Chapter 13

Priority Queues

Concepts:
- ▷ Priority queues
- ▷ Heaps
- ▷ Skew heaps
- ▷ Sorting with heaps
- ▷ Simulation

"Exactly!" said Mr. Wonka.
"I decided to invite five children
to the factory, and the one I liked best
at the end of the day
would be the winner!"
—Roald Dahl

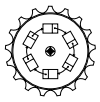
SOMETIMES A RESTRICTED INTERFACE IS A FEATURE. The *priority queue*, like an ordered structure, appears to keep its data in order. Unlike an ordered structure, however, the priority queue allows the user only to access its smallest element. The priority queue is also similar to the `Linear` structure: values are added to the structure, and they later may be inspected or removed. Unlike their `Linear` counterpart, however, once a value is added to the priority queue it may only be removed if it is the minimum value.[1] It is precisely this restricted interface to the priority queue that allows many of its implementations to run quickly.

Priority queues are used to schedule processes in an operating system, to schedule future events in a simulation, and to generally rank choices that are generated out of order.

Think triage.

13.1 The Interface

Because we will see many contrasting implementations of the priority queue structure, we describe it as abstractly as possible in Java—with an interface:



`PriorityQueue`

```
public interface PriorityQueue<E extends Comparable<E>>
{
    public E getFirst();
    // pre: !isEmpty()
    // post: returns the minimum value in priority queue

    public E remove();
```

[1] We will consider priority queues whose elements are ranked in ascending order. It is, of course, possible to maintain these queues in descending order with only a few modifications.

```
    // pre: !isEmpty()
    // post: returns and removes minimum value from queue

    public void add(E value);
    // pre: value is non-null comparable
    // post: value is added to priority queue

    public boolean isEmpty();
    // post: returns true iff no elements are in queue

    public int size();
    // post: returns number of elements within queue

    public void clear();
    // post: removes all elements from queue
}
```

Because they must be kept in order, the elements of a `PriorityQueue` are `Comparable`. In this interface the smallest values are found near the front of the queue and will be removed soonest.[2] The `add` operation is used to insert a new value into the queue. At any time a reference to the minimum value can be obtained with the `getFirst` method and is removed with `remove`. The remaining methods are similar to those we have seen before.

Notice that the `PriorityQueue` does not extend any of the interfaces we have seen previously. First, as a matter of convenience, `PriorityQueue` methods consume `Comparable` parameters and return `Comparable` values. Most structures we have encountered manipulate unconstrained generic `Objects`. Though similar, the `PriorityQueue` is not a `Queue`. There is, for example, no `dequeue` method. Though this might be remedied, it is clear that the `PriorityQueue` need not act like a first-in, first-out structure. At any time, the value about to be removed is the current minimum value. This value might have been the first value inserted, or it might have just recently "cut in line" before larger values. Still, the priority queue is just as general as the stack and queue since, with a little work, one can associate with inserted values a priority that forces any `Linear` behavior in a `PriorityQueue`. Finally, since the `PriorityQueue` has no `elements` method, it may not be traversed and, therefore, cannot be a `Collection`.

Exercise 13.1 *An alternative definition of a* `PriorityQueue` *might not take and return* `Comparable` *values. Instead, the constructor for a* `PriorityQueue` *could be made to take a* `Comparator`*. Recall that the* `compare` *method of the* `Comparator` *class needn't take a* `Comparable` *value. Consider this alternative definition. Will the code be simpler? When would we expect errors to be detected?*

---

[2] If explicit priorities are to be associated with values, the user may insert a `ComparableAssociation` whose key value is a `Comparable` such as an `Integer`. In this case, the associated value—the data element—need not be `Comparable`.

The simplicity of the abstract priority queue makes its implementation relatively straightforward. In this chapter we will consider three implementations: one based on use of an `OrderedStructure` and two based on a novel structure called a *heap*. First, we consider an example that emphasizes the simplicity of our interface.

13.2 Example: Improving the Huffman Code

Huffman

In the Huffman example from Section 12.8 we kept track of a pool of trees. At each iteration of the tree-merging phase of the algorithm, the two lightest-weight trees were removed from the pool and merged. There, we used an `OrderedStructure` to maintain the collection of trees:

```
OrderedList<huffmanTree> trees = new OrderedList<huffmanTree>();
// merge trees in pairs until one remains
Iterator ti = trees.iterator();
while (trees.size() > 1)
{
    // construct a new iterator
    ti = trees.iterator();
    // grab two smallest values
    huffmanTree smallest = (huffmanTree)ti.next();
    huffmanTree small = (huffmanTree)ti.next();
    // remove them
    trees.remove(smallest);
    trees.remove(small);
    // add bigger tree containing both
    trees.add(new huffmanTree(smallest,small));
}
// print only tree in list
ti  = trees.iterator();
Assert.condition(ti.hasNext(),"Huffman tree exists.");
huffmanTree encoding = (huffmanTree)ti.next();
```

To remove the two smallest objects from the `OrderedStructure`, we must construct an `Iterator` and indirectly remove the first two elements we encounter. This code can be greatly simplified by storing the trees in a `PriorityQueue`. We then remove the two minimum values:

Huffman2

```
PriorityQueue<huffmanTree> trees = new PriorityVector<huffmanTree>();
// merge trees in pairs until one remains
while (trees.size() > 1)
{
    // grab two smallest values
    huffmanTree smallest = (huffmanTree)trees.remove();
    huffmanTree small = (huffmanTree)trees.remove();
    // add bigger tree containing both
    trees.add(new huffmanTree(smallest,small));
```

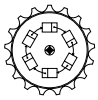
```
}
huffmanTree encoding = trees.remove();
```

After the merging is complete, access to the final result is also improved.

A number of interesting algorithms must have access to the minimum of a collection of values, and yet do not require the collection to be sorted. The extra energy required by an `OrderedVector` to keep all the values in order may, in fact, be excessive for some purposes.

