Chapter 4: Stacks and Queues

Overview

In this chapter we'll examine three data storage structures: the stack, the queue, and the priority queue. We'll begin by discussing how these structures differ from arrays; then we'll examine each one in turn. In the last section, we'll look at an operation in which the stack plays a significant role: parsing arithmetic expressions.

A Different Kind of Structure

There are significant differences between the data structures and algorithms we've seen in previous chapters and those we'll look at now. We'll discuss three of these differences before we examine the new structures in detail.

Programmer's Tools

The array—the data storage structure we've been examining thus far—as well as many other structures we'll encounter later in this book (linked lists, trees, and so on), are appropriate for the kind of data you might find in a database application. They're typically used for personnel records, inventories, financial data, and so on; data that corresponds to real-world objects or activities. These structures facilitate access to data: they make it easy to insert, delete, and search for particular items.

The structures and algorithms we'll examine in this chapter, on the other hand, are more often used as programmer's tools. They're primarily conceptual aids rather than full-fledged data storage devices. Their lifetime is typically shorter than that of the database-type structures. They are created and used to carry out a particular task during the operation of a program; when the task is completed, they're discarded.

Restricted Access

In an array, any item can be accessed, either immediately—if its index number is known—or by searching through a sequence of cells until it's found. In the data structures in this chapter, however, access is restricted: only one item can be read or removed at a given time.

The interface of these structures is designed to enforce this restricted access. Access to other items is (in theory) not allowed.

More Abstract

Stacks, queues, and priority queues are more abstract entities than arrays and many other data storage structures. They're defined primarily by their interface: the permissible operations that can be carried out on them. The underlying mechanism used to implement them is typically not visible to their user.

For example, the underlying mechanism for a stack can be an array, as shown in this chapter, or it can be a linked list. The underlying mechanism for a priority queue can be an array or a special kind of tree called a *heap*. We'll return to the topic of one data structure being implemented by another when we discuss Abstract Data Types (ADTs) in Chapter 5, "Linked Lists."

Stacks

A stack allows access to only one data item: the last item inserted. If you remove this item, then you can access the next-to-last item inserted, and so on. This is a useful capability in many programming situations. In this section, we'll see how a stack can be used to check whether parentheses, braces, and brackets are balanced in a computer program source file. At the end of this chapter, we'll see a stack playing a vital role in parsing (analyzing) arithmetic expressions such as 3*(4+5).

A stack is also a handy aid for algorithms applied to certain complex data structures. In Chapter 8, "Binary Trees," we'll see it used to help traverse the nodes of a tree. In Chapter 13, "Graphs," we'll apply it to searching the vertices of a graph (a technique that can be used to find your way out of a maze).

Most microprocessors use a stack-based architecture. When a method is called, its return address and arguments are pushed onto a stack, and when it returns they're popped off. The stack operations are built into the microprocessor.

Some older pocket calculators used a stack-based architecture. Instead of entering arithmetic expressions using parentheses, you pushed intermediate results onto a stack. We'll learn more about this approach when we discuss parsing arithmetic expressions in the last section in this chapter.

The Postal Analogy

To understand the idea of a stack, consider an analogy provided by the U. S. Postal Service. Many people, when they get their mail, toss it onto a stack on the hall table or into an "in" basket at work. Then, when they have a spare moment, they process the accumulated mail from the top down. First they open the letter on the top of the stack and take appropriate action—paying the bill, throwing it away, or whatever. When the first letter has been disposed of, they examine the next letter down, which is now the top of the stack, and deal with that. Eventually they work their way down to the letter on the bottom of the stack (which is now the top). Figure 4.1 shows a stack of mail.

This "do the top one first" approach works all right as long as you can easily process all the mail in a reasonable time. If you can't, there's the danger that letters on the bottom of the stack won't be examined for months, and the bills they contain will become overdue.

Of course, many people don't rigorously follow this top-to-bottom approach. They may, for example, take the mail off the bottom of the stack, so as to process the oldest letter first. Or they might shuffle through the mail before they begin processing it and put higher-priority letters on top. In these cases, their mail system is no longer a stack in the computer-science sense of the word. If they take letters off the bottom, it's a queue; and if they prioritize it, it's a priority queue. We'll look at these possibilities later.

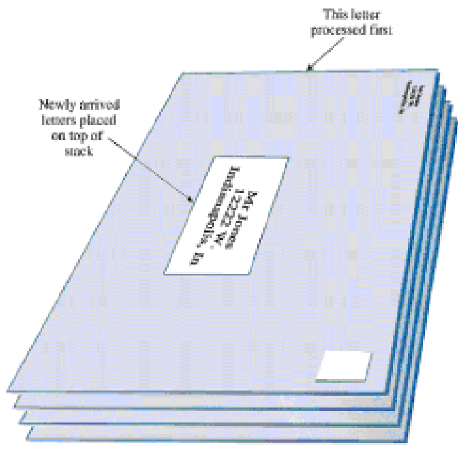

This letter processed first
Newly arrived letters placed on top of stack
Mr Jones
12222 W.
Indianapolis, In

Figure 4.1: A stack of letters

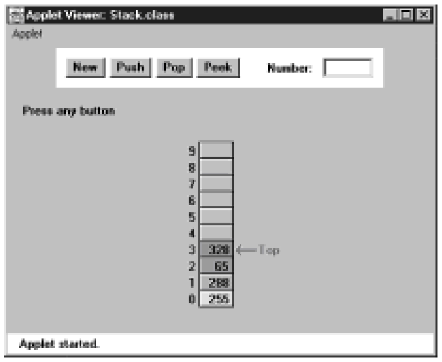
Another stack analogy is the tasks you perform during a typical workday. You're busy on a long-term project (A), but you're interrupted by a coworker asking you for temporary help with another project (B). While you're working on B, someone in accounting stops by for a meeting about travel expenses (C), and during this meeting you get an emergency call from someone in sales and spend a few minutes troubleshooting a bulky product (D). When you're done with call D, you resume meeting C; when you're done with C, you resume project B, and when you're done with B you can (finally!) get back to project A. Lower priority projects are "stacked up" waiting for you to return to them.

Placing a data item on the top of the stack is called *pushing* it. Removing it from the top of the stack is called *popping* it. These are the primary stack operations. A stack is said to be a Last-In-First-Out (LIFO) storage mechanism, because the last item inserted is the first one to be removed.

The Stack Workshop Applet

Let's use the Stack Workshop applet to get an idea how stacks work. When you start up the applet, you'll see four buttons: New, Push, Pop, and Peek, as shown in Figure 4.2.

The Stack Workshop applet is based on an array, so you'll see an array of data items. Although it's based on an array, a stack restricts access, so you can't access it as you would an array. In fact, the concept of a stack and the underlying data structure used to implement it are quite separate. As we noted earlier, stacks can also be implemented by other kinds of storage structures, such as linked lists.



Press any button

9	
8	
7	
6	
5	
4	
3	328
2	65
1	288
0	255

Applet started.

Figure 4.2: The Stack Workshop applet

New

The stack in the Workshop applet starts off with four data items already inserted. If you want to start with an empty stack, the New button creates a new stack with no items. The next three buttons carry out the significant stack operations.

Push

To insert a data item on the stack, use the button labeled Push. After the first press of this button, you'll be prompted to enter the key value of the item to be pushed. After typing it into the text field, a few more presses will insert the item on the top of the stack.

A red arrow always points to the top of the stack; that is, the last item inserted. Notice how, during the insertion process, one step (button press) increments (moves up) the Top arrow, and the next step actually inserts the data item into the cell. If you reversed the order, you'd overwrite the existing item at Top. When writing the code to implement a stack, it's important to keep in mind the order in which these two steps are executed.

If the stack is full and you try to push another item, you'll get the `Can't insert: stack is full` message. (Theoretically, an ADT stack doesn't become full, but the array implementing it does.)

Pop

To remove a data item from the top of the stack, use the Pop button. The value popped appears in the Number text field; this corresponds to a `pop()` routine returning a value.

Again, notice the two steps involved: first the item is removed from the cell pointed to by Top; then Top is decremented to point to the highest occupied cell. This is the reverse of the sequence used in the push operation.

The pop operation shows an item actually being removed from the array, and the cell color becoming gray to show the item has been removed. This is a bit misleading, in that deleted items actually remain in the array until written over by new data. However, they cannot be accessed once the Top marker drops below their position, so conceptually they are gone, as the applet shows.

When you've popped the last item off the stack, the Top arrow points to –1, below the lowest cell. This indicates that the stack is empty. If the stack is empty and you try to pop an item, you'll get the `Can't pop: stack is empty` message.

Peek

Push and pop are the two primary stack operations. However, it's sometimes useful to be able to read the value from the top of the stack without removing it. The peek operation does this. By pushing the Peek button a few times, you'll see the value of the item at Top copied to the Number text field, but the item is not removed from the stack, which remains unchanged.

Notice that you can only peek at the top item. By design, all the other items are invisible to the stack user.

Stack Size

Stacks are typically small, temporary data structures, which is why we've shown a stack of only 10 cells. Of course, stacks in real programs may need a bit more room than this, but it's surprising how small a stack needs to be. A very long arithmetic expression, for example, can be parsed with a stack of only a dozen or so cells.

Java Code for a Stack

Let's examine a program, `Stack.java`, that implements a stack using a class called `StackX`. Listing 4.1 contains this class and a short `main()` routine to exercise it.

Listing 4.1 The Stack.java Program

```
// Stack.java
// demonstrates stacks
// to run this program: C>java StackApp
import java.io.*;                 // for I/O
////////////////////////////////////////////////////////////////
class StackX
   {
   private int maxSize;        // size of stack array
   private double[] stackArray;
   private int top;            // top of stack

//--------------------------------------------------------------
-
   public StackX(int s)         // constructor
      {
      maxSize = s;             // set array size
      stackArray = new double[maxSize];  // create array
      top = -1;                // no items yet
      }

//--------------------------------------------------------------
-
   public void push(double j)  // put item on top of stack
      {
      stackArray[++top] = j;   // increment top, insert item
      }

//--------------------------------------------------------------
-
   public double pop()         // take item from top of stack
      {
      return stackArray[top--]; // access item, decrement top
      }

//--------------------------------------------------------------
-
   public double peek()        // peek at top of stack
      {
      return stackArray[top];
      }

//--------------------------------------------------------------
-
   public boolean isEmpty()    // true if stack is empty
      {
```

```
      return (top == -1);
      }

//--------------------------------------------------------------
-
   public boolean isFull()     // true if stack is full
      {
      return (top == maxSize-1);
      }

//--------------------------------------------------------------
-
   }  // end class StackX

////////////////////////////////////////////////////////////////

class StackApp
   {
   public static void main(String[] args)
      {
      StackX theStack = new StackX(10);  // make new stack
      theStack.push(20);               // push items onto stack
      theStack.push(40);
      theStack.push(60);
      theStack.push(80);

      while( !theStack.isEmpty() )     // until it's empty,
         {                             // delete item from
stack
         double value = theStack.pop();
         System.out.print(value);      // display it
         System.out.print(" ");
         }  // end while
      System.out.println("");
      }  // end main()

   }  // end class StackApp
```

The `main()` method in the `StackApp` class creates a stack that can hold 10 items, pushes 4 items onto the stack, and then displays all the items by popping them off the stack until it's empty. Here's the output:

```
80 60 40 20
```

Notice how the order of the data is reversed. Because the last item pushed is the first one popped; the 80 appears first in the output.

