# Trees: Level 3 Lesson 1

minimum-element-in-bst
search-a-node-in-bst
array-to-bst
Binary Tree to BST
Find Common Nodes in two BSTs
print-bst-elements-in-given-range

# **Binary Search Tree**

**Binary Search Tree** is a binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

# Minimum element in BST

Given a **Binary Search Tree**. The task is to find the minimum element in this given BST.

**Expected Time Complexity:** O(Height of the BST)

**Expected Auxiliary Space:** O(Height of the BST).

#### **Constraints:**

```
1 \le N \le 104
```

```
    int minValue(Node* root)
    {
        if(root->left == NULL)
        return root->data;
        minValue(root->left);
        6.
        // Code here
        8. }
        // Code here
```

# Search a node in BST

Given a **Binary Search Tree** and a node value X, find if node with value X is present in the BST or not.

**Expected Time Complexity:** O(Height of the BST)

**Expected Auxiliary Space:** O(1).

#### **Constraints:**

 $1 \le \text{Number of nodes} \le 105$ 

```
1. bool search(Node* root, int x)
2. {
3. if (root == NULL)
4. return false;
5. if (root->data == x)
6.
       return true;
7.
8.
    // Key is greater than root's key
     if (root->data < x)
9.
10.
      return search(root->right, x);
11.
12. // Key is smaller than root's key
13. return search(root->left, x);
14.}
```

# Print BST elements in given range

Given a Binary Search Tree and a range. Find all the numbers in the BST that lie in the given range.

**Note:** Elements greater than or equal to root go to the right side.

#### Example 1:

#### **Input:**

17

/ \

4 18

/ \

2 9

1 = 4, h = 24

**Output:** 4 9 17 18

#### Example 2:

#### **Input:**

16

/ \

7 20

/ \

1 10

$$1 = 13, h = 23$$

**Output:** 16 20

#### **Expected Time Complexity:** O(N).

**Expected Auxiliary Space:** O(Height of the BST).

#### **Constraints:**

18.}

```
1 \le Number of nodes \le 104
1 \le 1 \le h \le 105
   1. void nodes(Node*root,int low,int high,vector<int>&v)
   2. {
       if(root==NULL)
   3.
   4.
          Return;
   5. //inorder traversal
       nodes(root->left,low,high,v);
   6.
       if(root->data>=low&&root->data<=high)
   7.
   8.
          v.push back(root->data);
       nodes(root->right,low,high,v);
   9.
   10.}
   11.vector<int> printNearNodes(Node *root, int low, int high)
   12.{
   13. vector<int>v;
   14. nodes(root,low,high,v);
   15. //sort(v.begin(),v.end());
   16. return v;
   17. // your code goes here
```

### Array to BST

Given a sorted array. Convert it into a Height balanced Binary Search Tree (BST). Height balanced BST means a binary tree in which the depth of the two subtrees of every node never differ by more than 1.

#### Example 1:

**Input:** nums =  $\{1, 2, 3, 4\}$ 

**Output:** {3, 2, 1, 4}

#### **Explanation:**

The preorder traversal of the following BST formed is {3, 2, 1, 4}:

3

/ \

2 4

/

1

#### Example 2:

**Input:** nums =  $\{1,2,3,4,5,6,7\}$ 

**Output:** {4,2,1,3,6,5,7}

#### **Explanation:**

The preorder traversal of the following BST formed is  $\{4,2,1,3,6,5,7\}$ :

```
4
/\
2 6
/\ /\
1 357
```

#### **Expected Time Complexity:** O(n)

**Expected Space Complexity:** O(n)

#### **Constraints:**

```
1 \le |nums| \le 104
-10^4 \le nums[i] \le 10^4
```

```
void array(int left , int right ,vector<int>&nums,vector<int>&v)
1.
2.
        if(left<=right)</pre>
3.
4.
        int mid=(left+right)/2;
5.
6.
        v.push back(nums[mid]);
7.
        array(left,mid-1,nums,v);
8.
        array(mid+1,right,nums,v);
9.
        }
10.
        vector<int> sortedArrayToBST(vector<int>& nums)
11.
12.
        {
          int n=nums.size();
13.
14.
          vector<int>v;
15.
          array(0,n-1,nums,v);
16.
          return v;
17.
```

# Binary Tree to BST

Given a Binary Tree, convert it to Binary Search Tree in such a way that keeps the original structure of Binary Tree intact.

# Example 1: **Input:** 1 / \ 2 3 **Output:** 1 2 3 Example 2: **Input:** 1 / \ 2 3 / 4 **Output:** 1 2 3 4 **Explanation:** The converted BST will be 3 / \ 2 4

/

1

#### **Expected Time Complexity:** O(NLogN).

**Expected Auxiliary Space:** O(N).

#### **Constraints:**

 $1 \le \text{Number of nodes} \le 1000$ 

```
1. void inorder(Node*root, vector<int>&v)
2. {
3.
      if(root==NULL)
4.
       return;
5.
      inorder(root->left,v);
      v.push back(root->data);
6.
7.
      inorder(root->right,v);
8. }
9. void push el(Node*root, vector<int>&v,int &i)
10. {
11.
      if(root==NULL)
12.
       return;
      push el(root->left,v,i);
13.
14.
     root->data=v[i];
15.
      i++;
      push el(root->right,v,i);
16.
17. }
18. Node *binaryTreeToBST (Node *root)
19. { int i=0;
20.
       vector<int>v;
21.
       inorder(root,v);
       sort(v.begin(),v.end());
22.
       push el(root,v,i);
23.
       return root; }
24.
```

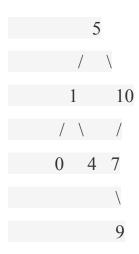
# Find Common Nodes in two BSTs

Given two Binary Search Trees. Find the nodes that are common in both of them, ie- find the intersection of the two BSTs.

#### Example 1:

**Input:** 

BST1:

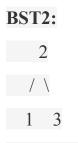


#### BST2:

10 / \ 7 20 / \ 4 9

**Output:** 4 7 9 10

# Example 2: Input: BST1: 10 / \ 2 11 / \ 1 3



**Output:** 1 2 3

**Expected Time Complexity:** O(N1 + N2) where N1 and N2 are the sizes of the 2 BSTs.

**Expected Auxiliary Space:** O(H1 + H2) where H1 and H2 are the heights of the 2 BSTs.

#### **Constraints:**

1 <= Number of Nodes <= 105