CSE 6010
Assignment 3
Modular Graph Functions

Initial Submission Due Date: 11:59pm on Thursday, October 3
Final Submission Due Date: 11:59pm on Thursday, October 10
Peer review assignment due Saturday, October 19 [2 extra days to account for fall break] will be posted separately
Submit codes as described herein to Gradescope through Canvas
48-hour grace period applies to all deadlines

In this assignment, you will develop functions in C to create and work with a directed graph that can be changed dynamically as the program runs. The idea will be to define a series of structs and functions to create and delete the graph, add and delete graph nodes and edges, and print graph information. The functions and necessary structs will be declared in a header (.h) file, defined in a related .c file, and called by the main function, which will reside in a separate .c file.

For this assignment, you will be given a starting graph.h file along with a sample main.c file. Your task will be to modify graph.c to implement the functions. Your graph.h and graph.c files will be tested with variations of main.c to assess the full functionality of your implementation.

To receive full credit, your code must be well structured and documented so that it is easy to understand. Be sure to include comments that explain your code statements and structure.

Main ideas:

用linked list储存信息

- The graph information must be stored using linked lists. Specifically, you will use an adjacency matrix representation of the graph to store the edges that can be reached from each nodes; because the number of nodes is not known in advance, you will also use a linked list to represent the list of nodes.
- The program will rely on (at least) three data structures, as listed below. You may choose to add additional information. Define these structs in graph.h; the initial graph.h file will only include enough information for main.c to properly interface with the Graph struct (basically, just the name).
    - A graph data structure that includes, at a minimum, a pointer to the first node in the node list of the graph.
    - A node data structure that includes, at a minimum, the node ID (an integer), a pointer to the relevant edge information for nodes that can be reached from that node (adjacency list), and a pointer to the next node.
    - An edge data structure, which will be part of the adjacency list of the "from" node associated with that edge, and which includes, at a minimum, the "to" node ID (an integer) for the edge, the edge weight (of type double), and a pointer to the next node on that adjacency list.
- The program will test functionality for creating and deleting a graph, adding and deleting a node, adding and deleting an edge, printing lists of nodes and edges, and

returning the numbers of nodes and edges. You likely will want to create some helper functions.

- Note that correctly deleting a node requires not only removing it from the list of nodes but also deleting any edges that arrive at that node.
- Many functions are declared with a return type of void (that is, they do not return a value). If it is helpful in your function definitions, it is ok to change the return type (e.g., to int). Note that the main function will not store the return value; however, it will still work properly without doing so.

Required functions (will be called directly from main.c):

1. Graph* initializeGraph();
   Purpose: Return a pointer to the Graph data structure.
2. void deleteGraph(Graph* myGraph);
   Purpose: Delete the graph and free all associated memory.
3. void addNode(Graph* myGraph, int myNodeID);
   Purpose: Add a node to the graph with the specified nodeID.
4. void addEdge(Graph* myGraph, int myFromNodeID, int myToNodeID, double myEdgeweight);
   Purpose: Add an edge to the graph with specified edge weight. The edge originates at the node with node ID myFromNodeID and terminates at the node with node ID myToNodeID.
5. void deleteNode(Graph* myGraph, int myNodeID);
   Delete the node with specified node ID. Be sure to also delete all edges that originate from or terminate at the node. Free all associated memory.
6. void deleteEdge(Graph* myGraph, int myFromNodeID, int myToNodeID);
   Delete the edge from the node with node ID myFromNodeID to the node with node ID myToNodeID, and free any associated memory.
7. int calcNumNodes(Graph* myGraph);
   Return the number of nodes in the graph.
8. int calcNumEdges(Graph* myGraph);
   Return the number of edges in the graph.
9. void printNodeList(Graph* myGraph);
   On a single line, print the node IDs separated by a space. Finish with a newline character. (The order in which the nodes are printed does not matter; also, it does not matter if you print a space after the final node ID.)
10. void printEdgeList(Graph* myGraph);
    Print all edges in the graph by visiting each node and all of the nodes that can be reached from that node. Each edge should be written using the format string "(%d, %d, %f)\n", where the three arguments of the ordered triple are the node ID of the origin ("from") node, the node ID of the destination ("to") node, and the edge weight. (The order in which the edges are listed does not matter, as long as the information for each edge is in the proper order.)

Required structs:

You will need to define at least three types of structs. The struct members are not fully specified beyond what is needed for main.c to work with the graph. You will need to complete the definition of the Graph struct and define the additional structs, including determining the struct members and their names.

1. A graph struct that includes, at a minimum, a pointer to the first node in the node list of the graph. The preliminary graph.h file includes the beginning of a definition for this struct, which includes a typdef for this struct with name Graph.
2. A struct for storing node information that includes, at a minimum, the node ID (an integer) and a pointer to the next node. You will need to completely specify this struct.
3. A struct for storing edge information that will be part of the adjacency list of the "from" node associated with that edge. It should include, at a minimum, the "to" node ID (an integer) for the edge, the edge weight (of type double), and a pointer to the next node on that adjacency list. You will need to completely specify this struct.

