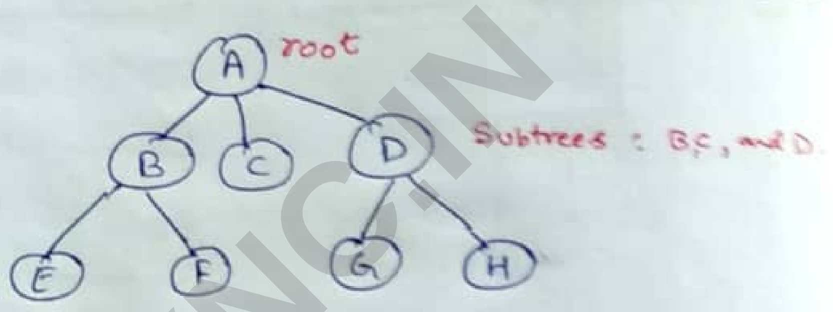
MODULE : 4 TREES

Tree : A tree is a set of finite set of one or more nodes that shows a parent-child relationship such that :

* There is a special node called the root node.

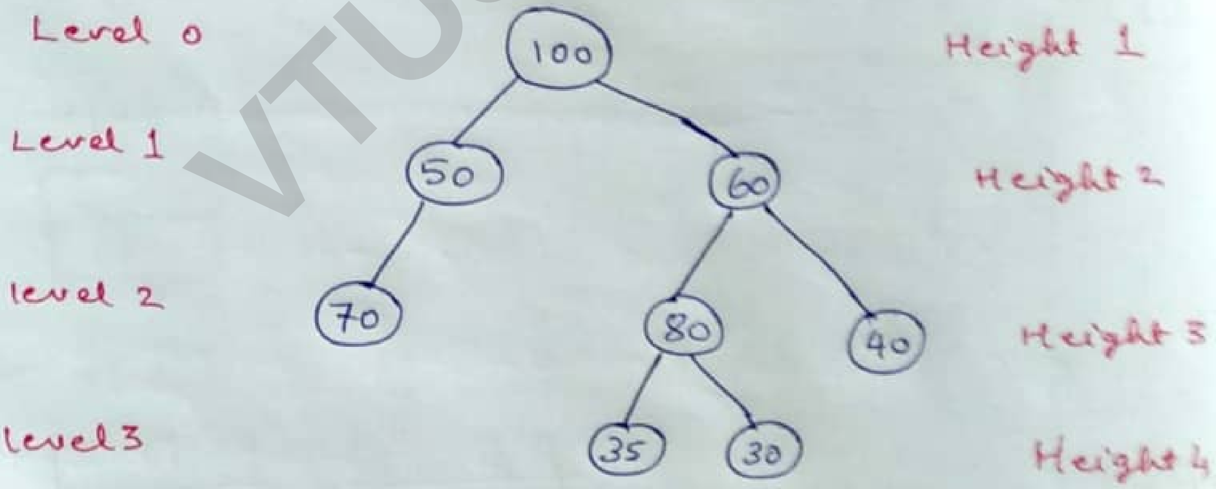
* The remaining nodes are partitioned into disjoint subsets $T_1, T_2, \ldots T_n$; $n \geq 0$ where $T_1, T_2, \ldots T_n$ which are all children of root node are themselves trees called Subtrees.



VTUSYNC.IN

Terminologies :-

Consider the foll. tree :



∴ Max height of the tree = 4

① **Root** : the first node at the top of the tree.
Eg: 100 is the root node in the above tree.

② **Parent** : A node having left subtree or right subtree or both. Eg: 60 is the parent of 80 and 40

Scanned by CamScanner

→ **Child**: node obtained from the Parent node.
Eg: 50 and 60 are Children of 100.

→ **Siblings**: two or more nodes having same Parent.
Eg: 50 and 60 are siblings.

→ **Ancestors**: The nodes obtained in the Path from the specified node x while moving upwards towards the root node are called ancestors.
Eg: 60 is the ancestor of 35, 30, 80 and 40.

→ **Descendants**: The nodes in the path below the Parent are called descendants. ie; the nodes that are all reachable from a node x while moving downwards are all called descendants of x.
Eg: All the nodes below 100 (ie: 50, 60, 70, 80, 40, 35, 30) are descendants of 100.

→ **Left descendants**: The nodes that lie towards left subtree of node x are called left descendants.
Eg: 50 and 70 are left descendants of 100.

→ **Right descendants**: The nodes that lie towards right subtree of node x are called right descendants.
Eg: The right descendants of 100 are 60, 80, 40, 35 and 30.

→ **Left subtree**: All the nodes that are all left descendants of a node x from the left subtree of x.
Eg: The left subtree of 60 are 80, 35 and 30.

→ **Right Subtree**: All the nodes that are all right descendants of a node x from the right subtree of x. Eg: The right subtree of 60 are 40.

→ **Degree**: The number of subtrees of a node is called its degree. Eg: 100 has two subtrees. ∴ degree of node 100 is 2.

VTUSYNC.IN

Scanned by CamScanner

→ Leaf: A node in a tree that has a degree of zero is called a leaf node.

Eg: 70, 35, 30 and 40.

→ Internal nodes: The nodes except leaf nodes in a tree are called Internal nodes.

Eg: 100, 50, 60 and 80.

→ External nodes: The NULL link of any node in a tree is an external node. Eg: rlink of 50, rlink and llink of nodes 70, 35, 30 and 40 are all external nodes.

→ Level: The distance of a node from the root is called level of the node.

Eg: The node 35 and 30 are at a distance of 3 nodes from the root node. So, their levels are 3.

→ Height/Depth: The height of the tree is the maximum level of any leaf in the tree.

Eg: Height of the above tree is 4.

Representation of Trees:-

① List Representation

② Left child- Right Sibling representation

③ Binary Tree / Left child- Right child Representation

→ List representation:-

* The root node comes first

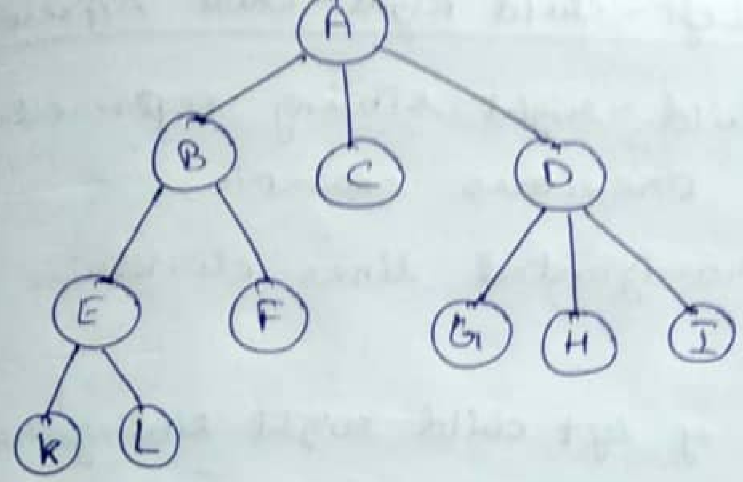
* It is immediately followed by a list of subtrees of that node.

* It is recursively repeated for each subtree.

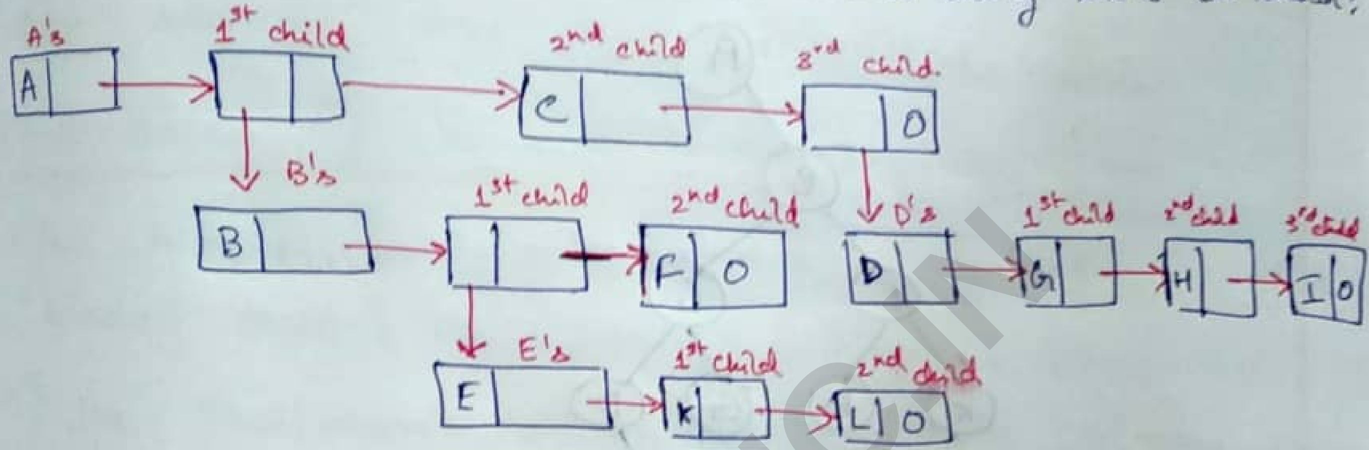
VTUSYNC.IN

Scanned by CamScanner

Eg:



