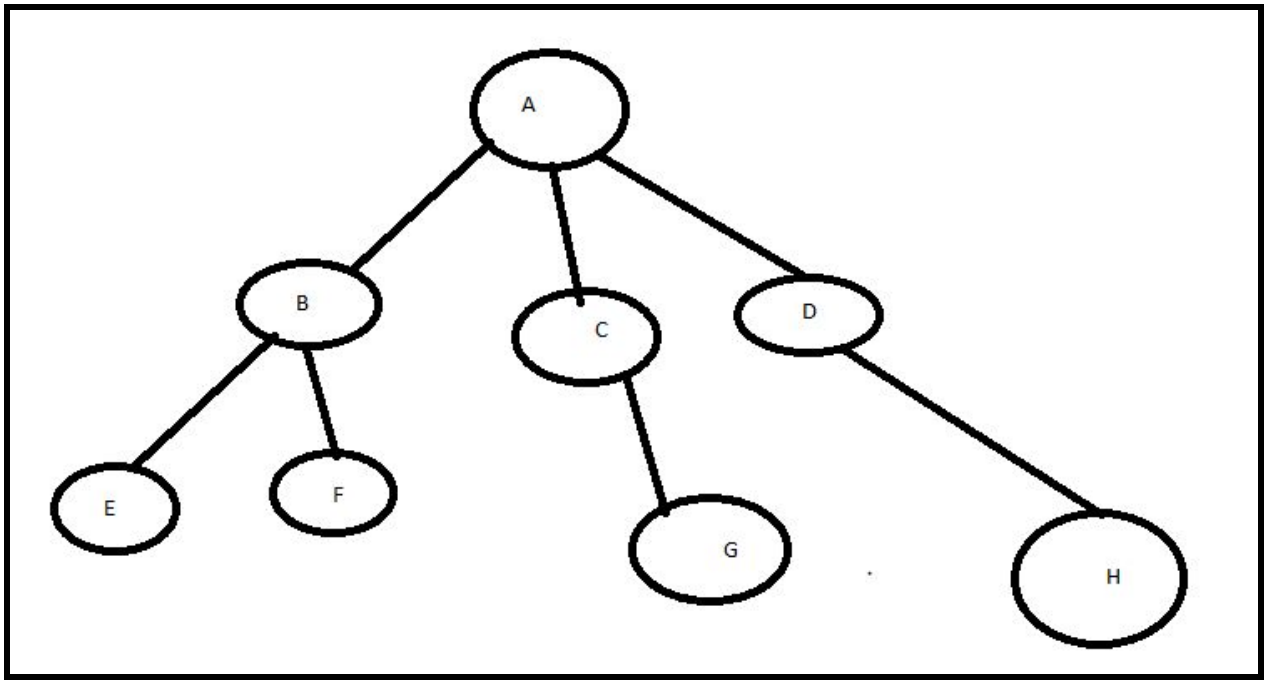
Trees

Contents

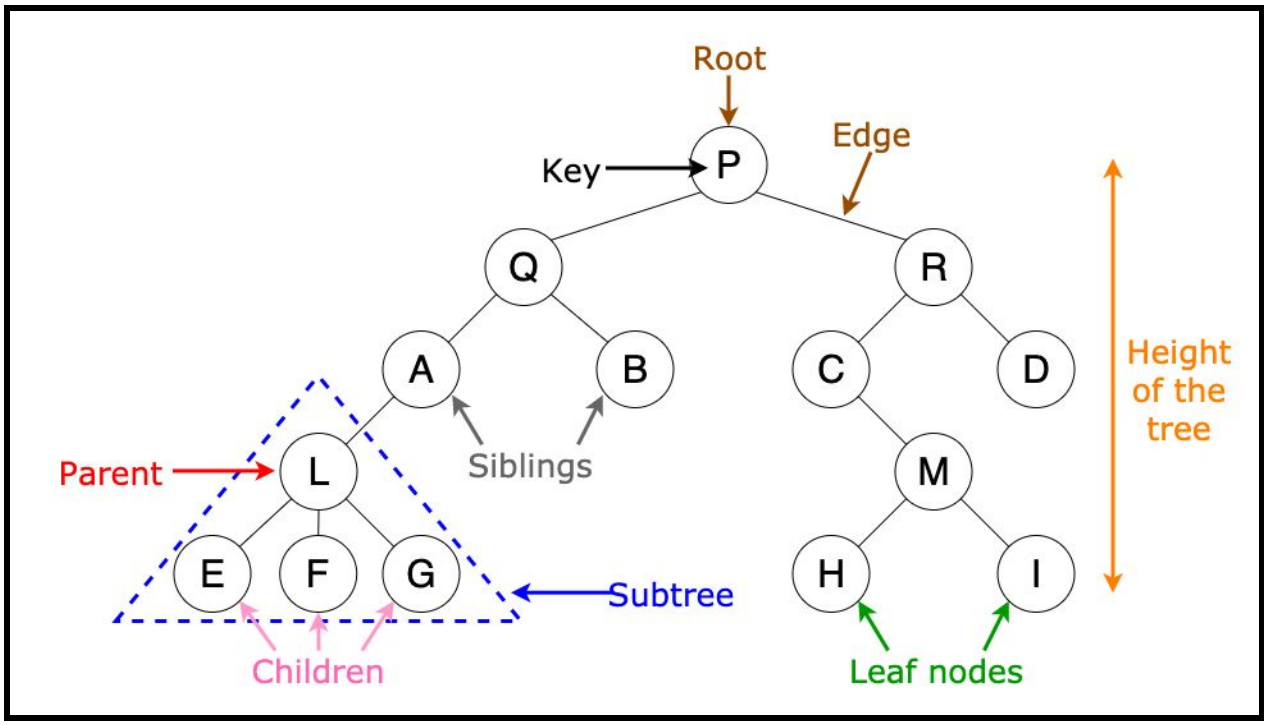
Author - Abhijeet Kumar

1. What is a Tree?

A tree is a data structure similar to Linked List but instead of each node pointing simply to the next node in a linear fashion , in trees it can point to multiple nodes. So we can say that tree in an example of non-linear data structures.



2. Properties of Trees



Root - The root of the tree is the node with no parents.There can be at most one root node in a tree. (node P in the above example).

