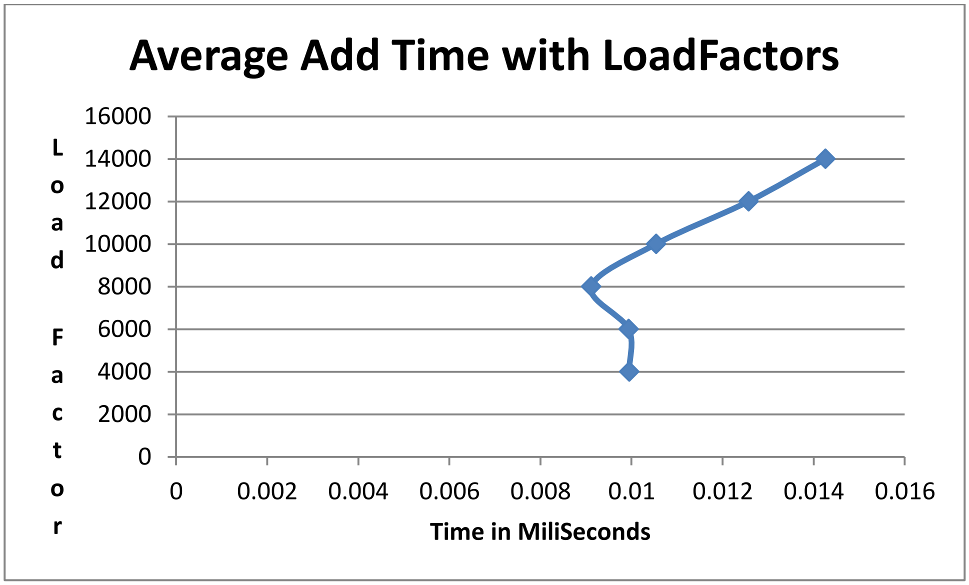
Christian Hansen
U0621884

Analysis Document

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

For quadratic probing, the loading factor λ means the amount of items in the array divided by the capacity of the array. Simply put it is the amount of the array that is full. Keeping the array half full (.5 or 1/2) or less will allow the performance of the hashtable to maintain the O(c) of the operations.

For separate chaining, the loading factor λ means the average length of each linked list. Using a timing experiment, I found that re hashing, at a load factor of above 4, performed better than others. For the experiment I added 2000 random words to a list, taking the average time of adding a word at any time. I performed this experiment for different load factor thresholds. Somewhere between a factor of 2-4 all performed very similarly. The data suggests that a threshold of 2 is the best.



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).

The BadHashFunctor hashes the string by simply returning the length of the string. This will perform poorly because any string of the same length will have the same hash, this will not distribute the strings very well and it will also not work at all well for arrays of large sizes. This will result in many collisions and if it is a large array then all the items will be placed on one side. Eventually Big O performance of this array as the size gets large will not be constant but rather linear O(N).

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).

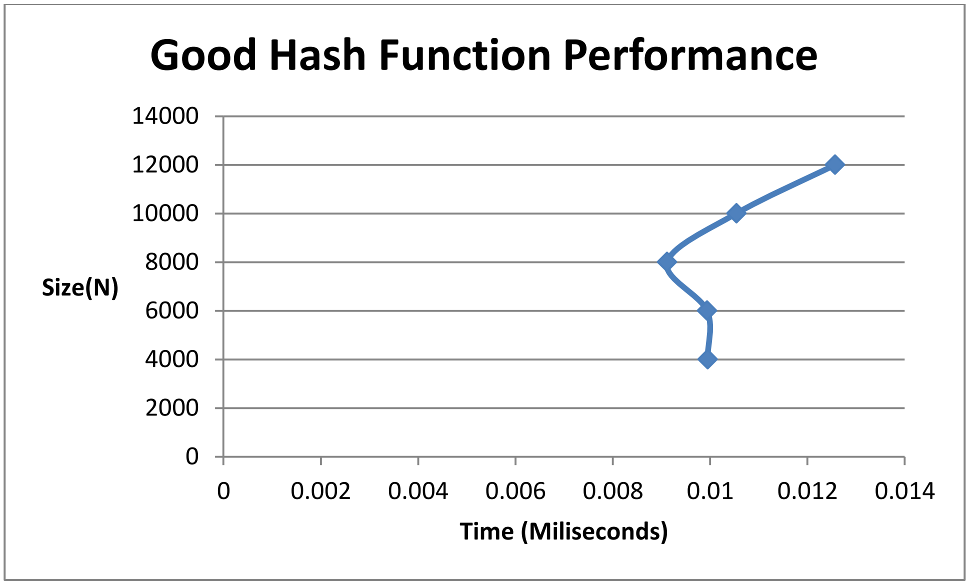
The MediocreHashFunctor hashes the string by add each char together and before adding the char it is multiplied by the index where it is located. Therefore this will create more unique strings. If two strings that have the same characters in different orders, this hash will create unique hashes for each, helping to avoid collisions. This hash will not create very large hash numbers though, which will cause problems in large arrays because the items will be concentrated on the left.

