STACKS :-

* A stack is a linear list of elements in which an element may be inserted or deleted at only one end (ie; called as the top of the stack)
* Stack is a data structure which works on the principle Last In First Out (LIFO) ie; the last element inserted will be the first element to be deleted.
* The various operations on stack are :
  1. PUSH - inserting an element into the stack.
  2. POP - removing/deleting an element from the stack.
  3. Overflow - check whether stack is full or not.
  4. Underflow - check whether stack is empty or not.

Abstract Datatype Stack Or ADT Stack :-

* An ADT stack is the one that shows the various operations that can be performed on stack.
* An ADT stack can be written as :

ADT stack is :
Objects: A set of zero or more elements
Functions: Parameters used : STACK_SIZE ∈ Positive integer, S ∈ stack, item ∈ element

	Return_type	Functionname	operations
1.	stack	Create(STACK_SIZE)	::= Creates an empty stack whose max. size is STACK_SIZE
2.	Boolean	IsFull(S, STACK_SIZE)	::= if no. of elements in S is STACK_SIZE return True else FALSE
3.	Stack	Push(S, item)	::= if stack is not full, insert item onto stack.

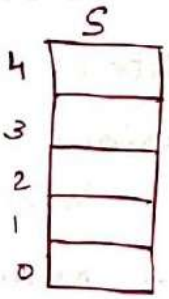
④ Boolean IsEmpty(s) ::= if no elements return True else FALSE

⑤ Element Pop(s) ::= if stack is empty return error condition as underflow, else return the element on top of the stack.

Implementation of Stacks (using Arrays):-

1.) Create(): The Create stack fn can be implemented as a 1D array ie;

```
#define SIZE 5
int S[SIZE];
int top = -1;
```



Here: SIZE → symbolic constant, specifies the max. no. of elements that can be inserted into stack.

S[SIZE]; → an array, to hold the elements of the stack.

② IsFull() (check for overflow):-

* In order to perform a Push operation we need to check whether the stack is completely filled with elements or not. ie;

```
if (top == SIZE - 1)
{
    Printf("Stack FULL - Overflow");
    return;
}
```

③ Push() :-

* Inserts an element into the stack.
* First checks whether sufficient space is available in the stack. If available then increments the value of top by 1, and then inserts the item into stack. ie;

```
Void Push()
{
    int item;
    if (top == SIZE -1)
    {
        Printf(" Stack Overflow\n");
        return;
    }
    Printf(" Enter an item\n");
    scanf(" %d", &item);
    S[++top] = item;
}
```

④ IsEmpty(): Before we delete an item from the stack, we must first check whether there are any element in the stack. ie;

```
if (top == -1)
{   printf(" Stack is Empty - Underflow\n");
    return;
}
```

∵ top = -1 indicates that the stack is empty.

⑤ pop() : To delete an item from the stack.

P.T.O

```
void pop()
{
    if(top == -1)
    {
     printf("Stack Underflow\n");
      return;
     }
    printf("Deleted item = %d ", s[top--]);
}
```

(5.) Display(): print the contents of the stack

```
void display()
{
    int i;
    if(top == -1)
     {
        pf("Stack is Empty-Underflow");
         return;
     }
    Printf("contents of the stack are:\n");
      for(i= top; i>=0; i--)
       printf("%d\n", s[i]);

}
```

⇒ Write a C program to implement stacks using arrays:

```
#include <stdio.h>
#define SIZE 5
int s[SIZE], top=-1;
```

```
void Push();
Void POP();
Void display();

int main()
{
    int ch;
    for(;;)  → infinite loop
    {
      printf(" 1. PUSH  2. POP  3. DISPLAY  4. EXIT\n");
      printf(" Enter your choice : \n");
      scanf ("%d", &ch);
      switch(ch)
      {
          case 1: Push(); break;
          case 2: POP(); break;
          case 3: display(); break;
          default: printf("Invalid"); exit(0);
      }
    }
    return 0;
}
```

```
void Push()
{
    — —
    — —
    — —
}
void POP()
{
    — —
    — —
    — —
}
```

```
void display()
{
    — — —
    — —
    — —
    — —
}
```

Stack implementation Using structures :-

```
#define SIZE 5
struct stack
{
    int a[SIZE];
    int top;
};
struct stack S;
S.top = -1;
void main()
{
    ____
    ____
    //same as previous program
    ____
    ____
}
void Push()
{
    int item;
    if(S.top == SIZE-1)
    {
        ____
        ____
    }
    printf("Enter Item:");
    scanf("%d", &item);
    S.top++;
    S.a[S.top] = item;
}
```

```
void pop()
{
    if(S.top == -1)
    {
        ____
        ____
    }
    pf("Item deleted = %d", S.a[S.top
    S.top--;
}
void display()
{
    if(S.top == -1)
    {
        ____
        ____
        ____
    }
    pf("stack contents:");
    for(i=S.top; i>=0; i--)
        pf("%d\n", S.a[i]);
}
```

Stacks Using Dynamic Arrays :-

* we can allocate or create a stack dynamically during run time — when we are not sure about the size of the stack, as follows:
* Assume that the stack size is 1 ie;

```
int stackSize = 1;
int *S;
int top = -1;
S = (int*) malloc( stackSize * sizeof(int));
Void Push()
{
    if(top == stackSize - 1)
    {
        Pf("Stack Full : Increase by 1 \n");
        stackSize++;
        S = (int *) realloc(S, stackSize * sizeof(int));
    }
    S[++top] = item;
}
Void Pop()
{
    if( top == -1)
    {   Pf(" Stack underflow");
        return;
    }
    Pf(" Item deleted = %d \n", S[top--]);
    Pf(" stack size decreased by 1 \n");
    stackSize--;
    S = (int *) realloc(S, stackSize * sizeof(int));
}
Main()
{
```

where *S ⇒ is a pointer which holds the memory locations, using malloc, realloc fns

Applications Of Stacks :-

1.) Used in Function calls:

Eg:
```
Void main()
{
    Pf("CS");
    f1();
    Pf("Job.");
}
void f1()
{
    Pf("Students");
    f2();
    Pf("Marks and");
}
void f2()
{
    Pf("Get Good");
}
```

2.) Used in conversion of expressions ie; arithmetic expressions.

