14 Stacks, Queues, And Linked Lists

Overview

This chapter requires that you have a solid understanding of arrays (Chapter Seven) and have studied Exceptions (Section 10.2) and interfaces (Sections 11.1-11.3).

- Section 14.1 discusses situations in which stack and queues can be useful. We define a stack to have the operations `push(Object)`, `pop()`, `peekTop()`, and `isEmpty()`. We define a queue to have the corresponding operations `enqueue(Object)`, `dequeue()`, `peekFront()`, and `isEmpty()`.
- Sections 14.2-14.3 present implementations of stacks and queues using arrays. The coding in these sections give you something to compare with the linked list coding developed next, so you better understand linked lists.
- Section 14.4 introduces the concept of linked lists and uses them to implement stacks and queues.
- Sections 14.5-14.7 implement a ListADT object as a linked list with trailer node, used to implement stacks and queues and to perform additional list operations recursively.
- Sections 14.8-14.9 discuss other variations: linked lists with header nodes, doubly-linked lists, circular linked lists, and even arrays of linked lists.

This chapter kills four birds with one stone: You develop a strong understanding of the stacks and queues, you strengthen your abilities in working with arrays, you develop a moderate facility with linked lists, and you learn to use recursion. But you only need study the first five sections to be able to go further in this book.

14.1 Applications Of Stacks And Queues

Your in-basket on your desk at work contains various jobs for you to do in the near future. Each time another job comes in, you put it on top of the pile. When you finish one job, you take another job from the pile to work on. Which one do you take?

You can model this situation in software with an **in-out data structure**, an object that contains several elements, allows you to add elements to it, and allows you to remove an element from it when you wish.

- If you always take the job on top, you are following a **Last-In-First-Out** principle (LIFO for short): Always take the job that has been in the pile for the shortest period of time. A data structure that implements this principle is called a **stack**. However, this principle may not be appropriate for a job-pile; some jobs may sit on the bottom of the stack for days.
- If you always take the job on the bottom, you are following a **First-In-First-Out** principle (FIFO for short): Always take the job that has been in the pile for the longest period of time. A data structure that implements this principle is called a **queue**. This principle guarantees that jobs do not sit overly long on the job-pile, but you may end up postponing very important jobs while working on trivial jobs that came in earlier.
- If you always take the job that has the highest priority, you are following a **Highest-Priority** principle: Always take the job in the pile that has the highest priority rating (according to whatever priority criterion you feel gives you the best chance of not getting grief from your boss). A data structure that implements this principle is called a **priority queue**. Priority queues will be discussed in Chapter Eighteen.

Another example where a queue would be important is in storing information about people who are waiting in line to buy tickets. Each new arrival goes to the rear of the queue (the `enqueue` operation), and the next ticket sold goes to the person who comes off the front of the queue (the `dequeue` operation).

If you are selling movie tickets, an ordinary queue is appropriate. But if they are tickets to fly on an airline, a priority queue might be better: You would want to serve people whose plane will be taking off within the hour before people who have longer to wait, and you would want to serve first-class customers before customers buying the cheap tickets.

Stock market application of stacks

An investor buys and sells shares of stock in various companies listed in the stock market. What if the person sells 120 shares of a particular company for $90 each at a time when the person owns 150 shares, of which 50 shares were bought in January at $60 each, 50 more in February at $70 each, and 50 more in March at $80 each? Which of those particular shares are being sold?

The federal tax laws require this investor to compute the profit and loss on a LIFO basis, i.e., the most recently purchased shares are sold first. So the profit on the sale is $500 on the 50 purchased in March, $1000 on the 50 purchased in February, and another $600 (20 shares at $30 each) on shares purchased in January. Therefore, an appropriate data structure for this situation is one stack of stock transactions for each of the companies that the person owns stock in. For this particular example of selling 120 shares, the software would remove from the top of the stack the first three transactions (the `pop` operation) and then add back the net transaction of 30 shares remaining at $60 each (the `push` operation).

Software to update the current stock holdings for the investor could be based on Transaction objects, each storing information about one stock transaction: company name, number of shares, price per share, and date of purchase. A text file could hold several lines of characters, each line giving the description of one transaction (a purchase of stock). The investor could start the program (which reads in the text file), enter several new buy-and-sell transactions, and then exit the program (which writes the updated information to a text file). A more detailed design is given in the accompanying design block.

STRUCTURED NATURAL LANGUAGE DESIGN for stock transactions

```
1.  Read the "transact.dat" file into an array of stacks, one stack per company.
2.  Repeat as long as the user has more transactions to process...
        2a.  Get the next transaction from the user (buy or sell).
        2b.  Modify the information in the array of stacks accordingly.
3.  Write the updated information to a file.
```

The left side of Figure 3.1 shows a sequence of `Push` and `Pop` operations for a stack, starting from an empty stack. The right side of the figure shows a sequence of `Enqueue` and `Dequeue` operations for a queue, starting from an empty queue.

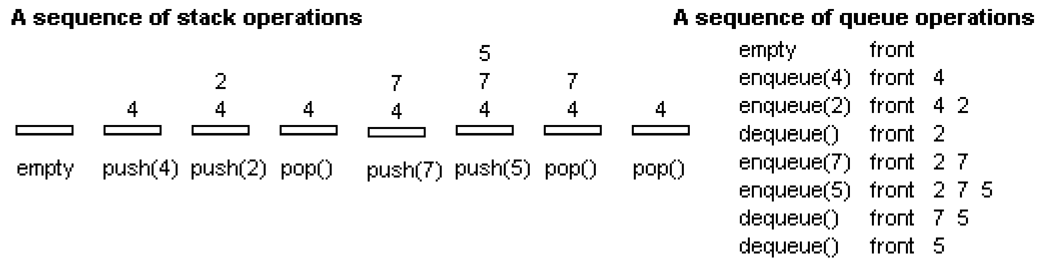


Figure 14.1 Effect of method calls for stacks and queues

Some other applications of stacks

You may need a program that reads in several lines of input, each containing what is supposed to be a properly-formed expression in algebra, and tells whether it is in fact legal. The expression could have several sets of grouping symbols of various kinds, `( )` and `[ ]` and `{ }`. Part of the parsing process is to make sure that these grouping symbols match up properly. The accompanying design block could be part of the logic for processing the algebraic expression.

STRUCTURED NATURAL LANGUAGE DESIGN for grouping symbols

```
1.  Get the next character on the input line; call it the currentChar.
2.  If the currentChar is one of '(', '[', or '{', then...
        Push the currentChar on the stack.
3.  Otherwise, if the currentChar is one of ')', ']', or '}', then...
        3a.  If the stack is empty then...
                Conclude that this is not a properly-formed expression.
        3b.  Pop the top character from the stack.
        3c.  If that is not the other half of the currentChar, then...
                Conclude that this is not a properly-formed expression.
4.  Otherwise [additional logic for other characters]...
```

A Java compiler uses a stack to handle method calls, which works roughly as follows (very roughly), assuming `calls` is the stack of method calls and an Activation object keeps track of information for one method call:

- `x = doStuff(y)` causes `calls.push (new Activation (arguments));`
- `z = 3` for a local variable `z` causes `calls.peekTop().setVariable(z, 3);`
- `return 5` causes `calls.pop(); x = 5.`

Stack and queue interfaces

We will develop several different implementations of stacks and queues in this chapter. We specify what is common to all by defining interfaces. An interface describes what operations the object can perform but does not give the coding. So it describes an **abstract data type**. We call these two interfaces **StackADT** and **QueueADT**. They are shown in Listing 14.1 (see next page). Each has an operation to add an element to the data structure (`push` and `enqueue`), an operation to remove an element (`pop` and `dequeue`, which return the element removed), and two query methods: an operation to see what would be removed (`peekTop` and `peekFront`), and an operation to tell whether the data structure has any elements to remove (`isEmpty`).

We later develop coding for ArrayQueue, an implementation of QueueADT, and ArrayStack, an implementation of StackADT. As an example of how their methods can be used, the following is an independent method that reverses the order of the elements on a given stack, using a queue for temporary storage (**independent** simply means that the method works right no matter what class it is in). The coding removes each value from the stack and puts it on the queue, with what was on top of the stack going to the front of the queue. Then it removes each value from the queue and puts it on the stack. So the value that was originally on top of the stack ends up on the bottom of the stack:

```
public static void reverse (StackADT stack)
{  QueueADT queue = new ArrayQueue();
   while ( ! stack.isEmpty())
      queue.enqueue (stack.pop());
   while ( ! queue.isEmpty())
      stack.push (queue.dequeue());
}  //======================
```

Listing 14.1 The StackADT and QueueADT interfaces

```
public interface StackADT            // not in the Sun library
{
   /** Tell whether the stack has no more elements. */

   public boolean isEmpty();


   /** Return the value that pop would give, without modifying
    *  the stack.  Throw an Exception if the stack is empty. */

   public Object peekTop();


   /** Remove and return the value that has been in the stack the
    *  least time.  Throw an Exception if the stack is empty. */

   public Object pop();


   /** Add the given value to the stack. */

   public void push (Object ob);
}
//##############################################################



public interface QueueADT            // not in the Sun library
{
   /** Tell whether the queue has no more elements. */

   public boolean isEmpty();


   /** Return the value that dequeue would give without modifying
    *  the queue.  Throw an Exception if the queue is empty. */

   public Object peekFront();


   /** Remove and return the value that has been in the queue the
    *  most time.  Throw an Exception if the queue is empty. */

   public Object dequeue();


   /** Add the given value to the queue. */

   public void enqueue (Object ob);
}
```

The rest of the examples in this section are not used elsewhere in this book, so you could skip them. But even if your instructor does not assign them for study, you should at least skim through them to get a better feeling for the various problems that stacks and queues can help you solve.

The classic Tower of Hanoi problem

You are given three stacks A, B, and C. Initially, B and C are empty, but A is not. Your job is to move the contents of A onto B without ever putting any object x on top of another object that was above x in the initial setup for A. Can you see how the following coding solves this problem, if it is called with n initially equal to A's size?

```
public void shift (int n, StackADT A, StackADT B, StackADT C)
{  if (n == 1)
      B.push (A.pop());
   else
   {  shift (n - 1, A, C, B);  // n-1 go from A onto C
      B.push (A.pop());
      shift (n - 1, C, B, A);  // n-1 go from C onto B
   }
}  //=====================
```

Anagrams

An interesting problem is to print out all the anagrams of a given word. That is, given an N-letter word with all letters different, print all the "words" you can form using each letter once, regardless of whether they are actual words in some language. The number of such rearrangements is N-factorial where N is the number of letters. The problem can be solved in several different ways, one of which uses one queue and one stack. We illustrate the process here and leave the coding as a major programming project.

Say the word is abcdefgh and you have just printed ehcgfdba. The very next word in alphabetical order that you can form with those eight letters is ehdabcfg (compare the two to see why). The way one iteration of the main loop of the process goes from a stack containing ehcgfdba to that same stack containing ehdabcfg is as follows:

Initially you have the letters on the stack in reverse order, and you have an empty queue:
1. Repeatedly pop a value from the stack and enqueue it on the queue, stopping as soon as you pop a value that comes alphabetically before the one you just enqueued. Call that value the "pivot".
2. Repeatedly dequeue a value and enqueue it, stopping as soon as you dequeue a value that comes alphabetically after the pivot. Push that value onto the stack.
3. Enqueue the pivot, then repeatedly dequeue a value and enqueue it, stopping as soon as you dequeue a value that is alphabetically before the pivot. Push it onto the stack.
4. Repeatedly dequeue a value from the queue and push it on the stack until the queue is empty.

Evaluating postfix expressions

Some calculators require you to enter arithmetic expressions in postfix notation. An **arithmetic postfix expression** is a sequence of numbers and operators `+ - * / %` where each operator is placed directly after the two values it is to operate on. Some examples of postfix expressions and the corresponding expressions in the normal **infix notation** that you are used to are as follows:

postfix notation	infix notation (fully parenthesized)
7 2 -	(7 - 2)
3 4 + 5 *	(3 + 4) * 5)
3 4 5 * +	(3 + (4 * 5))
3 7 + 8 2 / -	((3 + 7) - (8 / 2))
3 4 2 6 3 / - * +	(3 + (4 * (2 - (6 / 3))))

The reverse notation, where the operator comes directly before the two values it is to operate on, is called **prefix notation**. It is used in the Scheme programming language.

Assume that an input string containing an arithmetic postfix expression has been read and separated into numbers and operators stored on a queue in order. The numbers are stored as Integer objects (with an `intValue` method for getting the value) and the operators are stored as Character objects (with a `charValue` method for getting the value). Then the accompanying design block is logic for evaluating the expression. The method in the upper part of Listing 14.2 (see next page) applies this algorithm. It throws an Exception if the data values in the queue do not form a legal postfix expression.

