UNIT II
LINEAR DATA STRUCTURES – STACKS, QUEUES

Stack ADT – Evaluating arithmetic expressions- other applications- Queue ADT – circular queue Implementation – Double ended Queues – applications of queues

Stack ADT

Stack

Stack is abstract data type and linear data structure.

Concept of Stack

A Stack is data structure in which addition of new element or deletion of existing element always takes place at a same end. This end is known as the top of the stack. That means that it is possible to remove elements from a stack in reverse order from the insertion of elements into the stack.

One other way of describing the stack is as a last in, first out (LIFO) abstract data type and linear data structure.

Operations on Stack

The stack is basically performed two operations PUSH and POP. Push and pop are the operations that are provided for insertion of an element into the stack and the removal of an element from the stack, respectively.

PUSH: -	PUSH operation performed for the adding item to the stack.
POP: -	POP operation performed for removing an item from a stack.

Types of Implementation in Stack

1, Static Implementation (Array implementation of Stack)

2, Dynamic Implementation (Lined List Implementation of Stack)

Static Implementation of Stack

Routine to push an element onto the stack

```
void push(int X,stack S)

{

 if(Top==Arraysize-1)
```

```
        Error("Stack is full!!Insertion is not possible");
  else
   {
        Top=Top+1;
        S[Top]=X;
    }
}
```

Routine to check whether a stack is full

```
int Isfull(Stack S)
{
    if(Top==Arraysize-1)
      return(1);
}
```

Routine to check whether stack is empty

```
int Isempty(Stack S)
{
    if(Top==-1)
    return(1);
}
```

pop routine

```
void pop(Stack S)
{
  if(Top==-1)
       Errro("Empty stack! Deletion not possible");
else
{
      X=s[Top];
    Top=Top-1;
  }
}
```

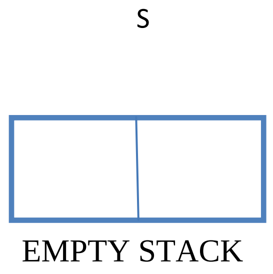
Routine to return top Element of the stack

```
int TopElement(Stack S)
{
   if(Top==-1)
   {
       Error("Empty stack!!No elements");
       return 0;
   }
 else
      return S[Top];
}
```

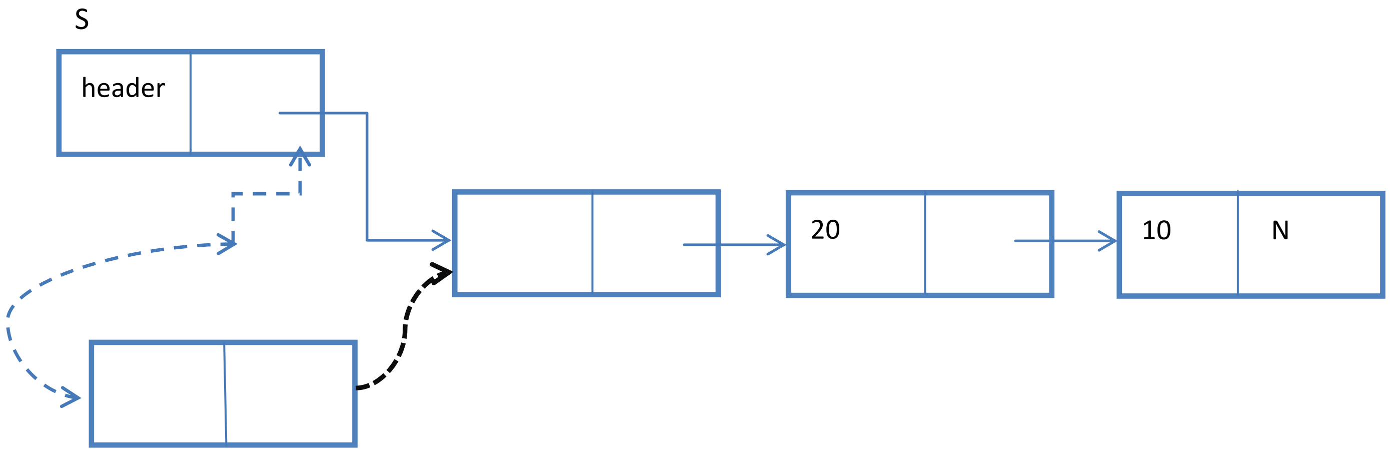
Linked list implementation of Stack

routine to check whether the stack is empty

```
int Isempty(Stack S)
{
 if(S->next==NULL)
 return(1);
}
```



push routine /*Inserts element at front of the list

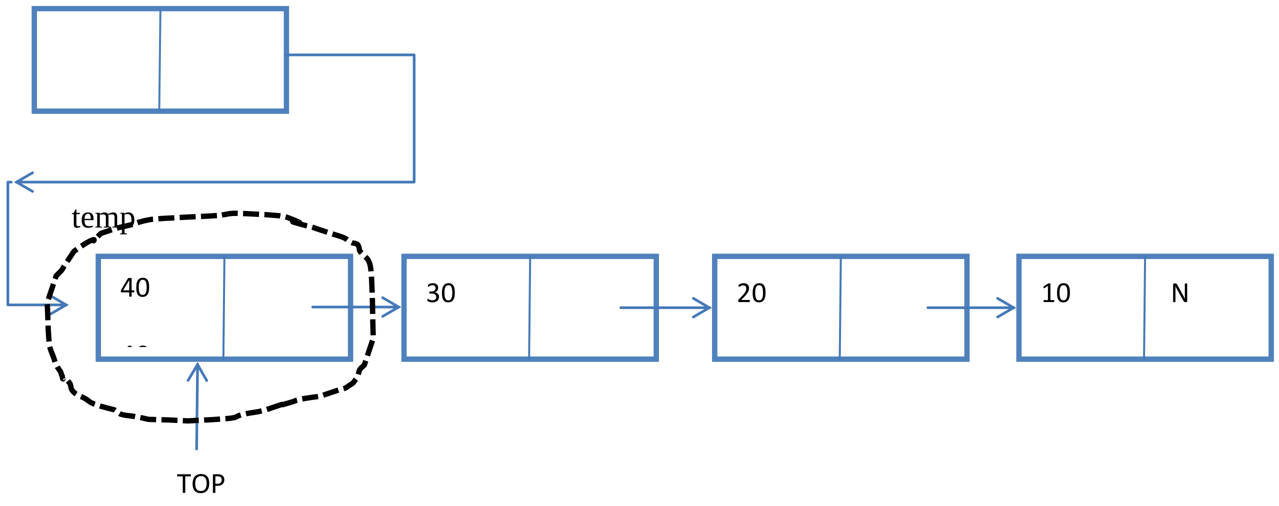


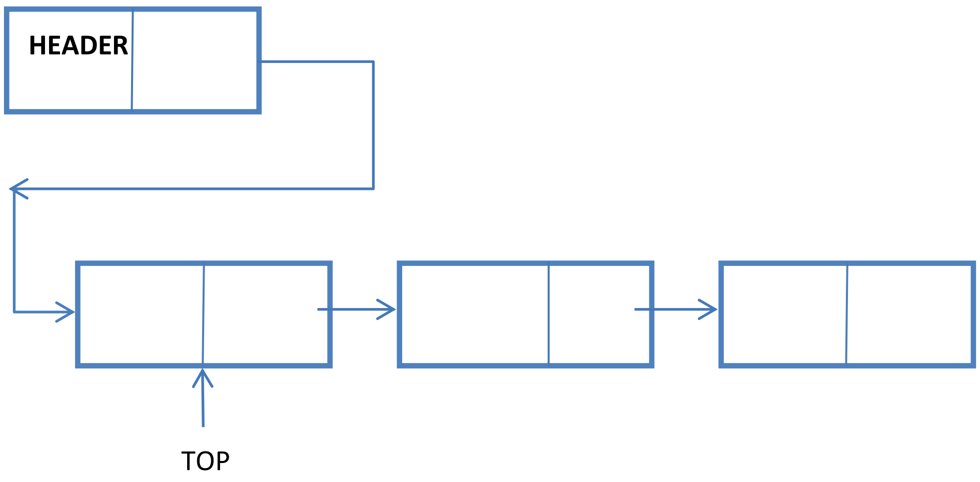
push routine /*Inserts element at front of the list

```
void push(int x, Stack S)
{
             Position newnode,Top;
            newnode=malloc(sizeof(Struct node));
            newnode->data=X;
            newnode->next=Top;
            S->next=newnode;
            Top=newnode;
}
```

pop routine /*Deletes the element at front of list

```
void pop(Stack S)
{
             Position temp,Top;
             Top=S->next;
             if(S->next==NULL)
                         Error("empty stack! Pop not possible");
             else
            {
                         temp=Top;
                         S->next=Top->next;
                         free(temp);
                         Top=s->next;
             }
}
```



