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).

   The BadHashFunctor function returns the length of the item parameter of type String. Strings that are the same length will always hash to the same value, regardless of their content. This will result in a lot of 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).

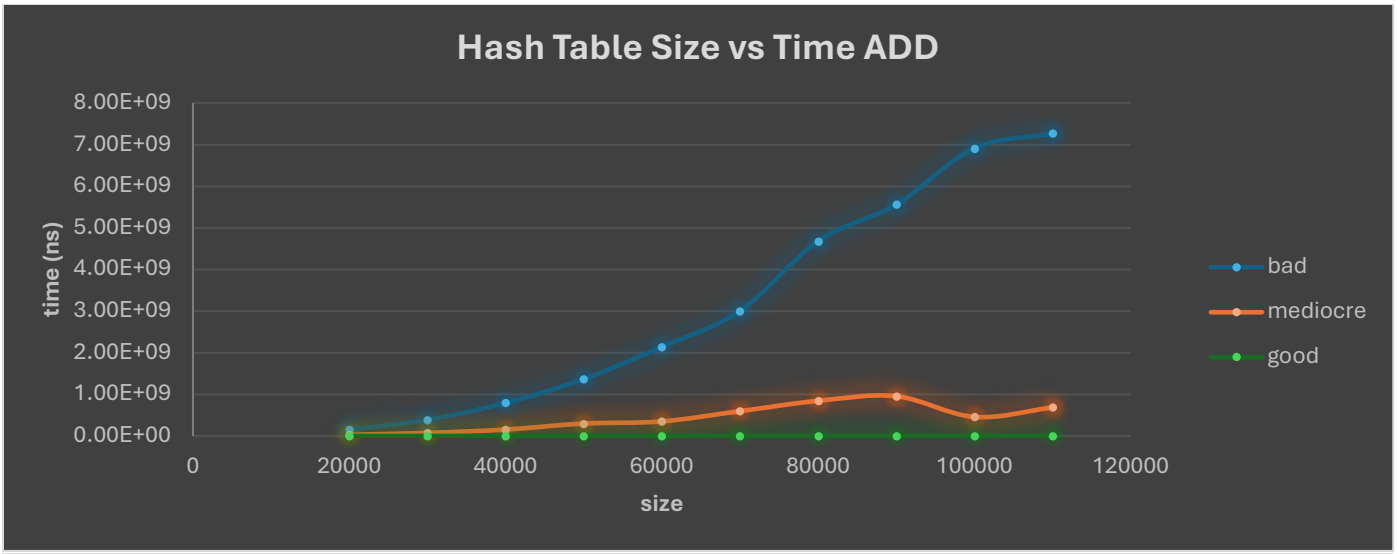
   The MediocreHashFunctor takes into account the content of the string but the hash values are not evenly distributed. So strings of the same length and that have the same characters but in different order will have the same hash values.

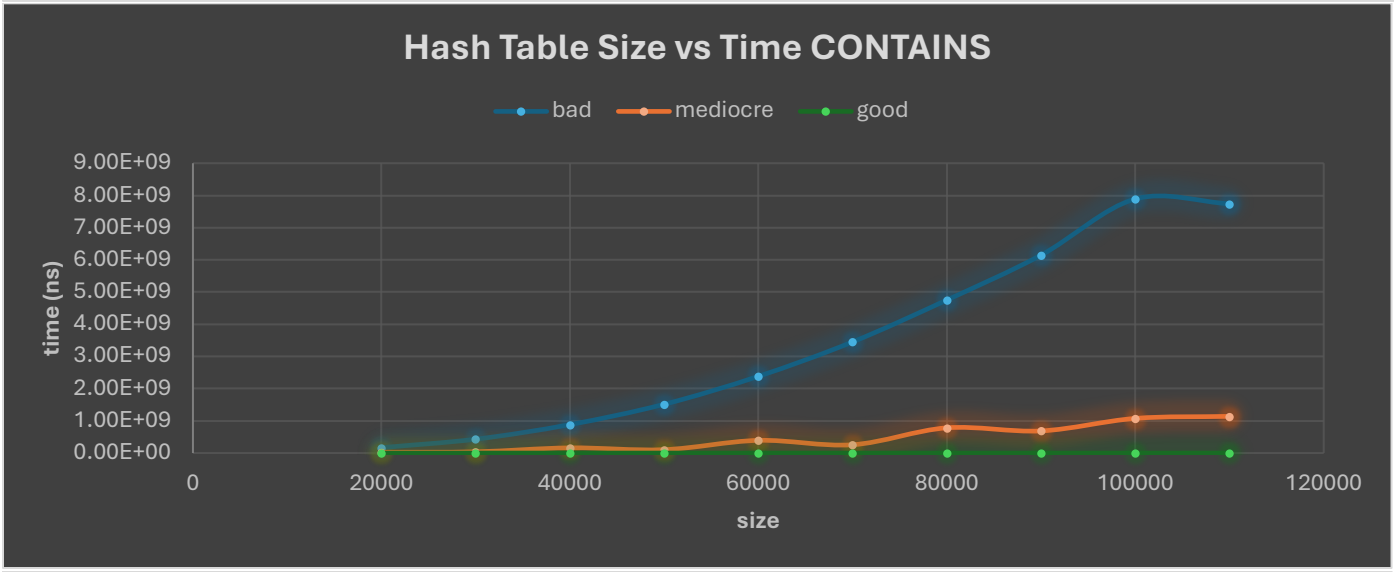
3. Explain the hashing function you used for GoodHashFunctor. Be sure to discuss why you expected it to perform well (i.e., result in few or no collisions).