13.3 A Vector-Based Implementation

Perhaps the simplest implementation of a `PriorityQueue` is to keep all the values in ascending order in a `Vector`. Of course, the constructor is responsible for initialization:



PriorityVector

```
protected Vector<E> data;

public PriorityVector()
// post: constructs a new priority queue
{
    data = new Vector<E>();
}
```

From the standpoint of adding values to the structure, the priority queue is very similar to the implementation of the `OrderedVector` structure. In fact, the implementations of the `add` method and the "helper" method `indexOf` are similar to those described in Section 11.2.2. Still, values of a priority queue are removed in a manner that differs from that seen in the `OrderedVector`. They are not removed by value. Instead, `getFirst` and the parameterless `remove` operate on the `Vector` element that is smallest (leftmost). The implementation of these routines is straightforward:

```
public E getFirst()
// pre: !isEmpty()
// post: returns the minimum value in the priority queue
{
    return data.get(0);
}

public E remove()
// pre: !isEmpty()
// post: removes and returns minimum value in priority queue
{
    return data.remove(0);
}
```

The `getFirst` operation takes constant time. The `remove` operation caches and removes the first value of the `Vector` with a linear-time complexity. This can-

not be easily avoided since the cost is inherent in the way we use the `Vector` (though see Problem 13.8).

It is interesting to see the evolution of the various types encountered in the discussion of the `PriorityVector`. Although the `Vector` took an entire chapter to investigate, the abstract notion of a vector seems to be a relatively natural structure here. Abstraction has allowed us to avoid considering the minute details of the implementation. For example, we assume that `Vectors` automatically extend themselves. The abstract notion of an `OrderedVector`, on the other hand, appears to be insufficient to directly support the specification of the `PriorityVector`. The reason is that the `OrderedVector` does not support `Vector` operations like the index-based `get(i)` and `remove(i)`. These could, of course, be added to the `OrderedVector` interface, but an appeal for symmetry might then suggest implementation of the method `add(i)`. This would be a poor decision since it would then allow the user to insert elements out of order.

Principle 21 *Avoid unnaturally extending a natural interface.*

Designers of data structures spend considerable time weighing these design trade-offs. While it is tempting to make the most versatile structures support a wide variety of extensions, it surrenders the interface distinctions between structures that often allow for novel, efficient, and safe implementations.

Exercise 13.2 *Although the* `OrderedVector` *class does not directly support the* `PriorityQueue` *interface, it nonetheless can be used in a protected manner. Implement the* `PriorityVector` *using a* `protected OrderedVector`*? What are the advantages and disadvantages?*

In Section 13.4 we discuss a rich class of structures that allow us to maintain a loose ordering among elements. It turns out that even a loose ordering is sufficient to implement priority queues.

13.4 A Heap Implementation

In actuality, it is not necessary to develop a complete ranking of the elements of the priority queue in order to support the necessary operations. It is only necessary to be able to quickly identify the *minimum* value and to maintain a relatively loose ordering of the remaining values. This realization is the motivation for a structure called a *heap*.

Definition 13.1 *A* heap *is a binary tree whose root references the minimum value and whose subtrees are, themselves, heaps.*

An alternate definition is also sometimes useful.

Definition 13.2 *A* heap *is a binary tree whose values are in ascending order on every path from root to leaf.*

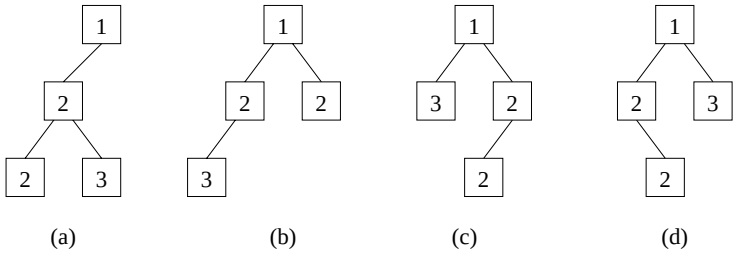


Figure 13.1 Four heaps containing the same values. Note that there is no ordering among siblings. Only heap (b) is complete.

We will draw our heaps in the manner shown in Figure 13.1, with the minimum value on the top and the possibly larger values below. Notice that each of the four heaps contains the same values but has a different structure. Clearly, there is a great deal of freedom in the way that the heap can be oriented—for example, exchanging subtrees does not violate the heap property (heaps (c) and (d) are mirror images of each other). While not every tree with these four values is a heap, many are (see Problems 13.17 and 13.18). This flexibility reduces the *friction* associated with constructing and maintaining a valid heap and, therefore, a valid priority queue. When friction is reduced, we have the potential for increasing the speed of some operations.

Principle 22 *Seek structures with reduced friction.*

This is completely obvious.

We will say that a heap is a *complete heap* if the binary tree holding the values of the heap is complete. Any set of n values may be stored in a complete heap. (To see this we need only sort the values into ascending order and place them in level order in a complete binary tree. Since the values were inserted in ascending order, every child is at least as great as its parent.) The abstract notion of a complete heap forms the basis for the first of two heap implementations of a priority queue.

13.4.1 Vector-Based Heaps

As we saw in Section 12.9 when we considered the implementation of Ahnentafel structures, any complete binary tree (and therefore any complete heap) may be stored compactly in a vector. The method involves traversing the tree in level order and mapping the values to successive slots of the vector. When we are finished with this construction, we observe the following (see Figure 13.2):

1. The root of the tree is stored in location 0. If non-`null`, this location references the minimum value of the heap.

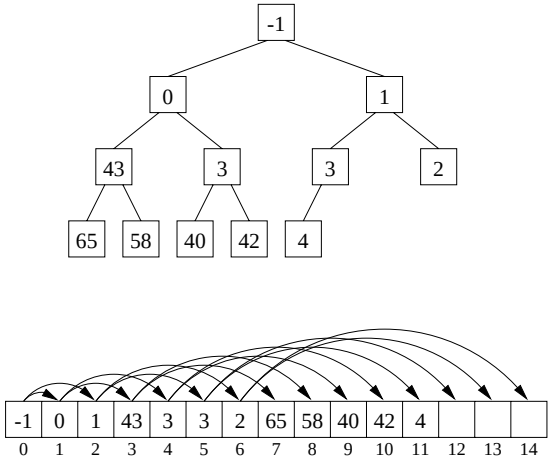
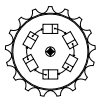


Figure 13.2 An abstract heap (top) and its vector representation. Arrows from parent to child are not physically part of the vector, but are indices computed by the heap's `left` and `right` methods.

---

2. The left child of a value stored in location i is found at location $2i + 1$.
3. The right child of a value stored in location i may be found at the location following the left child, location $2(i + 1) = (2i + 1) + 1$.
4. The parent of a value found in location i can be found at location $\lfloor \frac{i-1}{2} \rfloor$. Since division of integers in Java-like languages throws away the remainder for positive numbers, this expression is written `(i-1)/2`.

