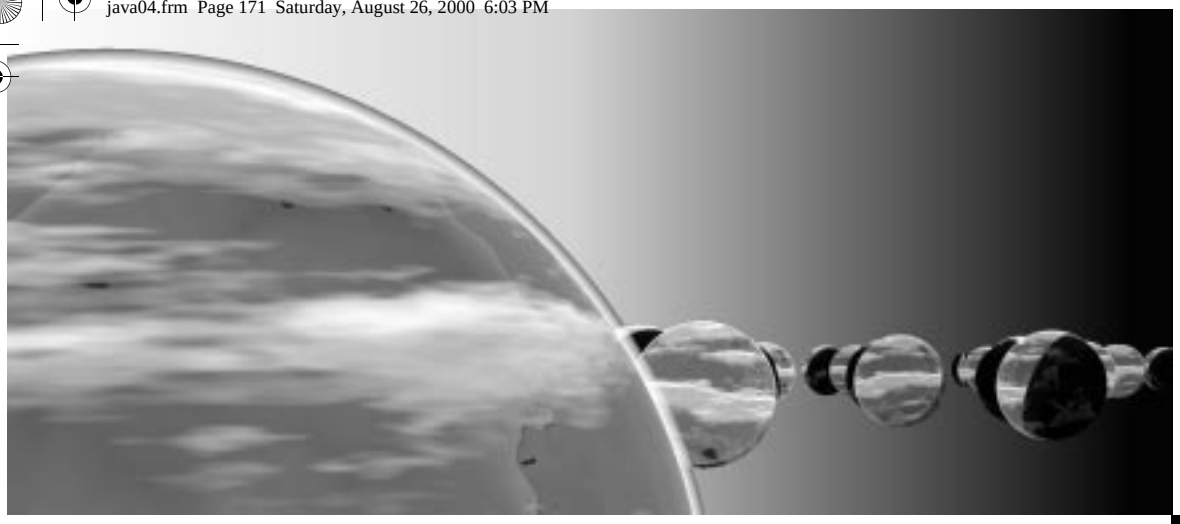


CHAPTER

4

Linked Lists

The simplest way to interrelate or link a set of elements is to line them up in a single list . . . For, in this case, only a single link is needed for each element to refer to its successor.

NIKLAUS WIRTH
Algorithms + Data Structures = Programs

We begin this chapter with a concrete discussion of a new data structure, the *linked list*, which is used to implement a list of elements arranged in some kind of order. The linked list structure uses memory that shrinks and grows as needed but in a different manner than arrays. The discussion of linked lists includes the specification and implementation of a node class, which incorporates the fundamental notion of a single element of a linked list.

linked lists are used to implement a list of elements arranged in some kind of order

Once you understand the fundamentals, linked lists can be used as part of an ADT, similar to the way that arrays have been used in previous ADTs. For example, linked lists can be used to reimplement the bag and sequence ADTs.

By the end of the chapter you will understand linked lists well enough to use them in various programming projects (such as the revised bag and sequence ADTs) and in the ADTs of future chapters. You will also know the advantages and drawbacks of using linked lists versus arrays for these ADTs.

4.1 FUNDAMENTALS OF LINKED LISTS

A **linked list** is a sequence of elements arranged one after another, with each element connected to the next element by a "link." A common programming practice is to place each element together with the link to the next element, resulting in a component called a **node**. A node is represented pictorially as a box with the element written inside the box and the link drawn as an arrow pointing out of the box. Several typical nodes are drawn in Figure 4.1. For example, the topmost node has the number 12 as its element. Most of the nodes in the figure also have an arrow pointing out of the node. These arrows, or **links**, are used to connect one node to another.

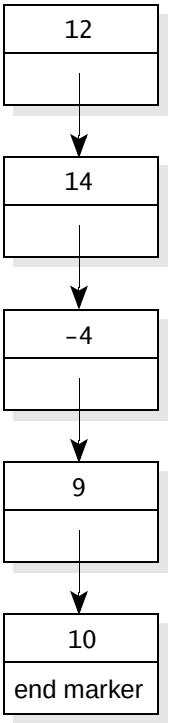


FIGURE 4.1
A Linked List Made of Nodes Connected with Links

The links are represented as arrows because they do more than simply connect two nodes. The links also place the nodes in a particular order. In Figure 4.1, the five nodes form a chain from top to bottom. The first node is linked to the second node; the second node is linked to the third node; and so on until we reach the last node. We must do something special when we reach the last node, since the last node is not linked to another node. In this special case, we will replace the link in this node with a note saying "end marker."

Declaring a Class for Nodes

Each node contains two pieces of information: an element (which is a number for these example nodes) and an arrow. But just *what* are those arrows? Each arrow points to another node, or you could say that each arrow *refers* to another node. With this in mind, we can implement a Java class for a node using two instance variables: an instance variable to hold the element, and a second instance variable that is a reference to another node. In Java, the two instance variables can be declared at the start of the class:

```
public class IntNode
{
   private int data;       // The element stored in this node
   private IntNode link;   // Reference to the next node in the list
   ...
```

We'll provide the methods later, in Sections 4.2 and 4.3. For now we want to focus on the instance variables, `data` and `link`. The data is simply an integer element, though we could have had some other kind of elements, perhaps double numbers, or characters, or whatever.

The second instance variable, called `link`, is a reference to another node. For example, the link variable in the first node is a reference to the second node. Our drawings will represent each link reference as an arrow leading from one node to another. In fact, we have previously used arrows to represent references to objects in Chapters 2 and 3, so these links are nothing new.

Head Nodes, Tail Nodes

When a program builds and manipulates a linked list, the list is usually accessed through references to one or more important nodes. The most common access is through the list's first node, which is called the **head** of the list. Sometimes we maintain a reference to the last node in a linked list. The last node is the **tail** of the list. We could also maintain references to other nodes in a linked list.

Each reference to a node used in a program must be declared as a node variable. For example, if we are maintaining a linked list with references to the head and tail, then we would declare two node variables:

```
IntNode head;
IntNode tail;
```

The program can now proceed to create a linked list, always ensuring that `head` refers to the first node and `tail` refers to the last node, as shown in Figure 4.2.

Building and Manipulating Linked Lists

Whenever a program builds and manipulates a linked list, the nodes are accessed through one or more references to nodes. Typically, a program includes a reference to the first node (the **head**) and a reference to the last node (the **tail**).

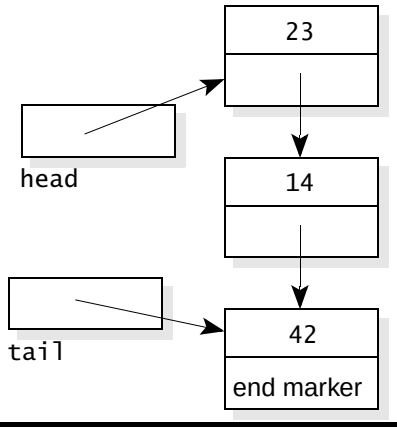
FIGURE 4.2 Node Declarations in a Program with a Linked List

Declaration from the IntNode Class

```
public class IntNode
{
   private int data;
   private IntNode link;
   ...
```

Declaring Two Nodes in a Program

```
IntNode head;
IntNode tail;
```



A computation might create a small linked list with three nodes, as shown here. The `head` and `tail` variables provide access to two nodes inside the list.

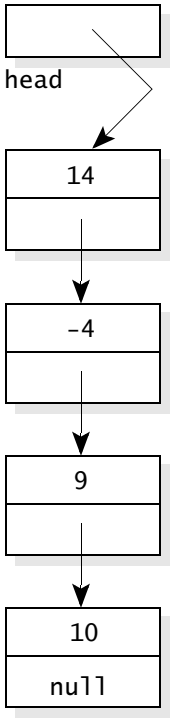


FIGURE 4.3
Linked List with the Null Reference at the Final Link

The Null Reference

Figure 4.3 illustrates a linked list with a reference to the head node and one new feature. Look at the link part of the final node. Instead of a reference, we have written the word `null`. The word `null` indicates the **null reference**, which is a special Java constant. You can use the null reference for any reference variable that has nothing to refer to. There are several common situations where the null reference is used:

- In Java, when a reference variable is first declared and there is not yet an object for it to refer to, it can be given an initial value of the null reference. Examples of this initial value are shown in Chapter 2 on page 50.
- The null reference is used for the link part of the final node of a linked list.
- When a linked list does not yet have any nodes, the null reference is used for the the head and tail reference variables. Such a list is called the **empty list**.

In a program, the null reference is written as the keyword `null`.

The Null Reference and Linked Lists

The null reference is a special Java value that can be used for any reference variable that has nothing to refer to.

The null reference occurs in the link part of the final node of a linked list.

A program that maintains a head and tail reference may set these references to null, which indicates that the list is empty (has no nodes).

PITFALL

Pitfall: Null Pointer Exceptions with Linked Lists

When a reference variable is null, it is a programming error to activate one of its methods or to try to access one of its instance variables. For example, a program may maintain a reference to the head node of a linked list, as shown here:

```
IntNode head;
```

Initially, the list is empty and `head` is the null reference. At this point, it is a programming error to activate one of `head`'s methods. The error would occur as a `NullPointerException`.

The general rules: Never activate a method of the null reference. Never try to access an instance variable of the null reference. In both cases, the result would be a `NullPointerException`.

Self-Test Exercises

1. Write the start of the class declaration for a node in a linked list. The data in each node is a double number.
2. Suppose a program builds and manipulates a linked list. What two special nodes would the program typically keep track of?
3. Describe two common uses for the null reference in the realm of linked lists.
4. What happens if you try to activate a method of the null reference?

4.2 METHODS FOR MANIPULATING NODES

We're ready to write methods for the `IntNode` class, which begins like this:

```
public class IntNode
{
   private int data;        // The element stored in this node
   private IntNode link;    // Reference to the next node in the list
   ...
```

There will be methods for creating, accessing, and modifying nodes, plus methods and other techniques for adding or removing nodes from a linked list. We begin with a constructor that's responsible for initializing the two instance variables of a new node.

Constructor for the Node Class

The node's constructor has two arguments, which are the initial values for the node's data and link variables, as specified here:

- **Constructor for the IntNode**

  `public IntNode(int initialData, IntNode initialLink)`

  Initialize a node with a specified initial data and link to the next node. Note that the `initialLink` may be the null reference, which indicates that the new node has nothing after it.

  Parameters:

  `initialData` – the initial data of this new node

  `initialLink` – a reference to the node after this new node—the reference may be null to indicate that there is no node after this new node.

  Postcondition:

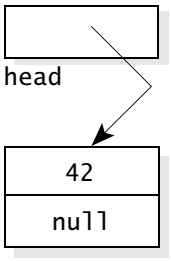
  This new node contains the specified data and link to the next node.

The constructor's implementation copies its two parameters to the instance variables `data` and `link`:

```
public IntNode(int initialData, IntNode initialLink)
{
   data = initialData;
   link = initialLink;
}
```

As an example, the constructor can be used by a program to create the first node of a linked list:

```
IntNode head;
head = new IntNode(42, null);
```



After these two statements, head refers to the `head` node of a small linked list that contains just one node with the number 42. We'll look at the formation of longer linked lists after we see four other basic node methods.

Getting and Setting the Data and Link of a Node

The node has an accessor method and a modification method for each of its two instance variables, as specified here:

getData, getLink, setData, setLink

- **getData**

  `public int getData( )`

  Accessor method to get the data from this node.

  Returns:

  the data from this node

- **getLink**

  `public IntNode getLink( )`

  Accessor method to get a reference to the next node after this node.

  Returns:

  a reference to the node after this node (or the null reference if there is nothing after this node)

- **setData**

  `public void setData(int newdata)`

  Modification method to set the data in this node.

  Parameters:

  `newData` – the new data to place in this node

  Postcondition:

  The data of this node has been set to `newData`.

- **setLink**

  `public void setLink(IntNode newLink)`

  Modification method to set the reference to the next node after this node.

  Parameters:

  `newLink` – a reference to the node that should appear after this node in the linked list (or the null reference if there should be no node after this node)

  Postcondition:

  The link to the node after this node has been set to `newLink`. Any other node (that used to be in this link) is no longer connected to this node.

The implementations of the four methods are each short. For example:

```
public void setLink(IntNode newLink)
{
   link = newLink;
}
```

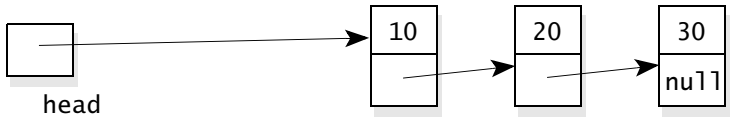
Public versus Private Instance Variables

In addition to `setLink`, there are the three other short methods that we'll leave for you to implement. You may wonder why bother having these short methods at all. Wouldn't it be simpler and more efficient to just make `data` and `link` public, and do away with the short methods altogether? Yes, public instance variables probably are simpler, and in Java the direct access of an instance variable is considerably more efficient than calling a method. On the other hand, debugging can be easier with access and modification methods in place because we can set breakpoints to see whenever an instance variable is accessed or modified. Also, private instance variables provide good information hiding so that later changes to the class won't affect programs that use the class.

Anyway, the public-versus-private question should be addressed for many of your classes, with the answer based on the intended use and required efficiency together with software engineering principles such as information hiding. For the classes in this text, we'll lean toward information hiding and avoid public instance variables.

Adding a New Node at the Head of a Linked List

New nodes can be added at the head of a linked list. To accomplish this, the program needs a reference to the head node of a list as shown here:

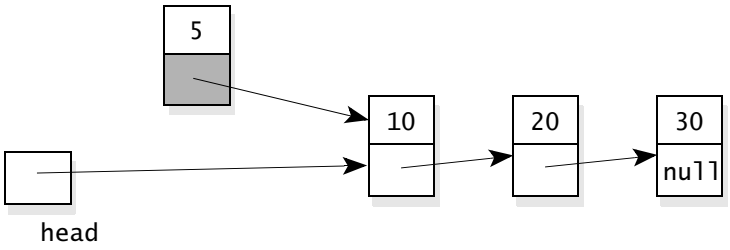


In this example, suppose that we want to add a new node to the front of this list, with 5 as the data. Using the node constructor, we can write

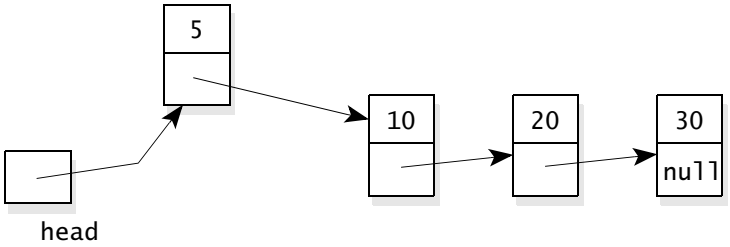
```
head = new IntNode(5, head);
```

how to add a new node at the head of a linked list

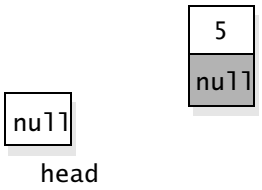
Let's step through the execution of this statement to see how the new node is added at the front of the list. When the constructor is executed, a new node is created with 5 as the data and with the link refering to the same node that `head` refers to. Here's what the picture looks like, with the link of the new node shaded:



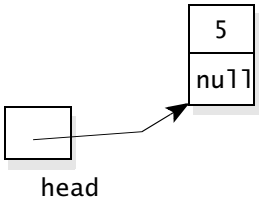
The constructor returns a reference to the newly created node, and in the statement we wrote `head = new IntNode(5, head)`. You can read this statement as saying "make `head` refer to the newly created node." Therefore, we end up with this situation:



By the way, the technique works correctly even if we start with an empty list (in which the head reference is null). In this case, the statement `head = new IntNode(5, head)` creates the first node of the list. To see this, suppose we start with a null head and execute the statement. The constructor creates a new node with 5 as the data and with head as the link. Since the head reference is null, the new node looks like this (with the link of the new node shaded):



After the constructor returns, `head` is assigned to refer to the new node, so the final situation looks like this:



As you can see, the statement `head = new IntNode(5, head)` has correctly added the first node to a list. If we are maintaining a reference to the tail node, then we would also set the tail to refer to this one node.

