Assignment Report: Level-Order Traversal for BST

Problem:

Provide a non-recursive method for traversing a tree level by level for a given Binary Search Tree (BST). Hence, this method will take a BST as an input parameter

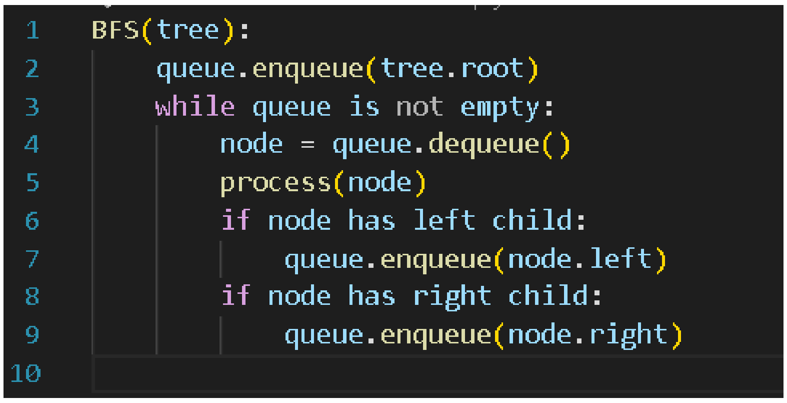
Approach:

1. Algorithm Design

BFS Algorithm

A level-order traversal algorithm using Breadth-First Search (BFS) was designed for a Binary Search Tree (BST). BFS is a breadth-first search algorithm that sequentially visits nodes at each level.

BFS Algorithm Pseudocode:



```
BFS(tree):
    queue.enqueue(tree.root)
    while queue is not empty:
        node = queue.dequeue()
        process(node)
        if node has left child:
            queue.enqueue(node.left)
        if node has right child:
            queue.enqueue(node.right)
```

Time Complexity Analysis for BFS Algorithm:

Processing each node: *O*(1)

Enqueuing and dequeuing each edge from a node *O*(Node's Degree).

c. Worst-Case Scenario:

In the worst-case scenario, BFS algorithm explores the entire graph, visiting each node and edge.

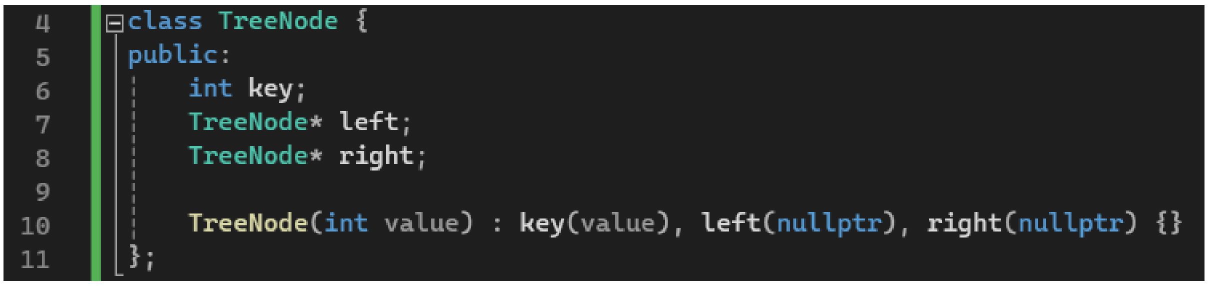
d. Complexity Calculation:

Time complexity of the BFS algorithm: $O(V+E)$.

2.Implementation of Binary Search Tree (BST) ADT in C++

Node Class

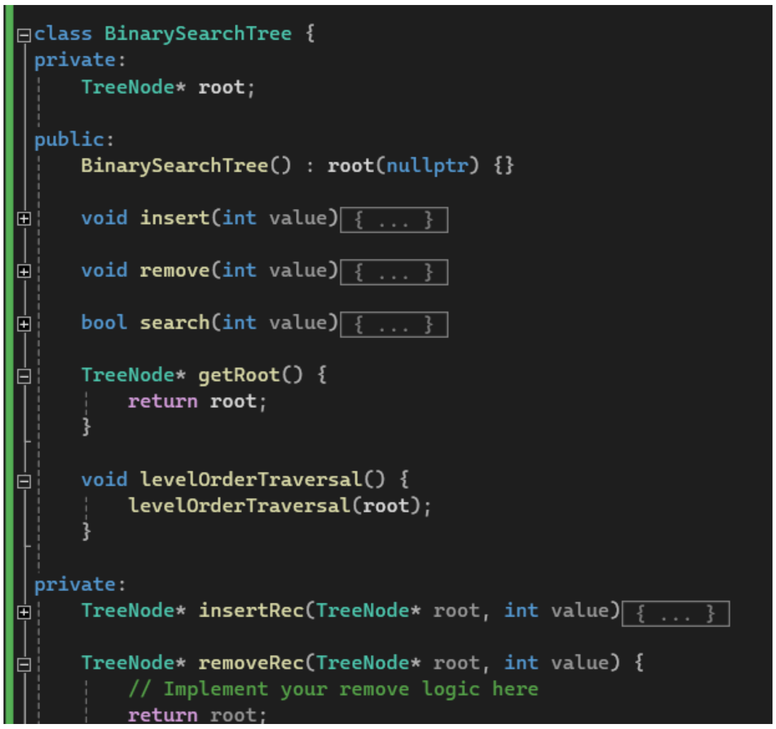
A TreeNode class was created to represent each node in the BST. This class includes a key value and left and right sub-trees.



```cpp
class TreeNode {
public:
    int key;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int value) : key(value), left(nullptr), right(nullptr) {}
};
```

BST Class

The **BinarySearchTree** class defines the general structure of a BST. It includes fundamental operations such as insertion, deletion, and search.