Edge - An edge refers to the link between parent and child nodes.

Key - Value at node is called key.

Parent - The node which is predecessor of any node is called parent node (node P,Q,R,A,C,L,M all are parent)

Children - The node which is below any node connected by edge is called child node.

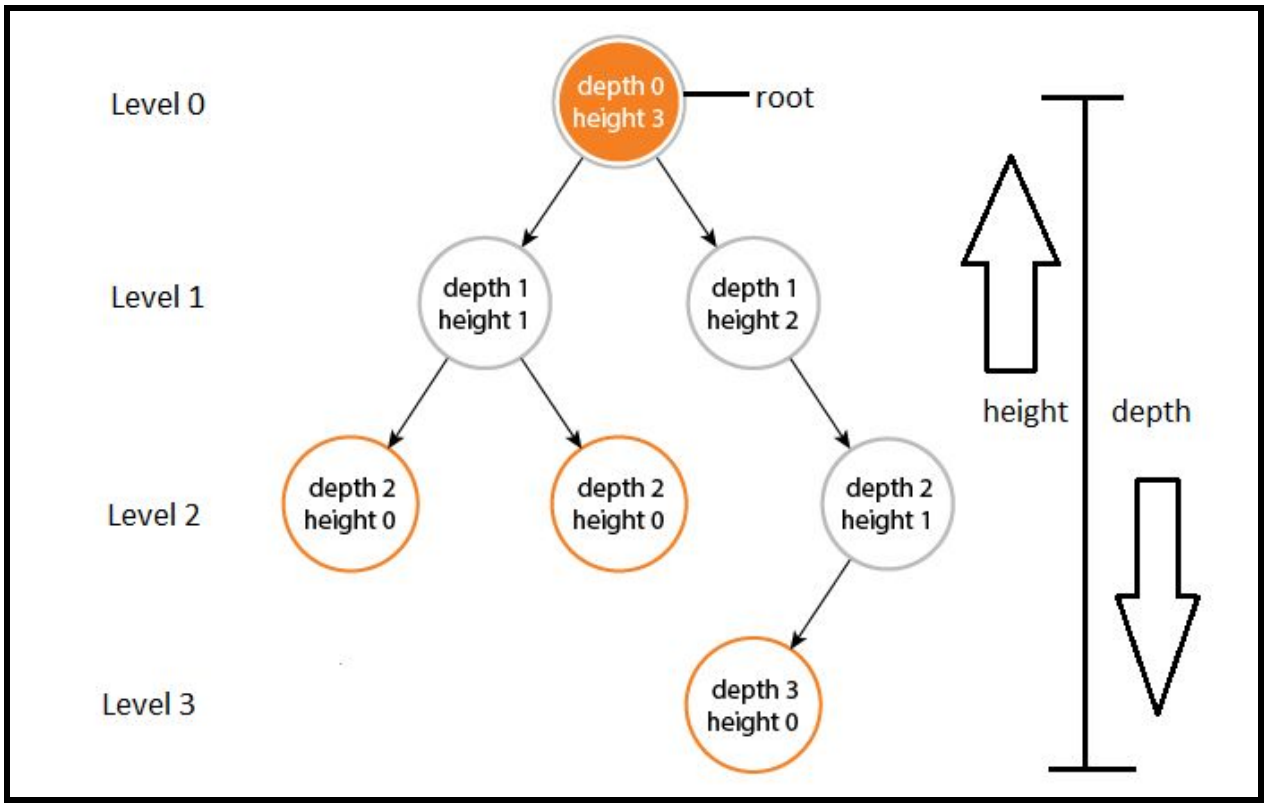
Siblings - Child node of same parent node are called siblings (node A and B are siblings).

