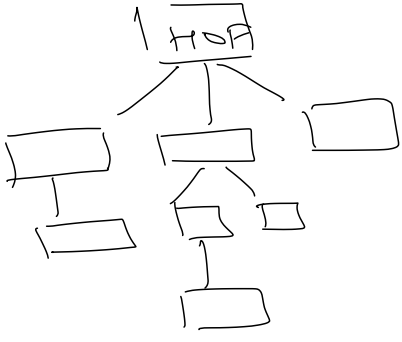
Tree

Linear Data Structure: Array, List.

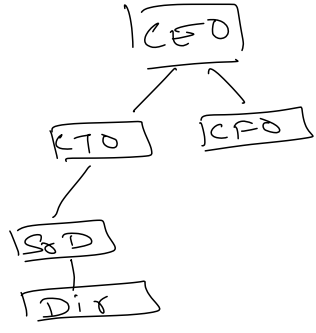
Hierarchical Data Structure: Tree, Graph.

↳ Data is stored in a non-linear manner.

Family Tree

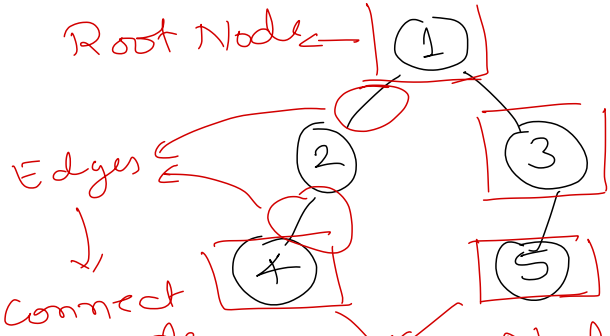


Org Chart



DOM - Document Object Model

→ Root, Nodes, Leaf, Children,



Nodes
↳ They store data.

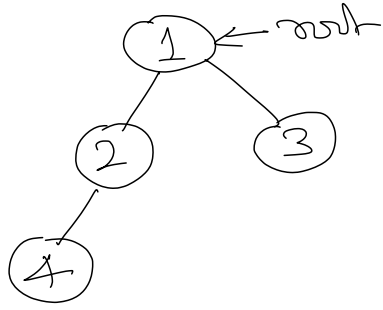
Edges
↓
Connect one node with another

Leaf

Node 1 is parent of child nodes 2 & 3.

Node 2 is parent of Node 4.

Root node do not have parent.
Leaf nodes do not have any childrens.



Path: Nodes visited to reach from a source node to destination.

Height: Length of the longest path from root to a leaf node.

