CHAPTER 5

Linked Lists

This chapter reviews Java references and introduces you to the data structure linked list. You will see algorithms for fundamental linked list operations such as insertion and deletion. The chapter also describes several variations of the basic linked list. As you will see, you can use a linked list and its variations when implementing many of the ADTs that appear throughout the remainder of this book. The material in this chapter is thus essential to much of the presentation in the following chapters.

5.1 Preliminaries

The ADT list, as described in the previous chapter, has operations to insert, delete, and retrieve items, given their positions within the list. A close examination of the array-based implementation of the ADT list reveals that an array is not always the best data structure to use to maintain a collection of data. An array has a **fixed size**—at least in most commonly used programming languages—but the ADT list can have an arbitrary length. Thus, in the strict sense, you cannot use an array to implement a list because it is certainly possible for the number of items in the list to exceed the fixed size of the array. When developing implementations for ADTs, you often are confronted with this fixed-size problem. In many contexts, you must reject an implementation that has a fixed size in favor of one that can grow dynamically.

In addition, although the most intuitive means of imposing an order on data is to sequence it physically, this approach has its disadvantages. In a physical ordering, the successor of an item *x* is the next data item in sequence after *x*, that is, the item "to the right" of *x*. An array orders its items physically and, as you saw in the previous chapter, when you use an array to implement a list, you must shift data when you insert or delete an item at a specified position. Shifting data can be a time-consuming process that you should avoid, if possible. What alternatives to shifting data are available?

To get a conceptual notion of a list implementation that does not involve shifting, consider Figure 5-1. This figure should help free you from the notion that the only way to maintain a given order of data is to store the data in that order. In these diagrams, each item of the list is actually *linked to* the next item. Thus, if you know where an item is, you can determine its successor, which can be anywhere physically. This flexibility not only allows you to insert and delete data items without shifting data, but it also allows you to increase the size of the list easily. If you need to insert a new item, you simply find its place in the list and set two **links.** Similarly, to delete an item, you find the item and change a link to bypass the item.

An item in a linked list references its successor

Because the items in this data structure are *linked* to one another, it is called a **linked list.** As you will see shortly, *a linked list is able to grow as needed*, whereas an array can hold only a fixed number of data items. In many applications, this flexibility gives a linked list a significant advantage.

Before we examine linked lists and their use in the implementation of an ADT, we will examine how Java references can be used to implement a linked list. Like many programming languages, Java allows one object to reference another, and you can use this ability to build a linked list. The next section reviews the mechanics of these references.

Object References

A reference contains the address of an object

When you declare a variable that refers to an object of a given class, you are creating a reference to the object. Note that an object of that class does not come into existence until you apply the *new* operator. A **reference variable,** or simply a **reference,** contains the location, or **address** in memory, of an object.

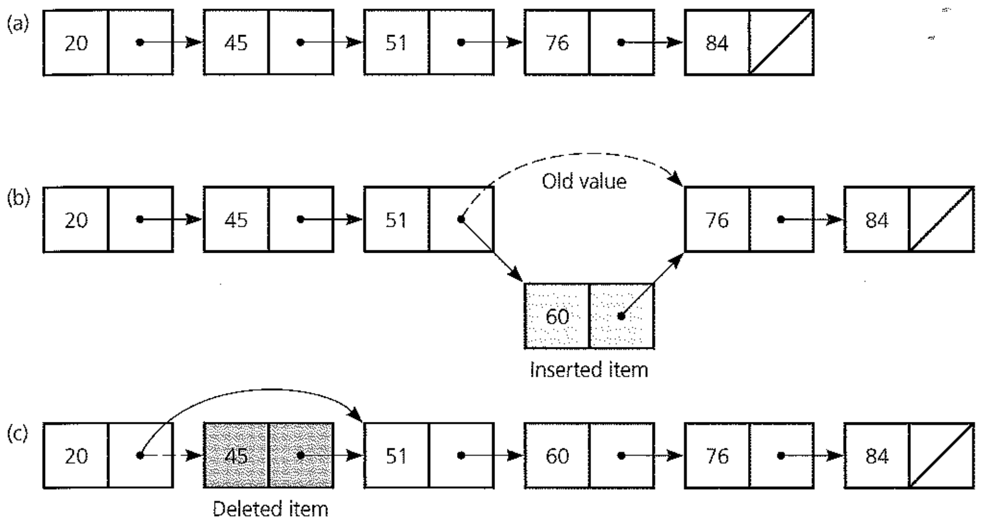


FIGURE 5-1

(a) A linked list of integers; (b) insertion; (c) deletion

By using a reference to a particular object, you can locate the object and, for example, access the object's public members.

Let's look at an example using the class *java.lang.Integer* to help us visualize this scenario:

```
Integer intRef;
intRef = new Integer(5);
```

The first line declares a reference variable *intRef* that can be used to locate an *Integer* object. The second line actually instantiates an *Integer* object and assigns its location to *intRef*. Figure 5-2 illustrates that there are now two separate entities in our program: a reference variable *intRef* that provides the location of an *Integer* object and an *Integer* object.

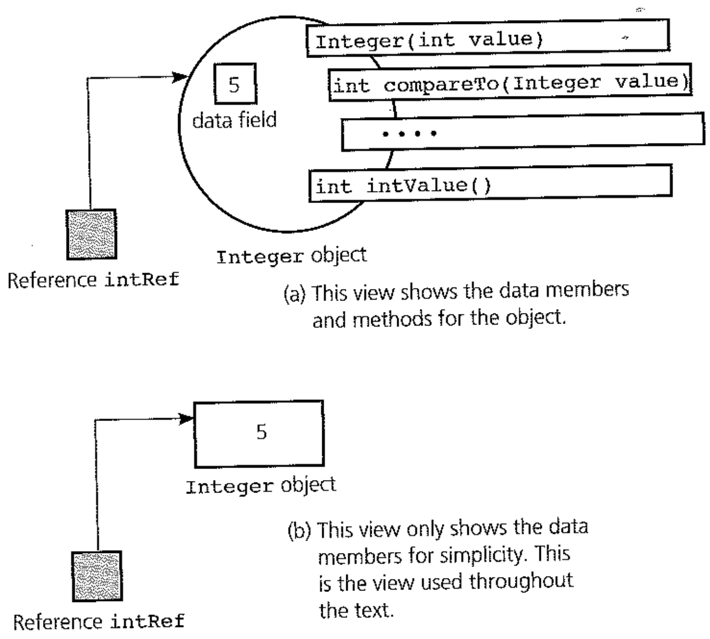
intRef references the newly instantiated ***Integer*** object

When you declare a reference variable as a data field within a class but do not instantiate an object for it in a constructor, it is initialized to *null*. You can use this constant *null* as the value of a reference to any type of object. This use indicates that the reference variable does not currently reference any object. For example, if *intRef* is declared as a data field, and you attempt to use it before you instantiate an *Integer* object for it, the exception *java.lang.NullPointerException* will be thrown at runtime. This exception indicates that you attempted to access an object by using a reference variable that contains a *null* value.

A reference variable as a data field of a class has the default value ***null***

When you declare a reference variable to be local to a method, no default value is provided. For example, let *p* be declared as a reference to an *Integer* object. If you attempt to use *p* to access an object before *p* is initialized, the compiler will give you the error message "*Variable p may not have been initialized.*"

A local reference variable has no default value



Integer(int value)
int compareTo(Integer value)
....
int intValue()
5
data field
Integer object
Reference intRef
(a) This view shows the data members and methods for the object.
5
Integer object
Reference intRef
(b) This view only shows the data members for simplicity. This is the view used throughout the text.

FIGURE 5-2

A reference to an ***Integer*** object

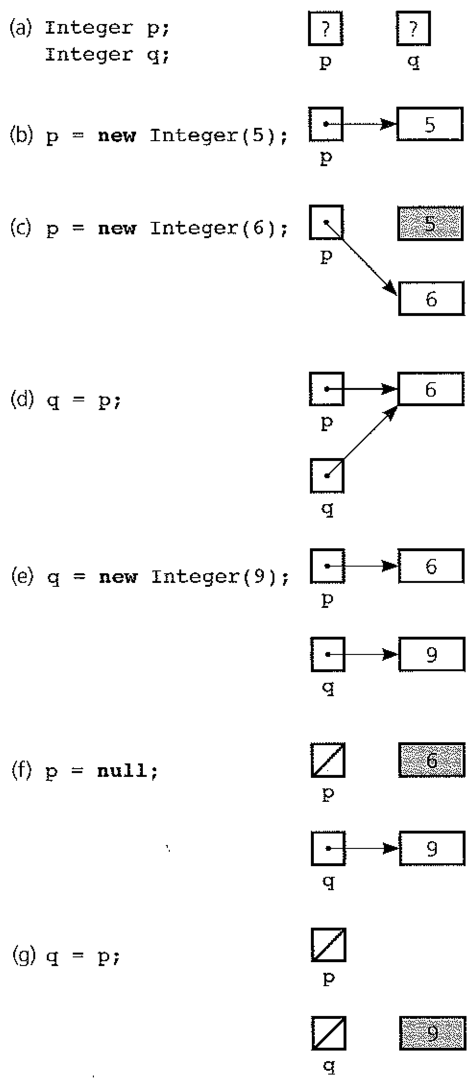
When one reference variable is assigned to another reference variable, both references then refer to the same object. For example,

```
Integer p, q;
p = new Integer(6);
q = p;
```

Figure 5-3d illustrates the result of this assignment. Now the *Integer* object has two references to it, *p* and *q*. The effect of the assignment operator is to cause the reference variable on the left side of the assignment to reference the same object as referenced by the right side of the assignment operator. Alternatively, you could let *q* reference a new object, as Figure 5-3e shows.

Suppose that you no longer need the value in a reference variable. That is, you do not want the reference variable to locate any particular object. You can explicitly assign the constant value *null*, discussed earlier, to a reference variable to indicate that it no longer references any object. When you remove all references to an object, the system marks the object for garbage collection, as shown in Figures 5-3c and 5-3f. The Java runtime environment will periodically run a method that returns the memory allocated to the marked objects back to the system for future use. In other programming languages, such as C++, the programmer must explicitly deallocate memory using special language constructs.

The system marks unreferenced objects for garbage collection



```
(a) Integer p;
    Integer q;

(b) p = new Integer(5);

(c) p = new Integer(6);

(d) q = p;

(e) q = new Integer(9);

(f) p = null;

(g) q = p;
```

FIGURE 5-3

(a) Declaring reference variables; (b) allocating an object; (c) allocating another object, with the dereferenced object marked for garbage collection; (d) assigning a reference; (e) allocating an object; (f) assigning `null` to a reference variable; (g) assigning a reference with a `null` value

When you declare an array of objects and apply the *new* operator, an array of references is actually created, not an array of objects. For example,

An array of objects is actually an array of references to the objects

```
Integer[] scores = new Integer[30];
```

creates an array of 30 references for *Integer* objects. You must instantiate actual *Integer* objects for each of the array references. For example, you might instantiate objects for the array just created as follows:

```
scores[0] = new Integer(7);
scores[1] = new Integer(9); // and so on ...
```

Equality operators compare values of reference variables, not the objects that they reference

When you use the equality operators (== and !=), you are actually comparing the values of the reference variables, not the objects that they reference. Suppose that you have the following class definition:

```
public class MyNumber {
  private int num;

  public MyNumber(int n) {
    num = n;
  } // end constructor

  public String toString() {
    return "My number is " + num;
  } // end toString
} // end class MyNumber
```

and you declare the following:

```
MyNumber x = new MyNumber(9);
MyNumber y = new MyNumber(9);
MyNumber z = x;
```

Although objects *x* and *y* contain the same data, the == operator returns *false*, since *x* and *y* refer to different objects. The expression *x* == *z* is *true* because the assignment statement *z = x* causes *z* to refer to the same object that *x* references. If you need to be able to compare objects field by field, you must redefine the *equals* method for the class, as discussed in Chapter 4.

When you pass an object to a method as an argument, the reference to the object is copied to the method's formal parameter

Parameter passing in Java can also be discussed in terms of reference variables. When a method is called and has parameters that are objects, the reference value of the actual argument is copied to a formal parameter reference variable. During the execution of the method, the object is accessed through the formal parameter reference variable. This provides the same result as if the original reference was used. Upon completion of the method, the references stored in these formal parameters are discarded, although the objects that the parameters reference may be retained.

This method of parameter passing helps to explain why the use of the *new* operator with a formal parameter in a method can produce unexpected results. For example, suppose you have the following method *changeNumber*, which uses the *MyNumber* class:

```
public void changeNumber(MyNumber n) {
  n = new MyNumber(5);
} // end changeNumber
```

and the following Java statements:

```
MyNumber x = new MyNumber(9);
changeNumber(x); // attempts to assign 5 to x
System.out.println(x);
```

The output is "My number is 9". Figure 5-4 demonstrates why this is the case. When the `changeNumber` method is invoked, the reference to object *x* is copied to the formal parameter reference *n* of `changeNumber`. During the execution of `changeNumber`, a new object is created for *n* to reference. But when the `changeNumber` method completes execution, the reference variable *n* is discarded. This causes the newly created object containing 5 to be marked for garbage collection. The value of the reference variable *x* remains unchanged; it still references the same object (containing 9) that it did before the `changeNumber` method was executed.

Note that ADT implementations and data structures that use Java references are said to be **reference based.**

KEY CONCEPTS

Java Reference Variables

1. The declaration

   ```
   Integer intRef;
   ```

   statically allocates a reference variable `intRef` whose value is `null`. When a reference variable contains `null`, it does not reference anything.
2. `intRef` can reference an `Integer` object. The statement

   ```
   intRef = new Integer(5);
   ```

   dynamically allocates an `Integer` object referenced by `intRef`. (However, see item 3 on this list.)
3. If, for some reason, `new` cannot instantiate an object of the class represented, it may throw a `java.lang.InstantiationException` or a `java.lang.IllegalAccessException`. Thus, you can place the following statement within a `try` block to test whether memory was successfully allocated:

   ```
   intRef = new Integer(5);
   ```
4. When the last reference to an object is removed, the object is marked for garbage collection.

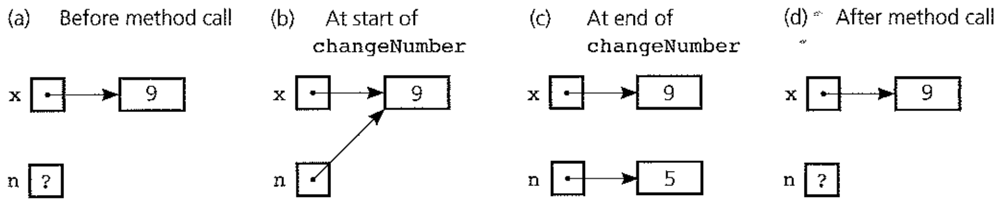


FIGURE 5-4

The value of a parameter does not affect the argument's value

Resizeable Arrays

When you declare an array in Java by using statements such as

The number of references in a Java array is of a fixed size

```
final int MAX_SIZE = 50;
double[] myArray = new double[MAX_SIZE];
```

the Java runtime environment reserves a specific number—*MAX_SIZE*, in this case—of references for the array. Once the array has been instantiated, it has a fixed size for the remainder of its lifetime. We have already discussed the problem this fixed-size data structure causes when your program has more than *MAX_SIZE* items to place into the array.

You can create the illusion of a resizeable array—an array that grows and shrinks as the program executes—by using an allocate and copy strategy with fixed-size arrays. If the array that you are currently using in the program reaches its capacity, you allocate a larger array and copy the references stored in the original array to the larger array. How much larger should the new array be? The increment size can be a fixed number of elements or a multiple of the current array size. The following statements demonstrate how you could accomplish this task for an array *myArray*:

Allocate a larger array

Copy the original array to the new larger array

```
if (capacityIncrement == 0) {
  capacity *= 2;
}
else {
  capacity += capacityIncrement;
}
// now create a new array using the updated
// capacity value
double [] newArray = new double[capacity];
// copy the contents of the original array
// to the new array
for (int i = 0; i < myArray.length; i++) {
  newArray[i] = myArray[i];
} // end for
```

```
// now change the reference to the original array
// to the new array
myArray = newArray;
```

In this example, *capacity* and *capacityIncrement* represent the capacity of the array and the size of the increment, respectively. Once you exceed the capacity of *myArray*, you allocate a larger array *newArray* according to the value of *capacityIncrement*. Note that if the *capacityIncrement* is zero, the array capacity doubles instead of increasing by a fixed amount. You must copy the values from the original array to the new array and then change the original array reference to reference the new array.

The classes *java.util.Vector* and *java.util.ArrayList* use a similar technique to implement a growable array of objects. The underlying implementation of *java.util.Vector* uses a fixed array of size *capacity* and has a *capacityIncrement* that you can change to suit your needs. Exercise 19 asks you to explore the *java.util.Vector* class to determine when resizing the underlying array will occur.

Subsequent discussion in this book will refer to both fixed-sized and resizeable arrays. Our array-based ADT implementations will use fixed-sized arrays for simplicity. The programming problems will ask you to create array-based implementations that use resizeable arrays.