Adding a New Node at the Head of a Linked List

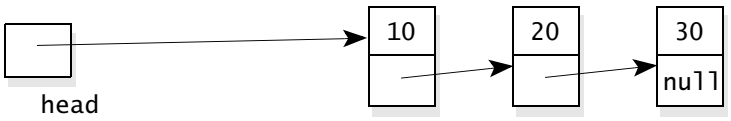
Suppose that `head` is the head reference of a linked list. Then this statement adds a new node at the front of the list with the specified new data:

```
head = new IntNode(newData, head);
```

This statement works correctly even if we start with an empty list (in which case the head reference is null).

Removing a Node from the Head of a Linked List

Nodes can be removed from the head of the linked list. To accomplish this, we need a reference to the head node of a list as shown here:

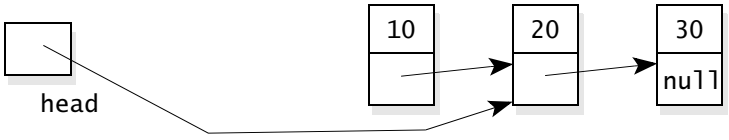


To remove the first node, we simply move the head, so that it refers to the next node. This is accomplished with one statement:

```
head = head.getLink( );
```

how to remove a node from the head of a linked list

The right side of the assignment, `head.getLink( )`, is a reference to the second node of the list. So, after the assignment, `head` refers to the second node, as shown at the top of the next page.



This picture is peculiar. It looks like we've still got a linked list with three nodes containing 10, 20, and 30. But if we start at the head, there are only the two nodes with 20 and 30. The node with 10 can no longer be accessed starting at the head, so it is not really part of the linked list any more. In fact, if a situation arises where a node can no longer be accessed from anywhere in a program, then the Java runtime system recognizes that the node has strayed, and the memory used by that node will be reused for other things. This technique of rounding up stray memory is called **garbage collection**, and it happens automatically for Java programs. In other programming languages, the programmer is responsible for identifying memory that is no longer used, and explicitly returning that memory to the runtime system.

automatic garbage collection has some inefficiency, but it's less prone to programming errors

What are the tradeoffs between automatic garbage collection and programmer-controlled memory handling? Automatic garbage collection is slower when a program is executing, but the automatic approach is less prone to errors and it frees the programmer to concentrate on more important issues.

Anyway, we'll remove a node from the front of a list with the statement `head = head.getLink( )`. This statement also works when the list has only one node, and we want to remove this one node. For example, consider this list:

In this situation, we can execute `head = head.getLink( )`. The `getLink( )` method returns the link of the head node—in other words, it returns null. So, the null reference is assigned to the head, ending up with this situation:

Now, the head is null, which indicates that the list is empty. If we are maintaining a reference to the tail, then we would also have to set the tail reference to null. The automatic garbage collection will take care of reusing the memory occupied by the one node.

Removing a Node from the Head of a Linked List

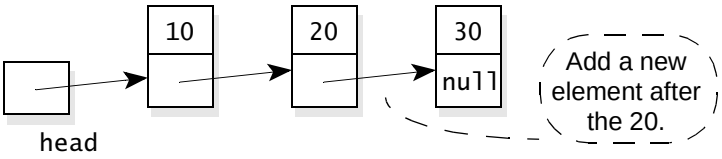
Suppose that `head` is the head reference of a linked list. Then this statement removes a node from the front of the list:

```
head = head.getLink( );
```

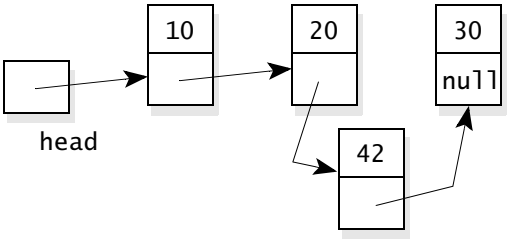
This statement works correctly even when the list has just one node (in which case the head reference becomes null).

Adding a New Node That Is Not at the Head

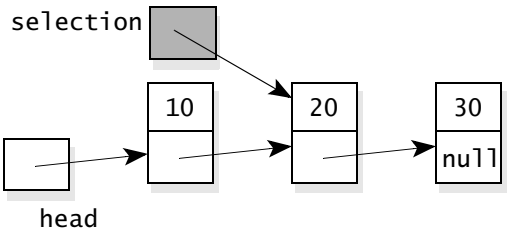
New nodes are not always placed at the head of a linked list. They may be added in the middle or at the tail of a list. For example, suppose you want to add the number 42 after the 20 in this list:



After the addition, the new, longer list has these four nodes:



Whenever a new node is not at the head, the process requires a reference to the node that is just *before* the intended location of the new node. In our example, we would require a reference to the node that contains 20, since we want to place the new node after this node. This special node is called the "selected node"—the new node will go just after the selected node. We'll use the name `selection` for a reference to the selected node. So to add an element after the 20, we would first have to set up `selection` as shown here:



Once a program has calculated `selection`, the new node with data of 42 can be added with a method of the `IntNode` class, specified here:

addNodeAfter

- **addNodeAfter**

  `public void addNodeAfter(int element)`

  Modification method to add a new node after this node.

  Parameters:

  `element` – the data to be placed in the new node

  Postcondition:

  A new node has been created and placed after this node. The data for the new node is `element`. Any other nodes that used to be after this node are now after the new node.

  Throws: `OutOfMemoryError`

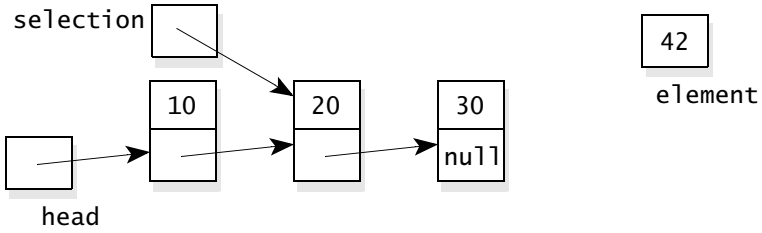
  Indicates that there is insufficient memory for a new `IntNode`.

For example, to add a new node with data 42 after the selected node, we can activate `selection.addNodeAfter(42)`.

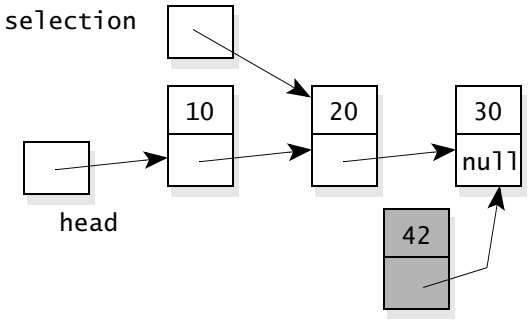
The implementation of `addNodeAfter` requires just one line, shown here:

```
public void addNodeAfter(int element)
{
   link = new IntNode(element, link);
}
```

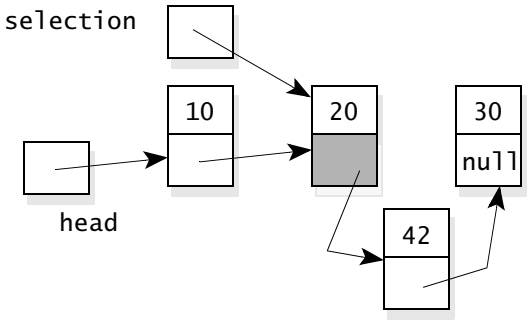
Let's see exactly what happens when we set up `selection` as shown earlier and then execute `selection.addNodeAfter(42)`. The value of element is 42, so we have the situation shown here:



The method executes `link = new IntNode(element, link)`, where `element` is 42 and `link` is from the selected node—in other words, `link` is `selection.link`. On the right side of the statement, the `IntNode` constructor is executed, and a new node is created with 42 as the `data` and with the `link` of the new node being the same as `selection.link`. The situation is shown at the top of the next page, with the new node shaded.



The constructor returns a reference to the newly created node, and in the assignment statement we wrote `link = new IntNode(element, link)`. You can read this statement as saying "change the link part of the selected node so that it refers to the newly created node." This change is made in the following drawing, which highlights the link part of the selected node:



After adding the new node with 42, you can step through the complete linked list, starting at the head node 10, then 20, then 42, and finally 30.

The approach we have used works correctly even if the selected node is the tail of a list. In this case, the new node is added after the tail. If we were maintaining a reference to the tail node, then we would have to update this reference to refer to the newly added tail.

Adding a New Node That is Not at the Head

Suppose that `selection` is a reference to a node of a linked list. Activating the following method adds a new node after the cursor node with `element` as the new data:

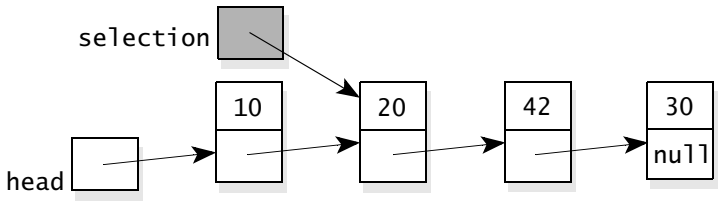
```
selection.addNodeAfter(element);
```

The implementation of `addNodeAfter` needs only one statement to accomplish its work:

```
link = new IntNode(element, link);
```

Removing a Node That Is Not at the Head

It is also possible to remove a node that is not at the head of a linked list. The approach is similar to adding a node in the middle of a linked list. To remove a midlist node, we must set up a reference to the node that is just *before* the node that we are removing. For example, in order to remove the 42 from the following list, we would need to set up `selection` as shown here:



As you can see, `selection` does not actually refer to the node that we are deleting (the 42); instead it refers to the node that is just before the condemned node. This is because the link of the *previous* node must be reassigned, hence we need a reference to this previous node. The removal method's specification is shown here:

removeNodeAfter

- **removeNodeAfter**

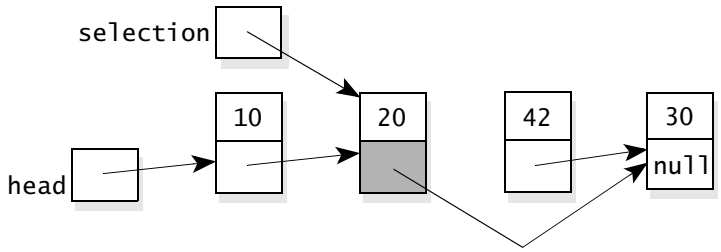
  `public void removeNodeAfter( )`

  Modification method to remove the node after this node.

  Precondition:
  This node must not be the tail node of the list.

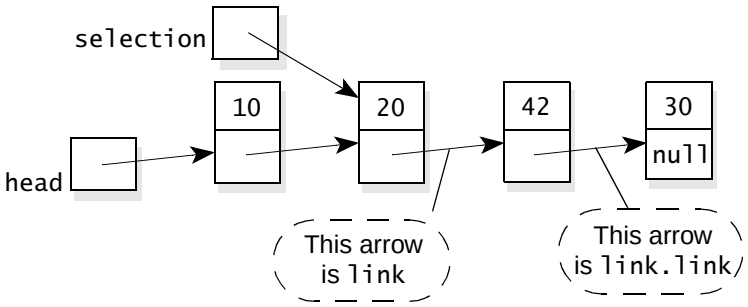
  Postcondition:
  The node after this node has been removed from the linked list. If there were further nodes after that one, they are still present on the list.

For example, to remove the 42 from the list drawn above, we would activate `selection.removeNodeAfter( )`. After the removal, the new list will look like this (with the changed link highlighted):



At this point, the node containing 42 is no longer part of the linked list. The list's first node contains 10, the next node has 20, and following the links we arrive at the third and last node containing 30. Java's automatic garbage collection will reuse the memory of the removed node.

As you can see from the example, the implementation of `removeNodeAfter` must alter the link of the node that activated the method. How is the alteration carried out? Let's go back to our starting position, but we'll put a bit more information in the picture:



To work through this example, you need some patterns that can be used within the method to refer to the various data and link parts of the nodes. Remember that we activated `selection.removeNodeAfter( )`. So the node that activated the method has 20 for its data, and its link is indicated by the caption "This arrow is `link`." So we can certainly use these two names within the method:

`data`	This is the data of the node that activated the method (20).
`link`	This is the link of the node that activated the method. This link refers to the node that we are removing.

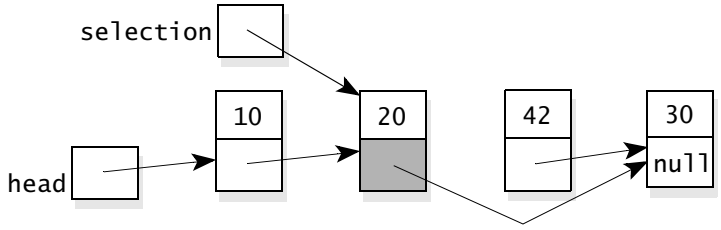
Because the name `link` refers to a node, we can also use the names `link.data` and `link.link`:

`link.data`	This notation means "go to the node that `link` refers to and use the `data` instance variable." In our example, `link.data` is 42.
`link.link`	This notation means "go to the node that `link` refers to and use the `link` instance variable." In our example, `link.link` is the reference to the node that contains 30.

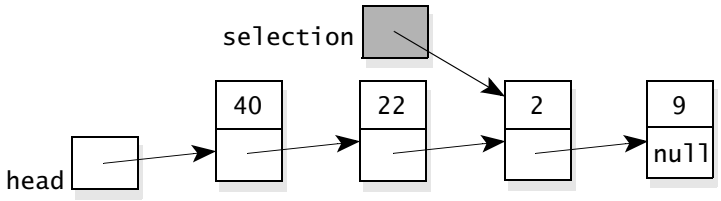
In the implementation of `removeNodeAfter`, we need to make `link` refer to the node that contains 30. So, using the notation just shown, we need to assign `link = link.link`. The complete implementation is at the top of the next page.

```
public void removeNodeAfter( )
{
   link = link.link;
}
```

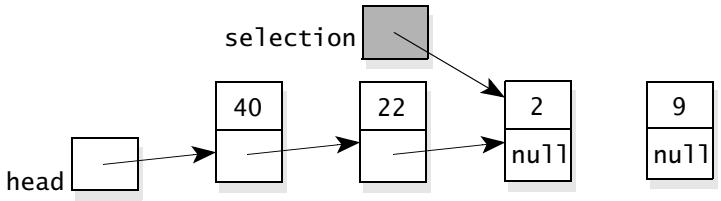
The notation `link.link` does look strange, but just read it from left to right so that it means "go to the node that link refers to and use the link instance variable." In our example, the final situation after assigning `link = link.link` is just what we want, as shown here:



The `removeNodeAfter` implementation works fine, even if we want to remove the tail node. Here's an example where we have set `selection` to refer to the node that's just before the tail of a small list:



When we activate `selection.removeNodeAfter( )`, the link of the selected node will be assigned the value `null` (which is obtained from the link of the next node). The result is this picture:



The tail node has been removed from the list. If the program maintains a reference to the tail node, then that reference must be updated to refer to the new tail.