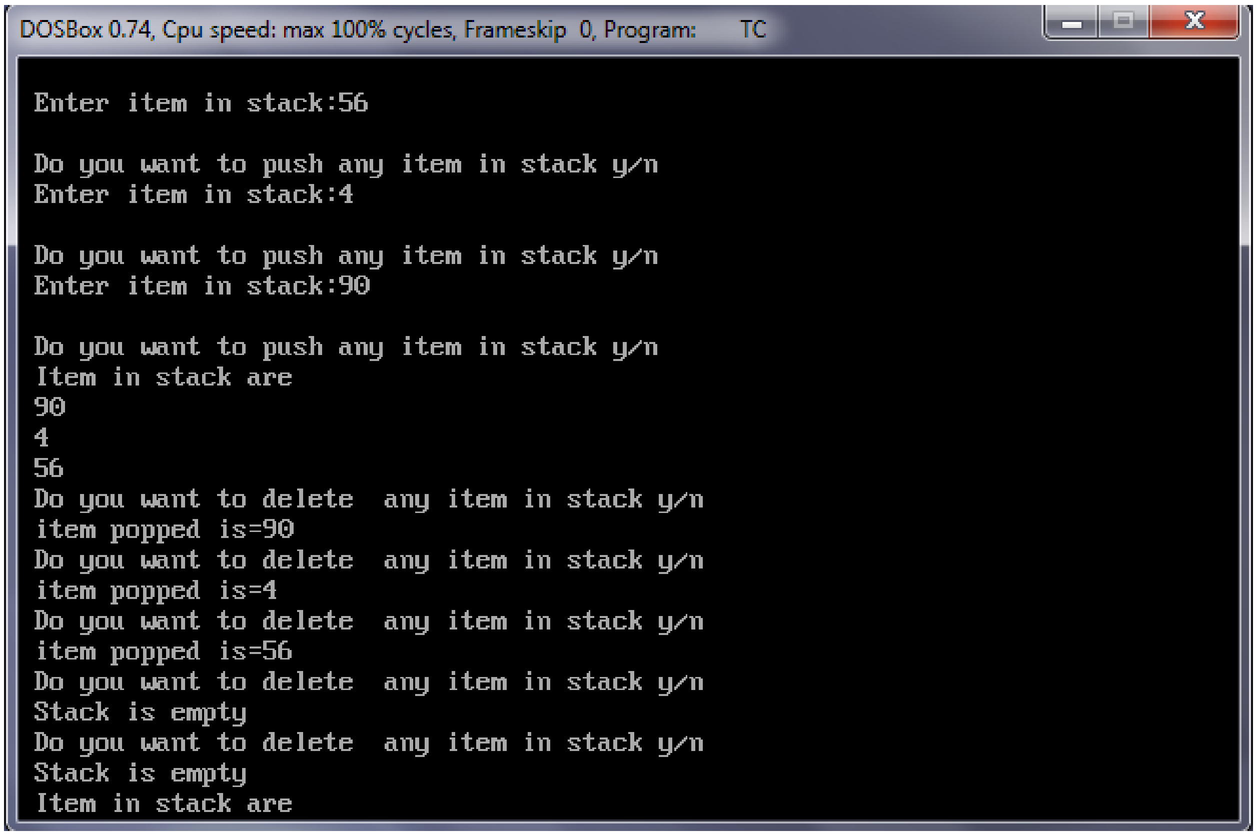


Return Top Element

```
int Topelement(Stack S)
{
   if(S->next==NULL)
     {
        error("Stack is empty");
        return 0;
   else
        return  S->next->data;
}
```

Implementation of stack using ’C'

```
/* static implementation of stack*/
#include<stdio.h>
#include<conio.h>
#define size 5
int stack[size];
int top;
void push()
{
        int n;
        printf("\n Enter item in stack");
        scanf("%d",&n);
        if(top==size-1)
        {
                printf("\nStack is Full");
        }
        else
        {
                top=top+1;
                stack[top]=n;
        }
}
void pop()
{
        int item;
        if(top==-1)
        {
                printf("\n Stack is empty");
        }
        else
        {
                item=stack[top];
                printf("\n item popped is=%d", item);
                top--;
        }
}
void display()
{
        int i;
        printf("\n item in stack are");
        for(i=top; i>=0; i--)
        printf("\n %d", stack[i]);
}
void main()
{
        char ch,ch1;
        ch ='y';
        ch1='y';
        top=-1;
        clrscr();
        while(ch!='n')
        {
                push();
                printf("\n Do you want to push any item in stack y/n");
                ch=getch();
        }
        display();
        while(ch1!='n')
        {
                printf("\n Do you want to delete  any item in stack y/n");
                ch1=getch();
                pop();
        }
        display();
        getch();
}
```



DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC

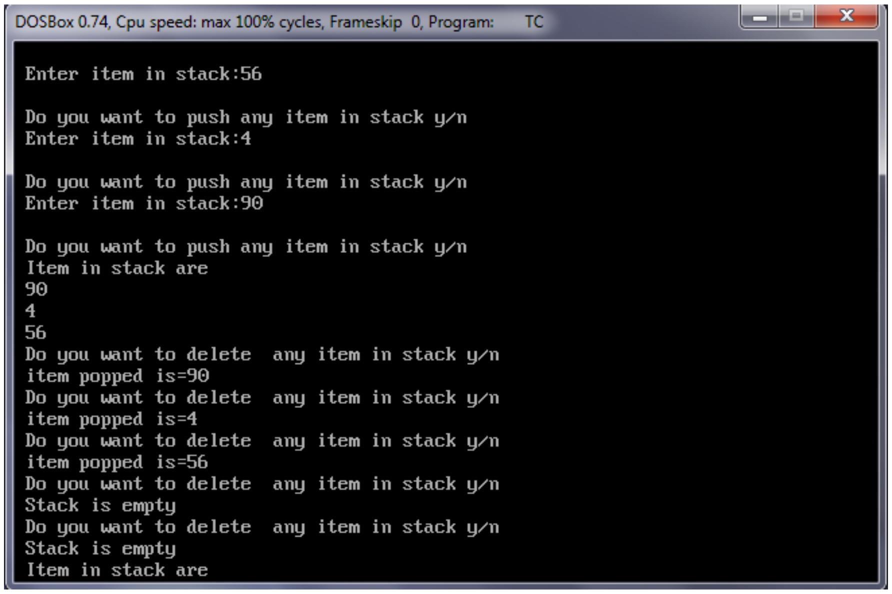
```
Enter item in stack:56

Do you want to push any item in stack y/n
Enter item in stack:4

Do you want to push any item in stack y/n
Enter item in stack:90

Do you want to push any item in stack y/n
Item in stack are
90
4
56
Do you want to delete  any item in stack y/n
item popped is=90
Do you want to delete  any item in stack y/n
item popped is=4
Do you want to delete  any item in stack y/n
item popped is=56
Do you want to delete  any item in stack y/n
Stack is empty
Do you want to delete  any item in stack y/n
Stack is empty
Item in stack are
```

Implementation of stack using 'C'

```
/* Dynamic  implementation of stack*/
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
struct node
{
       int item;
       struct node *next;
};
struct node *top;
void push()
{
         int n;
         struct node *nw;
         nw=(struct node*)malloc(sizeof(struct node));
         printf("\n Enter item in stack:");
         scanf("%d",&n);
         nw->item=n;
         nw->next=0;
         if(top==0)
         {
                        top=nw;
         }
         else
         {
                        nw->next=top;
                        top=nw;
         }
}
void pop()
{
     int item;
     struct node *ptr;
     if(top==0)
     {
           printf("\n Stack is empty");
     }
     else
     {
         item=top->item;
         ptr=top;
         printf("\n item popped is=%d", item);
         top=top->next;
         free(ptr);
```

```
    }
}
void display()
{
        struct node *ptr;
        printf("\n item in stack are");
        for(ptr=top; ptr!=0; ptr=ptr->next)
        printf("\n %d", ptr->item);
}
void main()
{
        char ch,ch1;
        ch ='y';
        ch1='y';
        top=0;
        clrscr();
        while(ch!='n')
        {
                push();
                printf("\n Do you want to push any item in stack y/n");
                ch=getch();
        }
        display();
        while(ch1!='n')
        {
                printf("\n Do you want to delete  any item in stack y/n");
                ch1=getch();
                pop();
        }
        display();
        getch();
}
```

Output


DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC

```
Enter item in stack:56

Do you want to push any item in stack y/n
Enter item in stack:4

Do you want to push any item in stack y/n
Enter item in stack:90

Do you want to push any item in stack y/n
Item in stack are
90
4
56
Do you want to delete  any item in stack y/n
item popped is=90
Do you want to delete  any item in stack y/n
item popped is=4
Do you want to delete  any item in stack y/n
item popped is=56
Do you want to delete  any item in stack y/n
Stack is empty
Do you want to delete  any item in stack y/n
Stack is empty
Item in stack are
```

C Operator Precedence Table

This page lists C operators in order of *precedence* (highest to lowest). Their *associativity* indicates in what order operators of equal precedence in an expression are applied.

Operator	Description	Associativity
( ) [ ] . -> ++ --	Parentheses (function call) (see Note 1) Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement (see Note 2)	left-to-right
++ -- + - ! ~ (*type*) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (convert value to temporary value of *type*) Dereference Address (of operand) Determine size in bytes on this implementation	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
| |	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left
,	Comma (separate expressions)	left-to-right

Note 1:

Parentheses are also used to group sub-expressions to force a different precedence; such parenthetical expressions can be nested and are evaluated from inner to outer.

Note 2:

Postfix increment/decrement have high precedence, but the actual increment or decrement of the operand is delayed (to be accomplished sometime before the statement completes execution). So in the statement **y = x * z++;** the current value of **z** is used to evaluate the expression (*i.e.*, **z++** evaluates to **z**) and **z** only incremented after all else is done. See **postinc.c** for another example.

Stack Applications

Application of Stack:

1. Parsing
2. Recursive Function
3. Calling Function
4. Expression Evaluation
5. Expression Conversion
    1. Infix to Postfix
    2. Infix to Prefix
    3. Postfix to Infix
    4. Prefix to Infix
6. Towers of Hanoi

Algorithm for Infix to Postfix Conversion

```
1. scan infix expression from left to right
2. if an operand is encountered add to P
3. if an operator is found
        i) repeatedly pop the operator from stack which are having higher precedence than the operator found
        ii) add the new operator to stack
4. if a right parenthesis found
        i) repeatedly pop the stack and add the poped operators to the expression until a left parenthesis is found.
        ii) remove the left parenthesis
5. stop
```

Example for Infix to Postfix Conversion

Example : The Given Infix expression is ((A+B)/ C)

Symbols Scanned	**Stack**	**Postfix form P**
(	(	
(	((	
A	((	A
+	((+	A
B	((+	AB
)	((	AB+
/	((/	AB+
C	((/	AB+C
)	((	AB+C/

EXPRESSION CONVERSION

Infix to Postfix

Infix :- Operators appears between operands

Eg. A+B

A/B+C

Postfix:- Operators appears after the operands

AB+

AB/C+

Prefix:- Operators appears before the operands

+AB

+/ABC

1. Evaluate Arithmetic Expressions
   1. Convert the given infix expression → Postfix expression
   2. Evaluate the postfix expression using stack.

