Chapter 6

Stacks, Queues, Recursion

*Difference betn Queues and stack.

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.

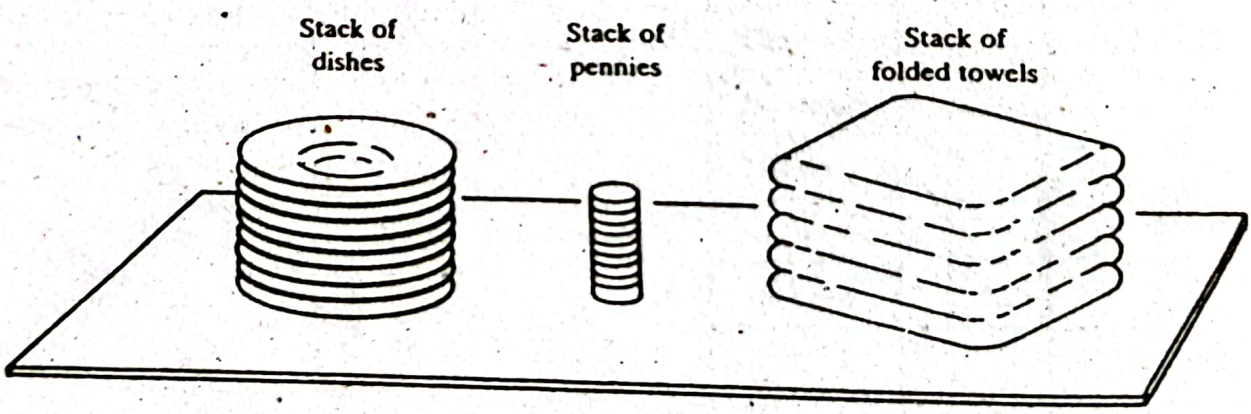


Fig. 6-1

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.

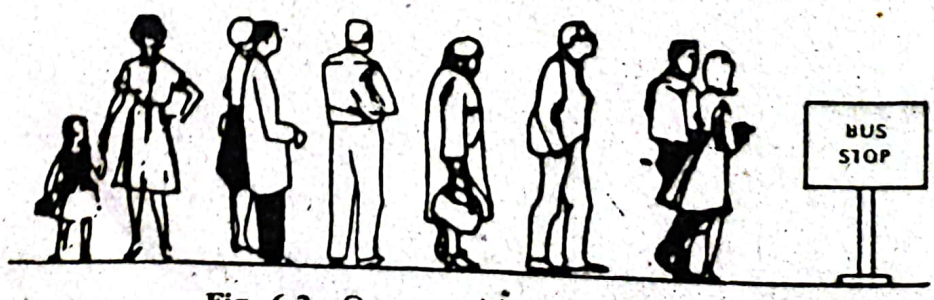


Fig. 6-2 Queue waiting for a bus.

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.

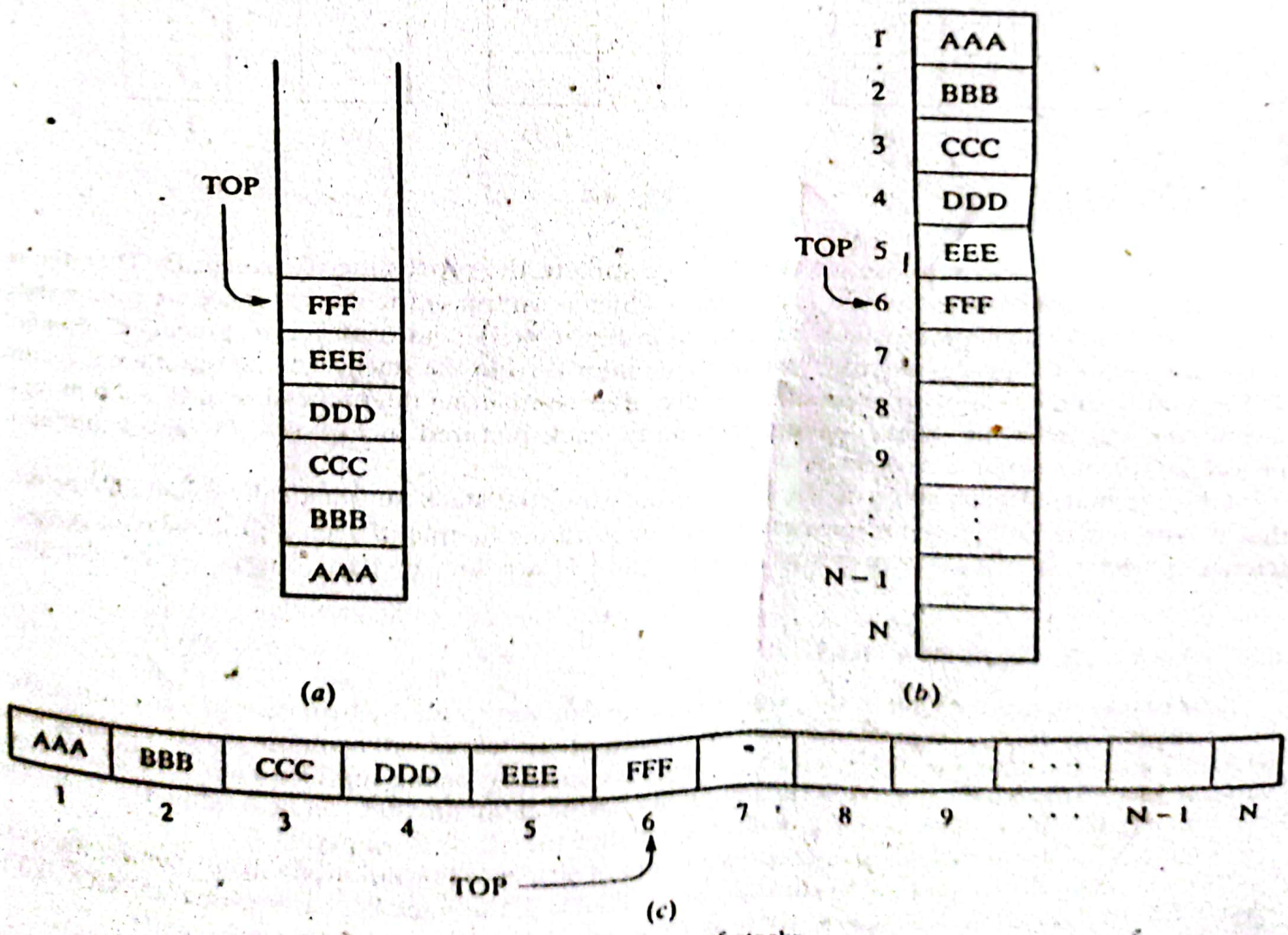


Fig. 6-3 Diagrams of stacks.

Consider again the AVAIL list of available nodes discussed in Chap. 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.

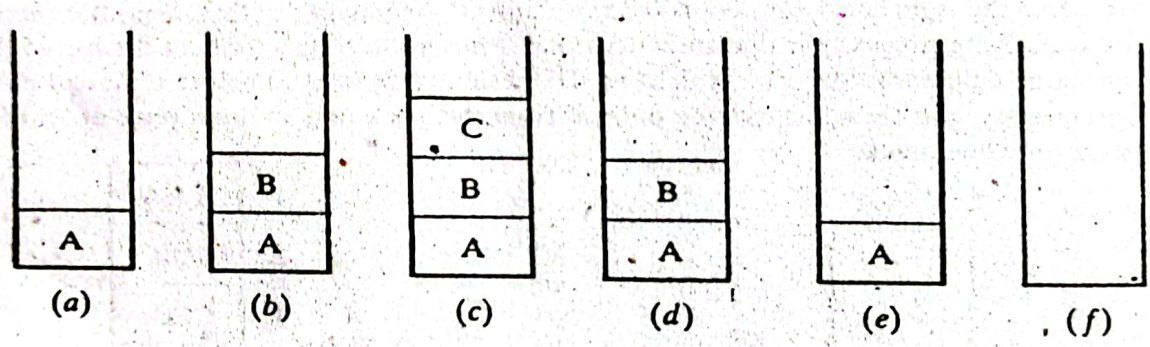


Fig. 6-4

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. 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.

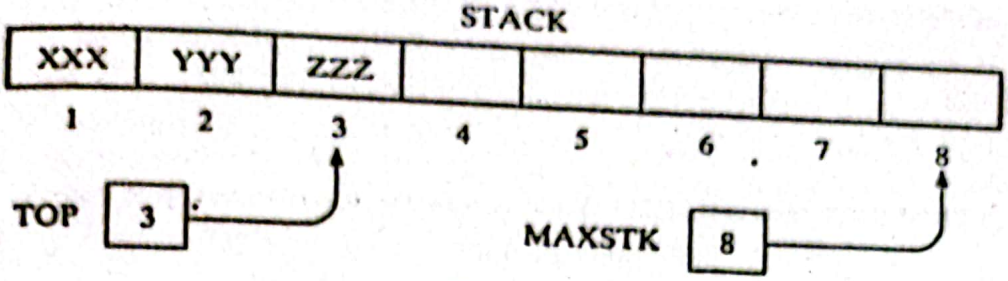


Fig. 6-5

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.

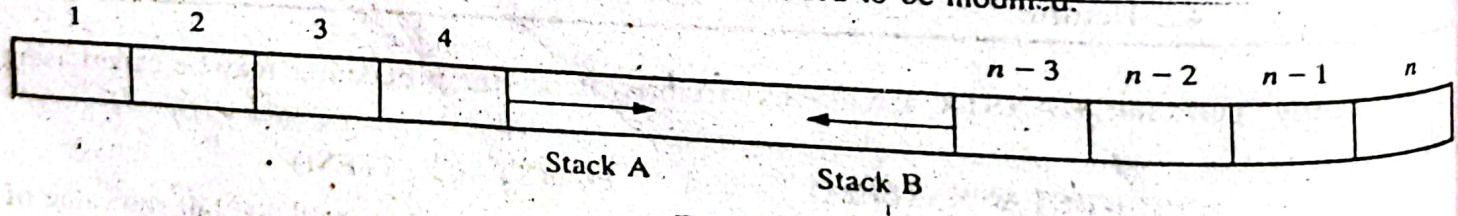


Fig. 6-6

6.4 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 6.4

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

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

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:

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

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 6.3: 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 6.5

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

(5+6)↑2−8*9+3
↓
postfix G convert

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. 6-7

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 6.3. 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 6-7 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.

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 6.4: 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 6.6

Consider the following arithmetic infix expression Q:

Q: A + (B * C - (D / E ↑ F) * G) * H

We simulate Algorithm 6.4 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	)	*	G	)	*	H	)
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)	(19)	(20)

The elements of Q have now been labeled from left to right for easy reference. Figure 6-8 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. 6-8

6.5 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 Chap. 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 6.7

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

In order to apply the reduction step again, the algorithm removes the top values, 6 and 12, from the stacks, leaving

LOWER: 1 UPPER: 4

and then applies the reduction step to the corresponding sublist A[6], A[7], . . . , A[12]. The reduction step changes this list as in Fig. 6-9. Observe that the second sublist has only one element. Accordingly, the algorithm pushes only the boundary values 6 and 10 of the first sublist onto the stacks to yield

LOWER: 1, 6 UPPER: 4, 10

And so on. The algorithm ends when the stacks do not contain any sublist to be processed by the reduction step.

A[6],	A[7],	A[8],	A[9],	A[10],	A[11],	A[12],
(90),	77,	60,	99,	55,	88,	(66)
66,	77,	60,	(99),	55,	88,	(90)
66,	77,	60,	(90),	55,	(88),	99
66,	77,	60,	88,	55,	(90),	99
			First sublist		Second sublist	

Fig. 6-9

The formal statement of our quicksort algorithm follows (on page 175). For notational convenience and pedagogical considerations, the algorithm is divided into two parts. The first part gives a procedure, called QUICK, which executes the above reduction step of the algorithm, and the second part uses QUICK to sort the entire list.

Observe that Step 2(*c*) (iii) is unnecessary. It has been added to emphasize the symmetry between Step 2 and Step 3. The procedure does not assume the elements of A are distinct. Otherwise, the condition LOC ≠ RIGHT in Step 2(*a*) and the condition LEFT ≠ LOC in Step 3(*a*) could be omitted.

The second part of the algorithm follows (on page 175). As noted above, LOWER and UPPER are stacks on which the boundary values of the sublists are stored. (As usual, we use NULL = 0.)

Procedure 6.5: QUICK(A, N, BEG, END, LOC)

Here A is an array with N elements. Parameters BEG and END contain the boundary values of the sublist of A to which this procedure applies. LOC keeps track of the position of the first element A[BEG] of the sublist during the procedure. The local variables LEFT and RIGHT will contain the boundary values of the list of elements that have not been scanned.

1. [Initialize.] Set LEFT := BEG, RIGHT := END and LOC := BEG.
2. [Scan from right to left.]
   (*a*) Repeat while A[LOC] ≤ A[RIGHT] and LOC ≠ RIGHT:
   RIGHT := RIGHT − 1.
   [End of loop.]
   (*b*) If LOC = RIGHT, then: Return.
   (*c*) If A[LOC] > A[RIGHT], then:
   (i) [Interchange A[LOC] and A[RIGHT].]
   TEMP := A[LOC], A[LOC] := A[RIGHT],
   A[RIGHT] := TEMP.
   (ii) Set LOC := RIGHT.
   (iii) Go to Step 3.
   [End of If structure.]
3. [Scan from left to right.]
   (*a*) Repeat while A[LEFT] ≤ A[LOC] and LEFT ≠ LOC:
   LEFT := LEFT + 1.
   [End of loop.]
   (*b*) If LOC = LEFT, then: Return.
   (*c*) If A[LEFT] > A[LOC], then
   (i) [Interchange A[LEFT] and A[LOC].]
   TEMP := A[LOC], A[LOC] := A[LEFT],
   A[LEFT] := TEMP.
   (ii) Set LOC := LEFT.
   (iii) Go to Step 2.
   [End of If structure.]

Algorithm 6.6: (Quicksort) This algorithm sorts an array A with N elements.

1. [Initialize.] TOP := NULL.
2. [Push boundary values of A onto stacks when A has 2 or more elements.]
   If N > 1, then: TOP := TOP + 1, LOWER[1] := 1, UPPER[1] := N.
3. Repeat Steps 4 to 7 while TOP ≠ NULL.
4. [Pop sublist from stacks.]
   Set BEG := LOWER[TOP], END := UPPER[TOP],
   TOP := TOP − 1.
5. Call QUICK(A, N, BEG, END, LOC). [Procedure 6.5.]
6. [Push left sublist onto stacks when it has 2 or more elements.]
   If BEG < LOC − 1, then:
   TOP := TOP + 1, LOWER[TOP] := BEG,
   UPPER[TOP] = LOC − 1.
   [End of If structure.]
7. [Push right sublist onto stacks when it has 2 or more elements.]
   If LOC + 1 < END, then:
   TOP := TOP + 1, LOWER[TOP] := LOC + 1,
   UPPER[TOP] := END.
   [End of If structure.]
   [End of Step 3 loop.]
8. Exit.

Complexity of the Quicksort Algorithm

The running time of a sorting algorithm is usually measured by the number $f(n)$ of comparisons required to sort n elements. The quicksort algorithm, which has many variations, has been studied extensively. Generally speaking, the algorithm has a worst-case running time of order $n^2/2$, but an average-case running time of order $n \log n$. The reason for this is indicated below.

The worst case occurs when the list is already sorted. Then the first element will require n comparisons to recognize that it remains in the first position. Furthermore, the first sublist will be empty, but the second sublist will have $n - 1$ elements. Accordingly, the second element will require $n - 1$ comparisons to recognize that it remains in the second position. And so on. Consequently, there will be a total of

$$f(n) = n + (n-1) + \cdots + 2 + 1 = \frac{n(n+1)}{2} = \frac{n^2}{2} + O(n) = O(n^2)$$

comparisons. Observe that this is equal to the complexity of the bubble sort algorithm (Sec. 4.6).

The complexity $f(n) = O(n \log n)$ of the average case comes from the fact that, on the average, each reduction step of the algorithm produces two sublists. Accordingly:

(1) Reducing the initial list places 1 element and produces two sublists.

(2) Reducing the two sublists places 2 elements and produces four sublists.

(3) Reducing the four sublists places 4 elements and produces eight sublists.

(4) Reducing the eight sublists places 8 elements and produces sixteen sublists.

And so on. Observe that the reduction step in the kth level finds the location of 2^{k-1} elements; hence there will be approximately $\log_2 n$ levels of reductions steps. Furthermore, each level uses at most n comparisons, so $f(n) = O(n \log n)$. In fact, mathematical analysis and empirical evidence have both shown that

$$f(n) \approx 1.4[n \log n]$$

is the expected number of comparisons for the quicksort algorithm.

