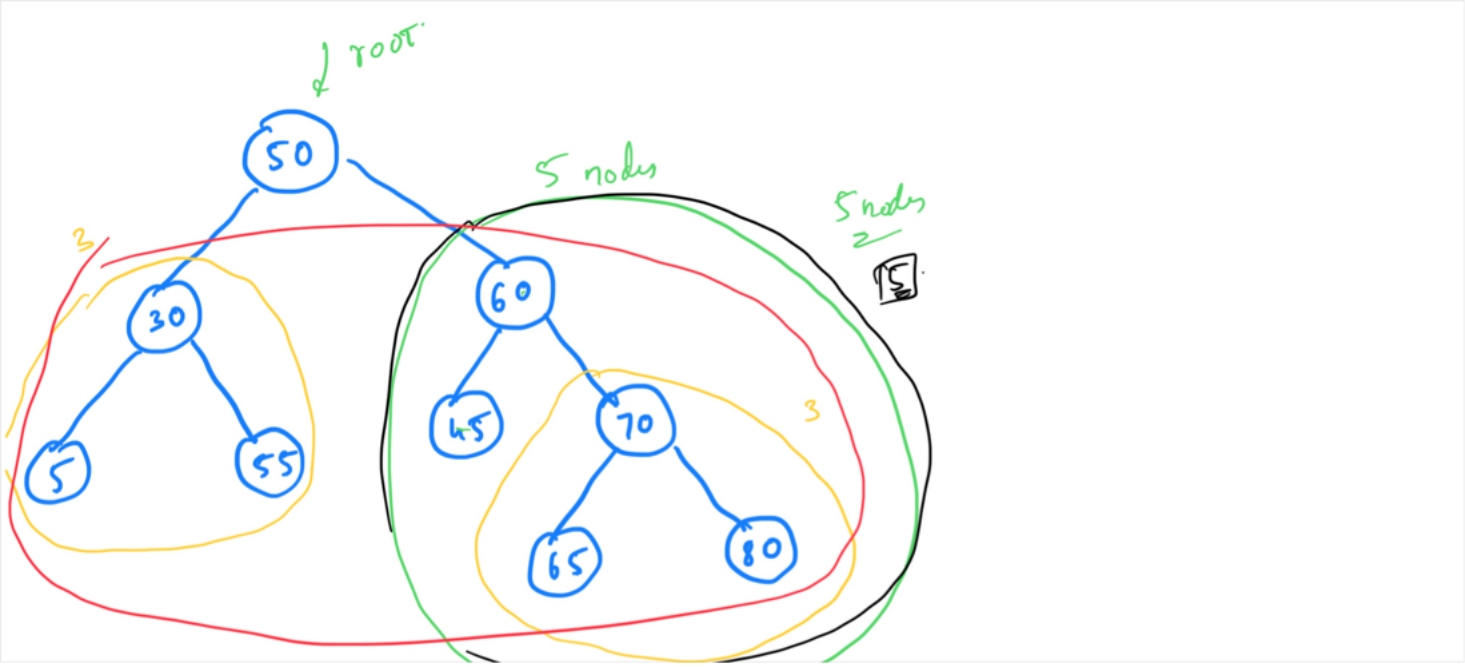
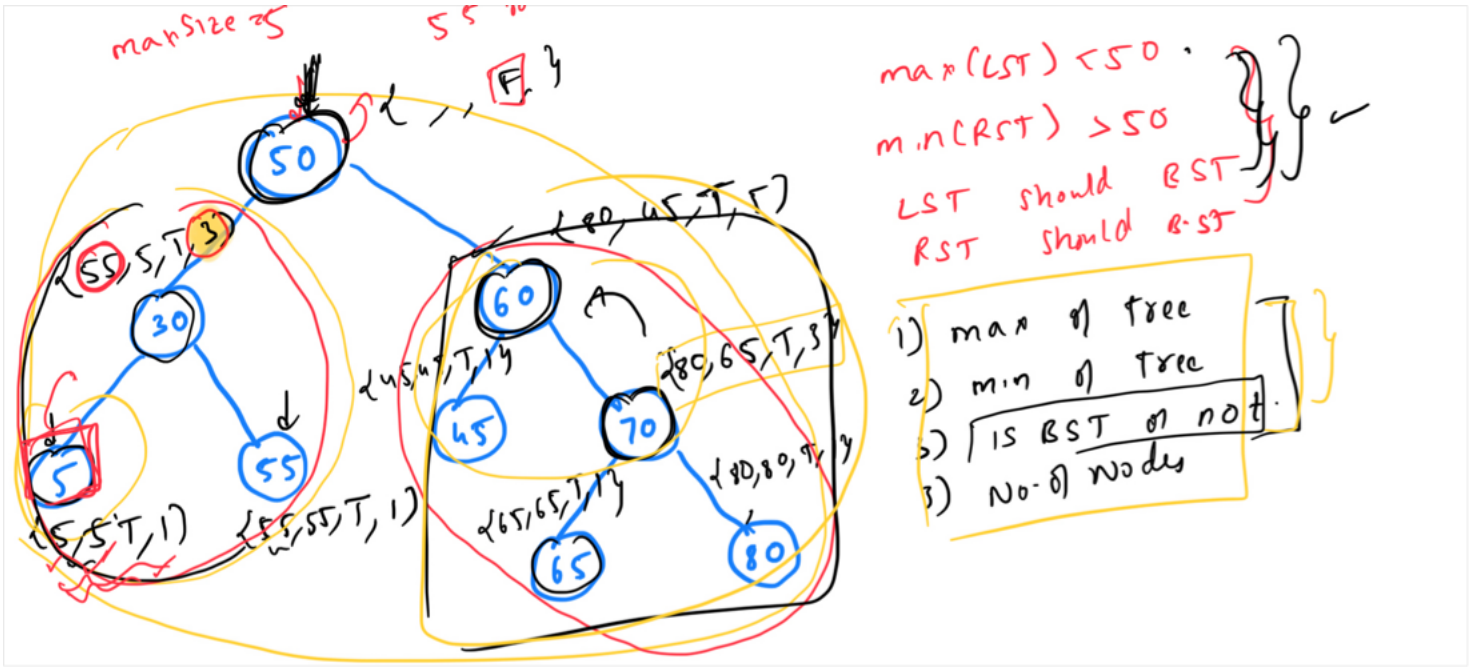
Problems on Trees

