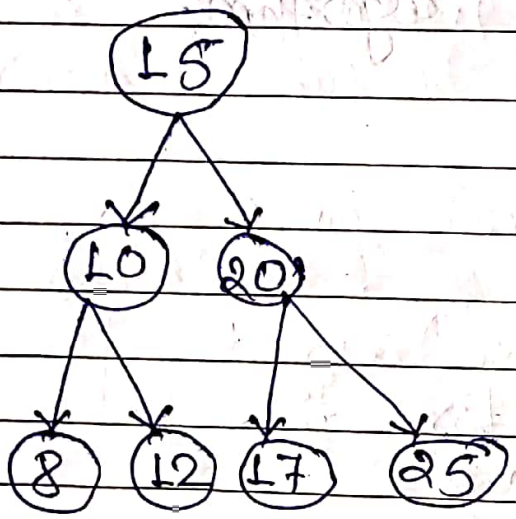
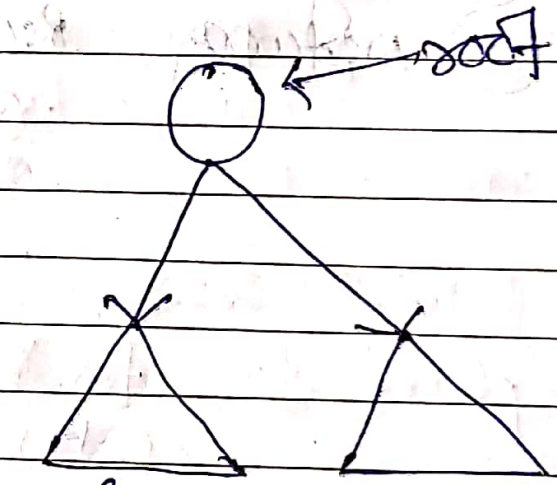
Binary Search Tree

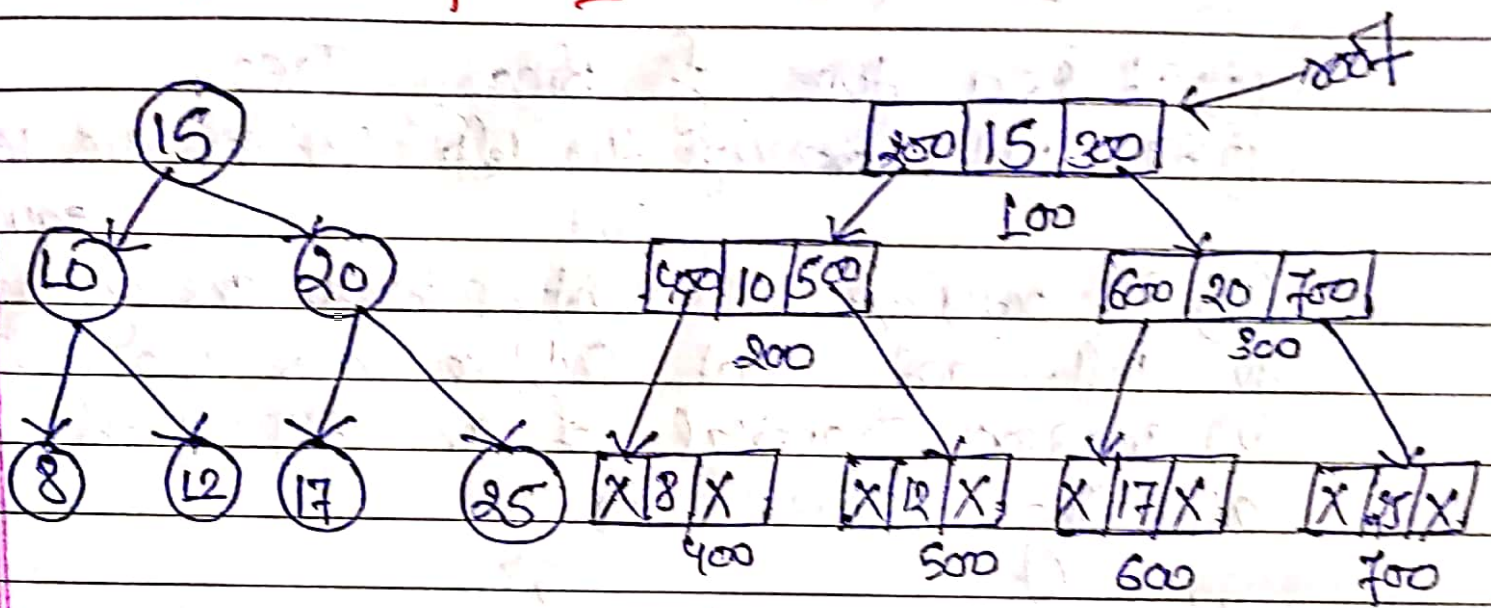
A binary tree in which for each node, value of all the nodes in left subtree is lesser or equal value of all the nodes in right subtree is greater.






```
node *getnewnode(int data){
    node *temp = new node();
    temp->data = data;
    temp->right = temp->left = NULL;
    return temp;
}
```

