

Applied Programming

Assignment#5



November 30, 2022

Muhammad talha Arif

22F-3769

# **Question#1**

Graphs (Robot Navigation System)

## **Code**

#include <iostream>

#include <string>

#include <queue>

#include <stack>

#include <fstream>

using namespace std;

struct Vertex

{

int row, column;

int cost;

};

class Graph

{

private:

int\*\* points;

bool graph\_initialized;

int start\_offset[2], goal\_offset[2];

int row, column;

void GraphOutput(int\*\* arr);

public:

Graph();

Graph(int\*\* arr, int row, int column, int\* start\_offset, int\* goal\_offset);

bool ValidState(int row, int column);

void BFS();

void DFS();

};

void Graph::GraphOutput(int \*\*arr)

{

int x;

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

{

for (int j = 0; j < column; j++)

{

x = arr[i][j];

if( x == 2 || x == 3 )

std::cout << "\* ";

else

std::cout << x << " ";

}

std::cout << endl;

}

}

Graph::Graph()

{

this->graph\_initialized = false;

}

Graph::Graph(int\*\* points, int row, int column, int\* start\_offset, int\* goal\_offset)

{

this->points = points;

this->row = row;

this->column = column;

this->start\_offset[0] = row - start\_offset[0];

this->start\_offset[1] = start\_offset[1] - 1;

this->goal\_offset[0] = row - goal\_offset[0];

this->goal\_offset[1] = goal\_offset[1] - 1;

this->graph\_initialized = true;

}

bool Graph::ValidState(int row, int column)

{

if (!graph\_initialized)

{

std::cout << "Graph is not initialized !!!" << endl;

return false;

}

if (points[start\_offset[0]][start\_offset[1]] == 1)

{

std::cout << "Start State is Blocked !!!" << endl;

return false;

}

if (points[goal\_offset[0]][goal\_offset[1]] == 1)

{

std::cout << "Goal State is Blocked !!!" << endl;

return false;

}

return true;

}

void Graph::BFS()

{

queue<Vertex> points\_queue;

bool goal\_found = false;

int\*\* arr;

arr = new int\* [row];

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

{

arr[i] = new int[column];

}

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

{

for (int j = 0; j < column; j++)

{

arr[i][j] = points[i][j];

}

}

points\_queue.push({ start\_offset[0], start\_offset[1], 0 });

Vertex p;

int total\_count = 0;

int total\_cost = 0;

std::cout << "Path: ";

while (!points\_queue.empty())

{

p = points\_queue.front();

total\_cost += p.cost;

std::cout << (row - p.row) << ":" << (p.column + 1) << " -> ";

points\_queue.pop();

if (p.row == goal\_offset[0] && p.column == goal\_offset[1])

{

std::cout << "Goal Found !!!" << endl;

goal\_found = true;

break;

}

// Up

if ((p.row - 1) >= 0)

{

if (arr[p.row - 1][p.column] == 0)

{

points\_queue.push({ p.row - 1, p.column, p.cost + 2 });

arr[p.row - 1][p.column] = 2;

}

}

// Diagonal

if ((p.row - 1) >= 0 && (p.column + 1) < column)

{

if (arr[p.row - 1][p.column + 1] == 0)

{

points\_queue.push({ p.row - 1, p.column + 1, p.cost + 3 });

arr[p.row - 1][p.column + 1] = 3;

}

}

// Right

if ((p.column + 1) < column)

{

if (arr[p.row][p.column + 1] == 0)

{

points\_queue.push({ p.row, p.column + 1, p.cost + 2 });

arr[p.row][p.column + 1] = 2;

}

}

total\_count++;

}

std::cout << "Total Iterations: " << total\_count << endl;

std::cout << "Total Cost: " << total\_cost << endl;

GraphOutput(arr);

}

void Graph::DFS()

{

stack<Vertex> points\_stack;

bool goal\_found = false;

int\*\* arr;

arr = new int\* [row];

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

{

arr[i] = new int[column];

}

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

{

for (int j = 0; j < column; j++)

{

arr[i][j] = points[i][j];

}

}

points\_stack.push({ start\_offset[0], start\_offset[1], 0 });

Vertex p;

int total\_count = 0;

int total\_cost = 0;

std::cout << "Path: ";

while (!points\_stack.empty())

{

p = points\_stack.top();

total\_cost += p.cost;

std::cout << (row - p.row) << ":" << (p.column + 1) << " -> ";

points\_stack.pop();

if (p.row == goal\_offset[0] && p.column == goal\_offset[1])

{

std::cout << "Goal Found !!!" << endl;

goal\_found = true;

break;

}

// Up

if ((p.row - 1) >= 0)

{

if (arr[p.row - 1][p.column] == 0)

{

points\_stack.push({ p.row - 1, p.column, p.cost + 2 });

arr[p.row - 1][p.column] = 2;

