Chapter 9

Linked Data Structures and Recursion

In this chapter, we look at two advanced programming techniques, recursion and linked data structures, and some of their applications. Both of these techniques are related to the seemingly paradoxical idea of defining something in terms of itself. This turns out to be a remarkably powerful idea.

A subroutine is said to be recursive if it calls itself, either directly or indirectly. That is, the subroutine is used in its own definition. Recursion can often be used to solve complex problems by reducing them to simpler problems of the same type.

A reference to one object can be stored in an instance variable of another object. The objects are then said to be "linked." Complex data structures can be built by linking objects together. An especially interesting case occurs when an object contains a link to another object that belongs to the same class. In that case, the class is used in its own definition. Several important types of data structures are built using classes of this kind.

9.1 Recursion

At one time or another, you've probably been told that you can't define something in terms of itself. Nevertheless, if it's done right, defining something at least partially in terms of itself can be a very powerful technique. A ***recursive*** definition is one that uses the concept or thing that is being defined as part of the definition. For example: An "ancestor" is either a parent or an ancestor of a parent. A "sentence" can be, among other things, two sentences joined by a conjunction such as "and." A "directory" is a part of a disk drive that can hold files and directories. In mathematics, a "set" is a collection of elements, which can themselves be sets. A "statement" in Java can be a `while` statement, which is made up of the word "while", a boolean-valued condition, and a statement.

Recursive definitions can describe very complex situations with just a few words. A definition of the term "ancestor" without using recursion might go something like "a parent, or a grandparent, or a great-grandparent, or a great-great-grandparent, and so on." But saying "and so on" is not very rigorous. (I've often thought that recursion is really just a rigorous way of saying "and so on.") You run into the same problem if you try to define a "directory" as "a file that is a list of files, where some of the files can be lists of files, where some of **those** files can be lists of files, and so on." Trying to describe what a Java statement can look like, without using recursion in the definition, would be difficult and probably pretty comical.

Recursion can be used as a programming technique. A ***recursive subroutine*** is one that calls itself, either directly or indirectly. To say that a subroutine calls itself directly means that its definition contains a subroutine call statement that calls the subroutine that is being defined. To say that a subroutine calls itself indirectly means that it calls a second subroutine which in turn calls the first subroutine (either directly or indirectly). A recursive subroutine can define a complex task in just a few lines of code. In the rest of this section, we'll look at a variety of examples, and we'll see other examples in the rest of the book.

9.1.1 Recursive Binary Search

Let's start with an example that you've seen before: the binary search algorithm from Subsection 7.4.1. Binary search is used to find a specified value in a sorted list of items (or, if it does not occur in the list, to determine that fact). The idea is to test the element in the middle of the list. If that element is equal to the specified value, you are done. If the specified value is less than the middle element of the list, then you should search for the value in the first half of the list. Otherwise, you should search for the value in the second half of the list. The method used to search for the value in the first or second half of the list is binary search. That is, you look at the middle element in the half of the list that is still under consideration, and either you've found the value you are looking for, or you have to apply binary search to one half of the remaining elements. And so on! This is a recursive description, and we can write a recursive subroutine to implement it.

Before we can do that, though, there are two considerations that we need to take into account. Each of these illustrates an important general fact about recursive subroutines. First of all, the binary search algorithm begins by looking at the "middle element of the list." But what if the list is empty? If there are no elements in the list, then it is impossible to look at the middle element. In the terminology of Subsection 8.2.1, having a non-empty list is a "precondition" for looking at the middle element, and this is a clue that we have to modify the algorithm to take this precondition into account. What should we do if we find ourselves searching for a specified value in an empty list? The answer is easy: If the list is empty, we can be sure that the value does not occur in the list, so we can give the answer without any further work. An empty list is a ***base case*** for the binary search algorithm. A base case for a recursive algorithm is a case that is handled directly, rather than by applying the algorithm recursively. The binary search algorithm actually has another type of base case: If we find the element we are looking for in the middle of the list, we are done. There is no need for further recursion.

The second consideration has to do with the parameters to the subroutine. The problem is phrased in terms of searching for a value in a list. In the original, non-recursive binary search subroutine, the list was given as an array. However, in the recursive approach, we have to able to apply the subroutine recursively to just a **part** of the original list. Where the original subroutine was designed to search an entire array, the recursive subroutine must be able to search part of an array. The parameters to the subroutine must tell it what part of the array to search. This illustrates a general fact that in order to solve a problem recursively, it is often necessary to generalize the problem slightly.

Here is a recursive binary search algorithm that searches for a given value in part of an array of integers:

```
/**
 * Search in the array A in positions numbered loIndex to hiIndex,
 * inclusive, for the specified value.   If the value is found, return
 * the index in the array where it occurs. If the value is not found,
 * return -1.  Precondition:  The array must be sorted into increasing
 * order.
 */
static int binarySearch(int[] A, int loIndex, int hiIndex, int value) {

   if (loIndex > hiIndex) {
         // The starting position comes after the final index,
         // so there are actually no elements in the specified
         // range.  The value does not occur in this empty list!
      return -1;
   }

   else {
         // Look at the middle position in the list.  If the
         // value occurs at that position, return that position.
         // Otherwise, search recursively in either the first
         // half or the second half of the list.
      int middle = (loIndex + hiIndex) / 2;
      if (value == A[middle])
         return middle;
      else if (value < A[middle])
         return binarySearch(A, loIndex, middle - 1, value);
      else   // value must be > A[middle]
         return binarySearch(A, middle + 1, hiIndex, value);
   }

} // end binarySearch()
```

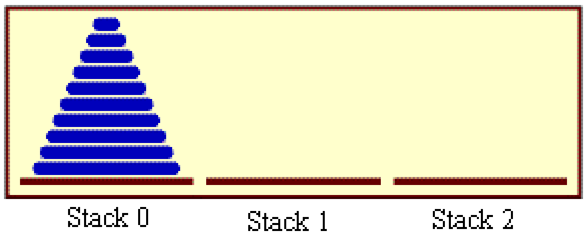
In this routine, the parameters `loIndex` and `hiIndex` specify the part of the array that is to be searched. To search an entire array, it is only necessary to call `binarySearch(A, 0, A.length - 1, value)`. In the two base cases—when there are no elements in the specified range of indices and when the value is found in the middle of the range—the subroutine can return an answer immediately, without using recursion. In the other cases, it uses a recursive call to compute the answer and returns that answer.

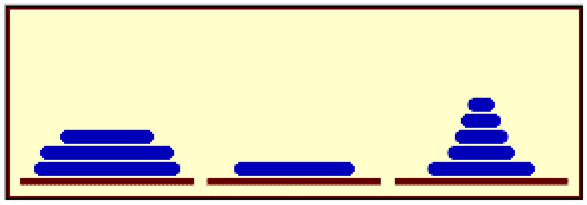
Most people find it difficult at first to convince themselves that recursion actually works. The key is to note two things that must be true for recursion to work properly: There must be one or more base cases, which can be handled without using recursion. And when recursion is applied during the solution of a problem, it must be applied to a problem that is in some sense smaller—that is, closer to the base cases—than the original problem. The idea is that if you can solve small problems and if you can reduce big problems to smaller problems, then you can solve problems of any size. Ultimately, of course, the big problems have to be reduced, possibly in many, many steps, to the very smallest problems (the base cases). Doing so might involve an immense amount of detailed bookkeeping. But the computer does that bookkeeping, not you! As a programmer, you lay out the big picture: the base cases and the reduction of big problems to smaller problems. The computer takes care of the details involved in reducing a big problem, in many steps, all the way down to base cases. Trying to think through this reduction in detail is likely to drive you crazy, and will probably make you think that recursion is hard. Whereas in fact, recursion is an elegant and powerful method that is often the simplest approach to solving a complex problem.

A common error in writing recursive subroutines is to violate one of the two rules: There must be one or more base cases, and when the subroutine is applied recursively, it must be applied to a problem that is smaller than the original problem. If these rules are violated, the result can be an ***infinite recursion***, where the subroutine keeps calling itself over and over, without ever reaching a base case. Infinite recursion is similar to an infinite loop. However, since each recursive call to the subroutine uses up some of the computer's memory, a program that is stuck in an infinite recursion will run out of memory and crash before long. (In Java, the program will crash with an exception of type `StackOverflowError`.)

9.1.2 Towers of Hanoi

Binary search can be implemented with a `while` loop, instead of with recursion, as was done in Subsection 7.4.1. Next, we turn to a problem that is easy to solve with recursion but difficult to solve without it. This is a standard example known as "The Towers of Hanoi." The problem involves a stack of various-sized disks, piled up on a base in order of decreasing size. The object is to move the stack from one base to another, subject to two rules: Only one disk can be moved at a time, and no disk can ever be placed on top of a smaller disk. There is a third base that can be used as a "spare." The starting situation for a stack of ten disks is shown in the top half of the following picture. The situation after a number of moves have been made is shown in the bottom half of the picture. These pictures are from the applet at the end of Section 9.5, which displays an animation of the step-by-step solution of the problem.




The stacks after a number of moves.

The problem is to move ten disks from Stack 0 to Stack 1, subject to certain rules. Stack 2 can be used as a spare location. Can we reduce this to smaller problems of the same type, possibly generalizing the problem a bit to make this possible? It seems natural to consider the size of the problem to be the number of disks to be moved. If there are `N` disks in Stack 0, we know that we will eventually have to move the bottom disk from Stack 0 to Stack 1. But before we can do that, according to the rules, the first `N-1` disks must be on Stack 2. Once we've moved the `N`-th disk to Stack 1, we must move the other `N-1` disks from Stack 2 to Stack 1 to complete the solution. But moving `N-1` disks is the same type of problem as moving `N` disks, except that it's a smaller version of the problem. This is exactly what we need to do recursion! The problem has to be generalized a bit, because the smaller problems involve moving disks from Stack 0 to Stack 2 or from Stack 2 to Stack 1, instead of from Stack 0 to Stack 1. In the recursive subroutine that solves the problem, the stacks that serve as the source and destination of the disks have to be specified. It's also convenient to specify the stack that is to be used as a spare, even though we could figure that out from the other two parameters. The base case is when there is only one disk to be moved. The solution in this case is trivial: Just move the disk in one step. Here is a version of the subroutine that will print out step-by-step instructions for solving the problem:

```
/**
 * Solve the problem of moving the number of disks specified
 * by the first parameter from the stack specified by the
 * second parameter to the stack specified by the third
 * parameter.  The stack specified by the fourth parameter
 * is available for use as a spare.  Stacks are specified by
 * number: 0, 1, or 2.
 */
static void TowersOfHanoi(int disks, int from, int to, int spare) {
   if (disks == 1) {
         // There is only one disk to be moved.  Just move it.
      System.out.println("Move a disk from stack number "
               + from + " to stack number " + to);
   }
   else {
         // Move all but one disk to the spare stack, then
         // move the bottom disk, then put all the other
         // disks on top of it.
      TowersOfHanoi(disks-1, from, spare, to);
      System.out.println("Move a disk from stack number "
               + from + " to stack number " + to);
      TowersOfHanoi(disks-1, spare, to, from);
   }
}
```

This subroutine just expresses the natural recursive solution. The recursion works because each recursive call involves a smaller number of disks, and the problem is trivial to solve in the base case, when there is only one disk. To solve the "top level" problem of moving `N` disks from Stack 0 to Stack 1, it should be called with the command `TowersOfHanoi(N,0,1,2)`. The subroutine is demonstrated by the sample program *TowersOfHanoi.java*.

Here, for example, is the output from the program when it is run with the number of disks set equal to 3:

```
Move a disk from stack number 0 to stack number 2
Move a disk from stack number 0 to stack number 1
Move a disk from stack number 2 to stack number 1
Move a disk from stack number 0 to stack number 2
Move a disk from stack number 1 to stack number 0
Move a disk from stack number 1 to stack number 2
Move a disk from stack number 0 to stack number 2
Move a disk from stack number 0 to stack number 1
Move a disk from stack number 2 to stack number 1
Move a disk from stack number 2 to stack number 0
Move a disk from stack number 1 to stack number 0
Move a disk from stack number 2 to stack number 1
Move a disk from stack number 0 to stack number 2
Move a disk from stack number 0 to stack number 1
Move a disk from stack number 2 to stack number 1
```

The output of this program shows you a mass of detail that you don't really want to think about! The difficulty of following the details contrasts sharply with the simplicity and elegance of the recursive solution. Of course, you really want to leave the details to the computer. It's much more interesting to watch the applet from Section 9.5, which shows the solution graphically. That applet uses the same recursive subroutine, except that the `System.out.println` statements are replaced by commands that show the image of the disk being moved from one stack to another.

There is, by the way, a story that explains the name of this problem. According to this story, on the first day of creation, a group of monks in an isolated tower near Hanoi were given a stack of 64 disks and were assigned the task of moving one disk every day, according to the rules of the Towers of Hanoi problem. On the day that they complete their task of moving all the disks from one stack to another, the universe will come to an end. But don't worry. The number of steps required to solve the problem for `N` disks is 2^N - 1, and 2^{64} - 1 days is over 50,000,000,000,000 years. We have a long way to go.

(In the terminology of Section 8.6, the Towers of Hanoi algorithm has a run time that is $\Theta(2^n)$, where n is the number of disks that have to be moved. Since the exponential function 2^n grows so quickly, the Towers of Hanoi problem can be solved in practice only for a small number of disks.)

* * *

By the way, in addtion to the graphical Towers of Hanoi applet at the end of this chapter, there are two other end-of-chapter applets in the on-line version of this text that use recursion. One is a maze-solving applet from the end of Section 11.5, and the other is a pentominos applet from the end of Section 10.5.

The Maze applet first builds a random maze. It then tries to solve the maze by finding a path through the maze from the upper left corner to the lower right corner. This problem is actually very similar to a "blob-counting" problem that is considered later in this section. The recursive maze-solving routine starts from a given square, and it visits each neighboring square and calls itself recursively from there. The recursion ends if the routine finds itself at the lower right corner of the maze.

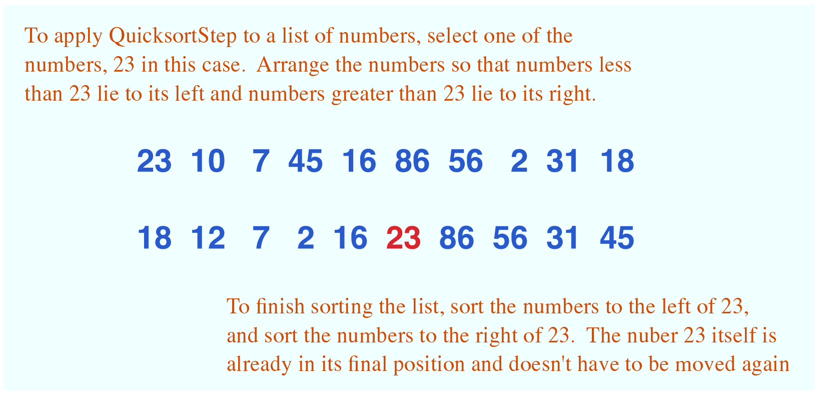
The Pentominos applet is an implementation of a classic puzzle. A pentomino is a connected figure made up of five equal-sized squares. There are exactly twelve figures that can be made in this way, not counting all the possible rotations and reflections of the basic figures. The problem is to place the twelve pentominos on an 8-by-8 board in which four of the squares have already been marked as filled. The recursive solution looks at a board that has already been partially filled with pentominos. The subroutine looks at each remaining piece in turn. It tries to place that piece in the next available place on the board. If the piece fits, it calls itself recursively to try to fill in the rest of the solution. If that fails, then the subroutine goes on to the next piece. A generalized version of the pentominos applet with many more features can be found at http://math.hws.edu/xJava/PentominosSolver/.

The Maze applet and the Pentominos applet are fun to watch, and they give nice visual representations of recursion.

9.1.3 A Recursive Sorting Algorithm

Turning next to an application that is perhaps more practical, we'll look at a recursive algorithm for sorting an array. The selection sort and insertion sort algorithms, which were covered in Section 7.4, are fairly simple, but they are rather slow when applied to large arrays. Faster sorting algorithms are available. One of these is ***Quicksort***, a recursive algorithm which turns out to be the fastest sorting algorithm in most situations.

