Stack

A stack is a linear structure in which items may be added or removed only at one end. *Figure 1.12* pictures three Every day examples of such a structure: a stack of dishes, a stack of plate and a stack of folded towels.

Stacks are also called last-in first-out (LIFO) lists. Other names used for stacks are "Piles" and "push-down lists. Stack has many important applications in computer science. The notion of *recursion* is fundamental in computer science. One way of simulating recursion is by means of stack structure. Let we learn the operations which are performed by the 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, that elements which are inserted last will be removed first. Special terminology is used for two basic operation associated with stacks.

Pictorial Representation:

Suppose that following 6 elements are pushed, in order, onto an empty stack:
AAA, BBB, CCC, DDD, EEE, FFF

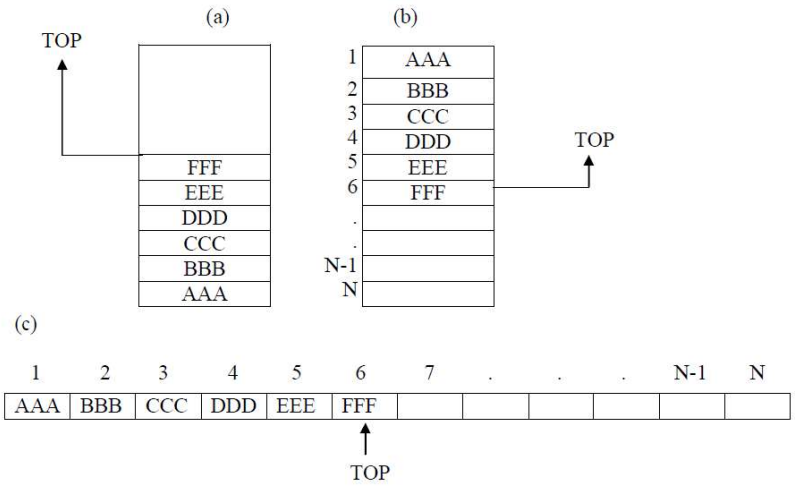


Figure 1.13 (a), (b) & (c)

Representation of Stack :

1. By using Linear Array
2. By using Linked List:

1. By Using Linear Array(**ARRAY REPRESENTATION OF STACKS)**

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 1.14 pictures such an array representation of a stack. Since TOP=3, the stacks has three elements, XXX, YYY and ZZZ; and since MAXSTK=8, there is room for 5 more items in the stack.

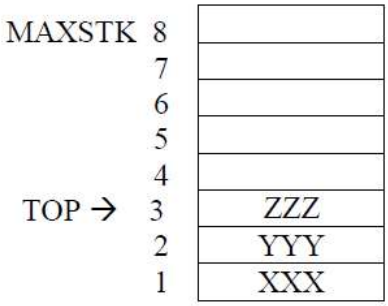


Figure 1.14

The operation of adding (pushing) an item onto a stack and the operation of removing (popping) and item from a stack are implemented by the following algorithm called PUSH and POP respectively. When we adding a new element, first, we must test whether there is a free space in the stack for the new item; if not, then we have the condition known as *overflow*.

Stacks implemented as arrays are useful if a fixed amount of data is to be used. However, if the amount of data is not a fixed size or the amount of the data fluctuates widely during the stack's life time, then an array is a poor choice for implementing a stack. For example, consider a call stack for a recursive procedure. First, it can be difficult to know how many times a recursive procedure will be called, making it difficult to decide upon array bounds. Second, it is possible for the recursive procedure to sometimes be called a small number of times, called a large number of times at other times. An array would be a poor choice, as you would have to declare it to be large enough that there is no danger of it running out of storage space when the procedure recurses many times. This can waste a significant amount of memory if the procedure normally only recurses a few times

Stack Terminology

1. MAXSIZE or N: Size of Stack, Number of element can store
2. TOP: index value of last element in stack
3. UnderFlow: No element in stack
4. Overflow: No space for inserting a new item in stack

Basic Operation in Stack

1.Push
2.Pop

Push

This operation adds or pushes another item onto the stack. Before insert any item in to stack we first check whether stack is full or not , if stack is no full than we insert the any item.

Algorithm:

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.

Example

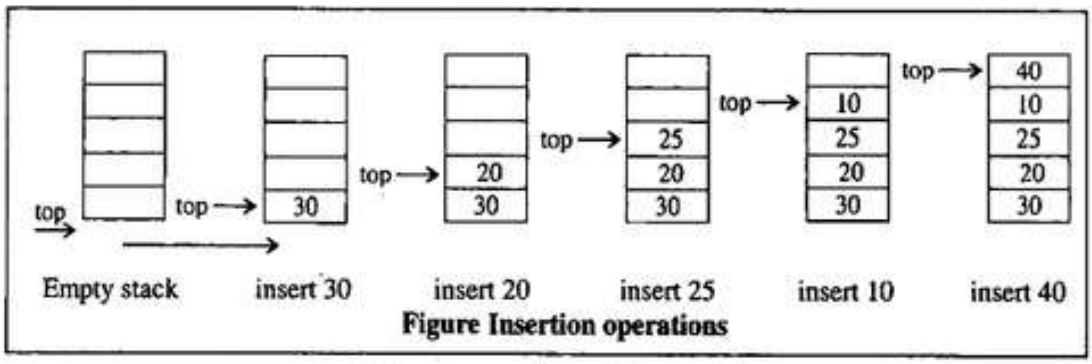


Figure Insertion operations

Fig. Insertion Operations

POP

This operation delete or remove item from top of the stack, before performing pop operation we will first check stack is empty or not, if stack is not empty we perform the operation.

Algorithm

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.

