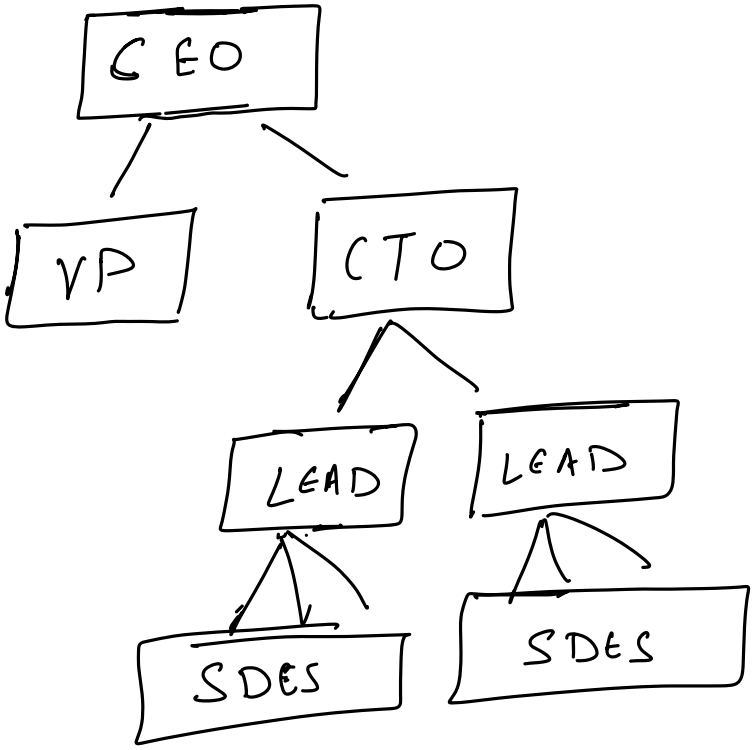
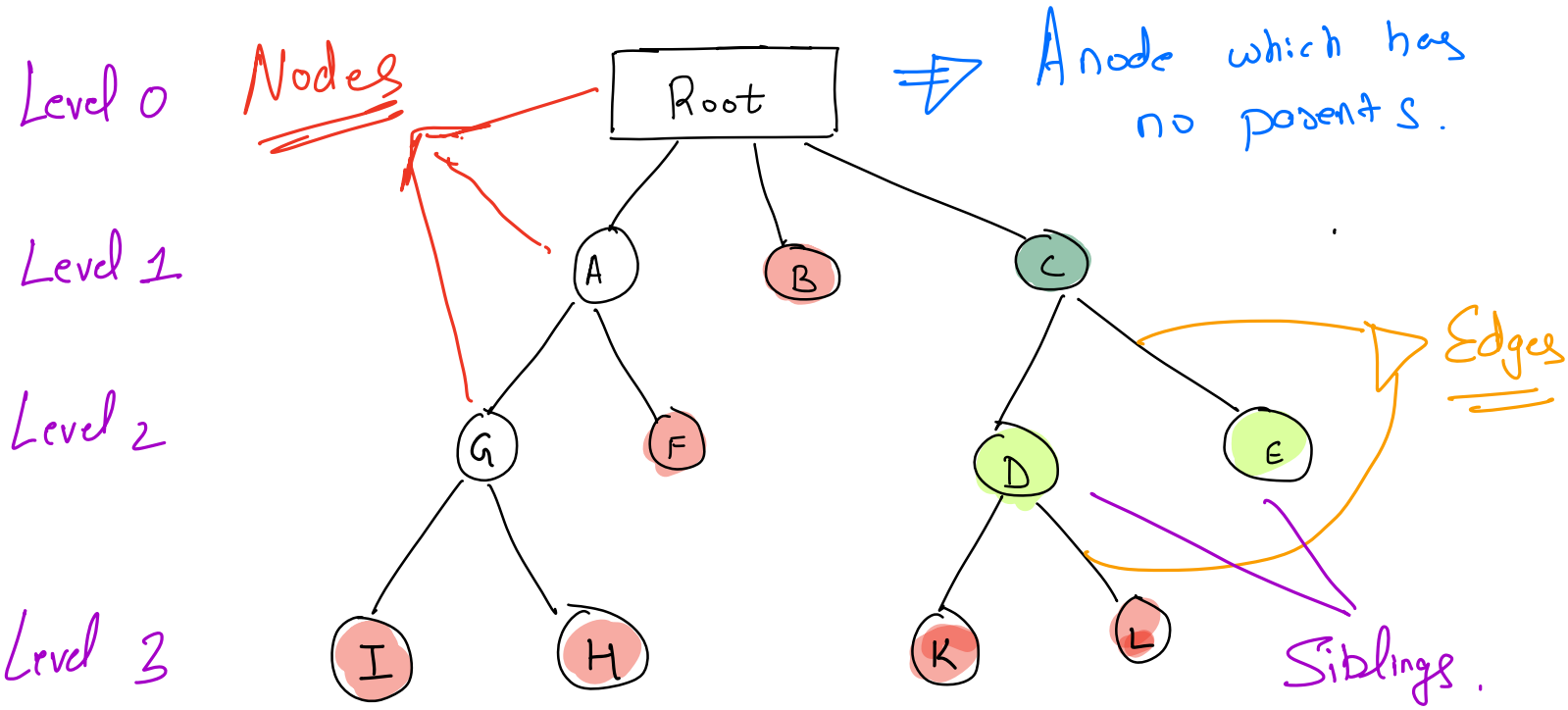
Hierarchical Data



TREES



1) A, B, C are children of Root.

