Project #4 - Hash Table Performance

Learning Objectives

- Implement a data structure to meet given specifications
- Design, implement, and use a closed hash table data structure
- Perform empirical analysis of algorithm performance
- Compare theoretical and empirical performance of a data structure

Overview

Your task for this assignment is to implement a closed hash table data structure, and to analyze the time complexity of the insert operation for your hash as a function of the load factor, α, of your hash table.

The HashTable class

Your hash table should be implemented as an array of `MAXHASH` objects of class Record. A Record contains an integer key, and a value, which can be of any type. The implementaiton of class Record will be provided for you in the file `Record.h`. The value of `MAXHASH` should initially be `#defined` to 1000.

Your hash table should accept integer keys, and any type of value. To implement the hash table, you should create a class template called HashTable, implemented inline in the file HashTable.h. Your class should support the following operations (for any class T):

- **`bool HashTable<T>::insert(int key, T value, int& collisions)`** – Insert a new key/value pair into the table. Duplicate keys are not allowed. This function should return **`true`** if the key/value pair is successfully inserted into the hash table, and **`false`** if the pair could not be inserted (for example, due to a duplicate key already found in the table). If the insertion is successful, the number of collisions occuring during the insert operation should be returned in **`collisions`**.

- **`bool HashTable<T>::remove(int key)`** – If there is a record with the given key in the hash table, it is deleted and the function returns **`true`**; otherwise the function returns **`false`**.

- **`bool HashTable<T>::find(int key, T& value)`** – If a record with the given key is found in the hash table, the function returns **`true`** and a copy of the value is returned in `value`. Otherwise the function returns **`false`**.

- **`float HashTable<T>::alpha()`** - returns the current loading factor, α, of the hash table.

- Your hash table should also overload **`operator<<`** such that `cout << myHashTable` prints all the key/value pairs in the table, along with their hash table slot, to cout. *In order to avoid difficulties that occur when combining templates with separate .h and .cpp files, you may include your* `operator<<` *function definition in your HashTable.h file.*

The hash and probe functions

Because this is a closed hash, you will need both a hash function and a probe function. You are free to implement any hash function you like. For example, you may choose to implement some form of the ELFhash, given in Chapter 9 of your textbook, or any other hash function that will work on integer keys. *Remember to cite your source if you use code written by anyone other than yourself!*

Your hash may use double-hashing (a.k.a. random rehash) or pseudo-random probing to resolve collisions. If you use double-hashing, take care to ensure that your probe function will attempt to use every slot in the hash table (e.g. that your stride is coprime with the size of your hash table).

