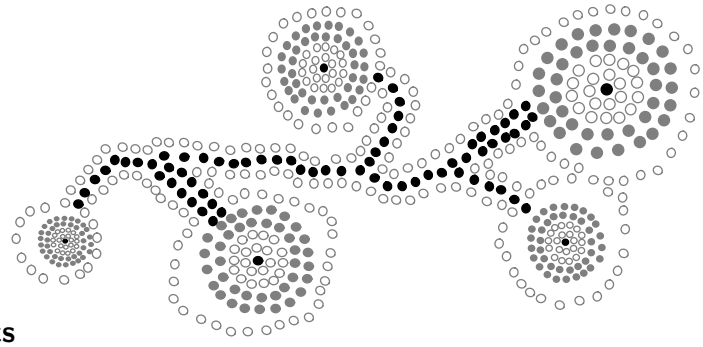
Chapter

9 Priority Queues



Contents

9.1 The Priority Queue Abstract Data Type

9.1.1 Priorities

In Chapter 6, we introduced the queue ADT as a collection of objects that are added and removed according to the ***first-in, first-out*** (***FIFO***) principle. A company's customer call center embodies such a model in which waiting customers are told "calls will be answered in the order that they were received." In that setting, a new call is added to the back of the queue, and each time a customer service representative becomes available, he or she is connected with the call that is removed from the front of the call queue.

In practice, there are many applications in which a queue-like structure is used to manage objects that must be processed in some way, but for which the first-in, first-out policy does not suffice. Consider, for example, an air-traffic control center that has to decide which flight to clear for landing from among many approaching the airport. This choice may be influenced by factors such as each plane's distance from the runway, time spent waiting in a holding pattern, or amount of remaining fuel. It is unlikely that the landing decisions are based purely on a FIFO policy.

There are other situations in which a "first come, first serve" policy might seem reasonable, yet for which other priorities come into play. To use another airline analogy, suppose a certain flight is fully booked an hour prior to departure. Because of the possibility of cancellations, the airline maintains a queue of standby passengers hoping to get a seat. Although the priority of a standby passenger is influenced by the check-in time of that passenger, other considerations include the fare paid and frequent-flyer status. So it may be that an available seat is given to a passenger who has arrived *later* than another, if such a passenger is assigned a better priority by the airline agent.

In this chapter, we introduce a new abstract data type known as a ***priority queue***. This is a collection of prioritized elements that allows arbitrary element insertion, and allows the removal of the element that has first priority. When an element is added to a priority queue, the user designates its priority by providing an associated ***key***. The element with the *minimum* key will be the next to be removed from the queue (thus, an element with key 1 will be given priority over an element with key 2). Although it is quite common for priorities to be expressed numerically, any Python object may be used as a key, as long as the object type supports a consistent meaning for the test `a < b`, for any instances `a` and `b`, so as to define a natural order of the keys. With such generality, applications may develop their own notion of priority for each element. For example, different financial analysts may assign different ratings (i.e., priorities) to a particular asset, such as a share of stock.

9.1.2 The Priority Queue ADT

Formally, we model an element and its priority as a key-value pair. We define the priority queue ADT to support the following methods for a priority queue P:

P.add(k, v): Insert an item with key k and value v into priority queue P.

P.min( ): Return a tuple, (k,v), representing the key and value of an item in priority queue P with minimum key (but do not remove the item); an error occurs if the priority queue is empty.

P.remove_min( ): Remove an item with minimum key from priority queue P, and return a tuple, (k,v), representing the key and value of the removed item; an error occurs if the priority queue is empty.

P.is_empty( ): Return True if priority queue P does not contain any items.

len(P): Return the number of items in priority queue P.

A priority queue may have multiple entries with equivalent keys, in which case methods min and remove_min may report an arbitrary choice of item having minimum key. Values may be any type of object.

In our initial model for a priority queue, we assume that an element's key remains fixed once it has been added to a priority queue. In Section 9.5, we consider an extension that allows a user to update an element's key within the priority queue.

Example 9.1: *The following table shows a series of operations and their effects on an initially empty priority queue P. The "Priority Queue" column is somewhat deceiving since it shows the entries as tuples and sorted by key. Such an internal representation is not required of a priority queue.*

Operation	Return Value	Priority Queue
P.add(5,A)		{(5,A)}
P.add(9,C)		{(5,A), (9,C)}
P.add(3,B)		{(3,B), (5,A), (9,C)}
P.add(7,D)		{(3,B), (5,A), (7,D), (9,C)}
P.min( )	(3,B)	{(3,B), (5,A), (7,D), (9,C)}
P.remove_min( )	(3,B)	{(5,A), (7,D), (9,C)}
P.remove_min( )	(5,A)	{(7,D), (9,C)}
len(P)	2	{(7,D), (9,C)}
P.remove_min( )	(7,D)	{(9,C)}
P.remove_min( )	(9,C)	{ }
P.is_empty( )	True	{ }
P.remove_min( )	"error"	{ }

9.2 Implementing a Priority Queue

In this section, we show how to implement a priority queue by storing its entries in a positional list L. (See Section 7.4.) We provide two realizations, depending on whether or not we keep the entries in L sorted by key.

9.2.1 The Composition Design Pattern

One challenge in implementing a priority queue is that we must keep track of both an element and its key, even as items are relocated within our data structure. This is reminiscent of a case study from Section 7.6 in which we maintain access counts with each element. In that setting, we introduced the ***composition design pattern***, defining an _Item class that assured that each element remained paired with its associated count in our primary data structure.

For priority queues, we will use composition to store items internally as pairs consisting of a key k and a value v. To implement this concept for all priority queue implementations, we provide a PriorityQueueBase class (see Code Fragment 9.1) that includes a definition for a nested class named _Item. We define the syntax a < b, for item instances a and b, to be based upon the keys.

```
1  class PriorityQueueBase:
2    """Abstract base class for a priority queue."""
3
4    class _Item:
5      """Lightweight composite to store priority queue items."""
6      __slots__ = '_key', '_value'
7
8      def __init__(self, k, v):
9        self._key = k
10       self._value = v
11
12     def __lt__(self, other):
13       return self._key < other._key       # compare items based on their keys
14
15   def is_empty(self):                     # concrete method assuming abstract len
16     """Return True if the priority queue is empty."""
17     return len(self) == 0
```

Code Fragment 9.1: A PriorityQueueBase class with a nested _Item class that composes a key and a value into a single object. For convenience, we provide a concrete implementation of is_empty that is based on a presumed __len__ impelementation.

9.2.2 Implementation with an Unsorted List

In our first concrete implementation of a priority queue, we store entries within an *unsorted list*. Our UnsortedPriorityQueue class is given in Code Fragment 9.2, inheriting from the PriorityQueueBase class introduced in Code Fragment 9.1. For internal storage, key-value pairs are represented as composites, using instances of the inherited _Item class. These items are stored within a PositionalList, identified as the _data member of our class. We assume that the positional list is implemented with a doubly-linked list, as in Section 7.4, so that all operations of that ADT execute in $O(1)$ time.

We begin with an empty list when a new priority queue is constructed. At all times, the size of the list equals the number of key-value pairs currently stored in the priority queue. For this reason, our priority queue __len__ method simply returns the length of the internal _data list. By the design of our PriorityQueueBase class, we inherit a concrete implementation of the is_empty method that relies on a call to our __len__ method.

Each time a key-value pair is added to the priority queue, via the add method, we create a new _Item composite for the given key and value, and add that item to the end of the list. Such an implementation takes $O(1)$ time.

The remaining challenge is that when min or remove_min is called, we must locate the item with minimum key. Because the items are not sorted, we must inspect all entries to find one with a minimum key. For convenience, we define a nonpublic _find_min utility that returns the *position* of an item with minimum key. Knowledge of the position allows the remove_min method to invoke the delete method on the positional list. The min method simply uses the position to retrieve the item when preparing a key-value tuple to return. Due to the loop for finding the minimum key, both min and remove_min methods run in $O(n)$ time, where n is the number of entries in the priority queue.