6.6 RECURSION

Recursion is an important concept in computer science. Many algorithms can be best described in terms of recursion. This section introduces this powerful tool, and Sec. 6.8 will show how recursion may be implemented by means of stacks.

Suppose P is a procedure containing either a Call statement to itself or a Call statement to a second procedure that may eventually result in a Call statement back to the original procedure P. Then P is called a *recursive procedure*. So that the program will not continue to run indefinitely, a recursive procedure must have the following two properties:

(1) There must be certain criteria, called *base criteria*, for which the procedure does not call itself.

(2) Each time the procedure does call itself (directly or indirectly), it must be closer to the base criteria.

A recursive procedure with these two properties is said to be *well-defined*.

Similarly, a function is said to be *recursively defined* if the function definition refers to itself. Again, in order for the definition not to be circular, it must have the following two properties:

(1) There must be certain arguments, called *base values*, for which the function does not refer to itself.

(2) Each time the function does refer to itself, the argument of the function must be closer to a base value.

A recursive function with these two properties is also said to be well-defined.

The following examples should help clarify these ideas.

Factorial Function

The product of the positive integers from 1 to n, inclusive, is called "n factorial" and is usually denoted by $n!$:

$$n! = 1 \cdot 2 \cdot 3 \cdots (n-2)(n-1)n$$

It is also convenient to define $0! = 1$, so that the function is defined for all nonnegative integers. Thus we have

$$0! = 1 \qquad 1! = 1 \qquad 2! = 1 \cdot 2 = 2 \qquad 3! = 1 \cdot 2 \cdot 3 = 6 \qquad 4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$$
$$5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120 \qquad 6! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 = 720$$

and so on. Observe that

$$5! = 5 \cdot 4! = 5 \cdot 24 = 120 \quad \text{and} \quad 6! = 6 \cdot 5! = 6 \cdot 120 = 720$$

This is true for every positive integer n; that is,

$$n! = n \cdot (n-1)!$$

Accordingly, the factorial function may also be defined as follows:

Definition 6.1: (Factorial Function)

(*a*) If $n = 0$, then $n! = 1$.
(*b*) If $n > 0$, then $n! = n \cdot (n-1)!$

Observe that this definition of $n!$ is recursive, since it refers to itself when it uses $(n-1)!$ However, (*a*) the value of $n!$ is explicitly given when $n = 0$ (thus 0 is the base value); and (*b*) the value of $n!$ for arbitrary n is defined in terms of a smaller value of n which is closer to the base value 0. Accordingly, the definition is not circular, or in other words, the procedure is well-defined.

EXAMPLE 6.8

Let us calculate 4! using the recursive definition. This calculation requires the following nine steps:

(1) $4! = 4 \cdot 3!$
(2) $3! = 3 \cdot 2!$
(3) $2! = 2 \cdot 1!$
(4) $1! = 1 \cdot 0!$
(5) $0! = 1$
(6) $1! = 1 \cdot 1 = 1$
(7) $2! = 2 \cdot 1 = 2$
(8) $3! = 3 \cdot 2 = 6$
(9) $4! = 4 \cdot 6 = 24$

That is:

Step 1. This defines 4! in terms of 3!, so we must postpone evaluating 4! until we evaluate 3! This postponement is indicated by indenting the next step.
Step 2. Here 3! is defined in terms of 2!, so we must postpone evaluating 3! until we evaluate 2!
Step 3. This defines 2! in terms of 1!

Step 4. This defines 1! in terms of 0!

Step 5. This step can explicitly evaluate 0!, since 0 is the base value of the recursive definition.

Steps 6 to 9. We backtrack, using 0! to find 1!, using 1! to find 2!, using 2! to find 3!, and finally using 3! to find 4! This backtracking is indicated by the "reverse" indention.

Observe that we backtrack in the reverse order of the original postponed evaluations. Recall that this type of postponed processing lends itself to the use of stacks. (See Sec. 6.2.)

The following are two procedures that each calculate n factorial.

Procedure 6.7A: FACTORIAL(FACT, N)
This procedure calculates N! and returns the value in the variable FACT.

1. If N = 0, then: Set FACT := 1, and Return.
2. Set FACT := 1. [Initializes FACT for loop.]
3. Repeat for K = 1 to N.
   Set FACT := K * FACT.
   [End of loop.]
4. Return.

Procedure 6.7B: FACTORIAL(FACT, N)
This procedure calculates N! and returns the value in the variable FACT.

1. If N = 0, then: Set FACT := 1, and Return.
2. Call FACTORIAL(FACT, N – 1).
3. Set FACT := N * FACT.
4. Return.

Observe that the first procedure evaluates N! using an iterative loop process. The second procedure, on the other hand, is a recursive procedure, since it contains a call to itself. Some programming languages, notably FORTRAN, do not allow such recursive subprograms.

Suppose P is a recursive procedure. During the running of an algorithm or a program which contains P, we associate a *level number* with each given execution of procedure P as follows. The original execution of procedure P is assigned level 1; and each time procedure P is executed because of a recursive call, its level is 1 more than the level of the execution that has made the recursive call. In Example 6.8, Step 1 belongs to level 1. Hence Step 2 belongs to level 2, Step 3 to level 3, Step 4 to level 4 and Step 5 to level 5. On the other hand, Step 6 belongs to level 4, since it is the result of a return from level 5. In other words, Step 6 and Step 4 belong to the same level of execution. Similarly, Step 7 belongs to level 3, Step 8 to level 2, and the final step, Step 9, to the original level 1.

The *depth* of recursion of a recursive procedure P with a given set of arguments refers to the maximum level number of P during its execution.

Fibonacci Sequence

The celebrated Fibonacci sequence (usually denoted by $F_0, F_1, F_2, \ldots$) is as follows:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

That is, $F_0 = 0$ and $F_1 = 1$ and each succeeding term is the sum of the two preceding terms. For example, the next two terms of the sequence are

$$34 + 55 = 89 \quad \text{and} \quad 55 + 89 = 144$$

A formal definition of this function follows:

Definition 6.2: (Fibonacci Sequence)

(*a*) If $n = 0$ or $n = 1$, then $F_n = n$.

(*b*) If $n > 1$, then $F_n = F_{n-2} + F_{n-1}$.

This is another example of a recursive definition, since the definition refers to itself when it uses F_{n-2} and F_{n-1}. Here (*a*) the base values are 0 and 1, and (*b*) the value of F_n is defined in terms of smaller values of n which are closer to the base values. Accordingly, this function is well-defined.

A procedure for finding the nth term F_n of the Fibonacci sequence follows.

Procedure 6.8: FIBONACCI(FIB, N)
This procedure calculates F_N and returns the value in the first parameter FIB.

1. If N = 0 or N = 1, then: Set FIB := N, and Return.
2. Call FIBONACCI(FIBA, N − 2).
3. Call FIBONACCI(FIBB, N − 1).
4. Set FIB := FIBA + FIBB.
5. Return.

This is another example of a recursive procedure, since the procedure contains a call to itself. In fact, this procedure contains two calls to itself. We note (see Prob. 6.16) that one can also write an iterative procedure to calculate F_n which does not use recursion.

Divide-and-Conquer Algorithms

Consider a problem P associated with a set S. Suppose A is an algorithm which partitions S into smaller sets such that the solution of the problem P for S is reduced to the solution of P for one or more of the smaller sets. Then A is called a divide-and-conquer algorithm.

Two examples of divide-and-conquer algorithms, previously treated, are the quicksort algorithm in Sec. 6.5 and the binary search algorithm in Sec. 4.7. Recall that the quicksort algorithm uses a reduction step to find the location of a single element and to reduce the problem of sorting the entire set to the problem of sorting smaller sets. The binary search algorithm divides the given sorted set into two halves so that the problem of searching for an item in the entire set is reduced to the problem of searching for the item in one of the two halves.

A divide-and-conquer algorithm A may be viewed as a recursive procedure. The reason for this is that the algorithm A may be viewed as calling itself when it is applied to the smaller sets. The base criteria for these algorithms are usually the one-element-sets. For example, with a sorting algorithm, a one-element set is automatically sorted; and with a searching algorithm, a one-element set requires only a single comparison.

Ackermann Function

The Ackermann function is a function with two arguments each of which can be assigned any nonnegative integer: 0, 1, 2, ... This function is defined as follows:

Definition 6.3: (Ackermann Function)

(*a*) If $m = 0$, then $A(m, n) = n + 1$.

(*b*) If $m \neq 0$ but $n = 0$, then $A(m, n) = A(m - 1, 1)$.

(*c*) If $m \neq 0$ and $n \neq 0$, then $A(m, n) = A(m - 1, A(m, n - 1))$

Once more, we have a recursive definition, since the definition refers to itself in parts (*b*) and (*c*). Observe that $A(m, n)$ is explicitly given only when $m = 0$. The base criteria are the pairs

$$(0, 0), \quad (0, 1), \quad (0, 2), \quad (0, 3), \ldots, (0, n), \ldots$$

Although it is not obvious from the definition, the value of any $A(m, n)$ may eventually be expressed in terms of the value of the function on one or more of the base pairs.

The value of $A(1, 3)$ is calculated in Prob. 6.17. Even this simple case requires 15 steps. Generally speaking, the Ackermann function is too complex to evaluate on any but a trivial example. Its importance comes from its use in mathematical logic. The function is stated here mainly to give another example of a classical recursive function and to show that the recursion part of a definition may be complicated.

6.7 TOWERS OF HANOI

The preceding section gave examples of some recursive definitions and procedures. This section shows how recursion may be used as a tool in developing an algorithm to solve a particular problem. The problem we pick is known as the Towers of Hanoi problem.

Suppose three pegs, labeled A, B and C, are given, and suppose on peg A there are placed a finite number n of disks with decreasing size. This is pictured in Fig. 6-10 for the case $n = 6$. The object of the game is to move the disks from peg A to peg C using peg B as an auxiliary. The rules of the game are as follows:

(*a*) Only one disk may be moved at a time. Specifically, only the top disk on any peg may be moved to any other peg.

(*b*) At no time can a larger disk be placed on a smaller disk.

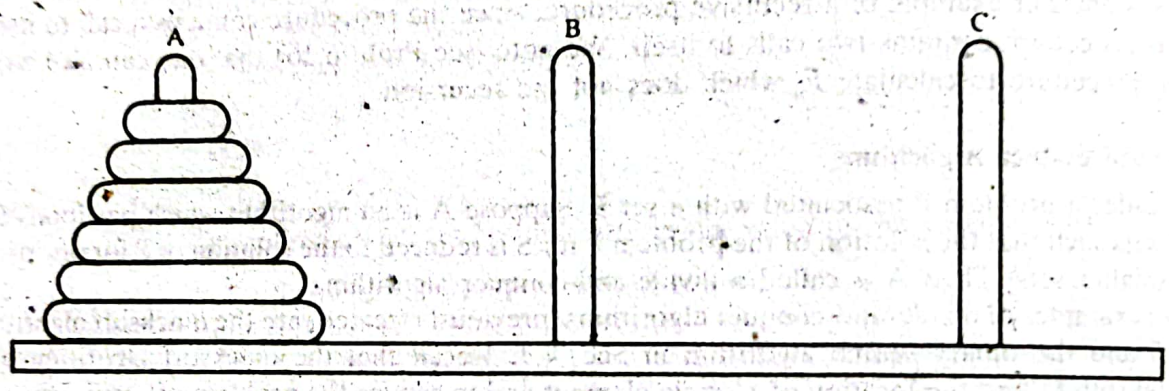


Fig. 6-10 Initial setup of Towers of Hanoi with $n = 6$.

Sometimes we will write X→Y to denote the instruction "Move top disk from peg X to peg Y," where X and Y may be any of the three pegs.

The solution to the Towers of Hanoi problem for $n = 3$ appears in Fig. 6-11. Observe that it consists of the following seven moves:

$n = 3$: Move top disk from peg A to peg C.
Move top disk from peg A to peg B.
Move top disk from peg C to peg B.
Move top disk from peg A to peg C.
Move top disk from peg B to peg A.
Move top disk from peg B to peg C.
Move top disk from peg A to peg C.

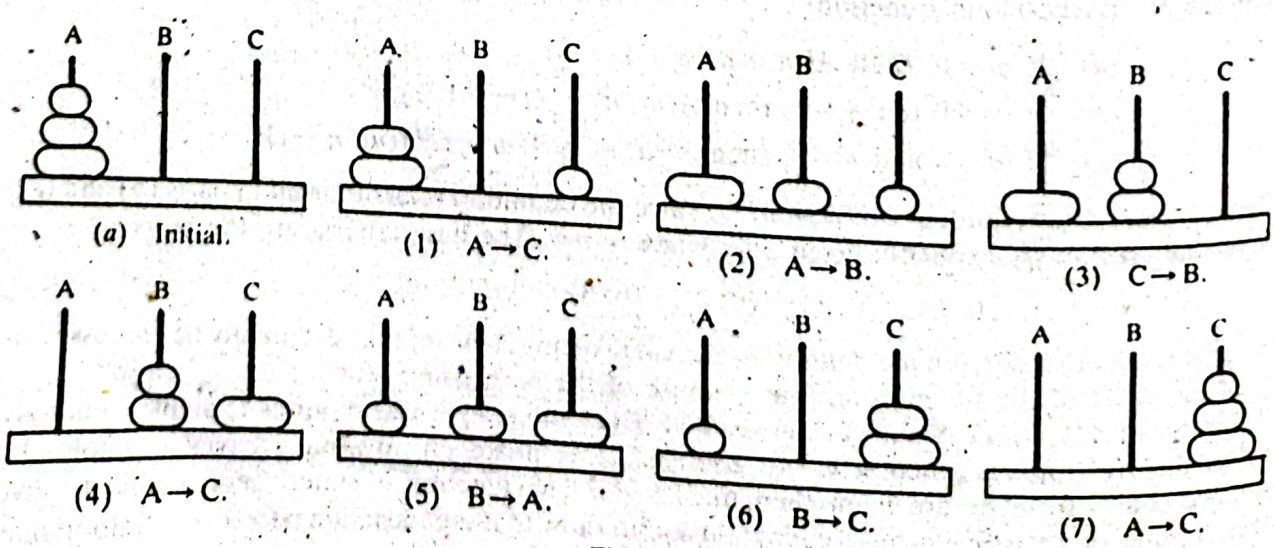


Fig. 6-11

In other words,

$n = 3$: $A \rightarrow C$, $A \rightarrow B$, $C \rightarrow B$, $A \rightarrow C$, $B \rightarrow A$, $B \rightarrow C$, $A \rightarrow C$

For completeness, we also give the solution to the Towers of Hanoi problem for $n = 1$ and $n = 2$:

$n = 1$: $A \rightarrow C$

$n = 2$: $A \rightarrow B$, $A \rightarrow C$, $B \rightarrow C$

Note that $n = 1$ uses only one move and that $n = 2$ uses three moves.

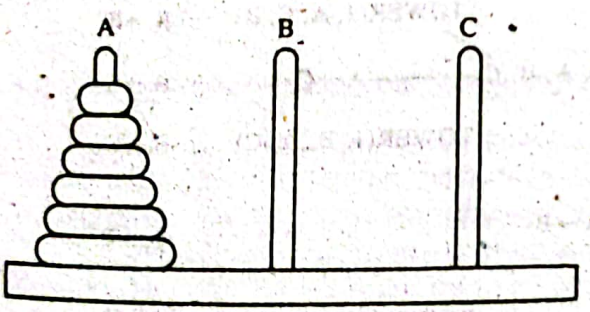
Rather than finding a separate solution for each n, we use the technique of recursion to develop a general solution. First we observe that the solution to the Towers of Hanoi problem for $n > 1$ disks may be reduced to the following subproblems:

(1) Move the top $n - 1$ disks from peg A to peg B.

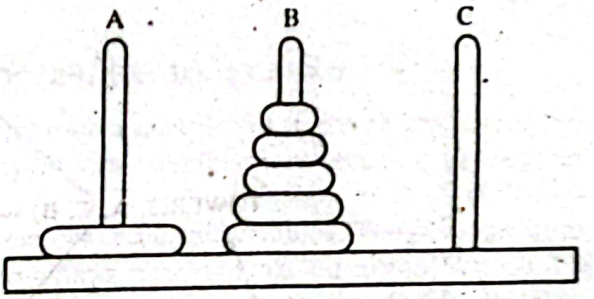
(2) Move the top disk from peg A to peg C: $A \rightarrow C$.

(3) Move the top $n - 1$ disks from peg B to peg C.