Infix expression:- A*B+(C-D/E)#

Read char	**Stack**	**Output**
A		A
*	*	A
B	+ *	AB
+	+	AB*
(	(+	AB*
C	(+	AB*C
-	- (+	AB*C

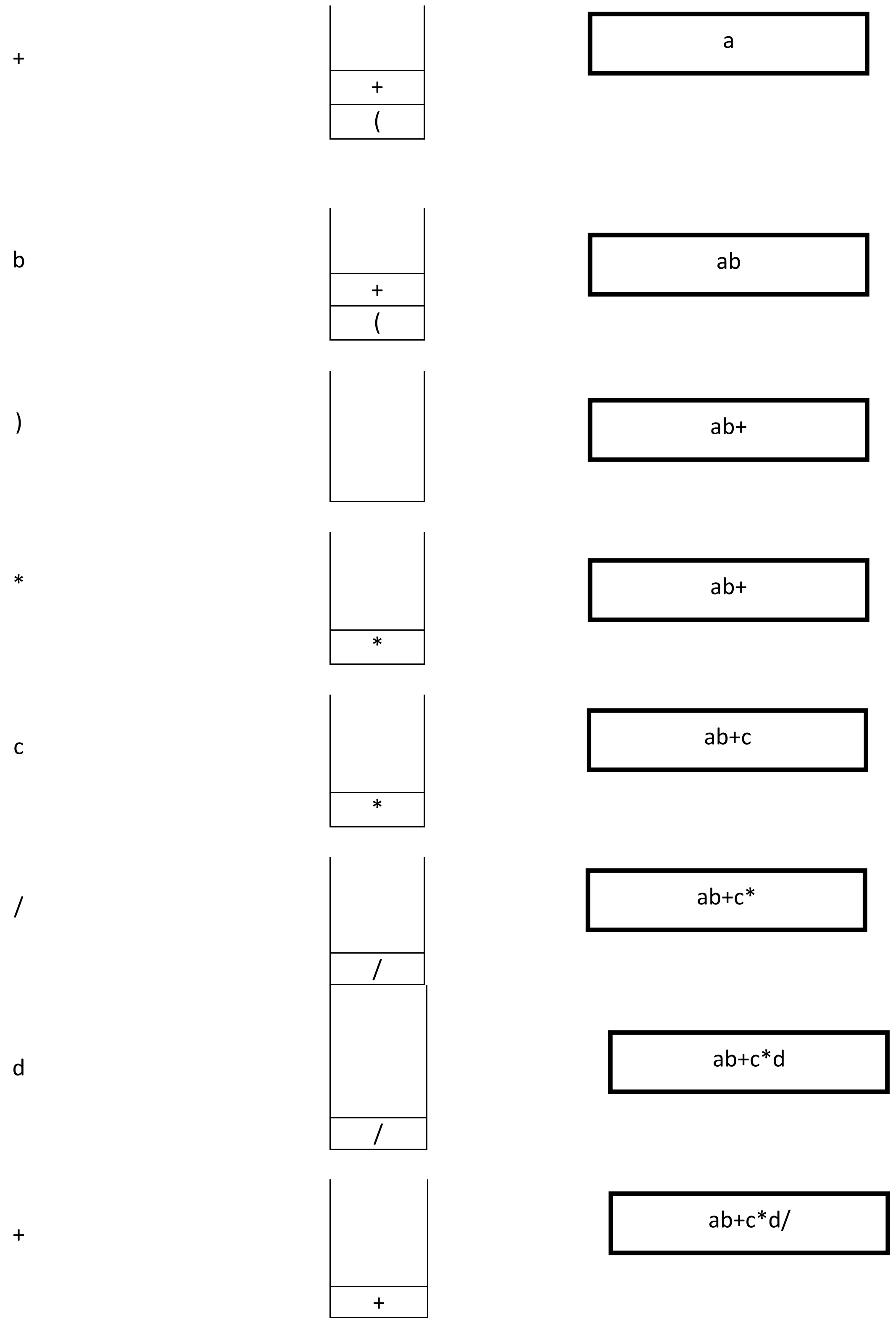
D	- (+	AB*CD
/	/ - (+	
E		AB*CDE/-+
)		
#		AB*CDE/-+

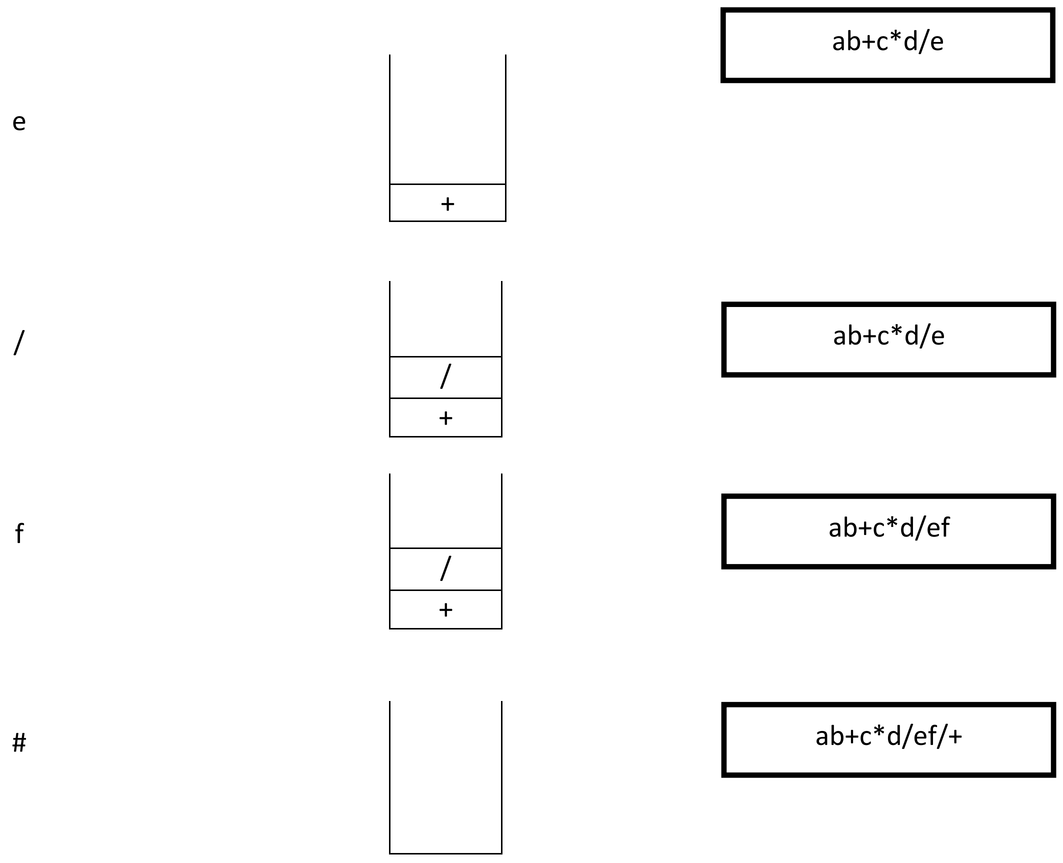
Eg:- A/B-C+D*E-A*C
AB/C-DE*+AC*-

Output: Postfix expression:- AB*CDE/-+

Example: Infix expression:- (a+b)*c/d+e/f#

Read char	**Stack**	**Output**
(	(	
a	(	a


+
+
(
a
b
+
(
ab
)
ab+
*
*
ab+
c
*
ab+c
/
/
ab+c*
d
/
ab+c*d
+
+
ab+c*d/



Postfix expression:- ab+c*d/ef/+

TOWERS OF HANOI

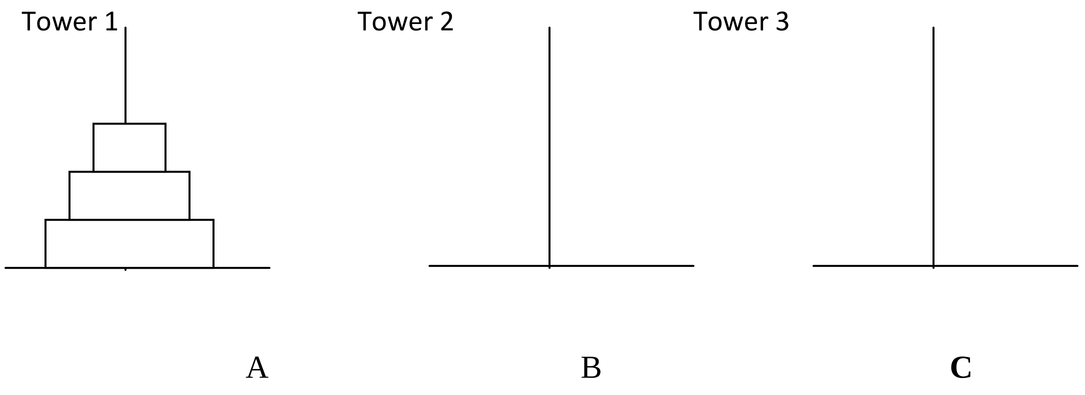
Towers of Hanoi can be easily implemented using recursion.Objective of the problem is moving a collection of N disks of decreasing size from one pillar to another pillar. The movement of the disk is restricted by the following rules.

Rule 1 : Only one disk could be moved at a time.

