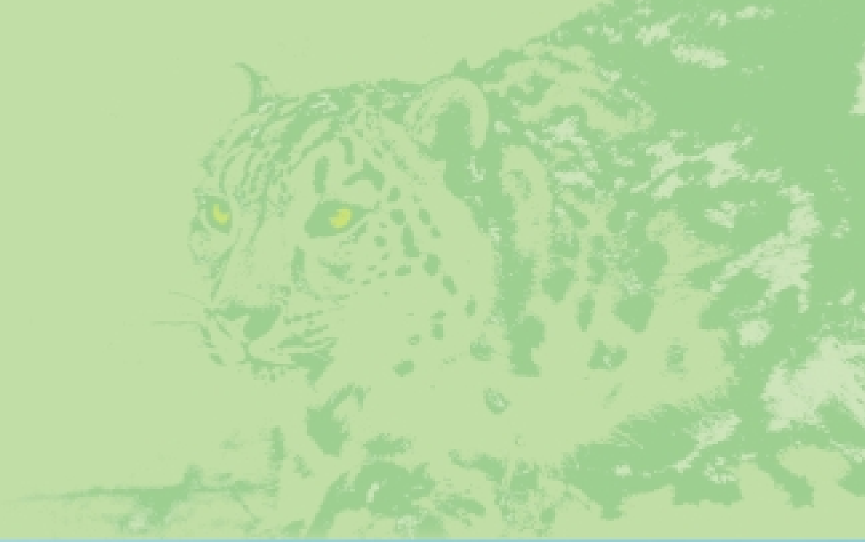
CHAPTER 14

Linked Data Structures



14 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

A **linked data structure** consists of capsules of data known as **nodes** that are connected via **links** that can be viewed as arrows. 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 14.1. In Display 14.1 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. The last node must be able to serve as a kind of end marker. Your code must be able to tell when it has reached the last node.

nodes and link

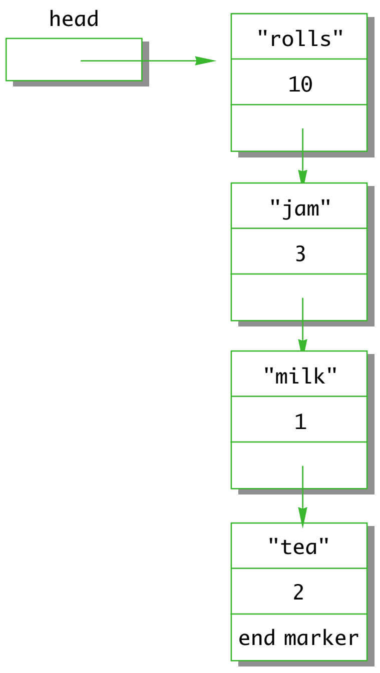
linked list

That's all very vague but is 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, 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 *trees*, a more elaborate linked data structure.

Display 14.1 Nodes and Links in a Linked List



PREREQUISITES

If you prefer, you may skip this chapter and go directly to Chapter 15 on vectors and collection classes or you may go directly to Chapter 16 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.

Section 14.1 on linked lists requires material from Chapters 1 through 5, simple uses of inner classes (Section 13.2 of Chapter 13), cloning (Section 13.1 of Chapter 13), and simple use of exceptions (Chapter 9). Section 14.2 requires all of this plus Section 14.1 and Chapter 11 on recursion.

14.1 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

linked list

nevertheless widely used. In this section we give examples of and develop techniques for working with linked lists in Java.

Example

A Simple Linked List Class

Display 14.1 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 14.2. 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.

link

Node

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.)

head node

head

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 14.1 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 node head is not itself the head of the linked list, but contains a reference to the head node (that is, to the first 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 14.3 contains a definition of a linked list class for a linked list like the one in Display 14.1. 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 14.1 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.

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 14.3, 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?

Display 14.2 **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 contains 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 14.3 A Linked List Class *(Part 1 of 2)*

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

```
public class LinkedList1
{
    private Node1 head;

    public LinkedList1()
    {
        head = null;
    }

    /**
     Adds a node at the start of the list with the specified data.
     The added node will be the first node in the list.
    */
    public void add(String itemName, int itemCount)
    {
        head = new Node1(itemName, itemCount, head);
    }

    /**
     Removes the head node and returns true if the list contains at least
     one node. Returns false if the list is empty.
    */
    public boolean deleteHeadNode()
    {
        if (head != null)
        {
            head = head.getLink();
            return true;
        }
        else
            return false;
    }

    /**
     Returns the number of nodes in the list.
    */
    public int size()
    {
        int count = 0;
        Node1 position = head;

```

Display 14.3 A Linked List Class *(Part 2 of 2)*

```
        while (position != null)
        {
            count++;
            position = position.getLink();
        }
        return count;
    }

    public boolean contains(String item)
    {
        return (find(item) != null);
    }

    /**
     Finds the first node containing the target item, and returns a
     reference to that node. If target is not in the list, null is returned.
    */
    private Node1 find(String target)
    {
        Node1 position = head;
        String itemAtPosition;
        while (position != null)
        {
            itemAtPosition = position.getItem();
            if (itemAtPosition.equals(target))
                return position;
            position = position.getLink();
        }
        return null; //target was not found
    }

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

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

This is the way you traverse an entire 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 14.4. 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.

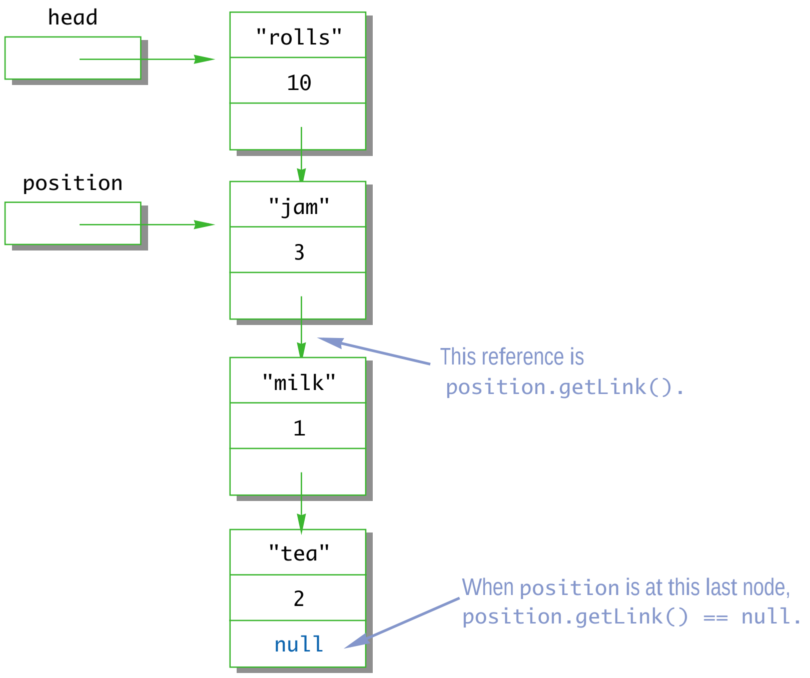
Indicating the End of a Linked List

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`.

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 one constructor sets this `head` instance variable to `null`, indicating that a newly created linked list is empty.

Display 14.4 **Traversing a Linked List**


head
position
"rolls"
10
"jam"
3
"milk"
1
"tea"
2
null
This reference is position.getLink().
When position is at this last node, position.getLink() == 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 it works out nicely for many linked list manipulation algorithms.

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 14.3), whose body is reproduced here:

traversing a linked list

```
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 14.4. 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 14.4. 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 14.4, 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 `add` 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. In other words, the variable `head` is set 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 an 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 14.5.

