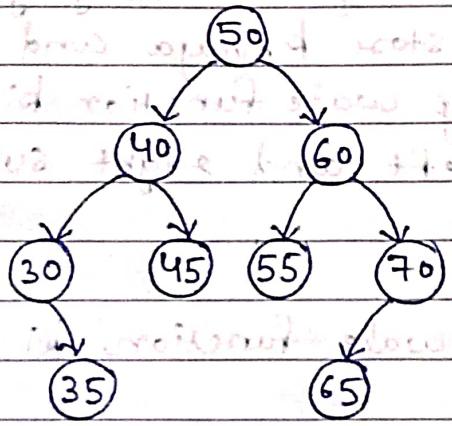
06/12/2023
Wednesday

Binary Search Tree –

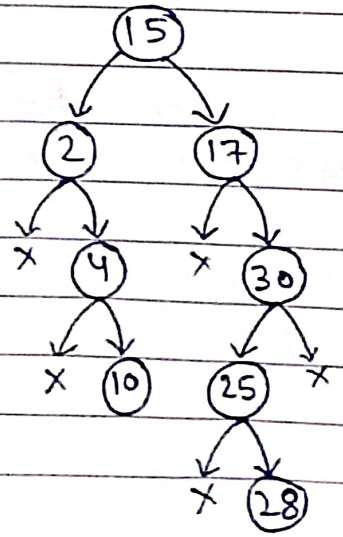
for every parent node in the tree, the data of the left child must be always smaller than its parent node and the data of the right child must be always greater than its parent node. This condition should be true at every node.



⇒ Right subtree data > Node → data > left subtree data

* let a tree be –

15 17 2 4 10 30 25 28 -1



* Insert a node in BST -

Code -

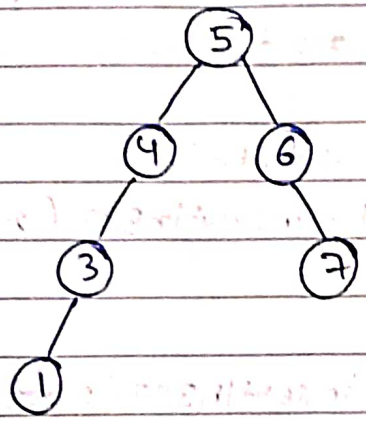
```
// this function inserts the data in BST
Node* insertInBST (Node* &root, int &data){

        if (root == NULL){
            root = new Node(data);
            return root;
        }

        if (data > root->data){
            root->right = insertInBST (root->right, data);
        }
        else{
            root->left = insertInBST(root->left, data);
        }

        return root;
    }
// this function takes the data as an input
void createBST (Node* &root){
        int data;
        cout<< "enter data: "<< endl;
        cin>> data;

        while (data != -1){
            root= insertInBST (root, data);
            cout<< "enter data: "<< endl;
            cin>> data;
        }
    }
```

T.C → worst case - O(n) when tree is skewed where n is no. of nodes in skewed tree. Normally, O(log n).

Note - The inorder traversal of binary search tree is always in sorted order.

Minimum value node in a BST -

leftmost node in a BST contains minimum value. If tree is empty, there is no minimum element, so return -1 in that case.



I/P - 5 4 6 3 N N 7 1

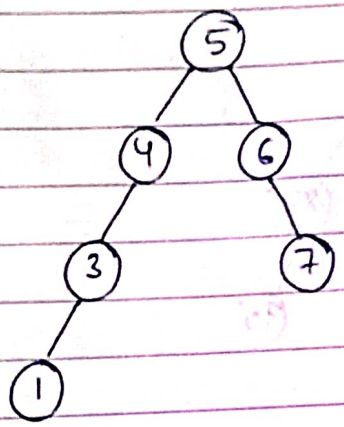
O/P - 1

Code -

```
int minValue (Node * root) {

    if (! root) return -1;

    Node * temp = root;
    while (temp -> left) temp = temp -> left;
    return temp -> data;
}
```

= Maximum value node in a BST -

Rightmost node in a BST contains maximum value. If tree is empty, there is no minimum element, so return -1 in that case.



I/P - 5 4 6 3 N N 7 1

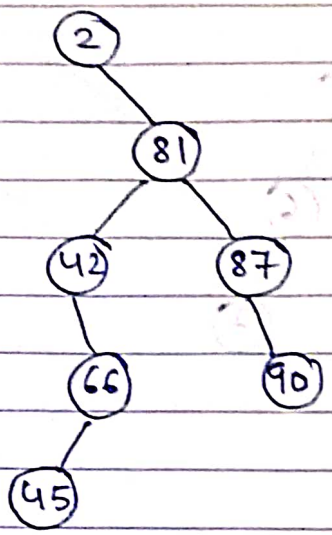
O/P - 7

Code -

```
int maxValue (Node * root) {

    if (!root) return -1;

    Node* temp= root;
    while (temp->right) temp= temp->right;
    return temp->data;
}
```

* **Search a node in BST -**

Given a binary search tree and a node value target, find if the node with value target is present in BST or not.



I/P- BST = 2 N 81 42 87 N 66 N 90 45 N
target = 87

O/P- true (1).

