Practical 1 Randomized Quicksort

#include <iostream>

#include <cstdlib>

#include <ctime>

using namespace std;

void swap(int& a, int& b) {

int temp = a;

a = b;

b = temp;

}

int partition(int arr[], int low, int high, int& comparisonCount) {

int randomIndex = low + rand() % (high - low + 1);

swap(arr[randomIndex], arr[high]);

int pivot = arr[high];

int i = low - 1;

for (int j = low; j <= high - 1; ++j) {

++comparisonCount;

if (arr[j] <= pivot) {

++i;

swap(arr[i], arr[j]);

}

}

swap(arr[i + 1], arr[high]);

return i + 1;

}

void randomizedQuickSort(int arr[], int low, int high, int& comparisonCount) {

if (low < high) {

int pivotIndex = partition(arr, low, high, comparisonCount);

randomizedQuickSort(arr, low, pivotIndex - 1, comparisonCount);

randomizedQuickSort(arr, pivotIndex + 1, high, comparisonCount);

}

}

int main() {

srand(static\_cast<unsigned>(time(0)));

int n;

cout << "Enter the number of elements: ";

cin >> n;

int\* arr = new int[n];

cout << "Enter the elements of the array:\n";

for (int i = 0; i < n; ++i) {

cin >> arr[i];

}

int comparisonCount = 0;

randomizedQuickSort(arr, 0, n - 1, comparisonCount);

cout << "\nSorted array:\n";

for (int i = 0; i < n; ++i) {

cout << arr[i] << " ";

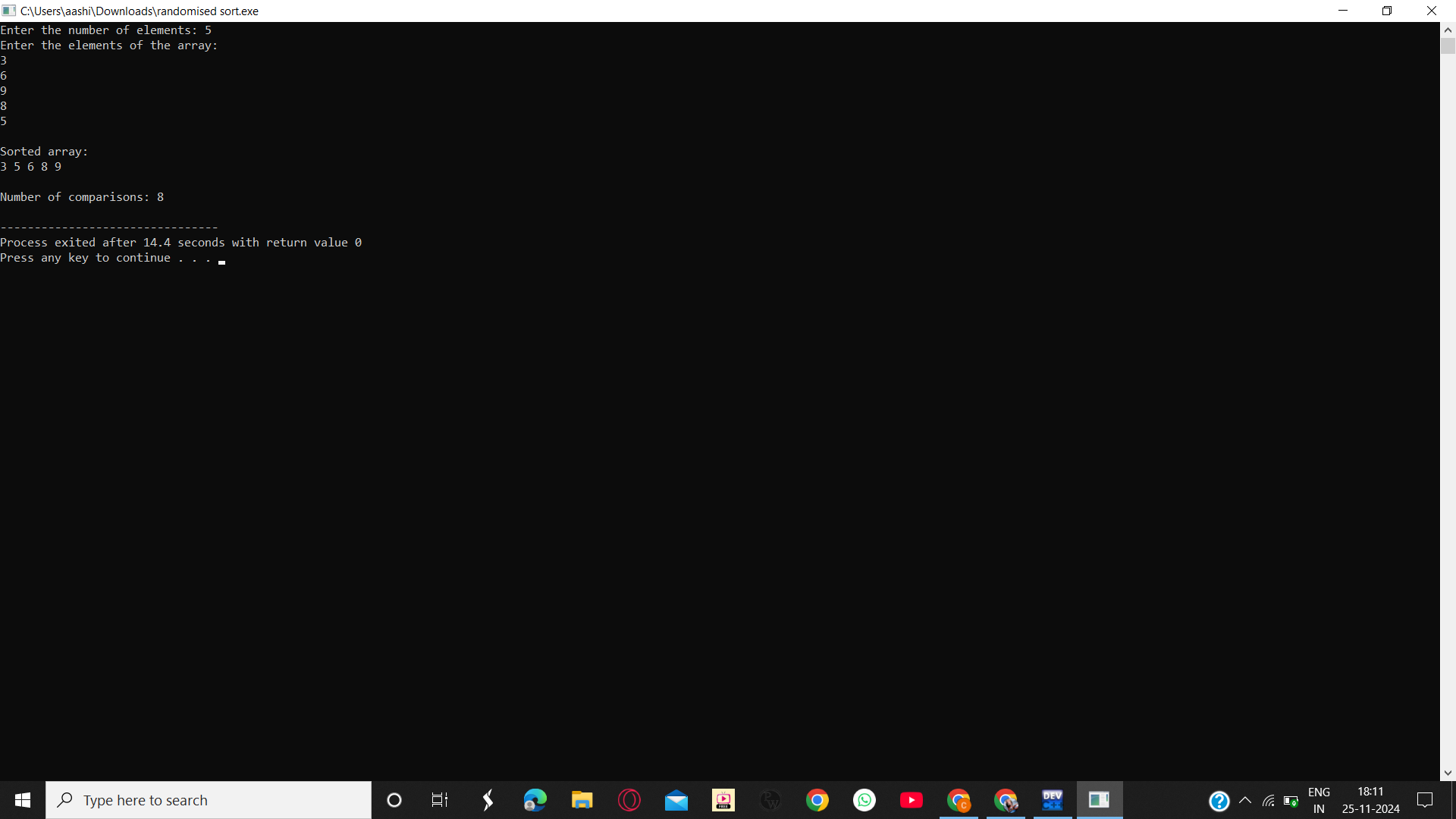
}

cout << "\n\nNumber of comparisons: " << comparisonCount << endl;

delete[] arr;

return 0;

}



Practical 2: Randomized select

#include <iostream>

#include <cstdlib>

#include <ctime>

using namespace std;

void swap(int& a, int& b) {

int temp = a;

a = b;

b = temp;

}

int partition(int arr[], int low, int high) {

int pivot = arr[high];

int i = low - 1;

for (int j = low; j < high; ++j) {

if (arr[j] <= pivot) {

++i;

swap(arr[i], arr[j]);

}

}

swap(arr[i + 1], arr[high]);

return i + 1;

}

int randomizedPartition(int arr[], int low, int high) {

int randomIndex = low + rand() % (high - low + 1);

swap(arr[randomIndex], arr[high]);

return partition(arr, low, high);

