7 Stacks

This chapter introduces the stack, a collection that has widespread use in computer science. The stack is the simplest collection to describe and implement. Yet it has fascinating applications, three of which we discuss later in the chapter. In conformance with the plan proposed in Chapter 6, we consider both a prototype and a professional version of stacks, and we present two standard implementations, one based on arrays and the other on linked structures. The chapter closes with a Case Study in which stacks play a central role, the evaluation of postfix arithmetic expressions.

7.1 Overview of Stacks

Stacks are linear collections in which access is completely restricted to just one end, called the top. The classical example is the stack of clean trays found in every cafeteria. Whenever a tray is needed, it is removed from the top of the stack, and whenever clean ones come back from the scullery, they are again placed on the top. No one ever takes some particularly fine tray from the middle of the stack, and it is even possible that trays near the bottom are never used. Stacks are said to adhere to a last in first-out protocol (LIFO). The last tray brought back from the scullery is the first one taken by a customer.

The operations for putting items on and removing items from a stack are called push and pop, respectively. Figure 7.1 shows a stack as it might appear at various stages. Initially, the stack is empty, and then an item called a is pushed. Next, three more items called b, c, and d are pushed, after which the stack is popped, and so forth.

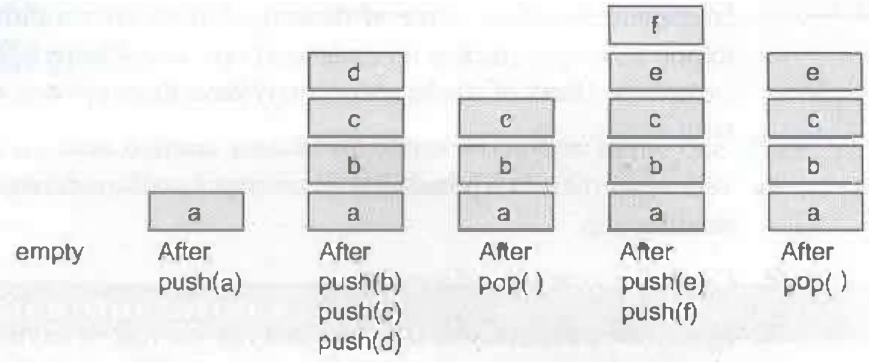

Figure 7.1 Some stages in the lifetime of a stack

Other everyday examples of stacks include plates and bowls in our kitchen cupboards and PEZ® dispensers. Although we continually add more papers to the top of the piles on our desks, these piles do not quite qualify because we often need to remove a long lost paper from the middle. With a genuine stack, the item we want next is always the one added most recently.

Applications of stacks in computer science are numerous. Here are just a few, including three we discuss in more detail later in the chapter:

- Parsing expressions in context-free programming languages—a problem in compiler design.
- Translating infix expressions to postfix form and evaluating postfix expressions—discussed later in the chapter.
- Backtracking algorithms—discussed later in the chapter and occurring in problems such as automated theorem proving and game playing.
- Managing computer memory in support of method calls—discussed later in the chapter.
- Supporting the "undo" feature in text editors, word processors, spreadsheet programs, drawing programs, and similar applications.
- Maintaining a history of the links visited by a Web browser.

7.2 A Stack Prototype

We now consider a stack prototype. Following the plan outlined in Chapter 6, we distinguish between fundamental and supporting operations. We have already introduced the fundamental operations `push` and `pop`. The supporting operations include:

`peek`	which returns the top item on a stack without removing it
`isEmpty`	which returns `true` if there are no more items on a stack
`isFull`	which returns `true` if a stack can hold no more items
`size`	which returns the number of items on a stack

In certain situations, some of these operations are invalid. For instance, attempting to pop an empty stack is an undefined operation that is best handled by throwing an exception. Users of stacks accordingly have three options when using the `pop` operation:

1. Whenever there is a possibility of an empty stack, send the message `isEmpty` before sending `pop`.
2. Catch the exception if it occurs.
3. Ignore the possibility of the error and run the risk of having the program terminate abnormally.

Either of the first two options is reasonable, and a programmer can choose the one that best fits the circumstances. Here is a list of the exceptions thrown by the stack prototype:

`peek` and `pop`	throw an `IllegalStateException` if the stack is empty
`push`	throws an `IllegalStateException` if the stack is full and an `IllegalArgumentException` if the item is `null`

Table 7.1 defines the interface for the stack prototype.