This version of the `StackX` class holds data elements of type `double`. As noted in the last chapter, you can change this to any other type, including object types.

`StackX` Class Methods

The constructor creates a new stack of a size specified in its argument. The fields of the stack comprise a variable to hold its maximum size (the size of the array), the array itself, and a variable `top`, which stores the index of the item on the top of the stack. (Note that we need to specify a stack size only because the stack is implemented using an array. If it had been implemented using a linked list, for example, the size specification would be unnecessary.)

The `push()` method increments `top` so it points to the space just above the previous `top`, and stores a data item there. Notice that `top` is incremented before the item is inserted.

The `pop()` method returns the value at `top` and then decrements `top`. This effectively removes the item from the stack; it's inaccessible, although the value remains in the array (until another item is pushed into the cell).

The `peek()` method simply returns the value at `top`, without changing the stack.

The `isEmpty()` and `isFull()` methods return true if the stack is empty or full, respectively. The `top` variable is at –1 if the stack is empty and `maxSize-1` if the stack is full.

Figure 4.3 shows how the stack class methods work.

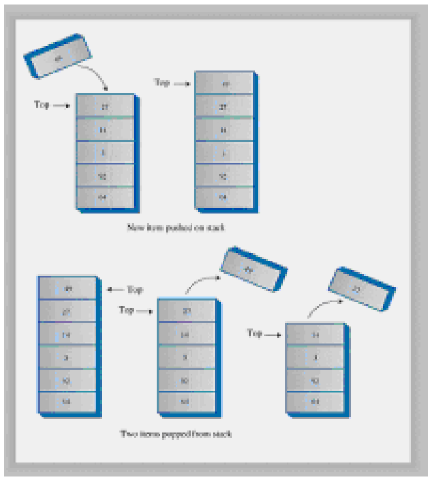


Figure 4.3: Operation of the `StackX` class methods

Error Handling

There are different philosophies about how to handle stack errors. What happens if you try to push an item onto a stack that's already full, or pop an item from a stack that's empty?

We've left the responsibility for handling such errors up to the class user. The user should always check to be sure the stack is not full before inserting an item:

```
if( !theStack.isFull() )
   insert(item);
else
   System.out.print("Can't insert, stack is full");
```

In the interest of simplicity, we've left this code out of the `main()` routine (and anyway, in this simple program, we know the stack isn't full because it has just been initialized). We do include the check for an empty stack when `main()` calls `pop()`.

Many stack classes check for these errors internally, in the `push()` and `pop()` methods. This is the preferred approach. In Java, a good solution for a stack class that discovers such errors is to throw an exception, which can then be caught and processed by the class user.

Stack Example 1: Reversing a Word

For our first example of using a stack, we'll examine a very simple task: reversing a word. When you run the program, it asks you to type in a word. When you press Enter, it displays the word with the letters in reverse order.

A stack is used to reverse the letters. First the characters are extracted one by one from the input string and pushed onto the stack. Then they're popped off the stack and displayed. Because of its last-in-first-out characteristic, the stack reverses the order of the characters. Listing 4.2 shows the code for the `reverse.java` program.

Listing 4.2 The reverse.java Program

```
// reverse.java
// stack used to reverse a string
// to run this program: C>java ReverseApp
import java.io.*;                  // for I/O
////////////////////////////////////////////////////////////////
class StackX
   {
   private int maxSize;
   private char[] stackArray;
   private int top;

//--------------------------------------------------------------
-
   public StackX(int max)     // constructor
      {
      maxSize = max;
      stackArray = new char[maxSize];
      top = -1;
      }

//--------------------------------------------------------------
-
   public void push(char j)  // put item on top of stack
      {
      stackArray[++top] = j;
      }

//--------------------------------------------------------------
-
   public char pop()         // take item from top of stack
      {
      return stackArray[top--];
      }

//--------------------------------------------------------------
-
   public char peek()        // peek at top of stack
```

```
      {
      return stackArray[top];
      }

//--------------------------------------------------------------
-
   public boolean isEmpty()  // true if stack is empty
      {
      return (top == -1);
      }

//--------------------------------------------------------------
-
   }  // end class StackX

////////////////////////////////////////////////////////////////

class Reverser
   {
   private String input;                // input string
   private String output;               // output string

//--------------------------------------------------------------
-
   public Reverser(String in)           // constructor
      { input = in; }

//--------------------------------------------------------------
-
   public String doRev()                // reverse the string
      {
      int stackSize = input.length();   // get max stack size
      StackX theStack = new StackX(stackSize);  // make stack

      for(int j=0; j<input.length(); j++)
         {
         char ch = input.charAt(j);     // get a char from
input
         theStack.push(ch);             // push it
         }
      output = "";
      while( !theStack.isEmpty() )
         {
         char ch = theStack.pop();      // pop a char,
         output = output + ch;          // append to output
         }
      return output;
      }  // end doRev()

//--------------------------------------------------------------
-
   }  // end class Reverser

////////////////////////////////////////////////////////////////
```

```
class ReverseApp
   {
   public static void main(String[] args) throws IOException
      {
      String input, output;
      while(true)
         {
         System.out.print("Enter a string: ");
         System.out.flush();
         input = getString();          // read a string from
kbd
         if( input.equals("") )        // quit if [Enter]
            break;
                                       // make a Reverser
         Reverser theReverser = new Reverser(input);
         output = theReverser.doRev(); // use it
         System.out.println("Reversed: " + output);
         }  // end while
      }  // end main()

//--------------------------------------------------------------
-
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }

//--------------------------------------------------------------
-

   }  // end class ReverseApp
```

We've created a class `Reverser` to handle the reversing of the input string. Its key component is the method `doRev()`, which carries out the reversal, using a stack. The stack is created within `doRev()`, which sizes it according to the length of the input string.

In `main()` we get a string from the user, create a `Reverser` object with this string as an argument to the constructor, call this object's `doRev()` method, and display the return value, which is the reversed string. Here's some sample interaction with the program:

```
Enter a string: part
Reversed: trap
Enter a string:
```

Stack Example 2: Delimiter Matching

One common use for stacks is to parse certain kinds of text strings. Typically the strings are lines of code in a computer language, and the programs parsing them are compilers.

To give the flavor of what's involved, we'll show a program that checks the delimiters in a line of text typed by the user. This text doesn't need to be a line of real Java code (although it could be) but it should use delimiters the same way Java does. The delimiters are the braces '{'and'}', brackets '['and']', and parentheses '('and')'. Each opening or left delimiter should be matched by a closing or right delimiter; that is, every '{' should be followed by a matching '}' and so on. Also, opening delimiters that occur later in the string should be closed before those occurring earlier. Examples:

```
c[d]         // correct
a{b[c]d}e    // correct
a{b(c]d}e    // not correct; ] doesn't match (
a[b{c}d]e}   // not correct; nothing matches final }
a{b(c)       // not correct; Nothing matches opening {
```

Opening Delimiters on the Stack

The program works by reading characters from the string one at a time and placing opening delimiters, when it finds them, on a stack. When it reads a closing delimiter from the input, it pops the opening delimiter from the top of the stack and attempts to match it with the closing delimiter. If they're not the same type (there's an opening brace but a closing parenthesis, for example), then an error has occurred. Also, if there is no opening delimiter on the stack to match a closing one, or if a delimiter has not been matched, an error has occurred. A delimiter that hasn't been matched is discovered because it remains on the stack after all the characters in the string have been read.

Let's see what happens on the stack for a typical correct string:

```
a{b(c[d]e)f}
```

Table 4.1 shows how the stack looks as each character is read from this string. The stack contents are shown in the second column. The entries in this column show the stack contents, reading from the bottom of the stack on the left to the top on the right.

As it's read, each opening delimiter is placed on the stack. Each closing delimiter read from the input is matched with the opening delimiter popped from the top of the stack. If they form a pair, all is well. Nondelimiter characters are not inserted on the stack; they're ignored.

Table 4.1: Stack contents in delimiter matching

Character Read	Stack Contents
A	
{	{
B	{
(	{(
C	{(
[	{([

D	{([
]	{(
E	{(
)	{
F	{
}	

This approach works because pairs of delimiters that are opened last should be closed first. This matches the last-in-first-out property of the stack.

Java Code for `brackets.java`

The code for the parsing program, `brackets.java`, is shown in Listing 4.3. We've placed `check()`, the method that does the parsing, in a class called `BracketChecker`.

Listing 4.3 The brackets.java Program

```
// brackets.java
// stacks used to check matching brackets
// to run this program: C>java BracketsApp
import java.io.*;                 // for I/O
////////////////////////////////////////////////////////////////
class StackX
   {
   private int maxSize;
   private char[] stackArray;
   private int top;

//--------------------------------------------------------------
-
   public StackX(int s)       // constructor
      {
      maxSize = s;
      stackArray = new char[maxSize];
      top = -1;
      }

//--------------------------------------------------------------
-
   public void push(char j)  // put item on top of stack
      {
      stackArray[++top] = j;
      }

//--------------------------------------------------------------
-
```

```
   public char pop()         // take item from top of stack
      {
      return stackArray[top--];
      }

//--------------------------------------------------------------
-
   public char peek()        // peek at top of stack
      {
      return stackArray[top];
      }

//--------------------------------------------------------------
-
   public boolean isEmpty()    // true if stack is empty
      {
      return (top == -1);
      }

//--------------------------------------------------------------
-
   }  // end class StackX

////////////////////////////////////////////////////////////////

class BracketChecker
   {
   private String input;                  // input string

//--------------------------------------------------------------
-
   public BracketChecker(String in)        // constructor
      { input = in; }

//--------------------------------------------------------------
-
   public void check()
      {
      int stackSize = input.length();      // get max stack
size
      StackX theStack = new StackX(stackSize);  // make stack

      for(int j=0; j<input.length(); j++)  // get chars in turn
         {
         char ch = input.charAt(j);        // get char
         switch(ch)
            {
            case '{':                      // opening symbols
            case '[':
            case '(':
               theStack.push(ch);          // push them
               break;

            case '}':                      // closing symbols
            case ']':
            case ')':
               if( !theStack.isEmpty() )   // if stack not
empty,
                  {
                  char chx = theStack.pop();  // pop and check
                  if( (ch=='}' && chx!='{') ||
                      (ch==']' && chx!='[') ||
                      (ch==')' && chx!='(') )
                     System.out.println("Error: "+ch+" at "+j);
                  }
               else                        // prematurely empty
                  System.out.println("Error: "+ch+" at "+j);
               break;
            default:    // no action on other characters
               break;
            }  // end switch
         }  // end for
      // at this point, all characters have been processed
      if( !theStack.isEmpty() )
         System.out.println("Error: missing right delimiter");
      }  // end check()

//--------------------------------------------------------------
-
   }  // end class BracketChecker

////////////////////////////////////////////////////////////////

class BracketsApp
   {
   public static void main(String[] args) throws IOException
      {
      String input;
      while(true)
         {
         System.out.print(
                      "Enter string containing delimiters: ");
         System.out.flush();
         input = getString();     // read a string from kbd
         if( input.equals("") )   // quit if [Enter]
            break;
                                  // make a BracketChecker
         BracketChecker theChecker = new BracketChecker(input);
         theChecker.check();      // check brackets
         }  // end while
      }  // end main()

//--------------------------------------------------------------
-
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
```

```
         BufferedReader br = new BufferedReader(isr);
         String s = br.readLine();
         return s;
         }

   //--------------------------------------------------------------
   -

      }  // end class BracketsApp
```

The `check()` routine makes use of the `StackX` class from the last program. Notice how easy it is to reuse this class. All the code you need is in one place. This is one of the payoffs for object-oriented programming.

