Chase Stephens

Assignment 10 analysis

**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 for a quadratic probing hash set represents the portion of the set that contains elements. The load factor for the quadratic probing should be less than 0.5 for best performance. The load factor for the separate chaining hash set represents the average length of each linked list. The load factor should not affect the performance much until it becomes large but in general, it is best to keep it below 1.

**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 just returns the string length cubed. This is bad because any two string of the same length will end up with the same hash value, which will be a common case and will therefor result in many collisions.

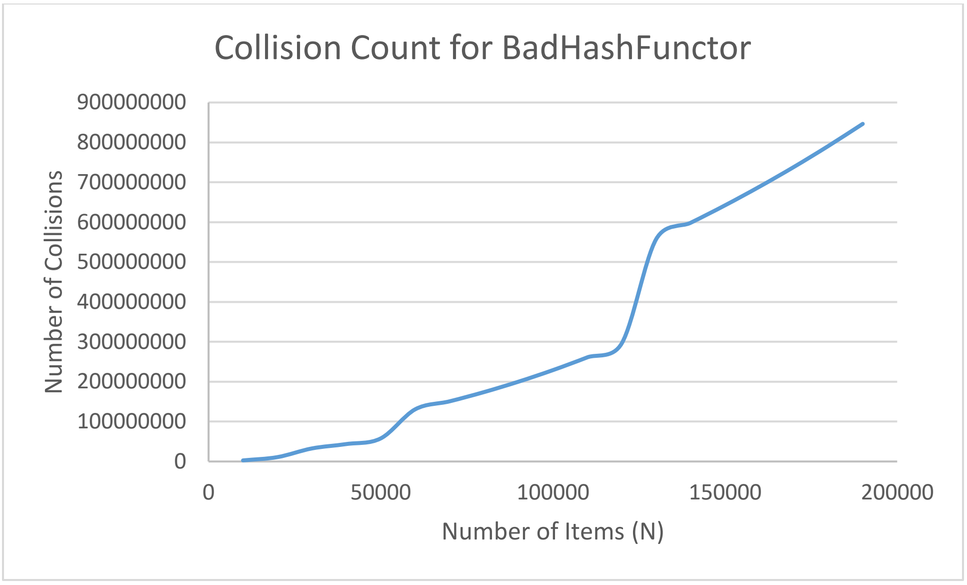
**3. Give and explain the hashing function you used for MediocreHashFunctor. Be sure to discuss why you expected it to perform moderately (i.e., result in some collisions).**

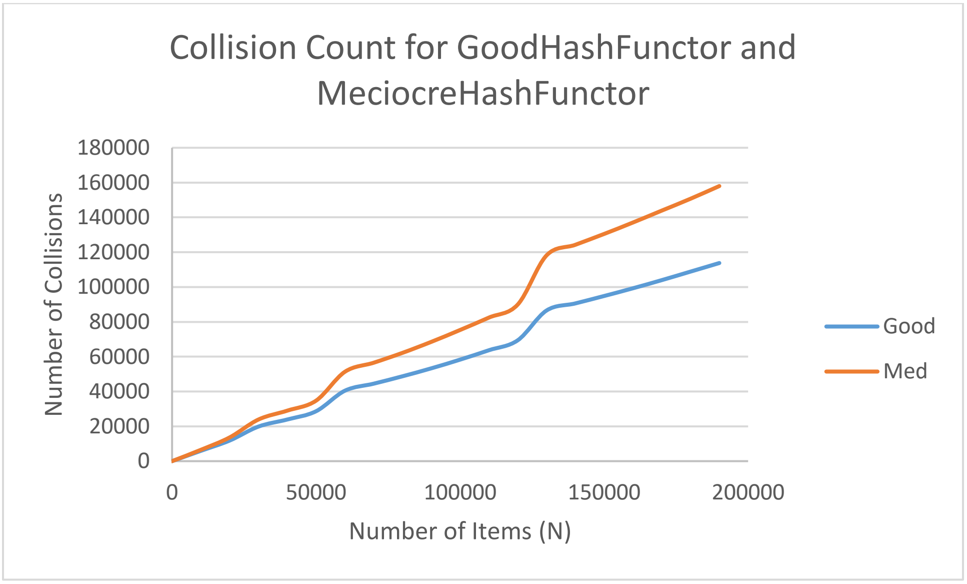
The MediocreHashFunctor is better than the BadHashFunctor because it is not just based off of the length, but also what is in the string. The function sums up the number representation of each character in the String and multiplies each one by ten. The reason this will not be good, however, is that even numbers are more likely to create the same multiples.