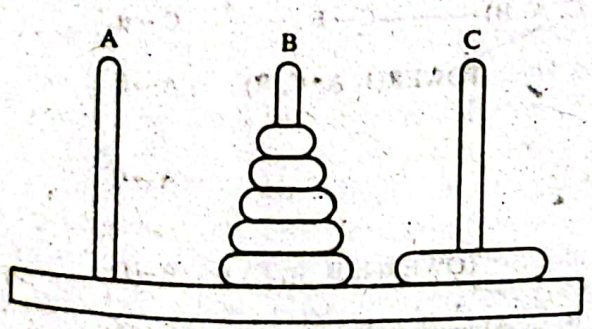
This reduction is illustrated in Fig. 6-12 for $n = 6$. That is, first we move the top five disks from peg A to peg B, then we move the large disk from peg A to peg C, and then we move the top five disks from peg B to peg C.



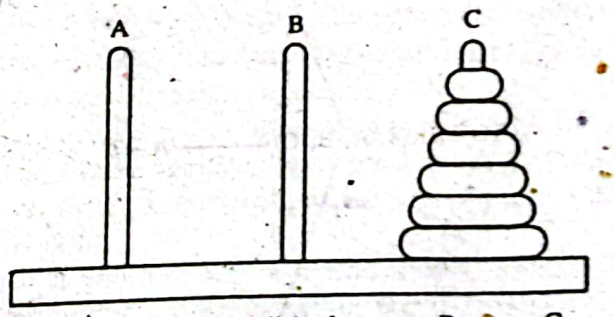
(*a*) Initial: $n = 6$.



(*b*) Move top five disks from peg A to peg B.



(*c*) Move top disk from peg A to peg C.



(*d*) Move top five disks from peg B to peg C.

Fig. 6-12

Let us now introduce the general notation

TOWER(N, BEG, AUX, END)

to denote a procedure which moves the top n disks from the initial peg BEG to the final peg END using the peg AUX as an auxiliary. When $n = 1$, we have the following obvious solution:

TOWER(1, BEG, AUX, END) consists of the single instruction $BEG \rightarrow END$

Furthermore, as discussed above, when $n > 1$, the solution may be reduced to the solution of the following three subproblems:

(1) TOWER(N − 1, BEG, END, AUX)
(2) TOWER(1, BEG, AUX, END) or BEG→END
(3) TOWER(N − 1, AUX, BEG, END)

Observe that each of these three subproblems may be solved directly or is essentially the same as the original problem using fewer disks. Accordingly, this reduction process does yield a recursive solution to the Towers of Hanoi problem.

Figure 6-13 contains a schematic diagram of the above recursive solution for

TOWER(4, A, B, C)

Observe that the recursive solution for $n = 4$ disks consists of the following 15 moves:

A→B	A→C	B→C	A→B	C→A	C→B	A→B	A→C
B→C	B→A	C→A	B→C	A→B	A→C	B→C	

In general, this recursive solution requires $f(n) = 2^n - 1$ moves for n disks.

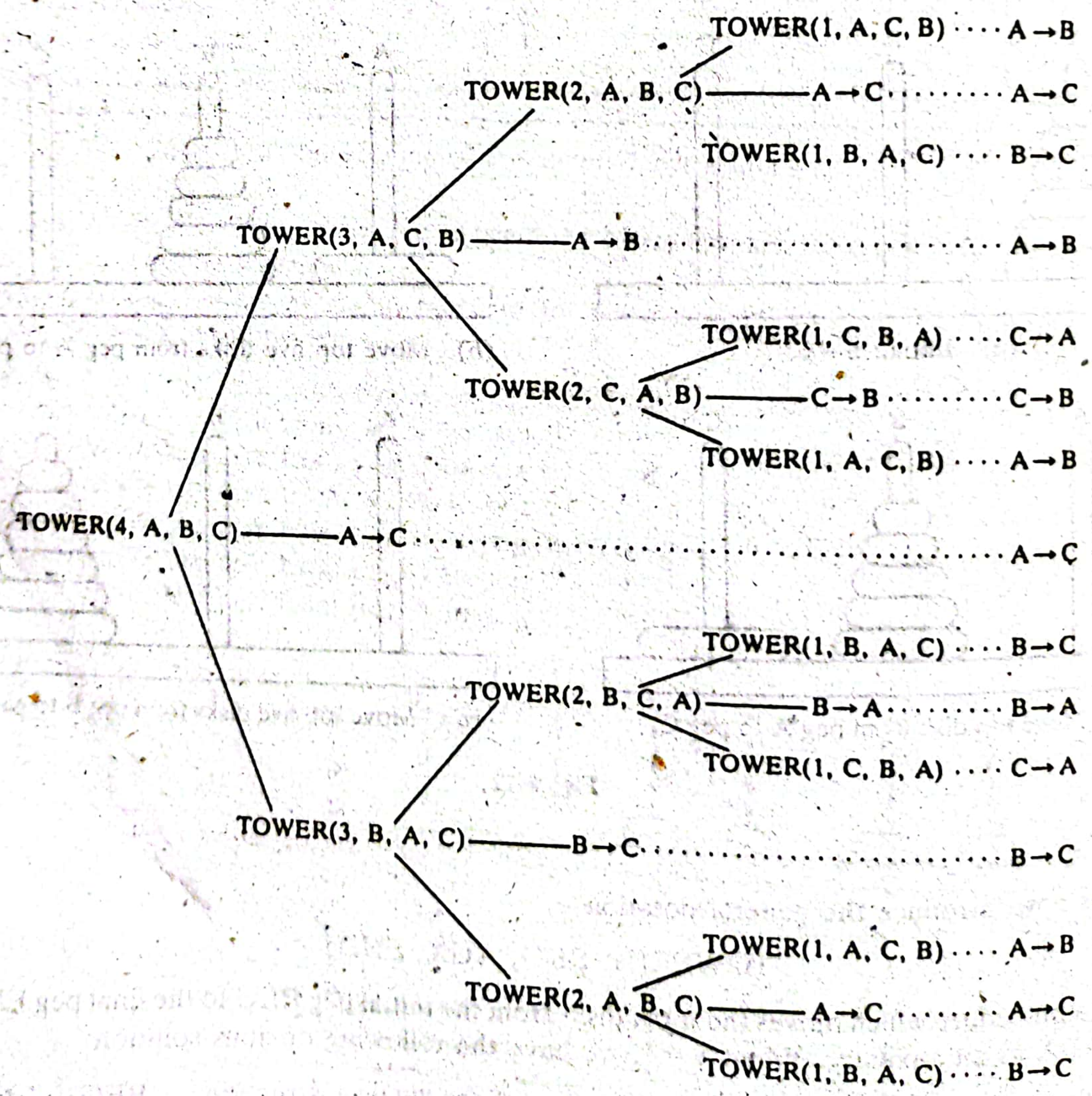


Fig. 6-13 Recursive solution to Towers of Hanoi problem for $n = 4$.

We summarize our investigation with the following formally written procedure.

Procedure 6.9: TOWER(N, BEG, AUX, END)
This procedure gives a recursive solution to the Towers of Hanoi problem for N disks.

1. If N = 1, then:
   (*a*) Write: BEG → END.
   (*b*) Return.
   [End of If structure.]
2. [Move N − 1 disks from peg BEG to peg AUX.]
   Call TOWER(N − 1, BEG, END, AUX).
3. Write: BEG → END.
4. [Move N − 1 disks from peg AUX to peg END.]
   Call TOWER(N − 1, AUX, BEG, END).
5. Return.

One can view this solution as a divide-and-conquer algorithm, since the solution for n disks is reduced to a solution for $n - 1$ disks and a solution for $n = 1$ disk.

6.8 IMPLEMENTATION OF RECURSIVE PROCEDURES BY STACKS

The preceding sections showed how recursion may be a useful tool in developing algorithms for specific problems. This section shows how stacks may be used to implement recursive procedures. It is instructive to first discuss subprograms in general.

Recall that a subprogram can contain both parameters and local variables. The parameters are the variables which receive values from objects in the calling program, called arguments, and which transmit values back to the calling program. Besides the parameters and local variables, the subprogram must also keep track of the return address in the calling program. This return address is essential, since control must be transferred back to its proper place in the calling program. At the time that the subprogram is finished executing and control is transferred back to the calling program, the values of the local variables and the return address are no longer needed.

Suppose our subprogram is a recursive program. Then each level of execution of the subprogram may contain different values for the parameters and local variables and for the return address. Furthermore, if the recursive program does call itself, then these current values must be saved, since they will be used again when the program is reactivated.

Suppose a programmer is using a high-level language which admits recursion, such as Pascal. Then the computer handles the bookkeeping that keeps track of all the values of the parameters, local variables and return addresses. On the other hand, if a programmer is using a high-level language which does not admit recursion, such as FORTRAN, then the programmer must set up the necessary bookkeeping by translating the recursive procedure into a nonrecursive one. This bookkeeping is discussed below.

Translation of a Recursive Procedure into a Nonrecursive Procedure

Suppose P is a recursive procedure. We assume that P is a subroutine subprogram rather than a function subprogram. (This is no loss in generality, since function subprograms can easily be written as subroutine subprograms.) We also assume that a recursive call to P comes only from the procedure P. (The treatment of indirect recursion lies beyond the scope of this text.)

The translation of the recursive procedure P into a nonrecursive procedure works as follows. First of all, one defines:

(1) A stack STPAR for each parameter PAR
(2) A stack STVAR for each local variable VAR
(3) A local variable ADD and a stack STADD to hold return addresses

Each time there is a recursive call to P, the current values of the parameters and local variables are pushed onto the corresponding stacks for future processing, and each time there is a recursive return to P, the values of parameters and local variables for the current execution of P are restored from the stacks. The handling of the return addresses is more complicated; it is done as follows.

Suppose the procedure P contains a recursive Call P, in Step K. Then there are two return addresses associated with the execution of this Step K:

(1) There is the current return address of the procedure P, which will be used when the current level of execution of P is finished executing.
(2) There is the new return address K + 1, which is the address of the step following the Call P and which will be used to return to the current level of execution of procedure P.

Some texts push the first of these two addresses, the current return address, onto the return address stack STADD, whereas some texts push the second address, the new return address K + 1, onto STADD. We will choose the latter method; since the translation of P into a nonrecursive procedure will then be simpler. This also means, in particular, that an empty stack STADD will indicate a return to the main program that initially called the recursive procedure P. (The alternative translation which pushes the current return address onto the stack is discussed in Prob. 6.20.)

The algorithm which translates the recursive procedure P into a nonrecursive procedure follows. It consists of three parts: (1) preparation, (2) translating each recursive Call P in procedure P and (3) translating each Return in procedure P.

(1) Preparation.
  (*a*) Define a stack STPAR for each parameter PAR, a stack STVAR for each local variable VAR, and a local variable ADD and a stack STADD to hold return addresses.
  (*b*) Set TOP := NULL.

(2) Translation of "Step K. Call P."
  (*a*) Push the current values of the parameters and local variables onto the appropriate stacks, and push the new return address [Step] K + 1 onto STADD.
  (*b*) Reset the parameters using the new argument values.
  (*c*) Go to Step 1. [The beginning of the procedure P.]

(3) Translation of "Step J. Return."
  (*a*) If STADD is empty, then: Return. [Control is returned to the main program.]
  (*b*) Restore the top values of the stacks. That is, set the parameters and local variables equal to the top values on the stacks, and set ADD equal to the top value on the stack STADD.
  (*c*) Go to Step ADD.

Observe that the translation of "Step K. Call P" does depend on the value of K, but that the translation of "Step J. Return" does not depend on the value of J. Accordingly, one need translate only one Return statement, for example, by using

Step L. Return.

as above and then replace every other Return statement by

Go to Step L.

This will simplify the translation of the procedure.

Towers of Hanoi, Revisited

Consider again the Towers of Hanoi problem. Procedure 6.9 is a recursive solution to the problem for n disks. We translate the procedure into a nonrecursive solution. In order to keep the steps analogous, we label the beginning statement TOP := NULL as Step 0. Also, only the Return statement in Step 5 will be translated, as in (3) on the preceding page.

Procedure 6.10: TOWER(N, BEG, AUX, END)
This is a nonrecursive solution to the Towers of Hanoi problem for N disks which is obtained by translating the recursive solution. Stacks STN, STBEG, STAUX, STEND and STADD will correspond, respectively, to the variables N, BEG, AUX, END and ADD.

0. Set TOP := NULL.
1. If N = 1, then:
   (*a*) Write: BEG → END.
   (*b*) Go to Step 5.
   [End of If structure.]
2. [Translation of "Call TOWER(N − 1, BEG, END, AUX)."]
   (*a*) [Push current values and new return address onto stacks.]
      (i) Set TOP := TOP + 1.
      (ii) Set STN[TOP] := N, STBEG[TOP] := BEG, STAUX[TOP] := AUX, STEND[TOP] := END, STADD[TOP] := 3.
   (*b*) [Reset parameters.]
   Set N := N − 1, BEG := BEG, AUX := END, END := AUX.
   (*c*) Go to Step 1.
3. Write: BEG → END.
4. [Translation of "Call TOWER(N − 1, AUX, BEG, END)."]
   (*a*) [Push current values and new return address onto stacks.]
      (i) Set TOP := TOP + 1.
      (ii) Set STN[TOP] := N, STBEG[TOP] := BEG, STAUX[TOP] := AUX, STEND[TOP] := END, STADD[TOP] := 5.
   (*b*) [Reset parameters.]
   Set N := N − 1, BEG := AUX, AUX := BEG, END := END.
   (*c*) Go to Step 1.
5. [Translation of "Return."]
   (*a*) If TOP := NULL, then: Return.
   (*b*) [Restore top values on stacks.]
      (i) Set N := STN[TOP], BEG := STBEG[TOP], AUX := STAUX[TOP], STEND[TOP], ADD := STADD[TOP].
      (ii) Set TOP := TOP − 1.
   (*c*) Go to Step ADD.

Suppose that a main program does contain the following statement:

Call TOWER(3, A, B, C)

We simulate the execution of the solution of the problem in Procedure 6.10, emphasizing the different levels of execution of the procedure. Each level of execution will begin with an initialization step where the parameters are assigned the argument values from the initial calling statement or from the

	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)	(k)	(l)
STN:	3	3, 2	3	3, 2	3		3	3, 2	3	3, 2	3	
STBEG:	A	A, A	A	A, A	A		A	A, B	A	A, B	A	
STAUX:	B	B, C	B	B, C	B		B	B, A	B	B, A	B	
STEND:	C	C, B	C	C, B	C		C	C, C	C	C, C	C	
STADD:	3	3, 3	3	3, 5	3		5	5, 3	5	5, 5	5	

Fig. 6-14 Stacks for TOWER(3, A, B, C).

recursive call in Step 2 or Step 4. (Hence each new return address is either Step 3 or Step 5.) Figure 6-14 shows the different stages of the stacks.

(*a*) (Level 1) The initial Call TOWER(3, A, B, C) assigns the following values to the parameters:

$$N := 3, \quad BEG := A, \quad AUX := B, \quad END := C$$

Step 1. Since $N \neq 1$, control is transferred to Step 2.
Step 2. This is a recursive call. Hence the current values of the variables and the new return address (Step 3) are pushed onto the stacks as pictured in Fig. 6-14(*a*).

(*b*) (Level 2) The Step 2 recursive call [TOWER(N − 1, BEG, END, AUX)] assigns the following values to the parameters:

$$N := N - 1 = 2, \quad BEG := BEG = A, \quad AUX := END = C, \quad END := AUX = B$$

Step 1. Since $N \neq 1$, control is transferred to Step 2.
Step 2. This is a recursive call. Hence the current values of the variables and the new return address (Step 3) are pushed onto the stacks as pictured in Fig. 6-14(*b*).

(*c*) (Level 3) The Step 2 recursive call [TOWER(N − 1, BEG, END, AUX)] assigns the following values to the parameters:

$$N := N - 1 = 1, \quad BEG := BEG = A, \quad AUX := END = B, \quad END := AUX = C$$

Step 1. Now $N = 1$. The operation BEG → END implements the move

$$A \rightarrow C$$

Then control is transferred to Step 5. [For the Return.]
Step 5. The stacks are not empty, so the top values on the stacks are removed, leaving Fig 6-14(*c*), and are assigned as follows:

$$N := 2, \quad BEG := A, \quad AUX := C, \quad END := B, \quad ADD := 3$$

Control is transferred to the preceding Level 2 at Step ADD.

(*d*) (Level 2) [Reactivated at Step ADD = 3.]
Step 3. The operation BEG → END implements the move

$$A \rightarrow B$$

Step 4. This is a recursive call. Hence the current values of the variables and the new return address (Step 5) are pushed onto the stacks as pictured in Fig. 6-14(*d*).

(*e*) (Level 3) The Step 4 recursive call [TOWER(N − 1, AUX, BEG, END)] assigns the following values to the parameters:

$$N := N - 1 = 1, \quad BEG := AUX = C, \quad AUX := BEG = A,$$
$$END := END = B$$

Step 1. Now N = 1. The operation BEG → END implements the move

$$C \rightarrow B$$

Then control is transferred to Step 5. [For the Return.]

