Jana Klopsch

U0854469

Analysis 10

1. What does the load factor Î» mean for each of the two collision-resolving strategies (quadratic probing and separate chaining) and for what value of Î» does each strategy have good performance?

For quadratic probing, the load factor, λ, is the number of elements in the hash table compared to the capacity of the hash table. When λ is equal to 0.5, quadratic probing has the best performance. For a $\lambda < 0.5$, when adding an element to the hash table, quadratic probing is guaranteed to find an empty spot as it can check every spot in the table (if the capacity is a prime number).

For separate chaining, λ is the average length of the linked lists. When λ is large, the performance of a hash table with separate chaining is hindered. Although further analysis would need to be done to determine the best size for λ, intuitively I would set it to be equal to the capacity. It seems that as the capacity is larger, we would want the load capacity to also be larger, and similarly for a smaller capacity. (My separate chaining implementation of a hash table does not rehash, so further analysis would need to be done to support my claim.)

2. Give and explain the hashing function you used for BadHashFunctor. Be sure to discuss why you expected it to perform badly (i.e., result in many collisions).

My BadHashFunctor returned the length of the String that was given. This is not a good hash function, as Strings differ by much more than just length. Many Strings are different from each other, by their characters, but they have the same length. Thus, this would be expected to have many collisions.

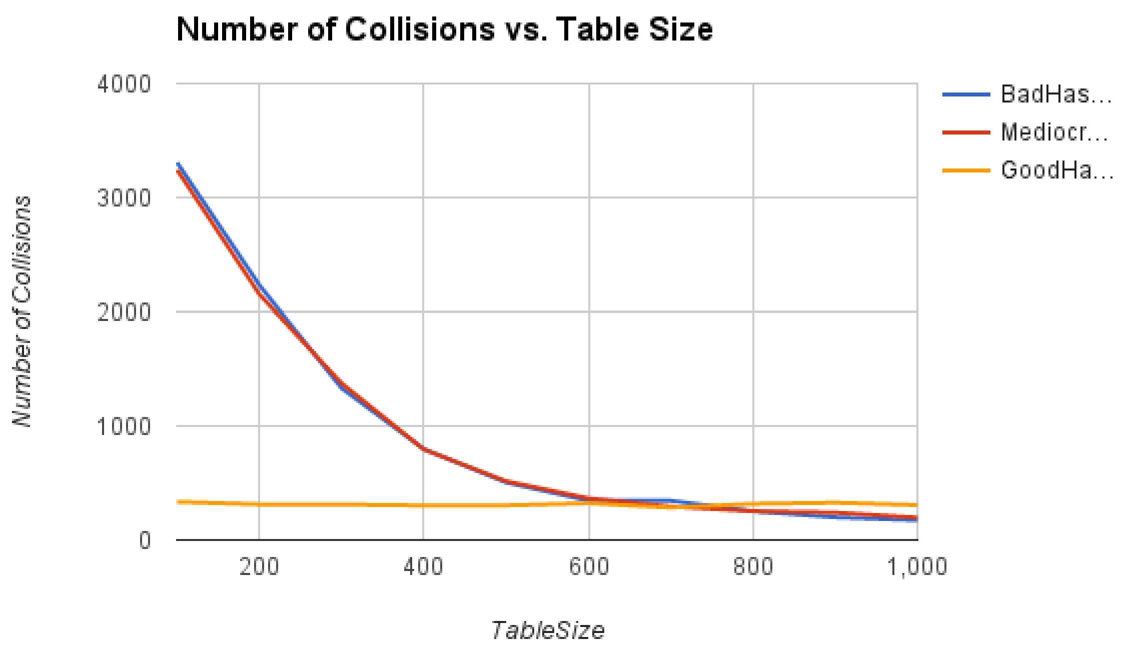
3. Give and explain the hashing function you used for MediocreHashFunctor. Be sure to discuss why you expected it to perform moderately (i.e., result in some collisions).

My MediocreHashFunctor returns the sum of the values of the first character and the length of the String. Thus, this narrows down words that are of the same length, and start with the same letter. Although this accounts for two different criteria for a String, the factor of the value of the first character can only differ from 25 other characters. Thus, this narrows the number of collisions, however collisions would still be expected, as these two factors do not vary much from word to word.

4. Give and explain the hashing function you used for GoodHashFunctor. Be sure to discuss why you expected it to perform well (i.e., result in few or no collisions).

My GoodHashFunctor takes the sum of all the characters in a String, and multiplies it by the length of the String. Because this accounts for all the different collections of letters in words, and lengths of words, this is much more narrow than the previous two hash functions. I chose to multiply the sum by the length of the String because this will give a more unique number than just by adding them together, resulting in fewer collisions (i.e. 18+5 is much different than 18*5). A predicted problem with this, is if the String passed in the hash function is a very long String, or contains many characters. This would be a problem as it would take much longer to calculate the hash value.

