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 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:

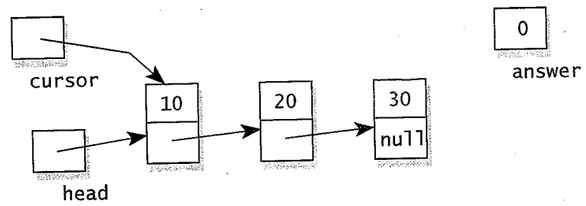
how to traverse all the nodes of a linked list

```
for (cursor = head; cursor != null; cursor = cursor.link)
{
    ...     Inside the body of the loop, you can 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`:

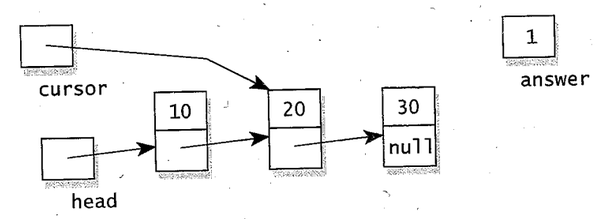
```
for (cursor = head; cursor != null; cursor = cursor.link)
   answer++;
```

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

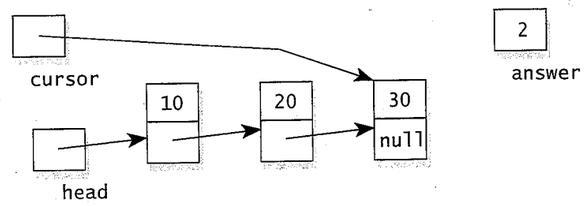


Notice that `cursor` refers to the same node that `head` refers to.

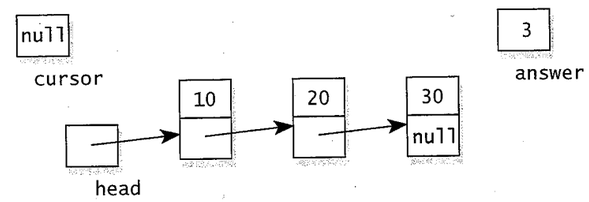
Since `cursor` is not `null`, we enter the body of the loop. Each iteration increments `answer` and then executes `cursor = cursor.link`. The effect of `cursor = cursor.link` is to copy the `link` part of the first node into `cursor` itself so that `cursor` ends up referring to the second node. In general, 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 at the top of the next page.



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`:



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

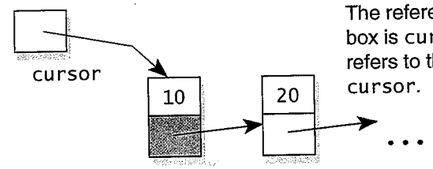
PROGRAMMING TIP

How to Traverse a Linked List

You should learn the important pattern for traversing a linked list, as used in the `listLength` method (see 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:



The reference in the shaded box is `cursor.link`, and it refers to the next node after `cursor`.

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 can 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

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 can 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

♦ **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.

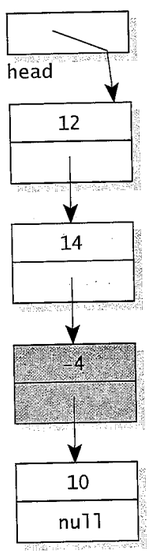


FIGURE 4.5 Example for `listSearch`

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, a reference to the node is immediately returned with the return

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;
}
```

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.

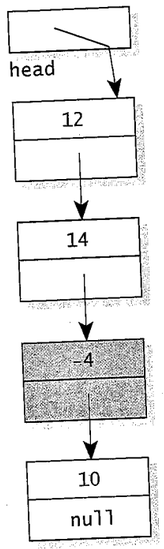


FIGURE 4.7 Example for `listPosition`

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 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 4.8 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 = 1; (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)

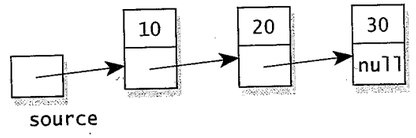
Returns:

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 in which 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 Step 2 through Step 4 on the top of the next page.

