Chapter 10 The Queue

The *queue* is a *first-in-first-out* (*FIFO*) collection, like a supermarket checkout line. In fact, speakers of U.K. English use the word *queue* in place of the word *line*. The queue has several applications in operating systems and is also important in *computer simulations*.

10.1 Our Current Model

In this chapter we will add classes `Queue` and `DLNode` to our evolving software design. Class `LinearNode` from Chapter 7 will be renamed `SLNode`. The current diagram appears below in Figure 10.1.

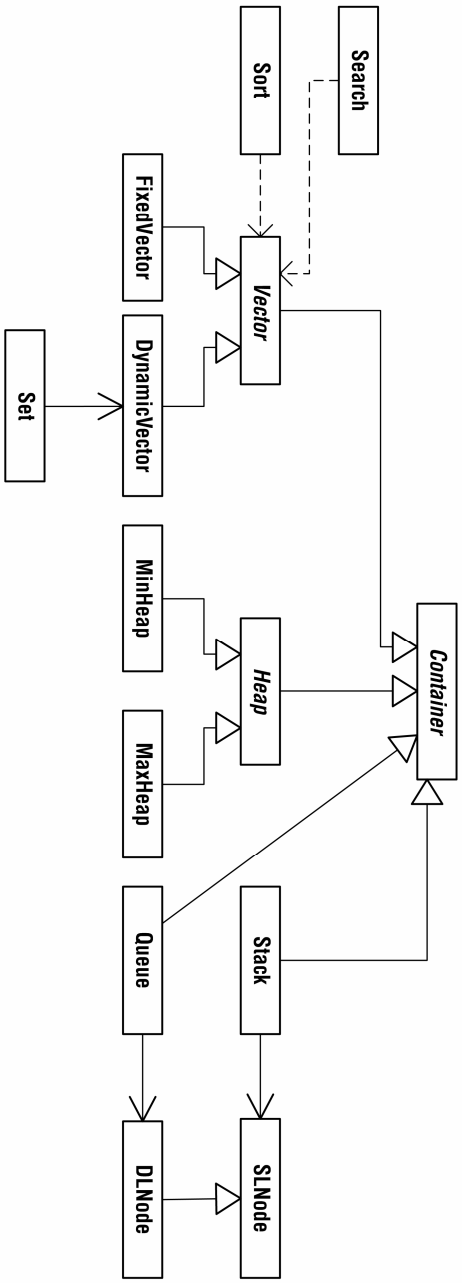

Fig. 10.1. Our current software design

10.2 The Queue ADT

The queue ADT appears below. Queue insertion and removal have traditionally been called `enqueue` and `dequeue`, respectively. We will call these operations `insertBack` and `removeFront`, however, for these names are more descriptive and will serve us better in Chapter 12.

```
queue: a linear collection of elements that can be accessed only
       at its ends; the current element of interest is called
       the front element; the last element added is called the
       back element

operations:

               clear() - Make the collection empty.
               front() - Get the front element of the collection.
    insertBack(element) - Make the given element the back element.
             isEmpty() - Is the collection empty?
          removeFront() - Remove the front element.
                size() - How many elements are in the collection?
```

10.3 Queue Implementation

The linked list used to implement class `Stack` in Chapter 7 had links going in only one direction, from the head node toward the tail node. Such a list is called *singly linked*. The queue can also be implemented using a singly linked list, but we will implement class `Queue` using a *doubly linked list*. Any node of a doubly linked list has two links; one link is the memory address of the node's predecessor, the other the memory address of the node's successor. Our doubly linked implementation of the queue will pave the way for the specialization presented in Chapter 12.

> We will henceforth refer to class `LinearNode` as `SLNode`.

10.3.1 Class DLNode

Class DLNode is a subclass of SLNode, for a doubly linked node is simply a singly linked node with an additional link. The code for class DLNode appears below.

```
public class DLNode extends SLNode
{
    DLNode prev;

    public DLNode(Object dat)
    {
        super(dat);
    }

    public DLNode(Object dat, DLNode pre, DLNode nxt)
    {
        super(dat, nxt);
        prev = pre;
    }
}
```

10.3.2 Class Queue

Since insertion into a queue takes place at the rear, our queue implementation will make use of a tail reference. The techniques used here are those presented in Chapter 7, except now we must deal with two links.

```
public class Queue extends Container
{
    DLNode head,
           tail;

    public void clear()
    {
        super.clear();
        head = tail = null;
    }

    public Object front()
    {
        if (isEmpty())
            return null;
        return head.data;
    }
```

```
    public void insertBack(Object element)
    {
        if (isEmpty())
        {
            head = tail = new DLNode(element);
            numItems++;
            return;
        }
        tail.next = new DLNode(element, tail, null);
        tail = (DLNode)tail.next;
        numItems++;
    }

    public Object removeFront()
    {
        Object temp = front();
        if (temp == null)
            return null;
        head = (DLNode)head.next;
        if (head != null)
            ((DLNode)head).prev = null;
        else
            tail = null;
        numItems--;
        return temp;
    }
}
```

10.4 Queue Applications

Queues are typically used to avoid loss in situations where congestion is possible, i.e., when requests for some resource may arrive faster than they can be serviced. A printer is an example of such a resource; you may have heard the term *print queue*. A print queue holds pending print jobs in the order of their arrival. Queues are also used in computer simulations.

computer simulation: a software application that seeks to imitate the salient features of a real-world system and to offer insight into the behavior of that system

An end-of-chapter exercise will ask you to design and code a computer simulation of an airport.

Exercises

1. Add attributes and operations to the class diagram presented at the beginning of the chapter.
2. How is a queue used by `java.io.BufferedWriter`?
3. Perform a time analysis of our queue implementation and of a dynamic-array queue implementation. Conclude that our implementation is more time-efficient. How do the two implementations compare with respect to their space requirements?
4. Design and code a class called `Airplane`, and code a simulation of a small single-runway airport. Airplanes waiting to take off join a queue on the ground. Planes waiting to land join a queue in the air. Only one plane can use the runway at a time. All planes in the air must land before any plane may take off.

   At each time step, your simulation should "decide" whether to generate a new plane. If a new plane is generated, your simulation must "decide" whether that plane is taking off or landing. These "decisions" should appear to be random, yet you should seek to avoid ridiculous scenarios. Your simulation should not have planes waiting a long time to take off, for example. Think carefully about how to avoid such things.

   Your simulation should run for what is its equivalent of one day. For example, if you decide to make each time step represent five minutes, then your simulation should run for 288 time steps. You may design and code a class called `Clock`, if you like.

   Your simulation should make it clear to the user what is happening at each time step. Below is some sample output to use as a guide.

```
The time is 12:05 PM.
There are 7 planes waiting to land.
There are 4 planes waiting to take off.
Plane #23 is cleared to land.

Press <Enter> to continue.

The time is 12:10 PM.
There are 6 planes waiting to land.
There are 4 planes waiting to take off.
Plane #26 is cleared to land.

Press <Enter> to continue.
```