Rule 2 : No larger disk could ever reside on a pillar on top of a smaller disk.

Rule 3 : A 3rd pillar could be used as an intermediate to store one or more disks, while they

were being moved from source to destination.



Initial Setup of Tower of Hanoi

RECURSIVE SOLUTION

N - represents the number of disks.

Step 1. If N = 1, move the disk from A to C.

Step 2. If N = 2, move the 1st disk from A to B. Then move the 2nd disk from A to C, The move the 1st
disk from B to C.

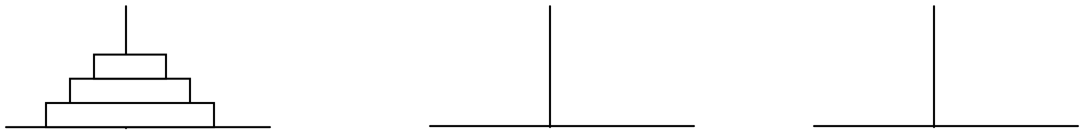
Step 3. If N = 3, Repeat the step (2) to more the first 2 disks from A to B using C as intermediate.Then the
3rd disk is moved from A to C. Then repeat the step (2) to move 2 disks from B to C using A as
intermediate.

In general, to move N disks. Apply the recursive technique to move N - 1 disks from A to B using C as an intermediate. Then move the Nth disk from A to C. Then again apply the recursive technique to move N - 1 disks from B to C using A as an intermediate.

RECURSIVE ROUTINE FOR TOWERS OF HANOI

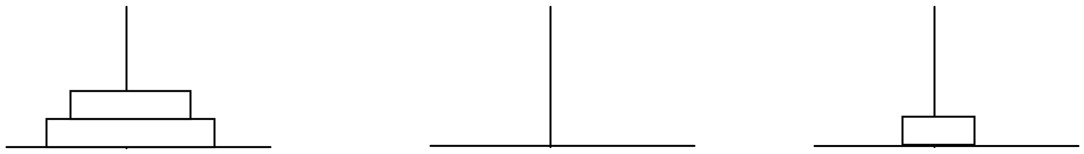
```
void hanoi (int n, char s, char d, char i)
{
/* n    no. of disks, s    source, d    destination i    intermediate */
if (n = = 1)
{
print (s, d);
return;
}
else
{
hanoi (n - 1, s, i, d);
print (s, d)
hanoi (n-1, i, d, s);
return;
}
}
```

Source Pillar Intermediate Pillar Destination Pillar

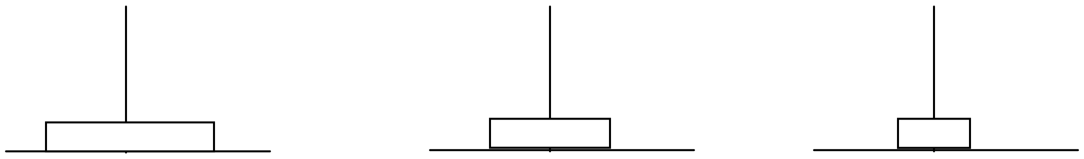


Tower 1 Tower 2 Tower 3

1. Move Tower1 to Tower3

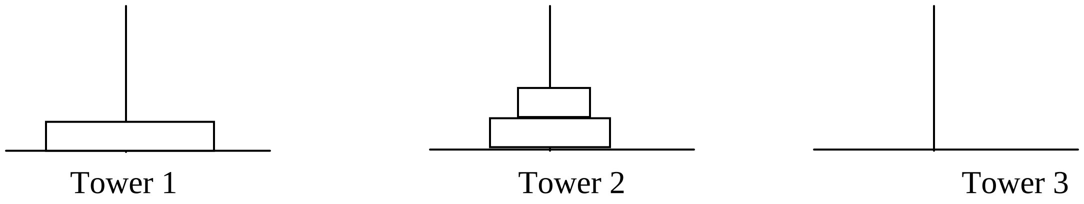


2. Move Tower1 to Tower2

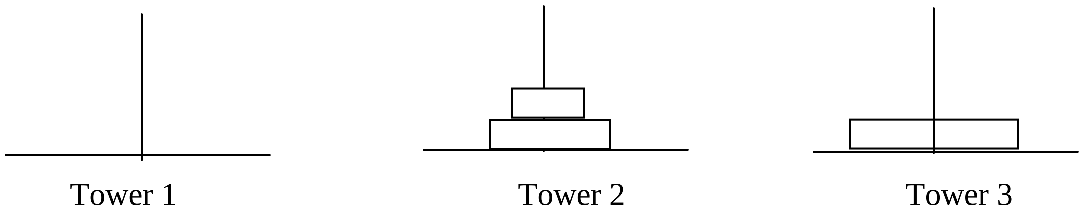


Tower 1 Tower 2 Tower 3

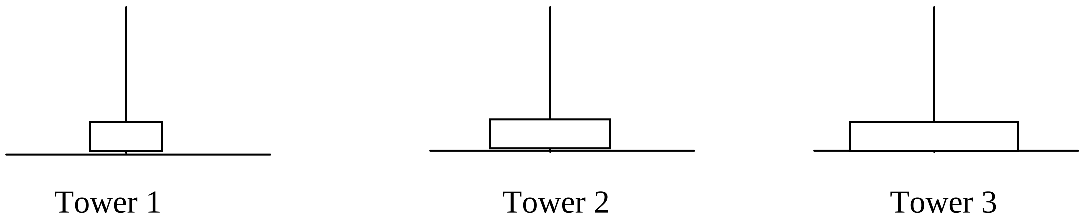
3. Move Tower 3 to Tower 2



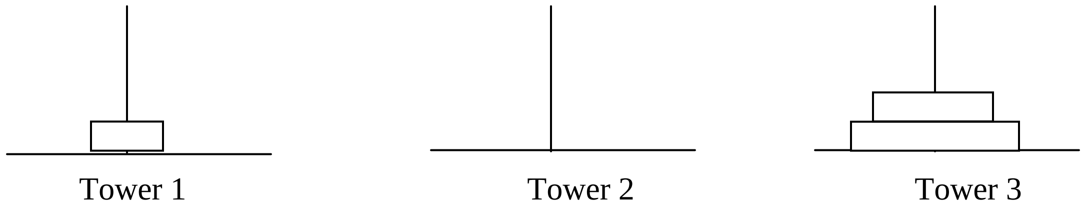
4. Move Tower 1 to Tower 3



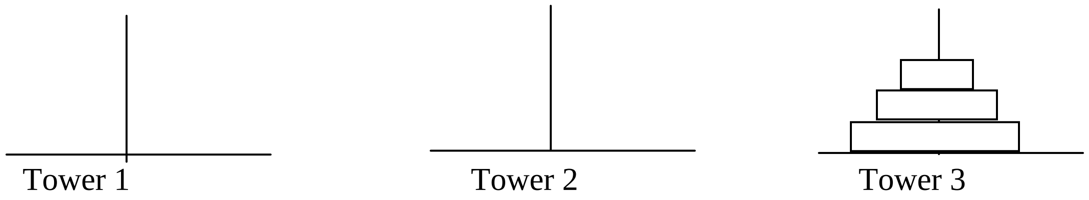
5. Move Tower 2 to Tower 1



6. Move Tower 2 to Tower 3



7.Move Tower 1 to Tower 3

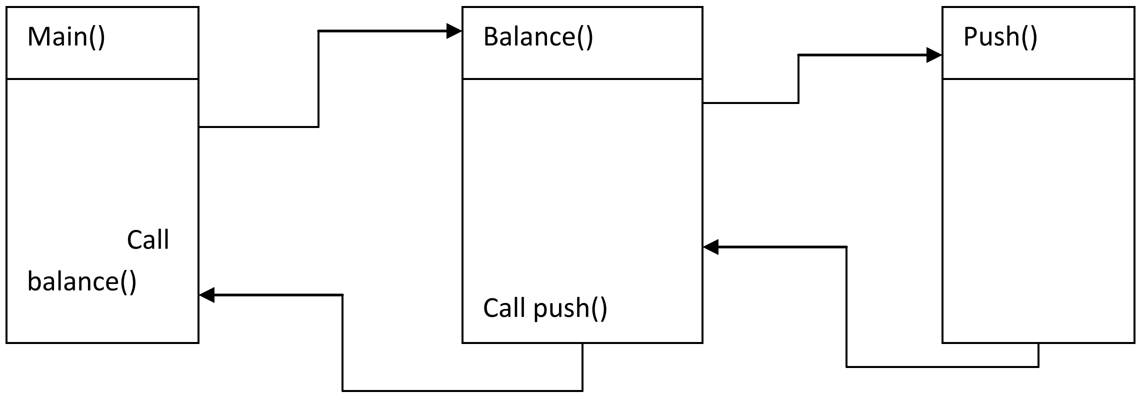


Since disks are moved from each tower in a LIFO manner,each tower may be considered as a Stack.Least Number of moves required to solve th eproblem according to our alforithm is given by,

$$O(N)=O(N-1)+1+O(N-1) =2^N-1$$

FUNCTION CALLS

When a call is made to a new function all the variables local to the calling routine need to be saved, otherwise the new function will overwrite the calling routine variables. Similarly the current location address in the routine must be saved so that the new function knows where to go after it is completed.



RECURSIVE FUNCTION TO FIND FACTORIAL

```
Int fact(int n)

{

int S;

if(n==1)

    return(1);

else

            S=n*fact(n-1);

            return(S)

}
```

BALANCING THE SYMBOLS

- Compilers check the programs for errors, a lack of one symbol will cause an error.
- A Program thet checks whether everything is balanced.
- Every right paranthesis should have its left paranthesis.
- Check for balancing the paranthesis brackets braces and ignore any other character

Read one character at a time until it encounters the delimiter `#'.

Step 1 : - If the character is an opening symbol, push it onto the stack.

