Chapter Six

Stacks, Queues, Recursion

6.1 INTRODUCTION

The linear lists and linear arrays discussed in the previous chapters allowed one to insert and delete elements at any place in the list—at the beginning, at the end, or in the middle. There are certain frequent situations in computer science when one wants to restrict insertions and deletions so that they can take place only at the beginning or the end of the list, not in the middle. Two of the data structures that are useful in such situations are *stacks* and *queues*.

A stack is a linear structure in which items may be added or removed only at one end. Figure 6.1 pictures three everyday examples of such a structure: a stack of dishes, a stack of pennies and a stack of folded towels. Observe that an item may be added or removed only from the top of any of the stacks. This means, in particular, that the last item to be added to a stack is the first item to be removed. Accordingly, stacks are also called last-in first-out (LIFO) lists. Other names used for stacks are "piles" and "push-down lists." Although the stack may seem to be a very restricted type of data structure, it has many important applications in computer science.

A queue is a linear list in which items may be added only at one end and items may be removed only at the other end. The name "queue" likely comes from the everyday use of the term. Consider a queue of people waiting at a bus stop, as pictured in Fig. 6.2. Each new person who comes takes his or her place at the end of the line, and when the bus comes, the people at the front of the line board first. Clearly, the first person in the line is the first person to leave. Thus queues are also called first-in first-out (FIFO) lists. Another example of a queue is a batch of jobs waiting to be processed, assuming no job has higher priority than the others.

The notion of *recursion* is fundamental in computer science. This topic is introduced in this chapter because one way of simulating recursion is by means of a stack structure.

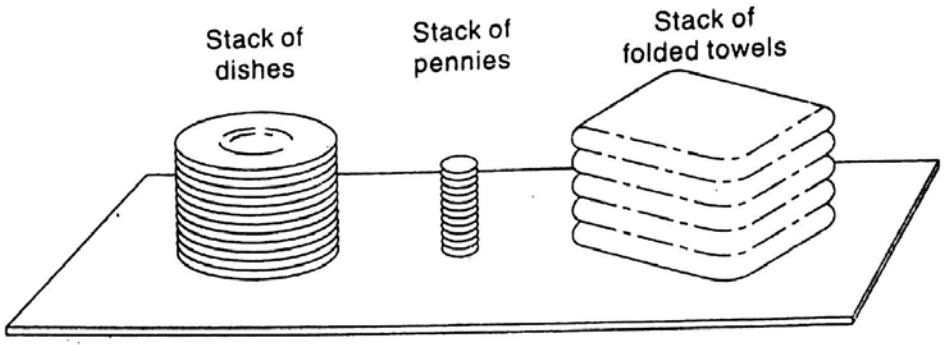


Fig. 6.1

Fig. 6.2 *Queue Waiting for a Bus*

6.2 STACKS

A *stack* is a list of elements in which an element may be inserted or deleted only at one end, called the *top* of the stack. This means, in particular, that elements are removed from a stack in the reverse order of that in which they were inserted into the stack.

Special terminology is used for two basic operations associated with stacks:

(a) "Push" is the term used to insert an element into a stack.
(b) "Pop" is the term used to delete an element from a stack.

We emphasize that these terms are used only with stacks, not with other data structures.

Example 6.1

Suppose the following 6 elements are pushed, in order, onto an empty stack:

AAA, BBB, CCC, DDD, EEE, FFF

Figure 6.3 shows three ways of picturing such a stack. For notational convenience, we will frequently designate the stack by writing:

STACK: AAA, BBB, CCC, DDD, EEE, FFF

The implication is that the right-most element is the top element. We emphasize that, regardless of the way a stack is described, its underlying property is that insertions and deletions can occur only at the top of the stack. This means EEE cannot be deleted before FFF is deleted, DDD cannot be deleted before EEE and FFF are deleted, and so on. Consequently, the elements may be popped from the stack only in the reverse order of that in which they were pushed onto the stack.