A summary of the running times for the UnsortedPriorityQueue class is given in Table 9.1.

Operation	Running Time
len	$O(1)$
is_empty	$O(1)$
add	$O(1)$
min	$O(n)$
remove_min	$O(n)$

Table 9.1: Worst-case running times of the methods of a priority queue of size n, realized by means of an unsorted, doubly linked list. The space requirement is $O(n)$.

```
1  class UnsortedPriorityQueue(PriorityQueueBase): # base class defines _Item
2    """A min-oriented priority queue implemented with an unsorted list."""
3
4    def _find_min(self):                    # nonpublic utility
5      """Return Position of item with minimum key."""
6      if self.is_empty( ):                  # is_empty inherited from base class
7        raise Empty('Priority queue is empty')
8      small = self._data.first( )
9      walk = self._data.after(small)
10     while walk is not None:
11       if walk.element( ) < small.element( ):
12         small = walk
13       walk = self._data.after(walk)
14     return small
15
16   def __init__(self):
17     """Create a new empty Priority Queue."""
18     self._data = PositionalList( )
19
20   def __len__(self):
21     """Return the number of items in the priority queue."""
22     return len(self._data)
23
24   def add(self, key, value):
25     """Add a key-value pair."""
26     self._data.add_last(self._Item(key, value))
27
28   def min(self):
29     """Return but do not remove (k,v) tuple with minimum key."""
30     p = self._find_min( )
31     item = p.element( )
32     return (item._key, item._value)
33
34   def remove_min(self):
35     """Remove and return (k,v) tuple with minimum key."""
36     p = self._find_min( )
37     item = self._data.delete(p)
38     return (item._key, item._value)
```

Code Fragment 9.2: An implementation of a priority queue using an unsorted list. The parent class PriorityQueueBase is given in Code Fragment 9.1, and the PositionalList class is from Section 7.4.

9.2.3 Implementation with a Sorted List

An alternative implementation of a priority queue uses a positional list, yet maintaining entries sorted by nondecreasing keys. This ensures that the first element of the list is an entry with the smallest key.

Our SortedPriorityQueue class is given in Code Fragment 9.3. The implementation of min and remove_min are rather straightforward given knowledge that the first element of a list has a minimum key. We rely on the first method of the positional list to find the position of the first item, and the delete method to remove the entry from the list. Assuming that the list is implemented with a doubly linked list, operations min and remove_min take $O(1)$ time.

This benefit comes at a cost, however, for method add now requires that we scan the list to find the appropriate position to insert the new item. Our implementation starts at the end of the list, walking backward until the new key is smaller than an existing item; in the worst case, it progresses until reaching the front of the list. Therefore, the add method takes $O(n)$ worst-case time, where n is the number of entries in the priority queue at the time the method is executed. In summary, when using a sorted list to implement a priority queue, insertion runs in linear time, whereas finding and removing the minimum can be done in constant time.

Comparing the Two List-Based Implementations

Table 9.2 compares the running times of the methods of a priority queue realized by means of a sorted and unsorted list, respectively. We see an interesting trade-off when we use a list to implement the priority queue ADT. An unsorted list supports fast insertions but slow queries and deletions, whereas a sorted list allows fast queries and deletions, but slow insertions.

Operation	Unsorted List	Sorted List
len	$O(1)$	$O(1)$
is_empty	$O(1)$	$O(1)$
add	$O(1)$	$O(n)$
min	$O(n)$	$O(1)$
remove_min	$O(n)$	$O(1)$

Table 9.2: Worst-case running times of the methods of a priority queue of size n, realized by means of an unsorted or sorted list, respectively. We assume that the list is implemented by a doubly linked list. The space requirement is $O(n)$.

```
1  class SortedPriorityQueue(PriorityQueueBase): # base class defines _Item
2    """A min-oriented priority queue implemented with a sorted list."""
3
4    def __init__(self):
5      """Create a new empty Priority Queue."""
6      self._data = PositionalList( )
7
8    def __len__(self):
9      """Return the number of items in the priority queue."""
10     return len(self._data)
11
12   def add(self, key, value):
13     """Add a key-value pair."""
14     newest = self._Item(key, value)              # make new item instance
15     walk = self._data.last( )          # walk backward looking for smaller key
16     while walk is not None and newest < walk.element( ):
17       walk = self._data.before(walk)
18     if walk is None:
19       self._data.add_first(newest)               # new key is smallest
20     else:
21       self._data.add_after(walk, newest)         # newest goes after walk
22
23   def min(self):
24     """Return but do not remove (k,v) tuple with minimum key."""
25     if self.is_empty( ):
26       raise Empty('Priority queue is empty.')
27     p = self._data.first( )
28     item = p.element( )
29     return (item._key, item._value)
30
31   def remove_min(self):
32     """Remove and return (k,v) tuple with minimum key."""
33     if self.is_empty( ):
34       raise Empty('Priority queue is empty.')
35     item = self._data.delete(self._data.first( ))
36     return (item._key, item._value)
```

Code Fragment 9.3: An implementation of a priority queue using a sorted list. The parent class PriorityQueueBase is given in Code Fragment 9.1, and the PositionalList class is from Section 7.4.

9.3 Heaps

The two strategies for implementing a priority queue ADT in the previous section demonstrate an interesting trade-off. When using an *unsorted* list to store entries, we can perform insertions in $O(1)$ time, but finding or removing an element with minimum key requires an $O(n)$-time loop through the entire collection. In contrast, if using a *sorted* list, we can trivially find or remove the minimum element in $O(1)$ time, but adding a new element to the queue may require $O(n)$ time to restore the sorted order.

In this section, we provide a more efficient realization of a priority queue using a data structure called a ***binary heap***. This data structure allows us to perform both insertions and removals in logarithmic time, which is a significant improvement over the list-based implementations discussed in Section 9.2. The fundamental way the heap achieves this improvement is to use the structure of a binary tree to find a compromise between elements being entirely unsorted and perfectly sorted.

9.3.1 The Heap Data Structure

A heap (see Figure 9.1) is a binary tree T that stores a collection of items at its positions and that satisfies two additional properties: a relational property defined in terms of the way keys are stored in T and a structural property defined in terms of the shape of T itself. The relational property is the following:

***Heap-Order Property*:** In a heap T, for every position p other than the root, the key stored at p is greater than or equal to the key stored at p's parent.

As a consequence of the heap-order property, the keys encountered on a path from the root to a leaf of T are in nondecreasing order. Also, a minimum key is always stored at the root of T. This makes it easy to locate such an item when min or remove_min is called, as it is informally said to be "at the top of the heap" (hence, the name "heap" for the data structure). By the way, the heap data structure defined here has nothing to do with the memory heap (Section 15.1.1) used in the run-time environment supporting a programming language like Python.

For the sake of efficiency, as will become clear later, we want the heap T to have as small a height as possible. We enforce this requirement by insisting that the heap T satisfy an additional structural property—it must be what we term ***complete***.

Complete Binary Tree Property: A heap T with height h is a ***complete*** binary tree if levels $0, 1, 2, \ldots, h-1$ of T have the maximum number of nodes possible (namely, level i has 2^i nodes, for $0 \leq i \leq h-1$) and the remaining nodes at level h reside in the leftmost possible positions at that level.