These relations may, of course, be encoded as functions. In Figure 13.2 we see the mapping of a heap to a vector, with tree relations indicated by arrows in the vector. Notice that while the vector is not maintained in ascending order, any path from the root to a leaf encounters values in ascending order. If the vector is larger than necessary, slots not associated with tree nodes can maintain a `null` reference. With this mapping in mind, we consider the constructor and `static` methods:



VectorHeap

```
protected Vector<E> data;  // the data, kept in heap order

public VectorHeap()
// post: constructs a new priority queue
{
    data = new Vector<E>();
}
```

```
public VectorHeap(Vector<E> v)
// post: constructs a new priority queue from an unordered vector
{
    int i;
    data = new Vector<E>(v.size()); // we know ultimate size
    for (i = 0; i < v.size(); i++)
    {   // add elements to heap
        add(v.get(i));
    }
}

protected static int parent(int i)
// pre: 0 <= i < size
// post: returns parent of node at location i
{
    return (i-1)/2;
}

protected static int left(int i)
// pre: 0 <= i < size
// post: returns index of left child of node at location i
{
    return 2*i+1;
}

protected static int right(int i)
// pre: 0 <= i < size
// post: returns index of right child of node at location i
{
    return 2*(i+1);
}
```

The functions `parent`, `left`, and `right` are declared `static` to indicate that they do not actually have to be called on any instance of a heap. Instead, their values are functions of their parameters only.

Principle 23 *Declare object-independent functions* `static`.

Now consider the addition of a value to a complete heap. We know that the heap is currently complete. Ideally, after the addition of the value the heap will remain complete but will contain one extra value. This realization forces us to commit to inserting a value in a way that ultimately produces a correctly structured heap. Since the first free element of the `Vector` will hold a value, we optimistically insert the new value in that location (see Figure 13.3). If, considering the path from the leaf to the root, the value is in the wrong location, then it must be "percolated upward" to the correct entry. We begin by comparing and, if necessary, exchanging the new value and its parent. If the values along the path are still incorrectly ordered, it must be because of the new value, and we continue to percolate the value upward until either the new value is the

root or it is greater than or equal to its current parent. The only values possibly exchanged in this operation are those appearing along the unique path from the insertion point. Since locations that change only become smaller, the integrity of other paths in the tree is maintained.

The code associated with percolating a value upward is contained in the function `percolateUp`. This function takes an index of a value that is possibly out of place and pushes the value upward toward the root until it reaches the correct location. While the routine takes an index as a parameter, the parameter passed is usually the index of the rightmost leaf of the bottom level.

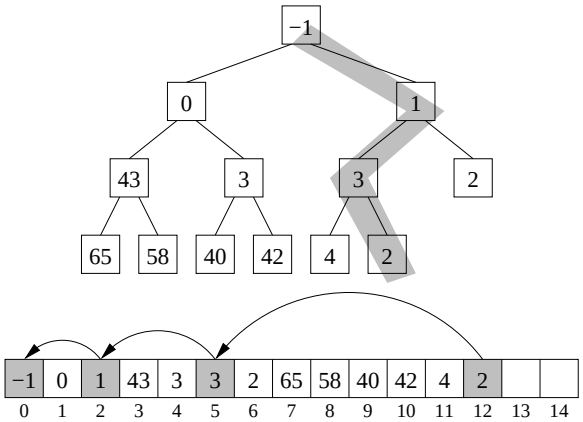
```
protected void percolateUp(int leaf)
// pre: 0 <= leaf < size
// post: moves node at index leaf up to appropriate position
{
    int parent = parent(leaf);
    E value = data.get(leaf);
    while (leaf > 0 &&
      (value.compareTo(data.get(parent)) < 0))
    {
        data.set(leaf,data.get(parent));
        leaf = parent;
        parent = parent(leaf);
    }
    data.set(leaf,value);
}
```

Adding a value to the priority queue is then only a matter of appending it to the end of the vector (the location of the newly added leaf) and percolating the value upward until it finds the correct location.

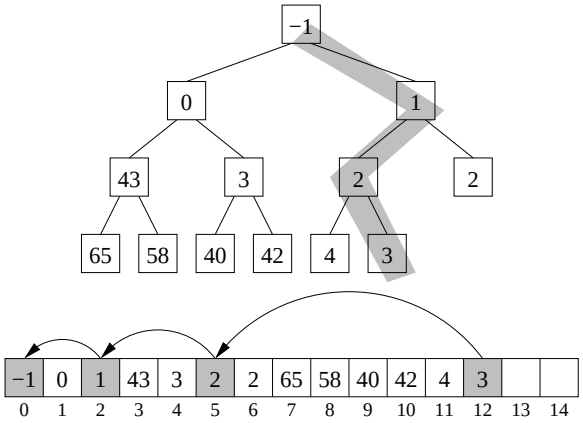
```
public void add(E value)
// pre: value is non-null comparable
// post: value is added to priority queue
{
    data.add(value);
    percolateUp(data.size()-1);
}
```

Let us consider how long it takes to accomplish the addition of a value to the heap. Remember that the tree that we are working with is an n-node complete binary tree, so its height is $\lfloor \log_2 n \rfloor$. Each step of the `percolateUp` routine takes constant time and pushes the new value up one level. Of course, it may be positioned correctly the first time, but the worst-case behavior of inserting the new value into the tree consumes $O(\log n)$ time. This performance is considerably better than the linear behavior of the `PriorityVector` implementation described in Section 13.3. What is the best time? It is constant when the value added is large compared to the values found on the path from the new leaf to the root.

What is the expected time? Be careful!



(a) Before



(b) After

Figure 13.3 The addition of a value (2) to a vector-based heap. (a) The value is inserted into a free location known to be part of the result structure. (b) The value is percolated up to the correct location on the unique path to the root.

