## EXERCISES FOR SECTION 6.5

### SELF-CHECK

1. Show the tree that would be formed for the following data items. Exchange the first and last items in each list, and rebuild the tree that would be formed if the items were inserted in the new order.
   a. happy, depressed, manic, sad, ecstatic
   b. 45, 30, 15, 50, 60, 20, 25, 90
2. Explain how the tree shown in Figure 6.13 would be changed if you inserted *mother*. If you inserted *jane*? Does either of these insertions change the height of the tree?
3. Show or explain the effect of removing the nodes *kept*, *cow* from the tree in Figure 6.13.
4. In Exercise 3 above, a replacement value must be chosen for the node *cow* because it has two children. What is the relationship between the replacement word and the word *cow*? What other word in the tree could also be used as a replacement for *cow*? What is the relationship between that word and the word *cow*?
5. The algorithm for deleting a node does not explicitly test for the situation where the node being deleted has no children. Explain why this is not necessary.
6. In Step 19 of the algorithm for deleting a node, when we replace the reference to a node that we are removing with a reference to its left child, why is it not a concern that we might lose the right subtree of the node that we are removing?

### PROGRAMMING

1. Write methods `contains` and `remove` for the `BinarySearchTree` class. Use methods `find` and `delete` to do the work.
2. Self-Check Exercise 4 indicates that two items can be used to replace a data item in a binary search tree. Rewrite method `delete` so that it retrieves the leftmost element in the right subtree instead. You will also need to provide a method `findSmallestChild`.
3. Write a `main` method to test a binary search tree. Write a `toString` method that returns the tree contents in ascending order (using an inorder traversal) with newline characters separating the tree elements.
4. Write a `main` method for the index generator that declares new `Scanner` and `IndexGenerator` objects. The `Scanner` can reference any text file stored on your hard drive.

## 6.6 Heaps and Priority Queues

In this section, we discuss a binary tree that is ordered but in a different way from a binary search tree. At each level of a heap, the value in a node is less than all values in its two subtrees. Figure 6.20 shows an example of a heap. Observe that 6 is the smallest value. Observe that each parent is smaller than its children and that each parent has two children, with the exception of node 39 at level 3 and the leaves. Furthermore, with the exception of 66, all leaves are at the lowest level. Also, 39 is the next-to-last node at level 3, and 66 is the last (rightmost) node at level 3.

**FIGURE 6.20**
Example of a Heap

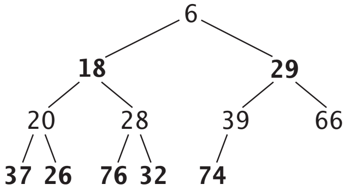


**FIGURE 6.21**
Inserting 8 into a Heap

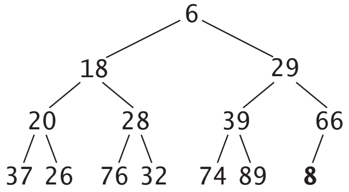


**FIGURE 6.22**
Swapping 8 and 66

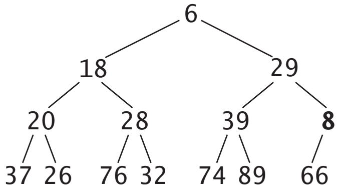
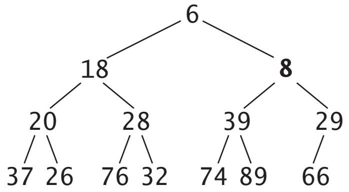


**FIGURE 6.23**
Swapping 8 and 29



More formally, a heap is a complete binary tree with the following properties:

- The value in the root is the smallest item in the tree.
- Every subtree is a heap.

## Inserting an Item into a Heap

We use the following algorithm for inserting an item into a heap. Our approach is to place each item initially in the bottom row of the heap and then move it up until it reaches the position where it belongs.

### Algorithm for Inserting in a Heap

```
1.  Insert the new item in the next position at the bottom of the heap.
2.  while new item is not at the root and new item is smaller than its parent
3.      Swap the new item with its parent, moving the new item up the heap.
```

New items are added to the last row (level) of a heap. If a new item is larger than or equal to its parent, nothing more need be done. If we insert 89 in the heap in Figure 6.20, 89 would become the right child of 39 and we are done. However, if the new item is smaller than its parent, the new item and its parent are swapped. This is repeated up the tree until the new item is in a position where it is no longer smaller than its parent. For example, let's add 8 to the heap shown in Figure 6.21. Since 8 is smaller than 66, these values are swapped as shown in Figure 6.22. Also, 8 is smaller than 29, so these values are swapped resulting in the updated heap shown in Figure 6.23. But 8 is greater than 6, so we are done.

