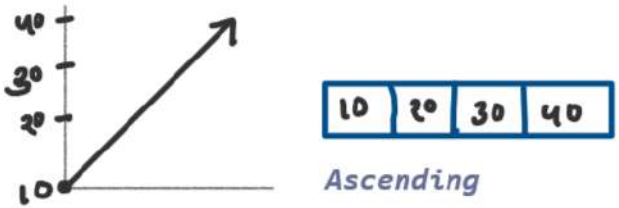
06/12/2023

BINARY SEARCH TREE
CLASS - 1

1. What is Binary Search Tree?

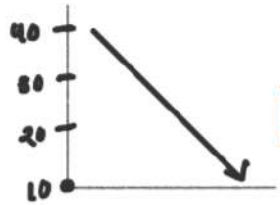
Binary search

- Binary search is always work on **MONOTONIC** array
- **MONOTONIC** -> **Ascending** or **Descending** order of array



10	20	30	40

Ascending



40	30	20	10

Descending

Arr

10	20	30	40	50	60	70	80
0	1	2	3	4	5	6	7

S=0

E=7

$mid = \left(\frac{S+E}{2}\right)$

Target = 70

Output = found

S=4
e=7
mid=5

50	60	70	80
4	5	6	7

S=6
e=7
mid=6

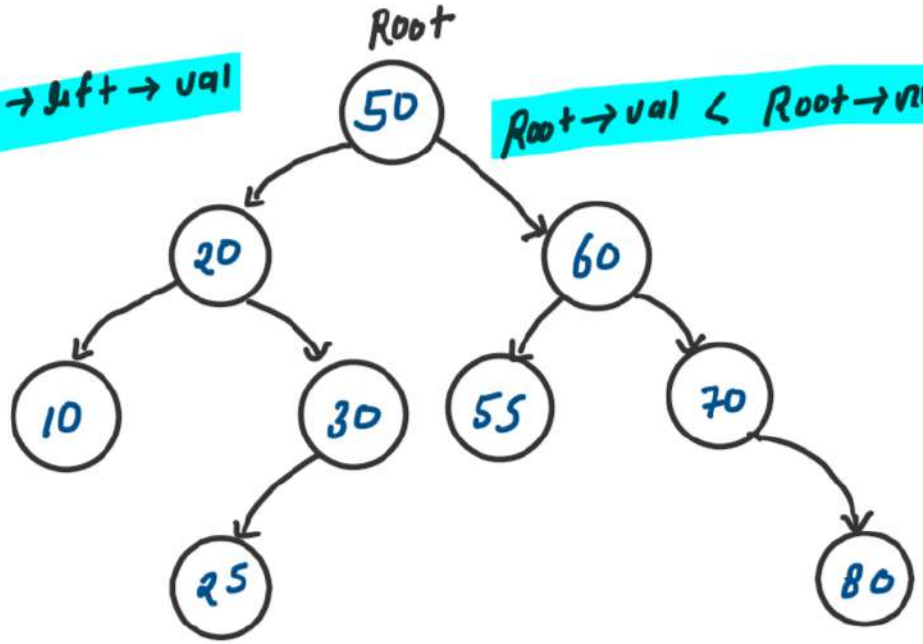
70	80
6	7

O/P = Target Found

Binary search Tree

Root->val > Root->left->val

Root->val < Root->right->val



50 > 20, 10, 30, 25

Left Sub Tree

50 < 60, 55, 70, 80

Right Sub Tree

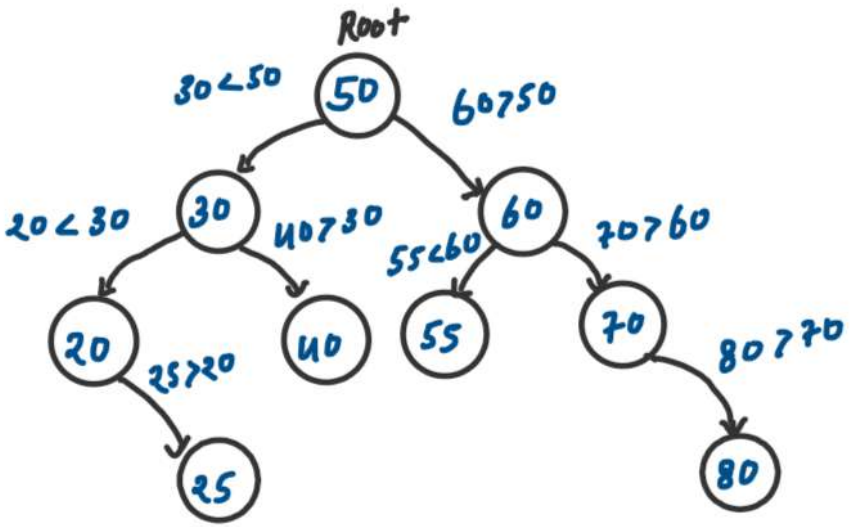
{Right > Node->val > Left}

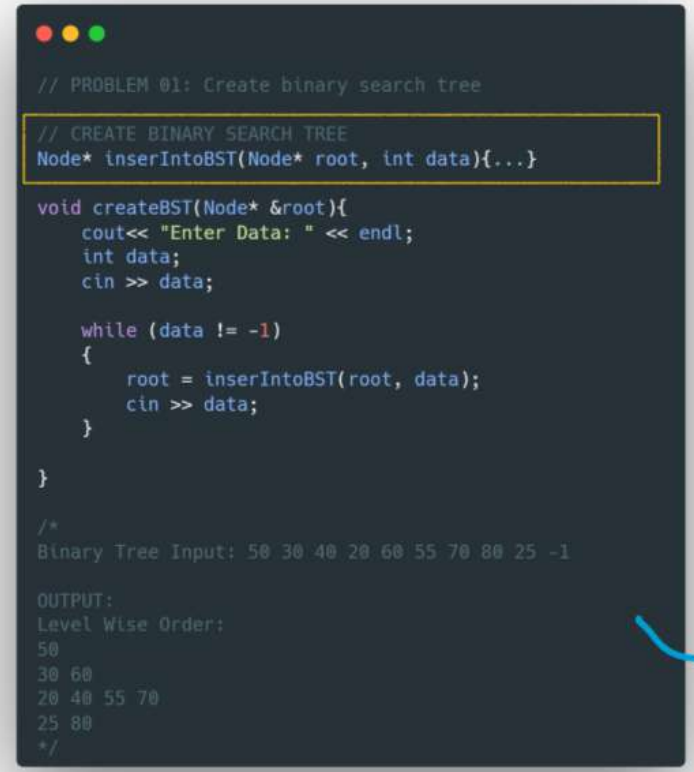
↳ Binary search tree property

2. Create Binary Search Tree

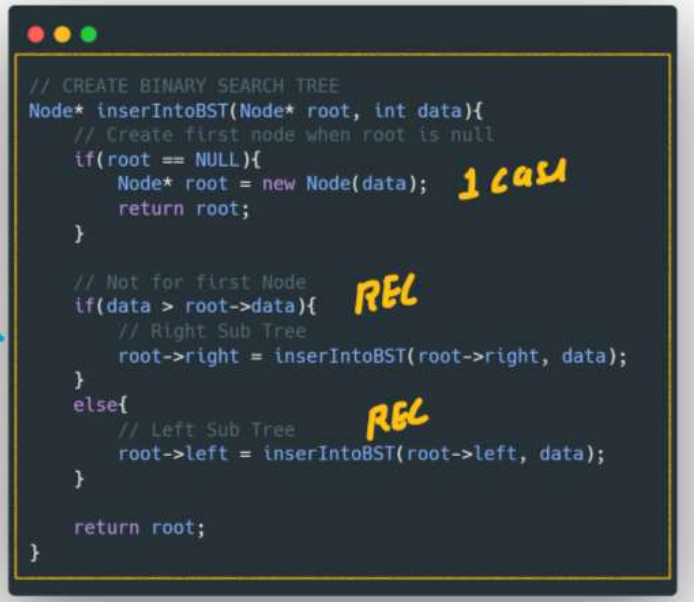
Input → 50 30 40 20 60 55 70 80 25 -1

-1 → STOP





```
// PROBLEM 01: Create binary search tree

// CREATE BINARY SEARCH TREE
Node* inserIntoBST(Node* root, int data){...}

void createBST(Node* &root){
    cout<< "Enter Data: " << endl;
    int data;
    cin >> data;

    while (data != -1)
    {