Example

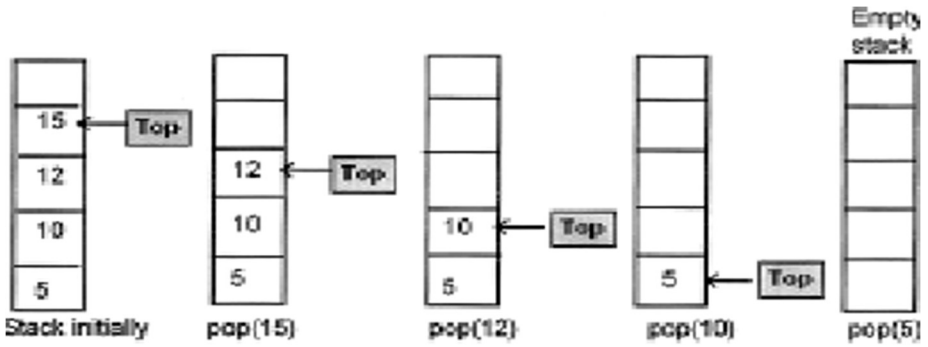


Fig.5.3: *Pop* operation with stack

Q.Perform the following operation on stack and given: MaxSize =5 , Top=2 and element A,B

Push(C),Push(D),Push(E),Push(F),Pop(),Pop(),Pop(),Push(G),Push(H)Pop(),Pop(),Pop(),Pop()

INDEX	1	2	3	4	5	
Push(C)	A	B	C			Top=3
Push(D)	A	B	C	D		Top=4
Push(E)	A	B	C	D	E	Top=5
Push(F)	A	B	C	D	E	Top=5 And Overflow
Pop()	A	B	C	D		Top=4
Pop()	A	B	C			Top=3
Pop()	A	B				Top=2
Push(G)	A	B	G			Top=3
Push(H)	A	B	G	H		Top=4
Pop()	A	B				Top=2
Pop()	A					Top=1
Pop()						Top=0
Pop()						Top=0 And Underflow

Stack Implementation By using Linked List

Top

Node Architecture

```
struct node
{
 int info;
 struct node *link;
} *top = NULL;
```

Push(item)

Dynamically: Push(item)	Statically: Push(INFO,LINK,TOP,AVAIL)

Step 1. [To check Overflow]
New1= (struct node *)malloc(sizeof(struct node));
If (New1==NULL)
Write : "Overflow" and return.
 	
Step 2. [Case 1 if stack is empty]
If (top == NULL) Then
top = new1
new1->link=NULL
Else
[case 2 already element in stack link in first]
New1->link=top
top= New1
[End if else]
 	
Step 3 [add item]
top->info = item;
 	
Step 4 Exit
	Step1. [To check Overflow]
	If (AVAIL==-1)
	Write:"Overflow" and Return
	
	Step 2. [Remove node from Avail list]
	New1=AVAIL
	AVAIL=LINK[AVAIL]
	
	Step 3. [add node in stack]
	[Case 1 if stack empty]
	If TOP==-1
	TOP=New1
	LINK[New1]=-1
	Else
	[Case 2 if stack not empty]
	LINK[NEW]=TOP
	TOP=NEW
	
	Step 4. [add item]
	INFO[NEW]=ITEM
	
	Step5. Exit.

Application Of Stack

1. **Arithmetic Expression; Polish Notation**
   There are three type of notation
   a. Infix
   b. PreFix(polish notation)
   c. PreFix(Reverse polish notaion)

Infix Expression

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

A + B C – D E * F G / H

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

(A + B) * C and A + (B * C)

by using either parentheses or 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.

PreFix Expression

Polish notation named after the polish mathematician Jan Lukasiewicz, refer to the notation in which the operator symbol is placed before its two operand.

+AB -CD *EF

Postfix Expression

Reverse Polish Notation refer to the analogous notation in which the operator symbol is placed after its two operand.

AB+ CD- EF*

Evaluation of Postfix Expression

Suppose P is an arithmetic expression written in postfix notation. The following *algorithm 1.3* uses a STACK to hold operands, evaluates P.

This algorithm finds the VALUE of an arithmetic expression P written in postfix notation.
Step 1. Add a right parenthesis ")"at the end of P. [This acts as a sentinel].
Step 2. Scan P from left to right and repeat Steps 3 and 4 for each element of until the sentinel ")" is encountered.
Step 3. If an operand is encountered, put it on STACK.
Step 4. If an operator (x) 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 (x) A.
c) Place the result of (b) back on STACK
[End of If structure.]
[End of Step 2 loop.]
Step 5. Set VALUE equal to the top element on STACK.
Step 6. Exit.

Example

P: 5 6 2 + * 12 4 / -

Step 1: [add ")" at end of P :] 5 6 2 + * 12 4 / -)

So we have

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

SN.	Symbol Scan	Stack	Remark
1	5	5	Operand encounter so push it on to stack so Top=1
2	6	5 6	Operand encounter so push it on to stack so Top=2
3	2	5 6 2	Operand encounter so push it on to stack so Top=3
4	+	5 8	Operator encounter so pop top two element from stack and perform operation and push the result(6+2=8) on to stack.
5	*	40	Again Operator encounter so pop top two element from stack and perform operation and result(5*8=40) get back to stack
6	12	40 12	Operand encounter so push it on to stack
7	4	40 12 4	Operand encounter so push it on to stack
8	/	40 3	
9	-	37	
10	)		) occur indicate expression is over so pop result from stack and print

Example 2.

