when you are satisfied that your program is correct, write a brief analysis document. The analysis document is 10% of your Assignment 7 grade. Ensure that your analysis document addresses the following.

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).

```
public class BadHashFunctor implements HashFunctor{
    3 usages  whitney
    @Override
    public int hash(String item) { return item.charAt(0); }
}
```

My bad functor returns the ascii value of the first character in the string. This is a bad hash functor because every string that starts with the same character will collide. (i.e. every string that has the first character of ‘s’ will collide). It’s also bad because in very large arrays (with the capacity in the hundreds or thousands), most indices would be empty with tons of collisions within the limited range of ascii values

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).

```
public class MediocreHashFunctor implements HashFunctor {
    3 usages  new *
    @Override
    public int hash(String item) {
        int hash = 0;
        for (int i = 0; i < item.length(); i++) {
            hash += item.charAt(i);
        }
        return hash * 19;
    }
```

My mediocre hash functor adds the values of all the ascii characters and then multiplies that value by 19 at the very end and returns it. This is better because it results in fewer collisions due to the values returned from the hash being more variable and larger than the value returned from the bad functor. The main problem with this functor is that inverted strings will return the

same values (i.e. “string” and “gnirts” will have the same hash returned). The value is multiplied by 19 at the end so help prevent small strings from causing many collisions if the array is large. This functor is better than the bad functor but strings with the same/similar characters would result in many collisions.

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).

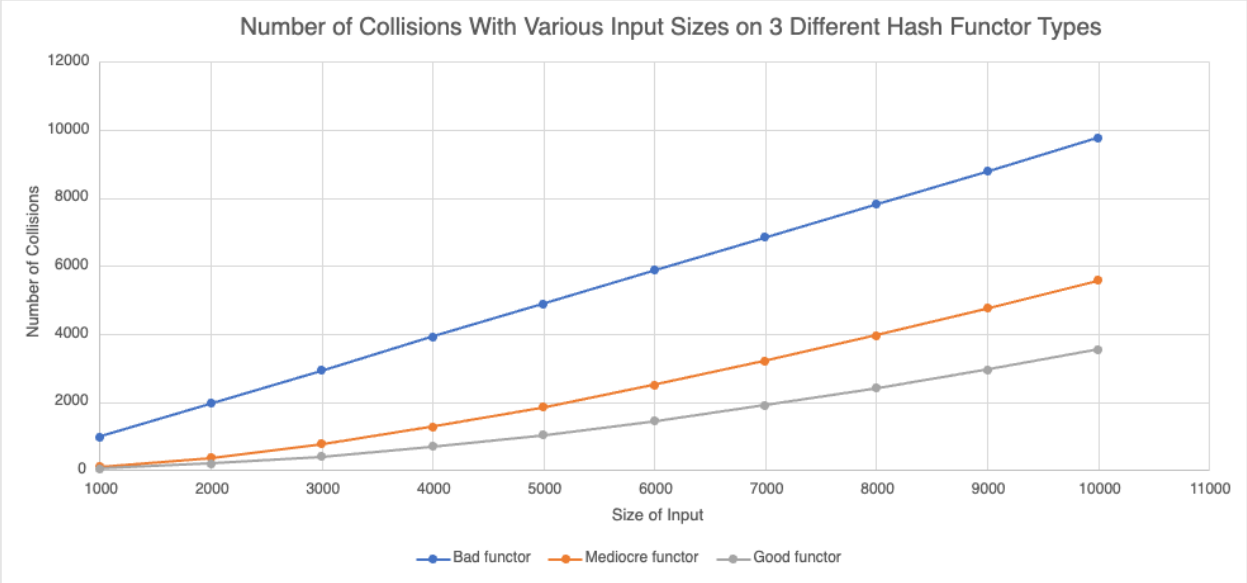
```
public class GoodHashFunctor implements HashFunctor {
    3 usages  new *
    @Override
    public int hash(String item) {
        int hash = 5381;
        int c;
        for (int i = 0; i < item.length(); i++) {
            c = item.charAt(i);
            hash += ((hash << 5) + hash) + c;
        }
        return hash;
```

My good hash functor is much more complex than the bad and mediocre hash functor, resulting in large and varied values returned. How it works is each character in the string is individually pulled out. Then the existing hash value will be equal to the current value plus the following: the current hash value added to the current hash value bit shifted 5 bits the right and then the ascii value of the character is added to that. The bit shifting 5 results in widely varied values when added to the current hash value. This more complex way of adding and bit shifting ensures that inverted strings will result in different values which will result in fewer collisions. The values returned from this will also be much larger which will help prevent smaller strings from being grouped near the beginning of the array as would happen with the bad function and could happen with the mediocre functor.

