Date due: Thursday, March 30, 11:59:59 p.m.

1 Introduction and purpose

In this project you will write a program that stores graphs. Graphs were covered in CMSC 132, so you should be familiar with them and the different representations that can be used for storing them, so we don't explain these here. You can refer to your 132 materials to refresh your memory of graphs if you need to. Your program **must** use **only** dynamically–allocated memory to store graphs. The purpose of the project is to get more experience using memory allocation, and to create a more complex linked data structure in C. Be sure to read the data structure requirements in Section 3 **carefully**. You will have to write a makefile that will compile your code with all of the public tests, so you will get more practice with makefiles.

This project is a more difficult project again– perhaps more challenging than Project #3– so start working hard on it **right away**. Note that the next project will rely upon this one.

Before asking for debugging help in the TAs' office hours, you must know how to use gdb and have already used gdb to throughly debug your code. Otherwise the TAs will not help figure out any problems with your code.

You will **lose credit** if any of your functions cast the return value of the memory allocation functions.

Due to the size of the course it is not feasible for us to be able to provide project information or help via email/ELMS messages, so we will be unable to answer such questions. You are welcome to ask any questions verbally during the TAs' office hours.

1.1 Extra credit and number of submissions

You can again get extra credit for this project. If you make **only one submission** that passes **all the public tests** you will get **3 extra–credit bonus points**. And if your single submission is made **at least 48 hours before the on–time deadline** you will get **3 additional extra credit bonus points**, for 6 extra credit points total. (Obviously you can't get the second 3 extra credit points if you don't get the first 3, but you can get the first 3 without getting these second 3.)

However, as before, if you make too many submissions you may again lose credit. To avoid making submissions that don't compile or don't pass all of the public tests you should compile the tests, run them, check your output, and fix any problems, all **before** submitting. **And** you should **write your makefile first, and test it before submitting**, as discussed below.

(If for some reason your code passes all the public tests on Grace, but doesn't work when you submit it, so you have to submit more than once– **and this is not due to an error on your part or a bug in your code or a problem with your makefile**– you can talk with me verbally in office hours about receiving the extra credit despite having more than one submission.)

2 Graphs in this project

Graphs in this project are directed. (An undirected graph can be simulated by a directed graph where every undirected edge is replaced by two directed edges, one in each direction.) Henceforth, the term *edge* means a **directed** edge. Graphs in this project will have the following properties:

- There can be at most one edge from a vertex x to a vertex y. (Of course, there can be another edge from y to x.)
- There can be at most one edge from a vertex x to itself. (The edge would be both outgoing and incoming for x.)
- A vertex can have no edges at all, either incoming or outgoing.
- Every vertex has a name, which is a (possibly empty) string with any length. Hence the name must be stored in dynamically–allocated memory.
- No two vertices in the same graph can have the same name, but vertices in different graphs can have the same names.
- Every edge has a nonnegative integer weight or cost (which could be 0).
- There is **no limit** to the number of vertices or to the number of edges that graphs can have. Hence vertices and edges must be stored in dynamically–allocated memory.

3 Data structure requirements

You will create your own data structure for storing the components of graphs. Our provided header file `string-graph.h` has the prototypes of the required functions, which use an undefined type `String_graph`. (It is named `String_graph` because it is a graph in which the data associated with vertices is character strings.) `string-graph.h` includes a header file `string-graph-datastructure.h` that is not provided. You must write this file (with its name spelled **exactly** as shown), containing your definition of `String_graph`. Please read carefully because you would lose **significant** credit if your data structure in `string-graph-datastructure.h` does not obey these requirements:

- It **must** use **only** dynamically–allocated data structures to store graphs.

  Some examples of dynamically allocated data structures are linked lists, binary search trees, and dynamically–allocated arrays, whose size is not known at compile time.

- You **must** use some type of dynamically–allocated data structure to store the vertices of graphs.

  Each vertex must have a pointer (or pointers) to a separate dynamically–allocated data structure that stores the outgoing edges of the vertex. In particular, you **must** represent graphs using either an adjacency list, an adjacency set, or an adjacency map. You **may not** use an adjacency matrix, an edge list, or an incidence matrix, some of which may not even have been covered in CMSC 132. You **may not** use any graph representation other than one of these three. This also means you **may not** invent your own graph representation. You **must** use either an adjacency list, an adjacency set, or an adjacency map.