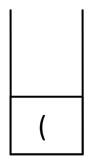
Step 2 : - If the character is a closing symbol, and if the stack is empty report an error as missing opening symbol.

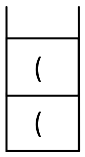
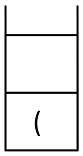
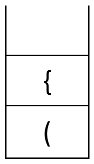
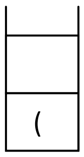
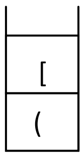
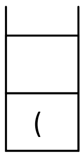
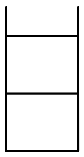
Step 3 : - If it is a closing symbol and if it has corresponding opening symbol in the stack, POP it from the stack. Otherwise, report an error as mismatched symbols.

Step 4 : - At the end of file, if the stack is not empty, report an error as Missing closing symbol. Otherwise, report as Balanced symbols.

Let us consider the expression as ((B*B)-{4*A*C}/[2*A]) #

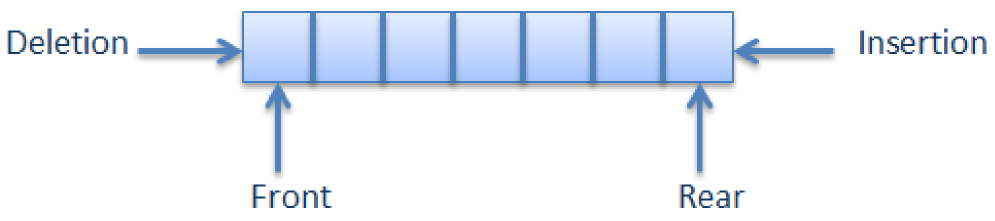
Thus all the paranthesis are balanced

((B*B)-{4*A*C}/[2*A]) #	
Read Character	**Stack**
(	

(	
)	
{	
}	
[	
]	
)	

Queues

Queues is a kind of abstract data type where items are inserted one end (rear end) known as **enqueue** operation and deleted from the other end(front end) known as **dequeue** operation. This makes the queue a **First-In-First-Out (FIFO)** data structure. The queue performs the function of a buffer.



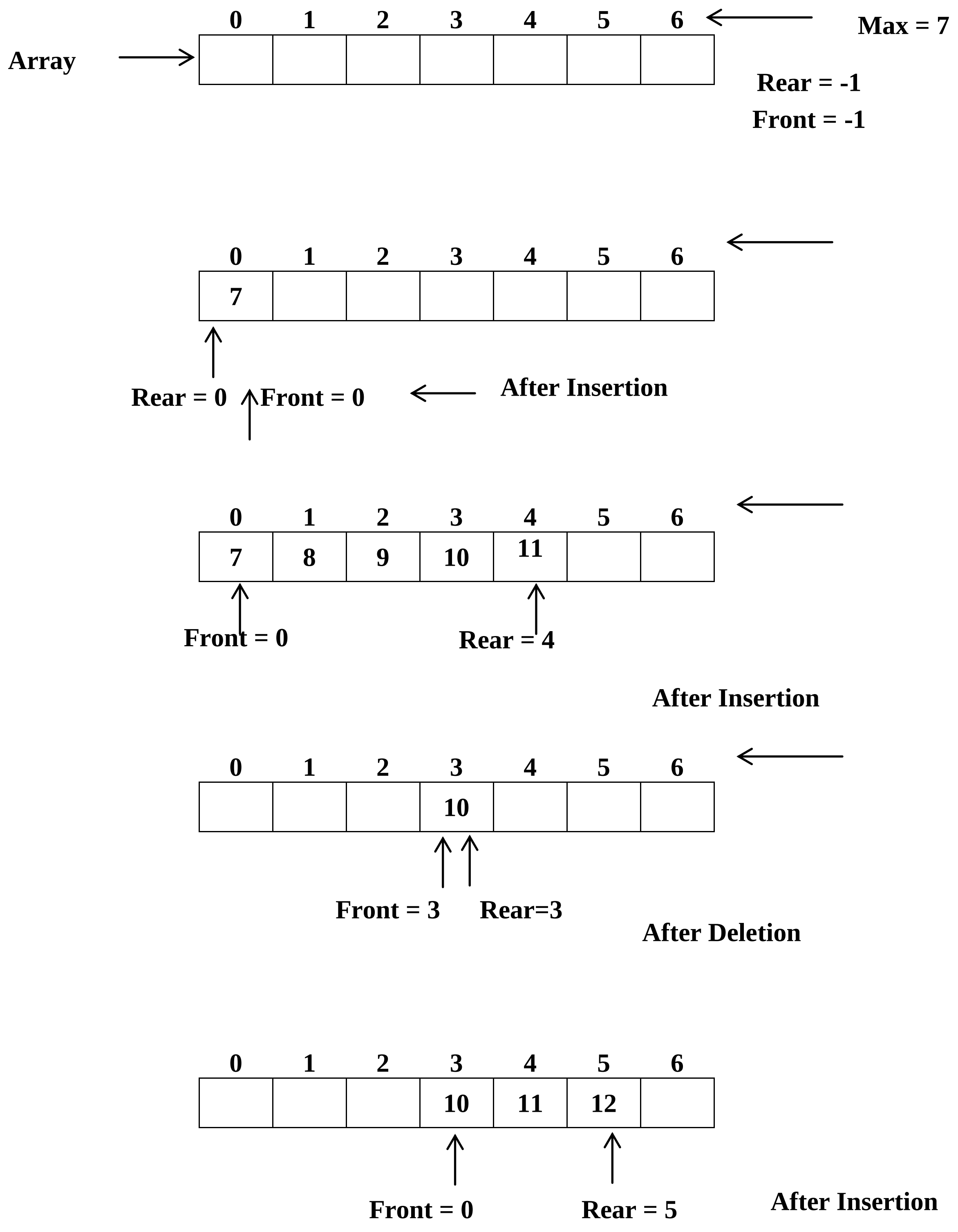
Implementation of Queues

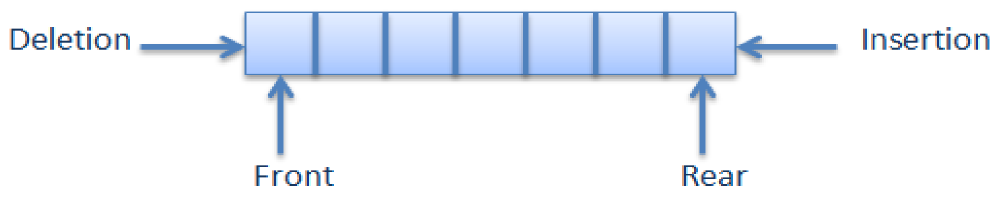
1. Implementation using Array (**Static Queue**)
2. Implementation using Linked List (**Dynamic Queue**)

Operation on Queues

Operation	Description
initialize()	Initializes a queue by adding the value of rear and font to -1.
enqueue()	Insert an element at the rear end of the queue.
dequeue()	Deletes the front element and return the same.
empty()	It returns true(1) if the queue is empty and return false(0) if the queue is not empty.
full()	It returns true(1) if the queue is full and return false(0) if the queue is not full.

Operation on queue





ROUTINE TO INSERT AN ELEMENT IN A QUEUE

```
void Enqueue (int X)
{
if (rear == max _ Arraysize-1)
print (" Queue overflow");
else
{
Rear = Rear + 1;
Queue [Rear] = X;
}
}
```

ROUTINE FOR DEQUEUE

```
void delete ( )
{
if (Front < 0)
print (" Queue Underflow");
else
{
X = Queue [Front];
if (Front = = Rear)
{
Front = 0;
Rear = -1;
}
else
Front = Front + 1 ;
} }
```

Application of queues

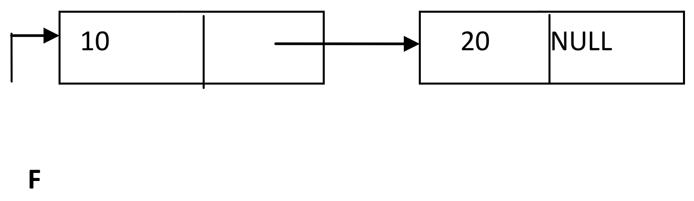
Queues are mostly used in operating systems.

- Waiting for a particular event to occur.
- Scheduling of processes.

IMPLEMENTATION OF QUEUE USING LINKED LIST

Enqueue operation is performed at the end of the list and Dequeue operation is performed at the front of the list.