        root = inserIntoBST(root, data);
        cin >> data;
    }

}

/*
Binary Tree Input: 50 30 40 20 60 55 70 80 25 -1

OUTPUT:
Level Wise Order:
50
30 60
20 40 55 70
25 80
*/
```



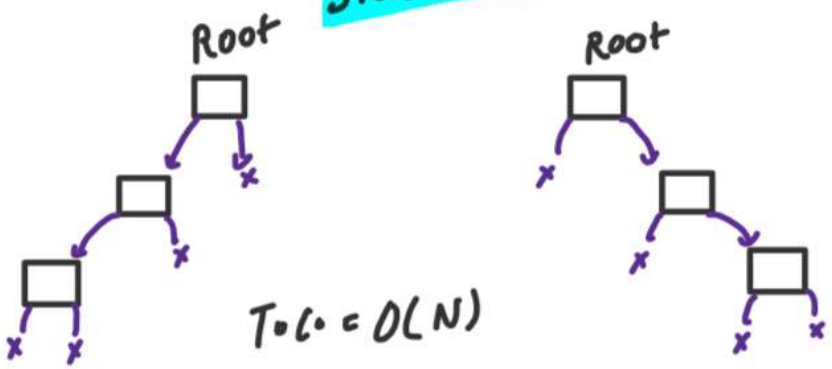
```
// CREATE BINARY SEARCH TREE
Node* inserIntoBST(Node* root, int data){
    // Create first node when root is null
    if(root == NULL){
        Node* root = new Node(data);
        return root;
    }

    // Not for first Node
    if(data > root->data){
        // Right Sub Tree
        root->right = inserIntoBST(root->right, data);
    }
    else{
        // Left Sub Tree
        root->left = inserIntoBST(root->left, data);
    }

    return root;
}
```

1 case

REC

REC

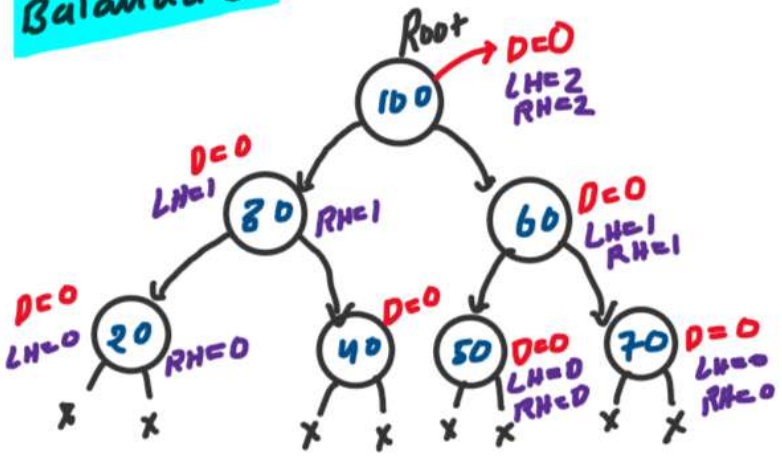
Skewed BT



$T.C. = O(N)$

Where, N is number of nodes in the Binary Tree

Balanced BT



$T.C. = O(\log N)$

Overall time complexity:
Worst Case: O(N)
Average Case: O(N log N)
Where N is number of nodes in binary search tree

Space Complexity: *O(H), Where H is height of BST*

3. Binary Search Tree Traversals

All Traversals are same as Binary Tree

PRE-ORDER (NLR)

50 30 20 25 40 60 55 70 80

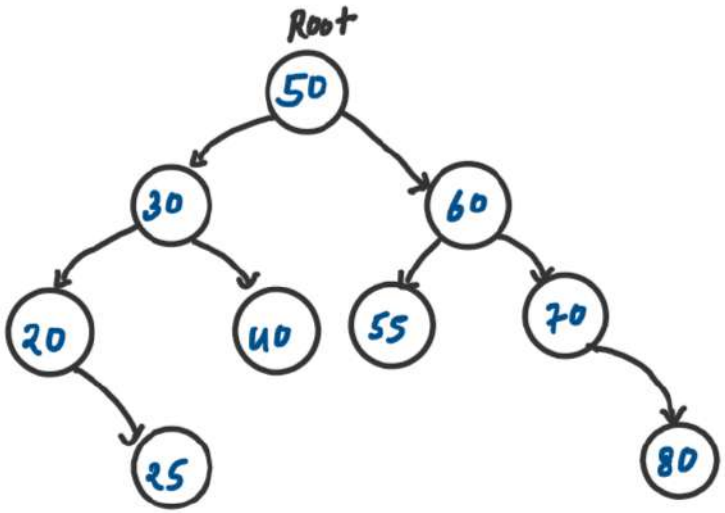
IN-ORDER (LNR)

20 25 30 40 50 55 60 70 80

POST-ORDER (LRN)