Step 5. The stacks are not empty; hence the top values on the stacks are removed, leaving Fig. 6-14(*e*), and they are assigned as follows:

$$N := 2, \quad BEG := A, \quad AUX := C, \quad END := B, \quad ADD := 5$$

Control is transferred to the preceding Level 2 at Step ADD.

(*f*) (Level 2) [Reactivation at Step ADD = 5.]

Step 5. The stacks are not empty; hence the top values on the stacks are removed, leaving Fig. 6-14(*f*), and they are assigned as follows:

$$N := 3, \quad BEG := A, \quad AUX := B, \quad END := C, \quad ADD := 3$$

Control is transferred to the preceding Level 1 at Step ADD.

(*g*) (Level 1) [Reactivation at Step ADD = 3.]

Step 3. The operation BEG → END implements the move

$$A \rightarrow C$$

Step 4. This is a recursive call. Hence the current values of the variables and the new return address (Step 5) are pushed onto the stacks as pictured in Fig. 6-14(*g*).

(*h*) (Level 2) The Step 4 recursive call [TOWER(N − 1, AUX, BEG, END)] assigns the following values to the parameters:

$$N := N - 1 = 2, \quad BEG := AUX = B, \quad AUX := BEG = A, \quad END := END = C$$

Step 1. Since N ≠ 1, control is transferred to Step 2.

Step 2. This is a recursive call. Hence the current values of the variables and the new return address (Step 3) are pushed onto the stacks as pictured in Fig. 6-14(*h*).

(*i*) (Level 3) The Step 2 recursive call [TOWER(N − 1, BEG, END, AUX)] assigns the following values to the parameters:

$$N := N - 1 = 1, \quad BEG := BEG = B, \quad AUX := END = C, \quad END := AUX = A$$

Step 1. Now N = 1. The operation BEG → END implements the move

$$B \rightarrow A$$

Then control is transferred to Step 5. [For the Return.]

Step 5. The stacks are not empty; hence the top values on the stacks are removed, leaving Fig. 6-14(*i*), and they are assigned as follows:

$$N := 2, \quad BEG := B, \quad AUX := A, \quad END := C, \quad ADD := 3$$

Control is transferred to the preceding Level 2 at Step ADD.

(*j*) (Level 2) [Reactivation at Step ADD = 3.]

Step 3. The operation BEG → END implements the move

$$B \rightarrow C$$

Step 4. This is a recursive call. Hence the current values of the variables and the new retu address (Step 5) are pushed onto the stacks as pictured in Fig. 6-14(*j*).

(*k*) (Level 3) The Step 4 recursive call [TOWER(N − 1, AUX, BEG, END)] assigns th following values to the parameters:

$$N := N - 1 = 1, \quad BEG := AUX = C, \quad AUX := BEG = B, \quad END := END = C$$

Step 1. Now N = 1. The operation BEG → END implements the move

$$A \rightarrow C$$

Then control is transferred to Step 5. [For the Return.]

Step 5. The stacks are not empty; hence the top values on the stacks are removed, leaving Fig. 6-14(*k*), and they are assigned as follows:

$$N := 2, \quad BEG := B, \quad AUX := A, \quad END := C, \quad ADD := 5$$

Control is transferred to the preceding Level 2 at Step ADD.

(*l*) (Level 2) [Reactivation at Step ADD = 5.]

Step 5. The stacks are not empty; hence the top values on the stacks are removed, leaving Fig. 6-14(*l*), and they are assigned as follows:

$$N := 3, \quad BEG := A, \quad AUX := B, \quad END := C, \quad ADD := 5$$

Control is transferred to the preceding Level 1 at Step ADD.

(*m*) (Level 1) [Reactivation at Step ADD = 5.]

Step 5. The stacks are now empty. Accordingly, control is transferred to the original main program containing the statement

Call TOWER(3, A, B, C)

Observe that the output consists of the following seven moves:

$$A \rightarrow C, \quad A \rightarrow B, \quad C \rightarrow B, \quad A \rightarrow C, \quad B \rightarrow A, \quad B \rightarrow C, \quad A \rightarrow C$$

This agrees with the solution in Fig. 6-11.

Summary

The Towers of Hanoi problem illustrates the power of recursion in the solution of various algorithmic problems. This section has shown how to implement recursion by means of stacks when using a programming language—notably FORTRAN or COBOL—which does not allow recursive programs. In fact, even when using a programming language—such as Pascal—which does support recursion, the programmer may want to use the nonrecursive solution, since it may be much less expensive than using the recursive solution.

6.9. QUEUES

A *queue* is a linear list of elements in which deletions can take place only at one end, called the *front*, and insertions can take place only at the other end, called the *rear*. The terms "front" and "rear" are used in describing a linear list only when it is implemented as a queue.

Queues are also called first-in first-out (FIFO) lists, since the first element in a queue will be the first element out of the queue. In other words, the order in which elements enter a queue is the order in which they leave. This contrasts with stacks, which are last-in first-out (LIFO) lists.

Queues abound in everyday life. The automobiles waiting to pass through an intersection form a queue, in which the first car in line is the first car through; the people waiting in line at a bank form a queue, where the first person in line is the first person to be waited on; and so on. An important example of a queue in computer science occurs in a time-sharing system, in which programs with the same priority form a queue while waiting to be executed. (Another structure, called a priority queue, is discussed in Sec. 6.11.)

EXAMPLE 6.9

Figure 6-15(*a*) is a schematic diagram of a queue with 4 elements, where AAA is the front element and DDD is the rear element. Observe that the front and rear elements of the queue are also, respectively, the first and last elements of the list. Suppose an element is deleted from the queue. Then it must be AAA. This yields the queue in Fig. 6-15(*b*), where BBB is now the front element. Next, suppose EEE is added to the queue and then FFF is added to the queue. Then they must be added at the rear of the queue, as pictured in Fig. 6-15(*c*). Note that FFF is now the rear element. Now suppose another element is deleted from the queue; then it must be BBB, to yield the queue in Fig. 6-15(*d*). And so on. Observe that in such a data structure, EEE will be deleted before FFF because it has been placed in the queue before FFF. However, EEE will have to wait until CCC and DDD are deleted.

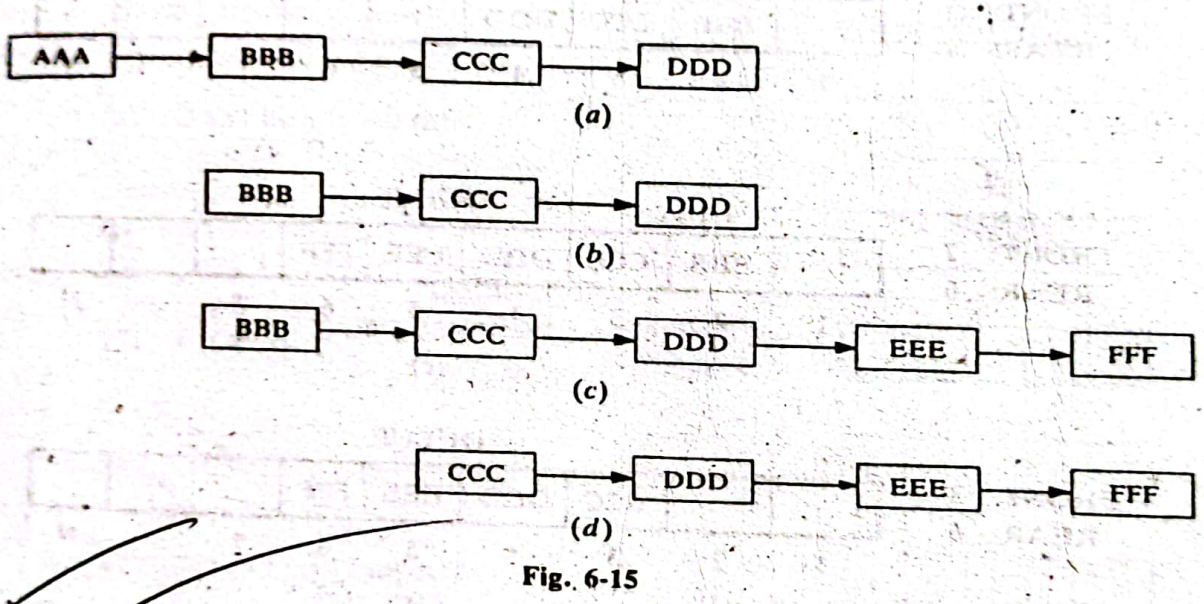


Fig. 6-15

Representation of Queues

Queues may be represented in the computer in various ways, usually by means of one-way lists or linear arrays. Unless otherwise stated or implied, each of our queues will be maintained by a linear array QUEUE and two pointer variables: FRONT, containing the location of the front element of the queue; and REAR, containing the location of the rear element of the queue. The condition FRONT = NULL will indicate that the queue is empty.

Figure 6-16 shows the way the array in Fig. 6-15 will be stored in memory using an array QUEUE with N elements. Figure 6-16 also indicates the way elements will be deleted from the queue and the way new elements will be added to the queue. Observe that whenever an element is deleted from the queue, the value of FRONT is increased by 1; this can be implemented by the assignment

$$\text{FRONT} := \text{FRONT} + 1$$

Similarly, whenever an element is added to the queue, the value of REAR is increased by 1; this can be implemented by the assignment

$$\text{REAR} := \text{REAR} + 1$$

This means that after N insertions, the rear element of the queue will occupy QUEUE[N] or, in other words, eventually the queue will occupy the last part of the array. This occurs even though the queue itself may not contain many elements.

Suppose we want to insert an element ITEM into a queue at the time the queue does occupy the last part of the array, i.e., when REAR = N. One way to do this is to simply move the entire queue to the beginning of the array, changing FRONT and REAR accordingly, and then inserting ITEM as above. This procedure may be very expensive. The procedure we adopt is to assume that the array

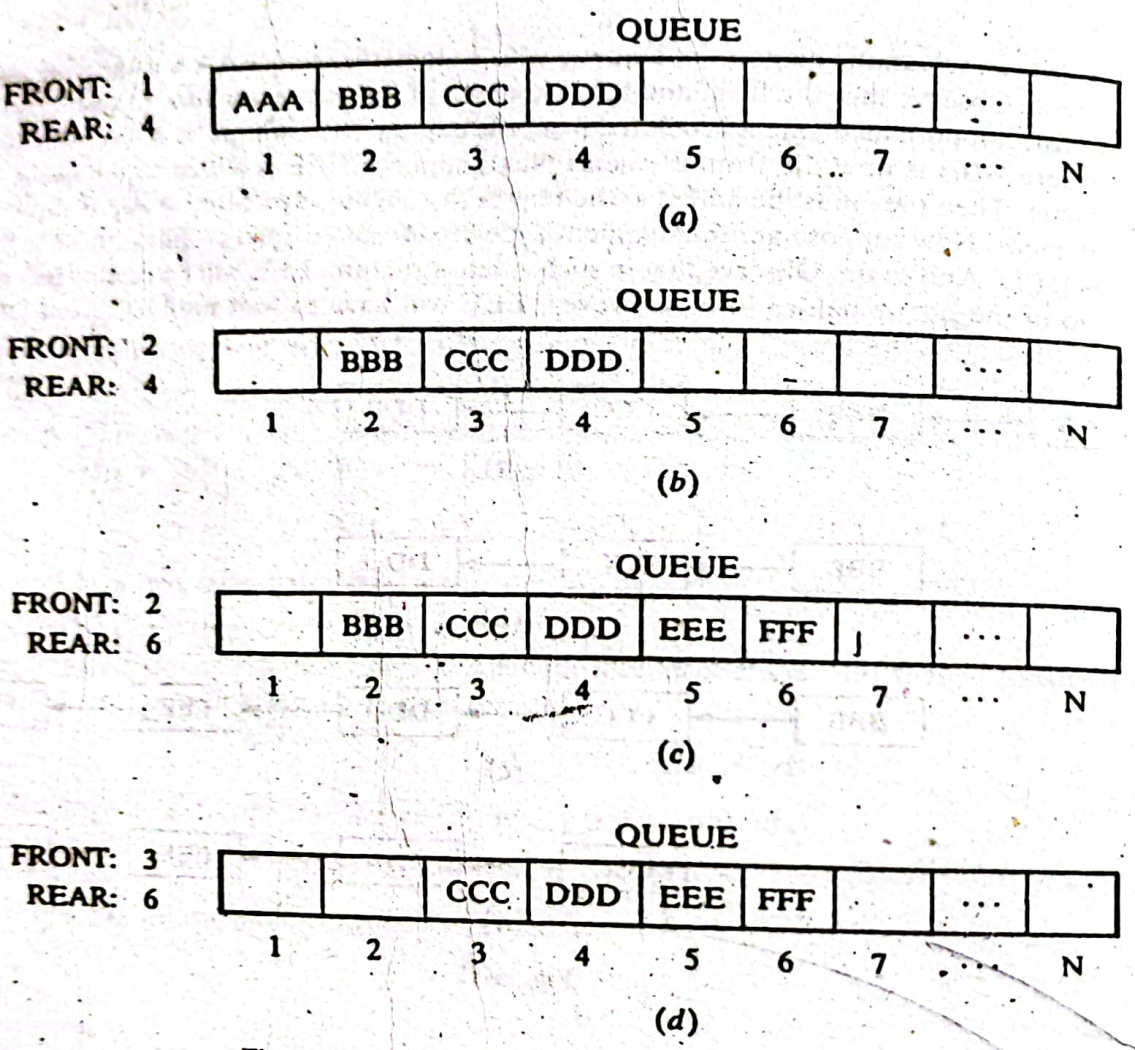


Fig. 6-16. Array representation of a queue.

QUEUE is circular, that is, that QUEUE[1] comes after QUEUE[N] in the array. With this assumption, we insert ITEM into the queue by assigning ITEM to QUEUE[1]. Specifically, instead of increasing REAR to N + 1, we reset REAR = 1 and then assign

$$\text{QUEUE[REAR]} := \text{ITEM}$$

Similarly, if FRONT = N and an element of QUEUE is deleted, we reset FRONT = 1 instead of increasing FRONT to N + 1. (Some readers may recognize this as modular arithmetic, discussed in Sec. 2.2.)

Suppose that our queue contains only one element, i.e., suppose that

$$\text{FRONT} = \text{REAR} \neq \text{NULL}$$

and suppose that the element is deleted. Then we assign

$$\text{FRONT} := \text{NULL} \quad \text{and} \quad \text{REAR} := \text{NULL}$$

to indicate that the queue is empty.

EXAMPLE 6.10

Figure 6-17 shows how a queue may be maintained by a circular array QUEUE with N = 5 memory locations. Observe that the queue always occupies consecutive locations except when it occupies locations at the beginning and at the end of the array. If the queue is viewed as a circular array, this means that it still occupies consecutive locations. Also, as indicated by Fig. 6-17(*m*), the queue will be empty only when FRONT = REAR and an element is deleted. For this reason, NULL is assigned to FRONT

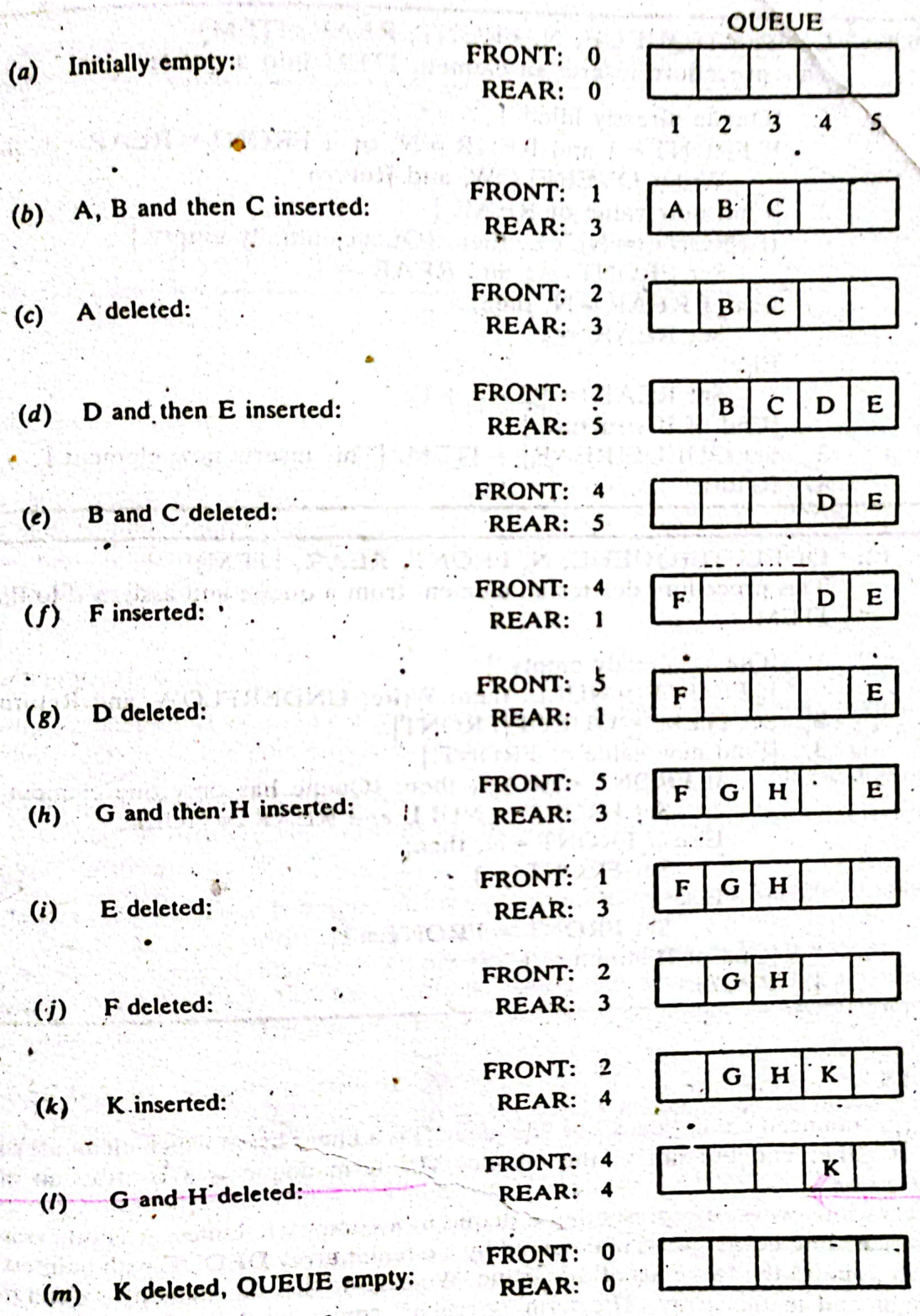


Fig. 6-17

We are now prepared to formaliy state our procedure QINSERT (Procedure 6.11), which inserts a data ITEM into a queue. The first thing we do in the procedure is to test for overflow, that is, to test whether or not the queue is filled.

