TREES - 2

- BST Basics
- Searching in BST
- Insertion in BST
- Smallest | Largest element in BST
- Deletion in BST
- Construct Balanced BST from Sorted Array
- Validate BST

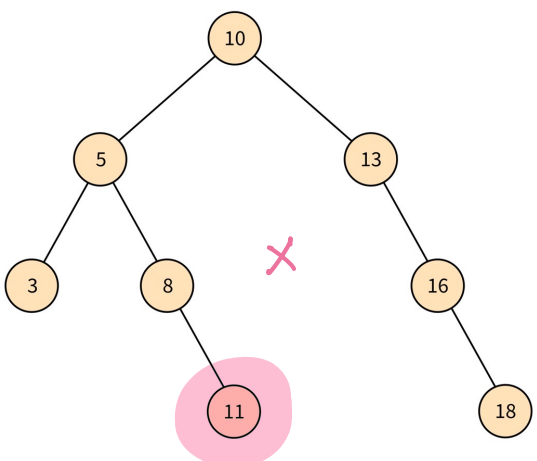
Binary Search Tree [BST]

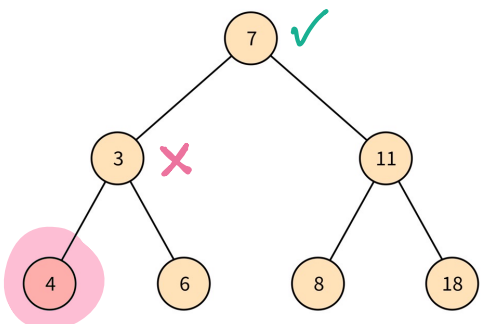
For all Nodes

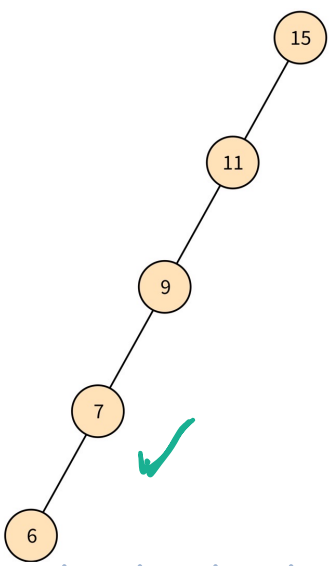
All elements in LST $\leq$ Node $<$ All elements in RST

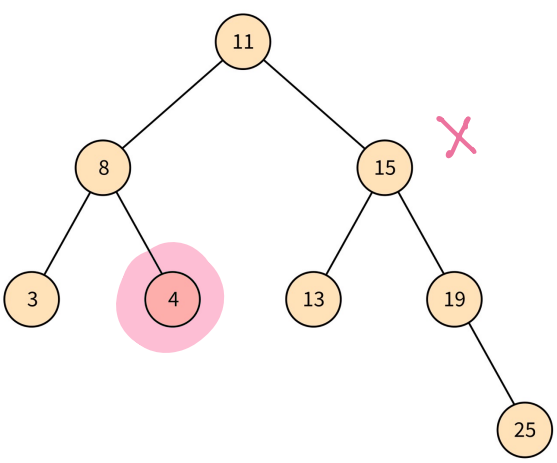
can be on either side

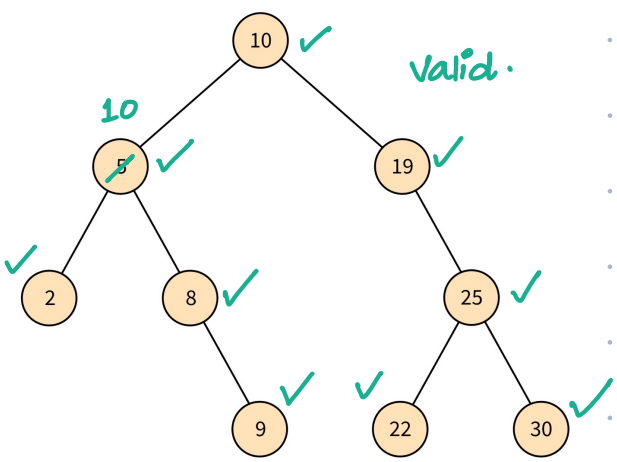
< *Question* > : Check if given tree is BST or not.



