Using an Array Circularly

In developing a more robust queue implementation, we allow the front of the queue to drift rightward, and we allow the contents of the queue to "wrap around" the end of an underlying array. We assume that our underlying array has fixed length N that is greater that the actual number of elements in the queue. New elements are enqueued toward the "end" of the current queue, progressing from the front to index $N-1$ and continuing at index 0, then 1. Figure 6.6 illustrates such a queue with first element E and last element M.

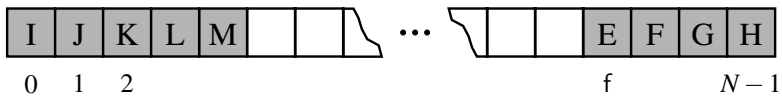


Figure 6.6: Modeling a queue with a circular array that wraps around the end.

Implementing this circular view is not difficult. When we dequeue an element and want to "advance" the front index, we use the arithmetic f = (f + 1) % N. Recall that the % operator in Python denotes the ***modulo*** operator, which is computed by taking the remainder after an integral division. For example, 14 divided by 3 has a quotient of 4 with remainder 2, that is, $\frac{14}{3} = 4\frac{2}{3}$. So in Python, 14 // 3 evaluates to the quotient 4, while 14 % 3 evaluates to the remainder 2. The modulo operator is ideal for treating an array circularly. As a concrete example, if we have a list of length 10, and a front index 7, we can advance the front by formally computing (7+1) % 10, which is simply 8, as 8 divided by 10 is 0 with a remainder of 8. Similarly, advancing index 8 results in index 9. But when we advance from index 9 (the last one in the array), we compute (9+1) % 10, which evaluates to index 0 (as 10 divided by 10 has a remainder of zero).

A Python Queue Implementation

A complete implementation of a queue ADT using a Python list in circular fashion is presented in Code Fragments 6.6 and 6.7. Internally, the queue class maintains the following three instance variables:

_data: is a reference to a list instance with a fixed capacity.

_size: is an integer representing the current number of elements stored in the queue (as opposed to the length of the _data list).

_front: is an integer that represents the index within _data of the first element of the queue (assuming the queue is not empty).

We initially reserve a list of moderate size for storing data, although the queue formally has size zero. As a technicality, we initialize the _front index to zero.

When front or dequeue are called with no elements in the queue, we raise an instance of the Empty exception, defined in Code Fragment 6.1 for our stack.

If we must draw a line between efficient and inefficient algorithms, therefore, it is natural to make this distinction be that between those algorithms running in polynomial time and those running in exponential time. That is, make the distinction between algorithms with a running time that is $O(n^c)$, for some constant $c > 1$, and those with a running time that is $O(b^n)$, for some constant $b > 1$. Like so many notions we have discussed in this section, this too should be taken with a "grain of salt," for an algorithm running in $O(n^{100})$ time should probably not be considered "efficient." Even so, the distinction between polynomial-time and exponential-time algorithms is considered a robust measure of tractability.

3.3.3 Examples of Algorithm Analysis

Now that we have the big-Oh notation for doing algorithm analysis, let us give some examples by characterizing the running time of some simple algorithms using this notation. Moreover, in keeping with our earlier promise, we illustrate below how each of the seven functions given earlier in this chapter can be used to characterize the running time of an example algorithm.

Rather than use pseudo-code in this section, we give complete Python implementations for our examples. We use Python's **list** class as the natural representation for an "array" of values. In Chapter 5, we will fully explore the underpinnings of Python's list class, and the efficiency of the various behaviors that it supports. In this section, we rely on just a few of its behaviors, discussing their efficiencies as introduced.

Constant-Time Operations

Given an instance, named `data`, of the Python list class, a call to the function, `len(data)`, is evaluated in constant time. This is a very simple algorithm because the list class maintains, for each list, an instance variable that records the current length of the list. This allows it to immediately report that length, rather than take time to iteratively count each of the elements in the list. Using asymptotic notation, we say that this function runs in $O(1)$ time; that is, the running time of this function is independent of the length, n, of the list.

Another central behavior of Python's list class is that it allows access to an arbitrary element of the list using syntax, `data[j]`, for integer index j. Because Python's lists are implemented as ***array-based sequences***, references to a list's elements are stored in a consecutive block of memory. The j^{th} element of the list can be found, not by iterating through the list one element at a time, but by validating the index, and using it as an offset into the underlying array. In turn, computer hardware supports constant-time access to an element based on its memory address. Therefore, we say that the expression `data[j]` is evaluated in $O(1)$ time for a Python list.

Index-Based For Loops

The simplicity of a standard for loop over the elements of a list is wonderful; however, one limitation of that form is that we do not know where an element resides within the sequence. In some applications, we need knowledge of the index of an element within the sequence. For example, suppose that we want to know *where* the maximum element in a list resides.

Rather than directly looping over the elements of the list in that case, we prefer to loop over all possible indices of the list. For this purpose, Python provides a built-in class named range that generates integer sequences. (We will discuss generators in Section 1.8.) In simplest form, the syntax range(n) generates the series of n values from 0 to $n-1$. Conveniently, these are precisely the series of valid indices into a sequence of length n. Therefore, a standard Python idiom for looping through the series of indices of a data sequence uses a syntax,

```
for j in range(len(data)):
```

In this case, identifier j is not an element of the data—it is an integer. But the expression data[j] can be used to retrieve the respective element. For example, we can find the *index* of the maximum element of a list as follows:

```
big_index = 0
for j in range(len(data)):
  if data[j] > data[big_index]:
    big_index = j
```

Break and Continue Statements

Python supports a **break** statement that immediately terminate a while or for loop when executed within its body. More formally, if applied within nested control structures, it causes the termination of the most immediately enclosing loop. As a typical example, here is code that determines whether a target value occurs in a data set:

```
found = False
for item in data:
  if item == target:
    found = True
    break
```

Python also supports a **continue** statement that causes the current *iteration* of a loop body to stop, but with subsequent passes of the loop proceeding as expected.

We recommend that the break and continue statements be used sparingly. Yet, there are situations in which these commands can be effectively used to avoid introducing overly complex logical conditions.

1.8 Iterators and Generators

In Section 1.4.2, we introduced the for-loop syntax beginning as:

for *element* **in** *iterable*:

and we noted that there are many types of objects in Python that qualify as being iterable. Basic container types, such as list, tuple, and set, qualify as iterable types. Furthermore, a string can produce an iteration of its characters, a dictionary can produce an iteration of its keys, and a file can produce an iteration of its lines. User-defined types may also support iteration. In Python, the mechanism for iteration is based upon the following conventions:

- An ***iterator*** is an object that manages an iteration through a series of values. If variable, i, identifies an iterator object, then each call to the built-in function, next(i), produces a subsequent element from the underlying series, with a StopIteration exception raised to indicate that there are no further elements.
- An ***iterable*** is an object, obj, that produces an *iterator* via the syntax iter(obj).

By these definitions, an instance of a list is an iterable, but not itself an iterator. With data = [1, 2, 4, 8], it is not legal to call next(data). However, an iterator object can be produced with syntax, i = iter(data), and then each subsequent call to next(i) will return an element of that list. The for-loop syntax in Python simply automates this process, creating an iterator for the give iterable, and then repeatedly calling for the next element until catching the StopIteration exception.

More generally, it is possible to create multiple iterators based upon the same iterable object, with each iterator maintaining its own state of progress. However, iterators typically maintain their state with indirect reference back to the original collection of elements. For example, calling iter(data) on a list instance produces an instance of the list_iterator class. That iterator does not store its own copy of the list of elements. Instead, it maintains a current *index* into the original list, representing the next element to be reported. Therefore, if the contents of the original list are modified after the iterator is constructed, but before the iteration is complete, the iterator will be reporting the *updated* contents of the list.

Python also supports functions and classes that produce an implicit iterable series of values, that is, without constructing a data structure to store all of its values at once. For example, the call range(1000000) does *not* return a list of numbers; it returns a range object that is iterable. This object generates the million values one at a time, and only as needed. Such a ***lazy evaluation*** technique has great advantage. In the case of range, it allows a loop of the form, **for** j **in** range(1000000):, to execute without setting aside memory for storing one million values. Also, if such a loop were to be interrupted in some fashion, no time will have been spent computing unused values of the range.

5.2.1 Referential Arrays

As another motivating example, assume that we want a medical information system to keep track of the patients currently assigned to beds in a certain hospital. If we assume that the hospital has 200 beds, and conveniently that those beds are numbered from 0 to 199, we might consider using an array-based structure to maintain the names of the patients currently assigned to those beds. For example, in Python we might use a list of names, such as:

```
['Rene', 'Joseph', 'Janet', 'Jonas', 'Helen', 'Virginia', ... ]
```

To represent such a list with an array, Python must adhere to the requirement that each cell of the array use the same number of bytes. Yet the elements are strings, and strings naturally have different lengths. Python could attempt to reserve enough space for each cell to hold the *maximum* length string (not just of currently stored strings, but of any string we might ever want to store), but that would be wasteful.

Instead, Python represents a list or tuple instance using an internal storage mechanism of an array of object ***references***. At the lowest level, what is stored is a consecutive sequence of memory addresses at which the elements of the sequence reside. A high-level diagram of such a list is shown in Figure 5.4.

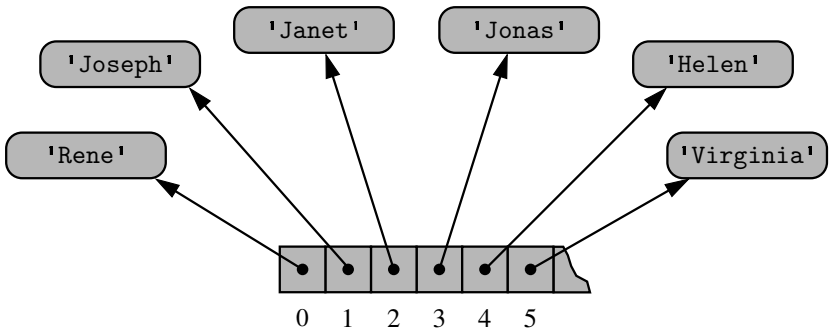


Figure 5.4: An array storing references to strings.

Although the relative size of the individual elements may vary, the number of bits used to store the memory address of each element is fixed (e.g., 64-bits per address). In this way, Python can support constant-time access to a list or tuple element based on its index.

In Figure 5.4, we characterize a list of strings that are the names of the patients in a hospital. It is more likely that a medical information system would manage more comprehensive information on each patient, perhaps represented as an instance of a Patient class. From the perspective of the list implementation, the same principle applies: The list will simply keep a sequence of references to those objects. Note as well that a reference to the **None** object can be used as an element of the list to represent an empty bed in the hospital.

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 efficient update operations, as long as the position within the list is identified. Unfortunately, we cannot perform fast searches on a standard linked list; for example, the binary search algorithm requires an efficient means for direct accessing an element of a sequence by index.

Skip lists provide a clever compromise to efficiently support search and update operations. A ***skip list*** S for a map M consists of a series of lists $\{S_0, S_1, \ldots, S_h\}$. Each list S_i stores a subset of the items of M sorted by increasing keys, plus items with two sentinel keys denoted $-\infty$ and $+\infty$, where $-\infty$ is smaller than every possible key that can be inserted in M and $+\infty$ is larger than every possible key that can be inserted in M. In addition, the lists in S satisfy the following:

- List S_0 contains every item of the map M (plus sentinels $-\infty$ and $+\infty$).
- For $i = 1, \ldots, h-1$, list S_i contains (in addition to $-\infty$ and $+\infty$) a randomly generated subset of the items in list S_{i-1}.
- List S_h contains only $-\infty$ and $+\infty$.

An example of a skip list is shown in Figure 10.10. It is customary to visualize a skip list S with list S_0 at the bottom and lists $S_1, \ldots, S_h$ above it. Also, we refer to h as the ***height*** of skip list S.

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

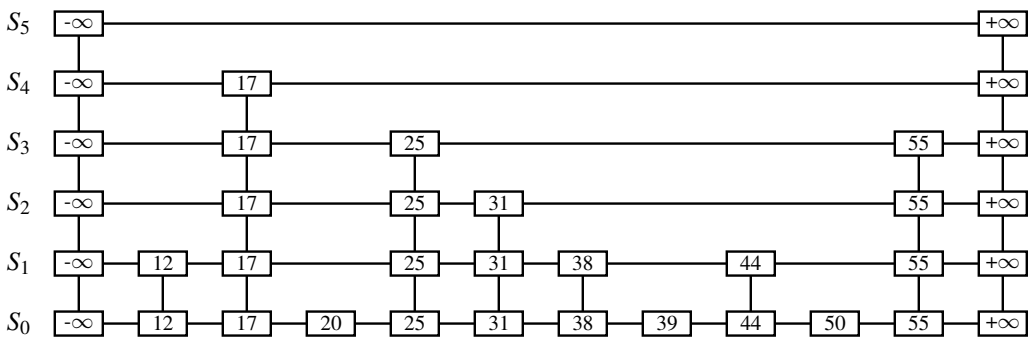


Figure 10.10: Example of a skip list storing 10 items. For simplicity, we show only the items' keys, not their associated values.

When the dequeue method is called, the current value of self._front designates the index of the value that is to be removed and returned. We keep a local reference to the element that will be returned, setting answer = self._data[self._front] just prior to removing the reference to that object from the list, with the assignment self._data[self._front] = None. Our reason for the assignment to None relates to Python's mechanism for reclaiming unused space. Internally, Python maintains a count of the number of references that exist to each object. If that count reaches zero, the object is effectively inaccessible, thus the system may reclaim that memory for future use. (For more details, see Section 15.1.2.) Since we are no longer responsible for storing a dequeued element, we remove the reference to it from our list so as to reduce that element's reference count.

The second significant responsibility of the dequeue method is to update the value of _front to reflect the removal of the element, and the presumed promotion of the second element to become the new first. In most cases, we simply want to increment the index by one, but because of the possibility of a wrap-around configuration, we rely on modular arithmetic as originally described on page 242.

Resizing the Queue

When enqueue is called at a time when the size of the queue equals the size of the underlying list, we rely on a standard technique of doubling the storage capacity of the underlying list. In this way, our approach is similar to the one used when we implemented a DynamicArray in Section 5.3.1.

However, more care is needed in the queue's _resize utility than was needed in the corresponding method of the DynamicArray class. After creating a temporary reference to the old list of values, we allocate a new list that is twice the size and copy references from the old list to the new list. While transferring the contents, we intentionally realign the front of the queue with index 0 in the new array, as shown in Figure 6.7. This realignment is not purely cosmetic. Since the modular arithmetic depends on the size of the array, our state would be flawed had we transferred each element to its same index in the new array.

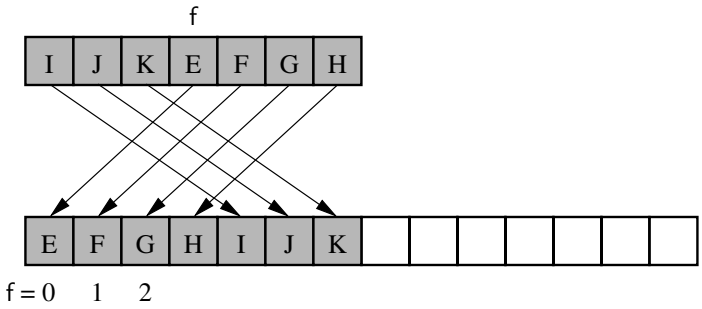


Figure 6.7: Resizing the queue, while realigning the front element with index 0.

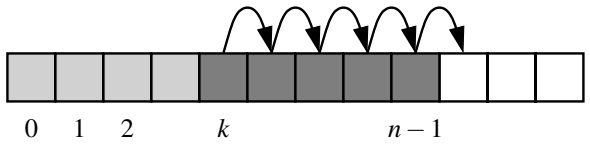


Figure 5.16: Creating room to insert a new element at index k of a dynamic array.

that process depends upon the index of the new element, and thus the number of other elements that must be shifted. That loop copies the reference that had been at index $n-1$ to index n, then the reference that had been at index $n-2$ to $n-1$, continuing until copying the reference that had been at index k to $k+1$, as illustrated in Figure 5.16. Overall this leads to an amortized $O(n-k+1)$ performance for inserting at index k.

When exploring the efficiency of Python's append method in Section 5.3.3, we performed an experiment that measured the average cost of repeated calls on varying sizes of lists (see Code Fragment 5.4 and Table 5.2). We have repeated that experiment with the insert method, trying three different access patterns:

- In the first case, we repeatedly insert at the beginning of a list,

```
for n in range(N):
  data.insert(0, None)
```

- In a second case, we repeatedly insert near the middle of a list,

```
for n in range(N):
  data.insert(n // 2, None)
```

- In a third case, we repeatedly insert at the end of the list,

```
for n in range(N):
  data.insert(n, None)
```

The results of our experiment are given in Table 5.5, reporting the *average* time per operation (not the total time for the entire loop). As expected, we see that inserting at the beginning of a list is most expensive, requiring linear time per operation. Inserting at the middle requires about half the time as inserting at the beginning, yet is still $\Omega(n)$ time. Inserting at the end displays $O(1)$ behavior, akin to append.

	N				
	100	1,000	10,000	100,000	1,000,000
$k = 0$	0.482	0.765	4.014	36.643	351.590
$k = n$ // 2	0.451	0.577	2.191	17.873	175.383
$k = n$	0.420	0.422	0.395	0.389	0.397

Table 5.5: Average running time of insert(k, val), measured in microseconds, as observed over a sequence of N calls, starting with an empty list. We let n denote the size of the current list (as opposed to the final list).

Python carefully extends the semantics of // and % to cases where one or both operands are negative. For the sake of notation, let us assume that variables n and m represent respectively the ***dividend*** and ***divisor*** of a quotient $\frac{n}{m}$, and that q = n // m and r = n % m. Python guarantees that q * m + r will equal n. We already saw an example of this identity with positive operands, as $6*4+3=27$. When the divisor m is positive, Python further guarantees that $0 \leq r < m$. As a consequence, we find that -27 // 4 evaluates to -7 and -27 % 4 evaluates to 1, as $(-7)*4+1=-27$. When the divisor is negative, Python guarantees that $m < r \leq 0$. As an example, 27 // -4 is -7 and 27 % -4 is -1, satisfying the identity $27=(-7)*(-4)+(-1)$.

The conventions for the // and % operators are even extended to floating-point operands, with the expression q = n // m being the integral floor of the quotient, and r = n % m being the "remainder" to ensure that q * m + r equals n. For example, 8.2 // 3.14 evaluates to 2.0 and 8.2 % 3.14 evaluates to 1.92, as $2.0*3.14+1.92=8.2$.

Bitwise Operators

Python provides the following bitwise operators for integers:

~	bitwise complement (prefix unary operator)
&	bitwise and
|	bitwise or
^	bitwise exclusive-or
<<	shift bits left, filling in with zeros
>>	shift bits right, filling in with sign bit

Sequence Operators

Each of Python's built-in sequence types (**str**, **tuple**, and **list**) support the following operator syntaxes:

s[j]	element at index *j*
s[start:stop]	slice including indices [start,stop)
s[start:stop:step]	slice including indices start, start + step, start + 2*step, ..., up to but not equalling or stop
s + t	concatenation of sequences
k * s	shorthand for s + s + s + ... (k times)
val **in** s	containment check
val **not in** s	non-containment check

Python relies on ***zero-indexing*** of sequences, thus a sequence of length n has elements indexed from 0 to $n-1$ inclusive. Python also supports the use of ***negative indices***, which denote a distance from the end of the sequence; index -1 denotes the last element, index -2 the second to last, and so on. Python uses a ***slicing***

Removing Elements from a List

Python's list class offers several ways to remove an element from a list. A call to pop( ) removes the last element from a list. This is most efficient, because all other elements remain in their original location. This is effectively an $O(1)$ operation, but the bound is amortized because Python will occasionally shrink the underlying dynamic array to conserve memory.

The parameterized version, pop(k), removes the element that is at index $k < n$ of a list, shifting all subsequent elements leftward to fill the gap that results from the removal. The efficiency of this operation is $O(n-k)$, as the amount of shifting depends upon the choice of index k, as illustrated in Figure 5.17. Note well that this implies that pop(0) is the most expensive call, using $\Omega(n)$ time. (see experiments in Exercise R-5.8.)