}

}

// Diagonal

if ((p.row - 1) >= 0 && (p.column + 1) < column)

{

if (arr[p.row - 1][p.column + 1] == 0)

{

points\_stack.push({ p.row - 1, p.column + 1, p.cost + 3 });

arr[p.row - 1][p.column + 1] = 3;

}

}

// Right

if ((p.column + 1) < column)

{

if (arr[p.row][p.column + 1] == 0)

{

points\_stack.push({ p.row, p.column + 1, p.cost + 2 });

arr[p.row][p.column + 1] = 2;

}

}

total\_count++;

}

std::cout << "Total Iterations: " << total\_count << endl;

std::cout << "Total Cost: " << total\_cost << endl;

GraphOutput(arr);

}

int main()

{

ifstream fileHandler;

fileHandler.open("Grid.txt");

int col, row;

int start[2];

int goal[2];

if (!fileHandler.is\_open())

{

std::cout << "File Not Found !!!" << endl;

}

else

{

fileHandler >> row;

fileHandler >> col;

fileHandler >> start[0];

fileHandler >> start[1];

fileHandler >> goal[0];

fileHandler >> goal[1];

int\*\* arr = new int\* [row];

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

{

arr[i] = new int[col];

}

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

{

for (int j = 0; j < col; j++)

{

fileHandler >> arr[i][j];

}

}

/\*int data[row][col] =

{

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0 },

{ 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },

{ 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1 },

{ 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1 },

{ 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0 },

{ 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0 },

{ 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0 },

{ 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1 },

{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 },

{ 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1 },

{ 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1 },

{ 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1 },

{ 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1 }

};

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

{

for (int j = 0; j < col; j++)

arr[i][j] = data[i][j];

}\*/

Graph graph(arr, col, row, start, goal);

std::cout << "====================\nBreadth-First Search\n====================" << endl;

graph.BFS();

std::cout << endl;

std::cout << "==================\nDepth-First Search\n==================" << endl;

graph.DFS();

std::cout << endl;

}

system("PAUSE");

return 0;

}

## **Grid.txt**

15 15

2 1

14 15

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 1 0 0 0 0 0 0 1 1 0 0 0 0 0

