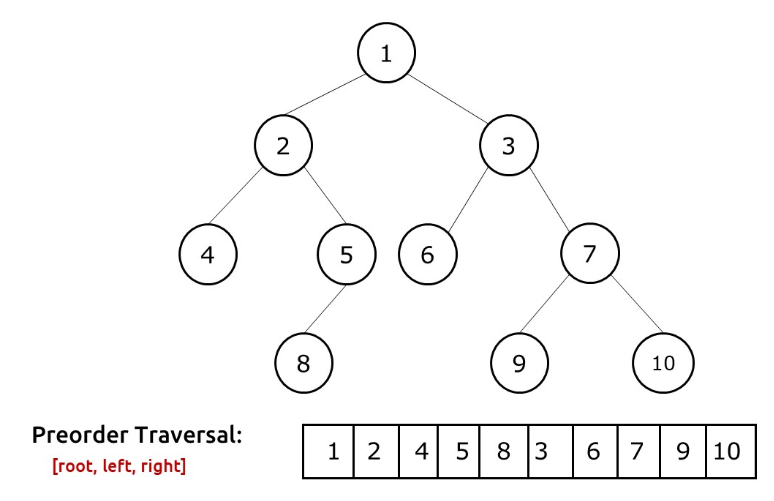
**01. Implementation of Preorder Traversal**

**Problem Statement:** Given a binary tree print the preorder traversal of binary tree.

**Example:**



**Solution 1: Iterative**

**Intuition:** In preorder traversal, the tree is traversed in this way: **root,**left, right. When we visit a node, we print its value, and then we want to visit the left child followed by the right child.

The fundamental problem we face in this scenario is that there is no way that we can move from a child to a parent.

To solve this problem, we use an explicit stack data structure. While traversing we can insert node values to the stack in such a way that we always get the next node value at the top of the stack.

**Approach:**

The algorithm approach can be stated as:

* We first take an explicit stack data structure and push the root node to it.(if the root node is not NULL).
* Then we use a while loop to iterate over the stack till the stack remains non-empty.
* In every iteration we first pop the stack’s top and print it.
* Then we first push the right child of this popped node and then push the left child, if they are not NULL. We do so because stack is a last-in-first-out(LIFO) data structure. We need to access the left child first, so we need to push it at the last.
* The execution continues and will stop when the stack becomes empty. In this process, we will get the preorder traversal of the tree.

**Code**

import java.util.\*;

class Node {

int data;

Node left, right;

Node(int data) {

this.data = data;

left = right = null;

}

}

class Main {

static ArrayList <Integer> preOrderTrav(Node root) {

ArrayList<Integer>preOrder = new ArrayList<Integer>();

if (root == null)

return preOrder;

Stack<Node>s = new Stack<>();

s.push(root);

while (!s.isEmpty()) {

Node topNode = s.peek();

preOrder.add(topNode.data);

s.pop();

if (topNode.right != null)

s.push(topNode.right);

if (topNode.left != null)

s.push(topNode.left);

}

return preOrder;

}

public static void main(String args[]) {

Node root = new Node(10);

root.left = new Node(20);

root.right = new Node(30);

root.left.left = new Node(40);

root.left.right = new Node(50);

root.left.right.left = new Node(80);

root.right.left = new Node(60);

root.right.right = new Node(70);

root.right.right.left = new Node(90);

root.right.right.right = new Node(100);

ArrayList<Integer>preOrder = new ArrayList<>();

preOrder = preOrderTrav(root);

System.out.print("The preOrder Traversal is : ");

for (int i = 0; i < preOrder.size(); i++) {

System.out.print(preOrder.get(i) + " ");

}

}

}

**Output:** The preOrder Traversal is : 1 2 4 5 8 3 6 7 9 10

**Time Complexity: O(N)**

Reason: We are traversing N nodes and every node is visited exactly once.

**Space Complexity: O(N)**

Reason: In the worst case, (a tree with every node having a single right child and left-subtree), the space complexity can be considered as O(N).

**Solution 2: Recursive**

**Iteration:** In preorder traversal, the tree is traversed in this way: **root,**left, right. When we visit a node, we print its value, and then we want to visit the left child followed by the right child.

The fundamental problem we face in this scenario is that there is no way that we can move from a child to a parent.

To solve this problem, we use recursion and the recursive call stack to locate ourselves back to the parent node when execution at a child node is completed.

**Approach:**

In preorder traversal, the tree is traversed in this way: **root, left, right**.

The algorithm approach can be stated as:

* We first visit the root node and before visiting its children we print its value.
* After this, we recursively visit its left child.
* Then we recursively visit the right child.
* If we encounter a node pointing to NULL, we simply return to its parent.

**Code:**

import java.util.\*;

class Node {

int data;

Node left, right;

Node(int data) {

this.data = data;

left = null;

right = null;

}

Node() {

}

}

public class Main {

static void preOrderTrav(Node root, ArrayList<Integer> preOrder) {

if (root == null)

return;

preOrder.add(root.data);

preOrderTrav(curr.left, preOrder);

preOrderTrav(curr.right, preOrder);

}

public static void main(String args[]) {

Node root = new Node(10);

root.left = new Node(20);

root.right = new Node(30);

root.left.left = new Node(40);

root.left.right = new Node(50);

root.left.right.left = new Node(80);

root.right.left = new Node(60);

root.right.right = new Node(70);

root.right.right.left = new Node(90);

root.right.right.right = new Node(100);

ArrayList <Integer> preOrder = new ArrayList<>();

preOrderTrav(root, preOrder);

System.out.print("The preOrder Traversal is : ");

for (int i = 0; i < preOrder.size(); i++) {

System.out.print(preOrder.get(i) + " ");

}

}

}

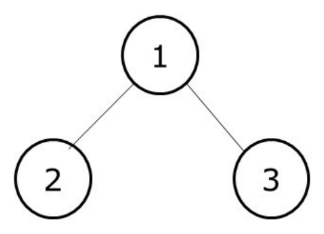
**Output:** The preOrder Traversal is : 1 2 4 5 8 3 6 7 9 10

**Time Complexity: O(N)**

Reason: We are traversing N nodes and every node is visited exactly once.

**Space Complexity: O(N)**

Reason: In the worst case, (a tree with every node having a single right child and left-subtree), the space complexity can be considered as O(N).

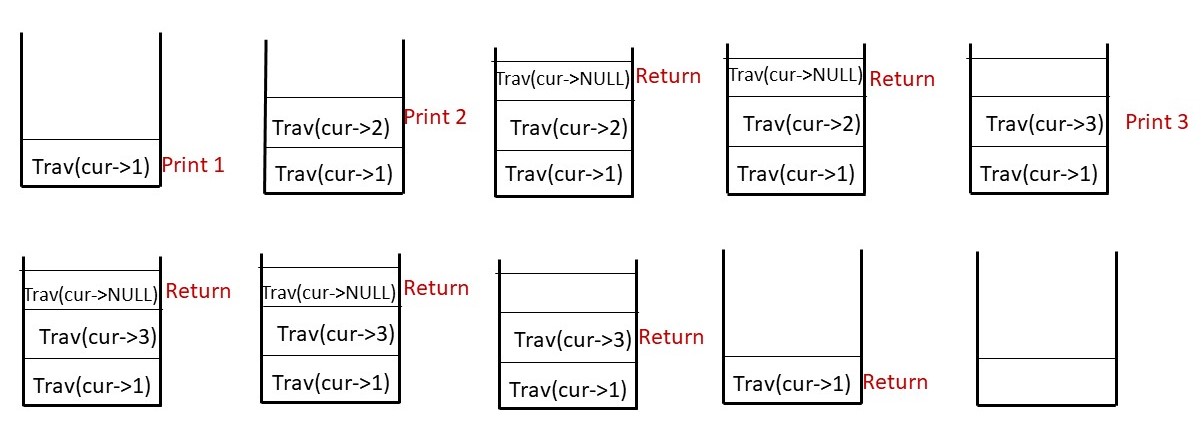
**Explanation:**

Preorder traversal of this tree: 1,2,3.

