CHAPTER

7 Queues

He who comes first, eats first.

EIKE VON REPKOW
Sachsenspiegel

A *queue* is a first-in/first-out data structure similar to a line of people at a ticket window. It can be used whenever you need a data structure that allows items to "wait their turn." Impressive algorithms that explore mazes of interconnected rooms use queues to keep track of which options have not yet been explored (though, of course, computer scientists have their own terminology—you'll see it as *breath-first search* in Chapter 14—for such problems). In this chapter, we discuss applications of the queue data structure, give two implementations of a queue ADT, and discuss the differences between queues and stacks. We also discuss and implement a more flexible kind of queue called a *priority queue*.

7.1 INTRODUCTION TO QUEUES

The word "queue" is pronounced as if you were saying the letter Q; it means the same thing as the word "line" when used in phrases like "waiting in a line." Every time you get in line at a supermarket or a bank or a ticket window, you are adding yourself to a queue. If everybody is polite, then people add themselves to the rear of the queue (the rear of the line), and the person at the front of the queue is always the person who is served first. The queue data structure works in the same way, and the abstract definition of a queue reflects this intuition.

Queue Definition

A **queue** is a data structure of ordered items such that items can be inserted only at one end (called the **rear**) and removed at the other end (called the **front**). The item at the front end of the queue is called the **first item**.

When we say that the items in a queue are *ordered,* we refer to the items' position in the queue. There is a first one (the front one), a second one, a third one, and so forth. We do *not* require that the items can be compared using the < operator. The items can be of any type. In this regard, the situation is the same as it was for a stack.

stacks versus queues

Because items must be removed in exactly the same order as they were added to the queue, a queue is called a first-in/first-out data structure (abbreviated FIFO). This differs from a stack, which is a last-in/first-out data structure, but apart from this difference, a stack and a queue are similar data structures. They differ only in the rule that determines which item is removed first. The contrast between stacks and queues is illustrated in Figure 7.1. In either structure, the items depicted are entered in the order A, B, C, and D. With a queue, they are removed in the same order: A, B, C, D. With a stack, they are removed in the reverse order: D, C, B, A.

FIFO

A queue is a first-in/first-out data structure. Items leave the queue in the same order in which they were put into the queue.

The Queue Class

In Figure 7.2, we specify the key methods of a queue. The specification follows the same general pattern as the generic stack of Chapter 6. As with our stack, the queue is a generic queue of objects, but this underlying type can be changed to any primitive data type without problems. Later we will implement the queue as a Java class in two different ways: The first will store the items in an array, whereas the second uses a linked list.

The queue's methods include operations to remove the item from the front of the queue, to add an item to the rear of the queue, and so on. The method for adding an item to the queue is often called an **insert** or **enqueue** operation (pronounced *en-q*), which is a shortening of "*en*tering the *queue*." But for adding an item, we will use the alternative name add, which is the name that Java uses. The method for removing an item from the queue is often called a **getFront** or **dequeue** operation (pronounced *de-q*), which means "*de*leting from the *queue*," but we will use the Java's name `remove`.

If a program attempts to remove an item from an empty queue, that is a kind of error called **queue underflow**. This is similar to a stack underflow (popping an empty stack), though there is one difference. Java provides a specific exception, `EmptyStackException`, to indicate a stack underflow. There is no specific exception for a queue underflow, so our `remove` method will throw `NoSuchElementException` to indicate an underflow. (`NoSuchElementException` is part of `java.util`, and many collection classes use it to indicate some kind of underflow.) Anyway, to help you avoid these errors, the queue class provides a boolean method to test for an empty queue and a second method to return the current number of items in the queue. (Java's queue also provides two alternative methods, `offer` and `poll`, that a queue can use instead of `add` and `remove`. The alternatives have a different kind of failure behavior that does not throw exceptions.)

FIGURE 7.1 Contrasting a Stack and a Queue

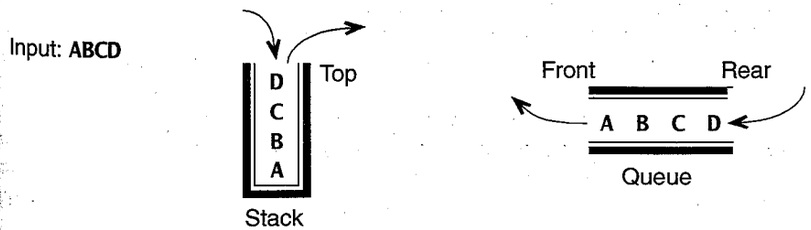


FIGURE 7.2 Specification of the Key Methods of a Generic Queue Class

Specification

These are the key methods of a typical queue class, which is based on Java's generic `Queue<E>` interface. Although this is a specification for a generic queue, we could also have a queue that contains primitive values (such as `int`) directly.

- **Constructor for the Generic Queue<E>**
  `public Queue( )`
  Initialize an empty queue.
  Postcondition:
  This queue is empty.

- **add**
  `public void add(E item)`
  Add a new item to the rear of this queue. The new item may be the null reference.
  Parameters:
  `item` – the item to be added to this queue
  Postcondition:
  The item has been added to the rear of this queue.
  Throws: `OutOfMemoryError`
  Indicates insufficient memory for adding a new item to this queue.
  Note:
  Java's version of this method returns a `boolean` value that is always true for a queue.

- **isEmpty**
  `public boolean isEmpty( )`
  Determine whether this queue is empty.
  Returns:
  `true` if this queue is empty; otherwise, `false`

- **remove**
  `public E remove( )`
  Get the front item, removing it from this queue.
  Precondition:
  This queue is not empty.
  Postcondition:
  The return value is the front item of this queue, and the item has been removed.
  Throws: `NoSuchElementException`
  Indicates that this queue is empty.

- **size**
  `public int size( )`
  Accessor method to determine the number of items in this queue.
  Returns:
  the number of items in this queue

We could add other operations, such as a method to look at the front item of the queue without removing it from the queue, but this minimal set of operations will be enough for our purposes.

Uses for Queues

Uses for queues are easy to find—we often use queues in our everyday affairs, such as when we wait in line at a bank. To get a feel for using a queue in an algorithm, we will first consider a simple example.

copying a word

Suppose you want a program to read a word and then write the word. This is so simple that you may wonder why we bother to consider this task, but it is best to start with a simple example. One way to accomplish this task is to read the input one letter at a time and place each letter in a queue. After the word is read, the letters in the queue are written out. Because a queue is a first-in/first-out data structure, the letters are written in the same order in which they were read. Here is the pseudocode for this approach:

```
// Echoing a word
1. Declare a queue of characters.
2. while (there are more characters of the word to read)
   {
     Read a character.
     Insert the character into the queue.
   }
3. while (the queue is not empty)
   {
     Get the front character from the queue.
     Write the character to the screen.
   }
```

simulation programs

Because queues occur in real-life situations, they are frequently used in simulation programs. For example, a program to simulate the traffic at an intersection might use a software queue to simulate the real-life situation of a growing line of automobiles waiting for a traffic light to change from red to green.

input/output buffering

Queues also appear in computer system software, such as the operating system that runs on your PC. For example, consider a program that reads input from the keyboard. We think of a program as directly reading its input from the keyboard. However, if you think of what actually happens when you give a line of input to a program, you will realize that the program does not necessarily read a character when the corresponding keyboard key is pressed. This allows you to type input to the program, and that input is saved in a queue by software that is part of the operating system. When the program asks for input, the operating system provides characters from the front of its queue. This is called **buffering** the input and is controlled by the PC's operating system software. In reality, a more sophisticated data structure is used rather than a queue, allowing you to back up

and retype part of the line. Also, this form of buffering data in a queue is often used when one computer component is receiving data from another, faster computer component. For example, if your fast CPU (central processor unit) is sending data to your printer, which is slow by comparison, then the data is buffered in a queue. By using the queue in this way, the CPU need not wait for the printer to finish printing the first character before the CPU sends the next character.

Self-Test Exercises for Section 7.1

1. What are the meanings of LIFO and FIFO?
2. What are the traditional names for the queue operations that add an item and remove an item from a queue?
3. What queue method can a programmer use to avoid queue underflow?
4. Suppose a program uses a queue of characters to read a word and echo it to the screen. For the input word `LINE`, trace the algorithm, giving all the activations of the operations `add` and `remove`.
5. Name some common situations in which a PC's operating system uses some kind of a queue.
6. Write pseudocode for an algorithm that reads an even number of characters. The algorithm then prints the first character, third character, fifth character, and so on. On a second output line, the algorithm prints the other characters. Use two queues to store the characters.

7.2 QUEUE APPLICATIONS

Java Queues

Before we actually implement the queue class, we'll show two applications that use a generic queue. In particular, our first application will use a Java queue that is declared in this way:

how to declare a simple Queue in Java

```
import java.util.LinkedList;
import java.util.Queue;
...
Queue<Character> q = new LinkedList<Character>( );
```

This looks unusual because the data type of the variable is `Queue<Character>`, but we are calling Java's `LinkedList` constructor. The reason is that Java does not actually have a separate `Queue` class. Instead, it has a `Queue` interface (as described in Section 5.5). Since Queue is only an interface, we cannot create Queue objects in Java, but there are several other Java classes (such as LinkedList) that implement the Queue interface. This interface includes all of the methods from Figure 7.2 on page 352 (as well as many other methods).

In any case, if you want a simple Java queue, consider using the `Queue` interface and the `LinkedList` class as shown on the previous page.

