15 Collections And Linked Lists

Overview

This chapter requires that you have a solid understanding of arrays (Chapter Seven) and some idea of recursion (Section 5.9 or 13.4). It is also valuable to have studied Sections 10.1-10.2 (Exceptions) and 14.1-14.3 (stacks and queues implemented with linked lists).

- Section 15.1 presents a software context in which you need programs that work with large collections of data.
- Section 15.2 introduces the official (Sun standard library) Collection interface and an array-based implementation of the methods that do not modify the Collection.
- Sections 15.3-15.5 describe linked lists and their application to implementing the non-modifying methods of the Collection interface, in some cases recursively.
- Sections 15.6-15.7 define the Iterator interface and show how it can be implemented in both the array and linked list forms.
- Section 15.8 implements the methods that modify a Collection object.
- Sections 15.9-15.10 discuss an implementation of ListIterator and List using doubly-linked lists.

This chapter should help you develop a strong understanding of the Collection and List interfaces, strengthen your abilities in working with arrays, and further develop facility with linked lists. In everyday programming, when you want to use a Collection or List object, you will tend to choose the "built-in" ArrayList class (Section 7.11) or some other standard library implementation rather than one you build yourself (in accordance with the "don't reinvent the wheel" principle). But you can only learn about linked lists by using them to code various methods, so that is what we do here.

15.1 Analysis And Design Of The Inventory Software

You have a client who needs a number of programs that read in two files of data and then process them. One file lists all the retail items that the company currently has on hand (its inventory). Another file lists all the retail items that the company has purchased and, according to its records, not yet sold. Obviously, the two lists should match up. However, discrepancies are common.

For the first such program, it is sufficient to tell whether the two lists are exactly equal and, if not, whether the inventory file at least contains all the retail items purchased (so the store has not had any retail items stolen). If even that is not true, then you need to say how many retail items are in each list. A reasonable plan for the program, developed after a bit more discussion with the client to clear up some details, is shown in the accompanying design block.

STRUCTURED NATURAL LANGUAGE DESIGN of the main logic

1. Read the "inven.dat" file into an appropriate object named `inventory`.
2. Read the "purch.dat" file into an appropriate object named `purchased`.
3. If one list has the same elements in the same order as the other list then...
   Print the message "a perfect match".
4. Otherwise, if the `inventory` list has no elements at all then...
   Print the message "we've been robbed".
5. Otherwise, if every element of the `purchased` list is in the `inventory` list then...
   Print the message "no losses and some gains".
6. Otherwise...
   Print the message "inventory has N and purchased has M"
   where N and M are the number of values in the respective lists.

Object design

We need a class of objects that store sequences of data items. We will call it the ArraySequence class. We will make it a Collection kind of class; Collection is an interface in the Sun standard library that specifies the names of thirteen methods, how they are called, and what they should accomplish. The Collection methods provide all the capabilities that you need for this program. The next section describes them in detail. Listing 15.1 shows how some of them are used to solve the client's problem.

Listing 15.1 The Inventory application program

```
public class Inventory
{
   /** Read 2 files of data and make comparisons between them. */

   public static void main (String[] args)
   {  ArraySequence inventory;
      ArraySequence purchased;
      try
      {  inventory = new ArraySequence ("inven.dat");
         purchased = new ArraySequence ("purch.dat");
      }catch (java.io.IOException e)
      {  throw new RuntimeException ("A file is defective.");
      }

      if (inventory.equals (purchased))
         System.out.println ("a perfect match");
      else if (inventory.isEmpty())
         System.out.println ("we've been robbed");
      else if (inventory.containsAll (purchased))
         System.out.println ("no losses and some gains");
      else
         System.out.println ("inventory has " + inventory.size()
                  + " and purchased has " + purchased.size());
   }  //=====================
}
```

The ArraySequence constructor takes a String value as a parameter, opens the disk file of that name, and reads in the values from the file one line at a time. The lines from the file are stored in the ArraySequence object in the order they were read. If the file is not accessible, the constructor throws an IOException; this main method catches it and notifies the caller of the method (it throws a new Exception instead of terminating the program because this main method may be called from another method).

The effects of the other methods used in this program should be clear when compared with the design block. In any case, they are explained in detail in the next section. Figure 15.1 is a UML diagram for this Inventory class.

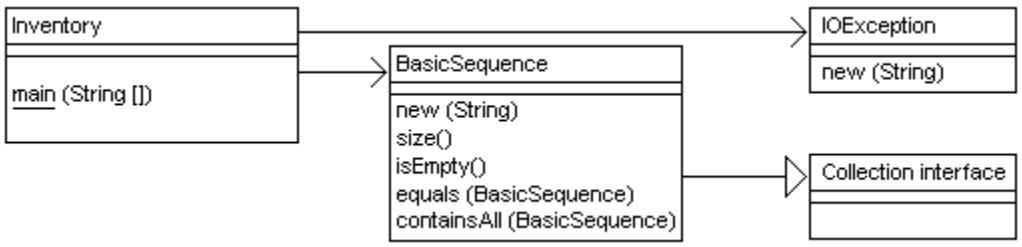

Figure 15.1 UML class diagram for Inventory

15.2 Implementing The Collection Interface With Arrays

A **sequence** is a collection of data values in a particular order. The data values are called the **elements** of the sequence. We define here the ArraySequence class with two constructors and several other methods, implemented using a partially-filled array. The sequence is allowed to have duplicates (two or more elements that are equal to each other), but it is not allowed to have an element be null.

The Collection interface

The Sun standard library contains a number of interfaces for describing ways of structuring collections of data. The most fundamental one is the **Collection** interface (in the `java.util` package) for storing Object values. The ArraySequence class will be an implementation of the Collection interface when we finish adding enough methods. Listing 15.2 (see next page) gives the documentation for the Collection interface, with full descriptions of those methods that do not change the elements in the Collection.

People use a number of variants of Collection implementations. The main variants are as follows:

- Some variants do not allow null as an element; some do.
- Some variants do not allow duplicates (i.e., all elements must be different from each other); some do.
- Some variants do not allow changes in the number or ordering of the elements once they are set by a constructor; some do. Those that do not allow changes have the last six listed methods throw an UnsupportedOperationException.
- This description of Collection specifies that the order of the elements is important; some variants of Collection do not require this.
- Some variants restrict the class of the objects stored, e.g., only Comparable objects can be stored in certain collections.

The specification for the ArraySequence implementation of Collection is that ordering is important, modifications and duplicates are allowed, but nulls are not. This section is concerned with the coding of the first seven methods using an array. A later section discusses the Iterator interface so we can implement the `iterator` method. After that, we discuss implementations of the remaining six methods.

Implementing the ArraySequence constructors

The ArraySequence class implements the Collection class using a partially-filled array, very much like the WorkerList class in Chapter Seven. Specifically, each ArraySequence object has two instance variables `itsItem` and `itsSize`. The former is an array that is filled with Objects in components indexed 0 up to but not including the int value `itsSize`. The values in higher-indexed components are not relevant to the logic. One ArraySequence constructor creates an empty Collection; it is left as an exercise.

Another ArraySequence constructor has the name of a disk file as its parameter. It is to read all the values from the disk file one at a time and put them in its array, starting from index 0. The first question is, how large do you make the array? The file could have 15 lines in it or 15,000. A reasonable approach is to start with an array of moderate size, say 100, and then replace it by one twice as large each time the one you have fills up. You first have to open the disk file with the specified name:

```
BufferedReader file = new BufferedReader
                      (new FileReader (fileName));
```

Listing 15.2 The Collection interface

```
/** The Collection interface specifies 13 method headings in
 * addition to those inherited from Object, e.g., equals. */

public interface Collection
{
   /** Return the number of elements in this sequence. */
   public int size();

   /** Tell whether this sequence has no elements in it. */
   public boolean isEmpty();

   /** Tell whether ob is an element of this sequence. */
   public boolean contains (Object ob);

   /** Tell whether every element of that sequence is
    *  somewhere in this sequence. */
   public boolean containsAll (Collection that);


   /** Return an array filled with the elements of this
    *  sequence in the same order. */
   public Object[] toArray();

   /** The same as the above if array.length < this.size().
    *  Otherwise fill the given array with the elements of the
    *  sequence in the same order and put a null value after
    *  those values if it will fit. */
   public Object[] toArray (Object[] array);

   /** Tell whether the two sequences have the same elements
    *  in the same order. */
   public boolean equals (Object ob);


   /** Return an iterator that goes through the elements of
    *  this sequence in an established order. */
   public Iterator iterator();


   //  The remaining six, to be described fully in Listing 15.10,
   //  are coded as follows when the Collection is unmodifiable:
   //  "throw new java.lang.UnsupportedOperationException();"
   public void clear();
   public boolean add (Object ob);
   public boolean addAll (Collection that);
   public boolean remove (Object ob);
   public boolean removeAll (Collection that);
   public boolean retainAll (Collection that);
}
```

Now you read one line at a time from the file into a String variable named perhaps `s`. When the value of `s` is null, you are at the end of the file and you can stop. Otherwise you copy `s` into the next available component `itsItem[itsSize]` and then increment `itsSize`. The coding for this constructor is in the upper part of Listing 15.3 (see next page).

Listing 15.3 The ArraySequence class of objects, parts left as exercises

```
import java.io.*;
import java.util.*;

public class ArraySequence implements Collection
{
   private Object[] itsItem;
   private int itsSize = 0;


   public ArraySequence (String fileName) throws IOException
   {  BufferedReader file = new BufferedReader                  //1
                          (new FileReader (fileName));          //2
      itsItem = new Object[100];                                //3
      String s = file.readLine();                               //4
      while (s != null)                                         //5
      {  if (itsSize == itsItem.length)                         //6
            itsItem = copyOf (itsItem, 2);                      //7
         itsItem[itsSize] = s;                                  //8
         itsSize++;                                             //9
         s = file.readLine();                                   //10
      }                                                         //11
   }  //======================

   private Object[] copyOf (Object[] given, int big)
   {  Object[] valueToReturn = new Object [given.length * big];
      for (int k = 0;  k < given.length;  k++)                  //13
         valueToReturn[k] = given[k];                           //14
      return valueToReturn;                                     //15
   }  //======================


   public ArraySequence (ArraySequence that)
   {  this.itsItem = copyOf (that.itsItem, 1);                  //16
      this.itsSize = that.itsSize;                              //17
   }  //======================


   public boolean equals (Object ob)
   {  if ( ! (ob instanceof ArraySequence))                     //18
         return false;                                          //19
      ArraySequence that = (ArraySequence) ob;                  //20
      if (this.itsSize != that.itsSize)                         //21
         return false;                                          //22
      for (int k = 0;  k < that.itsSize;  k++)                  //23
      {  if ( ! this.itsItem[k].equals (that.itsItem[k]))       //24
            return false;                                       //25
      }                                                         //26
      return true;                                              //27
   }  //======================
}
```

Another ArraySequence constructor just requires that you make the new `itsItem` an exact copy of the given one (the private `copyOf` method can be used for this as well as for doubling the size of an array, since it is written with an int parameter to specify whether the new array is twice the size of the old one or not). Then you record the size of the newly-constructed array. The coding is in the middle part of Listing 15.3.

Implementing the equals method

The `equals` method tests whether a given ArraySequence parameter has the same elements in the same order as the executor. The parameter is declared to be of type Object, not ArraySequence, because you want this `equals` method to override the one for the Object class. So you first check that the parameter is in fact a ArraySequence kind of Object. The condition `x instanceof Y` tells whether `x` is an object whose class is Y or extends Y or implements Y. This condition is false when `x` is null.

The accompanying design block is a reasonable plan for solving this problem. The coding is in the lower part of Listing 15.3. The other methods of the ArraySequence class are left for exercises.

STRUCTURED NATURAL LANGUAGE DESIGN for equals

1. If the parameter is not a ArraySequence kind of object then...
   The parameter is not equal to the executor.
2. Otherwise, if the parameter does not have the same number of elements as the executor then...
   The parameter is not equal to the executor.
3. Otherwise, for each element of the parameter do the following...
   If the current element of the parameter is not equal to the corresponding element of the executor then...
   The parameter is not equal to the executor.
4. The parameter is equal to the executor if you get to this point in the logic.

It is standard procedure to have a constructor that has a Collection parameter, but you cannot write such a method until later in this chapter, when you learn what an Iterator is.

