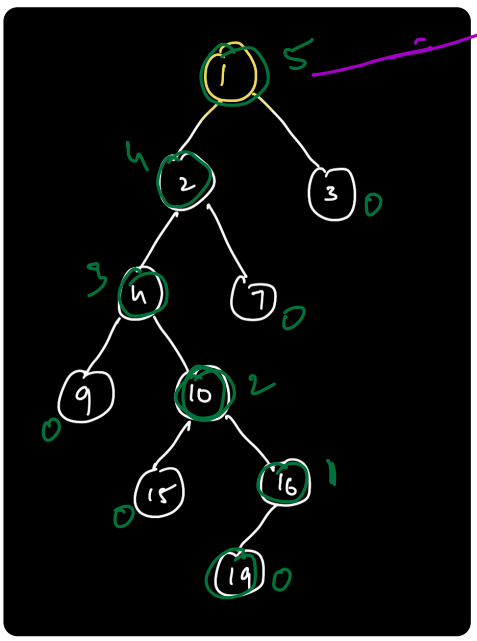
Basics of Trees - 2

Question: Height of tree

Height(tree) = No. of edges from root to the farthest leaf.



root

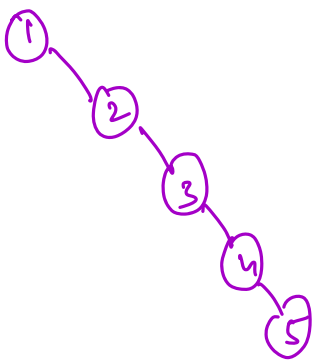
Height(leaf) = 0

height(node) = 1 + max(height(left), height(right));

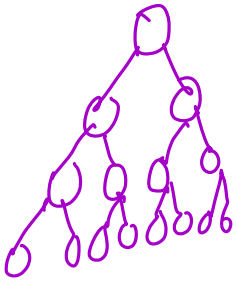
if max(0,1) = 1

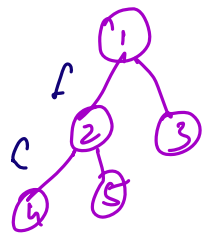
Height(root) = 5

Ex:



N = 5
height(tree) = 4



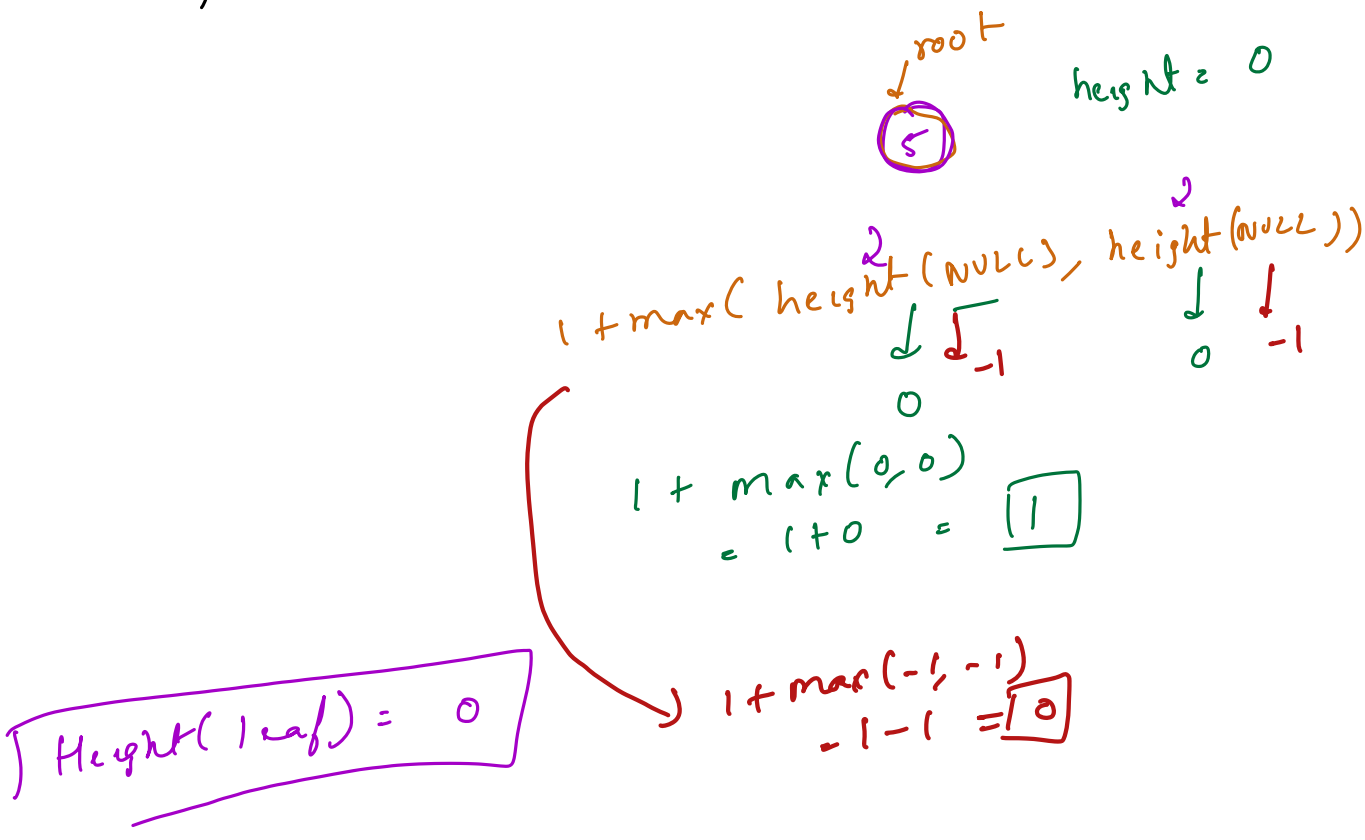


N = 5
height(tree) = 2

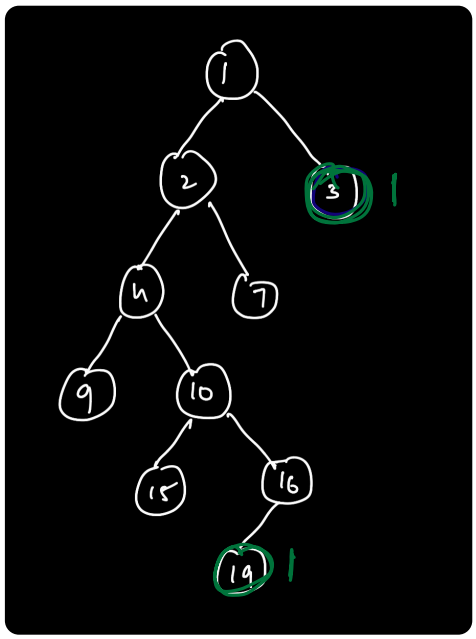
Assumption: height(root) returns height of the tree

