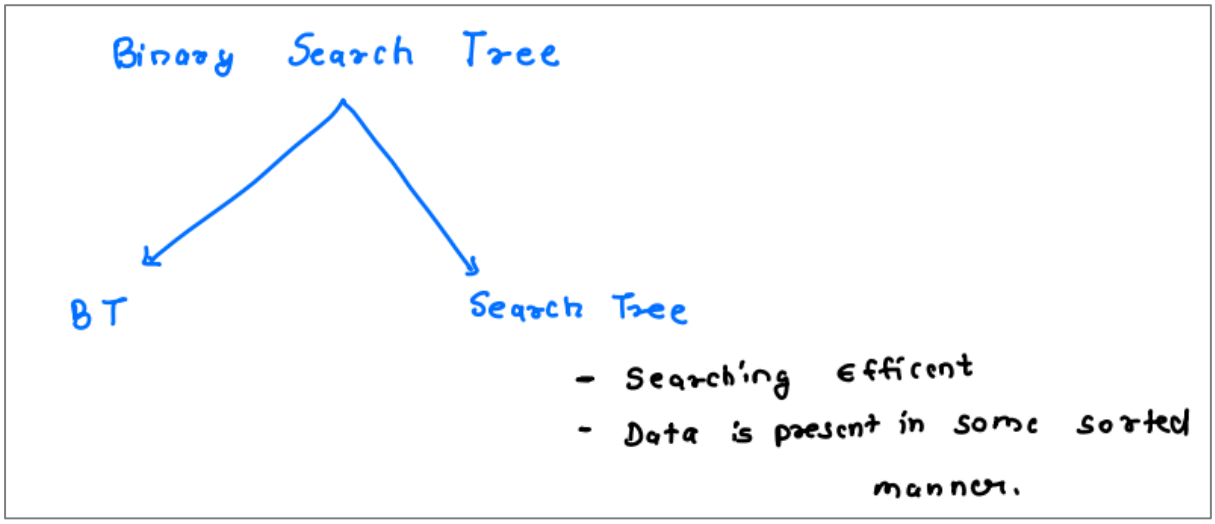
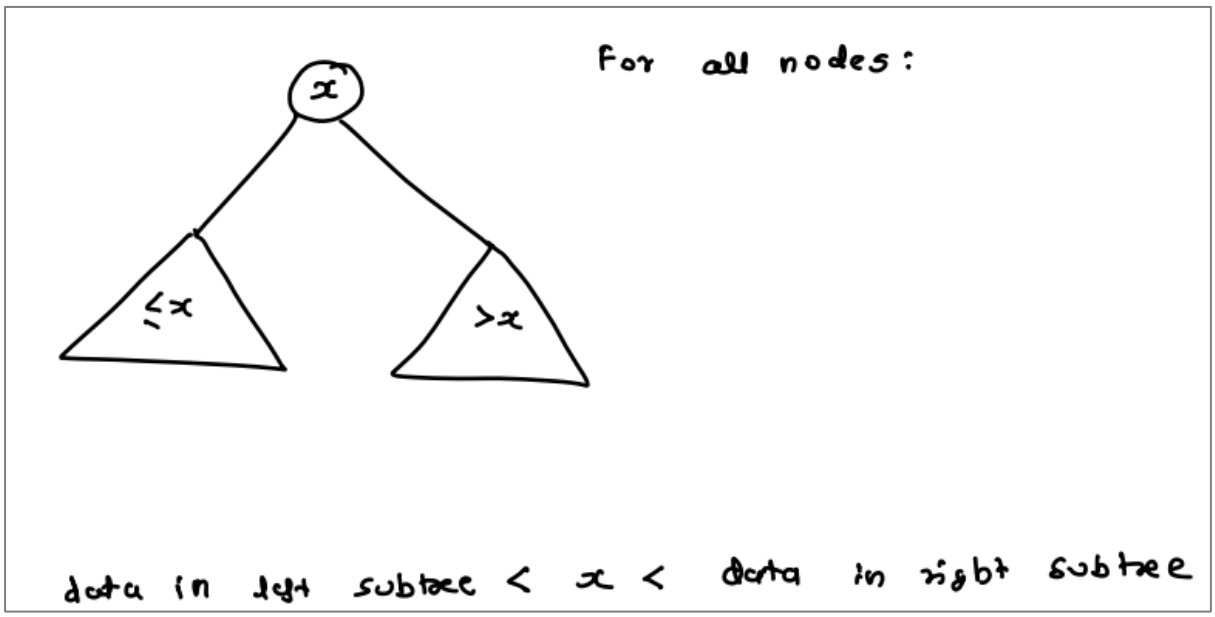
Trees 3: BST (Binary Search Tree)

Search tree → data is present in sorted order



In BST, for all nodes → data in left subtree < all data in right subtree



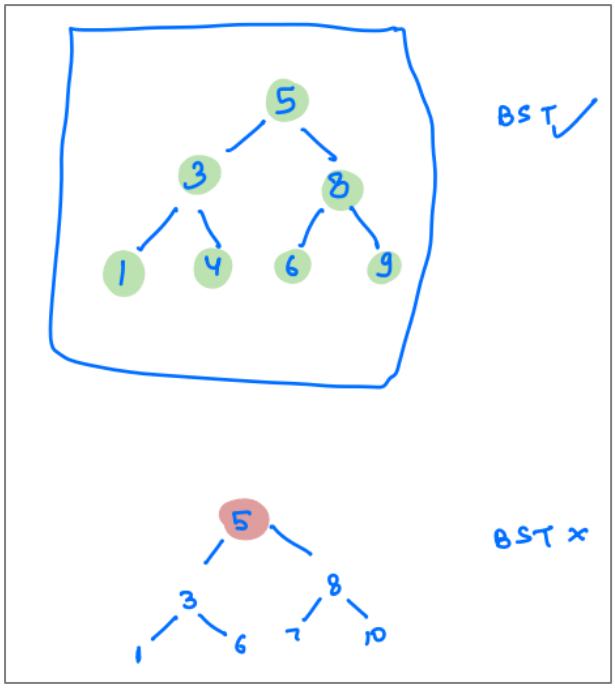
Example of BST:

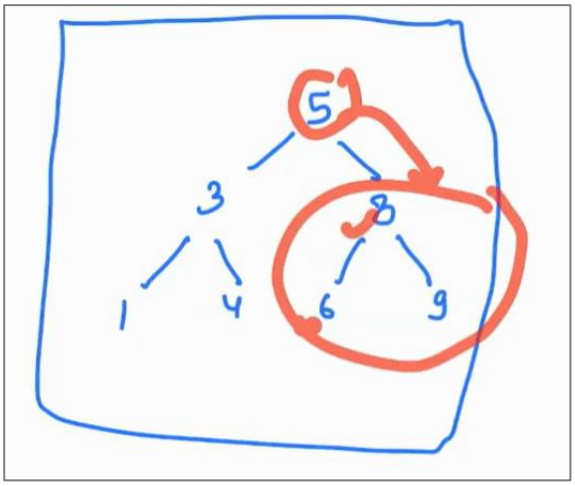
For 5, entire left subtree is smaller 3, 1, 4 and entire right subtree is larger 8,6,9

For 3, entire left subtree is smaller 1 and entire right subtree is larger 4.