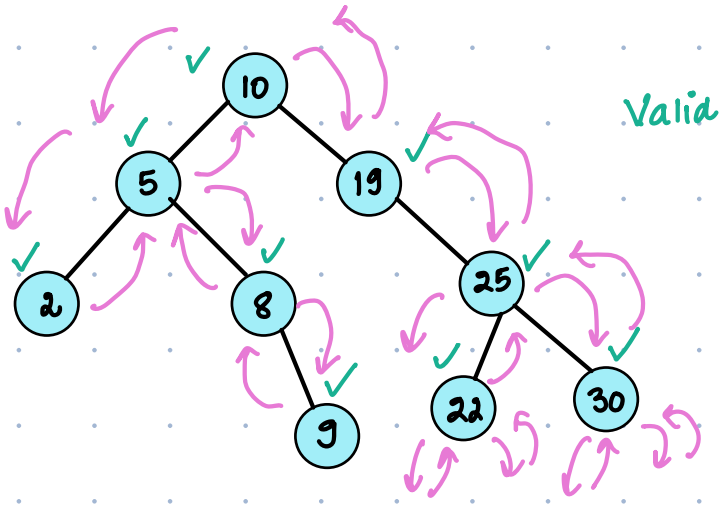








Special property of BST



Valid BST

L N R

Inorder : [2 5 8 9 10 19 22 25 30]

NOTE : For a BST, Inorder Traversal will be SORTED data in INCREASING ORDER.

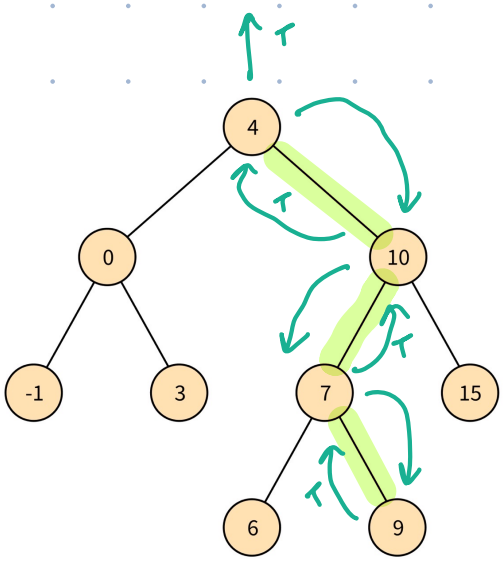
What is a Binary Search Tree (BST)?

23 users have participated

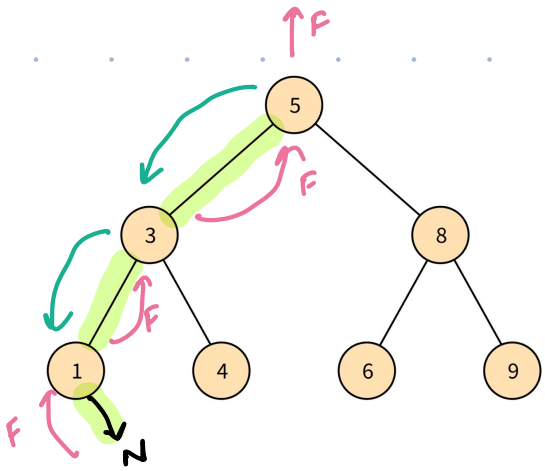
A	A tree with only two nodes	0%
B	A tree where the left child of a node has a value <= the node, and the right child has a value > the node	70% ✗
✓ C	A tree where for a node x, everything on the left has data <= x and on the right > x.	30%
D	A tree that has height log N.	0%

< *Question* > : Search an element K in Binary Search Tree.

