15 Linked Data Structures

see note if page cropped 5/18 836

15 Linked Data Structures

If somebody there chanced to be
Who loved me in a manner true
My heart would point him out to me
And I would point him out to you.

GILBERT AND SULLIVAN, *Ruddigore*

Introduction

node and link

linked list

head node

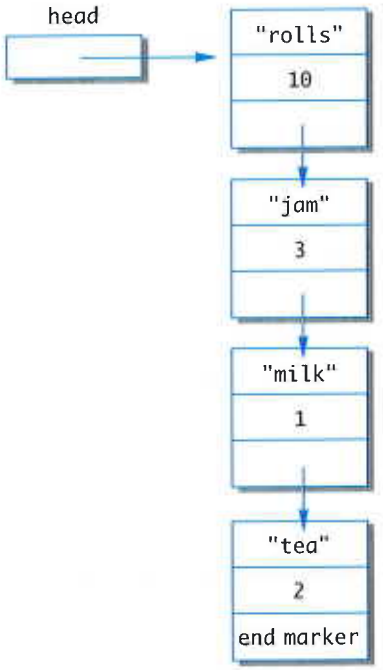
A linked data structure consists of capsules of data, known as **nodes**, that are connected via things called **links**. These links can be viewed as arrows and thought of as one-way passages from one node to another. The simplest kind of linked data structure consists of a single chain of nodes, each connected to the next by a link; this is known as a **linked list**. A sample linked list can be depicted as shown in Display 15.1. In Display 15.1 the nodes are represented by boxes that can each hold two kinds of data, a string and an integer, as in a shopping list. The links are depicted as arrows, which reflects the fact that your code must traverse the linked list in one direction without backing up. So there is a first node, a second node, and so on up to the last node. The first node is called the **head node**.

That information is all very vague but provides the general picture of what is going on in a linked list. It becomes concrete when you realize a linked list in some programming language. In Java, the nodes are realized as objects of a node class. The data in a node is stored via instance variables. The links are realized as references. Recall that a reference is simply a memory address. A reference is what is stored in a variable of a class type. So the link is realized as an instance variable of the type of the node class itself. In Java, a node in a linked list is connected to the next node by having an instance variable of the node type contain a reference (that is, a memory address) of where in memory the next node is stored.

Java comes with a `LinkedList` library class as part of the `java.util` package. It makes sense to use this library class, since it is well designed, well tested, and will save you a lot of work. However, using the library class will not teach you how to implement linked data structures in Java. To do that, you need to see an implementation of a simple linked data structure, such as a linked list. So to let you see how this sort of thing is done in Java, we will construct our own simplified example of a linked list.

After discussing linked lists we then go on to discuss more elaborate linked data structures, including sets, hash tables, and trees.

Display 15.1 Nodes and Links in a Linked List



Prerequisites

If you prefer, you may skip this chapter and go directly to Chapter 16 on collection classes or to Chapter 17 to begin your study of windowing interfaces using the Swing library. You have a good deal of flexibility in how you order the later chapters of this book.

This chapter requires material from Chapters 1 through 5, Chapter 14, and simple uses of inner classes (Section 13.2 of Chapter 13). Section 15.7 on trees additionally requires Chapter 11 on recursion.

Sections 15.2 through 15.7 do not depend on each other in any essential way. In particular, you may omit Section 15.2 on cloning and still read the following sections. Sections 15.2 through 15.7 do not depend in any essential way on the material on generic linked lists in subsections of Section 15.1.

15.1 Java Linked Lists

A chain is only as strong as its weakest link.

PROVERB

A linked list is a linked data structure consisting of a single chain of nodes, each connected to the next by a link. This is the simplest kind of linked data structure, but it is nevertheless widely used. In this section, we give examples of linked lists and develop techniques for defining and working with linked lists in Java.

EXAMPLE: A Simple Linked List Class

Display 15.1 is a diagram of a linked list. In the display the nodes are the boxes. In your Java code, a node is an object of some node class, such as the class `Node1` given in Display 15.2. Each node has a place (or places) for some data and a place to hold a link to another node. The links are shown as arrows that point to the node they "link" to. In Java, the links will be implemented as references to a node stored in an instance variable of the `node` type.

The `Node1` class is defined by specifying, among other things, an instance variable of type `Node1` that is named `link`. This allows each node to store a reference to another node of the same type. There is a kind of circularity in such definitions, but this circularity is allowed in Java. (One way to see that this definition is not logically inconsistent is to note that we can draw pictures, or even build physical models, of our linked nodes.)

The first node, or start node, in a linked list is called the head node. If you start at the head node, you can traverse the entire linked list, visiting each node exactly once. As you will see in Java code shortly, your code must intuitively "follow the link arrows." In Display 15.1 the box labeled `head` is not itself the head node; it is not even a node. The box labeled `head` is a variable of type `Node1` that contains a reference to the first node in the linked list—that is, a reference to the head node. The function of the variable `head` is that it allows your code to find that first or head node. The variable `head` is declared in the obvious way:

```
Node1 head;
```

In Java, a linked list is an object that in some sense contains all the nodes of the linked list. Display 15.3 contains a definition of a linked list class for a linked list like the one in Display 15.1. Notice that a linked list object does not directly contain all the nodes in the linked list. It only contains the instance variable `head` that contains a reference to the first or head node. However, every node can be reached from this first or head node. The `link` instance variable of the first and every `Node1` of the linked list contains a reference to the next `Node1` in the linked list. Thus, the arrows shown in the diagram in Display 15.1 are realized as references in Java. Each node object of a linked list contains (in its `link` instance variable) a reference to another object of the class `Node1`, and this other object contains a reference to another object of the class `Node1`, and so on until the end of the linked list. Thus, a linked list object, indirectly at least, contains all the nodes in the linked list.

Display 15.2 A Node Class

```
public class Node1
{
    private String item;
    private int count;
    private Node1 link;

    public Node1( )
    {
        link = null;
        item = null;
        count = 0;
    }
    public Node1(String newItem, int newCount, Node1 linkValue)
    {
        setData(newItem, newCount);
        link = linkValue;
    }

    public void setData(String newItem, int newCount)
    {
        item = newItem;
        count = newCount;
    }
    public void setLink(Node1 newLink)
    {
        link = newLink;
    }

    public String getItem( )
    {
        return item;
    }

    public int getCount( )
    {
        return count;
    }

    public Node1 getLink( )
    {
        return link;
    }
}
```

A node containe a reference to another node. That reference is the link to the next node.

We will define a number of node classes so we numbered the names as in **Node1**.

We will give a better definition of a node class later in this chapter.

Display 15.3 A Linked List Class (part 1 of 2)

```
1  public class LinkedList1
2  {
3      private Node1 head;
4
5      public LinkedList1( )
6      {
7          head = null;
8      }

9      /**
10      Adds a node at the start of the list with the specified data.
11      The added node will be the first node in the list.
12     */
13     public void addToStart(String itemName, int itemCount)
14     {
15         head = new Node1(itemName, itemCount, head);
16     }

17     /**
18      Removes the head node and returns true if the list contained at least
19      one node. Returns false if the list was empty.
20     */
21     public boolean deleteHeadNode( )
22     {
23         if (head != null)
24         {
25             head = head.getLink( );
26             return true;
27         }
28         else
29             return false;
30     }

31     /**
32      Returns the number of nodes in the list.
33     */
34     public int size( )
35     {
36         int count = 0;
37         Node1 position = head;
38
```

We will define a letter linked list class later in this chapter.

Display 15.3 A Linked List Class (part 2 of 2)

```
39          while (position != null)
40          {
41              count++;
42              position = position.getLink( );
43          }
44          return count;
45      }

46      public boolean contains(String item)
47      {
48          return (find(item) != null);
49      }

50      /**
51       Finds the first node containing the target item, and returns a
52       reference to that node. If target is not in the list, null is returned.
53      */
54      private Node1 find(String target)
55      {
56          Node1 position = head;
57          String itemAtPosition;
58          while (position != null)
59          {
60              itemAtPosition = position.getItem( );
61              if (itemAtPosition.equals(target))
62                  return position;
63              position = position.getLink( );
64          }
65          return null; //target was not found
66      }
67      public void outputList( )
68      {
69          Node1 position = head;
70          while (position != null)
71          {
72              System.out.println(position.getItem( ) + " "
73                                          + position.getCount( ));
74              position = position.getLink( );
75          }
76      }
77  }
78
```

This last node is indicated by the `link` *field being equal to* `null`.

This is the way you traverse an entire linked list.

Working with Linked Lists

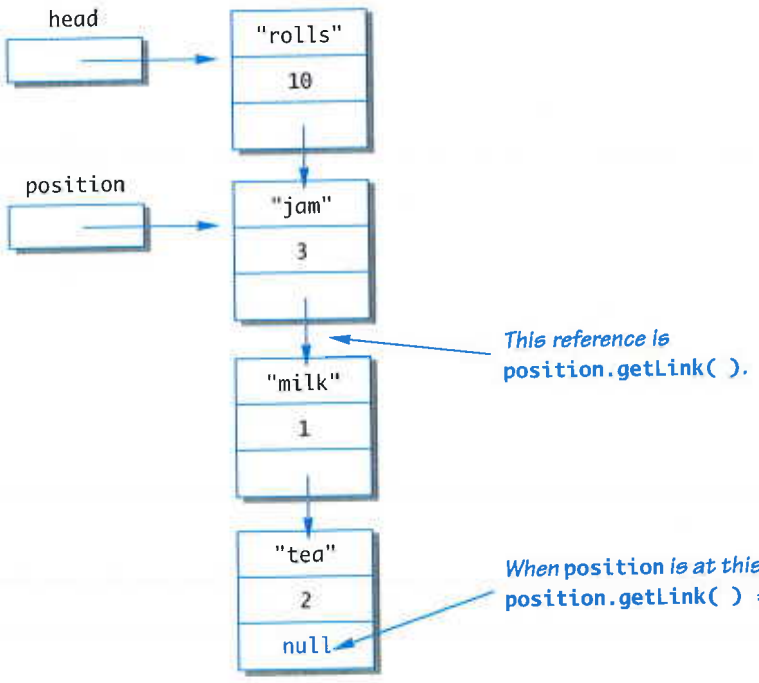
When dealing with a linked list, your code needs to be able to "get to" that first or head node, and you need some way to detect when the last node is reached. To get your code to the first node, you use a variable of type `Node1` that always contains a reference to the first node. In Display 15.3, the variable with a reference to the first node is named `head`. From that first or head node your code can follow the links through the linked list. But how does your code know when it is at the last node in a linked list?

In Java, you indicate the end of a linked list by setting the `link` instance variable of the last node in the linked list to `null`, as shown in Display 15.4. That way your code can test whether or not a node is the last node in a linked list by testing whether its link instance variable contains `null`. Remember that you check for a link being "equal" to `null` by using ==, not any equals method.

empty list

You also use `null` to indicate an empty linked list. The `head` instance variable contains a reference to the first node in the linked list, or it contains `null` if the linked list is empty (that is, if the linked list contains no nodes). The only constructor sets this `head` instance variable to `null`, indicating that a newly created linked list is empty.

Display 15.4 Traversing a Linked List



Indicating the End of a Linked List

The last node in a linked list should have its `link` instance variable set to `null`. That way, your code can check whether a node is the last node by checking whether its `link` instance variable is equal to `null`.

An Empty List Is Indicated by `null`

Suppose the variable `head` is supposed to contain a reference to the first node in a linked list. Linked lists usually start out empty. To indicate an empty linked list, you give the variable `head` the value `null`. This is traditional and works out nicely for many linked list manipulation algorithms.

traversing a linked list

Before we go on to discuss how nodes are added and removed from a linked list, let's suppose that the linked list already has a few nodes and that you want to write out the contents of all the nodes to the screen. You can do this with the method `outputList` (Display 15.3), whose body is reproduced here:

```
Node1 position = head;
while (position != null)
{
    System.out.println(position.getItem( ) + " "
                       + position.getCount( ));
    position = position.getLink( );
}
```

The method uses a local variable named `position` that contains a reference to one node. The variable `position` starts out with the same reference as the `head` instance variable, so it starts out positioned at the first node. The `position` variable then has its position moved from one node to the next with the assignment

```
position = position.getLink( );
```

This is illustrated in Display 15.4. To see that this assignment "moves" the `position` variable to the next node, note that the `position` variable contains a reference to the node pointed to by the position arrow in Display 15.4. So, `position` is a name for that node, and `position.link` is a name for the link to the next node. The value of `link` is produced with the accessor method `getLink`. Thus, a reference to the next node in the linked list is `position.getLink( )`. You "move" the position variable by giving it the value of `position.getLink( )`.

The method `outputList` continues to move the `position` variable down the linked list and outputs the data in each node as it goes along. When `position` reaches the last node, it outputs the data in that node and then again executes

```
position = position.getLink( );
```

If you study Display 15.4, you will see that when `position` leaves the last node, its value is set to `null`. At that point, we want to stop the loop, so we iterate the loop

```
while (position != null)
```

A similar technique is used to traverse the linked list in the methods `size` and `find`.

adding a node

Next let's consider how the method `addToStart` adds a node to the start of the linked list so that the new node becomes the first node in the list. It does this with the single statement

```
head = new Node1(itemName, itemCount, head);
```

The new node is created with

```
new Node1(itemName, itemCount, head)
```

which returns a reference to this new node. The assignment statement sets the variable `head` equal to a reference to this new node, making the new node the first node in the linked list. To link this new node to the rest of the list, we need only set the `link` instance variable of the new node equal to a reference to the *old first node*. But we have already done that: `head` used to point to the old first node, so if we use the name `head` on *the right-hand side of the assignment operator,* `head` will denote a reference to the old first node. Therefore, the new node produced by

```
new Node1(itemName, itemCount, head)
```

points to the old first node, which is just what we wanted. This is illustrated in Display 15.5.

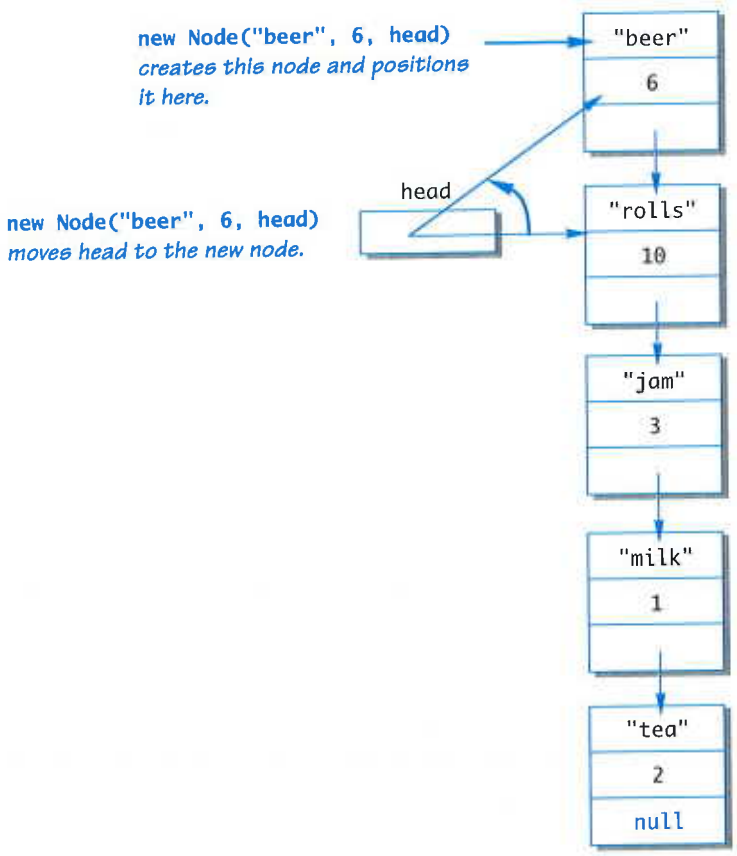
Later, we will discuss adding nodes at other places in a linked list, but the easiest place to add a node is at the start of the list. Similarly, the easiest place to delete a node is at the start of the linked list.

removing a node

The method `deleteHeadNode` removes the first node from the linked list and leaves the `head` variable pointing to (that is, containing a reference to) the old second node (which is now the first node) in the linked list. This is done with the following assignment:

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

Display 15.5 Adding a Node at the Start



This removes the first node from the linked list and leaves the linked list one node shorter. But what happens to the deleted node? At some point, Java will automatically collect it, along with any other nodes that are no longer accessible, and recycle the memory they occupy. This is known as **automatic garbage collection**.

automatic garbage collection

Display 15.6 contains a simple program that demonstrates how some of the methods in the class `LinkedList1` behave.

Display 15.6 A Linked List Demonstration

```
1  public class LinkedList1Demo
2  {
3      public static void main(String[] args)
4      {
5          LinkedList1 list = new LinkedList1( );
6          list.addToStart("Apples", 1);
7          list.addToStart("Bananas", 2);
8          list.addToStart("Cantaloupe", 3);
9          System.out.println("List has " + list.size( )
10                             + " nodes.");
11         list.outputList( );
12         if (list.contains("Cantaloupe"))
13             System.out.println("Cantaloupe is on list.");
14         else
15             System.out.println("Cantaloupe is NOT on list.");

16         list.deleteHeadNode( );

17         if (list.contains("Cantaloupe"))
18             System.out.println("Cantaloupe is on list.");
19         else
20             System.out.println("Cantaloupe is NOT on list.");

21         while (list.deleteHeadNode( ))
22             ; //Empty loop body
23         System.out.println("Start of list:");
24         list.outputList( );
25         System.out.println("End of list.");
26     }
27 }
```

Cantaloupe *is now in the head node.*

Empties the list. There is no loop body because the method **deleteHeadNode** *both performs an action on the list and returns a Boolean value.*

Sample Dialogue

```
List has 3 entries.
Cantaloupe 3
Bananas 2
Apples 1
Cantaloupe is on list.
Cantaloupe is NOT on list.
Start of list:
End of list.
```

Self-Test Exercises

1. What output is produced by the following code?

```
LinkedList1 list = new LinkedList1( );
list.addToStart("apple pie", 1);
list.addToStart("hot dogs", 12);
list.addToStart("mustard", 1);
list.outputList( );
```

2. Define a `boolean` valued method named `isEmpty` that can be added to the class `LinkedList1` (Display 15.3). The method returns `true` if the list is empty and `false` if the list has at least one node in it.

3. Define a `void` method named `clear` that can be added to the class `LinkedList1` (Display 15.3). The method has no parameters and it empties the list.

PITFALL: Privacy Leaks

It may help you to understand this section if you first review the Pitfall section of the same name in Chapter 5.

Consider the method `getLink` in the class `Node1` (Display 15.2). It returns a value of type `Node1`. That is, it returns a reference to a `Node1`. In Chapter 5, we said that if a method (such as `getLink`) returns a reference to an instance variable of a (mutable) class type, then the `private` restriction on the instance variable can easily be defeated because getting a reference to an object may allow a programmer to change the private instance variables of the object. There are a number of ways to fix this, the most straightforward of which is to make the class `Node1` a private inner class in the method `Node1`, as discussed in the next subsection.

There is no danger of a privacy leak with the class `Node1` when it is used in the class definition for `LinkedList1`. However, there is no way to guarantee that the class `Node1` will be used only in this way unless you take some precaution, such as making the class `Node1` a private inner class in the class `LinkedList1` `Node1`.

An alternate solution is to place both of the classes `Node1` and `LinkedList1` into a package, and change the `private` instance variable restriction to the `package` restriction as discussed in Chapter 7.

Note that this privacy problem can arise in any situation in which a method returns a reference to a private instance variable of a class type. The method `getItem( )` of the class `Node1` comes very close to having this problem. In this case, the method `getItem` causes no privacy leak, but only because the class `String` is not a mutable class (that is, it has no methods that will allow the user to change the value of the string without changing the reference). If instead of storing data of type `String` in our list we had stored data of some mutable class type, then defining an accessor method similarly to `getItem` would produce a privacy leak. ■

Node Inner Classes

You can make a linked list, or any similar data structures, self-contained by making the node class an inner class. In particular, you can make the class `LinkedList1` more self-contained by making `Node1` an inner class, as follows:

```
public class LinkedList1
{
    private class Node1
    {
      <The rest of the definition of Node1 can be
                                        the same as in Display 15.2.>

    }

    private Node1 head;
   <The constructor and methods in Display 15.3 are inserted here.>
}
```

Note that we've made the class `Node1` a private inner class. If an inner class is not intended to be used elsewhere, it should be made private. Making `Node1` a private inner class hides all objects of the inner class and avoids a privacy leak.