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).

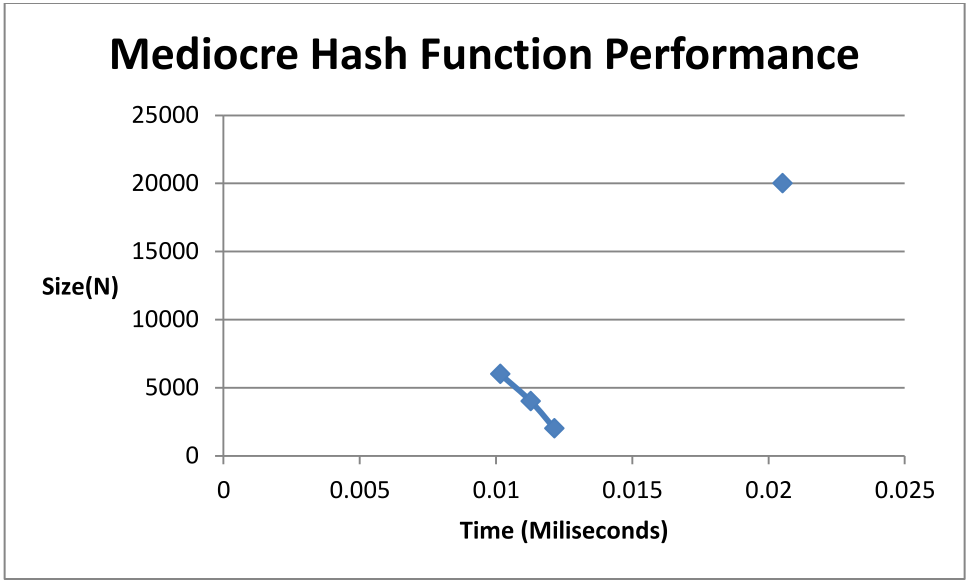
The GoodHashFunctor hashes the string by adding each char to a total and then returning the absolute value of that total. Each time before the char is added, it is multiplied by 17. Because 17 is a prime number and large it will help create large hashes. In case it exceeds the highest possible integer amount and overflowing occurs, the absolute value is taken of the number to be returned. With the total being multiplied by 17 before each char is added to the total, two strings containing the same characters will have unique numbers. High numbers will also be created and this will cause the hash function to work well with large arrays. I expect it to result in less collisions then the other hashes

5. Design and conduct an experiment to assess the quality and efficiency of each of your three hash functions. Carefully describe your experiment, so that anyone reading this document could replicate your results. Plot the results of your experiment. Since the organization of your plot(s) is not specified here, the labels and titles of your plot(s), as well as, your interpretation of the plots is critical.