Question: Find Largest subtree which is a BST



BST

1) Max (LST) < Root
2) Min (RST) > Root.
3) LST & RST should be BST.


50
30
60
5
55
45
70
65
80
max(LST) < 50
min(RST) > 50
LST should BST
RST should BST
1) max of tree
2) min of tree
3) IS BST or not.
3) No. of Nodes

Node ? int ? boolean ?

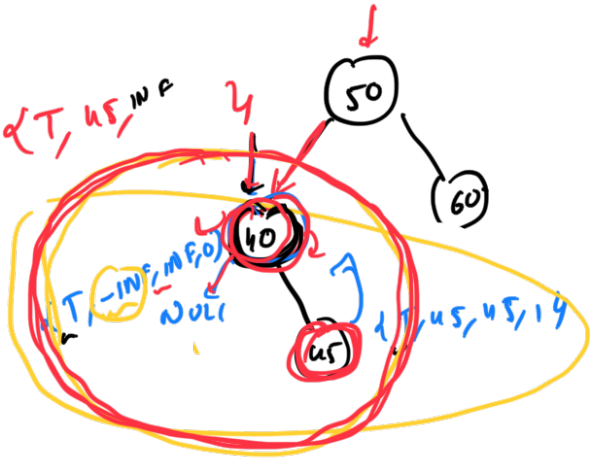
```
class Into {
    int max, min, count;
    boolean isBST;
}

int maxSize = -INF;
Into LargestBST(root) {
    if(root == NULL) {
        return {T, -INF, INF, 0};
    }
    Into left = LargestBST(root.left);
    Into right = LargestBST(root.right);
    if( left.isBST && right.isBST &&
        root > left.max && root < right.min ) {
        Into into;
        into.isBST = True;
        into.max = max(right.max, root.data)
        into.min = min(left.min, root.data)
        into.count = 1 + left.count + right.count;
        maxSize = max(maxSize, into.count);
    } else {
        into into;
        into.isBST = false;
    }
    return into;
}
```

Left, Right, Root

{INF, 0}

{T, US, INF

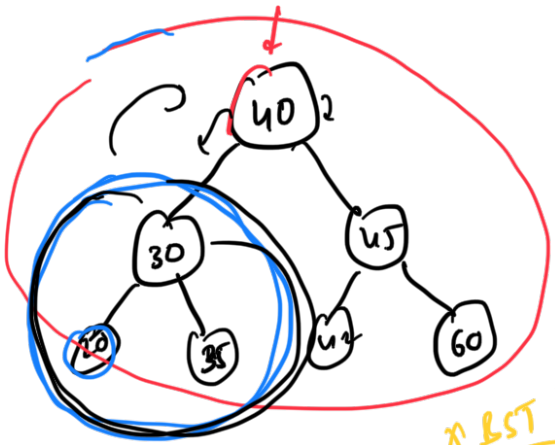
1) isBST = True
2) max = -INF
3) min = +INF



4) count = 0

left.max

-INF < 40 ✓
40 < 45 ✓



Info.min = min(left.min, root.data)

isBST
max = 55
min = 20
count = 3

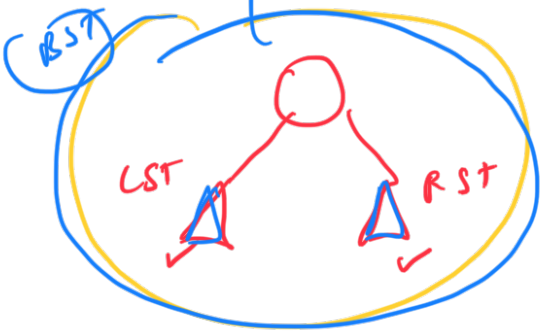
isBST = True
{ max =
min =
count = 0

1) 50 > max(LST)
2) 50 < min(RST)

