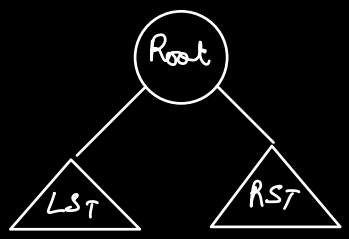
Q Given a BT. Check if it is a BST.

LST < Root.val < RST



LST , Root.val , RST

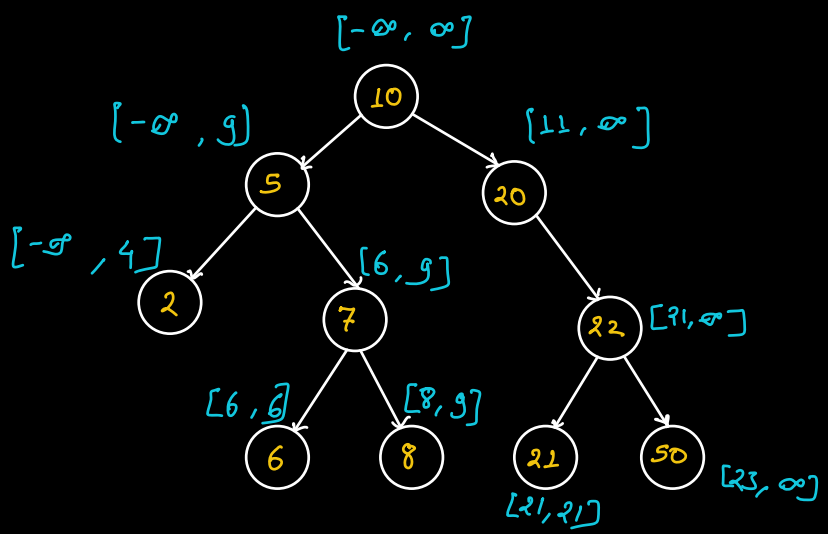
Inorder traversal of BST ⟶ Sorted in ASC order.

Inorder

* Store the inorder traversal in an array
* Check if it is sorted.

TC : O(N)
SC : O(N)

Pre-Order



```
boolean isBST(root, l, r) {          // l = -∞, r = ∞
                                     // l → Null, r → Null
    if(root == Null) ret true;

    if(l <= root.val && r >= root.val) {
        boolean left = isBST(root.left, l, root.val-1);
        boolean right = isBST(root.right, root.val+1, r);
        ret left && right;
    }
    ret false;
}
```

root.val → ∞

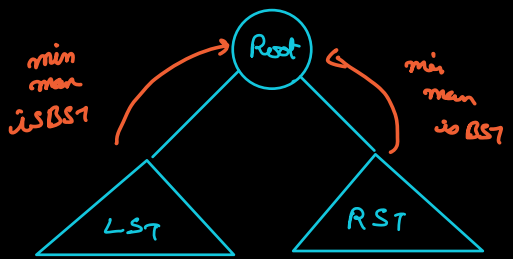
Post-Order

∀ nodes

node.val > max(LST)

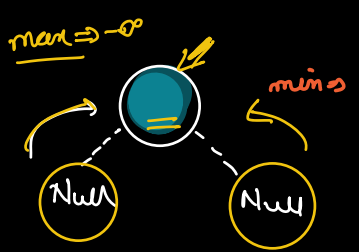
node.val < min(RST)

Root → min, max, isBST



Prob ;
Left
Right
root;

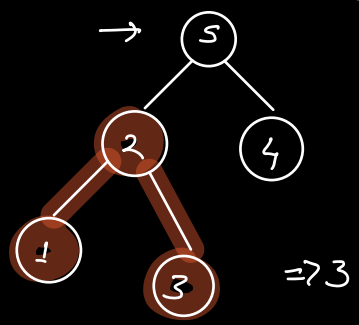
```
class TreeInfo {
    int min;
    int max;
    boolean isBST;
    public TreeInfo(int a, int b, boolean isBST) {
        this.min = a;
        this.max = b;
        this.isBST = isBST;
    }
}
```