}

int randomizedSelect(int arr[], int low, int high, int i) {

if (low == high) {

return arr[low];

}

int pivotIndex = randomizedPartition(arr, low, high);

int k = pivotIndex - low + 1;

if (i == k) {

return arr[pivotIndex];

} else if (i < k) {

return randomizedSelect(arr, low, pivotIndex - 1, i);

} else {

return randomizedSelect(arr, pivotIndex + 1, high, i - k);

}

}

int main() {

srand((unsigned int)time(0));

int n, i;

cout << "Enter the number of elements: ";

cin >> n;

int\* arr = new int[n];

cout << "Enter the elements of the array:\n";

for (int j = 0; j < n; ++j) {

cin >> arr[j];

}

cout << "Enter the value of i (1-based index): ";

cin >> i;

if (i < 1 || i > n) {

cout << "Invalid value of i. It must be between 1 and " << n << ".\n";

} else {

int result = randomizedSelect(arr, 0, n - 1, i);

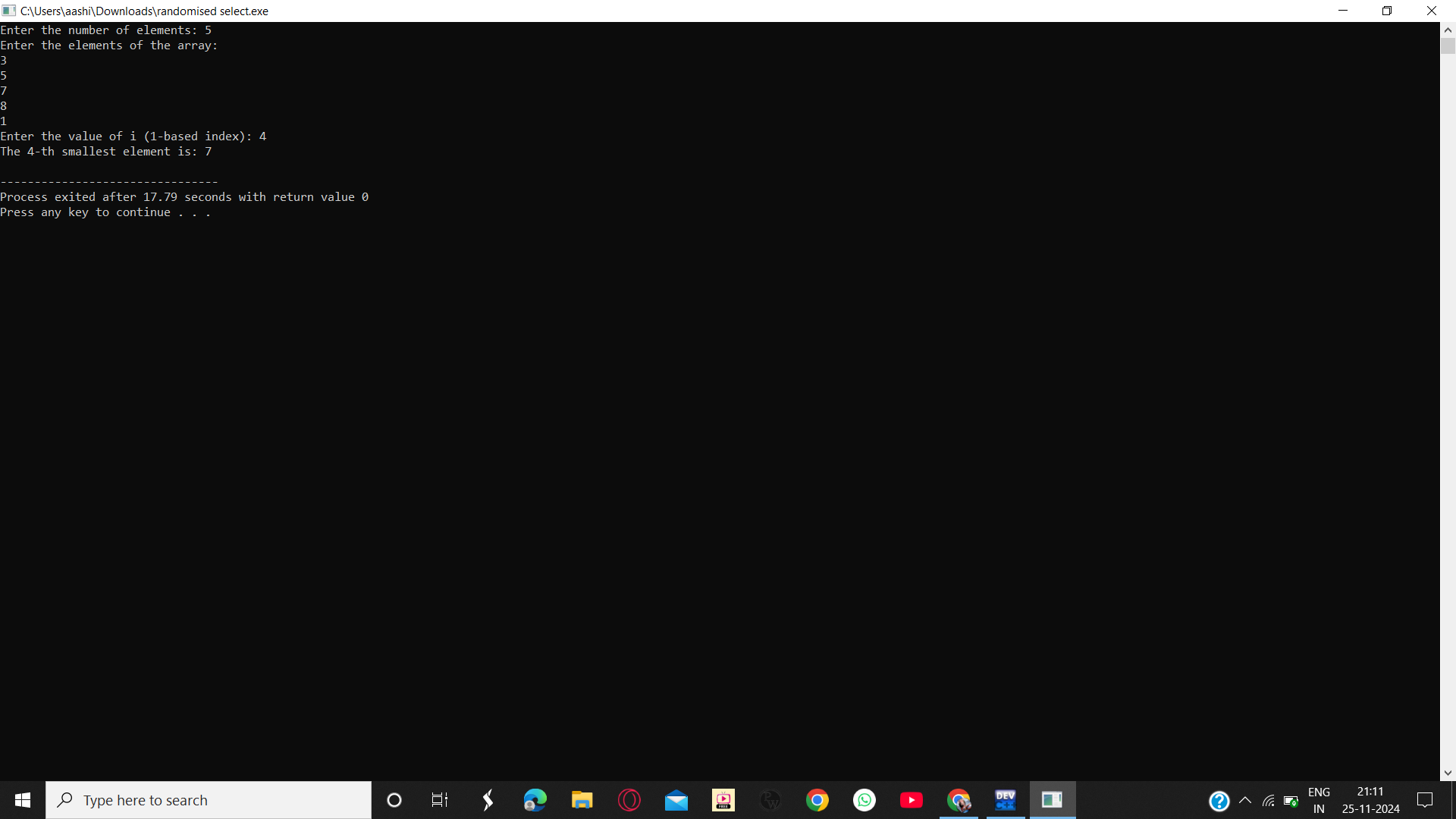
cout << "The " << i << "-th smallest element is: " << result << endl;

}

delete[] arr;

return 0;

}



Practical 3: Kruskal algorithm

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

struct Edge {

int src, dest, weight;

};

bool compareEdges(const Edge& e1, const Edge& e2) {

return e1.weight < e2.weight;

}

class DSU {

private:

vector<int> parent;

vector<int> rank;

public:

DSU(int n) {

parent = vector<int>(n);

rank = vector<int>(n, 0);

for (int i = 0; i < n; ++i) {

parent[i] = i;

}

}

int find(int x) {

if (parent[x] != x) {

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

}

return parent[x];

}

bool unionSets(int x, int y) {

int rootX = find(x);

int rootY = find(y);

if (rootX == rootY) return false;

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

parent[rootY] = rootX;

} else if (rank[rootX] < rank[rootY]) {

parent[rootX] = rootY;

} else {

parent[rootY] = rootX;

rank[rootX]++;

}

return true;

}

};

vector<Edge> kruskalMST(int vertices, vector<Edge>& edges) {

sort(edges.begin(), edges.end(), compareEdges);

DSU dsu(vertices);

vector<Edge> mst;

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

if (dsu.unionSets(edges[i].src, edges[i].dest)) {

mst.push\_back(edges[i]);

}

if (mst.size() == static\_cast<size\_t>(vertices - 1)) {

break;

}

}

return mst;

