CS3114 (Fall 2019)
PROGRAMMING ASSIGNMENT #2

Due Tuesday, Oct 22nd @ 11:00 PM for 100 points
Early bonus date: Sunday, Oct 20th @ 11:00 PM for a 10 point bonus

In this project, we continue our challenge of a course manager for CS3114, but in this project, we will have multiple data structures rather than a single one for student data records. We will have a separate student manager and a course manager. We will also include file processing functions to speed up data input and output procedure. Student data records will be separated from the index data structures. When the course manager quits, the student data records will be stored in a binary file and should also be loaded into the system later.

Student record:

In project 2 we will define student data record to have at least the following members:

Personal identification number (pid #): It is a nine-digit integer that will be loaded from a csv file at the beginning. You have the freedom to implement it in whatever data type (string, int or long all can handle that). When we enroll a student into the course manager, the pid # will be checked. Note that the pid # is unique. There won't be duplication in the provided csv file. This ***pid #*** will be used to determine the identity of a student, not the ***name*** as in project 1.

Name: This object includes a first name, a possible middle name, and a last name. Students with the same name will be allowed to be assigned to the same section from now on, as students' identity will be uniquely determined by their corresponding pid numbers (***pid #***).

Score: This will be the same as in project 1: Students' scores will be integers between 0 and 100. When first loaded, the default value for student scores can be either 0 or -1.

Grade: This will be the same as in project 1: Just follow the grade table on syllabus.

Student records will be managed by both a student manager and a course manager. The student manager manages the identity of a student (along with other personal information that we will not handle in our projects) and the course manager manages enrollment info, scores and grades.

Please note that our projects focus on managing the course. The student manager serves mainly as an identity check tool. We do not modify the identity info for a student in this projects at all. All student info will be loaded directly from an external text file (will be provided by our TAs) at the beginning. Later we don't remove or insert students to that database. Thus it will be fine to use a ***sorted array*** to store student info in the student manager, but ***you may also choose to use your well tested BST in project 1 for that purpose***. Student enrollment to the course and corresponding score records can also be loaded from an external text file. After the loading and command line operations, they will be fully managed by the two managers and be saved and loaded through binary files. Because of frequent insert and remove operations, course manager will be managed by BSTs.

Input and Output:
The program will be invoked from the command-line as:

java Coursemanager2 <command-file>

The name of the program is Coursemanager2. Parameter *command-file* is the name of the input file that holds the commands to be processed by the program.

The input for this project will consist of a series of commands (some with associated parameters, separated by spaces), one command for each line. A blank line may appear anywhere in the command file, and any number of spaces may separate parameters. You need not worry about checking for syntactic errors. That is, only the specified commands will appear in the file, and the specified parameters will always appear. However, you must check for logical errors. The commands will be read from the command file, and the output from processing the commands will be written to standard output. The program should terminate after reading the EOF mark. The commands are as follows. Detailed output message formats will be specified in a separate project format file (to be released by the TAs).

loadstudentdata <textfile name>

Here ***<textfile name>*** should carry an extension ".csv". This command will read this file line by line. This textfile will contain all necessary student info. Note that this file may contain students that are not in any sections of the course. The course enrollment and score information will be loaded separately. The text file will have syntax-error-free texts separated by commas. The content will have a format like the following:

Id # (exactly 9 digits), first name, middle name, last name

Here each line represents a student record. The middle name field for a line might be empty, but will still be separated by commas before and after it. Any number of spaces may appear between fields.

In our projects, you may assume that student information will not have conflict with each other. The pid # will be unique.

loadstudentdata <binary file name>

Here ***<binary file name>*** should have an extension ".data". This command will read the binary file based on the file structure. It should first decide how many records are stored and then load all of them into main memory. An index file should be built in the loading process. The binary file format will be given in the file system specification part of this document.

loadcoursedata <textfile name>

Here <textfile name> should carry an extension ".csv". This command will read this file that contains the enrollment information and maybe current scores for all sections in the course. We could have up to 20 sections in project 2, and there might be a temporary section used to merge all sections. We may load multiple files that contain information for one or multiple sections. If a student record gets loaded twice, a later loading will overwrite his score and grade previously loaded. But if an already enrolled student (identified by pid #) is loaded into a different section, that line should be rejected and a warning message should be printed.