STRUCTURED NATURAL LANGUAGE DESIGN for evaluating a postfix expression

```
1.  Create a stack for storing numbers not yet combined with other numbers.
2.  For each value you remove from the queue do...
        If it is a number then...
            Push it onto the stack.
        Otherwise, it is an operator, so...
            Pop the top two stack values and apply the operator to them.
            Push the result onto the stack.
3.  Return the one remaining number on the stack.
```

Converting infix notation to postfix notation

Modern compilers usually process the normal infix expressions you write in your programs to produce the corresponding postfix expressions in the compiled code. That makes it much easier for the runtime system to evaluate the expressions. The full evaluation algorithm is far too complex to present here. But a method that does the job for the special case of a fully parenthesized numeric expression is not too hard.

Assume that an input string containing a fully parenthesized normal infix expression has been read and separated into numbers and operators. They have been placed in a queue in the order they were read, stored as Integer objects and Character objects. Then a method for creating a new queue containing the corresponding postfix expression (suitable for input to the preceding method) is in the lower part of Listing 14.2. It puts all operators on a stack (line 32) and pops them when the expression is complete (line 30).

This independent method relies on two facts about postfix expressions: (1) Each operator must be moved to the point in the sequence where the right parenthesis that belongs to it occurs; (2) if several operators are "waiting" for their right parentheses, the next one to come along belongs to the most-recently-seen operator. You can verify these facts for the examples of postfix expressions shown earlier in this section.

Note: A <u>precondition for all exercises in this chapter</u> is that all stack, queue, and list parameters of methods are non-null.

Exercise 14.1 Write an independent method `public static Object removeSecond (StackADT stack)`: It removes and returns the element just below the top element if the stack has at least two elements, otherwise it simply returns null without modifying the stack.
Exercise 14.2 Compute the values of these two postfix expressions. Also write these two postfix expressions in ordinary algebraic notation without evaluating any of the parts:
(a) `12 10 3 5 + - /` (b) `5 4 - 3 2 - 1 - -`
Exercise 14.3 (harder) Write an independent method `public static void removeDownTo (StackADT stack, Object ob)`: It pops all values off the stack down to but not including the first element it sees that is equal to the second parameter. If none are equal, leave the stack empty. The `ob` parameter could be null.
Exercise 14.4 (harder) Write an independent method `public static Object removeSecond (QueueADT queue)`: It removes and returns the element just below the top element. Precondition: The queue has at least two elements. Hint: Create a new object totally different from any that could possibly be on the queue.

Listing 14.2 Independent methods for postfix expressions

```
public static int evaluatePostfix (QueueADT queue)
{  StackADT stack = new ArrayStack();                        //1
   while ( ! queue.isEmpty())                                //2
   {  Object data = queue.dequeue();                         //3
      if (data instanceof Integer)                           //4
         stack.push (data);                                  //5
      else                                                   //6
      {  char operator = ((Character) data).charValue();     //7
         int second = ((Integer) stack.pop()).intValue();    //8
         int first = ((Integer) stack.pop()).intValue();     //9
         if (operator == '+')                                //10
            stack.push (new Integer (first + second));       //11
         else if (operator == '-')                           //12
            stack.push (new Integer (first - second));       //13
         //etc.                                              //14
      }                                                      //15
   }                                                         //16
   int valueToReturn = ((Integer) stack.pop()).intValue();//17
   if ( ! stack.isEmpty())                                   //18
      throw new RuntimeException ("too many values");        //19
   return valueToReturn;                                     //20
}  //======================

public static QueueADT fromInfixToPostfix (QueueADT queue)
{  QueueADT postfix = new ArrayQueue();                      //21
   StackADT stack = new ArrayStack();                        //22
   while ( ! queue.isEmpty())                                //23
   {  Object data = queue.dequeue();                         //24
      if (data instanceof Integer)                           //25
         postfix.enqueue (data);                             //26
      else  // it is a parenthesis or an operator            //27
      {  char nonNumber = ((Character) data).charValue();    //28
         if (nonNumber == ')')                               //29
            postfix.enqueue (stack.pop());                   //30
         else if (nonNumber != '(')  // ignore left paren    //31
            stack.push (data);                               //32
      }                                                      //33
   }                                                         //34
   return postfix;                                           //35
}  //======================
```

Exercise 14.5* Write an independent method `public static void transfer (StackADT one, StackADT two)`: It transfers all elements in the first parameter onto the top of the second parameter, keeping the same order. So what was initially on top of `one` ends up on top of `two`. Hint: Use a third stack temporarily.

Exercise 14.6* Write an independent method `public static void reverse (QueueADT queue)`: It reverses the order of the elements on the queue.

Exercise 14.7** Write an independent method `public static void removeBelow (QueueADT queue, Object ob)`: It removes all values from the queue that come after the first element it sees that is equal to the second parameter. If none are equal, leave the queue as it was originally.

Exercise 14.8** Write an independent method named `interchange`: You have two StackADT parameters. All the items on each stack are to end up on the other stack in the same order, so that each stack has the items that the other stack had at the beginning. Use only one additional temporary stack.

14.2 Implementing Stacks With Arrays

Our **ArrayStack** implementation of StackADT stores values in a partially-filled array: We use an array `itsItem` of Objects and an int value `itsSize` that tells how many elements the stack has. We store the elements in indexes 0 through `itsSize-1`. The `pop` method tells the executor (i.e., the ArrayStack object carrying out the task) to decrement `itsSize` and then return the value at index `itsSize`. To `push` an element onto the stack, the executor puts the new element at index `itsSize` and then increments `itsSize` (just the opposite of `pop`). This logic is in Listing 14.3.

Removing an element or peeking at an element requires that there be an element in the data structure, so these methods first make sure that the stack is not empty. If it is empty, a runtime Exception is to be thrown. The obvious choice is an **IllegalStateException** (from `java.lang`). To throw the Exception, all you need do is execute the following statement. It immediately terminates execution of the method it is in:

```
   throw new IllegalStateException ("some explanatory message");
```

Listing 14.3 The ArrayStack class of objects

```
public class ArrayStack implements StackADT
{
   private Object[] itsItem = new Object [10];
   private int itsSize = 0;


   public boolean isEmpty()
   {  return itsSize == 0;
   }  //======================


   public Object peekTop()
   {  if (isEmpty())
         throw new IllegalStateException ("stack is empty");
      return itsItem[itsSize - 1];
   }  //======================


   public Object pop()
   {  if (isEmpty())
         throw new IllegalStateException ("stack is empty");
      itsSize--;
      return itsItem[itsSize];
   }  //======================


   public void push (Object ob)
   {  if (itsSize == itsItem.length)
      {  Object[] toDiscard = itsItem;
         itsItem = new Object [2 * itsSize];
         for (int k = 0;  k < itsSize;  k++)
            itsItem[k] = toDiscard[k];
      }
      itsItem[itsSize] = ob;
      itsSize++;
   }  //======================
}
```

When the ArrayStack object is first created, it starts with an array of size 10 (an arbitrary choice). Adding an element requires that there be room in the array. If not, you have to make the array bigger to hold the added element. Since you cannot simply add more components to an existing array, you create a new array that is twice as large and transfer the data to that other array. It becomes the new `itsItem` array. Figure 14.2 is the UML class diagram for the ArrayStack class.

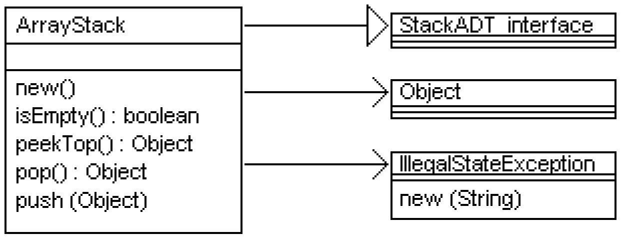


Figure 14.2 UML class diagram for the ArrayStack class

The **internal invariant** of a class of objects is the condition that (a) each method ensures is true when the method exits, so that (b) each method can rely on it being true when the method starts execution, because (c) no outside method can change things so that the condition becomes false (due to encapsulation). The purpose of an internal invariant is to describe the relationship of the abstract concept to the state of the instance variables (and sometimes class variables). The ArrayStack class implements the abstract concept of a stack as follows:

<u>Internal invariant for ArrayStacks</u>

- The int value `itsSize` is the number of elements in the abstract stack. These elements are stored in the array of Objects `itsItem` at the components indexed 0 up to but not including `itsSize`.
- If the stack is not empty, the top element of the stack is at index `itsSize-1`.
- If `k` is any positive integer less than `itsSize`, then the element at index `k-1` is the element that will be on top of the stack immediately after popping off the element at index `k`.

Linked-arrays implementation for faster execution

This array implementation requires copying many data values when the array was not initially big enough. But it can waste space if the array was initially too big. There is a way to avoid both these problems of execution time and space usage, but it requires a little more programmer effort. Specifically, when the array fills up, make another array of equal size and put the additional data values there, <u>without discarding the original array</u>.

Of course, you have to keep track of the "old" array somehow. Since an array is a kind of Object, and the new array has components for storing Objects, you can use one of those components in the new array to store a reference to the old array. For the ArrayStack `push` method in Listing 14.3, you could replace its if-statement by the following:

```
if (itsSize == itsItem.length - 1)
{  Object toSave = itsItem;
   itsItem = new Object [itsSize + 1];
   itsItem[itsSize] = toSave;
   itsSize = 0;
}
```

You need a corresponding change in the `pop` method: When the current array is empty, check to see if there is another "older" array and, if so, go on to that one. You could simply replace the coding of Listing 14.3's `pop` method by the following coding:

```
   if (itsSize == 0)
   {  itsSize = itsItem.length - 1;
      itsItem = (Object[]) itsItem[itsSize];
   }
   itsSize--;
   return itsItem[itsSize];
```

Technical Note This coding throws a NullPointerException if there is no older array, since then `itsItem` becomes null. However, this is precisely what the specifications call for -- they do not require that an IllegalStateException be thrown, only that some RuntimeException be thrown. And this coding saves time as compared with an explicit test. But if you insist on throwing the "right" kind of Exception, use a try/catch statement to catch the "undesirable" one and then throw an IllegalStateException in its place.

Efficiency

The previous discussion brings up an important point: **Efficiency** is not a matter of execution time alone; it is a combination of space, time, and programmer effort. Coding that executes faster may be less efficient if it uses more space or requires more effort by the programmer and by any future maintainer of the program. You have to consider all three of these factors in determining the efficiency of an algorithm.

There is a lot to be said for making efficiency the last consideration in developing software. Specifically, the first goal as you develop should be clarity of expression (make it easy to understand). The second goal should be correctness (avoid bugs). The third goal should be efficiency (make it fast or space-saving if the effort to do so is worth it).

Putting clarity first simply means that you work on keeping the coding clear as you develop it, rather than writing sloppily and cleaning it up later. You will find it much easier to write correct and efficient coding if you strive for clarity from the beginning.

Making your coding correct before you try for efficiency implies that you often choose the simplest way to get something done, as long as it works right. You may then come back to it and make it more efficient in terms of time and space (i.e., speed of execution or low use of RAM). One advantage is that you may then spot some bugs you missed the first time through. Moreover, it is well known that usually only about 10% of the coding is responsible for about 90% of the execution time, so you only have to work on a small part of the coding. This leaves the rest simple and straightforward, so that it will be easier to maintain in the future.

Another advantage is that, if your boss forces you to deliver the product before you have finished it, it is better to deliver a correct but slow version than a fast but buggy version. Slowness will irritate customers, but some bugs can kill people in certain software (e.g., for medical or military purposes). In such cases, delivering buggy software is unethical.

Exercise 14.9 How would the coding change in Listing 14.3 if you decided to have an instance variable `itsTop` keep track of the index of the top element (using -1 when the stack is empty), instead of using `itsSize` to keep track of the number of elements?
Exercise 14.10 (harder) Add a method `public boolean equals (Object ob)` to the ArrayStack class: The executor tells whether `ob` is an ArrayStack whose elements are equal to its own in the same order. Do not throw any Exceptions.
Exercise 14.11* Add a method `public int search (Object ob)` to the ArrayStack class: The executor tells how many elements would have to be popped to have ob removed from the stack; it returns -1 if `ob` is not in the stack.
Exercise 14.12* Write the `isEmpty` and `peekTop` methods for the linked-arrays approach to implementing a stack described in this section, without changing `itsItem`.

14.3 Implementing Queues With Arrays

You could implement QueueADT the same way as Listing 14.3 except that elements are removed at the other end of the array from that on which elements are added. So you could code `enqueue` the same as `push` but code `dequeue` as follows:

```
public Object dequeue()  // in a simplistic implementation
{  if (isEmpty())
      throw new IllegalStateException ("queue is empty");
   Object valueToReturn = itsItem[0];
   for (int k = 1;  k < itsSize;  k++)
      itsItem[k - 1] = itsItem[k];
   itsSize--;
   return valueToReturn;
}  //======================
```

Note that the for-statement copies the contents of `itsItem[1]` into the variable `itsItem[0]`, then the contents of `itsItem[2]` into `itsItem[1]`, etc. Copying in the opposite order would lose information. That is why the value to return is saved in a local variable before moving values down.