Exercise 15.1 Write the ArraySequence method `public int size()`.
Exercise 15.2 Write the ArraySequence method `public boolean isEmpty()`.
Exercise 15.3 Write the ArraySequence method `public boolean contains (Object ob)`.
Exercise 15.4 Write the ArraySequence constructor `public ArraySequence()` that creates an empty Collection.
Exercise 15.5 Write the ArraySequence method `public Object[] toArray()`.
Exercise 15.6* Write the ArraySequence method `public boolean containsAll (Collection that)`. You may throw an Exception if `that` is not a ArraySequence.
Exercise 15.7* Write the ArraySequence method `public Object[] toArray (Object[] array)`.

15.3 Linked Lists With A Nested Private Node Class

You may define one class X inside the body of another class with the modifier `static` in X's heading. This makes X a **nested class** of the other class. We define the NodeSequence class with two constructors analogous to those of ArraySequence, and with all the Collection methods, but we implement it as a **linked list of Nodes** with a nested private Node class. That is, we define the Node class within the body of the NodeSequence class. This keeps outside classes from directly changing the value in a Node belonging to a NodeSequence object.

The Node class is defined in Listing 15.4 (see next page). A Node object stores two pieces of information: a reference to a single piece of data (of type Object) and a reference to another Node object. A Node object's data is referenced by `itsData` and the Node that comes next after it in the linked list is referenced by `itsNext`.

Listing 15.4 The Node class

```
private static class Node
    // nested in the NodeSequence class
{
   public Object itsData;
   public Node itsNext;


   public Node (Object data, Node next)
   {  itsData = data;
      itsNext = next;
   }
}  //=====================
```

If for instance `p` refers to a particular Node, then `p.itsData` is the information at that position in the list and `p.itsNext` is the Node containing the information at the following position in the list. If, however, `p.itsData` is the last information in the list, then we normally set `p.itsNext` to null to indicate this.

The natural Node constructor is defined and no other Node methods. Since this is a private class in the NodeSequence class and it has public instance variables, the NodeSequence class can access the instance variables directly. However, the principle of encapsulation is not violated since no class outside the NodeSequence class can refer to the instance variables of a class that is declared privately inside another class. Note: Chapter Fourteen defined a Node class outside of any class and provided public methods to control access to its private variables. That way of defining the Node class is not better or worse than this one, just different. Both approaches maintain encapsulation.

Nodes in a NodeSequence object

A NodeSequence object will have an instance variable `itsFirst` for the first Node object on its list. Suppose a particular NodeSequence object has a linked list of two or more Nodes. Coding to print the two elements at the beginning of this list is as follows:

```
System.out.println (itsFirst.itsData.toString());
System.out.println (itsFirst.itsNext.itsData.toString());
```

Coding to add a new element "long" between the first and second Nodes could be as follows (illustrated in Figure 15.2). Actually, this coding will work even if only one Node is in the linked list, but it will throw a NullPointerException if `itsFirst` has the value null:

```
Node newNode = new Node ("long", itsFirst.itsNext);
itsFirst.itsNext = newNode;
```

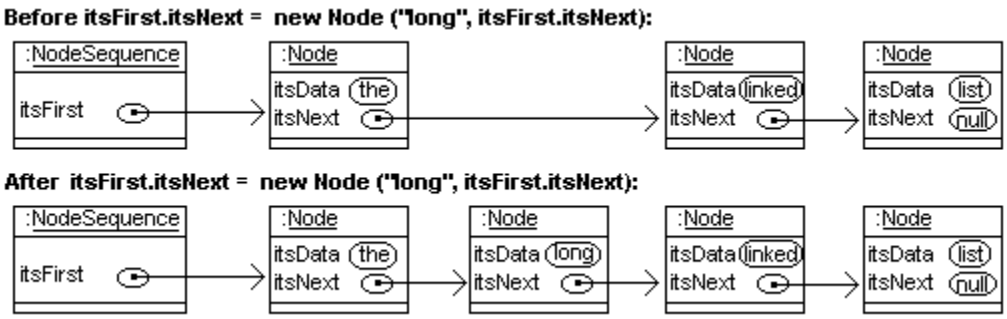

Figure 15.2 Inserting a new Node into a sequence of Nodes

If you want to add a new element named `givenData` at the front of the list, you could use the following coding, which will work even if no Node at all is in the linked list:

```
itsFirst = new Node (givenData, itsFirst);
```

Looping through a linked list

You know that the standard way you go through all the components in a partially-filled array one at a time, processing each one's data, is as follows:

```
for (int k = 0;  k < itsSize;  k++)
   processData (itsItem[k]);
```

This is precisely analogous to the standard way you go through all the nodes in a linked list one at a time, processing each one's data. Figure 15.3 shows the parallelism with the following coding:

```
for (Node p = itsFirst;  p != null;  p = p.itsNext)
   processData (p.itsData);
```

Components in an array	**Nodes in a linked list**
`k` indicates the current component	`p` indicates the current node
`k = 0` selects the first component	`p = itsFirst` selects the first node
`k == itsSize` if no more components	`p == null` if no more nodes
`k++` moves the indicator to the next component	`p = p.itsNext` moves the indicator to the next node
`itsItem[k]` is the data `k` indicates	`p.itsData` is the data `p` indicates

Figure 15.3 Parallelism of loops through arrays and loops through linked lists

The coding for the `contains` method in the ArraySequence array implementation is the following (as you saw in an exercise):

```
for (int k = 0;  k < itsSize;  k++)
{  if (itsItem[k].equals (ob))
      return true;
}
return false;
```

It follows that the coding for the `contains` method in the NodeSequence linked list implementation is the analogous logic you can see in the upper part of Listing 15.5 (see next page). Compare it piece-by-piece with the array form. This is a form of the Some-A-are-B looping action: Return `true` as soon as you see a good one, return `false` after you looked everywhere and saw nothing but bad ones.

The `containsAll` method requires a form of the All-A-are-B looping action: Return `false` as soon as you see a bad one, return `true` after you looked everywhere and saw nothing but good ones. That logic is in the middle part of Listing 15.5. It throws a ClassCastException if the parameter is not a NodeSequence. This is unavoidable for now, since you do not know iterators, but it is fixed in an exercise in Section 15.6.

The `size` method tells how many elements the sequence has. The NodeSequence coding has the standard loop heading and it uses the very common Count-cases looping action: Initialize a variable to 0 and then increment it once each time through the loop to see how many times the loop iterates. This logic is in the lower part of Listing 15.5.

Listing 15.5 The NodeSequence class of objects, part 1

```
/** A sequence of non-null values in a particular order. */

import java.io.*;
import java.util.*;

public class NodeSequence implements Collection
{
   private Node itsFirst = null;


   public NodeSequence()
   {  super();       // creates an empty NodeSequence        //1
   }  //======================


   public boolean contains (Object ob)
   {  for (Node p = itsFirst;  p != null;  p = p.itsNext)    //2
      {  if (p.itsData.equals (ob))                          //3
            return true;                                     //4
      }                                                      //5
      return false;                                          //6
   }  //======================


   public boolean containsAll (Collection that)
   {  for (Node p = ((NodeSequence) that).itsFirst;          //7
               p != null;  p = p.itsNext)                   //8
      {  if ( ! this.contains (p.itsData))                   //9
            return false;                                    //10
      }                                                      //11
      return true;                                           //12
   }  //======================


   public int size()
   {  int count = 0;                                         //13
      for (Node p = itsFirst;  p != null;  p = p.itsNext)    //14
         count++;                                            //15
      return count;                                          //16
   }  //======================
}
```

Exercise 15.8 Write the NodeSequence method `public boolean isEmpty()`.
Exercise 15.9 Write a NodeSequence method `public int howManyEqual (Object ob)`: The executor tells how many of its elements are equal to `ob`.
Exercise 15.10* Write a NodeSequence method `public boolean allAreStrings()`: The executor tells whether every element is a String value. Hint: Use the `instanceof` operator.
Exercise 15.11* Write a NodeSequence method `public Object removeFirst()`: The executor removes and returns the first element in its sequence. It returns null if its sequence is empty.
Exercise 15.12** Write a NodeSequence method `public void takeAway (Object ob)`: The executor removes all Objects from the front of the list down to but not including the first one that equals `ob`. Leave the NodeSequence empty if none are equal.

15.4 Implementing The Collection Interface With Linked Lists

The previous section began the coding of the NodeSequence class (Listing 15.5) with the "easy" methods. This section develops three more complex methods for that class. But first it is best to explicitly state the **internal invariant** for this class, i.e., the state of the NodeSequence object that every method preserves. It describes the connection between the user's abstract concept of the sequence of data values and the reality of Nodes.

Internal invariant for NodeSequences

- If the sequence does not contain any data values, `itsFirst` is null. Otherwise `itsFirst` refers to the first Node in a linked list of Nodes.
- For each Node `x` in that linked list, the value in `x.itsData` is one non-null data value in the abstract sequence of data values.
- The data values are in the linked list in the same order that they are in the abstract sequence of data values, with `itsFirst` containing the first one (if it exists).
- The Nodes in one NodeSequence are all different objects from those in any other.

Implementing the equals method

The coding for the `equals` method is not too much different from the coding for the ArraySequence `equals` method in the earlier Listing 15.3. You make sure the parameter actually is a NodeSequence and then you cast it to make the coding easier to read and faster to execute. You should not check the sizes to see if they are equal, because that takes a significant amount of time with linked lists (by contrast, each ArraySequence object knows its size). Next you go through the list of all values in the parameter one at a time. Since the coding in the ArraySequence method is

```
for (int k = 0;  k < that.itsSize;  k++)
{  if ( ! this.itsItem[k].equals (that.itsItem[k]))
      return false;
}
```

the analogous coding for the NodeSequence method should have two Node variables that progress through the two linked lists in tandem, something like this:

```
Node p = this.itsFirst;
for (Node q = that.itsFirst;  q != null;  q = q.itsNext)
{  if ( ! p.itsData.equals (q.itsData))
      return false;
   p = p.itsNext;
}
```

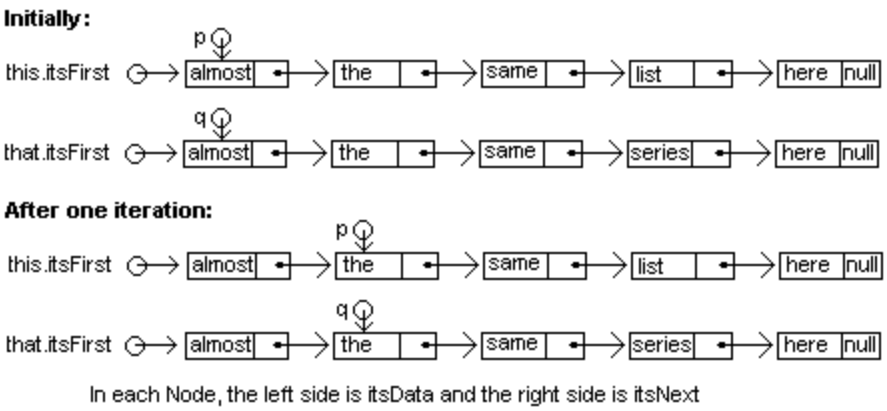

Figure 15.4 Two stages of execution of the equals method

Figure 15.4 shows how the two lists and these two position indicators look. However, you have not checked that the sizes of the two lists match, so it is possible that `p` could become null while the loop executes, which would throw a NullPointerException. You need a crash-guard; have the if-statement check `p == null` and, if so, return `false` without trying to evaluate `p.itsData`.

This reasoning leads to the coding in the upper part of Listing 15.6. Note that successful completion of the loop does not guarantee equality; you have to verify that the executor's list ran out at the same time as the parameter's list (verified in line 10).

Listing 15.6 The NodeSequence class of objects, part 2

```
// public class NodeSequence, 3 more of the methods

   public boolean equals (Object ob)
   {  if ( ! (ob instanceof NodeSequence))                    //1
         return false;                                        //2
      NodeSequence that = (NodeSequence) ob;                  //3
      Node p = this.itsFirst;                                 //4
      for (Node q = that.itsFirst; q != null; q = q.itsNext) //5
      {  if (p == null || ! p.itsData.equals (q.itsData))     //6
            return false;                                     //7
         p = p.itsNext;                                       //8
      }                                                       //9
      return p == null;                                       //10
   }  //=====================


   public NodeSequence (String fileName) throws IOException
   {  BufferedReader file = new BufferedReader                //11
                          (new FileReader (fileName));        //12
      String s = file.readLine();                             //13
      if (s != null)                                          //14
      {  this.itsFirst = new Node (s, null);                  //15
         Node previous = this.itsFirst;                       //16
         s = file.readLine();                                 //17
         while (s != null)                                    //18
         {  previous.itsNext = new Node (s, null);            //19
            previous = previous.itsNext;                      //20
            s = file.readLine();                              //21
         }                                                    //22
      }                                                       //23
   }  //=====================


   public NodeSequence (NodeSequence that)
   {  if (that.itsFirst != null)                              //24
      {  this.itsFirst = new Node (that.itsFirst.itsData, null);
         Node previous = this.itsFirst;                       //26
         for (Node p = that.itsFirst.itsNext;                 //27
                   p != null;  p = p.itsNext)                 //28
         {  previous.itsNext = new Node (p.itsData, null);    //29
            previous = previous.itsNext;                      //30
         }                                                    //31
      }                                                       //32
   }  //=====================
```

Implementing the NodeSequence constructor that uses a file

The logic for constructing a new NodeSequence object out of input from a disk file is only mildly different from the corresponding ArraySequence constructor in the earlier Listing 15.3. After you open the file and read the first line, you check it to see if you obtained null (line 14 of Listing 15.6). If so, the file is empty and so you simply leave `this.itsFirst` as null.