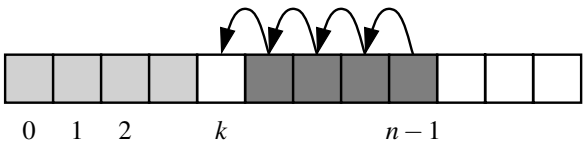


Figure 5.17: Removing an element at index k of a dynamic array.

The list class offers another method, named remove, that allows the caller to specify the *value* that should be removed (not the *index* at which it resides). Formally, it removes only the first occurrence of such a value from a list, or raises a ValueError if no such value is found. An implementation of such behavior is given in Code Fragment 5.6, again using our DynamicArray class for illustration.

Interestingly, there is no "efficient" case for remove; every call requires $\Omega(n)$ time. One part of the process searches from the beginning until finding the value at index k, while the rest iterates from k to the end in order to shift elements leftward. This linear behavior can be observed experimentally (see Exercise C-5.24).

```
1   def remove(self, value):
2     """Remove first occurrence of value (or raise ValueError)."""
3     # note: we do not consider shrinking the dynamic array in this version
4     for k in range(self._n):
5       if self._A[k] == value:                    # found a match!
6         for j in range(k, self._n - 1):          # shift others to fill gap
7           self._A[j] = self._A[j+1]
8         self._A[self._n - 1] = None              # help garbage collection
9         self._n -= 1                             # we have one less item
10        return                                   # exit immediately
11    raise ValueError('value not found')          # only reached if no match
```

Code Fragment 5.6: Implementation of remove for our DynamicArray class.

6.2.2 Array-Based Queue Implementation

For the stack ADT, we created a very simple adapter class that used a Python list as the underlying storage. It may be very tempting to use a similar approach for supporting the queue ADT. We could enqueue element e by calling append(e) to add it to the end of the list. We could use the syntax pop(0), as opposed to pop( ), to intentionally remove the ***first*** element from the list when dequeuing.

As easy as this would be to implement, it is tragically inefficient. As we discussed in Section 5.4.1, when pop is called on a list with a non-default index, a loop is executed to shift all elements beyond the specified index to the left, so as to fill the hole in the sequence caused by the pop. Therefore, a call to pop(0) always causes the worst-case behavior of $\Theta(n)$ time.

We can improve on the above strategy by avoiding the call to pop(0) entirely. We can replace the dequeued entry in the array with a reference to None, and maintain an explicit variable f to store the index of the element that is currently at the front of the queue. Such an algorithm for dequeue would run in $O(1)$ time. After several dequeue operations, this approach might lead to the configuration portrayed in Figure 6.5.

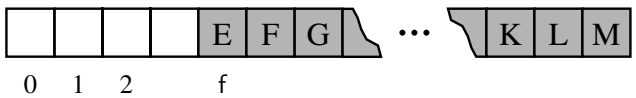


Figure 6.5: Allowing the front of the queue to drift away from index 0.

Unfortunately, there remains a drawback to the revised approach. In the case of a stack, the length of the list was precisely equal to the size of the stack (even if the underlying array for the list was slightly larger). With the queue design that we are considering, the situation is worse. We can build a queue that has relatively few elements, yet which are stored in an arbitrarily large list. This occurs, for example, if we repeatedly enqueue a new element and then dequeue another (allowing the front to drift rightward). Over time, the size of the underlying list would grow to $O(m)$ where m is the ***total*** number of enqueue operations since the creation of the queue, rather than the current number of elements in the queue.

This design would have detrimental consequences in applications in which queues have relatively modest size, but which are used for long periods of time. For example, the wait-list for a restaurant might never have more than 30 entries at one time, but over the course of a day (or a week), the overall number of entries would be significantly larger.

In Chapter 5 we carefully examined Python's array-based list class, and in Chapter 6 we demonstrated use of that class in implementing the classic stack, queue, and dequeue ADTs. Python's list class is highly optimized, and often a great choice for storage. With that said, there are some notable disadvantages:

1. The length of a dynamic array might be longer than the actual number of elements that it stores.
2. Amortized bounds for operations may be unacceptable in real-time systems.
3. Insertions and deletions at interior positions of an array are expensive.

In this chapter, we introduce a data structure known as a ***linked list***, which provides an alternative to an array-based sequence (such as a Python list). Both array-based sequences and linked lists keep elements in a certain order, but using a very different style. An array provides the more centralized representation, with one large chunk of memory capable of accommodating references to many elements. A linked list, in contrast, relies on a more distributed representation in which a lightweight object, known as a ***node***, is allocated for each element. Each node maintains a reference to its element and one or more references to neighboring nodes in order to collectively represent the linear order of the sequence.

We will demonstrate a trade-off of advantages and disadvantages when contrasting array-based sequences and linked lists. Elements of a linked list cannot be efficiently accessed by a numeric index k, and we cannot tell just by examining a node if it is the second, fifth, or twentieth node in the list. However, linked lists avoid the three disadvantages noted above for array-based sequences.

7.1 Singly Linked Lists

A ***singly linked list***, in its simplest form, is a collection of ***nodes*** that collectively form a linear sequence. Each node stores a reference to an object that is an element of the sequence, as well as a reference to the next node of the list (see Figures 7.1 and 7.2).

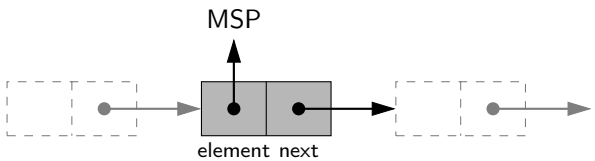


Figure 7.1: Example of a node instance that forms part of a singly linked list. The node's element member references an arbitrary object that is an element of the sequence (the airport code MSP, in this example), while the next member references the subsequent node of the linked list (or None if there is no further node).

7.3.1 Basic Implementation of a Doubly Linked List

We begin by providing a preliminary implementation of a doubly linked list, in the form of a class named _DoublyLinkedBase. We intentionally name the class with a leading underscore because we do not intend for it to provide a coherent public interface for general use. We will see that linked lists can support general insertions and deletions in $O(1)$ worst-case time, but only if the location of an operation can be succinctly identified. With array-based sequences, an integer index was a convenient means for describing a position within a sequence. However, an index is not convenient for linked lists as there is no efficient way to find the j^{th} element; it would seem to require a traversal of a portion of the list.

When working with a linked list, the most direct way to describe the location of an operation is by identifying a relevant node of the list. However, we prefer to encapsulate the inner workings of our data structure to avoid having users directly access nodes of a list. In the remainder of this chapter, we will develop two public classes that inherit from our _DoublyLinkedBase class to provide more coherent abstractions. Specifically, in Section 7.3.2, we provide a LinkedDeque class that implements the double-ended queue ADT introduced in Section 6.3; that class only supports operations at the ends of the queue, so there is no need for a user to identify an interior position within the list. In Section 7.4, we introduce a new PositionalList abstraction that provides a public interface that allows arbitrary insertions and deletions from a list.

Our low-level _DoublyLinkedBase class relies on the use of a nonpublic _Node class that is similar to that for a singly linked list, as given in Code Fragment 7.4, except that the doubly linked version includes a _prev attribute, in addition to the _next and _element attributes, as shown in Code Fragment 7.11.

```
class _Node:
  """Lightweight, nonpublic class for storing a doubly linked node."""
  __slots__ = '_element', '_prev', '_next'  # streamline memory

  def __init__(self, element, prev, next):          # initialize node's fields
    self._element = element                         # user's element
    self._prev = prev                               # previous node reference
    self._next = next                               # next node reference
```

Code Fragment 7.11: A Python _Node class for use in a doubly linked list.

The remainder of our _DoublyLinkedBase class is given in Code Fragment 7.12. The constructor instantiates the two sentinel nodes and links them directly to each other. We maintain a _size member and provide public support for __len__ and is_empty so that these behaviors can be directly inherited by the subclasses.

7.2 Circularly Linked Lists

In Section 6.2.2, we introduced the notion of a "circular" array and demonstrated its use in implementing the queue ADT. In reality, the notion of a circular array was artificial, in that there was nothing about the representation of the array itself that was circular in structure. It was our use of modular arithmetic when "advancing" an index from the last slot to the first slot that provided such an abstraction.

In the case of linked lists, there is a more tangible notion of a circularly linked list, as we can have the tail of the list use its next reference to point back to the head of the list, as shown in Figure 7.7. We call such a structure a ***circularly linked list***.

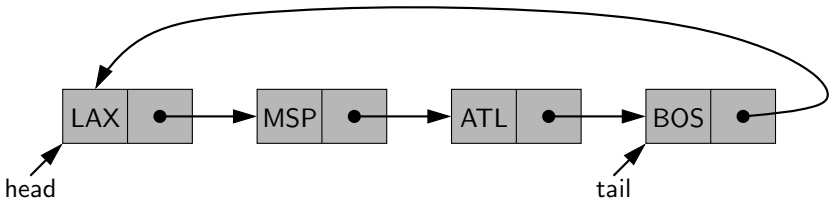


Figure 7.7: Example of a singly linked list with circular structure.

A circularly linked list provides a more general model than a standard linked list for data sets that are cyclic, that is, which do not have any particular notion of a beginning and end. Figure 7.8 provides a more symmetric illustration of the same circular list structure as Figure 7.7.

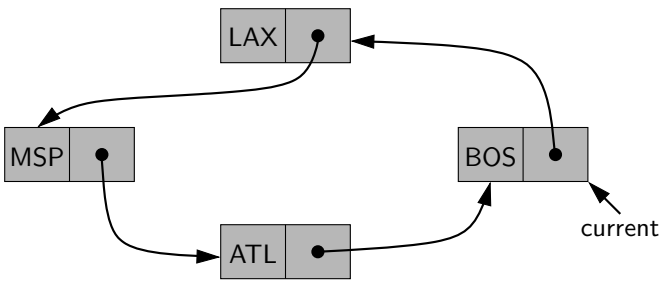


Figure 7.8: Example of a circular linked list, with current denoting a reference to a select node.

A circular view similar to Figure 7.8 could be used, for example, to describe the order of train stops in the Chicago loop, or the order in which players take turns during a game. Even though a circularly linked list has no beginning or end, per se, we must maintain a reference to a particular node in order to make use of the list. We use the identifier ***current*** to describe such a designated node. By setting current = current.next, we can effectively advance through the nodes of the list.

5.6 Multidimensional Data Sets

Lists, tuples, and strings in Python are one-dimensional. We use a single index to access each element of the sequence. Many computer applications involve multidimensional data sets. For example, computer graphics are often modeled in either two or three dimensions. Geographic information may be naturally represented in two dimensions, medical imaging may provide three-dimensional scans of a patient, and a company's valuation is often based upon a high number of independent financial measures that can be modeled as multidimensional data. A two-dimensional array is sometimes also called a ***matrix***. We may use two indices, say i and j, to refer to the cells in the matrix. The first index usually refers to a row number and the second to a column number, and these are traditionally zero-indexed in computer science. Figure 5.22 illustrates a two-dimensional data set with integer values. This data might, for example, represent the number of stores in various regions of Manhattan.

	0	1	2	3	4	5	6	7	8	9
0	22	18	709	5	33	10	4	56	82	440
1	45	32	830	120	750	660	13	77	20	105
2	4	880	45	66	61	28	650	7	510	67
3	940	12	36	3	20	100	306	590	0	500
4	50	65	42	49	88	25	70	126	83	288
5	398	233	5	83	59	232	49	8	365	90
6	33	58	632	87	94	5	59	204	120	829
7	62	394	3	4	102	140	183	390	16	26

Figure 5.22: Illustration of a two-dimensional integer data set, which has 8 rows and 10 columns. The rows and columns are zero-indexed. If this data set were named stores, the value of stores[3][5] is 100 and the value of stores[6][2] is 632.

A common representation for a two-dimensional data set in Python is as a list of lists. In particular, we can represent a two-dimensional array as a list of rows, with each row itself being a list of values. For example, the two-dimensional data

$$\begin{array}{ccccc} 22 & 18 & 709 & 5 & 33 \\ 45 & 32 & 830 & 120 & 750 \\ 4 & 880 & 45 & 66 & 61 \end{array}$$

might be stored in Python as follows.

```
data = [ [22, 18, 709, 5, 33], [45, 32, 830, 120, 750], [4, 880, 45, 66, 61] ]
```

An advantage of this representation is that we can naturally use a syntax such as data[1][3] to represent the value that has row index 1 and column index 3, as data[1], the second entry in the outer list, is itself a list, and thus indexable.

A Class for High Scores

To maintain a sequence of high scores, we develop a class named Scoreboard. A scoreboard is limited to a certain number of high scores that can be saved; once that limit is reached, a new score only qualifies for the scoreboard if it is strictly higher than the lowest "high score" on the board. The length of the desired scoreboard may depend on the game, perhaps 10, 50, or 500. Since that limit may vary depending on the game, we allow it to be specified as a parameter to our Scoreboard constructor.

Internally, we will use a Python list named _board in order to manage the GameEntry instances that represent the high scores. Since we expect the scoreboard to eventually reach full capacity, we initialize the list to be large enough to hold the maximum number of scores, but we initially set all entries to None. By allocating the list with maximum capacity initially, it never needs to be resized. As entries are added, we will maintain them from highest to lowest score, starting at index 0 of the list. We illustrate a typical state of the data structure in Figure 5.18.

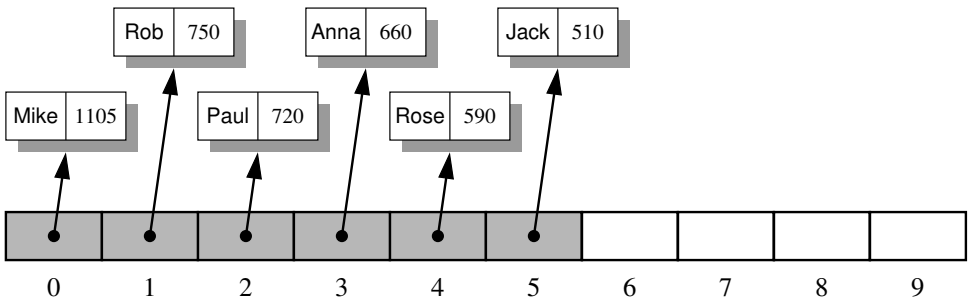


Figure 5.18: An illustration of an ordered list of length ten, storing references to six GameEntry objects in the cells from index 0 to 5, with the rest being None.

A complete Python implementation of the Scoreboard class is given in Code Fragment 5.8. The constructor is rather simple. The command

self._board = [**None**] * capacity

creates a list with the desired length, yet all entries equal to None. We maintain an additional instance variable, _n, that represents the number of actual entries currently in our table. For convenience, our class supports the __getitem__ method to retrieve an entry at a given index with a syntax board[i] (or None if no such entry exists), and we support a simple __str__ method that returns a string representation of the entire scoreboard, with one entry per line.

7.4 The Positional List ADT

The abstract data types that we have considered thus far, namely stacks, queues, and double-ended queues, only allow update operations that occur at one end of a sequence or the other. We wish to have a more general abstraction. For example, although we motivated the FIFO semantics of a queue as a model for customers who are waiting to speak with a customer service representative, or fans who are waiting in line to buy tickets to a show, the queue ADT is too limiting. What if a waiting customer decides to hang up before reaching the front of the customer service queue? Or what if someone who is waiting in line to buy tickets allows a friend to "cut" into line at that position? We would like to design an abstract data type that provides a user a way to refer to elements anywhere in a sequence, and to perform arbitrary insertions and deletions.

When working with array-based sequences (such as a Python list), integer indices provide an excellent means for describing the location of an element, or the location at which an insertion or deletion should take place. However, numeric indices are not a good choice for describing positions within a linked list because we cannot efficiently access an entry knowing only its index; finding an element at a given index within a linked list requires traversing the list incrementally from its beginning or end, counting elements as we go.

Furthermore, indices are not a good abstraction for describing a local position in some applications, because the index of an entry changes over time due to insertions or deletions that happen earlier in the sequence. For example, it may not be convenient to describe the location of a person waiting in line by knowing precisely how far away that person is from the front of the line. We prefer an abstraction, as characterized in Figure 7.14, in which there is some other means for describing a position. We then wish to model situations such as when an identified person leaves the line before reaching the front, or in which a new person is added to a line immediately behind another identified person.

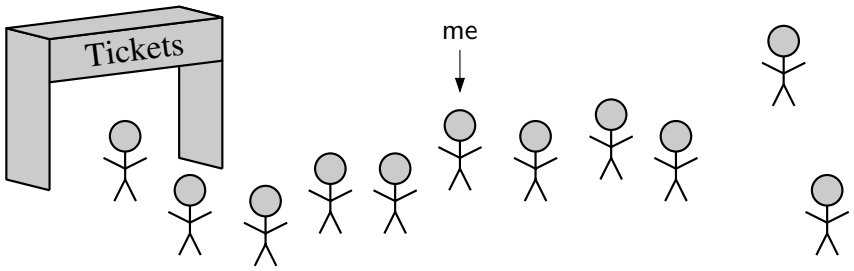


Figure 7.14: We wish to be able to identify the position of an element in a sequence without the use of an integer index.

9.3.3 Array-Based Representation of a Complete Binary Tree

The array-based representation of a binary tree (Section 8.3.2) is especially suitable for a complete binary tree T. We recall that in this implementation, the elements of T are stored in an array-based list A such that the element at position p in T is stored in A with index equal to the level number $f(p)$ of p, defined as follows:

- If p is the root of T, then $f(p) = 0$.
- If p is the left child of position q, then $f(p) = 2f(q) + 1$.
- If p is the right child of position q, then $f(p) = 2f(q) + 2$.

With this implementation, the elements of T have contiguous indices in the range $[0, n-1]$ and the last position of T is always at index $n-1$, where n is the number of positions of T. For example, Figure 9.4 illustrates the array-based representation of the heap structure originally portrayed in Figure 9.1.

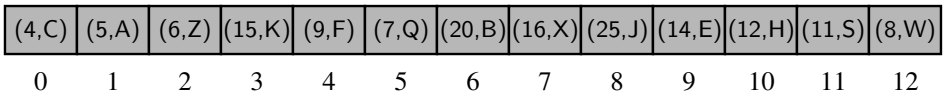


Figure 9.4: An array-based representation of the heap from Figure 9.1.

Implementing a priority queue using an array-based heap representation allows us to avoid some complexities of a node-based tree structure. In particular, the add and remove_min operations of a priority queue both depend on locating the last index of a heap of size n. With the array-based representation, the last position is at index $n-1$ of the array. Locating the last position of a complete binary tree implemented with a linked structure requires more effort. (See Exercise C-9.34.)

If the size of a priority queue is not known in advance, use of an array-based representation does introduce the need to dynamically resize the array on occasion, as is done with a Python list. The space usage of such an array-based representation of a complete binary tree with n nodes is $O(n)$, and the time bounds of methods for adding or removing elements become ***amortized***. (See Section 5.3.1.)

9.3.4 Python Heap Implementation

We provide a Python implementation of a heap-based priority queue in Code Fragments 9.4 and 9.5. We use an array-based representation, maintaining a Python list of item composites. Although we do not formally use the binary tree ADT, Code Fragment 9.4 includes nonpublic utility functions that compute the level numbering of a parent or child of another. This allows us to describe the rest of our algorithms using tree-like terminology of *parent*, *left*, and *right*. However, the relevant variables are integer indexes (not "position" objects). We use recursion to implement the repetition in the _upheap and _downheap utilities.

As another example, a text document can be viewed as a long sequence of characters. A word processor uses the abstraction of a ***cursor*** to describe a position within the document without explicit use of an integer index, allowing operations such as "delete the character at the cursor" or "insert a new character just after the cursor." Furthermore, we may be able to refer to an inherent position within a document, such as the beginning of a particular section, without relying on a character index (or even a section number) that may change as the document evolves.

A Node Reference as a Position?

One of the great benefits of a linked list structure is that it is possible to perform $O(1)$-time insertions and deletions at arbitrary positions of the list, as long as we are given a reference to a relevant node of the list. It is therefore very tempting to develop an ADT in which a node reference serves as the mechanism for describing a position. In fact, our _DoublyLinkedBase class of Section 7.3.1 has methods _insert_between and _delete_node that accept node references as parameters.

