Runtime Efficiency Analysis of Successful and Unsuccessful Searches in Open and Closed Hashing Functions

Sarah Nicholson
College of Charleston
973 C of C Complex
Charleston, SC 29424
(706) 513-8656
nicholsonsr@g.cofc.edu

ABSTRACT
In this paper, the average case time efficiencies of successful and unsuccessful searches of open and closed hashing will be explored. This will be done using two test files, two different hashing functions, and open and closed hash tables of different lengths. Also, the load factor will be used to determine the time efficiency of successful and unsuccessful searches. The paper will also explain how I developed the methods that are in each class, and what I conclude in the end.

General Terms
Your general terms must be any of the following 9 designated terms: Hashing, Open Hashing, Closed Hashing, Data Structures, Quadratic Probing, Separate Chaining, Run Time Efficiency, Open Addressing, Load Factor.

Keywords
Open Hashing, Closed Hashing, Hash Functions

1. INTRODUCTION
Hashing is based on the idea of equally distributing keys among a one-dimensional array, and in this report, two ways will be discussed. They are open hashing and closed hashing, of which both have an $\Theta(1)$ average runtime and $\Theta(n)$ worst case runtime when it comes to unsuccessful search and successful search. The number of comparisons it takes to find an element and the time it takes to do so, though, differ. Both open and closed hashing functions' goals are to minimize collisions, but not all collisions are avoidable. Therefore, these hashing functions must be able to resolve collisions. They do so in different ways which will be now be discussed.

1.1 Open Hashing
Open hashing, also called separate chaining, has been termed as such because it is not constricted to a hash table. Instead, it uses linked lists which are attached to the cells of a hash table to store the keys in. With open hashing, you have searching, insertion, and deletion methods which all have an average runtime of $\Theta(1)$ if the number of keys is about equal to the size of the hash table. Also, the average number of pointers (chain links) inspected in successful searches (S) and unsuccessful searches (U) are $S \approx 1 + \alpha / 2$ and $U = \alpha$, where α is the load factor which is the number of keys inserted into the hash table divided by the size of the table. S and U are also referred to as the average number of comparisons. It is ideal for the load factor to be a number close to 1 (multiple load factors will be used in my experiments and shown in a table) to find the average time efficiency.

1.2 Closed Hashing
Closed hashing, also referred to as open addressing, is a form of hashing where instead of storing keys in linked lists or an array list of arrays, all keys are instead stored in the hash table. This means that the table size has to at least be as large, if not larger, than the number of keys there are for all of the elements to be stored. There are different strategies to resolve collisions, but the method that will be focused on in this paper is quadratic probing. Quadratic probing helps to solve primary clustering within a hash table by checking elements in a sequence such as: $1^2 = 1$ element away, then $2^2 = 4$ elements away, then $3^2 = 9$ elements away, and so on. Once the table is more than half full, from doing quadratic probing, it is difficult to find an empty spot and this new problem that arises because of this is called secondary clustering. This in turn makes quadratic probing space inefficient due to requiring the table to remain less than half full so there are not any collisions. It is very similar to double hashing in that it also jumps to a certain amount of elements away instead of just going to the next cell in the hash table.

1.3 Hypothesis
The hypothesis that I developed is that if I successfully code and implement both open and closed hashing, then the unsuccessful and successful search times for average case will be as follows:

Open Hashing Average Number of Probes for Unsuccessful Searches: $U = \alpha$

Open Hashing Average Number of Probes for Successful Search Time: $S \approx 1 + \alpha / 2$

Closed Hashing Average Number of Probes for Unsuccessful Searches: $U = 1 / (1 - \alpha)$

Closed Hashing Average Number of Probes for Successful Searches: $S \approx (1 / \alpha) (1 + \ln(1 / 1-\alpha))$

I obtained these formulas that will be upheld by my experiments from a webpage from Carnegie Mellon University School of Computer Science. Also, I will discover that the most efficient function between the two is the open hashing function.

2. METHODS
Both the open hashing and closed hashing functions contain the same three methods, search, insert, and delete. Used in this assignment also is a method called hashCode which is able to be implemented by each data structure being used (SinglyLinkedList, ArrayList, etc.) that was developed in this class since they are

objects and therefore are able to inherit this method. It is a clean way of retrieving the hash code value of that said data structure. The hashCode method uses a hashing function within it in order to compute the hash codes of each element. It computes the hash codes by multiplying the key by 31 (a prime number) and then adding to it the value at a certain index. Since this method hashes whichever key you give to it, I used it as one of my hash functions and the other ones I defined separately in both my open and closed hashing classes. The other hash functions I used in my open and closed hashing classes are:

str = Math.abs(str.hashCode() % hashSize) in open hashing (which I found on a Duke University webpage and modified it) and key = Math.abs(key / 31) % hashSize in closed hashing (I just modified the function from the Duke University webpage).

2.1 Search Method

The search method in my open hashing function uses a key and sees if the key is equal to a cell. If the key is equal to the element in the hash list, then the value will be returned. If they are not equal, then a negative 1 will be returned.

The search method in my closed hashing function uses a key and hashes the key. It then searches for that hashed key by going through the hash table and returning null if the element in that spot is null or not equal to the hashed key or returning the element if it is equal.

2.2 Insert Method

In the insert method in open hashing, you enter the string value that you want to insert and it will compute a hash code for it and store it in the corresponding key value. If there is something already in that position, it will still be added since open hashing utilizes chaining. Also, the number of elements in the hash table will increase.

