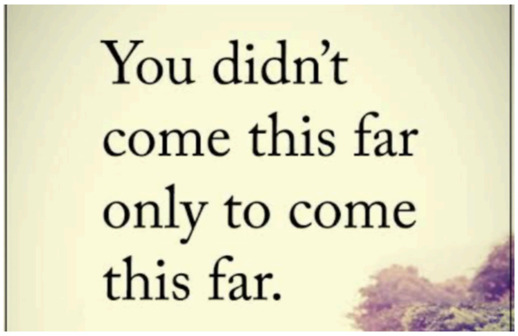


Today's Content

→ Trees intro

→ Naming Convention

→ Tree Traversal

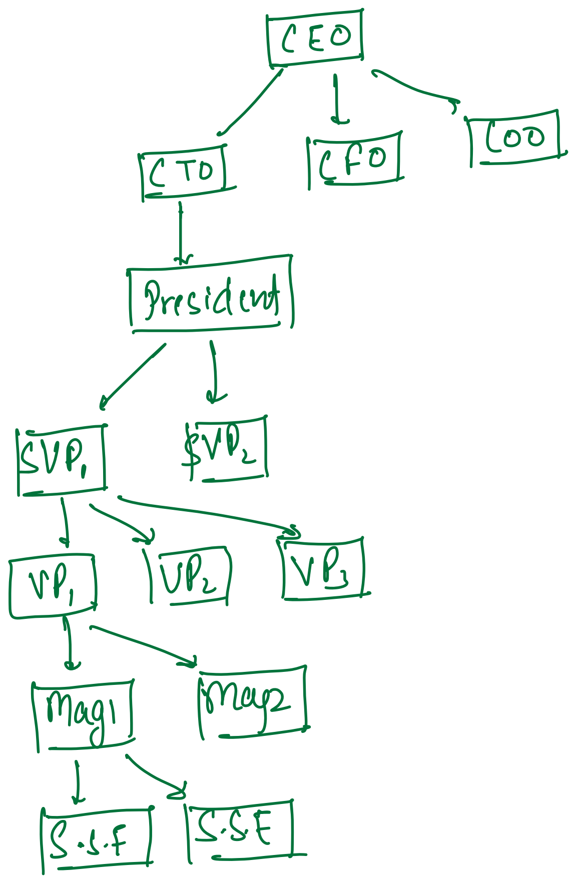
→ Basic Tree problems.

Linear Data Structure.

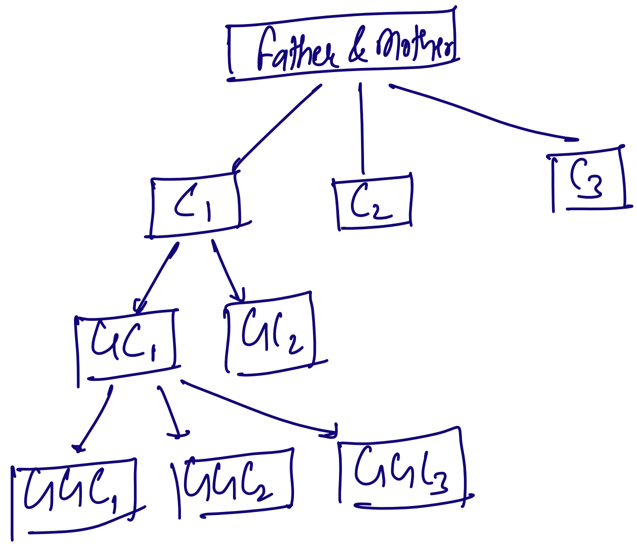
Arrays, linked list, stack, Queue.