The format of the textfile will be the following.

Section id, student Id #, first name, last name, score, grade

Again, each line represents one student record. The score and grade might be empty, but the first four fields will have full content. Any number of spaces may appear between fields. When loading the ***coursedata***, your system should have a ***studentdata*** file loaded already. Otherwise, you should print out an error message and reject the loading operation. Then when you load each line of the ***coursedata***, the student information should be verified through the already loaded studentinfo database. The id # should exist and the full name (both first and last names) should match with the ***studentdata*** record. You should also check if that student is already registered into a different section. If any of them don't match, you should print out a warning message, ignore that line and continue loading other lines. You have the freedom to design your data structure so that you can conveniently check a student's enrollment information.

loadcoursedata <binary file name>

Here ***<binary file name>*** should have an extension ".data". This command will read the binary file based on the binary file structure for the coursedata. When all student data are loaded into main memory, corresponding index data structures, such as three BSTs in project 2, will be built for each section based on the data. You may also want to build other index data structure to efficiently implement all functions listed here. The binary file format will be given in the file system specification part of this document.

section <number>

This command follows the same idea in our project 1. Note that we could have up to 20 sections in project 2. The section ids will be continuous from 1 to 20. It indicates that the following commands are all corresponding to this particular section unless a new "section" command overwrites this one. This command will affect the results of following commands. If no section command is given before, the default value is section 1.

insert <pid #> <first name> <last name>

Enroll a student with ***<pid #>*** and a name given by that ***<first name>*** and ***<last name>*** to the current section. The student info should be verified though your student manager, if the pid # does not exist, or if the names associated with that pid # do not match. This command should be rejected with proper error message printed. If the student info is verified, you need to also check if this student has already enrolled in a different section, if so, reject the insertion and print an error message. For that purpose, you may want to maintain the student enrollment info for this course in either the student manager or the course manager. After all above verification steps, this student can enroll in this section with a default score 0, and a default grade F.

Note that in project 2, same names may appear in the same section as long as their pid #s are different. In this command, you don't have to worry about invalid names. All names provided in our test cases will be valid (two strings separated by a space), but will have any combination of small and capital cases. If the insert command is rejected for whatever reason, the ***score*** command following it will not be valid.

searchid <pid #>

Search for the specific student record in the current section with the pid # given. If that pid # is not found in the current section, print out an error message. Otherwise, print out the corresponding record. This command is still a search. The name is changed to ***searchid*** so that it will not be confused with the ***search <name>*** command.

search <first name> <last name>

Search for all student records in the current section with the name given by ***<first name>*** and ***<last name>.*** Print out all the records associate with that name. Then print out the total number of records found.

search <name>

Report all students in the current section that have the specific ***<name>.*** Note that this name may appear in either first name or last name. All of them should be reported. Then a number should be reported about how many records have been found with that name. If only one record is found in this command, then a ***score*** command may follow. Otherwise, a ***score*** command will be considered as invalid.

You may decide to build a separate data structure to help search for <name>. Or you can simply use BST traversal for that purpose. It will be your own choice and we don't check what index structure you use to perform this function.

score <number>

This command is only valid when it follows a valid insert or search command, which means that we have a clear student record before the score command. Otherwise, print an error message. If this command is valid and the ***<number>*** is between 0 and 100, assign a score given by ***<number>*** for the specific student. Note that you will also update the corresponding BST that uses score as its key value. Updating a BST usually means two steps: You need to remove the previous record and then insert a new one in. Please carefully design this part.

