Trees (C++)

This C++ file is an outstanding, progressively built tutorial on the **Binary Tree data structure**, all encapsulated within a modern C++ class. It's structured as a series of commented-out sections, each adding a new layer of functionality, which makes it an excellent way to learn how to build a complex data structure from the ground up.

The journey begins with a foundational Tree class that includes a custom-built Queue for **level-order creation** and the classic **recursive traversals** (Pre-order, In-order, Post-order). It then evolves by adding **iterative traversals** that use the C++ Standard Template Library (std::stack), showcasing how to solve the same problems without recursion.

The file's highlight is the generateFromTraversal method, a powerful algorithm that can **reconstruct an entire tree** from just its pre-order and in-order traversal sequences. This is a classic and important computer science problem.

Finally, the class is expanded with a suite of analytical functions. It demonstrates how to recursively calculate essential properties like the tree's **height**, the total **count** of nodes, the **sum** of all data, and the number of nodes with a specific **degree** (e.g., leaf nodes, nodes with two children). The final, active code represents the most complete and feature-rich version of the Tree class.

**Binary tree using Recursion**

This first section (commented out) builds the foundational **Tree class** using an Object-Oriented approach. It includes its own custom Queue class as a helper to build the tree and perform traversals. It focuses on the most intuitive, recursive methods for tree operations.

* class Tree { private: Node \*root; ... }; This encapsulates the entire tree logic. The root pointer is kept **private** to protect the tree's structure, and all operations are exposed through a safe **public** interface.
* void Tree::CreateTree() { ... } This method builds the tree using a **level-order creation** strategy. It uses a **Queue** to process nodes level by level, asking the user for the left and right children of each node. This is a very common and systematic way to construct a tree from user input.
* void Tree::Preorder(Node \*p) { ... } This is the classic **recursive Pre-order** traversal ("Root-Left-Right"). The code directly follows the definition: it prints the current node's data, then makes a recursive call on its left child, and finally makes a recursive call on its right child.
* void Tree::Levelorder(Node \*p) { ... } This function performs a **breadth-first traversal**. It uses a Queue to visit all nodes at the current depth level before moving on to the next level, printing them in the order they are visited.
* int Tree::Height(Node \*root) { ... return (x > y ? x : y) + 1; } This function recursively calculates the **height** of the tree. It finds the height of the left subtree (x) and the right subtree (y), and the height of the current node is 1 + the height of its taller child.

**Binary tree using iterative approach**

This section (commented out) enhances the Tree class by adding **iterative** (non-recursive) versions of the traversals. These methods manually use a **Stack** from the C++ Standard Library to mimic the function call stack that recursion uses behind the scenes.

* void Tree::iterativePreorder() { ... } This method performs a Pre-order traversal ("Root-Left-Right") using a stack. The logic is to print the current node's data, push it onto the stack, and then move as far left as possible. When it can't go left anymore, it pops from the stack to backtrack and explore the right subtree.
* void Tree::iterativeInorder() { ... } This performs an In-order traversal ("Left-Root-Right"). The key difference is the timing of the print statement. It pushes nodes and goes left, but it only **prints a node's data *after* it has been popped** from the stack. This ensures the entire left subtree is visited before the parent node is printed.
* void Tree::iterativePostorder() { ... } This is the most complex iterative traversal ("Left-Right-Root"). This implementation uses a clever trick: it stores a pointer-sized integer (intptr\_t) on the stack. When a node is popped, if the value is positive, it's pushed back as a **negative number** to signify that its left side is done, and the algorithm proceeds to the right. If a negative value is popped, it means both children have been visited, and it's finally time to print the node's data.

**Generating tree from traversal**

This section (commented out) adds a powerful and highly practical feature to the Tree class: the ability to **reconstruct the entire tree** from its pre-order and in-order traversal sequences.

* Node\* Tree::generateFromTraversal(...) { ... } This is the recursive function that builds the tree.
* Node\* node = new Node(preorder[preIndex++]); The core insight is that the **first element in a pre-order traversal is always the root** of the current subtree. A new node is created with this value.
* int splitIndex = searchInorder(inorder, inStart, inEnd, node->data); Next, this root's value is found in the **in-order traversal**. This is the critical step.
* node->lchild = generateFromTraversal(..., inStart, splitIndex - 1, ...); Everything to the **left** of the root in the in-order array belongs to the **left subtree**. The function makes a recursive call to build this left subtree.
* node->rchild = generateFromTraversal(..., splitIndex + 1, inEnd, ...); Everything to the **right** of the root in the in-order array belongs to the **right subtree**. Another recursive call builds this right subtree.

**Height, count and sum of a tree**

This section (commented out) adds several useful analytical functions to the Tree class, allowing you to calculate key properties of the tree recursively.

* int Tree::Count(Node \*p) { ... return x + y + 1; } This function recursively **counts the total number of nodes**. The logic is simple and elegant: the total count is the count of nodes in the left subtree (x) plus the count of nodes in the right subtree (y) plus one (for the current node).
* int Tree::Sum(Node \*p) { ... return x + y + p->data; } This function recursively finds the **sum of all data** in the tree. The logic is similar to Count: the total sum is the sum of the left subtree (x) plus the sum of the right subtree (y) plus the data of the current node.
* int Tree::deg2NodeCount(Node \*p) { ... } This function counts only the nodes that have **exactly two children** (degree 2).
  + if (p->lchild && p->rchild) { return x + y + 1; } If the current node has both a left and a right child, it adds 1 to the count of such nodes found in its subtrees. Otherwise, it just returns the sum from its children.

**Counting leaf nodes**

This is the final, active code block and represents the most feature-complete version of the Tree class. It adds functions to count nodes based on their specific degree (number of children).

* int Tree::leafNodeCount(Node \*p) { ... } This function recursively counts the **leaf nodes** (nodes with zero children).
  + if (p->lchild == nullptr && p->rchild == nullptr) { return x + y + 1; } The logic is to check if the current node is a leaf. If it is, it adds 1 to the total count of leaf nodes found in its (empty) subtrees.
* int Tree::deg1ordeg2NodeCount(Node \*p) { ... } This function counts all **internal nodes** (nodes with at least one child).
  + if (p->lchild != nullptr || p->rchild != nullptr) { return x + y + 1; } It adds 1 to the count if the current node has either a left child OR a right child (or both).
* int Tree::deg1NodeCount(Node \*p) { ... } This function counts only the nodes that have **exactly one child**.
  + if ((p->lchild != nullptr && p->rchild == nullptr) || (p->lchild == nullptr && p->rchild != nullptr)) { ... } This condition uses an **exclusive OR (XOR)** logic. It adds 1 to the count only if a node has a left child but no right child, OR a right child but no left child.