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

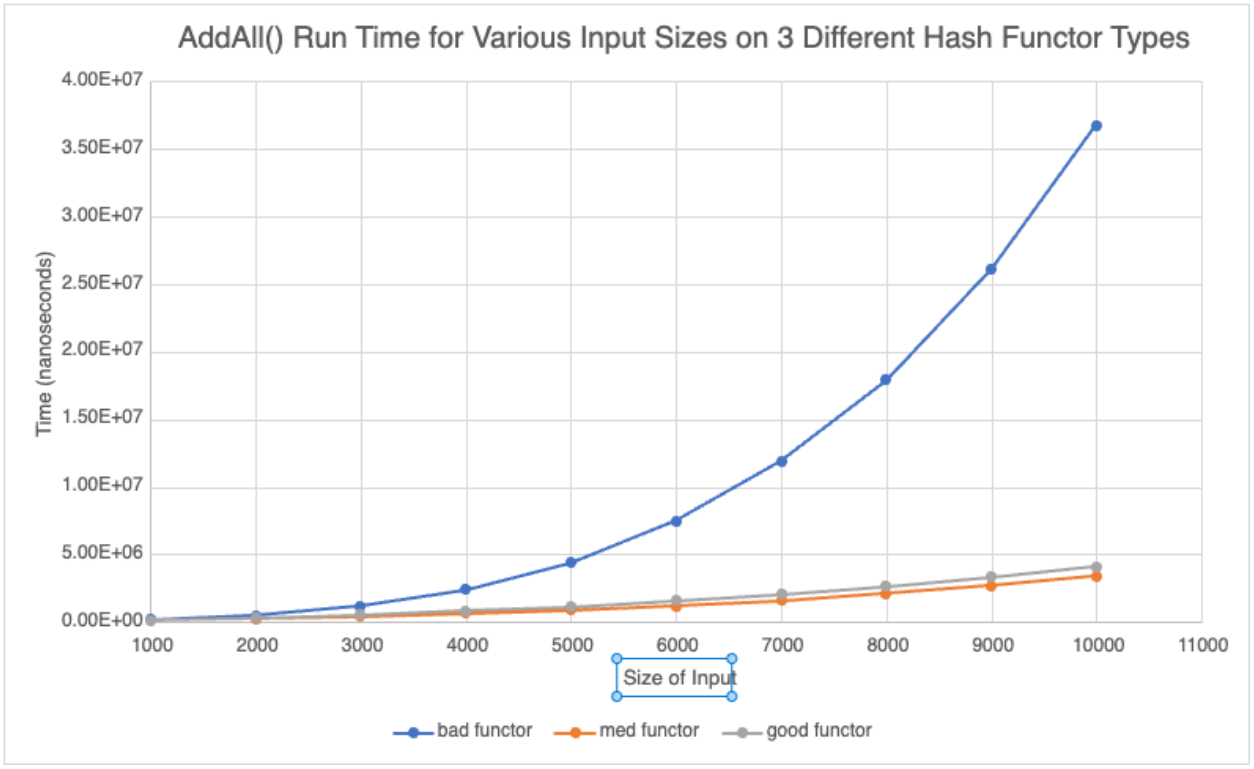
shows the actual running time required by various operations using each hash function for a variety of hash table sizes.

For the first experiment of testing collisions I created a hash table with a size of 10,000. From there I used each of the hash functors (good, bad and mediocre) to add input sizes from 1000 to 10,000 with increments of 1,000. I calculated the number of collisions by iterating through the hash table and for any index within the array, if there was more than 1 item within the linked list, I calculated the size of the linked list minus 1 (since the initial placement is not a collision). The graph looks about how I expected it to. The bad hash functor collisions is nearly linear with the increased size due to the poor placement resulting in many collisions (and unused space in the array). The mediocre one performed slightly better than I expected and then good one had less than half of the collisions of the bad functor