If the file is not empty, you put its first String value into a node and make that the first node on the NodeSequence object's node list (line 15). Now it gets a little tricky. You cannot add a node to the end of the linked list unless you change the `itsNext` value in the previous node. So you have to keep track of that previous node throughout the loop that reads String values in from the file.

You initialize a local variable `previous = this.itsFirst` (line 16), since that will be the node previous to the one you are going to add next. Now you write the standard loop for reading data from a file until you run out. The only difference is that the two statements

```
itsItem[itsSize] = s;
itsSize++;
```

in the body of the loop in the ArraySequence constructor are replaced by these two statements (lines 19-20), which do much the same thing:

```
previous.itsNext = new Node (s, null);
previous = previous.itsNext;
```

This coding is in the middle part of Listing 15.6. It illustrates an important design principle: If you have to process a sequence of values and the first value requires a different kind of processing from the rest of them, do not try to have a loop process all the values. Instead, process the first value before the loop and have the loop start its processing with the second value. If you were to try to write the coding for this constructor by having a while-statement but no if-statement, you would quickly see why this is a valuable principle.

Implementing the constructor that has a NodeSequence parameter

To create a NodeSequence object that has the same elements in the same order as a given NodeSequence parameter, you first verify that the parameter does not have an empty list. If it does not, you make the executor's first node a new Node object containing the data from the first node of the parameter. Initialize a local variable `previous = this.itsFirst` and enter a loop processing the second and all later elements on the parameter's list. This logic is in the lower part of Listing 15.6.

At each such element, link a new node after `previous` containing that element from the parameter and move `previous` on to that newly-constructed node (lines 29-30, basically the same as lines 19-20). Figure 15.5 (see next page) illustrates this coding. Note that again the first node on the executor's list must be processed differently from all other nodes, because the first node is the only one that does not have a previous node.

Exercise 15.13 How would you modify the `equals` method for the NodeSequence class to tell whether the executor is a "prefix" of a NodeSequence parameter, i.e., the elements of the executor are at the beginning of the parameter in the same order, but the parameter may have more elements besides those?
Exercise 15.14 (harder) Write the NodeSequence method `public Object[] toArray()`. Hint: Use `new Object[this.size()]`.

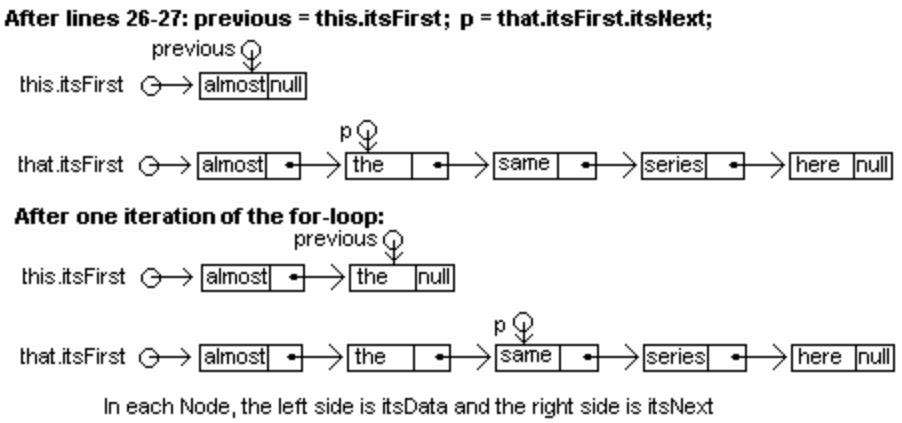

Figure 15.5 Two stages of execution of the second constructor

Exercise 15.15 (harder) Write a Node method `public void removeEvens()`: The executor removes every other Node from its linked list, i.e., the second, fourth, sixth, etc.
Exercise 15.16* Write another constructor for the NodeSequence class with an `Object[]` parameter: It constructs a Collection with the same elements in the same order as the array has, except it omits any null values that might be there. Hint: Work backwards from `given[given.length - 1]`.
Exercise 15.17* Write a NodeSequence method `public Collection reverse()`: The executor returns a new NodeSequence object with the same elements as the executor but in the opposite order.

15.5 Recursion With Linked Lists

The coding for each method in Listing 15.6 is lengthy and hard to follow. This is because the coding works with linked lists, which are naturally recursive structures, but it does not use recursion. A **naturally recursive structure** is a structure that can be implemented with a class X of objects that have instance variables of class X. Some people call these "self-referential objects" but technically they are not; `p.itsNext` refers to another Node object, not to p.

Deciding whether a String is a palindrome

Let us start with a recursion refresher, a method that uses recursion to tell whether a given String value reads the same forwards as backwards. Such a String is called a palindrome; an example is what Napoleon is rumored to have said: "able was I ere I saw elba". There are two kinds of palindromes, trivial and not:

- Any string that only has one character, or no characters at all, is trivially a palindrome.
- Any string with two or more characters is a palindrome when the first character is the same as the last and the smaller part in-between is a palindrome.

This logic is expressed quite naturally by the following independent method:

```
public static boolean isPalindrome (String s)  // independent
{  return s.length() <= 1
         || (s.charAt (0) == s.charAt (s.length() - 1)
           && isPalindrome (s.substring (1, s.length() - 1)));
}  //=====================
```

The equals method for NodeSequences

Two NodeSequence objects are equal if their linked lists of nodes have exactly the same data in the same order. So the `equals` method for NodeSequences is quite easy to work out if you put off most of the work to a separate `areEqual` method that tests whether two linked lists of nodes are equal:

```
public boolean equals (Object ob)
{  return ob instanceof NodeSequence
          && areEqual (this.itsFirst,
                       ((NodeSequence) ob).itsFirst);
}  //=====================
```

The `areEqual` method determines whether two linked lists have the same data values in the same order. There are two cases to consider, depending on whether one of the linked lists is empty or not:

- If either linked list is empty, then they are equal only if both are empty.
- If both are non-empty, then they are equal only if their first data values (in `itsData`) are equal to each other and their sublists (starting from `itsNext`) are also equal.

The coding for `areEqual` follows directly from those two cases:

```
private static boolean areEqual (Node one, Node two)
{  return (one == null  || two == null)  ?  one == two
          :  one.itsData.equals (two.itsData)
             && areEqual (one.itsNext, two.itsNext);
}  //=====================
```

See how much easier and more natural those two parts are together than the coding in Listing 15.6? Of course, you need a secondary method that recurses through the linked list of Nodes. The reason is that the `equals` method has a NodeSequence executor, and a NodeSequence is not a naturally recursive structure (since it does not have a NodeSequence instance variable). So `equals` calls the `areEqual` method for a pair of Node objects which store the naturally recursive linked lists.

The NodeSequence constructor using a file

The first NodeSequence constructor reads data from a disk file and creates a linked list from that sequence of data values. You first create the file, then you can call a recursive method to give you the linked list that the file provides:

```
public NodeSequence (String fileName) throws IOException
{  BufferedReader file = new BufferedReader
                         (new FileReader (fileName));
   this.itsFirst = readFrom (file);
}  //=====================
```

The `readFrom` method gets all the String values in the file and returns the linked list of Nodes containing those values in the order read. It first reads a single String value from the file and then sees which of two cases applies -- the String exists or not:

- If the String value does not exist, then return the empty linked list.
- If the String value exists, then return a non-empty linked list for which the first Node contains the String you just read as its data and the Node's sublist contains all the rest of the String values from the file.

The coding for `readFrom` following directly from those two cases:

```
private static Node readFrom (BufferedReader file)
                              throws IOException
{  String s = file.readLine();
   return s == null  ?  null : new Node (s, readFrom (file));
}  //======================
```

The NodeSequence constructor using another NodeSequence

To make a new NodeSequence that is a copy of another, you make a new linked list that is a copy of the given linked list:

```
public NodeSequence (NodeSequence given)
{  this.itsFirst = copyList (given.itsFirst);
}  //======================
```

This calls a private `copyList` method whose job is to return a copy of the given linked list. This recursive method sees which of two cases applies:

- If the list to copy is empty, then return an empty list as its copy.
- Otherwise, return a non-empty linked list for which the first Node contains the data in the first Node of the given linked list and the Node's sublist contains all the rest of the data values.

The coding for `copyList` following directly from those two cases:

```
private static Node copyList (Node toCopy)
{  return toCopy == null ? null : new Node (toCopy.itsData,
                                            copyList (toCopy.itsNext));
}  //======================
```

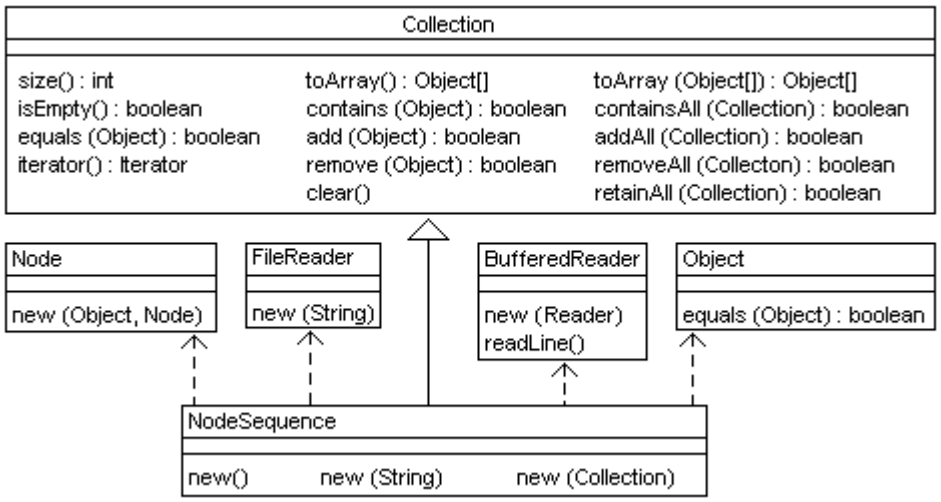

Figure 15.6 UML class diagram for the NodeSequence class

Exercise 15.18 Rewrite the `contains` method of Listing 15.5 by calling on a private recursive method with a Node parameter. Have only one statement in each method.
Exercise 15.19 (harder) Write a recursive Node method `public void removeEvens()`: The executor removes even-numbered Nodes from the linked list, i.e., the second, fourth, sixth, eighth, etc.
Exercise 15.20* Write a recursive Node method `public void omit(Object ob)`: The executor removes all Nodes after itself that contain a data value equal to `ob`.

Part B Enrichment And Reinforcement

15.6 Implementing The Iterator Interface For An Array-Based Collection

Your client needs a program that lists all purchased items that are missing from the inventory, both of which are stored in Collection objects. For this task you need to have an Iterator object. The Collection interface prescribes a method for which `someCollection.iterator()` returns an Iterator object connected to the Collection.

Definition of an Iterator

An Iterator provides the values in a Collection one at a time in some order. If it is a Collection for which order is important, it should always return the values in the Collection's own order. A class satisfies the **Iterator interface** (in the `java.util` package) if it has the following three instance methods:

- `hasNext()` tells whether there is an element of the Collection that has not yet been provided by the Iterator.
- `next()` advances to the next element to be provided and returns it.
- `remove()` deletes from the Collection the element that was returned by the most recent call of `next()`. After removal, `next()` provides what it would have provided without the removal. Note: Many implementations of Iterator specify that no one is to call its `remove` method; so it throws an UnsupportedOperationException if you call it.

Naturally, `next()` throws an Exception if there is no additional element and `remove()` throws an Exception if `next()` has not yet been called. The application program in Listing 15.7 (see next page) shows how an Iterator object can be used. The same coding would also work if "ArraySequence" were replaced by "NodeSequence" throughout. It is important that the value that `it.next()` returns be assigned to a variable, since the one value is used twice. If you called `it.next()` twice in the body of the loop, it would give you the next two values, not the same value twice.