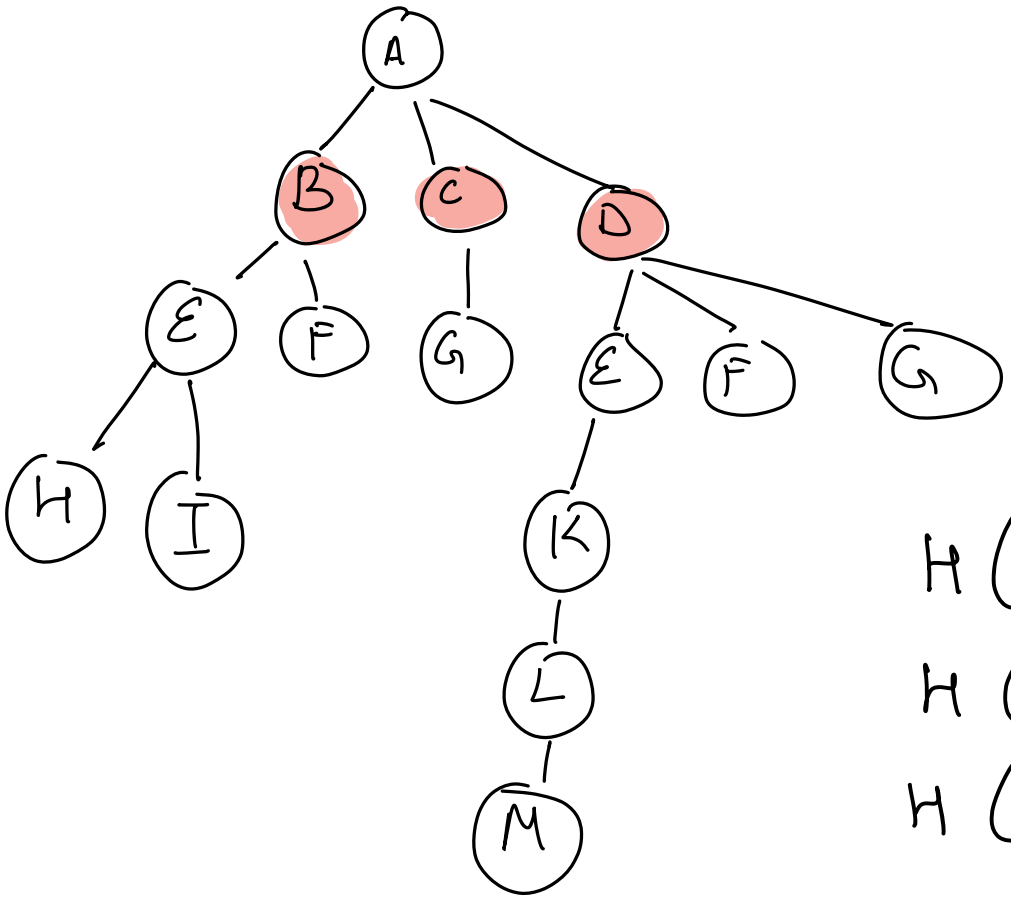
2) C is the parent of D & E.

3) D & E are siblings as they children of the same parent.

4) G, F, D, E are at same level.

5) Nodes having no child are leaf nodes.

Height of a node : length of the longest path starting from that node to any leaf node. We will go only to higher levels. Length is calculated using edges traversed.



H (D) = 4

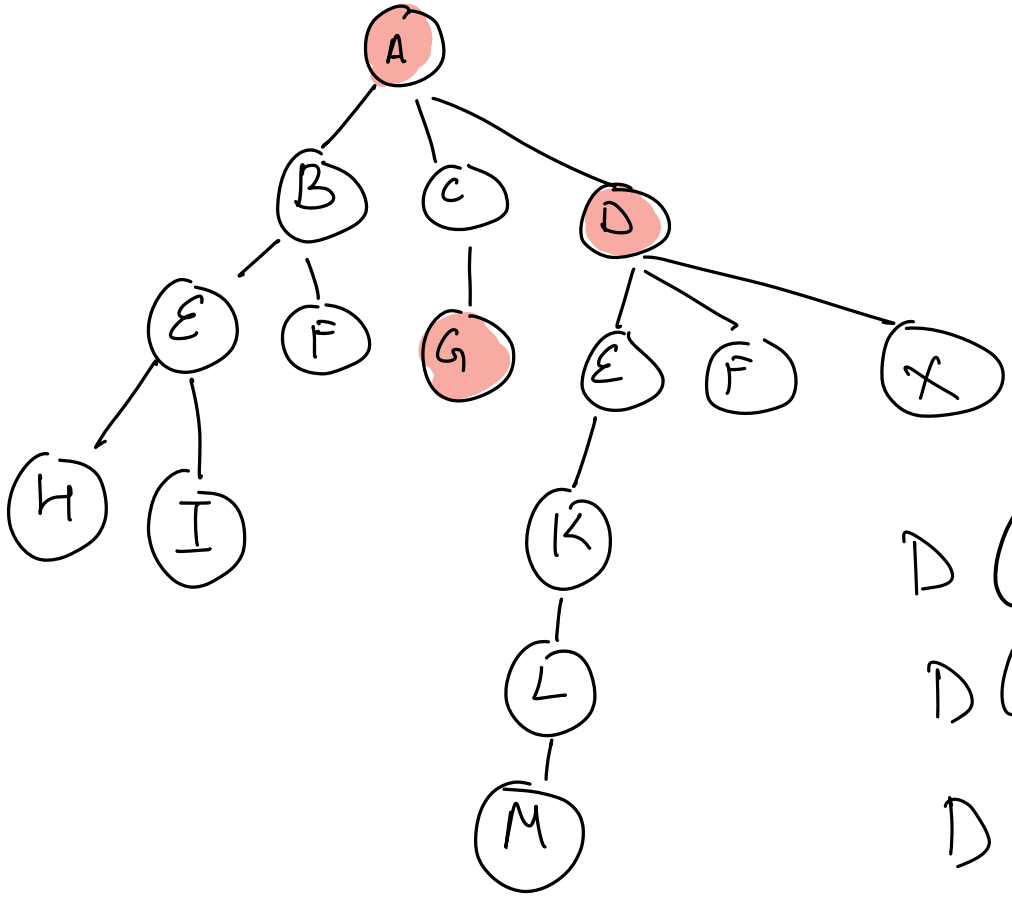
H (C) = 1

H (B) = 2

h(node) ⇒ max. height (children) + 1

Depth of node : The length of the path from that node to the root.

