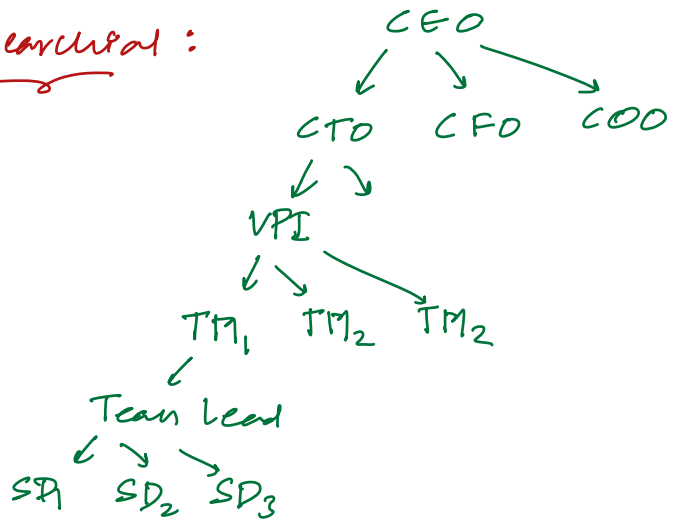
Today's Content:

- → Trees Basics & Terminologies
- → Tree Traversals
- → Size( ) / Height( ) / fillDepth( )
- → Search( )
- → Path Between 2 Nodes
- → level order traversal { if time permits }

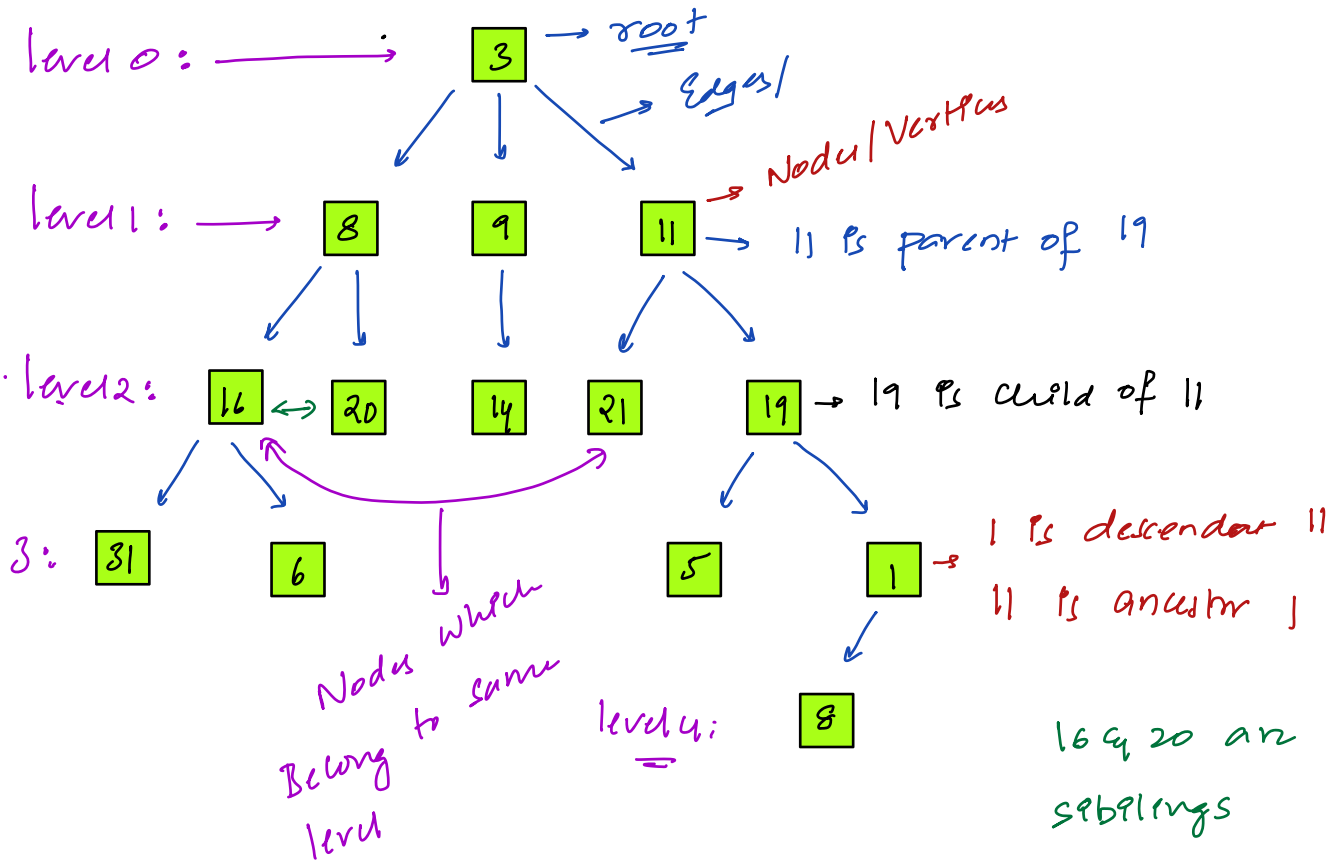
// linear data Structure → arrays / Stacks /