If you are going to make the class `Node1` a private inner class in the definition of `LinkedList1`, then you can safely simplify the definition of `Node1` by eliminating the accessor and mutator methods (the `set` and `get` methods) and just allowing direct access to the instance variables (`item`, `count`, and `link`) from methods of the outer class. In Display 15.7, we have written a class similar to `LinkedList1` in this way. The rewritten version, named `LinkedList2`, is like the class `LinkedList1` in Display 15.3 in that it has the same methods that perform basically the same actions. To keep the discussion simple, `LinkedList2` has only one data field instead of two. We could easily have retained the two data fields, but we wanted a notationally simple example without any distracting details. (See Self-Test Exercise 8 for a version that has the same kind of data in each node as in the nodes of `LinkedList1`.)

Display 15.7 A Linked List Class with a Node Inner Class (part 1 of 3)

```
1  public class LinkedList2
2  {
3      private class Node
4      {
5          private String item;
6          private Node link;

7          public Node( )
8          {
9              item = null;
10             link = null;
11         }
```

Display 15.7 A Linked List Class with a Node Inner Class (part 2 of 3)

```
12          public Node(String newItem, Node linkValue)
13          {
14              item = newItem;
15              link = linkValue;
16          }
17      }//End of Node inner class

18      private Node head;

19      public LinkedList2( )
20      {
21          head = null;
22      }
23      /**
24       Adds a node at the start of the list with the specified data.
25       The added node will be the first node in the list.
26      */
27      public void addToStart(String itemName)
28      {
29          head = new Node(itemName, head);
30      }

31      /**
32       Removes the head node and returns true if the list contained at least
33       one node. Returns false if the list was empty.
34      */
35      public boolean deleteHeadNode( )
36      {
37          if (head != null)
38          {
39              head = head.link;
40              return true;
41          }
42          else
43              return false;
44      }

45      /**
46       Returns the number of nodes in the list.
47      */
48      public int size( )
49      {
50          int count = 0;
51          Node position = head;
52          while (position != null)
53          {
54              count++;
```

(continued)

Display 15.7 A Linked List Class with a Node Inner Class (part 3 of 3)

```
55              position = position.link;
56          }
57          return count;
58      }

59      public boolean contains(String item)
60      {
61          return (find(item) != null);
62      }

63      /**
64       Finds the first node containing the target item, and returns a
65       reference to that node. If target is not in the list, null is returned.
66      */
67      private Node find(String target)
68      {
69          Node position = head;
70          String itemAtPosition;
71          while (position != null)
72          {
73              itemAtPosition = position.item;
74              if (itemAtPosition.equals(target))
75                  return position;
76              position = position.link;
77          }
78          return null; //target was not found
79      }
80      public void outputList( )
81      {
82          Node position = head;
83          while (position != null)
84          {
85              System.out.println(position.item );
86              position = position.link;
87          }
88      }

89      public boolean isEmpty( )
90      {
91          return (head == null);
92      }

93      public void clear( )
94      {
95          head = null;
96      }
97  }
```

Note that the outer class has direct access to the inner class's instance variables, such as `link`.

Self-Test Exercises

4. Would it make any difference if we changed the `Node` inner class in Display 15.7 from a private inner class to a public inner class?

5. Keeping the inner class `Node` in Display 15.7 as `private`, what difference would it make if any of the instance variables or methods in the class `Node` had its access modifiers changed from `private` to `public` or package access?

6. Why does the definition of the inner class `Node` in Display 15.7 not have the accessor and mutator methods `getLink`, `setLink`, or other `get` and `set` methods for the link fields similar to those in the class definition of `Node1` in Display 15.2?

7. Would it be legal to add the following method to the class `LinkedList2` in Display 15.7?

```
public Node startNode( )
{
    return head;
}
```

8. Rewrite the definition of the class `LinkedList2` in Display 15.7 so that it has data of a type named `Entry`, which is a public inner class. Objects of type `Entry` have two instance variables defined as follows:

```
private String item;
private int count;
```

This rewritten version of `LinkedList2` will be equivalent to `LinkedList1` in that it has the same methods doing the same things and it will hold equivalent data in its nodes.

EXAMPLE: A Generic Linked List

Display 15.8 shows a generic linked list with a type parameter `T` for the type of data stored in a node. This generic linked list has the same methods, coded in basically the same way, as our previous linked list (Display 15.7), but we have used a type parameter for the type of data in the nodes.

Display 15.10 contains a demonstration program for our generic linked list. The demonstration program uses the class `Entry`, defined in Display 15.9, as the type plugged in for the type parameter `T`. Note that if you want multiple pieces of data in each node, you simply use a class type that has multiple instance variables and plug in this class for the type parameter `T`.

Display 15.8 A Generic Linked List Class (part 1 of 3)

```
1  public class LinkedList3<T>
2  {
3      private class Node<T>
4      {
5          private T data;
6          private Node<T> link;
7          public Node( )
8          {
9               data = null;
10              link = null;
11         }

12         public Node(T newData, Node<T> linkValue)
13         {
14             data = newData;
15             link = linkValue;
16         }
17     }//End of Node<T> inner class

18     private Node<T> head;

19     public LinkedList3( )
20     {
21         head = null;
22     }

23     /**
24      Adds a node at the start of the list with the specified data.
25      The added node will be the first node in the list.
26     */
27     public void addToStart(T itemData)
28     {
29         head = new Node<T>(itemData, head);
30     }
31     /**
32      Removes the head node and returns true if the list contained at least
33      one node. Returns false if the list was empty.
34     */
35     public boolean deleteHeadNode( )
36     {
37         if (head != null)
38         {
39             head = head.link;
40             return true;
41         }
42         else
43             return false;
44     }
```

This linked list holds objects of type T. The type T should have well-defined **equals** *and* **toString** *methods.*

Display 15.8 A Generic Linked List Class (part 2 of 3)

```
45      /**
46       Returns the number of nodes in the list.
47      */
48      public int size( )
49      {
50          int count = 0;
51          Node<T> position = head;
52          while (position != null)
53          {
54              count++;
55              position = position.link;
56          }
57          return count;
58      }

59      public boolean contains(T item)
60      {
61          return (find(item) != null);
62      }
63      /**
64      Finds the first node containing the target item, and returns a
65      reference to that node. If target is not in the list, null is returned.
66      */
67      private Node<T> find(T target)
68      {
69          Node<T> position = head;
70          T itemAtPosition;
71          while (position != null)
72          {
73              itemAtPosition = position.data;
74              if (itemAtPosition.equals(target))
75                  return position;
76              position = position.link;
77          }
78          return null; //target was not found
79      }

80      /**
81       Finds the first node containing the target and returns a reference
82        to the data in that node. If target is not in the list, null is returned.
83      */
84      public T findData(T target)
85      {
86          Node<T> result = find(target);
87          if (result == null)
88              return null;
89          else
```

Type T must have well-defined **equals** *for this method to work.*

(continued)

Display 15.8 A Generic Linked List Class (part 3 of 4)

```
90              return result.data;
91          }
92          public void outputList( )
93          {
94              Node<T> position = head;
95              while (position != null)
96              {
97                  System.out.println(position.data);
98                  position = position.link;
99              }
100         }

101         public boolean isEmpty( )
102         {
103             return (head == null);
104         }

105         public void clear( )
106         {
107             head = null;
108         }
109         /*
110          For two lists to be equal they must contain the same data items in
111          the same order. The equals method of T is used to compare data items.
112         */
113         public boolean equals(Object otherObject)
114         {
115             if (otherObject == null)
116                 return false;
117             else if (getClass( ) != otherObject.getClass( ))
118                 return false;
119             else
120             {
121                 LinkedList3<T> otherList = (LinkedList3<T>)otherObject;
122                 if (size( ) != otherList.size( ))
123                     return false;
124                 Node<T> position = head;
125                 Node<T> otherPosition = otherList.head;
126                 while (position != null)
127                 {
128                     if (!(position.data.equals(otherPosition.data)))
129                         return false;
130                     position = position.link;
131                     otherPosition = otherPosition.link;
132                 }
133                 return true; //no mismatch was not found
134             }
135         }
136 }
```

Type T must have well-defined **toString** *methods for this to work.*

Change

3

missing rule

Display 15.9 A Sample Class for the Data in a Generic Linked List

```
1  public class Entry
2  {
3      private String item;
4      private int count;

5      public Entry(String itemData, int countData)
6      {
7          item = itemData;
8          count = countData;
9      }

10     public String toString( )
11     {
12         return (item + " " + count);
13     }

14     public boolean equals(Object otherObject)
15     {
16         if (otherObject == null)
17             return false;
18         else if (getClass( ) != otherObject.getClass( ))
19             return false;
20         else
21         {
22             Entry otherEntry = (Entry)otherObject;
23             return (item.equals(otherEntry.item)
24                       && (count == otherEntry.count));
25         }
26     }

       <There should be other constructors and methods, including accessor and
            mutator methods, but we do not use them in this demonstration.>
27 }
```

Display 15.10 A Generic Linked List Demonstration

```
1  public class GenericLinkedListDemo
2  {
3      public static void main(String[] args)
4      {
5          LinkedList3<Entry> list = new LinkedList3<Entry>( );
6
7          Entry entry1 = new Entry("Apples", 1);
8          list.addToStart(entry1);
9          Entry entry2 = new Entry("Bananas", 2);
10         list.addToStart(entry2);
11         Entry entry3 = new Entry("Cantaloupe", 3);
12         list.addToStart(entry3);
13         System.out.println("List has " + list.size( )
14                                + " nodes.");
15         list.outputList( );
16         System.out.println("End of list.");
17     }
18 }
```

Sample Dialogue

```
List has 3 nodes.
Cantaloupe 3
Bananas 2
Apples 1
End of list.
```

bitmapped

PITFALL: Using Node Instead of Node<T>

This pitfall is explained by example, using the LinkedList3<T> class in Display 15.8. However, the lesson applies to any generic linked structure with a node inner class. The type parameter need not be T and the node class name need not be Node, but for simplicity, we will use T and Node.

When defining the LinkedList3<T> class in Display 15.8, the type for a node is Node<T>; it is not Node. However, it is easy to forget the type specification <T> and write Node instead of Node<T>. If you omit the <T>, you may or may not get a compiler error message, depending on other details of your code. If you do get a compiler error message, it is likely to seem bewilderingly strange. The problem is that Node actually means something. (We do not have time to stop and explain what Node means, but it means something similar to a node with data type Object, rather than data type T.) Your only defense against this pitfall is to be very careful, and if you do get a bewildering compiler error message, look for a missing <T>.

bitmapped

PITFALL: (continued)

Sometimes a compiler warning message can be helpful when you make this mistake. If you get a warning that mentions a type cast from `Node` to `Node<T>`, look for an omitted `<T>`.

Finally, we should note that sometimes your code will compile and even run correctly if you omit the `<T>` from `Node<T>`. ■

The `equals` Method for Linked Lists

equals

The linked lists we presented in Displays 15.3 and 15.7 did not have an `equals` method. We did that to keep the examples simple and not detract from the main message. However, a linked list class should normally have an `equals` method.

There is more than one approach to defining a reasonable `equals` method for a linked list. The two most obvious are the following:

1. Two linked lists are equal if they contain the same data entries (possibly ordered differently).
2. Two linked lists are equal if they contain the same data entries in the same order; that is, the data in the first node of the calling object equals the data in the first node of the other linked list, the data in the two second nodes are equal, and so forth.

It is not true that one of these is the correct approach to defining an `equals` method and the other is incorrect. In different situations, you might want different definitions of `equals`. However, the most common way to define `equals` for a linked list is approach 2. A definition of `equals` that follows approach 2 and that can be added to the class `LinkedList2` in Display 15.7 is given in Display 15.11. The generic linked list in Display 15.8 also contains an `equals` method that follows approach 2.

Note that when we define `equals` for our linked list with type parameter `T`, we are trusting the programmer who wrote the definition for the type plugged in for `T`. We are assuming the programmer has redefined the `equals` method so that it provides a reasonable test for equality. Situations like this are the reason it is so important to always include an `equals` method in the classes you define.

Display 15.11 An `equals` Method for the Linked List in Display 15.7 (part 1 of 2)

```
1   /*
2    For two lists to be equal they must contain the same data items in
3    the same order.
4   */
5   public boolean equals(Object otherObject)
6    {
7       if (otherObject == null)
8           return false;
9       else if (getClass( ) != otherObject.getClass( ))
10          return false;
```

(continued)

Display 15.11 An equals Method for the Linked List in Display 15.7 (part 2 of 2)

```
11          else
12          {
13              LinkedList2 otherList = (LinkedList2)otherObject;
14              if (size( ) != otherList.size( ))
15                  return false;
16              Node position = head;
17              Node otherPosition = otherList.head;
18              while (position != null)
19              {
20                  if ( (!(position.item.equals(otherPosition.item))))
21                     return false;
22                  position = position.link;
23                  otherPosition = otherPosition.link;
24              }
25              return true; //A mismatch was not found
26          }
27      }
```

15.2 Copy Constructors and the clone Method ★

There are three ways to do anything:
The right way,
the wrong way,
and the army way.

Advice reputedly given to new army recruits

The way Java handles cloning, and object copying in general, is complicated and can be both subtle and difficult. Some authorities think that the clone method was done so poorly in Java that they prefer to ignore it completely and define their own methods for copying objects. I have some sympathy for that view, but before you dismiss Java's approach to cloning, it might be a good idea to see what the approach entails. Linked data structures, such as linked lists, are an excellent setting for discussing cloning because they are an excellent setting for discussing deep versus shallow copying.

This section first presents a relatively simple way to define copy constructors and the clone method, but this approach unfortunately produces only shallow copies. We then go on to present one way to produce a deep copy clone method and to do so within the official prescribed rules of the Java documentation.

Readers with very little programming experience may be better off skipping this entire section until they become more comfortable with Java. Other readers may prefer to read only the first subsection and possibly the immediately following Pitfall subsection.

Simple Copy Constructors and `clone` Methods ★

Display 15.12 contains a copy constructor and `clone` method definitions that could be added to the definition of the generic linked list class in Display 15.8. The real work is done by the private helping method `copyOf`, so our discussion focuses on the method `copyOf`.

The private method `copyOf` takes an argument that is a reference to the head node of a linked list and returns a reference to the head node of a copy of that linked list. The easiest way to do this would be to simply return the argument. This would, however, simply produce another name for the argument list. We do not want another name; we want another list. So, the method goes down the argument list one node at a time (with `position`) and makes a copy of each node. The linked list of the calling object is built up node by node by adding these new nodes to its linked list. However, there is a complication. We cannot simply add the new nodes at the head (start) end of the list being built. If we did, then the nodes would end up in the reverse of the desired order. So, the new nodes are added to the end of the linked list being built. The variable `end` of type `Node<T>` is kept positioned at the last node so that it is possible to add nodes at the end of the linked list being built. In this way a copy of the list in the calling object is created so that the order of the nodes is preserved.

The copy constructor is defined by using the private helping method `copyOf` to create a copy of the list of nodes. Other details of the copy constructor and the `clone` method are done in the standard way.

Although the copy constructor and the `clone` method each produce a new linked list with all new nodes, the new list is not truly independent because the data objects are not cloned. See the next Pitfall section for a discussion of this point. One way to fix this shortcoming is discussed in the Programming Tip subsection entitled "Use a Type Parameter Bound for a Better `clone`."

Exceptions ★

A generic data structure, such as the class `LinkedList` in Display 15.12, is likely to have methods that throw exceptions. Situations such as a `null` argument to the copy constructor might be handled differently in different situations, so it is best to throw a `NullPointerException` if this happens and let the programmer who is using the linked list handle the exception. This is what we did with the copy constructor in Display 15.12. A `NullPointerException` is an unchecked exception, which means that it need not be caught or declared in a `throws` clause. When thrown by a method of a linked list class, it can be treated simply as a run-time error message. The exception can instead be caught in a `catch` block if there is some suitable action that can be taken.

Display 15.12 A Copy Constructor and `clone` Method for a Generic Linked List (part 1 of 2)

```
1  public class LinkedList3<T> implements Cloneable
2  {
3      private class Node<T>
4      {
5          private T data;
6          private Node<T> link;

7          public Node( )
8          {
9               data = null;
10              link = null;
11         }

12         public Node(T newData, Node<T> linkValue)
13         {
14             data = newData;
15             link = linkValue;
16         }
17     }//End of Node<T> inner class

18     private Node<T> head;

   <All the methods from Display 15.8 are in the class definition,
                        but they are not repeated in this display.>

19     /**
20      Produces a new linked list, but it is not a true deep copy.
21      Throws a NullPointerException if other is null.
22    */
23     public LinkedList3(LinkedList3<T> otherList)
24     {
25         if (otherList == null)
26             throw new NullPointerException( );
27         if (otherList.head == null)
28             head = null;
29         else
30             head = copyOf(otherList.head);
31     }
32
33
```

This copy constructor and this **clone** *method do not make deep copies. We discuss one way to make a deep copy in the Programming Tip subsection "Use a Type Parameter Bound for a Better* **clone**.*"*

Display 15.12 A Copy Constructor and clone Method for a Generic Linked List (part 2 of 2)

```
34      public LinkedList3<T> clone( )
35      {
36          try
37          {
38              LinkedList3<T> copy =
39                                  (LinkedList3<T>)super.clone( );
40              if (head == null)
41                  copy.head = null;
42              else
43                  copy.head = copyOf(head);
44              return copy;
45          }
46          catch(CloneNotSupportedException e)
47          {//This should not happen.
48              return null; //To keep the compiler happy.
49          }
50      }

51      /*
52        Precondition: otherHead != null
53        Returns a reference to the head of a copy of the list
54        headed by otherHead. Does not return a true deep copy.
55      */
56      private Node<T> copyOf(Node<T> otherHead)
57      {
58          Node<T> position = otherHead;//moves down other's list.
59          Node<T> newHead; //will point to head of the copy list.
60          Node<T> end = null; //positioned at end of new growing list.
                                    Invoking clone with position.data would be illegal.
61          //Create first node:
62          newHead =
63                new Node<T>(position.data, null);
64          end = newHead;
65          position = position.link;

66          while (position != null)
67          {//copy node at position to end of new list.
68              end.link =
69                  new Node<T>(position.data, null);
70              end = end.link;
71              position = position.link;
72          }
                                    Invoking clone with position.data
                                    would be illegal.
73          return newHead;
74      }
75  }
```

PITFALL: The clone Method Is Protected in Object ★

When defining the copy constructor and clone method for our generic linked list (Display 15.12), we would have liked to have cloned the data in the list being copied. We would have liked to change the code in the helping method copyOf by adding invocations of the clone method as follows:

```
newHead =
      new Node((T)(position.data).clone( ), null);
end = newHead;
position = position.link;

while (position != null)
 {//copy node at position to end of new list.
    end.link =
        new Node((T)(position.data).clone( ), null);
    end = end.link;
    position = position.link;
}
```

This code is identical to code in copyOf except for the addition of the invocations of clone shown in red and the type casts shown in red. (The type casts are needed because Java thinks clone returns a value of type Object.)

If this modified code (with the clone method) would compile (and if the type plugged in for T has a well-defined clone method that makes a deep copy), then this modified code would produce a truly independent linked list with no references in common with the list being copied. Unfortunately, this code will not compile.

If you try to compile this code, you will get an error message saying that the method clone is protected in the class Object. True, we used the type T, not the type Object, but any class can be plugged in for T. So when the generic linked list is compiled, all Java knows about the type T is that it is a descendent class of Object. Since the designers of the Object class chose to make the method clone protected, you simply cannot use the clone method in the definition of methods such as copyOf.