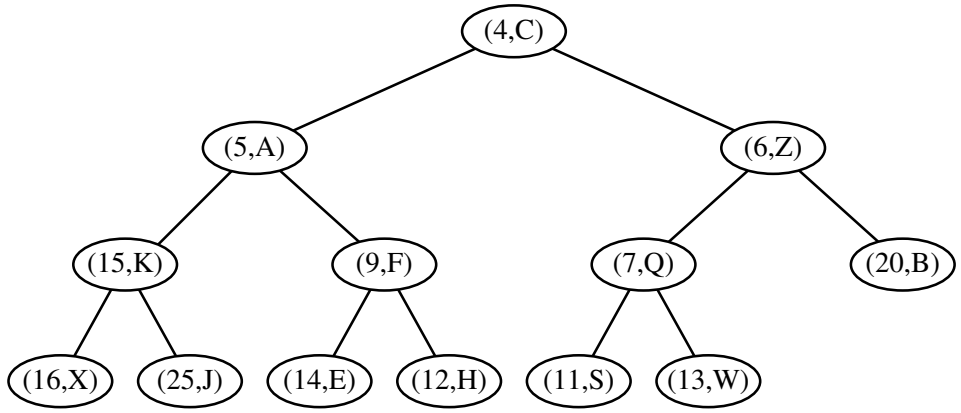


Figure 9.1: Example of a heap storing 13 entries with integer keys. The last position is the one storing entry $(13,W)$.

The tree in Figure 9.1 is complete because levels 0, 1, and 2 are full, and the six nodes in level 3 are in the six leftmost possible positions at that level. In formalizing what we mean by the leftmost possible positions, we refer to the discussion of ***level numbering*** from Section 8.3.2, in the context of an array-based representation of a binary tree. (In fact, in Section 9.3.3 we will discuss the use of an array to represent a heap.) A complete binary tree with n elements is one that has positions with level numbering 0 through $n-1$. For example, in an array-based representation of the above tree, its 13 entries would be stored consecutively from $A[0]$ to $A[12]$.

The Height of a Heap

Let h denote the height of T. Insisting that T be complete also has an important consequence, as shown in Proposition 9.2.

Proposition 9.2: *A heap T storing n entries has height $h = \lfloor \log n \rfloor$.*

Justification: From the fact that T is complete, we know that the number of nodes in levels 0 through $h-1$ of T is precisely $1+2+4+\cdots+2^{h-1} = 2^h - 1$, and that the number of nodes in level h is at least 1 and at most 2^h. Therefore

$$n \geq 2^h - 1 + 1 = 2^h \qquad \text{and} \qquad n \leq 2^h - 1 + 2^h = 2^{h+1} - 1.$$

By taking the logarithm of both sides of inequality $2^h \leq n$, we see that height $h \leq \log n$. By rearranging terms and taking the logarithm of both sides of inequality $n \leq 2^{h+1} - 1$, we see that $\log(n+1) - 1 \leq h$. Since h is an integer, these two inequalities imply that $h = \lfloor \log n \rfloor$. ■

9.3.2 Implementing a Priority Queue with a Heap

Proposition 9.2 has an important consequence, for it implies that if we can perform update operations on a heap in time proportional to its height, then those operations will run in logarithmic time. Let us therefore turn to the problem of how to efficiently perform various priority queue methods using a heap.

We will use the composition pattern from Section 9.2.1 to store key-value pairs as items in the heap. The len and is_empty methods can be implemented based on examination of the tree, and the min operation is equally trivial because the heap property assures that the element at the root of the tree has a minimum key. The interesting algorithms are those for implementing the add and remove_min methods.

Adding an Item to the Heap

Let us consider how to perform add(k,v) on a priority queue implemented with a heap T. We store the pair (k, v) as an item at a new node of the tree. To maintain the ***complete binary tree property***, that new node should be placed at a position p just beyond the rightmost node at the bottom level of the tree, or as the leftmost position of a new level, if the bottom level is already full (or if the heap is empty).

Up-Heap Bubbling After an Insertion

After this action, the tree T is complete, but it may violate the ***heap-order property***. Hence, unless position p is the root of T (that is, the priority queue was empty before the insertion), we compare the key at position p to that of p's parent, which we denote as q. If key $k_p \geq k_q$, the heap-order property is satisfied and the algorithm terminates. If instead $k_p < k_q$, then we need to restore the heap-order property, which can be locally achieved by swapping the entries stored at positions p and q. (See Figure 9.2c and d.) This swap causes the new item to move up one level. Again, the heap-order property may be violated, so we repeat the process, going up in T until no violation of the heap-order property occurs. (See Figure 9.2e and h.)

The upward movement of the newly inserted entry by means of swaps is conventionally called ***up-heap bubbling***. A swap either resolves the violation of the heap-order property or propagates it one level up in the heap. In the worst case, up-heap bubbling causes the new entry to move all the way up to the root of heap T. Thus, in the worst case, the number of swaps performed in the execution of method add is equal to the height of T. By Proposition 9.2, that bound is $\lfloor \log n \rfloor$.

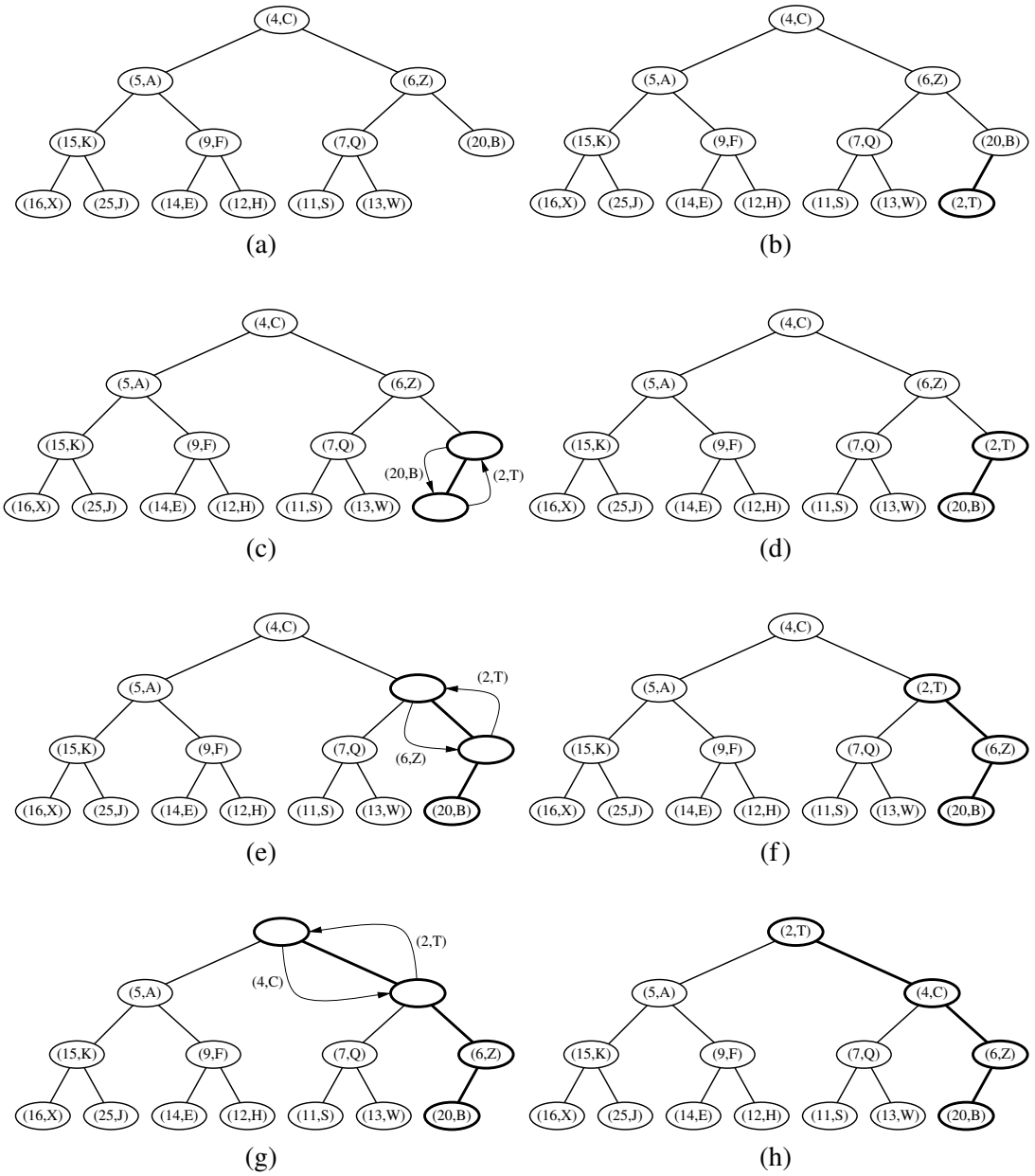


Figure 9.2: Insertion of a new entry with key 2 into the heap of Figure 9.1: (a) initial heap; (b) after performing operation add; (c and d) swap to locally restore the partial order property; (e and f) another swap; (g and h) final swap.