- You must use **at least one binary search tree or one linked list** in storing graphs. You can use more than one of these. You can use other dynamically–allocated data structures, such as dynamically–allocated arrays.

- You **may not** use any fixed–size arrays **anywhere** in your entire graph implementation. This means there should not be **any square braces (`[]`) anywhere in your header file `string-graph-datastructure.h`**. All strings used for vertex names must use dynamically–allocated memory.

The adjacency list, set, and map graph representations were covered in CMSC 132. You can refer to your 132 materials to refresh your memory of them if needed. Of course lists, sets, and maps are already present in the Java libraries, but not in C. However, CMSC 132 discussed (to varying extents, in some cases in significant detail) how to implement all of these. Note that lecture this semester showed a skeleton of a graph representation but it was incomplete because it didn't show how the collection of vertices were being stored. You can base your representation upon that, for example by storing the vertices in a binary search tree, a linked list, etc.

Note that the skeleton graph representation shown in class was for an adjacency list, which you don't have to use– you could use an adjacency map or an adjacency set instead.

If you choose to use (at least one) linked list, recall from CMSC 132 that there are many variations of lists (singly–linked, doubly–linked, tail reference, circularly–linked, extra dummy head node or sentinel, etc.). You may use any linked list variations that you like, and may use several linked list variations in combination. If you choose to use an adjacency map, recall that CMSC 132 discussed implementing maps using binary search trees, and you should have already written a binary search tree in Java in CMSC 132.

If you have any doubts about whether you are violating these requirements you can discuss your planned design with the TAs during their office hours before starting to code.

There are no requirements for your own tests– we won't even be looking at them– so these restrictions do not apply to your tests, only to your graph implementation.

You will not be graded directly on execution efficiency, but if your graph representation is very inefficient (in particular for large graphs), you might fail some tests and therefore lose credit for them.

4 Compiling using `make`

You must write a makefile, which will be used on the submit server to compile your code for all the public tests, so it must have a rule for each public test present in the project tarfile. Since you won't know how many secret tests there are, or what their dependencies should be, we will use our own makefile to compile them.

Do not wait to write your makefile. Write it first– before you even start writing your code. (That is why this section appears before the required functions are even described.) Writing your makefile first will preclude the need to

type commands for compiling the public tests, so you can save time and avoid making mistakes. Much more importantly, mistakes in writing a makefile can cause your code to not compile correctly– or to compile right on Grace, but not on the submit server. Writing your makefile first and using it all during development, every time you compile your code, will allow you to ensure it works right. So what you should do is to read the rest of this assignment to understand what you have to do, look at the public tests and figure out how they would have to be compiled and what their dependencies are, then write your makefile **first**. Then write your `String_graph` definition in `string-graph-datastructure.h`, and in it write the functions whose prototypes are in `string-graph.h`. The requirements for your makefile are:

- It **must** be in a file named `Makefile` (with a capital first letter). (Your code will not compile on the submit server if your makefile name is wrong.)
- It **must** use the five `gcc` options `-ansi`, `-pedantic-errors`, `-Wall`, `-fstack-protector-all`, and `-Werror`, which were mentioned in the first project, to build object files. (But do **not** use these options to create executable files.) It does not matter if your makefile contains `-g` (to be able to run `gdb` along with these required options.
- Your `Makefile` **must** contain the following targets:

  `all:` This target must appear first and must create executables for all the public tests (and only for the public tests).

  `string-graph.o:` This target must create the object file for `string-graph.c`.

  an object file for each public test source file: Because your makefile is **required** to use separate compilation, it will need to create an object file for each public test source file.

  Note that different tests will have different dependencies, which you can see by looking at what they include; in particular, some object files will depend on `compare-vertex-lists.h` but others will not.

  an executable for each public test: Your makefile will need to have a separate target to build each public test executable. These targets must have names ending in `.x` (`public01.x`, `public02.x`, etc.) and **must** build executables with the same filenames as those targets. (Your code will not work on the submit server if your makefile has incorrect target names or builds executables with the wrong names.) By looking at the source file for each public test you can see what object files have to be linked together to create it; some of them will require `compare-vertex-lists.o` to be linked as part of the executable.

