```
//LabSheet 6
//Ritik Sharma
//2301010342
#include <iostream>
#include <queue>
#include <vector>
#include <map>
#include <set>
#include <climits>

using namespace std;

// Node structure for trees
struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int value) : data(value), left(nullptr), right(nullptr) {}
};

// Binary Search Tree (BST) class
class BinarySearchTree {
public:
    TreeNode* root;

    BinarySearchTree() : root(nullptr) {}

    // Insert function
    TreeNode* insert(TreeNode* node, int value) {
        if (!node) return new TreeNode(value);
        if (value < node->data) node->left = insert(node->left, value);
        else node->right = insert(node->right, value);
        return node;
    }

    // In-order traversal
    void inOrder(TreeNode* node) {
        if (!node) return;
```

```
        inOrder(node->left);
        cout << node->data << " ";
        inOrder(node->right);
    }
};

// AVL Tree class with rotations
class AVLTree {
public:
    TreeNode* root;

    AVLTree() : root(nullptr) {}

    int height(TreeNode* node) {
        if (!node) return 0;
        return 1 + max(height(node->left), height(node->right));
    }

    int balanceFactor(TreeNode* node) {
        return node ? height(node->left) - height(node->right) : 0;
    }

    // Right rotation
    TreeNode* rotateRight(TreeNode* y) {
        TreeNode* x = y->left;
        TreeNode* T = x->right;
        x->right = y;
        y->left = T;
        return x;
    }

    // Left rotation
    TreeNode* rotateLeft(TreeNode* x) {
        TreeNode* y = x->right;
        TreeNode* T = y->left;
        y->left = x;
        x->right = T;
        return y;
```

```
    }

    // Insert with balancing
    TreeNode* insert(TreeNode* node, int value) {
        if (!node) return new TreeNode(value);
        if (value < node->data) node->left = insert(node->left, value);
        else if (value > node->data) node->right = insert(node->right,
value);
        else return node;

        int balance = balanceFactor(node);

        // Balancing cases
        if (balance > 1 && value < node->left->data) return
rotateRight(node);  // Left Left Case
        if (balance < -1 && value > node->right->data) return
rotateLeft(node); // Right Right Case
        if (balance > 1 && value > node->left->data) {                    //
Left Right Case
            node->left = rotateLeft(node->left);
            return rotateRight(node);
        }
        if (balance < -1 && value < node->right->data) {                    //
Right Left Case
            node->right = rotateRight(node->right);
            return rotateLeft(node);
        }
        return node;
    }
};

// Graph class with BFS, DFS, Dijkstra, and Prim's algorithms
class Graph {
    int V; // Number of vertices
    map<int, vector<pair<int, int>>> adjList; // adjacency list with
weights

public:
```

```
Graph(int vertices) : V(vertices) {}

void addEdge(int u, int v, int weight = 1) {
    adjList[u].emplace_back(v, weight);
    adjList[v].emplace_back(u, weight); // For undirected graph
}

// BFS traversal
void BFS(int start) {
    vector<bool> visited(V, false);
    queue<int> q;
    visited[start] = true;
    q.push(start);

    while (!q.empty()) {
        int u = q.front();
        q.pop();
        cout << u << " ";
        for (auto &[v, w] : adjList[u]) {
            if (!visited[v]) {
                visited[v] = true;
                q.push(v);
            }
        }
    }
}

// DFS traversal
void DFSUtil(int u, vector<bool> &visited) {
    visited[u] = true;
    cout << u << " ";
    for (auto &[v, w] : adjList[u]) {
        if (!visited[v]) {
            DFSUtil(v, visited);
        }
    }
}
```

```
void DFS(int start) {
    vector<bool> visited(V, false);
    DFSUtil(start, visited);
}

// Dijkstra's shortest path algorithm
void dijkstra(int src) {
    vector<int> dist(V, INT_MAX);
    set<pair<int, int>> s; // pair of (distance, vertex)
    dist[src] = 0;
    s.insert({0, src});

    while (!s.empty()) {
        int u = s.begin()->second;
        s.erase(s.begin());

        for (auto &[v, weight] : adjList[u]) {
            if (dist[u] + weight < dist[v]) {
                s.erase({dist[v], v});
                dist[v] = dist[u] + weight;
                s.insert({dist[v], v});
            }
        }
    }

    cout << "Dijkstra's shortest path from " << src << ":\n";
    for (int i = 0; i < V; ++i) {
        cout << "Distance to " << i << " is " << dist[i] << "\n";
    }
}

// Prim's algorithm for Minimum Spanning Tree
void primMST() {
    vector<int> key(V, INT_MAX), parent(V, -1);
    set<pair<int, int>> s;
    key[0] = 0;
    s.insert({0, 0});
```

```
        while (!s.empty()) {
            int u = s.begin()->second;
            s.erase(s.begin());

            for (auto &[v, weight] : adjList[u]) {
                if (weight < key[v]) {
                    s.erase({key[v], v});
                    key[v] = weight;
                    parent[v] = u;
                    s.insert({key[v], v});
                }
            }
        }

        cout << "Prim's Minimum Spanning Tree:\n";
        for (int i = 1; i < V; ++i) {
            cout << parent[i] << " - " << i << "\n";
        }
    }
};

// Main function to demonstrate all functionalities
int main() {
    // Binary Search Tree Example
    BinarySearchTree bst;
    bst.root = bst.insert(bst.root, 10);
    bst.insert(bst.root, 5);
    bst.insert(bst.root, 15);
    cout << "BST In-Order Traversal: ";
    bst.inOrder(bst.root);
    cout << "\n";

    // AVL Tree Example
    AVLTree avl;
    avl.root = avl.insert(avl.root, 30);
    avl.insert(avl.root, 20);
    avl.insert(avl.root, 10);
    cout << "AVL Root After Balancing: " << avl.root->data << "\n";
```

```
    // Graph Example
    Graph g(4);
    g.addEdge(0, 1, 1);
    g.addEdge(1, 2, 2);
    g.addEdge(0, 2, 4);
    cout << "BFS Traversal starting from node 0: ";
    g.BFS(0);
    cout << "\nDFS Traversal starting from node 0: ";
    g.DFS(0);
    cout << "\n";

    g.dijkstra(0);
    g.primMST();

    return 0;
}
```