The `main()` routine in the `BracketsApp` class repeatedly reads a line of text from the user, creates a `BracketChecker` object with this text string as an argument, and then calls the `check()` method for this `BracketChecker` object. If it finds any errors, the `check()` method displays them; otherwise, the syntax of the delimiters is correct.

If it can, the `check()` method reports the character number where it discovered the error (starting at 0 on the left), and the incorrect character it found there. For example, for the input string

```
a{b(c]d}e
```

the output from `check()` will be

```
Error: ] at 5
```

The Stack as a Conceptual Aid

Notice how convenient the stack is in the `brackets.java` program. You could have set up an array to do what the stack does, but you would have had to worry about keeping track of an index to the most recently added character, as well as other bookkeeping tasks. The stack is conceptually easier to use. By providing limited access to its contents, using the `push()` and `pop()` methods, the stack has made your program easier to understand and less error prone. (Carpenters will also tell you it's safer to use the right tool for the job.)

Efficiency of Stacks

Items can be both pushed and popped from the stack implemented in the `StackX` class in constant O(1) time. That is, the time is not dependent on how many items are in the stack, and is therefore very quick. No comparisons or moves are necessary.

Queues

The word *queue* is British for *line* (the kind you wait in). In Britain, to "queue up" means to get in line. In computer science a queue is a data structure that is similar to a stack, except that in a queue the first item inserted is the first to be removed (FIFO), while in a stack, as we've seen, the last item inserted is the first to be removed (LIFO). A queue works like the line at the movies: the first person to join the rear of the line is the first person to reach the front of the line and buy a ticket. The last person to line up is the last person to buy a ticket (or—if the show is sold out—to fail to buy a ticket). Figure 4.4 shows how this looks.

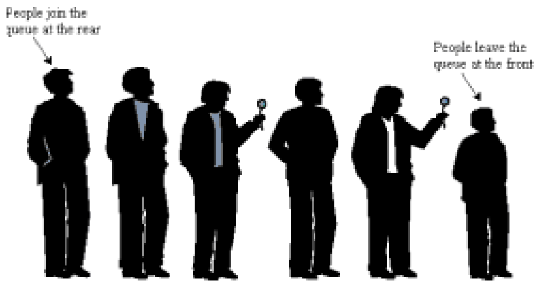

People join the queue at the rear
People leave the queue at the front

Figure 4.4: A queue of people

Queues are used as a programmer's tool as stacks are. We'll see an example where a queue helps search a graph in Chapter 13. They're also used to model real-world situations such as people waiting in line at a bank, airplanes waiting to take off, or data packets waiting to be transmitted over the Internet.

There are various queues quietly doing their job in your computer's (or the network's) operating system. There's a printer queue where print jobs wait for the printer to be available. A queue also stores keystroke data as you type at the keyboard. This way, if you're using a word processor but the computer is briefly doing something else when you hit a key, the keystroke won't be lost; it waits in the queue until the word processor has time to read it. Using a queue guarantees the keystrokes stay in order until they can be processed.

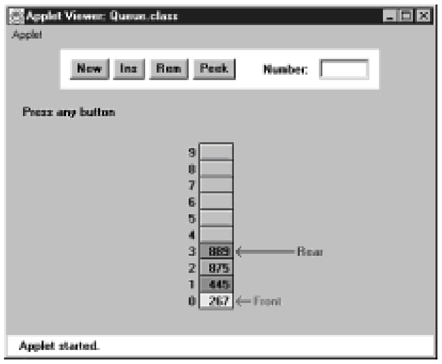
The Queue Workshop Applet

Start up the Queue Workshop applet. You'll see a queue with four items preinstalled, as shown in Figure 4.5.

This applet demonstrates a queue based on an array. This is a common approach, although linked lists are also commonly used to implement queues.

The two basic queue operations are inserting an item, which is placed at the rear of the queue, and *removing* an item, which is taken from the front of the queue. This is similar to a person joining the rear of a line of movie-goers, and, having arrived at the front of the line and purchased a ticket, removing themselves from the front of the line.

The terms for insertion and removal in a stack are fairly standard; everyone says *push* and *pop*. Standardization hasn't progressed this far with queues. *Insert* is also called *put* or *add* or *enque*, while *remove* may be called *delete*



Press any button

Applet started.

Figure 4.5: The Queue Workshop applet

The rear of the queue, where items are inserted, is also called the *back* or *tail* or *end*. The front, where items are removed, may also be called the *head*. We'll use the terms insert, remove, front, and rear.

Insert

By repeatedly pressing the Ins button in the Queue Workshop applet, you can insert a new item. After the first press, you're prompted to enter a key value for a new item into the Number text field; this should be a number from 0 to 999. Subsequent presses will insert an item with this key at the rear of the queue and increment the Rear arrow so it points to the new item.

Remove

Similarly, you can remove the item at the front of the queue using the Rem button. The person is removed, the person's value is stored in the Number field (corresponding to the `remove()` method returning a value) and the Front arrow is incremented. In the applet, the cell that held the deleted item is grayed to show it's gone. In a normal implementation, it would remain in memory but would not be accessible because Front had moved past it. The insert and remove operations are shown in Figure 4.6.

Unlike the situation in a stack, the items in a queue don't always extend all the way down to index 0 in the array. Once some items are removed, Front will point at a cell with a higher index, as shown in Figure 4.7.

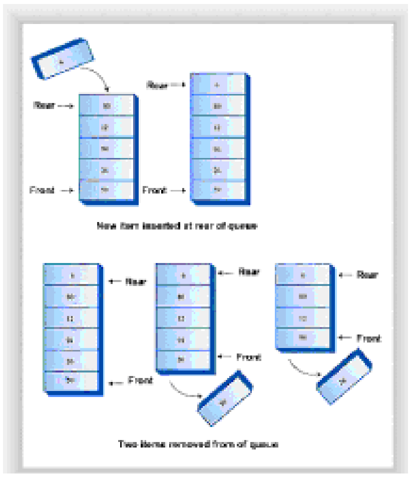

Rear
Front
Rear
Front
New item inserted at rear of queue
Rear
Front
Rear
Front
Rear
Front
Two items removed from of queue

Figure 4.6: Operation of the `Queue` class methods

Notice that in this figure Front lies below Rear in the array; that is, Front has a lower index. As we'll see in a moment, this isn't always true.

Peek

We show one other queue operation, peek. This finds the value of the item at the front of the queue without removing the item. (Like insert and remove, peek when applied to a queue is also called by a variety of other names.) If you press the Peek button, you'll see the value at Front transferred to the Number box. The queue is unchanged.

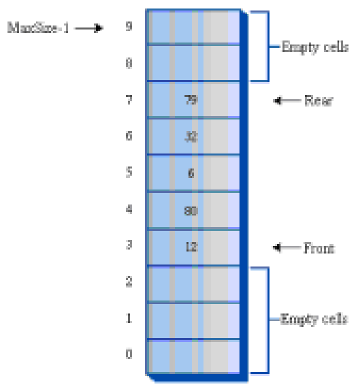


Figure 4.7: A queue with some items removed

This `peek()` method returns the value at the front of the queue. Some queue implementations have a `rearPeek()` and a `frontPeek()` method, but usually you want to know what you're about to remove, not what you just inserted.

New

If you want to start with an empty queue, you can use the New button to create one.

Empty and Full

If you try to remove an item when there are no more items in the queue, you'll get the `Can't remove, queue is empty` error message. If you try to insert an item when all the cells are already occupied, you'll get the `Can't insert, queue is full` message.

A Circular Queue

When you insert a new item in the queue in the Workshop applet, the Front arrow moves upward, toward higher numbers in the array. When you remove an item, Rear also moves upward. Try these operations with the Workshop applet to convince yourself it's true. You may find the arrangement counter-intuitive, because the people in a line at the movies all move forward, toward the front, when a person leaves the line. We could move all the items in a queue whenever we deleted one, but that wouldn't be very efficient. Instead we keep all the items in the same place and move the front and rear of the queue.

The trouble with this arrangement is that pretty soon the rear of the queue is at the end of the array (the highest index). Even if there are empty cells at the beginning of the array, because you've removed them with Rem, you still can't insert a new item because Rear can't go any further. Or can it? This situation is shown in Figure 4.8.

Wrapping Around

To avoid the problem of not being able to insert more items into the queue even when it's not full, the Front and Rear arrows *wrap around* to the beginning of the array. The result is a *circular queue* (sometimes called a *ring buffer*).

You can see how wraparound works with the Workshop applet. Insert enough items to bring the Rear arrow to the top of the array (index 9). Remove some items from the front of the array. Now, insert another item. You'll see the Rear arrow wrap around from index 9 to index 0; the new item will be inserted there. This is shown in Figure 4.9.

Insert a few more items. The Rear arrow moves upward as you'd expect. Notice that once Rear has wrapped around, it's now below Front, the reverse of the original arrangement. You can call this a *broken sequence*: the items in the queue are in two different sequences in the array.

Delete enough items so that the Front arrow also wraps around. Now you're back to the original arrangement, with Front below Rear. The items are in a single *contiguous sequence*.