Level = Depth.



D(D) = 1

D(G) = 2

D(A) = 0

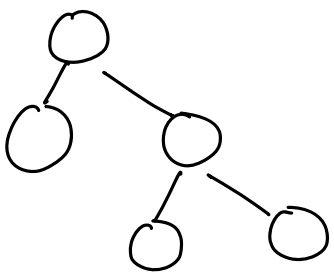
Types of Trees:

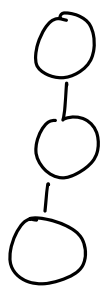
1) n-ary tree: nodes can have any number of children.

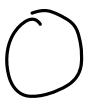
2) Binary Tree: nodes can have at max 2 children.

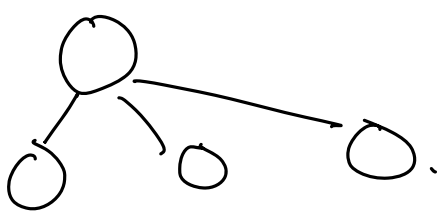
A node can have 0, 1 or 2 children.

Which one is a Binary tree

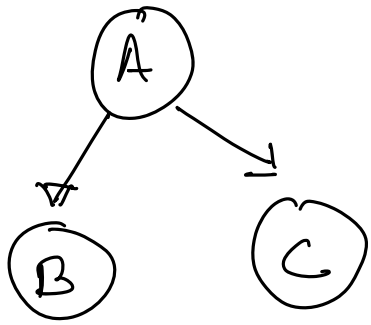
a) 

b) 

c) 

d) 

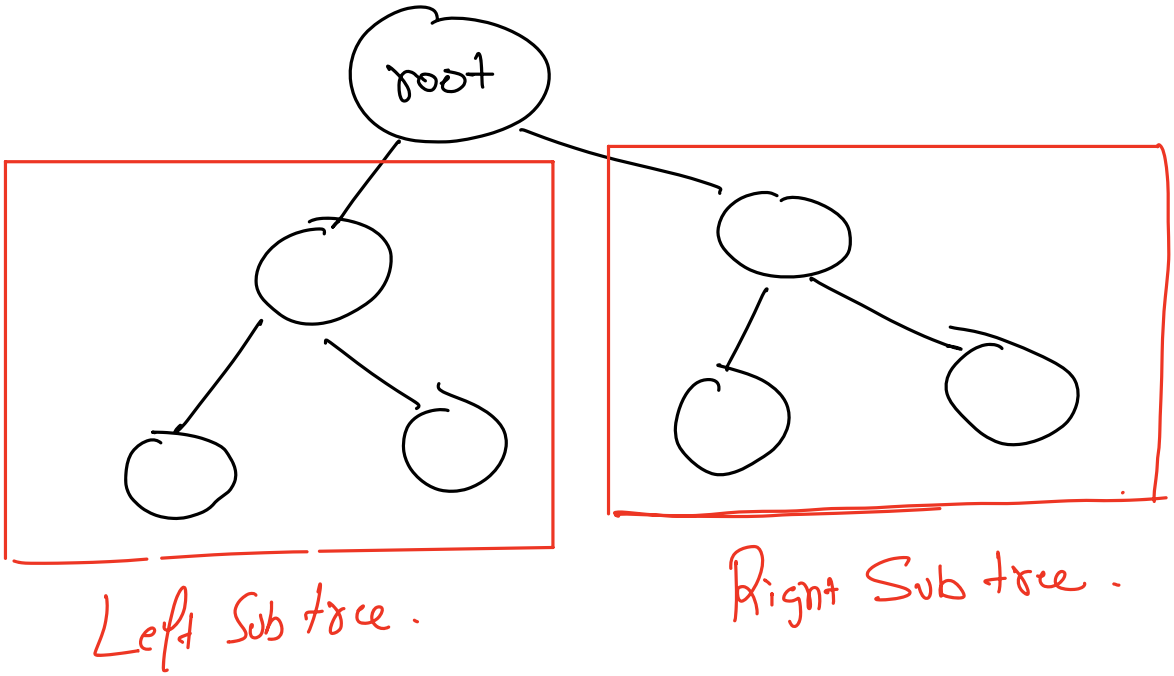
```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    TreeNode(int val) {
        this.val = val;
        this.left = null;
        this.right = null;
    }
}
```