An Iterator acts as a non-pushable stack containing the elements in the Collection. `it.next()` corresponds to `stack.pop()`; it takes the next available element out of the stack and returns it. `it.hasNext()` corresponds to `! stack.isEmpty()`.

A private nested class

An Iterator object constructed for a ArraySequence object needs access to the private instance variables of that sequence. Encapsulation is maintained if you make this ArraySequenceIterator class a private nested class of ArraySequence, which means that no outside class can mention the name ArraySequenceIterator. It is declared inside the ArraySequence class with the following class heading:

```
private static class ArraySequenceIterator implements Iterator
```

The instance method in the ArraySequence class that produces an Iterator for outside classes to use can be coded as follows. An outside class (such as LostItems in Listing 15.7) that calls this method must store the object the method returns in an Iterator variable, not a ArraySequenceIterator variable, because the latter name is private:

```
public  Iterator  iterator()  // in ArraySequence
{  return new ArraySequenceIterator (this);
}  //=====================
```

Listing 15.7 The LostItems application program

```
public class LostItems
{
   /** List all purchased items not in inventory. */

   public static void main (String[] args)
   {  java.util.Collection inventory;                          //1
      java.util.Collection purchased;                          //2
      try                                                      //3
      {  inventory = new ArraySequence ("inven.dat");          //4
         purchased = new ArraySequence ("purch.dat");          //5
      }catch (java.io.IOException e)                           //6
      {  throw new RuntimeException ("A file is defective.");//7
      }                                                        //8

      if (inventory.equals (purchased))                        //9
         System.out.println ("a perfect match");               //10
      else if ( ! inventory.containsAll (purchased))           //11
      {  System.out.println ("Listing all values we lost:"); //12
         java.util.Iterator it = purchased.iterator();         //13
         while (it.hasNext())                                  //14
         {  Object data = it.next();                           //15
            if ( ! inventory.contains (data))                  //16
               System.out.println (data.toString());          //17
         }                                                     //18
      }                                                        //19
   }  //=====================
}
```

The ArraySequence object is passed as a parameter to this `iterator` method so that its Iterator object can refer to its partially-filled array and to the size of that array.

Array implementation of an Iterator

For the ArraySequenceIterator class, we choose to keep track of the current position of the Iterator in an int instance variable named `itsPos`. Each time `next()` is executed, we add 1 to the value of `itsPos` and then return the element in `itsItem[itsPos]`.

We need to keep track of whether calling `remove` is allowed. We can do this with a boolean variable `isRemovable`. When the Iterator object is first created, this variable is initialized to `false`. Whenever `next` is called, this variable is made `true`.

The very first time we execute `next()`, we should get `itsItem[0]`. It follows that `itsPos` must be initialized to -1 so that adding 1 to it puts it at 0. And of course, we cannot execute `next()` if there is no element in `itsItem[itsPos+1]`, i.e., if `itsPos+1` equals `itsSize`. The coding for the constructor and the `hasNext` and `next` methods is in the upper part of Listing 15.8 (see next page).

Internal invariant for ArraySequenceIterators

- The instance variable `itsSeq` is the ArraySequence it iterates through.
- The instance variable `itsPos` is the int such that `itsSeq.itsItem[itsPos+1]` contains the element that `next()` will return, except `next()` is illegal when `itsPos+1 == itsSeq.itsSize`.
- The instance variable `isRemovable` tells whether a call of `remove` is allowed.

Listing 15.8 The ArraySequenceIterator nested class of objects

```
// This class goes inside the ArraySequence class

private static class ArraySequenceIterator implements Iterator
{
   private int itsPos = -1;      // next() is itsItem[itsPos+1]
   private boolean isRemovable = false;
   private ArraySequence itsSeq;


   public ArraySequenceIterator (ArraySequence givenSequence)
   {  itsSeq = givenSequence;                                  //1
   }  //=====================


   /** Tell whether there is a next element to be returned. */

   public boolean hasNext()
   {  return itsPos + 1 < itsSeq.itsSize;                      //2
   }  //=====================


   /** Advance to the next object to be returned and return it.
    *  Throw NoSuchElementException if hasNext() is false. */

   public Object next()
   {  if ( ! hasNext())                                        //3
         throw new NoSuchElementException ("hasNext is false");
      isRemovable = true;                                      //5
      itsPos++;                                                //6
      return itsSeq.itsItem[itsPos];                           //7
   }  //=====================


   /** Remove the object that was just returned by next().
    *  Throw IllegalStateException if next() has never been
    *  called, or if next() has not been called since the
    *  most recent call of remove(). */

   public void remove()
   {  if ( ! isRemovable)                                      //8
         throw new IllegalStateException ("nothing to remove");
      for (int k = itsPos + 1;  k < itsSeq.itsSize;  k++)      //10
         itsSeq.itsItem[k - 1] = itsSeq.itsItem[k];            //11
      itsSeq.itsSize--;                                        //12
      itsPos--;                                                //13
      isRemovable = false;  // no remove twice in a row        //14
   }  //=====================
}
```

The Iterator interface specifies that a **NoSuchElementException** be thrown if there is no next element to return. This Exception class is in the `java.util` package. The way you throw a NoSuchElementException object is quite simple -- just execute the following statement (the phrase in quotes is whatever you choose):

```
   throw new NoSuchElementException ("hasNext is false");
```

The remove method for Iterators

The `remove` method deletes the element that was returned by the most recent execution of `next()`. That of course is impossible if `next` has not yet been called or if the element it returned has already been removed. In such cases you are to throw an **IllegalStateException** (in `java.lang`).

When `remove` is called, you remove the element in `itsSeq.itsItem[itsPos]` (shifting other values down one) and make a note that another immediate call of `remove` is now forbidden (you cannot remove what is already gone). The next call of `next` should return the first value that was shifted down. That means that you should decrement `itsPos` in preparation for the next call of `next`. This coding is in the lower part of Listing 15.8. Figure 15.7 should clarify what is going on.

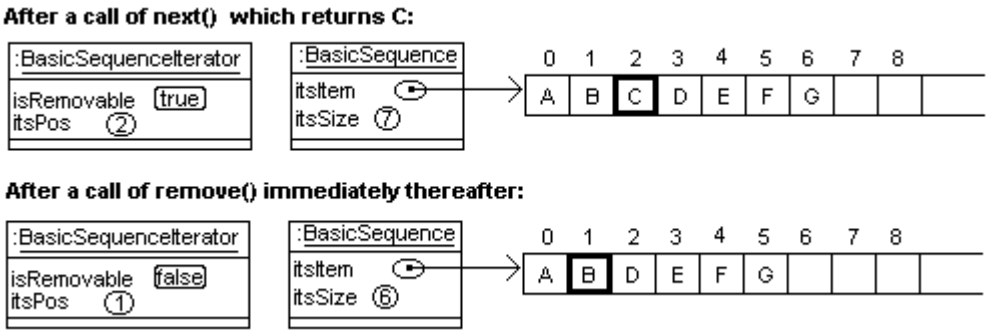

Figure 15.7 UML object diagrams for ArraySequenceIterator operations

Removal of `itsItem[itsPos]` requires shifting each element indexed `itsPos+1` and higher to the component indexed 1 less than itself. If you think about it a while, you will see why this shifting is easier if you work from `itsPos` toward `itsSize` rather than vice versa.

Exercise 15.21 What changes would you make in the `remove` method of Listing 15.8 if it specified that you are to return the Object that is removed?
Exercise 15.22 Write a generic Collection `containsAll` method, i.e., coding that works correctly for any Collection executor and for any Collection parameter. Hint: Use the parameter's Iterator to go through its elements one at a time and call `contains`.
Exercise 15.23 (harder) Modify Listing 15.7 to execute much faster on the precondition that every element of the `inventory` Collection is known to be in the `purchased` Collection and those elements are listed in the same order.
Exercise 15.24 (harder) If you studied inner classes in Chapter Ten, rewrite the ArraySequenceIterator class as an inner class. Also rewrite the `iterator` method.
Exercise 15.25* Rewrite the `remove` method in ArraySequenceIterator to explicitly state the executor wherever possible.
Exercise 15.26* Generalize the second ArraySequence constructor in Listing 15.3 to have a Collection parameter. Use its iterator to create the copy.
Exercise 15.27* Rewrite the NodeSequence `equals` method in Listing 15.6 to tell whether it has the same elements in the same order as its parameter, which can be any Collection object (i.e., do not return `false` just because it is not a NodeSequence). Hint: Instead of Node `q`, use an Iterator.
Exercise 15.28* Rewrite Listing 15.8 with a different internal invariant, namely, the element that `next()` returns is `itsItem[itsPos+1]` only when `isRemovable` is `true`. Otherwise `next()` returns `itsItem[itsPos]`.
Exercise 15.29* Draw the UML class diagram for the LostItems class.
Exercise 15.30** Rewrite the `remove` method in Listing 15.8 to shift elements down starting from the far end of the array, working from `itsSize-1` on down.

15.7 Implementing The Iterator Interface For A Linked-List-Based Collection

For the NodeSequence's Iterator class, you can use something analogous to what was just described for arrays: You keep track of the current position of the Iterator in a Node instance variable named `itsPos` . Each time you execute `next()`, you advance `itsPos` to the next Node and return the data in that Node. The primary problem occurs when you have to remove the data in the Node that `itsPos` refers to.

The easiest way to do that is to make a note of the Node just before `itsPos`. That is the Node that `itsPos` advances from when `next()` is executed. You could record that information in an instance variable named `itsPrevious`, since it is the Node before the element that can be removed. If you set `itsPrevious` to be `itsPos` when `remove()` is not allowed, you do not need an extra boolean instance variable to keep track of that information. Then the coding for `next` would be the exact analog of its coding for the array implementation, to wit:

```
if ( ! hasNext())
   throw new NoSuchElementException ("hasNext is false");
itsPrevious = itsPos;
itsPos = itsPos.itsNext;
return itsPos.itsData;
```

But now you have another problem: The first node does not have a node before it, so what do you initialize `itsPos` to? This problem is easily fixed: Construction of an iterator creates a **dummy header node** that links to `itsFirst` and initializes `itsPos` to that dummy header node. So the status of the implementation, i.e., the internal invariant, will always be as follows. Compare this description with the internal invariant in the previous section. In particular, `itsPrevious` gives the information required to do the equivalent of `itsPos--`, while the condition `itsPrevious != itsPos` gives the same information as the value of `isRemovable`:

Internal invariant for NodeSequenceIterators

- The instance variable `itsSeq` is the NodeSequence it iterates through.
- The instance variable `itsPos` is the Node such that `itsPos.itsNext.itsData` always contains the element that `next()` will return when `next()` is legal.
- `next()` is illegal when `itsPos.itsNext` is null.
- The instance variable `itsPrevious` is the Node before `itsPos` if a call of `remove` is allowed, otherwise `itsPrevious` equals `itsPos`.

What if the sequence is modified during iteration through it? That would cause unpredictable results in some cases (unless done by the iterator's own `remove` method, so it can make allowance for the removal). So you should not do that. Sun standard implementations of iterators are **fail-fast**: If the list structure is modified by any method other than the iterator's own methods, and then any of the iterator's methods are called, the iterator throws a runtime Exception.

To implement this, you could have each Collection object count all additions and removals with `itsChangeCounter`. Then each iterator object notes the value of `itsChangeCounter` when the iterator is created, and each call of `next` verifies that `itsChangeCounter` has not changed or, if it has, throws an Exception. For simplicity, we leave this out in Iterator codings in this book.

The coding for the constructor, `hasNext`, and `next` is in the upper part of Listing 15.9 (see next page). You must also add the following method to the NodeSequence class:

```
   public  Iterator  iterator()  // in NodeSequence
   {  return new NodeSequenceIterator (this);
   }  //=====================
```

Listing 15.9 The NodeSequenceIterator nested class of objects

```
// This class goes inside the NodeSequence class

private static class NodeSequenceIterator implements Iterator
{
   private Node itsPos;       // next() is itsPos.itsNext.itsData
   private Node itsPrevious;  // == itsPos when remove disallowed
   private NodeSequence itsSeq;


   public NodeSequenceIterator (NodeSequence givenSequence)
   {  itsSeq = givenSequence;                                  //1
      itsPos = new Node (null, itsSeq.itsFirst);               //2
      itsPrevious = itsPos;     // signals no remove allowed //3
   }  //=====================

   public boolean hasNext()
   {  return itsPos.itsNext != null;                           //4
   }  //=====================

   public Object next()
   {  if ( ! hasNext())                                        //5
         throw new NoSuchElementException ("hasNext is false");
      itsPrevious = itsPos;                                    //7
      itsPos = itsPos.itsNext;  //so now itsPrevious!=itsPos //8
      return itsPos.itsData;                                   //9
   }  //=====================

   public void remove()
   {  if (itsPrevious == itsPos)                               //10
         throw new IllegalStateException ("nothing to remove");
      itsPrevious.itsNext = itsPos.itsNext;                    //12
      if (itsSeq.itsFirst == itsPos)                           //13
         itsSeq.itsFirst = itsPos.itsNext;                     //14
      itsPos = itsPrevious;     // signals no remove allowed //15
   }  //=====================
}
```

The remove method for NodeSequenceIterator

The `remove` coding is far simpler than it was for the array implementation. As the internal invariant indicates, as long as `itsPrevious` is not equal to `itsPos`, you can just link from `itsPrevious` around the node `itsPos` to delete it from the linked list.