Main Logic: height(root) = 1 + max(height(root.left), height(root.right));

```
int height(root) {
    if(root == NULL) {
        return -1;
    }
    return 1 + max(height(root.left), height(root.right));
}
```



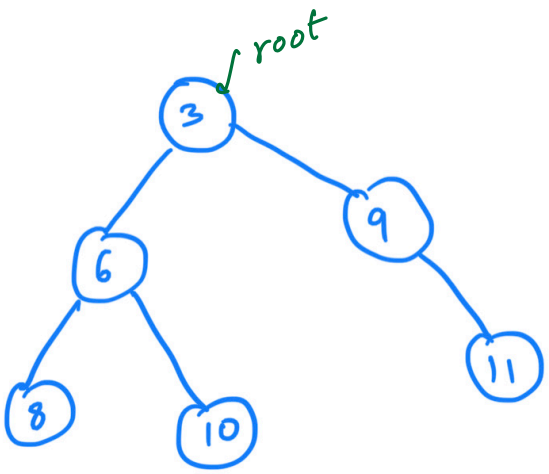
* Assignment: Height No. of nodes from root to the farthest leaf.



height(leaf) = 1

```
if(root == NULL){
    return 0;
}
```

Question: Given a binary tree, search for an element.



k = 10 ⇒ True

k = 9 ⇒ True

k = 13 ⇒ False

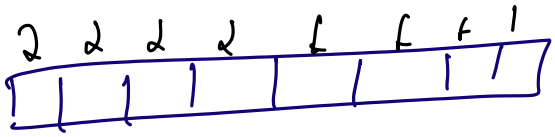
Simple Approach:-

Do either preordu, postordu, inordu

```
bool isPresent = false
```

```
if( root.val == k ){
    isPresent = true;
}
```

A:



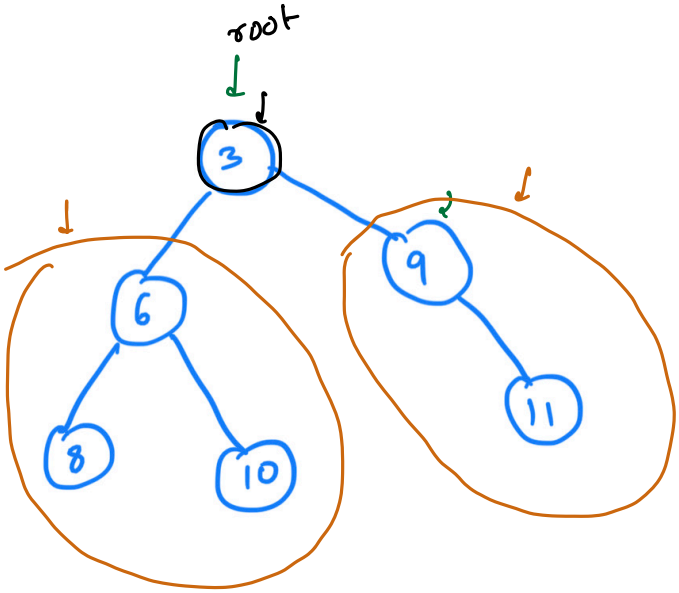

```
for(i=0; i<N; i++){
    if( A[i] == k ){
        return true;
    }
}
```

word ← iat

```
preordu (root){
if(root == NULL){
    return;
}
if( root.val == k ){
    return true;
}
print(root.val);
preordu (root.left);
preordu (root.right);
return;
}
```

⇒ isPresent = true;

```
if( isPresent == true ){
    TRUE;
}
else{
    FALSE;
}
```



k = 10

1 + nodes(root.left) + nodes(root.right);

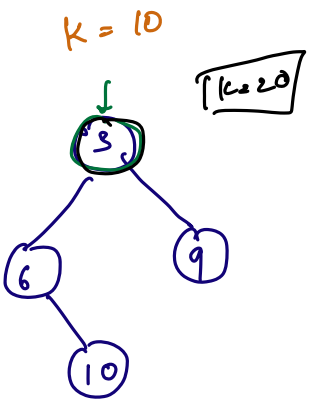
True / False -

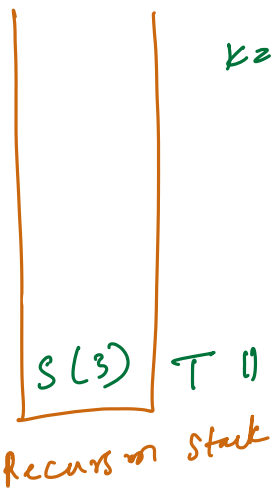
Assumption: search(root, k) checks if k is present in the tree or not.

```
boolean search(root, k) {
    if(root == NULL) {
        return false;
    }
    if(root.val == k) {
        return true;
    }
    return search(root.left, k) || search(root.right, k)
}
```

T || F => T
T || T => T
F || F => F

```
boolean search(root, k) {
    if(root == NULL) {
        return false;
    }
    if(root.val == k) {
        return true;
    }
    return search(root.left, k) || search(root.right, k);
}
```



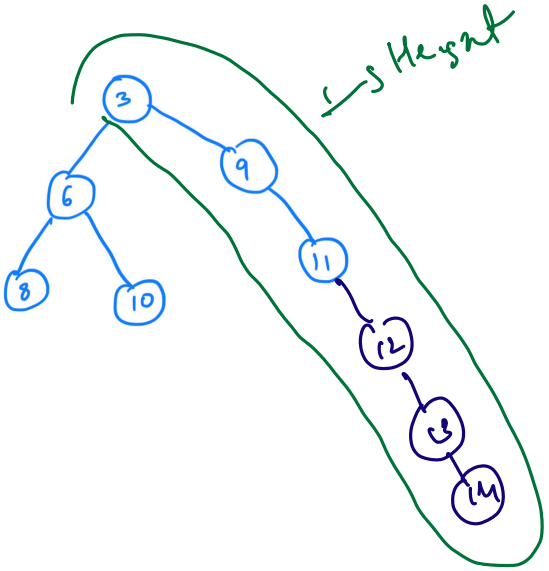


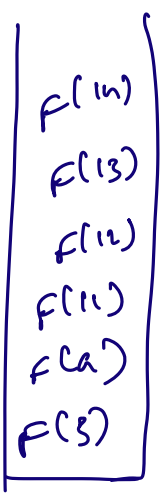
T.C : O(N)

S.C :