// Hierarchical :



Tree Basics & Terminologies

Tree is a (Hierarchy) structure



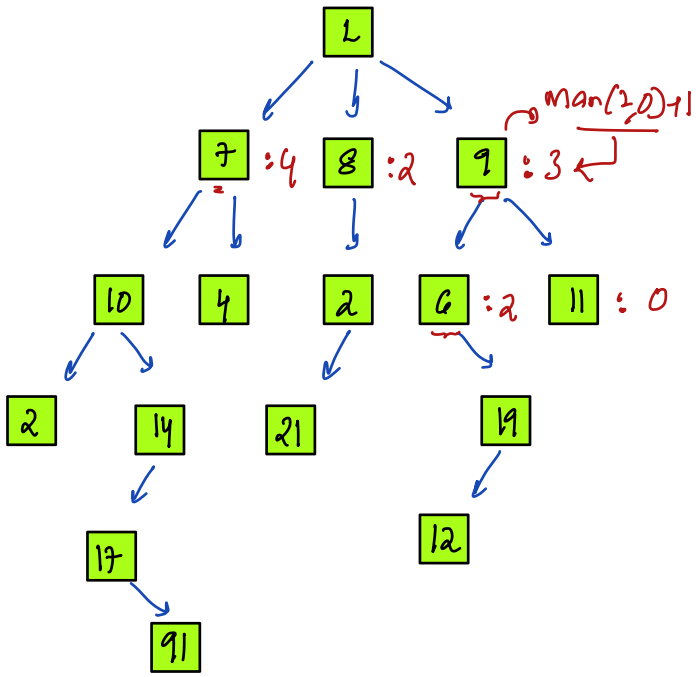
Root : is node with No parent

Leaf Node : Node with No child Nodes

Height of a Node:

Distance from the Node to it's farthest leaf Node

(Distance: No. of Edges)



Height (9) = 3

Height (10) = 3

Height (7) = 4

Height (91) = 0

Height (1) = 5

obs:

Height (Node) = 1 + max (child Node heights)

Edge Case: 6 = 1 + max (child Node) = 1 + 0 (no child nodes)

Height of Tree = Height (root Node)

Height of leaf Node = 0

Depth of a Node :