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();
```

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**.

garbage collection

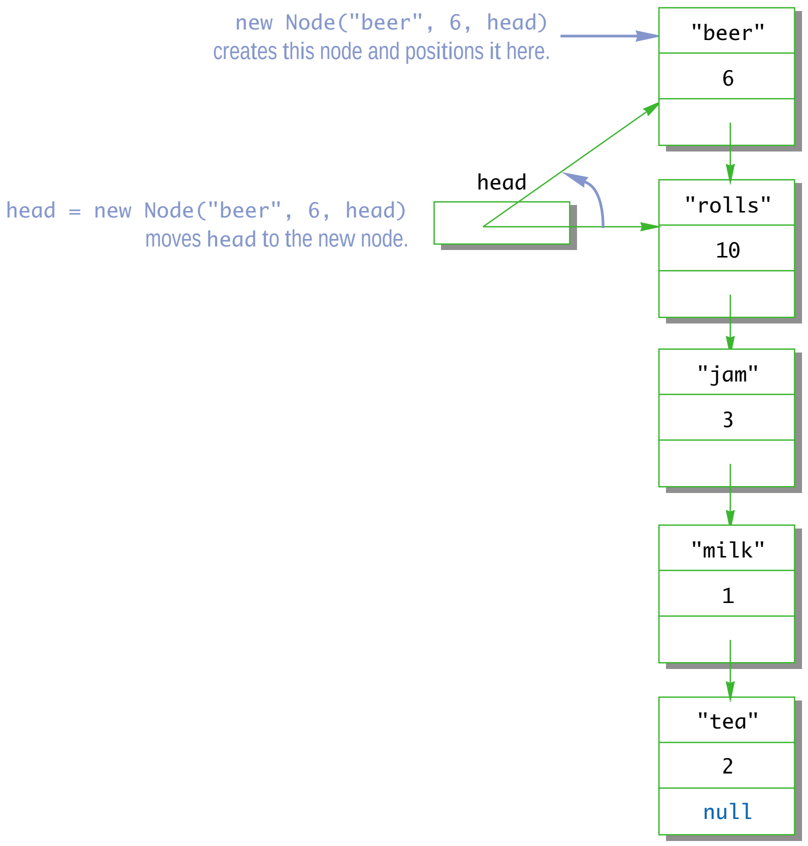
Display 14.6 contains a simple program that demonstrates how some of the methods in the class `LinkedList1` behave.

Self-Test Exercises

1. What output is produced by the following code?

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

Display 14.5 **Adding a Node at the Start**


new Node("beer", 6, head)
creates this node and positions it here.
head = new Node("beer", 6, head)
moves head to the new node.
head
"beer"
6
"rolls"
10
"jam"
3
"milk"
1
"tea"
2
null

Display 14.6 **A Linked List Demonstration *(Part 1 of 2)***

```
public class LinkedList1Demo
{
    public static void main(String[] args)
    {
        LinkedList1 list = new LinkedList1();
        list.add("Apples", 1);
        list.add("Bananas", 2);
        list.add("Cantaloupe", 3);
        System.out.println("List has " + list.size()
                                + " nodes.");
        list.outputList();
```

Cantaloupe is now in the head node.

Display 14.6 **A Linked List Demonstration *(Part 2 of 2)***

```
        if (list.contains("Cantaloupe"))
            System.out.println("Cantaloupe is on list.");
        else
            System.out.println("Cantaloupe is NOT on list.");

        list.deleteHeadNode();

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

        while (list.deleteHeadNode())
            ; //Empty loop body

        System.out.println("Start of list:");
        list.outputList();
        System.out.println("End of list.");
    }
}
```

Removes 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.
```

```
list.outputList();
```

2. Define a `boolean` valued method named `isEmpty` that can be added to the class `LinkedList1` (Display 14.3). 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 14.3). 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 14.2). 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 `LinkedList1`, as discussed in the next subsection.

Although there is no problem with the class definition of `Node1` when it is used in a class definition like `LinkedList1`, there is no way to guarantee that the class `Node1` will be used only in this way, unless you do something similar to making the class `Node1` a private inner class in the class `LinkedList1`.

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 linked lists 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 14.2.>
    }
```

```
        private Node1 head;
       <The constructor and methods in Display 14.3 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 14.7, we have rewritten the class `LinkedList1` in this way. The rewritten version, named `LinkedList2`, is equivalent to the class `LinkedList1` in Display 14.3 in that it has the same methods that perform the same actions.

Display 14.7 A Linked List Class with a Node Inner Class *(Part 1 of 3)*

```
public class LinkedList2
{
    private class Node2
    {
        private String item;
        private int count;
        private Node2 link;

        public Node2()
        {
             item = null;
             count = 0;
             link = null;
        }

        public Node2(String newItem, int newCount, Node2 linkValue)
        {
            item = newItem;
            count = newCount;
            link = linkValue;
        }
     }//End of Node2 inner class

    private Node2 head;

    public LinkedList2()
    {
        head = null;
    }
```

It makes no difference whether we make the instance variables of `Node2` public or private.

An inner class for the node class

We have simplified this class and the previous linked list class to keep them relatively short. These classes should have a copy constructor, an `equals` method, and a `clone` method. Our next linked list example includes these items.

Display 14.7 A Linked List Class with a Node Inner Class *(Part 2 of 3)*

```
    /**
     Adds a node at the start of the list with the specified data.
     The added node will be the first node in the list.
    */
    public void add(String itemName, int itemCount)
    {
        head = new Node2(itemName, itemCount, head);
    }

    /**
     Removes the head node and returns true if the list contains at least
     one node. Returns false if the list is empty.
    */
    public boolean deleteHeadNode()
    {
        if (head != null)
        {
            head = head.link;
            return true;
        }
        else
            return false;
    }

    /**
     Returns the number of nodes in the list.
    */
    public int size()
    {
        int count = 0;
        Node2 position = head;
        while (position != null)
        {
            count++;
            position = position.link;
        }
        return count;
    }

    public boolean contains(String item)
    {
        return (find(item) != null);
    }
```

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

Display 14.7 **A Linked List Class with a Node Inner Class *(Part 3 of 3)***

```
    /**
     Finds the first node containing the target item, and returns a
     reference to that node. If target is not in the list, null is returned.
    */
    private Node2 find(String target)
    {
        Node2 position = head;
        String itemAtPosition;
        while (position != null)
        {
            itemAtPosition = position.item;
            if (itemAtPosition.equals(target))
                return position;
            position = position.link;
        }
        return null; //target was not found
    }

    public void outputList()
    {
        Node2 position = head;
        while (position != null)
        {
            System.out.println(position.item + " "
                                  + position.count);
            position = position.link;
        }
    }

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

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

Node Inner Class

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.

Self-Test Exercises

4. Would it make any difference if we change the `Node1` inner class in Display 14.7 from a private inner class to a public inner class?
5. Keeping the inner class `Node1` in Display 14.7 as private, what difference would it make if any of the instance variables or methods in the class `Node1` have its access modifiers changed from what it is `public`, `private`, or package access?
6. Why does the definition of the inner class `Node2` in Display 14.7 not have the accessor and mutator methods `getLink`, `setLink`, or `get` and `set` methods for the `data` field as the class definition of `Node1` in Display 14.2 does?
7. Would it be legal to add the following method to the class `LinkedList2` in Display 14.7?