3.) Evaluation of Expressions : An arithmetic expression represented in the form of either postfix or Prefix.

4.) Used in Recursion.

Applications:

Polish Notation :

* An arithmetic expression can be of 3 types:

a) Infix Expression – placing the operator b/w the operands ie; a+b

b) Prefix Expression (Polish Notation) – placing the operator before the operands eg: +ab

c) Postfix Expression (Reverse Polish Notation) – Placing the operator after the operands ie; ab+

The computer usually evaluates an arithmetic expression in 2 steps:

a) First, it converts the expression (infix) into a postfix notation.

b) Second, it evaluates the postfix expression.

) Conversion / Transform Infix exprns into Postfix Expressions :-

⇒ Algorithm: -

tep 1; Push # onto the stack.

tep 2; Scan the expression from left to right & repeat Step 3 to 6 for each element.

ep 3: If a '(' Parenthesis is encountered, Push it onto stack.

ep 4: If an operand is encountered, add it into Postfix P.

tep 5: If an operator is encountered, then:

a) check whether the precedence of the current operator on top of the stack is greater than or equal to the precedence of the scanned operator, If so then go on Popping all the operators from the stack & place them in the postfix P.

b) Otherwise (else) Push the scanned operator onto stack

tep 6: If an ')' right Parenthesis is encountered, then go on popping all the items from the stack & place them in postfix expression; till we get the matching left '(' parenthesis.

Note: Do not Push the '(' left Parenthesis onto PostfixP.

Step 7: Check if any operator or symbols are present in the stack. If so, then pop them and place it onto postfix P.

⇒ Write a C program to convert infix into postfix expressions. (Refer Lab Program 4).

Example: Convert the foll expressions into Postfix expressions using the above algorithm.

① (a * b) + c
0 1 2 3 4 5 6

Initial: top = -1 and Pos = 0

Step	Sym	Postfix P	stack	Variables (optional)
Initial	-	-	#	top = 0, Post = 0
i = 0	(	-	#, (	top = 1, Post = 0
1	a	a	#, (	top = 1, Post = 1
2	*	a	#, (, *	top = 2, Post = 1
3	b	ab	#, (, *	top = 2, Post = 2
4	)	ab*	#,	top = 0, Post = 3
5	+	ab*	#, +	top = 1, Post = 3
6	c	ab*c	#, +	top = 1, Post = 4

⇒ Place the remaining operators ie; + into the postfix. ∴ **ab*c+** ⇒ Postfix expression

2.) (A - B) * (D / E)
0 1 2 3 4 5 6 7 8 9 10

Step	Sym	Postfix	stack	Variables
Initial	-	-	#	top = 0, Post = 0
i = 0	(		#, (	top = 1, Post = 0
1	A	A	#, (	top = 1, Post = 1
2	-	A	#, (, -	top = 2, Post = 1
3	B	AB	#, (, -	top = 2, Post = 2
4	)	AB-	#	top = 0, Post = 3
5	*	AB-	#, *	top = 1, Post = 3
6	(	AB-	#, *, (	top = 2, Post = 3
7	D	AB-D	#, *, (	top = 2, Post = 4
8	/	AB-D	#, *, (, /	top = 3, Post = 4

9	E	AB-DE	#, *, (, /	top = 3, postE = 5
10	)	AB-DE/	#, *	top = 1, postE = 6

∴ postfix = AB-DE/*

Note: Precedence of arithmetic operators.

^ ⇒ P = 4

%, /, * ⇒ P = 3

+, − ⇒ P = 2

(,) ⇒ P = 1

⇒ P = 0

Problems on postfix Expressions :-

1.) 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), *, +

= 12/(7-3), (2*(1+5)), +

= 12/4 + [2*6]

= 3 + 12

= 15 //

Convert the infix expression into postfix :-

(i) a+b*c/d

= a + [bc*] /d

= a + [bc*d/]

= abc*d/+

(ii) (A-B) * (D/E)

= (AB-) * (DE/)

= AB-DE/*

(iii) $(A+B \wedge D) / (E-F) + G$

$= (A+BD\wedge) / (EF-) + G$

$= (ABD\wedge+) / (EF-) + G$

$= (ABD\wedge + EF-/) + G$

$= ABD\wedge + EF-/G+$

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

$= A*(BD+)/E-F*(G+HK/)$

$= (ABD+*)/E-F*(GHK/+)$

$= (ABD+*E/)-F*(GHK/+)$

$= (ABD+*E/)-(FGHK/+*)$

$= ABD+*E/FGHK/+*-$

II.) Evaluation Of Postfix Expressions :-

Algorithm:

Step 1: Scan the given expressions from left to right.

Step 2 (a): If the symbol is an operand then Push its value onto the stack.

(b): If the symbol is an operator, then pop out two values from the stack & assigns them respectively to Opnd 1 and Opnd 2. Then perform the required operator ie;

res = opnd1 * opnd 2

Push the result back to the stack.

Step 3: Repeat Step 2 until all the symbols are over.

Step 4: Pop out the result and print it.

Example:- Trace or Evaluate the following postfix expression.

1) 78+65+*

step	Sym	Opnd1	Opnd2	Stack
i=0	7	-	-	7
1	8	-	-	7,8
2	+	7	8	15
3	6	-	-	15,6
4	5	-	-	15,6,5
5	+	6	5	15,11
6	*	15	11	165

∴ Result = 165//

2.) 5, 6, 2, +, *, 12, 4, /, −

	Sym	Opnd1	Opnd2	Stack
i=0	5	-	-	5
1	6	-	-	5,6
2	2	-	-	5,6,2
3	+	6	2	5,8
4	*	5	8	40
5	12	-	-	40,12
6	4	-	-	40,12,4
7	/	12	4	40,3
8	−	40	3	37

∴ Result = 37//

Recursion :- It is a programming technique in which the function calls by itself.

Types of Recursion: (1) Direct Recursion (2) Indirect Recursio

P.T.O

1.) Direct Recursion

```
Void main()
{
  Pf("Happy Dasara");
  main();
}
```

2) Indirect Recursion

```
void main()
{
  Pf("Happy Dasara");
  f1();
}
f1()
{
  main();
}
```

Requirements of Recursion :-

① Base criteria / Terminating condition - for which the procedure does not call itself.

