Assignment 7 Analysis
Samantha Pope

1. 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 returns the length of the string that is passed. I expect this to perform badly because many different words have the same length. This will cause a lot of collisions if the strings are the same size.

2. 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 ascii values of each character in the string. I think that this will perform better than the BadHashFunctor, since it will produce more variable results. It could still have collisions if strings add up to the same sum of ascii values. (ex: abc and cad would produce a collision).

3. 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).

I used a function that iterates over each character in a string. The hash was originally assigned to 7. Then it updates the hash by multiplying the hash number by 31. 31 is chosen because it is an odd prime number. Then we add the ASCII value of the current character. I found this formula from Ben's links on github. I expect this to perform well because we multiply by an odd, prime number. I also expect this to perform well because I expect different strings to have different ascii values at each spot. It is still possible for collisions to happen because of the pigeonhole principle.

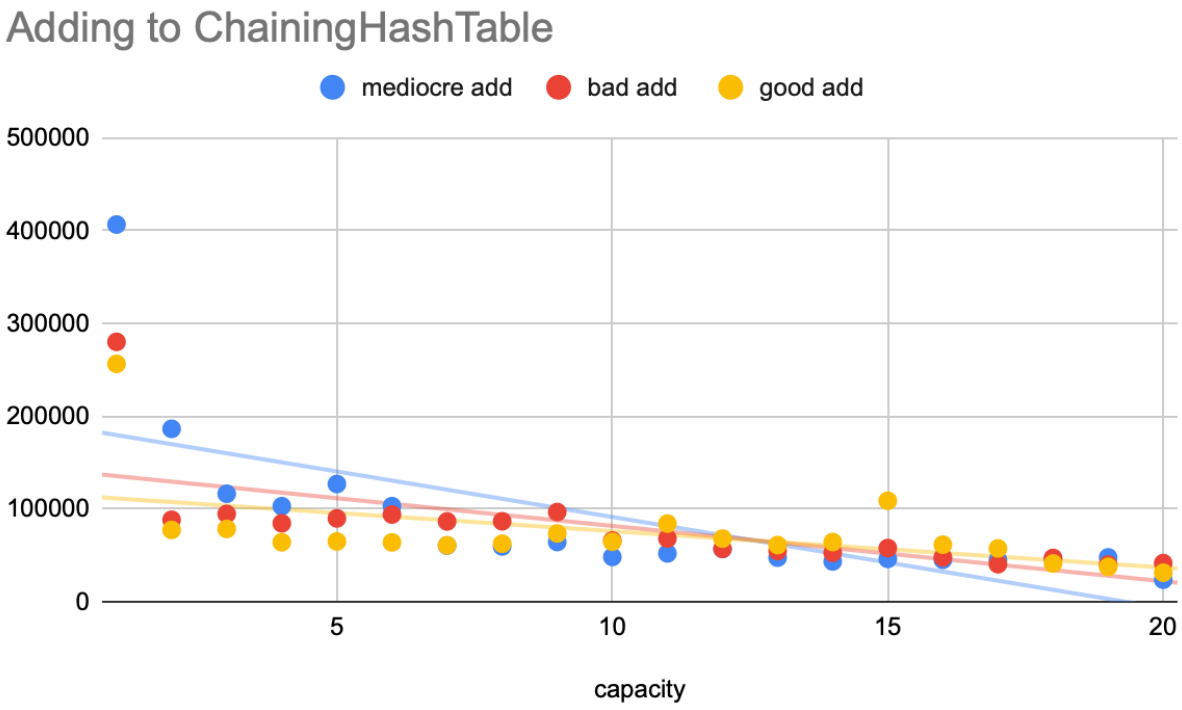
4. Design and conduct an experiment to assess the quality and efficiency of each of your three hash functions. Briefly explain the design of your experiment. Plot the results of your experiment. Since the organization of your plot(s) is not specified here, the labels and titles of your plot(s), as well as, your interpretation of the plots is important.