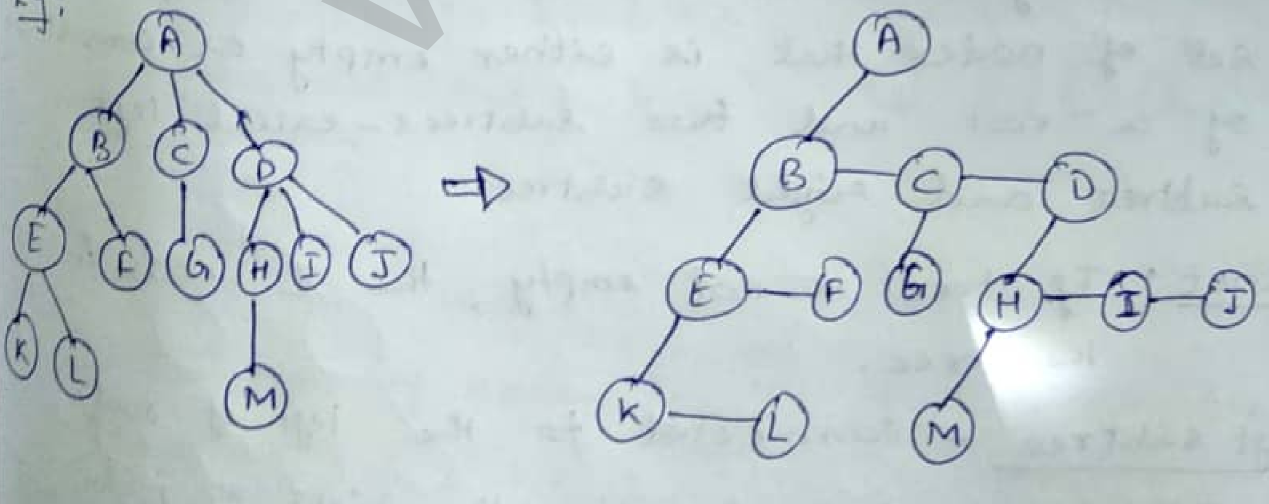
* The above tree can be represented using lists as shown:



→ Left-child Right-Sibling Representation:

* The left child of a node will be represented as it is shown in the tree, whereas the remaining siblings of a node are inserted horizontally to the left child in the representation.

Eg:



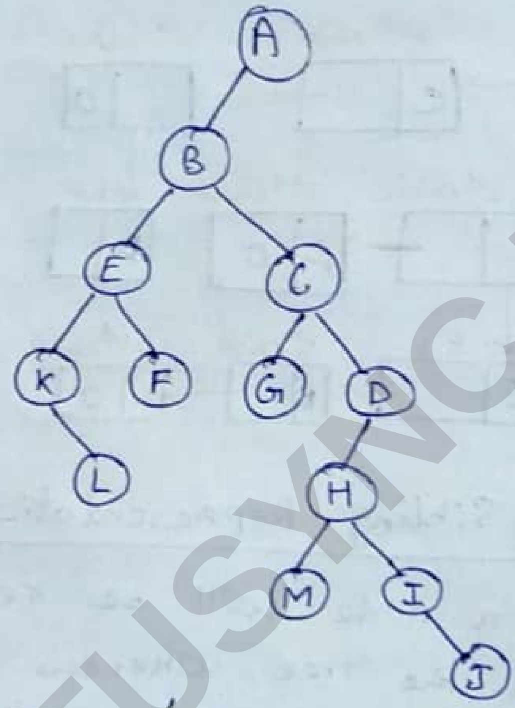
VTUSYNC.IN

Scanned by CamScanner

→ Binary Tree / Left-child Right-child representation

- * ① Obtain left child - right sibling representation as show in previous example.
- * Rotate the horizontal lines clockwise by 45° degrees.

Eg: The ~~abov~~ of left child right sibling representation can be converted into binary tree as shown:



Binary Tree (Degree 2 Trees):-

- * A binary tree is a tree which has finite set of nodes that is either empty or consist of a root and two subtrees - called left subtree and right subtree.

Root: If tree is not empty, the Ist node in the tree.

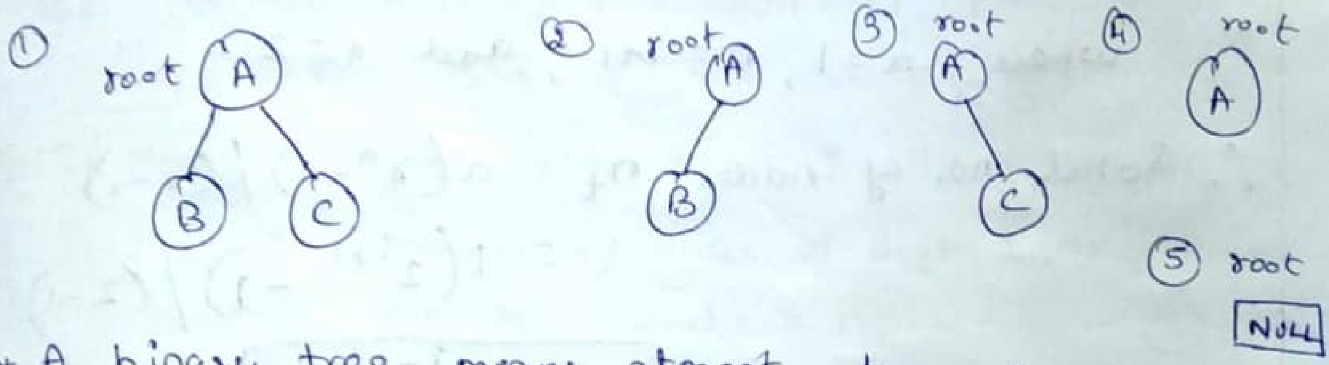
Left subtree - connected to the left of root.

Right Subtree - connected to the right of root.

VTUSYNC.IN

Scanned by CamScanner

Examples of B.T

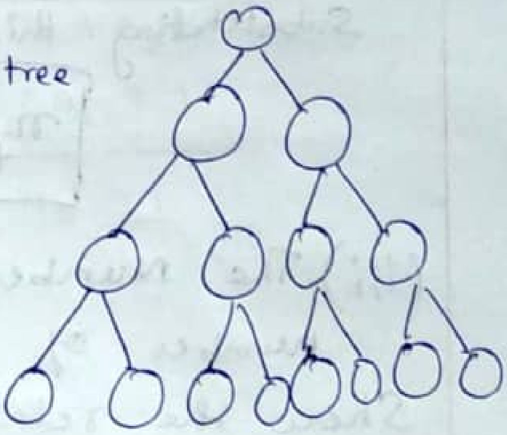


* A binary tree means atmost two; ie; zero, one or two subtrees are possible. But more than two subtrees are not permitted.

Properties Of Binary Tree :-

(i) The maximum no. of nodes on level i of a binary tree = 2^i for $i \geq 0$.

(ii) The maximum no. of nodes in a binary tree of depth $k = 2^k - 1$.



Proof: Consider the foll binary tree and observe the foll factors:

No. of nodes at level 0 = $\underline{1} = 2^0$

No. of nodes at level 1 = $2 = 2^1$

No. of nodes at level 2 = $4 = 2^2$

⋮

No. of nodes at level i = 2^i //

∴ Total No. of nodes in the full binary tree of level $i = 2^0 + 2^1 + 2^2 + \dots\dots 2^i$

The above series is a geometric Progression whose sum is given by:

VTUSYNC.IN

Scanned by CamScanner

$S = a(r^n - 1)/(r-1)$

where $a = 1$, $n = i+1$, and $r = 2$.

$\therefore$ Total no. of nodes; $n_t = a(r^n - 1)/(r-1)$

$= 1(2^{i+1} - 1)/(2-1)$

$$\boxed{n_t = 2^{i+1} - 1}$$

· Substituting $i = 3$ (from above binary tree, level=3)

we get: $n_t = 2^{3+1} - 1$

$= 16 - 1 = 15$ ⇒ which is total no. of nodes in a full binary tree.

⇒ The depth of the tree $k = $ max. level $+ 1$

$$\boxed{k = i + 1}$$

Substituting this value in above eqn, we get:

$$\boxed{n_t = 2^k - 1}$$

(iii) The Number of leaf nodes is equal to number of nodes of degree 2. (OR)

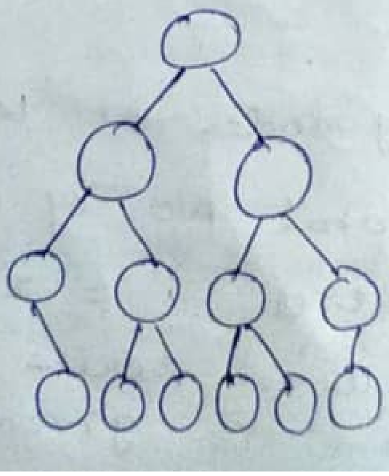
Show the relationship b/w the no. of leaf nodes and nodes of degree 2.