  Because your makefile is **required** to use separate compilation, it must link the appropriate object files necessary to build each executable, **not** directly compile multiple source files together to form executables. So there should never be more than **one** source (`.c`) file in any compilation command in your makefile.

  As above, different tests will have different dependencies and will require different object files to be linked together to create their executable.

  `clean:` This target must remove the object files and executables for all the public tests and `string-graph.o` from the current directory.

- Your makefile may contain additional targets, for example to build your own tests, as long as it has the targets described above. However, **do not add any of your own tests to the all target**, because things may not work on the submit server if you do.
- Your makefile should **not** directly compile header files in compilation commands (e.g., do **not** have compilation commands like `gcc -c string-graph.c string-graph.h`), or you will lose credit.
- Your makefile should have all needed dependencies, otherwise programs may not get built correctly, but it should not have any unnecessary dependencies, because that would lead to unnecessary compilations.
- Do **not** have a target in your makefile to create `compare-vertex-lists.o`, because you are not being given its source file `compare-vertex-lists.c`. Just use `compare-vertex-lists.o` in the linking rules for tests that call the function `compare_vertex_lists()`, without having a rule creating the object file. But your makefile must **recompile anything that uses this object file** if it were to change (for example, if we were to give you a new version of `compare-vertex-lists.o` to fix a bug).

  And if any files include `compare-vertex-lists.h` they must be recompiled if this header file were to be changed as well. (Even though you should not change it, we could need to give you a new version of it.)

- Your `clean` target should remove all object files **other than** `compare-vertex-lists.o`. (If your makefile removed `compare-vertex-lists.o` that would just force you to extract it again from the project tarfile after running `make clean`.) So don't use a wildcard in your `clean` target to remove `*.o` (instead just explicitly list all the object files that should be removed).
- Make has many features that were not explained in class, because they are difficult for a beginner to use correctly. You may **only use the features of make that were covered in class**. If you use features not explained in class your code may not compile on the submit server for reasons not worth explaining, and the TAs can not help fix makefiles using features not covered. **Also, you would lose credit for using any makefile features not covered in class.**

  Almost every past student who tried to use advanced features of make in this course made mistakes. So you should write a simple and straightforward makefile, using just the features of make that were covered.

If there is no makefile with your submission, or if your makefile does not satisfy these requirements, either your program will not compile at all on the submit server, or it might compile for the public tests but not for the secret tests.

Before submitting, you **must** run `make clean`, then run the separate commands `make public01.x`, `make public02.x`, `make public03.x`, etc., to ensure your makefile builds all the public tests correctly. (Use these separate commands, not `make` or `make all`.) Your makefile has to able to build the tests like this on the submit server– meaning when there are no executable or object files in the current directory– so test it in the same situation to ensure it works right then.

To reiterate, your `Makefile` **must** use separate compilation– each source file needed to form an executable must be separately compiled, then the independent object files linked together, to form the executable.

5 Functions to be written

The functions you have to write must be in the file `string-graph.c`. The functions do not produce any output; they modify their `String_graph` parameter or return a value. Some notes:

- **If a pointer passed into a parameter of any function is `NULL`, the function should not change anything**, and (with two exceptions) if it returns an integer value it should return 0. (To avoid repetition this is stated just once here, but don't forget about it when writing the functions.) Note that the last two functions described below should return **different** values if `NULL` is passed into their pointers parameters, as will be pointed out in their descriptions.
- It should be possible for a program that calls your functions to create multiple graphs and they should not conflict. In other words, initializing or modifying one graph should not affect any others in the same program.

Write one function at a time (entirely or even partially), then **test it** thoroughly (as soon as possible) before going on! Write and test helper functions (i.e., utility functions for your data structures) before writing code that uses them, and test them separately first as well. But write your makefile **first**, before even starting to write the graph functions.

5.1 Memory management

For simplicity in this project you may **assume** that **all memory allocations always succeed**.

You don't need to worry about memory leaks in this project. In fact, memory leaks will be unavoidable, because all of the functions you have to write either add components (vertices or edges) to a graph, or report information about a graph, but none of them remove any components of a graph, so code that calls your code has no way to free the memory used by `String_graph` variables once they are no longer needed.

