CS3114 (Fall 2019)
PROGRAMMING ASSIGNMENT #1

Due Thursday, Sep 26th @ 11:00 PM for 100 points

Due Tuesday, Sep 24th @ 11:00 PM for 10 point bonus

Last update: Sep 4th, 2019 (updates are marked with red)

Background: Virginia Tech has more than 25K students. All students have to be managed through different database. A simplified version would be a course manager for our CS3114 class. Now imagine that we have a course with three sections, each section has 80~120 students. How do we maintain this database efficiently with different functions? We need to design operations such as insertion (student data and scores), deletion (remove a student data or a whole section), search (range search or exact search), sorting, and some file processing (load a file, save a file) tasks. We will implement different data structures to perform functions that are necessary for such a database through four projects. Keep in mind that although our background is only for a course of relatively small size, our project design will take into consideration for scaling factors, in other words, for the situation that the number of records reaches hundreds of millions.

For project 1, you will create a simple class database to handle inserting, deleting, and performing queries on student names, and a few operations related to student scores. The goal of project 1 is to practice the Binary Search Tree (BST, see Section 7.11 of the OpenDSA textbook for more information about BST). All operations will be stored in memory for project 1. Later the data may be saved into or loaded from a binary file.

Invocation and I/O Files:

The name of the program is Coursemanager1 (this is the name where Web-CAT expects the main class to be). There is a single command line parameter that specifies the name of the command file. So, the program would be invoked from the command-line as:

```
java Coursemanager1 {command-file}
```

Your program will read a series of commands from the command file, with one command per line. No command line will require more than 80 characters. Each command requires certain outputs, whose details will be described by sample test file outputs that we will post. The formats for the commands are as follows. The commands are free-format in that any number of spaces may come before, between, or after the command name and its parameters. You do not have to worry about the syntax errors, but logic errors should be handled correctly.

section <number>

This command indicates that the following commands are all corresponding to this particular section unless a new "section" command overwrites this one. We will have up to three sections for our database. For this command, you do not have to print out anything, but it will affect the results of following commands. If no section command is given before, the default value is section 1.

insert <first name> <last name>

Insert a blank record for a student with name given by that <first name> and <last name>. This record should be put into a specific section. If that whole name is already in the specific section, you should print out an error message, reject the insertion, but print out the record (name and id) that is associated with this name. Otherwise, a unique 6-digit id should be generated and stored in this section. For the id, the first two digit indicate the section number, and the last four digits will represent the order that particular name was inserted starting from 0001. If a record is later removed, that id is not recycled for future usage. The record should also have a default score 0. Note that the same name may appear in different sections, but is not allowed to appear in the same section. In this command, you don't have to worry about invalid name. 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. When there is a student record with that name already in the system, the ***insert*** command fails, but you have an active student record in the system. The ***score*** command is valid in that case.

search <first name> <last name>
Search for a specific student record in the current section with the name given by <first name> and <last name>. If that whole name is not in the current section, you should print out an error message. Otherwise, print out the record for that student.

score <number>
This command is only valid when it follows an insert command or a successful search command. Otherwise, print an error message. If this command is valid, assign a score given by <number> for the specific student. You don't need to print out any message if it is successful. Note that in project one all scores are integers. 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 consider it invalid, to avoid confusion. In the ***score*** command, the <number> should be in the range between 0 and 100.

remove <first name> <last name>
Remove the record in the current section with the name given by *<first name> <last name>*. If there is no such a student, you should print out an error message and reject the remove command. Otherwise, the record associated with that name will be removed from the BST.

removesection <number>
Remove all records associated with a section specified by <number>. If no <number> is provided, the default section would be the current section. Note that this command only affects records for that specific section. That section still exists but all records associated with it are deleted. Note that this command does not change the id for the current section. If you are currently in section 1 and you ***removesectoin*** 3, you are still in section 1. Also after the removal of a section, the counter for student id should reset. The next insert will start with ??0001, where ?? represents the section id.

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.

dumpsection
Return a "dump" of the BST associated with the current section. The BST dump should print out each BST node (use the in-order traversal). For each BST node, print that node's student record. At the end, please print out the size of the BST.

grade
Go through the whole BST of the current section and check the score of each student, assign

corresponding grades based on our grade table (check our syllabus for it). Report how many students are in each grade level.

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.
Note: This function is designed so that we can practice iterator operator for BST. It is not the focus of project 1. Thus if you feel it is too hard for you, by design you only lose 5 points by finishing this function without the Iterator in project 1.

Implementation and Design Considerations:

In your design and implementation of the project 1, please consider possible future extensions of this project. We will extend your work in project 1 to future projects, so please take this into consideration. You should declare classes for sections and students. All data entries associated with a student should be declared in a class, and all student records in each section should be maintained in a separate BST, sorted by the last name. Note that if the last name is the same, the record order should be decided by the first name. It is possible that a name typed in is in an arbitrary combination of small and capital cases. So you should process the name in advance to make it uniform. Use compareTo() to determine the relative ordering of two records, and to determine if two names are identical.
Note that you are using the BST to maintain your list of student records, but the BST should be a general container class. Therefore, it should not be implemented to know anything about students. Meanwhile, the student class should not know about the BST either, the section class should though.

Be aware that for this project, the BST is being asked to do two things. First, the BST will handle searches on a full name, which acts as the student record's key value. The BST can do this efficiently, as it will organize its records using the name as the search key. But you also need to do things that the BST cannot handle well, including searching by the first name, marking grades, and searching for pairs with same scores. Right now these tasks can be done with traversal functions. You should design in anticipation of using multiple data structures in Project 2 to handle a combination of actions. Make sure that you handle these actions in a general way that does not require the BST to understand its data type.

The biggest implementation difficulty that you are likely to encounter relates to traversing the BST during the ***findpair*** command. The problem is that you need to make a complete traversal of the BST for each student record in the BST and compare it to all other student records. This leads to the question of how do you remember where you are in the “outer loop" of the operation during the processing of the “inner loop" of the operation. One design choice is to augment the BST with an iterator class. An iterator object tracks a current position within the BST, and has a method that permits the position of the iterator object within the BST to move forward. In this way, one iterator object can be tracking the current rectangle in the “outer loop" of the process, while a second iterator can be used to track the current rectangle for the “inner loop". For the ***findpair*** command, you need to determine the pairs without redundant comparisons. Iterator is a required piece of the project, although it will take only a few points though. You can bypass this requirement with a little penalty.

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. Some additional specific advice on a good standard to use:

- You should include a header comment, preceding main(), specifying the compiler and operating system used and the date completed.
- Your header comment should describe what your program does; don't just plagiarize language from this spec.
- 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.
- 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.

We can't help you with your code unless we can understand it. Therefore, you should no bring your code to the TAs 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 textbook code is not designed for the specific purpose of any specific project assignment, and is therefore likely to require modification. It may, however, provide a useful starting point. For external libraries, it is required that you cannot use libraries to avoid the implementation of the major functions in any project. If you are not sure, you can always ask your instructor for the usage of a specific library.

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. It is fine to find a partner across different sections. 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.