A B C D + + * C D A - + *

Where A=2, B=4, C=3, D=5

P: 2 4 3 5 + + * 3 5 2 - + *
Step 1: add ")" at end of P

P:	2	4	3	5	+	+	*	3	5	2	-	+	*
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)

SN.	Symbol Scan	Stack	Remark
1	2	2	Operand encounter so push it on to stack so Top=1
2	4	2 4	Operand encounter so push it on to stack so Top=2
3	3	2 4 3	Operand encounter so push it on to stack so Top=3
4	5	2 4 3 5	Operand encounter so push it on to stack so Top=4
5	+	2 4 8	Operator encounter so pop top two element from stack and perform operation and push the result(3+5=8) on to stack. Top=3
6	+	2 12	Operator encounter so pop top two element from stack and perform operation and push the result(4+8=12) on to stack. Top=2
7	*	24	Operator encounter so pop top two element from stack and perform operation and push the result(2*12=24) on to stack. Top=1
8	3	24 3	Operand encounter so push it on to stack so Top=2
9	5	24 3 5	Operand encounter so push it on to stack so Top=3
10	2	24 3 5 2	Operand encounter so push it on to stack so Top=4
11	-	24 3 3	Operator encounter so pop top two element from stack and perform operation and push the result (5-2=3) on to stack.
12	+	24 6	Operator encounter so pop top two element from stack and perform operation and push the result(3+3=6) on to stack.
13	*	144	Operator encounter so pop top two element from stack and perform operation and push the result(26*6=144) on to stack.
14	)		Ending of Postfix expression so pop the result : 144//

Transforming Infix to Postfix Expression

1. By General Method
2. By using stack

1. **By General Method:**

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

1	Exponentiation (↑ , $, ^)	R->L	If more than one exponent sign will occur fallow Right to Left i.e. first solve Right exponent
2	Multiplication (*) and division (/)	L->R	Fallow Left to right
3	Addition (+) and subtraction (-)	L->R	Fallow Left to right

Example: A+B*C-D/E^F

A+B*C-D/E^F	
1. A+B*C-D/E^F	Perform ^
2. A+B*C-D/EF^	Here * and / are same precedence so fallow R->L so in Right we have *
3. A+BC*- D / EF^	Now perform /
4. A+BC* - DEF^/	Here + and - are same precedence so fallow R->L so in Right we have +
5. ABC*+ - DEF^/	Now perform -
6. ABC*+ DEF^/-	

Example :

5*(6+2)-12/4	**(A+B)*C/D+E^F/G**	**6/2^3+4^5+5*9^7/4**
Soln:- 1. 5*(6+2)-12/4 2. 5 * (6+2) - 12 / 4 3. 5* 6 2 + - 12 / 4 4. 5 6 2 + * - 12 / 4 5. 5 6 2 + * - 12 4 / 6. 5 6 2 + * 12 4 / -	Soln: **(A+B)*C/D+E^F/G** 1. AB+ * C / D + E ^ F / G 2. AB+ * C / D + E F^ / G 3. AB+ C* / D + E F^ / G 4. AB+C *D/ + E F^ / G 5. AB+C*D/ + E F^ G/ 1. AB+C *D/ E F^ G/+ // Ans	Soln: 1. 6 / 2 ^ 3 + 4 ^ 5 + 5 * 9 ^ 7 / 4 2. 6 / 2 ^ 3 + 4 ^ 5 + 5 * 97^ / 4 3. 6 / 2 ^ 3 + 4 5^ + 5 * 9 7^ / 4 4. 6 / 2 3^ + 4 5^ + 5 * 9 7^ / 4 5. 6 2 3^/ + 4 5^ + 5 * 9 7^ / 4 6. 6 2 3^/ + 4 5^ + 5 9 7^ * / 4 7. 6 2 3^/ + 4 5^ + 5 9 7^ * 4/ 8. 6 2 3^/ + 4 5^ + 5 9 7^ * 4/ 9. 6 2 3^/ 4 5^+ + 5 9 7^ * 4/ 10. 6 2 3^/ 4 5^+ 5 9 7^ * 4/+

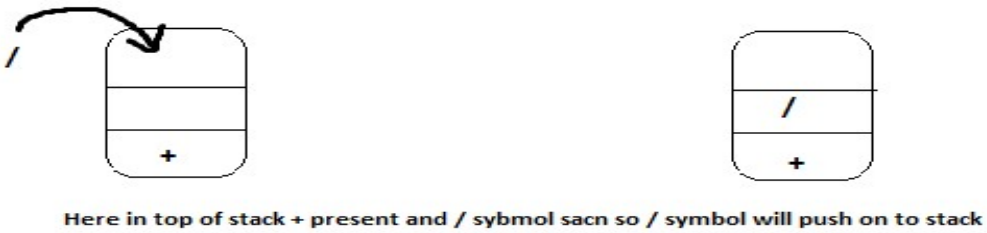
2. By Using Stack:

Let Q be an arithmetic expression written in infix notation. 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.

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 (x) 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 (x)
   b) Add (x) 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 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.

Important Note:



Here in top of stack + present and / sybmol sacn so / symbol will push on to stack

Here in top / present and * symbol scan, but precedence of * is equal to / so first pop then push / on to stack

Example: Consider the following arithmetic infix expression
Q: A + (B * C - (D / E ↑ F) * G) * H

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 * +

Example2.

A+B+C^D^E+(F^G^H)/I

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

SN.	Symbol	Stack	Expression Postfix
1	A	(	A
2	+	(+	A
3	B	(+	AB
4	+	(+	AB+
5	C	(+	AB+C
6	^	(+ ^	AB+C
7	D	(+ ^	AB+CD
8	^	(+ ^	AB+CD^
9	E	(+ ^	AB+CDE^
10	+	(+	AB+CDE^^+
11	(	(+(	AB+CDE^^+
12	F	(+(	AB+CDE^^+F
13	^	(+(^	AB+CDE^^+F
14	G	(+(^	AB+CDE^^+G
15	^	(+(^	AB+CDE^^+G^
16	H	(+(^	AB+CDE^^+G^H
17	)	(+	AB+CDE^^+G^H+
18	/	(+/	AB+CDE^^+G^H+
19	I	(+/	AB+CDE^^+G^H+I
20	)		AB+CDE^^+G^H+I/+

