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 int hash(String item) {

        return item.length();
}
```

Answer:
My Bad Hash Function simply returns the length of the string. In the case of inputting normal English words to the hash function, the index this hash function returns will always range between 1 to no more than 20. When the array size gets large, the distribution of this function output will largely skewed at the beginning of this function, which results in significant number of collisions since English words with the same length will be mapped to the same index all the time. The advantage of this hash function is the computation cost is really low.

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 int hash(String item) {
        int hashVal = 0;

        for(int i = 0; i < item.length(); i++) {
             hashVal += item.charAt(i);
        }
        return hashVal;
}
```

Answer:
My Mediocre Hash function sums up all the ASCII values of each character in the string, with equal weight to each of the character in the string. The Ascii value letter is roughtly between 50 – 100. So for normal English word with 5 to 10 characters, the result hashVal will be between 250 – 1000.

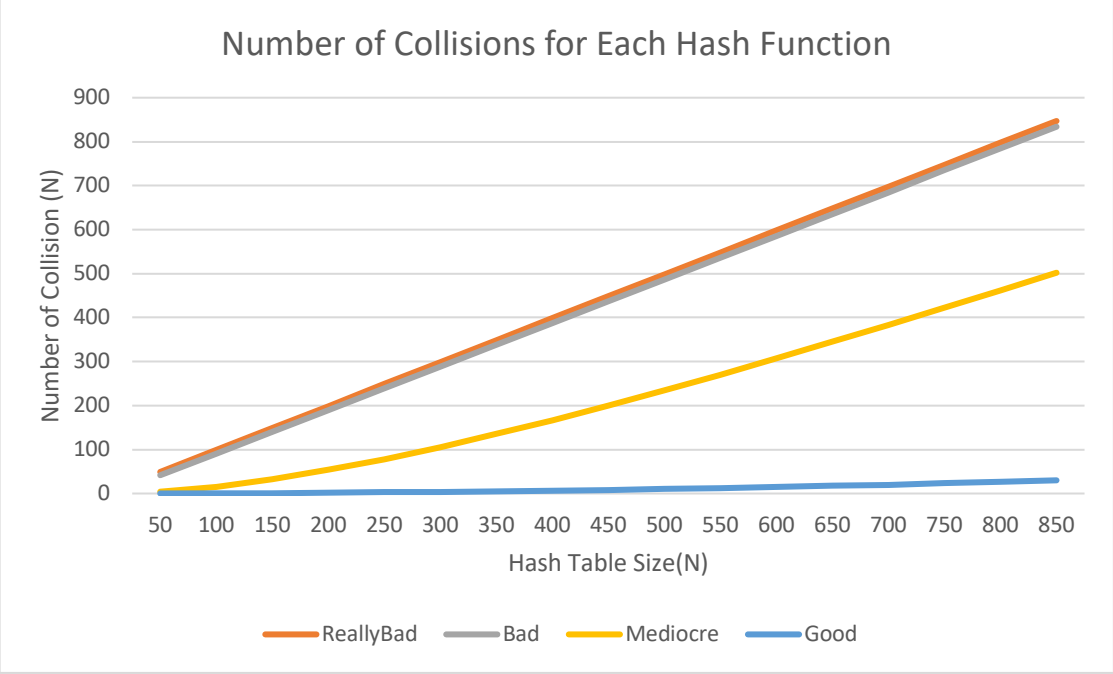
The reason why this results in fewer collision than BadHashFunction is that when the array size (Table capacity) is small, for example like 100, this hash function can perform a good job distributing the result index evenly along 0 – 100. However, when the array size gets really big like 10000. The hashVal (250 – 1000) will be largely skewed the beginning end of the array, leading to lots of collisions. To improve this hash function, we need to come up with a way to distribute evenly the resulting index.

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 int hash(String item) {
          int hashVal = 0;

          for(int i = 0; i < item.length(); i++) {
               hashVal = 33 * hashVal + item.charAt(i);
          }
          return hashVal;
     }
```

Answer:
This hash function gives weight to each character and the resulting hashVal will have a lot larger range, and therefore is able to distribute the result index evenly. When distributing evenly, the number of collisions will be reduced.