(-INF, INF)

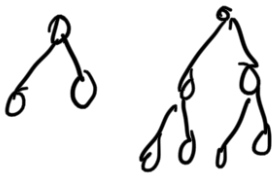
Empty

50 > +INF ✗

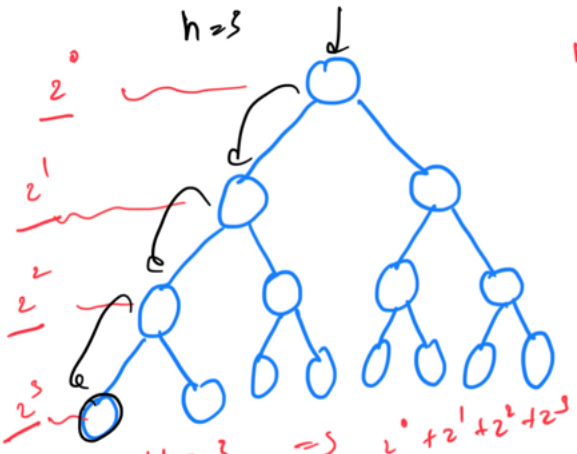


Question: Height of a perfect Binary tree with N nodes ✓

-> Perfect BT: All levels are completely filled



H = height of the tree



No. of terms = h+1

$N = 2^0 + 2^1 + 2^2 + \ldots + 2^h$

$\frac{a(r^n - 1)}{r - 1} = \frac{1(2^{h+1} - 1)}{2 - 1} = N$

$N = 2^{h+1} - 1$

$N + 1 = 2^{h+1}$

$(h+1) = \log_2(N+1)$

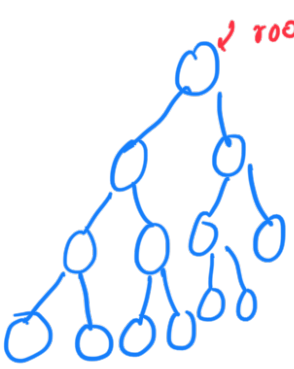
$h = \log_2(N+1) - 1$

$H = O(\log n)$

$N \geq 2^{H+1} - 1$

Question: Count no. of nodes in Complete B.T

→ All levels are filled except possibly the last level



All complete B.Ts ⇒ Perfect.

Perfect ⇒ Complete

Approach 1:

count(root) = 1 + count(root.left) + count(root.right)

if(root == N) return 0;

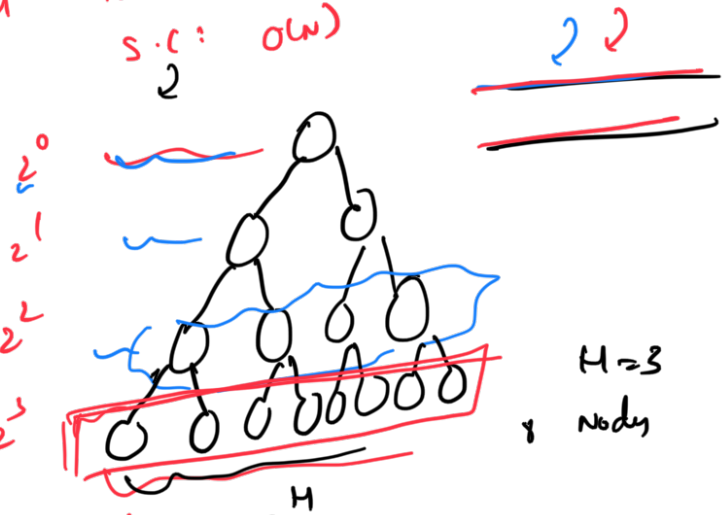
T.C: O(n), S.C: O(H)

Trees-1

Approach 2:

Level order traversal

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



$2^3 = 2^H$

Last: 2^H

$H = \log_2(N+1) - 1$

$2^h \rightarrow$

$2^{\log_2(N+1)} - 1 = \frac{2^{\log_2(N+1)}}{2} = \frac{(N+1)}{2}$

$2^{\log_2(N+1)} = (N+1)$

$2^{a-b} = \frac{2^a}{2^b}$

$\log_2^{N+1} = (N+1) = \frac{(N+1)}{2}$

O(N)

O(logn)

