18 Priority Queues And Heaps

Overview

You need to study at least the first part of Chapter Seventeen (Trees) before you study this chapter:

- Section 18.1 introduces the concept of a priority queue and discusses situations in which they can be useful, particularly file compression. A priority queue has the operations `add (Object)`, `removeMin()`, `peekMin()`, and `isEmpty()`.
- Section 18.2 presents the Comparator interface for generalizing ordering of data.
- Section 18.3 develops two implementations of the PriQue interface using arrays, one based on the insertion sort and the other based on the selection sort.
- Section 18.4 gives two corresponding implementations of PriQue using linked lists.
- Section 18.5 presents a fifth implementation that is more efficient when there are only a few different priority levels.
- Sections 18.6-18.7 describe how priority queues can be used directly for sorting. The TreeSort and HeapSort algorithms are discussed, as well as two implementations of PriQue based on binary trees and heaps. TreeSort is a variation of QuickSort.
- Sections 18.8-18.9 discuss the MergeSort algorithm for linked lists and go on to use a priority queue for sorting sequential files using the merging process. Two different implementations of sequential file sorting are presented.

18.1 File Compression Using Huffman Codes

All communications between spies employed by the ISPY Spy Company, Inc., are restricted to a vocabulary of 1024 very useful words. Each spy carries a dictionary in which each of the 1024 words has a 10-digit binary code, used in their digital messages (note that there are 1024 different 10-digit binary numbers, since $2^{10} = 1024$).

The company, having lost a few spies to the enemy due to messages taking too long to transmit, hires you to come up with a better plan for codes. You obtain a list of the frequency of use of each word in the last 10,000 messages and then develop a computer program that assigns binary codes to the 1024 words so that the average length of a message decreases by 23%, thereby saving the company an average of 2.7 spy lives per year and earning for yourself a $54,000 bonus.

At least, that is what you would do if you knew about the **Huffman code algorithm**. This algorithm provides the best possible assignment of binary codes as measured by average message length. The idea is that the words that are used frequently in spy messages (such as "war" and "kill") are assigned binary codes shorter than 10 binary digits (bits), while words used rarely (such as "peace" and "love") are assigned binary codes longer than 10 bits. That lowers the weighted average.

The algorithm begins by constructing 1024 2-node binary trees, each with a word in the left leaf of the root and the frequency with which that word occurs in the root (so each tree is "left-handed", in that it has an empty right subtree). These 1024 trees are put into a priority queue data structure. Among the services offered by a priority queue to the public are methods with the following headings for adding a data value to the structure and for removing and returning the data value with the lowest frequency:

```
public void add (Object ob)
public Object removeMin()
```

Next you repeat the following process 1023 times:

1. Remove from the priority queue the two trees with the smallest frequencies.
2. Combine those two trees into a single tree whose root contains the sum of the two frequencies.
3. Add that new combined tree to the priority queue data structure.

Eventually, this leaves the priority queue with a single binary tree containing all the words. The binary codes to be assigned to each word are easily derived from that tree by a process described later in this section.

Comparison with stacks and queues

A priority queue can be used for storing many kinds of data values, not just for storing binary trees that have a number in the root node. A priority queue uses a method that compares two data values to determine which has higher priority for removal from the data structure; in the case of the binary trees, a lower frequency gives a higher priority.

Assume you keep a pile of jobs to do, adding each new job you get to the top of the pile. The following contrasts priority queues with stacks and queues in terms of the way you choose the next job to carry out:

- If you always take the job on top, you are following a **Last-In-First-Out** principle (LIFO for short): Always take the job that has been in the pile for the shortest period of time. A data structure that implements this principle is called a **stack**.
- If you always take the job on the bottom, you are following a **First-In-First-Out** principle (FIFO for short): Always take the job that has been in the pile for the longest period of time. A data structure that implements this principle is called a **queue**.
- If you always take the job that has the highest priority, you are following a **Highest-Priority** principle: Always take the job in the pile that has the highest priority rating (according to whatever priority criterion you feel gives you the best chance of not getting grief from your boss). A data structure that implements this principle is called a **priority queue**.

The interface for priority queues

We will develop several different implementations of priority queues in this chapter. We specify what is common to all by defining a **PriQue** interface. An interface describes what operations the object can perform but does not give the coding. So it describes an **abstract data type**. The PriQue interface is in Listing 18.1 (see next page). A PriQue object has an operation to add an element to the data structure, an operation to remove an element, and two query methods: one to see what would be removed if requested, and one to tell whether the data structure has any elements to remove.

As an example of how a priority queue is used, the following coding applies the Huffman logic to a priority queue initially containing many of the 2-leaf binary trees. It leaves the priority queue with the single combined binary tree:

```
public void combineIntoOneTree (PriQue par)
{  TreeNode one = (TreeNode) par.removeMin();
   while (! par.isEmpty())
   {  TreeNode two = (TreeNode) par.removeMin();
      two.combineHuffmanly (one);
      par.add (two);
      one = (TreeNode) par.removeMin();
   }
   par.add (one);
}  //======================
```

Listing 18.1 The PriQue interface

```
public interface PriQue              // not in the Sun library
{
   /** Tell whether the priority queue has no more elements. */
   public boolean isEmpty();


   /** Return the object removeMin would return without modifying
    *  anything.  Throw an Exception if the queue is empty. */
   public Object peekMin();


   /** Delete the object of highest priority and return it.
    *  Priority is determined by a Comparator passed to the
    *  constructor.  Throw an Exception if the queue is empty. */
   public Object removeMin();


   /** Add the given element ob to the priority queue, so the
    *  priority queue has one more element than it had before.
    *  Precondition: The Comparator can be applied to ob. */
   public void add (Object ob);
}
```

For the given coding to work, you need to add a `combineHuffmanly` method to the TreeNode class of Chapter Seventeen that attaches the left subtree of `two` as the right subtree of `one` and then makes `one` the new left subtree of `two`. The method must also store the sum of the two frequencies in the root of `two`. This method is left as an exercise.

When you are done, you will have a "left-handed binary tree" (i.e., the right subtree is empty) with all 1024 words at the leafs and frequency values in the other nodes. Assign to each word the binary code corresponding to the path from itsRoot.itsLeft to the word, using 0 for a turn to the left and 1 for a turn to the right. For instance, if a word is stored in the leaf at (itsRoot.itsLeft).itsLeft.itsRight.itsLeft.itsRight.itsRight, its binary code will be 01011. This is provably the best possible correspondence of words to binary codes.

Figure 18.1 shows a smaller situation with eight words and their frequencies out of 100 occurrences ("peace" occurs 4 times and "war" appears 28 times). The first iteration of the Huffman algorithm combines the two rarest words, "peace" and "love", into one tree with frequency 9 and inserts it after the 7-tree and the 8-tree. The next iteration combines "push" and "hit" into one tree with frequency 15 and inserts it after the 10-tree. The third iteration would combine the 9-tree with the 10-tree.

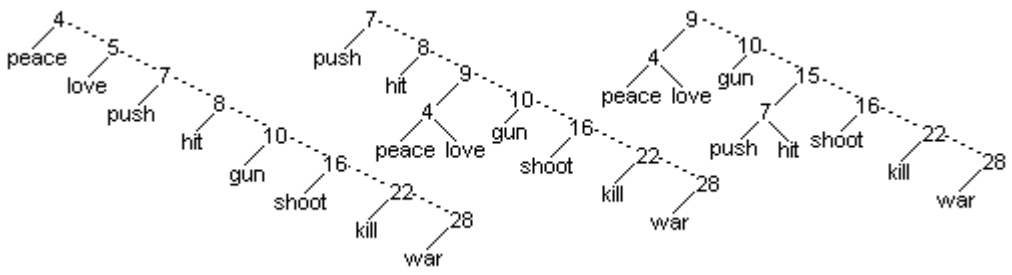
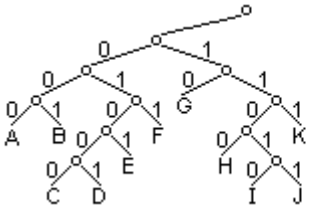


Figure 18.1 Huffman algorithm applied two times to 8 words

Figure 18.2 shows the binary codes that would be assigned to each of eleven 1-letter "words" if the final binary tree were what is shown on the left of the figure.



The binary codes for these 11 "words"

A 000	G 10
B 001	H 1100
C 01000	I 11010
D 01001	J 11011
E 0101	K 111
F 011	

Figure 18.2 A final Huffman tree and the resulting binary codes

File compression

Another situation in which Huffman coding is valuable is **file compression**. Consider a file of perhaps a million bytes of data. One **byte** is eight binary bits, so a byte ranges in value from -128 through 127. The UNIX file compression algorithm reads in the file and counts how many times each of the 256 byte values occurs. These frequencies are used in the Huffman coding algorithm to assign one sequence of bits to each byte value so that the more frequently-occurring byte values are assigned shorter bit sequences than the rarer byte values. This reduces the average bits per byte.

This process can lower the total size of the file by fifty percent or more. You can send the file, now perhaps less than half-a-million bytes, along with a 1K description of the encoding used, over the internet to another program that can decode the file using that description. The overall logic is in the accompanying design block.

STRUCTURED NATURAL LANGUAGE DESIGN for file compression

1. Read the given file, counting how many times each of the 256 different byte patterns occurs in that file. Each byte pattern is a "word" for purposes of this algorithm.
2. Put 256 different frequency+word binary trees into a priority queue, where higher priorities are given to trees with lower frequency.
3. Apply the Huffman process to obtain a single large binary tree that has 256 leaves, with one word in each leaf.
4. Calculate the binary code for each word from the path through the tree to that word. Some will be shorter than 8 bits and some longer but the average should be below 8.
5. Write a new file in which each byte pattern is replaced by its binary code.
6. Send the encoded file along with the short description of encoding.

Exercise 18.1 Write an independent method `public static void transfer (PriQue one, PriQue two)`: It transfers all elements from the first parameter into the second parameter. Precondition: Both parameters are non-null.
Exercise 18.2 After the third iteration of the Huffman algorithm for the example in Figure 18.1, what frequency is in the newly-inserted tree's root and what is in its left subtree?
Exercise 18.3 (harder) Write the `public void combineHuffmanly (TreeNode par)` TreeNode method as described in this section. Precondition: `par` is not null.
Exercise 18.4* Perform the fourth and fifth iterations of the Huffman algorithm for Figure 18.1 and show the sequence of binary trees.
Exercise 18.5* Perform the rest of the iterations of the Huffman algorithm for Figure 18.1 and show the sequence of binary trees. How much better is the average length of messages than the 3-bits-per-word length of messages without the coding?
Exercise 18.6** Write a program that applies the Huffman algorithm to any set of words and frequencies. Print a list of the words and binary codes, in their order left to right in the final binary tree. Use the TreeNode class, the String class for the words, and the Integer class for the frequencies. Note that this algorithm relies on the ability to store either of two kinds of Objects in a TreeNode.

18.2 Comparator Objects

A priority queue is appropriate when you have a number of jobs to do, each with its own priority. As time becomes available to do a job, you select the one with the highest priority. Whenever you receive another job to do, you add it to the list of jobs on hand. An example of this situation is a **printer queue**: A high-speed printer that serves a large number of users is continually receiving new print jobs to do, but prints those with highest priority first.

We assume that the priority is determined by an int-valued method named `compare` belonging to a **java.util.Comparator** object: For a given Comparator `test`, `test.compare(x,y)` returns a negative number if `x` has higher priority than `y`, a positive number if `x` has lower priority than `y`, and zero if they have the same priority. In other words, finding the value of higher priority corresponds to finding the smaller value (so priority 1 is the highest priority). The Sun library Comparator interface is shown in the upper part of Listing 18.2. The middle part of this listing contains a description of the two constructors that any PriQue implementation should have. The bottom part defines the most commonly-used kind of Comparator, the one that corresponds to `compareTo`.

Listing 18.2 The Comparator interface and how PriQues use it

```
public interface java.util.Comparator  // in the Sun library
{
   /** Tell whether one has higher priority than two.  Throw
    *  a ClassCastException if they cannot be compared with
    *  each other.  */
   public int compare (Object one, Object two);
}
//#################################################################



// the two standard constructors for an implementation of PriQue
public class Whatever implements PriQue
{
   private java.util.Comparator itsTest;


   public Whatever (java.util.Comparator givenTest)
   {  itsTest = givenTest;
   }  //======================


   public Whatever()
   {  itsTest = new Ascendor();
   }  //======================
}
//#################################################################



public class Ascendor implements java.util.Comparator
{
   public int compare (Object one, Object two)
   {  return ((Comparable) one).compareTo (two);
   }  //======================
}
```

When you construct a new PriQue object, you usually supply to the constructor the Comparator object you want that priority queue to use. So one constructor has a Comparator parameter that is assigned to an instance variable of the priority queue.

