CS3114/5040 (Fall 2023)
PROGRAMMING ASSIGNMENT #4

Due Wednesday, December 6 @ 11:00 PM for 150 points
Due Wednesday, December 6 @ 11:00 AM for 5 point bonus
Due Tuesday, December 5 @ 11:00 PM for 10 point bonus
Due Tuesday, December 5 @ 11:00 AM for 15 point bonus
Note: This project also has three intermediate milestones. See the Piazza forum for details.
Updated: 11/12/2023 @ 4:15pm

Assignment:

For this project, you will be doing some simple analysis on data from a subset of the Million Song database. The records that you will store for this project are artist names and track names. The two main data structures used in this project are a hash table and a graph.

You will implement a variation on the **closed hash table** that you wrote for Project 1. There will be two instances of the hash table, one for accessing artist names and the other for accessing song titles. For information on hash tables, see Chapter 10 on Hashing in the OpenDSA textbook. You will use the `sfold` string hash function described in OpenDSA (this method is given in the project starter kit) and you will use simple quadratic probing for your collision resolution method (the i'th probe step will be i^2 slots from the home slot). The key difference from what the book describes is that your hash tables must be **extensible**, exactly as in Project 1. That is, you will start with a hash table of a certain size (defined when the program starts). If the hash table would exceed 50% full due to an insert, then you will first replace the array with another that is twice the size, and rehash all of the records from the old array.

For this project you will also maintain a graph data structure. The graph will be implemented using the **adjacency list** data structure (see Chapter 14 in OpenDSA). The graph will have a node for each song and each artist, and will link a song node with an artist node if there is a recording of that song by that artist. Specifically, when you process an insert command, do the following:

1. If this is a new artist name or new song name, add new nodes to the graph as necessary.
2. Add a link in the recording graph between the nodes for the artist and the song.

You will need to come up with a reasonable way for expanding the number of nodes in the graph as necessary. Keep in mind that nodes can be removed from the graph as well.

As usual, the hash table needs to associate keys with values. In this project, the key is a string (either an artist name or a song title, depending on the hash table). The data to be associated with the one of these strings is the corresponding node in the graph. In other words, the hash table gives access to graph nodes.

Program Invocation:

The program will be invoked from the command line as:

```
java GraphProject {initHashSize} {commandFile}
```

The name of the program is `GraphProject`. Parameter `{initHashSize}` is the initial size for each of the two hash tables (in terms of slots). Your program will read from text file `{commandFile}` a series of commands, with one command per line. The program should terminate after reading the end of the file.

Input and Output:

You will process a series of commands to build a graph relating artist and song names. The formats for the commands are as follows. The commands are well formatted, meaning there will be no unnecessary spaces or other white space, blank lines, etc. All commands should generate a suitable output message (some have specific requirements defined below, or in the sample I/O files). All output should be written to standard output. Note that OpenDSA Module 2.7 uses an input command format very close to the one used in this project for the second example.

insert `{artist-name}<SEP>{song-name}`

Note that the characters `<SEP>` are literally a part of the string, and are used to separate the artist name from the song name. Check if `{artist-name}` appears in the artist hash table, and if it does not, add a node for that artist name to the graph, and store the string and node in the appropriate slot of the artist hash table. (A good approach is to create a small class to store the name and node together in a slot of the hash table.) Likewise, check if `{song-name}` appears in the song hash table, and if it does not, add a node for that song to the graph, and store the song name and graph node in the appropriate slot of the song hash table. You should print a message if the insert causes a hash table to expand in size.

remove `{artist|song} {name}`

Remove the specified artist or song name from the appropriate hash table, and remove that node from the graph. Report the outcome (whether the name appears, and whether it was successfully removed).

print `{artist|song|graph}`

Depending on the parameter value, you will print out either a complete listing of the artists contained in the database, or the songs, or else the graph. For artists or songs, print the graph much like you did in Project 1: simply move sequentially through the associated hash table, retrieving the strings and printing them in the order encountered (along with the slot number where it appears in the hash table). Then print the total number of artists or total number of songs.