The Quicksort algorithm is based on a simple but clever idea: Given a list of items, select any item from the list. This item is called the ***pivot***. (In practice, I'll just use the first item in the list.) Move all the items that are smaller than the pivot to the beginning of the list, and move all the items that are larger than the pivot to the end of the list. Now, put the pivot between the two groups of items. This puts the pivot in the position that it will occupy in the final, completely sorted array. It will not have to be moved again. We'll refer to this procedure as QuicksortStep.


To apply QuicksortStep to a list of numbers, select one of the numbers, 23 in this case. Arrange the numbers so that numbers less than 23 lie to its left and numbers greater than 23 lie to its right.
23 10 7 45 16 86 56 2 31 18
18 12 7 2 16 23 86 56 31 45
To finish sorting the list, sort the numbers to the left of 23, and sort the numbers to the right of 23. The nuber 23 itself is already in its final position and doesn't have to be moved again

QuicksortStep is not recursive. It is used as a subroutine by Quicksort. The speed of Quicksort depends on having a fast implementation of QuicksortStep. Since it's not the main point of this discussion, I present one without much comment.

```
/**
 * Apply QuicksortStep to the list of items in locations lo through hi
 * in the array A.  The value returned by this routine is the final
 * position of the pivot item in the array.
 */
 static int quicksortStep(int[] A, int lo, int hi) {

    int pivot = A[lo];  // Get the pivot value.

    // The numbers hi and lo mark the endpoints of a range
    // of numbers that have not yet been tested.  Decrease hi
    // and increase lo until they become equal, moving numbers
    // bigger than pivot so that they lie above hi and moving
    // numbers less than the pivot so that they lie below lo.
    // When we begin, A[lo] is an available space, since it used
    // to hold the pivot.

    while (hi > lo) {

       while (hi > lo && A[hi] > pivot) {
             // Move hi down past numbers greater than pivot.
             // These numbers do not have to be moved.
          hi--;
       }

       if (hi == lo)
          break;
```

```
            // The number A[hi] is less than pivot.  Move it into
            // the available space at A[lo], leaving an available
            // space at A[hi].

            A[lo] = A[hi];
            lo++;

            while (hi > lo && A[lo] < pivot) {
                  // Move lo up past numbers less than pivot.
                  // These numbers do not have to be moved.
               lo++;
            }

            if (hi == lo)
               break;

            // The number A[lo] is greater than pivot.  Move it into
            // the available space at A[hi], leaving an available
            // space at A[lo].

            A[hi] = A[lo];
            hi--;

         } // end while

         // At this point, lo has become equal to hi, and there is
         // an available space at that position.  This position lies
         // between numbers less than pivot and numbers greater than
         // pivot.  Put pivot in this space and return its location.

         A[lo] = pivot;
         return lo;

      }  // end QuicksortStep
```

With this subroutine in hand, Quicksort is easy. The Quicksort algorithm for sorting a list consists of applying QuicksortStep to the list, then applying Quicksort recursively to the items that lie to the left of the new position of the pivot and to the items that lie to the right of that position. Of course, we need base cases. If the list has only one item, or no items, then the list is already as sorted as it can ever be, so Quicksort doesn't have to do anything in these cases.

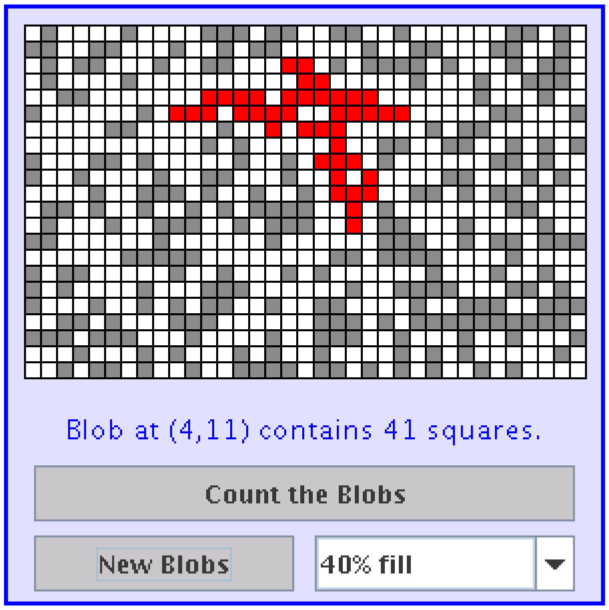
```
/**
 * Apply quicksort to put the array elements between
 * position lo and position hi into increasing order.
 */
static void quicksort(int[] A, int lo, int hi) {
   if (hi <= lo) {
         // The list has length one or zero.  Nothing needs
         // to be done, so just return from the subroutine.
      return;
   }
   else {
         // Apply quicksortStep and get the new pivot position.
         // Then apply quicksort to sort the items that
         // precede the pivot and the items that follow it.
      int pivotPosition = quicksortStep(A, lo, hi);
      quicksort(A, lo, pivotPosition - 1);
      quicksort(A, pivotPosition + 1, hi);
      }
   }
```

As usual, we had to generalize the problem. The original problem was to sort an array, but the recursive algorithm is set up to sort a specified part of an array. To sort an entire array, `A`, using the `quickSort()` subroutine, you would call `quicksort(A, 0, A.length - 1)`.

Quicksort is an interesting example from the point of view of the analysis of algorithms (Section 8.6), because its average case run time differs greatly from its worst case run time. Here is a very informal analysis, starting with the average case: Note that an application of quicksortStep divides a problem into two sub-problems. On the average, the subproblems will be of approximately the same size. A problem of size n is divided into two problems that are roughly of size n/2; these are then divided into four problems that are roughly of size n/4; and so on. Since the problem size is divided by 2 on each level, there will be approximately log(n) levels of subdivision. The amount of processing on each level is proportional to n. (On the top level, each element in the array is looked at and possibly moved. On the second level, where there are two subproblems, every element but one in the array is part of one of those two subproblems and must be looked at and possibly moved, so there is a total of about n steps in both subproblems combined. Similarly, on the third level, there are four subproblems and a total of about n steps in all four subproblems combined on that level. . . .) With a total of n steps on each level and approximately log(n) levels in the average case, the average case run time for Quicksort is $\Theta(n*\log(n))$. This analysis assumes that quicksortStep divides a problem into two approximately equal parts. However, in the worst case, each application of quicksortStep divides a problem of size n into a problem of size 0 and a problem of size n-1. This happens when the pivot element ends up at the beginning or end of the array. In this worst case, there are n levels of subproblems, and the worst-case run time is $\Theta(n^2)$. The worst case is very rare—it depends on the items in the array being arranged in a very special way, so the average performance of Quicksort can be very good even though it is not so good in certain rare cases. There are sorting algorithms that have both an average case and a worst case run time of $\Theta(n*\log(n))$. One example is MergeSort, which you can look up if you are interested.

9.1.4 Blob Counting

The program *Blobs.java* displays a grid of small white and gray squares. The gray squares are considered to be “filled” and the white squares are “empty.” For the purposes of this example, we define a “blob” to consist of a filled square and all the filled squares that can be reached from it by moving up, down, left, and right through other filled squares. If the user clicks on any filled square in the program, the computer will count the squares in the blob that contains the clicked square, and it will change the color of those squares to red. The program has several controls. There is a “New Blobs” button; clicking this button will create a new random pattern in the grid. A pop-up menu specifies the approximate percentage of squares that will be filled in the new pattern. The more filled squares, the larger the blobs. And a button labeled “Count the Blobs” will tell you how many different blobs there are in the pattern. You can try an applet version of the program in the on-line version of the book. Here is a picture of the program after the user has clicked one of the filled squares:



```
Blob at (4,11) contains 41 squares.
40% fill
```

Recursion is used in this program to count the number of squares in a blob. Without recursion, this would be a very difficult thing to implement. Recursion makes it relatively easy, but it still requires a new technique, which is also useful in a number of other applications.

The data for the grid of squares is stored in a two dimensional array of boolean values,

```
boolean[][]  filled;
```

The value of `filled[r][c]` is true if the square in row `r` and in column `c` of the grid is filled. The number of rows in the grid is stored in an instance variable named `rows`, and the number of columns is stored in `columns`. The program uses a recursive instance method named `getBlobSize()` to count the number of squares in the blob that contains the square in a given row `r` and column `c`. If there is no filled square at position `(r,c)`, then the answer is zero. Otherwise, `getBlobSize()` has to count all the filled squares that can be reached from the square at position `(r,c)`. The idea is to use `getBlobSize()` recursively to get the number of filled squares that can be reached from each of the neighboring positions, `(r+1,c)`, `(r-1,c)`, `(r,c+1)`, and `(r,c-1)`. Add up these numbers, and add one to count the square at `(r,c)` itself, and you get the total number of filled squares that can be reached from `(r,c)`. Here is an implementation of this algorithm, as stated. Unfortunately, it has a serious flaw: It leads to an infinite recursion!

```
int getBlobSize(int r, int c) {  // BUGGY, INCORRECT VERSION!!
      // This INCORRECT method tries to count all the filled
      // squares that can be reached from position (r,c) in the grid.
   if (r < 0 || r >= rows || c < 0 || c >= columns) {
         // This position is not in the grid, so there is
         // no blob at this position.  Return a blob size of zero.
      return 0;
   }
   if (filled[r][c] == false) {
        // This square is not part of a blob, so return zero.
      return 0;
   }
   int size = 1;  // Count the square at this position, then count the
```

```
                  //   the blobs that are connected to this square
                  //   horizontally or vertically.
      size += getBlobSize(r-1,c);
      size += getBlobSize(r+1,c);
      size += getBlobSize(r,c-1);
      size += getBlobSize(r,c+1);
      return size;
   }  // end INCORRECT getBlobSize()
```

Unfortunately, this routine will count the same square more than once. In fact, it will try to count each square infinitely often! Think of yourself standing at position (r,c) and trying to follow these instructions. The first instruction tells you to move up one row. You do that, and then you apply the same procedure. As one of the steps in that procedure, you have to move **down** one row and apply the same procedure yet again. But that puts you back at position (r,c)! From there, you move up one row, and from there you move down one row.... Back and forth forever! We have to make sure that a square is only counted and processed once, so we don't end up going around in circles. The solution is to leave a trail of breadcrumbs—or on the computer a trail of boolean values—to mark the squares that you've already visited. Once a square is marked as visited, it won't be processed again. The remaining, unvisited squares are reduced in number, so definite progress has been made in reducing the size of the problem. Infinite recursion is avoided!

A second boolean array, visited[r][c], is used to keep track of which squares have already been visited and processed. It is assumed that all the values in this array are set to false before getBlobSize() is called. As getBlobSize() encounters unvisited squares, it marks them as visited by setting the corresponding entry in the visited array to true. When getBlobSize() encounters a square that is already visited, it doesn't count it or process it further. The technique of "marking" items as they are encountered is one that used over and over in the programming of recursive algorithms. Here is the corrected version of getBlobSize(), with changes shown in italic:

```
   /**
    * Counts the squares in the blob at position (r,c) in the
    * grid.  Squares are only counted if they are filled and
    * unvisited.  If this routine is called for a position that
    * has been visited, the return value will be zero.
    */
   int getBlobSize(int r, int c) {
      if (r < 0 || r >= rows || c < 0 || c >= columns) {
            // This position is not in the grid, so there is
            // no blob at this position.  Return a blob size of zero.
         return 0;
      }
      if (filled[r][c] == false || visited[r][c] == true) {
           // This square is not part of a blob, or else it has
           // already been counted, so return zero.
         return 0;
      }
      visited[r][c] = true;   // Mark the square as visited so that
                              //    we won't count it again during the
                              //    following recursive calls.
      int size = 1;  // Count the square at this position, then count the
                  //   the blobs that are connected to this square
```

```
                    //   horizontally or vertically.
      size += getBlobSize(r-1,c);
      size += getBlobSize(r+1,c);
      size += getBlobSize(r,c-1);
      size += getBlobSize(r,c+1);
      return size;
   }  // end getBlobSize()
```

In the program, this method is used to determine the size of a blob when the user clicks on a square. After **getBlobSize()** has performed its task, all the squares in the blob are still marked as visited. The **paintComponent()** method draws visited squares in red, which makes the blob visible. The **getBlobSize()** method is also used for counting blobs. This is done by the following method, which includes comments to explain how it works:

```
/**
 * When the user clicks the "Count the Blobs" button, find the
 * number of blobs in the grid and report the number in the
 * message label.
 */
void countBlobs() {

   int count = 0; // Number of blobs.

   /* First clear out the visited array. The getBlobSize() method
      will mark every filled square that it finds by setting the
      corresponding element of the array to true.  Once a square
      has been marked as visited, it will stay marked until all the
      blobs have been counted.  This will prevent the same blob from
      being counted more than once. */

   for (int r = 0; r < rows; r++)
      for (int c = 0; c < columns; c++)
         visited[r][c] = false;

   /* For each position in the grid, call getBlobSize() to get the
      size of the blob at that position.  If the size is not zero,
      count a blob.  Note that if we come to a position that was part
      of a previously counted blob, getBlobSize() will return 0 and
      the blob will not be counted again. */

   for (int r = 0; r < rows; r++)
      for (int c = 0; c < columns; c++) {
         if (getBlobSize(r,c) > 0)
            count++;
      }

   repaint();  // Note that all the filled squares will be red,
               //   since they have all now been visited.

   message.setText("The number of blobs is " + count);

} // end countBlobs()
```

9.2 Linked Data Structures

Every useful object contains instance variables. When the type of an instance variable is given by a class or interface name, the variable can hold a reference to another object. Such a reference is also called a pointer, and we say that the variable ***points to*** the object. (Of course, any variable that can contain a reference to an object can also contain the special value `null`, which points to nowhere.) When one object contains an instance variable that points to another object, we think of the objects as being "linked" by the pointer. Data structures of great complexity can be constructed by linking objects together.

9.2.1 Recursive Linking

Something interesting happens when an object contains an instance variable that can refer to another object of the same type. In that case, the definition of the object's class is recursive. Such recursion arises naturally in many cases. For example, consider a class designed to represent employees at a company. Suppose that every employee except the boss has a supervisor, who is another employee of the company. Then the *Employee* class would naturally contain an instance variable of type *Employee* that points to the employee's supervisor:

```
/**
 * An object of type Employee holds data about one employee.
 */
public class Employee {

   String name;          // Name of the employee.

   Employee supervisor;  // The employee's supervisor.

      .
      .  // (Other instance variables and methods.)
      .

} // end class Employee
```

If `emp` is a variable of type *Employee*, then `emp.supervisor` is another variable of type *Employee*. If `emp` refers to the boss, then the value of `emp.supervisor` should be `null` to indicate the fact that the boss has no supervisor. If we wanted to print out the name of the employee's supervisor, for example, we could use the following Java statement:

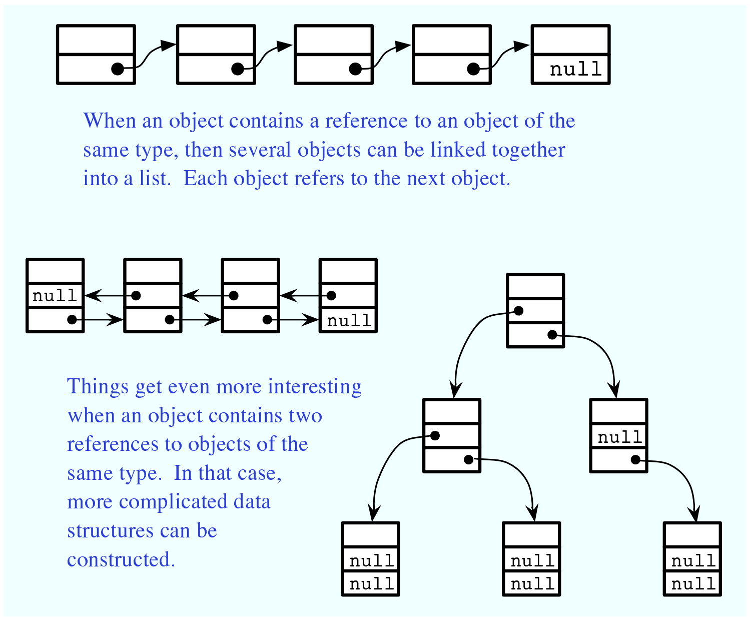
```
if ( emp.supervisor == null) {
   System.out.println( emp.name + " is the boss and has no supervisor!" );
}
else {
   System.out.print( "The supervisor of " + emp.name + " is " );
   System.out.println( emp.supervisor.name );
}
```

Now, suppose that we want to know how many levels of supervisors there are between a given employee and the boss. We just have to follow the chain of command through a series of `supervisor` links, and count how many steps it takes to get to the boss:

```
if ( emp.supervisor == null ) {
   System.out.println( emp.name + " is the boss!" );
}
else {
      Employee runner;  // For "running" up the chain of command.
      runner = emp.supervisor;
      if ( runner.supervisor == null) {
         System.out.println( emp.name  + " reports directly to the boss." );
      }
      else {
         int count = 0;
         while ( runner.supervisor != null ) {
            count++;  // Count the supervisor on this level.
            runner = runner.supervisor; // Move up to the next level.
         }
         System.out.println( "There are " + count
                                 + " supervisors between " + emp.name
                                 + " and the boss." );
      }
   }
```

As the `while` loop is executed, `runner` points in turn to the original employee, `emp`, then to `emp's` supervisor, then to the supervisor of `emp's` supervisor, and so on. The `count` variable is incremented each time `runner` "visits" a new employee. The loop ends when `runner.supervisor` is `null`, which indicates that `runner` has reached the boss. At that point, `count` has counted the number of steps between `emp` and the boss.

