Chapter 6 The Heap

We now turn to the *heap* abstract data type, its implementation, and its applications. Because the heap is more sophisticated than the vector, we must do some preliminary work before defining the ADT. First we present our current UML class diagram.

6.1 Our Current Model

Our current software model appears below as Figure 6.1. In this chapter we will add classes `Heap` and `MinHeap` to our evolving framework. Modeling and implementing classes `MaxHeap` and `Container` are left as exercises.

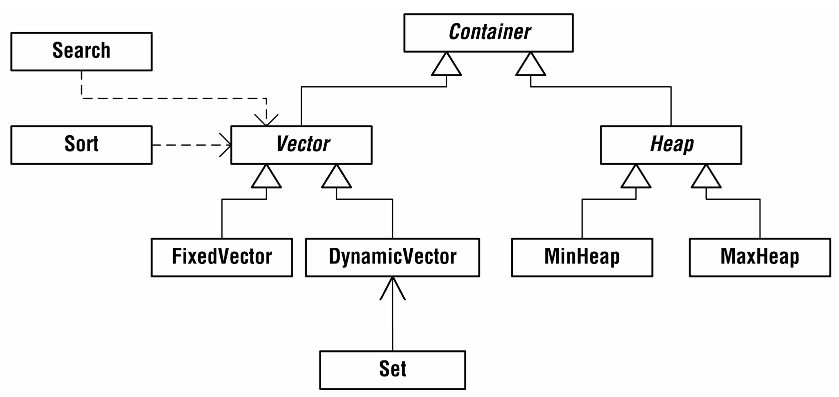


Fig. 6.1. Our current model, which incorporates abstract class `Heap` and its concrete subclasses, and which generalizes classes `Vector` and `Heap` by way of class `Container`

6.2 Heap Preliminaries

A *binary tree* is a collection of elements called *nodes*. The collection is either empty or it consists of a node called the *root*, along with two binary trees called the left and right *subtrees*. The roots of the left and right subtrees are *children* of the root; the root is the *parent* of these children. An *edge* connects each node of a binary tree to each of its children. A node with no children is called a *leaf*. A node having at least one child is called an *internal node*. An example binary tree is shown below in Figure 6.2.

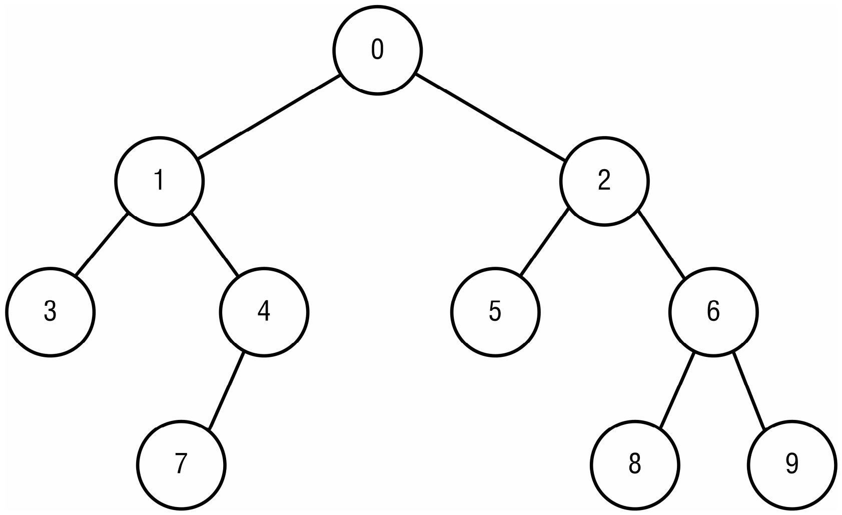


Fig. 6.2. An example binary tree. Node 0 is the root of the tree. Nodes 1 and 2 are the root's children. Node 1 is the root of node 0's left subtree. Node 4 is the root of node 1's right subtree. Node 4's right subtree is the empty tree. Nodes 3, 5, 7, 8, and 9 are leaves.

The *depth* of node *n* in a binary tree is the number of edges that must be traversed when traveling from the root of the tree to *n*. The *height* of the tree is one more than the depth of the deepest node. All nodes with depth *d* are said to be at *level d* of the tree.

A binary tree is called *complete* if it was built by filling its levels from left to right, starting at the root. The tree of Figure 6.1 is not complete. An example complete binary tree is shown below in Figure 6.3.