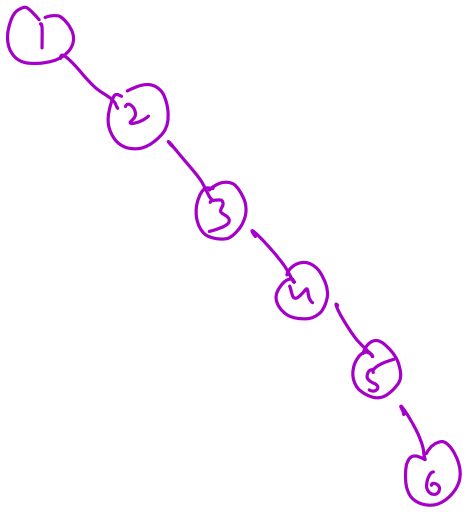
Recursion stack

S.C : # Max no. of active function calls at any point





S.C : O(H) => H is the height of tree



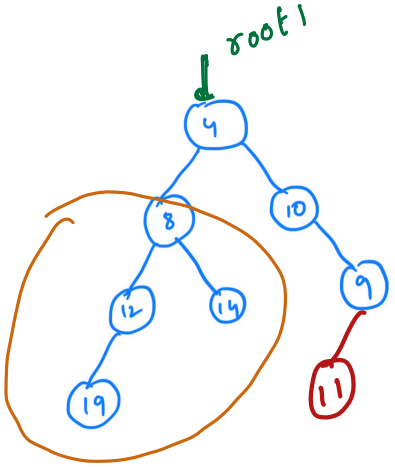
Skewed tree

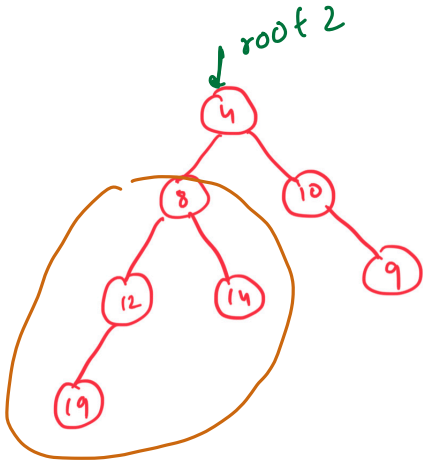
$N = 6$

$h = N-1$ $\Rightarrow O(N)$

$\Rightarrow (H+1) \Rightarrow O(H)$

Question: Check if 2 trees are similar





} True

Assumption: checkSimilar(root1, root2) checks if the trees are similar or not (Boolean)

Main logic:

```
root1.val == root2.val &&
checkSimilar(root1.left, root2.left) &&
checkSimilar(root1.right, root2.right);
```

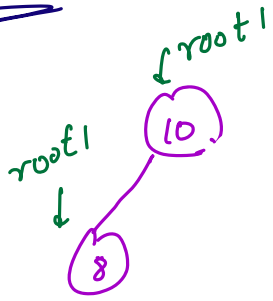
Base case:

```
bool checkSimilar(root1, root2){
    <Base Conditions>
    if(root1.val != root2.val){
        return false;
    }
    return checkSimilar(root1.left, root2.left) && checkSimilar(root1.right, root2.right)
}
```

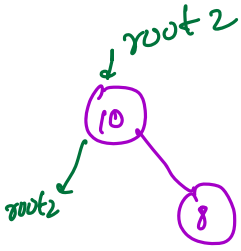
false

Base Cases :

1)



root1 = 8

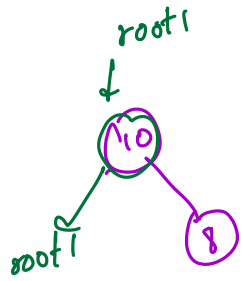


10. left = NULL

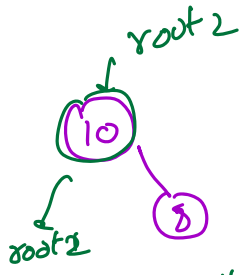
root2 = NULL

if one is NULL and other is not NULL, return false

2)



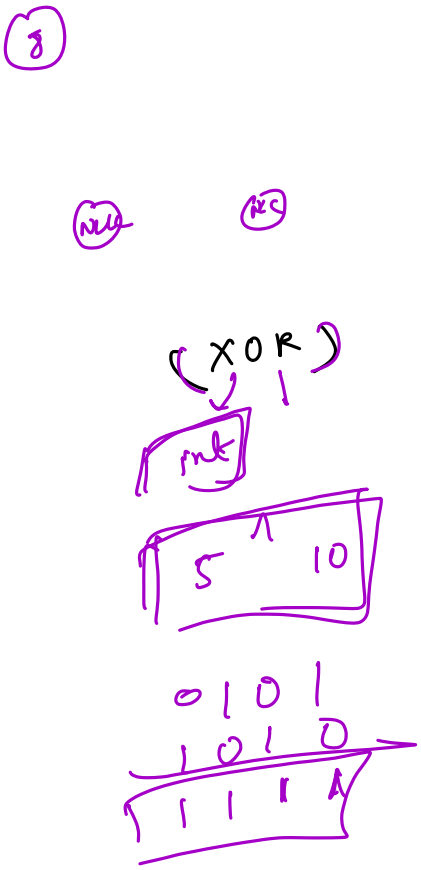
root1 = NULL



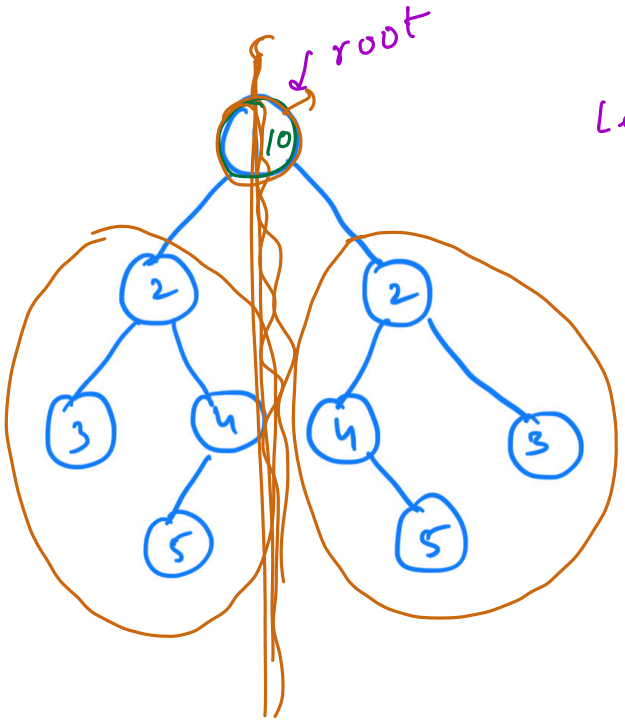
root2 = NULL

```
if(root1 == NULL && root2 == NULL){
    return true;
}
if(root1 == NULL || root2 == NULL){
    return false;
}
```

root1 = 10 root2 = NULL


8
XOR
1
5
10
0 1 0 1
1 0 1 0
1 1 1 1

Question: Check if a tree is symmetric

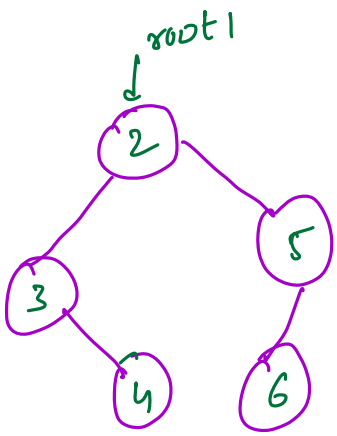


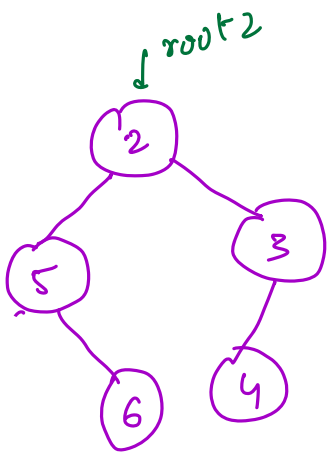
Left & Right are mirror images.