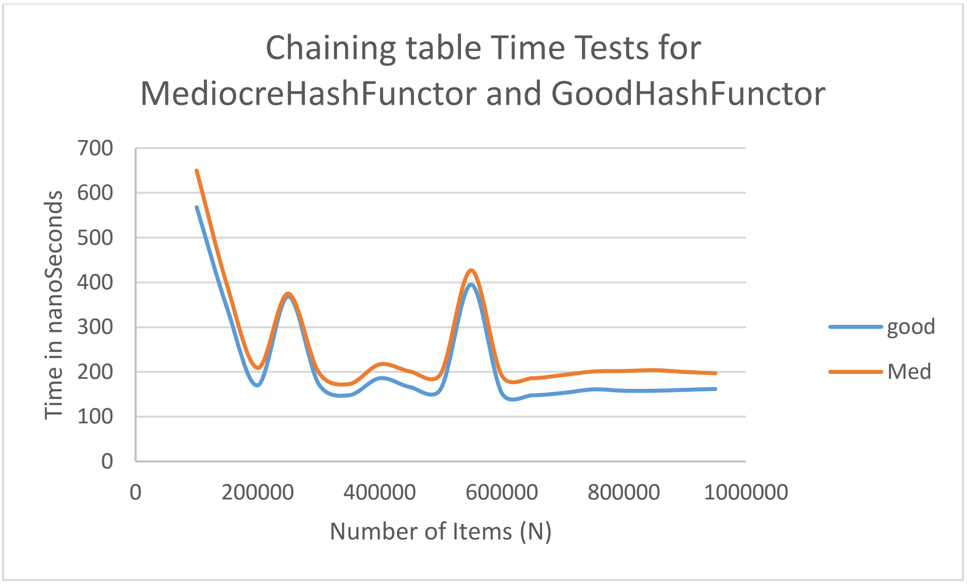
**4. Give and explain the hashing function you used for GoodHashFunctor. Be sure to discuss why you expected it to perform well (i.e., result in few or no collisions).**

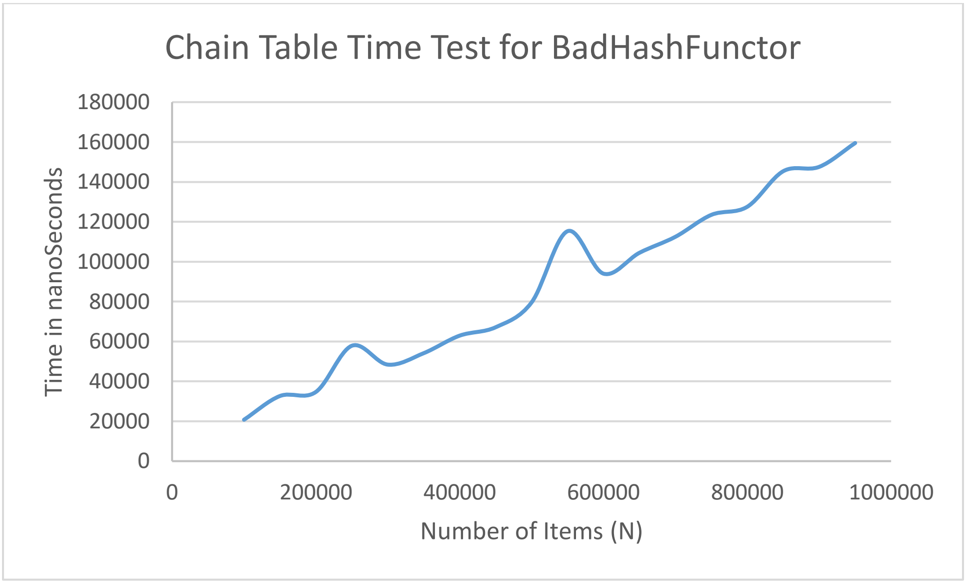
The GoodHashFunctor does that same thing that the MediocreHashFunctor does but with prime number. The prime numbers provide a greater chance of creating more unique multiples. This is because there are less common factors shared between each out, since the prime number cannot be broken down into smaller parts.

**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.**


Collision Count for BadHashFunctor
Number of Collisions
900000000
800000000
700000000
600000000
500000000
400000000
300000000
200000000
100000000
0
0
50000
100000
150000
200000
Number of Items (N)


Collision Count for GoodHashFunctor and MeciocreHashFunctor
Number of Collisions
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
0
50000
100000
150000
200000
Number of Items (N)
Good
Med