② Each time the procedure does call itself, it must be closer to be base criteria.

③ stack.

Recursion can be implemented for the foll techniques:

1.) Factorial of a Number :-

Denoted by $n! = 1.2.3 \ldots\ldots (n-2)(n-1).n$

Factorial function can be defined as:

a) If $n=0$, then $n! = 1$

b) If $n>0$, then $n! = n.(n-1)!$

Program:

```
int fact(int n)
{
    if(n==0)
        return 1;
    else
        return (n * fact(n-1));
}
```

```
main()
{
    int n;
    printf(" Enter a number \n");
    scanf("%d", &n);
    printf(" Factorial = %d", fact(n));
}
```

Output:

fact(5) → 120

⇩

5 * fact(4) → 24

⇩

4 * fact(3) → 6

⇩

3 * fact(2) → 2

⇩

2 * fact(1) → 1

⇩

1 * fact(0) → 1

2.) Fibonacci Sequence :-

The fibonacci sequence is as follows:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ....

ie; $F_0 = 0$ and $F_1 = 1$, and each succeeding term is the sum of the two preceding terms.

Definition: (Fibonacci)

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

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

Program:

```
int fib(int n)
{
    if(n==0) return 0;
    if(n==1) return 1;
    else
    return (fib(n-1) + fib(n-2));
}
```

```
main()
{
    int n;
    Printf("Enter the value of n");
    Scanf("%d", &n);
    Printf("Fibonacci=%d", fib(n));
}
```

3.) GCD and LCM of 2 Numbers:-

```
#include <stdio.h>
ind gcd(int, int);
int main()
{
    int m,n;
    Pf("Enter the values of m, n \n");
    sf("%d %d", &m, &n);
    Pf("GCD of 2 Numbers is %d", gcd(m,n));
    return 0;
}
int gcd(int a, int b)
{
    if(a==0) return b;
    if(b==0) return a;
    if(a>b)
        gcd(a%b, b);
    else
        gcd(a, b%a);
}
```

o/p:

gcd(15, 45)

↓

gcd(15, 45%15) → 0

o/p = 15

) Tower Of Hanoi :- Recursion may be used as a tool in solving the problem of Tower of Hanoi.

Problem statement : Consider 3 poles (Pegs) A, B and C. There are 'n' disks on pole A of different parameters and are placed one above the other, such that always smaller disk is placed above the larger disk.

Initially pole B and C are empty.

ules: (1). only one disk is moved at a time. ie; move only the top disk on any peg.

) Smaller disk is on top of the larger disk at any time.

) A disk can be moved from one pole to another only.

Solution :

) If there is only one disk, ie: n=1, then move that disk from A to C. (A → C)

.) If there are two disks ie: n=2, then move one disk from A to B, 2nd from A to C and then from B to C.

ie; A → B
A → C
B → C

3.) In general to move 'n' disks from A to C, the following recursive technique is used:

a) Move the top (n-1) disks from A to B. [A → B]

b) Move the nth disk from A to C. [A → C]

c.) Move the (n-1) disks from B to C. [B → C]

xample: For n=3; the solution consists of the following 7 moves:

(i) Move top disk from A to C
(ii) Move top disk from A to B
(iii) Move top disk from C to B
(iv) Move top disk from A to C
(v) " " " B to A
(vi) " " " B to C
(vii) " " " A to C

Algorithm for Tower of Hanoi

TOWER(n, BEG, AUX, END)

step 1 : If n = 1, then
a) write : BEG → END
b) Return

step 2: [Move n-1 disks from peg BEG to peg AUX]
call Tower(n-1, BEG, END, AUX)

step 3: write BEG → END

Step 4: [Move n-1 disks from peg AUX to peg END]
call Tower(n-1, AUX, BEG, END)

Step 5: Return.

4) Program to implement Tower of Hanoi:

Refer Lab. program 5.(b)

5) Ackermann Function :-

* The ackermann function is a function with 2 argument each of which can be assigned any non-negative integer: 0, 1, 2 . . . . This function is defined as:

a) If m = 0 ; then A(m, n) = n + 1

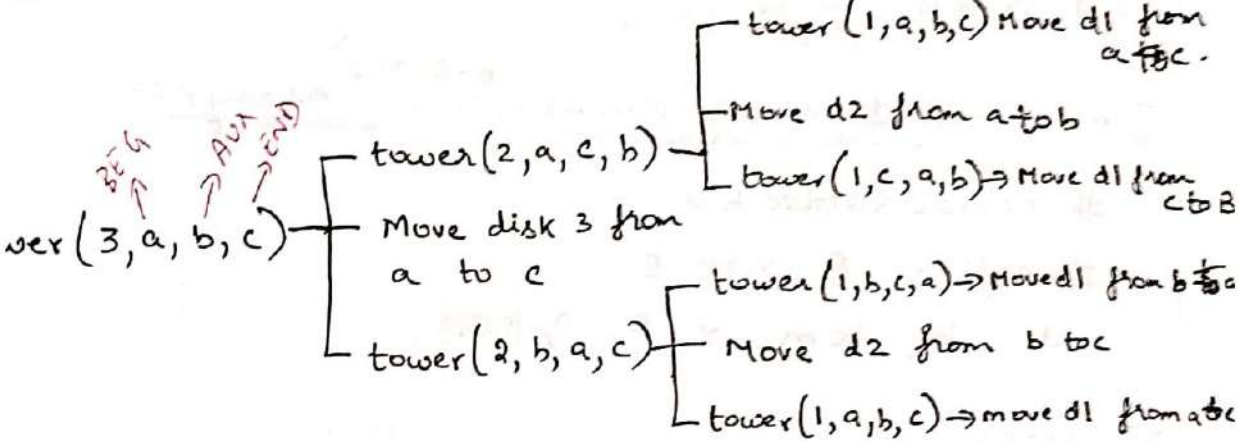
b) If $m \neq 0$ and $n \neq 0$, then

$$A(m,n) = A(m-1, A(m, n-1))$$

observe that $A(m,n)$ is explicitly given only when $m=0$. The base criteria are the pairs: (0,0), (0,1), (0,2) ....