Leave the coding of `isEmpty` unchanged. The coding of `peekFront` is the same as `peekTop` except you return `itsItem[0]`. That completes the rather inefficient "move-them-all" implementation of QueueADT.

A better implementation of queues

The implementation of queues just described is very wasteful of execution time. If for instance the queue normally has around 100 elements in it, then each call of `dequeue` requires shifting 100 values around.

A better way is to keep track of two places in the array, the index of the front of the queue and the index of the rear of the queue. Call these instance variables `itsFront` and `itsRear`. Forget about `itsSize`. You add values at the rear and remove them at the front of the queue (except of course you cannot remove anything from an empty queue). So `peekFront` returns `itsItem[itsFront]`. The implementation of `dequeue` in this **ArrayQueue** class is as follows. Note that it corresponds line-for-line with ArrayStack's `pop`:

```
public Object dequeue()           // in ArrayQueue
{  if (isEmpty())
      throw new IllegalStateException ("queue is empty");
   itsFront++;
   return itsItem[itsFront - 1];
}  //======================
```

Say `itsFront` is 10. Then if `itsRear` is 12, the queue has three values in it (at components 10, 11, and 12). If `itsRear` is 10, the queue has one value in it (at component 10). In general, the number of values in the queue is `itsRear - itsFront + 1`. So how do you tell when the queue is empty? When this expression has the value zero, namely, when `itsRear` is 1 less than `itsFront`. Since you should add the first element at index 0 for a newly created queue, `itsFront` should initially be 0, which means that `itsRear` must initially be -1 (1 less than `itsFront` because the queue is initially empty). This coding is in the upper part of Listing 14.4 (see next page).

Listing 14.4 The ArrayQueue class of objects

```
public class ArrayQueue implements QueueADT
{
   private Object[] itsItem = new Object [10];
   private int itsFront = 0;  //location of front element, if any
   private int itsRear = -1;  //location of rear element, if any


   public boolean isEmpty()
   {  return itsRear == itsFront - 1;
   }  //======================

   public Object peekFront()
   {  if (isEmpty())
         throw new IllegalStateException ("queue is empty");
      return itsItem[itsFront];
   }  //======================


   public Object dequeue()
   {  if (isEmpty())
         throw new IllegalStateException ("queue is empty");
      itsFront++;
      return itsItem[itsFront - 1];
   }  //======================

   public void enqueue (Object ob)
   {  if (itsRear == itsItem.length - 1)
         adjustTheArray();
      itsRear++;
      itsItem[itsRear] = ob;
   }  //======================


   private void adjustTheArray()
   {  if (itsFront > itsRear / 4)
      {  itsRear -= itsFront;
         for (int k = 0;  k <= itsRear;  k++)
            itsItem[k] = itsItem[k + itsFront];
         itsFront = 0;
      }
      else
      {  Object[] toDiscard = itsItem;
         itsItem = new Object [2 * itsRear];
         for (int k = 0;  k <= itsRear;  k++)
            itsItem[k] = toDiscard[k];
      }  // automatic garbage collection gets rid of toDiscard
   }  //======================
}
```

The enqueue method

The basic logic of the `enqueue` method is to increment `itsRear` and put the given value at that component:

```
   itsRear++;
   itsItem[itsRear] = ob;
```

But you have a problem: What if `itsRear` is already up to `itsItem.length - 1` and so there is no room to add the new value? This will happen once the number of calls of `enqueue` is `itsItem.length`. There are two possibilities when in this case: Either the array has plenty of room at the lower indexes or it does not. In the former case, you have a simple solution: Just move all the values back down to the front of the array en masse. We choose to do this as long as the array is no more than three-quarters full:

```
if (itsFront > itsRear / 4)
{  for (int k = 0;  k <= itsRear - itsFront;  k++)
       itsItem[k] = itsItem[k + itsFront];
}
```

Check that these values are correct: Clearly the first iteration moves the front value in `itsItem[itsFront]` down to `itsItem[0]`. And the last iteration moves `itsItem[itsRear - itsFront + itsFront]`, which is `itsItem[itsRear]`, which of course should be the last one moved. And it is moved down to `itsItem[itsRear - itsFront]`. This illustrates a key technique for verifying that a for-statement is correct: Calculate what happens on the first and last iteration; otherwise it is easy to be "off by one". Since the position of the values in the queue has changed, you also have to update two instance variables:

```
itsRear -= itsFront;
itsFront = 0;
```

But what if the array is nearly full when you are called upon to add another value? You simply do what was done in the earlier Listing 14.3 for ArrayStack: Transfer all the elements to another array that is twice as large. This coding is in the lower part of Listing 14.4. Handling the case when `itsRear` reaches the rear of the array is in a separate private method because it would otherwise obscure the basic logic of `enqueue`.

Another popular way of implementing a queue avoids all movement of data except when the queue is full: When `itsRear == itsItem.length - 1` but `itsFront` is at least 2, add the next value at index 0. That is, the rear starts over from zero, leaving the front where it was. Double the size of the array only when the element you are adding would fill the array (do not wait until after it is full). This "avoid-all-moves" approach is harder to understand and code than the "move-when-forced" approach used for Listing 14.4, but it executes somewhat faster.

Exercise 14.13 Add a method `public int size()` to the ArrayQueue class: The executor tells the number of values currently in the queue.
Exercise 14.14 Add a method `public String toString()` to the ArrayQueue class: The executor returns the concatenation of the string representation of all elements currently in the queue with a tab character before each element, in order from front to rear. This is very useful for debugging purposes.
Exercise 14.15 (harder) Write a method `public void removeAfter (Object ob)` that could be added to ArrayQueue: The executor removes all values from the queue that come after the element closest to the rear that is equal to the parameter. If none are equal, leave the queue as it was originally. Handle `ob == null` correctly.
Exercise 14.16* Add a method `public boolean equals (Object ob)` to the ArrayQueue class: The executor tells whether `ob` is an ArrayQueue with whose elements are equal to its own in the same order. Do not throw any Exceptions.
Exercise 14.17* Add a method `public void clear()` to the ArrayQueue class: The executor deletes all the elements it contains, thereby becoming empty.
Exercise 14.18* Write out the internal invariant for ArrayQueues.
Exercise 14.19** Rewrite Listing 14.4 to "flip the array over": Store the front value initially at index `itsItem.length-1` and have `itsRear` move toward index 0.
Exercise 14.20** Write the method `public Object dequeue()` for the "avoid-all-moves" method of implementing QueueADT.

14.4 Implementing Stacks And Queues With Linked Lists

We next implement a StackADT object as a **linked list of Nodes**. 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. If for instance you want to represent a sequence of three data values as a linked list of Nodes, you put the three data values in three different Node objects and have the first Node refer to the second Node, the second Node refer to the third, and the third Node not refer to any Node at all.

The Node class

The Node class is defined in Listing 14.5. 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 14.5 The Node class

```
public class Node
{
   private Object itsData;
   private Node itsNext;


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


   /** Return the data attribute of the Node. */

   public Object getValue()
   {  return itsData;
   }  //======================


   /** Return the next attribute of the Node. */

   public Node getNext()
   {  return itsNext;
   }  //======================


   /** Replace the data attribute of the Node. */

   public void setValue (Object data)
   {  itsData = data;
   }  //======================


   /** Replace the next attribute of the Node. */

   public void setNext (Node next)
   {  itsNext = next;
   }  //======================
}
```

The following statements create a linked list of Nodes having the words "the", "linked", "list" in that order. The first statement creates a Node object in `nodeA` whose data value is "list" and which is linked up to no node (indicated by having `itsNext` be null). The second statement creates another Node object in `nodeB` which is linked up to `nodeA`. The third statement creates another Node object in `nodeC` which is then linked up to the second node, `nodeB`. Figure 14.3 shows how the list will look when it is done:

```
   Node nodeA = new Node ("list", null);
   Node nodeB = new Node ("linked", nodeA);
   Node nodeC = new Node ("the", nodeB);
```

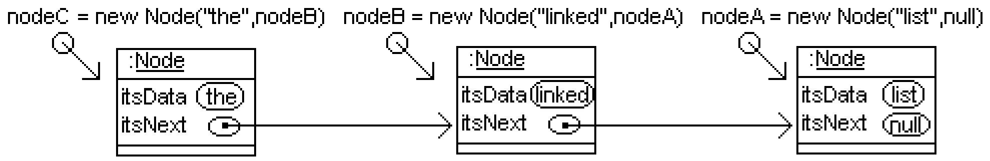

nodeC = new Node("the",nodeB)
nodeB = new Node("linked",nodeA)
nodeA = new Node("list",null)
:Node
itsData
the
itsNext
:Node
itsData
linked
itsNext
:Node
itsData
list
itsNext
null

Figure 14.3 UML object diagram of three nodes in a linked list

Nodes in a NodeStack object

We will create an implementation of StackADT named NodeStack. A **NodeStack** object will have an instance variable `itsTop` for the first Node object on its list. Initially `itsTop` is null, which is a signal that the stack is empty (after all, you cannot have any data without a Node to put it in). Suppose a particular NodeStack object has a linked list of two or more Nodes. Coding to swap the two data values in the first two Nodes could be as follows (the last two statements change only the data attribute of the Node):

```
   Object saved = itsTop.getValue();
   Node second = itsTop.getNext();
   itsTop.setValue (second.getValue());
   second.setValue (saved);
```

Coding to add a new data value "sam" at the beginning of a list could be as follows. Actually, this coding will work even when the linked list is empty (since then `itsTop` has the value null):

```
   Node newNode = new Node ("sam", itsTop);
   itsTop = newNode;
```

In fact, the `push` method for a linked list implementation of StackADT only needs one statement to add `ob` to the front of the list (the effect is illustrated in Figure 14.4):

```
   itsTop = new Node (ob, itsTop);
```

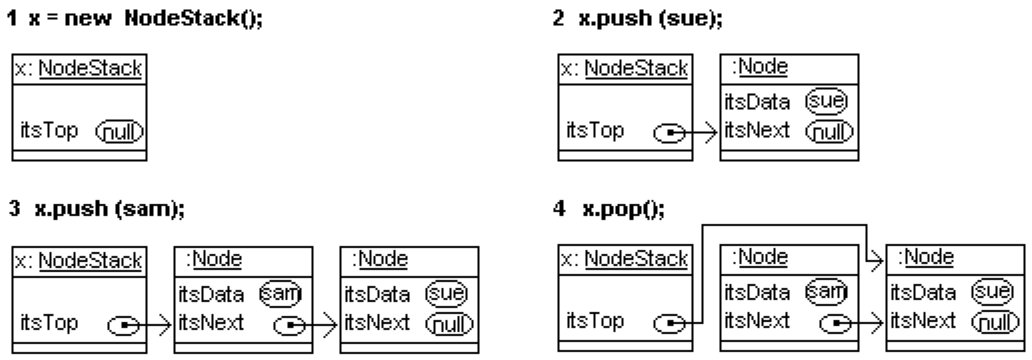


Figure 14.4 UML object diagrams for stack operations in NodeStack

The `peekTop` method is quite straightforward: First you check to make sure the stack is not empty (otherwise you throw an Exception). If the stack has at least one Node, the first Node is the top of the stack, so you simply return `itsTop.getValue()`. This coding is in the upper part of Listing 14.6 (see below).

The `pop` method is the most complex of the NodeStack methods: Once you check that the stack is not empty, you discard the first Node in the linked list, since it contains the data that you are to remove. This can be done with the following statement:

```
   itsTop = itsTop.getNext();
```

That statement stores a reference to the second Node on the list in the `itsTop` instance variable. If there was only one Node on the list in the first place, `itsTop.getNext()` has the value null, so `itsTop` is now null, which signals that the stack is empty. The coding for `pop` is in the lower part of Listing 14.6.

Internal invariant for NodeStacks
- If the abstract stack is empty, the Node value `itsTop` is null.
- If the abstract stack is not empty, `itsTop` refers to the first Node in a linked list of Nodes and `itsTop.getValue()` is the top element of the stack.
- If `p` is any Node in that linked list for which `p.getNext()` is not null, then the element `p.getNext().getValue()` is the element that will be on top of the stack immediately after popping off the element `p.getValue()`.
- The Nodes of one NodeStack are all different objects from those in any other.