BST Implementation




```
node *insert(node *root, int data){
    if (root==NULL){ // Empty tree
    return getnewnode(data);
    }
    else if (data <= root->data){
    root->left = insert(root->left, data);
    }
    else{
    root->right = Insert(root->right, data);
    }
    return root;
}
```

properties of BST

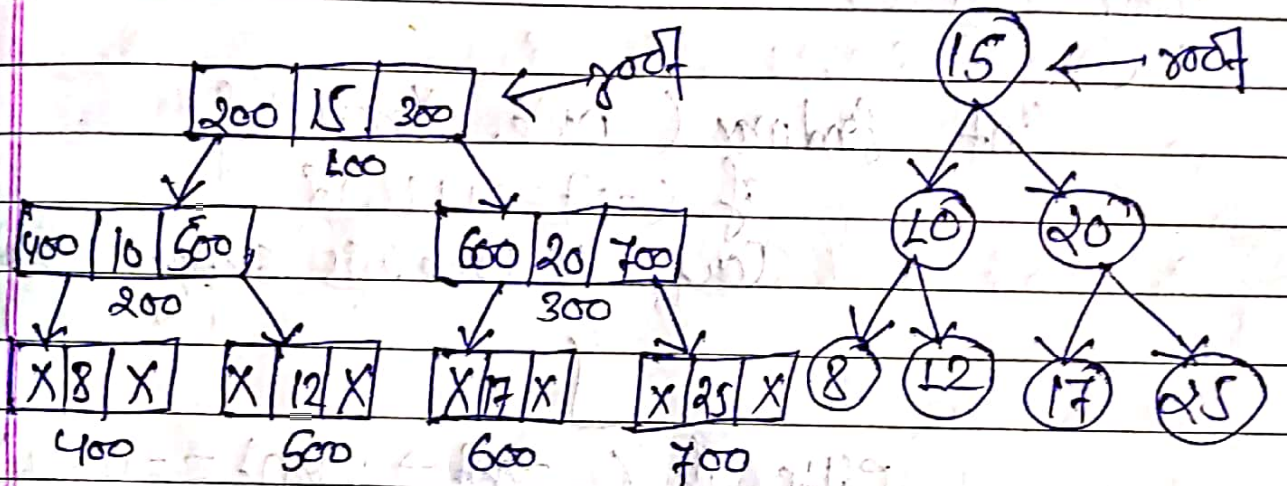
(i) It is a type of Binary Tree.
(ii) All nodes of the left subtree are lesser or equal.
(iii) All nodes of the right subtree are greater.
(iv) left and right subtree are also BST.
(v) Inorder Traversal of a BST gives an ascending sorted order.

```
//Searching an element.

bool search (node *root, int data){
    if (root == NULL){
        return 0;
    }
    else if (root->data == data){
        return 1;
    }
    else if (data <= root->data)
        return search (root->left, data);

    else{
        return search (root->right, data);
    }
}
```

// find min. and max. element in a BST



using iterative.

```
int findmin (node *root)
  if (root==NULL){
    cout<<" Tree is Empty "<<endl;
  }
  else{
    while (root->left != NULL){
      root = root->left;
    }
    return root->data;
  }
```