However, there is a special case: If the node to be removed is the first node on the linked list, then you have to reset `itsSeq.itsFirst` to refer to the currently-second node on the linked list. Either way, you make `itsPos` be `itsPrevious`, to prevent an additional call of `remove`. The full logic is in the lower part of Listing 15.9. Figure 15.8 illustrates how things change for a call of `next` followed by a call of `remove`.

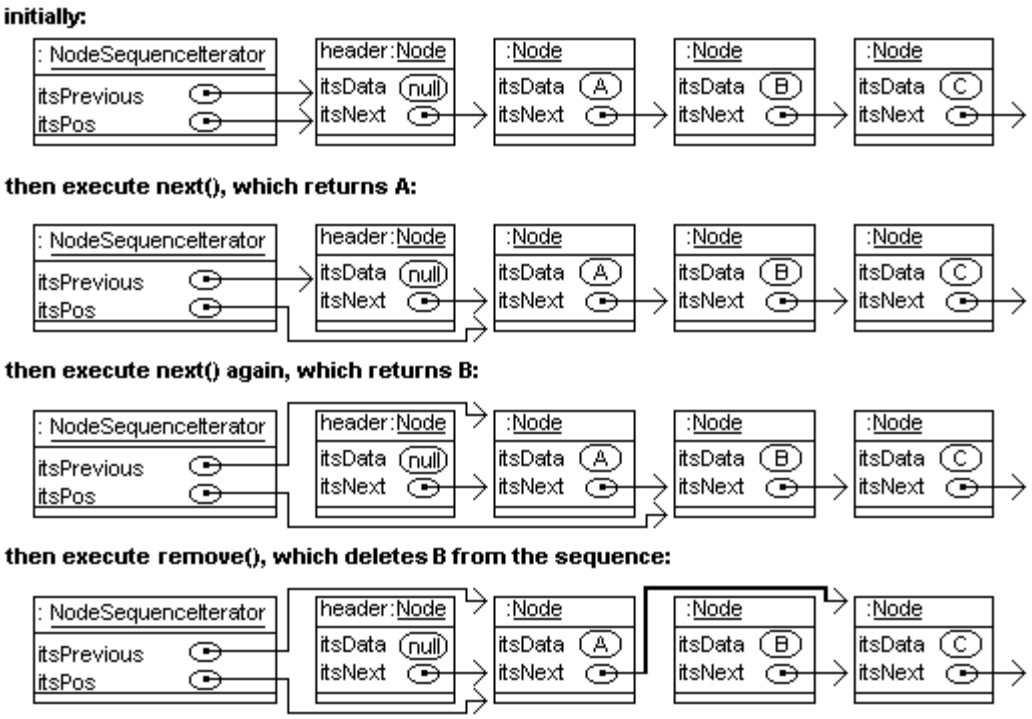

Figure 15.8 UML object diagrams for NodeSequenceIterator operations

Exercise 15.31 How would you change the NodeSequenceIterator class to have a `nextIndex` method which returns the index number of the element that would be returned by the next execution of `next()` (0 for the first, 1 for the second, etc.)?
Exercise 15.32 (harder) Generalize the second NodeSequence constructor in Listing 15.6 to have a Collection parameter. Use its iterator to create the copy recursively.
Exercise 15.33 (harder) What changes would you make in Listing 15.9 to initialize `itsPos` to null and thus avoid having a dummy header node?
Exercise 15.34* Revise Listing 15.9 for a different approach: Omit `itsPos` and keep `itsPrevious` with the same meaning. Have a boolean instance variable `isRemovable` to tell when you may remove a value. Adjust everything accordingly. Hint: `isRemovable` is true precisely when `itsPrevious != itsPos`.

15.8 Implementing A Modifiable Collection With Linked Lists

The Collection interface has six methods that modify the Collection. They are described in Listing 15.10 (see next page). This listing plus Listing 15.2 describe the Collection interface in its entirety.

Some implementations of the Collection interface do not allow modifications. The Java convention is that implementations that do not allow modifications are to have these six methods throw an **UnsupportedOperationException** (from `java.lang`), using e.g. the following statement (our two implementations will of course allow modifications):

```
throw new UnsupportedOperationException();
```

In fact, an implementation of Iterator is allowed to have the preceding statement as the body of the Iterator's `remove` method if the Collection is to be unmodifiable. In this sense, the six methods mentioned in Listing 15.10 are **optional operations** for a Collection and `remove` is an optional operation for an Iterator.

Listing 15.10 The Collection interface, part 2

```
/** A Collection class that does not guarantee maintaining a
 *  specific order ignores the ordering specifications here.
 *  A Collection class that does not allow null as an element
 *  throws a java.lang.IllegalArgumentException if you add null.
 *  A method that returns a boolean value returns true
 *  if and only if the method modifies the Collection. */

// public interface Collection, the rest of the 13 methods

   /** Make the Collection have no elements at all. */
   public void clear();

   /** Add the given Object at the end of this sequence.
    *  No effect if the Collection does not allow duplicates. */
   public boolean add (Object ob);

   /** Same as repeated add for each element in sequence. */
   public boolean addAll (Collection that);

   /** Remove the first instance of the given object from the
    *  Collection, if present. */
   public boolean remove (Object ob);

   /** Same as repeated remove for each element in that
    *  Collection. */
   public boolean removeAll (Collection that);

   /** Remove every element not in that Collection.  Keep the
    *  original order for those elements that remain. */
   public boolean retainAll (Collection that);
```

If you have a Collection implementation that does not maintain a particular order and that does not allow duplicates, the following expressions produce the set-union, set-intersection, and set-difference of Collections A and B, without changing A or B:

```
Collection union = new Collection (A).addAll (B);
Collection intersection = new Collection (A).retainAll (B);
Collection difference = new Collection (A).removeAll (B);
```

The **java.util.Set** interface in the standard library has the same methods as the Collection interface. The difference is that it does not allow duplicates (but it allows one null value) and the order of the elements is not necessarily guaranteed. The Sun library has the **java.util.HashSet** class that implements Set, as well as the **java.util.TreeSet** class that implements Set with elements sorted using `compareTo`.

Implementing the removeAll method

The `removeAll` method is not difficult if you just repeatedly call on the `remove` method. Specifically, you get an iterator from the Collection parameter to run down its sequence of elements and remove each one. Since you are charged with returning `true` when any change in the executor Collection is made, you can initialize a boolean variable to `false` and then change it to `true` any time one of the `remove` operations succeeds.

The coding for `removeAll` is in the upper part of Listing 15.11 (see next page). Note that it is **generic**: It could be put in any class that implements Collection and it will work right, assuming the other Collection methods it calls work right.

Listing 15.11 The NodeSequence class of objects, part 3

```
// public class NodeSequence, three more methods

   public boolean removeAll (Collection that)
   {  boolean changed = false;                                  //1
      Iterator it = that.iterator();                            //2
      while (it.hasNext())                                      //3
         changed = this.remove (it.next()) || changed;          //4
      return changed;                                           //5
   }  //=====================

   public boolean add (Object ob)
   {  if (ob == null)                                           //6
         throw new IllegalArgumentException ("no nulls allowed");
      if (itsFirst == null)                                     //8
         itsFirst = new Node (ob, null);                        //9
      else                                                      //10
         itsFirst.addLater (ob);                                //11
      return true;  // we accept duplicates of elements        //12
   }  //=====================

   public boolean remove (Object ob)
   {  if (itsFirst == null)                                     //13
         return false;                                          //14
      if (itsFirst.itsData.equals (ob))                         //15
      {  itsFirst = itsFirst.itsNext;                           //16
         return true;                                           //17
      }                                                         //18
      return itsFirst.removeLater (ob);                         //19
   }  //=====================


// private static class Node, 2 more methods

   public void addLater (Object ob)
   {  if (itsNext == null)                                      //20
         itsNext = new Node (ob, null);                         //21
      else                                                      //22
         itsNext.addLater (ob);                                 //23
   }  //=====================

   public boolean removeLater (Object ob)
   {  if (itsNext == null)                                      //24
         return false;                                          //25
      if (itsNext.itsData.equals (ob))                          //26
      {  itsNext = itsNext.itsNext;                             //27
         return true;                                           //28
      }                                                         //29
      return itsNext.removeLater (ob);                          //30
   }  //=====================
```

The `removeAll` coding uses two methods that not only take action but also return a value. Both are legitimized by the fact that they are part of the Sun standard library, but you can see that doing so much in one phrase can make the logic difficult to follow. Some people feel it would have been better if the Iterator class had been defined with two separate methods such as `getNext()` and `moveOn()` to take the place of the one `next()` method.

Implementing the NodeSequence add method

To add a given Object `ob` to the end of the linked list whose first node is `this.itsFirst`, you first must check that the Object parameter exists. If the parameter is null, you throw an **IllegalArgumentException** object (from `java.lang`). Then if `this.itsFirst` is null, you just create a new node to be the only node on the linked list, as follows:

```
this.itsFirst = new Node (ob, null);
```

If, on the other hand, the linked list already has at least one node, you need to add a node containing `ob` at the end of the linked list. Since the logic of `add` has already become rather complex, just ask the first node to do that job (i.e., call an instance method in the Node class).

The Node instance method you call could be named `addLater`. You ask a node on your linked list to add `ob` some place after it. There are two possibilities: Either the node X you ask is the last node, in which case X simply adds a new node after it containing `ob`, or else there is a node after X, in which case X asks that next node to add `ob` later. This coding is in the middle part of Listing 15.11. Figure 15.9 shows an example.

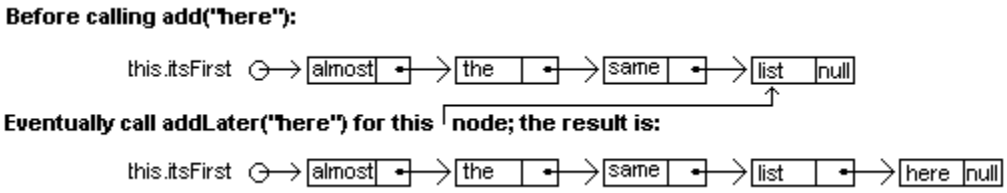

Figure 15.9

Designing the remove method

When you work out a complex logic, it usually helps to figure out what to do in the easy cases and postpone the hard cases to another method. For `remove`, if the executor has no nodes, do nothing. If it has nodes and the first node contains `ob`, delete that first node. Otherwise, ask that first node to do the removing and report back whether it was able to do so. This plan is formalized in the accompanying design block.

SNL DESIGN to remove ob

1. If the executor has no nodes then...
   Return `false` to indicate no change was made.
2. If the first node on the executor's linked list contains `ob` then...
   Make the currently-second node the new first node.
   Return `true` to indicate a change was made.
3. Ask the first node to remove `ob` from a node later in the linked list, if present.
4. Return `true` if `ob` was removed from a later node, otherwise return `false`.

What does that first node do when asked to remove `ob`? The same thing: If it has no node after it, it does nothing, but if the node after contains `ob`, it deletes the node after it, otherwise it asks the node after it to do the removing and report back whether it could do so. That gives the recursive `removeLater` method in the lower part of Listing 15.11. Figure 15.10 illustrates how this works.

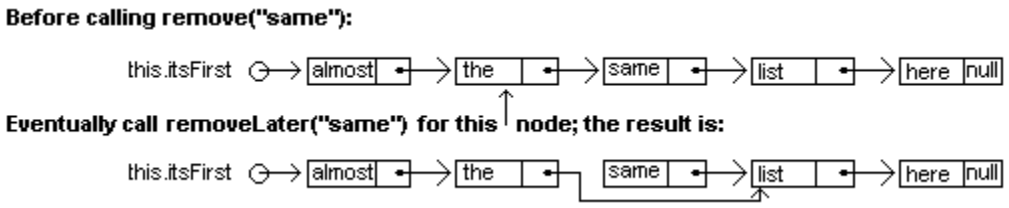

Figure 15.10 Effect of a call of remove("same")

If you compare the coding for `removeLater` with the coding for `remove`, you can see they are word-for-word the same except `itsNext` plays the role of `itsFirst`. And if you compare the coding for `addLater` with the middle four lines of `add`, they have the same resemblance.

