Tarun Thomas Tom

B230589EC

Week 4

Data Structures

# Codes

[taruntom1/DS-assignments](https://github.com/taruntom1/DS-assignments)

# Q1

Take a binary tree and do tree traversals

## Code

#include <iostream>

#include <stack>

#include <vector>

#include <queue>

using namespace std;

struct Node

{

    int data;

    Node\* left;

    Node\* right;

    Node(int val, Node\* l = nullptr, Node\* r = nullptr) : data(val), left(l), right(r) {}

};

class BinaryTree

{

    Node\* root;

public:

    BinaryTree(int data)

    {

        root = new Node(data);

    }

    void insert(int val)

    {

        std::queue<Node\*> q;

        q.push(root);

        Node\* temp;

        while (!q.empty())

        {

            temp = q.front();

            q.pop();

            if (temp->left != nullptr)

                q.push(temp->left);

            else

            {

                temp->left = new Node(val);

                break;

            }

            if (temp->right != nullptr)

                q.push(temp->right);

            else

            {

                temp->right = new Node(val);

                break;

            }

        }

    }

    std::vector<int> BFS\_Traversal()

    {

        std::queue<Node\*> q;

        std::vector<int> result;

        q.push(root);

        Node\* current;

        while (!q.empty())

        {

            current = q.front();

            q.pop();

            result.push\_back(current->data);

            if (current->left != nullptr)

                q.push(current->left);

            if (current->right != nullptr)

                q.push(current->right);

        }

        return result;

    }

    std::vector<int> DFS\_preorder()

    {

        std::stack<Node\*> s;

        std::vector<int> result;

        s.push(root);

        Node\* current;

        while (!s.empty())

        {

            current = s.top();

            s.pop();

            result.push\_back(current->data);

            if (current->right != nullptr)

                s.push(current->right);

            if (current->left != nullptr)

                s.push(current->left);

        }

        return result;

    }

    std::vector<int> DFS\_inorder()

    {

        std::stack<Node\*> s;

        std::vector<int> result;

        Node\* current = root;

        while (!s.empty() || current)

        {

            while (current)

            {

                s.push(current);

                current = current->left;

            }

            current = s.top();

            s.pop();

            result.push\_back(current->data);

            current = current->right;

        }

        return result;

    }

    std::vector<int> DFS\_postorder()

    {

        std::stack<Node\*> s1, s2;

        std::vector<int> result;

        Node\* current;

        s1.push(root);

        while (!s1.empty())

        {

            current = s1.top();

            s1.pop();

            s2.push(current);

            if (current->left)

                s1.push(current->left);

            if (current->right)

                s1.push(current->right);

        }

        while (!s2.empty())

        {

            result.push\_back(s2.top()->data);

            s2.pop();

        }

        return result;

    }

};

int main()

{

    BinaryTree tree(5);

    tree.insert(50);

    tree.insert(30);

    tree.insert(70);

    tree.insert(20);

    tree.insert(40);

    tree.insert(60);

    tree.insert(80);

    tree.insert(10);

    tree.insert(25);

    tree.insert(35);

    tree.insert(45);

    tree.insert(55);

    tree.insert(65);

    tree.insert(75);

    tree.insert(85);

    std::cout << "BFS Traversal: ";

    std::vector<int> bfs\_traversal = tree.BFS\_Traversal();

    for (int val : bfs\_traversal)

        std::cout << val << " ";

    std::cout << "\nDFS Preorder Traversal: ";

    std::vector<int> dfs\_traversal = tree.DFS\_preorder();

    for (int val : dfs\_traversal)

        std::cout << val << " ";

    std::cout << "\nDFS Inorder Traversal: ";

    dfs\_traversal = tree.DFS\_inorder();

    for (int val : dfs\_traversal)

        std::cout << val << " ";

    std::cout << "\nDFS Postorder Traversal: ";

    dfs\_traversal = tree.DFS\_postorder();

    for (int val : dfs\_traversal)

        std::cout << val << " ";

    return 0;

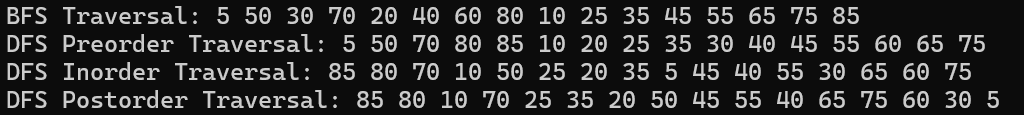
}

## Tree

A diagram of a company structure

AI-generated content may be incorrect.

