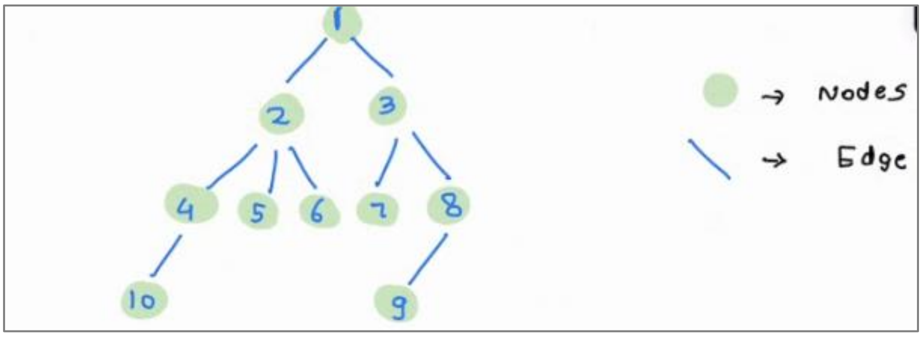
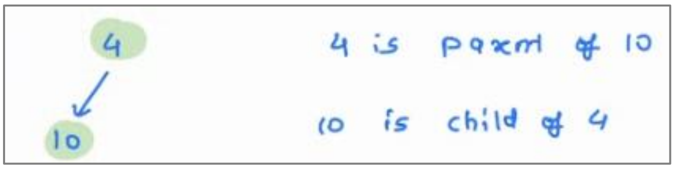
Trees1: Structure & Traversal

Nomenclature
Binary Tree
Traversals
Equal tree partition
What is a tree?

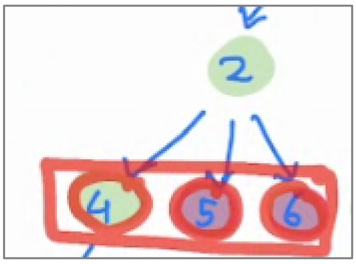
So far, we have seen LL, stacks, queue, arrays → These are Linear Data Structure (DS)

Tree → Non-linear DS → Hierarchical DS

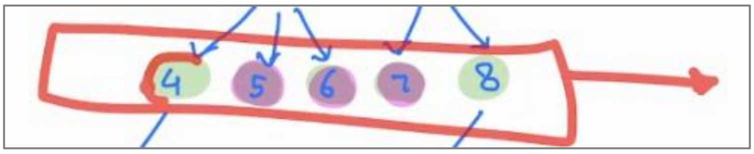




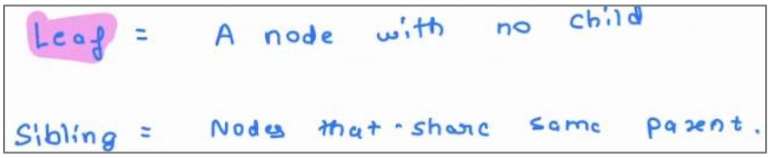
Here 5, 6,7,10 & 9 has no children → A node with no child is LEAF.



Siblings – 4, 5 & 6 are siblings

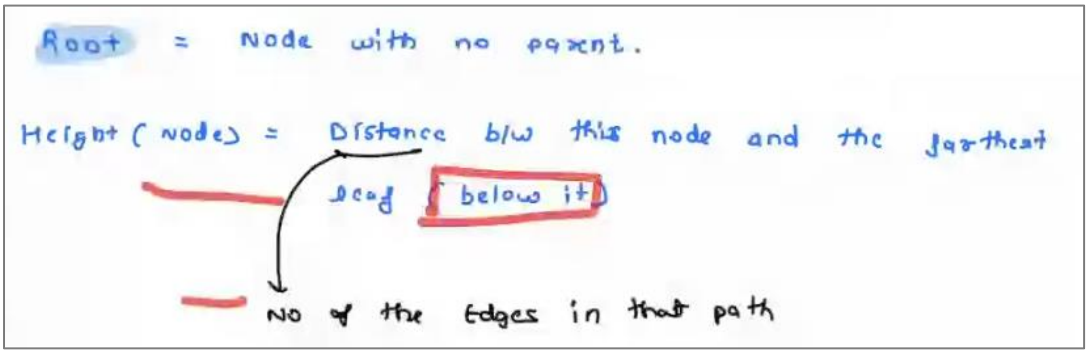


Cousins – Everyone on same level are cousins, even if their grandparents are different



Leaf = A node with no child

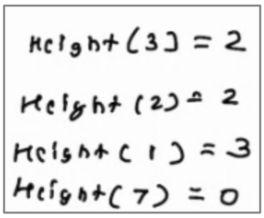
Sibling = Nodes that share same parent.



Root = Node with no parent.

Height (node) = Distance b/w this node and the farthest leaf (below it)

No of the Edges in that path

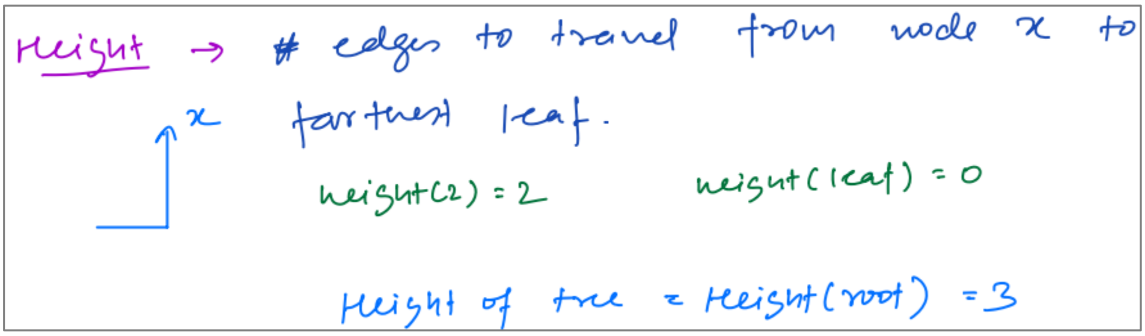


Height(3) = 2

Height (2) = 2

Height (1) = 3

Height(7) = 0



Height → # edges to travel from node x to farthest leaf.

x

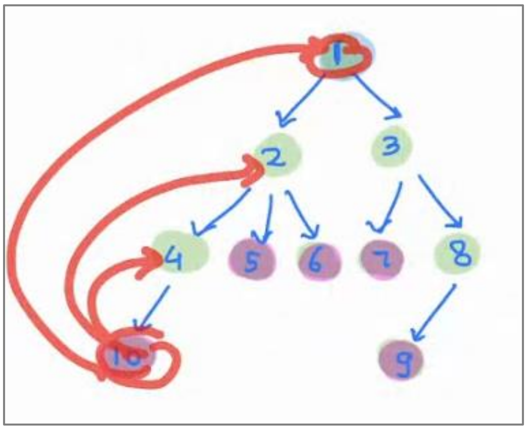
height(2) = 2

height(leaf) = 0

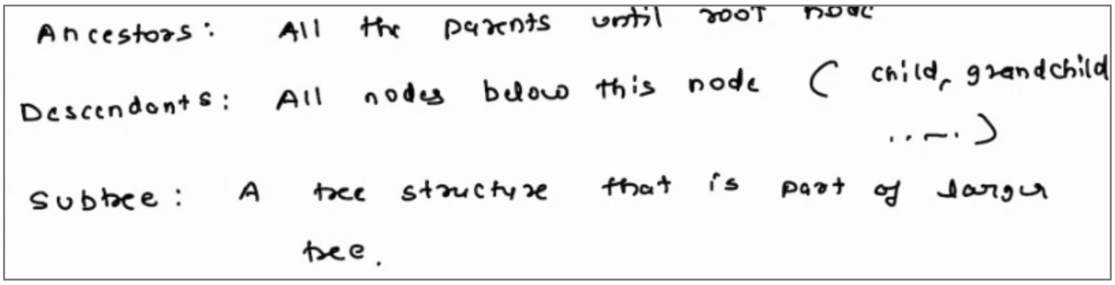
Height of tree = Height(root) = 3

Height of leaf = 0

Height is measured from the bottom to the top. Depth is measured from the top to bottom



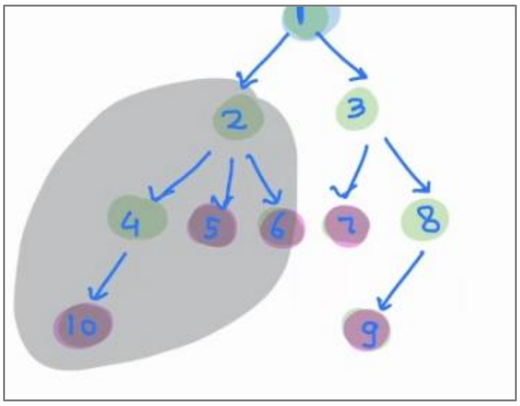
Height of the tree = Height of the root node

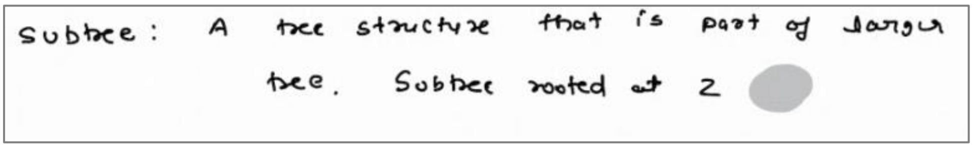


Ancestors: All the parents until root node

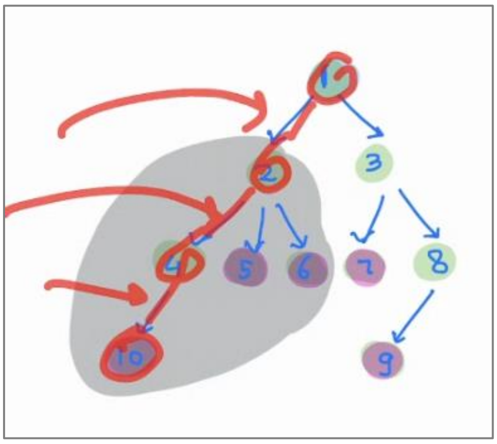
Descendants: All nodes below this node (child, grandchild ...)

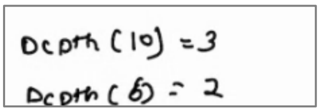
Subtree: A tree structure that is part of larger tree.





Subtree: A tree structure that is part of larger tree. Subtree rooted at 2 ●



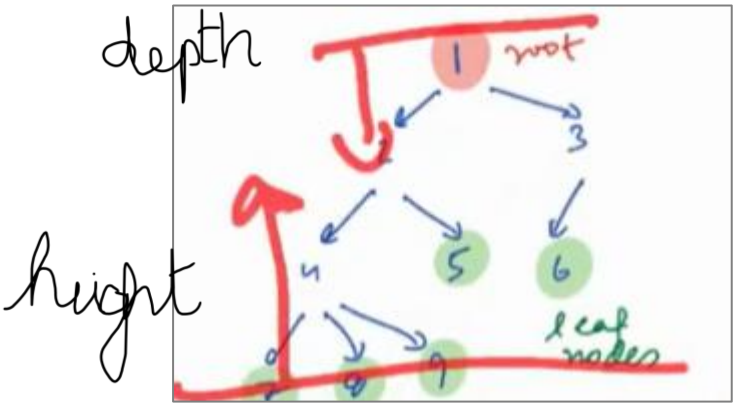


Depth(10) = 3

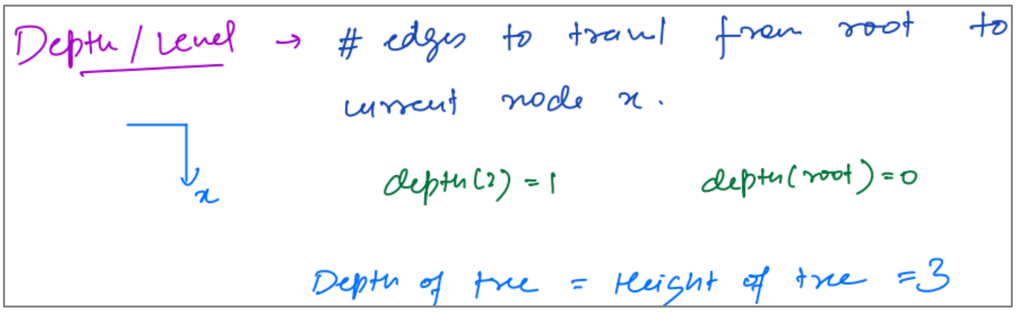
Depth(6) = 2

Depth of the entire tree: Depth from the farthest leaf node

Depth: Distance from root



Depth of root node =0

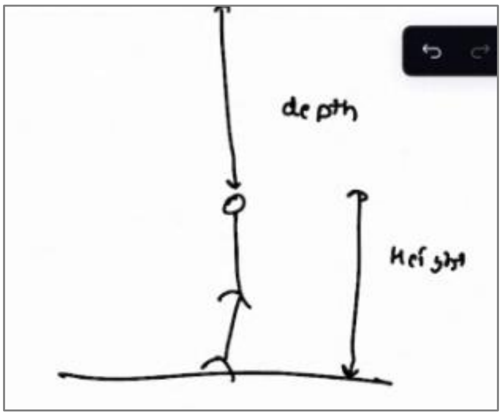


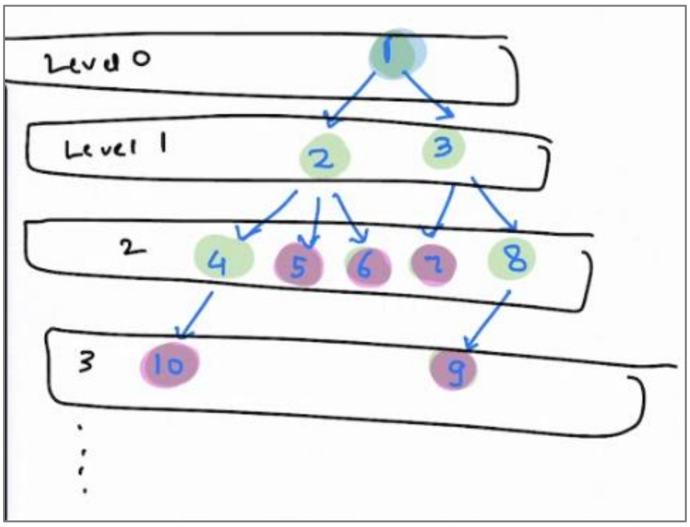
Depth / Level → # edges to travel from root to current node x.

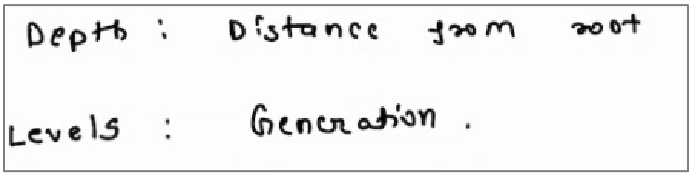
x

depth(2) = 1　　depth(root) = 0

Depth of tree = Height of tree = 3

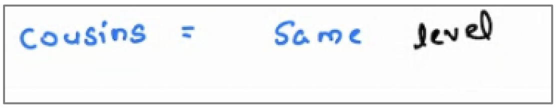






Depth : Distance from root

Levels : Generation .