## Removing an Item from a Heap

Removal from a heap is always from the top. The top item is first replaced with the last item in the heap (at the lower right-hand position) so that the heap remains a complete tree. If we used any other value, there would be a "hole" in the tree where that value used to be. Then the new item at the top is moved down the heap until it is in its proper position.

### Algorithm for Removal from a Heap

```
1.  Remove the item in the root node by replacing it with the last item in the heap (LIH).
2.  while item LIH has children, and item LIH is larger than either of its children
3.      Swap item LIH with its smaller child, moving LIH down the heap.
```

**FIGURE 6.24**
After Removal of 6

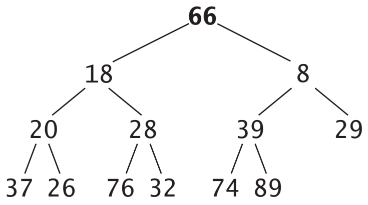


**FIGURE 6.25**
Swapping 66 and 8

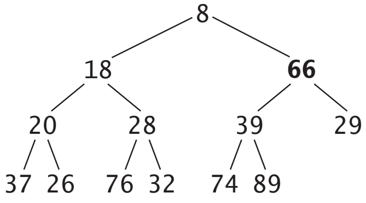


**FIGURE 6.26**
Swapping 66 and 29

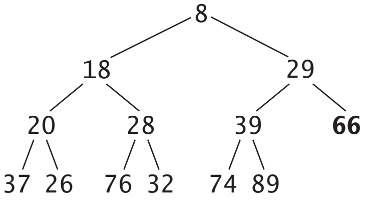
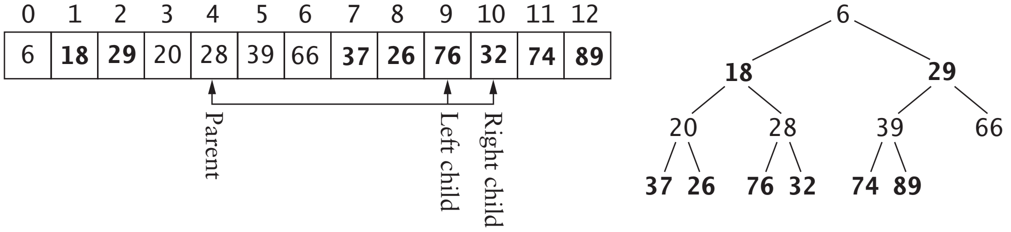


**FIGURE 6.27**
Internal Representation of the Heap



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 6 | **18** | **29** | 20 | 28 | 39 | 66 | **37** | **26** | **76** | **32** | **74** | **89** |

As an example, if we remove 6 from the heap shown in Figure 6.23, 66 replaces it as shown in Figure 6.24. Since 66 is larger than both of its children, it is swapped with the smaller of the two, 8, as shown in Figure 6.25. The result is still not a heap because 66 is larger than both its children. Swapping 66 with its smaller child, 29, restores the heap as shown in Figure 6.26.

## Implementing a Heap

Because a heap is a complete binary tree, we can implement it efficiently using an array (or `ArrayList`) instead of a linked data structure. We can use the first element (subscript 0) for storing a reference to the root data. We can use the next two elements (subscripts 1 and 2) for storing the two children of the root. We can use elements with subscripts 3, 4, 5, and 6 for storing the four children of these two nodes, and so on. Therefore, we can view a heap as a sequence of rows; each row is twice as long as the previous row. The first row (the root) has one item, the second row two, the third four, and so on. All of the rows are full except for the last one (see Figure 6.27).

Observe that the root, 6, is at position 0. The root's two children, 18 and 29, are at positions 1 and 2. For a node at position $p$, the left child is at $2p + 1$ and the right child is at $2p + 2$. A node at position $c$ can find its parent at $(c - 1) / 2$. Thus, as shown in Figure 6.27, children of 28 (at position 4) are at positions 9 and 10.