## Output



# Q2

2. Construct a BST and do the following on it:

1. Insert
2. Delete
3. Search
4. Max
5. Min
6. Predecessor
7. Successor

## Code

#include <iostream>

#include <stack>

#include <queue>

#include <vector>

#include <map>

#include <cmath>

#include <iomanip>

#include <string>

struct Node

{

    int data;

    Node \*left;

    Node \*right;

    Node(int val, Node \*l = nullptr, Node \*r = nullptr) : data(val), left(l), right(r) {}

};

class BST

{

private:

    Node \*root;

    void treeToMatrixRec(Node \*root, int row, int col, int height,

                         std::vector<std::vector<std::string>> &ans)

    {

        if (!root)

        {

            return;

        }

        // Calculate offset for child positions

        int offset = pow(2, height - row - 1);

        // Traverse the left subtree

        if (root->left)

        {

            treeToMatrixRec(root->left, row + 1, col - offset, height, ans);

        }

        // Place the current node's value in the matrix

        ans[row][col] = std::to\_string(root->data);

        // Traverse the right subtree

        if (root->right)

        {

            treeToMatrixRec(root->right, row + 1, col + offset, height, ans);

        }

    }

    // Function to convert the binary tree to a 2D matrix

    std::vector<std::vector<std::string>> treeToMatrix(Node \*root)

    {

        // Find the height of the tree

        int height = findHeight(root);

        // Rows are height + 1; columns are 2^(height+1) - 1

        int rows = height + 1;

        int cols = pow(2, height + 1) - 1;

        // Initialize 2D matrix with empty strings

        std::vector<std::vector<std::string>> ans(rows, std::vector<std::string>(cols, ""));

        // Populate the matrix using inorder traversal

        treeToMatrixRec(root, 0, (cols - 1) / 2, height, ans);

        return ans;

    }

    void print2DArray(std::vector<std::vector<std::string>> &arr)

    {

        for (auto &row : arr)

        {

            for (auto &cell : row)

            {

                if (cell.empty())

                {

                    std::cout << "  ";

                }

                else

                {

                    std::cout << cell;

                }

            }

            std::cout << std::endl;

        }

    }

public:

    BST() : root(nullptr) {}

    void insert(int val)

    {

        Node \*new\_node = new Node(val);

        if (root == nullptr)

            root = new\_node;

        else

        {

            std::queue<Node \*> q;

            q.push(root);

            Node \*temp;

            while (!q.empty())

            {

                temp = q.front();

                q.pop();

                if (temp->data > val)

                {

                    if (temp->left == nullptr)

                        temp->left = new\_node;

                    else

                        q.push(temp->left);

                }

                else

                {

                    if (temp->right == nullptr)

                        temp->right = new\_node;

                    else

                        q.push(temp->right);

                }

            }

        }

    }

    void deleteNode(Node \*node)

    {

        if (node->right)

        {

            Node \*parent;

            Node \*min\_node = successorLoc(node, &parent);

            node->data = min\_node->data;

            delete min\_node;

            parent->left = nullptr;

        }

        else if (node->left)

        {

            Node \*temp = node->left;

            delete node;

            node = temp;

        }

        else

        {

            delete node;

        }

    }

    std::vector<int> Traversal()

    {

        std::queue<Node \*> q;

        std::vector<int> result;

        q.push(root);

        Node \*temp;

        while (!q.empty())