QUEUE ADT



DECLARATION FOR LINKED LIST IMPLEMENTATION OF QUEUE ADT

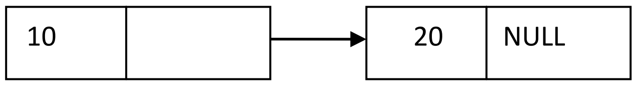
```
Struct Node;
typedef Struct Node * Queue;
int IsEmpty (Queue Q);
Queue CreateQueue (void);
void MakeEmpty (Queue Q);
void Enqueue (int X, Queue Q);
void Dequeue (Queue Q);
Struct Node
{
int Element;
Struct Node *Next;
}* Front = NULL, *Rear = NULL;
```

```
int IsEmpty (Queue Q) // returns boolean value /

{ // if Q is empty

if (Q→Next = = NULL) // else returns 0

return (1);

}
```

ROUTINE TO CHECK AN EMPTY QUEUE

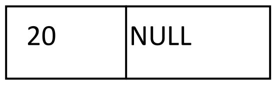
```
Struct CreateQueue ( )
{
Queue Q;
Q = Malloc (Sizeof (Struct Node));
if (Q = = NULL)
Error ("Out of Space");
MakeEmpty (Q);
return Q;
}
void MakeEmpty (Queue Q)
{
if (Q = = NULL)
                    Error ("Create Queue First");

else
while (! IsEmpty (Q)
Dequeue (Q);
```

i)

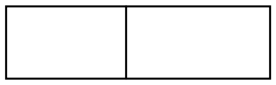


Dequeue

ii)



iii)

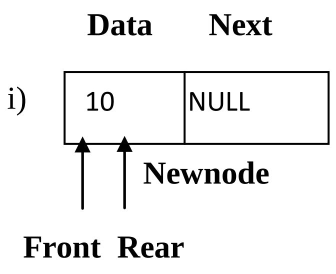


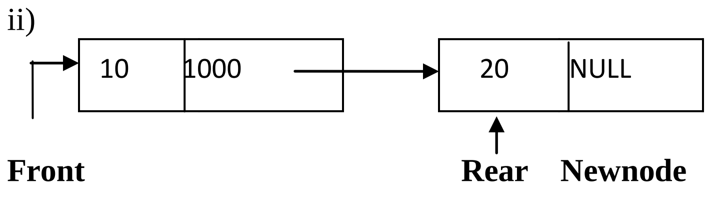
EmptyQueue

ROUTINE TO ENQUEUE AN ELEMENT IN QUEUE

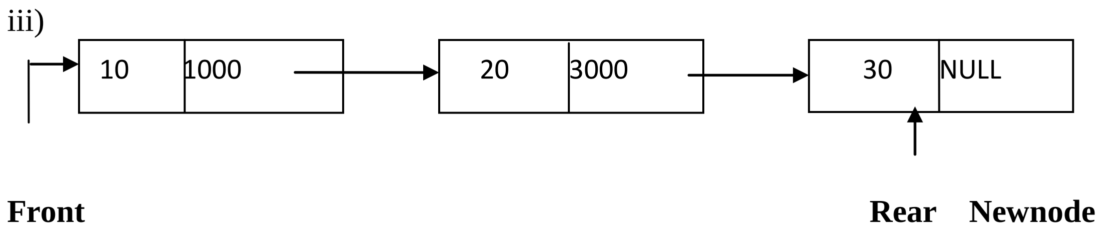
```
void Enqueue (int X)
{
Struct node *newnode;
newnode = Malloc (sizeof (Struct node));
if (Rear = = NULL)
{
newnode →data = X;
newnode→Next = NULL;
Front = newnode;
Rear = newnode;
}
else
{
newnode →data = X;
newnode →Next = NULL;
Rear →next = newnode;
Rear = newnode;
} }
```

Example:10,20,30





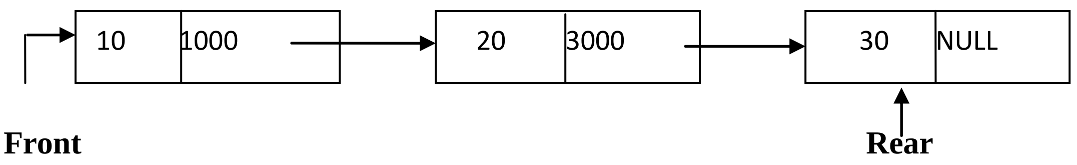
- Assign Newnode as rear



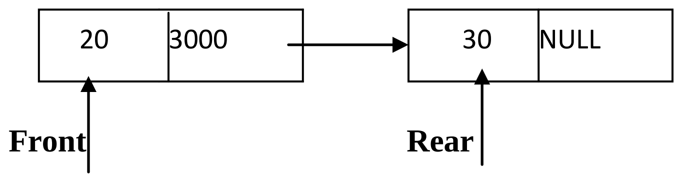
- Assign Newnode as rear

ROUTINE TO DEQUEUE AN ELEMENT FROM THE QUEUE

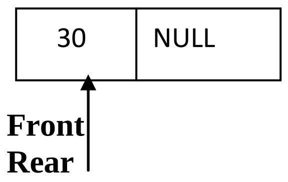
```
void Dequeue ( )
{
Struct node *temp;
if (Front = = NULL)
Error("Queue is underflow");
else
{
temp = Front;
if (Front = = Rear)
{
Front = NULL;
Rear = NULL;
}
else
Front = Front →Next;
Print (temp→data);
free (temp);
}
}
```



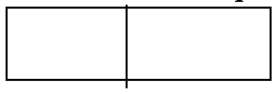
Assign the front element as temp and front→next as front to delete the temp data.



Assign the front element as temp and front→next as front to delete the temp data.



If front and rear are equal assign to null and delete the corresponding front element

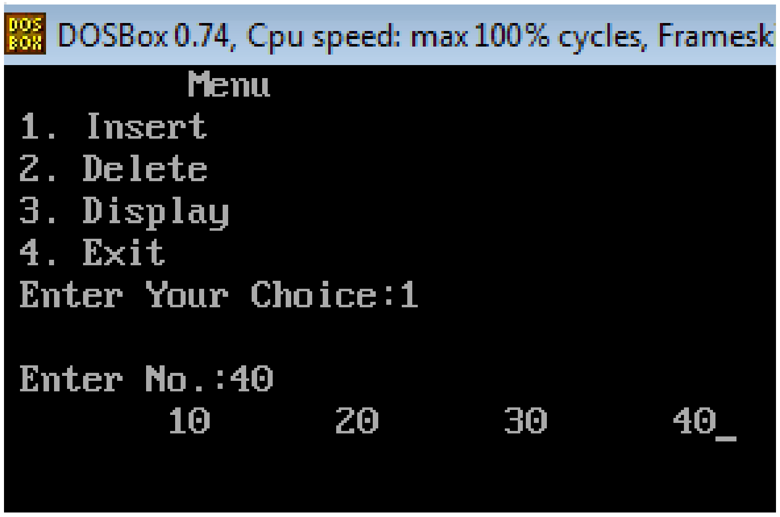


Front=rear=NULL

Array implementation of Queue

In this type of implementation we use array structure for data storage.

```
#include<stdio.h>
#include<conio.h>
#define SIZE 5
 int front=-1;
int rear=-1;
int q[SIZE];

void insert();
void del();
void display();

void main()
{
        int choice;
        clrscr();
        do
        {
                clrscr();
                printf("\t Menu");
                printf("\n 1. Insert");
                printf("\n 2. Delete");
                printf("\n 3. Display ");
                printf("\n 4. Exit");

                printf("\n Enter Your Choice:");
                scanf("%d",&choice);
                switch(choice)
                {
                        case 1:
                                insert();
                                display();
                                getch();
                                break;
                        case 2:
                                 del();
                                 display();
                                 getch();
                                 break;
                        case 3:
                                display();
                                getch();
                                break;
                        case 4:
                                printf("End of Program....!!!!");
                                getch();
                                exit(0);
                }
        }while(choice!=4);
}

void insert()
{
        int no;
        printf("\n Enter No.:");
        scanf("%d",&no);

        if(rear < SIZE-1)
        {
                q[++rear]=no;
                if(front==-1)
                front=0;// front=front+1;
        }
        else
        {
                printf("\n Queue overflow");
        }
}

void del()
{
        if(front==-1)
        {
                printf("\nQueue Underflow");
                return;
        }
        else
        {
                printf("\nDeleted Item:-->%d\n",q[front]);
        }
        if(front==rear)
        {
                front=-1;
                rear=-1;
        }
        else
        {
                front=front+1;
        }
}

void display()
{
        int i;
        if(front==-1)
        {
                printf("\nQueue is empty....");
                return;
        }
        for(i=front;i<=rear;i++)
                printf("\t%d",q[i]);
}
```



```
          Menu
1. Insert
2. Delete
3. Display
4. Exit
Enter Your Choice:1

Enter No.:40
        10      20      30      40_
```

Dynamic Queue

Structure of Dynamic Queue.

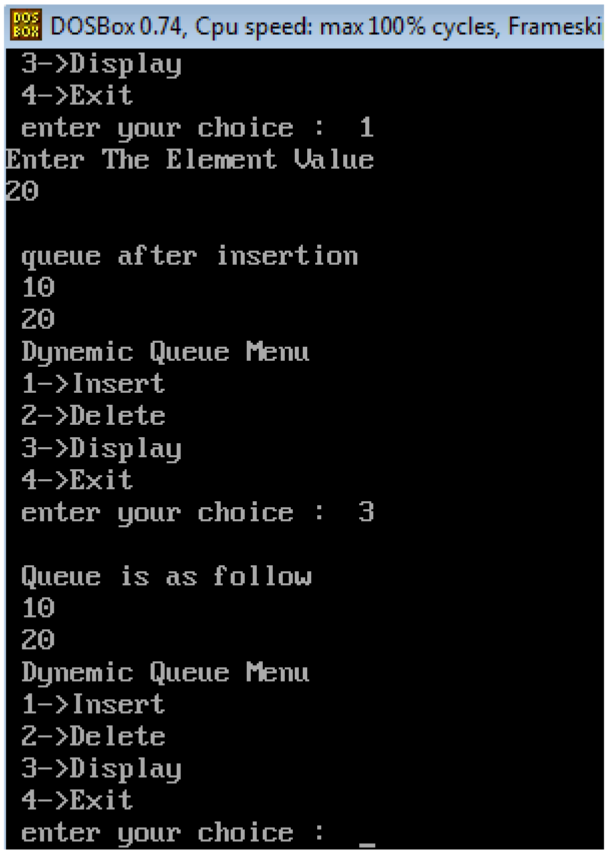
```
struct link

{

        int info;

        struct link *next;

}*front,*rear;
```