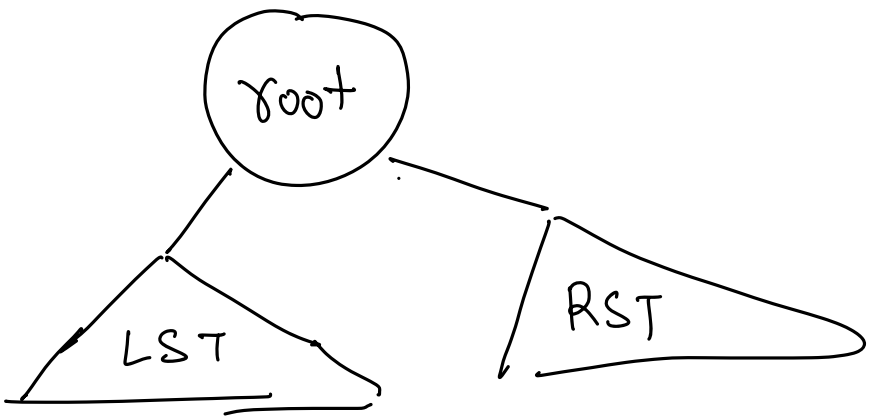


B: left child of A

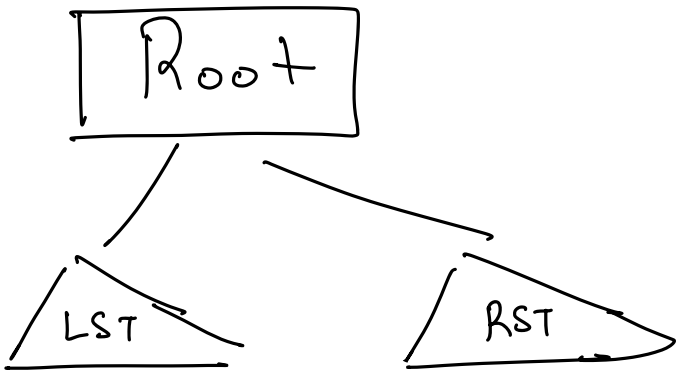
C: right child of A.

Binary Tree





Recursion will be used a lot to solve problems of trees because of the structure.



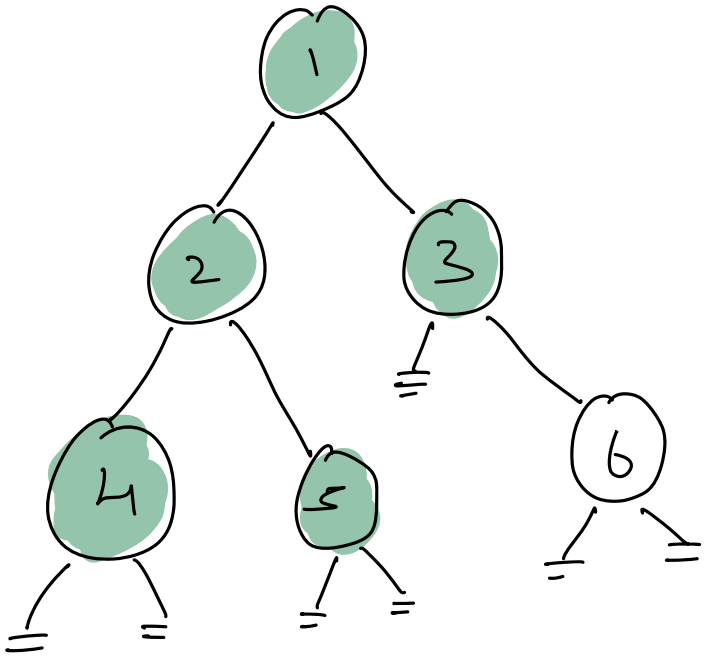
Possible ways of traversing.

Root LST RST	Root RST LST	LST Root RST	LST RST Root	RST LST Root	RST Root LST

LST will always be traversed before RST !

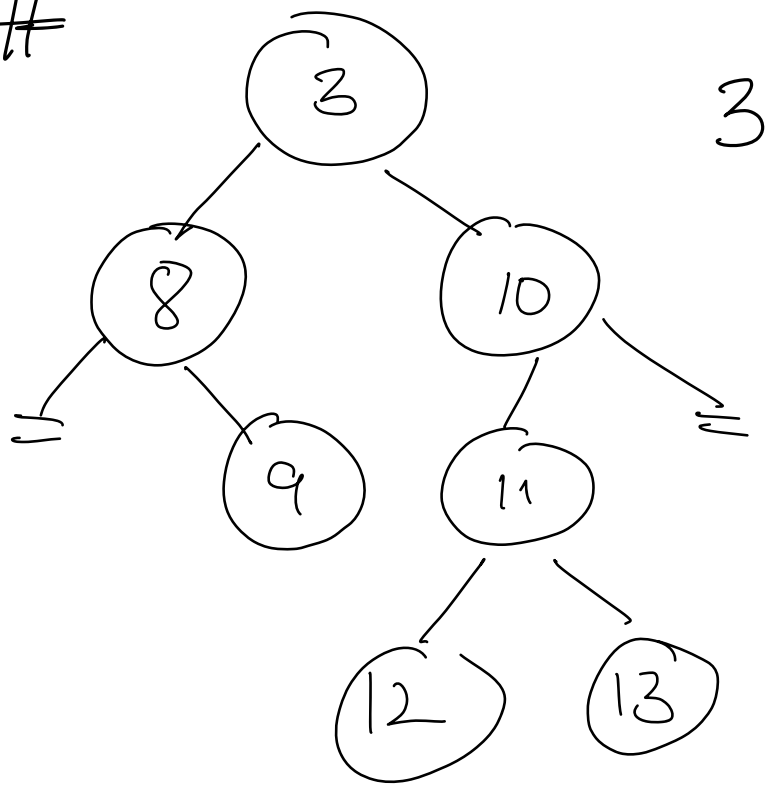
Root LST RST	~~Root RST LST~~	LST Root RST	LST RST Root	~~RST LST Root~~	~~RST Root LST~~
↓		↓	↓		
Preorder		Inorder	Postorder		