Simpler Problem:

Check if 2 trees are mirror images of each other.

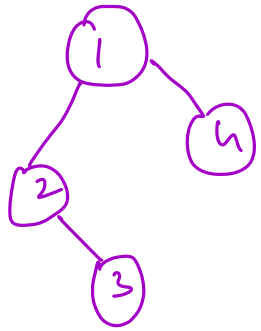
Assumption: isMirror(root1, root2) checks if 2 trees are mirror images of each other.



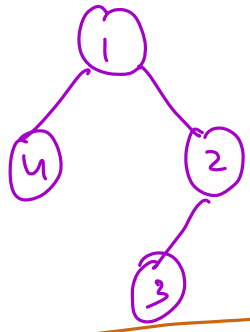


left root right

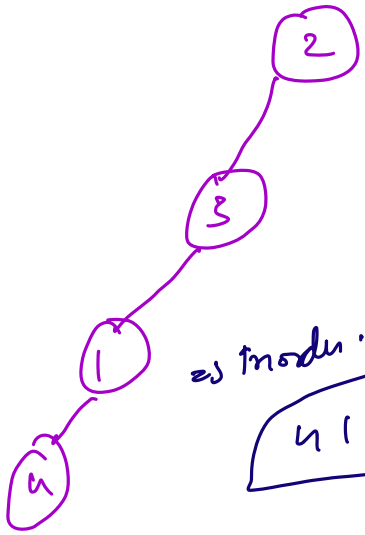
left root right



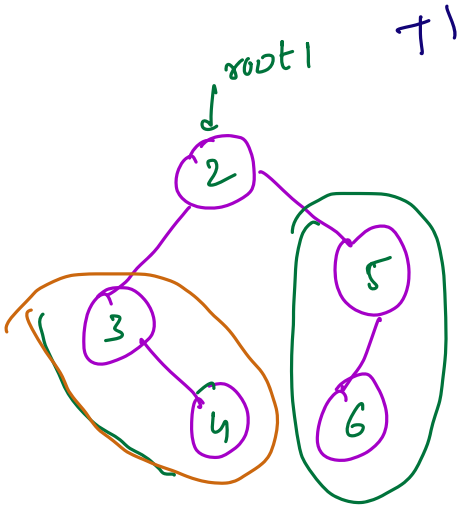
Inorder: 2 3 1 4

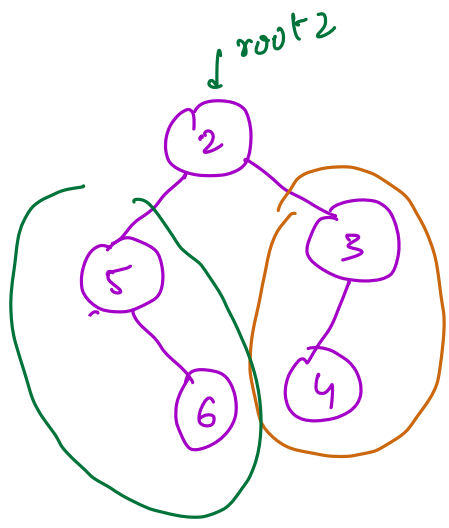


Inorder: 4 1 3 2



Inorder: 4 1 3 2

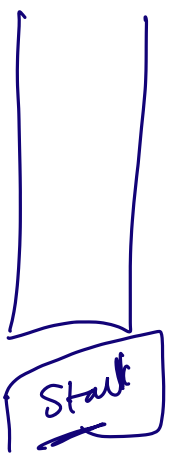


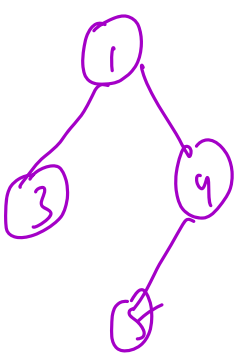


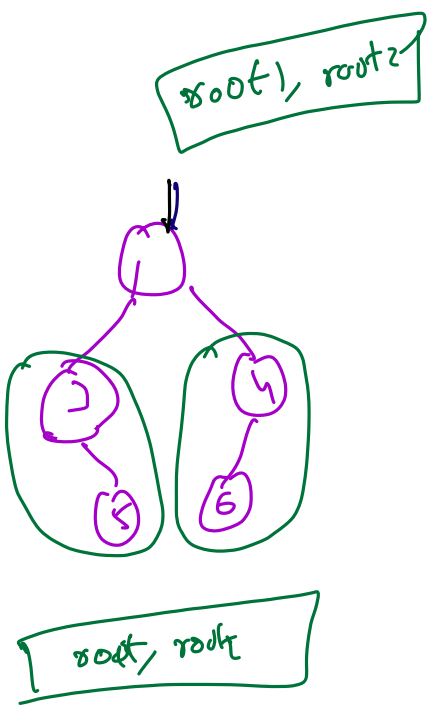
Main Logic :

1) Root values should be equal
2) LST (Tree1) = Mirror Image of RST (Tree2)
3) RST (Tree1) = Mirror Image of LST (Tree2)

```
bool isMirror(root1, root2){
if (root1 == NULL && root2 == NULL){
    return true;
}
if root1 == NULL || root2 == NULL){
    return false;
}
if(root1.val != root2.val){
    return false;
}
return isMirror(root1.left, root2.right)
    && isMirror(root1.right, root2.left);
}
```