Leaf Nodes - The node which has no children are called leaf nodes. (node E, F,G , H, I are all leaf nodes).

Height of a tree - Height of root node from the lowest level leaf node. (here height = 4)

Depth of a tree - Level away from the root node (Node A depth = 2, Node M depth =3)

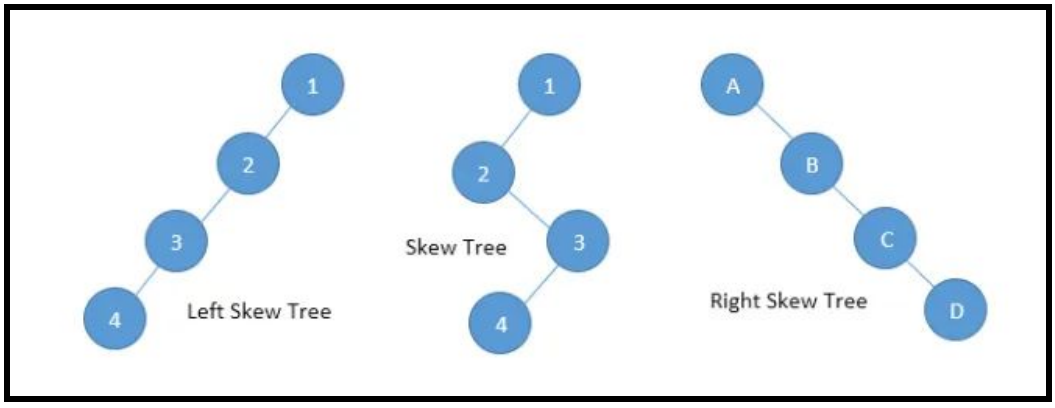
NOTE - The height of a tree would be the height of its root node or equivalently to the depth of its deepest node.



Level of Tree - Set of all the nodes at a given depth or at a given height are at the same level. (root at level 0).

NOTE - If every node has only one child except the root node then it is called skew tree.

- If a tree has only one child and on to its left is called a **left skew tree.**
- If a tree has only one child and on to its right is called a **right skew tree.**



3. Binary Trees

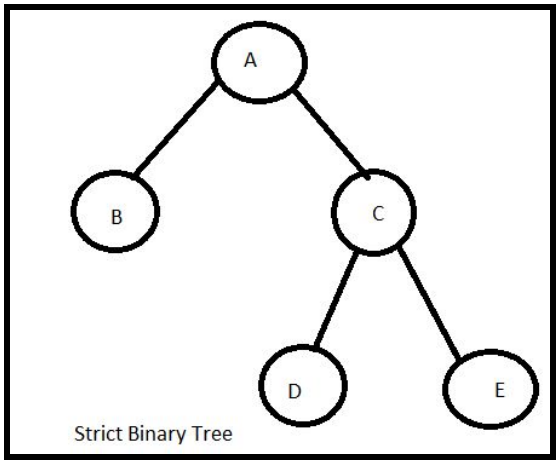
Introduction

A Tree is called a binary tree if it has at most two children.(can have 0 and 1 children).
Empty Tree is also a valid binary tree.
Left side of the root is called the left child of the binary tree and the right side is called right child of the binary tree.