The ackermann fn is too complex to evaluate and its importance comes from its use in mathematical logic.

Tracing for Tower of Hanoi : n=3



QUEUES :-

Queue is a linear list data structure which works on the principle FIFO (First In First Out) ie; element that is present in the queue for the longest time will get deleted first.

Here insertion and deletion takes place at different ends : rear and front.

rear : insertion into the queue takes place at the rear (back) end.

front : deletion from the queue takes place at the front end.

Advantages :- Used in application and System Software development. Eg: CPU process the jobs in FIFO order, Printer prints the file in FIFO, OS allocates the memory for the objects in FIFO order.

Types of Queues :-

1.) Ordinary Queues (Linear)
2.) Double ended Queues
3.) Circular Queues
4.) Priority Queues.

Implementation of Queues :-

1.) static Memory allocation – arrays, structures
2.) DMA – linked list.

Implementation of Queues (ordinary) using arrays :-

```
#include <stdio.h>
#define Q_SIZE 5
int ch, item, r, f, q[Q_SIZE];
```

i) Insert() :-

Algorithm : Step 1: Before inserting item, first check for overflow ie;
if (r == Q_SIZE - 1)
Step 2: Accept item
3: Increment r by 1 ie; r=r+1
4.) Assign q[r] = item
5.) end

Function

```
void insertrear()
{
  int item; r = -1;
  if( r == Q_SIZE - 1)
    {
      Pf(" Q overflow");
      return;
    }
```

```
    Printf("Enter the item:\n");
    scanf("%d", &item);
     q[++r] = item;
}
```

) delete():

Algorithm : Step 1: Check for underflow ie; if(f > r)

Step 2: Delete item ie; q[f]

Step 3. Increment f by 1 : f = f+1

Step 4. End

i function

```
void deletefront()
{
   f=0;
   if(f>r)  (or) if((f==0)&&(r==-1))
   {
     Pf("Q is empty"); return;
   }
   Pf("Element deleted =%d\n", q[f++]);
}
```

) display():

```
void display()
{  int i;
   if(f>r)  (or) if((f==0)&&(r==-1))
    {
      Pf("Q is Empty"); return;
    }
   Pf("contents of Queue are:\n");
     for(i=f; i<=r; i++)
       Pf("%d\n", q[i]);
}
```

using arrays:

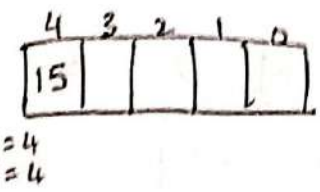
```
#include<stdio.h>
#define Q_SIZE 5
int ch, r, f, i, item, q[Q_SIZE];

void insertrear();
void deletefront();
void display();

void main()
{
   f = 0; r = -1;
   for(;;)
   {
      Pf("1. INSERT 2. DELETE 3. DISPLAY 4. EXIT\n");
      Pf("Enter your choice\n");
      Sf("%d", &ch);
      switch(ch)
      {
        case 1: insertrear(); break;
        case 2: deletefront(); break;
        case 3: display(); break;
        case 4: exit(0); break;
        default: Pf("Invalid choice");
      }
   }
}
```

```
void insertrear()
{
   ____ _
   ___ _ _
   _ _ _
}
void deletefront()
{
   ____ __
   ____ __
}
```

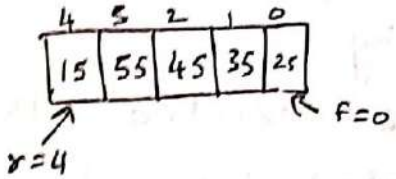
```
void display()
{
   ____ _
   ___ _
   _____
}
```

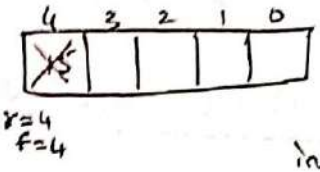
Advantages of OQ:- Simple and easy to implement.

Disadvantages:

① 

Here insertion is not possible even though enough empty spaces are available

② 



After deleting 15, f=5, r=4 now insertion is not possible even though entire Q is empty

Solution: Reset f = 0, r = -1, when f > r

ADT (ABStract Data Type) Queues

Objects: a finite Ordered list with zero or more elements.

Functions:

① Queue Create(Q_SIZE) ::= Create an emty queue whose max size is Q_SIZE

② Boolean IsFullQ(queue, Q_SIZE) ::= if(no.ofele == Q_SIZE) return TRUE else FALSE

③ Queue AddQ(queue, item) ::= if (IsFull(queue)) queue full else insert item at rear end.

④ Boolean IsEmptyQ(queue) ::= if (queue == create(Q_SIZE)) return TRUE else FALSE.

⑤ Element DeleteQ (queue) ::= if (IsEmptyQ(queue)) return true else remove item at front end.

CIRCULAR QUEUES:- (CQ)

* It is one of the best data Structures which uses FIFO. ie; elements are stored efficiently in an array so as to wrap around - so that end of the queue is followed by the front of the queue.
* CQ is used in application & system S/w development. It overcomes the drawback of OQ.
In OQ, if r == SIZE-1; insertion is not possible

Whereas in CQ, we can insert the data in circular fashion. ∴ the memory is used optimally.

* Before insertion the value of r is calculated by : $r = (r+1) \% SIZE$

* After deletion the value of f is calculated by

$$f = (f+1) \% SIZE$$

* To check for overflow & underflow, global variable count is maintained.

If count == SIZE → then Q is Full.

If count == 0 , → Q is empty.

* The operations of CQ are:
  - insert
  - delete
  - display.

Implementation of Circular Queues (using arrays)

```
#define SIZE 5
int q[SIZE], f=0, r=-1, count=0;
Void insertcq()
{
    int item;
    if(count == SIZE)
    {
      Pf("Q is FULL"); return;
    }
    r = (r+1) % SIZE;
    Pf("Enter the item\n");
    sf("%d", &item);
    q[r] = item;
    count++;
}
```

```
void deleteCQ()
{
    if( count == 0)
    {
        Pf(" Q is empty ");
        return;
    }
    Pf(" Element deleted = %d\n", q[f]);
    f = (f+1) % SIZE;
    count--;
}
void display()
{
  int i, j;
  if(count == 0)
  { Pf(" Q is empty "); return;
  }
  j = f;
  Pf(" contents of Q are: ");
     for(i = 1; i <= count; i++)
     { Pf("%d\n", q[j]); j = (j+1) % SIZE;
     }
  }
```

Advantages: optimal utilization of memory.