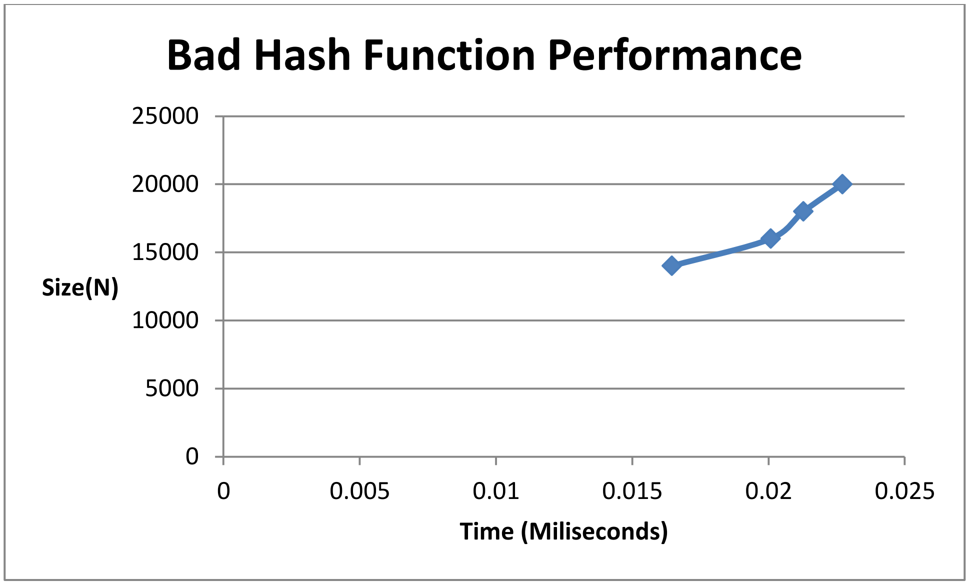
I made three different graphs that show the average add time adding random strings starting from 0 and going to the size amount (2000, 4000, etc). This clearly shows that the good hash function performs the best. The Mediocre performs second best and the bad performs the worst. When the array needed to be expanded, that time was taken into account. The capacity began at 13. Unfortunately something happened to the charts below, I did not have time to fix it (I understand whatever grade you give me for it.)