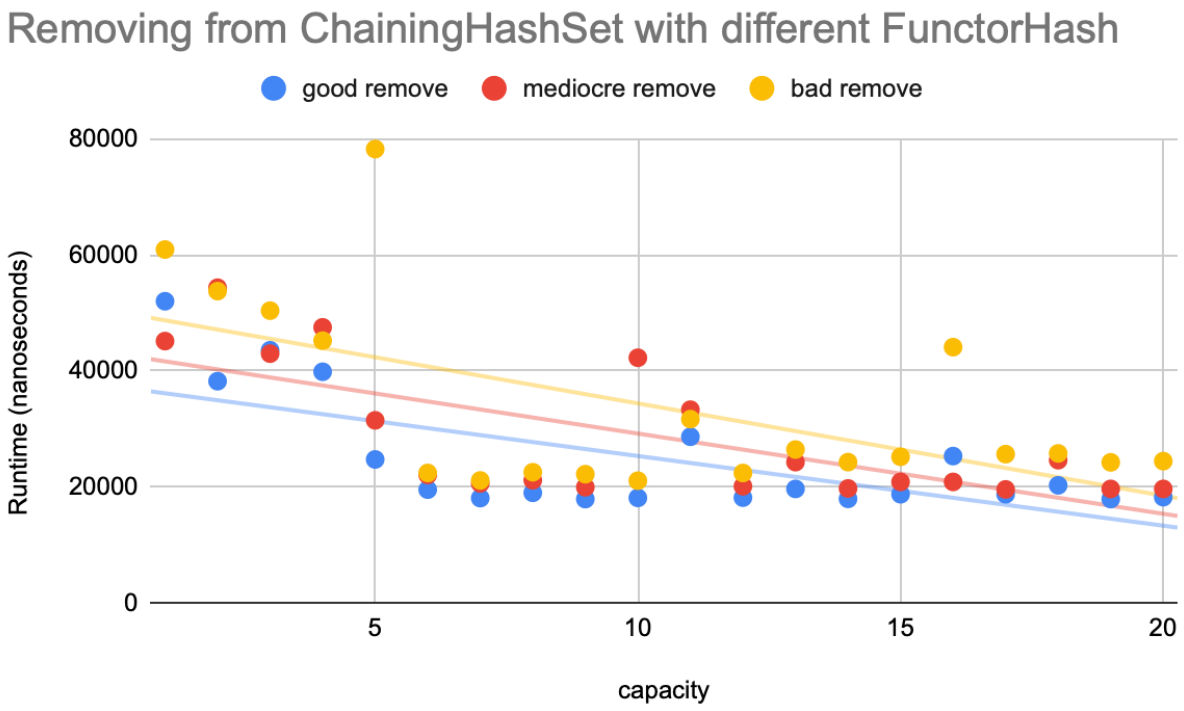
<u>RUNTIME GRAPHS:</u>

I tested the three hashFunctors runtime by creating an array of strings from sizes of 2^10 to 2^20. I did this by appending a random double to a string, using a seeded Random object. I had three separate ChainingHash objects that I passed each different

hashFunctor to. I then plotted the amount of time it took to add the array to my ChainingHash object and plotted the runtime against the arraySize in the first graph. Then I tested the amount of time it took to both add and remove the array of strings from the ChainingHash objects and plotted it in the second graph.

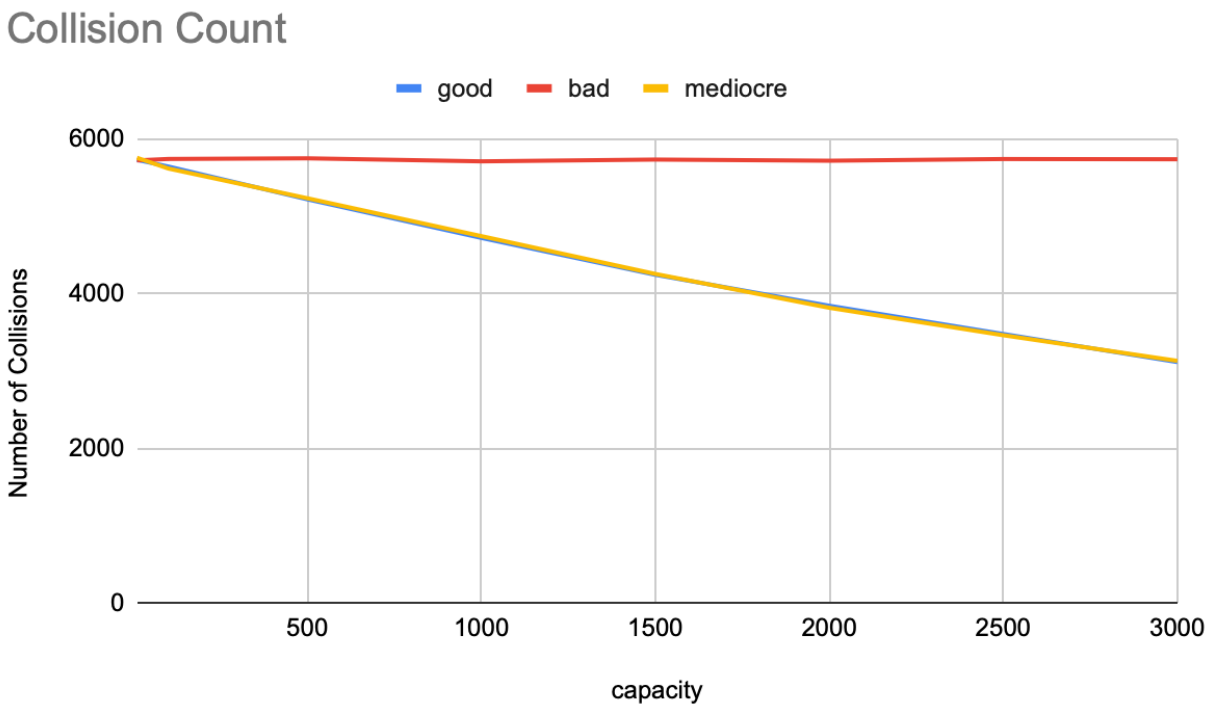
CHAINING HASH TABLE TESTS:

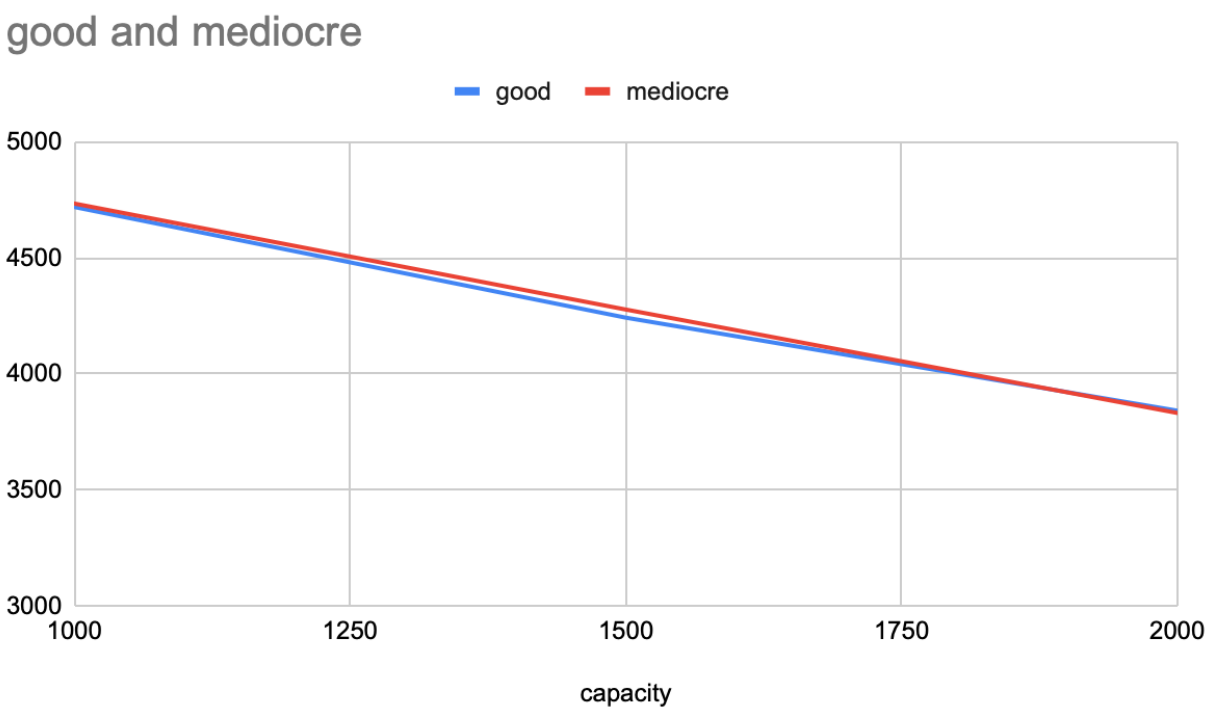




COLLISION GRAPHS:

I made a method that added up the amount of collisions in a ChainingHashTable object. It looked at the buckets with sizes larger than one and then added up those sizes to return an overall collision count. I then passed multiple capacities to the ChainingHashTable objects (from 50 to 3000) and plotted the number of collisions for each of the three hashFunctor ChainingHashTable objects. For each capacity, the program initializes a new instance of ChainingHashTable and populates it with a fixed set of 6000 randomly generated strings. Each string is constructed by appending a character (converted from a randomly generated integer) to the prefix "String". The random nature of the data aims to mimic a typical usage scenario and provide a realistic distribution of hash values. After populating the hash table, the program calculates and reports the number of collision occurrences—a scenario where more than one string is hashed to the same bucket in the table. By iterating through various capacities, the experiment is designed to reveal how the capacity of the hash table impacts its tendency to experience collisions, to test effectiveness of the hashfunctor.