Distance of a Node from Root Node

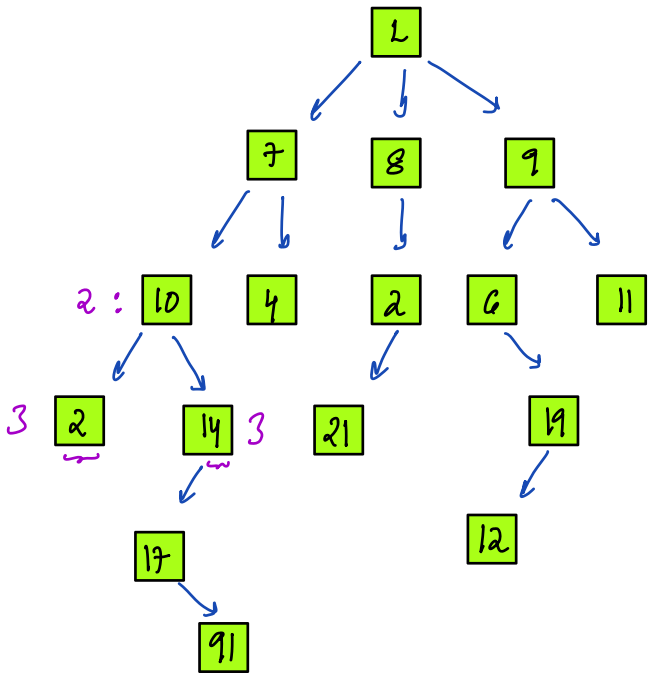
Depth(8) = 1

Depth(10) = 2

Depth(21) = 3

Depth(19) = 3

Depth(1) = 0



obs1

depth (Node) = d

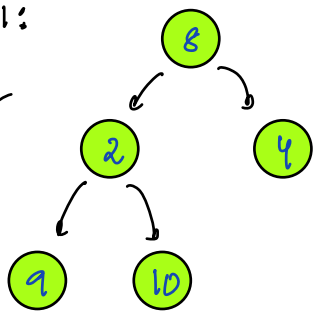
↳ depth of it's child Node = d+1

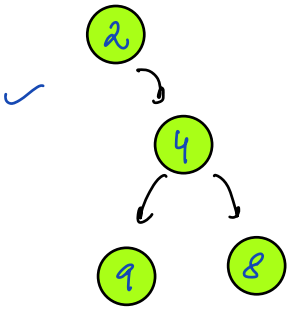
obs2: depth (Root Node) = 0

Binary Tree: For every node, no: of child nodes ≤ 2

0 or 1 or 2

Eg1:

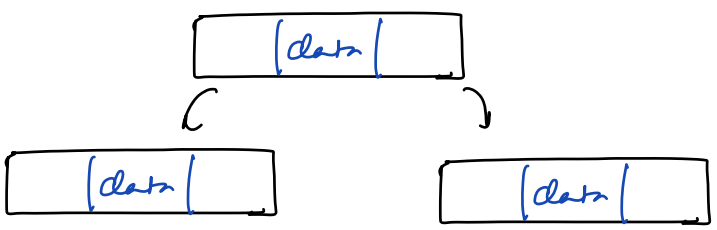


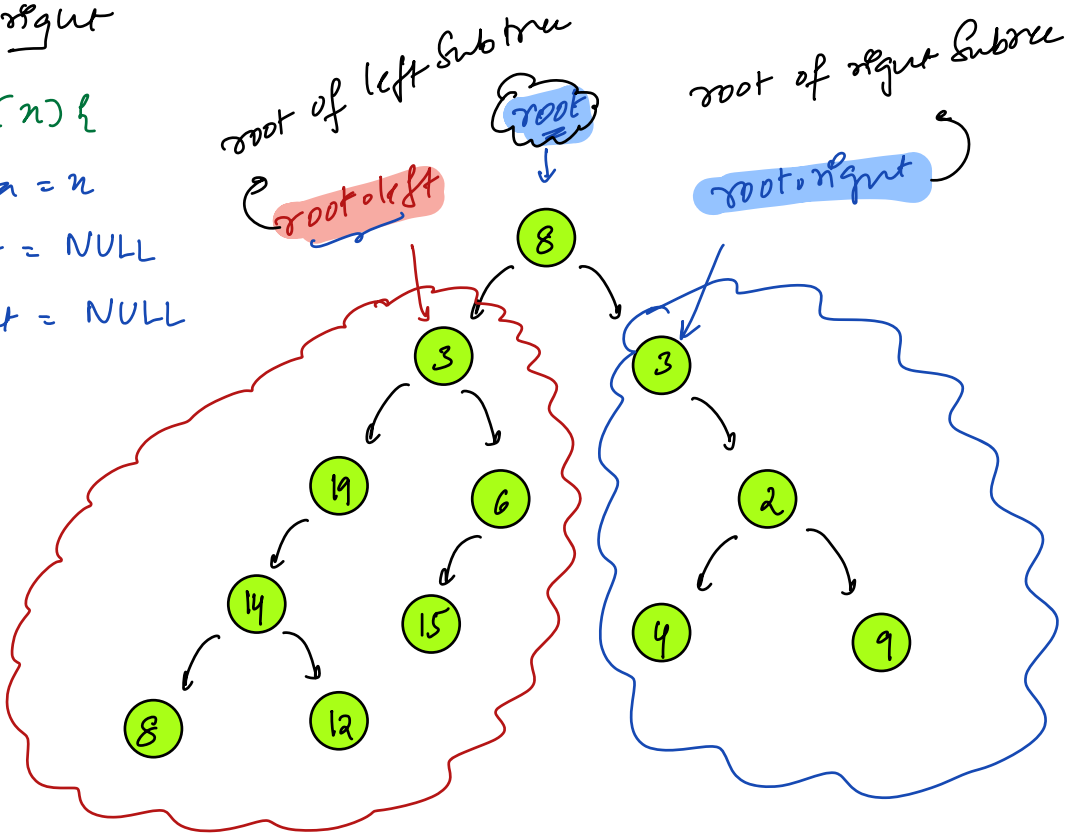


: 0 child nodes

(Node Structure)

```
Class Node {
    int data;
    Node left
    Node right

    Node(n) {
        data = n
        left = NULL
        right = NULL
    }
}
```





