Binary Trees.

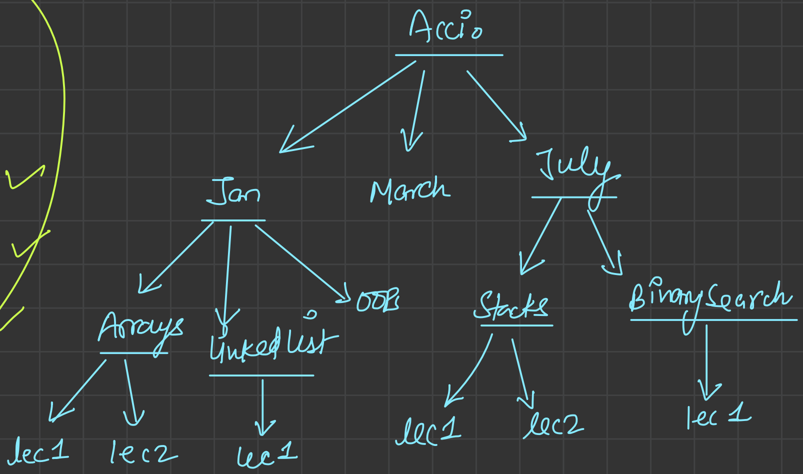
→ Non linear data structure.

Data.

1. family tree ✓
2. Org. chart ✓
3. file System ✓

→ Trees

Heirarchy present

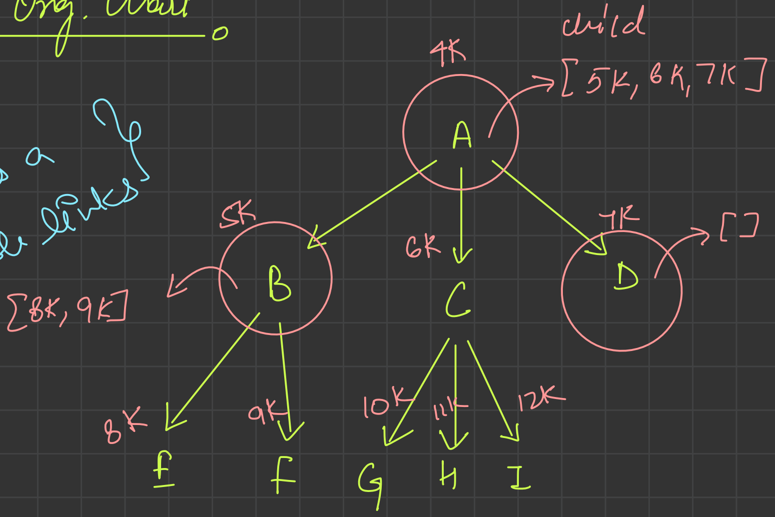


Non - linear

→ Trees!

Company Org. Chart

{Each node has a variable links}



```
class Node
{
    String data;
    ArrayList<Node> child;
}
```

generic Tree

datastructure

Binary Trees

0/1

└> two options

└> Atmost two childrens!

0, 1 or 2 childs!