using recursion

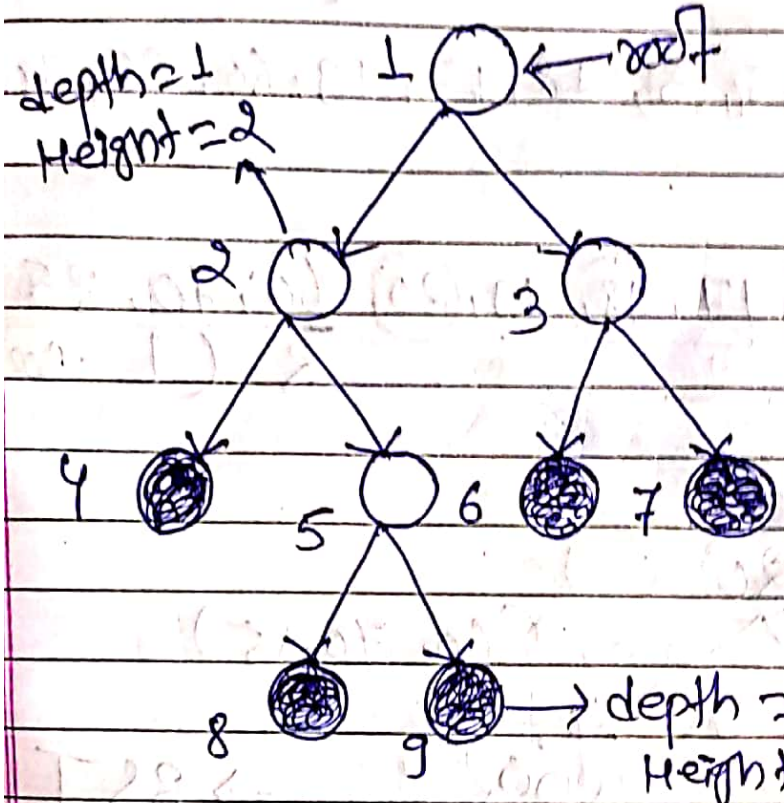
```
int findmax ( node *root ){
        if (root==NULL){
        cout<<"Tree is Empty"<<endl;
        }
    else if ( root->right==NULL){
        return root->data;
        }
    return findmax ( root->right);
    }
```

// find height of a binary tree.

There are two convention to define Height of BST.

(i) no. of nodes on longest path from root to the deepest node.

(ii) no. of edges on longest path from root to the deepest node.




```
int findHeight(node *root){
    if(root==NULL){
        return -1;
    }
    return max(findHeight(root->left), findHeight
                (root->right))+1;
}
```

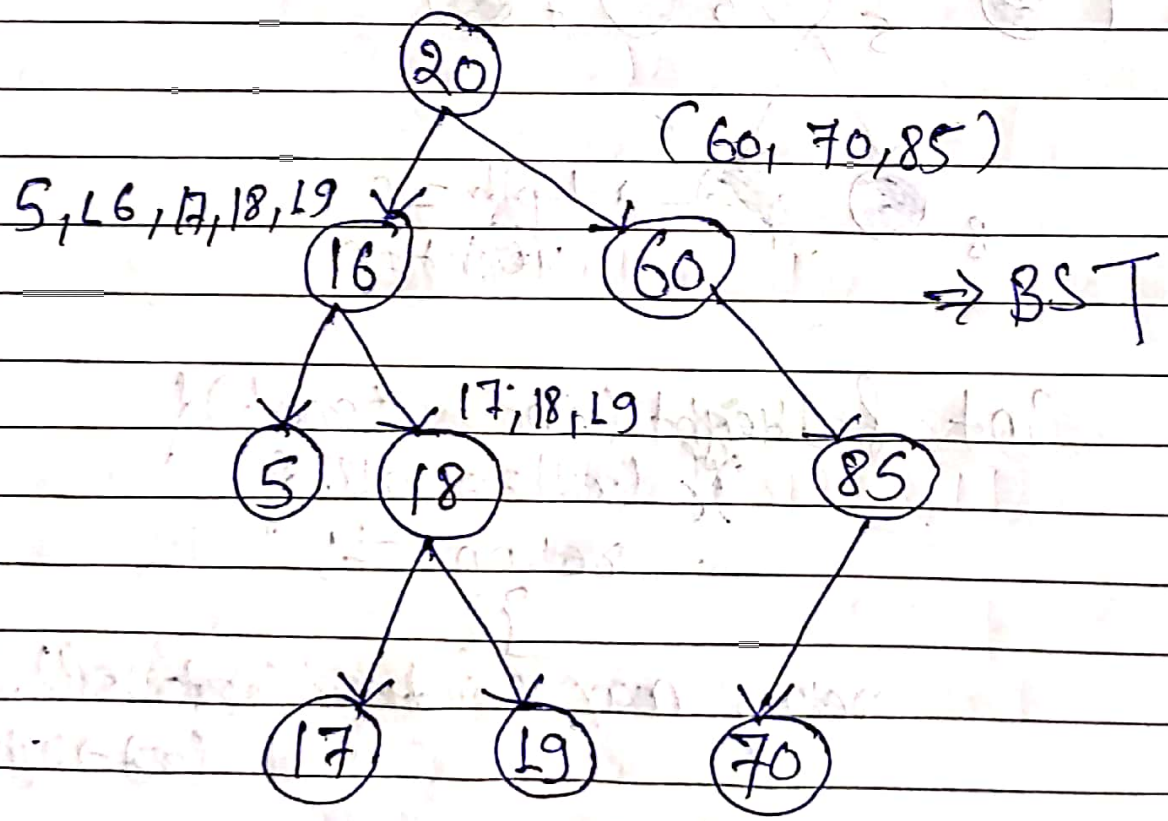
// Construction of BST when only preorder is given.

