Unit 3

★ **Queue :-** Linear data structure in which insertion is done at the end of point k/a the rear and deletion at the first end point. k/a the front.

- Front and rear are Queue pointers.

2 operations of queue in array :-

1. Insert
2. Delete

1. QINSERT (Queue , N , Front , Rear , Ele).

1. If Rear = N then
   write : overflow and exit.

2. If Front = 0 then
   Front := Rear := 1

   else
   rear := rear +1

3. Set Queue [rear] := Ele

4. Exit.

2. QDELETE (Queue, N, Front , Rear , ele)

1. If Front = 0 then
   write : underflow and exit

2. Set element := Queue [Front]

3. If front = Rear
Set front := rear := 0.

else
Set front := front + 1

4. Exit.

★ Circular Queue :- In circular queue all nodes are treated as circular. Last node is connected back to the first node. Circular queue is also k/a Ring Buffer.

- It is an abstract data types.
- Circular queue contains a collection of data which allow insertion of data at the end of the queue and deletion of data at the begining of the queue.

2 operation of circular queue in an array :-

1. Insert
2. Delete

1. Insert (Queue, N, Front, Rear, ele)

1. If (FRONT = 1 and Rear = N) or (Front = Rear + 1)
write : overflow and exit.

2. If front = 0 then
front := Rear := 1

else
rear := rear + 1

3. Set Queue [Rear] := Ele.
4. Exit

2. DELETE (Queue, N, Front, Rear, Ele)
1. If front = 0 then
   write: Underflow and exit

2. Set ele := Queue [Front]

3. If front = Rear
   Set front := Rear := 0

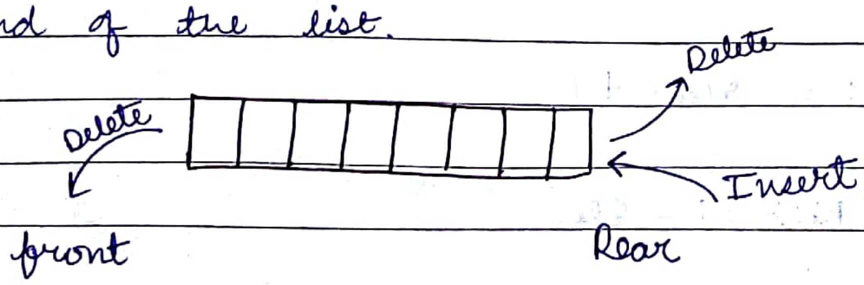
   else if front = N then
   Set front := 1

   else
   front := front + 1.

4. Exit.

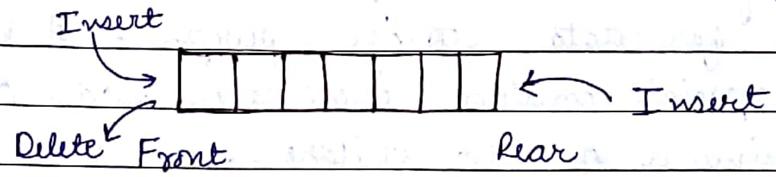
★ DEQueue :- (Double ended Queue) :→
In double ended Queue, insert and delete operation can be occur at both ends that is front and rear of the queue.

There are 2 types of DEQueue :-
1. Input restricted Queue :-
- It is a dequeue, which allows insertion at only 1 end, rear end.
- It allows deletion at both ends, rear and front end of the list.



2. Output restricted Queue :-

- It is a dequeue, which allows deletion at only one end front end
- It allows insertion at both ends, rear and front ends, of the list



★ Priority Queue :-

For Priority queue items are ordered by key values so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa.

iority/Queue	1	2	3	4	5	6	7	8
1		AA						
2	BB	CC	DD	LL				
3	MM							
4	HH	KK						
5					EE	FF	GG	PP
6			II	JJ				

X

★ TREE :- It is a non-linear data structure.

• Tree represents the node connected by edges.

1. Binary tree :- Binary Tree is a special datastructure used for data storage purpose. A binary tree is a special condition that each node can have a maximum of two children.

2. complete binary tree :- A complete binary tree is a tree that is completely filled, with the possible exception of the bottom level. The bottom level is filled from left to right.

Binary Search Tree :- A binary search tree is a binary tree it may be empty. If it is not empty then it satisfy the following properties.

(i) every node (element) has the key (value) and no two element (node) have the same value.

(ii) The value in the left sub tree are smaller than ~~the~~ value in the root.

(iii) The values in right sub tree are greater than the value in the root.

(iv) The left and right sub tree are also binary search tree.