$1 \to 3 = 1$

$1 \to 2 \to 4 = 2$

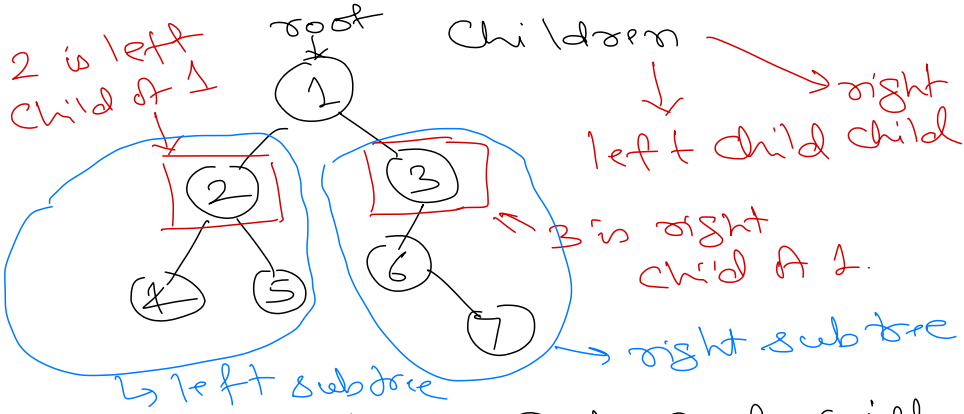
Height = 2

Depth: Length of the path from root to that node

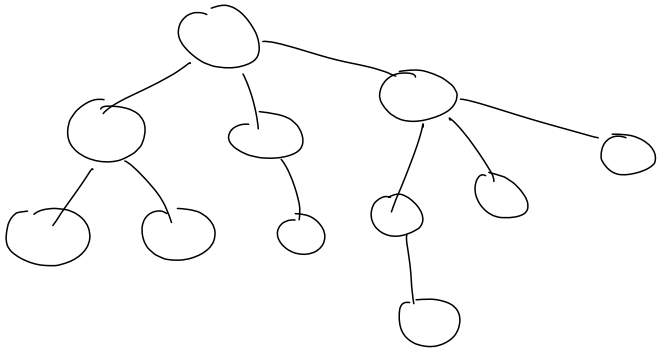
Depth of 2,
Path of 2 from root: $1 \to 2 = 1$
Depth of 2 = 1

Types of trees

→ Based on number of child nodes.
  → Binary Tree: Each node will have max of 2

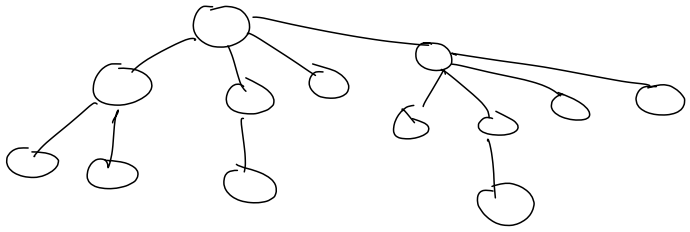


→ Ternary tree: Each node will have max of 3 childrens.



→ n-ary Tree: Each node will have max of N childrens.

N = 4



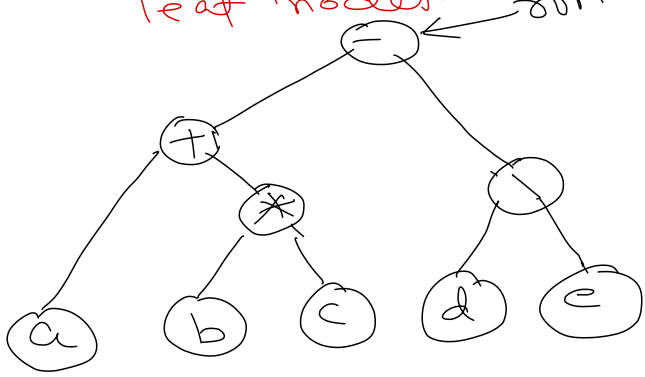
↳ Based on how data is stored/ organised.

→ Expression tree.

a + b * c - d / e

Operands are stored at leaf nodes.

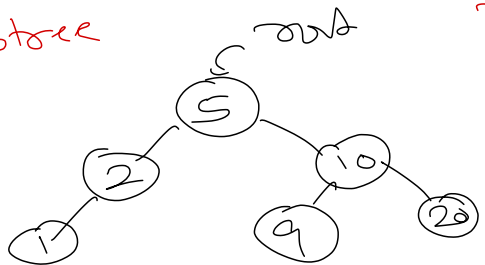
Operators are stored at non leaf nodes.



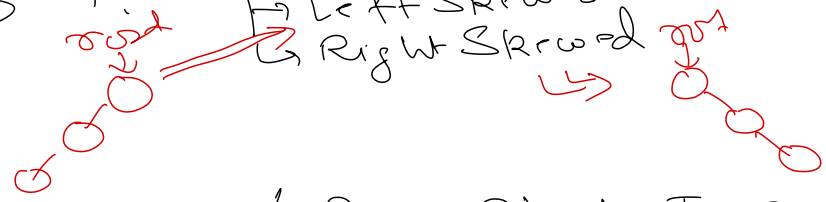
→ BST : Binary Search Tree

↳ Each node satisfies following property

Data of nodes in left subtree < Node Data < Data of Nodes in right subtree.

