When you are satisfied that your program is correct, write a brief analysis document. The analysis document is 30% of your Assignment 10 grade. Ensure that your analysis document addresses the following.

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?

The load factor lambda is the number that defines what fraction of the array is full. For a quadratic probe hash table, the hashing works best when lambda is approximately 0.5, and for separate chaining, it is optimal at 0.75.

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).

For BadHashFunctor I elected to go for the worst possible hash, which is assigning each hash to 1, so that aside from the first element, each element has a collision. Even with quadratic probing each element would go through the same hash and rehash, so the collisions would be 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).

For MediocreHashFunctor I chose to do a hash based off of the numbers associated with characters, and each string is just a sum of its characters. This creates a bigger variety of hashes, but still has quite a few collisions, especially because there are several combinations of characters that amount to the same hash.

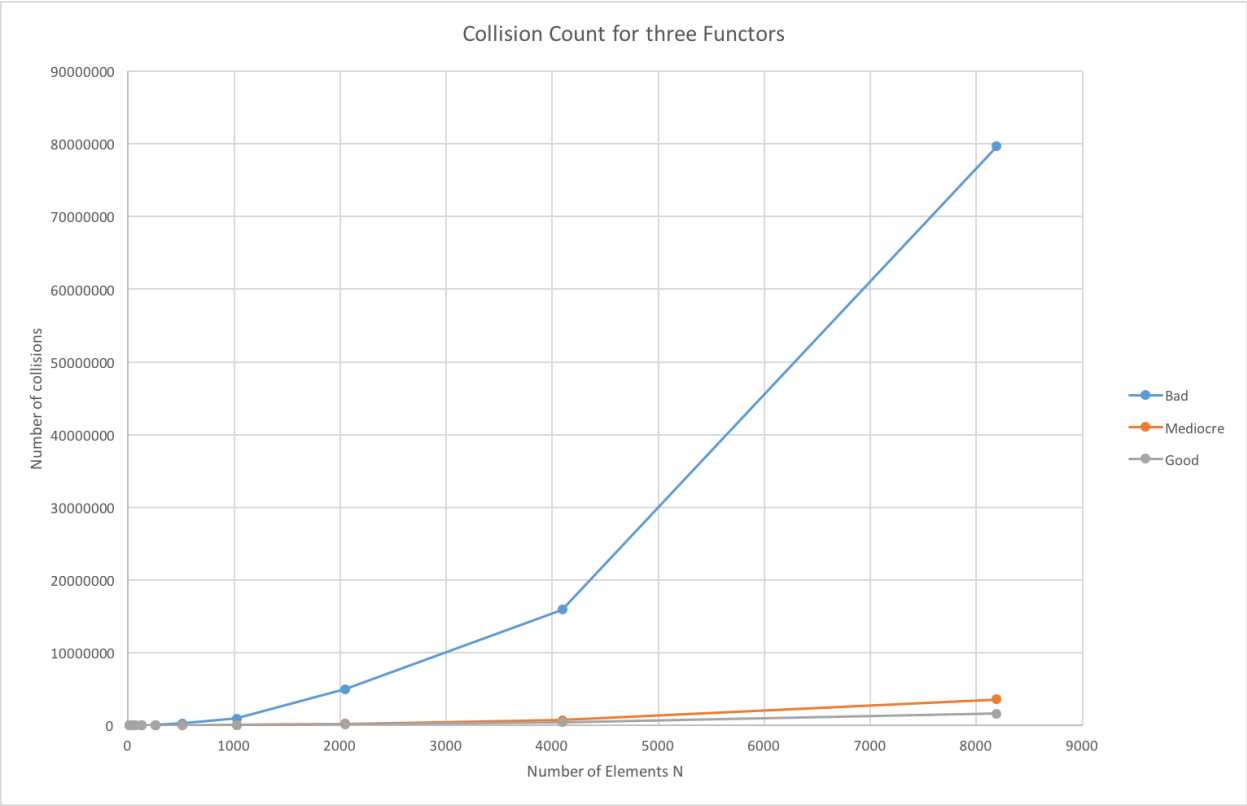
4. Give and explain the hashing function you used for GoodHashFunctor. Be sure to discuss why you expected it to perform well (i.e., result in few or no collisions).

For GoodHashFunctor, I chose to take the sum of each character as before, but multiply each successive character by a prime number (in my case, 23), and take the mod of 100 before adding to the total sum. The prime number makes it so that there are less hashes with the same quotients because they're not divisible by several numbers, and randomizes them more. It should create more unique 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.