Preorder :- (20) 16, 5, 18, 17, 19, 60, 85, 70 (Root-L-R)

Soln :-

→ Inorder : 5, (16), 17, (18), 19, (20), (60), 70, 85

L (L - Root - R)



// Construct a BST from given postorder traversal.

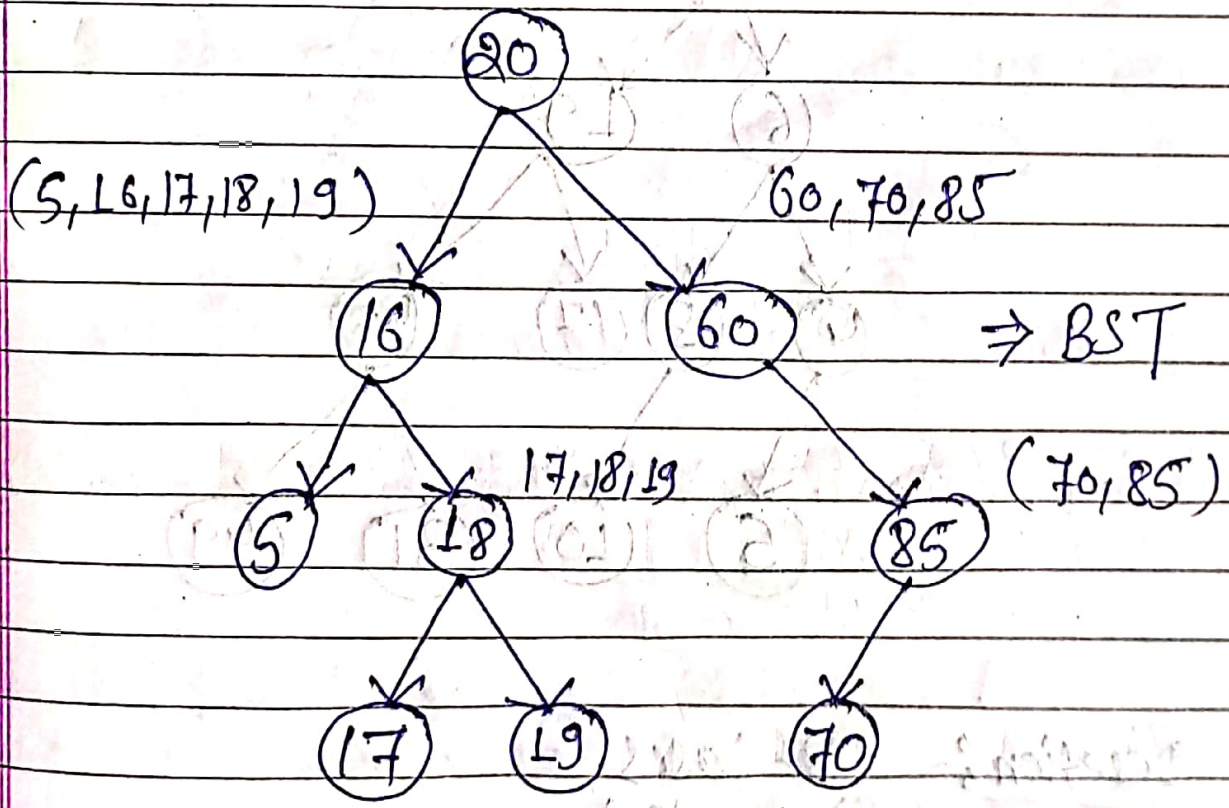
postorder :- 5, 17, 19, 18, 16, 70, 85, 60, 20 (L-R-Root)

soln :-

→

Inorder :- 5, 16, 17, 18, 19, 20, 60, 70, 85 (L-Root-R)