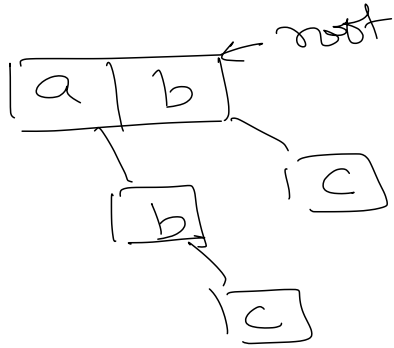


- Skewed tree
  - Left Skewed
  - Right Skewed

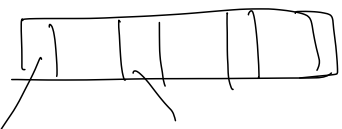


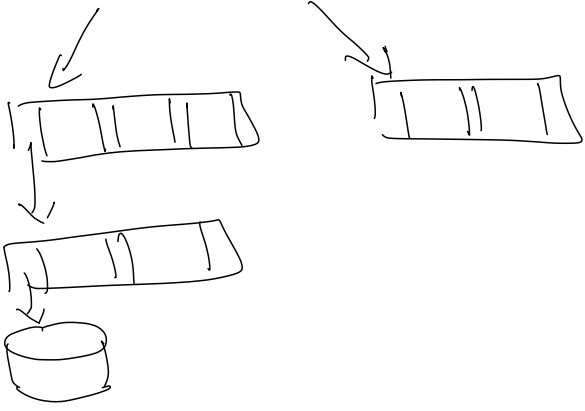
- AVL / Red-Black Tree
  - Height balanced BST
- Trie : Dictionary
  - N-ary tree.

ab a abc bc



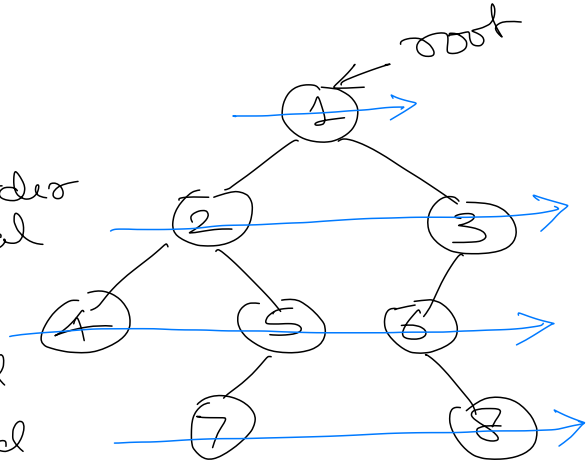
- B-Tree : Self balancing tree
  - N-ary tree
    - File Systems
    - Data base Systems.
      - Indexing.

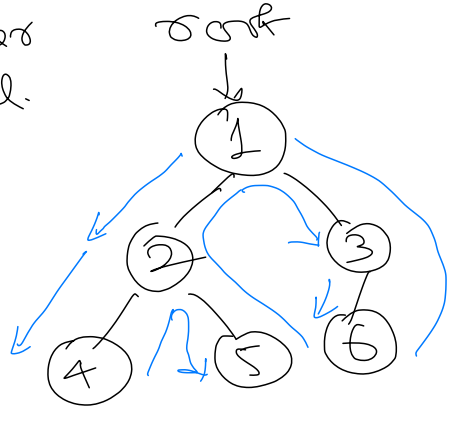




Traversal

- BFS :⇒ Level Order Traversal
- DFS
  - InOrder Traversal
  - PreOrder Traversal
  - PostOrder Traversal.



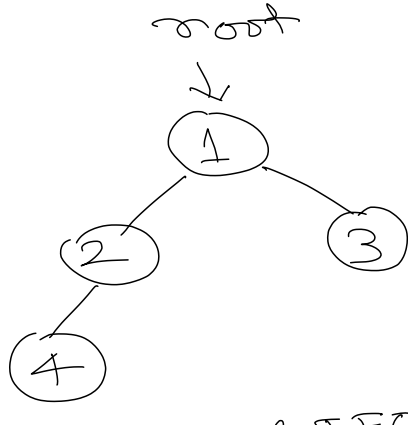


Process nodes in left subtree
Process node's data
Process nodes in right subtree.