25 20 40 30 55 80 70 60 50

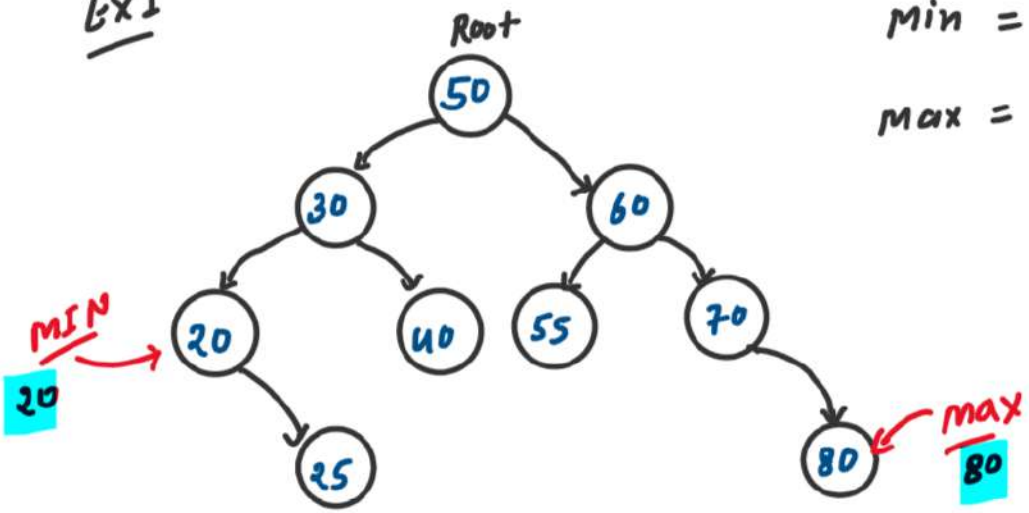
In-Order Always Print Nodes in Increasing Order



*Time and **space complexity**: O(N), where N is total number of nodes in BST*

4. Min and Max Value in BST

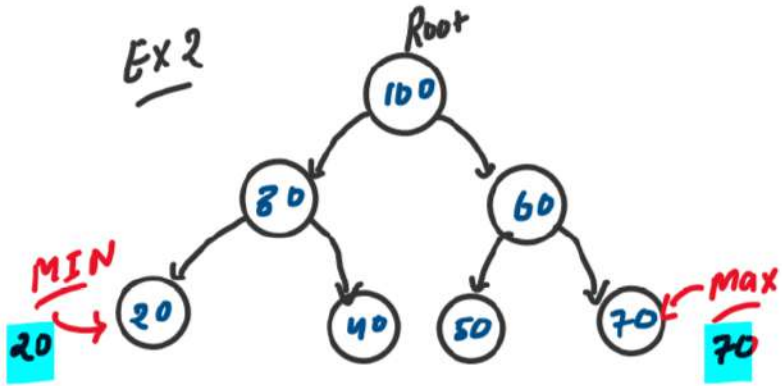
Ex 1



Min = 20 → min value always Root ke left me milegi

Max = 80 → max value always Root ki Right me milegi

Ex 2



```cpp
// PROBLEM 04: Min and Max value in BST

// Min value of BST
Node* minValue(Node* root){
    if(root == NULL){
        // No min value of BST
        return NULL;
    }

    Node* temp = root;
    while(temp->left != NULL){
        temp = temp->left;
    }
    return temp;
}

// Max value of BST
Node* maxValue(Node* root){
    if(root == NULL){
        // No max value of BST
        return NULL;
    }

    Node* temp = root;
    while(temp->right != NULL){
        temp = temp->right;
    }
    return temp;
}

/*
Binary Tree Input: 50 30 40 20 60 55 70 80 25 -1

OUTPUT:
Min Value: 20
Max Value: 80
*/
```

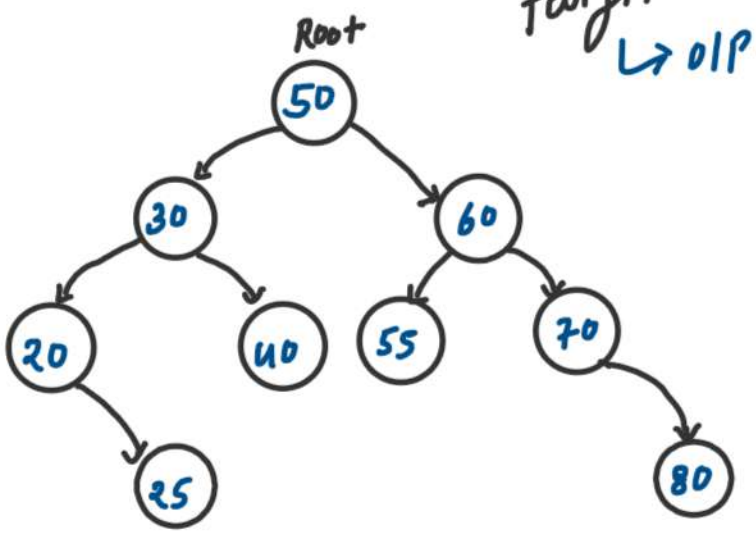
Time complexity:
Skewed BST: O(N) or O(H) in the worst case
Balanced BST: O(log N) in the average case

Space Complexity: *O(H)*
In the worst case: H is equal to O(N)
In the average case: H is equal to O(log N)