Reference-Based Linked Lists

A linked list, such as the one in Figure 5-1, contains components that are linked to one another. Each component—usually called a **node**—contains both data and a "link" to the next item. Typically, such links are Java reference variables; another possibility is mentioned at the end of this section. Although you have seen most of the mechanics of references, using references to implement a linked list is probably not yet completely clear to you. Consider now how you can set up such a linked list.

A node in a linked list is an object

Each node of the list can be implemented as an object. For example, if you want to create a linked list of integers, you could use the following class definition, as Figure 5-5 illustrates:

A node definition that is not desirable because its data fields are public

```
public class IntegerNode {
  public int item;
  public IntegerNode next;
} // end class IntegerNode
```

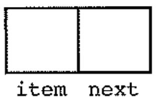


FIGURE 5-5
A node

However, this type of definition violates our rule that data fields must be declared private. We could rewrite this class with accessor and mutator methods as follows:

A node for a linked list of integers

```
public class IntegerNode {
  private int item;
  private IntegerNode next;
  public void setItem(int newItem) {
    item = newItem;
  } // end setItem

  public int getItem() {
    return item;
  } // end getItem

  public void setNext(IntegerNode nextNode) {
    next = nextNode;
  } // end setNext

  public IntegerNode getNext() {
    return next;
  } // end getNext

} // end class IntegerNode
```

and use this class as follows:

Defining a reference to a node

```
IntegerNode n1 = new IntegerNode();
IntegerNode n2 = new IntegerNode();
n1.setItem(5); // set item in first node
n2.setItem(9); // set item in second node
n1.setNext(n2); // link the nodes
```

This scenario is depicted in Figure 5-6. The mutator methods *setItem* and *setNext* initialize the data fields of the two nodes. Note how *setNext* links the nodes by making the first node reference the second.

We could improve this class by adding constructors, as follows:

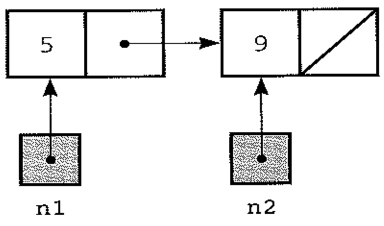


FIGURE 5-6

The result of linking two instances of ***IntegerNode***

```
public class IntegerNode {
  private int item;
  private IntegerNode next;

  public IntegerNode(int newItem) {
    item = newItem;
    next = null;
  } // end constructor

  public IntegerNode(int newItem, IntegerNode nextNode) {
    item = newItem;
    next = nextNode;
  } // end constructor
    // setItem, getItem, setNext, getNext as before
    ...
}   // end class IntegerNode
```

Constructors

This definition of a node restricts the data to a single integer field. Since we would like to have this class be as reusable as possible, it would be better to change the data field to be of type *Object*. Recall that every class in Java is ultimately derived from the class *Object* through inheritance. This means that any class created in Java could use this node definition for storing objects. Let's first examine the revised class, using objects for data:

A node for a linked list of objects

```
public class Node {
  private Object item;
  private Node next;

  public Node(Object newItem) {
    item = newItem;
    next = null;
  } // end constructor

  public Node(Object newItem, Node nextNode) {
    item = newItem;
    next = nextNode;
  } // end constructor

  public void setItem(Object newItem) {
    item = newItem;
  } // end setItem

  public Object getItem() {
    return item;
  } // end getItem
  public void setNext(Node nextNode) {
    next = nextNode;
  } // end setNext

  public Node getNext() {
    return next;
  } // end getNext
} // end class Node
```

You can use this class as follows:

```
Node n = new Node(new Integer(6));
Node first = new Node(new Integer(9), n);
```

Figure 5-7 illustrates this scenario. The constructors are used to initialize the data field and a link value that is either *null* or provided as an argument. Although the data portion of each node in a linked list can reference an instance of any class, the figure illustrates data items that are instances of the class *java.lang.Integer*.

To complete our general description of the linked list, we must consider two other issues. First, what is the value of the data field *next* in the last node in the list? By setting this field to *null*, you can easily detect when you are at the end of the linked list.

Second, nothing so far references the beginning of the linked list. If you cannot get to the beginning of the list, you cannot get to the second node in the list, and if you cannot get to the second node in the list, you cannot get to the third node in the list, and so on. The solution is to have an additional reference variable whose sole purpose is to locate the first node in the linked list. Such a variable is often called ***head***.

The head of a linked list references the list's first node

Observe in Figure 5-8 that the reference variable *head* is different from the other reference variables in the diagram in that it is not within one of the nodes. Rather, it is a simple reference variable that is external to the linked list, whereas the *next* data fields are internal reference variables within the nodes of

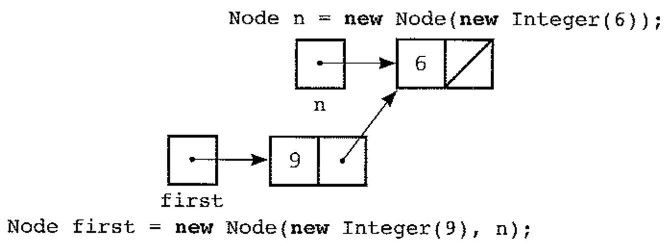

Node n = new Node(new Integer(6));
6
n
9
first
Node first = new Node(new Integer(9), n);

FIGURE 5-7

Using the ***Node*** constructor to initialize a data field and a link value

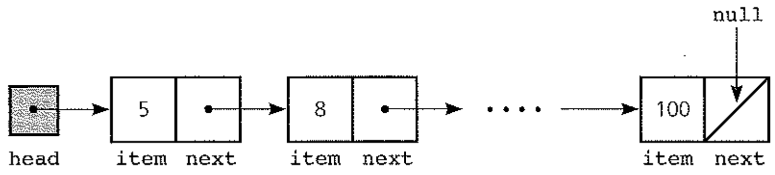


FIGURE 5-8

A ***head*** reference to a linked list

the list. The variable *head* simply enables you to access the list's beginning. Also, note that *head* always exists, even at times when there are no nodes in the linked list. The statement

```
Node head = null;
```

creates the variable *head*, whose value is initially *null*. This indicates that *head* does not reference anything, and therefore that this list is empty.

It is a common mistake to think that before you can assign *head* a value, you must execute the statement *head = new Node()*. This misconception is rooted in the belief that the variable *head* does not exist unless you create a new node. This is not at all true; *head* is a reference variable waiting to be assigned a value. Thus, for example, you can assign *null* to *head* without first using *new*. In fact, the sequence

```
head = new Node();  // Don't really need to use new here
head = null;     // since we lose the new Node object here
```

A common misconception

destroys the contents of the only reference—*head*—to the newly created node, as Figure 5-9 illustrates. Thus, you have needlessly created a new node and then made it inaccessible. Remember that when you remove the last reference from a node, the system marks it for garbage collection.

As was mentioned earlier, you do not need references to implement a linked list. Programming Problem 10 at the end of this chapter discusses an implementation that uses an array to represent the items in a linked list. Although sometimes useful, such implementations are unusual.

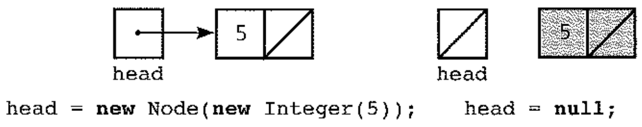


FIGURE 5-9

A lost node

5.2 Programming with Linked Lists

The previous section illustrated how you can use reference variables to implement a linked list. This section begins by developing algorithms for displaying the data portions of such a linked list and for inserting items into and deleting items from a linked list. These linked list operations are the basis of many of the data structures that appear throughout the remainder of the book. Thus, the material in this section is essential to much of the discussion in the following chapters.

Displaying the Contents of a Linked List

Suppose now that you have a linked list, as was pictured in Figure 5-8, and that you want to display the data in the list. A high-level pseudocode solution is

```
Let a variable curr reference the first node in
    the linked list
while (the curr reference is not null) {
  Display the data portion of the current node
  Set the curr reference to the next field of the
      current node
}  // end while
```

This solution requires that you keep track of the current position within the linked list. Thus, you need a reference variable *curr* that references the current node. Initially, *curr* must reference the first node. Since *head* references the first node, simply copy *head* into *curr* by writing

```
Node curr = head;
```

To display the data portion of the current node, you can use the statement[1]

```
System.out.println(curr.getItem());
```

Finally, to advance the current position to the next node, you write

```
curr = curr.getNext();
```

Figure 5-10 illustrates this action. If the previous assignment statement is not clear, consider

```
temp = curr.getNext();
curr = temp;
```

1. See Chapter 1 for a discussion of *System.out.println* with object parameters.

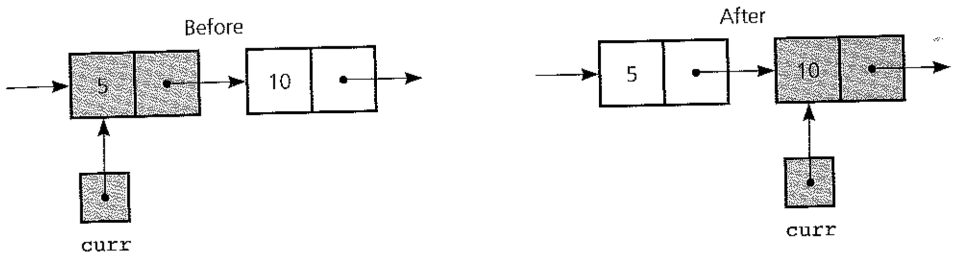


FIGURE 5-10

The effect of the assignment *curr = curr.getNext()*

and then convince yourself that the intermediate variable *temp* is not necessary.

These ideas lead to the following loop in Java:

```
// Display the data in a linked list that head
// references.
// Loop invariant: curr references the next node to be
// displayed
for (Node curr = head; curr != null; curr = curr.getNext()) {
  System.out.println(curr.getItem());
} // end for
```

The variable *curr* references each node in a nonempty linked list during the course of the *for* loop's execution, and so the data portion of each node is displayed. After the last node is displayed, *curr* becomes *null* and the *for* loop terminates. When the list is empty—that is, when *head* is *null*—the *for* loop is correctly skipped.

A common error in the *for* statement is to compare *curr.getNext()* instead of *curr* with *null*. When *curr* references the last node of a nonempty linked list, *curr.getNext()* is *null*, and so the *for* loop would terminate before displaying the data in the last node. In addition, when the list is empty—that is, when *head* and therefore *curr* are *null*—*curr.getNext()* will throw a *NullPointerException*. Such references are incorrect and should be avoided.

A traverse operation visits each node in the linked list

Displaying a linked list is an example of a common operation, **list traversal.** A traversal sequentially **visits** each node in the list until it reaches the end of the list. Our example displays the data portion of each node when it visits the node. Later in this book, you will see that you can do other useful things to a node during a visit.

Displaying a linked list does not alter it; you will now see operations that modify a linked list by deleting and inserting nodes. These operations assume that the linked list has already been created. Ultimately, you will see how to build a linked list by inserting nodes into an initially empty list.

Deleting a Specified Node from a Linked List

So that you can focus on how to delete a particular node from a linked list, assume that the linked list shown in Figure 5-11 already exists. Notice that, in addition to *head*, the diagram includes two external reference variables, *curr* and *prev*. The task is to delete the node that *curr* references. As you soon will see, you also need *prev* to complete the deletion. For the moment, do not worry about how to establish *curr* and *prev*.

As Figure 5-11 indicates, you can delete a node *N*, which *curr* references, by altering the value of the reference *next* in the node that precedes *N*. You need to set this data field to reference the node that follows *N*, thus bypassing *N* on the chain. (The dashed line indicates the old reference value.) Notice that this reference change does not directly affect node *N*. Since *curr* still references node *N*, the node remains in existence, and it references the same node that it referenced before the deletion. However, the node has effectively been deleted from the linked list. For example, the method *displayList* from the previous section would not display the contents of node *N*. If *curr* is the only reference to node *N*, when we change *curr* to reference another node or set it equal to *null*, node *N* is marked for garbage collection.

To accomplish this deletion, notice first that if only the reference *curr* points to *N*, you would have no direct way to access the node that precedes *N*. After all, you cannot follow the links in the list backward. However, notice that the reference variable *prev* in Figure 5-11 references the node that precedes *N* and makes it possible for you to alter that node's *next* data field. Doing so deletes node *N* from the linked list. The following assignment statement is all that you need to delete the node that *curr* references:

Deleting an interior node

```
prev.setNext(curr.getNext());
```

A question comes to mind at this point:

- Does the previous method work for any node *N*, regardless of where in the linked list it appears?

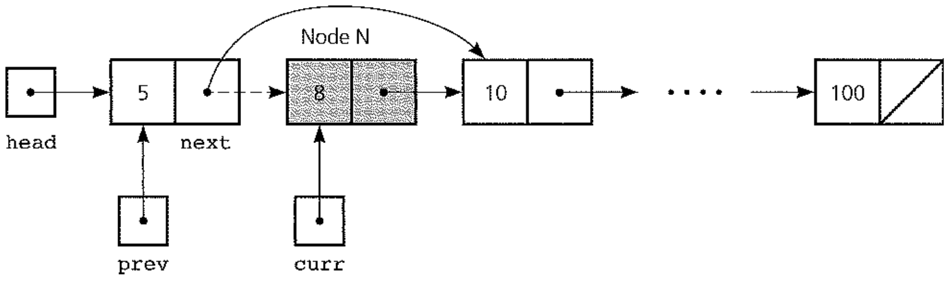


FIGURE 5-11

Deleting a node from a linked list

No, the method does not work if the node to be deleted is the *first* node in the list, because it certainly does not make sense to assert that *prev* references the node that precedes this node! Thus, ***deletion of the first node in a linked list is a special case,*** as Figure 5-12 depicts. In this case, *curr* references the first node and *prev* is *null*.

When you delete the first node of the list, you must change the value of *head* to reflect the fact that, after the deletion, the list has a new first node. That is, the node that was second prior to the deletion is now first. You make this change to *head* by using the assignment statement

```
head = head.getNext();
```

Deleting the first node is a special case

As was the case for the deletion of an interior node, the *head* reference now bypasses the old first node. Notice also that if the node to be deleted is the *only* node in the list—and thus it is both the first node and the last node—the previous assignment statement assigns the value *null* to the variable *head*. Recall that the value *null* in *head* indicates an empty list, and so this assignment statement handles the deletion of the only node in a list correctly.

If the node *N* is no longer needed, you should change the *next* data field of the node *N* to *null* and also the value of *curr* to *null*, as the following statements show:

```
curr.setNext(null);
curr = null;
```

This serves two purposes. First, the reference variables *curr* and *next* (in node *N*) can't be inadvertently followed, leading to subtle errors later in the program. Second, the system can now use this returned memory and possibly even reallocate it to your program as a result of the *new* operator.

So far, we have deleted the node *N* that *curr* references, given a reference variable *prev* to the node that precedes *N*. However, another question remains:

- How did the variables *curr* and *prev* come to reference the appropriate nodes?

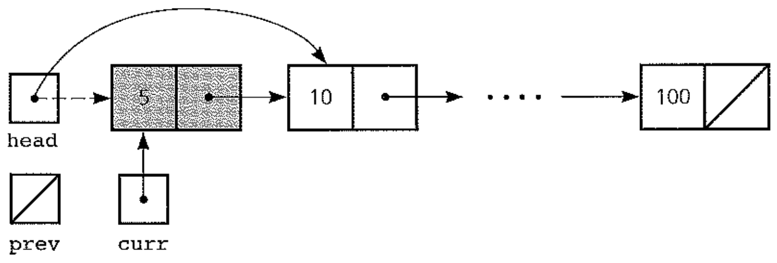


FIGURE 5-12

Deleting the first node

To answer this question, consider the context in which you might expect to delete a node. In one common situation, you need to delete a node that you specify by position. Such is the case if you use a linked list to implement an ADT list. In another situation, you need to delete a node that contains a particular data value. Such is the case if you use a linked list to implement an ADT sorted list. In both of these situations, you do not pass the values of *curr* and *prev* to the deletion method, but instead the method establishes these values as its first step by searching the linked list for the node *N* that either is at a specified position or contains the data value to be deleted. Once the method finds the node *N*—and the node that precedes *N*—the deletion of *N* proceeds as described previously. The details of determining *curr* and *prev* for deletion are actually the same as for insertion, and they appear in the next section.

To summarize, the deletion process has three high-level steps:

Three steps to delete a node from a linked list

1. Locate the node that you want to delete.
2. Disconnect this node from the linked list by changing references.
3. Return the node to the system.

Later in this chapter, we will incorporate this deletion process into the implementation of the ADT list.

