# An Empirical Analysis of Hashing Algorithms

Luke Glayat
College of Charleston
CSCI 230
Dr. Munsell

## Introduction

In computer programming terms, hashing algorithms are the key behind implementing a hashmap, also known as a dictionary or hashtable. There are various ways to implement one, however they all follow the same basic principles. The purpose of this data structure is to store values into them and to have extremely fast access to the value via its key which is also its index. It utilizes the benefits of using ArrayLists or Lists, which is the ability to quickly index an item, and a Singly Linked List, which are extremely space efficient. The secret of a hashmap is the hash method which takes in an input from any ADT, and attempts to generate a unique index or key for the object to be stored at. The utility of this can be seen in the time efficiency of the insert, delete, and search methods which theoretically, should yield a constant time efficiency.

The main complication that arises from this hash function is when two objects are inserted into the hash function, and are given the same index. This is what we call *collisions*, and there are two very common methods of tackling this issue. The first method is called *separate chaining* and it is used in *Open Hashing* to store multiple values into one index using pointers to a Linked List of values or another list itself. The second method is named *linear probing* which is used in *Closed Hashing.* Linear Probing simply checks to see if the spot at the generated index is full, and if it is, the function keeps searching for the next available spot to place the object in. These methods of collision-handling both increase the time complexity, however the true determinant of efficiency relies upon the hashing method itself, and how often a collision occurs.

## Hypothesis

My hypothesis is that, given the same hashing method for average search, searching with Open Hashing will be more time efficient on the whole. My prediction is that clustering on Closed Hashmaps can be far more detrimental to the efficiency of the search method, because the entire list itself has the possibility to cluster. The potential of clustering is less harmful on Open Hashing because clustering on only one sub-list in the Open Hashing only affects that sub-list. Also, I would hypothesize that constant time efficiency would only be achieved in an absolutely perfect hashing method which is highly improbable because of the difficulty of creating a perfect hashing method.

## Methods

To determine the efficiency of both of these Hashing methods, I created two java classes (OpenHashing.java and ClosedHashing.java) and ran them through a series of tests in the java class (TestHashing.java). The implementation of the ClosedHashing class is simply an array of fixed size which was twice the size of the array of strings I tested with. I chose to implement OpenHashing with an ArrayList the size of the string array, with each subarray being an ArrayList as well. The ArrayList class features a grow method which automatically creates a new Array List of double the size of the previous one if the max capacity is reached and something new is inserted. Iteration is still necessary for every sub-array however, giving me the theoretical time efficiency of using a Singly Linked List for this part.

Two hashing methods were used for testing on each Open and Closed. The first hashing method iterates through each character in the string passed in and computes the summation of an integer named *h*, whose initial value is 7. For each character *h* is multiplied by 29, added to the value of the k sub I character, and then remainder division is used by the size of the size of the Hashmap. The purpose of multiplying the integer *h* by 29 is because prime numbers are quite commonly used in hashing functions due to the unique products they yield.

```
int hash = 7;
for (int i = 0; i < word.length(); i++) {
    hash = (hash*29 + word.charAt(i))%tableSize;
}
return hash;
```

The second hash method does the same iteration and summation as the first, however the integer *h* is multiplied by the length of the string *l* instead of 29. The purpose this is to hopefully utilize a more unique variable such as the length will have a positive effect on distributing the keys more evenly.

```
int hash = 7;
for (int i = 0; i < word.length(); i++) {
    hash = (hash*word.length() + word.charAt(i))%tableSize;
}
return hash;
```

For testing purposes, a class named TestHashing.java was developed. I then acquired a large text file in order to add every word in the file to my own Hashmaps so I could get a practical example of how efficient they both may be. This file was passed through a helper method to process the text into a string array without any special characters or spaces.

## Results

When processed, the file yielded a string array of with the size of 567. In every test hashmap, I populated them with the first half of the string arrays elements. Once they were ready for analysis, I ran a nanosecond timer and searched for every element that was inserted into each hashmap. Since the only the first half were in

the hashmaps, half of the searches returned false. This yielded my results for "Successful and Unsuccessful Searches". The x-axis indicates the elapsed search time in nanoseconds, with the y-axis showing to the trial number.

| | Open | Closed |
|---|---|---|
| Load Factor | 1.00 | .5 |
| Successful Searches | 283 | 283 |
| Unsuccessful Searches | 284 | 284 |

**Table 1. Load Factor and amount of searches done for testing**

## Closed Hashing

When the constructors for the hashmaps were developed, I passed specific sizes for each of the Open and Closed tests. The Closed method used an array of twice the size of the string array for the possibility of no collisions with an ideal hashing method. This ensures all of the hashmaps never get too full, and comparison between the methods will be as accurate as possible. When comparing the results of the two methods, the first one was slightly more efficient. This can be seen from Figures 1 through 4. The total calculated average for the successful search tests were 1206 and 1105 nanoseconds respectively. For the unsuccessful tests were much more similar that the results seemed negligible, however still indicated that the second hashing method outperformed the first slightly with averages of 580 compared to 863 nanoseconds. It appears that using the strings length for the hashing method instead of the same prime number 29 was slightly better at distributing the keys more evenly.