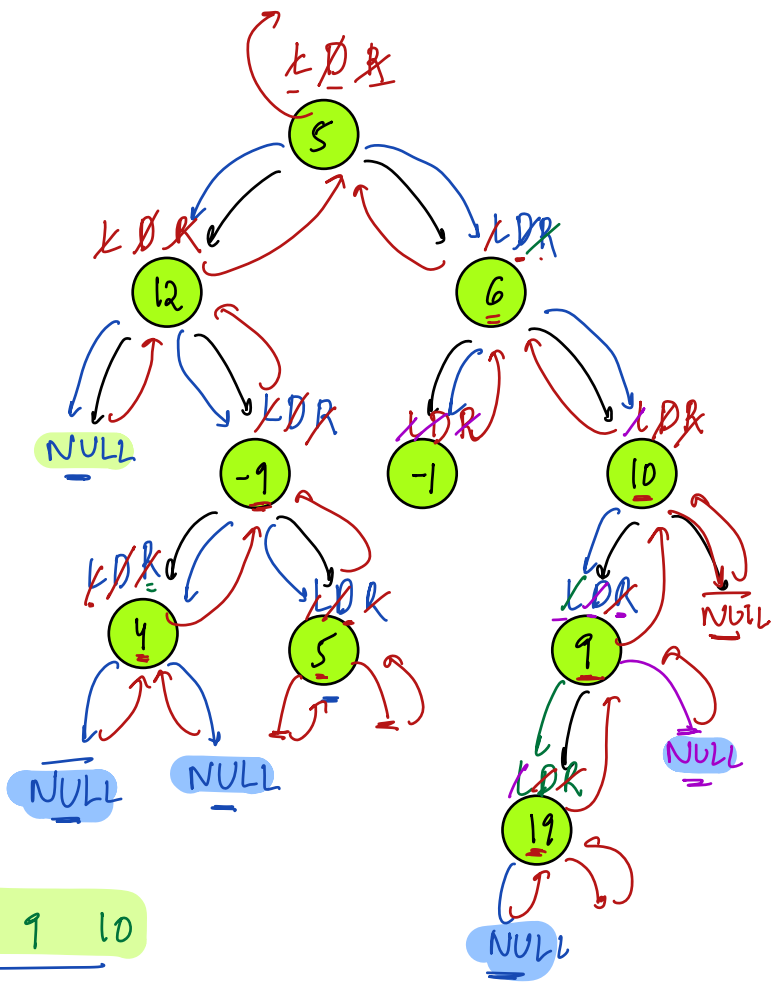
// Recursion Basics :

- → Assumption : decide what your function q assume it works.
- → Main Logic : Solving problem using subproblem.
- → Base Condition : When to stop.

// DS → (Trave / Iterate)

// Tree Traversals :

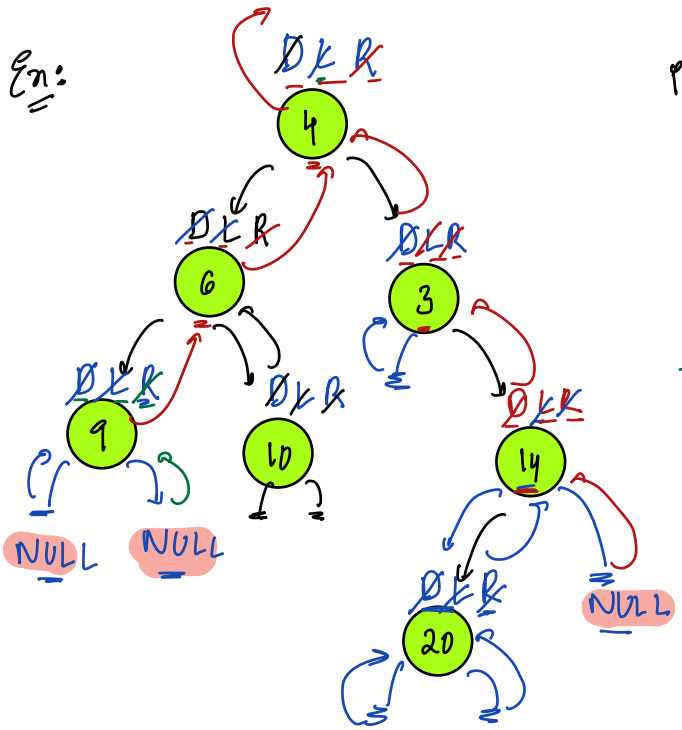
1) preorder : D L R
2) Inorder : L D R
3) postorder : L R D



