Project 3 – Hash Tables (Hash Maps)

Computer Science 2413 – Data Structures – Fall 2019

Due by 11:59 pm CST on Wednesday, 13 November 2019

This project is individual work. Each student must complete this assignment independently.

User Request:

"Create a simple system to read, store, merge, purge, sort, search, and write drilling data using a more complete array library, a linked list library, error checking, user interaction, and providing fast lookup by timestamp."

Objectives:

1	Use C++ file IO to read and write files, while using C++ standard I/O (`cin` and `cout`) for user interaction, using appropriate exception handling and giving appropriate error messages.	*5 points*
2	Encapsulate primitive arrays inside a somewhat complete templated class that provides controlled access to the array data, retains information on array capacity and use, and can be used to store data of any class or primitive type. Integrate appropriate exception handling into the templated array class.	*5 points*
3	Efficiently sort and search the data based on the field specified by the user (using comparator).	*5 points*
4	Store drilling data in a linked list maintained in order, sorted by timestamp. Add and remove entries from the drilling record linked list while maintaining its order. Integrate appropriate exception handling into classes that implement linked lists.	*5 points*
5	Create and use a hash table with drilling record timestamps as keys for efficient insertion and retrieval of drilling data based on time. This hash table must store records within an array and resize appropriately to maintain efficiency.	*25 points*
6	Encapsulate hash tables inside a templated class that provides controlled access to the hash table data, retains information on hash table capacity and load factor, and can be used to store data of any class or primitive type.	*10 points*
7	Handle collisions in the hash table using separate chaining implemented using your `OULinkedList` class.	*10 points*
8	Integrate appropriate exception handling into the templated hash table class.	*5 points*
7	Provide a design document explaining and justifying alternative implementation choices for a hash table.	*10 points*
8	Develop and use an appropriate design.	*10 points*
9	Use proper documentation and formatting.	*10 points*

Description:

For this project, you will revise and improve Driller 2.0 from Project 2 in one important way. You are encouraged to reuse and build on your code from Project 2. *Driller 3.0* will have the same basic functionality as Driller 2.0, but it will have one major change "under the hood"– because it is believed

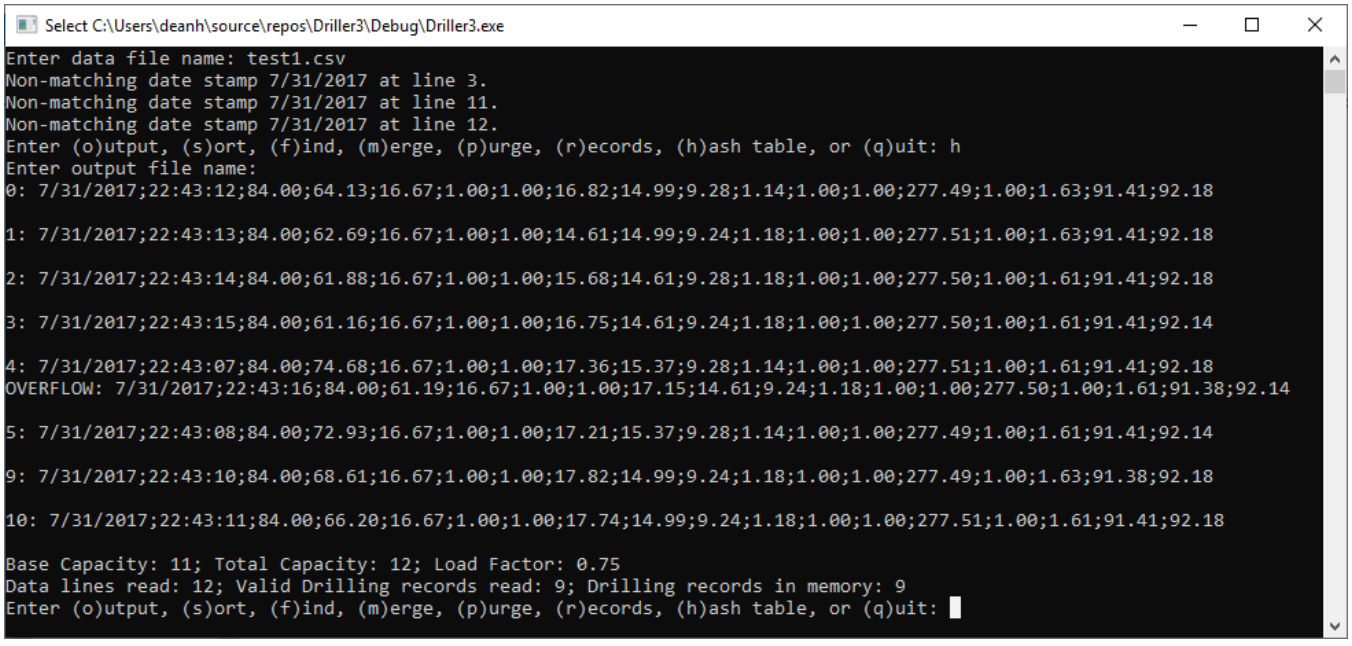
that users will most often want to search for drilling data by timestamp, the list of timestamps will be used as keys to a hash table that stores pointers to the associated data. This will allow for drilling data to be looked up by timestamp in constant time, that is, in Θ(1) time, unless the hash table becomes too full or the data are extremely skewed in some unlikely way. Of course, your table should resize if it becomes too full. (Note that Driller 3.0 will still store the list of drilling data using linked list data structures during initial reading, merging, and purging and will still store the list of drilling data using your templated array for sorting and searching using fields besides timestamp.)