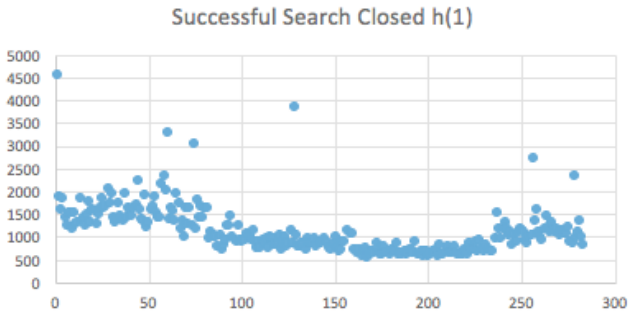


**Figure 1. Result of the first hashing function in a using closed hashing shown in nanoseconds plotted against the number of successful searches**

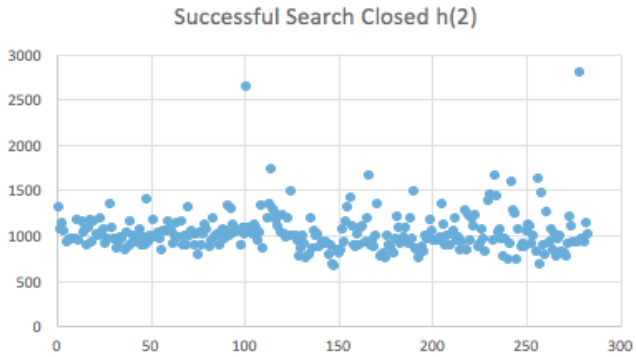


**Figure 2. Result of the second hashing function in a using closed hashing shown in nanoseconds plotted against the number of successful searches**

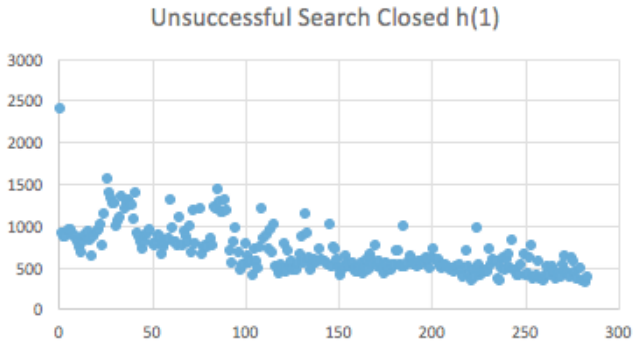


**Figure 3. Result of the first hashing function in a using closed hashing shown in nanoseconds plotted against the number of unsuccessful searches**

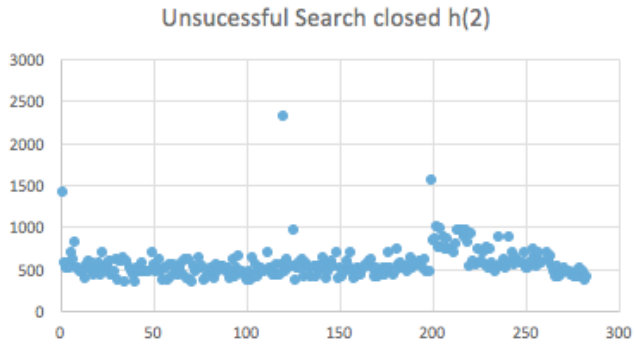


**Figure 4. Result of the second hashing function in a using closed hashing shown in nanoseconds plotted against the number of unsuccessful searches**

In both tests the second method outperformed the other. The conclusion still cannot be drawn however that the hashmap provides a constant average time efficiency.

| | Successful | Unsuccessful |
|---|---|---|
| Hash Method 1 | 1206 | 863 |
| Hash Method 2 | 1778 | 789 |

**Table 2. Closed hashing search averages**

## Open Hashing

In similar manner to the Closed Hashing tests, I passed in the exact size of the array into the constructors of the Open Hashing classes so that, in a perfect world, the elements in the array could all be distributed evenly through the map if the hashing method. From figures 4 through 8 the results of the Open Hashing test are displayed. Judging from this set of results, it there is a hint that my hypothesis may have been correct. Specifically, every test result in this category was slightly more time efficient than the Closed method.

