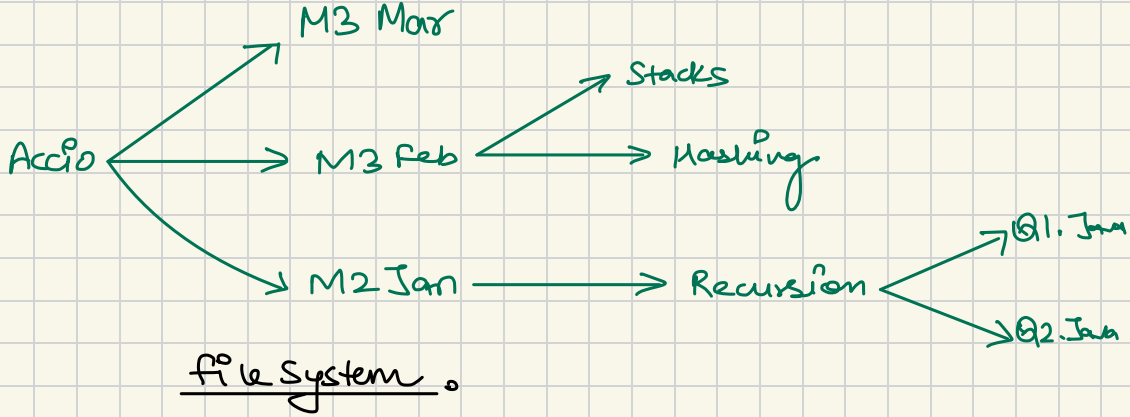
Binary Trees

→ Non-linear Data Structure

Data

- familychart
- filesystem

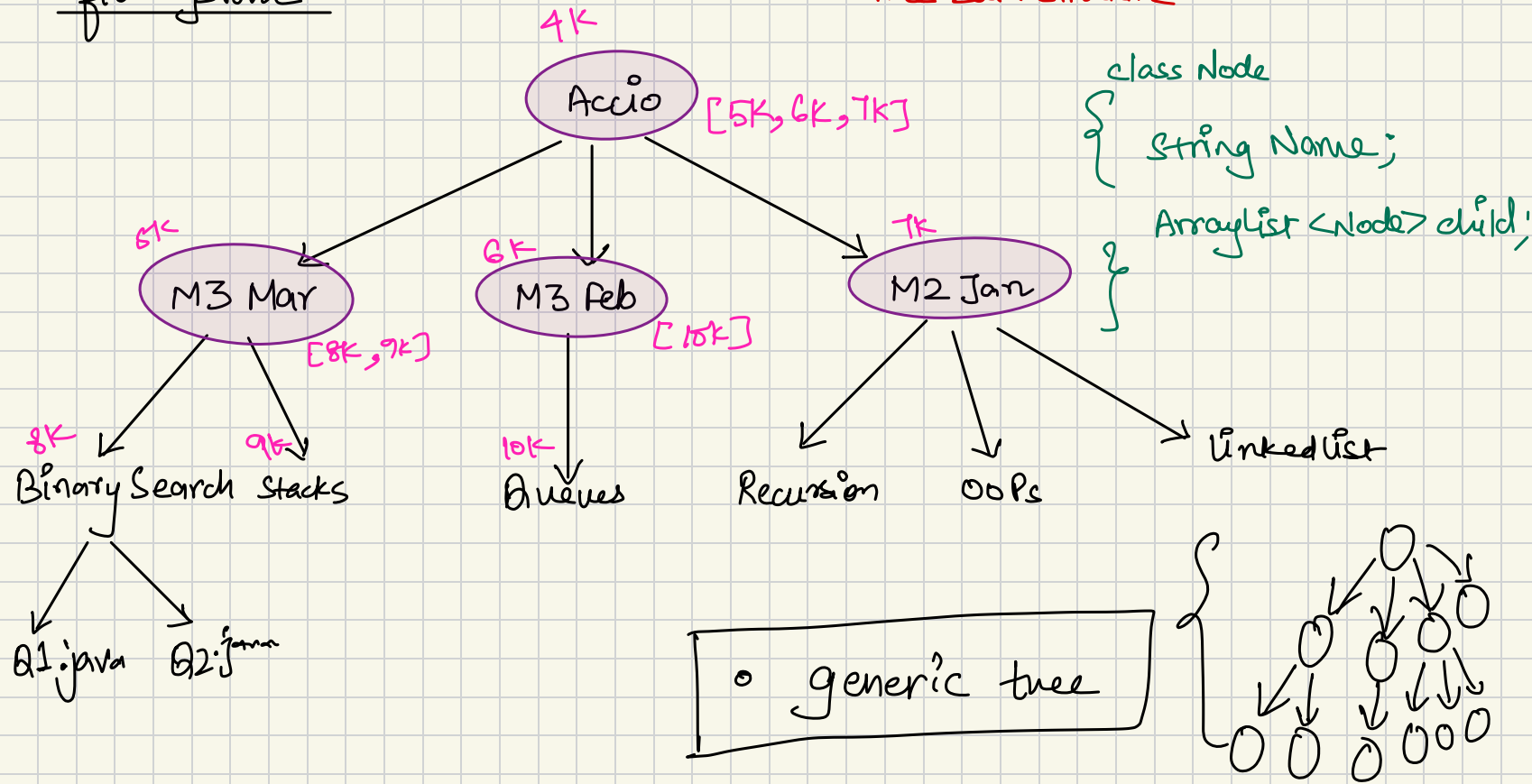
} Hierarchy



file System

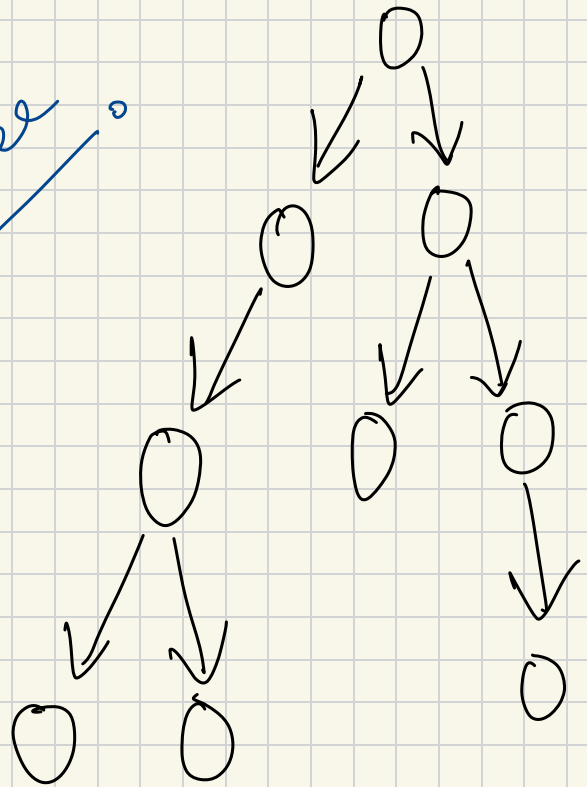
file system

Tree Data Structure



class Node
{
String Name;
Arraylist <Node> child;