An appropriate Comparator class for the Huffman tree situation, assuming roots of trees contain Integers, is as follows. Then you would create the PriQue object to hold Huffman trees using e.g. `PriQue queue = new ArrayOutPriQue (new HuffmanComp())` (ArrayOutPriQue will be defined shortly as an implementation of PriQue):

```
public class HuffmanComp implements java.util.Comparator
{
   public int compare (Object one, Object two)
   {  return ((Integer) ((TreeNode) one).getData()).intValue()
          - ((Integer) ((TreeNode) two).getData()).intValue();
   }  //======================
}
```

The `compare` method sounds almost like the standard `compareTo` method that any Comparable class of objects has. Sometimes we will use the standard `compareTo` method to determine priority. As a convenience, an implementation of PriQue should have a constructor with no parameters, so that the new priority queue will use the standard `compareTo` method that belongs to the Comparable objects being stored in the priority queue. For instance, if you want to store String objects in alphabetical order, you can use `PriQue queue = new ArrayOutPriQue()`. This uses an Ascendor object (defined in Listing 18.2) to do the comparing.

The reason for using some Comparator's `compare` method rather than the data values' `compareTo` method is that we sometimes want to prioritize a class of objects on one criterion and sometimes on another. Comparators allow us to pass as a parameter an object that brings with it the appropriate `compare` function. Such an object is called a **functor** or **function object**, since its only purpose is to supply a function. Functors have instance methods but usually do not have instance variables.

Relation to stacks and (ordinary) queues

The PriQue methods correspond to the stack and queue methods described in Chapter Fourteen:

- `isEmpty` is the same as StackADT's `isEmpty` and QueueADT's `isEmpty`.
- `add` corresponds to `push` and `enqueue`;
- `peekMin` corresponds to `peekTop` and `peekFront`;
- `removeMin` corresponds to `pop` and `dequeue`;

The only real difference is in the effect of `removeMin` and `pop` and `dequeue`: `pop` delivers the object that has been in the data structure for the shortest period of time, `dequeue` delivers the object that has been in the data structure for the longest period of time, and `removeMin` delivers the object that has the highest priority. A PriQue implementation is **stable** if, in case of ties in priority, the element that has been in the data structure the longest is always removed first.

Another example

Suppose you have several classes with a `getCost()` method, all declared to implement this Priceable interface:

```
public interface Priceable
{     public int getCost();
}
```

Then you could use `PriQue queue = new ArrayOutPriQue (new ByCost());` to create the priority queue, with higher costs indicating higher priority, if you have the following definition:

```
public class ByCost implements java.util.Comparator
{
   public int compare (Object one, Object two)
   {  return ((Priceable) two).getCost()
           - ((Priceable) one).getCost();
   }  //======================
}
```

Whatever Comparator you define, it should have the properties of a **total ordering**. This means that, given two objects referenced by `sam` and `sue` for which `compare(sam, sue)` does not throw an Exception, the following should be true:

- If `sam.equals(sue)` is true, then `compare(sam,sue)` should be zero.
- The value of `compare(sue,sam)` should be the negative of `compare(sam, sue)` (i.e., one is positive and the other is negative, or else both are zero).
- If `compare(sam,x)` is positive and `compare(x,sue)` is positive, then `compare(sam,sue)` should be positive too. This is **transitivity**.

If the `compare` method you define can have `compare(sam,sue)` be zero only when `sam.equals(sue)` is true, it is said to be **consistent with `equals`**. If the `compare` method you define is not consistent with `equals`, then instances of some standard library classes (such as SortedSet) may refuse to let you add certain objects to the data structure.

Exercise 18.7 The java.awt.RectangularShape class in the Sun standard library has several subclasses such as Rectangle2D and Ellipse2D. The RectangularShape class has two instance methods `getX()` and `getY()` to retrieve the x- and y-coordinates of its upper-left corner on a Graphics page. A Graphics page has its non-negative coordinates numbered from 0 up starting at the upper-left corner of the page. Create a Comparator for RectangularShape objects so that higher objects have higher priority or, if both objects are at the same height, the one that is further right takes priority.

Exercise 18.8* Essay: Explain why any `compare` method that has the properties of a total ordering will also have this property: If `compare(x,y)` is negative and `compare(y,z)` is negative, then `compare(x,z)` is negative too.

Exercise 18.9* For the BigCircle class shown in Listing 17.13, write a constructor that has a Comparator object as a parameter. Use that Comparator object in the coding of the `get` method for BigCircles.

18.3 Implementing Priority Queues With Arrays

One obvious way to manage the values in a priority queue is to store them in a partially-filled array in the order in which they will be taken out, so that the next value to remove is always at `itsItem[itsSize-1]` (analogous to ArrayStack in Listing 14.2). This plan almost completely determines the coding for the four methods. Listing 18.3 contains this **ArrayOutPriQue** class. Note that `isEmpty`, `peekMin`, and `removeMin` are coded exactly the same as `isEmpty`, `peekTop`, and `pop` for ArrayStack. Reminder: IllegalStateException is in the java.lang package.

Listing 18.3 An array implementation of PriQue, partially done

```
public class ArrayOutPriQue implements PriQue
{
   private Object[] itsItem = new Object[10];
   private int itsSize = 0;
   private java.util.Comparator itsTest;


   public ArrayOutPriQue (java.util.Comparator givenTest)
   {  itsTest = givenTest;                                    //1
   }  //======================

   public ArrayOutPriQue()
   {  itsTest = new Ascendor();                               //2
   }  //======================

   public boolean isEmpty()
   {  return itsSize == 0;                                    //3
   }  //======================

   public Object peekMin()
   {  if (isEmpty())                                          //4
         throw new IllegalStateException ("priority Q is empty");
      return itsItem[itsSize - 1];                            //6
   }  //======================

   public Object removeMin()
   {  if (isEmpty())                                          //7
         throw new IllegalStateException ("priority Q is empty");
      itsSize--;                                              //9
      return itsItem[itsSize];                                //10
   }  //======================

   public void add (Object ob)
   {  if (itsSize == itsItem.length)                          //11
      {  }  // left as an exercise                           //12
      int k = itsSize;                                        //13
      while (k > 0 && itsTest.compare (ob, itsItem[k - 1]) >= 0)
      {  itsItem[k] = itsItem[k - 1];                         //15
         k--;                                                 //16
      }                                                       //17
      itsItem[k] = ob;                                        //18
      itsSize++;                                              //19
   }  //======================
}
```

If the `add` method is called when `itsSize` is `itsItem.length`, you need to increase the size of the array. This part of the coding of `add` is left as an exercise. In any case, when the `add` method is called, you move values up until you open up a spot where the new value goes to keep all values in order; it generally goes at an index `k` such that `itsTest.compareTo (ob, itsItem[k - 1]) < 0`.

Figure 18.3 illustrates the process (small numbers indicate high priority): Adding an element of priority 5 requires shifting the three elements of higher priorities 4, 2, and 1 further toward the rear of the array to make room for the 5 between the 6 and the 4.

indexes of components:	0	1	2	3	4	5	6
current status of the array:	8	6	4	2	1	no data	no data
call add(5):	8	6	**5**	4	2	1	no data
call removeMin():	8	6	5	4	2	no data	no data

Figure 18.3 For ArrayOutPriQue: Call add(5), then call removeMin()

Internal invariant for ArrayOutPriQues

- The int value `itsSize` is the number of elements in the abstract priority queue. These elements are stored in the array of Objects `itsItem` at the components indexed 0 up to but not including `itsSize`.
- If `k` is any positive integer less than `itsSize`, then the element at index `k` has either equal or higher priority than the element at index `k-1`; and if their priorities are equal, the element at index `k` has been in the queue longer. In particular, the element at index `itsSize-1` has the highest priority of all elements in the priority queue.

Storing the elements in the order they come in

An alternative implementation for a priority queue is to just put the next value added in the array at index `itsSize`. That way, data values remain in the order they are put in the structure. This means that `isEmpty` and `add` are coded the same as are `isEmpty` and `push` for ArrayStack.

When the `peekMin` method is called, you have to search through the array to find the value of highest priority and return it. In case of a tie, you select the one with the smaller index (since it came in earlier). Coding for the `peekMin` method of this **ArrayInPriQue** class of objects could be as shown in the upper part of Listing 18.4 (see next page).

If two elements are tied for the highest priority, this `removeMin` method does not necessarily return the one that has been on the queue for the longest time, so it is not a stable PriQue implementation (whereas ArrayOutPriQue is). Correcting this defect is left as an exercise. Notation: "ArrayIn" reminds you they are stored in an array in the order of input to the data structure; "ArrayOut" reminds you they are stored in an array in the order of output from the data structure.

The call of `itsTest.compare` can invoke the `compare` method that uses a String's `compareTo` method or the `compare` method that uses a Priceable object's `getCost` method or any of a number of other possible codings. It is therefore a polymorphic method call.

Listing 18.4 The ArrayInPriQue class of objects, partially done

```
public class ArrayInPriQue implements PriQue
{
   private Object[] itsItem = new Object[10];
   private int itsSize = 0;
   private java.util.Comparator itsTest;


   // the two constructors and isEmpty are as for ArrayOutPriQue
   // the add method is left as an exercise


   public Object peekMin()
   {  return itsItem[searchMin()];                              //1
   }  //======================


   private int searchMin()
   {  if (isEmpty())                                            //2
         throw new IllegalStateException ("priority Q is empty");
      int best = 0;                                             //4
      for (int k = 1;  k < itsSize;  k++)                       //5
      {  if (itsTest.compare (itsItem[k], itsItem[best]) < 0)//6
            best = k;                                           //7
      }                                                         //8
      return best;                                              //9
   }  //======================


   public Object removeMin()
   {  int best = searchMin();                                   //10
      itsSize--;                                                //11
      Object valueToReturn = itsItem[best];                     //12
      itsItem[best] = itsItem[itsSize];                         //13
      return valueToReturn;                                     //14
   }  //======================
}
```

Exercise 18.10 Write the missing part of the `add` method for ArrayOutPriQue.
Exercise 18.11 Write the entire `add` method for ArrayInPriQue.
Exercise 18.12 Revise the `removeMin` method for ArrayInPriQue so it always returns elements of equal priority in first-in-first-out order (i.e., maintain stability). Do so by replacing the next-to-last statement by coding that moves elements down while keeping their original order.
Exercise 18.13* Essay: Explain why you may delete both two-line if-statements beginning `if(isEmpty())` in Listing 18.3 without violating the specifications for PriQue, but you may not delete the two-line if-statement in Listing 18.4.
Exercise 18.14* Write out the internal invariant for the ArrayInPriQue class.
Exercise 18.15* Write a complete implementation of PriQue for which the element of highest priority is always kept in `itsItem[itsSize-1]` and the rest are in the order in which they were entered. Note that this makes the `peekMin` method execute very quickly. Do not try to make this implementation stable.
Exercise 18.16* A while-condition that makes two tests (as for the while-condition in the `add` method of Listing 18.3) is moderately slower than a loop that makes one test. Rewrite that loop to first test `itsTest.compare (ob, itsItem[0])` and then execute one of two different loops, each with a condition that only makes one test.

18.4 Implementing Priority Queues With Linked Lists

You can store the elements of a priority queue in a linked list instead of in an array. This section discusses two different ways to do this, analogous to the two ways implemented in the previous section. We put a private nested Node class in each of these linked list implementations, to maintain encapsulation.

Implementing PriQue with a linked list in the order elements go out

You could keep the data in order of priority, with the highest-priority data in the first Node of the linked list. Then when you want to remove it or just get it, you have it immediately available. This **NodeOutPriQue** class needs an instance variable to record the first Node on the linked list; call it `itsFirst`. The only tricky part is the `add` method, since it has to put the given element into the linked list after every value of equal or higher priority.

The coding of `add` is much easier if you always keep an extra Node at the end of the linked list with null data in that **trailer node**. So a NodeOutPriQue priority queue is empty whenever `itsFirst.itsNext` is null, not when `itsFirst` is null. The coding for `isEmpty` and `removeMin` is in the upper part of Listing 18.5 (see next page). We omit the coding for the two constructors (they are the same as usual) and `peekMin` (it is left as an exercise).

For the `add` method, you need to search through the linked list for the Node that contains the value that should come after the given element in the list. Have a variable `p` refer to that Node. The statement `p = p.itsNext` moves `p` from whichever Node it is on to the next Node. The loop stops at the trailer node if not before, i.e., when `p.itsNext == null`. Then you can insert the given element `ob` in that Node `p` and make a new Node after `p` to contain the data value that was in `p`, as follows:

```
p.itsNext = new Node (p.itsData, p.itsNext);
p.itsData = ob;
```

This would not work if `ob` is to be added at the end of a standard linked list, since then `p` would be null. But with the trailer node, when `ob` is to be added at the end, the statement above copies the information from the trailer node into a new trailer node and has `ob` replace the null data in the old trailer node. The coding is in the lower part of Listing 18.5. Figure 18.4 shows stages in the execution of the `add` method, where the numbers indicate the priority of the data value.