For 8, entire left subtree is smaller 6 and entire right subtree is larger 9

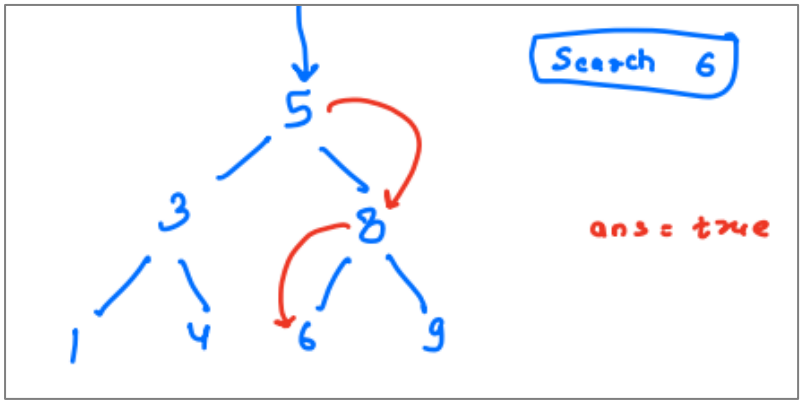
so this is a BST



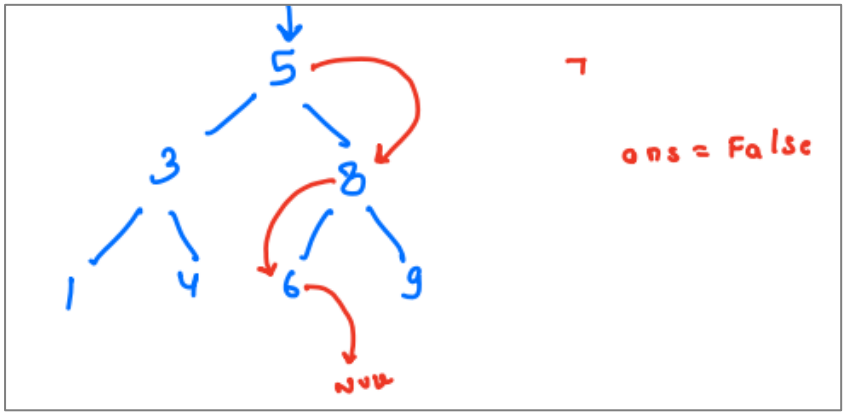


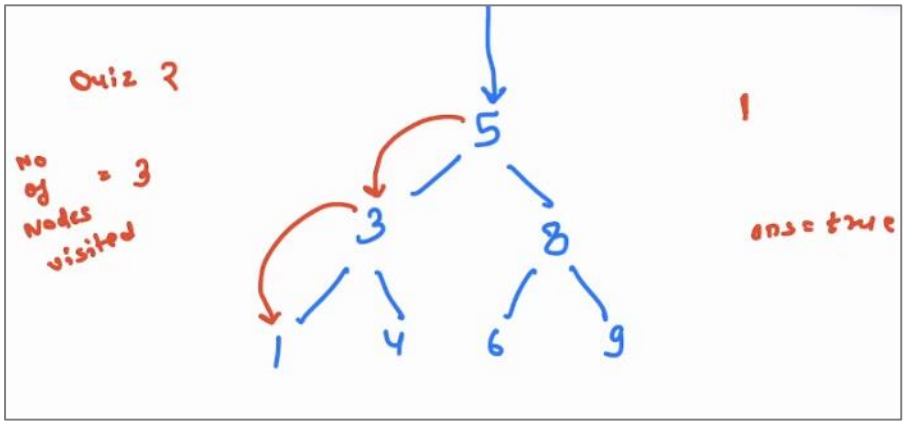
Searching

now search 6. If 6>5(root node) we can jump to right side since the data is sorted in BST. Now you are at 8, you want to search 6, if 6 is present where it would be in the left of 8 since 6<8 and we found 6.



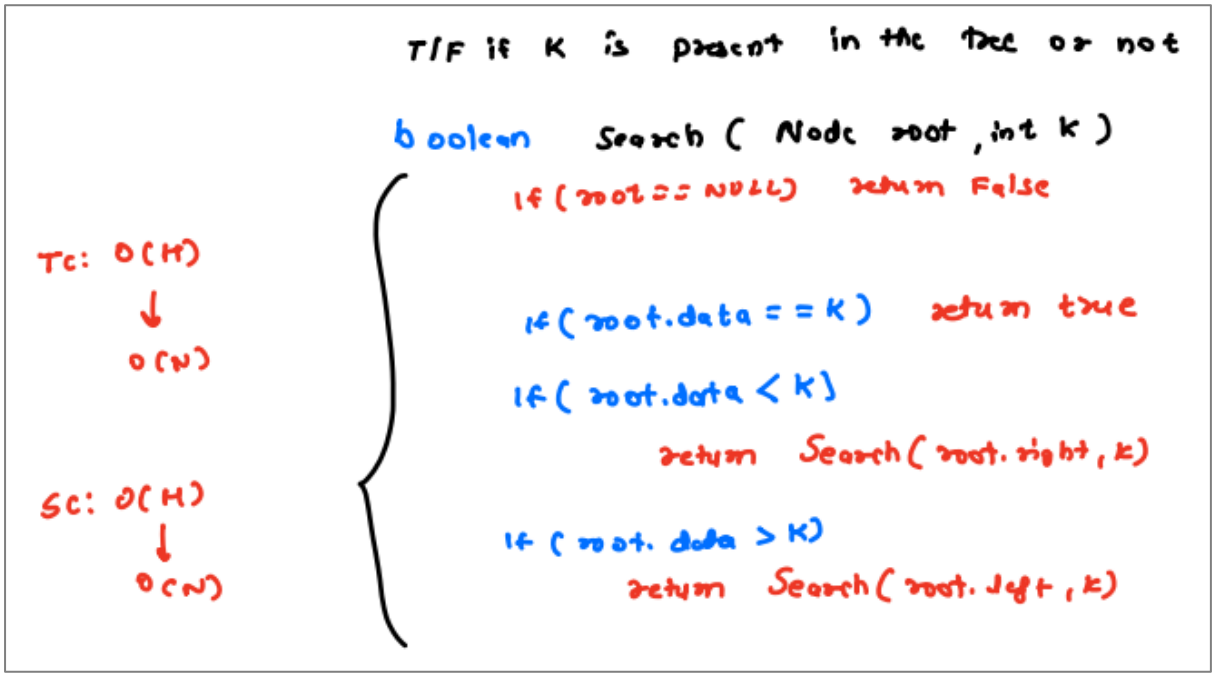
For 7 we found null after 6 so 7 is not there in BST

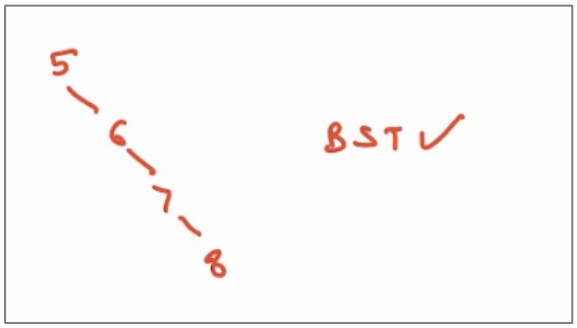




Can implement using iterative & recursion. But here we are using recursion

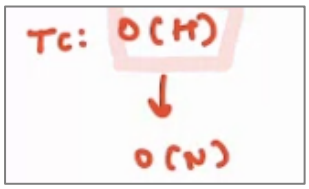
Code for Search





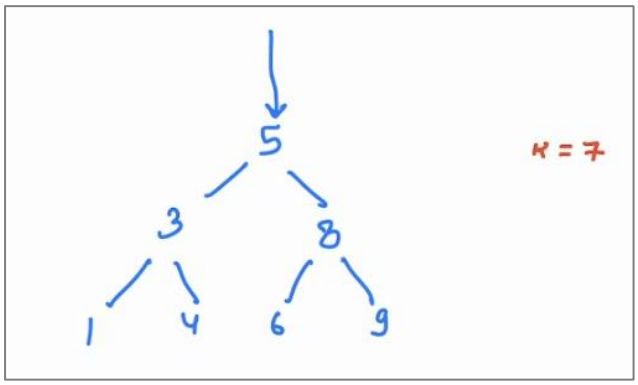
This is a BST but it is not balanced

With each iteration, I am going one level down so TC would be height of the tree



SC = O(H) → O(N)

Insertion in BST

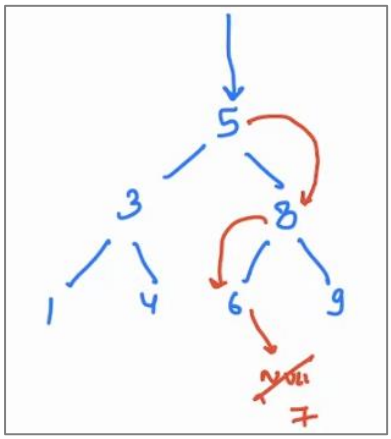


And k =7 for insertion. I am standing at 5 where should i go → left or right? Since 5<7 ➔ go right