        {

            temp = q.front();

            q.pop();

            result.push\_back(temp->data);

            if (temp->left != nullptr)

                q.push(temp->left);

            if (temp->right != nullptr)

                q.push(temp->right);

        }

        return result;

    }

    Node \*search(int key, Node \*node = nullptr)

    {

        if (node == nullptr)

            node = root;

        while (node != nullptr && key != node->data)

        {

            if (key < node->data)

                node = node->left;

            else

                node = node->right;

        }

        return node;

    }

    Node \*minLoc(Node \*node, Node \*\*parent = nullptr)

    {

        while (node->left->left)

        {

            node = node->left;

        }

        if (parent)

            \*parent = node;

        return node->left;

    }

    Node \*maxLoc(Node \*node, Node \*\*parent = nullptr)

    {

        while (node->right->right)

        {

            node = node->right;

        }

        if (parent)

            \*parent = node;

        return node->right;

    }

    int min(Node \*node = nullptr)

    {

        if (node == nullptr)

            node = root;

        return minLoc(node)->data;

    }

    int max(Node \*node = nullptr)

    {

        if (node == nullptr)

            node = root;

        return maxLoc(node)->data;

    }

    Node \*predecessorLoc(Node \*node, Node \*\*parent = nullptr)

    {

        Node \*curr = node->left;

        return maxLoc(curr, parent);

    }

    Node \*successorLoc(Node \*node, Node \*\*parent = nullptr)

    {

        Node \*curr = node->right;

        return minLoc(curr, parent);

    }

    int predecessor(Node \*node)

    {

        if (node == nullptr)

            node = root;

        Node \*predecessor = predecessorLoc(node);

        return predecessor->data;

    }

    int successor(Node \*node)

    {

        if (node == nullptr)

            node = root;

        Node \*successor = successorLoc(node);

        return successor->data;

    }

    int findHeight(Node \*root)

    {

        if (!root)

        {

            return -1;

        }

        int leftHeight = findHeight(root->left);

        int rightHeight = findHeight(root->right);

        return std::max(leftHeight, rightHeight) + 1;

    }

    Node \*getRoot()

    {

        return root;

    }

    void printTree()

    {

        std::vector<std::vector<std::string>> result = treeToMatrix(root);

        print2DArray(result);

    }

};

int main()

{

    BST tree;

    // insert elements to ensure height of at least 4

    tree.insert(50);

    tree.insert(30);

    tree.insert(70);

    tree.insert(20);

    tree.insert(40);

    tree.insert(60);

    tree.insert(80);

    tree.insert(10);

    tree.insert(25);

    tree.insert(35);

    tree.insert(45);

    tree.insert(55);

    tree.insert(65);

    tree.insert(75);

    tree.insert(85);

    std::cout << "Inserted elements" << std::endl;

    std::cout << "Tree:" << std::endl;

    tree.printTree();

    std::cout << std::endl;

    // Delete an element

    int deleteKey = 30;

    tree.deleteNode(tree.search(deleteKey));

    std::cout << "Deleted " << deleteKey << std::endl;

    std::cout << "Searching for " << deleteKey << " after deletion: "

              << (tree.search(deleteKey) ? "Found" : "Not Found") << std::endl;

    std::cout << "Tree:" << std::endl;

    tree.printTree();

    std::cout << std::endl;

    // Search for an element

    int searchKey = 40;

    std::cout << "Searching for " << searchKey << ": "

              << (tree.search(searchKey) ? "Found" : "Not Found")

              << " Memory location :" << tree.search(searchKey) << std::endl

              << std::endl;

    // Find minimum and maximum

    std::cout << "Minimum value: " << tree.min() << std::endl;

    std::cout << "Maximum value: " << tree.max() << std::endl;

    // Find minimum and maximum under a subtree

    int subtreeKey = 70;

    std::cout << "Minimum value under subtree " << subtreeKey << ": "

              << tree.min(tree.search(subtreeKey)) << std::endl;

    std::cout << "Maximum value under subtree" << subtreeKey << ": "

              << tree.max(tree.search(subtreeKey)) << std::endl;