Table 7.1

The Interface for the Stack Prototype (StackPT)

Fundamental Methods

`void`	**`push(Object item)`** Pushes an item onto the top of this stack. Throws an exception if the item is `null` or the stack is full.
`Object`	**`pop()`** Returns the item at the top of this stack and removes it from the stack. Throws an exception if the stack is empty.

Supporting Methods

`Object`	**`peek()`** Returns the item at the top of this stack without removing it from the stack. Throws an exception if the stack is empty.
`boolean`	**`isEmpty()`** Returns `true` if this stack contains no items.
`boolean`	**`isFull()`** Returns `true` if this stack is full and can accept no more items.
`int`	**`size()`** Returns the number of items in this stack.

7.3 Using a Stack

Now that we have defined a stack, we demonstrate how to use one. Table 7.2 shows how the operations listed earlier affect a stack.

Table 7.2

The Effects of Stack Operations

Operation	State of the Stack After the Operation	Value Returned	Comment
			Initially, the stack is empty.
`push(a)`	`a`		The stack contains the single item `a`.
`push(b)`	`a b`		`b` is the top item on the stack.
`push(c)`	`a b c`		`c` is the top item.
`isEmpty()`	`a b c`	false	The stack is not empty.
`size()`	`a b c`	3	The stack contains three items.
`peek()`	`a b c`	`c`	Return the top item on the stack without removing it.
`pop()`	`a b`	`c`	Remove the top item from the stack and return it. `b` is now the top item.
`pop()`	`a`	`b`	Remove and return `b`.
`pop()`		`a`	Remove and return `a`.
`isEmpty()`		`true`	The stack is empty.
`peek()`		`exception`	Peeking at an empty stack throws an exception.
`pop()`		`exception`	Popping an empty stack throws an exception.
`push(d)`	`d`		`d` is the top item.

Before using a stack in a program, we declare a variable of the interface type and instantiate an object of an implementation type. For instance:

```
StackPT stk1 = new ArrayStackPT();          //Declare and instantiate two stacks
StackPT stk2 = new LinkedStackPT();
```

The classes `ArrayStackPT` and `LinkedStackPT` are implementations of the `StackPT` interface and are described in Section 7.4. But before getting there, let us look at a simple application.

7.3.1 Matching Parentheses

Compilers need to determine if the bracketing symbols in expressions are balanced correctly. For example, every opening [should be followed by a properly positioned closing] and every (by a). Here are some examples:

(...)...(...)	Balanced	
(...)...(...	Unbalanced	Missing a closing) at the end
)...(...(...)	Unbalanced	The closing) at the beginning has no matching opening (and one of the opening (s has no closing)
[...(...)...]	Balanced	
[...(...]...)	Unbalanced	The bracketed sections are not nested properly

In these examples, three dots represent arbitrary strings that contain no bracketing symbols. As a first attempt at solving the problem of whether brackets balance, we might simply count the number of left and right parentheses. If the expression balances, the two counts are equal. However, the converse is not true. If the counts are equal, the brackets do not necessarily balance. The third example provides a counterexample.

A more complex approach, using a stack, does work. To check an expression,

1. We scan across it, pushing opening brackets onto a stack.
2. On encountering a closing bracket, if the stack is empty or if the item on the top of the stack is not an opening bracket of the same type, we know the brackets do not balance.
3. If the item on the top of the stack is of the right type, we pop the stack and continue scanning the expression.
4. When we reach the end of the expression, the stack should be empty, and if it is not, we know the brackets do not balance.

A Java method that implements this strategy is:

```
boolean bracketsBalance (String exp){              //exp represents the expression
   StackPT stk = new LinkedStackPT();                        //Create a new stack
   for (int i = 0; i < exp.length(); i++){             //Scan across the expression
      char ch = exp.charAt(i);
      if (ch == '[' || ch == '('){     //Push an opening bracket onto the stack
         stk.push (new Character(ch));
      }else if (ch == ']' || ch == ')'){              //Process a closing bracket
         if (stk.isEmpty())           //If the stack is empty, then not balanced
            return false;
         char charFromStack = ((Character)stk.pop()).charValue();
```

```
        if (ch == ']' && charFromStack != '[' ||   //If the opening and closing
           (ch == ')' && charFromStack != '('))   //brackets are of different
          return false;                          //types, then not balanced
      }
    }
    return stk.isEmpty();                  //If the stack is empty, then balanced,
}                                                  //else not balanced
```

7.5 Three Applications of Stacks

We now discuss three applications of stacks. First, we present algorithms for evaluating arithmetic expressions. These algorithms apply to problems in compiler design, and we will use them in the chapter's case study and in one of the exercises. Second, we describe a general technique for using stacks to solve backtracking problems. The exercises explore applications of the technique. Third, we examine the role of stacks in computer memory management. Not only is this topic interesting in its own right, but it provides a foundation for understanding recursion (see Chapter 10).