}

int main() {

int vertices, edgesCount;

cout << "Enter the number of vertices: ";

cin >> vertices;

cout << "Enter the number of edges: ";

cin >> edgesCount;

vector<Edge> edges(edgesCount);

cout << "Enter the edges (source, destination, weight):\n";

for (int i = 0; i < edgesCount; ++i) {

cin >> edges[i].src >> edges[i].dest >> edges[i].weight;

}

vector<Edge> mst = kruskalMST(vertices, edges);

cout << "\nMinimum Spanning Tree:\n";

int totalWeight = 0;

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

cout << mst[i].src << " - " << mst[i].dest << " : " << mst[i].weight << endl;

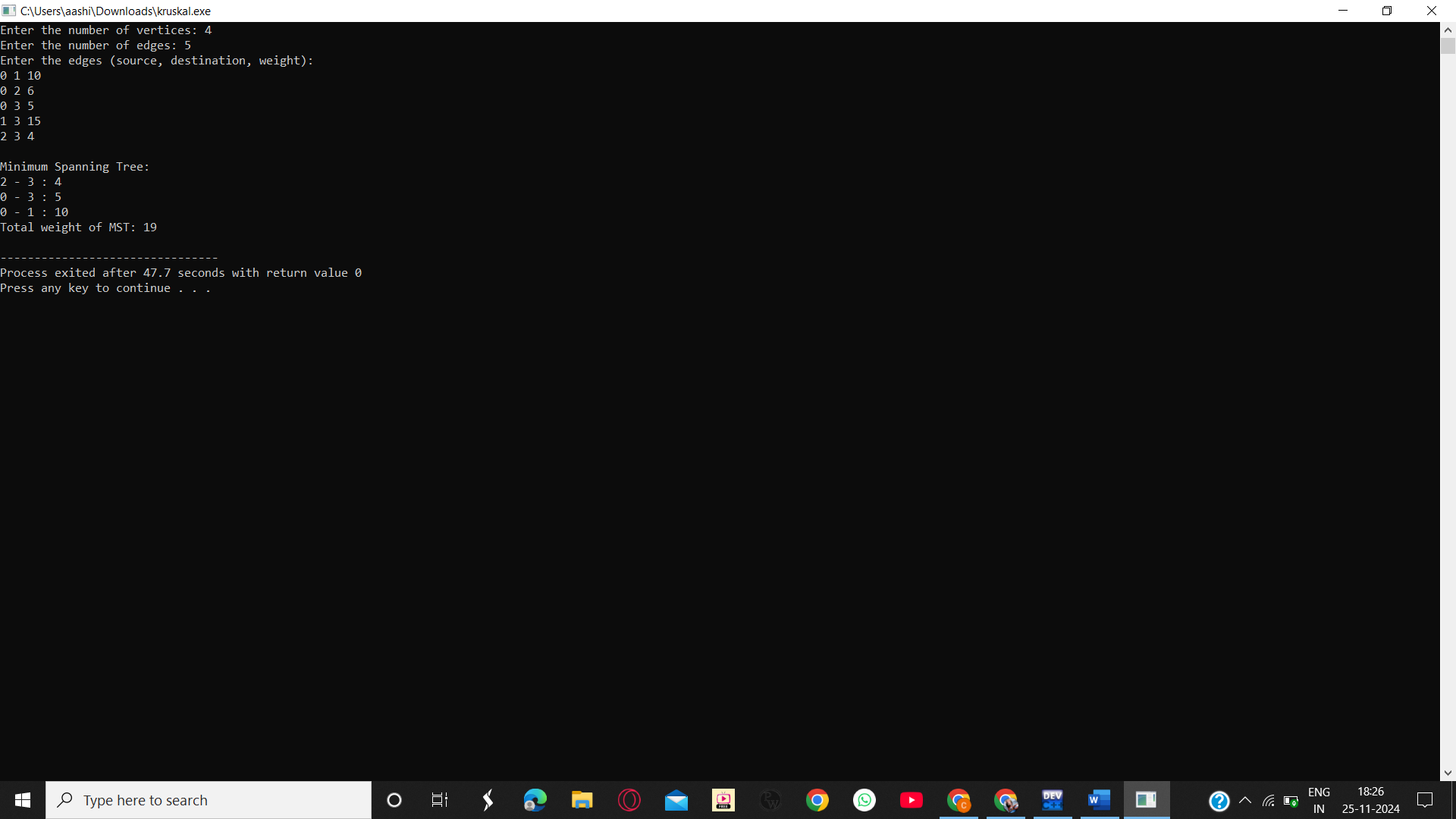
totalWeight += mst[i].weight;

}

cout << "Total weight of MST: " << totalWeight << endl;

return 0;

}



Practical 4: bellman ford algorithm

#include <iostream>

#include <vector>

#include <climits>

using namespace std;

struct Edge {

int src, dest, weight;

};

bool bellmanFord(int vertices, int edgesCount, vector<Edge>& edges, int source, vector<int>& distances) {

distances.clear();

distances.resize(vertices, INT\_MAX);

distances[source] = 0;

for (int i = 0; i < vertices - 1; ++i) {

for (size\_t j = 0; j < edges.size(); ++j) {

Edge edge = edges[j];

if (distances[edge.src] != INT\_MAX && distances[edge.src] + edge.weight < distances[edge.dest]) {

distances[edge.dest] = distances[edge.src] + edge.weight;

}

}

}

for (size\_t j = 0; j < edges.size(); ++j) {

Edge edge = edges[j];

if (distances[edge.src] != INT\_MAX && distances[edge.src] + edge.weight < distances[edge.dest]) {

return false;

}

}

return true;

}

int main() {

int vertices, edgesCount, source;

cout << "Enter the number of vertices: ";

cin >> vertices;

cout << "Enter the number of edges: ";

cin >> edgesCount;

vector<Edge> edges(edgesCount);

cout << "Enter the edges (source, destination, weight):\n";

for (int i = 0; i < edgesCount; ++i) {

cin >> edges[i].src >> edges[i].dest >> edges[i].weight;

}

cout << "Enter the source vertex: ";

cin >> source;

vector<int> distances;

if (bellmanFord(vertices, edgesCount, edges, source, distances)) {

cout << "\nShortest distances from source " << source << ":\n";

for (int i = 0; i < vertices; ++i) {

if (distances[i] == INT\_MAX) {

cout << "Vertex " << i << " is unreachable\n";