Listing 14.6 The NodeStack class of objects

```
public class NodeStack implements StackADT
{
   private Node itsTop = null;


   public boolean isEmpty()
   {  return itsTop == null;
   }  //======================


   public Object peekTop()
   {  if (isEmpty())
         throw new IllegalStateException ("stack is empty");
      return itsTop.getValue();
   }  //======================


   public Object pop()
   {  if (isEmpty())
         throw new IllegalStateException ("stack is empty");
      Node toDiscard = itsTop;
      itsTop = itsTop.getNext();
      return toDiscard.getValue();
   }  //======================


   public void push (Object ob)
   {  itsTop = new Node (ob, itsTop);
   }  //======================
}
```

Useful simile The choice between a car and a lawn mower is analogous to the choice between a stack and a queue. The former depends on whether you want to go somewhere or mow the lawn; the latter depends on whether you want a LIFO or FIFO data structure. For the former, you need to know the functions of the various controls (brake, accelerator, starter); for the latter, you need to know the preconditions and postconditions of the methods. The driving force of a car or lawn mower can be gasoline or electricity; correspondingly, the internal invariant of a stack or queue can be a partially-filled array or a standard linked list. The coding of a method is mostly determined by its precondition and postcondition plus the internal invariant, just as the construction of a machine's control is mostly determined by its function plus the driving force.

Implementing queues with linked lists

A linked list is an excellent way to implement QueueADT, as long as you keep track of both the front and the rear of the list. That way, you can quickly add an element or remove an element. For this **NodeQueue** class, begin by declaring two instance variables `itsFront` and `itsRear`, each referring to a Node. In general, the queue's `itsFront` will always refer to the Node containing the first data value (if any) and the queue's `itsRear` will always refer to the Node containing the last data value (if any).

An empty queue has null for `itsFront`, since there are no data values and so no Nodes at all. The `isEmpty` , `dequeue`, and `peekFront` methods are precisely the same as `isEmpty`, `pop`, and `peekTop` are for NodeStack except of course `itsFront` is used in place of `itsTop`. The implementation so far is in the top part of Listing 14.7 (see next page). We repeat here the comment descriptions of the methods so you do not have to refer back to the earlier listing of QueueADT.

The enqueue method

To enqueue a new data value to the end of the queue, you normally create a new Node attached to the Node referred to by `itsRear`. Since that new Node is now the last Node, you change the value of `itsRear` to refer to the new Node.

However, if the queue is empty, you cannot attach a new Node to the last Node, because there is no last Node. In that case, you need to have the queue's `itsFront` refer to the new Node, because the new Node is the first Node. But you also need to have the queue's `itsRear` refer to the new Node, because the new Node is also the last Node. This coding is in the lower part of Listing 14.7.

Exercise 14.21 Write a method `public Object last()` that could be added to NodeQueue: Return the last data value in the queue; return null if the queue is empty.
Exercise 14.22 Write a method `public void dup()` that could be in NodeStack: The executor pushes the element that is already on top of the stack (so it now occurs twice). Throw an Exception if the stack is empty.
Exercise 14.23 Rewrite the `enqueue` method to execute faster by not assigning the newly-constructed Node to a local variable, but instead assigning it directly where it goes.
Exercise 14.24 Write a method `public int size()` that could be added to NodeQueue: Return the number of elements in the queue.
Exercise 14.25 Write a method `public void append (NodeQueue queue)` that could be in NodeQueue: The executor appends `queue`'s elements in the same order and sets `queue` to be empty. Precondition: The executor is not empty.
Exercise 14.26 What change would you make in the answer to the previous exercise to remove the precondition?
Exercise 14.27* Write a method `public void swap2and3()` that could be added to NodeStack: The executor swaps the second element with the third element in the stack. It has no effect if the stack has less than three elements.

Listing 14.7 The NodeQueue class of objects

```
public class NodeQueue implements QueueADT
{
   private Node itsFront = null;
   private Node itsRear;


   /** Tell whether the queue has no more elements. */

   public boolean isEmpty()
   {  return itsFront == null;
   }  //======================


   /** Return the value that dequeue would give without modifying
    *  the queue.  Throw an Exception if the queue is empty. */

   public Object peekFront()
   {  if (isEmpty())
         throw new IllegalStateException ("queue is empty");
      return itsFront.getValue();
   }  //======================


   /** Remove and return the value that has been in the queue the
    *  most time.  Throw an Exception if the queue is empty. */

   public Object dequeue()
   {  if (isEmpty())
         throw new IllegalStateException ("queue is empty");
      Node toDiscard = itsFront;
      itsFront = itsFront.getNext();
      return toDiscard.getValue();
   }  //======================


   /** Add the given value to the queue. */

   public void enqueue (Object ob)
   {  Node toBeAdded = new Node (ob, null);
      if (isEmpty())
         itsFront = toBeAdded;
      else
         itsRear.setNext (toBeAdded);
      itsRear = toBeAdded;
   }  //======================
}
```

Exercise 14.28* Write a NodeStack method `public int size()`: The executor tells the number of elements currently on the stack. Also add an instance variable `itsSize` to keep track of the current number so `size()` can be coded as a single statement.
Exercise 14.29* Write a method `public Object firstAdded()` that could be added to NodeQueue: The executor tells what was the first element ever added to it, even if it was removed later. It returns null if the queue has always been empty.
Exercise 14.30* Write out the internal invariant for NodeQueues.

14.5 The ListADT Abstract Class

We could implement StackADT using inheritance (making a subclass of Node) rather than composition (making a class of objects that have Node instance variables). However, it is simpler if we write a revision of the Node class rather than an extension, because we do not want the `setNext()` method to be publicly available to other classes. We will create a class like Node but more powerful -- it includes all four of the services of StackADT plus several others appropriate to lists.

The ListADT definition

A **ListADT** object offers client classes two basic services besides the four StackADT methods. We define ListADT as an abstract class rather than as an interface because that allows us to provide over a dozen additional convenience methods in the class (i.e., methods whose coding can be written using just the basic six methods). A concrete implementation of ListADT can then override these convenience codings or not, whichever works best. Any client of any of the various ListADT implementations we will write can use the convenience methods as needed.

Listing 14.8 shows the essential part of ListADT (postponing the convenience methods). The method that lets us iterate through the list is a method named `theRest`. If `x` is a non-empty ListADT, `x.theRest()` is the **sublist** consisting of the second and all later data values. For instance, `x.theRest().pop()` removes the second data element on x's list. But if `x` has just one data value, `x.theRest()` is empty and `x.theRest().theRest()` is null.

Listing 14.8 The abstract ListADT class of objects (to be added to later)

```
public abstract class ListADT implements StackADT
{
   /** Return the portion of this list that contains all values
    *  after the first value. Return null if the list is empty.*/

   public abstract ListADT theRest();


   /** Replace the data value at the front of the list by ob.
    *  No effect if this list is empty. */

   public abstract void setTop (Object ob);


// The four StackADT methods (descriptions are in Listing 14.1)

   public final boolean isEmpty()
   {  return theRest() == null;
   }  //======================

   public abstract Object peekTop();

   public abstract Object pop();

   public abstract void push (Object ob);
}
```

Your coding can have a semicolon for the body of any instance method as long as you put "abstract" in each such method heading and also in the class heading. That makes it an abstract class. You then declare a "concrete" subclass of ListADT that actually has coding for abstract methods. For the six methods named in Listing 14.8, `isEmpty` cannot be overridden (since it is `final`) and the other five must be overridden (since they are `abstract`). The methods in the next listing may be overridden.

Any implementation of ListADT has the responsibility to make sure that two lists cannot have a sublist in common unless one is a sublist of the other. That is, whenever `x.theRest() == y.theRest()`, then `x == y`. An implementation of ListADT must also assure that a list cannot be a sublist of itself and that a constructor is provided, with no parameters, that makes an empty list. These conditions are the **contract** for ListADT.

But wait! There's more!

We call this the ListADT class because a ListADT object represents a list of data values, not just a stack of data values. This class has additional methods for changing or finding out about data values not at the top of the list. For instance, `sam.addLast (ob)` puts the data value `ob` at the end of `sam`'s list, and `sam.size()` returns the number of data values currently in the list.

The method call `someList.addLast (ob)` works as follows: If the list is empty, `ob` is added to that list to be its only data value. But if the list is not empty, then the executor "asks" its direct sublist `this.theRest()` to add `ob` to the end of the list. That other list then goes through the same process as the executor did (inserts `ob` if empty, otherwise asks its direct sublist to perform the `addLast` operation). The coding for the `addLast` method is in the upper part of Listing 14.9.

Listing 14.9 The ListADT class of objects, part 2

```
// public abstract class ListADT, continued

   /** Add the given data value at the end of this list. */

   public void addLast (Object ob)
   {  if (this.isEmpty())
         this.push (ob);
      else
         theRest().addLast (ob);
   }  //======================


   /** Return the number of data values in this list. */

   public int size()
   {  return this.isEmpty() ? 0 : 1 + theRest().size();
   }  //======================


   /* These ListADT methods are described in the exercises
   public void clear()         // remove all data, leave it empty
   public void copyTo (ListADT par)  // insert all at the front
   public void addAll (ListADT par)  // append all at its end
   public Object[] toArray (Object[] par)  // copy it to an array
   public String toString()  // return a String representation */
```

Figure 14.5 illustrates this process when the list initially has two data values A and B. It shows NodeList objects rather than ListADT objects because NodeList is a "concrete" subclass of ListADT you will see shortly; the NodeList `y` on the left is an empty list.

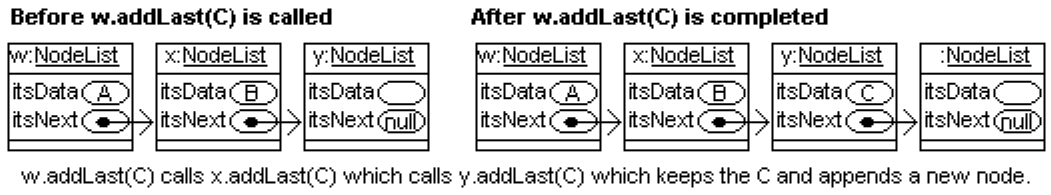

Before w.addLast(C) is called
After w.addLast(C) is completed
w:NodeList
x:NodeList
y:NodeList
:NodeList
itsData
itsNext
A
B
C
null
w.addLast(C) calls x.addLast(C) which calls y.addLast(C) which keeps the C and appends a new node.

Figure 14.5 What addLast does when called with a three-node linked list

We can do things this way because `this.theRest()` is itself a ListADT object. And each ListADT object can be asked to perform the `addLast` operation. A process in which one object asks another object from the same class to carry out the same process is called a **recursive process**. The principle is simple: An instance method of a class can be called for any object of that class, even if the method call is within the coding of that instance method.

This kind of recursive logic works when the method is called for the direct sublist of the executor because that sublist has a shorter list of data values, according to the contract for ListADT. So eventually the process must end with an empty list, if not sooner.

The method call `sam.size()` returns the number of data values in the list represented by `sam`. This is again a recursive process: If the executor is empty, it knows that the size of its list is zero; otherwise the size of its list must be 1 more than the size of the rest of the list. This coding is in the lower part of Listing 14.9. The exercises describe more ListADT methods and code two of them.

Limits of recursion

Once you develop a moderate level of comfort using recursion, you will often find that it is easier to develop a recursive logic than to use a loop. However, most recursive logics can be written with a loop without too much difficulty. For instance, the `addLast` method could have been coded this way (`pos` denotes a particular position in the list):

```
public void addLast2 (Object ob)
{  ListADT pos = this;
   while ( ! pos.isEmpty())
      pos = pos.theRest();
   pos.push (ob);
}  //=====================
```

A method call is "uncompleted" if the runtime system has not yet returned from the method called. The **stack trace** for a statement is the list of all the "uncompleted" method calls that have been executed up to that statement. For instance, if you call the `addLast` method initially for a list with three data values, the statement `this.push (ob)` will not execute until during the fourth call of `addLast`; at that point the stack trace contains four calls of `addLast`. We say that the last call of `addLast` (the one that actually does the pushing) has a **recursion level** of four.

The software throws a **StackOverflowError** when recursion goes too deeply. That limit is typically several thousand on many machines. But in general, logic that can produce a recursion level of more than a hundred or two should be rewritten to use a loop (as illustrated above for the `addLast` method).

Coding the NodeList subclass of ListADT

We make a subclass of ListADT named **NodeList**. A NodeList object has the same internal structure as a Node object, though it has different methods. If it represents a list with five data values, it will be the first node in a linked list of six nodes, the last node having no data (so it is called a **trailer node**). In general, N data values in the list require N+1 nodes. The first node in the list contains the data that is on top of the stack, the second node in the list contains the data that will be on top after the first one is popped, etc. The coding for `theRest`, `peekTop`, and `setTop` follows logically. It is in the upper part of Listing 14.10 (see below). The default constructor creates an empty list.

You might think that `setTop` has an effect for an empty list. However, no outside class can detect any difference from having no effect, and that is all that counts. In a sense, it is okay to lie if (a) you can be sure of getting away with it and (b) it speeds execution.