}

Binary Tree



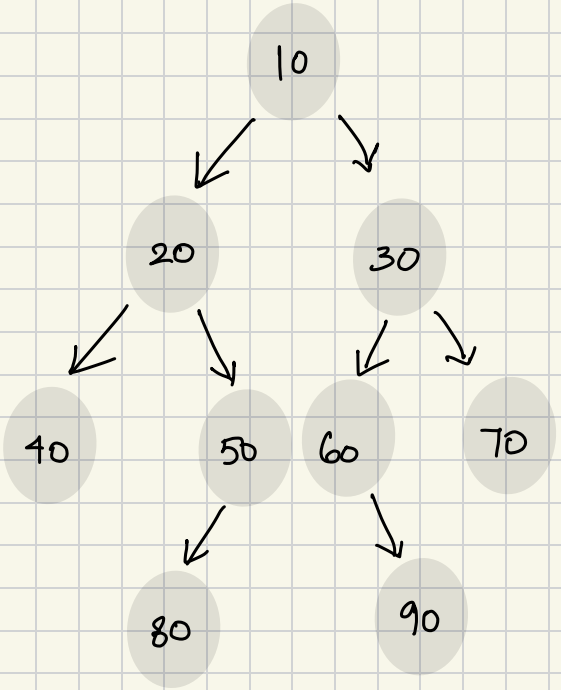
{ Each person can have max 2 child

↓

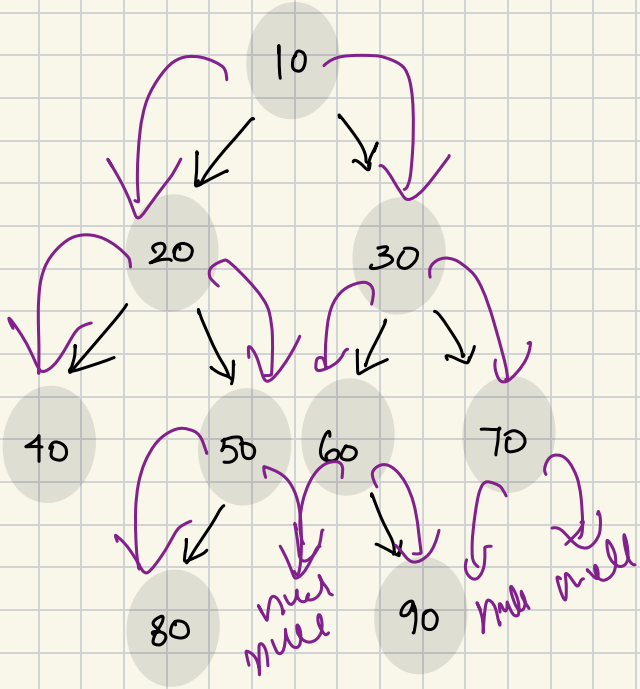
i.e. 0, 1 or 2 childs

Binary Tree

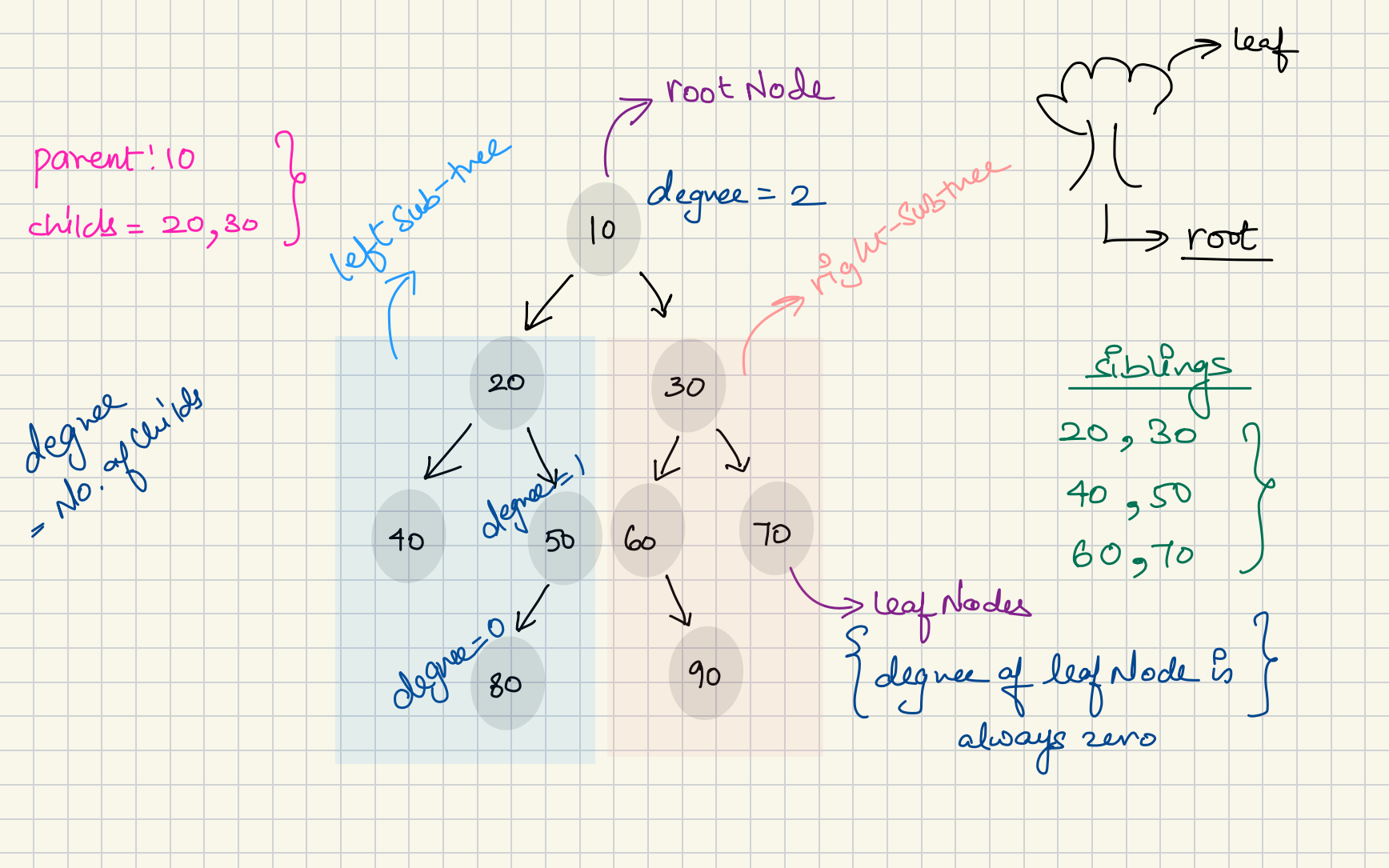
Pls skip for teaching purpose



Binary Tree.

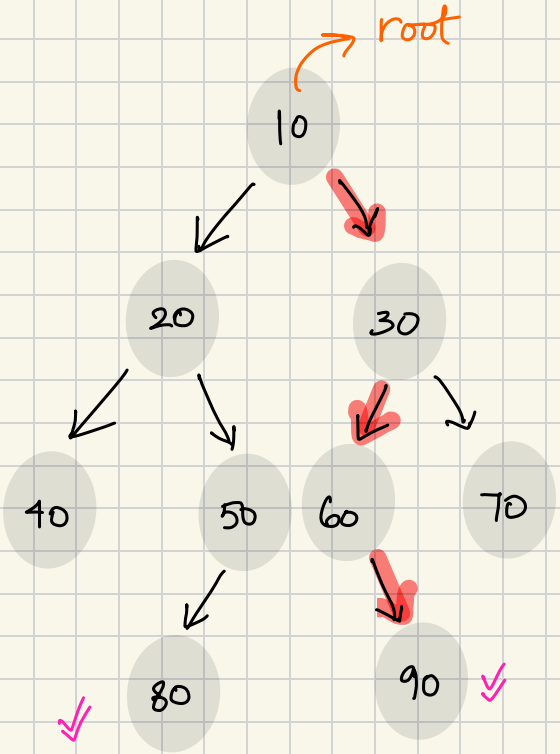



```
class Node
{
    int data;
    Node left;
    Node right;
}
```


parent: 10
childs = 20, 30
root Node
degree = 2
left sub-tree
right-subtree
10
20
30
40
50
60
70
80
90
degree=1
degree=0
leaf
root
degree = No. of childs
Siblings
20, 30
40, 50
60, 70
Leaf Nodes
degree of leaf Node is always zero