    // Find predecessor and successor

    int key = 50;

    std::cout << "Predecessor of " << key << ": "

              << tree.predecessor(tree.search(key)) << std::endl;

    std::cout << "Successor of " << key << ": "

              << tree.successor(tree.search(key)) << std::endl;

    key = 70;

    std::cout << "Predecessor of " << key << ": "

              << tree.predecessor(tree.search(key)) << std::endl;

    std::cout << "Successor of " << key << ": "

              << tree.successor(tree.search(key)) << std::endl;

    return 0;

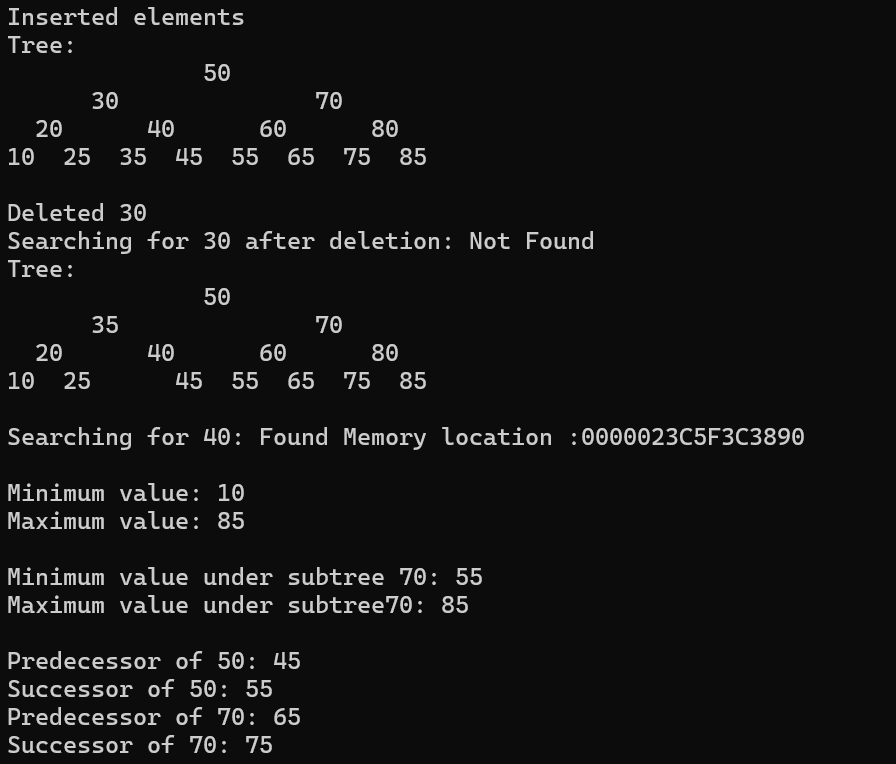
}

## Tree

A diagram of a diagram

AI-generated content may be incorrect.

## Output



# Q3

Implement the following graph algorithms

1. BFS
2. DFS
3. PRIM’S ALGORITHM
4. KRUSKAL’S ALGORITHM
5. DJIKSTRA’S ALGORITHM

## Code

#include <iostream>

#include <queue>

#include <stack>

#include <vector>

#include <set>

#include <unordered\_set>

#include <unordered\_map>

#include <functional>

class Node;

struct edge\_to\_cost

{

    Node \*to;

    int cost;

};

class Node

{

public:

    int data;

    std::vector<edge\_to\_cost> edges;

    Node(int val)

    {

        data = val;

    }

    void newNeighbour(int data, int cost)

    {

        edges.push\_back(edge\_to\_cost{new Node(data), cost});

    }

    void linkNeighbours(const std::vector<Node \*> &newNeighbours, const std::vector<int> &newCosts)

    {

        if (newNeighbours.size() != newCosts.size())

        {

            std::cerr << "Error: Neighbours and costs must have the same size!" << std::endl;

            return;

        }

        for (size\_t i = 0; i < newNeighbours.size(); ++i)