Pre-order ⇒ Root LST RST



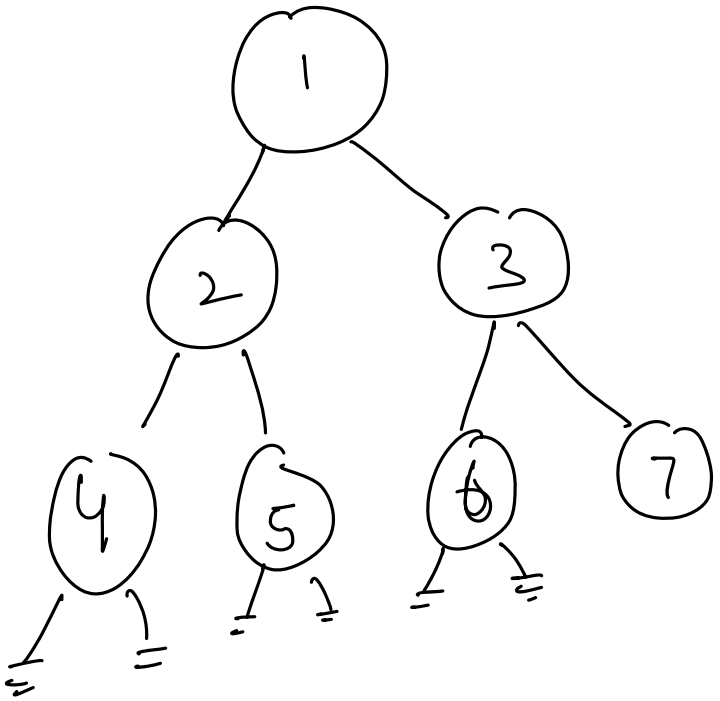
1, 2, 4, 5, 3, 6

#



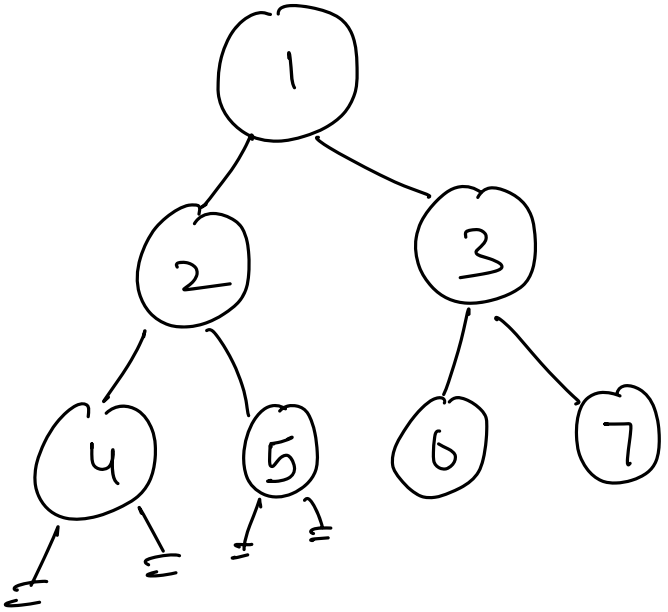
3, 8, 9, 10, 11, 12, 13

Inorder ⇒ LST, Root, RST



4, 2, 5, 1, 6, 3, 7

Post order : LST, RST, Root



4, 5, 2, 6, 7, 3, 1

```
void pre_order (Node root) {
    if (root == null)          1
        return;

    print (root.val);          2
    pre_order (root.left);     3
    pre_order (root.right);    4.
}
```

3

1) Preorder = 1, 2, 3, 4

2) Inorder = 1, 3, 2, 4

3) Postorder = 1, 3, 4, 2

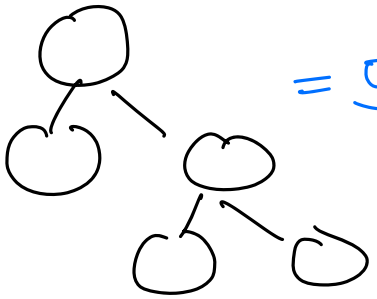
TC: O(n)

SC: O(h)
↳ height of tree.

SC: O(n): Worst Case.

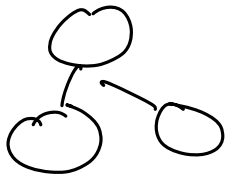
Q1 Find the number of nodes in a tree.
You will be given the root.

Ex1



= 5

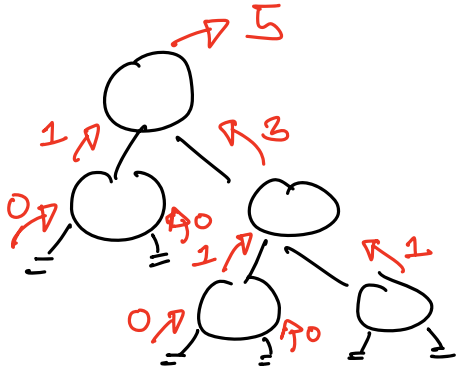
Ex2



= 3

```
int count_nodes (Node root) {
    if (root == null)
        return 0

    return count_nodes (root.left)
           +
           count_nodes (root.right)
           +
           1
```

3

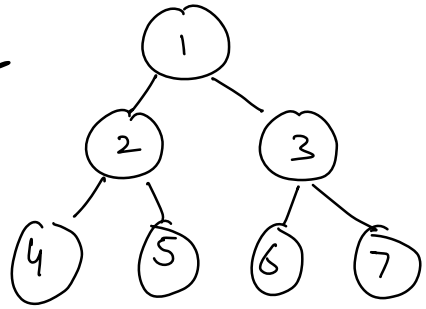


TC : O(n)

SC : O(n)

Q2 Find the sum of all nodes in a tree.

Ex1



⇒ 28

```
int count_nodes ( Node root) {
    if (root == null)
        return 0

    return count_nodes (root.left)
           +
           count_nodes (root.right)
           +
           root.val;
}
```

Q3 Find height of a the tree TODO

↳ height (root)!