⇒ Let No. of nodes of degree $0 = n_0$

Let No. of nodes of degree $1 = n_1$

Let no. of nodes of degree $2 = n_2$

Let total no. of nodes in the tree is: $n = n_0 + n_1 + n_2$ ---- eqn①



VTUSYNC.IN

Scanned by CamScanner

→ Observe that total no. of nodes is equal to the total no. of branches (B) plus One.

ie; $n = B + 1$ ---- eqn ②

→ If there is nodes with degree 1, No. of branches = 1

∴ For n_1 number of nodes of degree 1, ~~No. of branches~~ No. of branches = n_1 ---- ③

→ If there is node with degree 2, no. of branches = 2

∴ For n_2 number of nodes of degree 2, No. of branches $2n_2$ ... ④

By adding ③ & ④, we get total No. of branches:

$$B = n_1 + 2n_2 \text{ ——— } ⑤$$

Sub. ⑤ in ②, we get: $n = n_1 + 2n_2 + 1$ ——— ⑥

The relation b/w No. of leaf nodes and no. of nodes of degree-2 can be obtained by subtracting: ⑥ from ①, ie:

$$\begin{aligned} n &= n_0 + n_1 + n_2 \\ -n &= n_1 + 2n_2 + 1 \\ \hline 0 &= n_0 - n_2 - 1 \end{aligned}$$

By rearranging $\boxed{n_0 = n_2 + 1}$

Types Of Binary Tree :-

① Strictly Binary Tree (Full B.T)

② Skewed B.T

③ Complete B.T.

④ Expression tree

⑤ Binary search tree

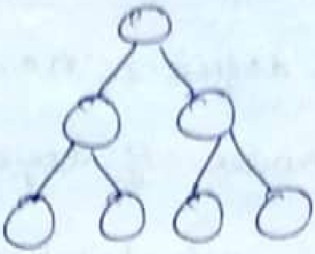
VTUSYNC.IN

Scanned by CamScanner

① Strictly binary Tree / Full Binary Tree :-

* A binary tree having 2^i nodes in any given level i, is called as strictly B.T. Here, every node other than the leaf node has two children.

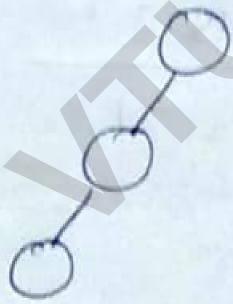
Eg:



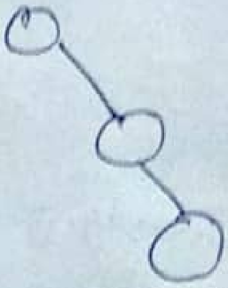
② Skewed Tree :-

* A tree consisting of only left subtree or only right subtree is called skewed tree.

* A tree with only left subtrees is called left skewed tree, and a tree with only right subtree is called right skewed tree.



Left skewed tree



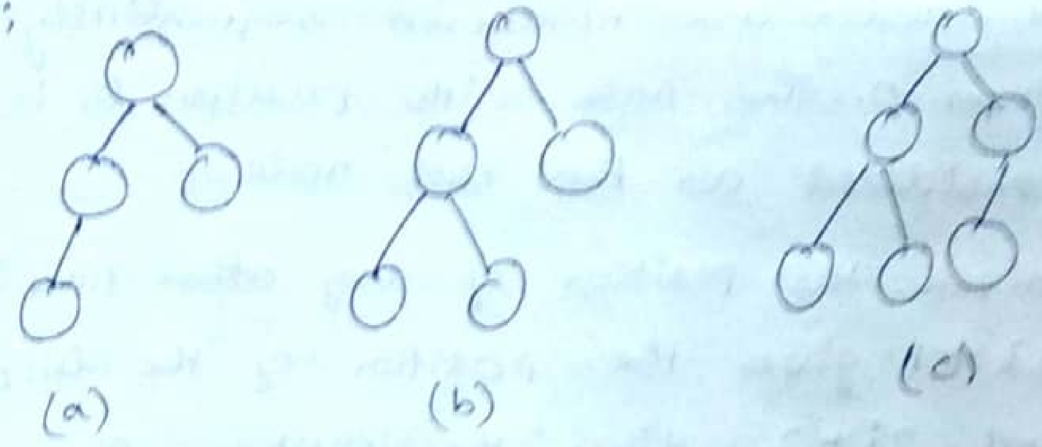
Right skewed tree.

③ Complete Binary Tree : is a binary tree in which every level, except possibly the last level is completely filled. Also all the nodes should be filled only from left to right.(at the leaf level)

VTUSYNC.IN

Scanned by CamScanner

Eg:



(a) (b) (c)

(iv) Expression Tree : // discussed later

(v) Binary search tree : // discussed later.

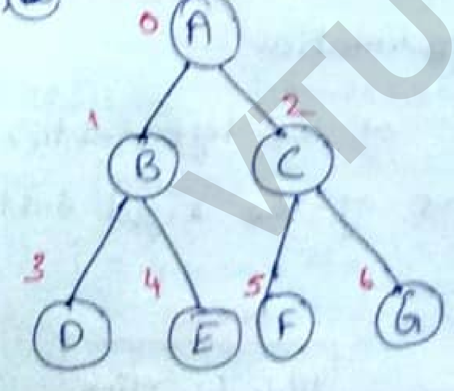
Binary Tree Representation :-

① Array Representation

② Linked Representation.

① Array Representation : A tree can be represented using a sequential array representation

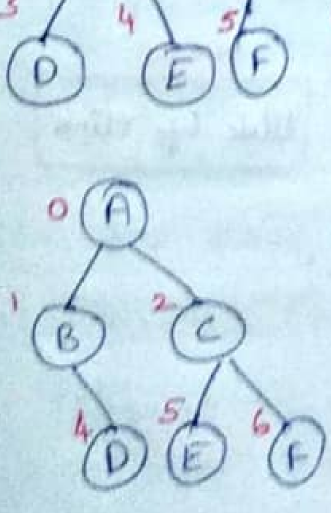
Eg: ⓐ



Array representation.

0	1	2	3	4	5	6
A	B	C	D	E	F	G

b)



0	1	2	3	4	5	6
A	B	C		D	E	F

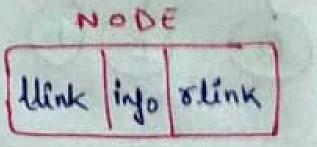
VTUSYNC.IN

- * The nodes are numbered sequentially from 0. The node with position 0 is considered as the root node.
- * Given the position of any other node i, 2i+1 gives the position of the left child and 2i+2 gives the position of the right child.
- * If i is the position of the left child, i+1 gives the position of the right child.
- * If i is the position of the right child, i-1 gives the position of the left child.
- * Given the position of any node i, the Parent position is given by (i-1)/2.

② Linked List Representation :-

- * A node in the tree can have 3 fields:

  info - contains the information
  llink - Contains address of the left subtree.
  rlink - Contains address of the right subtree.



```
struct node
{
   int info;
   struct node *llink, *rlink;
};

typedef struct node* NODE;
```

VTUSYNC.IN

Scanned by CamScanner

* A pointer Variable root can be used to point to the root node always. If the root points to NULL it indicates that the tree is empty. The pointer Variable root can be declared and initialized as :

```
Struct node *root = NULL;
        (or)
    NODE root = NULL;
```

Binary Tree Traversal :-

* Traversal is a method of visiting each node of a tree exactly once in a systematic Order.
* During traversal, one may print the info field of each node visited.
* There are 3 different traversal techniques:

a) preorder b) postorder c) Inorder

a) **PreOrder Traversal** - can be recursively defined as follows :

(i) Process the root Node (N)

(ii) Traverse the left subtree (L)

(iii) Traverse the right subtree (R)