In this example, the `supervisor` variable is quite natural and useful. In fact, data structures that are built by linking objects together are so useful that they are a major topic of study in computer science. We'll be looking at a few typical examples. In this section and the next, we'll be looking at ***linked lists***. A linked list consists of a chain of objects of the same type, linked together by pointers from one object to the next. This is much like the chain of supervisors between `emp` and the boss in the above example. It's also possible to have more complex situations, in which one object can contain links to several other objects. We'll look at an example of this in Section 9.4.


null
When an object contains a reference to an object of the same type, then several objects can be linked together into a list. Each object refers to the next object.
null
null
Things get even more interesting when an object contains two references to objects of the same type. In that case, more complicated data structures can be constructed.
null
null
null
null
null
null
null

9.2.2 Linked Lists

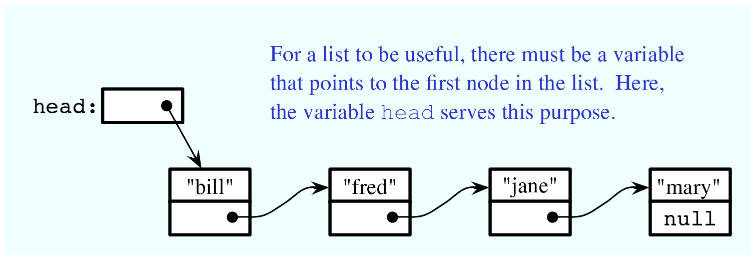
For most of the examples in the rest of this section, linked lists will be constructed out of objects belonging to the class *Node* which is defined as follows:

```
class Node {
   String item;
   Node next;
}
```

The term ***node*** is often used to refer to one of the objects in a linked data structure. Objects of type *Node* can be chained together as shown in the top part of the above picture. Each node holds a *String* and a pointer to the next node in the list (if any). The last node in such a list can always be identified by the fact that the instance variable `next` in the last node holds the value `null` instead of a pointer to another node. The purpose of the chain of nodes is to represent a list of strings. The first string in the list is stored in the first node, the second string is stored in the second node, and so on. The pointers and the node objects are used to build the structure, but the data that we are interested in representing is the list of strings. Of course, we could just as easily represent a list of integers or a list of *JButtons* or a list of any other type of data by changing the type of the `item` that is stored in each node.

Although the *Nodes* in this example are very simple, we can use them to illustrate the common operations on linked lists. Typical operations include deleting nodes from the list, inserting new nodes into the list, and searching for a specified *String* among the `items` in the list. We will look at subroutines to perform all of these operations, among others.

For a linked list to be used in a program, that program needs a variable that refers to the first node in the list. It only needs a pointer to the first node since all the other nodes in the list can be accessed by starting at the first node and following links along the list from one node to the next. In my examples, I will always use a variable named `head`, of type *Node*, that points to the first node in the linked list. When the list is empty, the value of `head` is null.


head:
For a list to be useful, there must be a variable that points to the first node in the list. Here, the variable head serves this purpose.
"bill"
"fred"
"jane"
"mary"
null

9.2.3 Basic Linked List Processing

It is very common to want to process all the items in a linked list in some way. The common pattern is to start at the head of the list, then move from each node to the next by by following the pointer in the node, stopping when the `null` that marks the end of the list is reached. If `head` is a variable of type *Node* that points to the first node in the list, then the general form of the code is:

```
Node runner;    // A pointer that will be used to traverse the list.
runner = head;  // Start with runner pointing to the head of the list.
while ( runner != null ) {     // Continue until null is encountered.
   process( runner.item );     // Do something with the item in the current node.
```

```
      runner = runner.next;       // Move on to the next node in the list.
   }
```

Our only access to the list is through the variable `head`, so we start by getting a copy of the value in `head` with the assignment statement `runner = head`. We need a **copy** of `head` because we are going to change the value of `runner`. We can't change the value of `head`, or we would lose our only access to the list! The variable `runner` will point to each node of the list in turn. When `runner` points to one of the nodes in the list, `runner.next` is a pointer to the next node in the list, so the assignment statement `runner = runner.next` moves the pointer along the list from each node to the next. We know that we've reached the end of the list when `runner` becomes equal to `null`. Note that our list-processing code works even for an empty list, since for an empty list the value of `head` is `null` and the body of the while loop is not executed at all. As an example, we can print all the strings in a list of *Strings* by saying:

```
Node runner = head;
while ( runner != null ) {
   System.out.println(  runner.item );
   runner = runner.next;
}
```

The `while` loop can, by the way, be rewritten as a `for` loop. Remember that even though the loop control variable in a `for` loop is often numerical, that is not a requirement. Here is a `for` loop that is equivalent to the above `while` loop:

```
for ( Node runner = head; runner != null; runner = runner.next ) {
   System.out.println( runner.item );
}
```

Similarly, we can traverse a list of integers to add up all the numbers in the list. A linked list of integers can be constructed using the class

```
public class IntNode {
   int item;       // One of the integers in the list.
   IntNode next;   // Pointer to the next node in the list.
}
```

If `head` is a variable of type *IntNode* that points to a linked list of integers, we can find the sum of the integers in the list using:

```
int sum = 0;
IntNode runner = head;
while ( runner != null ) {
   sum = sum + runner.item;   // Add current item to the sum.
   runner = runner.next;
}
System.out.println("The sum of the list items is " + sum);
```

It is also possible to use recursion to process a linked list. Recursion is rarely the natural way to process a list, since it's so easy to use a loop to traverse the list. However, understanding how to apply recursion to lists can help with understanding the recursive processing of more complex data structures. A non-empty linked list can be thought of as consisting of two parts: the ***head*** of the list, which is just the first node in the list, and the ***tail*** of the list, which consists of the remainder of the list after the head. Note that the tail is itself a linked list and that it is shorter than the original list (by one node). This is a natural setup for recursion, where the problem of processing a list can be divided into processing the head and recursively processing the tail. The base case occurs in the case of an empty list (or sometimes in the case of a list of length one). For example, here is a recursive algorithm for adding up the numbers in a linked list of integers:

```
if the list is empty then
   return 0 (since there are no numbers to be added up)
otherwise
   let listsum = the number in the head node
   let tailsum be the sum of the numbers in the tail list (recursively)
   add tailsum to listsum
   return listsum
```

One remaining question is, how do we get the tail of a non-empty linked list? If `head` is a variable that points to the head node of the list, then `head.next` is a variable that points to the second node of the list—and that node is in fact the first node of the tail. So, we can view `head.next` as a pointer to the tail of the list. One special case is when the original list consists of a single node. In that case, the tail of the list is empty, and `head.next` is `null`. Since an empty list is represented by a null pointer, `head.next` represents the tail of the list even in this special case. This allows us to write a recursive list-summing function in Java as

```
/**
 *  Compute the sum of all the integers in a linked list of integers.
 *  @param head a pointer to the first node in the linked list
 */
public static int addItemsInList( IntNode head ) {
   if ( head == null ) {
         // Base case:  The list is empty,  so the sum is zero.
      return 0;
   }
   else {
         // Recursive case:  The list is non-empty.  Find the sum of
         // the tail list, and add that to the item in the head node.
         // (Note that this case could be written simply as
         //     return head.item + addItemsInList( head.next );)
      int listsum = head.item;
      int tailsum = addItemsInList( head.next );
      listsum = listsum + tailsum;
      return listsum;
   }
}
```

I will finish by presenting a list-processing problem that is easy to solve with recursion, but quite tricky to solve without it. The problem is to print out all the strings in a linked list of strings in the **reverse** of the order in which they occur in the list. Note that when we do this, the item in the head of a list is printed out after all the items in the tail of the list. This leads to the following recursive routine. You should convince yourself that it works, and you should think about trying to do the same thing without using recursion:

```
public static void printReversed( Node head ) {
   if ( head == null ) {
         // Base case:  The list is empty, and there is nothing to print.
      return;
   }
   else {
```

```
            // Recursive case:  The list is non-empty.
         printReversed( head.next );  // Print strings in tail, in reverse order.
         System.out.println( head.item );  // Print string in head node.
      }
   }
```

* * *

In the rest of this section, we'll look at a few more advanced operations on a linked list of strings. The subroutines that we consider are instance methods in a class, *StringList*. An object of type *StringList* represents a linked list of nodes. The class has a private instance variable named `head` of type *Node* that points to the first node in the list, or is null if the list is empty. Instance methods in class *StringList* access `head` as a global variable. The source code for *StringList* is in the file *StringList.java*, and it is used in the sample program *ListDemo.java*.

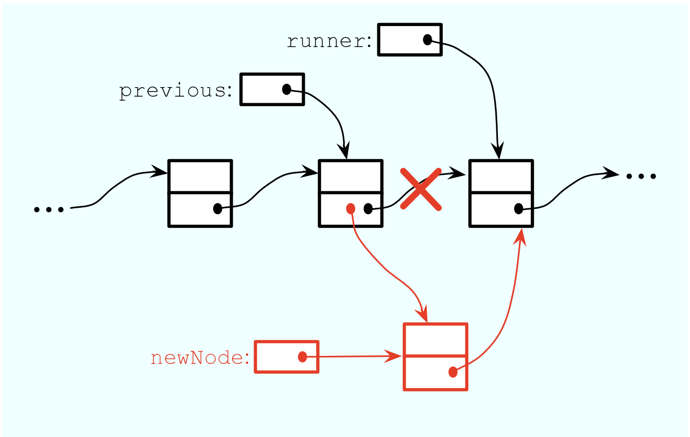
Suppose we want to know whether a specified string, `searchItem`, occurs somewhere in a list of strings. We have to compare `searchItem` to each `item` in the list. This is an example of basic list traversal and processing. However, in this case, we can stop processing if we find the item that we are looking for.

```
/**
 * Searches the list for a specified item.
 * @param searchItem the item that is to be searched for
 * @return true if searchItem is one of the items in the list or false if
 *    searchItem does not occur in the list.
 */
public boolean find(String searchItem) {

   Node runner;    // A pointer for traversing the list.

   runner = head;  // Start by looking at the head of the list.
                   //   (head is an instance variable! )

   while ( runner != null ) {
         // Go through the list looking at the string in each
         // node.  If the string is the one we are looking for,
         // return true, since the string has been found in the list.
      if ( runner.item.equals(searchItem) )
         return true;
      runner = runner.next;  // Move on to the next node.
   }

   // At this point, we have looked at all the items in the list
   // without finding searchItem.  Return false to indicate that
   // the item does not exist in the list.

   return false;

} // end find()
```

It is possible that the list is empty, that is, that the value of `head` is `null`. We should be careful that this case is handled properly. In the above code, if `head` is `null`, then the body of the `while` loop is never executed at all, so no nodes are processed and the return value is `false`. This is exactly what we want when the list is empty, since the `searchItem` can't occur in an empty list.

9.2.4 Inserting into a Linked List

The problem of inserting a new item into a linked list is more difficult, at least in the case where the item is inserted into the middle of the list. (In fact, it's probably the most difficult operation on linked data structures that you'll encounter in this chapter.) In the *StringList* class, the `items` in the nodes of the linked list are kept in increasing order. When a new item is inserted into the list, it must be inserted at the correct position according to this ordering. This means that, usually, we will have to insert the new item somewhere in the middle of the list, between two existing nodes. To do this, it's convenient to have two variables of type *Node*, which refer to the existing nodes that will lie on either side of the new node. In the following illustration, these variables are `previous` and `runner`. Another variable, `newNode`, refers to the new node. In order to do the insertion, the link from `previous` to `runner` must be "broken," and new links from `previous` to `newNode` and from `newNode` to `runner` must be added:



Inserting a new node
into the middle of a list

Once we have `previous` and `runner` pointing to the right nodes, the command "`previous.next = newNode;`" can be used to make `previous.next` point to the new node, instead of to the node indicated by `runner`. And the command "`newNode.next = runner`" will set `newNode.next` to point to the correct place. However, before we can use these commands, we need to set up `runner` and `previous` as shown in the illustration. The idea is to start at the first node of the list, and then move along the list past all the items that are less than the new item. While doing this, we have to be aware of the danger of "falling off the end of the list." That is, we can't continue if `runner` reaches the end of the list and becomes `null`. If `insertItem` is the item that is to be inserted, and if we assume that it does, in fact, belong somewhere in the middle of the list, then the following code would correctly position `previous` and `runner`:

```
Node runner, previous;
previous = head;     // Start at the beginning of the list.
runner = head.next;
while ( runner != null && runner.item.compareTo(insertItem) < 0 ) {
   previous = runner;  // "previous = previous.next" would also work
   runner = runner.next;
}
```

(This uses the `compareTo()` instance method from the *String* class to test whether the item in the node is less than the item that is being inserted. See Subsection 2.3.2.)

This is fine, except that the assumption that the new node is inserted into the middle of the list is not always valid. It might be that `insertItem` is less than the first item of the list. In that case, the new node must be inserted at the head of the list. This can be done with the instructions

```
newNode.next = head;   // Make newNode.next point to the old head.
head = newNode;        // Make newNode the new head of the list.
```

It is also possible that the list is empty. In that case, `newNode` will become the first and only node in the list. This can be accomplished simply by setting `head = newNode`. The following `insert()` method from the *StringList* class covers all of these possibilities:

```
/**
 * Insert a specified item to the list, keeping the list in order.
 * @param insertItem the item that is to be inserted.
 */
public void insert(String insertItem) {

   Node newNode;          // A Node to contain the new item.
   newNode = new Node();
   newNode.item = insertItem;  // (N.B.  newNode.next is null.)

   if ( head == null ) {
         // The new item is the first (and only) one in the list.
         // Set head to point to it.
      head = newNode;
   }
   else if ( head.item.compareTo(insertItem) >= 0 ) {
         // The new item is less than the first item in the list,
         // so it has to be inserted at the head of the list.
      newNode.next = head;
      head = newNode;
   }
   else {
         // The new item belongs somewhere after the first item
         // in the list.  Search for its proper position and insert it.
      Node runner;     // A node for traversing the list.
      Node previous;   // Always points to the node preceding runner.
      runner = head.next;   // Start by looking at the SECOND position.
      previous = head;
      while ( runner != null && runner.item.compareTo(insertItem) < 0 ) {
            // Move previous and runner along the list until runner
            // falls off the end or hits a list element that is
            // greater than or equal to insertItem.  When this
            // loop ends, runner indicates the position where
            // insertItem must be inserted.
         previous = runner;
         runner = runner.next;
      }
      newNode.next = runner;     // Insert newNode after previous.
      previous.next = newNode;
   }

}  // end insert()
```

If you were paying close attention to the above discussion, you might have noticed that there is one special case which is not mentioned. What happens if the new node has to be inserted at the **end** of the list? This will happen if all the items in the list are less than the new item. In fact, this case is already handled correctly by the subroutine, in the last part of the `if` statement. If `insertItem` is greater than all the items in the list, then the `while` loop will end when `runner` has traversed the entire list and become `null`. However, when that happens, `previous` will be left pointing to the last node in the list. Setting `previous.next = newNode` adds `newNode` onto the end of the list. Since `runner` is `null`, the command `newNode.next = runner` sets `newNode.next` to `null`, which is the correct value that is needed to mark the end of the list.