Hierarchical D.S. (Tree)

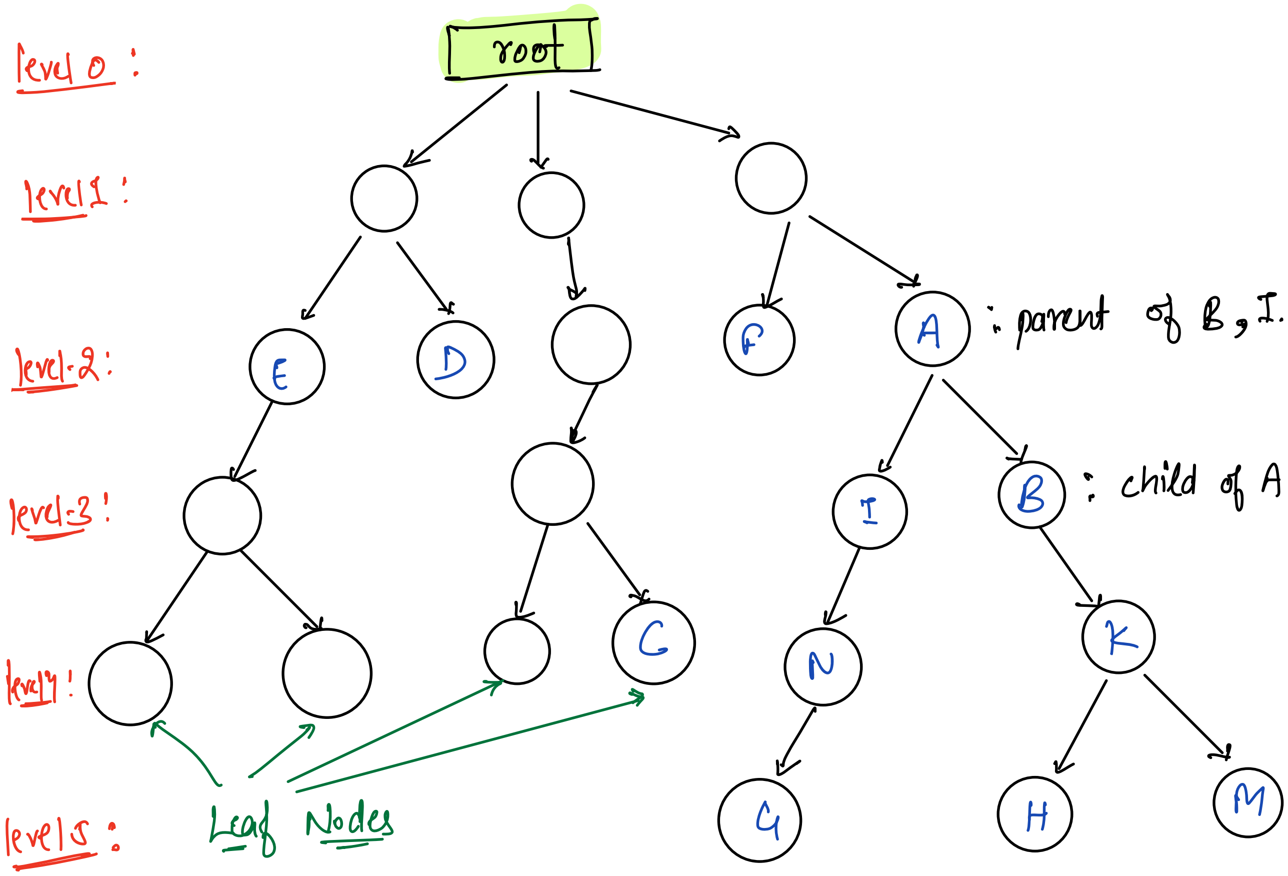
Company organisation



Family Tree



Eg → folder Structure.



① A is ancestor of B, I, N, K, G, H, M

② I, B, N, K, G, H, M are descendents of A.