        {

            edges.push\_back(edge\_to\_cost{newNeighbours[i], newCosts[i]});

            newNeighbours[i]->edges.push\_back({this, newCosts[i]});

        }

    }

};

template <typename Func, typename... Args>

void bfs(Node \*start, Func func, Args &&...args)

{

    std::unordered\_set<Node \*> visited;

    std::queue<Node \*> q;

    q.push(start);

    visited.insert(start);

    while (!q.empty())

    {

        auto curr = q.front();

        q.pop();

        func(curr, std::forward<Args>(args)...);

        for (auto edge : curr->edges)

        {

            if (visited.find(edge.to) == visited.end())

            {

                q.push(edge.to);

                visited.insert(edge.to);

            }

        }

    }

}

std::vector<int> dfs(Node \*start)

{

    std::set<Node \*> visited;

    std::stack<Node \*> s;

    std::vector<int> result;

    s.push(start);

    visited.insert(start);

    while (!s.empty())

    {

        auto curr = s.top();

        s.pop();

        result.push\_back(curr->data);

        for (auto edge : curr->edges)

        {

            if (visited.find(edge.to) == visited.end())

            {

                s.push(edge.to);

                visited.insert(edge.to);

            }

        }

    }

    return result;

}

struct Edge

{

    Node \*from;

    Node \*to;

    int cost;

};

struct compare

{

    bool operator()(const Edge &a, const Edge &b)

    {

        return a.cost > b.cost;

    }

};

std::vector<Edge> prims(Node \*start)

{

    std::priority\_queue<Edge, std::vector<Edge>, compare> pq;

    std::set<Node \*> visited;

    std::vector<Edge> result;

    for (const auto &edge : start->edges)

    {

        pq.push(Edge{start, edge.to, edge.cost});

    }

    visited.insert(start);

    while (!pq.empty())

    {

        auto curr = pq.top();

        pq.pop();

        if (visited.find(curr.to) != visited.end())

            continue;

        visited.insert(curr.to);

        result.push\_back(curr);

        for (auto edge : curr.to->edges)

        {

            pq.push(Edge{curr.to, edge.to, edge.cost});

        }

    }

    return result;

}

void getEdges(Node \*node, std::priority\_queue<Edge, std::vector<Edge>, compare> &pq)

{

    if (node == nullptr)

        return;

    for (auto edge : node->edges)

        pq.push(Edge{node, edge.to, edge.cost});

}

void getNodes(Node \*node, std::vector<Node \*> &nodes)

{

    if (node != nullptr)

    {

        nodes.push\_back(node);

    }

}

class DSU

{

    std::unordered\_map<Node \*, Node \*> parent;

    std::unordered\_map<Node \*, int> rank;

public:

    void makeSet(Node \*node)

    {

        parent[node] = node;

        rank[node] = 0;

    }

    Node \*find(Node \*node)

    {

        if (parent[node] != node)

        {

            parent[node] = find(parent[node]);

        }

        return parent[node];

    }

    void unionSets(Node \*x, Node \*y)

    {

        Node \*rootX = find(x);

        Node \*rootY = find(y);

        if (rootX != rootY)

        {

            if (rank[rootX] > rank[rootY])

            {

                parent[rootY] = rootX;

            }

            else if (rank[rootY] > rank[rootX])

            {

                parent[rootX] = rootY;

            }

            else

            {

                parent[rootY] = rootX;

                rank[rootX]++;

            }

        }

    }

};

std::vector<Edge> kruskalsDSU(Node \*start)

{

    std::priority\_queue<Edge, std::vector<Edge>, compare> edges;

    DSU dsu;

    std::vector<Node \*> nodes;

    bfs(start, getNodes, nodes);

    for (auto node : nodes)

    {

        dsu.makeSet(node);

    }

    bfs(start, getEdges, edges);

    std::vector<Edge> mst\_edges;

    while (!edges.empty() && mst\_edges.size() < (nodes.size() - 1))