PROGRAMMING EXAMPLE: Palindromes

A **palindrome** is a string that reads the same forward and backward; that is, the letters are the same whether you read them from right to left or from left to right. For example, the one-word string "radar" is a palindrome. A more complicated example of a palindrome is the following sentence:

```
Able was I ere I saw Elba
```

Palindromes are fun to make up, and they even have applications in at least one area—the analysis of genetic material.

Suppose we want a program to read a line of text and tell us if the line is a palindrome. We can do this by using both a stack and a queue. We will read the line of text into both a stack and a queue and then write out the contents of the stack and the contents of the queue. The line that is written using the queue is written forward, and the line that is written using the stack is written backward. Now, if those two output lines are the same, then the input string must be a palindrome. Of course, the program need not actually write out the contents of the stack and the queue. The program can simply compare the contents of the stack and the queue character by character to see if they would produce the same string of characters.

A program that checks for palindromes in the way we just outlined is given in Figure 7.3. This program uses Java's stack class defined in Figure 6.1 on page 306 as well as a queue that is declared using the technique on page 354. In this program, we treat both the uppercase and lowercase versions of a letter as being the same character. This is because we want to consider a sentence as reading the same forward and backward even though it might start with an uppercase letter and end with a lowercase letter. For example, the string "Able was I ere I saw Elba" when written backward reads "ablE was I ere I saw elbA". The two strings match, provided we agree to consider upper- and lowercase versions of a letter as being equal. The treatment of the letters' cases is accomplished with a useful Java method `Character.toUpperCase`. This method has one character as an argument. If this character is a lowercase letter, then the method converts it to the corresponding uppercase letter and returns this value. Otherwise, the method returns the character unchanged.

the toUpperCase method converts lowercase letters to uppercase letters

Our program also ignores many characters, requiring only that the *alphabetic letters* on the line read the same forward and backward. This way, we can find more palindromes. If we look only at alphabetic letters, discarding spaces and punctuation, then there are many more palindromes. However, they are not always easy to spot. For example, you might not immediately recognize the following as a palindrome:

```
Straw? No, too stupid a fad. I put soot on warts.
```

the isLetter method determines which characters are letters

Depending on your current frame of mind, you may think that discovering such sentences is a stupid fad, but nevertheless, our program ignores blanks and punctuation. Therefore, according to our program, the preceding is a palindrome. The determination of whether a character is a letter is accomplished with another Java method. The method, called `Character.isLetter`, returns `true` if its single argument is one of the alphabetic characters.

A sample dialogue from the palindrome program is shown at the end of Figure 7.3. As we often do, we have presented a minimal program to concentrate on the new material being presented. Before the program is released to users, it should be enhanced in a number of ways to make it more robust (such as better handling of a possible stack or queue overflow) and more friendly (such as allowing more than one sentence to be entered).

FIGURE 7.3 A Program to Recognize Palindromes

Java Application Program

```
// FILE: Palindrome.java
// This program reads strings from the keyboard, and determines whether each input
// line is a palindrome. The program ignores everything except alphabetic letters, and it
// ignores the difference between upper- and lowercase letters.

import java.util.LinkedList; // See page 354 for why we use a Linked List
import java.util.Queue;      // Provides the Queue interface
import java.util.Stack;      // Provides the Stack class
import java.util.Scanner;    // Provides the Scanner class (see Appendix B)

public class Palindrome
{
   public static void main(String[ ] args)
   {
      Scanner stdin = new Scanner(System.in); // Keyboard input
      String line;                            // One input line

      do
      {
         System.out.print("Your expression (or return to end): ");
         line = stdin.nextLine( );
         if (is_palindrome(line))
            System.out.println("That is a palindrome.");
         else
            System.out.println("That is not a palindrome.");
      }
      while (line.length( ) != 0);
   }
```

(continued)

(FIGURE 7.3 continued)

```
   public static boolean is_palindrome(String input)
   // The return value is true if and only if the input string is a palindrome.
   // All non-letters are ignored, and the case of the letters is also ignored.
   // See page 354 for an explanation of using Java's LinkedList class as a Queue.
   {
      Queue<Character> q = new LinkedList<Character>( );
      Stack<Character> s = new Stack<Character>( );
      Character letter;    // One character from the input string
      int mismatches = 0; // Number of spots that mismatched
      int i;              // Index for the input string

      for (i = 0; i < input.length( ); i++)
      {
         letter = input.charAt(i);
         if (Character.isLetter(letter))
         {
            q.add(letter);
            s.push(letter);
         }
      }

      while (!q.isEmpty( ))
      {
         if (q.remove( ) != s.pop( ))
            mismatches++;
      }

      // If there were no mismatches, then the string was a palindrome.
      return (mismatches == 0);
   }

}
```

Part of a Sample Dialogue

```
Your expression (or return to end):
Straw? No, too stupid a fad. I put soot on warts.
That is a palindrome.
Your expression (or return to end):
Able were you ere you saw Elba.
That is not a palindrome.
```

advantages of using existing implementations

You may have thought of other ways to solve the palindrome problem. (If not, give it a try now!) Your alternative solution may be simpler in some ways than the solution that uses a stack and a queue, but using the stack and queue has another advantage to keep in mind: Once the stack and queue classes are implemented, they can be used in many algorithms without worrying about their implementation details. This is a primary advantage of using an existing implementations of abstract data types.

PROGRAMMING EXAMPLE: Car Wash Simulation

The *Handy-Dandy Hand Car Wash Company* has decided to modernize and change its image. It has installed a fast, fully automated, car-washing mechanism that can wash a car in one to ten minutes. It will soon reopen under its new name, *The Automatic Auto Wash Emporium*. The company wants to know the most efficient way to use its new car-washing mechanism. If the mechanism is used on the fast setting, it can wash a car in one minute, but because of the high pressure required to operate at such speed, the mechanism uses a great deal of water and soap at this setting. At slower settings, it takes longer to wash a car but uses less soap and water. The company wants to know how many customers will be served and how long customers will have to wait in line when the washing mechanism is used at one of the slower speeds. The company also wants to know whether its new motto, "You Ought to Autowash your Auto," will be effective. We respectfully refuse comment on the motto, but we agree to write a program that will simulate automobiles waiting in line for a car wash. This way, the manager of the car wash can see how the speed of the car wash, the length of the line, and various other factors interact.

Car Wash Simulation—Specification

Our specification for the simulation consists of input and output descriptions.

Input. The program has three input items: (1) the amount of time needed to wash one car (in total seconds); (2) the probability that a new customer arrives during any given second (we assume that, at most, one customer arrives in a second); and (3) the total length of time to be simulated (in seconds).

Output. The program produces two pieces of output information: (1) the number of customers serviced in the simulated time; and (2) the average time that a customer spent in line during the simulation (in seconds).

Car Wash Simulation—Design

We will carry out a design of the program in a way that is common for many simulation tasks. The approach is to propose a collection of related object types that correspond to real-world objects in the situation we are simulating. There

are many possibilities—our particular approach focuses on the use of our queue class, which will be used to simulate a line of customers waiting to have their cars washed. We first discuss the queue and then go on to propose the other objects needed for the simulation.

the queue

We need to simulate a queue of customers, but we do not have real live customers, so we must decide how we will represent them. There are many ways to represent customers: We could use their names and place them in a queue of names; we could assign an arbitrary number to each customer and store that number in a queue of numbers; we could represent each customer by the make and year of the customer's automobile or even by how dirty the automobile is. However, none of these representations has any relevance to the specified simulation. For this simulation, all we need to know about a customer is how long the customer waits in the queue. Hence, a good way to represent a customer is to use a number that represents the time at which the customer entered the queue. Thus, our queue will be a queue of numbers. In a more complex simulation, it would be appropriate to implement the customers as objects, and one of the customer's methods would return the customer's arrival time.

Key Design Concept

Determine which properties of a real-world object are relevant to the problem at hand.

The numbers that record the times at which customers enter the queue are called **time stamps**. Our simulation works in seconds, so a time stamp is just the number of simulated seconds that have passed since the start of the simulation. When the customer (represented by the time stamp) is removed from the queue, we can easily calculate the time the customer spent waiting: The time spent waiting is the total number of seconds simulated so far minus the time stamp.

propose a list of related object types

Now let's discuss other objects that will be useful in the simulation program. In addition to the queue, we propose three other object types, listed here:

Washer. A washer is an object that simulates the automatic car washing mechanism.

BooleanSource. An object of this class provides a sequence of boolean values. Some of the values in the sequence are `true`, and some are `false`. During the simulation, we will have one `BooleanSource` object that we query once per simulated second. If the query returns `true` as its response, this indicates that a new customer has arrived during the simulated second; a `false` return value indicates that no customer has arrived during the simulated second.

Averager. An averager computes the average of a group of numbers. For example, we might send the following four numbers into an averager: 10, 20, 2, and 12. The averager could then tell us that the average of these numbers is 11.0. The averager can also tell us how many numbers it has processed—in our example, the averager processed four numbers. We'll use an averager to keep track of the average waiting time and the total number of cars washed.

write pseudocode indicating how the objects are used

A good way to design a simulation program is to first write pseudocode illustrating how the proposed objects will be used. From this pseudocode, we then extract the necessary methods for each object. The simulation program we have in mind will carry out the simulation using one-second intervals, and to make the

FIGURE 7.4 The Car Wash Simulation

Pseudocode

1. Declare a queue of integers, which will be used to keep track of arrival times of customers who are waiting to wash their cars. We also declare the following objects:

   (a) A `Washer`: The washer's constructor has an argument indicating the amount of time (in seconds) needed by the washer to wash one car.

   (b) A `BooleanSource`: The constructor has an argument that specifies how frequently the `BooleanSource` returns `true` (indicating how often customers arrive).