K = 9



K = 2

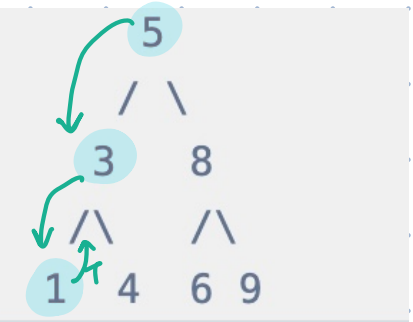


< / > Code

```
boolean search (Node root, int K) {
    if (root == NULL)
        return false;

    if (K == root.val)
        return true;

    else if (K < root.val){    // Search in LST
        return search (root.left, K);
    }
    else{    // Search in RST
        return search (root.right, K);
    }
}
```

T.C : O(H)
S.C : O(H)

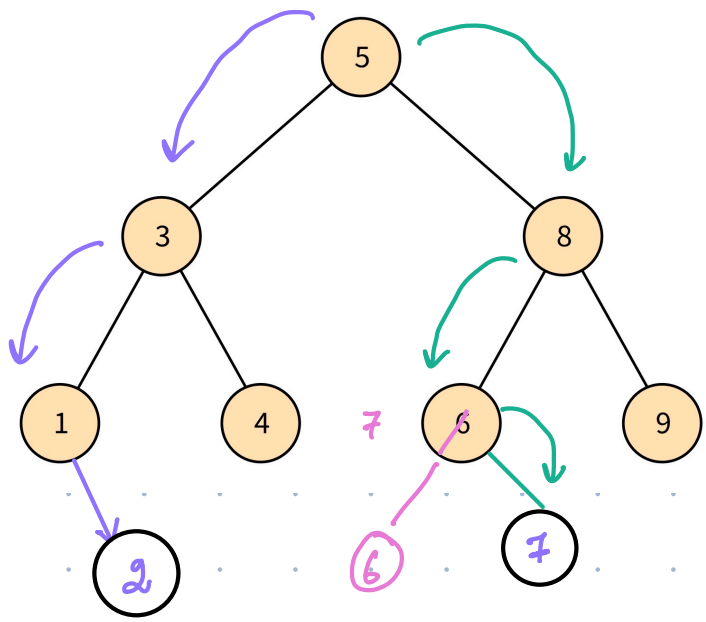
QUIZ



ans = 3

What is the number of nodes you need to visit to find the number 1 in the following BST?

Insertion in B.S.T



insert (7)

insert (2)

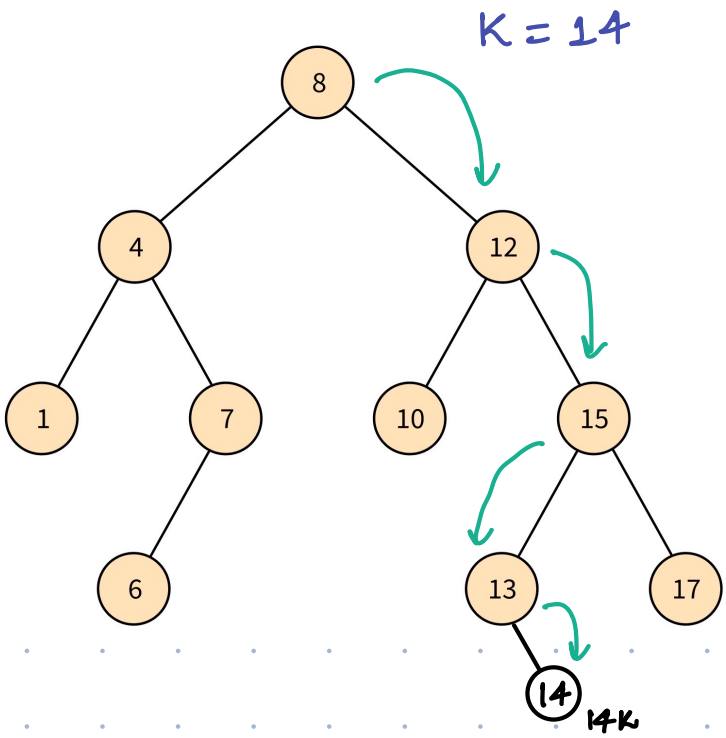
Search for correct position of node & insert it there.