Handle Base Case poopesly!

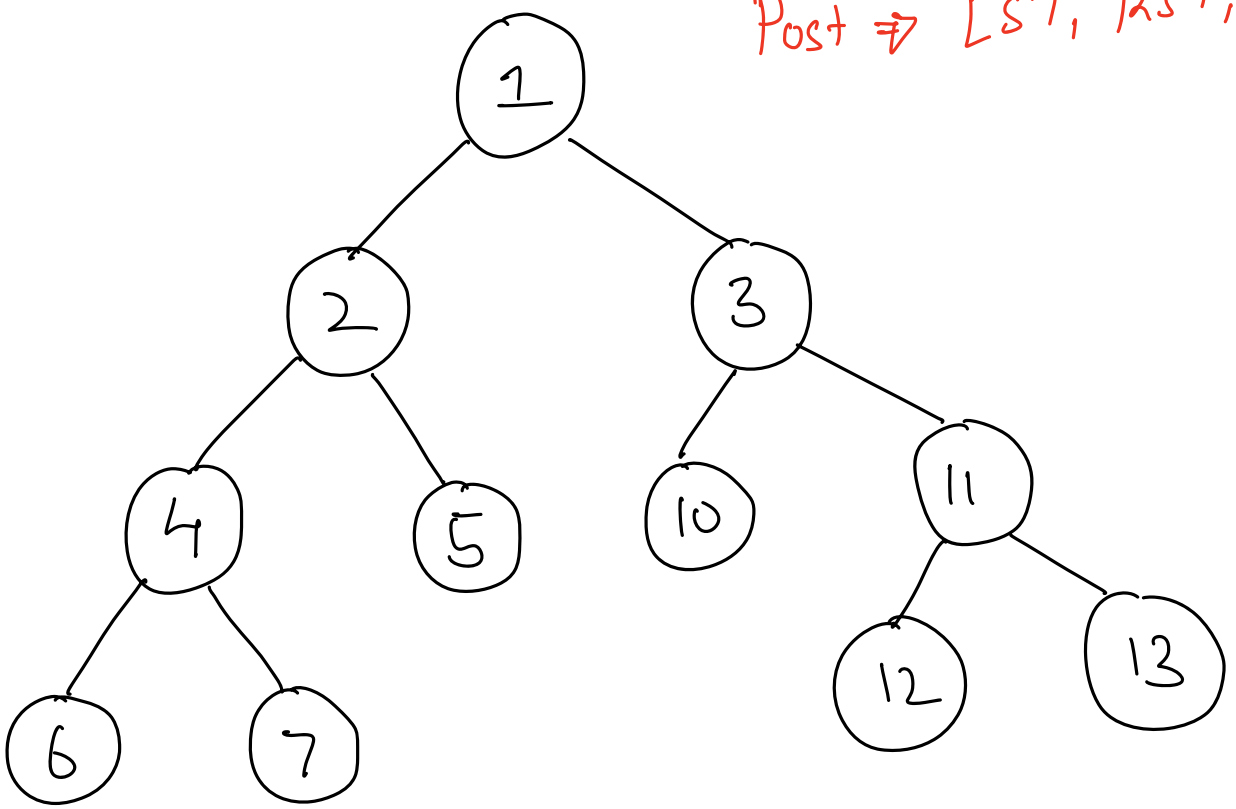
Q3 Given the inorder and postorder traversal of a tree, Construct the binary tree. All values are distinct.

	0	1	2	3	4	5	6	7	8	9	10
in[] =	6	4	7	2	5	1	10	3	12	11	13

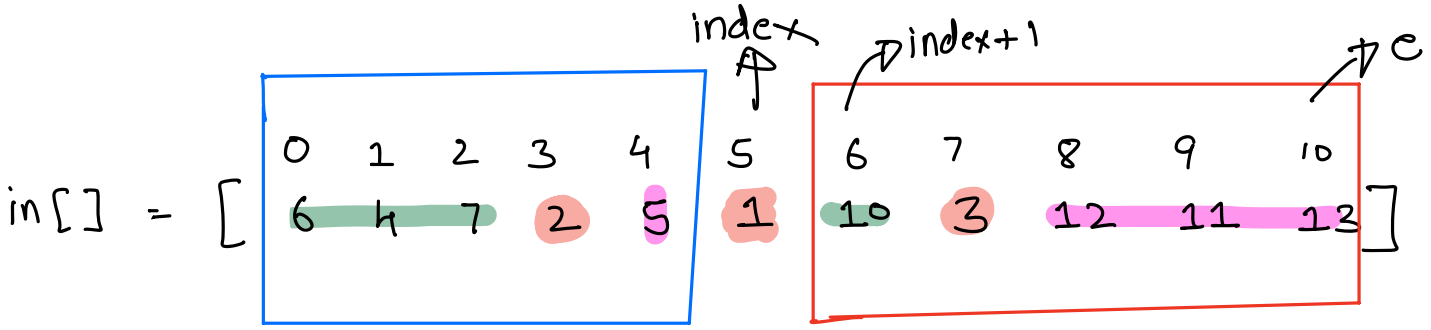
	0	1	2	3	4	5	6	7	8	9	10
post[] =	6	7	4	5	2	10	12	13	11	3	1

