14/05/2023

Binary Search Tree (BST)

We have studied about binary search on the array. For applying it our array should be sorted.

i/p → {10, 20, 30, 40, 50, 60}
target → {10}

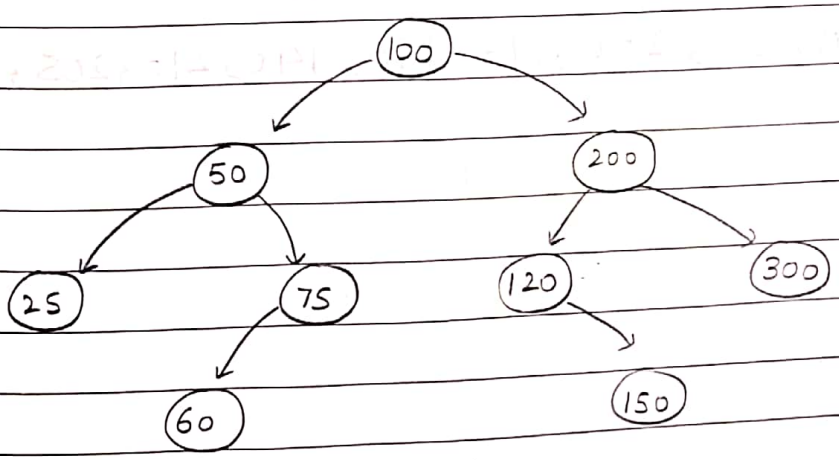
$mid = \frac{0+5}{2} = 2$

arr[mid] == target (X)
arr[mid] > target (✓)

10, 20

$mid = \frac{0+1}{2} = 0$

arr[mid] == target (✓)
Time complexity = O(logn)



BST can be defined as
root → data > (left subtree data)
root → data < (right subtree data)

Note → In BST, if there are duplicate values then it would be specified in the questions.

100 > 50, 25, 75, 60
100 < 200, 120, 300, 150
} Valid for root node

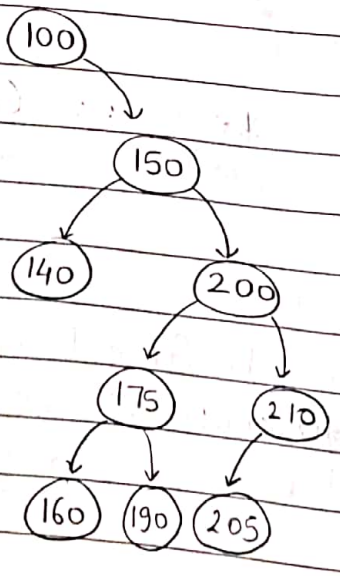
50 > 25
50 < 75, 60
25 } leaf nodes
60

200 > 150, 120 200 < 300
120 < 150
150 and 300 are leaf nodes.

Note → The property which we discussed should be valid for all the nodes. For leaf nodes, we assume it to be true.

Ex → Create a BST with following data.

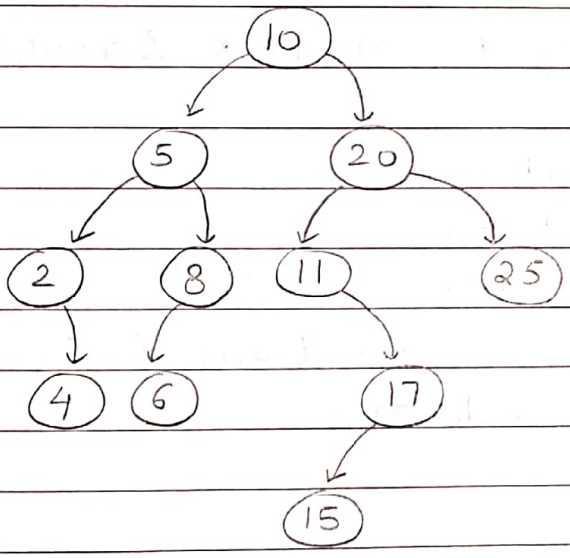
100, 150, 200, 175, 160, 140, 210, 205, 190



Just insert data according to the definition of BST.

Ex→ Create BST with following data

10, 20, 5, 11, 17, 2, 4, 8, 6, 25, 15



The above is how we create a binary search tree. Here we follow -1 as stopping criteria just like binary trees.