Removing the Item with Minimum Key

Let us now turn to method remove_min of the priority queue ADT. We know that an entry with the smallest key is stored at the root r of T (even if there is more than one entry with smallest key). However, in general we cannot simply delete node r, because this would leave two disconnected subtrees.

Instead, we ensure that the shape of the heap respects the ***complete binary tree property*** by deleting the leaf at the *last* position p of T, defined as the rightmost position at the bottommost level of the tree. To preserve the item from the last position p, we copy it to the root r (in place of the item with minimum key that is being removed by the operation). Figure 9.3a and b illustrates an example of these steps, with minimal item $(4,C)$ being removed from the root and replaced by item $(13,W)$ from the last position. The node at the last position is removed from the tree.

Down-Heap Bubbling After a Removal

We are not yet done, however, for even though T is now complete, it likely violates the heap-order property. If T has only one node (the root), then the heap-order property is trivially satisfied and the algorithm terminates. Otherwise, we distinguish two cases, where p initially denotes the root of T:

- If p has no right child, let c be the left child of p.
- Otherwise (p has both children), let c be a child of p with minimal key.

If key $k_p \leq k_c$, the heap-order property is satisfied and the algorithm terminates. If instead $k_p > k_c$, then we need to restore the heap-order property. This can be locally achieved by swapping the entries stored at p and c. (See Figure 9.3c and d.) It is worth noting that when p has two children, we intentionally consider the *smaller* key of the two children. Not only is the key of c smaller than that of p, it is at least as small as the key at c's sibling. This ensures that the heap-order property is locally restored when that smaller key is promoted above the key that had been at p and that at c's sibling.

Having restored the heap-order property for node p relative to its children, there may be a violation of this property at c; hence, we may have to continue swapping down T until no violation of the heap-order property occurs. (See Figure 9.3e–h.) This downward swapping process is called ***down-heap bubbling***. A swap either resolves the violation of the heap-order property or propagates it one level down in the heap. In the worst case, an entry moves all the way down to the bottom level. (See Figure 9.3.) Thus, the number of swaps performed in the execution of method remove_min is, in the worst case, equal to the height of heap T, that is, it is $\lfloor \log n \rfloor$ by Proposition 9.2.

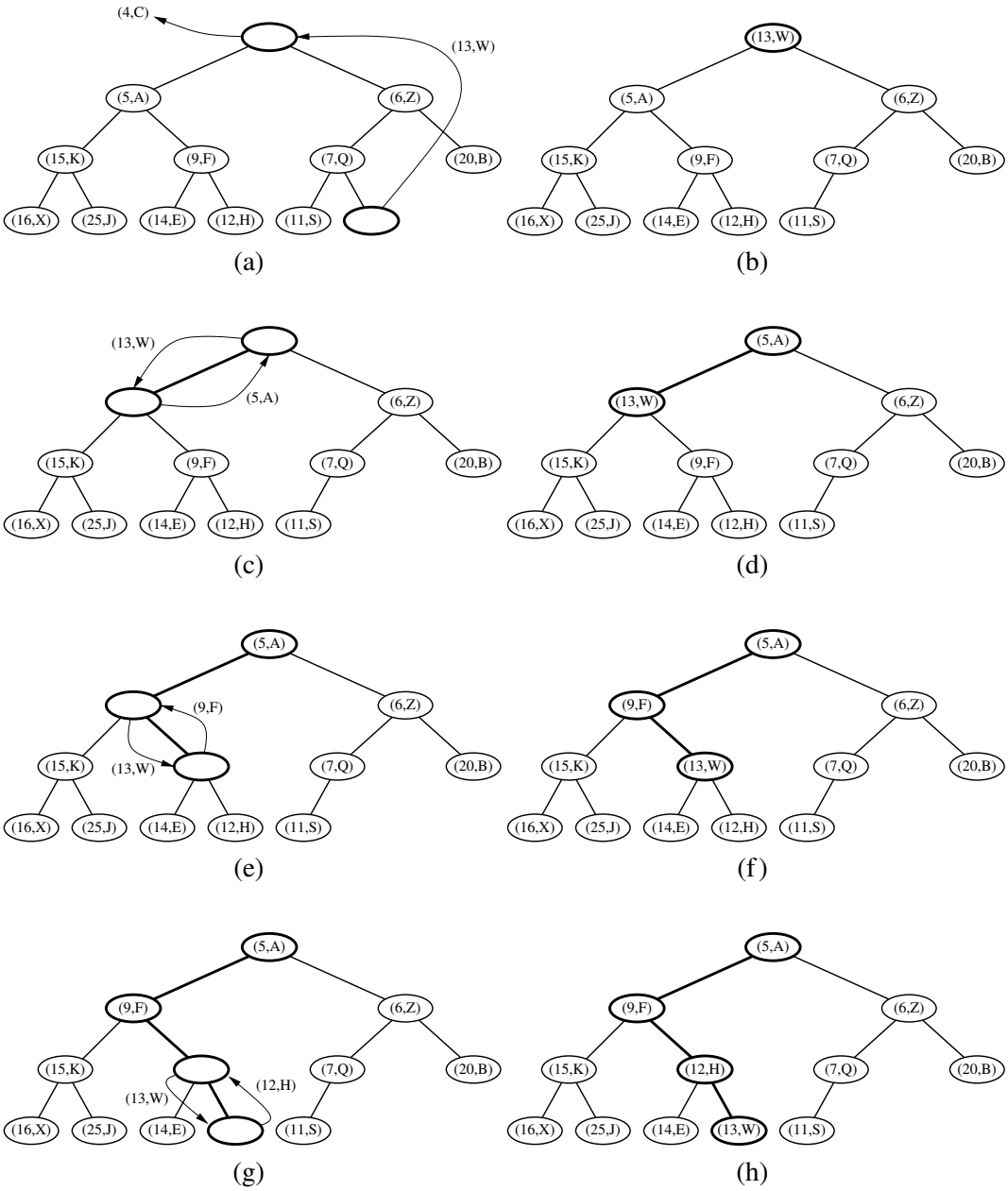


Figure 9.3: Removal of the entry with the smallest key from a heap: (a and b) deletion of the last node, whose entry gets stored into the root; (c and d) swap to locally restore the heap-order property; (e and f) another swap; (g and h) final swap.

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.

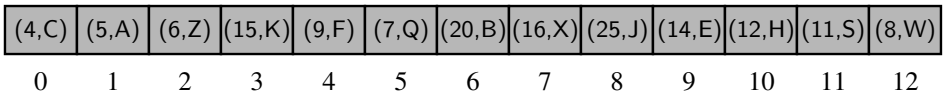


Figure 9.4: An array-based representation of the heap from Figure 9.1.