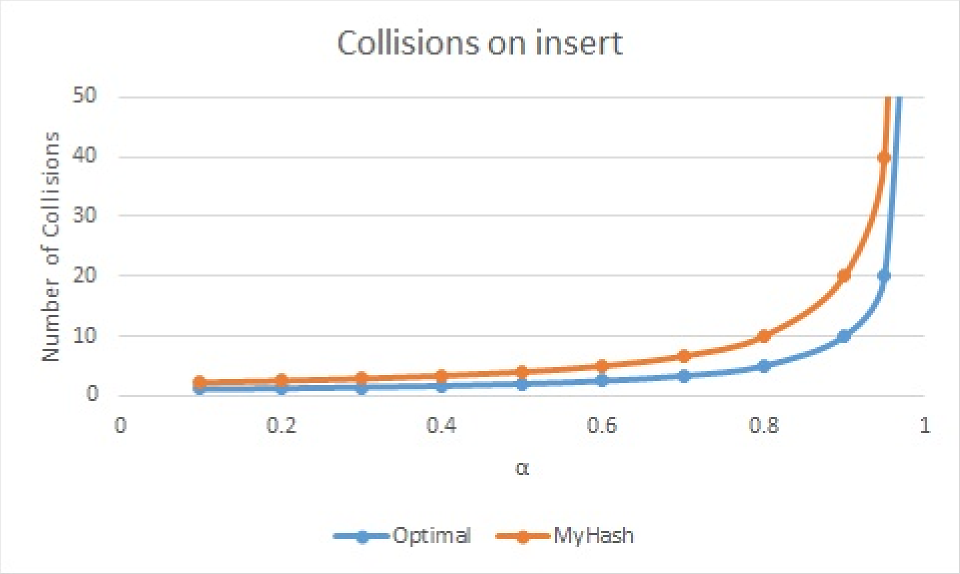
Analysis of your hash

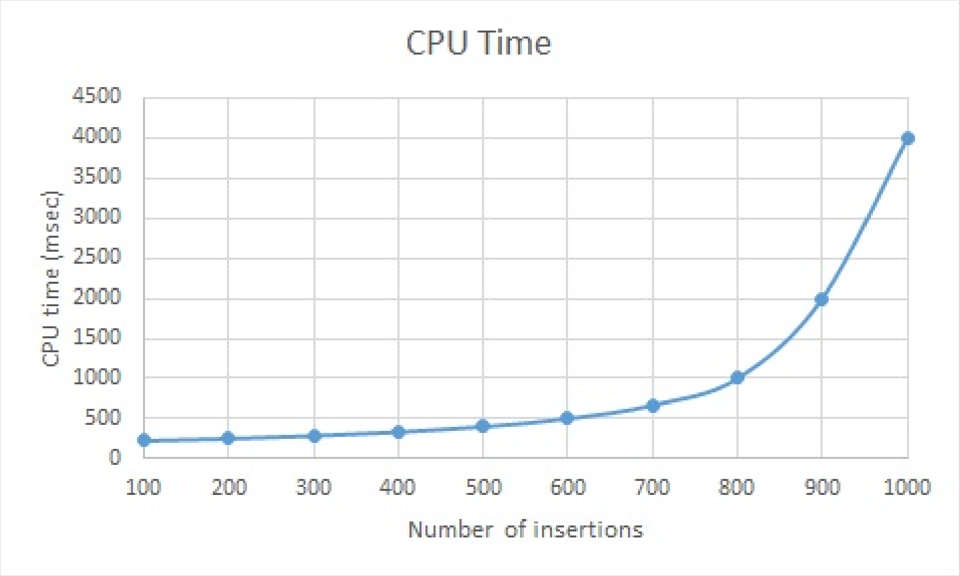
You will need to analyze your hash for performance for both sequential keys and for random keys. First, design a test harness to insert sequential integer keys into your hash. Examine the hashed locations to determine if your hash function is distributing the keys well among the hash table slots. Next, you should test your hash table using a driver that inserts records with random keys into the hash table. Again, determine how well your hash function is distributing the values among hash table slots.

Next you should test the number of probes for insertions as the table becomes more and more full. Based on your test results, you should provide a plot of the average number of probes for inserts as a function of alpha. (This means you will need to do multiple inserts/removes in order to determine the mean value. You should describe your methods for determining this value in your report.) Your plot should compare the performance of your hash to the optimal value (1/(1-α)), as shown below. In addition, you should use Visual Studio's profiling tools to determine the total cpu time for *n* insertions into your hash table, where *n* ranges from 100 to 1000, in steps of 100.

You should prepare a brief report explaining your choice of hash function, describing your testing and data collection methods, and comparing the performance of your hash table to the theoretical optimal performance. Describe any primary and/or secondary clustering you observe, and explain how it is affecting the performance of your hash table. You should also note whether your CPU time experiements match the results of your collision tests, and explain any differences you observe.

Example plots for collisions and CPU time:





Turn in and Grading

- The HashTable class should be implemented as an inline class in the file HashTable.h -- it should be implemented as a template to allow any type of value to be stored in the table.
- Please zip your entire project directory into a single file called Project4_YourLastName.zip.
- Also turn in your report, including your performance graphs, as a single PDF file.

This project is worth 50 points, distributed as follows:

Task	Points
`HashTable::insert` stores key/value pairs in the hash table using appropriate hashing and collision resolution, and correctly rejects duplicate keys and reports correct collision counts. Insert correctly re-uses space from previously-deleted records.	5
`HashTable::find` correctly finds records in the hash table using appropriate hashing and collision resolution, and returns either the value associated with the search key or `false` when the requested key is not present in the hash table.	5

`HashTable::remove` correctly finds and deletes records from the table, without interfering with subsequent search or insert operations.	5
`AVLTree::alpha` correctly calculates and returns the load factor of the table.	5
`operator<<` is correctly overloaded to print the hash table slot, key, and value of all records in the table	5
The hash table avoids excessive primary and secondary clustering	5
Code is well organized, well documented, and properly formatted. Variable names are clear, and readable. Classes are declared and implemented in seperate (.cpp and .h) files.	5
The HashTable is implemented as a template, allowing any type for the record values, and is correctly implemented in separate .h and .cpp files.	5
The report clearly explains the hash function and probe functions used. Collision and CPU time graphs, as described above, are included and correct. The report correctly identifies any primary/secondary clustering that is occuring in the hash table. The report is readable and clear.	10