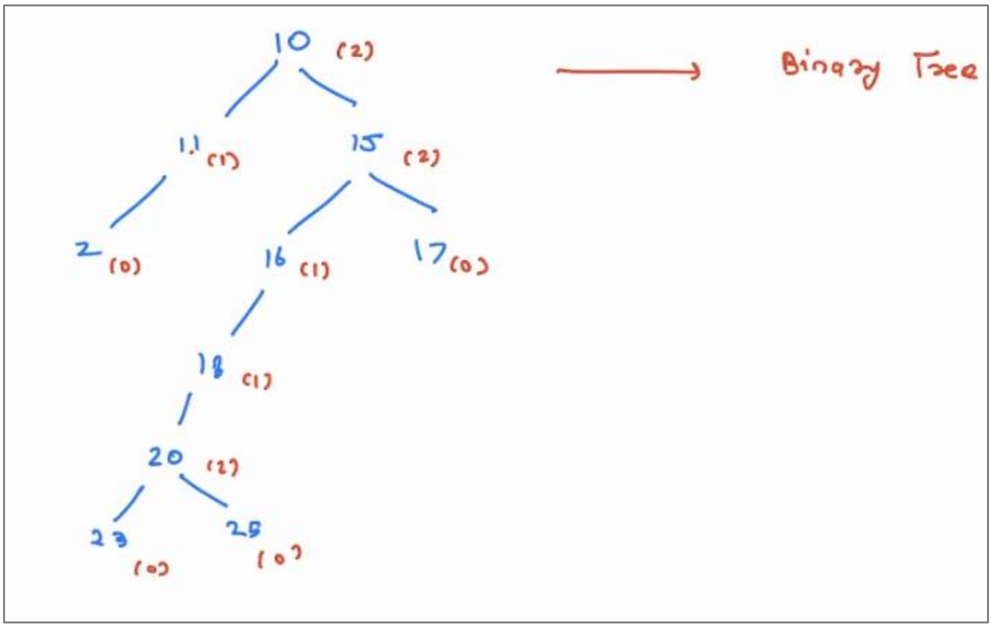


Cousins = Same level

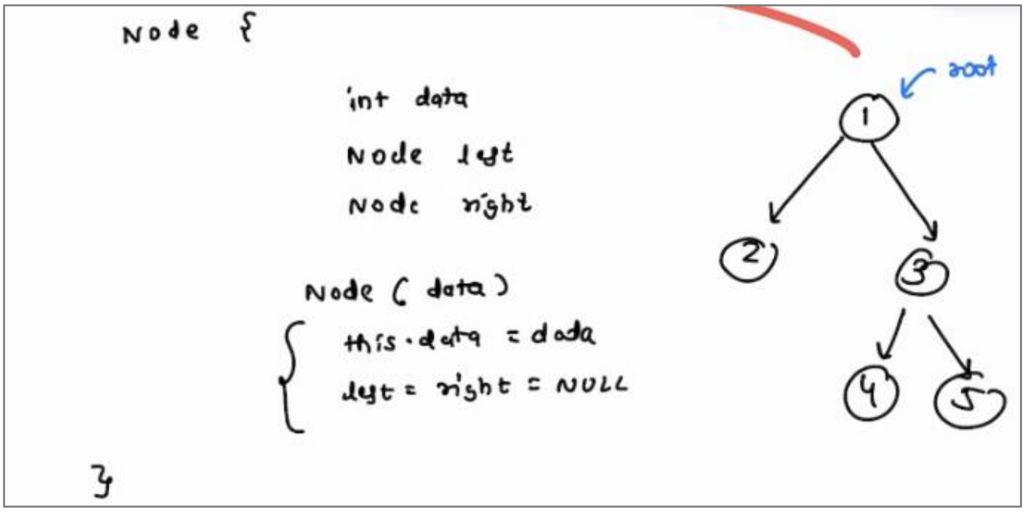
Binary Tree

Each node having atmost 2 child.

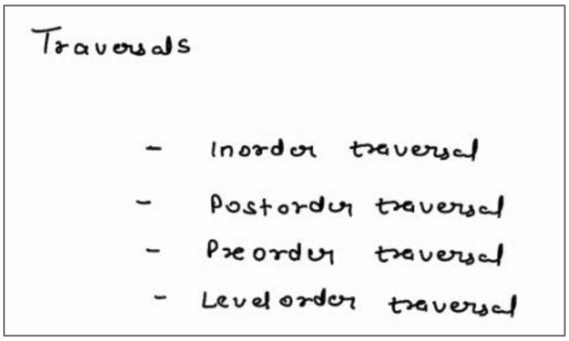
#of child can be 0, 1 or 2.



Every node has atmost 2 child



```
Node {

        int data
        Node left
        Node right

      Node ( data )
    {   this.data = data
        left = right = NULL

}
```



Traversals

- Inorder traversal
- Postorder traversal
- Preorder traversal
- Level order traversal

Inorder Traversal

L = Left subtree

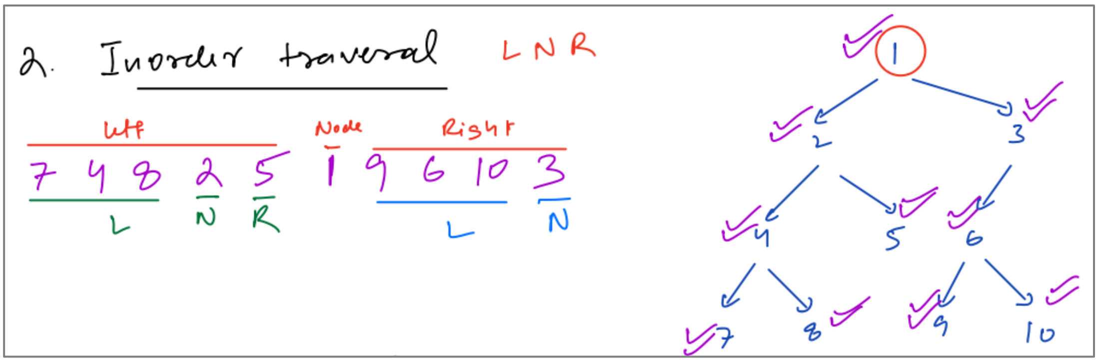
R = right subtree

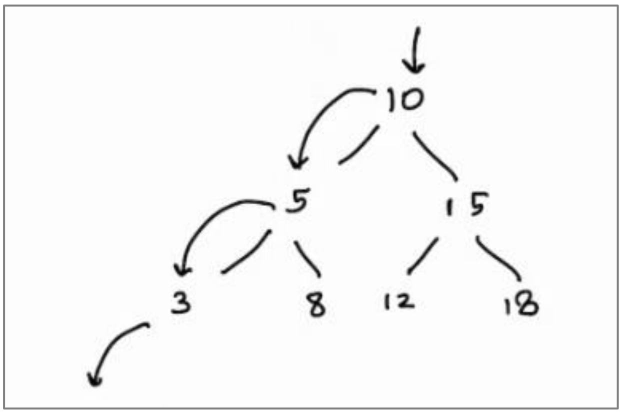
N = node

LNR for each Node

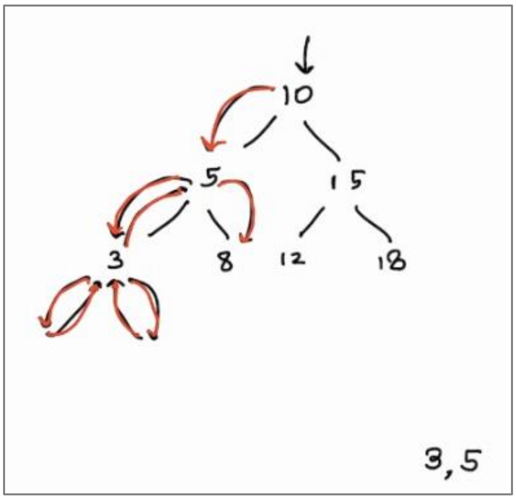
Inorder Traversal

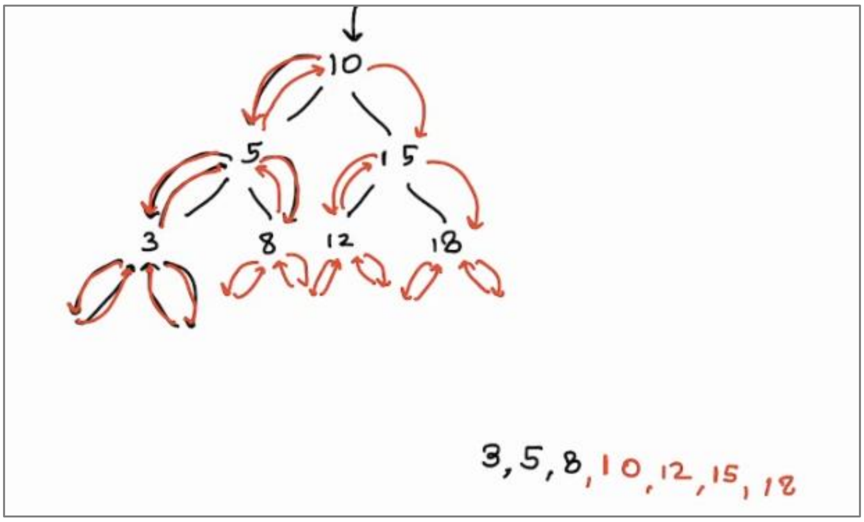
Traverse LNR for each node

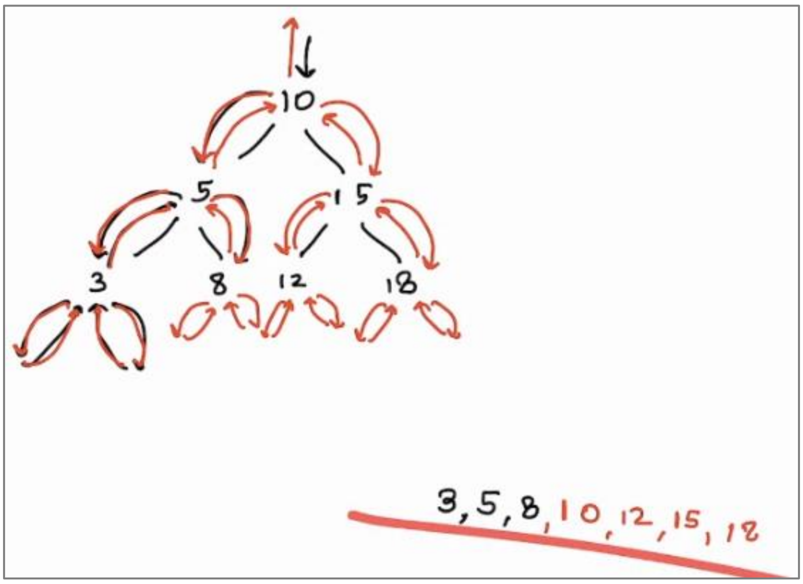




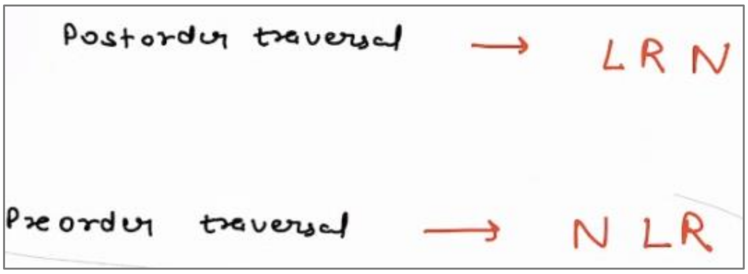
First traverse in the left then traverse right







This is how inorder traversal of the tree look like



Postorder traversal → L R N

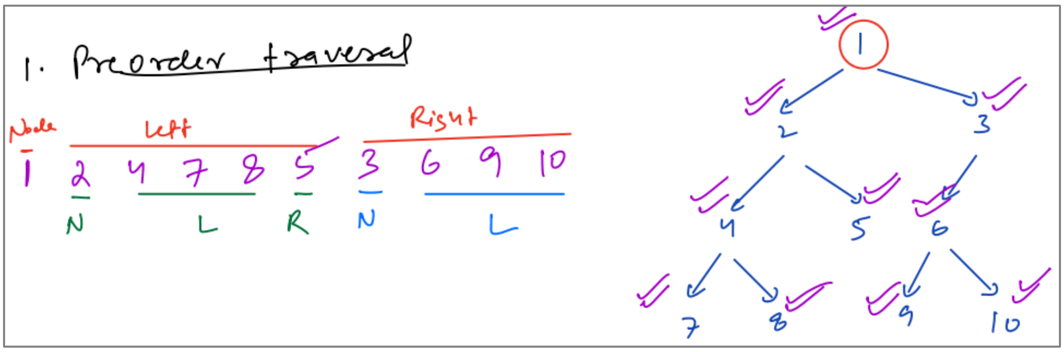
Preorder traversal → N L R

Postorder → Node will be post

Preorder → Node will be pre

Inorder → Node will be in between

Pre-order Traversal

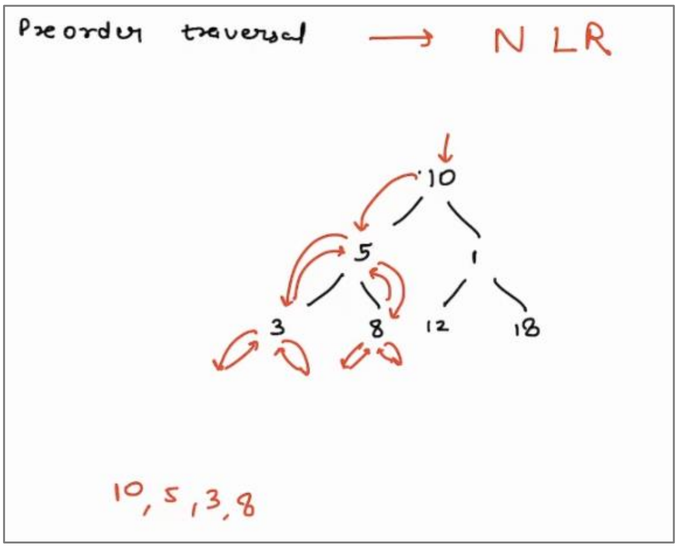


1. Preorder traversal

Node | Left | Right

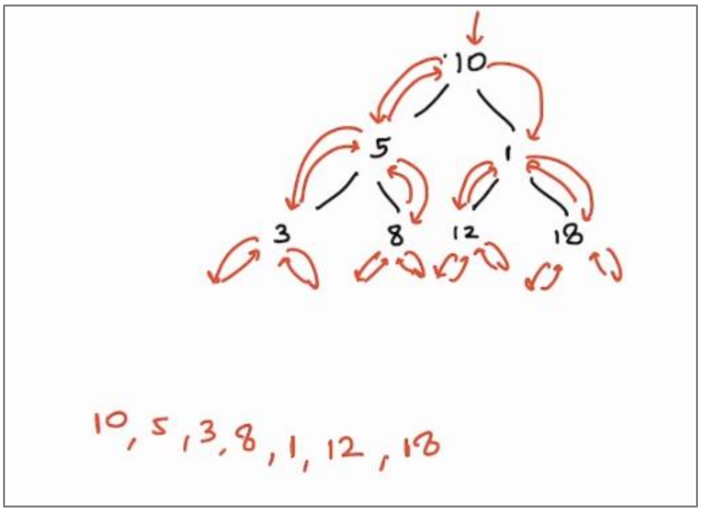
1 2 4 7 8 5 3 6 9 10

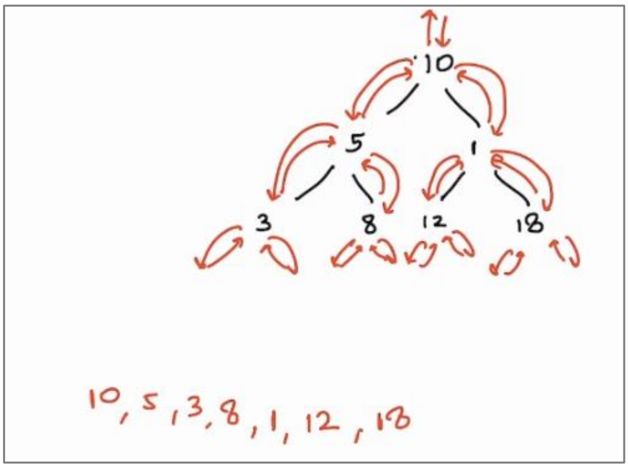
N L R N L



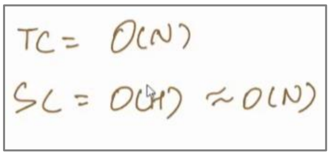
Preorder traversal → N L R

10, 5, 3, 8





This is Preorder traversal of the entire tree

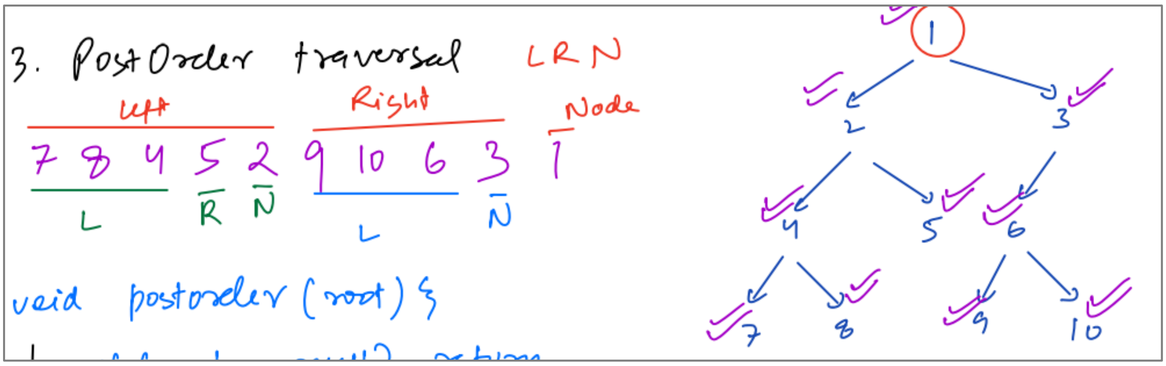


TC = O(N)