N =

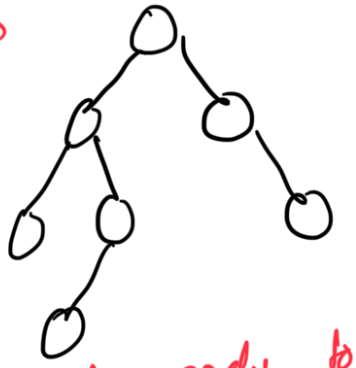
Trey-2

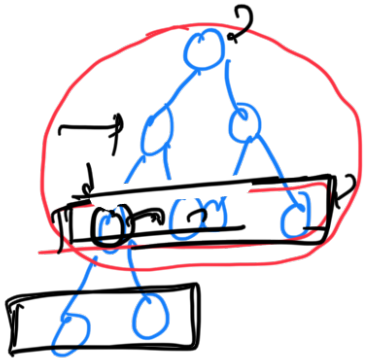
2 int, 2 int

Approach:

Complete Binary Tree :

Random

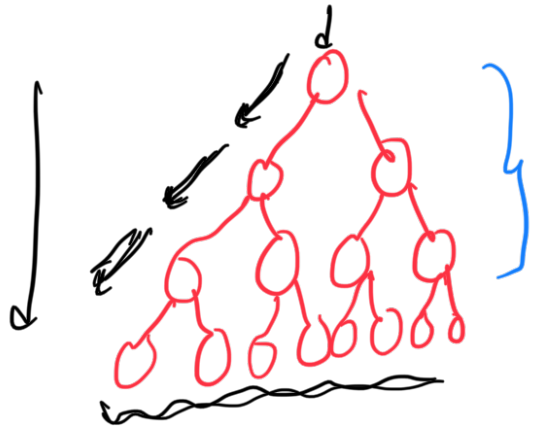




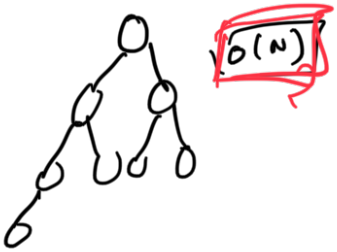
Visit all nodes to find the count

Visit all nodes

T.C : O(N)



$N = 2^{h+1} - 1$



Height = O(N)

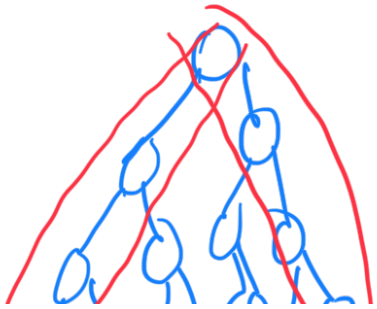
↓

O(H)

$N = 2^{h+1} - 1$

-> When can we say a Complete B.T is a perfect Binary Tree :

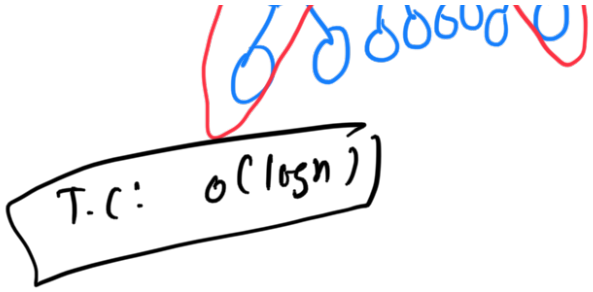
-> ✓ Last Level is filled in Complete B.T



T.C: O(H)

