CSE225L – Data Structures and Algorithms Lab
Lab 15
Graph

In today's lab we will design and implement the Graph ADT.

graphtype.h

```
#ifndef GRAPHTYPE_H_INCLUDED
#define GRAPHTYPE_H_INCLUDED
#include "stacktype.h"
#include "quetype.h"
template<class VertexType>
class GraphType
{
    public:
        GraphType();
        GraphType(int maxV);
        ~GraphType();
        void MakeEmpty();
        bool IsEmpty();
        bool IsFull();
        void AddVertex(VertexType);
        void AddEdge(VertexType,
VertexType, int);
        int WeightIs(VertexType,
VertexType);
        void GetToVertices(VertexType,
QueType<VertexType>&);
        void ClearMarks();
        void MarkVertex(VertexType);
        bool IsMarked(VertexType);
        void DepthFirstSearch(VertexType,
VertexType);
        void BreadthFirstSearch(VertexType,
VertexType);
    private:
        int numVertices;
        int maxVertices;
        VertexType* vertices;
        int **edges;
        bool* marks;
};
#endif // GRAPHTYPE_H_INCLUDED
```

heaptype.cpp

```
#include "graphtype.h"
#include "stacktype.cpp"
#include "quetype.cpp"
#include <iostream>
using namespace std;
const int NULL_EDGE = 0;

template<class VertexType>
GraphType<VertexType>::GraphType()
{
    numVertices = 0;
    maxVertices = 50;
    vertices = new VertexType[50];
    edges = new int*[50];
    for(int i=0;i<50;i++)
        edges[i] = new int [50];
    marks = new bool[50];
}
template<class VertexType>
GraphType<VertexType>::GraphType(int maxV)
{
    numVertices = 0;
    maxVertices = maxV;
    vertices = new VertexType[maxV];
    edges = new int*[maxV];
    for(int i=0;i<maxV;i++)
        edges[i] = new int [maxV];
    marks = new bool[maxV];
}
template<class VertexType>
GraphType<VertexType>::~GraphType()
{
    delete [] vertices;
    delete [] marks;
    for(int i=0;i<maxVertices;i++)
        delete [] edges[i];
    delete [] edges;
}
template<class VertexType>
void GraphType<VertexType>::MakeEmpty()
{
    numVertices = 0;
}
template<class VertexType>
bool GraphType<VertexType>::IsEmpty()
{
    return (numVertices == 0);
}
template<class VertexType>
bool GraphType<VertexType>::IsFull()
{
    return (numVertices == maxVertices);
}
template<class VertexType>
void GraphType<VertexType>::AddVertex(VertexType
vertex)
{
    vertices[numVertices] = vertex;
    for (int index=0; index<numVertices; index++)
    {
        edges[numVertices][index] = NULL_EDGE;
        edges[index][numVertices] = NULL_EDGE;
    }
    numVertices++;
}
template<class VertexType>
int IndexIs(VertexType* vertices, VertexType
vertex)
{
    int index = 0;
    while (!(vertex == vertices[index]))
        index++;
    return index;
}
template<class VertexType>
void GraphType<VertexType>::ClearMarks()
{
    for(int i=0; i<maxVertices; i++)
        marks[i] = false;
}
template<class VertexType>
void GraphType<VertexType>::MarkVertex(VertexType
vertex)
{
    int index = IndexIs(vertices, vertex);
    marks[index] = true;
}
template<class VertexType>
bool GraphType<VertexType>::IsMarked(VertexType
vertex)
{
    int index = IndexIs(vertices, vertex);
    return marks[index];
}
```

```
template<class VertexType>
void GraphType<VertexType>::AddEdge(VertexType fromVertex, VertexType toVertex, int weight)
{
    int row = IndexIs(vertices, fromVertex);
    int col= IndexIs(vertices, toVertex);
    edges[row][col] = weight;
}
template<class VertexType>
int GraphType<VertexType>::WeightIs(VertexType fromVertex, VertexType toVertex)
{
    int row = IndexIs(vertices, fromVertex);
    int col= IndexIs(vertices, toVertex);
    return edges[row][col];
}
template<class VertexType>
void GraphType<VertexType>::GetToVertices(VertexType vertex, QueType<VertexType>& adjVertices)
{
    int fromIndex, toIndex;
    fromIndex = IndexIs(vertices, vertex);
    for (toIndex = 0; toIndex < numVertices; toIndex++)
        if (edges[fromIndex][toIndex] != NULL_EDGE)
            adjVertices.Enqueue(vertices[toIndex]);
}
```

```
template<class VertexType>
void
GraphType<VertexType>::DepthFirstSearch(Vertex
Type startVertex, VertexType endVertex)
{
    StackType<VertexType> stack;
    QueType<VertexType> vertexQ;
    bool found = false;
    VertexType vertex, item;

    ClearMarks();
    stack.Push(startVertex);
    do
    {
        vertex = stack.Top();
        stack.Pop();
        if (vertex == endVertex)
        {
            cout << vertex << " ";
            found = true;
        }
        else
        {
            if (!IsMarked(vertex))
            {
                MarkVertex(vertex);
                cout << vertex << " ";
                GetToVertices(vertex,vertexQ);
                while (!vertexQ.IsEmpty())
                {
                    vertexQ.Dequeue(item);
                    if (!IsMarked(item))
                        stack.Push(item);
                }
            }
        }
    } while (!stack.IsEmpty() && !found);
    cout << endl;
    if (!found)
        cout << "Path not found." << endl;
}
```