Height of a Binary Tree

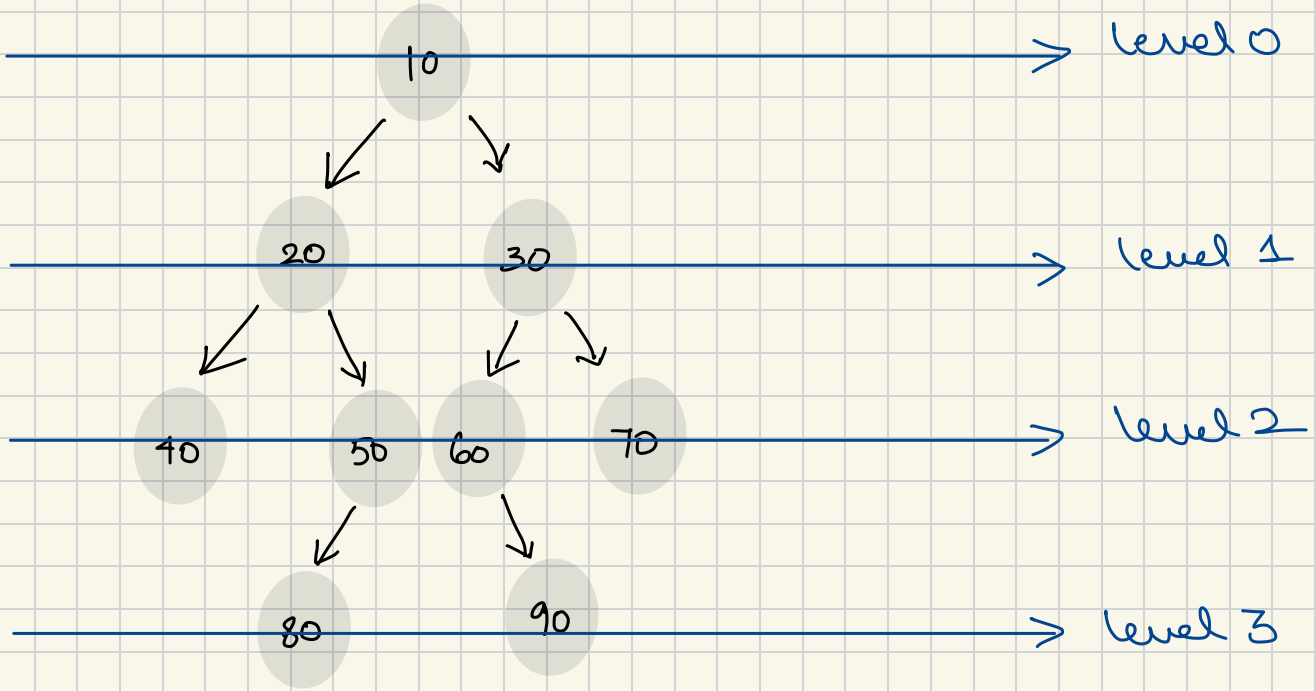
- dist. b/w root Node & deepest leaf Node



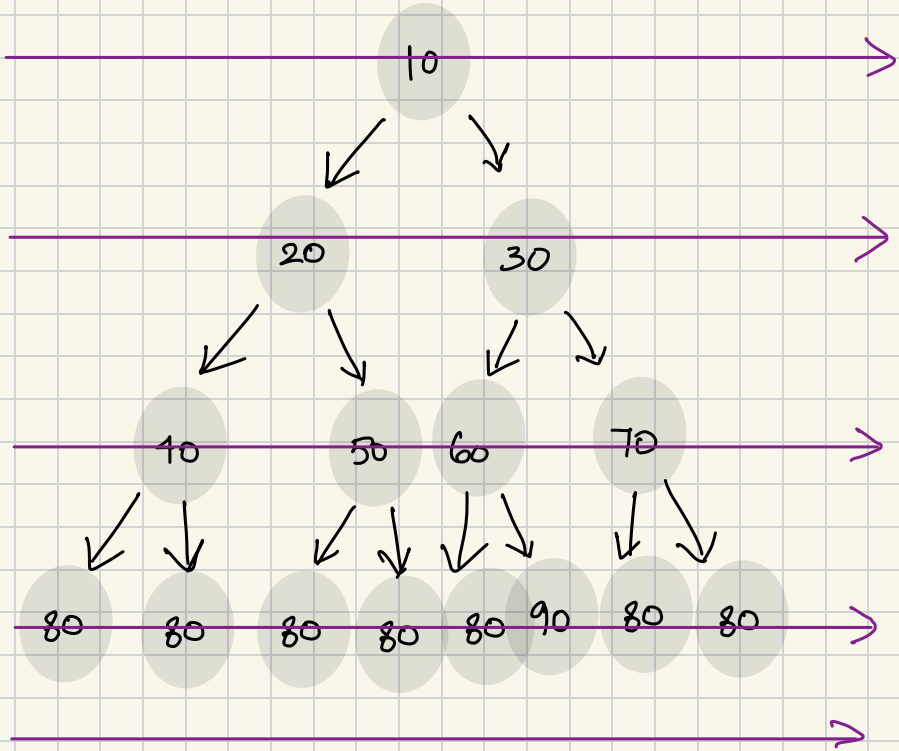
height = 4 { in terms of Nodes }

height = 3 { in terms of Edges }

Levels in a Binary Tree



Perfect Binary Tree - {No. of Nodes in each level (l) = 2^l}



level 0 = 1 → 2^0

level 1 = 2 → 2^1

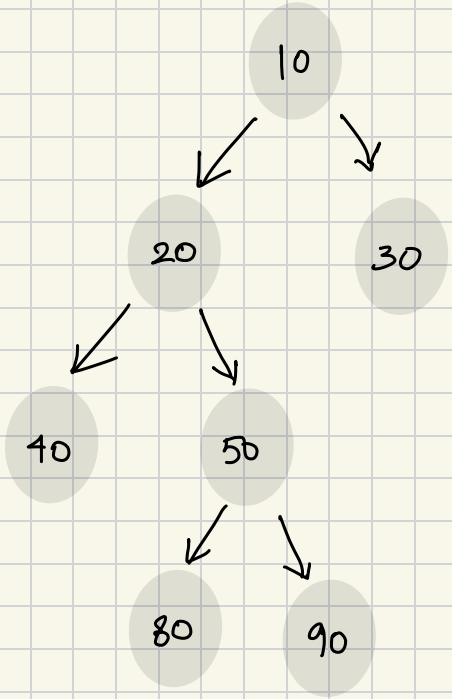
level 2 = 4 → 2^2

level 3 = 8 → 2^3

level k = 2^k (Nodes)