2. Create a new node for the head node of the new list 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;
```

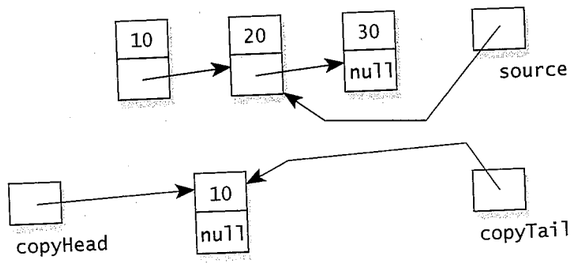
4. After Step 3 completes, return `copyHead` (a reference to the head node of the list we created).

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.

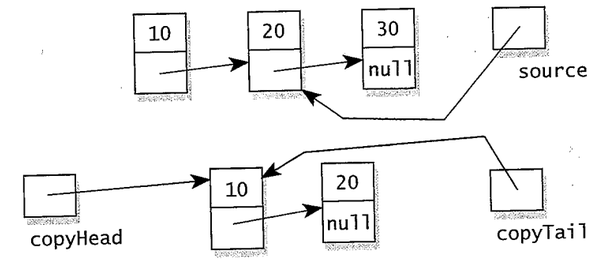
As an example, consider again the three-node list with data 10, 20, and 30. The first two steps of the pseudocode are carried out, and then we enter the body of the while-loop. We execute the first statement of the loop: `source = source.link`. At this point, the variables look like this:



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`:

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

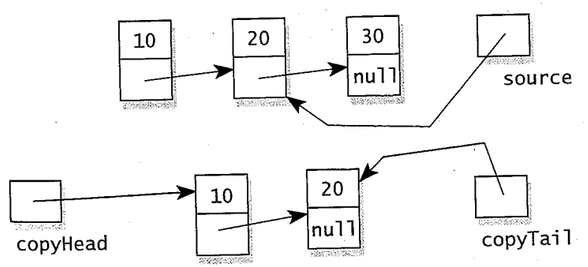
This activation adds a new node to the end of the list we are creating (i.e., *after* the node referred to by `copyTail`), and the data in the new node is the number 20 (i.e., the data from `source.data`). Immediately after adding the new node, the variables look like this:



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

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

This is the usual way in which 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`.

FIGURE 4.9 A Static Method to Copy a Linked List

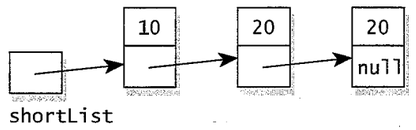
Implementation

```
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;
}
```

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

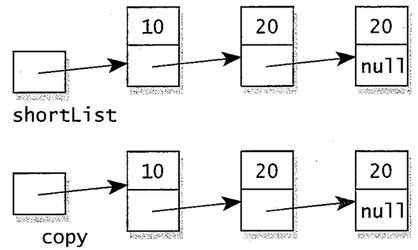

At this point, `shortList` is the head of a small list shown here:



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

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

Now 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:

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`

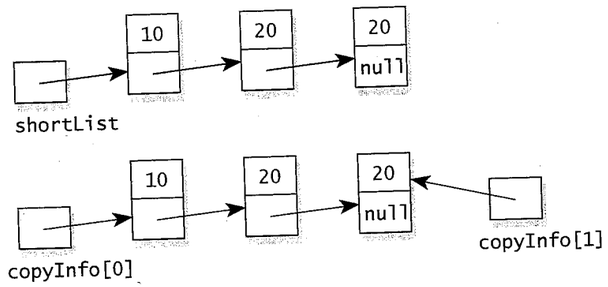
Indicates that there is insufficient memory for the new list.

The `listCopyWithTail` method 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 and 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:



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

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 in which 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 201. 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. This is because we intend to use the node classes ourselves to build various collection classes. The different collection classes 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.

(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.

(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 201-204 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 196.

   public static IntNode[ ] listCopyWithTail(IntNode source)
   || See the implementation in Figure 4.10 on page 199.

   public static int listLength(IntNode head)
   || See the implementation in Figure 4.4 on page 188.
```

(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 192.

public static IntNode listSearch(IntNode head, int target)
|| See the implementation in Figure 4.6 on page 191.
```

(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 for Section 4.3

11. 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?
12. Suppose `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 referring to the last node in the list?
13. 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 201 (to see which methods can throw an `OutOfMemoryError`).
14. Suppose `head` is a head reference for a linked list with just one node. What will `head` be after `head = head.getLink( )`?
15. 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.
16. Suppose `head` is a head reference for a linked list. Also suppose `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.

17. Suppose a program sets up a linked list with an `IntNode` variable called `head` to refer to the first node of the list (or `head` is `null` if the list is empty). Write a few lines of Java code that will print all the numbers.
18. Implement a new static method for the `IntNode` class with one parameter that is a head reference for a linked list. The return value of the method is the number of times that the number 42 appears on the list.
19. Implement a new static method for the `IntNode` class with one parameter that is a head reference for a linked list. The return value of the method is a boolean value that is true if the list contains at least one copy of 42; otherwise, false.
20. Implement a new static method for the `IntNode` class with one parameter that is a head reference for a linked list. The return value of the method is the sum of all the numbers on the list.
21. Implement a new static method for the `IntNode` class with one parameter that is a head reference for a linked list. The function adds a new node to the tail of the list with the data equal to 42.
22. Implement a new static method for the `IntNode` class with one parameter that is a head reference for a linked list. The method removes the first node after the head that contains the number 42 (if there is such a node). Do not call any other methods to do any of the work.
23. Suppose `p`, `q`, and `r` are all references to nodes in a linked list with 15 nodes. The variable `p` refers to the first node, `q` refers to the 8th node, and `r` refers to the last node. Write a few lines of code that will make a new copy of the list. Your code should set *three* new variables called `x`, `y`, and `z` so that `x` refers to the first node of the copy, `y` refers to the 8th node of the copy, and `z` refers to the last node of the copy. Your code may not contain any loops, but it can use the other `IntNode` methods.