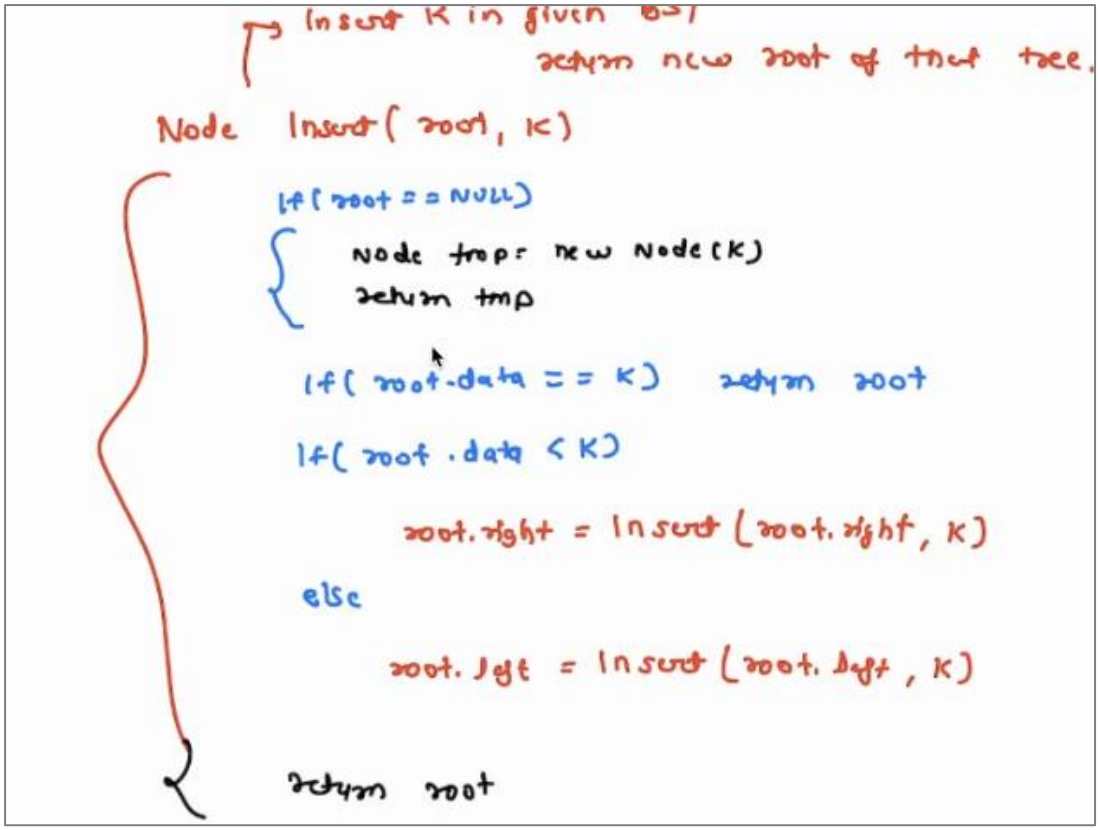
At 8, since 8>7 → go left

At 6 → since 6<7 → go right → null

Since we encounter null → insert 7



NoTe: if a BST becomes unbalanced due to insertion → tree will balance itself k/a self balancing tree or AVL tree by internal movement. We will always insert at leaf node

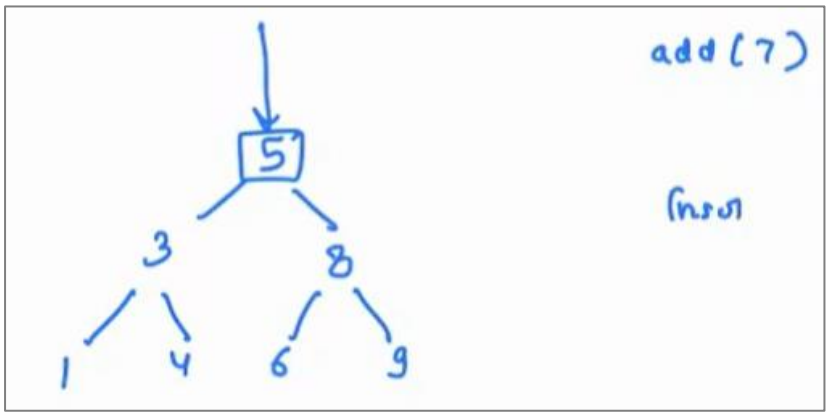


if(root.data ==k) then don't insert k

insert function will insert k and will give the root of right subtree

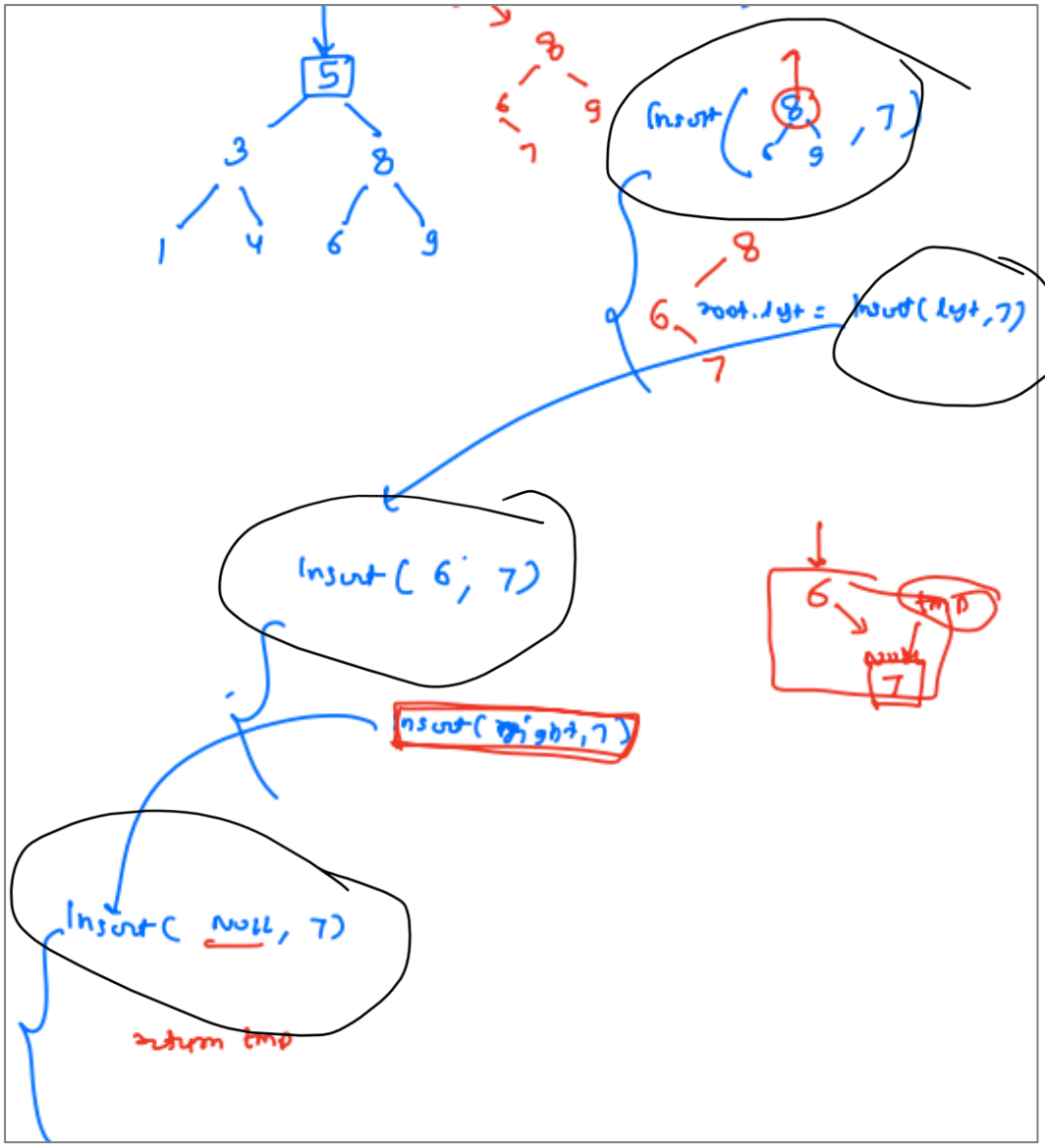
In place of root in insert function, we are giving root of right subtree as complete tree

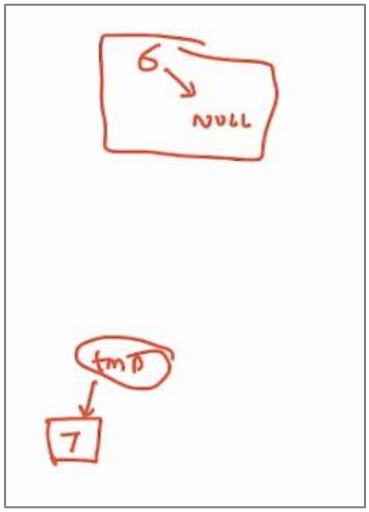
Dry Run



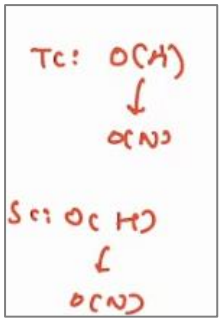
We have to add 7, currently we are at 5 root node. We got to 8 subtree → go left

We get 6; since 6 >7





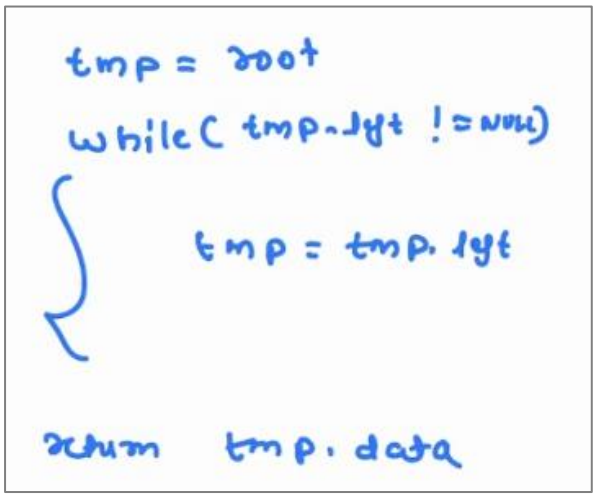
I will get the reference of new node tmp in my recursive calls and it will link my 6 to 7. Since we are going one level down



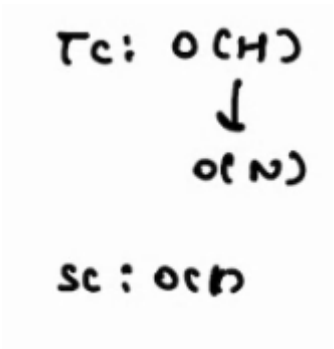
SC:-O(H) coz we are inserting one element at a time. Before inserting another element, that space will be freed up.