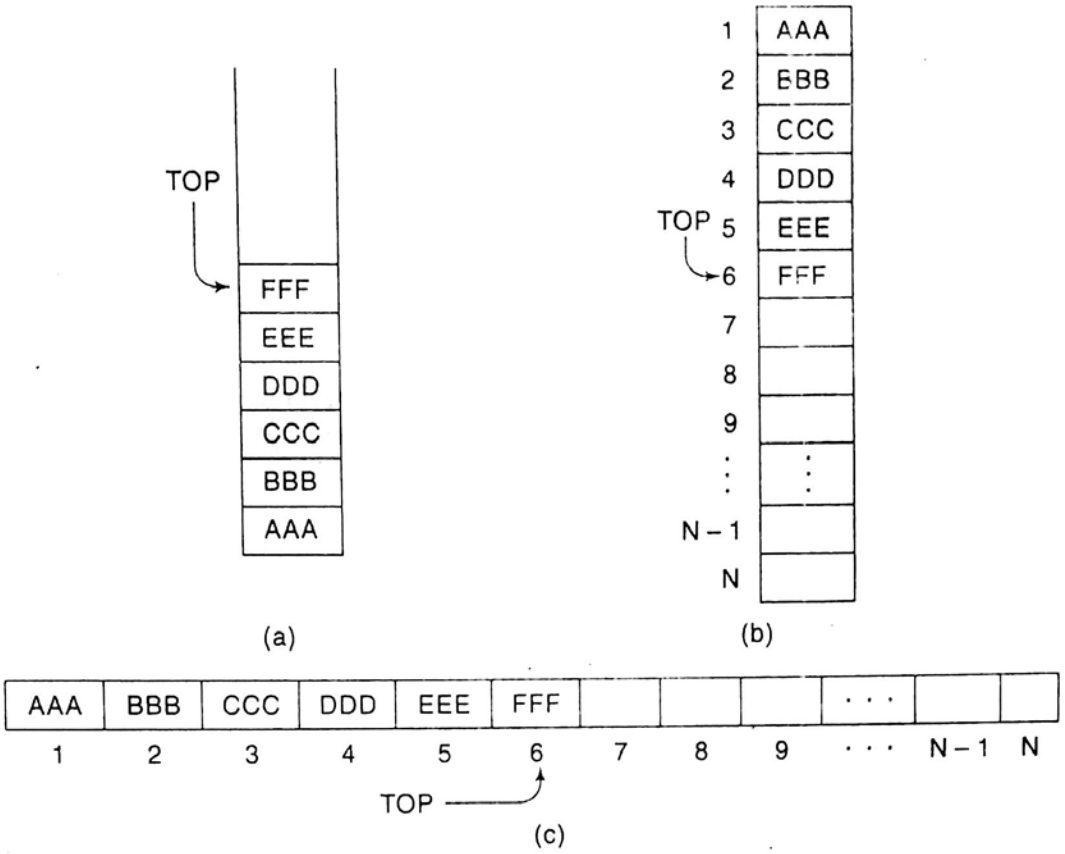


Fig. 6.3 *Diagrams of Stacks*

Consider again the AVAIL list of available nodes discussed in Chapter 5. Recall that free nodes were removed only from the beginning of the AVAIL list, and that new available nodes were inserted only at the beginning of the AVAIL list. In other words, the AVAIL list was implemented as a stack. This implementation of the AVAIL list as a stack is only a matter of convenience rather than an inherent part of the structure. In the following subsection we discuss an important situation where the stack is an essential tool of the processing algorithm itself.

Postponed Decisions

Stacks are frequently used to indicate the order of the processing of data when certain steps of the processing must be postponed until other conditions are fulfilled. This is illustrated as follows.

Suppose that while processing some project A we are required to move on to project B, whose completion is required in order to complete project A. Then we place the folder containing the data of A onto a stack, as pictured in Fig. 6.4(a), and begin to process B. However, suppose that while processing B we are led to project C, for the same reason. Then we place B on the stack above A, as pictured in Fig. 6.4(b), and begin to process C. Furthermore, suppose that while processing C we are likewise led to project D. Then we place C on the stack above B, as pictured in Fig. 6.4(c), and begin to process D.

On the other hand, suppose we are able to complete the processing of project D. Then the only project we may continue to process is project C, which is on top of the stack. Hence we remove folder C from the stack, leaving the stack as pictured in Fig. 6.4(d), and continue to process C.

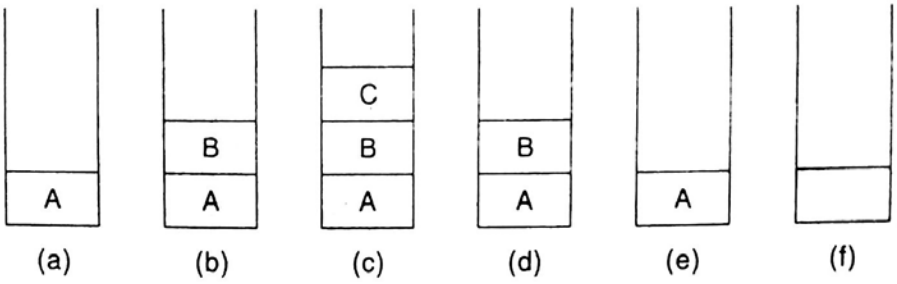


Fig. 6.4

Similarly, after completing the processing of C, we remove folder B from the stack, leaving the stack as pictured in Fig. 6.4(e), and continue to process B. Finally, after completing the processing of B, we remove the last folder, A, from the stack, leaving the empty stack pictured in Fig. 6.4(f), and continue the processing of our original project A.