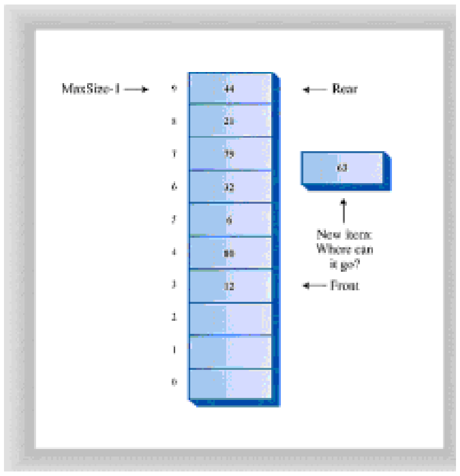


Figure 4.8: Rear arrow at the end of the array

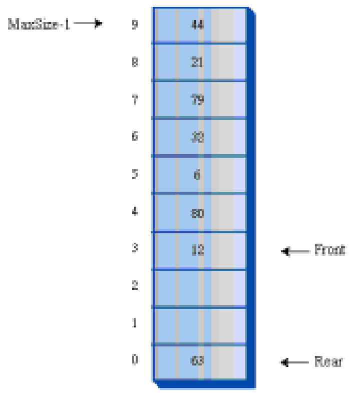


Figure 4.9: Rear arrow wraps around

Java Code for a Queue

The `queue.java` program features a `Queue` class with `insert()`, `remove()`, `peek()`, `isFull()`, `isEmpty()`, and `size()` methods.

The `main()` program creates a queue of five cells, inserts four items, removes three items, and inserts four more. The sixth insertion invokes the wraparound feature. All the items are then removed and displayed. The output looks like this:

```
40 50 60 70 80
```

Listing 4.4 shows the `Queue.java` program.

Listing 4.4 The Queue.java Program

```
// Queue.java
// demonstrates queue
// to run this program: C>java QueueApp
import java.io.*;                 // for I/O
////////////////////////////////////////////////////////////////
class Queue
   {
   private int maxSize;
   private int[] queArray;
   private int front;
   private int rear;
   private int nItems;

//--------------------------------------------------------------
-
   public Queue(int s)         // constructor
      {
      maxSize = s;
      queArray = new int[maxSize];
      front = 0;
      rear = -1;
      nItems = 0;
      }

//--------------------------------------------------------------
-
   public void insert(int j)   // put item at rear of queue
      {
      if(rear == maxSize-1)         // deal with wraparound
         rear = -1;
      queArray[++rear] = j;         // increment rear and
insert
      nItems++;                     // one more item
      }

//--------------------------------------------------------------
-
   public int remove()         // take item from front of queue
      {
      int temp = queArray[front++]; // get value and incr front
      if(front == maxSize)          // deal with wraparound
         front = 0;
      nItems--;                     // one less item
      return temp;
      }

//--------------------------------------------------------------
-
   public int peekFront()      // peek at front of queue
      {
```

```
      return queArray[front];
      }

//--------------------------------------------------------------
-
   public boolean isEmpty()    // true if queue is empty
      {
      return (nItems==0);
      }

//--------------------------------------------------------------
-
   public boolean isFull()     // true if queue is full
      {
      return (nItems==maxSize);
      }

//--------------------------------------------------------------
-
   public int size()           // number of items in queue
      {
      return nItems;
      }

//--------------------------------------------------------------
-
   }  // end class Queue

////////////////////////////////////////////////////////////////

class QueueApp
   {
   public static void main(String[] args)
      {
      Queue theQueue = new Queue(5);  // queue holds 5 items

      theQueue.insert(10);            // insert 4 items
      theQueue.insert(20);
      theQueue.insert(30);
      theQueue.insert(40);

      theQueue.remove();              // remove 3 items
      theQueue.remove();              //    (10, 20, 30)
      theQueue.remove();

      theQueue.insert(50);            // insert 4 more items
      theQueue.insert(60);            //    (wraps around)
      theQueue.insert(70);
      theQueue.insert(80);

      while( !theQueue.isEmpty() )    // remove and display
         {                            //    all items
         int n = theQueue.remove();   // (40, 50, 60, 70, 80)
         System.out.print(n);
```

```
            System.out.print(" ");
            }
         System.out.println("");
         }  // end main()

      }  // end class QueueApp
```

We've chosen an approach in which `Queue` class fields include not only `front` and `rear`, but also the number of items currently in the queue: `nItems`. Some queue implementations don't use this field; we'll show this alternative later.

The `insert()` Method

The `insert()` method assumes that the queue is not full. We don't show it in `main()`, but normally you should only call `insert()` after calling `isFull()` and getting a return value of false. (It's usually preferable to place the check for fullness in the `insert()` routine, and cause an exception to be thrown if an attempt was made to insert into a full queue.)

Normally, insertion involves incrementing `rear` and inserting at the cell `rear` now points to. However, if `rear` is at the top of the array, at `maxSize-1`, then it must wrap around to the bottom of the array before the insertion takes place. This is done by setting `rear` to –1, so when the increment occurs `rear` will become 0, the bottom of the array. Finally `nItems` is incremented.

The `remove()` Method

The `remove()` method assumes that the queue is not empty. You should call `isEmpty()` to ensure this is true before calling `remove()`, or build this error-checking into `remove()`.

Removal always starts by obtaining the value at `front` and then incrementing `front`. However, if this puts `front` beyond the end of the array, it must then be wrapped around to 0. The return value is stored temporarily while this possibility is checked. Finally, `nItems` is decremented.

The `peek()` Method

The `peek()` method is straightforward: it returns the value at `front`. Some implementations allow peeking at the rear of the array as well; such routines are called something like `peekFront()` and `peekRear()` or just `front()` and `rear()`.

The `isEmpty()`, `isFull()`, and `size()` Methods

The `isEmpty()`, `isFull()`, and `size()` methods all rely on the `nItems` field, respectively checking if it's 0, if it's `maxSize`, or returning its value.

Implementation Without an Item Count

The inclusion of the field `nItems` in the `Queue` class imposes a slight overhead on the `insert()` and `remove()` methods in that they must respectively increment and decrement this variable. This may not seem like an excessive penalty, but if you're dealing with huge numbers of insertions and deletions, it might influence performance.

Accordingly, some implementations of queues do without an item count and rely on the `front` and `rear` fields to figure out whether the queue is empty or full and how many items are in it. When this is done, the `isEmpty()`, `isFull()`, and `size()` routines become surprisingly complicated because the sequence of items may be either broken or contiguous, as we've seen.

Also, a strange problem arises. The `front` and `rear` pointers assume certain positions when the queue is full, but they can assume these exact same positions when the queue is empty. The queue can then appear to be full and empty at the same time.

This problem can be solved by making the array one cell larger than the maximum number of items that will be placed in it. Listing 4.5 shows a `Queue` class that implements this no-count approach. This class uses the no-count implementation.

Listing 4.5 The Queue Class Without nItems

```
class Queue
   {
   private int maxSize;
   private int[] queArray;
   private int front;
   private int rear;

//--------------------------------------------------------------
-
   public Queue(int s)          // constructor
      {
      maxSize = s+1;                  // array is 1 cell larger
      queArray = new int[maxSize];  // than requested
      front = 0;
      rear = -1;
      }

//--------------------------------------------------------------
-
   public void insert(int j)   // put item at rear of queue
      {
      if(rear == maxSize-1)
         rear = -1;
      queArray[++rear] = j;
      }

//--------------------------------------------------------------
-
   public int remove()         // take item from front of queue
      {
      int temp = queArray[front++];
      if(front == maxSize)
         front = 0;
      return temp;
      }

//--------------------------------------------------------------
-
   public int peek()           // peek at front of queue
      {
      return queArray[front];
      }

//--------------------------------------------------------------
-
   public boolean isEmpty()    // true if queue is empty
      {
      return ( rear+1==front || (front+maxSize-1==rear) );
      }

//--------------------------------------------------------------
-
   public boolean isFull()     // true if queue is full
      {
      return ( rear+2==front || (front+maxSize-2==rear) );
      }

//--------------------------------------------------------------
-
   public int size()           // (assumes queue not empty)
      {
      if(rear >= front)             // contiguous sequence
         return rear-front+1;
      else                          // broken sequence
         return (maxSize-front) + (rear+1);
      }

//--------------------------------------------------------------
-

   }  // end class Queue
```

Notice the complexity of the `isFull()`, `isEmpty()`, and `size()` methods. This no-count approach is seldom needed in practice, so we'll refrain from discussing it in detail.

Efficiency of Queues

As with a stack, items can be inserted and removed from a queue in O(1) time.

Deques

A *deque* is a double-ended queue. You can insert items at either end and delete them from either end. The methods might be called `insertLeft()` and `insertRight()`, and `removeLeft()` and `removeRight()`.

If you restrict yourself to `insertLeft()` and `removeLeft()` (or their equivalents on the right), then the deque acts like a stack. If you restrict yourself to `insertLeft()` and `removeRight()` (or the opposite pair), then it acts like a queue.

A deque provides a more versatile data structure than either a stack or a queue, and is sometimes used in container class libraries to serve both purposes. However, it's not used as often as stacks and queues, so we won't explore it further here.

Priority Queues

A priority queue is a more specialized data structure than a stack or a queue. However, it's a useful tool in a surprising number of situations. Like an ordinary queue, a priority queue has a front and a rear, and items are removed from the front. However, in a priority queue, items are ordered by key value, so that the item with the lowest key (or in some implementations the highest key) is always at the front. Items are inserted in the proper position to maintain the order.

Here's how the mail sorting analogy applies to a priority queue. Every time the postman hands you a letter, you insert it into your pile of pending letters according to its priority. If it must be answered immediately (the phone company is about to disconnect your modem line), it goes on top, while if it can wait for a leisurely answer (a letter from your Aunt Mabel), it goes on the bottom.

When you have time to answer your mail, you start by taking the letter off the top (the front of the queue), thus ensuring that the most important letters are answered first. This is shown in Figure 4.10.

Like stacks and queues, priority queues are often used as programmer's tools. We'll see one used in finding something called a minimum spanning tree for a graph, in Chapter 14, "Weighted Graphs."

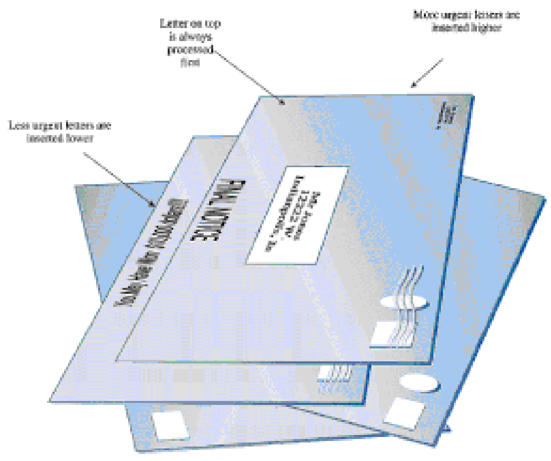

Letter on top is always processed first
More urgent letters are inserted higher
Less urgent letters are inserted lower
FINAL NOTICE

Figure 4.10: Letters in a priority queue

Also, like ordinary queues, priority queues are used in various ways in certain computer systems. In a preemptive multitasking operating system, for example, programs may be placed in a priority queue so the highest-priority program is the next one to receive a time-slice that allows it to execute.

In many situations you want access to the item with the lowest key value (which might represent the cheapest or shortest way to do something). Thus the item with the smallest key has the highest priority. Somewhat arbitrarily, we'll assume that's the case in this discussion, although there are other situations in which the highest key has the highest priority.