Problem Statement 1- Find smallest element in BST

Given a BST, find smallest element



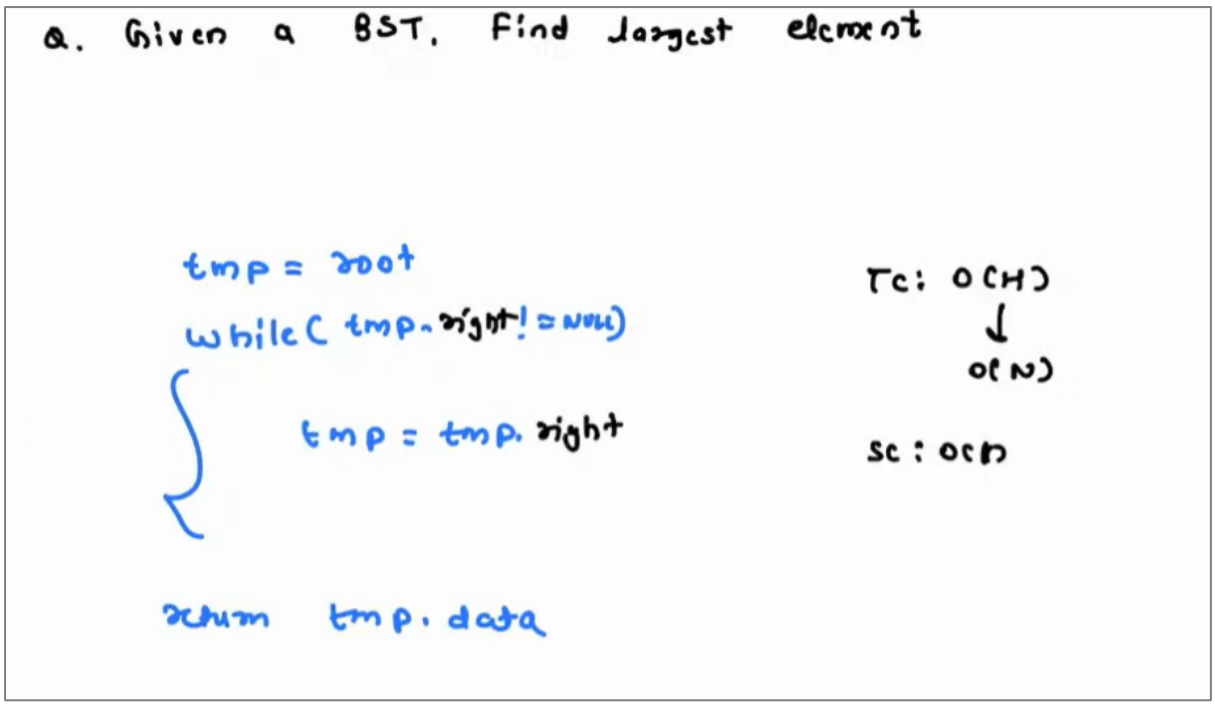
You will go left left to find the smallest element and when while loop completes, before reaching null print the element,



Problem Statement 2- Find largest element in BST

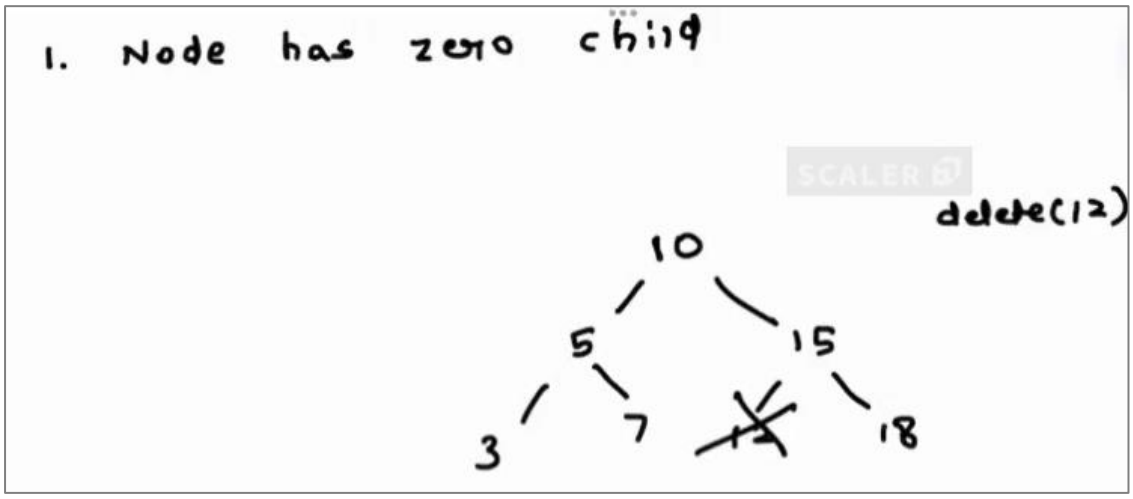
Given a BST, find largest element

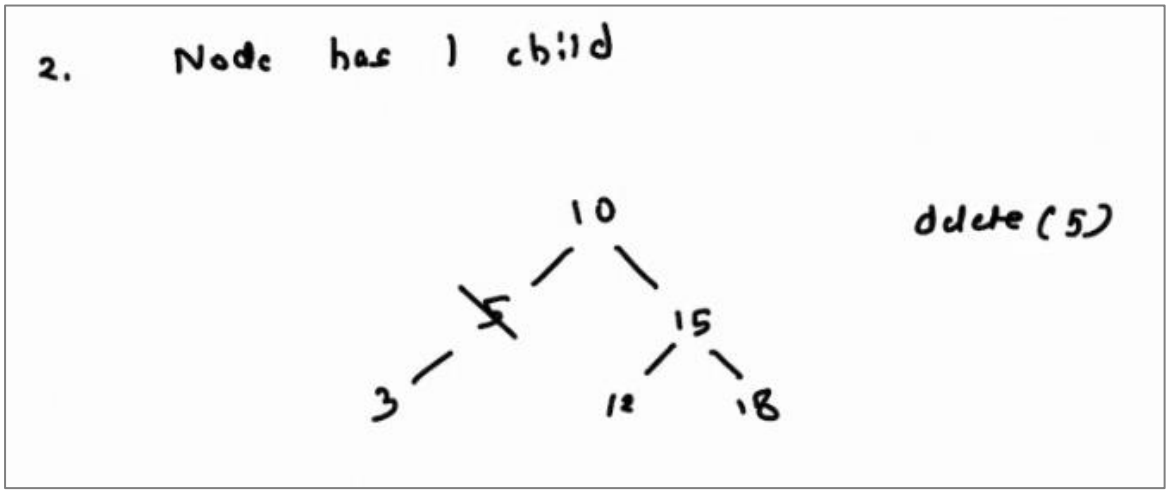
For largest element, I'll keep going right, right until we reach the largest element.



Deletion in BST

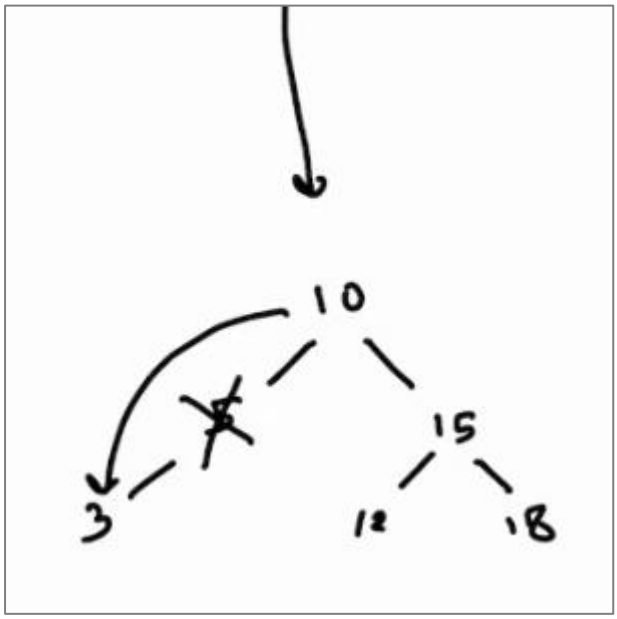
If the node has no child, just delete the node