Inserting a Node into a Specified Position of a Linked List

Figure 5-13 illustrates the technique of inserting a new node into a specified position of a linked list. You insert the new node, which the reference variable *newNode* references, between the two nodes that *prev* and *curr* reference. As the diagram suggests, you can accomplish the insertion by using the following pair of assignment statements:

Inserting a node between nodes

```
newNode.setNext(curr);
prev.setNext(newNode);
```

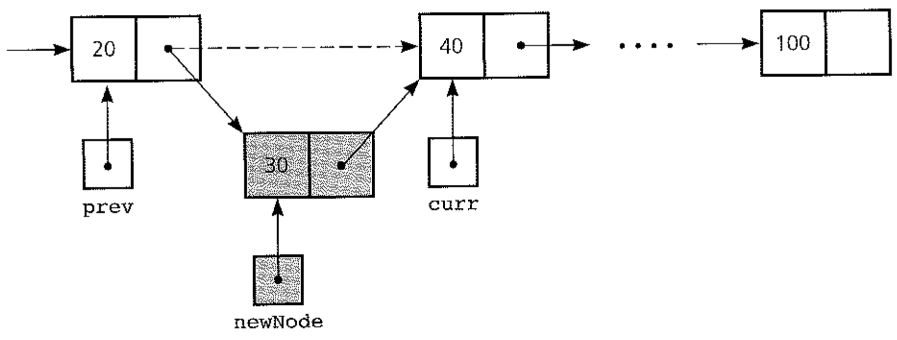


FIGURE 5-13

Inserting a new node into a linked list

The following two questions are analogous to those previously asked about the deletion of a node:

- How did the variables *newNode*, *curr*, and *prev* come to reference the appropriate nodes?
- Does the method work for inserting a node into any position of a linked list?

The answer to the first question, like the answer to the analogous question for deletion, is found by considering the context in which you will use the insertion operation. You establish the values of *curr* and *prev* by traversing the linked list until you find the proper position for the new item. You then use the *new* operator to create a new node that references the item as follows:

```
newNode = new Node(item);
```

Creating a node for the new item

You can now insert the node into the list, as was just described.

The answer to the second question is that *insertion, like deletion, must account for special cases.* First, consider the insertion of a node at the beginning of the linked list, as shown in Figure 5-14. You must make *head* reference the new node, and the new node must reference the node that had been at the beginning of the list. You accomplish this by using these statements:

```
newNode.setNext(head);
head = newNode;
```

Inserting a node at the beginning of a linked list

Observe that if the list is empty before the insertion, *head* is *null*, so the *next* reference of the new item is set to *null*. This step is correct because the new item is the last item—as well as the first item—on the list.

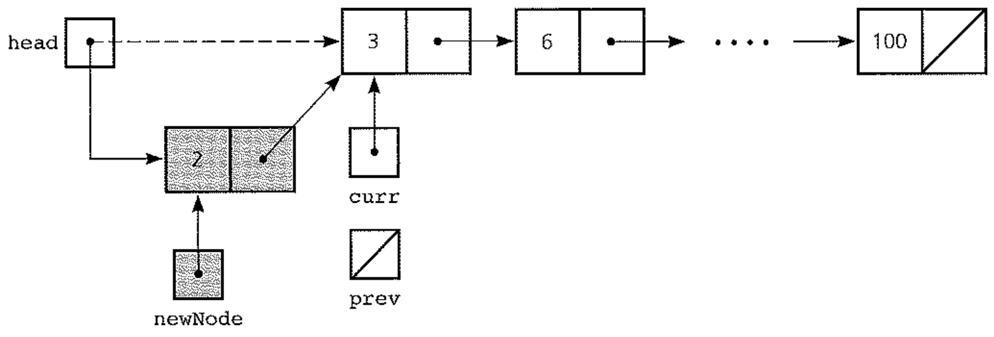


FIGURE 5-14

Inserting at the beginning of a linked list

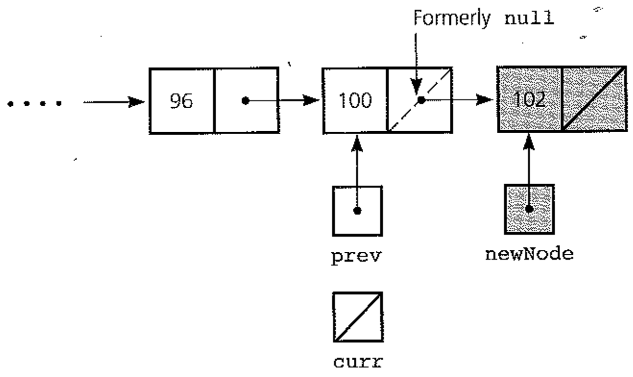


FIGURE 5-15

Inserting at the end of a linked list

Figure 5-15 shows the insertion of a new node at the end of a linked list. This insertion is potentially a special case because the intention of the pair of assignment statements

If ***curr*** is ***null***, inserting at the end of a linked list is not a special case

```
newNode.setNext(curr);
prev.setNext(newNode);
```

is to insert the new node *between* the node that *curr* references and the node that *prev* references. If you are to insert the new node at the end of the list, what node should *curr* reference? In this situation, it makes sense to view the value of *curr* as *null* because, as you traverse the list, *curr* becomes *null* as it moves past the end of the list. Observe that if *curr* has the value *null* and *prev* references the last node on the list, the previous pair of assignment statements will indeed insert the new node at the end of the list. Thus, insertion at the end of a linked list is not a special case.

To summarize, the insertion process requires three high-level steps:

Three steps to insert a new node into a linked list

1. Determine the point of insertion.
2. Create a new node and store the new data in it.
3. Connect the new node to the linked list by changing references.

Determining *curr* and *prev*. Let us now examine in more detail how to determine the references *curr* and *prev* for the insertion operation just described. As was mentioned, this determination depends on the context in which you will insert a node. As an example, consider a linked list of integers that are sorted into ascending order using the *IntegerNode* class in the previous section. To simplify the discussion, assume that the integers are distinct; that is, no duplicates are present in the list.

To determine the point at which the value *newValue* should be inserted into a sorted linked list, you must traverse the list from its beginning until you find the appropriate place for *newValue*. This appropriate place is just before the node that contains the first data item greater than *newValue*. You know that you will need a reference *curr* to the node that is to follow the new node; that is, *curr* references the node that contains the first data item greater than *newValue*. You also need a reference *prev* to the node that is to precede the new node; that is, *prev* references the node that contains the last data item smaller than *newValue*. Thus, as you traverse the linked list, you keep both a current reference *curr* and a ***trailing*** reference *prev*. When you reach the node that contains the first value larger than *newValue*, the trailing reference *prev* references the previous node. At this time, you can insert the new node between the two nodes that *prev* and *curr* reference, as was described earlier.

A first attempt at some pseudocode follows:

A first attempt at a solution

```
// determine the point of insertion into a sorted
// linked list

// initialize prev and curr to start the traversal
// from the beginning of the list
prev = null
curr = head

// advance prev and curr as long as
// newValue > the current data item
// Loop invariant: newValue > data items in all
// nodes at and before the node that prev references
while (newValue > curr.getItem()) { // causes a problem!
  prev = curr
  curr = curr.getNext()
} // end while
```

Unfortunately, the *while* loop causes a problem when the new value is greater than all the values in the list, that is, when the insertion will be at the end of the linked list (or when the linked list is empty). Eventually, the *while* statement compares *newValue* to the value in the last node. During that execution of the loop, *curr* is assigned the value *null*. After this iteration, *newValue* is again compared to *curr.getItem()*, which, when *curr* is *null*, will throw the exception *NullPointerException*.

To solve this problem, you need another test in the termination condition of the *while* statement so that the loop exits when *curr* becomes *null*. Thus, you replace the *while* statement with

```
while (curr != null && newValue > curr.getItem())
```

The revised pseudocode is

The correct solution

```
// determine the point of insertion into a sorted
// linked list

// initialize prev and curr to start the traversal
// from the beginning of the list
```

```
prev = null
curr = head

// advance prev and curr as long as newValue > the
// current data item; do not go beyond end of list
// Loop invariant: newValue > data items in all
// nodes at and before the node that prev references
while (curr != null && newValue > curr.getItem()) {
  prev = curr
  curr = curr.getNext()
}  // end while
```

Notice how the *while* statement also solves the problem of inserting a node at the end of the linked list. In the case where *newValue* is greater than all the values in the list, *prev* references the last node in the list and *curr* becomes *null*, thus terminating the *while* loop. (See Figure 5-16.) Therefore, as you saw earlier, you can insert the new node at the end of the list by using the standard pair of assignment statements

Insertion at the end of a linked list is not a special case

```
newNode.setNext(curr);
prev.setNext(newNode);
```

Now consider the insertion of a node at the beginning of the linked list. This situation arises when the value to be inserted is *smaller* than all the values currently in the list. In this case, the *while* loop in the previous pseudocode is never entered, so *prev* and *curr* maintain their original values, as Figure 5-17 illustrates. In particular, *prev* maintains its original value of *null*. This is the only situation in which the value of *prev* is equal to *null* after execution of the *while* loop ends. Thus, you can detect an insertion at the beginning of the list by comparing *prev* to *null*.

When ***prev*** is ***null***, insertion will be at the beginning of the linked list

Observe that the solution also correctly handles insertion into an empty linked list as an insertion at the beginning of the list. When the list is empty, the statement *curr = head* assigns *curr* an initial value of *null*, and thus the *while* loop is never entered. Therefore, *prev* maintains its original value of *null*, indicating an insertion at the beginning of the list.

Insertion into an empty linked list is really an insertion at the beginning of the list

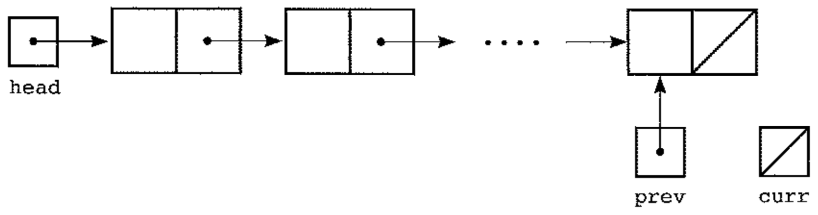


FIGURE 5-16

When ***prev*** references the last node and ***curr*** is ***null***, insertion will be at the end of the linked list

A little thought should convince you that the solution that determines the point of insertion also works for deletion. If you want to delete a given integer from a linked list of sorted integers, you obviously want to traverse the list until you find the node that contains the value sought. The previous pseudocode will do just that: *curr* will reference the desired node and *prev* either will reference the preceding node or, if the desired node is first on the list, will be *null*, as shown in Figure 5-17.

The following Java statements implement the previous pseudocode:

A Java solution for the point of insertion or deletion

```
// determine the point of insertion or deletion
// for a sorted linked list
// Loop invariant: newValue > data items in all
// nodes at and before the node that prev references
for ( prev = null, curr = head;
      (curr != null) && (newValue > curr.getItem())
      prev = curr, curr = curr.getNext() ) {
  // no statements in loop body
} // end for
```

Recall that the *&&* (*and*) operator in Java does not evaluate its second operand if its first operand is *false*. Thus, when *curr* becomes *null*, the loop exits without attempting to evaluate *curr.getItem()*. It is, therefore, essential that *curr != null* be first in the logical expression.

Notice that this implementation relies on the ability to compare one value to another by using the built-in greater than (>) operation for the primitive data type *int*. Suppose instead that the items in the list are of data type *Object*. As mentioned in Chapter 4, you can compare objects if they implement the interface *java.lang.Comparable* and have an implementation of the method *compareTo*.

You can then use the following code to find the location of the item *newValue* of type *Comparable* within the list:

Determining the point of insertion or deletion for a sorted linked list of objects

```
// determine the point of insertion or deletion
// for a sorted linked list of objects
// Loop invariant: newValue > data items (using
```

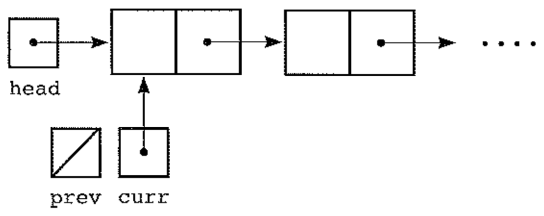


FIGURE 5-17

When ***prev*** is ***null*** and ***curr*** references the first node, insertion or deletion will be at the beginning of the linked list

```
// compareTo method) in all nodes at and before
// the node that prev references
for ( prev = null, curr = head;
      (curr != null ) &&
      (newValue.compareTo(curr.getItem()) > 0);
      prev = curr, curr = curr.getNext() ) {
} // end for
```

Use **compareTo** to compare objects

The *compareTo* method defines the criteria to decide when objects are equal or when one object is less than or greater than another. This in turn can be used to create a sorted list of objects based upon the criteria defined by the new comparison method.

Determining the values of *curr* and *prev* is simpler when you insert or delete a node by position instead of by its value. This determination is necessary when you use a linked list to implement the ADT list, as you will see next.

A Reference-Based Implementation of the ADT List

This section considers how you can use Java references instead of an array to implement the ADT list. Unlike the array-based implementation, a reference-based implementation does not shift items during insertion and deletion operations. It also does not impose a fixed maximum length on the list—except, of course, as imposed by the storage limits of the system.

As in Chapter 4, and as we will do in the rest of the book, we will implement this ADT as a Java class. For the array-based implementation, we wrote declarations for public methods corresponding to the operation of the ADT list. These declarations will appear unchanged in the reference-based implementation.

You need to represent the items in the ADT list and its length. Figure 5-18 indicates one possible way to represent this data by using references. The variable *head* references a linked list of the items in the ADT list, where the first node in the linked list contains the first item in the ADT list and so on. The variable *numItems* is an integer that is the current number of items in the list. Both *head* and *numItems* will be private data fields of our class.

head and ***numItems*** are private data fields

As you saw previously, you use two references—*curr* and *prev*—to manipulate a linked list. These reference variables will be local to the methods that need them; they are not appropriate data fields of the class.

curr and ***prev*** should not be data fields of the class

Recall that the ADT list operations for insertion, deletion, and retrieval specify the position number *i* of the relevant item. In an attempt to obtain values for *curr* and *prev* from *i*, suppose that you define a method *find(i)*

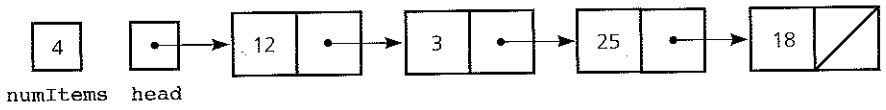


FIGURE 5-18

A reference-based implementation of the ADT list that returns a reference to the *i*th node in the linked list. If `find` provides a reference `curr` to the *i*th node, how will you get a reference `prev` to the previous node, that is, to the (*i* – 1)th node? You can get the value of `prev` by invoking `find(i-1)`. Instead of calling `find` twice, however, note that once you have `prev`, `curr` is simply `prev.getNext()`. The only exception to using `find` in this way is for the first node, but you know immediately from *i* whether the operation involves the first node. If it does, you know the reference to the first node, namely `head`, without invoking `find`.

The method `find` is not an ADT operation. Because `find` returns a reference, you would not want any client to call it. Such clients should be able to use the ADT without knowledge of the references that the implementation uses. It is perfectly reasonable for the implementation of an ADT to define variables and methods that the rest of the program should not access. Therefore, `find` is a private method that only the implementations of the ADT operations call.

`find` is a private method

The following interface specification developed in Chapter 4 will be used for the reference-based implementation of the ADT list. The pre- and postconditions for the ADT list operations are the same as for the array-based implementation that you saw in Chapter 4; they are omitted here to save space.

```
// ****************************************************
// Interface for the ADT list
// ****************************************************
public interface ListInterface {
  // list operations:
  public boolean isEmpty();
  public int size();
  public void add(int index, Object item)
                       throws ListIndexOutOfBoundsException;
  public void remove(int index)
                       throws ListIndexOutOfBoundsException;
  public Object get(int index)
                       throws ListIndexOutOfBoundsException;
  public void removeAll();
} // end ListInterface
```

The implementation of the list begins as follows:

```
// ****************************************************
// Reference-based implementation of ADT list.
// ****************************************************
public class ListReferenceBased implements ListInterface {
  // reference to linked list of items
  private Node head;
  private int numItems; // number of items in list

  // definitions of constructors and methods
    . . .
```

You include the implementations of the class's methods at this point in the class as well as any private methods that may be needed. We now examine each of these implementations.

Default constructor. The default constructor simply initializes the data fields *numItems* and *head*:

Default constructor

```
public ListReferenceBased() {
  numItems = 0;
  head = null;
} // end default constructor
```

Since the variables *numItems* and *head* are initialized to these same values by default, this constructor is really not necessary. But if you have other constructors defined and you want to allow for a constructor without parameters, it must be defined explicitly. In general, it's a good idea to define all constructors explicitly.