The coding for `push` creates a new node to go between the current first node and second node (lines 6, 8, 10), copies the data value from the current first node into that new node (line 7), and then puts `ob` in the first node (line 9). That makes `ob` the top value on the stack. The data value that was originally the top value is now the second value on the stack, and the data value (if any) that was originally second is now third.

Listing 14.10 The NodeList class of objects

```
public class NodeList extends ListADT
{
   private Object itsData = null;
   private NodeList itsNext = null;

   public ListADT theRest()
   {  return itsNext;                                           //1
   }  //======================

   public void setTop (Object ob)
   {  itsData = ob;                                             //2
   }  //======================

   public Object peekTop()
   {  if (itsNext == null)  // so it represents an empty list//3
         throw new IllegalStateException ("nothing there");  //4
      return itsData;                                           //5
   }  //======================

   public void push (Object ob)
   {  NodeList toAdd = new NodeList();                          //6
      toAdd.itsData = this.itsData;                             //7
      toAdd.itsNext = this.itsNext;                             //8
      this.itsData = ob;                                        //9
      this.itsNext = toAdd;                                     //10
   }  //======================

   public Object pop()
   {  Object valueToReturn = this.itsData;                      //11
      NodeList toDiscard = this.itsNext;                        //12
      this.itsData = toDiscard.itsData;                         //13
      this.itsNext = toDiscard.itsNext;                         //14
      toDiscard.itsNext = null;   // make this list empty    //15
      return valueToReturn;                                     //16
   }  //======================
}
```

The NodeList class has by default a single constructor with no parameters, which constructs an empty list.

Coding NodeList's pop method

The `pop` logic in Listing 14.10 copies both parts of the following node into the executor node (lines 13 and 14), so it is as if that following node were never there. The link from the discarded node is erased (line 15) to be sure that two different nodes never link to the same node. If an outside class retains a reference to the discarded node, it is just one more empty list. Note that an empty list need not have null data, though it usually does.

If the stack is empty when `pop` is called, then `toDiscard` has the value null, so execution of line 13 will throw a NullPointerException. The specifications for the `pop` method only require that some Exception be thrown when the stack is empty, not any particular kind of Exception. So this coding does what the specifications require.

Similarly, in the NodeStack class of the earlier Listing 14.6, we could save execution time by eliminating the first two lines of `peekTop` and `pop`, since those two if-statements are redundant. However, we need the if-statement in the `peekTop` method of Listing 14.10 because `return itsData` does not throw an Exception when the list is empty.

Note that we can now implement QueueADT as a subclass of NodeList, since `addLast`, `peekTop`, and `pop` do just what `enqueue`, `peekFront`, and `dequeue` do, respectively. Such a subclass is called an **adapter**, since it adapts methods of one class to implement methods of another class. This is left as an exercise. Later in this chapter we will discuss other subclasses of ListADT, such as CardList, DoublyLinked, HeaderList, and ListQueue; the ListQueue implements queues so that `addLast` works much faster.

A class with the methods `pop`, `push`, and `peekTop` for data values at the front, plus the equivalent methods `addLast`, `removeLast`, and `getLast` for data values at the rear (the last two described in the next section), is called a **deque** (pronounced "deck").

Exercise 14.31 Write a ListADT method `public void clear()`: The executor discards all data values on its list, leaving itself empty.
Exercise 14.32 Override the `clear` method of the preceding exercise with a NodeList method that executes much faster.
Exercise 14.33 (harder) Write the ListADT method `public void copyTo (ListADT par)`: The executor adds to the front of the given ListADT all of its data values in the same order they occur in the executor's list. Precondition: `par` is not null.
Exercise 14.34* Write the ListADT method `public void addAll (ListADT par)`: The executor adds all of the data values in `par` to the end of its own list, in the same order as they are in `par`. Call on the `copyTo` method mentioned in the previous exercise. Precondition: `par` is not null.
Exercise 14.35* Write a ListADT constructor `public ListADT (Object[] par)`: Create a ListADT object whose data values are all the values in the components of the array, in the same order. It should be an empty list if the parameter has zero components, and it should throw an Exception if the parameter is null.
Exercise 14.36* Write a ListADT method `public Object[] toArray (Object[] par)`: The executor returns an array whose components contain all the data values in the executor's list, in the same order. Return the parameter unless its length is less than the number of data values in the list, in which case return a newly-created array whose length is the size of the executor. Throw an Exception if `par` is null.
Exercise 14.37* Write a class that extends the NodeList class and implements the QueueADT interface. Write the least amount of coding possible.
Exercise 14.38* Essay question: Explain why a NodeList never represents a circular linked list and two different NodeList objects cannot have the same `theRest` value.

Part B Enrichment And Reinforcement

14.6 Additional Linked List Operations Using Recursion

The Sun standard library interface `java.util.List` has over twenty different methods. They are the basis for the ListADT methods described in the preceding section and in this one. Many of those methods find or modify a value given an index in the list. Another method searches for the index of a given object.

Using zero-based indexes

A call of `sam.remove (4)` removes the data value at index 4 in the list, i.e., 4 nodes past `sam`. Index values are zero-based, e.g., `sam.remove (0)` removes the first data value in the list and `sam.remove (1)` removes the second data value in the list.

The method call `sam.remove (n)` removes from `sam`'s list the data value that is n nodes past `sam`. This is again a recursive process: If the index is zero, the executor knows that it must remove the data value it contains, so it performs the `pop` operation. If the index is positive, the executor asks the node at the front of the rest of the list to remove the data value whose index in the sublist is 1 less. For instance, `x.remove (2)` calls `remove (1)` for the node after `x`, which calls `remove (0)` for the second node after `x`, which pops the data value from that second node after `x`.

If `remove` is called with the index negative or larger than `size()`, `theRest` will eventually become null. That causes the expression `theRest().remove (index-1)` to throw a NullPointerException. This coding is in the upper part of Listing 14.11.

Listing 14.11 The ListADT class of objects, part 3

```
// public abstract class ListADT, continued

   /** Remove and return the data value at the given index (zero-
    *  based). Throw an Exception if no data is at that index. */

   public Object remove (int index)
   {  return index == 0 ? pop() : theRest().remove (index-1); //1
   }  //======================

   /** Insert the data value at the given index (zero-based).
    *  Throw an Exception if index < 0 or index > size(). */

   public void add (int index, Object ob)
   {  if (index == 0)                                         //2
         push (ob);                                           //3
      else                                                    //4
         theRest().add (index - 1, ob);                       //5
   }  //======================

   /** Return the last data value in this ListADT.
    *  Throw an Exception if the ListADT is empty. */

   public Object getLast()
   {  return theRest().isEmpty() ? peekTop()                  //6
                                 : theRest().getLast();       //7
   }  //======================
```

The method call `sam.add (n, ob)` inserts `ob` at index n, so that everything that was originally at index n or higher now has an index 1 larger. This is again a recursive process. If the coding for `add` in the middle part of Listing 14.11 is not clear, compare it carefully with the coding for `remove` and for `addLast`. Note that `add (0, ob)` and `remove (0)` are the same as `push (ob)` and `pop()`.

The `getLast()` call returns the last data value in the list. The executor looks to see if `theRest` is empty, which means that the executor is a list with just one data value, which is therefore the last data value. If however there is additional data, the executor asks `theRest` to return the last data value in its list. So this is again a recursive process. If the original list is empty, the executor throws a NullPointerException, since `theRest` is null. This coding is in the lower part of Listing 14.11.

Searching for the index of a data value

The method call `sam.indexOf (ob)` returns the index where the data value `ob` is in the list. If `ob` occurs several times in the list, `sam.indexOf (ob)` returns the index of the first occurrence. If the data value `ob` is not in the list at all, the method call returns -1.

In other words, if `ob` is n nodes past the executor and not in any node in between, `sam.indexOf (ob)` returns n. For instance, if `ob` is the first data value in `sam`'s list, `sam.indexOf (ob)` returns zero.

It is easiest to separate this logic into two parts, depending on whether the data value is null or not. So the `indexOf` method in the upper part of Listing 14.12 (see next page) calls one of two different private methods: `indexOfNull (0)` finds the first index where null occurs, and `indexOf (0, ob)` finds the first index where the given non-null object `ob` occurs. The zero parameter is the index for the executor. In general, `someNode.indexOfNull (n)` tells the zero-based index of null in the original list when `someNode` is n nodes past the original node.

The logic of `indexOfNull` is as follows: If the executor represents the empty list, null is not a data value in that list, so you return -1 to signal that result. Otherwise, if null is the data value in the executor node, you return the index of that node in the original list. In all other cases, you ask the rest of the list for the index of null in the original list (the node at the front of the rest of the list has index 1 higher in the original list than the executor had). The search for a non-null data value has almost exactly the same logic (line 10 differs from line 4, but the rest are the same)

The method call `sam.contains (ob)` tells whether the list contains the given object as a data value. Since you also have to allow for null here, it is simplest to find out whether the `indexOf` method returns -1 or not. This coding is in the bottom part of Listing 14.12.

Iterating through a ListADT

Clients of the ListADT class will often want to go through each data value on a list in a way not already provided by one of the ListADT methods. For instance, if you know that the objects stored as data are all Integer objects, for which the `intValue()` method returns the int equivalent of their values, you can use the following independent recursive method to find the total of those int values:

```
public static int totalValue (ListADT list)
{  return list.isEmpty() ? 0
         : ((Integer) list.peekTop()).intValue()
               + totalValue (list.theRest());
}  //=====================
```

Listing 14.12 The ListADT class of objects, part 4

```
// public abstract class ListADT, continued

   /** Return the lowest index where ob occurs.
    *  Return -1 if ob does not occur anywhere in the list. */

   public int indexOf (Object ob)
   {  return ob == null ? indexOfNull (0) : indexOf (0, ob); //1
   }  //======================

   private int indexOfNull (int index)
   {  if (isEmpty())                                         //2
         return -1;                                          //3
      else if (null == peekTop())                            //4
         return index;                                       //5
      else                                                   //6
         return theRest().indexOfNull (index + 1);           //7
   }  //======================

   // Precondition:  ob is not null.

   private int indexOf (int index, Object ob)
   {  if (isEmpty())                                         //8
         return -1;                                          //9
      else if (ob.equals (peekTop()))                        //10
         return index;                                       //11
      else                                                   //12
         return theRest().indexOf (index + 1, ob);           //13
   }  //======================

   /** Tell whether the parameter is one of the data values
    *  in the list. */

   public boolean contains (Object ob)
   {  return indexOf (ob) != -1;                             //14
   }  //======================

   /* These ListADT methods are described in the exercises
   public Object get (int index)       // return the data there
   public void setLast (Object ob)     // replace the last data
   public void set (int index, Object ob)  // replace that data
   public Object removeLast()    // remove the last and return it
   public boolean remove (Object ob)   // remove it if you can
   public int lastIndexOf (Object ob)
   public boolean equals (ListADT that)
   public String toString()
   public boolean containsAll (ListADT that)     */
```

This recursive logic just says that the sum of the values on an empty list is zero, but the sum for a non-empty list is the `intValue` of the first data value plus the sum of the values for the rest of the list. But you cannot simply append `.intValue()` to an Object expression. You have to class cast `(Integer)` the Object expression to tell the compiler that the object will be of the Integer subclass type at runtime. And you have to use parentheses to show the compiler exactly what expression you want to cast.

If a client class has a ListADT object for which each null data value is to be replaced by a particular value x, the following independent recursive method would work right:

```
public static void replaceNullsBy (ListADT list, Object x)
{  if ( ! list.isEmpty())
   {  if (list.peekTop() == null)
         list.setTop (x);
      replaceNullsBy (list.theRest(), x);
   }
}  //======================
```

This process could be done non-recursively as follows. Compare the two codings to see how they differ (both have the precondition that `list` is not null):

```
public static void replaceNullsBy2 (ListADT list, Object x)
{  while ( ! list.isEmpty())
   {  if (list.peekTop() == null)
         list.setTop (x);
      list = list.theRest();
   }
}  //======================
```

Exercise 14.39 Write a ListADT method `public Object get (int index)` recursively: The executor returns the data value at the given index. It throws an Exception if the index is out of range.
Exercise 14.40 Write a ListADT method `public void setLast (Object ob)` recursively: The executor replaces the data value at the end of its list with `ob`. It throws an Exception if the list is empty.
Exercise 14.41 Write an independent non-recursive method `public static int numNulls (ListADT list)`: Return the number of null data values in the list.
Exercise 14.42 (harder) Write a ListADT method `public boolean containsAsSubList (ListADT par)` recursively: The executor tells whether `par` is some sublist of itself.
Exercise 14.43 (harder) Rewrite `public void add (int index, Object ob)` of Listing 14.11 non-recursively. Compare this closely with the recursive version to see why recursive logic is very often conceptually easier to develop.
Exercise 14.44 (harder) Write a ListADT method `public boolean equals (ListADT that)` recursively: The executor tells whether its list contains the same data values in the same order as in the parameter list. Precondition: No data values are null.
Exercise 14.45* Write a ListADT method `public Object removeLast()` recursively: The executor removes the data value at the end of its list and returns it. It throws an Exception if the list is empty.
Exercise 14.46* Write a ListADT method `public void set (int index, Object ob)` recursively: The executor replaces the data value at the given index with `ob`. It throws an Exception if the index is negative or greater than `size() - 1`.
Exercise 14.47* Rewrite the `remove` method in Listing 14.11 non-recursively.
Exercise 14.48* Rewrite the `indexOfNull` method in Listing 14.12 non-recursively.
Exercise 14.49* Write a ListADT method `public String toString()`: The executor returns the String values of all its data values, in the order they occur in its list, with "; " after each of them.
Exercise 14.50* Write a ListADT method `public boolean containsAll (ListADT that)` recursively using just one statement: The executor tells whether its list contains all the data values that are in the parameter's list. Call on the `contains` method appropriately.
Exercise 14.51** Write a ListADT method `public boolean remove (Object ob)`: The executor removes the first occurrence of `ob` in its list. Do not look at any Node more than once. Return true if `ob` was in the list; return false if it was not.
Exercise 14.52** Write a ListADT method `public int lastIndexOf (Object ob)`: The executor returns the index of the last occurrence of `ob` in its list; the executor returns -1 if `ob` is not one of the data values in its list.