Next, we consider the removal of the minimum value (see Figures 13.4 and 13.5). It is located at the root of the heap, in the first slot of the vector. The removal of this value leaves an empty location at the top of the heap. Ultimately, when the operation is complete, the freed location will be the rightmost leaf of the bottom level, the last element of the underlying vector. Again, our approach is first to construct a tree that is the correct shape, but potentially not a heap, and then perform transformations on the tree that both maintain its shape and bring the structure closer to being a heap. Thus, when the minimum value is removed, the rightmost leaf on the bottom level is removed and re-placed at the root of the tree (Figure 13.4a and b). At this point, the tree is the correct shape, but it may not be a heap because the root of the tree is potentially too large. Since the subtrees remain heaps, we need to ensure the root of the tree is the minimum value contained in the tree. We first find the minimum child and compare this value with the root (Figure 13.5a). If the root value is no greater, the minimum value is at the root and the entire structure is a heap. If the root is larger, then it is exchanged with the true minimum—the smallest child—pushing the large value downward. At this point, the root of the tree has the correct value. All but one of the subtrees are unchanged, and the shape of the tree remains correct. All that has happened is that a large value has been pushed down to where it may violate the heap property in a subtree. We then perform any further exchanges recursively, with the value sinking into smaller subtrees (Figure 13.5b), possibly becoming a leaf. Since any single value is a heap, the recursion must stop by the time the newly inserted value becomes a leaf.

We're "heaping in shape."

Here is the code associated with the pushing down of the root:

```
protected void pushDownRoot(int root)
// pre: 0 <= root < size
// post: moves node at index root down
//   to appropriate position in subtree
{
    int heapSize = data.size();
    E value = data.get(root);
    while (root < heapSize) {
        int childpos = left(root);
        if (childpos < heapSize)
        {
            if ((right(root) < heapSize) &&
              ((data.get(childpos+1)).compareTo
               (data.get(childpos)) < 0))
            {
                childpos++;
            }
            // Assert: childpos indexes smaller of two children
            if ((data.get(childpos)).compareTo
                (value) < 0)
            {
                data.set(root,data.get(childpos));
```

```
                root = childpos; // keep moving down
            } else { // found right location
                data.set(root,value);
                return;
            }
        } else { // at a leaf! insert and halt
            data.set(root,value);
            return;
        }
    }
}
```

The `remove` method simply involves returning the smallest value of the heap, but only after the rightmost element of the vector has been pushed downward.

```
public E remove()
// pre: !isEmpty()
// post: returns and removes minimum value from queue
{
    E minVal = getFirst();
    data.set(0,data.get(data.size()-1));
    data.setSize(data.size()-1);
    if (data.size() > 1) pushDownRoot(0);
    return minVal;
}
```

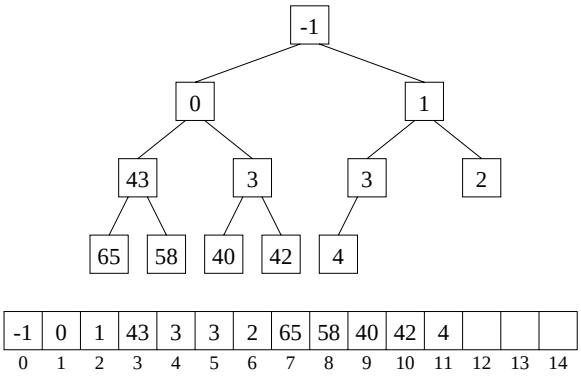
Each iteration in `pushDownRoot` pushes a large value down into a smaller heap on a path from the root to a leaf. Therefore, the performance of `remove` is $O(\log n)$, an improvement over the behavior of the `PriorityVector` implementation.

Since we have implemented all the required methods of the `PriorityQueue`, the `VectorHeap` implements the `PriorityQueue` and may be used wherever a priority queue is required.

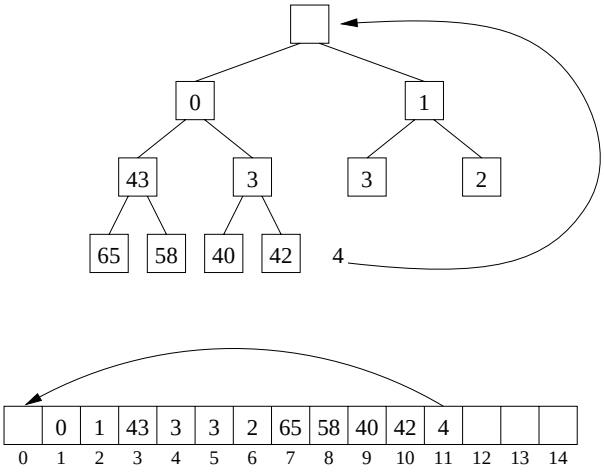
The advantages of the `VectorHeap` mechanism are that, because of the unique mapping of complete trees to the `Vector`, it is unnecessary to explicitly store the connections between elements. Even though we are able to get improved performance over the `PriorityVector`, we do not have to pay a space penalty. The complexity arises, instead, in the code necessary to support the insertion and removal of values.

13.4.2 Example: Heapsort

Any priority queue, of course, can be used as the underlying data structure for a sorting mechanism. When the values of a heap are stored in a `Vector`, an empty location is potentially made available when they are removed. This location could be used to store a removed value. As the heap shrinks, the values are stored in the newly vacated elements of the `Vector`. As the heap becomes empty, the `Vector` is completely filled with values in descending order.

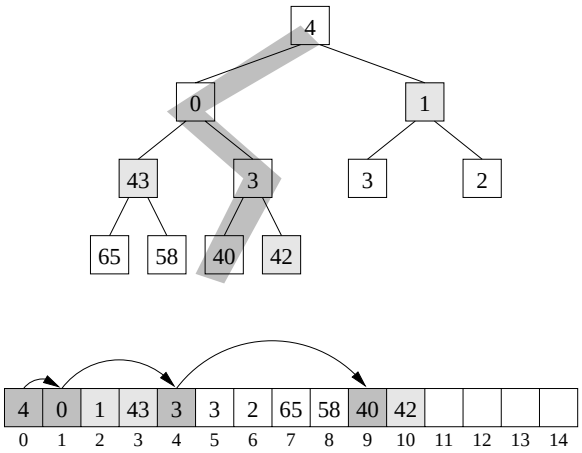


(a)

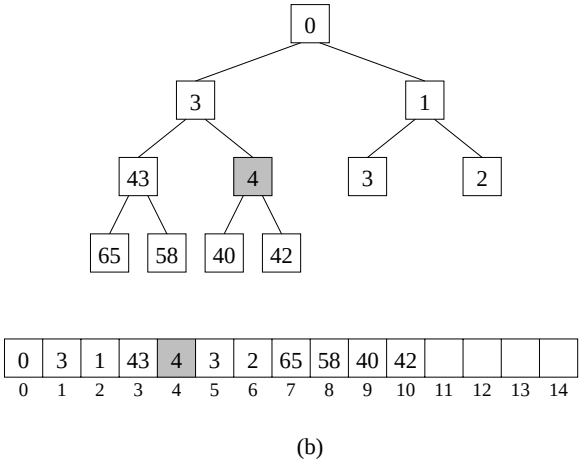


(b)

Figure 13.4 Removing a value from the heap shown in (a) involves moving the rightmost value of the vector to the top of the heap as in (b). Note that this value is likely to violate the heap property but that the subtrees will remain heaps.