In this project our test cases will only use integers for the score number. Particularly note that logically we only assign the score when there is an active student record. If a ***search*** command returns multiple student records, the ***score*** command is considered invalid. In a case that a ***score*** command follows any command that is not ***insert*** or ***search***, we will also consider it invalid, to avoid confusion.

remove <pid #>

Remove the record in the current section with the given ***pid #***. If there is no such a student, or this student is not enrolled in this section, you should print out an error message and reject the remove command. Otherwise, the record associated with that name will be removed from the current section. When you remove a student, all index data structures for this section should be updated, while the array that stores full student enrollment info will be labeled as removed (tombstone technique, as described in later description). You may use different strategy to handle this situation, but a removed student should not appear in any of your search results or dump reports.

remove <first name> <last name>

Remove the record in the current section with the name given by ***<first name> <last name>.*** If there are multiple students with this full name or there is no such a student in the current section, reject this **remove** command and print out corresponding error messages. Otherwise, the record associated with that name will be removed from the current section. Again, when you remove a student, all index data structure for this section should be updated, while the array that stores full student enrollment info will be labeled as removed.

clearsection

Remove all records associated with the current section. Note that this command only affects records in this section. This section still exists but all records associated with it are removed.

dumpsection

Return a “dump” of the three BSTs associated with the current section. The three BSTs are based on different key values: **pid #, name, or score**. Please dump them in that order. The BST dump should print out each BST node using the ***in-order traversal***. For each BST node, print that node's brief student record (pid #, name, and score). After all three BSTs get printed, please print out the size of the data. More details about the dump command will be posted by our TAs later.

grade

Go through the current section and check the score of each student, assign corresponding grades based on our grade table (check our syllabus for it).

stat

Report how many students are in each grade level, from A to F.

list <grade>

List all students that are in particular grade level(s) given by <grade>. Then print out the total number of students listed. If a star is associated with a character, such as "B*", please report all students with "B+", "B", and "B-". Or <grade> maybe is a specific "b" or "B-", you should count all students only with that grade then. Our command may provide a lower case character, so you should process it first.

findpair <score>

Report all student pairs whose scores are within a difference given by (less than or equal to) <score> in the current section. If no <score> is given, then the default value is 0, which means that the pair should have the same score. At the end of the report, a number should be reported about how many pairs were found in this command. Note that you can only report once for each pair. For example, if A, B and C have the same score, then there should be three pairs: (A, B), (A, C), and (B, C). You should not report (B, A) if (A, B) is reported earlier. The order of the pair does not matter.

merge

Merge all course sections into one separate section. This command can only be used for a section that does not contain any student records. For example, if there are totally 4 sections, then ***merge*** command cannot be called within section 1-4, but can be from section 5-21. Commands for this new section are only for grading and stat purpose. Thus only ***clearsection, dumpsection, stat, list, and findpair*** commands will be valid for this section. Student records, enrollment info should not be affected from this section. **After a merge command is called, if the user switch to an individual section and modify some student data (such as insert a new student or change a student score), it does not change the merged section. A new merge command must be called to reflect that change.**

When the student data get saved, this section will not be saved either.

savestudentdata <filename>

Save all student data into a binary file, <filename> should carry an extension ".data". The file should start with a string (binary format) "VTSTUDENTS". Then next four bytes is an integer (type int) that indicates how many student record data are saved in this file. After that, student data get saved in sequence as the way they were loaded from the ***loadstudentdata*** command.

NOTE: In our project we will always use UTF-8 for encoding of strings. Thus I will not specify how many bytes are used to represent strings, but I will specify how many characters are there. Please note the difference in this revision to the initial version.

The format of this binary file should be:
A string: "VTSTUDENTS" (10 characters in UTF-8 format),
An int: record # n (4 bytes),
Then there will be n records in the following format:
pid # (long, 8 bytes),
first name (a string end with '$', varying length in UTF-8 format, the rest are the same),
middle name (a string end with '$') and
last name (a string end with '$'),
a delimiter string: "GOHOKIES".
When we store strings for names, each string will end with a special character "$". In our test cases, we will make sure that each string in the name will not exceed 16 characters, including the

'$'.

Note that in a binary file we don't really need a delimiter. But to help people debug, I think a delimiter might help. In some design, your student manager might want to know student enrollment information for the course. That should be updated when you load course data. The file for student data does not contain that info.

savecoursedata <filename>

Save all course data into a binary file, <filename> should carry an extension ".data". The file should start with a string (binary format) "CS3114atVT". Then next four bytes is an integer (type int) that indicates how many sections are included in this file. After that, student data get saved in sequence.