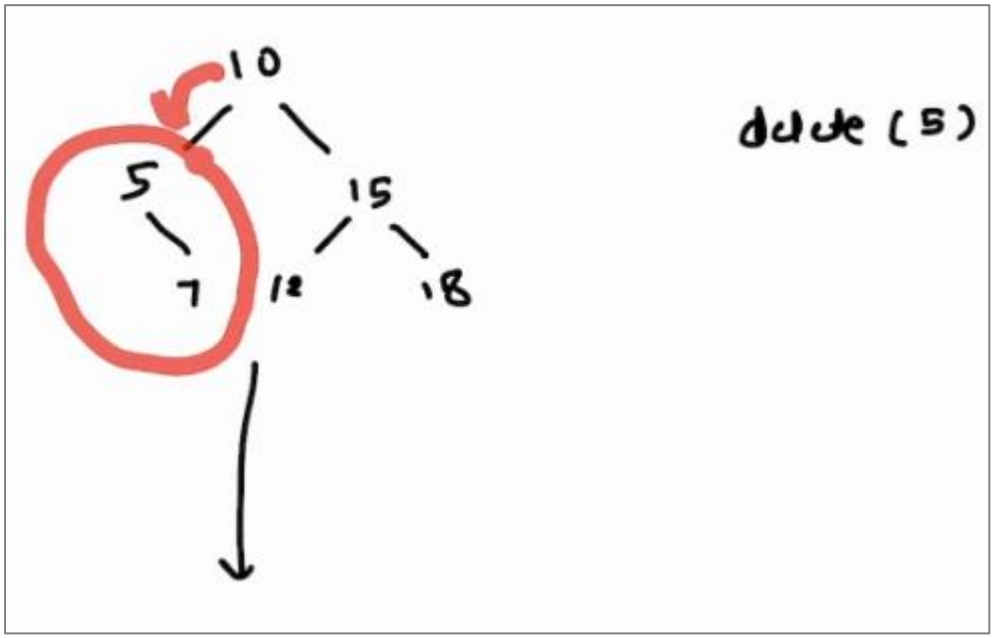


If node we want to delete which has 1 child.

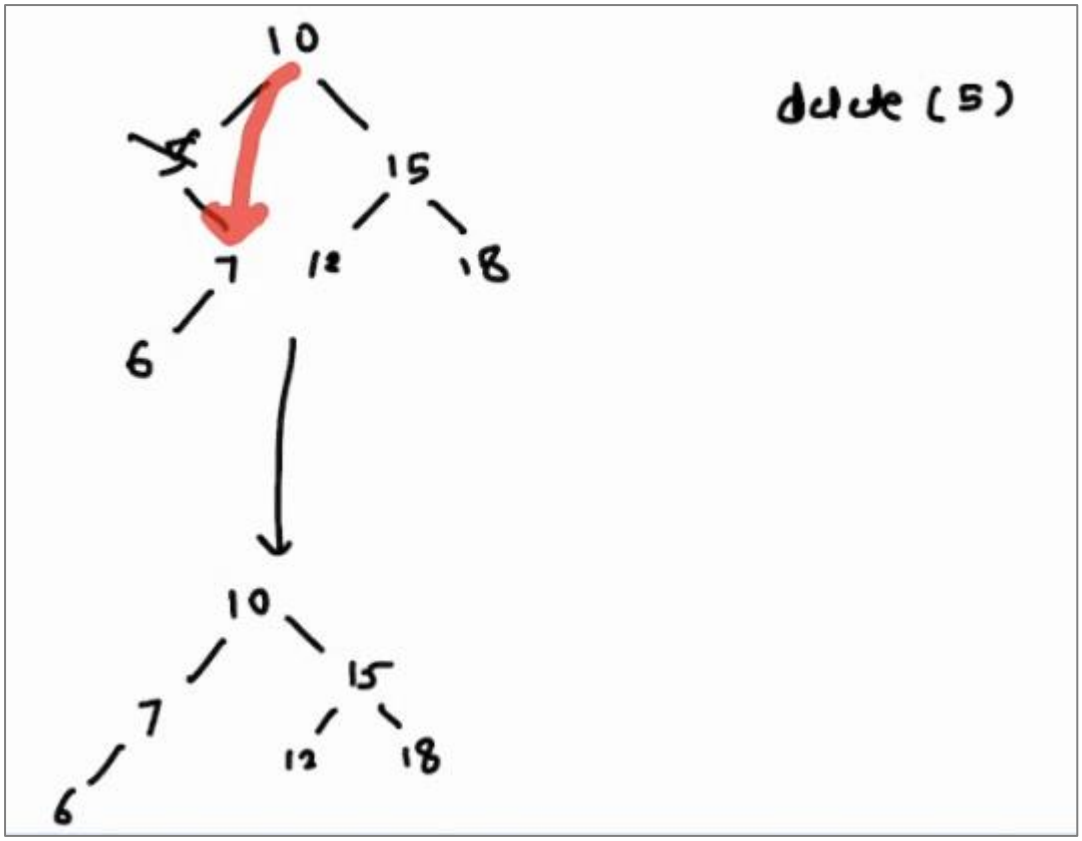
Now if the node = 5 has a child. 3 lost a parent & 10 lost a child.
Connect 3 with 10.



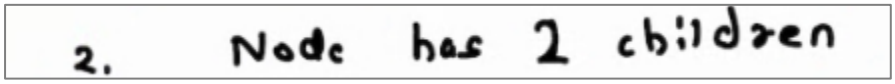
Now if we delete 5 here,
delete 5 → 10 wants a left child and 7 doesn't care left or right
attach 7 as left child



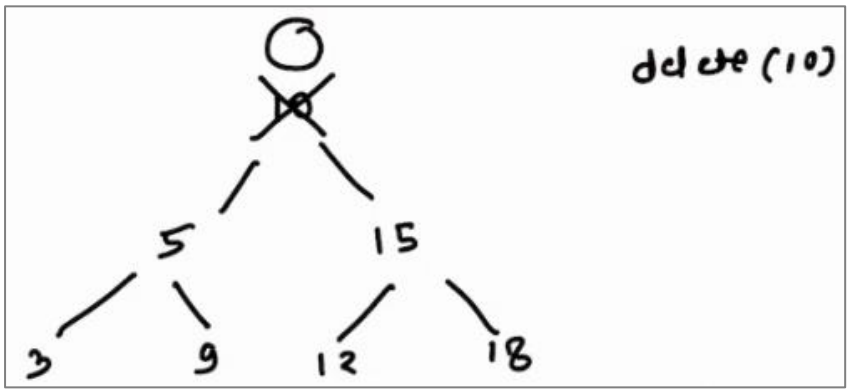
Another scenario, here left of 7 remains left of 7.