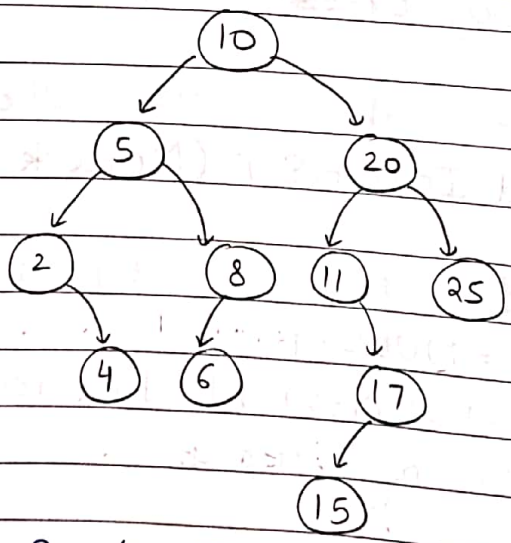
Code

```
Node * insertIntoBST (Node * root, int data)
{
    //1st node case (empty tree)
    if (root == NULL) {
        root = new Node (data);
        return root;
    }
    //insert into left
    if (root → data > data)
        root → left = insertIntoBST (root → left,
                                     data);
```

```
//insert into right
else{
    root->right = insertIntoBST(root->right,
                                data);
}
return root;
}
//called in main function
void takeInput(Node* &root){

    int data;
    cin>>data
    while(data!=-1){
        root = insertIntoBST(root,data);
        cin>>data;
    }
}
```

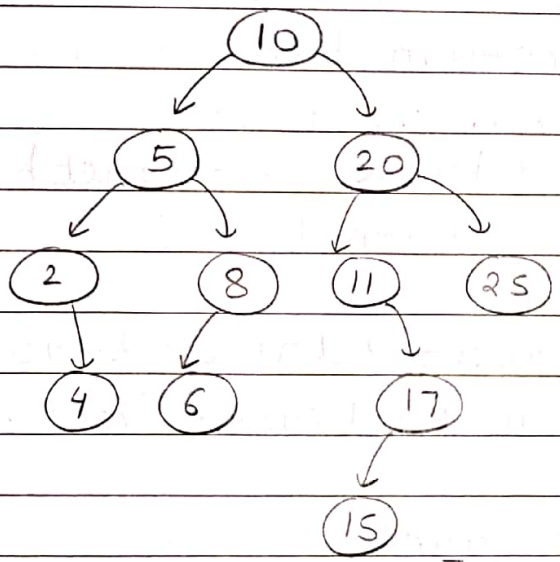
Note→ The codes of inorder, preorder & postorder traversal is same as that of binary trees.



Inorder: 2 4 5 6 8 10 11 15 17 20 25
Preorder: 10 5 2 4 8 6 20 11 17 15 25
Postorder: 4 2 6 8 5 15 17 11 25 20 10

Note → It is important to note inorder traversal of the Binary Search Tree is sorted.

Searching in Binary Search Tree



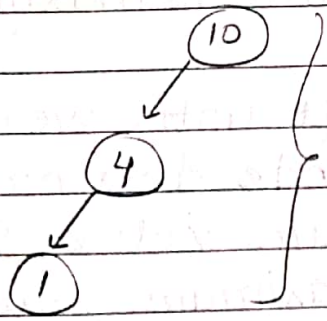
Target = 17

1) root → data == target] found
2) root → data > target] left
3) root → data < target] right

10 → 20 → 11 → (17)
↳ found

Time complexity in average case = O(h)
Height of BST in average case is logn.

Height of BST in worst case is O(n)
↳ skewed tree



Skewed BST

Code

```
bool findNode (Node * root, int target){
    // Base case
    if (root == NULL)
        return false;
    // Found
    if (root → data == target)
        return true;
    //Search in left
    else if (root → data > target)
        return findNode (root → left, target);
    }
    //Search in right
    else {
        return findNode (root → right, target);
    }
}
```

Note → Here order of recursive calls is not of concern. It can be in any order.

Maximum & minimum in BST

* Approach-1 ⇒ Find inorder and then 1st value is minimum and last value is maximum.

* Approach-2 ⇒ Go to left until we get left node such that left node does not exist further to get minimum value. Similarly go till right to get maximum value.