Observe that, at each stage of the above processing, the stack automatically maintains the order that is required to complete the processing. An important example of such a processing in computer science is where A is a main program and B, C and D are subprograms called in the order given.

6.3 ARRAY REPRESENTATION OF STACKS

Stacks may be represented in the computer in various ways, usually by means of a one-way list or a linear array. Unless otherwise stated or implied, each of our stacks will be maintained by a linear array STACK; a pointer variable TOP, which contains the location of the top element of the stack; and a variable MAXSTK which gives the maximum number of elements that can be held by the stack. The condition TOP = 0 or TOP = NULL will indicate that the stack is empty.

Figure 6.5 pictures such an array representation of a stack. (For notational convenience, the array is drawn horizontally rather than vertically.) Since TOP = 3, the stack has three elements, XXX, YYY and ZZZ; and since MAXSTK = 8, there is room for 5 more items in the stack.

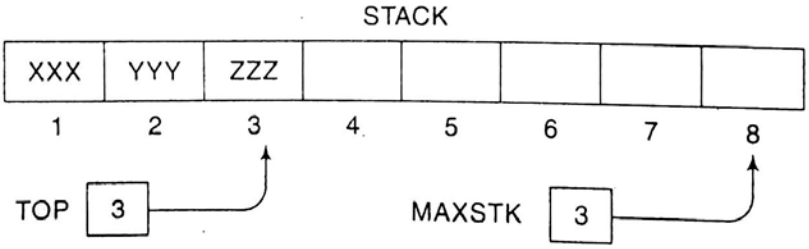


Fig. 6.5

The operation of adding (pushing) an item onto a stack and the operation of removing (popping) an item from a stack may be implemented, respectively, by the following procedures, called PUSH and POP. In executing the procedure PUSH, one must first test whether there is room in the stack for the new item; if not, then we have the condition known as overflow. Analogously, in executing the procedure POP, one must first test whether there is an element in the stack to be deleted; if not, then we have the condition known as underflow.

Procedure 6.1: PUSH(STACK, TOP, MAXSTK, ITEM)
This procedure pushes an ITEM onto a stack.

1. [Stack already filled?]
   If TOP = MAXSTK, then: Print: OVERFLOW, and Return.
2. Set TOP := TOP + 1. [Increases TOP by 1.]
3. Set STACK[TOP] := ITEM. [Inserts ITEM in new TOP position.]
4. Return.

Procedure 6.2: POP(STACK, TOP, ITEM)
This procedure deletes the top element of STACK and assigns it to the variable ITEM.

1. [Stack has an item to be removed?]
   If TOP = 0, then: Print: UNDERFLOW, and Return.
2. Set ITEM := STACK[TOP]. [Assigns TOP element to ITEM.]
3. Set TOP := TOP – 1. [Decreases TOP by 1.]
4. Return.

Frequently, TOP and MAXSTK are global variables; hence the procedures may be called using only

PUSH(STACK, ITEM) and POP(STACK, ITEM)

respectively. We note that the value of TOP is changed before the insertion in PUSH but the value of TOP is changed after the deletion in POP.

Example 6.2

(a) Consider the stack in Fig. 6.5. We simulate the operation PUSH(STACK, WWW):

1. Since TOP = 3, control is transferred to Step 2.
2. TOP = 3 + 1 = 4.
3. STACK[TOP] = STACK[4] = WWW.
4. Return.

Note that WWW is now the top element in the stack.

(b) Consider again the stack in Fig. 6.5. This time we simulate the operation POP(STACK, ITEM):

1. Since TOP = 3, control is transferred to Step 2.
2. ITEM = ZZZ.
3. TOP = 3 – 1 = 2.
4. Return.

Observe that STACK[TOP] = STACK[2] = YYY is now the top element in the stack.

Minimizing Overflow

There is an essential difference between underflow and overflow in dealing with stacks. Underflow depends exclusively upon the given algorithm and the given input data, and hence there is no direct control by the programmer. Overflow, on the other hand, depends upon the arbitrary choice of the programmer for the amount of memory space reserved for each stack, and this choice does influence the number of times overflow may occur.