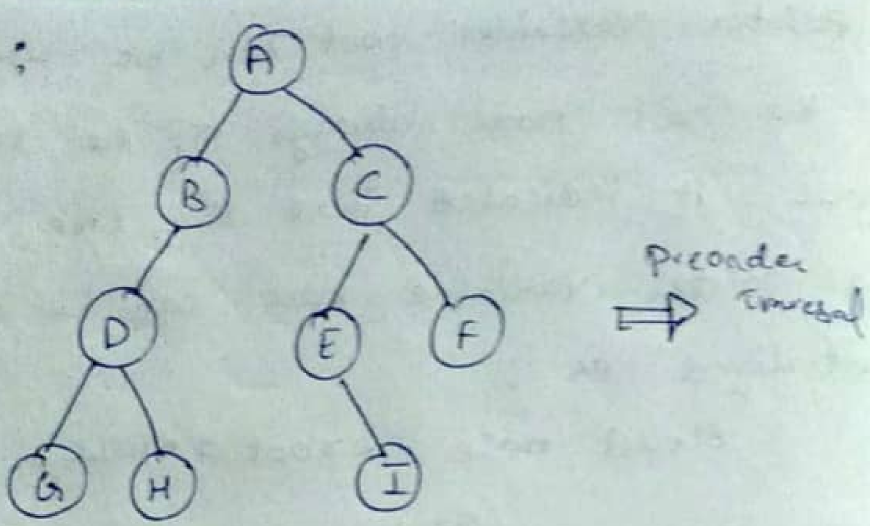
Function for preorder

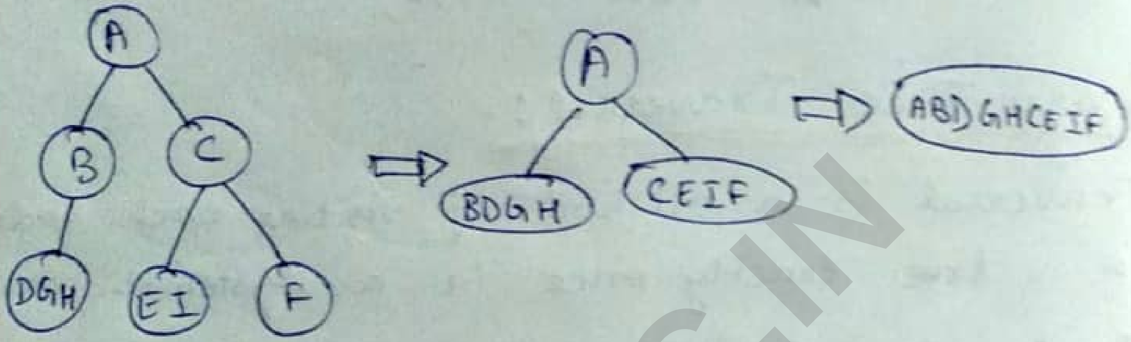
```
Void preorder(NODE root)
{
    if(root == NULL) return;
    printf("%d", root → info); // visit the node
    Preorder (root → llink); // visit left S.T recursively
    Preorder (root → rlink); // visit right S.T recursively
}
```

VTUSYNC.IN

Scanned by CamScanner

Example :





b) Postorder Traversal : can be represented as follow:

(i) Traverse the left subtree (L)

(ii) Traverse the right subtree (R)

(iii) Process the root node (N)

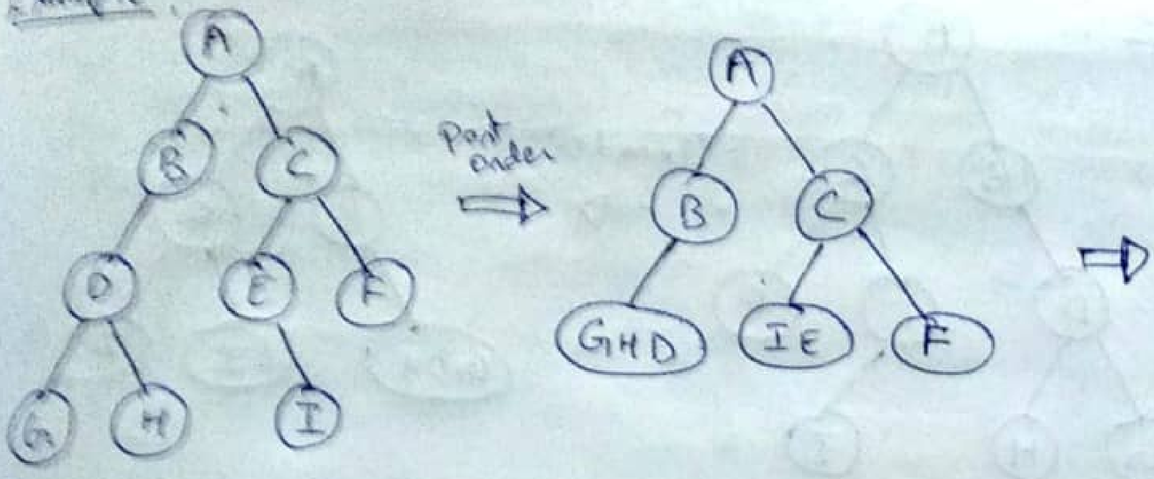
Function : void Postorder(NODE root)

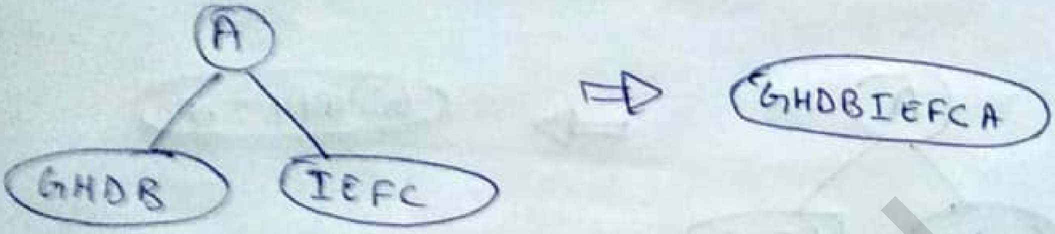
```
{
    if( root ==NULL) return;
    Postorder(root -> llink);
    Postorder( root -> rlink).
    printf("%d", root -> info).
}.
```

VTUSYNC.IN

Scanned by CamScanner

Example:





Answer = GHDBIEFCA

c) Inorder Traversal :- can be represented as:

(i) Traverse the left subtree (L)

(ii) Process the root Node (N)

(iii) Traverse the right subtree (R)

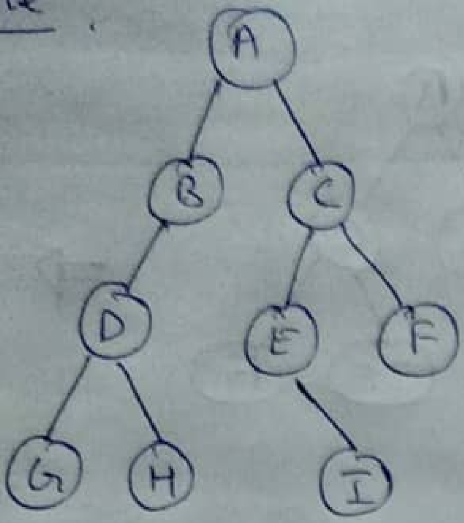
Function

```
void Inorder(NODE root)
{
    if( root == NULL) return;
    Inorder( root → llink);
    printf(" %d", root →info);
    Inorder( root → rlink);
}
```

VTUSYNC.IN

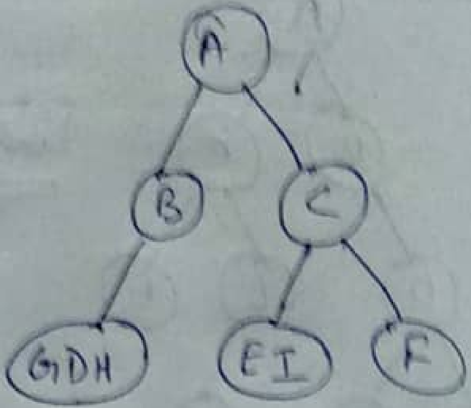
Scanned by CamScanner

Example :

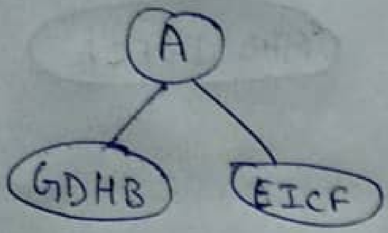


Inorder

⇒



⇒

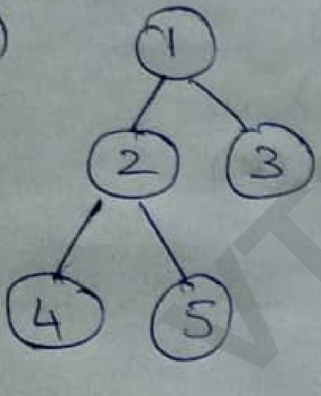


⇒ GDHBAEIcF

Answer = GDHBAEIcF

Solve the following using Binary tree traversal

(i)



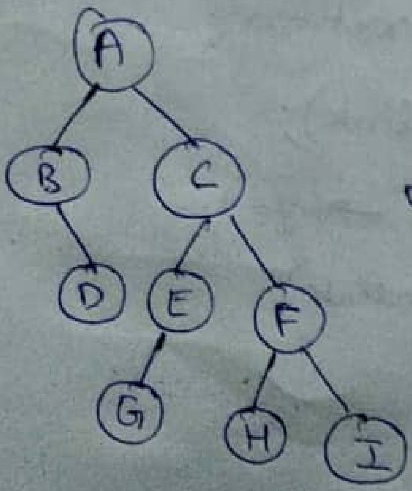
Answer :

⇒ Inorder = 4 2 5·1 3

Preorder = 1 2 4 5 3

Postorder = 4 5 2 3 1

(ii)



Inorder - BDAGECHFI

⇒ Preorder - ABDCEGFHI

PostOrder - DBGEHIFCA

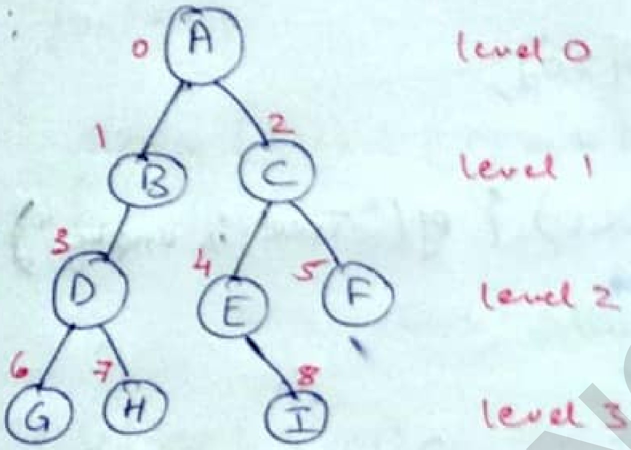
VTUSYNC.IN

Scanned by CamScanner

d) Level Order Traversal :-

- The nodes in a tree are numbered starting with the root on level 0, followed by nodes on level 1, level 2 and so on, moving from left to right.
- visiting the nodes using the ordering suggested by the node numbering is called level order traversing. This kind of traversal requires queues.