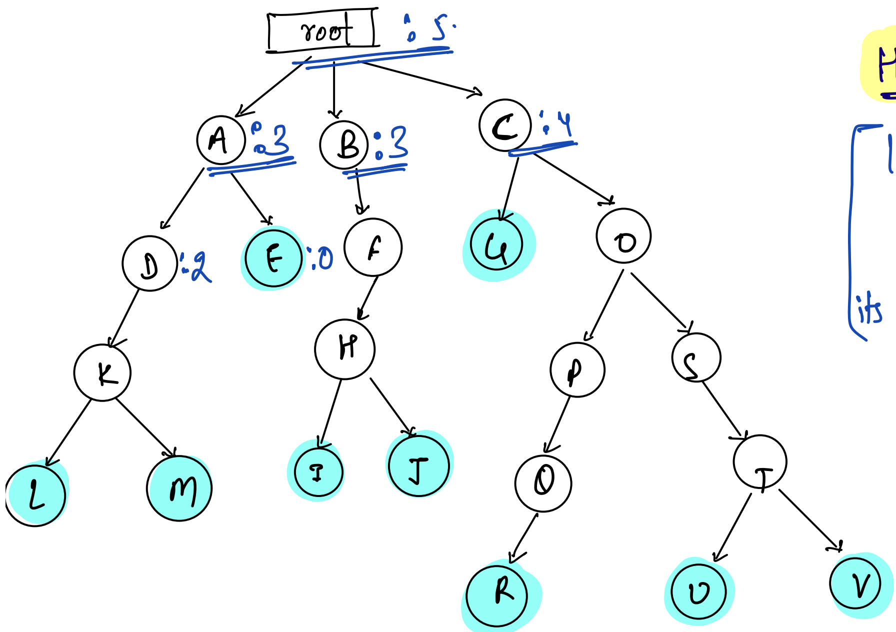
③ I and B → children of same parent.
∴ Sibling nodes.

④ Nodes which don't have any child are known as leaf node.

⑤ Root node → node present at level-0
→ node which doesn't have parent node.
→ node which is ancestor of all other nodes.

Height of a Node.

[length of the longest path from node, to any of its descendent leaf node.]



[Note → Path is calculated based on edges only.]

$H(A) = 3$

$H(B) = 3$

$H(C) = 4$

obs 1.

$ht(\text{node}) = \max(ht \text{ of all children}) + 1$

obs 2.

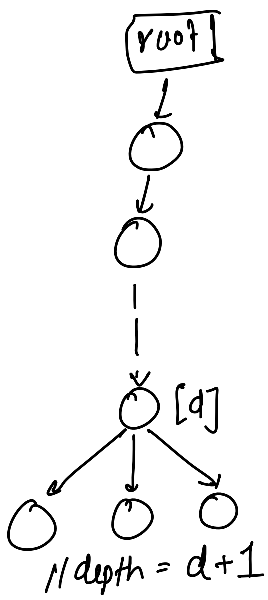
$ht(\text{leaf node}) = 0$

depth of the node.

length of path from root node to curr node.

$d(\text{root}) = 0$

$d(A) = d(B) = d(C) = 1$



obs1:

depth(node) = 1+ depth of its parent node.

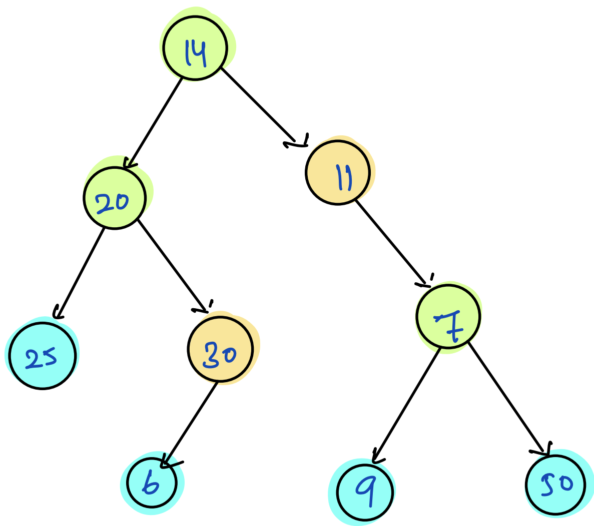
obs 2.

depth(root) = 0

obs3.