Why was the clone method labeled protected in Object? Apparently for security reasons. If a class could use the clone method unchanged from Object, then that would open the possibility of copying sections of memory unchanged and unchecked and so might give unauthorized memory access. The problem is made more serious by the fact that Java is used to run programs on other machines across the Internet.

The way Java defines the clone method in Object and the way it specifies how clone should be defined in other classes is controversial. Do not be surprised if some future version of Java handles the clone method differently. But for now, you are stuck with these clone problems.

In many situations, the version of copyOf in Display 15.12 (without the use of clone) is good enough, but there is a way to get a true deep copy. One way to get a deep copy is to somehow restrict the type T to classes that do have a public clone method that makes a deep copy. Something like this can be done and is discussed in the Programming Tip "Use a Type Parameter Bound for a Better clone." ■

TIP: Use a Type Parameter Bound for a Better clone ★

One way to overcome the problem discussed in the previous Pitfall section is to place a bound on the type parameter T (in Display 15.12) so that it must satisfy some suitable interface. There is no standard interface that does the job, but it is very easy to define such an interface. The interface `PubliclyCloneable` given in Display 15.13 is just the interface we need. This short, simple interface guarantees all that we need to define generic linked lists whose clone method returns a deep copy.

Note that any class that implements the `PubliclyCloneable` interface has the following three properties:

1. The class implements the `Cloneable` interface. (This happens automatically because `PubliclyCloneable` extends `Cloneable`.)
2. The class has a public `clone` method.
3. The `clone` method for the class makes a deep copy (in the officially sanctioned way).

Condition 3 is not enforced by the Java compiler or run-time software, but like all interface semantics, it is the responsibility of the programmer defining the class to ensure that condition 3 is satisfied.

It is now easy to define our generic linked list whose `clone` method produces a deep copy. The definition is given in Display 15.14. We have already discussed the main points involved in this definition. The following Programming Example subsection discusses some of the minor, but possibly unclear, details of the definition.[1] ■

Display 15.13 The `PubliclyCloneable` Interface

```
1  /*
2   The programmer who defines a class implementing this interface
3   has the responsibility to define clone so it makes a deep copy
4   (in the officially sectioned way.)
5  */

6  public interface PubliclyCloneable extends Cloneable
7  {
8      public Object clone( );
9  }
```

Any class that implements **PubliclyCloneable** *must have a public* **clone** *method.*

Any class that implements **PubliclyCloneable** *automatically implements* **Cloneable.**

[1] You might wonder whether we could use a type parameter in the `PubliclyCloneable` interface and so avoid some type casts in the definition `copyOf`. We could do that, but that may be more trouble than it is worth and, at this introductory level of presentation, would be an unnecessary distraction.

Display 15.14 A Generic Linked List with a Deep Copy clone Method (part 1 of 3)

```
1  public class LinkedList<T extends PubliclyCloneable>
2                                          implements PubliclyCloneable
3  {
4      private class Node<T>
5      {
6          private T data;
7          private Node<T> link;

8          public Node( )
9          {
10             data = null;
11             link = null;
12         }

13         public Node(T newData, Node<T> linkValue)
14         {
15             data = newData;
16             link = linkValue;
17         }
18     }//End of Node<T> inner class

19     private Node<T> head;

20     public LinkedList( )
21     {
22         head = null;
23     }
24     /**
25      Produces a new linked list, but it is not a true deep copy.
26      Throws a NullPointerException if other is null.
27     */
28     public LinkedList(LinkedList<T> otherList)
29     {
30         if (otherList == null)
31             throw new NullPointerException( );
32         if (otherList.head == null)
33             head = null;
34         else
35             head = copyOf(otherList.head);
36     }
37
38     public LinkedList<T> clone( )
39     {
40         try
41         {
42             LinkedList<T> copy =
43                                 (LinkedList<T>)super.clone( );
```

Display 15.14 A Generic Linked List with a Deep Copy **clone** Method (part 2 of 3)

```
44              if (head == null)
45                  copy.head = null;
46              else
47                  copy.head = copyOf(head);
48              return copy;
49          }
50          catch(CloneNotSupportedException e)
51          {//This should not happen.
52              return null; //To keep the compiler happy.
53          }
54      }

55       /*
56        Precondition: otherHead != null
57        Returns a reference to the head of a copy of the list
58        headed by otherHead. Returns a true deep copy.
59      */
60      private Node<T> copyOf(Node<T> otherHead)
61      {
62          Node<T> position = otherHead; //moves down other's list.
63          Node<T> newHead; //will point to head of the copy list.
64          Node<T> end = null; //positioned at end of new growing list.

65          //Create first node:
66          newHead =
67                new Node<T>((T)(position.data).clone( ), null);
68                end = newHead;
69          position = position.link;
70          while (position != null)
71          {//copy node at position to end of new list.
72              end.link =
73                  new Node<T>((T)(position.data).clone( ), null);
74              end = end.link;
75              position = position.link;
76          }

77          return newHead;
78      }
79
80      public boolean equals(Object otherObject)
81      {
82          if (otherObject == null)
83              return false;
84          else if (getClass( ) != otherObject.getClass( ))
85              return false;
86          else
87          {
88              LinkedList<T> otherList = (LinkedList<T>)otherObject;
```

This definition of **copyOf** *gives a deep copy of the linked list.*

(continued)

Display 15.14 A Generic Linked List with a Deep Copy `clone` Method (part 3 of 3)

```
        <The rest of the definition is the same as in Display 15.8. The only difference
        between this definition of equals and the one in Display 15.8 is that we
        have replaced the class name LinkedList3<T> with LinkedList<T>.>

89      }

      <All the other methods from Display 15.8 are in the class definition,
       but are not repeated in this display. >

90      public String toString( )
91      {
92          Node<T> position = head;
93          String theString = "";
94          while (position != null)
95          {
96              theString = theString + position.data + "\n";
97              position = position.link;
98          }
99          return theString;
100     }
101 }
```

We added a **`toString`** *method so* **`LinkedList<T>`** *would have all the properties we want* `T` *to have.*

EXAMPLE: A Linked List with a Deep Copy `clone` Method

We have already discussed how and why the `clone` method of the generic linked list class in Display 15.14 returns a deep copy. Let's now look at some of the other details and see an example of using this linked list class.

Note the definition of the `clone` method. Why did we not simplify it to the following?

```
public LinkedList<T> clone( )
{
    return new LinkedList<T>(this);
}
```

This simple, alternative definition would still return a deep copy of the linked list and would work fine in most situations. It is likely that you would not notice any difference if you used this definition of `clone` in place of the one given in Display 15.14.

The only reason for all the other detail in the `clone` method definition given in Display 15.14 is to define the `clone` method as specified in the Java documentation. The reason that the Java documentation asks for those details has to do with security issues. (Some might say that there are three ways to define a `clone` method: the right way, the wrong way, and the Java way. This extra detail is the Java way.)

EXAMPLE: (continued)

If you look only quickly at Display 15.14 you might think the following at the start of the definition is an unimportant detail:

```
implements PubliclyCloneable
```

However, it ensures that the linked list class implements the `Cloneable` interface. In order for a class to have a Java-approved `clone` method, it must implement the `Cloneable` interface. It also allows you to make linked lists of linked lists and have a deep copy `clone` method in the linked list of linked lists.

A sample class that implements the `PubliclyCloneable` interface is given in Display 15.15. Display 15.16 shows a demonstration program that makes a deep copy `clone` of a linked list of objects of this sample class.

Display 15.15 A `PubliclyCloneable` Class (part 1 of 2)

```
1  public class StockItem implements PubliclyCloneable
2  {
3      private String name;
4      private int number;

5      public StockItem( )
6      {
7          name = null;
8          number = 0;
9      }

10     public StockItem(String nameData, int numberData)
11     {
12         name = nameData;
13         number = numberData;
14     }

15     public void setNumber(int newNumber)
16     {
17         number = newNumber;
18     }
```

(continued)

Display 15.15 A `PubliclyCloneable` Class (part 2 of 2)

```
19      public void setName(String newName)
20      {
21          name = newName;
22      }

23      public String toString( )
24      {
25          return (name + " " + number);
26      }

27      public Object clone( )
28      {
29         try
30         {
31            return super.clone( );
32         }
33         catch(CloneNotSupportedException e)
34         {//This should not happen.
35            return null; //To keep compiler happy.
36         }
37      }
38

39      public boolean equals(Object otherObject)
40      {
41          if (otherObject == null)
42              return false;
43          else if (getClass( ) != otherObject.getClass( ))
44              return false;
45          else
46          {
47              StockItem otherItem = (StockItem) otherObject;
48              return (name.equalsIgnoreCase(otherItem.name)
49                        && number == otherItem.number);
50          }
51      }
52  }
```

Display 15.16 Demonstration of Deep Copy clone

```
1  public class DeepDemo
2  {
3      public static void main(String[] args)
4      {
5          LinkedList<StockItem> originalList =
6                                      new LinkedList<StockItem>( );
7          originalList.addToStart(new StockItem("red dress", 1));
8          originalList.addToStart(new StockItem("black shoe", 2));

9          LinkedList<StockItem> copyList = originalList.clone( );
10         if (originalList.equals(copyList))
11             System.out.println("OK, Lists are equal.");

12         System.out.println("Now we change copyList.");
13         StockItem dataEntry =
14                    copyList.findData(new StockItem("red dress", 1));
15         dataEntry.setName("orange pants");

16         System.out.println("originalList:");
17         originalList.outputList( );

18         System.out.println("copyList:");
19         copyList.outputList( );
20
21         System.out.println("Only one list is changed.");
22     }
23 }
```

Sample Dialogue

```
OK, Lists are equal.
Now we change copyList.
originalList:
black shoe 2
red dress 1
copyList:
black shoe 2
orange pants 1
Only one list is changed.
```

TIP: Cloning Is an "All or Nothing" Affair

If you define a `clone` method, then you should do so following the official Java guidelines, as we did in Display 15.14. In particular, you should always have the class implement the `Cloneable` interface. If you define a `clone` method in any other way, you may encounter problems in some situations. If you want to have a method for producing copies of objects but do not want to follow the official guidelines on how to define a `clone` method, then use some other name for your "`clone`-like" method, such as `copier`, or make do with just a copy constructor. ■

Self-Test Exercises

9. In the definition of `copyOf` in Display 15.14, can we replace

```
newHead =
        new Node<T>((T)(position.data).clone( ), null);
```

with the following, which uses the copy constructor of `T` instead of the clone method of `T`?

```
newHead =
        new Node<T>(new T(position.data), null);
```

10. The definition of the `clone` method in Display 15.14 returns a value of type `LinkedList<T>`. But the class being defined implements the `PubliclyCloneable` interface, and that interface says the value returned must be of type `Object`. Is something wrong?

15.3 Iterators

Play it again, Sam.

Attributed (incorrectly) to the movie *Casablanca*, which contains similar lines.[2]

When you have a collection of objects, such as the nodes of a linked list, you often need to step through all the objects in the collection one at a time and perform some action on each object, such as writing it out to the screen or in some way editing the data in each object. An **iterator** is any object that allows you to step through the list in this way.

iterator

[2] There is a Woody Allen movie with this title, but it is based on the misquote from *Casablanca*, which was in common use before the movie came out.

Defining an Iterator Class

In Display 15.17, we have rewritten the class `LinkedList2` from Display 15.7 so that it has an inner class for iterators and a method `iterator( )` that returns an iterator for its calling object. We have made the inner class `List2Iterator` public so that we can have variables of type `List2Iterator` outside the class `LinkedList2`, but we do not otherwise plan to use the inner class `List2Iterator` outside of the outer class `LinkedList2`.

Use of iterators for the class `LinkedList2` is illustrated by the program in Display 15.18. Note that, given a linked list named `list`, an iterator for `list` is produced by the method `iterator` as follows:

```
LinkedList2.List2Iterator i = list.iterator( );
```

The iterator `i` produced in this way can only be used with the linked list named `list`. Be sure to notice that outside of the class, the type name for the inner class iterator must include the name of the outer class as well as the inner iterator class. The class name for one of these iterators is

```
LinkedList2.List2Iterator
```

Display 15.17 A Linked List with an Iterator (part 1 of 3)

```
1  import java.util.NoSuchElementException;

2  public class LinkedList2
3  {
4      private class Node
5      {
6          private String item;
7          private Node link;

       <The rest of the definition of the Node inner class is given in Display 15.7.>

8      }//End of Node inner class

9      /**
10      If the list is altered any iterators should invoke restart or
11      the iterator's behavior may not be as desired.
12     */
13     public class List2Iterator
14     {
15         private Node position;
16         private Node previous; //previous value of position
```

This is the same as the class in Displays 15.7 and 15.11 except that the **`List2Iterator`** *inner class and the* **`iterator( )`** *method have been added.*

An inner class for iterators for **`LinkedList2`**.

(continued)

Display 15.17 A Linked List with an Iterator (part 2 of 3)

```
17          public List2Iterator( )
18          {
19              position = head; //Instance variable head of outer class.
20              previous = null;
21          }

22          public void restart( )
23          {
24              position = head; //Instance variable head of outer class.
25              previous = null;
26          }

27          public String next( )
28          {
29              if (!hasNext( ))
30                  throw new NoSuchElementException( );

31              String toReturn = position.item;
32              previous = position;
33              position = position.link;
34              return toReturn;
35          }

36          public boolean hasNext( )
37          {
38              return (position != null);
39          }
40          /**
41           Returns the next value to be returned by next( ).
42           Throws an IllegalStateExpression if hasNext( ) is false.
43          */
44          public String peek( )
45          {
46               if (!hasNext( ))
47                  throw new IllegalStateException( );
48               return position.item;
49          }

50          /**
51           Adds a node before the node at location position.
52           previous is placed at the new node. If hasNext( ) is
53           false, then the node is added to the end of the list.
54           If the list is empty, inserts node as the only node.
55          */
56          public void addHere(String newData)
57          {
58              if (position == null && previous != null)
59                    // at end of the list, add to end
60                    previous.link = new Node(newData, null);
```

Display 15.17 A Linked List with an Iterator (part 3 of 3)

```
61              else if (position == null || previous == null)
62                    // list is empty or position is head node
63                    LinkedList2.this.addtoStart(newData);
64              else
65              {  // previous and position are consecutive nodes
66                    Node temp = new Node(newData, position)
67                    previous.link = temp;
68                    previous = temp;
69              }
70          }
71          /**
72           Changes the String in the node at location position.
73           Throws an IllegalStateException if position is not at a node,
74          */
75          public void changeHere(String newData)
```

<The rest of the method changeHere is Self-Test Exercise 13.>

```
76          /**
77           Deletes the node at location position and
78           moves position to the "next" node.
79           Throws an IllegalStateException if the list is empty.
80          */
81          public void delete( )
82          {
83              if (position == null)
84                    throw new IllegalStateException( );
85              else if (previous == null)
86              {     // remove node at head
87                    head = head.link;
88                    position = head;
89              }
90              else // previous and position are consecutive nodes
91              {
92                    previous.link = position.link;
93                    position = position.link;
94              }
95          }
96      private Node head;

97      public List2Iterator iterator( )
98      {
99          return new List2Iterator( );
100     }
```

If `list` *is an object of the class* `LinkedList2`*, then* `list.iterator( )` *returns an iterator for list.*

<The other methods and constructors are identical to those in Displays 15.7 and 15.11.>

```
101 }
```

Display 15.18 Using an Iterator (part 1 of 2)

```
1  public class IteratorDemo
2  {
3      public static void main(String[] args)
4      {
5          LinkedList2 list = new LinkedList2( );
6          LinkedList2.List2Iterator i = list.iterator( );

7          list.addToStart("shoes");
8          list.addToStart("orange juice");
9          list.addToStart("coat");

10         System.out.println("List contains:");
11         i.restart( );
12         while(i.hasNext( ))
13             System.out.println(i.next( ));
14         System.out.println( );

15         i.restart( );
16         i.next( );
17         System.out.println("Will delete the node for " + i.peek( ));
18         i.delete( );

19         System.out.println("List now contains:");
20         i.restart( );
21         while(i.hasNext( ))
22             System.out.println(i.next( ));
23         System.out.println( );

24         i.restart( );
25         i.next( );
26         System.out.println("Will add one node before " + i.peek( ));
27         i.addHere("socks");
28         System.out.println("List now contains:");
29         i.restart( );
30         while(i.hasNext( ))
31             System.out.println(i.next( ));
32         System.out.println( );
33         System.out.println("Changing all items to credit card.");
34         i.restart( );
35         while(i.hasNext( ))
36         {
37             i.changeHere("credit card");
38             i.next( );
39         }
40         System.out.println( );
```

Display 15.18 Using an Iterator (part 2 of 2)

```
41          System.out.println("List now contains:");
42          i.restart( );
43          while(i.hasNext( ))
44              System.out.println(i.next( ));
45          System.out.println( );
46      }
47  }
```

Sample Dialogue

```
List contains:
coat
orange juice
shoes

Will delete the node for orange juice
List now contains:
coat
shoes

Will add one node before shoes
List now contains:
coat
socks
shoes

Changing all items to credit card.

List now contains:
credit card
credit card
credit card
```

The basic method for cycling through the elements in the linked list using an iterator is illustrated by the following code from the demonstration program:

```
System.out.println("List now contains:");
i.restart( );
while(i.hasNext( ))
    System.out.println(i.next( ));
```

The iterator is named `i` in this code. The iterator `i` is reset to the beginning of the list with the method invocation `i.restart( )`, and each execution of `i.next( )` produces the next data item in the linked list. After all the data items in all the nodes have been returned by `i.next( )`, the Boolean `i.hasNext( )` becomes `false` and the `while` loop ends.

Internally, the local variable `position` references the current node in the linked list, whereas the local variable `previous` references the node linking to the current node. The purpose of the `previous` variable will be seen when adding and deleting nodes. In the constructor and the `restart( )` method, `position` is set to `head` and `previous` is set to `null`.

To determine if the end of the list has been reached, `hasNext( )` returns whether or not `position` is `null`:

```
return (position != null);
```

To step through the list, the `next( )` method first throws an exception if we have reached the end of the list:

```
if (!hasNext( ))
throw new NoSuchElementException( );
```

Otherwise, the method retrieves the string value of the iterator referenced by `position` in the variable `toReturn`, advances `previous` to reference the current position, advances `position` to the next node in the list, and returns the string:

```
String toReturn = position.item;
previous = position;
position = position.link;
return toReturn;
```

The definition of the method `changeHere` is left as a Self-Test Exercise. (If necessary you can look up the definition in the answer to the Self-Test Exercise.) The techniques for adding and deleting nodes are discussed in the next subsection.

The Java `Iterator` Interface

Java has an interface named `Iterator` that specifies how Java would like an iterator to behave. It is in the package `java.util` (and so requires that you import this package). Our iterators do not quite satisfy this interface, but they are in the same general spirit as that interface and could be easily redefined to satisfy the `Iterator` interface.

The `Iterator` interface is discussed in Chapter 16.

Adding and Deleting Nodes

To add or delete a node in a linked list, you normally use an iterator and add or delete a node at the (approximate) location of the iterator. Since deleting is a little easier than adding a node, we will discuss deleting first.

Display 15.19 shows the technique for deleting a node. The linked list is an object of the class `LinkedList2` (Display 15.17). The variables `position` and `previous` are the instance variables of an iterator for the linked list object. These variables each hold a reference to a node, indicated with an arrow. Each time `next( )` is invoked, `previous` and

position reference subsequent nodes in the list. As indicated in Display 15.19, the node at location position is deleted by the following two lines of code:

```
previous.link = position.link;
position = position.link;
```

In Display 15.19, next( ) has been invoked twice, so position is referencing the node with "shoes" and previous is referencing the node with "socks".