Representation of a binary tree :-

1. Array
2. linked list.

Traversing in Binary search tree :-

1. Inorder Traversing Left Node Right
2. Pre order Traversing Node Left Right
3. Pre order Traversing Left Right Node

Inorder Traversing using recursion :-

```
Void inorder ( node * root)
{       if ( root != NULL)
        {
            inorder ( root → left );
            printf ("%d", root → info);
            inorder ( root → right);
        }
}
```

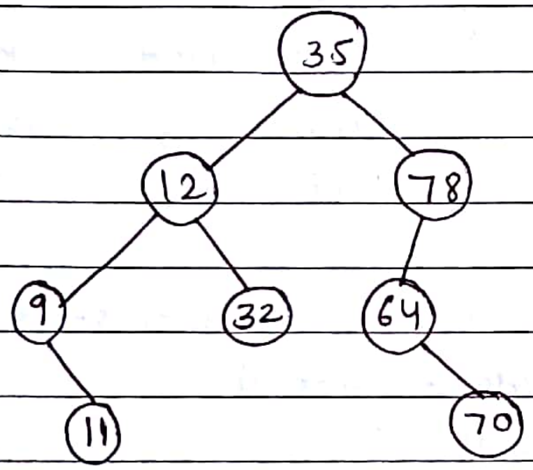
(1) Traversing in a Binary tree :-

(a) Inorder Traversing using stack :-

INORDER (Info, left, right, root).

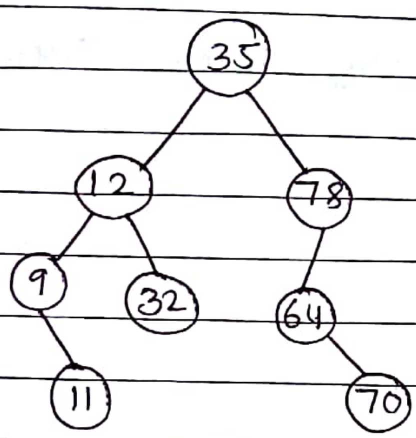
1. Set top := 1 , stack [1] := NULL and Ptr := start
2. Repeat while ptr != NULL
   a) Set TOP := TOP + 1 and stack [top] := ptr.
   b) Set Ptr := left [Ptr].
3. Set Ptr := Stack [Top] and Top := Top - 1.
4. Repeat steps 5 to 7 while Ptr ≠ NULL
5. Apply process to info [Ptr]
6. If right [Ptr] ≠ NULL then

(a) Set Ptr := right [Ptr]
(b) Go to step 2
7. Set Ptr := stack [Top] and Top := Top - 1.
8. Exit.



Inorder :- 9, 11, 12, 32, 35, 64, 70, 78.

(b) Preorder Traversing using stack :-
PREORDER (Info , left , right , root)
1. Set top := 1 , stack [1] := NULL and Ptr := ~~stack~~ start
2. Repeat step 3 to 5 while Ptr != NULL
3. Apply process to Info [Ptr]
4. If right [Ptr] ≠ NULL then
   set top := top + 1 and stack [top] := right [Ptr]
5. If left [Ptr] ≠ NULL then
   Set Ptr := left [Ptr]
   else
   Set ptr := stack [top] and top := top - 1.
6. Exit.

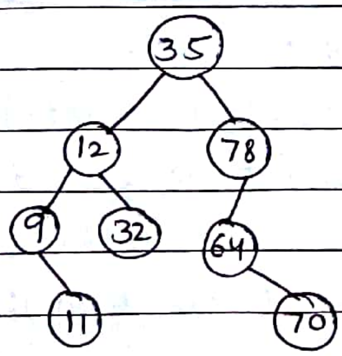


Preorder :-

(c) Postorder Traversing using stack :-

POSTORDER (info, left, right, root).

1. Set TOP := 1, stack[1] := NULL and Ptr := root.
2. Repeat steps 3 to 5 while Ptr $\neq$ NULL.
3. Set TOP := TOP + 1 and stack[TOP] := ptr
4. If right[Ptr] $\neq$ NULL then
   Set TOP := top + 1 and stack[Top] := − right[Ptr].
5. Set Ptr := left[Ptr]
6. Set Ptr := stack[top] and top := top − 1.
7. Repeat while Ptr > 0
   (a) Apply process to info[Ptr]
   (b) Set Ptr := stack[top] and top := top − 1.
8. If Ptr < 0 then
   (a) Set ptr := − ptr
   (b) Go to step 2.
9. Exit.



Operations on Binary search tree :-

1. Traversing
2. Searching
3. Insertion
4. Deletion

```
Algorithm Rsearch
{
    if (t=0) then
       return 0;
    else if (x = t → data) then
         return t;
    else if (x < t → data) then
         return RSearch (t→ lchild, x);
    else
         return RSearch (t → rchild, x);
}
```

```
Algorithm iterative lsearch ( root, x)
{
    found : = false
       t: = root
       while ((t ≠0) and not found) do
    {
         if (x = (t → data)) then
              found := true;
         else if (x < (t → data)) then
              t: = (t → lchild);
           else
              t: = (t → rchild);
    }
```

```
if (not found) then
        return 0
else
        return t;
}

Algorithm Insertion (root, x)
{
    found := false;
      p := root;
      while (p ≠ Null and not found) do
      {
            q := p
            if (x = (p → data)) then
            found := true;
            else if (x < (p → data)) then
                p := (p → lchild);
            else
                p := (p → rchild);
      }
        if (not found) then
        {
            p := new Treenode;
            (p → lchild) := NULL;
            (p → rchild) := NULL;
            (p → data) := x;
        }
        if (root ≠ NULL) then
        {
            if (x < (q → data)) then
                (q → lchild) := p;
```

```
else
    (q -> rchild) := p;
}
  else
      root := p;
  }
}
```

X—