Types of Binary Tree

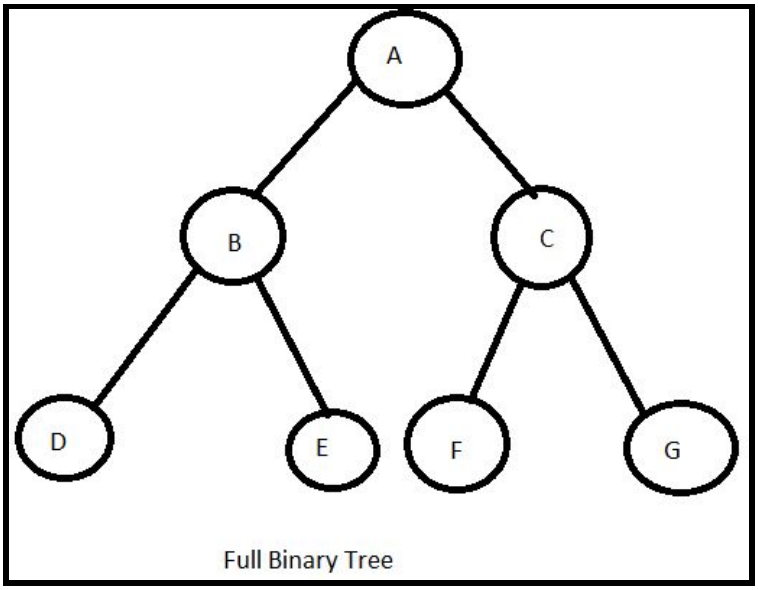
1. **Strict Binary Tree**

A Binary Tree is called a strict binary tree if it has exactly two children or no children.



2. Full Binary Tree

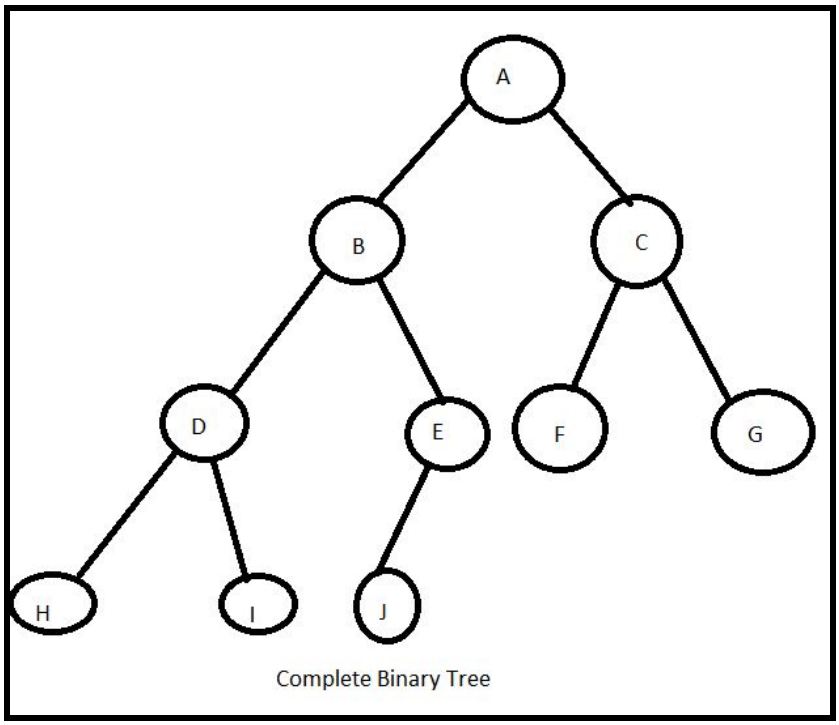
A Binary Tree is call full binary tree if it has all the tree with two children except the leaf node.



Full Binary Tree

3. Complete Binary Tree

A Binary Tree is called a complete binary tree if all the levels are completely filled except the last level and the last level has all keys as left as possible.