The format of this binary file should be:
A string: "CS3114atVT"(10 characters),
An int: section # m (4 bytes)
Then after it there should be information for all sections (in sequence, from section 1 to m). Each section will follow the following format:
An int: record # n (4 bytes), indicates how many students enrolled in this section.
Then there will be n data in the following format (no delimiter between student records)
    a pid # (long, 8 bytes),
    first name (a string end with '$'),
    last name (a string end with '$'. Here in coursedata we don't store middle name.)
    score (int, 4 bytes),
    grade (a string of exactly two characters, such as "B-" or "B ").
When all students info are done for a section, put
    a section delimiter string: "GOHOKIES" (8 characters),
and then start another section.

When loading the binary file, student and course information should be retrieved in the same format. In your code, you can use the delimiter string to check if you are off. The studentdata and coursedata don't have index info. You will need to build them when you load the data from binary files.

clearcoursedata

Save the coursedata info (we don't check that) and clear all BST and array space associated with the course. This command is designed so that we can test your save and load commands. You don't need to overwrite those old BSTs or arrays. But when you later load your data from a binary file, your BSTs and arrays should be reconstructed from the loading.

Implementation:

In your design, there should be two manager classes. A student manager manages the student identity info, and a course manager manages the enrollment and gradebook info for all sections. For the student manager, a simple BST or a sorted array with key value in pid # will be enough for the management purpose (we don't check this data structure). For the course manager, you should have an array of sections. Then under each section, there will be three BSTs and an array that stores the actual data. The BST will store a key value (pid #, name, or score) and an index for the actual record. With that index, you can visit the array for the actual data. Note that here BST functions for course manager can also be achieved by sorted arrays. So if you failed your project 1 and don't want to reuse your project 1 BST, it will be fine to use sorted array for project 2. A 10-point penalty will be applied though.

Note that ordinary remove operation in an array takes O(n) complexity and the movement will affect the index for other records. Here we will simplify the remove step. When we remove a record, we simply mark a flag for that record from 1 to 0 or from True to False (This flag is called a tombstone). Then later when we insert a new record, we append the record to the end of the array. We will not worry about the wasted space here in project 2. Later in future projects we will implement a memory manager to handle that.

Later when we save the working data to the binary file, we will ignore those student data marked with a tombstone. Student data get saved in a sequential way as they are in the array. When the binary data get loaded, note that those BSTs will have to be reconstructed. So the BSTs will get changed when you save the data, clear the manager and then load the data back.

Programming Standards:

You must conform to good programming/documentation standards. Some specifics:

- You must include a header comment, preceding main(), specifying the compiler and operating system used and the date completed.
- Your header comment must describe what your program does; don't just plagiarize language from this spec.
- You must include a comment explaining the purpose of every variable or named constant you use in your program.
- You must use meaningful identifier names that suggest the meaning or purpose of the constant, variable, function, etc.
- Always use named constants or enumerated types instead of literal constants in the code.
- Precede every major block of your code with a comment explaining its purpose. You don't have to describe how it works unless you do something so sneaky it deserves special recognition.
- You must use indentation and blank lines to make control structures more readable.
- Precede each function and/or class method with a header comment describing what the function does, the logical significance of each parameter (if any), and pre- and post-conditions.
- Decompose your design logically, identifying which components should be objects and what operations should be encapsulated for each.

Neither the GTAs nor the instructors will help any student debug an implementation 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 taken from the textbook. Note that the textbook 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.

Testing:

Sample data files will be posted to help you test your program. This is not the data file that will be used in grading your program. The test data provided to you will attempt to exercise the various syntactic elements of the command specifications. It makes no effort to be comprehensive in terms of testing the data structures required by the program. Thus, while the test data provided should be useful, you should also do testing on your own test data to ensure that your program works correctly.

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. There is no limit to the number of submissions that you may make.

You are required to submit your own test cases (should cover at least 70% of your code) with your program. Of course, your program must pass your own test cases. Your grade will be fully determined by test cases that are provided by the graders (TAs). 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 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 (and encouraged) to work with a partner on this project. When you work with a partner, then only one member of the pair will make a submission. Both names and emails **must** be included in the documentation and selected on any Web-CAT submission. The last submission from either of the pair members is will be graded.

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 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.