However, such direct use of nodes would violate the object-oriented design principles of abstraction and encapsulation that were introduced in Chapter 2. There are several reasons to prefer that we encapsulate the nodes of a linked list, for both our sake and for the benefit of users of our abstraction.

- It will be simpler for users of our data structure if they are not bothered with unnecessary details of our implementation, such as low-level manipulation of nodes, or our reliance on the use of sentinel nodes. Notice that to use the _insert_between method of our _DoublyLinkedBase class to add a node at the beginning of a sequence, the header sentinel must be sent as a parameter.
- We can provide a more robust data structure if we do not permit users to directly access or manipulate the nodes. In that way, we ensure that users cannot invalidate the consistency of a list by mismanaging the linking of nodes. A more subtle problem arises if a user were allowed to call the _insert_between or _delete_node method of our _DoublyLinkedBase class, sending a node that does not belong to the given list as a parameter. (Go back and look at that code and see why it causes a problem!)
- By better encapsulating the internal details of our implementation, we have greater flexibility to redesign the data structure and improve its performance. In fact, with a well-designed abstraction, we can provide a notion of a non-numeric position, even if using an array-based sequence.

For these reasons, instead of relying directly on nodes, we introduce an independent ***position*** abstraction to denote the location of an element within a list, and then a complete ***positional list ADT*** that can encapsulate a doubly linked list (or even an array-based sequence; see Exercise P-7.46).

1.4.2 Loops

Python offers two distinct looping constructs. A **while** loop allows general repetition based upon the repeated testing of a Boolean condition. A **for** loop provides convenient iteration of values from a defined series (such as characters of a string, elements of a list, or numbers within a given range). We discuss both forms in this section.

While Loops

The syntax for a **while** loop in Python is as follows:

```
while condition:
  body
```

As with an **if** statement, *condition* can be an arbitrary Boolean expression, and *body* can be an arbitrary block of code (including nested control structures). The execution of a while loop begins with a test of the Boolean condition. If that condition evaluates to True, the body of the loop is performed. After each execution of the body, the loop condition is retested, and if it evaluates to True, another iteration of the body is performed. When the conditional test evaluates to False (assuming it ever does), the loop is exited and the flow of control continues just beyond the body of the loop.

As an example, here is a loop that advances an index through a sequence of characters until finding an entry with value `'X'` or reaching the end of the sequence.

```
j = 0
while j < len(data) and data[j] != 'X':
  j += 1
```

The len function, which we will introduce in Section 1.5.2, returns the length of a sequence such as a list or string. The correctness of this loop relies on the short-circuiting behavior of the **and** operator, as described on page 12. We intentionally test `j < len(data)` to ensure that j is a valid index, prior to accessing element `data[j]`. Had we written that compound condition with the opposite order, the evaluation of `data[j]` would eventually raise an IndexError when `'X'` is not found. (See Section 1.7 for discussion of exceptions.)

As written, when this loop terminates, variable j's value will be the index of the leftmost occurrence of `'X'`, if found, or otherwise the length of the sequence (which is recognizable as an invalid index to indicate failure of the search). It is worth noting that this code behaves correctly, even in the special case when the list is empty, as the condition `j < len(data)` will initially fail and the body of the loop will never be executed.

A group of related variables can be stored one after another in a contiguous portion of the computer's memory. We will denote such a representation as an ***array***. As a tangible example, a text string is stored as an ordered sequence of individual characters. In Python, each character is represented using the Unicode character set, and on most computing systems, Python internally represents each Unicode character with 16 bits (i.e., 2 bytes). Therefore, a six-character string, such as `'SAMPLE'`, would be stored in 12 consecutive bytes of memory, as diagrammed in Figure 5.2.

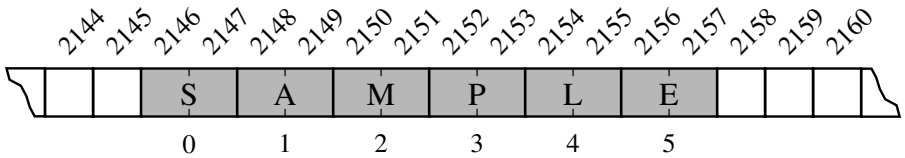


Figure 5.2: A Python string embedded as an array of characters in the computer's memory. We assume that each Unicode character of the string requires two bytes of memory. The numbers below the entries are indices into the string.

We describe this as an *array of six characters*, even though it requires 12 bytes of memory. We will refer to each location within an array as a ***cell***, and will use an integer ***index*** to describe its location within the array, with cells numbered starting with 0, 1, 2, and so on. For example, in Figure 5.2, the cell of the array with index 4 has contents `L` and is stored in bytes 2154 and 2155 of memory.

Each cell of an array must use the same number of bytes. This requirement is what allows an arbitrary cell of the array to be accessed in constant time based on its index. In particular, if one knows the memory address at which an array starts (e.g., 2146 in Figure 5.2), the number of bytes per element (e.g., 2 for a Unicode character), and a desired index within the array, the appropriate memory address can be computed using the calculation, $\mathsf{start} + \mathsf{cellsize} * \mathsf{index}$. By this formula, the cell at index 0 begins precisely at the start of the array, the cell at index 1 begins precisely cellsize bytes beyond the start of the array, and so on. As an example, cell 4 of Figure 5.2 begins at memory location $2146 + 2 \cdot 4 = 2146 + 8 = 2154$.

Of course, the arithmetic for calculating memory addresses within an array can be handled automatically. Therefore, a programmer can envision a more typical high-level abstraction of an array of characters as diagrammed in Figure 5.3.

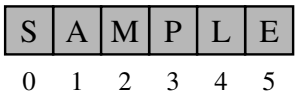


Figure 5.3: A higher-level abstraction for the string portrayed in Figure 5.2.

6.3.3 Deques in the Python Collections Module

An implementation of a deque class is available in Python's standard collections module. A summary of the most commonly used behaviors of the collections.deque class is given in Table 6.4. It uses more asymmetric nomenclature than our ADT.

Our Deque ADT	**collections.deque**	Description
len(D)	len(D)	number of elements
D.add_first( )	D.appendleft( )	add to beginning
D.add_last( )	D.append( )	add to end
D.delete_first( )	D.popleft( )	remove from beginning
D.delete_last( )	D.pop( )	remove from end
D.first( )	D[0]	access first element
D.last( )	D[−1]	access last element
	D[j]	access arbitrary entry by index
	D[j] = val	modify arbitrary entry by index
	D.clear( )	clear all contents
	D.rotate(k)	circularly shift rightward k steps
	D.remove(e)	remove first matching element
	D.count(e)	count number of matches for e

Table 6.4: Comparison of our deque ADT and the collections.deque class.

The collections.deque interface was chosen to be consistent with established naming conventions of Python's list class, for which append and pop are presumed to act at the end of the list. Therefore, appendleft and popleft designate an operation at the beginning of the list. The library deque also mimics a list in that it is an indexed sequence, allowing arbitrary access or modification using the D[j] syntax.

The library deque constructor also supports an optional maxlen parameter to force a fixed-length deque. However, if a call to append at either end is invoked when the deque is full, it does not throw an error; instead, it causes one element to be dropped from the opposite side. That is, calling appendleft when the deque is full causes an implicit pop from the right side to make room for the new element.

The current Python distribution implements collections.deque with a hybrid approach that uses aspects of circular arrays, but organized into blocks that are themselves organized in a doubly linked list (a data structure that we will introduce in the next chapter). The deque class is formally documented to guarantee $O(1)$-time operations at either end, but $O(n)$-time worst-case operations when using index notation near the middle of the deque.

9.5.2 Implementing an Adaptable Priority Queue

In this section, we provide a Python implementation of an adaptable priority queue as an extension of our HeapPriorityQueue class from Section 9.3.4. To implement a Locator class, we will extend the existing _Item composite to add an additional field designating the current index of the element within the array-based representation of our heap, as shown in Figure 9.10.

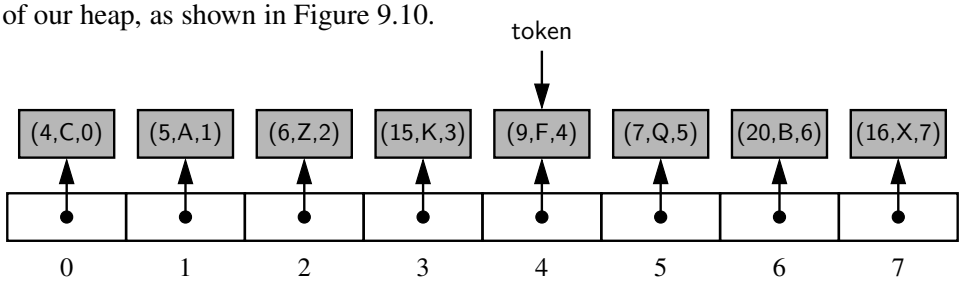


Figure 9.10: Representing a heap using a sequence of locators. The third element of each locator instance corresponds to the index of the item within the array. Identifier token is presumed to be a locator reference in the user's scope.

The list is a sequence of references to locator instances, each of which stores a key, value, and the current index of the item within the list. The user will be given a reference to the Locator instance for each inserted element, as portrayed by the token identifier in Figure 9.10.

When we perform priority queue operations on our heap, and items are relocated within our structure, we reposition the locator instances within the list and we update the third field of each locator to reflect its new index within the list. As an example, Figure 9.11 shows the state of the above heap after a call to remove_min( ). The heap operation caused the minimum entry, (4,C), to be removed, and the entry, (16,X), to be temporarily moved from the last position to the root, followed by a down-heap bubble phase. During the down-heap, element (16,X) was swapped

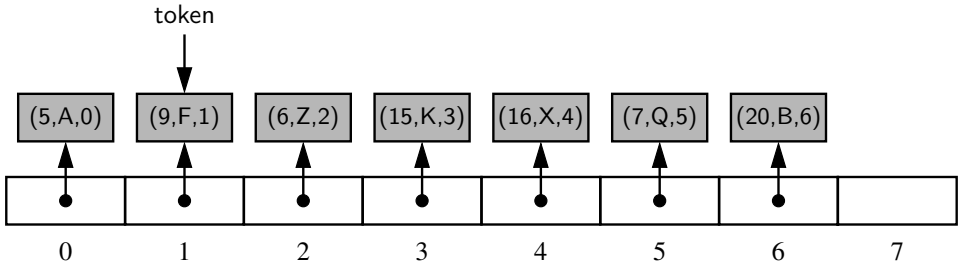


Figure 9.11: The result of a call to remove_min( ) on the heap originally portrayed in Figure 9.10. Identifier token continues to reference the same locator instance as in the original configuration, but the placement of that locator in the list has changed, as has the third field of the locator.

```
40   def enqueue(self, e):
41     """Add an element to the back of queue."""
42     if self._size == len(self._data):
43       self._resize(2 * len(self.data))          # double the array size
44     avail = (self._front + self._size) % len(self._data)
45     self._data[avail] = e
46     self._size += 1
47
48   def _resize(self, cap):                       # we assume cap >= len(self)
49     """Resize to a new list of capacity >= len(self)."""
50     old = self._data                            # keep track of existing list
51     self._data = [None] * cap                   # allocate list with new capacity
52     walk = self._front
53     for k in range(self._size):                 # only consider existing elements
54       self._data[k] = old[walk]                 # intentionally shift indices
55       walk = (1 + walk) % len(old)              # use old size as modulus
56     self._front = 0                             # front has been realigned
```

Code Fragment 6.7: Array-based implementation of a queue (continued from Code Fragment 6.6).

The implementation of __len__ and is_empty are trivial, given knowledge of the size. The implementation of the front method is also simple, as the _front index tells us precisely where the desired element is located within the _data list, assuming that list is not empty.

Adding and Removing Elements

The goal of the enqueue method is to add a new element to the back of the queue. We need to determine the proper index at which to place the new element. Although we do not explicitly maintain an instance variable for the back of the queue, we compute the location of the next opening based on the formula:

avail = (self._front + self._size) % len(self._data)

Note that we are using the size of the queue as it exists ***prior*** to the addition of the new element. For example, consider a queue with capacity 10, current size 3, and first element at index 5. The three elements of such a queue are stored at indices 5, 6, and 7. The new element should be placed at index (front + size) = 8. In a case with wrap-around, the use of the modular arithmetic achieves the desired circular semantics. For example, if our hypothetical queue had 3 elements with the first at index 8, our computation of (8+3) % 10 evaluates to 1, which is perfect since the three existing elements occupy indices 8, 9, and 0.

ride the remove_min method because the only change in behavior for the adaptable priority queue is again provided by the overridden _swap method.

The update and remove methods provide the core new functionality for the adaptable priority queue. We perform robust checking of the validity of a locator that is sent by a caller (although in the interest of space, our displayed code does not do preliminary type-checking to ensure that the parameter is indeed a Locator instance). To ensure that a locator is associated with a current element of the given priority queue, we examine the index that is encapsulated within the locator object, and then verify that the entry of the list at that index is the very same locator.

In conclusion, the adaptable priority queue provides the same asymptotic efficiency and space usage as the nonadaptive version, and provides logarithmic performance for the new locator-based update and remove methods. A summary of the performance is given in Table 9.4.

```
1  class AdaptableHeapPriorityQueue(HeapPriorityQueue):
2    """A locator-based priority queue implemented with a binary heap."""
3
4    #------------------------------ nested Locator class ------------------------------
5    class Locator(HeapPriorityQueue._Item):
6      """Token for locating an entry of the priority queue."""
7      __slots__ = '_index'                    # add index as additional field
8
9      def __init__(self, k, v, j):
10       super().__init__(k,v)
11       self._index = j
12
13   #------------------------------ nonpublic behaviors ------------------------------
14   # override swap to record new indices
15   def _swap(self, i, j):
16     super()._swap(i,j)                      # perform the swap
17     self._data[i]._index = i                # reset locator index (post-swap)
18     self._data[j]._index = j                # reset locator index (post-swap)
19
20   def _bubble(self, j):
21     if j > 0 and self._data[j] < self._data[self._parent(j)]:
22       self._upheap(j)
23     else:
24       self._downheap(j)
```

Code Fragment 9.8: An implementation of an adaptable priority queue (continued in Code Fragment 9.9). This extends the HeapPriorityQueue class of Code Fragments 9.4 and 9.5

13.5.4 Search Engine Indexing

The World Wide Web contains a huge collection of text documents (Web pages). Information about these pages are gathered by a program called a ***Web crawler***, which then stores this information in a special dictionary database. A Web ***search engine*** allows users to retrieve relevant information from this database, thereby identifying relevant pages on the Web containing given keywords. In this section, we present a simplified model of a search engine.

Inverted Files

The core information stored by a search engine is a dictionary, called an ***inverted index*** or ***inverted file***, storing key-value pairs (w, L), where w is a word and L is a collection of pages containing word w. The keys (words) in this dictionary are called ***index terms*** and should be a set of vocabulary entries and proper nouns as large as possible. The elements in this dictionary are called ***occurrence lists*** and should cover as many Web pages as possible.

We can efficiently implement an inverted index with a data structure consisting of the following:

1. An array storing the occurrence lists of the terms (in no particular order).
2. A compressed trie for the set of index terms, where each leaf stores the index of the occurrence list of the associated term.

The reason for storing the occurrence lists outside the trie is to keep the size of the trie data structure sufficiently small to fit in internal memory. Instead, because of their large total size, the occurrence lists have to be stored on disk.

With our data structure, a query for a single keyword is similar to a word-matching query (Section 13.5.1). Namely, we find the keyword in the trie and we return the associated occurrence list.

When multiple keywords are given and the desired output are the pages containing ***all*** the given keywords, we retrieve the occurrence list of each keyword using the trie and return their intersection. To facilitate the intersection computation, each occurrence list should be implemented with a sequence sorted by address or with a map, to allow efficient set operations.

In addition to the basic task of returning a list of pages containing given keywords, search engines provide an important additional service by ***ranking*** the pages returned by relevance. Devising fast and accurate ranking algorithms for search engines is a major challenge for computer researchers and electronic commerce companies.

For Loops

Python's **for**-loop syntax is a more convenient alternative to a while loop when iterating through a series of elements. The for-loop syntax can be used on any type of ***iterable*** structure, such as a list, tuple str, set, dict, or file (we will discuss iterators more formally in Section 1.8). Its general syntax appears as follows.

```
for element in iterable:
  body                          # body may refer to 'element' as an identifier
```

For readers familiar with Java, the semantics of Python's for loop is similar to the "for each" loop style introduced in Java 1.5.

As an instructive example of such a loop, we consider the task of computing the sum of a list of numbers. (Admittedly, Python has a built-in function, sum, for this purpose.) We perform the calculation with a for loop as follows, assuming that data identifies the list:

```
total = 0
for val in data:
  total += val                  # note use of the loop variable, val
```

The loop body executes once for each element of the data sequence, with the identifier, val, from the for-loop syntax assigned at the beginning of each pass to a respective element. It is worth noting that val is treated as a standard identifier. If the element of the original data happens to be mutable, the val identifier can be used to invoke its methods. But a reassignment of identifier val to a new value has no affect on the original data, nor on the next iteration of the loop.

As a second classic example, we consider the task of finding the maximum value in a list of elements (again, admitting that Python's built-in max function already provides this support). If we can assume that the list, data, has at least one element, we could implement this task as follows:

```
biggest = data[0]               # as we assume nonempty list
for val in data:
  if val > biggest:
    biggest = val
```

Although we could accomplish both of the above tasks with a while loop, the for-loop syntax had an advantage of simplicity, as there is no need to manage an explicit index into the list nor to author a Boolean loop condition. Furthermore, we can use a for loop in cases for which a while loop does not apply, such as when iterating through a collection, such as a set, that does not support any direct form of indexing.

1.12 Exercises

For help with exercises, please visit the site, www.wiley.com/college/goodrich.

Reinforcement

R-1.1 Write a short Python function, is_multiple(n, m), that takes two integer values and returns True if n is a multiple of m, that is, $n = mi$ for some integer i, and False otherwise.

R-1.2 Write a short Python function, is_even(k), that takes an integer value and returns True if k is even, and **False** otherwise. However, your function cannot use the multiplication, modulo, or division operators.

R-1.3 Write a short Python function, minmax(data), that takes a sequence of one or more numbers, and returns the smallest and largest numbers, in the form of a tuple of length two. Do not use the built-in functions min or max in implementing your solution.

R-1.4 Write a short Python function that takes a positive integer n and returns the sum of the squares of all the positive integers smaller than n.

R-1.5 Give a single command that computes the sum from Exercise R-1.4, relying on Python's comprehension syntax and the built-in sum function.

R-1.6 Write a short Python function that takes a positive integer n and returns the sum of the squares of all the odd positive integers smaller than n.

R-1.7 Give a single command that computes the sum from Exercise R-1.6, relying on Python's comprehension syntax and the built-in sum function.

R-1.8 Python allows negative integers to be used as indices into a sequence, such as a string. If string s has length n, and expression s[k] is used for index $-n \leq k < 0$, what is the equivalent index $j \geq 0$ such that s[j] references the same element?

R-1.9 What parameters should be sent to the range constructor, to produce a range with values 50, 60, 70, 80?

R-1.10 What parameters should be sent to the range constructor, to produce a range with values 8, 6, 4, 2, 0, -2, -4, -6, -8?

R-1.11 Demonstrate how to use Python's list comprehension syntax to produce the list [1, 2, 4, 8, 16, 32, 64, 128, 256].

R-1.12 Python's random module includes a function choice(data) that returns a random element from a non-empty sequence. The random module includes a more basic function randrange, with parameterization similar to the built-in range function, that return a random choice from the given range. Using only the randrange function, implement your own version of the choice function.

C-1.22 Write a short Python program that takes two arrays a and b of length n storing **int** values, and returns the dot product of a and b. That is, it returns an array c of length n such that $c[i] = a[i] \cdot b[i]$, for $i = 0, \ldots, n-1$.