To delete the node referenced by position, the link from the previous node is set to positions link. As shown in Display 15.19, this removes the linked list's reference to that node. The variable position is then set to the next node in the list to remove any references to the deleted node. As far as the linked list is concerned, the old node is no longer on the linked list. But the node is still in the computer's memory. If there are no longer any references to the deleted node, then the storage that it occupies should be made available for other uses. In many programming languages, you, the programmer, must keep track of items such as deleted nodes and must give explicit commands to return their memory for recycling. This is called **garbage collecting** or **explicit memory management**. In Java, this is done for you automatically, or, as it is ordinarily phrased, Java has automatic garbage collection.

garbage collecting
explicit memory management

Note that there are special cases that must be handled for deletion. First, if the list is empty, then nothing can be deleted and the delete( ) method throws an exception. Second, if the node to delete is the head of the list, then there is no previous node to update. Instead, head is set to head.link to bypass the first node in the list and set a new head node.

Display 15.20 shows the technique for adding a node. We want to add a new node between the nodes named by previous and position. In Display 15.20, previous and position are variables of type Node, and each contains a reference to a node indicated with an arrow. Thus, the new node goes between the two nodes referenced by previous and position. In Display 15.20, the method next( ) has been invoked twice to advance previous to "orange juice" and position to "shoes".

A constructor for the class Node does a lot of the work for us: It creates the new node, adds the data, and sets the link field of the new node to reference the node named by position. All this is done with the following:

```
new Node(newData, position)
```

So that we can recognize the node with newData in it when we study Display 15.20, let's assume that newData holds the string "socks". The following gets us from the first to the second picture:

```
temp = new Node(newData, position);
```

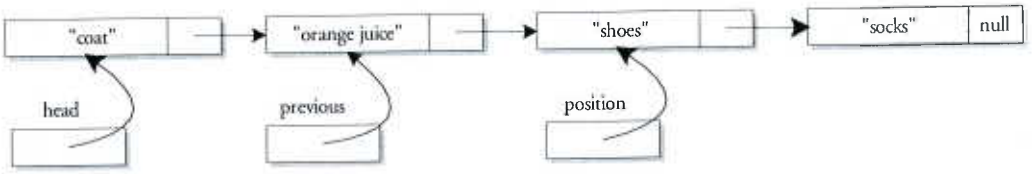
To finish the job, all we need to do is link the previous node to the new node. We want to move the arrow to the node named by temp. The following finishes our job:

```
previous.link = temp;
```

The new node is inserted in the desired place, but the picture is not too clear. The fourth picture is the same as the third one; we have simply redrawn it to make it neater.

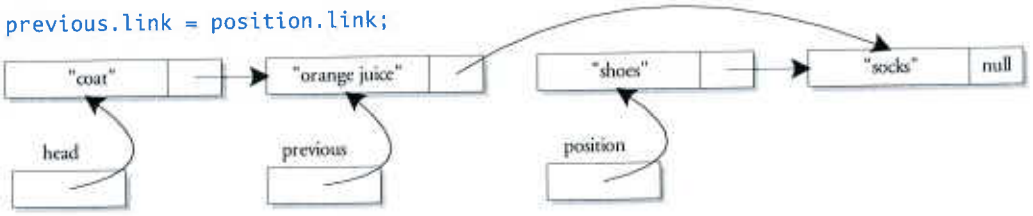
Display 15.19 Deleting a Node

1. Existing list with the iterator positioned at "shoes"



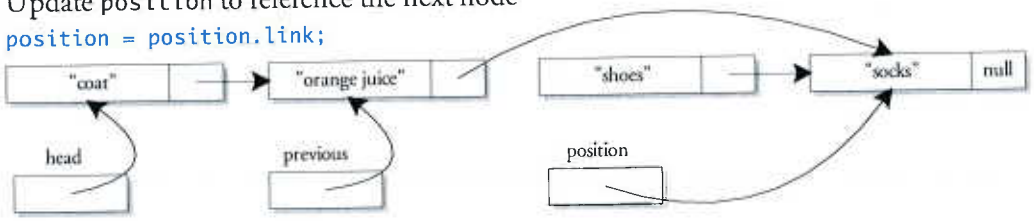
2. Bypass the node at `position` from `previous`

```
previous.link = position.link;
```



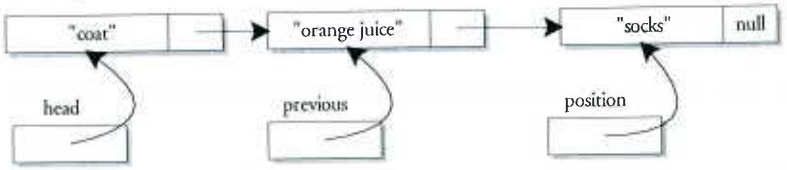
3. Update `position` to reference the next node

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



Since no variable references the node "shoes" Java will automatically recycle the memory allocated for it.

4. Same picture with deleted node not shown



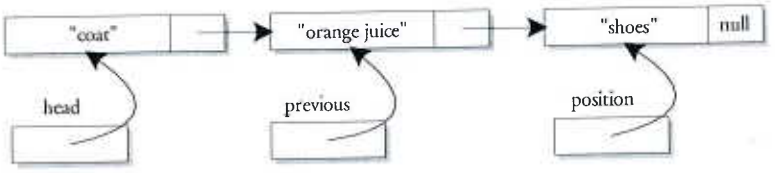
Au/Ed/comp: s/b before start of new section

Yes, ideally

835

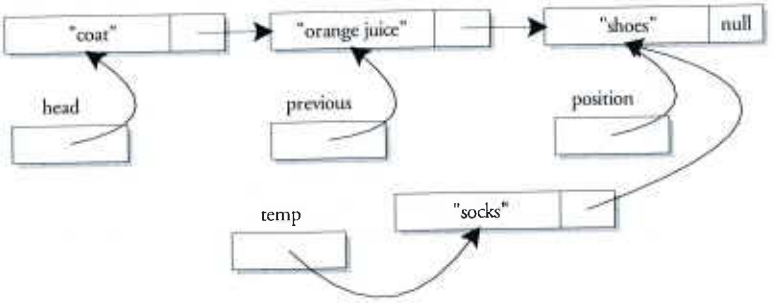
Display 15.20 Adding a Node between Two Nodes

1. Existing list with the iterator positioned at "shoes"



2. Create new Node with "socks" linked to "shoes"

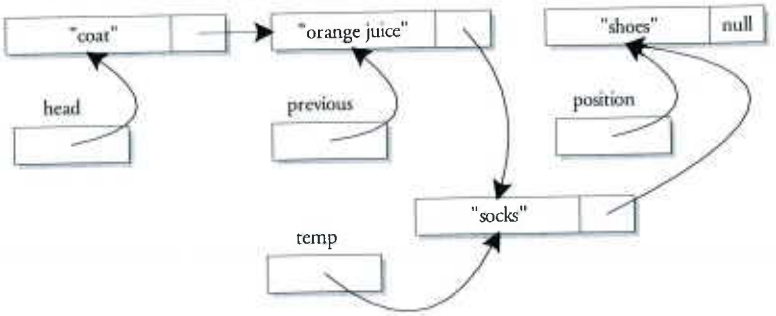
```
temp = new Node (newData, position); // newData is "socks"
```



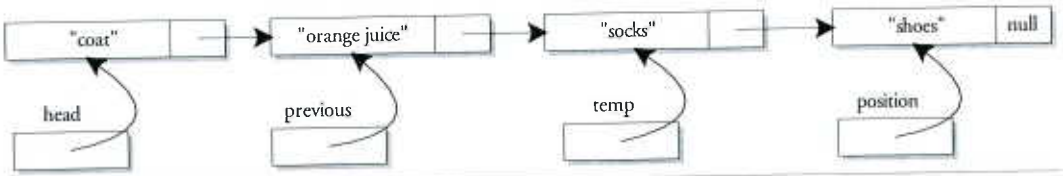
Local variable of type Node

3. Make previous link to the Node temp

```
previous. link = temp;
```



4. Picture redrawn for clarity, but structurally identical to picture 3



To summarize, the following two lines insert a new node with `newData` as its data. The new node is inserted between the nodes named by `previous` and `position`.

```
temp = new Node(newData, position);
previous.link = temp;
```

`previous`, `position`, and `temp` are all variables of type `Node`. (When we use this code, `previous` and `position` will be instance variables of an iterator and `temp` will be a local variable.)

Just like deletion, special cases exist for insertion that must be handled. If the list is empty, then addition is done by adding to the front of the list. If the `position` variable is `null`, then the new node should be added to the end of the list.

Self-Test Exercises

11. Consider a variant of the class in Display 15.17 with no `previous` local variable. In other words, there is no reference kept to the node that links to the current node position. How could we modify the `delete` method to delete the `position` node and still maintain a correct list? The solution is less efficient than the version that uses `previous`.
12. Consider a variant of the class in Display 15.17 with no `previous` local variable. In other words, there is no reference kept to the node that links to the current node `position`. Write a method `addAfterHere(String newData)` that adds a new node after the node in `position`.
13. Complete the definition of the method `changeHere` in the inner class `List2Iterator` in Display 15.17.
14. Given an iterator pointing somewhere in a linked list, does `i.next( )` return the value that `i` is referencing prior to the invocation of `i.next( )` or does it return the value of the next node in the list?

15.4 Variations on a Linked List

I have called this principle, by which each slight variation, if useful, is preserved, by the term Natural Selection.

CHARLES DARWIN, The Origin of Species

In this section, we discuss some variations on linked lists, including the two data structures known as stacks and queues. Stacks and queues need not involve linked lists, but one common way to implement a stack or a queue is to use a linked list.

Doubly Linked List

doubly linked list

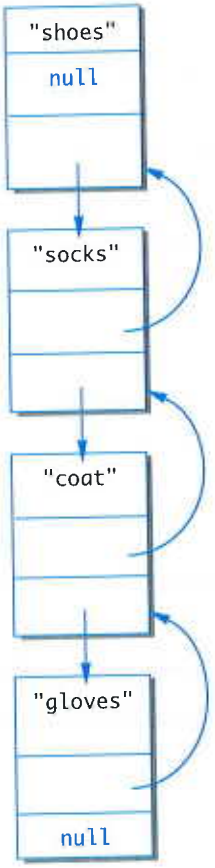
An ordinary linked list allows you to move down the list in only one direction (following the links). A **doubly linked list** has one link that has a reference to the next node and one that has a reference to the previous node. In some cases the link to the previous node can simplify our code. For example, we will no longer need to have a `previous` instance variable to remember the node that links to the current position. Diagrammatically, a doubly linked list looks like the sample list in Display 15.21.

The node class for a doubly linked list can begin as follows:

```
private class TwoWayNode
{
    private String item;
    private TwoWayNode previous;
    private TwoWayNode next;

    ...
```

Display 15.21 A Doubly Linked List



The constructors and some of the methods in the doubly linked list class will require changes (from the singly linked case) in their definitions to accommodate the extra link. The major changes are to the methods that add and delete nodes. To make our code a little cleaner, we can add a new constructor that sets the previous and next nodes:

```
public TwoWayNode(String newItem, TwoWayNode previousNode, TwoWayNode nextNode)
{
    item = newItem;
    next = nextNode;
    previous = previousNode;
}
```

To add a new `TwoWayNode` to the front of the list requires setting links on two nodes instead of one. The general process is shown in Display 15.22. In the `addToStart` method we first create a new `TwoWayNode`. Because the new node will go on the front of the list, we set the previous link to `null` and the `next` link to the current head:

```
TwoWayNode newHead = new TwoWayNode(itemName, null, head);
```

Next we must set the `previous` link on the old head node to reference the new head. We can do this by setting `head.previous = newHead`, but we must take care to ensure that `head` is not `null` (i.e., the list is not empty). Finally, we can set `head` to `newHead`.

```
if (head != null)
{
    head.previous = newHead;
}
head = newHead;
```

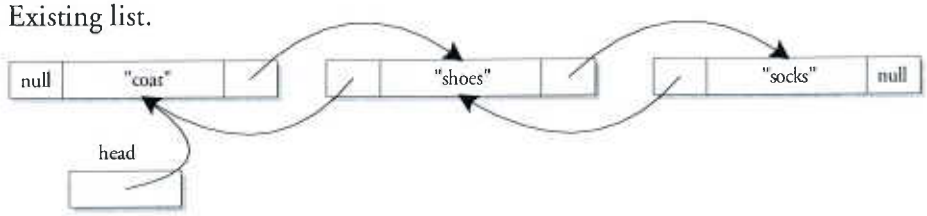
To delete a node from the doubly linked list also requires updating the references on both sides of the node to delete. Thanks to the backward link there is no need for an instance variable to keep track of the previous node in the list, as was required for the singly linked list. The general process of deleting a node referenced by `position` is shown in Display 15.23. Note that some cases must be handled separately, such as deleting a node from the beginning or the end of the list.

The process of inserting a new node into the doubly linked list is shown in Display 15.24. In this case we will insert the new node in front of the iterator referenced by `position`. Note that there are also special cases for the insert routine when inserting to the front or adding to the end of the list. Only the general case of inserting between two existing nodes is shown in Display 15.24.

A complete example of a doubly linked list is shown in Display 15.25. The code in Display 15.25 is modified from the code in Display 15.17. Use of the doubly linked list is virtually identical to use of a singly linked list. Display 15.26 demonstrates addition, deletion, and insertion into the doubly linked list.

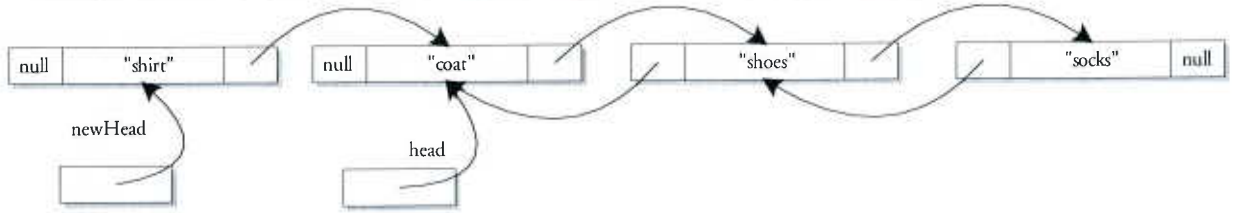
Display 15.22 Adding a Node to the Front of a Doubly Linked List

1. Existing list.



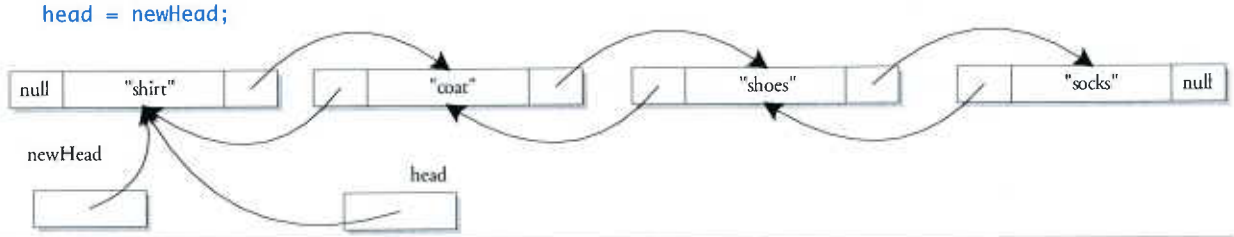
2. Create new TwoWayNode linked to "coat"

```
TwoWayNode newHead = new TwoWayNode (itemName, null, head) // itemName = "shirt"
```



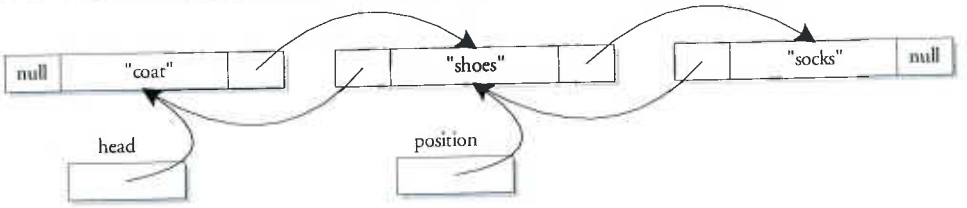
3. Set backward link and set new head

```
head. previous = newHead;
head = newHead;
```



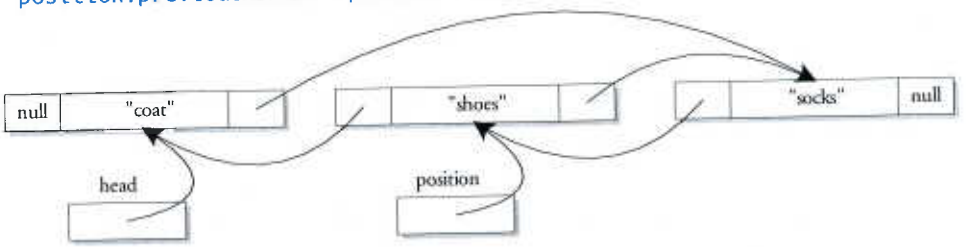
Display 15.23 Deleting a Node from a Doubly Linked List

1. Existing list with an iterator referencing "shoes"



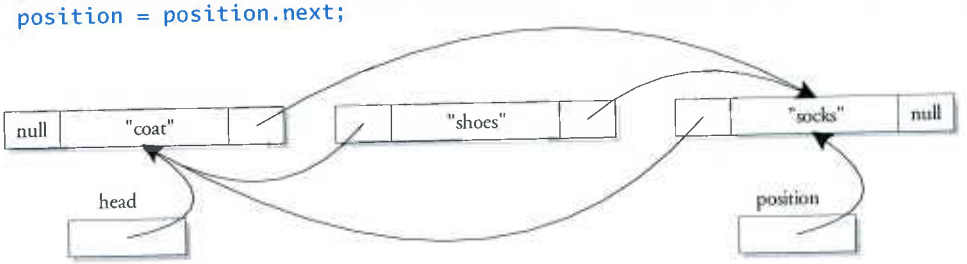
2. Bypass the "shoes" node from the next link of the previous node

```
position.previous.next = position.next;
```

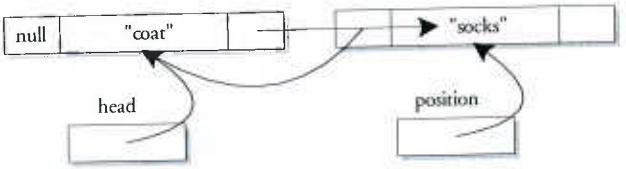


3. Bypass the "shoes" node from the previous link of the next node and move position off the deleted node

```
position. next.previous = position.previous;
position = position.next;
```

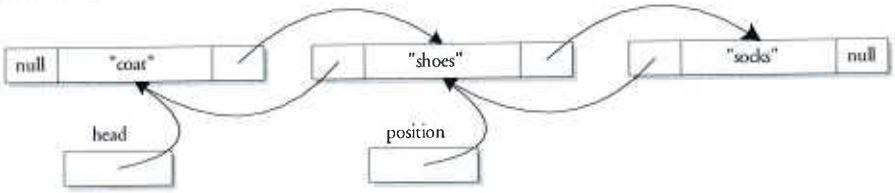


4. Picture redrawn for clarity with the "shoes" node removed since there are no longer references pointing to this node.



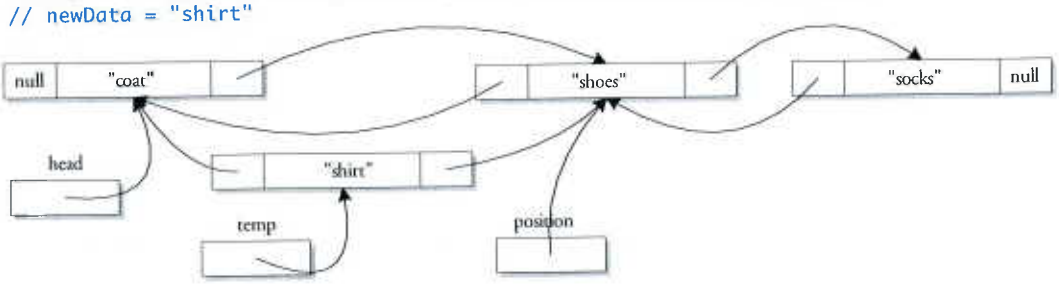
Display 15.24 Inserting a Node into a Doubly Linked List