Linked list implementation of Queue

```
#include<stdio.h>
#include<conio.h>
struct link
{
        int info;
        struct link *next;
}*front,*rear;

void insert_q(int no);

int delete_q();

void display();

void main()
{
        int ch,no;
        rear=NULL;
        front=NULL;
        clrscr();
        while(1)
        {
                printf("\n Dynemic Queue Menu");
                printf("\n 1->Insert ");
                printf("\n 2->Delete ");
                printf("\n 3->Display");
                printf("\n 4->Exit   ");
                printf("\n enter your choice :  ");
                scanf("%d",&ch);

                switch(ch)
                {
                        case 1:
                        {
                                printf("Enter The Element Value\n");
                                scanf("%d",&no);
                                insert_q(no);
                                printf("\n queue after insertion");
                                display();
                                break;
                        }
                        case 2:
                        {
                                no=delete_q();
                                printf("\n The deleted element = %d",no);
                                printf("\n queue after deletion");
                                display();
                                break;
                        }

                        case 3:
                        {
                                printf("\n Queue is as follow");
                                display();
                                break;
                        }

                        case 4:
                        {
                                printf("Exit....!!!!");
                                getch();
                                exit(0);
                                break;
                        }
                        default :
                           printf("\n Wrong choice...!!!! " );

                 }

        }

}

void insert_q(int no)
```

```
{
                struct link *new1;
                new1=(struct link*)malloc(sizeof(struct link));
                new1->info=no;
                new1->next=NULL;
                if(rear==NULL||front==NULL)
                {
                          front=new1;
                }
                else
                {
                          rear->next=new1;
                }
                rear=new1;

}

int delete_q()
{
      struct link *t;
      int no;

      if(front==NULL||rear==NULL)
      {
                printf("\n queue is Under Flow");
                getch();
                return;

      }

      else
      {
                t=front;
                no=t->info;
                front=front->next;
                free(t);

      }
      return(no);

}

void display()

{
      struct link *t;
```

```
        t=front;
        if(front==NULL||rear==NULL)
        {
                printf("\nQueue is empty");
                getch();
                exit(0);
        }
        while(t!=NULL)
        {
                printf("\n %d",t->info);
                t=t->next;
        }


}
```

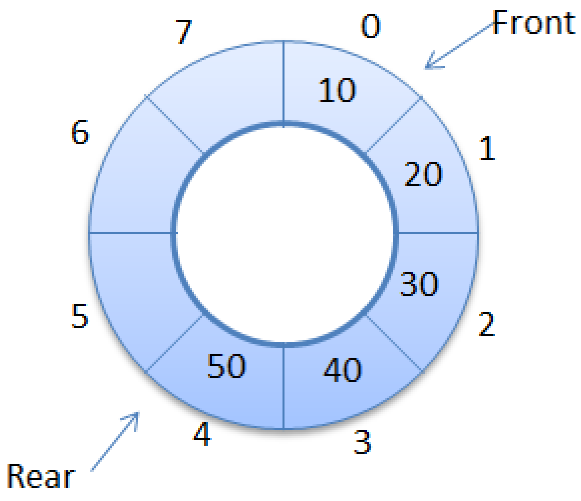

DOSBox 0.74, Cpu speed: max 100% cycles, Frameski

```
 3->Display
 4->Exit
 enter your choice :  1
Enter The Element Value
20

 queue after insertion
 10
 20
 Dynemic Queue Menu
 1->Insert
 2->Delete
 3->Display
 4->Exit
 enter your choice :  3

 Queue is as follow
 10
 20
 Dynemic Queue Menu
 1->Insert
 2->Delete
 3->Display
 4->Exit
 enter your choice :  _
```

Circular Queue

A circular queue is an abstract data type that contains a collection of data which allows addition of data at the end of the queue and removal of data at the beginning of the queue. Circular queues have a fixed size.

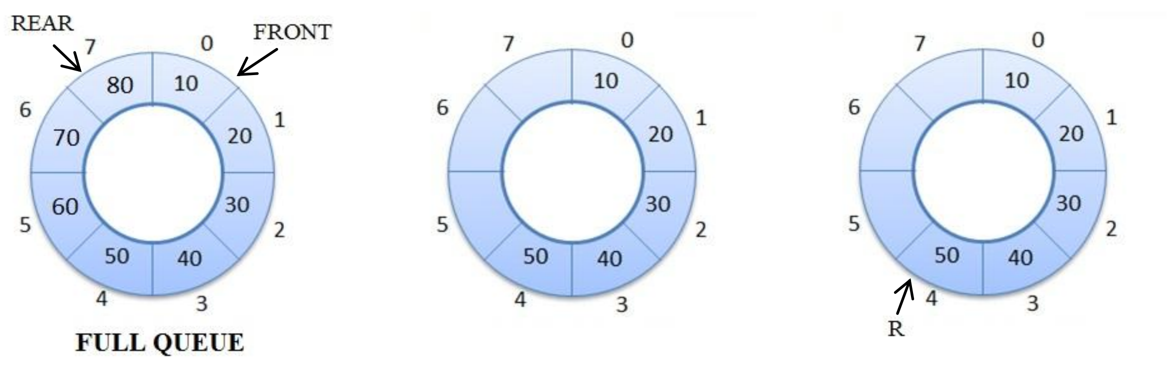
Circular queue follows FIFO principle. Queue items are added at the rear end and the items are deleted at front end of the circular queue.



Enqueue Routine in Circular List

```
Void CEnqueue (int X,Circularqueue CQ)
{
    if(Front==(rear+1)%Arraysize)
            Error("Queue is full!!Queue overflow");
    else if(front== -1)
   {
          front=front+1;
          rear=rear+1;
          CQ[rear]=x;
   }
   else
```

```
        {
            rear=(rear+1)%Arraysize;
            CQ[rear]=X;
        }

}
```

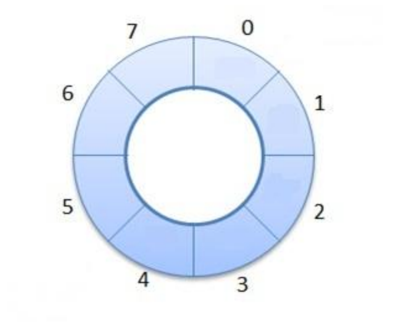


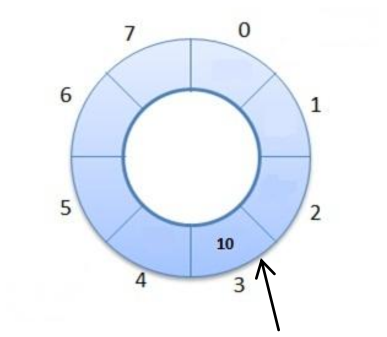
Empty Queue
F = -1
R = -1

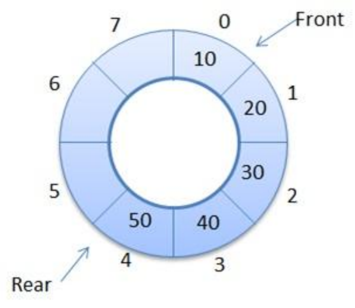
DEQUEUE ROUTINE IN CIRCULAR LIST

```
void dequeue (circularqueue CQ)
{
if(Front== - 1)
        Empty("Empty Queue!");
else if(Front==rear)
  {
     X=CQ[Front];
     Front=-1;
     Rear=-1;
  }
else
```

```
  {
    X=CQ[Front];
    Front=(Front+1)%Arraysize;
  }
}
```







F= -1
R= -1 **Empty Queue**

F,R

Routine for display

```
void display(circularqueue CQ , int i , int j)
{
   if(front==0&&rear==-1)
   {
      print("Queue is underflow”);
      exit();
   }
   if(front>rear)
   {
      for(i=0;i<=rear;i++)
         return CQ[i];
```

```
        for(j=front;j<=max-1;j++)
            return CQ[j];
        return CQ[rear];
        return CQ[front];
    }
    else
    {
        for(i=front;i<=rear;i++)
        {
return CQ[i];
        }
        return Q[rear];
        return Q[front];
    }
}
```

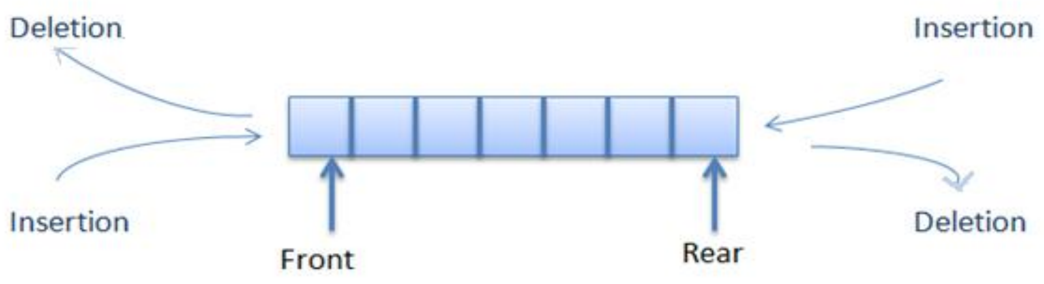
Implementation of Circular Queue

```
    #include<stdio.h>
    #define max 3
    int q[10],front=0,rear=-1;
    void main()
    {
        int ch;
        void insert();
        void delet();
        void display();

        printf("\nCircular Queue operations\n");
    printf("1.insert\n2.delete\n3.display\n4.exit\n");
    while(1)
    {
        printf("Enter your choice:");
        scanf("%d",&ch);
        switch(ch)
        {
        case 1: insert();
            break;
        case 2: delet();
            break;
        case 3:display();
            break;
        case 4:exit();
        default:printf("Invalid option\n");
        }
    }
}
 void insert()
{
    int x;
           if((front==0&&rear==max-1)||(front>0&&rear==front-1))
        printf("Queue is overflow\n");
    else
    {
        printf("Enter element to be insert:");
        scanf("%d",&x);
        if(rear==max-1&&front>0)
        {
            rear=0;
            q[rear]=x;
        }
        else
        {
            if((front==0&&rear==-1)||(rear!=front-1))
                q[++rear]=x;
        }
    }
}
void  delete()
{
    int a;
    if((front==0)&&(rear==-1))
    {
        printf("Queue is underflow\n");
        getch();
        exit();
    }
    if(front==rear)
    {
        a=q[front];
        rear=-1;
        front=0;
    }
    else
        if(front==max-1)
        {
            a=q[front];
            front=0;
        }
        else a=q[front++];
        printf("Deleted element is:%d\n",a);
```

