Project 3 – Hash Tables (Hash Maps)

Computer Science 2413 – Data Structures – Fall 2016

Due by 11:00 pm CST on Thursday, 17 November 2016

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

User Request:

"Create a simple system to read, merge, purge, sort, search, and write merchant vessel data with error checking and providing fast lookup by name."

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 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. Integrate appropriate exception handling into classes that implement searching and sorting.	*5 points*
4	Store merchant vessel data in a linked list maintained in order, sorted by country name. Remove entries from the merchant vessel linked list while maintaining its order. Integrate appropriate exception handling into classes that implement linked lists.	*5 points*
5	Use a linked hash table with country names as keys for efficient retrieval of merchant vessel data based on country name.	*25 points*
6	Encapsulate linked hash tables inside a templated class that provides controlled access to the linked hash table data, retains information on linked hash table capacity and load factor, and can be used to store data of any class or primitive type.	*10 points*
7	Integrate appropriate exception handling into the templated linked hash table class.	*5 points*
8	Provide a design document explaining and justifying implementation choices for your linked hash table.	*10 points*
9	Use a sophisticated hash function and/or collision resolution method.	*10 bonus points*
►	Develop and use an appropriate design.	*15 points*
►	Use proper documentation and formatting.	*15 points*

Description:

For this project, you will revise and improve MVP2.0 from Project 2 in one important way. You are encouraged to reuse and build on your code from Project 2. MVP3.0 will have the same basic functionality as MVP2.0 but it will have one major change "under the hood"– because it is believed that users will most often want to search for merchant vessel data by country name, the list of country names will be used as keys to a hash table that stores pointers to the associated data. This will allow for

merchant vessel data to be looked up by country name in constant time, that is, in Θ(1) time. (Note that MVP3.0 will still store the list of merchant vessel data using linked list data structures during merging and purging and will still store the list of merchant vessel data using a resizable array during sorting and searching using fields besides country name.)

Operational Issues:

From a user interface perspective, MVP3.0 will behave as described for MVP2.0, except that there will be two additional data printing/display option, as follows.

If the user enters 'H' for Hash display, MVP3.0 will list the merchant vessel data by country, one country per line, in the order they are stored in the hash table, each prepended with the bucket number (hash code) where it is stored. If a bucket is empty, the bucket number should still be displayed, followed by "NULL." If a bucket contains more than one item, each should be listed in its order within the bucket, one per line. If a bucket overflows somewhere besides another bucket, 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. 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.

If the user enters 'L' for Linked display, MVP3.0 will list the merchant vessel data by country, one country per line, in the order they are stored in the linked list embedded in the hash table. Note that this should be the same order as seen with the 'A' (alphabetical) option that runs through the linked list used for merging and purging.

Implementation Issues:

In most areas, MVP3.0 will be implemented just as was MVP2.0. This includes how MVP 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 merchant vessel data is stored in a linked list when data is being merged and purged. The big implementation change will be the data structure used in the code to find merchant vessel data based on country name. For MVP3.0, you are no longer allowed to sort and search by country name using the array, you need to use a hash table instead. For this reason, you should use a linked hash table where the linked list interwoven into the table is ordered by country name.

Beyond these requirements, you have a lot of freedom to design your hash table implementation. For this reason, you should include an 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 the hash function you chose, the bucket size you chose for your hash table, the hash collision resolution strategy that you chose, and the load factor you chose, along with design justifications for each.

You are free to use any hash function you wish, so long as it results in hash codes within the bounds for your hash table. If you choose to implement a hash function that distributes keys close to uniformly and randomly throughout the table, you should note this in your documentation for the TA, so that he can assign you bonus points (up to 5 points, depending on the sophistication of the hash function used and your explanation of why its distribution is close to uniform and random).

You are free to use any bucket size for your hash table and any hash collision resolution strategy you wish. You should justify both of these design choices, noting that some bucket sizes may be more appropriate for some collision resolution strategies and vice versa. If you choose to implement a particularly sophisticated collision resolution strategy, you should note this in your documentation for the TA, so that he can assign you bonus points (up to 5 points, depending on the sophistication of the

strategy and your explanation of why it is appropriate for your hash table).

Note that when you first create your hash table, you will know the amount of data it is to contain. This is because you can wait to create it until you have read in the first file of data. Similarly, when you merge or purge data, you can use your adjusted list to create a new hash table for the adjusted amount of data. However, you should not attempt to fill your hash table completely, because this will almost surely cause too many collisions and degrade its performance below that of a sorted array. Instead, you should choose an appropriate load factor for your table.

If you choose to do so, you may use the linked list of the hash table for the merge/purge linked list. **If you choose to do this, be sure to note it in the design document.**

You should implement the display/printing options as follows.

A should be accomplished by traversing the linked list used for merging and purging.

O should be accomplished by traversing the array used for searching and sorting.

L should be accomplished by traversing the linked list embedded within the hash table. (Note that this might be the same linked list as for option A.)

H should be accomplished by traversing the hash table in bucket order. If your buckets are larger than size one, each bucket should be traversed in order before proceeding to the next bucket. If your hash table overflows to anything besides other buckets, the overflow data structures should also be traversed in order before proceeding to the next bucket.

The only libraries you may use for this assignment are `iostream`, `iomanip`, `string`, and `fstream` (`#include <iostream>`, `#include <iomanip>`, `#include <string>`, `#include <fstream>`).

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 source code and design document to the appropriate dropbox in D2L by **11:00 pm CST on Thursday, 17 November 2016**.