   (c) An `Averager`.

2. `for (currentSecond = 0; currentSecond <` the simulation length`; currentSecond++)`
   `{`
   Each iteration of this loop simulates the passage of one second of time, as follows:
   Ask the `BooleanSource` whether a new customer arrives during this second, and if so, enter the `currentSecond` into the queue.
   `if` (the `Washer` is not busy and the queue is not empty)
   `{`
   Remove the next integer from the queue and call this integer `next`.
   This integer is the arrival time of the customer whose car we will now wash.
   So, compute how long the customer had to wait (`currentSecond - next`) and send this value to the `Averager`. Also, tell the `Washer` that it should start washing another car.
   `}`
   Indicate to the `Washer` that another simulated second has passed. This allows the `Washer` to correctly determine whether it is still busy.
   `}`

3. At this point, the simulation is completed. We can get and print two items of information from the `Averager`: (1) how many numbers the `Averager` was given (i.e., the number of customers whose cars were washed); and (2) the average of all the numbers it was given (i.e., the average waiting time for the customers, expressed in seconds).

simulation simpler, we assume that at most one customer arrives during any particular second. The simulation contains a loop that is iterated once for each simulated second. In each loop iteration, all the activities that take place in one second, such as the possible arrival of a new customer or the removal of a customer from the queue, will be simulated. When the loop terminates, the simulation is over, and the needed output values are obtained from the averager. The basic pseudocode for the simulation is shown in Figure 7.4.

BooleanSource. During each simulated second, the `BooleanSource` provides us with a single boolean value indicating whether a new customer has arrived

during that second. A `true` value indicates that a customer has arrived; `false` indicates that no customer arrived. With this in mind, we propose two methods: a constructor and a method called `query`.

The constructor for the `BooleanSource` has one argument, which is the probability that the `BooleanSource` returns `true` to a query. The probability is expressed as a decimal value between 0 and 1. For example, suppose our program uses the name `arrival` for its `BooleanSource`, and we want to simulate the situation in which a new customer arrives during 1% of the simulated seconds. Then our program would have the following declaration:

```
BooleanSource arrival = new BooleanSource(0.01);
```

The second method of the `BooleanSource` can be called to obtain the next value in the `BooleanSource`'s sequence of values. Here is the specification:

♦ **query (method of the BooleanSource)**
`public boolean query( )`
Get the next value from this `BooleanSource`.
Returns:
The return value is either `true` or `false`; the probability of a `true` value is determined by the argument that was given to the constructor.

There are several ways of generating random boolean values, but at this specification stage, we don't need to worry about such implementation details.

Averager. The averager has a constructor that initializes the averager so that it is ready to accept numbers. The numbers will be given to the averager one at a time through a method called `addNumber`. For example, suppose our averager is named `waitTimes`, and the next number in the sequence is 10. Then we will activate `waitTimes.addNumber(10)`; the averager also has two methods to retrieve its results: `average` and `howManyNumbers`, as specified here:

♦ **Constructor for the Averager**
`public Averager( )`
Initialize an `Averager` so that it is ready to accept numbers.

♦ **addNumber**
`public void addNumber(double value)`
Give another number to this `Averager`.

♦ **average**
`public double average( )`
The return value is the average of all numbers given to this `Averager`.

♦ **howManyNumbers**
`public int howManyNumbers( )`
Provide a count of how many numbers have been given to this `Averager`.

Notice that the argument to `addNumber` is actually a double number rather than an integer. This will allow us to use the averager in situations in which the

sequence is more than just whole numbers. Also, the class will have some limitations (for example, the total count of numbers given to an averager cannot exceed `Integer.MAX_VALUE`). But we'll deal with those limitations during the implementation stage.

Washer. The simulation program requires one washer object. This washer is initialized with its constructor, and each time another second passes, the simulation program indicates the passage of a second to the washer. This suggests the following constructor and method:

- **Constructor for the Washer**

  `public Washer(int s)`

  Initialize a `Washer`.

  Postcondition:

  This `Washer` has been initialized so that it takes `s` seconds to complete one wash cycle.

- **reduceRemainingTime**

  `public void reduceRemainingTime( )`

  Reduce the remaining time in the current wash cycle by one second.

  Postcondition:

  If a car is being washed, then the remaining time in the current wash cycle has been reduced by one second.

The other two responsibilities of a washer are to tell the simulation program whether the washing mechanism is currently available and to begin the washing of a new car. These responsibilities are accomplished with two methods:

- **startWashing**

  `public void startWashing( )`

  Start a wash cycle for this `Washer`.

  Precondition:

  `isBusy( )` is `false`.

  Postcondition:

  This `Washer` has started simulating one wash cycle.

  Throws: `IllegalStateException`

  Indicates that this `Washer` is busy.

- **isBusy**

  `public boolean isBusy( )`

  Determine whether this `Washer` is currently busy.

  Returns:

  `true` if this `Washer` is busy (in a wash cycle); otherwise `false`.

Car Wash Simulation—Implementing the Car Wash Classes

We have completed a specification for the three new classes that will be used in the car wash simulation. We'll implement these three classes in separate files: `Averager.java` (Figure 7.5 on page 365), `BooleanSource.java` (Figure 7.6 on page 367), and `Washer.java` (Figure 7.7 on page 368). All three are part of the package `edu.colorado.simulations`. The implementations are straightforward, but we'll provide a little discussion.

Implementation of the Averager. The implementation of the averager is a direct implementation of the definition of "average" and some straightforward details. The class has two instance variables: one to keep track of how many numbers the averager has been given and another to keep track of the sum of all those numbers. When the `average` method is activated, the method returns the average calculated as the sum of all the numbers divided by the count of how many numbers the averager was given.

Notice that the averager does *not* need to keep track of all the numbers individually. It is sufficient to keep track of the sum of the numbers and the count of how many numbers there were. However, the specification does indicate some limitations connected with possible arithmetic overflows.

Implementation of the BooleanSource. The `BooleanSource` class has one instance variable, `probability`, which stores the probability that an activation of `query` will return `true`. The implementation of the `query` method first uses the `Math.random` method to generate a random number between 0 and 1 (including 0 but not 1). Hence, if the instance variable `probability` is the desired probability of returning `true`, then `query` should return `true` provided the following relationship holds: `Math.random( ) < probability`.

For example, suppose we want a 10% chance that `query` returns `true` so that `probability` is `0.1`. If `random` returns a value less than `0.1`, then `query` will return `true`. The chance that `rand` returns a value less than `0.1` is 10% since `0.1` marks a point that is 10% of the way through `random`'s output range. Therefore, there is about a 10% chance that `Math.random( ) < probability` will be `true`. It is this boolean expression that is used in the return statement of `query`.

Implementation of the Washer. The `Washer` class has two instance variables. The first instance variable, `secondsForWash`, is the number of seconds needed for one complete wash cycle. This variable is set by the constructor and remains constant thereafter. The second instance variable, `washTimeLeft`, keeps track of how many seconds until the current wash is completed. This value can be zero if the washer is not currently busy.

The washer's `reduceRemainingTime` method is activated to subtract one second from the remaining washing time. So the `reduceRemainingTime` method checks whether a car is currently being washed. If there is a car being washed, then the method subtracts one from `washTimeLeft`.

The washer's `isBusy` method simply checks whether `washTimeLeft` is greater than zero. If so, there is a car in the washing mechanism. Otherwise, the washing mechanism is ready for another car.

When the car-washing mechanism is not busy, the `startWashing` method can be activated to start another car through the washer. The method starts the wash by setting `washTimeLeft` equal to `secondsForWash`.

Car Wash Simulation—Implementing the Simulation Method

We can now implement the simulation pseudocode from Figure 7.4 on page 360. The implementation is shown as a method in Figure 7.8 on page 370. This method could be activated from a main program that interacts with a user. There are three parameters for the method: (1) an integer, `washTime`, which is the amount of time needed to wash one car; (2) a double number, `arrivalProb`, which is the probability that a customer arrives during any particular second; and (3) another integer, `totalTime`, which is the total number of seconds to be simulated. The method writes a copy of its parameters to the screen and then runs the simulation.

Most of the simulation work is carried out in the large for-loop, where the local variable `currentSecond` runs from 1 to `totalTime`. This loop parallels the large loop from the original pseudocode (Step 2 in Figure 7.4 on page 360).

After the loop finishes, the simulation method obtains two pieces of information from the averager and writes these items to `System.out`.

Self-Test Exercises for Section 7.2

7. How would you modify the palindromes program so that it indicates the first position in the input string that violates the palindrome property? For example, consider the input "Able were you ere you saw Elba." This looks like a palindrome until you see the first "e" in "were," so a suitable output would be:

```
That is not a palindrome.
Mismatch discovered at: Able we
```

8. How would you modify the palindromes program so that uppercase and lowercase versions of letters are considered different? In the modified program, the string "able was I ere I saw elba" would still be considered a palindrome, but the string "Able was I ere I saw Elba" would no longer be considered a palindrome since, among other things, the first and last letters, "A" and "a," are not the same under these changed rules.
9. Can a single program use both a stack and a queue?
10. Describe at least one assumption we made about the real-world car wash in order to make the simulation more manageable.
11. Use short sentences to describe the three main actions that occur during each second of simulated time in the car wash simulation.

12. When the car wash simulation finishes, there could still be some numbers in the queue. What do these numbers represent from the real world? (For a method of handling these leftover numbers, see Programming Project 8 on page 396.)
13. Our method in Figure 7.8 on page 370 uses an averager, and the method does not check that we don't give the averager more than `Integer.MAX_VALUE` numbers. Is this a potential problem for a long simulation?