Infix to prefix conversion

1. By General Method
2. By using Stack

1. By General Method:

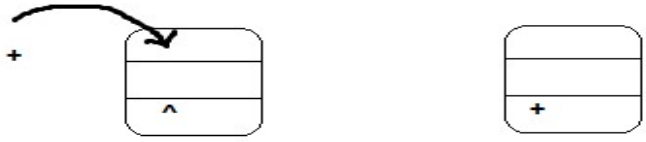
a) A+B*C	**b) A+B*C-D/E^F**	**c) A+B/C*D E-F*G/H**
1. A + B * C 2. A + *BC 3. +A*BC	1. A + B * C - D / E ^ F 2. A + B * C - D / ^ EF 3. A + *B C - D / ^ EF 4. A + *B C - /D^ EF 5. +A *B C - /D^ EF 6. - +A *B C /D^ EF	1. (A + B / C * D) / (E – F * G / H) 2. (A + / B C * D) / (E – F * G / H) 3. (A + */ B C D) / (E – F * G / H) 4. (+A*/ B C D) / (E – F * G / H) 5. (+A*/ B C D) / (E – *FG / H) 6. (+A*/ B C D) / (E – /*FG H) 7. (+A*/ B C D) / (- E /*FG H) 8. /+A*/ B C D - E /*FG H
d) **5*(6+2)-12/4**	e) **(A+B)*C/D+E^F/G**	f) **6/2^3+4^5+5*9^7/4**
1. 5 * (6 + 2) – 12 / 4 2. 5 * +6 2 – 12 / 4 3. * 5+ 6 2 – 12 / 4 4. * 5+ 6 2 – / 12 4 5. - * 5+ 6 2 / 12 4		

2. **By Using Stack:**

Step 1. Reverse the input string
Step 2. Examine the next element in the input
Step 3. If it is operand add it to the output string
Step 4. If it is closing parenthesis push it on stack
Step 5. If it is an operator then
  a. If stack is empty, push operation on stack
  b. If the top of stack is ")" push operator on stack
  c. If it has same or higher priority then the top of stack, push it
  d. Else pop the operator & add it to output string, repeat 5
Step 6. If it is "(" pop operator and add them to string s until a ")" is encountered. POP and discard ")"
Step 7. If there is more input go to step 2
Step 8. If there is no more input, unstack the remaining operators & add them
Step 9. Reverse the output string

Important Notes:

Here in TOS we have + operator which is equal precedence with - operator so only perform push operation



Here in TOS we have ^ operator which is higher precedence with + operator so perform first POP and then Push operation

Example: P: A+B*(C+D)

Step 1: Reverse the Expression:

P:	)	D	+	C	(	*	B	+	A
	1	2	3	4	5	6	7	8	9

SN	Scan	Stack	Expression
1	)	)	
2	D	)	D
3	+	)+	D
4	C	)+	DC
5	(		DC+
6	*	*	DC+
7	B	*	DC+B
8	+	+	DC+B*A
9	A	+	DC+B*A
			DC+B*A+
	Now Reverse the resultant expression **+A*B+CD** **i**s Answer		

Recursion

Recursion is an important concept in computer science. Many algorithms can be best described in terms of recursion. Let we discuss, how recursion may be useful tool in developing algorithms for specific problems. Consider 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*. 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.

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 defined that 0! = l. Thus we have,

Definition: (Factorial Function)

If $n = 1$, then $n = 1$.

If $n > 1$, then $n! = n \cdot (n-1)!$

OR

$$F(n) = \begin{cases} 1 & \text{if } n==1 \\ n*F(n-1) & \text{if } n>1 \end{cases}$$

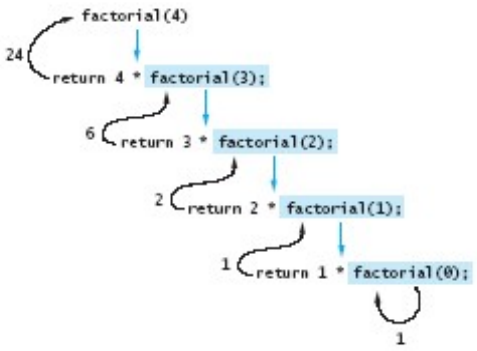
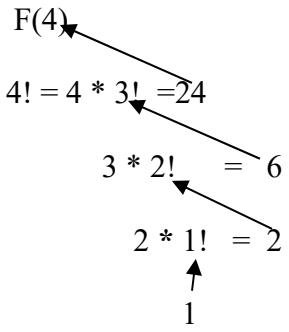


FIGURE 7.5
Trace of factorial(4)



Tower of Hanoi Problem

Suppose three pegs (Towers) A, B, C are given and suppose on peg A there are placed a finite number n of disks with decreasing size. This picture include 3 disks, n=3. 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.

Algorithm: 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]

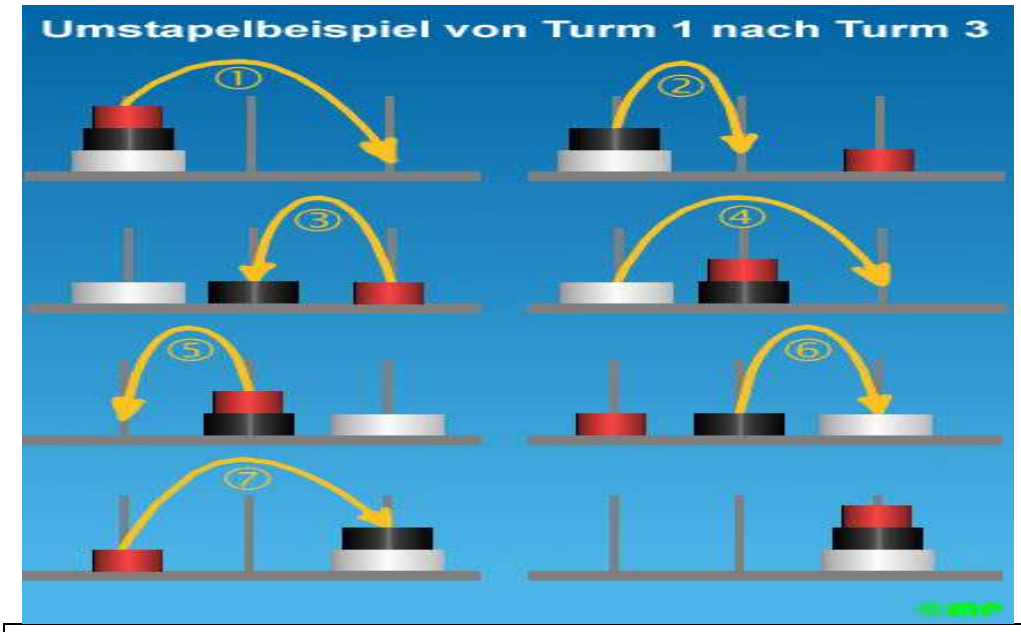
Call Tower (N-1, BEG, END, AUX).