Exercise 15.35 Write the NodeSequence method `public void clear()`.
Exercise 15.36 (harder) What changes would you make in Listing 15.11 to have `addLater` and `removeLater` be private class methods in the NodeSequence class?
Exercise 15.37 (harder) Write the NodeSequence method `public boolean retainAll (Collection that)` as follows: Repeatedly delete the executor's first node until you see that the Collection parameter contains the data in its first node (or the executor becomes empty). Then go through each node in the executor's linked list one at a time, deleting the nodes after it whose data is not in the Collection parameter.
Exercise 15.38* Write a NodeSequence method `public void doubleUp()`: The executor ends up with twice as many nodes, each element occurring twice in a row.
Exercise 15.39* Write the generic NodeSequence method `public boolean addAll (Collection that)` to repeatedly call the executor's `add` method.
Exercise 15.40* Write the ArraySequence method `public boolean add (Object ob)`.
Exercise 15.41* Rewrite the `add` method for NodeSequence without using recursion.
Exercise 15.42* Rewrite the `remove` method for NodeSequence without using recursion (use a for-loop).

15.9 Implementing The ListIterator Interface For A Linked List

An ordinary Iterator allows you to remove an element you come across in the iteration, but it does not allow you to add an element at a specific position in the sequence, nor does it allow you to replace one element by another at a specific position. For this capability you need a ListIterator kind of object. **ListIterator** is an interface in the `java.util` package that extends the Iterator interface. If `lit` is a ListIterator, then `lit.add(ob)` and `lit.set(ob)` are method calls that do just what you want.

However, the ListIterator interface requires two methods named `hasPrevious` and `previous`, which do the same as `hasNext` and `next`, respectively, except they go backwards in the list. This you do not want (how do you go backwards in a straightforward linked list?). Not to worry -- in accordance with the Java convention, you just include implementations of these methods with the standard coding that lets people know not to use them:

```
throw new UnsupportedOperationException();
```

It would be preferable if the Sun standard library offered a SequenceIterator interface that had only the `add` and `set` methods in addition to the Iterator methods. The SequenceIterator interface would extend Iterator and ListIterator would extend SequenceIterator. But apparently they did not think of that.

The ListIterator interface has the nine non-constructor methods named in Listing 15.12, which completes the implementation of NodeSequenceIterator. The ListIterator operations `nextIndex` and `previousIndex` are unsupported. They use zero-based indexing, e.g., if `next()` would return the fourth element of the list, then `nextIndex()` returns 3 and `previousIndex()` returns 2. We add another method in the NodeSequence and ArraySequence classes with the same coding as for the `iterator` method but with the following heading:

```
public ListIterator listIterator()
```

Listing 15.12 The NodeSequenceIterator nested class of objects, revised

```
private static class NodeSequenceIterator implements ListIterator
{
   private Node itsPos;       // next() is itsPos.itsNext.itsData
   private Node itsPrevious;  // == itsPos when remove disallowed
   private NodeSequence itsSeq;


   /** Replace the object last returned by next().  Throw an
    *  IllegalStateException if removal is not allowed. */

   public void set (Object ob)
   {  if (ob == null)
         throw new IllegalArgumentException ("no nulls allowed");
      if (itsPrevious == itsPos)
         throw new IllegalStateException ("nothing to replace");
      itsPos.itsData = ob;
   }  //=====================


   /** Add the given object just before the element that will be
    *  returned by next(), or at the end if hasNext() is false.
    *  Disallow set or remove until next is used again. */

   public void add (Object ob)
   {  if (ob == null)
         throw new IllegalArgumentException ("no nulls allowed");
      itsPos.itsNext = new Node (ob, itsPos.itsNext);
      itsPos = itsPos.itsNext;
      if (itsSeq.itsFirst == itsPos.itsNext)
         itsSeq.itsFirst = itsPos;
      itsPrevious = itsPos;   // so no one can remove it
   }  //=====================

   // hasNext(), next(), remove(), and NodeSequenceIterator()
   // are already implemented in Listing 15.9

   // The following four do not apply to sequences
   public boolean hasPrevious()  // is there a previous one?
   { throw new UnsupportedOperationException(); }
   public Object previous()      // return the one before
   { throw new UnsupportedOperationException(); }
   public int nextIndex()     // index of what next() returns
   { throw new UnsupportedOperationException(); }
   public int previousIndex() // index of what previous() returns
   { throw new UnsupportedOperationException(); }
}
```

Implementation of the set and add methods

The `set` and `remove` methods can only be called if `next` has been called with no intervening call of `add` or `remove`. That is, calling `next` makes a value available for removing or replacing, and calling `add` or `remove` leaves no value available for removing or replacing. So if `set` is allowed, you just replace the data in the node `itsPos.itsNext`. The coding for this is in the upper part of Listing 15.12.

The `add` method is allowed anytime; when `hasNext()` is `false`, you add at the end of the sequence. You first check that no one is trying to add null. Then you create a new node and link it in after the node referenced by `itsPos`. Then you can set `itsPos` to be that new node, since executing `next()` should return the element after the one just added. This coding is in the middle part of Listing 15.12. Figure 15.11 should clarify what is going on here.

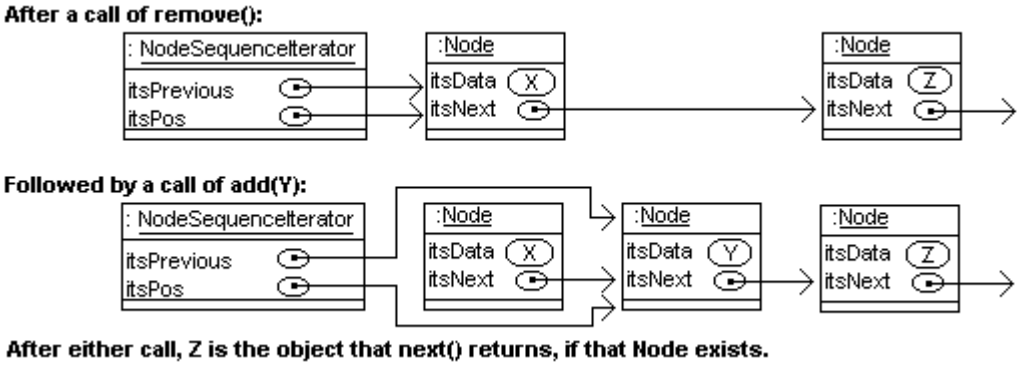

Figure 15.11 UML object diagram before and after a call of add(Y)

One special case occurs: If you added a node before the first node in the sequence, then you have to reset `itsSeq.itsFirst` to now indicate the newly-added node.

The header-node variant

An implementation of NodeSequenceIterator that simplifies the coding and lowers the execution time has the NodeSequence object create one dummy header node that all of its iterators use. This **header-node implementation** of a sequence is left as a major programming project. Since that dummy header will store in `itsNext` the first node that contains data, the NodeSequence object does not need to keep track of that data Node separately. So `itsFirst` can instead record the one dummy header node that all iterators use, as follows:

```
public NodeSequence()  // header-node implementation
{  itsFirst = new Node (null, null);
}  //=====================
```

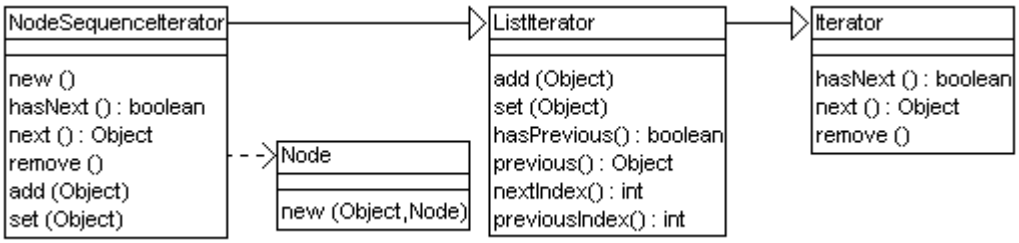

Figure 15.12 UML class diagram for NodeSequenceIterator

Implementing stacks and queues with sequences

You can easily implement a stack (described in Listing 14.1) as a subclass of any class that implements a modifiable Collection in which the order is important. The `peekTop()` call is to return the value on top of the stack, so it could be written as follows:

```
public Object peekTop()    // in a stack subclass
{  return iterator().next();
}  //======================
```

If the stack is empty, `peekTop()` throws an Exception, as it should. The `push(ob)` call is to add a value to the top of the stack, so it could be as follows:

```
public void push (Object ob)    // in a stack subclass
{  iterator().add (ob);
}  //======================
```

The `pop()` call is to remove the value on top of the stack; it is a bit more complex:

```
public Object pop()  // in a stack subclass
{  Iterator it = iterator();
   Object valueToReturn = it.next();
   it.remove();
   return valueToReturn;
}  //======================
```

A queue class (also described in Listing 14.1) can be implemented just as easily. The `dequeue` method is coded the same as `pop` and the `peekFront` method is coded the same as `peekTop`. The `isEmpty()` method for both stacks and queues is the one inherited from Collection. The `enqueue(ob)` call is just one statement:

```
public Object enqueue (Object ob)    // in a queue subclass
{  this.add (ob);   // put it at the end of the list
}  //======================
```

Exercise 15.43 Write the ArraySequenceIterator method `public void set (Object ob)`: It replace the current value by `ob`.
Exercise 15.44 Write the stack `push` and `pop` methods to be added to the NodeSequence class without using an iterator; just code them directly in terms of Nodes.
Exercise 15.45* Add another instance variable `itsSize` to the NodeSequence class. Modify everything that has to be modified in Listings 15.6 through 15.12 so that the `size` method can simply return `itsSize`.
Exercise 15.46* Write an independent method `public static Object findMax (Collection par)`: It finds the largest value in the Collection. Precondition: All elements of the Collection are mutually Comparable.
Exercise 15.47* Essay: Explain what can go wrong if one creates a ArraySequenceIterator, then calls its methods several times, then uses the `add` or `remove` methods in the ArraySequence class, and then calls more methods in ArraySequenceIterator.
Exercise 15.48** Essay: Same as the preceding exercise, but for Nodes instead.

15.10 Implementing The ListIterator Interface For A Doubly-Linked List

The ListIterator interface has methods to allow client classes to move backward one step in the sequence of values (described in the last four methods of Listing 15.12). This is highly inefficient with standard linked lists. It becomes quite easy if you define a new kind of node that records the node before it as well as the node after it, as shown in Listing 15.13. This Node class would be defined inside a class named TwoWaySequence that implements the Collection interface.

Listing 15.13 The Node class for the TwoWaySequence class

```
private static class Node // inside TwoWaySequence
{
   public Object itsData;
   public Node itsNext;
   public Node itsPrevious;


   public Node (Object data, Node next, Node previous)
   {  itsData = data;
      itsNext = next;
      itsPrevious = previous;
   }  //=====================
}
```

Implementing the TwoWaySequence class

The coding for the TwoWaySequence class is greatly simplified if we use a dummy header node. That is, a TwoWaySequence object has one instance variable `itsHead`, which has null for `itsData`. `itsHead.itsNext` is a Node containing the first element on the list, and the `itsNext` value for that node is a node containing the second element on the list. This continues to the last node containing data, whose `itsNext` value is the header node (so it is a **circular list**).

For every case in which `p.itsNext` is the Node `q`, it will be true that `q.itsPrevious` is `p`, and vice versa. This is called a **doubly-linked list**. If the sequence is empty, then its linked list consists only of the header node, and so `itsHead.itsNext` is `itsHead` and also `itsHead.itsPrevious` is `itsHead`.

