Lab assignment 5

Implement Binary search tree and perform the following operations:

1. Insert
2. Delete
3. Display Tree levelwise

Code

```
#include <iostream>
#include <queue>

using namespace std;

// Represents a node in the binary search tree
struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int value) : data(value), left(nullptr),
right(nullptr) {}
};

// Represents a binary search tree
class BinarySearchTree {
```

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

    // Inserts a node with given data into the binary
search tree
    void insert(int data) {
        root = insertNode(root, data);
    }

    // Deletes a node with given data from the binary
search tree
    void deleteNode(int data) {
        root = deleteNodeRecursive(root, data);
    }

    // Displays the binary search tree level by level
    void displayLevelWise() {
        if (root == nullptr) {
            return;
        }

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

```
        while (!q.empty()) {
            int levelSize = q.size();
            for (int i = 0; i < levelSize; i++) {
                Node* node = q.front();
                q.pop();
                cout << node->data << " ";

                if (node->left) {
                    q.push(node->left);
                }
                if (node->right) {
                    q.push(node->right);
                }
            }
            cout << endl;
        }
    }

private:
    Node* root;

    // Recursively inserts a node into the binary
search tree
    Node* insertNode(Node* node, int data) {
```

```
        if (node == nullptr) {
            return new Node(data);
        }

        if (data < node->data) {
            node->left = insertNode(node->left, data);
        } else {
            node->right = insertNode(node->right, data);
        }

        return node;
    }

    // Recursively deletes a node from the binary search tree
    Node* deleteNodeRecursive(Node* node, int data) {
        if (node == nullptr) {
            return node;
        }

        if (data < node->data) {
            node->left = deleteNodeRecursive(node->left, data);
```

```
        } else if (data > node->data) {
            node->right =
deleteNodeRecursive(node->right, data);
        } else {
            // Case 1: No children
            if (node->left == nullptr && node->right ==
nullptr) {
                delete node;
                return nullptr;
            }
            // Case 2: One child
            else if (node->left == nullptr) {
                Node* temp = node->right;
                delete node;
                return temp;
            } else if (node->right == nullptr) {
                Node* temp = node->left;
                delete node;
                return temp;
            }
            // Case 3: Two children
            else {
                Node* temp = findMinNode(node->right);
                node->data = temp->data;
```

```
            node->right =
deleteNodeRecursive(node->right, temp->data);
        }
    }

    return node;
}

// Finds the node with the minimum value in the
given subtree
Node* findMinNode(Node* node) {
    while (node->left != nullptr) {
        node = node->left;
    }
    return node;
}
};

int main() {
    BinarySearchTree bst;

    // Inserting nodes
    bst.insert(8);
    bst.insert(3);
```

```
    bst.insert(10);
    bst.insert(1);
    bst.insert(6);
    bst.insert(14);
    bst.insert(4);
    bst.insert(7);
    bst.insert(13);

    cout << "Binary Search Tree (level-wise):" <<
endl;
    bst.displayLevelWise();

    // Deleting node 6
    bst.deleteNode(6);

    cout << "Binary Search Tree after deleting 6
(level-wise):" << endl;
    bst.displayLevelWise();

    return 0;
}
```

Output

```
Binary Search Tree (level-wise):
8
3 10
1 6 14
4 7 13
Binary Search Tree after deleting 6 (level-wise):
8
3 10
1 7 14
4 13
```