1. Existing list with an iterator referencing "shoes"



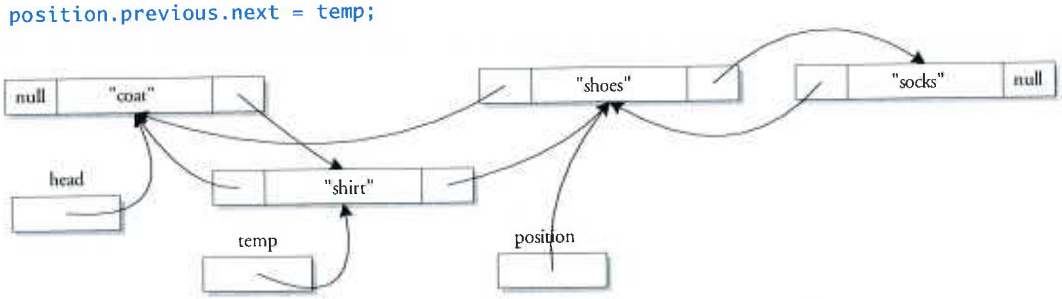
2. Create new TwoWayNode with previous linked to "coat" and next to "shoes"

```
TwoWayNode temp = newTwoWayNode (newData, position.previous, position);
// newData = "shirt"
```



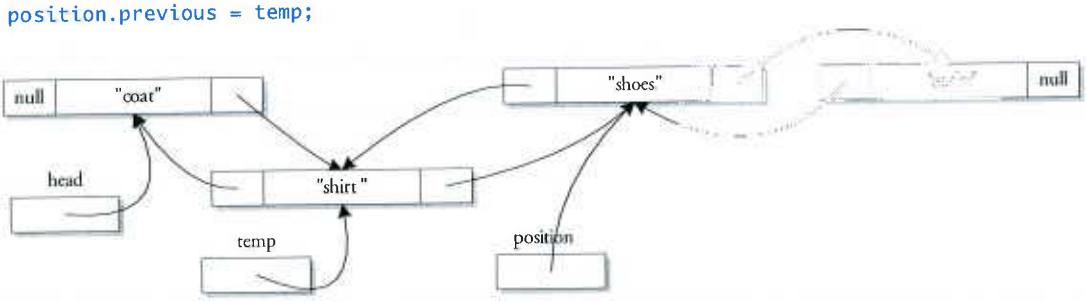
3. Set next link from "coat" to the new node of "shirt"

```
position.previous.next = temp;
```



4. Set previous link from "shoes" to the new node of "shirt"

```
position.previous = temp;
```



Display 15.25 A Doubly Linked List with an Iterator (part 1 of 3)

```
1  import java.util.NoSuchElementException;

2  public class DoublyLinkedList
3  {
4      private class TwoWayNode
5      {
6          private String item;
7          private TwoWayNode previous;
8          private TwoWayNode next;

9          public TwoWayNode( )
10         {
11              item = null;
12              next = null;
13              previous = null;
14         }
15         public TwoWayNode(String newItem, TwoWayNode previousNode, TwoWayNode nextNode)
16         {
17             item = newItem;
18             next = nextNode;
19             previous = previousNode;
20         }
21     }//End of TwoWayNode inner class

22     public class DoublyLinkedIterator
23     {
24         // We do not need a previous node when using a doubly linked list
25         private TwoWayNode position = null;

26         public DoublyLinkedIterator( )
27         {
28             position = head;
29         }
30         public void restart( )
31         {
32             position = head;
33         }
34         public String next( )
35         {
36             if (!hasNext( ))
37                 throw new IllegalStateException( );
38             String toReturn = position.item;
39             position = position.next;
40             return toReturn;
41         }
```

Display 15.25 A Doubly Linked List with an Iterator (part 2 of 3)

```
42              public void insertHere(String newData)
43              {
44                    if (position == null && head != null)
45                    {
46                          // Add to end.  First move a temp
47                          // pointer to the end of the list
48                          TwoWayNode temp = head;
49                          while (temp.next != null)
50                                temp = temp.next;
51                          temp.next = new TwoWayNode(newData, temp, null);
52                    }
53                    else if (head == null || position.previous == null)
54                          // at head of list
55                          DoublyLinkedList.this.addToStart (newData);
56                    else
57                    {
58                          // Insert before the current position
59                          TwoWayNode temp = new TwoWayNode(newData,
                                  position.previous, position);
60                          position.previous.next = temp;
61                          position.previous = temp;
62                    }
63              }

64              public void delete( )
65              {
66                    if (position == null)
67                          throw new IllegalStateException( );
68                    else if (position.previous == null)
69                    { // Deleting first node
70                          head = head.next;
71                          position = head;
72                    }
73                    else if (position.next == null)
74                    { // Deleting last node
75                          position.previous.next = null;
76                          position = null;
77                    }
78                    else
79                    {
80                          position.previous.next = position.next;
81                          position.next.previous = position.previous;
82                          position = position.next;
83                    }
84              }
85        }     // DoublyLinkedIterator
```

(continued)

Display 15.25 A Doubly Linked List with an Iterator (part 3 of 3)

```
86      private TwoWayNode head;

87      public DoublyLinkedIterator iterator( )
88      {
89          return new DoublyLinkedIterator( );
90      }

91      public DoublyLinkedList( )
92      {
93          head = null;
94      }

95      /**
96       The added node will be the first node in the list.
97      */
98      public void addToStart(String itemName)
99      {
100         TwoWayNode newHead = new TwoWayNode(itemName, null, head);
101         if (head != null)
102         {
103               head.previous = newHead;
104         }
105         head = newHead;
106     }
    <The methods hasNext, peek, clear, and isEmpty are identical
     to those in Display 15.17. Other methods would also normally
     be defined here, such as deleteHeadNode, size, outputList,
     equals, clone, find, or contains. They have been left off to
     simplify the example.>
107 }     // DoublyLinkedList
```

Display 15.26 Using a Doubly Linked List with an Iterator (part 1 of 2)

```
1   public class DoublyLinkedListDemo
2   {
3         public static void main(String[] args)
4         {
5               DoublyLinkedList list = new DoublyLinkedList( );
6               DoublyLinkedList.DoublyLinkedIterator i = list.iterator( );

7               list.addToStart("shoes");
8               list.addToStart("orange juice");
9               list.addToStart("coat");
```

Display 15.26 Using a Doubly Linked List with an Iterator (part 2 of 2)

```
10              System.out.println("List contains:");
11              i.restart( );
12              while (i.hasNext( ))
13                    System.out.println(i.next( ));
14              System.out.println( );

15              i.restart( );
16              i.next( );
17              i.next( );
18              System.out.println("Delete " + i.peek( ));
19              i.delete( );

20              System.out.println("List now contains:");
21              i.restart( );
22              while (i.hasNext( ))
23                    System.out.println(i.next( ));
24              System.out.println( );

25              i.restart( );
26              i.next( );
27              System.out.println("Inserting socks before " + i.peek( ));
28              i.insertHere("socks");

29              i.restart( );
30              System.out.println("List now contains:");
31              while (i.hasNext( ))
32                    System.out.println(i.next( ));
33              System.out.println( );
34        }
35  }
```

Sample Dialogue

```
List contains:
coat
orange juice
shoes

Delete shoes
List now contains:
Coat
Orange juice

Inserting socks before orange juice
List now contains:
coat
socks
orange juice
```

Self-Test Exercises

15. What operations are easier to implement with a doubly linked list compared with a singly linked list? What operations are more difficult?

16. If the `addToStart` method from Display 15.25 were removed, how could we still add a new node to the head of the list?

The Stack Data Structure

stack

A **stack** is not necessarily a linked data structure, but it can be implemented as a linked list. A stack is a data structure that removes items in the reverse of the order in which they were inserted. So if you insert `"one"`, then `"two"`, and then `"three"` into a stack and then remove them, they will come out in the order `"three"`, then `"two"`, and finally `"one"`. Stacks are discussed in more detail in Chapter 11. A linked list that inserts and deletes only at the head of the list (such as the one in Display 15.3 or in Display 15.8) is, in fact, a stack.

push and pop

You can imagine the stack data structure like a stack of trays in a cafeteria. You can **push** a new tray on top of the stack to make a taller stack. Alternately, you can **pop** the topmost tray off the stack until there are no more trays to remove. A definition of a `Stack` class is shown in Display 15.27 that is based on the linked list from Display 15.3. A short demonstration program is shown in Display 15.28. The `addToStart` method has been renamed to `push` to use stack terminology. Similarly, the `deleteHeadNode` method has been renamed to `pop` and returns the `String` from the top of the stack. Although not shown here to keep the definition simple, it would be appropriate to add other methods such as `peek`, `clone`, or `equals` or to convert the class to use a generic data type.

Stacks

A *stack* is a last-in/first-out data structure; that is, the data items are retrieved in the opposite order to which they were placed in the stack.

Display 15.27 A `Stack` Class (part 1 of 2)

```
1  import java.util.NoSuchElementException;

2  public class Stack
3  {
4        private class Node
5        {
6              private String item;
7              private Node link;
```

Display 15.27 A Stack Class (part 2 of 2)

```
8               public Node( )
9               {
10                    item = null;
11                    link = null;
12              }
13              public Node(String newItem, Node linkValue)
14              {
15                    item = newItem;
16                    link = linkValue;
17              }
18        }//End of Node inner class

19        private Node head;

20        public Stack( )
21        {
22              head = null;
23        }

24        /**
25              This method replaces addToStart
26        */
27        public void push(String itemName)
28        {
29              head = new Node(itemName, head);
30        }
31        /**
32              This method replaces deleteHeadNode and
33              also returns the value popped from the list
34        */
35        public String pop( )
36        {
37              if (head == null)
38                    throw  new IllegalStateException( );
39              else
40              {
41                    String returnItem = head.item;
42                    head = head.link;
43                    return returnItem;
44              }
45        }
46        public boolean isEmpty( )
47        {
48              return (head == null);
49        }
50  }
```

Display 15.28 Stack Demonstration Program

```
1  public class StackExample
2  {
3        public static void main(String[] args)
4        {
5              Stack stack = new Stack( );
6              stack.push("Billy Rubin");
7              stack.push("Lou Pole");
8              stack.push("Polly Ester");
9              while (!stack.isEmpty( ))
10             {
11                   String s = stack.pop( );
12                   System.out.println(s);
13             }
14       }
15 }
```

Items come out of the stack in the reverse order that they were added.

Sample Dialogue

```
Polly Ester
Lou Pole
Billy Rubin
```

Self-Test Exercise

17. Display 15.27 does not contain a `peek( )` method. Normally this method would return the data on the top of the stack without popping it off. How could a user of the `Stack` class get the same functionally as `peek( )` even though it is not defined?

The Queue Data Structure

queue

A stack is a last-in/first-out data structure. Another common data structure is a **queue**, which handles data in a first-in/first-out fashion. A queue is like a line at the bank. Customers add themselves to the back of the line and are served from the front of the line. A queue can be implemented with a linked list. However, a queue needs a pointer at both the head of the list and at the **tail** (that is, the other end) of the linked list, because action takes place in both locations. It is easier to remove a node from the head of a linked list than from the tail of the linked list. So, a simple implementation will remove nodes from the head of the list (which we will now call the **front** of the list) and we will add nodes to the tail end of the list, which we will now call the **back** of the list (or the back of the queue).

tail

front

back

The definition of a simple Queue class that is based on a linked list is given in Display 15.29. A short demonstration program is given in Display 15.30. We have not made our queue a generic queue to keep the definition simple, but it would be routine to replace the data type String with a type parameter.

Queue

A *queue* is a first-in/first-out data structure; that is, the data items are removed from the queue in the same order that they were added to the queue.

Self-Test Exercise

18. Complete the definition of the method addToBack in Display 15.29.

Display 15.29 A Queue Class (part 1 of 2)

```
1  public class Queue
2  {
3      private class Node
4      {
5          private String item;
6          private Node link;

7          public Node( )
8          {
9               item = null;
10              link = null;
11         }

12         public Node(String newItem, Node linkValue)
13         {
14             item = newItem;
15             link = linkValue;
16         }
17     }//End of Node inner class

18     private Node front;
19     private Node back;

20     public Queue( )
21     {
22         front = null;
23         back = null;
24     }
```

(continued)

Display 15.29 A Queue Class (part 2 of 2)

```
25      /**
26       Adds a String to the back of the queue.
27      */
28      public void addToBack(String itemName)
        <The definition of this method is Self-Test Exercise 18.>

29      public boolean isEmpty( )
30      {
31          return (front == null);
32      }

33      public void clear( )
34      {
35          front = null;
36          back = null;
37      }
38      /**
39       Returns the String in the front of the queue.
40       Returns null if queue is empty.
41      */
42      public String whoIsNext( )
43      {
44          if (front == null)
45              return null;
46          else
47              return front.item;
48      }
49
50      /**
51       Removes a String from the front of the queue.
52       Returns false if the list is empty.
53      */
54      public boolean removeFront( )
55      {
56          if (front != null)
57          {
58              front = front.link;
59              return true;
60          }
61          else
62              return false;
63      }
64  }
```

Display 15.30 Demonstration of the Queue Class

```
1  public class QueueDemo
2  {
3      public static void main(String[] args)
4      {
5          Queue q = new Queue( );

6          q.addToBack("Tom");
7          q.addToBack("Dick");
8          q.addToBack("Harriet");
9          while(!q.isEmpty( ))
10         {
11             System.out.println(q.whoIsNext( ));
12             q.removeFront( );
13         }
14         System.out.println("The queue is empty.");
15     }
16 }
```

Items come out of the queue in the same order that they went into the queue.

Sample Dialogue

```
Tom
Dick
Harriet
The queue is empty.
```

Items come out of the queue in the same order that they went into the queue.

In order to have some terminology to discuss the efficiency of our Queue class and linked list algorithms, we first present some background on how the efficiency of algorithms is usually measured.

Running Times and Big-*O* Notation

If you ask a programmer how fast his or her program is, you might expect an answer like "two seconds." However, the speed of a program cannot be given by a single number. A program will typically take a longer amount of time on larger inputs than it will on smaller inputs. You would expect that a program for sorting numbers would take less time to sort ten numbers than it would to sort one thousand numbers. Perhaps it takes two seconds to sort ten numbers, but ten seconds to sort one thousand numbers. How, then, should the programmer answer the question "How fast is your program?" The programmer would have to give a table of values showing how long the program took for different sizes of input. For example, the table might be as shown in Display 15.31. This table does not give a single time, but instead gives different times for a variety of different input sizes.

Display 15.31 Some Values of a Running Time Function

Input Size	Running Time
10 numbers	2 seconds
100 numbers	2.1 seconds
1,000 numbers	10 seconds
10,000 numbers	2.5 minutes

function

The table is a description of what is called a **function** in mathematics. Just as a (non-void) Java method takes an argument and returns a value, so too does this function take an argument, which is an input size, and returns a number, which is the time the program takes on an input of that size. If we call this function T, then $T(10)$ is 2 seconds, $T(100)$ is 2.1 seconds, $T(1{,}000)$ is 10 seconds, and $T(10{,}000)$ is 2.5 minutes. The table is just a sample of some of the values of this function T. The program will take some amount of time on inputs of every size. So although they are not shown in the table, there are also values for $T(1)$, $T(2)$, . . . , $T(101)$, $T(102)$, and so forth. For any positive integer N, $T(N)$ is the amount of time it takes for the program to sort N numbers. The function T is called the **running time** of the program.

running time

So far we have been assuming that this sorting program will take the same amount of time on any list of N numbers. That need not be true. Perhaps it takes much less time if the list is already sorted or almost sorted. In that case, $T(N)$ is defined to be the time taken by the "hardest" list, that is, the time taken on that list of N numbers that makes the program run the longest. This is called the **worst-case running time**. In this chapter we will always mean worst-case running time when we give a running time for an algorithm or for some code.

worst-case running time

The time taken by a program or algorithm is often given by a formula, such as $4N + 3$, $5N + 4$, or N^2. If the running time $T(N)$ is $5N + 5$, then on inputs of size N the program will run for $5N + 5$ time units.

Below is some code to search an array `a` with N elements to determine whether a particular value `target` is in the array:

```
int i = 0;
boolean found = false;
while (( i < N) && !(found))
{
    if (a[i] == target)
        found = true;
    else
        i++;
}
```

We want to compute some estimate of how long it will take a computer to execute this code. We would like an estimate that does not depend on which computer we use, either because we do not know which computer we will use or because we might use several different computers to run the program at different times.

One possibility is to count the number of "steps," but it is not easy to decide what a step is. In this situation the normal thing to do is count the number of **operations**. The term *operations* is almost as vague as the term *step*, but there is at least some agreement in practice about what qualifies as an operation. Let us say that, for this Java code, each application of any of the following will count as an operation: =, <, &&, !, [], ==, and ++. The computer must do other things besides carry out these operations, but these seem to be the main things that it is doing, and we will assume that they account for the bulk of the time needed to run this code. In fact, our analysis of time will assume that everything else takes no time at all and that the total time for our program to run is equal to the time needed to perform these operations. Although this is an idealization that clearly is not completely true, it turns out that this simplifying assumption works well in practice, and so it is often made when analyzing a program or algorithm.

Even with our simplifying assumption, we still must consider two cases: Either the value `target` is in the array or it is not. Let us first consider the case when `target` is not in the array. The number of operations performed will depend on the number of array elements searched. The operation = is performed two times before the loop is executed. Since we are assuming that `target` is not in the array, the loop will be executed N times, one for each element of the array. Each time the loop is executed, the following operations are performed: <, &&, !, [], ==, and ++. This adds five operations for each of N loop iterations. Finally, after N iterations, the Boolean expression is again checked and found to be `false`. This adds a final three operations (<, &&, !).[3] If we tally all these operations, we get a total of $6N + 5$ operations when the target is not in the array. We will leave it as an exercise for the reader to confirm that if the target is in the array, then the number of operations will be $6N + 5$ *or less*. Thus, the worst-case running time is $T(N) = 6N + 5$ operations for any array of N elements and any value of `target`.

We just determined that the worst-case running time for our search code is $6N + 5$ operations. But an operation is not a traditional unit of time, like a nanosecond, second, or minute. If we want to know how long the algorithm will take on some particular computer, we must know how long it takes that computer to perform one operation. If an operation can be performed in one nanosecond, then the time will be $6N + 5$ nanoseconds. If an operation can be performed in one second, the time will be $6N + 5$ seconds. If we use a slow computer that takes ten seconds to perform an operation, the time will be $60N + 50$ seconds. In general, if it takes the computer c nanoseconds to perform one operation, then the actual running time will be approximately

[3] Because of short-circuit evaluation, `!(found)` is not evaluated, so we actually get two, not three, operations. However, the important thing is to obtain a good upper bound. If we add in one extra operation, that is not significant.

$c(6N+5)$ nanoseconds. (We said *approximately* because we are making some simplifying assumptions and therefore the result may not be the absolutely exact running time.) This means that our running time of $6N+5$ is a very crude estimate. To get the running time expressed in nanoseconds, you must multiply by some constant that depends on the particular computer you are using. Our estimate of $6N+5$ is only accurate to within a constant multiple.

big-O notation

Estimates on running time, such as the one we just went through, are normally expressed in something called **big-O notation**. (The O is the letter "Oh," not the digit zero.) Suppose we estimate the running time to be, say, $6N+5$ operations, and suppose we know that no matter what the exact running time of each different operation may turn out to be, there will always be some constant factor c such that the real running time is less than or equal to $c(6N+5)$. Under these circumstances, we say that the code (or program or algorithm) runs in time $O(6N+5)$. This is usually read as "big-O of $6N+5$." We need not know what the constant c will be. In fact, it will undoubtedly be different for different computers, but we must know that there is one such c for any reasonable computer system. If the computer is very fast, the c might be less than 1—say, 0.001. If the computer is very slow, the c might be very large—say, 1,000. Moreover, since changing the units (say from nanosecond to second) only involves a constant multiple, there is no need to give any units of time.

Be sure to notice that a big-O estimate is an upper-bound estimate. We always approximate by taking numbers on the high side rather than the low side of the true count. Also notice that when performing a big-O estimate, we need not determine an exact count of the number of operations performed. We only need an estimate that is correct up to a constant multiple. If our estimate is twice as large as the true number, that is good enough.