SC = O(H) ≈ O(N)

Depth of (recursive) stack = height of tree When you go to the end of the tree, → height of the tree → max elements in the stack → when you print an element, you delete elements form the stack as well

Post Order Traversal



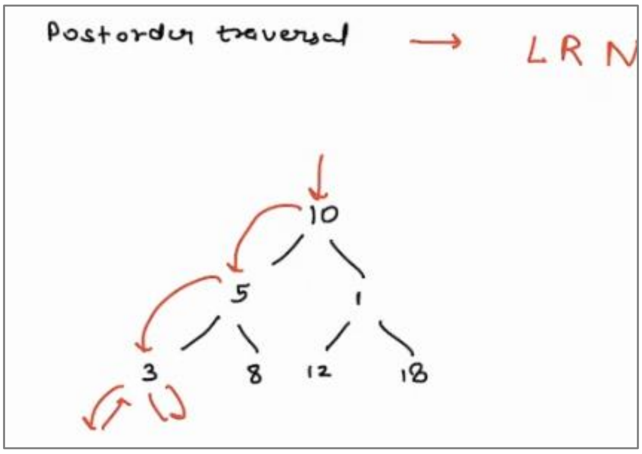
3. PostOrder traversal LRN

Left — Right — Node

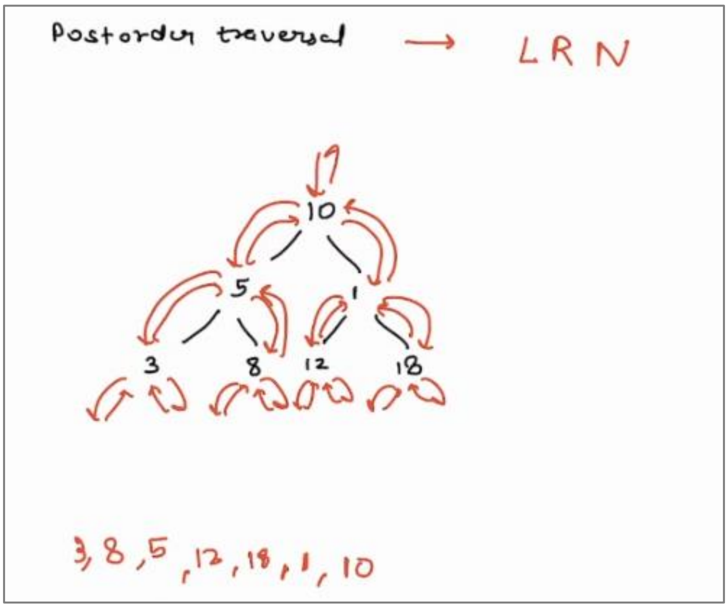
7 8 4 5 2 9 10 6 3 1

L $\bar{R}$ $\bar{N}$ L $\bar{N}$

void postorder(root) {

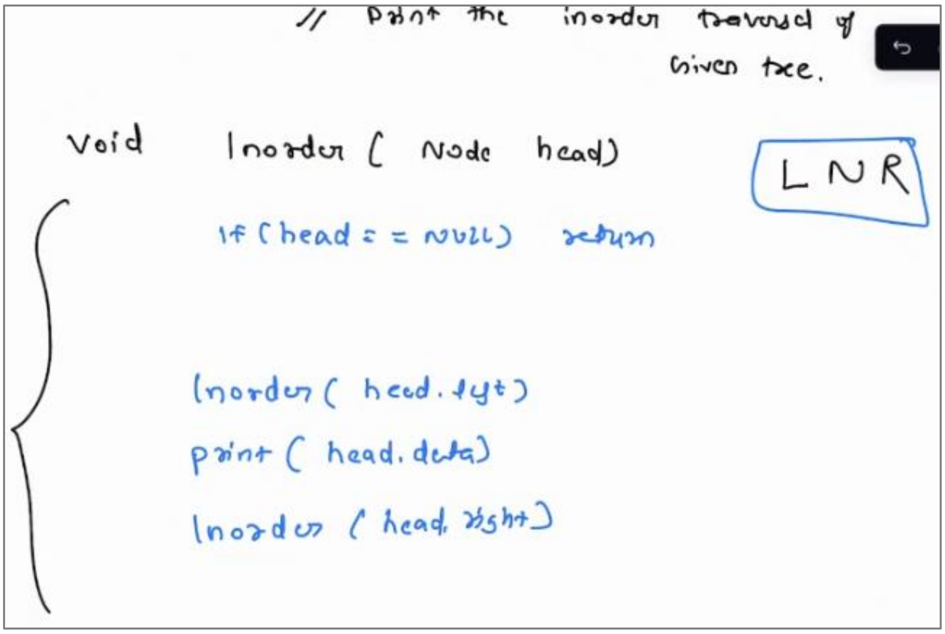


Postorder traversal → LRN



Postorder traversal → LRN

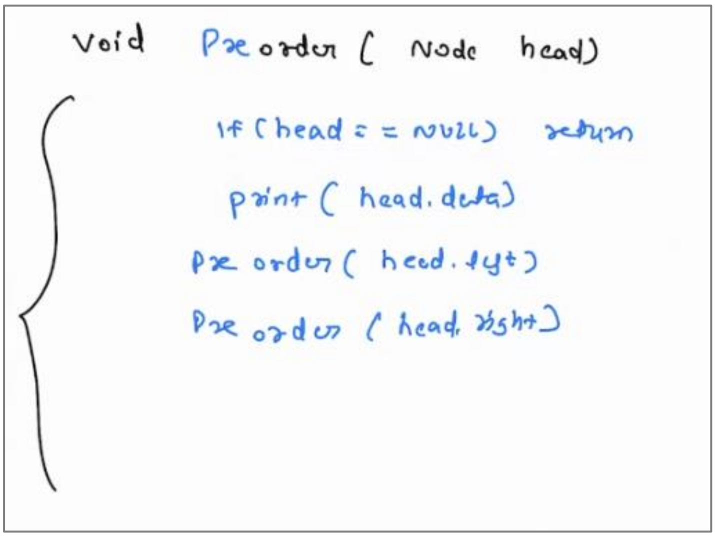
3, 8, 5, 12, 18, 1, 10



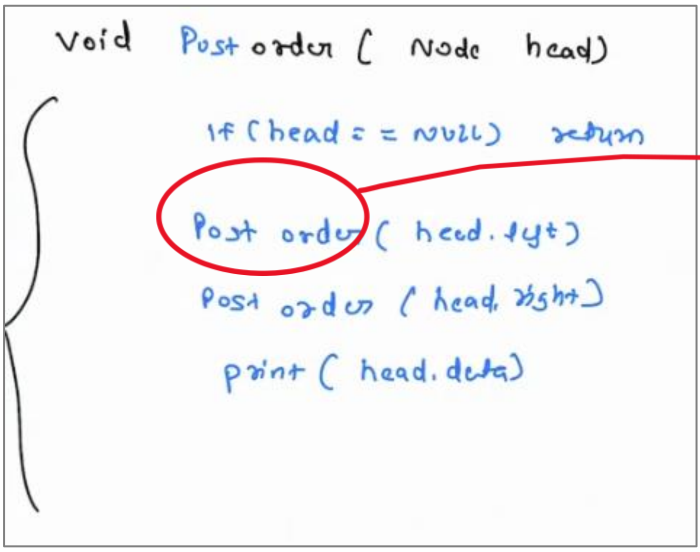
// Print the inorder traversal of Given tree.

LNR

```
void Inorder ( Node head)
    if (head == NULL) return

    Inorder ( head.left)
    print ( head.data)
    Inorder ( head.right)
```



```
void Preorder ( Node head)
    if (head == NULL) return
    print ( head.data)
    Pre order ( head.left)
    Pre order ( head.right)
```

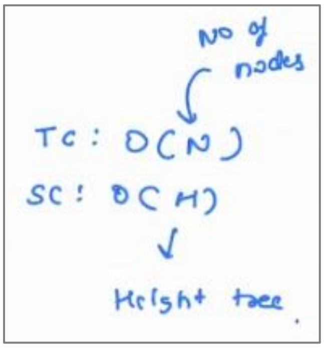


```
void Post order ( Node head)
    if (head == NULL) return
    Post order ( head.left)
    Post order ( head.right)
    print ( head.data)
```

recursion calls

Each recursive call will make TC as 3 *N

TC → iterating the each node only once → O(N) → no. of nodes



No of nodes

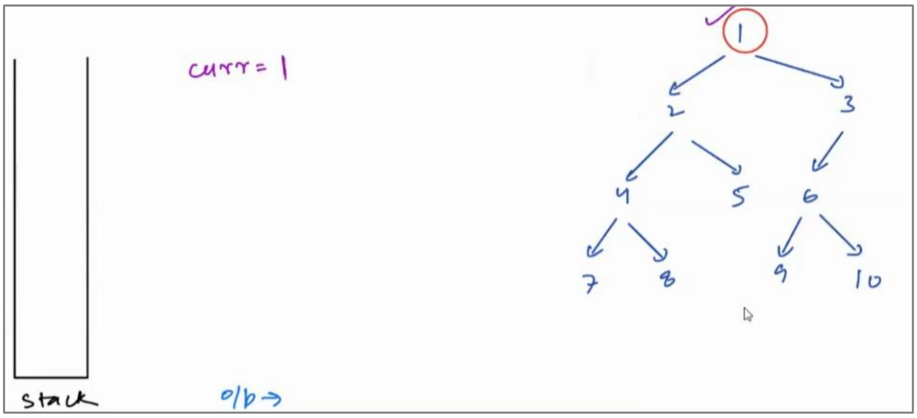
TC : O(N)

SC : O(H)

Height tree.

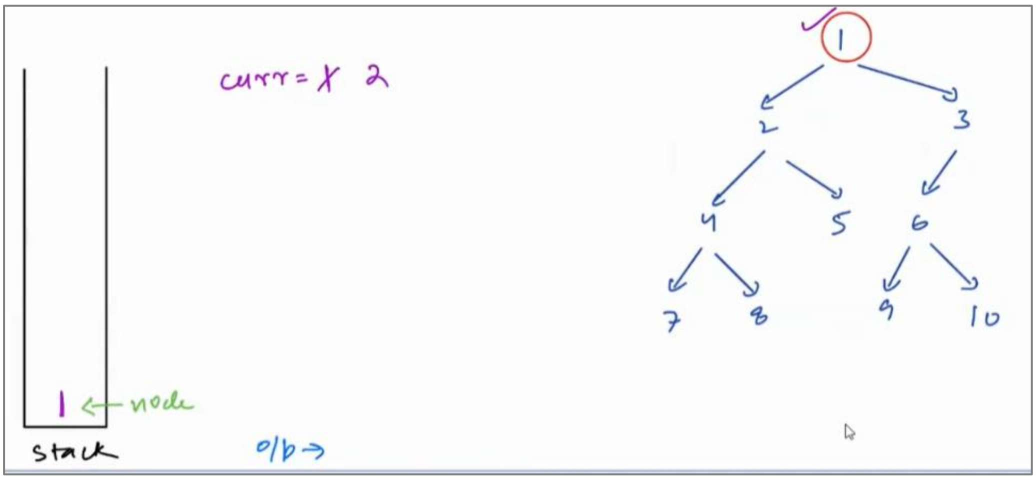
SC would be O(N) in worst case

Iterative Inorder Traversal

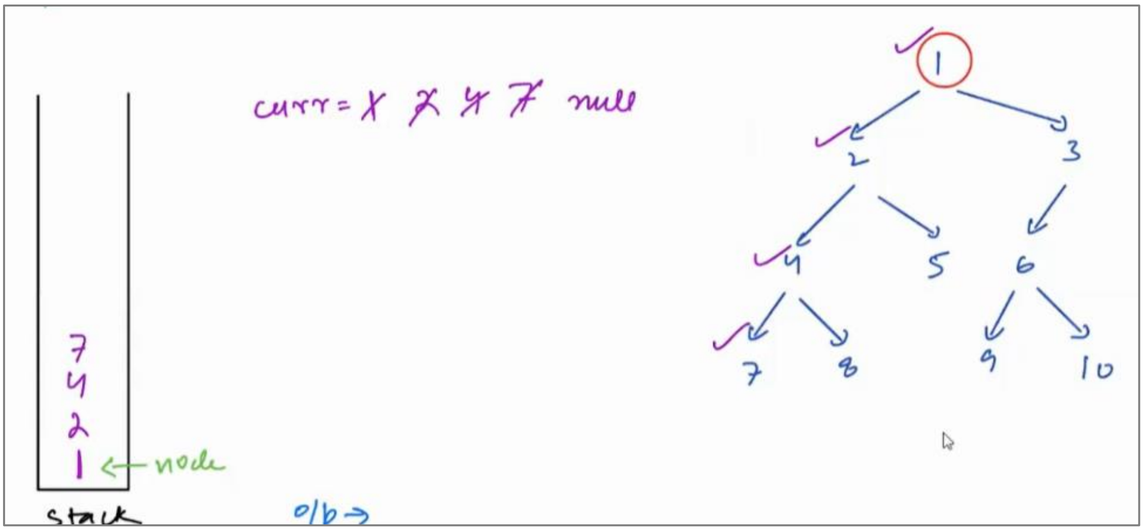


Initially curr = 1, now check whether left of 1 exist or not, here in this case yes now update the value of curr to 2 and you need to store 1 somewhere so that I can go to 1 again.