Initially, we pass the root node pointing to 1 to our traversal function. The algorithm steps are as follows:

* As we are doing a preorder traversal, the first thing we will do is to print the current node value, i.e 1.
* Then we need to move left but how do we do so? Remember that our nodes only have pointers to the children and not to the parent, therefore we can move only from parent to child and not from a child to the parent.
* The answer to this question is **recursion**. We call the same function with the current node pointing to the left child, i.e 2. This second function is then pushed to our call stack. We do our execution which is to visit node 2 print it and then again recursively call its both children. As both of them point to NULL, we will return from them and execution of the second function stops.
* This second function will then be removed from our call stack and we will return to the first function. Then we again recursively call the function for the right child and do the execution.

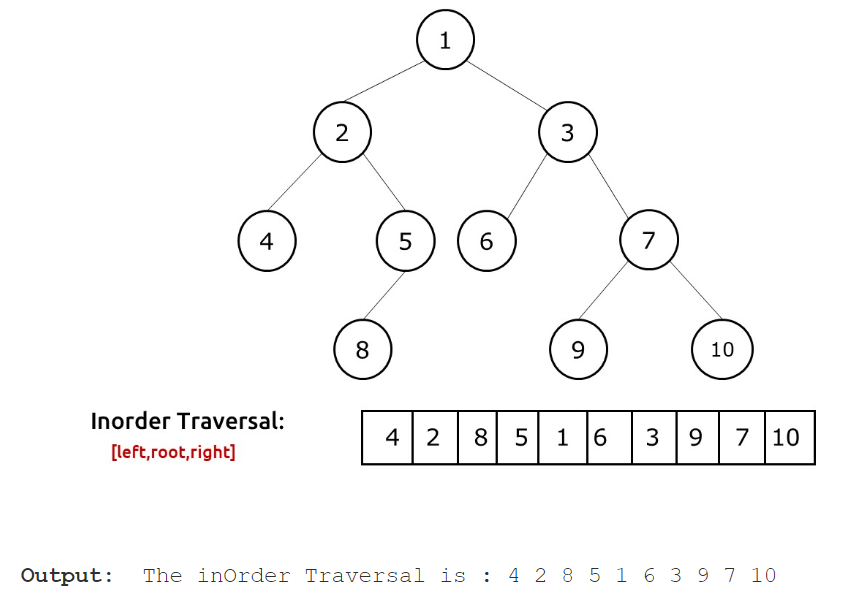
The call stack diagram will help to understand the recursion better.



**02. Implementation of Inorder Traversal**

**Problem Statement:** Given a Binary Tree. Find and print the inorder traversal of Binary Tree.

**Example:**



**Solution 1: Iterative**

**Intuition:**In Inorder traversal, the tree is traversed in this way: **root,**left, right. We first visit the left child, after returning from it we print the current node value, then we visit the right child.

The fundamental problem we face in this scenario is that there is no way that we can move from a child to a parent.

To solve this problem, we use an explicit stack data structure. While traversing we can insert node values to the stack in such a way that we always get the next node value at the top of the stack.

**Approach:**

The algorithm approach can be stated as:

* We first take an explicit stack data structure and set an infinite loop.
* In every iteration we check whether our current node is pointing to NULL or not.
* If it is not pointing to null, we simply push the current value to the stack and move the current node to its left child.
* If it is pointing to NULL, we first check whether the stack is empty or not. If the stack is empty, it means that we have traversed the tree and we break out of the loop.
* If the stack is not empty, we pop the top value of the stack, print it and move the current node to its right child.

**Stack is a Last-In-First-Out (LIFO)** data structure, therefore when we encounter a node, we simply push it to the stack and try to find nodes on its left.

When the current node points to NULL, it means that there is nothing left to traverse and we should move to the parent.

This parent is always placed at the top of the stack. If the stack is empty, then we had already traversed the whole tree and should stop the execution.

**Code:**

import java.util.\*;

class Node {

int data;

Node left, right;

Node(int data) {

this.data = data;

left = null;

right = null;

}

}

class TUF {

static ArrayList < Integer > inOrderTrav(Node curr) {

ArrayList < Integer > inOrder = new ArrayList < > ();

Stack < Node > s = new Stack < > ();

while (true) {

if (curr != null) {

s.push(curr);

curr = curr.left;

} else {

if (s.isEmpty()) break;

curr = s.peek();

inOrder.add(curr.data);

s.pop();

curr = curr.right;

}

}

return inOrder;