List operations. The methods *isEmpty* and *size* have straightforward implementations:

```
public boolean isEmpty() {
  return numItems == 0;
} // end isEmpty

public int size() {
  return numItems;
} // end size
```

Because a linked list does not provide direct access to a specified position, the retrieval, insertion, and deletion operations must all traverse the list from its beginning until the specified point is reached. The method *find* performs this traversal and has the following implementation:

Private method to locate a particular node

```
private Node find(int index) {
// --------------------------------------------------
// Locates a specified node in a linked list.
// Precondition: index is the number of the desired
// node. Assumes that 1 <= index <= numItems+1
// Postcondition: Returns a reference to the desired
// node.
// --------------------------------------------------
  Node curr = head;
  for (int skip = 1; skip < index; skip++) {
    curr = curr.getNext();
  } // end for
  return curr;
} // end find
```

The precondition for *find* requires the *index* to be in the proper range. The *get* operation calls *find* to locate the desired node:

Retrieved by position

```
public Object get(int index)
              throws ListIndexOutOfBoundsException {
  if (index >= 1 && index <= numItems) {
    // get reference to node, then data in node
    Node curr = find(index);
    Object dataItem = curr.getItem();
    return dataItem;
  }
  else {
    throw new ListIndexOutOfBoundsException(
                        "List index out of bounds on get");
  } // end if
} // end get
```

The reference-based implementations of the insertion and deletion operations use the linked list processing techniques developed earlier in this chapter. To insert an item after the first item of a list, you must first obtain a reference to the preceding item. Insertion into the first position of a list is a special case.

Insertion at a given position

```
public void add(int index, Object item)
               throws ListIndexOutOfBoundsException {
  if (index >= 1 && index <= numItems+1) {
    if (index == 1) {
      // insert the new node containing item at
      // beginning of list
      Node newNode = new Node(item, head);
      head = newNode;
    }
    else {
      Node prev = find(index-1);

      // insert the new node containing item after
      // the node that prev references
      Node newNode = new Node(item, prev.getNext());
      prev.setNext(newNode);
    } // end if
    numItems++;
  }
  else {
    throw new ListIndexOutOfBoundsException(
                        "List index out of bounds on add");
  } // end if
}  // end add
```

The *remove* operation is analogous to insertion. To delete an item that occurs after the first item of a list, you must first obtain a reference to the item that precedes it. Removal from the first position of a list is a special case.

Deletion from a given position

```
public void remove(int index)
                  throws ListIndexOutOfBoundsException {
  if (index >= 1 && index <= numItems) {
    if (index == 1) {
      // delete the first node from the list
      head = head.getNext();
    }
    else {
      Node prev = find(index-1);
      // delete the node after the node that prev
      // references, save reference to node
      Node curr = prev.getNext();
      prev.setNext(curr.getNext());
    } // end if
    numItems--;
  } // end if
  else {
    throw new ListIndexOutOfBoundsException(
                 "List index out of bounds on remove");
  } // end if
}   // end remove
```

The *removeAll* operation simply sets the head reference to *null*, making the nodes in the list unreachable and thus marking them for garbage collection.

```
public void removeAll() {
  // setting head to null causes list to be
  // unreachable and thus marked for garbage
  // collection
  head = null;
  numItems = 0;
} // end removeAll
```

Comparing Array-Based and Reference-Based Implementations

Typically, the various implementations that a programmer contemplates for a particular ADT have advantages and disadvantages. When you must select an implementation, you should weigh these advantages and disadvantages before you make your choice. As you will see, the decision among possible implementations of an ADT is one that you must make time and time again. This section compares the two implementations of the ADT list that you have seen as an example of how you should proceed in general.

The array-based implementation that you saw in Chapter 4 appears to be a reasonable approach. An array behaves like a list, and arrays are easy to use. However, as was already mentioned, an array has a fixed size; it is possible for the number of items in the list to exceed this fixed size. In practice, when choosing among implementations of an ADT, you must ask the question, does the fixed-size restriction of an array-based implementation present a problem in the context of a particular application? The answer to this question depends on two factors. The obvious factor is whether or not, for a given application, you can predict in advance the maximum number of items in the ADT at any one time. If you cannot, it is quite possible that an operation—and hence the entire program—will fail because the ADT in the context of a particular application requires more storage than the array can provide.

Arrays are easy to use, but they have a fixed size

Can you predict the maximum number of items in the ADT?

On the other hand, if, for a given application, you can predict in advance the maximum number of items in the ADT list at any one time, you must explore a more subtle factor: Would you waste storage by declaring an array to be large enough to accommodate this maximum number of items? Consider a case in which the maximum number of items is large, but you suspect that this number rarely will be reached. For example, suppose that your list could contain as many as 10,000 items, but the actual number of items in the list rarely exceeds 50. If you declare 10,000 array locations at compilation time, at least 9,950 array locations will be wasted most of the time. In both of the previous cases, the array-based implementation given in Chapter 4 is not desirable.

Will an array waste storage?

What if you used a resizeable array? Because you would use the *new* operator to allocate a larger array dynamically, you would be able to provide as much storage as the list needs (within the bounds of the particular computer, of course). Thus, you would not have to predict the maximum size of the list. However, if you doubled the size of the array each time you reached the end of the array—which is a reasonable approach to enlarging the array—you still might have many unused array locations. In the example just given, you could allocate an array of 50 locations initially. If you actually have 10,000 items in your list, array doubling will eventually give you an array of 12,800 locations, 2,800 more than you need. Remember also that you waste time by copying the array each time you need more space.

Increasing the size of a resizeable array can waste storage and time

Now suppose that your list will never contain more than 25 items. You could allocate enough storage in the array for the list and know that you would waste little storage when the list contained only a few items. With respect to its size, an array-based implementation is perfectly acceptable in this case.

An array-based implementation is a good choice for a small list

A reference-based implementation can solve any difficulties related to the fixed size of an array-based implementation. You use the *new* operator to allocate storage dynamically, so you do not need to predict the maximum size of the list. Because you allocate memory one item at a time, the list will be allocated only as much storage as it needs. Thus, you will not waste storage.

Linked lists do not have a fixed size

There are other differences between the array-based and reference-based implementations. These differences affect both the time and memory requirements of the implementations. Any time you store a collection of data in an array or a linked list, the data items become ordered; that is, there is a first item, a second item, and so on. This order implies that a typical item has a predecessor and a successor. In an array `anArray`, the location of the next item after the item in `anArray[i]` is *implicit*—it is in `anArray[i+1]`. In a linked list, however, you *explicitly* determine the location of the next item by using the reference in the current node. This notion of an implicit versus explicit next item is one of the primary differences between an array and a linked list. Therefore, an advantage of an array-based implementation is that it does not have to store explicit information about where to find the next data item, thus requiring less memory than a reference-based implementation.

The item after an array item is implied; in a linked list, an item explicitly references the next item

An array-based implementation requires less memory than a reference-based implementation

Another, more important advantage of an array-based implementation is that it can provide **direct access** to a specified item. For example, if you use the array `items` to implement the ADT list, you know that the item associated with list position *i* is stored in `items[i-1]`. Accessing either `items[0]` or `items[49]` takes the same amount of time. That is, the **access time** is constant for an array.

You can access array items directly with equal access time

On the other hand, if you use a linked list to implement the ADT list, you have no way of immediately accessing the node that contains the i^{th} item. To get to the appropriate node, you use the `next` data fields to traverse the linked list from its beginning until you reach the i^{th} node. That is, you access the first node and get the reference to the second node, access the second node and get the reference to the third node, and so on until you finally access the i^{th} node. Clearly, the time it takes you to access the first node is less than the time it takes to access the 50th node. The access time for the i^{th} node depends on *i*.

You must traverse a linked list to access its i^{th} node

The time to access the i^{th} node in a linked list depends on *i*

The type of implementation chosen will affect the efficiency of the operations of the ADT list. An array-based `get` is almost instantaneous, regardless of which list item you access. A reference-based retrieval operation like `get`, however, requires *i* steps to access the i^{th} item in the list.

You already know that the array-based implementation of the ADT list requires you to shift the data when you insert items into or delete items from the list. For example, if you delete the first item of a 20-item list, you must shift 19 items. In general, deleting the i^{th} item of a list of *n* items requires $n - i$ shifts. Thus, `remove` requires $n - 1$ shifts to delete the first item, but zero shifts to delete the last item. The list insertion operation `add` has similar requirements.

Insertion into and deletion from a linked list do not require you to shift data

In contrast, you do not need to shift the data when you insert items into or delete items from the linked list of a reference-based implementation. The methods `add` and `remove` require essentially the same effort, regardless of the length of the list or the position of the operation within the list, once you know the point of insertion or deletion. Finding this point, however, requires a list traversal, the time for which will vary depending on where in the list the operation will occur. Recall that the private method `find` performs this traversal. If you examine the definition of `find`, you will see that `find(i)` requires *i* assignment operations. Thus, `find`'s effort increases with *i*.

Insertion into and deletion from a linked list require a list traversal

We will continue to compare various solutions to a problem throughout this book. Chapter 10 will introduce a more formal way to discuss the efficiency of algorithms. Until then, our discussions will be informal.

Passing a Linked List to a Method

How can a method access a linked list? It is sufficient for the method to have access to the list's head reference. From this variable alone, the method can access the entire linked list. In the reference-based implementation of the ADT list that you saw earlier in this chapter, the head reference *head* to the linked list that contains the ADT's items is a private data field of the class *ListReferenceBased*. The methods of this class use *head* directly to manipulate the linked list.

A method with access to a linked list's ***head*** reference has access to the entire list

Would you ever want *head* to be an argument of a method? Certainly not for methods outside of the class, because such methods should not have access to the class's underlying data structure. Although on the surface, it would seem that you would never need to pass the head reference to a method, that is not the case. Recursive methods, for example, might need the head reference as an argument. You will see examples of such methods in the next section. Realize that these methods must not be public members of their class. If they were, clients could access the linked list directly, thereby violating the ADT's wall.

As Figure 5-19 illustrates, when *head* is an actual argument to a method, its value is copied into the corresponding formal parameter. The method then can access and alter the nodes in the list. However, the method cannot modify *head*'s value (recall our earlier example in Figure 5-4). This is fine for situations, such as a search method, that do not modify the list. But what should you do if you want to write a method that may need to modify the *head* reference? For example, a method that inserts a node at the beginning of the list will need to modify the *head* reference. One solution is for the return value of the method to be the new value for the *head* reference. Such an example will appear in the next section.

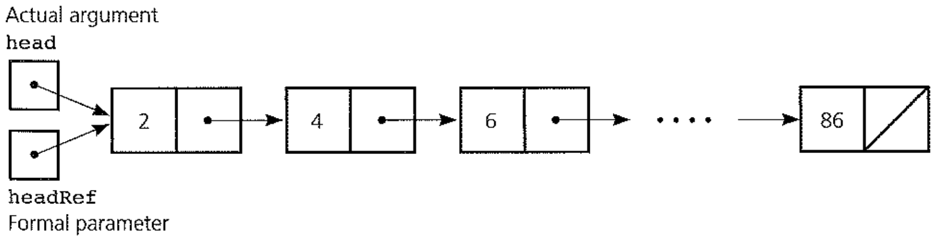


FIGURE 5-19

A head reference as an argument

Processing Linked Lists Recursively

It is possible, and sometimes desirable, to process linked lists recursively. This section examines recursive traversal and insertion operations on a linked list. If the recursive methods in this section are members of a class, they should not be public because they require the linked list's head reference as an argument.

Traversal. Suppose that you want to display the elements in a list referenced by *head*. That is, you want to write the objects in the order in which they appear in the linked list. The recursive strategy is simply

```
Write the first node of the list
Write the list minus its first node
```

The following Java method implements this strategy:

A recursive traversal method

```
private static void writeList(Node nextNode) {
// ---------------------------------------------------------
// Writes a list of objects.
// Precondition: The linked list is referenced by nextNode.
// Postcondition: The list is displayed. The linked list
// and nextNode are unchanged.
// ---------------------------------------------------------
  if (nextNode != null) {
    // write the first data object
    System.out.println(nextNode.getItem());
    // write the list minus its first node
    writeList(nextNode.getNext());
  }  // end if
}  // end writeList
```

This method is uncomplicated. It requires that you have direct access only to the first node of the list. The linked list provides this direct access because the list's first node, referenced by *head*, contains the list's first data item. Furthermore, you can easily pass the list minus its first node to *writeList*. If *head* references the beginning of the list, *head.getNext()* references the list minus its first node. You should compare *writeList* to the iterative technique that we used earlier in this chapter to display a linked list.

Compare the recursive ***writeList*** to the iterative technique on page 234

Now suppose that you want to display the list backward. Chapter 3 already developed two recursive strategies for writing a string *s* backward. Recall that the strategy of the method *writeBackward* is

writeBackward strategy

```
Write the last character of string s
Write string s minus its last character backward
```

The strategy of the method *writeBackward2* is

```
Write string s minus its first character backward
Write the first character of string s
```

writeBackward2 strategy

We can easily translate these strategies to linked lists. The method *writeBackward* translates to

```
Write the last node of the list
Write the list minus its last node backward
```

writeListBackward strategy

and the strategy of the method *writeBackward2* translates to

```
Write the list minus its first node backward
Write the first node of the list
```

writeListBackward2 strategy

You saw that these two strategies work equally well when an array is used. However, when a linked list is used, the first strategy is very difficult to implement: If `nextNode` references the node that contains the first node of the list, how do you get to the last node? Even if you had some way to get to the last node in the list quickly, it would be very difficult for you to move toward the front of the list at each recursive call. That is, it would be difficult for you to access the ends of the successively shorter lists that the recursive calls generate. (Later you will see a doubly linked list, which would solve this problem.)

This discussion illustrates one of the primary disadvantages of linked lists: Whereas an array provides direct access to any of its items, a linked list does not. Fortunately, however, the strategy of method *writeBackward2* requires that you have direct access only to the first character of the string. This access is the same that *writeListBackward2* requires: The list's head reference *nextNode* locates the first node in the list, and *nextNode.getNext()* references the list minus the first node.

writeListBackward2 is much easier to implement recursively than ***writeListBackward***

The following Java method implements the *writeListBackward2* strategy for a linked list:

```java
private static void writeListBackward2(Node nextNode) {
// ---------------------------------------------------
// Writes a list of objects backwards.
// Precondition: The linked list is referenced by
// nextNode.
// Postcondition: The list is displayed backwards. The
// linked list and nextNode are unchanged.
// ---------------------------------------------------
  if (nextNode != null) {
    // write the list minus its first node backward
    writeListBackward2(nextNode.getNext());
    // write the data object in the first node
    System.out.println(nextNode.getItem());
  } // end if
}   // end writeListBackward2
```

Self-Test Exercise 8 asks you to trace this method. This trace will be similar to the box trace in Figure 3-9. Exercise 5 asks you to write an iterative version of this method. Which version is more efficient?

Insertion. Now view the insertion of a node into a sorted linked list from a new perspective—that is, recursively. Later in this book you will need a recursive algorithm to perform an insertion into a linked structure. Interestingly, recursive insertion eliminates the need for both a trailing reference and a special case for inserting into the beginning of the list.

Consider the following recursive view of a sorted linked list: A linked list is sorted if its first data item is less than its second data item and the list that begins with the second data item is sorted. More formally, you can state this definition as follows:

The linked list that *head* references is a sorted linked list if

head is `null` (the empty list is a sorted linked list)

or

head.getNext() is *null* (a list with a single node is a sorted linked list)

or

head.getItem() < *head.getNext().getItem()*, and *head.getNext()* references a sorted linked list

You can base a recursive insertion on this definition. Notice that the following method inserts the node at one of the base cases—either when the list is empty or when the new data item is smaller than all the data items in the list. In both cases, you need to insert the new data item at the beginning of the list.

```
private static Node insertRecursive(Node headNode,
                        java.lang.Comparable newItem) {
  if ( (headNode == null) ||
       (newItem.compareTo(headNode.getItem()) < 0) ) {
    // base case: insert newItem at the beginning of the
    // linked list that nextNode references
    Node newNode = new Node(newItem, headNode);
    headNode = newNode;
  }
  else { //insert into rest of linked list
    Node nextNode = insertRecursive(headNode.getNext(),newItem);
    headNode.setNext(nextNode);
  } // end if
  return headNode;
} // end insertRecursive
```

First, consider the context for *insertRecursive*. Recall from Chapter 4 that the ADT operation *sortedAdd(newItem)* inserts *newItem* into its proper order in the sorted list. As a public method of the class, *sortedAdd* would call *insertRecursive* to do the insertion recursively. However, *insertRecursive* requires the linked list's head reference as an argument. Since the reference *head* is private and hidden from the client, you would not want *insertRecursive* to be an ADT operation. Thus, you would make it private.

To see how *insertRecursive* works, consider that *sortedAdd* will invoke *insertRecursive* by using the statement

```
head = insertRecursive(head, newItem);
```

Insertion occurs at the base case

Although *insertRecursive* does not maintain a trailing reference, inserting the new node is easy when the base case is reached. Note that within *insertRecursive*, *headNode* references the beginning of the sorted linked list. You use *headNode* to make the new node reference the first node in the original list and then change *headNode* so that it references the new node. Since *insertRecursive* returns *headNode*, *sortedAdd*'s assignment to *head* makes *head* reference the new node as required.