[depth = level]

Binary Tree [2]. Every node can have at-max 2 children.
0, 1, 2, ~~3, 4, 5 ...~~



→ leaf nodes.

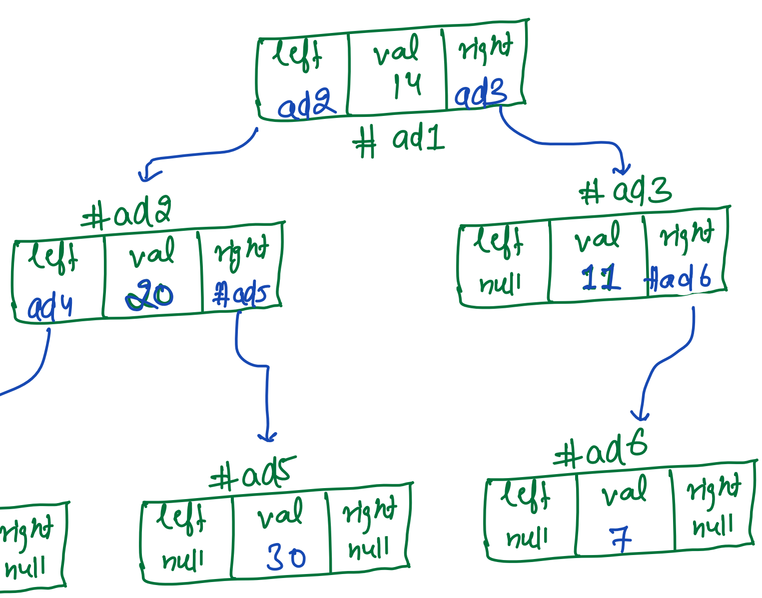
→ having exactly 2 children.

→ nodes with single child.

```
class Node {
    int val;
    Node left;   // obj. reference, holds address of node object
    Node right;  // obj. reference, holds address of node object.

    Node (x) {
        val = x
        left = null
        right = null
    }
}
```

```
Node root = new Node(14);
```




```
root.left = new Node(20);
root.right = new Node(11);
```

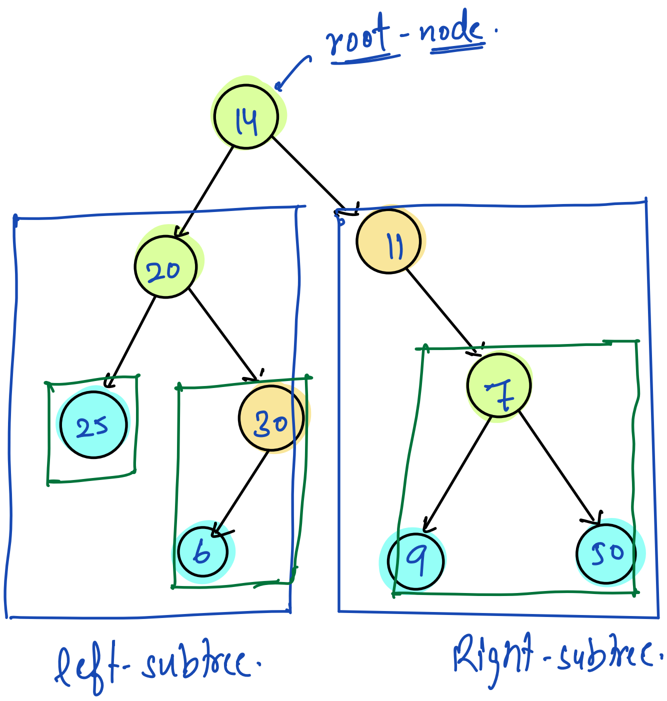
obs.: If root node is given, we can traverse entire tree.

//Note:→ for tree construction, we are going to use serialization & de-serialization.

[#Advance module]

Traversals → [pre-order, in-order, post-order] ← Int.