(a)



(b)

Figure 13.5 Removing a value (continued). In (a) the newly inserted value at the root is pushed down along a shaded path following the smallest children (lightly shaded nodes are also considered in determining the path). In (b) the root value finds, over several iterations, the correct location along the path. Smaller values shift upward to make room for the new value.

Unfortunately, we cannot make assumptions about the structure of the values initially found in the `Vector`; we are, after all, sorting them. Since this approach depends on the values being placed in a heap, we must consider one more operation: a constructor that "heapifies" the data found in a `Vector` passed to it:

```
public VectorHeap(Vector<E> v)
// post: constructs a new priority queue from an unordered vector
{
    int i;
    data = new Vector<E>(v.size()); // we know ultimate size
    for (i = 0; i < v.size(); i++)
    {   // add elements to heap
        add(v.get(i));
    }
}
```

The process of constructing a heap from an unordered `Vector` obviously takes the time of n `add` operations, each of which is $O(\log n)$. The worst-case cost of "heapifying" is, therefore, $O(n \log n)$. (This can be improved—see Problem 13.10.)

Now, the remaining part of the heapsort—removing the minimum values and placing them in the newly freed locations—requires n `remove` operations. This phase also has worst-case complexity $O(n \log n)$. We have, therefore, another sorting algorithm with $O(n \log n)$ behavior and little space overhead.

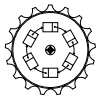
The feature of a heap that makes the sort so efficient is its short height. The values are always stored in as full a tree as possible and, as a result, we may place a logarithmic upper bound on the time it takes to insert and remove values. In Section 13.4.3 we investigate the use of unrestricted heaps to implement priority queues. These structures have *amortized* cost that is equivalent to heaps built atop vectors.

13.4.3 Skew Heaps

The performance of `Vector`-based heaps is directly dependent on the fact that these heaps are complete. Since complete heaps are a minority of all heaps, it is reasonable to ask if efficient priority queues might be constructed from unrestricted heaps. The answer is yes, if we relax the way we measure performance.

We consider, here, the implementation of heaps using dynamically structured binary trees. A direct cost of this decision is the increase in space. Whereas a `Vector` stores a single reference, the binary tree node keeps an additional three references. These three references allow us to implement noncomplete heaps, called *skew heaps*, in a space-efficient manner (see Problem 13.21). Here are the protected data and the constructor for this structure:

```
protected BinaryTree<E> root;
```



SkewHeap

```
protected final BinaryTree<E> EMPTY = new BinaryTree<E>();
protected int count;

public SkewHeap()
// post: creates an empty priority queue
{
    root = EMPTY;
    count = 0;
}
```

Notice that we keep track of the size of the heap locally, rather than asking the `BinaryTree` for its size. This is simply a matter of efficiency, but it requires us to maintain the value within the `add` and `remove` procedures. Once we commit to implementing heaps in this manner, we need to consider the implementation of each of the major operators.

The implementation of `getFirst` simply references the value stored at the root. Its implementation is relatively straightforward:

```
public E getFirst()
// pre: !isEmpty()
// post: returns the minimum value in priority queue
{
    return root.value();
}
```

As with all good things, this will eventually seem necessary.

Before we consider the implementation of the `add` and `remove` methods, we consider a (seemingly unnecessary) operation, `merge`. This method takes two heaps and merges them together. This is a *destructive* operation: the elements of the participating heaps are consumed in the construction of the result. Our approach will be to make `merge` a recursive method that considers several cases. First, if either of the two heaps participating in the merge is empty, then the result of the merge is the other heap. Otherwise, both heaps contain at least a value—assume that the minimum root is found in the left heap (if not, we can swap them). We know, then, that the result of the merge will be a reference to the root node of the left heap. To see how the right heap is merged into the left we consider two cases:

1. If the left heap has no left child, make the right heap the left child of the left heap (see Figure 13.6b).

2. Otherwise, exchange the left and right children of the left heap. Then merge (the newly made) left subheap of the left heap with the right heap (see Figure 13.6d).

Notice that if the left heap has one subheap, the right heap becomes the left subheap and the merging is finished. Here is the code for the `merge` method:

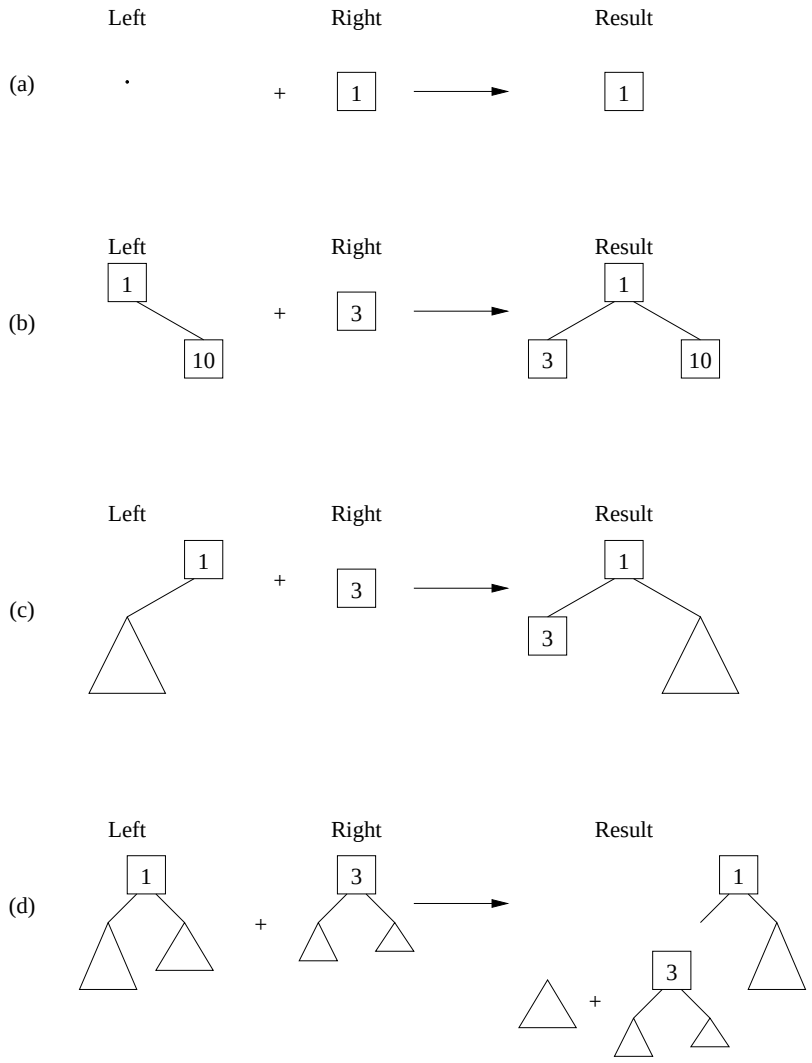


Figure 13.6 Different cases of the `merge` method for `SkewHeaps`. In (a) one of the heaps is empty. In (b) and (c) the right heap becomes the left child of the left heap. In (d) the right heap is merged into what was the right subheap.

```
protected static <E extends Comparable<E>>
    BinaryTree<E> merge(BinaryTree<E> left, BinaryTree<E> right)
{
    if (left.isEmpty()) return right;
    if (right.isEmpty()) return left;
    E leftVal = left.value();
    E rightVal = right.value();
    BinaryTree<E> result;
    if (rightVal.compareTo(leftVal) < 0)
    {
        result = merge(right,left);
    } else {
        result = left;
        // assertion left side is smaller than right
        // left is new root
        if (result.left().isEmpty())
        {
            result.setLeft(right);
        } else {
            BinaryTree<E> temp = result.right();
            result.setRight(result.left());
            result.setLeft(merge(temp,right));
        }
    }
    return result;
}
```

Once the `merge` method has been defined, we find that the process of adding a value or removing the minimum is relatively straightforward. To add a value, we construct a new `BinaryTree` containing the single value that is to be added. This is, in essence, a one-element heap. We then merge this heap with the existing heap, and the result is a new heap with the value added:

```
public void add(E value)
// pre: value is non-null comparable
// post: value is added to priority queue
{
    BinaryTree<E> smallTree = new BinaryTree<E>(value,EMPTY,EMPTY);
    root = merge(smallTree,root);
    count++;
}
```

To remove the minimum value from the heap we must extract and return the value at the root. To construct the smaller resulting heap we detach both subtrees from the root and merge them together. The result is a heap with all the values of the left and right subtrees, but not the root. This is precisely the result we require. Here is the code:

```
public E remove()
// pre: !isEmpty()
```

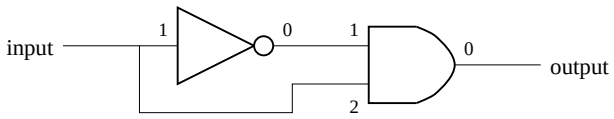


Figure 13.7 A circuit for detecting a rising logic level.

```
// post: returns and removes minimum value from queue
{
    E result = root.value();
    root = merge(root.left(),root.right());
    count--;
    return result;
}
```

The remaining priority queue methods for skew heaps are implemented in a relatively straightforward manner.

Because a skew heap has unconstrained topology (see Problem 13.16), it is possible to construct examples of skew heaps with degenerate behavior. For example, adding a new maximum value can take $O(n)$ time. For this reason we cannot put very impressive bounds on the performance of any individual operation. The skew heap, however, is an example of a self-organizing structure: inefficient operations spend some of their excess time making later operations run more quickly. If we are careful, time "charged against" early operations can be *amortized* or redistributed to later operations, which we hope will run very efficiently. This type of analysis can be used, for example, to demonstrate that $m > n$ skew heap operations applied to a heap of size n take no more than $O(m \log n)$ time. On average, then, each operation takes $O(\log n)$ time. For applications where it is expected that a significant number of requests of a heap will be made, this performance is appealing.

13.5 Example: Circuit Simulation

Consider the electronic digital circuit depicted in Figure 13.7. The two devices shown are *logic gates*. The wires between the gates propagate electrical signals. Typically a zero voltage is called *false* or *low*, while a potential of 3 volts or more is *true* or *high*.

The triangular gate, on the left, is an *inverter*. On its output (pin 0) it "inverts" the logic level found on the input (pin 1): false becomes true and true becomes false. The gate on the right is an *and-gate*. It generates a true on pin 0 exactly when both of its inputs (pins 1 and 2) are true.

The action of these gates is the result of a physical process, so the effect of the inputs on the output is delayed by a period of time called a *gate delay*. Gate delays depend on the complexity of the gate, the manufacturing process, and environmental factors. For our purposes, we'll assume the gate delay of the inverter is 0.2 nanosecond (ns) and the delay of the and-gate is 0.8 ns.

The question is, what output is generated when we toggle the input from low to high and back to low several times? To determine the answer we can build the circuit, or simulate it in software. For reasons that will become clear in a moment, simulation will be more useful.

The setup for our simulation will consist of a number of small classes. First, there are a number of components, including an `Inverter`; an `And`; an input, or `Source`; and a voltage sensor, or `Probe`. When constructed, gates are provided gate delay values, and `Sources` and `Probes` are given names. Each of these components has one or more pins to which wires can be connected. (As with real circuits, the outputs should connect only to inputs of other components!) Finally, the voltage level of a particular pin can be set to a particular level. As an example of the interface, we list the `public` methods for the `And` gate:

Circuit

```
class And extends Component
{
    public And(double delay)
    // pre: delay >= 0.0ns
    // post: constructs and gate with indicated gate delay

    public void set(double time, int pinNum, int level)
    // pre: pinNum = 1 or 2, level = 0/3
    // post: updates inputs and generates events on
    //       devices connected to output
}
```

Notice that there is a time associated with the `set` method. This helps us document when different events happen in the component. These events are simulated by a comparable `Event` class. This class describes a change in logic level on an input pin for some component. As the simulation progresses, `Events` are created and scheduled for simulation in the future. The ordering of `Events` is based on an event time. Here are the details:

```
class Event implements Comparable<Event>
{
    protected double time;    // time of event
    protected int level;      // voltage level
    protected Connection c;   // gate/pin

    public Event(Connection c, double t, int l)
    // pre: c is a valid pin on a gate
    // post: constructs event for time t to set pin to level l
    {
        this.c = c;
```

```
        time = t;
        level = l;
    }

    public void go()
    // post: informs target component of updated logic on pin
    {
        c.component().set(time,c.pin(),level);
    }

    public int compareTo(Event other)
    // pre: other is non-null
    // post: returns integer representing relation between values
    {
        Event that = (Event)other;
        if (this.time < that.time) return -1;
        else if (this.time == that.time) return 0;
        else return 1;
    }
}
```

The `Connection` mentioned here is simply a component's input pin.

