Simple GIS

Arrays, C `struct` types, Searching, Sorting

A geographic information system (GIS) is a software system that provides storage, retrieval, search, and update facilities for data representing spatial relationships such as a simple map. There are many ways to organize such data, and indeed many different varieties of data that might qualify. For this project, we will restrict our attention to very simple map data representing adjacency relationships among cities.

The map we need to model may be viewed as a simple mathematical graph, consisting of nodes representing cities and edges representing possible transportation routes between these cities. In general, the nodes and edges that make up a graph may both have attributes that must be represented. For now, we view each city as having a small set of attributes: name, state, and a list of other cities that may be reached directly from this city. For now, we view transportation routes as having no interesting attributes (aside from existence).

The GIS will store a list of cities, including all the attributes identified above. The GIS will provide its client with the ability to add new cities, to add new transportation routes between cities, to look up and report the attributes of a particular city, to delete transportation routes, and to delete cities.

In this case, the client will simply be another component of the overall system; the client will receive commands via a script file, and which will then interact with the GIS to carry out those commands and to report the results.

Note well: this project is the first in a sequence of assignments, each of which will extend and modify earlier assignments. Giving careful thought to design is essential. Aside from the fact that we will be evaluating your design, poor decisions now may cost you extra effort and frustration on later assignments.

Input Data

On startup, the client will be provided with three input files. Each is guaranteed to conform to the descriptions that follow.

The first input file will contain data about the cities that are included in the map that is being modeled. Aside from a header section, each line of the city data file will be of the form:

```
<city name><tab><state abbreviation><newline>
```

The file will begin with a three-line header, which is of no consequence to the client.

The second input file will contain data about transportation routes that are included in the map that is being modeled. Aside from another three-line header, each line of the route data file will be of the form:

```
<city name><tab><state abbreviation><tab><city name><tab>
     <state abbreviation><tab><distance><newline>
```

(Note: the line break shown above is only due to the width of this page. All of the data fields are on a single line of the file.)

The third input file will contain commands to be carried out, and comments. Any line beginning with a semicolon character (`';'`) is considered a comment, and should be ignored by the client. The commands, and their effects, are described in a later section.

System Initialization

On startup, the client will read the city data file, and cause the city data to be added to the GIS. In this project, the GIS is limited to storing records for no more than 500 cities; if the city data file contains more than that, the extra ones are not added to the GIS. Note that the system does not allow two records to store the same city name and state; if a duplicate city entry is found, nothing is added to the GIS and an error message should be logged (see the log file description later).

If there are cities in the city data file that are not added to the GIS, an error message should be logged to indicate that happened. It is NOT necessary to log every extra city, just that there were extras.

Next, the client will read the route data file, and cause that data to be added to the GIS. Each route specifies two cities; both must already be in the GIS in order for the route to be added. Note that routes are inherently bi-directional. If a route specifies an unknown city, the route is not added to the GIS. In this project, each city is limited to having no more than ten immediate neighbors. In some cases, the route data file may include more than ten routes for a particular city; in that case, the extra ones are not added to the GIS. The GIS also only allows a single direct route between the same two cities. If a duplicate route is given, nothing is added to the GIS, and an error message should be logged.

If a transportation route specifies an unknown city, or cannot be added because one of the cities already has ten direct neighbors, an error message should be logged to indicate what happened.

Once both these files have been processed, the system should proceed to the script file and process the commands found there. Each command must be echoed to the log, and the output generated for each command must be clearly delimited from the output from other commands. See the section on the log for formatting details.

Script File

The script file may contain any of the following commands. Each command is terminated by a single newline character. Key words and phrases that will be used verbatim are shown in **`bold`**.

`find city``<tab><city name><tab><state abbreviation>`

Search the GIS for records matching the given city name and state. If a record is found, log the index at which the record was found, the city name and state, and a list of the indices at which all the neighbors of that city occur. If no record is found, log an error message.