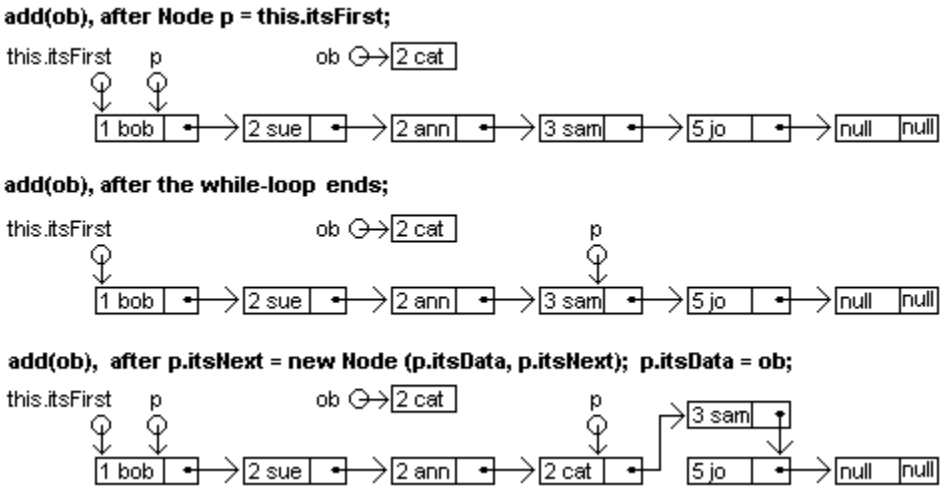


Figure 18.4 Adding a new element of priority 2 in NodeOutPriQue

Listing 18.5 The NodeOutPriQue class of objects, partially done

```
public class NodeOutPriQue implements PriQue
{
   private Node itsFirst = new Node (null, null); // trailer node
   private java.util.Comparator itsTest;


   public boolean isEmpty()
   {  return itsFirst.itsNext == null;                        //1
   }  //======================


   public Object removeMin()
   {  if (isEmpty())                                          //2
         throw new IllegalStateException ("priority Q is empty");
      Node toDiscard = itsFirst;                              //4
      itsFirst = itsFirst.itsNext;                            //5
      return toDiscard.itsData;                               //6
   }  //======================


   public void add (Object ob)
   {  Node p = this.itsFirst;                                 //7
      while (p.itsNext != null                                //8
                  && itsTest.compare (ob, p.itsData) >= 0) //9
      {  p = p.itsNext;                                       //10
      }                                                       //11
      p.itsNext = new Node (p.itsData, p.itsNext);            //12
      p.itsData = ob;                                         //13
   }  //======================


   private static class Node
   {
      public Object itsData;
      public Node itsNext;

      public Node (Object data, Node next)
      {  itsData = data;                                      //14
         itsNext = next;                                      //15
      }
   }  //======================
}
```

Implementing PriQue with a linked list in the order elements come in

The next implementation is **NodeInPriQue**, which implements a priority queue similar to ArrayInPriQue: Add each element to the front of the linked list as it comes in; when you need to produce the element of highest priority, search through the list to find it. Each NodeInPriQue object is to have an instance variable `itsFirst` to note the first Node on its linked list. Like NodeOutPriQue, the linked list is made with a private nested Node class and a trailer node.

The private `searchMin` method goes through the linked list to find the node containing the data value of highest priority. If the queue is empty, the `searchMin` coding throws an Exception when `p.itsNext` is evaluated, as it should. The `peekMin` method simply returns the data value in the node that `searchMin` finds. This coding is in the middle part of Listing 18.6 (see next page).

Listing 18.6 The NodeInPriQue class of objects, using trailer nodes

```
public class NodeInPriQue implements PriQue
{
   private Node itsFirst = new Node (null, null); // trailer node
   private java.util.Comparator itsTest;


   // the 2 constructors are the same as usual (see Listing 18.2)
   // the private Node class is the same as for NodeOutPriQue


   public boolean isEmpty()
   {  return itsFirst.itsNext == null;                          //1
   }  //======================


   public Object peekMin()
   {  return searchMin().itsData;                               //2
   }  //======================


   private Node searchMin()
   {  Node best = itsFirst;                                     //3
      for (Node p = itsFirst.itsNext;  p.itsNext != null;       //4
                                       p = p.itsNext)           //5
      {  if (itsTest.compare (p.itsData, best.itsData) <= 0) //6
            best = p;                                           //7
      }                                                         //8
      return best;                                              //9
   }  //======================


   public Object removeMin()
   {  Node best = searchMin();                                  //10
      Object valueToReturn = best.itsData;                      //11
      Node toDiscard = best.itsNext;                            //12
      best.itsData = toDiscard.itsData;                         //13
      best.itsNext = toDiscard.itsNext;                         //14
      return valueToReturn;                                     //15
   }  //======================


   public void add (Object ob)
   {  itsFirst = new Node (ob, itsFirst);                       //16
   }  //======================
}
```

An empty NodeInPriQue object is constructed with `itsFirst` being a reference to a trailer node with null for `itsData` and null for `itsNext`. Adding another data value just requires creating a new node to contain it and putting it at the beginning of the list:

```
itsFirst = new Node (ob, itsFirst);
```

The `removeMin` method does the same as `peekMin` except that it also deletes the value to be returned. You have to avoid changing the order of the remaining elements. To delete the element in a Node referenced by `best`, it is sufficient to replace it by the element in the Node following `best` and then delete the Node following `best`. This is

where the existence of the trailer node comes in handy. It guarantees that there will always be a Node following `best`, the trailer node if nothing else. The coding for the `removeMin` method is in the lower part of Listing 18.6.

Reminder Nesting only affects visibility: Methods inside NodeInPriQue can access the public variables of the Node class but outside methods cannot. So the principle of encapsulation is not violated by making Node's instance variables public.

An alternative to having a trailer node is to search through the list for the Node before the one that contains the highest-priority element and set that Node's `itsNext` value to the one after the one containing the highest-priority element. This is more complicated, so we leave that for a (hard) exercise. You could try it if you want, however, so you will understand why this book prefers to use a trailer node.

Exercise 18.17 Write the `peekMin` method for NodeOutPriQue.
Exercise 18.18 Rewrite the `removeMin` method for NodeInPriQue to not use a local variable for the Node to be deleted.
Exercise 18.19 How would you revise the `add` method for NodeOutPriQue if you were not to allow two elements with equal priority to be in the priority queue?
Exercise 18.20 Add a method `public int size()` to the NodeOutPriQue class: The executor counts and returns the number of values currently in the priority queue.
Exercise 18.21 Add a method `public String toString()` to the NodeOutPriQue class: The executor returns the concatenation of the string representation of each element currently in the queue with a tab character before each element, in order front to rear. This is very useful for debugging purposes.
Exercise 18.22 Write a method `public void removeAbove (Object ob)` that could be added to NodeOutPriQue: The executor removes all values from the stack that have higher priority than the parameter. Precondition: `itsTest` applies to `ob`.
Exercise 18.23 (harder) Same as the previous exercise, except for NodeInPriQue.
Exercise 18.24 (harder) Rewrite the `add` method for NodeInPriQue to add the given element in the second position of the linked list if the list is not empty and the element does not have higher priority than the data in the first Node. This will be used in the next exercise.
Exercise 18.25* Rewrite the `removeMin` method for NodeInPriQue so that, in conjunction with the `add` method of the preceding exercise, the element that is to be removed next is always `itsFirst.itsData` and the rest are in the order they were added. Note that this makes `peekMin` execute much faster.
Exercise 18.26* Explain why the condition in the private `searchMin` method of Listing 18.6 should not be written with `< 0` instead of `<= 0` for the comparison.
Exercise 18.27* Write out the internal invariant for the NodeOutPriQue class.
Exercise 18.28* Write a method `public void removeBelow (Object ob)` that could be added to NodeOutPriQue: The executor removes all values from the priority queue that have lower priority than the given element. Precondition: `itsTest` applies to `ob`.
Exercise 18.29* Same as the preceding exercise, but for NodeInPriQue.
Exercise 18.30* Rewrite the full implementation of NodeInPriQue so that each element is added to the end of a linked list but without looping (i.e., the reverse ordering). Hint: Add an instance variable that keeps track of the last node in the linked list.
Exercise 18.31* Essay: Explain why both of the linked list implementations of PriQue in this section are stable.
Exercise 18.32** Rewrite the full implementation of NodeOutPriQue without a trailer node, so that the number of Nodes equals the number of elements.
Exercise 18.33** Rewrite the full implementation of NodeInPriQue without a trailer node, so that the number of Nodes equals the number of elements.
Exercise 18.34** Write a method `public void add (NodeOutPriQue queue)` that could be in NodeOutPriQue: The executor interweaves `queue` 's elements in priority order and sets `queue` to be empty. Do not call on the existing `add` method.

Part B Enrichment And Reinforcement

18.5 Implementing Priority Queues With Linked Lists Of Queues

Some situations have only a few priority levels. For instance, you might have only five different priority levels with hundreds of elements at each level. If you add an element with low priority, you may have to go through many hundreds of elements to find the place where your given element is to be inserted.

In such a situation, the logic would execute substantially faster if you had just five regular queues, one for each priority level. Then you could go to the correct queue in at most five steps and quickly append the given element at the end of that queue. You could have a linked list of queues instead of a linked list of elements. You keep these queues in order of the priorities of the elements on them. The queue with the highest priority is first in the linked list.

This NodeGroupPriQue implementation uses a private nested Node class where `itsData` is a NodeQueue value instead of an Object value. Reminder: NodeQueue, an implementation of QueueADT, has these four methods for working with a FIFO queue:

- `q.isEmpty()` tells whether the queue has elements.
- `q.enqueue(ob)` adds ob to the rear of the queue.
- `q.dequeue()` removes and returns the element at the front of the queue.
- `q.peekFront()` returns the element at the front of the queue without removing it.

Each NodeGroupPriQue object has an instance variable `itsFirst` that keeps track of the linked list of queues; all of the queues on the list are to be non-empty. It is highly convenient to have a trailer node at the end of this linked list. Figure 18.5 shows roughly how this whole thing is structured: A NodeGroupPriQue's `itsFirst` is a Node whose `itsData` is a NodeQueue whose `itsFront` and `itsRear` are Nodes in a linked list of data (but the first kind of Node is a nested class; the second kind is stand-alone).

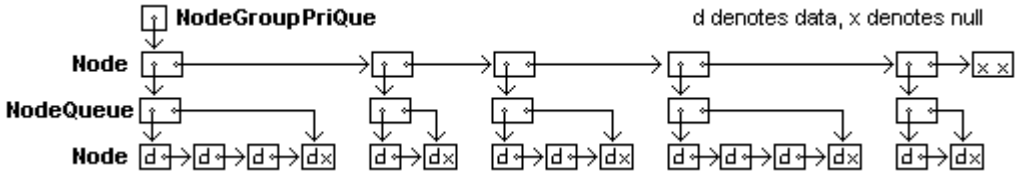


Figure 18.5 Linked list of Nodes p for which p.itsData is a NodeQueue

The isEmpty, peekMin, and removeMin methods

The `isEmpty` method only needs to check that `itsNext` for the first Node on the linked list is null. The element to be returned by the `peekMin` method is the front element of the queue with the highest priority. Since that is the first queue in the linked list, you return the `peekFront` value for that queue (since the queue is known to be non-empty). This coding is in the upper part of Listing 18.7 (see next page). Note that it will throw an Exception if there is no data, which is what it should do.

The `removeMin` method is about the same as the `peekMin` method except it returns `itsFirst.itsData.dequeue()`, which removes the front element on the queue of elements of highest priority. The call of `removeMin` might leave that first queue empty. Since the internal invariant of this implementation requires that you only store non-empty queues, you must discard the first queue if it has become empty as a consequence of dequeueing an element from it. This coding is in the middle part of Listing 18.7.