Finally, to orchestrate the simulation, we use a priority queue to correctly simulate the order of events. The following method simulates a circuit by removing events from the priority queue and setting the logic level on the appropriate pins of the components. The method returns the time of the last event to help monitor the progress of the simulation.

```
public class Circuit
{
    static PriorityQueue<Event> eventQueue; // main event queue

    public static double simulate()
    // post: run simulation until event queue is empty;
    //       returns final clock time
    {
        double low = 0.0;       // voltage of low logic
        double high = 3.0;      // voltage of high logic
        double clock = 0.0;
        while (!eventQueue.isEmpty())
        {   // remove next event
            Event e = eventQueue.remove();
            // keep track of time
            clock = e.time;
            // simulate the event
            e.go();
        }
        System.out.println("-- circuit stable after "+clock+" ns --");
        return clock;
    }
```

```
}
```

As events are processed, the logic level on a component's pins are updated. If the inputs to a component change, new `Events` are scheduled one gate delay later for each component connected to the output pin. For `Sources` and `Probes`, we write a message to the output indicating the change in logic level. Clearly, when there are no more events in the priority queue, the simulated circuit is stable. If the user is interested, he or she can change the logic level of a `Source` and resume the simulation by running the `simulate` method again.

We are now equipped to simulate the circuit of Figure 13.7. The first portion of the following code sets up the circuit, while the second half simulates the effect of toggling the input several times:

```
public static void main(String[] args)
{
    int low = 0;    // voltage of low logic
    int high = 3;   // voltage of high logic
    eventQueue = new SkewHeap<Event>();
    double time;

    // set up circuit
    Inverter not = new Inverter(0.2);
    And and = new And(0.8);
    Probe output = new Probe("output");
    Source input = new Source("input",not.pin(1));

    input.connectTo(and.pin(2));
    not.connectTo(and.pin(1));
    and.connectTo(output.pin(1));

    // simulate circuit
    time = simulate();
    input.set(time+1.0,0,high); // first: set input high
    time = simulate();
    input.set(time+1.0,0,low);  // later: set input low
    time = simulate();
    input.set(time+1.0,0,high); // later: set input high
    time = simulate();
    input.set(time+1.0,0,low);  // later: set input low
    simulate();
}
```

When run, the following output is generated:

```
1.0 ns: output now 0 volts
-- circuit stable after 1.0 ns --
2.0 ns: input set to 3 volts
2.8 ns: output now 3 volts
3.0 ns: output now 0 volts
```

```
-- circuit stable after 3.0 ns --
4.0 ns: input set to 0 volts
-- circuit stable after 5.0 ns --
6.0 ns: input set to 3 volts
6.8 ns: output now 3 volts
7.0 ns: output now 0 volts
-- circuit stable after 7.0 ns --
8.0 ns: input set to 0 volts
-- circuit stable after 9.0 ns --
```

When the input is moved from low to high, a short spike is generated on the output. Moving the input to low again has no impact. The spike is generated by the *rising edge* of a signal, and its width is determined by the gate delay of the inverter. Because the spike is so short, it would have been difficult to detect it using real hardware.[3] Devices similar to this edge detector are important tools for detecting changing states in the circuits they monitor.

13.6 Conclusions

We have seen three implementations of priority queues: one based on a `Vector` that keeps its entries in order and two others based on heap implementations. The `Vector` implementation demonstrates how any ordered structure may be adapted to support the operations of a priority queue.

Heaps form successful implementations of priority queues because they relax the conditions on "keeping data in priority order." Instead of maintaining data in sorted order, heaps maintain a competition between values that becomes progressively more intense as the values reach the front of the queue. The cost of inserting and removing values from a heap can be made to be as low as $O(\log n)$.

If the constraint of keeping values in a `Vector` is too much (it may be impossible, for example, to allocate a single large chunk of memory), or if one wants to avoid the uneven cost of extending a `Vector`, a dynamic mechanism is useful. The `SkewHeap` is such a mechanism, keeping data in general heap form. Over a number of operations the skew heap performs as well as the traditional `Vector`-based implementation.

Self Check Problems

Solutions to these problems begin on page 449.

13.1 Is a `PriorityQueue` a `Queue`?

13.2 Is a `PriorityQueue` a `Linear` structure?

[3] This is a very short period of time. During the time the output is high, light travels just over 2 inches!

13.3 How do you interpret the weight of a Huffman tree? How do you interpret the depth of a node in the tree?

13.4 What is a min-heap?

13.5 `Vector`-based heaps have $O(\log n)$ behavior for insertion and removal of values. What structural feature of the underlying tree guarantees this?

13.6 Why is a `PriorityQueue` useful for managing simulations base on events?

Problems

Solutions to the odd-numbered problems begin on page 481.

13.1 Draw the state of a `HeapVector` after each of the values 3, 4, 7, 0, 2, 8, and 6 are added, in that order.

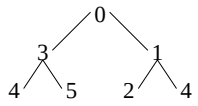
13.2 Consider the heap

0	2	1	3	7	4	6	8

a. What does this heap look like when drawn as a tree?

b. What does this heap look like (in array form) when a value is removed?

13.3 Below is a `SkewHeap`. What does it look like after a value is removed?



13.4 How might you use priorities to simulate a LIFO structure with a priority queue?

13.5 Is a `VectorHeap` a `Queue`? Is it an `OrderedStructure`?

13.6 How might you use priorities to simulate a FIFO structure with a priority queue?

13.7 Suppose a user built an object whose `compareTo` and `equals` methods were inconsistent. For example, values that were `equals` might also return a negative value for `compareTo`. What happens when these values are added to a `PriorityVector`? What happens when these values are added to a `SkewHeap`?

13.8 We have seen that the cost of removing a value from the `PriorityVector` takes linear time. If elements were stored in descending order, this could be reduced to constant time. Compare the ascending and descending implementations, discussing the circumstances that suggest the use of one implementation over the other.