### Insertion into a Heap Implemented as an `ArrayList`

We will use an `ArrayList` for storing our heap because it is easier to expand and contract than an array. Figure 6.28 shows the heap after inserting 8 into position 13. This corresponds to inserting the new value into the lower right position as shown in the figure, right. Now we need to move 8 up the heap, by comparing it to the values stored in its ancestor nodes. The parent (66) is in position 6 (13 minus 1 is 12, divided by 2 is 6). Since 66 is larger than 8, we need to swap as shown in Figure 6.29.

Now the child is at position 6 and the parent is at position 2 (6 minus 1 is 5, divided by 2 is 2). Since the parent, 29, is larger than the child, 8, we must swap again as shown in Figure 6.30.

The child is now at position 2 and the parent is at position 0. Since the parent is smaller than the child, the heap property is restored. In the heap insertion and removal algorithms that follow, we will use `table` to reference the `ArrayList` that stores the heap. We will use `table[index]` to represent the element at position `index` of `table`. In the actual code, a subscript cannot be used with an `ArrayList`.

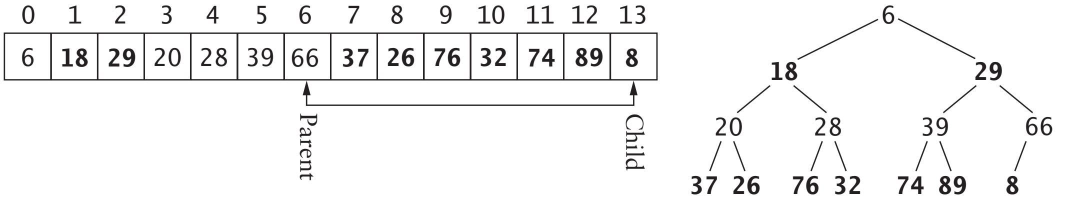


**FIGURE 6.28**
Internal Representation of Heap after Insertion

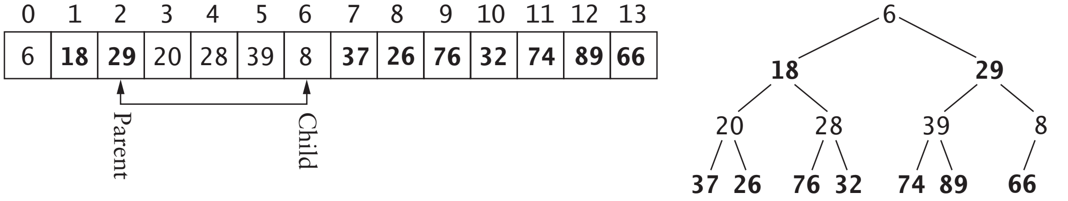


**FIGURE 6.29**
Internal Representation of Heap after First Swap

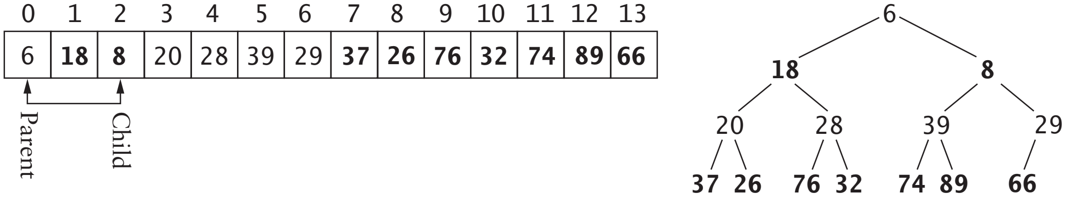


**FIGURE 6.30**
Internal Representation of Heap after Second Swap

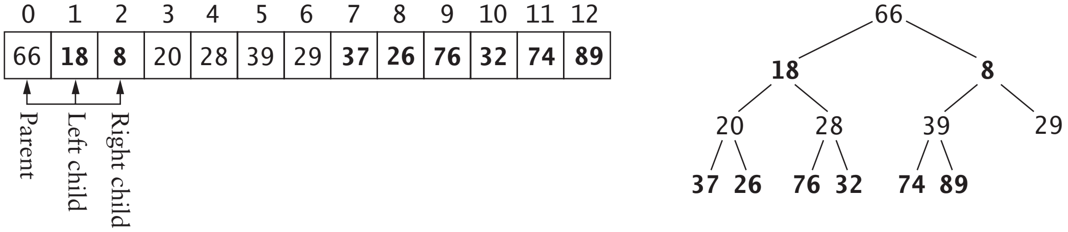


**FIGURE 6.31**
Internal Representation of Heap after 6 Is Removed

### Insertion of an Element into a Heap Implemented as an `ArrayList`

```
1.  Insert the new element at the end of the ArrayList and set child to
    table.size() - 1.
2.  Set parent to (child - 1) / 2.
3.  while (parent >= 0 and table[parent] > table[child])
4.      Swap table[parent] and table[child].
5.      Set child equal to parent.
6.      Set parent equal to (child - 1) / 2.
```