1 1 0 0 1 1 0 0 1 1 0 0 0 1 1

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 1 1 1 1 1 1 0 0 0 1 1

0 1 1 0 0 0 0 0 0 0 0 0 0 1 1

0 1 1 0 0 0 1 1 0 0 1 1 0 0 0

0 1 1 0 1 1 1 1 1 0 1 1 0 0 0

0 0 0 0 1 1 1 1 1 0 1 1 0 0 0

0 1 0 0 1 1 1 1 1 0 1 1 0 0 1

0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

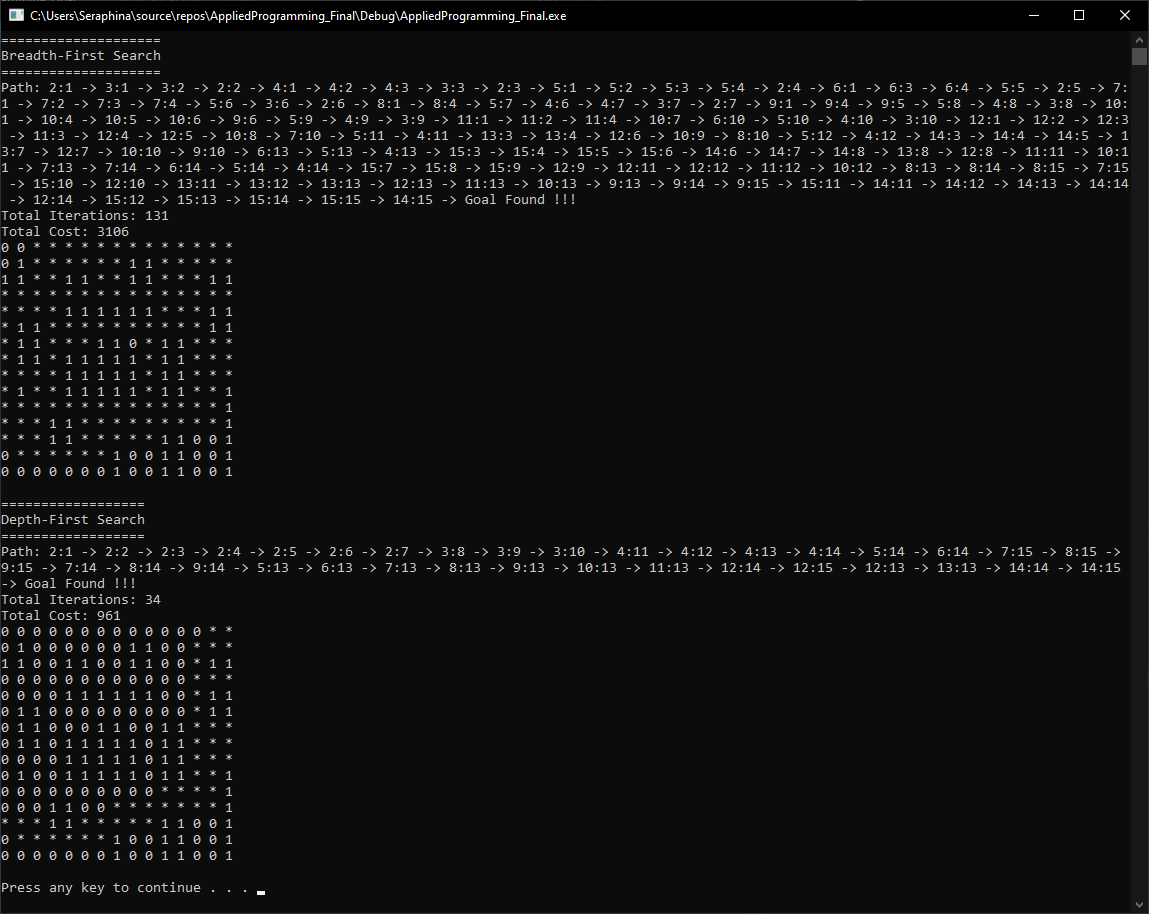
0 0 0 1 1 0 0 0 0 0 0 0 0 0 1

0 0 0 1 1 0 0 0 0 0 1 1 0 0 1

0 0 0 0 0 0 0 1 0 0 1 1 0 0 1

0 0 0 0 0 0 0 1 0 0 1 1 0 0 1

## **Screenshot**



# **Question#2**

1. Consider the following graph and write a program that finds the adjacency matrix of the given graph. Show the adjacency matrix of the graph as well as the number of vertices that share the common edges.

## **Code**

#include <iostream>

#include <string>

#include <vector>

using namespace std;

template <class T>

struct VertexNode

{

T vertex;

int cost;

};

template <class T>

struct Vertex

{

T vertex;

vector<VertexNode<T>> list;

};

template <class T>

class Graph

{

private:

Vertex<T> \*vertex;

int size, unique\_vertex;

bool IsNewVertex(T v)

{

bool flag = true;

for (int i = 0; i < unique\_vertex; i++)

{

if (vertex[i].vertex == v)

{

flag = false;

break;

}

}

return flag;

}

bool CanAddMoreVertex(char v)

{

if (IsNewVertex(v))

{

if (unique\_vertex == size)

{

return false;

}

vertex[unique\_vertex].vertex = v;

unique\_vertex++;

}

return true;

}

int GetVertex(T v)

{

int index = -1;

for (int i = 0; i < unique\_vertex; i++)

{

if (vertex[i].vertex == v)

{

index = i;

break;

}

}

return index;

}

public:

Graph()

{

vertex = NULL;

size = unique\_vertex = 0;

}

Graph(int s)

{

vertex = new Vertex<T>[s];

size = s;

unique\_vertex = 0;

}

void SetGraphLimit(int s)

{

Graph(s);

}

void addEdge(T source, T destination, int cost)

{

if (!vertex)

{

cout << "Graph size is not initialized !!!" << endl;

return;

}

if (!CanAddMoreVertex(source) || !CanAddMoreVertex(destination))

{

cout << "Limit Overflow !!! " << source << "--->" << destination << ", cannot create a link" << endl;

return;

}

#ifdef DEBUG

cout << source << "--->" << destination << "(" << cost << ")" << endl;

#endif // DEBUG

vertex[GetVertex(source)].list.push\_back({ destination, cost });

vertex[GetVertex(destination)].list.push\_back({ source, cost });

}

void displayMatrix()

{

T\* vertexNames = new T[unique\_vertex];

cout << "x ";

for (int i = 0; i < unique\_vertex; i++)

{

vertexNames[i] = vertex[i].vertex;

cout << vertexNames[i] << " ";

}

cout << endl;

int\*\* matrix = new int\* [unique\_vertex];

for (int i = 0; i < unique\_vertex; i++)

{

matrix[i] = new int[unique\_vertex];

}

for (int i = 0; i < unique\_vertex; i++)

{

for (int j = 0; j < unique\_vertex; j++)

{

matrix[i][j] = 0;

}

}

for (int i = 0; i < unique\_vertex; i++)

{

cout << vertex[i].vertex << " ";

for (int j = 0; j < unique\_vertex; j++)

{

for (auto k : vertex[i].list)

{

if (k.vertex == vertexNames[j])

{

matrix[i][j] = 1;

break;

}

}

}

for (int k = 0; k < unique\_vertex; k++)

{

cout << matrix[i][k] << " ";

}

cout << endl;

}

}

void displayEdge()

{

for (int i = 0; i < unique\_vertex; i++)

{

cout << vertex[i].vertex << ": ";

for (auto i : vertex[i].list)