} else {

cout << "Vertex " << i << ": " << distances[i] << endl;

}

}

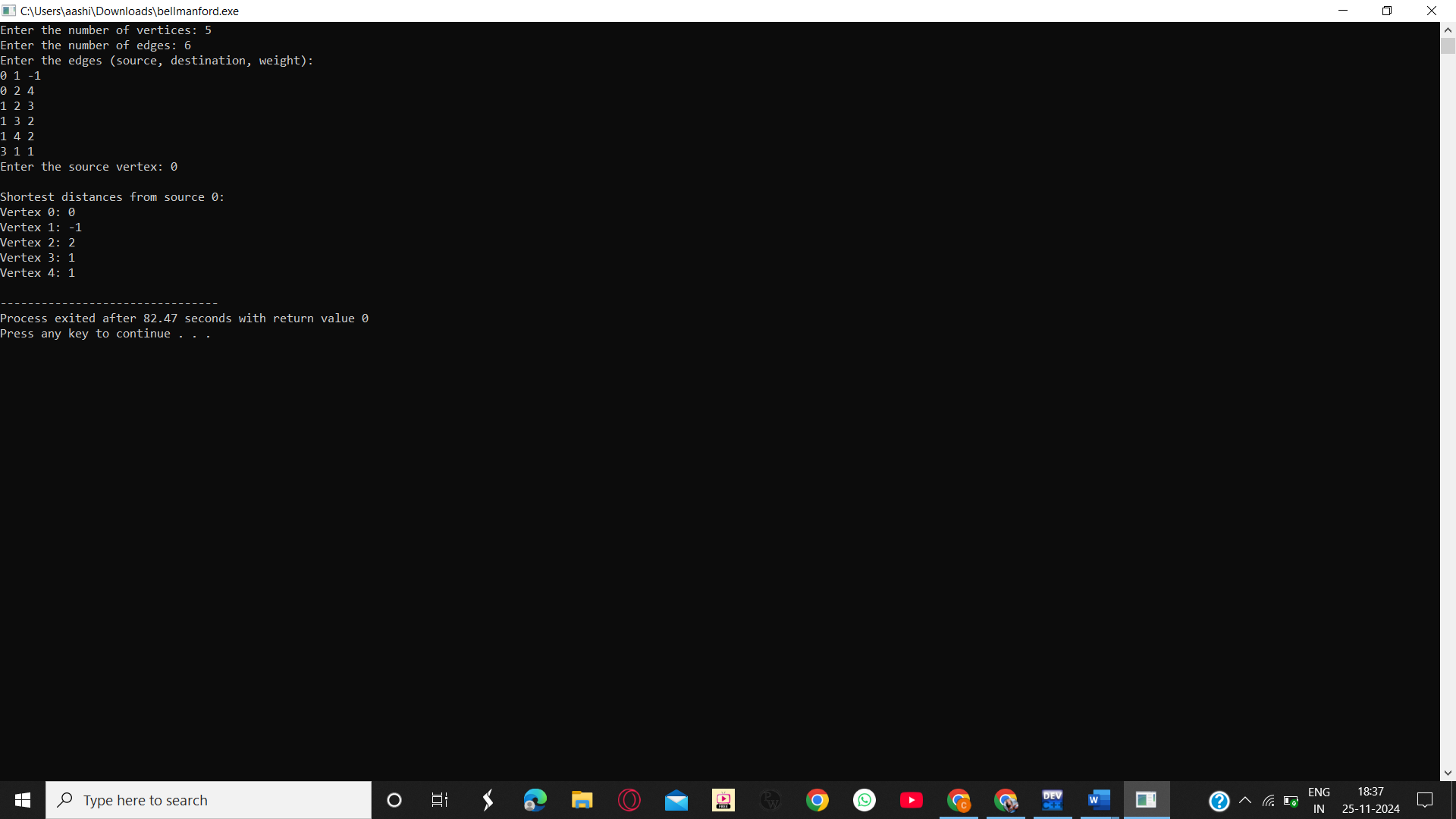
} else {

cout << "\nGraph contains a negative weight cycle. Shortest paths cannot be determined.\n";

}

return 0;

}



Practical 5: B Tree

#include <iostream>

using namespace std;

class btreenode {

public:

int\* keys;

int keytally;

int t;

bool leaf;

btreenode\*\* pointers;

btreenode(int degree, bool isleaf) {

t = degree;

leaf = isleaf;

keytally = 0;

keys = new int[2 \* t - 1];

pointers = new btreenode\*[2 \* t];

}

~btreenode() {

delete[] keys;

for (int i = 0; i <= keytally; i++) {

delete pointers[i];

}

delete[] pointers;

}

void traverse();

btreenode\* search(int k);

void notfull(int k);

void split(int i, btreenode\* y);

void deletekey(int k);

void deletefromleaf(int idx);

void deletefromnonleaf(int idx);

int getpredecessor(int idx);

int getsuccessor(int idx);

void fill(int idx);

void borrowfromprev(int idx);

void borrowfromnext(int idx);

void merge(int idx);

};

class btree {

btreenode\* root;

int t;

public:

btree(int degree) {

t = degree;

root = nullptr;

}

~btree() {

delete root;

}

void traverse() {

if (root != nullptr) {

root->traverse();

} else {

cout << "The tree is empty." << endl;

}

}

btreenode\* search(int k) {

if (root == nullptr) {

return nullptr;

}

return root->search(k);

}

void insert(int k);

void deletekey(int k);

};

void btreenode::traverse() {

int i;

for (i = 0; i < keytally; i++) {

if (!leaf) {

pointers[i]->traverse();

}

cout << keys[i] << " ";

}

if (!leaf) {

pointers[i]->traverse();

}

}

btreenode\* btreenode::search(int k) {

int i = 0;

while (i < keytally && keys[i] < k) {

i++;

}

if (i < keytally && keys[i] == k) {

return this;

}

if (leaf) {

return nullptr;

}

return pointers[i]->search(k);

}

void btree::insert(int k) {

if (root == nullptr) {

root = new btreenode(t, true);

root->keys[0] = k;

root->keytally = 1;