Listing 18.7 The NodeGroupPriQue class of objects

```
public class NodeGroupPriQue implements PriQue
{
   private Node itsFirst = new Node (null, null); // trailer node
   private java.util.Comparator itsTest;


   // the 2 constructors are the same as usual (see Listing 18.2)

   public boolean isEmpty()
   {  return itsFirst.itsNext == null;                          //1
   }  //======================

   public Object peekMin()
   {  return itsFirst.itsData.peekFront();                      //2
   }  //======================

   public Object removeMin()
   {  Object valueToReturn = itsFirst.itsData.dequeue();        //3
      if (itsFirst.itsData.isEmpty())  // after the dequeue     //4
         itsFirst = itsFirst.itsNext;  // lose the queue        //5
      return valueToReturn;
   }  //======================

   public void add (Object ob)
   {  Node p = this.itsFirst;                                   //6
      while (p.itsNext != null                                  //7
             && itsTest.compare (ob, p.itsData.peekFront()) > 0)
      {  p = p.itsNext;                                         //9
      }                                                         //10
      if (p.itsNext == null                                     //11
             || itsTest.compare (ob, p.itsData.peekFront()) < 0)
      {  p.itsNext = new Node (p.itsData, p.itsNext);           //13
         p.itsData = new NodeQueue();                           //14
      }                                                         //15
      p.itsData.enqueue (ob);                                   //16
   }  //======================


   private static class Node
   {
      public QueueADT itsData;
      public Node itsNext;

      public Node (QueueADT data, Node next)
      {  itsData = data;                                        //17
         itsNext = next;                                        //18
      }
   }  //======================
}
```

You should compare the coding throughout this Listing 18.7 with the coding for NodeOutPriQue in the earlier Listing 18.5 to see how similar they are. In fact, if no two elements can have the same priority, the NodeGroupPriQue implementation becomes the same as NodeOutPriQue except for an awful lot of extraneous creating and discarding of queue objects. Figure 18.6 (see next page) is the UML class diagram for the NodeGroupPriQue class.

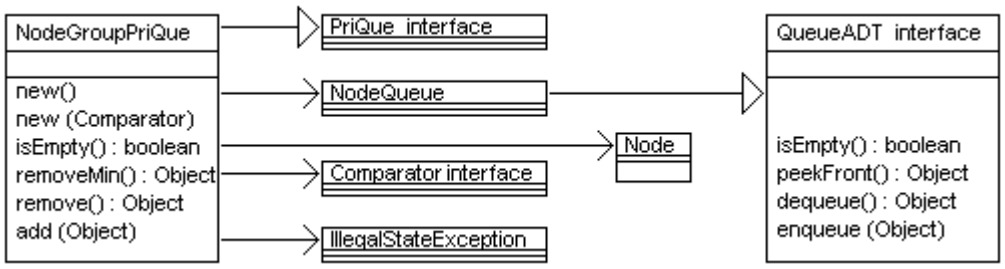


Figure 18.6 UML class diagram of the NodeGroupPriQue class

The add method

To add an element, you need to search for the first Node that contains a queue whose elements do not have higher priority than the given element. Of course, you do not go beyond the last Node in the linked list, which is the trailer node. So the condition for the while-statement is the logical equivalent of the following:

```
p.itsNext != null && ob > p.itsData.peekFront()
```

By contrast, the while-condition for NodeOutPriQue was the equivalent of the following, because you needed to go past not only elements of higher priority but also those of the same priority that had been on the linked list longer:

```
p.itsNext != null && ob >= p.itsData
```

Once you find the right Node `p`, see if it has a queue of elements with the same priority as the element you are supposed to add. If not (because the Node is empty or it has a queue of elements with lower priority), insert a new empty queue just before the queue in the Node you have found. In either case, you should now `enqueue` the given element onto that queue. This coding is in the lower part of Listing 18.7.

A variation

The implementation of PriQue just described works well as long as you know there are only a few different priority values, on the order of 5 to 20. You do not have to know what those values are.

Now if you know that the priority values are int values in the range from 1 to e.g. 100, you can have an implementation that uses an array of QueueADT objects, one for each int value. Adding a new element would involve finding its priority value `pv` and then storing that element on the queue at index `pv` in the array. And removing the element of highest priority would require searching the array for the first non-empty queue and dequeueing the front element on that queue. This is left as a major programming project.

Exercise 18.35 List the changes that Listing 18.7 would require if the `itsData` field of the Node class were declared as Object rather than as QueueADT.
Exercise 18.36* Write out the internal invariant for NodeGroupPriQue.
Exercise 18.37* Revise the Node class for Listing 18.7 to have a third instance variable `itsKey`, which is any element that is or has been on the queue stored in `itsData`. Use this revision to rewrite the `add` method to execute faster.

18.6 Sorting Using Priority Queues; The TreeSort Algorithm

A priority queue can be used to sort a large number of values. If you add them one at a time to the priority queue without ever calling `removeMin`, and then remove them one at a time until the priority queue is empty, they come out in the order determined by `compare`. If you have an Ascendor object (i.e., using `compareTo`), they come out in their natural ascending order. Specifically, the following method would sort the values in an ordinary queue, assuming that the given priority queue is initially empty:

```
public static void sort (QueueADT source, PriQue piq)
{  while ( ! source.isEmpty())                    // Loop #1
      piq.add (source.dequeue());
   while ( ! piq.isEmpty())                       // Loop #2
      source.enqueue (piq.removeMin());
}
```

In other situations, you might want a method to sort all the values in an array:

```
public static void sort (Object[] source, PriQue piq)
{  for (int k = source.length - 1;  k >= 0;  k--)  // Loop #1
      piq.add (source[k]);
   for (int k = 0;  k < source.length;  k++)       // Loop #2
      source[k] = piq.removeMin();
}
```

Either of the following statements does an InsertionSort, since the priority queue's `add` method inserts each value into its current list of values in ascending order and the `removeMin` method does virtually no work for these two "outie" implementations:

```
sort (source, new ArrayOutPriQue());
sort (source, new NodeOutPriQue());
```

Either of the following statements does a SelectionSort, since the priority queue's `removeMin` method runs through the entire list of values to select the minimum one and the `add` method does virtually no work for these two "innie" implementations:

```
sort (source, new ArrayInPriQue());
sort (source, new NodeInPriQue());
```

For the two InsertionSort cases, each call of `add` is a big-oh of N operation and each call of `removeMin` is big-oh of 1. So Loop #1 of the `sort` method takes big-oh of N^2 time, but Loop #2 takes big-oh of N time. For the two SelectionSort cases, each call of `removeMin` is a big-oh of N operation and each call of `add` is big-oh of 1. So Loop #2 of the `sort` method takes big-oh of N^2 time, but Loop #1 takes big-oh of N time.

When you use the priority queue implementation described at the end of the preceding section to sort values, for situations where priorities are int values in a limited range, you are using a BucketSort (which is more thoroughly described in Section 13.7).

Using the QuickSort logic

The QuickSort algorithm suggests another way to implement a priority queue that can execute much faster on average than the four elementary implementations mentioned above. You do it by keeping track of all the pivot elements used in sorting the elements so far. When you want to add another element `x`, you compare it with the first pivot (the first element that was added to the priority queue). If `x` has higher priority than the first pivot, it goes to the left of the first pivot; otherwise `x` goes to the right of the first pivot.

In the first case you compare `x` with the pivot that was used to split up the group of elements with higher priority than the first pivot. In the second case you compare `x` with the pivot that was used to split up the group of elements without higher priority than the first pivot. Either way, you put `x` to the left of that second pivot if `x` has higher priority than it, otherwise you put it to the right of that second pivot.

This continues until you see that `x` goes to one side of a pivot that does not already have any elements on that side. Then you put `x` in that position; it will be the pivot for future additions to the data structure at that point. However, if you use an array, you now have to move a lot of elements to make room for the new element, often more than half of them. This is a big-oh of N operation, which loses all the advantage of the QuickSort. Also, how are you going to keep track of all those pivot relationships?

Using TreeNodes

The standard solution is to use TreeNodes to store the values. In a TreeNode, `itsData` is the pivot for its group of elements. `itsLeft` refers to the TreeNode containing the pivot that was used for all elements that were compared with `itsData` and found to be of higher priority. `itsRight` refers to the TreeNode containing the pivot for all the elements that were compared with `itsData` and found to be of lesser or equal priority. A priority queue with this implementation could be called **TreePriQue**; see Listing 18.8.

Listing 18.8 The TreePriQue class of objects

```
public class TreePriQue implements PriQue
{
   private TreeNode itsRoot = TreeNode.ET;
   private java.util.Comparator itsTest;


   // the 2 constructors are the same as usual (see Listing 18.2)

   public boolean isEmpty()
   {  return itsRoot.isEmpty();
   }  //======================


   public Object peekMin()
   {  return itsRoot.firstNode().getData();
   }  //======================


   public void add (Object ob)
   {  if (itsRoot == TreeNode.ET)
         itsRoot = new TreeNode (ob);
      else
         itsRoot.add (ob, itsTest);
   }  //======================


   public Object removeMin()
   {  if (isEmpty())
         throw new IllegalStateException ("priority Q is empty");
      TreeNode[] newRoot = {itsRoot};
      Object valueToReturn = itsRoot.removeFirst (newRoot);
      itsRoot = newRoot[0];
      return valueToReturn;
   }  //======================
}
```

The first element added to the priority queue, with which you compare all other elements, is stored in a TreeNode named `itsRoot`. This is the only piece of information that the priority queue object has to keep track of. So you tell whether a priority queue is empty by seeing whether `itsRoot` value is `ET`. If you ask the priority queue to `get` the element of the highest priority, it asks `itsRoot` to find out and tell it (unless `itsRoot` is `ET`, in which case it throws an Exception). This coding is in the upper part of Listing 18.8. It calls the `firstNode` method given for the TreeNode class in Listing 17.2.

When you ask a TreePriQue object to add another element, there are two cases: If the TreePriQue object is empty, then the new element is added as the root value. Otherwise it asks `itsRoot` to add the element. This coding is in the middle part of Listing 18.8. It calls the `add` method in the TreeNode class, which is coded later in this section.

When you ask a TreePriQue object to remove the element of highest priority, it asks `itsRoot` to do so for it. But if it is empty, it throws an Exception. You could just code the `removeMin` method for a structure as `return itsRoot.removeFirst()` except for one problem: Sometimes it is the data in the root that is to be deleted, in which case you have to change `itsRoot` to be the root of whatever tree remains. So the TreeNode `removeFirst` method has to return two values: the `valueToReturn` and the new value of `itsRoot`. But a Java method can only return one value.

If you pass `itsRoot` as a parameter, any change that the method makes to the formal parameter does not change the value of the actual parameter `itsRoot`. A standard solution to the problem of getting two return values is to make one parameter be an array with one component. The method called can then change the value stored in that component if it needs to. The calling method can then use that new value to do what it has to do. This coding is in the lower part of Listing 18.8.

Figure 18.7 shows what we want to have happen when elements are added to and removed from the tree. Each little circle represents a TreeNode. The circle with the priority code 6 in it is `itsRoot`. The circle with the priority code 3 in it is `itsRoot.itsLeft` and the higher circle with the 8 in it is `itsRoot.itsRight`. Figure 18.7 traces the action of adding a value with priority 4, then adding a value with priority 7, then removing the value with the highest priority.

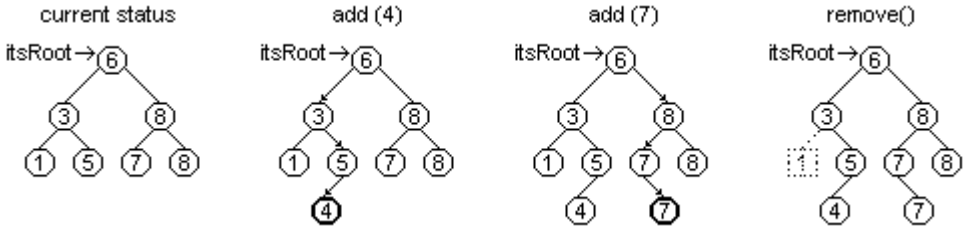


Figure 18.7 Effect on TreeNodes of some TreePriQue method calls

For the example in this figure, the `removeFirst` method would not change the parameter. But if the root node with the priority code 6 in it had no nodes to its left, a call of `removeFirst` would change the parameter: `root[0]` = the node with the 8.

Additions to the TreeNode class

The two new methods required for the TreeNode class are in Listing 18.9 (see next page). The `firstNode` method and the `getData` method are in Listing 17.2.

The `removeFirst` method first verifies that there is a data value to the left of the executor (line 1). If not, it must return the data value in the executor, and also "return" the TreeNode that will replace the executor in the TreePriQue structure (line 2; this "return" is done via the 1-element array parameter).