Next we give a procedure QDELETE (Procedure 6.12), which deletes the first element from a queue, assigning it to the variable ITEM. The first thing we do is to test for underflow, i.e., to test whether or not the queue is empty.

Procedure 6.11: QINSERT(QUEUE, N, FRONT, REAR, ITEM)
This procedure inserts an element ITEM into a queue.

```
1. [Queue already filled?]
   If FRONT = 1 and REAR = N, or if FRONT = REAR + 1, then:
      Write: OVERFLOW, and Return.
2. [Find new value of REAR.]
   If FRONT := NULL, then: [Queue initially empty.]
      Set FRONT := 1 and REAR := 1.
   Else if REAR = N, then:
      Set REAR := 1.
   Else:
      Set REAR := REAR + 1.
   [End of If structure.]
3. Set QUEUE[REAR] := ITEM. [This inserts new element.]
4. Return.
```

Procedure 6.12: QDELETE(QUEUE, N, FRONT, REAR, ITEM)
This procedure deletes an element from a queue and assigns it to the variable ITEM.

```
1. [Queue already empty?]
   If FRONT := NULL, then: Write: UNDERFLOW, and Return.
2. Set ITEM := QUEUE[FRONT].
3. [Find new value of FRONT.]
      If FRONT = REAR, then: [Queue has only one element to start.]
         Set FRONT := NULL and REAR := NULL.
      Else if FRONT = N, then:
         Set FRONT := 1.
      Else:
         Set FRONT := FRONT + 1.
   [End of If structure.]
4. Return.
```

6.10 DEQUES

A *deque* (pronounced either "deck" or "dequeue") is a linear list in which elements can be added or removed at either end but not in the middle. The term deque is a contraction of the name *double-ended queue*.

There are various ways of representing a deque in a computer. Unless it is otherwise stated or implied, we will assume our deque is maintained by a circular array DEQUE with pointers LEFT and RIGHT, which point to the two ends of the deque. We assume that the elements extend from the left end to the right end in the array. The term "circular" comes from the fact that we assume that DEQUE[1] comes after DEQUE[N] in the array. Figure 6-18 pictures two deques, each with 4 elements maintained in an array with N = 8 memory locations. The condition LEFT = NULL will be used to indicate that a deque is empty.

There are two variations of a deque—namely, an input-restricted deque and an output-restricted deque—which are intermediate between a deque and a queue. Specifically, an *input-restricted deque* is a deque which allows insertions at only one end of the list but allows deletions at both ends of the list; and an *output-restricted deque* is a deque which allows deletions at only one end of the list but allows insertions at both ends of the list.

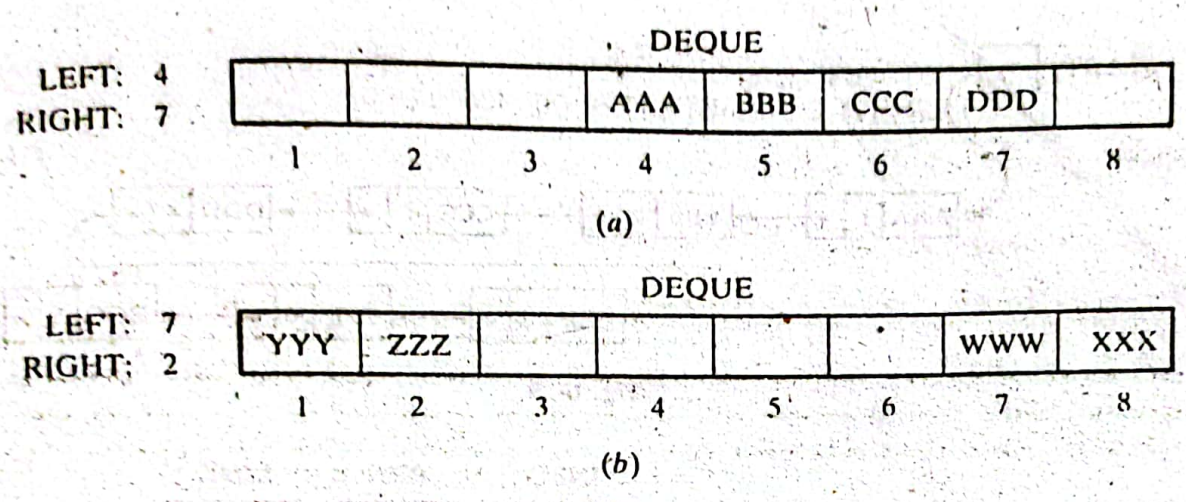


Fig. 6-18

The procedures which insert and delete elements in deques and the variations on those procedures are given as supplementary problems. As with queues, a complication may arise (*a*) when there is overflow, that is, when an element is to be inserted into a deque which is already full, or (*b*) when there is underflow, that is, when an element is to be deleted from a deque which is empty. The procedures must consider these possibilities.

6.11 PRIORITY QUEUES

A *priority queue* is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules:

(1) An element of higher priority is processed before any element of lower priority.

(2) Two elements with the same priority are processed according to the order in which they were added to the queue.

A prototype of a priority queue is a timesharing system: programs of high priority are processed first, and programs with the same priority form a standard queue.

There are various ways of maintaining a priority queue in memory. We discuss two of them here: one uses a one-way list, and the other uses multiple queues. The ease or difficulty in adding elements to or deleting them from a priority queue clearly depends on the representation that one chooses.

One-Way List Representation of a Priority Queue

One way to maintain a priority queue in memory is by means of a one-way list, as follows:

(*a*) Each node in the list will contain three items of information: an information field INFO, a priority number PRN and a link number LINK.

(*b*) A node X precedes a node Y in the list (1) when X has higher priority than Y or (2) when both have the same priority but X was added to the list before Y. This means that the order in the one-way list corresponds to the order of the priority queue.

Priority numbers will operate in the usual way: the lower the priority number, the higher the priority.

EXAMPLE 6.11

Figure 6-19 shows a schematic diagram of a priority queue with 7 elements. The diagram does not tell us whether BBB was added to the list before or after DDD. On the other hand, the diagram does tell us that BBB was inserted before CCC, because BBB and CCC have the same priority number and BBB appears before CCC in the list. Figure 6-20 shows the way the priority queue may appear in memory using linear arrays INFO, PRN and LINK. (See Sec. 5.2.)

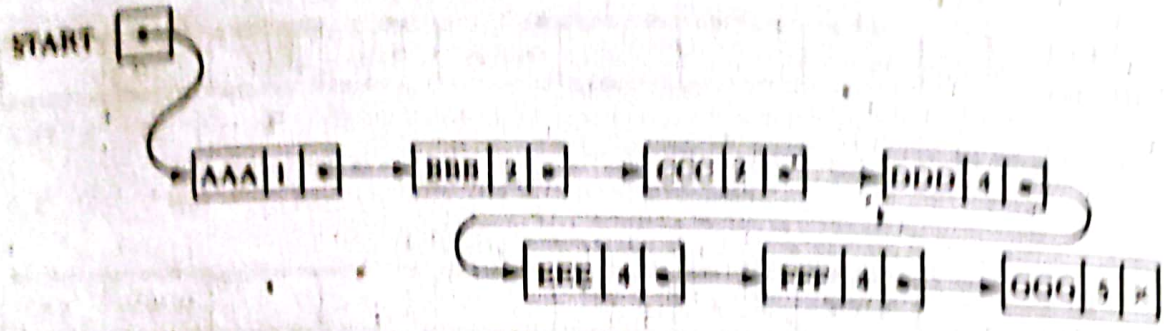


Fig. 6-19

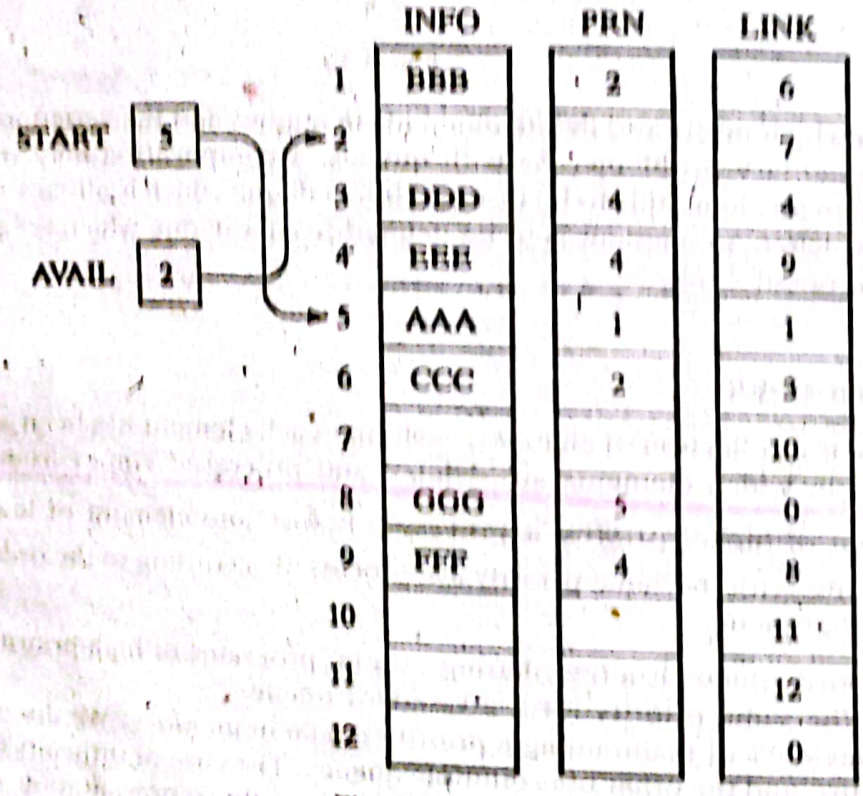


Fig. 6-20

The main property of the one-way list representation of a priority queue is that the element in the queue that should be processed first always appears at the beginning of the one-way list. Accordingly, it is a very simple matter to delete and process an element from our priority queue. The outline of the algorithm follows.

Algorithm 6.13: This algorithm deletes and processes the first element in a priority queue which appears in memory as a one-way list.

1. Set ITEM := INFO[START]. [This saves the data in the first node.]
2. Delete first node from the list.
3. Process ITEM.
4. Exit.

The details of the algorithm, including the possibility of underflow, are left as an exercise.

Adding an element to our priority queue is much more complicated than deleting an element from the queue, because we need to find the correct place to insert the element. An outline of the algorithm follows.

Algorithm 6.14: This algorithm adds an ITEM with priority number N to a priority queue which is maintained in memory as a one-way list.

(*a*) Traverse the one-way list until finding a node X whose priority number exceeds N. Insert ITEM in front of node X.

(*b*) If no such node is found, insert ITEM as the last element of the list.

The above insertion algorithm may be pictured as a weighted object "sinking" through layers of elements until it meets an element with a heavier weight.

The details of the above algorithm are left as an exercise. The main difficulty in the algorithm comes from the fact that ITEM is inserted before node X. This means that, while traversing the list, one must also keep track of the address of the node preceding the node being accessed.

EXAMPLE 6.12

Consider the priority queue in Fig. 6-19. Suppose an item XXX with priority number 2 is to be inserted into the queue. We traverse the list, comparing priority numbers. Observe that DDD is the first element in the list whose priority number exceeds that of XXX. Hence XXX is inserted in the list in front of DDD, as pictured in Fig. 6-21. Observe that XXX comes after BBB and CCC, which have the same priority as XXX. Suppose now that an element is to be deleted from the queue. It will be AAA, the first element in the list. Assuming no other insertions, the next element to be deleted will be BBB, then CCC, then XXX, and so on.

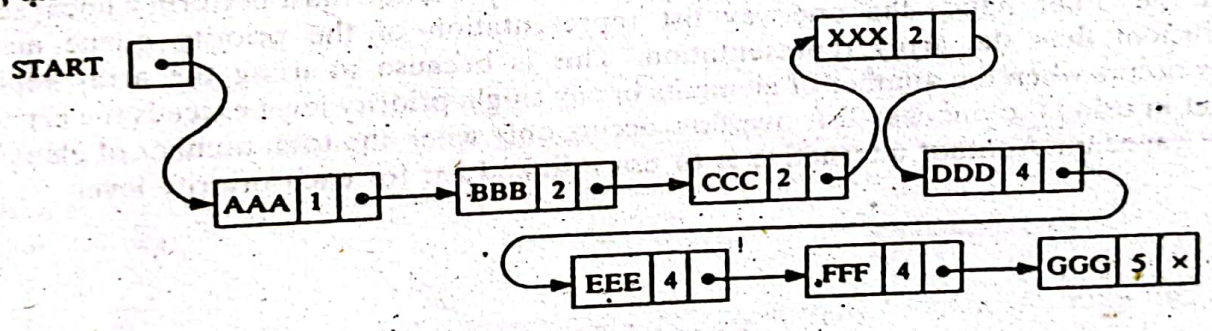


Fig. 6-21

Array Representation of a Priority Queue

Another way to maintain a priority queue in memory is to use a separate queue for each level of priority (or for each priority number). Each such queue will appear in its own circular array and must have its own pair of pointers, FRONT and REAR. In fact, if each queue is allocated the same amount of space, a two-dimensional array QUEUE can be used instead of the linear arrays. Figure 6-22 indicates this representation for the priority queue in Fig. 6-21. Observe that FRONT[K] and REAR[K] contain, respectively, the front and rear elements of row K of QUEUE, the row that maintains the queue of elements with priority number K.

	FRONT	REAR
1	2	2
2	1	3
3	0	0
4	5	1
5	4	4

	1	2	3	4	5	6
1		AAA				
2	BBB	CCC	XXX			
3						
4	FFF				DDD	EEE
5				GGG		

Fig. 6-22

The following are outlines of algorithms for deleting and inserting elements in a priority queue that is maintained in memory by a two-dimensional array QUEUE, as above. The details of the algorithms are left as exercises.

Algorithm 6.15: This algorithm deletes and processes the first element in a priority queue maintained by a two-dimensional array QUEUE.

1. [Find the first nonempty queue.]
   Find the smallest K such that FRONT[K] ≠ NULL.
2. Delete and process the front element in row K of QUEUE.
3. Exit.

Algorithm 6.16: This algorithm adds an ITEM with priority number M to a priority queue maintained by a two-dimensional array QUEUE.

1. Insert ITEM as the rear element in row M of QUEUE.
2. Exit.

Summary

Once again we see the time-space tradeoff when choosing between different data structures for a given problem. The array representation of a priority queue is more time-efficient than the one-way list. This is because when adding an element to a one-way list, one must perform a linear search on the list. On the other hand, the one-way list representation of the priority queue may be more space-efficient than the array representation. This is because in using the array representation, overflow occurs when the number of elements in any single priority level exceeds the capacity for that level, but in using the one-way list, overflow occurs only when the total number of elements exceeds the *total* capacity. Another alternative is to use a linked list for each priority level.