13.9 What methods would have to be added to the `OrderedVector` class to make it possible to implement a `PriorityVector` using only a private `OrderedVector`?

13.10 Reconsider the "heapifying" constructor discussed in Section 13.4.2. Instead of adding n values to an initially empty heap (at a cost of $O(n \log n)$), suppose we do the following: Consider each interior node of the heap in order of decreasing array index. Think of this interior node as the root of a potential subheap. We know that its subtrees are valid heaps. Now, just push this node down into its (near-)heap. Show that the cost of performing this heapify operation is linear in the size of the `Vector`.

13.11 Design a more efficient version of `HeapVector` that keeps its values in order only when necessary: When values are added, they are appended to the end of the existing heap and a `nonHeap` flag is set to `true`. When values are removed, the `nonHeap` flag is checked and the `Vector` is heapified if necessary. What are the worst-case and best-case running times of the `add` and `remove` operations? (You may assume that you have access to the heapify of Problem 13.10.)

13.12 Consider the unordered data:

4	2	7	3	1	0	5	6

What does this `Vector` look like after it has been heapified?

13.13 Consider the in-place `Vector`-based heapsort.

a. A min-heap is particularly suited to sorting data in place into which order: ascending or descending?

b. What is the worst-case time complexity of this sort?

c. What is the best-case time complexity of this sort?

13.14 Suppose we are given access to a min-heap, but not the code that supports it. What changes to the *comparable data* might we make to force the min-heap to work like a max-heap?

13.15 Suppose we are to find the kth largest element of a heap of n values. Describe how we might accomplish this efficiently. What is the worst-case running time of your method? Notice that if the problem had said "set of n values," we would require a heapify operation like that found in Problem 13.10.

13.16 Demonstrate that any binary tree that has the heap property can be generated by inserting values into a skew heap in an appropriate order. (This realization is important to understanding why an amortized accounting scheme is necessary.)

13.17 Suppose you are given n distinct values to store in a full heap—a heap that is maintained in a full binary tree. Since there is no ordering between children in a heap, the left and right subheaps can be exchanged. How many equivalent heaps can be produced by only swapping children of a node?

13.18 Given n distinct values to be stored in a heap, how many heaps can store the values? (Difficult.)

13.19 What proportion of the binary trees holding n distinct values are heaps?

13.20 Suppose that n randomly selected (and uniformly distributed) numbers are inserted into a complete heap. Now, select another number and insert it into the heap. How many levels is the new number expected to rise?

13.21 The mapping strategy that takes a complete binary tree to a `vector` can actually be used to store general trees, albeit in a space-inefficient manner. The strategy is to allocate enough space to hold the lowest, rightmost leaf, and to maintain null references in nodes that are not currently being used. What is the worst-case `Vector` length needed to store an n-element binary tree?

13.22 Write an `equals` method for a `PriorityVector`. It returns `true` if each pair of corresponding elements removed from the structures would be equal. What is the complexity of the `equals` method? (Hint: You may not need to remove values.)

13.23 Write an `equals` method for a `HeapVector`. It returns `true` if each pair of corresponding elements removed from the structures would be equal. What is the complexity of the `equals` method? (Hint: You may need to remove values.)

13.24 Write an `equals` method for a `SkewHeap`. It returns `true` if each pair of corresponding elements removed from the structures would be equal. What is the complexity of the `equals` method? (Hint: You may need to remove values.)

13.25 Show that the implementation of the `PriorityVector` can be improved by not actually keeping the values in order. Instead, only maintain the minimum value at the left. Demonstrate the implementation of the `add` and `remove` methods.

13.26 Suppose you are a manufacturer of premium-quality videocassette recorders. Your XJ-6 recorder allows the "user" to "program" 4096 different future events to be recorded. Of course, as the time arrives for each event, your machine is responsible for turning on and recording a program.

a. What information is necessary to correctly record an event?

b. Design the data structure(s) needed to support the XJ-6.

13.7 Laboratory: Simulating Business

Objective. To determine if it is better to have single or multiple service lines.

Discussion. When we are waiting in a fast food line, or we are queued up at a bank, there are usually two different methods of managing customers:

1. Have a single line for people waiting for service. Every customer waits in a single line. When a teller becomes free, the customer at the head of the queue moves to the teller. If there are multiple tellers free, one is picked randomly.

2. Have multiple lines—one for each teller. When customers come in the door they attempt to pick the line that has the shortest wait. This usually involves standing in the line with the fewest customers. If there are multiple choices, the appropriate line is selected randomly.

It is not clear which of these two methods of queuing customers is most efficient. In the single-queue technique, tellers appear to be constantly busy and no customer is served before any customer that arrives later. In the multiple-queue technique, however, customers can take the responsibility of evaluating the queues themselves.

Note, by the way, that some industries (airlines, for example) have a mixture of both of these situations. First class customers enter in one line, while coach customers enter in another.

Procedure. In this lab, you are to construct a simulation of these two service mechanisms. For each simulation you should generate a sequence of customers that arrive at random intervals. These customers demand a small range of services, determined by a randomly selected service time. The simulation is driven by an event queue, whose elements are ranked by the event time. The type of event might be a customer arrival, a teller freeing up, etc.

For the single line simulation, have the customers all line up in a single queue. When a customer is needed, a single customer (if any) is removed from the customer queue, and the teller is scheduled to be free at a time that is determined by the service time. You must figure out how to deal with tellers that are idle—how do they wait until a customer arrives?

For the multiple line simulation, the customers line up at their arrival time, in one of the shortest teller queues. When a teller is free, it selects the next customer from its dedicated queue (if any). A single event queue is used to drive the simulation.

To compare the possibilities of these two simulations, it is useful to run the same random customers through both types of queues. Think carefully about how this might be accomplished.

Thought Questions. Consider the following questions as you complete the lab:

1. Run several simulations of both types of queues. Which queue strategy seems to process all the customers fastest?

2. Is their a difference between the average wait time for customers between the two techniques?

3. Suppose you simulated the ability to jump between lines in a multiple line simulation. When a line has two or more customers than another line, customers move from the end one line to another until the lines are fairly even. You see this behavior frequently at grocery stores. Does this change the type of underlying structure you use to keep customers in line?

4. Suppose lines were dedicated to serving customers of varying lengths of service times. Would this improve the average wait time for a customer?

Notes: