Trees

No preq → Stacks & Queue
OOPS

Generic Tree - Lec 1	Introduction, DFS(Preorder & Postorder), Generic Solver
Generic Tree - Lec 2	Mirror, Remove Leaves, Linearize (2 Approaches)
Generic Tree - Lec 3	Max Sum Subtree, Diameter, Iterative DFS
Generic Tree - Lec 4	BFS - Level Order, Linewise (3 Approaches), ZigZag
Generic Tree - Lec 5	Find, N2R Path, LCA - Generic Tree, Distance b/w 2 Nodes
Generic Tree - Lec 6	Trees - Same, Mirror, Symmetric, Travel & Change Strategy
Binary Tree - Lec 1	Introduction, DFS (Recursive & Iterative), BFS - Level Order
Binary Tree - Lec 2	Find, N2R Path, R2L Path in Range, LCA - Binary Tree
Binary Tree - Lec 3	K Levels Down, K Distance Away, Trees - Same, Mirror
Binary Tree - Lec 4	Single Child Nodes, Left Cloned - I, II, Remove Leaves, Tilt
Binary Tree - Lec 5	Is BST, Is Balanced Tree, Diameter (3 Approaches)
BST - Lec 1	Intro, Construction, Generic Solver, Insertion, Removal
BST - Lec 2	Replace with larger sum, LCA - BST, Target Sum Pair

→ DBMS
→ OS
} → Live classes → Weekdays (7:00 - 8:30 PM) 5 classes per week } From next monday

→ System Design (OOPS + LLD + HLD) → weekends (11:00 to 1:00 PM)

Generic Tree / N Ary Tree

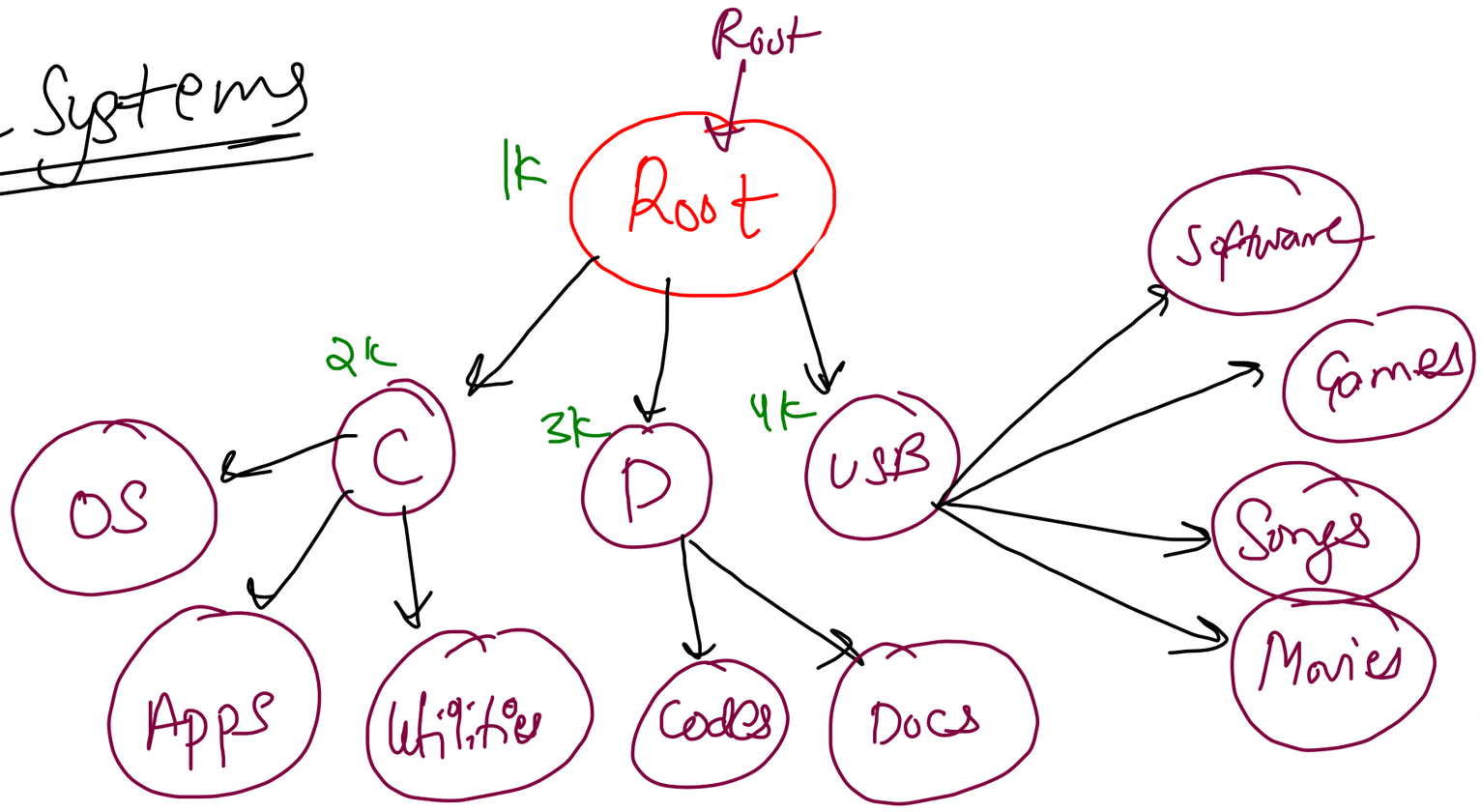
Data Structures

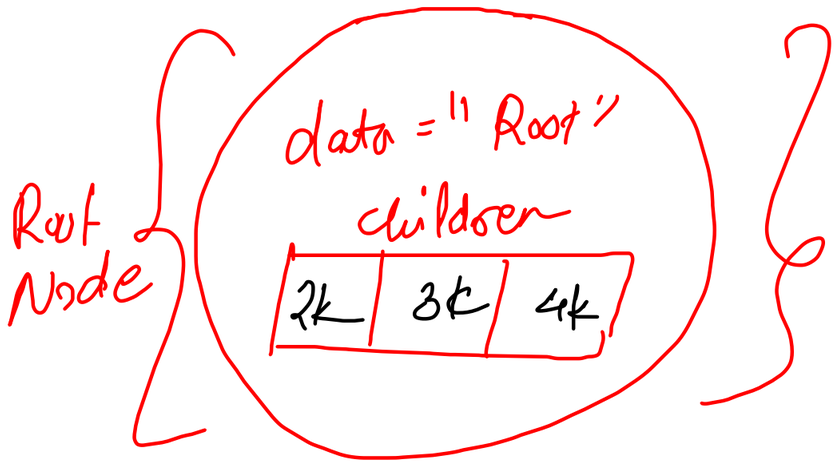
- → Array / String
- → ArrayList / String Builder } Contiguous } Linear
- → LinkedList } Non - contiguous
- → Stack & Queue

- → Generic Tree / N Ary Tree
- → Binary Tree.
- → Binary Search Tree } Non - contiguous & Non - linear

eg Heirarchy

File Systems






```
listNode {
    int data
    listNode next;
}
```

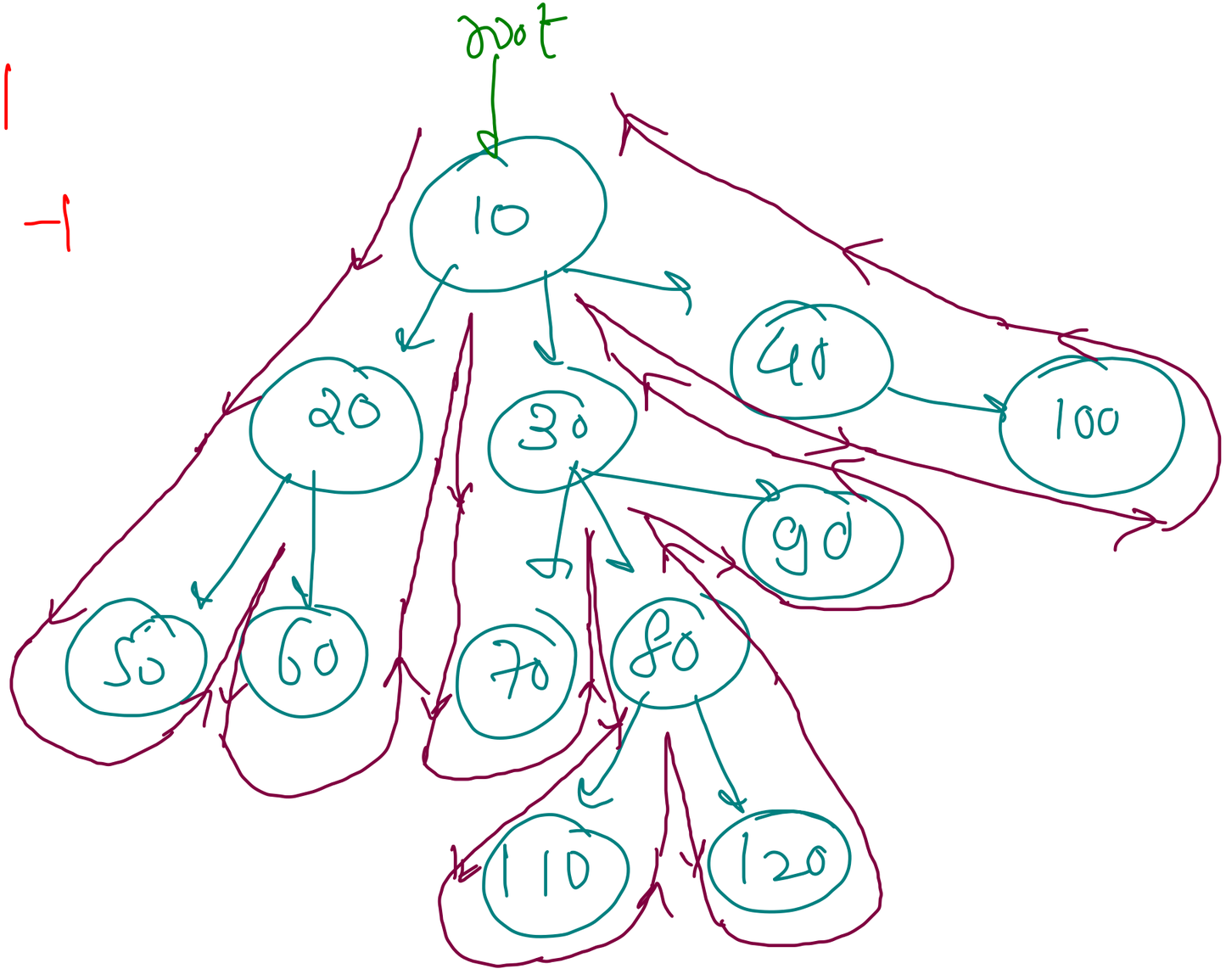
```
TreeNode {
    int data;
    ArrayList<TreeNode> children;
}
```

10 20 50 -1 60 -1 -1

30 70 -1 80 110 -1 120 -1

-1 90 -1 -1 40 100

-1 -1 -1



(-1) → marker
no child of current node

```java
private static class Node {
    int data;
    ArrayList<Node> children;

    Node(){
        this.data = 0;
        this.children = new ArrayList<>();
    }

    Node(int data){
        this.data = data;
        this.children = new ArrayList<>();
    }
}
```

Tree Node class

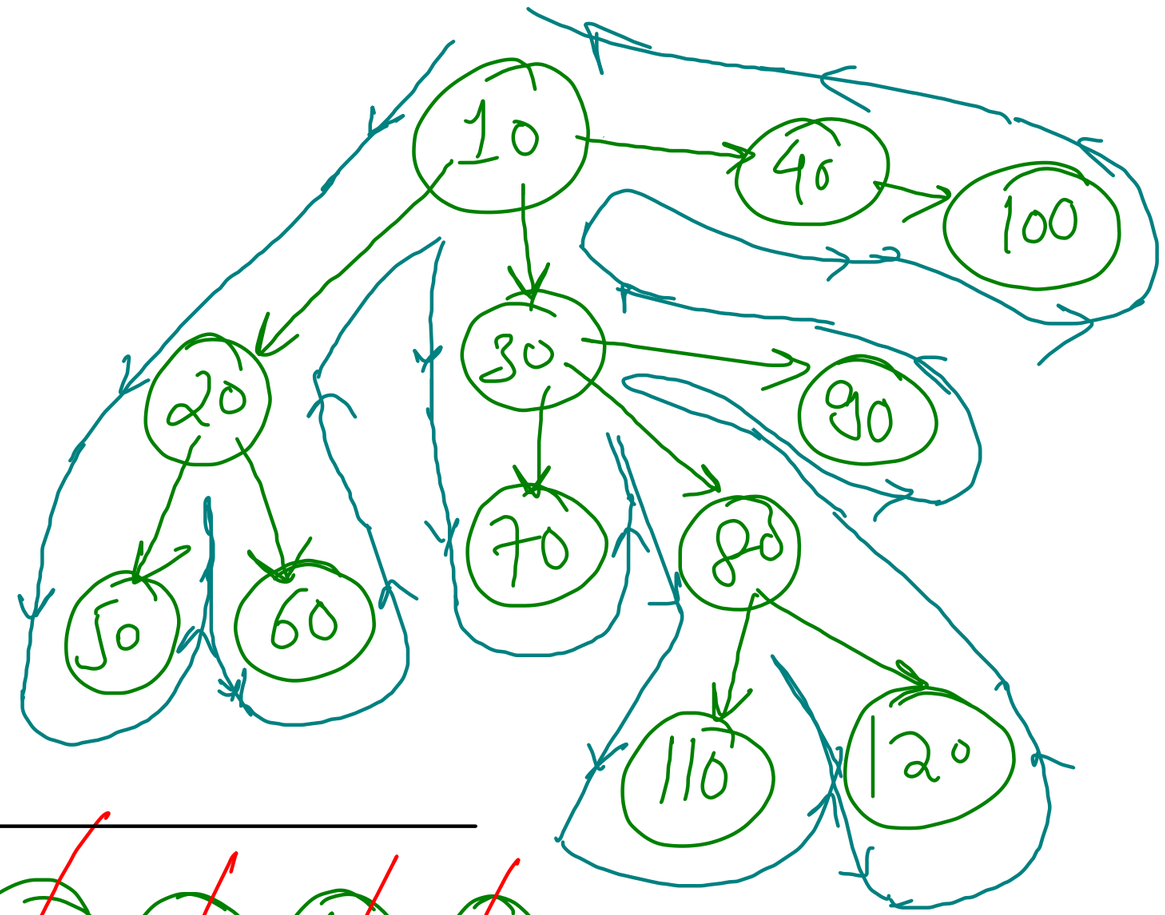
```java
public static void main(String[] args) throws Exception {
  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
  int n = Integer.parseInt(br.readLine());
  int[] arr = new int[n];
  String[] values = br.readLine().split(" ");
  for (int i = 0; i < n; i++) {
    arr[i] = Integer.parseInt(values[i]);
  }

  Node root = construct(arr);
  display(root);
}
```

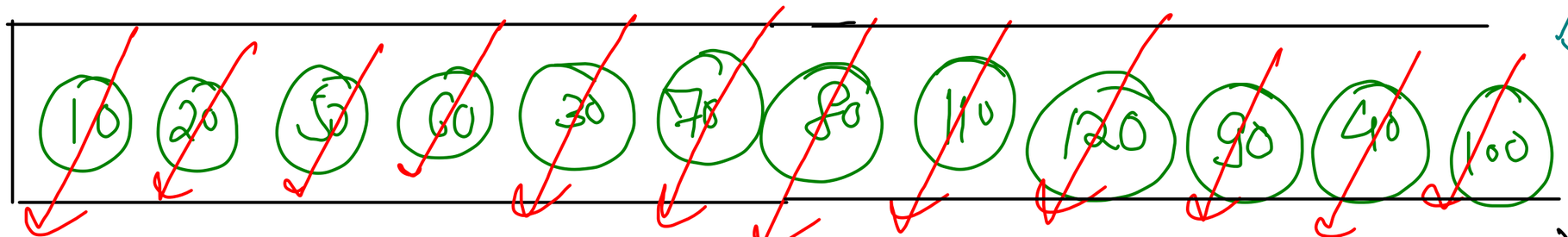
10 20 50 -1 60 -1 -1

30 70 -1 80 110 -1 120 -1

-1 90 -1 -1 40 100

-1 -1 -1



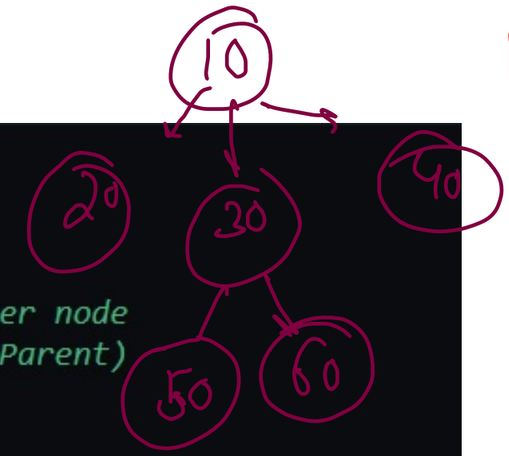


0 / -1 ⇒ node ⇒ (i) creation (ii) stk.top(parent) → child add
else root node

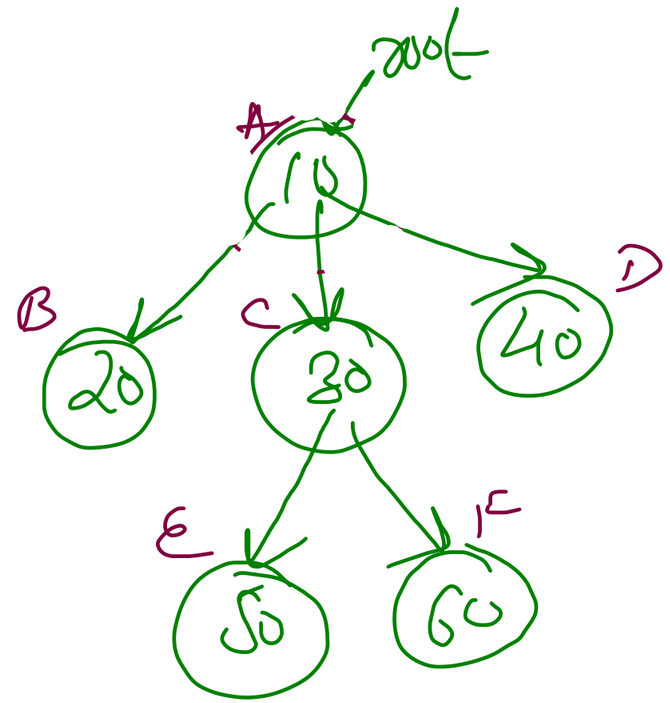
(iii) preorder ⇒ stk.push(node)

==-1 ⇒ node(stk.pop()) ⇒ postorder

```java
public static Node construct(int[] arr) {
    Node root;
    Stack<Node> stk = new Stack<>();
    for(int i=0; i<arr.length; i++){
        if(arr[i] == -1){ // end of child marker node
            stk.pop(); // Postorder (Child -> Parent)
        } else {
            // 1. Creation of Node
            Node curr = new Node(arr[i]);

            if(stk.size() == 0){
                // Current Node is the root Node (Root has no parent)
                root = curr;
            } else {
                // 2. Make Current node as child of parent (stk.top())
                stk.peek().children.add(curr);
            }

            // 3. Preorder (Push curr node in stack)
            stk.push(curr);
        }
    }
    return root;
}
```

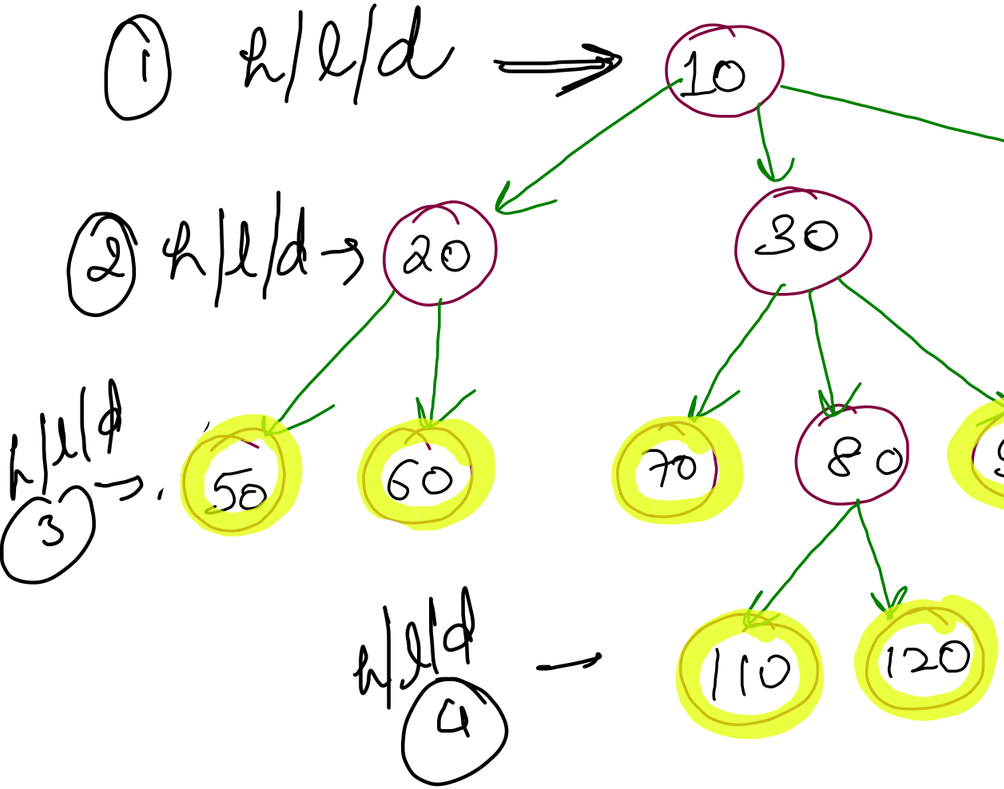


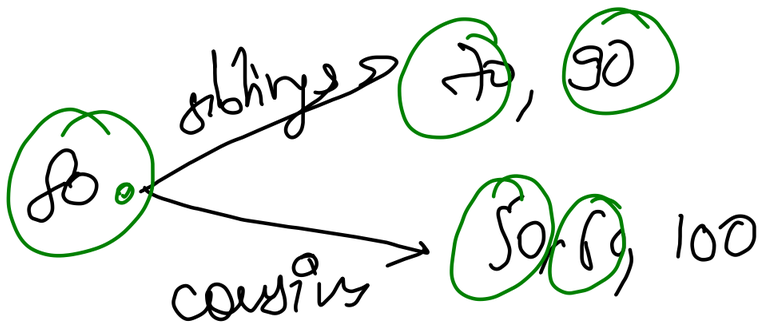
10 20 -1 30 50 -1
60 -1 -1 40 -1 -1



A
data = 10
children
= B C D

C
data = 30
children
E F





→ Root Node (10)

→ Parent, children (10 ⇄ 20, 30, 40)

→ Ancestors

→ Descendants

→ Subtree

→ Cousins

→ Siblings

→ Leaf node

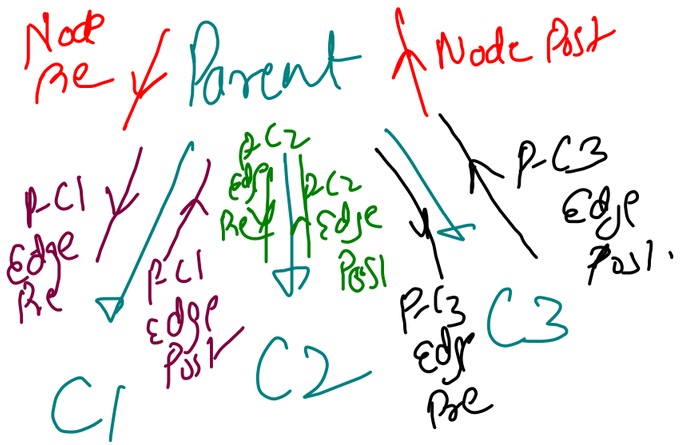
→ Single child node

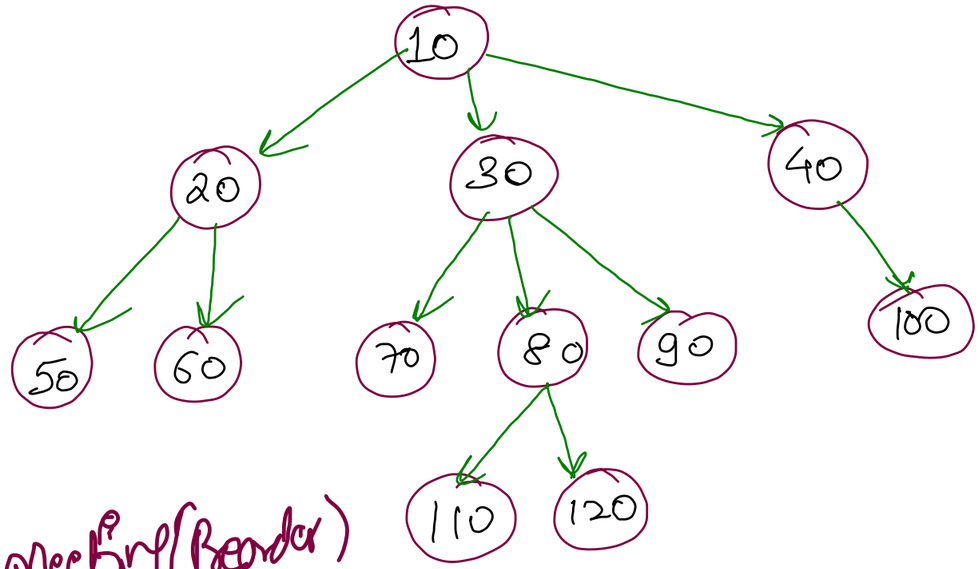
→ level

→ depth

→ height

→ Degree





→ Preorder → Node, Edge

→ Postorder → Node, Edge

Meeting (Boundary)

Emp

10 → 20, 30, 40, .

20 → 50, 60, .

50 → .

60 → .

30 → 70, 80, 90, .

70 → .

80 → 110, 120, .

110 → .

120 → .

90 → .

40 → 100, .

100 → .

```java
public static void display(Node node) {
    if(node == null){
        // Tree is not present (No nodes)
        return;
    }

    // Node Preorder (Meeting Expectation)
    System.out.print(node.data + " -> ");
    for(Node children: node.children){
        System.out.print(children.data + ", ");
    }
    System.out.println(".");

    for(Node children: node.children){
        // Edge Preorder
        display(children);
        // Edge Postorder
    }

    // Node Postorder
}
```

1 ← Edges

1 ← Edges

Depth First Traversal
(DFS)

↳ Tree → Postorder
↓ ↓
Preorder Inorder

N nodes in tree

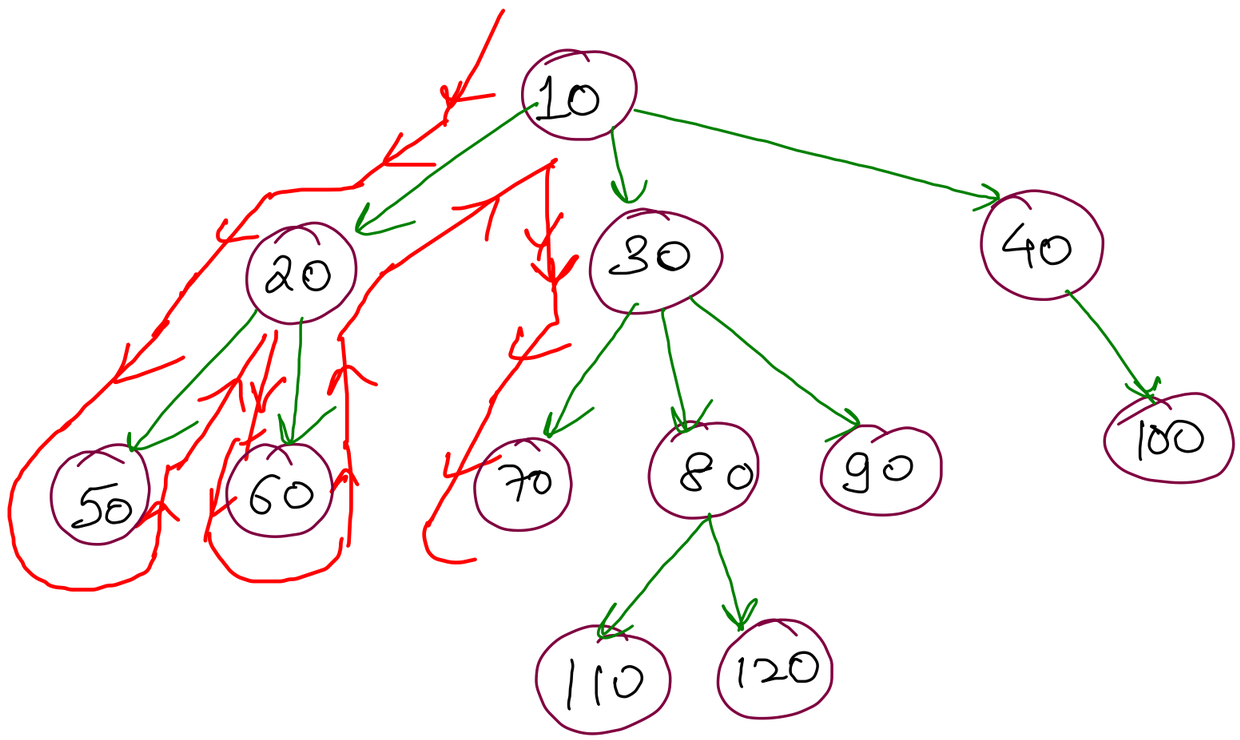
$E = N-1$ edges

O(N) → time complexity

O(N + E)
↓ ↓
no of nodes + no of edges

space complexity

O(H) { H = Height of tree }
↳ O(N) worst case { Skewed tree }
↳ O(log N) average case { balanced tree }




```
Node Pre 10
Edge Pre 10-20
Node Pre 20
Edge Pre 20-50
Node Pre 50
Node Post 50
Edge Post 20-50
Edge Pre 20-60
Node Pre 60
Node Post 60
Edge Post 20-60
Edge Post 10-20
```

```
Edge Pre 10-30
Node Pre 30
Edge Pre 30-70
Node Pre 70
```

```
:
:
```

```java
public static void traversals(Node node){
    System.out.println("Node Pre " + node.data);
    for(Node child: node.children){
        System.out.println("Edge Pre " + node.data + "--" + child.data);
        traversals(child);
        System.out.println("Edge Post " + node.data + "--" + child.data);
    }
    System.out.println("Node Post " + node.data);
}
```

N E(N-1) E(N-1) N

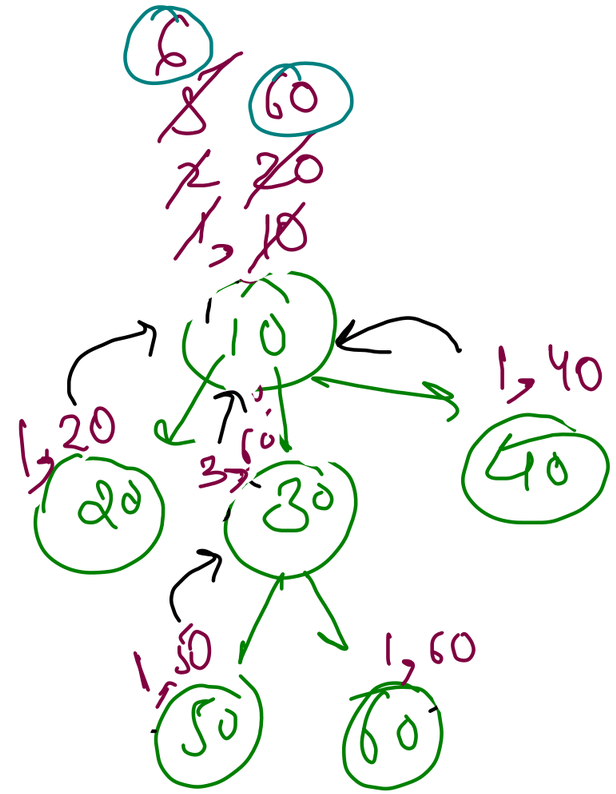
$$O(N + E + E + N) \simeq O(N)$$

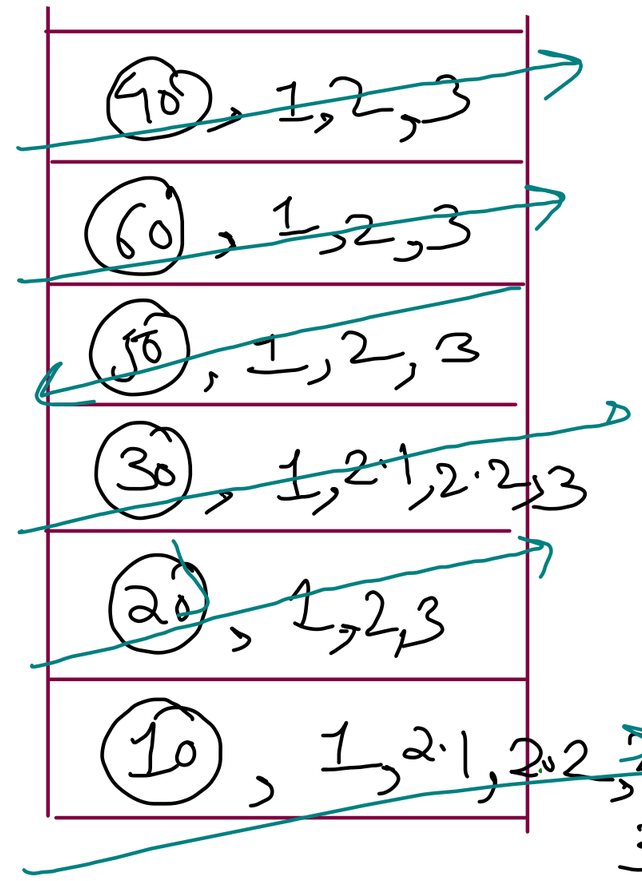
```java
public static int size(Node node){
    int count = 1;
    for(Node children: node.children){
        count += size(children);
    }
    return count;
}
```

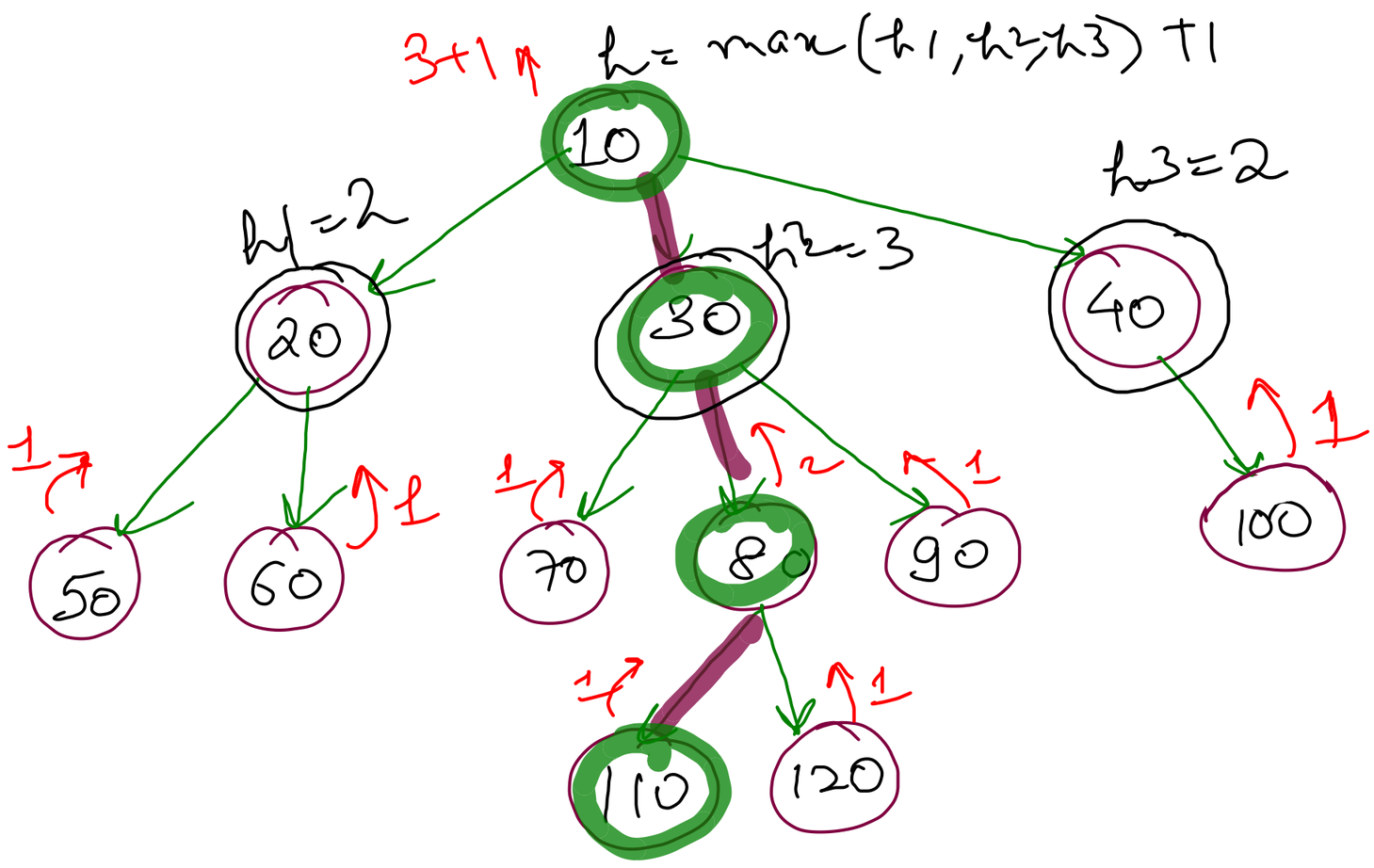
```java
public static int max(Node node) {
    int max = node.data;
    for(Node child: node.children){
        max = Math.max(max, max(child));
    }
    return max;
}
```

O(N) Time

O(H) Recursion call stack space








```java
public int maxDepth(Node root) {
    if(root == null) return 0;
    int height = 0;
    for(Node child: root.children){
        height = Math.max(height, maxDepth(child));
    }
    return 1 + height;
}
```

O(N) Time

O(H) Recursion call stack

Space

Height (Node) = Height (Edge) + 1

Note if tree = null height (Edge) = height (node) = 0

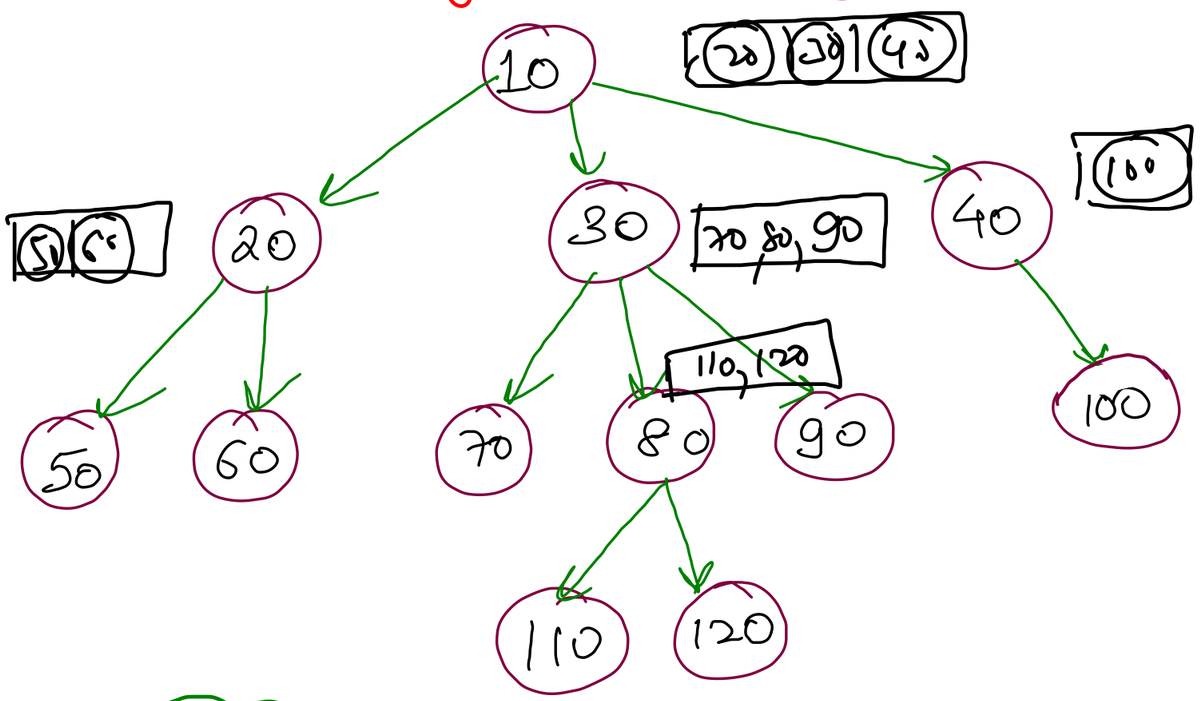
height on tree

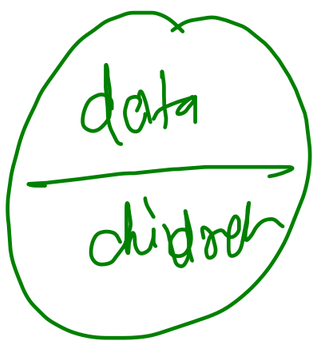
Root Node ↔ Deepest leaf Distance

on the basis of edge (3)

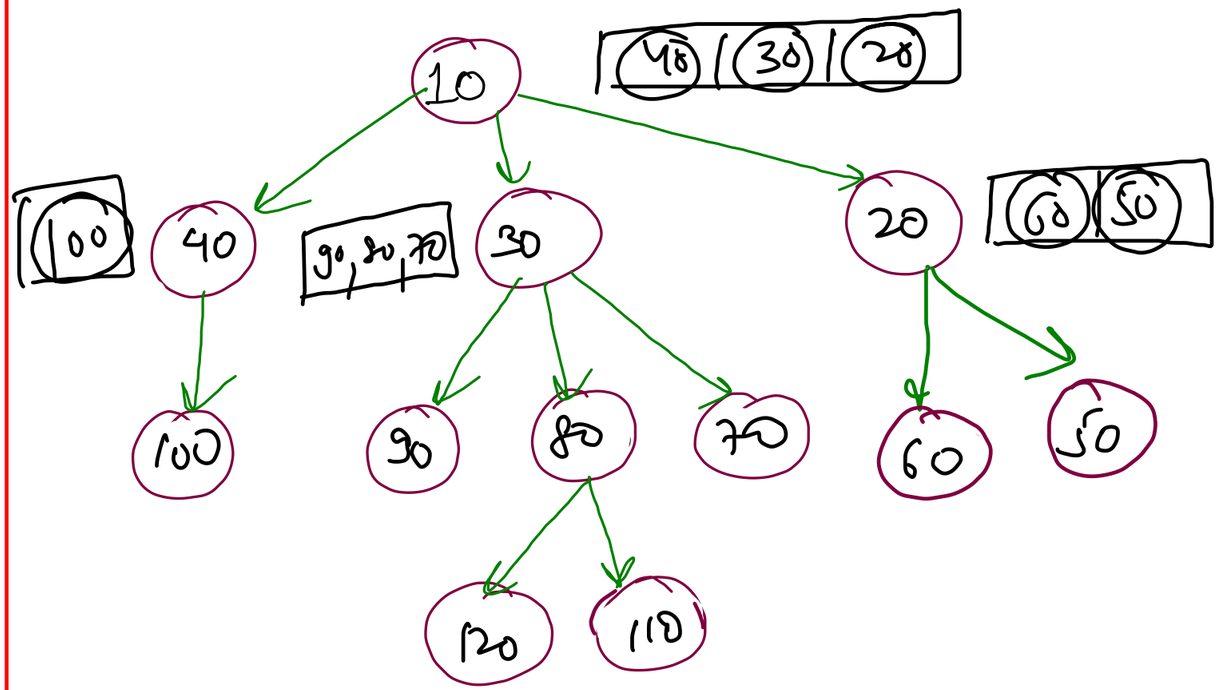
on the basis of node (4)

Original NAry Tree





Mirror NAry Tree



```java
public static void reverse(Node node){
    int i = 0, j = node.children.size() - 1;
    while(i < j){
        Node left = node.children.get(i);
        Node right = node.children.get(j);
        node.children.set(i, right);
        node.children.set(j, left);
        i++; j--;
    }
}

public static void mirror(Node node){
    if(node == null) return;

    reverse(node);
    for(Node child: node.children){
        mirror(child);
    }
}
```

DFS → for arrayList reversal

$O(N * k)$ Time Complexity

k Any Tree ⇒ k children for each node

$O(H)$ Recursion call stack {DFS}

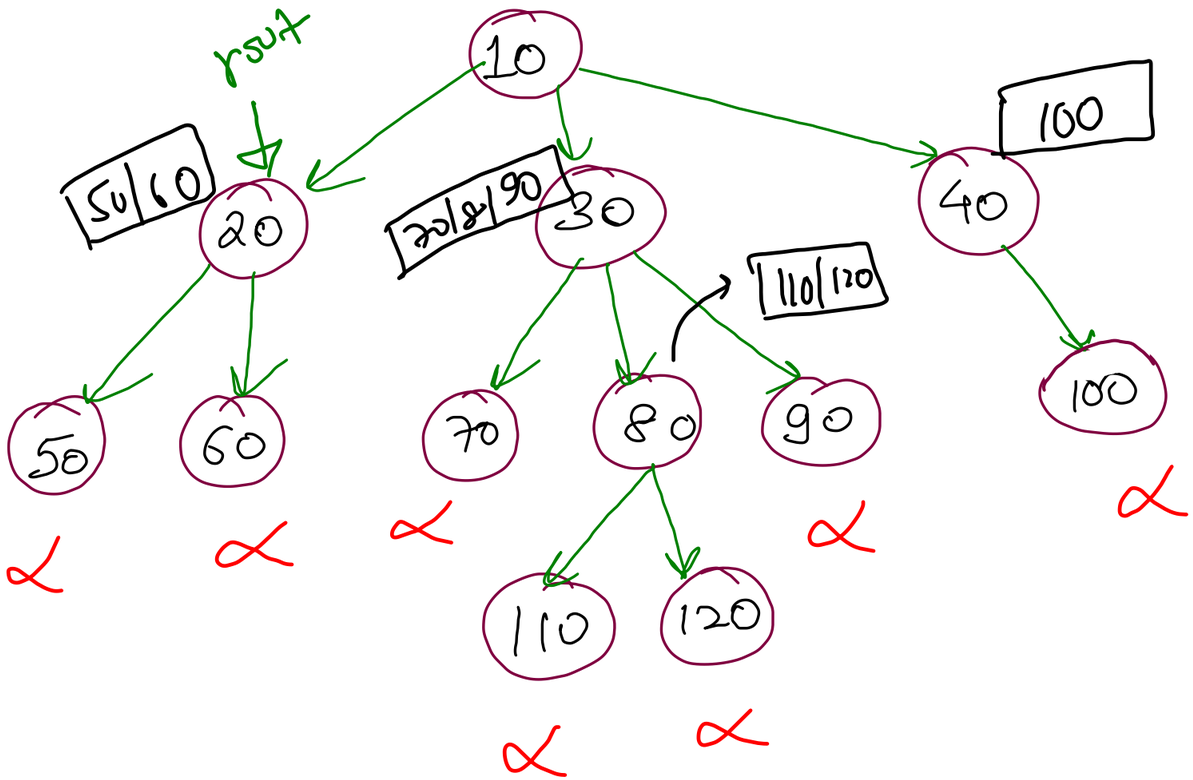
input space { Tree Data Structure }

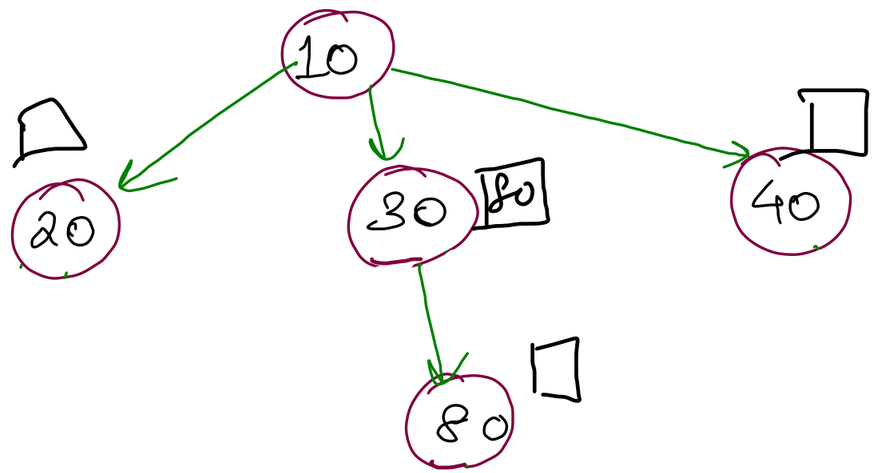
→ $O(N+E)$

↑ data & Nnodes ↑ arrayList size

We can reverse the arrayList in any area of node-pre, node-post

Remove leaf Nodes

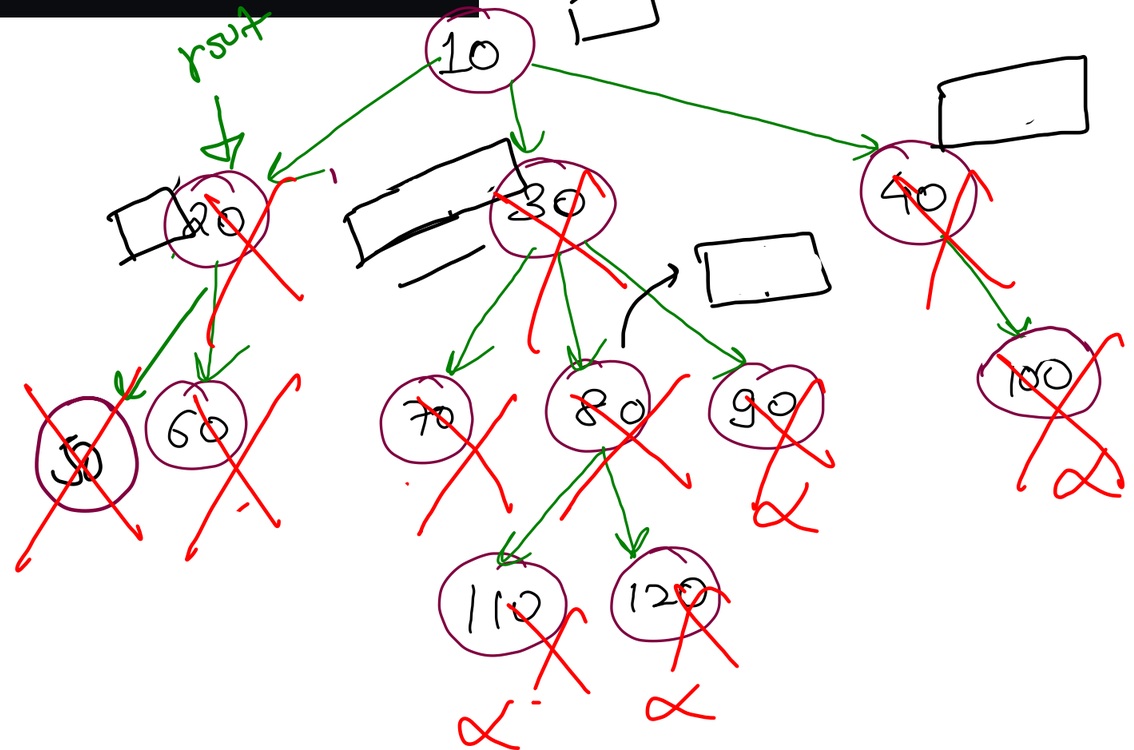




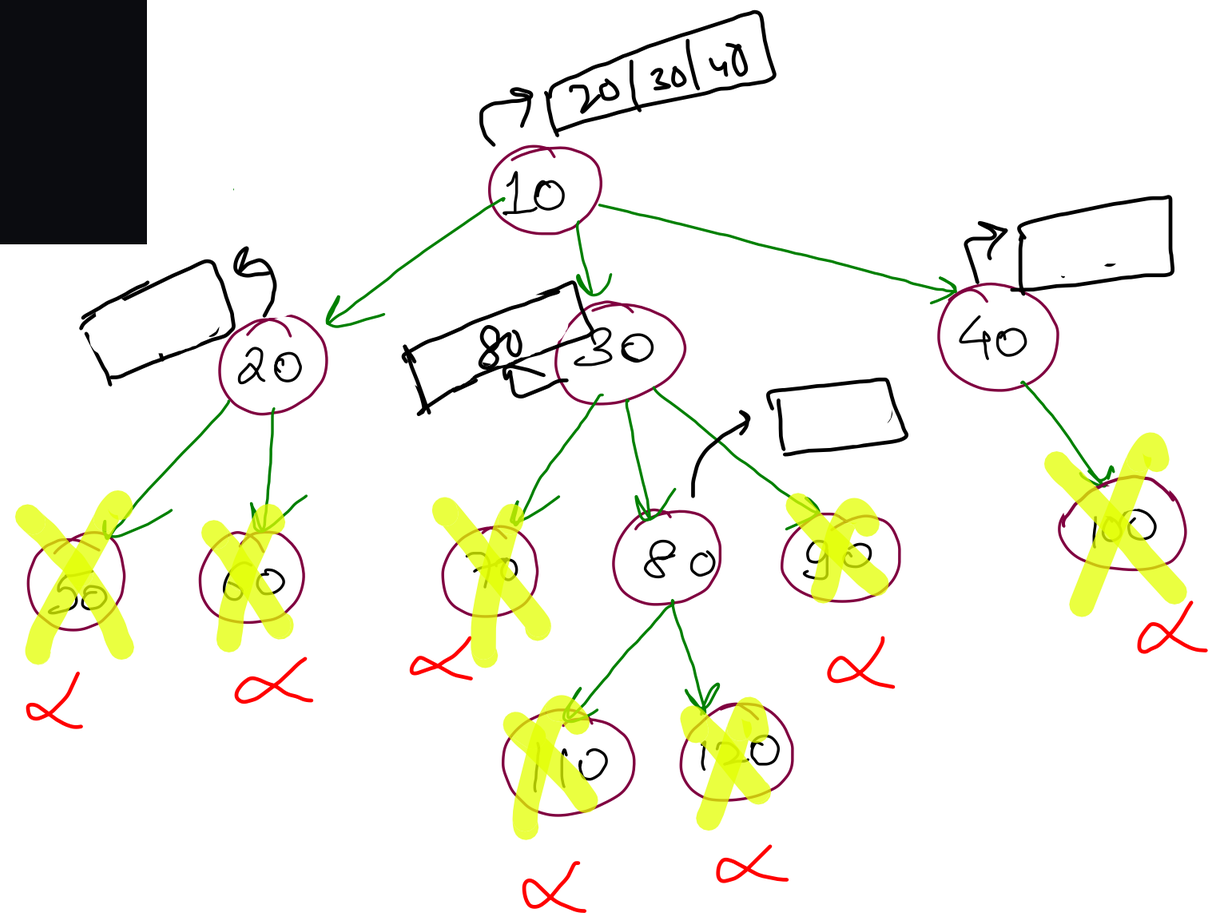
```java
public static void removeLeaves(Node node) {
    if(node == null) return;

    for(Node child: node.children){
        removeLeaves(child);
    }

    // We are deleting leaf child nodes in postorder
    for(int i=node.children.size()-1; i>=0; i--){
        Node child = node.children.get(i);

        if(child.children.size() == 0){
            // My Child Node is a Leaf Node
            node.children.remove(i);
            // Remove that Child Node from my Children ArrayList
            // We have removed the edge linking between the leaf node and it's parent
        }
    }
}
```

" After removal of leaf, non-leaf node also become leaf ".

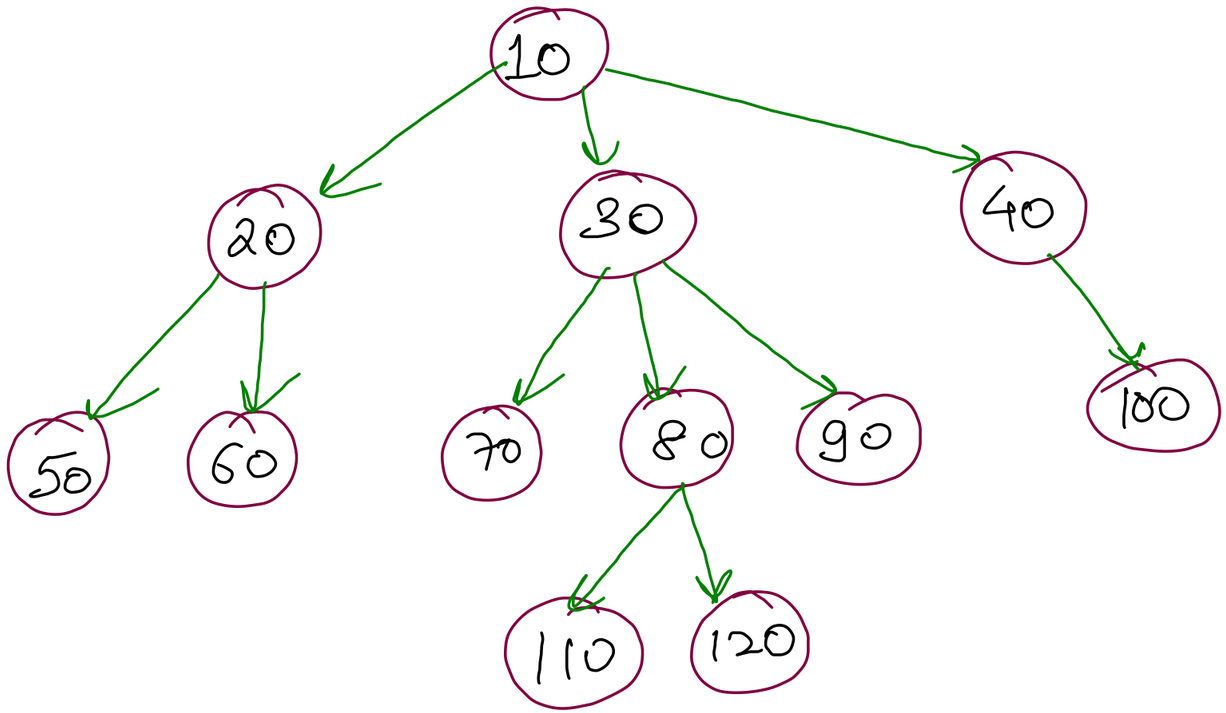
10 not-node is remaining only

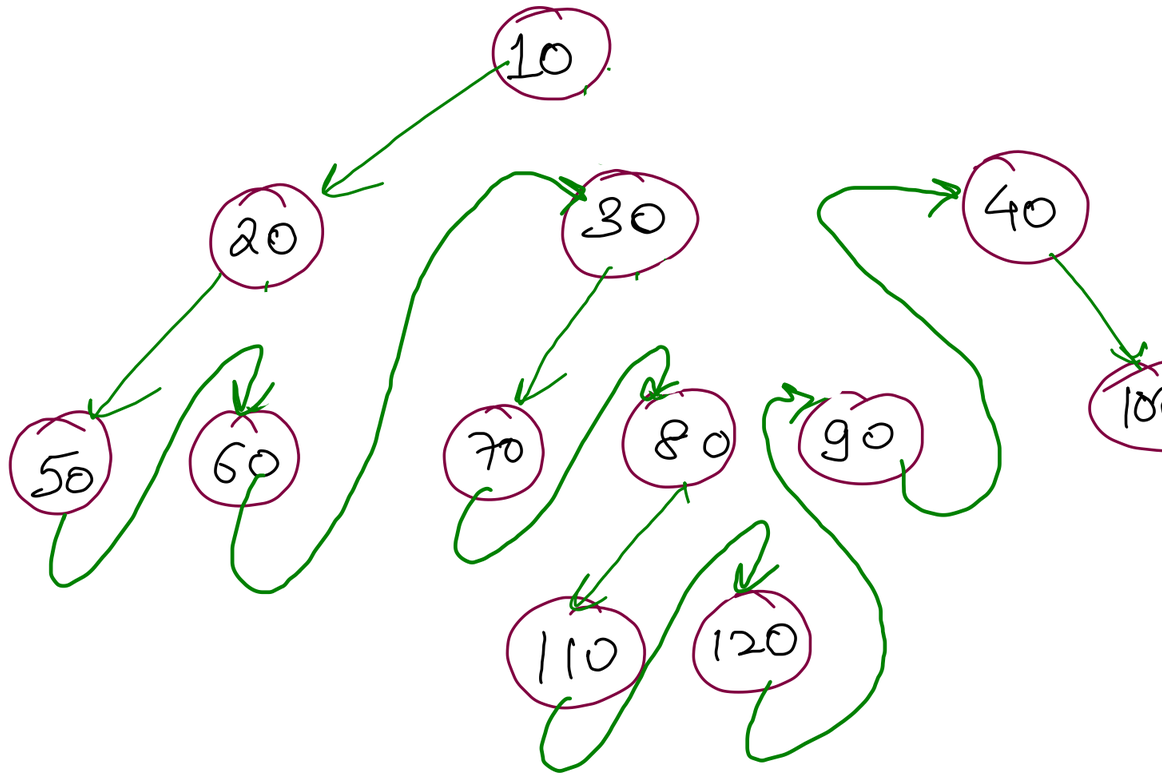


```java
public static void removeLeaves(Node node) {
    if(node == null) return;

    // We are deleting leaf child nodes in preorder
    for(int i=node.children.size()-1; i>=0; i--){
        Node child = node.children.get(i);

        if(child.children.size() == 0){
            // My Child Node is a Leaf Node
            node.children.remove(i);
            // Remove that Child Node from my Children ArrayList
            // We have removed the edge linking between the leaf node and it's parent
        }
    }

    for(Node child: node.children){
        removeLeaves(child);
    }
}
```

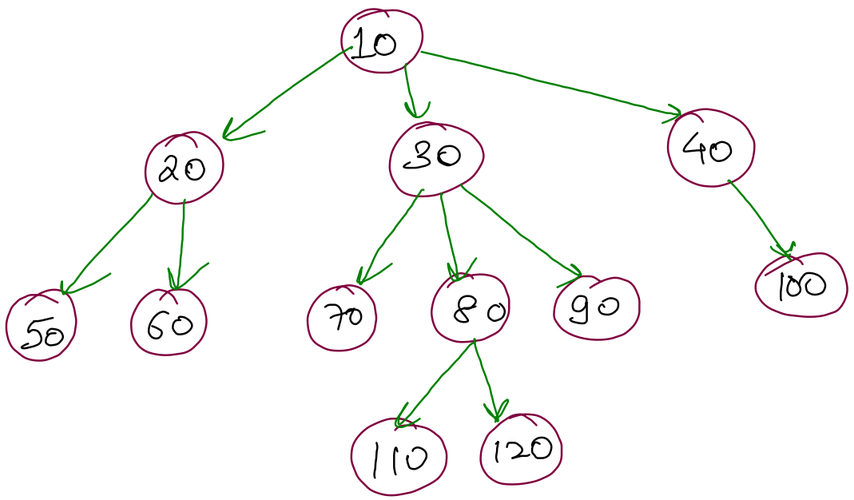


Linearize Generic Tree

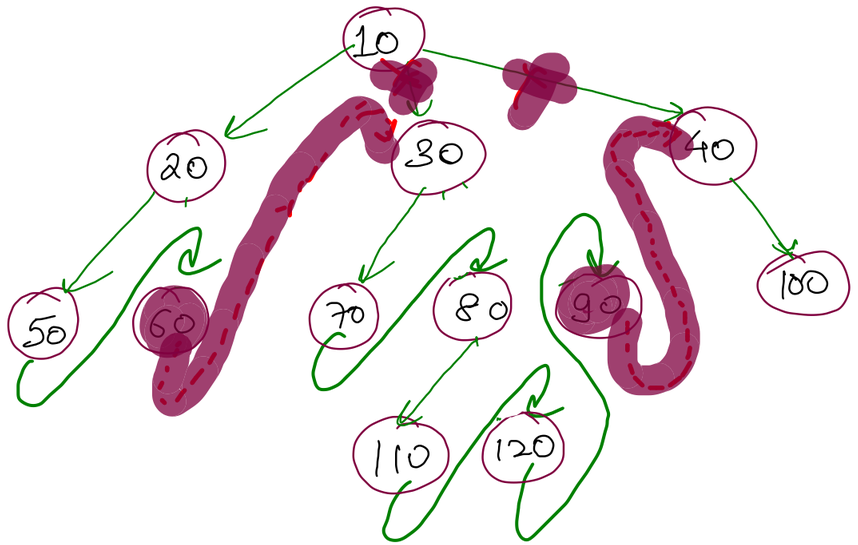




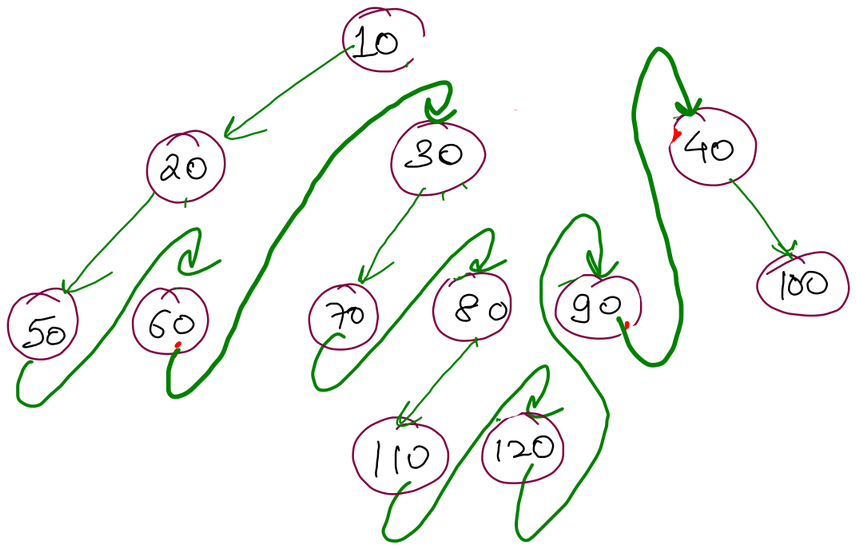
Preorder linearize

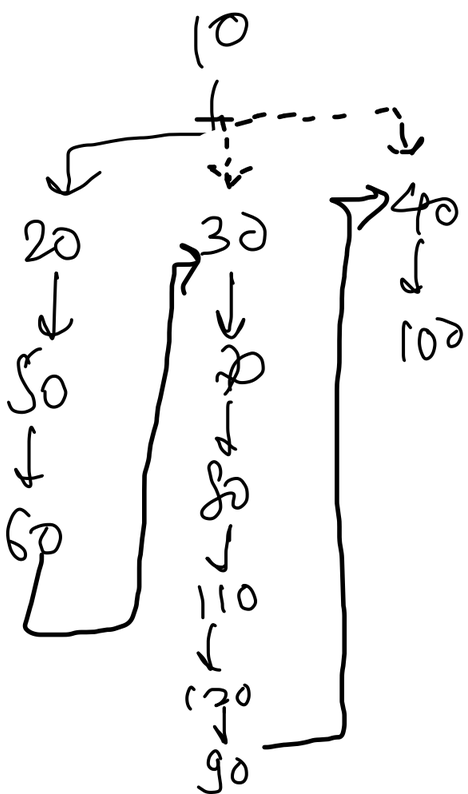


forth call

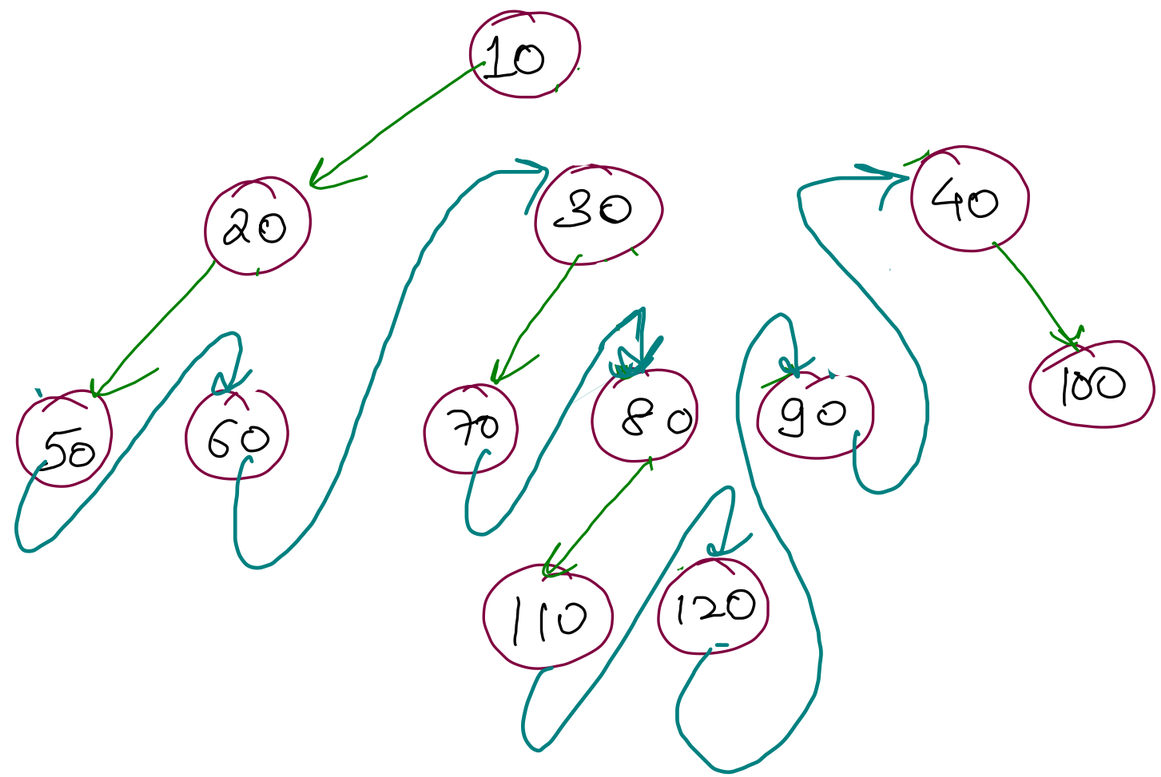


Meeting (Postorder) Expectation





```java
public static Node getTail(Node node){
    while(node != null && node.children.size() > 0){
        node = node.children.get(0);
    }
    return node;
}
public static void linearize(Node node){
    if(node == null) return;

    for(Node child: node.children){
        linearize(child);
    }

    for(int i=node.children.size()-1; i>0; i--){
        Node rightChild = node.children.get(i);
        Node leftChildTail = getTail(node.children.get(i - 1));
        leftChildTail.children.add(rightChild);
        node.children.remove(i);
    }
}
```



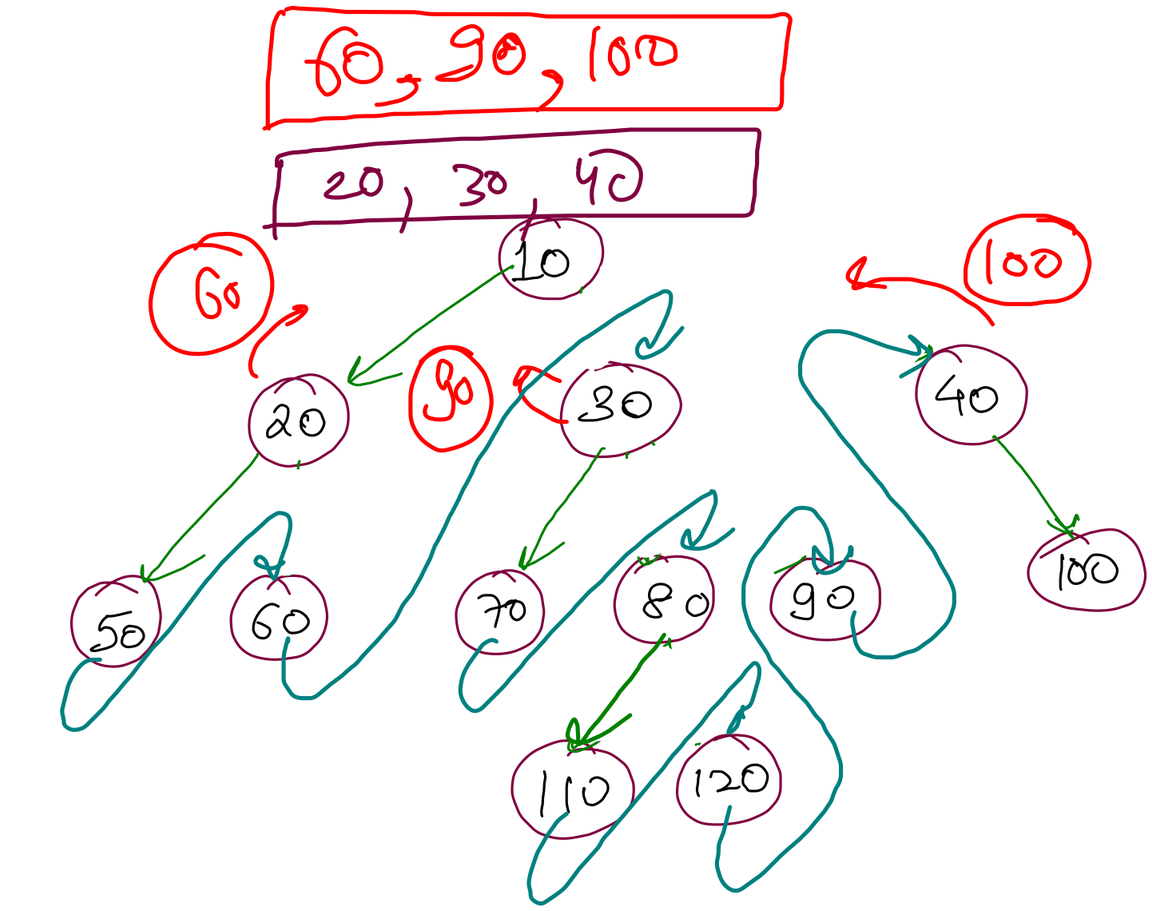
$O(N*N)$ Time Complexity

DFS

gettail

```java
public static Node linearize(Node node){
    if(node == null) return null;

    ArrayList<Node> tails = new ArrayList<>();
    for(Node child: node.children){
        tails.add(linearize(child));
    }

    for(int i=node.children.size()-1; i>0; i--){
        Node rightChild = node.children.get(i);
        Node leftChildTail = tails.get(i - 1);
        leftChildTail.children.add(rightChild);
        node.children.remove(i);
    }

    if(tails.size() == 0) return node;
    return tails.get(tails.size() - 1);
}
```

for leaf node



O(N) Time Comp

↑

DFS