named as

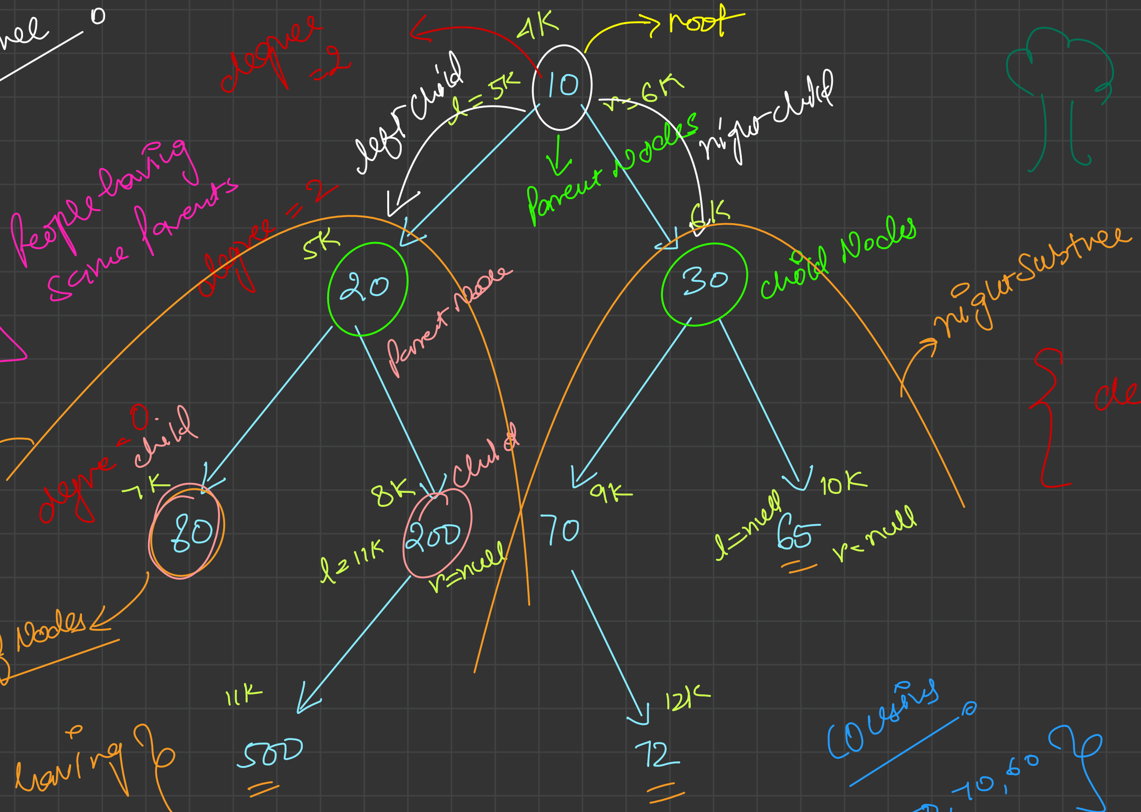
└> left child

└> right child }

```
class Node
{
  int data;
  Node left;
  Node right;
}
```

Binary Tree

Siblings → People having same parents
- 20, 30
- 80, 200
- 70, 65