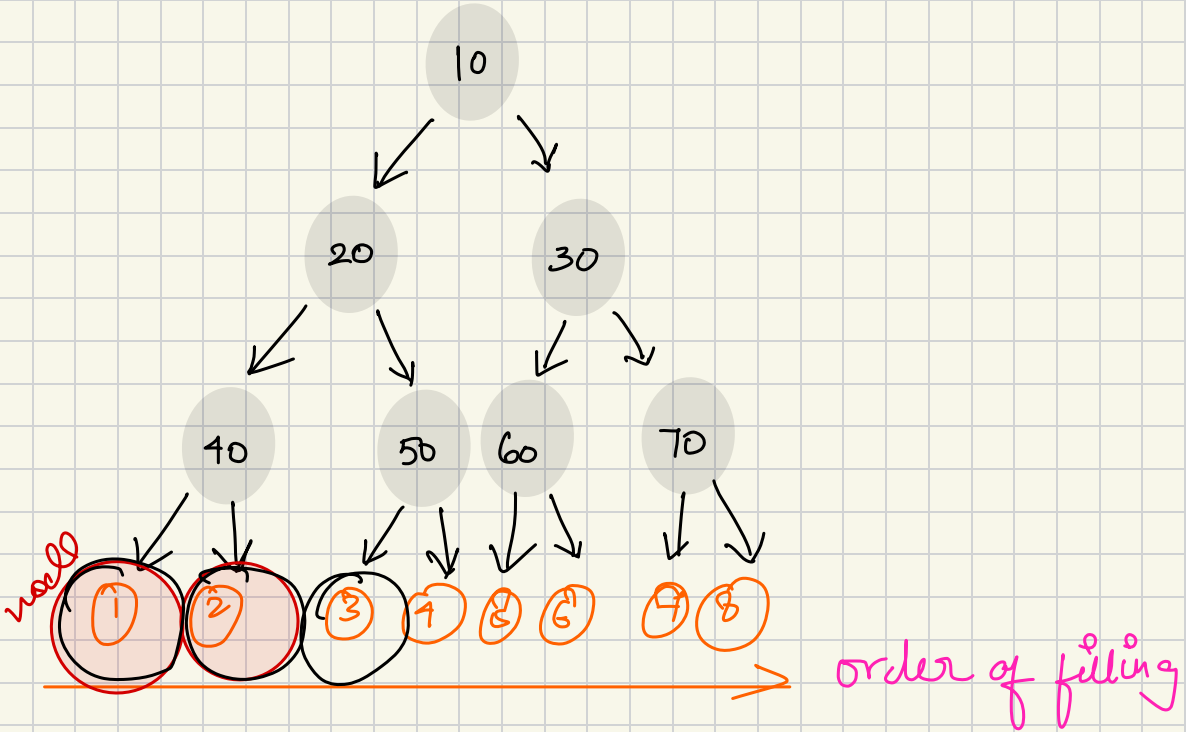
Full Binary Tree

→ Each Node either has 0 or 2 child



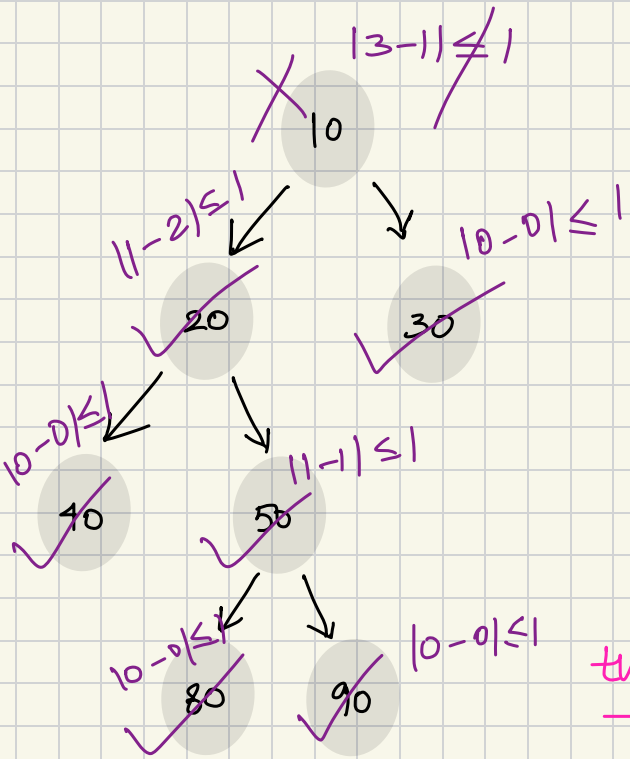
Complete Binary Tree.