Besides providing quick access to the item with the smallest key, you also want a priority queue to provide fairly quick insertion. For this reason, priority queues are, as we noted earlier, often implemented with a data structure called a heap. We'll look at heaps in Chapter 12. In this chapter, we'll show a priority queue implemented by a simple array. This implementation suffers from slow insertion, but it's simpler and is appropriate when the number of items isn't high or insertion speed isn't critical.

The PriorityQ Workshop Applet

The PriorityQ Workshop applet implements a priority queue with an array, in which the items are kept in sorted order. It's an *ascending-priority* queue, in which the item with the smallest key has the highest priority and is accessed with `remove()`. (If the highest-key item were accessed, it would be a *descending-priority* queue.)

The minimum-key item is always at the top (highest index) in the array, and the largest item is always at index 0. Figure 4.11 shows the arrangement when the applet is started. Initially there are five items in the queue.

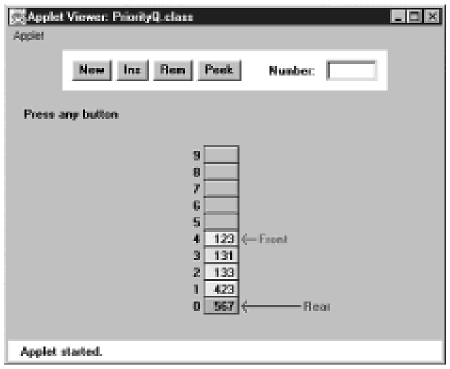


Figure 4.11: The PriorityQ Workshop applet

Insert

Try inserting an item. You'll be prompted to type the new item's key value into the Number field. Choose a number that will be inserted somewhere in the middle of the values already in the queue. For example, in Figure 4.11 you might choose 300. Then, as you repeatedly press Ins, you'll see that the items with smaller keys are shifted up to make room. A black arrow shows which item is being shifted. Once the appropriate position is found, the new item is inserted into the newly created space.

Notice that there's no wraparound in this implementation of the priority queue. Insertion is slow of necessity because the proper in-order position must be found, but deletion is fast. A wraparound implementation wouldn't improve the situation. Note too that the Rear arrow never moves; it always points to index 0 at the bottom of the array.

Delete

The item to be removed is always at the top of the array, so removal is quick and easy; the item is removed and the Front arrow moves down to point to the new top of the array. No comparisons or shifting are necessary.

In the PriorityQ Workshop applet, we show Front and Rear arrows to provide a comparison with an ordinary queue, but they're not really necessary. The algorithms know that the front of the queue is always at the top of the array at `nItems-1`, and they insert items in order, not at the rear. Figure 4.12 shows the operation of the `PriorityQ` class methods.

Peek and New

You can peek at the minimum item (find its value without removing it) with the Peek button, and you can create a new, empty, priority queue with the New button.

Other Implementation Possibilities

The implementation shown in the PriorityQ Workshop applet isn't very efficient for insertion, which involves moving an average of half the items.

Another approach, which also uses an array, makes no attempt to keep the items in sorted order. New items are simply inserted at the top of the array. This makes insertion very quick, but unfortunately it makes deletion slow, because the smallest item must be searched for. This requires examining all the items and shifting half of them, on the average, down to fill in the hole. Generally, the quick-deletion approach shown in the Workshop applet is preferred.

For small numbers of items, or situations where speed isn't critical, implementing a priority queue with an array is satisfactory. For larger numbers of items, or when speed is critical, the heap is a better choice.

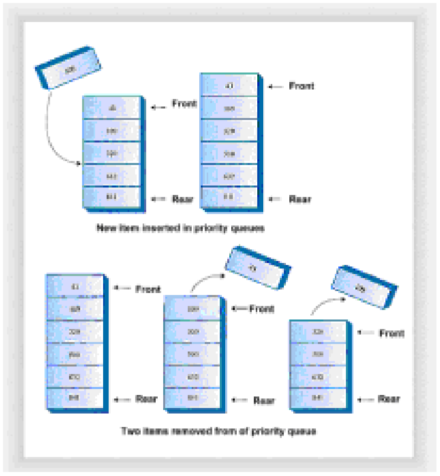

Front
Rear
Front
Rear
New item inserted in priority queues
Front
Rear
Front
Rear
Front
Rear
Two items removed from of priority queue

Figure 4.12: Operation of the `PriorityQ` class methods

Java Code for a Priority Queue

The Java code for a simple array-based priority queue is shown in Listing 4.6.

Listing 4.6 The priorityQ.java Program

```
// priorityQ.java
// demonstrates priority queue
// to run this program: C>java PriorityQApp
import java.io.*;                  // for I/O
////////////////////////////////////////////////////////////////

class PriorityQ
   {
   // array in sorted order, from max at 0 to min at size-1
   private int maxSize;
   private double[] queArray;
   private int nItems;

//-------------------------------------------------------------
   public PriorityQ(int s)          // constructor
      {
```

```
      maxSize = s;
      queArray = new double[maxSize];
      nItems = 0;
      }

//-------------------------------------------------------------
   public void insert(double item)  // insert item
      {
      int j;

      if(nItems==0)                         // if no items,
         queArray[nItems++] = item;         // insert at 0
      else                                  // if any items,
         {
         for(j=nItems-1; j>=0; j--)         // start at end,
            {
            if( item > queArray[j] )        // if new item
larger,
               queArray[j+1] = queArray[j]; // shift upward
            else                            // if smaller,
               break;                       // done shifting
            }  // end for
         queArray[j+1] = item;              // insert it
         nItems++;
         }  // end else (nItems > 0)
      }  // end insert()

//-------------------------------------------------------------
   public double remove()           // remove minimum item
      { return queArray[--nItems]; }

//-------------------------------------------------------------
   public double peekMin()          // peek at minimum item
      { return queArray[nItems-1]; }

//-------------------------------------------------------------
   public boolean isEmpty()         // true if queue is empty
      { return (nItems==0); }

//-------------------------------------------------------------
   public boolean isFull()          // true if queue is full
      { return (nItems == maxSize); }

//-------------------------------------------------------------
   }  // end class PriorityQ

////////////////////////////////////////////////////////////////

class PriorityQApp
   {
   public static void main(String[] args) throws IOException
      {
      PriorityQ thePQ = new PriorityQ(5);
      thePQ.insert(30);
      thePQ.insert(50);
      thePQ.insert(10);
      thePQ.insert(40);
      thePQ.insert(20);

      while( !thePQ.isEmpty() )
         {
        double item = thePQ.remove();
         System.out.print(item + " ");  // 10, 20, 30, 40, 50
         }  // end while
      System.out.println("");
      }  // end main()

//--------------------------------------------------------------

   }  // end class PriorityQApp
```

In `main()` we insert five items in random order and then remove and display them. The smallest item is always removed first, so the output is

```
10, 20, 30, 40, 50
```

The `insert()` method checks if there are any items; if not, it inserts one at index 0. Otherwise, it starts at the top of the array and shifts existing items upward until it finds the place where the new item should go. Then it inserts it and increments `nItems`. Note that if there's any chance the priority queue is full you should check for this possibility with `isFull()` before using `insert()`.

The `front` and `rear` fields aren't necessary as they were in the `Queue` class, because, as we noted, `front` is always at `nItems-1` and `rear` is always at 0.

The `remove()` method is simplicity itself: it decrements `nItems` and returns the item from the top of the array. The `peekMin()` method is similar, except it doesn't decrement `nItems`. The `isEmpty()` and `isFull()` methods check if `nItems` is 0 or `maxSize`, respectively.

Efficiency of Priority Queues

In the priority-queue implementation we show here, insertion runs in O(N) time, while deletion takes O(1) time. We'll see how to improve insertion time with heaps in Chapter 12.

Parsing Arithmetic Expressions

So far in this chapter, we've introduced three different data storage structures. Let's shift gears now and focus on an important application for one of these structures. This application is *parsing* (that is, analyzing) arithmetic expressions like 2+3 or 2*(3+4) or ((2+4)*7)+3*(9–5), and the storage structure it uses is the stack. In the `brackets.java` program, we saw how a stack could be used to check whether delimiters were formatted correctly. Stacks are used in a similar, although more complicated, way for parsing arithmetic expressions.

In some sense this section should be considered optional. It's not a prerequisite to the rest of the book, and writing code to parse arithmetic expressions is probably not something you need to do every day, unless you are a compiler writer or are designing pocket calculators. Also, the coding details are more complex than any we've seen so far. However, it's educational to see this important use of stacks, and the issues raised are interesting in their own right.

As it turns out, it's fairly difficult, at least for a computer algorithm, to evaluate an arithmetic expression directly. It's easier for the algorithm to use a two-step process:

1. Transform the arithmetic expression into a different format, called postfix notation.

2. Evaluate the postfix expression.

Step 1 is a bit involved, but step 2 is easy. In any case, this two-step approach results in a simpler algorithm than trying to parse the arithmetic expression directly. Of course, for a human it's easier to parse the ordinary arithmetic expression. We'll return to the difference between the human and computer approaches in a moment.

Before we delve into the details of steps 1 and 2, we'll introduce postfix notation.

Postfix Notation

Everyday arithmetic expressions are written with an *operator* (+, –, *, or /) placed between two *operands* (numbers, or symbols that stand for numbers). This is called *infix* notation, because the operator is written inside the operands. Thus we say 2+2 and 4/7, or, using letters to stand for numbers, A+B and A/B.

In postfix notation (which is also called Reverse Polish Notation, or RPN, because it was invented by a Polish mathematician), the operator *follows* the two operands. Thus A+B becomes AB+, and A/B becomes AB/. More complex infix expressions can likewise be translated into postfix notation, as shown in Table 4.2. We'll explain how the postfix expressions are generated in a moment.

Table 4.2: Infix and postfix expressions

Infix	Postfix
A+B–C	AB+C–
A*B/C	AB*C/
A+B*C	ABC*+
A*B+C	AB*C+
A*(B+C)	ABC+*
A*B+C*D	AB*CD*+
(A+B)*(C–D)	AB+CD–*
((A+B)*C)–D	AB+C*D–

A+B*(C–D/(E+F))	ABCDEF+/–*+

Some computer languages also have an operator for raising a quantity to a power (typically the ^ character), but we'll ignore that possibility in this discussion.

Besides infix and postfix, there's also a *prefix* notation, in which the operator is written before the operands: +AB instead of AB+. This is functionally similar to postfix but seldom used.

Translating Infix to Postfix

The next several pages are devoted to explaining how to translate an expression from infix notation into postfix. This is a fairly involved algorithm, so don't worry if every detail isn't clear at first. If you get bogged down, you may want to skip ahead to the section, "Evaluating Postfix Expressions." In understanding how to create a postfix expression, it may be helpful to see how a postfix expression is evaluated; for example, how the value 14 is extracted from the expression 234+*, which is the postfix equivalent of 2*(3+4). (Notice that in this discussion, for ease of writing, we restrict ourselves to expressions with single-digit numbers, although these expressions may evaluate to multidigit numbers.)