Disadvantage: ① May become infinite loop.
② Keeping track of r and f is difficult.

Circular Queues using Dynamic Arrays:-

* The various operations that can be performed on CQ using dynamic arrays are:

① Create(): Let us assume Q_SIZE is 1

```
int QSIZE = 1;
int *q, f = 0, r = -1, count = 0;
```

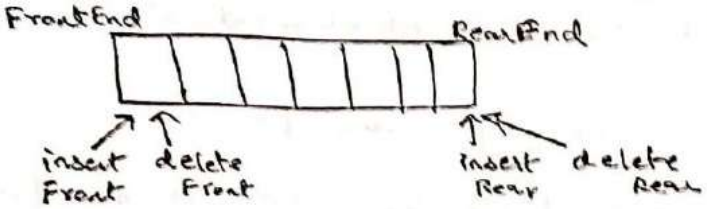
② InsertQ(): Before inserting, check for sufficient space in the queue. But once the queue is Full, we can increase the size of Queue using realloc()

```
Void insertQ()
{
    if (count == QSIZE)
    {
        Pf("Q Full - Increase Size by 1");
        QSIZE ++;
        q = (int*) realloc (QSIZE, sizeOf(int));
    }
    if (f > r)
    {
        for (i = QSIZE - 2; i >= f; i--)
        {
            q[i+1] = q[i];
        }
        f++;
    }
    r = (r+1) % QSIZE;
    q[r] = item;
    count++;
}
```

* Delete() and display() — are same as in CQ arrays.

DEQUEUES (Double-ended Queues / deck / deques)

* A deque is a linear list in which elements can be added or removed at either end.
* Insertion is possible at both front and rear ends. Similarly deletion can be done at front and rear end.



* The Operations performed are:

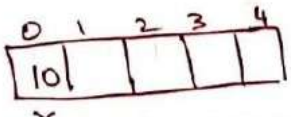
a) InsertFront()
b) InsertRear()
c) DeleteFront()
d) DeleteRear()
e) display.

Implementation of Double-ended Queue: Using Arrays

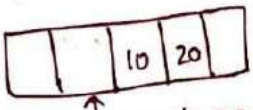
```
#define SIZE 5
int q[SIZE];
int f=0, r=-1;
```

a) InsertFront(): insertion of element at the front end.

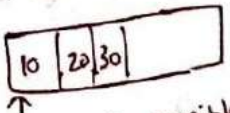
```
Void insertfront()
{
    if(f==0 && r==-1)  // q is empty then insert item
    {
        q[++r] = item;
        return;
    }
    if( f != 0)        // Insert when items are Present
    {
        q[--f] = item;
        return;
    }
    Printf("Front Insertion Not Possible");
}
```



Insert item=10 after ++r



insert new item before the value 10



insertion not possible; an item already exists at Front

b) Insertrear(): Insertion of element at rear end

// Refer Ordinary Queues for insertrear

c) deleteFront(): deleting an item at the front

// Refer Ordinary Queue for deletefront

d) deleterear(): deleting an item at rear end.

```
void deleterear()
{
  if(f>r)
  {
    Pf("Queue underflow");
    f=0, r=-1; //reset Queue
    return;
  }
  Pf("Element deleted = %d\n", q[r--]);
}
```

e) display()

```
void display()
{ int i;
  if(f>r)
  {
    Pf("Q is Empty - Underflow");
    f=0; r=-1;
    return;
  }
  Pf("Contents of Queues are:");
  for(i=f; i<=r; i++)
    Pf("%d\n", q[i]);
}
```

main() function for Deque

```
void main()
{ int ch;
  for(;;)
  {
    Pf(" 1.INSERTFRONT 2.INSERTREAR 3. DELETEFRONT
        4. DELETEREAR - 5.DISPLAY 6. EXIT \n");
    Pf(" Enter your choice:");
    sf("%d", &ch);
    switch (ch)
    {
      case 1: Pf(" Enter Item \n"); Sf("%d", &item);
              Insertfront(); break;
      case 2: Pf(" Enter Item \n"); sf("%d", &item);
              Insertrear(); break;
      case 3: deletefront() break;
      case 4: deleterear(); break;
      case 5: display(); break;
      default: exit(0);
    }
  }
}
```

PRIORITY QUEUES :-

* The priority queue is a special type of data structure in which items can be deleted or inserted based on priority.
* Always an element having highest priority is processed first.
* If the elements in the queue are of the same priority, then the element which is inserted first into the queue is processed.

Applications :-

① Used in Operating Systems for the design of Job Scheduling Algorithms.

Priority Queues are classified into:

a) Ascending Priority Queue - Here elements are inserted in any order. But while deleting an item from the queue, only the smallest element is removed first.

b.) Descending priority Queue - Here also elements are inserted in any order. But while deleting an element, only the largest element is deleted first.

Implementation of Priority Queue

Design 1: Using ascending Priority queue - where the smallest element is removed first. Here the elements are added at the rear end.

Design 2: The 2nd technique is to insert the items based on the priority. Here we assume the item to be inserted itself denotes the priority. So the items with least value can be considered as the items with highest priority and elements with highest ~~priority~~ value can be considered as the items with least priority. So, we insert the elements into queue in such a way that they are always ordered in increasing order. Hence highest Priority of elements are at the front end hence delete elements at the front end.