To understand the previous remarks, consider the case in which the new item is to be inserted at the beginning of the original list that has the external reference *head*. In this case, no recursive calls are made, and consequently when the base case is reached—that is, when *newItem.com-pareTo (headNode.getItem()) < 0*—the actual argument that corresponds to *headNode* is *head*, as Figure 5-20a

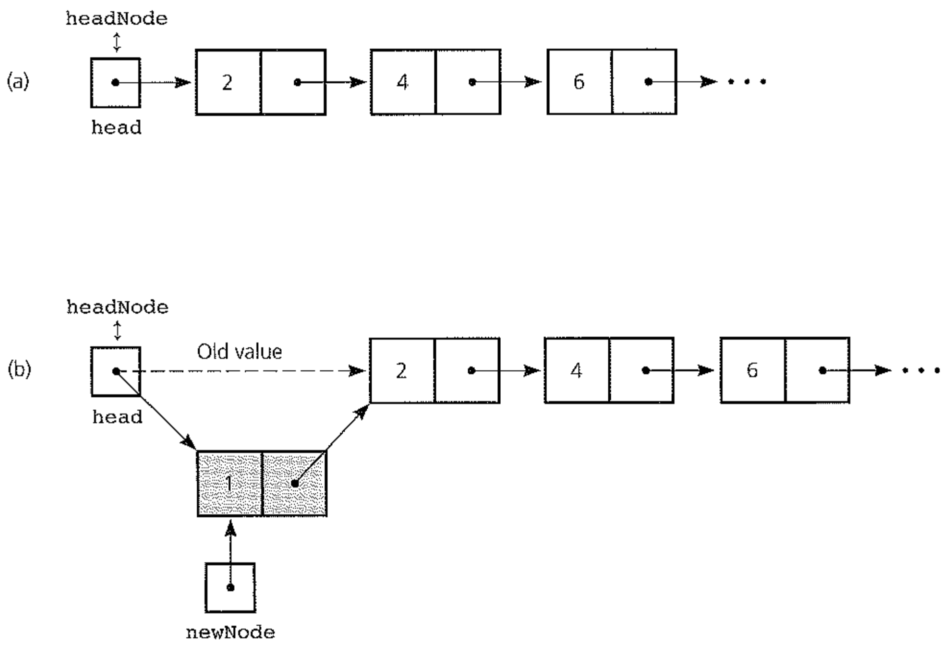


FIGURE 5-20

(a) A sorted linked list; (b) the assignment made for insertion at the beginning of the list illustrates. The assignment `headNode = newNode` then sets the method's return value to reference the new node. Upon return of `insertRecursive`, the statement

```
head = insertRecursive(head, newItem);
```

assigns the return value to `head`, as Figure 5-20b shows.

The general case in which the new item is inserted into the interior of the list that `head` references is similar. When `insertRecursive` is first called, the `else` clause of the `if` statement executes, making a recursive call to `insertRecursive`. When the base case is reached, what is the actual argument that corresponds to `headNode`? It is the `next` reference of the node that should precede the new node, as Figure 5-21 illustrates. Therefore, the base case returns a reference to the new node. The `else` clause assigns this reference to `nextNode`, and the call to `setNext` sets the next reference of the appropriate node to reference the new node.

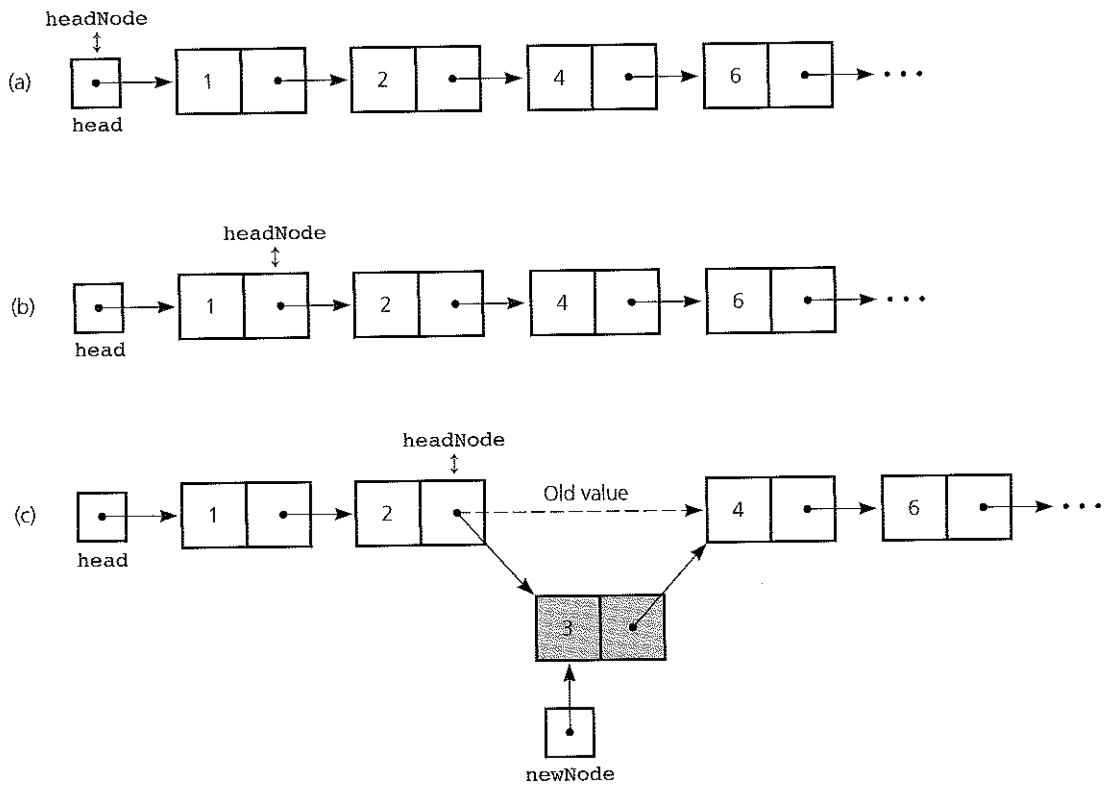


FIGURE 5-21

(a) The initial call `insertRecursive(head,newItem)`; (b) the first recursive call; (c) the second recursive call inserts at the beginning of the list that `headNode` references

When the original call to *insertRecursive* returns *headNode*, its value is unchanged from its original value of *head*. Thus, the assignment

```
head = insertRecursive(head, newItem);
```

leaves the value of *head* unchanged.

Although it could be argued that you should perform the operations on a sorted linked list recursively (after all, recursion does eliminate special cases and the need for a trailing reference), the primary purpose in presenting the recursive *insertRecursive* is to prepare you for the binary search tree algorithms to be presented in Chapter 11.

5.3 Variations of the Linked List

This section briefly introduces several variations of the linked list that you have just seen. These variations are often useful, and you will encounter them later in this text. Many of the implementation details are left as exercises. Note that in addition to the data structures discussed in this section, it is possible to have other data structures, such as arrays of references to linked lists and linked lists of linked lists. These data structures are also left as exercises.

Tail References

In many situations, you simply want to add an item to the end of a list. For example, maintaining a list of requests for a popular book at the local library would require that new requests for the book be placed at the end of a waiting list. You could use an ADT list called *waitingList* as follows:

```
waitingList.add(request, waitingList.size()+1);
```

This statement adds *request* to the end of *waitingList*. Recall that in implementing *add* to insert an item at the position indicated, we used the method *find* to traverse the list to that position. Note that this statement actually performs these four steps:

1. Allocate a new node for the linked list.
2. Set the reference in the last node in the list to reference the new node.
3. Put the new *request* in the new node.
4. Set the reference in the new node to *null*.

Each time you add a new request, you must get to the last node in the linked list. One way to accomplish this is to traverse the list each time you add a new request. A much more efficient method uses a **tail reference** *tail* to remember where the end of the linked list is—just as *head* remembers where the beginning of the list is. Like *head*, *tail* is external to the list. Figure 5-22 illustrates a linked list of integers that has both *head* and *tail* references.

Use a ***tail*** reference to facilitate adding nodes to the end of a linked list

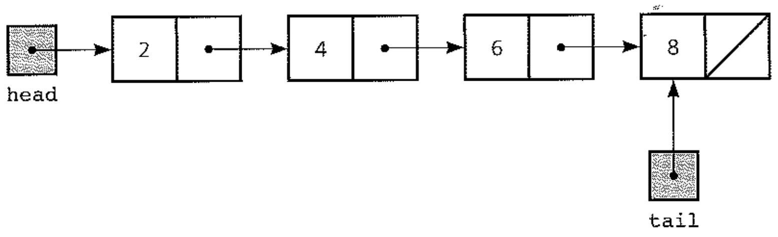


FIGURE 5-22

A linked list with ***head*** and ***tail*** references

With *tail* pointing to the end of the linked list, you can perform Steps 1 through 4 by using the single statement

```
tail.setNext(new Node(request, null));
```

This statement sets the *next* reference in the last node in the list to point to a newly allocated node. You then update *tail* so that it references the new last node by writing *tail = tail.getNext();*. You thus have an easy method for adding a new item to the end of the list. Initially, however, when you insert the first item into an empty linked list, *tail*—like *head*—is *null*. We leave the details of a solution as an exercise.

Treat the first insertion as a special case

Circular Linked Lists

When you use a computer that is part of a network, you share the services of another computer—called a **server**—with many other users. A similar sharing of resources occurs when you access a central computer by using a remote terminal. The system must organize the users so that only one user at a time has access to the shared computer. By ordering the users, the system can give each user a turn. Because users regularly enter and exit the system (by logging on or logging off), a linked list of user names allows the system to maintain order without shifting names when it makes insertions to and deletions from the list. Thus, the system can traverse the linked list from the beginning and give each user on the list a turn on the shared computer. What must the system do when it reaches the end of the list? It must return to the beginning of the list. However, the fact that the last node of a linked list does not reference another node can be an inconvenience.

If you want to access the first node of a linked list after accessing the last node, you must resort to the head reference. Suppose that you change the *next* portion of the list's last node so that, instead of containing *null*, it references the first node. The result is a **circular linked list,** as illustrated in Figure 5-23. In contrast, the linked list you saw earlier is said to be a **linear linked list.**

Every node in a circular linked list has a successor

Every node in a circular linked list references a successor, so you can start at any node and traverse the entire list. Although you could think of a circular

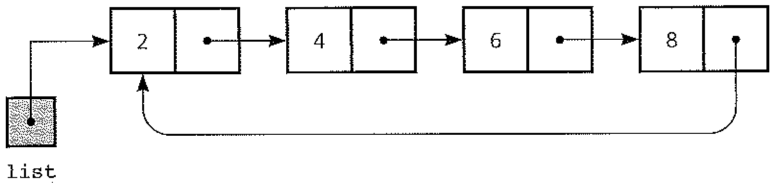


FIGURE 5-23

A circular linked list

list as not having either a beginning or an end, you still would have an external reference to one of the nodes in the list. Thus, it remains natural to think of both a first and a last node in a circular list. If the external reference locates the "first" node, you still would have to traverse the list to get to the last node. However, if the external reference—call it *list*—references the "last" node, as it does in Figure 5-24, you can access both the first and last nodes without a traversal, because *list.getNext()* references the first node.

A *null* value in the external reference indicates an empty list, as it did for a linear list. However, no node in a circular list contains *null* in its *next* reference. Thus, you must alter the algorithm for detecting when you have traversed an entire list. By simply comparing the current reference *curr* to the external reference *list*, you can determine when you have traversed the entire circular list. For example, the following Java statements display the data portions of every node in a circular list, assuming that *list* references the "last" node:

No node in a circular linked list contains ***null***

Write the data in a circular linked list

```
// display the data in a circular linked list;
// list references its last node
if (list != null) {
  // list is not empty
  Node first = list.getNext(); // reference first node

  Node curr = first;        // start at first node
```

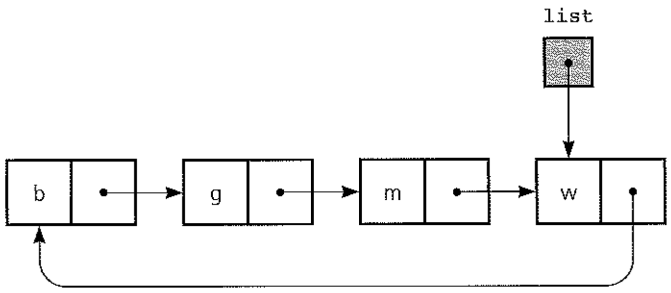


FIGURE 5-24

A circular linked list with an external reference to the last node

```
    // Loop invariant: curr references next node to
    // display
    do {
      // write data portion
      System.out.println(curr.getItem());
      curr = curr.getNext();      // reference next node
    } while (curr != first);      // list traversed?
  }  // end if
```

Operations such as insertion into and deletion from a circular linked list are left as exercises.

Dummy Head Nodes

Both the insertion and deletion algorithms presented earlier for linear linked lists require a special case to handle action at the first position of a list. Many people prefer a method that eliminates the need for the special case. One such method is to add a **dummy head node**—as Figure 5-25 depicts—that is always present, even when the linked list is empty. In this way, the item at the first position of the list is actually in the second node. Also, the insertion and deletion algorithms initialize *prev* to reference the dummy head node, rather than *null*. Thus, for example, in the deletion algorithm, the statement

```
prev.setNext(curr.getNext());
```

deletes from the list the node that *curr* references, regardless of whether or not this node is the first element in the list.

Despite the fact that a dummy head node eliminates the need for a special case, handling the first list position separately can, in general, be less distracting than altering the list's structure by adding a dummy head node. However, dummy head nodes are useful with doubly linked lists, as you will see in the next section.

Doubly Linked Lists

Suppose that you wanted to delete a particular node from a linked list. If you were able to locate the node directly without a traversal, you would not have established a trailing reference to the node that precedes it in the list. Without

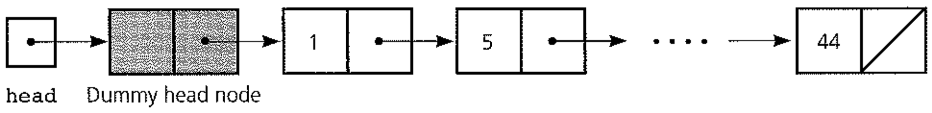


FIGURE 5-25

A dummy head node

a trailing reference, you would be unable to delete the node. You could overcome this problem if you had a way to back up from the node that you wished to delete to the node that precedes it. A **doubly linked list** solves this problem because each of its nodes has references to both the next node and the previous node.

Each node in a doubly linked list references both its predecessor and its successor

Consider a sorted linked list of customer names such that each node contains, in addition to its data field, two reference variables, *precede* and *next*. As usual, the *next* reference of node *N* references the node that follows *N* in the list. The *precede* data field references the node that precedes *N* in the list. Figure 5-26 shows the form of this sorted linked list of customers.

Notice that if *curr* references a node *N*, you can get a reference to the node that precedes *N* in the list by using the assignment statement

```
prev = curr.getPrecede();
```

A doubly linked list thus allows you to delete a node without traversing the list to establish a trailing reference.

Because there are more references to set, the mechanics of inserting into and deleting from a doubly linked list are a bit more involved than for a singly linked list. In addition, the special cases at the beginning or the end of the list are more complicated. It is common to eliminate the special cases by using a dummy head node. Although dummy head nodes may not be worthwhile for singly linked lists, the more complicated special cases for doubly linked lists make them very attractive.

Dummy head nodes are useful in doubly linked lists

As Figure 5-27a shows, the external reference *listHead* always references the dummy head node. Notice that the dummy head node has the same data type as the other nodes in the list; thus it also contains *precede* and *next* references. You can link the list so that it becomes a **circular doubly linked list.** The *next* reference of the dummy head node then references the first "real node"—for example, the first customer name—in the list, and the *precede* reference of the first real node refers back to the dummy head node. Similarly, the *precede* reference of the dummy head node references the last node in the list, and the *next* reference of the last node references the dummy head node. Note that the dummy head node is present even when the list is empty. In this

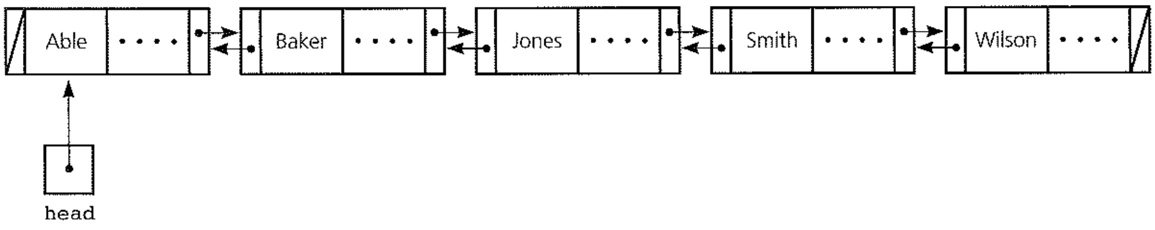


FIGURE 5-26

A doubly linked list

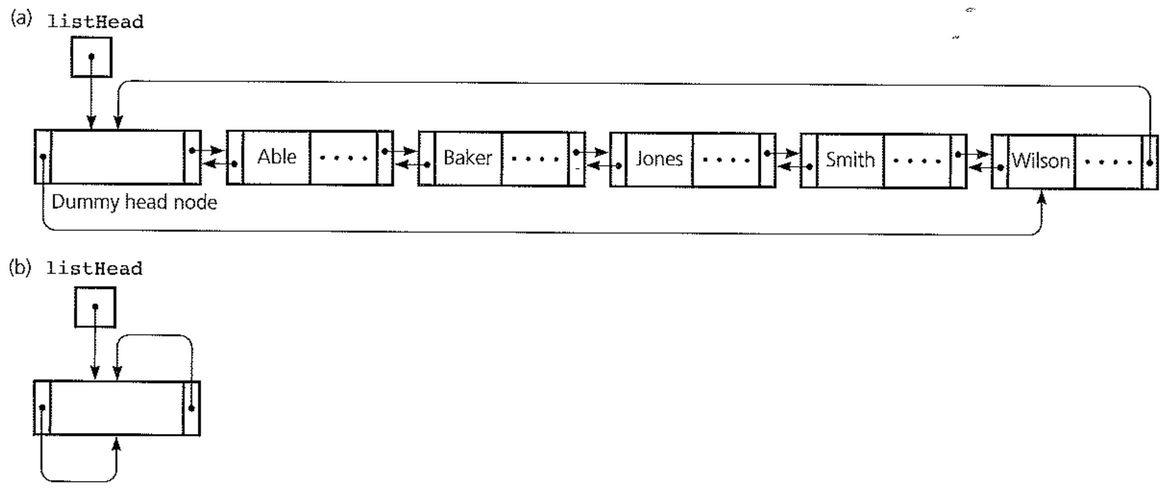


FIGURE 5-27

(a) A circular doubly linked list with a dummy head node; (b) an empty list with a dummy head node

case, both reference variables of the dummy head node reference the head node itself, as Figure 5-27b illustrates.