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.

```
1  class HeapPriorityQueue(PriorityQueueBase): # base class defines _Item
2    """A min-oriented priority queue implemented with a binary heap."""
3    #------------------------------ nonpublic behaviors ------------------------------
4    def _parent(self, j):
5      return (j-1) // 2
6
7    def _left(self, j):
8      return 2*j + 1
9
10   def _right(self, j):
11     return 2*j + 2
12
13   def _has_left(self, j):
14     return self._left(j) < len(self._data)      # index beyond end of list?
15
16   def _has_right(self, j):
17     return self._right(j) < len(self._data)     # index beyond end of list?
18
19   def _swap(self, i, j):
20     """Swap the elements at indices i and j of array."""
21     self._data[i], self._data[j] = self._data[j], self._data[i]
22
23   def _upheap(self, j):
24     parent = self._parent(j)
25     if j > 0 and self._data[j] < self._data[parent]:
26       self._swap(j, parent)
27       self._upheap(parent)                      # recur at position of parent
28
29   def _downheap(self, j):
30     if self._has_left(j):
31       left = self._left(j)
32       small_child = left                        # although right may be smaller
33       if self._has_right(j):
34         right = self._right(j)
35         if self._data[right] < self._data[left]:
36           small_child = right
37       if self._data[small_child] < self._data[j]:
38         self._swap(j, small_child)
39         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.

```
40    #------------------------------ public behaviors ------------------------------
41    def __init__(self):
42      """Create a new empty Priority Queue."""
43      self._data = [ ]
44
45    def __len__(self):
46      """Return the number of items in the priority queue."""
47      return len(self._data)
48
49    def add(self, key, value):
50      """Add a key-value pair to the priority queue."""
51      self._data.append(self._Item(key, value))
52      self._upheap(len(self._data) - 1)          # upheap newly added position
53
54    def min(self):
55      """Return but do not remove (k,v) tuple with minimum key.
56
57      Raise Empty exception if empty.
58      """
59      if self.is_empty( ):
60        raise Empty('Priority queue is empty.')
61      item = self._data[0]
62      return (item._key, item._value)
63
64    def remove_min(self):
65      """Remove and return (k,v) tuple with minimum key.
66
67      Raise Empty exception if empty.
68      """
69      if self.is_empty( ):
70        raise Empty('Priority queue is empty.')
71      self._swap(0, len(self._data) - 1)         # put minimum item at the end
72      item = self._data.pop( )                   # and remove it from the list;
73      self._downheap(0)                          # then fix new root
74      return (item._key, item._value)
```

Code Fragment 9.5: An implementation of a priority queue using an array-based heap (continued from Code Fragment 9.4).

9.3.5 Analysis of a Heap-Based Priority Queue

Table 9.3 shows the running time of the priority queue ADT methods for the heap implementation of a priority queue, assuming that two keys can be compared in $O(1)$ time and that the heap T is implemented with an array-based or linked-based tree representation.

In short, each of the priority queue ADT methods can be performed in $O(1)$ or in $O(\log n)$ time, where n is the number of entries at the time the method is executed. The analysis of the running time of the methods is based on the following:

- The heap T has n nodes, each storing a reference to a key-value pair.
- The height of heap T is $O(\log n)$, since T is complete (Proposition 9.2).
- The min operation runs in $O(1)$ because the root of the tree contains such an element.
- Locating the last position of a heap, as required for add and remove_min, can be performed in $O(1)$ time for an array-based representation, or $O(\log n)$ time for a linked-tree representation. (See Exercise C-9.34.)
- In the worst case, up-heap and down-heap bubbling perform a number of swaps equal to the height of T.

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

*amortized, if array-based

Table 9.3: Performance of a priority queue, P, realized by means of a heap. We let n denote the number of entries in the priority queue at the time an operation is executed. The space requirement is $O(n)$. The running time of operations min and remove_min are amortized for an array-based representation, due to occasional resizing of a dynamic array; those bounds are worst case with a linked tree structure.

We conclude that the heap data structure is a very efficient realization of the priority queue ADT, independent of whether the heap is implemented with a linked structure or an array. The heap-based implementation achieves fast running times for both insertion and removal, unlike the implementations that were based on using an unsorted or sorted list.

9.3.6 Bottom-Up Heap Construction ⋆

If we start with an initially empty heap, n successive calls to the add operation will run in $O(n \log n)$ time in the worst case. However, if all n key-value pairs to be stored in the heap are given in advance, such as during the first phase of the heap-sort algorithm, there is an alternative ***bottom-up*** construction method that runs in $O(n)$ time. (Heap-sort, however, still requires $\Theta(n \log n)$ time because of the second phase in which we repeatedly remove the remaining element with smallest key.)

In this section, we describe the bottom-up heap construction, and provide an implementation that can be used by the constructor of a heap-based priority queue.

For simplicity of exposition, we describe this bottom-up heap construction assuming the number of keys, n, is an integer such that $n = 2^{h+1} - 1$. That is, the heap is a complete binary tree with every level being full, so the heap has height $h = \log(n+1) - 1$. Viewed nonrecursively, bottom-up heap construction consists of the following $h + 1 = \log(n+1)$ steps:

1. In the first step (see Figure 9.5b), we construct $(n+1)/2$ elementary heaps storing one entry each.
2. In the second step (see Figure 9.5c–d), we form $(n+1)/4$ heaps, each storing three entries, by joining pairs of elementary heaps and adding a new entry. The new entry is placed at the root and may have to be swapped with the entry stored at a child to preserve the heap-order property.
3. In the third step (see Figure 9.5e–f), we form $(n+1)/8$ heaps, each storing 7 entries, by joining pairs of 3-entry heaps (constructed in the previous step) and adding a new entry. The new entry is placed initially at the root, but may have to move down with a down-heap bubbling to preserve the heap-order property.

   ⋮

i. In the generic i^{th} step, $2 \leq i \leq h$, we form $(n+1)/2^i$ heaps, each storing $2^i - 1$ entries, by joining pairs of heaps storing $(2^{i-1} - 1)$ entries (constructed in the previous step) and adding a new entry. The new entry is placed initially at the root, but may have to move down with a down-heap bubbling to preserve the heap-order property.

   ⋮

$h+1$. In the last step (see Figure 9.5g–h), we form the final heap, storing all the n entries, by joining two heaps storing $(n-1)/2$ entries (constructed in the previous step) and adding a new entry. The new entry is placed initially at the root, but may have to move down with a down-heap bubbling to preserve the heap-order property.

We illustrate bottom-up heap construction in Figure 9.5 for $h = 3$.

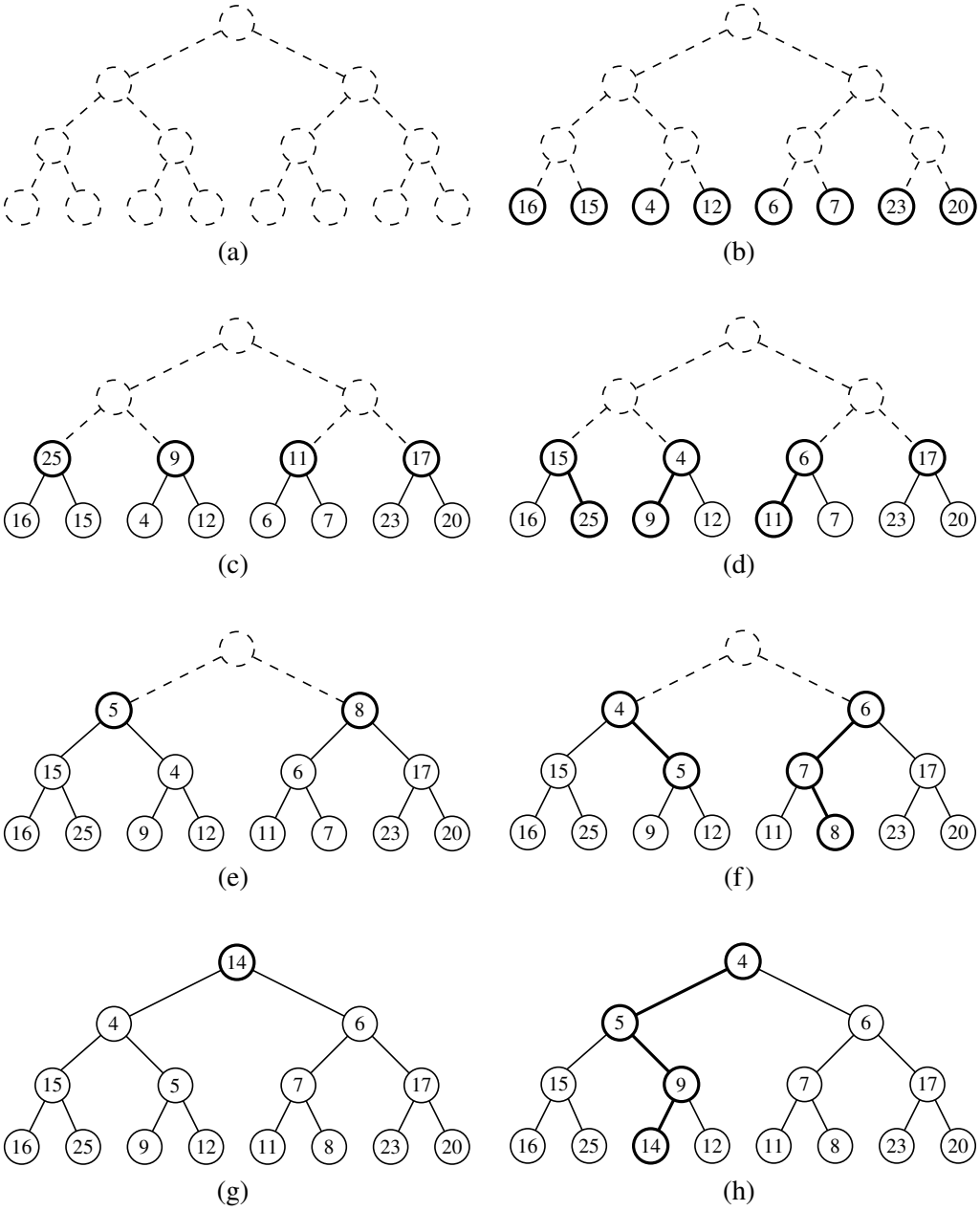


Figure 9.5: Bottom-up construction of a heap with 15 entries: (a and b) we begin by constructing 1-entry heaps on the bottom level; (c and d) we combine these heaps into 3-entry heaps, and then (e and f) 7-entry heaps, until (g and h) we create the final heap. The paths of the down-heap bubblings are highlighted in (d, f, and h). For simplicity, we only show the key within each node instead of the entire entry.

Python Implementation of a Bottom-Up Heap Construction

Implementing a bottom-up heap construction is quite easy, given the existence of a "down-heap" utility function. The "merging" of two equally sized heaps that are subtrees of a common position p, as described in the opening of this section, can be accomplished simply by down-heaping p's entry. For example, that is what happened to the key 14 in going from Figure 9.5(f) to (g).

With our array-based representation of a heap, if we initially store all n items in arbitrary order within the array, we can implement the bottom-up heap construction process with a single loop that makes a call to _downheap from each position of the tree, as long as those calls are ordered starting with the *deepest* level and ending with the root of the tree. In fact, that loop can start with the deepest nonleaf, since there is no effect when down-heap is called at a leaf position.

In Code Fragment 9.6, we augment the original HeapPriorityQueue class from Section 9.3.4 to provide support for the bottom-up construction of an initial collection. We introduce a nonpublic utility method, _heapify, that calls _downheap on each nonleaf position, beginning with the deepest and concluding with a call at the root of the tree. We have redesigned the constructor of the class to accept an optional parameter that can be any sequence of (k,v) tuples. Rather than initializing self._data to an empty list, we use a list comprehension syntax (see Section 1.9.2) to create an initial list of item composites based on the given contents. We declare an empty sequence as the default parameter value so that the default syntax HeapPriorityQueue( ) continues to result in an empty priority queue.