## Removal from a Heap Implemented as an `ArrayList`

In removing elements from a heap, we must always remove and save the element at the top of the heap, which is the smallest element. We start with an `ArrayList` that has been organized to form a heap. To remove the first item (6), we begin by replacing the first item with the last item and then removing the last item. This is illustrated in Figure 6.31. The new value of the root (position 0) is larger than both of its children (18 in position 1 and 8 in position 2). The smaller of the two children (8 in position 2) is swapped with the parent as shown in

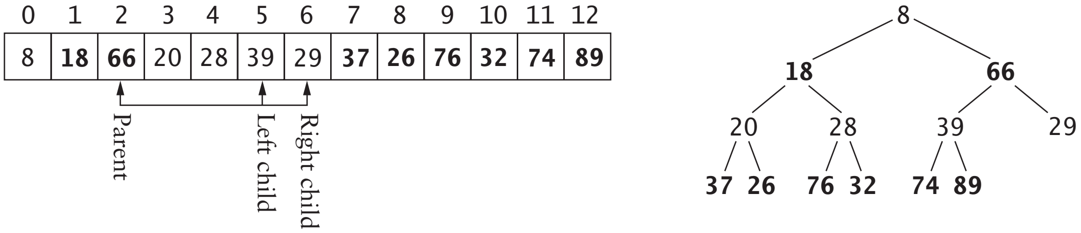


**FIGURE 6.32** Internal Representation of Heap after 8 and 66 Are Swapped

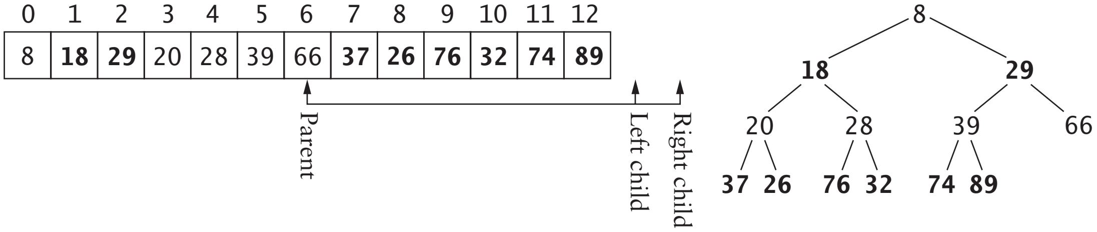


**FIGURE 6.33** Internal Representation of Heap after Swap of 66 and 29

Figure 6.32. Next, 66 is swapped with the smaller of its two new children (29), and the heap is restored (Figure 6.33).

The algorithm for removal from a heap implemented as an `ArrayList` follows.

### Removing an Element from a Heap Implemented as an `ArrayList`

```
1.   Remove the last element (i.e., the one at size() - 1) and set the item at 0
     to this value.
2.   Set parent to 0.
3.   while (true)
4.       Set leftChild to (2 * parent) + 1 and rightChild to leftChild + 1.
5.       if leftChild >= table.size()
6.           Break out of loop.
7.           Assume minChild (the smaller child) is leftChild.
8.           if rightChild < table.size() and
             table[rightChild] < table[leftChild]
9.               Set minChild to rightChild.
10.          if table[parent] > table[minChild]
11.              Swap table[parent] and table[minChild].
12.              Set parent to minChild.
             else
13.              Break out of loop.
```

The loop (Step 3) is terminated under one of two circumstances: either the item has moved down the tree so that it has no children (line 5 is true), or it is smaller than both its children (line 10 is false). In these cases, the loop terminates (line 6 or 13). This is shown in Figure 6.33. At this point the heap property is restored, and the next smallest item can be removed from the heap.