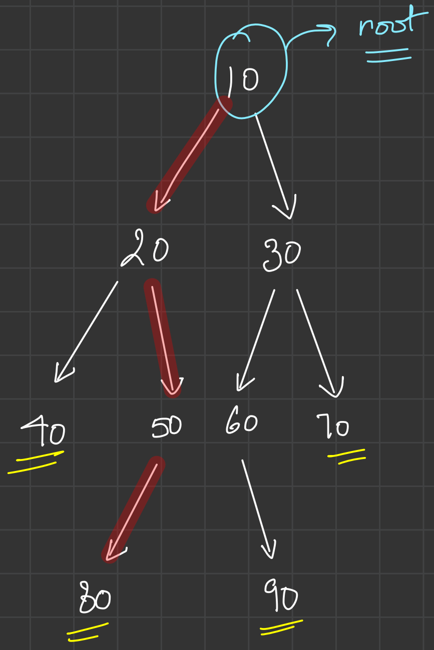


{ degree = No. of childs

left subtree

Leaf Nodes → { Nodes having 0 child }

Cousins
- 80, 200, 70, 60
- 500, 72



height of the tree

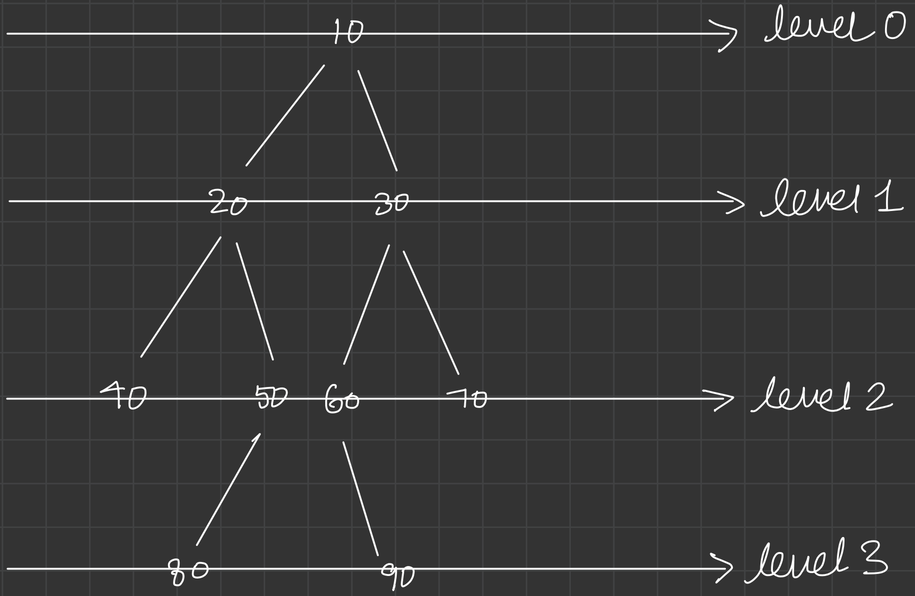
= dist b/w root Node and deepest leaf Node