```
def __init__(self, contents=( )):
  """Create a new priority queue.

  By default, queue will be empty. If contents is given, it should be as an
  iterable sequence of (k,v) tuples specifying the initial contents.
"""
  self._data = [ self._Item(k,v) for k,v in contents ]      # empty by default
  if len(self._data) > 1:
    self._heapify( )

def _heapify(self):
  start = self._parent(len(self) - 1)       # start at PARENT of last leaf
  for j in range(start, -1, -1):            # going to and including the root
    self._downheap(j)
```

Code Fragment 9.6: Revision to the HeapPriorityQueue class of Code Fragments 9.4 and 9.5 to support a linear-time construction given an initial sequence of entries.

Asymptotic Analysis of Bottom-Up Heap Construction

Bottom-up heap construction is asymptotically faster than incrementally inserting n keys into an initially empty heap. Intuitively, we are performing a single down-heap operation at each position in the tree, rather than a single up-heap operation from each. Since more nodes are closer to the bottom of a tree than the top, the sum of the downward paths is linear, as shown in the following proposition.

Proposition 9.3: *Bottom-up construction of a heap with n entries takes $O(n)$ time, assuming two keys can be compared in $O(1)$ time.*

Justification: The primary cost of the construction is due to the down-heap steps performed at each nonleaf position. Let π_v denote the path of T from nonleaf node v to its "inorder successor" leaf, that is, the path that starts at v, goes to the right child of v, and then goes down leftward until it reaches a leaf. Although, π_v is not necessarily the path followed by the down-heap bubbling step from v, the length $\|\pi_v\|$ (its number of edges) is proportional to the height of the subtree rooted at v, and thus a bound on the complexity of the down-heap operation at v. We can bound the total running time of the bottom-up heap construction algorithm based on the sum of the sizes of paths, $\sum_v \|\pi_v\|$. For intuition, Figure 9.6 illustrates the justification "visually," marking each edge with the label of the nonleaf node v whose path π_v contains that edge.

We claim that the paths π_v for all nonleaf v are edge-disjoint, and thus the sum of the path lengths is bounded by the number of total edges in the tree, hence $O(n)$. To show this, we consider what we term "right-leaning" and "left-leaning" edges (i.e., those going from a parent to a right, respectively left, child). A particular right-leaning edge e can only be part of the path π_v for node v that is the parent in the relationship represented by e. Left-leaning edges can be partitioned by considering the leaf that is reached if continuing down leftward until reaching a leaf. Each nonleaf node only uses left-leaning edges in the group leading to that nonleaf node's inorder successor. Since each nonleaf node must have a different inorder successor, no two such paths can contain the same left-leaning edge. We conclude that the bottom-up construction of heap T takes $O(n)$ time. ■

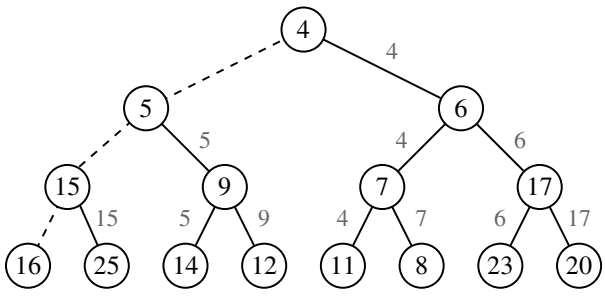


Figure 9.6: Visual justification of the linear running time of bottom-up heap construction. Each edge e is labeled with a node v for which π_v contains e (if any).

9.3.7 Python's heapq Module

Python's standard distribution includes a heapq module that provides support for heap-based priority queues. That module does not provide any priority queue class; instead it provides functions that allow a standard Python list to be managed as a heap. Its model is essentially the same as our own, with n elements stored in list cells $L[0]$ through $L[n-1]$, based on the level-numbering indices with the smallest element at the root in $L[0]$. We note that heapq does not separately manage associated values; elements serve as their own key.

The heapq module supports the following functions, all of which presume that existing list L satisfies the heap-order property prior to the call:

heappush(L, e): Push element e onto list L and restore the heap-order property. The function executes in $O(\log n)$ time.

heappop(L): Pop and return the element with smallest value from list L, and reestablish the heap-order property. The operation executes in $O(\log n)$ time.

heappushpop(L, e): Push element e on list L and then pop and return the smallest item. The time is $O(\log n)$, but it is slightly more efficient than separate calls to push and pop because the size of the list never changes. If the newly pushed element becomes the smallest, it is immediately returned. Otherwise, the new element takes the place of the popped element at the root and a down-heap is performed.

heapreplace(L, e): Similar to heappushpop, but equivalent to the pop being performed before the push (in other words, the new element cannot be returned as the smallest). Again, the time is $O(\log n)$, but it is more efficient that two separate operations.

The module supports additional functions that operate on sequences that do not previously satisfy the heap-order property.

heapify(L): Transform unordered list to satisfy the heap-order property. This executes in $O(n)$ time by using the bottom-up construction algorithm.

nlargest(k, iterable): Produce a list of the k largest values from a given iterable. This can be implemented to run in $O(n + k \log n)$ time, where we use n to denote the length of the iterable (see Exercise C-9.42).

nsmallest(k, iterable): Produce a list of the k smallest values from a given iterable. This can be implemented to run in $O(n + k \log n)$ time, using similar technique as with nlargest.

9.4 Sorting with a Priority Queue

In defining the priority queue ADT, we noted that any type of object can be used as a key, but that any pair of keys must be comparable to each other, and that the set of keys be naturally ordered. In Python, it is common to rely on the $<$ operator to define such an order, in which case the following properties must be satisfied:

- **Irreflexive property**: $k \not< k$.
- **Transitive property**: if $k_1 < k_2$ and $k_2 < k_3$, then $k_1 < k_3$.

Formally, such a relationship defines what is known as a ***strict weak order***, as it allows for keys to be considered equal to each other, but the broader equivalence classes are ***totally ordered***, as they can be uniquely arranged from smallest to largest due to the transitive property.

As our first application of priority queues, we demonstrate how they can be used to sort a collection C of comparable elements. That is, we can produce a sequence of elements of C in increasing order (or at least in nondecreasing order if there are duplicates). The algorithm is quite simple—we insert all elements into an initially empty priority queue, and then we repeatedly call remove_min to retrieve the elements in nondecreasing order.

An implementation of this algorithm is given in Code Fragment 9.7, assuming that C is a positional list. (See Chapter 7.4.) We use an original element of the collection as both a key and value when calling P.add(element, element).

