Assignment07 Analysis

Corinne Jones

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 ascii value of the initial char in the string. I expect this to result in many collisions since it will return a limited range of numbers which will then be condensed into the appropriate range for the array. The range of numbers will be 65-90 (if the letter is capitalized) and 97-122 (if the letter is lowercase). If something like a dictionary were to be used, this would further limit the ascii range to mostly the lowercase numbers. If something like a list of Names were to be used, this would further limit the ascii range to mostly the uppercase numbers. This simplistic hash function concentrates a large number of strings to a smaller set of hash values, potentially leading to more collisions.

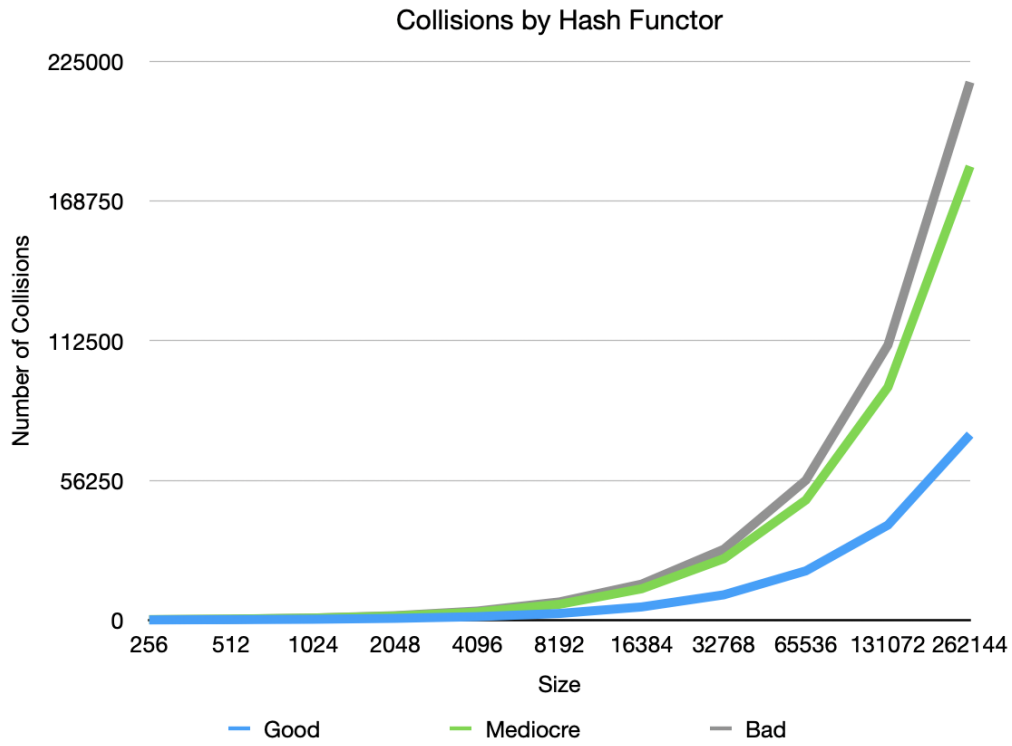
2. Explain the hashing function you used for MediocreHashFunctor. Be sure to discuss why you expected it to perform moderately (i.e., result in some collisions).

   The “mediocre” hash functor I used was pulled off of stack overflow, and was provided by a person named Jerry Coffin. This function rotates the results by some number of bits. (This person felt adamant that the djb2 algorithm by Dan Bernstein should not be widely used, which was recommended in all other resources provided, so in the name of pettiness it was chosen as my mediocre function). This method tries to balance simplicity, speed, and distribution. It is not a cryptographic functor, as is the djb2. Supposedly, a cryptographic functor used to be slower, but now the difference is negligible. It isn’t as well used or sophisticated, but still provides good results.

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).