Where H is height of BST and N is number of nodes in BST

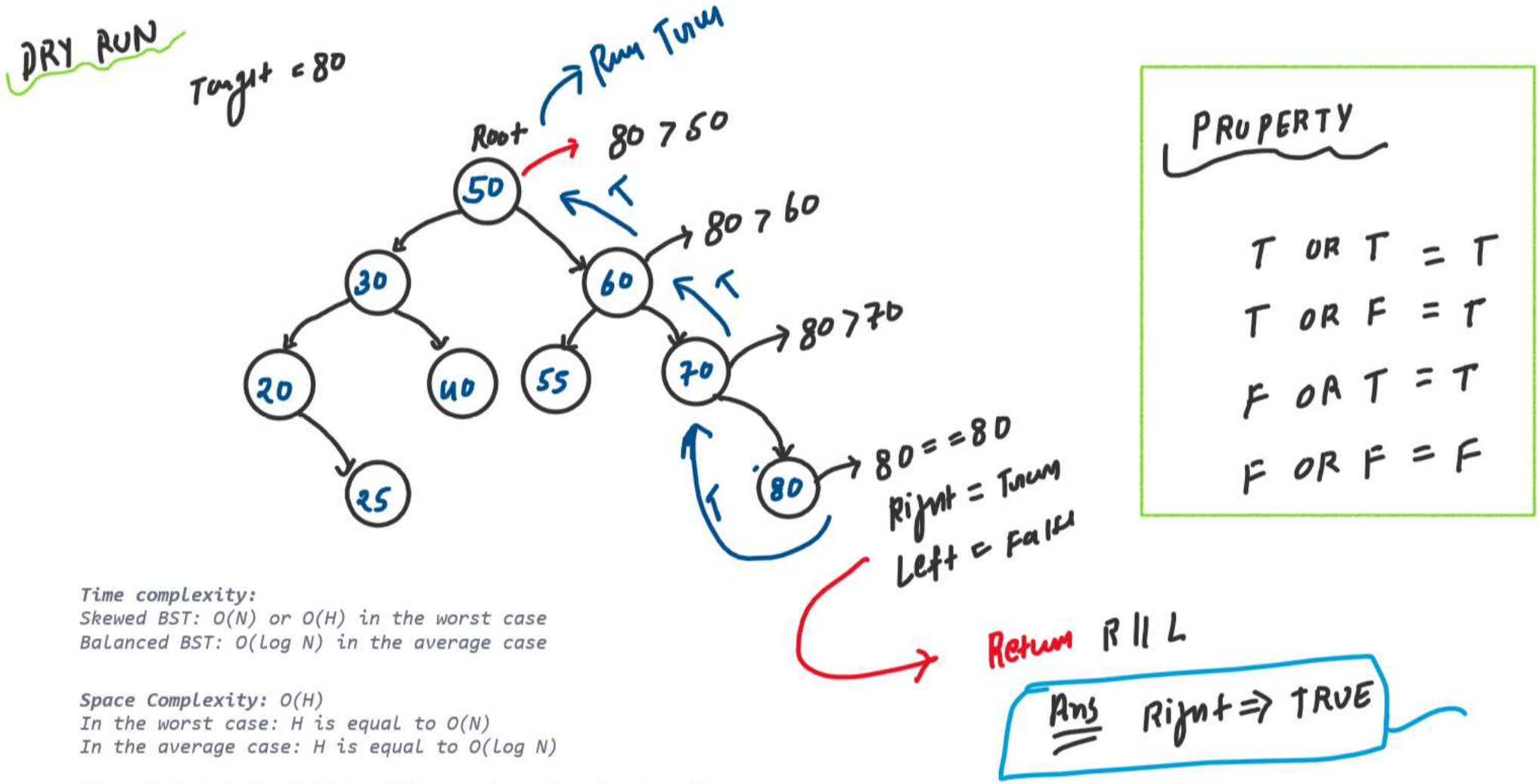
📂 5. Target Value is Present or not in BST

Ex 1



Target = 50
↳ O/P = True

```cpp
// PROBLEM 04: Target value present or not in BST

bool searchInBST(Node* root, int target){
    // Base Case
    if(root == NULL){
        return false;
    }

    // 1 case hum solve kr lenge
    if(target == root->data){
        return true;
    }

    bool right = false;
    bool left = false;

    // ab recursion solve kar lega
    if(target > root->data){
        right = searchInBST(root->right, target);
    }
    if(target < root->data){
        left = searchInBST(root->left, target);
    }

    return right || left;
}

/*
Binary Tree Input: 50 30 40 20 60 55 70 80 25 -1

Enter the target: 80
OUTPUT: FOUND

Enter the target: 2222
OUTPUT: NOT FOUND
*/
```



Time complexity:
Skewed BST: O(N) or O(H) in the worst case
Balanced BST: O(Log N) in the average case

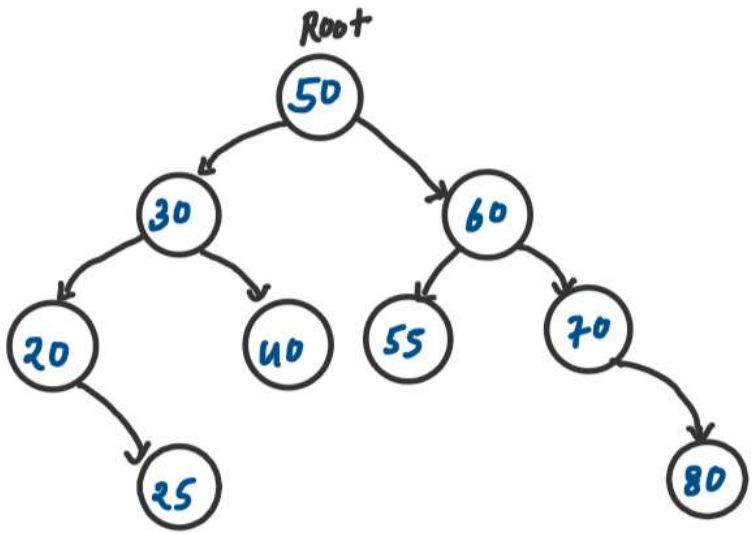
Space Complexity: O(H)
In the worst case: H is equal to O(N)
In the average case: H is equal to O(Log N)