## Performance of the Heap

Method `remove` traces a path from the root to a leaf, and method `insert` traces a path from a leaf to the root. This requires at most *h* steps, where *h* is the height of the tree. The largest heap of height *h* is a full tree of height *h*. This tree has $2^h - 1$ nodes. The smallest heap of height $h$ is a complete tree of height $h$, consisting of a full tree of height $h - 1$, with a single node as the left child of the leftmost child at height $h - 1$. Thus, this tree has $2^{(h-1)}$ nodes. Therefore, both `insert` and `remove` are $O(\log n)$ where $n$ is the number of items in the heap.

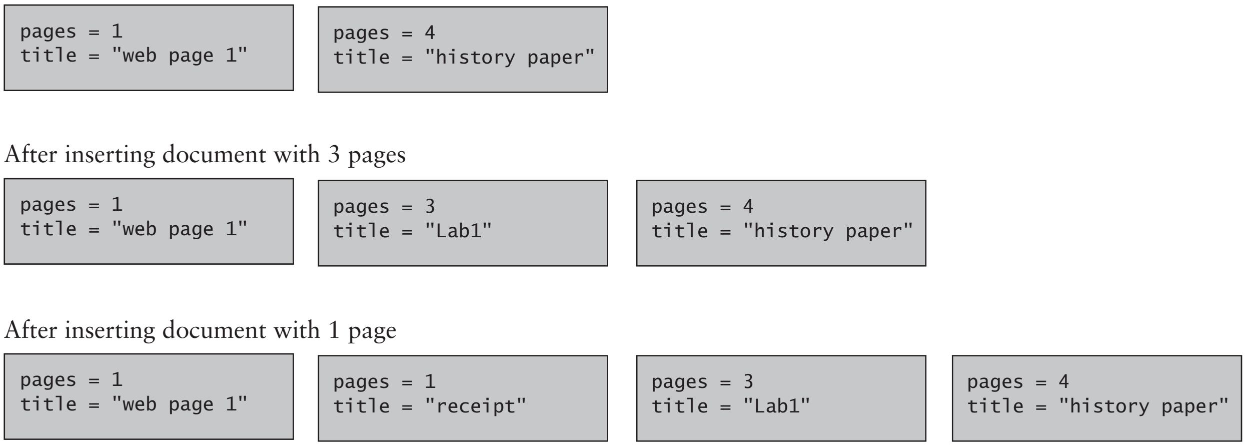
## Priority Queues

In computer science, a heap is used as the basis of a very efficient algorithm for sorting arrays, called heapsort, which you will study in Chapter 8. The heap is also used to implement a special kind of queue called a priority queue. However, the heap is not very useful as an abstract data type (ADT) on its own. Consequently, we will not create a `Heap` interface or code a class that implements it. Instead we will incorporate its algorithms when we implement a priority queue class and heapsort.

Sometimes a FIFO (first-in-first-out) queue may not be the best way to implement a waiting line. In a print queue, you might want to print a short document before some longer documents that were ahead of the short document in the queue. For example, if you were waiting by the printer for a single page to print, it would be very frustrating to have to wait until several documents of 50 pages or more were printed just because they entered the queue before yours did. Therefore, a better way to implement a print queue would be to use a priority queue. A *priority queue* is a data structure in which only the highest priority item is accessible. During insertion, the position of an item in the queue is based on its priority relative to the priorities of other items in the queue. If a new item has higher priority than all items currently in the queue, it will be placed at the front of the queue and, therefore, will be removed before any of the other items inserted in the queue at an earlier time. This violates the FIFO property of an ordinary queue.

**EXAMPLE 6.12** Figure 6.34 sketches a print queue that at first (top of diagram) contains two documents. We will assume that each document's priority is inversely proportional to its page count (priority is $\frac{1}{\text{page count}}$). The middle queue shows the effect of inserting a document three pages long. The bottom queue shows the effect of inserting a second one-page document. It follows the earlier document with that page length.

**FIGURE 6.34**
Insertion into a Priority Queue



After inserting document with 3 pages

After inserting document with 1 page

**TABLE 6.6**
Methods of the `PriorityQueue<E>` Class

| Method | Behavior |
|---|---|
| `boolean offer(E item)` | Inserts an item into the queue. Returns **`true`** if successful; returns **`false`** if the item could not be inserted |
| `E remove()` | Removes the smallest entry and returns it if the queue is not empty. If the queue is empty, throws a `NoSuchElementException` |
| `E poll()` | Removes the smallest entry and returns it. If the queue is empty, returns **`null`** |
| `E peek()` | Returns the smallest entry without removing it. If the queue is empty, returns **`null`** |
| `E element()` | Returns the smallest entry without removing it. If the queue is empty, throws a `NoSuchElementException` |

## The `PriorityQueue` Class

Java provides a `PriorityQueue<E>` class that implements the `Queue<E>` interface given in Chapter 4. The differences are in the specification for the `peek`, `poll`, and `remove` methods. These are defined to return the smallest item in the queue rather than the oldest item in the queue. Table 6.6 summarizes the methods of the `PriorityQueue<E>` class.