C-1.23 Give an example of a Python code fragment that attempts to write an element to a list based on an index that may be out of bounds. If that index is out of bounds, the program should catch the exception that results, and print the following error message:
"`Don't try buffer overflow attacks in Python!`"

C-1.24 Write a short Python function that counts the number of vowels in a given character string.

C-1.25 Write a short Python function that takes a string s, representing a sentence, and returns a copy of the string with all punctuation removed. For example, if given the string `"Let's try, Mike."`, this function would return `"Lets try Mike"`.

C-1.26 Write a short program that takes as input three integers, a, b, and c, from the console and determines if they can be used in a correct arithmetic formula (in the given order), like "$a+b=c$," "$a=b-c$," or "$a*b=c$."

C-1.27 In Section 1.8, we provided three different implementations of a generator that computes factors of a given integer. The third of those implementations, from page 41, was the most efficient, but we noted that it did not yield the factors in increasing order. Modify the generator so that it reports factors in increasing order, while maintaining its general performance advantages.

C-1.28 The ***p-norm*** of a vector $v = (v_1, v_2, \ldots, v_n)$ in n-dimensional space is defined as

$$\|v\| = \sqrt[p]{v_1^p + v_2^p + \cdots + v_n^p}.$$

For the special case of $p = 2$, this results in the traditional ***Euclidean norm***, which represents the length of the vector. For example, the Euclidean norm of a two-dimensional vector with coordinates $(4,3)$ has a Euclidean norm of $\sqrt{4^2+3^2} = \sqrt{16+9} = \sqrt{25} = 5$. Give an implementation of a function named norm such that norm(v, p) returns the p-norm value of v and norm(v) returns the Euclidean norm of v. You may assume that v is a list of numbers.

Loop Invariants

The final justification technique we discuss in this section is the ***loop invariant***. To prove some statement $\mathcal{L}$ about a loop is correct, define $\mathcal{L}$ in terms of a series of smaller statements $\mathcal{L}_0, \mathcal{L}_1, \ldots, \mathcal{L}_k$, where:

1. The ***initial*** claim, $\mathcal{L}_0$, is true before the loop begins.
2. If $\mathcal{L}_{j-1}$ is true before iteration j, then $\mathcal{L}_j$ will be true after iteration j.
3. The final statement, $\mathcal{L}_k$, implies the desired statement $\mathcal{L}$ to be true.

Let us give a simple example of using a loop-invariant argument to justify the correctness of an algorithm. In particular, we use a loop invariant to justify that the function, find (see Code Fragment 3.9), finds the smallest index at which element val occurs in sequence S.

```
1  def find(S, val):
2    """Return index j such that S[j] == val, or -1 if no such element."""
3    n = len(S)
4    j = 0
5    while j < n:
6      if S[j] == val:
7        return j              # a match was found at index j
8      j += 1
9    return -1
```

Code Fragment 3.9: Algorithm for finding the first index at which a given element occurs in a Python list.

To show that find is correct, we inductively define a series of statements, $\mathcal{L}_j$, that lead to the correctness of our algorithm. Specifically, we claim the following is true at the beginning of iteration j of the **while** loop:

$\mathcal{L}_j$: *val* is not equal to any of the first j elements of S.

This claim is true at the beginning of the first iteration of the loop, because j is 0 and there are no elements among the first 0 in S (this kind of a trivially true claim is said to hold ***vacuously***). In iteration j, we compare element *val* to element $S[j]$ and return the index j if these two elements are equivalent, which is clearly correct and completes the algorithm in this case. If the two elements *val* and $S[j]$ are not equal, then we have found one more element not equal to *val* and we increment the index j. Thus, the claim $\mathcal{L}_j$ will be true for this new value of j; hence, it is true at the beginning of the next iteration. If the while loop terminates without ever returning an index in S, then we have $j = n$. That is, $\mathcal{L}_n$ is true—there are no elements of S equal to *val*. Therefore, the algorithm correctly returns -1 to indicate that *val* is not in S.

5.4 Efficiency of Python's Sequence Types

In the previous section, we began to explore the underpinnings of Python's list class, in terms of implementation strategies and efficiency. We continue in this section by examining the performance of all of Python's sequence types.

5.4.1 Python's List and Tuple Classes

The *nonmutating* behaviors of the list class are precisely those that are supported by the tuple class. We note that tuples are typically more memory efficient than lists because they are immutable; therefore, there is no need for an underlying dynamic array with surplus capacity. We summarize the asymptotic efficiency of the nonmutating behaviors of the list and tuple classes in Table 5.3. An explanation of this analysis follows.

Operation	Running Time
len(data)	$O(1)$
data[j]	$O(1)$
data.count(value)	$O(n)$
data.index(value)	$O(k+1)$
value **in** data	$O(k+1)$
data1 == data2 (similarly !=, <, <=, >, >=)	$O(k+1)$
data[j:k]	$O(k-j+1)$
data1 + data2	$O(n_1+n_2)$
c * data	$O(cn)$

Table 5.3: Asymptotic performance of the nonmutating behaviors of the list and tuple classes. Identifiers data, data1, and data2 designate instances of the list or tuple class, and n, n_1, and n_2 their respective lengths. For the containment check and index method, k represents the index of the leftmost occurrence (with $k = n$ if there is no occurrence). For comparisons between two sequences, we let k denote the leftmost index at which they disagree or else $k = \min(n_1, n_2)$.

Constant-Time Operations

The length of an instance is returned in constant time because an instance explicitly maintains such state information. The constant-time efficiency of syntax data[j] is assured by the underlying access into an array.

P-7.45 An array A is ***sparse*** if most of its entries are empty (i.e., None). A list L can be used to implement such an array efficiently. In particular, for each nonempty cell $A[i]$, we can store an entry (i,e) in L, where e is the element stored at $A[i]$. This approach allows us to represent A using $O(m)$ storage, where m is the number of nonempty entries in A. Provide such a SparseArray class that minimally supports methods __getitem__(j) and __setitem__(j, e) to provide standard indexing operations. Analyze the efficiency of these methods.

P-7.46 Although we have used a doubly linked list to implement the positional list ADT, it is possible to support the ADT with an array-based implementation. The key is to use the composition pattern and store a sequence of position items, where each item stores an element as well as that element's current index in the array. Whenever an element's place in the array is changed, the recorded index in the position must be updated to match. Given a complete class providing such an array-based implementation of the positional list ADT. What is the efficiency of the various operations?

P-7.47 Implement a CardHand class that supports a person arranging a group of cards in his or her hand. The simulator should represent the sequence of cards using a single positional list ADT so that cards of the same suit are kept together. Implement this strategy by means of four "fingers" into the hand, one for each of the suits of hearts, clubs, spades, and diamonds, so that adding a new card to the person's hand or playing a correct card from the hand can be done in constant time. The class should support the following methods:

- add_card(r, s): Add a new card with rank r and suit s to the hand.
- play(s): Remove and return a card of suit s from the player's hand; if there is no card of suit s, then remove and return an arbitrary card from the hand.
- __iter__(): Iterate through all cards currently in the hand.
- all_of_suit(s): Iterate through all cards of suit s that are currently in the hand.

Chapter Notes

A view of data structures as collections (and other principles of object-oriented design) can be found in object-oriented design books by Booch [17], Budd [20], Goldberg and Robson [42], and Liskov and Guttag [71]. Our positional list ADT is derived from the "position" abstraction introduced by Aho, Hopcroft, and Ullman [6], and the list ADT of Wood [104]. Implementations of linked lists are discussed by Knuth [64].

C-10.39 Although keys in a map are distinct, the binary search algorithm can be applied in a more general setting in which an array stores possibly duplicative elements in nondecreasing order. Consider the goal of identifying the index of the *leftmost* element with key greater than or equal to given k. Does the _find_index method as given in Code Fragment 10.8 guarantee such a result? Does the _find_index method as given in Exercise R-10.21 guarantee such a result? Justify your answers.

C-10.40 Suppose we are given two sorted search tables S and T, each with n entries (with S and T being implemented with arrays). Describe an $O(\log^2 n)$-time algorithm for finding the k^{th} smallest key in the union of the keys from S and T (assuming no duplicates).

C-10.41 Give an $O(\log n)$-time solution for the previous problem.

C-10.42 Suppose that each row of an $n \times n$ array A consists of 1's and 0's such that, in any row of A, all the 1's come before any 0's in that row. Assuming A is already in memory, describe a method running in $O(n \log n)$ time (not $O(n^2)$ time!) for counting the number of 1's in A.

C-10.43 Given a collection C of n cost-performance pairs (c, p), describe an algorithm for finding the maxima pairs of C in $O(n \log n)$ time.

C-10.44 Show that the methods above(p) and prev(p) are not actually needed to efficiently implement a map using a skip list. That is, we can implement insertions and deletions in a skip list using a strictly top-down, scan-forward approach, without ever using the above or prev methods. (Hint: In the insertion algorithm, first repeatedly flip the coin to determine the level where you should start inserting the new entry.)

C-10.45 Describe how to modify a skip-list representation so that index-based operations, such as retrieving the item at index j, can be performed in $O(\log n)$ expected time.

C-10.46 For sets S and T, the syntax S ^ T returns a new set that is the symmetric difference, that is, a set of elements that are in precisely one of S or T. This syntax is supported by the special __xor__ method. Provide an implementation of that method in the context of the MutableSet abstract base class, relying only on the five primary abstract methods of that class.

C-10.47 In the context of the MutableSet abstract base class, describe a concrete implementation of the __and__ method, which supports the syntax S & T for computing the intersection of two existing sets.

C-10.48 An ***inverted file*** is a critical data structure for implementing a search engine or the index of a book. Given a document D, which can be viewed as an unordered, numbered list of words, an inverted file is an ordered list of words, L, such that, for each word w in L, we store the indices of the places in D where w appears. Design an efficient algorithm for constructing L from D.

2.3.5 Example: Range Class

As the final example for this section, we develop our own implementation of a class that mimics Python's built-in range class. Before introducing our class, we discuss the history of the built-in version. Prior to Python 3 being released, range was implemented as a function, and it returned a list instance with elements in the specified range. For example, range(2, 10, 2) returned the list [2, 4, 6, 8]. However, a typical use of the function was to support a for-loop syntax, such as **for** k **in** range(10000000). Unfortunately, this caused the instantiation and initialization of a list with the range of numbers. That was an unnecessarily expensive step, in terms of both time and memory usage.

The mechanism used to support ranges in Python 3 is entirely different (to be fair, the "new" behavior existed in Python 2 under the name xrange). It uses a strategy known as ***lazy evaluation***. Rather than creating a new list instance, range is a class that can effectively represent the desired range of elements without ever storing them explicitly in memory. To better explore the built-in range class, we recommend that you create an instance as r = range(8, 140, 5). The result is a relatively lightweight object, an instance of the range class, that has only a few behaviors. The syntax len(r) will report the number of elements that are in the given range (27, in our example). A range also supports the __getitem__ method, so that syntax r[15] reports the sixteenth element in the range (as r[0] is the first element). Because the class supports both __len__ and __getitem__, it inherits automatic support for iteration (see Section 2.3.4), which is why it is possible to execute a for loop over a range.

At this point, we are ready to demonstrate our own version of such a class. Code Fragment 2.6 provides a class we name Range (so as to clearly differentiate it from built-in range). The biggest challenge in the implementation is properly computing the number of elements that belong in the range, given the parameters sent by the caller when constructing a range. By computing that value in the constructor, and storing it as self._length, it becomes trivial to return it from the __len__ method. To properly implement a call to __getitem__(k), we simply take the starting value of the range plus k times the step size (i.e., for k=0, we return the start value). There are a few subtleties worth examining in the code:

- To properly support optional parameters, we rely on the technique described on page 27, when discussing a functional version of range.
- We compute the number of elements in the range as
  max(0, (stop − start + step − 1) // step)
  It is worth testing this formula for both positive and negative step sizes.
- The __getitem__ method properly supports negative indices by converting an index −k to len(self)−k before computing the result.

with its left child, (5,A), at index 1 of the list, then swapped with its right child, (9,F), at index 4 of the list. In the final configuration, the locator instances for all affected elements have been modified to reflect their new location.

It is important to emphasize that the locator instances have not changed identity. The user's token reference, portrayed in Figures 9.10 and 9.11, continues to reference the same instance; we have simply changed the third field of that instance, and we have changed where that instance is referenced within the list sequence.

With this new representation, providing the additional support for the adaptable priority queue ADT is rather straightforward. When a locator instance is sent as a parameter to update or remove, we may rely on the third field of that structure to designate where the element resides in the heap. With that knowledge, the update of a key may simply require an up-heap or down-heap bubbling step to reestablish the heap-order property. (The complete binary tree property remains intact.) To implement the removal of an arbitrary element, we move the element at the last position to the vacated location, and again perform an appropriate bubbling step to satisfy the heap-order property.

Python Implementation

Code Fragments 9.8 and 9.9 present a Python implementation of an adaptable priority queue, as a subclass of the HeapPriorityQueue class from Section 9.3.4. Our modifications to the original class are relatively minor. We define a public Locator class that inherits from the nonpublic _Item class and augments it with an additional _index field. We make it a public class because we will be using locators as return values and parameters; however, the public interface for the locator class does not include any other functionality for the user.

To update locators during the flow of our heap operations, we rely on an intentional design decision that our original class uses a nonpublic _swap method for all data movement. We override that utility to execute the additional step of updating the stored indices within the two swapped locator instances.

We provide a new _bubble utility that manages the reinstatement of the heap-order property when a key has changed at an arbitrary position within the heap, either due to a key update, or the blind replacement of a removed element with the item from the last position of the tree. The _bubble utility determines whether to apply up-heap or down-heap bubbling, depending on whether the given location has a parent with a smaller key. (If an updated key coincidentally remains valid for its current location, we technically call _downheap but no swaps result.)

The public methods are provided in Code Fragment 9.9. The existing add method is overridden, both to make use of a Locator instance rather than an _Item instance for storage of the new element, and to return the locator to the caller. The remainder of that method is similar to the original, with the management of locator indices enacted by the use of the new version of _swap. There is no reason to over-

12.3.2 Additional Optimizations for Quick-Sort

An algorithm is ***in-place*** if it uses only a small amount of memory in addition to that needed for the original input. Our implementation of heap-sort, from Section 9.4.2, is an example of such an in-place sorting algorithm. Our implementation of quick-sort from Code Fragment 12.5 does not qualify as in-place because we use additional containers L, E, and G when dividing a sequence S within each recursive call. Quick-sort of an array-based sequence can be adapted to be in-place, and such an optimization is used in most deployed implementations.

Performing the quick-sort algorithm in-place requires a bit of ingenuity, however, for we must use the input sequence itself to store the subsequences for all the recursive calls. We show algorithm inplace_quick_sort, which performs in-place quick-sort, in Code Fragment 12.6. Our implementation assumes that the input sequence, S, is given as a Python list of elements. In-place quick-sort modifies the input sequence using element swapping and does not explicitly create subsequences. Instead, a subsequence of the input sequence is implicitly represented by a range of positions specified by a leftmost index a and a rightmost index b. The

```
1  def inplace_quick_sort(S, a, b):
2    """Sort the list from S[a] to S[b] inclusive using the quick-sort algorithm."""
3    if a >= b: return                          # range is trivially sorted
4    pivot = S[b]                               # last element of range is pivot
5    left = a                                   # will scan rightward
6    right = b-1                                # will scan leftward
7    while left <= right:
8      # scan until reaching value equal or larger than pivot (or right marker)
9      while left <= right and S[left] < pivot:
10       left += 1
11     # scan until reaching value equal or smaller than pivot (or left marker)
12     while left <= right and pivot < S[right]:
13       right -= 1
14     if left <= right:                          # scans did not strictly cross
15       S[left], S[right] = S[right], S[left]                  # swap values
16       left, right = left + 1, right - 1                      # shrink range
17
18   # put pivot into its final place (currently marked by left index)
19   S[left], S[b] = S[b], S[left]
20   # make recursive calls
21   inplace_quick_sort(S, a, left - 1)
22   inplace_quick_sort(S, left + 1, b)
```

Code Fragment 12.6: In-place quick-sort for a Python list S.

The importance of the Floyd-Warshall algorithm is that it is much easier to implement than DFS, and much faster in practice because there are relatively few low-level operations hidden within the asymptotic notation. The algorithm is particularly well suited for the use of an adjacency matrix, as a single bit can be used to designate the reachability modeled as an edge (u,v) in the transitive closure.

However, note that repeated calls to DFS results in better asymptotic performance when the graph is sparse and represented using an adjacency list or adjacency map. In that case, a single DFS runs in $O(n+m)$ time, and so the transitive closure can be computed in $O(n^2+nm)$ time, which is preferable to $O(n^3)$.

Python Implementation

We conclude with a Python implementation of the Floyd-Warshall algorithm, as presented in Code Fragment 14.10. Although the original algorithm is described using a series of directed graphs $\vec{G}_0, \vec{G}_1, \ldots, \vec{G}_n$, we create a single copy of the original graph (using the deepcopy method of Python's copy module) and then repeatedly add new edges to the closure as we progress through rounds of the Floyd-Warshall algorithm.

The algorithm requires a canonical numbering of the graph's vertices; therefore, we create a list of the vertices in the closure graph, and subsequently index that list for our order. Within the outermost loop, we must consider all pairs i and j. Finally, we optimize by only iterating through all values of j after we have verified that i has been chosen such that (v_i, v_k) exists in the current version of our closure.

```
1  def floyd_warshall(g):
2    """Return a new graph that is the transitive closure of g."""
3    closure = deepcopy(g)                    # imported from copy module
4    verts = list(closure.vertices( ))        # make indexable list
5    n = len(verts)
6    for k in range(n):
7      for i in range(n):
8        # verify that edge (i,k) exists in the partial closure
9        if i != k and closure.get_edge(verts[i],verts[k]) is not None:
10         for j in range(n):
11           # verify that edge (k,j) exists in the partial closure
12           if i != j != k and closure.get_edge(verts[k],verts[j]) is not None:
13             # if (i,j) not yet included, add it to the closure
14             if closure.get_edge(verts[i],verts[j]) is None:
15               closure.insert_edge(verts[i],verts[j])
16   return closure
```

Code Fragment 14.10: Python implementation of the Floyd-Warshall algorithm.

4.1.3 Binary Search

In this section, we describe a classic recursive algorithm, ***binary search***, that is used to efficiently locate a target value within a sorted sequence of n elements. This is among the most important of computer algorithms, and it is the reason that we so often store data in sorted order (as in Figure 4.4).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

Figure 4.4: Values stored in sorted order within an indexable sequence, such as a Python list. The numbers at top are the indices.

When the sequence is ***unsorted***, the standard approach to search for a target value is to use a loop to examine every element, until either finding the target or exhausting the data set. This is known as the ***sequential search*** algorithm. This algorithm runs in $O(n)$ time (i.e., linear time) since every element is inspected in the worst case.

When the sequence is ***sorted*** and ***indexable***, there is a much more efficient algorithm. (For intuition, think about how you would accomplish this task by hand!) For any index j, we know that all the values stored at indices $0,\ldots,j-1$ are less than or equal to the value at index j, and all the values stored at indices $j+1,\ldots,n-1$ are greater than or equal to that at index j. This observation allows us to quickly "home in" on a search target using a variant of the children's game "high-low." We call an element of the sequence a ***candidate*** if, at the current stage of the search, we cannot rule out that this item matches the target. The algorithm maintains two parameters, low and high, such that all the candidate entries have index at least low and at most high. Initially, low $= 0$ and high $= n-1$. We then compare the target value to the median candidate, that is, the item data[mid] with index

$$\text{mid} = \lfloor (\text{low} + \text{high})/2 \rfloor .$$

We consider three cases:

- If the target equals data[mid], then we have found the item we are looking for, and the search terminates successfully.
- If target $<$ data[mid], then we recur on the first half of the sequence, that is, on the interval of indices from low to mid -1.
- If target $>$ data[mid], then we recur on the second half of the sequence, that is, on the interval of indices from mid $+1$ to high.

An unsuccessful search occurs if low $>$ high, as the interval [low, high] is empty.

12.2.2 Array-Based Implementation of Merge-Sort

We begin by focusing on the case when a sequence of items is represented as an (array-based) Python list. The merge function (Code Fragment 12.1) is responsible for the subtask of merging two previously sorted sequences, S_1 and S_2, with the output copied into S. We copy one element during each pass of the while loop, conditionally determining whether the next element should be taken from S_1 or S_2. The divide-and-conquer merge-sort algorithm is given in Code Fragment 12.2.