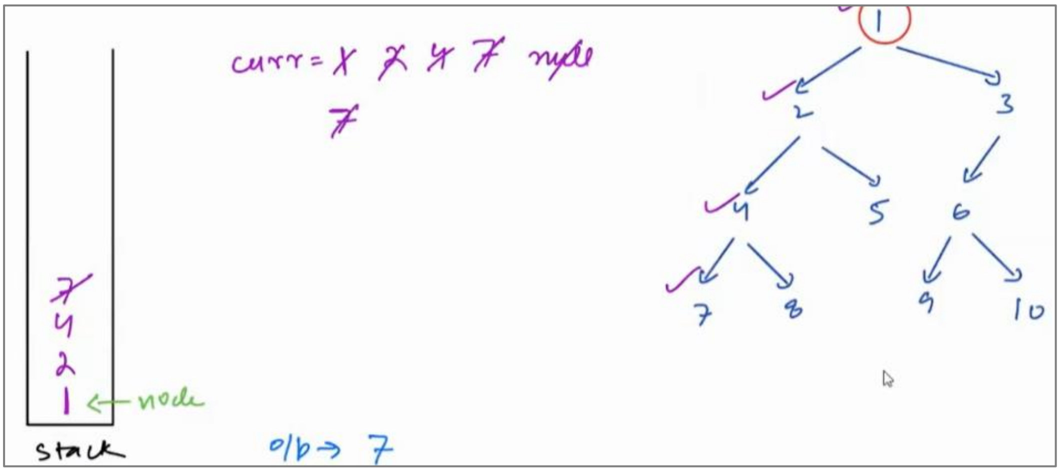
So I store 1 in the stack. When we store 1 in stack→ this is complete node and not only data



Now we are at 2, is there left of 2 → yes, go to 4 and store 2. Go to 7 → store 4. Is there left of 7 → No null. Store 7 and go to null

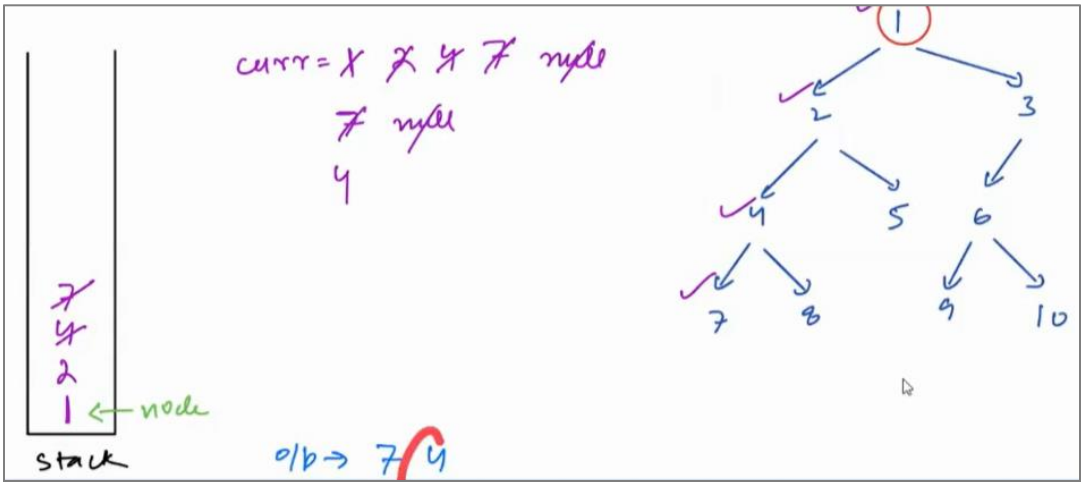


Since this is null → I will pop the last element in the stack which is 7.

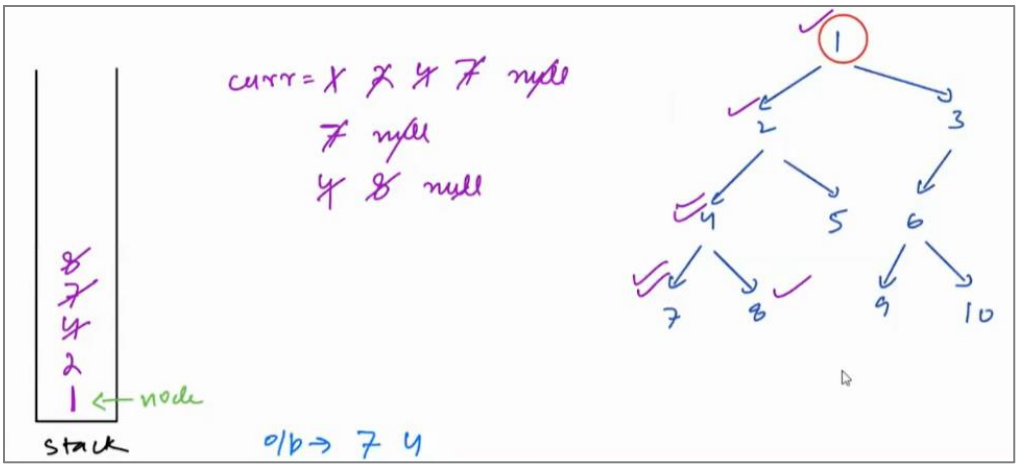


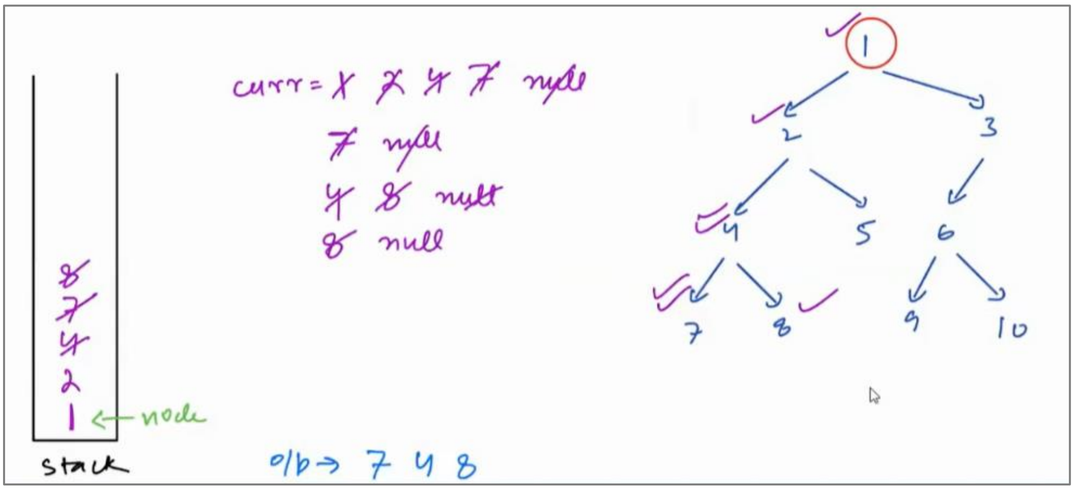
Now when I'll pop 7 means its left is completely done. So print 7.

Now go to right of 7 which is null, but we will not push 7 in the stack. Now we reach null, we will pop from the stack → we will pop 4 and print 4 coz its left is completely done

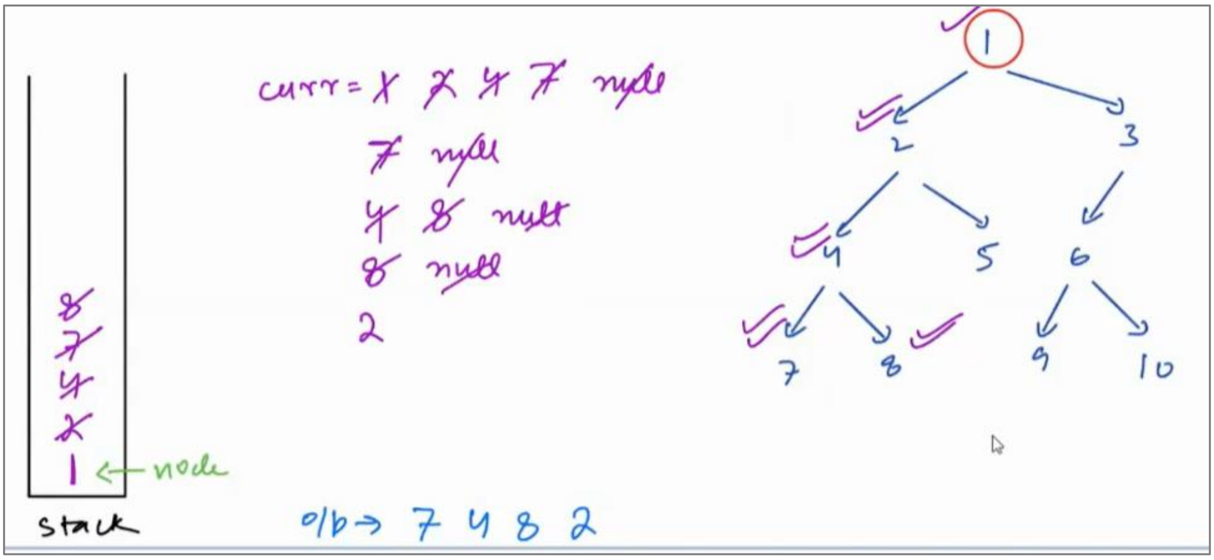


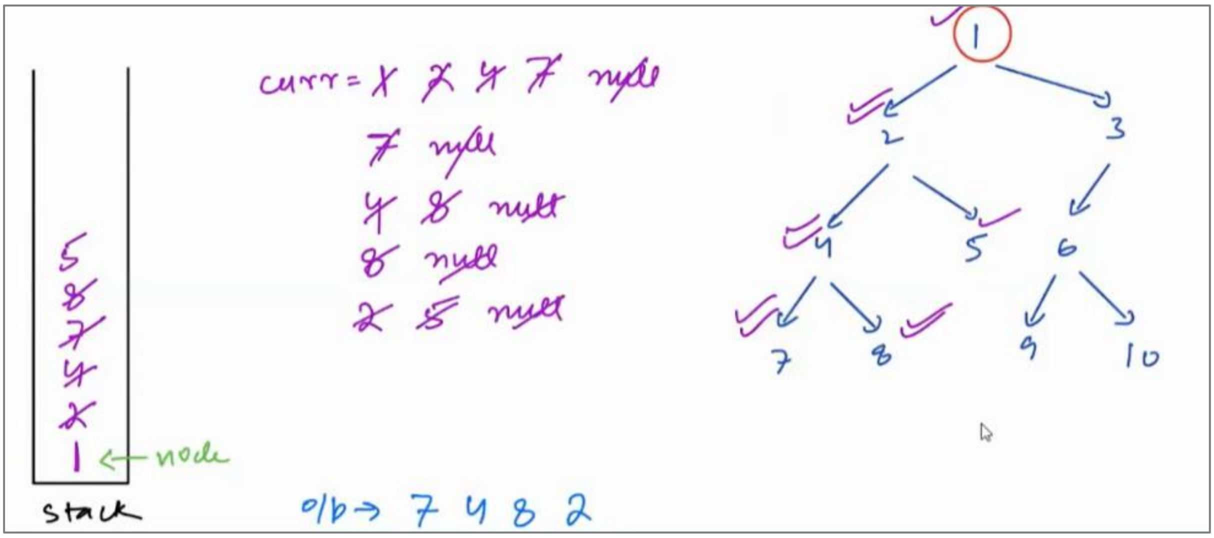
Now we will go to the right of 4 which is 8→ push in stack and go to the left of 8 which is null → pop and print 8

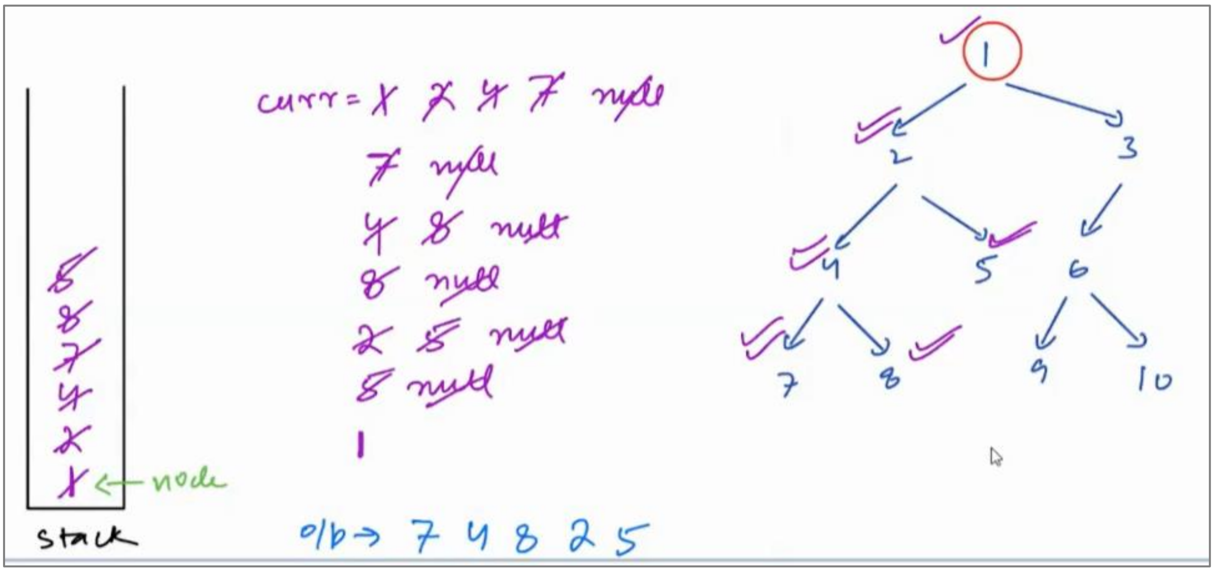


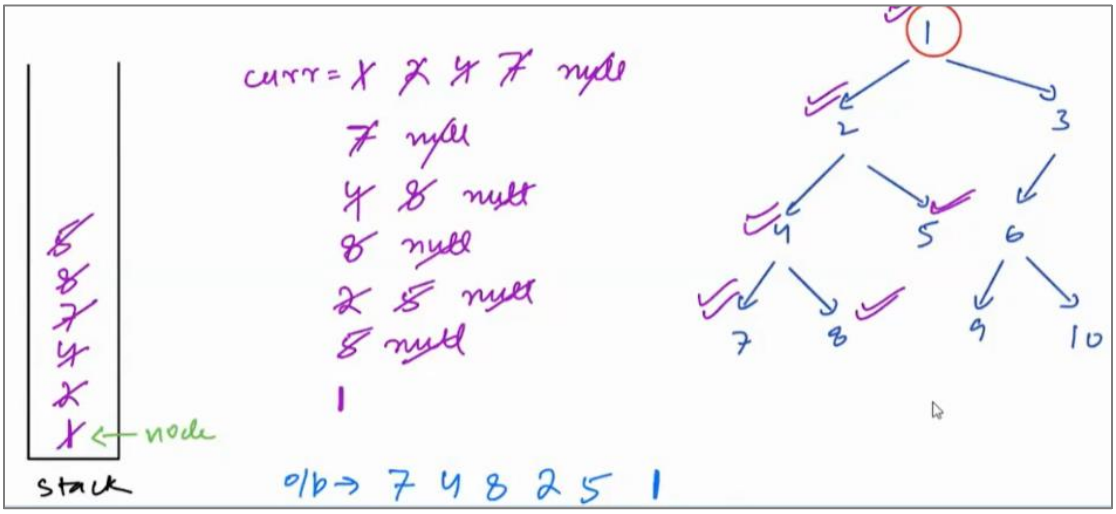


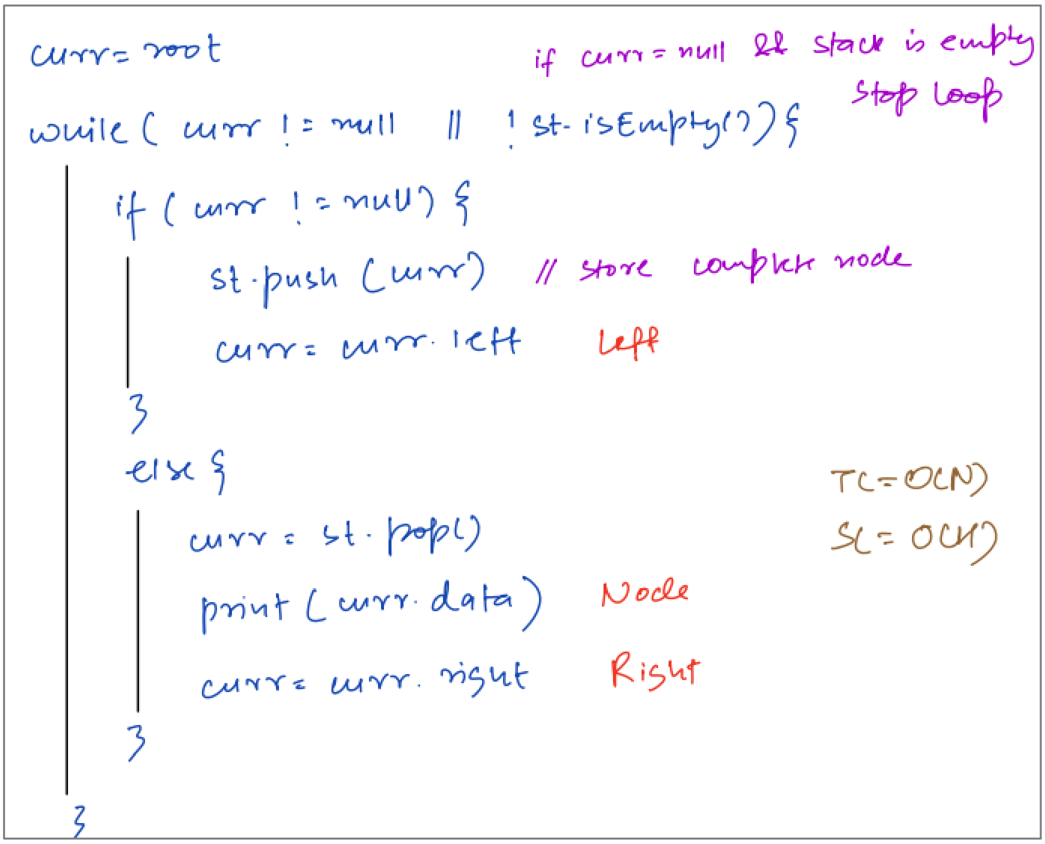
Now go to the right of 8 which is null → pop again from the stack → here last element is 2 → pop and print 2 → which means 2's left is done.