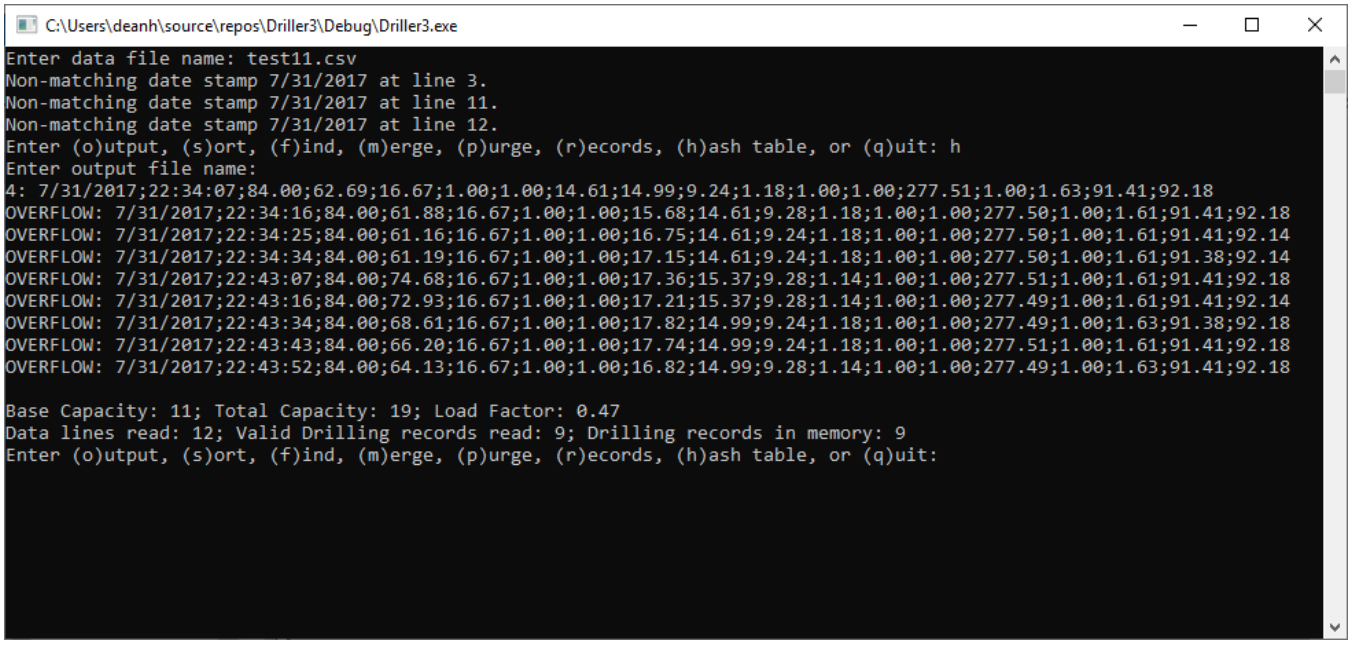
Operational Issues:

From a user interface perspective, Driller 3.0 will behave as described for Driller 2.0, except that there will be one additional data saving/display option, as follows.