7.5.1 Evaluating Arithmetic Expressions

We are so accustomed to evaluating simple arithmetic expressions that we give little conscious thought to the rules involved, and we are surprised by the difficulty of writing an algorithm to do the same thing. It turns out that an indirect approach to the problem works best. First, we transform an expression from its familiar infix form to a postfix form, and then we evaluate the postfix form. In the infix form, each operator is located between its operands, whereas in the postfix form, an operator immediately follows its operands. Table 7.3 gives several simple examples.

Table 7.3

Some Infix and Postfix Expressions

Infix Form	Postfix Form	Value
`34`	`34`	34
`34 + 22`	`34 22 +`	56
`34 + 22 * 2`	`34 22 2 * +`	78
`34 * 22 + 2`	`34 22 * 2 +`	750
`(34 + 22) * 2`	`34 22 + 2 *`	112

There are similarities and differences between the two forms. In both, operands appear in the same order; however, the operators do not. The infix form sometimes requires parentheses; the postfix form never does. Infix evaluation involves rules of precedence; postfix evaluation applies operators as soon as they are encountered. For instance, consider the steps in evaluating the infix expression `34 + 22 * 2` and the equivalent postfix expression `34 22 2 * +`.

Infix evaluation: 34 + 22 * 2 → 34 + 44 → 78

Postfix evaluation: 34 22 2 * + → 34 44 + → 78

We now present stack-based algorithms for transforming infix expressions to postfix and for evaluating the resulting postfix expressions. In combination, these algorithms allow us to evaluate an infix expression. In presenting the algorithms, we ignore the effects of syntax errors, but return to the issue in the Case Study and the exercises.

Evaluating Postfix Expressions

Evaluation is the simpler process and consists of three steps:

1. Scan across the expression from left to right.
2. On encountering an operator, apply it to the two preceding operands and replace all three by the result.
3. Continue scanning until reaching the expression's end, at which point only the expression's value remains.

To express this procedure as a computer algorithm, we use a stack of operands. In the algorithm, the term token refers to either an operand or an operator:

```
create a new stack
while there are more tokens in the expression
   get the next token
   if the token is an operand
      push the operand onto the stack
   else if the token is an operator
      pop the top two operands from the stack
      use the operator to evaluate the two operands just popped
      push the resulting operand onto the stack
   end if
end while
return the value at the top of the stack
```

The time complexity of the algorithm is $O(n)$, where n is the number of tokens in the expression (see the exercises). Table 7.4 shows a trace of the algorithm as it applies to the expression `4 5 6 * + 3 -`.

Table 7.4

Tracing the Evaluation of a Postfix Expression

Postfix Expression: `4 5 6 * + 3 -` Resulting Value: `31`

Portion of Postfix Expression Scanned So Far	Operand Stack	Comment
		No tokens have been seen yet. The stack is empty.
`4`	`4`	Push the operand 4.
`4 5`	`4 5`	Push the operand 5.
`4 5 6`	`4 5 6`	Push the operand 6.
`4 5 6 *`	`4 30`	Replace the top two operands by their product.
`4 5 6 * +`	`34`	Replace the top two operands by their sum.
`4 5 6 * + 3`	`34 3`	Push the operand 3.
`4 5 6 * + 3 -`	`31`	Replace the top two operands by their difference.
		Pop the final value.

Transforming Infix to Postfix

We now show how to translate expressions from infix to postfix. For the sake of simplicity, we restrict our attention to expressions involving the operators *, /, +, and – (an exercise at the end of the chapter enlarges the set of operators). As usual, multiplication and division have higher precedence than addition and subtraction, except when parentheses override the default order of evaluation.