Example:



output of level Order: ABCDEFGHI

Function to Print a tree using level order traversal

```
void   levelOrder (NODE root)
{
     NODE  q[SIZE], cur;
      int  f=0, r = -1;
      q[++r] = root;     // Insert root node into queue
      while( f <= r)
     {
        cur = q[f++];         // delete from queue
        pf("%d", cur->info);   // visit the node
        if(cur->llink != NULL)    // insert Left S.T into queue
            q[++r] = cur->llink;
        if(cur->rlink != NULL)    // insert right S.T into queue
            q[++r] = cur->rlink;
     }
     printf("\n");
}
```

VTUSYNC.IN

Scanned by CamScanner

Iterative traversals of Binary Tree :-

① Iterative (Non-recursive) Preorder Traversal :

* W.K.T in Preorder, the root node is visited first, then the left subtree is traversed in preorder and finally right subtree is traversed.

Function: (uses stack)

```
void  preorder(NODE  root)
{
        NODE  cur, S[20];
        int  top = -1;
        if(root == NULL) { pf(" Tree is empty"); return;}
        cur = root;
        for(;;)
      {
         while ( cur != NULL)
         {
              pf(" %d", cur -> info);
              S[++top] = cur;
              cur = cur -> llink;
         }
         if( top != -1)
          {
               cur = S[top--];
               cur = cur -> rlink;
          }
          else
               return;
       }
}
```

VTUSYNC.IN

Scanned by CamScanner

② Iterative Inorder Traversal

Function:

```
  void  Inorder( NODE  root)
  {
        NODE  cur, S[20];
         int  top = -1;
        if(root == NULL){ pf(" Tree Empty"); return; }
        cur = root;
        for(;;)
      {
         while (cur != NULL)
        {      S[++top] = cur;
                 cur = cur -> llink;
         }
        if( top != -1)
        {
             cur= S[top--];
             printf("%d", cur -> info);
              cur = cur -> rlink;
         }
        else
               return;
        }
    }
```

③ Iterative PostOrder Traversal

```
 void  postOrder (NODE  root)
 {
     struct  stack
     {
         NODE  address;
         int  flag;
     };
```

VTUSYNC.IN

Scanned by CamScanner

```
NODE cur;
struct stack s[20]; int top = -1;
if(root == NULL) { - - - - - - - }
cur = root;
for(;;)
{
    while(cur != NULL)
    {
        top++;
        s[top].address = cur;
        s[top].flag = 1;
        cur = cur -> llink;
    }
    while (s[top].flag < 0)
    {
        cur = s[top].address;
        top--;
        pf("%d", cur -> info);
        if(top == -1) return;
    }
    cur = s[top].address;
    cur = cur -> rlink;
    s[top].flag = -1;
}
}
```

Note: Here each node will be stacked twice once during traversing left subtree and another while traversing towards right. To distinguish that traversing the right S.T. is over, we set flag to -1.

* If flag is -ve, then right S.T is traversed and one can visit the node. Otherwise traverse the right subtree. This process is repeated till stack is empty.

VTUSYNC.IN

Scanned by CamScanner

Additional Binary Tree Operations :-

1) Function to copy a tree

```
NODE copy(NODE root)
{
    NODE temp;
    if(root == NULL) return NULL;
    temp= (NODE *) malloc (sizeof(NODE));
    temp→info = root →info;
    temp→lptr = copy(root →lptr);
    temp→rptr = copy(root →rptr);
    return temp;
}
```

2) Function to check whether two trees are equal(same) or not (or Test Equality of Two trees)

```
int equal(NODE r1, NODE r2)
{
    if(r1== NULL && r2 == NULL) return 1; //Trees are same
    if(r1== NULL && r2!= NULL) return 0; //Trees are different
    if(r1!= NULL && r2==NULL) return 0; //Trees are different
    if(r1→ info != r2→info) return 0;
    if(r1→ info == r2 →info) return 1;
    return equal(r1→llink, r2→llink) &&
           equal(r1→rlink, r2→rlink);
}
```

VTUSYNC.IN

Disadvantages of Binary Tree :-

① More than 50% of link fields have NULL (\0) values, hence more memory space is wasted.

② Traversing is time consuming, coz stack is used for both recursive and non-recursive programs. Time is wasted in push & pop activities.

③ Only downward movements are possible.

④ computations of predecessor and successor of given nodes is time consuming.

* All these disadvantages can be overcomed using Threaded Binary Trees.

Threaded Binary Trees :-

Definition: In Threaded Binary Tree, the NULL links can be replaced by pointers called threads, to other nodes in the tree (which facilitate upward movement in the tree)

* Various types of Threaded B.T : In-Threaded B.T, Post-threaded B.T, Pre-threaded B.T.

a.) In-Threaded B.T :-

* In a binary tree, if llink (left link) of any node contains \0 (null) and if it is replaced by address of the inorder predecessor, then the resultant tree is called left-in-threaded binary tree.

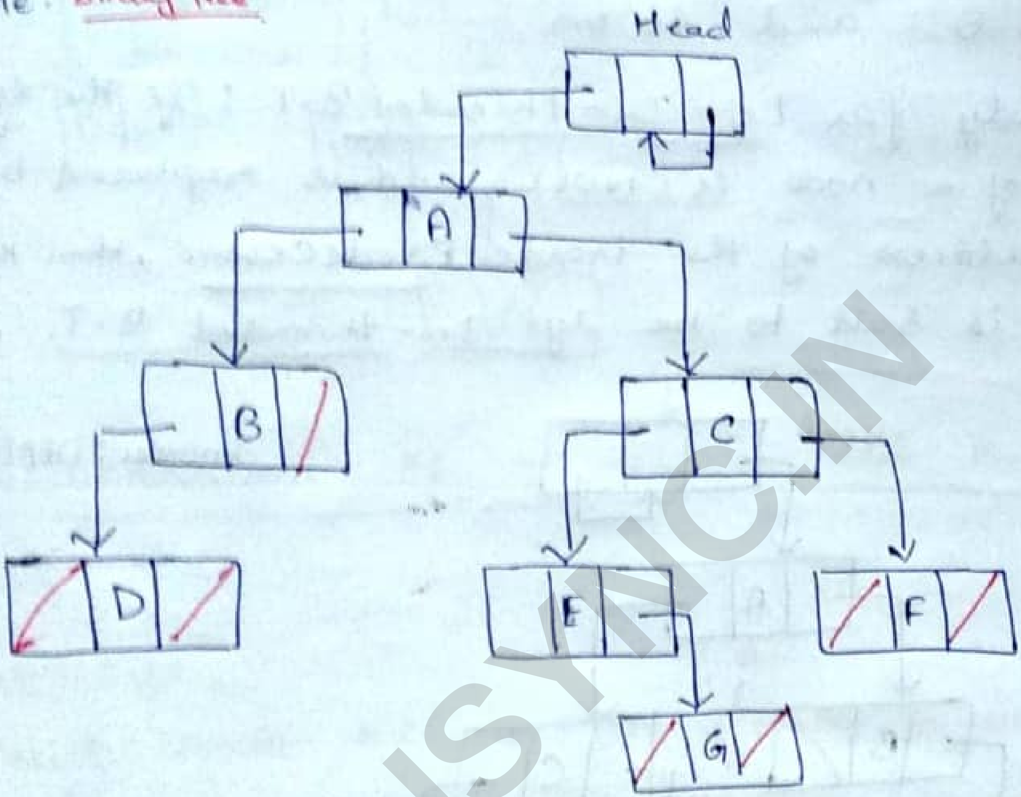
VTUSYNC.IN

Scanned by CamScanner

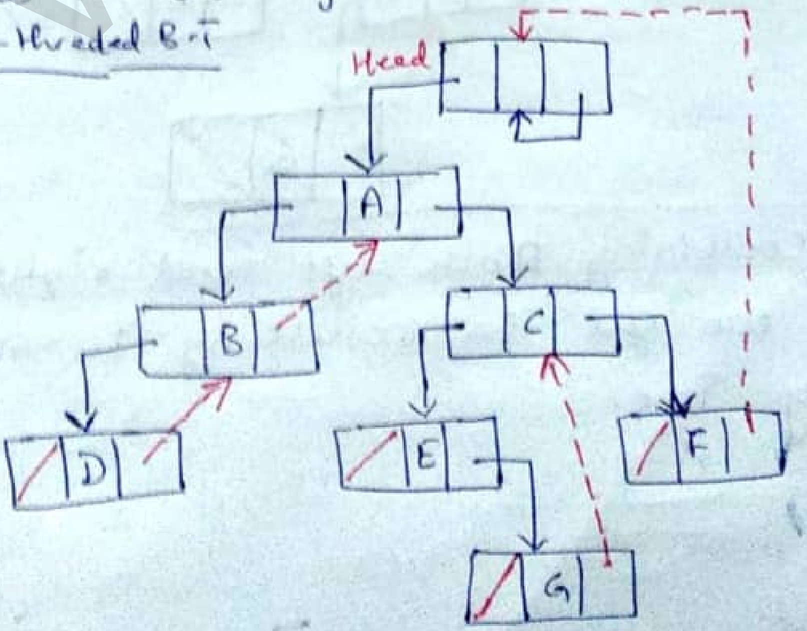
* In a binary tree, if rlink of a node is NULL and if it is replaced by address of inorder succesor, the resultant tree is right In-threaded B.T.