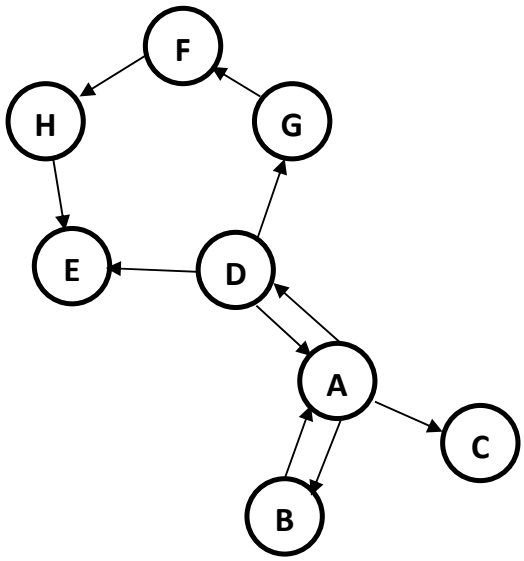
```
template<class VertexType>
void
GraphType<VertexType>::BreadthFirstSearch(Vertex
Type startVertex, VertexType endVertex)
{
    QueType<VertexType> queue;
    QueType<VertexType> vertexQ;

    bool found = false;
    VertexType vertex, item;

    ClearMarks();
    queue.Enqueue(startVertex);
    do
    {
        queue.Dequeue(vertex);
        if (vertex == endVertex)
        {
            cout  << vertex << " ";
            found = true;
        }
        else
        {
            if (!IsMarked(vertex))
            {
                MarkVertex(vertex);
                cout  << vertex << " ";
                GetToVertices(vertex, vertexQ);

                while (!vertexQ.IsEmpty())
                {
                    vertexQ.Dequeue(item);
                    if (!IsMarked(item))
                        queue.Enqueue(item);
                }
            }
        }
    } while (!queue.IsEmpty() && !found);
    cout << endl;
    if (!found)
        cout << "Path not found." << endl;
}
```

Now generate the **Driver file (main.cpp)** where you perform the following tasks:

Operation to Be Tested and Description of Action	Input Values	Expected Output
• Generate the following graph. Assume that all edge costs are 1. 		
• Outdegree of a particular vertex in a graph is the number of edges going out from that vertex to other vertices. For instance the outdegree of vertex **B** in the above graph is 1. Add a member function `OutDegree` to the `GraphType` class which returns the outdegree of a given vertex. `int  OutDegree(VertexType v);`		
• Add a member function to the class which determines if there is an edge between two vertices. `bool FoundEdge(VertexType u, VertexType v);`		
• Print the outdegree of the vertex **D**.		3
• Print if there is an edge between vertices **A** and **D**.		There is an edge.
• Print if there is an edge between vertices **B** and **D**.		There is no edge.
• Use depth first search in order to find if there is a path from **B** to **E**.		B A D G F H E
• Use depth first search in order to find if there is a path from **E** to **B**.		E Path not found.
• Use breadth first search in order to find if there is a path from **B** to **E**.		B A C D E
• Use breadth first search in order to find if there is a path from **E** to **B**.		E Path not found.
• Modify the `BreadthFirstSearch` function so that it also prints the length of the shortest path between two vertices.		
• Determine the length of the shortest path from **B** to **E**.		3