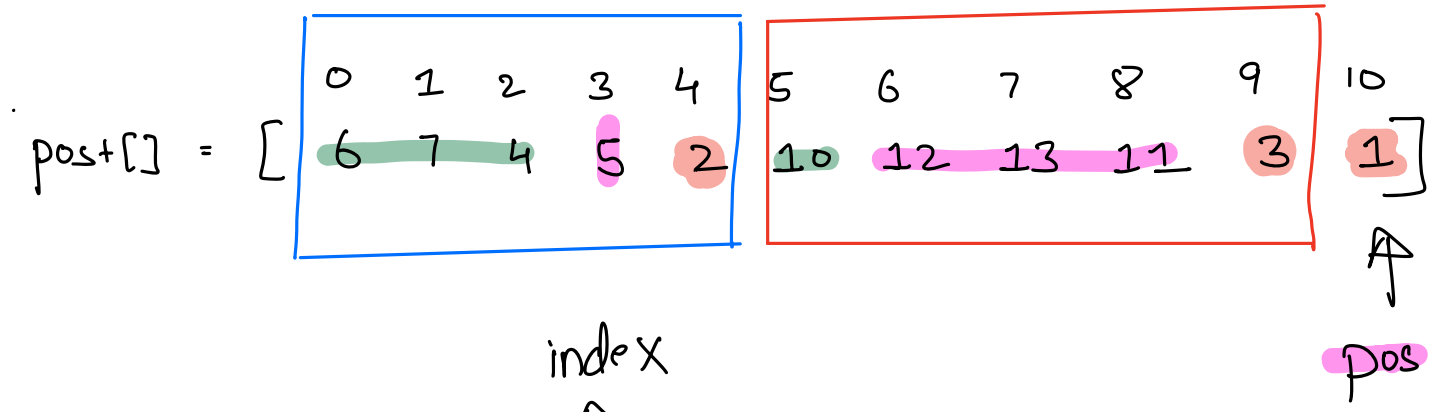
In ⇒ LST, Root, RST

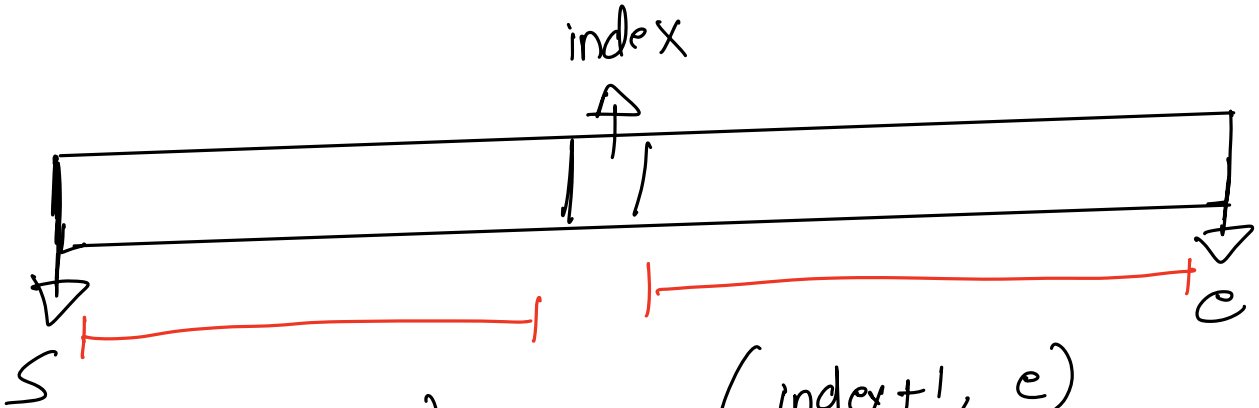
Post ⇒ LST, RST, Root



→ Last Element of Post-order traversal will be root.







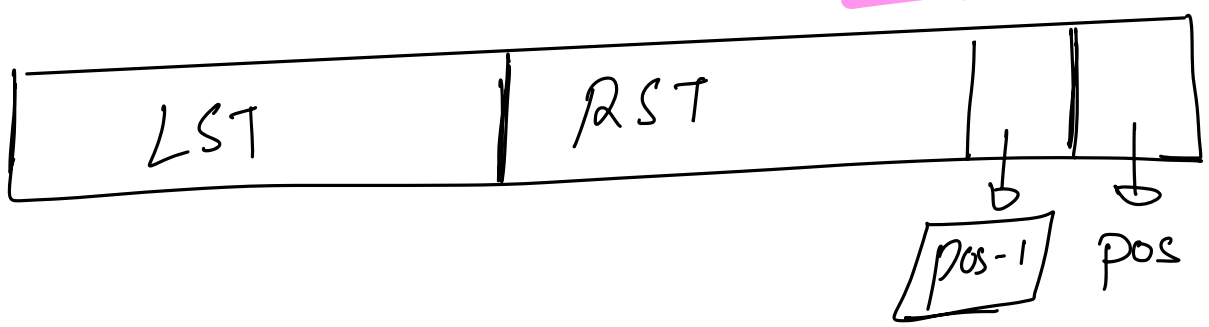
(s, index-1)

(index+1, e)

[index+1, e]

= e - (index+1) + 1

= e - index



initial call : construct Tree (0, n-1, n-1)

Pseudo Code !