How Humans Evaluate Infix

How do you translate infix to postfix? Let's examine a slightly easier question first: how does a human evaluate a normal infix expression? Although, as we stated earlier, this is difficult for a computer, we humans do it fairly easily because of countless hours in Mr. Klemmer's math class. It's not hard for us to find the answer to 3+4+5, or 3*(4+5). By analyzing how we do this, we can achieve some insight into the translation of such expressions into postfix.

Roughly speaking, when you "solve" an arithmetic expression, you follow rules something like this:

1. You read from left to right. (At least we'll assume this is true. Sometimes people skip ahead, but for purposes of this discussion, you should assume you must read methodically, starting at the left.)

2. When you've read enough to evaluate two operands and an operator, you do the calculation and substitute the answer for these two operands and operator. (You may also need to solve other pending operations on the left, as we'll see later.)

3. This process is continued—going from left to right and evaluating when possible—until the end of the expression.

In Tables 4.3, 4.4, and 4.5 we're going to show three examples of how simple infix expressions are evaluated. Later, in Tables 4.6, 4.7, and 4.8, we'll see how closely these evaluations mirror the process of translating infix to postfix.

To evaluate 3+4–5, you would carry out the steps shown in Table 4.3.

Table 4.3: Evaluating 3+4–5

Item Read	Expression Parsed So Far	Comments

3	3	
+	3+	
4	3+4	
–	7	When you see the –, you can evaluate 3+4.
	7–	
5	7–5	
End	2	When you reach the end of the expression, you can evaluate 7–5.

You can't evaluate the 3+4 until you see what operator follows the 4. If it's a * or / you need to wait before applying the + sign until you've evaluated the * or /.

However, in this example the operator following the 4 is a –, which has the same precedence as a +, so when you see the – you know you can evaluate 3+4, which is 7. The 7 then replaces the 3+4. You can evaluate the 7–5 when you arrive at the end of the expression.

Figure 4.13 shows how this looks in more detail. Notice how you go from left to right reading items from the input, and then, when you have enough information, you go from right to left, recalling previously examined input and evaluating each operand-operator-operand combination.

Because of precedence relationships, it's a bit more complicated to evaluate 3+4*5, as shown in Table 4.4.

Table 4.4: Evaluating 3+4*5

Item Read	Expression Parsed So Far	Comments
3	3	
+	3+	
4	3+4	
*	3+4*	Can't evaluate 3+4, because * is higher precedence than +.
5	3+4*5	When you see the 5, you can evaluate 4*5.

	3+20	
End	23	When you see the end of the expression, you can evaluate 3+20.

Here you can't add the 3 until you know the result of 4*5. Why not? Because multiplication has a higher precedence than addition. In fact, both * and / have a higher precedence than + and –, so all multiplications and divisions must be carried out before any additions or subtractions (unless parentheses dictate otherwise; see the next example).

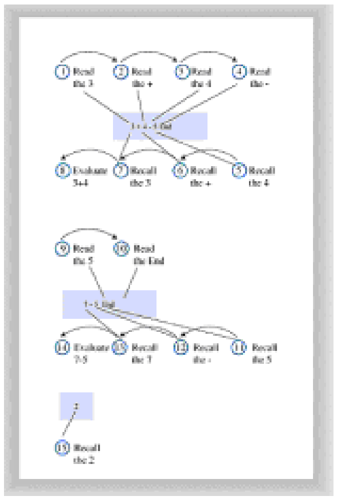


Figure 4.13: Evaluating 3+4*5

Often you can evaluate as you go from left to right, as in the last example. However, you need to be sure, when you come to an operand-operator-operand combination like A+B, that the operator on the right side of the B isn't one with a higher precedence than the +. If it does have a higher precedence, as in this example, you can't do the addition yet. However, once you've read the 5, the multiplication can be carried out because it has the highest priority; it doesn't matter if a * or / follows the 5. However, you still can't do the addition until you've found out what's beyond the 5. When you find there's nothing beyond the 5 but the end of the expression, you can go ahead and do the addition. Figure 4.14 shows this process.

Parentheses can by used to override the normal precedence of operators. Table 4.5 shows how you would evaluate 3*(4+5). Without the parentheses you'd do the multiplication first; with them you do the addition first.

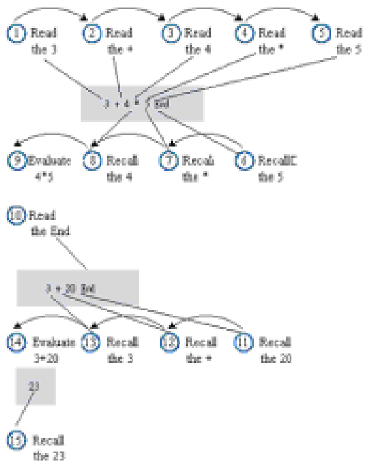


Figure 4.14: Evaluating 3*(4+5)

Table 4.5: Evaluating 3*(4+5)

Item Read	Expression Parsed So Far	Comments
3	3	
*	3*	
(	3*(	
4	3*(4	Can't evaluate 3*4 because of parentheses.
+	3*(4+	
5	3*(4+5	Can't evaluate 4+5 yet.
)	3*(4+5)	When you see the ')' you can evaluate 4+5.
	3*9	When you've evaluated 4+5, you can evaluate 3*9.
	27	
End		Nothing left to evaluate.

Here we can't evaluate anything until we've reached the closing parenthesis. Multiplication has a higher or equal precedence compared to the other operators, so ordinarily we could carry out 3*4 as soon as we see the 4. However, parentheses have an even higher precedence than * and /. Accordingly, we must evaluate anything in parentheses before using the result as an operand in any other calculation. The closing parenthesis tells us we can go ahead and do the addition. We find that 4+5 is 9, and once we know this, we can evaluate 3*9 to obtain 27. Reaching the end of the expression is an anticlimax because there's nothing left to evaluate. The process is shown in Figure 4.15.

As we've seen, in evaluating an infix arithmetic expression, you go both forward and backward through the expression. You go forward (left to right) reading operands and operators. When you have enough information to apply an operator, you go backward, recalling two operands and an operator and carrying out the arithmetic.

Sometimes you must defer applying operators if they're followed by higher precedence operators or by parentheses. When this happens you must apply the later, higher-precedence, operator first; then go backward (to the left) and apply earlier operators.

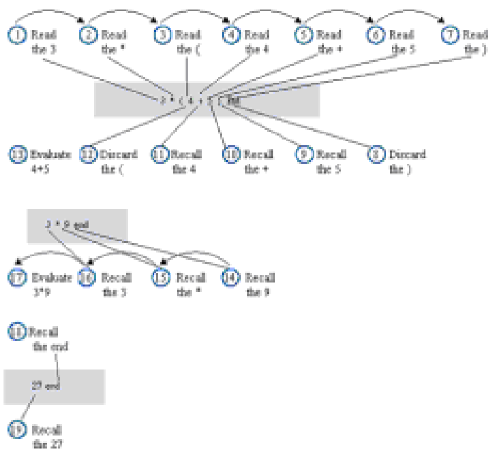


Figure 4.15: Evaluating 3*(4+5)

We could write an algorithm to carry out this kind of evaluation directly. However, as we noted, it's actually easier to translate into postfix notation first.

How Humans Translate Infix to Postfix

To translate infix to postfix notation, you follow a similar set of rules to those for evaluating infix. However, there are a few small changes. You don't do any arithmetic. The idea is not to evaluate the infix expression, but to rearrange the operators and operands into a different format: postfix notation. The resulting postfix expression will be evaluated later.

As before, you read the infix from left to right, looking at each character in turn. As you go along, you copy these operands and operators to the postfix output string. The trick is knowing when to copy what.

If the character in the infix string is an operand, you copy it immediately to the postfix string. That is, if you see an A in the infix, you write an A to the postfix. There's never any delay: you copy the operands as you get to them, no matter how long you must wait to copy their associated operators.

Knowing when to copy an operator is more complicated, but it's the same as the rule for evaluating infix expressions. Whenever you could have used the operator to evaluate part of the infix expression (if you were evaluating instead of translating to postfix), you instead copy it to the postfix string.

Table 4.6 shows how A+B–C is translated into postfix notation.

Table 4.6: Translating A+B–C into postfix

Character Read from Infix Expression	Infix Expression Parsed So Far	Postfix Expression Written So Far	Comments
A	A	A	
+	A+	A	
B	A+B	AB	
–	A+B–	AB+	When you see the –, you can copy the + to the postfix string.
C	A+B–C	AB+C	
End	A+B–C	AB+C–	When you reach the end of the expression, you can copy the –.

Notice the similarity of this table to Table 4.3, which showed the evaluation of the infix expression 3+4–5. At each point where you would have done an evaluation in the earlier table, you instead simply write an operator to the postfix output.

Table 4.7 shows the translation of A+B*C to postfix. This is similar to Table 4.4, which covered the evaluation of 3+4*5.

Table 4.7: Translating A+B*C to postfix

Character Read from Infix Expression	Infix Expression Parsed So Far	Postfix Expression Written So Far	Comments
A	A	A	
+	A+	A	
B	A+B	AB	

*	A+B*	AB	Can't copy the +, because * is higher precedence than +.
C	A+B*C	ABC	When you see the C, you can copy the *.
	A+B*C	ABC*	
End	A+B*C	ABC*+	When you see the end of the expression, you can copy the +.

As the final example, Table 4.8 shows how A*(B+C) is translated to postfix. This is similar to evaluating 3*(4+5) in Table 4.5. You can't write any postfix operators until you see the closing parenthesis in the input.

Table 4.8: Translating 3*(4+5) into postfix

Character Read from Infix Expression	Infix Expression Parsed So Far	Postfix Expression Written So Far	Comments
A	A	A	
*	A*	A	
(	A*(	A	
B	A*(B	AB	Can't copy * because of parenthesis.
+	A*(B+	AB	
C	A*(B+C	ABC	Can't copy the + yet.
)	A*(B+C)	ABC+	When you see the) you can copy the +.
	A*(B+C)	ABC+*	When you've copied the +, you can copy the*.

End	A*(B+C)	ABC+*	Nothing left to copy.

As in the numerical evaluation process, you go both forward and backward through the infix expression to complete the translation to postfix. You can't write an operator to the output (postfix) string if it's followed by a higher-precedence operator or a left parenthesis. If it is, the higher precedence operator, or the operator in parentheses, must be written to the postfix before the lower priority operator.

Saving Operators on a Stack

You'll notice in both Table 4.7 and Table 4.8 that the order of the operators is reversed going from infix to postfix. Because the first operator can't be copied to the output until the second one has been copied, the operators were output to the postfix string in the opposite order they were read from the infix string. A longer example may make this clearer. Table 4.9 shows the translation to postfix of the infix expression A+B*(C–D). We include a column for stack contents, which we'll explain in a moment.