FIGURE 7.5 Specification and Implementation for the `Averager` Class

Class Averager

❖ **public class Averager from the package edu.colorado.simulations**
An `Averager` computes an average of a sequence of numbers.

Specification

♦ **Constructor for the Averager**
`public Average( )`
Initialize an `Averager`.

Postcondition:
This `Averager` has been initialized and is ready to accept a sequence of numbers.

♦ **addNumber**
`public void addNumber(double value)`
Give another number to this `Averager`.

Parameters:
`value` – the next number to give to this `Averager`

Precondition:
`howManyNumbers( ) < Integer.MAX_VALUE`

Postcondition:
This `Averager` has accepted `value` as the next number in the sequence of numbers.

Throws: `IllegalStateException`
Indicates that `howManyNumbers( )` is `Integer.MAX_VALUE`.

♦ **average**
`public double average( )`
Provide an average of all numbers given to this `Averager`.

Returns:
the average of all the numbers that have been given to this `Averager`

Note:
If `howManyNumbers( )` is zero, then the answer is `Double.NaN` ("not a number") because the average of an empty set of numbers is not defined. The answer may also be `Double.POSITIVE_INFINITY` or `Double.NEGATIVE_INFINITY` if there has been an arithmetic overflow during the summing of all the numbers.

(continued)

(FIGURE 7.5 continued)

♦ **howManyNumbers**

`public int howManyNumbers( )`

Provide a count of how many numbers have been given to this `Averager`.

Returns:

the count of how many numbers have been given to this `Averager`

Implementation

```
// File: Averager.java from the package edu.colorado.simulations
// Complete documentation is available on page 365 or from the Averager link in
//   http://www.cs.colorado.edu/~main/docs/

package edu.colorado.simulations;

public class Averager
{
   private int count;  // How many numbers have been given to this averager
   private double sum; // Sum of all the numbers given to this averager

   public Averager( )
   {
      count = 0;
      sum = 0;
   }

   public void addNumber(double value)
   {
      if (count == Integer.MAX_VALUE)
         throw new IllegalStateException("Too many numbers.");
      count++;
      sum += value;
   }

   public double average( )
   {
      if (count == 0)
         return Double.NaN;
      else
         return sum/count;
   }

   public int howManyNumbers( )
   {
      return count;
   }
}
```


FIGURE 7.6 Specification and Implementation of the `BooleanSource` Class

Class BooleanSource

❖ **public class BooleanSource from the package edu.colorado.simulations**

A `BooleanSource` provides a random sequence of `boolean` values.

Specification

♦ **Constructor for the BooleanSource**

`public BooleanSource(double p)`

Initialize a `BooleanSource`.

Precondition: `0 <= p` and `p <= 1`.

Postcondition:

This `BooleanSource` has been initialized so that `p` is the approximate probability of returning `true` in any subsequent activation of the `query` method.

Throws: `IllegalArgumentException`

Indicates that `p` is outside of its legal range.

♦ **query (method of the BooleanSource)**

`public boolean query( )`

Get the next value from this `BooleanSource`.

Returns:

The return value is either `true` or `false`, with the probability of a `true` value being determined by the argument that was given to the constructor.

Implementation

```
// File: BooleanSource.java from the package edu.colorado.simulations
// Complete documentation is listed above, or available from the BooleanSource link at
//   http://www.cs.colorado.edu/~main/docs/

package edu.colorado.simulations;
public class BooleanSource
{
   private double probability; // The probability of query( ) returning true.

   public BooleanSource(double p)
   {
      if ((p < 0) || (1 < p))
         throw new IllegalArgumentException("Illegal p: " + p);
      probability = p;
   }

   public boolean query( )
   {
      return (Math.random( ) < probability);
   }
```

FIGURE 7.7 Specification and Implementation for the `Washer` Class

Class Washer

❖ **public class Washer from the package edu.colorado.simulations**
A `Washer` simulates a simple washing machine.

Specification

♦ **Constructor for the Washer**
`public Washer(int s)`
Initialize a `Washer`.
Parameter:
`s` - the number of seconds required for one wash cycle of this `Washer`
Postcondition:
This `Washer` has been initialized so that it takes `s` seconds to complete one wash cycle.

♦ **isBusy**
`public boolean isBusy( )`
Determine whether this `Washer` is currently busy.
Returns:
`true` if this `Washer` is busy (in a wash cycle); otherwise `false`

♦ **reduceRemainingTime**
`public void reduceRemainingTime( )`
Reduce the remaining time in the current wash cycle by one second.
Postcondition:
If a car is being washed, then the remaining time in the current wash cycle has been reduced by one second.

♦ **startWashing**
`public void startWashing( )`
Start a wash cycle for this `Washer`.
Precondition:
`isBusy( )` is `false`.
Postcondition:
This `Washer` has started simulating one wash cycle. Therefore, `isBusy( )` will return `true` until the required number of simulated seconds has passed.
Throws: `IllegalStateException`
Indicates that this `Washer` is busy.

(continued)

(FIGURE 7.7 continued)

Implementation

```
// File: Washer.java from the package edu.colorado.simulations
// Complete documentation is available on page 368 or from the Washer link in
//   http://www.cs.colorado.edu/~main/docs/

package edu.colorado.simulations;

public class Washer
{
   private int secondsForWash;  // Seconds for a single wash
   private int washTimeLeft;    // Seconds until this washer is no longer busy

   public Washer(int s)
   {
      secondsForWash = s;
      washTimeLeft = 0;
   }


   public boolean isBusy( )
   {
      return (washTimeLeft > 0);
   }


   public void reduceRemainingTime( )
   {
      if (washTimeLeft > 0)
         washTimeLeft--;
   }


   public void startWashing( )
   {
      if (washTimeLeft > 0)
         throw new IllegalStateException("Washer is already busy.");
      washTimeLeft = secondsForWash;
   }

}
```

FIGURE 7.8 Specification and Implementation of the Car Wash Method

Specification

♦ **carWashSimulate**

```
public static void carWashSimulate
(int washTime, double arrivalProb, int TotalTime)
```

Simulate the running of a car washer for a specified amount of time.

Parameters:

`washTime` - the number of seconds required to wash one car
`arrivalProb` - the probability of a customer arriving in any second, for example 0.1 is 10%
`totalTime` - the total number of seconds for the simulation

Precondition:

`washTime` and `totalTime` are positive; `arrivalProb` lies in the range 0 to 1.

Postcondition:

The method has simulated a car wash in which `washTime` is the number of seconds needed to wash one car, `arrivalProb` is the probability of a customer arriving in any second, and `totalTime` is the total number of seconds for the simulation. Before the simulation, the method has written its three parameters to `System.out`. After the simulation, the method has written two pieces of information to `System.out`: (1) the number of cars washed, and (2) the average waiting time for customers that had their cars washed. (Customers that are still in the queue are not included in this average.)

Throws: `IllegalArgumentException`

Indicates that one of the arguments violates the precondition.

Sample Output

The `carWashSimulate` method could be part of an interactive or noninteractive Java program. For example, a noninteractive program might activate

```
carWashSimulate(240, 0.0025, 6000);
```

which can produce this output:

```
Seconds to wash one car: 240
Probability of customer arrival during a second: 0.0025
Total simulation seconds: 6000
Customers served: 13
Average wait: 111.07692307682308 sec
```

The actual output may be different because of variations in the random number generator that is used in the `BooleanSource`.

(continued)

(FIGURE 7.8 continued)

Implementation

```
public static void carWashSimulate
(int washTime, double arrivalProb, int totalTime)
{
   Queue<Integer> arrivalTimes = new LinkedList<Integer>( );
   int next;
   BooleanSource arrival = new BooleanSource(arrivalProb);
   Washer machine = new Washer(washTime);
   Averager waitTimes = new Averager( );
   int currentSecond;

   // Write the parameters to System.out.
   System.out.println("Seconds to wash one car: " + washTime);
   System.out.print("Probability of customer arrival during a second: ");
   System.out.println(arrivalProb);
   System.out.println("Total simulation seconds: " + totalTime);

   // Check the precondition:
   if (washTime <= 0 || arrivalProb < 0 || arrivalProb > 1 || totalTime < 0)
      throw new IllegalArgumentException("Values out of range.");

   for (currentSecond = 0; currentSecond < totalTime; currentSecond++)
   {  // Simulate the passage of one second of time.

      // Check whether a new customer has arrived.
      if (arrival.query( ))
         arrivalTimes.add(currentSecond);

      // Check whether we can start washing another car.
      if ((!machine.isBusy( ))  &&  (!arrivalTimes.isEmpty( )))
      {
         next = arrivalTimes.remove( );
         waitTimes.addNumber(currentSecond - next);
         machine.startWashing( );
      }

      // Subtract one second from the remaining time in the current wash cycle.
      machine.reduceRemainingTime( );
   }

   // Write the summary information about the simulation.
   System.out.println("Customers served: " + waitTimes.howManyNumbers( ));
   if (waitTimes.howManyNumbers( ) > 0)
      System.out.println("Average wait: " + waitTimes.average( ) + " sec");
}
```

7.3 IMPLEMENTATIONS OF THE QUEUE CLASS

A queue seems conceptually simpler than a stack because we notice queues in our everyday lives. However, the implementation of a queue, while similar to that of a stack, is more complicated than the implementation of a stack. As was the case with the stack class, we will give two implementations of our queue class: an implementation that stores the items in an array and an implementation using a linked list. Each implementation will be a generic Queue of Java objects, but as we discussed earlier, we could easily build a queue of any of the Java primitive types to obtain other queues.