Generally speaking, the number of elements in a stack fluctuates as elements are added to or removed from a stack. Accordingly, the particular choice of the amount of memory for a given stack involves a time-space tradeoff. Specifically, initially reserving a great deal of space for each stack will decrease the number of times overflow may occur; however, this may be an expensive use of the space if most of the space is seldom used. On the other hand, reserving a small amount of space for each stack may increase the number of times overflow occurs; and the time required for resolving an overflow, such as by adding space to the stack, may be more expensive than the space saved.

Various techniques have been developed which modify the array representation of stacks so that the amount of space reserved for more than one stack may be more efficiently used. Most of these techniques lie beyond the scope of this text. We do illustrate one such technique in the following example.

Example 6.3

Suppose a given algorithm requires two stacks, A and B. One can define an array STACKA with n_1 elements for stack A and an array STACKB with n_2 elements for stack B. Overflow will occur when either stack A contains more than n_1 elements or stack B contains more than n_2 elements.

Suppose instead that we define a single array STACK with $n = n_1 + n_2$ elements for stacks A and B together. As pictured in Fig. 6.6, we define STACK[1] as the bottom of stack A and let A "grow" to the right, and we define STACK[n] as the bottom of stack B and let B "grow" to the left. In this case, overflow will occur only when A and B together have more than $n = n_1 + n_2$ elements. This technique will usually decrease the number of times overflow occurs even though we have not increased the total amount of space reserved for the two stacks. In using this data structure, the operations of PUSH and POP will need to be modified.

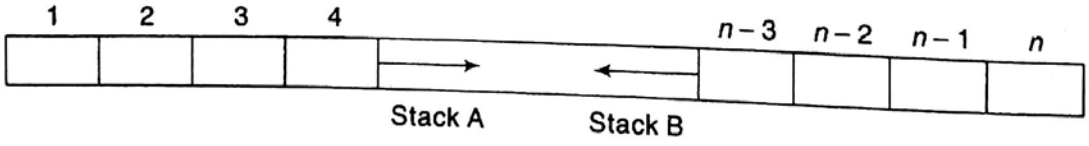


Fig. 6.6

6.4 LINKED REPRESENTATION OF STACKS

In this section we discuss the representation of stacks using a one-way list or singly linked list. The advantages of linked lists over arrays have already been underlined. It is in this perspective that one appreciates the use of a linked list for stacks in comparison to that of arrays.

The linked representation of a stack, commonly termed linked stack is a stack that is implemented using a singly linked list. The INFO fields of the nodes hold the elements of the stack and the LINK fields hold pointers to the neighboring element in the stack. The START pointer of the linked list behaves as the TOP pointer variable of the stack and the null pointer of the last node in the list signals the bottom of stack. Figure 6.7 illustrates the linked representation of the stack STACK shown in Fig. 6.5.

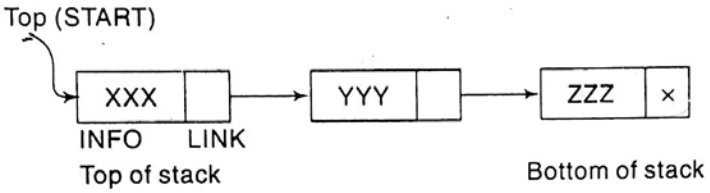


Fig. 6.7

A push operation into STACK is accomplished by inserting a node into the front or start of the list and a pop operation is undertaken by deleting the node pointed to by the START pointer. Figure 6.8 and Fig. 6.9 illustrate the push and pop operation on the linked stack STACK shown in Fig. 6.7.