Table 4.9: Translating A+B*(C–D) to postfix

Character Read from Infix Expression	Infix Expression Parsed So Far	Postfix Expression Written So Far	Stack Contents
A	A	A	
+	A+	A	+
B	A+B	AB	+
*	A+B*	AB	+*
(	A+B*(	AB	+*(
C	A+B*(C	ABC	+*(
–	A+B*(C–	ABC	+*(–
D	A+B*(C–D	ABCD	+*(–
)	A+B*(C–D)	ABCD–	+*(
	A+B*(C–D)	ABCD–	+*(
	A+B*(C–D)	ABCD–	+*
	A+B*(C–D)	ABCD–*	+
	A+B*(C–D)	ABCD–*+	

Here we see the order of the operands is +*– in the original infix expression, but the reverse order, –*+, in the final postfix expression. This happens because * has higher precedence than +, and –, because it's in parentheses, has higher precedence than *.

This order reversal suggests a stack might be a good place to store the operators while we're waiting to use them. The last column in Table 4.9 shows the stack contents at various stages in the translation process.

Popping items from the stack allows you to, in a sense, go backward (right to left) through the input string. You're not really examining the entire input string, only the operators and parentheses. These were pushed on the stack when reading the input, so now you can recall them in reverse order by popping them off the stack.

The operands (A, B, and so on) appear in the same order in infix and postfix, so you can write each one to the output as soon as you encounter it; they don't need to be stored on a stack.

Translation Rules

Let's make the rules for infix-to-postfix translation more explicit. You read items from the infix input string and take the actions shown in Table 4.10. These actions are described in pseudocode, a blend of Java and English.

In this table, the < and >= symbols refer to the operator precedence relationship, not numerical values. The `opThis` operator has just been read from the infix input, while the `opTop` operator has just been popped off the stack.

Table 4.10: Translation rules

Item Read from Input(Infix)	Action
Operand	Write it to output (postfix)
Open parenthesis (	Push it on stack
Close parenthesis)	While stack not empty, repeat the following:
	Pop an item,
	If item is not (, write it to output
	Quit loop if item is (
Operator (`opThis`)	If stack empty,
	Push `opThis`
	Otherwise,

	While stack not empty, repeat:
	Pop an item,
	If item is (, push it, or
	If item is an operator (`opTop`), and
	If `opTop < opThis`, push `opTop`, or
	If `opTop >= opThis`, output `opTop`
	Quit loop if `opTop < opThis` or item is (
	Push `opThis`
No more items	While stack not empty,
	Pop item, output it.

It may take some work to convince yourself that these rules work. Tables 4.11, 4.12, and 4.13 show how the rules apply to three sample infix expressions. These are similar to Tables 4.6, 4.7, and 4.8, except that the relevant rules for each step have been added. Try creating similar tables by starting with other simple infix expressions and using the rules to translate some of them to postfix.

Table 4.11: Translation Rules Applied to A+B–C

Character Read from Infix	Infix Parsed So Far	Postfix Written So Far	Stack Contents	Rule
A	A	A		Write operand to output.
+	A+	A	+	If stack empty, push `opThis`.
B	A+B	AB	+	Write operand to output.
–	A+B–	AB		Stack not empty, so pop item.
	A+B–	AB+		`opThis` is –, `opTop` is +, `opTop>=opThis`, so output `opTop`.

	A+B–	AB+	–	Then push `opThis`.
C	A+B–C	AB+C	–	Write operand to output.
End	A+B–C	AB+C-		Pop leftover item, output it.

Table 4.12: Translation rules applied to A+B*C

Character Read from Infix	Infix Parsed So Far	Postfix Written So Far	Stack Contents	Rule
A	A	A		Write operand to postfix.
+	A+	A	+	If stack empty, push `opThis`.
B	A+B	AB	+	Write operand to output.
*	A+B*	AB	+	Stack not empty, so pop `opTop`.
	A+B*	AB	+	`opThis` is *, `opTop` is + `opTop<opThis`, so push `opTop`.
	A+B*	AB	+*	Then push `opThis`.
C	A+B*C	ABC	+*	Write operand to output.
End	A+B*C	ABC*	+	Pop leftover item, output it.
	A+B*C	ABC*+		Pop leftover item, output it.

Table 4.13: Translation Rules Applied to A*(B+C)

Character Read from Infix	Infix Parsed So Far	Postfix Written So Far	Stack Contents	Rule
A	A	A		Write operand to postfix.
*	A*	A	*	If stack empty, push `opThis`.
(	A*(	A	*(	Push (on stack.
B	A*(B	AB	*(	Write operand to postfix.
+	A*(B+	AB	*	Stack not empty, so pop item.
	A*(B+	AB	*(	It's (, so push it.
	A*(B+	AB	*(+	Then push `opThis`.
C	A*(B+C	ABC	*(+	Write operand to postfix.
)	A*(B+C)	ABC+	*(	Pop item, write to output.
	A*(B+C)	ABC+	*	Quit popping if (.
End	A*(B+C)	ABC+*		Pop leftover item, output it.

Java Code to Convert Infix to Postfix

Listing 4.7 shows the `infix.java` program, which uses the rules of Table 4.10 to translate an infix expression to a postfix expression.

Listing 4.7 The infix.java Program

```
// infix.java
// converts infix arithmetic expressions to postfix
// to run this program: C>java InfixApp
import java.io.*;              // for I/O
```

```
////////////////////////////////////////////////////////////////
class StackX
   {
   private int maxSize;
   private char[] stackArray;
   private int top;

//--------------------------------------------------------------
-
   public StackX(int s)       // constructor
      {
      maxSize = s;
      stackArray = new char[maxSize];
      top = -1;
      }

//--------------------------------------------------------------
-
   public void push(char j)  // put item on top of stack
      { stackArray[++top] = j; }

//--------------------------------------------------------------
-
   public char pop()         // take item from top of stack
      { return stackArray[top--]; }

//--------------------------------------------------------------
-
   public char peek()        // peek at top of stack
      { return stackArray[top]; }

//--------------------------------------------------------------
-
   public boolean isEmpty()  // true if stack is empty
      { return (top == -1); }

//--------------------------------------------------------------
   public int size()         // return size
      { return top+1; }

//--------------------------------------------------------------
-
   public char peekN(int n)  // return item at index n
      { return stackArray[n]; }

//--------------------------------------------------------------
-
   public void displayStack(String s)
      {
      System.out.print(s);
      System.out.print("Stack (bottom-->top): ");
      for(int j=0; j<size(); j++)
         {
         System.out.print( peekN(j) );
         System.out.print(' ');
         }
      System.out.println("");
      }

//--------------------------------------------------------------
-
   }  // end class StackX

////////////////////////////////////////////////////////////////

                  // infix to postfix conversion
   {
   private StackX theStack;
   private String input;
   private String output = "";

//--------------------------------------------------------------
-
   public InToPost(String in)   // constructor
      {
      input = in;
      int stackSize = input.length();
      theStack = new StackX(stackSize);
      }

//--------------------------------------------------------------
-
   public String doTrans()      // do translation to postfix
      {
      for(int j=0; j<input.length(); j++)
         {
         char ch = input.charAt(j);
         theStack.displayStack("For "+ch+" "); // *diagnostic*
         switch(ch)
            {
            case '+':               // it's + or -
            case '-':
               gotOper(ch, 1);      // go pop operators
               break;               //   (precedence 1)
            case '*':               // it's * or /
            case '/':
               gotOper(ch, 2);      // go pop operators
               break;               //   (precedence 2)
            case '(':               // it's a left paren
               theStack.push(ch);   // push it
               break;
           case ')':               // it's a right paren
               gotParen(ch);        // go pop operators
               break;
            default:                // must be an operand
               output = output + ch; // write it to output
               break;
            }  // end switch
         }  // end for
      while( !theStack.isEmpty() )      // pop remaining opers
         {
         theStack.displayStack("While ");  // *diagnostic*
         output = output + theStack.pop(); // write to output
         }
      theStack.displayStack("End   ");     // *diagnostic*
      return output;                    // return postfix
      }  // end doTrans()

//--------------------------------------------------------------
-
   public  void gotOper(char opThis, int prec1)
      {                                 // got operator from
input
      while( !theStack.isEmpty() )
         {
         char opTop = theStack.pop();
         if( opTop == '(' )             // if it's a '('
            {
            theStack.push(opTop);       // restore '('
            break;
            }
         else                           // it's an operator
            {
            int prec2;                  // precedence of new op

            if(opTop=='+' || opTop=='-') // find new op prec
               prec2 = 1;
            else
               prec2 = 2;
            if(prec2 < prec1)           // if prec of new op
less
               {                        //    than prec of old
               theStack.push(opTop);    // save newly-popped op
               break;
               }
            else                        // prec of new not less
               output = output + opTop;  // than prec of old
            }  // end else (it's an operator)
         }  // end while
      theStack.push(opThis);         // push new operator
      }  // end gotOp()

//--------------------------------------------------------------
-
   public  void gotParen(char ch)
      {                              // got right paren from
input
      while( !theStack.isEmpty() )
         {
         char chx = theStack.pop();
         if( chx == '(' )            // if popped '('
            break;                   // we're done
         else                         // if popped operator
            output = output + chx;  // output it
         }  // end while
      }  // end popOps()

//--------------------------------------------------------------
-
   }  // end class InToPost

////////////////////////////////////////////////////////////////

class InfixApp
   {
   public static void main(String[] args) throws IOException
      {
      String input, output;
      while(true)
         {
         System.out.print("Enter infix: ");
         System.out.flush();
         input = getString();          // read a string from kbd
         if( input.equals("") )        // quit if [Enter]
            break;
                                       // make a translator
         InToPost theTrans = new InToPost(input);
         output = theTrans.doTrans(); // do the translation
         System.out.println("Postfix is " + output + '\n');
         }  // end while
      }  // end main()

//--------------------------------------------------------------
-
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }

//--------------------------------------------------------------
-

   }  // end class InfixApp
```

The `main()` routine in the `InfixApp` class asks the user to enter an infix expression. The input is read with the `readString()` utility method. The program creates an `InToPost` object, initialized with the input string. Then it calls the `doTrans()` method for this object to perform the translation. This method returns the postfix output string, which is displayed.

The `doTrans()` method uses a `switch` statement to handle the various translation rules shown in Table 4.10. It calls the `gotOper()` method when it reads an operator and the `gotParen()` method when it reads a closing parenthesis ')'. These methods implement the second two rules in the table, which are more complex than other rules.

