Trees

This C++ file is a fantastic, in-depth guide to the **Binary Tree** data structure, written in a procedural C-style. It's perfectly structured to teach two critical aspects of trees: how they are built and how they are traversed.

The file's core lesson is the comparison between solving problems **recursively** versus **iteratively**. It first demonstrates the classic, elegant recursive functions for Pre-order, In-order, and Post-order traversals. It then provides a masterclass in the iterative approach, which avoids recursion by explicitly using a **Stack** data structure.

A key highlight is that the file builds its own helper data structures (**Queue** and **Stack**) from scratch using arrays. The createTree function uses a Queue to build the tree in a **level-order** fashion, which is a standard and intuitive method. The iterative traversal functions then rely on the custom Stack to manage the node visitation order, perfectly illustrating the deep connection between these fundamental data structures.

**Creating Binary Tree (Commented Out Section)**

This first section (commented out) provides the foundational logic for creating a Binary Tree and performing the classic **recursive traversals**. It's the simplest and most direct way to understand tree algorithms.

* struct Node { ... }; struct Queue { ... }; These define the essential building blocks. The Node struct contains the data and pointers to the left and right children. The Queue struct is a helper data structure that is crucial for building the tree.
* void createTree() { ... } This function builds the tree using a **level-order construction** method.
  + It starts by creating the root node and adding it to a **Queue**.
  + It then enters a while loop that continues as long as the queue is not empty.
  + Inside the loop, it **dequeues** a parent node, asks the user to input its left and right children, and **enqueues** any new children that are created. This process ensures the tree is built one level at a time.
* void preOrder(struct Node \*p) { ... } This is the classic **recursive Pre-order** traversal. The logic directly follows the "Root-Left-Right" definition:
  1. **Print** the current node's data (p->data).
  2. Make a recursive call on the **left** child (preOrder(p->lChild)).
  3. Make a recursive call on the **right** child (preOrder(p->RChild)).

**Binary Tree using Iterative Approach**

This is the main, active part of the file. It focuses on performing tree traversals **without using recursion**. This is achieved by manually managing the traversal path with a **Stack**, which mimics the program's call stack that is used during recursion.

* struct Stack { ... }; This defines a custom **Stack** data structure from scratch using an array. The stack is the most important tool for iterative tree traversals, as it allows us to "remember" which nodes to return to after exploring a subtree.
* void iterativePreorder(struct Node \*p) { ... } This function performs a Pre-order traversal ("Root-Left-Right") using a stack.
  + while (p != NULL || !isStackEmpty(stk)) { ... } The loop continues as long as there's a current node to process OR there are nodes in the stack to backtrack to.
  + if (p != NULL) { printf("%d ", p->data); push(&stk, p); p = p->lchild; } This part handles the "Root-Left" logic. It **prints the data immediately** (Root), pushes the current node onto the stack to save its place, and then goes as far left as possible.
  + else { p = pop(&stk); p = p->rchild; } When it can't go left any further (p is NULL), it **pops a node from the stack** (backtracks) and tries to explore its right subtree.
* void iterativeInorder(struct Node \*p) { ... } This function performs an In-order traversal ("Left-Root-Right"). The logic is very similar to pre-order, but the print statement is moved.
  + if (p != NULL) { push(&stk, p); p = p->lchild; } This part handles the "Left" logic. It pushes the current node and goes as far left as possible, but it **does not print yet**.
  + else { p = pop(&stk); printf("%d ", p->data); p = p->rchild; } When it can't go left anymore, it **pops a node and prints its data now** (Root). This ensures all left children are visited before their parent. Then, it explores the right subtree.
* void iterativePostorder(struct Node \*p) { ... } This is the most complex iterative traversal ("Left-Right-Root"). It needs to know whether it's returning to a parent from its left child or its right child. This implementation uses a clever trick with pointers to keep track of this.
  + temp = (intptr\_t)pop(&stk); It pops a value from the stack, which could be a normal pointer or a modified one.
  + if (temp > 0) { push(&stk, (struct Node \*)-temp); ... } If the popped address is **positive**, it means we have just finished exploring the left subtree of this node. To mark this, we **push the address back onto the stack as a negative number**. This is a signal that the left side is done. Then, we proceed to explore the right subtree.
  + else { printf("%d ", ((struct Node \*)(-temp))->data); ... } If the popped address is **negative**, it means we have already explored both the left and right subtrees. We make the address positive again to access the node, **print its data** (Root), and set p to NULL to signal that we should continue popping from the stack.