CHAPTER 4

Linked Lists

LEARNING OBJECTIVES

When you complete Chapter 4, you will be able to...

- design and implement methods to manipulate nodes in a linked list, including inserting new nodes, removing nodes, searching for nodes, and processing (such as copying) that involves all the nodes of a list.
- design and implement collection classes that use linked lists to store a collection of elements, generally using a node class to create and manipulate the linked lists.
- analyze problems that can be solved with linked lists and, when appropriate, propose alternatives to simple linked lists, such as doubly linked lists and lists with dummy nodes.

CHAPTER CONTENTS

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

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

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.

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 shown 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.

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."

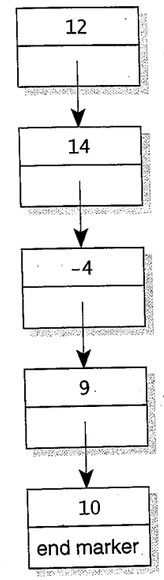


FIGURE 4.1
A Linked List Made of Nodes Connected with Links

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;    // Refers to the next node in the list
   ...
```

We'll provide the methods later (Sections 4.2 and 4.3). For now we want to focus on the two 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 `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**).

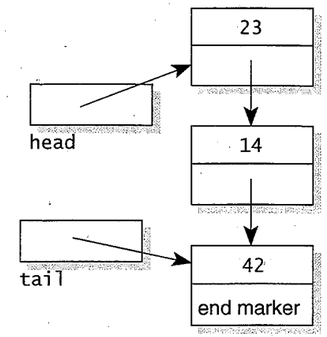
FIGURE 4.2 Node Declarations in a Program with a Linked List

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.

The Null Reference

Figure 4.3 illustrates a linked list with 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 in which the null reference is used.

The Null Reference and Linked Lists

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 54.

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 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`.

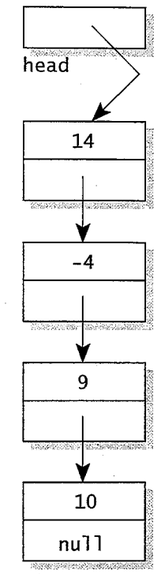


FIGURE 4.3 Linked List with the Null Reference at the Final Link

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 for Section 4.1

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. Write another node declaration, but this time the data in each node should include both a double number and an integer. (Yes, a node can have many instance variables for data, but each instance variable needs to have a distinct name.)
3. Suppose a program builds and manipulates a linked list. What two special nodes would the program typically keep track of?
4. Describe two uses for the null reference in the realm of linked lists.
5. How many nodes are on an empty list? What are the values of the head and tail references for an empty list?
6. 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;    // Refers 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 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:

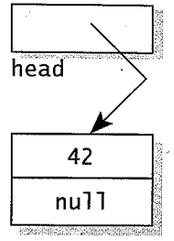
This new node contains the specified data and link to the next node.

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 the linked list shown in the margin.

```
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**

`public int getData( )`

Accessor method to get the data from this node.

Returns:

the data from this node

getData, getLink, setData, setLink

♦ **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 short. For example:

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

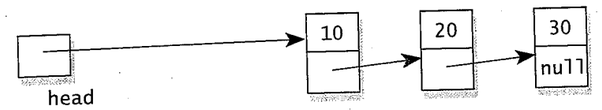
Public Versus Private Instance Variables

In addition to `setLink`, there are the three other short methods that we'll leave for you to implement. You may wonder why you should 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.

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:

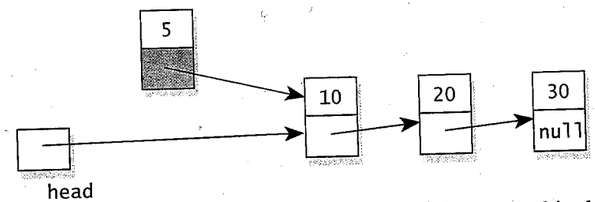


In this example, suppose we want to add a new node to the front of this list, with 5 as the data. Using the node constructor, we can write:

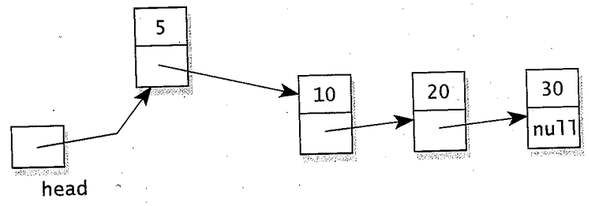
```
head = new IntNode(5, head);
```

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

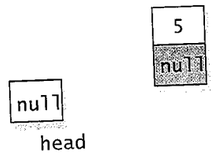
Let's step through the execution of this statement to see how the new node is added at the front of the list. When the constructor is executed, a new node is created with 5 as the data and with the link referring to the same node that `head` refers to. Here's what the picture looks like, with the link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



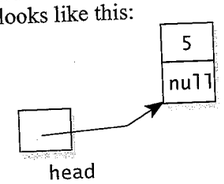
The constructor returns a reference to the newly created node, and in the statement we wrote `head = new IntNode(5, head)`. You can read this statement as saying "make `head` refer to the newly created node." Therefore, we end up with this situation:



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

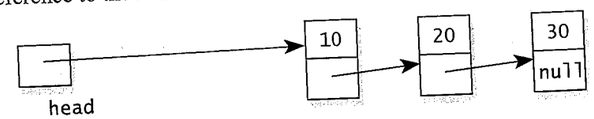
Suppose head is the head reference of a linked list. Then this statement adds a new node at the front of the list with the specified new data:

```
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

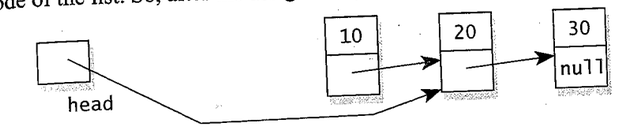
Nodes can be removed from the head of the linked list. To accomplish this, we need a reference to the head node of a list:



To remove the first node, we simply move the head to the second 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:

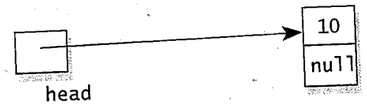


This picture is peculiar. It looks like we've still got a linked list with three nodes containing 10, 20, and 30. But if we start at the head, there are only the two nodes with 20 and 30. The node with 10 can no longer be accessed starting at the head, so it is not really part of the linked list any more. In fact, if a situation arises in which 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.

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.

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

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

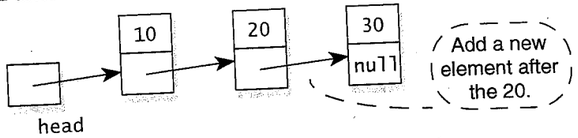
Suppose `head` is the head reference of a non-empty linked list (so that `head` is not null). Then this statement removes a node from the front of the list:

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

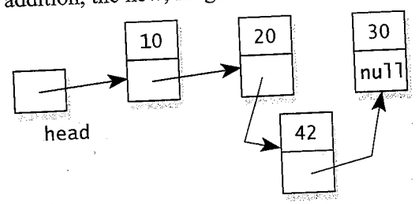
This statement works correctly even when the list has just one node (in which case the head reference becomes null).

Adding a New Node That Is Not at the Head

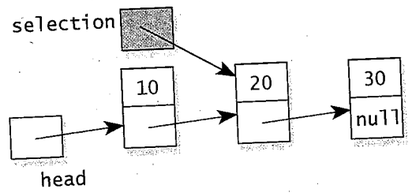
New nodes are not always placed at the head of a linked list. They may be added in the middle or at the tail of a list. For example, suppose you want to add the number 42 after the 20 in this list:



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` this way:



be added with a method of the IntNode class, specified as follows:

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.

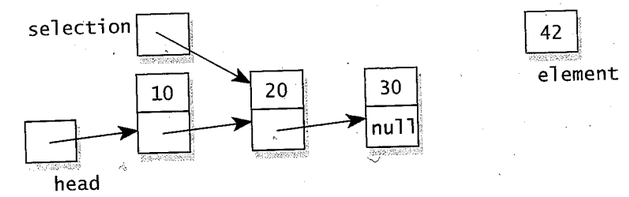
Throws: `OutOfMemoryError`
Indicates that there is insufficient memory for a new `IntNode`.

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

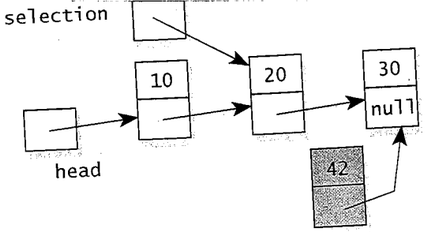
The implementation of `addNodeAfter` requires just one line:

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

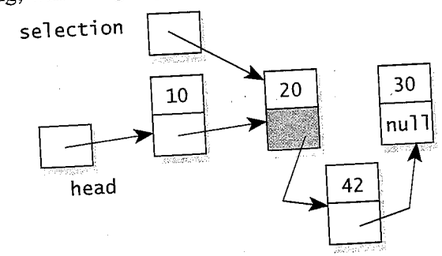
Let's see exactly what happens when we set up `selection` as shown earlier and then execute `selection.addNodeAfter(42)`. The value of `element` is 42, so we have this situation:



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 on 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

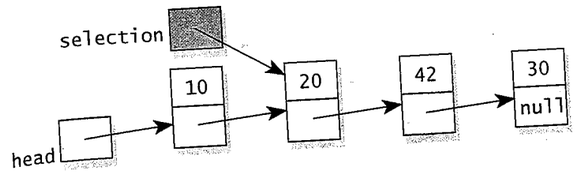