14.7 Sorting And Shuffling With Linked Lists

You often need to work with lists of values that can be ordered using the standard `compareTo` method from the Comparable interface. This requires that all the data values be Comparable and non-null. Moreover, you have to be able to compare any two of them without causing a ClassCastException. Under these assumptions, you can write coding to find the smallest data value in a non-empty ListADT named `list` as follows:

```
Comparable smallestSoFar = (Comparable) list.peekTop();
for (ListADT p = list.theRest(); ! p.isEmpty();
              p = p.theRest())
{  if (smallestSoFar.compareTo (p.peekTop()) > 0)
      smallestSoFar = (Comparable) p.peekTop();
}
```

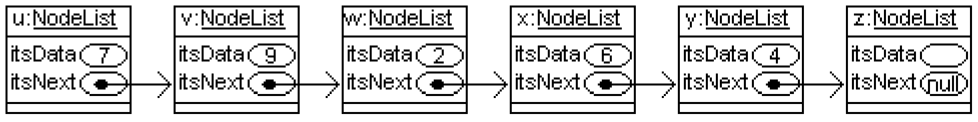
In some situations you want to keep a list of mutually Comparable values in **ascending order** (each one larger than or equal to the one before it in the list). This requires that, when you add a data value to the list, you find the first data value that is larger or equal and insert your data value just before that data value you found. Of course, if the list is empty, or if all of the data values already in it are smaller than the one you are inserting, then you insert your data value at the end of the list.

The `insertInOrder` method in the upper part of Listing 14.13 accomplishes this task. If the executor's list is empty, or if your value to insert is not larger than the first value in the list, the executor pushes your data value at the front of its list; otherwise it asks `theRest` to insert it later. Note how similar it is to `add` in the earlier Listing 14.11.

Listing 14.13 The ListADT class of objects, part 5

```
// public abstract class ListADT, continued

   /** Add the given value to the list before the first data
    *  value that is greater-equal to it, using compareTo.
    *  Add it at the end of the list if there is no such value.
    *  Precondition:  ob is non-null and is Comparable to all
    *  data values currently in the list. */


   public void insertInOrder (Comparable ob)
   {  if (this.isEmpty() || ob.compareTo (peekTop()) <= 0)
         this.push (ob);
      else
         theRest().insertInOrder (ob);
   }  //======================


   /** Rearrange the data values to be in ascending order.  Throw
    *  an Exception unless all values are mutually Comparable. */


   public void insertionSort()
   {  if ( ! this.isEmpty())
      {  theRest().insertionSort();
         this.insertInOrder ((Comparable) this.pop());
      }
   }  //======================
```

You can use this `insertInOrder` method to sort all of the data values on an existing ListADT in ascending order. The basic logic, called the **Insertion Sort logic**, is in the lower part of Listing 14.13: To sort a non-empty list, first sort the sublist consisting of all the values after the first one, then pop the first one off the list and insert it where it goes.



Insertion sort puts the last four data values in order (2, 4, 6, 9) and then inserts the 7 before the 9.
Selection sort first removes the 2 and puts it in front, then sorts the remaining five (7, 9, 6, 4).

Figure 14.6 Sorting a linked list

Another logic for sorting values in ascending order finds the smallest value in the list, removes it, pushes it on the front of the list, and then sorts the rest of the list. This is called the **Selection Sort logic**. The `removeSmallest` method is an exercise:

```
public void selectionSort()
{  if ( ! this.isEmpty())
   {  this.push (this.removeSmallest());
      theRest().selectionSort();
   }
}  //======================
```

These sorting logics execute rather slowly for more than several hundred values. Moreover, recursive processes that run more than a thousand deep tend to fail due to RAM limitations (it varies from 1000 to 10,000 or more, depending on the computer). Algorithms that avoid both of these problems are presented in another chapter.

An application using NodeLists

Assume you have a client who spends a lot of time playing this simple Solitaire game:

1. Use a standard 52-card deck -- thirteen different ranks of cards with four cards of each rank, e.g., 4 of clubs, 4 of spades, 4 of hearts, and 4 of diamonds.
2. Shuffle the cards and spread them out in sequence face up.
3. Remove any two adjacent cards of the same suit or of the same rank.
4. Repeat the step above until all cards are removed (so you win) or you cannot remove two adjacent cards (so you lose).

For instance, if playing the game reduces the deck to four cards in the order 4 of clubs, 5 of clubs, 5 of spades, 4 of hearts, it is illegal to remove the last pair. Removing the first pair (two clubs) leaves the 5 of spades and 4 of hearts, which loses; but removing the middle pair (two 5s) leaves the 4 of clubs and 4 of hearts, which wins. The client wants to know (and is willing to pay you to find out) the probability of winning the game if one always removes the first pair in the display that can be removed.

You decide to use the **Monte Carlo method** to find the probability: Play the game for maybe 10,000 random arrangements of cards and estimate the theoretical probability of winning to be the actual number of wins divided by 10,000. If you have say 100,000 or more test runs, the estimate will be more accurate, but you decide the client is not paying you enough for that.

So you design a class of objects that represent Cards, with these instance methods:

`someCard.getSuit()` tells which suit the Card is in.
`someCard.getRank()` tells which numeric rank the Card has (e.g., 13 for a King).

You decide to represent a deck of cards in a particular order with a **CardList**, which you make a subclass of NodeList. Each of its data values is a Card object. In designing this object class, you decide it should offer the following services to outside classes:

`public void shuffleAllCards (int numToDo)` puts them in random order if numToDo is the number of Cards in the list
`public boolean removeAllSucceeds()` tells whether you win the game by removing the first available pair of cards each time

Implementing the CardList class

To `shuffleAllCards` for 52 cards, you only need to pick a random number 0 up to but not including 52, remove the card at that (zero-based) index, put that card at the front of the deck, and then shuffle the remaining 51 cards the same way. This logic can be coded as shown in the upper part of Listing 14.14 (see next page).

To find out whether `removeAllSucceeds`, you first see if this list is empty (in which case it succeeds). If not, you try to remove the first legally-removable pair of Cards. If you do not succeed in doing so, then return false from `removeAllSucceeds`, otherwise find out whether `removeAllSucceeds` for the list that has now been shortened by two cards. This recursive logic is in the middle part of Listing 14.14.

The logic for `removeTheFirstSucceeds` is more complex. After a moderate amount of thought, you come up with the algorithm design in the following design block. Once you have this design, the coding in the lower part of Listing 14.14 then follows easily.

STRUCTURED NATURAL LANGUAGE DESIGN for removeTheFirstSucceeds

```
1. If the list has less than two Cards then...
         You cannot legally remove any pair of cards, so return false.
2. If the first two cards have either the same rank or the same suit then...
         Remove those two cards from the deck.
         Return true.
3. Apply the process just described to the part of the list after the first card.
```

Personal confession: I thought I was so competent that I could just write the code for this method directly, doing the design in my head. Some time later I wrote out the design block for this book. Only then did I realize that my code had a bug. I could not see the bug when the logic was written in Java, but it was clear when the logic was written in English. This just goes to confirm a basic programming fact: First write out the design in English to greatly reduce both the number of errors you need to find later and the total development time for error-free coding.

Exercise 14.53 (harder) Write a ListADT method `public Object removeSmallest()`: The executor removes and returns the smallest data value in its list. Precondition: The list is not empty and all data values are mutually Comparable.
Exercise 14.54 (harder) Write a ListADT method `public Object findLargest()`: The executor returns the largest data value in its list. Precondition: The list is not empty and all data values in the list are mutually Comparable.
Exercise 14.55* Write a ListADT method `public void swapSmallest()`: The executor swaps its first data value with its smallest data (put each in the other's node). Revise the `selectionSort` method to use this method appropriately.
Exercise 14.56* Write a ListADT method `public void removeAbove (Object ob)`: The executor removes all data values from its list that are larger than `ob`. Precondition: All data values including `ob` are mutually Comparable.

Listing 14.14 The most interesting part of the CardList class of objects

```
public class CardList extends NodeList
{
   private java.util.Random randy = new java.util.Random();


   /** Put the numToDo data values in a random order.
    *  Throw an Exception if numToDo > this.size(). */

   public void shuffleAllCards (int numToDo)
   {  ListADT list = this;                                   //1
      for ( ;  numToDo >= 2;  numToDo--)                     //2
      {  list.push (list.remove (randy.nextInt (numToDo)));  //3
         list = list.theRest();                              //4
      }                                                      //5
   }  //======================


   /** Repeatedly remove the first legally-removable pair of
    *  cards. Return true if this leaves the list empty, false
    *  if not. Precondition: All data values are Card objects. */

   public boolean removeAllSucceeds()
   {  return this.isEmpty() || (removeTheFirstSucceeds (this)//6
                            && this.removeAllSucceeds());//7
   }  //======================


   /** Remove the first legally-removable pair of cards from list
    *  if you can.  Return false if there is no legal move. */

   private boolean removeTheFirstSucceeds (ListADT list)
   {  if (list.theRest().isEmpty()) //positioned at last card//8
         return false;                                       //9
      Card first = (Card) list.peekTop();                    //10
      Card second = (Card) list.theRest().peekTop();         //11
      if (first.getRank() == second.getRank()                //12
                || first.getSuit() == second.getSuit())      //13
      {  list.pop();                                         //14
         list.pop();                                         //15
         return true;                                        //16
      }                                                      //17
      return removeTheFirstSucceeds (list.theRest());        //18
   }  //======================
}
```

Exercise 14.57* Write a ListADT method `public boolean isAscending()`: The executor tells whether all of its list's data values are in ascending order. Precondition: The list is not empty and all data values are mutually Comparable.
Exercise 14.58** Write a ListADT method `public void bubbleSort()`: The executor rearranges the data values to be in ascending order by leaving most of the work to the method in the following exercise.
Exercise 14.59** Write a ListADT method `public void bubble()`: The executor goes through the list in order, swapping any two adjacent data values for which the first is larger than the second. This is a slow way to get the largest value moved to the rear. Precondition: All data values are mutually Comparable.

14.8 Doubly-Linked Lists And Lists With Header Nodes

Sometimes we want to move from a certain position in a list to the position just before it. We would like to be able to use a method call such as `someList.theOneBefore()`, which would be the opposite of `someList.theRest()`. In other words, the following two conditions would be true whenever `someList` is non-empty and is a sublist of another list:

```
   someList == someList.theOneBefore().theRest();
   someList == someList.theRest().theOneBefore();
```

We cannot do this with NodeLists -- there is no way for a given NodeList object to be able to tell you the list that it is part of. We can fix this by making a different subclass of ListADT objects, each one knowing the list it is `theRest` of. We will call this the **DoublyLinked** class, because each node is generally linked to two nodes.

Coding the DoublyLinked class

It is fairly obvious that this subclass should have one extra instance variable for each DoublyLinked object to keep track of the DoublyLinked object for which it is `theRest`. We call it `itsPrevious`. We store null in this `itsPrevious` variable if the object is not `theRest` for any list. A Node that represents an empty list should have null for both linkages, since there is no node before or after it. The instance method `theOneBefore` returns the list for which the executor is `theRest`. It return null if there is no such containing list. This coding is in the upper part of Listing 14.15 (see next page).

The DoublyLinked coding for `push` is in the middle part of Listing 14.15. It has exactly the same five statements (lines 2-6) that NodeList's `push` has, except of course for the type of the `toAdd` variable. Then it corrects the `itsPrevious` values for the two nodes that are affected (lines 7-9). Specifically, the node after `toAdd` links back to `toAdd` and `toAdd` links back to the executor.

