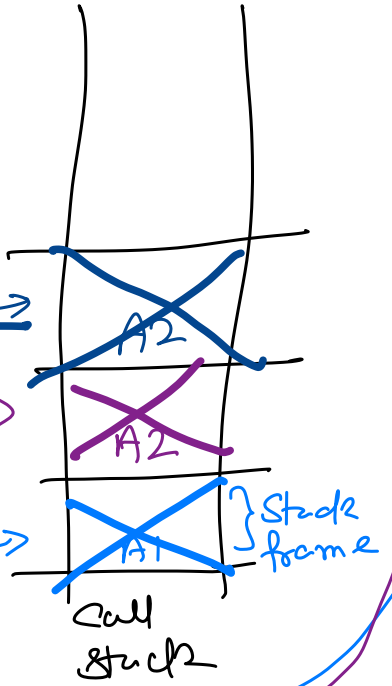
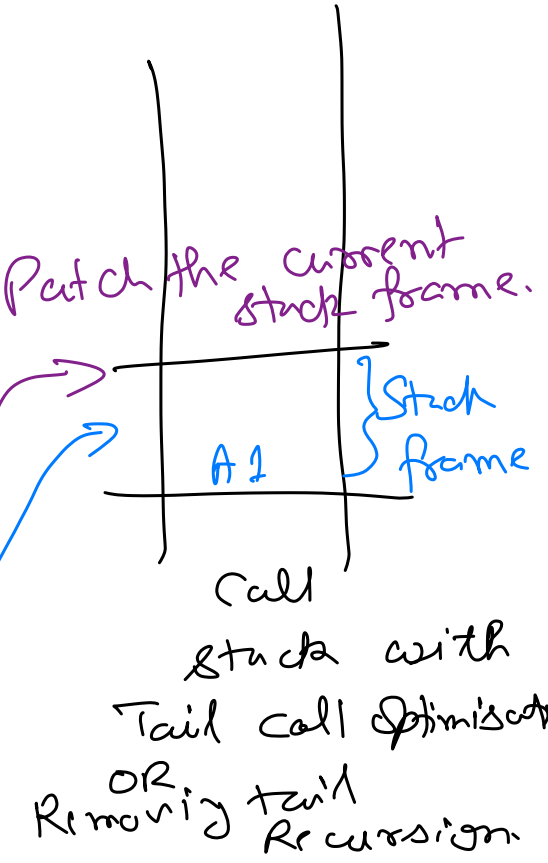
Tail Recursion / Tail Call

Last statement in a recursive function is recursive call.

```
... f1() {
      :
    if (..) return;
      :
A2  f1();   ← Tail Recursion.
}

f1(); A1
```



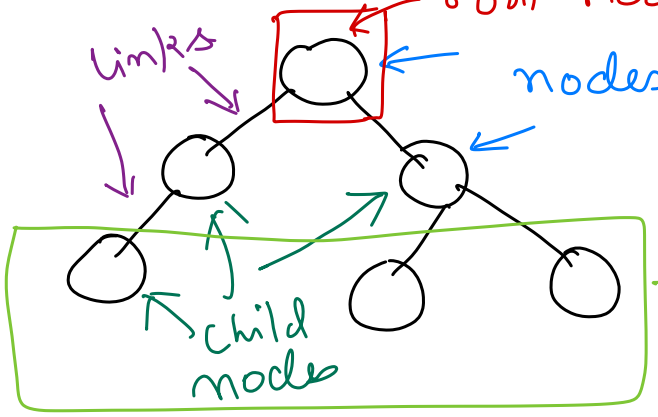


Tree

→ Non-linear / Hierarchical data structure.

→ Tree is a collection of nodes.

→ Relationship between nodes in a tree is of type parent-child.



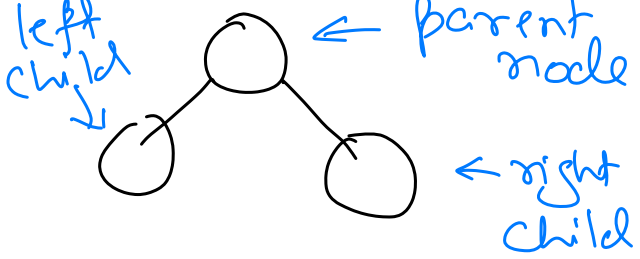
Types of tree

Max number of child nodes any node in tree can have.

2 — Binary Tree

3 — Ternary Tree

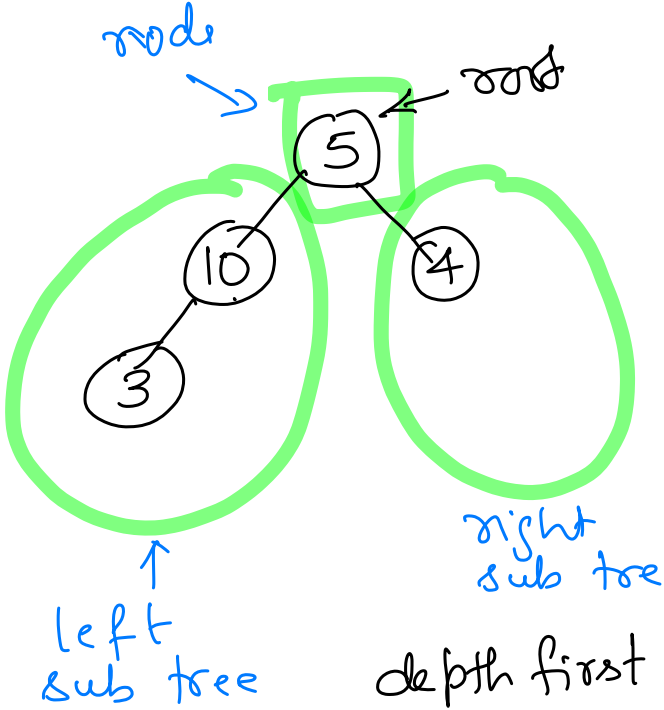
... n — n-ary Tree



Data stored

- Expression tree.
- Family tree.
- Search Tree
  - Binary Search Tree
  - Height Balanced Search Tree (AVL, Red-Black, 2-3 Tree, B-Tree).
  - Trie

Binary Tree Traversals



- Depth first
  - Inorder → 2 1 3
  - Postorder → 2 3 1
  - Preorder → 1 2 3
- Breadth first
  - Level order

depth first traverse (root) =

{ done, if root is empty

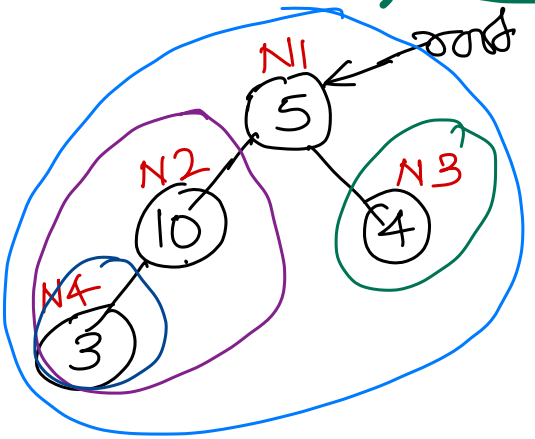
1. process root
2. traverse (root's left child)
3. traverse (root's right child)

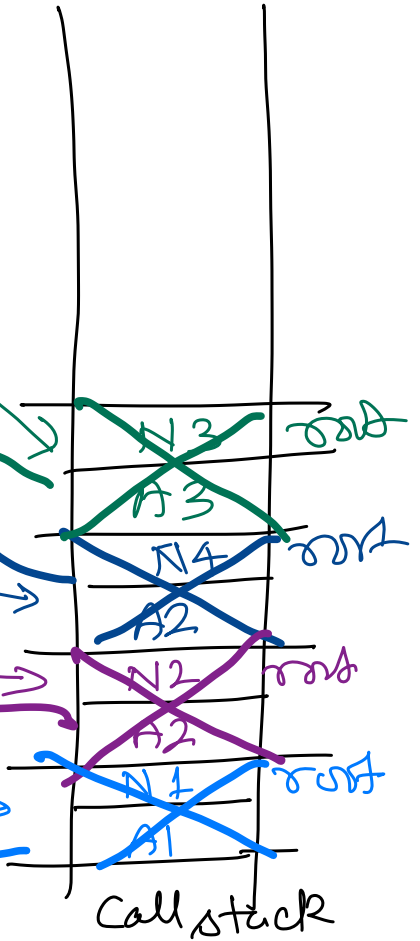
} otherwise

Empty tree

root → empty

PreOrder(root)
- if (root is empty) then
  - Stop.
- Process root node's data. → Print
- If (root node's left child exists) then
  - PreOrder(root's left child). A2
- If (root node's right child exists) then
  - PreOrder(root's right child) A3
- Stop.





PreOrder (root) A1

O/P: 5 10 3 4

PostOrder(root)
- if (root is empty) then
  - Stop.
- If (root node's left child exists) then
  - PostOrder(root's left child).
- If (root node's right child exists) then
  - PostOrder(root's right child).
- Process root node's data.
- Stop.

O/P of Post Order

~~10 3 4 5~~

3 10 4 5

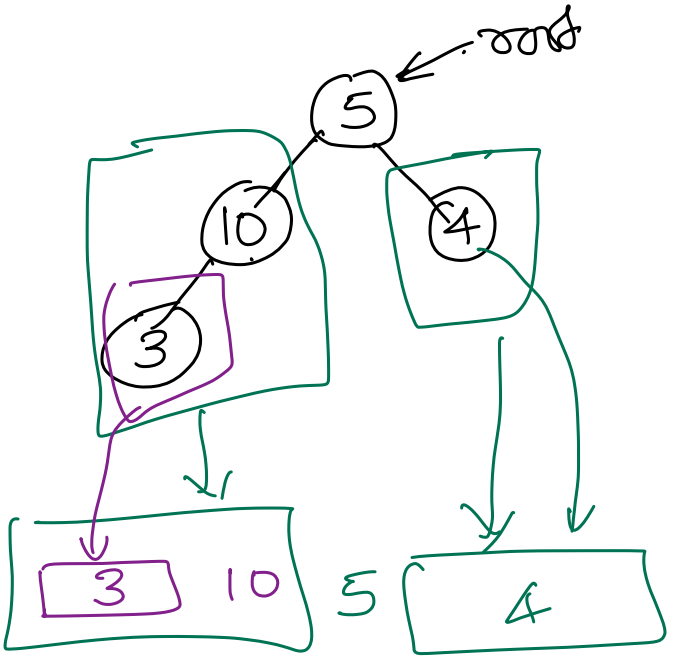
O/P of Inorder

~~10 3 5 4~~

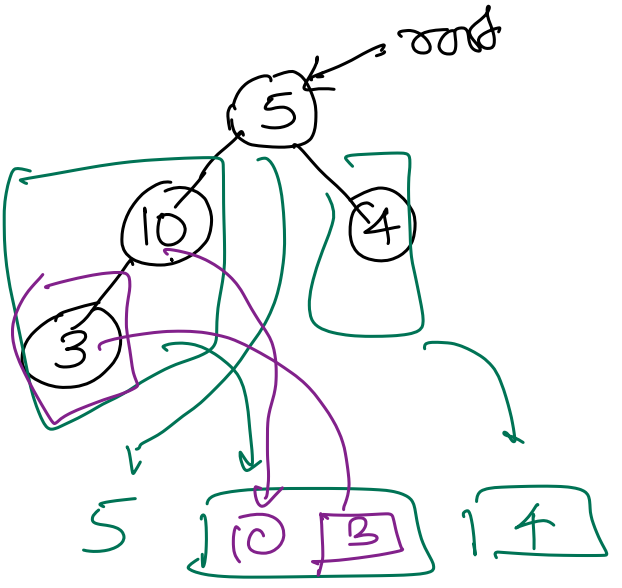
3 10 5 4

InOrder(root)
- if (root is empty) then → Stop.
- If (root node's left child exists) then → InOrder(root's left child).
- Process root node's data.
- If (root node's right child exists) then → InOrder(root's right child).
- Stop.

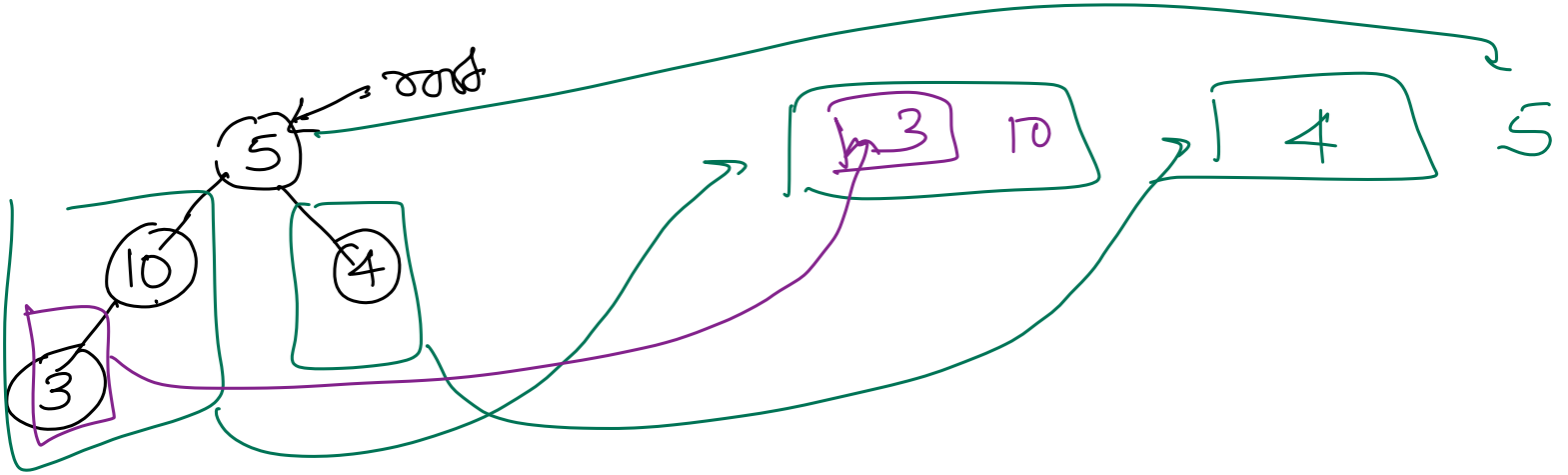
```
if (root == null)
    return;
if (root.left != null)
    InOrder(root.left);
Print root.data;
if (root.right != null)
    InOrder(root.right);
```



Inorder



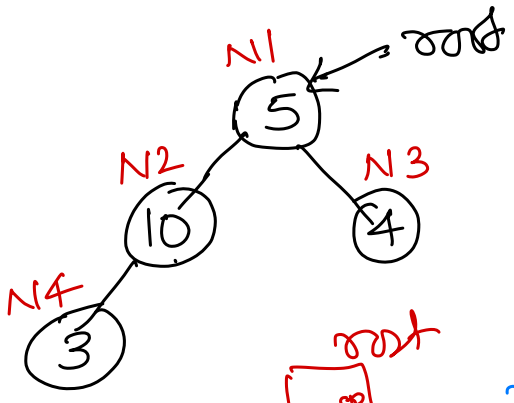
Pre order



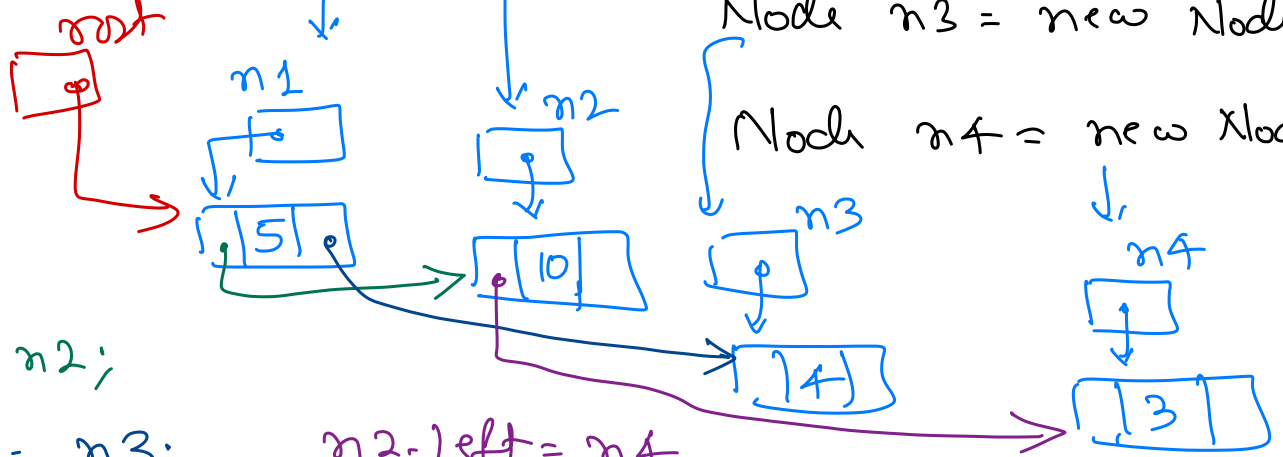
Post order

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

Hard Coding

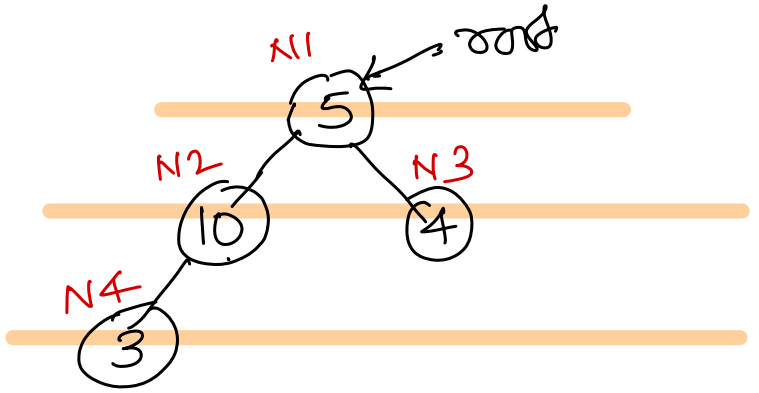



```
Node n1 = new Node(5);
Node n2 = new Node(10);
Node n3 = new Node(4);
Node n4 = new Node(3);
```




```
n1.left = n2;
n1.right = n3;
n2.left = n4
root = n1;
```

Level Order Traversal



Queue

~~N1~~ ~~N2~~ ~~N3~~ ~~N4~~

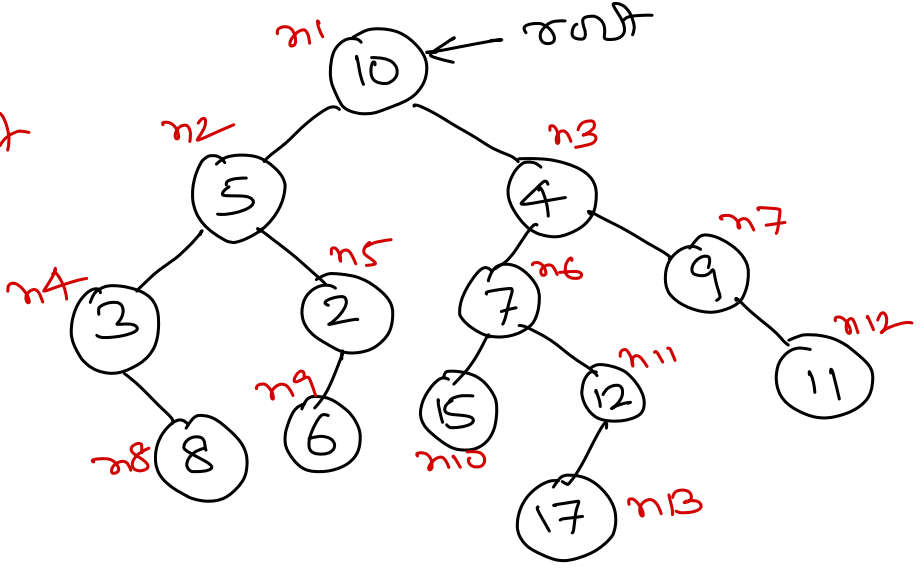
O/p 5 10 4 3

current → ~~N1~~
~~N2~~
~~N3~~ N4

LevelOrderTraversal(root)
- if (root is empty) then
  - Stop.
- Add the root node to the queue.
- while (queue is not empty) do
  - Get a node from queue.
  - Process the node.
  - Add the non-empty childs of the node to the queue.
- Stop.

Exercise

→ Find O/P of Pre Order & PostOrder Traversal



→ Hard code this tree in JAVA code.

→ Find O/P of Inorder & Level Order traversal.

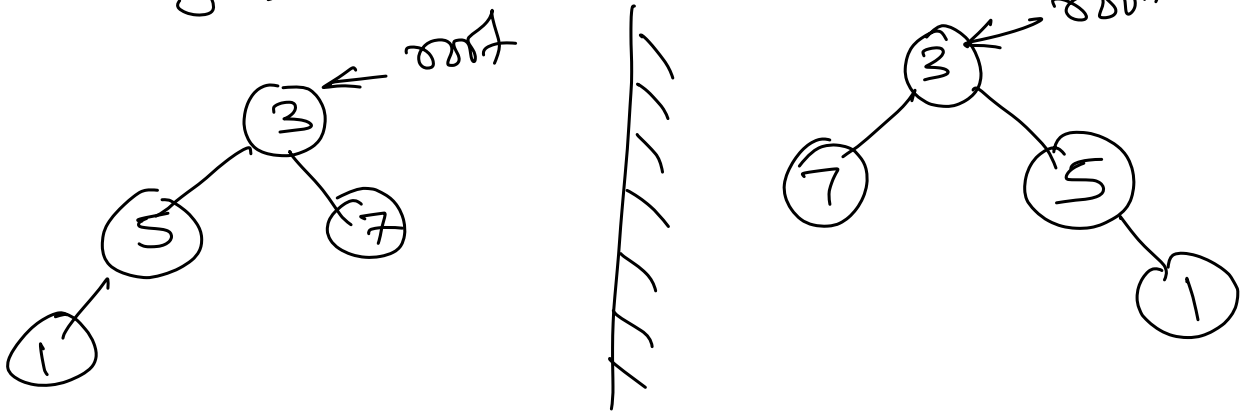
① Count number of nodes in a binary tree.

$$\text{CountNodes}(root) = \begin{cases} 0, & \text{if root is empty.} \\ 1 + \text{CountNodes}(root\text{'s left child}) \\ \quad + \text{CountNodes}(root\text{'s right child}) \end{cases}$$

② Count number of leaf nodes in a binary tree.

③ Count frequency of occurrence of a value in binary tree.

④ Invert a binary tree / Find mirror image of binary tree.



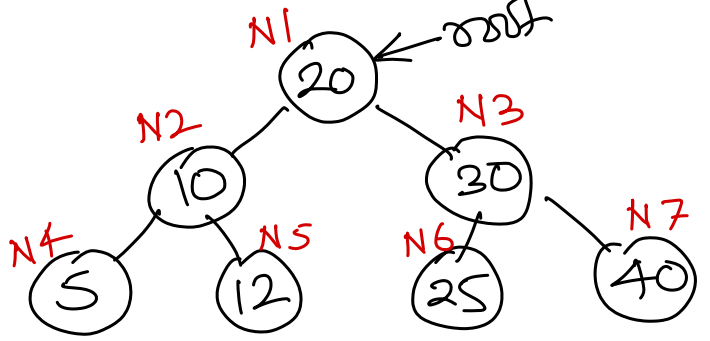
⑤ Print using level order but O/P of each level on seperate lines.

Binary Search Tree (BST)

→ A binary tree in which each node satisfies BST property.

BST Property

Value of nodes in left subtree < Parent node value < Value of nodes in right subtree.



Search (2)

current → ~~N1~~

~~N2~~

~~N4~~

empty

NOT FOUND

Search (25)

current → ~~N1~~

25 > N1.data

→ ~~N3~~

25 < N3.data

→ N6

FOUND

Search (root, value)

- if (root is empty) return false;
- if (root's data = value) return true;
- if (value < root's data)
  return Search (root's left, value);
- return Search (root's right, value);

Search(root, value) ← Removing tail recursion

- result = false;
- while (root is NOT empty)
  - if (root's data = value) return true;
  - if (value < root's data)
    - root = root's left child
  - else
    - root = root's right child.
- return result;

Search (element)
- Set current to root.
- while (current is not empty) do
  - if (current node's data = element) then
    - Element found.
    - Stop.
  - if (element < current node's data) then
    - Move current to current's left child.
  Else
    - Move current to current's right child.
- Element NOT found.
- Stop