Inorder:

12 4 -9 5 5 -1 6 19 9 10

Eg:



Inorder: 9 6 10 4 3 20 14

preorder: 4 6 9 10 3 14 20

postorder: TODO

// Pseudo code:

Ass: (print Entire Tree in Inorder)

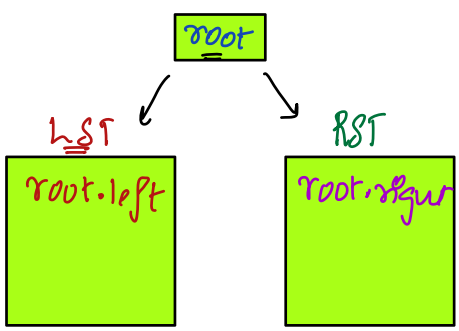
```
void Inorder(Node root){
    1  if(root == NULL) {return;}
    2  Inorder(root.left)
    3  print(root.data)
    4  Inorder(root.right)
}
```

preorder: 1 3 2 4

Inorder: 1 2 3 4

postorder: 1 2 4 3

Idea:



Idea:

Inorder (Tree):

→ print Entire LST in Inorder

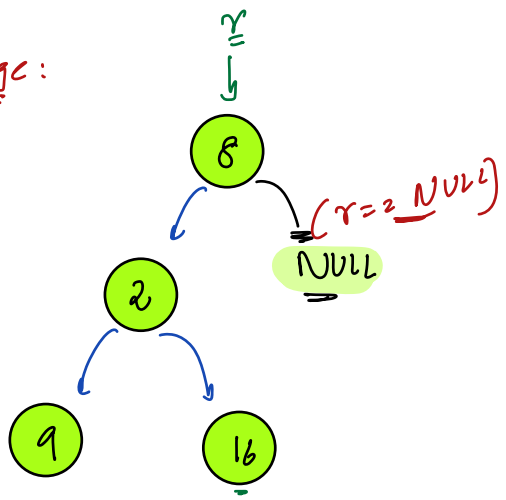
→ print (root.data)

→ print Entire RST in Inorder

10:30 pm (break)

```
void Inorder(Node root){
    (NULL pointer Exception : if root == NULL)
    if (root->left == NULL && root->right == NULL){
        print(root.data) return
    }
    Inorder(root.left)
    print(root.data)
    Inorder(root.right)
}
```

Edge:

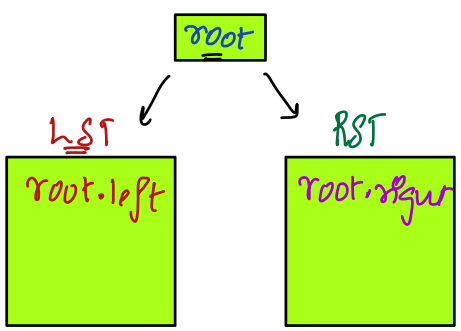


Inorder: 9 2 16 8

Total node = 1 + (Total node in LST) + (Total node in RST)

(Total node in LST) → size(root.left)

(Total node in RST) → size(root.right)

