Binary Tree Representation in C++

Introduction:

- A binary tree is a tree data structure in which each node has at most two children, referred to as the left child and the right child.
- Binary trees are used to implement various data structures such as binary search trees, heap trees, and expression trees.
- In C++, a binary tree can be represented using pointers to its nodes.

Node Structure:

- The first step in representing a binary tree in C++ is to define the structure of the node.
- The node structure should contain the data to be stored in the node, and two pointers, one to the left child and one to the right child.
- The node structure can be defined as follows:

```
 C++
struct Node {
  int data;
  Node* left;
  Node* right;
};
```


Creating a Binary Tree:

- Once the node structure is defined, a binary tree can be created by creating nodes and linking them together.
- The root node can be represented by a pointer to the first node in the tree.
- The following function can be used to create a new node:

```
 C++
Node* newNode(int data) {
  Node* node = new Node();
  node->data = data;
  node->left = NULL;
  node->right = NULL;
  return node;
}
```

```
};

void levelOrderTraversal(Node* root) {
    if (root == nullptr) {
        return;
    }

    queue<Node*> q;
    q.push(root);

    while (!q.empty()) {
        Node* node = q.front();
        q.pop();
        cout << node->data << " ";

        if (node->left != nullptr) {
            q.push(node->left);
        }

        if (node->right != nullptr) {
            q.push(node->right);
        }
    }
}
```


The **levelOrderTraversal** function takes a pointer to the root node of the binary tree and performs a level order traversal of the tree using a queue. The function first checks if the root node is null, and if so, it simply returns.

It then creates an empty queue and pushes the root node onto it. It then enters a loop that continues until the queue is empty. In each iteration of the loop, it pops a node from the front of the queue, prints its value, and then pushes its left and right children onto the queue (if they exist).

Java Code

Java

```
class Node {
    int data;
    Node left;
    Node right;

    Node(int data) {
        this.data = data;
        this.left = null;
        this.right = null;
```

```
        current = current.right;
    }
}
```

The **iterativeInorderTraversal** function takes a **Node** object representing the root node of the binary tree and performs an iterative inorder traversal of the tree using a stack. The function first checks if the root node is null, and if so, it simply returns.

It then creates an empty stack and initializes a **Node** object **current** to the root node. It enters a loop that continues until **current** is null and the stack is empty. In each iteration of the loop, it pushes all the left child nodes of the current node onto the stack until it reaches a null node. It then pops a node from the top of the stack, prints its value, and then sets **current** to its right child. The loop then repeats until all the nodes of the binary tree have been visited.

Both the C++ and Java implementations have a time complexity of O(n), where n is the number of nodes in the binary tree. The space complexity is O(h), where h is the height of the binary tree. This is because the maximum number of nodes that can be in the stack at any given time is equal to the height of the tree.