We illustrate a step of the merge process in Figure 12.5. During the process, index i represents the number of elements of S_1 that have been copied to S, while index j represents the number of elements of S_2 that have been copied to S. Assuming S_1 and S_2 both have at least one uncopied element, we copy the smaller of the two elements being considered. Since $i + j$ objects have been previously copied, the next element is placed in $S[i + j]$. (For example, when $i + j$ is 0, the next element is copied to $S[0]$). If we reach the end of one of the sequences, we must copy the next element from the other.

```
1  def merge(S1, S2, S):
2    """Merge two sorted Python lists S1 and S2 into properly sized list S."""
3    i = j = 0
4    while i + j < len(S):
5      if j == len(S2) or (i < len(S1) and S1[i] < S2[j]):
6        S[i+j] = S1[i]              # copy ith element of S1 as next item of S
7        i += 1
8      else:
9        S[i+j] = S2[j]              # copy jth element of S2 as next item of S
10       j += 1
```

Code Fragment 12.1: An implementation of the merge operation for Python's array-based list class.

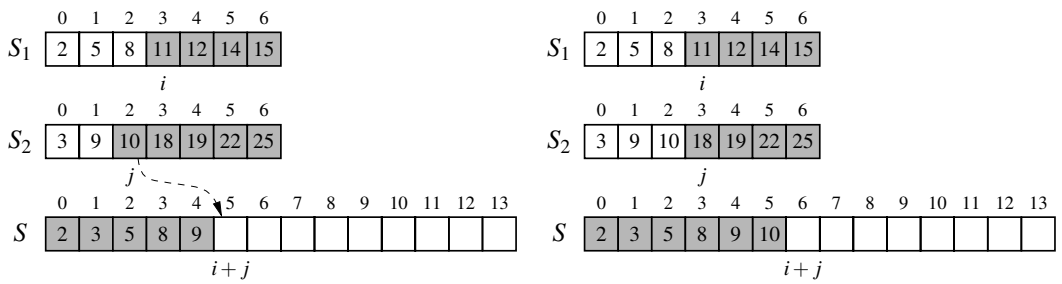


Figure 12.5: A step in the merge of two sorted arrays for which $S_2[j] < S_1[i]$. We show the arrays before the copy step in (a) and after it in (b).

13.1.1 Notations for Strings and the Python str Class

We use character strings as a model for text when discuss algorithms for text processing. Character strings can come from a wide variety of sources, including scientific, linguistic, and Internet applications. Indeed, the following are examples of such strings:

$$S = \texttt{"CGTAAACTGCTTTAATCAAACGC"}$$
$$T = \texttt{"http://www.wiley.com"}$$

The first string, S, comes from DNA applications, and the second string, T, is the Internet address (URL) for the publisher of this book. We refer to Appendix A for an overview of the operations supported by Python's **str** class.

To allow fairly general notions of a string in our algorithm descriptions, we only assume that characters of a string come from a known ***alphabet***, which we denote as Σ. For example, in the context of DNA, there are four symbols in the standard alphabet, $\Sigma = \{A, C, G, T\}$. This alphabet Σ can, of course, be a subset of the ASCII or Unicode character sets, but it could also be something more general. Although we assume that an alphabet has a fixed finite size, denoted as $|\Sigma|$, that size can be nontrivial, as with Python's treatment of the Unicode alphabet, which allows for more than a million distinct characters. We therefore consider the impact of $|\Sigma|$ in our asymptotic analysis of text-processing algorithms.

Several string-processing operations involve breaking large strings into smaller strings. In order to be able to speak about the pieces that result from such operations, we will rely on Python's ***indexing*** and ***slicing*** notations. For the sake of notation, we let S denote a string of length n. In that case, we let S[j] refer to the character at index j for $0 \leq j \leq n-1$. We let notation S[j:k] for $0 \leq j \leq k \leq n$ denote the slice (or ***substring***) of S consisting of characters S[j] up to and including S[k−1], but not S[k]. By this definition, note that substring S[j : j + m] has length m and that substring S[j:j] is trivially the ***null string***, having length 0. In accordance with Python conventions, the substring S[j:k] is also the null string when $k < j$.

In order to distinguish some special kinds of substrings, let us refer to any substring of the form S[0:k] for $0 \leq k \leq n$ as a ***prefix*** of S; such a prefix results in Python when the first index is omitted from slice notation, as in S[:k]. Similarly, any substring of the form S[j:n] for $0 \leq j \leq n$ is a ***suffix*** of S; such a suffix results in Python when the second index is omitted from slice notation, as in S[j:]. For example, if we again take S to be the string of DNA given above, then `"CGTAA"` is a prefix of S, `"CGC"` is a suffix of S, and `"C"` is both a prefix and suffix of S. Note that the null string is a prefix and a suffix of any string.

notation to describe subsequences of a sequence. Slices are described as half-open intervals, with a start index that is included, and a stop index that is excluded. For example, the syntax `data[3:8]` denotes a subsequence including the five indices: $3, 4, 5, 6, 7$. An optional "step" value, possibly negative, can be indicated as a third parameter of the slice. If a start index or stop index is omitted in the slicing notation, it is presumed to designate the respective extreme of the original sequence.

Because lists are mutable, the syntax `s[j] = val` can be used to replace an element at a given index. Lists also support a syntax, **del** `s[j]`, that removes the designated element from the list. Slice notation can also be used to replace or delete a sublist.

The notation `val` **in** `s` can be used for any of the sequences to see if there is an element equivalent to `val` in the sequence. For strings, this syntax can be used to check for a single character or for a larger substring, as with `'amp'` **in** `'example'`.

All sequences define comparison operations based on ***lexicographic order***, performing an element by element comparison until the first difference is found. For example, [5, 6, 9] < [5, 7] because of the entries at index 1. Therefore, the following operations are supported by sequence types:

`s == t`	equivalent (element by element)
`s != t`	not equivalent
`s < t`	lexicographically less than
`s <= t`	lexicographically less than or equal to
`s > t`	lexicographically greater than
`s >= t`	lexicographically greater than or equal to

Operators for Sets and Dictionaries

Sets and frozensets support the following operators:

`key` **in** `s`	containment check
`key` **not in** `s`	non-containment check
`s1 == s2`	s1 is equivalent to s2
`s1 != s2`	s1 is not equivalent to s2
`s1 <= s2`	s1 is subset of s2
`s1 < s2`	s1 is proper subset of s2
`s1 >= s2`	s1 is superset of s2
`s1 > s2`	s1 is proper superset of s2
`s1 | s2`	the union of s1 and s2
`s1 & s2`	the intersection of s1 and s2
`s1 - s2`	the set of elements in s1 but not s2
`s1 ^ s2`	the set of elements in precisely one of s1 or s2

Note well that sets do not guarantee a particular order of their elements, so the comparison operators, such as <, are not lexicographic; rather, they are based on the mathematical notion of a subset. As a result, the comparison operators define

Adding an Entry

The most interesting method of the Scoreboard class is add, which is responsible for considering the addition of a new entry to the scoreboard. Keep in mind that every entry will not necessarily qualify as a high score. If the board is not yet full, any new entry will be retained. Once the board is full, a new entry is only retained if it is strictly better than one of the other scores, in particular, the last entry of the scoreboard, which is the lowest of the high scores.

When a new score is considered, we begin by determining whether it qualifies as a high score. If so, we increase the count of active scores, _n, unless the board is already at full capacity. In that case, adding a new high score causes some other entry to be dropped from the scoreboard, so the overall number of entries remains the same.

To correctly place a new entry within the list, the final task is to shift any inferior scores one spot lower (with the least score being dropped entirely when the scoreboard is full). This process is quite similar to the implementation of the insert method of the list class, as described on pages 204–205. In the context of our scoreboard, there is no need to shift any None references that remain near the end of the array, so the process can proceed as diagrammed in Figure 5.19.

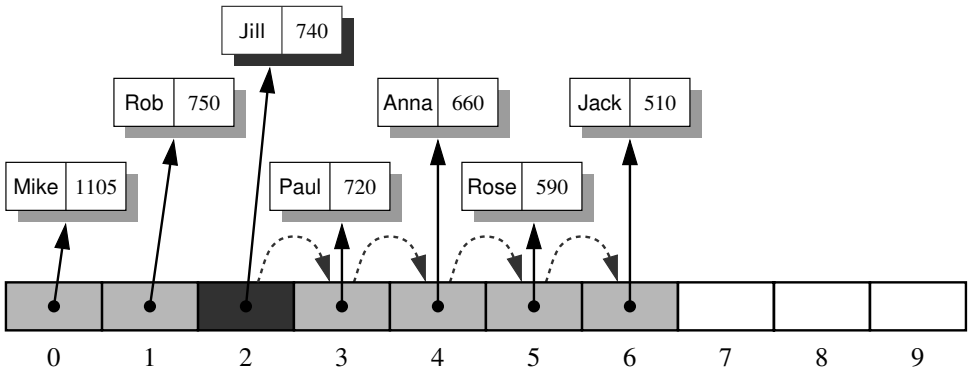


Figure 5.19: Adding a new GameEntry for Jill to the scoreboard. In order to make room for the new reference, we have to shift the references for game entries with smaller scores than the new one to the right by one cell. Then we can insert the new entry with index 2.

To implement the final stage, we begin by considering index j = self._n − 1, which is the index at which the last GameEntry instance will reside, after completing the operation. Either j is the correct index for the newest entry, or one or more immediately before it will have lesser scores. The while loop at line 34 checks the compound condition, shifting references rightward and decrementing j, as long as there is another entry at index $j-1$ with a score less than the new score.

7.6.2 Using a List with the Move-to-Front Heuristic

The previous implementation of a favorites list performs the access(e) method in time proportional to the index of e in the favorites list. That is, if e is the k^{th} most popular element in the favorites list, then accessing it takes $O(k)$ time. In many real-life access sequences (e.g., Web pages visited by a user), once an element is accessed it is more likely to be accessed again in the near future. Such scenarios are said to possess ***locality of reference***.

A ***heuristic***, or rule of thumb, that attempts to take advantage of the locality of reference that is present in an access sequence is the ***move-to-front heuristic***. To apply this heuristic, each time we access an element we move it all the way to the front of the list. Our hope, of course, is that this element will be accessed again in the near future. Consider, for example, a scenario in which we have n elements and the following series of n^2 accesses:

- element 1 is accessed n times
- element 2 is accessed n times
- $\cdots$
- element n is accessed n times.

If we store the elements sorted by their access counts, inserting each element the first time it is accessed, then

- each access to element 1 runs in $O(1)$ time
- each access to element 2 runs in $O(2)$ time
- $\cdots$
- each access to element n runs in $O(n)$ time.

Thus, the total time for performing the series of accesses is proportional to

$$n + 2n + 3n + \cdots + n \cdot n = n(1 + 2 + 3 + \cdots + n) = n \cdot \frac{n(n+1)}{2},$$

which is $O(n^3)$.

On the other hand, if we use the move-to-front heuristic, inserting each element the first time it is accessed, then

- each subsequent access to element 1 takes $O(1)$ time
- each subsequent access to element 2 takes $O(1)$ time
- $\cdots$
- each subsequent access to element n runs in $O(1)$ time.

So the running time for performing all the accesses in this case is $O(n^2)$. Thus, the move-to-front implementation has faster access times for this scenario. Still, the move-to-front approach is just a heuristic, for there are access sequences where using the move-to-front approach is slower than simply keeping the favorites list ordered by access counts.

7.7 Link-Based vs. Array-Based Sequences

We close this chapter by reflecting on the relative pros and cons of array-based and link-based data structures that have been introduced thus far. The dichotomy between these approaches presents a common design decision when choosing an appropriate implementation of a data structure. There is not a one-size-fits-all solution, as each offers distinct advantages and disadvantages.

Advantages of Array-Based Sequences

- *Arrays provide $O(1)$-time access to an element based on an integer index.* The ability to access the k^{th} element for any k in $O(1)$ time is a hallmark advantage of arrays (see Section 5.2). In contrast, locating the k^{th} element in a linked list requires $O(k)$ time to traverse the list from the beginning, or possibly $O(n-k)$ time, if traversing backward from the end of a doubly linked list.
- *Operations with equivalent asymptotic bounds typically run a constant factor more efficiently with an array-based structure versus a linked structure.* As an example, consider the typical enqueue operation for a queue. Ignoring the issue of resizing an array, this operation for the ArrayQueue class (see Code Fragment 6.7) involves an arithmetic calculation of the new index, an increment of an integer, and storing a reference to the element in the array. In contrast, the process for a LinkedQueue (see Code Fragment 7.8) requires the instantiation of a node, appropriate linking of nodes, and an increment of an integer. While this operation completes in $O(1)$ time in either model, the actual number of CPU operations will be more in the linked version, especially given the instantiation of the new node.
- *Array-based representations typically use proportionally less memory than linked structures.* This advantage may seem counterintuitive, especially given that the length of a dynamic array may be longer than the number of elements that it stores. Both array-based lists and linked lists are referential structures, so the primary memory for storing the actual objects that are elements is the same for either structure. What differs is the auxiliary amounts of memory that are used by the two structures. For an array-based container of n elements, a typical worst case may be that a recently resized dynamic array has allocated memory for $2n$ object references. With linked lists, memory must be devoted not only to store a reference to each contained object, but also explicit references that link the nodes. So a singly linked list of length n already requires $2n$ references (an element reference and next reference for each node). With a doubly linked list, there are $3n$ references.

1.7 Exception Handling

Exceptions are unexpected events that occur during the execution of a program. An exception might result from a logical error or an unanticipated situation. In Python, ***exceptions*** (also known as ***errors***) are objects that are ***raised*** (or ***thrown***) by code that encounters an unexpected circumstance. The Python interpreter can also raise an exception should it encounter an unexpected condition, like running out of memory. A raised error may be ***caught*** by a surrounding context that "handles" the exception in an appropriate fashion. If uncaught, an exception causes the interpreter to stop executing the program and to report an appropriate message to the console. In this section, we examine the most common error types in Python, the mechanism for catching and handling errors that have been raised, and the syntax for raising errors from within user-defined blocks of code.

Common Exception Types

Python includes a rich hierarchy of exception classes that designate various categories of errors; Table 1.6 shows many of those classes. The Exception class serves as a base class for most other error types. An instance of the various subclasses encodes details about a problem that has occurred. Several of these errors may be raised in exceptional cases by behaviors introduced in this chapter. For example, use of an undefined identifier in an expression causes a NameError, and errant use of the dot notation, as in foo.bar( ), will generate an AttributeError if object foo does not support a member named bar.

Class	Description
Exception	A base class for most error types
AttributeError	Raised by syntax obj.foo, if obj has no member named foo
EOFError	Raised if "end of file" reached for console or file input
IOError	Raised upon failure of I/O operation (e.g., opening file)
IndexError	Raised if index to sequence is out of bounds
KeyError	Raised if nonexistent key requested for set or dictionary
KeyboardInterrupt	Raised if user types ctrl-C while program is executing
NameError	Raised if nonexistent identifier used
StopIteration	Raised by next(iterator) if no element; see Section 1.8
TypeError	Raised when wrong type of parameter is sent to a function
ValueError	Raised when parameter has invalid value (e.g., sqrt(−5))
ZeroDivisionError	Raised when any division operator used with 0 as divisor

Table 1.6: Common exception classes in Python

4.3.1 Maximum Recursive Depth in Python

Another danger in the misuse of recursion is known as ***infinite recursion***. If each recursive call makes another recursive call, without ever reaching a base case, then we have an infinite series of such calls. This is a fatal error. An infinite recursion can quickly swamp computing resources, not only due to rapid use of the CPU, but because each successive call creates an activation record requiring additional memory. A blatant example of an ill-formed recursion is the following:

```
def fib(n):
  return fib(n)                    # fib(n) equals fib(n)
```

However, there are far more subtle errors that can lead to an infinite recursion. Revisiting our implementation of binary search in Code Fragment 4.3, in the final case (line 17) we make a recursive call on the right portion of the sequence, in particular going from index mid+1 to high. Had that line instead been written as

```
return binary_search(data, target, mid, high)      # note the use of mid
```

this could result in an infinite recursion. In particular, when searching a range of two elements, it becomes possible to make a recursive call on the identical range.

A programmer should ensure that each recursive call is in some way progressing toward a base case (for example, by having a parameter value that decreases with each call). However, to combat against infinite recursions, the designers of Python made an intentional decision to limit the overall number of function activations that can be simultaneously active. The precise value of this limit depends upon the Python distribution, but a typical default value is 1000. If this limit is reached, the Python interpreter raises a RuntimeError with a message, `maximum recursion depth exceeded`.

For many legitimate applications of recursion, a limit of 1000 nested function calls suffices. For example, our binary_search function (Section 4.1.3) has $O(\log n)$ recursive depth, and so for the default recursive limit to be reached, there would need to be 2^{1000} elements (far, far more than the estimated number of atoms in the universe). However, in the next section we discuss several algorithms that have recursive depth proportional to n. Python's artificial limit on the recursive depth could disrupt such otherwise legitimate computations.

Fortunately, the Python interpreter can be dynamically reconfigured to change the default recursive limit. This is done through use of a module named sys, which supports a getrecursionlimit function and a setrecursionlimit. Sample usage of those functions is demonstrated as follows:

```
import sys
old = sys.getrecursionlimit( )       # perhaps 1000 is typical
sys.setrecursionlimit(1000000)       # change to allow 1 million nested calls
```

Advantages of Link-Based Sequences

- *Link-based structures provide worst-case time bounds for their operations.* This is in contrast to the amortized bounds associated with the expansion or contraction of a dynamic array (see Section 5.3).

  When many individual operations are part of a larger computation, and we only care about the total time of that computation, an amortized bound is as good as a worst-case bound precisely because it gives a guarantee on the sum of the time spent on the individual operations.

  However, if data structure operations are used in a real-time system that is designed to provide more immediate responses (e.g., an operating system, Web server, air traffic control system), a long delay caused by a single (amortized) operation may have an adverse effect.

- *Link-based structures support $O(1)$-time insertions and deletions at arbitrary positions.* The ability to perform a constant-time insertion or deletion with the PositionalList class, by using a Position to efficiently describe the location of the operation, is perhaps the most significant advantage of the linked list.

  This is in stark contrast to an array-based sequence. Ignoring the issue of resizing an array, inserting or deleting an element from the end of an array-based list can be done in constant time. However, more general insertions and deletions are expensive. For example, with Python's array-based list class, a call to insert or pop with index k uses $O(n-k+1)$ time because of the loop to shift all subsequent elements (see Section 5.4).

  As an example application, consider a text editor that maintains a document as a sequence of characters. Although users often add characters to the end of the document, it is also possible to use the cursor to insert or delete one or more characters at an arbitrary position within the document. If the character sequence were stored in an array-based sequence (such as a Python list), each such edit operation may require linearly many characters to be shifted, leading to $O(n)$ performance for each edit operation. With a linked-list representation, an arbitrary edit operation (insertion or deletion of a character at the cursor) can be performed in $O(1)$ worst-case time, assuming we are given a position that represents the location of the cursor.

The level numbering function f suggests a representation of a binary tree T by means of an array-based structure A (such as a Python list), with the element at position p of T stored at index $f(p)$ of the array. We show an example of an array-based representation of a binary tree in Figure 8.13.

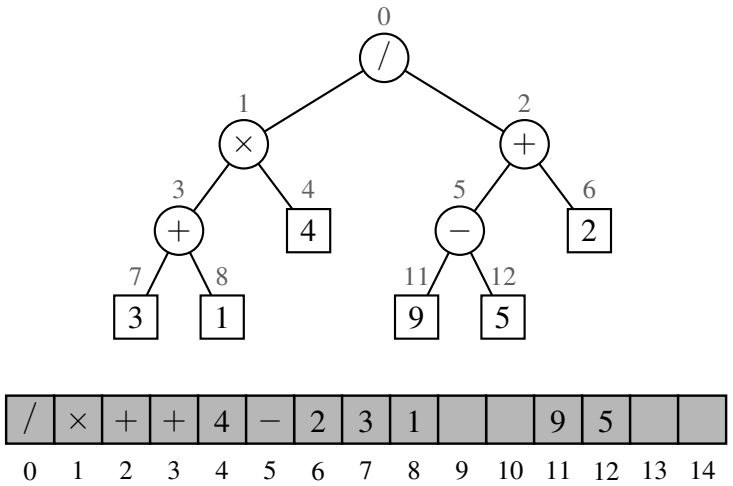


Figure 8.13: Representation of a binary tree by means of an array.