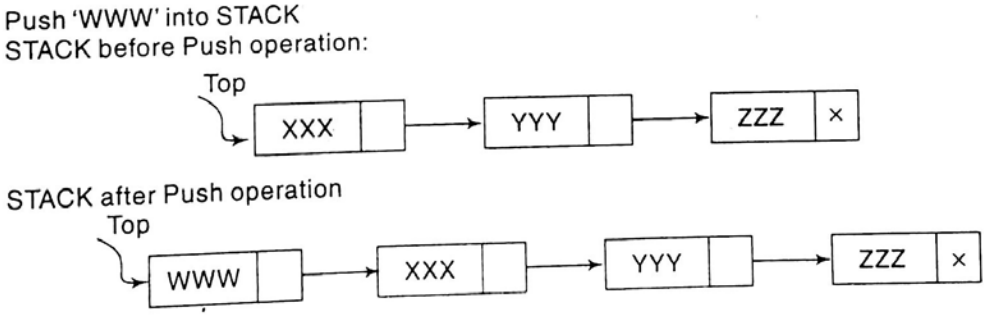


Fig. 6.8

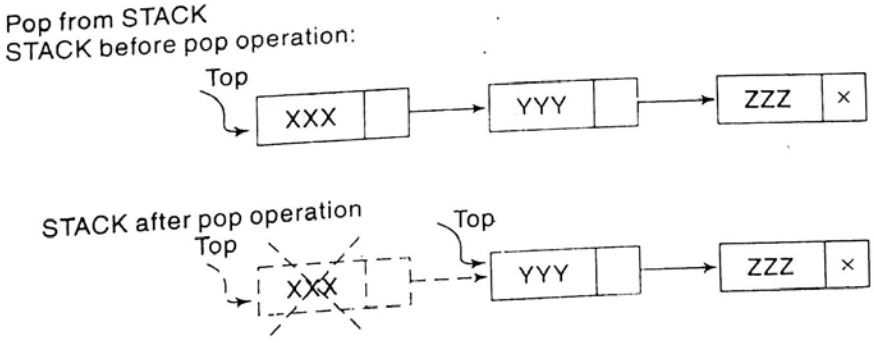


Fig. 6.9

The array representation of stack calls for the maintenance of a variable MAXSTK which gives the maximum number of elements that can be held by the stack. Also, it calls for the checking of OVERFLOW in the case of push operation (TOP=MAXSTK) and UNDERFLOW in the case of pop operation (TOP=0). In contrast, the linked representation of stacks is free of these requirements. There is no limitation on the capacity of the linked stack and hence it can support as many push operations (insertion of nodes) as the free-storage list (the AVAIL list) can support. This dispenses with the need to maintain the MAXSTK variable and consequently on the checking of OVERFLOW of the linked stack during a push operation.

Procedure 6.3: PUSH_LINKSTACK(INFO, LINK, TOP, AVAIL, ITEM)
This procedure pushes an ITEM into a linked stack

1. [Available space?] If AVAIL = NULL, then Write OVERFLOW and Exit
2. [Remove first node from AVAIL list]
   Set NEW := AVAIL and AVAIL := LINK[AVAIL].
3. Set INFO[NEW] := ITEM [Copies ITEM into new node]
4. Set LINK[NEW] := TOP [New node points to the original top node in the stack]
5. Set TOP = NEW [Reset TOP to point to the new node at the top of the stack]
6. Exit.

Procedure 6.4: POP_LINKSTACK(INFO, LINK, TOP, AVAIL, ITEM)
This procedure deletes the top element of a linked stack and assigns it to the variable ITEM

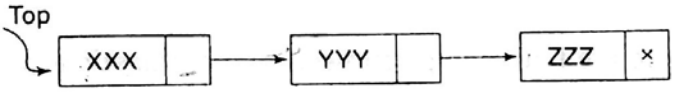
1. [Stack has an item to be removed?]
   IF TOP = NULL then Write: UNDERFLOW and Exit.
2. Set ITEM := INFO[TOP] [Copies the top element of stack into ITEM]
3. Set TEMP := TOP and TOP = LINK[TOP]
   [Remember the old value of the TOP pointer in TEMP and reset TOP to point to the next element in the stack]
4. [Return deleted node to the AVAIL list]
   Set LINK[TEMP] = AVAIL and AVAIL = TEMP.
5. Exit.

However, the condition TOP = NULL may be retained in the pop procedure to prevent deletion from an empty linked stack and the condition AVAIL = NULL to check for available space in the free-storage list.

Example 6.4

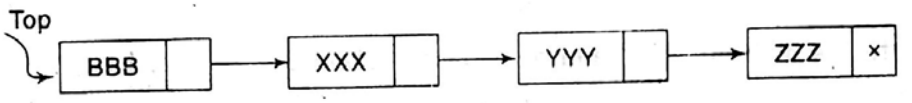
Consider the linked stack shown in Fig. 6.7, the snapshots of the stack structure on execution of the following operations are shown in Fig. 6.10:

(i) Push BBB (ii) Pop (iii) Pop (iv) Push MMM
Original linked stack:



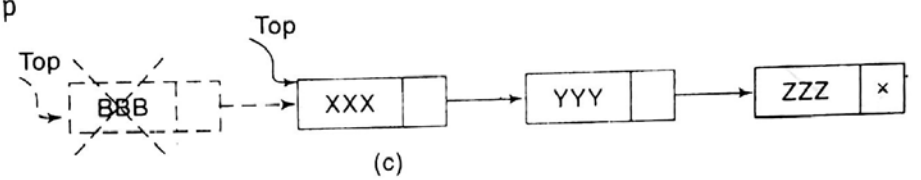
(a)