An order-of-magnitude estimate, such as the previous $6N+5$, contains a parameter for the size of the task solved by the algorithm (or program or piece of code). In our sample case, this parameter N was the number of array elements to be searched. Not surprisingly, it takes longer to search a larger number of array elements than it does to search a smaller number of array elements. Big-O running-time estimates are always expressed as a function of the size of the problem. In this chapter, all our algorithms will involve a range of values in some container. In all cases N will be the number of elements in that range.

The following is an alternative, pragmatic way to think about big-O estimates:

Only look at the term with the highest exponent and do not pay attention to constant multiples.

For example, all of the following are $O(N^2)$:

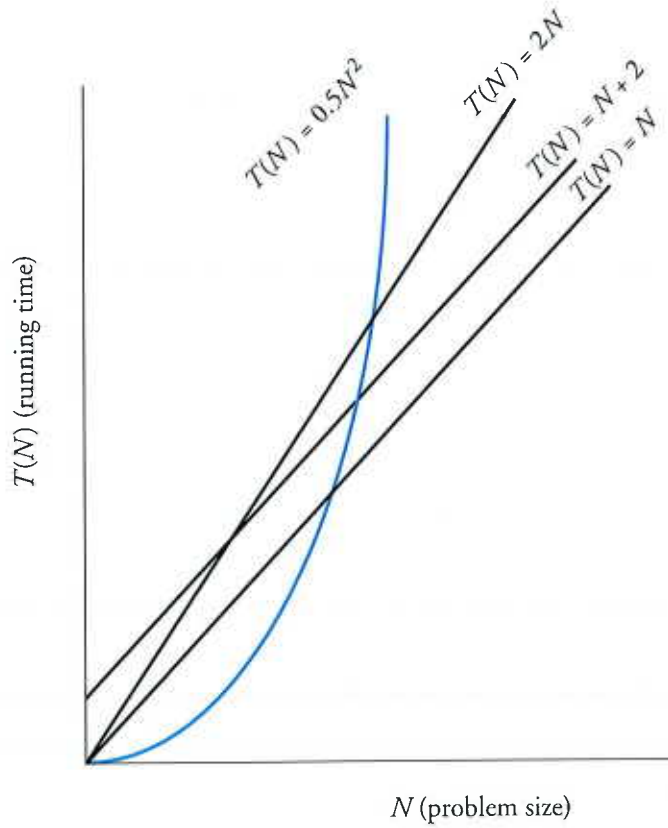
$$N^2+2N+1, \quad 3N^2+7, \quad 100N^2+N$$

All of the following are $O(N^3)$:

$$N^3+5N^2+N+1, \quad 8N^3+7, \quad 100N^3+4N+1$$

These big-*O* running-time estimates are admittedly crude, but they do contain some information. They will not distinguish between a running time of $5N + 5$ and a running time of $100N$, but they do let us distinguish between some running times and so determine that some algorithms are faster than others. Look at the graphs in Display 15.32 and notice that all the graphs for functions that are $O(N)$ eventually fall below the graph for the function $0.5N^2$. The result is inevitable: An $O(N)$ algorithm will always run faster than any $O(N^2)$ algorithm, provided we use large enough values of N. Although an $O(N^2)$ algorithm could be faster than an $O(N)$ algorithm for the problem size you are handling, programmers have found that, in practice, $O(N)$ algorithms perform better than $O(N)$ algorithms for most practical applications that are intuitively "large." Similar remarks apply to any other two different big-*O* running times.

Display 15.32 Comparison of Running Times



linear running time
quadratic running time

Some terminology will help with our descriptions of generic algorithm running times. **Linear running time** means a running time of $T(N) = aN + b$. A linear running time is always an $O(N)$ running time. **Quadratic running time** means a running time with a highest term of N^2. A quadratic running time is always an $O(N^2)$ running time. We will also occasionally have logarithms in running-time formulas. Those normally are given without any base, since changing the base is just a constant multiple. If you see log N, think log base 2 of N, but it would not be wrong to think log base 10 of N. Logarithms are very slow growing functions. So, an $O(\log N)$ running time is very fast.

In many cases, our running-time estimates will be better than big-O estimates. In particular, when we specify a linear running time, that is a tight upper bound and you can think of the running time as being exactly $T(N) = cN$, although the c is still not specified.

Self-Test Exercises

19. Show that a running time $T(N) = aN + b$ is an $O(N)$ running time. (*Hint:* The only issue is the plus b. Assume N is always at least 1.)

20. Show that for any two bases a and b for logarithms, if a and b are both greater than 1, then there is a constant c such that $\log_a N \leq c (\log_b N)$. Thus, there is no need to specify a base in $O(\log N)$. That is, $O(\log_a N)$ and $O(\log_b N)$ mean the same thing.

Efficiency of Linked Lists

Now that we know about big-O notation, we can express the efficiency of various methods for our linked data structures. As an example of analyzing the run-time efficiency of an algorithm, consider the `find` method for the linked list class in Display 15.3. This method starts at the head of the list and sequentially iterates through each node to see whether it matches the target. If the linked list contains many nodes, then we might get lucky if the target is found at the head of the list. In this case the computer only had to execute one step: Check the head of the list for the target. In the worst case the compute might have to search through all n nodes before finding (or not finding) the target. In this case the computer had to execute n steps. The worst case will obviously take longer to execute than the best case. On average we might expect to search through about half of the list before finding the target. This would require $n/2$ steps. In our big-O notation, the find operation is $O(n)$. However, the `addToStart` method requires only linking a new node to the head of the list. This runs in $O(1)$ steps (that is, a constant upper bound on the running time that is independent of the size of the input).

Next we shall briefly examine more elaborate data structures that are capable of performing find operations in fewer steps. However, a detailed treatment of these more advanced data structures is beyond the scope of this chapter. The goal of this chapter is to teach you the basic techniques for constructing and manipulating data structures based on nodes and links (that is, nodes and references). The linked lists served as good examples for our discussion.

15.5 Hash Tables with Chaining

Seek, and ye shall find.

MATTHEW 7:7

hash table
hash map

A **hash table** or **hash map** is a data structure that efficiently stores and retrieves data from memory. There are many ways to construct a hash table; in this section we will use an array in combination with singly linked lists. In the previous section we saw that a linked list generally requires linear, or $O(n)$, steps to determine if a target is in the list. In contrast, a hash table has the potential to execute a fixed number of steps to look up a target, regardless of the size of n. We saw that a constant-time lookup is written $O(1)$. However, the hash table we will present may still require n steps, but such a case is unlikely.

An object is stored in a hash table by associating it with a *key*. Given the key, we can retrieve the object. Ideally, the key is unique to each object. If the object has no intrinsically unique key, then we can use a **hash function** to compute one. In most cases the hash function computes a number.

hash function

For example, let's use a hash table to store a dictionary of words. Such a hash table might be useful to make a spell checker—words missing from the hash table might not be spelled correctly. We will construct the hash table with a fixed array in which each array element references a linked list. The key computed by the hash function will map to the index of the array. The actual data will be stored in a linked list at the hash value's index. Display 15.33 illustrates the idea with a fixed array of ten entries. Initially each entry of the array `hashArray` contains a reference to an empty singly linked list. First we add the word `"cat"`, which has been assigned the key or hash value of 2 (we'll show how this was computed shortly). Next we add `"dog"` and `"bird"`, which are assigned hash values of 4 and 7, respectively. Each of these strings is inserted as the head of the linked list using the hash value as the index in the array. Finally, we add `"turtle"`, which also has a hash of 2. Since `"cat"` is already stored at index 2, we now have a **collision**. Both `"turtle"` and `"cat"` map to the same index in the array. When this occurs in a hash table with **chaining**, we simply insert the new node onto the existing linked list. In our example there are now two nodes at index 2: `"turtle"` and `"cat"`.

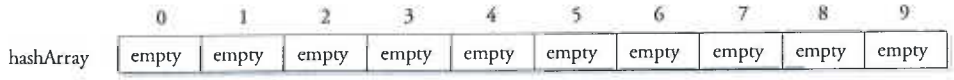
collision
chaining

To retrieve a value from the hash table, we first compute the hash value of the target. Next we search the linked list that is stored at `hashArray[hashValue]` for the target, using an iterator to sequentially search the linked list. If the target is not found in this linked list, then the target is not stored in the hash table. If the size of the linked list is small then the retrieval process will be quick.

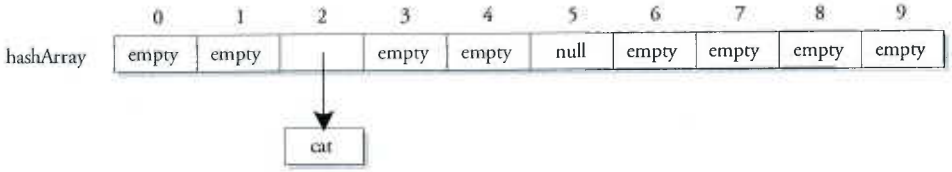
Display 15.33 Constructing a Hash Table

1. Existing hash table initialized with ten empty linked lists

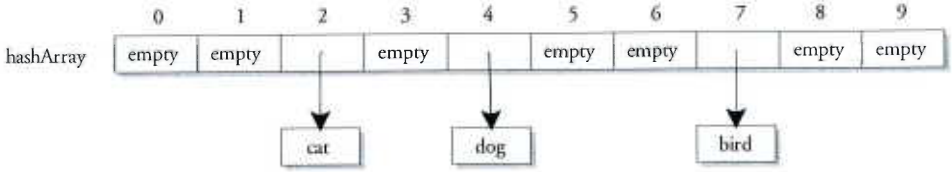
```
hashArray = new LinkedList 3[SIZE]; // SIZE = 10
```



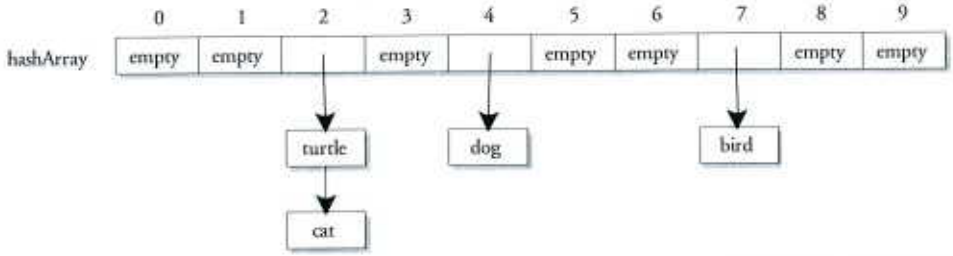
2. After adding "cat" with hash of 2



3. After adding "dog" with hash of 4 and "bird" with hash of 7



4. After adding "turtle" with hash of 2 – collision and chained to linked list with "cat"



A Hash Function for Strings

A simple way to compute a numeric hash value for a string is to sum the ASCII value of every character in the string and then compute the modulus of the sum using the size of the fixed array. A subset of ASCII codes is given in Appendix 3. Code to compute the hash value is shown below:

```
private int computeHash(String s)
{
   int hash = 0;
   for (int i = 0; i < s.length( ); i++)
   {
       hash += s.charAt(i);
   }
   return hash % SIZE;// SIZE = 10 in example
}
```

For example, the ASCII codes for the string "dog" are as follows:

```
d    -> 100
o    -> 111
g    -> 103
```

The hash function is computed as follows:

```
Sum              =   100 + 111 + 103          = 314
Hash = Sum % 10  =   314 % 10                 = 4
```

In this example we first compute an unbounded value, the sum of the ASCII values in the string. However, the array was defined to only hold a finite number of elements. To scale the sum to the size of the array, we compute the modulus of the sum with respect to the size of the array, which is 10 in the example. In practice the size of the array is generally a prime number larger than the number of items that will be put into the hash table.[4] The computed hash value of 4 serves like a fingerprint for the string "dog". However, different strings may map to the same value. We can verify that "cat" maps to (99 + 97 + 116) % 10 = 2 and also that "turtle" maps to (116 + 117 + 114 + 116 + 108 + 101) % 10 = 2.

A complete code listing for a hash table class is given in Display 15.34, and a demonstration is provided in Display 15.35. The hash table definition in Display 15.34 uses an array in which each element is a `LinkedList2` class defined in Display 15.7.

Display 15.34 A Hash Table Class (part 1 of 2)

```
1  public class HashTable
2  {
3        // Uses the generic LinkedList2 class from Display 15.7
4        private LinkedList2[] hashArray;
5        private static final int SIZE = 10;

6        public HashTable( )
7        {
8             hashArray = new LinkedList2[SIZE];
```

(continued)

[4] A prime number avoids common divisors after modulus that can lead to collisions.

Display 15.34 A Hash Table Class (part 2 of 2)

```
9               for (int i=0; i < SIZE; i++)
10                    hashArray[i] = new LinkedList2( );
11        }

12        private int computeHash(String s)
13        {
14              int hash = 0;
15              for (int i = 0; i < s.length( ); i++)
16              {
17                    hash += s.charAt(i);
18              }
19              return hash % SIZE;
20        }

21        /**
22         Returns true if the target is in the hash table,
23         false if it is not.
24        */
25        public boolean containsString(String target)
26        {
27              int hash = computeHash(target);
28              LinkedList2 list = hashArray[hash];
29              if (list.contains(target))
30                    return true;
31              return false;
32        }
33        /**
34         Stores or puts string s into the hash table
35        */
36        public void put(String s)
37        {
38              int hash = computeHash(s);// Get hash value
39              LinkedList2 list = hashArray[hash];
40              if (!list.contains(s))
41              {
42                    // Only add the target if it's not already
43                    // on the list.
44                    hashArray[hash].addToStart(s);
45              }
46        }
47  } // End HashTable class
```

Display 15.35 Hash Table Demonstration

```
1  public class HashTableDemo
2  {
3        public static void main(String[] args)
4        {
5              HashTable h = new HashTable( );

6              System.out.println("Adding dog, cat, turtle, bird");
7              h.put("dog");
8              h.put("cat");
9              h.put("turtle");
10             h.put("bird");
11             System.out.println("Contains dog? " +
12                    h.containsString("dog"));
13             System.out.println("Contains cat? " +
14                   h.containsString("cat"));
15             System.out.println("Contains turtle? " +
16                   h.containsString("turtle"));
17             System.out.println("Contains bird? " +
18                   h.containsString("bird"));

19             System.out.println("Contains fish? " +
20                   h.containsString("fish"));
21             System.out.println("Contains cow? " +
22                   h.containsString("cow"));
23       }
24 }
```

Sample Dialogue

```
Adding dog, cat, turtle, bird
Contains dog? true
Contains cat? true
Contains turtle? true
Contains bird? true
Contains fish? false
Contains cow? False
```

Efficiency of Hash Tables

The efficiency of our hash table depends on several factors. First, let's examine some extreme cases. The worst-case run-time performance occurs if every item inserted into the table has the same hash key. Everything will then be stored in a single linked list. With n items, the find operation will require $O(n)$ steps. Fortunately, if the items that we insert are somewhat random, the probability that all of them will hash to the same key is highly unlikely. In contrast, the best-case run-time performance occurs if every item inserted into the table has a different hash key. This means that there will be no collisions, so the find operation will require constant, or $O(1)$, steps because the target will always be the first node in the linked list.

We can decrease the chance of collisions by using a better hash function. For example, the simple hash function that sums each letter of a string ignores the ordering of the letters. The words `"rat"` and `"tar"` would hash to the same value. A better hash function for a string `s` is to multiply each letter by an increasing weight depending upon the position in the word. For example:

```
int hash = 0;
for (int i = 0; i < s.length( ); i++)
{
    hash = 31 * hash + s.charAt(i);
}
```

Another way to decrease the chance of collisions is by making the hash table bigger. For example, if the hash table array stored 10,000 entries but we are only inserting 1,000 items, then the probability of a collision is much smaller than if the hash table array stored only 1,000 entries. However, a drawback to creating an extremely large hash table array is wasted memory. If only 1,000 items are inserted into the 10,000-entry hash table then at least 9,000 memory locations will go unused. This illustrates the **time-space tradeoff**. It is usually possible to increase run-time performance at the expense of memory space, and vice versa.

time-space tradeoff

Self-Test Exercises

21. Suppose that every student in your university is assigned a unique nine-digit ID number. You would like to create a hash table that indexes ID numbers to an object representing a student. The hash table has a size of N, where N has less than nine digits. Describe a simple hash function that you can use to map from ID number to a hash index.

22. Write an `outputHashTable( )` method for the `HashTable` class that outputs every item stored in the hash table.

15.6 Sets

There are two classes in good society in England. The equestrian classes and the neurotic classes.

GEORGE BERNARD SHAW, *Heartbreak House*

A set is a collection of elements in which order and multiplicity are ignored. Many problems in computer science can be solved with the aid of a set data structure. A variation on linked lists is a straightforward way to implement a set. In this implementation the items in each set are stored using a singly linked list. The `data` variable contains a reference to an object we wish to store in the set, whereas the `link` variable refers to the next `Node<T>` in the list (which in turn contains a reference to the next object to store in the set). The node class for a generic set of objects can begin as follows:

```
private class Node<T>
{
    private T data;
    private Node<T> link;

    ...
```

A complete listing is provided in Display 15.37. The `Node` class is a private inner class, similar to how we constructed the generic `LinkedList3<T>` class in Display 15.8. In fact, the set operations of `add`, `contains`, `output`, `clear`, `size`, and `isEmpty` are virtually identical to those from Display 15.8. The `add` method (which was `addToStart`) has been slightly changed to prevent duplicate items from being added into the set. Display 15.36 illustrates two sample sets stored using this data structure. The set `round` contains `"peas"`, `"ball"`, and `"pie"`, whereas the set `green` contains `"peas"` and `"grass"`. Since the linked list is storing a reference to each object in the set, it is possible to place an item in multiple sets by referencing it from multiple linked lists. In Display 15.36, `"peas"` is in both sets since it is round and green.

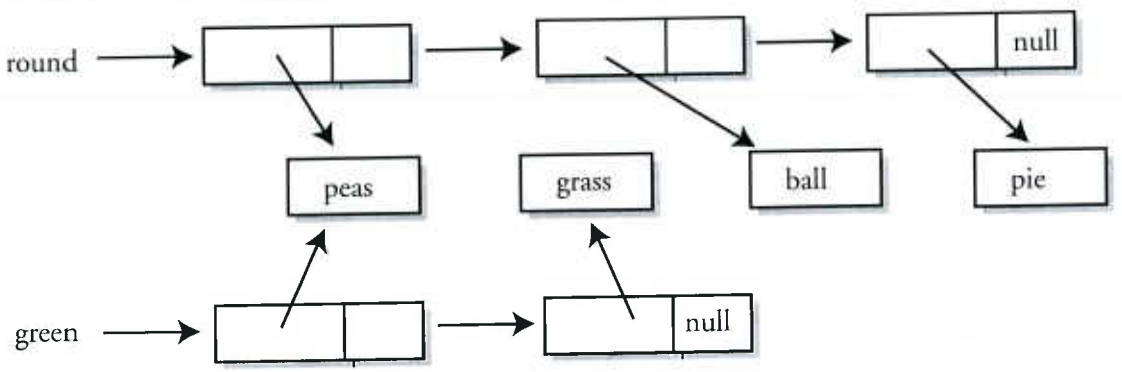
Fundamental Set Operations

The fundamental operations that our set class should support are as follows:

- Add Element. Add a new item into a set.
- Contains. Determine if a target item is a member of the set.
- Union. Return a set that is the union of two sets.
- Intersection. Return a set that is the intersection of two sets.

We should also make an iterator so that every element can be retrieved from a set. This is left as a programming project for the reader. Other useful set operations include methods to retrieve the cardinality of the set and to remove items from the set.

Display 15.36 Sets Using Linked Lists



Code to implement sets is provided in Display 15.37. The `add` method is similar to adding a node to the front of a linked list. The `head` variable always references the first node in the list. The `contains` method is identical to the `find` method for a singly linked list. We simply loop through every item in the list looking for the target.

The `union` method combines the elements in the calling object's set with the elements from the set of the input argument, `otherSet`. To union these sets we first create a new empty `Set<T>` object. Next, we iterate through both the calling object's set and `otherSet`'s set. All elements are added (which creates new references to the items in the set) to the new set. The `add` method enforces uniqueness, so we don't have to check for duplicate elements in the `union` method.