9.2.5 Deleting from a Linked List

The delete operation is similar to insert, although a little simpler. There are still special cases to consider. When the first node in the list is to be deleted, then the value of `head` has to be changed to point to what was previously the second node in the list. Since `head.next` refers to the second node in the list, this can be done by setting `head = head.next`. (Once again, you should check that this works when `head.next` is `null`, that is, when there is no second node in the list. In that case, the list becomes empty.)

If the node that is being deleted is in the middle of the list, then we can set up `previous` and `runner` with `runner` pointing to the node that is to be deleted and with `previous` pointing to the node that precedes that node in the list. Once that is done, the command "`previous.next = runner.next`;" will delete the node. The deleted node will be garbage collected. I encourage you to draw a picture for yourself to illustrate this operation. Here is the complete code for the `delete()` method:

```
/**
 * Delete a specified item from the list, if that item is present.
 * If multiple copies of the item are present in the list, only
 * the one that comes first in the list is deleted.
 * @param deleteItem the item to be deleted
 * @return true if the item was found and deleted, or false if the item
 *    was not in the list.
 */
public boolean delete(String deleteItem) {

   if ( head == null ) {
         // The list is empty, so it certainly doesn't contain deleteString.
      return false;
   }
   else if ( head.item.equals(deleteItem) ) {
          // The string is the first item of the list.  Remove it.
      head = head.next;
      return true;
   }
   else {
         // The string, if it occurs at all, is somewhere beyond the
         // first element of the list.  Search the list.
      Node runner;     // A node for traversing the list.
      Node previous;   // Always points to the node preceding runner.
      runner = head.next;   // Start by looking at the SECOND list node.
      previous = head;
```

```
         while ( runner != null && runner.item.compareTo(deleteItem) < 0 ) {
                // Move previous and runner along the list until runner
                // falls off the end or hits a list element that is
                // greater than or equal to deleteItem.  When this
                // loop ends, runner indicates the position where
                // deleteItem must be, if it is in the list.
            previous = runner;
            runner = runner.next;
         }
         if ( runner != null && runner.item.equals(deleteItem) ) {
                // Runner points to the node that is to be deleted.
                // Remove it by changing the pointer in the previous node.
            previous.next = runner.next;
            return true;
         }
         else {
                // The item does not exist in the list.
            return false;
         }
      }

   } // end delete()
```

9.3 Stacks, Queues, and ADTs

A linked list is a particular type of data structure, made up of objects linked together by pointers. In the previous section, we used a linked list to store an ordered list of *Strings*, and we implemented `insert`, `delete`, and `find` operations on that list. However, we could easily have stored the list of *Strings* in an array or *ArrayList*, instead of in a linked list. We could still have implemented the same operations on the list. The implementations of these operations would have been different, but their interfaces and logical behavior would still be the same.

The term ***abstract data type***, or ***ADT***, refers to a set of possible values and a set of operations on those values, without any specification of how the values are to be represented or how the operations are to be implemented. An "ordered list of strings" can be defined as an abstract data type. Any sequence of *Strings* that is arranged in increasing order is a possible value of this data type. The operations on the data type include inserting a new string, deleting a string, and finding a string in the list. There are often several different ways to implement the same abstract data type. For example, the "ordered list of strings" ADT can be implemented as a linked list or as an array. A program that only depends on the abstract definition of the ADT can use either implementation, interchangeably. In particular, the implementation of the ADT can be changed without affecting the program as a whole. This can make the program easier to debug and maintain, so ADTs are an important tool in software engineering.

In this section, we'll look at two common abstract data types, ***stacks*** and ***queues***. Both stacks and queues are often implemented as linked lists, but that is not the only possible implementation. You should think of the rest of this section partly as a discussion of stacks and queues and partly as a case study in ADTs.

9.3.1 Stacks

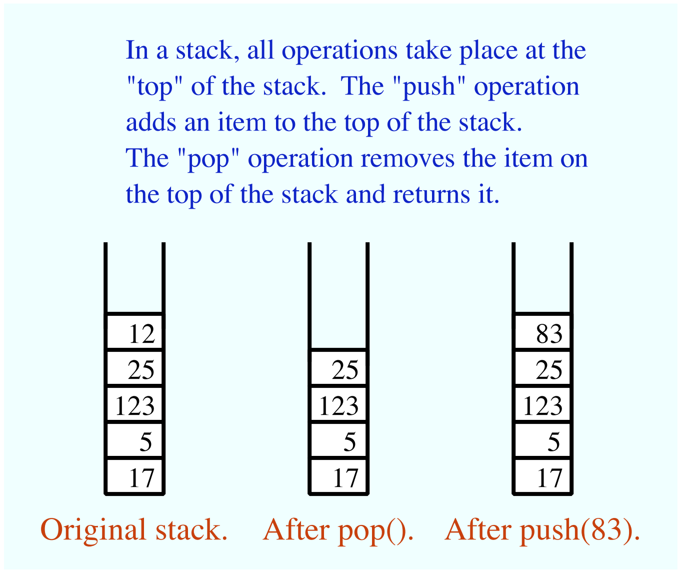
A stack consists of a sequence of items, which should be thought of as piled one on top of the other like a physical stack of boxes or cafeteria trays. Only the top item on the stack is accessible at any given time. It can be removed from the stack with an operation called ***pop***. An item lower down on the stack can only be removed after all the items on top of it have been popped off the stack. A new item can be added to the top of the stack with an operation called ***push***. We can make a stack of any type of items. If, for example, the items are values of type **int**, then the push and pop operations can be implemented as instance methods

- `void push (int newItem)` — Add newItem to top of stack.
- `int pop()` — Remove the top int from the stack and return it.

It is an error to try to pop an item from an empty stack, so it is important to be able to tell whether a stack is empty. We need another stack operation to do the test, implemented as an instance method

- `boolean isEmpty()` — Returns true if the stack is empty.

This defines a "stack of ints" as an abstract data type. This ADT can be implemented in several ways, but however it is implemented, its behavior must correspond to the abstract mental image of a stack.


In a stack, all operations take place at the "top" of the stack. The "push" operation adds an item to the top of the stack. The "pop" operation removes the item on the top of the stack and returns it.
12
25
123
5
17
25
123
5
17
83
25
123
5
17
Original stack.
After pop().
After push(83).

In the linked list implementation of a stack, the top of the stack is actually the node at the head of the list. It is easy to add and remove nodes at the front of a linked list—much easier than inserting and deleting nodes in the middle of the list. Here is a class that implements the "stack of ints" ADT using a linked list. (It uses a static nested class to represent the nodes of the linked list. If the nesting bothers you, you could replace it with a separate *Node* class.)

```
public class StackOfInts {

   /**
    * An object of type Node holds one of the items in the linked list
    * that represents the stack.
    */
```

```
      private static class Node {
         int item;
         Node next;
      }

      private Node top;  // Pointer to the Node that is at the top of
                         //   of the stack.  If top == null, then the
                         //   stack is empty.

      /**
       * Add N to the top of the stack.
       */
      public void push( int N ) {
         Node newTop;         // A Node to hold the new item.
         newTop = new Node();
         newTop.item = N;     // Store N in the new Node.
         newTop.next = top;   // The new Node points to the old top.
         top = newTop;        // The new item is now on top.
      }

      /**
       * Remove the top item from the stack, and return it.
       * Throws an IllegalStateException if the stack is empty when
       * this method is called.
       */
      public int pop() {
         if ( top == null )
            throw new IllegalStateException("Can't pop from an empty stack.");
         int topItem = top.item;  // The item that is being popped.
         top = top.next;          // The previous second item is now on top.
         return topItem;
      }

      /**
       * Returns true if the stack is empty.  Returns false
       * if there are one or more items on the stack.
       */
      public boolean isEmpty() {
         return (top == null);
      }

   } // end class StackOfInts
```

You should make sure that you understand how the `push` and `pop` operations operate on the linked list. Drawing some pictures might help. Note that the linked list is part of the `private` implementation of the *StackOfInts* class. A program that uses this class doesn't even need to know that a linked list is being used.

Now, it's pretty easy to implement a stack as an array instead of as a linked list. Since the number of items on the stack varies with time, a counter is needed to keep track of how many spaces in the array are actually in use. If this counter is called `top`, then the items on the stack are stored in positions `0`, `1`, ..., `top-1` in the array. The item in position `0` is on the bottom of the stack, and the item in position `top-1` is on the top of the stack. Pushing an item onto the stack is easy: Put the item in position `top` and add `1` to the value of `top`. If we don't want to put a limit on the number of items that the stack can hold, we can use the dynamic array techniques from Subsection 7.3.2. Note that the typical picture of the array would show the stack "upside down", with the top of the stack at the bottom of the array. This doesn't matter. The array is just an implementation of the abstract idea of a stack, and as long as the stack operations work the way they are supposed to, we are OK. Here is a second implementation of the *StackOfInts* class, using a dynamic array:

```
public class StackOfInts {  // (alternate version, using an array)

   private int[] items = new int[10];  // Holds the items on the stack.

   private int top = 0;  // The number of items currently on the stack.

   /**
    * Add N to the top of the stack.
    */
   public void push( int N ) {
       if (top == items.length) {
               // The array is full, so make a new, larger array and
               // copy the current stack items into it.
           int[] newArray = new int[ 2*items.length ];
           System.arraycopy(items, 0, newArray, 0, items.length);
           items = newArray;
       }
       items[top] = N;  // Put N in next available spot.
       top++;           // Number of items goes up by one.
   }

   /**
    * Remove the top item from the stack, and return it.
    * Throws an IllegalStateException if the stack is empty when
    * this method is called.
    */
   public int pop() {
       if ( top == 0 )
          throw new IllegalStateException("Can't pop from an empty stack.");
       int topItem = items[top - 1]  // Top item in the stack.
       top--;    // Number of items on the stack goes down by one.
       return topItem;
   }

   /**
    * Returns true if the stack is empty.  Returns false
    * if there are one or more items on the stack.
    */
   public boolean isEmpty() {
      return (top == 0);
   }

} // end class StackOfInts
```

Once again, the implementation of the stack (as an array) is private to the class. The two versions of the *StackOfInts* class can be used interchangeably, since their public interfaces are identical.

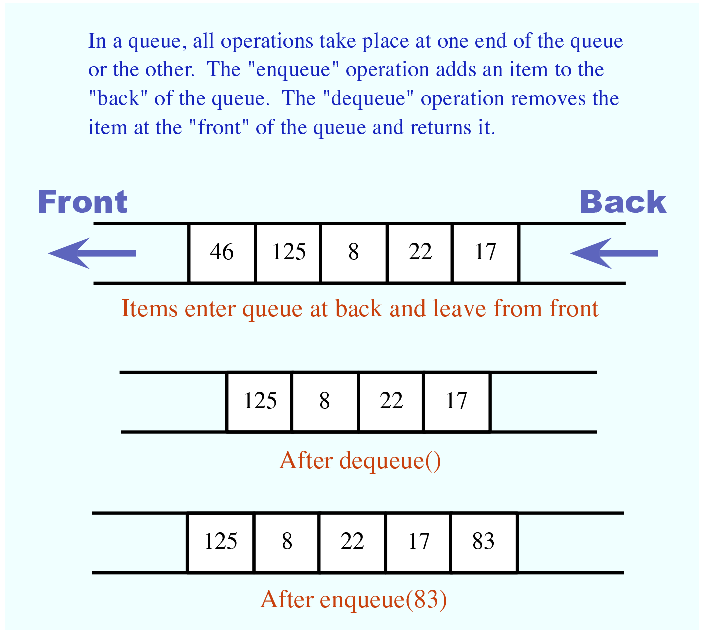
* * *

It's interesting to look at the run time analysis of stack operations. (See Section 8.6). We can measure the size of the problem by the number of items that are on the stack. For the linked list implementation of a stack, the worst case run time both for the `push` and for the `pop` operation is $\Theta(1)$. This just means that the run time is less than some constant, independent of the number of items on the stack. This is easy to see if you look at the code. The operations are implemented with a few simple assignment statements, and the number of items on the stack has no effect.

For the array implementation, on the other hand, a special case occurs in the `push` operation when the array is full. In that case, a new array is created and all the stack items are copied into the new array. This takes an amount of time that is proportional to the number of items on the stack. So, although the run time for `push` is usually $\Theta(1)$, the worst case run time is $\Theta(n)$, where n is the number of items on the stack.

9.3.2 Queues

Queues are similar to stacks in that a queue consists of a sequence of items, and there are restrictions about how items can be added to and removed from the list. However, a queue has two ends, called the front and the back of the queue. Items are always added to the queue at the back and removed from the queue at the front. The operations of adding and removing items are called ***enqueue*** and ***dequeue***. An item that is added to the back of the queue will remain on the queue until all the items in front of it have been removed. This should sound familiar. A queue is like a "line" or "queue" of customers waiting for service. Customers are serviced in the order in which they arrive on the queue.


In a queue, all operations take place at one end of the queue or the other. The "enqueue" operation adds an item to the "back" of the queue. The "dequeue" operation removes the item at the "front" of the queue and returns it.
Front
Back
46
125
8
22
17
Items enter queue at back and leave from front
125
8
22
17
After dequeue()
125
8
22
17
83
After enqueue(83)

A queue can hold items of any type. For a queue of `ints`, the enqueue and dequeue operations can be implemented as instance methods in a "*QueueOfInts*" class. We also need an instance method for checking whether the queue is empty:

- `void enqueue(int N)` — Add N to the back of the queue.
- `int dequeue()` — Remove the item at the front and return it.
- `boolean isEmpty()` — Return true if the queue is empty.

A queue can be implemented as a linked list or as an array. An efficient array implementation is a little trickier than the array implementation of a stack, so I won't give it here. In the linked list implementation, the first item of the list is at the front of the queue. Dequeueing an item from the front of the queue is just like popping an item off a stack. The back of the queue is at the end of the list. Enqueueing an item involves setting a pointer in the last node of the current list to point to a new node that contains the item. To do this, we'll need a command like "`tail.next = newNode;`", where `tail` is a pointer to the last node in the list. If `head` is a pointer to the first node of the list, it would always be possible to get a pointer to the last node of the list by saying:

```
Node tail;    // This will point to the last node in the list.
tail = head;  // Start at the first node.
while (tail.next != null) {
   tail = tail.next;  // Move to next node.
}
// At this point, tail.next is null, so tail points to
// the last node in the list.
```

However, it would be very inefficient to do this over and over every time an item is enqueued. For the sake of efficiency, we'll keep a pointer to the last node in an instance variable. This complicates the class somewhat; we have to be careful to update the value of this variable whenever a new node is added to the end of the list. Given all this, writing the *QueueOfInts* class is not all that difficult:

```
public class QueueOfInts {

   /**
    * An object of type Node holds one of the items
    * in the linked list that represents the queue.
    */
   private static class Node {
      int item;
      Node next;
   }

   private Node head = null;  // Points to first Node in the queue.
                              // The queue is empty when head is null.

   private Node tail = null;  // Points to last Node in the queue.

   /**
    * Add N to the back of the queue.
    */
   public void enqueue( int N ) {
      Node newTail = new Node();  // A Node to hold the new item.
      newTail.item = N;
      if (head == null) {
            // The queue was empty.  The new Node becomes
            // the only node in the list.  Since it is both
            // the first and last node, both head and tail
            // point to it.
         head = newTail;
         tail = newTail;
      }
      else {
            // The new node becomes the new tail of the list.
            // (The head of the list is unaffected.)
         tail.next = newTail;
         tail = newTail;
      }
   }

   /**
    * Remove and return the front item in the queue.
    * Throws an IllegalStateException if the queue is empty.
    */
   public int dequeue() {
      if ( head == null)
          throw new IllegalStateException("Can't dequeue from an empty queue.");
      int firstItem = head.item;
      head = head.next;  // The previous second item is now first.
      if (head == null) {
            // The queue has become empty.  The Node that was
            // deleted was the tail as well as the head of the
            // list, so now there is no tail.  (Actually, the
            // class would work fine without this step.)
         tail = null;
      }
      return firstItem;
   }

   /**
    * Return true if the queue is empty.
    */
   boolean isEmpty() {
      return (head == null);
   }

} // end class QueueOfInts
```

Queues are typically used in a computer (as in real life) when only one item can be processed at a time, but several items can be waiting for processing. For example:

- In a Java program that has multiple threads, the threads that want processing time on the CPU are kept in a queue. When a new thread is started, it is added to the back of the queue. A thread is removed from the front of the queue, given some processing time, and then—if it has not terminated—is sent to the back of the queue to wait for another turn.
- Events such as keystrokes and mouse clicks are stored in a queue called the "event queue". A program removes events from the event queue and processes them. It's possible for several more events to occur while one event is being processed, but since the events are stored in a queue, they will always be processed in the order in which they occurred.
- A web server is a progam that receives requests from web browsers for "pages." It is easy for new requests to arrive while the web server is still fulfilling a previous request. Requests that arrive while the web server is busy are placed into a queue to await processing. Using a queue ensures that requests will be processed in the order in which they were received.