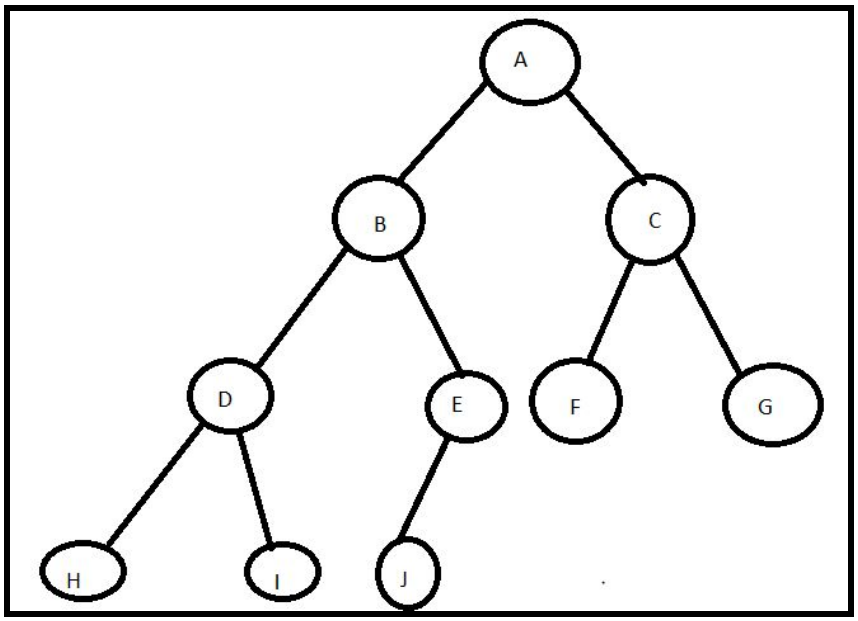


Complete Binary Tree

Properties of Binary Tree

NOTE - For properties assume height of tree (h) and height of root node (0)

- The number of nodes in a full binary tree is $2^{h+1} - 1$.
  Ex- Height (h) = 2 , so number of mode = $2^{2+1} - 1$ = 8-1= 7
- The maximum number of node at level (l) = 2^{l} .
  Ex- Level (l) = 2 => 2^{2} = 4 (nodes at level 2).
- Total number of leaf nodes in a binary tree = Total number of nodes with two children + 1.
  Eg:-



In the above given figure Total leaf nodes =5 (H, I, J, F, G) and total nodes with two children = 4 (A, B , C , D) .

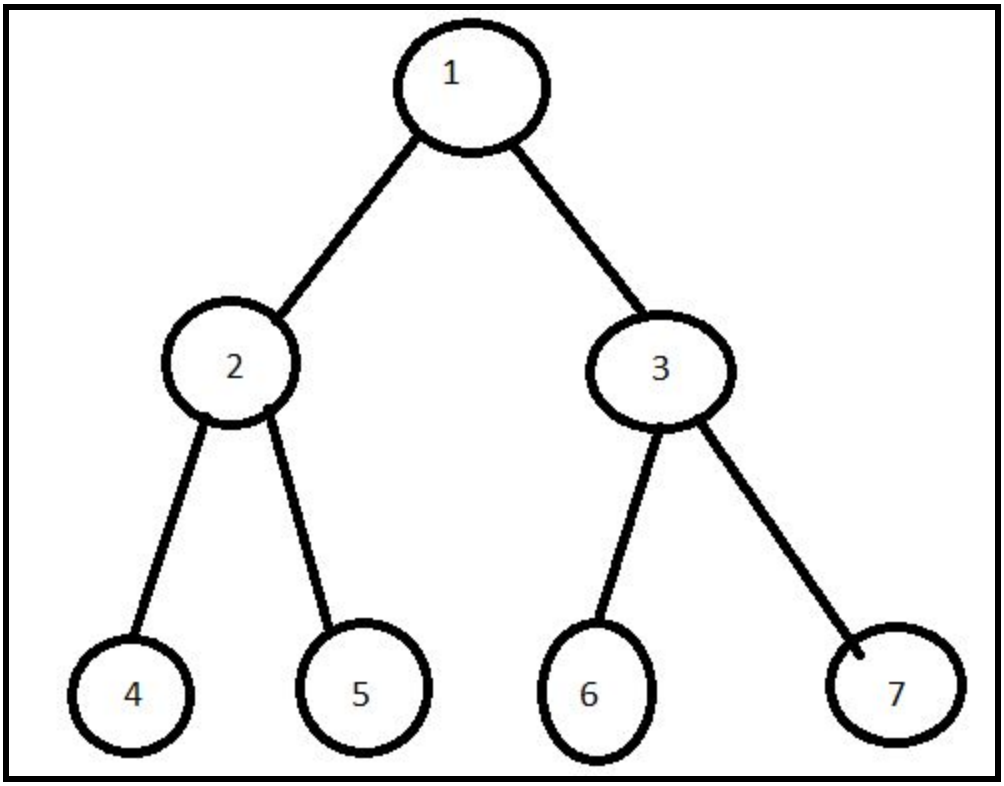
Structure of Binary Trees

```java
public class BinaryTreeNode {
    public int data;
    public BinaryTreeNode left;
    public BinaryTreeNode right;
    public BinaryTreeNode(){}

    public BinaryTreeNode(int data) {
        this.data = data;
        this.left=this.right=null;
    }

}
```

Bean class for a tree . It is similar to a doubly linked list as you can relate. Left and right are again a BinaryTreeNode class.