Where each level is completely filled, except the last level, where Nodes are as left positioned as possible



Balanced Binary Tree

→ a binary tree where each node is balanced



Balanced Node

abs. diff of LST height and RST height ≤ 1

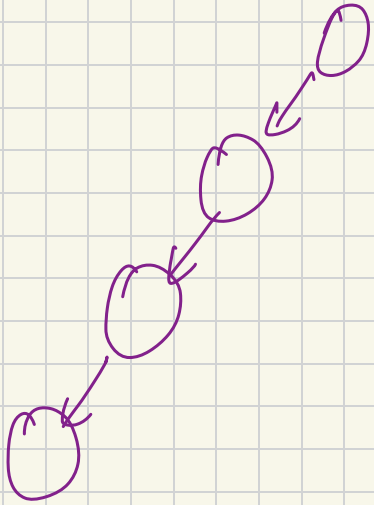
$$|h(LST) - h(RST)| \leq 1$$

this tree is not balanced!

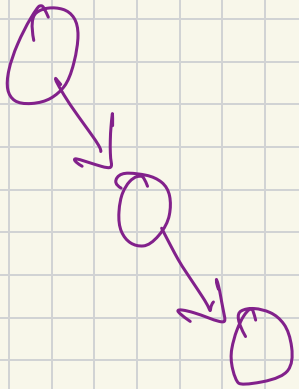
Skew Tree

① Left Skewed tree

Left child, or No child

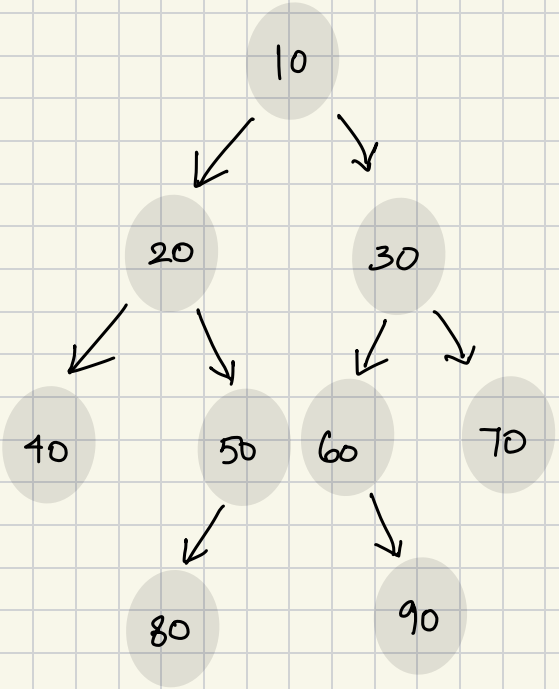


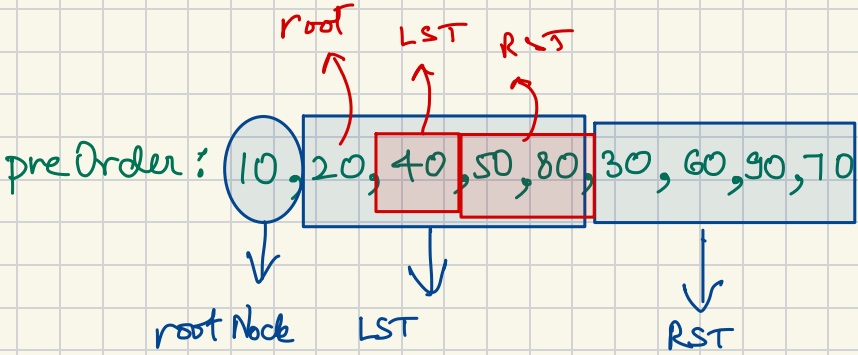
② right skewed tree



Traversal over a tree.

- Pre order traversal





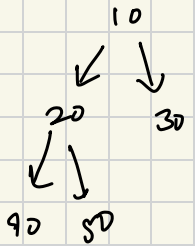
Pre order

- print root
- pre order LST
- pre order RST

faith: print preorder of the tree from the given root.

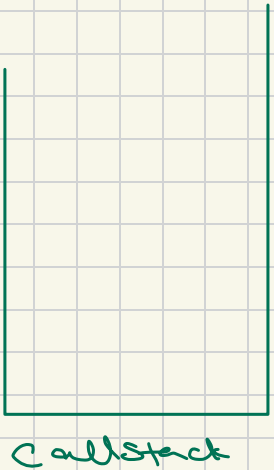
```
Void preOrder(Node root)
{
    if(root == null)
        return;

    print(root)
    preOrder(root.left);
    preOrder(root.right);
}
```

```java
// This function performs a preorder traversal of a binary tree.
// In preorder traversal, we visit the root first, then the left subtree, and finally the right subtree.
public static void preorderTraversal(Node root) {
    // Base case: If the current node is null, we expect nothing to be done.
    // This is our expectation for the smallest or simplest input.
    if (root == null) {
        return;
    }

    // Our faith here is that we can process the current node.
    // So, we do exactly that by printing the node's value. This is the "root" part of Root-Left-Right.
    System.out.print(root.data + " ");

    // By making a recursive call on the left child, we have faith that this call will
    // correctly perform a preorder traversal of the left subtree. We don't need to know
    // how it's done, we just expect that it will be done.
    preorderTraversal(root.left);

    // Similarly, we have faith that a recursive call on the right child will
    // correctly perform a preorder traversal of the right subtree. Again, our expectation
    // is that the traversal will be completed without needing to understand the specifics.
    preorderTraversal(root.right);
}
```