The `pop` method has exactly the same six statements that NodeList's `pop` has (lines 10-14 and 18), but we add line 15 to make the deleted node empty, and we add lines 16 and 17 to have the node after the one deleted (if any) refer back to the executor instead of to the deleted node. The class cast in lines 9 and 17 is needed because (a) `theRest` is declared to return a ListADT value but we know that the value returned is in fact a DoublyLinked value, and (b) we cannot use `itsPrevious` with a ListADT expression.

Backing structures

Note that if you have ListADT objects `x` and `rest` where `rest == x.theRest()`, whether `x` is a DoublyLinked or NodeList, then `x.pop()` changes `rest` to represent an empty list. This is because `x` and `rest` are **backed by** the same list, and any change in `x` or `rest` causes a change in the backing list. However, `x.push()` does not change the essential nature of any other ListADT object `y` -- `y.peekTop()` and `y.theRest()` remain as they were before the `push` operation.

In the Sun standard library of utilities, an Iterator is similar to what we are working with. The standard library is stricter -- any attempt to use an Iterator object after any change is made in its backing list by another Iterator object will throw an Exception. This is called the **fail-fast principle**.

Listing 14.15 The DoublyLinked class of objects

```
public class DoublyLinked extends ListADT
{
   private Object itsData = null;
   private DoublyLinked itsNext = null;
   private DoublyLinked itsPrevious = null;


   /** Return the DoublyLinked object for which theRest is this
    *  list, if any. But return null if there is no such list. */

   public DoublyLinked theOneBefore()
   {  return itsPrevious;                                    //1
   }  //======================

   public void push (Object ob)
   {  DoublyLinked toAdd = new DoublyLinked();               //2
      toAdd.itsData = this.itsData;                          //3
      toAdd.itsNext = this.itsNext;                          //4
      this.itsData = ob;                                     //5
      this.itsNext = toAdd;                                  //6
      toAdd.itsPrevious = this;                              //7
      if (toAdd.theRest() != null)                           //8
         ((DoublyLinked)toAdd.theRest()).itsPrevious = toAdd;//9
   }  //======================

   public Object pop()
   {  Object valueToReturn = this.itsData;                   //10
      DoublyLinked toDiscard = this.itsNext;                 //11
      this.itsData = toDiscard.itsData;                      //12
      this.itsNext = toDiscard.itsNext;                      //13
      toDiscard.itsNext = null;   // make this list empty    //14
      toDiscard.itsPrevious = null;                          //15
      if (this.theRest() != null)                            //16
         ((DoublyLinked) this.theRest()).itsPrevious = this; //17
      return valueToReturn;                                  //18
   }  //======================

   // The following 3 methods are coded the same as for NodeList
   public ListADT theRest()
   public Object peekTop()
   public void setTop (Object ob)
}
```

Linked lists with header nodes

An irritating thing about a linked list with a trailer node is that, when you `push` a value at a particular node position, you have to first copy that node's data into the new node that you put after the current node. If you could just put the new data in the new node, that would save one assignment statement. And when you `pop` a value, you have to copy into the current node the data value from the node to discard; you could save two assignment statements if you could just return the data value in that discarded node.

The standard solution for this situation is to put every data value one node later in the linked list of nodes than the way we have been doing it. For instance, a NodeList with three data values A, B, C stores them in the first, second, and third nodes, with nothing in particular in the fourth node. Instead, we will put A in the second node, B in the third node, C in the fourth node, and nothing in particular (usually null) in the first node. The first node is then called a **header node**, since it is not a data-containing node and the rest are.

We will create a subclass of ListADT using this "more efficient" implementation and call it **HeaderList**. The coding for `theRest` is the same as in NodeList and DoublyLinked. But when an outside class calls the `peekTop` method for a HeaderList node, we have to return the data in the following node. Similarly, a call of HeaderList's `setTop` method by outside classes should put the data in the following node, but have no effect if there is no following node. These three methods are coded in the upper part of Listing 14.16.

You should compare the coding for `push` in the middle part of Listing 14.16 with the `push` coding for NodeList (which is repeated here for your convenience). The statements that are different are noted in comments:

```
   public void push (Object ob)  // for the NodeList class
   {  NodeList toAdd = new NodeList();  // equivalent to line 5
      toAdd.itsData = this.itsData;     // copied from other node
      toAdd.itsNext = this.itsNext;     // same as line 7
      this.itsData = ob;                // stored in current node
      this.itsNext = toAdd;             // same as line 8
   }  //======================
```

Listing 14.16 The HeaderList class of objects

```
public class HeaderList extends ListADT
{
   private Object itsData = null;
   private HeaderList itsNext = null;


   public ListADT theRest()
   {  return itsNext;                                         //1
   }  //======================

   public Object peekTop()        // throws Exception if empty
   {  return itsNext.itsData;                                 //2
   }  //======================

   public void setTop (Object ob)
   {  if (itsNext != null)                                    //3
         itsNext.itsData = ob;                                //4
   }  //======================

   public void push (Object ob)
   {  HeaderList toAdd = new HeaderList();                    //5
      toAdd.itsData = ob;                                     //6
      toAdd.itsNext = this.itsNext;                           //7
      this.itsNext = toAdd;                                   //8
   }  //======================

   public Object pop()
   {  HeaderList toDiscard = this.itsNext;                    //9
      this.itsNext = toDiscard.itsNext;                       //10
      toDiscard.itsNext = null;   // make this list empty     //11
      return toDiscard.itsData;                               //12
   }  //======================
}
```

A similar comparison shows that NodeList's `pop` has exactly the same four statements as HeaderList's `pop`, plus two additional assignments, so the coding for `pop` saves two assignments. However, every use of `peekTop` or `setTop` for a HeaderList object costs an extra referencing operation, and methods that progress through the list may use `peekTop` at every node. So perhaps the header node implementation executes more slowly than the trailer node implementation overall, for most software that uses ListADTs. Figure 14.6 shows the relations of the various classes so far that implement StackADT using Nodes.

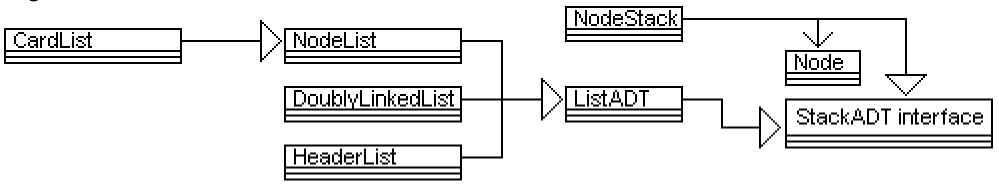


Figure 14.7 UML class diagrams for various implementations of StackADT

Further variations

If you use a ListADT to implement QueueADT and have `enqueue` call the `addLast` method, it has to go through all the nodes to get to the end. This can take a long time. But if you define a **ListQueue** subclass of HeaderList that stores in the data part of the header node a reference to the last node in the linked list, you can perform an `enqueue` operation extremely quickly, regardless of how many data values are in the list. ListQueue requires adding two more methods to the HeaderList class to allow access to the data part of the header node. This is left as a major programming project.

The hallmark of the ListADT class is the `theRest` method, which returns a sublist of the executor. Any change to the sublist causes the corresponding change in the executor's list. You might think that this capability forces you to have a linked list. But all the methods of ListADT can be implemented using a partially-filled array. Specifically, the analogue of a ListADT object would have two instance variables, one an object containing an array plus its size, the other keeping track of the current index in that array. Then `theRest` would return a new object with the same array/size object value but a different index. This is left as a major programming project.

Technical note You could simply make DoublyLinked a subclass of NodeList if the phrase `new NodeList()` in NodeList's `push` method produced a new DoublyLinked object rather than a NodeList object. But you can get the same effect if you replace that phrase by the following (using reflection): `this.getClass().getInstance()`.

Exercise 14.60 Write coding that swaps the data value in a DoublyLinked object named `sam` with the data value in the node before it, as long as both data values exist.
Exercise 14.61* Essay question: Since `theRest`, `setTop`, and `peekTop` are coded the same in both NodeList and DoublyLinked, why is it you cannot put them in a common superclass instead of having the duplication?
Exercise 14.62* Say you have a NodeList X with 6 data values, A,B,C,D,E,F in that order. Say you have four sublists cSub = X.theRest().theRest(), dSub = cSub.theRest(), eSub = dSub.theRest(), and fSub = eSub.theRest(). (a) What is the effect of dSub.pop() on each of the four sublists? (b) What is the effect of dSub.push(M) on each of the four sublists? (c) & (d) Same two questions except for HeaderList. Now maybe you can see why the Sun standard library specifies that this sort of thing is to throw an Exception.
Exercise 14.63** Revise DoublyLinked to be a subclass of NodeList by adding and using an instance method `cons` that in each class should return a newly-created node of that subclass type. How can you guarantee that each subclass of NodeList that overrides `cons` does not have it return a node that is already linked from some other node?

14.9 Applications: Josephus And The Many-Colored WebLinks

Modern personal computers use **timeslicing**: The user has several tasks going at the same time (usually with one active window per task, some of which may be showing on the monitor). The computer keeps all active tasks on a list. It gives a small amount of time to one task, then to the next task, then the next, and so on. It circles around to the original task when it finishes the last one.

The most convenient way to store the active tasks is in a circular list -- when you are at the last task and move on, you move on to the first task. In fact, it is not even necessary to have a "first task" and a "last task"; you just have a circle of tasks.

We could define a subclass of ListADT for which repetitions of `x = x.getRest()` never produce an empty list (so it could loop forever). But then it would not obey the **semantics** of StackADT (i.e., the meanings of the methods), even though it has the form required. Instead, we illustrate how to use a regular NodeList object to solve problems involving true circular lists, at only a small sacrifice in speed.

The Josephus Problem

The Josephus problem is a famous problem in algorithm design. It can best be explained as the solution to one-potato-two-potato: Children sometimes choose which of them is to be given a prize by what they think is a random process: They stand in a circle and count, "1 potato, 2 potato, 3 potato, 4; 5 potato, 6 potato, 7 potato, more; Y-O-U spells you!". This is a count of 13 around the circle. Then they eliminate the child it ends on. This is repeated until only one child is left, which is the child who gets the prize.

All you need do is create a Josephus object `sam` and add virtual children to it. Then the following statement prints the winner. Study the coding in Listing 14.17 to make sure you understand how it directly implements the algorithm sketched in the preceding paragraph.

```
   System.out.println (sam.josephus (13));
```

Listing 14.17 The JosephusList class of objects

```
public class JosephusList extends NodeList
{
   /** Remove every nth one circularly until only one is left.
    *  Return that one.  Precondition:  The list is not empty. */

   public Object josephus (int numToCount)
   {  ListADT circle = this;
      while ( ! this.theRest().isEmpty())  // more than 1 left
         circle = popAfterGoingForward (circle, numToCount);
      return this.peekTop();
   }  //======================


   private ListADT popAfterGoingForward (ListADT circle, int num)
   {  for (int k = 0;  k < num;  k++)
      {  circle = circle.theRest();
         if (circle.isEmpty())
            circle = this;
      }
      circle.pop();
      return circle;
   }  //======================
}
```

The web-link problem

A customer asks you to develop part of the software used to maintain an internet search engine. Your software is to roam the worldwide web and store information about which http links are on which pages. The client categorizes the webpages by their background colors, with a different non-negative integer value for each background color. Your software is to be able to accept a given color category, webpage, and http link and store that information. And it must be able to list all webpages found for a given color category, each with its own sublist of the http links on that page.

You design a WebData class of objects to have the following public methods:

```
new WebData (int numCategories)  // construct an object that keeps one
    // list of page/link associations for each of numCategories color categories.
void addLink (int category, Object page, Object link)
    // add that one page/link association to the data base for that category.
void listAll (int category) // list all page/link associations for the category.
```

For the internal design of the WebData class, it is appropriate to use an array of linked lists, one for each category. An array allows very fast access to the correct list. You then design a WebList class of linked list objects to have the following public methods:

```
void addLink (Object page, Object link)
    // add that one page/link association to the executor's list of associations.
void listAll()   // print all page/link associations in the executor's list.
```

This design so far makes it very easy to code the WebData class, calling on WebList methods as needed. Listing 14.18 has this coding, minus `listAll` (left as an exercise).

Listing 14.18 The WebData class of objects, partially done

```
public class WebData
{
   public final int MAX;
   private WebList[] itsItem;


   public WebData (int numCategories)
   {  MAX = (numCategories > 0) ? numCategories : 1;
      itsItem = new WebList [MAX];
      for (int k = 0;  k < MAX;  k++)
         itsItem[k] = new WebList();
   }  //======================

   /** Add the given page/link association of the given category.
    *  Ignore any category outside of the range 0..MAX-1. */

   public void addLink (int category, Object page, Object link)
   {  if (page != null && category >= 0 && category < MAX)
         itsItem[category].addLink (page, link);
   }  //======================
}
```

The WebList class

Further discussion with the client tells you that the information sent to a WebData object usually has several consecutive associations with the same page value, because the search of one particular page may yield half-a-dozen http links on it, producing half-a-dozen `addLink` calls in a row with the same page value.