3. Write: BEG->END.

4. [Move N-1 disks from peg AUX to peg END]

Call TOWER (N-1, AUX, BEG, END).

Return.




```
C program:
int count=0;
void tower_of_hanoi(int n,char beg,char aux,char end)
{
if(n==1)
{
printf("%c --> %c\n\n",beg,end);
count++;
}
else
{
tower_of_hanoi(n-1,beg,end,aux);
tower_of_hanoi(1,beg,aux,end);//or printf("%c --> %c\n\n",beg,end);

tower_of_hanoi(n-1,aux,beg,end);
}
}
void main()
{
int n;
clrscr();
printf("Enter the number of disks\n");
scanf("%d",&n);
tower_of_hanoi(n,'A','B','C');
printf("No of Steps are %d",count);
getch();
}
```

Example: Tower of Hanoi for n=3 **Steps:**

Tower (3, A, B, C)
- Tower(2, A, C, B)
  - Tower (1, A, B, C) ---------- A→C
  - ---------- A→B
  - Tower (1, C, A, B)---------- C→B
- ---------- A→C
- Tower (2, B, A, C)
  - Tower(1,B,A,C)----------B→ C
  - ---------- B→C
  - Tower (1, A, B, C) ---------- A→C

Minimum Movement required to solve Problem is 2^N -1

Example: If Number of disk is 5 calculate min movement required to solve tower of Hanoi problem

Ans: Here N=5

So Minimum Movement $=2^N-1$

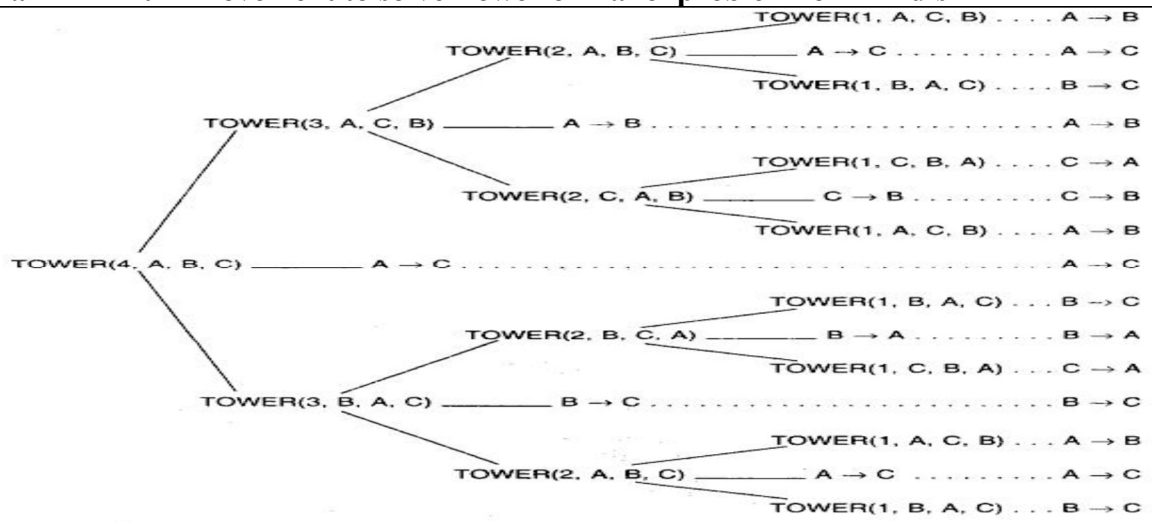
$= 2^5-1$

$=32-1$

$=31$

Question

Write all minimum movement to solve Tower of Hanoi problem for n=4 disk



for $n = 4$ disks consists of the following 15 moves:

$A \to B \quad A \to C \quad B \to C \quad A \to B \quad C \to A \quad C \to B \quad A \to B \quad A \to C$

$B \to C \quad B \to A \quad C \to A \quad B \to C \quad A \to B \quad A \to C \quad B \to C$

Question

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 (3355, 7)

(a) Q (2,3) = 0 since 2 < 3

Given Q(a,b) = Q(a-b,b) + 1 if b ≤a

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 (∴ Q(2,3) = 0)

= 4

(b) Each time b is subtracted from a, the value of Q is increased by 1. Hence Q(a,b) finds the integer quotient when a is divided by b. Thus Q(3355, 7) = 479

Advantages of Recursion

1. Although at most of the times a problem can be solved without recursion, but in some situations in programming, it is a must to use recursion. For example, a program to display a list of all files of the system

cannot be solved without recursion.
2. The recursion is very flexible in data structure like stacks, queues, linked list and quick sort.
3. Using recursion, the length of the program can be reduced

Disadvantages of Recurssion

1. It requires extra storage space. The recursive calls and automatic variables are stored on the stack. For every recursive calls separate memory is allocated to automatic variables with the same name.
2. If the programmer forgets to specify the exit condition in the recursive function, the program will execute out of memory. In such a situation user has to press Ctrl+ break to pause and stop the function.
3. The recursion function is not efficient in execution speed and time.
4. If possible, try to solve a problem with iteration instead of recursion

Queue

A queue is a linear structure in which element may be inserted at one end called the *rear*, and the deleted at the other end called the *front*. *In* pictures a queue of people waiting at a bus stop. Queues are also called first-in first-out (FIFO) lists. An important example of a queue in computer science occurs in a timesharing system, in which

programs with the same priority form a queue while waiting to be executed. Similar to stack operations, operations that are define a queue.

