Imp. ⊛ ⊛

Lecture 73

Binary Search Trees

Date
Page

⊛ **What is a Binary Search Tree**

(Binary Tree)

Left me sare chote / Right me sare bade

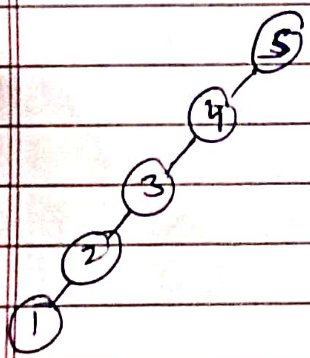
→ **Advantages**

1) Insertion, Deletion, Search $\Big\} O(\log_2 n)$

2) Sorted Order
easier to find min. & max. element.

→ **Disadvantages**

Worst case Complexity



Search(1)
$O(n)$ → Worst case
Unbalanced tree
→ skewed in one direction.

→ **Applications**

→ Sets → Sorted order
→ Maps
→ Priority queues

→ **Traversal**

① Preorder	② Inorder	③ Postorder
Print root → value	Traverse root → left	Traverse root → left
Traverse root → left	~~Traverse~~ Print root → value	Traverse root → right
Traverse root → right	Traverse root → right	Print root → value

#Code

```
# include <iostream>
using namespace std;

→ Class Node {
public:
        int Value;
        Node* left;
        Node* right;
        Node (int v) {
              value = v;
              left = Right = NULL;
        }
};
⇒ Class BST {
public:
      Node * root;
      BST () {
           root = NULL;
      }
};
⇒ Void InsertBST (int Val, Node *&root) {
        Node* newNode = new Node(Val);
        if (root == NULL) {
              root = newNode;
              return;
        }
        Node *Current = root;
        while (true) {
              if (current -> value > Val) {
                    if (Current -> left == NULL) {
                          current -> left = newNode;
```

Date
Page

```
            current = current -> left;
        } else {
            if ((current -> right == NULL) {
                    current -> right = newNode;
                    return;
            }
            current = current -> right;
        }
    }
}
                                   One More funct for traversal
int main () {
        BST bst1;
//For Recursive =>  ~~Insert BST (bst1.root, 3)~~ bst1.root = InsertBST(
                                                         bst1.root, 5)
        InsertBST (3, bst1.root);
        InsertBST (7, bst1.root);
        InsertBST (4, bst1.root);
        ~~return 0;~~  InorderTraversal (bst1.root);
&       Cout << Searching (bst1.root, 4) << endl;
        return 0;
}
```

② Recursive func. for inserting a Node.

```
InsertBST (Node* root, int Val) {
//base case
    if (root == NULL) {
        Node* newNode = new Node(Val);
        return new Node;
    }
    if (root -> value > val) {                          // Recursive
        root -> left = InsertBST (root -> left, val);        Code
    } else if (root -> value < val) {
        root -> right = InsertBST (root -> right, val);
    }
    return root;
}
```

```
{ InorderTraversal( Node* root) {

      InorderTraversal (root -> left);
      Cout << root -> value << "  ";
      InorderTraversal (root -> right);
  }
```

Date

Page

Searching :-

```
bool Searching (Node* root, key) {
// base case
    if (root == NULL) {
        return false;
    }
    if (root -> value == key) {
        return true;
    }
// recursive case
    if (root -> value > key) {
        return Searching (root -> left, key);
    }
    if (root -> value < key) {
        return Searching (root -> right, key);
    }
}
```

Deletion O(log n)

```
Node* deleteBST(Node* root, int key) {

    if (root == NULL) {
        return root;
    }
    if (root -> value > key) {
        root -> left = deleteBST(root -> left, key);
    } else if (root -> value < key) {
        root -> right = deleteBST (root -> right, key);
    } else { // root is the node to be deleted.
//child has 0 Node;
        if (root -> left == NULL && root -> right == NULL) {
            free (root);
            return NULL;
        }
```

Date
Page

```
//node has 1 child node.
    else if(root->left==NULL){
        Node* temp = root->right;
        free(root);
        return temp;
    }
    else if(root->right==NULL){
        Node* temp = root->left;
        free(root);
        return temp;
    }
    // node has two child nodes
    else{
        Node* justSmallerNode = largestNodeBST(root->left);
        root->value = justSmallerNode->value;
        root->left = deleteBST(root->left, justSmallerNode
                                              ->Value);
    }
  }.
}.return root;
}
//-> Deletion se pehle
    Node* largestNodeBST(Node* root){
        Node* curr = root;
        while(curr && curr->right != NULL){
            curr = curr->right;
        }
        return curr;
    }

int main(){

    bst1.root = deleteBST(bst1.root, 4);
    InorderTraversal(bst1.root);
    cout<<endl;
    return 0;
}
```

$S.C = T.C = O(n)$

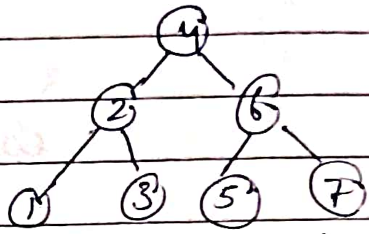
(height diff → -1,0,1)

Q ☆ Sorted Array to Balanced BST & Print the preorder traversal of the BST Created