A circular doubly linked list eliminates special cases for insertion and deletion

By using a circular doubly linked list, you can perform insertions and deletions without special cases: Inserting into and deleting from the first or last position is the same as for any other position. Consider, for example, how to delete the node *N* that *curr* references. As Figure 5-28 illustrates, you need to

1. Change the *next* reference of the node that precedes *N* so that it references the node that follows *N*.
2. Change the *precede* reference of the node that follows *N* so that it references the node that precedes *N*.

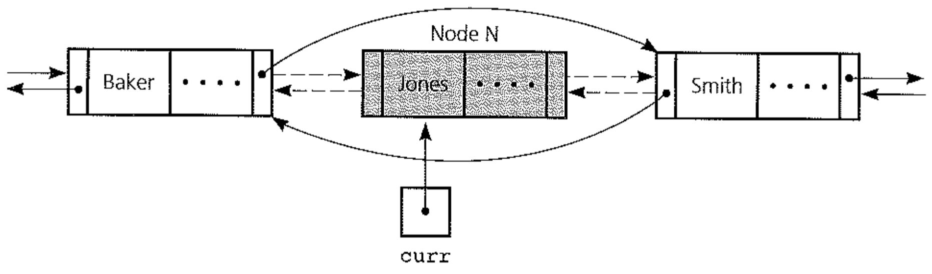


FIGURE 5-28

Reference changes for deletion

The following Java statements accomplish these two steps:

Deleting a node

```
// delete the node that curr references
curr.getPrecede().setNext(curr.getNext());
curr.getNext().setPrecede(curr.getPrecede());
```

You should convince yourself that these statements work even when the node to be deleted is the first, last, or only data (nonhead) node in the list.

Now consider how to insert a node into a circular doubly linked list. In general, the fact that the list is doubly linked does not mean that you avoid traversing the list to find the proper place for the new item. For example, if you insert a new customer name, you must find the proper place within the sorted linked list for the new node. The following pseudocode sets *curr* to reference the node that contains the first name greater than *newName*. Thus, *curr* will reference the node that is to follow the new node on the list:

Traverse the list to locate the insertion point

```
// find the insertion point
curr =  listHead.getNext() // reference first node, if any
while (curr != listHead and newName > curr.getItem()) {
  curr.getNext()
} // end while
```

Notice that if you want to insert the new node either at the end of the list or into an empty list, the loop will set *curr* to reference the dummy head node.

As Figure 5-29 illustrates, once *curr* references the node that is to follow the new node, you need to

1. Set the *next* reference in the new node to reference the node that is to follow it.
2. Set the *precede* reference in the new node to reference the node that is to precede it.

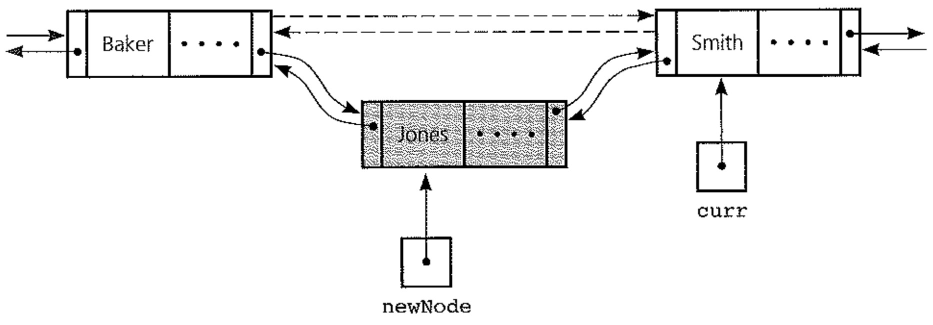


FIGURE 5-29

Reference changes for insertion

3. Set the *precede* reference in the node that is to follow the new node so that it references the new node.
4. Set the *next* reference in the node that is to precede the new node so that it references the new node.

The following Java statements accomplish these four steps, assuming that *newNode* references the new node:

Inserting a node

```
// insert the new node that newNode references before
// the node referenced by curr
newNode.setNext(curr);
newNode.setPrecede(curr.getPrecede());
curr.setPrecede(newNode);
newNode.getPrecede().setNext(newNode);
```

You should convince yourself that these statements work even when you insert the node into the beginning of a list; at the end of a list, in which case *curr* references the head node; or into an empty list, in which case *curr* also references the head node.

5.4 Application: Maintaining an Inventory

Imagine that you have a part-time job at the local movie rental store. Realizing that you know a good deal about computers, the store owner asks you to write an interactive program that will maintain the store's inventory of DVDs that are for sale. The inventory consists of a list of movie titles and the following information associated with each title:

- **Have value:** number of DVDs currently in stock.
- **Want value:** number of DVDs that should be in stock. (When the have value is less than the want value, more DVDs are ordered.)
- **Wait list:** list of names of people waiting for the title if it is sold out.

Because the owner plans to turn off the power to the computer when the store is closed, your inventory program will not be running at all times. Therefore, the program must save the inventory in a file before execution terminates and later restore the inventory when it is run again.

Program input and output are as follows:

Input

- A file that contains a previously saved inventory.
- A file that contains information on an incoming shipment of DVDs. (See command D.)
- Single-letter commands—with arguments where necessary—that inquire about or modify the inventory and that the user will enter interactively.

Output

- A file that contains the updated inventory. (Note that you remove from the inventory all items whose have values and want values are zero and whose wait lists are empty. Thus, such items do not appear in the file.)
- Output as specified by the individual commands.

The program should be able to execute the following commands:

Program commands

H	(help)	Provide a summary of the available commands.
I <*title*>	(inquire)	Display the inventory information for a specified title.
L	(list)	List the entire inventory (in alphabetical order by title).
A <*title*>	(add)	Add a new title to the inventory. Prompt for initial want value.
M <*title*>	(modify)	Modify the want value for a specified title.
D	(delivery)	Take delivery of a shipment of DVDs, assuming that the clerk has entered the shipment information (titles and counts) into a file. Read the file, reserve DVDs for the people on the wait list, and update the have values in the inventory accordingly. Note that the program must add an item to the inventory if a delivered title is not present in the current inventory.
O	(order)	Write a purchase order for additional DVDs based on a comparison of the have and want values in the inventory, so that the have value is brought up to the want value.
R	(return)	Write a return order based on a comparison of the have and want values in the inventory and decrease the have values accordingly (make the return). The purpose is to reduce the have value to the want value.
S <*title*>	(sell)	Decrease the count for the specified title by 1. If the title is sold out, put a name on the wait list for the title.
Q	(quit)	Save the inventory and wait lists in a file and terminate execution.

The problem-solving process that starts with a statement of the problem and ends with a program that effectively solves the problem—that is, a program that meets its specification—has three main stages:

1. The design of a solution
2. The implementation of the solution
3. The final set of refinements to the program

Realize, however, that you cannot complete one stage in total isolation from the others. Also realize that at many steps in the development of a solution, you must make choices. Although the following discussion may give the impression that the choices are clear-cut, this is not always the case. In reality, both the trade-offs between choices and the false starts (wrong choices considered) are often numerous.

This problem primarily involves data management and requires certain program commands. These commands suggest the following operations on the inventory:

Operations on the inventory

- List the inventory in alphabetical order by title (L command).
- Find the inventory item associated with a title (I, M, D, O, and S commands).
- Replace the inventory item associated with a title (M, D, R, and S commands).
- Insert new inventory items (A and D commands).

Recall that each title might have an associated wait list of people who are waiting for that title. You must be able to

- Add new people to the end of the wait list when they want a DVD that is sold out (S command).
- Delete people from the beginning of the wait list when new DVDs are delivered (D command).
- Display the names on a wait list for a particular title (I and L commands).

In addition, you must be able to

- Save the current inventory and associated wait lists when program execution terminates (Q command).
- Restore the current inventory and associated wait lists when program execution begins again.

You could think of these operations as part of an ADT inventory. Your next step should be to specify each of the operations fully. Since this chapter is about linked lists and implementation issues, the completion of the specifications will be left as an exercise. We will turn our attention to a data structure that could implement the inventory.

Each data item in the ADT inventory represents a movie and contains a title, the number of DVDs in stock (a have value), the number desired (a want value), and a wait list. How will you represent the wait list? First, you need to decide what information you want to store in the wait list. For example, you might want to keep track of the full name and phone number of each person, and create a class `Customer` containing data fields for the first and last names along with the telephone number of the person. This class might be structured as follows:

A customer in the wait list

```
public class Customer {
  private String lastName;
  private String firstName;
  private String phone;

  public Customer(String first, String last, String phone) {
    // to be implemented
    . . .
  } // end constructor

  public String toString() {
    // to be implemented
    . . .
  } // end toString
} // end class Customer
```

This definition contains the minimum number of methods required to use instances of the *Customer* class in our inventory problem. You may also decide that you want to keep additional information about a person, such as their address. The *toString* method is provided for printing purposes.

Now that you have decided what information to keep in the wait list, how will you implement the wait list itself in the ADT inventory? Could you use any of the implementations of the ADT list we developed previously? To make this decision, you must review the requirements of the wait list as stated in the inventory problem and then see if the ADT list will be able to meet these requirements. The inventory problem requires you to be able to add to the end of the wait list and delete from the beginning of the wait list. Clearly, removing an item from the beginning of the list is easy: You can simply use the ADT list operation *remove* with an index value of 1. Adding an item to the end of the list is also fairly easy. You know the size of the list from the ADT list operation *size()*, and you could use the ADT list operation *add* with an index value of *size()* + 1 to place an item at the end of the wait list.

The ADT list is not the best choice for data that must be in alphabetical order

One of the requirements of the inventory problem is that the L command list the inventory in alphabetical order by movie title. Will you be able to use the ADT list in a way that will support this requirement? Not easily, since the ADT list is based on index position, not on a sorted order. A better choice is the ADT sorted list. Not only does it maintain the data in a sorted order for us, but it also provides an operation *locateIndex(item)* that can be used to search for an item in the sorted list.

If the ADT sorted list is not yet implemented, should you use an array-based implementation or a reference-based implementation? If you use an array to contain the items, you can use a binary search. Inserting and deleting items, however, requires you to shift array elements. Using a linked list for the items avoids these data shifts but makes a binary search impractical. (How do you quickly locate the middle item in a linked list?) Weighing these trade-offs, we choose a linked list to implement the ADT sorted list.

To summarize, we have made the following choices:

A sorted list represents the inventory

- The inventory is a sorted list of data items (the ADT sorted list implemented as a linked list of data items), sorted by the title that each item represents.
- Each inventory item contains a title, a have value, a want value, and a list of customers (the wait list).

Figure 5-30 and the following Java statements summarize these choices:

```
public class StockItem implements java.lang.Comparable {
  public String title;
  private int have, want;
  private ListReferenceBased waitingList;

  // various constructors for StockItem
  . . .
```

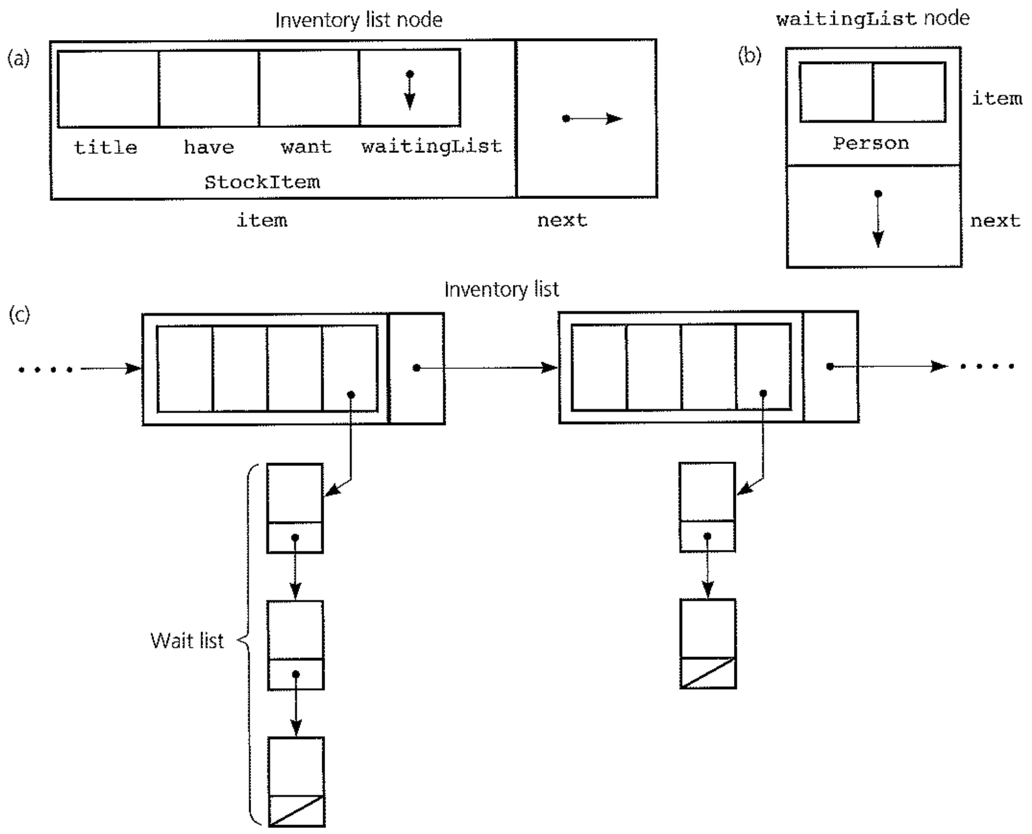


FIGURE 5-30

(a) Inventory list node; (b) wait list node; (c) orthogonal structure for the inventory

```
  public void addToWaitingList(String lastName,
                                String firstName, String phone) {
  // add a person to the waiting list
    waitingList.addSorted(
                  new Customer(lastName, firstName, phone);
  } // end addToWaitingList

  public String toString() {
    // for displaying StockItem instances
    . . .
  } // end toString

  public int compareTo(Object rhs) {
  // define how StockItems are compared
     return title.compareTo(((StockItem)rhs).title);
  } // end compareTo

// mutator and accessor methods for other data fields
  . . .

}  // end class StockItem
```

You declare the inventory as follows:

```
SortedList inventory = new SortedList();
```

Before you can proceed with the implementation, you must consider how you will save the inventory in a file. Java provides **object serialization,** a process that transforms an object into a stream of bytes that you can save and restore from a file. The most powerful aspect of object serialization is that when you write any object to a file, any other objects that are referenced by that object are also written to the file. As mentioned in Chapter 1, to enable this feature, you place an *implements Serializable* clause in each class that has instances that will be written to the file. Thus, to write the inventory successfully, you would include the clause in the classes *ListReferenceBased*, *Node*, *SortedList*, *StockItem*, and *Customer*. Then, when you write an inventory object to a file, all of the stock items in the inventory list and their wait lists are also be placed in the file. Here is the code that accomplishes that task:

```
try {
  FileOutputStream fos = new
                      FileOutputStream("inventory.dat");
  ObjectOutputStream oos = new ObjectOutputStream(fos);
  oos.writeObject(inventory);
  fos.close();
} // end try
catch (Exception e) {
    System.out.println(e);
  } // end catch
```

Restoring the inventory is also straightforward:

```
ListReferenceBased restoredInventory;
try {
  FileInputStream fis = new
                        FileInputStream("inventory.dat");
  ObjectInputStream ois = new ObjectInputStream(fis);
  Object o = ois.readObject();
  restoredInventory = (ListReferencedBased) o;
  System.out.println(restoredInventory);
} // end try
catch (Exception e) {
  System.out.println(e);
} // end catch
```

The completion of this solution is left as an exercise.