Code - Recursively

```
bool search (Node* root, int target){

    if (root==NULL) return 0;

    if (root->data == target) return 1;

    if (root->data >target) return search (root->left,
                                           target);
    else return search (root->right, target);
}
```

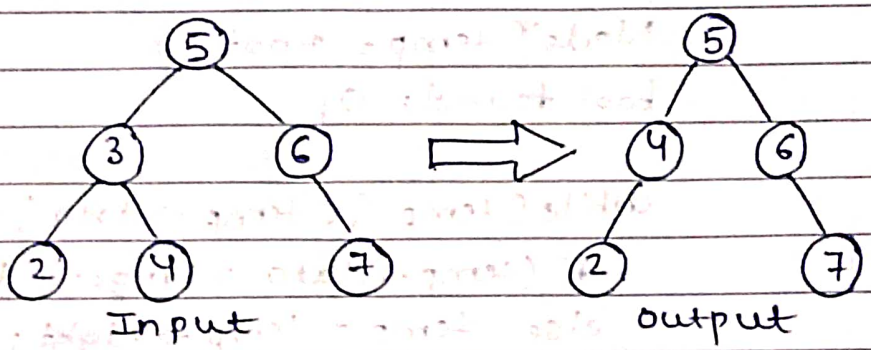
Code - using loop

```
bool search (Node* root, int target){

    if (!root) return 0;

    Node* temp = root;
    bool found = 0;

    while (temp && temp->data != target){
      if (temp->data > target) temp = temp->left;
      else temp = temp->right;
    }
    if (temp) found = 1;

    found = found ? 1 : 0;
    return found;
}
```

T.C. → worst case - O(n), skewed tree
Normally, O(log n).
for every case, we can say, T.C. will be O(height).

* **Delete node in a BST (leetcode-450)**

Given root node reference of a BST and a key, delete the node with the given key in the BST. Return the updated node of the BST.



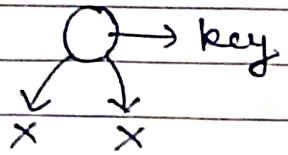
I/P - root = [5, 3, 6, 2, 4, null, 7], key = 3

O/P - [5, 4, 6, 2, null, null, 7]

Approach -

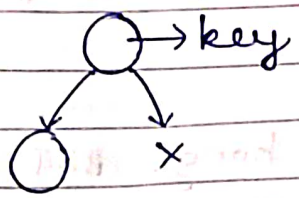
= If current node is a key node, 4 cases are there -

Case 1 - key node is a leaf node i.e.
root -> left == NULL && root -> right == NULL



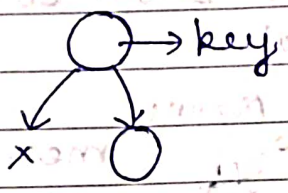
- Delete root.
- return null.

Case 2 - root->left != NULL && root->right == NULL



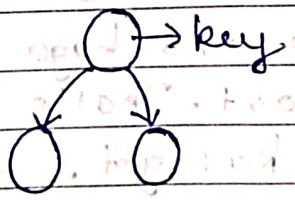
- Store the left subtree of the key node.
- Delete root i.e. key node.
- return stored left subtree.

Case 3 - root->left == NULL && root->right != NULL



- Store the right subtree of the key node.
- Delete root i.e. key node.
- return stored right subtree.

Case 4 - root->left != NULL && root->right != NULL



Delete the key node such that nature of BST remains maintained. It can be done in two ways -

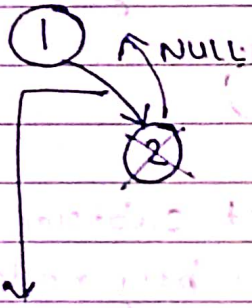
(1) Find max Value node in left subtree of key node and replace it with key node. Delete max Value node.

(2) Find min value node in right subtree of key node and replace it with key node. Delete min value node.

≈ If current node data is greater than key node data, then go to left in BST else go to right.

Note - Jo bhi changes hongi BST mei, unhe update krte chalna hai. Jha bhi changes hue hai either left or right of root, use left or right mei respectively lgana bhi hai. Hume btana pdega ki jo node ya root update hoke aari hai use kha lgana hai.

For instance, muze below BST mei 2 wali node ko delete krna hai -



mene 2 wali node ko delete kr diya or return kr diya null.

Agr mei simply return krdu or root ke right ko update na kru to ye pointer abhi bhi memory mei us deleted location pr point kr rha hoga which we don't want. That's why update root ka right.

Code -

```
class Solution{
public:

TreeNode* minNode (TreeNode* root) {
    if (!root) return NULL;

    TreeNode* temp = root;
    while (temp->left) temp = temp->left;
    return temp;
  }

TreeNode * deleteNode (TreeNode* root, int key){

    if (!root) return NULL;

    if (root->val == key){
      // 4cases
      if (!root->left && !root->right){
        delete root;
        return NULL;
      }
    else if (root->left && !root->right){
        TreeNode* childSubtree = root->left;
        delete root;
        return childSubtree;
      }
    else if (!root->left && root->right){
        TreeNode* childSubtree = root->right;
        delete root;
        return childSubtree;
      }
```

```
    else {
      // Find min value node in right subtree
       TreeNode* mini = minNode(root->right);
       root->val = mini->val; // replacement
      // delete min value node from right subtree
      root->right = deleteNode(root->right, mini->val);
      return root;
    }
  }
  else if (root->val > key)
    root->left = deleteNode(root->left, key);
  else
    root->right = deleteNode(root->right, key);

  return root;
  }
};
```

T.C. - O(height)

→ Concluding, insertion, deletion, searching, all of them can be done in time complexity, O(height).