Programming Assignment #5

Due Monday, March 28, 2016 at 11:59 pm

This assignment gives you a chance to test your knowledge of hash tables. Specifically, you will develop a closed-addressing-based hash table that uses chaining for collision resolution. The class will be templated and is named ChHashTable (for Chaining Hash Table). It has a simple public interface as follows:

Method	Description
`ChHashTable(const HTConfig& Config, ObjectAllocator* allocator = 0);`	Constructor. *ObjectAllocator* – Since this is a node-based container, it will benefit from using an object allocator. *Config* - The configuration for the hash table. The HTConfig struct has these members: • *InitialTableSize* - The number of slots in the table initially. • *HashFunc* - The hash function used in all cases. • *MaxLoadFactor* - The maximum "fullness" of the table. • *GrowthFactor* - The factor by which the table grows. • *FreeProc* - The method provided by the client that may need to be called when data in the table is removed.
`~ChHashTable();`	Destructor.
`void insert(const char *Key, const T& Data);`	Inserts a Key/Data pair into the table. Throws an exception if the data cannot be inserted. (E_DUPLICATE, E_NO_MEMORY)
`const T& find(const char *Key) const;`	Finds the data in key and returns a constant reference to the data. Throws an exception if Key isn't found. (E_ITEM_NOT_FOUND)
`void remove(const char *Key);`	Removes a Key/Data pair by key. Throws an exception if the pair cannot be removed. (E_ITEM_NOT_FOUND)
`void clear(void);`	Removes all Key/Data pairs in the table, freeing the data if necessary. This does not free the hash table itself.
`HTStats GetStats(void) const;`	Returns a struct that contains information on the status of the table for debugging and testing. The struct is defined in the header file.

Notes:

1) The constructor takes a pointer to an *ObjectAllocator*. You will use this for all allocations/deallocations in your class. You don't own this allocator, so don't destroy it. If this is 0, you will simply use **new** and **delete** in your code to allocate the nodes.
2) When an insertion will cause the maximum load factor to be surpassed, you must grow the table *before* inserting. Since this table uses closed-addressing, the load factor will be greater than 1.
3) If the client provides a callback function for freeing data, you need to call this function and pass the data upon deleting the node that contains the data. The client may pass 0 (NULL), meaning that there is nothing to free. PLEASE CHECK THIS VALUE AND DON'T BLINDLY CALL IT.
4) There is a public method that exposes the internal state of the table (e.g. number of probes, expansions, etc.) This facilitates testing and debugging at the client level.
5) It is important that you track the number of probes (searches) and expansions (growing the table) throughout the lifetime of the table correctly. You should strive to match the output of the sample driver. Be sure to count **every** time you search for an item in the table. This includes inserts, deletes, and searches (find). Anytime you look for an item after hashing its key, this is a probe. The sample driver demonstrates this in detail. Please look at the driver as an example.
6) Note that you are inserting items at the head of the lists. You still must look through the entire list when inserting a new item to detect duplicates. If you are re-inserting an item (due to having just grown the table) you **DO NOT** need to walk the entire list. (You're re-inserting, therefore, the item can't be in the table yet. Failure to account for this will cause your probe count to be off.)
7) Since this implementation uses a node-based container, you should not allocate unnecessary nodes. Specifically, when re-inserting nodes, **DO NOT** delete the old node and create a new one. Just re-insert the existing node into its proper place.
8) You are not allowed to use anything from the STL (e.g. vector, list, etc.), so you'll need to do the array (for the table) and linked-list (for collisions) management yourself. These tasks should not be challenging for fourth-semester students.
9) There are no copy constructors or assignment operators to worry about this time.
10) Additional notes are on the web page for this assignment.

Growing the Table
When the maximum *load factor* will be surpassed, you must expand the table. This means you need to allocate a new table and re-insert all of the existing key/data pairs from the old table into the new one as demonstrated in class. Since the table size has changed, each key will hash to a new index. When expanding the table, you will use the *GrowthFactor* that was supplied to the constructor. To keep the size of the hash table a prime number, use the included *GetClosestPrime* function to calculate the new table size. So in the *GrowTable* method, you would have code similar to this:

```
    double factor = std::ceil(TableSize_ * Config_.GrowthFactor_);  // Need to include <cmath>
unsigned new_size = GetClosestPrime(static_cast<unsigned>(factor)); // Get new prime size
```

Strictly speaking, you don't have to have the table size a prime number when doing closed-addressing. However, this assignment is part of a two-part hashtable assignment so as to compare open-addressing with closed-addressing. I wanted to keep the table sizes the same for both implementations, and the open-addressing technique requires this. You don't have to do much extra work because the prime number generating code is included for you to use.

When deciding whether or not to grow the table, do not first check to see if the inserted item is a duplicate. If you do this, you will have a higher probe count. Since 99.9% of the time the item will not be a duplicate (duplicates are the exceptional case, afterall), you want to grow the table first, and then insert it into the table. This simply means we are proactively growing the table because we expect the inserted item is unlikely to be a duplicate. If the item does turn out to be a duplicate, we will have grown the table unnecessarily. But, again, duplicates are the exception, not the rule, so this will likely have no impact at all on performance.

Testing
As always, testing represents the largest portion of work and insufficient testing is a big reason why a program receives a poor grade. (My driver programs take longer to create than the implementation file itself.) A sample driver program for this assignment is available. You should use the driver program as an example and create additional code to thoroughly test all functionality with a variety of cases. (Don't forget stress testing.) See the sample driver for examples.

Remember that, due to the class being templated, you will include the implementation file at the bottom of the header as we have done in the past:

```
#include "ChHashTable.cpp"
```

What to submit
You must submit your header and implementation files (ChHashTable.h, ChHashTable.cpp) and the compiled help file (index.chm) via the appropriate submission page.

Source Files	Description
ChHashTable.h	The header files. **No** implementation code allowed (except for the code already included.) The public interface must be exactly as described above.
ChHashTable.cpp	The implementation file. All implementation for ChHashTable code goes here. You must document the functions in detail with Doxygen tags in these files.

Your code must compile using the compilers specified in this assignment to receive credit. Note that you must not submit any other files in the zip file other than the 3 files specified. Details about what to submit and how are posted on the course website and in the syllabus.

Make sure your name and other information is on all documents.