Good Hash Function Performance



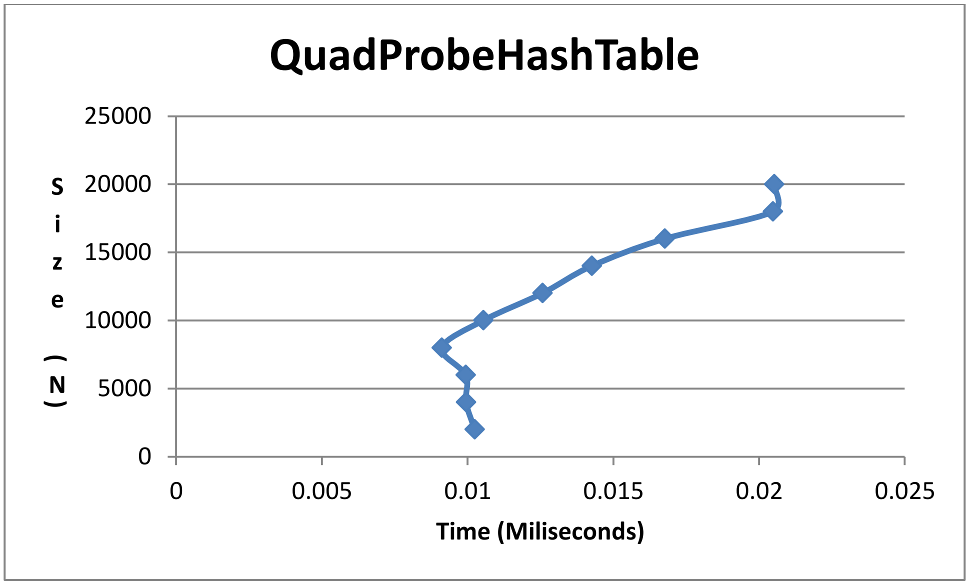
Mediocre Hash Function Performance

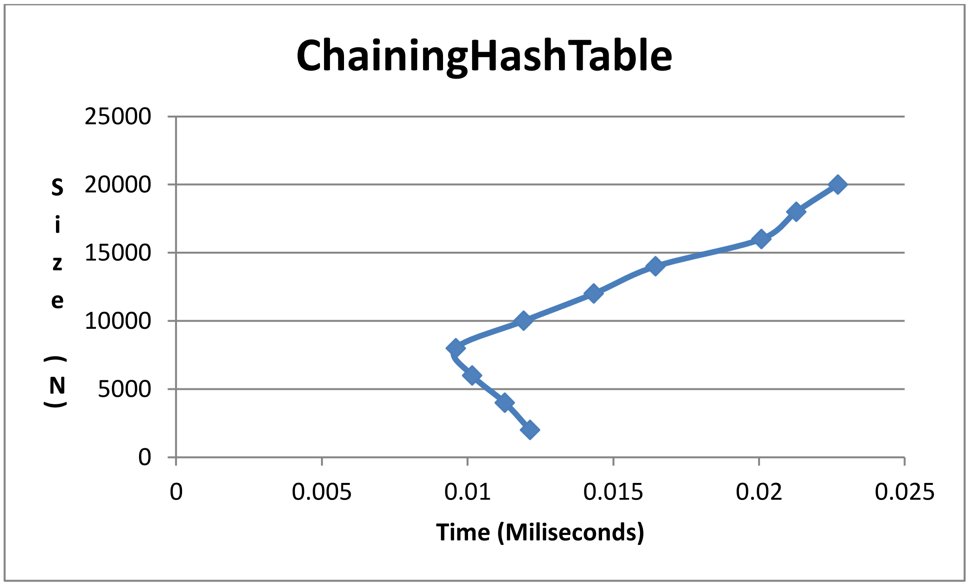


6. Design and conduct an experiment to assess the quality and efficiency of each of your two hash tables. Carefully describe your experiment, so that anyone reading this document could replicate your results. Plot the results of your experiment. Since the organization of your plot(s) is not specified here, the labels and titles of your plot(s), as well as, your interpretation of the plots is critical.

A recommendation for this experiment is to create two plots: one that shows the number of collisions incurred by each hash table using the hash function in GoodHashFunctor, and one that shows the actual running time required by each hash table using the hash function in GoodHashFunctor.

I added items to the tables going to 2000 and then calculated the average add time of each item. I did the same for the rest of the sizes increasing by 2000 and going to 20000. The QuadProbeHashTable performs better.





7. What is the cost of each of your three hash functions (in Big-O notation)? Note that the problem size (N) for your hash functions is the length of the String, and has nothing to do with the hash table itself. Did each of your hash functions perform as you expected (i.e., do they result in the expected number of collisions)? (Be sure to explain how you made these determinations.)

The bad hash function peforms in constant time O(c). This is because it just uses the length of the string, so this operation should perform in the same amount of time regardless.

The mediocre hash function performs in linear time. This is because it runs a for loop and iterates through the length of the string. This would be O(N).

The good hash function performs in linear time. This is because it runs a for loop and iterates through the length of the string. This would be O(N).

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

The lower the load factor the less possibility of a collision. Choosing at what load factor to expand and rehash is very critical to performance. Re-hashing at the right load factor, using quadratic probing, and using a good hash function are all critical to keeping the hash table performance to Big O(c)

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

For the QuadProbeHashTable, I would keep another array of booleans each one would correspond with the spot on the HashTable. If the item is found in the HashTable, but is false on the boolean array then it has been removed. We can't remove the item from the array since the items locations depend on one another. If the array is expanded and rehashed, then items that are no longer counted as in the array are not deposited into the array.

For the ChainingHashTable, items can be removed directly from the HashTable. The items are placed into the different LinkedLists, they don't depend on one another for location. An item can be removed from the LinkedList and it won't affect others because the item will just be searched for in the LinkedList that they correspond to.

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 is possible to make the HashSet generic. The type of the backing array would be of the type that the user passes in. The user would need to pass in a functor that works with their data type as well. Since you cannot instantiate a basic array of generic type, the array would be of type object.

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

I spent 12 hours on this assignment.