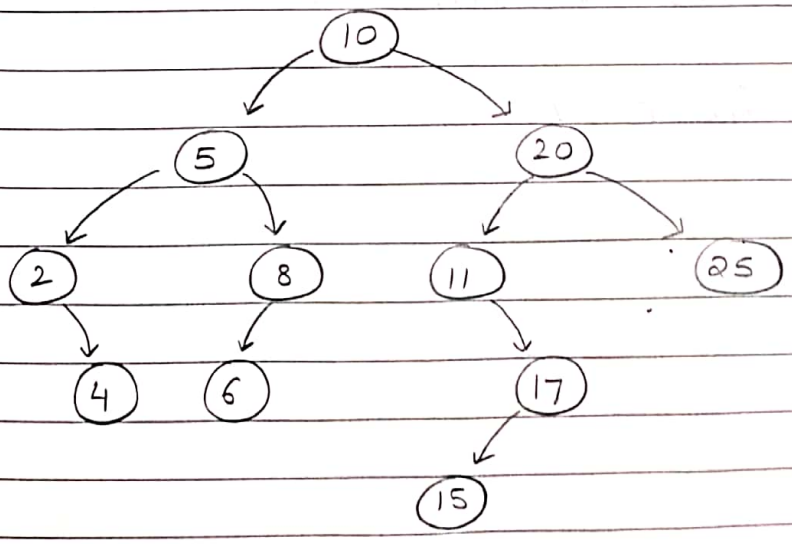
Code

```
int findMin (Node * root){

    Node * temp = root;
    if (temp == NULL)
            return -1;
    while (temp→left != NULL)
            temp = temp→left;
    return temp→data;
}
int findMax (Node * root){

    Node * temp = root;
    if (temp == NULL)
            return -1;
    while (temp→right != NULL)
            temp = temp→right;
    return temp→data;
}
```

Inorder predecsor / Successor



Inorder : 2 4 5 6 8 10 11 15 17 20 25

Inorder predeccesor of 11 : 10
Inorder successor of 11 : 15

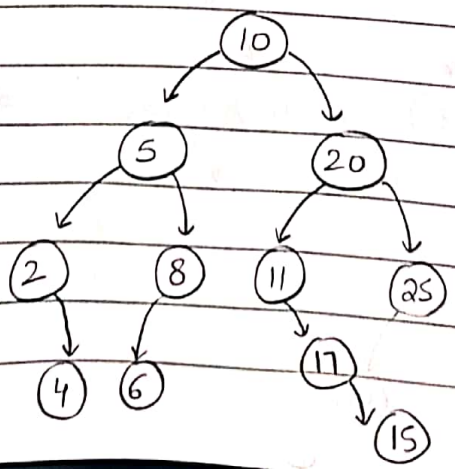
Here we don't have to store the inorder traversal.

Note → There are very high chance of getting a question of variation of deletion in BST.

* Inorder predeccesor means left subtree's maximum value.
* Inorder successor means right subtree's minimum value.

But the above 2 statements are not always true. Like 11 does not have anything in the left subtree but it has inorder predecessor as 10. If we need to find for 11, then we need to stove the inorder traversal and then we can find it.

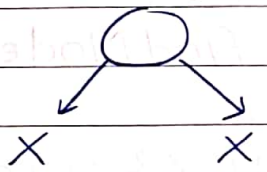
Deletion in BST



target = 25

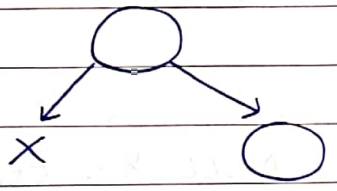
Step-1 ⇒ Search for 25 in the BST

Case-1



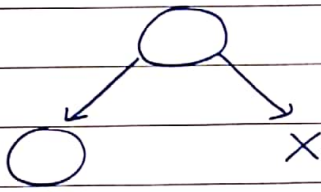
Return NULL

Case-2



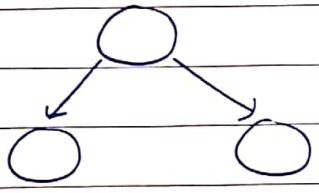
Return root→right

Case-3



Return root→left

Case-4



Suppose we have to delete 5 in the BST. We have to delete in such a way that after deletion, tree is BST.

In this case instead of 5, place inorder predeccesor & delete that node. (inorder predecesor node).

Code

```
Node * deleteNodeInBST (Node *root,
int target) {
```

```
// Base case
if (root == NULL)
        return root;
if (root -> data == target) {
    // Case 1 : Leaf node
    if (root -> left == NULL && root -> right == NULL) {
            return NULL;
    }
    // Case 2 : Left child not exist
    else if (root -> left == NULL && root -> right != NULL)
        Node * child = root -> right;
        return child;
    }
    // Case 3 : Right child not exist
    else if (root -> left != NULL && root -> right == NULL)
        Node * child = root -> left;
        return child;
    }
    // Case 4 : Both child exist
    else {
        int inorderPre = findMax (root -> left);
        root -> data = inorderPre;
        root -> left = deleteNodeInBST (root -> left,
        inorderPre);
        return root;
    }
} // Search in right subtree
else if (root -> data < target) {

    root -> right = deleteNodeInBST (root -> right,
    target);
}
```

```
else { // search in left subtree
        root->left = deleteNodeInBST(root->left,
        target);
}
return root;
}
```