In all cases, Java's automatic garbage collection takes care of reusing the memory of the removed node.

Removing a Node That is Not at the Head

Suppose that `selection` is a reference to a node of a linked list. Activating the following method removes the node after the selected node:

```
selection.removeNodeAfter( );
```

The implementation of `removeNodeAfter` needs only one statement to accomplish its work:

```
link = link.link;
```

Pitfall: Null Pointer Exceptions with `removeNodeAfter`

PITFALL

The `removeNodeAfter` method has a potential problem. What happens if the tail node activates `removeNodeAfter`? This is a programming error because `removeNodeAfter` would try to remove the node after the tail node, and there is no such node. The precondition of `removeNodeAfter` explicitly states that it must not be activated by the tail node. Still, what will happen in this case? For the tail node, `link` is the null reference, so trying to access the instance variable `link.link` will result in a `NullPointerException`.

When we write the complete specification of the node methods, we will include a note indicating the possibility of a `NullPointerException` in this method.

Self-Test Exercises

5. Suppose that `head` is a head reference for a linked list of integers. Write a few lines of code that will add a new node with the number 42 as the second element of the list. (If the list was originally empty, then 42 should be added as the first node instead of the second.)
6. Suppose that `head` is a head reference for a linked list of integers. Write a few lines of code that will remove the second node of the list. (If the list originally had only one node, then remove that node instead; if it had no nodes, then leave the list empty.)
7. Examine the techniques for adding and removing a node at the head. Why are these techniques implemented as static methods rather than ordinary `IntNode` methods?
8. Write some code that could appear in a main program. The code should declare head and tail references for a linked list of integers, then add nodes with the numbers 1 through 100 (in that order). Throughout the program, `head` and `tail` should remain valid references to the head and tail nodes.

4.3 MANIPULATING AN ENTIRE LINKED LIST

We can now write programs that use linked lists. Such a program declares some references to nodes, such as a head and tail reference. The nodes are manipulated with the methods and other techniques that we have already seen. But all these methods and techniques deal with just one or two nodes at an isolated part of the linked list. Many programs also need techniques for carrying out computations on an entire list, such as computing the number of nodes on a list. This suggests that we should write a few more methods for the `IntNode` class—methods that carry out some computation on an entire linked list. For example, we can provide a method with this heading:

```
public static int listLength(IntNode head)
```

The `listLength` method computes the number of nodes in a linked list. The one parameter, `head`, is a reference to the head node of the list. For example, the last line of this code prints the length of a short list:

```
IntNode small; // Head reference for a small list
small = new IntNode(42, null);
small.addNodeAfter(17);
System.out.println(IntNode.listLength(small)); // Prints 2
```

By the way, the `listLength` return value is `int`, so that the method can be used only if a list has fewer than `Integer.MAX_VALUE` nodes. Beyond this length, the `listLength` method will return a wrong answer because of arithmetic overflow. We'll make a note of the potential problem in the `listLength` specification.

Notice that `listLength` is a static method. It is not activated by any one node; instead we activate `IntNode.listLength`. But why is it a *static* method—wouldn't it be easier to write an ordinary method that is activated by the head node of the list? Yes, an ordinary method might be easier, but a static method is better because a static method can be used even for an empty list. For example, these two statements create an empty list and print the length of that list:

```
IntNode empty = null; // empty is null, representing an empty list
System.out.println(IntNode.listLength(empty)); // Prints 0
```

An ordinary method could not be used to compute the length of the empty list, because the head reference is null.

Manipulating an Entire Linked List

To carry out computations on an entire linked list, we will write static methods in the `IntNode` class. Each such method has one or more parameters that are references to nodes in the list. Most of the methods will work correctly even if the references are null (indicating an empty list).

Computing the Length of a Linked List

Here is the complete specification of the `listLength` method that we've been discussing:

listLength

- **listLength**

  `public static int listLength(IntNode head)`

  Compute the number of nodes in a linked list.

  Parameters:

  `head` – the head reference for a linked list (which may be an empty list with a null head)

  Returns:

  the number of nodes in the list with the given head

  Note:

  A wrong answer occurs for lists longer than `Int.MAX_VALUE`, because of arithmetic overflow.

The precondition indicates that the parameter, `head`, is the head reference for a linked list. If the list is not empty, then `head` refers to the first node of the list. If the list is empty, then `head` is the null reference (and the method returns zero, since there are no nodes).

Our implementation uses a reference variable to step through the list, counting the nodes one at a time. Here are the three steps of the pseudocode, using a reference variable named `cursor` to step through the nodes of the list one at a time. (We often use the name `cursor` for such a variable, since "cursor" means "something that runs through a structure.")

1. Initialize a variable named `answer` to zero (this variable will keep track of how many nodes we have seen so far).
2. Make `cursor` refer to each node of the list, starting at the head node. Each time `cursor` moves, add one to `answer`.
3. `return answer`.

Both `cursor` and `answer` are local variables in the method.

The first step initializes `answer` to zero, because we have not yet seen any nodes. The implementation of Step 2 is a for-loop, following a pattern that you should use whenever *all of the nodes of a linked list must be traversed*. The general pattern looks like this:

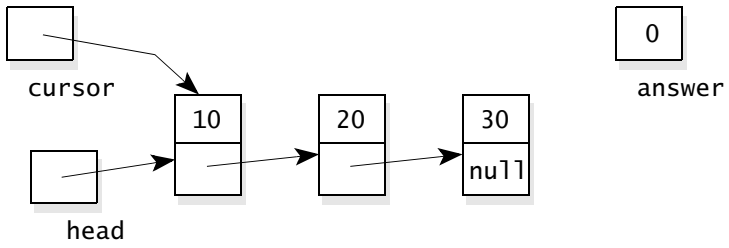
how to traverse all the nodes of a linked list

```
for (cursor = head; cursor != null; cursor = cursor.link)
{
    ...     Inside the body of the loop, you may
            carry out whatever computation is
            needed for a node in the list.
}
```

For the `listLength` method, the "computation" inside the loop is simple because we are just counting the nodes. Therefore, in our body we will just add one to `answer`, as shown here:

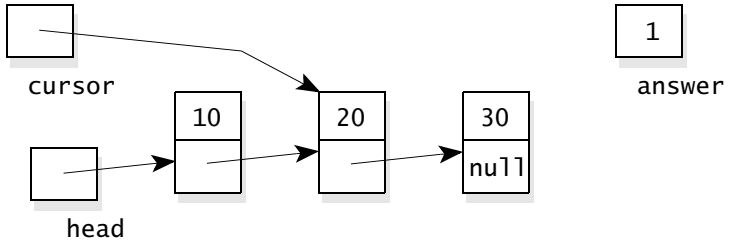
```
for (cursor = head; cursor != null; cursor = cursor.link)
   answer++;
```

Let's examine the loop on an example. Suppose that the linked list has three nodes containing the numbers 10, 20, and 30. After the loop initializes (with `cursor = head`), we have this situation:

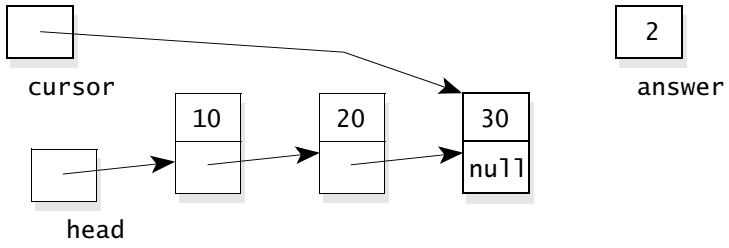


Notice that `cursor` refers to the same node that `head` refers to.

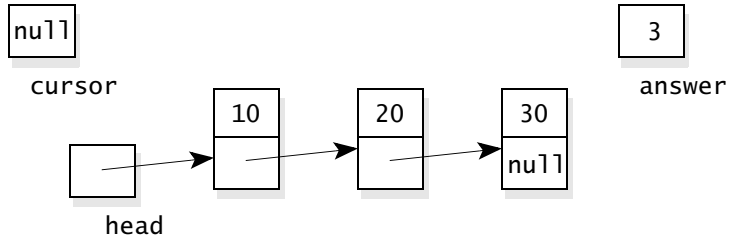
Since `cursor` is not `null`, we enter the body of the loop. Each iteration increments `answer` and then executes `cursor = cursor.link`. The effect of `cursor = cursor.link` is to copy the `link` part of the first node into `cursor` itself, so that `cursor` ends up refering to the second node. In general, the statement `cursor = cursor.link` moves `cursor` to the next node. So, at the completion of the loop's first iteration, the situation is this:



The loop continues. After the second iteration, `answer` is 2, and `cursor` refers to the third node of the list, as shown here:



Each time we complete an iteration of the loop, `cursor` refers to some location in the list, and `answer` is the number of nodes *before* this location. In our example, we are about to enter the loop's body for the third and last time. During the last iteration, `answer` is incremented to 3, and `cursor` becomes `null`, as shown here:



The variable `cursor` has become `null` because the loop control statement `cursor = cursor.link` copied the `link` part of the third node into `cursor`. Since this `link` part is `null`, the value in `cursor` is now `null`.

At this point, the loop's control test `cursor != null` is false. The loop ends, and the method returns the answer 3. The complete implementation of the `listLength` method is shown in Figure 4.4.

FIGURE 4.4 A Static Method to Compute the Length of a Linked List

Implementation

```
public static int listLength(IntNode head)
{
    IntNode cursor;
    int answer;

    answer = 0;
    for (cursor = head; cursor != null; cursor = cursor.link)
       answer++;

    return answer;
}
```

Step 2 of the pseudocode

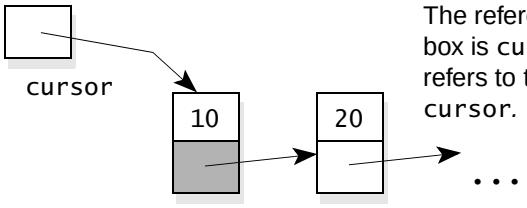
TIP

Programming Tip: How to Traverse a Linked List

You should learn the important pattern for traversing a linked list, as used in the `listLength` method (Figure 4.4). The same pattern can be used whenever you need to step through the nodes of a linked list one at a time.

The first part of the pattern concerns moving from one node to another. Whenever we have a variable that refers to some node, and we want the variable to refer to the next node, we must use the `link` part of the node. Here is the reasoning that we follow:

1. Suppose `cursor` refers to some node;
2. Then `cursor.link` refers to the next node (if there is one), as shown here:



3. To move `cursor` to the next node, we use one of these assignment statements:

```
cursor = cursor.link;
```

or

```
cursor = cursor.getLink( );
```

Use the first version, `cursor = cursor.link`, if you have access to the `link` instance variable (inside one of the `IntNode` methods). Otherwise, use the second version, `cursor = cursor.getLink( )`. In both cases, if there is no next node, then `cursor.link` will be `null`, and therefore our assignment statement will set `cursor` to `null`.

The key is to know that the assignment statement `cursor = cursor.link` moves `cursor` so that it refers to the next node. If there is no next node, then the assignment statement sets `cursor` to `null`.

The second part of the pattern shows how to traverse all of the nodes of a linked list, starting at the head node. The pattern of the loop looks like this:

```
for (cursor = head; cursor != null; cursor = cursor.link)
{
    ...
}
```

Inside the body of the loop, you may carry out whatever computation is needed for a node in the list.

You'll find yourself using this pattern continually in methods that manipulate linked lists.

Pitfall: Forgetting to Test the Empty List

PITFALL

Methods that manipulate linked lists should always be tested to ensure that they have the right behavior for the empty list. When `head` is `null` (indicating the empty list), our `listLength` method should return 0. Testing this case shows that `listLength` does correctly return 0 for the empty list.

Searching for an Element in a Linked List

In Java, a method may return a reference to a node. Hence, when the job of a subtask is to find a single node, it makes sense to implement the subtask as a method that returns a reference to that node. Our next method follows this pattern, returning a reference to a node that contains a specified element. The specification is given here:

- **listSearch**

  `public static IntNode listSearch(IntNode head, int target)`

  Search for a particular piece of data in a linked list.

  Parameters:

  `head` – the head reference for a linked list (which may be an empty list with a null head)

  `target` – a piece of data to search for

  Returns:

  The return value is a reference to the first node that contains the specified target. If there is no such node, the null reference is returned.

As indicated by the return type of `IntNode`, the method returns a reference to a node in a linked list. The node is specified by a parameter named `target`, which is the integer that appears in the sought-after node. For example, the activation `IntNode.listSearch(head, -4)` in Figure 4.5 will return a reference to the shaded node.

Sometimes, the specified target does not appear in the list. In this case, the method returns the null reference.

The implementation of `listSearch` is shown in Figure 4.6. Most of the work is carried out with the usual traversal pattern, using a local variable called `cursor` to step through the nodes one at a time:

```
for (cursor = head; cursor != null; cursor = cursor.link)
{
   if (target == the data in the node that cursor refers to)
      return cursor;
}
```

As the loop executes, `cursor` refers to the nodes of the list, one after another. The test inside the loop determines whether we have found the sought-after node, and if so, then a reference to the node is immediately returned with the

listSearch

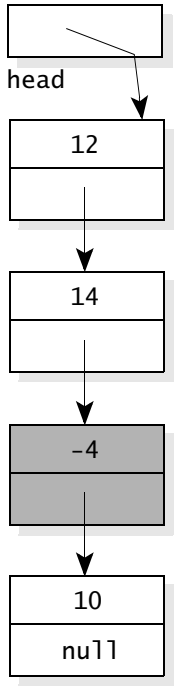


FIGURE 4.5
Example for `listSearch`

FIGURE 4.6 A Static Method to Search for a Target in a Linked List

Implementation

```
public static IntNode listSearch(IntNode head, int target)
{
    IntNode cursor;

    for (cursor = head; cursor != null; cursor = cursor.link)
       if (target == cursor.data)
          return cursor;

    return null;
}
```

return statement `return cursor`. When a return statement occurs like this, inside a loop, the method returns without ado—the loop is not run to completion.

On the other hand, should the loop actually complete by eventually setting `cursor` to `null`, then the sought-after node is not on the list. According to the method's postcondition, the method returns `null` when the node is not on the list. This is accomplished with one more return statement—`return null`—at the end of the method's implementation.

Finding a Node by Its Position in a Linked List

Here's another method that returns a reference to a node in a linked list:

listPosition

- **listPosition**

  `public static IntNode listPosition(IntNode head, int position)`

  Find a node at a specified position in a linked list.

  Parameters:

  `head` – the head reference for a linked list (which may be an empty list with a null head)

  `position` – a node number

  Precondition:

  `position > 0`

  Returns:

  The return value is a reference to the node at the specified position in the list. (The head node is position 1, the next node is position 2, and so on.) If there is no such position (because the list is too short), then the null reference is returned.

  Throws: `IllegalArgumentException`

  Indicates that `position` is not positive.

In this method, a node is specified by giving its position in the list, with the head node at position 1, the next node at position 2, and so on. For example, with the linked list from Figure 4.7, `IntNode.listPosition(head, 3)` will return a reference to the shaded node. Notice that the first node is number 1, not number 0 as in an array. The specified position might also be larger than the length of the list, in which case, the method returns the null reference.