Where H is height of BST and N is number of nodes in BST

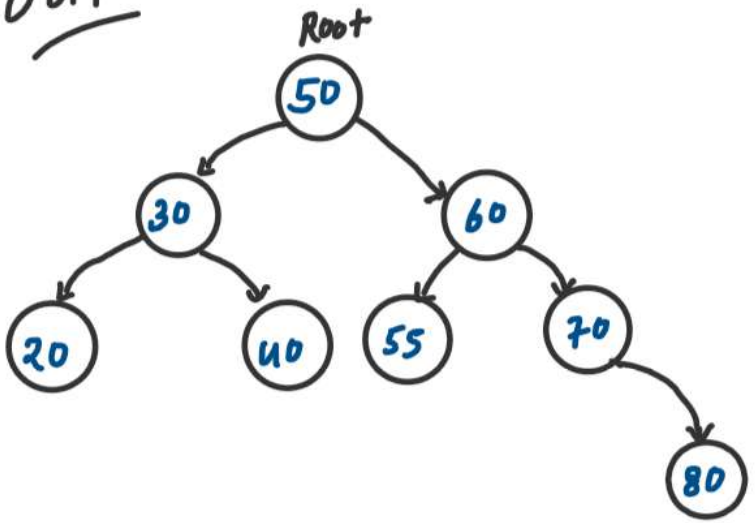
6. Delete Node from BST

Ex 1

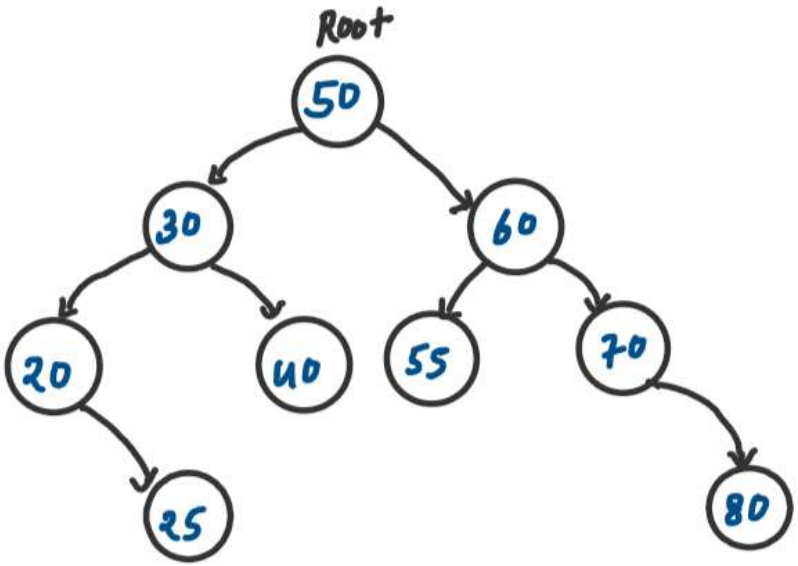
Target = 25



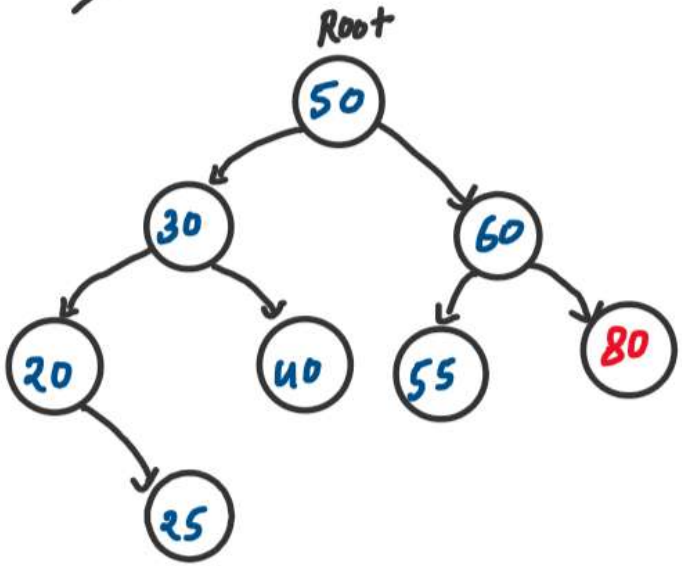
Output



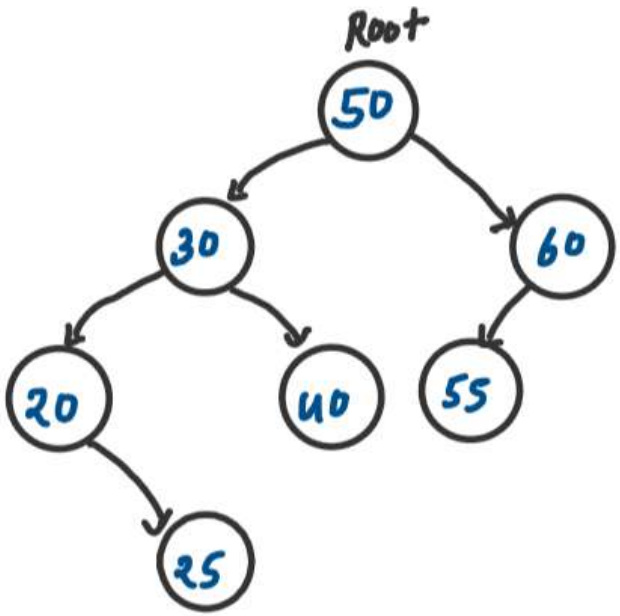
EX2 Target = 70



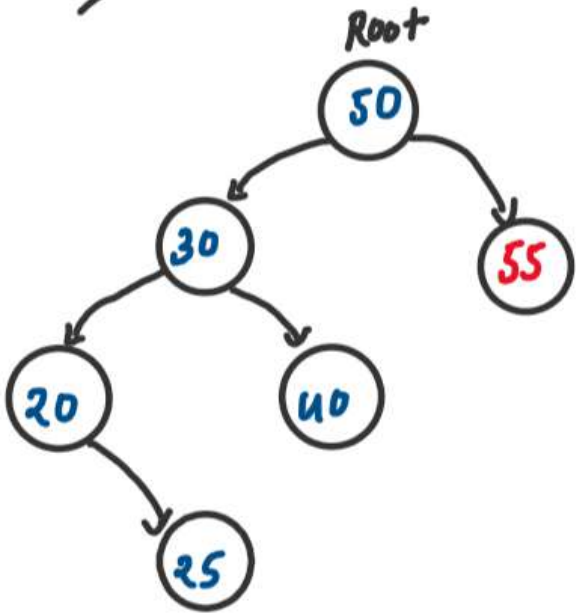
Output



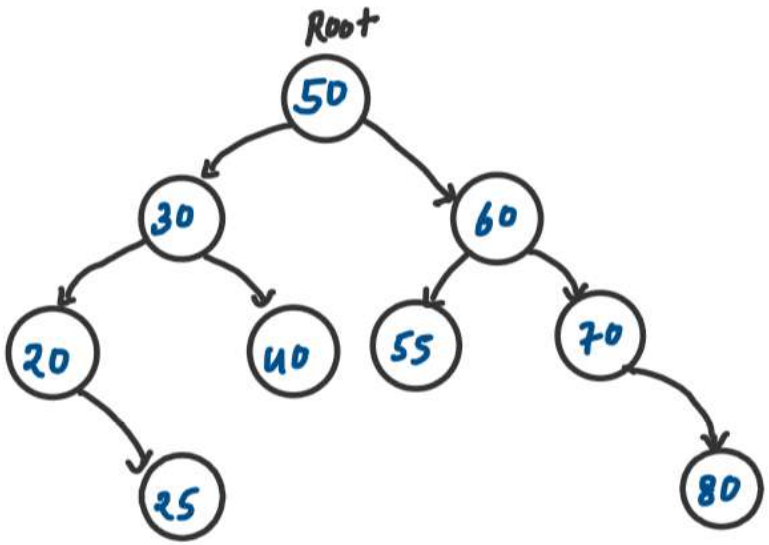
EX3 Target = 60



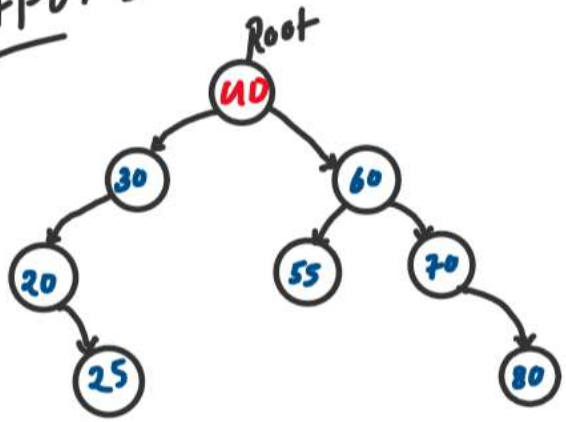
Output



Ex 4 Target = 50

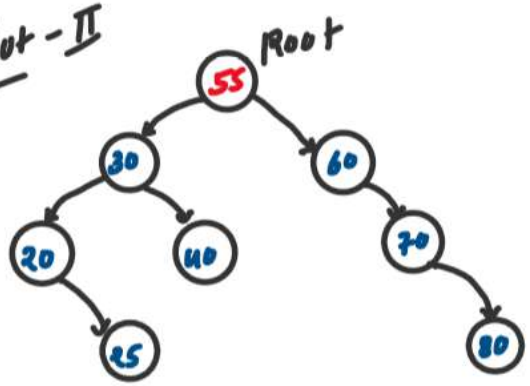


Output I

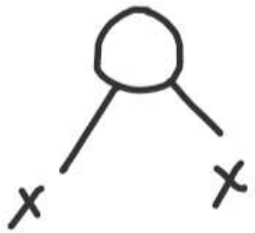


OR

Output - II



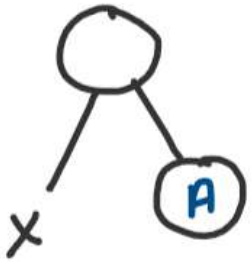
CASE1 Leaf Node



return Null

EX1

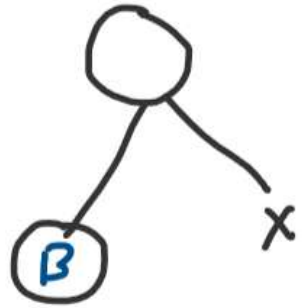
CASE2



return A

EX2

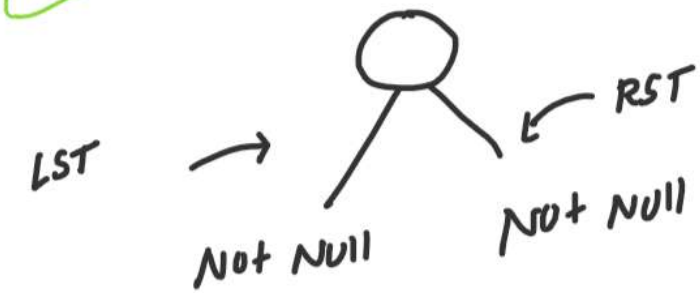
CASE3



return B

EX3

CASE 4



Rule

LST ⇒ Max Node Find Karlo
↳ Replace with target Node
↳ delete max Node from LST

OR

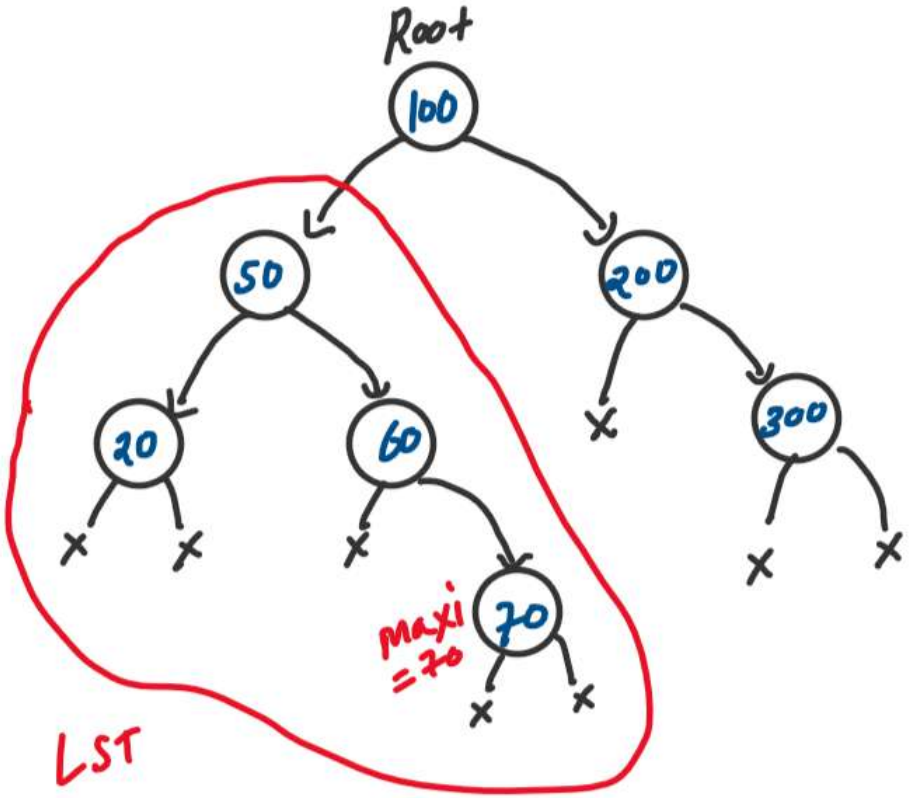
RST ⇒ min Node Find Karlo
↳ Replace with target Node
↳ delete min Node from RST

```cpp
// Delete node from BST
Node* deleteFromBST(Node* root, int target){
    // Base case
    if(root == NULL){
        return NULL;
    }

    // Step 01: target ko find krlo
    // Target root v ho skta hai
    if(target == root->data){
      ...
    }

    // Target root ke left me v ho skta hai
    else if(target < root->data){
        root->left = deleteFromBST(root->left, target);
    }
    // Target root ke right me v ho skta hai