`find``<tab><city name>`

This is similar to the `find city` command, except that multiple matches are possible. If matching records are found, log the same information for each as for the `find city` command. If no matches are found, log an error message.

```
add route<tab><city name><tab><state abbrev><tab><city name><tab>
           <state abbrev><tab><distance>
```

If the specified route is not found in the GIS, add it, subject to the same rules as given for the route data file. If there is a problem, log an error message. Note this never results in the addition of a new city to the GIS.

`add city``<tab><city name><tab><state abbrev>`

If the specified city is not found in the GIS, add it subject to the same rules as given for the city data file. If there is a problem, log an error message.

`remove route``<tab><city name><tab><state abbrev><tab><city name><tab><state abbrev>`

If the specified route is found in the GIS, remove it. If not, log an error message. Note this never results in the deletion of a city from the GIS.

`remove city``<tab><city name><tab><state abbrev>`

If the specified city is found in the GIS, remove it and all of the routes that are incident to it. If not, log an error message.

dump`<tab>`

Log all of the city entries in the GIS, including the same information as specified in the `find city` command. Note this command is included only for your testing purposes, and it will not be used in any of the scripts when your implementation is evaluated.

exit`<tab>`

Immediately stop processing the script file and shut down. No output is produced.

Each command given in the script file will be syntactically correct, according to the specifications above. Some will raise logical issues, described in the previous section. The system should stop processing the script gracefully if it runs out of commands without finding an explicit `exit` command.

Legend:

`<city name>`	string, not containing a tab
`<state abbreviation>`	string, not containing a tab
`<distance>`	non-negative integer value

There is no guaranteed constraint on the length of city names, so your implementation should handle storing city names of arbitrary length. However, for formatting purposes, you may assume that no city name will be longer than 25 characters.

Log

During execution the system will produce a log, written to file. The log file must conform precisely to the following specifications.

The first line of the log file identifies the programmer; the exact formatting and contents are up to you. The second line of the log file must be blank. The remainder of the log file consists of a sequence of logical records, one for each command that was processed. The records must be separated by non-blank lines containing a sequence of delimiting characters; the sample log files will use a line of hyphens, but you are free to make another choice.

The first line of each command record simply echoes the command that is being processed. This may be a verbatim copy of the command as read from the file, or reformatted according to your taste.

Note: none of the lines described so far will be assigned any points when your output is evaluated. That is why you are given so much flexibility in the details; however, if you omit any of the specified lines you will surely lose points. (So, read the section in the *Curator Student Guide* that describes how scoring is done.)

For each command, except the `exit` command, the system must make a log entry reporting the result of processing that command. These lines will be assigned point values, and you must follow the specifications for them precisely. The logical content of each of the command reports has been described in the previous section. What has not been specified, yet, are the formatting constraints, and the precise contents of error messages. Rather than give a tedious list here, these will simply be illustrated in the posted log files that will appear shortly on the course website.

Implementation Constraints

This specification imposes a number of limitations on how you are allowed to implement your solution. Some of these constraints may appear to be unjustified. They are not. Regardless, failure to conform to them will certainly result in a grade penalty on the project.

- You are forbidden to use anything that is not ISO Standard C++. The course notes present only Standard C++. If you are not sure about something, ask. In particular, you may not use any of the non-Standard abominations commonly presented in the AP Computer Science course. These include, but are not necessarily limited to, `apstring` and `apvector`.
- You are forbidden to explicitly use any dynamic allocation in this assignment (e.g., `new`, `delete`, `malloc`). Dynamic allocation is covered later in the course, and one of the later assignments will require you to modify this implementation to incorporate it.

- You are forbidden to use any STL containers, except `string`, on this assignment. In view of the restrictions given above, you are thus limited to using statically-allocated arrays for storing lists. This will impose some functional limitations on your solution; that's OK.
- You are forbidden to write classes. We will cover the C++ class mechanism in some detail, and you will convert this implementation to take advantage of it for a later assignment. But for now, you are limited to the Standard C++ classes which are not otherwise banned (e.g., I/O streams, string objects).