Binary Tree Traversals

So basically traversal means visiting all nodes of a tree. There are different mechanisms to do that. In this part we will understand that. If you are reading this then I assume that you must be aware of linear data structures. We have seen that in linear data structure elements are visited in sequential order but in trees there are many different ways to achieve this.
We will use the same example shown below for all our traversal



In trees we have different approaches to traverse

- Preorder (DLR) - Data then Left then RIght
- Inorder (LDR) - Left then Data then Right
- Postorder (LRD) - Left then Right then Data

Preorder (DLR)

- So in Preorder traversal, each node is processed before (pre) either of its subtrees.
- This is the simplest traversal which I think and is easy to understand.
- In this, the node is processed before the subtrees but still we require some information that needs to be maintained while processing down the tree.
- In the given example 1 is processed and after that the left subtree but after processing the left subtree we still require the root information to move to right.
- For that reason we will use **Stack** which maintains LIFO strategy.

Let's take a look at the
first then after that will look at the iterative approach.

```java
public class BinaryTree {
    BinaryTreeNode root;
    public  BinaryTree(){ root=null;}     //constructor to initialize root
    public void preOrder(BinaryTreeNode root){
        if(root!=null)
        {
            System.out.print(root.data);
            System.out.print(" ");
            preOrder(root.left);
            System.out.println("");
            System.out.println("root value-"+root.data);
            preOrder(root.right);
            System.out.println("root value-"+root.data);
        }
    }
```

```java
    public static void main(String[]args){
     BinaryTree tree = new BinaryTree();
       tree.root = new BinaryTreeNode( data: 1);
       tree.root.left = new BinaryTreeNode( data: 2);
       tree.root.right=new BinaryTreeNode( data: 3);
       tree.root.left.left= new BinaryTreeNode( data: 4);
       tree.root.left.right= new BinaryTreeNode( data: 5);
       tree.root.right.left= new BinaryTreeNode( data: 6);
       tree.root.right.right= new BinaryTreeNode( data: 7);
       tree.preOrder(tree.root);
    }
}
```

After running this code you will get the output like this:- **1 2 4 5 3 6 7**

```
"C:\Program Files\Java\jdk1.8.0_231\bin\java.exe" ...
1 2 4
root value-4
root value-4

root value-2
5
root value-5
root value-5
root value-2

root value-1
3 6
root value-6
root value-6

root value-3
7
root value-7
root value-7
root value-3
root value-1

Process finished with exit code 0
```

So you easily analyse from this output ,how this recursive function is working and traversing the complete tree in preorder.
Now we will see the **iterative approach** of preorder traversal.
So for an iterative approach we will use stack as I mentioned above.

- Create an empty **stack** and push root to the stack if it is not null.
- Create a List to store the element traversed in stack.
- While iterating through the tree , follow this
    - Pop the element from the stack and print it.
    - Push the right child of the popped element to stack.
    - Push the left child of the popped element to stack.
- And here it's done.

NOTE - Push right child of the popped element before the left child to make sure that left subtree is processed first.

Now let’s make our hands dirty on it.

```java
import java.util.*;
public class BinaryTreeIterative {
    BinaryTreeNode root;
    public  BinaryTreeIterative(){ root=null;}
    public void preOrderIterative(BinaryTreeNode root){
        ArrayList<Integer>result=new ArrayList<>();
        Stack<BinaryTreeNode>elements=new Stack<>();
        if(root==null)
           System.out.println("Empty tree");
        elements.push(root);
        while(!elements.isEmpty()){
            BinaryTreeNode node = elements.pop();
            result.add(node.data);
            if(node.right!=null)      //condition to keep in mind
                elements.push(node.right);
            if(node.left!=null)
                elements.push(node.left);
        }
        System.out.println(result);
    }
```

```
public static void main(String[]args){
    BinaryTreeIterative tree = new BinaryTreeIterative();
    tree.root = new BinaryTreeNode( data: 1);
    tree.root.left = new BinaryTreeNode( data: 2);
    tree.root.right=new BinaryTreeNode( data: 3);
    tree.root.left.left= new BinaryTreeNode( data: 4);
    tree.root.left.right= new BinaryTreeNode( data: 5);
    tree.root.right.left= new BinaryTreeNode( data: 6);
    tree.root.right.right= new BinaryTreeNode( data: 7);
    tree.preOrderIterative(tree.root);
}
```

Here also we get the same output **1 2 4 5 3 6 7.**
Time Complexity - O(n).