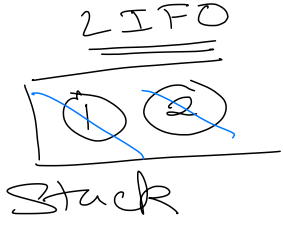
Preorder
↳ Process node's data
Process nodes left subtree
Process nodes right subtree

Post Order
↳ Process nodes left subtree
Process nodes right subtree
Process node's data

PreOrder (root)



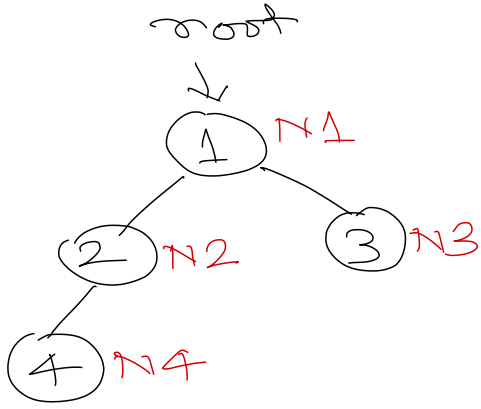
1. Process root's data.
   Print 1
2. Goto Node 2
3. Print 2
4. Goto Node 4.
5. Print 4.
6. Pop parent node. for 4
   ↳ Node 2
7. Pop parent node for 2
   ↳ Node 1.



8. Goto Node 3

9. Print 3.

Algorithm for PreOrder Traversal
PreOrder(root)
- if root is empty then
  - Stop.
- Process root node's data.
- If root node's left child exists then
  - PreOrder(root's left child). C2
- If root node's right child exists then
  - ProOrder(root's right child). C3



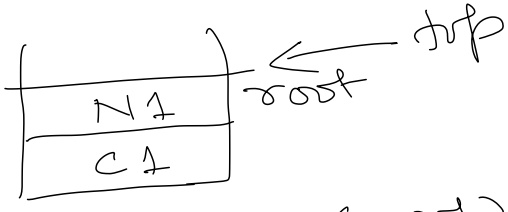
PreOrder (root) C1

O/p : 1 2 4 3

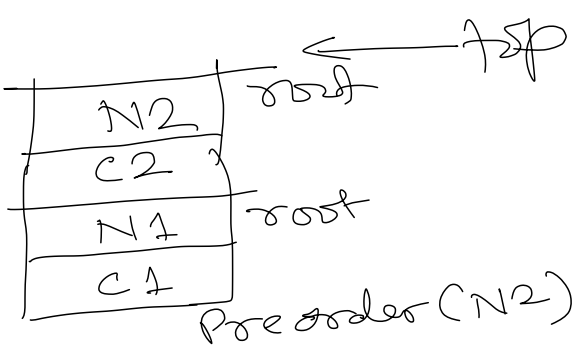
Function argument root

Stack Frame for PreOrder call

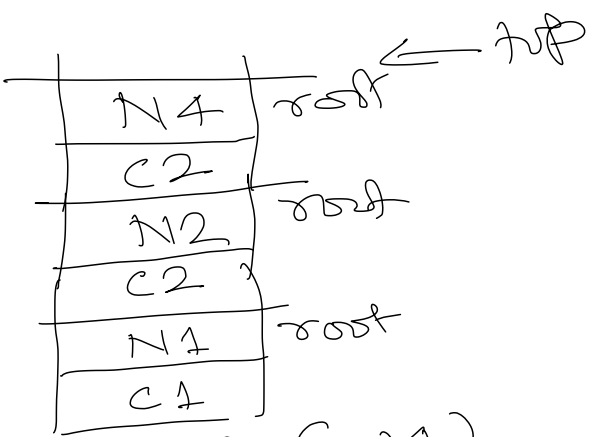
Return Address.



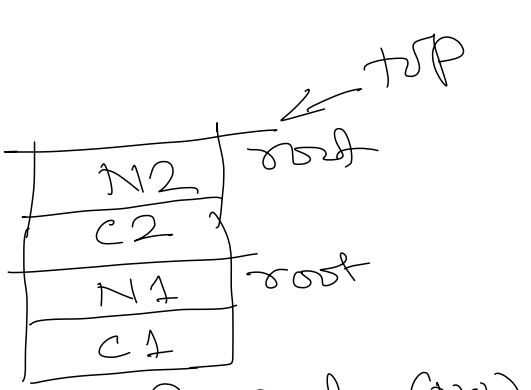
PreOrder(root)



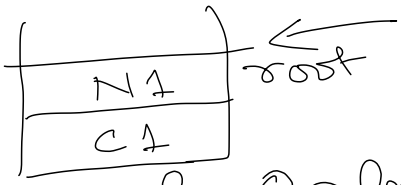
PreOrder(N2)



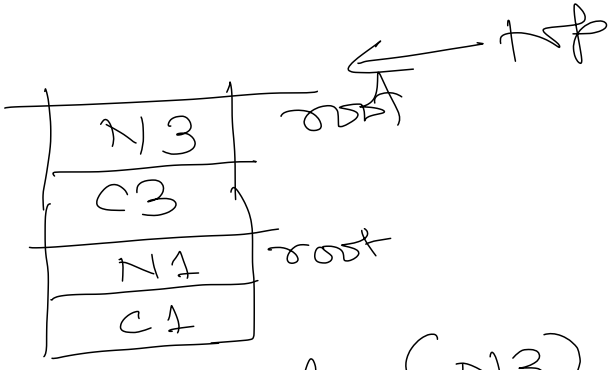
PreOrder(N4)



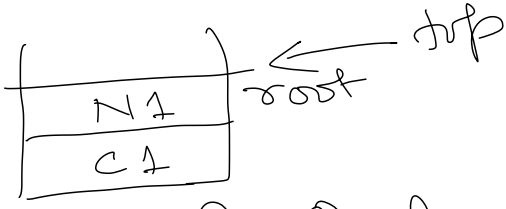
After PreOrder(N4) is over.



After PreOrder(N2) is over.



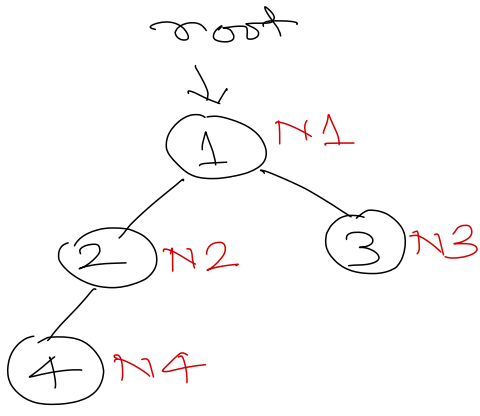
PreOrder(N3)



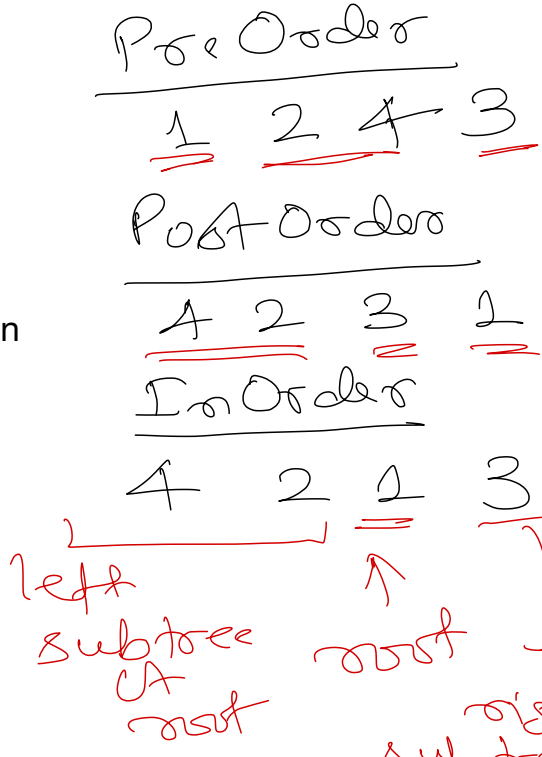
After PreOrder(N3) is over.