We will be covering separate compilation fairly early in the term, but it will not be supported in the evaluation of this project. If you already understand separate compilation, you are encouraged to develop your implementation as a collection of separate source files. However, when you submit your solution to the Curator System, you must submit a single source file containing the entire implementation, and that is what the Curator will attempt to compile.

There are also some constraints on the logical functionality of your solution, imposed in order to guarantee that there is only one correct log for each set of input data.

When adding a new city the GIS will append it to the end of the list; this means that cities are not stored in a optimal order as far as searching is concerned. We will deal with that issue in the next project.

When adding a new route to the GIS, you must add new neighbor entries to two city records. Neighbor entries are to be stored in the order they are added, so new entries always go at the end of the city record's neighbor list.

When removing a city, the GIS will not physically remove the entry from its list. This is for simplicity; physically removing a city record would require shifting the contents of the list, and that would require updating many of the neighbor entries to account for the shifting. While that is possible, it is excessively expensive. So, removing a city will involve marking the record in some special way to indicate that it is unused. That could result in significant wasted space over time. We may also deal with that issue in a later project.

Removing a route does not raise the same issues, so there is nothing special there; you will simply delete neighbor entries from the relevant neighbor lists.

Evaluation

Do not waste submissions to the Curator to test your program! There is no point in submitting your program until you have verified that it produces correct results on the sample data files that will be provided. If you waste all of your submissions because you have not tested your program adequately then you will receive a low score on this assignment. You will not be given extra submissions.

Your submitted program will be assigned a score, out of 200, based upon the runtime testing performed by the Curator System. The TAs will also be evaluating your submission of this program for documentation quality, and a few good coding practices. This will result in a deduction (ideally zero) that will be applied to your score from the Curator to yield your final score for this project.

Read the *Programming Standards* page on the CS 1704 website for general guidelines. You should comment your code in some detail. In particular:

- You should have a header comment identifying yourself, and describing what the program does.
- Every constant and variable you declare should have a comment explaining its logical significance in the program.
- Every major block of code should have a comment describing its purpose.
- Every function implementation must have a header comment. The format of the header comment is described in the course notes, as well as on the *Programming Standards* page on the course website.
- Adopt a consistent indentation style and stick to it.

Understand that the list of requirements given here is not a complete repetition of the *Programming Standards* page on the course website. It is possible that requirements listed there will be applied, even if they are not listed here.

As stated in the course policies, the submission that receives the highest score will be evaluated. If two or more of your submissions are tied for the highest score, then the earliest of those submissions WILL be evaluated. Therefore, DO NOT make undocumented, incomplete submissions to the Curator and then complain that you didn't want those to be evaluated. I will grant NO exceptions.

File Names

We will use fixed file names for this assignment. The three input files are named `CityData.txt`, `RouteData.txt`, and `GISScript.txt`, respectively. The log file is named `GISLog.txt`.

If you use the wrong names for these files, you will get poor results when you submit to the Curator.

Submitting Your Program

You will submit this assignment to the Curator System (read the *Student Guide*), and it will be graded automatically. Instructions for submitting, and a description of how the grading is done, are contained in the *Student Guide*.

You will be allowed up to five submissions for this assignment. Use them wisely. Test your program thoroughly before submitting it. Make sure that your program produces correct results for every sample input file posted on the course website. If you do not get a perfect score, analyze the problem carefully and test your fix with the input file returned as part of the Curator e-mail message, before submitting again. The highest score you achieve will be counted.

The *Student Guide* and submission link can be found at: `http://www.cs.vt.edu/curator/`

Pledge

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the pledge statement (provided on the course website) in the header comment for your program.

Failure to include the pledge statement may result in a substantial grade penalty.