We've included a `displayStack()` method to display the entire contents of the stack in the `StackX` class. In theory, this isn't playing by the rules; you're only supposed to access the item at the top. However, as a diagnostic aid it's useful to see the contents of the stack at each stage of the translation. Here's some sample interaction with `infix.java`:

```
Enter infix: Input=A*(B+C)-D/(E+F)
For A Stack (bottom-->top):
For * Stack (bottom-->top):
For ( Stack (bottom-->top): *
For B Stack (bottom-->top): * (
For + Stack (bottom-->top): * (
For C Stack (bottom-->top): * ( +
For ) Stack (bottom-->top): * ( +
For - Stack (bottom-->top): *
For D Stack (bottom-->top): -
Parsing Arithmetic ExpressionsFor / Stack (bottom-->top): -
For ( Stack (bottom-->top): - /
For E Stack (bottom-->top): - / (
For + Stack (bottom-->top): - / (
For F Stack (bottom-->top): - / ( +
For ) Stack (bottom-->top): - / ( +
While Stack (bottom-->top): - /
While Stack (bottom-->top): -
End   Stack (bottom-->top):
Postfix is ABC+*DEF+/-
```

The output shows where the `displayStack()` method was called (from the `for` loop, the `while` loop, or at the end of the program) and within the `for` loop, what character has just been read from the input string.

You can use single-digit numbers like 3 and 7 instead of symbols like A and B. They're all just characters to the program. For example:

```
Enter infix: Input=2+3*4
For 2 Stack (bottom-->top):
For + Stack (bottom-->top):
For 3 Stack (bottom-->top): +
For * Stack (bottom-->top): +
For 4 Stack (bottom-->top): + *
While Stack (bottom-->top): + *
While Stack (bottom-->top): +
End   Stack (bottom-->top):
Postfix is 234*+
```

Of course, in the postfix output, the 234 means the separate numbers 2, 3, and 4.

The `infix.java` program doesn't check the input for errors. If you type an incorrect infix expression, the program will provide erroneous output or crash and burn.

Experiment with this program. Start with some simple infix expressions, and see if you can predict what the postfix will be. Then run the program to verify your answer. Pretty soon, you'll be a postfix guru, much sought-after at cocktail parties.

Evaluating Postfix Expressions

As you can see, it's not trivial to convert infix expressions to postfix expressions. Is all this trouble really necessary? Yes, the payoff comes when you evaluate a postfix expression. Before we show how simple the algorithm is, let's examine how a human might carry out such an evaluation.

How Humans Evaluate Postfix

Figure 4.16 shows how a human can evaluate a postfix expression using visual inspection and a pencil.

Start with the first operator on the left, and draw a circle around it and the two operands to its immediate left. Then apply the operator to these two operands—performing the actual arithmetic—and write down the result inside the circle. In the figure, evaluating 4+5 gives 9.

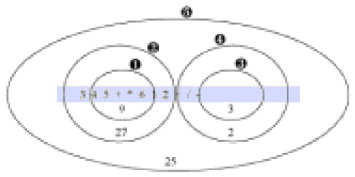


Figure 4.16: Visual approach to postfix evaluation of 345+*612+/-

Now go to the next operator to the right, and draw a circle around it, the circle you already drew, and the operand to the left of that. Apply the operator to the previous circle and the new operand, and write the result in the new circle. Here 3*9 gives 27. Continue this process until all the operators have been applied: 1+2 is 3, and 6/3 is 2. The answer is the result in the largest circle: 27–2 is 25.

Rules for Postfix Evaluation

How do we write a program to reproduce this evaluation process? As you can see, each time you come to an operator, you apply it to the last two operands you've seen. This suggests that it might be appropriate to store the operands on a stack. (This is the opposite of the infix to postfix translation algorithm, where *operators* were stored on the stack.) You can use the rules shown in Table 4.14 to evaluate postfix expressions.

Table 4.14: Evaluating a postfix expression

Item Read from Postfix Expression	Action
Operand	Push it onto the stack.
Operator	Pop the top two operands from the stack, and apply the operator to them. Push the result.

When you're finished, pop the stack to obtain the answer. That's all there is to it. This process is the computer equivalent of the human circle-drawing approach of Figure 4.16.

Java Code to Evaluate Postfix Expressions

In the infix-to-postfix translation, we used symbols (A, B, and so on) to stand for numbers. This worked because we weren't performing arithmetic operations on the operands, but merely rewriting them in a different format.

Now we want to evaluate a postfix expression, which means carrying out the arithmetic and obtaining an answer. Thus the input must consist of actual numbers. To simplify the coding we've restricted the input to single-digit numbers.

Our program evaluates a postfix expression and outputs the result. Remember numbers are restricted to one digit. Here's some simple interaction:

```
Enter postfix: 57+
5 Stack (bottom-->top):
7 Stack (bottom-->top): 5
+ Stack (bottom-->top): 5 7
Evaluates to 12
```

You enter digits and operators, with no spaces. The program finds the numerical equivalent. Although the input is restricted to single-digit numbers, the results are not; it doesn't matter if something evaluates to numbers greater than 9. As in the `infix.java` program, we use the `displayStack()` method to show the stack contents at each step. Listing 4.8 shows the `postfix.java` program.

Listing 4.8 The postfix.java Program

```
// postfix.java
// parses postfix arithmetic expressions
// to run this program: C>java PostfixApp
import java.io.*;              // for I/O
////////////////////////////////////////////////////////////////
class StackX
   {
   private int maxSize;
   private int[] stackArray;
   private int top;

//--------------------------------------------------------------
-
   public StackX(int size)      // constructor
      {
      maxSize = size;
      stackArray = new int[maxSize];
      top = -1;
      }

//--------------------------------------------------------------
-
   public void push(int j)     // put item on top of stack
```

```
      { stackArray[++top] = j; }

//--------------------------------------------------------------
-
   public int pop()            // take item from top of stack
      { return stackArray[top--]; }

//--------------------------------------------------------------
-
   public int peek()           // peek at top of stack
      { return stackArray[top]; }

//--------------------------------------------------------------
-
   public boolean isEmpty()    // true if stack is empty
      { return (top == -1); }

//--------------------------------------------------------------
-
   public boolean isFull()     // true if stack is full
      { return (top == maxSize-1); }

//--------------------------------------------------------------
-
   public int size()           // return size
      { return top+1; }

//--------------------------------------------------------------
-
   public int peekN(int n)     // peek at index n
      { return stackArray[n]; }

//--------------------------------------------------------------
-
   public void displayStack(String s)
      {
      System.out.print(s);
      System.out.print("Stack (bottom-->top): ");
      for(int j=0; j<size(); j++)
         {
         System.out.print( peekN(j) );
         System.out.print(' ');
         }
      System.out.println("");
      }

//--------------------------------------------------------------
-
   }  // end class StackX

////////////////////////////////////////////////////////////////

class ParsePost
   {
   private StackX theStack;
   private String input;

//--------------------------------------------------------------
-
   public ParsePost(String s)
      { input = s; }

//--------------------------------------------------------------
-
   public int doParse()
      {
      theStack = new StackX(20);             // make new stack
      char ch;
      int j;
      int num1, num2, interAns;

      for(j=0; j<input.length(); j++)       // for each char,
         {
         ch = input.charAt(j);              // read from input
         theStack.displayStack(""+ch+" ");  // *diagnostic*
         if(ch >= '0' && ch <= '9')         // if it's a number
            theStack.push( (int)(ch-'0') ); //   push it
         else                               // it's an operator
            {
            num2 = theStack.pop();          // pop operands
            num1 = theStack.pop();
            switch(ch)                      // do arithmetic
               {
               case '+':
                  interAns = num1 + num2;
                  break;
               case '-':
                  interAns = num1 - num2;
                  break;
               case '*':
                  interAns = num1 * num2;
                  break;
               case '/':
                  interAns = num1 / num2;
                  break;
               default:
                  interAns = 0;
               }  // end switch
           theStack.push(interAns);        // push result
            }  // end else
         }  // end for
      interAns = theStack.pop();            // get answer
      return interAns;
      }  // end doParse()
   }  // end class ParsePost

////////////////////////////////////////////////////////////////
```

```
class PostfixApp
   {
   public static void main(String[] args) throws IOException
      {
      String input;
      int output;

      while(true)
         {
         System.out.print("Enter postfix: ");
         System.out.flush();
         input = getString();         // read a string from kbd
         if( input.equals("") )       // quit if [Enter]
            break;
                                      // make a parser
         ParsePost aParser = new ParsePost(input);
         output = aParser.doParse();  // do the evaluation
         System.out.println("Evaluates to " + output);
         }  // end while
      }  // end main()

//--------------------------------------------------------------
-
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }

//--------------------------------------------------------------
-

   }  // end class PostfixApp
```

The `main()` method in the `PostfixApp` class gets the postfix string from the user and then creates a `ParsePost` object, initialized with this string. It then calls the `doParse()` method of `ParsePost` to carry out the evaluation.

The `doParse()` method reads through the input string, character by character. If the character is a digit, it's pushed onto the stack. If it's an operator, it's applied immediately to the two operators on the top of the stack. (These operators are guaranteed to be on the stack already, because the input string is in postfix notation.)

The result of the arithmetic operation is pushed onto the stack. Once the last character (which must be an operator) is read and applied, the stack contains only one item, which is the answer to the entire expression.

Here's some interaction with more complex input: the postfix expression 345+*612+/–, which we showed a human evaluating in Figure 4.16. This corresponds to the infix 3*(4+5)–6/(1+2). (We saw an equivalent translation using letters instead of numbers in the last section: A*(B+C)–D/(E+F) in infix is ABC+*DEF+/– in postfix.) Here's how the postfix is evaluated by the `postfix.java` program:

```
Enter postfix: 345+*612+/-
3 Stack (bottom-->top):
4 Stack (bottom-->top): 3
5 Stack (bottom-->top): 3 4
+ Stack (bottom-->top): 3 4 5
* Stack (bottom-->top): 3 9
6 Stack (bottom-->top): 27
1 Stack (bottom-->top): 27 6
2 Stack (bottom-->top): 27 6 1
+ Stack (bottom-->top): 27 6 1 2
/ Stack (bottom-->top): 27 6 3
- Stack (bottom-->top): 27 2
Evaluates to 25
```

As with the last program, `postfix.java` doesn't check for input errors. If you type in a postfix expression that doesn't make sense, results are unpredictable.

Experiment with the program. Trying different postfix expressions and seeing how they're evaluated will give you an understanding of the process faster than reading about it.

Summary

- Stacks, queues, and priority queues are data structures usually used to simplify certain programming operations.
- In these data structures, only one data item can be accessed.
- A stack allows access to the last item inserted.
- The important stack operations are pushing (inserting) an item onto the top of the stack and popping (removing) the item that's on the top.
- A queue allows access to the first item that was inserted.
- The important queue operations are inserting an item at the rear of the queue and removing the item from the front of the queue.
- A queue can be implemented as a circular queue, which is based on an array in which the indices wrap around from the end of the array to the beginning.
- A priority queue allows access to the smallest (or sometimes the largest) item.
- The important priority queue operations are inserting an item in sorted order and removing the item with the smallest key.
- These data structures can be implemented with arrays or with other mechanisms such as linked lists.
- Ordinary arithmetic expressions are written in infix notation, so-called because the operator is written between the two operands.
- In postfix notation, the operator follows the two operands.
- Arithmetic expressions are typically evaluated by translating them to postfix notation and then evaluating the postfix expression.