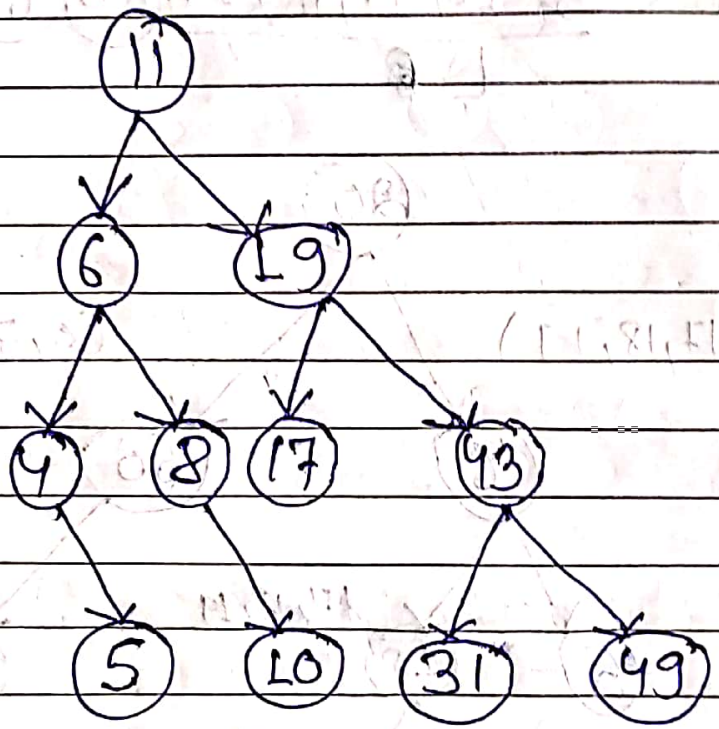
L | R



// Draw Binary Search tree By inserting the following numbers from left to right.
11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31.