The implementation of `listPosition` is shown in Figure 4.8. It uses a variation of the list traversal technique that we have already seen. The variation is useful when we want to move to a particular node in a linked list and we know the ordinal position of the node (such as position number 1, position number 2, and so on). We start by setting a reference variable, `cursor`, to the head node of the list. A loop then moves the `cursor` forward the correct number of spots, as shown here:

```
cursor = head;
for (i = 1; (i < position) && (cursor != null); i++)
    cursor = cursor.link;
```

Each iteration of the loop executes `cursor = cursor.link` to move the `cursor` forward one node. Normally, the loop stops when `i` reaches `position`, and `cursor` refers to the correct node. The loop can also stop if `cursor` becomes `null`, indicating that `position` was larger than the number of nodes on the list.

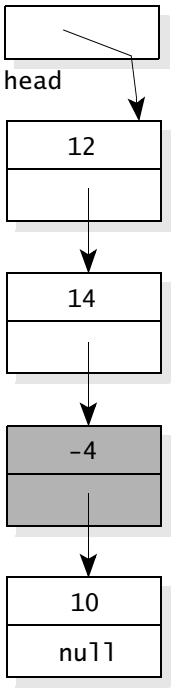


FIGURE 4.7
Example for `listPosition`

FIGURE 4.8 A Static Method to Find a Particular Position in a Linked List

Implementation

```
public static IntNode listPosition(IntNode head, int position)
{
   IntNode cursor;
   int i;

   if (position <= 0)
      throw new IllegalArgumentException("position is not positive.");

   cursor = head;
   for (i = 0; (i < position) && (cursor != null); i++)
      cursor = cursor.link;

   return cursor;
}
```

Copying a Linked List

Our next static method makes a copy of a linked list, returning a head reference for the newly created copy. Here is the specification:

listCopy

- **listCopy**

  `public static IntNode listCopy(IntNode source)`

  Copy a list.

  Parameters:

  `source`–the head reference for a linked list that will be copied (which may be an empty list where `source` is null)

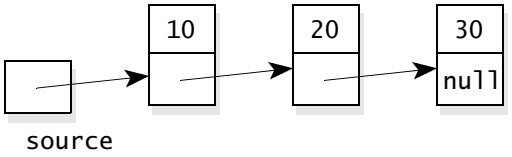
  Returns:

  The method has made a copy of the linked list starting at `source`. The return value is the head reference for the copy.

  Throws: `OutOfMemoryError`

  Indicates that there is insufficient memory for the new list.

For example, suppose that `source` refers to the following list:



The `listCopy` method creates a completely separate copy of the three-node list. The copy of the list has its own three nodes, which also contain the numbers 10, 20, and 30. The return value is a head reference for the new list, and the original list remains intact.

The pseudocode begins by handling one special case—the case where the original list is the empty list (so that `source` is null). In this case the method simply returns null, indicating that its answer is the empty list. So, the first step of the pseudocode is:

1. `if (source == null)`, then return `null`.

After dealing with the special case, the method uses two local variables called `copyHead` and `copyTail`, which will be maintained as the head and tail references for the new list. The pseudocode for creating this new list is given in the next three steps:

2. Create a new node for the head node of the new list that we are creating. Make both `copyHead` and `copyTail` refer to this new node, which contains the same data as the head node of the source list.

3. Make `source` refer to the second node of the original list, then the third node, then the fourth node, and so on until we have traversed all of the original list. At each node that `source` refers to, add one new node to the tail of the new list, and move `copyTail` forward to the newly added node, as follows:

```
copyTail.addNodeAfter(source.data);
copyTail = copyTail.link;
```

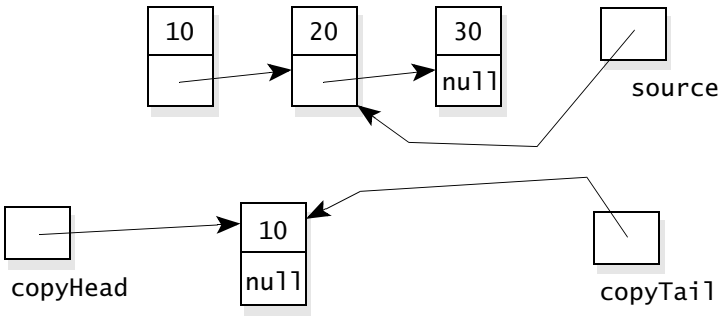
4. After Step 3 completes, return `copyHead` (a reference to the head node of the list that we created).

Step 3 of the pseudocode is completely implemented by this loop:

```
while (source.link != null)
{  // There are more nodes, so copy the next one.
   source = source.link;
   copyTail.addNodeAfter(source.data);
   copyTail = copyTail.link;
}
```

The while-loop starts by checking `source.link != null` to determine whether there is another node to copy. If there is another node, then we enter the body of the loop and move `source` forward with the assignment statement `source = source.link`. The second and third statements in the loop add a node at the tail end of the newly created list and move `copyTail` forward.

As an example, consider again the three-node list with data 10, 20, and 30. The first two steps of the pseudocode are carried out and then we enter the body of the while-loop. We execute the first statement of the loop: `source = source.link`. At this point, the variables look like this:

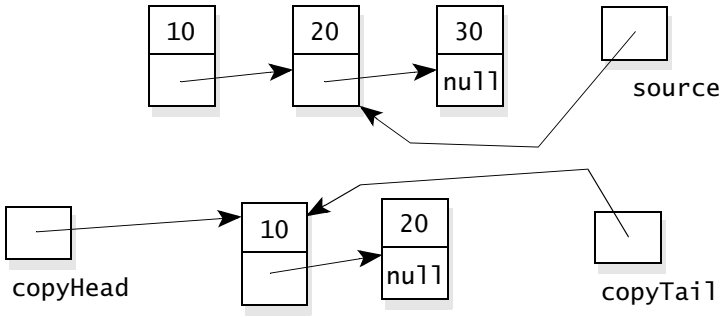


Notice that we have already copied the first node of the linked list. During the first iteration of the while-loop, we will copy the second node of the linked

list—the node that is now referred to by `source`. The first part of copying the node works by activating one of our other methods, `addNodeAfter`, as shown here:

```
copyTail.addNodeAfter(source.data);
```

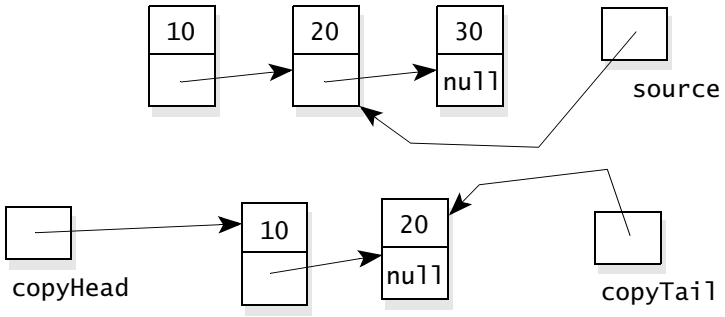
This activation adds a new node to the end of the list that we are creating (i.e., *after* the node referred to by `copyTail`), and the data in the new node is the number 20 (i.e., the data from `source.data`). Immediately after adding the new node, the variables look like this:



The last statement in the while-loop body moves `copyTail` forward to the new tail of the new list, as shown here:

```
copyTail = copyTail.link;
```

This is the usual way that we make a node reference "move to the next node," as we have seen in other methods such as `listSearch`. After moving `copyTail`, the variables look like this:



In this example, the body of the while-loop will execute one more time to copy the third node to the new list. Then the loop will end, and the method returns the new head reference, `copyHead`.

The complete implementation of `listCopy` is shown in Figure 4.9.

FIGURE 4.9 A Static Method to Copy a Linked List

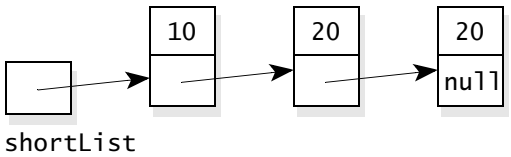
Implementation

```
public static IntNode listCopy(IntNode source)
{
   IntNode copyHead;
   IntNode copyTail;

   // Handle the special case of an empty list.
   if (source == null)
      return null;

   // Make the first node for the newly created list.
   copyHead = new IntNode(source.data, null);
   copyTail = copyHead;

   // Make the rest of the nodes for the newly created list.
   while (source.link != null)
   {
      source = source.link;
      copyTail.addNodeAfter(source.data);
      copyTail = copyTail.link;
   }

   // Return the head reference for the new list.
   return copyHead;
}
```

Here's an example of how the `listCopy` method might be used in a program:

```
IntNode shortList;
IntNode copy;

shortList = new IntNode(10, null);
shortList.addNodeAfter(20);
shortList.addNodeAfter(20);
```

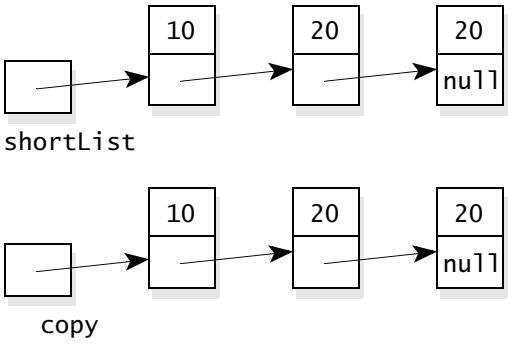
At this point, `shortList` is the head of a small list shown here:



We could now use `listCopy` to make a second copy of this list:

```
copy = IntNode.listCopy(shortList);
```

At this point, we have two separate lists:



why is listCopy a static method?

Keep in mind that `listCopy` is a static method, so we must write the expression `IntNode.listCopy(shortList)` rather than `shortList.listCopy( )`. This may seem strange—why not make `listCopy` an ordinary method? The answer is that an ordinary method could not copy the empty list (because the empty list is represented by the null reference).

A Second Copy Method, Returning Both Head and Tail References

We're going to have a second way to copy a list, with a slightly different specification, shown here:

listCopyWithTail

- **listCopyWithTail**

  `public static IntNode[ ] listCopyWithTail(IntNode source)`

  Copy a list, returning both a head and tail reference for the copy.

  Parameters:

  `source` – the head reference for a linked list that will be copied (which may be an empty list where `source` is null)

  Returns:

  The method has made a copy of the linked list starting at `source`. The return value is an array where the `[0]` element is a head reference for the copy and the `[1]` element is a tail reference for the copy.

  Throws: `OutOfMemoryError`

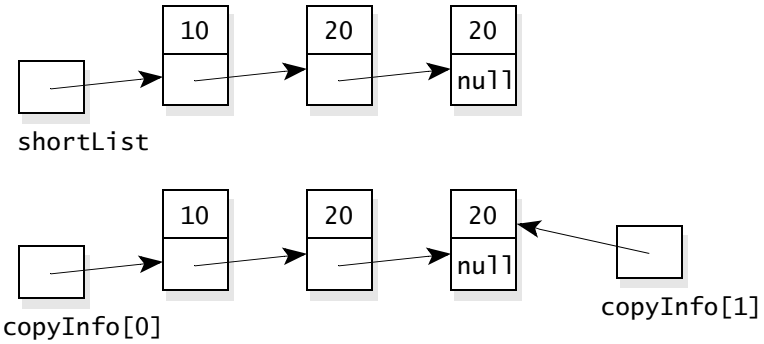
  Indicates that there is insufficient memory for the new list.

The `listCopyWithTail` makes a copy of a list, but the return value is more than a head reference for the copy. Instead, the return value is an array with two components. The [0] component of the array contains the head reference for the new list, and the [1] component contains the tail reference for the new list. The `listCopyWithTail` method is important because many algorithms must copy a list and obtain access to both the head and tail nodes of the copy.

As an example, a program can create a small list, then create a copy with both a head and tail reference for the copy:

```
IntNode shortList;
IntNode copyInfo[ ];

shortList = new IntNode(10, null);
shortList.addNodeAfter(20);
shortList.addNodeAfter(20);
copyInfo = IntNode.listCopyWithTail(source);
```

At this point, `copyInfo[0]` is the head reference for a copy of the short list, and `copyInfo[1]` is the tail reference for the same list, as shown here:



The implementation of `listCopyWithTail` is shown in the first part of Figure 4.10. It's nearly the same as `listCopy`, except there is an extra local variable called `answer`, which is an array of two `IntNode` components. These two components are set to the head and tail of the new list, and the method finishes with the return statement: `return answer`.

Programming Tip: A Method Can Return an Array

TIP

The return value from a method can be an array. This is useful if the method returns more than one piece of information. For example, `listCopyWithTail` returns an array with two components containing the head and tail references for a new list.

FIGURE 4.10 A Second Static Method to Copy a Linked List

Implementation

```
public static IntNode[ ] listCopyWithTail(IntNode source)
{
   // Notice that the return value is an array of two IntNode components.
   // The [0] component is the head reference for the new list and
   // the [1] component is the tail reference for the new list.
   // Also notice that the answer array is automatically initialized to contain
   // two null values. Arrays with components that are references are always
   // initialized this way.
   IntNode copyHead;
   IntNode copyTail;
   IntNode[ ] answer = new IntNode[2];

   // Handle the special case of an empty list.
   if (source == null)
      return answer; // The answer has two null references.

   // Make the first node for the newly created list.
   copyHead = new IntNode(source.data, null);
   copyTail = copyHead;

   // Make the rest of the nodes for the newly created list.
   while (source.link != null)
   {
      source = source.link;
      copyTail.addNodeAfter(source.data);
      copyTail = copyTail.link;
   }

   // Return the head and tail reference for the new list.
   answer[0] = copyHead;
   answer[1] = copyTail;
   return answer;
}
```

Copying Part of a Linked List

Sometimes a program needs to copy only part of a linked list rather than the entire list. The task can be done by a static method, `listPart`, which copies part of a list, as specified next.

listPart

- **listPart**

```
public static IntNode[ ] listPart
  (IntNode start, IntNode end)
```

Copy part of a list, providing a head and tail reference for the new copy.

Parameters:
`start` and `end` – references to two nodes of a linked list

Precondition:
`start` and `end` are non-null references to nodes on the same linked list, with the `start` node at or before the `end` node.

Returns:
The method has made a copy of part of a linked list, from the specified `start` node to the specified `end` node. The return value is an array where the `[0]` component is a head reference for the copy and the `[1]` component is a tail reference for the copy.

Throws: `IllegalArgumentException`
Indicates that `start` and `end` do not satisfy the precondition.

Throws: `OutOfMemoryError`
Indicates that there is insufficient memory for the new list.

The `listPart` implementation is given as part of the complete `IntNode` class in Figure 4.11 on page 204. In all, there is one constructor, five ordinary methods, and six static methods. The class is placed in a package called `edu.colorado.nodes`.

Using Linked Lists

Any program can use linked lists created from our nodes. Such a program must have this import statement:

```
import edu.colorado.nodes.IntNode;
```

nodes with different kinds of data

The program can then use the various methods to build and manipulate linked lists. In fact, the `edu.colorado.nodes` package includes many different kinds of nodes: `IntNode`, `DoubleNode`, `CharNode`, etc. You can get these classes from `http://www.cs.colorado.edu/~main/edu/colorado/nodes`. (There is also a special kind of node that can handle many different kinds of data, but you'll have to wait until Chapter 5 for that.)

For a programmer to use our nodes, the programmer must have some understanding of linked lists and our specific nodes. It might be better if we use the node classes ourselves to build various collection classes. The different collection classes that we build can be used by any programmer, with no knowledge of nodes and linked lists. This is what we will do in the rest of the chapter, providing two ADTs that use the linked lists.

FIGURE 4.11 Specification and Implementation of the `IntNode` Class

Class IntNode

❖ **public class IntNode from the package edu.colorado.nodes**

An `IntNode` provides a node for a linked list with integer data in each node. Lists can be of any length, limited only by the amount of free memory on the heap. But beyond `Integer.MAX_VALUE`, the answer from `listLength` is incorrect because of arithmetic overflow.