Priority Queue

```
Void insert()
{
  int f;
  if( r == SIZE - 1)
    { pf(" Q is FULL"); return;
    }
      j = r;
    while( j >= 0 && item < q[j]) // Find appr posit for the new ite
      {
        q[j+1] = q[j];
          j--;
      }
      q[j+1] = item;
      r = r+1;
}
```

Complete Program

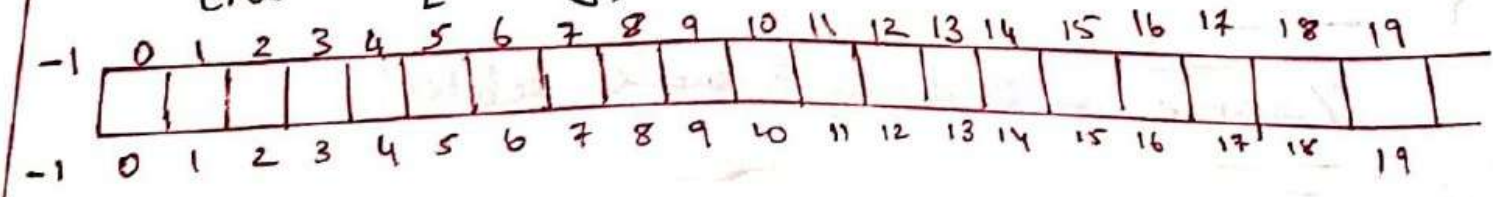
```
#include <stdio.h>
#define SIZE 5
int q[SIZE], f=0, r=-1, item, ch;

Void insert()
{
  - -
  ---
  ---.
}
Void deletefront()
{
  ---
  ---
  //Same as Ordinary Queues's - delete()
  --.
}
```

```
void display()
{
    ---------
    // same as QQ's - display
}
main()
{
    for(;;)
    {
        pf("1. INSERT 2. DELETE ---- \n");
        pf(" Enter UR choice ");
        sf("%d", & ch)
        switch (ch)
        {
            case 1: pf(" Enter Item: \n");
                    sf("%d", & item);
                    insert();
                    break;
            case 2 : delete(); break;
            case 3 : display(); break
            default : exit(0);
        } }
}
```

Multiple Stacks and Queues :-

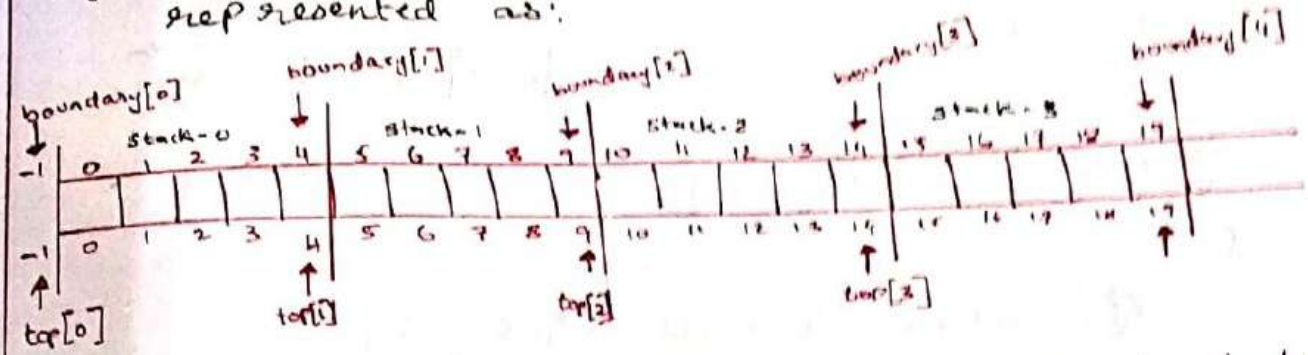
Let us, implement multiple stacks using arrays :

```
# define SIZE 20
int s[SIZE];
```



* Now let us divide the array into 'n' stacks.

Eg: If n = 4, then 4 empty stacks can be pictorially represented as:



* To find the initial value of top of each stack, i.e; top[0], top[1], top[2], top[3], use the expression:

$$top[j] = size/n * j - 1 \quad \text{where } j = 0 \text{ to } n$$

ie; $j = 0$; $top[0] = 20/4 * 0 - 1 = -1$

$j = 1$; $top[1] = 20/4 * 1 - 1 = 4$

$j = 2$; $top[2] = 20/4 * 2 - 1 = 9$

$j = 3$; $top[3] = 20/4 * 3 - 1 = 14$

∴, we can write as follows:

```
for(j=0; j<=n; j++)
  {
    top[j] = SIZE/n * j - 1;
  }
```

* To find the boundary, of each stack, copy initial value of top[0] to boundary[0], and so on.

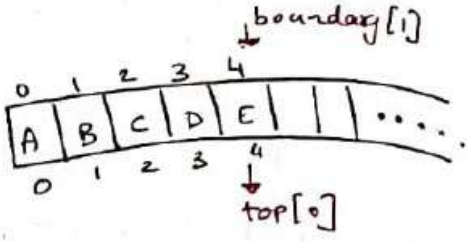
ie;

```
for(j=0; j<=n; j++)
  {
    boundary[j] = top[j];
  }
```

OR

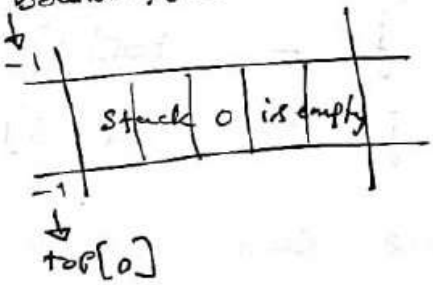
```
boundary[j] = top[j] = SIZE/n * j - 1;
```

* Push() : First check for overflow then insert item.




```
Void push()
{
  if(top[i] == boundary[i+1])
  {
     Pf("Stack is full in %od", i);
      return;
  }
  s[++top[i]] = item;
}
```

* Pop() : First check for underflow, then try to delete.




```
Void POP()
{
  if(top[i] == boundary[i])
  {
     Pf("Stack %d is empty", i);
      return;
  }
  Pf("Item deleted = %d", s[top[i]--]);
}
```

* display()

```
void display()
{
    if(top[i] == boundary[i])
```

```
{
    pf("Stack %d is empty", i)
    return;
}
for(j = boundary[i]+1 ; j<= top[i] ; j++)
    pf("%d\n", s[j]);
}
```

complete program

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 20
int s[SIZE];
#define MAX_STACKS 10
int boundary[MAX_STACKS];
int top[MAX_STACKS];
int n, i, j, item, ch;