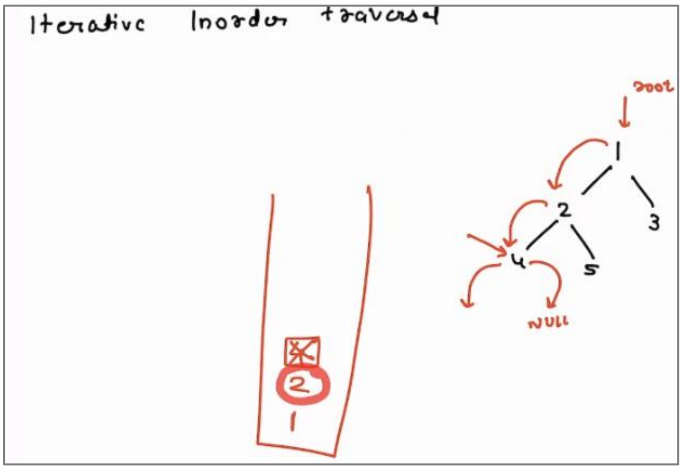

curr = ~~1~~ ~~2~~ ~~4~~ ~~7~~ ~~null~~
~~7~~ ~~null~~
~~4~~ ~~8~~ ~~null~~
~~8~~ ~~null~~
~~2~~ ~~5~~ null
5
~~8~~
~~7~~
~~4~~
~~2~~
1 ← node
stack
o/p → 7 4 8 2
1
2
3
4
5
6
7
8
9
10


curr = ~~1~~ ~~2~~ ~~4~~ ~~7~~ ~~null~~
~~7~~ ~~null~~
~~4~~ ~~8~~ ~~null~~
~~8~~ ~~null~~
~~2~~ ~~5~~ ~~null~~
~~5~~ ~~null~~
1
~~5~~
~~8~~
~~7~~
~~4~~
~~2~~
~~1~~ ← node
stack
o/p → 7 4 8 2 5
1
2
3
4
5
6
7
8
9
10


curr = ~~1~~ ~~2~~ ~~4~~ ~~7~~ ~~null~~
~~7~~ ~~null~~
~~4~~ ~~8~~ ~~null~~
~~8~~ ~~null~~
~~2~~ ~~5~~ ~~null~~
~~5~~ ~~null~~
1
~~5~~
~~8~~
~~7~~
~~4~~
~~2~~
~~1~~ ← node
stack
o/p → 7 4 8 2 5 1
1
2
3
4
5
6
7
8
9
10



```
curr = root                                    if curr = null && stack is empty
                                                 stop loop
while ( curr != null || ! st.isEmpty() ) {
    if ( curr != null) {
        st.push (curr)    // store complete node
        curr = curr.left       Left
    }
    else {                                            TC = O(N)
        curr = st.pop()                               SC = O(N)
        print (curr.data)      Node
        curr = curr.right      Right
    }
}
```



We are at root node. In Inorder traversal, we have to go to left, → so we go to left, at 2 but we have to save the data → so that once the left tree is done, you know where to go to →here we use stack to save the data

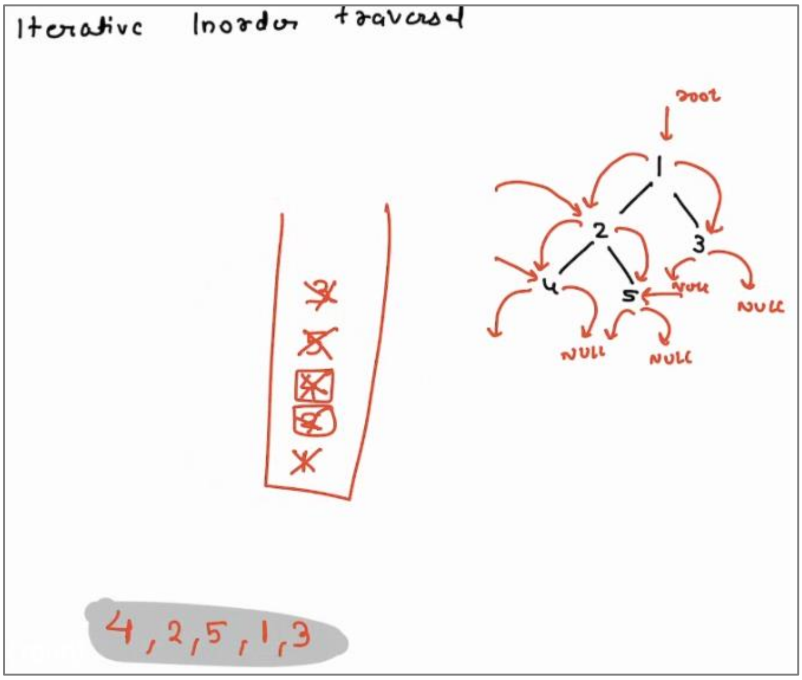
So saving 1 in stack, once I save and now go to the left subtree

Once the left of 2 is done, you want to come back to 2 or come back to 1 again? → 2 so you want to go to the recent element → Last element you want to go back to → Last in first out → kinda a stack

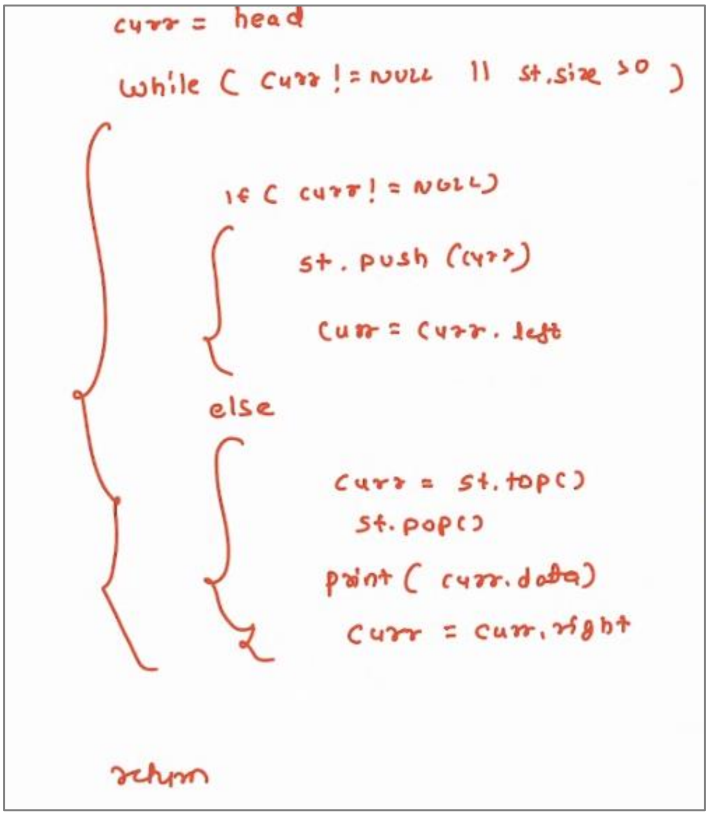
Go to 4 and then left of 4 but there is null → go back to 4 and pop

Go to the right of 2 which is 5. So i will insert 5 in the stack, and go to the right. Then I encounter null, pop it and print it.

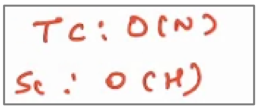
Whenever you go 1st time, insert



Implementation



```
curr = head
while ( curr != NULL || st.size >0 )
{
    if ( curr != NULL)
    {
        st.push (curr)
        curr = curr.left
    }
    else
    {
        curr = st.top()
        st.pop()
        print ( curr.data)
        curr = curr.right
    }
}
return
```

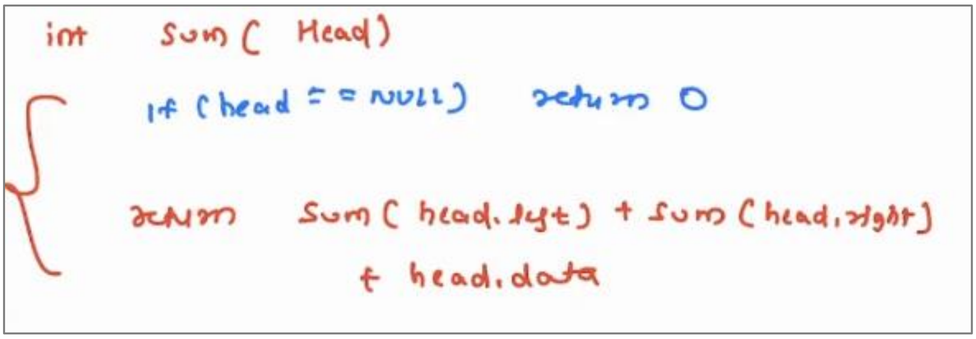


TC : O(N)
SC : O(H)

Problem Statement 1-

Given a tree, find the sum of the tree

ans = 1+2+3+6
= 12

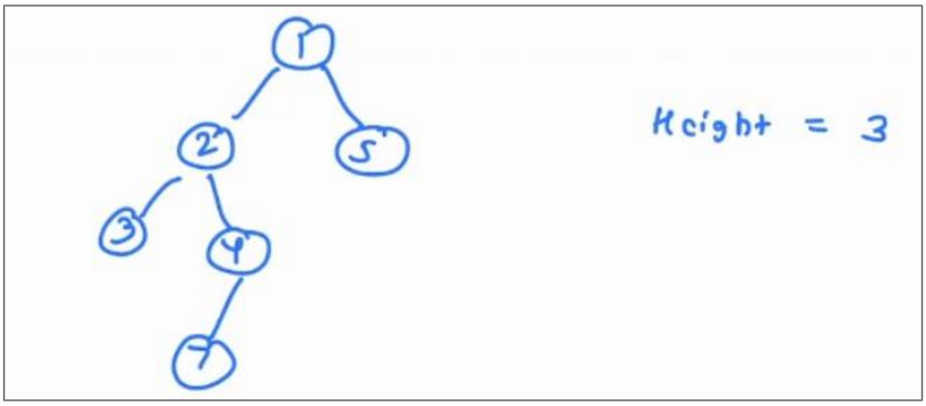


```
int    sum ( Head)
{
    if (head == NULL)   return 0

    return   sum( head.left) + sum(head.right)
                 + head.data
}
```

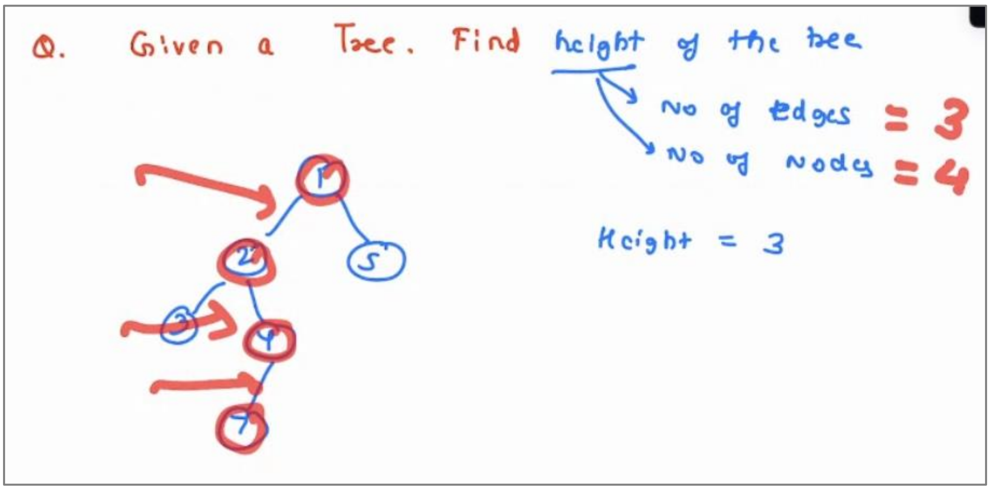
Make a recursive call for head.left, head.right and head.data

Problem Statement 2-

Q. Given a Tree. Find height of the tree



Height = 3



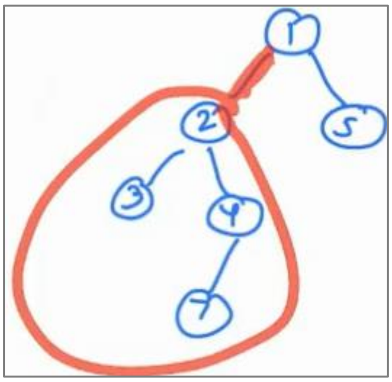
Q. Given a Tree. Find height of the tree

No of edges = 3

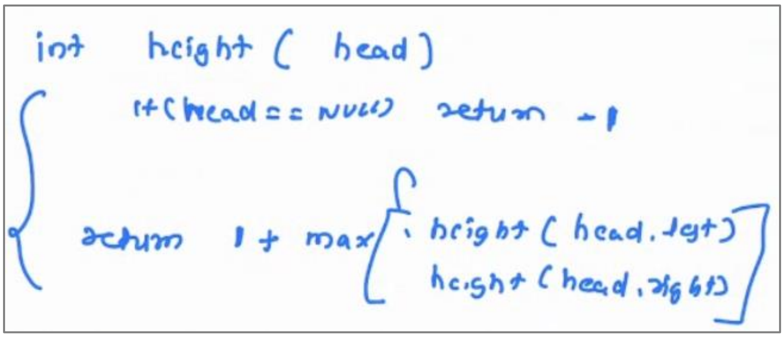
No of nodes = 4

Height = 3

Definition of height varies but mostly we are using this definition → No. of edges



Since we don't know whether height of left subtree is more or height of right subtree → so we will calculate height of both subtree and add 1 to it → which is the edge connecting to the root node