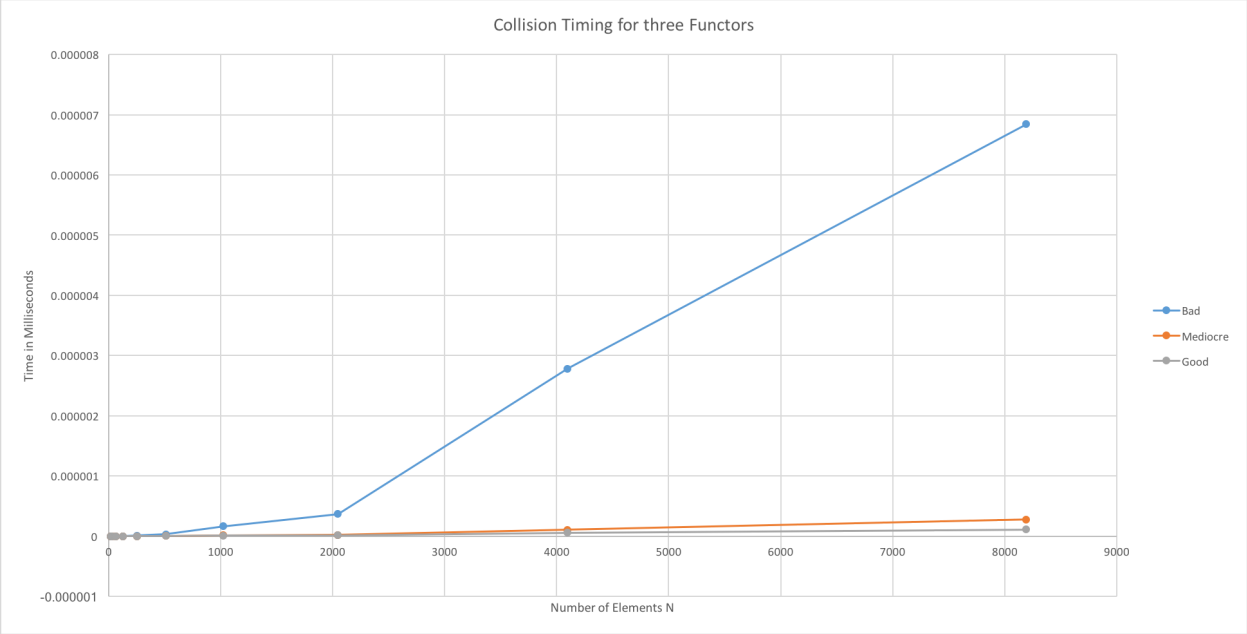
A recommendation for this experiment is to create two plots: one that shows the number of collisions incurred by each hash function for a variety of hash table sizes, and one that shows the actual running time required by each hash function for a variety of hash table sizes. You may use either type of table for this experiment.

For this problem, the first experiment was along the guidelines presented. I created three quadratic probe hash tables, one with a bad hash functor, one with a mediocre one, and one with the good one. I created a method for counting the number of collisions when adding elements 1 to N to the hash table. N was anywhere from 2^4 to 2^{13}, I ran tests on all 10 of those ranges. Here are the results:



As expected, the BadHashFunctor performed terribly, and the other two were pretty steady, with the good hashfunctor outperforming the mediocre hash functor.

For the timing experiment it was an identical setup, but now I used a method that timed the addition of the same amounts of elements rather than counting collisions. The results are as follows:

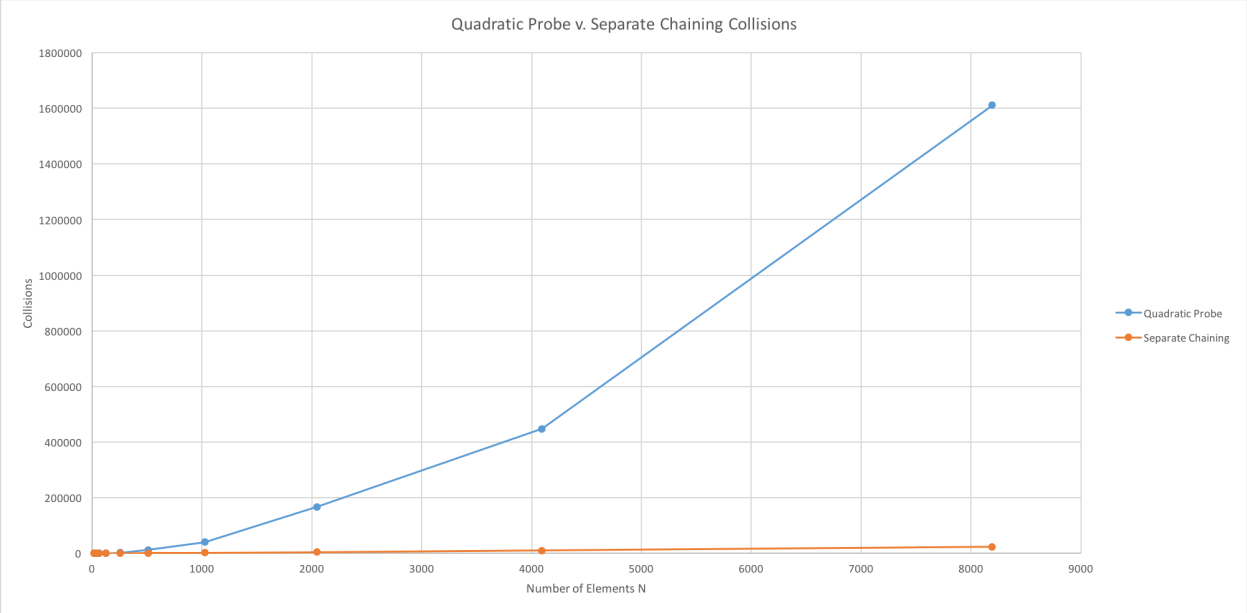


The BadHashFunctor performed on O(N) behavior as expected, and the other two performed on O(c) behavior as desired, with the mediocre hash functor performing pretty close to the good one, but still not quite good enough.

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.

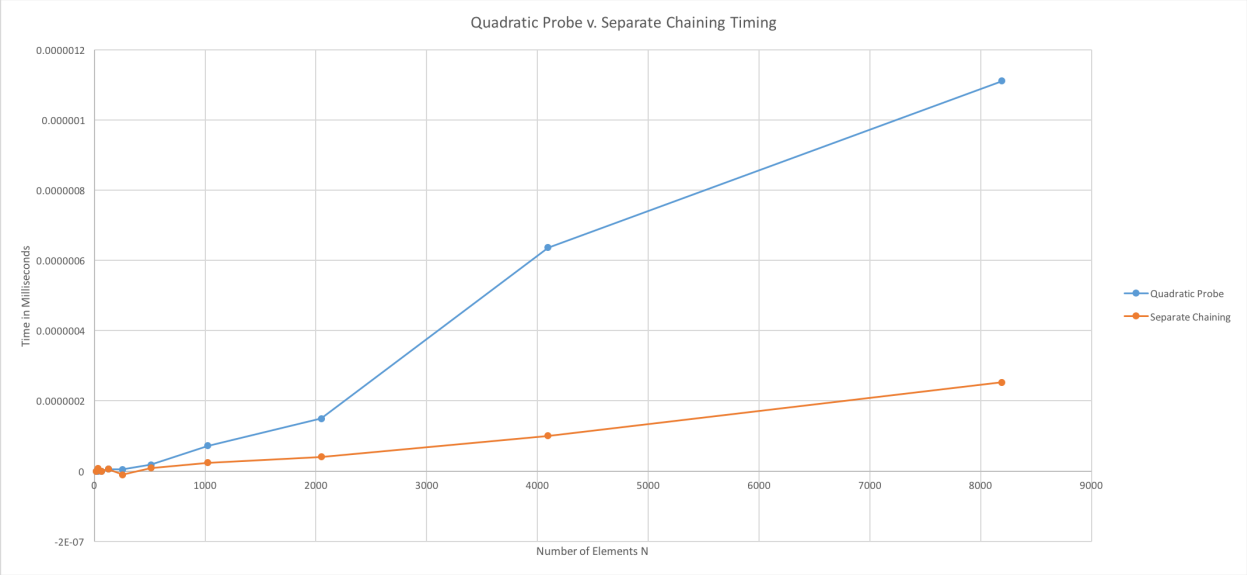
A recommendation for this experiment is to create two plots: one that shows the number of collisions incurred by each hash table using the hash function in GoodHashFunctor, and one that shows the actual running time required by each hash table using the hash function in GoodHashFunctor.

As suggested, in this experiment, I used the collision count method described in question 5 and made a quadratic probe hash table with the good hash functor, and then made a similar method that would count collisions but for the separate chaining hash table. The results of these tests are as follows:



The SeparateChaining Hash table worked a lot better than the quadratic probing one

For timing, the same method was used as number five, but again with one method timing a quadratic probe hash table and one timing a separate hash chain, using the same lists and list sizes. Here are the results.



The separate chaining outperformed quadratic probing as expected.

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 functor performed in O(N), and the other two performed in what seemed to be close to O(c) but wasn't quite constant. In relation to each other they performed as expected, but the good hash function didn't work as well as optimally it should have, judging by the graphs not being horizontal, but tilted upward.

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

The load factor determines at what point the elements need to be rehashed. Rehashing too early takes up a lot of time in refilling the table. Rehashing too late risks creating a lot more collisions than are necessary.

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

For separate chaining hash tables you would delete references to the object in a linked list and reassign heads and tails as necessary just like a regular linked list. For quadratic probing you'd set the item at the location to null or mark it as deleted. You would find it in the same method as you did for add.

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 a hash table generic, by using comparators or comparable in the hash function as needed.

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

5-7 hours