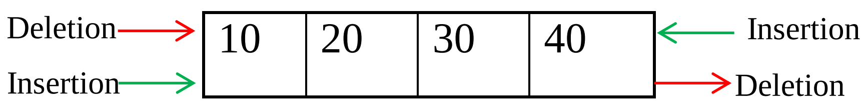
```
}
 void display()
{
    int i,j;
    if(front==0&&rear==-1)
    {
        printf("Queue is underflow\n");
        getch();
        exit();
    }
    if(front>rear)
    {
        for(i=0;i<=rear;i++)
            printf("\t%d",q[i]);
        for(j=front;j<=max-1;j++)
            printf("\t%d",q[j]);
        printf("\nrear is at %d\n",q[rear]);
        printf("\nfront is at %d\n",q[front]);
    }
    else
    {
        for(i=front;i<=rear;i++)
        {
            printf("\t%d",q[i]);
        }
        printf("\nrear is at %d\n",q[rear]);
        printf("\nfront is at %d\n",q[front]);
    }
    printf("\n");
}
```

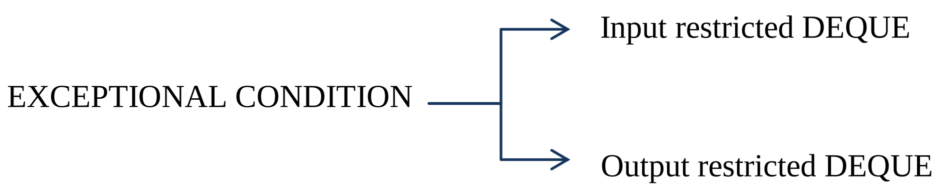
Double-Ended Queue

A double-ended queue is an abstract data type similar to an simple queue, it allows you to insert and delete from both sides means items can be added or deleted from the front or rear end.



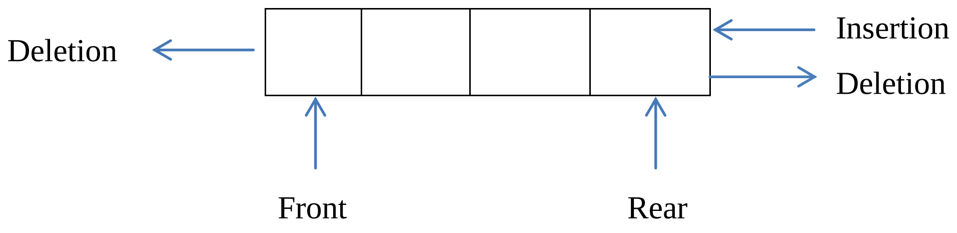
In DEQUE, insertion and deletion operations are performed at both the ends.





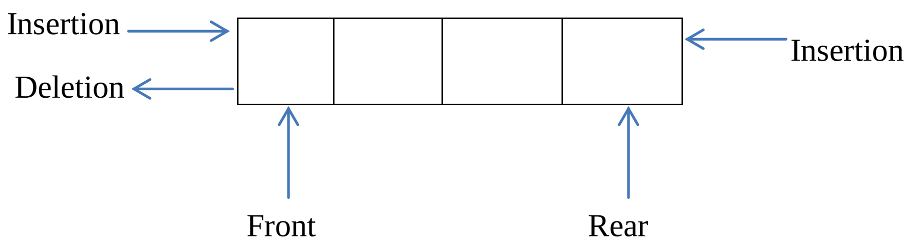
INPUT RESTRICTED DEQUE

Here insertion is allowed at **one end** and deletion is allowed at **both ends.**



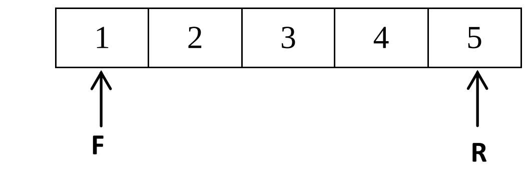
OUTPUT RESTRICTED DEQUE

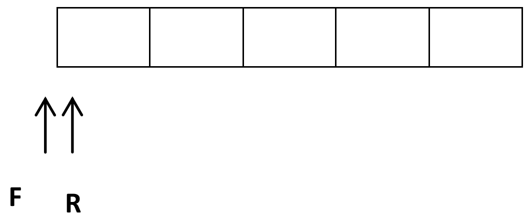
Here insertion is allowed at **both ends** and deletion is allowed at **one end.**

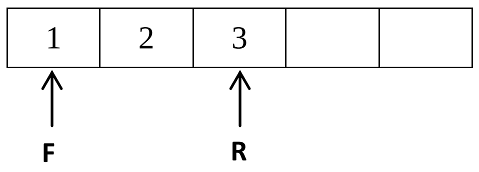


Routine For Insertion At Rear End

```
Void Insert_Rear(int X, DEQueue DQ)
{
If(REAR==Arraysize-1)
  Error("Full Queue!!!! Insertion not possible");
else if(REAR == -1)
{
        FRONT=FRONT+1;
        REAR=REAR+1;
        DQ[REAR] = X;
}
else
{
        REAR=REAR+1;
        DQ[REAR]=X;
}
}
```

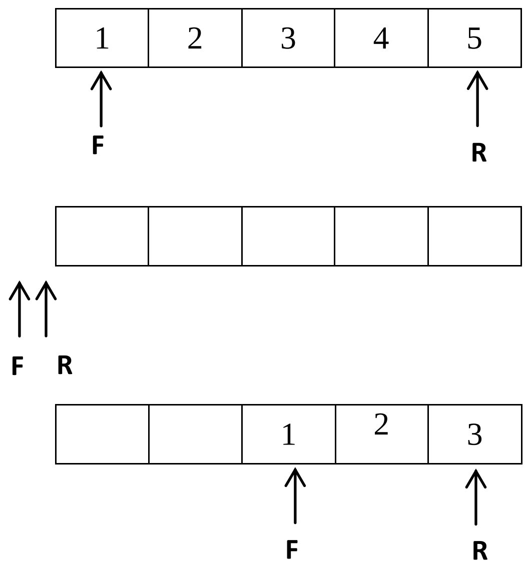






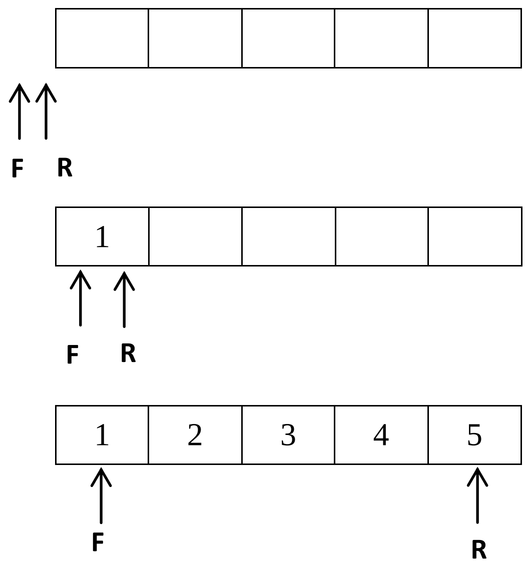
Routine for Insertion at font end

```
Void Insert_Front(int X, DEQueue DQ)
{
If(FRONT==0)
  Error("Element present in Front!!!!! Insertion not possible");
else if(FRONT == -1)
{
        FRONT=FRONT+1;
        REAR=REAR+1;
        DQ[FRONT] = X;
}
else
{
        FRONT=FRONT-1;
        DQ[FRONT]=X;
}
}
```



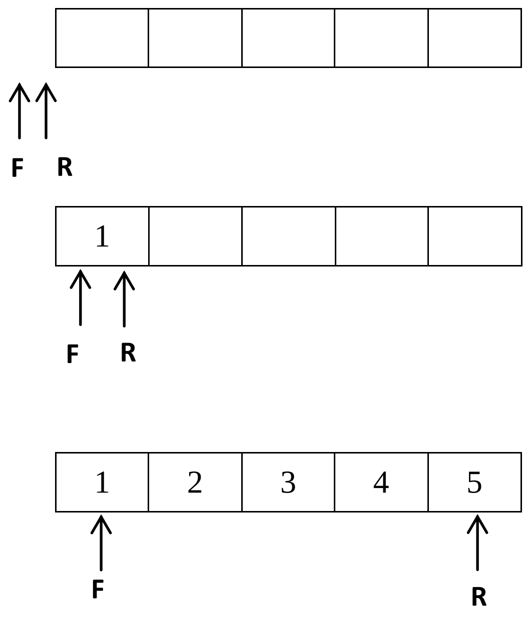
Routine for Deletion from front end

```
Void Delete_Front(DEQueue DQ)
{
If(FRONT==-1)
 Error("Empty queue!!!! Deletion not possible");
Else if(FRONT==REAR)
{
X=DQ[FRONT];
FRONT=-1;
REAR=-1;
}
Else
{
X=DQ[FRONT];
FRONT=FRONT+1;
}
}
```



Routine for Deletion from Rear End

```
Void Delete_Rear(DEQueue DQ)
{
If(REAR==-1)
 Error("Empty queue!!!! Deletion not possible");
else if(FRONT==REAR)
{
X=DQ[REAR];
FRONT=-1;
REAR=-1;
}
Else
{
X=DQ[REAR];
REAR=REAR-1;
}
}
```



APPLICATIONS OF QUEUE

* Batch processing in an operating system

* To implement Priority Queues.

* Priority Queues can be used to sort the elements using Heap Sort.

* Simulation.

* Mathematics user Queueing theory.

* Computer networks where the server takes the jobs of the client as per the queue strategy.