Specification

◆ **Constructor for the IntNode**

`public IntNode(int initialData, IntNode initialLink)`

Initialize a node with a specified initial data and link to the next node. Note that the `initialLink` may be the null reference, which indicates that the new node has nothing after it.

Parameters:

`initialData` – the initial data of this new node

`initialLink` – a reference to the node after this new node—this reference may be null to indicate that there is no node after this new node.

Postcondition:

This new node contains the specified data and link to the next node.

◆ **addNodeAfter**

`public void addNodeAfter(int element)`

Modification method to add a new node after this node.

Parameters:

`element` – the data to be placed in the new node

Postcondition:

A new node has been created and placed after this node. The data for the new node is `element`. Any other nodes that used to be after this node are now after the new node.

Throws: `OutOfMemoryError`

Indicates that there is insufficient memory for a new `IntNode`.

◆ **getData**

`public int getData( )`

Accessor method to get the data from this node.

Returns:

the data from this node

◆ **getLink**

`public IntNode getLink( )`

Accessor method to get a reference to the next node after this node.

Returns:

a reference to the node after this node (or the null reference if there is nothing after this node)

(continued)

(FIGURE 4.11 continued)

- **listCopy**

  `public static IntNode listCopy(IntNode source)`

  Copy a list.

  Parameters:

  `source` – the head reference for a linked list that will be copied (which may be an empty list where `source` is null)

  Returns:

  The method has made a copy of the linked list starting at `source`. The return value is the head reference for the copy.

  Throws: `OutOfMemoryError`

  Indicates that there is insufficient memory for the new list.

- **listCopyWithTail**

  `public static IntNode[ ] listCopyWithTail(IntNode source)`

  Copy a list, returning both a head and tail reference for the copy.

  Parameters:

  `source` – the head reference for a linked list that will be copied (which may be an empty list where `source` is null)

  Returns:

  The method has made a copy of the linked list starting at `source`. The return value is an array where the `[0]` element is a head reference for the copy and the `[1]` element is a tail reference for the copy.

  Throws: `OutOfMemoryError`

  Indicates that there is insufficient memory for the new list.

- **listLength**

  `public static int listLength(IntNode head)`

  Compute the number of nodes in a linked list.

  Parameters:

  `head` – the head reference for a linked list (which may be an empty list with a null head)

  Returns:

  the number of nodes in the list with the given head

  Note:

  A wrong answer occurs for lists longer than `Int.MAX_VALUE`, because of arithmetic overflow.

(continued)

(FIGURE 4.11 continued)

- **listPart**

  `public static IntNode[ ] listPart(IntNode start, IntNode end)`

  Copy part of a list, providing a head and tail reference for the new copy.

  Parameters:

  `start` and `end` – references to two nodes of a linked list

  Precondition:

  `start` and `end` are non-null references to nodes on the same linked list, with the `start` node at or before the `end` node.

  Returns:

  The method has made a copy of part of a linked list, from the specified `start` node to the specified `end` node. The return value is an array where the `[0]` component is a head reference for the copy and the `[1]` component is a tail reference for the copy.

  Throws: `IllegalArgumentException`

  Indicates that `start` and `end` do not satisfy the precondition.

  Throws: `OutOfMemoryError`

  Indicates that there is insufficient memory for the new list.

- **listPosition**

  `public static IntNode listPosition(IntNode head, int position)`

  Find a node at a specified position in a linked list.

  Parameters:

  `head` – the head reference for a linked list (which may be an empty list with a null head)
  `position` – a node number

  Precondition:

  `position > 0`

  Returns:

  The return value is a reference to the node at the specified position in the list. (The head node is position 1, the next node is position 2, and so on.) If there is no such position (because the list is too short), then the null reference is returned.

  Throws: `IllegalArgumentException`

  Indicates that `position` is zero.

- **listSearch**

  `public static IntNode listSearch(IntNode head, int target)`

  Search for a particular piece of data in a linked list.

  Parameters:

  `head` – the head reference for a linked list (which may be an empty list with a null head)
  `target` – a piece of data to search for

  Returns:

  The return value is a reference to the first node that contains the specified target. If there is no such node, the null reference is returned.

(continued)

(FIGURE 4.11 continued)

- **removeNodeAfter**

  `public void removeNodeAfter( )`

  Modification method to remove the node after this node.

  Precondition:
  This node must not be the tail node of the list.

  Postcondition:
  The node after this node has been removed from the linked list. If there were further nodes after that one, they are still present on the list.

  Throws: `NullPointerException`
  Indicates that this was the tail node of the list, so there is nothing after it to remove.

- **setData**

  `public void setData(int newdata)`

  Modification method to set the data in this node.

  Parameters:
  `newData` – the new data to place in this node

  Postcondition:
  The data of this node has been set to `newData`.

- **setLink**

  `public void setLink(IntNode newLink)`

  Modification method to set a reference to the next node after this node.

  Parameters:
  `newLink` – a reference to the node that should appear after this node in the linked list (or the null reference if there should be no node after this node)

  Postcondition:
  The link to the node after this node has been set to `newLink`. Any other node (that used to be in this link) is no longer connected to this node.

Implementation

```
// File: IntNode.java from the package edu.colorado.nodes
// Documentation is available from pages 204–207 or from the IntNode link in
//   http://www.cs.colorado.edu/~main/docs/

package edu.colorado.nodes;
```

(continued)

(FIGURE 4.11 continued)

```
public class IntNode
{
   // Invariant of the IntNode class:
   //   1. The node's integer data is in the instance variable data.
   //   2. For the final node of a list the link part is null.
   //      Otherwise, the link part is a reference to the next node of the list.
   private int data;
   private IntNode link;

   public IntNode(int initialData, IntNode initialLink)
   {
      data = initialData;
      link = initialLink;
   }

   public void addNodeAfter(int element)
   {
      link = new IntNode(element, link);
   }

   public int getData( )
   {
      return data;
   }

   public IntNode getLink( )
   {
      return link;
   }

   public static IntNode listCopy(IntNode source)
   || See the implementation in Figure 4.9 on page 199.

   public static IntNode[ ] listCopyWithTail(IntNode source)
   || See the implementation in Figure 4.10 on page 202.

   public static int listLength(IntNode head)
   || See the implementation in Figure 4.4 on page 191.
```

(continued)

(FIGURE 4.11 continued)

```
public static IntNode[ ] listPart(IntNode start, IntNode end)
{
   // Notice that the return value is an array of two IntNode components.
   // The [0] component is the head reference for the new list and
   // the [1] component is the tail reference for the new list.
   IntNode copyHead;
   IntNode copyTail;
   IntNode[ ] answer = new IntNode[2];

   // Check for illegal null at start or end.
   if (start == null)
      throw new IllegalArgumentException("start is null");
   if (end == null)
      throw new IllegalArgumentException("end is null");

   // Make the first node for the newly created list.
   copyHead = new IntNode(start.data, null);
   copyTail = copyHead;

   // Make the rest of the nodes for the newly created list.
   while (start != end)
   {
      start = start.link;
      if (start == null)
         throw new IllegalArgumentException
         ("end node was not found on the list");
      copyTail.addNodeAfter(start.data);
      copyTail = copyTail.link;
   }

   // Return the head and tail reference for the new list.
   answer[0] = copyHead;
   answer[1] = copyTail;
   return answer;
}

public static IntNode listPosition(IntNode head, int position)
|| See the implementation in Figure 4.8 on page 195.

public static IntNode listSearch(IntNode head, int target)
|| See the implementation in Figure 4.6 on page 194.
```

(continued)

(FIGURE 4.11 continued)

```
    public void removeNodeAfter( )
    {
       link = link.link;
    }

    public void setData(int newData)
    {
       data = newData;
    }

    public void setLink(IntNode newLink)
    {
       link = newLink;
    }
}
```

Self-Test Exercises

9. Look at `http://www.cs.colorado.edu/~main/edu/colorado/nodes`. How many different kinds of nodes are there? If you implemented one of these nodes, what extra work would be required to implement another?
10. Suppose that `locate` is a reference to a node in a linked list (and it is not the null reference). Write an assignment statement that will make `locate` move to the next node in the list. You should write two versions of the assignment—one that can appear in the `IntNode` class itself, and another that can appear outside of the class. What do your assignment statements do if `locate` was already refering to the last node in the list?
11. Which of the node methods use `new` to allocate at least one new node? Check your answer by looking at the documentation in Figure 4.11 on page 204 (to see which methods can throw an `OutOfMemoryError`).
12. Suppose that `head` is a head reference for a linked list with just one node. What will `head` be after `head = head.getLink( )`?
13. What technique would you use if a method needs to return more than one `IntNode`, such as a method that returns both a head and tail reference for a list.
14. Suppose that `head` is a head reference for a linked list. Also suppose that `douglass` and `adams` are two other `IntNode` variables. Write one assignment statement that will make `douglass` refer to the first node in the list that contains the number 42. Write a second assignment statement that will make `adams` refer to the 42nd node of the list. If these nodes don't exist, then the assignments should set the variables to null.

4.4 THE BAG ADT WITH A LINKED LIST

We're ready to write an ADT that is implemented with a linked list. We'll start with the familiar bag ADT, which we have previously implemented with an array (Section 3.2). At the end of this chapter we'll compare the advantages and disadvantages of these different implementations. But first, let's see how a linked list is used in our second bag implementation.

Our Second Bag—Specification

The advantage of using a familiar ADT is that you already know most of the specification. The specification, given in Figure 4.12, is nearly identical to our previous bag. The major difference is that our new bag has no worries about capacity: There is no initial capacity and no need for an `ensureCapacity` method. This is because our planned implementation—storing the bag's elements on a linked list—can easily grow and shrink by adding and removing nodes from the linked list.

the new bag class is called IntLinkedBag

The new bag class will be called `IntLinkedBag`, meaning that the underlying elements are integers, the implementation will use a linked list, and the collection itself is a bag. The `IntLinkedBag` will be placed in the same package that we used in Chapter 3—`edu.colorado.collections`, as shown in the specification of Figure 4.12.

FIGURE 4.12 Specification and Implementation of the `IntLinkedBag` Class

Class IntLinkedBag

❖ **public class IntLinkedBag from the package edu.colorado.collections**

An `IntLinkedBag` is a collection of `int` numbers.

Limitations:

(1) Beyond `Int.MAX_VALUE` elements, `countOccurrences`, `size` and `grab` are wrong.

(2) Because of the slow linear algorithms of this class, large bags have poor performance.

Specification

◆ **Constructor for the IntLinkedBag**

```
public IntLinkedBag( )
```

Initialize an empty bag.

Postcondition:

This bag is empty.

(continued)

(FIGURE 4.12 continued)

- **add**

  `public void add(int element)`

  Add a new element to this bag.

  Parameters:
  `element` – the new element that is being added

  Postcondition:
  A new copy of the element has been added to this bag.

  Throws: `OutOfMemoryError`
  Indicates insufficient memory for adding a new element.

- **addAll**

  `public void addAll(IntLinkedBag addend)`

  Add the contents of another bag to this bag.

  Parameters:
  `addend` – a bag whose contents will be added to this bag

  Precondition:
  The parameter, `addend`, is not null.

  Postcondition:
  The elements from `addend` have been added to this bag.

  Throws: `NullPointerException`
  Indicates that `addend` is null.

  Throws: `OutOfMemoryError`
  Indicates insufficient memory to increase the size of this bag.

- **clone**

  `public Object clone( )`

  Generate a copy of this bag.

  Returns:
  The return value is a copy of this bag. Subsequent changes to the copy will not affect the original, nor vice versa. The return value must be typecast to an `IntLinkedBag` before it is used.

  Throws: `OutOfMemoryError`
  Indicates insufficient memory for creating the clone.

- **countOccurrences**

  `public int countOccurrences(int target)`

  Accessor method to count the number of occurrences of a particular element in this bag.

  Parameters:
  `target` – the element that needs to be counted

  Returns:
  the number of times that `target` occurs in this bag

(continued)

(FIGURE 4.12 continued)

- **grab**

  `public int grab( )`

  Accessor method to retrieve a random element from this bag.

  Precondition:
  This bag is not empty.

  Returns:
  a randomly selected element from this bag

  Throws: `IllegalStateException`
  Indicates that the bag is empty.

- **remove**

  `public boolean remove(int target)`

  Remove one copy of a specified element from this bag.

  Parameters:
  `target` – the element to remove from the bag

  Postcondition:
  If `target` was found in this bag, then one copy of `target` has been removed and the method returns `true`. Otherwise this bag remains unchanged and the method returns `false`.

- **size**

  `public int size( )`

  Accessor method to determine the number of elements in this bag.

  Returns:
  the number of elements in this bag

- **union**

  `public static IntLinkedBag union(IntLinkedBag b1, IntLinkedBag b2)`

  Create a new bag that contains all the elements from two other bags.

  Parameters:
  `b1` – the first of two bags
  `b2` – the second of two bags

  Precondition:
  Neither `b1` nor `b2` is null.

  Returns:
  a new bag that is the union of `b1` and `b2`

  Throws: `NullPointerException`
  Indicates that one of the arguments is null.

  Throws: `OutOfMemoryError`
  Indicates insufficient memory for the new bag.

The grab Method

The new bag has one other minor change, which is specified as part of Figure 4.12. Just for fun, we've included a new method called `grab`, which returns a randomly selected element from a bag. Later we'll use the `grab` method in some game-playing programs.

Our Second Bag—Class Declaration

Our plan has been laid. We will implement the new bag by storing the elements on a linked list. The class will have two private instance variables: (1) a reference to the head of a linked list that contains the elements of the bag; and (2) an `int` variable that keeps track of the length of the list. The second instance variable isn't really needed since we could use `listLength` to determine the length of the list. But by keeping the length in an instance variable, the length can be quickly determined by accessing the variable (a constant time operation). This is in contrast to actually counting the length by traversing the list (a linear time operation).

In any case, we can now write an outline for our implementation. The class goes in the `package edu.colorado.collections`, and we import the node class from `edu.colorado.nodes.IntNode`. Then we declare our new bag class with two instance variables as shown here:

```
package edu.colorado.collections;
import edu.colorado.nodes.IntNode;

class IntLinkedBag
{
   private IntNode head;      // Head reference for the list
   private int manyNodes;     // Number of nodes on the list

   Method implementations will be placed here later.
}
```

To avoid confusion over how we are using our linked list, we now make an explicit statement of the invariant for our second design of the bag ADT.

Invariant for the Second Bag ADT

1. The elements in the bag are stored on a linked list.
2. The head reference of the list is stored in the instance variable `head`.
3. The total number of elements in the list is stored in the instance variable `manyNodes`.

The Second Bag—Implementation

With our invariant in mind, we can implement each of the methods, starting with the constructor. The key to simple implementations is to use the node methods whenever possible.

Constructor. The constructor sets `head` to be the null reference (indicating the empty list) and sets `manyNodes` to zero. Actually, these two values (null and zero) are the default values for the instance variables, so one possibility is to not implement the constructor at all. When we implement no constructor, Java provides an automatic no-arguments constructor that initializes all instance variables to their default values. If we take this approach, then our implementation should include a comment to indicate that we are using Java's automatic no-arguments constructor.

However, we will actually implement the constructor, as shown here:

constructor

```
public IntLinkedBag( )
{
   head = null;
   manyNodes = 0;
}
```

Having an actual implementation makes it easier to make future changes. Also, without the implementation, we could not include a Javadoc comment to specify exactly what the constructor does.