No. of Edges

= 3

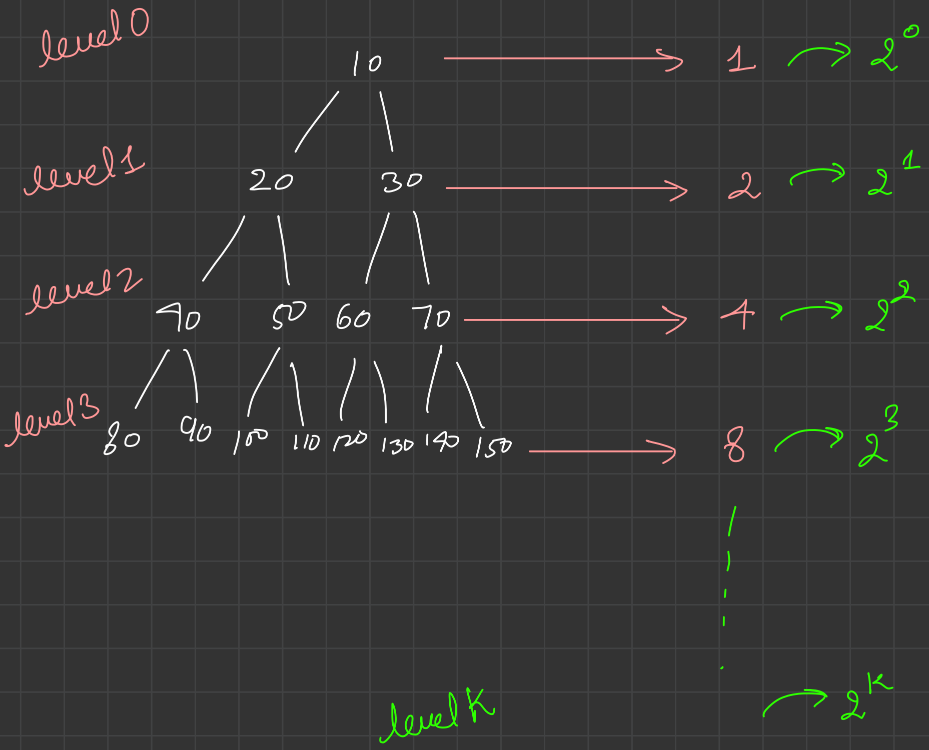
No. of Nodes

= 4


10
level 0
20
30
level 1
40
50
60
70
level 2
80
90
level 3

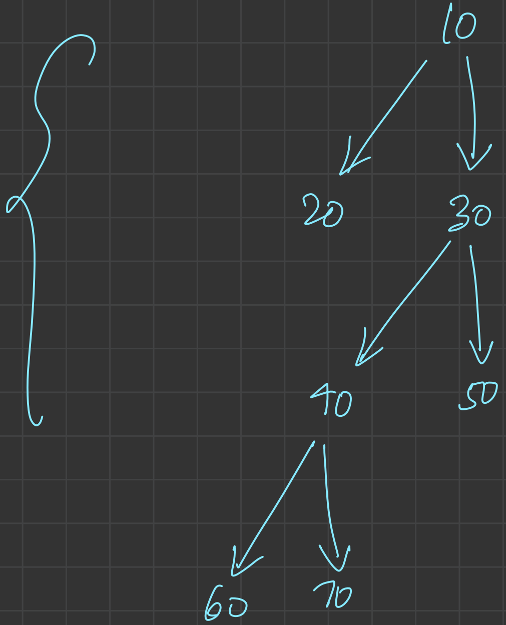
Perfect Binary Tree.

{ No. of people in each level (l) = 2^l }



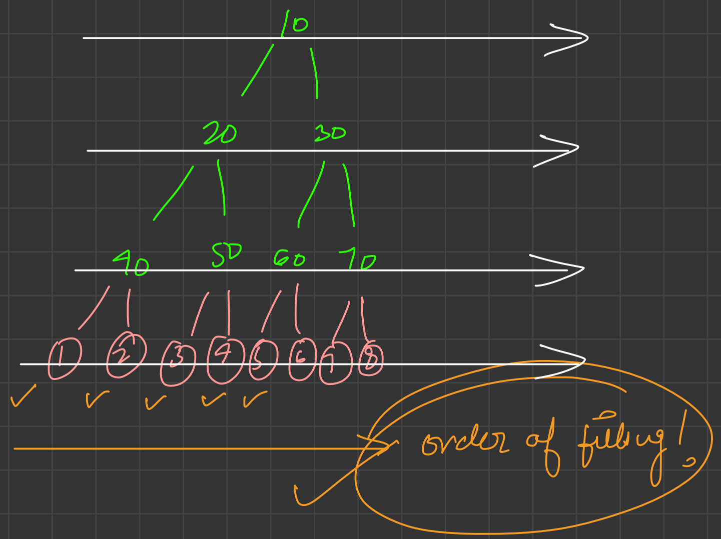
Full Binary Tree

→ Each Node will Either have 0 or 2 children



Complete Binary Tree

where each level is completely filled except, people in last level are as left positioned as possible



Balanced Binary Tree

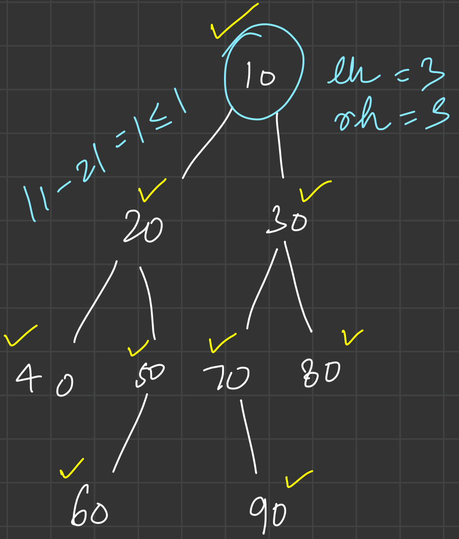
↳ when each node of the tree is balanced!