(i) Push BBB



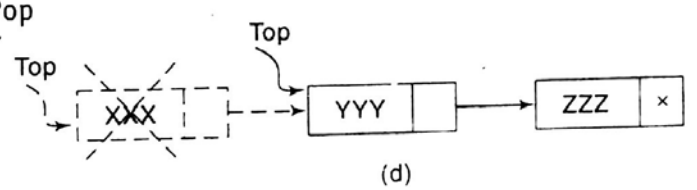
(b)

(ii) Pop



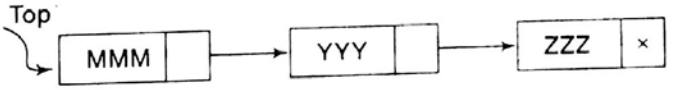
(c)

(iii) Pop



(d)

(iv) Push MMM



(e)

Fig. 6.10

6.5 ARITHMETIC EXPRESSIONS; POLISH NOTATION

Let Q be an arithmetic expression involving constants and operations. This section gives an algorithm which finds the value of Q by using reverse Polish (postfix) notation. We will see that the stack is an essential tool in this algorithm.

Recall that the binary operations in Q may have different levels of precedence. Specifically, we assume the following three levels of precedence for the usual five binary operations:

Highest:	Exponentiation (↑)
Next highest:	Multiplication (*) and division (/)
Lowest:	Addition (+) and subtraction (–)

(Observe that we use the BASIC symbol for exponentiation.) For simplicity, we assume that Q contains no unary operation (e.g., a leading minus sign). We also assume that in any parenthesis-free expression, the operations on the same level are performed from left to right. (This is not standard, since some languages perform exponentiations from right to left.)

Example 6.5

Suppose we want to evaluate the following parenthesis-free arithmetic expression:

$$2 \uparrow 3 + 5 * 2 \uparrow 2 - 12 / 6$$

First we evaluate the exponentiations to obtain

$$8 + 5 * 4 - 12 / 6$$

Then we evaluate the multiplication and division to obtain 8 + 20 – 2. Last, we evaluate the addition and subtraction to obtain the final result, 26. Observe that the expression is traversed three times, each time corresponding to a level of precedence of the operations.

Polish Notation

For most common arithmetic operations, the operator symbol is placed between its two operands. For example,

$$A + B \qquad C - D \qquad E * F \qquad G/H$$

This is called *infix notation*. With this notation, we must distinguish between

$$(A + B) * C \qquad \text{and} \qquad A + (B * C)$$

by using either parentheses or some operator-precedence convention such as the usual precedence levels discussed above. Accordingly, the order of the operators and operands in an arithmetic expression does not uniquely determine the order in which the operations are to be performed.

Polish notation, named after the Polish mathematician Jan Lukasiewicz, refers to the notation in which the operator symbol is placed before its two operands. For example,

$$+AB \qquad -CD \qquad *EF \qquad /GH$$

We translate, step by step, the following infix expressions into Polish notation using brackets [ ] to indicate a partial translation:

$$(A + B) * C = [+AB] * C = *+ ABC$$
$$A + (B * C) = A + [*BC) = + A*BC$$
$$(A + B)/(C - D) = [+AB]/[-CD] = / + AB - CD$$

The fundamental property of Polish notation is that the order in which the operations are to be performed is completely determined by the positions of the operators and operands in the expression. Accordingly, one never needs parentheses when writing expressions in Polish notation.

Reverse Polish notation refers to the analogous notation in which the operator symbol is placed after its two operands:

$$AB+ \quad CD- \quad EF* \quad GH/$$

Again, one never needs parentheses to determine the order of the operations in any arithmetic expression written in reverse Polish notation. This notation is frequently called *postfix* (or *suffix*) *notation*, whereas *prefix notation* is the term used for Polish notation, discussed in the preceding paragraph.

The computer usually evaluates an arithmetic expression written in infix notation in two steps. First, it converts the expression to postfix notation, and then it evaluates the postfix expression. In each step, the stack is the main tool that is used to accomplish the given task. We illustrate these applications of stacks in reverse order. That is, first we show how stacks are used to evaluate postfix expressions, and then we show how stacks are used to transform infix expressions into postfix expressions.

Evaluation of a Postfix Expression

Suppose P is an arithmetic expression written in postfix notation. The following algorithm, which uses a STACK to hold operands, evaluates P.

Algorithm 6.5: This algorithm finds the VALUE of an arithmetic expression P written in postfix notation.

1. Add a right parenthesis ")" at the end of P. [This acts as a sentinel.]
2. Scan P from left to right and repeat Steps 3 and 4 for each element of P until the sentinel ")" is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator $\otimes$ is encountered, then:
   (a) Remove the two top elements of STACK, where A is the top element and B is the next-to-top element.
   (b) Evaluate B $\otimes$ A.
   (c) Place the result of (b) back on STACK.
   [End of If structure.]
   [End of Step 2 loop.]
5. Set VALUE equal to the top element on STACK.
6. Exit.

