```
// Raul Martinez
// raul.martinez01@utrgv.edu

#include <iostream>
#include <string>
#include <list> //the stl's doubly linked list
#include <vector>
#include <ctime>
#include <cmath>
#include "minHeap.h"

using namespace std;

class directedGraph
{
private:
    class vertex;
    class edge;

    class vertex
    {
    public:
        int data;
        int ind;
        double weight;

        list<edge*> adjList;
        vertex * pred;

        vertex(int x)
        {
            pred = NULL;
            data = x;
            weight = 0;
        }
    };

    class edge
    {
    public:
        double weight;
        vertex * start;
        vertex * end;

        edge(vertex * v, vertex * u, double w)
        {
            start = v;
            end = u;
            weight = w;
        }
    };

public:
    vector<vertex*> vertexList;
    vector<vertex*> predList;
    minHeap<vertex*> mh;
```

```
        ~directedGraph()
        {
            //free all the (dynamically allocated) vertices.
            for (vector<vertex*>::iterator itr = vertexList.begin(); itr !=
                vertexList.end(); itr++)
            {
                delete (*itr);
            }
        }

        //add a new vertex with data value x to the graph
        void addVertex(int x)
        {
            vertexList.push_back(new vertex(x));
        }

        //add a directed edge going from x to y
        void addDirectedEdge(int x, int y, double w)
        {
            vertex * u = vertexList[x];
            vertex * v = vertexList[y];

            v->adjList.push_back(new edge(v, u, w));
        }

        // Runs the Dijkstra alogrithm
        void dijkstra(vertex * start)
        {
            // set weights from all vertices to INFINITY
            for (int i = 0; i < vertexList.size(); i++)
            {
                vertexList[i]->weight=INFINITY;
                vertexList[i]->pred = NULL;
            }

            start->weight = 0;

            // Insert vertices into minHeap
            for (vector<vertex*>::iterator itr = vertexList.begin(); itr !=
                vertexList.end(); itr++)
            {
                mh.insert((*itr));
            }

            // Relax edges using minHeap
            while (!mh.empty())
            {
                vertex * v = mh.extractMin();

                for (list<edge*>::iterator itr = v->adjList.begin(); itr != v->
                    adjList.end(); itr++)
                {
                    relax(*itr);
                }
            }
```

```
    }

    // Relax method for Dijkstra
    void relax(edge * e)
    {
        vertex * v = e->start;
        vertex * u = e->end;

        if (v->weight + e->weight < u->weight)
        {
            u->weight = v->weight + e->weight;
            u->pred = v;
            mh.bubbleUp(u->ind);
        }
    }

    // Uses Dijkstra algorithm to find shortest path
    void shortestPath(int x, int y)
    {
        int cost = 0;
        vertex * a = vertexList[x];
        vertex * b = vertexList[y];

        dijkstra(a);

        // Adds up cost of shortest path
        while (b->pred != NULL)
        {
            predList.push_back(b);
            cost = cost + abs(b->weight - b->pred->weight);
            b = b->pred;
        }

        // Print out shortest path
        while (!predList.empty())
        {
            vertex * s = predList.back();
            cout << s->data << ", ";
            predList.pop_back();
        }

        cout << endl << "Cost: "<< cost << endl;
    }
};
```