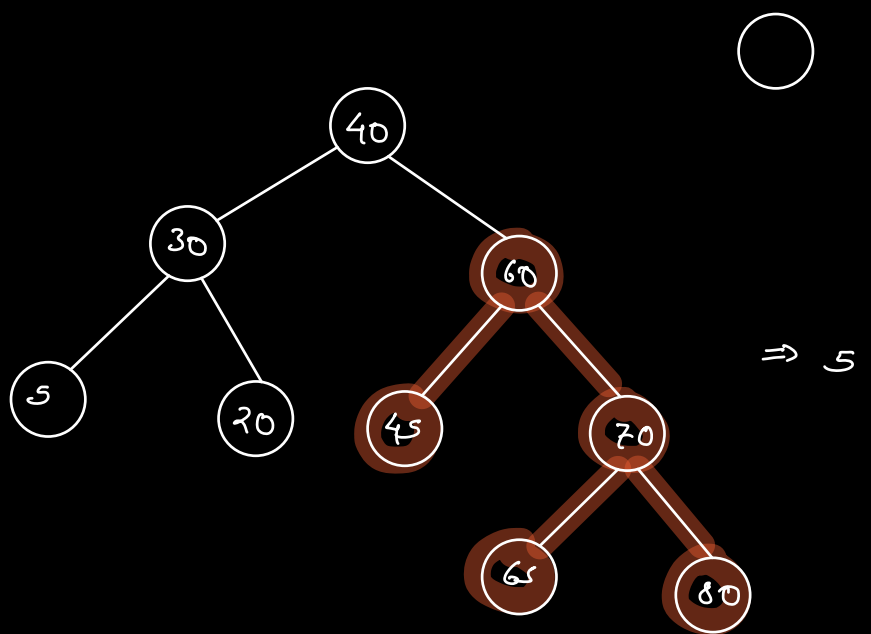

```
TreeInfo isBST(root) {
    if (root == null)
        ret new TreeInfo(+∞ (min), -∞ (max), true);
    TreeInfo left = isBST(root.left);
    TreeInfo right = isBST(root.right);
    if (left.isBST && right.isBST
        && root.val >= left.max && root.val < right.min) {
        ret new TreeInfo(min(root.val, left.min, right.min),
                         max(root.val, left.max, right.max), true)
    }
    ret new TreeInfo(min(root.val, left.min, right.min),
                     max(root.val, left.max, right.max), false);
}
```

Q Given a BT. Return the size of max BST sub-tree inside the BT.

→ No. of nodes



⇒ 3

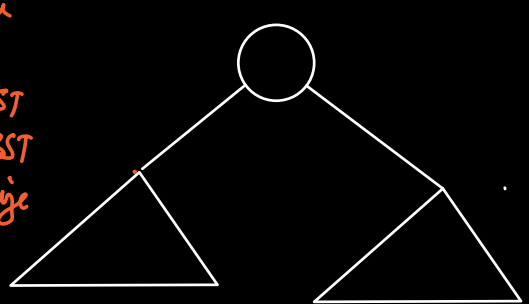


⇒ 5

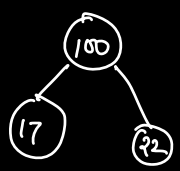
5 30 20 40 45 60 65 70 80

```
Class TreeInfo {
    boolean isBST;
    int max;
    int min
    int maxBST;
    int CurSize;
}
```

max
min
isBST
maxBST
CurSize

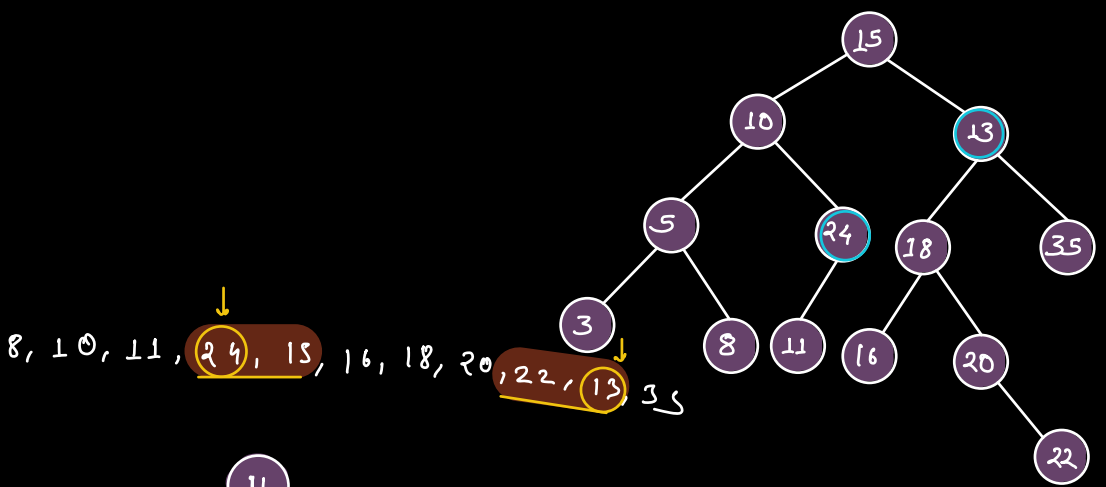


```
TreeInfo maxBST(root) {

    if (root == null) {
        // HW
    }

    TreeInfo l = maxBST(root.left);
    TreeInfo r = maxBST(root.right);
    // if subTree rooted at root is a BST
    if(l.isBST && r.isBST && l.max < root.val && r.min > root.val) {

        ret new TreeInfo(true, r.max, l.min, l.curSize + r.curSize + 1);

    }
    else

        ret new TreeInfo(false, max(l.max, r.max, root.val), min(l.min, r.min, root.val),

        max(l.maxBST, r.maxBST), l.curSize + r.curSize + 1);

}
```