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



Answer:
In the above experiment, the X-axis is the hash Table size, ranging from 50 to 850, with 50 incremented by 50. For each table size, I addAll() an array list of English words with the size of corresponding hash table size and store the number of collision, with 100 iterations. The Y-axis is the average number of collisions for each table size. In terms of capacity, I chose 10,000 array size. The reason for choosing this large number is that the skewness of each hash function's distribution will be obvious to us.

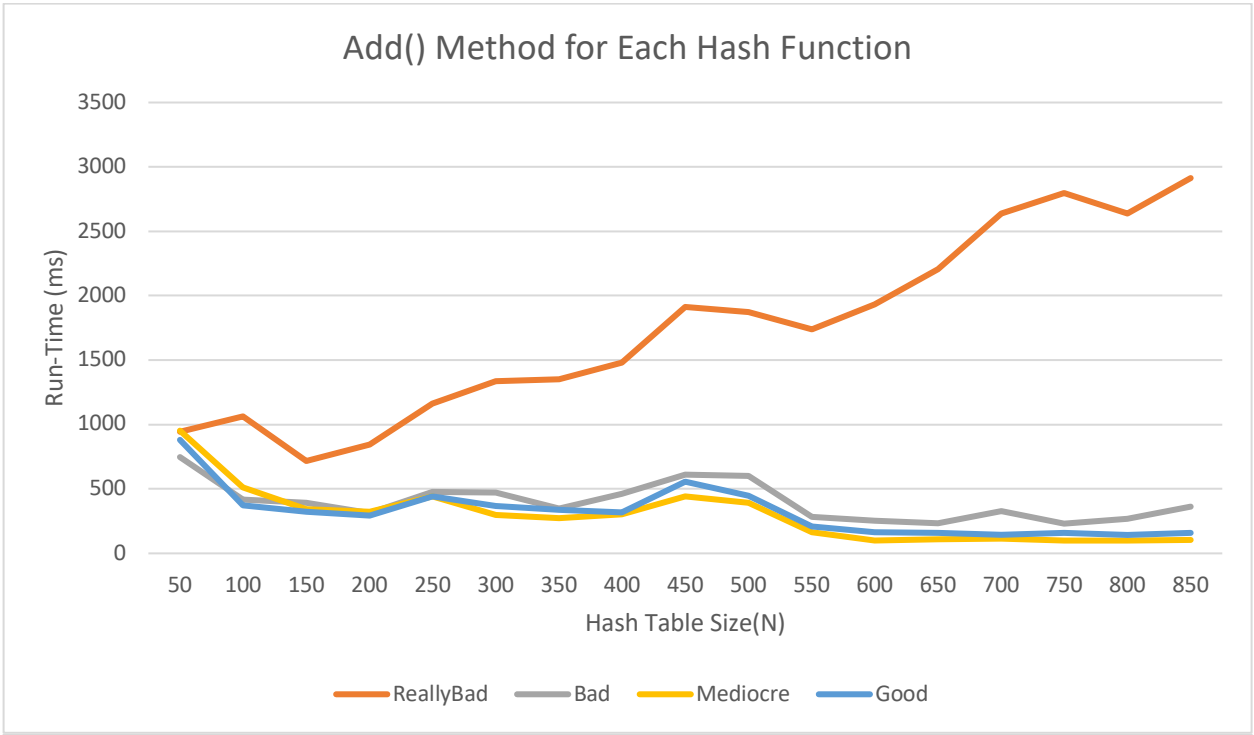
I also added a really bad hash function for illustrating purposes:

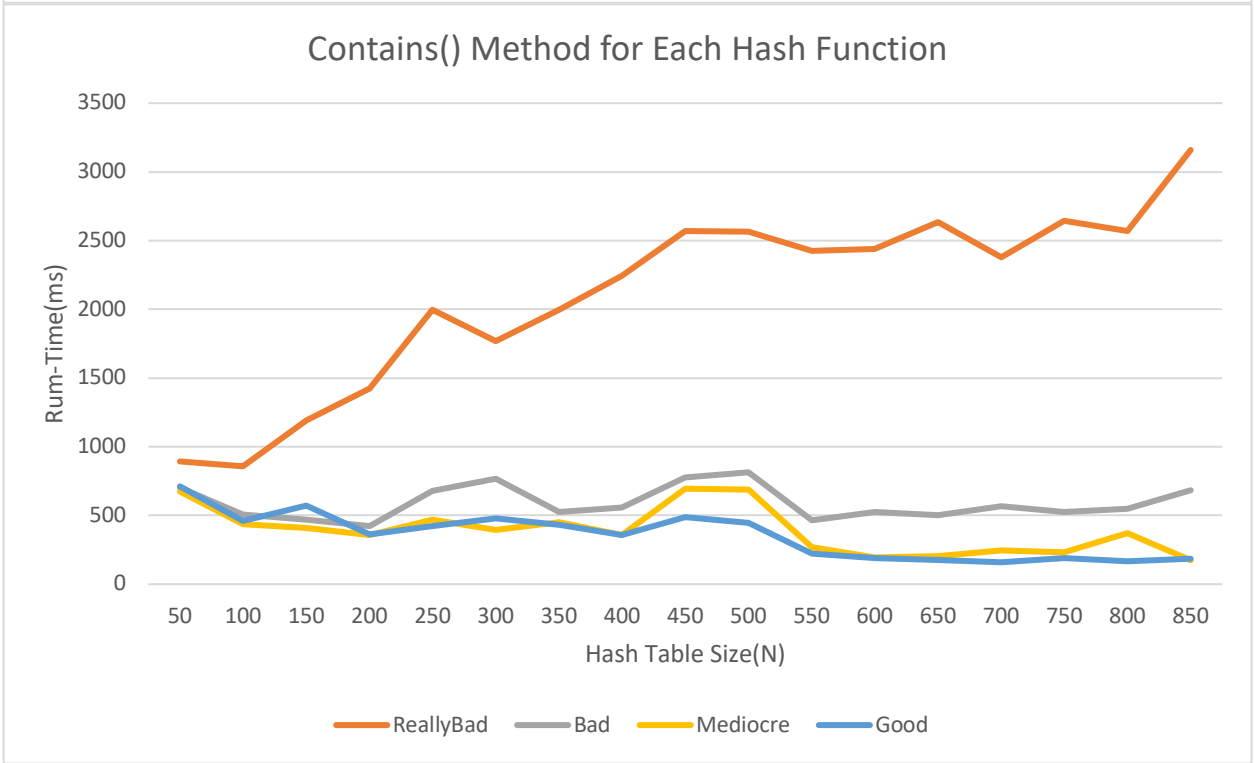
```
@Override
        public int hash(String item) {

             return 0;
        }
```

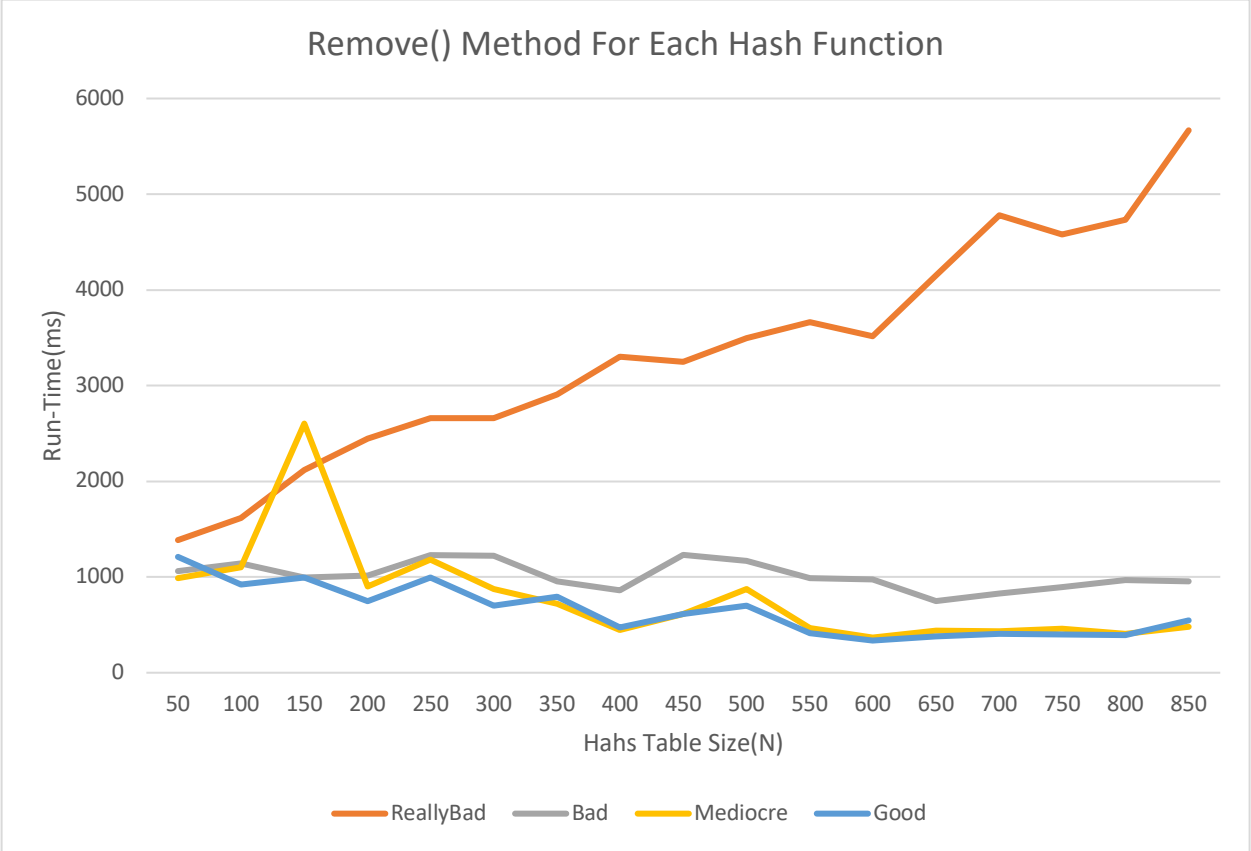
From the above graph we can see that for both ReallyBad and Bad hash functions, the number of collisions almost equals to the number of hash table elements added. The reason is that for ReallyBad hash function, the index returns is always 0, so at the 0 index cell the linked list keeps growing, so the number of collisions precisely equal to