Balanced Node

↳ $\boxed{|lh - rh| \le 1}$ ✓

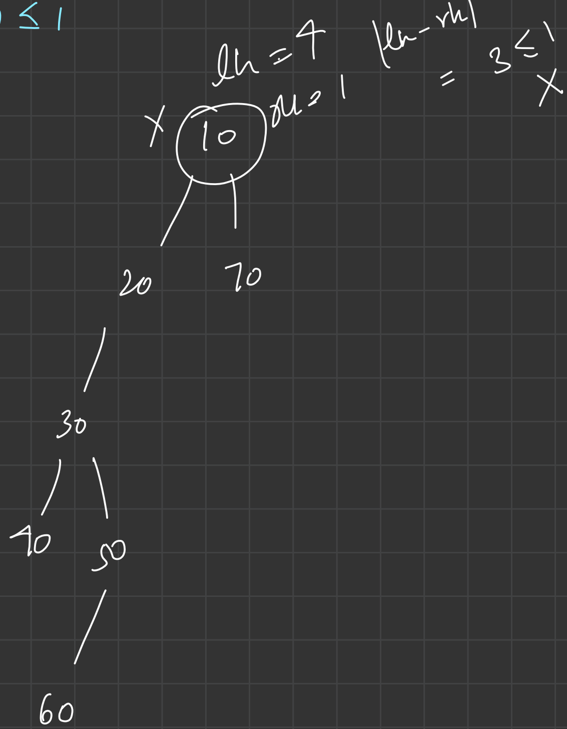
{ left sub tree height

right subtree height }



|lh-rh| = 0 ≤ 1

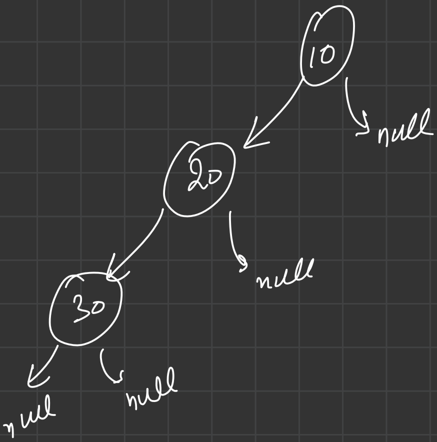
Yes tree is Balanced!



Skew Tree.

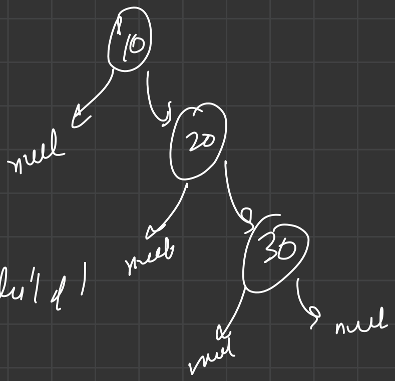
left Skew Tree.

→ where each node either left child or No child!



right-skew tree.

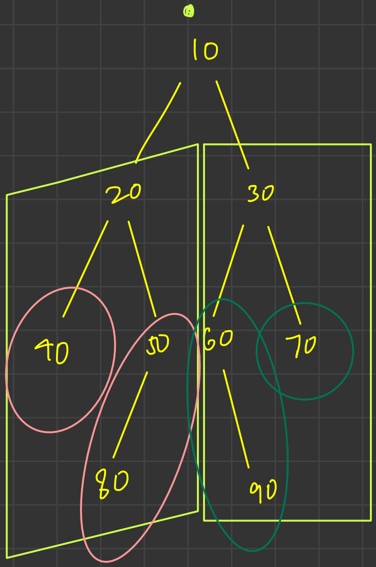
→ either right child or No child!