Type of Queue:

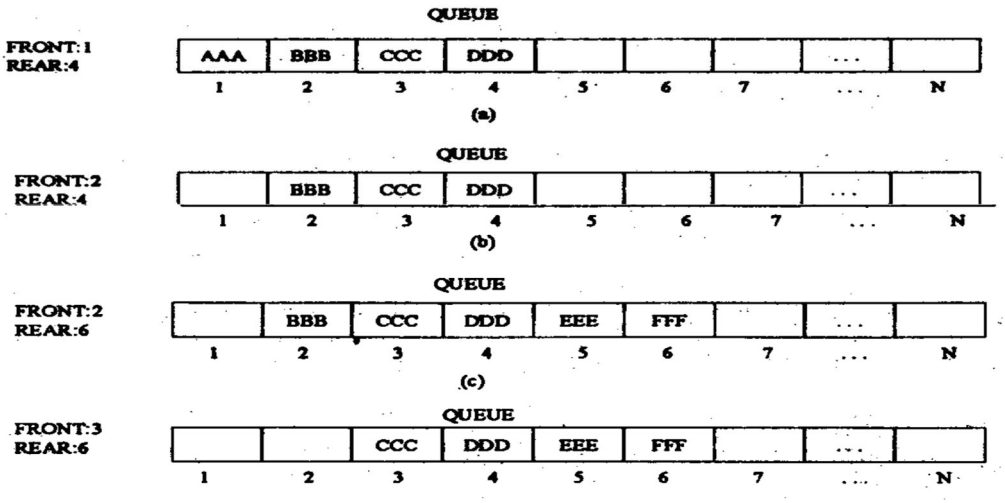
1. Linear Queue
2. Circular Queue
3. DEQUE
4. Priority Queue

REPRESENTATION OF QUEUES

Queues may be represented in the computer in various ways, usually by means of **one way lists** or **linear arrays**.

By Using Linear Array

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.



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 FRONT := 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 REAR := REAR + 1

Assume 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,

QUEUE[REAR]:= ITEM

Similarly, if FRONT=N and an element of QUEUE is deleted, we reset FRONT=1 instead of increasing FRONT to N+1. Suppose that our queue contains only one element, i.e., suppose that FRONT = REAR = NULL

Linear Queue:

In linear Queue we can insert the element from one end called Rear and we can delete the element only from Front end.The condition of overflow is when Rear(R)= =N.

Number of element in Linaer Queue= R-F+1

Basic operation in Linear Queue

1. Insertion
2. Deletion

Algorithm: **Insert_LQ(LQ,F,R,N,Item)**

Here LQ is Linear Queue, F is front value, R is Rear Value , N is Size of LQ and Item is element that we want to insert.

Step 1: [Check OverFlow]
If R equal to N Then:
Write "OverFlow"
Return

	B	C	D	E	F
1	2	3	4	5	6

Front:2 and Rear: 6
N=6 here R==N so Queue is "Overflow"

Step 2: [Check Queue is Empty]
If R==0 Then:
Set R=1
Set F=1
Else
Set R=R+1

1	2	3	4	5	6

Front:0 and Rear:0
So After Insertion Front and Rear will become 1.

Other Wise

	B	C			
1	2	3	4	5	6

Step 3:[Insert item]
LQ[R]=Item

Front :2 and Rear:3
So After Insertion Front and Rear will become 3+1= 4

Step 4: Exit

Algorithm: **Delete_LQ((LQ,F,R,N,Item)**

Here LQ is Linear Queue, F is front value, R is Rear Value , N is Size of LQ and deleted element will store in item.

Step 1: [Check UnderFlow]
If F equal to 0 Then:
Write "UnderFlow"
Return

UnderFlow Condition

Here F=0 and R=0 so Queue is empty so we can not perform delete operation this is underflow condition.

Step 2:[Store in Item]
Item=LQ[F]

Step 3 : [Check Single element]
If F==R Then:
Set R=0
Set F=0
Else
Set F=F+1

Single Element

		F			
1	2	3	4	5	6

Here R=3 and F=3 So Single element so after deletion F =0 and R=0

Step 4: Exit

Other wise

	B	F	G		
1	2	3	4	5	6

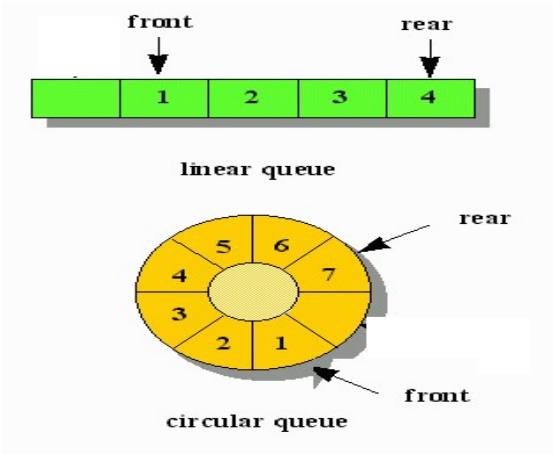
Here F=2 and R=4 after deletion F=3

Circular Queue:

Circular queue is another form of a linear queue in which the last position is connected to the first position of the list. The circular queue is similar to linear queue has two ends, the front end and the rear end. The rear end is where we insert elements and front end is where we delete elements. You can traverse in a circular queue in only one direction.

Initially the front and rear ends are at same positions. When you insert elements the rear pointer moves one by one until the front end is reach end. If the next position of the rear is front, the queue is said to be fully occupied. Beyond this you cannot insert any data. But if you delete any data, you can insert the data accordingly.

When you delete the elements the front pointer moves one by one until the rear pointer is reached. If the front pointer reaches the rear pointer, both their positions are initialized to -1. And the queue is said to be empty.



Insertion:

INSERT_CQ (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.

[check overflow]

A	B	C	D	E
1	2	3	4	5

Here suppose F=1 and R=5
So its overflow condition

G	H	C	D	E	F
1	2	3	4	5	6

Here suppose F=3 and R=2
So its again overflow condition

[otherwise]

		A	B	C
1	2	3	4	5

Here suppose F=3 and R=5
here R=5 then after insertion R will become 1.