In broad terms, the algorithm scans, from left to right, a string containing an infix expression and simultaneously builds a string containing the equivalent postfix expression. Operands are copied from the infix string to the postfix string as soon as they are encountered. However, operators must be held back on a stack until operators of greater precedence have been copied to the postfix string ahead of them. Here is a more detailed statement of the process:

1. Start with an empty postfix expression and an empty stack, which will hold operators and left parentheses.
2. Scan across the infix expression from left to right.
3. On encountering an operand, append it to the postfix expression.
4. On encountering an operator, pop off the stack all operators that have equal or higher precedence and append them to the postfix expression. Then push the scanned operator onto the stack.

5. On encountering a left parenthesis, push it onto the stack.
6. On encountering a right parenthesis, shift operators from the stack to the postfix expression until meeting the matching left parenthesis, which is discarded.
7. On encountering the end of the infix expression, transfer the remaining operators from the stack to the postfix expression.

Examples in Tables 7.5 and 7.6 illustrate the procedure.

Table 7.5

Tracing the Conversion of an Infix Expression to a Postfix Expression

Infix Expression: `4 + 5 * 6 - 3` Equivalent Postfix Expression: `4 5 6 * + 3 -`

Portion of Infix Expression Scanned So Far	Operator Stack	Postfix Expression	Comment
			No characters have been seen yet. The stack and PE are empty.
`4`		`4`	Append 4 to the PE.
`4 +`	`+`	`4`	Push + onto the stack.
`4 + 5`	`+`	`4 5`	Append 5 to the PE.
`4 + 5 *`	`+ *`	`4 5`	Push * onto the stack.
`4 + 5 * 6`	`+ *`	`4 5 6`	Append 6 to the PE.
`4 + 5 * 6 -`	`-`	`4 5 6 * +`	Pop * and +, append them to the PE, and push –.
`4 + 5 * 6 - 3`	`-`	`4 5 6 * + 3`	Append 3 to the PE.
		`4 5 6 * + 3 -`	Pop the remaining operators off the stack and append them to the PE.

Table 7.6

Tracing the Conversion of an Infix Expression to a Postfix Expression

Infix Expression: (4 + 5) * (6 - 3) Equivalent Postfix Expression: 4 5 + 6 3 - *

Portion of Infix Expression Scanned So Far	Operator Stack	Postfix Expression	Comment
			No characters have been seen yet. The stack and PE are empty.
(	(		Push (onto the stack.
(4	(	4	Append 4 to the PE.
(4 +	(+	4	Push + onto the stack.
(4 + 5	(+	4 5	Append 5 to the PE.
(4 + 5)	(+	4 5 +	Pop the stack until (is encountered and append operators to the PE.
(4 + 5) *	*	4 5 +	Push * onto the stack.
(4 + 5) * (	* (	4 5 +	Push (onto the stack.
(4 + 5) * (6	* (	4 5 + 6	Append 6 to the PE.
(4 + 5) * (6 -	* (-	4 5 + 6	Push – onto the stack.
(4 + 5) * (6 - 3	* (-	4 5 + 6 3	Append 3 to the PE.
(4 + 5) * (6 - 3)	*	4 5 + 6 3 -	Pop stack until (is encountered and append items to the PE.
		4 5 + 6 3 - *	Pop the remaining operators off the stack and append them to the PE.

We leave it to the reader to determine the time complexity of this process and to incorporate the process into a programming project that extends the Case Study (see the exercises).

7.5.2 Backtracking

There are two principal techniques for implementing backtracking algorithms: One uses stacks and the other recursion. Here we explore the use of stacks; in Chapter 10, we consider recursion. To rephrase what was said in Chapter 1, a backtracking algorithm begins in a predefined starting state and then moves from state to state in search of a desired ending state. At any point along the way, when there is a choice between several alternative states, the algorithm picks one, possibly at random, and continues. If the algorithm reaches a state representing an undesirable outcome, it backs up to the last point at which there was an unexplored alternative and tries it. In

this way, the algorithm either exhaustively searches all states, or it reaches the desired ending state.

The role of a stack in the process is to remember the alternative states that occur at each juncture. To be more precise:

```
create an empty stack
push the starting state onto the stack
while the stack is not empty
   pop the stack and examine the state
   if the state represents an ending state
      return SUCCESSFUL CONCLUSION
   else if the state has not been visited previously
      mark the state as visited
      push onto the stack all unvisited adjacent states
   end if
end while
return UNSUCCESSFUL CONCLUSION
```

Later in the book, when we encounter trees and graphs, we will recognize this algorithm as a depth-first search. Although short, the algorithm is subtle, and an illustration is needed to elucidate its workings.