} else {

if (root->keytally == 2 \* t - 1) {

btreenode\* s = new btreenode(t, false);

s->pointers[0] = root;

s->split(0, root);

int i = 0;

if (s->keys[0] < k) {

i = 1;

}

s->pointers[i]->notfull(k);

root = s;

} else {

root->notfull(k);

}

}

}

void btreenode::notfull(int k) {

int i = keytally - 1;

if (leaf) {

while (i >= 0 && keys[i] > k) {

keys[i + 1] = keys[i];

i--;

}

keys[i + 1] = k;

keytally++;

} else {

while (i >= 0 && keys[i] > k) {

i--;

}

i++;

if (pointers[i]->keytally == 2 \* t - 1) {

split(i, pointers[i]);

if (keys[i] < k) {

i++;

}

}

pointers[i]->notfull(k);

}

}

void btreenode::split(int i, btreenode\* y) {

btreenode\* z = new btreenode(y->t, y->leaf);

z->keytally = t - 1;

for (int j = 0; j < t - 1; j++) {

z->keys[j] = y->keys[j + t];

}

if (!y->leaf) {

for (int j = 0; j < t; j++) {

z->pointers[j] = y->pointers[j + t];

}

}

y->keytally = t - 1;

for (int j = keytally; j >= i + 1; j--) {

pointers[j + 1] = pointers[j];

}

pointers[i + 1] = z;

for (int j = keytally - 1; j >= i; j--) {

keys[j + 1] = keys[j];

}

keys[i] = y->keys[t - 1];

keytally++;

}

void btree::deletekey(int k) {

if (!root) {

cout << "The tree is empty." << endl;

return;

}

root->deletekey(k);

if (root->keytally == 0) {

btreenode\* temp = root;

root = root->leaf ? nullptr : root->pointers[0];

delete temp;

}

}

void btreenode::deletekey(int k) {

int idx = 0;

while (idx < keytally && keys[idx] < k) {

idx++;

}

if (idx < keytally && keys[idx] == k) {

if (leaf) {

deletefromleaf(idx);

} else {

deletefromnonleaf(idx);

}

} else {

if (leaf) {

cout << "The key " << k << " is not in the tree." << endl;

return;

}

bool flag = (idx == keytally);

if (pointers[idx]->keytally < t) {

fill(idx);

}

pointers[idx]->deletekey(k);

}

}

void btreenode::deletefromleaf(int idx) {

for (int i = idx + 1; i < keytally; ++i) {

keys[i - 1] = keys[i];

}

keytally--;

}

void btreenode::deletefromnonleaf(int idx) {

int k = keys[idx];

if (pointers[idx]->keytally >= t) {

int pred = getpredecessor(idx);

keys[idx] = pred;

pointers[idx]->deletekey(pred);

} else if (pointers[idx + 1]->keytally >= t) {

int succ = getsuccessor(idx);

keys[idx] = succ;

pointers[idx + 1]->deletekey(succ);

} else {

merge(idx);

pointers[idx]->deletekey(k);

}

}

int btreenode::getpredecessor(int idx) {

btreenode\* cur = pointers[idx];

while (!cur->leaf) {

cur = cur->pointers[cur->keytally];

}

return cur->keys[cur->keytally - 1];

}

int btreenode::getsuccessor(int idx) {

btreenode\* cur = pointers[idx + 1];

while (!cur->leaf) {

cur = cur->pointers[0];

}

return cur->keys[0];

}

void btreenode::fill(int idx) {

if (idx != 0 && pointers[idx - 1]->keytally >= t) {

borrowfromprev(idx);

} else if (idx != keytally && pointers[idx + 1]->keytally >= t) {

borrowfromnext(idx);

} else {

merge(idx);

}

}

void btreenode::borrowfromprev(int idx) {

btreenode\* child = pointers[idx];

btreenode\* sibling = pointers[idx - 1];

for (int i = child->keytally - 1; i >= 0; --i) {

child->keys[i + 1] = child->keys[i];

}

if (!child->leaf) {

for (int i = child->keytally; i >= 0; --i) {

child->pointers[i + 1] = child->pointers[i];

}

}

child->keys[0] = keys[idx - 1];

if (!leaf) {

child->pointers[0] = sibling->pointers[sibling->keytally];

}

keys[idx - 1] = sibling->keys[sibling->keytally - 1];

child->keytally += 1;

sibling->keytally -= 1;

}

void btreenode::borrowfromnext(int idx) {

btreenode\* child = pointers[idx];

btreenode\* sibling = pointers[idx + 1];

child->keys[child->keytally] = keys[idx];

if (!child->leaf) {

child->pointers[child->keytally + 1] = sibling->pointers[0];

}

keys[idx] = sibling->keys[0];

for (int i = 1; i < sibling->keytally; ++i) {

sibling->keys[i - 1] = sibling->keys[i];

}

if (!sibling->leaf) {

for (int i = 1; i <= sibling->keytally; ++i) {

sibling->pointers[i - 1] = sibling->pointers[i];

}

}

child->keytally += 1;

sibling->keytally -= 1;

}

void btreenode::merge(int idx) {

btreenode\* child = pointers[idx];

btreenode\* sibling = pointers[idx + 1];

child->keys[t - 1] = keys[idx];

for (int i = 0; i < sibling->keytally; ++i) {

child->keys[i + t] = sibling->keys[i];

}

if (!child->leaf) {

for (int i = 0; i <= sibling->keytally; ++i) {

child->pointers[i + t] = sibling->pointers[i];

}

}

for (int i = idx + 1; i < keytally; ++i) {

keys[i - 1] = keys[i];

}

for (int i = idx + 2; i <= keytally; ++i) {

pointers[i - 1] = pointers[i];

}

child->keytally += sibling->keytally + 1;

keytally--;

delete sibling;

}

void displaymenu() {

cout << "\n------- B-Tree Menu -------" << endl;

cout << "1. Insert a key" << endl;

cout << "2. Delete a key" << endl;

cout << "3. Search for a key" << endl;

cout << "4. Traverse the tree" << endl;

cout << "5. Exit" << endl;

}

int main() {

int degree;

cout << "Enter the degree of the B-tree: ";

cin >> degree;

btree bt(degree);

int choice, key;

while (true) {

displaymenu();