We note that, when Step 5 is executed, there should be only one number on STACK.

Example 6.6

Consider the following arithmetic expression P written in postfix notation:

P: 5, 6, 2, +, *, 12, 4, /, -

(Commas are used to separate the elements of P so that 5, 6, 2 is not interpreted as the number 562.) The equivalent infix expression Q follows:

Q: 5 * (6 + 2) - 12 / 4

Note that parentheses are necessary for the infix expression Q but not for the postfix expression P.

We evaluate P by simulating Algorithm 6.5. First we add a sentinel right parenthesis at the end of P to obtain

P:	5,	6,	2,	+,	*,	12,	4,	/,	-,	)
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)

The elements of P have been labeled from left to right for easy reference. Figure 6.11 shows the contents of STACK as each element of P is scanned. The final number in STACK, 37, which is assigned to VALUE when the sentinel ")" is scanned, is the value of P.

Symbol Scanned		STACK
(1)	5	5
(2)	6	5, 6
(3)	2	5, 6, 2
(4)	+	5, 8
(5)	*	40
(6)	12	40, 12
(7)	4	40, 12, 4
(8)	/	40, 3
(9)	–	37
(10)	)	

Fig. 6.11

Transforming Infix Expressions into Postfix Expressions

Let Q be an arithmetic expression written in infix notation. Besides operands and operators, Q may also contain left and right parentheses. We assume that the operators in Q consist only of exponentiations (↑), multiplications (*), divisions (/), additions (+) and subtractions (–), and that they have the usual three levels of precedence as given above. We also assume that operators on the same level, including exponentiations, are performed from left to right unless otherwise indicated by parentheses. (This is not standard, since expressions may contain unary operators and some languages perform the exponentiations from right to left. However, these assumptions simplify our algorithm.)

The following algorithm transforms the infix expression Q into its equivalent postfix expression P. The algorithm uses a stack to temporarily hold operators and left parentheses. The postfix expression P will be constructed from left to right using the operands from Q and the operators which are removed from STACK. We begin by pushing a left parenthesis onto STACK and adding a right parenthesis at the end of Q. The algorithm is completed when STACK is empty.

Algorithm 6.6: POLISH(Q, P)

Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

1. Push "(" onto STACK, and add ")" to the end of Q.
2. Scan Q from left to right and repeat Steps 3 to 6 for each element of Q until the STACK is empty:
3. If an operand is encountered, add it to P.
4. If a left parenthesis is encountered, push it onto STACK.
5. If an operator ⊗ is encountered, then:

(a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has the same precedence as or higher precedence than ⊗.
(b) Add ⊗ to STACK.
[End of If structure.]

6. If a right parenthesis is encountered, then:
(a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) until a left parenthesis is encountered.
(b) Remove the left parenthesis. [Do not add the left parenthesis to P.]
[End of If structure.]
[End of Step 2 loop.]

7. Exit.

The terminology sometimes used for Step 5 is that ⊗ will "sink" to its own level.

Example 6.7

Consider the following arithmetic infix expression Q:

Q: A + (B * C - (D / E ↑ F) * G) * H

We simulate Algorithm 6.6 to transform Q into its equivalent postfix expression P. First we push "(" onto STACK, and then we add ")" to the end of Q to obtain:

Q:	A	+	(	B	*	C	-	(	D	/	E	↑	F	)	*
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)
											G	)	*	H	)
											(16)	(17)	(18)	(19)	(20)

The elements of Q have now been labeled from left to right for easy reference. Figure 6.12 shows the status of STACK and of the string P as each element of Q is scanned. Observe that

(1) Each operand is simply added to P and does not change STACK.
(2) The subtraction operator (-) in row 7 sends * from STACK to P before it (-) is pushed onto STACK.
(3) The right parenthesis in row 14 sends ↑ and then / from STACK to P, and then removes the left parenthesis from the top of STACK.
(4) The right parenthesis in row 20 sends * and then + from STACK to P, and then removes the left parenthesis from the top of STACK.

After Step 20 is executed, the STACK is empty and

P: A B C * D E F ↑ / G * - H * +

which is the required postfix equivalent of Q.