One advantage of an array-based representation of a binary tree is that a position p can be represented by the single integer $f(p)$, and that position-based methods such as root, parent, left, and right can be implemented using simple arithmetic operations on the number $f(p)$. Based on our formula for the level numbering, the left child of p has index $2f(p)+1$, the right child of p has index $2f(p)+2$, and the parent of p has index $\lfloor (f(p)-1)/2 \rfloor$. We leave the details of a complete implementation as an exercise (R-8.18).

The space usage of an array-based representation depends greatly on the shape of the tree. Let n be the number of nodes of T, and let f_M be the maximum value of $f(p)$ over all the nodes of T. The array A requires length $N = 1 + f_M$, since elements range from $A[0]$ to $A[f_M]$. Note that A may have a number of empty cells that do not refer to existing nodes of T. In fact, in the worst case, $N = 2^n - 1$, the justification of which is left as an exercise (R-8.16). In Section 9.3, we will see a class of binary trees, called "heaps" for which $N = n$. Thus, in spite of the worst-case space usage, there are applications for which the array representation of a binary tree is space efficient. Still, for general binary trees, the exponential worst-case space requirement of this representation is prohibitive.

Another drawback of an array representation is that some update operations for trees cannot be efficiently supported. For example, deleting a node and promoting its child takes $O(n)$ time because it is not just the child that moves locations within the array, but all descendants of that child.

9.4.2 Heap-Sort

As we have previously observed, realizing a priority queue with a heap has the advantage that all the methods in the priority queue ADT run in logarithmic time or better. Hence, this realization is suitable for applications where fast running times are sought for all the priority queue methods. Therefore, let us again consider the pq_sort scheme, this time using a heap-based implementation of the priority queue.

During Phase 1, the i^{th} add operation takes $O(\log i)$ time, since the heap has i entries after the operation is performed. Therefore this phase takes $O(n \log n)$ time. (It could be improved to $O(n)$ with the bottom-up heap construction described in Section 9.3.6.)

During the second phase of pq_sort, the j^{th} remove_min operation runs in $O(\log(n-j+1))$, since the heap has $n-j+1$ entries at the time the operation is performed. Summing over all j, this phase takes $O(n \log n)$ time, so the entire priority-queue sorting algorithm runs in $O(n \log n)$ time when we use a heap to implement the priority queue. This sorting algorithm is better known as ***heap-sort***, and its performance is summarized in the following proposition.

Proposition 9.4: *The heap-sort algorithm sorts a collection C of n elements in $O(n \log n)$ time, assuming two elements of C can be compared in $O(1)$ time.*

Let us stress that the $O(n \log n)$ running time of heap-sort is considerably better than the $O(n^2)$ running time of selection-sort and insertion-sort (Section 9.4.1).

Implementing Heap-Sort In-Place

If the collection C to be sorted is implemented by means of an array-based sequence, most notably as a Python list, we can speed up heap-sort and reduce its space requirement by a constant factor using a portion of the list itself to store the heap, thus avoiding the use of an auxiliary heap data structure. This is accomplished by modifying the algorithm as follows:

1. We redefine the heap operations to be a *maximum-oriented* heap, with each position's key being at least as *large* as its children. This can be done by recoding the algorithm, or by adjusting the notion of keys to be negatively oriented. At any time during the execution of the algorithm, we use the left portion of C, up to a certain index $i-1$, to store the entries of the heap, and the right portion of C, from index i to $n-1$, to store the elements of the sequence. Thus, the first i elements of C (at indices $0,\ldots,i-1$) provide the array-list representation of the heap.
2. In the first phase of the algorithm, we start with an empty heap and move the boundary between the heap and the sequence from left to right, one step at a time. In step i, for $i = 1,\ldots,n$, we expand the heap by adding the element at index $i-1$.

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 efficient 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 defines 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 five 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__.

Searching for Occurrences of a Value

Each of the count, index, and __contains__ methods proceed through iteration of the sequence from left to right. In fact, Code Fragment 2.14 of Section 2.4.3 demonstrates how those behaviors might be implemented. Notably, the loop for computing the count must proceed through the entire sequence, while the loops for checking containment of an element or determining the index of an element immediately exit once they find the leftmost occurrence of the desired value, if one exists. So while count always examines the n elements of the sequence, index and __contains__ examine n elements in the worst case, but may be faster. Empirical evidence can be found by setting data = list(range(10000000)) and then comparing the relative efficiency of the test, 5 **in** data, relative to the test, 9999995 **in** data, or even the failed test, -5 **in** data.

Lexicographic Comparisons

Comparisons between two sequences are defined lexicographically. In the worst case, evaluating such a condition requires an iteration taking time proportional to the length of the *shorter* of the two sequences (because when one sequence ends, the lexicographic result can be determined). However, in some cases the result of the test can be evaluated more efficiently. For example, if evaluating [7, 3, ...] $<$ [7, 5, ...], it is clear that the result is True without examining the remainders of those lists, because the second element of the left operand is strictly less than the second element of the right operand.

Creating New Instances

The final three behaviors in Table 5.3 are those that construct a new instance based on one or more existing instances. In all cases, the running time depends on the construction and initialization of the new result, and therefore the asymptotic behavior is proportional to the *length* of the result. Therefore, we find that slice data[6000000:6000008] can be constructed almost immediately because it has only eight elements, while slice data[6000000:7000000] has one million elements, and thus is more time-consuming to create.

Mutating Behaviors

The efficiency of the mutating behaviors of the list class are described in Table 5.3. The simplest of those behaviors has syntax data[j] = val, and is supported by the special __setitem__ method. This operation has worst-case $O(1)$ running time because it simply replaces one element of a list with a new value. No other elements are affected and the size of the underlying array does not change. The more interesting behaviors to analyze are those that add or remove elements from the list.

Operation	Running Time
data[j] = val	$O(1)$
data.append(value)	$O(1)^*$
data.insert(k, value)	$O(n-k+1)^*$
data.pop( )	$O(1)^*$
data.pop(k) **del** data[k]	$O(n-k)^*$
data.remove(value)	$O(n)^*$
data1.extend(data2) data1 += data2	$O(n_2)^*$
data.reverse( )	$O(n)$
data.sort( )	$O(n \log n)$

*amortized

Table 5.4: Asymptotic performance of the mutating behaviors of the list class. Identifiers data, data1, and data2 designate instances of the list class, and n, n_1, and n_2 their respective lengths.

Adding Elements to a List

In Section 5.3 we fully explored the append method. In the worst case, it requires $\Omega(n)$ time because the underlying array is resized, but it uses $O(1)$ time in the amortized sense. Lists also support a method, with signature insert(k, value), that inserts a given value into the list at index $0 \le k \le n$ while shifting all subsequent elements back one slot to make room. For the purpose of illustration, Code Fragment 5.5 provides an implementation of that method, in the context of our DynamicArray class introduced in Code Fragment 5.3. There are two complicating factors in analyzing the efficiency of such an operation. First, we note that the addition of one element may require a resizing of the dynamic array. That portion of the work requires $\Omega(n)$ worst-case time but only $O(1)$ amortized time, as per append. The other expense for insert is the shifting of elements to make room for the new item. The time for

```
1  def insert(self, k, value):
2    """Insert value at index k, shifting subsequent values rightward."""
3    # (for simplicity, we assume 0 <= k <= n in this verion)
4    if self._n == self._capacity:                # not enough room
5      self._resize(2 * self._capacity)           # so double capacity
6    for j in range(self._n, k, -1):              # shift rightmost first
7      self._A[j] = self._A[j-1]
8    self._A[k] = value                           # store newest element
9    self._n += 1
```

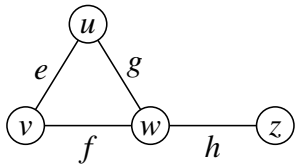
Code Fragment 5.5: Implementation of insert for our DynamicArray class.

14.2.4 Adjacency Matrix Structure

The ***adjacency matrix*** structure for a graph G augments the edge list structure with a matrix A (that is, a two-dimensional array, as in Section 5.6), which allows us to locate an edge between a given pair of vertices in *worst-case* constant time. In the adjacency matrix representation, we think of the vertices as being the integers in the set $\{0,1,\ldots,n-1\}$ and the edges as being pairs of such integers. This allows us to store references to edges in the cells of a two-dimensional $n \times n$ array A. Specifically, the cell $A[i,j]$ holds a reference to the edge (u,v), if it exists, where u is the vertex with index i and v is the vertex with index j. If there is no such edge, then $A[i,j] = \text{None}$. We note that array A is symmetric if graph G is undirected, as $A[i,j] = A[j,i]$ for all pairs i and j. (See Figure 14.7.)

The most significant advantage of an adjacency matrix is that any edge (u,v) can be accessed in worst-case $O(1)$ time; recall that the adjacency map supports that operation in $O(1)$ *expected* time. However, several operation are less efficient with an adjacency matrix. For example, to find the edges incident to vertex v, we must presumably examine all n entries in the row associated with v; recall that an adjacency list or map can locate those edges in optimal $O(\deg(v))$ time. Adding or removing vertices from a graph is problematic, as the matrix must be resized.

Furthermore, the $O(n^2)$ space usage of an adjacency matrix is typically far worse than the $O(n+m)$ space required of the other representations. Although, in the worst case, the number of edges in a ***dense*** graph will be proportional to n^2, most real-world graphs are ***sparse***. In such cases, use of an adjacency matrix is inefficient. However, if a graph is dense, the constants of proportionality of an adjacency matrix can be smaller than that of an adjacency list or map. In fact, if edges do not have auxiliary data, a Boolean adjacency matrix can use one bit per edge slot, such that $A[i,j] = \text{True}$ if and only if associated (u,v) is an edge.



(a)

		0	1	2	3
u	0		e	g	
v	1	e		f	
w	2	g	f		h
z	3			h	

(b)

Figure 14.7: (a) An undirected graph G; (b) a schematic representation of the auxiliary adjacency matrix structure for G, in which n vertices are mapped to indices 0 to $n-1$. Although not diagrammed as such, we presume that there is a unique Edge instance for each edge, and that it maintains references to its endpoint vertices. We also assume that there is a secondary edge list (not pictured), to allow the edges( ) method to run in $O(m)$ time, for a graph with m edges.

2.3.4 Iterators

Iteration is an important concept in the design of data structures. We introduced Python's mechanism for iteration in Section 1.8. In short, an ***iterator*** for a collection provides one key behavior: It supports a special method named `__next__` that returns the next element of the collection, if any, or raises a `StopIteration` exception to indicate that there are no further elements.

Fortunately, it is rare to have to directly implement an iterator class. Our preferred approach is the use of the ***generator*** syntax (also described in Section 1.8), which automatically produces an iterator of yielded values.

Python also helps by providing an automatic iterator implementation for any class that defines both `__len__` and `__getitem__`. To provide an instructive example of a low-level iterator, Code Fragment 2.5 demonstrates just such an iterator class that works on any collection that supports both `__len__` and `__getitem__`. This class can be instantiated as `SequenceIterator(data)`. It operates by keeping an internal reference to the data sequence, as well as a current index into the sequence. Each time `__next__` is called, the index is incremented, until reaching the end of the sequence.

```
1  class SequenceIterator:
2    """An iterator for any of Python's sequence types."""
3
4    def __init__(self, sequence):
5      """Create an iterator for the given sequence."""
6      self._seq = sequence              # keep a reference to the underlying data
7      self._k = -1                      # will increment to 0 on first call to next
8
9    def __next__(self):
10     """Return the next element, or else raise StopIteration error."""
11     self._k += 1                      # advance to next index
12     if self._k < len(self._seq):
13       return(self._seq[self._k])      # return the data element
14     else:
15       raise StopIteration( )          # there are no more elements
16
17   def __iter__(self):
18     """By convention, an iterator must return itself as an iterator."""
19     return self
```

Code Fragment 2.5: An iterator class for any sequence type.

2.4.3 Abstract Base Classes

When defining a group of classes as part of an inheritance hierarchy, one technique for avoiding repetition of code is to design a base class with common functionality that can be inherited by other classes that need it. As an example, the hierarchy from Section 2.4.2 includes a Progression class, which serves as a base class for three distinct subclasses: ArithmeticProgression, GeometricProgression, and FibonacciProgression. Although it is possible to create an instance of the Progression base class, there is little value in doing so because its behavior is simply a special case of an ArithmeticProgression with increment 1. The real purpose of the Progression class was to centralize the implementations of behaviors that other progressions needed, thereby streamlining the code that is relegated to those subclasses.

In classic object-oriented terminology, we say a class is an ***abstract base class*** if its only purpose is to serve as a base class through inheritance. More formally, an abstract base class is one that cannot be directly instantiated, while a ***concrete class*** is one that can be instantiated. By this definition, our Progression class is technically concrete, although we essentially designed it as an abstract base class.

In statically typed languages such as Java and C++, an abstract base class serves as a formal type that may guarantee one or more ***abstract methods***. This provides support for polymorphism, as a variable may have an abstract base class as its declared type, even though it refers to an instance of a concrete subclass. Because there are no declared types in Python, this kind of polymorphism can be accomplished without the need for a unifying abstract base class. For this reason, there is not as strong a tradition of defining abstract base classes in Python, although Python's abc module provides support for defining a formal abstract base class.

Our reason for focusing on abstract base classes in our study of data structures is that Python's collections module provides several abstract base classes that assist when defining custom data structures that share a common interface with some of Python's built-in data structures. These rely on an object-oriented software design pattern known as the ***template method pattern***. The template method pattern is when an abstract base class provides concrete behaviors that rely upon calls to other abstract behaviors. In that way, as soon as a subclass provides definitions for the missing abstract behaviors, the inherited concrete behaviors are well defined.

As a tangible example, the collections.Sequence abstract base class defines behaviors common to Python's list, str, and tuple classes, as sequences that support element access via an integer index. More so, the collections.Sequence class provides concrete implementations of methods, count, index, and __contains__ that can be inherited by any class that provides concrete implementations of both __len__ and __getitem__. For the purpose of illustration, we provide a sample implementation of such a Sequence abstract base class in Code Fragment 2.14.

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.

A preferred approach to producing an indented table of contents is to redesign a top-down recursion that includes the current depth as an additional parameter. Such an implementation is provided in Code Fragment 8.23. This implementation runs in worst-case $O(n)$ time (except, technically, the time it takes to print strings of increasing lengths).

```
1  def preorder_indent(T, p, d):
2    """Print preorder representation of subtree of T rooted at p at depth d."""
3    print(2*d*' ' + str(p.element( )))          # use depth for indentation
4    for c in T.children(p):
5      preorder_indent(T, c, d+1)                # child depth is d+1
```

Code Fragment 8.23: Efficient recursion for printing indented version of a preorder traversal. On a complete tree T, the recursion should be started with form preorder_indent(T, T.root( ), 0).

In the example of Figure 8.20, we were fortunate in that the numbering was embedded within the elements of the tree. More generally, we might be interested in using a preorder traversal to display the structure of a tree, with indentation and also explicit numbering that was not present in the tree. For example, we might display the tree from Figure 8.2 beginning as:

```
Electronics R'Us
  1 R&D
  2 Sales
    2.1 Domestic
    2.2 International
      2.2.1 Canada
      2.2.2 S. America
```

This is more challenging, because the numbers used as labels are implicit in the structure of the tree. A label depends on the index of each position, relative to its siblings, along the path from the root to the current position. To accomplish the task, we add a representation of that path as an additional parameter to the recursive signature. Specifically, we use a list of zero-indexed numbers, one for each position along the downward path, other than the root. (We convert those numbers to one-indexed form when printing.)

At the implementation level, we wish to avoid the inefficiency of duplicating such lists when sending a new parameter from one level of the recursion to the next. A standard solution is to share the same list instance throughout the recursion. At one level of the recursion, a new entry is temporarily added to the end of the list before making further recursive calls. In order to "leave no trace," that same block of code must remove the extraneous entry from the list before completing its task. An implementation based on this approach is given in Code Fragment 8.24.

Analysis

We conclude by analyzing the performance of our SortedTableMap implementation. A summary of the running times for all methods of the sorted map ADT (including the traditional map operations) is given in Table 10.3. It should be clear that the __len__, find_min, and find_max methods run in $O(1)$ time, and that iterating the keys of the table in either direction can be peformed in $O(n)$ time.

The analysis for the various forms of search all depend on the fact that a binary search on a table with n entries runs in $O(\log n)$ time. This claim was originally shown as Proposition 4.2 in Section 4.2, and that analysis clearly applies to our _find_index method as well. We therefore claim an $O(\log n)$ worst-case running time for methods __getitem__, find_lt, find_gt, find_le, and find_ge. Each of these makes a single call to _find_index, followed by a constant number of additional steps to determine the appropriate answer based on the index. The analysis of find_range is a bit more interesting. It begins with a binary search to find the first item within the range (if any). After that, it executes a loop that takes $O(1)$ time per iteration to report subsequent values until reaching the end of the range. If there are s items reported in the range, the total running time is $O(s + \log n)$.

In contrast to the efficient search operations, update operations for a sorted table may take considerable time. Although binary search can help identify the index at which an update occurs, both insertions and deletions require, in the worst case, that linearly many existing elements be shifted in order to maintain the sorted order of the table. Specifically, the potential call to _table.insert from within __setitem__ and _table.pop from within __delitem__ lead to $O(n)$ worst-case time. (See the discussion of corresponding operations of the list class in Section 5.4.1.)

In conclusion, sorted tables are primarily used in situations where we expect many searches but relatively few updates.

Operation	Running Time
len(M)	$O(1)$
k in M	$O(\log n)$
M[k] = v	$O(n)$ worst case; $O(\log n)$ if existing k
del M[k]	$O(n)$ worst case
M.find_min( ), M.find_max( )	$O(1)$
M.find_lt(k), M.find_gt(k) M.find_le(k), M.find_ge(k)	$O(\log n)$
M.find_range(start, stop)	$O(s + \log n)$ where s items are reported
iter(M), reversed(M)	$O(n)$

Table 10.3: Performance of a sorted map, as implemented with SortedTableMap. We use n to denote the number of items in the map at the time the operation is performed. The space requirement is $O(n)$.

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

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 $h(k) = 3k \bmod 17$.