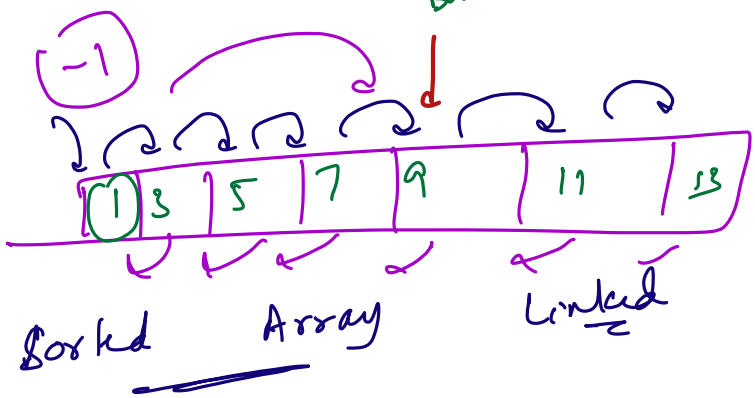


Binary Search Tree

Binary → At most 2 children

Search → O(log n) → Balanced search

Tree → Hierarchical DS

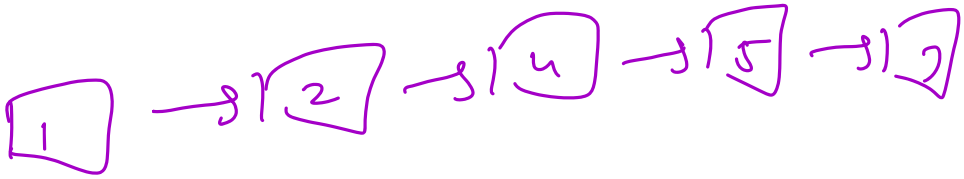


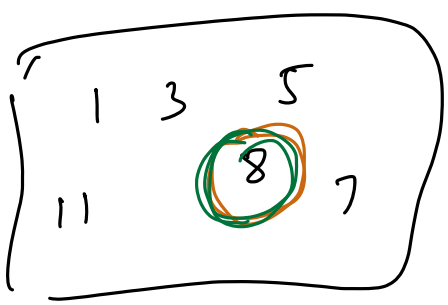
Sorted Array — Linked

BST → Ordering

Hash Set / Hash Map

	Sorted Array	Linked	BST	Hash Set / Hash Map
Search:	O(log n)	O(N)	O(log n)	O(1)
Insert:	O(n)	O(N)	O(log n)	O(1)
Delete:	O(N)	O(N)	O(log n)	O(1)





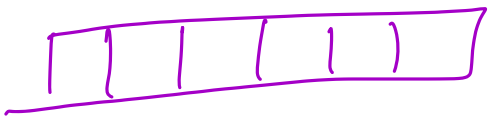
3

Data is ordered

O(log n)

Tree Map / Tree Set: Balanced BST (Red-black / AVL)

S.C : O(1)



S.C : O(1)

Unordered_map => Hash Table → O(1)

ordered_map => B.B.S.T → O(log n)

↓

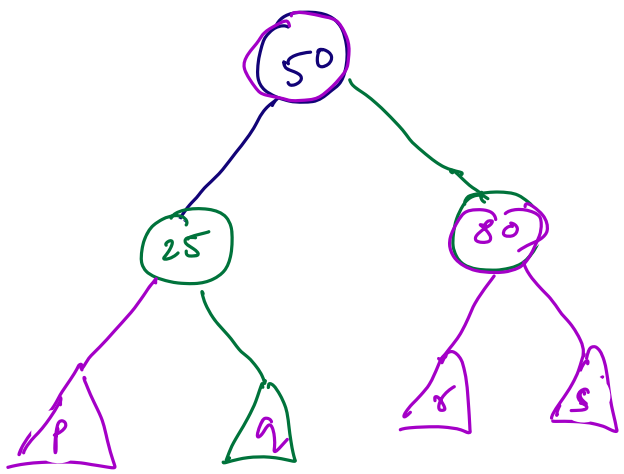
TreeSet / TreeMap

Binary Search Tree

1) All nodes of L.S.T should be less than the root

2) All nodes of R.S.T should be greater than Root

3) L.S.T and R.S.T should be B.S.T



a < 50

b > 50

p < 25

q > 25 && q < 50

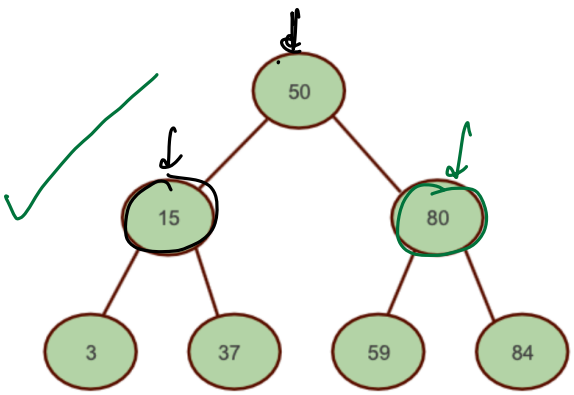
r < 80 && r > 50

s > 80

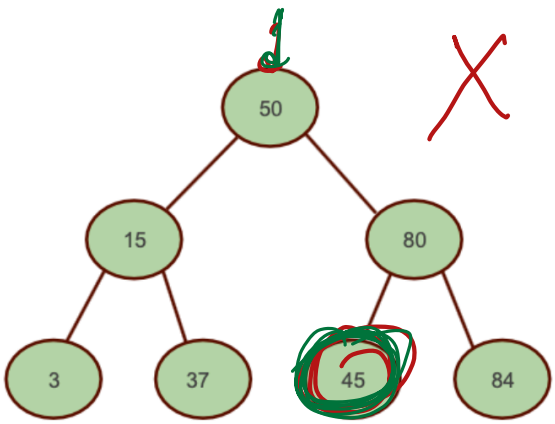
Duplicates

1) Always insert in L.S.T ✓
2) Always insert in R.S.T ✓
3) Maintain a count at every Node