5.5 The Java Collections Framework

The Java Collections Framework (JCF) provides classes for common ADTs

Many modern programming languages, such as Java, provide classes that implement many of the more commonly used ADTs. In Java, many of these classes are defined in the Java Collections Framework or JCF. The JCF contains a number of classes and interfaces that can be applied to nearly any type of data.

Many of the ADTs that are presented in this text have a corresponding class or interface in the JCF. For example, a `List` interface is defined in the JCF that is similar to the `ListInterface` specification presented earlier in this chapter. You may be wondering why we spend so much time developing ADTs in this text if they are already provided in the JCF. There are many reasons for doing so; here are just a few:

- Developing simple ADTs provides a foundation for learning other ADTs.
- You may find yourself working in a language that does not provide any predefined ADTs. You need to have the ability to develop ADTs on your own, and hence understand the process.
- If the ADTs defined by the language you are using are not sufficient, you may need to develop your own or enhance existing ones.

A collections framework includes interfaces, implementations, and algorithms

A collections framework is a unified architecture for representing and manipulating collections. It includes *interfaces*, or ADTs representing collections; *implementations*, or concrete implementations of collection interfaces; and *algorithms*, or methods that perform useful computations, such as sorting and searching, on objects that implement collection interfaces. These algorithms are *polymorphic* because the same method can be used on many different implementations of the appropriate collections interface.

The JCF also contains iterators. Iterators provide a way to cycle through the contents of a collection. Before we can discuss the JCF further, we will give a brief overview of generics and iterators.

Generics

The JCF relies heavily on Java generics. Generics allow you to develop classes and interfaces and defer certain data-type information until you are actually ready to use the class or interface. For example, our list interface was developed independently from the type of the list items by using the `Object` class. With generics, this data type is left as a data-type parameter in the definition of the class or interface. The start of the definition of the class or interface is followed by <*E*>, where the data-type parameter *E* represents the data type that the client code will specify. Here is an example of a simple generic class:

Generic classes allow data-type information to be deferred

```
public class MyClass<E> {
  private E theData;
  private int n;

  public MyClass() {
    n = 0;
  } // end constructor

  public MyClass(E initData) {
    n = 0;
    theData = initData;
  } // end constructor

  public void setData(E newData) {
    theData = newData;
  } // end setData

  public E getData() {
    return theData;
  }
} // end MyClass
```

Chapter 9 describes in more detail how to create your own generics.

When you (the client) declare instances of the class, you specify the actual data type that the parameter represents. This data type cannot be a primitive type, only object types are allowed. For example, a simple program that uses this generic class could begin as follows:

Only object types are allowed for data type parameters

```
static public void main(String[] args) {
  MyClass<String> a = new MyClass<String>();
  Double d = new Double(6.4);
  MyClass<Double> b = new MyClass<Double>(d);
```

```
a.setData("Sarah");
System.out.println(a.getData() + ", " + b.getDãta());
```

Notice how the declarations of *a* and *b* specify the data type of *MyClass*'s data member *theData*. Also note that when we previously used *Object* as the return type, we often had to cast the result back to the desired type that is no longer required when using generics.

Iterators

Iterators are used to cycle through items in a collection

An **iterator** is an object that gives you the ability to cycle through the items in a collection in much the same way that we used a reference to traverse a linked list. If you have an iterator call *iter*, you can access the next item in the collection by suing the notation *iter.next()*.

The JFC provides two primary iterator interfaces, *java.util.Iterator* and *java.util.ListIterator*. The *Iterator* interface is defined as follows:

```
public interface Iterator<E> {
  public boolean hasNext();
       // Returns true if the iteration has more elements.

  public E next();
       // Returns the next element in the iteration.

  public void remove()
             throws UnsupportedOperationException,
                    IllegalStateException;
       // Removes from the underlying collection the last
       // element returned by the iterator (optional
       // operation).
} // end Iterator
```

The method next is used to return the next element in the collection. When an iterator is initially created, it is positioned so that the first call to *next* on the iterator object will return the initial element in the collection. The method *hasNext* can be used to determine if another element is available in the collection.

Unsupported iterator methods will throw an exception

Notice that one of the operations, *remove*, can throw the exception *UnsupportedOperationException*. The expectation is that the *remove* operation will simply throw this exception if the operation is not available in the class that implements the interface.

Iterators are an integral part of all of the classes and interfaces used for representing collections in the JCF. Note that just as you can use inheritance to derive new classes, you can use inheritance to derive new interfaces, often called **subinterfaces.** The basis for the ADT collections in the JCF is the interface *java.util.Iterable*, with the subinterface *java.util.Collection*:

```
public interface Iterable<E> {
  Iterator<E> iterator();
  // Returns an iterator over the elements in this collection
}
public interface Collection<E> extends Iterable<E> {
  // Only a portion of the Collection interface is shown here.
  // See the J2SE documentation for a complete listing of
  // methods

  boolean add(E o);
    // Ensures that this collection contains the specified
    // element (optional operation).

  boolean remove(Object o);
    // Removes a single instance of the specified element from
    // this collection, if it is present (optional operation).

  void clear();
    // Removes all of the elements from this collection
    // (optional operation).

  boolean contains(Object o);
    // Returns true if this collection contains the specified
    // element.

  boolean equals(Object o);
    // Compares the specified object with this collection for
    // equality.

  boolean isEmpty()
    // Returns true if this collection contains no elements.

  int size();
    // Returns the number of elements in this collection.

  Object[] toArray();
    //Returns an array containing all of the elements in this
    // collection.
} // end Collection
```

Thus, every ADT collection in the JCF will have a method to return an iterator object for the underlying collection. The following example shows how an iterator can be used with the JCF list class `LinkedList`:

```
import java.util.LinkedList;
import java.util.Iterator;
```

```
public class TestLinkedList {
  static public void main(String[] args) {
    LinkedList<Integer> myList = new LinkedList<Integer>();

    Iterator iter = myList.iterator();
    if (!iter.hasNext()) {
      System.out.println("The list is empty");
    } // end if

    for (int i=1; i <= 5; i++) {
      myList.add(new Integer(i));
    } // end for

     iter = myList.iterator();
    while (iter.hasNext()) {
      System.out.println(iter.next());
    } // end while
  } // end main
} // end TestLinkedList
```

The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling the *remove* method.

Another example of a subinterface in the JCF is *java.util.ListIterator*, derived from the *java.util.Iterator interface*:

```
public interface ListIterator<E> extends Iterator<E> {

  void add(E o);
    // Inserts the specified element into the list (optional
    // operation).

  boolean hasNext();
    // Returns true if this list iterator has more elements when
    // traversing the list in the forward direction.

  boolean hasPrevious();
    // Returns true if this list iterator has more elements when
    // traversing the list in the reverse direction.

  E next();
    // Returns the next element in the list.

  int nextIndex();
    // Returns the index of the element that would be returned
    // by a subsequent call to next.
  E previous();
    // Returns the previous element in the list.

  int previousIndex();
    // Returns the index of the element that would be returned
    // by a subsequent call to previous.

  void remove();
    // Removes from the list the last element that was
    // returned by next or previous (optional operation).

  void set(E o);
    // Replaces the last element returned by next or previous
    // with the specified element (optional operation).
} // end ListIterator
```

The *ListIterator* interface extends *Iterator* by providing support for **bidirectional** access to the collection as well as adding or changing elements in the collection. A bidirectional iterator enables you to move to either the next or previous element in the collection.

Bidirectional iterators allow you to move forward or back through a collection

Chapter 9 describes how to create your own iterator.

The Java Collections Framework *List* Interface

The JCF provides an interface *java.util.List* that is quite similar to the list interface created in Chapter 4. The JCF *List* interface supports an ordered collection, also known as a sequence. Like the *ListInterface* presented in this text, users can specify by position (integer index) where elements are added to and removed from the list, though the position numbering starts at zero (not one as in *ListInterface*). Though the interface provides methods based upon positional access to the elements, the time to execute these methods may be proportional to the index value, depending on the implementing class. As such, it is usually preferable to use an iterator instead of index access when possible to locate and process elements in a list.

The JCF ***List*** interface supports an ordered collection

Declarations for all the methods in the *List* interface are shown here, even the ones inherited from the *Collection* interface. Notice that the methods *iterator*, *add*, *remove*, and *equals* place additional stipulations beyond those specified in the *Collection* interface. The *List* interface also provides other methods not shown here that allow multiple elements to be inserted and removed at any point in the list.

The ***List*** interface inherits methods from the ***Collection*** interface

Notice that the *List* interface provides a *ListIterator* that allows bidirectional access in addition to the normal operations that the *Iterator* interface provides. There is also a method to obtain a list iterator that starts at a specified position in the list.

The JCF *List* interface is derived from the JCF *Collection* interface:

```
public interface List<E> extends Collection<E>
  // Only a portion of the List interface is shown here.
  // See the J2SE documentation for a complete listing of
  // methods

  boolean add(E o);
    // Appends the specified element to the end of this list
    // (optional operation).

  void add(int index, E element);
    // Inserts the specified element at the specified position in
    // this list (optional operation).

  void clear();
    // Removes all of the elements from this list (optional
    // operation).

  boolean contains(Object o);
    // Returns true if this list contains the specified element.

  boolean equals(Object o);
    // Compares the specified object with this list for equality.

  E get(int index);
    // Returns the element at the specified position in this
    // list.

  int indexOf(Object o);
    // Returns the index in this list of the first occurrence of
    // the specified element, or -1 if this list does not contain
    // this element.

  boolean isEmpty();
    // Returns true if this list contains no elements.

  Iterator<E> iterator();
    // Returns an iterator over the elements in this list in
    // proper sequence.

  ListIterator<E> listIterator();
    // Returns a list iterator of the elements in this list (in
    // proper sequence).

  ListIterator<E> listIterator(int index);
    // Returns a list iterator of the elements in this list (in
    // proper sequence), starting at the specified position in
    // this list.
  E remove(int index);
    // Removes the element at the specified position in this list
    // (optional operation).

  boolean remove(Object o);
    // Removes the first occurrence in this list of the specified
    // element (optional operation).

  E set(int index, E element);
    // Replaces the element at the specified position in this
    // list with the specified element (optional operation).

  int size();
    // Returns the number of elements in this list.

  List<E> subList(int fromIndex, int toIndex);
    // Returns a view of the portion of this list between the
    // specified fromIndex, inclusive, and toIndex,
    // exclusive.

  Object[] toArray();
    // Returns an array containing all of the elements in this
    // list in proper sequence.
} // end List
```

The JCF provides numerous classes that implement the *List* interface, including *LinkedList*, *ArrayList*, and *Vector*. Here is an example of how the JCF class *ArrayList* is used to maintain a grocery list:

```
import java.util.ArrayList;
import java.util.Iterator;

public class GroceryList {

  static public void main(String[] args) {
    ArrayList<String> groceryList = new ArrayList<String>();
    Iterator<String> iter;

    groceryList.add("apples");
    groceryList.add("bread");
    groceryList.add("juice");
    groceryList.add("carrots");
    groceryList.add("ice cream");

    System.out.println("Number of items on my grocery list: "
                       + groceryList.size());
    System.out.println("Items are: ");
    iter = groceryList.listIterator();
    while (iter.hasNext()) {
      String nextItem = iter.next();
      System.out.println(groceryList.indexOf(nextItem)+") "
                               + nextItem);
    } // end while

  } // end main

} // end GroceryList
```

The output of this program is

```
Number of items on my grocery list: 5
Items are:
0) apples
1) bread
2) juice
3) carrots
4) ice cream
```

Clearly it is more efficient to use a counter to number the items than to use the method *indexOf*; it was done for illustrative purposes.

Summary

1. You can use reference variables to implement the data structure known as a linked list by using a class definition such as the following:

   ```
   public class Node {
     private Object item;
     private Node next;
     // constructors, accessors, and mutators
     ...
   } // end class Node
   ```

2. Each reference in a linked list is a reference to the next node in the list. For example, if **nodeRef** is a variable of type ***Node*** that references a node in this linked list,
   - *nodeRef.getItem()* is the data portion of the node.
   - *nodeRef.getNext()* references the next node.

3. Algorithms for inserting data into and deleting data from a linked list both involve these steps: Traverse the list from the beginning until you reach the appropriate position; perform reference changes to alter the structure of the list. In addition, you use the ***new*** operator to dynamically allocate a new node for insertion. When all references to a node are removed, the node is automatically marked for garbage collection.

4. Inserting a new node at the beginning of a linked list or deleting the first node of a linked list are cases that you treat differently from insertions and deletions anywhere else in the list.

5. An array-based implementation uses an implicit ordering scheme—for example, the item that follows `anArray[i]` is stored in `anArray[i+1]`. A reference-based implementation uses an explicit ordering scheme—for example, to find the item that follows the one in node *N*, you follow node *N*'s reference.

6. You can access any element of an array directly, but you must traverse a linked list to access a particular node. Therefore, the access time for an array is constant, whereas the access time for a linked list depends upon the location of the node within the list.

7. You can insert items into and delete items from a reference-based linked list without shifting data. This characteristic is an important advantage of a linked list over an array.

8. Although you can use the `new` operator to allocate memory dynamically for either an array or a linked list, you can increase the size of a linked list one node at a time more efficiently than an array. When you increase the size of a resizeable array, you must copy the original array elements into the new array and then deallocate the original array.

9. A binary search of a linked list is impractical because you cannot quickly locate its middle item.

10. You can use recursion to perform operations on a linked list. Such use will eliminate special cases and the need for a trailing reference.

11. The recursive insertion algorithm for a sorted linked list works because each smaller linked list is also sorted. When the algorithm makes an insertion at the beginning of one of these lists, the inserted node will be in the proper position in the original list. The algorithm is guaranteed to terminate because each smaller list contains one fewer node than the preceding list and because the empty list is a base case.

12. A tail reference can be used to facilitate locating the end of a list. This is especially useful when an append operation is required.

13. In a circular linked list, the last node references the first node, so that every node has a successor. If the list's external reference references the last node instead of the first node, you can access both the last node and the first node without traversing the list.

14. Dummy head nodes provide a method for eliminating the special cases for insertion into and deletion from the beginning of a linked list. The use of dummy head nodes is a matter of personal taste for singly linked lists, but it is helpful for a doubly linked list.

15. A doubly linked list allows you to traverse the list in either direction. Each node references its successor as well as its predecessor. Because insertions and deletions with a doubly linked list are more involved than with a singly linked list, it is convenient to use both a dummy head node and a circular organization to eliminate complicated special cases for the beginning and end of the list.

16. If you plan on storing the data contained in a linked list to a file, be sure to place the implements `Serializable` clause in each class that has instances that will be written to the file.

17. A generic class or interface enables you to defer the choice of certain data-type information until its use.

18. The Java Collections Framework contains interfaces, implementations, and algorithms for many common ADTs.

19. A collection is an object that holds other objects. An iterator cycles through the contents of a collection.

Cautions

1. An uninitialized reference variable has the value `null`. Attempting to use a reference with a value of `null` will cause a `NullPointerException` to be thrown.
2. The sequence

   ```
   Integer intRef = new Integer(5);
   intRef = null;
   ```

   allocates a memory cell and then destroys the only means of accessing it. Do not use `new` when you simply want to assign a value to a reference.
3. Insertions into and deletions from the beginning of a linked list are special cases unless you use a dummy head node. Failure to recognize this fact can result in a `null` reference being used, causing a `NullPointerException` to be thrown.
4. When traversing a linked list by using the reference variable `curr`, you must be careful not to reference `curr` after it has "passed" the last node in the list, because it will have the value `null` at that point. For example, the loop

   ```
   while (value > curr.getItem())
     curr.getNext();
   ```

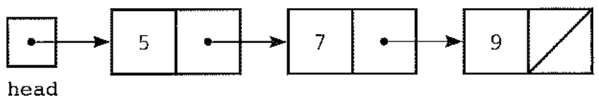
   is incorrect if `value` is greater than all the data values in the linked list, because `curr` becomes `null`. Instead, you should write

   ```
   while ((curr != null) && (value > curr.getItem()))
     curr.getNext();
   ```

   Because Java uses short-circuit evaluation of logical expressions, if `curr` becomes `null`, the expression `curr.getItem()` is not evaluated.
5. A doubly linked list is a data structure that programmers tend to overuse. However, a doubly linked list is appropriate to use when you have direct access to a node. In such cases, you would not have traversed the list from its beginning. If the list were singly linked, you would not have a reference to the preceding node. Because doubly linking the list provides an easy way to get to the node's predecessor as well as its successor, you can, for example, delete the node readily.