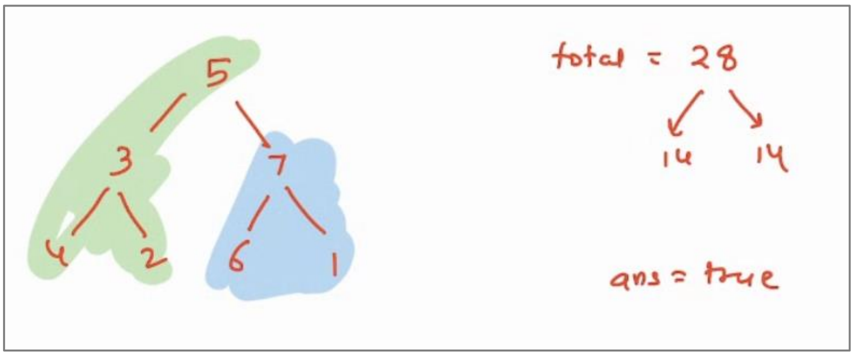


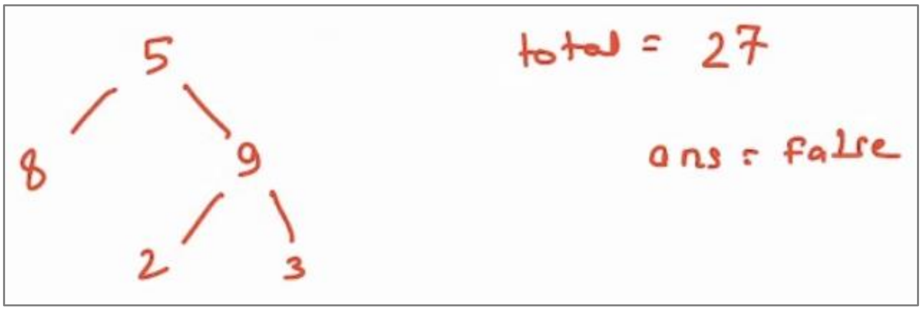
```
int height ( head)
{
    if(head == NULL) return -1

    return 1 + max{ height(head.left)
                    height(head.right) }
}
```

Equal Tree Partition

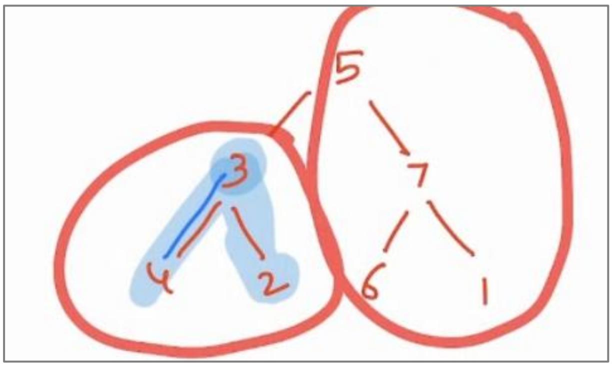
Problem Statement 3 –

Given root of a Binary Tree. Check if you can partition tree into 2 continuous portion of equal sum.

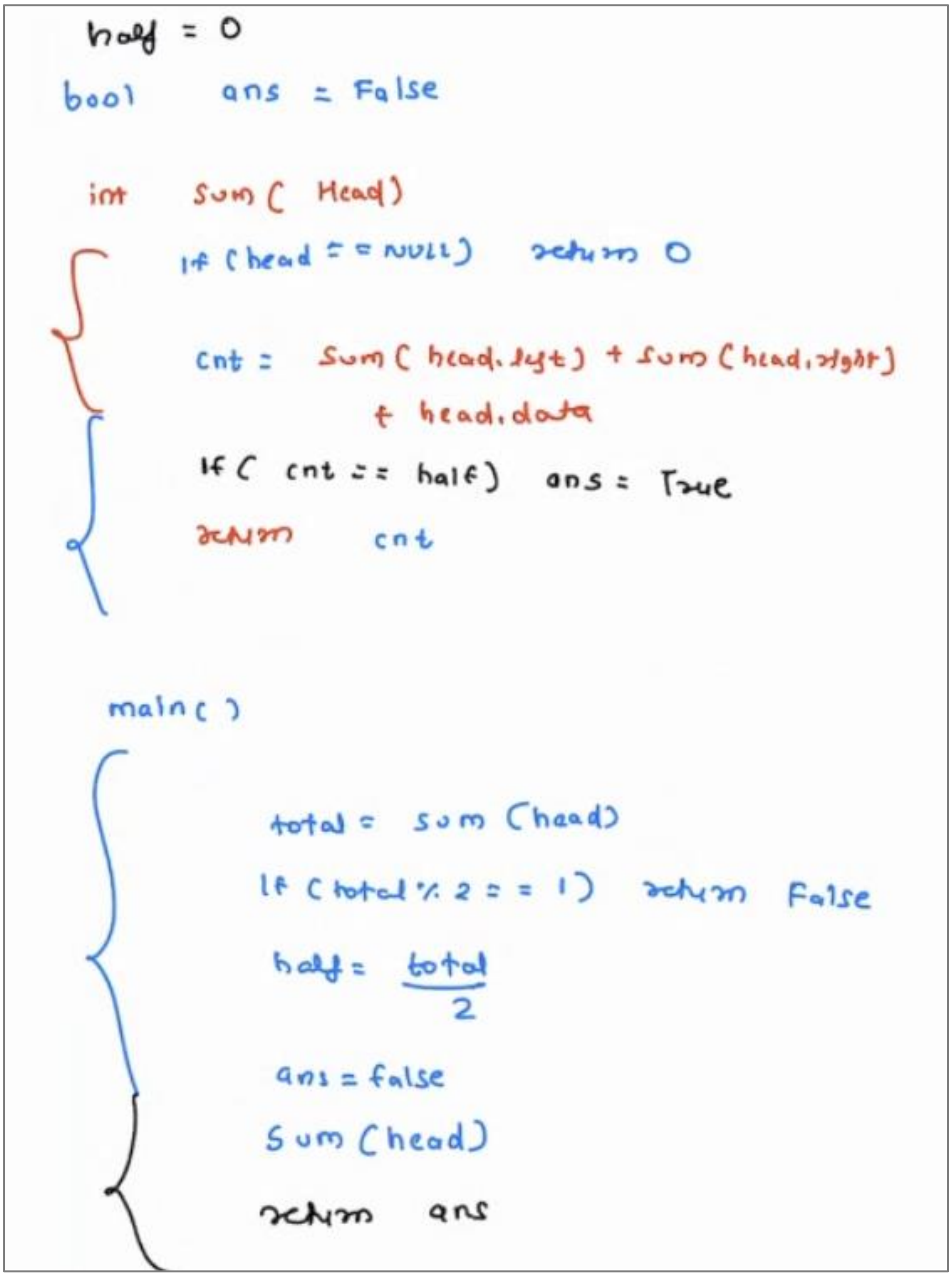




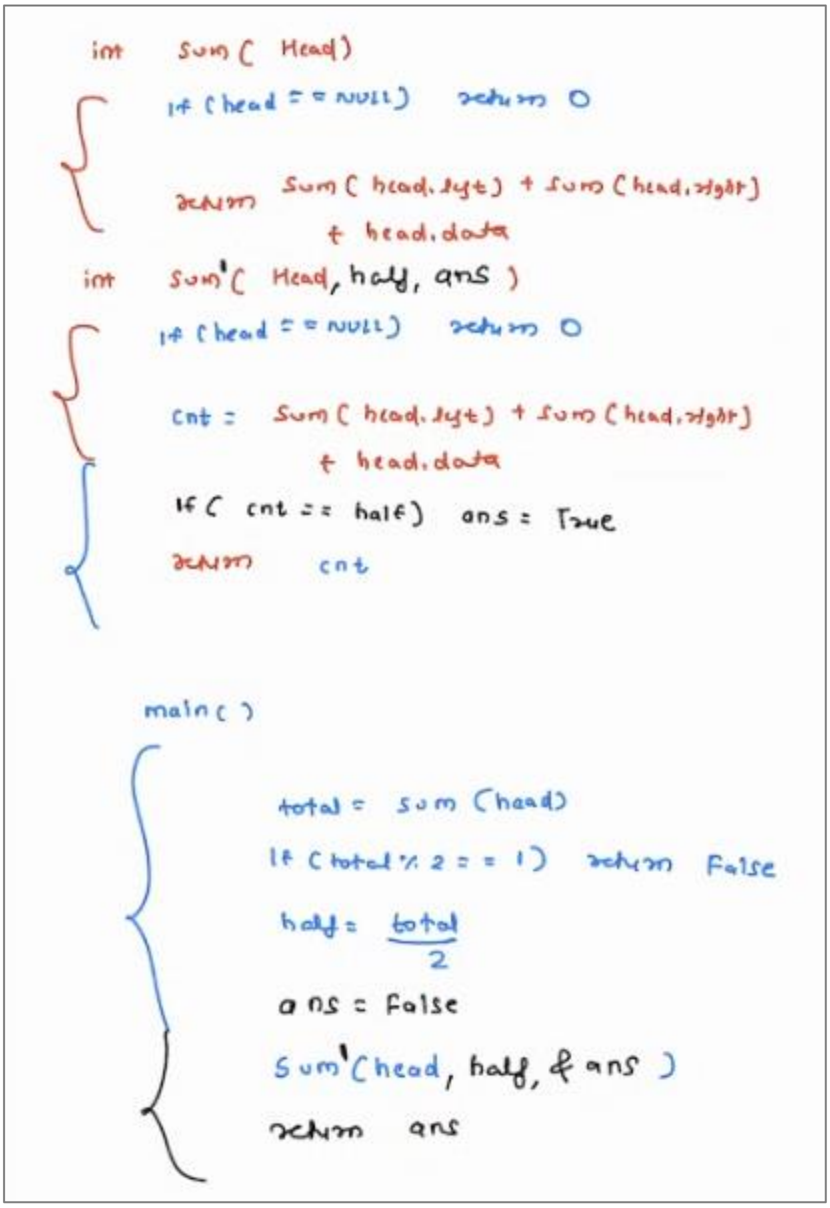
Since the sum is odd, we cannot divide this in 2 equal parts.



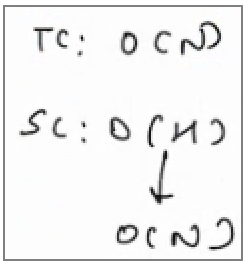
left one is a subtree and right one is not a subtree but a continuous subtree. Continuous means all are connected to each other



```
half = 0
bool   ans  = False

int   sum(  Head)
{
      if (head == NULL)   return 0

      cnt =  sum( head.left) + sum (head.right)
                + head.data
      if( cnt == half)  ans = True
      return   cnt
}

main( )
{
        total =  sum (head)
        if (total % 2 == 1)  return  False
        half = $\frac{total}{2}$
        ans = false
        sum (head)
        return  ans
}
```



```
int   sum( Head)
{
    if (head == NULL)   return 0

    return   sum( head.left) + sum(head.right)
             + head.data
}
int   sum'( Head, half, ans )
{
    if (head == NULL)   return 0

    cnt =   sum( head.left) + sum(head.right)
            + head.data
    if( cnt == half)   ans = True
    return    cnt
}

    main( )
    {
        total =  sum (head)
        if (total % 2 == 1)   return  False
        half =  total/2
        ans = False
        sum'(head, half, &ans )
        return   ans
    }
```



TC: O(N)

SC: O(H)

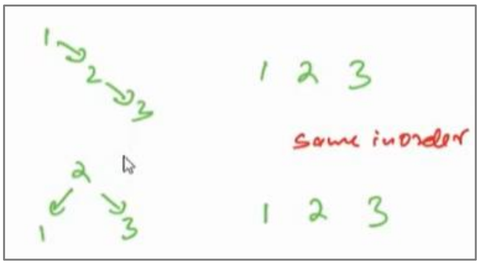
↓

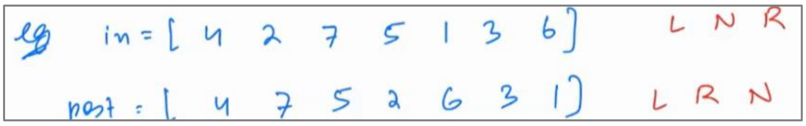
O(N)

Problem Statement 3- Construct a tree from Inorder and Postorder

Construct binary tree from the given inorder and post order traversal(distinct nodes)

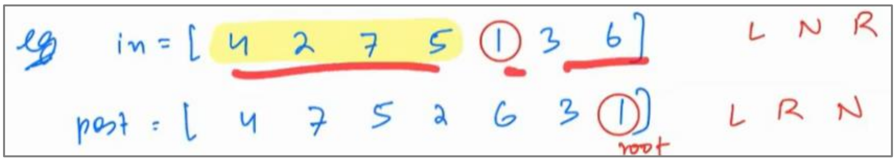
NOTE: Only from Inorder we cannot create a tree. We can create a tree using
In Order + Post Order ✓
In Order + Pre Order ✓
but not from
Pre Order + Post Order ✗




eg in = [4 2 7 5 1 3 6] L N R

post : [4 7 5 2 6 3 1] L R N

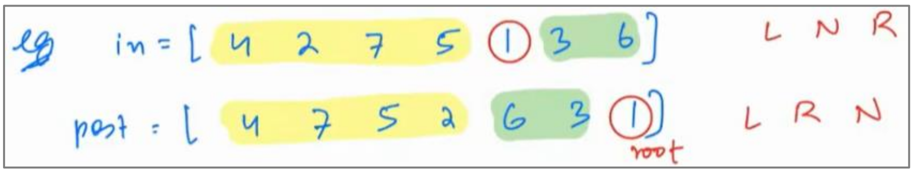
Now from here we can understand that 1 is the root node as LRN in post order sequence.


eg in = [4 2 7 5 ① 3 6] L N R

post : [4 7 5 2 6 3 ①] L R N

root

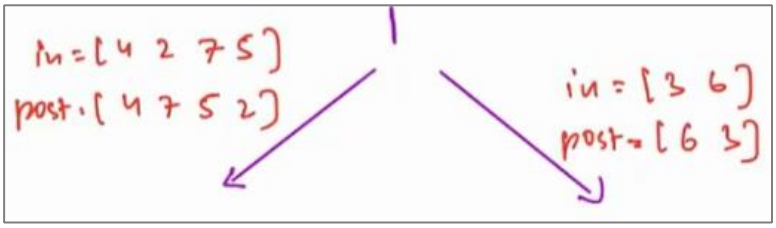
From preorder sequence, the left of 1 would be left subtree and right of 1 would be right subtree.