Queues are said to implement a ***FIFO*** policy: First In, First Out. Or, as it is more commonly expressed, first come, first served. Stacks, on the other hand implement a ***LIFO*** policy: Last In, First Out. The item that comes out of the stack is the last one that was put in. Just like queues, stacks can be used to hold items that are waiting for processing (although in applications where queues are typically used, a stack would be considered "unfair").

* * *

To get a better handle on the difference between stacks and queues, consider the sample program *DepthBreadth.java*. I suggest that you run the program or try the applet version that can be found in the on-line version of this section. The program shows a grid of squares. Initially, all the squares are white. When you click on a white square, the program will gradually mark all the squares in the grid, starting from the one where you click. To understand how the program does this, think of yourself in the place of the program. When the user clicks a square, you are handed an index card. The location of the square—its row and column—is written on the card. You put the card in a pile, which then contains just that one card. Then, you repeat the following: If the pile is empty, you are done. Otherwise, take an index card from the pile. The index card specifies a square. Look at each horizontal and vertical neighbor of that square. If the neighbor has not already been encountered, write its location on a new index card and put the card in the pile.

While a square is in the pile, waiting to be processed, it is colored red; that is, red squares have been encountered but not yet processed. When a square is taken from the pile and processed, its color changes to gray. Once a square has been colored gray, its color won't change again. Eventually, all the squares have been processed, and the procedure ends. In the index card analogy, the pile of cards has been emptied.

The program can use your choice of three methods: Stack, Queue, and Random. In each case, the same general procedure is used. The only difference is how the "pile of index cards" is managed. For a stack, cards are added and removed at the top of the pile. For a queue, cards are added to the bottom of the pile and removed from the top. In the random case, the card to be processed is picked at random from among all the cards in the pile. The order of processing is very different in these three cases.

You should experiment with the program to see how it all works. Try to understand how stacks and queues are being used. Try starting from one of the corner squares. While the process is going on, you can click on other white squares, and they will be added to the pile. When you do this with a stack, you should notice that the square you click is processed immediately, and all the red squares that were already waiting for processing have to wait. On the other hand, if you do this with a queue, the square that you click will wait its turn until all the squares that were already in the pile have been processed.

* * *

Queues seem very natural because they occur so often in real life, but there are times when stacks are appropriate and even essential. For example, consider what happens when a routine calls a subroutine. The first routine is suspended while the subroutine is executed, and it will continue only when the subroutine returns. Now, suppose that the subroutine calls a second subroutine, and the second subroutine calls a third, and so on. Each subroutine is suspended while the subsequent subroutines are executed. The computer has to keep track of all the subroutines that are suspended. It does this with a stack.

When a subroutine is called, an ***activation record*** is created for that subroutine. The activation record contains information relevant to the execution of the subroutine, such as its local variables and parameters. The activation record for the subroutine is placed on a stack. It will be removed from the stack and destroyed when the subroutine returns. If the subroutine calls another subroutine, the activation record of the second subroutine is pushed onto the stack, on top of the activation record of the first subroutine. The stack can continue to grow as more subroutines are called, and it shrinks as those subroutines return.

9.3.3 Postfix Expressions

As another example, stacks can be used to evaluate ***postfix expressions***. An ordinary mathematical expression such as `2+(15-12)*17` is called an ***infix expression***. In an infix expression, an operator comes in between its two operands, as in "`2 + 2`". In a postfix expression, an operator comes after its two operands, as in "`2 2 +`". The infix expression "`2+(15-12)*17`" would be written in postfix form as "`2 15 12 - 17 * +`". The "`-`" operator in this expression applies to the two operands that precede it, namely "`15`" and "`12`". The "`*`" operator applies to the two operands that precede it, namely "`15 12 -`" and "`17`". And the "`+`" operator applies to "`2`" and "`15 12 - 17 *`". These are the same computations that are done in the original infix expression.

Now, suppose that we want to process the expression "`2 15 12 - 17 * +`", from left to right and find its value. The first item we encounter is the `2`, but what can we do with it? At this point, we don't know what operator, if any, will be applied to the `2` or what the other operand might be. We have to remember the `2` for later processing. We do this by pushing it onto a stack. Moving on to the next item, we see a `15`, which is pushed onto the stack on top of the `2`. Then the `12` is added to the stack. Now, we come to the operator, "`-`". This operation applies to the two operands that preceded it in the expression. We have saved those two operands on the stack. So, to process the "`-`" operator, we pop two numbers from the stack, `12` and `15`, and compute `15 - 12` to get the answer `3`. This `3` must be remembered to be used in later processing, so we push it onto the stack, on top of the `2` that is still waiting there. The next item in the expression is a `17`, which is processed by pushing it onto the stack, on top of the `3`. To process the next item, "`*`", we pop two numbers from the stack. The numbers are `17` and the `3` that represents the value of "`15 12 -`". These numbers are multiplied, and the result, `51` is pushed onto the stack. The next item in the expression is a "`+`" operator, which is processed by popping `51` and `2` from the stack, adding them, and pushing the result, `53`, onto the stack. Finally, we've come to the end of the expression. The number on the stack is the value of the entire expression, so all we have to do is pop the answer from the stack, and we are done! The value of the expression is `53`.

Although it's easier for people to work with infix expressions, postfix expressions have some advantages. For one thing, postfix expressions don't require parentheses or precedence rules. The order in which operators are applied is determined entirely by the order in which they occur in the expression. This allows the algorithm for evaluating postfix expressions to be fairly straightforward:

```
Start with an empty stack
for each item in the expression:
    if the item is a number:
       Push the number onto the stack
    else if the item is an operator:
       Pop the operands from the stack  // Can generate an error
       Apply the operator to the operands
       Push the result onto the stack
    else
       There is an error in the expression
Pop a number from the stack  // Can generate an error
if the stack is not empty:
      There is an error in the expression
   else:
      The last number that was popped is the value of the expression
```

Errors in an expression can be detected easily. For example, in the expression "2 3 + *", there are not enough operands for the "*" operation. This will be detected in the algorithm when an attempt is made to pop the second operand for "*" from the stack, since the stack will be empty. The opposite problem occurs in "2 3 4 +". There are not enough operators for all the numbers. This will be detected when the 2 is left still sitting in the stack at the end of the algorithm.

This algorithm is demonstrated in the sample program *PostfixEval.java*. This program lets you type in postfix expressions made up of non-negative real numbers and the operators "+", "-", "*", "/", and "^". The "^" represents exponentiation. That is, "2 3 ^" is evaluated as 2^3. The program prints out a message as it processes each item in the expression. The stack class that is used in the program is defined in the file *StackOfDouble.java*. The *StackOfDouble* class is identical to the first *StackOfInts* class, given above, except that it has been modified to store values of type **double** instead of values of type **int**.

The only interesting aspect of this program is the method that implements the postfix evaluation algorithm. It is a direct implementation of the pseudocode algorithm given above:

```
/**
 *  Read one line of input and process it as a postfix expression.
 *  If the input is not a legal postfix expression, then an error
 *  message is displayed.  Otherwise, the value of the expression
 *  is displayed.  It is assumed that the first character on
 *  the input line is a non-blank.
 */
private static void readAndEvaluate() {

   StackOfDouble stack;  // For evaluating the expression.

   stack = new StackOfDouble();  // Make a new, empty stack.

   TextIO.putln();

   while (TextIO.peek() != '\n') {

      if ( Character.isDigit(TextIO.peek()) ) {
            // The next item in input is a number.  Read it and
            // save it on the stack.
         double num = TextIO.getDouble();
         stack.push(num);
         TextIO.putln("   Pushed constant " + num);
      }
      else {
            // Since the next item is not a number, the only thing
            // it can legally be is an operator.  Get the operator
            // and perform the operation.
         char op;  // The operator, which must be +, -, *, /, or ^.
         double x,y;     // The operands, from the stack, for the operation.
         double answer;  // The result, to be pushed onto the stack.
         op = TextIO.getChar();
         if (op != '+' && op != '-' && op != '*' && op != '/' && op != '^') {
               // The character is not one of the acceptable operations.
            TextIO.putln("\nIllegal operator found in input: " + op);
            return;
         }
         if (stack.isEmpty()) {
            TextIO.putln("   Stack is empty while trying to evaluate " + op);
            TextIO.putln("\nNot enough numbers in expression!");
            return;
         }
         y = stack.pop();
         if (stack.isEmpty()) {
            TextIO.putln("   Stack is empty while trying to evaluate " + op);
            TextIO.putln("\nNot enough numbers in expression!");
            return;
         }
         x = stack.pop();
         switch (op) {
         case '+':
            answer = x + y;
            break;
         case '-':
            answer = x - y;
            break;
         case '*':
            answer = x * y;
            break;
         case '/':
            answer = x / y;
            break;
         default:
            answer = Math.pow(x,y);  // (op must be '^'.)
         }
         stack.push(answer);
         TextIO.putln("   Evaluated " + op + " and pushed " + answer);
      }

      TextIO.skipBlanks();

   }  // end while

   // If we get to this point, the input has been read successfully.
   // If the expression was legal, then the value of the expression is
   // on the stack, and it is the only thing on the stack.

   if (stack.isEmpty()) {  // Impossible if the input is really non-empty.
      TextIO.putln("No expression provided.");
      return;
   }

   double value = stack.pop();  // Value of the expression.
   TextIO.putln("   Popped " + value + " at end of expression.");

   if (stack.isEmpty() == false) {
      TextIO.putln("   Stack is not empty.");
      TextIO.putln("\nNot enough operators for all the numbers!");
      return;
   }

   TextIO.putln("\nValue = " + value);
   } // end readAndEvaluate()
```

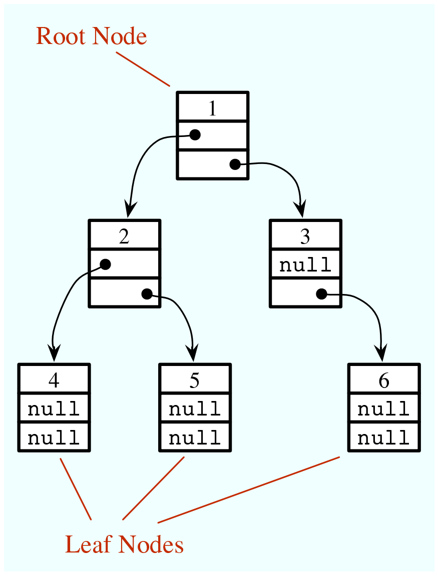
Postfix expressions are often used internally by computers. In fact, the Java virtual machine is a "stack machine" which uses the stack-based approach to expression evaluation that we have been discussing. The algorithm can easily be extended to handle variables, as well as constants. When a variable is encountered in the expression, the value of the variable is pushed onto the stack. It also works for operators with more or fewer than two operands. As many operands as are needed are popped from the stack and the result is pushed back on to the stack. For example, the ***unary minus*** operator, which is used in the expression "`-x`", has a single operand. We will continue to look at expressions and expression evaluation in the next two sections.

9.4 Binary Trees

We have seen in the two previous sections how objects can be linked into lists. When an object contains two pointers to objects of the same type, structures can be created that are much more complicated than linked lists. In this section, we'll look at one of the most basic and useful structures of this type: ***binary trees***. Each of the objects in a binary tree contains two pointers, typically called `left` and `right`. In addition to these pointers, of course, the nodes can contain other types of data. For example, a binary tree of integers could be made up of objects of the following type:

```
   class TreeNode {
      int item;        // The data in this node.
      TreeNode left;   // Pointer to the left subtree.
      TreeNode right;  // Pointer to the right subtree.
   }
```

The `left` and `right` pointers in a `TreeNode` can be `null` or can point to other objects of type `TreeNode`. A node that points to another node is said to be the ***parent*** of that node, and the node it points to is called a ***child***. In the picture below, for example, node 3 is the parent of node 6, and nodes 4 and 5 are children of node 2. Not every linked structure made up of tree nodes is a binary tree. A binary tree must have the following properties: There is exactly one node in the tree which has no parent. This node is called the ***root*** of the tree. Every other node in the tree has exactly one parent. Finally, there can be no loops in a binary tree. That is, it is not possible to follow a chain of pointers starting at some node and arriving back at the same node.



A node that has no children is called a ***leaf***. A leaf node can be recognized by the fact that both the left and right pointers in the node are `null`. In the standard picture of a binary tree, the root node is shown at the top and the leaf nodes at the bottom—which doesn't show much respect for the analogy to real trees. But at least you can see the branching, tree-like structure that gives a binary tree its name.

9.4.1 Tree Traversal

Consider any node in a binary tree. Look at that node together with all its descendents (that is, its children, the children of its children, and so on). This set of nodes forms a binary tree, which is called a ***subtree*** of the original tree. For example, in the picture, nodes 2, 4, and 5 form a subtree. This subtree is called the ***left subtree*** of the root. Similarly, nodes 3 and 6 make up the ***right subtree*** of the root. We can consider any non-empty binary tree to be made up of a root node, a left subtree, and a right subtree. Either or both of the subtrees can be empty. This is a recursive definition, matching the recursive definition of the *TreeNode* class. So it should not be a surprise that recursive subroutines are often used to process trees.