```
public Node2 startNode()
{
    return head;
}
```

Example

A Generic Linked List

Display 14.8 shows a linked list whose Node class has a single data item of type `Object`. This means that the linked list can hold objects of any class type (including the possibility of array objects). In most other respects this linked list has the same methods, coded in basically the same way, as our previous linked list (Display 14.7), but we have added a copy constructor, an `equals` method, and a `clone` method, as well as a couple of private helping methods. We will discuss the `equals` method first.

`equals`

The `equals` method is defined so that two linked lists are equal if, and only if, they are of the same size (same number of nodes) and the data in the nodes is pairwise equal: 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 is equal, and so forth. To test if the data in two nodes is equal, the method uses the `equals` method for the data in the node of the calling object. Since the data is an object of type `Object`, we know it has an `equals` method. We are trusting the programmer who wrote the class definition for the class of the data in the nodes. 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.

The private helping method `copyOf` is used in defining both the copy constructor and the `clone` method. 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

Display 14.8 A Generic Linked List Class *(Part 1 of 5)*

```
public class GenericLinkedList implements Cloneable
{
    private class Node
    {
        private Object data;
        private Node link;

        public Node()
        {
             data = null;
             link = null;
        }

        public Node(Object newData, Node linkValue)
        {
            data = newData;
            link = linkValue;
        }
    }//End of Node inner class

    private Node head;

    public GenericLinkedList()
    {
        head = null;
    }

    /**
     Throws a NullPointerException if other is null.
     Caution: the data in the linked lists are not cloned;
     only a reference to the data Object is copied
    */
    public GenericLinkedList(GenericLinkedList otherList)
    {
        if (otherList == null)
            throw new NullPointerException();
        if (otherList.head == null)
            head = null;
        else
            head = copyOf(otherList.head);
    }
```

This linked list holds objects of any kind. However, the objects should have well-defined `equals` and `toString` methods.

Warning: This is not an ideal copy constructor. The copy created and `otherList` share references to data objects. However, as explained in the text, this problem cannot be fixed.

A `NullPointerException` need not be caught or declared in a `throws` clause.

Display 14.8 A Generic Linked List Class *(Part 2 of 5)*

```java
    //Precondition: otherHead != null
    //Returns a reference to the head of a copy of the
    //list headed by otherHead.
    private Node copyOf(Node otherHead)
    {
        Node position = otherHead;//moves down other's list.
        Node newHead; //will point to head of the copy list.
        Node end = null; //positioned at end of new growing list.
        //Create first node:
        newHead = new Node(position.data, null);
        end = newHead;
        position = position.link;

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

        return newHead;
     }

    public Object clone()
    {
        try
        {
            GenericLinkedList copy =
                              (GenericLinkedList)super.clone();
            copy.head = copyOf(this.head);
            return copy;
        }
        catch(CloneNotSupportedException e)
        {//This should not happen.
            return null; //To keep the compiler happy.
        }
    }

    /**
     Adds a node at the head of the list with the newData.
    */
    public void add(Object newData)
    {
        head = new Node(newData, head);
    }
```

Warning: This is not an ideal `clone` method. The copy produced and the original list (the calling object) share references to data objects. However, as explained in the text, this problem cannot be fixed.

Display 14.8 **A Generic Linked List Class *(Part 3 of 5)***

```
    /**
     Removes the head node and returns true if the list contains at least
     one node. Returns false if the list is empty.
    */
    public boolean deleteHeadNode()
    {
        if (head != null)
        {
            head = head.link;
            return true;
        }
        else
            return false;
    }

    /**
     Returns the number of nodes in the list.
    */
    public int size()
    {
        int count = 0;
        Node position = head;
        while (position != null)
        {
            count++;
            position = position.link;
        }
        return count;
    }

    public boolean contains(Object target)
    {
        return (find(target) != null);
    }

    /**
     Finds the first node containing the target item, and returns a
     reference to that node. If target is not in the list, null is returned.
    */
    private Node find(Object target)
    {
        Node position = head;
        Object dataAtPosition;
```

Display 14.8 A Generic Linked List Class *(Part 4 of 5)*

```
        while (position != null)
        {
            dataAtPosition = position.data;
            if (target.equals(dataAtPosition))
                return position;
            position = position.link;
        }
        return null; //target was not found
    }

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

    public void clear()
    {
        head = null;
    }

    public boolean equals(Object otherList)
    {
        if (otherList == null)
            return false;
        else if (getClass() != otherList.getClass())
            return false;
        else if (size() != ((GenericLinkedList)otherList).size())
            return false;
        else
            return compareLists((GenericLinkedList)otherList);
    }

    //Precondition: size() == otherList.size().
    //Returns true if node by node, objects are equal.
    private boolean compareLists(GenericLinkedList otherList)
    {
        boolean match = true;//so far
        Node position = head;
        Node otherPosition = otherList.head;
```

`target`'s `equals` method is used to test if the data in a node "is the same as" `target`.

Display 14.8 A Generic Linked List Class *(Part 5 of 5)*

```
            while (match && position != null)
            {
                if (!position.equals(otherPosition))
                    match = false;
                position = position.link;
                otherPosition = otherPosition.link;
            }
            return match;
        }

        public void outputList()
        {
            Node position = head;
            while (position != null)
            {
                System.out.println(position.data);
                position = position.link;
            }
        }

    }
```

Objects in the linked list use their own `equals` method to test if they equal an object in `otherList`.

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 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 a copy of the list in the calling object is created.

The copy constructor and the `clone` method are 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 `clone` method each produce a new linked list with all new nodes, the new lists are not truly independent because the data object is not cloned. See the next Pitfall section for a discussion of this point.

Pitfall

The clone Method Is Protected in Object

When defining the copy constructor and so the `clone` method for our generic linked list (Display 14.8), 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((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((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.

If this modified code (with the `clone` method) would compile, it 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`. Since the designers of the `Object` class chose to make the method `clone` protected, you simply cannot use the `clone` method in the `GenericLinkedList` class. So, we simply did the best we could with the copy constructor and `clone` method of the class `GenericLinkedList`. We have no way to make a completely independent copy in this situation.

Why was the `clone` method labeled `protected` in `Object`? Apparently for security reasons. If a class could inherit 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.

You might object that, since `GenericLinkedList` has no base class, it has an implicit base class of `Object` and so it should inherit a protected method such as `clone`. That's a good point. The explanation is a bit subtle. Let's explore it a bit more.

The following is a useless method, except that it does show that the method `clone`, which was inherited from `Object`, is in some sense available in `GenericLinkedList`. The following method will compile with no problems if added to `GenericLinkedList`:

```
public void justATest()
{
        GenericLinkedList toy = new GenericLinkedList();
        GenericLinkedList toy2 =
                      (GenericLinkedList)(toy.clone());
    }
```

Now let's return to our discussion of the method copyOf in which we could not use the inherited method clone. If the inherited method clone could be invoked in justATest, why can't it be invoked in the method copyOf, as follows?