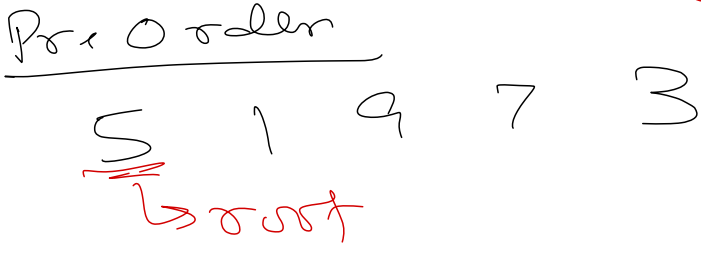
After PreOrder(root/N1) is over.

Algorithm for PostOrder Traversal
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.



Algorithm for InOrder Traversal
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).





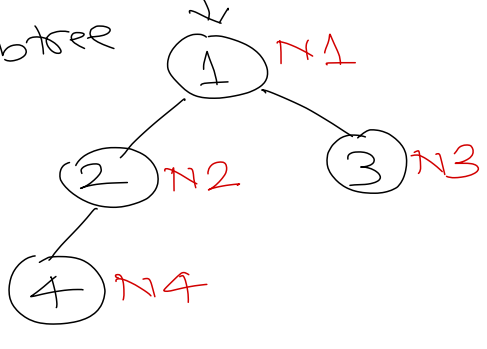
```
class Node {
    int data;
    Node lchild;
    Node rchild;
}

interface BinaryTree {
    int[] preOrder();
    int[] postOrder();
    int[] inOrder();
}
```

→ Count number of nodes in a binary tree.

= 0, if tree is empty.

= Nodes in left subtree of root + nodes in right subtree of root + 1 (for root node.)



→ Find height of a binary tree.

```
int Height();
```

→ Count frequency of an element in a binary tree

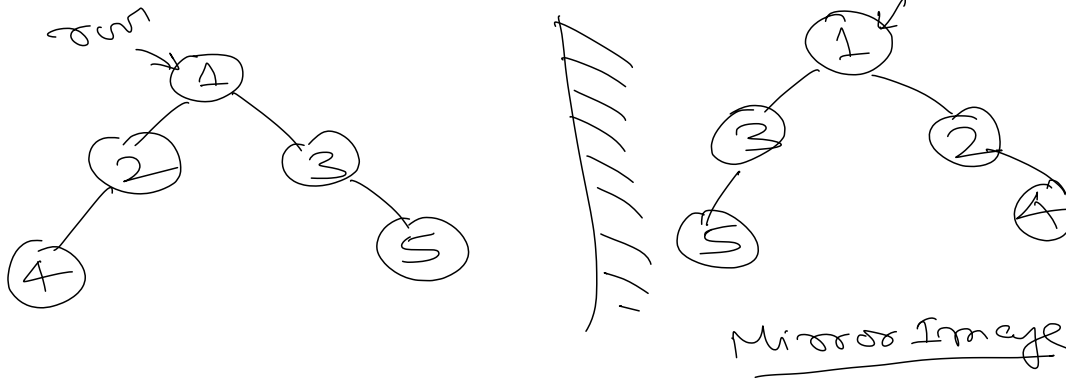
```
int FindFrequency(int elem);
```

→ Count number of leaf nodes in a binary tree

```
int CountLeaf();
```

→ Convert tree into mirror image.

```
void MakeMirror();
```



Mirror Image

→ Find depth of a node.

int NodeDepth(int elem)

NodeDepth(5)