Consider the problem of counting the nodes in a binary tree. (As an exercise, you might try to come up with a non-recursive algorithm to do the counting, but you shouldn't expect to find one.) The heart of problem is keeping track of which nodes remain to be counted. It's not so easy to do this, and in fact it's not even possible without an auxiliary data structure such as a stack or queue. With recursion, however, the algorithm is almost trivial. Either the tree is empty or it consists of a root and two subtrees. If the tree is empty, the number of nodes is zero. (This is the base case of the recursion.) Otherwise, use recursion to count the nodes in each subtree. Add the results from the subtrees together, and add one to count the root. This gives the total number of nodes in the tree. Written out in Java:

```
/**
 * Count the nodes in the binary tree to which root points, and
 * return the answer.  If root is null, the answer is zero.
 */
static int countNodes( TreeNode root ) {
   if ( root == null )
```

```
         return 0;  // The tree is empty.  It contains no nodes.
      else {
         int count = 1;   // Start by counting the root.
         count += countNodes(root.left);  // Add the number of nodes
                                          //     in the left subtree.
         count += countNodes(root.right); // Add the number of nodes
                                          //    in the right subtree.
         return count;  // Return the total.
      }
   } // end countNodes()
```

Or, consider the problem of printing the items in a binary tree. If the tree is empty, there is nothing to do. If the tree is non-empty, then it consists of a root and two subtrees. Print the item in the root and use recursion to print the items in the subtrees. Here is a subroutine that prints all the items on one line of output:

```
   /**
    * Print all the items in the tree to which root points.
    * The item in the root is printed first, followed by the
    * items in the left subtree and then the items in the
    * right subtree.
    */
   static void preorderPrint( TreeNode root ) {
      if ( root != null ) {  // (Otherwise, there's nothing to print.)
         System.out.print( root.item + " " );  // Print the root item.
         preorderPrint( root.left );   // Print items in left subtree.
         preorderPrint( root.right );  // Print items in right subtree.
      }
   } // end preorderPrint()
```

This routine is called "preorderPrint" because it uses a ***preorder traversal*** of the tree. In a preorder traversal, the root node of the tree is processed first, then the left subtree is traversed, then the right subtree. In a ***postorder traversal***, the left subtree is traversed, then the right subtree, and then the root node is processed. And in an ***inorder traversal***, the left subtree is traversed first, then the root node is processed, then the right subtree is traversed. Printing subroutines that use postorder and inorder traversal differ from `preorderPrint` only in the placement of the statement that outputs the root item:

```
   /**
    * Print all the items in the tree to which root points.
    * The item in the left subtree printed first, followed
    * by the items in the right subtree and then the item
    * in the root node.
    */
   static void postorderPrint( TreeNode root ) {
      if ( root != null ) {  // (Otherwise, there's nothing to print.)
         postorderPrint( root.left );   // Print items in left subtree.
         postorderPrint( root.right );  // Print items in right subtree.
         System.out.print( root.item + " " );  // Print the root item.
      }
   } // end postorderPrint()


   /**
    * Print all the items in the tree to which root points.
```

```
 * The item in the left subtree printed first, followed
 * by the item in the root node and then the items
 * in the right subtree.
 */
static void inorderPrint( TreeNode root ) {
   if ( root != null ) {  // (Otherwise, there's nothing to print.)
      inorderPrint( root.left );   // Print items in left subtree.
      System.out.print( root.item + " " );  // Print the root item.
      inorderPrint( root.right );  // Print items in right subtree.
   }
} // end inorderPrint()
```

Each of these subroutines can be applied to the binary tree shown in the illustration at the beginning of this section. The order in which the items are printed differs in each case:

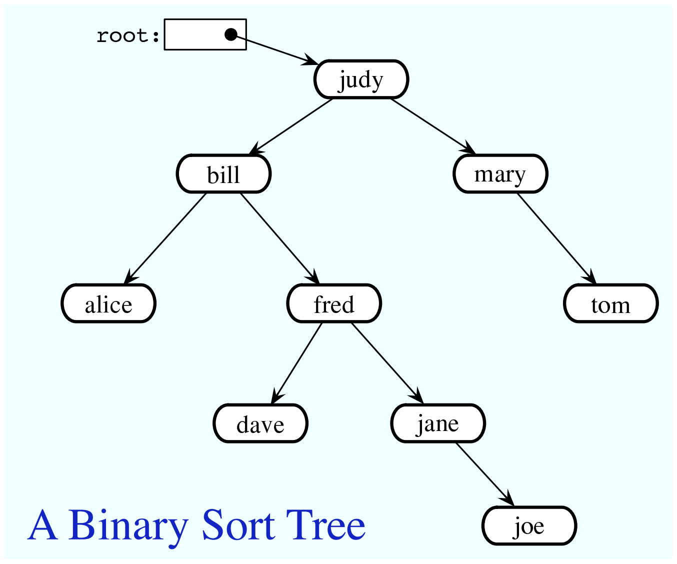
```
preorderPrint outputs:   1  2  4  5  3  6

postorderPrint outputs:  4  5  2  6  3  1

inorderPrint outputs:    4  2  5  1  3  6
```

In `preorderPrint`, for example, the item at the root of the tree, `1`, is output before anything else. But the preorder printing also applies to each of the subtrees of the root. The root item of the left subtree, `2`, is printed before the other items in that subtree, `4` and `5`. As for the right subtree of the root, `3` is output before `6`. A preorder traversal applies at all levels in the tree. The other two traversal orders can be analyzed similarly.

9.4.2 Binary Sort Trees

One of the examples in Section 9.2 was a linked list of strings, in which the strings were kept in increasing order. While a linked list works well for a small number of strings, it becomes inefficient for a large number of items. When inserting an item into the list, searching for that item's position requires looking at, on average, half the items in the list. Finding an item in the list requires a similar amount of time. If the strings are stored in a sorted array instead of in a linked list, then searching becomes more efficient because binary search can be used. However, inserting a new item into the array is still inefficient since it means moving, on average, half of the items in the array to make a space for the new item. A binary tree can be used to store an ordered list of strings, or other items, in a way that makes both searching and insertion efficient. A binary tree used in this way is called a ***binary sort tree.***

A binary sort tree is a binary tree with the following property: For every node in the tree, the item in that node is greater than every item in the left subtree of that node, and it is less than or equal to all the items in the right subtree of that node. Here for example is a binary sort tree containing items of type *String*. (In this picture, I haven't bothered to draw all the pointer variables. Non-null pointers are shown as arrows.)



Binary sort trees have this useful property: An inorder traversal of the tree will process the items in increasing order. In fact, this is really just another way of expressing the definition. For example, if an inorder traversal is used to print the items in the tree shown above, then the items will be in alphabetical order. The definition of an inorder traversal guarantees that all the items in the left subtree of "judy" are printed before "judy", and all the items in the right subtree of "judy" are printed after "judy". But the binary sort tree property guarantees that the items in the left subtree of "judy" are precisely those that precede "judy" in alphabetical order, and all the items in the right subtree follow "judy" in alphabetical order. So, we know that "judy" is output in its proper alphabetical position. But the same argument applies to the subtrees. "Bill" will be output after "alice" and before "fred" and its descendents. "Fred" will be output after "dave" and before "jane" and "joe". And so on.

Suppose that we want to search for a given item in a binary search tree. Compare that item to the root item of the tree. If they are equal, we're done. If the item we are looking for is less than the root item, then we need to search the left subtree of the root—the right subtree can be eliminated because it only contains items that are greater than or equal to the root. Similarly, if the item we are looking for is greater than the item in the root, then we only need to look in the right subtree. In either case, the same procedure can then be applied to search the subtree. Inserting a new item is similar: Start by searching the tree for the position where the new item belongs. When that position is found, create a new node and attach it to the tree at that position.

Searching and inserting are efficient operations on a binary search tree, provided that the tree is close to being ***balanced***. A binary tree is balanced if for each node, the left subtree of that node contains approximately the same number of nodes as the right subtree. In a perfectly balanced tree, the two numbers differ by at most one. Not all binary trees are balanced, but if the tree is created by inserting items in a random order, there is a high probability that the tree is approximately balanced. (If the order of insertion is not random, however, it's quite possible for the tree to be very unbalanced.) During a search of any binary sort tree, every comparison eliminates one of two subtrees from further consideration. If the tree is balanced, that means cutting the number of items still under consideration in half. This is exactly the same as the binary search algorithm, and the result, is a similarly efficient algorithm.

In terms of asymptotic analysis (Section 8.6), searching, inserting, and deleting in a binary search tree have average case run time Θ(log(n)). The problem size, n, is the number of items in the tree, and the average is taken over all the different orders in which the items could have been inserted into the tree. As long the actual insertion order is random, the actual run time can be expected to be close to the average. However, the worst case run time for binary search tree operations is Θ(n), which is much worse than Θ(log(n)). The worst case occurs for certain particular insertion orders. For example, if the items are inserted into the tree in order of increasing size, then every item that is inserted moves always to the right as it moves down the tree. The result is a "tree" that looks more like a linked list, since it consists of a linear string of nodes strung together by their `right` child pointers. Operations on such a tree have the same performance as operations on a linked list. Now, there are data structures that are similar to simple binary sort trees, except that insertion and deletion of nodes are implemented in a way that will always keep the tree balanced, or almost balanced. For these data structures, searching, inserting, and deleting have both average case and worst case run times that are Θ(log(n)). Here, however, we will look at only the simple versions of inserting and searching.

The sample program *SortTreeDemo.java* is a demonstration of binary sort trees. The program includes subroutines that implement inorder traversal, searching, and insertion. We'll look at the latter two subroutines below. The `main()` routine tests the subroutines by letting you type in strings to be inserted into the tree.

In this program, nodes in the binary tree are represented using the following static nested class, including a simple constructor that makes creating nodes easier:

```
/**
 * An object of type TreeNode represents one node in a binary tree of strings.
 */
private static class TreeNode {
   String item;      // The data in this node.
   TreeNode left;    // Pointer to left subtree.
   TreeNode right;   // Pointer to right subtree.
   TreeNode(String str) {
          // Constructor.  Make a node containing str.
      item = str;
   }
}  // end class TreeNode
```

A static member variable of type *TreeNode* points to the binary sort tree that is used by the program:

```
private static TreeNode root;  // Pointer to the root node in the tree.
                               // When the tree is empty, root is null.
```

A recursive subroutine named `treeContains` is used to search for a given item in the tree. This routine implements the search algorithm for binary trees that was outlined above:

```
/**
 * Return true if item is one of the items in the binary
 * sort tree to which root points.   Return false if not.
 */
static boolean treeContains( TreeNode root, String item ) {
   if ( root == null ) {
          // Tree is empty, so it certainly doesn't contain item.
      return false;
   }
   else if ( item.equals(root.item) ) {
```

```
             // Yes, the item has been found in the root node.
         return true;
      }
      else if ( item.compareTo(root.item) < 0 ) {
             // If the item occurs, it must be in the left subtree.
         return treeContains( root.left, item );
      }
      else {
             // If the item occurs, it must be in the right subtree.
         return treeContains( root.right, item );
      }
   }  // end treeContains()
```

When this routine is called in the `main()` routine, the first parameter is the static member variable `root`, which points to the root of the entire binary sort tree.

It's worth noting that recursion is not really essential in this case. A simple, non-recursive algorithm for searching a binary sort tree follows the rule: Start at the root and move down the tree until you find the item or reach a null pointer. Since the search follows a single path down the tree, it can be implemented as a `while` loop. Here is non-recursive version of the search routine:

```
private static boolean treeContainsNR( TreeNode root, String item ) {
   TreeNode runner;  // For "running" down the tree.
   runner = root;    // Start at the root node.
   while (true) {
      if (runner == null) {
            // We've fallen off the tree without finding item.
         return false;
      }
      else if ( item.equals(node.item) ) {
            // We've found the item.
         return true;
      }
      else if ( item.compareTo(node.item) < 0 ) {
            // If the item occurs, it must be in the left subtree,
            // So, advance the runner down one level to the left.
         runner = runner.left;
      }
      else {
            // If the item occurs, it must be in the right subtree.
            // So, advance the runner down one level to the right.
         runner = runner.right;
      }
   }  // end while
} // end treeContainsNR();
```

The subroutine for inserting a new item into the tree turns out to be more similar to the non-recursive search routine than to the recursive. The insertion routine has to handle the case where the tree is empty. In that case, the value of `root` must be changed to point to a node that contains the new item:

```
root = new TreeNode( newItem );
```

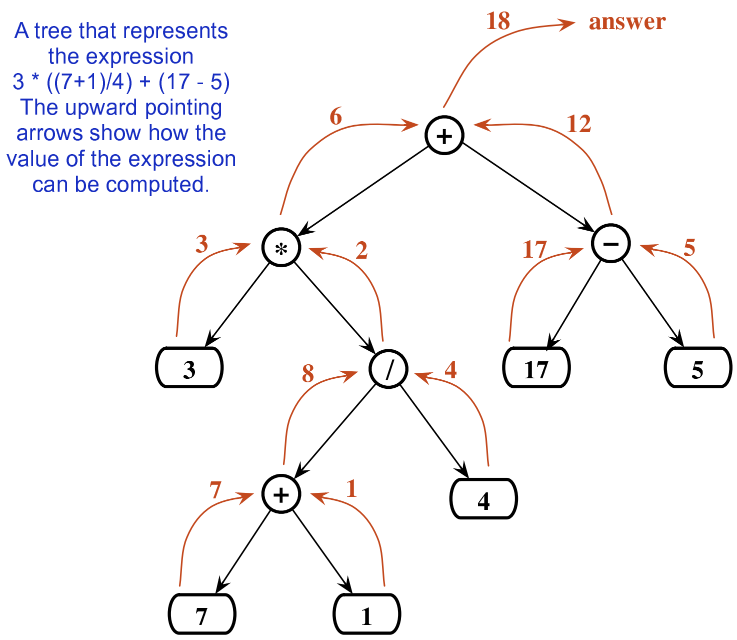
But this means, effectively, that the root can't be passed as a parameter to the subroutine, because it is impossible for a subroutine to change the value stored in an actual parameter. (I should note that this is something that **is** possible in other languages.) Recursion uses parameters in an essential way. There are ways to work around the problem, but the easiest thing is just to use a non-recursive insertion routine that accesses the static member variable `root` directly. One difference between inserting an item and searching for an item is that we have to be careful not to fall off the tree. That is, we have to stop searching just **before** `runner` becomes `null`. When we get to an empty spot in the tree, that's where we have to insert the new node:

```
/**
 * Add the item to the binary sort tree to which the global variable
 * "root" refers.  (Note that root can't be passed as  a parameter to
 * this routine because the value of root might change, and a change
 * in the value of a formal parameter does not change the actual parameter.)
 */
private static void treeInsert(String newItem) {
   if ( root == null ) {
          // The tree is empty.  Set root to point to a new node containing
          // the new item.  This becomes the only node in the tree.
      root = new TreeNode( newItem );
      return;
   }
   TreeNode runner;  // Runs down the tree to find a place for newItem.
   runner = root;   // Start at the root.
   while (true) {
      if ( newItem.compareTo(runner.item) < 0 ) {
             // Since the new item is less than the item in runner,
             // it belongs in the left subtree of runner.  If there
             // is an open space at runner.left, add a new node there.
             // Otherwise, advance runner down one level to the left.
         if ( runner.left == null ) {
            runner.left = new TreeNode( newItem );
            return;  // New item has been added to the tree.
         }
         else
            runner = runner.left;
      }
      else {
             // Since the new item is greater than or equal to the item in
             // runner, it belongs in the right subtree of runner.  If there
             // is an open space at runner.right, add a new node there.
             // Otherwise, advance runner down one level to the right.
         if ( runner.right == null ) {
            runner.right = new TreeNode( newItem );
            return;  // New item has been added to the tree.
         }
         else
            runner = runner.right;
      }
   } // end while
}  // end treeInsert()
```

9.4.3 Expression Trees

Another application of trees is to store mathematical expressions such as `15*(x+y)` or `sqrt(42)+7` in a convenient form. Let's stick for the moment to expressions made up of numbers and the operators `+`, `-`, `*`, and `/`. Consider the expression `3*((7+1)/4)+(17-5)`. This expression is made up of two subexpressions, `3*((7+1)/4)` and `(17-5)`, combined with the operator "+". When the expression is represented as a binary tree, the root node holds the operator `+`, while the subtrees of the root node represent the subexpressions `3*((7+1)/4)` and `(17-5)`. Every node in the tree holds either a number or an operator. A node that holds a number is a leaf node of the tree. A node that holds an operator has two subtrees representing the operands to which the operator applies. The tree is shown in the illustration below. I will refer to a tree of this type as an ***expression tree***.

Given an expression tree, it's easy to find the value of the expression that it represents. Each node in the tree has an associated value. If the node is a leaf node, then its value is simply the number that the node contains. If the node contains an operator, then the associated value is computed by first finding the values of its child nodes and then applying the operator to those values. The process is shown by the upward-directed arrows in the illustration. The value computed for the root node is the value of the expression as a whole. There are other uses for expression trees. For example, a postorder traversal of the tree will output the postfix form of the expression.