```
head = new Node((position.data).clone(), null);
```

The problem is that position.data is of type Object, not of type GenericLinkedList. In our example of the method justATest, which did work, toy.clone() was okay because toy was of type GenericLinkedList and this was taking place inside the definition of GenericLinkedList. The inherited method clone is only available to the class GenericLinkedList. For an object of type Object, the protected method clone acts the same as if it were private in Object. This general point is discussed in the Pitfall section of Chapter 7 entitled "A Restriction on Protected Access." It may help to review that Pitfall section.

One solution to this problem is to not use the class Object as the type for the data object, but to instead use a more specialized class. For example, you can use the class Employee from Display 7.2, with the usual equals (Display 7.8) and clone (Self-Test Exercise 17 of Chapter 13) methods added, and then you can have a linked list of objects from any descendent classes such as the classes HourlyEmployee (Display 7.3 with the usual equals and clone methods added) and SalariedEmployee (Display 7.5 with the usual equals and clone methods added). Moreover, since the clone method is redefined to be public in the class Employee, we could use the clone method to clone the data object in a node. This means the linked list can create truly independent copies of a linked list. A class called LinkedListOfEmployees could be defined just like the class GenericLinkedList except for the following three changes:

1. The name GenericLinkedList is replaced by LinkedListOfEmployees wherever it occurs.
2. The name Object is replaced by Employee every place except that the return type for clone is left as Object and the type for the parameter to equals is left as Object.
3. The definition of the method copyOf has invocations of clone added as shown in Display 14.9.

The resulting class is described in Display 14.9. The version of LinkedListOfEmployees given on the accompanying CD is complete and includes all the parts omitted from Display 14.9. (The CD also includes the definitions of the classes Date, Employee, HourlyEmployee, and SalariedEmployee updated to include the usual equals and clone methods.)

extra code on CD

Note that, unlike the class GenericLinkedList, the class LinkedListOfEmployees has a clone method that produces a truly independent copy of the calling object; that is, a copy with no references in common with the linked list of the calling object.

Note that there are two places in LinkedListOfEmployees where we did not replace Object with Employee: The return type for the method clone is left as Object and the type for the parameter to equals is left as Object. We did this so that the clone and equals methods would be overrides of the inherited clone and equals methods.

Display 14.9 A Linked List with a True clone Method *(Part 1 of 2)*

```
public class LinkedListOfEmployees implements Cloneable
{
    private class Node
    {
        private Employee data;
        private Node link;

        public Node()
        {
             data = null;
             link = null;
        }

        public Node(Employee newData, Node linkValue)
        {
            data = newData;
            link = linkValue;
        }
    }//End of Node inner class

    private Node head;

    /**
     Throws a NullPointerException if other is null. Produces a completely
     independent copy with no references in common with otherList.
    */
    public LinkedListOfEmployees(LinkedListOfEmployees otherList)
    {
        if (otherList == null)
            throw new NullPointerException();
        if (otherList.head == null)
            head = null;
        else
            head = copyOf(otherList.head);
    }

    //Precondition: otherHead != null
    //Returns a reference to the head of a copy of the
    //list headed by otherHead. Does a deep copy.
    private Node copyOf(Node otherHead)
    {
        Node position = otherHead;//moves down other's list.
        Node newHead; //will point to head of the copy list.
        Node end = null; //positioned at end of new growing list.
```

Copy constructor produces a completely independent copy (a deep copy) of its argument.

Display 14.9 **A LinkedList with a True clone Method *(Part 2 of 2)***

```
        //Create first node:
        newHead =
              new Node((Employee)((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((Employee)((position.data).clone()), null);
            end = end.link;
            position = position.link;
        }

        return newHead;
    }

    public Object clone()
    {
        try
        {
            LinkedListOfEmployees copy =
                                  (LinkedListOfEmployees)super.clone();
            copy.head = copyOf(this.head);
            return copy;
        }
        catch(CloneNotSupportedException e)
        {//This should not happen.
            return null; //To keep the compiler happy.
        }
    }
     public boolean equals(Object otherList)
    {
        if (otherList == null)
            return false;
        else if (getClass() != otherList.getClass())
            return false;
        else if (size() != ((LinkedListOfEmployees)otherList).size())
            return false;
        else
            return compareLists((LinkedListOfEmployees)otherList);
    }
```

You need these type casts because `clone` returns a value of type `Object`.

This is a true `clone` method that returns a completely independent copy (a deep copy) of the calling object.

<All the other methods and constructors are the same as in `GenericLinkedList` in Display 14.8, except that `GenericLinkedList` is replaced with `LinkedListOfEmployees` and `Object` is replaced with `Employee`, throughout.>

```
}
```

Tip

Deep Copy versus Shallow Copy

Contrast the copy constructors in the linked lists GenericLinkedList in Display 14.8 and LinkedListOfEmployees in Display 14.9. The difference is in the methods normalCopy which are used in the copy constructors of the two classes.

In the method copyOf for the class LinkedListOfEmployees, each new node is given a clone of the data in the old node. That way, the old list and the new copy have no references in common, so any change to the new list will not have any effect on the old list (and vice versa). This kind of copy is called a **deep copy**. A deep copy of an object (such as a linked list) is a copy that has no references in common with the original. So, the deep copy and the original are completely independent copies, and changes to the copy will have no effect on the original.

Making a deep copy usually involves using the clone method on items inside the thing being copied. This requires that the clone methods used in making a deep copy do themselves make deep copies.

Any copy that is not a deep copy is called a **shallow copy**. For example, the copy constructor in the linked list GenericLinkedList in Display 14.8 makes a shallow copy. The reason that it is a shallow copy is that, in the method normalCopy for the class GenericLinkedList, each new node is given only a reference to the data in the old node, rather than a clone of the data. So, changes to the new list can have an effect on the old list.

There are degrees of shallowness. For example, if we redefine the copy constructor for the class GenericLinkedList in Display 14.8 so that the copy constructor simply copies the reference in the head instance variable to the head instance variable of the new copy, that would be an even shallower copy. In this case the list and its copy would be the same and would have all references in common. However, we are usually concerned with only whether or not a copy is a deep copy. So, we lump all kinds of shallow copies together and use the single term *shallow copy* for all kinds of shallow copies.

In the majority of situations a deep copy is preferable to a shallow copy. In particular, a copy constructor and a clone method should normally make a deep copy if that is possible. There are cases, although not as common, where you might want a shallow copy.

EXCEPTIONS

A generic data structure, such as GenericLinkedList in Display 14.8 or even LinkedListOfEmployees in Display 14.9, 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 constructors in GenericLinkedList in Display 14.8 and LinkedListOfEmployees in Display 14.9. A NullPointerException is the kind of exception that need not be caught or declared in a throws clause. When thrown by a method of a linked list class, it can be treated as simply a run-time error message or the exception can instead be caught in a catch block if there is some suitable action that can be taken.

Self-Test Exercises

8. Would it be legal to change the parameter type of `equals` in the class `LinkedListOfEmployees` (Display 14.9) from `Object` to `LinkedListOfEmployees`? You should not do this, but the question is not whether you should do this, but whether it is legal.

9. Would it be legal to change the return type of `clone` in the class `LinkedListOfEmployees` (Display 14.9) from `Object` to `LinkedListOfEmployees`? You should not do this, but the question is not whether you should do this, but whether it is legal.

ITERATORS

iterator

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 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.

ITERATORS

Suppose you have a collection of data items, such as a linked list. Any object that allows you to step through the collection one item at a time so that each item is visited exactly once in one full cycle of iterations is called an **iterator**.

In Display 14.10 we have rewritten the class `LinkedListOfEmployees` from Display 14.9 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 `EmployeeIterator` public so that we can have variables of type `EmployeeIterator` outside the class `LinkedListOfEmployees`, but we do not otherwise plan to use the inner class `EmployeeIterator` outside of the outer class `LinkedListOfEmployees`.

Use of iterators for the class `LinkedListOfEmployees` is illustrated by the program in Display 14.11. Note that, given a link list named `list`, an iterator for `list` is produced by the method `iterator` as follows:

```
LinkedListOfEmployees.EmployeeIterator 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

```
LinkedListOfEmployees.EmployeeIterator
```

Display 14.10 A Linked List with an Iterator *(Part 1 of 2)*

```
import java.util.NoSuchElementException;

public class LinkedListOfEmployees
{
    private class Node
    {
        private Employee data;
        private Node link;
      <The full definition of the Node inner class is given in Display 14.9.>
}//End of Node inner class

    /**
     If the list is altered any iterators should invoke restart or
     the iterator's behavior may not be as desired.
    */
    public class EmployeeIterator
    {
        private Node position;
        private Node previous;//previous value of position

        public EmployeeIterator()
        {
            position = head; //Instance variable head of outer class.
            previous = null;
        }

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

        public Employee next()
        {
            if (!hasNext())
                throw new NoSuchElementException();

            Employee toReturn = position.data;
            previous = position;
            position = position.link;
            return toReturn;
        }
```

This is the same as the class in Display 14.9 except that the `EmployeeIterator` inner class and the `iterator()` method have been added.