}

public static void main(String args[]) {

Node root = new Node(1);

root.left = new Node(2);

root.right = new Node(3);

root.left.left = new Node(4);

root.left.right = new Node(5);

root.left.right.left = new Node(8);

root.right.left = new Node(6);

root.right.right = new Node(7);

root.right.right.left = new Node(9);

root.right.right.right = new Node(10);

ArrayList < Integer > inOrder;

inOrder = inOrderTrav(root);

System.out.println("The inOrder Traversal is : ");

for (int i = 0; i < inOrder.size(); i++) {

System.out.print(inOrder.get(i) + " ");

}

}

}

**Output:** The inOrder Traversal is : 4 2 8 5 1 6 3 9 7 10

**Time Complexity: O(N)**.

Reason: We are traversing N nodes and every node is visited exactly once.

**Space Complexity: O(N)**

Reason: In the worst case (a tree with just left children), the space complexity will be O(N).

**Solution 2: Recursive**

**Approach:**In inorder traversal, the tree is traversed in this way: **left,**root, **right**.

The algorithm approach can be stated as:

* We first recursively visit the left child and go on till we find a leaf node.
* Then we print that node value.
* Then we recursively visit the right child.
* If we encounter a node pointing to NULL, we simply return to its parent.

**Code:**

import java.util.\*;

class Node {

int data;

Node left, right;

Node(int data) {

this.data = data;

left = null;

right = null;

}

}

class TUF {

static void inOrderTrav(Node curr, ArrayList < Integer > inOrder) {

if (curr == null)

return;

inOrderTrav(curr.left, inOrder);

inOrder.add(curr.data);

inOrderTrav(curr.right, inOrder);

}

public static void main(String args[]) {

Node root = new Node(1);

root.left = new Node(2);

root.right = new Node(3);

root.left.left = new Node(4);

root.left.right = new Node(5);

root.left.right.left = new Node(8);

root.right.left = new Node(6);

root.right.right = new Node(7);

root.right.right.left = new Node(9);

root.right.right.right = new Node(10);

ArrayList < Integer > inOrder = new ArrayList < > ();

inOrderTrav(root, inOrder);

System.out.println("The inOrder Traversal is : ");

for (int i = 0; i < inOrder.size(); i++) {

System.out.print(inOrder.get(i) + " ");

}

}

}

**Output:** The inOrder Traversal is : 4 2 8 5 1 6 3 9 7 10

**Time Complexity: O(N)**.

Reason: We are traversing N nodes and every node is visited exactly once.

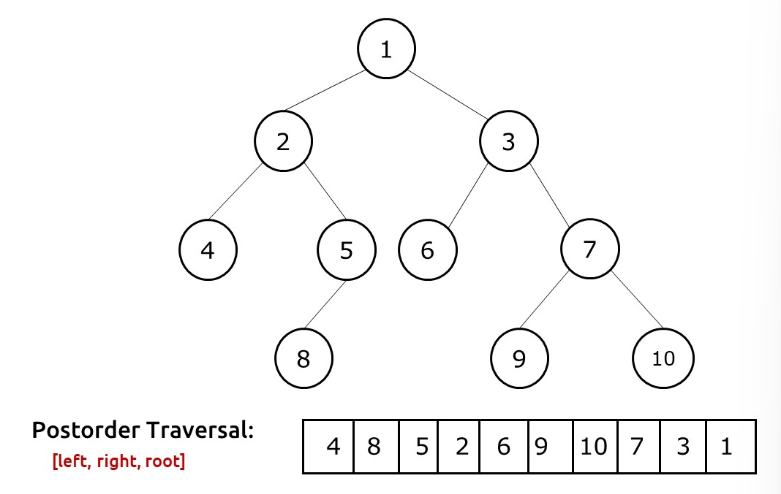
**Space Complexity: O(N)**

Reason: Space is needed for the recursion stack. In the worst case (skewed tree), space complexity can be O(N)

**03. Implementation of Postorder Traversal**

**Problem Statement:** Write a program for the postorder traversal of a binary tree.

**Example:**



**Solution 1: Recursive**

**Approach:**In postorder traversal, the tree is traversed in this way: **left, right**, **root**.

The algorithm approach can be stated as:

* We first recursively visit the left child and go on left till we find a node pointing to NULL.
* Then we return to its parent.
* Then we recursively visit the right child.
* After we have returned from the right child as well, only then will we print the  current node value.