Suppose there are just five states, as shown in Figure 7.7, with states 1 and 5 representing the start and end, respectively. Lines between states indicate adjacency. Starting in state 1, the algorithm proceeds to state 4 and then to 3. In state 3, the algorithm recognizes that all adjacent states have been visited and resumes the search in state 2, which leads directly to state 5 and the end. Table 7.7 contains a detailed trace of the algorithm.

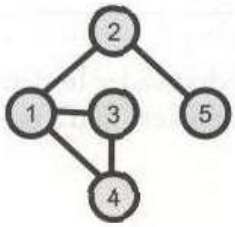

Figure 7.7

Table 7.7

The Trace of the Backtracking Algorithm

Next Step in Algorithm	Stack	Current State	Visited States
Push 1.	1	none yet	
Pop the stack and note the state.		1	
Mark the state as visited.		1	1
Push unvisited adjacent states.	2 3 4	1	1
Pop stack and note the state.	2 3	4	1
Mark the state as visited.	2 3	4	1 4
Push unvisited adjacent states.	2 3 3	4	1 4
Pop the stack and note the state.	2 3	3	1 4
Mark the state as visited.	2 3	3	1 4 3
All adjacent states already visited.	2 3	3	1 4 3
Pop the stack and note the state.	2	3	1 4 3
Do nothing, location already visited.	2	3	1 4 3
Pop the stack and note the state.		2	1 4 3
Mark the state as visited.		2	1 4 3 2
Push unvisited adjacent states.	5	2	1 4 3 2
Pop the stack and note the state.		5	1 4 3 2
SUCCESS: this is the ending state.		5	1 4 3 2

Notice in the trace that the algorithm goes directly from state 3 to state 2 without having to backup through states 4 and 1 first. This is a beneficial side effect of using a stack.

It would be interesting to calculate the time complexity of the foregoing algorithm. However, two crucial pieces of information are missing:

1. The complexity of deciding if a state has been visited.
2. The complexity of listing states adjacent to a given state.

If, for the sake of argument, we assume that both of these processes are O(1), then the algorithm as a whole is O(n), where n represents the total number of states (see the exercises).

This discussion has been a little abstract, but at the end of the section, there are several exercises involving the application of backtracking to maze problems.

7.5.3 Memory Management

During a program's execution, both its code and data occupy computer memory. Although the exact manner in which a computer manages memory depends on the programming language and operating system involved, we can present the following simplified, yet reasonably realistic, overview. The emphasis must be on the word "simplified" because a detailed discussion is beyond the book's scope.

As you probably already know, a Java compiler translates a Java program into bytecodes. A complex program called the Java Virtual Machine (JVM) then executes these. The memory or ***run-time environment*** controlled by the JVM is divided into six regions, as shown on the left side Figure 7.8.