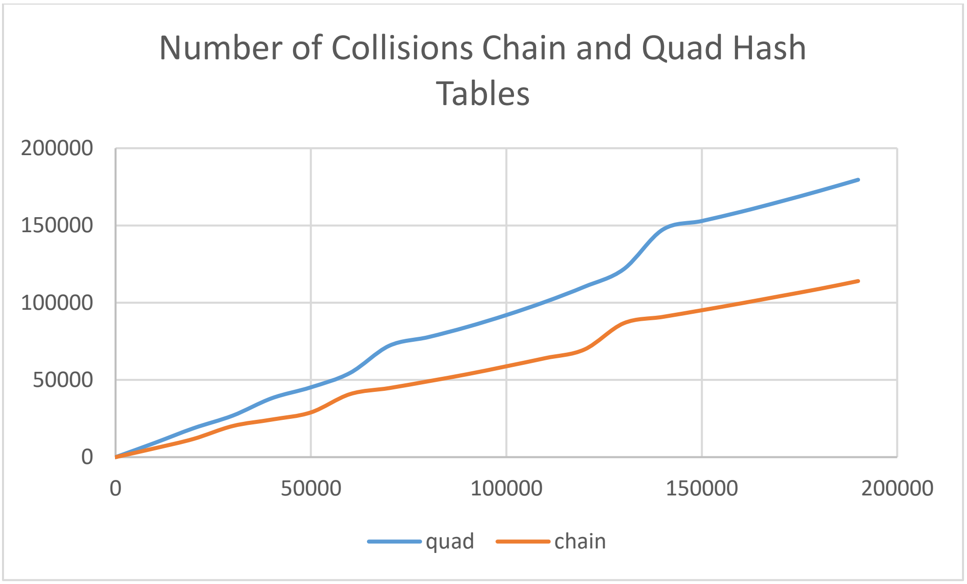


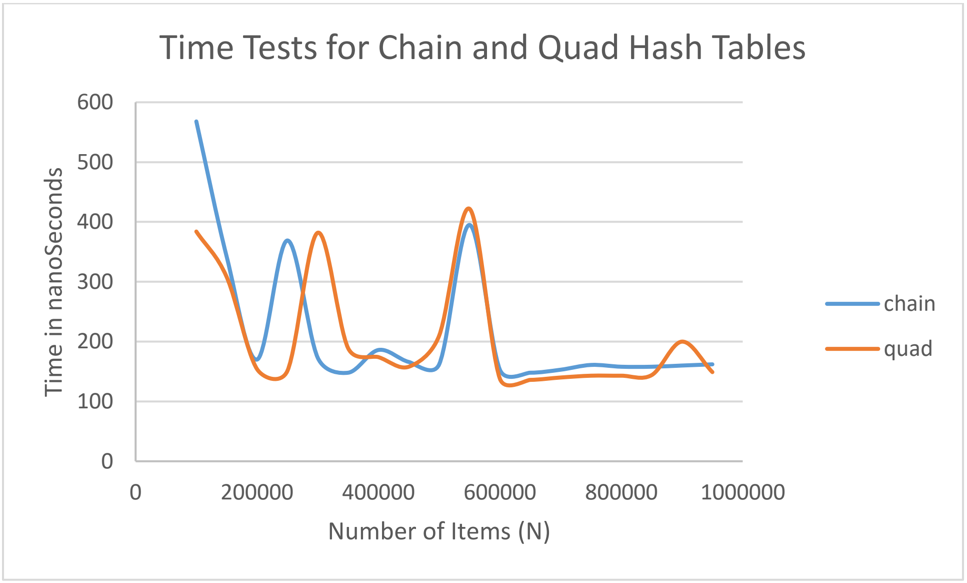
For comparing each hash function, the ChainingHashTable was used. The collision test involved adding 2000 items to the table and printing out the total collision count after each 10000 items were added. The items consisted of random strings with a length of up to 15. This was repeated for each hash function. For the time tests, many items were added and the time required to add each item was averaged. The first size tested was 150,000 and was incremented by 50,000 up to 950,000. This was also tested for the good, bad, and mediocre hash function.

For each test the BadHashFunction had significantly worse results and was therefore placed in its own graph. The GoodHashFunction was slightly better that the MeciocreHashFunction on time, with an even larger difference in the number of collisions.

**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.**





For these tests, the GoodHashFunctor was used in all cases. The number of collisions were compared for the Quad and Chain hash tables. The same method for both runtime and measuring collisions were used for these tests. The ChainingHashTable resulted in much less collisions. There were some spikes on the graph likely caused from the tables rehashing. The Chaining table is set to rehash at lambda = 1 while the quad table is set to rehash at lambda = 0.5. The time is much harder to tell since the tables are occasional rehashing on some of the add calls which significantly increase the time for that add step. It does appear as though the ChainingHashTable did perform a little bit better on average.

**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.)**

For the BadHashFunctor, the complexity should just be O(C). This is because only the length of the string is used for the calculation. For both the MediocreHashFunctor and the GoodHashFunctor, the complexity should be O(N). This is because each character in the string is analyzed. Both methods call a loop that iterates form 0 to the string length.

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

A larger load factor increases the run time and number of collisions.

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

The object needed to be removed would be hashed. Then the index would be checked, if the item in the location did not match, then the next one would be checked, and so on in the same order that the collisions are being dealt with. If a null space is reached before the item is located, false would be returned. Otherwise, when the item was found, it would be removed. Then the next item would be looked at, if it had the same hash it would be moved back to the spot of the removed item. This would continue until either a null space or an item with a different hash is found. The method would then return true.

**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?**

Yes. If we were to use comparable items, the just changing the method arguments to type and making the array Generic. It would not require much to do this. We could also make a constructor to pass in a comparator if the items are to be compared in a different way. This would make the table work for all items.

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

10