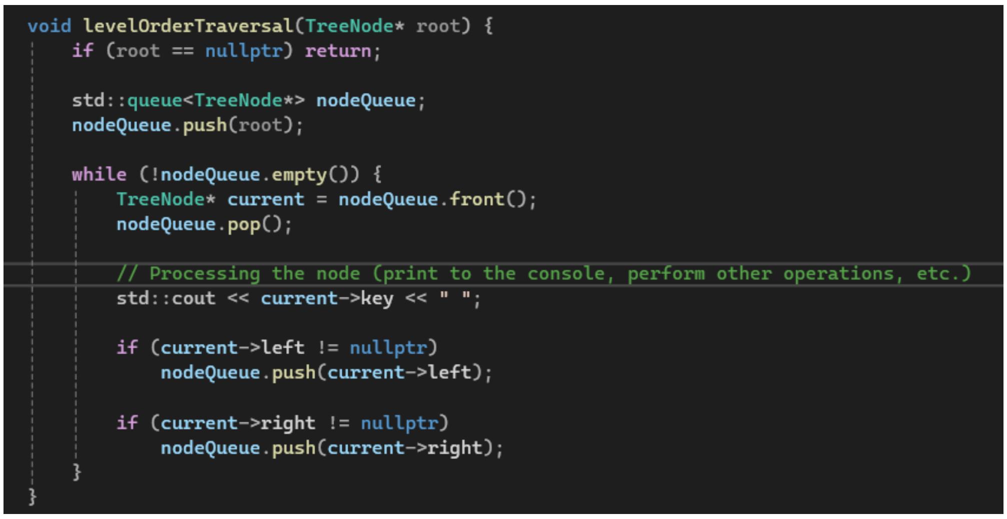


```cpp
class BinarySearchTree {
private:
    TreeNode* root;

public:
    BinarySearchTree() : root(nullptr) {}

    void insert(int value){ ... }

    void remove(int value){ ... }

    bool search(int value){ ... }

    TreeNode* getRoot() {
        return root;
    }

    void levelOrderTraversal() {
        levelOrderTraversal(root);
    }

private:
    TreeNode* insertRec(TreeNode* root, int value){ ... }

    TreeNode* removeRec(TreeNode* root, int value) {
        // Implement your remove logic here
        return root;
```

3. Level-Order Traversal Method

Implementation of BFS Algorithm

The BFS algorithm was implemented to traverse nodes level by level in the BST. A queue data structure was used for this purpose.

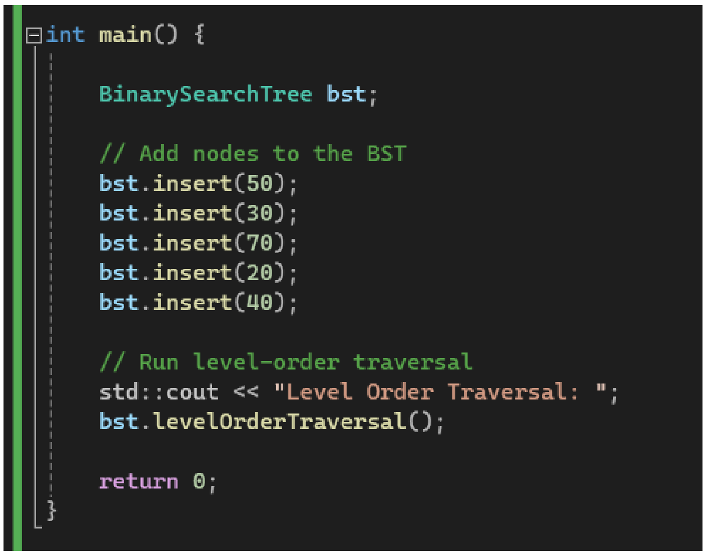


```
void levelOrderTraversal(TreeNode* root) {
    if (root == nullptr) return;

    std::queue<TreeNode*> nodeQueue;
    nodeQueue.push(root);

    while (!nodeQueue.empty()) {
        TreeNode* current = nodeQueue.front();
        nodeQueue.pop();

        // Processing the node (print to the console, perform other operations, etc.)
        std::cout << current->key << " ";

        if (current->left != nullptr)
            nodeQueue.push(current->left);

        if (current->right != nullptr)
            nodeQueue.push(current->right);
    }
}
```

4. Main Method

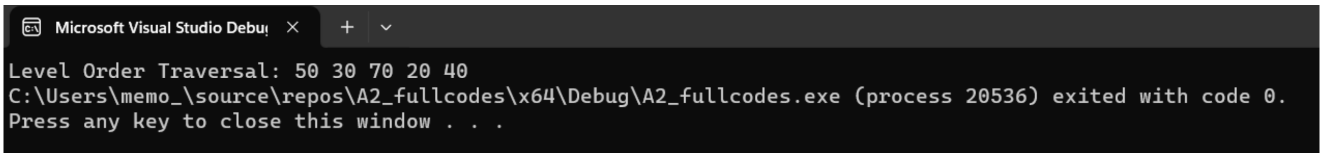
Running the Program

A main function was implemented to create a BST, add nodes, and demonstrate the level-order traversal method.



```
int main() {

    BinarySearchTree bst;

    // Add nodes to the BST
    bst.insert(50);
    bst.insert(30);
    bst.insert(70);
    bst.insert(20);
    bst.insert(40);

    // Run level-order traversal
    std::cout << "Level Order Traversal: ";
    bst.levelOrderTraversal();

    return 0;
}
```

Output:



```
Level Order Traversal: 50 30 70 20 40
C:\Users\memo_\source\repos\A2_fullcodes\x64\Debug\A2_fullcodes.exe (process 20536) exited with code 0.
Press any key to close this window . . .
```