Number of Collisions:

capacity	good	mediocre

10	5729	5721
100	5643	5632
500	5218	5231
1000	4719	4734
1500	4242	4277
2000	3841	3831
2500	3476	3490
3000	3113	3143

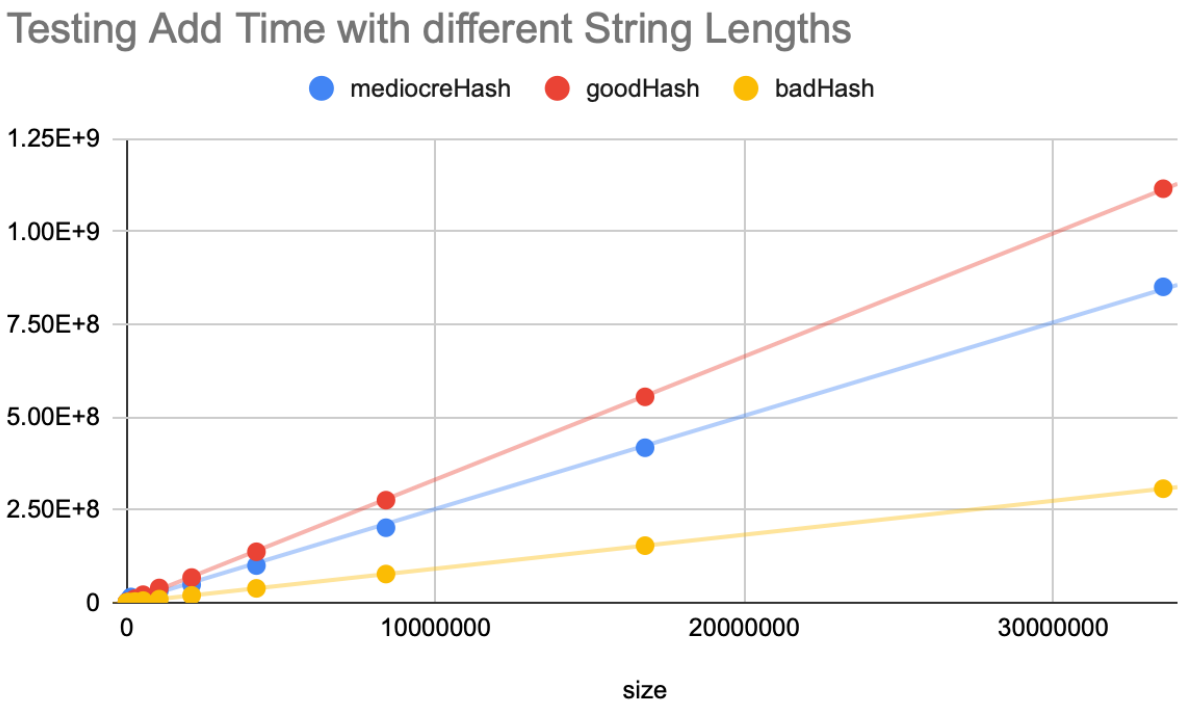
My mediocre and good hash functions had almost exactly the same amount of collisions, however the good hash function had a smaller amount of collisions as shown by the zoomed in graph and table.

5. 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)?

My goodHashFunction has a runtime of O(N), it iterates through every item to get the ascii value of each character. My mediocreHashFunction has a runtime of O(N), it iterates through each item to access the ascii value of each character . My badHashfunction has a runtime of O(1) because it just returns the length of the string.

I expected my goodHashFunction to perform the fastest. However, I think that the O(N) cost of the mediocre and good hash functions are what led to the longer runtimes in the good and mediocre cases. I still think that the badHash function had more collisions, which is why I ran many tests to see how I can better show the increase in collisions in the bad and mediocre hash functions.

The graph below tested the amount of time it took each hash method, without making a chainingHashTable object, to add things to the hash to show the runtime of just the hash functions and verify that it had the runtimes I expected. It shows that the badHash call takes less time than the good and mediocre hash.


Testing Add Time with different String Lengths
mediocreHash
goodHash
badHash
1.25E+9
1.00E+9
7.50E+8
5.00E+8
2.50E+8
0
0
10000000
20000000
30000000
size