentation***. The problem is that it becomes more difficult to find large continuous chunks of memory, when needed, even though an equivalent amount of memory may be unused (yet fragmented). Therefore, we would like to minimize fragmentation as much as possible.

There are two kinds of fragmentation that can occur. ***Internal fragmentation*** occurs when a portion of an allocated memory block is unused. For example, a program may request an array of size 1000, but only use the first 100 cells of this array. There is not much that a run-time environment can do to reduce internal fragmentation. ***External fragmentation***, on the other hand, occurs when there is a significant amount of unused memory between several contiguous blocks of allocated memory. Since the run-time environment has control over where to allocate memory when it is requested, the run-time environment should allocate memory in a way to try to reduce external fragmentation as much as reasonably possible.

Several heuristics have been suggested for allocating memory from the heap so as to minimize external fragmentation. The ***best-fit algorithm*** searches the entire free list to find the hole whose size is closest to the amount of memory being requested. The ***first-fit algorithm*** searches from the beginning of the free list for the first hole that is large enough. The ***next-fit algorithm*** is similar, in that it also searches the free list for the first hole that is large enough, but it begins its search from where it left off previously, viewing the free list as a circularly linked list (Section 7.2). The ***worst-fit algorithm*** searches the free list to find the largest hole of available memory, which might be done faster than a search of the entire free list

About the Authors

Michael Goodrich received his Ph.D. in Computer Science from Purdue University in 1987. He is currently a Chancellor's Professor in the Department of Computer Science at University of California, Irvine. Previously, he was a professor at Johns Hopkins University. He is a Fulbright Scholar and a Fellow of the American Association for the Advancement of Science (AAAS), Association for Computing Machinery (ACM), and Institute of Electrical and Electronics Engineers (IEEE). He is a recipient of the IEEE Computer Society Technical Achievement Award, the ACM Recognition of Service Award, and the Pond Award for Excellence in Undergraduate Teaching.

Roberto Tamassia received his Ph.D. in Electrical and Computer Engineering from the University of Illinois at Urbana-Champaign in 1988. He is the Plastech Professor of Computer Science and the Chair of the Department of Computer Science at Brown University. He is also the Director of Brown's Center for Geometric Computing. His research interests include information security, cryptography, analysis, design, and implementation of algorithms, graph drawing and computational geometry. He is a Fellow of the American Association for the Advancement of Science (AAAS), Association for Computing Machinery (ACM) and Institute for Electrical and Electronic Engineers (IEEE). He is also a recipient of the Technical Achievement Award from the IEEE Computer Society.

Michael Goldwasser received his Ph.D. in Computer Science from Stanford University in 1997. He is currently a Professor in the Department of Mathematics and Computer Science at Saint Louis University and the Director of their Computer Science program. Previously, he was a faculty member in the Department of Computer Science at Loyola University Chicago. His research interests focus on the design and implementation of algorithms, having published work involving approximation algorithms, online computation, computational biology, and computational geometry. He is also active in the computer science education community.

Additional Books by These Authors

- M.T. Goodrich and R. Tamassia, *Data Structures and Algorithms in Java*, Wiley.
- M.T. Goodrich, R. Tamassia, and D.M. Mount, *Data Structures and Algorithms in C++*, Wiley.
- M.T. Goodrich and R. Tamassia, *Algorithm Design: Foundations, Analysis, and Internet Examples*, Wiley.
- M.T. Goodrich and R. Tamassia, *Introduction to Computer Security*, Addison-Wesley.
- M.H. Goldwasser and D. Letscher, *Object-Oriented Programming in Python*, Prentice Hall.

prune-and-search, 571–573
pseudo-code, 64
pseudo-random number generator, 49–50, 438
Pugh, William, 458
puzzle solver, 175–176
Python heap, 699
Python interpreter, 2
Python interpreter stack, 703

quadratic function, 117
quadratic probing, 419
queue, 239
  abstract data type, 240
  array implementation, 241–246
  linked-list implementation, 264–265
quick-sort, 550–561

radix-sort, 565–566
Raghavan, Prabhakar, 458, 580
raise statement, **34–35**, 38
Ramachandran, Vijaya, 361
random access memory (RAM), 185
Random class, 50
random module, 49, **49–50**, 225, 438
random seed, 50
random variable, 729
randomization, 438
randomized quick-select, 572
randomized quick-sort, 557
randrange function, 50, 51, 225
range class, 22, 27, 29, 80–81
re module, 49
reachability, 623, 638
recurrence equation, 162, 546, 573, 576
recursion, 149–179, 703–704
  binary, 174
  depth limit, 168, 528
  linear, 169–173
  multiple, 175–176
  tail, 178–179
  trace, 151, 161, 703
red-black tree, 512–525
  depth property, 512
  recoloring, 516
  red property, 512
  root property, 512
Reed, Bruce, 400
reference, 187
reference count, 209, 701
reflexive property, 385, 537
rehashing, 420
reserved words, 4
return statement, 24
reusability, 57, 58
reversed function, 29
Ribeiro-Neto, Berthier, 618
Rivest, Ronald, 535, 696
Robson, David, 298
robustness, 57
root objects, 700
root of a tree, 301
rotation, 475
round function, 29
round-robin, 267
running time, 110