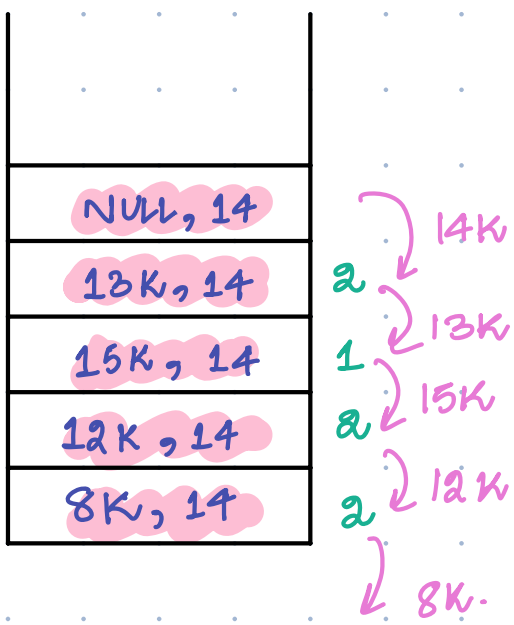
< / > Code

```
Node Insertion (Node root, int K) {
    if (root == NULL)
        return new Node(K);

    if (root.val == K)
        return root;

    else if (K < root.val) {
        root.left = Insertion(root.left, K);   ——— (1)
    }

    else {
        root.right = Insertion(root.right, K);   ——— (2)
    }
}
```

T.C : O(H)
S.C : O(H)

*dry-run





QUIZ

Where does the node with the smallest value resides in a BST?

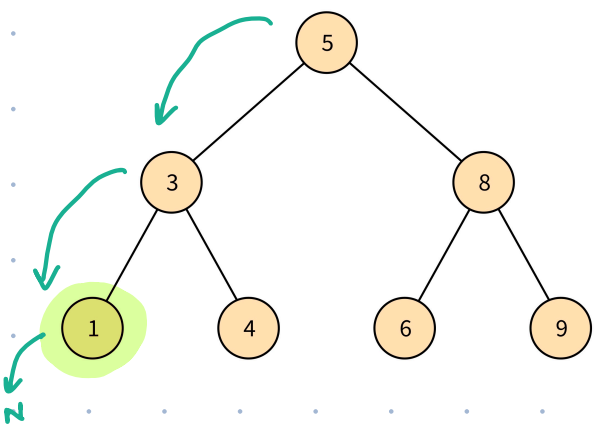
[ ] We keep on going left and we get the smallest one.

[ ] Depends on the tree.

[ ] We keep on going right and we get the smallest one.

[ ] The root node.

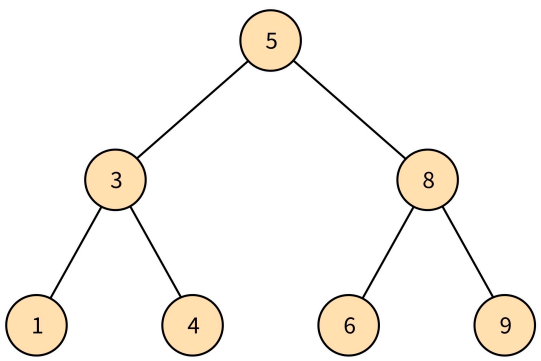
Find the Smallest Element in B.S.T




```
curr = root;
while (curr.left != NULL) {
    curr = curr.left;
}
return curr.val;
```

Keep on going left till there is nothing left on left side.

Find the Largest Element in B.S.T



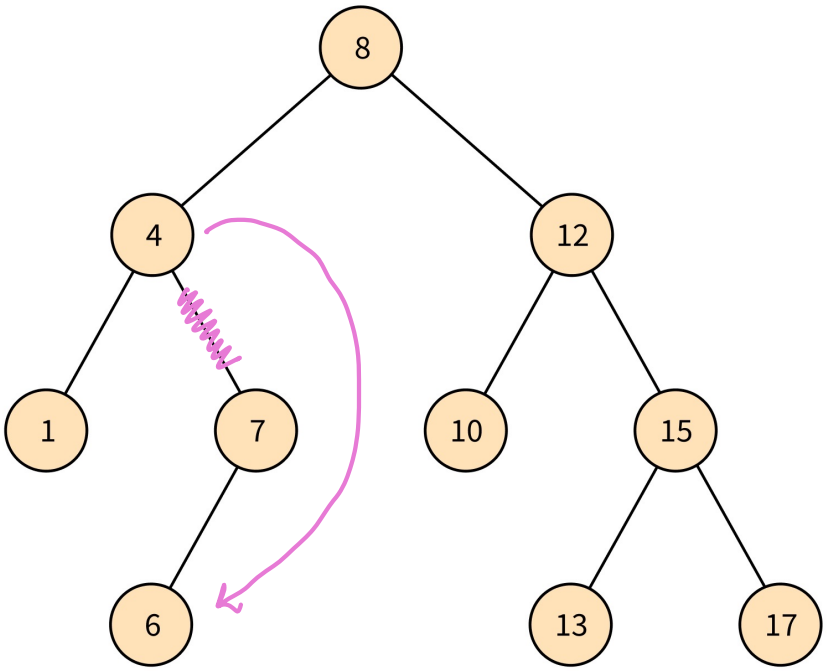

```
curr = root;
while (curr.right != NULL) {
    curr = curr.right;
}
return curr.val;
```