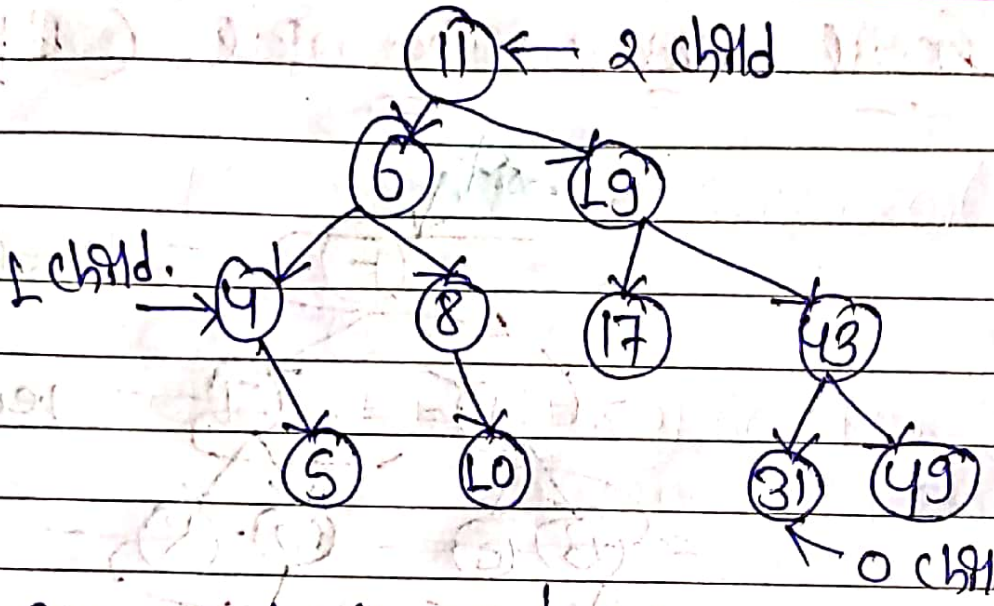
Sol^n :-

→

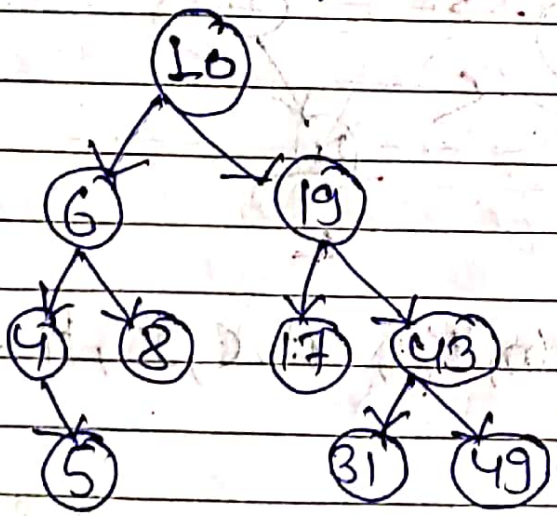


Deletion :- 3-cases

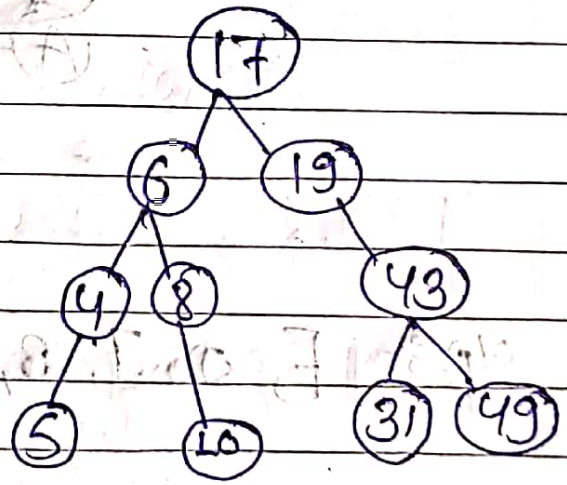
(i) 0 child
(ii) 1 child
(iii) 2 child
↳ (i) Inorder predecessor
→ largest element from left subtree
↳ (ii) Inorder successor
→ smallest element from right subtree



Inorder predecessor



Inorder successor

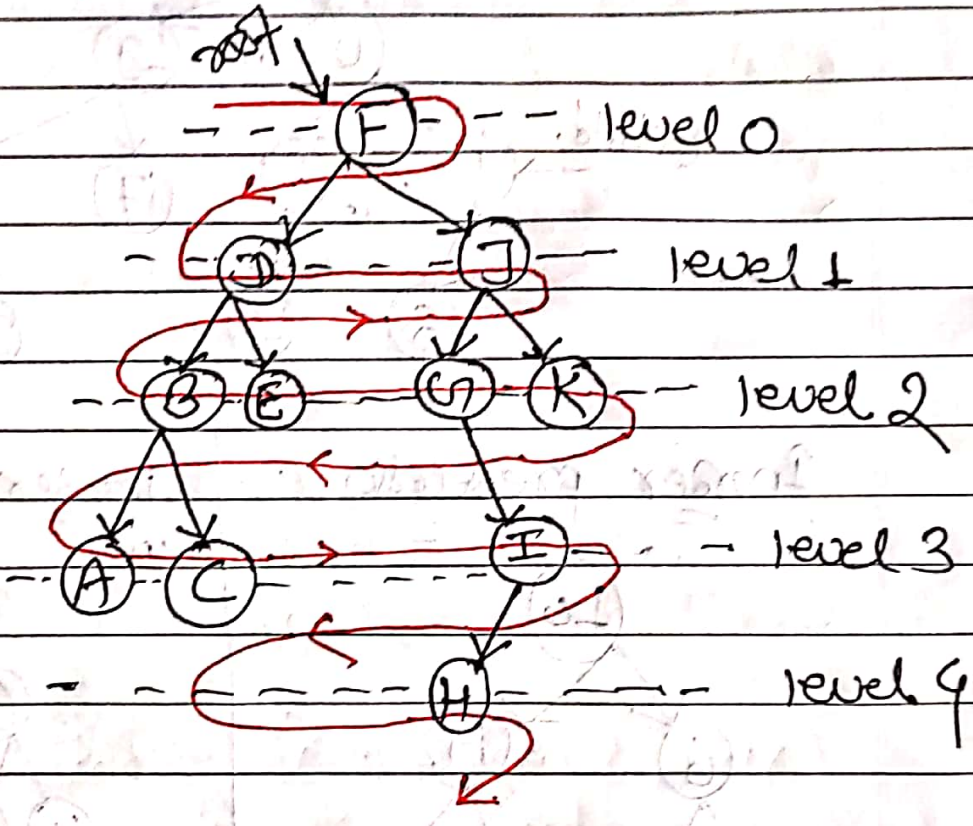


4, 5, 6, 8, 10, 11, 17, 19, 31, 43, 49

Inorder predecessor

Inorder successor.

level order Traversal (BFS)

