Solutions to Exercises in Chapter 7

7.2 Implementation of the auxiliary method to expand the array representing a queue:

```
private void expand () {
// Make the elems array longer.
  Object[] newElems = new Object[2*elems.length];
  int j = front;
  for (int i = 0; i < length; i++) {
    newElems[i] = elems[j++];
    if (j == elems.length)  j = 0;
  }
  elems = newElems;
  front = 0;  rear = length;
}
```

7.3 We could drop instance variable `rear`, since its value can be computed from `front` and `length` whenever required, using Equation 7.1. Modify the `addLast` operation as follows:

```
public void addLast (Object elem) {
// Add elem as the rear element of this queue.
  if (length == elems.length)  expand();
  int rear = (front + length) % elems.length;
  elems[rear] = elem;
  length++;
}
```

Also, remove all occurrences of `rear` in other operations.

(*Note:* We could similarly drop instance variable `front`. But we cannot drop instance variable `length`, since it would be impossible to tell whether the queue is empty or full when `front` and `rear` are equal.)

7.4 It would be pointless to implement the queue ADT using a DLL, since none of the operations needs to access any node's predecessor.

7.6 A UNIX pipe connecting process P_1 to process P_2 can be implemented by a queue of bytes, q. Initially q is empty. Whenever P_1 writes a byte to the pipe, that byte is added to the rear of q. Whenever P_2 reads a byte from the pipe, that byte is removed from the rear of q.

(*Note:* Since processes P_1 and P_2 are concurrent, we must *synchronize* the queue operations, i.e., ensure that only one operation is called at a time. If the `addLast` and `removeFirst` operations were called at the same time, the instance variables representing the queue would be left in an unpredictable state.)

7.7 The keyboard driver can communicate with the application program via a queue whose elements are characters. The driver adds characters to the rear of the queue, and the application removes them from the front.

(*Note:* As in Exercise 7.6, we must synchronize the queue operations.)

To handle a keyboard (version that ignores all control characters):

1. Make character queue *q* empty.
2. Repeat indefinitely:
   - 2.1. Accept a character *char* from the keyboard.
   - 2.2. If *char* is a graphic character:
     - 2.2.1. Echo *char* to the screen.
     - 2.2.2. Add *char* to the rear of *q*.
   - 2.3. If *char* is a control character:
     - 2.3.1. Do nothing.

To handle a keyboard (version that handles DELETE but ignores all other control characters):

1. Make character queue *q* empty.
2. Repeat indefinitely:
   - 2.1. Accept a character *char* from the keyboard.
   - 2.2. If *char* is a graphic character:
     - 2.2.1. Echo *char* to the screen.
     - 2.2.2. Add *char* to the rear of *q*.
   - 2.3. If *char* is DELETE:
     - 2.3.1. Backspace the screen cursor, blanking out the character there.
     - 2.3.2. Remove the rearmost character of *q*.
   - 2.4. If *char* is a control character other than DELETE:
     - 2.4.1. Do nothing.

(*Note:* Step 2.3.2 removes the *rearmost* (last) element of the queue. But that is not an operation of the standard queue ADT, so we must instead use a special kind of queue, namely the *double-ended queue* of Exercise 7.8.)

7.8 A contract for a deque ADT is shown in Program S7.1.

A DLL implementation of deques is outlined in Program S7.2. With this implementation, all deque operations have time complexity *O*(1).

7.9 The algorithm to reorder a train from *input* to *output*, using *siding*, is essentially the same as the algorithm of Exercise 6.13 (to reorder a train from *input* to *output*, using *spur*). The difference is that *siding* is a queue, so the step "Move car *c′* from *input* to *siding*" should be interpreted as moving a car to the *rear* of *siding*, whereas "Move car *c′* from *siding* to *input*" should be interpreted as moving a car from the *front* of *siding*.

7.10 Suppose that we have *s* sidings, numbered 0, …, *s*–1. Then we can assign cars to sidings according to their car numbers. For example, we can assign car *c* to the siding numbered (*c* modulo *s*). On average, each siding will contain only about 1/*s* times as many cars as in Exercise 6.13, and the excess number of car movements will be reduced by about 1/*s*.

```
public interface Deque {

    // Each Deque object is a deque (double-ended queue) whose elements are
    // objects.

    //////////// Accessors ////////////

    public boolean isEmpty ();
    // Return true if and only if this deque is empty.

    public int length ();
    // Return this deque's length.

    public Object getFirst ();
    // Return the element at the front of this deque. Throw a
    // NoSuchElementException if this deque is empty.
```

```
   public Object getLast ();
   // Return the element at the rear of this deque. Throw a
   // NoSuchElementException if this deque is empty.

   //////////// Transformers ////////////

   public void clear ();
   // Make this deque empty.

   public void addFirst (Object elem);
   // Add elem as the front element of this deque.

   public void addLast (Object elem);
   // Add elem as the rear element of this deque.

   public Object removeFirst ();
   // Remove and return the front element from this deque. Throw a
   // NoSuchElementException if this deque is empty.

   public Object removeLast ();
   // Remove and return the rear element from this deque. Throw a
   // NoSuchElementException if this deque is empty.

}
```

Program S7.1 A contract for a deque ADT.

```
public class LinkedDeque implements Deque {

   // Each LinkedDeque object is a deque (double-ended queue) whose
   // elements are objects.

   // This deque is represented as follows: its length is held in length;
   // front and rear are links to the first and last nodes of a DLL
   // containing its elements.
   private DLLNode front, rear;
   private int length;

   //////////// Constructor ////////////

   public LinkedDeque () {
   // Construct a deque, initially empty.
      front = rear = null;
      length = 0;
   }

   //////////// Accessors ////////////

   …

   public Object getLast () {
   // Return the element at the rear of this deque. Throw a
   // NoSuchElementException if this deque is empty.
      if (rear == null)
         throw new NoSuchElementException();
      return rear.element;
   }
```

Program S7.2 Outline of implementation of unbounded deques using DLLs *(continued on next page)*.

```
//////////// Transformers ////////////
…
public void addFirst (Object elem) {
// Add elem as the front element of this deque.
   DLLNode newest = new DLLNode(elem, null, null);
   if (front != null)
      front.pred = newest;
   else
      rear = newest;
   front = newest;
   length++;
}

public Object removeLast () {
// Remove and return the rear element of this deque. Throw a
// NoSuchElementException if this deque is empty.
   if (rear == null)
      throw new NoSuchElementException();
   Object rearElem = rear.element;
   rear = rear.pred;
   if (rear == null)  front = null;
   length--;
   return rearElem;
}

}
```

Program S7.2 Outline of implementation of unbounded deques using DLLs *(continued)*.