* An In-threaded B.T is one which is both left In-threaded ~~B.T~~ and right in-threaded.

Example: Binary Tree



In the above tree, if the right link of a node is NULL and if it is replaced by the address of the inorder Successor as shown using dotted lines, then the tree is said to be right in-threaded B.T

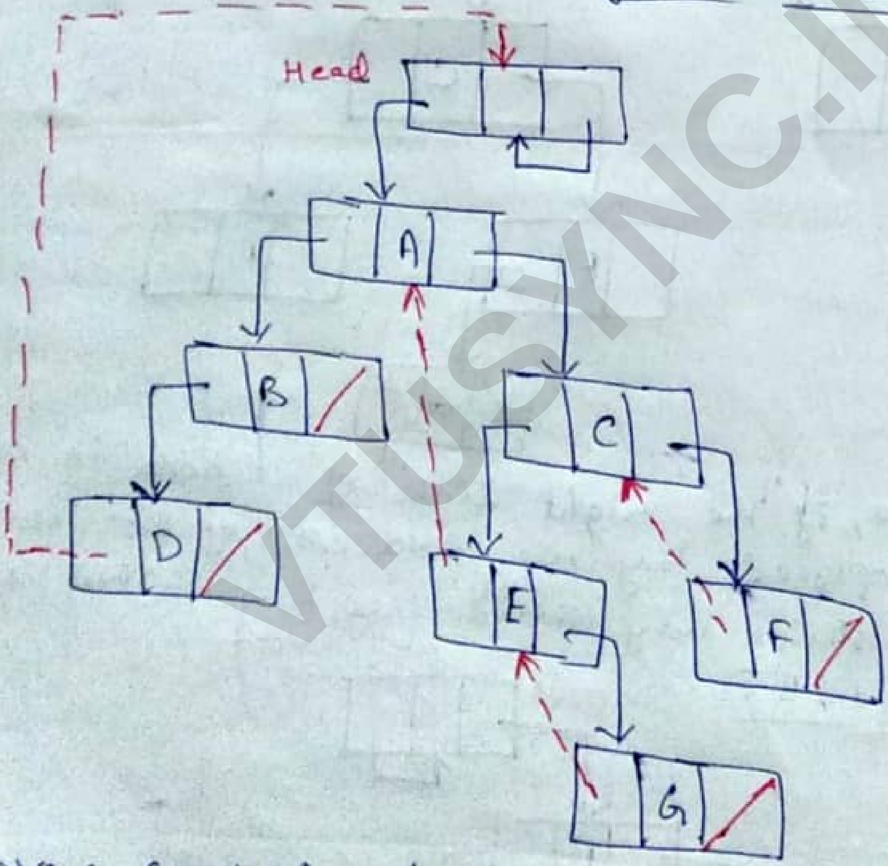


VTUSYNC.IN

Scanned by CamScanner

* That is, the inorder traversal for the above tree is DBAEGCF.
∴ for the nodes whose right link is NULL, check who comes after them (or) which node comes after it. From the above traversal B comes after D, thus there is a link from D to B and so on.

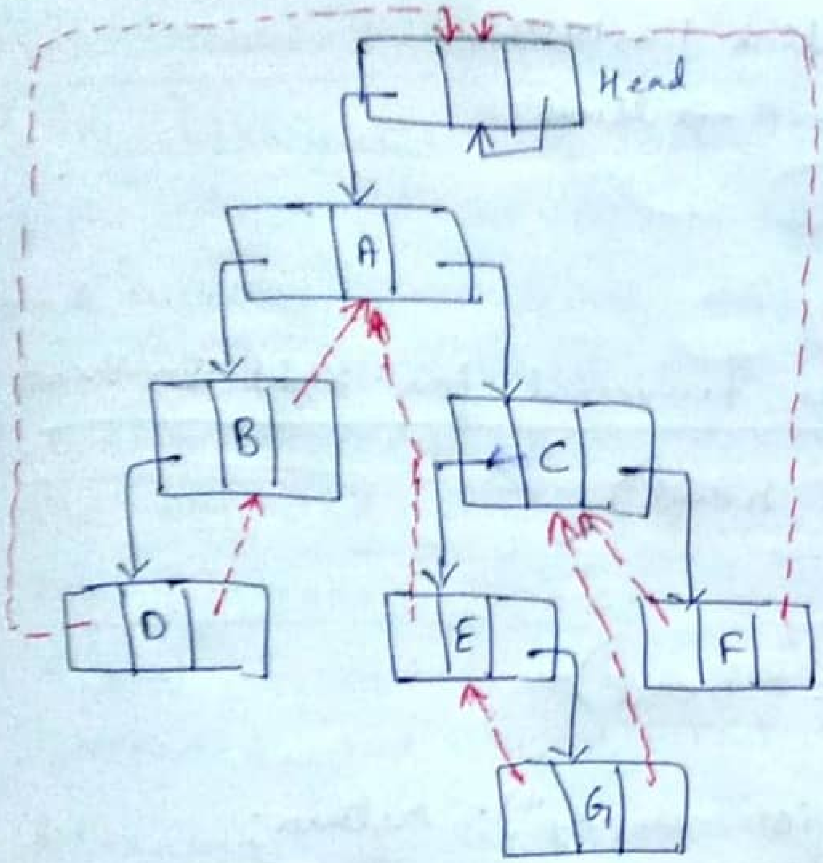
* Similarly for Left In-threaded B.T : If the left link of a node is NULL and is replaced by the address of the inorder Predecessor, then the tree is said to be left In-threaded B.T.



Inorder: DBAEGCF

* Now combining Both left and right In-threaded B.T, we get the resulting In-Threaded binary Tree.

VTUSYNC.IN



Implementation of a In-Threaded B.T

```
struct node
{
   int info;
   struct node *llink;      // pointer to left S.T
   struct node *rlink;      // Pointer to right S.T
   int lthread;             // 1 indicates a thread, a 0
                               indicates Ordinary link
   int rthread;             // 1 indicates a thread else Ordinary link
};
typedef struct node * NODE;
```

Function to find inorder successor for Right In-thread

```
NODE inorder_successor(NODE x)
{
      NODE temp;
      temp = x -> rlink;
      if(x -> rthread ==1) return temp;
```

VTUSYNC.IN

Scanned by CamScanner

```
    while(temp → llink != NULL)
          temp = temp → llink;
    return  temp;
  }
```

Function for Inorder traversal for right In-thread

```
void inorder(NODE head)
{
   NODE  temp;
   if(head → link == head)
    {
        pf("Tree  is  empty"); return;
    }
   pf("The  Inorder  traversal  is \n");
    temp= head;
   for(;;)
   {
      temp=  inorder_successor(temp);
      if(temp==head) return;
      pf("%d", temp → info);
   }
}
```

Function to find inorder predecessor:

/* Same as inorder successor except change rthread with lthread and llink with rlink */

```
   NODE  inorderPredecessor(---)
   {  ________
       ________
       ________
       ________
    }
```

VTUSYNC.IN

Scanned by CamScanner

(b) Pre-threaded B.T's :-

* In a binary tree, if llink and rlink is replaced with the address of its preorder predecessor and successor respectively, then the resulting tree is called Pre-threaded B.T. (OR)

* A pre-threaded B.T consists of both left pre-threaded and right pre-threaded B.T.

(c) Post-Threaded B.T's:-

* A post-threaded B.T consists of both left post-threaded and right post threaded B.T.

Advantages and disadvantages of Threaded B.T's:

Advantages: ① Traversing is faster, and easy to implement.

② computations of predecessor and successor of given nodes is very easy and efficient.

③ wastage of memory is avoided by storing NULL values with address of some other nodes.

④ Can move in upward and also downward directions.

Disadvantages: ① Here extra fields are required to check whether a link is a thread or not. Hence occupy more memory.

② Insertion & deletion of a node consumes more time than other type of trees.

Binary Search Tree (BST):-

* A BST is a binary tree in which for each node x in the tree, elements in the left subtree are less than info(x) and elements in the right subtree are greater than info(x).