[level-order, vertical-level order, diagonal traversal] ← Adv.



= root.left

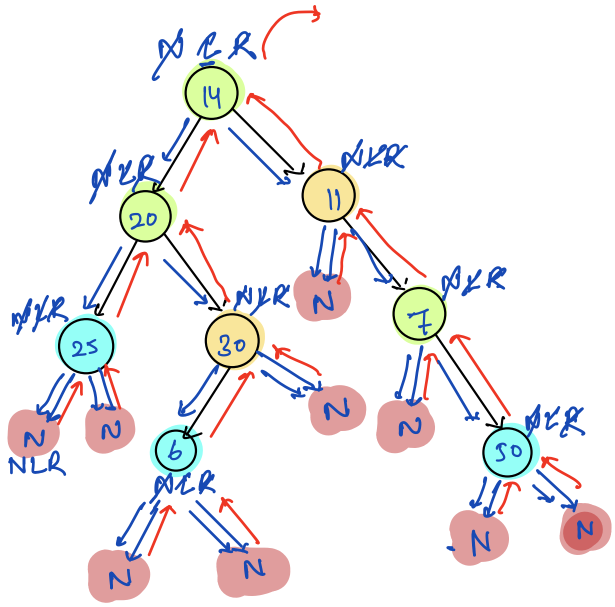
(20) → root of l.s.t

(11) → root of r.s.t

= root.right

Tree Traversal.

→ pre-order. [N L R]

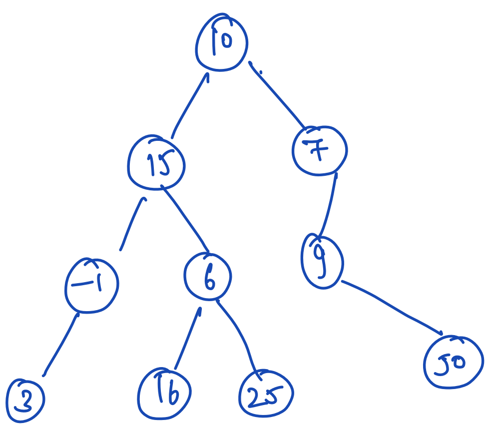


step1: Print root.val

step2: go to left sub-tree & print entire left sub-tree in pre-order.

step3: go to right sub-tree & print entire right sub-tree in pre-order.

[o/p. → 14, 20, 25, 30, 6, 11, 7, 50]



(NLR)
pre → 10, 15, -1, 3, 6, 16, 25, 7, 9, 50

[In-order → L N R
Post-order → L R N] #Todo

Pseudo-code.

// Assumption → Given root node, print entire tree pre-order.

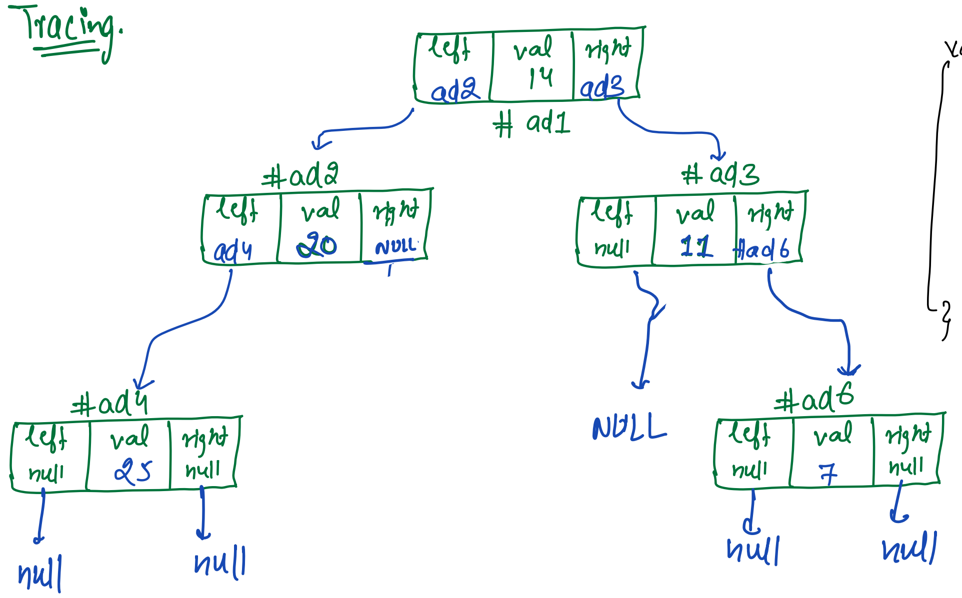
```
void pre-Order ( r ) {
    if (r == NULL) {return}
    print( r.val );
    pre-Order ( r.left );
    pre-Order ( r.right );
}
```