Void push()
{
    ____ ____
    _____
    ____ _
}

Void pop()
{
    _____
    _____
    _____
}
Void display()
{ _____
```

```
void main()
{
    pf("Enter No of Queues\n");
    sf("%d", &n);
    for(j=0; j<=n; j++)
boundary[j] = top[i] = SIZE/n*j-1;
    for(;;)
    {
    pf("stack Number: ");
        for(j=0;j<n; j++)
            pf("%d", j);
pf("Enter stack No : To Peyform Operations\n");
sf("%d", &i);
pf("1:Push 2: Pop 3: display 4: Exit \n");
pf("Enter UR choice\n");
sf("%d", &ch);
switch(ch)
{ case 1: pf("Enter Item"); sf("%d", &ite
            push(); break;
 case 2: pop(); break;
 case 3: display(); break;
 default : exit(0);
 }
}
```

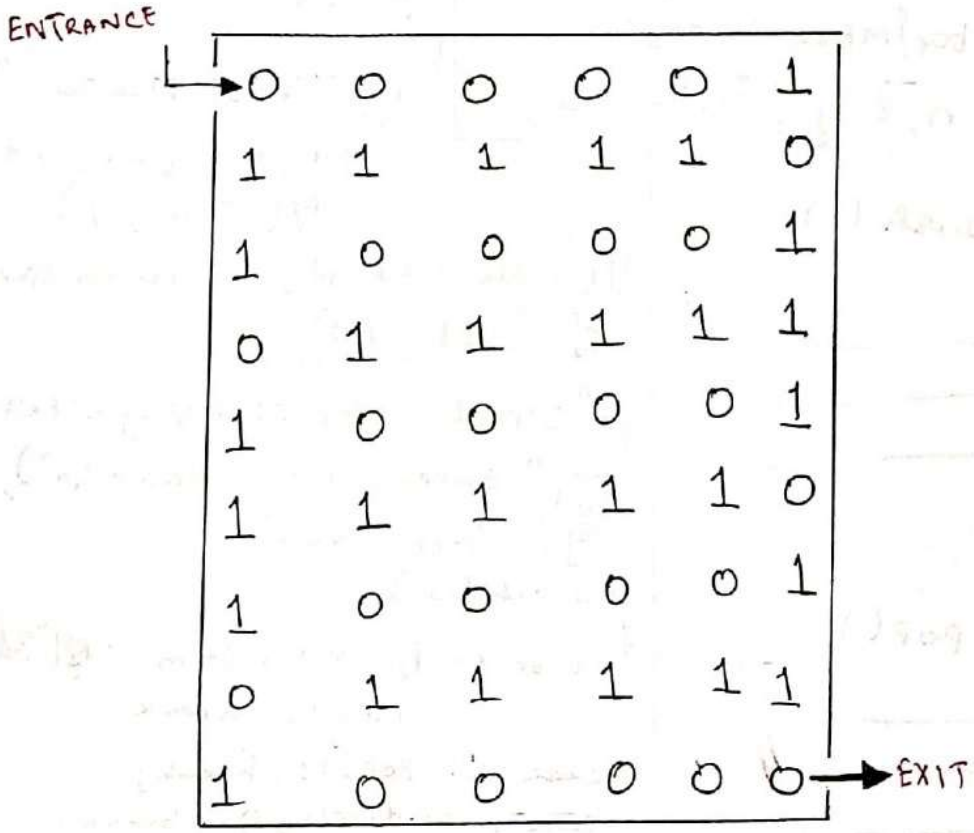
The Mazing Problem :-

* A maze is a confusing network of paths, through which it is very difficult to find one's way.
* The experimental psychologists train the rats to search mazes for food. A maze problem is a nice application of stack.

Maze Representation: A maze is represented as a two dimensional array in which:

- Zero's (0's) represent the open paths
- One's (1's) represent barriers

Example: A simple maze, along with path is shown:



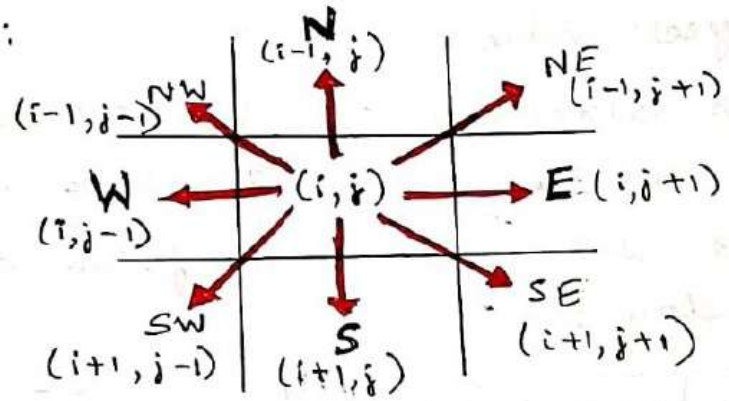
* We assume that the rat starts at the top left and is to exit at the bottom right.

* The maze can be represented as 2D array, the location of the rat in the maze can be determined by the row and column position.

* Let (i, j) is the current position of rat in the maze where i is the row index and j is the column index.

* Now the rat can move in 8 possible directions: North, Northeast, Southeast, South, Southwest, West and Northwest ie: N, NE, E, SE, S, SW, W, NW resp.

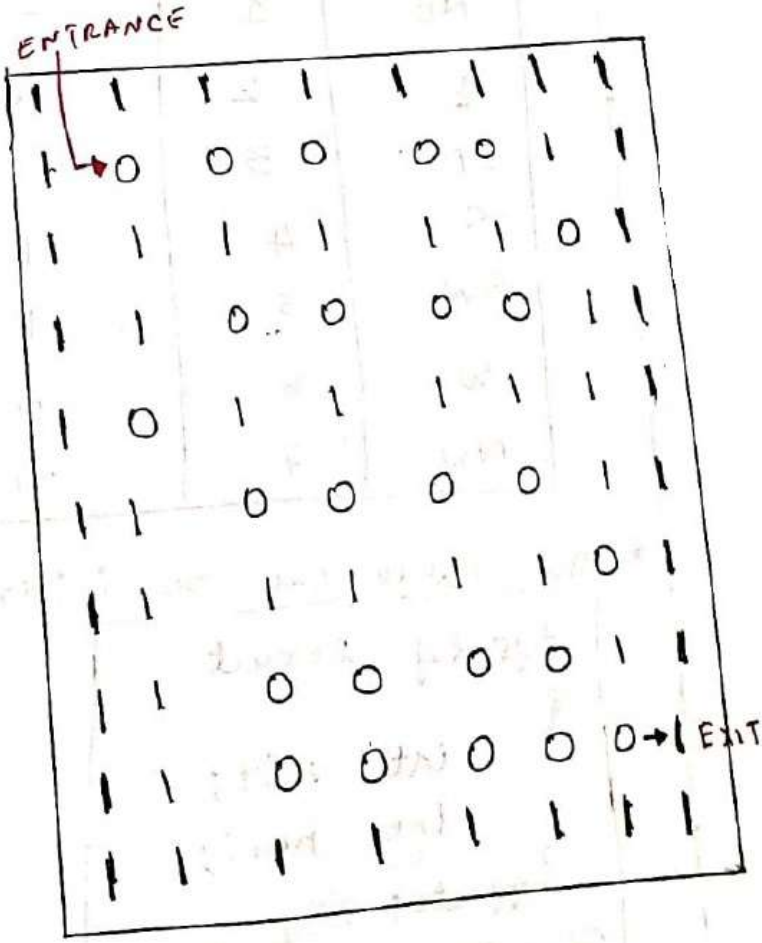
Figure:



* The rat can be moved in all 8 directions. But, this is not true all the time.

If the position of rat is on any of the border, then it cannot move in all 8 directions.

To avoid checking for these border conditions, we can surround the maze by a border of 1's as shown in figure.



Thus, a maze of m×n will require (m+2) × (n+2) size.

* Now the rat starts at position (1,1) and leaves at bottom right of the maze from the position (9,6).
* Observe from the 8 directions figure, that either we add −1, or 0 or 1 to (i,j) to get new position. These values are called Offset values.
* Thus, the 8 directions can be represented using the numbers 0 to 7 along with above offset values in the form of a table below:

Name	dir	move[dir].vert	move[dir].hori
N	0	−1	0
NE	1	−1	1
E	2	0	1
SE	3	1	1
S	4	1	0
SW	5	1	−1
W	6	0	−1
NW	7	−1	−1

* The table can be initialized using array of structures