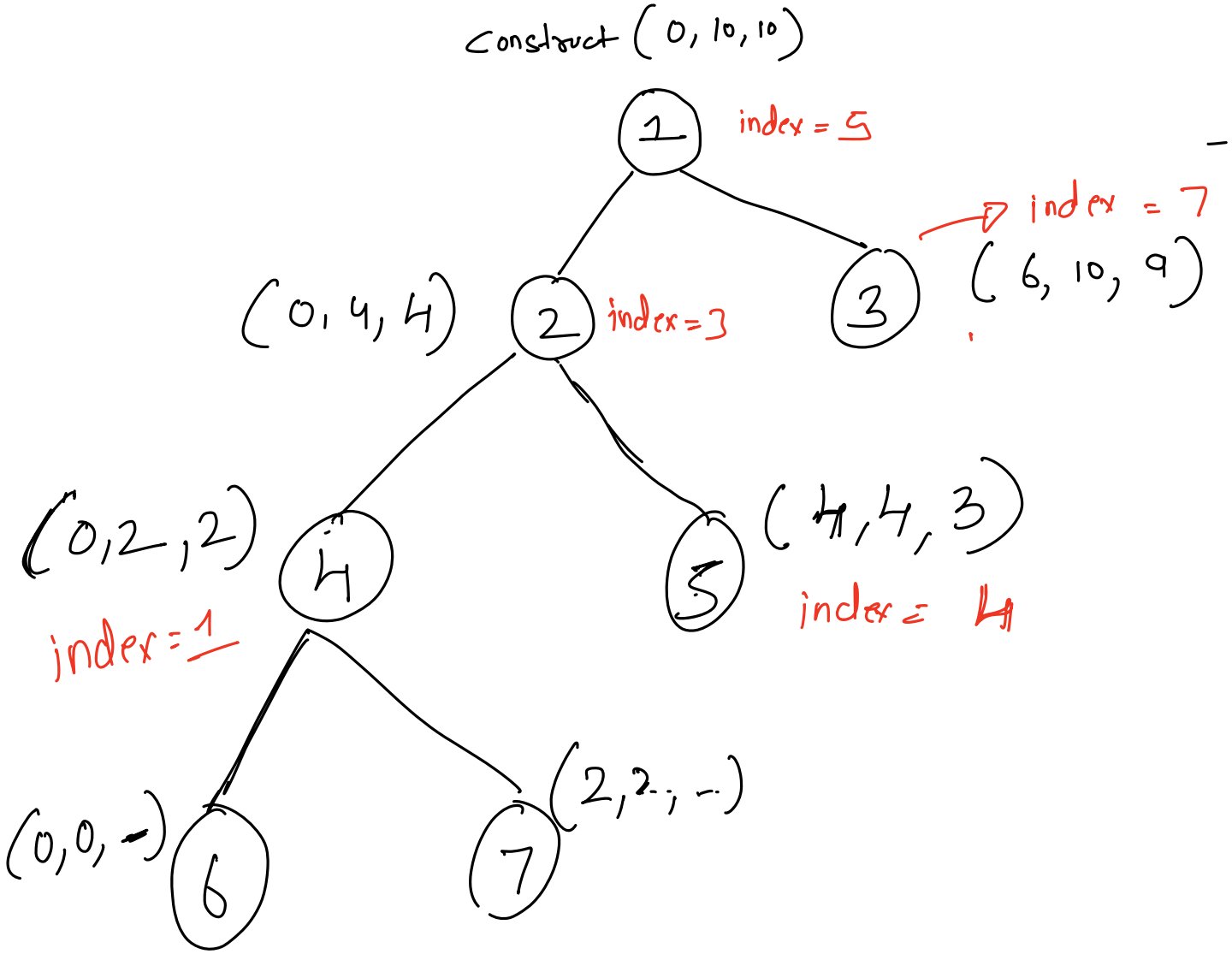
```
in[], post[];  map<int, int> mp:

Node constructTree(int s, int e, int pos) {
    int root_val = post[pos]
    Node root = new Node(root_val);
    int index = mp[root_val];

    root.left = ConstructTree(s, index-1, pos-(e-index)-1);
    root.right = ConstructTree(index+1, e, pos-1);

    return root;
}
```

Base Case: `if (s == e) {`

```
        Node root = new Node(in[s])
        return root;
    }
if (s > e)
    return null;
```

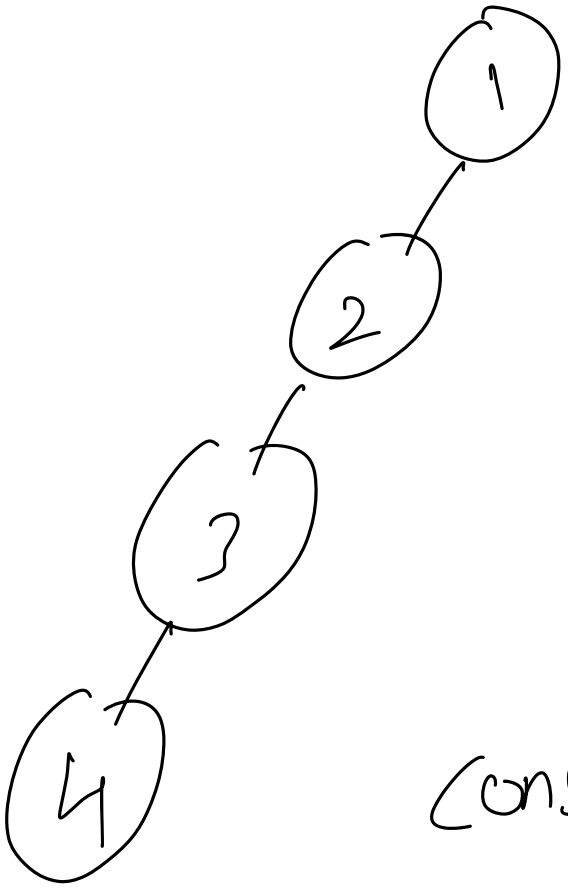
	0	1	2	3	4	5	6	7	8	9	10
in[] =	6	4	7	2	5	1	10	3	12	11	13

	0	1	2	3	4	5	6	7	8	9	10
post[] =	6	7	4	5	2	10	12	13	11	3	1



TC : O(n)

SC : O(n) — Hashmap
— Stack Space.



in ⇒ 4 3 2 1

post ⇒ 4 3 2 1

construct (0, 3, 3)

const (0, 2)

const (4, 3)

size