"`Enter (o)utput, (s)ort, (f)ind, (m)erge, (p)urge, (r)ecords, (h)ash table, or (q)uit: `"

If the user enters 'h' for hash table display, Driller 3.0 will list the drilling records, one drilling record per line, in the order they are stored in the hash table, each prepended with the bucket number (hash code) where it is stored, followed by a colon. If a bucket is empty, the bucket number and colon should be skipped. If a bucket overflows, the overflow items should be prepended with "OVERFLOW:" and be displayed in their overflow order before proceeding to the next bucket. There should also be a blank line displayed between buckets. All of this will be followed by a blank line, then a line giving some data about the hash table, namely, its "base capacity" (the size of the array, not counting any overflow), its "total capacity" (the size of the array plus any separate chaining links used for overflow), and its load factor. This is immediately followed by the usual line stating the number of data lines read, valid drilling records read, and drilling records in memory. As with the other data saving/displaying options, if the user enters 'h' for hash display, Driller 3.0 will prompt for an output filename and send the output to standard out if no file name is provided. Note that this is thought of as a debug display as it is unlikely to be of use to an end user but may help you to debug your project. Examples follow.





Implementation Issues:

In most areas, Driller 3.0 will be implemented just as was Driller 2.0. This includes how Driller reads files and prints data, carries out user interaction via standard in and standard out, encapsulates C primitive arrays, how exception handling is implemented for arrays and similar classes, and how the list of drilling records is stored in a linked list when data is initially read in and when it is being merged and purged. The big implementation change will be the data structure used in the code to find drilling data based on timestamp. For Driller 3.0, you are no longer allowed to sort and search by timestamp using the array, you need to use a hash table instead. This hash table will use an array of `OULinkedLists` to hold the data and resolve collisions through separate chaining. For this hash table, we will keep the hash function simple by using adding up the ASCII values of the characters in each timestamp and wrapping hash values that are too large for the array using modulus table capacity. However, to avoid too many collisions for likely data subsampling, we will make the table capacity always be a prime number, following a predefined resizing schedule. The table will resize to the next larger size in the schedule when it exceeds its maximum load factor (which will have a default value of 80% but should be able to be set to other values via a constructor), and resize down to the next smaller size in the schedule when it falls below its minimum load factor (which will have a default value of 30% but should be able to be set to other values via the same constructor).

Of course, there are many alternatives that could be selected for hash table implementations. For this reason, you should include a design document with your submission. Please make this a PDF file and name it "**design.pdf**" in your submission. In this document, you should describe one alternative hash function you could have chosen and one alternative collision resolution strategy that you could have chosen, along with a brief analysis of how these alternatives would influenced the complexity and efficiency of your code.

Note that when you first create your hash table in Driller, you will know the amount of data it is to contain. This is because you will wait to create it until you have read in the first file of data. Similarly, when you merge or purge data, you will use your adjusted list to create a new hash table for the adjusted

amount of data. This means that when Driller creates a new hash table, it should specify this value. However, your member function for adding entries to the hash table should be capable of automatic resizing if the table exceeds the maximum load factor or falls below the minimum load factor.

Be sure to **use all provided code**, **use efficient mutator methods** (e.g., don't make new arrays unless doubling or halving the array), and **check whether memory is available** on the stack when using `new` in your `ResizableArray`, `OULinkedList`, and `HashTable` classes and throw `ExceptionMemoryNotAvailable` if `new` returns `NULL`.

Be sure to use good object-oriented design in this project. That includes appropriate use of encapsulation, inheritance, overloading, overriding, accessibility modifiers, etc.

Be sure to use good code documentation. This includes header comments for all classes and methods, explanatory comments for each section of code, meaningful variable and method names, consistent indentation, etc.

You may write your program from scratch or may start from programs for which the source code is freely available on the web or through other sources (such as friends or student organizations). If you do not start from scratch, you must give a complete and accurate accounting of where all of your code came from and indicate which parts are original or changed, and which you got from which other source. Failure to give credit where credit is due is academic fraud and will be dealt with accordingly.

Due Date:

You must submit an electronic copy of your Driller 3.0 project to zyLabs and send the score from zyLabs to Canvas and also submit your design document to the appropriate dropbox in Canvas by **11:59 pm CST on Wednesday, 13 November 2019**.