Each WebList object (one per color category) contains many NodeLists of http links, one NodeList per page. To make it easy to tell which NodeList is for which page, you could store the page at the front of the NodeList and then all its http links after it. You decide to have each WebList object store in an instance variable `itsPrior` the NodeList for the page most recently retrieved from it. Initially, the NodeList value a WebList stores is assigned null, because no one has yet retrieved anything from it. The accompanying design block then provides a reasonable algorithm for the `addLink` method.

STRUCTURED NATURAL LANGUAGE DESIGN for WebList's addLink

```
1.  Let itsPrior denote the most recently retrieved item on this WebList.
2.  If itsPrior is null or its first data value is not the given page then...
        Assign to itsPrior the NodeList for the given page, creating it if need be.
3.  Insert the given http link second on the itsPrior list (since the given page is first).
```

The coding for the WebList class is in Listing 14.19 minus the `listAll` method, which is left as an exercise. Note that what we have for this software is an array of linked lists (WebLists) of data values, each data value being a linked list (a NodeList) of links. Interesting, isn't it?

Listing 14.19 The WebList class of objects

```java
/** A WebList contains zero or more non-empty NodeList objects.
 *  itsPrior is a reference to one of them if not null.  Each of
 *  those NodeLists contains 1 web page followed by its links. */

public class WebList extends NodeList
{
   private ListADT itsPrior = null;


   public void addLink (Object page, Object link)
   {  if (itsPrior == null || ! itsPrior.peekTop().equals (page))
         itsPrior = findMatchEvenIfYouHaveToMakeIt (page);
      itsPrior.theRest().push (link);  // push after the page
   }  //======================


   private ListADT findMatchEvenIfYouHaveToMakeIt (Object page)
   {  ListADT pos = this;
      for ( ; ! pos.isEmpty(); pos = pos.theRest())
      {  if (((ListADT) pos.peekTop()).peekTop().equals (page))
            return (ListADT) pos.peekTop();
      }
      ListADT toAdd = new NodeList();
      toAdd.push (page);
      pos.push (toAdd);  // putting it at the end of this WebList
      return toAdd;
   }  //======================
}
```

Exercise 14.64 Write the WebData method `public void listAll (int category)`.
Exercise 14.65* Write the WebList method `public void listAll()`.

*14.10 About Stack And Vector (*Sun Library)*

The **java.util.Vector** class was the original class designed by Sun Microsystems, Inc., to create and modify a collection of Objects. It has been replaced by ArrayList, and its use is now discouraged. Programmers are strongly encouraged to use ArrayList or something else from the Collection or Map hierarchy in new software they write, but Vector will be retained in the Sun standard library so that existing software will continue to function correctly.

You should be aware of the Vector class because, if you find occasion to maintain older software written in Java, you will often find it using the Vector class. Vector has been "retrofitted" to have all of the methods required by the List interface. But it still provides the following antiquated methods in additon (indexes are zero-based):

- `new Vector()` creates an empty Vector.
- `new Vector(someInt)` creates an empty Vector with a capacity of `someInt`. The capacity will increase if enough elements are added to the Vector.
- `someVector.size()` is the number of elements actually in the Vector.
- `someVector.elementAt(indexInt)` is `get(indexInt)`.
- `someVector.setElementAt(ob, indexInt)` replaces the element at the given index with `ob`. It throws an Exception if the index is negative or greater than `size() - 1`.
- `someVector.addElement(ob)` adds `ob` to the end of the list.
- `someVector.removeElementAt(indexInt)` is `remove(indexInt)`.
- `someVector.removeAllElements()` is `clear()`.
- `someVector.insertElementAt(ob, indexInt)` is `add(indexInt, ob)`.
- `someVector.firstElement()` returns the element at index 0.
- `someVector.lastElement()` returns the element at index `someVector.size()-1`.
- `someVector.ensureCapacity(maxCapacity)` extends the underlying array, if necessary, to have at least `maxCapacity` components.
- `someVector.elements()` returns an Enumeration object, which is the earlier version of Iterator (described in Chapter 15).

Stack objects (from java.util)

The Stack class is a subclass of Vector. It has the following methods. Both `pop` and `peek` throw a java.util.EmptyStackException if the Stack is empty:

- `new Stack()` creates an empty Stack.
- `someStack.push(someObject)` adds the object to the top of the Stack.
- `someStack.pop()` removes and returns the object on top of the Stack.
- `someStack.peek()` returns the object that is on top of the Stack.
- `someStack.empty()` tells whether the stack contains no objects.
- `someStack.search(someObject)` returns 1 if the object is on top, 2 if it is second, 3 if it is third, etc. It returns -1 if the object is not in the Stack.

14.11 Review Of Chapter Fourteen

- Stacks and queues are data structures that allow you to add an element, remove a particular element, see if they are empty, or see what you would get if you removed an element. The particular element you get depends on the structure: A **stack** gives you the element that has been there the shortest period of time, and a **queue** gives you the element that has been there the longest period of time.
- This book defines two similar interfaces, both with an `isEmpty()` method plus a method to add an element, a method to remove the required element, and a method to see what element would be removed. The **StackADT** interface has methods `push(ob)`, `pop()`, and `peekTop()`. The **QueueADT** interface has methods `enqueue(ob)`, `dequeue()`, and `peekFront()`.
- **Postfix algebraic notation** puts each binary operator directly after the two operands. As a consequence, it is never necessary to parenthesize the expression; a legal postfix expression can only be interpreted in one way. Evaluation of a postfix expression is most easily done with a stack.
- **Efficiency** is a composite of space (RAM usage), time (speed of execution), and effort (the work the programmer does to develop and maintain the coding).
- This chapter defined two StackADT implementations, ArrayStack and NodeStack. It also defined two QueueADT implementations, ArrayQueue and NodeQueue.
- A **list** structure allows you to inspect the elements at any position in the list. Most list structures also allow you to add or remove elements at any position. This chapter presents several ways of implementing the **ListADT** interface. Its hallmark is a method call **`theRest`** which returns, for any non-empty list, the sublist containing all its elements except the first. We can add to it the capability to move backwards in the list if we use a **doubly-linked list**.

Answers to Selected Exercises

14.1
```
public static Object removeSecond (StackADT stack)
{    if (stack.isEmpty())
          return null;
     Object first = stack.pop();
     Object valueToReturn = stack.isEmpty()  ?  null : stack.pop();
     stack.push (first);
     return valueToReturn;
}
```
14.2 The values are (12 / (10 - (3 + 5))) = 6 and (((5 - 4) - ((3 - 2) - 1)) = 1.

14.3
```
public static void removeDownTo (StackADT stack, Object ob)
{    if (ob == null)
     {    while ( ! stack.isEmpty() && stack.peekTop() != null)
               stack.pop();
     }
     else
     {    while ( ! stack.isEmpty() && ! ob.equals (stack.peekTop()))  // not the opposite order
               stack.pop();
     }
}
```
14.4
```
public static Object removeSecond (QueueADT queue)
{    Object firstOne = queue.dequeue();
     Object valueToReturn = queue.dequeue();
     Object endMarker = new Double (0.0);  // anything brand-new works here
     queue.enqueue (endMarker);
     queue.enqueue (firstOne);
     while (queue.peekFront() != endMarker)  // you cannot use the .equals test here
          queue.enqueue (queue.dequeue());
     queue.dequeue();  // remove the endMarker, so firstOne is now at the front
     return valueToReturn;
}
```

```
14.9    The declaration of itsTop should be:   private int itsTop = -1;
        Replace itsSize by itsTop+1 everywhere else in the coding except where you apply ++ or --.
14.10   public boolean equals (Object ob)
        {    if ( (! (ob instanceof ArrayStack))  ||  (this.itsSize != ((ArrayStack) ob).itsSize))
                 return false;
             ArrayStack given = (ArrayStack) ob;   // for efficiency
             for (int k = 0;  k < this.itsSize;  k++)
             {    if ( ! this.itsItem[k].equals (given.itsItem[k]))
                      return false;
             }
             return true;
        }
14.13   public int size()
        {    return itsRear - itsFront + 1;
        }
14.14   public String toString()
        {    String valueToReturn = "";
             for (int k = itsFront;  k <= itsRear;  k++)
                 valueToReturn += '\t' + itsItem[k];  // uses toString() if not null, "null" if it is null
             return valueToReturn;
        }
14.15   public void removeAfter (Object ob)
        {    int spot = itsRear;  // we will set spot to the location of ob in the queue, if ob is there
             if (ob == null)
             {    while (spot >= itsFront &&  itsItem[spot] != null)
                      spot--;
             }
             else
             {    while (spot >= itsFront &&  ! ob.equals (itsItem[spot]))
                      spot--;
             }
             if (spot >= itsFront)      // then spot is the last location of ob in the queue
                 itsRear = spot;     // no need to actually erase the deleted values
        }
14.21   public Object last()
        {    return isEmpty()  ?  null  :  itsRear.getValue();
        }
14.22   public void dup()
        {    if (isEmpty())
                 throw new IllegalStateException ("stack is empty");
             itsTop = new Node (itsTop.getValue(), itsTop);
        }
14.23   public void enqueue (Object ob)
        {    if (isEmpty())
             {    itsFront = new Node (ob, null);
                  itsRear = itsFront;
             }
             else
             {    itsRear.setNext (new Node (ob, null));
                  itsRear = itsRear.getNext();
             }
        }
14.24   public int size()
        {    int count = 0;
             for (Node p = itsFront;  p != null;  p = p.getNext())
                 count++;
             return count;
        }
14.25   public void append (NodeQueue queue)
        {    if (queue.isEmpty())
                 return;
             this.itsRear.setNext (queue.itsFront);
             this.itsRear = queue.itsRear;
             queue.itsFront = null;
        }
14.26   Insert after the return statement in the preceding method coding these 3 lines:
        if (this.isEmpty())
             this.itsFront = queue.itsFront;
        else
14.31   public void clear()   // in ListADT
        {    while ( ! isEmpty())
                 pop();
        }
```

14.32
```
public void clear()     // in NodeList
{       itsNext = null;
}
```
14.33
```
public void copyTo (ListADT par)  // in ListADT
{       if ( ! this.isEmpty())
        {       par.push (this.peekTop ());
                this.theRest().copyTo (par.theRest());
        }
}
```
14.39
```
public Object get (int index)
{       return (index == 0)  ?  peekTop()  :  theRest().get (index - 1);
}
```
14.40
```
public void setLast (Object ob)
{       if (theRest().isEmpty())
                setTop (ob);
        else
                theRest().setLast (ob);
}
```
14.41
```
public static int numNulls (ListADT list)
{       int count = 0;
        for ( ; ! list.isEmpty();  list = list.theRest())
        {       if (list.peekTop() == null)
                        count++;
        }
        return count;
}
```
14.42
```
public boolean containsAsSublist (ListADT par)
{       return this == par  ||  (! this.isEmpty() && this.theRest().containsAsSublist (par));
}
```
14.43
```
public void add (int index, Object ob)
{       if (index < 0)
                throw new IndexOutOfBoundsException ("index cannot be negative");
        ListADT position = this;
        for ( ;  index > 0;  index--)
                position = position.theRest();
        position.push (ob);  // throws an Exception if index > size()
}
```
14.44
```
public boolean equals (ListADT that)
{       if (this.isEmpty())
                return that.isEmpty();  // i.e., both are empty
        else if (that.isEmpty())
                return false;
        else
                return this.peekTop().equals (that.peekTop())
                                && this.theRest().equals (that.theRest());
}
```
14.53
```
public Object removeSmallest()
{       Comparable smallestSoFar = (Comparable) this.peekTop();
        ListADT spot = this;
        for (ListADT p = this.theRest();  ! p.isEmpty();  p = p.theRest())
        {       if (smallestSoFar.compareTo (p.peekTop()) > 0)
                {       smallestSoFar = (Comparable) p.peekTop();
                        spot = p;
                }
        }
        return spot.pop();   // which is in fact smallestSoFar
}
```
14.54 Code findLargest the same as removeSmallest in the preceding exercise except:
(a) remove the two statements that assign a value to spot.
(b) change the last statement to return largestSoFar.
(c) rename smallestSoFar as largestSoFar and change "> 0" to "< 0".

14.60
```
if (sam.isEmpty() || sam.theOneBefore() == null)
        return;
Object saved = sam.peekTop();
sam.setTop (sam.theOneBefore().peekTop());
sam.theOneBefore().setTop (saved);
```
14.64
```
public void listAll (int category)
{       if (category >= 0 && category < itsItem.length)
                itsItem[category].listAll();
}
```