Listing 18.9 Additions to accomodate TreePriQue

```
// The following 2 methods are added to the TreeNode class

   /** Delete and return the first data value in a non-empty
    *  tree.  If that data value is in the executor, assign
    *  to root the TreeNode to the executor's right. */

   public Object removeFirst (TreeNode[] root)
   {  if (itsLeft == ET)                                        //1
      {  root[0] = this.itsRight;                               //2
         return this.itsData;                                   //3
      }                                                         //4
      TreeNode p = this;                                        //5
      while (p.itsLeft.itsLeft != ET)                           //6
         p = p.itsLeft; // p becomes parent of leftmost node    //7
      Object valueToReturn = p.itsLeft.itsData;                 //8
      p.itsLeft = p.itsLeft.itsRight; // it may be ET           //9
      return valueToReturn;                                     //10
   }  //======================


   /** Add ob to the tree, keeping the binary search property.
    *  Precondition:  this is not an empty tree.  */

   public void add (Object ob, java.util.Comparator test)
   {  if (test.compare (ob, itsData) < 0)                       //11
      {  if (itsLeft == ET)                                     //12
            itsLeft = new TreeNode (ob);                        //13
         else                                                   //14
            itsLeft.add (ob, test);                             //15
      }                                                         //16
      else                                                      //17
      {  if (itsRight == ET)                                    //18
            itsRight = new TreeNode (ob);                       //19
         else                                                   //20
            itsRight.add (ob, test);                            //21
      }                                                         //22
   }  //======================
```

If the executor of `removeFirst` has a TreeNode to its left (which therefore contains a higher-priority element than `this.itsData`), it can find the parent of the furthest-left Node and assign it to `p` (lines 5-7). It can then return the data value in `p.itsLeft`, since that data value has the highest priority of any. But first it must delete the Node `p.itsLeft` from the tree structure, replacing it by whatever was to the right of it (either null or another TreeNode). This coding is in the upper part of Listing 18.9.

The `add` method in the lower part of Listing 18.9 shows what a nonempty TreeNode object does when you ask it to `add` an element named `ob`: First it sees whether `ob` has higher priority than `itsData` (line 11). If so, `ob` goes to its left. If it does not have a TreeNode to its left (line 12), it can make a new one to hold `ob` (line 13), otherwise it can ask the TreeNode to its left to add `ob` (line 15). If `ob` goes to the right of the TreeNode executor, it works things out the same way but on the right instead of on the left. To accomplish all of this, the `add` method needs to have the Comparator object so it can make comparisons. So that Comparator object is passed as a parameter.

The TreeSort algorithm

The TreeNode implementation of a priority queue gives yet another way to sort an array of data: Add the values one at a time to a TreePriQue object until they are all in there. Then remove them one at a time; they will come out in order.

This **TreeSort** algorithm is essentially the QuickSort algorithm but with links rather than arrays. It executes about as fast as the QuickSort algorithm, but it takes up more space (one extra TreeNode of space for each data object means you need at least 3*N object references for N pieces of data). Both adding and removing are big-oh of log(N) operations on average for random sequences of data values. You can guarantee big-oh of log(N) as the worst-case behavior if you use a red-black or AVL tree, as described in Chapter Seventeen.

The coding in Listing 18.10 shows how the TreeSort algorithm could be written in the standard form used throughout Chapter Thirteen: You are to put the first `size` values of the array named `item` in ascending order using `compareTo`. It uses the traversal method from Listing 17.9. Obviously, it would be more efficient to have the last four lines transfer the data directly to the array instead of using a queue. This is left as an exercise.

Listing 18.10 Another sorting algorithm for the CompOp class

```
public static void treeSort (Comparable[] item, int size)
{  if (size <= 1)
      return;
   java.util.Comparator itsTest = new Ascendor();
   TreeNode root = new TreeNode (item[0]);
   for (int k = 1;  k < size;  k++)
      root.add (item[k], itsTest);  // coded in Listing 18.9
   NodeQueue queue = new NodeQueue();
   root.inorderTraverseLR (queue);  // coded in Listing 17.9
   for (int k = 0;  k < size;  k++)
      item[k] = (Comparable) queue.dequeue();
}  //======================
```

Historical Note Older programming languages that did not have the interface concept used "procedural parameters" instead. Specifically, you could write a method one of whose parameters was the `compare` method heading. The coding for your method called on `compare` as needed. A statement that called your method had to pass in a reference to the implementation of `compare` that it wanted your method to use. Java passes Comparator objects as parameters instead of passing `compare` methods.

Exercise 18.38 How would you code a `worstData` method for the TreeNode class to find the element with the lowest priority in a non-empty tree?
Exercise 18.39 (harder) It is often useful for debugging programs to have a method that prints all the values in a data structure. Write one for Listing 18.9 using `System.out.println`: Print all values in and below a given non-empty TreeNode in order of priority (left to right).
Exercise 18.40* Replace the last four lines of the `treeSort` method by coding that transfers the data values in the tree directly to the array. Have it call a recursive independent class method not in the TreeNode class. Write that method.
Exercise 18.41* Give an example of a sequence of data values that causes both `add` and `removeMin` to execute in big-oh of N time for a TreePriQue.
Exercise 18.42** Explain why TreePriQue is stable, and thus `treeSort` is a stable sorting algorithm, even though the `quickSort` it is based on is not.

18.7 Implementing Priority Queues Using Heaps; The HeapSort Algorithm

Adding one element to an ArrayOutPriQue or NodeOutPriQue object has a worst-case execution time that is big-oh of N, where N is the number of items already in the data structure. Removal is quite fast at big-oh of 1. For an ArrayInPriQue or NodeInPriQue object, adding has a worst-case execution time that is only big-oh of 1, but worst-case execution time for removal is big-oh of N.

This section shows you how to use a "heap" as the basis for a priority queue that executes very fast. Adding one value to the heap has a worst-case execution time that is big-oh of log(N), where N is the number of elements currently in the priority queue. And removal of one value from the heap also has a worst-case execution time that is big-oh of log(N). This is a great improvement over the InsertionSort and SelectionSort implementations discussed in the previous section, or even the TreeSort (which has average case big-oh of log(N) for both, but worst-case is big-oh of N for each).

Implementing a priority queue as a heap

The heap logic requires that you think of each component in an array as having two components as its "children". Specifically, the first component, `itsItem[0]`, has the next two components at indexes 1 and 2 as its children. Those two have the next four, at indexes 3 through 6, as their children -- 3 and 4 are the children of 1, and 5 and 6 are the children of 2. Those four have the next eight components as their children -- 7 and 8 are the children of 3, 9 and 10 are the children of 4, 11 and 12 are the children of 5, etc.

Figure 18.8 shows this relationship as a sort of genealogy tree. You start with one "node" of the tree at the top, then draw two children below it, then two children below each of those, repeating for as many elements as you have to store. Then you number the top node 0, number the next level with the next two integers 1 and 2, number the third level with the next 4 integers (3 through 6), number the fourth level with the next 8 integers (7 through 14), etc.

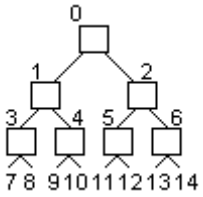


Figure 18.8 A tree

When you look at which numbers are "children" of which other numbers, you see that it can be expressed as a formula: The **children** of index n are indexes `2*n+1` and `2*n+2`. We also say that the elements at indexes `2*n+1` and `2*n+2` of the array are the children of the element at index `n`. Also, we say the element at index `n` is their parent; the formula for the **parent** of an index `k` is therefore `(k - 1) / 2`.

Our heap-based implementation **HeapPriQue** uses a partially-filled array (instance variables `Object[] itsItem` and `int itsSize`). You keep the array organized so that no child has higher priority than its parent. This is the **heap property**. That means that the data value with maximum priority is in `itsItem[0]`.

Adding a data value to the heap

Whenever you add another element to the priority queue whose data is already in `itsItem[0]` through `itsItem[itsSize-1]`, you start at `itsItem[itsSize]` and have the new data value "sift up" towards index 0. This means that you compare it with its parent (which would be at `itsItem[(itsSize-1)/2]`). If the parent has equal or higher priority than the new value, put the new data value in `itsItem[itsSize]`. Otherwise move the parent down to that component, then compare the new data value with the parent of the parent. Repeat until you can insert the new data value in a way that maintains the heap property.

Figure 18.9 illustrates this process for the first nine values added. The values inside the components are integers, to simplify the description, though of course objects are normally used. On each iteration (reading left to right on each level), one additional component is brought into compliance by swapping it up the tree until it is greater than all of its children. Data values are added in the order 15, 18, 12, 16, 17, 14, 11, 10, 13.

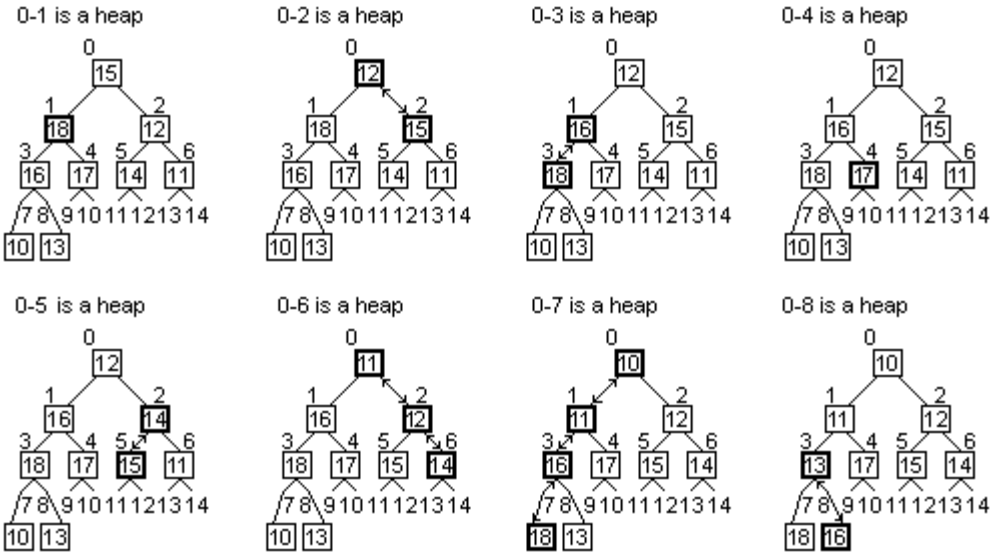


Figure 18.9 Adding values to the array, maintaining a heap

The coding in Listing 18.11 (see next page) embodies this logic for the `add` method. Compare it carefully with the coding for ArrayOutPriQue's `add` method in Listing 18.3. It is exactly the same except that `k - 1` has been replaced by `(k - 1) / 2` throughout. In other words, it looks like the `insertInOrder` logic except that each additional step is twice as close to index 0 as the step before instead of being just 1 component closer. For instance, when `add` is called for whatever element is at index 127, it is inserted into the sequence of values at indexes 63, 31, 15, 7, 3, 1, and 0. That makes the `add` method a big-oh of log(N) process instead of big-oh of N. An exercise makes this coding more efficient though not as clear.

Removing a data value from the heap

Whenever you remove a data value from the priority queue, you remove it from `itsItem[0]`. Then you readjust the array to be a heap without the removed value. This coding is in the lower part of Listing 18.11. The readjusting of the array is left to a private method named siftDown.

To readjust the array, take `itsItem[itsSize-1]` from the rear part of the array and insert it somewhere in the rest of the array; we will call that value `toInsert`.

Removing the highest-priority data value from index 0 left an empty component. Compare its two children at indexes 1 and 2 to see which has higher priority. That child (call it `kid`) moves up to index 0, which leaves an empty spot where it was. You then compare the two children of that empty spot to see which has higher priority and move that one up to the empty spot, leaving another empty spot on the third level (at index 3, 4, 5, or 6). The moving-up part is coded as `itsItem[empty] = itsItem[kid]`.

Listing 18.11 The HeapPriQue class of objects, partially done

```
public class HeapPriQue implements PriQue
{
   private Object[] itsItem = new Object[10];
   private int itsSize = 0;
   private java.util.Comparator itsTest;


   // the two constructors and isEmpty are as for ArrayOutPriQue

   public Object peekMin()
   {  if (isEmpty())                                          //1
         throw new IllegalStateException ("priority Q is empty");
      return itsItem[0];                                      //3
   }  //======================

   public void add (Object ob)
   {  if (itsSize == itsItem.length)                          //4
      {  }  // left as an exercise in an earlier section     //5
      int k = itsSize;                                        //6
      while (k > 0 && itsTest.compare (ob,                    //7
                              itsItem[(k - 1) / 2]) < 0)      //8
      {  itsItem[k] = itsItem[(k - 1) / 2];                   //9
         k = (k - 1) / 2;                                     //10
      }                                                       //11
      itsItem[k] = ob;                                        //12
      itsSize++;                                              //13
   }  //======================

   public Object removeMin()
   {  if (isEmpty())                                          //14
         throw new IllegalStateException ("priority Q is empty");
      Object valueToReturn = itsItem[0];                      //16
      itsSize--;                                              //17
      if (itsSize >= 2)                                       //18
         siftDown (itsItem[itsSize]);                         //19
      else if (itsSize == 1)                                  //20
         itsItem[0] = itsItem[1];                             //21
      return valueToReturn;                                   //22
   }  //======================
}
```

Keep this up until either the empty spot does not have two children or else both of its children have lower priority than `toInsert` has. Then put `toInsert` in the empty spot and stop. This coding is in Listing 18.12 (see next page). For example, in the final tree on the bottom-right of Figure 18.9, removing 10 would require that 11 move to index 0 (since 11 is smaller than its sibling 12), then 13 to index 1 (since 13 is smaller than its sibling 17), then 16 to index 3 (since 16 is smaller than its sibling 18). Figure 18.10 shows this application of `removeMin` and the next one as well (see next page).