$H = O(\log n)$

Proof : $H = \log(N+1) - 1$

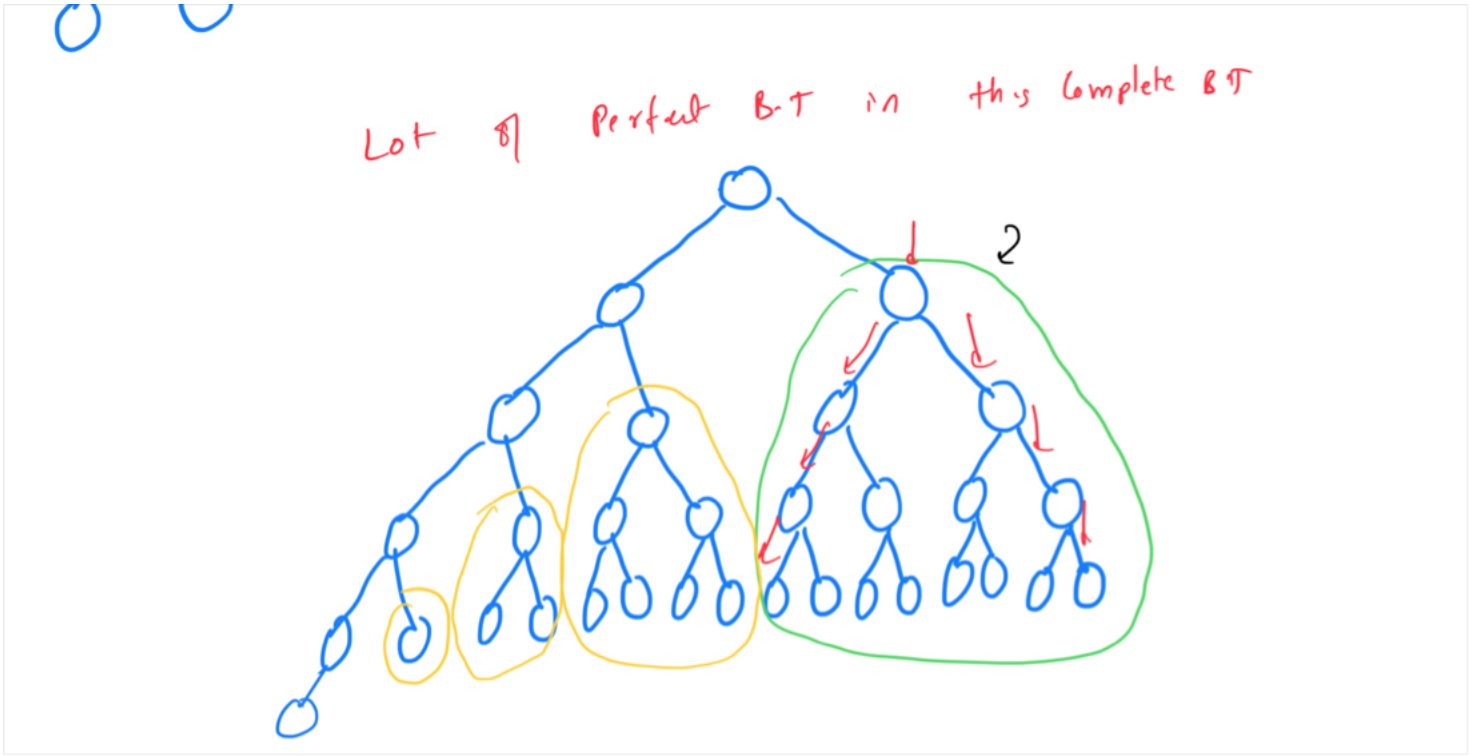


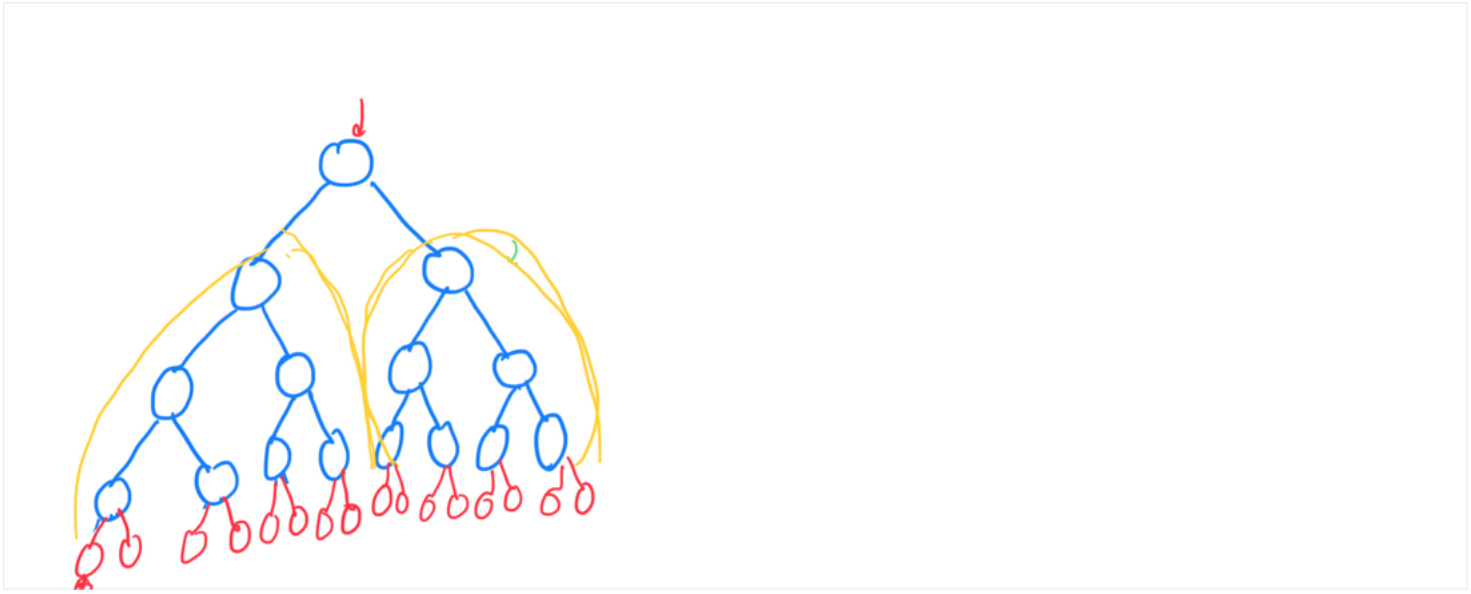
T.C: O(log n)