## Using a Heap as the Basis of a Priority Queue

The smallest item is always removed first from a priority queue (the smallest item has the highest priority) just as it is for a heap. Because insertion into and removal from a heap is **O**(log *n*), a heap can be the basis for an efficient implementation of a priority queue. We will call our class `KWPriorityQueue` to differentiate it from class `PriorityQueue` in the `java.util` API, which also uses a heap as the basis of its implementation.

A key difference is that class `java.util.PriorityQueue` class uses an array of type `Object[]` for heap storage. We will use an `ArrayList` for storage in `KWPriorityQueue` because the size of an `ArrayList` automatically adjusts as elements are inserted and removed. To insert an item into the priority queue, we first insert the item at the end of the `ArrayList`. Then, following the algorithm described earlier, we move this item up the heap until it is smaller than its parent.

To remove an item from the priority queue, we take the first item from the `ArrayList`; this is the smallest item. We then remove the last item from the `ArrayList` and put it into the first position of the `ArrayList`, overwriting the value currently there. Then, following the algorithm described earlier, we move this item down until it is smaller than its children or it has no children.

### Design of `KWPriorityQueue` Class

The design of the `KWPriorityQueue<E>` class is shown in Table 6.7. The data field `theData` is used to store the heap. We discuss the purpose of data field `comparator` shortly. We have added methods `compare` and `swap` to those shown earlier in Table 6.6. Method `compare` compares its two arguments and returns a type `int` value indicating their relative ordering. The class heading and data field declarations follow.

```
import java.util.*;

/** The KWPriorityQueue implements the Queue interface
    by building a heap in an ArrayList. The heap is structured
    so that the “smallest” item is at the top.
 */
```

```
public class KWPriorityQueue<E> extends AbstractQueue<E>
             implements Queue<E> {

    // Data Fields
    /** The ArrayList to hold the data. */
    private ArrayList<E> theData;
    /** An optional reference to a Comparator object. */
    Comparator<E> comparator = null;

    // Methods
    // Constructor
    public KWPriorityQueue() {
        theData = new ArrayList<>();
    }
. . .
```

**TABLE 6.7**
Design of `KWPriorityQueue<E>` Class

| Data Field | Attribute |
|---|---|
| `ArrayList<E> theData` | An `ArrayList` to hold the data |
| `Comparator<E> comparator` | An optional object that implements the `Comparator<E>` interface by providing a `compare` method |
| **Method** | **Behavior** |
| `KWPriorityQueue()` | Constructs a heap-based priority queue that uses the elements' natural ordering |
| `KWPriorityQueue (Comparator<E> comp)` | Constructs a heap-based priority queue that uses the `compare` method of `Comparator comp` to determine the ordering of the elements |
| `private int compare(E left, E right)` | Compares two objects and returns a negative number if object `left` is less than object `right`, zero if they are equal, and a positive number if object `left` is greater than object `right` |
| `private void swap(int i, int j)` | Exchanges the object references in `theData` at indexes `i` and `j` |

## The `offer` Method

The `offer` method appends the new item to the `ArrayList theData`. It then moves this item up the heap until the `ArrayList` is restored to a heap.

```
/** Insert an item into the priority queue.
    pre: The ArrayList theData is in heap order.
    post: The item is in the priority queue and
          theData is in heap order.
    @param item The item to be inserted
    @throws NullPointerException if the item to be inserted is null.
 */
@Override
public boolean offer(E item) {
    // Add the item to the heap.
    theData.add(item);
    // child is newly inserted item.
    int child = theData.size() - 1;
    int parent = (child - 1) / 2;  // Find child's parent.
    // Reheap
    while (parent >= 0 && compare(theData.get(parent),
                                  theData.get(child)) > 0) {
```

```
            swap(parent, child);
            child = parent;
            parent = (child - 1) / 2;
        }
        return true;
    }
```