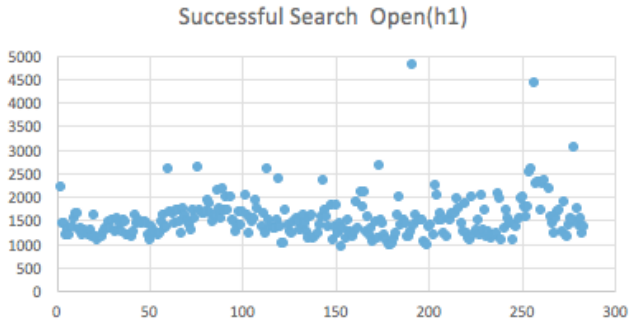


**Figure 5. Result of the first hashing function in a using open hashing shown in nanoseconds plotted against the number of successful searches**

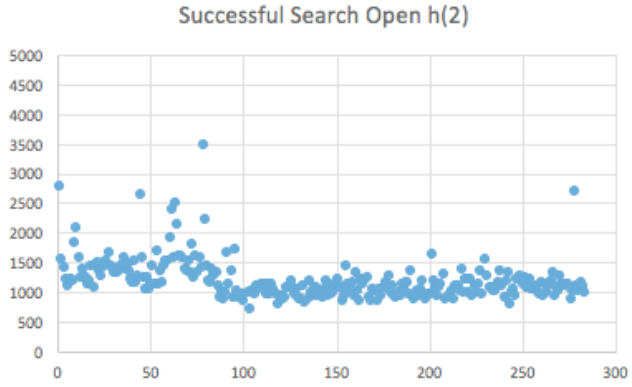


**Figure 6. Result of the second hashing function in a using open hashing shown in nanoseconds plotted against the number of successful searches**

Similar to the Closed Hashing test, the average unsuccessful searches are about 20% faster than the average successful as well. One thing to point out is that although the distribution of search times seems to slow down in some more of the early tests, a large majority of every single test ran has very few outliers and is more than not distributed very well. Regarding the outliers, clustering does still appear to be present on a rare occasion when searching.

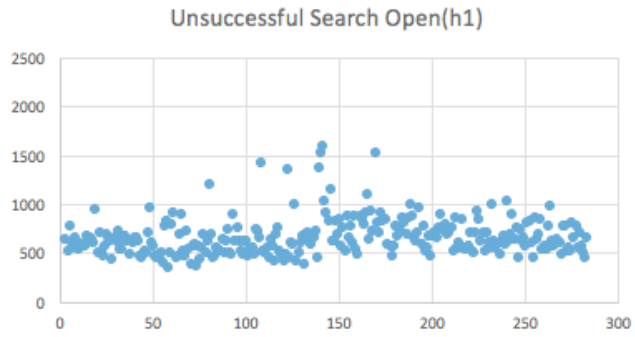


**Figure 7. Result of the first hashing function in a using open hashing shown in nanoseconds plotted against the number of unsuccessful searches**

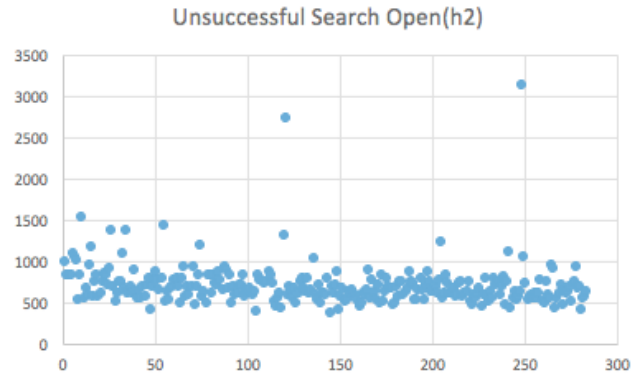


**Figure 8. Result of the second hashing function in a using open hashing shown in nanoseconds plotted against the number of unsuccessful searches**

| | Successful | Unsuccessful |
|---|---|---|
| Hash Method 1 | 1106 | 580 |
| Hash Method 2 | 1284 | 715 |

**Table 3. Open hashing search averages**

## Conclusion

In conclusion, an empirical analysis done between Open and Closed Hashing performance testing as well as two different hashing methods yielded important takeaways on whether my hypothesis was correct. The tests did in fact support my hypothesis. Firstly, based upon this data taken from the tests I believe that an average, constant time efficiency was clearly not achieved. In the comparison between the performance between Open and Closed, the results pointed to my hypothesis being correct that the use of separate chaining is more efficient than linear probing. It should be noted that there were outliers in both result sets indicating the evidence of clustering on both sides. In addition, the second hashing method used outperformed the first method on every single sample taken most due to using the strings length instead of an arbitrary prime number.

## REFERENCES


[1] Levintin, A. Introduction to the Design and Analysis of Algorithms. Pearson, 2011.

[2] Giles, C. Scot. USING THE ANSI DRIVER. Available from http://www.textfiles.com/programming/ansi_tut.txt Accessed 10 December 2016