Programming Style In Listing 18.12, some people would say that lines 8 and 9 should be replaced by a `break` statement. That would immediately terminate the loop and proceed with the statement at line 15. However, using a `break` statement to exit a loop generally makes coding harder to understand. In this case, the coding as shown actually executes faster without the `break` statement. You will not see the `break` statement used in this book. Any loop complex enough for a `break` statement is complex enough to put in a separate method where you use a `return` statement in place of a `break`.

Listing 18.12 The private siftDown method for HeapPriQue

```
/** Given that itsItem[0]..itsItem[itsSize] is a heap,
 *  in effect replace itsItem[0] by toInsert and then make
 *  the minimal changes to swap toInsert down so that
 *  itsItem[0]...itsItem[itsSize-1] is a heap again.  */

private void siftDown (Object toInsert)
{  int empty = 0;                                              //1
   int kid = 1;             // empty's child on the left      //2
   while (kid < itsSize)    // there are two children         //3
   {  if (itsTest.compare (itsItem[kid + 1],                   //4
                           itsItem[kid]) < 0)                  //5
         kid++;             // use the child on the right     //6
      if (itsTest.compare (toInsert, itsItem[kid]) < 0)        //7
      {  itsItem[empty] = toInsert;                            //8
         return;                                               //9
      }                                                        //10
      itsItem[empty] = itsItem[kid];                           //11
      empty = kid;                                             //12
      kid = 2 * empty + 1;  // empty's child on the left       //13
   }                                                           //14
   itsItem[empty] = toInsert;                                  //15
}  //======================
```

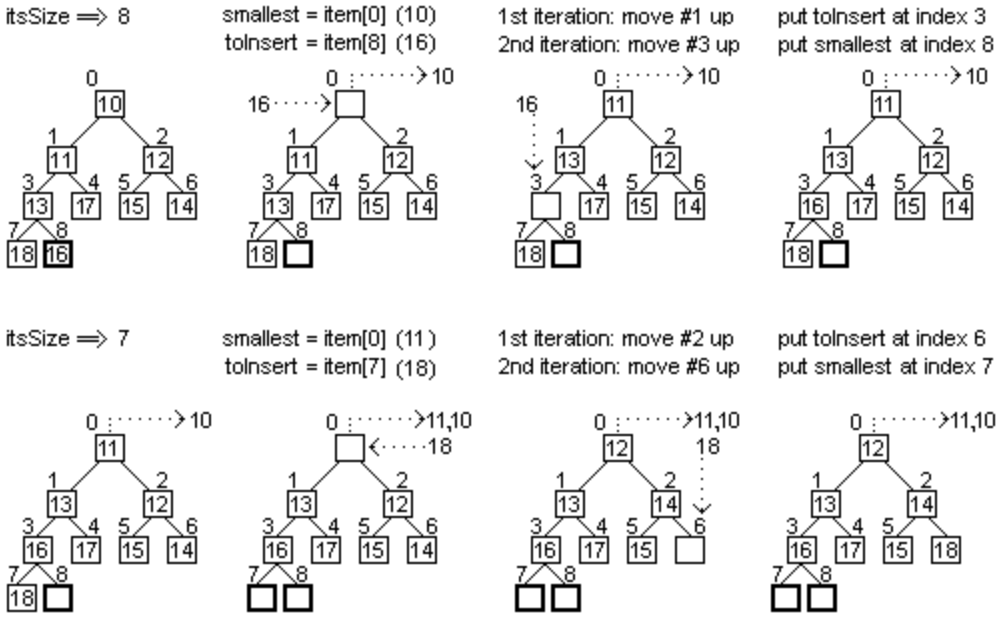


Figure 18.10 Two consecutive calls of the siftDown method

The HeapSort algorithm

A method to sort a given array of a given number of Comparable values using the **HeapSort** logic calls on a HeapPriQue constructor that assigns the given array to `itsItem`, then repeatedly applies the `add` method to create the initial heap. Finally, it repeatedly applies the `siftDown` method to get the data in descending order. Writing the constructor itself is left as an exercise. Execution time is big-oh of N*log(N).

```
public static void heapSort (Comparable[] item, int size)
{  new HeapPriQue (item, size);
}  //======================
```

For this repeated application of `siftDown`, note that the picture shows the values coming out of the array in the order 10, 11, etc., smallest to largest. But for the special constructor, we can put the values in the components that are opening up at the high indexes of the array (10 into index 8, later 11 into index 7, etc.). Of course, then they are in descending order, not ascending order. So all you need to do is reverse the meaning of the compare method to first obtain a heap with the larger values towards index 0, then repeatedly apply `siftDown` to get the values in ascending order. This is a very easy adjustment to make.

If the values are now in ascending order, they are in fact a heap with smaller values (i.e., higher priority) towards index 0. You can then continue to add and remove values to that heap, or simply stop if all you wanted was to have the values in ascending order.

Note the similarity with the SelectionSort logic: We select the largest of the remaining values and swap it with the element in the component where that largest value goes. But the process of selecting and then readjusting the heap executes in big-oh of log(N) time rather than in big-oh of N time. The reason is that the adjusting to restore the heap condition jumps through the index values, doubling the size of the jump at each iteration. By contrast, the SelectionSort has to go through every single value.

Comparison with other sorting methods

As the preceding discussion shows, the HeapSort requires big-oh of N*log(N) time, even in the worst case. This is far better worst-case performance than the QuickSort logic, which can sometimes degenerate to big-oh of N^2 execution time. Of course, the MergeSort also has big-oh of N*log(N) worst-case execution time, but it requires an extra array for storage, which doubles the storage requirements. The HeapSort does not require any significant extra storage (two or three variables for `kid`, `empty`, etc.).

On the other hand, the HeapSort is the most complex of these three sorting methods, and it is the slowest in terms of average execution time. HeapSort is also difficult to understand, particularly the fact that, after you organize all the values into a heap, they are still not sorted. A sample run with 20 different random sets of 100,000 Double values gave average execution times of 2.69 seconds for HeapSort, 1.45 seconds for QuickSort (Listing 13.5), and 1.35 seconds for MergeSort (Listing 13.6). In fact, the MergeSort beat the QuickSort in 18 of the 20 runs.

Exercise 18.43 Rewrite the `add` method of Listing 18.11 so that `(k - 1) / 2` is only calculated once for each value of `k`. Use a local int variable named `parent`.
Exercise 18.44 Write out a complete trace of the execution of `heapSort` on the array of values {3, 7, 4, 6}.
Exercise 18.45 Explain why the coding would on average execute more slowly if the tests in lines 18 and 20 of the `removeMin` method were done in the opposite order.
Exercise 18.46 (harder) Count the maximum number of possible comparisons of elements that `heapSort` makes when `size` is 3, then count them when `size` is 7.
Exercise 18.47* Explain why the ArrayOutPriQue implementation of a priority queue keeps the highest-priority element at the largest index in the array, but the HeapPriQue implementation keeps it at index 0, the smallest index in the array.
Exercise 18.48* Show that, if you make a heap and then perform an InsertionSort on it, you still have big-oh of N^2 execution time for the worst case. Hint: What is the worst case for the order of the elements in the lower half of the array?

Exercise 18.49* A faster version of the `heapSort` has the first stage make `item[k]` through `item[size-1]` a heap for values of k decreasing from `size/2` to 0. Write a private method `heapify (Comparable[] item, int size)` to do this. Explain why this first stage then executes in big-oh of N time.
Exercise 18.50** Write the new HeapPriQue constructor called by the `heapSort` constructor. Hint: The Comparator object should be the reverse of Ascendor, so that the initial heap has the largest value in index 0. Then each value removed from the heap should go directly to the end of the array. The final result is then in ascending order.
Exercise 18.51** Determine whether HeapPriQue is stable and give sound reasons.

18.8 MergeSort For A Linked List; Recurrence Relations

This section provides you more practice with linked lists and recursion, as well as reviewing the merge sort logic. This review is preparatory to the next section, which discusses a good method for sorting data values in a very large file.

Most sorting algorithms for arrays can be coded for linked lists as well. Clearly, the logic in NodeOutPriQue can easily be shaped to provide a linked list InsertionSort and the logic in NodeInPriQue can easily be shaped to provide a linked list SelectionSort. We next develop the MergeSort for linked lists.

The easiest version of merge sort for a linked list is done with a header node. For instance, the HeaderList class at the end of Chapter Fourteen uses header nodes. That is, a HeaderList object has two instance variables, itsData and itsNext, where itsNext is a HeaderList object. But we do not store data in the first node on the list.

So a public instance method to sort such a list could be coded as follows. This coding passes the part of the list after the header node, as well as the number of data values in that list, to a recursive method named `sorted`. That method will use the header node `this` to store information during execution of the merging part of the algorithm:

```
public void sort()
{  itsNext = sorted (itsNext, size());
}  //======================
```

For the MergeSort logic, we are to divide the list into two lists of equal size (or differing by 1 if necessary), sort each one separately, and then merge them back together into one list sorted in ascending order. First, we return the linked list immediately if it has less than two elements, since it is already sorted. Otherwise, we run half-way down the list and set `end` to the Node at the half-way point (node number `size / 2`). The second half of the list starts in `end.itsNext`. We break the list into the two equal parts, call the `sorted` method for each part, and then call a private `merged` method to merge the two sorted lists into one long sorted list. This coding is in the upper part of Listing 18.13 (see next page). It could be part of the HeaderList class if we just change "Node" to "HeaderList" throughout.

In this `merged` method, `one` and `two` denote the two sorted linked lists to be combined and placed on the completely sorted list. We add nodes from `one` and `two` to the rear of the list that begins with the header node `this`. When we exit the method, we return the first node after the header node `this`, which will be the first node in the completely sorted list.

Listing 18.13 The MergeSort logic for a linked list with no data in the first Node

```
public void sort()
{  itsNext = sorted (itsNext, size());                          //1
}  //======================

private Node sorted (Node item, int size)
{  if (size < 2)                                                //2
      return item;                                              //3
   int halfSize = size / 2;                                     //4
   Node end = item;                                             //5
   for (int k = 1;  k < halfSize;  k++)                         //6
      end = end.itsNext;                                        //7
   Node secondHalf = end.itsNext;                               //8
   end.itsNext = null;                                          //9
   return merged (sorted (item, halfSize),                      //10
                  sorted (secondHalf, size - halfSize));        //11
}  //======================

private Node merged (Node one, Node two)
{  Node rear = this;  // last node of sorted                    //12
   while (one != null && two != null)                           //13
   {  if (((Comparable) one.itsData)                            //14
                     .compareTo (two.itsData) <= 0)             //15
      {  rear.itsNext = one;                                    //16
         one = one.itsNext;                                     //17
      }                                                         //18
      else                                                      //19
      {  rear.itsNext = two;                                    //20
         two = two.itsNext;                                     //21
      }                                                         //22
      rear = rear.itsNext;                                      //23
   }                                                            //24
   rear.itsNext = (one == null)  ?  two  :  one;                //25
   return this.itsNext;                                         //26
}  //======================
```

So the `merged` logic repeatedly looks at the first node on each of `one` and `two`; whichever contains the smaller data is attached to the rear of the sorted list and detached from its own list (that is, detached from the `one` list or the `two` list). When one of the lists runs out of Nodes, the remainder of the other list is attached to the rear of the sorted list and the result is returned. This coding is in the lower part of Listing 18.13. Figure 18.11 shows how a list of eight data values would be sorted using this coding.