10, 20, 40, 50, 30

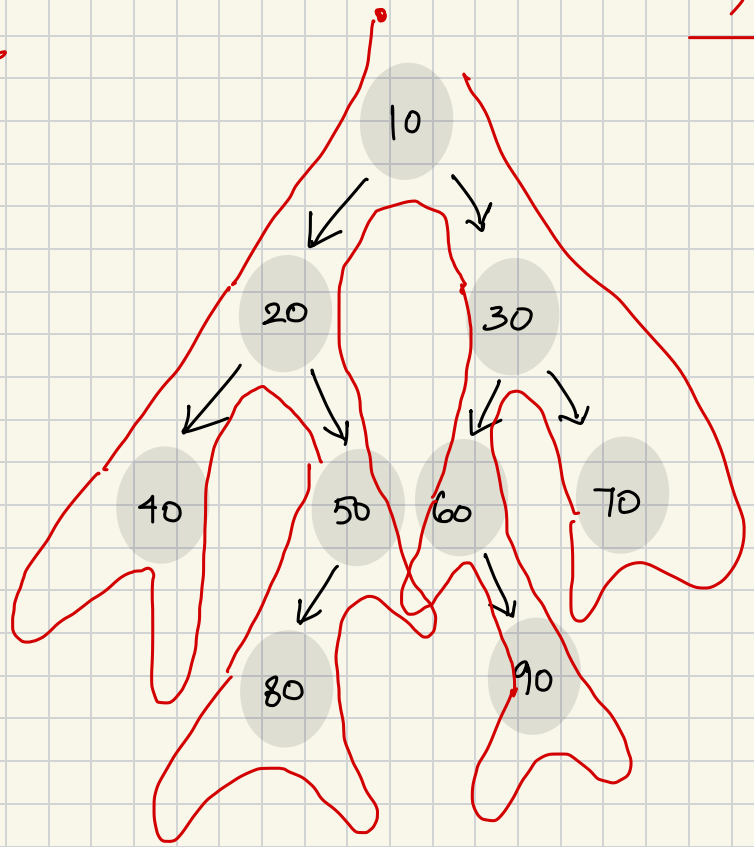
10 20 40 50 30



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

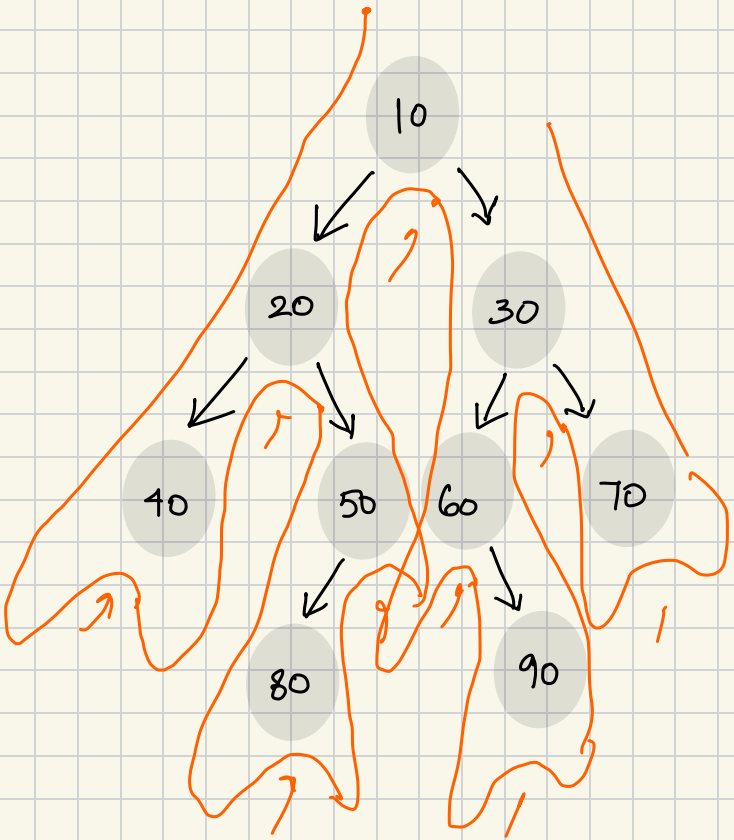
technique to write preorder

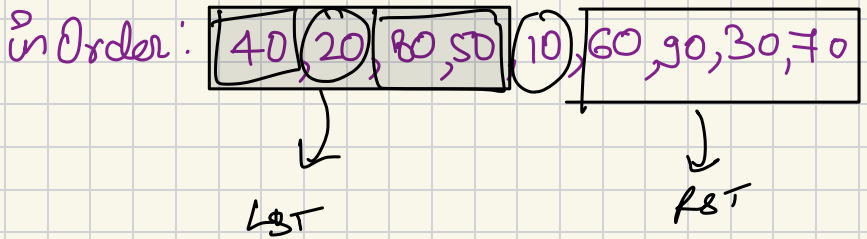
Eular path



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

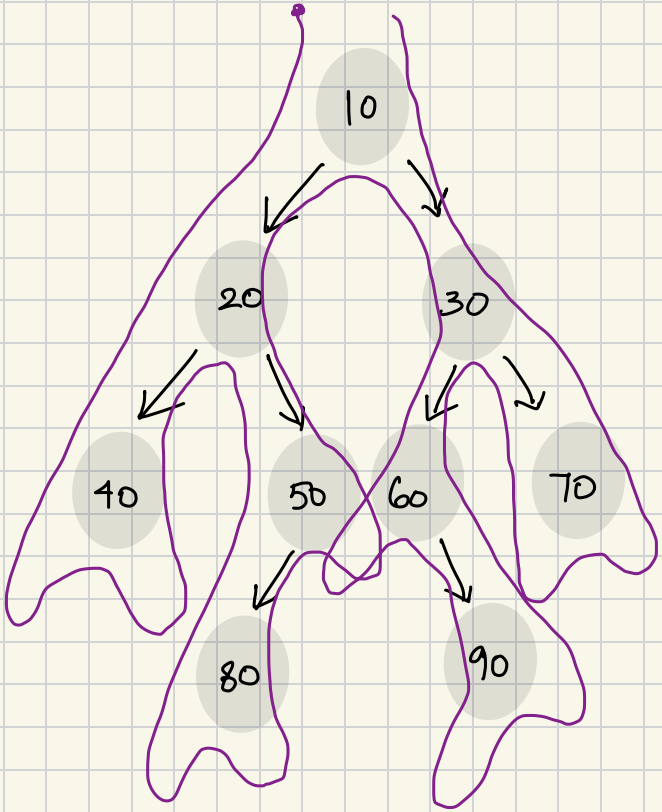
In Order traversal





- Inorder (LST)
- print (root)
- inorder (RST)

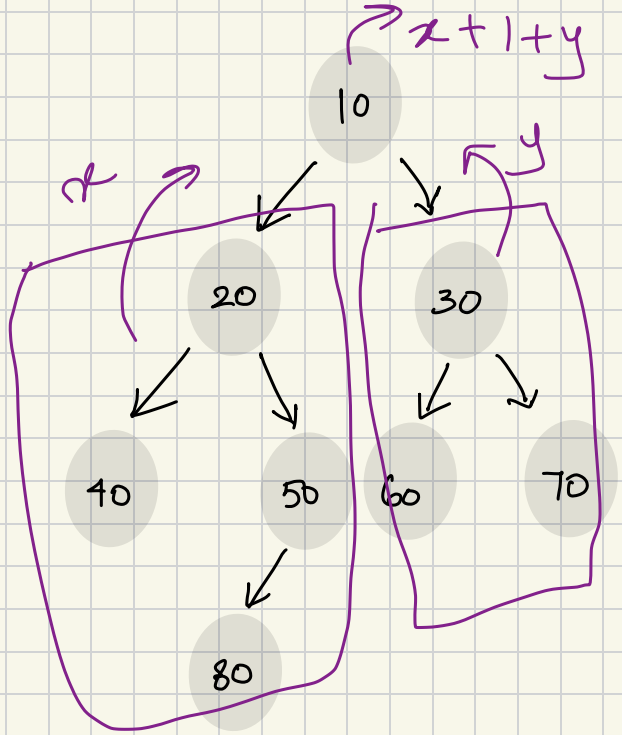
post Order Traversal