Solved Problems

STACKS

6.1 Consider the following stack of characters, where STACK is allocated N = 8 memory cells:

STACK: A, C, D, F, K, —, —, —

(For notational convenience, we use "—" to denote an empty memory cell.) Describe the stack as the following operations take place:

(*a*) POP(STACK, ITEM)
(*b*) POP(STACK, ITEM)
(*c*) PUSH(STACK, L)
(*d*) PUSH(STACK, P)
(*e*) POP(STACK, ITEM)
(*f*) PUSH(STACK, R)
(*g*) PUSH(STACK, S)
(*h*) POP(STACK, ITEM)

The POP procedure always deletes the top element from the stack, and the PUSH procedure always adds the new element to the top of the stack. Accordingly:

(*a*) STACK: A, C, D, F, —, —, —, —
(*b*) STACK: A, C, D, —, —, —, —, —
(*c*) STACK: A, C, D, L, —, —, —, —
(*d*) STACK: A, C, D, L, P, —, —, —
(*e*) STACK: A, C, D, L, —, —, —, —
(*f*) STACK: A, C, D, L, R, —, —, —
(*g*) STACK: A, C, D, L, R, S, —, —
(*h*) STACK: A, C, D, L, R, —, —, —

6.2 Consider the data in Prob. 6.1. (*a*) When will overflow occur? (*b*) When will C be deleted before D?

(*a*) Since STACK has been allocated N = 8 memory cells, overflow will occur when STACK contains 8 elements and there is a PUSH operation to add another element to STACK.

(*b*) Since STACK is implemented as a stack, C will never be deleted before D.

6.3 Consider the following stack, where STACK is allocated N = 6 memory cells:

STACK: AAA, DDD, EEE, FFF, GGG, _____

Describe the stack as the following operations take place: (*a*) PUSH(STACK, KKK), (*b*) POP(STACK, ITEM), (*c*) PUSH(STACK, LLL), (*d*) PUSH(STACK, SSS), (*e*) POP(STACK, ITEM) and (*f*) PUSH(STACK, TTT).

(*a*) KKK is added to the top of STACK, yielding

STACK: AAA, DDD, EEE, FFF, GGG, KKK

(*b*) The top element is removed from STACK, yielding

STACK: AAA, DDD, EEE, FFF, GGG, _____

(*c*) LLL is added to the top of STACK, yielding

STACK: AAA, DDD, EEE, FFF, GGG, LLL

(*d*) Overflow occurs, since STACK is full and another element SSS is to be added to STACK.

No further operations can take place until the overflow is resolved—by adding additional space for STACK, for example.

6.4 Suppose STACK is allocated N = 6 memory cells and initially STACK is empty, or, in other words, TOP = 0. Find the output of the following module:

1. Set AAA := 2 and BBB := 5.
2. Call PUSH(STACK, AAA).
   Call PUSH(STACK, 4).
   Call PUSH(STACK, BBB + 2).
   Call PUSH(STACK, 9).
   Call PUSH(STACK, AAA + BBB).
3. Repeat while TOP ≠ 0:
   Call POP(STACK, ITEM).
   Write: ITEM.
   [End of loop.]
4. Return.

Step 1. Sets AAA = 2 and BBB = 5.

Step 2. Pushes AAA = 2, 4, BBB + 2 = 7, 9 and AAA + BBB = 7 onto STACK, yielding

STACK: 2, 4, 7, 9, 7, —

Step 3. Pops and prints the elements of STACK until STACK is empty. Since the top element is always popped, the output consists of the following sequence:

7, 9, 7, 4, 2

Observe that this is the reverse of the order in which the elements were added to STACK.

6.5 Suppose a given space S of N contiguous memory cells is allocated to K = 6 stacks. Describe ways that the stacks may be maintained in S.

Suppose no prior data indicate that any one stack will grow more rapidly than any of the other stacks. Then one may reserve N/K cells for each stack, as in Fig. 6-23(*a*), where $B_1, B_2, \ldots, B_6$ denote, respectively, the bottoms of the stacks. Alternatively, one can partition the stacks into pairs and reserve 2N/K cells for each pair of stacks, as in Fig. 6-23(*b*). The second method may decrease the number of times overflow will occur.

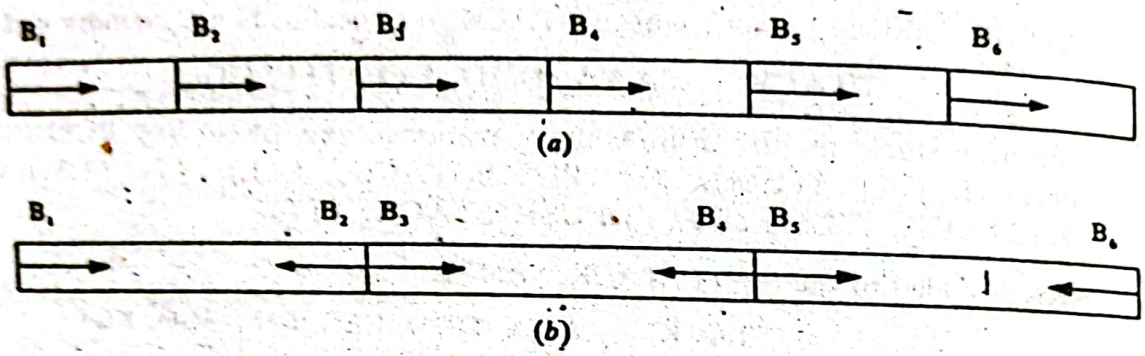


Fig. 6-23

POLISH NOTATION

6.6 Translate, by inspection and hand, each infix expression into its equivalent postfix expression:

(*a*) $(A-B)*(D/E)$ (*b*) $(A+B\uparrow D)/(E-F)+G$

(*c*) $A*(B+D)/E-F*(G+H/K)$.

Using the order in which the operators are executed, translate each operator from infix to postfix notation. (We use brackets [ ] to denote a partial translation.)

(*a*) $(A-B)*(D/E) = [AB-]*[DE/] = AB-DE/*$

(*b*) $(A+B\uparrow D)/(E-F)+G = (A+[BD\uparrow])/[EF-]+G = [ABD\uparrow+]/[EF-]+G$
$= [ABD\uparrow+EF-/]+G = ABD\uparrow+EF-/G+$

(*c*) $A*(B+D)/E-F*(G+H/K) = A*[BD+]/E-F*(G+[HK/])$
$= [ABD+*]/E-F*[GHK/+]$
$= [ABD+*E/]-[FGHK/+*]$
$= ABD+*E/FGHK/+*-$

Observe that we did translate more than one operator in a single step when the operands did not overlap.

6.7 Consider the following arithmetic expression P, written in postfix notation:

P: 12, 7, 3, −, /, 2, 1, 5, +, *, +

(*a*) Translate P, by inspection and hand, into its equivalent infix expression.

(*b*) Evaluate the infix expression.

(*a*) Scanning from left to right, translate each operator from postfix to infix notation. (We use brackets [ ] to denote a partial translation.)

$P = 12, [7-3], /, 2, 1, 5, +, *, +$
$= [12/(7-3)], 2, 1, 5, +, *, +$
$= [12/(7-3)], 2, [1+5], *, +$
$= [12/(7-3)], [2*(1+5)], +$
$= 12/(7-3)+2*(1+5)$

(*b*) Using the infix expression, we obtain:

$$P = 12/(7 - 3) + 2 * (1 + 5) = 12/4 + 2 * 6 = 3 + 12 = 15$$

6.8 Consider the postfix expression P in Prob. 6.7. Evaluate P using Algorithm 6.3.

First add a sentinel right parenthesis at the end of P to obtain:

P: 12, 7, 3, −, /, 2, 1, 5, +, *, +,)

Scan P from left to right. If a constant is encountered, put it on a stack, but if an operator is encountered, evaluate the two top constants on the stack. Figure 6-24 shows the contents of STACK as each element of P is scanned. The final number, 15, in STACK, when the sentinel right parenthesis is scanned, is the value of P. This agrees with the result in Prob. 6.7(*b*).

Symbol	STACK
12	12
7	12, 7
3	12, 7, 3
−	12, 4
/	3
2	3, 2
1	3, 2, 1
5	3, 2, 1, 5
+	3, 2, 6
*	3, 12
+	15
)	15

Fig. 6-24

6.9 Consider the following infix expression Q:

$$Q: \quad ((A + B) * D) \uparrow (E - F)$$

Use Algorithm 6.4 to translate Q into its equivalent postfix expression P.

First push a left parenthesis onto STACK, and then add a right parenthesis to the end of Q to obtain

Q: ((A + B) * D) ↑ (E − F))

(Note that Q now contains 16 elements.) Scan Q from left to right. Recall that (1) if a constant is encountered, it is added to P; (2) if a left parenthesis is encountered, it is put on the stack; (3) if an operator is encountered, it "sinks" to its own level; and (4) if a right parenthesis is encountered, it "sinks" to the first left parenthesis. Figure 6-25 shows pictures of STACK and the string P as each element of Q is scanned. When STACK is empty, the final right parenthesis has been scanned and the result is

P: A B + D * E F − ↑

which is the required postfix equivalent of Q.

6.10 Translate, by inspection and hand, each infix expression into its equivalent prefix expression:

(*a*) (A − B) * (D / E)

(*b*) (A + B ↑ D)/(E − F) + G

Symbol	STACK	Expression P
(	((	
(	(((	
A	(((	A
+	(((+	A
B	(((+	A B
)	((	A B +
*	((*	A B +
D	((*	A B + D
)	(	A B + D *
↑	(↑	A B + D *
(	(↑ (	A B + D *
E	(↑ (	A B + D * E
−	(↑ (−	A B + D * E
F	(↑ (−	A B + D * E F
)	(↑	A B + D * E F −
)		A B + D * E F − ↑

Fig. 6-25

Is there any relationship between the prefix expressions and the equivalent postfix expressions obtained in Prob. 6.6.

Using the order in which the operators are executed, translate each operator from infix to prefix notation.

(*a*) $(A - B) * (D/E) = [-AB] * [/DE] = * - A B / D E$

(*b*) $(A + B \uparrow D)/(E - F) + G = (A + [\uparrow BD])/[-EF] + G$
$= [+A\uparrow BD]/[-EF] + G$
$= [/+A\uparrow BD - EF] + G$
$= + / + A \uparrow B D - E F G$

The prefix expression is not the reverse of the postfix expression. However, the order of the operands—A, B, D and E in part (*a*) and A, B, D, E, F and G in part (*b*)—is the same for all three expressions, infix, postfix and prefix.

QUICKSORT

6.11 Suppose S is the following list of 14 alphabetic characters:

(D) A T A S T R U C T U R E (S)

Suppose the characters in S are to be sorted alphabetically. Use the quicksort algorithm to find the final position of the first character D.

Beginning with the last character S, scan the list from right to left until finding a character which precedes D alphabetically. It is C. Interchange D and C to obtain the list:

(C) A T A S T R U (D) T U R E S

Beginning with this C, scan the list toward D, i.e., from left to right, until finding a character which succeeds D alphabetically. It is T. Interchange D and T to obtain the list:

C A (D) A S (T) R U T T U R E S

Beginning with this T, scan the list toward D until finding a character which precedes D. It is A. Interchange D and A to obtain the list:

C A (A) (D) S T R U T T U R E S

Beginning with this A, scan the list toward D until finding a character which succeeds D. There is no such letter. This means D is in its final position. Furthermore, the letters before D form a sublist consisting of all letters preceding D alphabetically, and the letters after D form a sublist consisting of all the letters succeeding D alphabetically, as follows:

C A A (D) S T R U T T U R E S

Sublist Sublist

Sorting S is now reduced to sorting each sublist.

6.12 Suppose S consists of the following $n = 5$ letters:

(A) B C D (E)

Find the number C of comparisons to sort S using quicksort. What general conclusion can one make, if any?

Beginning with E, it takes $n - 1 = 4$ comparisons to recognize that the first letter A is already in its correct position. Sorting S is now reduced to sorting the following sublist with $n - 1 = 4$ letters:

A (B) C D (E)

Beginning with E, it takes $n - 2 = 3$ comparisons to recognize that the first letter B in the sublist is already in its correct position. Sorting S is now reduced to sorting the following sublist with $n - 2 = 3$ letters:

A B (C) D (E)

Similarly, it takes $n - 3 = 2$ comparisons to recognize that the letter C is in its correct position, and it takes $n - 4 = 1$ comparison to recognize that the letter D is in its correct position. Since only one letter is left, the list is now known to be sorted. Altogether we have:

$$C = 4 + 3 + 2 + 1 = 10 \text{ comparisons}$$

Similarly, using quicksort, it takes

$$C = (n-1) + (n-2) + \cdots + 2 + 1 = \frac{n(n-1)}{2} = \frac{n^2}{2} + 0(n) = O(n^2)$$

comparisons to sort a list with n elements when the list is already sorted. (This can be shown to be the worst case for quicksort.)

6.13 Consider the quicksort algorithm. (*a*) Can the arrays LOWER and UPPER be implemented as queues rather than as stacks? Why? (*b*) How much extra space is needed for the quicksort algorithm, or, in other words, what is the space complexity of the algorithm?

(*a*) Since the order in which the subsets are sorted does not matter, LOWER and UPPER can be implemented as queues, or even deques, rather than as stacks.

(*b*) Quicksort algorithm is an "in-place" algorithm; that is, the elements remain in their places except for interchanges. The extra space is required mainly for the stacks LOWER and UPPER. On the average, the extra space required for the algorithm is proportional to $\log n$, where n is the number of elements to be sorted.

RECURSION

6.14 Let a and b denote positive integers. Suppose a function Q is defined recursively as follows:

$$Q(a, b) = \begin{cases} 0 & \text{if } a < b \\ Q(a-b, b) + 1 & \text{if } b \le a \end{cases}$$

(a) Find the value of $Q(2, 3)$ and $Q(14, 3)$.

(b) What does this function do? Find $Q(5861, 7)$.

(a)
$$\begin{aligned} Q(2, 3) &= 0 \quad \text{since} \quad 2 < 3 \\ Q(14, 3) &= Q(11, 3) + 1 \\ &= [Q(8, 3) + 1] + 1 = Q(8, 3) + 2 \\ &= [Q(5, 3) + 1] + 2 = Q(5, 3) + 3 \\ &= [Q(2, 3) + 1] + 3 = Q(2, 3) + 4 \\ &= 0 + 4 = 4 \end{aligned}$$

(b) Each time b is subtracted from a, the values of Q is increased by 1. Hence $Q(a, b)$ finds the quotient when a is divided by b. Thus,

$$Q(5861, 7) = 837$$

6.15 Let n denote a positive integer. Suppose a function L is defined recursively as follows:

$$L(n) = \begin{cases} 0 & \text{if } n = 1 \\ L(\lfloor n/2 \rfloor) + 1 & \text{if } n > 1 \end{cases}$$

(Here $\lfloor k \rfloor$ denotes the "floor" of k, that is, the greatest integer which does not exceed k. See Sec. 2.2.)

(a) Find $L(25)$.

(b) What does this function do?

(a)
$$\begin{aligned} L(25) &= L(12) + 1 \\ &= [L(6) + 1] + 1 = L(6) + 2 \\ &= [L(3) + 1] + 2 = L(3) + 3 \\ &= [L(1) + 1] + 3 = L(1) + 4 \\ &= 0 + 4 = 4 \end{aligned}$$

(b) Each time n is divided by 2, the value of L is increased by 1. Hence L is the greatest integer such that

$$2^L \le n$$

Accordingly, this function finds

$$L = \lfloor \log_2 n \rfloor$$

6.16 Suppose the Fibonacci numbers $F_{11} = 89$ and $F_{12} = 144$ are given.

(a) Should one use recursion or iteration to obtain F_{16}? Find F_{16}.

(b) Write an iterative procedure to obtain the first N Fibonacci numbers F[1], F[2], . . . , F[N], where $N \ge 2$. (Compare this with the recursive Procedure 6.8.)

(a) The Fibonacci numbers should be evaluated by using iteration (that is, by evaluating from the bottom up), rather than by using recursion (that is, evaluating from the top down).

Recall that each Fibonacci number is the sum of the two preceding Fibonacci numbers. Beginning with F_{11} and F_{12} we have

$$F_{13} = 89 + 144 = 233, \qquad F_{14} = 144 + 233 = 377, \qquad F_{15} = 233 + 377 = 610$$

and hence

$$F_{16} = 377 + 610 = 987$$

(*b*) **Procedure P6.16:** FIBONACCI(F, N)
This procedure finds the first N Fibonacci numbers and assigns them to an array F.

1. Set F[1] := 1 and F[2] := 1.
2. Repeat for L = 3 to N:
   Set F[L] := F[L − 1] + F[L − 2].
   [End of loop.]
3. Return.

(We emphasize that this iterative procedure is much more efficient than the recursive Procedure 6.8.)

