#include<bits/stdc++.h>

using namespace std;

#define V 5

void primMST(vector<pair<int, int>> adj[]) {

priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

vector<int> key(V, INT\_MAX);

vector<int> parent(V, -1);

vector<bool> inMST(V, false);

int src = 0;

pq.push(make\_pair(0, src));

key[src] = 0;

while (!pq.empty()) {

int u = pq.top().second;

pq.pop();

inMST[u] = true;

for (auto x : adj[u]) {

int v = x.first;

int weight = x.second;

if (!inMST[v] && key[v] > weight) {

key[v] = weight;

pq.push(make\_pair(key[v], v));

parent[v] = u;

}

}

}

cout << "Edge \tWeight\n";

for (int i = 1; i < V; i++)

cout << parent[i] << " - " << i << "\t" << key[i] << "\n";

}

int main() {

vector<pair<int, int>> adj[V] = {

{{1, 2}, {3, 6}},

{{0, 2}, {2, 3}, {3, 8}, {4, 5}},

{{1, 3}, {4, 7}},

{{0, 6}, {1, 8}},

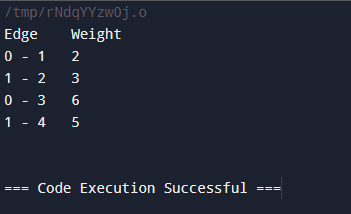
{{1, 5}, {2, 7}}

};

primMST(adj);

return 0;

}



#include<bits/stdc++.h>

using namespace std;

#define V 5

void dijkstra(vector<pair<int, int>> adj[], int src) {

priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

vector<int> dist(V, INT\_MAX);

pq.push(make\_pair(0, src));

dist[src] = 0;

while (!pq.empty()) {

int u = pq.top().second;

pq.pop();

for (auto x : adj[u]) {

int v = x.first;

int weight = x.second;

if (dist[u] + weight < dist[v]) {

dist[v] = dist[u] + weight;

pq.push(make\_pair(dist[v], v));

}

}

}

cout << "Vertex \t Distance from Source\n";

for (int i = 0; i < V; i++)

cout << i << " \t\t " << dist[i] << "\n";

}

int main() {

vector<pair<int, int>> adj[V] = {

{{1, 4}, {2, 8}},

{{0, 4}, {2, 2}, {3, 5}},

{{0, 8}, {1, 2}, {3, 7}, {4, 9}},

{{1, 5}, {2, 7}, {4, 3}},

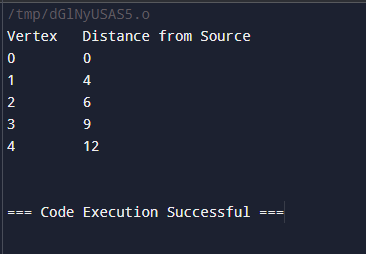
{{2, 9}, {3, 3}}

};

dijkstra(adj, 0);

return 0;

}



Huffman

#include <iostream>

#include <queue>

#include <unordered\_map>

#include <string>

using namespace std;

// Node structure for the Huffman tree

struct HuffmanNode {

char character;

int frequency;

HuffmanNode\* left;

HuffmanNode\* right;

HuffmanNode(char c, int freq) : character(c), frequency(freq), left(nullptr), right(nullptr) {}

};

// Comparator for priority queue

struct Compare {

bool operator()(HuffmanNode\* left, HuffmanNode\* right) {

return left->frequency > right->frequency; // Min-heap

}

};

class HuffmanCoding {

public:

HuffmanNode\* buildTree(const unordered\_map<char, int>& freq) {

priority\_queue<HuffmanNode\*, vector<HuffmanNode\*>, Compare> pq;

for (const auto& pair : freq) pq.push(new HuffmanNode(pair.first, pair.second));

while (pq.size() > 1) {

HuffmanNode\* left = pq.top(); pq.pop();

HuffmanNode\* right = pq.top(); pq.pop();

HuffmanNode\* newNode = new HuffmanNode('\0', left->frequency + right->frequency);

newNode->left = left; newNode->right = right;

pq.push(newNode);

}

return pq.top();

}

void generateCodes(HuffmanNode\* root, const string& code, unordered\_map<char, string>& codes) {

if (!root) return;

if (!root->left && !root->right) codes[root->character] = code;

generateCodes(root->left, code + "0", codes);

generateCodes(root->right, code + "1", codes);

}

string encode(const string& input, const unordered\_map<char, string>& codes) {

string encoded;

for (char c : input) encoded += codes.at(c);

return encoded;

}

string decode(HuffmanNode\* root, const string& encoded) {

string decoded;

HuffmanNode\* current = root;

for (char bit : encoded) {

current = (bit == '0') ? current->left : current->right;

if (!current->left && !current->right) {

decoded += current->character;

current = root;

}

}

return decoded;

}

};

int main() {

string input = "huffman coding is a data compression algorithm";

unordered\_map<char, int> frequencies;

// Calculate frequencies

for (char c : input) frequencies[c]++;

HuffmanCoding huffman;

HuffmanNode\* root = huffman.buildTree(frequencies);

unordered\_map<char, string> codes;

huffman.generateCodes(root, "", codes);

string encoded = huffman.encode(input, codes);

cout << "Encoded String: " << encoded << endl;

cout << "Decoded String: " << huffman.decode(root, encoded) << endl;

return 0;

}