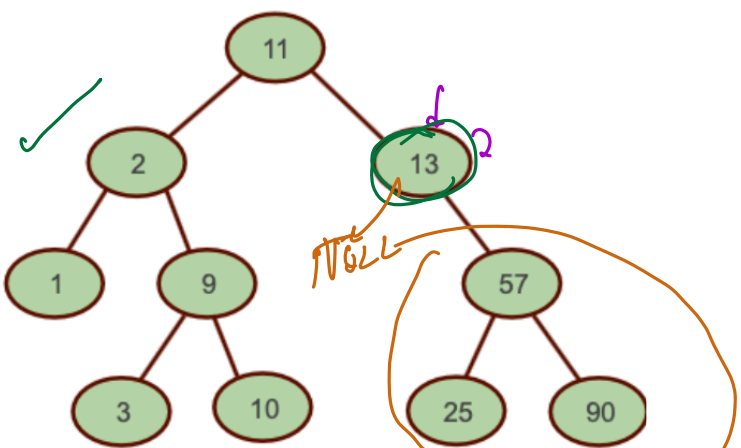
val: 10
count: 2



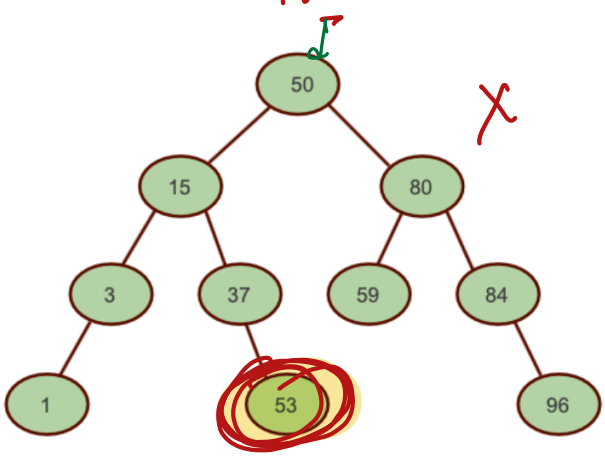
Tree 1



Tree 2

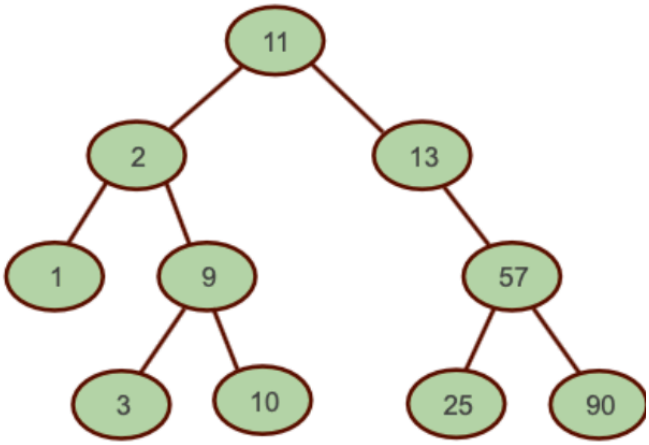


Tree 3



Tree 4

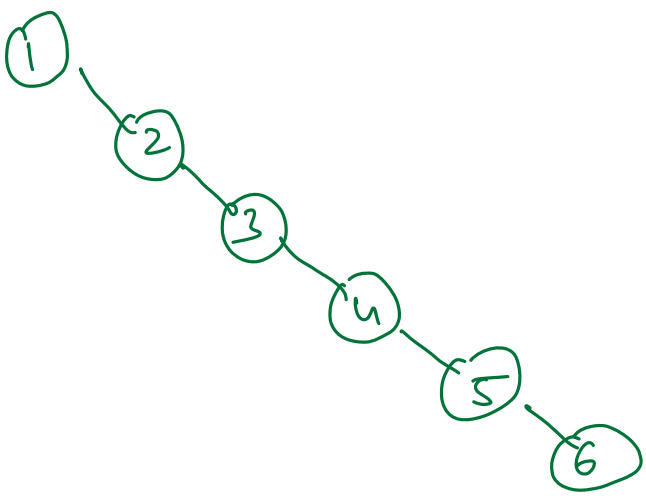
Inorder Traversal :



Left | Root | Right

Inorder: 1 2 3 9 10 11 13 25 57 90

Inorder traversal of BST is always (Sorted)



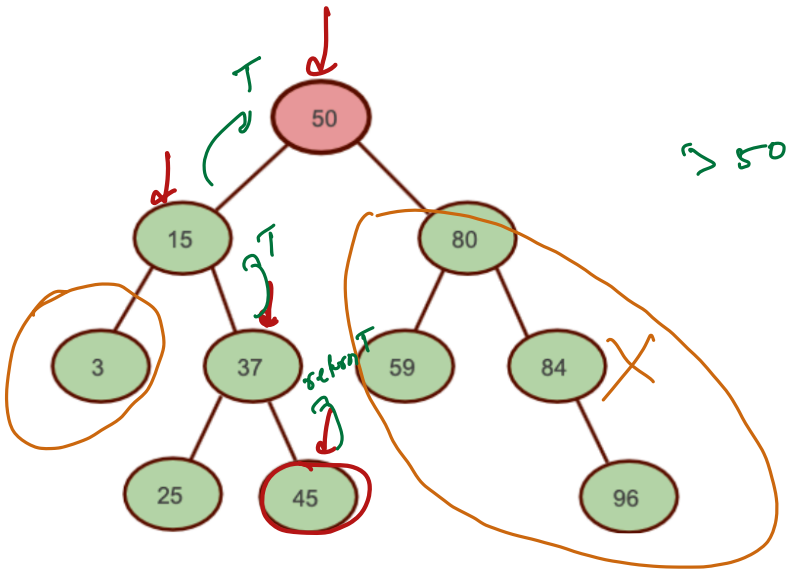
Question: Check if a B.T is BST or not?