```
1  def pq_sort(C):
2    """Sort a collection of elements stored in a positional list."""
3    n = len(C)
4    P = PriorityQueue( )
5    for j in range(n):
6      element = C.delete(C.first( ))
7      P.add(element, element)          # use element as key and value
8    for j in range(n):
9      (k,v) = P.remove_min( )
10     C.add_last(v)                    # store smallest remaining element in C
```

Code Fragment 9.7: An implementation of the pq_sort function, assuming an appropriate implementation of a PriorityQueue class. Note that each element of the input list C serves as its own key in the priority queue P.

With a minor modification to this code, we can provide more general support, sorting elements according to an ordering other than the default. For example, when working with strings, the $<$ operator defines a ***lexicographic ordering***, which is an extension of the alphabetic ordering to Unicode. For example, we have that `'12'` $<$ `'4'` because of the order of the first character of each string, just as

'apple' < 'banana'. Suppose that we have an application in which we have a list of strings that are all known to represent integral values (e.g., '12'), and our goal is to sort the strings according to those integral values.

In Python, the standard approach for customizing the order for a sorting algorithm is to provide, as an optional parameter to the sorting function, an object that is itself a one-parameter function that computes a key for a given element. (See Sections 1.5 and 1.10 for a discussion of this approach in the context of the built-in max function.) For example, with a list of (numeric) strings, we might wish to use the value of int(s) as a key for a string s of the list. In this case, the constructor for the int class can serve as the one-parameter function for computing a key. In that way, the string '4' will be ordered before string '12' because its key int('4') < int('12'). We leave it as an exercise to support such an optional key parameter for the pq_sort function. (See Exercise C-9.46.)

9.4.1 Selection-Sort and Insertion-Sort

Our pq_sort function works correctly given any valid implementation of the priority queue class. However, the running time of the sorting algorithm depends on the running times of the operations add and remove_min for the given priority queue class. We next discuss a choice of priority queue implementations that in effect cause the pq_sort computation to behave as one of several classic sorting algorithms.

Selection-Sort

If we implement P with an unsorted list, then Phase 1 of pq_sort takes $O(n)$ time, for we can add each element in $O(1)$ time. In Phase 2, the running time of each remove_min operation is proportional to the size of P. Thus, the bottleneck computation is the repeated "selection" of the minimum element in Phase 2. For this reason, this algorithm is better known as ***selection-sort***. (See Figure 9.7.)

As noted above, the bottleneck is in Phase 2 where we repeatedly remove an entry with smallest key from the priority queue P. The size of P starts at n and incrementally decreases with each remove_min until it becomes 0. Thus, the first operation takes time $O(n)$, the second one takes time $O(n-1)$, and so on. Therefore, the total time needed for the second phase is

$$O(n+(n-1)+\cdots+2+1) = O\left(\textstyle\sum_{i=1}^{n} i\right).$$

By Proposition 3.3, we have $\sum_{i=1}^{n} i = n(n+1)/2$. Thus, Phase 2 takes time $O(n^2)$, as does the entire selection-sort algorithm.

		Collection C	*Priority Queue P*
Input		$(7,4,8,2,5,3)$	$()$
Phase 1	(a)	$(4,8,2,5,3)$	(7)
	(b)	$(8,2,5,3)$	$(7,4)$
	$\vdots$	$\vdots$	$\vdots$
	(f)	$()$	$(7,4,8,2,5,3)$
Phase 2	(a)	(2)	$(7,4,8,5,3)$
	(b)	$(2,3)$	$(7,4,8,5)$
	(c)	$(2,3,4)$	$(7,8,5)$
	(d)	$(2,3,4,5)$	$(7,8)$
	(e)	$(2,3,4,5,7)$	(8)
	(f)	$(2,3,4,5,7,8)$	$()$

Figure 9.7: Execution of selection-sort on collection $C = (7,4,8,2,5,3)$.

Insertion-Sort

If we implement the priority queue P using a sorted list, then we improve the running time of Phase 2 to $O(n)$, for each remove_min operation on P now takes $O(1)$ time. Unfortunately, Phase 1 becomes the bottleneck for the running time, since, in the worst case, each add operation takes time proportional to the current size of P. This sorting algorithm is better known as ***insertion-sort*** (see Figure 9.8); in fact, our implementation for adding an element to a priority queue is almost identical to a step of insertion-sort as presented in Section 7.5.

The worst-case running time of Phase 1 of insertion-sort is

$$O(1+2+\ldots+(n-1)+n) = O\left(\textstyle\sum_{i=1}^{n} i\right).$$

Again, by Proposition 3.3, this implies a worst-case $O(n^2)$ time for Phase 1, and thus, the entire insertion-sort algorithm. However, unlike selection-sort, insertion-sort has a *best-case* running time of $O(n)$.

		Collection C	*Priority Queue P*
Input		$(7,4,8,2,5,3)$	$()$