An inner class for iterators for `LinkedListOfEmployees`.

Display 14.10 A Linked List with an Iterator *(Part 2 of 2)*

```
        public boolean hasNext()
        {
            return (position != null);
        }

        /**
         Returns the next value to be returned by next().
         Throws an IllegalStateExpression if hasNext() is false.
        */
        public Employee peek()
        {
             if (!hasNext())
                throw new IllegalStateException();
             return position.data;
        }

        /**
         Adds a node before the node at location position.
         previous is placed at the new node. If hasNext() is
         false, then the node is added to the end of the list.
         If the list is empty, inserts node as the only node.
        */
        public void add(Employee newData)
        <The rest of the method add is Self-Test Exercise 10.>

        /**
         Deletes the node at location position and
         moves position to the "next" node.
         Throws an IllegalStateException if the list is empty.
        */
        public void delete()
        <The rest of the method delete is Self-Test Exercise 11.>
    }//End of EmployeeIterator inner class

    private Node head;

    public EmployeeIterator iterator()
    {
        return new EmployeeIterator();
    }
<The other methods and constructors are identical to those in Display 14.9.>
}
```

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

Display 14.11 **Using an Iterator *(Part 1 of 2)***

```
public class IteratorDemo
{
    public static void main(String[] args)
    {
        LinkedListOfEmployees list = new LinkedListOfEmployees();
        list.add(new Employee("Sandy Hair", new Date(1, 1, 2000)));
        list.add(new HourlyEmployee(
                "Dusty Rhodes", new Date(2, 2, 2001), 25.00, 40.0));
        list.add(new SalariedEmployee(
                      "Chuck Steak", new Date(3, 3, 2002), 80000));

        LinkedListOfEmployees.EmployeeIterator i = list.iterator();

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

        i.restart();
        i.next();
        System.out.println("Will delete node for "
                                                  + (i.peek()).getName());
        i.delete();

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

        i.restart();
        i.next();
        System.out.println("Will add one node before "
                                        + (i.peek()).getName());
        i.add(new Employee(
                      "Natalie Dressed", new Date(4, 4, 2003)));
        System.out.println("List now contains:");
        i.restart();
        while(i.hasNext())
            System.out.println(i.next());
```

You can add any kind of Employee to the linked list.

Display 14.11 **Using an Iterator *(Part 2 of 2)***

```
            System.out.println();
            System.out.println("Changing all names to Kilroy.");
            i.restart();
            while(i.hasNext())
                (i.next()).setName("Kilroy");
            System.out.println();

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

`next()` returns a reference, so you can modify the data in a node.

Sample Dialogue

```
List contains:
Chuck Steak Mar 3, 2002
$80000.0 per year
Dusty Rhodes Feb 2, 2001
$25.0 per hour for 40.0 hours
Sandy Hair Jan 1, 2000

Will delete node for Dusty Rhodes
List now contains:
Chuck Steak Mar 3, 2002
$80000.0 per year
Sandy Hair Jan 1, 2000

Will add one node before Sandy Hair
List now contains:
Chuck Steak Mar 3, 2002
$80000.0 per year
Natalie Dressed Apr 4, 2003
Sandy Hair Jan 1, 2000

Changing all names to Kilroy.

List now contains:
Kilroy Mar 3, 2002
$80000.0 per year
Kilroy Apr 4, 2003
Kilroy Jan 1, 2000
```

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.

As we have defined the iterator inner class `EmployeeIterator`, the method `next` returns a reference to the data item in a node. This allows you to modify the data in the nodes as the iterator cycles through all the nodes in the collection, This is illustrated by the following code from that program in Display 14.11:

```
System.out.println("Changing all names to Kilroy.");
i.restart();
while(i.hasNext())
    (i.next()).setName("Kilroy");
```

You can also define iterators that do not allow the programmer to change the linked list, as explained in the upcoming subsection entitled “Immutable Iterators.”

The definitions of the methods `add` and `delete` are left as Self-Test Exercises. However, we will give the basic technique for adding and deleting nodes in the middle of a linked list. (And if you cannot get the definitions after that, you can look up the definition in the answers to the Self-Test Exercises.) The techniques for adding and deleting nodes are in the next subsection.

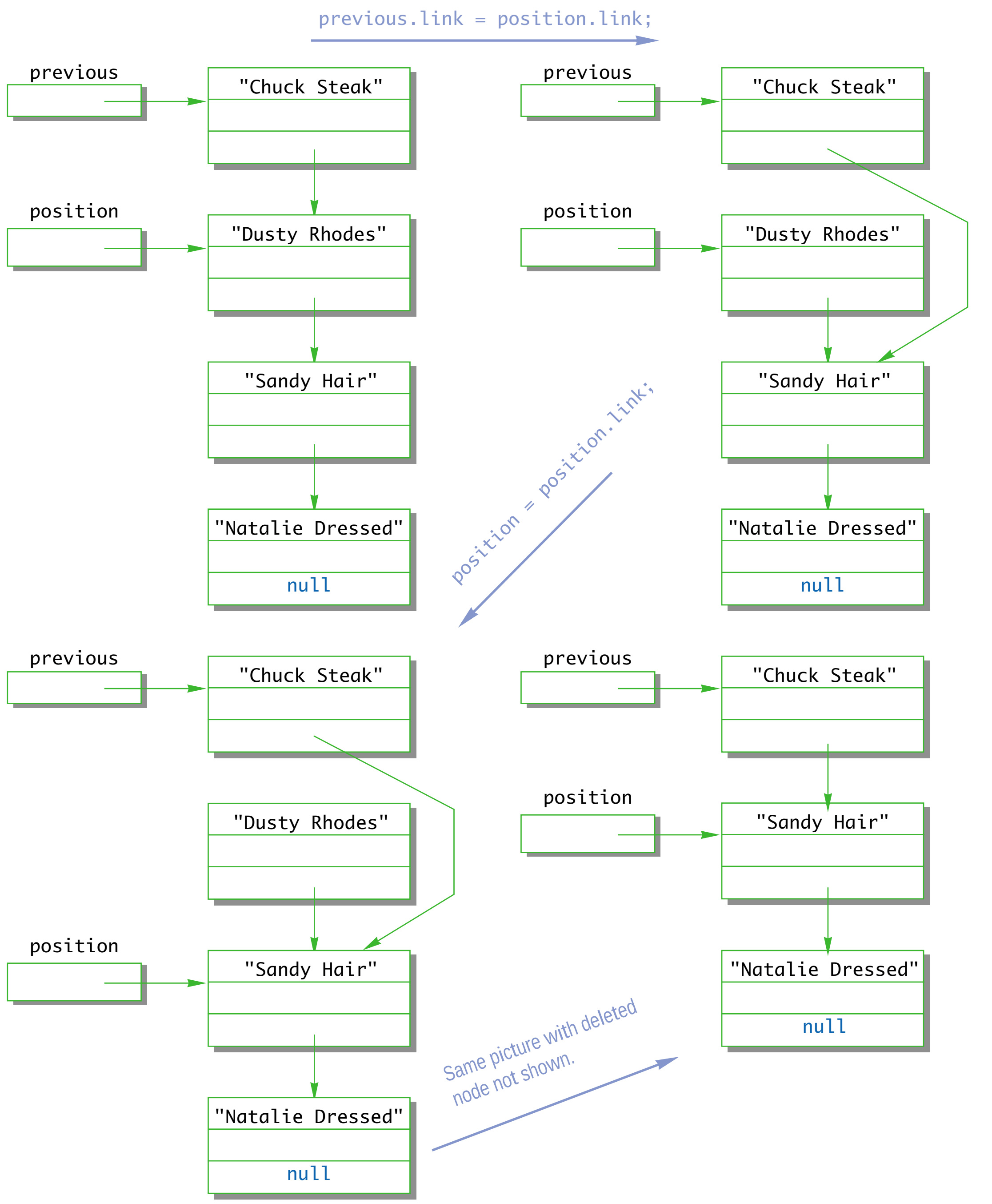
ADDING AND DELETING NODES

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 14.12 shows the technique for deleting a node. The linked list is an object of the class `LinkedListOfEmployees`. The variables `position` and `previous` are the instance variables of an iterator for the linked list object. The variables `position` and `previous` each hold a reference to a node, indicated with an arrow. Much of the data in the data object of a node is not shown; only the employee’s name is shown. As indicated in Display 14.12, the node at location `position` is deleted by the following two lines of code:

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

Display 14.12 Deleting a Node


previous.link = position.link;
previous
position
"Chuck Steak"
"Dusty Rhodes"
"Sandy Hair"
"Natalie Dressed"
null
position = position.link;
Same picture with deleted node not shown.

When your code deletes a node from a linked list (as in Display 14.12), it removes the linked list's reference to that node. So, as far as the linked list is concerned, the 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**.

Display 14.13 shows the technique for adding a node. We want to add a new node between the nodes named by `previous` and `position`. In Display 14.13, `previous` and `position` are variables of type `Node` and each contains a reference to a node indicated with an arrow. So, the new node goes between the two nodes circled in the first picture.

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

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

So we can recognize the node with `newData` in it, let's assume the name part of the data in `newData` is `"Sandy Hair"`. 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 move the circled arrow in the second picture. 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.

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`.

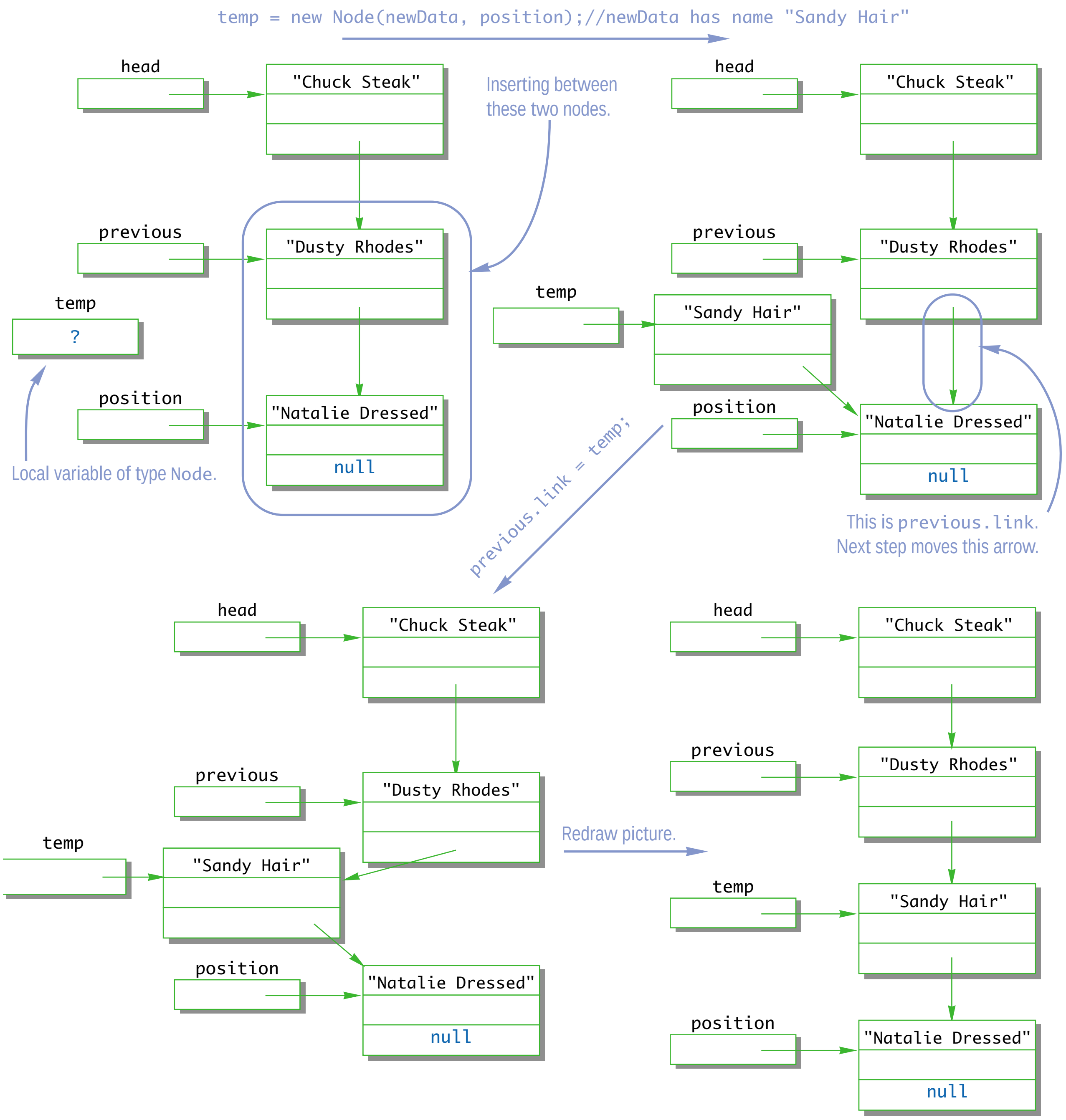
```
temp = new Node(newData, position);
previous.link = temp;
```

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

IMMUTABLE ITERATORS

An **immutable iterator** is one that cannot use the mutator methods of the data object to change a data object in the linked list. If you do not want to use an iterator to modify the elements in a linked list, you can make it an immutable iterator by having the `next` method return a clone of the data rather than a reference to the data. This will reduce the

Display 14.13 **Adding a Node between Two Nodes**



temp = new Node(newData, position);//newData has name "Sandy Hair"
head
"Chuck Steak"
Inserting between these two nodes.
previous
"Dusty Rhodes"
temp
?
position
"Natalie Dressed"
null
Local variable of type Node.
"Sandy Hair"
This is previous.link. Next step moves this arrow.
previous.link = temp;
Redraw picture.

risk of privacy leaks. To obtain an immutable iterator for the class `LinkedListOfEmployees`, redefine the `next` method in Display 14.10 to the one shown in Display 14.14.

Note that an immutable iterator can modify a linked list by using the methods `add` and `remove`. However, it does not allow somebody to receive a reference to a data object and later change it unexpectedly.

If you change the definition of `next` in `LinkedListOfEmployees` to the one in Display 14.14, then when you run the program in Display 14.11, the output would change to the following:

```
List contains:
Chuck Steak Mar 3, 2002
$80000.0 per year
Dusty Rhodes Feb 2, 2001
$25.0 per hour for 40.0 hours
Sandy Hair Jan 1, 2000

Will delete node for Dusty Rhodes
List now contains:
Chuck Steak Mar 3, 2002
$80000.0 per year
Sandy Hair Jan 1, 2000

Will add one node before Sandy Hair
List now contains:
Chuck Steak Mar 3, 2002
$80000.0 per year
Natalie Dressed Apr 4, 2003
Sandy Hair Jan 1, 2000

Changing all names to Kilroy:

List now contains:
Chuck Steak Mar 3, 2002
$80000.0 per year
Natalie Dressed Apr 4, 2003
Sandy Hair Jan 1, 2000
```

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 15.

Display 14.14 A next Method for an Immutable Iterator

Except for the definition of the method next, this definition of LinkedListOfEmployees is identical to the one given in Display 14.10.