D		A	B	C
1	2	3	4	5

So Here R=1 and F=3

Deletion

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

[check underflow]

1	2	3	4	5

Here F=0 and R=0
So its Underflow condition we cant perform delete op^{n}.

			G		
1	2	3	4	5	6

Here suppose F=4 and R=4 // means only 1 item exist
So after deletion

1	2	3	4	5	6

F=0 and R=0

[otherwise]

D	E			C
1	2	3	4	5

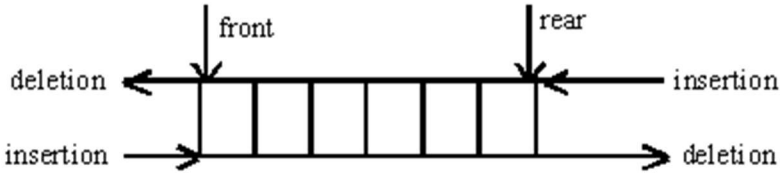
Here suppose F=5 and R=2
here F=5 so after deletion F will become 1.

D	E			
1	2	3	4	5

So Here R=2 and F=1

DEQUES

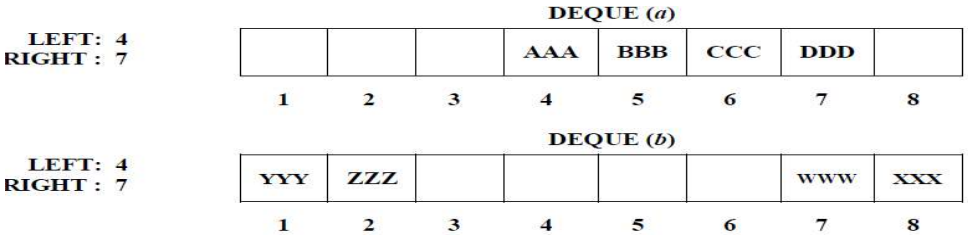
A deque (pronounced either "**deck**" or "**dequeue**") is a linear list in which elements can be added or removed at either end but not in the middle. The term deque refers to the name double-ended queue.



There are two variations of a deque - namely, an ***input-restricted deque*** and an ***outputrestricted deque*** - which are intermediate between a deque and a queue. 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 buy allows insertions at both ends of the list.

A deque is commonly implemented as a circular array with two variables LEFT and RIGHT taking care of the active ends of the deque. Example . illustrates the working of a deque with insertions and deletions permitted at both ends.

In 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.



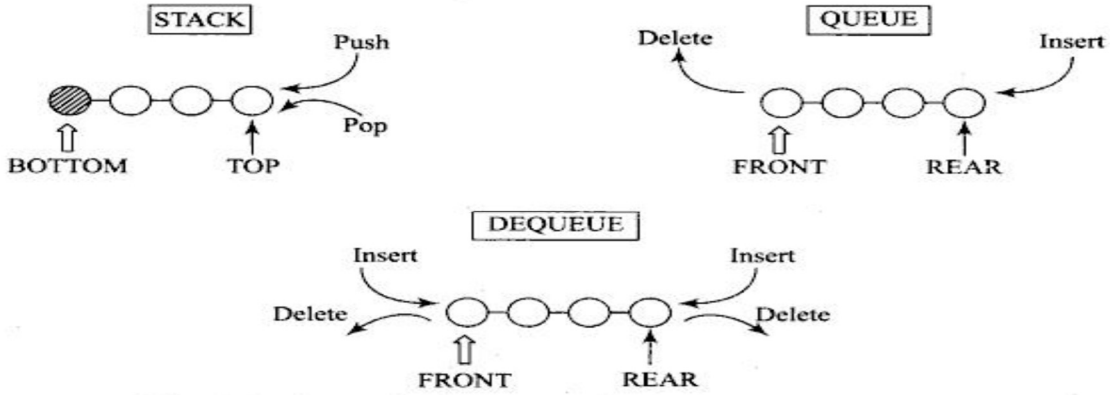


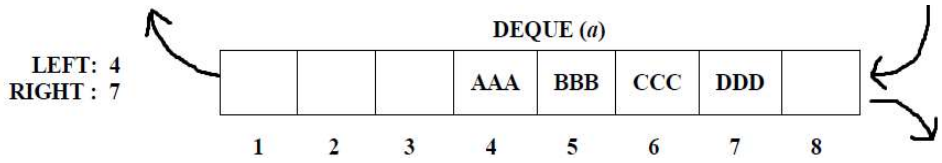
Fig. 5.8 *A stack, a queue and a deque—a comparison*

There are four algorithm used to perform insertion and deletion operation in DEQ.

1. Insertion of element at left side(Front end) of Deq
2. Insertion of element at Right side(Rear end) of Deq
3. Deletion of element from Left side(Front end) of Deq
4. Deletion of element from Right side(Rear end) of Deq

Input Restricted DEQUE:

input restricted Deq allow insertions at only one end. And deletion can perform from both end.

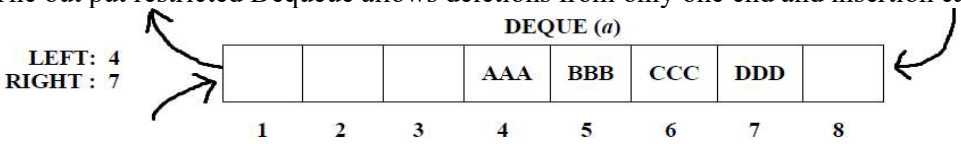


So here we use only three algorithm

1. Insertion of element at Right side(Rear end) of Deq
2. Deletion of element from Left side(Front end) of Deq
3. Deletion of element from Right side(Rear end) of Deq

Output Restricted DEQUE

The out put restricted Dequeue allows deletions from only one end and insertion can perform from both end



Algorithm to add an element from front side into DeQueue :

enq_front (): [Insert element at front side]
LEFT=front (F),RIGHT=Rear(R) and initial values are -1,-1 and Q[] is an array max represent the size of a queue
step1. Start
step2.[Check the queue is full or not]
if (LEFT=1 AND RIGHT=N) OR (RIGHT= =LEFT+1)
Write " over flow" return.
step3. Else
LEFT=LEFT-1
step4. Insert the element at pointer f as Q[LEFT] = element
step5. Stop