Inorder (LDR) - Left then Data(root) then Right

- Traversal of root is done in between the left subtree and right subtree.

NOTE - Inorder is used in BST(Binary Search tree) to print the nodes in increasing order. (If you are given a BST then just apply Inorder to print nodes in increasing order)

So let's first try a **recursive approach** and then will move to an iterative approach.

```
public void inorderRecursive(BinaryTreeNode root){
        if(root!=null)
        {
            System.out.println("");
            System.out.println("root data 1-"+root.data);
            inorderRecursive(root.left);
            System.out.print("element printed="+root.data);
            System.out.print(" ");
            inorderRecursive(root.right);
            System.out.println("");
            System.out.println("root data 2-"+root.data);
        }
}
```

Output of Inorder traversal - **4 2 5 1 6 3 7**

By Debugging the code or by adding the print statement you can check how your code is working.
Now quickly move to an **iterative approach.**
Below is the step wise execution of the problem.

- Create an empty stack.
- Initialize a Node name current and assign root value to it.
- **Add the current to stack and point current to current.left until current is NULL.**
- If current points to NULL and stack is not empty do
    - Pop element from stack.
    - Print the popped item and make current =>popped_element.right.
    - Again follow bullet point 3 which is in bold.

Let's look into the practical solution of it.

```
public void inorderIterative(BinaryTreeNode root){
    ArrayList<Integer>result=new ArrayList<>();  // to store the result
    Stack<BinaryTreeNode>stack_element=new Stack<>();  // empty stack
    BinaryTreeNode current=root;
    while(current!=null || (!stack_element.isEmpty())){
        while(current!=null){
            stack_element.push(current);
            current=current.left;
        }
        current= stack_element.pop();
        result.add(current.data);
        current=current.right;
    }
    System.out.println(result);
}
```

Time Complexity - O(n)
Look how easy it is.

Postorder(LRD) - Left then Right then Data(root)

- In Postorder traversal root is visited at the last.

Recursive approach

- Recursively traverse the left subtree.

- Recursively traverse the right subtree.
- Visit the root node.

```
public void postOrderRecursive(BinaryTreeNode root){
    if(root!=null)
    {
        System.out.println("");
       // System.out.println("root data 1-"+root.data);
        postOrderRecursive(root.left);
     //   System.out.println("root data 2-"+root.data);
        postOrderRecursive(root.right);
        System.out.print("element printed="+root.data);
        System.out.println("");

    }
}
```

Output of Postorder traversal - **4 5 2 6 7 3 1**

Iterative Approach for Postorder Traversal.
We have two approaches: one is using two stacks and one is using one stack.
First we will see with **two stacks**.

- It is similar to preorder just insert root.left first into the stack then after that push root.right to the stack and you are done.

```java
public void postorderIterative(BinaryTreeNode node) {
    ArrayList<Integer> result = new ArrayList<>();
    Stack<BinaryTreeNode> stack_elements = new Stack<>();
    Stack<BinaryTreeNode> stack2 = new Stack<>();
    if(node==null)
        System.out.println("Empty Tree");
    else{
        stack_elements.push(node);
        while(!stack_elements.isEmpty()){
            BinaryTreeNode current=stack_elements.pop();
            stack2.push(current);
            if(current.left!=null)
                stack_elements.push(current.left);
            if(current.right!=null)
                stack_elements.push(current.right);
        }
    }
    while(!stack2.empty()){
        result.add(stack2.pop().data);
    }
    System.out.println(result);
}
```

Look how simple it is with two stacks.

Postorder Traversal using single Stack.

- In Preorder and Inorder it is straightforward that after visiting a node we just pop it out from the stack but here in Postorder that is not the case.
- After traversing the left tree we will visit the current node and after processing the right subtree again we will visit the same node.
- But point to be noted that we have to process the node on the second visit.

So we can just quickly make Algorithm for this and after that we will move to implementation.