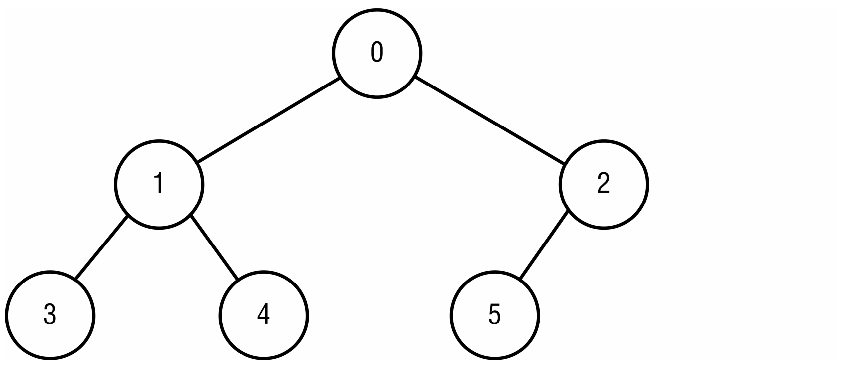


Fig. 6.3. A complete binary tree. In a complete binary tree of height *h*, all levels are full except perhaps level *h* - 1.

6.3 The Heap ADT

A heap is a *partially ordered* complete binary tree. To say that a heap is partially ordered is to say that there is some relationship between the value of a node and the values of its children. In a *min-heap*, the value of a node is less than or equal to the values of its children. In a *max-heap*, the value of a node is greater than or equal to the values of its children. Consequently, the smallest (largest) value in a min-heap (max-heap) is at the heap's root. The heap ADT appears below.

```
heap: a partially ordered complete binary tree

operations:

            clear() - Make the collection empty.
    insert(element) - Add a new element to the collection.
          isEmpty() - Is the collection empty?
             peek() - Get the value at the root.
           remove() - Remove the value at the root.
             size() - How many elements are in the collection?
```

6.4 Heap Implementation

Even though a tree is obviously not a linear structure, the heap ADT is typically implemented as an array container because the complete binary tree lends itself to an elegant and straightforward array implementation.

To see why, consider the complete binary tree from Figure 6.2. Observe that when the nodes are numbered as shown, the number of the left child of node *n* is 2*n* + 1, the number of the right child of node *n* is 2*n* + 2, and the number of the parent of node *n* is (*n* - 1) / 2, where / indicates integer division. If we let a node's number be its location in an array, then we can organize a complete binary tree in a linear fashion and use these simple mathematical relationships between a node's location and the locations of its parent and children to treat the data as if they are in fact organized as a binary tree.