{

cout << i.vertex << "(" << i.cost << "), ";

}

cout << endl;

}

}

};

int main()

{

Graph<char> graph(6);

graph.addEdge('a', 'b', 7);

graph.addEdge('a', 'c', 9);

graph.addEdge('a', 'f', 14);

graph.addEdge('b', 'd', 15);

graph.addEdge('b', 'c', 10);

graph.addEdge('c', 'd', 11);

graph.addEdge('c', 'f', 2);

graph.addEdge('d', 'e', 6);

graph.addEdge('e', 'f', 9);

cout << "Matrix: " << endl;

graph.displayMatrix();

cout << endl;

cout << "Common Edges: " << endl;

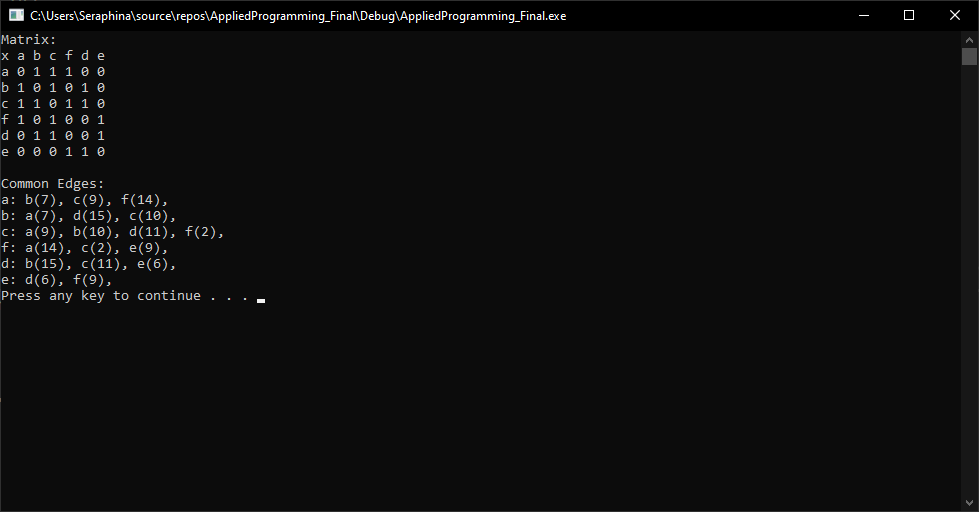
graph.displayEdge();

system("PAUSE");

return 0;

}

## **Screenshot**



1. Given an undirected Graph of N vertices, 1 to N and M edges in the form of a 2D array. The array Arr [] [], where every row consists of two numbers X and Y, which denote that there is an edge between X and Y.

## **Code**

#include <iostream>

#include <vector>

using namespace std;

int N = 8, M = 7;

const int array\_rows = 7, array\_cols = 2;

class Graph

{

private:

vector<int>\* vertex;

int\*\* matrix;

int rows, columns;

public:

Graph();

Graph(int rows, int columns);

void ConvertToAdjacencyMatrix(int arr[][array\_cols]);

void ConvertToAdjacencyList();

void DisplayAdjacencyMatrix();

void DisplayAdjacencyList();

};

Graph::Graph()

{

vertex = NULL;

rows = columns = -1;

}

Graph::Graph(int r, int c)

{

vertex = new vector<int>[r];

matrix = new int\* [r];

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

{

matrix[i] = new int[c];

}

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

{

for (int j = 0; j < c; j++)

{

matrix[i][j] = 0;

}

}

rows = r;

columns = c;

}

void Graph::ConvertToAdjacencyMatrix(int arr[][array\_cols])

{

if (rows == -1 || columns == -1)

{

cout << "Graph not Initialized..." << endl;

}

int v, lv;

for (int i = 0; i < array\_rows; i++)

{

v = arr[i][0] -1;

lv = arr[i][1] - 1;

matrix[v][lv] = 1;

matrix[lv][v] = 1;

}

}

void Graph::ConvertToAdjacencyList()

{

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

{

vertex[i].push\_back(i+1);

for (int j = 0; j < columns; j++)

{

if (matrix[i][j] == 1)

{

vertex[i].push\_back(j + 1);

}

}

}

}

void Graph::DisplayAdjacencyMatrix()

{

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

{

cout << "--";

}

cout << "\nx |";

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

{

cout << (i+1) << " ";

}

cout << endl;

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

{

cout << "--";

}

cout << endl;

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

{

cout << (i + 1) << " |";

for (int j = 0; j < columns; j++)

{

cout << matrix[i][j] << " ";

}

cout << endl;

}

}

void Graph::DisplayAdjacencyList()

{

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

{

for (auto i : vertex[i])

{

cout << i << " --> ";

}

cout << " NULL" << endl;

}

}

int main()

{

int Arr[array\_rows][array\_cols] =

{

{1, 2},

{2, 3},

{4, 5},

{1, 5},

{6, 1},

{7, 4},

{3, 8}

};