5.2 `void graph_init(String_graph *const graph)`

This function should initialize its graph parameter, causing it to be an empty graph with no vertices or edges. Exactly what it will do depends upon the way you decide to represent and store the components of a graph (meaning what your definition of `String_graph` is, and how it will need to be initialized). Note that the **caller** created the variable that the parameter `graph` points to, and is just passing its address into this function. This function may or may not have to allocate memory for the `String_graph` that its parameter `graph` points to– depending on your data structure– but the variable that `graph` points to **already exists**– it was created in whatever code is **calling** this function– so it should **not**

itself be allocated by this function. The parameter should just be **initialized**, assuming its contents are currently just uninitialized (garbage).

5.2.1 Valid sequences of calls to the functions

Every valid sequence of calls to your functions on a `String_graph` variable must satisfy these properties:

- `graph_init()` must be the first function called on any `String_graph` variable.
- `graph_init()` must not be called again later on any nonempty `String_graph` variable (meaning one that is storing any components).

The effect of any sequence of calls to the functions that does not satisfy these properties is undefined. Ensuring that these properties are not violated is the responsibility of the ***caller*** of your functions (which includes your own tests); your graph functions do **not** have to try to detect violations of these properties, and in fact they have no way to do so.

5.3 `short add_vertex_to_graph(String_graph *const graph, const char new_vertex[])`

This function should add a new vertex with name `new_vertex` to its parameter graph, which as of when it is first created will have no associated edges. If there is already a vertex with name `new_vertex` present in the graph the function should return 0 without modifying the graph, otherwise it should return 1 after adding the vertex to the graph.

The function must copy the vertex name into a **newly-created dynamically–allocated** array of exactly the right size needed to store the name. This ensures that the name is stored without wasting memory space. It also ensures correctness if the user of your functions later modifies or frees the memory of the argument string that is passed in.

To emphasize: the new vertex's name must be **new memory that this function allocates**, **not** just point to the parameter `new_vertex` that was passed in. In other words, this function should **not** just be doing pointer aliasing.

You will **lose credit** if any of your functions cast the return value of the memory allocation functions.

5.4 `short vertex_count(const String_graph *const graph)`

This function should return the number of vertices that have successfully been added to and are currently in its parameter graph, which will always be zero or more.

5.5 `short isa_vertex(const String_graph *const graph, const char name[])`

This function should return 1 if its parameter graph contains a vertex with name exactly equal to `name` (spelled and capitalized identically), and 0 if not.

5.6 `char **get_vertex_list(const String_graph *const graph)`

If its parameter graph itself is `NULL` this function should just return `NULL`. Otherwise it should return a **newly–created dynamically–allocated** array, say `x`, of **pointers to characters**, where the pointers point to **copies** of the names of the vertices in the graph. The names pointed to by the pointers in `x` must be in **increasing lexicographic (dictionary) order** (which is the order produced by the string library comparison functions). Each name must be stored in a **newly–created dynamically–allocated** character array; that is, the pointers in `x` should **not** just be aliases of the names stored somewhere in your graph data structure.

If there are n vertices in the graph, the array returned must have $n+1$ elements, where the last element must be `NULL`. This allows the user to iterate over the elements of the array (the vertex names) and detect when the last element has been reached. (So if a graph has no vertices the function must return a dynamically allocated array of **one element**, which will be `NULL`.)

We are supplying you (in the project tarfile) with a header file `compare-vertex-lists.h` and a corresponding object file `compare-vertex-lists.o` that contains a compiled function `compare_vertex_lists()`, which can be used to compare the results produced by this function against expected results. See the comment in `compare-vertex-lists.h` for more explanation. Some of our tests will use this function when testing `get_vertex_list()`, and your own tests can use it for this as well. Note that Section 4 discussed how your makefile has to handle `compare-vertex-lists.o` and `compare-vertex-lists.h`.