R-10.15 Our HashMapBase class maintains a load factor $\lambda \leq 0.5$. 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 _find_index method from Code Fragment 10.8, in the context of the SortedTableMap class:

```
def _find_index(self, k, low, high):
  if high < low:
    return high + 1
  else:
    mid = (low + high) // 2
    if self._table[mid]._key < k:
      return self._find_index(k, mid + 1, high)
    else:
      return self._find_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 flips, as well.

divide step is performed by scanning the array simultaneously using local variables left, which advances forward, and right, which advances backward, swapping pairs of elements that are in reverse order, as shown in Figure 12.14. When these two indices pass each other, the division step is complete and the algorithm completes by recurring on these two sublists. There is no explicit "combine" step, because the concatenation of the two sublists is implicit to the in-place use of the original list.

It is worth noting that if a sequence has duplicate values, we are not explicitly creating three sublists L, E, and G, as in our original quick-sort description. We instead allow elements equal to the pivot (other than the pivot itself) to be dispersed across the two sublists. Exercise R-12.11 explores the subtlety of our implementation in the presence of duplicate keys, and Exercise C-12.33 describes an in-place algorithm that strictly partitions into three sublists L, E, and G.

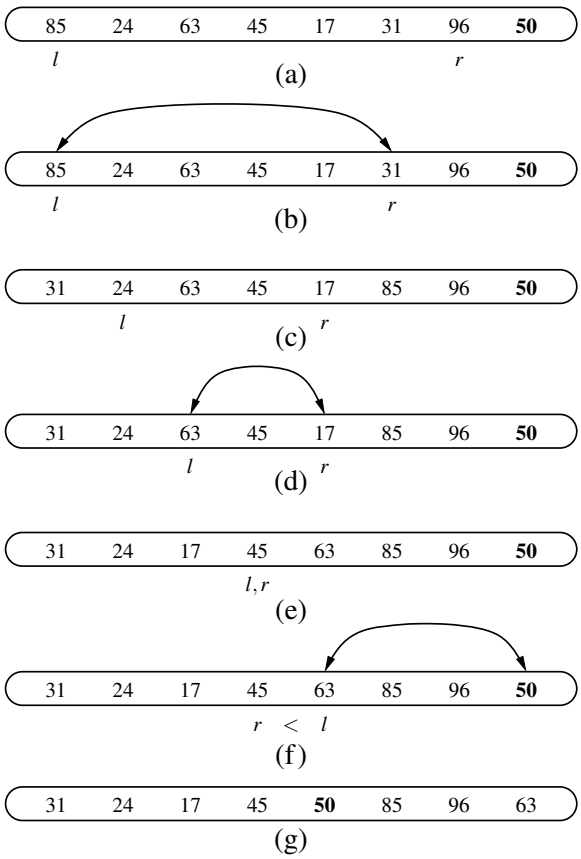


Figure 12.14: Divide step of in-place quick-sort, using index l as shorthand for identifier left, and index r as shorthand for identifier right. Index l scans the sequence from left to right, and index r scans the sequence from right to left. A swap is performed when l is at an element as large as the pivot and r is at an element as small as the pivot. A final swap with the pivot, in part (f), completes the divide step.

A common tool for developing an initial high-level design for a project is the use of ***CRC cards***. Class-Responsibility-Collaborator (CRC) cards are simple index cards that subdivide the work required of a program. The main idea behind this tool is to have each card represent a component, which will ultimately become a class in the program. We write the name of each component on the top of an index card. On the left-hand side of the card, we begin writing the responsibilities for this component. On the right-hand side, we list the collaborators for this component, that is, the other components that this component will have to interact with to perform its duties.

The design process iterates through an action/actor cycle, where we first identify an action (that is, a responsibility), and we then determine an actor (that is, a component) that is best suited to perform that action. The design is complete when we have assigned all actions to actors. In using index cards for this process (rather than larger pieces of paper), we are relying on the fact that each component should have a small set of responsibilities and collaborators. Enforcing this rule helps keep the individual classes manageable.

As the design takes form, a standard approach to explain and document the design is the use of UML (Unified Modeling Language) diagrams to express the organization of a program. UML diagrams are a standard visual notation to express object-oriented software designs. Several computer-aided tools are available to build UML diagrams. One type of UML figure is known as a ***class diagram***. An example of such a diagram is given in Figure 2.3, for a class that represents a consumer credit card. The diagram has three portions, with the first designating the name of the class, the second designating the recommended instance variables, and the third designating the recommended methods of the class. In Section 2.2.3, we discuss our naming conventions, and in Section 2.3.1, we provide a complete implementation of a Python CreditCard class based on this design.

Class:	CreditCard	
Fields:	_customer _bank _account	_balance _limit
Behaviors:	get_customer( ) get_bank( ) get_account( ) make_payment(amount)	get_balance( ) get_limit( ) charge(price)

Figure 2.3: Class diagram for a proposed CreditCard class.

The second advanced technique is the use of the @abstractmethod decorator immediately before the __len__ and __getitem__ methods are declared. That declares these two particular methods to be abstract, meaning that we do not provide an implementation within our Sequence base class, but that we expect any concrete subclasses to support those two methods. Python enforces this expectation, by disallowing instantiation for any subclass that does not override the abstract methods with concrete implementations.

The rest of the Sequence class definition provides tangible implementations for other behaviors, under the assumption that the abstract __len__ and __getitem__ methods will exist in a concrete subclass. If you carefully examine the source code, the implementations of methods __contains__, index, and count do not rely on any assumption about the self instances, other than that syntax len(self) and self[j] are supported (by special methods __len__ and __getitem__, respectively). Support for iteration is automatic as well, as described in Section 2.3.4.

In the remainder of this book, we omit the formality of using the abc module. If we need an "abstract" base class, we simply document the expectation that subclasses provide assumed functionality, without technical declaration of the methods as abstract. But we will make use of the wonderful abstract base classes that are defined within the collections module (such as Sequence). To use such a class, we need only rely on standard inheritance techniques.

For example, our Range class, from Code Fragment 2.6 of Section 2.3.5, is an example of a class that supports the __len__ and __getitem__ methods. But that class does not support methods count or index. Had we originally declared it with Sequence as a superclass, then it would also inherit the count and index methods. The syntax for such a declaration would begin as:

```
class Range(collections.Sequence):
```

Finally, we emphasize that if a subclass provides its own implementation of an inherited behaviors from a base class, the new definition overrides the inherited one. This technique can be used when we have the ability to provide a more efficient implementation for a behavior than is achieved by the generic approach. As an example, the general implementation of __contains__ for a sequence is based on a loop used to search for the desired value. For our Range class, there is an opportunity for a more efficient determination of containment. For example, it is evident that the expression, 100000 **in** Range(0, 2000000, 100), should evaluate to True, even without examining the individual elements of the range, because the range starts with zero, has an increment of 100, and goes until 2 million; it must include 100000, as that is a multiple of 100 that is between the start and stop values. Exercise C-2.27 explores the goal of providing an implementation of Range.__contains__ that avoids the use of a (time-consuming) loop.

```
1  import ctypes                                    # provides low-level arrays
2
3  class DynamicArray:
4    """A dynamic array class akin to a simplified Python list."""
5
6    def __init__(self):
7      """Create an empty array."""
8      self._n = 0                                  # count actual elements
9      self._capacity = 1                           # default array capacity
10     self._A = self._make_array(self._capacity)   # low-level array
11
12   def __len__(self):
13     """Return number of elements stored in the array."""
14     return self._n
15
16   def __getitem__(self, k):
17     """Return element at index k."""
18     if not 0 <= k < self._n:
19       raise IndexError('invalid index')
20     return self._A[k]                            # retrieve from array
21
22   def append(self, obj):
23     """Add object to end of the array."""
24     if self._n == self._capacity:                # not enough room
25       self._resize(2 * self._capacity)           # so double capacity
26     self._A[self._n] = obj
27     self._n += 1
28
29   def _resize(self, c):                          # nonpublic utitity
30     """Resize internal array to capacity c."""
31     B = self._make_array(c)                      # new (bigger) array
32     for k in range(self._n):                     # for each existing value
33       B[k] = self._A[k]
34     self._A = B                                  # use the bigger array
35     self._capacity = c
36
37   def _make_array(self, c):                      # nonpublic utitity
38     """Return new array with capacity c."""
39     return (c * ctypes.py_object)( )             # see ctypes documentation
```

Code Fragment 5.3: An implementation of a DynamicArray class, using a raw array from the ctypes module as storage.

```
1  class Scoreboard:
2    """Fixed-length sequence of high scores in nondecreasing order."""
3
4    def __init__(self, capacity=10):
5      """Initialize scoreboard with given maximum capacity.
6
7      All entries are initially None.
8      """
9      self._board = [None] * capacity          # reserve space for future scores
10     self._n = 0                              # number of actual entries
11
12   def __getitem__(self, k):
13     """Return entry at index k."""
14     return self._board[k]
15
16   def __str__(self):
17     """Return string representation of the high score list."""
18     return '\n'.join(str(self._board[j]) for j in range(self._n))
19
20   def add(self, entry):
21     """Consider adding entry to high scores."""
22     score = entry.get_score( )
23
24     # Does new entry qualify as a high score?
25     # answer is yes if board not full or score is higher than last entry
26     good = self._n < len(self._board) or score > self._board[-1].get_score( )
27
28     if good:
29       if self._n < len(self._board):         # no score drops from list
30         self._n += 1                         # so overall number increases
31
32       # shift lower scores rightward to make room for new entry
33       j = self._n - 1
34       while j > 0 and self._board[j-1].get_score( ) < score:
35         self._board[j] = self._board[j-1]    # shift entry from j-1 to j
36         j -= 1                               # and decrement j
37       self._board[j] = entry                 # when done, add new entry
```

Code Fragment 5.8: Python code for a Scoreboard class that maintains an ordered series of scores as GameEntry objects.

```
1  def insertion_sort(A):
2    """Sort list of comparable elements into nondecreasing order."""
3    for k in range(1, len(A)):                # from 1 to n-1
4      cur = A[k]                              # current element to be inserted
5      j = k                                   # find correct index j for current
6      while j > 0 and A[j−1] > cur:           # element A[j-1] must be after current
7        A[j] = A[j−1]
8        j −= 1
9      A[j] = cur                              # cur is now in the right place
```

Code Fragment 5.10: Python code for performing insertion-sort on a list.

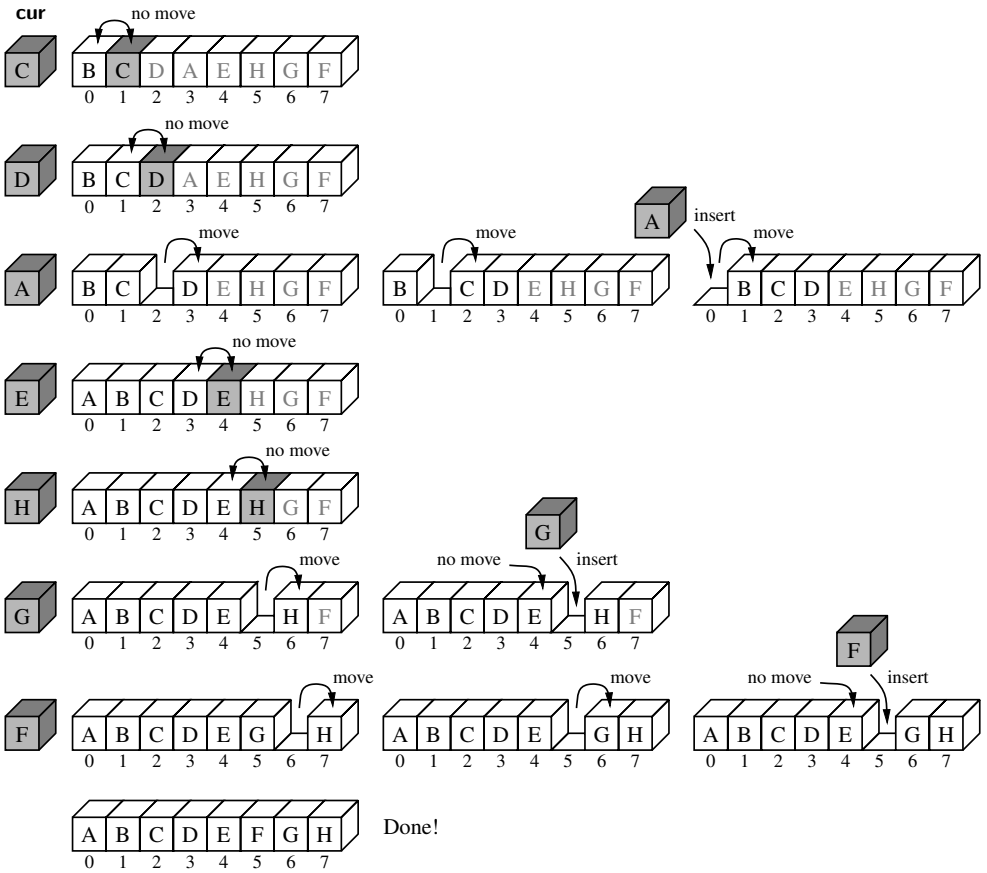


Figure 5.20: Execution of the insertion-sort algorithm on an array of eight characters. Each row corresponds to an iteration of the outer loop, and each copy of the sequence in a row corresponds to an iteration of the inner loop. The current element that is being inserted is highlighted in the array, and shown as the cur value.

We can represent a replacement rule using another string to describe the translation. As a concrete example, suppose we are using a Caesar cipher with a three-character rotation. We can precompute a string that represents the replacements that should be used for each character from A to Z. For example, A should be replaced by D, B replaced by E, and so on. The 26 replacement characters in order are `'DEFGHIJKLMNOPQRSTUVWXYZABC'`. We can subsequently use this translation string as a guide to encrypt a message. The remaining challenge is how to quickly locate the replacement for each character of the original message.

Fortunately, we can rely on the fact that characters are represented in Unicode by integer code points, and the code points for the uppercase letters of the Latin alphabet are consecutive (for simplicity, we restrict our encryption to uppercase letters). Python supports functions that convert between integer code points and one-character strings. Specifically, the function ord(c) takes a one-character string as a parameter and returns the integer code point for that character. Conversely, the function chr(j) takes an integer and returns its associated one-character string.

In order to find a replacement for a character in our Caesar cipher, we need to map the characters `'A'` to `'Z'` to the respective numbers 0 to 25. The formula for doing that conversion is j = ord(c) − ord(`'A'`). As a sanity check, if character c is `'A'`, we have that $j = 0$. When c is `'B'`, we will find that its ordinal value is precisely one more than that for `'A'`, so their difference is 1. In general, the integer j that results from such a calculation can be used as an index into our precomputed translation string, as illustrated in Figure 5.21.

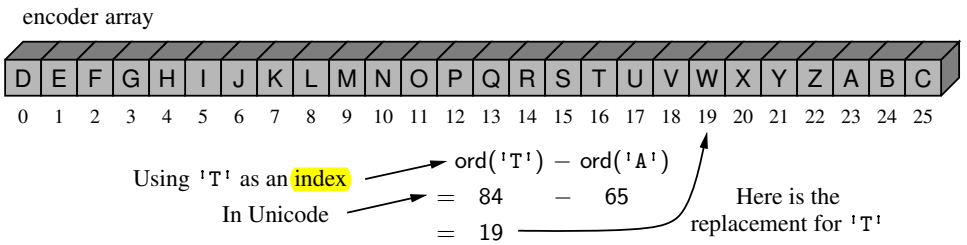


Figure 5.21: Illustrating the use of uppercase characters as indices, in this case to perform the replacement rule for Caesar cipher encryption.

In Code Fragment 5.11, we develop a Python class for performing the Caesar cipher with an arbitrary rotational shift, and demonstrate its use. When we run this program (to perform a simple test), we get the following output.

```
Secret:  WKH HDJOH LV LQ SODB; PHHW DW MRH'V.
Message: THE EAGLE IS IN PLAY; MEET AT JOE'S.
```

The constructor for the class builds the forward and backward translation strings for the given rotation. With those in hand, the encryption and decryption algorithms are essentially the same, and so we perform both by means of a nonpublic utility method named _transform.

In an HTML document, portions of text are delimited by ***HTML tags***. A simple opening HTML tag has the form "<name>" and the corresponding closing tag has the form "</name>". For example, we see the <body> tag on the first line of Figure 6.3(a), and the matching </body> tag at the close of that document. Other commonly used HTML tags that are used in this example include:

- body: document body
- h1: section header
- center: center justify
- p: paragraph
- ol: numbered (ordered) list
- li: list item

Ideally, an HTML document should have matching tags, although most browsers tolerate a certain number of mismatching tags. In Code Fragment 6.5, we give a Python function that matches tags in a string representing an HTML document. We make a left-to-right pass through the raw string, using index j to track our progress and the find method of the str class to locate the '<' and '>' characters that define the tags. Opening tags are pushed onto the stack, and matched against closing tags as they are popped from the stack, just as we did when matching delimiters in Code Fragment 6.4. By similar analysis, this algorithm runs in $O(n)$ time, where n is the number of characters in the raw HTML source.

```
1  def is_matched_html(raw):
2    """Return True if all HTML tags are properly match; False otherwise."""
3    S = ArrayStack( )
4    j = raw.find('<')                    # find first '<' character (if any)
5    while j != −1:
6      k = raw.find('>', j+1)             # find next '>' character
7      if k == −1:
8        return False                     # invalid tag
9      tag = raw[j+1:k]                   # strip away < >
10     if not tag.startswith('/'):        # this is opening tag
11       S.push(tag)
12     else:                              # this is closing tag
13       if S.is_empty( ):
14         return False                   # nothing to match with
15       if tag[1:] != S.pop( ):
16         return False                   # mismatched delimiter
17     j = raw.find('<', k+1)             # find next '<' character (if any)
18   return S.is_empty( )                 # were all opening tags matched?
```

Code Fragment 6.5: Function for testing if an HTML document has matching tags.

A Bottom-Up (Nonrecursive) Merge-Sort

There is a nonrecursive version of array-based merge-sort, which runs in $O(n \log n)$ time. It is a bit faster than recursive merge-sort in practice, as it avoids the extra overheads of recursive calls and temporary memory at each level. The main idea is to perform merge-sort bottom-up, performing the merges level by level going up the merge-sort tree. Given an input array of elements, we begin by merging every successive pair of elements into sorted runs of length two. We merge these runs into runs of length four, merge these new runs into runs of length eight, and so on, until the array is sorted. To keep the space usage reasonable, we deploy a second array that stores the merged runs (swapping input and output arrays after each iteration). We give a Python implementation in Code Fragment 12.4. A similar bottom-up approach can be used for sorting linked lists. (See Exercise C-12.29.)

```
1  def merge(src, result, start, inc):
2    """Merge src[start:start+inc] and src[start+inc:start+2*inc] into result."""
3    end1 = start+inc                    # boundary for run 1
4    end2 = min(start+2*inc, len(src))   # boundary for run 2
5    x, y, z = start, start+inc, start   # index into run 1, run 2, result
6    while x < end1 and y < end2:
7      if src[x] < src[y]:
8        result[z] = src[x]; x += 1      # copy from run 1 and increment x
9      else:
10       result[z] = src[y]; y += 1      # copy from run 2 and increment y
11     z += 1                            # increment z to reflect new result
12   if x < end1:
13     result[z:end2] = src[x:end1]      # copy remainder of run 1 to output
14   elif y < end2:
15     result[z:end2] = src[y:end2]      # copy remainder of run 2 to output
16
17 def merge_sort(S):
18   """Sort the elements of Python list S using the merge-sort algorithm."""
19   n = len(S)
20   logn = math.ceil(math.log(n,2))
21   src, dest = S, [None] * n           # make temporary storage for dest
22   for i in (2**k for k in range(logn)):  # pass i creates all runs of length 2i
23     for j in range(0, n, 2*i):        # each pass merges two length i runs
24       merge(src, dest, j, i)
25     src, dest = dest, src             # reverse roles of lists
26   if S is not src:
27     S[0:n] = src[0:n]                 # additional copy to get results to S
```

Code Fragment 12.4: An implementation of the nonrecursive merge-sort algorithm.

Example 6.5: *The following table shows a series of operations and their effects on an initially empty deque D of integers.*

Operation	Return Value	Deque
D.add_last(5)	–	[5]
D.add_first(3)	–	[3, 5]
D.add_first(7)	–	[7, 3, 5]
D.first( )	7	[7, 3, 5]
D.delete_last( )	5	[7, 3]
len(D)	2	[7, 3]
D.delete_last( )	3	[7]
D.delete_last( )	7	[ ]
D.add_first(6)	–	[6]
D.last( )	6	[6]
D.add_first(8)	–	[8, 6]
D.is_empty( )	False	[8, 6]
D.last( )	6	[8, 6]

6.3.2 Implementing a Deque with a Circular Array

We can implement the deque ADT in much the same way as the ArrayQueue class provided in Code Fragments 6.6 and 6.7 of Section 6.2.2 (so much so that we leave the details of an ArrayDeque implementation to Exercise P-6.32). We recommend maintaining the same three instance variables: _data, _size, and _front. Whenever we need to know the index of the back of the deque, or the first available slot beyond the back of the deque, we use modular arithmetic for the computation. For example, our implementation of the last( ) method uses the index

```
back = (self._front + self._size - 1) % len(self._data)
```

Our implementation of the ArrayDeque.add_last method is essentially the same as that for ArrayQueue.enqueue, including the reliance on a _resize utility. Likewise, the implementation of the ArrayDeque.delete_first method is the same as ArrayQueue.dequeue. Implementations of add_first and delete_last use similar techniques. One subtlety is that a call to add_first may need to wrap around the beginning of the array, so we rely on modular arithmetic to circularly *decrement* the index, as

```
self._front = (self._front - 1) % len(self._data)          # cyclic shift
```

The efficiency of an ArrayDeque is similar to that of an ArrayQueue, with all operations having $O(1)$ running time, but with that bound being amortized for operations that may change the size of the underlying list.