$\boxed{\begin{array}{l} T.C \to O(N) \\ S.C \to O(\text{ht of the tree}) \end{array}}$

$\Downarrow$

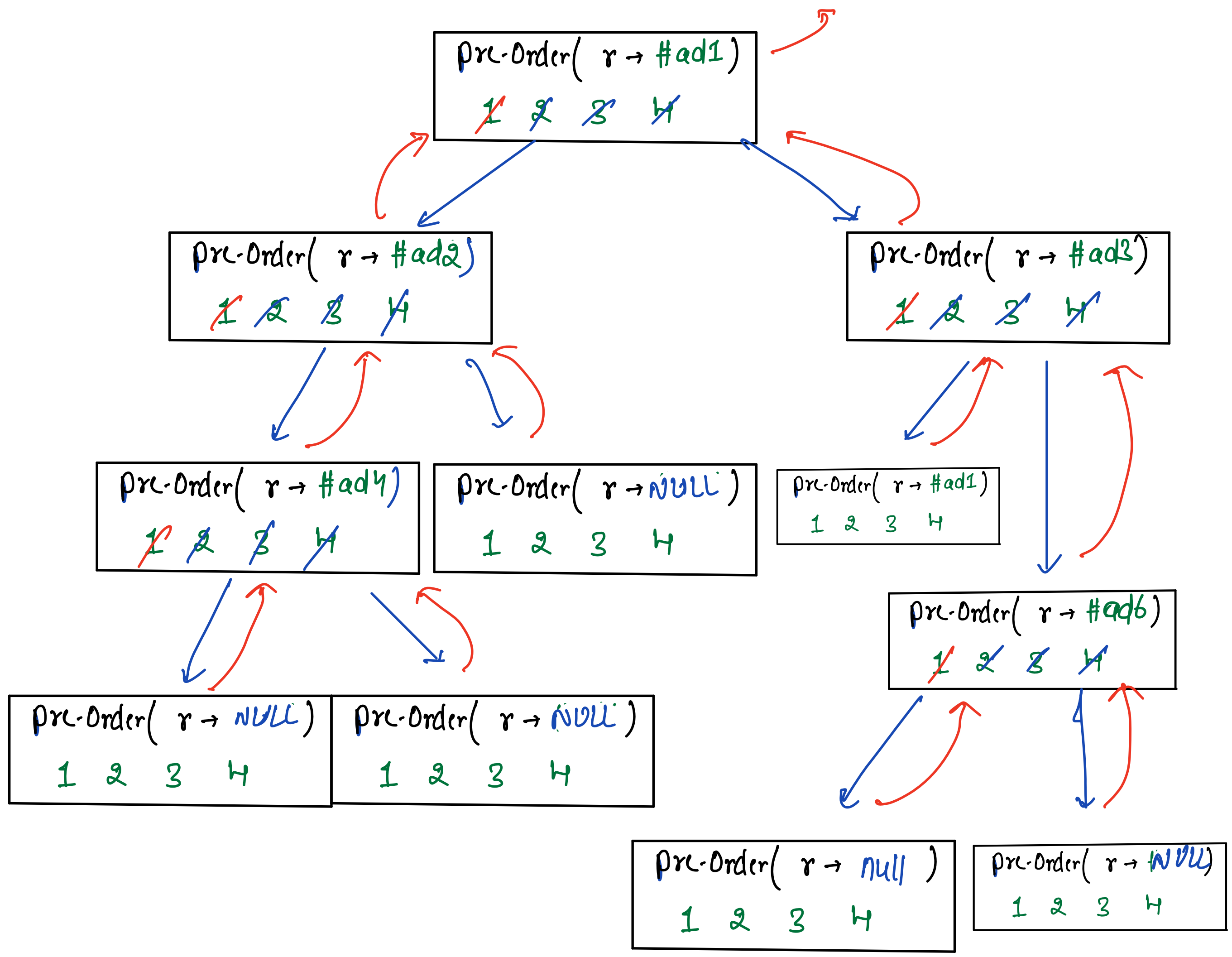
Stack size space used by recursion.