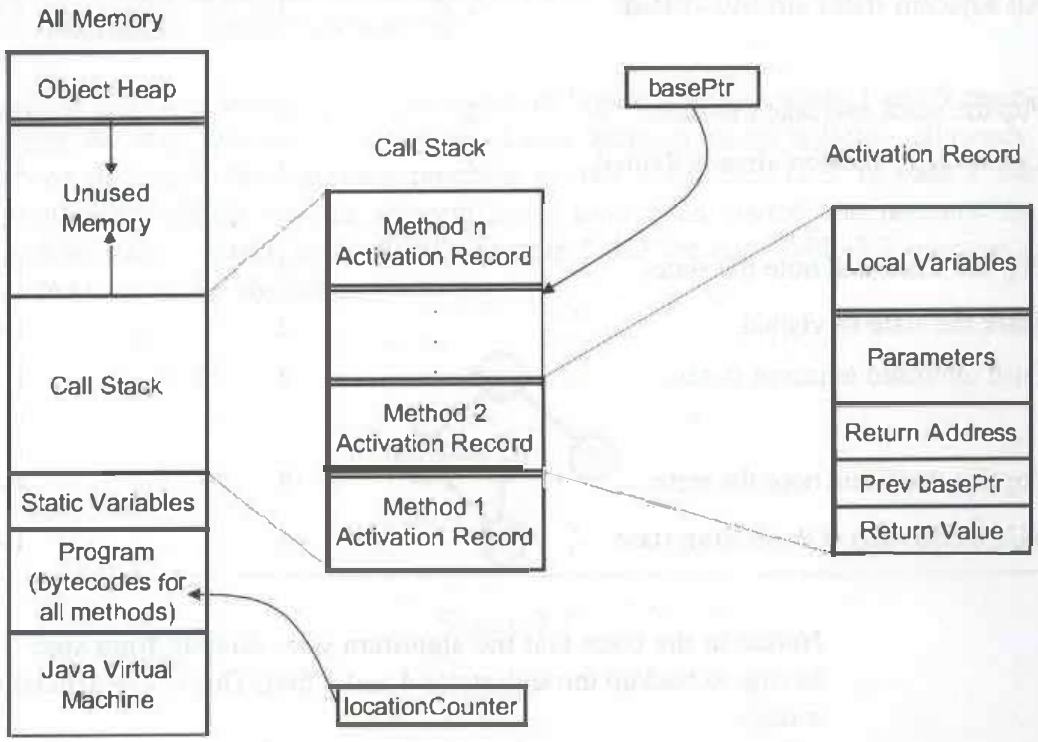

Figure 7.8 The architecture of a run-time environment

Working up from the bottom, these regions contain:

- The ***Java Virtual Machine,*** which executes a Java program. Internal to the JVM are two variables, which we call **locationCounter** and **basePtr.** The **locationCounter** points at the instruction the **JVM** will execute next. The **basePtr** points at the top activation record's base. More is said about these variables soon.
- Bytecodes for all the methods of our ***program.***
- The program's ***static variables.***
- The ***call stack.*** Every time a method is called, an ***activation record*** is created and pushed onto the call stack. When a method finishes execution and returns control to the method that called it, the activation record is popped off the stack. The total number of activation records on the stack equals the number of method calls currently in various stages of execution. At the bottom of the stack is the activation record for the method `main`. Above it is the activation record for the method currently called by `main`, and so forth. More will be said about activation records in a moment.
- ***Unused memory.*** This region's size grows and shrinks in response to the demands of the call stack and the object heap.
- The ***object heap.*** In Java, all objects exist in a region of memory called the heap. When an object is instantiated, the JVM must find space for the object on the heap, and when the object is no longer needed, the JVM's garbage collector recovers the space for future use. When low on space, the heap extends further into the region marked *Unused Memory.*

The activation records shown in the figure contain two types of information. The regions labeled **Local Variables** and **Parameters** hold data needed by the executing method. The remaining regions hold data that allows the JVM to pass control backward from the currently executing method to the method that called it.

When a method is called, the JVM

1. Creates the method's activation record and pushes it onto the call stack (the activation record's bottom three regions are fixed in size, and the top two vary depending on the number of parameters and local variables used by the method).
2. Saves the **basePtr**'s current value in the region labeled **Prev basePtr** and sets the **basePtr** to the new activation record's base.
3. Saves the **locationCounter**'s current value in the region labeled **Return Address** and sets the **locationCounter** to the first instruction of the called method.
4. Copies the calling parameters into the region labeled **Parameters.**
5. Initializes local variables as required.
6. Starts executing the called method at the location indicated by the **locationCounter.**

While a method is executing, local variables and parameters in the activation record are referenced by adding an offset to the **basePtr.** Thus, no matter where an activation record is located in memory, the local variables and parameters can be accessed correctly provided the **basePtr** has been initialized properly.

Just before returning, a method stores its return value in the location labeled **Return Value.** The value can be a reference to an object, or it can be a primitive such as an integer or character. Because the return value always resides at the bottom of the activation record, the calling method knows exactly where to find it.

When a method has finished executing, the JVM

1. Reestablishes the settings needed by the calling method by restoring the values of the **locationCounter** and the **basePtr** from values stored in the activation record.
2. Pops the activation record from the call stack.
3. Resumes execution of the calling method at the location indicated by the **locationCounter**.

Exercises

1. Translate by hand the following infix expressions to postfix form:

   a. 33 – 15 * 6
   b. 11 * (6 + 2)
   c. 17 + 3 – 5
   d. 22 – 6 + 33 / 4

2. Evaluate by hand the following postfix expressions:

   a. 10 5 4 + *
   b. 10 5 * 6 –
   c. 22 2 4 * /
   d. 33 6 + 3 4 / +

3. Perform a complexity analysis for postfix evaluation.

4. Perform a complexity analysis for infix to postfix conversion.

5. Compute the complexity of the depth-first search algorithm given that the processes for determining adjacent states and whether or not a state has been visited are both O(1). (*Hint:* Consider the maximum number of times each state gets pushed onto the stack.)