    {

        Edge curr = edges.top();

        edges.pop();

        if (dsu.find(curr.from) != dsu.find(curr.to))

        {

            mst\_edges.push\_back(curr);

            dsu.unionSets(curr.from, curr.to);

        }

    }

    return mst\_edges;

}

std::vector<Edge> kruskals(Node \*start)

{

    std::priority\_queue<Edge, std::vector<Edge>, compare> edges;

    std::vector<std::unordered\_set<Node \*>> visited\_groups;

    std::vector<Edge> result;

    bfs(start, getEdges, edges);

    while (!edges.empty())

    {

        Edge curr = edges.top();

        edges.pop();

        int i = 0;

        int edge\_from\_group = -1;

        int edge\_to\_group = -1;

        for (auto group : visited\_groups)

        {

            if (group.find(curr.from) != group.end())

            {

                edge\_from\_group = i;

            }

            if (group.find(curr.to) != group.end())

            {

                edge\_to\_group = i;

            }

            if (edge\_from\_group != -1 && edge\_to\_group != -1)

                break;

            i++;

        }

        if (edge\_from\_group == -1 && edge\_to\_group == -1)

        {

            visited\_groups.push\_back(std::unordered\_set<Node \*>{curr.from, curr.to});

            result.push\_back(curr);

        }

        else if (edge\_from\_group == -1)

        {

            visited\_groups[edge\_to\_group].insert(curr.from);

            result.push\_back(curr);

        }

        else if (edge\_to\_group == -1)

        {

            visited\_groups[edge\_from\_group].insert(curr.to);

            result.push\_back(curr);

        }

        else if (edge\_from\_group != edge\_to\_group)

        {

            visited\_groups[edge\_from\_group].insert(visited\_groups[edge\_to\_group].begin(), visited\_groups[edge\_to\_group].end());

            visited\_groups.erase(visited\_groups.begin() + edge\_to\_group);

            result.push\_back(curr);

        }

    }

    return result;

}

struct dijkstras\_pq\_element

{

    Node \*curr;

    Node \*prev;

    int dist;

    bool operator>(const dijkstras\_pq\_element &other) const

    {

        return dist > other.dist;

    }

};

std::unordered\_map<Node \*, std::pair<std::vector<Node \*>, int>> dijkstras(Node \*start)

{

    std::unordered\_map<Node \*, std::pair<int, Node \*>> table;

    std::unordered\_map<Node \*, std::pair<std::vector<Node \*>, int>> paths;

    std::priority\_queue<dijkstras\_pq\_element, std::vector<dijkstras\_pq\_element>, std::greater<>> pq;

    pq.push({start, nullptr, 0});

    while (!pq.empty())

    {

        dijkstras\_pq\_element curr = pq.top();

        pq.pop();

        // Skip if node has already been visited

        if (table.find(curr.curr) != table.end())

            continue;

        table[curr.curr] = {curr.dist, curr.prev};

        for (auto edge : curr.curr->edges)

        {

            int new\_dist = curr.dist + edge.cost;

            if (table.find(edge.to) == table.end() || new\_dist < table[edge.to].first)

            {

                pq.push({edge.to, curr.curr, new\_dist});

            }

        }

    }

    // Reconstruct paths

    for (const auto &[node, info] : table)

    {

        int dist = info.first;

        Node \*prev = info.second;

        std::vector<Node \*> path;

        Node \*current = node;

        while (prev)

        {

            path.push\_back(prev);

            prev = table[prev].second;

        }

        std::reverse(path.begin(), path.end());

        paths[node] = {path, dist};

    }

    return paths;

}

int main()

{

    // Create a vector to hold pointers to Node objects

    std::vector<Node \*> nodes;

    // Initialize nodes with values from 0 to 10

    for (int i = 0; i < 11; i++)

    {

        nodes.push\_back(new Node(i));