Phase 1	(a)	$(4,8,2,5,3)$	(7)
	(b)	$(8,2,5,3)$	$(4,7)$
	(c)	$(2,5,3)$	$(4,7,8)$
	(d)	$(5,3)$	$(2,4,7,8)$
	(e)	(3)	$(2,4,5,7,8)$
	(f)	$()$	$(2,3,4,5,7,8)$
Phase 2	(a)	(2)	$(3,4,5,7,8)$
	(b)	$(2,3)$	$(4,5,7,8)$
	$\vdots$	$\vdots$	$\vdots$
	(f)	$(2,3,4,5,7,8)$	$()$

Figure 9.8: Execution of insertion-sort on collection $C = (7,4,8,2,5,3)$.

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 $i-1$.

3. In the second phase of the algorithm, we start with an empty sequence and move the boundary between the heap and the sequence from right to left, one step at a time. At step i, for $i = 1, \ldots, n$, we remove a maximum element from the heap and store it at index $n - i$.

In general, we say that a sorting algorithm is ***in-place*** if it uses only a small amount of memory in addition to the sequence storing the objects to be sorted. The variation of heap-sort above qualifies as in-place; instead of transferring elements out of the sequence and then back in, we simply rearrange them. We illustrate the second phase of in-place heap-sort in Figure 9.9.

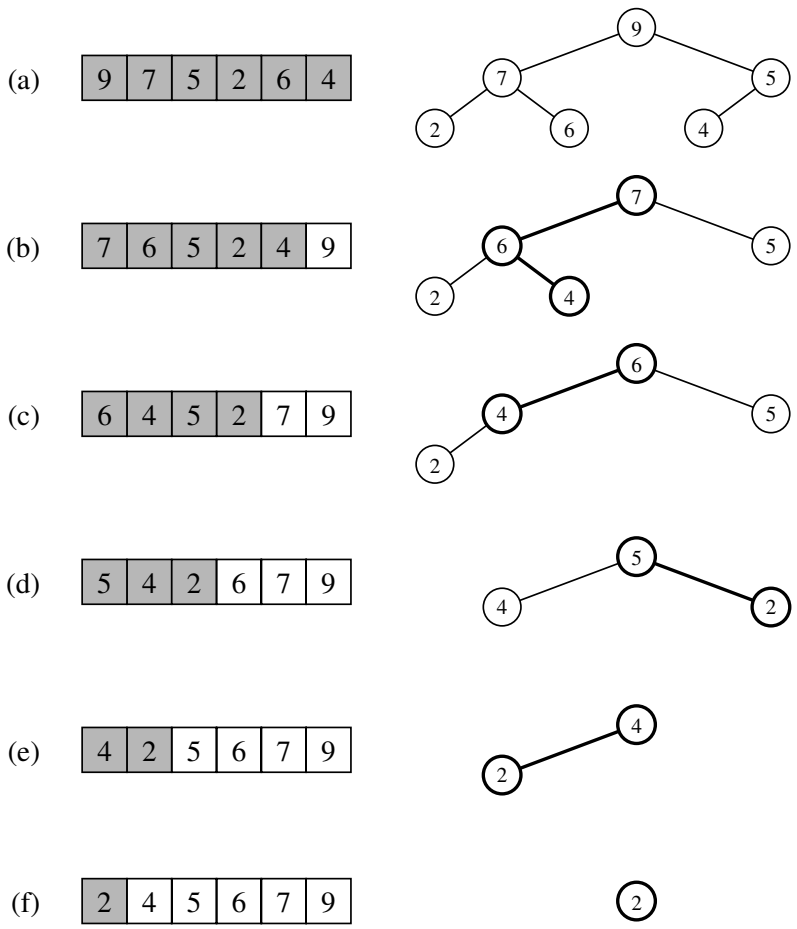


Figure 9.9: Phase 2 of an in-place heap-sort. The heap portion of each sequence representation is highlighted. The binary tree that each sequence (implicitly) represents is diagrammed with the most recent path of down-heap bubbling highlighted.

9.5 Adaptable Priority Queues

The methods of the priority queue ADT given in Section 9.1.2 are sufficient for most basic applications of priority queues, such as sorting. However, there are situations in which additional methods would be useful, as shown by the scenarios below involving the standby airline passenger application.

- A standby passenger with a pessimistic attitude may become tired of waiting and decide to leave ahead of the boarding time, requesting to be removed from the waiting list. Thus, we would like to remove from the priority queue the entry associated with this passenger. Operation remove_min does not suffice since the passenger leaving does not necessarily have first priority. Instead, we want a new operation, remove, that removes an arbitrary entry.
- Another standby passenger finds her gold frequent-flyer card and shows it to the agent. Thus, her priority has to be modified accordingly. To achieve this change of priority, we would like to have a new operation update allowing us to replace the key of an existing entry with a new key.

We will see another application of adaptable priority queues when implementing certain graph algorithms in Sections 14.6.2 and 14.7.1.

In this section, we develop an ***adaptable priority queue*** ADT and demonstrate how to implement this abstraction as an extension to our heap-based priority queue.

9.5.1 Locators

In order to implement methods update and remove efficiently, we need a mechanism for finding a user's element within a priority queue that avoids performing a linear search through the entire collection. To support our goal, when a new element is added to the priority queue, we return a special object known as a ***locator*** to the caller. We then require the user to provide an appropriate locator as a parameter when invoking the update or remove method, as follows, for a priority queue P:

P.update(loc, k, v): Replace key and value for the item identified by locator loc.

P.remove(loc): Remove the item identified by locator loc from the priority queue and return its (key,value) pair.

The locator abstraction is somewhat akin to the Position abstraction used in our positional list ADT from Section 7.4, and our tree ADT from Chapter 8. However, we differentiate between a locator and a position because a locator for a priority queue does not represent a tangible placement of an element within the structure. In our priority queue, an element may be relocated within our data structure during an operation that does not seem directly relevant to that element. A locator for an item will remain valid, as long as that item remains somewhere in the queue.

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.

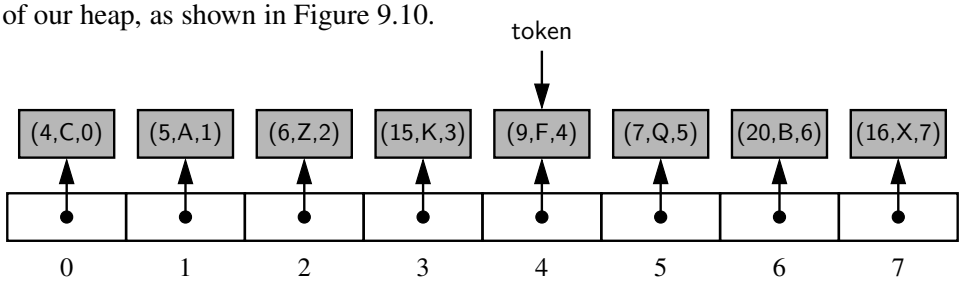


Figure 9.10: Representing a heap using a sequence of locators. The third element of each locator instance corresponds to the index of the item within the array. Identifier token is presumed to be a locator reference in the user's scope.

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

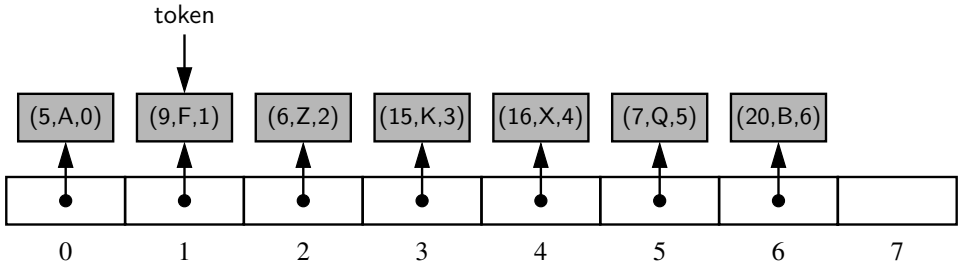


Figure 9.11: The result of a call to remove_min( ) on the heap originally portrayed in Figure 9.10. Identifier token continues to reference the same locator instance as in the original configuration, but the placement of that locator in the list has changed, as has the third field of the locator.

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-

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

```
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 $O(n)$.