Algorithm to add an element from Rear(Right) side into DeQueue :

enq_back ():[Insert element at Left side]
step1. Start
step2. Check the queue is full or not as if (RIGHT == max-1) if yes queue is full
step3. If false update the RIGHT= RIGHT +1
step4. Insert the element at pointer r as Q[RIGHT] = element
step5. Stop

Algorithm to delete an element from the DeQueue

deq_front()
step1. Start
step2. Check the queue is empty or not as if (f == r) if yes queue is empty
step3. If false update pointer f as f = f+1 and delete element at position f as element = Q[f]
step4. If (f== r) reset pointer f and r as f=r=-1
step5. Stop

Deletion from DEQ at rear side

deq_back()
step1. Start
step2. Check the queue is empty or not as if (f == r) if yes queue is empty
step3. If false delete element at position r as element = Q[r]
step4. Update pointer r as r = r-1

step5. If (f == r) reset pointer f and r as f = r= -1
step6. Stop

Exampple

Example 5.4 Let DEQ[1:6] be a deque implemented as a circular array. The contents of DEQ and that of LEFT and RIGHT are as given below:

At Starting:

DEQ: LEFT: 3 RIGHT: 5

[1]	[2]	[3]	[4]	[5]	[6]
		R	T	S	

(i) Insert *X* at the left end and *Y* at the right end

DEQ: LEFT: 2 RIGHT: 6

	F				R
[1]	[2]	[3]	[4]	[5]	[6]
	X	R	T	S	Y

(ii) Delete twice from the right end

DEQ: LEFT: 2 RIGHT: 4

	F		R		
[1]	[2]	[3]	[4]	[5]	[6]
	X	R	T		

(iii) Insert *G*, *Q* and *M* at the left end

DEQ: LEFT: 5 RIGHT: 4

			R	F	
[1]	[2]	[3]	[4]	[5]	[6]
G	X	R	T	M	Q

(iv) Insert *J* at the right end

Here no insertion is possible since the deque is full. Observe the condition LEFT=RIGHT+1 when the deque is full.

(v) Delete twice from the left end

LEFT: 1 RIGHT: 4

F			R		
[1]	[2]	[3]	[4]	[5]	[6]
G	X	R	T		

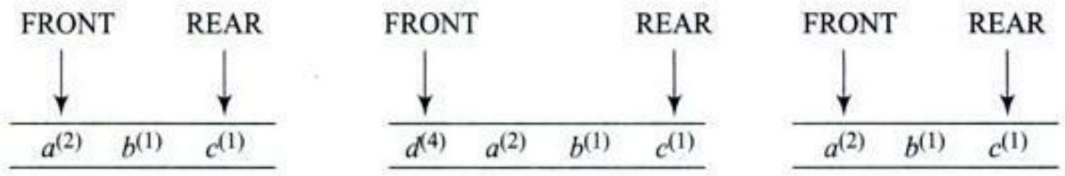
The following operations demonstrate the working of the deque *DEQ* which supports insertions and deletions at both ends.

It is easy to observe that for insertions at the left end, LEFT is decremented by 1 (mod *n*) and for insertions at the right end RIGHT is incremented by 1 (mod *n*). For deletions at the left end, LEFT is incremented by 1 (mod *n*) and for deletions at the right end, RIGHT is decremented by 1 (mod *n*) where *n* is the capacity of the deque. Again, before performing a deletion if LEFT=RIGHT, then it implies that there is only one element and in such a case after deletion set LEFT =RIGHT=NIL to indicate that the deque is empty.

priority queue

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) Initial priority queue (b) Insert $d^{(4)}$ (c) Delete

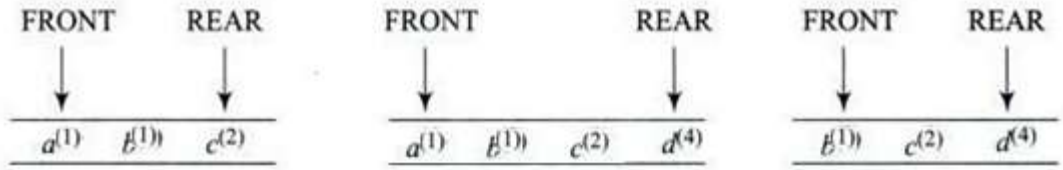
$x^{(y)}$: x is the element with priority y.

Type of Priority Queue

1. Asscending order Priority Queue
2. Descending order Priority Queue

Ascending order Priority Queue:

In ascending order priority queue the highest priority element is placed at last i.e all element are arrange in ascending order of its priority.

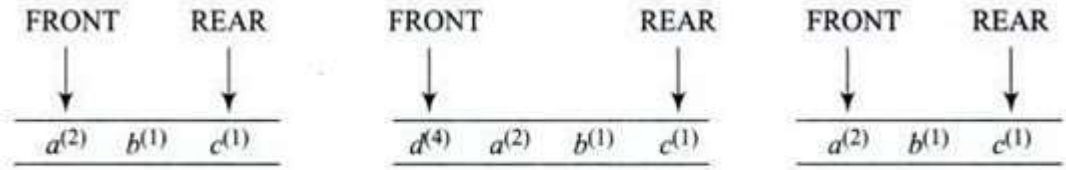


(a) Initial priority queue (b) Insert $d^{(4)}$ (c) Delete

$x^{(y)}$: x is the element with priority y.

Descending order priority queue:

In descending order priority queue the lowest priority element is placed at last i.e all element are arrange in ascending order of its priority



(a) Initial priority queue (b) Insert $d^{(4)}$ (c) Delete

$x^{(y)}$: x is the element with priority y.

Application

In a multiuser system, there will be several programs competing for use of the central processor at one time. The programs have a priority value associated to them and are held in a priority queue. The program with the highest priority is given first use of the central processor.

**********End of Unit 3**********