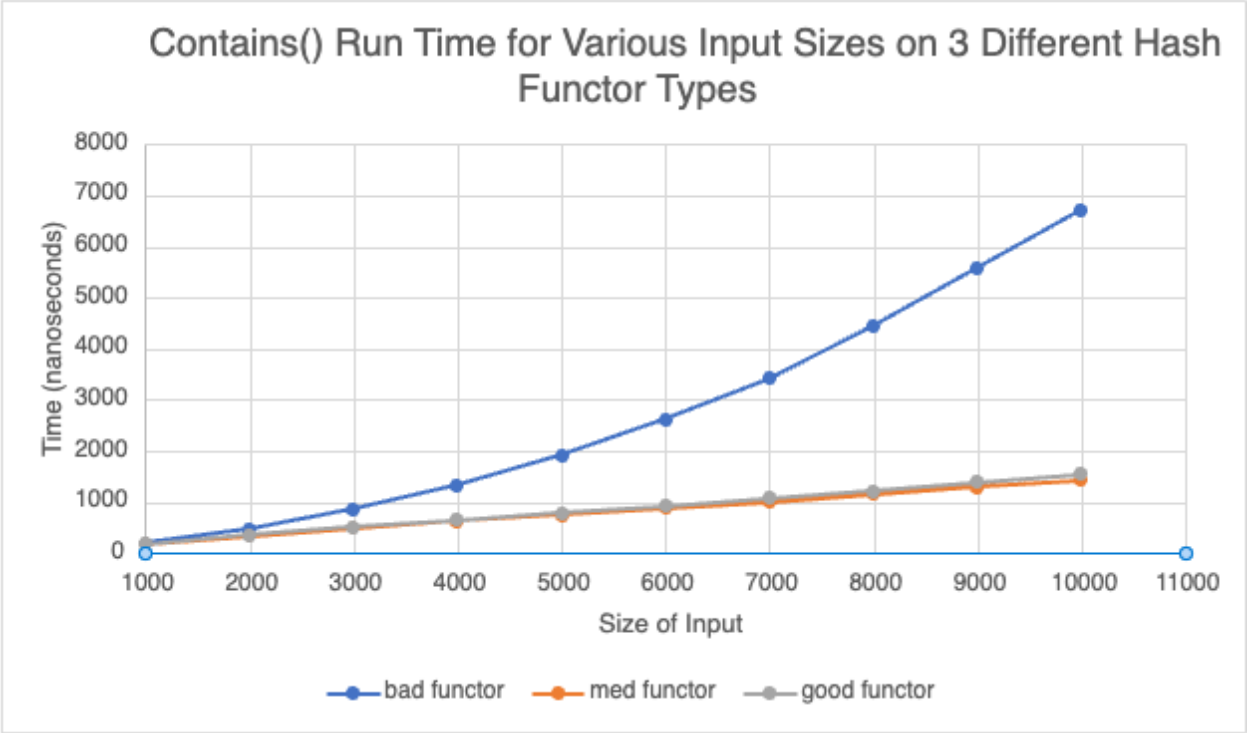


Below are the run times for addAll(), remove(), and contains() on increasing input sizes from 1000 to 10,000 with increments of 1000.
As can be seen, for addAll(), the bad hash functor takes significantly longer than the other 2. This is because before a string can be added, it first checks if it is already contained within the hash table. To do this, the hash functor returns the index within the array that the string would be at (if contained) then it travels along the linked list at that index checking if each element along the linked list is the element we are looking for. Because there are so many collisions and so many wasted index locations with the bad hash functor, this takes much longer to check if the element is contained before adding it. This is the time consuming part of the addAll() and takes increasingly longer as the input size increases since there are more and more elements to check for each item being added.

The graphs for timing on contains() and remove() are similar but to a smaller scale since only 1 element is being removed or checking if it is contained within the hash table. The bad hash functor still results in long linked lists that will need to be checked to find the given element


AddAll() Run Time for Various Input Sizes on 3 Different Hash Functor Types
Time (nanoseconds)
4.00E+07
3.50E+07
3.00E+07
2.50E+07
2.00E+07
1.50E+07
1.00E+07
5.00E+06
0.00E+00
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
11000
Size of Input
bad functor
med functor
good functor


Remove() Run Time for Various Input Sizes on 3 Different Hash Functor Types
Time (nanoseconds)
16000
14000
12000
10000
8000
6000
4000
2000
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
11000
Size of Input
bad functor
med functor
good functor



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 cost of the bad hash functor is O(1) since it just takes the first character's ascii value and returns it. The cost of the mediocre hash functor is O(N) since it adds up the values of each character's ascii value before manipulating it. The good hash functor is also O(N) since it also adds the value of each ascii character while manipulating it along the way. Overall my hash functors performed as expected. I didn't expect my mediocre hash functor to perform as well as it did in terms of run time for specific methods, but thinking about it more I realized that the dramatic reduction in collisions would be expected to have a big impact on the run time. The bad functor I definitely expected to have a huge number of collisions due to so few index locations being used.

Upload your solution (.pdf only) through Canvas.