# Binary Tree Concepts and Traversal Implementations

# 1. Concepts of Binary Tree Types

# I. Full Binary Tree A binary tree in which every node has either 0 or 2 children.

# II. Complete Binary Tree A binary tree where all levels are fully filled except possibly the last level, which is filled from left to right.

# III. Perfect Binary Tree A binary tree where all interior nodes have two children, and all leaves are at the same level.

# IV. Balanced Binary Tree A binary tree where the height of the left and right subtrees of every node differ by no more than 1.

# V. Degenerate Binary Tree A binary tree in which each parent node has only one child, forming a structure similar to a linked list.

# VI. Continuous Binary Tree A binary tree in which the absolute difference between values of adjacent nodes is 1.

# VII. Foldable Binary Tree A binary tree that can be folded into a mirror image along its center.

# VIII. Expression Tree A binary tree where each internal node represents an operator and each leaf node represents an operand.

# IX. Mirror Tree A binary tree that is the mirror image of another binary tree, with all left and right children swapped.

# Binary Tree Traversal Implementations in C++

## Binary Tree Node Structure

#include <iostream>  
#include <stack>  
using namespace std;  
  
struct TreeNode {  
 int data;  
 TreeNode\* left;  
 TreeNode\* right;  
 bool rightThread; // Only needed for threaded trees  
  
 TreeNode(int val) : data(val), left(nullptr), right(nullptr), rightThread(false) {}  
};

## 1. Recursive Traversals

### Preorder (Root, Left, Right)

void preorder(TreeNode\* node) {  
 if (node == nullptr) return;  
 cout << node->data << " ";  
 preorder(node->left);  
 preorder(node->right);  
}

### Inorder (Left, Root, Right)

void inorder(TreeNode\* node) {  
 if (node == nullptr) return;  
 inorder(node->left);  
 cout << node->data << " ";  
 inorder(node->right);  
}

### Postorder (Left, Right, Root)

void postorder(TreeNode\* node) {  
 if (node == nullptr) return;  
 postorder(node->left);  
 postorder(node->right);  
 cout << node->data << " ";  
}

## 2. Iterative Traversals

### Preorder (Root, Left, Right)

void iterativePreorder(TreeNode\* root) {  
 if (root == nullptr) return;  
 stack<TreeNode\*> stack;  
 stack.push(root);  
 while (!stack.empty()) {  
 TreeNode\* node = stack.top();  
 cout << node->data << " ";  
 stack.pop();  
 if (node->right) stack.push(node->right);  
 if (node->left) stack.push(node->left);  
 }  
}

### Inorder (Left, Root, Right)

void iterativeInorder(TreeNode\* root) {  
 stack<TreeNode\*> stack;  
 TreeNode\* curr = root;  
 while (curr != nullptr || !stack.empty()) {  
 while (curr != nullptr) {  
 stack.push(curr);  
 curr = curr->left;  
 }  
 curr = stack.top();  
 stack.pop();  
 cout << curr->data << " ";  
 curr = curr->right;  
 }  
}

### Postorder (Left, Right, Root)

void iterativePostorder(TreeNode\* root) {  
 if (root == nullptr) return;  
 stack<TreeNode\*> stack1, stack2;  
 stack1.push(root);  
 while (!stack1.empty()) {  
 TreeNode\* node = stack1.top();  
 stack2.push(node);  
 stack1.pop();  
 if (node->left) stack1.push(node->left);  
 if (node->right) stack1.push(node->right);  
 }  
 while (!stack2.empty()) {  
 cout << stack2.top()->data << " ";  
 stack2.pop();  
 }  
}

## 3. Threaded Binary Tree Traversal

### Inorder Threaded Traversal

void inorderThreaded(TreeNode\* root) {  
 TreeNode\* curr = leftmost(root);  
 while (curr != nullptr) {  
 cout << curr->data << " ";  
 if (curr->rightThread) curr = curr->right;  
 else curr = leftmost(curr->right);  
 }  
}  
  
TreeNode\* leftmost(TreeNode\* node) {  
 while (node != nullptr && node->left != nullptr)  
 node = node->left;  
 return node;  
}