6.4.1 Abstract Class `Heap`

The min-heap and max-heap differ only with respect to their `percolate` and `sift` helper operations. Thus we implement the remaining operations as methods of abstract class `Heap` and subclass `Heap` with `MinHeap` and `MaxHeap`. The code for class `Heap` appears below.

```
public abstract class Heap
{
    protected static final int DEFAULT_CAPACITY = 100;
    protected Comparable[] data;  // We need Comparable data
                                  // because a heap is partially
    protected int numItems;       // ordered.

    public Heap()
    {
        data = new Comparable[DEFAULT_CAPACITY];
    }

    public Heap(int initCapacity)
    {
        if (initCapacity <= 0)
            data = new Comparable[DEFAULT_CAPACITY];
        else
            data = new Comparable[initCapacity];
    }

    public void clear()
    {
        for (int j = 0; j < numItems; j++)
            data[j] = null;
        numItems = 0;
    }
```

```
/*
 precondition: If the array is too large and is consequently
               wasting space, this method contracts it. If the
               array is not wasting space, then the method
               simply returns.
postcondition: The capacity of the array is equal to the size
               of the collection.
*/

public void contract()
{
    if (size() == data.length)
        return;
    Comparable[] newData = new Comparable[size()];
    for (int j = 0; j < size(); j++)
        newData[j] = data[j];
    data = newData;
}

// This method is a helper for insert() and varies between the
// min-heap and the max-heap. It will be explained along with
// the implementation of class MinHeap.

protected abstract void percolate();

/*
 precondition: N/A
postcondition: The given element has been added to the
               collection and percolate() has restored the
               partial ordering, if necessary.
*/

public void insert(Comparable element)
{
    if (isFull())
    {
        Comparable[] newData =
            new Comparable[data.length * 2];
        for (int j = 0; j < numItems; j++)
            newData[j] = data[j];
        data = newData;
    }
    data[numItems++] = element;  // Always append the new
    percolate();                 // element, then percolate to
}                                // restore the heap ordering.

public boolean isEmpty()
{
    return numItems == 0;
}

// Is the node at the given location a leaf?

protected boolean isLeaf(int pos)
{
    return 2 * pos + 1 >= size();
}

// Where is the left child of the node at location 'pos'?

protected int leftChild(int pos)
{
    if (pos < 0)
        return -1;
    return 2 * pos + 1;
}
```

```
// Where is the right child of the node at location 'pos'?

protected int rightChild(int pos)
{
    if (pos < 0)
        return -1;
    return 2 * pos + 2;
}

// Where is the parent of the node at location 'pos'?

protected int parent(int pos)
{
    if (pos < 1)
        return -1;
    return (pos - 1) / 2;
}

/*
 precondition: One cannot peek at the root of an empty tree.
               If the collection is empty, then this method
               returns null.
postcondition: The data at the top of the heap has been
               returned.
*/

protected Comparable peek()
{
    if (isEmpty())
        return null;
    return data[0];
}

// This method is a helper for remove() and varies between the
// min-heap and the max-heap. It will be explained along with
// the implementation of class MinHeap.

protected abstract void sift();

/*
 precondition: One cannot remove the root of an empty tree. If
               the collection is empty, then this method
               returns null.
postcondition: The element at the top of the heap has been
               removed from the collection and returned.
*/

protected Comparable remove()
{
    if (isEmpty())
        return null;
    swap(data, 0, size() - 1);  // Swap the root item and the
                                // last item,...
    Comparable root = data[size() - 1];  // then save the root
                                         // item.
    data[--numItems] = null; // Delete the root from the
                             // array.
    if (size() != 0)         // If the heap is non-empty, sift
        sift();              // to restore the heap ordering.
    return root;
}

public int size()
{
    return numItems;
}
```

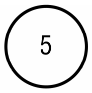
```
    protected void swap(Comparable[] arr, int first, int second)
    {
        Comparable temp = arr[first];
        arr[first] = arr[second];
        arr[second] = temp;
    }
}
```

6.4.2 Class `MinHeap`

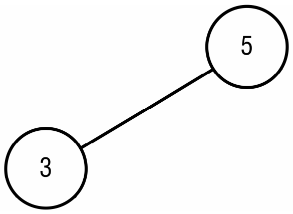
We now implement class `MinHeap`. The min-heap's `percolate` and `sift` helper operations, which restore the min-heap's partial ordering after insertion or removal, respectively, differ from the analogous max-heap operations because the two kinds of heap differ in their orderings.

Insertion into a Min-Heap

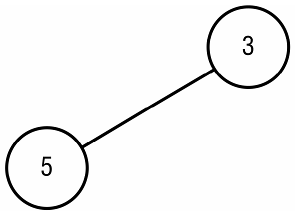
Let us insert into a min-heap the values 5, 3, 9, 7, 1, and 6, in that order. The insertion of 5 gives the heap shown below.



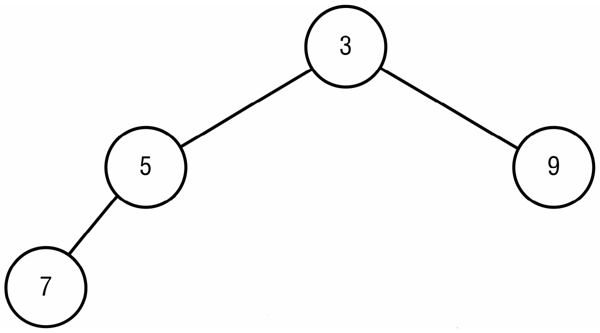
The next available location is the root's left child, and so the value 3 is placed there.



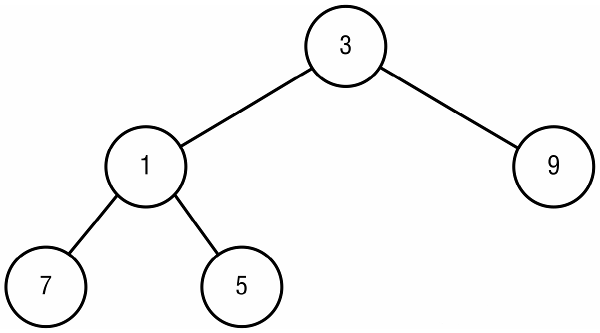
Because the resulting tree is no longer a min-heap, the `percolate` operation is called upon to restore the heap ordering. Starting with the newly placed element, `percolate` swaps the current element with its parent until the root is reached or until the heap ordering has been restored. In this case, the 3 is swapped with the value at the root, yielding the heap shown below.