Just tell 10 to start referencing 7, nothing changes



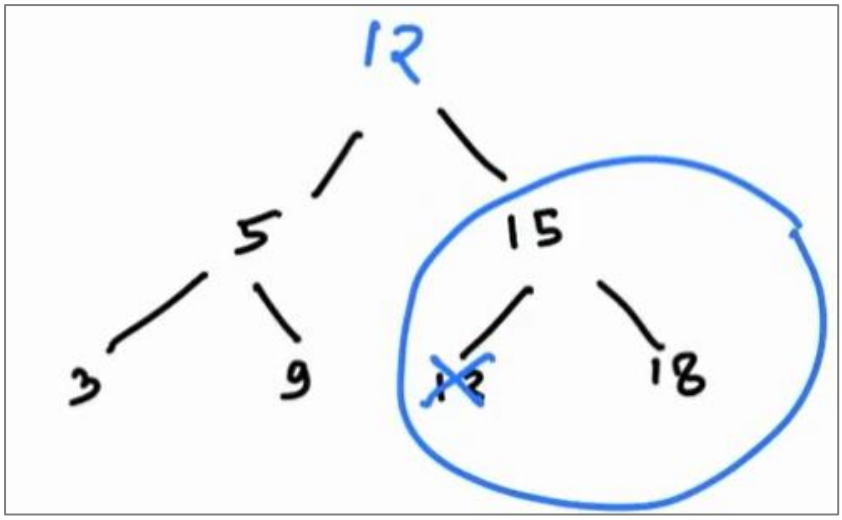
CASE3: **Delete root node**



Now someone has to take the root node position else the tree will be fell apart

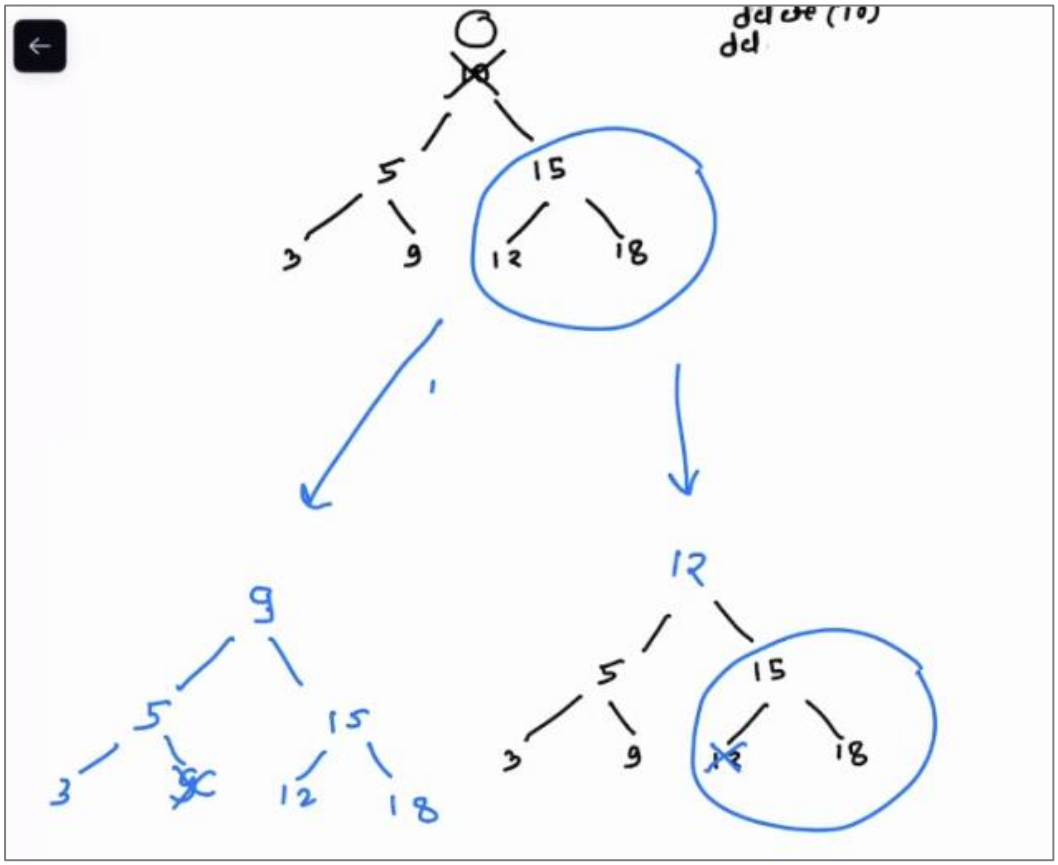
Lets say we want someone from the right side to come and take that position such that every element on the right side should be larger than it

So, I should pick the smallest value so that right side > than root element then only the new root value will be smaller than values on the right side and larger than the values on the left side

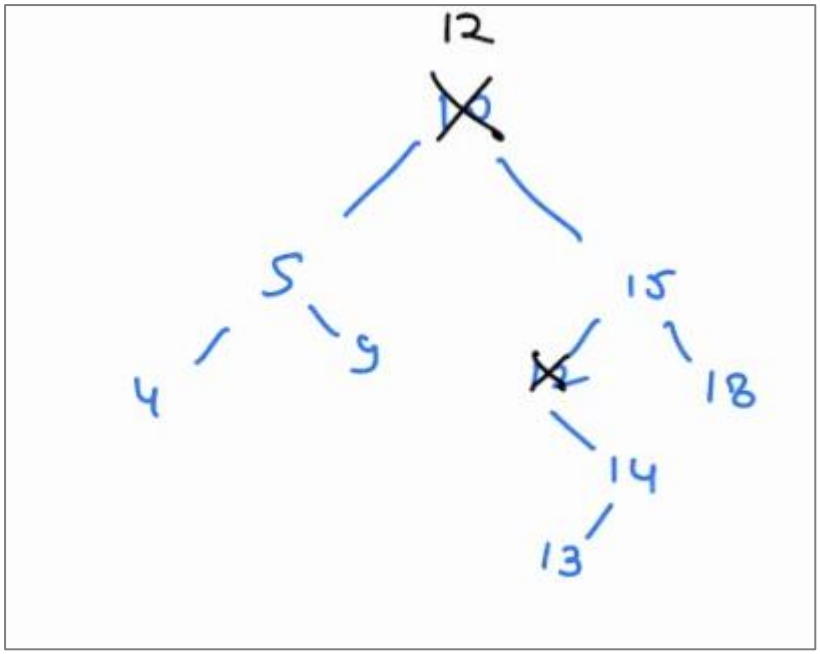


Another arrangement is pick something from the left side → such that every element in the left side would be smaller than that.

So we will pick the largest number from the left side



Now there is one issue with this, lets take another example



you went left → left until left is not null, here we found 12 but we will add 12 at root but at the same time delete 12 at its first position.

But what about 14 & 13 which might require to perform case 1 or case 2 where I have no or 1 child but not 2 child

If 12 have 2 child, then it would not be the smallest element so here we will not get case 3 again.

```
if ( root.data < K)
```

If root< K → I will ask my right subtree to delete ans insert the smallest element on right side

```
else if ( root.data > K)
        root.left = delete ( root.left , K)
```

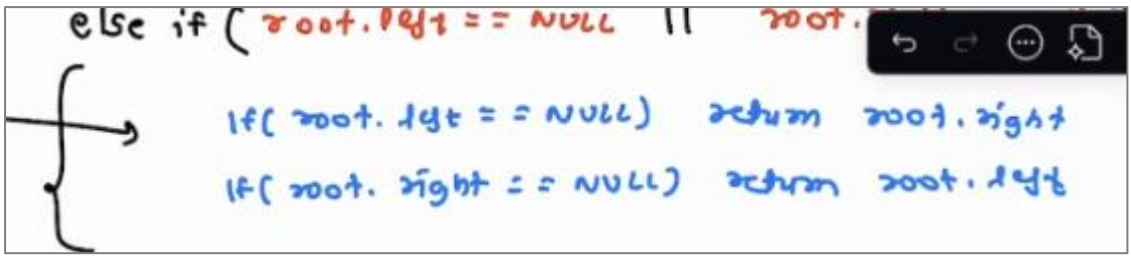
Case1

```
if( root.left == NULL && root.right == NULL)
{   return NULL
```

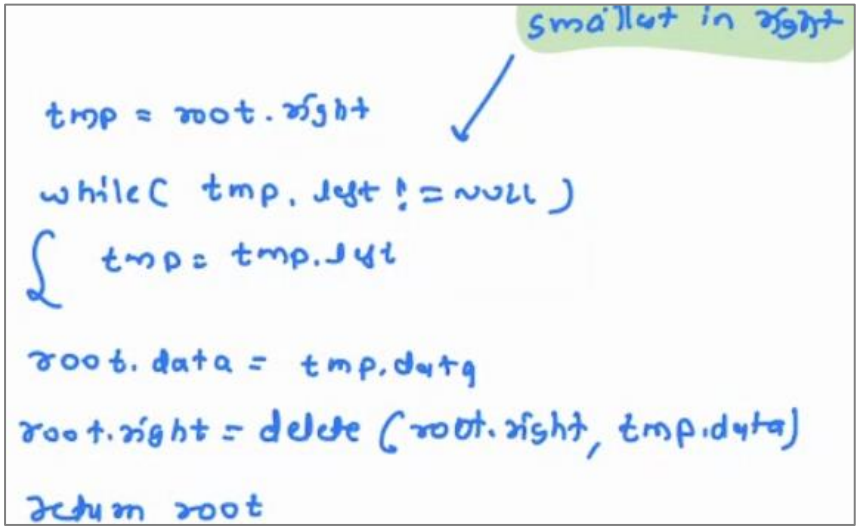
Case 2

If left side is null → right child is the root

If right side is null → left child is the root

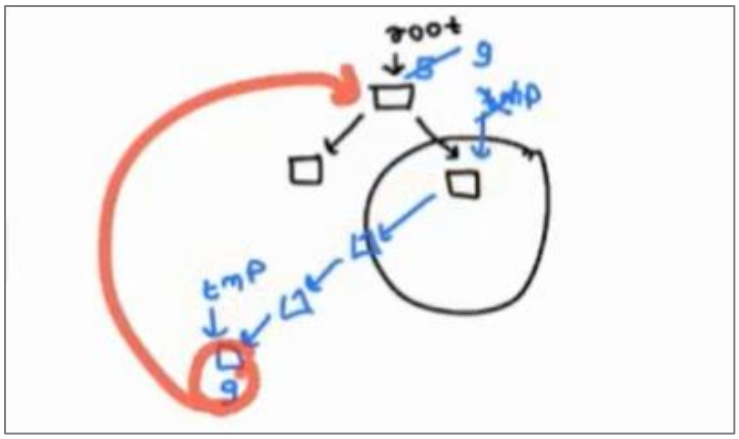


Case 3: When a node has two child



I choose smallest element on the right side to be replaced as root node. tmp = root.right

So for smallest element I go left → left → left until I get smallest element → through while loop