Graph graph(N, M);

cout << "================\nAdjacenty Matrix\n================" << endl << endl;

graph.ConvertToAdjacencyMatrix(Arr);

graph.DisplayAdjacencyMatrix();

cout << endl;

cout << "==============\nAdjacenty List\n==============" << endl << endl;

graph.ConvertToAdjacencyList();

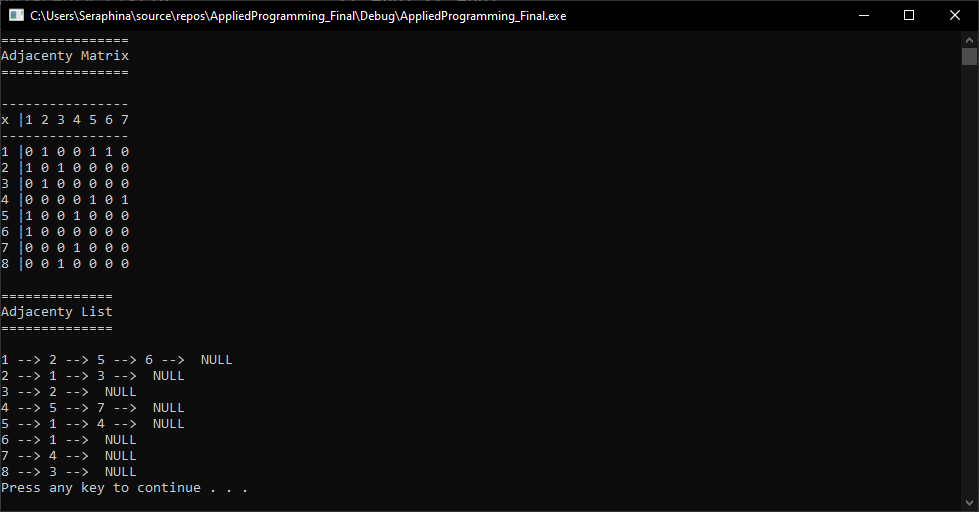
graph.DisplayAdjacencyList();

system("PAUSE");

return 0;

}

## **Screenshot**



# **Question#3**

Graphs: BFS and DFS

## **Code**

#include <iostream>

#include <vector>

#include <queue>

#include <stack>

using namespace std;

template <class T>

struct VertexNode

{

T vertex;

int cost;

};

template <class T>

struct Vertex

{

T vertex;

vector<VertexNode<T>> list;

};

//#define DEBUG // uncomment to check addEdge

template <class T>

class Graph

{

private:

Vertex<T>\* vertex;

int size, unique\_vertex;

bool IsNewVertex(T v)

{

bool flag = true;

for (int i = 0; i < unique\_vertex; i++)

{

if (vertex[i].vertex == v)

{

flag = false;

break;

}

}

return flag;

}

bool CanAddMoreVertex(T v)

{

if (IsNewVertex(v))

{

if (unique\_vertex == size)

{

return false;

}

vertex[unique\_vertex].vertex = v;

unique\_vertex++;

}

return true;

}

int GetVertexIndex(T v)

{

int index = -1;

for (int i = 0; i < unique\_vertex; i++)

{

if (vertex[i].vertex == v)

{

index = i;

break;

}

}

return index;

}

public:

Graph()

{

vertex = NULL;

size = unique\_vertex = -1;

}

Graph(int s)

{

vertex = new Vertex<T>[s];

size = s;

unique\_vertex = 0;

}

void SetVertexLimit(int s)

{

Graph(s);

}

void addEdge(T source, T destination, int cost)

{

if (!vertex)

{

cout << "Graph size is not initialized !!!" << endl;

return;

}

if (!CanAddMoreVertex(source) || !CanAddMoreVertex(destination))

{

cout << "Limit Overflow !!! " << source << "--->" << destination << ", cannot create a link" << endl;

return;

}

#ifdef DEBUG

cout << source << "--->" << destination << "(" << cost << ")" << endl;

#endif // DEBUG

vertex[GetVertexIndex(source)].list.push\_back({ destination, cost });

vertex[GetVertexIndex(destination)].list.push\_back({ destination, cost });

}

void BFS(T source, T destination)

{

queue<VertexNode<T>> q;

bool\* visited = new bool[size];

int total\_cost = 0;

int total\_count = 0;

bool goal\_found = false;

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

{

visited[i] = false;

}

int index = GetVertexIndex(source);

if (index == -1)

{

cout << "Source doesn't Exist in List !!!" << endl;

return;

}

visited[index] = true;

q.push({ vertex[index].vertex, 0 });

while (!q.empty())

{

VertexNode<T> last = q.front();

q.pop();

total\_cost += last.cost;

if (last.vertex == destination)

{

goal\_found = true;

break;

}

cout << last.vertex << " ---> ";

for (auto i : vertex[GetVertexIndex(last.vertex)].list)