Input:

n = 7, Elements = 1, 2, 3, 4, 5, 6, 7

Output → Preorder :→ 4 2 1 3 6 5 7

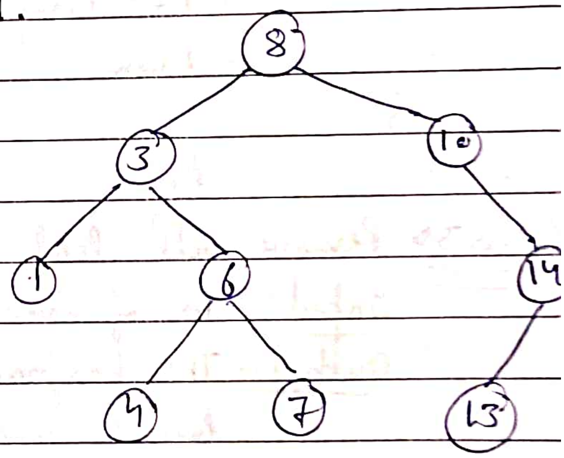


Sol →

```
Node* SortedArraytoBST(Vector<int> V, int start, int end){
    // base case
    if (start > end){
        return NULL;
    }
    int mid = (start + end)/2;

    Node* root = new Node(V[mid]);
    // Recursive Case
//left Sub tree     root → left = SortedArraytoBST(V, start, mid-1);
// Right Sub tree   root → right = SortedArraytoBST(V, mid+1, end);

    return root;
}
~~Node~~ Void preorderTraversal (Node* Root) {
    if (root == NULL) return 0;
    Cout << root → value << " ";
    preorderTraversal (root → left);
    preorderTraversal (root → right);

}

int main () {
    int n;
    Cin >> n;
    Vector<int> V(n);
```

```
for(int i=0; i<n; i++){
        cin>> V[i];
}
BST bst;
    bst.root = SortedArraytoBST(V, 0, n-1);

preorderTraversal(bst.root);

return 0;
}
```

Ques 2 ⟹ Given a BST and two values. You need to find the LCA i.e. Lowest Common Ancestor of the two nodes provided both the nodes exist in the BST.



Input:

n = 9
values = [8, 3, 1, 6, 4, 7, 10, 14, 13]
node-1 = 3
node-2 = 13

Output ⟹ LCA = 8

Sol ⟹

```
Node * ~~Common~~ LowestCommonAncestor(Node* root, Node* Node1, Node* Node2)
  {
     if (root == NULL) return NULL;

     if (root->value > Node1->value && root->value > Node2->value){
          //LCA will lie in left subtree
          return LowestCommonAncestor(root->left, Node1, Node2);
     }
     if (root-> value < Node1->value && root->value < Node2->value){
          //LCA will lie in right subtree
          return LowestCommonAncestor(root->right, Node1, Node2)
     }
```

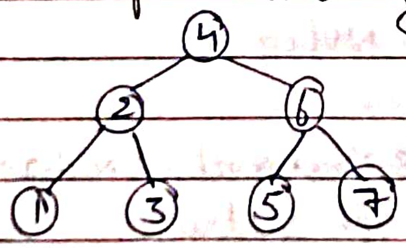
```
        // if root value lies b/w Node1 and Node2
        // or if root value is equal to any of node values.
        return root;
}
int main () {
        BST bst1;
        bst1.root = InsertBST (bst1.root, 3)
                        :
                        :
        Node *node1 = new Node(2);
        Node * node2 = new Node (6);

        Node * temp = LowestCommonAncestor(bst1.root, node1, node2);
        cout << temp -> value << endl;
        return 0;
}
```

Ques 3> Remove all leaf nodes from BST

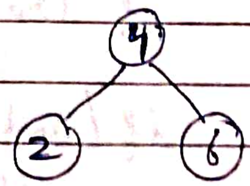
Input:- no. of nodes to the BST, followed by the nodes value.

Output:- The program outputs the preorder traversal of BST before and after removing the leaf nodes.

BST before removing leaf Nodes



BST after Removing leaf Nodes



Sol->

```
Node * removeLeafNodes (Node* root) {
        //base case.
        if (root == NULL) {
                return NULL;
        }
```

```
if (root -> left == NULL && root -> right == NULL) {
    return NULL;
}
//recursive case
root -> left = removeLeafNodes (root -> left);
root -> right = removeLeafNodes (root -> right);
return root;
}
Void preOrderTraversal (Node* root) {

    Cout << root -> value << " ";
    preOrderTraversal (root -> left);
    preOrderTraversal (root -> right);
}
int main () {

    bst1.root = removeLeafNodes (bst1.root);
    preOrderTraversal (bst1.root);
    cout << endl;
    return 0;
}
```

Ques4 -> Inorder Predecessor or successor for a given tree in BST.

Sol ->

```
Void InOrderPreSuccBST(Node* root, Node* &pre, Node* &Succ, int key) {
    if (root == NULL){
        return;
    }
    if (root -> value == key) {
        //pre-> right most node in left subtree
        if (root -> left != NULL) {
            Node* temp = root -> left
            while (temp -> right != NULL){ temp = temp -> right;}
            pre = temp;
        }
```

$T.C = S.C. = O(h)$


```
        //succ -> leftmost node in right subtree
        if (root->right != NULL){
            Node* temp = root->right;
            while (temp->left != NULL){
                temp = temp->left;
            }
            succ = temp;
        }
        return;
    }
    if (root->value > key){
        succ = root;
        InorderpreSuccBST(root->left, pre, succ, key);
    }
    else if(root->value < key){
        pre = root;
        InorderpreSuccBST(root->right, pre, succ, key)
    }
}
int main () {
    BST bst1;
    :
    :

    Node* pre = NULL;
    Node* succ = NULL;
    InOrderpreSuccBST( bst1.root, pre, succ, 4);
    if (pre != NULL){
        Cout << "pre - " << pre->value << endl;
    }else {
        cout << "pre- NULL" << endl;
    }
    if (succ != NULL){
    else        same
```