5. Design and conduct an experiment to assess the quality and efficiency of each of your three hash functions.



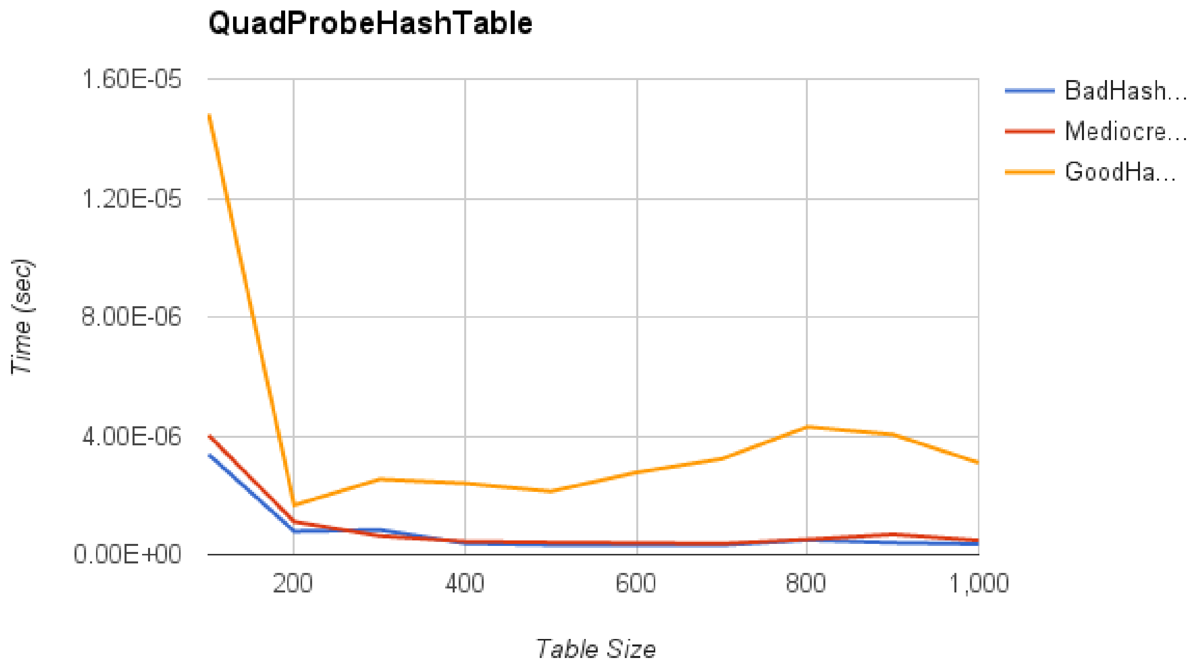
This experiment counts the number of collisions that occur for each HashFunctor, for a QuadProbeHashTable. The experiment is run three times, each time using a different HashFunctor. Within the QuadProbeHashTable class, an integer variable was added, “collisions”, which was incremented each time a String was added to the HashTable and a collision occurred. In the timing experiment, for HashTable sizes ranging from 100 to 1000 (incremented by 100), 1000 Strings were added to the Table, and the average number of collisions is the data used for the above plot. The Strings used in this experiment are created in a random String generator method, ranging in length from 1 to the size of table times 2.

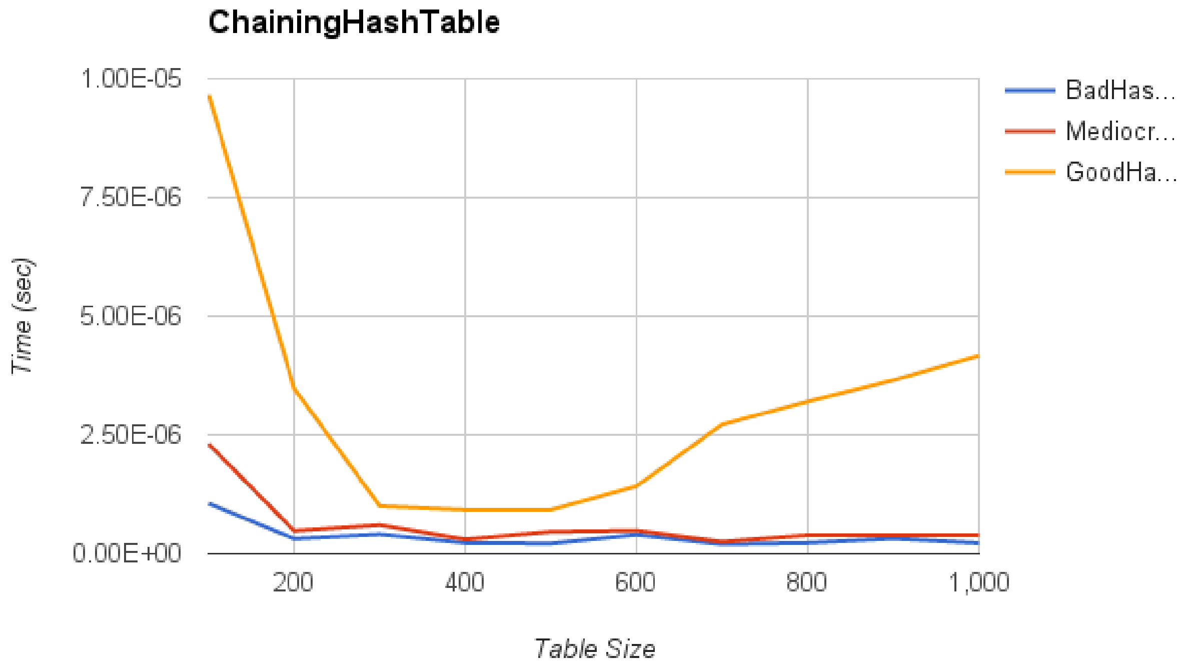
In looking at the data collected, it can be concluded that the GoodHashFunctor outperforms the other HashFuntors for small table sizes, however seems to be outperformed by both for larger table sizes. BadHashFunctor and MediocreHashFunctor had very similar collision totals, which can be accounted for by that fact that their hash functions are pretty similar. The only thing that differs is the value of the first character being added to the length in the mediocre hash, which wouldn't change the value very much since there are only 26 values that could possibly be added to the value. The reason these two HashFunctors would outperform the good hash for large table size is because of how the words were being generated. Because the table size is much larger, it is less likely to end up with very similar length words for the same amount of iterations; and the table size is much larger, they end up being more evenly distributed. For the GoodHashFunctor, the size of the table will not affect the hash values, and thus would not affect the number of collisions. The data for the timing performance of these functors on a QuadProbeTable is in the following question.