Samet, Hanan, 719
Schaffer, Russel, 400
scheduling, 400
scope, **46–47**, 96, 98, 701
  global, 46, 96
  local, 23–25, 46, 96
Scoreboard class, 211–213
script, 2
search engine, 612
search table, 428–433
search tree, 460–535
Sedgewick, Robert, 400, 535
seed, 50, 438
selection, 571–573
selection-sort, 386–387
self identifier, 69
self-loop, 622
sentinel, 270–271
separate chaining, 417
sequential search, 155
set class, 7, 11, **446**
set comprehension, 43
shallow copy, 101, 188
Sharir, Micha, 361
short-circuit evaluation, 12, 20, 208
shortest path, 660–669
  Dijkstra's algorithm, 661–669

Appendix A
Character Strings in Python

A string is a sequence of characters that come from some ***alphabet***. In Python, the built-in **str** class represents strings based upon the Unicode international character set, a 16-bit character encoding that covers most written languages. Unicode is an extension of the 7-bit ASCII character set that includes the basic Latin alphabet, numerals, and common symbols. Strings are particularly important in most programming applications, as text is often used for input and output.

A basic introduction to the str class was provided in Section 1.2.3, including use of string literals, such as `'hello'`, and the syntax str(obj) that is used to construct a string representation of a typical object. Common operators that are supported by strings, such as the use of + for concatenation, were further discussed in Section 1.3. This appendix serves as a more detailed reference, describing convenient behaviors that strings support for the processing of text. To organize our overview of the str class behaviors, we group them into the following broad categories of functionality.

Searching for Substrings

The operator syntax, pattern **in** s, can be used to determine if the given pattern occurs as a substring of string s. Table A.1 describes several related methods that determine the number of such occurrences, and the index at which the leftmost or rightmost such occurrence begins. Each of the functions in this table accepts two optional parameters, start and end, which are indices that effectively restrict the search to the implicit slice s[start:end]. For example, the call s.find(pattern, 5) restricts the search to s[5:].

Calling Syntax	Description
s.count(pattern)	Return the number of non-overlapping occurrences of pattern
s.find(pattern)	Return the index starting the leftmost occurrence of pattern; else -1
s.index(pattern)	Similar to find, but raise ValueError if not found
s.rfind(pattern)	Return the index starting the rightmost occurrence of pattern; else -1
s.rindex(pattern)	Similar to rfind, but raise ValueError if not found

Table A.1: Methods that search for substrings.

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 define red-black trees so that each red-black tree T has a corresponding $(4,8)$ tree, and vice versa.

C-15.14 Describe a modified version of the B-tree insertion algorithm so that each time we create an overflow 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 filled 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 define an ***order-d B-skip list*** to be such a representation of a skip list structure, where each block contains at least $\lceil d/2 \rceil$ 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 fit 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-find) ADT (from Section 14.7.3) so that the union and find 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 efficient external-memory algorithm for finding 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 offline 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 efficient external-memory algorithm that determines whether an array of n integers contains a value occurring more than $n/2$ times.

15.4.1 Multiway Merging

In a standard merge-sort (Section 12.2), the merge process combines two sorted sequences into one by repeatedly taking the smaller of the items at the front of the two respective lists. In a d-way merge, we repeatedly find the smallest among the items at the front of the d sequences and place it as the next element of the merged sequence. We continue until all elements are included.

In the context of an external-memory sorting algorithm, if main memory has size M and each block has size B, we can store up to M/B blocks within main memory at any given time. We specifically choose $d = (M/B) - 1$ so that we can afford to keep one block from each input sequence in main memory at any given time, and to have one additional block to use as a buffer for the merged sequence. (See Figure 15.5.)