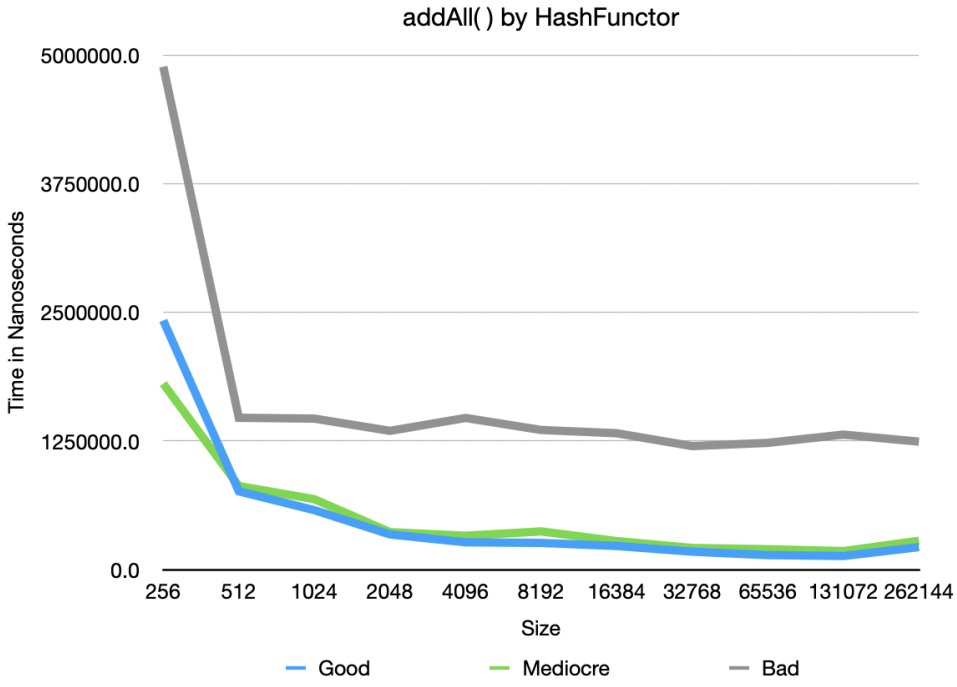
   The “good” hash functor I used was taken from one of the resources we were given and follows the formula for djb2 by Dan Bernstein. This is a widely used and trusted hash functor, being extensively tested and proven effective in applications. Why the “magic number” 33 works so well has never been fully explained, but it tends to be well-distributed, which helps minimize collisions. It also is known for its “avalanche effect,” meaning that even a small change to the input can result in a completely different hash value.

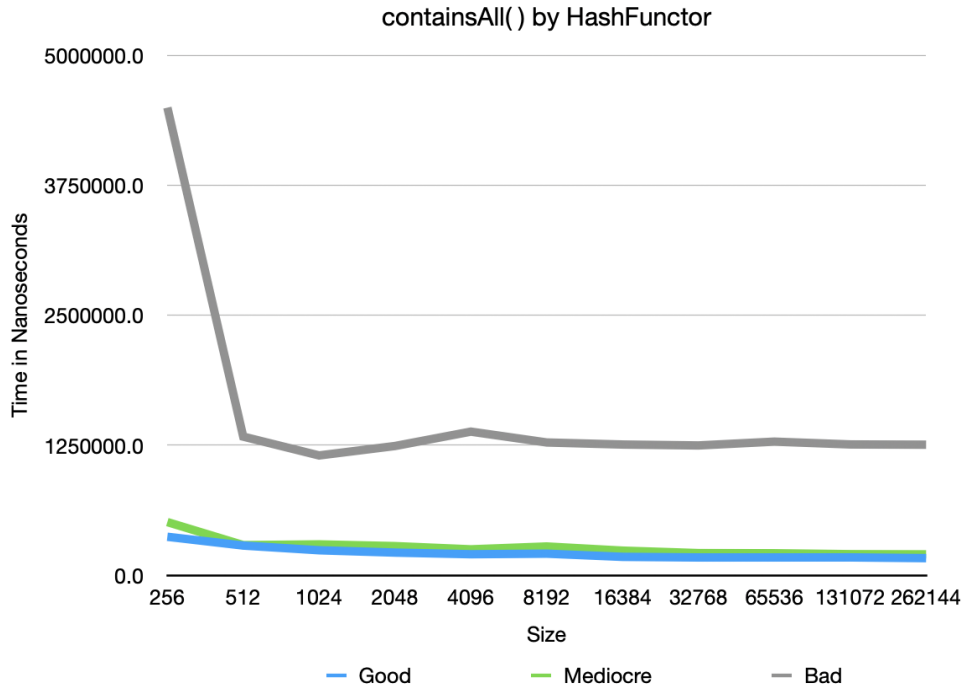
4. Design and conduct an experiment to assess the quality and efficiency of each of your three hash functions. Briefly explain the design of your experiment. Plot the results of your experiment. Since the organization of your plot(s) is not specified here, the labels and titles of your plot(s), as well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