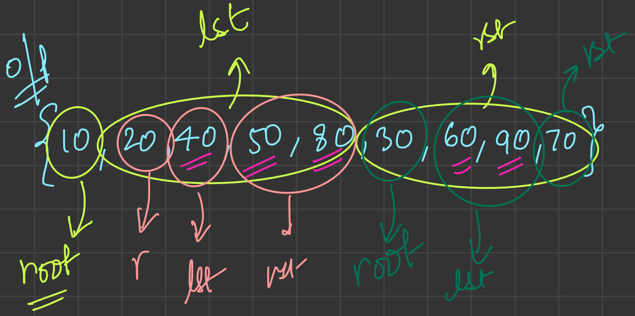


Traversal on trees.

(1.) Pre Order Traversal.

→ root
→ lst
→ rst }.



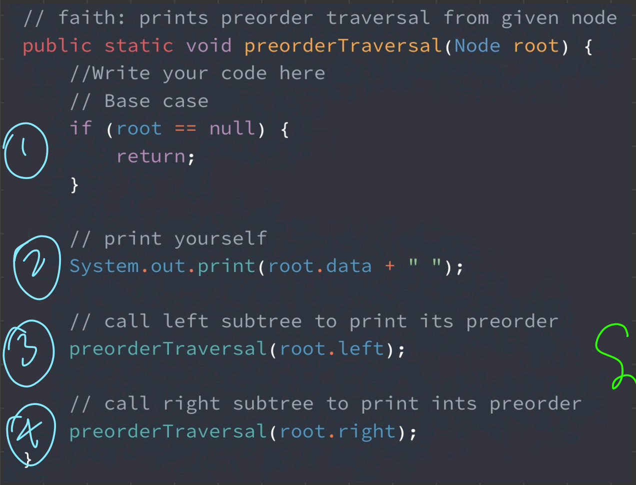


o/p {10, 20, 40, 50, 80, 30, 60, 90, 70}

```
print (root)
preOrder( lst)      } → preOrder
pre Order (rst)
```

faith: prints preorder from given root.

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

        print (root);

        preOrder( root. left);

        preOrder(root . right);

}
```



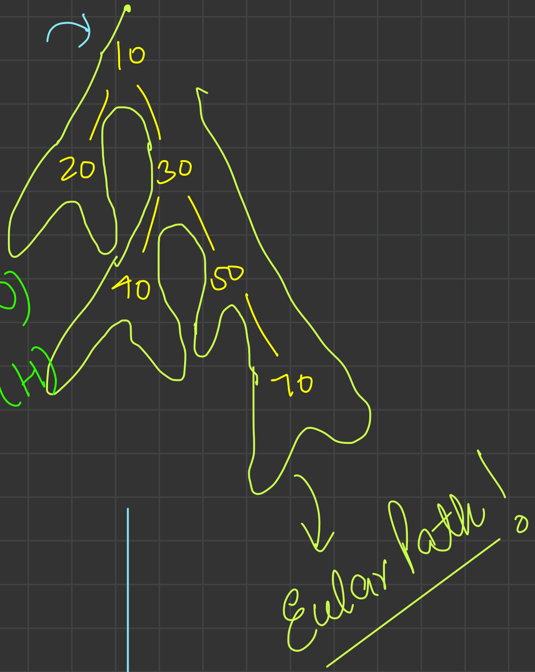
```
// faith: prints preorder traversal from given node
public static void preorderTraversal(Node root) {
    //Write your code here
    // Base case
    if (root == null) {
        return;
    }

    // print yourself
    System.out.print(root.data + " ");

    // call left subtree to print its preorder
    preorderTraversal(root.left);