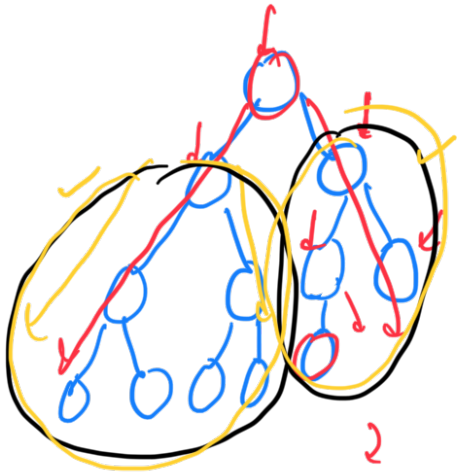
Complete → Balanced

O(log n)

Complete BT:







RST is complete BT
1 subtree would be perfect
both the subtrees are complete

Assumption: nodes in a complete binary tree.

getNodeCount(root) returns the no. of

```
int getNodyCount(root) {
if(root == NULL) return 0;

int lh = getLeftHeight(root);
int rh = getRightHeight(root);
if(lh == rh){
    return 2^(h+1) - 1;
}
else
    return 1 + getNodyCount(root.left) +
               getNodyCount(root.right);
}
```

O(logn)

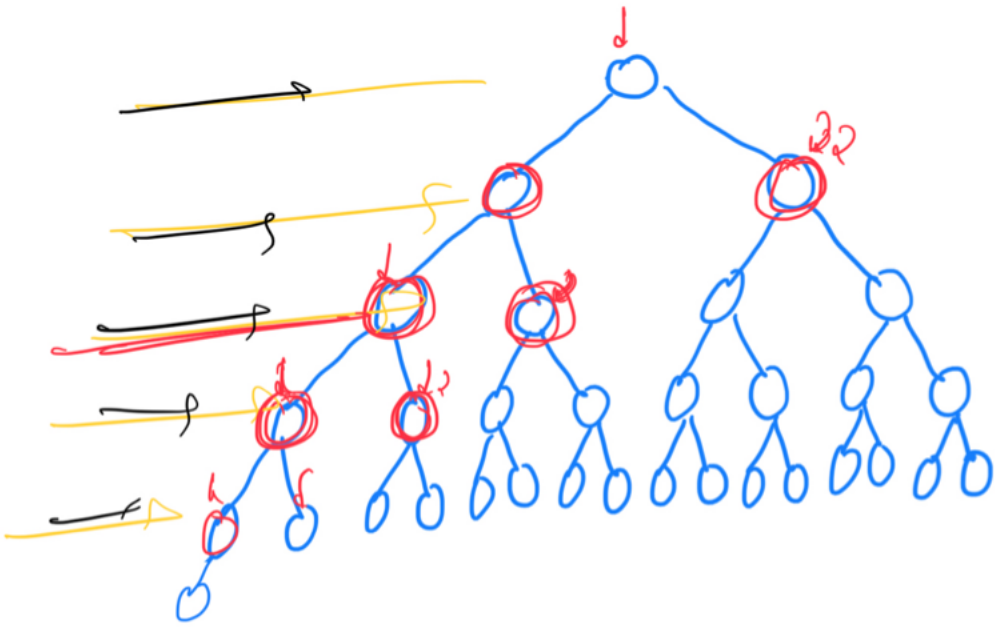
O(N)

Million Dollar Question

```
while(root != NULL)
    root = root.left;
    h++;
}
```

T.C: # Functions calls × T.C per function call

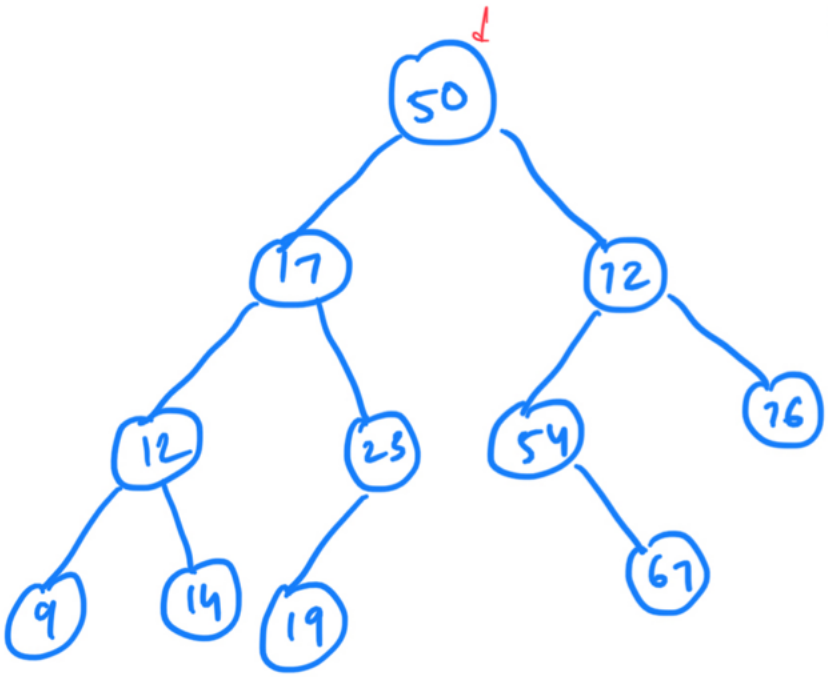
2 logn × O(logn) = O(log²n)



