Project 4 – AVL Trees

Computer Science 2413 – Data Structures – Fall 2019
Due by 11:59 pm CST on Thursday, 5 December 2019

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

User Request:

"Create a simple system to efficiently 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:

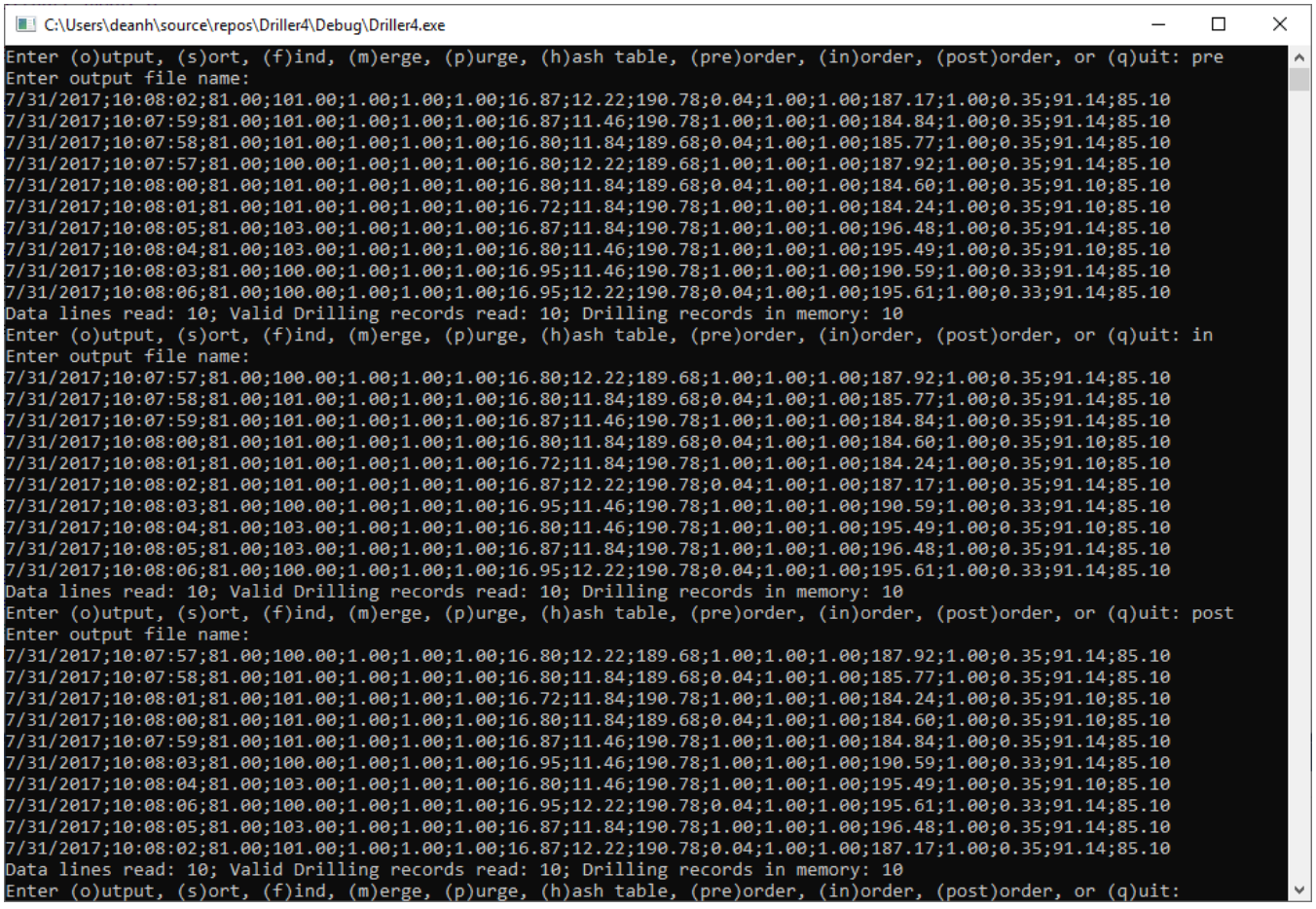
1	Use C++ file IO to read and write files, while using C++ standard I/O (`cin` and `cout`) for user interaction, using appropriate exception handling and giving appropriate error messages.	*5 points*
2	Encapsulate primitive arrays in a templated class that provides controlled access to the data, retains 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	Encapsulate linked lists in a templated class that provides controlled access to the data, retains information on list size, and can be used to store data of any class or primitive type. Integrate appropriate exception handling into the templated linked list class.	*5 points*
5	Encapsulate hash tables in a templated class that provides controlled access to the data, retains information on table capacity and load factor, can be used to store data of any class or primitive type, and handles collisions using separate chaining. Integrate appropriate exception handling into the templated hash table class.	*10 points*
6	Store drilling data in an AVL tree sorted by timestamp.	*20 points*
7	Find drilling data stored in the AVL tree based on timestamp.	*5 points*
8	Remove drilling data from the AVL tree based on timestamp.	*10 points*
9	Provide pre-order, in-order, and post-order traversals of the AVL tree.	*10 points*
10	Encapsulate AVL trees inside a templated class that provides controlled access to the AVL tree data, retains information on AVL tree size (number of entries), and can be used to store data of any class or primitive type.	*10 points*
11	Integrate appropriate exception handling into the templated AVL tree class.	*5 points*
12	Develop and use an appropriate design.	*5 points*
13	Use proper documentation and formatting.	*5 points*