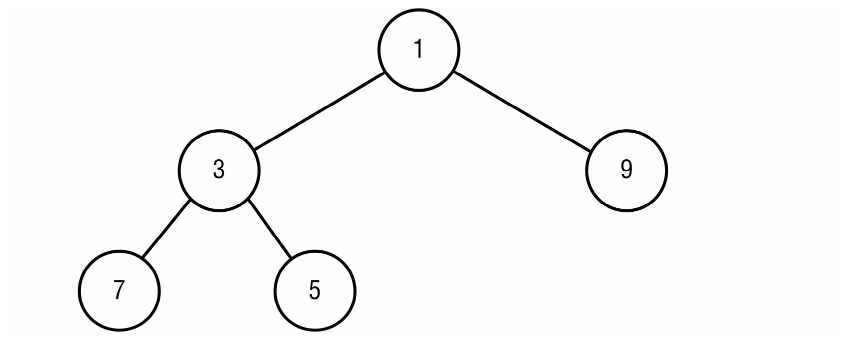
Now the 9 is placed at the next available location, as the root's right child. This does not destroy the min-heap ordering, nor does the insertion of the 7 as 5's left child. Our heap is now in the state shown below.



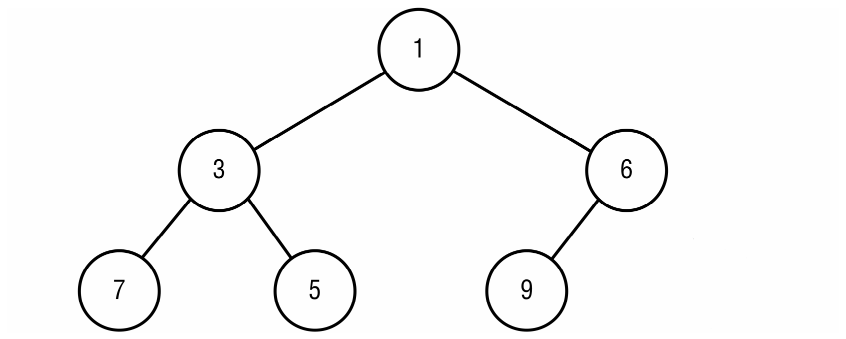
Placing the 1 as 5's right child again destroys the partial ordering, and so `percolate` has work to do. The 1 is first swapped with the 5, but this is not sufficient to restore the ordering because the value of the root is not less than or equal to the value of its left child, as shown below.



Consequently, `percolate` performs a second swap, placing 1 at the root and 3 as the root's left child. The resulting heap appears below.



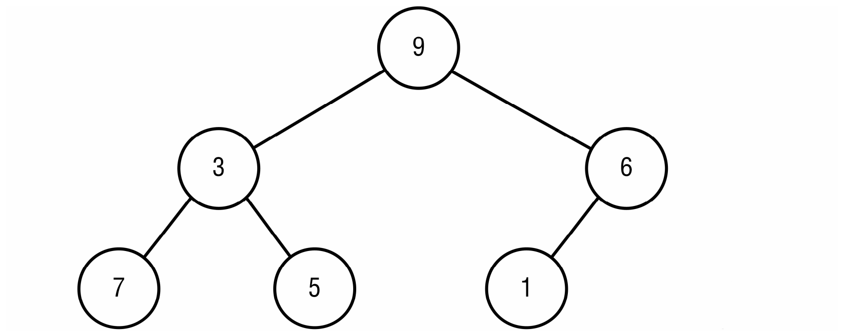
The 6 is now placed at 9's left child, which gives an incorrect relationship between the 6 and the 9. Operation `percolate` remedies the problem by swapping the 6 and the 9, yielding the final heap shown below.