Followup

↓ 2hr → 1.5hr

Question: K^{th} smallest Element in a B.S.T



k = 5 => 23

k = 7 =>

k = 50

Approach1 :

→ Do Inorder Traversal

→ Return the kth element

k iterations are enough

T.C: O(n)

S.C: O(N)

B.S.T

Approach2 : count = 0

inorder(root) {

if(root == NULL) return

inorder(root.left); ?)

count++;

if(count == k)

ans = root.data;

if(count < k)

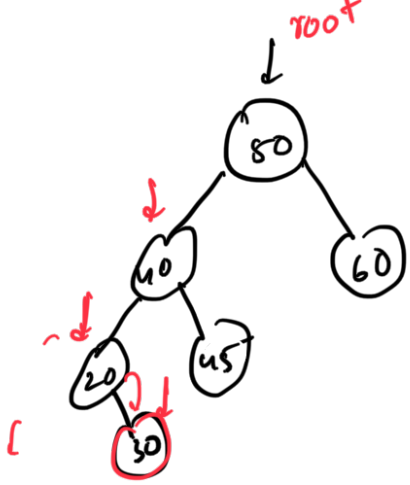
inorder(root.right);

}

k^th small

O(k) = O(N)

Left Root Right



T.C: O(n) → O(k)

S.C: O(H)

↓

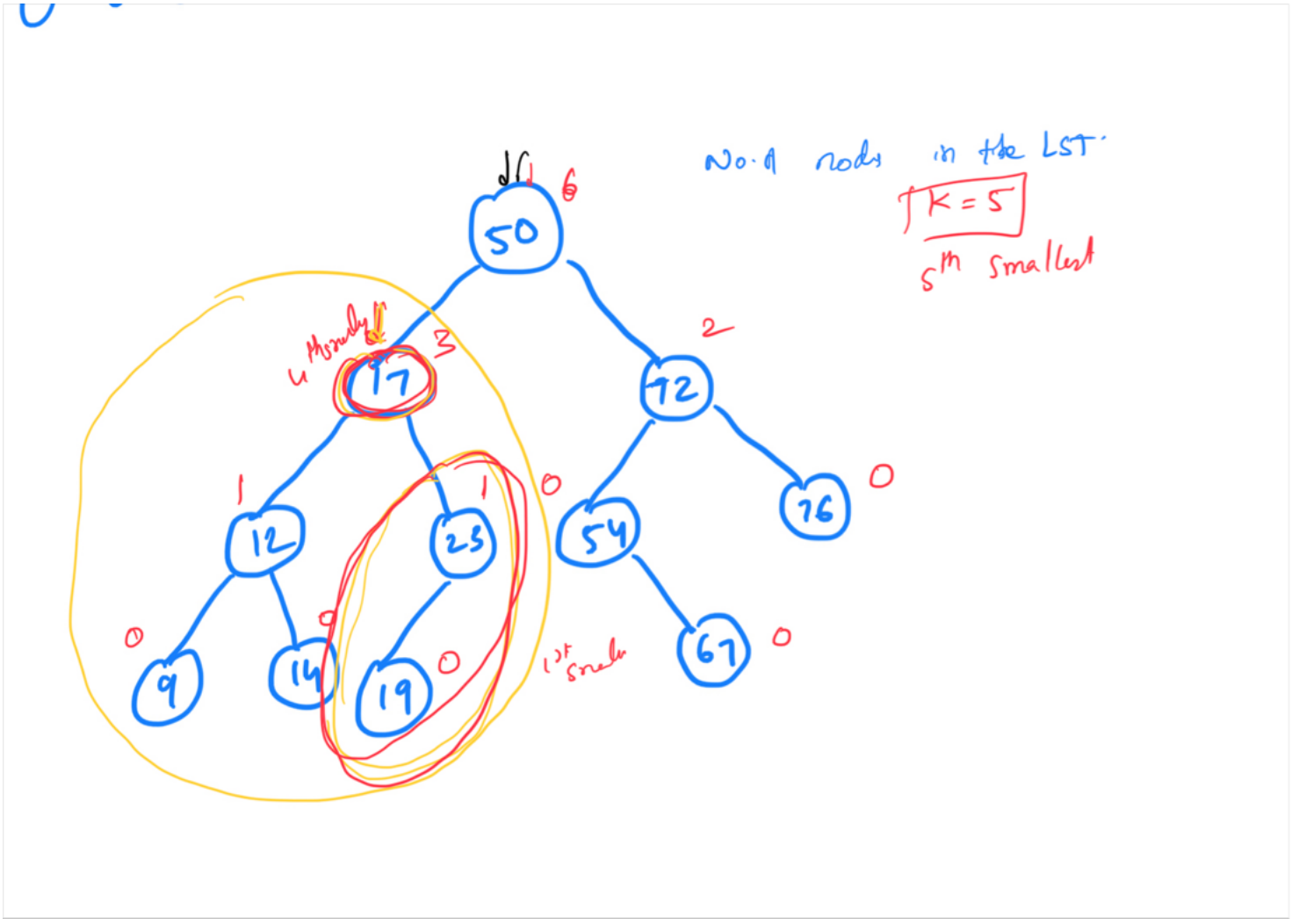
Recursion stack.