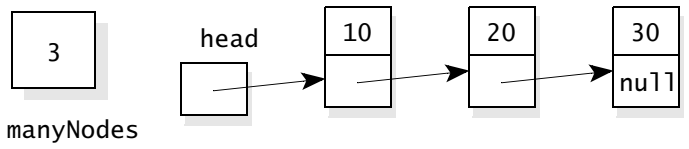
The `clone` Method. The `clone` method needs to create a copy of a bag. The `IntLinkedBag` `clone` method will follow the pattern introduced in Chapter 2 on page 78. Therefore, the start of the `clone` method is the code shown here:

clone method

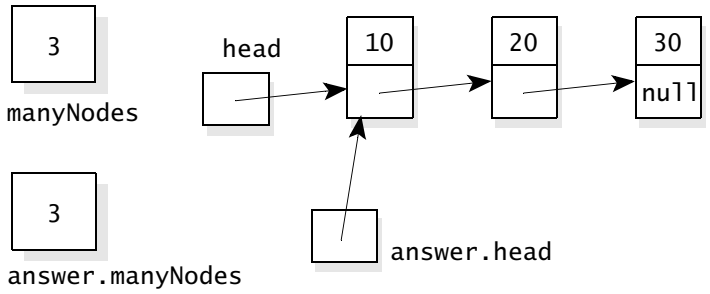
```
public Object clone( )
{  // Clone an IntLinkedBag.
   IntLinkedBag answer;

   try
   {
      answer = (IntLinkedBag) super.clone( );
   }
   catch (CloneNotSupportedException e)
   {  // This exception should not occur. But if it does, it would
      // probably indicate a programming error that made
      // super.clone unavailable. The most common error would be
      // forgetting the "Implements Cloneable" clause.
      throw new RuntimeException
      ("This class does not implement Cloneable.");
   }
   ...
```

This is the same as the start of the `clone` method for our Chapter 3 bag. As with the Chapter 3 bag, this code uses the `super.clone` method to make `answer` be an exact copy of the bag that activated the `clone` method. With the Chapter 3 bag, we needed some extra statements at the end of the `clone` method—otherwise the original bag and the clone would share the same array.

Our new bag, using a linked list, runs into a similar problem. To see this problem, consider a bag that contains three elements, as shown here:



Now, suppose we activate `clone( )` to create a copy of this bag. The clone method executes the statement `answer = (IntLinkedBag) super.clone( )`. What does `super.clone( )` do? It creates a new `IntLinkedBag` object and `answer` will refer to this new `IntLinkedBag`. But the new `IntLinkedBag` has instance variables (`answer.manyNodes` and `answer.head`) that are merely copied from the original. So, after `answer = (IntLinkedBag) super.clone( )`, the situation looks like this (where `manyNodes` and `head` are the instance variables from the original bag that activated the `clone` method):

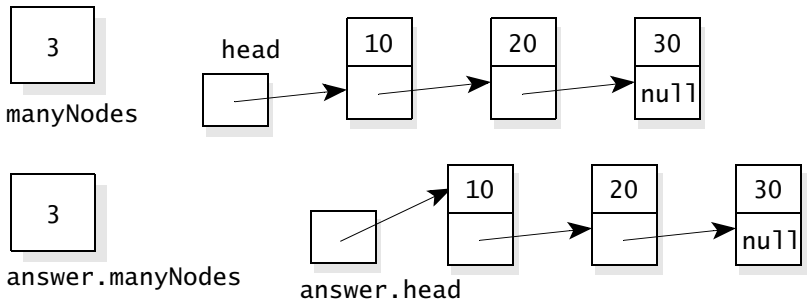


As you can see, `answer.head` refers to the original's head node. Subsequent changes to the linked list will affect both the original and the clone. This is incorrect behavior for a clone. To fix the problem, we need an additional statement before the return of the `clone` method. The purpose of the statement is to create a new linked list for the clone's `head` instance variable to refer to. Here's the statement:

```
answer.head = IntNode.listCopy(head);
```

This statement activates the `listCopy` method. The argument, `head`, is the head reference from the linked list of the bag that we are copying. When the assign-

ment statement finishes, `answer.head` will refer to the head node of the new list. Here's the situation after the copying:



The new linked list for `answer` was created by copying the original linked list. Subsequent changes to `answer` will not affect the original, nor will changes to the original affect `answer`. The complete `clone` method, including the extra statement at the end, is shown in Figure 4.13.

FIGURE 4.13 Implementation of the Second Bag's `clone` Method

Implementation

```
public Object clone( )
{
   { // Clone an IntLinkedBag.
   IntLinkedBag answer;

   try
   {
      answer = (IntLinkedBag) super.clone( );
   }
   catch (CloneNotSupportedException e)
   {  // This exception should not occur. But if it does, it would probably indicate a
      // programming error that made super.clone unavailable. The most common
      // error would be forgetting the "Implements Cloneable"
      // clause at the start of this class.
      throw new RuntimeException
      ("This class does not implement Cloneable.");
   }

   answer.head = IntNode.listCopy(head);

   return answer;
}
```

This step creates a new linked list for answer. The new linked list is separate from the original array so that subsequent changes to one will not affect the other.

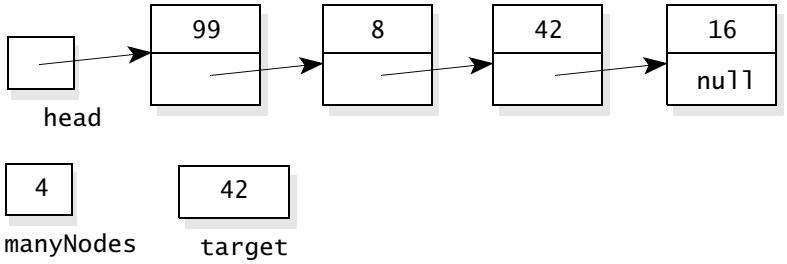
TIP

Programming Tip: Cloning a Class That Contains a Linked List

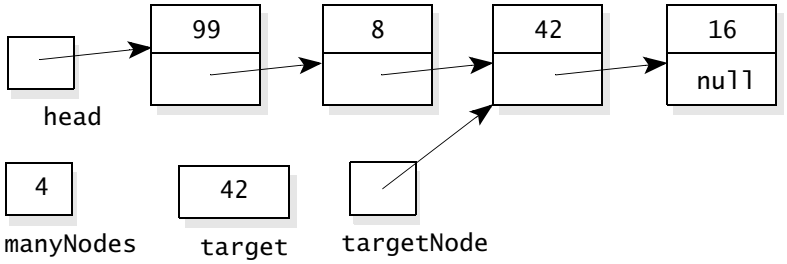
If the instance variables of a class contain a linked list, then the clone method needs extra work before it returns. The extra work creates a new linked list for the clone's instance variable to refer to.

The `remove` Method. There are two approaches to implementing the `remove` method. The first approach uses the nodes removal methods—changing the head if the removed element is at the head of the list, and using the ordinary `removeNodeAfter` to remove an element that is farther down the line. This first approach is fine, although it does require a bit of thought because `removeNodeAfter` requires a reference to the node that is just *before* the element that you want to remove. We could certainly find this "before" node, but not by using the node's `listSearch` method.

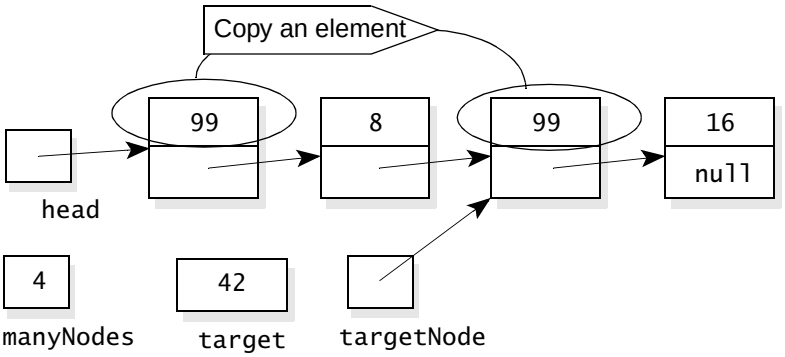
The second approach actually uses `listSearch` to obtain a reference to the node that contains the element to be deleted. For example, suppose our target is the number 42 in the bag shown here:



Our approach begins by setting a local variable named `targetNode` to refer to the node that contains our target. This is accomplished with the assignment `targetNode = listSearch(head, target)`. After the assignment, the `targetNode` is set this way:



Now we can remove the target from the list with two more steps. First, copy the data from the head node to the target node, as shown at the top of the next page.



After this step, we have certainly removed the target, but we are left with two 99s. So, we proceed to a second step: Remove the head node (that is, one of the copies of 99). These steps are all implemented in the `remove` method shown in Figure 4.14. The only other steps in the implementation are a test to ensure that the target is actually in the bag and subtracting one from `manyNodes`.

FIGURE 4.14 A Method to Remove an Element from the Second Version of the Bag

Implementation

```
public boolean remove(int target)
{
   IntNode targetNode; // The node that contains the target

   targetNode = IntNode.listSearch(head, target);
   if (targetNode == null)
      // The target was not found, so nothing is removed.
      return false;
   else
   {  // The target was found at targetNode. So copy the head data to targetNode
      // and then remove the extra copy of the head data.
      targetNode.setData(head.getData( ));
      head = head.getLink( );
      manyNodes--;
      return true;
   }
}
```

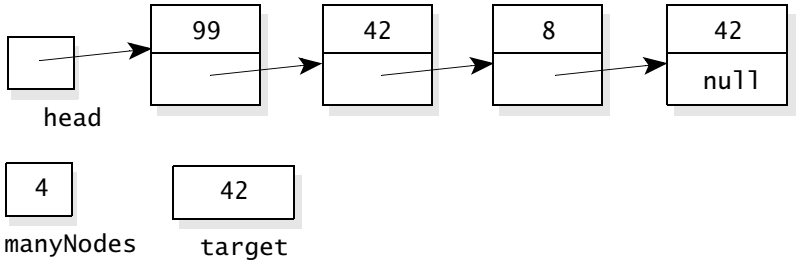
TIP

Programming Tip: How to Choose between Different Approaches

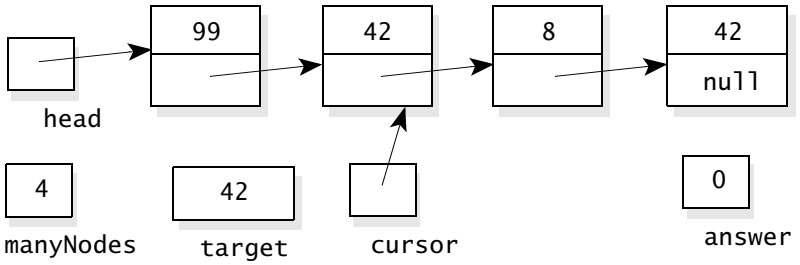
We had two possible approaches for the `remove` method. How do we select the better approach? Normally, when two different approaches have equal efficiency, we will choose the approach that makes better use of the node's methods. This saves us work and also reduces the chance of new errors from writing new code to do an old job. In the case of `remove` we choose the second approach because it made better use of `listSearch`.

The `countOccurrences` Method. Two possible approaches come to mind for the `countOccurrences` method. One of the approaches simply steps through the linked list one node at a time, checking each piece of data to see whether it is the sought-after target. We count the occurrences of the target and return the answer. The second approach uses `listSearch` to find the first occurrence of the target, then uses `listSearch` again to find the next occurrence, and so on until we have found all occurrences of the target. The second approach makes better use of the node's methods, so that is the approach we will take.

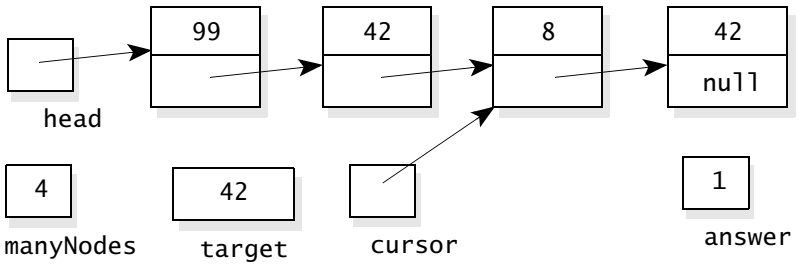
As an example of the second approach to the `countOccurrences` method, suppose we want to count the number of occurrences of 42 in the bag shown here:



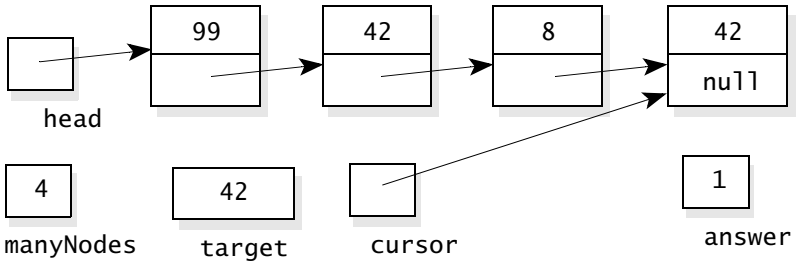
We'll use two local variables: `answer`, which keeps track of the number of occurrences that we have seen so far, and `cursor`, which is a reference to a node in the list. We initialize `answer` to zero, and we use `listSearch` to make `cursor` refer to the first occurrence of the target (or to be `null` if there are no occurrences). After this initialization, we have this situation:



Next, we enter a loop. The loop stops when `cursor` becomes `null`, indicating that there are no more occurrences of the target. Each time through the loop we do two steps: (1) Add one to `answer`, and (2) Move `cursor` to refer to the next occurrence of the target (or to be `null` if there are no more occurrences). Can we use a node method to execute Step 2? At first, it might seem that the node methods are of no use, since `listSearch` finds the *first* occurrence of a given target. But there is an approach that will use `listSearch` together with the `cursor` to find the *next* occurrence of the target. The approach begins by moving cursor to the next node in the list, using the statement `cursor = cursor.getLink( )`. In our example, this results in the following:



As you can see, `cursor` now refers to a node in the middle of a linked list. But, any time that a variable refers to a node in the middle of a linked list, we can pretend that the node is the head of a smaller linked list. In our example, `cursor` refers to the head of a two-element list containing the numbers 8 and 42. Therefore, we can use `cursor` as an argument to `listSearch` in the assignment statement `cursor = IntNode.listSearch(cursor, target)`. This statement moves `cursor` to the next occurrence of the target. This occurrence could be at the cursor's current spot, or it could be farther down the line. In our example, the next occurrence of 42 is farther down the line, so `cursor` is moved as shown here:



Eventually there will be no more occurrences of the target and `cursor` becomes `null`, ending the loop. At that point the method returns `answer`. The complete implementation of `countOccurrences` is shown in Figure 4.15.

FIGURE 4.15 Implementation of countOccurrences for the Second Version of the Bag

Implementation

```
public int countOccurrences(int target)
{
   int answer;
   IntNode cursor;

   answer = 0;
   cursor = IntNode.listSearch(head, target);
   while (cursor != null)
   {  // Each time that cursor is not null, we have another occurrence of target, so we
      // add one to answer and then move cursor to the next occurrence of the target.
      answer++;
      cursor = cursor.getLink( );
      cursor = IntNode.listSearch(cursor, target);
   }
   return answer;
}
```

Finding the Next Occurrence of an Element

The situation: A variable named `cursor` refers to a node in a linked list that contains a particular element called `target`.

The task: Make `cursor` refer to the next occurrence of `target` (or `null` if there are no more occurrences).