    }

    // Establish links between nodes with corresponding edge weights

    nodes[0]->linkNeighbours({nodes[1], nodes[2], nodes[4]}, {5, 3, 7});

    nodes[1]->linkNeighbours({}, {}); // No outgoing edges

    nodes[2]->linkNeighbours({nodes[1], nodes[3], nodes[5], nodes[6]}, {4, 6, 2, 8});

    nodes[3]->linkNeighbours({}, {}); // No outgoing edges

    nodes[4]->linkNeighbours({nodes[7]}, {4});

    nodes[5]->linkNeighbours({nodes[6]}, {2});

    nodes[6]->linkNeighbours({nodes[8], nodes[10]}, {5, 7});

    nodes[7]->linkNeighbours({nodes[9]}, {4});

    nodes[8]->linkNeighbours({nodes[10]}, {3});

    nodes[9]->linkNeighbours({nodes[10]}, {1});

    nodes[10]->linkNeighbours({}, {}); // No outgoing edges

    // Perform Breadth-First Search (BFS) and store result in bfsresult vector

    std::vector<Node \*> bfsresult;

    bfs(nodes[0], getNodes, bfsresult);

    std::cout << "BFS Traversal Order:" << std::endl;

    // Print BFS traversal order

    for (auto i : bfsresult)

    {

        std::cout << i->data << " ";

    }

    std::cout << std::endl

              << std::endl;

    // Perform Depth-First Search (DFS) and store result in dfsresult vector

    std::vector<int> dfsresult = dfs(nodes[0]);

    std::cout << "DFS Traversal Order:" << std::endl;

    // Print DFS traversal order

    for (auto i : dfsresult)

    {

        std::cout << i << " ";

    }

    std::cout << std::endl

              << std::endl;

    // Print the edges of the graph

    std::cout << "Edges" << std::endl;

    std::priority\_queue<Edge, std::vector<Edge>, compare> pq;

    bfs(nodes[0], getEdges, pq);

    while (!pq.empty())

    {

        std::cout << pq.top().from->data << "-->" << pq.top().to->data << " "

                  << pq.top().cost << std::endl;

        pq.pop();

    }

    // Perform Prim's algorithm to find the Minimum Spanning Tree (MST)

    std::cout << "Prim's:" << std::endl;

    std::vector<Edge> primsResult = prims(nodes[0]);

    int total\_edge\_weight = 0;

    for (auto e : primsResult)

    {

        if (e.from)

        {

            std::cout << e.from->data << "-->" << e.to->data << " "

                      << e.cost << std::endl;

            total\_edge\_weight += e.cost;

        }

    }

    std::cout << "total edge weight: " << total\_edge\_weight << std::endl

              << std::endl;

    std::cout << "Kruskal's:" << std::endl;

    std::vector<Edge> kruskalsResult = kruskals(nodes[0]);

    total\_edge\_weight = 0;

    for (auto e : kruskalsResult)

    {

        if (e.from)

        {

            std::cout << e.from->data << "-->" << e.to->data << " "

                      << e.cost << std::endl;

            total\_edge\_weight += e.cost;

        }

    }

    std::cout << "total edge weight: " << total\_edge\_weight << std::endl

              << std::endl;

    std::cout << "Kruskal's DSU:" << std::endl;

    std::vector<Edge> kruskalsResultDSU = kruskalsDSU(nodes[0]);

    total\_edge\_weight = 0;

    for (auto e : kruskalsResultDSU)

    {

        if (e.from)

        {

            std::cout << e.from->data << "-->" << e.to->data << " " << e.cost

                      << std::endl;

            total\_edge\_weight += e.cost;

        }

    }

    std::cout << "total edge weight: " << total\_edge\_weight << std::endl

              << std::endl;

    std::cout << "Dijkstra's Algorithm:" << std::endl;

    std::unordered\_map<Node \*, std::pair<std::vector<Node \*>, int>> dijkstra\_result = dijkstras(nodes[0]);

    for (auto element : dijkstra\_result)

    {

        std::cout << element.first->data << " -> ";

        for (auto node : element.second.first)

        {

            std::cout << node->data << " -> ";