## The poll Method

The poll method first saves the item at the top of the heap. If there is more than one item in the heap, the method removes the last item from the heap and places it at the top. Then it moves the item at the top down the heap until the heap property is restored. Next it returns the original top of the heap.

```
    /** Remove an item from the priority queue
        pre: The ArrayList theData is in heap order.
        post: Removed smallest item, theData is in heap order.
        @return The item with the smallest priority value or null if empty.
     */
    @Override
    public E poll() {
        if (isEmpty()) {
            return null;
        }
        // Save the top of the heap.
        E result = theData.get(0);
        // If only one item then remove it.
        if (theData.size() == 1) {
            theData.remove(0);
            return result;
        }

        /* Remove the last item from the ArrayList and place it into
           the first position. */
        theData.set(0, theData.remove(theData.size() - 1));
        // The parent starts at the top.
        int parent = 0;
        while (true) {
            int leftChild = 2 * parent + 1;
            if (leftChild >= theData.size()) {
                break; // Out of heap.
            }
            int rightChild = leftChild + 1;
            int minChild = leftChild;  // Assume leftChild is smaller.
            // See whether rightChild is smaller.
            if (rightChild < theData.size()
                && compare(theData.get(leftChild),
                theData.get(rightChild)) > 0) {
                minChild = rightChild;
            }
            // assert: minChild is the index of the smaller child.
            // Move smaller child up heap if necessary.
            if (compare(theData.get(parent),
                        theData.get(minChild)) > 0) {
                swap(parent, minChild);
                parent = minChild;
            } else { // Heap property is restored.
                break;
            }
        }
        return result;
    }
```

## The Other Methods

The iterator and size methods are implemented via delegation to the corresponding ArrayList methods. Method isEmpty tests whether the result of calling method size is 0 and is inherited from class AbstractCollection (a super interface to AbstractQueue). Methods peek and remove (based on poll) must also be implemented; they are left as exercises. Methods add and element are inherited from AbstractQueue where they are implemented by calling methods offer and peek, respectively.

## Using a Comparator

How do we compare elements in a PriorityQueue? In many cases, we will insert objects that implement Comparable<E> and use their natural ordering as specified by method compareTo. However, we may need to insert objects that do not implement Comparable<E>, or we may want to specify a different ordering from that defined by the object's compareTo method. For example, files to be printed may be ordered by their name using the compareTo method, but we may want to assign priority based on their length. The Java API contains the Comparator<E> interface, which allows us to specify alternative ways to compare objects. An implementer of the Comparator<E> interface must define a compare method that is similar to compareTo except that it has two parameters (see Table 6.7).

To indicate that we want to use an ordering that is different from the natural ordering for the objects in our heap, we will provide a constructor that has a Comparator<E> parameter. The constructor will set data field comparator to reference this parameter. Otherwise, comparator will remain null. To match the form of this constructor in the java.util.PriorityQueue class, we provide a first parameter that specifies the initial capacity of ArrayList theData.

```
/** Creates a heap-based priority queue with the specified initial
    capacity that orders its elements according to the specified
    comparator.
    @param cap The initial capacity for this priority queue
    @param comp The comparator used to order this priority queue
    @throws IllegalArgumentException if cap is less than 1
 */
public KWPriorityQueue(int cap, Comparator<E> comp) {
    if (cap < 1)
        throw new IllegalArgumentException();
    theData = new ArrayList<>();
    comparator = comp;
}
```

## The compare Method

If data field comparator references a Comparator<E> object, method compare will delegate the task of comparing its argument objects to that object's compare method. If comparator is **null**, the natural ordering of the objects should be used, so method compare will delegate to method compareTo. Note that parameter left is cast to type Comparable<E> in this case. In the next example, we show how to write a Comparator class.

```
/** Compare two items using either a Comparator object's compare method
    or their natural ordering using method compareTo.
    @pre: If comparator is null, left and right implement Comparable<E>.
    @param left One item
    @param right The other item
    @return Negative int if left less than right,
            0 if left equals right,
            positive int if left > right
    @throws ClassCastException if items are not Comparable
 */
```

```
@SuppressWarnings("unchecked")
private int compare(E left, E right) {
    if (comparator != null) {   // A Comparator is defined.
        return comparator.compare(left, right);
    } else {                    // Use left's compareTo method.
        return ((Comparable<E>) left).compareTo(right);
    }
}
```

**EXAMPLE 6.13** The class PrintDocument is used to define documents to be printed on a printer. This class implements the Comparable interface, but the result of its compareTo method is based on the name of the file being printed. The class also has a getSize method that gives the number of bytes to be transmitted to the printer and a getTimeStamp method that gets the time that the print job was submitted. Instead of basing the ordering on file names, we want to order the documents by a value that is a function of both size and the waiting time of a document. If we were to use either time or size alone, small documents could be delayed while big ones are printed, or big documents would never be printed. By using a priority value that is a combination, we achieve a balanced usage of the printer.

