Weighted Graph Class Implementation
Cpt S 223 Homework Assignment
Washington State University

Assignment Instructions (read all instructions carefully before you write any code):

For this assignment, you will implement a Graph class for a directed, weighted graph. Nodes in the graph will have a string for their name. Note that the edge declarations in the input commands will require you to add two edges (see description of "set" command for more information).

Set up a main function to read commands from stdin:

In this assignment, you will read simple commands from standard input. You will display your name and ID number on a single line to start with and then enter the command processing loop. There will be one command per line, so your main function (pseudo-code) will be something like:

```
cout << "Student Name, ID#\n";
while (true)
{
  string cmd = ReadLine();
  if (cmd.StartsWith("//")) { continue; }
  else if (cmd.StartsWith("cmd1name")) { HandleCmd1(); }
  else if (cmd.StartsWith("cmd2name")) { HandleCmd2(); }
  else if (cmd.StartsWith("quit")) { break; }
}
cout << "Done\n";
```

Most command strings will be much like C function calls in terms of syntax. You must support the following commands:

// : This is not actually a command that requires any action, but you must identify comment lines that start with the double forward slash during command processing. These are meant to allow for description of the commands and should just be ignored.

set(node1,node2,node3,...|n1,n2,w1;n3,n4,w2) : Sets the current graph data, completely clearing any prior graph data. You should be creating a new graph object (and deleting the old) every time you process this command. You may assume that this command will be called at least once before any of the other commands appear (but comment lines may occur before it).

The set command parameter has two main pieces, node declarations first and then edge declarations, with the pipe character (|) in between them. When processing this command, take what's between the parentheses and split based on the pipe character.

The first piece consists of node declarations.

- Node names are separated by commas, but note that there is no comma after the very last node in the list (there will be a pipe character instead).
- Node names may consist of any alpha-numeric characters as well as spaces.
- You may assume that the same node name does not appear more than once.

The second piece consists of edge declarations.

- Each edge declaration is of the form: **node1_name,node2_name,weight**
- Edge declarations are separated by semicolons, but note that there is no semicolon after the very last edge in the list (there will be a closing paren. from the set command).
- There are three pieces for an edge declaration and these pieces are separated by commas.
- Important: The edge declaration requires you to add 2 edges to your graph, since your graph is directed. So add node1,node2 as well as node2,node1, both with the same weight.
- Parse the weights as double values.
- Each node name in an edge declaration will exactly match a node name from the nodes list. Recall that it's valid for a graph to have nodes without any edges connected to them, so there may be nodes in the node list that never are referenced in the list of edges.
- The edge list may be empty. In this case, you'd see the pipe character immediately followed by the closing parentheses character.

There is no output on screen produced by this command.

degree(node_name) : Displays the degree for the specified node as an integer. This value is displayed on its own line with no spaces before or after. Recall that for a directed graph this is the sum of the in-degree and out-degree. Display -1 if the node does not exist in the graph.

shortest_path_length(start_node,end_node) : Computes the length of the shortest path from the start node to the end node. Recall that a path length is the sum of edge weights as you traverse from start to end. The length is rounded to the nearest integer and displayed as such on its own line (this is mainly to eliminate the possibilities of differences in number of decimal places in the output). If no path exists between the specified start and ending nodes, then “no path” (without quotes, all lower-case) is displayed on its own line.

shortest_path(start_node,end_node) : Displays the sequence of nodes for the shortest path from the start to end node on one line. The path is displayed as a sequence of node names, with a comma after each, including the last one on the line. If there are multiple shortest paths then arbitrarily choose one. The grading process will involve using nodes for which there is only 1 shortest path. If no path exists between the specified start and ending nodes, then “no path” (without quotes, all lower-case) is displayed on its own line.

quit() : Breaks the command processing loop and terminates your program. Display the line “Done” as the very last thing before returning from main.

Notes:

- See in input/output text files included in the zip file to get an idea of exactly what the commands look like before you start writing the code.
- Remember to write additional test cases of your own.
- Put comments in your code! Give a brief explanation for each function at a minimum.
- Look at the sample input/output files in the zip to get an idea of what the commands and their corresponding outputs look like. As usual, develop your own additional test cases on top of these, since they are samples only and not meant to rigorously test your code.
- Your code must compile and run properly using g++ on Linux. As usual, test with I/O redirection: **./yourappname < inFileName.txt**
  and compare with the expected output.