→ Find Inorder

→ Check if it's sorted

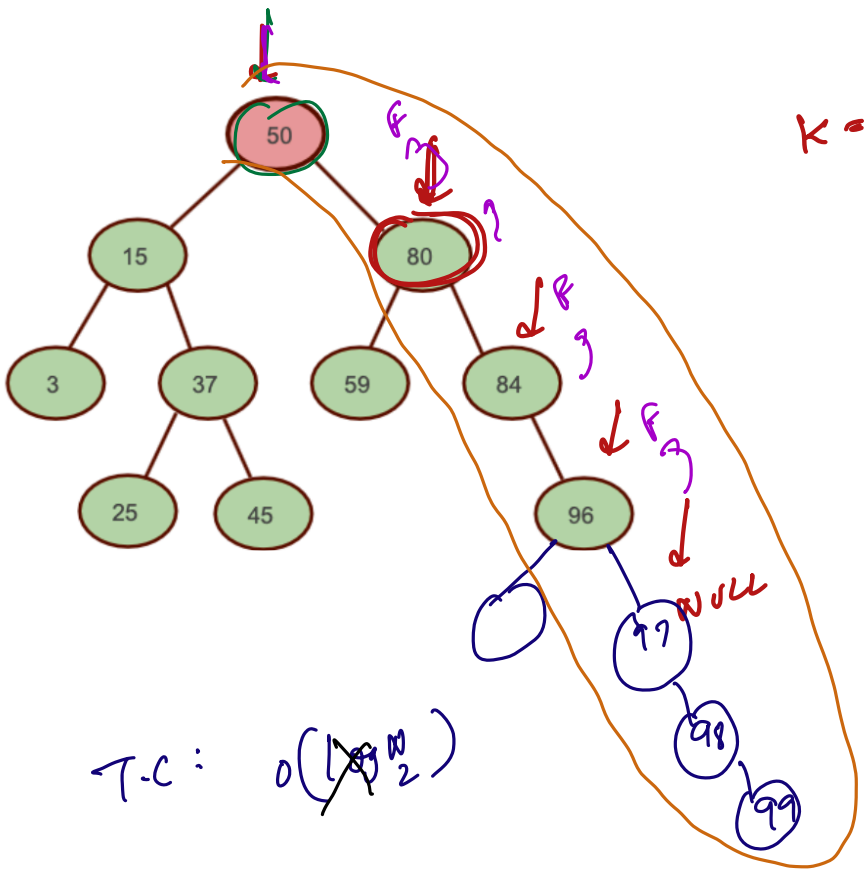
Question: Search an element in B.S.T

k = 45



> 50

k = 50



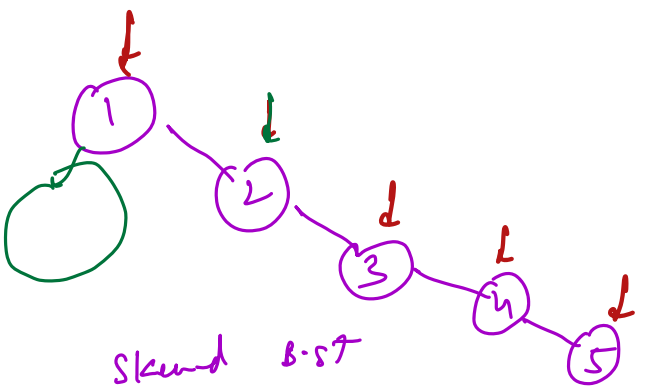
k = 100

k = 100

⇒ H iterations

T.C : $O(\log_2 N)$

H is height of tree



k = 10

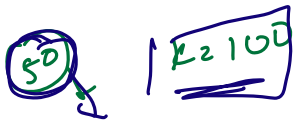
Skewed B.S.T

T.C : O(H)

H in the worst case is N

=> O(N)

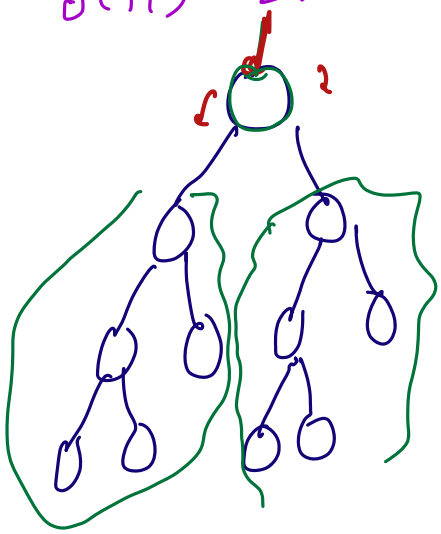
S.C : O(H) => O(N)

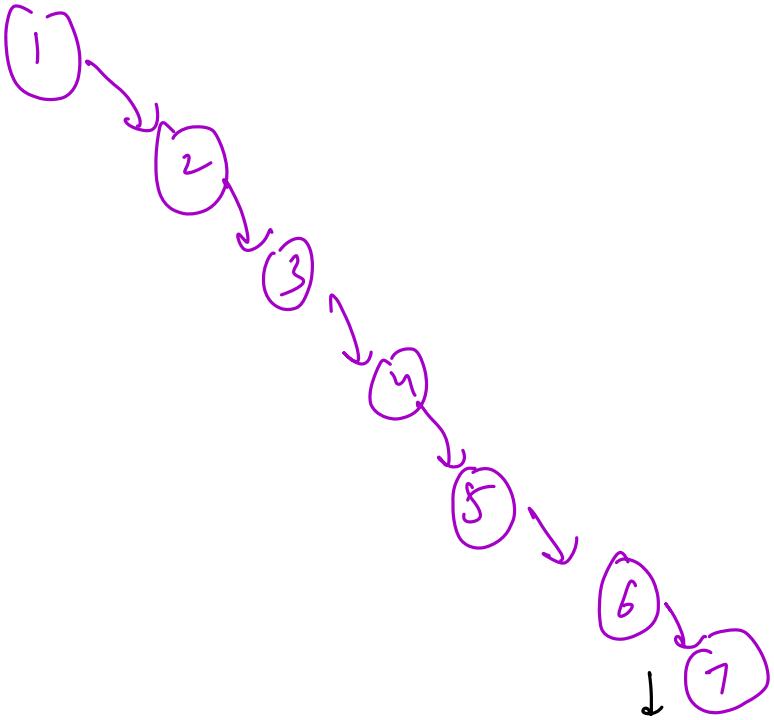



```
boolean search (root, k ) {
    if(root == NULL){
        return false;
    }
    if(root.val == k) {
        return true;
    }
    if( root.val < k) {
        return search( root.right, k);
    }
    else {
        return search(root.left, k);
    }
}
```

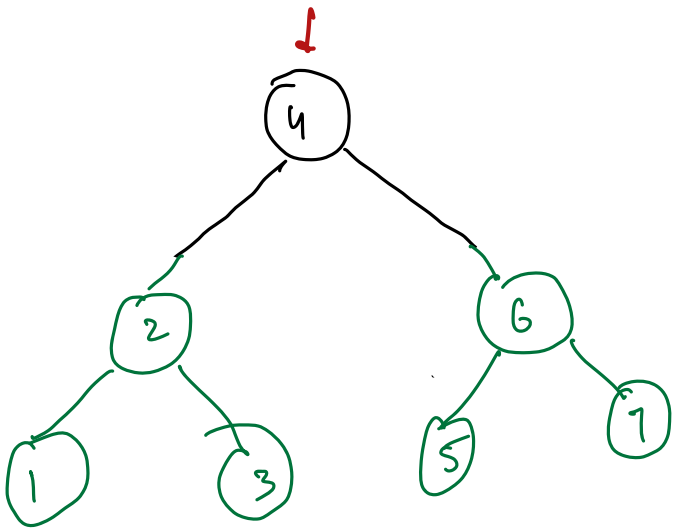
T.C : O(H) => O(N)

S.C : O(H) => O(N)





