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" returned the size of the word as it's hash. This is like to perform poorly and cause many collisions because the size of words don't vary all that much, especially compared to potentially very large arrays. For example: if we utilize an array of 1,000 linked-lists it's unlikely that any after the first twenty or so will store any words at all. It's also worth noting that using this hashing method will cause some overlap between words, which is less than ideal. Using this scheme "in" and "as" will return the same hash, for example.

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 all of the characters in the word. This is certainly better than just returning the size of the word, but will still have the same problems as the "BadHashFunctor" did, just less frequently. This will result in larger and more disparate values for words, but they are still relatively small as the size of the array increases. It's reasonable to think that this method might return a hashed value of 2,000 but it's unlikely that it will ever reach 20,000, for instance. This also will cause some overlap - anagrams like "aft" and "fat" will produce the same value.

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).
My "GoodHashFunctor" takes an initial value of 3 and then multiplies it by 7 before adding the ASCII value of the first character. This value is then again multiplied by 7 and added to the ASCII value of the second character and this process is continued until the end of the string is reached. This is highly unlikely to produce identical Hashes because the value is dependent both on the value of each character and the order that they are in. The values will also grow very large very quickly and are more likely to outpace the size of the array which will also help ensure an even distribution. I expect no overlap and a reasonably good disruption throughout the HashTable.

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.
I ran two experiments similar to what was recommended here.

The first (figure 1) shows the proportion of lists (buckets) that contain at least one element for a number of different hash table sizes. A greater incidence of collisions

would result in fewer buckets containing an element and our different hash functions perform as we would expect. The “best” function has a relatively large number of filled buckets as the hash table grows, the “mediocre” function still does a decent job of filling buckets for relatively small numbers (up until ~1024 it matches the behavior of the “best function”), and our “worst” function falls off very quickly.

The second (figure 2) shows the time in nanoseconds required to run a .contains() method on hash tables of varying sizes. Both the “mediocre” and “best” hash functions are comparable there but they both dramatically outcompete our “bad” hash function. As we would expect the cost to run a .contains() method lessens as out Hash Table grows in size.

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)?

Recall that our “bad” Hash Function simply returns the size of the input string, therefore we expect the cost to be O(1), or the constant cost required to call the “.size()” method. Both our “mediocre” and “good” Hash Functions have to iterate over the size of the String and run mathematical operations on them. Subsequently, would expect them to be O(N).

Overall, each of the functions ran as I expected. I was a little bit surprised that our .contains() plot didn’t show more dramatic improvement when comparing the “mediocre” and “good” functions, but it’s possible that a larger number of input words would have made that difference more apparent.

Figure 1 -

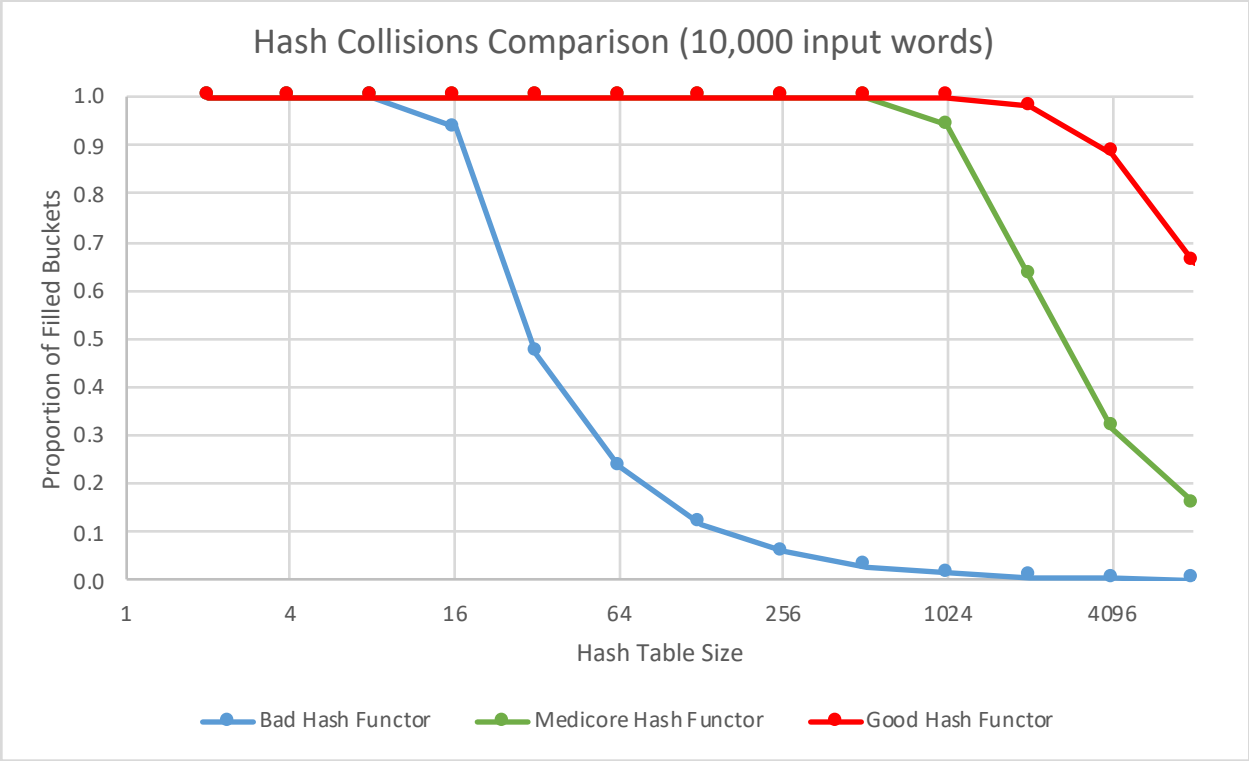

Figure 2 -