```
import java.util.NoSuchElementException;

public class LinkedListOfEmployees
{
    private class Node
    {
        private Employee data;
        private Node link;

      <The full definition of the Node inner class is given in Display 14.9.>

    }//End of Node inner class

    public class EmployeeIterator
    {
        private Node position;
        private Node previous;//previous value of position

        public Employee next()
        {
            if (!hasNext())
                throw new NoSuchElementException();

            Employee toReturn = (Employee)((position.data).clone());
            previous = position;
            position = position.link;
            return toReturn;
        }

<The rest of the definition of the EmployeeIterator is given in Display 14.9.>

    }//End of Iterator inner class

    private Node head;

    public EmployeeIterator iterator()
    {
        return new EmployeeIterator();
    }

<The other methods and constructors for LinkedListOfEmployees
                are identical to those in Display 14.9.>
}
```

Note that the add and remove methods can still modify the linked list, but the mutator methods, like setName, can no longer change the original data item because they would be working on a clone of the data object contained in the list.

Self-Test Exercises

10. Complete the definition of the method add in the inner class EmployeeIterator in Display 14.10.

11. Complete the definition of the method delete in the inner class EmployeeIterator in Display 14.10.

VARIATIONS ON A LINKED LIST

Sometimes it is handy to have a reference to the last node in a linked list. This last node is often called the **tail** of the list, so the linked list definition might begin as follows: tail

```
public class LinkedListOfEmployees
{
    <Inner class definitions.>
    private Node head;
    private Node tail;
```

To fully carry out this addition to the class LinkedListOfEmployees in Display 14.9, the constructors and methods must be modified to accommodate this new reference tail, but the details of doing so are routine.

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. Diagrammatically, a doubly linked list looks like the sample list in Display 14.15. doubly linked list

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

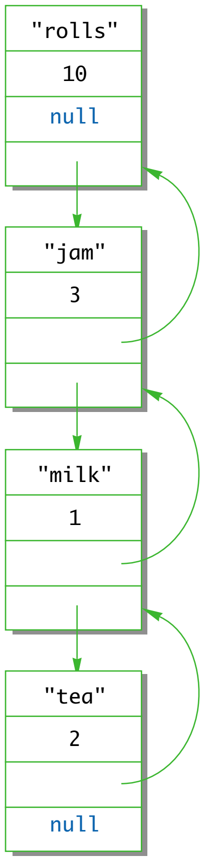
```
private class TwoWayNode
{
    private Object data;
    private TwoWayNode previous;
    private TwoWayNode next;
```

The constructors and some of the methods in the doubly linked list class will have changes (from the singly linked case) in their definitions to accommodate the extra link.

THE STACK DATA STRUCTURE

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 stack

Display 14.15 **A Doubly Linked List**



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 14.3) is in fact a stack.

14.2 Trees

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

Anonymous

A detailed treatment of trees 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. However, the tree data structure will be 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 14.16. 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 `int`s is outlined in Display 14.16. 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 14.3). 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 14.16) 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.

leaf node

By analogy to an empty linked list, an empty tree is denoted by setting the link variable `root` equal to `null`.

empty tree

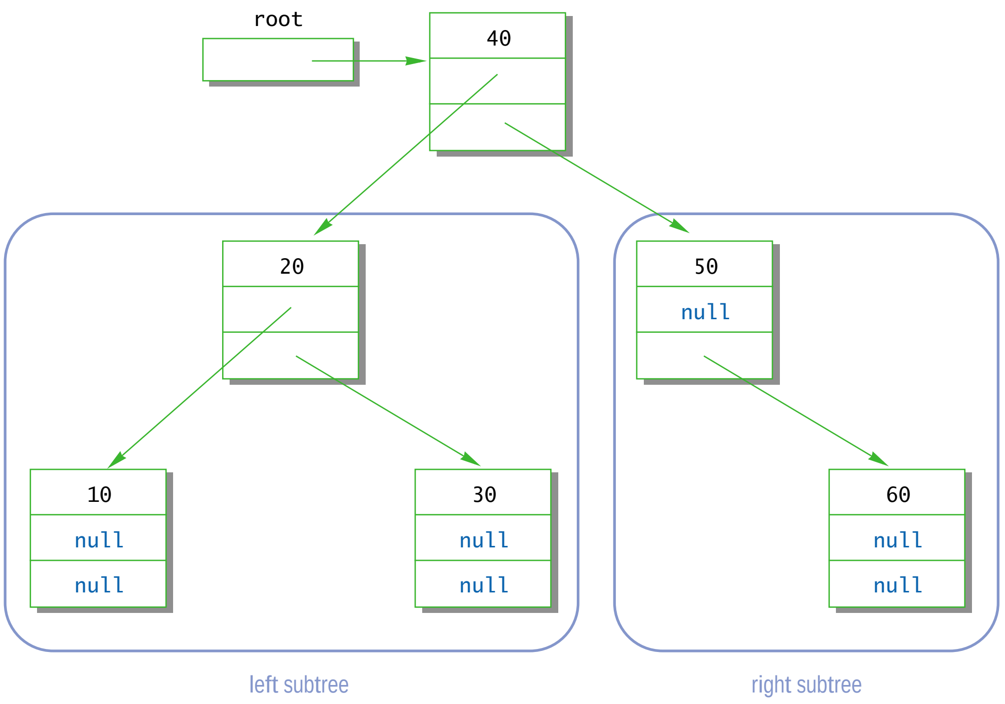
Note that a tree has a recursive structure. Each tree has, in effect, two subtrees whose root nodes are the nodes pointed to by the `leftLink` and `rightLink` of the root node. These two subtrees are circled in color in Display 14.16. This natural recursive structure makes trees particularly amenable 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

preorder

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

Display 14.16 A Binary Tree



```
public class IntTree
{
    public class IntTreeNode
    {
        private int data;
        private IntTreeNode leftLink;
        private IntTreeNode rightLink;
    } //End of IntTreeNode inner class

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

}
```

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

Inorder Processing

inorder

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

Postorder Processing

postorder

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

The tree in Display 14.16 has numbers that were stored in the tree in a special way known as the **Binary Search Tree Storage Rule**. The rule is as follows:

Binary Search Tree Storage Rule

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.)

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

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 11.8. 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 tree that satisfies the Binary Search Tree Storage Rule.

Example

A Binary Search Tree Class ✣

Display 14.17 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 14.18 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.

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:

The node *aNode*.

The left subtree with root node *aNode*`.leftLink`.

The right subtree with root node *aNode*`.rightLink`.

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

Processing the left subtree with root node *aNode*`.leftLink`.

Processing the node *aNode*.

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 the possibility of subtrees other than the entire tree.

The algorithm for `isInSubtree` expressed in pseudo code is

```
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);
else //item > link.data
     return (The result of searching the tree with
                root node subTreeRoot.rightLink);
```

Display 14.17 A Binary Search Tree for Integers *(Part 1 of 2)*