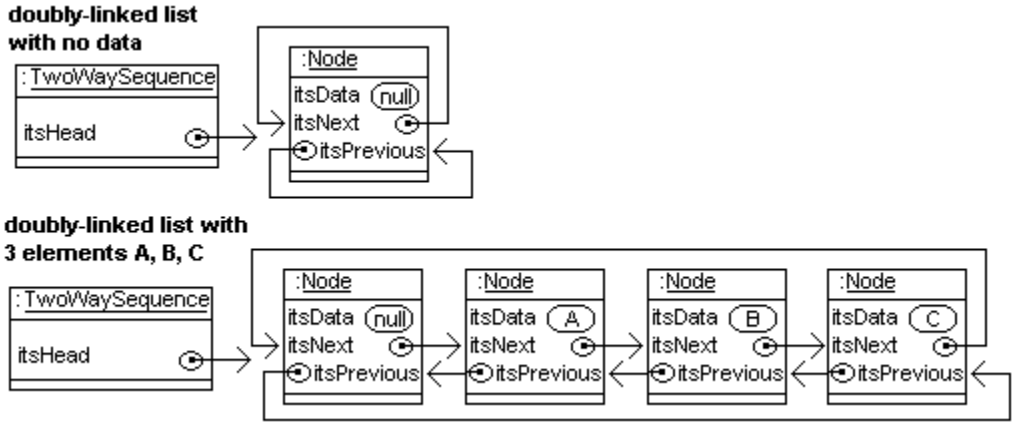

Figure 15.13 Two different doubly-linked lists

The `contains` method for a doubly-linked list requires the Some-A-are-B logic: You have a Node variable `p` go through each node that contains data (starting with the first one `p = itsHead.itsNext`) and return `true` if you see a node where `itsData` equals the parameter. But if you run out of nodes to look in (when `p == itsHead`), return `false`. This method and a constructor are in the upper part of Listing 15.14.

Listing 15.14 The TwoWaySequence class of objects, partial listing

```
import java.io.*;
import java.util.*;

public class TwoWaySequence implements Collection
{
   private final Node itsHead = new Node (null, null, null);


   public TwoWaySequence()
   {  itsHead.itsNext = itsHead;                                  //1
      itsHead.itsPrevious = itsHead;                              //2
   }  //=====================

   public boolean contains (Object ob)
   {  for (Node p = itsHead.itsNext; p != itsHead; p = p.itsNext)
      {  if (p.itsData.equals (ob))                               //4
            return true;                                          //5
      }                                                           //6
      return false;                                               //7
   }  //=====================


   public TwoWaySequence (Collection that)
   {  itsHead.itsNext = itsHead;                                  //8
      itsHead.itsPrevious = itsHead;                              //9
      Iterator it = that.iterator();                              //10
      while (it.hasNext())                                        //11
      {  Node last = itsHead.itsPrevious;                         //12
         last.itsNext = new Node (it.next(), itsHead, last);      //13
         itsHead.itsPrevious = last.itsNext;                      //14
      }                                                           //15
   }  //=====================


   public boolean equals (Object ob)
   {  if ( ! (ob instanceof Collection))                          //16
         return false;                                            //17
      Node p = this.itsHead.itsNext;                              //18
      Iterator it = ((Collection) ob).iterator();                 //19
      while (it.hasNext())                                        //20
      {  if (p == this.itsHead || ! p.itsData.equals (it.next()))
            return false;                                         //22
         p = p.itsNext;                                           //23
      }                                                           //24
      return p == this.itsHead;  // != means p has more than it
   }  //=====================

   public Iterator iterator()
   {  return new TwoWaySequenceIterator (this);                   //26
   }  //=====================
}
```

The constructor that makes a copy of a given Collection parameter runs through each element produced by the parameter's iterator, each time finding its `last` node (the one before `itsHead`) and creating a new node linked after that `last` node containing the iterator's element. This coding is in the middle part of Listing 15.14.

For the `equals` method, you first make sure that the parameter is in fact a Collection kind of object, otherwise you return `false`. You then get an iterator for the parameter and start with a local node variable `p` equal to the first node on the executor's list that contains data. Now verify that the iterator's `next()` value equals `p`'s data at each point. Advance with `p = p.itsNext` each time (the iterator automatically moves on). You also have to make sure that the iterator and `p` run out of values at the same time. This coding is in the lower part of Listing 15.14. The rest of the TwoWaySequence methods are left as exercises.

Implementing the ListIterator class

The doubly-linked list makes the ListIterator easier to implement. You can make `itsPos` always be the node before the one containing the data that `next()` will return. If `next()` is illegal, then `itsPos` is just before the header node. Execution of `previous()` always returns the element immediately before the one that `next()` would return (except if there is no element before it). Execution of `next()` immediately after a call of `previous` returns the same value that was returned by `previous()`.

When `remove` is called, the one removed is determined by the direction in which the iterator last moved, i.e., you remove the result of the most recent execution of `next()` or `previous()`. So you need to keep track of that direction, say in an int variable named `itsDirection`: +1 if `next()` was the most recent call, -1 if `previous()` was, 0 if removal is not even allowed.

A ListIterator must also be able to return the index (zero-based) of the element that a call of `previous` will return (-1 if there is no previous value) and the index of the element that a call of `next` will return (`size()` if there is none). The easiest way to do that is for the iterator to have another instance variable that keeps track of the index of the node to which it currently refers. Call it `itsIndex`. So `previousIndex()` returns `itsIndex`.

Internal invariant for TwoWaySequenceIterators

- The instance variable `itsPos` is the Node such that `itsPos.itsNext.itsData` always contains the element that `next()` will return, except `next()` is illegal when `itsPos.itsNext` is the header node (recognized by having `itsData == null`).
- The instance variable `itsIndex` is the zero-based index of the element that a call of `previous()` would return; it is -1 if `previous()` is illegal.
- The instance variable `itsDirection` is 0 if `remove()` is illegal, otherwise it is +1 or -1 depending on whether `next()` or `previous()` was the most recent method call.

The only thing the `hasNext` method has to do is to say whether the node after `itsPos` contains any data, i.e., is not the header node. The coding for `hasNext` and the constructor is in the upper part of Listing 15.15 (see next page).

The basic idea of the `previous` method is to back up `itsPos` by one node to be `itsPos.itsPrevious`, make a note that `itsDirection` is -1, and return the element in the original `itsPos` node. However, if `itsPos` was at the header node, a call of `itsPrevious` should throw a NoSuchElementException. The coding for the `previous` method is in the middle part of Listing 15.15.

Listing 15.15 The TwoWaySequenceIterator nested class of objects

```
private static class TwoWaySequenceIterator
                     implements ListIterator
{
   // Internal invariant: itsPos.itsNext is the header node if
   // hasNext() is false; otherwise, itsPos.itsNext is the node
   // containing the data that next() will return.

   private Node itsPos;
   private int itsDirection = 0;  // signals remove() not allowed
   private int itsIndex = -1;     // returned by previousIndex()


   public TwoWaySequenceIterator (TwoWaySequence given)
   {  itsPos = given.itsHead;                                  //1
   }  //=====================

   public boolean hasNext()
   {  return itsPos.itsNext.itsData != null;                   //2
   }  //=====================

   public Object previous()
   {  if (itsPos.itsData == null)                              //3
         throw new NoSuchElementException ("cannot back up");//4
      itsPos = itsPos.itsPrevious;                             //5
      itsDirection = -1;                                       //6
      itsIndex--;                                              //7
      return itsPos.itsNext.itsData;                           //8
   }  //=====================

   public void add (Object ob)
   {  if (ob == null)                                          //9
         throw new IllegalArgumentException ("no nulls allowed");
      itsPos = new Node (ob, itsPos.itsNext, itsPos);          //11
      itsPos.itsNext.itsPrevious = itsPos;                     //12
      itsPos.itsPrevious.itsNext = itsPos;                     //13
      itsDirection = 0;                                        //14
      itsIndex++;                                              //15
   }  //=====================

   // the first three of the following are left as exercises
   public Object next()              { return null; }
   public void remove()              {              }
   public void set (Object ob)       {              }
   public boolean hasPrevious()      { return itsIndex >= 0; }
   public int nextIndex()            { return itsIndex + 1;  }
   public int previousIndex()        { return itsIndex;      }
}
```

The `add` method for the TwoWaySequenceIterator can begin by creating a new node containing the given data. Then the new node is linked in after `itsPos` and before `itsPos.itsNext`, and `itsDirection` is set to 0. The value of `itsPos` has to become the new node, so `itsIndex` has to be incremented. The coding for the `add` method is in the lower part of Listing 15.15. The rest of the methods are left as exercises.

The List interface and the LinkedList implementation

The **List** interface in the Sun standard library is a sub-interface of Collection (which corresponds to a class extension). It has ten methods in addition to those of Collection, as follows. The first five specify the index where the action is to take place. These List methods throw Exceptions if the index values are out of range.

- `someList.get(indexInt)` returns the Object at that index.
- `someList.set(indexInt, someObject)` replaces the Object at that index by `someObject` and returns the Object that was replaced.
- `someList.add(indexInt, someObject)` inserts `someObject` at that index.
- `someList.remove(indexInt)` removes and returns the Object at that index.
- `someList.addAll(indexInt, someCollection)` adds the entire Collection at the specified index and returns `true`.
- `someList.indexOf(someObject)` returns the first index at which the Object occurs; it returns -1 if the Object is not in the list.
- `someList.lastIndexOf(someObject)` returns the last index instead.
- `someList.subList(fromInt, toInt)` returns a List containing the elements at index `fromInt` on up to but not including `toInt`.
- `someList.listIterator()` returns a new iterator, ready to start at the beginning of the list.
- `someList.listIterator(indexInt)` returns a new iterator ready to start at the specified index, so that next() produces get(indexInt).

The **java.util.LinkedList** class implements the List interface and has six additional methods that perform all of the operations of a stack or queue (and then some):

- `addFirst(someObject)`, `removeFirst()`, and `getFirst()` can be used for the stack operations `push(someObject)`, `pop()`, and `peekTop()`.
- `addLast(someObject)`, `removeLast()`, and `getLast()` do the same thing except at the rear of the LinkedList instead of at the front.

Exercise 15.49 Write the TwoWaySequence method public boolean `isEmpty()`.
Exercise 15.50 Write the TwoWaySequence method `public int size()`.
Exercise 15.51 (harder) Write the TwoWaySequence method `public boolean add (Object ob)` to add ob at the end of the sequence.
Exercise 15.52 (harder) Write the TwoWaySequenceIterator method `public Object next()`.
Exercise 15.53 (harder) Write the TwoWaySequenceIterator method `public void remove()`.
Exercise 15.54* Write the TwoWaySequence method `public void clear()`.
Exercise 15.55* Write the constructor for the TwoWaySequence class that has a String parameter naming the file from which String values are read.
Exercise 15.56* Write the TwoWaySequence method `public boolean remove (Object ob)`.
Exercise 15.57* Write the TwoWaySequenceIterator method `public void set (Object ob)`.
Exercise 15.58** Add the `push`, `pop`, and `peekTop` methods to TwoWaySequence to provide full stack capabilities. Code them to execute as fast as possible.
Exercise 15.59** Add the `addLast`, `removeLast`, and `getLast` methods to TwoWaySequence with the same function as those of LinkedList. Code them to execute as fast as possible.
Exercise 15.60** Write the NodeSequenceIterator method `public Object previous()` to be added to Listing 15.12 (no references to the previous node are stored in any node). You will need a loop to find the node before the current node.

*15.11 About AbstractList and AbstractCollection (*Sun Library)*

If you want to write an implementation of the Collection class, you have to code 13 methods plus some constructors. If you want to write an implementation of the List class, you have to code 23 methods plus some constructors. The AbstractCollection and AbstractList classes are intended to save you most of that trouble.

The AbstractCollection class

AbstractCollection is a class in `java.util` that implements the Collection interface. If you declare a class to be a subclass of AbstractCollection, you only have to write the `iterator` method and the `size` method (overriding those abstract methods in the AbstractCollection class). Your iterator must implement `hasNext` and `next`, though it can leave `remove` to throw an UnsupportedOperationException.

Generic coding for all the other Collection methods is provided for you. It uses the iterator method you provide. For instance, the coding for `contains` might be as follows (this generic Collection class allows null to be in the Collection). You may override this and the other pre-coded methods for efficiency:

```
public boolean contains (Object ob)
{  Iterator it = this.iterator();
   while (it.hasNext())
   {  if ((ob == null && it.next() == null)
             || (ob != null && ob.equals (it.next()))
         return true;
   }
   return false;
}  //======================
```

If you want your subclass of AbstractCollection to be modifiable, you have to code the `add` method for the AbstractCollection and the `remove` method for the Iterator.

The AbstractList class

AbstractList is a class in `java.util` that implements the List interface (specified in the preceding section). If you declare a class to be a subclass of AbstractList, you only have to write the `get` method and the `size` method (overriding those abstract methods in the AbstractList class). Coding for all the other List methods is provided for you, though the basic methods that modify the List object throw an UnsupportedOperationException. You may override any of the pre-coded methods for efficiency if you want.

If you want your subclass of AbstractList to be modifiable, you have to override one or more of `set(int,Object)`, `remove(int)`, and `add(int,Object)`, since these are the only methods that throw an UnsupportedOperationException. The AbstractList class provides a ListIterator implementation on top of the methods you provide.