6. Write a program that solves a maze problem. In this particular version of the problem, a hiker must find a path to the top of a mountain. Assume that the hiker leaves a parking lot, marked P, and explores the maze until she reaches the top of a mountain, marked T. Figure 7.9 shows what a particular maze looks like.

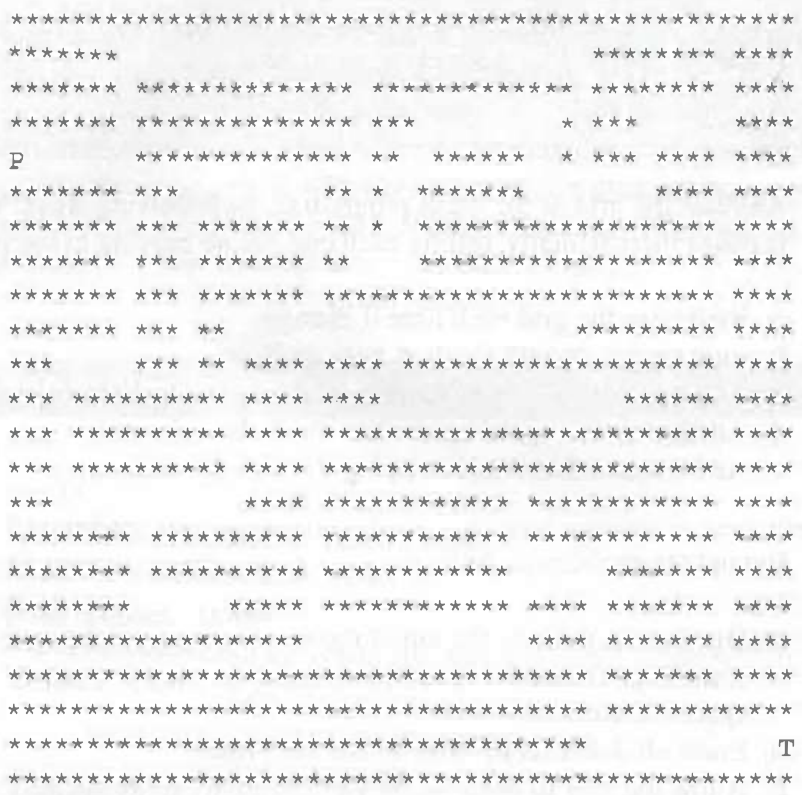

Figure 7.9 A maze problem

Java source code for the program's interface can be found on the CD together with executable code for the program as a whole. You should run the program a few times before writing your own solution. The interface allows the user to enter a picture of the maze as a grid of characters. The character * marks a barrier, and P and T mark the parking lot and mountain top, respectively. A blank space marks a step along a path. The interface includes a Solve button, and when the user selects it, the view should send a message to the model with the contents of the grid as a string parameter. The model then attempts to find a path through the maze and returns "solved" or "unsolved" to the view depending on the outcome. In the model, begin by representing the maze as a matrix of characters (P, T, *, or space) and during the search mark

each visited cell with a dot. Redisplay the grid at the end with the dots included. Here is the backtracking algorithm that is at the core of the solution:

```
Instantiate a stack
Locate the character 'P' in the matrix
Push its location onto the stack
While the stack is not empty
   Pop a location, loc, off the stack
   If the matrix contains 'T' at position loc then
      A path has been found
      Break
   Endif
   Store a dot in the matrix at position loc
   Examine the cells of the matrix adjacent to loc and for each one that
     contains a space, push its location onto the stack
Endwhile
There is no solution
```

7. Animate the grid of the maze program in the following ways. You should add these features incrementally, testing each one before moving to the next:

a. Redisplay the grid each time it changes.
b. Indicate the current location with an X.
c. Add a button to restore the map and run the algorithm again.
d. Add buttons to speed up or slow down the animation.
e. Add a button for single stepping through the animation.

8. Extend the problem as follows:

a. Display the route to the top of the mountain as a string in the form F4RF2LF15 (meaning "forward 4 spaces, turn right, forward 2 spaces, turn left, forward 15 spaces"), etc.
b. Erase all dots except those on the final route.
c. Allow the user to place Ms at various points along the paths, where M stands for a mushroom. The program should find all the mushrooms, pick them, and retrace its steps back to the parking lot. Animate the program so that the user can follow the course of the program.

9. Compute the time complexity of all algorithms involved in the maze program.