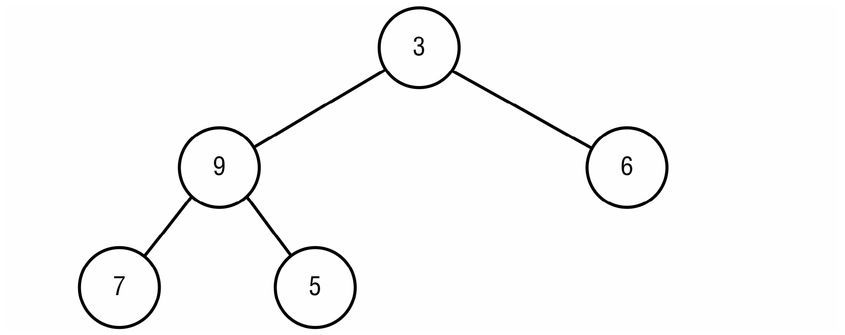
Removing from a Min-Heap

The heap ADT's `remove` operation extracts the value at the root. In an array implementation this is done by swapping the value at the root with the last value, after which the `sift` operation is called upon to restore the partial ordering, if necessary. Let us remove the value at the top of the min-heap at which we arrived in the last section.

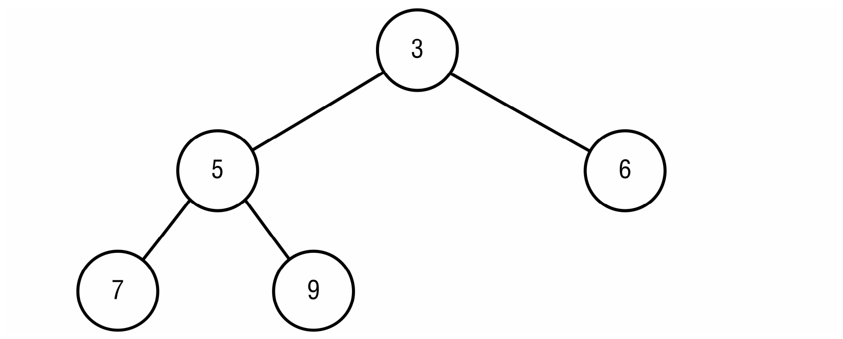
The value at the root, 1, is first swapped with the last value, 9, yielding the tree shown below.



After the `remove` operation has finished, the 1 will no longer be an element of the collection, and so 1's relationship with its parent does not concern us. But placing 9 at the root has destroyed the ordering. Operation `sift` must restore the ordering, starting at the root. Swapping the 9 and the 6 would give us the correct relationship between the root and its right child, but an incorrect ordering between the root and its left child, 3, would remain. The solution to this dilemma is to swap not the 9 and the 6 but the 9 and the 3, which gives the tree shown below.



Clearly, an additional swap is required to restore the ordering, and again the misplaced value must be swapped with the smaller of its two children. Swapping the 9 and the 5 yields the new min-heap.



Now the collection's size can be decremented and its minimum value returned.

The Implementation

The code for class `MinHeap` is shown below.

```
public class MinHeap extends Heap
{
    public MinHeap() {}

    public MinHeap(int initCapacity)
    {
        super(initCapacity);
    }

    // The peek() operation for the min-heap has traditionally
    // been called peekMin(). Class MinHeap offers public method
    // peekMin() by wrapping a call to class Heap's protected
    // peek() method.

    public Comparable peekMin()
    {
        return peek();
    }

    /*
     precondition: N/A
    postcondition: The min-heap's partial ordering has been
                   restored.
    */

    protected void percolate()
    {
        int pos = size() - 1;
        while (pos != 0 &&
               data[pos].compareTo(data[parent(pos)]) < 0)
        {
            swap(data, pos, parent(pos));
            pos = parent(pos);
        }
    }
    /*
     precondition: N/A
    postcondition: The min-heap's partial ordering has been
                   restored.
    */

    protected void sift()
    {
        int pos = 0,
            j,
            rc;
        while (! isLeaf(pos))  // While we have not reached a
        {                      // leaf...
            j = leftChild(pos);
            rc = rightChild(pos);

            // Select the smaller of the left child and the right
            // child.

            if (rc < size() && data[j].compareTo(data[rc]) > 0)
                j = rc;

            // If the item at index 'pos' is less than or equal to
            // the smaller of its children, then we are finished.

            if (data[pos].compareTo(data[j]) <= 0)
                return;

            // Otherwise, swap the item at 'pos' with the smaller
            // of its children.

            swap(data, pos, j);
            pos = j;             // Move to index j and continue.
        }
    }

    public Comparable removeMin()
    {
        return remove();
    }
}
```

6.5 Heap Applications

Two important applications of the heap ADT are *priority queuing* and sorting. Here we shall discuss the use of a max-heap as a *priority queue* and show how to use a min-heap for sorting.