    else{
        root->right = deleteFromBST(root->right, target);
    }

    return root;
}
```

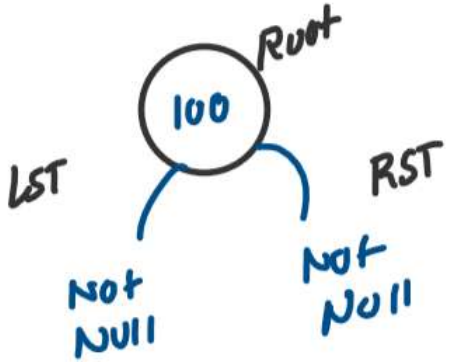
```cpp
    // Step 01: target ko find krlo
    // Target root v ho skta hai
    if(target == root->data){
        // Step 02: target ko delete krdo
        // Agar target mil jata hai to 4 cases me delete kiya jayega

        // Case 1: leaf node
        if(root->left == NULL && root->right == NULL){
            delete root;
            return NULL;
        }
        // Case 2: left me NULL and right NON NULL
        else if(root->left == NULL && root->right != NULL){
            Node* childSubTreeA = root->right;
            delete root;
            return childSubTreeA;
        }
        // Case 3: left non null and right null
        else if(root->left != NULL && root->right == NULL){
            Node* childSubTreeB = root->left;
            delete root;
            return childSubTreeB;
        }
        // Case 4: left NON NULL && right NON NULL
        else if(root->left != NULL && root->right != NULL){
            // Left Sub tree se MAX value find krlo
            Node* maxi = maxValue(root->left);

            // Replace maxi with target value
            root->data = maxi->data;

            // Delete actual maxi->data (YANHA MENE GALTI KARI THI)
            root->left = deleteFromBST(root->left, maxi->data);
            return root;
        }
    }
```