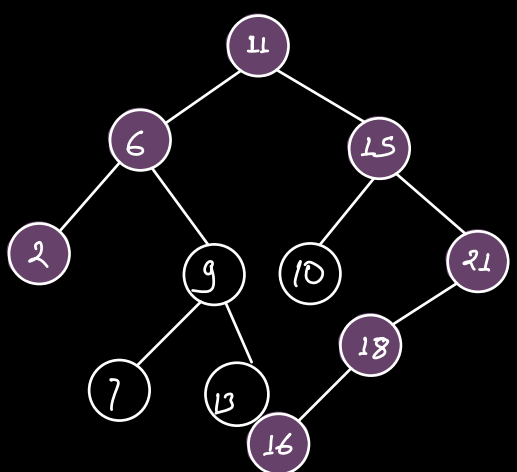


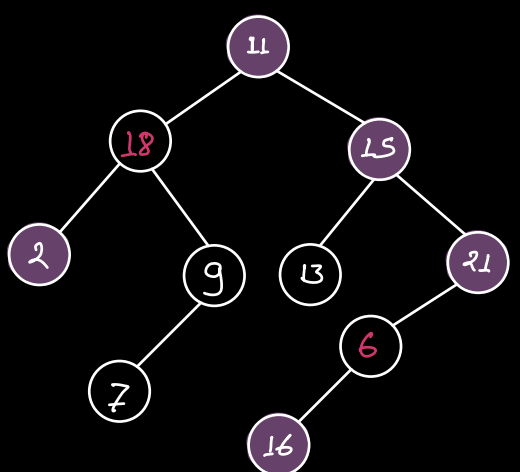

```
class TreeInfo {
    boolean isBST;
    int max;
    int min;
    int maxBST;
    int curSize
}
```

Q Given a BST. Where two nodes have been swapped. Fix it.



3, 5, 8, 10, 11, 24, 15, 16, 18, 20, 22, 13, 35





2, 6, 7, 9, 13, 11, 10, 15, 16, 18, 21

2, 18, 7, 9, 11, 13, 15, 16, 6, 21

m1 m2 m2

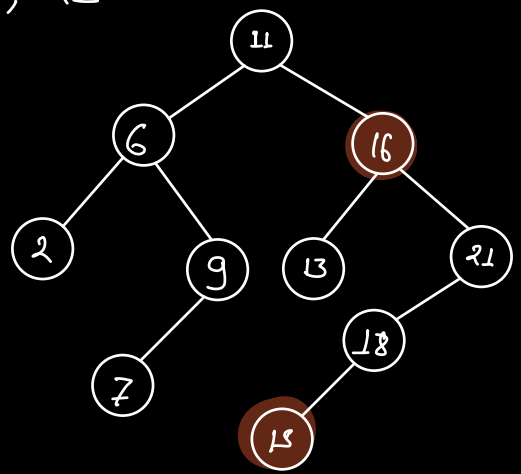
2, 6, 7, 9, 18, 13, 15, 16, 11, 21

n_1 n_2

2, 6, 7, 9, 11, 13, 16, 15, 18, 21

n_1, n_2 > null

```
if( cur.val < prev.val){
    if(n1 == null){
        n1 = prev;
        n2 = cur;
    }
    else{
        n2 = cur;
    }
}
```



```
n1 = null;
n2 = null;
prev = null,
```
Class var

```
void recover(root){
    if(root == null)
        return;
    recover(root.left);
    if(prev != null){
        if(root.val < prev.val){
            if(n1 == null){
                n1 = prev;
                n2 = root;
            }
            else{
                n2 = root;
            }
        }
    }
    prev = root;
    recover(root.right);
}
```

// Assumption
recover(node) →
Fixed the anamolies & update n1 & n2 for the subtree rooted at node.

TC : O(N)
SC : O(N)