Keep on going right till there is nothing left on RIGHT SIDE.

Deletion in B.S.T

NOTE : All nodes are DISTINCT.

After Deletion of a Node, B.S.T property should hold for remaining tree.



delete (6)

delete (7)

delete (8)

1. Leaf node → Break the connection.
2. Node with single child → Break the connection with node & make connection with its children.
3. Node with both the children

All values in LST < Node.val < All values in RST.

Max OF LST < Node.val < Min OF RST.

NOTE : We need to ensure that the tree still remains BST after deletion also so replace the node value with either

MAX OF LST or Min OF RST.

↓ Rightmost value of LST.

↓ Leftmost value OF RST.

Node with 1 child or its a leaf Node.

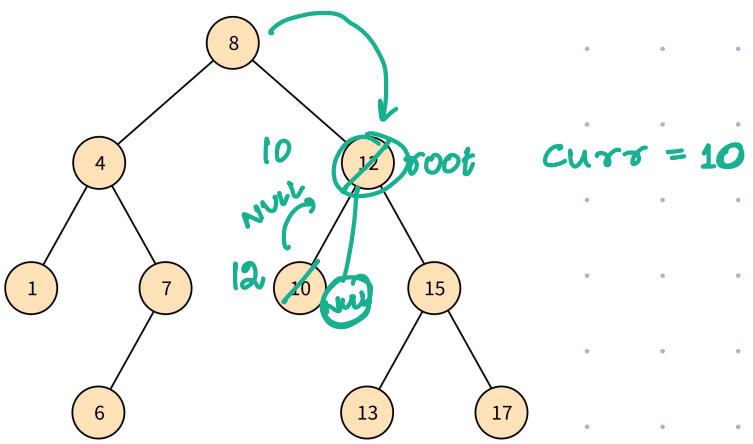
</> Code

```
Node delete(Node root, int K) {

    if (root == NULL)
        return NULL;

    if (root.val == K) {

        // Case 1 : Leaf Node
        if (root.left == NULL && root.right == NULL) {
            return NULL;
        }

        // Case 2 : Node with 1 child.
        else if (root.left == NULL || root.right == NULL) {
            if (root.left == NULL) {
                return root.right;
            }
            else {
                return root.left;
            }
        }
        else {      Case 3 : Node with 2 children.
            // Find largest value of LST.
            Node curr = root.left;
            while (curr.right != NULL)
                curr = curr.right;

            // Swap largest value of LST with Root Node
            swap(curr.val, root.val);

            // Make Recursive call to LST since node to be deleted is now present in LST.
            root.left = delete(root.left, K);
        }
    }
```

T.C : O(H)
S.C : O(H)

```
    else if (K < root.val){      // Recur in LST
        root.left = delete(root.left, K);
    }
    else {                       // Recur in RST
        root.right = delete(root.right, K);
    }

    return root;
}
```

*dry-run



delete(12)

QUIZ What is the purpose of balancing a Binary Search Tree?