The `intersection` method is similar to the `union` method in that it also creates a new empty `Set<T>` object. In this case we populate the set with items that are common to both the calling object's set and `otherSet`'s set. This is accomplished by iterating through every item in the calling object's set. For each item, we invoke the `contains` method for `otherSet`. If `contains` returns `true`, then the item is in both sets and can be added to the new set.

A short demonstration program is shown in Display 15.38.

Display 15.37 `Set<T>` Class (part 1 of 4)

```
1  // Uses a linked list as the internal data structure
2  // to store items in a set.
3  public class Set<T>
4  {
5      private class Node<T>
6      {
7          private T data;
8          private Node<T> link;
```

Display 15.37 Set<T> Class (part 2 of 4)

```
 9          public Node( )
10          {
11               data = null;
12               link = null;
13          }
14          public Node(T newData, Node<T> linkValue)
15          {
16              data = newData;
17              link = linkValue;
18          }
19       }//End of Node<T> inner class

20       private Node<T> head;

21         public Set( )
22         {
23               head = null;
24         }
25         /**
26          Add a new item to the set. If the item
27          is already in the set, false is returned;
28          otherwise, true is returned.
29         */
30         public boolean add(T newItem)
31         {
32               if (!contains(newItem))
33               {
34                     head = new Node<T>(newItem, head);
35                     return true;
36               }
37               return false;
38         }

39       public boolean contains(T item)
40       {
41           Node<T> position = head;
42           T itemAtPosition;
43           while (position != null)
44           {
45               itemAtPosition = position.data;
46               if (itemAtPosition.equals(item))
47                   return true;
48               position = position.link;
49           }
50           return false; //target was not found
51       }
```

(continued)

Display 15.37 Set<T> Class (part 3 of 4)

```
52      public void output( )
53      {
54          Node position = head;
55          while (position != null)
56      {
57              System.out.print(position.data.toString( ) + " ");
58              position = position.link;
59          }
60          System.out.println( );
61      }

62        /**
63         Returns a new set that is the union
64         of this set and the input set.
65        */
66        public Set<T> union(Set<T> otherSet)
67        {
68              Set<T> unionSet = new Set<T>( );
69              // Copy this set to unionSet.
70              Node<T> position = head;
71              while (position != null)
72              {
73                    unionSet.add(position.data);
74                    position = position.link;
75              }
76              // Copy otherSet items to unionSet.
77              // The add method eliminates any duplicates.
78              position = otherSet.head;
79              while (position != null)
80              {
81                    unionSet.add(position.data);
82                    position = position.link;
83              }
84              return unionSet;
85        }

86         /**
87          Returns a new set that is the intersection
88          of this set and the input set.
89         */
90         public Set<T> intersection(Set<T> otherSet)
91         {
92               Set<T> interSet = new Set<T>( );
93               // Copy only items in both sets.
94              Node<T> position = head;
95              while (position != null)
96              {
97                    if (otherSet.contains(position.data))
```

Display 15.37 Set<T> Class (part 4 of 4)

```
98                          interSet.add(position.data);
99                      position = position.link;
100                 }
101                 return interSet;
102             }
103     }
```

The `clear`, `size`, *and* `isEmpty` *methods are identical to those in Display 15.8 for the* `LinkedList3` *class.*

"public" in code font

Display 15.38 Set Class Demo (part 1 of 2)

```
1   class SetDemo
2   {
3       public class static void main(String[] args)
4       {
5           // Round things
6           Set round = new Set<String>( );
7           // Green things
8           Set green = new Set<String>( );

9           // Add some data to both sets
10          round.add("peas");
11          round.add("ball");
12          round.add("pie");
13          round.add("grapes");

14          green.add("peas");
15          green.add("grapes");
16          green.add("garden hose");
17          green.add("grass");

18          System.out.println("Contents of set round: ");
19          round.output( );
20          System.out.println("Contents of set green: ");
21          green.output( );
22          System.out.println( );

23          System.out.println("ball in set round? " +
24                round.contains("ball"));
25          System.out.println("ball in set green? " +
26                green.contains("ball"));
```

(continued)

Display 15.38 Set Class Demo (part 2 of 2)

```
27              System.out.println("ball and peas in same set? " +
28                    ((round.contains("ball") &&
29                     (round.contains("peas"))) ||
30                    (green.contains("ball") &&
31                     (green.contains("peas"))));

32              System.out.println("pie and grass in same set? " +
33                    ((round.contains("pie") &&
34                     (round.contains("grass"))) ||
35                    (green.contains("pie") &&
36                     (green.contains("grass"))));

37              System.out.print("Union of green and round: ");
38              round.union(green).output( );
39              System.out.print("Intersection of green and round: ");
40              round.intersection(green).output( );
41        }
42  }
```

Sample Dialogue

```
Contents of set round:
grapes pie ball peas
Contents of set green:
Grass garden hose grapes peas

ball in set round?  true
ball in set green?  false
ball and peas in same set? true
pie and grass in same set? false
Union of green and round: garden hose grass peas ball pie grapes
Intersection of green and round: peas grapes
```

Efficiency of Sets Using Linked Lists

We can analyze the efficiency of our set data structure in terms of the fundamental set operations. Adding an item to the set always inserts a new node on the front of the list. This requires constant, or $O(1)$, steps. The `contains` method iterates through the entire set looking for the target, which requires $O(n)$ steps. When we invoke the `union` method for sets A and B, it iterates through both sets and adds each item into a new set. If there are n items in set A and m items in set B, then $n + m$ add methods are invoked. However, there is a hidden cost because the `add` method searches through its entire list for any duplicates before a new item is added. Although beyond the scope of this text, the additional cost results in $O(m + n)^2$ steps. Finally, the `intersection` method applied to sets A and B invokes the `contains` method of set B for each item in

set A. Since the `contains` method requires $O(m)$ steps for each item in set A, then this requires $O(m) \times O(n)$ steps, or $O(mn)$ steps. These are inefficient methods in our implementation of sets. A different approach to represent the set—for example, one that used hash tables instead of a linked list—could result in an `intersection` method that runs in $O(n + m)$ steps. Nevertheless, our linked list implementation would probably be fine for an application that uses small sets or for an application that does not frequently invoke the `intersection` method, and we have the benefit of relatively simple code that is easy to understand.

If we really needed the efficiency, then we could maintain the same interface to the `Set<T>` class but replace our linked list implementation with something else. If we used the hash table implementation from Section 15.5, then the `contains` method could run in $O(1)$ steps instead of $O(n)$ steps. It might seem like the `intersection` method will now run in $O(n)$ steps, but by switching to a hash table it becomes more difficult to iterate through the set of items. Instead of traversing a single linked list to retrieve every item in the set, the hash table version must now iterate through the hash table array and then for each index in the array iterate through the linked list at that index. If the array is size N and the number of items in the hash table is n, then the iteration time becomes $O(N + n)$. In practice, we would expect N to be larger than n. So although we have decreased the number of steps it takes to look up an item, we have increased the number of steps it takes to iterate over every item. If this was troublesome, you could overcome this problem with an implementation of `Set<T>` that used both a linked list (to facilitate iteration) and a hash table (for fast lookup). However, the complexity of the code is significantly increased using such an approach. You are asked to explore the hash table implementation in Programming Project 10.

Self-Test Exercises

23. Write a method named `difference` that returns the difference between two sets. The method should return a new set that has items from the first set that are not in the second set. For example, if `setA` contains {1, 2, 3, 4} and `setB` contains {2, 4, 5}, then `setA.difference(setB)` should return the set {1, 3}.
24. What is the run time of the `difference` method for the previous question? Give your answer using big-O notation.

15.7 Trees

I think that I shall never see a data structure as useful as a tree.

ANONYMOUS

The tree data structure is an example of a more complicated data structure made with links. Moreover, trees are a very important and widely used data structure. So, we will

briefly outline the general techniques used to construct and manipulate trees. This section is only a very brief introduction to trees to give you the flavor of the subject.

This section uses recursion, which is covered in Chapter 11.

Tree Properties

A tree is a data structure that is structured as shown in Display 15.39. In particular, in a tree you can reach any node from the top (root) node by some path that follows the links. Note that there are no cycles in a tree. If you follow the links, you eventually get to an "end." A definition for a tree class for this sort of tree of ints is outlined in Display 15.39. Note that each node has two references to other nodes (two links) coming from it. This sort of tree is called a **binary tree**, because each node has exactly two link instance variables. There are other kinds of trees with different numbers of link instance variables, but the binary tree is the most common case.

binary tree

The instance variable named `root` serves a purpose similar to that of the instance variable `head` in a linked list (Display 15.3). The node whose reference is in the root instance variable is called the **root node**. Any node in the tree can be reached from the root node by following the links.

root node

The term *tree* may seem like a misnomer. The root is at the top of the tree, and the branching structure looks more like a root branching structure than a tree branching structure. The secret to the terminology is to turn the picture (Display 15.39) upside down. The picture then does resemble the branching structure of a tree, and the root node is where the tree's root would begin. The nodes at the ends of the branches with both link instance variables set to `null` are known as **leaf nodes**, a terminology that may now make some sense. By analogy to an empty linked list, an **empty tree** is denoted by setting the link variable `root` equal to `null`.

leaf node

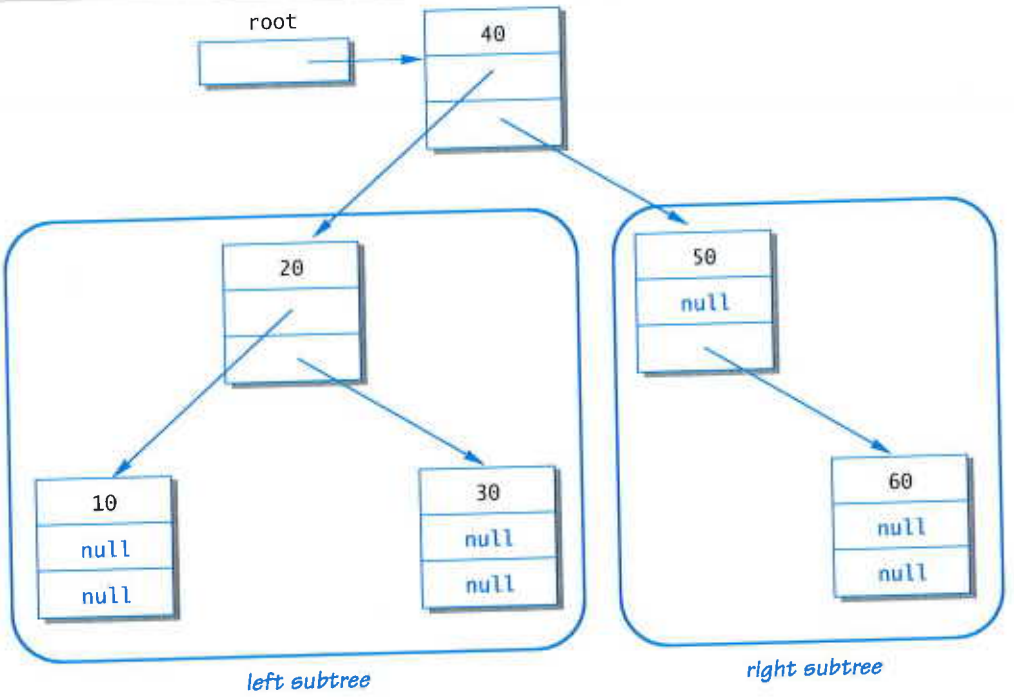
empty tree

Note that a tree has a recursive structure. Each tree has, in effect, two subtrees whose root nodes are the nodes pointed to by the `leftLink` and `rightLink` of the root node. These two subtrees are circled in Display 15.39. This natural recursive structure makes trees particularly amenable to recursive algorithms. For example, consider the task of searching the tree in such a way that you visit each node and do something with the data in the node (such as writing it out to the screen). There is a general plan of attack that goes as follows:

Preorder Processing

1. Process the data in the root node.
2. Process the left subtree.
3. Process the right subtree.

Display 15.39 A Binary Tree




```
1  public class IntTree
2  {
3      public class IntTreeNode
4      {
5          private int data;
6          private IntTreeNode leftLink;
7          private IntTreeNode rightLink;
8      } //End of IntTreeNode inner class

9      private IntTreeNode root;
     <The methods and other inner classes are not shown.>

10 }
```

You obtain a number of variants on this search process by varying the order of these three steps. Two more versions follow:

inorder

Inorder Processing

1. Process the left subtree.
2. Process the data in the root node.
3. Process the right subtree.

postorder

Postorder Processing

1. Process the left subtree.
2. Process the right subtree.
3. Process the data in the root node.

Binary Search Tree Storage Rule

The tree in Display 15.39 has numbers that were stored in the tree in a special way known as the **Binary Search Tree Storage Rule**. The rule is summarized in the following box.

Binary Search Tree Storage Rule

1. All the values in the left subtree are less than the value in the root node.
2. All the values in the right subtree are greater than or equal to the value in the root node.
3. This rule applies recursively to each of the two subtrees.

(The base case for the recursion is an empty tree, which is always considered to satisfy the rule.)

binary search tree

A tree that satisfies the Binary Search Tree Storage Rule is referred to as a **binary search tree**.

Note that if a tree satisfies the Binary Search Tree Storage Rule and you output the values using the Inorder Processing method, then the numbers will be output in order from smallest to largest.

For trees that follow the Binary Search Tree Storage Rule and that are short and fat rather than tall and thin, values can be very quickly retrieved from the tree using a binary search algorithm that is similar in spirit to the binary search algorithm we presented in Display 11.6. The topic of searching and maintaining a binary storage tree to realize this efficiency is a large topic that goes beyond what we have room for here. However, we give one example of a class for trees that satisfy the Binary Search Tree Storage Rule.

EXAMPLE: A Binary Search Tree Class ★

Display 15.40 contains the definition of a class for a binary search tree that satisfies the Binary Search Tree Storage Rule. For simplicity, this tree stores integers, but a routine modification can produce a similar tree class that stores objects of any class that implements the `Comparable` interface. Display 15.41 demonstrates the use of this tree class. Note that no matter in which order the integers are inserted into the tree, the output, which uses inorder traversal, outputs the integers in sorted order.

(continued on page 868)

876

Display 15.40 A Binary Search Tree for Integers (part 1 of 2)

```
1  /**
2   Class invariant: The tree satisfies the binary search tree storage rule.
3  */
4  public class IntTree
5  {
6      private static class IntTreeNode
7      {
8          private int data;
9          private IntTreeNode leftLink;
10         private IntTreeNode rightLink;
11
12         public IntTreeNode(int newData, IntTreeNode newLeftLink,
13                                         IntTreeNode newRightLink)
14         {
15             data = newData;
16             leftLink = newLeftLink;
17             rightLink = newRightLink;
18         }
19     } //End of IntTreeNode inner class

20     private IntTreeNode root;

21     public IntTree( )
22     {
23         root = null;
24     }

25     public void add(int item)
26     {
27         root = insertInSubtree(item, root);
28     }

29     public boolean contains(int item)
30     {
31         return isInSubtree(item, root);
32     }

33     public void showElements( )
34     {
35         showElementsInSubtree(root);
36     }
```

The only reason this inner class is static is that it is used in the `static` *methods* **insertInSubtree**, **isInSubtree**, *and* **showElementsInSubtree**

This class should have more methods. This is just a sample of possible methods

(continued)

Display 15.40 A Binary Search Tree for Integers (part 2 of 2)

```
37      /**
38       Returns the root node of a tree that is the tree with root node
39       subTreeRoot, but with a new node added that contains item.
40      */
41      private static IntTreeNode insertInSubtree(int item,
42                                                 IntTreeNode subTreeRoot)
43      {
44          if (subTreeRoot == null)
45              return new IntTreeNode(item, null, null);
46          else if (item < subTreeRoot.data)
47          {
48              subTreeRoot.leftLink = insertInSubtree(item, subTreeRoot.leftLink);
49              return subTreeRoot;
50          }
51          else //item >= subTreeRoot.data
52          {
53              subTreeRoot.rightLink = insertInSubtree(item, subTreeRoot.rightLink);
54              return subTreeRoot;
55          }
56      }

57      private static boolean isInSubtree(int item, IntTreeNode subTreeRoot)
58      {
59          if (subTreeRoot == null)
60              return false;
61          else if (subTreeRoot.data == item)
62              return true;
63          else if (item < subTreeRoot.data)
64              return isInSubtree(item, subTreeRoot.leftLink);
65          else //item >= link.data
66              return isInSubtree(item, subTreeRoot.rightLink);
67      }

68      private static void showElementsInSubtree(IntTreeNode subTreeRoot)
69      { //Uses inorder traversal.
70          if (subTreeRoot != null)
71          {
72              showElementsInSubtree(subTreeRoot.leftLink);
73              System.out.print(subTreeRoot.data + " ");
74              showElementsInSubtree(subTreeRoot.rightLink);
75          } //else do nothing. Empty tree has nothing to display.
76      }
77  }
```

Display 15.41 Demonstration Program for the Binary Search Tree

```
1  import java.util.Scanner;

2  public class BinarySearchTreeDemo
3  {
4     public static void main(String[] args)
5     {
6        Scanner keyboard = new Scanner(System.in);
7        IntTree tree = new IntTree( );

8        System.out.println("Enter a list of nonnegative integers.");
9        System.out.println("Place a negative integer at the end.");
10       int next = keyboard.nextInt( );
11       while (next >= 0)
12       {
13           tree.add(next);
14           next = keyboard.nextInt( );
15       }

16       System.out.println("In sorted order:");
17       tree.showElements( );
18    }
19 }
```

Sample Dialogue

```
Enter a list of nonnegative integers.
Place a negative integer at the end.
40
30
20
10
11
22
33
44
-1
In sorted order:
10 11 20 22 30 33 40 44
```

EXAMPLE: (continued)

The methods in this class make extensive use of the recursive nature of binary trees. If `aNode` is a reference to any node in the tree (including possibly the root node), then the entire tree with root `aNode` can be decomposed into three parts:

1. The node `aNode`.
2. The left subtree with root node `aNode.leftLink`.
3. The right subtree with root node `aNode.rightLink`.

The left and right subtrees do themselves satisfy the Binary Search Tree Storage Rule, so it is natural to use recursion to process the entire tree by doing the following:

1. Processing the left subtree with root node `aNode.leftLink`
2. Processing the node `aNode`
3. Processing the right subtree with root node `aNode.rightLink`

Note that we processed the root node after the left subtree (inorder traversal). This guarantees that the numbers in the tree are output in the order smallest to largest. The method `showElementsInSubtree` uses a very straightforward implementation of this technique.

Other methods are a bit more subtle in that only one of the two subtrees needs to be processed. For example, consider the method `isInSubtree`, which returns `true` or `false` depending on whether or not the parameter item is in the tree with root node `subTreeRoot`. To see if the item is anyplace in the tree, you set `subTreeRoot` equal to the root of the entire tree, as we did in the method `contains`. However, to express our recursive algorithm for `isInSubtree`, we need to allow for the possibility of subtrees other than the entire tree.

The algorithm for `isInSubtree` expressed in pseudocode is as follows:

```
if (The root node subTreeRoot is empty.)
    return false;
else if (The node subTreeRoot contains item.)
    return true;
else if (item < subTreeRoot.data)
    return (The result of searching the tree with
                                        root node subTreeRoot.leftLink);
    return (The result of searching the tree with else //item > link.data
                                        root node subTreeRoot.rightLink);
```

The reason this algorithm gives the correct result is that the tree satisfies the Binary Search Tree Storage Rule, so we know that if

```
item < subTreeRoot.data
```

EXAMPLE: (continued)

then `item` is in the left subtree (if it is anywhere in the tree), and if

```
item > subTreeRoot.data
```

then `item` is in the right subtree (if it is anywhere in the tree).