A tree that represents the expression 3 * ((7+1)/4) + (17 - 5) The upward pointing arrows show how the value of the expression can be computed.
18
answer
6
12
3
2
17
5
8
4
7
1

An expression tree contains two types of nodes: nodes that contain numbers and nodes that contain operators. Furthermore, we might want to add other types of nodes to make the trees more useful, such as nodes that contain variables. If we want to work with expression trees in Java, how can we deal with this variety of nodes? One way—which will be frowned upon by object-oriented purists—is to include an instance variable in each node object to record which type of node it is:

```
enum NodeType { NUMBER, OPERATOR }   // Possible kinds of node.
```

```
class ExpNode {  // A node in an expression tree.

    NodeType kind;  // Which type of node is this?
    double number;  // The value in a node of type NUMBER.
    char op;        // The operator in a node of type OPERATOR.
    ExpNode left;   // Pointers to subtrees,
    ExpNode right;  //     in a node of type OPERATOR.

    ExpNode( double val ) {
          // Constructor for making a node of type NUMBER.
       kind = NodeType.NUMBER;
       number = val;
    }

    ExpNode( char op, ExpNode left, ExpNode right ) {
          // Constructor for making a node of type OPERATOR.
       kind = NodeType.OPERATOR;
       this.op = op;
       this.left = left;
       this.right = right;
    }

 } // end class ExpNode
```

Given this definition, the following recursive subroutine will find the value of an expression tree:

```
static double getValue( ExpNode node ) {
      // Return the value of the expression represented by
      // the tree to which node refers.  Node must be non-null.
    if ( node.kind == NodeType.NUMBER ) {
          // The value of a NUMBER node is the number it holds.
       return node.number;
    }
    else {  // The kind must be OPERATOR.
            // Get the values of the operands and combine them
            //    using the operator.
       double leftVal = getValue( node.left );
       double rightVal = getValue( node.right );
       switch ( node.op ) {
          case '+':  return leftVal + rightVal;
          case '-':  return leftVal - rightVal;
          case '*':  return leftVal * rightVal;
          case '/':  return leftVal / rightVal;
          default:   return Double.NaN;  // Bad operator.
       }
    }
 } // end getValue()
```

Although this approach works, a more object-oriented approach is to note that since there are two types of nodes, there should be two classes to represent them, *ConstNode* and *BinOpNode*. To represent the general idea of a node in an expression tree, we need another class, *ExpNode*. Both *ConstNode* and *BinOpNode* will be subclasses of *ExpNode*. Since any actual node will be either a `ConstNode` or a *BinOpNode*, *ExpNode* should be an abstract class. (See Subsection 5.5.5.) Since one of the things we want to do with nodes is find their values, each class should have an instance method for finding the value:

```
abstract class ExpNode {
       // Represents a node of any type in an expression tree.

    abstract double value();  // Return the value of this node.

} // end class ExpNode


class ConstNode extends ExpNode {
       // Represents a node that holds a number.

    double number;  // The number in the node.

    ConstNode( double val ) {
          // Constructor.  Create a node to hold val.
       number = val;
    }

    double value() {
          // The value is just the number that the node holds.
       return number;
    }

 } // end class ConstNode


 class BinOpNode extends ExpNode {
       // Represents a node that holds an operator.

    char op;        // The operator.
    ExpNode left;   // The left operand.
    ExpNode right;  // The right operand.

    BinOpNode( char op, ExpNode left, ExpNode right ) {
          // Constructor.  Create a node to hold the given data.
       this.op = op;
       this.left = left;
       this.right = right;
    }

    double value() {
          // To get the value, compute the value of the left and
          // right operands, and combine them with the operator.
        double leftVal = left.value();
        double rightVal = right.value();
        switch ( op ) {
            case '+':  return leftVal + rightVal;
            case '-':  return leftVal - rightVal;
            case '*':  return leftVal * rightVal;
            case '/':  return leftVal / rightVal;
            default:   return Double.NaN;  // Bad operator.
         }
    }

 } // end class BinOpNode
```

Note that the left and right operands of a *BinOpNode* are of type *ExpNode*, not *BinOpNode*. This allows the operand to be either a *ConstNode* or another *BinOpNode*—or any other type of *ExpNode* that we might eventually create. Since every *ExpNode* has a `value()` method, we can call `left.value()` to compute the value of the left operand. If `left` is in fact a *ConstNode*, this will call the `value()` method in the *ConstNode* class. If it is in fact a *BinOpNode*, then `left.value()` will call the `value()` method in the *BinOpNode* class. Each node knows how to compute its own value.

Although it might seem more complicated at first, the object-oriented approach has some advantages. For one thing, it doesn't waste memory. In the original *ExpNode* class, only some of the instance variables in each node were actually used, and we needed an extra instance variable to keep track of the type of node. More important, though, is the fact that new types of nodes can be added more cleanly, since it can be done by creating a new subclass of *ExpNode* rather than by modifying an existing class.

We'll return to the topic of expression trees in the next section, where we'll see how to create an expression tree to represent a given expression.

9.5 A Simple Recursive Descent Parser

I have always been fascinated by language—both natural languages like English and the artificial languages that are used by computers. There are many difficult questions about how languages can convey information, how they are structured, and how they can be processed. Natural and artificial languages are similar enough that the study of programming languages, which are pretty well understood, can give some insight into the much more complex and difficult natural languages. And programming languages raise more than enough interesting issues to make them worth studying in their own right. How can it be, after all, that computers can be made to "understand" even the relatively simple languages that are used to write programs? Computers, after all, can only directly use instructions expressed in very simple machine language. Higher level languages must be translated into machine language. But the translation is done by a compiler, which is just a program. How could such a translation program be written?

9.5.1 Backus-Naur Form

Natural and artificial languages are similar in that they have a structure known as grammar or syntax. Syntax can be expressed by a set of rules that describe what it means to be a legal sentence or program. For programming languages, syntax rules are often expressed in ***BNF*** (Backus-Naur Form), a system that was developed by computer scientists John Backus and Peter Naur in the late 1950s. Interestingly, an equivalent system was developed independently at about the same time by linguist Noam Chomsky to describe the grammar of natural language. BNF cannot express all possible syntax rules. For example, it can't express the fact that a variable must be defined before it is used. Furthermore, it says nothing about the meaning or semantics of the langauge. The problem of specifying the semantics of a language—even of an artificial programming langauge—is one that is still far from being completely solved. However, BNF does express the basic structure of the language, and it plays a central role in the design of translation programs.

In English, terms such as "noun", "transitive verb," and "prepositional phrase" are ***syntactic categories*** that describe building blocks of sentences. Similarly, "statement", "number," and "while loop" are syntactic categories that describe building blocks of Java programs. In BNF, a syntactic category is written as a word enclosed between "<" and ">". For example: `<noun>`, `<verb-phrase>`, or `<while-loop>`. A ***rule*** in BNF specifies the structure of an item in a given syntactic category, in terms of other syntactic categories and/or basic symbols of the language. For example, one BNF rule for the English language might be

```
<sentence>  ::=  <noun-phrase> <verb-phrase>
```

The symbol "`::=`" is read "can be", so this rule says that a `<sentence>` can be a `<noun-phrase>` followed by a `<verb-phrase>`. (The term is "can be" rather than "is" because there might be other rules that specify other possible forms for a sentence.) This rule can be thought of as a recipe for a sentence: If you want to make a sentence, make a noun-phrase and follow it by a verb-phrase. Noun-phrase and verb-phrase must, in turn, be defined by other BNF rules.

In BNF, a choice between alternatives is represented by the symbol "`|`", which is read "or". For example, the rule

```
<verb-phrase>  ::=  <intransitive-verb>  |
                    ( <transitive-verb> <noun-phrase> )
```

says that a `<verb-phrase>` can be an `<intransitive-verb>`, or a `<transitive-verb>` followed by a `<noun-phrase>`. Note also that parentheses can be used for grouping. To express the fact that an item is optional, it can be enclosed between "`[`" and "`]`". An optional item that can be repeated one or more times is enclosed between "`[`" and "`]...`". And a symbol that is an actual part of the language that is being described is enclosed in quotes. For example,

```
<noun-phrase>  ::=  <common-noun> [ "that" <verb-phrase> ]  |
                    <common-noun> [ <prepositional-phrase> ]...
```

says that a `<noun-phrase>` can be a `<common-noun>`, optionally followed by the literal word "`that`" and a `<verb-phrase>`, or it can be a `<common-noun>` followed by zero or more `<prepositional-phrase>'s`. Obviously, we can describe very complex structures in this way. The real power comes from the fact that BNF rules can be **recursive**. In fact, the two preceding rules, taken together, are recursive. A `<noun-phrase>` is defined partly in terms of `<verb-phrase>`, while `<verb-phrase>` is defined partly in terms of `<noun-phrase>`. For example, a `<noun-phrase>` might be "the rat that ate the cheese", since "ate the cheese" is a `<verb-phrase>`. But then we can, recursively, make the more complex `<noun-phrase>` "the cat that caught the rat that ate the cheese" out of the <common-noun> "the cat", the word "that" and the `<verb-phrase>` "caught the rat that ate the cheese". Building from there, we can make the `<noun-phrase>` "the dog that chased the cat that caught the rat that ate the cheese". The recursive structure of language is one of the most fundamental properties of language, and the ability of BNF to express this recursive structure is what makes it so useful.

BNF can be used to describe the syntax of a programming language such as Java in a formal and precise way. For example, a `<while-loop>` can be defined as

```
<while-loop>  ::=  "while" "(" <condition> ")" <statement>
```

This says that a `<while-loop>` consists of the word "while", followed by a left parenthesis, followed by a `<condition>`, followed by a right parenthesis, followed by a `<statement>`. Of course, it still remains to define what is meant by a condition and by a statement. Since a statement can be, among other things, a `while` loop, we can already see the recursive structure of the Java language. The exact specification of an `if` statement, which is hard to express clearly in words, can be given as

```
<if-statement>  ::=
             "if" "(" <condition> ")" <statement>
             [ "else" "if" "(" <condition> ")" <statement> ]...
             [ "else" <statement> ]
```

This rule makes it clear that the "`else`" part is optional and that there can be, optionally, one or more "`else if`" parts.

9.5.2 Recursive Descent Parsing

In the rest of this section, I will show how a BNF grammar for a language can be used as a guide for constructing a parser. A parser is a program that determines the grammatical structure of a phrase in the language. This is the first step to determining the meaning of the phrase—which for a programming language means translating it into machine language. Although we will look at only a simple example, I hope it will be enough to convince you that compilers can in fact be written and understood by mortals and to give you some idea of how that can be done.

The parsing method that we will use is called ***recursive descent parsing***. It is not the only possible parsing method, or the most efficient, but it is the one most suited for writing compilers by hand (rather than with the help of so called "parser generator" programs). In a recursive descent parser, every rule of the BNF grammar is the model for a subroutine. Not every BNF grammar is suitable for recursive descent parsing. The grammar must satisfy a certain property. Essentially, while parsing a phrase, it must be possible to tell what syntactic category is coming up next just by looking at the next item in the input. Many grammars are designed with this property in mind.

I should also mention that many variations of BNF are in use. The one that I've described here is one that is well-suited for recursive descent parsing.

* * *

When we try to parse a phrase that contains a syntax error, we need some way to respond to the error. A convenient way of doing this is to throw an exception. I'll use an exception class called *ParseError*, defined as follows:

```
/**
 * An object of type ParseError represents a syntax error found in
 * the user's input.
 */
private static class ParseError extends Exception {
   ParseError(String message) {
      super(message);
   }
} // end nested class ParseError
```

Another general point is that our BNF rules don't say anything about spaces between items, but in reality we want to be able to insert spaces between items at will. To allow for this, I'll always call the routine `TextIO.skipBlanks()` before trying to look ahead to see what's coming up next in input. `TextIO.skipBlanks()` skips past any whitespace, such as spaces and tabs, in the input, and stops when the next character in the input is either a non-blank character or the end-of-line character.

Let's start with a very simple example. A "fully parenthesized expression" can be specified in BNF by the rules

```
<expression>  ::=  <number>  |
                   "(" <expression> <operator> <expression> ")"

<operator>  ::=  "+" | "-" | "*" | "/"
```

where `<number>` refers to any non-negative real number. An example of a fully parenthesized expression is "`(((34-17)*8)+(2*7))`". Since every operator corresponds to a pair of parentheses, there is no ambiguity about the order in which the operators are to be applied. Suppose we want a program that will read and evaluate such expressions. We'll read the expressions from standard input, using *TextIO*. To apply recursive descent parsing, we need a subroutine for each rule in the grammar. Corresponding to the rule for `<operator>`, we get a subroutine that reads an operator. The operator can be a choice of any of four things. Any other input will be an error.

```
/**
 * If the next character in input is one of the legal operators,
 * read it and return it.  Otherwise, throw a ParseError.
 */
static char getOperator() throws ParseError {
   TextIO.skipBlanks();
   char op = TextIO.peek();
   if ( op == '+' || op == '-' || op == '*' || op == '/' ) {
      TextIO.getAnyChar();
      return op;
   }
   else if (op == '\n')
      throw new ParseError("Missing operator at end of line.");
   else
      throw new ParseError("Missing operator.  Found \"" +
            op + "\" instead of +, -, *, or /.");
} // end getOperator()
```

I've tried to give a reasonable error message, depending on whether the next character is an end-of-line or something else. I use `TextIO.peek()` to look ahead at the next character before I read it, and I call `TextIO.skipBlanks()` before testing `TextIO.peek()` in order to ignore any blanks that separate items. I will follow this same pattern in every case.

When we come to the subroutine for `<expression>`, things are a little more interesting. The rule says that an expression can be either a number or an expression enclosed in parentheses. We can tell which it is by looking ahead at the next character. If the character is a digit, we have to read a number. If the character is a "(", we have to read the "(", followed by an expression, followed by an operator, followed by another expression, followed by a ")". If the next character is anything else, there is an error. Note that we need recursion to read the nested expressions. The routine doesn't just read the expression. It also computes and returns its value. This requires semantical information that is not specified in the BNF rule.

```
/**
 * Read an expression from the current line of input and return its value.
 * @throws ParseError if the input contains a syntax error
 */
private static double expressionValue() throws ParseError {
   TextIO.skipBlanks();
   if ( Character.isDigit(TextIO.peek()) ) {
         // The next item in input is a number, so the expression
         // must consist of just that number.  Read and return
         // the number.
      return TextIO.getDouble();
   }
   else if ( TextIO.peek() == '(' ) {
```

```
            // The expression must be of the form
            //         "(" <expression> <operator> <expression> ")"
            // Read all these items, perform the operation, and
            // return the result.
         TextIO.getAnyChar();  // Read the "("
         double leftVal = expressionValue();  // Read and evaluate first operand.
         char op = getOperator();             // Read the operator.
         double rightVal = expressionValue(); // Read and evaluate second operand.
         TextIO.skipBlanks();
         if ( TextIO.peek() != ')' ) {
               // According to the rule, there must be a ")" here.
               // Since it's missing, throw a ParseError.
            throw new ParseError("Missing right parenthesis.");
         }
         TextIO.getAnyChar();  // Read the ")"
         switch (op) {   //  Apply the operator and return the result.
         case '+':  return leftVal + rightVal;
         case '-':  return leftVal - rightVal;
         case '*':  return leftVal * rightVal;
         case '/':  return leftVal / rightVal;
         default:   return 0;  // Can't occur since op is one of the above.
                               // (But Java syntax requires a return value.)
         }
      }
      else {  // No other character can legally start an expression.
         throw new ParseError("Encountered unexpected character, \"" +
               TextIO.peek() + "\" in input.");
      }
   } // end expressionValue()
```

I hope that you can see how this routine corresponds to the BNF rule. Where the rule uses "|" to give a choice between alternatives, there is an `if` statement in the routine to determine which choice to take. Where the rule contains a sequence of items, "(" `<expression>` `<operator>` `<expression>` ")", there is a sequence of statements in the subroutine to read each item in turn.

When `expressionValue()` is called to evaluate the expression `(((34-17)*8)+(2*7))`, it sees the "(" at the beginning of the input, so the `else` part of the `if` statement is executed. The "(" is read. Then the first recursive call to `expressionValue()` reads and evaluates the subexpression `((34-17)*8)`, the call to `getOperator()` reads the "+" operator, and the second recursive call to `expressionValue()` reads and evaluates the second subexpression `(2*7)`. Finally, the ")" at the end of the expression is read. Of course, reading the first subexpression, `((34-17)*8)`, involves further recursive calls to the `expressionValue()` routine, but it's better not to think too deeply about that! Rely on the recursion to handle the details.

You'll find a complete program that uses these routines in the file *SimpleParser1.java*.