Inorder: 1 2 3 4 5 6 7

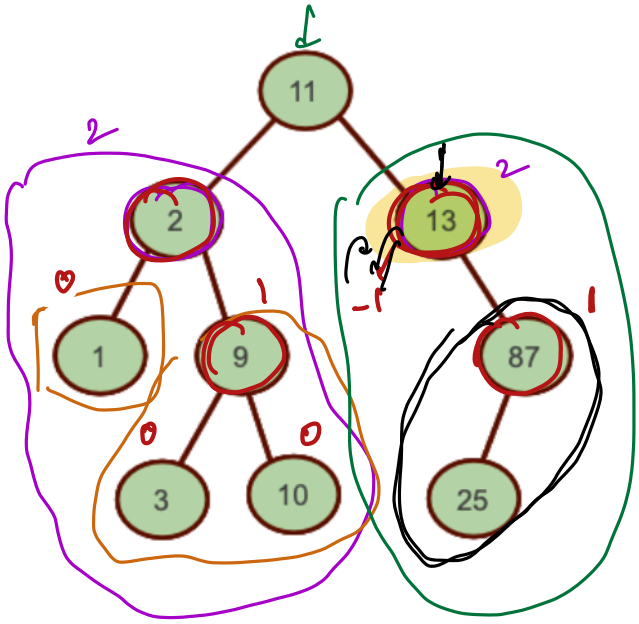


Balanced Tree

=> The height of a balanced tree : $O(\log N)$

For all nodes,

abs (Height(LST) − Height(RST)) ≤ 1



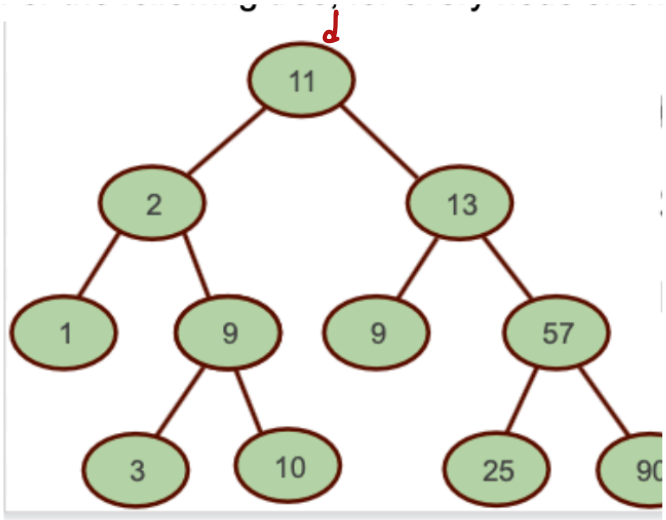
✗ Balanced Tree

abs (2 − 2) = 0

abs(0 − 1) = 1

abs(0 − 0) = 0

(1 − (−1)) = 2



≤ 1

⟹ Balanced Tree

Height of Balanced Tree : O(log n)

Balanced BST ⇒ O(H) ⟹ O(log n)

Question: Given a binary check if it is balanced or not

```
boolean isBalanced = True;

int height(root) {
    if(root == NULL) {
        return -1;
    }
    hl = height(root.left);
    hr = height(root.right);
    if( abs(hl - hr) > 1 ) {
        isBalanced = False;
    }
    return 1 + max(hl, hr);
}

main() {
    isBalanced = True;
    height(root);
    print(isBalanced);
}
```

```
class {
    int height;
    bool isBalanced;
}
```

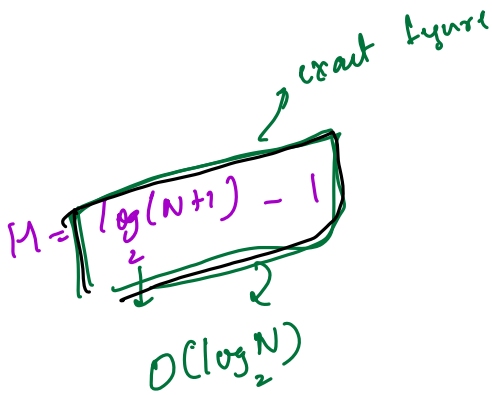
Function: Given a tree, it would return the height as well as if it is balanced or not.

```
class output {
    int height;
    boolean isBalanced;
}
```

Output

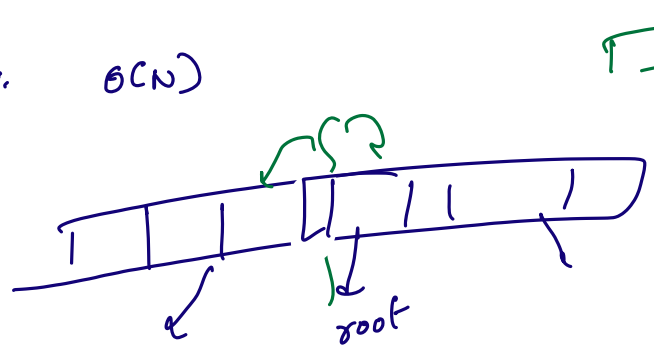
```
getheightIsBalanced(root){
if(root==NULL){
    return {-1, True};
}
Output left = getheightIs...(root.left)
Output right = getheight....(root.right)
if((left.ht - right.ht) > 1)
    return {left, False};
if(left.isBalanced == False || right.isBalanced == False)
    return {left, False};
}
```

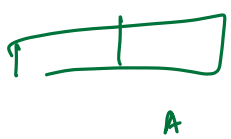
```
if(root==NULL){
    return {-1, True};
}
```



$\log_2^N \Rightarrow O(\log_2 N)$

Inorder: $O(N)$



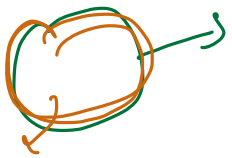


$$T(N) = T\left(\frac{N}{2}\right) + T\left(\frac{N}{2}\right) + O(1)$$

$$T(N) = 2T\left(\frac{N}{2}\right) + O(1)$$

$T(N) = O(N)$

$O(N) + O(N) =$ $T(N) = O(N)$



```
class Node {
    int val;
    Node left;
    Node right;
    int count;
}
```

arr

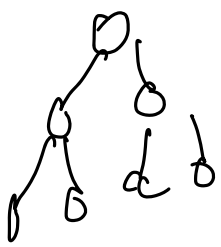
3	7	1	1	3	7	6	5	7

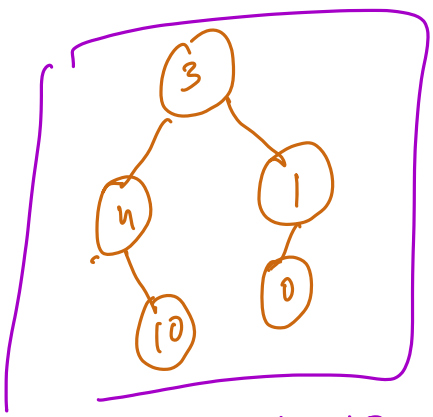
$O(\log n)$

$k = 20$

$O(\log n)$

$\Rightarrow O(n)$

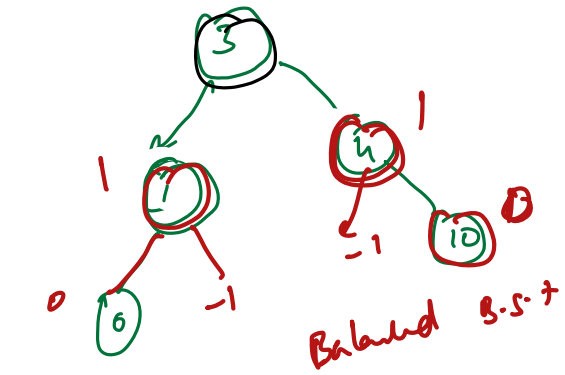




left root right

$\Rightarrow$ 4 10 3 0 1

Sort: 0 1 3 4 10



Balanced B.S.T

(1-1) = 0

(0-(-1)) = 1

(-1-0) = 1

root1 == NULL && root2 != NULL