HashMaps : Building and Searching
Lab Assignment #9

by
Justin Anthony Timberlake

CS 303 Algorithms and Data Structures
October 25, 2017

1. Problem Specification

The purpose of this lab was to first implement a HashMap class with the methods for puting new objects of type HashEntry into the table with a specific hash function as well as with either linear or quadratic probing, printing out the values in an the table (optional), and finally methods to get a corresponding String value given a key in the form of a long. Finally, values for the UPC records from the last two labs were to be read into 3 different hash tables (one for each hashing method), and the associated runtime to put and get (with the entries in the same "inpus.dat" file as before) for each kind of hashing was to be printed out for comparison.

2. Program Design

The steps taken to develop the program were:

a) Using the classes HashMap and HashEntry provided, develop the put(), get(), as well as linear and quadratic probing methods for the three kinds of hashing into a HashMap using objects of HashEntry type.

b) Develop test cases for each kind of hashing before also developing corresponding get() methods for linear and quadratic probing methods. Also develop a printing method in the HashMap class to help with testing by showing all values in a given HashMap to ensure they are hashing to the correct places.

c) While initially stumped with an infinite loop when putting values from the UPC records into a HashMap, I adjusted the size to 4 times the value of total records in the file (4 * 177,650 = 710,600), so that the initial given hash function and its secondary function ((7*hash + 1)%table_size) would not yield an infinite loop for any values being put into the table.

d) Copied over the same print statements from the last two lab assignments for searching for the values from "input.dat" in the existing tables and made slight modifications to fit for each of the HashMaps.

No additional methods were used in the primary (driver) class.

The following methods were used in secondary class, HashEntry.

a) HashEntry(long key, String value) - constructor for objects of type HashEntry, setting the value equal to the given String and the key value equal to the given long.

b) getKey() - returns the key long of the object.

c) getValue() - returns the value String of the object.

d) setValue(String val) – sets the value attribute of the object equal to "val".

The following methods were used in secondary class, HashMap.

a) HashMap(int size) - constructor for objects of type HashMap, setting the TABLE_SIZE equal to size and the table equal to a new array of HashEntry objects (length of array is size as well).

b) get(long key) – takes a given long key and hashes into the table with it, searching for the associated HashEntry with that exact long. It only checks the places which could have been hashed to in the put() method (using the same hash function). Once it finds an empty spot in the table, it will terminate and return the string of "not here" as there is no other place in which the associated HashEntry could be located in the table.

c) put(long key, String value) – takes a given long key and String value and hashes a HashEntry object made from them into the table. It does so using the hash function of hash = key %TABLE_SIZE and a secondary function of hash = (7*hash+1)%TABLE_SIZE for when there are collisions and the HashEntry cannot be placed in the initial spot it is hashed to.

d) printTable() - this will print every position in the current HashMap table, including empty positions along with the number of the positional index. It was not used in the end result, but was helpful for testing purposes.

e) linearProbe(long key, String value) – exactly the same type of method as put(), but the secondary hash function is (hash+1)%TABLE_SIZE

f) getLinearProbe(long key) – exactly the same type of method as get(), but it will only return the values of HashEntries which were hashed in with linearProbe, given collisions occurred.

g) quadraticProbe(long key, String value) – exactly the same type of method as put(), but the secondary hash function is (hash+square*square)%TABLE_SIZE, where square increments from 1 onward so long as there is a collision with the given HashEntry.

h) getQuadraticProbe(long key) – exactly the same type of method as get(), but it will only return the values of HashEntries which were hashed in with quadraticProbe, given collisions occurred.

3. Testing Plan

Once the given classes had all of their methods properly implemented, the only thing to do was to use print statements to compare the times it took to build a HashMap with all of the given UPC values using the first hash function, linear, and quadratic probing. The associated values for each set of building and searching operations is given below.

4. Test Cases

For building the given 177, 650 records in the “UPC.csv” file into a HashMap, the total time elapsed varied a lot for some reason on all types of hashing. However, one given result said that it took 42,939,662 nanoseconds for standard put(), 53,355,364 nanoseconds for linear probing, and 27,175,095 nanoseconds for the quadratic probing methods of building the HashMap.

In contrast, for building a RBTree (with randomized values), it took 1,779,015,687 nanoseconds in a given test, whereas giving an already sorted list of items to the BST took 76,461,176,294 nanoseconds.

For searching the given 17 values in the “input.dat” file out of the aforementioned 177,650 records, the times elapsed for standard get(), getLinearProbe() and getQuadraticProbe() respectively were 21,333 nanoseconds, 11,852 nanoseconds, and 20,938 nanoseconds. However, these values also varied greatly.

Meanwhile, the total time elapsed for the RBTree was 142,619 nanoseconds, where this same search took 1,986,371 nanoseconds with the BST.

5. Analysis and Conclusions

Though the numbers for each test did vary quite a bit from each experiment to the next, at least in this experiment, it seems that across the board, the fastest build time is using linear probing with a HashMap. For searching the given values within said HashMap or an RBTree or BST, the fastest search method appears to be using linear probing. However, if we weigh that with the significantly longer build time for linear probing, then it appears that quadratic probing truly is the fastest and most efficient in net build and search time given these values for building and searching.

6. References

All code in this assignment was either presented in prior labs and assignments or provided through means of pseudocode within the in-class portion of the lab.