* * *

Fully parenthesized expressions aren't very natural for people to use. But with ordinary expressions, we have to worry about the question of operator precedence, which tells us, for example, that the "`*`" in the expression "`5+3*7`" is applied before the "`+`". The complex expression "`3*6+8*(7+1)/4-24`" should be seen as made up of three "terms", `3*6`, `8*(7+1)/4`, and `24`, combined with "`+`" and "`-`" operators. A term, on the other hand, can be made up of several factors combined with "`*`" and "`/`" operators. For example, `8*(7+1)/4` contains the factors `8`, `(7+1)` and `4`. This example also shows that a factor can be either a number or an expression in parentheses. To complicate things a bit more, we allow for leading minus signs in expressions, as in "`-(3+4)`" or "`-7`". (Since a `<number>` is a positive number, this is the only way we can get negative numbers. It's done this way to avoid "`3 * -7`", for example.) This structure can be expressed by the BNF rules

```
<expression>  ::=  [ "-" ] <term> [ ( "+" | "-" ) <term> ]...
<term>  ::=  <factor> [ ( "*" | "/" ) <factor> ]...
<factor>  ::=  <number>  |  "(" <expression> ")"
```

The first rule uses the "`[ ]...`" notation, which says that the items that it encloses can occur zero, one, two, or more times. This means that an `<expression>` can begin, optionally, with a "-". Then there must be a `<term>` which can optionally be followed by one of the operators "+" or "-" and another `<term>`, optionally followed by another operator and `<term>`, and so on. In a subroutine that reads and evaluates expressions, this repetition is handled by a `while` loop. An `if` statement is used at the beginning of the loop to test whether a leading minus sign is present:

```
/**
 * Read an expression from the current line of input and return its value.
 * @throws ParseError if the input contains a syntax error
 */
private static double expressionValue() throws ParseError {
   TextIO.skipBlanks();
   boolean negative;  // True if there is a leading minus sign.
   negative = false;
   if (TextIO.peek() == '-') {
      TextIO.getAnyChar();  // Read the minus sign.
      negative = true;
   }
   double val;  // Value of the expression.
   val = termValue();
   if (negative)
      val = -val;
   TextIO.skipBlanks();
   while ( TextIO.peek() == '+' || TextIO.peek() == '-' ) {
          // Read the next term and add it to or subtract it from
          // the value of previous terms in the expression.
      char op = TextIO.getAnyChar();  // Read the operator.
      double nextVal = termValue();
      if (op == '+')
         val += nextVal;
      else
         val -= nextVal;
      TextIO.skipBlanks();
   }
   return val;
} // end expressionValue()
```

The subroutine for `<term>` is very similar to this, and the subroutine for `<factor>` is similar to the example given above for fully parenthesized expressions. A complete program that reads and evaluates expressions based on the above BNF rules can be found in the file *SimpleParser2.java*.

9.5.3 Building an Expression Tree

Now, so far, we've only evaluated expressions. What does that have to do with translating programs into machine language? Well, instead of actually evaluating the expression, it would be almost as easy to generate the machine language instructions that are needed to evaluate the expression. If we are working with a "stack machine", these instructions would be stack operations such as "push a number" or "apply a + operation". The program *SimpleParser3.java* can both evaluate the expression and print a list of stack machine operations for evaluating the expression.

It's quite a jump from this program to a recursive descent parser that can read a program written in Java and generate the equivalent machine language code—but the conceptual leap is not huge.

The `SimpleParser3` program doesn't actually generate the stack operations directly as it parses an expression. Instead, it builds an expression tree, as discussed in Section 9.4, to represent the expression. The expression tree is then used to find the value and to generate the stack operations. The tree is made up of nodes belonging to classes *ConstNode* and *BinOpNode* that are similar to those given in Section 9.4. Another class, *UnaryMinusNode*, has been introduced to represent the unary minus operation. I've added a method, `printStackCommands()`, to each class. This method is responsible for printing out the stack operations that are necessary to evaluate an expression. Here for example is the new `BinOpNode` class from *SimpleParser3.java*:

```
private static class BinOpNode extends ExpNode {
   char op;        // The operator.
   ExpNode left;   // The expression for its left operand.
   ExpNode right;  // The expression for its right operand.
   BinOpNode(char op, ExpNode left, ExpNode right) {
          // Construct a BinOpNode containing the specified data.
      assert op == '+' || op == '-' || op == '*' || op == '/';
      assert left != null && right != null;
      this.op = op;
      this.left = left;
      this.right = right;
   }
   double value() {
          // The value is obtained by evaluating the left and right
          // operands and combining the values with the operator.
      double x = left.value();
      double y = right.value();
      switch (op) {
      case '+':
         return x + y;
      case '-':
         return x - y;
      case '*':
         return x * y;
      case '/':
         return x / y;
      default:
         return Double.NaN;  // Bad operator!
      }
   }
   void  printStackCommands() {
```

```
            // To evaluate the expression on a stack machine, first do
            // whatever is necessary to evaluate the left operand, leaving
            // the answer on the stack.  Then do the same thing for the
            // second operand.  Then apply the operator (which means popping
            // the operands, applying the operator, and pushing the result).
         left.printStackCommands();
         right.printStackCommands();
         TextIO.putln("  Operator " + op);
      }
   }
```

It's also interesting to look at the new parsing subroutines. Instead of computing a value, each subroutine builds an expression tree. For example, the subroutine corresponding to the rule for `<expression>` becomes

```
static ExpNode expressionTree() throws ParseError {
       // Read an expression from the current line of input and
       // return an expression tree representing the expression.
   TextIO.skipBlanks();
   boolean negative;  // True if there is a leading minus sign.
   negative = false;
   if (TextIO.peek() == '-') {
      TextIO.getAnyChar();
      negative = true;
   }
   ExpNode exp;   // The expression tree for the expression.
   exp = termTree();  // Start with a tree for first term.
   if (negative) {
         // Build the tree that corresponds to applying a
         // unary minus operator to the term we've
         // just read.
      exp = new UnaryMinusNode(exp);
   }
   TextIO.skipBlanks();
   while ( TextIO.peek() == '+' || TextIO.peek() == '-' ) {
           // Read the next term and combine it with the
           // previous terms into a bigger expression tree.
        char op = TextIO.getAnyChar();
        ExpNode nextTerm = termTree();
           // Create a tree that applies the binary operator
           // to the previous tree and the term we just read.
        exp = new BinOpNode(op, exp, nextTerm);
        TextIO.skipBlanks();
   }
   return exp;
} // end expressionTree()
```

In some real compilers, the parser creates a tree to represent the program that is being parsed. This tree is called a ***parse tree***. Parse trees are somewhat different in form from expression trees, but the purpose is the same. Once you have the tree, there are a number of things you can do with it. For one thing, it can be used to generate machine language code. But there are also techniques for examining the tree and detecting certain types of programming errors, such as an attempt to reference a local variable before it has been assigned a value. (The Java compiler, of course, will reject the program if it contains such an error.) It's also possible to manipulate the tree to ***optimize*** the program. In optimization, the tree is transformed to make the program more efficient before the code is generated.

And so we are back where we started in Chapter 1, looking at programming languages, compilers, and machine language. But looking at them, I hope, with a lot more understanding and a much wider perspective.

Exercises for Chapter 9

1. In many textbooks, the first examples of recursion are the mathematical functions *factorial* and *fibonacci*. These functions are defined for non-negative integers using the following recursive formulas:

```
factorial(0)  =  1
factorial(N)  =  N*factorial(N-1)   for N > 0

fibonacci(0)  =  1
fibonacci(1)  =  1
fibonacci(N)  =  fibonacci(N-1) + fibonacci(N-2)   for N > 1
```

Write recursive functions to compute `factorial(N)` and `fibonacci(N)` for a given non-negative integer `N`, and write a `main()` routine to test your functions.

(In fact, *factorial* and *fibonacci* are really not very good examples of recursion, since the most natural way to compute them is to use simple `for` loops. Furthermore, *fibonacci* is a particularly bad example, since the natural recursive approach to computing this function is extremely inefficient.)

2. Exercise 7.6 asked you to read a file, make an alphabetical list of all the words that occur in the file, and write the list to another file. In that exercise, you were asked to use an ArrayList<String> to store the words. Write a new version of the same program that stores the words in a binary sort tree instead of in an arraylist. You can use the binary sort tree routines from *SortTreeDemo.java*, which was discussed in Subsection 9.4.2.

3. Suppose that linked lists of integers are made from objects belonging to the class

```
class ListNode {
   int item;       // An item in the list.
   ListNode next;  // Pointer to the next node in the list.
}
```

Write a subroutine that will make a copy of a list, with the order of the items of the list reversed. The subroutine should have a parameter of type *ListNode*, and it should return a value of type *ListNode*. The original list should not be modified.

You should also write a `main()` routine to test your subroutine.

4. Subsection 9.4.1 explains how to use recursion to print out the items in a binary tree in various orders. That section also notes that a non-recursive subroutine can be used to print the items, provided that a stack or queue is used as an auxiliary data structure. Assuming that a queue is used, here is an algorithm for such a subroutine:

```
Add the root node to an empty queue
while the queue is not empty:
   Get a node from the queue
   Print the item in the node
   if node.left is not null:
      add it to the queue
   if node.right is not null:
      add it to the queue
```

Write a subroutine that implements this algorithm, and write a program to test the subroutine. Note that you will need a queue of *TreeNodes*, so you will need to write a class to represent such queues.

(Note that the order in which items are printed by this algorithm is different from all three of the orders considered in Subsection 9.4.1.)

5. In Subsection 9.4.2, I say that "if the [binary sort] tree is created by inserting items in a random order, there is a high probability that the tree is approximately balanced." For this exercise, you will do an experiment to test whether that is true.

The ***depth*** of a node in a binary tree is the length of the path from the root of the tree to that node. That is, the root has depth 0, its children have depth 1, its grandchildren have depth 2, and so on. In a balanced tree, all the leaves in the tree are about the same depth. For example, in a perfectly balanced tree with 1023 nodes, all the leaves are at depth 9. In an approximately balanced tree with 1023 nodes, the average depth of all the leaves should be not too much bigger than 9.

On the other hand, even if the tree is approximately balanced, there might be a few leaves that have much larger depth than the average, so we might also want to look at the maximum depth among all the leaves in a tree.

For this exercise, you should create a random binary sort tree with 1023 nodes. The items in the tree can be real numbers, and you can create the tree by generating 1023 random real numbers and inserting them into the tree, using the usual `treeInsert()` method for binary sort trees. Once you have the tree, you should compute and output the average depth of all the leaves in the tree and the maximum depth of all the leaves. To do this, you will need three recursive subroutines: one to count the leaves, one to find the sum of the depths of all the leaves, and one to find the maximum depth. The latter two subroutines should have an **int**-valued parameter, `depth`, that tells how deep in the tree you've gone. When you call this routine from the main program, the `depth` parameter is 0; when you call the routine recursively, the parameter increases by 1.

6. The parsing programs in Section 9.5 work with expressions made up of numbers and operators. We can make things a little more interesting by allowing the variable "x" to occur. This would allow expression such as "`3*(x-1)*(x+1)`", for example. Make a new version of the sample program *SimpleParser3.java* that can work with such expressions. In your program, the `main()` routine can't simply print the value of the expression, since the value of the expression now depends on the value of `x`. Instead, it should print the value of the expression for `x=0`, `x=1`, `x=2`, and `x=3`.

The original program will have to be modified in several other ways. Currently, the program uses classes *ConstNode*, *BinOpNode*, and *UnaryMinusNode* to represent nodes in an expression tree. Since expressions can now include `x`, you will need a new class, *VariableNode*, to represent an occurrence of `x` in the expression.

In the original program, each of the node classes has an instance method, "`double value()`", which returns the value of the node. But in your program, the value can depend on `x`, so you should replace this method with one of the form "`double value(double xValue)`", where the parameter `xValue` is the value of `x`.

Finally, the parsing subroutines in your program will have to take into account the fact that expressions can contain `x`. There is just one small change in the BNF rules for the expressions: A `<factor>` is allowed to be the variable `x`:

```
<factor>  ::=  <number>  |  <x-variable>  |  "(" <expression> ")"
```

where `<x-variable>` can be either a lower case or an upper case "X". This change in the BNF requires a change in the `factorTree()` subroutine.

7. This exercise builds on the previous exercise, Exercise 9.6. To understand it, you should have some background in Calculus. The derivative of an expression that involves the variable `x` can be defined by a few recursive rules:

- The derivative of a constant is 0.
- The derivative of `x` is 1.
- If `A` is an expression, let `dA` be the derivative of `A`. Then the derivative of `-A` is `-dA`.
- If `A` and `B` are expressions, let `dA` be the derivative of `A` and let `dB` be the derivative of `B`. Then the derivative of `A+B` is `dA+dB`.
- The derivative of `A-B` is `dA-dB`.
- The derivative of `A*B` is `A*dB + B*dA`.
- The derivative of `A/B` is `(B*dA - A*dB) / (B*B)`.

For this exercise, you should modify your program from the previous exercise so that it can compute the derivative of an expression. You can do this by adding a derivative-computing method to each of the node classes. First, add another abstract method to the *ExpNode* class:

```
abstract ExpNode derivative();
```

Then implement this method in each of the four subclasses of *ExpNode*. All the information that you need is in the rules given above. In your main program, instead of printing the stack operations for the original expression, you should print out the stack operations that define the derivative. Note that the formula that you get for the derivative can be much more complicated than it needs to be. For example, the derivative of `3*x+1` will be computed as `(3*1+0*x)+0`. This is correct, even though it's kind of ugly, and it would be nice for it to be simplified. However, simplifying expressions is not easy.

As an alternative to printing out stack operations, you might want to print the derivative as a fully parenthesized expression. You can do this by adding a `printInfix()` routine to each node class. It would be nice to leave out unnecessary parentheses, but again, the problem of deciding which parentheses can be left out without altering the meaning of the expression is a fairly difficult one, which I don't advise you to attempt.

(There is one curious thing that happens here: If you apply the rules, as given, to an expression tree, the result is no longer a tree, since the same subexpression can occur at multiple points in the derivative. For example, if you build a node to represent `B*B` by saying "`new BinOpNode('*',B,B)`", then the left and right children of the new node are actually the same node! This is not allowed in a tree. However, the difference is harmless in this case since, like a tree, the structure that you get has no loops in it. Loops, on the other hand, would be a disaster in most of the recursive tree-processing subroutines that we have written, since it would lead to infinite recursion.)

Quiz on Chapter 9

1. Explain what is meant by a *recursive* subroutine.

2. Consider the following subroutine:

```
static void printStuff(int level) {
    if (level == 0) {
       System.out.print("*");
    }
    else {
       System.out.print("[");
       printStuff(level - 1);
       System.out.print(",");
       printStuff(level - 1);
       System.out.println("]");
    }
}
```

Show the output that would be produced by the subroutine calls `printStuff(0)`, `printStuff(1)`, `printStuff(2)`, and `printStuff(3)`.

3. Suppose that a linked list is formed from objects that belong to the class

```
class ListNode {
   int item;       // An item in the list.
   ListNode next;  // Pointer to next item in the list.
}
```

Write a subroutine that will count the number of zeros that occur in a given linked list of **ints**. The subroutine should have a parameter of type `ListNode` and should return a value of type **int**.

4. What are the three operations on a *stack?*

5. What is the basic difference between a stack and a queue?

6. What is an *activation record*? What role does a stack of activation records play in a computer?

7. Suppose that a binary tree of integers is formed from objects belonging to the class

```
class TreeNode {
   int item;       // One item in the tree.
   TreeNode left;  // Pointer to the left subtree.
   TreeNode right; // Pointer to the right subtree.
}
```

Write a recursive subroutine that will find the sum of all the nodes in the tree. Your subroutine should have a parameter of type `TreeNode`, and it should return a value of type **int**.

8. What is a *postorder traversal* of a binary tree?

9. Suppose that a `<multilist>` is defined by the BNF rule

```
<multilist>  ::=  <word>  |  "(" [ <multilist> ]... ")"
```

where a `<word>` can be any sequence of letters. Give five different `<multilist>'s` that can be generated by this rule. (This rule, by the way, is almost the entire syntax of the programming language `LISP`! `LISP` is known for its simple syntax and its elegant and powerful semantics.)

10. Explain what is meant by *parsing* a computer program.