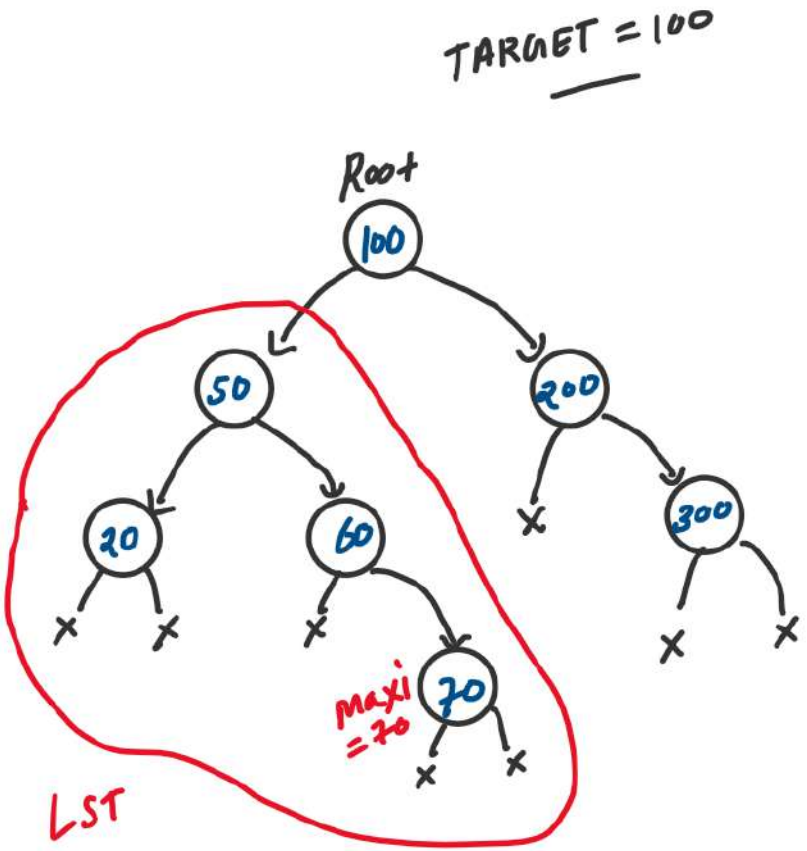
DRY RUN

TARGET = 100

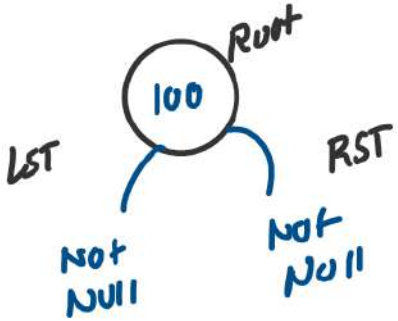


(1) Target ko Find karo





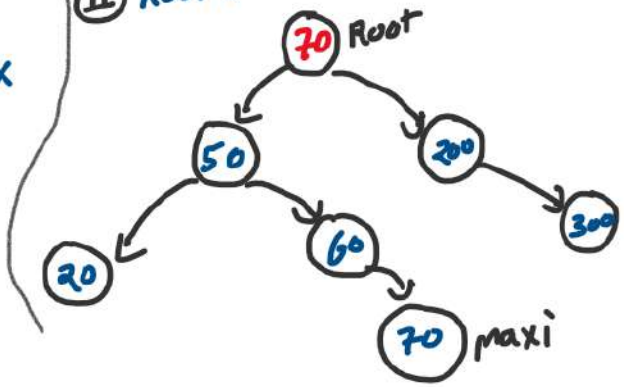
(1) Target ko Find Karo



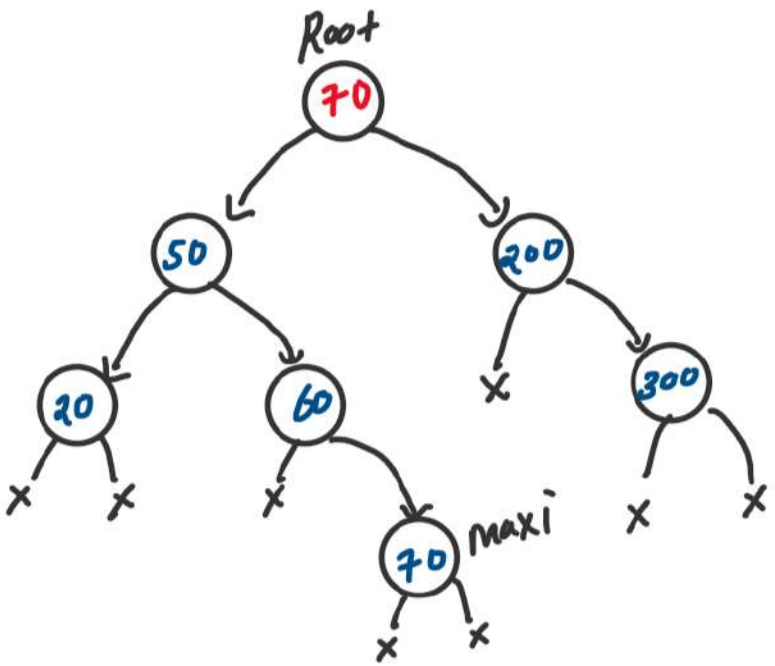
(2) Maxi value node from LST

(I) Maxi = 

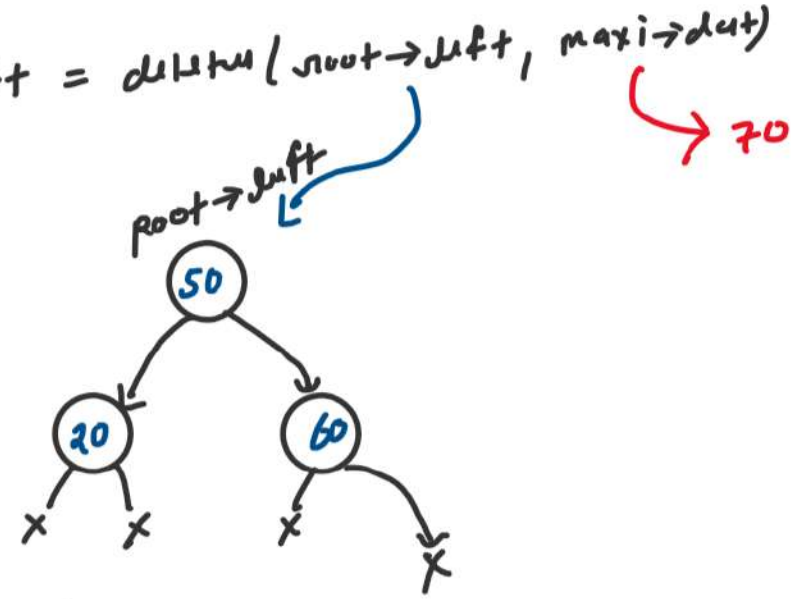
(II) Root → data = maxi → data



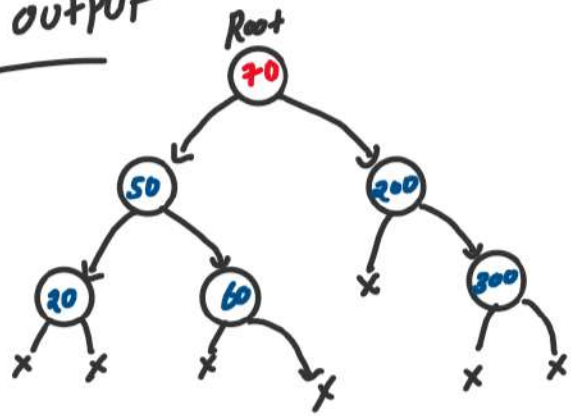
(II) Root→data = maxi→data