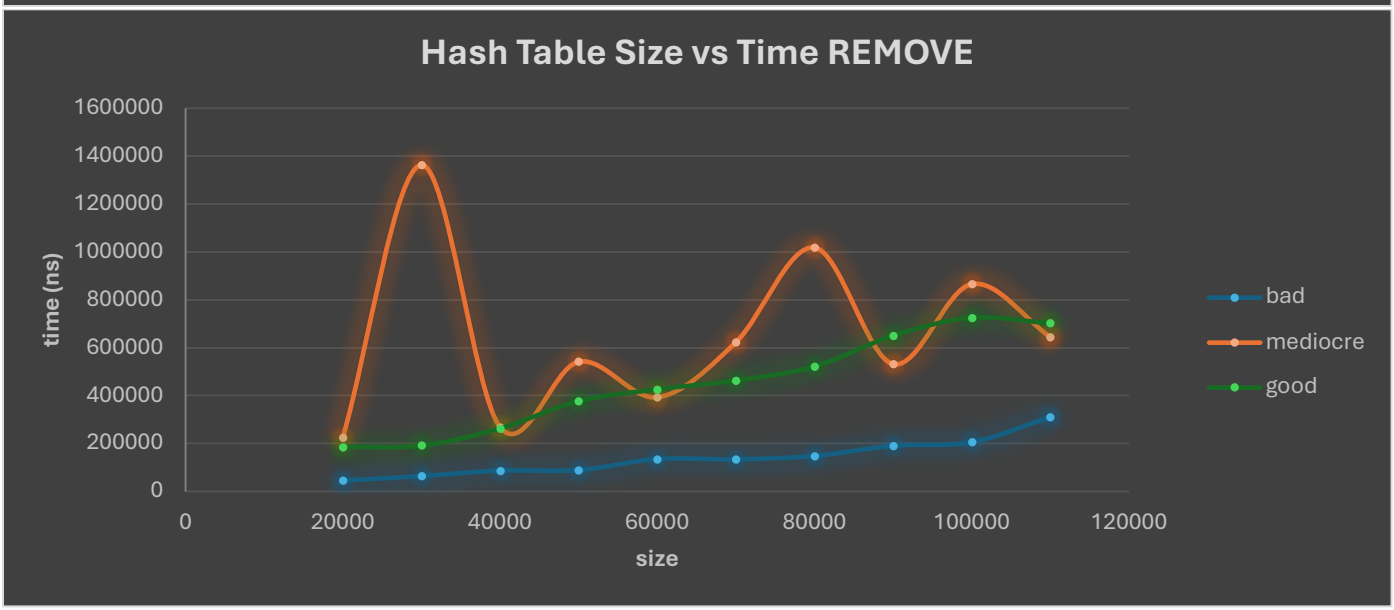
   The GoodHashFunctor uses the position of characters in the string, effectively reducing collisions. It also uses a polynomial rolling hash function and the 31x multiplier spreads out the hash values.

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.

   Three timing classes were made. Each implements the TimerTemplate abstract class authored by Ben Jones. Each class tests a different method (add, contains, and remove). All classes have the same 3 member variables. An instance of ChainingHashTable, a List<Strings> that will be used to add items to the hash table, and a functor. Each classes constructor is the same. It calls the parent TimerTemplate constructor to initialize shared timing logic. It also stores the provided hash functor for use in the hash table when we loop through the functors during the actual timing. The setup will require an initialization of the ChainingHashTable that sets the size to 10 times the value of n, and the functor. The array list of strings will be initialized and populated with strings that are equal to “String” + i. This populates the array list with n unique strings. For the add timing class, the timingIteration method will use a for each loop to add every string in the array list to the hash table. For contains and removes, this adding will be done in the set up. For contains and remove, the timingIteration class will time how long it takes to check if each item is contained and each item is removed, respectively. The compensationIteration method will include an empty for each loop to measure baseline overhead of iterating over data. The main method will include the problemSizes which are 10000 – 100000 and timesToLoop = 10. Run the timer for each functor and print the results to a csv file.


Hash Table Size vs Time ADD
8.00E+09
7.00E+09
6.00E+09
5.00E+09
4.00E+09
3.00E+09
2.00E+09
1.00E+09
0.00E+00
time (ns)
0
20000
40000
60000
80000
100000
120000
size
bad
mediocre
good


Hash Table Size vs Time CONTAINS
bad
mediocre
good
9.00E+09
8.00E+09
7.00E+09
6.00E+09
5.00E+09
4.00E+09
3.00E+09
2.00E+09
1.00E+09
0.00E+00
time (ns)
0
20000
40000
60000
80000
100000
120000
size


Hash Table Size vs Time REMOVE
1600000
1400000
1200000
1000000
800000
600000
400000
200000
0
time (ns)
0
20000
40000
60000
80000
100000
120000
size
bad
mediocre
good

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)?

   BadHashFunctor

   Returns the length of the string.

   The Time Complexity is O(1) because it performs a constant number of operations no matter what the input size is. The Space Complexity is also O(1) because it uses no additional data structures

   High number of collisions expected and observed because multiple strings have the same length and map to the same constant value

   MediocreHashFunctor

   Sums the ASCII values of all characters in the string

   Time Complexity is O(N) because it iterates through every character in the string (N characters). Space Complexity O(1) because it uses a single integer variable to store the sum.

   Performs better than the BadHashFunctor because it takes the characters in the string into account. But it has weaknesses because Strings with the same characters but in different order will hash to the same value

   GoodHashFunctor

   Uses a polynomial hash algorithm.

   Time Complexity O(N) because it iterates through every characer in the string (N characters). Performs a constant number of operations for each. Space Complexity O(1) as it uses a single integer variable to compute the hash.

   Minimizes collisions because it generates more unique hash codes by factoring in the order of characters, the characters themselves, and a multiplier. It distributes strings more uniformly across the hash table.