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

Size of Binary tree

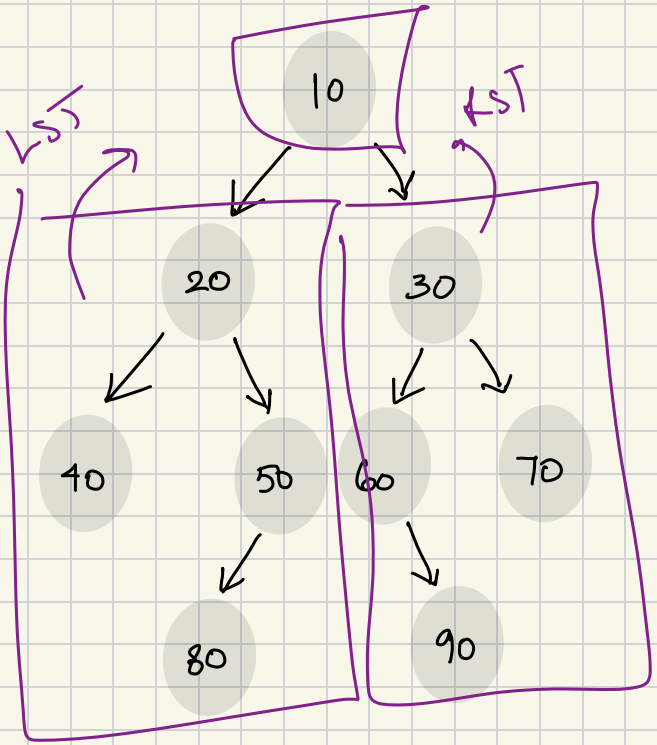
- No. of Nodes in this BT



faith: returns size of BT from root

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

    int x = size(root.left)
    int y = size(root.left)

    return x+1+y;
}
```

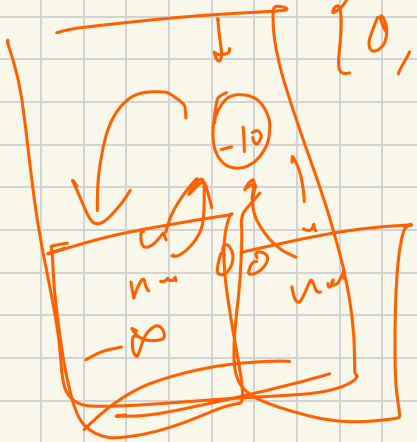
Sum of BT : sum of val of all nodes of BT



sum = LST + RST + root.val

Int sum(Node root)

Maximum in BT



$\{0, -10, 0\} = 0$