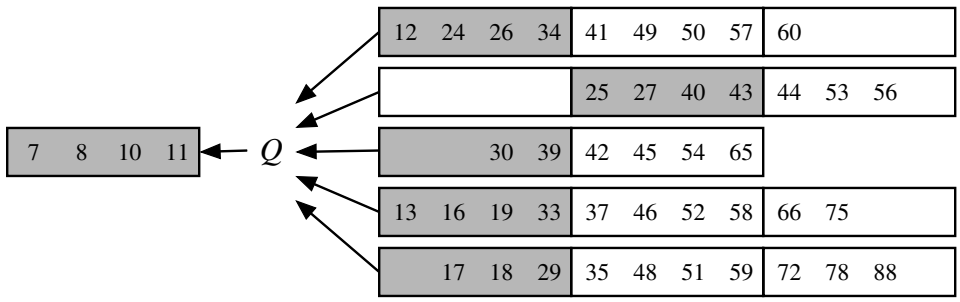

Figure 15.5: A d-way merge with $d = 5$ and $B = 4$. Blocks that currently reside in main memory are shaded.

We maintain the smallest unprocessed element from each input sequence in main memory, requesting the next block from a sequence when the preceding block has been exhausted. Similarly, we use one block of internal memory to buffer the merged sequence, flushing that block to external memory when full. In this way, the total number of transfers performed during a single d-way merge is $O(n/B)$, since we scan each block of list S_i once, and we write out each block of the merged list S' once. In terms of computation time, choosing the smallest of d values can trivially be performed using $O(d)$ operations. If we are willing to devote $O(d)$ internal memory, we can maintain a priority queue identifying the smallest element from each sequence, thereby performing each step of the merge in $O(\log d)$ time by removing the minimum element and replacing it with the next element from the same sequence. Hence, the internal time for the d-way merge is $O(n \log d)$.

Proposition 15.3: *Given an array-based sequence S of n elements stored compactly in external memory, we can sort S with $O((n/B)\log(n/B)/\log(M/B))$ block transfers and $O(n \log n)$ internal computations, where M is the size of the internal memory and B is the size of a block.*

15.3.2 B-Trees

A version of the (a,b) tree data structure, which is the best-known method for maintaining a map in external memory, is called the "B-tree." (See Figure 15.4.) A ***B-tree of order*** d is an (a,b) tree with $a = \lceil d/2 \rceil$ and $b = d$. Since we discussed the standard map query and update methods for (a,b) trees above, we restrict our discussion here to the I/O complexity of B-trees.

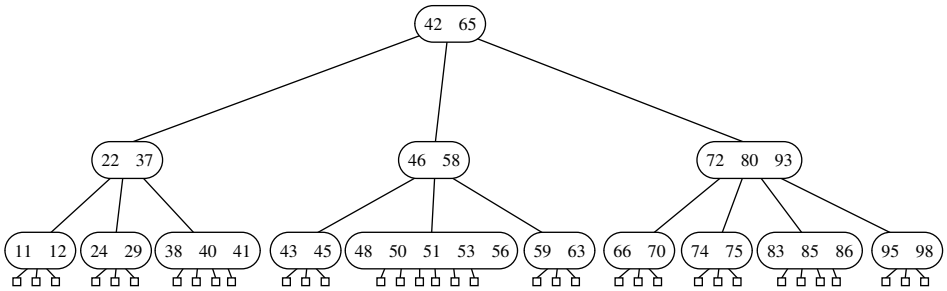

Figure 15.4: A B-tree of order 6.

An important property of B-trees is that we can choose d so that the d children references and the $d-1$ keys stored at a node can fit compactly into a single disk block, implying that d is proportional to B. This choice allows us to assume that a and b are also proportional to B in the analysis of the search and update operations on (a,b) trees. Thus, $f(b)$ and $g(b)$ are both $O(1)$, for each time we access a node to perform a search or an update operation, we need only perform a single disk transfer.

As we have already observed above, each search or update requires that we examine at most $O(1)$ nodes for each level of the tree. Therefore, any map search or update operation on a B-tree requires only $O(\log_{\lceil d/2 \rceil} n)$, that is, $O(\log n / \log B)$, disk transfers. For example, an insert operation proceeds down the B-tree to locate the node in which to insert the new entry. If the node would ***overflow*** (to have $d+1$ children) because of this addition, then this node is ***split*** into two nodes that have $\lfloor (d+1)/2 \rfloor$ and $\lceil (d+1)/2 \rceil$ children, respectively. This process is then repeated at the next level up, and will continue for at most $O(\log_B n)$ levels.

Likewise, if a remove operation results in a node ***underflow*** (to have $\lceil d/2 \rceil - 1$ children), then we move references from a sibling node with at least $\lceil d/2 \rceil + 1$ children or we perform a ***fusion*** operation of this node with its sibling (and repeat this computation at the parent). As with the insert operation, this will continue up the B-tree for at most $O(\log_B n)$ levels. The requirement that each internal node have at least $\lceil d/2 \rceil$ children implies that each disk block used to support a B-tree is at least half full. Thus, we have the following:

Proposition 15.2: *A B-tree with n entries has I/O complexity $O(\log_B n)$ for search or update operation, and uses $O(n/B)$ blocks, where B is the size of a block.*