the number of elements added to the table. For Bad hash function, since the English words being added are all no more than 10 length, so the index skews at 0 – 10 cells. When the number of elements added to the table grows, the number of collisions grows significantly.

In terms of Mediocre Hash Function, since the hash table capacity is quite large == 10,000 and hashVal has to % 10, 000 each time, so the resulting index will skew around to front 10% - 20% of the base array, which results in many collisions.

In terms of Good Hash function, the resulting index distributes evenly, and therefore the numbers of collisions are minimized to the least.


Add() Method for Each Hash Function
Run-Time (ms)
3500
3000
2500
2000
1500
1000
500
0
50 100 150 200 250 300 350 400 450 500 550 600 650 700 750 800 850
Hash Table Size(N)
ReallyBad
Bad
Mediocre
Good


Contains() Method for Each Hash Function
Rum-Time(ms)
3500
3000
2500
2000
1500
1000
500
0
50 100 150 200 250 300 350 400 450 500 550 600 650 700 750 800 850
Hash Table Size(N)
ReallyBad
Bad
Mediocre
Good



Answer:

In the run-time experiment, the X-axis is the hash Table size, ranging from 50 to 850, with 50 incremented by 50. For each table size, I addAll() an array list of English words with the size of corresponding hash table size. For each table size, I iterate 100 time and for each iteration I perform different operations such as add() or remove() on the existing hash table, eventually I calculated the average running time.

Since we are using separate chaining here, the cost of adding is constant, and that's the reason why Bad, Mediocre, and good perform similarly regardless of table size.

In the case of remove(), and contain(), the expected cost will be linear relative to the size of corresponding linked list of that index. Since we are using java's built-in library and length of the linked list is not significantly large, so the run-time appears constant.

One thing to notice is that for ReallyBad Hash function, the run-time increases with the number of elements stored in the hash table. I suspect the reason to be is that calling new and creating a large linked list on the heap is expensive.

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)?

Answer:
Bad Hash Function is O(1)
Mediocre Hash Function and Good Hash Funciton is O(N), with N being the length of the string. It does result in the expected number of collisions.