Array Implementation of a Queue

As we did with the stack class, we will implement the queue class as a partially filled array. With a queue, we add items at one end of the array and remove them from the other end. Hence, we will be accessing the used portion of the array at both ends, increasing the size of the used portion at one end and decreasing the size of the used portion at the other end. This differs from our use of a partially filled array for a stack in that the stack accessed just one end of the array.

keeping track of both ends of the partially filled array

Because we now need to keep track of both ends of the used portion of the array, we will have *two* variables to keep track of how much of the array is used: One variable, called `front`, indicates the first index currently in use, and one variable, called `rear`, indicates the last index currently in use. If `data` is the array name, then the queue items will be in the array components:

```
data[front], data[front + 1], ... data[rear].
```

To add an item, we increment `rear` by one and then store the new item in the component `data[rear]`, where `rear` is now one larger than it was before. To get the next item from the queue, we retrieve `data[front]` and then increment `front` by one so that `data[front]` is then the item that used to be second.

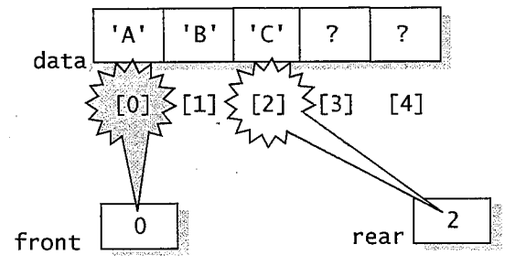
There is one problem with this plan. The variable `rear` is incremented but never decremented. Hence, it will quickly reach the end of the array. At that point, it seems as if adding more items requires the array to increase capacity. Yet there is likely to be room in the array. In a normal application, the variable `front` would also be incremented from time to time (when items are removed from the queue). This will free up the array locations with index values less than `front`. There are several ways to reuse these freed locations.

One straightforward approach for using the freed array locations is to maintain all the queue items so that `front` is always equal to 0 (the first index of the array). When `data[0]` is removed, we move all the items in the array down one location, so the value of `data[1]` is moved to `data[0]`, and then all other items are also moved down one. This approach will work, but it is inefficient. Every time we remove an item from the queue, we must move every item in the queue. Fortunately, there is a better approach.

circular array

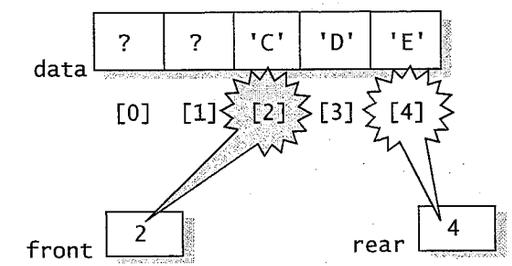
We do not need to move all the array elements. When the `rear` index reaches the end of the array, we can simply start using the available locations at the front of the array. One way to think of this arrangement is to think of the array as being bent into a circle so that the first component of the array is immediately after the last component of the array. In this way, the successor of the last array index is the first array index. In this circular arrangement, the free index positions are always "right after" `data[rear]`.

For example, suppose we have a queue of characters with a capacity of 5, and the queue currently contains three items `'A'`, `'B'`, and `'C'`. Perhaps these values are stored with `front` equal to 0 and `rear` equal to 2, as shown here:

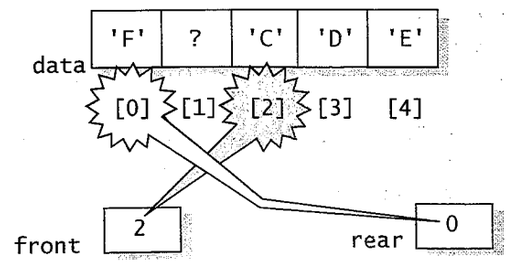


The question marks indicate unused spots in the array.

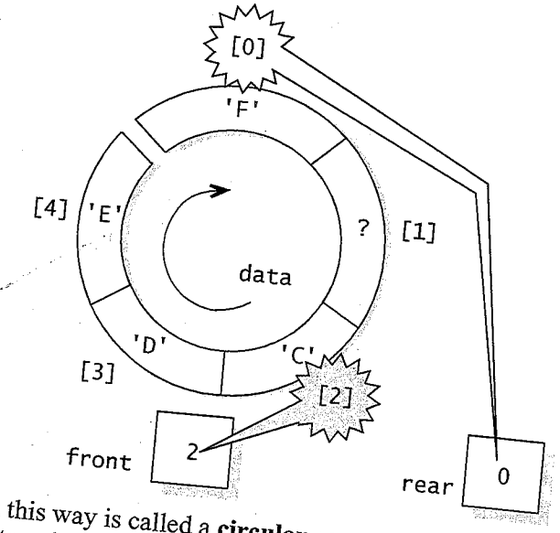
Let's remove two items (the `'A'` and `'B'`) and add two more items to the rear of this queue, perhaps the characters `'D'` and `'E'`. The result is shown here:



At this point, `front` is 2, `rear` is 4, and the queue items range from `data[2]` to `data[4]`. Suppose that now we add another character, `'F'`, to the queue. The new item cannot go after `rear` since we have hit the end of the array. Instead, we go to the front of the array, adding the new `'F'` at location `data[0]`, as shown here:



This may look peculiar with the rear index of 0 being *before* the front index of 2. But keep in mind the *circular* view of the array. With this view, the queue's items start at data[front] and continue forward. If you reach the end of the array, then come back to data[0] and keep going until you find the rear. It may help to actually view the array as bent into a circle, with the final array element attached back to the front, as shown here:



An array used in this way is called a **circular array**.

We now turn to a detailed presentation of our queue using the idea of a circular array. The specification and implementation for our queue are given in Figure 7.9 on page 376. The queue's items are held in an array, data, which is a private instance variable. The private instance variables front and rear hold the indexes for the front and the rear of the queue, as we have discussed. Whenever the queue is nonempty, the items begin at the location data[front], usually continuing forward (to higher indexes) in the array. If the items reach the end of the array, then they continue at the first location, data[0]. In any case, data[rear] is the last item in the queue. One other private instance variable, manyItems, records the number of items that are in the queue. We will use manyItems to check whether the queue is empty and also to produce the value returned by the size method.

The Queue class implementation also has a new feature: a *private* method called nextIndex. This is a method that we think will be useful for the implementation, but it is not part of the public specification. We don't want other programmers to use this method; it is just for our own use in implementing a specific kind of queue. A private method such as this is called a **helper method**.

the nextIndex helper method

The nextIndex helper method allows us to step easily through the array one index after another with wraparound at the end. The activation of nextIndex(i) usually returns i+1, with one exception. When i is equal to the last index of the array, nextIndex(i) returns zero (the first index of the array).

PROGRAMMING TIP

USE HELPER METHODS TO IMPROVE CLARITY

When a class requires some small computation that is likely to be used several times, consider implementing the computation with a helper method (that is, a private method). This will improve the clarity of your other code. Because a helper method is private, it does not need to be included in the documentation that's intended for other programmers, but you may write a precondition/postcondition contract for your own use.

The remaining implementation details are a straightforward implementation of a circular array, following this invariant:

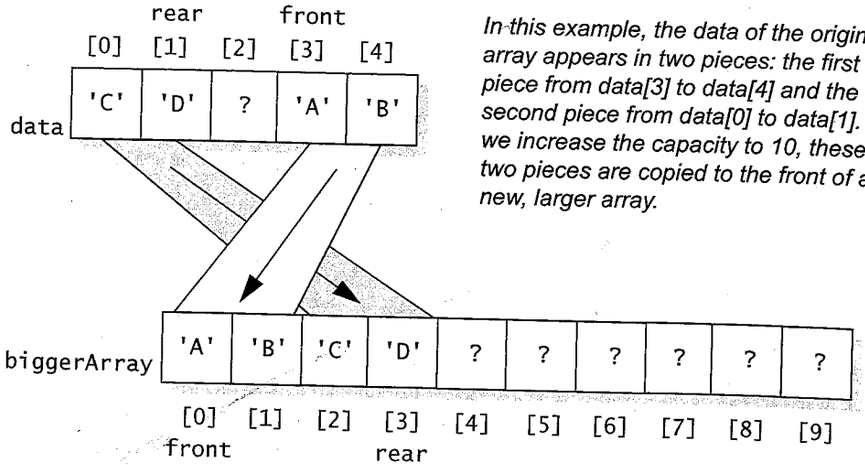
Invariant of the ArrayQueue Class

1. The number of items in the queue is stored in the instance variable manyItems.
2. For a nonempty queue, the items are stored in a circular array beginning at data[front] and continuing through data[rear].
3. For an empty queue, manyItems is zero and data is a reference to an array, but we are not using any of the array, and we don't care about front and rear.

The ensureCapacity method requires a bit of thought. The implementation starts by checking whether the current length of the data array is at least as big as the requested capacity. If so, the array is already big enough, and we return with no work. Otherwise, we must allocate a new larger array and copy the elements from the original array to the new array. The allocation and copying are handled with three cases:

ensureCapacity implementation

- If manyItems is zero, then we just allocate the new, larger array. There are no items to copy.
- If manyItems is nonzero and front ≤ rear, then we allocate the new array and copy data[front] ... data[rear] into the new array. This is not difficult because all the items are contiguous. We copy them into the new array with a single activation of System.arraycopy.
- If manyItems is nonzero and front > rear, then we allocate the new array, but it takes more care to copy the items into the new array. The items at the front of the queue are located at data[front] to the end of data. These items are followed by the items from data[0] to data[rear]. These two segments of items are copied into the new array by two separate activations of System.arraycopy, as shown at the top of the next page.