5.7 short create_graph_edge(String_graph *const graph, const char source[], const char dest[], int weight)

This function should add to its parameter graph a new edge with weight `weight` that goes from vertex `source` to vertex `dest`. (Note that because the graph is directed, this edge is independent of an edge from `dest` to `source`, if there is one.) If the edge is added the function should return 1. Some special cases are:

- If `weight` is negative the function should not modify anything in the graph and just return 0.
- If the graph does not already have vertices with names `source` and `dest`, then (assuming the parameter `weight` is nonnegative) the function should first add vertices with whichever of these names don't already exist, then add the edge from `source` to `dest`.
- If there is already an edge going from `source` to `dest` (and `weight` is nonnegative), its weight should be **changed** to the parameter `weight`, and 1 should be returned. (Recall that there can be at most one edge from `source` to `dest` in graphs in this project.)

Note that the integer storing weight does **not** itself need to be dynamically allocated. It just needs to be **inside** (part of) something that is dynamically allocated.

You will **lose credit** if any of your functions cast the return value of the memory allocation functions.

5.8 short get_graph_edge_weight(const String_graph *const graph, const char source[], const char dest[])

This function should return the weight of the edge in its parameter graph that goes from vertex `source` to vertex `dest`.

It should return -1 in any of these cases: (a) if the graph does not have vertices with names `source` and `dest`, (b) if the graph has vertices `source` and `dest` in but there is no edge from `source` to `dest`, or (c) if any of its parameters are `NULL`. (The return value of this function for `NULL` parameters is one of the two exceptions to the first item in Section 5 above.)

5.9 short neighbor_count(const String_graph *const graph, const char vertex[])

This function should return the number of neighbors of vertex `vertex` in its parameter `graph`. A neighbor of the parameter vertex is a vertex, say x, such that there is an edge from the parameter vertex to x, i.e., the number of neighbors of a vertex equals the number of its outgoing edges. Note that a vertex can be its own neighbor. Also, it is possible for a vertex to have no neighbors, if it has no outgoing edges.

The function should return -1 in these cases: (a) if there is no vertex with name `vertex` in the graph; or (b) if either of its parameters is `NULL`. (The return value of this function for `NULL` parameters is the other exception to the first item in Section 5 above.)

A Development procedure review

A.1 Obtaining the project files, compiling, checking your results, and submitting

Log into the Grace machines and use commands similar to those from before:

```
cd ~/216
tar -zxvf ~/216public/project07/project07.tgz
```

This will create a directory `project07` that contains the necessary files for the project, including the header file `string-graph.h` and the public tests. You **must** have your coursework in your special course disk space for this class. After extracting the files from the tarfile, `cd` to the `project07` directory and write your makefile, in a file named `Makefile` (spelled **exactly** that way), to compile your code with all of the public tests. Then, also in the `project07` directory, create a header file named `string-graph-datastructure.h`, containing the type definitions for your graph. And also in the `project07` directory, create a source file named exactly `string-graph.c` that will include `string-graph.h`, and in it write the functions whose prototypes are in `string-graph.h`.

You will not be compiling your code from the command line; instead you will use `make` and your makefile. `diff` can be used as before to check whether your code passes a public test or not, for example:

```
make public01.x
public01.x | diff - public01.output
```

(You can also run `make` from Emacs if desired.)

Running `submit` from the project directory will submit your project, but **before** you submit you **must** make sure you have passed all the public tests, by compiling and running them yourself.

Unless you have versions of all required functions that will at least compile, your program will fail to compile at all on the submit server. (Suggestion– create skeleton versions of all functions when starting to code, that just have an appropriate return statement.)

A.2 Grading criteria

Your grade for this project will be based on:

public tests	40 points
secret tests	45 points
programming style	15 points

B Project–specific requirements, suggestions, and other notes

- **Write helper functions** for manipulating your data structure, that are called from the required functions. If you find you are writing duplicative code in more than one function, stop and turn it into a helper function. If you find that are you are starting to write any function that is longer than 25–30 lines of code stop and carefully consider if you can factor out part of it into a helper function.
- Carefully read the data structure requirements in Section 3.