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter the key to insert: ";

cin >> key;

bt.insert(key);

bt.traverse();

break;

case 2:

cout << "Enter the key to delete: ";

cin >> key;

bt.deletekey(key);

bt.traverse();

break;

case 3:

cout << "Enter the key to search for: ";

cin >> key;

if (bt.search(key)) {

cout << "Key " << key << " found in the tree." << endl;

} else {

cout << "Key " << key << " not found." << endl;

}

break;

case 4:

cout << "Traversing the B-tree: ";

bt.traverse();

cout << endl;

break;

case 5:

cout << "Exiting program." << endl;

return 0;

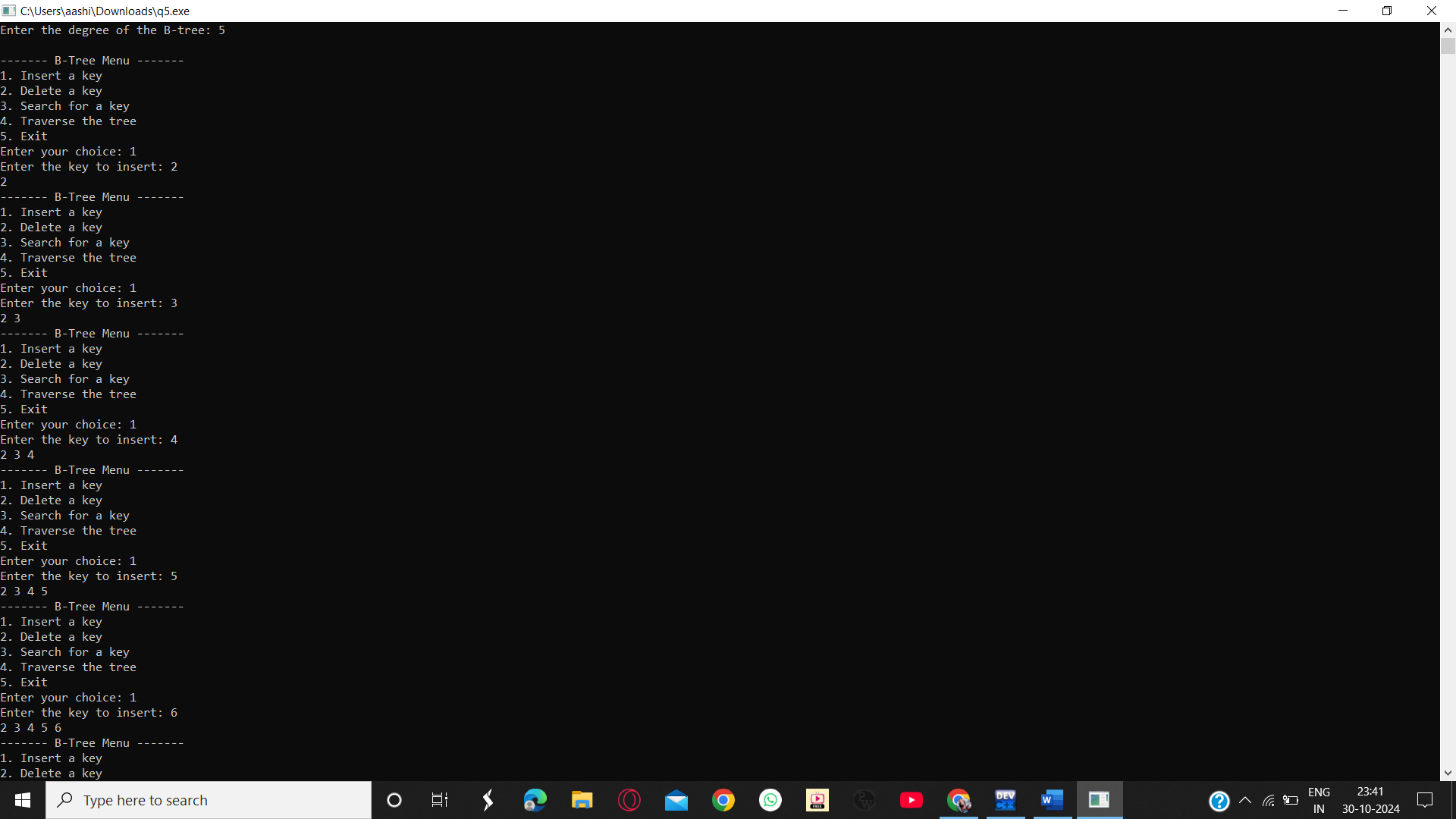
default:

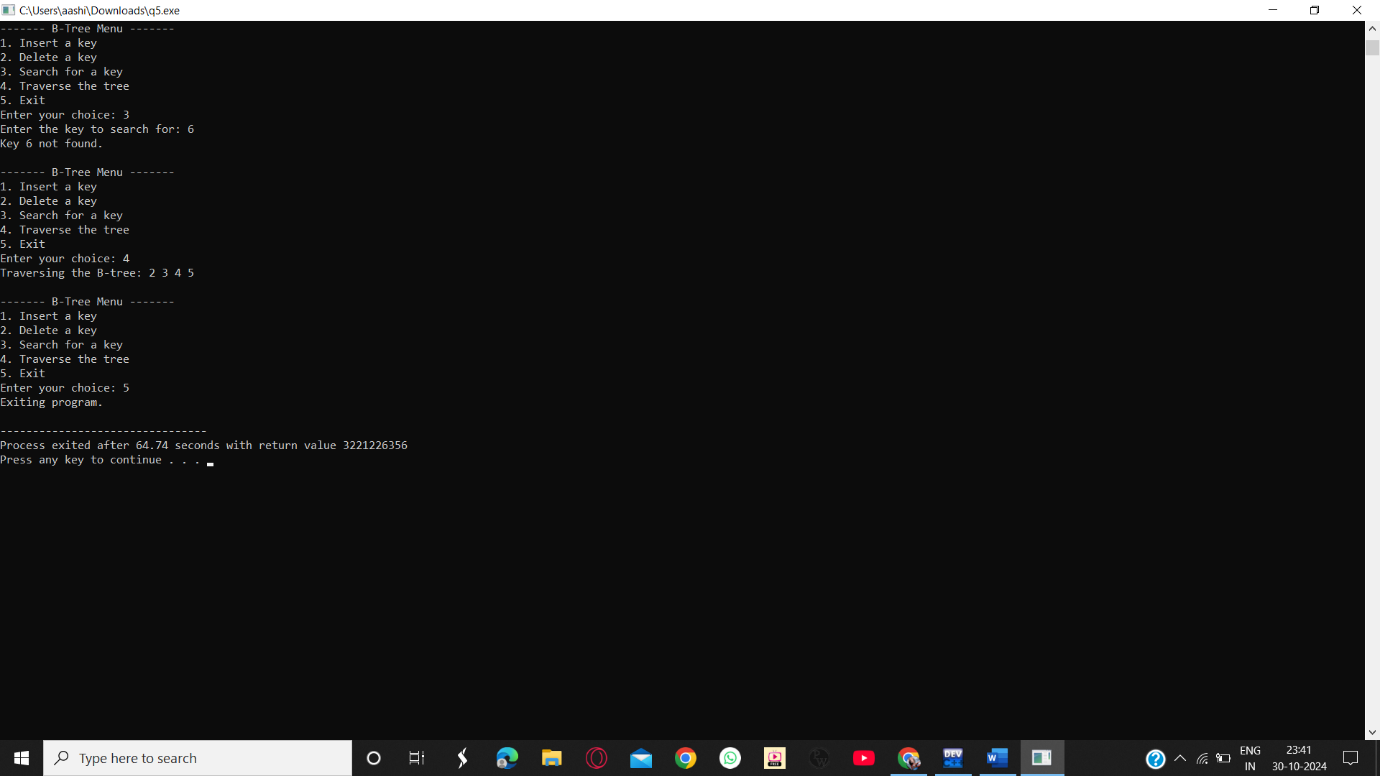
cout << "Invalid choice! Please try again." << endl;

break;

}

}

}



Practical 6: KMP algorithm

#include <iostream>

#include <string>

using namespace std;

void fillFailure(string str, int failure[]) {

int n = str.length();

int len = 0;

failure[0] = 0;

int i = 1;

while (i < n) {

if (str[i] == str[len]) {

len++;

failure[i] = len;

i++;

} else {

if (len == 0) {

failure[i] = 0;

i++;

} else {

len = failure[len - 1];

}

}

}

}

int kmp(string pat, string txt) {

int n = txt.length();

int m = pat.length();

int failure[m];

fillFailure(pat, failure);

int i = 0;

int j = 0;

while (i < n) {

if (pat[j] == txt[i]) {

i++;

j++;

}

if (j == m) {

return i - j;

j = failure[j - 1];

} else if (i < n && pat[j] != txt[i]) {

if (j == 0) {

i++;

} else {

j = failure[j - 1];

}

}

}

return -1;

}

int main() {

string pat;

string txt;

cout << "Enter the text: ";

cin >> txt;

cout << "Enter the pattern: ";

cin >> pat;

int result = kmp(pat, txt);

if (result != -1) {

cout << "Pattern found at index " << result << endl;