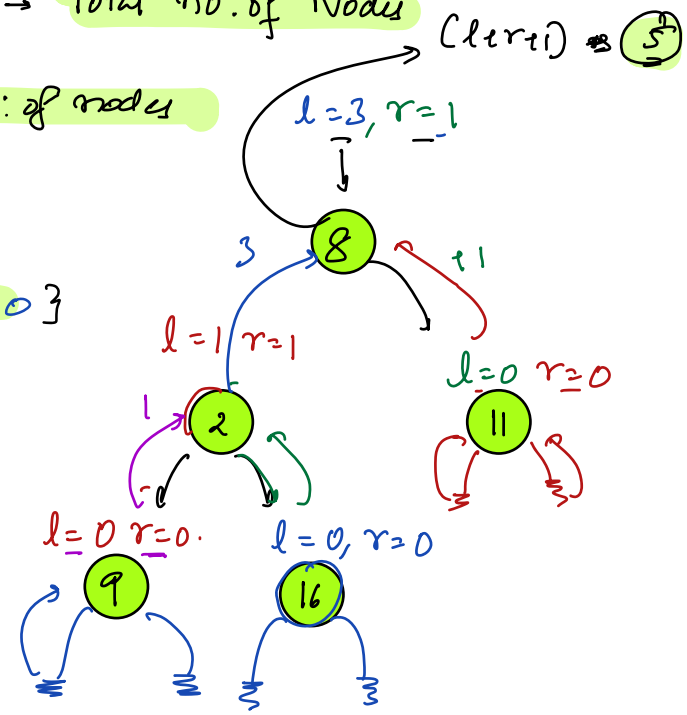


//

Calculate size of the Tree :→ Total no: of Nodes

Ass: given root node, return no: of nodes

```
int size(Node root){
    if(root == NULL){return 0}
    int l = size(root.left)
    int r = size(root.right)
    return (l+r+1)
}
```

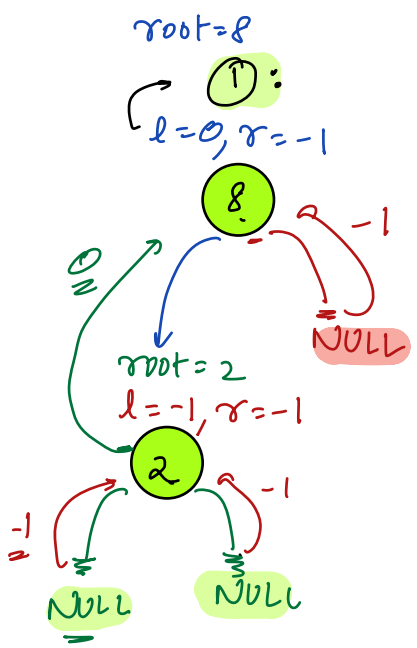


// Calculate Height of Tree

Ass: given root node, get height of Tree

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

to get correct ans?



// Height of leaf Node = 0

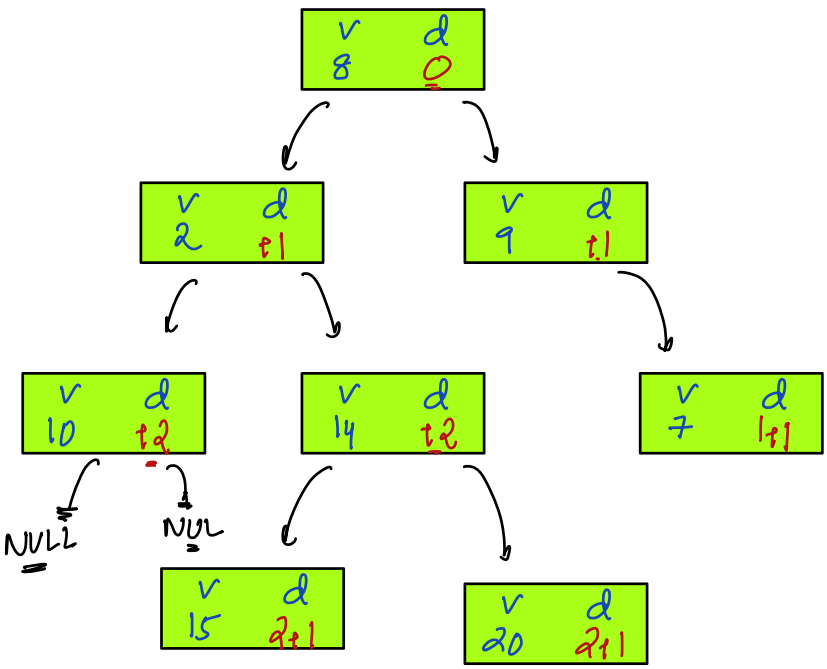
// Height(Node) = 1 + Max (Height of LST, Height of RST)

// Fill Depth

```
Class Node {
    Node left
    Node right
    Int val
    Int depth
}
```