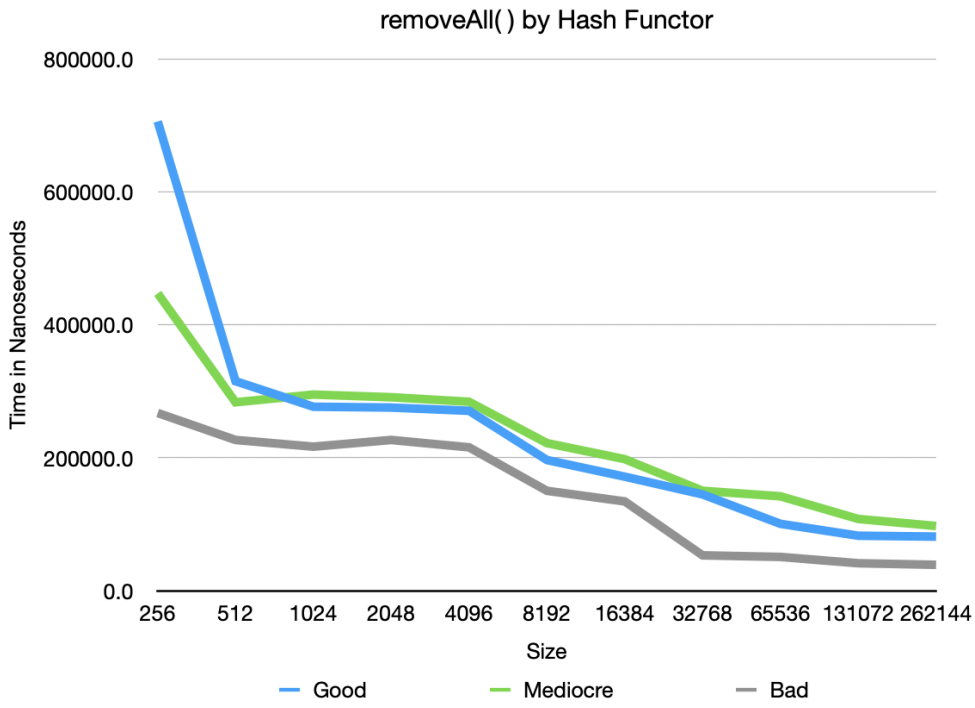


Graph 1 This graph demonstrates the number of collisions in the hash table in a variety of hash table sizes using three different functors (explained above) 1) a good hash functor 2) a mediocre hash functor and 3) a bad hash functor. The “good” hash functor has the fewest number of collisions, followed by the “mediocre,” and then “bad.” The Y axis is labeled Number of Collisions, and the X axis is labeled size of the hash table. This graph shows expected results.

This experiment was designed by first, initializing a hash table to the appropriate size, and then populating the hash table with random strings. The same number of strings were created as there is size in the hash table. Collisions were then checked by iterating through the hash table, and if an index at any point was >1 (meaning there were collisions at that index), each item in the LinkedList was added to an overall count.


addAll( ) by HashFunctor
5000000.0
3750000.0
2500000.0
1250000.0
0.0
Time in Nanoseconds
256
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
Size
Good
Mediocre
Bad


containsAll( ) by HashFunctor
5000000.0
3750000.0
2500000.0
1250000.0
0.0
Time in Nanoseconds
256
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
Size
Good
Mediocre
Bad



Graph 2, 3, 4 These graphs demonstrate the running time for three different method calls (addAll, containsAll, and removeAll) on different sized hash tables using three different functors (explained above) 1) a good hash functor 2) a mediocre hash functor and 3) a bad hash functor. The addAll( ) and containsAll( ) methods demonstrate that “good” executes the fastest, then “mediocre,” then “bad.” The Y axis labels the time in nanoseconds and the X axis labels the size of the hash table. Each graph gets slower as the hash table gets bigger, which makes sense since the bigger table would result in less collisions overall.

These three experiments were set up by first, initializing three different hash tables of the targeted size. Each hash table had a different hash functor, one of the three explained previously. After the hash table was created, the dictionary.txt (from a previous assignment) was read in from a file and converted to a linked list. This linked list was then added to each of the three hash tables. For the addAll() method, it was timed how long it took to read this file into each hash table. For the next two methods, the file was read in and added to the hash tables without timing. The next experiment targeted the containsAll( ) method, and the same file read in was passed through the containsAll( ) method and was timed. For the third experiment targeting removeAll( ), the file was passed through to have every item removed, and this was timed. Each of these methods were tested on a variety of array sizes.

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)?

   The good hash functor resulted in a runtime of O(N). This had the lowest collision rate, as expected. This hash functor iterates over each char in the string once, leading the time complexity to be proportional to the length of the string. The mediocre hash functor resulted in a runtime of O(N). This had the medial number of collisions, as expected. Although not as expected, it was only slightly better than the bad hash functor. If would have been more mediocre had it been equidistant from the good and bad hash functors. It also iterated over each char in the string once, as well as did one rotation operation for each char. The bad hash functor had an O(1) time, since it always checked the initial value in the string, no matter what. The length of the string did not matter. This had the highest collision rate, as was expected.