```
/**
 Class invariant: The tree satisfies the binary search tree storage rule.
*/
public class IntTree
{
    private static class IntTreeNode
    {
        private int data;
        private IntTreeNode leftLink;
        private IntTreeNode rightLink;

        public IntTreeNode(int newData, IntTreeNode newLeftLink,
                                        IntTreeNode newRightLink)
        {
            data = newData;
            leftLink = newLeftLink;
            rightLink = newRightLink;
        }
    } //End of IntTreeNode inner class


    private IntTreeNode root;

    public IntTree()
    {
        root = null;
    }

    public void add(int item)
    {
        root = insertInSubtree(item, root);
    }

    public boolean contains(int item)
    {
        return isInSubtree(item, root);
    }

    public void showElements()
    {
        showElementsInSubtree(root);
    }
```

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.

Display 14.17 A Binary Search Tree for Integers *(Part 2 of 2)*

```
    /**
     Returns the root node of a tree that is the tree with root node
     subTreeRoot, but with a new node added that contains item.
    */
    private static IntTreeNode insertInSubtree(int item,
                                               IntTreeNode subTreeRoot)
    {
        if (subTreeRoot == null)
            return new IntTreeNode(item, null, null);
        else if (item < subTreeRoot.data)
        {
            subTreeRoot.leftLink = insertInSubtree(item, subTreeRoot.leftLink);
            return subTreeRoot;
        }
        else //item >= subTreeRoot.data
        {
            subTreeRoot.rightLink = insertInSubtree(item, subTreeRoot.rightLink);
            return subTreeRoot;
        }
    }

    private static boolean isInSubtree(int item, IntTreeNode subTreeRoot)
    {
        if (subTreeRoot == null)
            return false;
        else if (subTreeRoot.data == item)
            return true;
        else if (item < subTreeRoot.data)
            return isInSubtree(item, subTreeRoot.leftLink);
        else //item >= link.data
            return isInSubtree(item, subTreeRoot.rightLink);
    }

    private static void showElementsInSubtree(IntTreeNode subTreeRoot)
    {//Uses inorder traversal.
        if (subTreeRoot != null)
        {
            showElementsInSubtree(subTreeRoot.leftLink);
            System.out.print(subTreeRoot.data + " ");
            showElementsInSubtree(subTreeRoot.rightLink);
        }//else do nothing. Empty tree has nothing to display.
    }
}
```

Display 14.18 **Demonstration Program for the Binary Search Tree**

```java
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class BinarySearchTreeDemo
{
    public static void main(String[] args) throws IOException
    {
        BufferedReader keyboard =
                new BufferedReader(new InputStreamReader(System.in));
        IntTree tree = new IntTree();

        System.out.print("Enter a list of nonnegative integers,");
        System.out.println(" one per line.");
        System.out.println("Place a negative integer at the end.");
        String nextNumberString = keyboard.readLine();
        int next = Integer.parseInt(nextNumberString);
        while (next >= 0)
        {
            tree.add(next);
            nextNumberString = keyboard.readLine();
            next = Integer.parseInt(nextNumberString);
        }

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

SAMPLE DIALOGUE

```
Enter a list of nonnegative integers, one per line.
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
```

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
```

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`. But, 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.

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 11.8). This should not be a surprise since the two algorithms are in fact very similar.[1] 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.

balanced tree

Self-Test Exercises

12. Suppose that the code for the method `showElementsInSubtree` in Display 14.17 were changed so that

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

were change to

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

Will the numbers still be output in ascending order?

13. How can you change the code for the method `showElementsInSubtree` in Display 14.17 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 linked data structure).

[1] For those who may be familiar with the notation, the worst-case running time is $O(\log n)$, where n is the number of nodes in the tree.

- 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.
- You can use an iterator to step through the elements of a collection, such as the elements in a linked list.
- 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
```

extra code on CD

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

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

extra code on CD

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 `Node1` a public inner class, it could be used outside the definition of `LinkedList2`, while 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 `Node1`, the modifiers `private` and package access are equivalent to `public`.

6. Since the outer class has direct access to the instance variables of the inner class `Node2`, no access or mutator methods are needed for `Node2`.

7. It would be legal, but it would be pretty much a useless method, since you cannot use the type `Node2` outside of the class `LinkedList2`. For example, outside of the class `LinkedList2` the following is illegal (`listObject` is of type `LinkedList2`):

   ```
   Node2 v = listObject.startNode(); //Illegal
   ```

   while 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. It is legal, but will result in `LinkedListOfEmployees` having two `equals` methods: the one inherited from the class `Object` that has a parameter of type `Object`, and the one with the parameter of type `LinkedListOfEmployees`. This would be overloading the method name `equals`, not overriding the method `equals`.

9. It is not legal since the class `LinkedListOfEmployees` inherits a method named `clone()` and you cannot have two methods with the same name and parameter list but with different returned types; or, to rephrase it, you cannot overload a method name based on the type returned.

10.
```
public void add(Employee newData)
{
    if (position == null && previous != null)
    //if at end of list
        previous.link =
                     new Node(newData, null);
    else if (position == null || previous == null)
    //else if list is empty or position is at head node.
        LinkedListOfEmployees.this.add(newData);
    else//previous and position are located
        //at two consecutive nodes.
    {
        Node temp = new Node(newData, position);
        previous.link = temp;
        previous = temp;
    }
}
```

11.
```
public void delete()
{
    if (position == null)
        throw new IllegalStateException();
    else if (previous == null)
    {//remove node at head
        head = head.link;
        position = head;
    }
```

```
        else //previous and position are at two nodes.
        {
            previous.link = position.link;
            position = position.link;
        }
    }
```

12. No.

13. 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

1. Give the definition of a class for a doubly linked list of data items of type `Object`. 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.

2. Design and implement a class that is a class for polynomials. The polynomial

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0$$

will be implemented as a linked list. Each node will contain an `int` value for the power of *x* and an `int` value for the corresponding coefficient. The class operations should include addition, subtraction, multiplication, and evaluation of a polynomial. Overload the operators +, –, and * for addition, subtraction, and multiplication. Evaluation of a polynomial is implemented as a method named `evaluation` that has one argument of type `int`. The `evaluation` method returns the value obtained by plugging in its argument for *x* and performing the indicated operations. Include four constructors: a default constructor, a copy constructor, a constructor with a single argument of type `int` that produces the polynomial that has only one constant term that is equal to the constructor argument, and a constructor with two arguments of type `int` that produces the one-term polynomial whose coefficient and exponent are given by the two arguments. (In the above notation the polynomial produced by the one-argument constructor is of the simple form consisting of only a_0. The polynomial produced by the two-argument constructor is of the slightly more complicated form $a_n x^n$.) Include a method to input a polynomial. When the user inputs a polynomial, the user types in the following:

a_n`x^`n `+` a_{n-1}`x^`n–1 `+...+` a_0

However, if a coefficient a_i is zero, the user may omit the term a_i`x^` i. For example, the polynomial

$3x^4 + 7x^2 + 5$

can be input as

```
3x^4 + 7x^2 + 5
```

It could also be input as

```
3x^4 + 0x^3 + 7x^2 + 0x^1 + 5
```

If a coefficient is negative, a minus sign is used in place of a plus sign, as in the following examples:

```
3x^5 – 7x^3 + 2x^1 – 8
–7x^4 + 5x^2 + 9
```

A minus sign at the front of the polynomial, as in the second of the above two examples, applies only to the first coefficient; it does not negate the entire polynomial. To simplify input, you can assume that polynomials are always entered one per line and that there will always be a constant term a_0. If there is no constant term, the user enters zero for the constant term, as in the following:

```
12x^8 + 3x^2 + 0
```

Polynomials are output in the same format. In the case of output, the terms with zero coefficients are not output. Include a `toString` method and define it so that `System.out.println` will output polynomials in this format. Also be sure to include `equals` and `clone` methods. Write a suitable test program.

3. Complete the definition of the binary search tree class `IntTree` in Display 14.17 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.

For additional online Programming Projects, click the CodeMate icons below.

14.4 CODEMATE 14.5 CODEMATE