The method with the following heading uses techniques very much like those used in `isInSubtree`:

```
private IntTreeNode insertInSubtree(
                    int item, IntTreeNode subTreeRoot)
```

However, there is something new here. We want the method `insertInSubtree` to insert a new node with the data item into the tree with root node `subTreeRoot`. But in this case we want to deal with `subTreeRoot` as a variable and not use it only as the value of the variable `subTreeRoot`. For example, if `subTreeRoot` contains `null`, then we want to change the value of `subTreeRoot` to a reference to a new node containing `item`. However, Java parameters cannot change the value of a variable given as an argument. (Review the discussion of parameters in Chapter 5 if this sounds unfamiliar.) So, we must do something a little different. To change the value of the variable `subTreeRoot`, we return a reference to what we want the new value to be, and we invoke the method `subTreeRoot` as follows:

```
subTreeRoot = insertInSubtree(item, subTreeRoot);
```

That explains why the method `insertInSubtree` returns a reference to a tree node, but we still have to explain why we know it returns a reference to the desired (modified) subtree.

Note that the method `insertInSubtree` searches the tree just as the method `isInSubtree` does, but it does not stop if it finds `item`; instead, it searches until it reaches a leaf node—that is, a node containing `null`. This `null` is where the item belongs in the tree, so it replaces `null` with a new subtree containing a single node that contains `item`. You may need to think about the method `insertInSubtree` a bit to see that it works correctly; allow yourself some time to study the method `insertInSubtree` and be sure you are convinced that after the addition, like so,

```
subTreeRoot = insertInSubtree(item, subTreeRoot);
```

the tree with root node `subTreeRoot` still satisfies the Binary Search Tree Storage Rule.

The rest of the definition of the class `IntTree` is routine.

missing blue star element

Efficiency of Binary Search Trees

When searching a tree that is as short as possible (all paths from root to a leaf differ by at most one node), the search method `isInSubtree`, and hence also the method `contains`, is about as efficient as the binary search on a sorted array (Display 11.6). This should not be a surprise since the two algorithms are in fact very similar. In big-*O* notation, the worst-case running time is $O(\log n)$, where n is the number of nodes in the tree. That means that searching a short, fat binary tree is very efficient. To obtain this efficiency, the tree does not need to be as short as possible so long as it comes close to being as short as possible. As the tree becomes less short and fat and more tall and thin, the efficiency falls off until, in the extreme case, the efficiency is the same as that of searching a linked list with the same number of nodes.

Maintaining a tree so that it remains short and fat as nodes are added is a topic that is beyond the scope of what we have room for in this book. (The technical term for short and fat is *balanced*.) We will only note that if the numbers that are stored in the tree arrive in random order, then with very high probability the tree will be short and fat enough to realize the efficiency discussed in the previous paragraph.

Self-Test Exercises

25. Suppose that the code for the method `showElementsInSubtree` in Display 15.40 were changed so that

```
showElementsInSubtree(subTreeRoot.leftLink);
System.out.print(subTreeRoot.data + " ");
showElementsInSubtree(subTreeRoot.rightLink);
```

were changed to

```
System.out.print(subTreeRoot.data + " ");
showElementsInSubtree(subTreeRoot.leftLink);
showElementsInSubtree(subTreeRoot.rightLink);
```

Will the numbers still be output in ascending order?

26. How can you change the code for the method `showElementsInSubtree` in Display 15.40 so that the numbers are output from largest to smallest instead of from smallest to largest?

Chapter Summary

- A linked list is a data structure consisting of objects known as nodes, such that each node contains data and also a reference to one other node so that the nodes link together to form a list.
- Setting a link instance variable to `null` indicates the end of a linked list (or other linked data structure). `null` is also used to indicate an empty linked list (or other empty linked data structure).
- You can make a linked list (or other linked data structure) self-contained by making the node class an inner class of the linked list class.
- In many situations, a `clone` method or copy constructor is best defined so that it makes a deep copy.
- You can use an iterator to step through the elements of a collection, such as the elements in a linked list.
- Nodes in a doubly linked list have two links—one to the previous node in the list and one to the next node. This makes some operations such as insertion and deletion slightly easier.
- A stack is a data structure in which elements are removed in the reverse of the order they were added to the stack. A queue is a data structure in which elements are removed in the same order that they were added to the queue.
- Big-*O* notation specifies an upper bound for how many steps or how long a program will take to run based on the size of the input to the program. This can be used to analyze the efficiency of an algorithm.
- A hash table is a data structure that is used to store objects and retrieve them efficiently. A hash function is used to map an object to a value that can then be used to index the object.
- Linked lists can be used to implement sets, including common operations such as union, intersection, and set membership.
- A binary tree is a branching linked data structure consisting of nodes that each have two link instance variables. A tree has a special node called the root node. Every node in the tree can be reached from the root node by following links.
- If values are stored in a binary tree in such a way that the Binary Search Tree Storage Rule is followed, then there are efficient algorithms for reaching values stored in the tree.

Answers to Self-Test Exercises

1.
```
mustard 1
hot dogs 12
apple pie 1
```

2. This method has been added to the class `LinkedList1` on the accompanying CD.

```
public boolean isEmpty( )
{
    return (head == null);
}
```

3. This method has been added to the class `LinkedList1` on the accompanying CD.

```
public void clear( )
{
    head = null;
}
```

If you defined your method to remove all nodes using the `deleteHeadNode` method, your method is doing wasted work.

4. Yes. If we make the inner class `Node` a public inner class, it could be used outside the definition of `LinkedList2`, whereas leaving it as private means it cannot be used outside the definition of `LinkedList2`.

5. It would make no difference. Within the definition of an outer class there is full access to the members of an inner class whatever the inner class member's access modifier is. To put it another way, inside the private inner class `Node`, the modifiers `private` and `package access` are equivalent to `public`.

6. Because the outer class has direct access to the instance variables of the inner class `Node`, no access or mutator methods are needed for `Node`.

7. It would be legal, but it would be pretty much a useless method, because you cannot use the type `Node` outside of the class `LinkedList2`. For example, outside of the class `LinkedList2`, the following is illegal (`listObject` is of type `LinkedList2`):

```
Node v = listObject.startNode( ); //Illegal
```

whereas the following would be legal outside of the class `LinkedList2` (although it's hard to think of anyplace you might use it):

```
Object v = listObject.startNode( );
```

8.
```
public class LinkedList2
{
    public class Entry
    {
      private String item;
      private int count;

      public Entry( )
      {
          item = null;
          count = 0;
      }

      public Entry(String itemData, int countData)
      {
          item = itemData;
          count = countData;
      }

      public void setItem(String itemData )
      {
          item = itemData;
      }

      public void setCount(int countData)
      {
          count = countData;
      }

      public String getItem( )
      {
          return item;
      }

      public int getCount(  )
      {
          return count;
      }
    } // End of Entry inner class

    private class Node
    {
        private Entry item;
        private Node link;

        public Node( )
        {
             item = null;
             link = null;
        }
```

```
        public Node(Entry newItem, Node linkValue)
        {
            item = newItem;
            link = linkValue;
        }
    } //End of Node inner class

    private Node head;

    <Other definitions from LinkedList2 go here>

} // End of LinkedList2 class
```

The rest of the definition of `LinkedList2` is essentially the same as in Display 15.7, but with the type `String` replaced by `Entry`. A complete definition is given in the subdirectory named "Exercise 8" on the CD that accompanies this text.

9. No, `T` is not guaranteed to have a copy constructor. Even if `T` has a copy constructor, it is illegal to use `T` with `new` like this.

10. No, you can use any descendent class of `Object` (which means any class type) as the returned type, because the value returned will still be of type `Object`.

11. The `delete` method must now search through the list to find the previous node and then change the link to bypass the current position. This is less efficient than the code in Display 15.17 since the reference to the previous node is already set.

```
public void delete( )
{
  if (position == null)
  {
    throw new IllegalStateException( );
  }
  else
  {
    Node current = head;
    Node previous = null;
    while (current != null)
    {
      if (current == position)
      {
        // Found the node to delete
        // Check if we're at the head
        if (previous == null)
        {
          head = head.link;
          position = head;
        }
        else// Delete in middle of list
        {
```

```
                previous.link = position.link;
                position = position.link;
              }
              return;
            }
            previous = current;  // Advance references
            current = current.link;
          }
        }
      }
```

12. One problem with adding after the iterator's position is that there is no way to add to the front of the list. It would be possible to make a special case in which the new node were added to the front (e.g., if `position` is `null`, add the new data to the head) if desired.

```
public void addAfterHere(String newData)
{
  if (position == null && head != null)
  {
    // At end of list; can't add here
    throw new IllegalStateException( );
  }
  else if (head == null)
    // at head of empty list, add to front
    LinkedList2Iter.this.addToStart(newData);
  else
  {
    // Add after current position
    Node temp = new Node(newData, position.link);
    position.link = temp;
  }
}
```

13.
```
public void changeHere(String newData)
{
    if (position == null)
        throw new IllegalStateException( );
    else
        position.item = newData;
}
```

14. When invoking `i.next( )`, the value of the node that `i` is referencing is copied to a local variable, the iterator moves to the next node in the link, and then the value of the local variable is returned. Therefore, the value that `i` is referencing prior to the invocation is returned.

15. Insertion and deletion is slightly easier with the doubly linked list because we no longer need a separate instance variable to keep track of the previous node due to the previous link. However, all operations require updating more links (e.g., both the next and previous instead of just the previous).

16. Use the iterator:

```
DoublyLinkedList.DoublyLinkedIterator i = list.iterator( );
i.restart( );
i.insertHere("Element At Front");
```

17. Pop the top of the stack and then push it back on:

```
String s = stack.pop( );
Stack.push(s);
// s contains the string on the top of the stack
```

18.
```
public void addToBack(String itemName)
{
    Node newEntry =
          new Node(itemName, null);
    if (front == null) //empty queue
    {
        back = newEntry;
        front = back;
    }
    else
    {
        back.link = newEntry;
        back = back.link;
    }
}
```

19. Just note that $aN + b \leq (a + b)N$, as long as $1 \leq N$.

20. This is mathematics, not Java. So, = will mean *equals*, not assignment.

    First note that $\log_a N = (\log_a b)(\log_b N)$.

    To see this first identity, just note that if you raise a to the power $\log_a N$, you get N, and if you raise a to the power $(\log_a b)(\log_b N)$ you also get N.

    If you set $c = (\log_a b)$, you get $\log_a N = c\,(\log_b N)$.

21. The simplest hash function is to map the ID number to the range of the hash table using the modulus operator:

```
hash = ID % N;      // N is the hash table size
```

22.
```
public void outputHashTable( )
{
  for (int i=0; i< SIZE; i++)
  {
    if (hashArray[i].size( ) > 0)
          hashArray[i].outputList( );
  }
}
```

23. This code is similar to `intersection`, but adds elements if they are not in `otherSet`:

```
public Set<T> difference(Set<T> otherSet)
{
 Set<T> diffSet = new Set<T>( );
 // Copy only items in this set but not otherSet
Node<T> position = head;
while (position != null)
{
   if (!otherSet.contains(position.data))
         diffSet.add(position.data);
   position = position.link;
}
return diffSet;
}
```

24. As implemented above the complexity is identical to the `intersection` method. For every element in the set, we invoke the `contains` method of `otherSet`. This requires $O(nm)$ steps, where n is the number of items in the calling object's set and m is the number of items in `otherSet`'s set.

25. No.

26. Change

```
showElementsInSubtree(subTreeRoot.leftLink);
System.out.print(subTreeRoot.data + " ");
showElementsInSubtree(subTreeRoot.rightLink);
```

to

```
showElementsInSubtree(subTreeRoot.rightLink);
System.out.print(subTreeRoot.data + " ");
showElementsInSubtree(subTreeRoot.leftLink);
```

Programming Projects

myCodeMate *Many of these Programming Projects can be solved using AW's MyCodeMate. To access these please go to:* www.mycodemate.com.

VideoNote
Solution to Programming Project 15.1

1. In an ancient land, the beautiful princess Eve had many suitors. She decided on the following procedure to determine which suitor she would marry. First, all of the suitors would be lined up one after the other and assigned numbers. The first suitor would be number 1, the second number 2, and so on up to the last suitor, number n. Starting at the suitor in the first position, she would then count three suitors down the line (because of the three letters in her name), and the third suitor would be eliminated from winning her hand and removed from the line.

Eve would then continue, counting three more suitors, and eliminate every third suitor. When she reached the end of the line, she would continue counting from the beginning.

For example, if there were six suitors, the elimination process would proceed as follows:

123456	Initial list of suitors; start counting from 1.
12456	Suitor 3 eliminated; continue counting from 4.
1245	Suitor 6 eliminated; continue counting from 1.
125	Suitor 4 eliminated; continue counting from 5.
15	Suitor 2 eliminated; continue counting from 5.
1	Suitor 5 eliminated; 1 is the lucky winner.

Write a program that creates a circular linked list of nodes to determine which position you should stand in to marry the princess if there are *n* suitors. Your program should simulate the elimination process by deleting the node that corresponds to the suitor that is eliminated for each step in the process.

2. Although the `long` data type can store large integers, it can't store extremely large values such as an integer with 200 digits. Create a `HugeNumber` class that uses a linked list of digits to represent integers of arbitrary length. The class should have a method to add a new most significant digit to the existing number so that longer and longer numbers can be created. Also add methods to reset the number and to return the value of the huge integer as a `String` along with appropriate constructor or accessor methods. Write code to test your class.

   Note: Use of a doubly linked list will make the next problem easier to implement.

3. Add a copy constructor to the `HugeNumber` class described in the previous problem that makes a deep copy of the input `HugeNumber`. Also create an `add` method that adds an input `HugeNumber` to the instance's `HugeNumber` value and returns a new `HugeNumber` that is set to the sum of the two values. Write code to test the additions to your class.

4. Give the definition of a generic class that uses a doubly linked list of data items. Include a copy constructor, an `equals` method, a `clone` method, a `toString` method, a method to produce an iterator, and any other methods that would normally be expected. Write a suitable test program.

5. Complete the definition of the binary search tree class `IntTree` in Display 15.39 by adding the following: Make `IntTree` implement the `Cloneable` interface, including the definition of a `clone` method; add a copy constructor; add an `equals` method; add a method named `sameContents` as described later in this project; add a `toString` method; and add a method to produce an iterator. Define `equals` so that two trees are equal if (and only if) the two trees have the exact same shape and have the same numbers in corresponding nodes. The `clone` method and the copy

constructor should each produce a deep copy that is equal to the original list according to the `equals` method. The `boolean` valued method `sameContents` has one parameter of type `IntTree` and returns `true` if the calling object and the argument tree contain exactly the same numbers, and returns `false` otherwise. Note that `equals` and `sameContents` are not the same. Also, write a suitable test program.

6. Write an `addSorted` method for the generic linked list from Display 15.8 such that the method adds a new node in the correct location so that the list remains in sorted order. Note that this will require that the type parameter `T` extend the `Comparable` interface. Write a suitable test program.

7. Add a `remove` method and an iterator for the `Set` class in Display 15.37. Write a suitable test program.

8. The hash table from Display 15.34 hashed a string to an integer and stored the same string in the hash table. Modify the program so that instead of storing strings it stores `Employee` objects as defined in Display 7.2. Use the `name` instance variable as the input to the hash function. The modification will require changes to the linked list, since the `LinkedList2` class only created linked lists of strings. For the most generality, modify the hash table so that it uses the generic `LinkedList3` class defined in Display 15.8. You will also need to add a `get` method that returns the `Employee` object stored in the hash table that corresponds to the input name. Test your program by adding and retrieving several names, including names that hash to the same slot in the hash table.

9. Display 15.34 and 15.35 provide the beginnings of a spell checker. Refine the program to make it more useful. The modified program should read in a text file, parse each word, see if it is in the hash table, and, if not, output the line number and word of the potentially misspelled word. Discard any punctuation in the original text file. Use the `words.txt` file as the basis for the hash table dictionary. This file can be found on the CD accompanying the textbook and also on the book's website. The file contains 45,407 common words and names in the English language. Note that some words are capitalized. Test your spell checker on a short text document.

10. Change the `Set<T>` class of Display 15.37 so that internally it uses a hash table to store its data instead of a linked list. The headers of the public methods should remain the same so that a program such as the demonstration in Display 15.38 should still work without requiring any changes. Add a constructor that allows the user of the new `Set<T>` class to specify the size of the hash table array.

    For an additional challenge, implement the set using both a hash table and a linked list. Items added to the set should be stored using both data structures. Any operation requiring lookup of an item should use the hash table, and any operation requiring iteration through the items should use the linked list.

Next, define a class named `Circle` that implements `Shape`. The `Circle` class should have an instance variable for the radius, a constructor that sets the radius, accessor/mutator methods for the radius, and an implementation of the `area` method. Also define a class named `Rectangle` that implements `Shape`. The `Rectangle` class should have instance variables for the height and width, a constructor that sets the height and width, accessor mutator methods for the height and width, and an implementation of the `area` method.

The following test code should then output the area of the `Circle` and `Rectangle` objects:

```
public static void main(String[] args)
{
  Circle c = new Circle(4);  // Radius of 4
  Rectangle r = new Rectangle(4,3); // Height=4, Width=3
  ShowArea(c);
  ShowArea(r);
}

public static void ShowArea(Shape s)
{
      double area = s.area();
      System.out.println("The area of the shape is " + area);
}
```

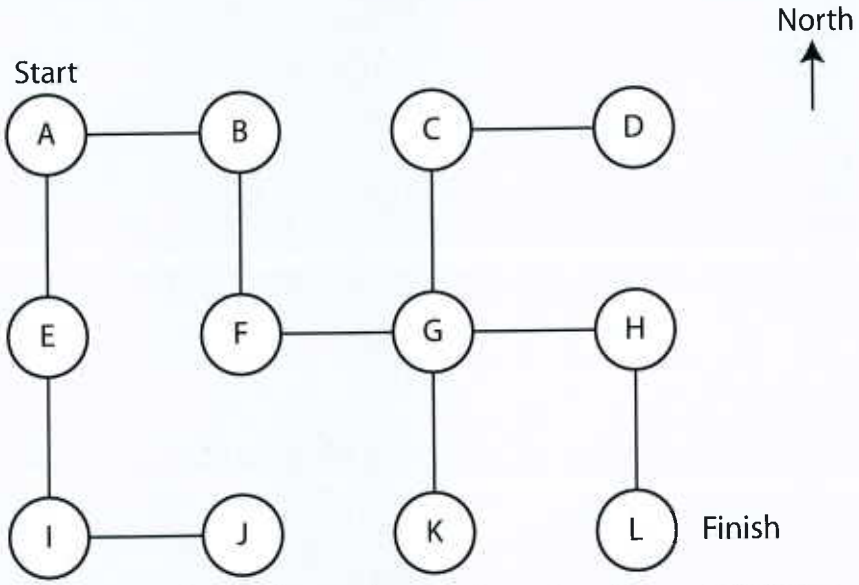
Chapter 14

7. Programming Project 6.13 implemented a simple trivia game using an array of `Trivia` objects. Redo this project but use an `ArrayList` of `Trivia` objects instead of an array. The runtime behavior should remain identical to before.

Chapter 15

11. The following figure is called a *graph*. The circles are called *nodes* and the lines are called *edges*. An edge connects two nodes. You can interpret the graph as a maze of rooms and passages. The nodes can be thought of as rooms and an edge connects one room to another. Note that each node has at most four edges in the graph below.

11. The following figure is called a *graph*. The circles are called *nodes* and the lines are called *edges*. An edge connects two nodes. You can interpret the graph as a maze of rooms and passages. The nodes can be thought of as rooms and an edge connects one room to another. Note that each node has at most four edges in the graph below.



Write a program that implements the maze above using references to instances of a `Node` class. Each node in the graph will correspond to an instance of `Node`. The edges correspond to links that connect one node to another and can be represented in `Node` as instance variables that reference another `Node` class. Start the user in node A. The user's goal is to reach the finish in node L. The program should output possible moves in the north, south, east, or west direction. Sample execution is shown below.

```
You are in room A of a maze of twisty little passages, all alike. You
can go east or west.
E
You are in room B of a maze of twisty little passages, all alike. You
can go west or south.
S
You are in room F of a maze of twisty little passages, all alike. You
can go north or east.
E
```