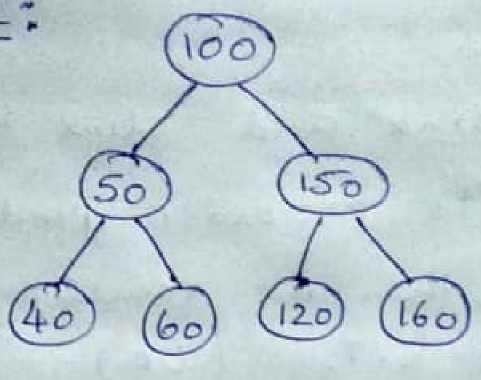
* Every node in the tree should satisfy this condition, if left and right subtree exists.

* A B.ST can be empty.

VTUSYNC.IN

Scanned by CamScanner

Example:



Common Operations of Binary Search Tree (BST):-

① Traversal - Inorder, preorder and postorder.

B.ST traversal → same as in Binary tree, refer lab program - BST.

② Searching - Search for a specific item in tree.

③ Insertion - Insert an item into BST.

④ Deletion - Deleting a node from BST.

Searching a BST for an item:-

* Suppose we wish to search for a node whose key value is K.
* Begin the search from the root node. If root is NULL, the search tree contains no nodes and hence the search is unsuccessful.
* Otherwise, compare K with key value in root. If K equals the root's key, then the search terminates successfully.
* If K is less than root's key, then search the left subtree of the root.
* If K is larger than root's key, then search the right subtree of the root.
* Function to search for an item in BST using Recursion is as follows:

VTUSYNC.IN

```
NODE  search (int item, NODE root)
{   if(root == NULL) return root;  //Item not found
    if(item == root->info) return root;   //Item Found.
    if(item < root->info)
          return search(item, root->llink);
    else
          return search(item, root->rlink);
}
```

Function to search in BST using Iteration

```
NODE  search(int item, NODE root)
{
     NODE  cur;
     if (root == NULL) return NULL;  //empty Tree
     cur = root;
     while (cur != NULL)
     {     if(item == cur->info) return cur;
           if(item < cur->info)
                cur = cur->llink;
           else
                cur = cur->rlink;
     }
     return NULL;     //key Not found
}
```

Inserting an element (node) into a BST :-

* To insert a node K, we must verify that K is different from those of existing keys.

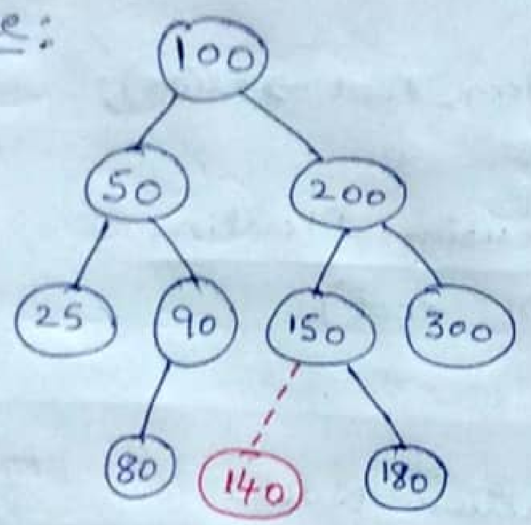
* Firstly, create a node using malloc function and assign a key value (info) to it.

VTUSYNC.IN

Scanned by CamScanner

* If tree is empty initially, then the newly created node becomes the first node in the tree.
* If ~~head~~ root is not empty (NULL), then the tree exists. Hence we have to insert the newly created node at its appropriate position in the tree. (without breaking the property of BST).

Example:



To insert node 140 into the tree, 1st compare with the root and 140. If same then, we can't insert. Otherwise check whether 140 is less than or greater than 100. Repeat this until you reach the node 150. And 140 is inserted as the left child of 150.

Function:

```
NODE insert(int item, NODE root)
{
    NODE temp, cur, prev;
    temp=(NODE *) malloc (sizeof(NODE)).
    temp->info = item;
   temp-> llink= temp-> rlink = NULL;
  if (root == NULL) return temp;
  prev = NULL;
  cur= root;
  while (cur != NULL)
  {
      prev = cur;
      if (item == cur->info)
      {
          pf(" Duplicate Items are not allowed");
              free(temp).
          return root;
      }
```

VTUSYNC.IN

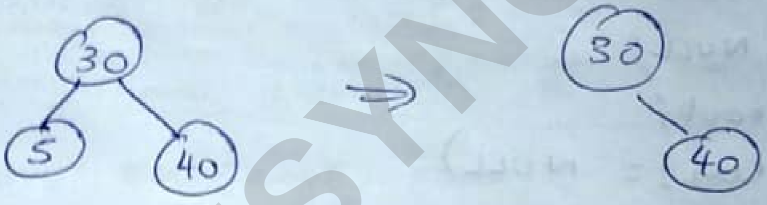
Scanned by CamScanner

```
if (item < cur→info)
    cur = cur→llink;
else
    cur = cur → rlink;
}
 if(item < prev→info)
    prev → llink=temp;
 else
    prev → rlink = temp;
 return root;
}
```

Deletion from a BST :-

* Deletion of a leaf node is simple.

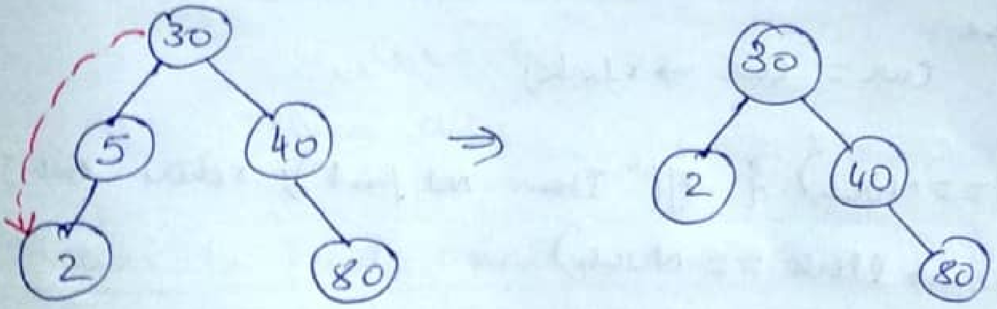
Eg:



Here to delete 5 from the tree, the left child field of its parent(30) should be set to NULL.

* Deletion of a non-leaf node that has only one child is also easy.

Eg:



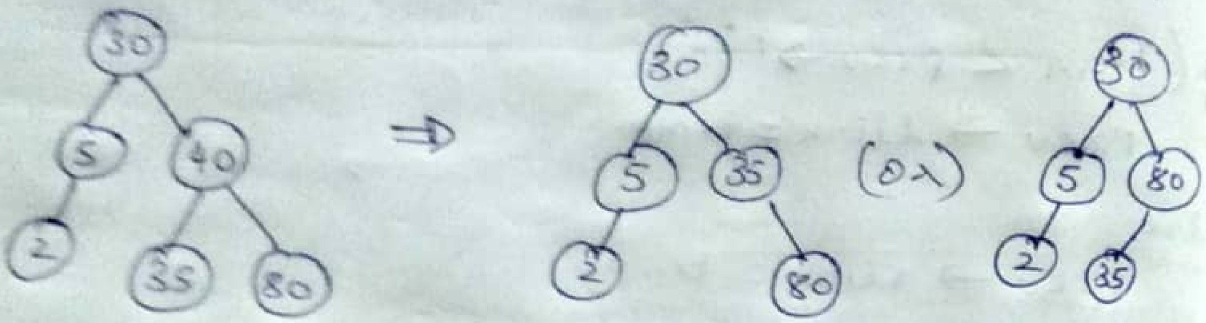
Here to delete 5 from the tree, we simply change the pointer link from the Parent node (30) to the single child node 2.

VTUSYNC.IN

Scanned by CamScanner

* To delete a nonleaf node that has 2 children, the deleted node is replaced by either the largest node in the Right subtree or the smallest node in the left subtree.

Eg: To delete item 40 from the tree below:



Function to delete an item

```
NODE  delete(int item, NODE root)
{
  NODE cur, Parent, suc, q;
  if(root == NULL) { pf("Tree empty"); return root;}
 Parent = NULL;
  cur = root;
  while(cur != NULL)
  {
      if(item == cur->info) break;
       Parent = cur;
      if(item < cur->info)
           cur = cur->llink;
      else
           cur = cur->rlink;
  }
 if(cur == NULL) { pf("Item not found"); return root;}
  if(cur->llink == NULL)
        q = cur->rlink;
 else if(cur->rlink == NULL)
        q = cur->llink;
```

VTUSYNC.IN

Scanned by CamScanner

```
else
{
    suc= cur → rlink;
     while (suc → llink != NULL)
          suc = suc → llink;

    suc → llink = cur → llink;
       q = cur → rlink;
  }
  if (cur == parent → llink)
       Parent → llink = q;
  else
      Parent → rlink = q;
   free(cur);
  return root;
}
```

Other Operations on BST :-

1.) To find maximum value in a BST :

Function :

```
        NODE max(NODE root)
       {
           NODE cur;
           if (root == NULL) return root;
            cur= root;
           while( cur → rlink != NULL)
                 Cur = Cur → rlink;
           return cur;
         }
```

2.) Function to find minimum value in a BST :

```
   NODE min(NODE root)
   {
       NODE cur;
       if (root == NULL) return root;
```

VTUSYNC.IN

Scanned by CamScanner

```
    cur = root;
    while (cur → llink != NULL)
         cur = cur → llink;
    return cur;
}
```

3.) To find the height of the tree ie; the Maximum level

Recursive function definition;

$$\text{Height}(root) = \begin{cases} -1 \\ 1+\max(\text{height}(root \rightarrow llink), \\ \quad \text{height}(root \rightarrow rlink)) \end{cases}$$

➩
```
   int height (NODE root)
   {
       if(root == NULL) return -1;
       else
       return (1 + max(height (root → llink), height(root → rlink)));
   }
  int max(int a, int b)
  {
     return (a>b) ? a : b;
  }
```

4.) To count the nodes in a tree: Here we can use any of the traversal technique and increment the counter whenever a node is visited.