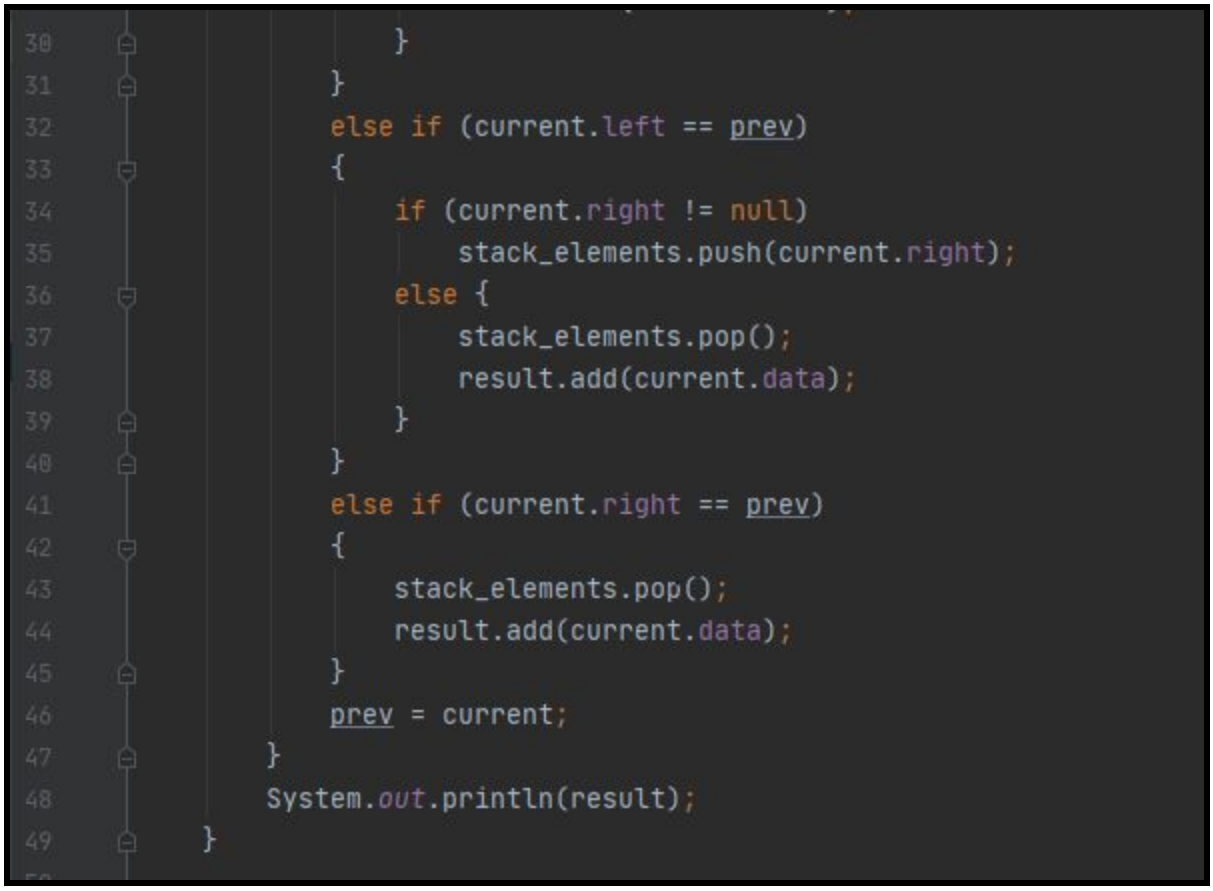
1. Create an Empty stack and a List (to store the result).
2. While root is not NULL. Push root right and then root to Stack and after that make root to point root left child.
3. When root == NULL , pop the top element from stack and set it to root and check whether it has the right child or not,

a. If the right child exists and is at the top of stack , then remove the right child from stack ,push back the root and set root to root's right child.
b. Else print root's data and set root to NULL.

4. Repeat steps 2 and 3 till the stack is not empty.

I know this looks somehow complicated so first try to visualize it in your copies after that move to implementation.

```java
public void postorderIterative(BinaryTreeNode node)
{
    ArrayList<Integer>result=new ArrayList<>();
    Stack<BinaryTreeNode> stack_elements = new Stack<>();
    if (node == null)
        System.out.println(result);
    stack_elements.push(node);
    BinaryTreeNode prev = null;
    while (!stack_elements.isEmpty())
    {
        BinaryTreeNode current = stack_elements.peek();
        if (prev == null || prev.left == current ||
                prev.right == current)
        {
            if (current.left != null)
                stack_elements.push(current.left);
            else if (current.right != null)
                stack_elements.push(current.right);
            else
            {
                stack_elements.pop();
                result.add(current.data);
            }
```



```
            }
        }
        else if (current.left == prev)
        {
            if (current.right != null)
                stack_elements.push(current.right);
            else {
                stack_elements.pop();
                result.add(current.data);
            }
        }
        else if (current.right == prev)
        {
            stack_elements.pop();
            result.add(current.data);
        }
        prev = current;
    }
    System.out.println(result);
}
```

This is all about the basics of trees and some basic traversal algorithms.

HAPPY CODING