```
typedef struct
{
    int vert;
    int hori;
} offsets;
Offsets move[8];
```

* To find more moves from the current position, we can find the next valid position of rat in the maze at a particular direction using:

```
nextRow = row + move[dir].vert;
nextCol = col + move[dir].hori;
```

* As we move through the maze, we have the choice of several directions of movement. Since we do not know which choice is best, we save our current position and arbitrarily pick a possible move. By saving our current position, we can return to it and try another path if we take a wrong path. We examine the possible moves starting from the north and moving clockwise. We use a 2D array mark - to record the maze positions already checked. We initialize this array's entries to zero. When we visit a position maze[row][col], we change mark[row][col] to 1.

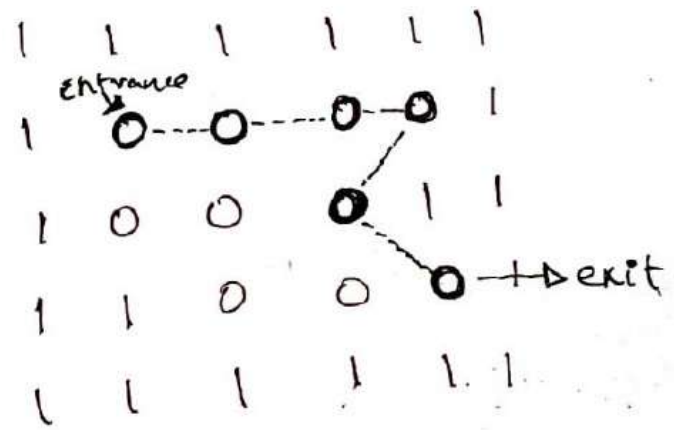
Program:

```
#define EXIT_ROW 12
#define EXIT_COL 15
typedef struct
{
    int vert;
    int hori;
} offsets;
offsets move[8] = { {-1, 0}, {-1, 1}, {0, 1}, {1, 1}, {1, 0},
                    {1, -1}, {0, -1}, {-1, -1} };

typedef struct
{
    int row;
    int col;
    int dir;
} element;
enum {FALSE, TRUE};
```

```
int maze[20][20] = {  0  1  2  3  4  5
                  0 { 1, 1, 1, 1, 1, 1},
                  1 { 1, 0, 0, 0, 0, 1},
                  2 { 1, 0, 0, 0, 1, 1}
                  3 { 1, 1, 0, 0, 0, 1},
                  4 { 1, 1, 1, 1, 1, 1}
                  };

void main()
{
   int r, c, nextRow, nextCol, dir, found = FALSE, top, i;
   int mark[20][20] = {0};
   element s[200], pos;
   mark[1][1] = 1;
   pos.row = 1, pos.col = 1;     // Rat starts as Pos(1,1)
   pos.dir = 0;
   s[top = 0] = pos;             //Push the currentPos onto stack
   while (top != -1 && !found)
   {
      pos = s[top--];
      row = pos.row;
      col = pos.col;
      dir = pos.dir;
     while(dir < 8 && !found)
     {
        nextRow = row + move[dir].vert;
        nextCol = col + move[dir].hori;
       if(nextRow == EXIT_Row && nextCol == EXIT_Col)
          found = TRUE;
```

```
else if ((MAZE[nextRow][nextCol] == 0 && mark[nextRow]
                                        [nextCol] == 0)
  {
     mark[nextRow][nextCol] = 1;
     Pos.row = row, Pos.col = col, Pos.dir = ++dir;
     S[++top] = Pos;
     row = nextRow, col = nextCol, dir = 0;
  }
  else
    ++dir;
 }
}
if(found)    //Print the path position
{
     for(i=0; i <= top; i++)
          Pf("%d %d\t", S[i].row, S[i].col);

          Pf("%d %d\t", EXIT_ROW, EXIT_COL);
  }
  else
     Pf("Maze has no path\n");
}
```

Resultant Path for the above example



Path of Movements:
(1,1), (1,2), (1,3), (1,4), (2,3), (3,4)