The solution:

```
cursor = cursor.getLink( );
cursor = IntNode.listSearch(cursor, target);
```

The `grab` Method. The bag has a new `grab` method, specified here:

- **grab**
  `public int grab( )`
  Accessor method to retrieve a random element from this bag.
  Precondition:
  This bag is not empty.
  Returns:
  a randomly selected element from this bag
  Throws: `IllegalStateException`
  Indicates that the bag is empty.

The implementation will start by generating a random `int` value between 1 and the size of the bag. The random value can then be used to select a node from the bag, and we'll return the data from the selected node. So, the body of the method will look something like this:

```
i = some random int value between 1 and the size of the bag;
cursor = listPosition(head, i);
return cursor.getData( );
```

Of course the trick is to generate "some random `int` value between 1 and the size of the bag." The Java class libraries can help. Within `java.lang.math` is a method that (sort of) generates random numbers, with this specification:

- **Math.random**
  `public static double random( )`
  Generates a pseudorandom number in the range 0.0 to 1.0.
  Returns:
  a pseudorandom number in the range 0.0 and 1.0 (this return value may be zero, but it's always less than 1.0)

The values returned by `Math.random` are not truly random. They are generated by a simple rule. But the numbers *appear* random and so the method is referred to as a **pseudorandom number generator**. For most applications, a pseudorandom number generator is a close enough approximation to a true random number generator. In fact, a pseudorandom number generator has one advantage over a true random number generator: The sequence of numbers it produces is repeatable. If run twice with the same initial conditions, a pseudorandom number generator will produce exactly the same sequence of numbers. This is handy when you are debugging programs that use these sequences. When an error is discovered, the corrected program can be tested with the *same* sequence of pseudorandom numbers that produced the original error.

Math.random

But at this point we don't need a complete memoir on pseudorandom numbers. All we need is a way to use the `Math.random` to generate a number between

1 and the size of the bag. The following assignment statement does the trick:

```
// Set i to a random number from 1 to the size of the bag:
i = (int) (Math.random( ) * manyNodes) + 1;
```

Let's look at how the expression works. The method `Math.random` gives us a number that's in the range 0.0 to 1.0. The value is actually in the "half-open range" of [0 .. 1), which means that the number could be from zero up to, but not including, 1. Therefore, the expression `Math.random( ) * manyNodes` is in the range zero to `manyNodes`—or to be more precise, from zero up to, but not including, `manyNodes`.

The operation `(int)` is an operation that truncates a double number, keeping only the integer part. Therefore, `(int) (Math.random( ) * manyNodes)` is an `int` value that could be from zero to `manyNodes-1`. The expression cannot actually be `manyNodes`. Since we want a number from 1 to `manyNodes`, we add one, resulting in `i = (int) (Math.random( ) * manyNodes) + 1`. This assignment statement is used in the complete `grab` implementation shown in Figure 4.16.

The Second Bag—Putting the Pieces Together

The remaining methods are straightforward. For example, the `size` method just returns `manyNodes`. All these methods are given in the complete implementation of Figure 4.17. Take particular notice of how the bag's `addAll` method is implemented. The implementation makes a copy of the linked list for the bag that's being added. This copy is then attached at the front of the linked list for the bag that's being added to. The bag's `union` method is implemented by using the `addAll` method.

FIGURE 4.16 Implementation of a Method to Grab a Random Element

Implementation

```
public int grab( )
{
   int i; // A random value between 1 and the size of the bag
   IntNode cursor;

   if (manyNodes == 0)
      throw new IllegalStateException("Bag size is zero.");

   i =  (int) (Math.random( ) * manyNodes) + 1;
   cursor = IntNode.listPosition(head, i);
   return cursor.getData( );
}
```

FIGURE 4.17 Implementation of Our Second Bag Class

Implementation

```
// FILE: IntLinkedBag.java from the package edu.colorado.collections
// Documentation is available in Figure 4.12 on page 211 or from the IntLinkedBag link in
//   http://www.cs.colorado.edu/~main/docs/

package edu.colorado.collections;
import edu.colorado.nodes.IntNode;

public class IntLinkedBag implements Cloneable
{
   // INVARIANT for the Bag ADT:
   //   1. The elements in the Bag are stored on a linked list.
   //   2. The head reference of the list is in the instance variable head.
   //   3. The total number of elements in the list is in the instance variable manyNodes.
   private IntNode head;
   private int manyNodes;

   public IntLinkedBag( )
   {
      head = null;
      manyNodes = 0;
   }

   public void add(int element)
   {
      head = new IntNode(element, head);
      manyNodes++;
   }

   public void addAll(IntLinkedBag addend)
   {
      IntNode[ ] copyInfo;

      if (addend == null)
         throw new IllegalArgumentException("addend is null.");
      if (addend.manyNodes > 0)
      {
         copyInfo = IntNode.listCopyWithTail(addend.head);
         copyInfo[1].setLink(head); // Link the tail of the copy to my own head...
         head = copyInfo[0];        // and set my own head to the head of the copy.
         manyNodes += addend.manyNodes;
      }
   }
```

(continued)

(FIGURE 4.17 continued)

```
public Object clone( )
```
See the implementation in Figure 4.13 on page 217.

```
public int countOccurrences(int target)
```
See the implementation in Figure 4.15 on page 222.

```
public int grab( )
```
See the implementation in Figure 4.16 on page 224.

```
public boolean remove(int target)
```
See the implementation in Figure 4.14 on page 219.

```
public int size( )
{
   return manyNodes;
}


public static IntLinkedBag union(IntLinkedBag b1, IntLinkedBag b2)
{
   if (b1 == null)
      throw new IllegalArgumentException("b1 is null.");
   if (b2 == null)
      throw new IllegalArgumentException("b2 is null.");

   IntLinkedBag answer = new IntLinkedBag( );

   answer.addAll(b1);
   answer.addAll(b2);
   return answer;
}
}
```

Self-Test Exercises

15. Suppose you want to use a bag where the elements are double numbers instead of integers. How would you do this?
16. Write a few lines of code to declare a bag of integers and place the integers `42` and `8` in the bag. Then grab a random integer from the bag, printing it. Finally print the size of the bag.
17. In general, which is preferable: an implementation that uses the node methods, or an implementation that manipulates a linked list directly?
18. Suppose that `p` is a reference to a node in a linked list of integers and that `p.getData( )` has a copy of an integer called `d`. Write two lines of code that will move `p` to the next node that contains a copy of `d` (or set `p` to `null` if there is no such node). How can you combine your two statements into just one?
19. Describe the steps taken by `countOccurrences` if the target is not in the bag.
20. Describe one of the boundary values for testing `remove`.
21. Write an expression that will give a random integer between –10 and 10.
22. Do big-*O* time analyses of the bag's methods.

4.5 PROGRAMMING PROJECT: THE SEQUENCE ADT WITH A LINKED LIST

In Section 3.3 on page 133 we gave a specification for a sequence ADT that was implemented using an array. Now you can reimplement this ADT using a *linked list* as the data structure rather than an array. Start by rereading the ADT's specification on page 138, then return here for some implementation suggestions.

The Revised Sequence ADT—Design Suggestions

Using a linked list to implement the sequence ADT seems natural. We'll keep the elements stored on a linked list, in their sequence order. The "current" element on the list can be maintained by an instance variable that refers to the node that contains the current element. When the `start` method is activated, we set this cursor to refer to the first node of the linked list. When `advance` is activated, we move the cursor to the next node on the linked list.

With this in mind, we propose five private instance variables for the new sequence class:

- The first variable, `manyNodes`, keeps track of the number of nodes in the list.
- `head` and `tail`—these are references to the head and tail nodes of the linked list. If the list has no elements, then these references are both `null`. The reason for the tail reference is the `addAfter` method. Normally this method adds a new element immediately after the current element. But if there is no current element, then `addAfter` places its new element at the tail of the list, so it makes sense to maintain a connection with the list's tail.
- `cursor`—refers to the node with the current element (or `null` if there is no current element).
- `precursor`—refers to the node before current element (or `null` if there is no current element, or the current element is the first node). Can you figure out why we propose a *pre*cursor? The answer is the `addBefore` method, which normally adds a new element immediately *before* the current element. But there is no node method to add a new node before a specified node. We can only add new nodes after a specified node. Therefore, the `addBefore` method will work by adding the new element *after* the precursor node—which is also just *before* the cursor node.

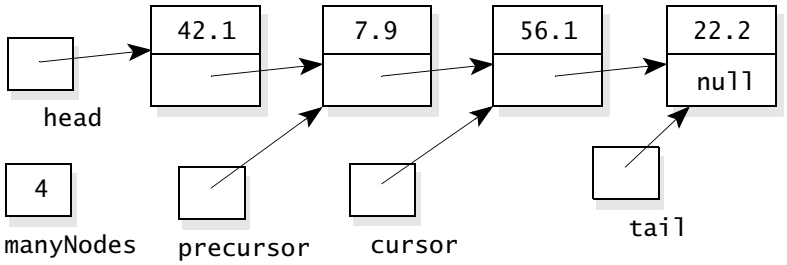
The sequence class that you implement could have integer elements, or double number elements, or several other possibilities. The choice of element type will determine which kind of node you use for the linked list. If you choose integer elements, then you will use `edu.colorado.nodes.IntNode`. All the node classes, including `IntNode`, are available for you to view at `http://www.cs.colorado.edu/~main/edu/colorado/nodes`.

For this programming project, you should use double numbers for the elements and follow these guidelines:

- The name of the class is `DoubleLinkedSeq`;
- You'll use `DoubleNode` for your node class;
- Put your class in a package `edu.colorado.collections`;
- Follow the specification from Figure 4.18 on page 230. This specification is also available at the `DoubleLinkedSeq` link in
  `http//www.cs.colorado.edu/~main/docs/`

Notice that the specification states a limitation that beyond `Int.MAX_VALUE` elements the `size` method does not work (though all other methods should be okay).

Here's an example of the five instance variables for a sequence with four elements and the current element at the third location:



Notice that `cursor` and `precursor` are *references* to two nodes—one right after the other.

what is the invariant of the new list ADT?

Start your implementation by writing the invariant for the new sequence ADT. You might even write the invariant in large letters on a sheet of paper and pin it up in front of you as you work. Each of the methods counts on that invariant being true when the method begins. And each method is responsible for ensuring that the invariant is true when the method finishes.

As you implement each modification method, you might use the following matrix to increase your confidence that the method works correctly.

	`manyNodes`	`head`	`tail`	`cursor`	`precursor`
An empty list					
A nonempty list with no current element					
Current element at the head					
Current element at the tail					
Current element not at head or tail					

Here's how to use the matrix: Suppose you have just implemented one of the modification methods such as `addAfter`. Go through the matrix one row at a time, executing your method with pencil and paper. For example, with the first row of the matrix you would try `addAfter` to see its behavior for an empty list. As you execute each method by hand, keep track of the five instance variables, and put five check marks in the row to indicate that the final values correctly satisfy the invariant.

The Revised Sequence ADT—Clone Method

The sequence class has a `clone` method to make a new copy of a sequence. The sequence that you are copying activates the `clone` method, and we'll call it the "original sequence." As with all `clone` methods, you should start with the pattern from page 78. After activating `super.clone`, the extra work must make a separate copy of the linked list for the clone, and correctly set the clone's head, tail, cursor, and precursor. We suggest that you handle the work with the following three cases:

- If the original sequence has no current element, then simply copy the original's linked list with `listCopy`. Then set both `precursor` and `cursor` to null.
- If the current element of the original sequence is its first element, then copy the original's linked list with `listCopy`. Then set `precursor` to null, and set `cursor` to refer to the head node of the newly created linked list.
- If the current element of the original sequence is after its first element, then copy the original's linked list in two pieces using `listPart`: The first piece goes from the head node to the precursor; the second piece goes from the cursor to the tail. Put these two pieces together by making the link part of the precursor node refer to the cursor node. The reason for copying in two separate pieces is to easily set the precursor and cursor of the newly created list.

FIGURE 4.18 Specification for the Second Version of the `DoubleLinkedSeq` Class

Class DoubleLinkedSeq

❖ **public class DoubleLinkedSeq from the package edu.colorado.collections**

A `DoubleLinkedSeq` is a sequence of double numbers. The sequence can have a special "current element," which is specified and accessed through four methods that are not available in the bag class (`start`, `getCurrent`, `advance`, and `isCurrent`).

Limitations:

Beyond `Int.MAX_VALUE` elements, the `size` method does not work.

Specification

◆ **Constructor for the DoubleLinkedSeq**

```
public DoubleLinkedSeq( )
```

Initialize an empty sequence.

Postcondition:

This sequence is empty.

(continued)

(FIGURE 4.18 continued)

- **addAfter and addBefore**

```
public void addAfter(double element)
public void addBefore(double element)
```

Adds a new element to this sequence, either before or after the current element.

Parameters:
`element` – the new element that is being added

Postcondition:
A new copy of the element has been added to this sequence. If there was a current element, `addAfter` places the new element after the current element and `addBefore` places the new element before the current element. If there was no current element, `addAfter` places the new element at the end of the sequence and `addBefore` places the new element at the front of the sequence. The new element always becomes the new current element of the sequence.

Throws: `OutOfMemoryError`
Indicates insufficient memory for a new node.

- **addAll**

```
public void addAll(DoubleLinkedSeq addend)
```

Place the contents of another sequence at the end of this sequence.

Parameters:
`addend` – a sequence whose contents will be placed at the end of this sequence

Precondition:
The parameter, `addend`, is not null.

Postcondition:
The elements from `addend` have been placed at the end of this sequence. The current element of this sequence remains where it was, and the `addend` is also unchanged.

Throws: `NullPointerException`
Indicates that `addend` is null.

Throws: `OutOfMemoryError`
Indicates insufficient memory to increase the size of the sequence.

- **advance**

```
public void advance( )
```

Move forward, so that the current element is now the next element in the sequence.

Precondition:
`isCurrent( )` returns `true`.

Postcondition:
If the current element was already the end element of the sequence (with nothing after it), then there is no longer any current element. Otherwise, the new element is the element immediately after the original current element.

Throws: `IllegalStateException`
Indicates that there is no current element, so `advance` may not be called.

(continued)

(FIGURE 4.18 continued)

- **clone**

  `public Object clone( )`

  Generate a copy of this sequence.

  Returns:
  The return value is a copy of this sequence. Subsequent changes to the copy will not affect the original, nor vice versa. The return value must be typecast to a `DoubleLinkedSeq` before it is used.

  Throws: `OutOfMemoryError`
  Indicates insufficient memory for creating the clone.

- **concatenation**

  `public static DoubleLinkedSeq concatenation (DoubleLinkedSeq s1, DoubleLinkedSeq s2)`

  Create a new sequence that contains all the elements from one sequence followed by another.

  Parameters:
  `s1` – the first of two sequences
  `s2` – the second of two sequences

  Precondition:
  Neither `s1` nor `s2` is null.

  Returns:
  a new sequence that has the elements of `s1` followed by `s2` (with no current element)

  Throws: `IllegalArgumentException`
  Indicates that one of the arguments is null.

  Throws: `OutOfMemoryError`
  Indicates insufficient memory for the new sequence.

- **getCurrent**

  `public double getCurrent( )`

  Accessor method to determine the current element of the sequence.

  Precondition:
  `isCurrent( )` returns `true`.

  Returns:
  the current element of the sequence

  Throws: `IllegalStateException`
  Indicates that there is no current element.

- **isCurrent**

  `public boolean isCurrent( )`

  Accessor method to determine whether this sequence has a specified current element that can be retrieved with the `getCurrent` method.

  Returns:
  `true` (there is a current element) or `false` (there is no current element at the moment)

(continued)

(FIGURE 4.18 continued)

- **removeCurrent**

  `public void removeCurrent( )`
  Remove the current element from this sequence.