    // call right subtree to print ints preorder
    preorderTraversal(root.right);
}
```

1

2

3

4



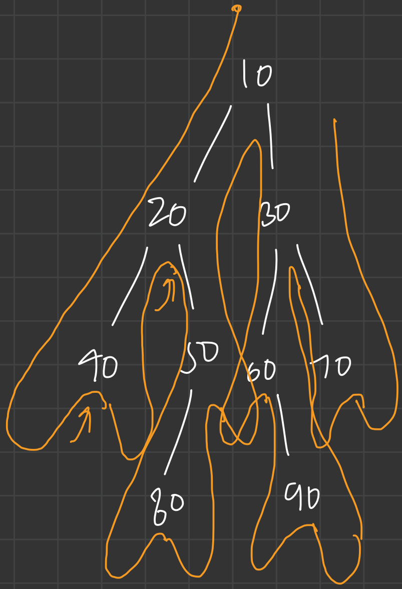
TC : O(N)

SC : O(H)

o/p

10, 20, 30, 40, 50, 70

In Order Traversal → lst + root + rst.



o/p ↳ 40, 20, 80, 50, 10, 60, 90, 30, 70

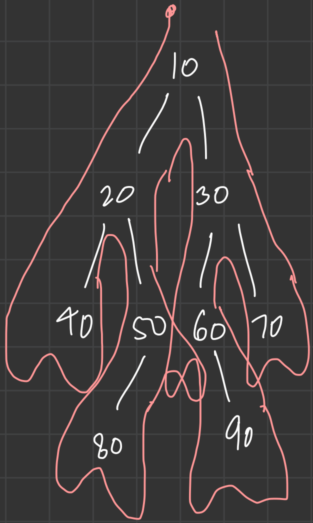
lst root rst

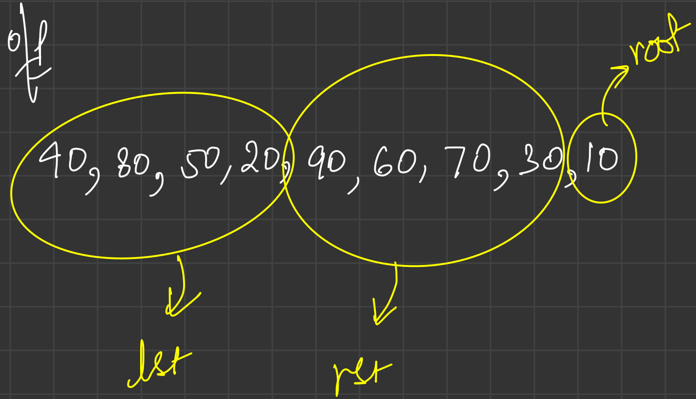
```
void inorder ( Node root)
{
        Base Case !
    inorder (root.left);
    print (root);
    inorder (root.right);
}
```

(40, 20, 80, 50, 10, 60, 90, 30, 70)

Post Order Traversal.

→ left, right, root.



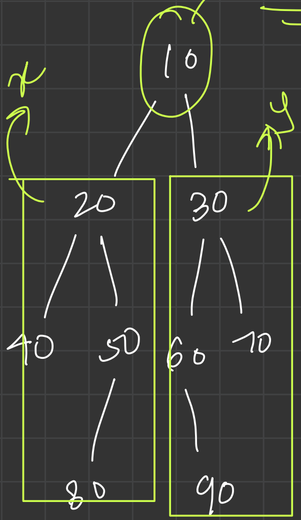


40, 80, 50, 20, 90, 60, 70, 30, 10

{40, 80, 50, 20, 90, 60, 70, 30, 10}

Size of Tree. {No. of Nodes}

$x + y + 1$ = size



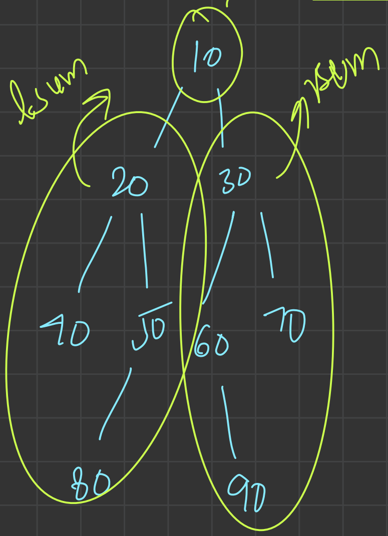
size = 9

faith: returns size of the tree from root

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

    int lstSize = size(root.left);
    int rstSize = size(root.right);
    return lstSize + rstSize + 1;
}
```

Sum of tree

↳ Sum of values of Each Node!



lsum + rsum + root.data

overall sum!