{

index = GetVertexIndex(i.vertex);

if (!visited[index])

{

q.push({ i.vertex, i.cost });

visited[index] = true;

}

}

total\_count++;

}

if (goal\_found)

cout << "Goal Reached !!!" << endl;

else

cout << "Goal not Reached !!!" << endl;

std::cout << "Total Iterations: " << total\_count << endl;

std::cout << "Total Cost: " << total\_cost << endl;

}

void DFS\_util(T destination, int v, bool\* visited, int& total\_cost, int& total\_count, bool& goal\_found)

{

visited[v] = true;

total\_count++;

int index;

for (auto i : vertex[v].list)

{

index = GetVertexIndex(i.vertex);

if (visited[index] == false)

{

total\_cost += i.cost;

if (i.vertex == destination)

{

goal\_found = true;

return;

}

cout << i.vertex << " ---> ";

DFS\_util(destination, GetVertexIndex(i.vertex), visited, total\_cost, total\_count, goal\_found);

}

if (goal\_found)

{

return;

}

}

}

void DFS(T source, T destination)

{

stack<VertexNode<T>> s;

bool\* visited = new bool[size];

int total\_cost = 0;

int total\_count = 0;

bool goal\_found = false;

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

{

visited[i] = false;

}

int index = GetVertexIndex(source);

if (index == -1)

{

cout << "Source doesn't Exist in List !!!" << endl;

return;

}

for (int i = 0; i < unique\_vertex; i++)

{

if (visited[i] == false && !goal\_found)

{

cout << vertex[i].vertex << " ---> ";

DFS\_util(destination, i, visited, total\_cost, total\_count, goal\_found);

}

}

if (goal\_found)

cout << "Goal Reached !!!" << endl;

else

cout << "Goal not Reached !!!" << endl;

std::cout << "Total Iterations: " << total\_count << endl;

std::cout << "Total Cost: " << total\_cost << endl;

}

void displayEdge()

{

for (int i = 0; i < unique\_vertex; i++)

{

cout << vertex[i].vertex << ": ";

for (auto i : vertex[i].list)

{

cout << "--->" << i.vertex << "(" << i.cost << ")";

}

cout << endl;

}

}

};

int main()

{

Graph<string> graph(20);

graph.addEdge("Arad", "Zerind", 75);

graph.addEdge("Arad", "Sibiu", 140);

graph.addEdge("Arad", "Timisoara", 118);

graph.addEdge("Bucharest", "Glurgiu", 90);

graph.addEdge("Urziceni", "Glurgiu", 85);

graph.addEdge("Cralova", "Pitesti", 138);

graph.addEdge("Drobeta", "Cralova", 120);

graph.addEdge("Fagaras", "Bucharest", 211);

graph.addEdge("Hirsova", "Eforie", 86);

graph.addEdge("Iasi", "Neamt", 87);

graph.addEdge("Lugoj", "Mehadia", 70);

graph.addEdge("Mehadia", "Drobeta", 75);

graph.addEdge("Oradea", "Sibiu", 151);

graph.addEdge("Pitesti", "Bucharest", 101);

graph.addEdge("Rimnicu Vilcea", "Pitesti", 97);

graph.addEdge("Rimnicu Vilcea", "Cralova", 146);

graph.addEdge("Sibiu", "Fagaras", 99);

graph.addEdge("Sibiu", "Rimnicu Vilcea", 80);

graph.addEdge("Timisoara", "Lugoj", 111);

graph.addEdge("Urziceni", "Hirsova", 98);

graph.addEdge("Urziceni", "Vaslui", 142);

graph.addEdge("Vaslui", "Iasi", 92);

graph.addEdge("Zerind", "Oradea", 71);

cout << "====================\nBreadth-First Search\n====================" << endl;

graph.BFS("Arad", "Bucharest");

cout << endl;

cout << "==================\nDepth-First Search\n==================" << endl;

graph.DFS("Arad", "Bucharest");

cout << endl;

cout << "=============\nDisplay Graph\n=============" << endl;

graph.displayEdge();