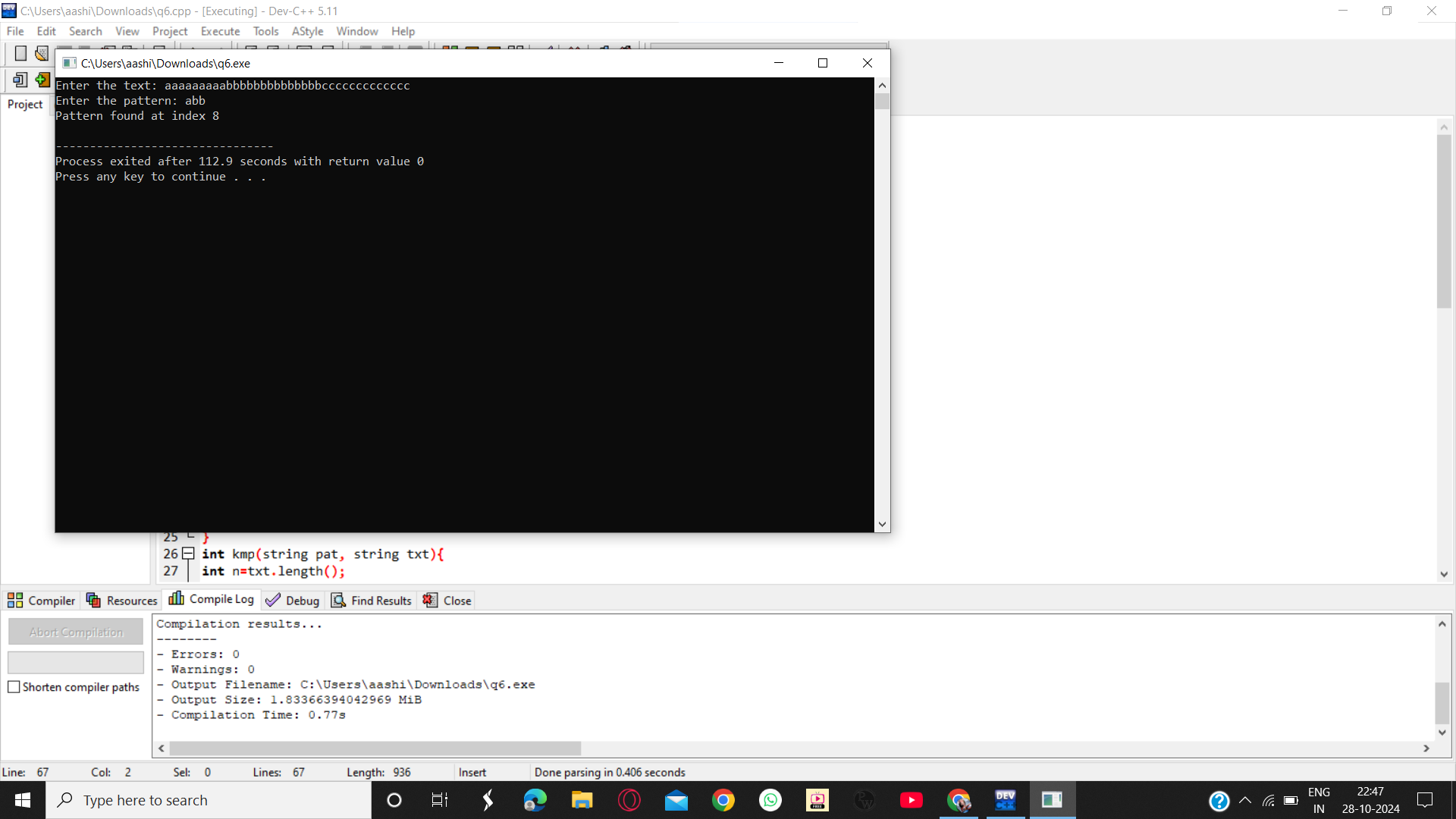
} else {

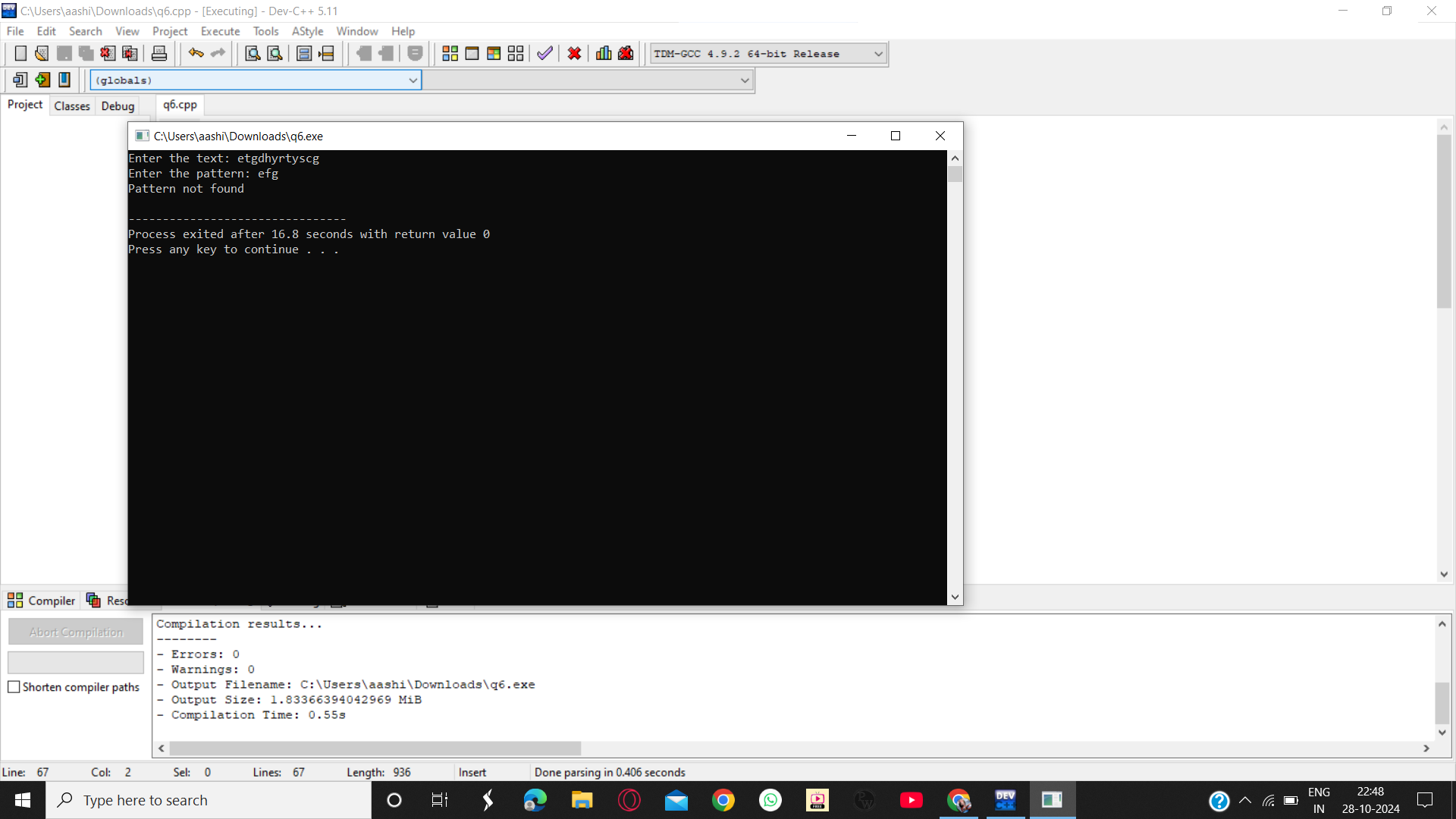
cout << "Pattern not found" << endl;

}

return 0;

}





Practical 7: Suffix Trie

#include <iostream>

#include <string>

using namespace std;

struct Node {

Node\* children[26];

bool isEndOfWord;

Node() {

for (int i = 0; i < 26; i++) {

children[i] = NULL;

}

isEndOfWord = false;

}

};

void insertSuffix(Node\* root, string suffix) {

Node\* curr = root;

for (int i = 0; i < suffix.length(); i++) {

char c = suffix[i];

int index = c - 'a';

if (curr->children[index] == NULL) {

curr->children[index] = new Node();

}

curr = curr->children[index];

}

curr->isEndOfWord = true;

}

void bruteForceSuffixTrie(Node\* root, string text) {

for (int i = 0; i < text.length(); i++) {

insertSuffix(root, text.substr(i));

}

}

bool search(Node\* root, string pattern) {

Node\* curr = root;

for (int i = 0; i < pattern.length(); i++) {

char c = pattern[i];

int index = c - 'a';

if (curr->children[index] == NULL) {

return false;

}

curr = curr->children[index];

}

return curr->isEndOfWord;

}

int main() {

Node\* root = new Node();

string text;

cout << "Enter the text: ";

getline(cin, text);

bruteForceSuffixTrie(root, text);

while (true) {

cout << "Enter a pattern to search (or 'quit' to exit): ";

string pattern;

getline(cin, pattern);

if (pattern == "quit") {

break;

}

if (search(root, pattern)) {

cout << "Pattern found!" << endl;

} else {

cout << "Pattern not found." << endl;

}

}

return 0;

}