Assign smallest element as root node → root.data = tmp.data

Now from my right subtree, delete the smallest element in the first position

Calling the recursive call to delete the element

// delete K from tree & return new root

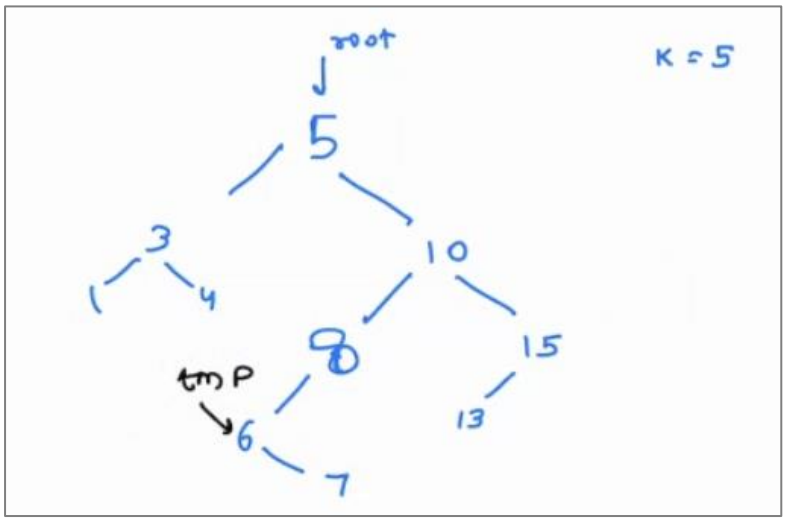
```
Node delete ( Root, K)
    if (root == NULL) return NULL

    if ( root.data < K)
    {       root.right = delete ( root.right, K)
    else if ( root.data > K)
    {       root.left = delete ( root.left , K)
    else
        if( root.left == NULL && root.right == NULL)
C1          return NULL
        else if ( root.left == NULL || root.right == NULL)
C2          if( root.left == NULL) return root.right
            if( root.right == NULL) return root.left
        else
C3          tmp = root.right          (smallest in right)
            while( tmp.left != NULL )
            {   tmp = tmp.left
            root.data = tmp.data
            root.right = delete (root.right, tmp.data)
            return root
```

Tc: O(H) → O(N)

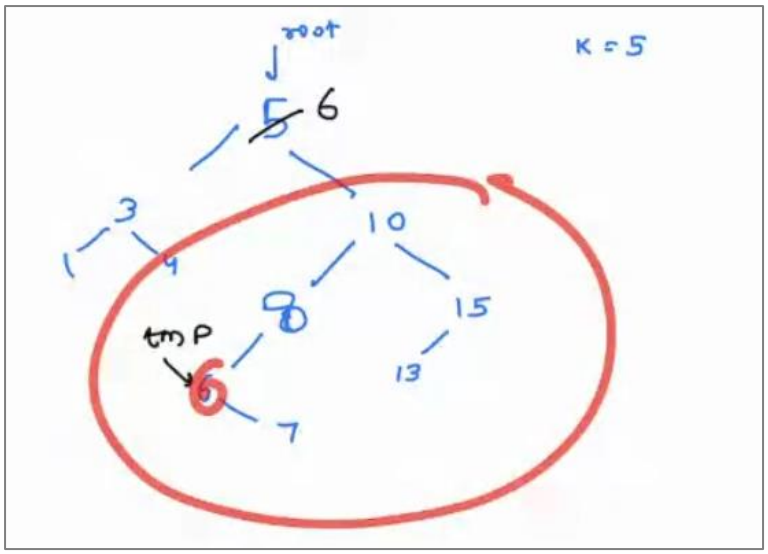
Sc: O(H)

root

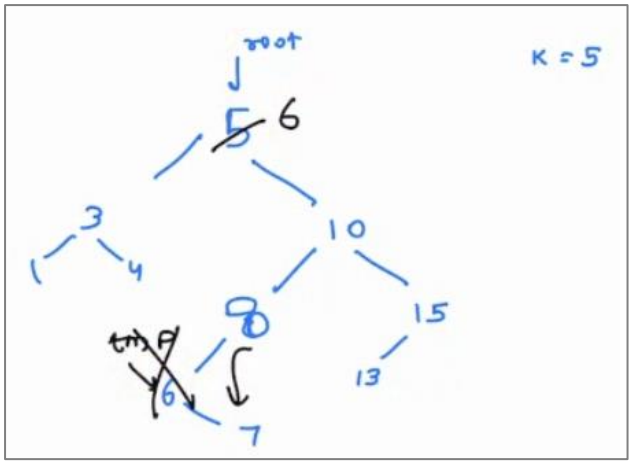
TC: Since we are going down , down so in worst case it will be height of the tree O(h)
Example: Here if K = 5 and we have to delete 5 element. Then I can either choose to take largest element in left subtree or smallest element in right subtree to be the root node. I choose smallest element in the right subtree, so when I go left → left, I found 6 to be the perfect candidate to become root node.

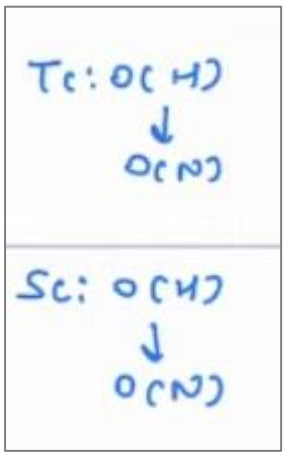


Put 6 as root element and delete 6 from below



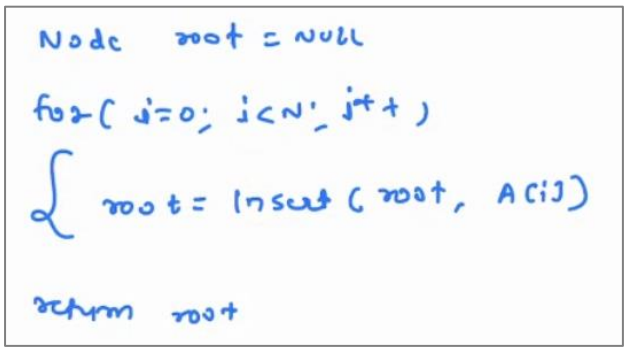
Using my recursive call (Case 2) where my node element to be deleted has only once child. That recursive will automatically assign 8 as the parent of 7.



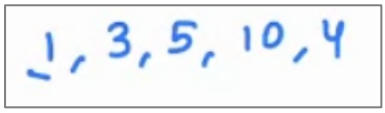


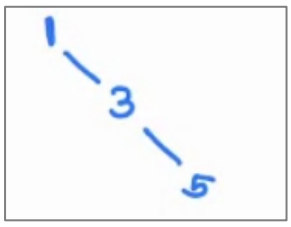
Problem Statement 3- Construct a BST

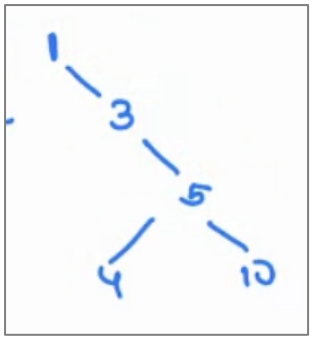
Given N nodes. Construct a BST from it



Example:

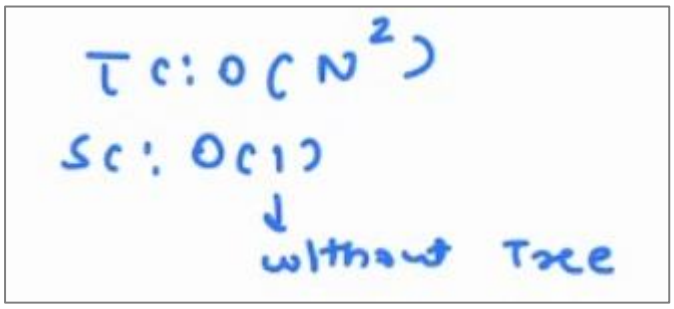






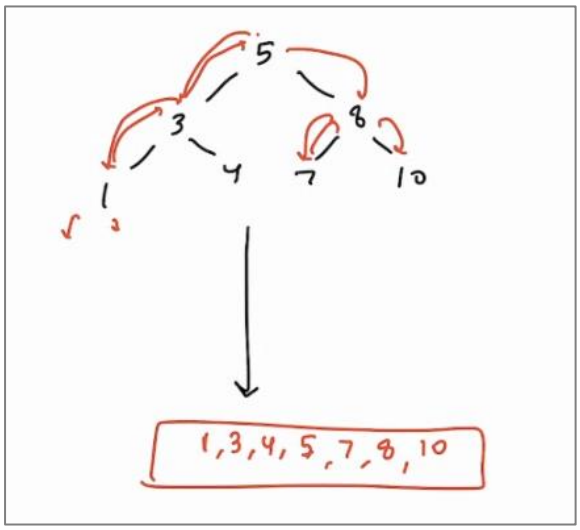
insert 1 → pick 3 → insert on the right since 3 > 1.
Pick 5 → insert on the right since 3<5
Pick 10 → insert on the right since 10> 5
Pick 4 → insert left of 5 since 4<5.

SC:-O(H) coz we are inserting one element at a time. Before inserting another element, that space will be freed up.