Sum = 450

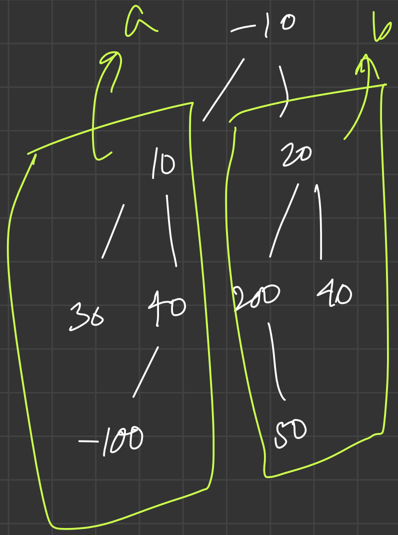
faith: returns Sum of tree starting from root!

```
int Sum(Node root)
{


}
```

Max^{m} in a tree

↳ { max^{m} value present in a tree }

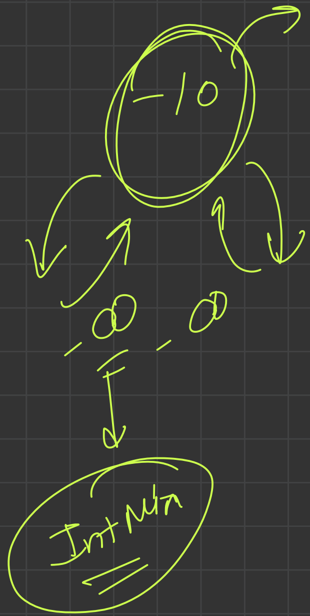
max(a, b, root)



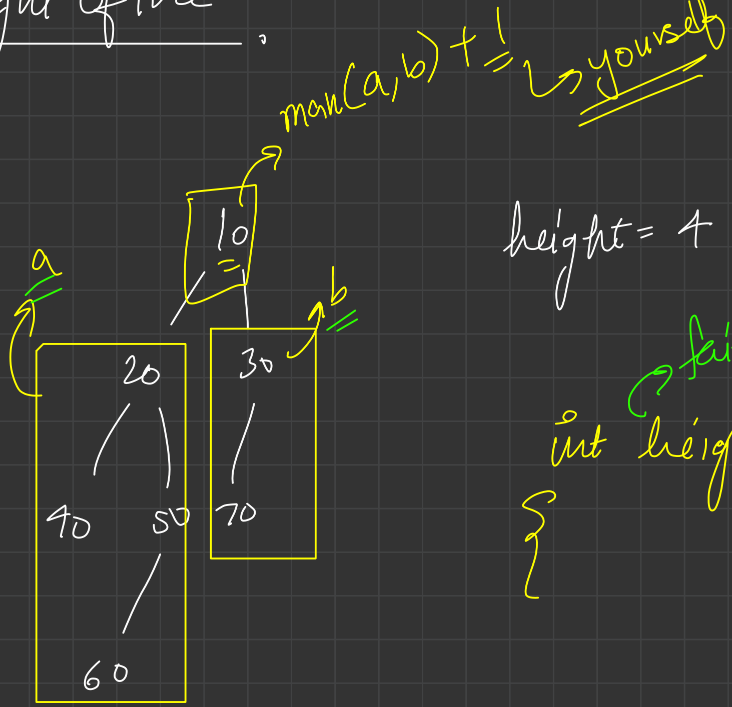
faith: returns max^{m} value in the tree from root.

```
int maxOfTree(Node root)
{

}
```

Base Case



height of tree



height = 4

faith: returns height of tree starting from root.

```
int height(Node root)
{

}
```