In the insert method in closed hashing, you insert a string value and unlike open hashing, they are each placed i^2 positions away from each other in order to utilize quadratic probing. Two elements cannot be stored in the same position in the table, and in order to make sure a position is found for a certain element, it is best to keep the table less than half full. Also, in order to avoid adding an element to a cell outside of the boundary of the hash table, the modulo operator is used so it is guaranteed that it will stay within the bounds of the table. It will instead loop around to the beginning of the table.

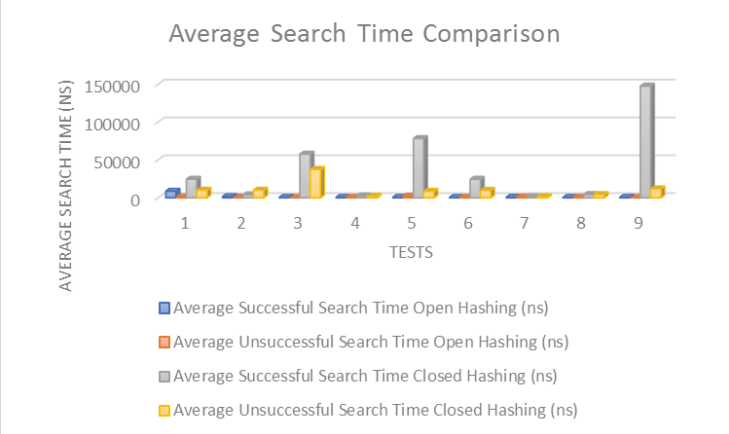
2.3 Delete Method

In the delete method in open hashing, you enter in the value of what you want to delete, and it will look at the hash code of that value in order to delete it. Also, the number of elements in the hash table will decrease.

In the delete method in closed hashing, you can enter in either a key or a value to delete the element in the hash table. It will go to the i^2 one away to find it in order to delete it. If the element you are trying to delete is null, an exception will be thrown, but if it is found, a lazy delete will take place. A lazy delete is an alternative to rehashing everything after a delete. This is when instead of deleting the key, the key is instead replaced with a special hold value. Lazy delete is being used since I am working with text files of a fixed size, so there is no need to rehash the hash table since there is no way of exceeding the text size and having to grow an array, list, etc.

3. RESULTS

This is a graph that outlines the comparisons between open and closed hashing with both unsuccessful and successful searches. Each test used different load factors to analyze the searches. The different load factors are below (rounded off to three decimals). Most of open and closed hashing's load factors only differed by a very small amount also. In order to change the load factors, I either increased or decreased the number of cells in the hash tables.



Load Factors	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9
Open Hashing	1	2.969	0.252	4.010	0.126	0.506	7.340	2.016	0.063
Closed Hashing	1	1	0.252	2.773	0.126	0.506	7.151	1	0.063

3.1 Unsuccessful Searches Results

For the unsuccessful search time for my open and closed hashing, the results upheld the hypothesis. In order to search for an element with open hashing that isn't there, one would on average travel as many probes as the load factor. The smaller the hash table, the bigger the load factor which correlates with there being more probes since it has to search longer chains, also. But for quadratic probing, this is different since you can't go sequentially in order. To check for the unsuccessful searches, I read in a text file into my testing class and compared it with a text file of the same length but each string in the file was different. This guaranteed that none of the strings would match up. I also did the same for closed hashing and as you can see from the graph, the average unsuccessful search times for closed hashing were greater than all the ones for open hashing.

The unsuccessful search times for open and closed hashing are, for the most part, smaller than the successful search times except for when the hash table is much larger than the number of elements in it. For open hashing, this is because you have to search each link in the cell to find the string. For closed hashing, this is because since you are going to the cell i^2 away instead of just going to the next cell.

3.2 Successful Searches Results

For the successful search time for my open and closed hashing, the results upheld the hypothesis. To check for the successful searches, I read in a text file into my testing class and compared that text file with the same text file to make sure that everything

could be found. I did this for closed hashing as well. What I found is that the average successful search time is greater for closed hashing than open hashing. This is because with open hashing, you go directly to the cell that has the same key and search the linked list extending out of that cell for the element. But for closed hashing, since quadratic probing was used, you would have to go to cells i^2 away in order to search for the correct string.

4. CONCLUSION

From my experiments and analyses, the average time unsuccessful and successful searches do indeed uphold the claims of my hypothesis. Also, open hashing being more efficient than closed hashing has been successfully proven. Even though open hashing may be considered simple, it has proven itself to be the most useful and efficient way to do hashing. This is due to how it utilizes lists outside of the hash table, unlike closed hashing which is constricted to the hash table.

5. REFERENCES

[1] Average Case Costs with Separate Chaining. N.p., n.d. Web. 25 Apr. 2017. <http://cseweb.ucsd.edu/~kube/cls/100/Lectures/lec16/lec16-32.html>.

[2] Computer Science Department of Duke University *ArrayListHash*. <https://www.cs.duke.edu/courses/cps100/spring08/code/hash/ArrayListHash.java >.

[3] Shun, Julian. "Hash Tables and Associative Arrays." Algorithms and Data Structures (n.d.): 81-98. Lecture 24 — Hash Tables. Web. 26 Apr. 2017. < http://www.cs.cmu.edu/afs/cs/academic/class/15210-f13/www/lectures/lecture24.pdf >.