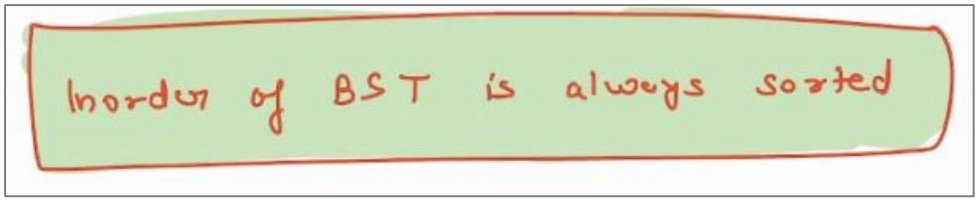


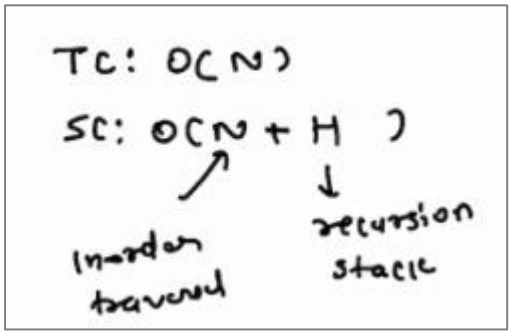
Problem Statement- Check if it is BST

Given a BT. Check if it is BST or not

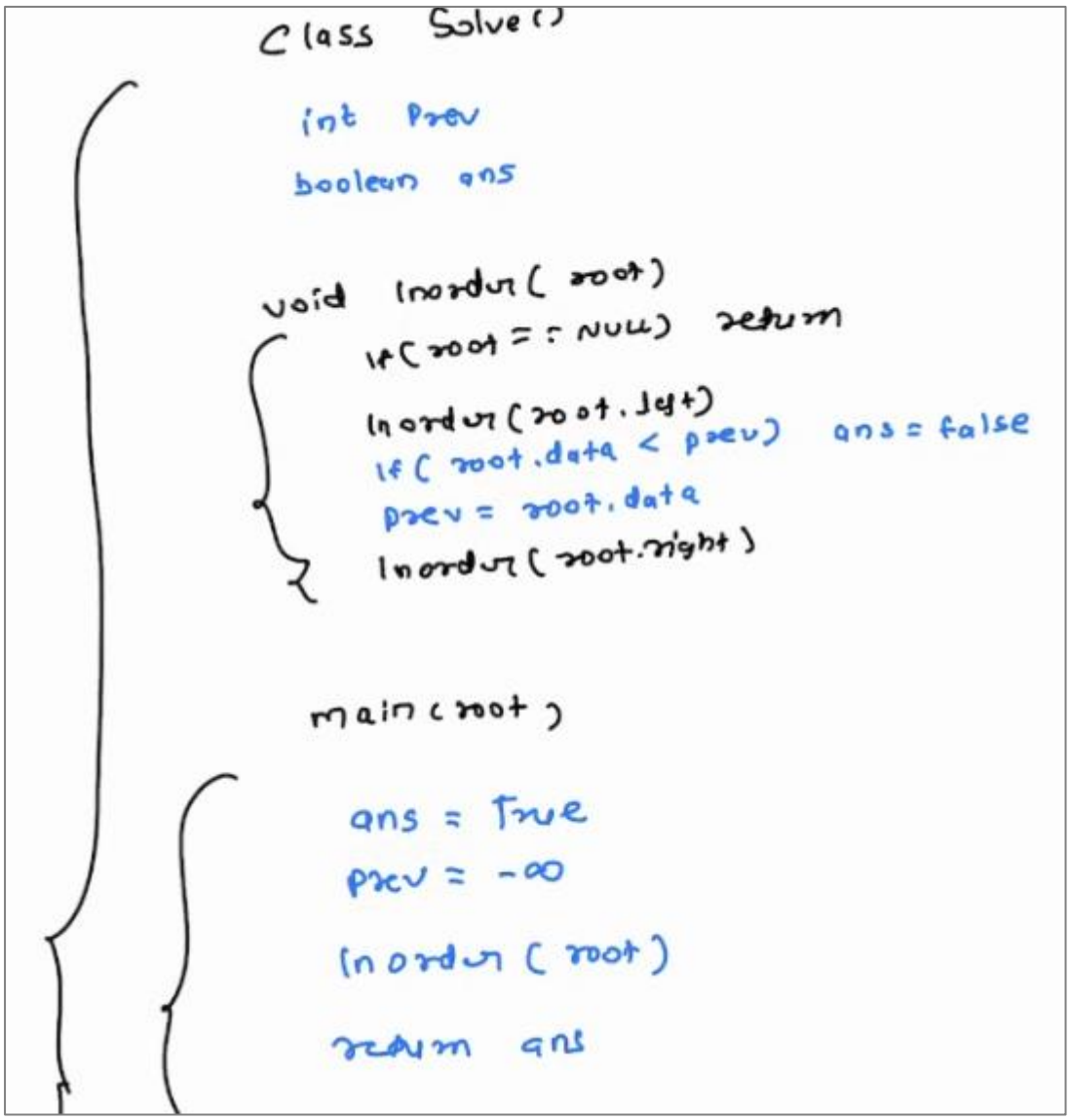


Do inorder traversal and check whether the inorder traversal is sorted or not → because in BST inorder traversal is always sorted



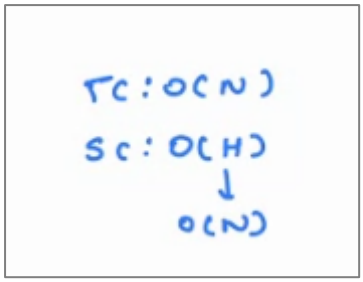


SC = O(N) since we are storing inorder traversal



TC = O(N)

SC = O(N)



Problem Statement 1 - Valid Binary Search Tree

Problem Description

You are given a binary tree represented by root **A**. You need to check if it is a Binary Search Tree or not.

Assume a BST is defined as follows:

1) The left subtree of a node contains only nodes with keys less than the node's key.

2) The right subtree of a node contains only nodes with keys greater than the node's key.

3) Both the left and right subtrees must also be binary search trees.

Problem Constraints

1 <= Number of nodes in binary tree <= 10^5

0 <= node values <= 2^{32}-1

Input Format

First and only argument is head of the binary tree A.

Output Format

Return 0 if false and 1 if true.

Example Input

Input 1:

```
  1
 / \
2   3
```

Input 2:

```
 2
/ \
1   3
```

Example Output

Output 1: 0

Output 2: 1

Example Explanation

Explanation 1:

2 is not less than 1 but is in left subtree of 1.

Explanation 2:

Satisfies all conditions.

Actual Code

```
public class Solution {
```

```
  private int prev =Integer.MIN_VALUE;
  private boolean ans = true;

  public int isValidBST(TreeNode A) {
     inorder(A);
     return ans?1:0;
  }
  private void inorder(TreeNode A){
     if(A == null) return;
     inorder(A.left);
      if(A.val < prev) ans = false;
      prev = A.val;
     inorder(A.right);
  }
}
```

Problem Statement -2 Sorted Array To Balanced BST

Problem Description

Given an array where elements are sorted in ascending order, convert it to a height Balanced Binary Search Tree (BBST).

Balanced tree : a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of every node never differ by more than 1.

Problem Constraints

1 <= length of array <= 100000

Input Format

First argument is an integer array A.

Output Format

Return a root node of the Binary Search Tree.

Example Input

Input 1: A : [1, 2, 3]

Input 2: A : [1, 2, 3, 5, 10]

Example Output

Output 1:

```
   2
  / \
 1   3
```

Output 2:

```
    3
   / \
  2   5
 /     \
1       10
```

Example Explanation

Explanation 1: You need to return the root node of the Binary Tree.

Actual Code

```
public class Solution {
  public TreeNode sortedArrayToBST(final int[] A) {
    int start =0;
    int end = A.length-1;

    TreeNode root = construct(A, start, end);
    return root;
  }
  private TreeNode construct(int []A, int start, int end){
    if(start > end) return null;

    int mid = (start+end)/2;
    TreeNode root = new TreeNode(A[mid]);

    root.left = construct(A, start, mid-1);
    root.right = construct(A, mid+1, end);
    return root;
  }
}
```

Problem Statement 3 - Search in BST

Problem Description

Given a Binary Search Tree **A**. Check whether there exists a node with value **B** in the BST.

Problem Constraints

1 <= Number of nodes in binary tree <= 10^5

0 <= B <= 10^6

Input Format

First argument is a root node of the binary tree, A.

Second argument is an integer B.

Output Format

Return 1 if such a node exist and 0 otherwise

Example Input

Input 1:

```
      15
     /  \
```

```
      12     20
     / \    / \
    10  14  16  27
   /
  8

  B = 16
```

Input 2:

```
       8
      / \
     6  21
    / \
   1   7
  B = 9
```

Example Output

Output 1: 1

Output 2: 0

Example Explanation

Explanation 1: Node with value 16 is present.

Explanation 2: There is no node with value 9.

Actual Code

```
public class Solution {
  public int solve(TreeNode A, int B) {
    return search(A, B);
  }
  private int search(TreeNode A, int B){
    if(A == null) return 0;
    if(A.val == B){
      return 1;
    }else if(A.val < B){
      return search(A.right, B);
    }else if(A.val > B){
      return search(A.left, B);
    }
    return 0;
  }
}
```