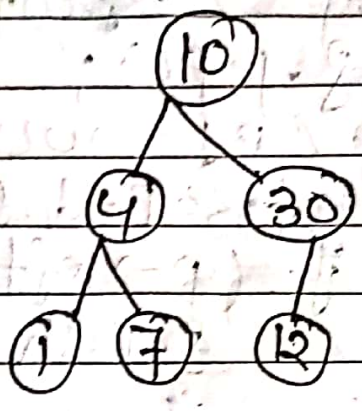


O/P :- F, D, J, B, E, G, K, A, C, I, H.

```
void levelorder(node *root){
    if(root==NULL){
        return;
    }
    queue<node*> q;
    q.push(root);
    q.push(NULL);
```

```
while (!q.empty()){
    node *p = q.front();
    q.pop();
    if (p!=NULL){
        cout << p->data << " ";
        if (p->left!=NULL){
            q.push(p->left);
        }
        if (p->right!=NULL){
            q.push(p->right);
        }
    }
    else if (!q.empty()){
        q.push(NULL);
    }
}
}
```

check for BST



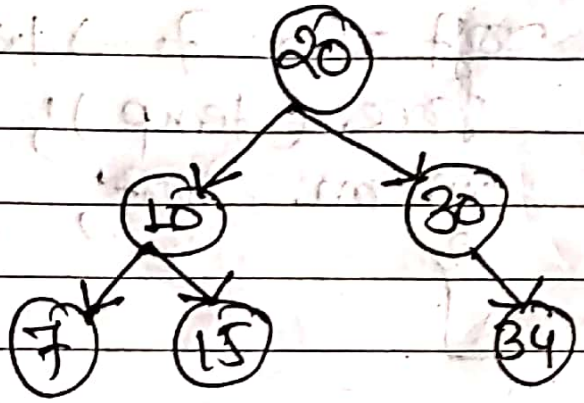
Strategy = min.allowed < node
max.allowed > node.

node (min, max)

left (min, node.) right (node., max)

```
bool Is_BST (node *root, int MIN, int MAX){
    if (root == NULL)
        return 1;
    }
    else if (root->data <= MIN || root->data >= MAX){
        return 0;
    }
    return Is_BST(root->left, MIN, root->data)
        && Is_BST(root->right, root->data, MAX);
}
```

Delete a node from BST



Case 1 :- No child.

i.e. leaf node.

```
if (root->left==NULL && root->right==NULL){
    free(root);
    root=NULL;
    return root;
}
```

Case 2 :- one child

i.e. 30.

```
else if (root->left==NULL){
    node *temp=root;
    root=root->right;
    free(temp);
    return root;
}
```

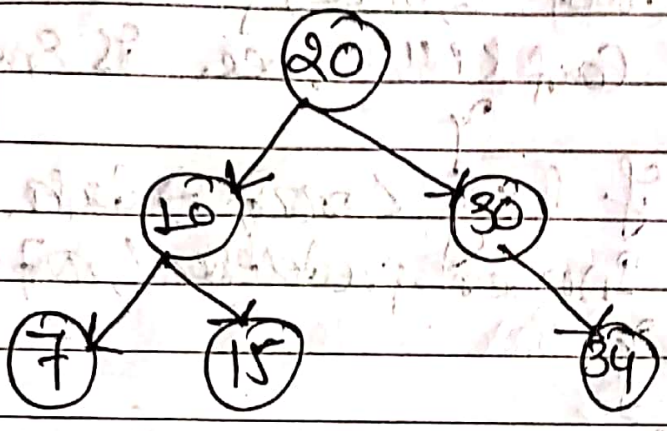
```
else if (root->right==NULL){
    node *temp = root;
    root = root->left;
    free(temp);
    return root;
}
```

// Case 3 :- 2 children

```
else {
node *temp = Inordered_predecessor(root);
    root->data = temp->data;
root->left = Delete(root->left, temp->data);
}
```

```
node * Inordered_predecessor(node *root){
    root = root->left;
    while(root->right != NULL){
        root = root->right;
    }
    return root;
}
```

Inorder predecessor



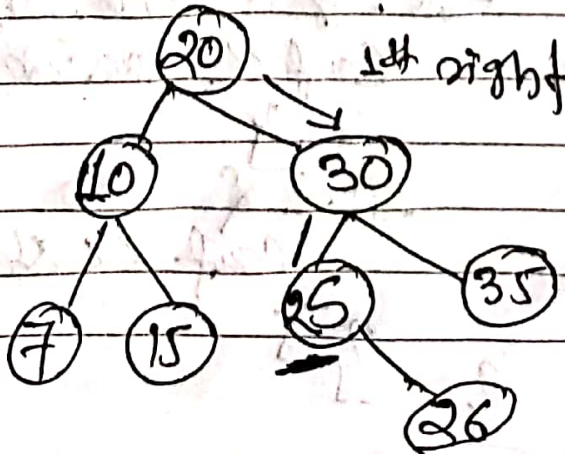
Case 1 :- If Tree node having a left child.