eg in = [4 2 7 5 ① 3 6] L N R

post : [4 7 5 2 6 3 ①] L R N

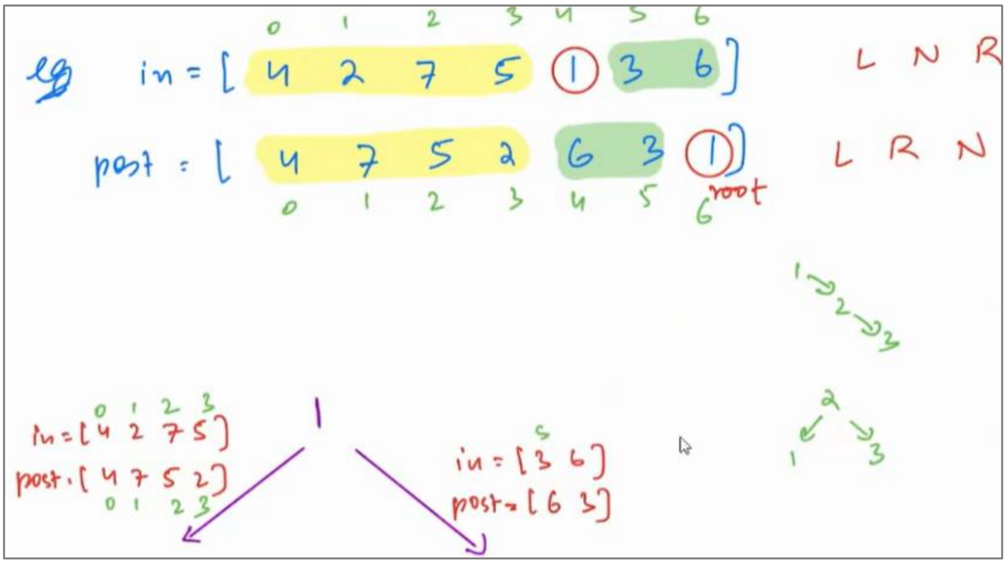
root

So we have separated left right and node for both in & post order sequences

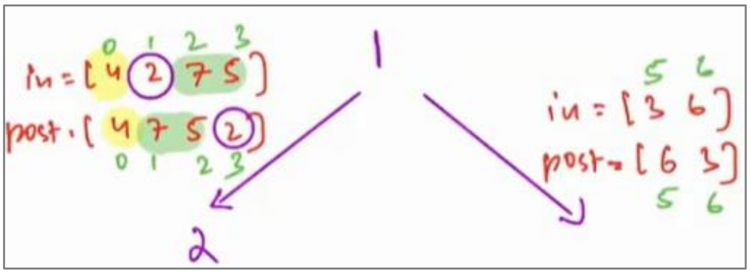


This is the alignment

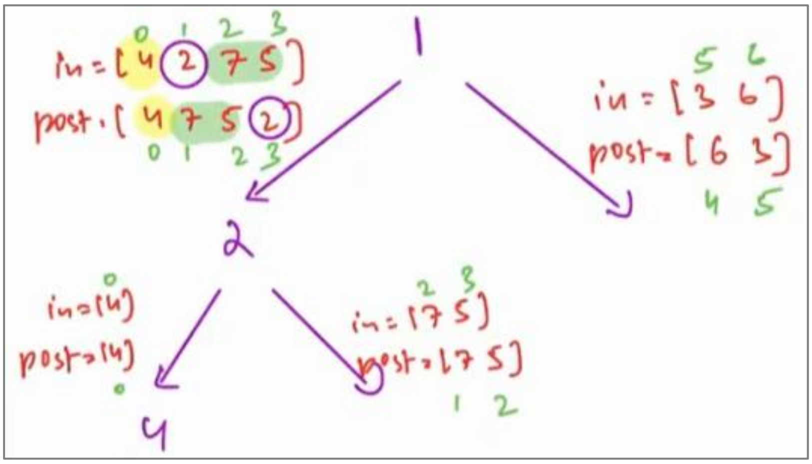
Mark indexes also



Root node in the subtree is 2. and we have left right distribution. yellow highlighted will be the left and green highlighted will be the right

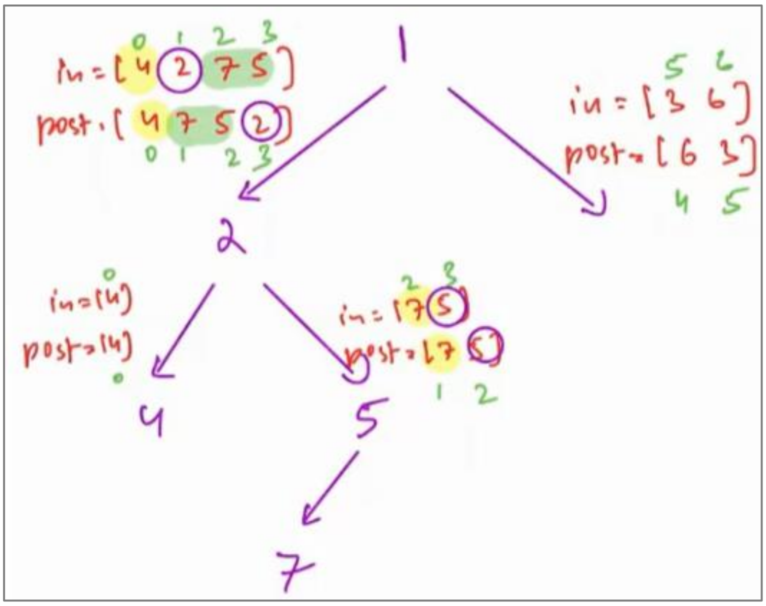


We will do inorder and preorder recursively(again)

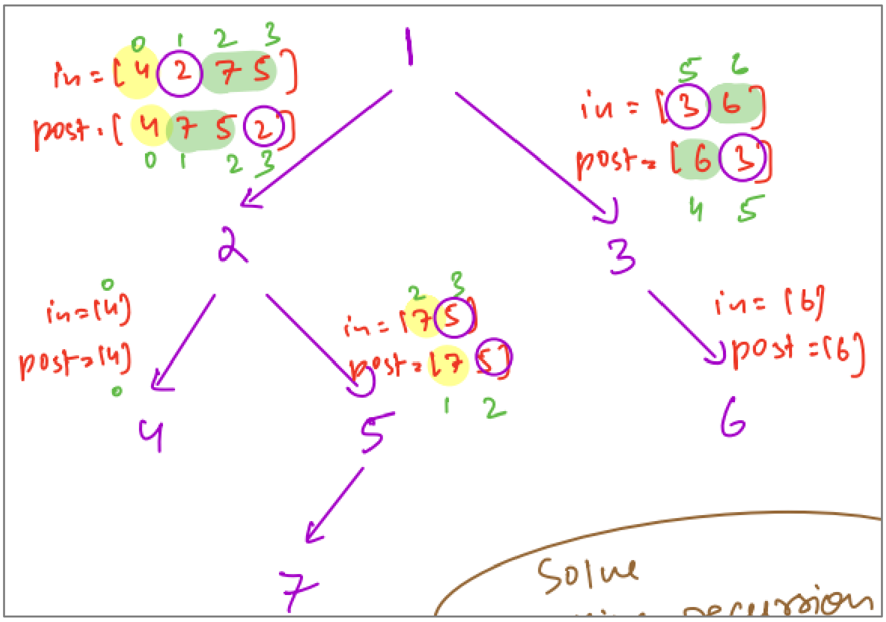


In the left only one node is there → so right just 4 here

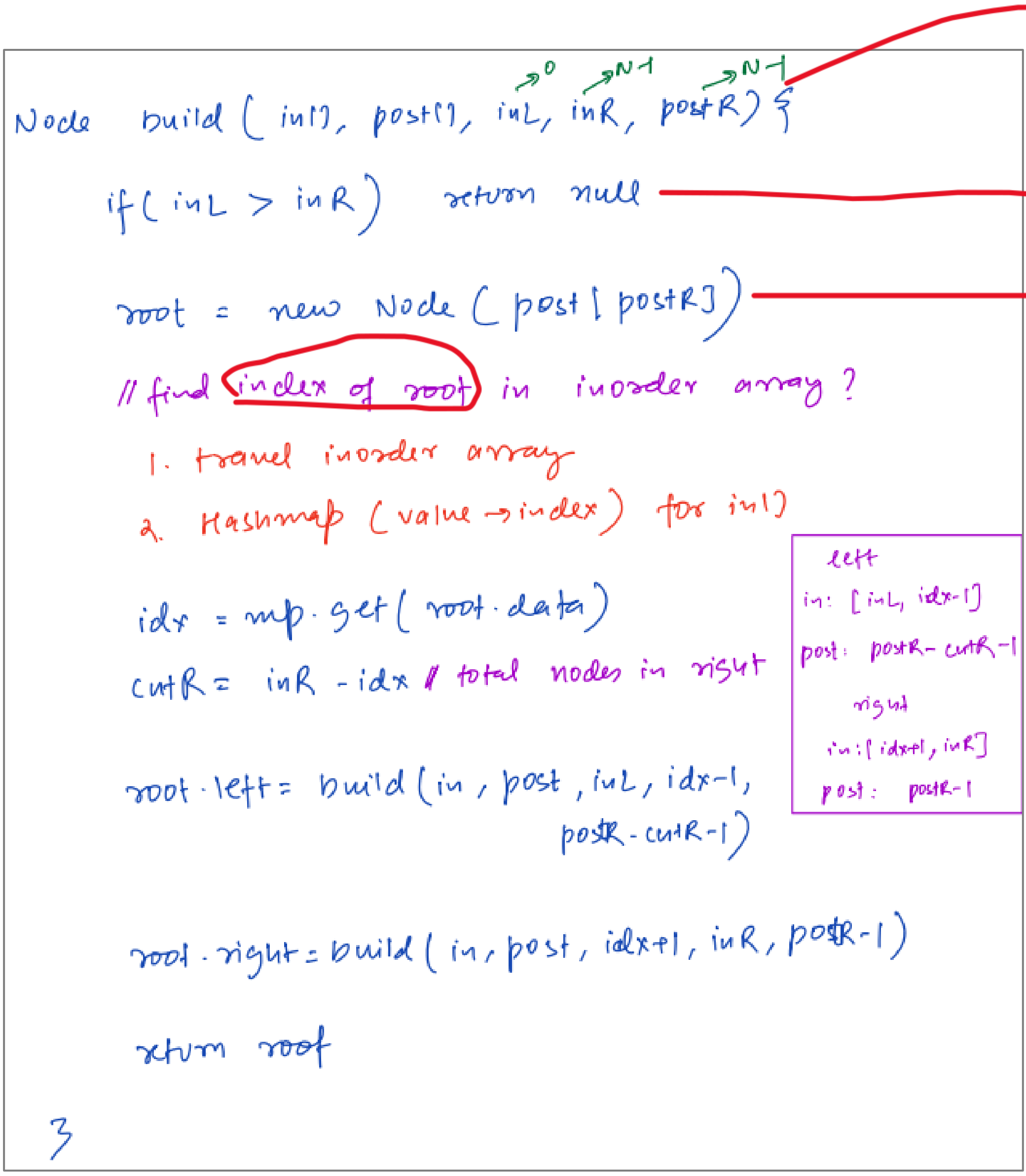
On the right hand side, 5 is the root node as in post order last element is the root node (LRN)



Left is 7, and there is no right



Here 3 is the root node. As 6 is on the right side of 3 which in inorder means LNR → so 6 is on the right side



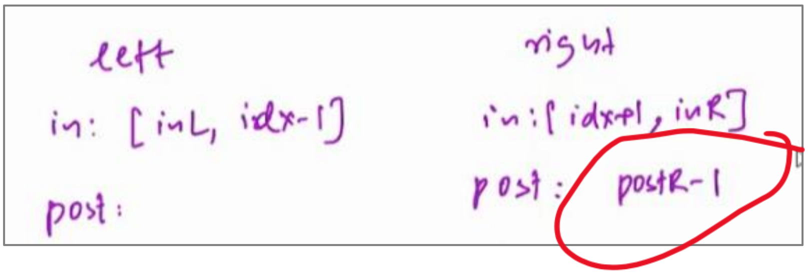
```
                               →0    →N-1    →N-1
Node  build ( in[], post[], inL, inR, postR) {
    if( inL > inR)  return null
    root = new Node ( post [ postR])
    // find index of root in inorder array ?
      1. travel inorder array
      2. Hashmap ( value → index) for in[]
    idx = mp.get( root.data)
    cutR = inR - idx // total nodes in right
    root.left = build (in, post, inL, idx-1,
                                 postR - cutR-1)
    root.right = build ( in, post, idx+1, inR, postR-1)
    return root
}
```

left
in: [inL, idx-1]
post: postR- cutR-1
right
in: [idx+1, inR]
post: postR-1

We require in order left & right index and post order right index

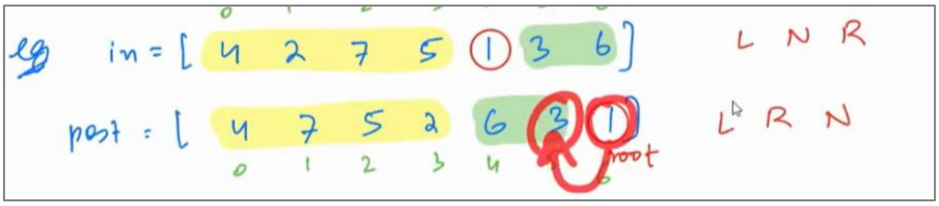
If inL > inR → this will be out of bound so there would null node

Root node would be at the last in preorder but we cannot say at n-1 index since in a subtree the last node would have different index but would be a parent node



left
in: [inL, idx-1]
post:

right
in: [idx+1, inR]
post: postR-1



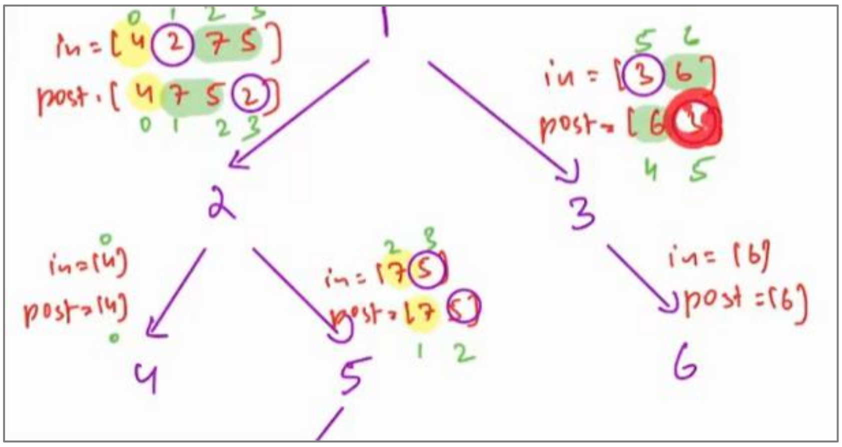
eg in = [4 2 7 5 ① 3 6] L N R

post : [4 7 5 2 6 3 ①] L R N

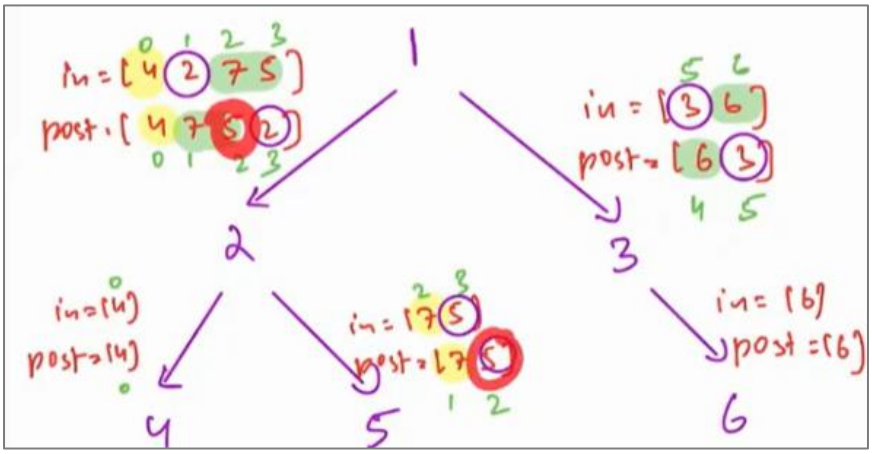
0 1 2 3 4

root

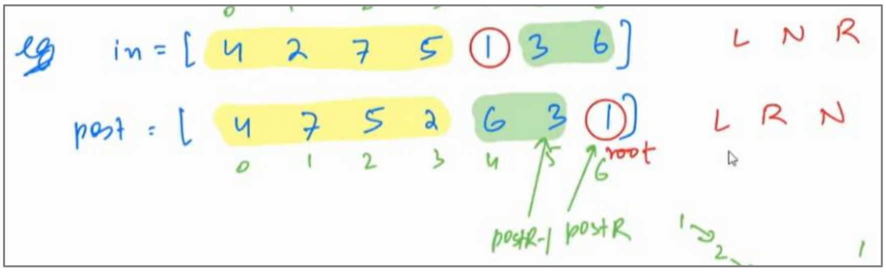
postR is the index of root node(1) and 3 can be said as postR -1 is the parent node index of right subarray as we can see below:

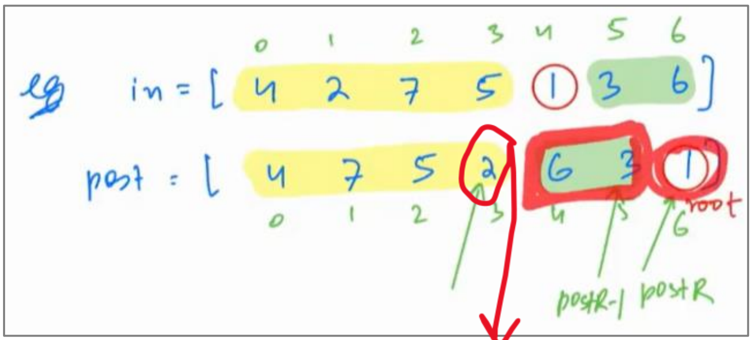


For the right subarray 3 was parent node which was placed at postR-1



Always for the right siubarray the new postRight index will be PostR-1 (one before the root node)



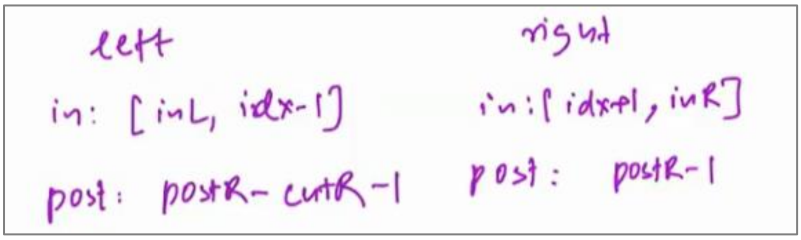


postLeft = postR-cntR-1
= 6-2 -1 =3

And for getting the index of parent node of left subarray → index of root node minus the number of elements of right subarray

cnt = inR – idx = 6 -4 =2 two elements in the right subarray

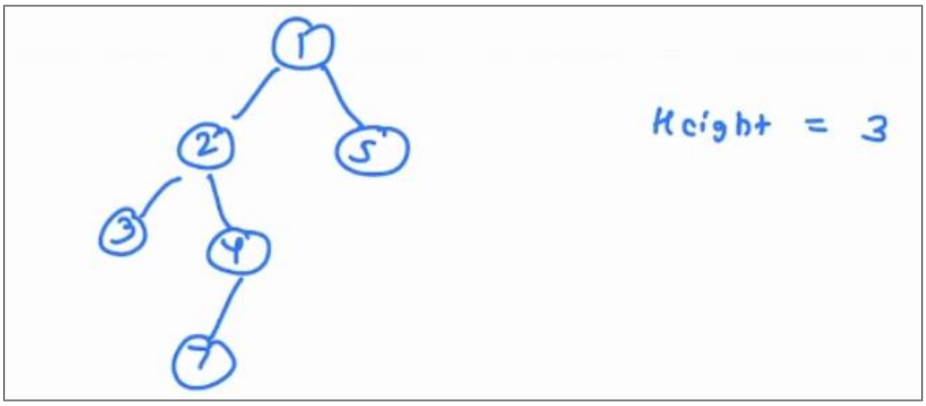
For post it will be postLeft = postR-cntR-1 = 6-2 -1 =3

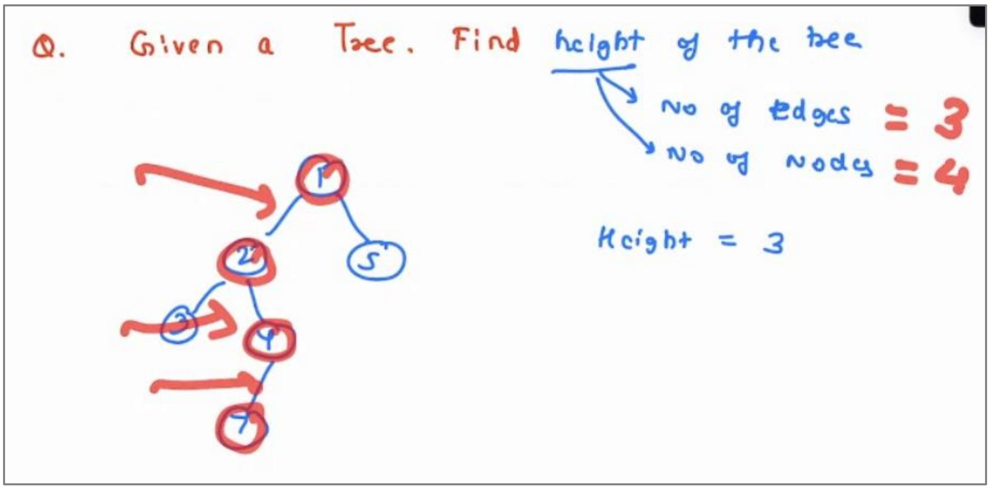


left	right
in: [inL, idx-1]	in:[idx+1, inR]
post: postR- cntR-1	post: postR-1

Problem Statement 2-

Q. Given a Tree. Find height of the tree





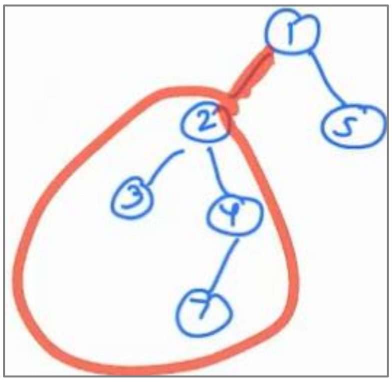
Q. Given a Tree. Find height of the tree

No of edges = 3

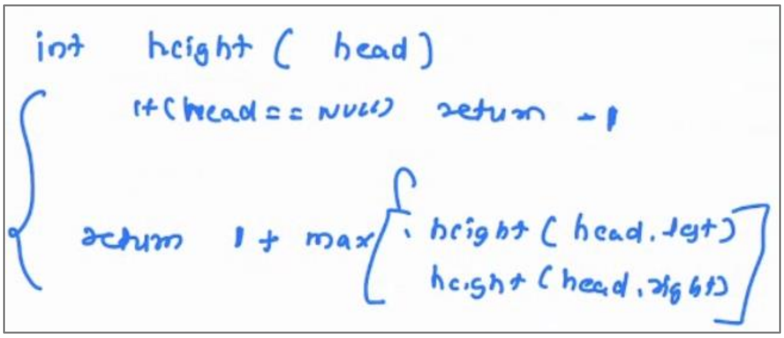
No of nodes = 4

Height = 3

Definition of height varies but mostly we are using this definition → No. of edges



Since we don’t know whether height of left subtree is more or height of right subtree → so we will calculate height of both subtree and add 1 to it → which is the edge connecting to the root node



```
int  height ( head)
{
      if(head == NULL)  return -1

  return  1 + max[ height(head.left)
                   height(head.right) ]
}
```

Problem Statement 1 - Inorder Traversal

Problem Description

Given a binary tree, return the inorder traversal of its nodes' values.

Problem Constraints

1 <= number of nodes <= 10^5

Input Format

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

Output Format

Return an integer array denoting the inorder traversal of the given binary tree.

Example Input

Input 1:

```
  1
   \
    2
   /
  3
```

Input 2:

```
  1
 / \
6   2
```

  /
 3

Example Output

Output 1:

[1, 3, 2]

Output 2:

[6, 1, 3, 2]

Example Explanation

Explanation 1:

The Inorder Traversal of the given tree is [1, 3, 2].

Explanation 2:

The Inorder Traversal of the given tree is [6, 1, 3, 2].

```
public class Solution {
   public ArrayList<Integer> inorderTraversal(TreeNode A) {
      ArrayList<Integer> res = new ArrayList<>();
      Stack<TreeNode>st = new Stack<>();
      TreeNode curr = A;
      while(curr != null || !st.isEmpty()){
         if(curr != null){
            st.push(curr);
            curr = curr.left;
         }else{
            curr = st.pop();
            res.add(curr.val);
            curr = curr.right;
         }

      }
      return res;

   }
}
```

Problem Statement 2 - Preorder Traversal

Problem Description

Given a binary tree, return the preorder traversal of its nodes values.

Problem Constraints

1 <= number of nodes <= 10^5

Input Format

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

Output Format

Return an integer array denoting the preorder traversal of the given binary tree.

Example Input

Input 1:

 1
  \

```
   2
  /
 3
```

Input 2:

```
  1
 / \
6   2
   /
  3
```

Example Output

Output 1:

[1, 2, 3]

Output 2:

[1, 6, 2, 3]

Example Explanation

Explanation 1:

The Preoder Traversal of the given tree is [1, 2, 3].

Explanation 2:

The Preoder Traversal of the given tree is [1, 6, 2, 3].

```
public class Solution {
    public ArrayList<Integer> preorderTraversal(TreeNode A) {
        ArrayList<Integer> res = new ArrayList<>();
        Stack<TreeNode> st = new Stack<>();
        TreeNode curr = A;
        if(A ==null) return res;

        st.push(A);
        while(!st.isEmpty()){
            curr = st.pop();
            res.add(curr.val);

            if(curr.right !=null){
                st.push(curr.right);
            }
            if(curr.left !=null){
                st.push(curr.left);
            }
        }
        return res;

    }
}
```

//Using Recursion

```
import java.util.ArrayList;

public class Solution {
    public ArrayList<Integer> preorderTraversal(TreeNode A) {
```

```
    ArrayList<Integer> result = new ArrayList<>();
    preorderHelper(A, result);
    return result;
  }

  private void preorderHelper(TreeNode node, ArrayList<Integer> result) {
    if (node == null) {
      return;
    }
    result.add(node.val); // Add root value
    preorderHelper(node.left, result); // Traverse left subtree
    preorderHelper(node.right, result); // Traverse right subtree
  }
}
```

Problem Statement 3- Path Sum

Problem Description

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.

Problem Constraints

1 <= number of nodes <= 10^5

-100000 <= B, value of nodes <= 100000

Input Format

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

Second argument is an integer B denoting the sum.

Output Format

Return 1, if there exist root-to-leaf path such that adding up all the values along the path equals the given sum. Else, return 0.

Example Input

Input 1:

```
 Tree:
      5
     / \
    4   8
   /   / \
  11  13  4
 / \       \
7   2       1
```

B = 22

Input 2:

```
 Tree:
      5
     /  \
    4    8
   /     / \
```

```
  -11   -13  4
```

B = -1

Example Output

Output 1:

1

Output 2:

0

Example Explanation

Explanation 1:

There exist a root-to-leaf path 5 -> 4 -> 11 -> 2 which has sum 22. So, return 1.

Explanation 2:

There is no path which has sum -1.

```
public class Solution {
   public int hasPathSum(TreeNode A, int B) {
      if(A == null) return 0;

      //int sum -= A.val;
      if(A.left == null && A.right == null && B == A.val){
         return 1;
      }
      int leftCheck = hasPathSum(A.left, B - A.val);
      int rightCheck = hasPathSum(A.right, B - A.val);

      return leftCheck ==1 || rightCheck ==1 ?1:0;

   }
}
```

Problem Statement 4 - Equal Tree Partition

Given a binary tree **A**. Check whether it is possible to partition the tree to two trees which have equal sum of values after removing exactly one edge on the original tree.

Problem Constraints

1 <= size of tree <= 100000

0 <= value of node <= 10^9

Input Format

First and only argument is head of tree A.

Output Format

Return 1 if the tree can be partitioned into two trees of equal sum else return 0.

Example Input

Input 1:

```
            5
           /  \
```

```
   3   7
  / \  / \
 4  6  5  6
```

Input 2:

```
    1
   / \
  2   10
      / \
     20  2
```

Example Output

Output 1: 1

Output 2: 0

Example Explanation

Explanation 1:

Remove edge between 5(root node) and 7:

```
Tree 1 =                           Tree 2 =
        5                               7
       /                               / \
      3                                5   6
     / \
    4   6
```

Sum of Tree 1 = Sum of Tree 2 = 18

Explanation 2:

The given Tree cannot be partitioned.

Actual Code

```
public class Solution {
  public int solve(TreeNode a) {
    HashMap<Long, Integer> map = new HashMap<>();
    long tot = populate(a, map);

    //handling edge case where total sum of the tree =0; When the total sum of the tree is 0,
    if (tot == 0){           // the tree can only be split into two subtrees with equal sums if there is more
      return map.getOrDefault(tot, 0) >1?1:0;          // than one subtree whose sum is 0.
    }
    return tot %2 == 0 && map.containsKey(tot/2)?1:0;
  }
  public long populate(TreeNode a, HashMap<Long, Integer> map){
    if(a ==null) return 0;

    long sum = a.val + populate(a.left, map) + populate(a.right, map);
    map.put(sum, map.getOrDefault(sum, 0) +1);
    return sum;
```

```
  }
}
```

//map.getOrDefault(tot, 0): It attempts to get the value associated with the key tot in the HashMap. If the key tot does not exist in the map, it returns the default value specified, which is 0 in this case.

// Handle the special case where tot == 0:
If the total sum is zero, we need to verify if there is more than one subtree with a sum of zero (to ensure a valid split).

- If there is more than one subtree with a sum of 0, return 1 (indicating a valid split).
- Otherwise, return 0 (no valid split).

// return tot %2 == 0 && map.containsKey(tot/2)?1:0;
For other cases, if tot is even, check if there exists a subtree with a sum equal to tot / 2. If such a subtree exists, return 1; otherwise, return 0.

Problem Statement 5 - Postorder Traversal

Given a binary tree, return the Postorder traversal of its nodes values.

Problem Constraints

1 <= number of nodes <= 10^5

Input Format

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

Output Format

Return an integer array denoting the Postorder traversal of the given binary tree.

Example Input

Input 1:

```
  1
   \
    2
   /
  3
```

Input 2:

```
  1
 / \
6   2
   /
  3
```

Example Output

Output 1:

[3, 2, 1]

Output 2:

[6, 3, 2, 1]

Example Explanation

Explanation 1:

The Preoder Traversal of the given tree is [3, 2, 1].

Explanation 2:

The Preoder Traversal of the given tree is [6, 3, 2, 1].

Actual Code

```
public class Solution {
  public ArrayList<Integer> postorderTraversal(TreeNode A) {
    ArrayList<Integer> res = new ArrayList<>();
    if(A==null) return res;
    traversal(A, res);
    return res;
  }
  private void traversal(TreeNode A, ArrayList<Integer> res){
    if(A == null) return;
    traversal(A.left, res);
    traversal(A.right, res);
    res.add(A.val);
  }


}
```

Problem Statement 6 - Sum binary tree or not

Problem Description

Given a binary tree. Check whether the given tree is a **Sum-binary Tree** or not.

Sum-binary Tree is a Binary Tree where the value of a every node is equal to sum of the nodes present in its left subtree and right subtree.

An empty tree is Sum-binary Tree and sum of an empty tree can be considered as 0. A leaf node is also considered as SumTree.

Return 1 if it sum-binary tree else return 0.

Problem Constraints

1 <= length of the array <= 100000

0 <= node values <= 50

Input Format

The only argument given is the root node of tree A.

Output Format

Return 1 if it is sum-binary tree else return 0.

Example Input

Input 1:

```
    26
   /  \
  10   3
```

/ \   \
4 6   3
Input 2:
    26
   /   \
  10   3
 / \   \
4 6   4

Example Output

Output 1: 1

Output 2: 0

Example Explanation

Explanation 1:

All leaf nodes are considered as SumTree.
Value of Node 10 = 4 + 6.
Value of Node 3 = 0 + 3
Value of Node 26 = (10 + 4 + 6) + 6

Explanation 2:

Sum of left subtree and right subtree is 27 which is not equal to the value of root node which is 26.

Actual Code

```
public class Solution {
  public int solve(TreeNode A) {
    return checksum(A)? 1:0;
  }
  private boolean checksum(TreeNode A){
    if(A == null) return true;
    if(A.left == null && A.right == null) return true;

    int l = sumtree(A.left);
    int r = sumtree(A.right);

    if(A.val == l + r && checksum(A.left) && checksum(A.right)){
      return true;
    }
    return false;
  }
  private int sumtree(TreeNode A){
    if(A == null) return 0;
    return A.val + sumtree(A.left) + sumtree(A.right);
  }
```

```
}
```