➩
```
 void count (NODE root)
 {
     if( root == NULL) return;
     count(root → llink);
     count ++;
     count ( root → rlink);
 }
```

VTUSYNC.IN

Scanned by CamScanner

5.) To count the leaf nodes (ie; terminal nodes) in a BST

```
void countleaf(NODE root)
{
    if(root == NULL) return ;
    countleaf(root → llink);
    if(root → llink == NULL && root → rlink == NULL)
     count++;
     countleaf(root → rlink);
  }
```

Application of Trees : Evaluation of Expressions.

Expression Trees :-

* A sequence of operators and Operands that reduces to a single value is called as an expression.

Expression Tree: is a binary tree that satisfy the following properties:

(i) Any leaf is an Operand

(ii) The root and internal nodes are operators.

(iii) The subtrees represent sub expressions with root of the subtree as an operator.

⇨ How an Infix Expression can be expressed using Expression Tree ?

* An Infix expression consisting of operators and operands can be represented using a binary tree with root as the operator.

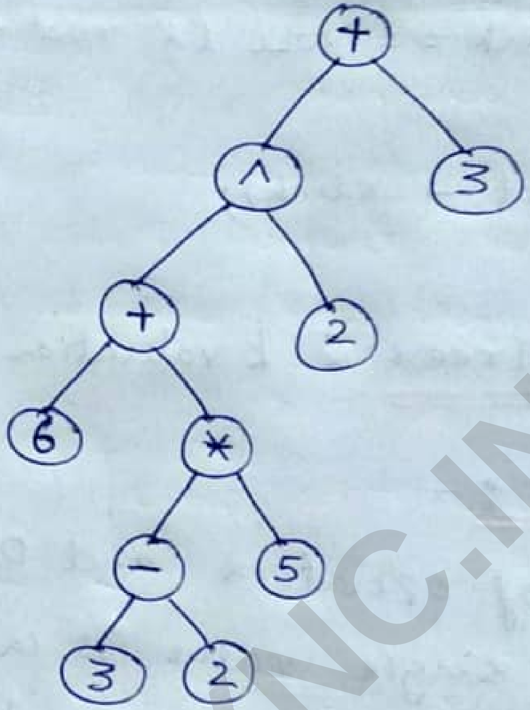
* The left and right subtrees are the left and right operands of that operator.

VTUSYNC.IN

Scanned by CamScanner

* A node containing an operator is not a leaf whereas a node containing an operand is a leaf.

<u>Example</u> : For the Infix expression $((6+(3-2)*5)^{\wedge}2+3)$ can be represented using expression tree as follows:



* when we perform the foll traversal operations for the above tree, we get :

i) Preorder traversal ➩ + ^ + 6 * - 3 2 5 2 3

ii) Postorder traversal ➩ 6 3 2 - 5 * + 2 ^ 3 +

* Observe that if the expression tree is traversed in preorder, we get prefix expression, and if we traverse in postorder, we get postfix expressions Also if we traverse in inorder, we get Infix expression without Parenthesis.

VTUSYNC.IN

Scanned by CamScanner

→ How to create a binary tree for the postfix Expression (or) Procedure to obtain an expression tree from the postfix Expression?

Step 1: Scan the symbol from left to right.

Step 2: Create a node for the scanned symbol.

Step 3: If the symbol is an operand push the corresponding node onto the stack.

Step 4: If the symbol is an operator, pop one node from the stack and attach to the right of the corresponding node with an operator. The first popped node represents the right operand. pop one more node from the stack and attach to the left. Push the node corresponding to the operator on to the stack.

Step 5: Repeat step 1 for each symbol in the postfix expression. Finally when scanning of all the symbols from the postfix expression is over, address of the root node of the expression tree is on top of the stack.

C Function to create an expression tree for the postfix Expr:

```
NODE   createExp(char postfix[])
{
  NODE   temp, s[20]; int i, k;
  char  symbol;
  for(i=k=0; (symbol = postfix[i])!='\0'; i++)
  {
     temp=(NODE *) malloc (sizeof (NODE));
       temp→info = symbol;
      temp→llink = temp→rlink = NULL;
      if(isalnum(symbol))   s[k++]=temp;
      else
      {   temp→rlink = s[--k];
          temp→llink= s[--k];
          s[k++]= temp;
      }
```

VTUSYNC.IN

Scanned by CamScanner

```
 }
  return s[--k];
}
```

⇒ * Evaluation of Expression:-

* In Expression tree, whenever an operator is encountered, evaluate the expression in the left subtree and evaluate the expression in the right subtree and perform the operation.
* The recursive definition is as:

$$
\text{Eval(root)} = \begin{cases} \text{Eval(root} \rightarrow \text{llink) Op Eval(root} \rightarrow \text{rlink)} & \text{when root} \rightarrow \text{info is operator} \\ \text{root} \rightarrow \text{info} - \text{'0'} & \text{when root} \rightarrow \text{info is an operand} \end{cases}
$$

Function:

```
float Eval(NODE root)
{
    float num;
    switch(root->info)
    {
        Case '+': return Eval(root->llink) + Eval(root->rlink);
        Case '-': return Eval(root->llink) - Eval(root->rlink);
        case '*': return Eval(root->llink) * Eval(root->rlink);
        case '/': return Eval(root->llink) / Eval(root->rlink);
        Case '$' :
```

VTUSYNC.IN

Scanned by CamScanner

```
case '^': return pow(Eval(root → llink), Eval(root → rlink));

default: if(isalpha(root → info))
         {
             pf("%c", root → info);
             sf("%f", & num);
             return num;
         }
         else
                 return root → info = '0';

  }
}
```

* Write a C program to evaluate an expression using Expression Tree:

```
#include <stdio.h>
#define STACK_SIZE 20
#include <stdlib.h>
#include <math.h>
struct node
{
   char info;
   struct node *llink, *rlink;
};
typedef struct node* NODE;
void display(NODE root, int level)   // To display Tree
{
      int i;
      if(root == NULL) return;
      display(root → rlink, level + 1);
      for(i = 0; i < level; i++) printf(" ");
      printf("%c\n", root → info);
```

VTUSYNC.IN

Scanned by CamScanner

```
   display( root → llink, level + 1);
 }
NODE  createExp( char  postfix[]) //Fn to create a tree
 {
        ________        // Refer Creation of Expression Tree
          ________
           ________
   }
float Eval(NODE root)
 {
      ________
       ________          // Refer Evaluation of Expression
        ________
    }
main()
{
    char postfix[40]; float res;
    NODE  root = NULL;
    pf(" Enter postfix expression \n");
    sf(" %s ", postfix);
    root = CreateExp(postfix);
    pf(" The Expression Tree is :\n");
    display (root, 1);
    res = Eval(root).
     pf(" Result of postfix Expn = %f ", res);
 }
```

<u>Output</u>: Enter the postfix expression: abc-d*+e^f+

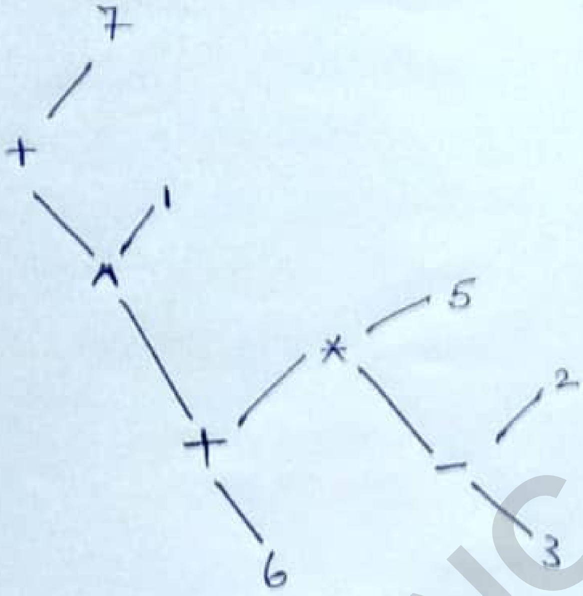
VTUSYNC.IN

Scanned by CamScanner

e=1, a=6, b=3, C=2, d=5, f=7

Result = 18.000000

The expression tree is:



VTUSYNC.IN