CS 3100 – Data Structures and Algorithms

Project #3 – 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 a *non-negative* integer key, and a value, which can be of any type. You will need to implement Record as a template class.

The value of MAXHASH should initially be #defined as 1000 when you turn in your project, though you may need to use a smaller value during development and testing of your code in order to explore its behavior.

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

bool insert(int key, T value, int& collisions) – Create a new Record object for this key/value pair and insert it into the table, if possible. 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 or because there is no more room). If the insertion is successful, the value of the parameter `collisions` should be increased by the number of collisions that occurred during the insert operation.

bool remove(int key, T& value) – If there is a record with the given key in the hash table, the parameter `value` is set to that record's value, the record is removed from the hash table, and the function returns **true**; otherwise the function returns **false** and the parameter `value` has no meaning.

bool 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** and the parameter `value` has no meaning.

float 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. If a particular slot of the hash table is empty, it should not be included in the output. *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.*

Keep in mind that some supplementary methods will also likely be needed. For example, you will need appropriate constructors in both your HashTable and Record classes, and getter/setter methods in your Record class.

You can do basic testing of your hash table using the test.cpp file provided.

The hash and probe functions

For this project, you will need to implement two hash functions and two probe functions. One hash function should just be the key modulo MAXHASH. Likewise, one probe function should be linear probing.

For the other hash function, you are free to implement anything 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! Also, keep in mind that you should not submit any code that you do not understand – I reserve the right to ask you to explain how your hash function works before giving you a grade.*

For the second probe function, you 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

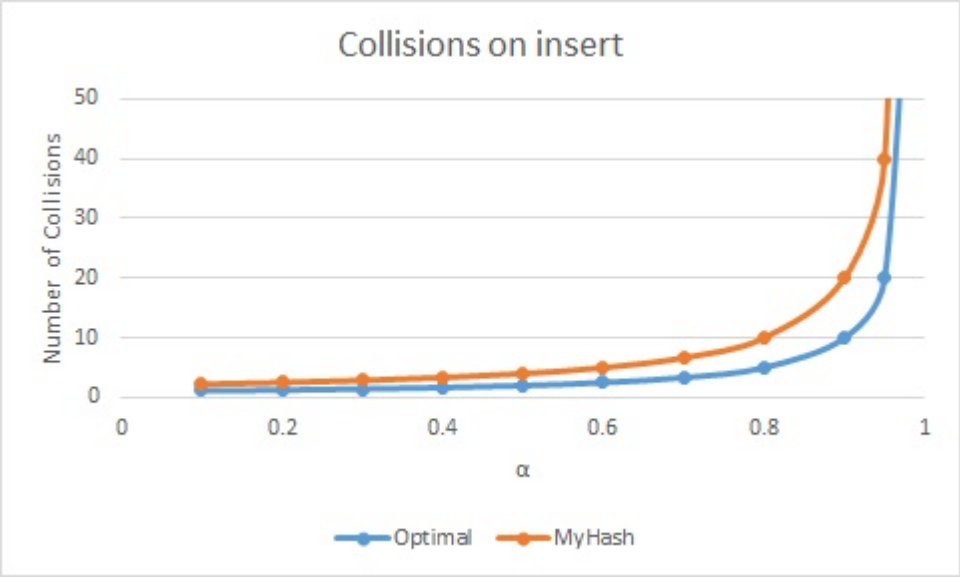
You will need to analyze your hash for performance for both sequential keys and for random keys. First, design a test harness to insert sequential integer keys into your hash. For each combination of hash and probe function, examine the hashed locations to determine if your approach is distributing the 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 ($1/(1-\alpha)$), as shown below.

You should prepare a brief report comparing and contrasting all four combinations of hash and probe 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.

Example plots for collisions:

Note that instead of one MyHash line, you will have four lines: one for each combination of the two hash functions and two probe functions.



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. Similarly, the Record class should be implemented as an inline class in a file called Record.h. You do not need to turn in your driver/testing code.

Please zip your entire project directory into a single file called Project3_YourLastName.zip.
Also turn in your report, including your performance graphs, as a single PDF file.

This project is worth 100 points, distributed as follows:

[10] The hash table stores Record objects, not just keys

[5] The insert method adds new items to the hash table when possible

[5] The insert method rejects duplicate keys

[5] The insert method rejects insertions when the table if full

[5] The insert method re-uses space from previously deleted records (i.e. it overwrites tombstones)

[5] The find method correctly determines when the requested key is in the hash table

[5] The find method correctly returns false when the requested key is not in the hash table

[5] The find method correctly fills in the value parameter when the requested key is in the hash table

[5] The remove method correctly deletes items from the table when they exist

[5] The remove method does not interfere with subsequent search or insert operations (i.e. tombstones are placed correctly)

[5] The << operator is correctly overloaded to print the hash table slot, key, and value of all records in the table

[5] The alpha method correctly calculates and returns the load factor of the table

[5] The hash table avoids excessive primary and secondary clustering

[10] Code is well organized, well documented, and properly formatted. Variable names are clear, and readable.

[20] The report clearly explains the hash function and probe functions used. A collision graph, as described above, are included and correct. The report correctly identifies any primary/secondary clustering that is occurring in the hash table. The report is readable and clear.