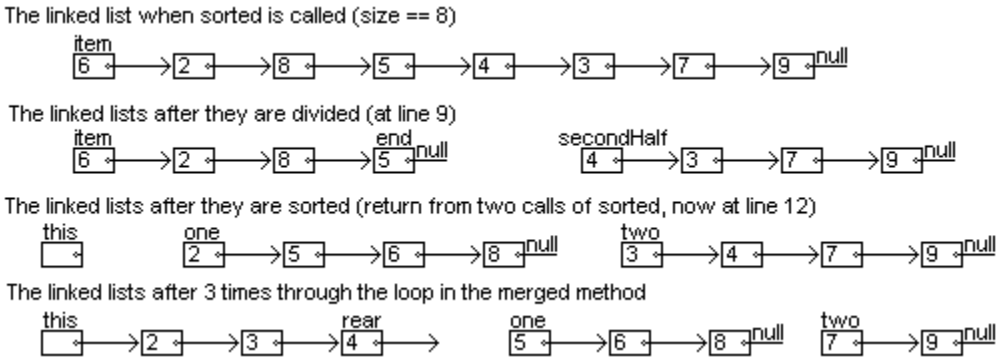


Figure 18.11 Stages in the execution of the merge sort

Recurrence relations

If you study the coding in Listing 18.13 carefully, you see that the maximum number of comparisons of data made by the MergeSort coding for n data values can be expressed by a recursive formula, written directly from the recursive coding. This is called a **recurrence relation**:

```
compsMS (n) = 0 if n <= 1, otherwise
compsMS (n) = compsMS (n/2) + compsMS (n - n/2) + (n - 1)
```

Any recurrence relation implies a corresponding recursive method for calculating the value. Call this method with any value of n to have the computer calculate it for you:

```
public static int compsMS (int n)  // for MergeSort
{  return n <= 1  ?  0
          : compsMS (n / 2) + compsMS (n - n / 2) + (n - 1);
}  //======================
```

Four more recurrence relations

The TreeSort coding towards the end of Section 18.6 adds data values one at a time to a binary search tree and then traverses the tree to obtain the values in order. When is the minimum number of comparisons for all the additions made? When half of the data values compared with the data in any given TreeNode go to the left and the rest go to the right. In that case, the number of comparisons made in building a tree of n nodes would be expressed by the following recurrence relation. Note that the first data value goes in the root and each data value thereafter is compared with the root, which makes n-1 root comparisons:

```
compsTS (n) = 0 if n <= 1, otherwise
compsTS (n) = compsTS (n/2) + compsTS (n - n/2) + (n - 1)
```

This is of course the same recurrence relation for compsMS, so the best case for TreeSort is the same as the worst case for MergeSort.

The InsertionSort repeatedly inserts a new value in an existing list of values where it goes in ascending order. So the maximum number of comparisons made can be expressed by the following recurrence relation (this is also the recurrence relation for SelectionSort):

```
compsIS (n) = 0 if n <= 1, otherwise
compsIS (n) = compsIS (n - 1) + (n - 1)
```

The HeapSort repeatedly inserts a new value in a heap of values to maintain the max-heap property. For instance, inserting the 64th value (at index 63) will require comparisons with at most the values at indexes 31, 15, 7, 3, 1, and 0. That is, the number of comparisons is at most log2(n). So an upper limit on the number of comparisons made during the insertion process that builds the heap can be expressed by the following recurrence relation:

```
compsHSBuild (n) = 0 if n <= 1, otherwise
compsHSBuild (n) = compsHSBuild (n - 1) + log2(n)
```

If you do the HeapSort directly on n data values without using a priority queue, you can build the initial heap faster. Define the "subheap at index k" to be the value at index k plus its two children plus their children, etc., going no further than n of course. Then you build the subheap at a given index k only after you have built the subheap for both of its children. A bit of thought and drawing pictures expresses the number of comparisons required in the worst case by the following recurrence relation, at least for any n that is 1 less than a power of 2 (i.e., n being one of 1, 3, 7, 15, 31, 63, etc.):

```
compsHSBuild2(n) = 0 if n <= 1, otherwise
compsHSBuild2(n) = 2*compsHSBuild2(n/2) + 2*log2((n + 1)/2)
```

Closed forms of recurrence relations

We would like to have an upper bound on the value of compsXX(n) as a function of n, without the recursive call of the function. This is called a **closed form**. We can prove by induction that compsMS is bounded above by n*log2(n) (and thus so is compsTS, since they have the same recurrence relation). We use here the facts that log2(2) is 1, log2(1) is 0, and log2(n/x) = log2(n) - log2(x):

*To Prove compsMS(n) <= n * log2(n) for any n >= 1 that is a power of 2.*
Basic Step when n = 1: compsMS(n) is zero, which is less-equal 1 * log2(1).
Inductive Step To show that compsMS(n) <= n * log2(n) in situations where n > 1 and we know the relation is true for all smaller powers of 2, we reason as follows:

compsMS(n) = 2 * compsMS(n/2) + (n - 1) // combines the first two terms
<= 2 * (n/2 * log2(n/2)) + (n - 1) // since it is true for smaller powers of 2
= n * (log2(n/2)) + (n - 1)
= n * (log2(n) - log2(2)) + (n - 1)
= n * log2(n) - n * 1 + n - 1 = n * log2(n) - 1 < n * log2(n)

Conclusion The truth of the assertion for all positive powers of 2 follows by the Induction Principle from the Basis Step and the Inductive Step.

We can also prove by induction that compsHSBuild is bounded above by n*log2(n):

*To Prove compsHSBuild(n) <= n * log2(n) for any n >= 1.*
Basic Step when n = 1: compsHSBuild(n) is zero, which is less-equal 1 * log2(1).
Inductive Step To show that compsHSBuild(n) <= n * log2(n) in situations where n > 1 and we know the relation is true for all smaller values of n, we reason as follows:

compsHSBuild(n) = compsHSBuild(n-1) + log2(n) // known recurrence relation
<= (n-1) * log2(n-1) + log2(n) // since it is true for smaller values of n
<= (n-1) * log2(n) + log2(n)
= n * log2(n)

Conclusion The truth of the assertion for all positive n follows by the Induction Principle from the Basis Step and the Inductive Step.

The proofs that compsIS(n) <= $(n-1)^2$ and that compsHSBuild2(n) < 2*n are left as exercises. The latter fact shows that this way of building a heap executes in big-oh of N time.

Exercise 18.52 (harder) Rewrite the `merged` method to not use a header node or create any extra nodes.
Exercise 18.53** Revise the `merged` method on the precondition that `one`'s first Node contains the smallest data value. Run down the `one` list, inserting Nodes from `two`'s list wherever appropriate, then return the `one` list. Rewrite the `sorted` method to use this new `merged` method by calling it from two different places in the coding. Is this a faster implementation of the MergeSort logic?
Exercise 18.54** Prove that compsIS(n) <= $(n-1)^2$ for all positive values of n.
Exercise 18.55** Prove that compsHSBuild2(n) = 2*n - 2*log2(n+1) for all positive numbers that are 1 less than a power of 2.
Exercise 18.56** Write a recurrence relation for the number of comparisons required for binary search in an array and prove inductively a good upper limit on them.

18.9 External Sorting: File Merge Using A Priority Queue

Sometimes we have so much data to put in sorted order that we cannot fit it all into RAM. Say it is stored in a hard-disk file and there are 2 million records to sort (a **record** is the set of current values of instance variables of an object). Our problem is to produce a new hard-disk file (we will call it SORTED.DAT) that contains all of those records in ascending order of IDs.

The piles-of-files algorithm

It is desirable to do as much of the sorting as possible in RAM, so a reasonable first step is to read as many values as we can into RAM, sort them, and write them out to a file. Then we read some more and write those to another file, etc. For instance, if we can handle 10,000 records at a time in RAM, we could end up with our original 2 million records in 200 different files. Then we could merge them together into one large file as shown in the accompanying design block (`numFiles` is 200 for this example).

STRUCTURED NATURAL LANGUAGE DESIGN for merging 200 files

1. Read the first record from each file into the corresponding component of an array of `numFiles` Comparable objects. Call the array `item`.
2. Find the smallest of those array components. Say it is at index `k`.
3. Write `item[k]` to the SORTED.DAT file.
4. If file number `k` is now empty, delete its entry from the array, otherwise get another value from the corresponding file `k` to go in `item[k]`.
5. Repeat steps 2 through 4 until done.

The total execution time for this algorithm is what is required for:
(a) the internal sorting, 200 groups of 10,000, plus
(b) reading and writing 2 million records 2 times each, and also
(c) making 199 comparisons 2 million times for a total of 398 million comparisons.
That last calculation generalizes to `N * (N / 10000)` for N records, so overall this is a big-oh of N-squared algorithm.

Object design

This design block is far too specific. We should not be making a commitment to unsorted arrays of values at this point in the design, or to a specific kind of file. To start with, we should just require some kind of file object that can provide the next object in a sequential file when asked or accept a new object to add to the end of the file. A reasonable object design is the following:

```
public class ObjectFile  // for generic files of objects
{   // Open the file of that name for output; open a temporary file if the name is "".
    public ObjectFile (String name)                 {  }
    // Add ob to the end of the file.
    public void writeObject (Object ob)             {  }
    // Change over to providing input, not output.
    public void openForInput()                      {  }
    // Retrieve the next available object in the file.
    public Object readObject()          { return null; }
    // Switch back to providing output, not input.
    public void openForOutput()                     {  }
    // Tell whether the input file has no more values.
    public boolean isEmpty()           { return false; }
}
```

We can then leave the details of how this is done in terms of the Sun standard library for the coding of this class's methods. You might want to look into Sun's ObjectInputStream class and serialization for a good way to implement this class.

To sort the data 10,000 units at a time (or whatever is appropriate), we basically need an object that provides two services: It can read 10,000 or so values from a given ObjectFile that has been opened for input; and it can write those 10,000 or so values in order to a given ObjectFile that has been opened for output. It could be as follows:

```
public class ObjectFileSorter  // for sorters of ObjectFiles
{   // Create the object capable of holding max values.
    public ObjectFileSorter (int max)                  {  }
    // Read max values from the given file, except stop reading at the end of the file.
    public void readManyFromFile (ObjectFile file)     {  }
    // Write all values you have to the given file in increasing order using compareTo.
    public void writeManyToFile (ObjectFile file)      {  }
}
```

The upper part of Listing 18.14 (see next page) shows the structure of the object that converts one very large unsorted file to a very large sorted file. For the constructor, you supply the names of the unsorted file and the sorted file. Both are initially open for output (according to the specifications for the ObjectFile class), so you have to open the unsorted file for input.

The `makeSortedFiles` method in the middle part of Listing 18.14 creates an ObjectFileSorter object from a numeric parameter that tells how many objects you want to have in the sorter at one time. Then it repeatedly reads 10,000 (or whatever) values from the unsorted file, writes them in sorted order to a temporary output file, and adds the temporary file along with its first value to a data structure named `itsData`.

For the process of merging 200 files (or however many we have), we need a kind of data structure that can store pairs consisting of a file plus the next available value from that file. When we get a value from this object, we want to receive the pair with the smallest value using the `compareTo` method. A priority queue class is acceptable for this purpose.

A priority queue can also do most of the task required of an ObjectFileSorter. That is, `readManyFromFile` can repeatedly read a data value and add it to a priority queue. And `writeManyToFile` can repeatedly call `removeMin` for that priority queue and write the result to the file.

Using only four files

A prime difficulty with this piles-of-files algorithm is that many systems do not allow you to keep more than a dozen or so files open at any one time. So we next present a method that only uses four files.

Stage 1 (for the `makeSortedFiles` method) Write the sorted groups of 10,000 (or whatever) alternately to just 2 files. That is, the first group goes into file `one`, the second into file `two`, the third into file `one` again, the fourth into file `two` again, the fifth into file `one`, the sixth into file `two`, etc. So for the example of 2 million data values, you end up with 100 groups of 10,000 in each of the 2 files.

Listing 18.14 File merge with piles of files

```
import ObjectFile;
import ObjectFileSorter;
import PriQue;

public class ManyFilesMerger
{
   private ObjectFile itsInFile;  // the original unsorted input
   private ObjectFile itsOutFile; // the final sorted output
   private HeapPriQue itsData = new HeapPriQue();


   public ManyFilesMerger (String inf, String outf)
   {  itsInFile = new ObjectFile (inf);   // for output      //1
      itsInFile.openForInput();        // but we need input  //2
      itsOutFile = new ObjectFile (outf); // for output      //3
   } //============================================

   /** Step 1:  Make many files, each sorted. */

   public void makeSortedFiles (int maxToSort)
   {  ObjectFileSorter sorter = new ObjectFileSorter (maxToSort);
      while ( ! itsInFile.isEmpty())                         //5
      {  ObjectFile tempFile = new ObjectFile ("");          //6
         sorter.readManyFromFile (itsInFile);                //7
         sorter.writeManyToFile (tempFile);                  //8
         tempFile.openForInput();                            //9
         itsData.add (new Pair (tempFile.readObject(),tempFile));
      }                                                      //11
   } //============================================

   /** Step 2:  Merge the many files into just one sorted file.*/

   public void mergeFiles()
   {  while ( ! itsData.isEmpty())                           //12
      {  Pair p = (Pair) itsData.removeMin();                //13
         itsOutFile.writeObject (p.itsData);                 //14
         if (! p.itsFile.isEmpty())                          //15
         {  p.itsData = p.itsFile.readObject();              //16
            itsData.add (p);                                 //17
         }                                                   //18
      }                                                      //19
   } //============================================

   private static class Pair implements Comparable
   {  public Object itsData;
      public final ObjectFile itsFile;

      public Pair (Object data, ObjectFile file)
      {  itsData = data;                                     //20
         itsFile = file;                                     //21
      }

      public int compareTo (Object ob)
      {  return ((Comparable) this.itsData).compareTo        //22
                     (((Pair) ob).itsData);                  //23
      }
   } //============================================
}
```

It will be difficult to make use of this data unless you can tell where one group of 10,000 ends and the next begins. So you need a special object value, different from any other, that you can use to mark the boundary between groups. Call this value `itsSentinel`. You will also find it convenient to have exactly the same number of groups in each of the two files; so if the last group goes in file `one`, just write `itsSentinel` again to `two`. That makes it an empty group of sorted values. The implementation for this part of the algorithm is in the upper and middle parts of Listing 18.15 (see next page).