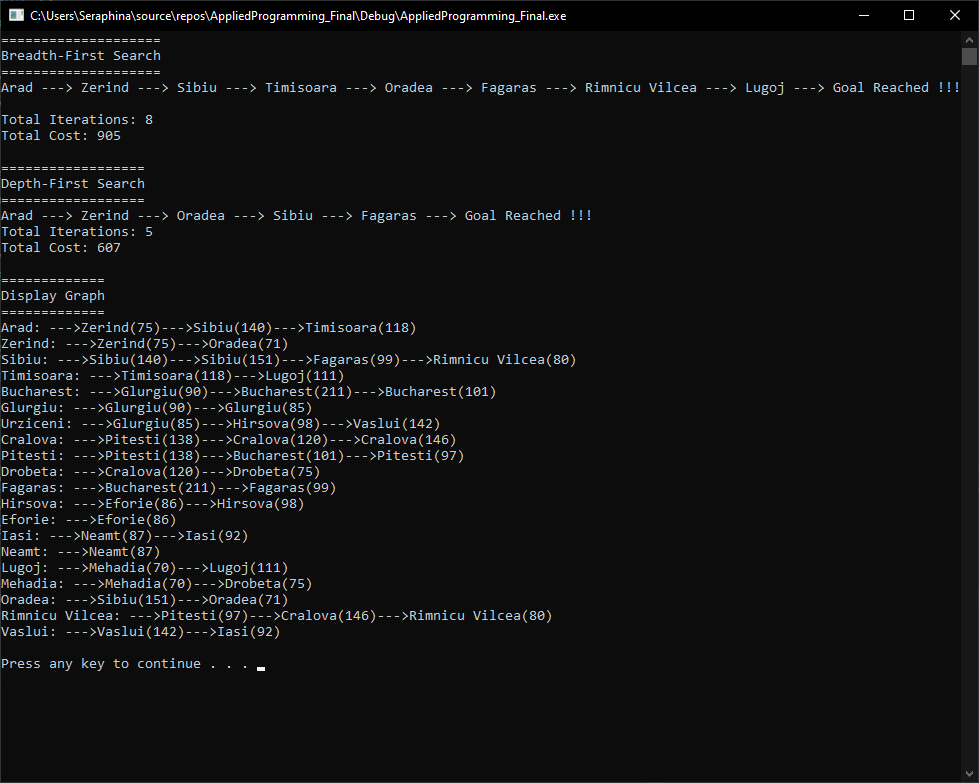
cout << endl;

system("PAUSE");

return 0;

}

## **Screenshot**



# **Question#4**

1. Heap – Using Min heap

## **Code**

#include <iostream>

#include <stdlib.h>

#include <iomanip>

using namespace std;

#define GetParent(x) (i-1)/2

#define GetLeftChild(x) (2\*i+1)

#define GetRightChild(x) (2\*i+2)

void swap(int\* lhs, int\* rhs)

{

int \*temp = lhs;

lhs = rhs;

rhs = temp;

}

class Heap

{

private:

int\* nodes;

int size, current\_size;

void Heapify(int i);

public:

Heap();

void SetLimit(int s);

void InsertKey(int key);

void DeleteRoot();

void Print();

bool IsEmpty();

int Top();

};

void Heap::Heapify(int i)

{

int lhs = GetLeftChild(i);

int rhs = GetRightChild(i);

int smallest = i;

if (lhs < current\_size && nodes[lhs] < nodes[smallest])

{

smallest = lhs;

}

if (rhs < current\_size && nodes[rhs] < nodes[smallest])

{

smallest = rhs;

}

if (smallest != i)

{

swap(nodes[i], nodes[smallest]);

Heapify(smallest);

}

}

Heap::Heap()

{

nodes = NULL;

size = current\_size = -1;

}

void Heap::SetLimit(int s)

{

nodes = new int[s];

size = s;

current\_size = 0;

}

void Heap::DeleteRoot()

{

if (current\_size == 0)

{

cout << "No element found in heap " << endl;

return;

}

int last\_element = nodes[current\_size-1];

nodes[0] = last\_element;

current\_size--;

Heapify(0);

}

void Heap::InsertKey(int key)

{

if (size == -1)

{

cout << "Heap::SetLimit(size), where `size` is -1..." << endl;

return;

}

if (size == current\_size )

{

cout << "Unable to insert " << key << " !!!" << endl;

return;

}

nodes[current\_size] = key;

for (int i = current\_size; i != 0 && nodes[GetParent(i)] > nodes[i]; i = GetParent(i))

{

swap(nodes[i], nodes[GetParent(i)]);

}

current\_size++;

}

void Heap::Print()

{

cout << "[ ";

for (int i = 0; i < current\_size; i++)

{

cout << nodes[i] << " ";

}

cout << "]" << endl;

}

bool Heap::IsEmpty()

{

return current\_size == 0;

}

int Heap::Top()

{

return nodes[0];

}

int main()

{

Heap heap;

int size = 0;

do

{

cout << "Enter Size: ";

cin >> size;

} while (size < 0);

heap.SetLimit(size);

#define MAXIMUM\_NUMBER 1000

cout << "Generating " << size << " random numbers" << endl;

int x;

srand(time(NULL));

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

{

x = rand() % MAXIMUM\_NUMBER + 1;

cout << x << " ";

heap.InsertKey(x);

}

cout << endl;

cout << "Heap::Print() " << endl;

heap.Print();

cout << endl;

while (!heap.IsEmpty())

{

cout << "Deleting: " << heap.Top() << endl;

heap.DeleteRoot();

}

cout << endl;

cout << "Heap::Print() " << endl;

heap.Print();