The Sun standard library provides the **ArrayList** class, which is a complete implementation of List using an array. Use this class for situations where it is not worthwhile to develop your own implementation tailored to a particular piece of software. Its use is illustrated in Section 7.11. The iterator for ArrayList is fail-fast: If the ArrayList object is modified by any method other than the iterator's own `remove` or `add` method, and then any method is called for that iterator, the iterator throws a runtime Exception.

15.12 Review Of Chapter Fifteen

About the Java language:

- You may declare a class inside of another class X, called a **nested class**. If the word "static" appears before "class", this is no different from declaring it outside X except for visibility: Private variables of X are accessible in the nested class, and public variables of the nested class are accessible in X. The nested class itself is not accessible to classes outside of X if the nested class is declared to be a private member of the class. For statements and declarations inside X, the nested class shadows (supercedes) any outside class of the same name.
- A class of objects with an instance variable of that same class is called **naturally recursive**. Naturally recursive nodes are used to form a **linked list**. An extra node with no data at the beginning of the list is a **header node**; it simplifies the coding.
- A **circular list** has its last node link up to its first node.
- A **doubly-linked list** has each node link up to the preceding node as well as the node after it.

About the java.util.Collection interface:

- A **Collection** of **elements** may not allow duplicates (in which case it is a **java.util.Set** kind of object) or may not guarantee a particular ordering or may not allow null (and so throw a **java.lang.IllegalArgumentException** if you try to add null).
- The methods that modify a Collection may be **optional**, which means they may throw a **java.lang.UnsupportedOperationException**. A **sequence**, as the term is used in this book, is a Collection for which a particular order is guaranteed, duplicates are allowed, but null is not allowed.
- `someCollection.size()` returns the number of elements in the executor.
- `someCollection.isEmpty()` tells whether the executor has any elements.
- `someCollection.clear()` removes all elements from the executor.
- `someCollection.iterator()` returns an Iterator over the executor's elements.
- `someCollection.toArray()` returns an array containing the executor's elements.
- `someCollection.toArray(anArrayOfObjects)` returns an array containing the executor's elements. It will be the array parameter if the parameter has room (with null at the end if room), otherwise it will be a newly-created array.
- `someCollection.containsAll(aCollection)` tells whether the executor contains every element of the parameter.
- `someCollection.contains(anObject)` tells whether `anObject` is one of the executor's elements.
- `someCollection.add(anObject)` adds `anObject` to the Collection unless `anObject` is already in there and the Collection does not allow duplicates. It returns `true` if and only if the Collection changed, as do the other four methods listed below.
- `someCollection.remove(anObject)` deletes one instance of `anObject` from the executor unless `anObject` was not in there in the first place.
- `someCollection.addAll(aCollection)` in essence repeatedly executes `add` for each element of the parameter.
- `someCollection.removeAll(aCollection)` in essence repeatedly executes `remove` for each element of the parameter.
- `someCollection.retainAll(aCollection)` in essence repeatedly executes `remove` for each element of the executor that the parameter does not contain.

About the java.util.Iterator interface:

- `someIterator.next()` returns the next available element. A call of `next()` throws a **java.util.NoSuchElementException** if called when none is available. Repetition of `next()` calls produces each element of the Collection one time.
- `someIterator.hasNext()` tells whether `next` has more elements available.
- `someIterator.remove()` removes the element most recently returned by `next`. It throws a **java.lang.IllegalStateException** if that element has already been removed or if `next` has not been called. If removal is not allowed, it throws a java.lang.UnsupportedOperationException.

About the java.util.List interface:

- The List interface extends the Collection interface, adding the following ten methods:
- `someList.get(indexInt)` returns the Object at that index.
- `someList.set(indexInt, someObject)` puts `someObject` at that index and returns the Object that it replaces.
- `someList.add(indexInt, someObject)` inserts `someObject` at the specified index.
- `someList.remove(indexInt)` removes and returns the Object at that index.
- `someList.addAll(indexInt, someCollection)` adds the entire Collection at the specified index and returns `true`.
- `someList.indexOf(someObject)` returns the first index at which `someObject` occurs; it returns -1 if `someObject` is not in the list.
- `someList.lastIndexOf(someObject)` returns the last index at which `someObject` occurs; it returns -1 if `someObject` is not in the list..
- `someList.subList(fromInt, toInt)` returns a List containing the elements at index `fromInt` on up to but not including `toInt`.
- `someList.listIterator()` returns a new iterator, ready to start at the beginning of the list.
- `someList.listIterator(indexInt)` return a new iterator ready to start at the specified index, so that `next()` produces the value `get(indexInt)`.

About the java.util.ListIterator interface:

- The ListIterator interface extends the Iterator interface, adding the following six methods. The `remove` and `set` methods delete/replace the element most recently returned by `next` or `previous`, except they throw a java.lang.IllegalStateException if `next` has not yet been called, or if neither `next` nor `previous` has been called since the last call of `remove` or `add`.
- `someListIterator.add(someObject)` puts `someObject` just before the element that `next` would return, or at the end if `hasNext()` is `false`.
- `someListIterator.set(someObject)` puts `someObject` in place of the element that the most recent call of `next` or `previous` returned.
- `someListIterator.nextIndex()` returns the zero-based index of the element that a call of `next` would return; it returns the number of elements if `hasNext()` is `false`.
- `someListIterator.previous()` returns the element immediately before the element `next` would return; it returns the last element if `hasNext()` is `false`. It throws a java.util.NoSuchElementException if such an element does not exist.
- `someListIterator.hasPrevious()` tells whether `previous` has more elements available to be returned.
- `someListIterator.previousIndex()` returns 1 less than what `nextIndex()` returns.

Answers to Selected Exercises

15.1
```
public int size()
{       return itsSize;
}
```
15.2
```
public boolean isEmpty()
{       return itsSize == 0;
}
```
15.3
```
public boolean contains (Object ob)
{       for (int k = 0;  k < itsSize;  k++)
        {       if (itsItem[k].equals (ob))
                        return true;
        }
        return false;
}
```
15.4
```
public ArraySequence()
{       itsItem = new Object[100];   // an arbitrary choice of length
}
```
15.5
```
public Object[] toArray()
{       return copyOf (itsItem, 1);
}
```
15.8
```
public boolean isEmpty()
{       return itsFirst == null;
}
```
15.9
```
public int howManyEqual (Object ob)
{       int count = 0;
        for (Node p = this.itsFirst;  p != null;  p = p.itsNext)
        {       if (p.itsData.equals (ob))
                        count++;
        }
        return count;
}
```
15.13 Replace the if statement in the body of the for-statement by the following:
```
if (p == null)
        return true;
else if (! p.itsData.equals (q.itsData))
        return false;
```
15.14
```
public Object[] toArray()
{       Object[] valueToReturn = new Object [this.size()];
        int count = 0;
        for (Node p = this.itsFirst;  p != null;  p = p.itsNext)
        {       valueToReturn[count] = p.itsData;
                count++;
        }
        return valueToReturn;
}
```
15.15
```
public void removeEvens()     // in Node
{       for (Node p = this;  p != null && p.itsNext != null;  p = p.itsNext)
                p.itsNext = p.itsNext.itsNext;
}
```
15.18
```
public boolean contains (Object ob)
{       return isIn (itsFirst, ob);
}
private static boolean isIn (Node pos, Object ob)
{       return pos != null && (pos.itsData.equals (ob) || isIn (pos.itsNext, ob));
}
```
15.19
```
public void removeEvens()  // in Node
{       if (itsNext != null)
                itsNext = itsNext.itsNext;
        if (itsNext != null)   // which it could be after the previous statement is executed
                itsNext.removeEvens();
}
```

15.21 Put the following statement before the for-statement:
Object valueToReturn = itsItem[itsPos];
Put the following statement at the end of the method body:
return valueToReturn;

15.22
```
public boolean containsAll (Collection that)
{       Iterator it = that.iterator();
        while (it.hasNext())
        {       if ( ! this.contains (it.next()))
                        return false;
        }
        return true;
}
```

15.23 Replace the while statement by the following:
```
Iterator inven = inventory.iterator();
Object stopper = inven.hasNext()  ?  inven.next()  :  null;
while (it.hasNext())
{       Object data = it.next();
        if ( ! data.equals (stopper))
                System.out.println (data.toString());
        else
                stopper = inven.hasNext()  ?  inven.next()  :  null;
}
```

15.24 In Listing 15.8, remove the constructor, the declaration of itsSeq, and all uses of "itsSeq." in the coding of methods, since an inner class can refer to itsSize and itsItem directly. Also omit "static" from the class heading; that is what makes it an inner class.
```
public Iterator iterator()
{       return this.new ArraySequenceIterator();
}
```

15.31 Add another instance variable to the NodeSequenceIterator class, declared as follows:
private int itsIndex = 0;
Add one statement to the next method: itsIndex++;
Add one statement to the remove method: itsIndex--;
Add the following instance method:
```
public int nextIndex()
{       return itsIndex;
}
```

15.32
```
public NodeSequence (Collection that)
{       this.itsFirst = copyList (that.iterator());
}
private static Node copyList (Iterator it)   // uses recursion
{       return it.hasNext()  ?  new Node (it.next(), copyList (it))  :  null;
}
```

15.33 Initialize itsPos = null in the constructor. Replace the return statement in hasNext by:
return (itsPos == null) ? itsSeq.itsFirst != null : itsPos.itsNext != null;
Replace the next-to-last statement in next by:
itsPos = (itsPos == null) ? itsSeq.itsFirst : itsPos.itsNext;
Replace the last four lines in remove by the following (adding an "else"):
```
if (itsSeq.itsFirst  == itsPos)
        itsSeq.itsFirst = itsPos.itsNext;
else
        itsPrevious.itsNext = itsPos.itsNext;
itsPos = itsPrevious;
```

15.35
```
public void clear()
{       itsFirst = null;
}
```

15.36 1. Move addLater and removeLater to the NodeSequence class, changing the method headings as follows:
private static void addLater (Node current, Object ob)
private static boolean removeLater (Node current, Object ob)
2. Put "current." before each itsNext as a stand-alone variable in addLater or removeLater.
3. Replace the statements in add and remove that call these two methods by the following:
addLater (this.itsFirst, ob);
return removeLater (this.itsFirst, ob);
4. Replace the statements in addLater and removeLater that call these two methods by:
addLater (current.itsNext, ob);
return removeLater (current.itsNext, ob);

```
15.37   public boolean retainAll (Collection that)
        {   boolean valueToReturn = false;
            while (this.itsFirst != null && ! that.contains (this.itsFirst.itsData))
            {   this.itsFirst = this.itsFirst.itsNext;
                valueToReturn = true;
            }
            if (this.itsFirst == null)
                return valueToReturn;
            Node p = this.itsFirst;
            while (p.itsNext != null)
            {   if ( ! that.contains (p.itsNext.itsData))
                {   p.itsNext = p.itsNext.itsNext;
                    valueToReturn = true;
                }
                else
                    p = p.itsNext;
            }
            return valueToReturn;
        }
15.43   public void set (Object ob)
        {   if ( ! isRemovable)
                throw new IllegalStateException();
            itsSeq.itsItem[pos] = ob;
        }
15.44   public void push (Object ob)
        {   itsFirst = new Node (ob, itsFirst);
        }
        public Object pop ()
        {   if (itsFirst == null)
                throw new NoSuchElementException();
            Object valueToReturn = itsFirst.itsData;
            itsFirst = itsFirst.itsNext;
            return valueToReturn;
        }
15.49   public boolean isEmpty()
        {   return itsHead.itsNext == itsHead;
        }
15.50   public int size()
        {   int count = 0;
            for (Node p = itsHead.itsNext; p != itsHead;  p = p.itsNext)
                count++;
            return count;
        }
15.51   public boolean add (Object ob)
        {   if (ob == null)
                throw new IllegalArgumentException ("no nulls allowed");
            itsHead.itsPrevious = new Node (ob, itsHead, itsHead.itsPrevious);
            itsHead.itsPrevious.itsPrevious.itsNext = itsHead.itsPrevious;
            return true;
        }
15.52   public Object next()
        {   if (itsPos.itsNext.itsData == null)   // the sequence's header node
                throw new NoSuchElementException ("already at end");
            itsPos = itsPos.itsNext;
            itsDirection = 1;
            itsIndex++;
            return itsPos.itsData;
        }
15.53   public void remove()
        {   if (itsDirection == 0)
                throw new IllegalStateException ("cannot remove");
            if (itsDirection == 1)
            {   itsPos = itsPos.itsPrevious;  // so either way, itsPos is before the data to be removed
                itsIndex--;
            }
            itsDirection = 0;  // so it cannot be removed again
            itsPos.itsNext = itsPos.itsNext.itsNext;
            itsPos.itsNext.itsPrevious = itsPos;
        }
```