In this example, the data of the original array appears in two pieces: the first piece from data[3] to data[4] and the second piece from data[0] to data[1]. If we increase the capacity to 10, these two pieces are copied to the front of a new, larger array.

The `trimToSize` method has three similar cases that are similar to the `ensureCapacity` method.

All the methods and their specifications are given in Figure 7.9.

FIGURE 7.9 Specification and Implementation of the Array Version of the Generic Queue Class

Generic Class ArrayQueue

❖ **public class ArrayQueue<E> from the package edu.colorado.collections**

An `ArrayQueue` is a queue of references to `E` objects.

Limitations:

(1) The capacity of one of these queues can change after it's created, but the maximum capacity is limited by the amount of free memory on the machine. The constructors, `clone`, `ensureCapacity`, `add`, and `trimToSize` will result in an `OutOfMemoryError` when free memory is exhausted.

(2) A queue's capacity cannot exceed the largest integer 2,147,483,647 (`Integer.MAX_VALUE`). Any attempt to create a larger capacity results in failure due to an arithmetic overflow.

(continued)

(FIGURE 7.9 continued)

Specification

♦ **Constructor for the ArrayQueue<E>**

`public ArrayQueue( )`

Initialize an empty queue with an initial capacity of 10. Note that the add method works efficiently (without needing more memory) until this capacity is reached.

Postcondition:
This queue is empty and has an initial capacity of 10.

Throws: `OutOfMemoryError`
Indicates insufficient memory for: `new Object[10]`.

♦ **Second Constructor for the ArrayQueue<E>**

`public ArrayQueue(int initialCapacity)`

Initialize an empty queue with a specified initial capacity. Note that the add method works efficiently (without needing more memory) until this capacity is reached.

Parameters:
`initialCapacity` – the initial capacity of this queue

Precondition:
`initialCapacity` is non-negative.

Postcondition:
This queue is empty and has the given initial capacity.

Throws: `IllegalArgumentException`
Indicates that `initialCapacity` is negative.

Throws: `OutOfMemoryError`
Indicates insufficient memory for: `new Object[initialCapacity]`.

♦ **add—isEmpty—remove—size**

```
public void add(E)
public boolean isEmpty( )
public E remove( )
public int size( )
```

These are the standard queue specifications from Figure 7.2 on page 352.

♦ **clone**

`public ArrayQueue<E> clone( )`

Generate a copy of this queue.

Returns:
The return value is a copy of this queue. Subsequent changes to the copy will not affect the original, nor vice versa.

Throws: `OutOfMemoryError`
Indicates insufficient memory for creating the clone.

(continued)

(FIGURE 7.9 continued)

- **getCapacity**
  `public int getCapacity( )`
  Accessor method to determine the current capacity of this queue. The add method works efficiently (without needing more memory) until this capacity is reached.
  Returns:
  the current capacity of this queue

- **trimToSize**
  `public void trimToSize( )`
  Reduce the current capacity of this queue to its actual size (i.e., the number of items it contains).
  Postcondition:
  This queue's capacity has been changed to its current size.
  Throws: `OutOfMemoryError`
  Indicates insufficient memory for altering the capacity.

Implementation

```
// File: ArrayQueue.java from the package edu.colorado.collections
// Complete documentation is available on pages 376-378 or from the ArrayQueue link in
//   http://www.cs.colorado.edu/~main/docs/

package edu.colorado.collections;
import java.util.NoSuchElementException;

public class ArrayQueue<E> implements Cloneable
{
  // Invariant of the ArrayQueue class:
  //   1. The number of items in the queue is in the instance variable manyItems.
  //   2. For a nonempty queue, the items are stored in a circular array beginning at
  //      data[front] and continuing through data[rear].
  //   3. For an empty queue, manyItems is zero and data is a reference to an array, but
  //      we don't care about front and rear.
  private E[ ] data;
  private int manyItems;
  private int front;
  private int rear;
```

(continued)

(FIGURE 7.9 continued)

- **ensureCapacity**
  `public void ensureCapacity(int minimumCapacity)`
  Change the current capacity of this queue.
  Parameters:
  `minimumCapacity` – the new capacity for this queue
  Postcondition:
  This queue's capacity has been changed to at least `minimumCapacity`. If the capacity was already at or greater than `minimumCapacity`, then the capacity is left unchanged.
  Throws: `OutOfMemoryError`
  Indicates insufficient memory for: `new Object[minimumCapacity]`.

- **getCapacity**
  `public int getCapacity( )`
  Accessor method to determine the current capacity of this queue. The add method works efficiently (without needing more memory) until this capacity is reached.
  Returns:
  the current capacity of this queue

- **trimToSize**
  `public void trimToSize( )`
  Reduce the current capacity of this queue to its actual size (i.e., the number of items it contains).
  Postcondition:
  This queue's capacity has been changed to its current size.
  Throws: `OutOfMemoryError`
  Indicates insufficient memory for altering the capacity.

Implementation

```
// File: ArrayQueue.java from the package edu.colorado.collections
// Complete documentation is available on pages 376-378 or from the ArrayQueue link in
//   http://www.cs.colorado.edu/~main/docs/

package edu.colorado.collections;
import java.util.NoSuchElementException;

public class ArrayQueue<E> implements Cloneable
{
  // Invariant of the ArrayQueue class:
  //   1. The number of items in the queue is in the instance variable manyItems.
  //   2. For a nonempty queue, the items are stored in a circular array beginning at
  //      data[front] and continuing through data[rear].
  //   3. For an empty queue, manyItems is zero and data is a reference to an array, but
  //      we don't care about front and rear.
  private E[ ] data;
  private int manyItems;
  private int front;
  private int rear;
```

(continued)

(FIGURE 7.9 continued)

```
public ArrayQueue( )
{
   final int INITIAL_CAPACITY = 10;
   manyItems = 0;
   data = (E[]) new Object[INITIAL_CAPACITY];
   // We don't care about front and rear for an empty queue.
}

public ArrayQueue(int initialCapacity)
{
   if (initialCapacity < 0)
      throw new IllegalArgumentException
      ("initialCapacity is negative: " + initialCapacity);
   manyItems = 0;
   data = (E[]) new Object[initialCapacity];
   // We don't care about front and rear for an empty queue.
}

public void add(E item)
{
   if (manyItems == data.length)
   {
      // Double the capacity and add 1; this works even if manyItems is 0. However, in
      // case that manyItems*2 + 1 is beyond Integer.MAX_VALUE, there will be an
      // arithmetic overflow and the bag will fail.
      ensureCapacity(manyItems*2 + 1);
   }

   if (manyItems == 0)
   {
      front = 0;
      rear = 0;
   }
   else
      rear = nextIndex(rear);

   data[rear] = item;
   manyItems++;
}
```

(continued)

(FIGURE 7.9 continued)

```
public ArrayQueue<E> clone( )
{  // Clone an ArrayQueue<E>.
   ArrayQueue<E> answer;

   try
   {
      answer = (ArrayQueue<E>) super.clone( );
   }
   catch (CloneNotSupportedException e)
   {
      // This exception should not occur. But if it does, it would probably indicate a
      // programming error that made super.clone unavailable.
      // The most common error would be forgetting the "Implements Cloneable"
      // clause at the start of this class.
      throw new RuntimeException
      ("This class does not implement Cloneable.");
   }

   answer.data = data.clone( );

   return answer;
}

public int getCapacity( )
{
   return data.length;
}

public boolean isEmpty( )
{
   return (manyItems == 0);
}

private int nextIndex(int i)
{
   See the answer to Self-Test Exercise 17.
}
```

(continued)

(FIGURE 7.9 continued)

```
public void ensureCapacity(int minimumCapacity)
{
   E[ ]biggerArray;
   int n1, n2;

   if (data.length >= minimumCapacity)
      // No change needed.
      return;
   else if (manyItems == 0)
      // Just increase the size of the array because the queue is empty.
      data = (E[]) new Object[minimumCapacity];
   else if (front <= rear)
   {  // Create larger array and copy data[front]...data[rear] into it.
      biggerArray = (E[]) new Object[minimumCapacity];
      System.arraycopy(data, front, biggerArray, front, manyItems);
      data = biggerArray;
   }
   else
   {  // Create a bigger array, but be careful about copying items into it. The queue items
      // occur in two segments. The first segment goes from data[front] to the end of the
      // array, and the second segment goes from data[0] to data[rear]. The variables n1
      // and n2 will be set to the number of items in these two segments. We will copy
      // these segments to biggerArray[0...manyItems-1].
      biggerArray = (E[]) new Object[minimumCapacity];
      n1 = data.length - front;
      n2 = rear + 1;
      System.arraycopy(data, front, biggerArray, 0, n1);
      System.arraycopy(data, 0, biggerArray, n1, n2);
      front = 0;
      rear = manyItems-1;
      data = biggerArray;
   }
}

public E remove( )
{
   E answer;

   if (manyItems == 0)
      throw new NoSuchElementException("Queue underflow.");
   answer = data[front];
   front = nextIndex(front);
   manyItems--;
   return answer;
}
```

(continued)

(FIGURE 7.9 continued)

```
public int size( )
{
   return manyItems;
}

public void trimToSize( )
{
   || See the answer to Self-Test Exercise 18.
}
}
```

Linked List Implementation of a Queue

A queue can also be implemented as a linked list. One end of the linked list is the front, and the other end is the rear of the queue. The approach uses two references to nodes: One refers to the first node (`front`), and the other refers to the last node (`rear`), as diagrammed here for a queue with three items:

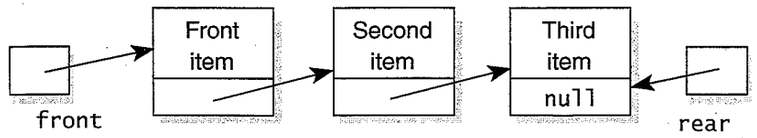


Figure 7.10 on page 385 shows the specification and implementation for a queue class that is implemented in this way. The class, called `LinkedQueue`, has no capacity problems, so it is considerably simpler than the array version. Here is the invariant we use in the linked list implementation:

Invariant of the LinkedQueue Class

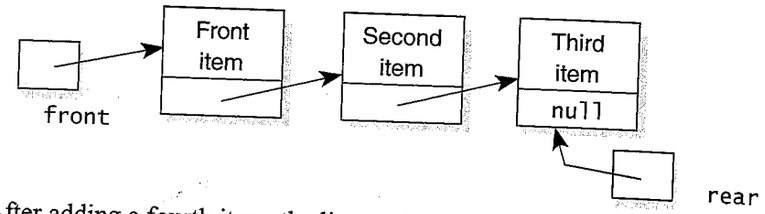
1. The number of items in the queue is stored in the instance variable `manyNodes`.
2. The items in the queue are stored in a linked list, with the front of the queue stored at the head node and the rear of the queue stored at the final node.
3. For a nonempty queue, the instance variable `front` is the head reference of the linked list of items, and the instance variable `rear` is the tail reference of the linked list. For an empty queue, both `front` and `rear` are the null reference.

Each of the queue methods (except the constructors) can assume that the invariant is valid when the method is activated, and each method must ensure that the invariant is valid when the method finishes its work. Notice that the invariant includes an instance variable called `manyNodes` to keep track of the total number

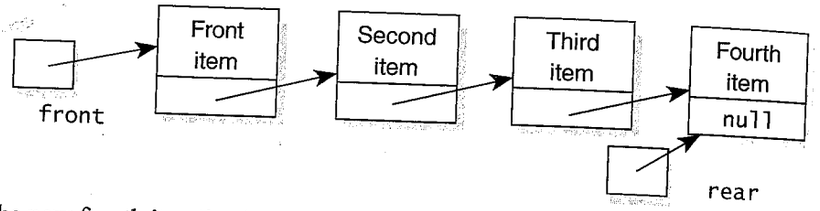
of items in the queue. We could get by without `manyNodes`, but its presence makes the `size` method quicker.

Most of the methods' work will be accomplished by the methods of our generic `Node` class that we developed in Chapters 4 and 5 (with a complete implementation in Appendix E). We'll look at the implementations of two queue methods—`add` and `remove`—in some detail.

The add method. The add operation adds a node at the rear of the queue. For example, suppose we start with three items shown here:



After adding a fourth item, the list would look like this:



The new fourth item is placed in a newly created node at the end of the list. Normally, this is accomplished by two statements:

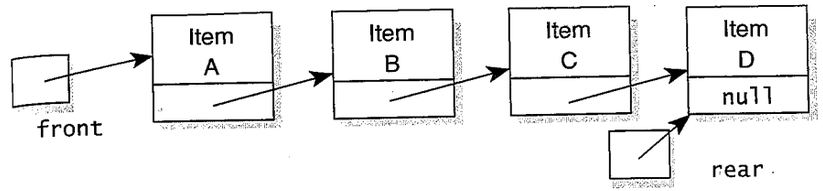
```
rear.addNodeAfter(item);
rear = rear.getLink( );
```

To add the first item, we need a slightly different approach because the empty list has only null references for the front and rear. In this case, we should add the new item at the front of the list and then assign `rear` to also refer to the new node, as shown in this code:

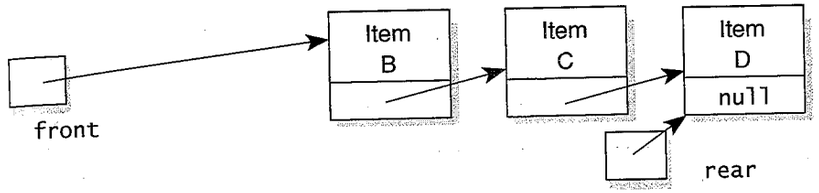
```
if (isEmpty( ))
{  // Insert first item.
   front = new Node<E>(item, null);
   rear = front;
}
else
{  // Insert an item that is not the first.
   rear.addNodeAfter(item);
   rear = rear.getLink( );
}
```

In fact, this code is most of the add method. The only other work is to add one to the `manyNodes` instance variable.

The remove method. The `remove` method removes a node from the front of the queue. For example, suppose we start with this queue:



In this example, the `remove` method will return the item that is labeled "Item A." This item will also be removed from the queue so that, when `remove` returns, the list will have only three items shown here:



The implementation of `remove` uses the technique from page 176 to remove a node from the head. Most of the work is carried out by these statements:

```
E answer;
...
answer = front.getData( );
front = front.getLink( );
...
return answer;
```

The only other work in the complete implementation is to check the precondition (that the queue has some items), and if we remove the final item of the queue, then we must also set `rear` to null.

FIGURE 7.10 Specification and Implementation for the Linked List Version of the Queue Class

Generic Class LinkedQueue

❖ **public class LinkedQueue<E> from the package edu.colorado.collections**

A `LinkedQueue<E>` is a queue of references to `E` objects.

Limitations:

Beyond `Int.MAX_VALUE` items, `size` is wrong.

(continued)

(FIGURE 7.10 continued)

Specification

- **Constructor for the LinkedQueue<E>**

  `public LinkedQueue( )`

  Initialize an empty queue.

  Postcondition:

  This queue is empty.

- **add—isEmpty—remove—size**

  `public void add( )`
  `public boolean isEmpty( )`
  `public E remove( )`
  `public long size( )`

  These are the standard queue specifications from Figure 7.2 on page 352.

- **clone**

  `public E clone( )`

  Generate a copy of this queue.

  Returns:

  The return value is a copy of this queue. Subsequent changes to the copy will not affect the original, nor vice versa.

  Throws: `OutOfMemoryError`

  Indicates insufficient memory for creating the clone.

Implementation

```
// File: LinkedQueue.java from the package edu.colorado.collections
// Complete documentation is written above or is available from the LinkedQueue link in
//   http://www.cs.colorado.edu/~main/docs/
package edu.colorado.collections;
import java.util.NoSuchElementException;
import edu.colorado.nodes.Node;

public class LinkedQueue<E> implements Cloneable
{
  // Invariant of the LinkedQueue<E> class:
  //   1. The number of items in the queue is stored in the instance variable manyNodes.
  //   2. The items in the queue are stored in a linked list, with the front of the queue
  //      stored at the head node and the rear of the queue at the final node.
  //   3. For a nonempty queue, the instance variable front is the head reference of the
  //      linked list of items, and the instance variable rear is the tail reference of the
  //      linked list. For an empty queue, both front and rear are the null reference.
  private int manyNodes;
  private Node<E> front;
  private Node<E> rear;

  public LinkedQueue( )
  {
    front = null;
    rear = null;
  }
```

(continued)

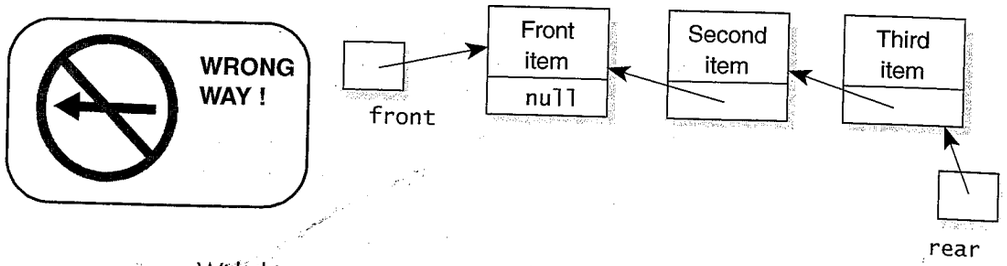
(FIGURE 7.10 continued)

```
  public void add(E item)
  {
    if (isEmpty( ))
    {  // Insert first item.
      front = new Node<E>(item, null);
      rear = front;
    }
    else
    {  // Insert an item that is not the first.
      rear.addNodeAfter(item);
      rear = rear.getLink( );
    }
    manyNodes++;
  }

  public LinkedQueue<E> clone( )
  { See the answer to Self-Test Exercise 19. }

  public boolean isEmpty( )
  {
    return (manyNodes == 0);
  }

  public E remove( )
  {
    E answer;

    if (manyNodes == 0)
      // NoSuchElementException is from java.util and its constructor has no argument.
      throw new NoSuchElementException("Queue underflow.");
    answer = front.getData( );
    front = front.getLink( );
    manyNodes--;
    if (manyNodes == 0)
      rear = null;
    return answer;
  }

  public int size( )
  {
    return manyNodes;
  }

}
```

PITFALL

FORGETTING WHICH END IS WHICH

We implemented our queue with the front of the queue at the head of the list and the rear of the queue at the tail of the list. As it turns out, this was not an arbitrary choice. Can you see why? What would have happened if we had tried to do things the other way around, as shown in this wrong diagram?



With this wrong arrangement of the queue's front and rear, we can still implement the add operation, adding a new node at the rear of the queue. We also have easy access to the front item. But it will be difficult to actually remove the front item. After the removal, the `front` must be positioned so that it refers to the next node in the queue, and there is no constant time algorithm to accomplish that (though a linear time algorithm could start at the rear and move through the whole list).

So, keep in mind that it's easy to insert and remove at the head of a linked list, but insertion is the only easy operation at the tail.