  Note that in the recent linked list examples, which are in `~/216public`, the list head pointer (and the tail pointer that one example had) were **not** themselves dynamically allocated. But all of the list nodes, the first of which is pointed to by the head pointer, **were** dynamically allocated. Also note that while the example linked lists are storing integers, the integer field of each node was **not** individually dynamically allocated– it is just inside a node that **was** dynamically allocated. These examples illustrate what is meant by a data structure that uses only dynamically–allocated memory– that all of the actual nodes of the linked lists are dynamically allocated in the heap, but not necessarily every field of each node. (Of course fields of some types must be dynamically allocated, as explained in Section 3.)
- Your submission consists of three required user–written files, `Makefile`, `string-graph-datastructure.h`, and `string-graph.c`.
- Be sure to reread the data structure requirements in Section 3.
- Recall that the beginning of Section 5 says what the functions should do if parameters are `NULL`.
- Be careful not to have any pointer aliasing in writing your functions!
- **Do not** write code using loops (or recursion) that has the same effect as any string library functions. If you need to perform an operation on strings and there is a string library function already written that accomplishes that task, you are expected to use it, otherwise you will **lose credit**.
- You will **lose credit** if any of your functions cast the return value of the memory allocation functions. Besides being completely unnecessary, in some cases this can mask certain errors in code.
- You **cannot** modify anything in the header files `string-graph.h` or `compare-vertex-lists.h`, or add anything to them, because your submission will be compiled on the submit server using our versions of these files. You cannot write any new header files of your own other than `string-graph-datastructure.h`.

  You **may not** add any source (.c) files other than `string-graph.c`, so all your source code **must** be in this file.

  Do not write a `main()` function in `string-graph.c` because your code won't compile, since our tests already have `main()` functions. Write any tests in your own separate source files, and compile them together with `string-graph.o` (which will be created from `string-graph.c`).

- You can **only** use C language features that have been covered in class up through Chapter 12 in the Reek text (subject to what the project style guide says is good style).
- If your code compiles on Grace but not on the submit server, you may have changed the provided header files, which you were not supposed to do, or something in your account setup may be wrong. Run `check-account-setup` and come to the TAs' office hours for help if you can't fix any problems that it identifies on your own. (Other causes of this could be: you put your code in a subdirectory of the `project06` directory, you added source or header files, **or** your `Makefile` is wrong, **and** you did not test it as described in Section 4.)
- The only "public" functions in `string-graph.c` are the ones whose prototypes already appear in the file `string-graph.h`. Any helper functions that you write will be "private", so they should be declared `static`, and their prototypes should be placed at the **top of `string-graph.c`**, **not** in `string-graph-datastructure.h`. (And their prototypes should definitely not be put in `string-graph.h`, because if you modify it your code probably won't even compile on the submit server.)
- Make periodic backup copies of your project code! When you are getting close to being finished with the project, copy your entire `project07` directory to one with a different name, so you have a copy of everything in it.
- For this project you will **lose one point** from your final project score for every submission that you make in excess of four submissions, for any reason.
- Recall that all your projects must work on **at least half of the public tests** (by the end of the semester) in order for you to be eligible to pass the course. See the project policies handout for full details.
- If you have a problem with your code and have to come to the TAs' office hours for debugging help you **must** come with tests you have written yourself that illustrate the problem (**not** the public tests), what cases it occurs in, and what cases it doesn't occur in. In particular you will need to show the **smallest** test you were able to write that illustrates the problem, so whatever the cause is can be narrowed down as much as possible before the TAs even start helping you. You **must** also have used the `gdb` debugger, explained in discussion section, and be prepared to show the TAs how you attempted to debug your program using it and what results you got.

  To emphasize: **the TAs will not look at your program's results on any of the public tests in office hours. You must have written your own tests to receive any help with your program code.**

C Academic integrity

Please **carefully read** the academic honesty section of the syllabus. **Any evidence** of impermissible cooperation on projects, use of disallowed materials or resources, publicly providing others access to your project code online, or unauthorized use of computer accounts, **will be submitted** to the Office of Student Conduct, which could result in an XF for the course, or suspension or expulsion from the University. Be sure you understand what you are and what you are not permitted to do in regards to academic integrity when it comes to projects. These policies apply to all students, and the Student Honor Council does not consider lack of knowledge of the policies to be a defense for violating them. More information is in the course syllabus – please review it now.

The academic integrity requirements also apply to any test data for projects, which must be **your own original work**. Exchanging test data or working together to write test cases is also prohibited.