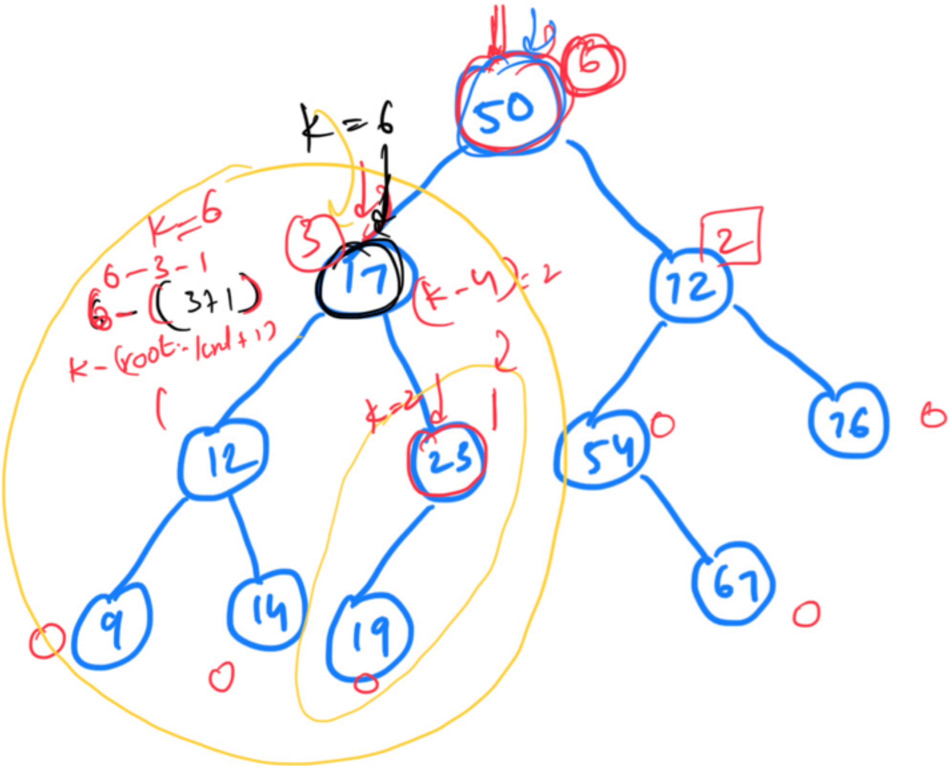
O(k)

```
void Inorder(root)
{
    if(count < k)
        Inorder(root.left)
    count++;
    if(count <= k)
        Inorder(root.left)
}
```

O(K) Recursion

Approach 3 :





K = 7

K = 6

O(n)

1+1 = 2

PostOrder

→ nody(root) = nody(root.left) + nody(root.right) + 1

nody.count =

T.C : O(n)

if(K > N)

Node kth SmallestElement(root, K) {

→ if(root == NULL) return NULL;

if(root.lcount + 1 == K)
return root;

else if(K < root.lcount + 1)
return kthSmallest(root.left, K)

else {
return kth smallest(root.right, ↓);
K - root.lcount - 1

}

}

if(K > N)
NO SENSE

K - (lcount + 1)

K - root.lcount - 1.

K - lcount - 1

T.C : O(H)

Scenario

Multiple queries which ask for kth smallest element

Approach

Store Inorder Traversal in array ⇒ O(N)

O(1)

⇒

Approaches:

Q queries → O(H)

Scenarios:

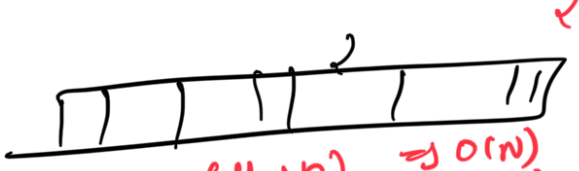
→ Find k^{th} smallest elements

→ Insert

→ Delete

Approach 1:

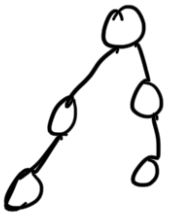
Inorder :



Insertion: O(H + n) → O(N)

Deletion: O(H + n) → O(N)

k^{th} Smallest : O(1)



O(H)

Balanced B.ST

Approach 2 :

Insertion: O(H)

Deletion: O(H)

k^{th} smallest : O(H)

} O(log N)

AVL

Red-Black