Suppose `selection` is a reference to a node of a linked list. Activating the following method adds a new node after the selected node (using `element` as the new data):

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

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

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

It is also possible to remove a node after the midlist node, we must set up a reference to the node that is just *before* the node we are removing. For example, 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 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:

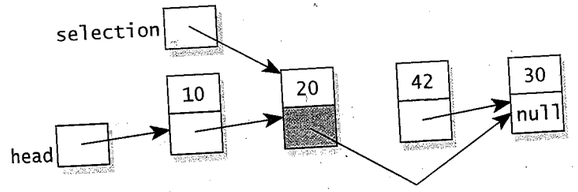
removeNodeAfter

♦ **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.

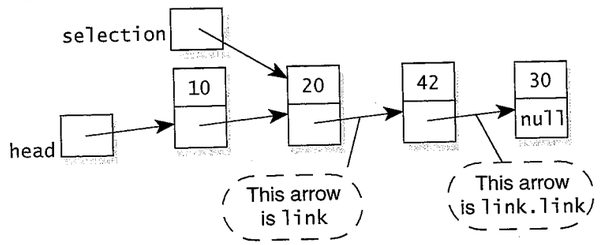
Postcondition:
The node after this node has been removed from the linked list. If there were further nodes after that one, they are still present on the list.

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.

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.

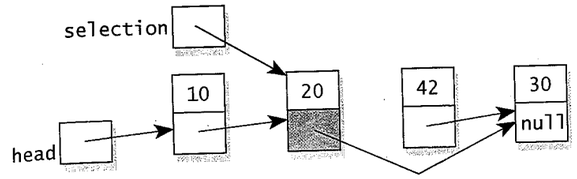
Because the name `link` refers to a node, we can also use the names `link.data` and `link.link`:

`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.

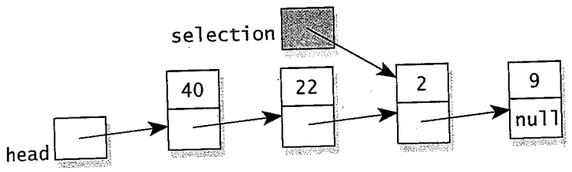
In the implementation of `removeNodeAfter`, we need to make `link` refer to the node that contains 30. So, using the notation just shown, we need to assign `link = link.link`. The complete implementation is at the top of the next page.

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

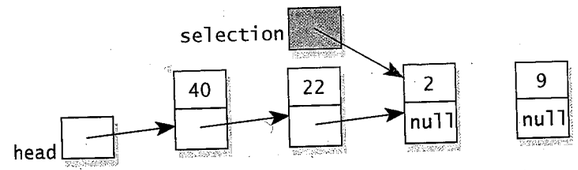
The notation `link.link` does look strange, but just read it from left to right so that it means "go to the node that `link` refers to and use the `link` instance variable." In our example, the final situation after assigning `link = link.link` is just what we want:



The `removeNodeAfter` implementation works fine, even if we want to remove the tail node. Here's an example in which 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.

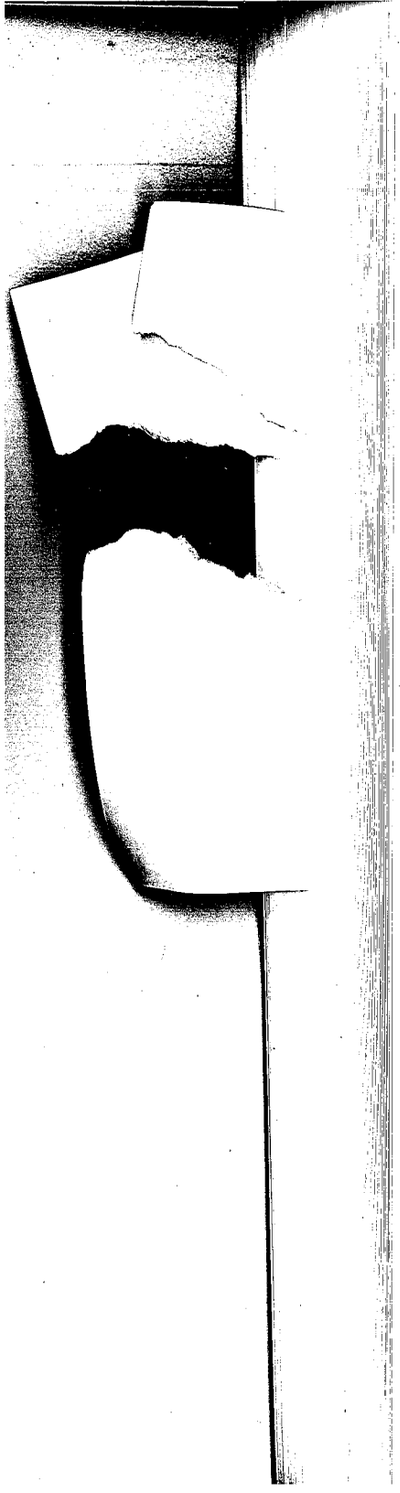
Removing a Node That Is Not at the Head

Suppose `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

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 for Section 4.2

7. Suppose `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.)
8. Suppose `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.)
9. 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?
10. Write some code that could appear in a main program. The code should declare `head` and `tail` references for a linked list of integers and then create a list of nodes with the data being integers 1 through 100 (in that order). After each of the nodes is added, `head` and `tail` should still be valid references to the head and tail nodes of the current list.