Stage 2 (for the `mergeFiles` method) is the merging process:

1. Open files `one` and `two` for input and create two additional files `out1` and `out2` for output.
2. Combine the first 10,000 from file `one` with the first 10,000 from file `two` and write the resulting sorted group of 20,000 to file `out1`. This requires less than 20,000 comparisons using the standard merging logic.
3. Combine the second 10,000 from file `one` with the second 10,000 from file `two` and write the resulting sorted group of 20,000 to file `out2`. Again, you only need less than 20,000 comparisons.
4. Repeat steps 2 and 3 alternately until files `one` and `two` are empty. Now you have 50 groups of 20,000 in each of the two files `out1` and `out2`.
5. Open files `one` and `two` for output and files `out1` and `out2` for input.
6. Combine the first 20,000 from file `out1` with the first 20,000 from file `out2`; write those 40,000 to `one`.
7. Combine the second 20,000 from file `out1` with the second 20,000 from file `out2`; write them to file `two`.
8. Repeat steps 6 and 7 alternately until files `out1` and `out2` are empty. Now you have 25 groups of 40,000 in each of the files `one` and `two`.

Surely you can see where this is going. After the two passes through the data described above, you have six more passes to get it down to one completely sorted file of 2 million in just one file. You then write all of its values to the output file (except for the sentinel value at the end, of course). Note that you will occasionally get a group left over that has no group to be merged with (when you have an odd number of groups), in which case you just copy it into the appropriate file. The bottom part of Listing 18.15 contains this logic, except that the key merging logic is left as an exercise.

The total execution time for this algorithm is what is required for:
(a) the internal sorting, 200 groups of 10,000, plus
(b) reading and writing 2 million records 9 times each, and also
(c) making 2 million comparisons 8 times each for a total of 16 million comparisons.
The 9 and the 8 in this analysis are log2 (200) + 1 and log2(200). This generalizes to `N * log2(N / 10000)` for N records, so overall this is a big-oh of N * log(N) algorithm. But in real-life situations, it is slower than the piles-of-files method described first, since reading and writing disk records is excruciatingly slow.

Exercise 18.57 The ManyFilesMerger object in Listing 18.14 expects that a client class will call first the `makeSortedFiles` method and, directly after that, the `mergeFiles` method. What happens if a client class calls `mergeFiles` first?
Exercise 18.58 (harder) Still referring to the situation in the preceding exercise, what happens if a client class calls `makeSortedFiles` twice in a row without calling `mergeFiles`?
Exercise 18.59* Modify Listing 18.15 so that no client class can call `mergeFiles` before it calls `makeSortedFiles` and no client class can call either of those methods twice in a row. Hint: Add a boolean instance variable that tells whether `makeSortedFiles` has been called without an immediately following call of `mergeFiles`; use it appropriately.
Exercise 18.60*** Write the recursive coding for `mergeToOneFile` in Listing 18.15.

Listing 18.15 File merge with just four files

```
import ObjectFile;
import ObjectFileSorter;
import PriQue;

public class FourFilesMerger
{
   private ObjectFile itsInFile;  // the original unsorted input
   private ObjectFile itsOutFile; // the final sorted output
   private ObjectFile one, two;   // two scratch files
   private Object itsSentinel; // sentinel value to mark the end


   public FourFilesMerger (String inf, String outf, Object sent)
   {  itsInFile = new ObjectFile (inf);   // for output     //1
      itsInFile.openForInput();       // but we need input  //2
      itsOutFile = new ObjectFile (outf); // for output     //3
      itsSentinel = sent;                                   //4
   } //===============================================


   public void makeSortedFiles (int maxToSort)
   {  ObjectFileSorter sorter = new ObjectFileSorter (maxToSort);
      one = new ObjectFile ("");          // for output     //6
      two = new ObjectFile ("");          // for output     //7
      while ( ! itsInFile.isEmpty())                        //8
      {  sorter.readManyFromFile (itsInFile);               //9
         sorter.writeManyToFile (one);                      //10
         one.writeObject (itsSentinel);                     //11
         if ( ! itsInFile.isEmpty())                        //12
         {  sorter.readManyFromFile (itsInFile);            //13
            sorter.writeManyToFile (two);                   //14
         }                                                  //15
         two.writeObject (itsSentinel);                     //16
      }                                                     //17
   } //===============================================


   public void mergeFiles()
   {  if (one != null && ! one.isEmpty())                   //18
      {  one = mergeToOneFile (one, two,                    //19
               new ObjectFile (""), new ObjectFile (""));   //20
         Object data = one.readObject();                    //21
         while ( ! one.isEmpty())                           //22
         {  itsOutFile.writeObject (data);                  //23
            data = one.readObject();                        //24
         }                                                  //25
      }                                                     //26
   } //===============================================


   /** Return a file containing all the values in increasing
    *  order, plus a sentinel at the end. */

   private ObjectFile mergeToOneFile (ObjectFile in1,
           ObjectFile in2, ObjectFile out1, ObjectFile out2)
   {  return null;  // left as an exercise
   } //===============================================
}
```

18.10 Review Of Chapter Eighteen

- Stacks, queues, and priority queues are data structures that allow you to add an element, remove a particular element, see if they are empty, or see what you would get if you removed an element. The particular element you get depends on the structure: A **priority queue** gives you the element of highest priority (in case of a tie, the element that has been there longest). The **PriQue** interface has the methods `isEmpty()`, `add(ob)`, `removeMin()`, and `peekMin()`.
- A **java.util.Comparator** object has a method `compare(Object, Object)` that returns an int with the same meaning as for the standard `compareTo` method. This kind of functor object gives you full flexibility in choosing how things are prioritized.
- A priority queue implementation is **stable** if, in case of ties in priority, the element that has been in the data structure the longest is always removed first.
- This chapter presented several implementations of PriQue: ArrayOutPriQue and NodeOutPriQue (based on the InsertionSort), ArrayInPriQue and NodeInPriQue (based on the SelectionSort), NodeGroupPriQue (when there are very few different priority levels), TreePriQue (based on the QuickSort or its equivalent TreeSort), and HeapPriQue (based on the HeapSort logic).
- A **HeapSort** arranges the elements in an array so each element is greater than or equal to its two "children", then repeatedly moves the first element out and readjusts the heap structure. The children of index k are index 2*k+1 and 2*k+2.

Answers to Selected Exercises

18.1
```
public static void transfer (PriQue one, PriQue two)
{   while ( ! one.isEmpty())
        two.add (one.removeMin());
}
```

18.2 The third iteration combines the trees with 9 and 10 in their roots to obtain a tree with 19 in its root. This 19-node would be inserted between 16 and 22. Its left subtree would be a frequency of 9 with "gun" in its right subtree and the existing "peace"-4-"love" subtree as its left subtree.

18.3
```
public void combineHuffmanly (TreeNode par)  // in the TreeNode class
{   par.itsRight = this.itsLeft;
    this.itsLeft = par;
    this.itsData = new Integer (((Integer) this.itsData).intValue() + ((Integer) par.itsData).intValue());
}
```

18.7
```
Use the following Comparator class:
public class ByPosition implements java.util.Comparator
{   public int compare (Object one, Object two)
    {   RectangularShape a = (RectangularShape) one;
        RectangularShape b = (RectangularShape) two;
        return a.getY() != b.getY()  ?  a.getY() - b.getY()  :  b.getX() - a.getX();
    }
}
```

18.10
```
Object[] toDiscard = itsItem;
itsItem = new Object [itsItem.length * 2];
for (int k = 0;  k < itsSize;  k++)
    itsItem[k] = toDiscard[k];
```

18.11
```
public void add (Object ob)
if (itsSize == itsItem.length)
{ }  // same as for the preceding exercise
itsItem[itsSize] = ob;
itsSize++;
```

18.12
```
Replace the next-to-last statement in the removeMin method by the following:
for (int k = best;  k < itsSize;  k++)
    itsItem[k] = itsItem[k + 1];
```

18.17
```
public Object peekMin()
{   if (isEmpty())
        throw new IllegalStateException ("priority Q is empty");
    return itsFirst.itsData;
}
```

18.18
```
Replace lines 12-14 of NodeInPriQue's removeMin method by the following two:
best.itsData = best.itsNext.itsData;
best.itsNext = best.itsNext.itsData;
```

18.19 In the add method in Listing 18.5, replace the while-loop by the following four lines:

```
while (p.itsNext != null && itsTest.compare (ob, p.itsData) > 0)  // note change in the second condition
    p = p.itsNext;
if (p.itsNext != null && itsTest.compare (ob, p.itsData) == 0)
    return;
```

18.20
```
public int size()
{   int count = 0;
    for (Node p = itsFirst;  p.itsNext != null;  p = p.itsNext)
        count++;
    return count;
}
```

18.21
```
public String toString()
{   String valueToReturn = "";
    for (Node p = itsFirst;  p.itsNext != null;  p = p.itsNext)
        valueToReturn += '\t' + p.itsData.toString();
    return valueToReturn;
}
```

18.22
```
public void removeAbove (Object ob)   // in NodeOutPriQue
{   while (itsFirst.itsNext != null && itsTest.compare (itsFirst.itsData, ob) < 0)
        itsFirst = itsFirst.itsNext;
}
```

18.23
```
public void removeAbove (Object ob)   // in NodeInPriQue
{   while (itsFirst.itsNext != null && itsTest.compare (itsFirst.itsData, ob) < 0)  // check the first one
        itsFirst = itsFirst.itsNext;
    if (itsFirst.itsNext != null)
    {   Node p = itsFirst;
        while (p.itsNext.itsNext != null)   // so the next node has data to be compared
            if (itsTest.compare (p.itsNext.itsData, ob) < 0)
                p.itsNext = p.itsNext.itsNext;
            else
                p = p.itsNext;
    }
}
```

18.24
```
public void add (Object ob)  // revision for NodeInPriQue
{   if ( ! isEmpty() && itsTest.compare (ob, itsFirst.itsData) >= 0)
        itsFirst.itsNext = new Node (ob, itsFirst.itsNext);
    else
        itsFirst = new Node (ob, itsFirst);
}
```

18.35 In peekMin write: return ((QueueADT) itsFirst.itsData).peekFront();
In removeMin write: Object valueToReturn = ((QueueADT) itsFirst.itsData).dequeue();
Also in removeMin write: if (((QueueADT) itsFirst.itsData).isEmpty())
In add replace "p.itsData." by "((QueueADT) p.itsData)." in all three places.

18.38
```
public Object worstData()
{   return itsRight == ET  ?  itsData  :  itsRight.worstData();
}
```

18.39
```
public void printData()
{   if (isEmpty())
        return;
    itsLeft.printData();
    System.out.println (itsData.toString());
    itsRight.printData();
}
```

18.43 Just before the while in line 7, define: int parent = (k - 1) / 2;
Replace the phrase (k - 1) / 2 by parent in three places: lines 8, 9, and 10.
Just before the end of the loop body at line 9, update: parent = (k - 1) / 2.

18.44 The add method swaps 3 and 7, leaves 4 where it is, then swaps 6 and 3. Result: {7, 6, 4, 3}.
The siftDown method swaps 7 and 3, then swaps 3 with 6. Result: {6, 3, 4, 7}. The next call of siftDown swaps 6 and 4. The next call of siftDown swaps 4 and 3. Final result: {3, 4, 6, 7}.

18.45 Currently, itsSize >= 2 requires only 1 test, but itsSize < 2 requires 2 tests (at both lines 18 and 20).
The swap would require 2 tests except when itsSize==1, which happens less often than itsSize >= 2.

18.46 3 when size is 3: 2 for add, 1 for siftDown. 25 when size is 7: 2*1+4*2 for the two levels of add, then (4*4-2)+1 for siftDown.

18.52 Change the last statement to be return result. Then insert the following at the beginning:

```
Node result = ((Comparable) one.itsData).compareTo (two.itsData) < 0  ?  one  :  two;
if (result == one)
    one = one.itsNext;
else
    two = two.itsNext;
```

18.57 Nothing happens if mergeFiles is called first, since itsData is empty until after makeSortedFiles executes.

18.58 In effect, nothing happens if makeSortedFiles is called twice. True, on the second time another new ObjectFileSorter object is created. But itsInfile is empty (caused by the preceding call of this method), so the method terminates immediately and the newly-created object is garbage collected.