6.17 Use the definition of the Ackermann function (Definition 6.3) to find $A(1, 3)$.

We have the following 15 steps:

(1) $A(1, 3) = A(0, A(1, 2))$
(2) $A(1, 2) = A(0, A(1, 1))$
(3) $A(1, 1) = A(0, A(1, 0))$
(4) $A(1, 0) = A(0, 1)$
(5) $A(0, 1) = 1 + 1 = 2$
(6) $A(1, 0) = 2$
(7) $A(1, 1) = A(0, 2)$
(8) $A(0, 2) = 2 + 1 = 3$
(9) $A(1, 1) = 3$
(10) $A(1, 2) = A(0, 3)$
(11) $A(0, 3) = 3 + 1 = 4$
(12) $A(1, 2) = 4$
(13) $A(1, 3) = A(0, 4)$
(14) $A(0, 4) = 4 + 1 = 5$
(15) $A(1, 3) = 5$

The forward indention indicates that we are postponing an evaluation and are recalling the definition, and the backward indention indicates that we are backtracking.

Observe that the first formula in Definition 6.3 is used in Steps 5, 8, 11 and 14, the second formula in Step 4 and the third formula in Steps 1, 2 and 3. In the other Steps we are backtracking with substitutions.

6.18 Suppose a recursive procedure P contains only one recursive call:

Step K. Call P.

Indicate the reason that the stack STADD (for the return addresses) is not necessary.

Since there is only one recursive call, control will always be transferred to Step K + 1 on a Return, except for the final Return to the main program. Accordingly, instead of maintaining the stack STADD

(and the local variable ADD) we simply write

(c) Go to Step K + 1

instead of

(c) Go to Step ADD

in the translation of "Step J. Return." (See Sec. 6.8.)

6.19 Rewrite the solution to the Towers of Hanoi problem so it uses only one recursive call instead of two.

One may view the pegs A and B symmetrically. That is, we apply the steps

Move N − 1 disks from A to B, and then apply A → C
Move N − 2 disks from B to A, and then apply B → C
Move N − 3 disks from A to B, and then apply A → C
Move N − 4 disks from B to A, and then apply B → C

and so on. Accordingly, we can iterate a single recursive call, interchanging BEG and AUX after each iteration, as follows:

Procedure P6.19: TOWER(N, BEG, AUX, END)

1. If N = 0, then: Return.
2. Repeat Steps 3 to 5 for K = N, N − 1, N − 2, . . . , 1.
3. Call TOWER(K − 1, BEG, END, AUX).
4. Write: BEG → END.
5. [Interchange BEG and AUX.]
   Set TEMP := BEG, BEG := AUX, AUX := TEMP.
   [End of Step 2 loop.]
6. Return.

Observe that we use N = 0 as a base value for the recursion instead of N = 1. Either one may be used to yield a solution.

6.20 Consider the stack implementation algorithm in Sec. 6.8 for translating a recursive procedure into a nonrecursive procedure. Recall that, at the time of a recursive call, we pushed the new return address rather than the current return address onto the stack STADD. Suppose we decide to push the current return address onto the stack STADD. (Many texts do this.) What changes must then take place in the translation algorithm?

The main change is that, at the time of a Return to the preceding execution level, the current value of ADD determines the location of the Return, not the value of ADD after the stack values have been popped. Accordingly, the value of ADD must be saved, by setting SAVE := ADD, then the stack values are popped, and then control is transferred to Step SAVE. Another change is that one must initially assign ADD := Main and then Return to the main calling program when ADD = Main, not when the stacks are empty. The formal algorithm follows.

(1) Preparation.
   (a) Define a stack STPAR for each parameter PAR, a stack STVAR for each local variable VAR and a local variable ADD and a stack STADD to hold return addresses.
   (b) Set TOP := NULL and ADD := Main.

(2) Translation of "Step K. Call P."
   (a) Push the current values of the parameters and local variables and the current return address ADD onto the appropriate stacks.
   (b) Reset the parameters using the new argument values, and set ADD := [Step] K + 1.
   (c) Go to Step 1. [The beginning of the procedure P.]

(3) Translation of "Step J. Return."
(a) If ADD = Main, then: Return. [Control is transferred to the main program.]
(b) Set SAVE := ADD.
(c) Restore the top values of the stacks. That is, set the parameters and local variables equal to the top values on the stacks, and set ADD equal to the top value on the stack STADD.
(d) Go to Step SAVE.

(Compare this translation algorithm with the algorithm in Sec. 6.8.)

QUEUES, DEQUES

6.21 Consider the following queue of characters, where QUEUE is a circular array which is allocated six memory cells:

FRONT = 2, REAR = 4 QUEUE: __, A, C, D, __, __

(For notational convenience, we use "__" to denote an empty memory cell.) Describe the queue as the following operations take place:

(a) F is added to the queue.
(b) two letters are deleted.
(c) K, L and M are added to the queue.
(d) two letters are deleted.
(e) R is added to the queue.
(f) two letters are deleted.
(g) S is added to the queue.
(h) two letters are deleted.
(i) one letter is deleted.
(j) one letter is deleted.

(a) F is added to the rear of the queue, yielding

FRONT = 2, REAR = 5 QUEUE: __, A, C, D, F, __

Note that REAR is increased by 1.

(b) The two letters, A and C, are deleted, leaving

FRONT = 4, REAR = 5 QUEUE: __, __, __, D, F, __

Note that FRONT is increased by 2.

(c) K, L and M are added to the rear of the queue. Since K is placed in the last memory cell of QUEUE, L and M are placed in the first two memory cells. This yields

FRONT = 4, REAR = 2 QUEUE: L, M, __, D, F, K

Note that REAR is increased by 3 but the arithmetic is modulo 6:

REAR = 5 + 3 = 8 = 2 (mod 6)

(d) The two front letters, D and F are deleted, leaving

FRONT = 6, REAR = 2 QUEUE: L, M, __, __, __, K

(e) R is added to the rear of the queue, yielding

FRONT = 6, REAR = 3 QUEUE: L, M, R, __, __, K

(f) The two front letters, K and L, are deleted, leaving

FRONT = 2, REAR = 3 QUEUE: __, M, R, __, __, __

Note that FRONT is increased by 2 but the arithmetic is modulo 6:

FRONT = 6 + 2 = 8 = 2 (mod 6)

(g) S is added to the rear of the queue, yielding

FRONT = 2, REAR = 4 QUEUE: __, M, R, S, __, __

(h) The two front letters, M and R, are deleted, leaving

FRONT = 4, REAR = 4 QUEUE: __, __, __, S, __, __

(i) The front letter S is deleted. Since FRONT = REAR, this means that the queue is empty; hence we assign NULL to FRONT and REAR. Thus

FRONT = 0, REAR = 0 QUEUE: __, __, __, __, __, __

(j) Since FRONT = NULL, no deletion can take place. That is, underflow has occurred.

6.22 Suppose each data structure is stored in a circular array with N memory cells.

(a) Find the number NUMB of elements in a queue in terms of FRONT and REAR.

(b) Find the number NUMB of elements in a deque in terms of LEFT and RIGHT.

(c) When will the array be filled?

(a) If FRONT ≤ REAR, then NUMB = REAR − FRONT + 1. For example, consider the following queue with N = 12:

FRONT = 3, REAR = 9 QUEUE: __, __, *, *, *, *, *, *, *, __, __, __

Then NUMB = 9 − 3 + 1 = 7, as pictured.

If REAR < FRONT, then FRONT − REAR − 1 is the number of empty cells, so

NUMB = N − (FRONT − REAR − 1) = N + REAR − FRONT + 1

For example, consider the following queue with N = 12:

FRONT = 9, REAR = 4 QUEUE: *, *, *, *, __, __, __, __, *, *, *, *

Then NUMB = 12 + 4 − 9 + 1 = 8, as pictured.

Using arithmetic modulo N, we need only one formula, as follows:

NUMB = REAR − FRONT + 1 (mod N)

(b) The same result holds for deques except that FRONT is replaced by RIGHT. That is,

NUMB = RIGHT − LEFT + 1 (mod N)

(c) With a queue, the array is full when

(i) FRONT = 1 and REAR = N or (ii) FRONT = REAR + 1

Similarly, with a deque, the array is full when

(i) LEFT = 1 and RIGHT = N or (ii) LEFT = RIGHT + 1

Each of these conditions implies NUMB = N.

6.23 Consider the following deque of characters where DEQUE is a circular array which is allocated six memory cells:

LEFT = 2, RIGHT = 4 DEQUE: __, A, C, D, __, __

Describe the deque while the following operations take place.

(a) F is added to the right of the deque.

(b) Two letters on the right are deleted.

(c) K, L and M are added to the left of the deque.

(d) One letter on the left is deleted.

(e) R is added to the left of the deque.

(*f*) S is added to the right of the deque.

(*g*) T is added to the right of the deque.

(*a*) F is added on the right, yielding

LEFT = 2, RIGHT = 5 DEQUE: __, A, C, D, F, __

Note that RIGHT is increased by 1.

(*b*) The two right letters, F and D, are deleted, yielding

LEFT = 2, RIGHT = 3 DEQUE: __, A, C, __, __, __

Note that RIGHT is decreased by 2.

(*c*) K, L and M are added on the left. Since K is placed in the first memory cell, L is placed in the last memory cell and M is placed in the next-to-last memory cell. This yields

LEFT = 5, RIGHT = 3 DEQUE: K, A, C, __, M, L

Note that LEFT is decreased by 3 but the arithmetic is modulo 6:

$$\text{LEFT} = 2 - 3 = -1 = 5 \pmod 6$$

(*d*) The left letter, M, is deleted, leaving

LEFT = 6, RIGHT = 3 DEQUE: K, A, C, __, __, L

Note that LEFT is increased by 1.

(*e*) R is added on the left, yielding

LEFT = 5, RIGHT = 3 DEQUE: K, A, C, __, R, L

Note that LEFT is decreased by 1.

(*f*) S is added on the right, yielding

LEFT = 5, RIGHT = 4 DEQUE: K, A, C, S, R, L

(*g*) Since LEFT = RIGHT + 1, the array is full, and hence T cannot be added to the deque. That is, overflow has occurred.

6.24 Consider a deque maintained by a circular array with N memory cells.

(*a*) Suppose an element is added to the deque. How is LEFT or RIGHT changed?

(*b*) Suppose an element is deleted. How is LEFT or RIGHT changed?

(*a*) If the element is added on the left, then LEFT is decreased by 1 (mod N). On the other hand, if the element is added on the right, then RIGHT is increased by 1 (mod N).

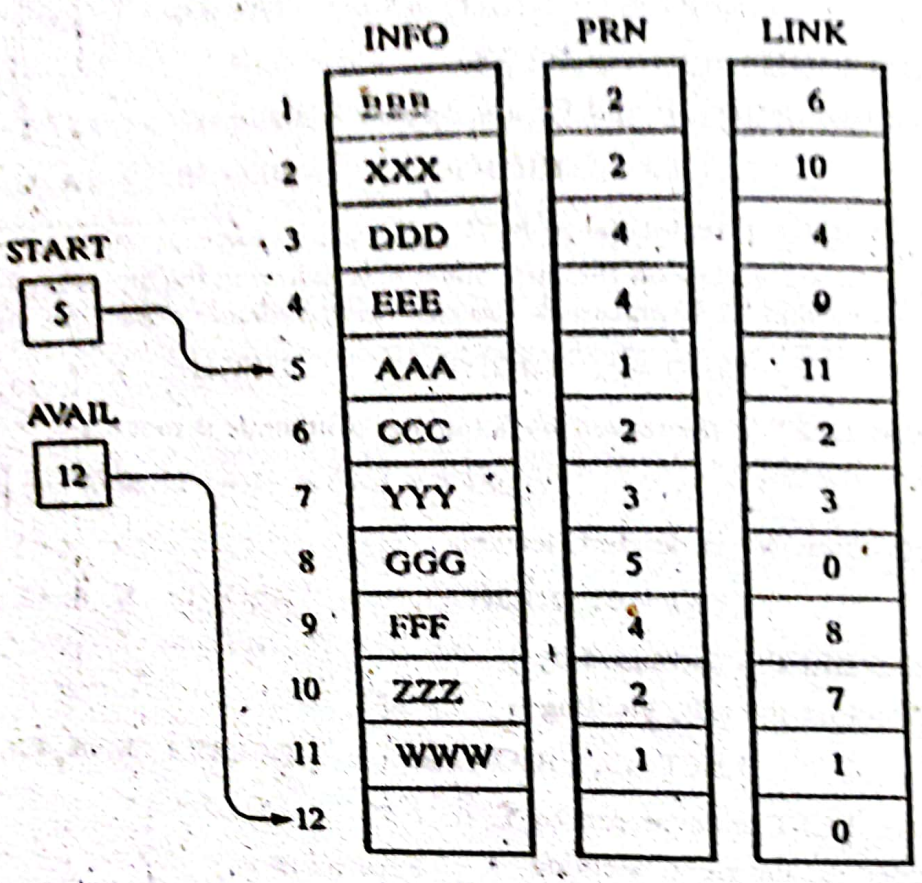
(*b*) If the element is deleted from the left, then LEFT is increased by 1 (mod N). However if the element is deleted from the right, then RIGHT is decreased by 1 (mod N). In the case that LEFT = RIGHT before the deletion (that is, when the deque has only one element), then LEFT and RIGHT are both assigned NULL to indicate that the deque is empty.

PRIORITY QUEUES

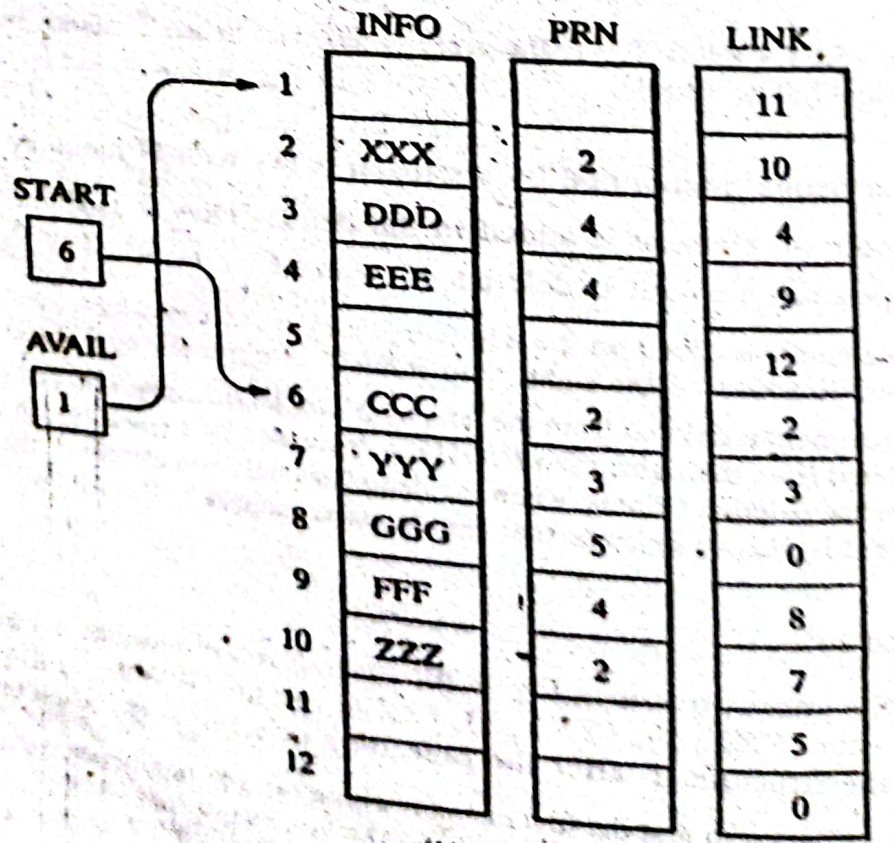
6.25 Consider the priority queue in Fig. 6-20, which is maintained as a one-way list. (*a*) Describe the structure after (XXX, 2), (YYY, 3), (ZZZ, 2) and (WWW, 1) are added to the queue. (*b*) Describe the structure if, after the preceding insertions, three elements are deleted.

(*a*) Traverse the list to find the first element whose priority number exceeds that of XXX. It is DDD, so insert XXX before DDD (after CCC) in the first empty cell, INFO[2]. Then traverse the list to find the first element whose priority number exceeds that of YYY. Again it is DDD. Hence insert YYY before DDD (after XXX) in the next empty cell, INFO[7]. Then traverse the list to find the first

element whose priority number exceeds that of ZZZ. It is YYY. Hence insert ZZZ before YYY (after XXX) in the next empty cell, INFO[10]. Last, traverse the list to find the first element whose priority number exceeds that of WWW. It is BBB. Hence insert WWW before BBB (after AAA) in the next empty cell, INFO[11]. This finally yields the structure in Fig. 6-26(*a*).