Self-Test Exercises

1. Given the following declarations, and the list shown, draw a picture which shows the result of each sequence of statements given below. If something illegal is done (as noted by the compiler), circle the offending statement and explain why it is illegal. Assume they all begin with this list:



```
class Node {
  private int item;
  private Node next;

  public void setItem(int x) {
    item = x;
  } // end setItem

  public int  getItem() {
    return x;
  } // end getItem

  public void setNext(Node n) {
    next = n;
  } // end setNext

  public Node getNext() {
    return next;
  } // end getNext
} // end Node

Node head, p, q;
int x;
```

a.
```
p = new Node();
p.setItem(3);
p.setNext(null);
head = p;
```

b.
```
x = 11;
p = new Node();
p = x;
p.setNext(head.getNext());
head = p;
```

c.
```
p = new Node();
p.setItem(3);
p.setNext(head.getNext());
head = p;
```

2. Consider the algorithm for deleting a node from a linked list that this chapter describes.

   a. Is deletion of the last node of a linked list a special case? Explain.

   b. Is deletion of the only node of a one-node linked list a special case? Explain.

   c. Does deleting the first node take more effort than deleting the last node? Explain.

3. a. Write Java statements that create the linked list pictured in Figure 5-31, as follows. Beginning with an empty linked list, first create and attach a node for J, then create and attach a node for E, and finally create and attach a node for B.

   b. Repeat Part *a*, but instead create and attach nodes in the order B, E, J.

4. Consider the sorted linked list of single characters in Figure 5-31. Suppose that `prev` references the first node in this list and `curr` references the second node.

   a. Write Java statements that delete the second node. (*Hint:* First modify Figure 5-31.)

   b. Now assume that `curr` references the first node of the remaining two nodes of the original list. Write Java statements that delete the first node.

   c. Now `head` references the only node that is left in the list. Write Java statements that insert a new node that contains A into the list so that the list remains sorted.

   d. Revise Figure 5-31 so that your new diagram reflects the results of the previous deletions and insertion.

5. Write a Java method that displays only the i^{th} integer in a linked list of integers. Assume that $i \geq 1$ and that the linked list contains at least i nodes.

6. How many assignment operations does the method that you wrote for Self-Test Exercise 5 require?

7. Write a recursive Java method that retrieves the i^{th} integer in a linked list of integers. Assume that $i \geq 1$ and that the linked list contains at least i nodes. (*Hint:* If $i = 1$, return the first integer in the list; otherwise, retrieve the $(i - 1)^{th}$ integer from the rest of the list.)

8. Trace the execution of `writeListBackward2(head)`, where `head` references the linked list of the characters pictured in Figure 5-31. (`writeListBackward2` appears on page 253 of this chapter.)

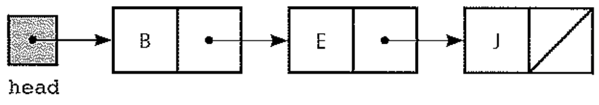


FIGURE 5-31

Linked list for Self-Test Exercises 3, 4, and 8

Exercises

1. This exercise assumes that you have completed Self-Test Exercise 4 and know the final status of the linked list. For each of the following, write the Java statements that perform the requested operation on the list. Also draw a picture of the status of the list after each operation is complete. When you delete a node from the list, return it to the system. All insertions into the list should maintain the list's sorted order. Do not use any of the methods that were presented in this chapter.

   a. Assume that `prev` references the first node and `curr` references the second node. Insert F into the list.

   b. Assume that `prev` references the second node and that `curr` references the third node of the list after you revised it in Part *a*. Delete the last node of the list.

   c. Assume that `prev` references the last node of the list after you revised it in Part *b*, and assume that `curr` is `null`. Insert G into the list.

2. Consider a linked list of items that are in no particular order.

   a. Write a method that inserts a node at the beginning of the linked list and a method that deletes the first node of the linked list.

   b. Repeat Part *a*, but this time perform the insertion and deletion at the end of the list instead of at the beginning.

   c. Repeat Part *b*, but this time assume that the list has a tail reference as well as a head reference.

3. Write a method to count the number of items in a linked list

   a. Iteratively

   b. Recursively

4. Write a method that will delete from a linked list of integers the node that contains the largest integer. Can you do this with a single traversal of the list?

5. The section "Processing Linked Lists Recursively" discussed the traversal of a linked list.

   a. Compare the efficiencies of an iterative method that displays a linked list with the method `writeList`.

   b. Write an iterative method that displays a linked list backward. Compare the efficiencies of your method with the method `writeListBackward2`.

6. Write a method to merge two linked lists of integers that are sorted into ascending order. The result should be a third linked list that is the sorted combination of the original lists. Do not destroy the original lists.

7. Compare the number of operations required to display each node in a linked list of integers with the number of operations required to display each item in an array of integers. A loop that displays the items in a linked list is given in the section "Displaying the Contents of a Linked List."

8. Compare the array-based and reference-based implementations of the ADT list operation `remove(index)`. Describe the work required for various values of `index` under each implementation. What are the implications for efficiency if the cost of shifting data is large compared to the cost of following a reference? When would this situation occur? What if the costs are approximately the same?

9. Assume that the reference `list` references the last node of a circular linked list like the one in Figure 5-24. Write a loop that displays the data portion of every node in the list.

10. Write the pseudocode for a method that inserts a new node to the end of a doubly linked list.

11. Write a method that inserts a new node at the start of a doubly linked list.

12. Using the `Sphere` class given on pages 197–199, write a program that creates 4 instances of the `Sphere` class (assigning a radius to each instance), and organizes them into a linked list. Include a method that prints out the statistics of all the `Sphere` instances in the list.

13. Write a method that deletes the *i*th node from a circular linked list.

14. Imagine a circular linked list of integers that are sorted into ascending order, as Figure 5-32a illustrates. The external reference `list` references the last node, which contains the largest integer. Write a method that revises the list so that its data elements are sorted into descending order, as Figure 5-32b illustrates. Do not allocate new nodes.

15. Revise the implementations of the ADT list operations `add` and `remove` under the assumption that the linked list has a dummy head node.

16. Add the operations `save` and `restore` to the reference-based implementation of the ADT list. These operations use a file to save and restore the list items.

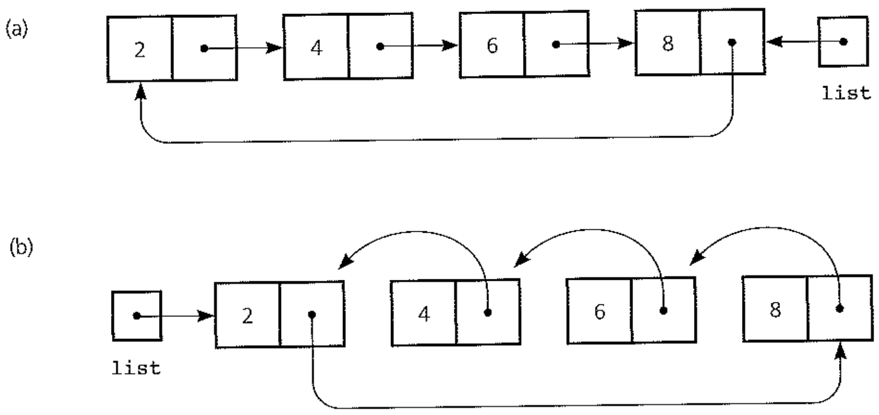


FIGURE 5-32

Two circular linked lists

17. Consider the sorted doubly linked list shown in Figure 5-27. This list is circular and has a dummy head node. Suppose that *newName* contains a name that you want to add to or delete from this list. Write two Java methods: one that inserts a new node containing *newName* into its proper sorted order within the list, and a second that deletes the node containing *newName*.

18. Repeat Exercise 17 for the sorted doubly linked list shown in Figure 5-26. This list is not circular and does not have a dummy head node. Watch out for the special cases at the beginning and end of the list.

19. The class *java.util.Vector* implements a growable array of objects. Here is a subset of the methods available in the class *java.util.Vector*:

    Constructors
    ```
    Vector();
    // Constructs an empty vector.

    Vector(int initialCapacity, int capacityIncrement)
    // Constructs an empty vector with the specified
    // initial capacity and capacity increment.
    ```

    Methods
    ```
    void addElement(Object obj);
    // Adds the specified component to the end of this
    // vector, increasing its size by one.

    int capacity();
    // Returns the current capacity of this vector.

    Object elementAt(int index);
    // Returns the component at the specified index.

    void removeElementAt(int index);
    // Deletes the component at the specified index.

    int size();
    // Returns the number of components in this vector.
    ```

    Complete the following tasks:

    a. Declare a vector using the default constructor. Add five string objects to the vector. What do methods *size* and *capacity* return?

    b. Write a loop to print the five strings stored in the vector.

    c. How many string objects must be added to the vector to cause the capacity to change? Is this answer consistent with Part *a*? What is the new capacity?

    d. Remove all of the elements fromt the vector. Does the capacity change? Why do you think the *Vector* class behaves this way?

20. You can have a linked list of linked lists, as Figure 5-30 indicates. Assume the Java definitions on page 000. Suppose that *curr* references a desired stock item (node) in the inventory list. Write some Java statements that add a customer to the end of the wait list associated with the node referenced by *curr*.

Programming Problems

1. Chapter 4 introduced the ADT sorted list, which maintains its data in sorted order. For example, a sorted list of names would be maintained in alphabetical order, and a sorted list of numbers would be maintained in either increasing or decreasing order. The operations for a sorted list are summarized on pages 117–118.

   Some operations—`sortedIsEmpty`, `sortedSize`, and `sortedGet`, for example—are just like those for the ADT list. Insertion and deletion operations, however, are by value, not by position as they are for a list. For example, when you insert an item into a sorted list, you do not specify where in the list the item belongs. Instead, the insertion operation determines the correct position of the item by comparing its value with those of the existing items on the list. A new operation, `locateIndex`, determines from the value of an item its numerical position within the sorted list.

   Note that the specifications given in Chapter 4 do not say anything about duplicate entries in the sorted list. Depending on your application, you might allow duplicates, or you might want to prevent duplicates from entering the list. For example, a sorted list of social security numbers probably should disallow duplicate entries. In this example, an attempt to insert a social security number that already exists in the sorted list would fail.

   Write a nonrecursive, reference-based implementation of the ADT sorted list of integers as a Java class such that

   a. Duplicates are allowed

   b. Duplicates are not allowed, and operations must prevent duplicates from entering the list

2. Repeat Programming Problem 1, but write a recursive, reference-based implementation instead. Recall from this chapter that the recursive methods must be in the private section of the class.

3. Write an implementation of the ADT list that uses a resizeable array to represent the list items.

4. Write a reference-based implementation of the ADT two-ended list, which has insertion and deletion operations at both ends of the list,

   a. Without a tail reference

   b. With a tail reference

5. Implement the node structure, including the constructor, for a circular doubly linked list with a dummy head node. Complete the `insert` and `remove` methods for this list, as described in the section "Doubly Linked Lists."

6. Implement the ADT character string by using a linked list of characters. Include typical operations such as append one string to another; extract a substring; find the index of the leftmost occurrence of a character in a string; determine whether one string is a substring of another. Implement the string so that you can obtain its length without traversing the linked list and counting.

7. Consider a sparse implementation of the ADT polynomial that stores only the terms with nonzero coefficients. For example, you can represent the polynomial p in Exercise 12 of Chapter 4 with the linked list in Figure 5-33.

   a. Complete the sparse implementation.

   b. Define a traverse operation for the ADT polynomial that will allow you to add two sparse polynomials without having to consider terms with zero coefficients explicitly.

8. When you play a board or card game or when you use a shared computing resource, you get a turn and then wait until everyone else has had a turn. Although the number of players in a game remains relatively static, the number of users of a shared computing service fluctuates. Let's assume that this fluctuation will occur.

   Design an ADT that keeps track of turns within a group of people. You should be able to add or delete people and determine whose turn occurs now.

   Begin with a given group of people; assign these people an initial order. (This order can be random or specified by the user.) The first new person joining the group should get a turn after all others have had an equal number of turns. Each subsequent new person should get a turn after the person who joined the group most recently has a turn.

   Also design an ADT to represent a person. (You can be conservative with the amount of data that this ADT contains.) The data elements that your first ADT stores are instances of the ADT person.

   Implement your ADTs as Java classes. Use a circular linked list as the data structure that keeps track of turns. Implement this linked list by using references and not by using any other class.

   Write a program that uses and, therefore, tests, your ADTs completely. Your program should process several insertion and deletion operations and should demonstrate that people are given turns correctly.

9. Repeat Programming Problem 8, using the ADT you created to represent a customer and the JCF class `LinkedList`—a class that implements the JCF `List` interface.

10. Occasionally, a linked structure that does not use references is useful. One such structure uses an array whose items are "linked" by array indexes. Figure 5-34a illustrates an array of nodes that represents the linked list in Figure 5-31. Each node has two data fields, `item` and `next`. The `next` data field is an integer index to the array element that contains the next node in the linked list. Note that the `next` data field of the last node contains –1. The integer variable `head` contains the index of the first node in the list.

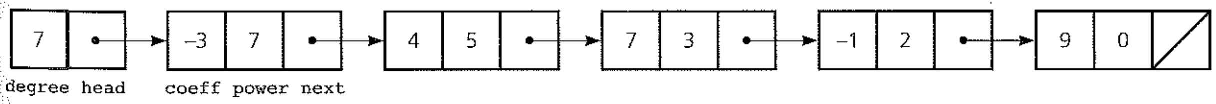


FIGURE 5-33

A sparse polynomial

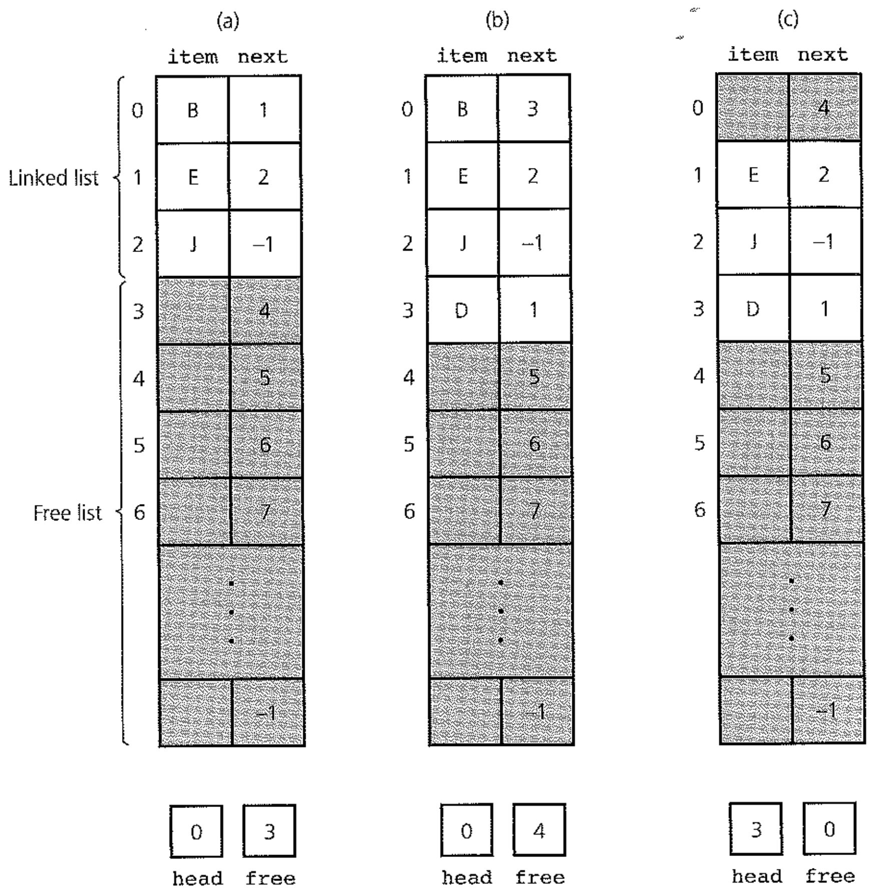


FIGURE 5-34

(a) An array-based implementation of the linked list in Figure 5-31; (b) after inserting D in sorted order; (c) after deleting B

The array elements that currently are not a part of the linked list make up a **free list** of available nodes. These nodes form another linked list, with the integer variable `free` containing the index of the first free node. To insert an item into the original linked list, you take a free node from the beginning of the free list and insert it into the linked list (Figure 5-34b). When you delete an item from the linked list, you insert the node into the beginning of the free list (Figure 5-34c). In this way, you can avoid shifting data items.

Implement the ADT list by using this array-based linked list.

11. Write the program for the DVD inventory problem that this chapter describes.

12. Modify and expand the inventory program that you wrote for the previous programming problem. Here are a few suggestions:

    a. Add the ability to manipulate more than one inventory with the single program.

    b. Add the ability to keep various statistics about each of the inventory items (such as the average number sold per week for the last 10 weeks).

    c. Add the ability to modify the have value for an inventory item (for example, when a DVD is damaged or returned by a customer). Consider the implications for maintaining the relationship between a have value and the size of the corresponding wait list.

    d. Make the wait lists more sophisticated. For example, keep names and addresses; mail letters automatically when a DVD comes in.

    e. Make the ordering mechanism more sophisticated. For instance, do not order DVDs that have already been ordered but have not yet been delivered.