Description:

For this project, you will revise and improve Driller 3.0 from Project 3 in one important way. You are encouraged to reuse and build on your code from Project 3. *Driller 4.0* will have the same basic functionality as Driller 3.0 but it will have one major change "under the hood"– because it was very time-inefficient to keep the list of drilling records in a linked list that was always sorted by timestamp while data was read in, Driller 4.0 will instead keep an AVL tree of drilling data using timestamps as keys. Note that Driller 4.0 will still copy the data to a hash table for fast lookup by timestamp and will still store the data to be sorted on any field and searched using any field besides timestamp using a resizable array.

Operational Issues:

From a user interface perspective, Driller 4.0 will behave as described for Driller 3.0, except that the merge and purge operations may take noticeably less time and there will be three additional data printing/display options: 'pre', 'in', and 'post' for pre-, in-, and post-order traversal of the AVL tree, respectively. Note that, like the 'h' display of Driller 3.0, these are thought of as a debug displays as they are unlikely to be of use to an end user but may help you to debug your project. An example follows.



Relatedly, note that option 'r' has been removed, since the associated linked list has been removed.

Implementation Issues:

In most areas, Driller 4.0 will be implemented just as was Driller 3.0. This includes how Driller reads files and prints data; carries out user interaction via standard in and standard out; encapsulates C primitive arrays; implements exception handling for arrays, linked lists, and hash tables; and stores the lists of drilling records in both an array (for sorting and finding based on most data fields) and a hash table (for fast lookups based on timestamp). The big implementation change will be the data structure used to store the drilling records while reading files and merging and purging data. For Driller 4.0, you are no longer allowed to store this data in a linked list; you need to use an AVL tree instead.

In particular, any time a file is read in, each valid record should be added, one at a time, to an initially empty AVL tree, with invalid records rejected without any attempt to add them to the tree and duplicates rejected when the `insert` method of your AVL class returns `false`.

- If the file just read in is the first data file read in when Driller 4.0 starts up, the tree will be retained for future use (such as merging and purging, and for the debug displays of pre-order, in-order, and post-order traversal) and its data items will be copied to the resizable array and the hash table.
- If the file just read in was read in for the purpose of merging, then the just-created AVL tree and the previously existing AVL tree will be merged by conducting an in-order traversal of the new tree (containing the items to be merged), adding all the unique items and replacing all of the duplicates in the original tree. During this traversal, the unique items will also be added to the hash table and the newer duplicates will replace the older ones in the hash table. After this merge operation, the old array will be discarded and the data items from the revised tree will be added to the resizable array.
- If the file just read in was read in for the purpose of purging, then the just-created AVL tree and the previously existing AVL tree will be combined by conducting an in-order traversal of the new tree (containing the records to be purged) and calling `remove()` on each of its records. During this traversal, the records to be purged will also be purged from the hash table by calling `remove()`. After this purge operation, the tree of records to be purged will be discarded, along with the old array and the data items from the revised tree will be copied to the resizable array.

Note that your `remove()` function in `AVLTree` must replace an interior node with its *in-order successor*.

Note that each time you create a new `ResizableArray` from an AVL tree, you should use a constructor that specifies how many elements you plan to `insert`. Moreover, copying to the resizable array should be accomplished by conducting an in-order traversal of the tree and calling `add()` for each record.

While this description suggests that using the AVL tree in place of the linked list for reading in files and checking for duplicates is more time-efficient, you should consider whether that is really the case. 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 analyze the time efficiency of reading into an AVL rather than a linked list while checking for duplicates. In this document, you should also analyze the time and space efficiency of the processes described above for merging and purging data by reading the new file into its own tree, then traversing it to add/remove records to/from the existing tree.

Be sure to **use all provided code**, **use efficient mutator methods**, and **check whether memory is available** on the stack when using `new` in your `ResizableArray`, `OULinkedList`, `HashTable`, and `AVLTree` 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 4.0 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 Thursday, 5 December 2019**.