(III) Root→left = deleteNode(root→left, maxi→data)

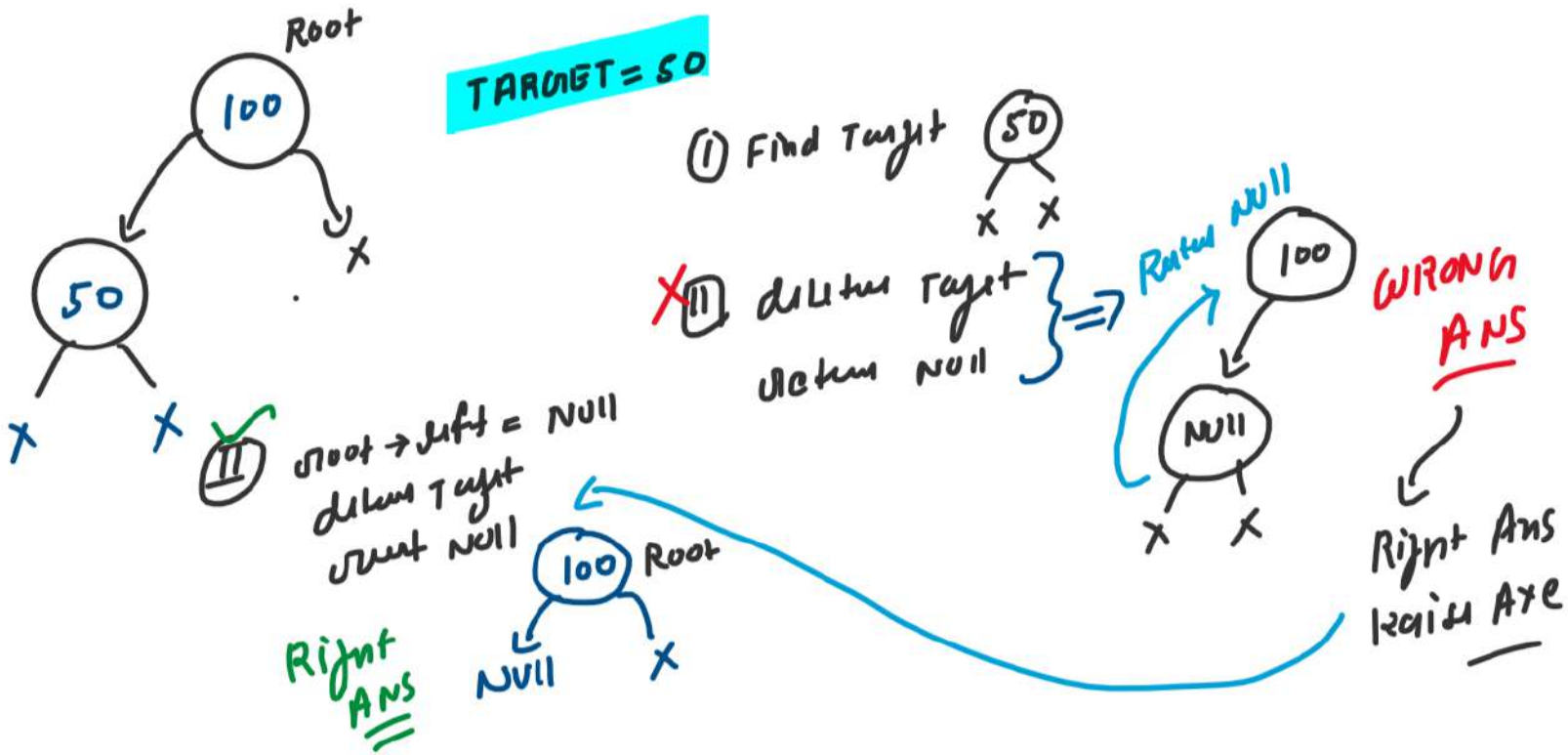


Final output



DRY RUN

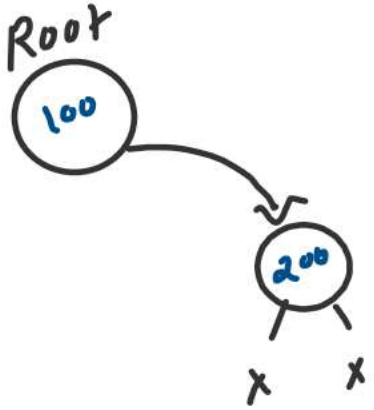
EX-2


Root
100
50
X
X
X
II
TARGET = 50
(I) Find Target
50
X X
X II delete target
delete NULL
Return NULL
100
NULL
X X
WRONG ANS
Right Ans
Recall Axe
root → left = NULL
delete Target
return NULL
100
Root
NULL
X
Right ANS

DRY RUN

EX3

Target = 200



(I) Find Target

(II) root->right = delete(root->right, Target)

NULL