Symbol Scanned		STACK	Expression P
(1)	A	(	A
(2)	+	(+	A
(3)	(	(+ (	A
(4)	B	(+ (	A B
(5)	*	(+ (*	A B
(6)	C	(+ (*	A B C
(7)	−	(+ (−	A B C *
(8)	(	(+ (− (	A B C *
(9)	D	(+ (− (	A B C * D
(10)	/	(+ (− (/	A B C * D
(11)	E	(+ (− (/	A B C * D E
(12)	↑	(+ (− (/ ↑	A B C * D E
(13)	F	(+ (− (/ ↑	A B C * D E F
(14)	)	(+ (−	A B C * D E F ↑ /
(15)	*	(+ (− *	A B C * D E F ↑ /
(16)	G	(+ (− *	A B C * D E F ↑ / G
(17)	)	(+	A B C * D E F ↑ / G * −
(18)	*	(+ *	A B C * D E F ↑ / G * −
(19)	H	(+ *	A B C * D E F ↑ / G * − H
(20)	)		A B C * D E F ↑ / G * − H * +

Fig. 6.12

6.6 QUICKSORT, AN APPLICATION OF STACKS

Let A be a list of n data items. "Sorting A" refers to the operation of rearranging the elements of A so that they are in some logical order, such as numerically ordered when A contains numerical data, or alphabetically ordered when A contains character data. The subject of sorting, including various sorting algorithms, is treated mainly in Chapter 9. This section gives only one sorting algorithm, called *quicksort*, in order to illustrate an application of stacks.

Quicksort is an algorithm of the divide-and-conquer type. That is, the problem of sorting a set is reduced to the problem of sorting two smaller sets. We illustrate this "reduction step" by means of a specific example.

Suppose A is the following list of 12 numbers:

(44) 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, (66)

The reduction step of the quicksort algorithm finds the final position of one of the numbers; in this illustration, we use the first number, 44. This is accomplished as follows. Beginning with the last number, 66, scan the list from right to left, comparing each number with 44 and stopping at the first number less than 44. The number is 22. Interchange 44 and 22 to obtain the list

(22) 33, 11, 55, 77, 90, 40, 60, 99, (44) 88, 66

(Observe that the numbers 88 and 66 to the right of 44 are each greater than 44.) Beginning with 22, next scan the list in the opposite direction, from left to right, comparing each number with 44 and stopping at the first number greater than 44. The number is 55. Interchange 44 and 55 to obtain the list

22, 33, 11, (44), 77, 90, 40, 60, 99, (55), 88, 66

(Observe that the numbers 22, 33 and 11 to the left of 44 are each less than 44.) Beginning this time with 55, now scan the list in the original direction, from right to left, until meeting the first number less than 44. It is 40. Interchange 44 and 40 to obtain the list

22, 33, 11, (40), 77, 90, (44), 60, 99, 55, 88, 66

(Again, the numbers to the right of 44 are each greater than 44.) Beginning with 40, scan the list from left to right. The first number greater than 44 is 77. Interchange 44 and 77 to obtain the list

22, 33, 11, (40), (44), 90, (77), 60, 99, 55, 88, 66

(Again, the numbers to the left of 44 are each less than 44.) Beginning with 77, scan the list from right to left seeking a number less than 44. We do not meet such a number before meeting 44. This means all numbers have been scanned and compared with 44. Furthermore, all numbers less than 44 now form the sublist of numbers to the left of 44, and all numbers greater than 44 now form the sublist of numbers to the right of 44, as shown below:

22, 33, 11, 40, (44), 90, 77, 60, 99, 55, 88, 66

First sublist Second sublist

Thus 44 is correctly placed in its final position, and the task of sorting the original list A has now been reduced to the task of sorting each of the above sublists.

The above reduction step is repeated with each sublist containing 2 or more elements. Since we can process only one sublist at a time, we must be able to keep track of some sublists for future processing. This is accomplished by using two stacks, called LOWER and UPPER, to temporarily "hold" such sublists. That is, the addresses of the first and last elements of each sublist, called its *boundary values*, are pushed onto the stacks LOWER and UPPER, respectively; and the reduction step is applied to a sublist only after its boundary values are removed from the stacks. The following example illustrates the way the stacks LOWER and UPPER are used.

Example 6.8

Consider the above list A with $n = 12$ elements. The algorithm begins by pushing the boundary values 1 and 12 of A onto the stacks to yield

LOWER: 1 UPPER: 12

In order to apply the reduction step, the algorithm first removes the top values 1 and 12 from the stacks, leaving

LOWER: (empty) UPPER: (empty)

and then applies the reduction step to the corresponding list A[1], A[2], ..., A[12]. The reduction step, as executed above, finally places the first element, 44, in A[5]. Accordingly, the algorithm pushes the boundary values 1 and 4 of the first sublist and the boundary values 6 and 12 of the second sublist onto the stacks to yield

LOWER: 1, 6 UPPER: 4, 12