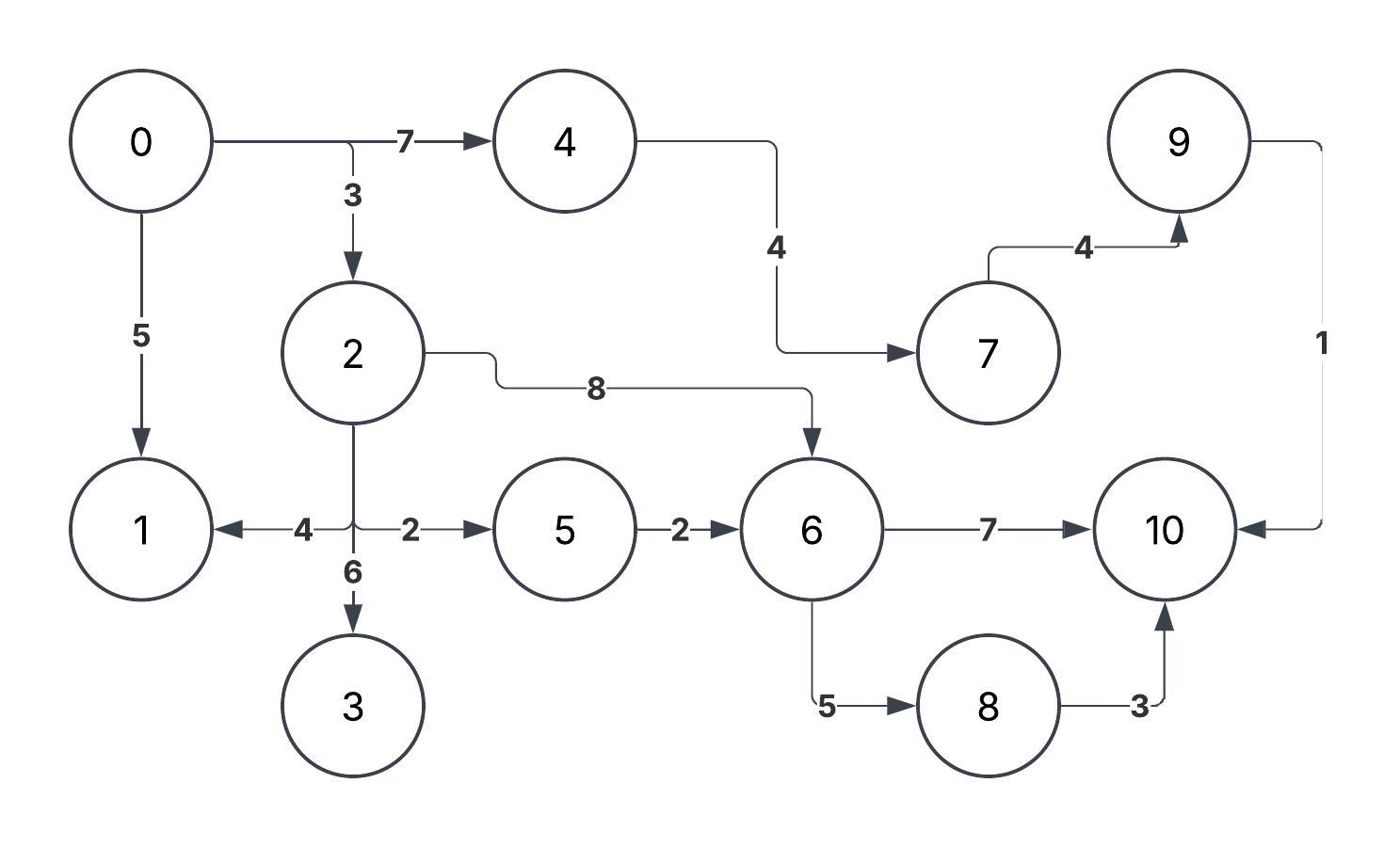
        }

        std::cout << "cost: " << element.second.second << std::endl;

    }

}

## Graph



Ignore the arrows, the tree is undirected

## Output

A black background with white text

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.