In Java 8, the Comparator interface is also defined as a functional interface (see Table 6.2). Its abstract method compare takes two arguments of the same type and returns an integer indicating their ordering. We can pass a lambda expression as a function parameter to the constructor that creates a KWPriorityQueue object. This method will implement the abstract compare method and determine the ordering of objects in the print queue.

The compare method for printQueue specified in the following fragment uses the weighted sum of the size and time stamp for documents left and right using the weighting factors P1 and P2. Method Double.compare in this fragment compares two double values and returns a negative value, 0, or a positive value depending on whether leftValue is <, equal to, or > rightValue.

```
final double P1 = 0.8;
final double P2 = 0.2;
Queue<PrintDocument> printQueue =
    new KWPriorityQueue<>(25, (left, right) -> {
        double leftValue = P1 * left.getSize() + P2 * left.getTimeStamp();
        double rightValue = P1 * right.getSize() + P2 * right.getTimeStamp();
        return Double.compare(leftValue, rightValue);
    });
```

Earlier versions of Java could not implement the Comparator object by passing a lambda expression to a constructor. The textbook website discusses how to define the Comparator object before Java 8.

## EXERCISES FOR SECTION 6.5

### SELF-CHECK

1. Show the heap that would be used to store the words *this, is, the, house, that, jack, built,* assuming they are inserted in that sequence. Exchange the order of arrival of the first and last words and build the new heap.
2. Draw the heaps for Exercise 1 above as arrays.

3. Show the result of removing the number 18 from the heap in Figure 6.26. Show the new heap and its array representation.
4. The heaps in this chapter are called min heaps because the smallest key is at the top of the heap. A max heap is a heap in which each element has a key that is smaller than its parent, so the largest key is at the top of the heap. Build the max heap that would result from the numbers 15, 25, 10, 33, 55, 47, 82, 90,18 arriving in that order.
5. Show the printer queue after receipt of the following documents:

| time stamp | size |
|---|---|
| 1100 | 256 |
| 1101 | 512 |
| 1102 | 64 |
| 1103 | 96 |

### PROGRAMMING

1. Complete the implementation of the `KWPriorityQueue` class. Write method `swap`. Also write methods `peek`, `remove`, `isEmpty`, and `size`.

## 6.7 Huffman Trees

In Section 6.1, we showed the Huffman coding tree and how it can be used to decode a message. We will now implement some of the methods needed to build a tree and decode a message. We will do this using a binary tree and a `PriorityQueue` (which also uses a binary tree).

A straight binary coding of an alphabet assigns a unique binary number *k* to each symbol in the alphabet $a_k$. An example of such a coding is Unicode, which is used by Java for the **`char`** data type. There are 65,536 possible characters, and they are assigned a number between 0 and 65,535, which is a string of 16 binary digit ones. Therefore, the length of a message would be 16 × *n*, where *n* is the total number of characters in the message. For example, the message “go eagles” contains 9 characters and would require 9 × 16 or 144 bits. As shown in the example in Section 6.1, a Huffman coding of this message requires just 38 bits.

Table 6.8, based on data published in Donald Knuth, *The Art of Computer Programming, Vol 3: Sorting and Searching* (Addison-Wesley, 1973), p. 441, represents the relative frequencies of the letters in English text and is the basis of the tree shown in Figure 6.35. The letter *e* occurs an average of 103 times every 1000 letters, or 10.3 percent of the letters are *e*s. (This is a useful table to know if you are a fan of *Wheel of Fortune*.) We can use this Huffman tree to encode and decode a file of English text. However, files may contain other symbols or may contain these symbols in different frequencies from what is found in normal English. For this reason, you may want to build a custom Huffman tree based on the contents of the file you are encoding. You would from attach this tree to the encoded file so that it can be used to decode the file. We discuss how to build a Huffman tree in the next case study.