  Precondition:
  `isCurrent( )` returns `true`.

  Postcondition:
  The current element has been removed from the sequence, and the following element (if there is one) is now the new current element. If there was no following element, then there is now no current element.

  Throws: `IllegalStateException`
  Indicates that there is no current element, so `removeCurrent` may not be called.

- **size**

  `public int size( )`
  Accessor method to determine the number of elements in this sequence.

  Returns:
  the number of elements in this sequence

- **start**

  `public void start( )`
  Set the current element at the front of the sequence.

  Postcondition:
  The front element of this sequence is now the current element (but if the sequence has no elements at all, then there is no current element).

Self-Test Exercises

23. Suppose a sequence contains your three favorite numbers, and the current element is the first element. Draw the instance variables of this sequence using our implementation.

24. Write a new method to remove a specified element from a sequence of double numbers. The method has one parameter (the element to remove). After the removal, the current element should be the element after the removed element (if there is one).

25. Which of the sequence methods use the `new` operator to allocate at least one new node?

26. Which of the sequence methods use `DoubleNode.listPart`?

4.6 ARRAYS VS. LINKED LISTS VS. DOUBLY LINKED LISTS

Many ADTs can be implemented with either arrays or linked lists. Certainly the bag and the sequence ADT could each be implemented with either approach.

Which approach is better? There is no absolute answer. But there are certain operations that are better performed by arrays and others where linked lists are preferable. This section provides some guidelines.

Arrays Are Better at Random Access. The term **random access** refers to examining or changing an arbitrary element that is specified by its position in a list. For example: *What is the 42nd element in the list*? Or another example: *Change the element at position 1066 to a 7*. These are constant time operations for an array. But, in a linked list, a search for the i^{th} element must begin at the head and will take $O(i)$ time. Sometimes there are ways to speed up the process, but even improvements remain linear time in the worst case.

If an ADT makes significant use of random access operations, then an array is better than a linked list.

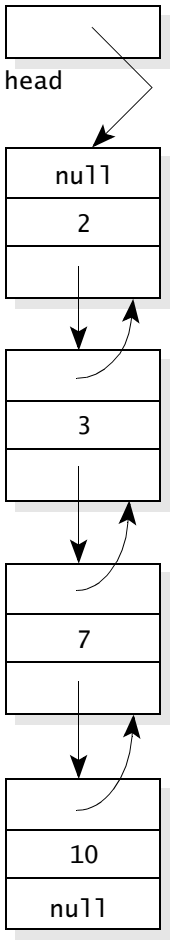


FIGURE 4.19
Doubly Linked List

Linked Lists Are Better at Additions/Removals at a Cursor. Our sequence ADT maintains a *cursor* that refers to a "current element." Typically, a cursor moves through a list one element at a time without jumping around to random locations. If all operations occur at the cursor, then a linked list is preferable to an array. In particular, additions and removals at a cursor generally are linear time for an array (since elements that are after the cursor must *all* be shifted up or back to a new index in the array). But these operations are constant time operations for a linked list. Also remember that effective additions and removals in a linked list generally require maintaining both a cursor and a *precursor* (which refers to the node before the cursor).

If an ADT's operations take place at a cursor, then a linked list is better than an array.

Doubly Linked Lists Are Better for a Two-way Cursor. Sometimes list operations require a cursor that can move forward and backward through a list—a kind of **two-way cursor**. This situation calls for a **doubly linked list**, which is like an ordinary linked list, except that each node contains two references: one linking to the next node and one linking to the previous node. An example of a doubly linked list of integers is shown in Figure 4.19 (in the margin). A possible start of a declaration for a doubly linked list of elements is the following:

```
public class DIntNode
{
   private int data;
   private DIntNode backlink;
   private DIntNode forelink;
   ...
```

The `backlink` refers to the previous node, and the `forelink` refers to the next node in the list.

If an ADT's operations take place at a two-way cursor, then a doubly linked list is the best implementation.

Resizing Can Be Inefficient for an Array. A collection class that uses an array generally provides a method to allow a programmer to adjust the capacity as needed. But changing the capacity of an array can be inefficient. The new memory must be allocated and initialized, and the elements are then copied from the old memory to the new memory. If a program can predict the necessary capacity ahead of time, then capacity is not a big problem, since the object can be given sufficient capacity from the outset. But sometimes the eventual capacity is unknown and a program must continually adjust the capacity. In this situation, a linked list has advantages. When a linked list grows, it grows one node at a time, and there is no need to copy elements from old memory to new memory.

If an ADT is frequently adjusting its capacity, then a linked list may be better than an array.

Making the Decision

Your decision on what kind of implementation to use is based on your knowledge of which operations occur in the ADT, which operations you expect to be performed most often, and whether you expect your arrays to require frequent capacity changes. Figure 4.20 summarizes these considerations.

Self-Test Exercises

27. What underlying data structure is quickest for random access?
28. What underlying data structure is quickest for additions/removals at a cursor?
29. What underlying data structure is best if a cursor must move both forward and backward?
30. What is the typical worst-case time analysis for changing the capacity of a collection class that is implemented with an array?

FIGURE 4.20 Guidelines for Choosing between an Array and a Linked List

Frequent random access operations	Use an array
Operations occur at a cursor	Use a linked list
Operations occur at a two-way cursor	Use a doubly linked list
Frequent capacity changes	A linked list avoids resizing inefficiency

CHAPTER SUMMARY

- A *linked list* consists of nodes; each *node* contains some data and a link to the next node in the list. The link part of the final node contains the null reference.
- Typically, a linked list is accessed through a *head reference* that refers to the *head node* (i.e., the first node). Sometimes a linked list is accessed elsewhere, such as through the *tail reference* that refers to the last node.
- You should be familiar with the methods of our node class, which provides fundamental operations to manipulate linked lists. These operations follow basic patterns that every programmer uses.
- Our linked lists can be used to implement ADTs. Such an ADT will have one or more private instance variables that are references to nodes in a linked list. The methods of the ADT will use the node methods to manipulate the linked list.
- You have seen two ADTs implemented with linked lists: a bag and a sequence. You will see more in the chapters that follow.
- ADTs often can be implemented in many different ways, such as by using an array or using a linked list. In general, arrays are better at *random access*; linked lists are better at additions/removals at a *cursor.*
- A *doubly linked list* has nodes with two references: one to the next node and one to the previous node. Doubly linked lists are a good choice for supporting a cursor that moves forward and backward.

? Solutions to Self-Test Exercises

1.
```
public class DoubleNode
{
   double data;
   DoubleNode link;
   ...
```

2. The head node and the tail node.

3. The null reference is used for the link part of the final node of a linked list; it is also used for the head and tail references of a list that doesn't yet have any nodes.

4. A `NullPointerException` is thrown.

5. Using techniques from Section 4.2:
```
if (head == null)
    head = new IntNode(42, null);
else
    head.addNodeAfter(42);
```

6. Using techniques from Section 4.2:
```
if (head != null)
{
   if (head.getLink( ) == null)
      head = null;
   else
      head.removeNodeAfter( );
}
```

7. They cannot be implemented as ordinary methods of the `IntNode` class because they must change the head reference (making it refer to a new node).

8.
```
IntNode head;
IntNode tail;
int i;
head = new IntNode(1, null);
tail = head;
for (i = 2; i <= 100; i++)
{
   tail.addNodeAfter(i);
   tail = tail.getLink( );
}
```

9. There are eight different nodes for the eight primitive data types (boolean, int, long, byte, short, double, float, and char). These are called `BooleanNode`, `IntNode`, and so on. There is one more class simply called `Node`, which will be discussed in Chapter 5. The data type in the `Node` class is Java's `Object` type. So there are nine different nodes in all.

If you implement one of these nine node types, implementing another one takes little work—just change the type of the data and the type of any method parameters that refer to the data.

10. Within the `IntNode` class you may write:
```
locate = locate.link;
```
Elsewhere you must write:
```
locate = locate.getLink( );
```
If `locate` is already referring to the last node before the assignment statement, then the assignment will set `locate` to null.

11. The `new` operator is used in the methods: `addNodeAfter`, `listCopy`, `listCopyWithTail`, and `listPart`.

12. It will be the null reference.

13. The `listCopyWithTail` method does exactly this by returning an array with two `IntNode` components.

14.
```
douglass =
    IntNode.listSearch(head, 42);
adams =
    IntNode.listPosition(head, 42);
```

15. Use `DoubleNode` instead of `IntNode`. There are a few other changes, such as changing some parameters from `int` to `double`.

16. We could write this code:
```
IntLinkedBag exercise =
   new IntLinkedBag( );
exercise.add(42);
exercise.add(8);
System.out.println
  (exercise.grab( ));
System.out.println
  (exercise.size( ));
```

17. Generally we will choose the approach that makes the best use of the node methods. This saves us work and also reduces the chance of new errors from writing new code to do an old job. The preference would change if the other approach offered better efficiency.

18. The two lines of code that we have in mind:
```
p = p.getLink( );
p = listSearch(p, d);
```
These two lines are the same as the single line:
```
p = listSearch(p.getLink( ), d);
```

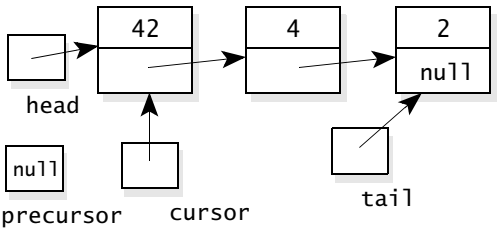
19. When the target is not in the bag, the first assignment statement to `cursor` will set it to null. This means that the body of the loop will not execute at all, and the method returns the answer zero.

20. Test the case where you are removing the last element from the bag.

21. `(int) (Math.random( ) * 21) - 10`

22. All the methods are constant time except for `remove`, `grab`, `countOccurrences`, and `clone` (all of which are linear); the `addAll` method (which is *O(n)*, where *n* is the size of the addend); and the `union` method (which is *O(m+n)*, where *m* and *n* are the sizes of the two bags).

23. `manyNodes` is 3, and these are the other instance variables:



24. First check that the element occurs somewhere in the sequence. If it doesn't, then return with no work. If the element is in the sequence, then set the current element to be equal to this element, and activate the ordinary remove method.

25. The two `add` methods both allocate dynamic memory, as do `addAll`, `clone`, and `concatenation`.

26. The `clone` method should use `listPart`, as described on page 230.

27. Arrays are quickest for random access.

28. Linked lists are quickest for additions/removals at a cursor.

29. A doubly linked list.

30. At least *O(n)*, where *n* is the size of the array prior to changing the size. If the new array is initialized, then there is also *O*(*m*) work, where *m* is the size of the new array.

PROGRAMMING PROJECTS

1 For this project, you will use the bag of integers from Section 4.4. The bag includes the `grab` method from Figure 4.16 on page 224. Use this class in an applet that has three components:

1. A button
2. A small text field
3. A large text area

Each time the button is clicked, the applet should read an integer from the text field, and put this integer in a bag. Then a random integer is grabbed from the bag and a message is printed in the text area—something like "My favorite number is now...."

2 Write a new static method for the node class. The method has one parameter which is a head node for a linked list of integers. The method computes a new linked list, which is the same as the original list, but in which all repetitions are removed. The method's return value is a head reference for the new list.

3 Write a method with three parameters. The first parameter is a head reference for a linked list of integers, and the next two parameters are integers `x` and `y`. The method should write a line to `System.out`, containing all integers in the list that are between the first occurrence of `x` and the first occurrence of `y`.

4 Write a method with one parameter that is a head reference for a linked list of integers. The method creates a new list that has the same elements as the original list, but in the reverse order. The method returns a head reference for the new list.

5 Write a method that has two linked list head references as parameters. Assume that linked lists contain integer data, and on each list, every element is less than the next element on the same list. The method should create a new linked list that contains all the elements on both lists, and the new linked list should also be ordered (so that every element is less than the next element on

the list). The new linked list should not eliminate duplicate elements (i.e., if the same element appears on both input lists, then two copies are placed in the newly constructed linked list). The method should return a head reference for the newly constructed linked list.

6 Write a method that starts with a single linked list of integers and a special value called the splitting value. The elements of the list are in no particular order. The method divides the nodes into two linked lists: one containing all the nodes that contain an element less than the splitting value and one that contains all the other nodes. If the original linked list had any repeated integers (i.e., any two or more nodes with the same element in them), then the new linked list that has this element should have the same number of nodes that repeat this element. It does not matter whether you preserve the original linked list or destroy it in the process of building the two new lists, but your comments should document what happens to the original linked list. The method returns two head references—one for each of the linked lists that were created.

7 Write a method that takes a linked list of integers and rearranges the nodes so that the integers stored are sorted into the order smallest to largest, with the smallest integer in the node at the head of the list. Your method should preserve repetitions of integers. If the original list had any integers occurring more than once, then the changed list will have the same number of each integer. For concreteness you will use lists of integers, but your method should still work if you replace the integer type with any other type for which the less-than operation is defined. Use the following specification:

```
IntNode listSort(IntNode head)
// Postcondition: The return value is a head
// reference of a linked list with exactly the
// same entries as the original list (including
// repetitions if any), but the entries
// in this list are sorted from smallest to
// largest. The original linked list is no longer
// available.
```

Your method will implement the following algorithm (which is often called **selection sort**): The algorithm removes nodes one at a time from the original list and adds the nodes to a second list until all the nodes have been moved to the second list. The second list will then be sorted.

```
// Pseudocode for selection sort
while (the first list still has some nodes)
{
    1. Find the node with the largest element
       of all the nodes in the first list.
    2. Remove this node from the first list.
    3. Add this node at the head of the second
       list.
}
```

Note that your method will move entire nodes, not just elements, to the second list. Thus, the first list will get shorter and shorter until it is an empty list. Your method should not need to use the `new` operator since it is just moving nodes from one list to another (not creating new nodes).

8 Implement a new method for the bag class from Section 4.4. The new method allows you to subtract the contents of one bag from another. For example, suppose that `x` has seven copies of the number 3 and `y` has two copies of the number 3. Then after activating `x.subtract(y)`, the bag `x` will have five remaining copies of the number 3.

9 Implement the `Sequence` class from Section 4.5. You may wish to provide some additional useful methods, such as: (1) a method to add a new element at the front of the sequence; (2) a method to remove the element at the front of the sequence; (3) a method to add a new element at the end of the sequence; (4) a method that makes the last element of the sequence become the current element; (5) a method that returns the i^{th} element of the sequence (starting with the 0^{th} at the front); (6) a method that makes the i^{th} element become the current element.

10 You can represent an integer with any number of digits by storing the integer as a linked list of digits. A more efficient representation will store a larger integer in each node. Design and implement an ADT for unbounded whole numbers in which a number is implemented as a linked list of integers. Each node will hold an integer less than or equal to 999. The number represented is the concatenation of the numbers in the nodes. For example, if there are four nodes with the four integers 23, 7, 999, and 0, then this represents the number 23,007,999,000. Note that the number in a node is always considered to be three digits long. If it is not three digits long, then leading zeros are added to make it three digits long. Include methods for the usual integer operators to work with your new class.

11 Revise one of the collection classes from Chapter 3, so that it uses a linked list. Some choices are: (a) the set (Project 6 on page 168); (b) the sorted sequence (Project 7 on page 169).

12 Revise the Chapter 3 polynomial class from Project 8 on page 169, so that the coefficients are stored in a linked list and there is no maximum degree. Include an operation to allow you to multiply two polynomials in the usual way. For example:

$$(3x^2 + 7) * (2x + 4) = (6x^3 + 12x^2 + 14x + 28)$$