When the `graph` option is given you will do the following:

1. Compute connected components on the graph. (You will use the Union/FIND algorithm for this purpose, see OpenDSA Module 13.2.) Print out the number of connected components, and the number of nodes in the largest connected component.

2. Compute (and print) the diameter for the largest connected component using Floyd's algorithm for computing all-pairs shortest paths (OpenDSA Module 14.8). If there are multiple largest connected components, print the largest diameter of those largest components. When implementing Floyd's algorithm, all edges will have a weight of 1.

Programming Standards:

You must conform to good programming/documentation standards. Web-CAT will provide feedback on its evaluation of your coding style, and be used for style grading. Beyond meeting Web-CAT's checkstyle requirements, here are some additional requirements regarding programming standards.

- You should include a comment explaining the purpose of every variable or named constant you use in your program.
- You should use meaningful identifier names that suggest the meaning or purpose of the constant, variable, function, etc. Use a consistent convention for how identifier names appear, such as "camel casing".
- Always use named constants or enumerated types instead of literal constants in the code.
- Source files should be under 600 lines.
- There should be a single class in each source file. You can make an exception for small inner classes (less than 100 lines including comments) if the total file length is less than 600 lines.

We can't help you with your code unless we can understand it. Therefore, you should not bring your code to the GTAs or the instructors for debugging help unless it is properly documented and exhibits good programming style. Be sure to begin your internal documentation right from the start.

You may only use code you have written, either specifically for this project or for earlier programs, or code provided by the instructor. Note that the OpenDSA code is not designed for the specific purpose of this assignment, and is therefore likely to require modification. It may, however, provide a useful starting point.

Java Data Structures Classes:

You are not permitted to use Java classes that implement complex data structures. This includes `ArrayList`, `HashMap`, `Vector`, or any other classes that implement lists, hash tables, or extensible arrays. (You may of course use the standard array operators.) You may use typical classes for string processing, byte array manipulation, parsing, etc.

If in doubt about which classes are permitted and which are not, you should ask. There will be penalties for using classes that are considered off limits.

Deliverables:

You will implement your project using Eclipse, and you will submit your project using the Eclipse plugin to Web-CAT. Links to the Web-CAT client are posted at the class website. If you make multiple submissions, only your last submission will be evaluated unless you arrange otherwise with the GTA. There is no limit to the number of submissions that you may make.

You are required to submit your own test cases with your program, and part of your grade will be determined by how well your test cases test your program, as defined by Web-CAT's evaluation of code coverage. Of course, your program must pass your own test cases. Part of your grade will also be determined by test cases that are provided by the graders. Web-CAT will report to you which test files have passed correctly, and which have not. Note that you will **not** be given a copy of these test files, only a brief description of what each accomplished in order to guide your own testing process in case you did not pass one of our tests.

When structuring the source files of your project, use a flat directory structure; that is, your source files will all be contained in the project "src" directory. Any subdirectories in the project will be ignored.

You are permitted to work with a partner on this project. While the partner need not be the same as who you worked with on any other projects this semester, you may only work with a single partner during the course of one project unless you get special permission from the course instructor. When you work with a partner, then **only one member of the pair** will make a submission. Be

sure both names are included in the documentation. Whatever is the final submission from either of the pair members is what we will grade unless you arrange otherwise with the GTA.

Pledge:

Your project submission must include a statement, pledging your conformance to the Honor Code requirements for this course. Specifically, you must include the following pledge statement near the beginning of the file containing the function main() in your program. The text of the pledge will also be posted online.

```
// On my honor:
//
// - I have not used source code obtained from another current or
//   former student, or any other unauthorized source, either
//   modified or unmodified.
//
// - All source code and documentation used in my program is
//   either my original work, or was derived by me from the
//   source code published in the textbook for this course.
//
// - I have not discussed coding details about this project with
//   anyone other than my partner (in the case of a joint
//   submission), instructor, ACM/UPE tutors or the TAs assigned
//   to this course. I understand that I may discuss the concepts
//   of this program with other students, and that another student
//   may help me debug my program so long as neither of us writes
//   anything during the discussion or modifies any computer file
//   during the discussion. I have violated neither the spirit nor
//   letter of this restriction.
```

Programs that do not contain this pledge will not be graded.