Recommended additional functionality:

- Functions:
  - You likely will need to include some helper functions for creating new structs.
- Others:
  - You may find some functionality easier to implement by extending the structs to include additional information. This is fine, as long as your code works with the sample main.c file.
  - We recommend that you use a dummy node for the first node on the linked list for the nodes and on the linked list for each adjacency list. (See lecture notes from August 20 for more on the advantages of a dummy node.) If you choose to use it, you may implement the dummy node in any way that is consistent with proper functionality.
  - You may choose whether to sort the linked lists. (The autograder will sort any output, so it will not matter how you store/print out the lists.)
- Behavior for special cases: To keep things simple, if your code encounters either of the following cases, exit the program by executing the command `exit(1)`. Before exiting, be sure to free all memory to avoid memory leaks.
  - Delete a node that is not part of a non-empty graph.
  - Delete an edge that is not part of a non-empty graph.

Things we will not explicitly test:

- Duplicate node: Adding a node that is already part of the node list for the graph.
- Duplicate edge: Adding an edge whose "to" and "from" node IDs correspond to an edge that is already in the graph (whether the edge weight is the same or not).
- Printing the lists of vertices or edges for an empty graph.

Initial submission (worth 1 point): Submit your code to the Initial Submission on Gradescope. Your initial submission will not be graded for correctness or even tested but rather will be graded based on the appearance of a good-faith effort to complete the majority of the assignment. Any time you wish to test your code for correctness, please submit it to the Final Submission on Gradescope.

Final submission (worth 11 points): Submit your code to the Final Submission on Gradescope. The required files are **graph.c** (which you will specify entirely) and **graph.h** (which you will extend by defining data structures). Do not zip the files or upload a directory; instead, submit these two files directly.

Some hints:

- Start early!
- Identify a logical sequence for implementing the desired functionality. Then implement one piece at a time and verify each piece works properly before proceeding to implement the next. Working through the tests outlined below in sequence may be helpful.
- Be sure to test variations: e.g., delete a node/edge that is located first, last, and in the middle of the corresponding list.
- None of the test cases should take long to run if implemented correctly. If running your code takes more than a few seconds, you can force it to stop with the keyboard Ctrl+C or command C-c (hold control and press the "c" key).
- Note that calling `free()` releases the memory associated with the item pointed to by the pointer. The pointer remains a declared variable and can be reused. (This may be helpful to keep in mind, for example, for deleting the graph and freeing all associated memory.

Grading

Program correctness will be assessed using the autograder on Gradescope; valgrind will be used to detect memory errors, including failure to free all allocated memory. Detailed grading specifications are described below. Numbers in parentheses indicate the total points available for a specific component.

1. **Initial submission (1)**: Graded manually. The point is earned for making meaningful progress toward completing the assignment. The program does not need to compile or run correctly at this stage.
2. **Autograder (8)**
   a. **Test 0: Compile the program (0)**: This test is not worth any points, but it will show the output if errors are encountered compiling your program on the autograder. This output should be helpful if your code compiles on your machine but not on the autograder. Fix these errors before moving on to anything else!

b. **Test 1: Creating and deleting a graph without edges (0.75)**: This test adds nodes to an initially empty graph, prints the node list, prints the number of nodes and number of edges, then deletes the graph. (0.5 for correct behavior + 0.25 for valgrind)
c. **Test 2: Adding and deleting nodes to/from a graph without edges (1.25)**: This test adds nodes to an initially empty graph; tests a combination of node deletions along with further node additions, interspersed with statements to print the node list and/or number of nodes and number of edges; and finally deletes the graph. (1 for correct behavior + 0.25 for valgrind)
d. **Test 3: Adding edges (1.25)**: This test adds nodes to an initially empty graph, adds edges, prints the node list, prints the number of nodes and number of edges, then deletes the graph. (1 for correct behavior + 0.25 for valgrind)
e. **Test 4: Deleting edges (1.75)**: This test adds nodes to an initially empty graph; adds edges; tests a combination of edge deletions along with further node and edge additions, interspersed with statements to print the node list, edge list, and/or number of nodes and number of edges; and finally deletes the graph. (1.5 for correct behavior + 0.25 for valgrind)
f. **Test 5: Testing delete behavior when an item does not exist (0.75)**: This test comprises three separate tests.
    i. Test 5a (0.15) attempts to delete a graph with 0 nodes. (0.1 for correct behavior + 0.05 for valgrind). In this case, the program should not exit with an error code, but should ensure that any memory associated with the empty graph structure has been freed.
    ii. Test 5b (0.3) adds nodes and edges, then attempts to delete a node that is not part of the graph. Free all memory, then call `exit(1)`. (0.2 for correct behavior + 0.1 for valgrind)
    iii. Test 5c (0.3) adds nodes and edges, then attempts to delete an edge that is not part of the graph. Free all memory, then call `exit(1)`. (0.2 for correct behavior + 0.1 for valgrind)
g. **Test 6: Deleting nodes (2.25)**: This test adds nodes to an initially empty graph; adds edges; tests a combination of node and edge deletions along with further node and edge additions, interspersed with statements to print the node list, edge list, and/or number of nodes and number of edges; and finally deletes the graph. (2 for correct behavior + 0.25 for valgrind)

3. **Program structure and implementation (1.5)**: Graded manually. Points are earned for implementing the algorithms and data structures correctly as described in the assignment and making reasonable decisions when designing your program.
4. **Program style and documentation (1.5)**: Graded manually. Points are earned for using abstractions to avoid redundant code and making your program readable by using appropriate comments, variable names, whitespace, indentation, etc. Write your code so someone else can easily follow what is going on.

The autograder will run immediately when the program is submitted and should present the results quickly. The manually graded components of the assignment will not be available until after the grades are published. Autograder feedback will only be available with the final submission for the assignment, as the initial submission will not be graded for correctness.