[ ] To make it visually appealing

[ ] To ensure all nodes have the same value

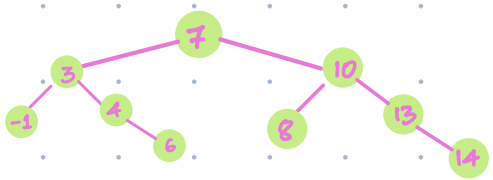
[ ] To maintain efficient search, insert, and delete operations

[ ] Balancing is not necessary in a Binary Search Tree

Construct Balanced B.S.T from sorted array

diff in ht. of subtree ≤ 1

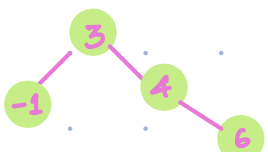
0	1	2	3	4	5	6	7	8
-1	3	4	6	7	8	10	13	14



*dry-run

0	1	2	3	4	5	6	7	8
-1	3	4	6	7	8	10	13	14

$m = 0+8/2 = 4$

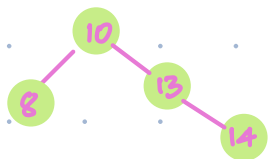


0	1	2	3
-1	3	4	6

$m = 0+3/2 = 1$

7

5	6	7	8
8	10	13	14

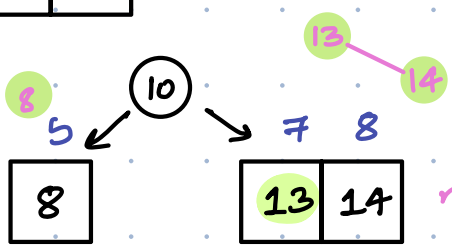


$m = 5+8/2 = 6$

0	2	3
-1	4	6

$m = 0+0/2 = 0$

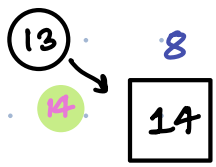
$m = \frac{2+3}{2} = 2$



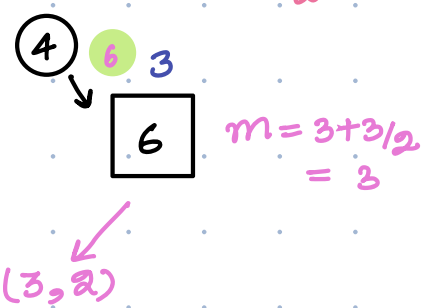
8

13	14

$m = 7+8/2 = 7$



14



$m = 3+3/2 = 3$

(3, 2)

< / > Code

```
Node Construct (int[] A, int s, int e) {
    if(s>e) return NULL;

    m = (s+e)/2;
    Node root = new Node(m);
    root.left = construct (A, s, m-1);
    root.right = construct ( A, m+1, e);
    return root;

}
```

N : size of A[]

T.C : O(N)

S.C : O(H)

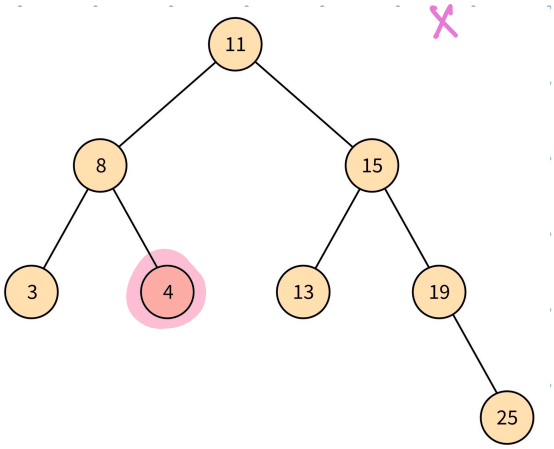
Check if given Binary Tree is a B.S.T

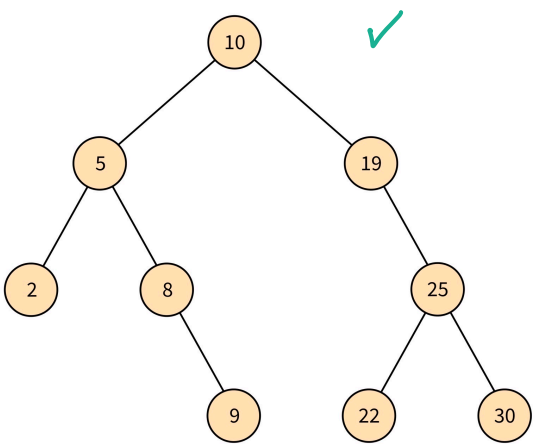
BST is defined as follows:

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

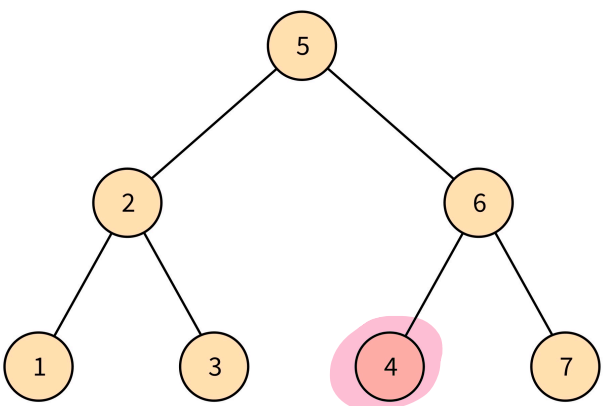
2) The right subtree of a node contains only nodes with keys greater than the node's key.

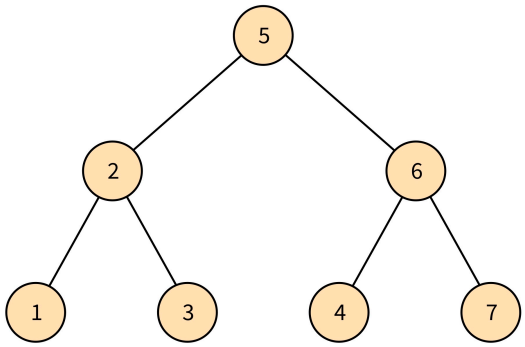
3) Both the left and right subtrees must also be binary search trees.





QUIZ





Approach : Find Inorder of given tree & then check if it SORTED or not.

T.C : O(N)

S.C : O(N)

Can we do it without taking additional array ?

Approach : Instead of storing the Inorder in array & later checking if it is SORTED or n't we can check it on the go.

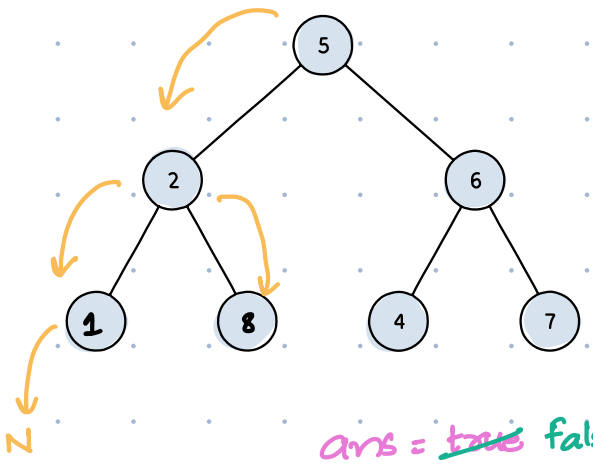
</> Code

Node prev = NULL; ans = true;

T.C : O(N)
S.C : O(H)

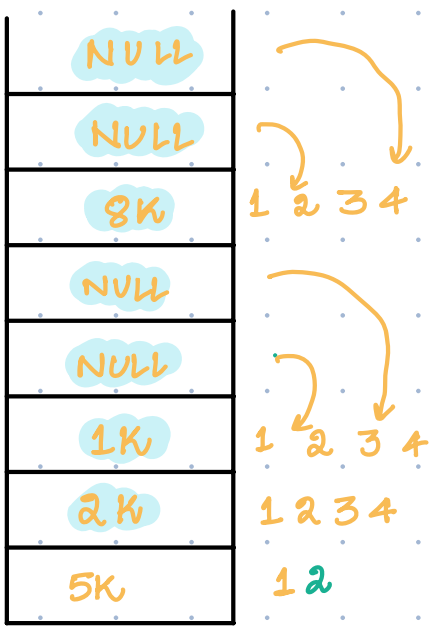
```
void inorder(Node root){
    if (root == NULL) return;
    inorder(root.left);          — (1)
    if( prev != NULL && root.val <= prev.val){   — (2)
        ans = false;
        return;
    }
    prev = root;                 — (3)
    inorder(root.right);         — (4)
}
```

*dry-run



ans = ~~true~~ false

prev = ~~NULL~~ ~~1K~~, ~~2K~~, 8K



1