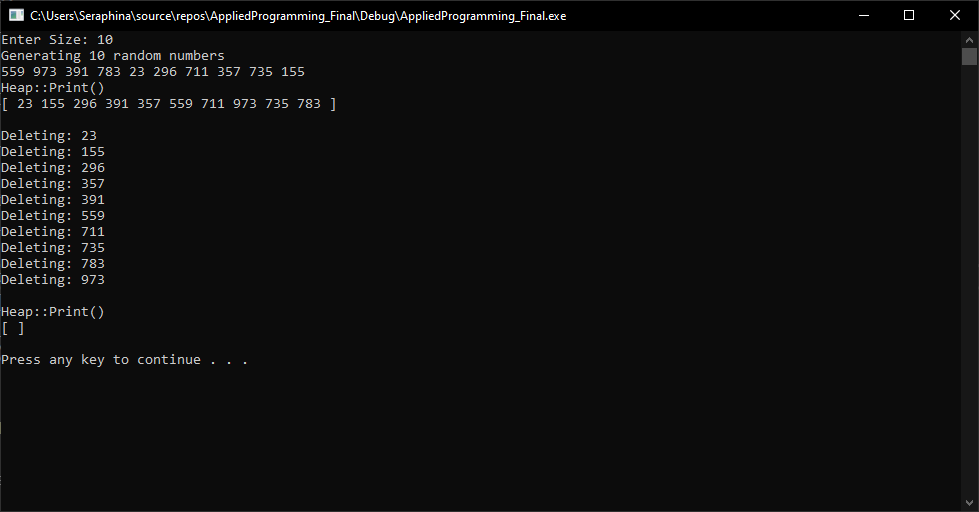
cout << endl;

system("PAUSE");

return 0;

}

## **Screenshot**



1. Heap – Using Max heap

## **Code**

#include <iostream>

#include <stdlib.h>

#include <iomanip>

using namespace std;

#define GetParent(x) (i-1)/2

#define GetLeftChild(x) (2\*i+1)

#define GetRightChild(x) (2\*i+2)

void swap(int\* lhs, int\* rhs)

{

int \*temp = lhs;

lhs = rhs;

rhs = temp;

}

class Heap

{

private:

int\* nodes;

int size, current\_size;

void Heapify(int i);

public:

Heap();

void SetLimit(int s);

void InsertKey(int key);

void DeleteRoot();

void Print();

bool IsEmpty();

int Top();

};

void Heap::Heapify(int i)

{

int lhs = GetLeftChild(i);

int rhs = GetRightChild(i);

int smallest = i;

if (lhs < current\_size && nodes[lhs] > nodes[smallest])

{

smallest = lhs;

}

if (rhs < current\_size && nodes[rhs] > nodes[smallest])

{

smallest = rhs;

}

if (smallest != i)

{

swap(nodes[i], nodes[smallest]);

Heapify(smallest);

}

}

Heap::Heap()

{

nodes = NULL;

size = current\_size = -1;

}

void Heap::SetLimit(int s)

{

nodes = new int[s];

size = s;

current\_size = 0;

}

void Heap::DeleteRoot()

{

if (current\_size == 0)

{

cout << "No element found in heap " << endl;

return;

}

int last\_element = nodes[current\_size-1];

nodes[0] = last\_element;

current\_size--;

Heapify(0);

}

void Heap::InsertKey(int key)

{

if (size == -1)

{

cout << "Heap::SetLimit(size), where `size` is -1..." << endl;

return;

}

if (size == current\_size )

{

cout << "Unable to insert " << key << " !!!" << endl;

return;

}

nodes[current\_size] = key;

for (int i = current\_size; i != 0 && nodes[GetParent(i)] < nodes[i]; i = GetParent(i))

{

swap(nodes[i], nodes[GetParent(i)]);

}

current\_size++;

}

void Heap::Print()

{

cout << "[ ";

for (int i = 0; i < current\_size; i++)

{

cout << nodes[i] << " ";

}

cout << "]" << endl;

}

bool Heap::IsEmpty()

{

return current\_size == 0;

}

int Heap::Top()

{

return nodes[0];

}

int main()

{

Heap heap;

int size = 0;

do

{

cout << "Enter Size: ";

cin >> size;

} while (size < 0);

heap.SetLimit(size);

#define MAXIMUM\_NUMBER 1000

cout << "Generating " << size << " random numbers" << endl;

int x;

srand(time(NULL));

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

{

x = rand() % MAXIMUM\_NUMBER + 1;

cout << x << " ";

heap.InsertKey(x);

}

cout << endl;

cout << "Heap::Print() " << endl;

heap.Print();

cout << endl;

while (!heap.IsEmpty())

{

cout << "Deleting: " << heap.Top() << endl;

heap.DeleteRoot();

}

cout << endl;

cout << "Heap::Print() " << endl;

heap.Print();

cout << endl;

system("PAUSE");

return 0;

}

## **Screenshot**