// BT, value are added

// for every node you need to add depth of node



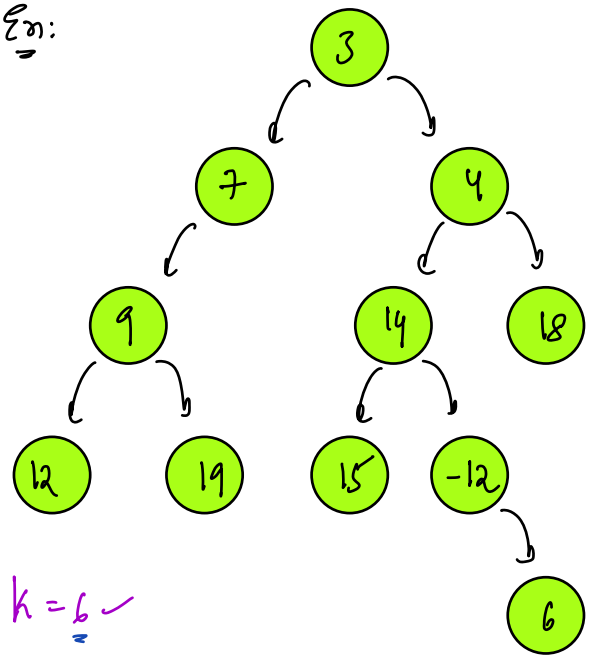

```
void fillDepth(Node root, Int d) {
    if(root == NULL) { return }
    root.depth = d;
    fillDepth(root.left, d+1)
    fillDepth(root.right, d+1)
}
```

// fillDepth(root, 0)

// Initially depth of root Node = 0

// Given a B.T which contains all unique values, Search if there exists a k in BT

Eg:

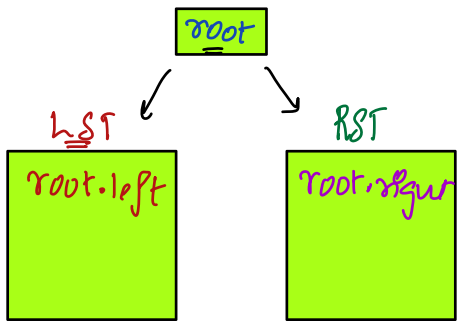


k = 6 ✓

k = 19 ✓

k = 25 ✗

Ans: If k is present in Tree return True else return False

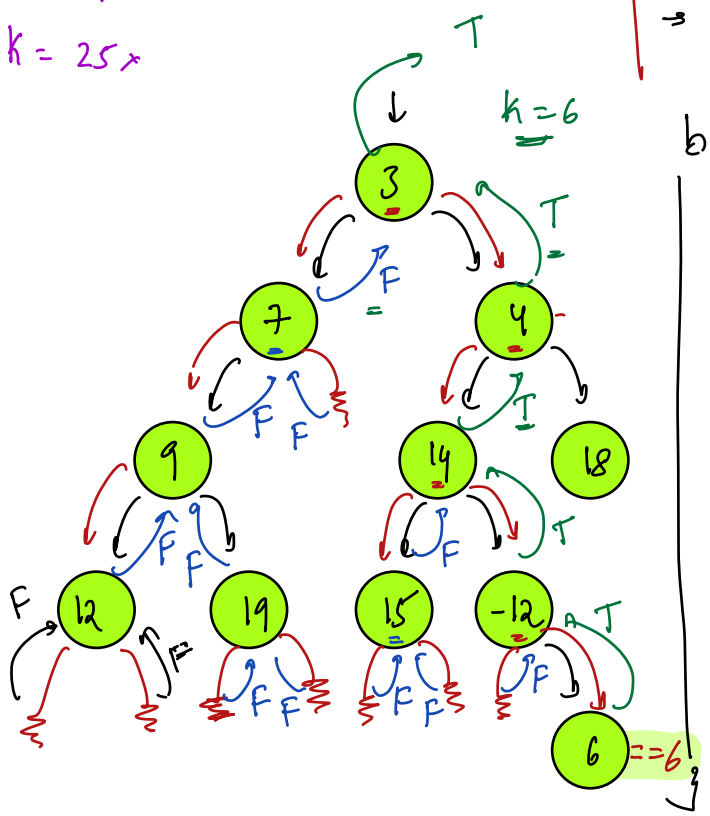


→ root.data == k

→ Check if k is in LST : Subpro

→ Check if k is in RST : Subpro

→ If not present in above check False

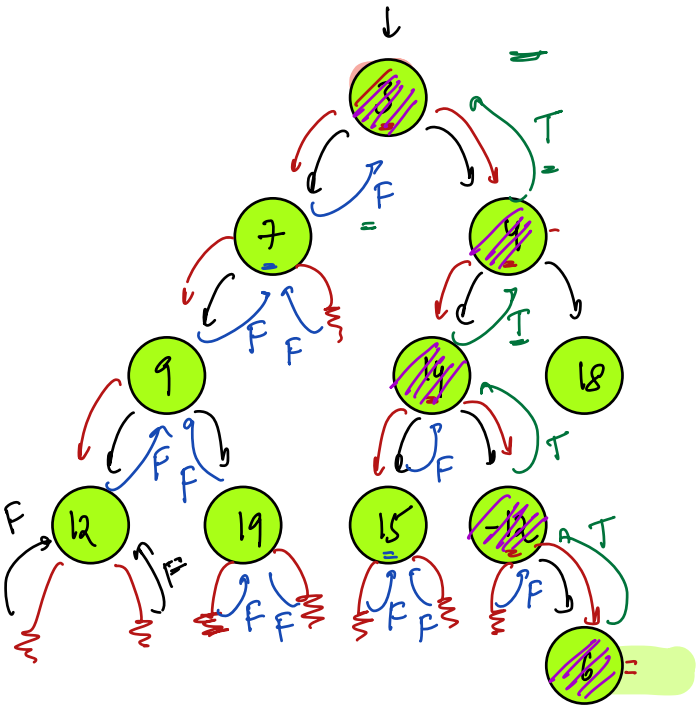



```
bool check(Node root, int k){
    if(root == NULL){return false}

    if(root.data == k){
        return True
    }

    return
        {check(root.left, k) ||
         check(root.right, k)}
}
```

// Given a B.T which contains all unique values,

get the path of Node k

↳ (k is present)



k == 6

path: { You need to return address }

3 → 4 → 14 → -12 → 6

Nodes which return true

They belong to the path

// Storing Paths //

```
li<Nodes> li; → global variable

bool check(Node root, int k,
{
  if(root == NULL){return false}

  if(root.data == k){
    li.add(root); return True;
  }

  if( check(root.left, k) || check(root.right, k) ){
    li.add(root)   } root also belongs to path
    return True
  }
  else return False
}
```

// now you have your path.

// code:

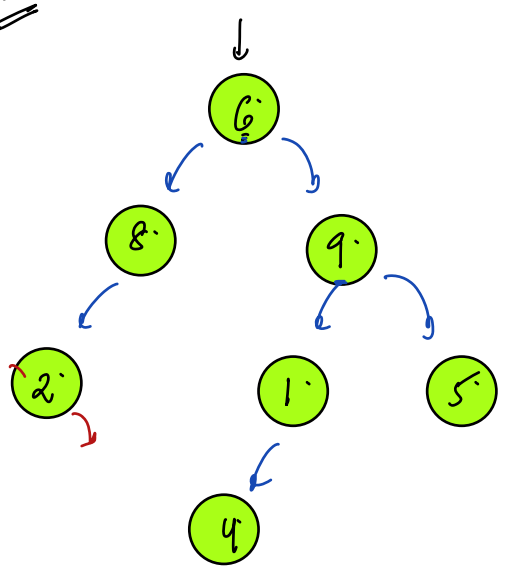
```
li.add(6)
li.add(-12)
li.add(14)
li.add(4)
li.add(3)
```

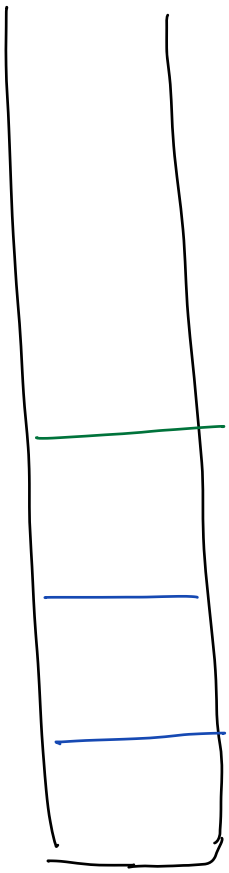
li: | 6 | -12 | 14 | 4 | 3 |

→ leaf to root

reverse, root to leaf

Doubt?





// ArrayList<Node> li

li.add(4)
li.add(1)
li.add(9)
li.add(6)

li: [4 | 1 | 9 | 6]

→ node to root
→ reverse to get
root → node