```
1  class EulerTour:
2    """Abstract base class for performing Euler tour of a tree.
3
4    _hook_previsit and _hook_postvisit may be overridden by subclasses.
5    """
6    def __init__(self, tree):
7      """Prepare an Euler tour template for given tree."""
8      self._tree = tree
9
10   def tree(self):
11     """Return reference to the tree being traversed."""
12     return self._tree
13
14   def execute(self):
15     """Perform the tour and return any result from post visit of root."""
16     if len(self._tree) > 0:
17       return self._tour(self._tree.root( ), 0, [ ])          # start the recursion
18
19   def _tour(self, p, d, path):
20     """Perform tour of subtree rooted at Position p.
21
22     p          Position of current node being visited
23     d          depth of p in the tree
24     path       list of indices of children on path from root to p
25     """
26     self._hook_previsit(p, d, path)                          # "pre visit" p
27     results = [ ]
28     path.append(0)             # add new index to end of path before recursion
29     for c in self._tree.children(p):
30       results.append(self._tour(c, d+1, path))      # recur on child's subtree
31       path[-1] += 1             # increment index
32     path.pop( )                 # remove extraneous index from end of path
33     answer = self._hook_postvisit(p, d, path, results)       # "post visit" p
34     return answer
35
36   def _hook_previsit(self, p, d, path):                      # can be overridden
37     pass
38
39   def _hook_postvisit(self, p, d, path, results):            # can be overridden
40     pass
```

Code Fragment 8.28: An EulerTour base class providing a framework for performing Euler tour traversals of a tree.

```
class HeapPriorityQueue(PriorityQueueBase): # base class defines _Item
  """A min-oriented priority queue implemented with a binary heap."""
  #------------------------------ nonpublic behaviors ------------------------------
  def _parent(self, j):
    return (j-1) // 2

  def _left(self, j):
    return 2*j + 1

  def _right(self, j):
    return 2*j + 2

  def _has_left(self, j):
    return self._left(j) < len(self._data)      # index beyond end of list?

  def _has_right(self, j):
    return self._right(j) < len(self._data)     # index beyond end of list?

  def _swap(self, i, j):
    """Swap the elements at indices i and j of array."""
    self._data[i], self._data[j] = self._data[j], self._data[i]

  def _upheap(self, j):
    parent = self._parent(j)
    if j > 0 and self._data[j] < self._data[parent]:
      self._swap(j, parent)
      self._upheap(parent)                      # recur at position of parent

  def _downheap(self, j):
    if self._has_left(j):
      left = self._left(j)
      small_child = left                        # although right may be smaller
      if self._has_right(j):
        right = self._right(j)
        if self._data[right] < self._data[left]:
          small_child = right
      if self._data[small_child] < self._data[j]:
        self._swap(j, small_child)
        self._downheap(small_child)             # recur at position of small child
```

Code Fragment 9.4: An implementation of a priority queue using an array-based heap (continued in Code Fragment 9.5). The extends the PriorityQueueBase class from Code Fragment 9.1.

```
25    def add(self, key, value):
26      """Add a key-value pair."""
27      token = self.Locator(key, value, len(self._data)) # initiaize locator index
28      self._data.append(token)
29      self._upheap(len(self._data) - 1)
30      return token
31
32    def update(self, loc, newkey, newval):
33      """Update the key and value for the entry identified by Locator loc."""
34      j = loc._index
35      if not (0 <= j < len(self) and self._data[j] is loc):
36        raise ValueError('Invalid locator')
37      loc._key = newkey
38      loc._value = newval
39      self._bubble(j)
40
41    def remove(self, loc):
42      """Remove and return the (k,v) pair identified by Locator loc."""
43      j = loc._index
44      if not (0 <= j < len(self) and self._data[j] is loc):
45        raise ValueError('Invalid locator')
46      if j == len(self) - 1:                  # item at last position
47        self._data.pop( )                     # just remove it
48      else:
49        self._swap(j, len(self)-1)            # swap item to the last position
50        self._data.pop( )                     # remove it from the list
51        self._bubble(j)                       # fix item displaced by the swap
52      return (loc._key, loc._value)
```

Code Fragment 9.9: An implementation of an adaptable priority queue (continued from Code Fragment 9.8).

Operation	Running Time
len(P), P.is_empty( ), P.min( )	$O(1)$
P.add(k,v)	$O(\log n)^*$
P.update(loc, k, v)	$O(\log n)$
P.remove(loc)	$O(\log n)^*$
P.remove_min( )	$O(\log n)^*$

*amortized with dynamic array

Table 9.4: Running times of the methods of an adaptable priority queue, P, of size n, realized by means of our array-based heap representation. The space requirement is $O(n)$.

P-13.48 Perform an experimental analysis of the efficiency (number of character comparisons performed) of the brute-force and KMP pattern-matching algorithms for varying-length patterns.

P-13.49 Perform an experimental analysis of the efficiency (number of character comparisons performed) of the brute-force and Boyer-Moore pattern-matching algorithms for varying-length patterns.

P-13.50 Perform an experimental comparison of the relative speeds of the brute-force, KMP, and Boyer-Moore pattern-matching algorithms. Document the relative running times on large text documents that are then searched using varying-length patterns.

P-13.51 Experiment with the efficiency of the `find` method of Python's `str` class and develop a hypothesis about which pattern-matching algorithm it uses. Try using inputs that are likely to cause both best-case and worst-case running times for various algorithms. Describe your experiments and your conclusions.

P-13.52 Implement a compression and decompression scheme that is based on Huffman coding.

P-13.53 Create a class that implements a standard trie for a set of ASCII strings. The class should have a constructor that takes a list of strings as an argument, and the class should have a method that tests whether a given string is stored in the trie.

P-13.54 Create a class that implements a compressed trie for a set of ASCII strings. The class should have a constructor that takes a list of strings as an argument, and the class should have a method that tests whether a given string is stored in the trie.

P-13.55 Create a class that implements a prefix trie for an ASCII string. The class should have a constructor that takes a string as an argument, and a method for pattern matching on the string.

P-13.56 Implement the simplified search engine described in Section 13.5.4 for the pages of a small Web site. Use all the words in the pages of the site as index terms, excluding stop words such as articles, prepositions, and pronouns.

P-13.57 Implement a search engine for the pages of a small Web site by adding a page-ranking feature to the simplified search engine described in Section 13.5.4. Your page-ranking feature should return the most relevant pages first. Use all the words in the pages of the site as index terms, excluding stop words, such as articles, prepositions, and pronouns.

Sending the wrong number, type, or value of parameters to a function is another common cause for an exception. For example, a call to abs('hello') will raise a TypeError because the parameter is not numeric, and a call to abs(3, 5) will raise a TypeError because one parameter is expected. A ValueError is typically raised when the correct number and type of parameters are sent, but a value is illegitimate for the context of the function. For example, the int constructor accepts a string, as with int('137'), but a ValueError is raised if that string does not represent an integer, as with int('3.14') or int('hello').

Python's sequence types (e.g., list, tuple, and str) raise an IndexError when syntax such as data[k] is used with an integer k that is not a valid index for the given sequence (as described in Section 1.2.3). Sets and dictionaries raise a KeyError when an attempt is made to access a nonexistent element.

1.7.1 Raising an Exception

An exception is thrown by executing the **raise** statement, with an appropriate instance of an exception class as an argument that designates the problem. For example, if a function for computing a square root is sent a negative value as a parameter, it can raise an exception with the command:

```
raise ValueError('x cannot be negative')
```

This syntax raises a newly created instance of the ValueError class, with the error message serving as a parameter to the constructor. If this exception is not caught within the body of the function, the execution of the function immediately ceases and the exception is propagated to the calling context (and possibly beyond).

When checking the validity of parameters sent to a function, it is customary to first verify that a parameter is of an appropriate type, and then to verify that it has an appropriate value. For example, the sqrt function in Python's math library performs error-checking that might be implemented as follows:

```
def sqrt(x):
  if not isinstance(x, (int, float)):
    raise TypeError('x must be numeric')
  elif x < 0:
    raise ValueError('x cannot be negative')
  # do the real work here...
```

Checking the type of an object can be performed at run-time using the built-in function, isinstance. In simplest form, isinstance(obj, cls) returns True if object, obj, is an instance of class, cls, or any subclass of that type. In the above example, a more general form is used with a tuple of allowable types indicated with the second parameter. After confirming that the parameter is numeric, the function enforces an expectation that the number be nonnegative, raising a ValueError otherwise.

```
1  from abc import ABCMeta, abstractmethod          # need these definitions
2
3  class Sequence(metaclass=ABCMeta):
4    """Our own version of collections.Sequence abstract base class."""
5
6    @abstractmethod
7    def __len__(self):
8      """Return the length of the sequence."""
9
10   @abstractmethod
11   def __getitem__(self, j):
12     """Return the element at index j of the sequence."""
13
14   def __contains__(self, val):
15     """Return True if val found in the sequence; False otherwise."""
16     for j in range(len(self)):
17       if self[j] == val:                          # found match
18         return True
19     return False
20
21   def index(self, val):
22     """Return leftmost index at which val is found (or raise ValueError)."""
23     for j in range(len(self)):
24       if self[j] == val:                          # leftmost match
25         return j
26     raise ValueError('value not in sequence')    # never found a match
27
28   def count(self, val):
29     """Return the number of elements equal to given value."""
30     k = 0
31     for j in range(len(self)):
32       if self[j] == val:                          # found a match
33         k += 1
34     return k
```

Code Fragment 2.14: An abstract base class akin to collections.Sequence.

This implementation relies on two advanced Python techniques. The first is that we declare the ABCMeta class of the abc module as a ***metaclass*** of our Sequence class. A metaclass is different from a superclass, in that it provides a template for the class definition itself. Specifically, the ABCMeta declaration assures that the constructor for the class raises an error.

3.1.1 Moving Beyond Experimental Analysis

Our goal is to develop an approach to analyzing the efficiency of algorithms that:

1. Allows us to evaluate the relative efficiency of any two algorithms in a way that is independent of the hardware and software environment.
2. Is performed by studying a high-level description of the algorithm without need for implementation.
3. Takes into account all possible inputs.

Counting Primitive Operations

To analyze the running time of an algorithm without performing experiments, we perform an analysis directly on a high-level description of the algorithm (either in the form of an actual code fragment, or language-independent pseudo-code). We define a set of ***primitive operations*** such as the following:

- Assigning an identifier to an object
- Determining the object associated with an identifier
- Performing an arithmetic operation (for example, adding two numbers)
- Comparing two numbers
- Accessing a single element of a Python list by index
- Calling a function (excluding operations executed within the function)
- Returning from a function.

Formally, a primitive operation corresponds to a low-level instruction with an execution time that is constant. Ideally, this might be the type of basic operation that is executed by the hardware, although many of our primitive operations may be translated to a small number of instructions. Instead of trying to determine the specific execution time of each primitive operation, we will simply count how many primitive operations are executed, and use this number t as a measure of the running time of the algorithm.

This operation count will correlate to an actual running time in a specific computer, for each primitive operation corresponds to a constant number of instructions, and there are only a fixed number of primitive operations. The implicit assumption in this approach is that the running times of different primitive operations will be fairly similar. Thus, the number, t, of primitive operations an algorithm performs will be proportional to the actual running time of that algorithm.

Measuring Operations as a Function of Input Size

To capture the order of growth of an algorithm's running time, we will associate, with each algorithm, a function $f(n)$ that characterizes the number of primitive operations that are performed as a function of the input size n. Section 3.2 will introduce the seven most common functions that arise, and Section 3.3 will introduce a mathematical framework for comparing functions to each other.

5.2 Low-Level Arrays

To accurately describe the way in which Python represents the sequence types, we must first discuss aspects of the low-level computer architecture. The primary memory of a computer is composed of bits of information, and those bits are typically grouped into larger units that depend upon the precise system architecture. Such a typical unit is a ***byte***, which is equivalent to 8 bits.

A computer system will have a huge number of bytes of memory, and to keep track of what information is stored in what byte, the computer uses an abstraction known as a ***memory address***. In effect, each byte of memory is associated with a unique number that serves as its address (more formally, the binary representation of the number serves as the address). In this way, the computer system can refer to the data in "byte #2150" versus the data in "byte #2157," for example. Memory addresses are typically coordinated with the physical layout of the memory system, and so we often portray the numbers in sequential fashion. Figure 5.1 provides such a diagram, with the designated memory address for each byte.

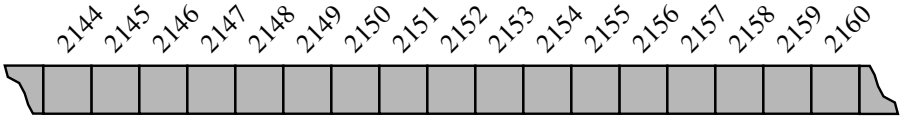


Figure 5.1: A representation of a portion of a computer's memory, with individual bytes labeled with consecutive memory addresses.

Despite the sequential nature of the numbering system, computer hardware is designed, in theory, so that any byte of the main memory can be efficiently accessed based upon its memory address. In this sense, we say that a computer's main memory performs as ***random access memory (RAM)***. That is, it is just as easy to retrieve byte #8675309 as it is to retrieve byte #309. (In practice, there are complicating factors including the use of caches and external memory; we address some of those issues in Chapter 15.) Using the notation for asymptotic analysis, we say that any individual byte of memory can be stored or retrieved in $O(1)$ time.

In general, a programming language keeps track of the association between an identifier and the memory address in which the associated value is stored. For example, identifier `x` might be associated with one value stored in memory, while `y` is associated with another value stored in memory. A common programming task is to keep track of a sequence of related objects. For example, we may want a video game to keep track of the top ten scores for that game. Rather than use ten different variables for this task, we would prefer to use a single name for the group and use index numbers to refer to the high scores in that group.

5.4.2 Python's String Class

Strings are very important in Python. We introduced their use in Chapter 1, with a discussion of various operator syntaxes in Section 1.3. A comprehensive summary of the named methods of the class is given in Tables A.1 through A.4 of Appendix A. We will not formally analyze the efficiency of each of those behaviors in this section, but we do wish to comment on some notable issues. In general, we let n denote the length of a string. For operations that rely on a second string as a pattern, we let m denote the length of that pattern string.

The analysis for many behaviors is quite intuitive. For example, methods that produce a new string (e.g., capitalize, center, strip) require time that is linear in the length of the string that is produced. Many of the behaviors that test Boolean conditions of a string (e.g., islower) take $O(n)$ time, examining all n characters in the worst case, but short circuiting as soon as the answer becomes evident (e.g., islower can immediately return False if the first character is uppercased). The comparison operators (e.g., $==$, $<$) fall into this category as well.

Pattern Matching

Some of the most interesting behaviors, from an algorithmic point of view, are those that in some way depend upon finding a string pattern within a larger string; this goal is at the heart of methods such as __contains__, find, index, count, replace, and split. String algorithms will be the topic of Chapter 13, and this particular problem known as ***pattern matching*** will be the focus of Section 13.2. A naive implementation runs in $O(mn)$ time case, because we consider the $n-m+1$ possible starting indices for the pattern, and we spend $O(m)$ time at each starting position, checking if the pattern matches. However, in Section 13.2, we will develop an algorithm for finding a pattern of length m within a longer string of length n in $O(n)$ time.

Composing Strings

Finally, we wish to comment on several approaches for composing large strings. As an academic exercise, assume that we have a large string named document, and our goal is to produce a new string, letters, that contains only the alphabetic characters of the original string (e.g., with spaces, numbers, and punctuation removed). It may be tempting to compose a result through repeated concatenation, as follows.

```
# WARNING: do not do this
letters = ''                          # start with empty string
for c in document:
  if c.isalpha( ):
    letters += c                      # concatenate alphabetic character
```

```
class CaesarCipher:
  """Class for doing encryption and decryption using a Caesar cipher."""

  def __init__(self, shift):
    """Construct Caesar cipher using given integer shift for rotation."""
    encoder = [None] * 26                    # temp array for encryption
    decoder = [None] * 26                    # temp array for decryption
    for k in range(26):
      encoder[k] = chr((k + shift) % 26 + ord('A'))
      decoder[k] = chr((k - shift) % 26 + ord('A'))
    self._forward = ''.join(encoder)         # will store as string
    self._backward = ''.join(decoder)        # since fixed

  def encrypt(self, message):
    """Return string representing encripted message."""
    return  self._transform(message, self._forward)

  def decrypt(self, secret):
    """Return decrypted message given encrypted secret."""
    return  self._transform(secret, self._backward)

  def _transform(self, original, code):
    """Utility to perform transformation based on given code string."""
    msg = list(original)
    for k in range(len(msg)):
      if msg[k].isupper( ):
        j = ord(msg[k]) - ord('A')           # index from 0 to 25
        msg[k] = code[j]                     # replace this character
    return ''.join(msg)

if __name__ == '__main__':
  cipher = CaesarCipher(3)
  message = "THE EAGLE IS IN PLAY; MEET AT JOE'S."
  coded = cipher.encrypt(message)
  print('Secret: ', coded)
  answer = cipher.decrypt(coded)
  print('Message:', answer)
```

Code Fragment 5.11: A complete Python class for the Caesar cipher.

10.2.2 Collision-Handling Schemes

The main idea of a hash table is to take a bucket array, A, and a hash function, h, and use them to implement a map by storing each item (k,v) in the "bucket" $A[h(k)]$. This simple idea is challenged, however, when we have two distinct keys, k_1 and k_2, such that $h(k_1) = h(k_2)$. The existence of such ***collisions*** prevents us from simply inserting a new item (k,v) directly into the bucket $A[h(k)]$. It also complicates our procedure for performing insertion, search, and deletion operations.

Separate Chaining

A simple and efficient way for dealing with collisions is to have each bucket $A[j]$ store its own secondary container, holding items (k,v) such that $h(k) = j$. A natural choice for the secondary container is a small map instance implemented using a list, as described in Section 10.1.5. This ***collision resolution*** rule is known as ***separate chaining***, and is illustrated in Figure 10.6.

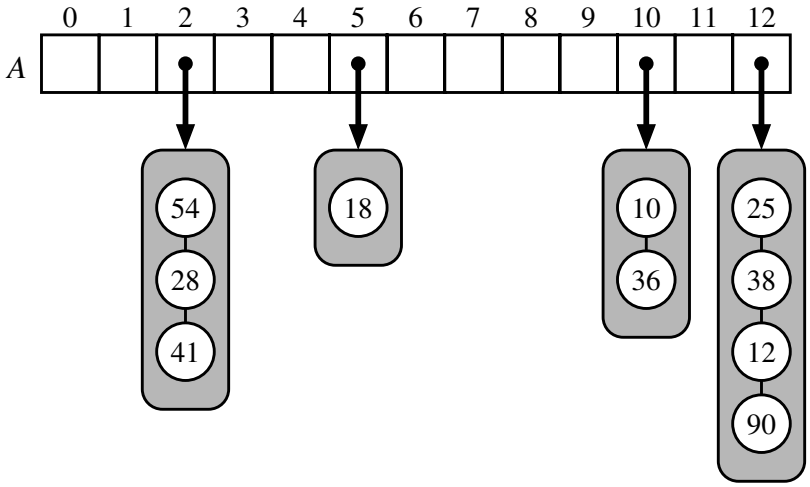


Figure 10.6: A hash table of size 13, storing 10 items with integer keys, with collisions resolved by separate chaining. The compression function is $h(k) = k \bmod 13$. For simplicity, we do not show the values associated with the keys.

In the worst case, operations on an individual bucket take time proportional to the size of the bucket. Assuming we use a good hash function to index the n items of our map in a bucket array of capacity N, the expected size of a bucket is n/N. Therefore, if given a good hash function, the core map operations run in $O(\lceil n/N \rceil)$. The ratio $\lambda = n/N$, called the ***load factor*** of the hash table, should be bounded by a small constant, preferably below 1. As long as λ is $O(1)$, the core operations on the hash table run in $O(1)$ expected time.