(i) root → left

(ii) find out the right most node.

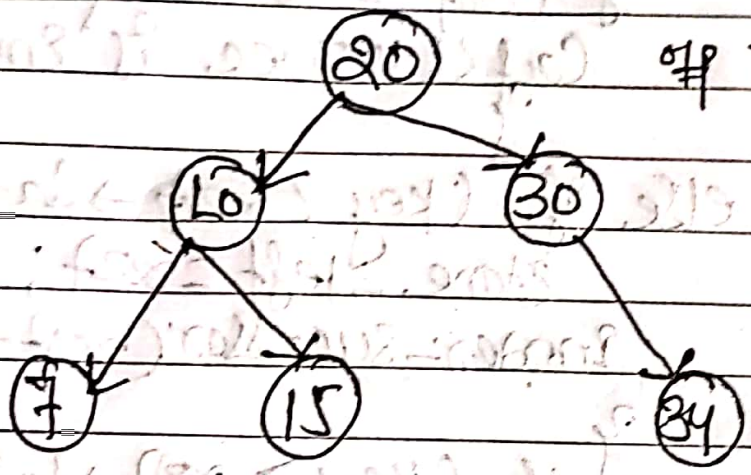
Case 2 :- If tree don't have a left child.

If the node 'key' don't have a left subtree, search that node from the root and the node from where we take the last right is the answer.



```
void Inorder-predecessor (node *root, int key){
    if (root==NULL){
        Cout<<"Tree is Empty"<<endl;
    }
    else if (key < root->data){
        Inorder-predecessor (root->left, key);
    }
    else if (key > root->data){
        store-right = root; //global variable
        Inorder-predecessor (root->right, key);
    }
    else { // left subtree & present.
        if (root->left != NULL){
            root = root->left;
            while (root->right != NULL){
                root = root->right;
            }
            Cout<< root->data;
        }
        else if (store-right == NULL){
            Cout<<"predecessor doesn't exist"<<endl;
        }
        else{
            Cout<< store-right->data;
        }
    }
}
```

Inorder Successor



7, 10, 15, 20, 30, 34

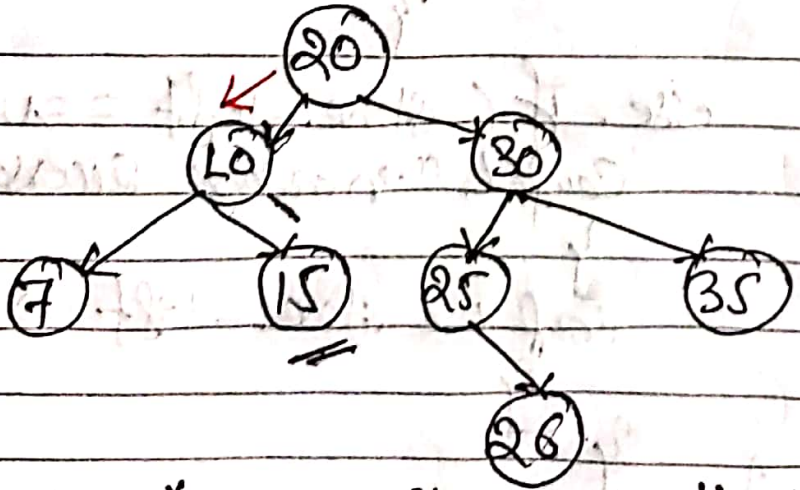
Case 1 :- If node have right child.

(i) root → right

(ii) find out the left most node.

Case 2 :- If node don't have right child.

If the node (key) don't have a right sub tree, search that node from the root and the node from where we take the last left is the answer.



node * store_left = NULL; // global variable

```
void Inorder_successor(node *root, int key){
    if (root == NULL){
        cout << "Tree is Empty" << endl;
    }
    else if (key < root->data){
        store_left = root;
        Inorder_successor(root->left, key);
    }
    else if (key > root->data){
        Inorder_successor(root->right, key);
    }
    else{
        if (root->right != NULL){
            root = root->right;
            while (root->left != NULL){
                root = root->left;
            }
            cout << root->data;
        }
        else if (store_left == NULL){
            cout << "Inorder successor doesn't exist";
        }
        cout << store_left->data;
    }
}
```