Self-Test Exercises for Section 7.3

14. Under what circumstances is a helper method useful?
15. Write a new accessor method that returns a copy of the item at the rear of the queue. Use the array version of the queue.
16. A programmer who sees our array implementation of a queue shown in Figure 7.9 on page 376 gives us the following suggestion: "Why not eliminate the `manyItems` instance variable since the number of items in the queue can be determined from the values of `front` and `rear`?" Is there a problem with the suggestion?
17. Implement the `nextIndex` method for the array version of the queue.
18. Implement the `trimToSize` method for the array version of the queue.
19. Implement the `clone` method for the linked list version of the queue. You should use `Node.listCopyWithTail`.
20. Write a new accessor method that returns a copy of the item at the rear of the queue. Use the linked list version of the queue.
21. Write another accessor method that returns a copy of the item at the *rear* of the queue. Use the linked list version of the queue.

22. What goes wrong if we try to put the front of the queue at the tail of the linked list?

7.4 PRIORITY QUEUES

Using a queue ensures that customers are served in the exact order in which they arrive. However, we often want to assign priorities to customers and serve the higher priority customers before those of lower priority. For example, a hospital emergency room will handle the most severely injured patients first, even if they are not "first in line." A computer operating system that keeps a queue of programs waiting to use some resource, such as a printer, may give interactive programs a higher priority than batch-processing programs that will not be picked up by the user until the next day, or it might give higher priority to short programs and lower priority to longer programs. A **priority queue** is a data structure that stores items along with a priority for each item. Items are removed in order of priorities. The highest priority item is removed first. If several items have equally high priorities, then the one that was placed in the priority queue first is the one removed first.

higher numbers indicate a higher priority

For example, suppose the following customer names and priorities are entered into a priority queue in the order given:

Ginger Snap, priority 0
Natalie Attired, priority 3
Emanual Transmission, priority 2
Gene Pool, priority 3
Kay Sera, priority 2

The higher numbers indicate a higher priority. The names would be removed in the following order: first Natalie Attired (with the highest priority), then Gene Pool, then Emanual Transmission, then Kay Sera, and finally Ginger Snap (with the lowest priority). Note that both Natalie Attired and Gene Pool have priority 3, which is the highest priority, but Natalie Attired entered the priority queue before Gene Pool, so Natalie Attired is removed before Gene Pool.

Priority Queue ADT—Specification

Figure 7.11 gives the specification for some methods of a priority queue class. When an item is added to a priority queue, it is entered with a specified priority, which is an integer in the range 0...`highest`, where `highest` is a parameter to the priority queue constructor. Items are removed from the queue in order of priority. For example, suppose `highest` is 2. In that case, items of priority 2 are removed first. When there are no items of priority 2 left, then items of priority 1 are removed. When there are no items of priority 1 left, then items of priority 0 are removed. Items with the same priority are removed in the order in which they were entered into the priority queue.

FIGURE 7.11 Specification of the Key Methods of a Generic `PriorityQueue` Class

Specification

These are the constructor, `remove`, and `add` methods of a generic priority queue class.

- **Constructor for the PriorityQueue<E>**

  `public PriorityQueue(int highest)`

  Initialize an empty priority queue.

  Parameters:

  `highest` – the highest priority allowed in this priority queue

  Precondition:

  `highest >= 0.`

  Postcondition:

  This priority queue is empty.

  Throws: `IllegalArgumentException`

  Indicates that `highest` is negative.

- **add**

  `public void add(E item, int priority)`

  Add a new item to this priority queue.

  Parameters:

  `item` – the item to be added to this queue

  `priority` – the priority of the new item

  Precondition:

  `0 <= priority` and `priority` is no more than the highest priority.

  Postcondition:

  The item has been added to this priority queue.

  Throws: `IllegalArgumentException`

  Indicates an illegal priority.

  Throws: `IllegalArgumentException`

  Indicates insufficient memory for adding a new item to this priority queue.

- **remove**

  `public E remove( )`

  Get the highest priority item, removing it from this priority queue.

  Precondition:

  This queue is not empty.

  Postcondition:

  The return value is the highest priority item of this queue, and the item has been removed. If several items have equal priority, then the one that entered first will come out first.

  Throws: `NoSuchElementException`

  Indicates that this queue is empty..

Priority Queue Class—An Implementation That Uses an Ordinary Queue

You may have noticed that the priority queue has a lot in common with the ordinary queue. We used the same names for the methods, and only add has an extra parameter for the priority of the new item. This suggests that much of the priority queue work has already been accomplished in the queue implementation—but what is the best way to take advantage of this?

One possibility is to provide the priority queue with an array of ordinary queues. For example, suppose `highest` is 2. In this case, a priority queue can be implemented using three ordinary queues: one to hold the items with priority 0, another queue for the items of priority 1, and a third queue for the lofty items with priority 2. Using this idea, we could implement a priority queue by defining a private instance variable that is an array of ordinary queues, as shown in the highlighted line here:

```
// Import a class for an ordinary Queue:
import edu.colorado.collections.ArrayQueue;

public class PriorityQueue<E>
{
    private ArrayQueue<E>[ ] queues
    ...
```

an array of ordinary queues

In this implementation, the constructor allocates the memory for the array of queues with the statement:

```
queues = (ArrayQueue<E>[]) new Object[highest+1];
```

The array has `highest+1` elements (from `queues[0]` through `queues[highest]`).

The items with priority 0 are stored in the ordinary queue called `queues[0]`. Items with priority 1 are stored in `queues[1]`. And so on, up to the ordinary queue called `queues[highest]`. To enter an item into the priority queue with a given priority, all we need to do is enter the item into the correct ordinary queue. For example, suppose `highest` is 3 and items are put into the priority queue in the following order:

Ginger Snap, priority 0
Natalie Attired, priority 3
Emanual Transmission, priority 2
Gene Pool, priority 3
Kay Sera, priority 2

each item of the priority queue is placed in one of the ordinary queues

After these items, the array of ordinary queues looks like this:

queues[3]: Natalie Attired (at the front), followed by Gene Pool
queues[2]: Emanual Transmission (at the front), followed by Kay Sera
queues[1]: empty
queues[0]: Ginger Snap

When an item needs to be removed, we move down through the ordinary queues, starting with the highest priority, until we find a nonempty queue. We then remove the front item from this nonempty queue. This process could be made more efficient if we also keep a member variable to indicate which is the highest numbered ordinary queue that is non-empty.

With this approach, we can implement a priority queue as an array of ordinary queues and do not need to duplicate a lot of the previous work. In fact, it is possible to implement the priority queue with no instance variables other than the `queues` array. However, some efficiency is gained by adding two other instance variables.

The first is a number, `totalSize`, which keeps track of the total number of items in all of the queues. With this instance variable, a `size` method for the priority queue can operate in constant time.

The second extra instance variable is an integer, which keeps track of the highest priority that is currently in the priority queue. This makes remove more efficient since, without the extra instance variable, we must always search from `highest` downward until we find a nonempty queue. With the extra private instance variable, `remove` can go straight to the correct queue and remove an item. (Of course, this extra instance variable also must be maintained correctly so that its value is always the priority of the highest item.) We leave the remainder of the details of writing this array-based implementation for Programming Project 4 on page 396.

Priority Queue ADT—A Direct Implementation

If the number of possible priorities is large, then an array of queues might be impractical. In this case, you might think of several alternatives. One possibility is to implement the priority queue as an ordinary linked list in which the data in each node contains two things: the item from the queue and the priority of that item. This implementation works regardless of how large the priority range is. We will leave the details of the implementation as another exercise (Programming Project 5 on page 396).

Java's Priority Queue

Java has a different form of a priority queue in `java.util.PriorityQueue`. This generic class, `PriorityQueue<E>`, has a generic type parameter for the elements in the queue, and these elements are usually comparable to each other (rather than having a separate priority value for each insertion into the queue).

Self-Test Exercises for Section 7.4

23. Implement the add method of the priority queue. Use the implementation that has an array called `queues` of ordinary queues.
24. Suppose you know that the number of priorities that will be used in a priority queue application is small, but you do not know what the actual priorities will be. For example, suppose you know that the priorities can be any integer values in the range 0 to 1,000,000, but you also know that there will be at most 25 different numbers used. How might you implement the priority queue as an array of ordinary queues in such a way that the array size is much smaller than 1,000,000?

CHAPTER SUMMARY

- A queue is a first-in/first-out data structure.
- A queue can be used to buffer data that is being sent from a fast computer component to a slower component. Queues also have many other applications: in simulation programs, operating systems, and elsewhere.
- A queue can be implemented as a partially filled circular array.
- A queue can be implemented as a linked list.
- When implementing a stack, you need keep track of only one end of the list of items, but when implementing a queue, you need to keep track of both ends of the list.
- A priority queue can be implemented as an array of ordinary queues or in various ways as a linked list.

Solutions to Self-Test Exercises

1. LIFO (Last-In/First-Out) and FIFO (First-In/First-Out) refer to the order in which entries must be removed.
2. For adding: insert or enqueue ("enter queue"); for removing: getFront and dequeue ("delete from queue")
3. The `isEmpty` function tests for an empty queue.
4. The operations are add `'L'`, add `'I'`, add `'N'`, add `'E'`, followed by four `remove` operations that return `'L'`, then `'I'`, then `'N'`, then `'E'`.
5. Reading input from a keyboard and sending output to a printer.
6. Read the characters two at a time. Each pair of characters has the first character placed in queue number 1 and the second character placed in queue number 2. After all the reading is done, print all characters from queue number 1 on one line, and print all characters from queue number 2 on a second line.