Tracing.




```
void pre-Order ( r ) {
    if (r == NULL) {return}
    print( r.val );
    pre-Order ( r.left );
    pre-Order ( r.right );
}
```

o/p [14, 20, 25, 11, 7]



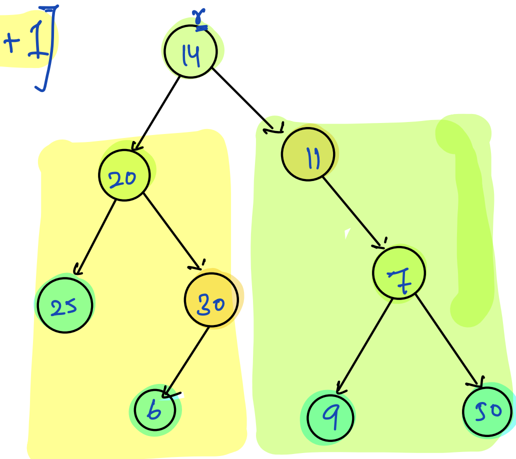
Tree Problems.

1. Size (Node root)
2. Sum (Node root)
3. Height (Node root) #todo

— Recursive codes only & you can't use Global variables.

① // Assumption Given root node, return no. of nodes.

[Size(tree) = Size(left sub tree) + Size(right subtree) + 1]



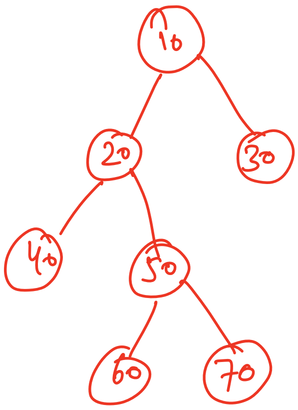

```
int size ( Node root) {
    if(root == null) { return 0}
    int l = size(root.left);
    int r = size(root.right);
    return l+r+1 ;
}
```

② // Assumption→ Given root node, return sum of all nodes.

Sum(All nodes in tree) = Sum(All nodes in left subtree) + Sum(All nodes in right sub-tree) + root.val.

```
int sum ( Node root) {
    if (root == null) {return 0}
    int l = sum(root.left);
    int r = sum(root.right);
    return l+r + root.val;
}
```

Note → In your assigments, length of path is calculated in terms of nodes.



[length from 10 → 60 ⇒ 4]

---

solve.

PF: (A)

System.out.println( );

1 - A

```
if (A == 1) {
        print( 1 + " ");
        return;
}
else {
        solve(A-1);
        print(A + " ");

}
```

```
fun( x, n){
    if (n == 0) return 1
    else if ( n % 2 == 0) {
        return fun(x * x, n/2);
    }
    else {
        return x * fun(x * x, (n-1)/2);
    }
}
```

$x \quad n$

$\underline{\underline{2, 10}}$

$$x^n = \underbrace{x * x * x * x --- x}_{n \text{ times}}$$