6.5.1 Priority Queuing

priority queue: a collection of elements that is organized by importance or priority

An everyday example of priority queuing is the triage used in hospital emergency rooms, on battlefields, or at disaster sites. A battlefield doctor, for example, may choose which soldiers to treat next based not on their arrival times but on their need for immediate medical care. The unfortunate soldier with a head wound will likely be treated immediately upon his arrival, whereas the soldier with a wounded hand or foot will be treated only after the critically wounded.

The most common computer-science application of priority queuing is in scheduling programs for execution by a *multi-tasking operating system*.

multi-tasking operating system: an operating system that allows two or more programs to share the attention of a single CPU; such an operating system supports two or more simultaneous users and/or allows a single user to execute two or more programs at once

A multi-tasking operating system may use a max-heap for scheduling. The next program to be executed is the one with the highest priority; that program is represented by the entry at the root of the max-heap.

6.5.2 The Heap Sort Algorithm and Its Time Analysis

The sorting algorithm that employs a min-heap is called the heap sort. The heap sort's implementation, shown below, is straightforward. First a min-heap is built using the elements of the input vector. Then repeated calls to `removeMin` are used to replace the elements of the vector in sorted order.

```
public class Sort
{
    public static void heapSort(Vector vec)
    {
        int j,
            n = vec.size();
        MinHeap heap = new MinHeap(n);
        for (j = 0; j < n; j++)
            heap.insert((Comparable)vec.elementAt(j));
        for (j = 0; j < n; j++)
            vec.replace(j, heap.removeMin());
    }
}
```

The fundamental operations of the heap sort are heap operations `insert` and `removeMin`, and so we must determine the growth rates for these operations in order to determine the growth rate of the heap sort algorithm. The growth rates for the `insert` and `removeMin` operations depend on the height of the tree, for in the worst case an insertion or removal will result in an element being percolated or sifted, respectively, across all levels of the tree. The height of a complete binary tree with *n* nodes is $\lceil \lg(n) \rceil$ if *n* is not a power of 2 and is $\lg(n) + 1$ if *n* is a power of 2, as the reader should verify. Thus the `insert` and `removeMin` operations are in $O(\lg(n))$ in the worst case. ($\lceil\ \rceil$ denotes the *least-integer*, or *ceiling*, function, which is defined below.)

$\lceil x \rceil$ = the smallest integer greater than or equal to x

Building a min-heap from a vector of size *n* requires *n* insertions, and so in the worst case the building of the heap requires approximately $\lg(n!)$ operations and is therefore in $O(n \lg(n))$. Placing the *n* elements back in the vector requires *n* removals; thus the sorting also requires approximately $\lg(n!)$ operations in the worst case, putting it in $O(n \lg(n))$. Consequently, the heap sort performs approximately $2 \lg(n!)$ operations in the worst case and is therefore in $O(n \lg(n))$ in the worst case. This represents a significant improvement over the sorting algorithms presented in Chapter 5.

Exercises

1. Containers `Vector` and `Heap` both have field `numItems` and methods `clear`, `isEmpty`, and `size`. Put the field and the three methods in an abstract class called `Container`, and change `Vector` and `Heap` so that they inherit from `Container`. Be sure to override method `clear` in classes `Vector` and `Heap`.
2. Implement class `MaxHeap`.
3. Add attributes and operations to the UML class diagram shown in Figure 6.1.
4. Recall that class `Vector`'s array is of type `Object`, despite the fact that the methods of classes `Search` and `Sort` require their `Vector`s to contain `Comparable` elements. Class `Heap`'s array, on the other hand, is of type `Comparable`. Why should this be so?
5. Why is the heap ADT a better choice for priority queuing than the vector ADT?
6. What are the heap sort's best-case and average-case running times?
7. Show the max-heap that results from insertion of the following values, in the order given: 8, 5, 12, 4, 3, 2, 9, 7, 6.
8. Show the heap that results from deletion of the maximum value from the max-heap built for Exercise 7.
9. Design and code a battlefield triage simulation. Use an instance of `java.util.Random` to furnish your simulation with pseudorandom arrival of wounded, assignment of priority, and required time of treatment. Your simulation should allow the user to decide how many doctors are available.