(*a*)



(*b*)

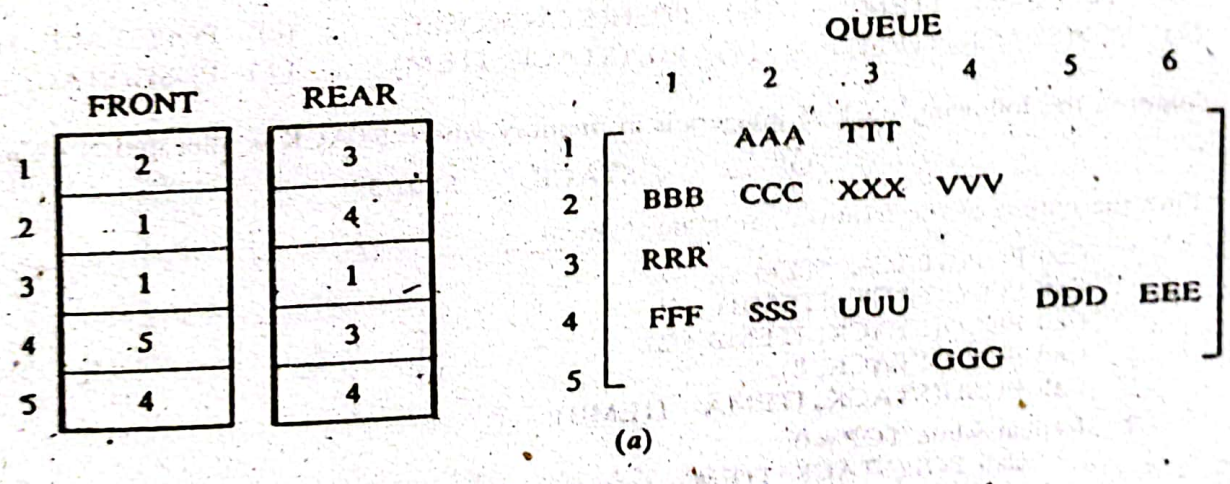
Fig. 6-26

(*b*) The first three elements in the one-way list are deleted. Specifically, first AAA is deleted and its memory cell INFO[5] is added to the AVAIL list. Then WWW is deleted and its memory cell INFO[11] is added to the AVAIL list. Last, BBB is deleted and its memory cell INFO[1] is added to the AVAIL list. This finally yields the structure in Fig. 6-26(*b*).

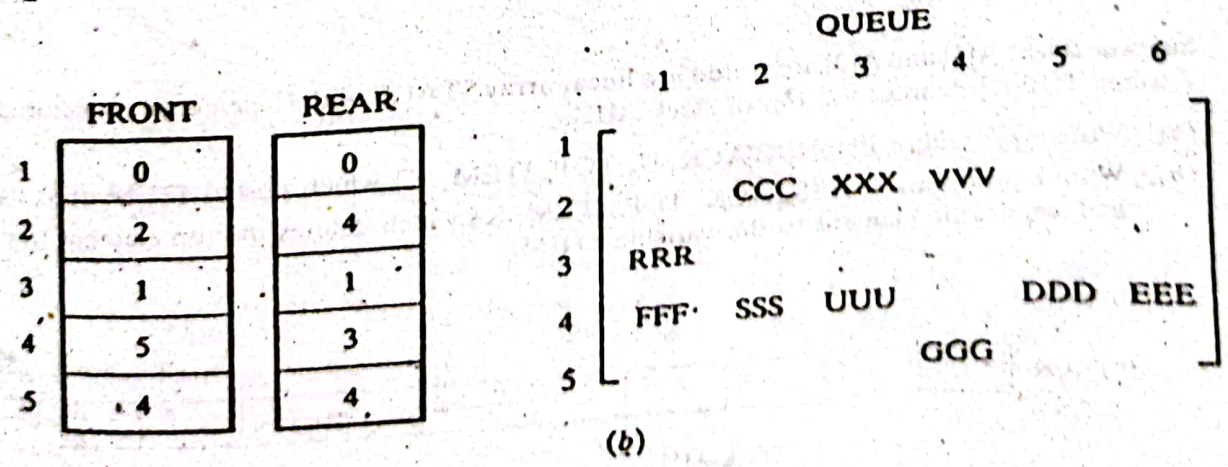
Remark: Observe that START and AVAIL are changed accordingly.

6.26 Consider the priority queue in Fig. 6-22, which is maintained by a two-dimensional array QUEUE. (*a*) Describe the structure after (RRR, 3), (SSS, 4), (TTT, 1), (UUU, 4) and (VVV, 2) are added to the queue. (*b*) Describe the structure if, after the preceding insertions, three elements are deleted.

(*a*) Insert each element in its priority row. That is, add RRR as the rear element in row 3, add SSS as the rear element in row 4, add TTT as the rear element in row 1, add UUU as the rear element in row 4 and add VVV as the rear element in row 2. This yields the structure in Fig. 6-27(*a*). (As noted previously, insertions with this array representation are usually simpler than insertions with the one-way list representation.)



(*a*)



(*b*)

Fig. 6-27

(*b*) First delete the elements with the highest priority in row-1. Since row 1 contains only two elements, AAA and TTT, then the front element in row 2, BBB, must also be deleted. This finally leaves the structure in Fig. 6-27(*b*).

Remark: Observe that, in both cases, FRONT and REAR are changed accordingly.

Supplementary Problems

STACKS

6.27 Consider the following stack of city names:

STACK: London, Berlin, Rome, Paris, _____, _____

(*a*) Describe the stack as the following operations take place:

(i) PUSH(STACK, Athens), (iii) POP(STACK, ITEM) (v) PUSH(STACK, Moscow)
(ii) POP(STACK, ITEM) (iv) PUSH(STACK, Madrid) (vi) POP(STACK, ITEM)

(*b*) Describe the stack if the operation POP(STACK, ITEM) deletes London.

6.28 Consider the following stack where STACK is allocated N = 4 memory cells:

STACK: AAA, BBB, _____, _____

Describe the stack as the following operations take place:

(*a*) POP(STACK, ITEM) (*c*) PUSH(STACK, EEE) (*e*) POP(STACK, ITEM)
(*b*) POP(STACK, ITEM) (*d*) POP(STACK, ITEM) (*f*) PUSH(STACK, GGG)

6.29 Suppose the following stack of integers is in memory where STACK is allocated N = 6 memory cells:

TOP = 3 STACK: 5, 2, 3, __, __, __

Find the output of the following program segment:

1. Call POP(STACK, ITEMA).
   Call POP(STACK, ITEMB).
   Call PUSH(STACK, ITEMB + 2).
   Call PUSH(STACK, 8).
   Call PUSH(STACK, ITEMA + ITEMB).
2. Repeat while TOP ≠ 0:
   Call POP(STACK, ITEM).
   Write: ITEM.
   [End of loop.]

6.30 Suppose stacks A[1] and A[2] are stored in a linear array STACK with N elements, as pictured in Fig. 6-28. Assume TOP[K] denotes the top of stack A[K].

(*a*) Write a procedure PUSH(STACK, N, TOP, ITEM, K) which pushes ITEM onto stack A[K].

(*b*) Write a procedure POP(STACK, TOP, ITEM, K) which deletes the top element from stack A[K] and assigns the element to the variable ITEM.

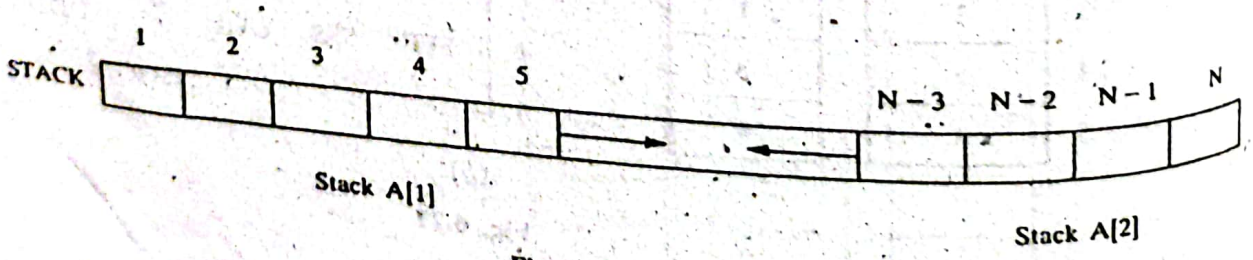


Fig. 6-28

ARITHMETIC EXPRESSIONS; POLISH EXPRESSIONS

6.31 *Translate*, by inspection and hand, each infix expression into its equivalent postfix expression:

(*a*) (A − B)/((D + E) * F) (*b*) ((A + B)/D) ↑ ((E − F) * G)

6.32 Translate, by inspection and hand, each infix expression in Prob. 6.31 into its equivalent prefix expression.

6.33 Evaluate each of the following parenthesis-free arithmetic expressions:

(*a*) 5 + 3 ↑ 2 − 8 / 4 * 3 + 6

(*b*) 6 + 2 ↑ 3 + 9 / 3 − 4 * 5

6.34 Consider the following parenthesis-free arithmetic expression:

E: 6 + 2 ↑ 3 ↑ 2 − 4 * 5

Evaluate the expression E, (*a*) assuming that exponentiation is performed from left to right, as are the other operations, and (*b*) assuming that exponentiation is performed from right to left.

6.35 Consider each of the following postfix expressions:

P_1: 5, 3, +, 2, *, 6, 9, 7, −, /, −

P_2: 3, 5, +, 6, 4, −, *, 4, 1, −, 2, ↑, +

P_3: 3, 1, +, 2, ↑, 7, 4, −, 2, *, +, 5, −

Translate, by inspection and hand, each expression into infix notation and then evaluate.

6.36 Evaluate each postfix expression in Prob. 6.35, using Algorithm 6.3.

6.37 Use Algorithm 6.4 to translate each infix expression into its equivalent postfix expression:

(*a*) (A − B)/((D + E) * F) (*b*) ((A + B)/D)↑((E − F) * G)

(Compare with Prob. 6.31.)

RECURSION

6.38 Let J and K be integers and suppose Q(J, K) is recursively defined by

$$Q(J, K) = \begin{cases} 5 & \text{if } J < K \\ Q(J - K, K + 2) + J & \text{if } J \geq K \end{cases}$$

Find Q(2, 7), Q(5, 3) and Q(15, 2).

6.39 Let A and B be nonnegative integers. Suppose a function GCD is recursively defined as follows:

$$GCD(A, B) = \begin{cases} GCD(B, A) & \text{if } A < B \\ A & \text{if } B = 0 \\ GCD(B, MOD(A, B)) & \text{otherwise} \end{cases}$$

(Here MOD(A, B), read "A modulo B," denotes the remainder when A is divided by B.) (*a*) Find GCD(6, 15), GCD(20, 28) and GCD(540, 168). (*b*) What does this function do?

6.40 Let N be an integer and suppose H(N) is recursively defined by

$$H(N) = \begin{cases} 3 * N & \text{if } N < 5 \\ 2 * H(N - 5) + 7 & \text{otherwise} \end{cases}$$

(*a*) Find the base criteria of H and (*b*) find H(2), H(8) and H(24).

6.41 Use Definition 6.3 (of the Ackermann function) to find $A(2, 2)$.

6.42 Let M and N be integers and suppose F(M, N) is recursively defined by

$$F(M, N) = \begin{cases} 1 & \text{if } M = 0 \text{ or } M = N = 1 \\ F(M-1, N) + F(M-1, N-1) & \text{otherwise} \end{cases}$$

(*a*) Find F(4, 2), F(1, 5) and F(2, 4). (*b*) When is F(M, N) undefined?

6.43 Let A be an integer array with N elements. Suppose X is an integer function defined by

$$X(K) = X(A, N, K) = \begin{cases} 0 & \text{if } K = 0 \\ X(K-1) + A(K) & \text{if } 0 < K \le N \\ X(K-1) & \text{if } K > N \end{cases}$$

Find X(5) for each of the following arrays:

(*a*) N = 8, A: 3, 7, −2, 5, 6, −4, 2, 7 (*b*) N = 3, A: 2, 7, −4

What does this function do?

6.44 Show that the recursive solution to the Towers of Hanoi problem in Sec. 6.7 requires $f(n) = 2^n - 1$ moves for n disks. Show that no other solution uses fewer than $f(n)$ moves.

6.45 Suppose S is a string with N characters. Let SUB(S, J, L) denote the substring of S beginning in the position J and having length L. Let A//B denote the concatenation of strings A and B. Suppose REV(S, N) is recursively defined by

$$REV(S, N) = \begin{cases} S & \text{if } N = 1 \\ SUB(S, N, 1)//REV(SUB(S, 1, N-1), N-1) & \text{otherwise} \end{cases}$$

(*a*) Find REV(S, N) when (i) N = 3, S = *abc* and (ii) N = 5, S = *ababc*. (*b*) What does this function do?

QUEUES; DEQUES

6.46 Consider the following queue where QUEUE is allocated 6 memory cells:

FRONT = 2, REAR = 5 QUEUE: _____, London, Berlin, Rome, Paris, _____

Describe the queue, including FRONT and REAR, as the following operations take place: (*a*) Athens is added, (*b*) two cities are deleted, (*c*) Madrid is added, (*d*) Moscow is added, (*e*) three cities are deleted and (*f*) Oslo is added.

6.47 Consider the following deque where DEQUE is allocated 6 memory cells:

LEFT = 2, RIGHT = 5 DEQUE: _____, London, Berlin, Rome, Paris, _____

Describe the deque, including LEFT and RIGHT, as the following operations take place:

(*a*) Athens is added on the left.
(*b*) Two cities are deleted from the right.
(*c*) Madrid is added on the left.
(*d*) Moscow is added on the right.
(*e*) Two cities are deleted from the right.
(*f*) A city is deleted from the left.
(*g*) Oslo is added on the left.

6.48 Suppose a queue is maintained by a circular array QUEUE with N = 12 memory cells. Find the number of elements in QUEUE if (*a*) FRONT = 4, REAR = 8; (*b*) FRONT = 10, REAR = 3; and (*c*) FRONT = 5, REAR = 6 and then two elements are deleted.

6.49 Consider the priority queue in Fig. 6-26(*b*), which is maintained as a one-way list.

(*a*) Describe the structure if two elements are deleted.
(*b*) Describe the structure if, after the preceding deletions, the elements (RRR, 3), (SSS, 1), (TTT, 3) and (UUU, 2) are added to the queue.
(*c*) Describe the structure if, after the preceding insertions, three elements are deleted.

6.30 Consider the priority queue in Fig. 6-37(*b*), which is maintained by a two-dimensional array QUEUE.

(*a*) Describe the structure if two elements are deleted.
(*b*) Describe the structure if, after the preceding deletions, the elements (*JJJ*, 3), (KKK, 1), (LLL, 4) and (MMM, 3) are added to the queue.
(*c*) Describe the structure if, after the preceding insertions, six elements are deleted.

Programming Problems

6.31 Translate Quicksort into a subprogram QUICK(A; N) which sorts the array A with *N* elements. Test the program using

(*a*) 44, 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66
(*b*) D, A, T, A, S, T, R, U, C, T, U, R, E, S

6.32 Write a program which gives the solution to the Towers of Hanoi problem for *n* disks. Test the program using (*a*) $n = 3$ and (*b*) $n = 4$.

6.33 Translate Algorithm 6.4 into a subprogram POLISH(Q, P) which transforms an infix expression Q into its equivalent postfix expression P. Assume each operand is a single alphabetic character, and use the usual symbols for addition (+), subtraction (−), multiplication (*) and division (/), but use the symbol ↑ or $ for exponentiation. (Some programming languages do not accept ↑.) Test the program using
(*a*) ((A + B) * D) $ (E − F) (*b*) A + (B * C − (D / E $ F) * G) * H

6.34 Suppose a priority queue is maintained as a one-way list as illustrated in Fig. 6-20.

(*a*) Write a procedure

INSPQL(INFO, PRN, LINK, START, AVAIL, ITEM, N)

which adds an ITEM with priority number N to the queue. (See Algorithm 6.14.)

(*b*) Write a procedure

DELPQL(INFO, PRN, LINK, START, AVAIL, ITEM)

which removes an element from the queue and assigns the element to the variable ITEM. (See Algorithm 6.13.)

Test the procedures, using the data in Prob. 6.25.

6.35 Suppose a priority queue is maintained by a two-dimensional array as illustrated in Fig. 6-22.

(*a*) Write a procedure

INSPQA(QUEUE, FRONT, REAR, ITEM, M)

which adds an ITEM with priority number M to the queue. (See Algorithm 6.16.)

(*b*) Write a procedure

DELPQA(QUEUE, FRONT, REAR, ITEM)

which removes an element from the queue and assigns the element to the variable ITEM. (See Algorithm 6.15.)

Test the procedures, using the data in Prob. 6.26. (Assume that QUEUE has ROW number of rows and COL number of columns, where ROW and COL are global variables.)