6. Design and conduct an experiment to assess the quality and efficiency of each of your two hash tables.

This experiment add method for each HashFunctor, for a QuadProbeHashTable and a ChainingHashTable. The experiment is run three times, each time using a different HashFunctor. In the timing experiment, for HashTable sizes ranging from 100 to 1000 (incremented by 100), 1000 Strings were added to the Table, and the average time for the add method is the data collected in the following graphs. The Strings used in this experiment are created in a random String generator method, ranging in length from 1 to the size of table times 2.

In the collected data, BadHashFunctor and MediocreHashFunctor have the most consistent behavior, and most efficient times, while ChainingHashTable seems to be more efficient than QuadProbeHashTable. As expected for a hash table, the add method seems to have Big-Oh complexity O(c). For smaller table sizes, there appears to be a slight peak for the bad and mediocre functors, and a major peak for the good functor. This could be due to having more collisions in smaller table sizes for the bad/mediocre functors. For the good functor, this could be the time it takes to calculate the hash for the Strings inputted.


QuadProbeHashTable
1.60E-05
1.20E-05
8.00E-06
4.00E-06
0.00E+00
Time (sec)
200
400
600
800
1,000
Table Size
BadHash...
Mediocre...
GoodHa...


ChainingHashTable
1.00E-05
7.50E-06
5.00E-06
2.50E-06
0.00E+00
Time (sec)
200
400
600
800
1,000
Table Size
BadHas...
Mediocr...
GoodHa...

7. What is the cost of each of your three hash functions (in Big-O notation)?

For BadHashFunctor, the Big-Oh complexity is O(c). The only thing this functor calculates is the length of the String, which can be done in constant time. MediocreHashFunctor has Big-Oh complexity O(c) as well, as it only looks at the first letter of the String, and the length, which are two constant operations. GoodHashFunctor has Big-Oh complexity O(N), because it has to sum the value of each character in the String, and then multiply it by the length. Looking at, and evaluating each individual character in a String would be linear time. Calculating the length and multiplying the sum by the length are both constant time actions, which gives the overall complexity O(N).

8. How does the load factor Î» affect the performance of your hash tables?

For the QuadProbeTable, the load factor determines when to rehash the table. Thus, once the table is more than half full, it is rehashed into a table with the capacity the next prime number of twice the original size. Doing this ensures that an empty spot can be found for any String being inserted into the table, which means that the performance of the add and contains methods is at most linear time. Although load factor was not added into ChainingHashTable, it would have ensured that the LinkedLists backing the table did not get too long, ensuring O(c) complexity. By having the load factor determine when to rehash a hash table, running time complexity is ensured to never be more than O(N).

9. Describe how you would implement a remove method for your hash tables.

I would change the backing array to hold nodes instead of Strings. These nodes would contain the data of the String, but also a Boolean value. If an item is removed, if it is contained in the hash table, this Boolean value would be set to true (but false when initially added to table).

10. As specified, your hash table must hold String items. Is it possible to make your implementation generic (i.e., to work for items of AnyType)? If so, what changes would you make?

It would be possible, but there would need to be more checks in the code. As stated in question 9, the array the holds the table could be made of nodes, which could be constructed to hold a generic value, not just a String. In addition, there would need to be a hash function provided for any type of data passed into the hash table (e.g. if we constructed a hash table for LibraryBooks, we would need a hash function for LibraryBooks).

11. How many hours did you spend on this assignment?

10