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).
The way I create the badHashFunctor, I choose the first character of the string and times the length of that string.
So the same length and same first character will have collisions. Thus, more collisions than others.

```
public int badHash(String str) {
    if (str == null) {
        return -1;
    }
    return str.charAt(0) * str.length();

}
@Override
public int hash(String item) {
    if (badHash(item) == -1) {
        return -1;
    }
    return badHash(item);
}
```

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).**
Mediocre functor loop through a string and keep plus that ascii value to the output string.
I expect it to be moderate bc it takes 27^8 possible keys. The keys are strings of 8 ASCII letters and spaces which are in the
range of 65 - 95. Hence, the sums of 8 char values will be in the range of 520 - 760.

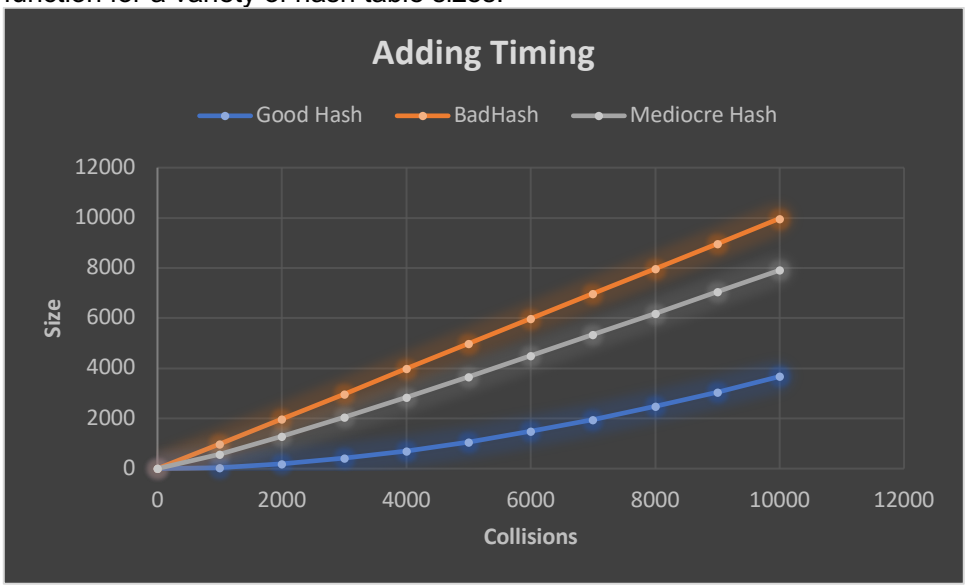
```
public int mediocreHash(String str) {
    int hash = 1;
    for (int i = 0; i < str.length(); i++) {
        hash *= str.charAt(i) / (i+1);
    }
    return hash;
}
@Override
public int hash(String item) {
    return mediocreHash(item);
}
```

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 hash number will be different after everytime it calculate and plus the string of character to that hash number.
Hence, the possibility of collisions are less.

```
public int goodHash(String str) {
    int hash = 5381;
    for (int i = 0; i < str.length(); i++) {
        hash = hash * 33 + str.charAt(i);
```

```
    }
    return hash;
}
@Override
public int hash(String item) {
    return goodHash(item);
}
```

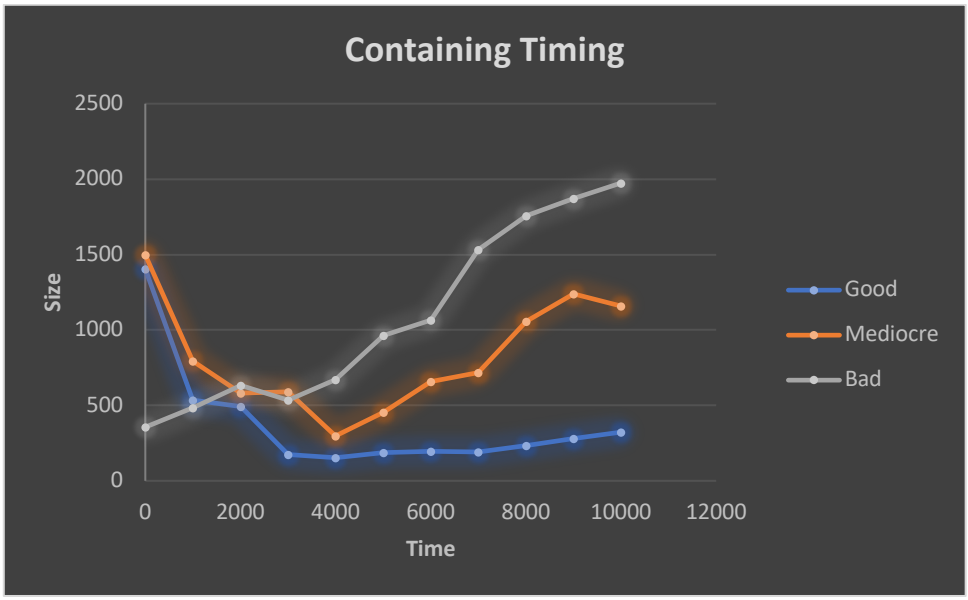
4. Design and conduct an experiment to assess the quality and efficiency of each of your three hash functions. Briefly explain the design of your experiment. Plot the results of your experiment. Since the organization of your plot(s) is not specified here, the labels and titles of your plot(s), as well 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.



Basically I add the random array string to the test array and every time the for loop run the size of the array size will increase.

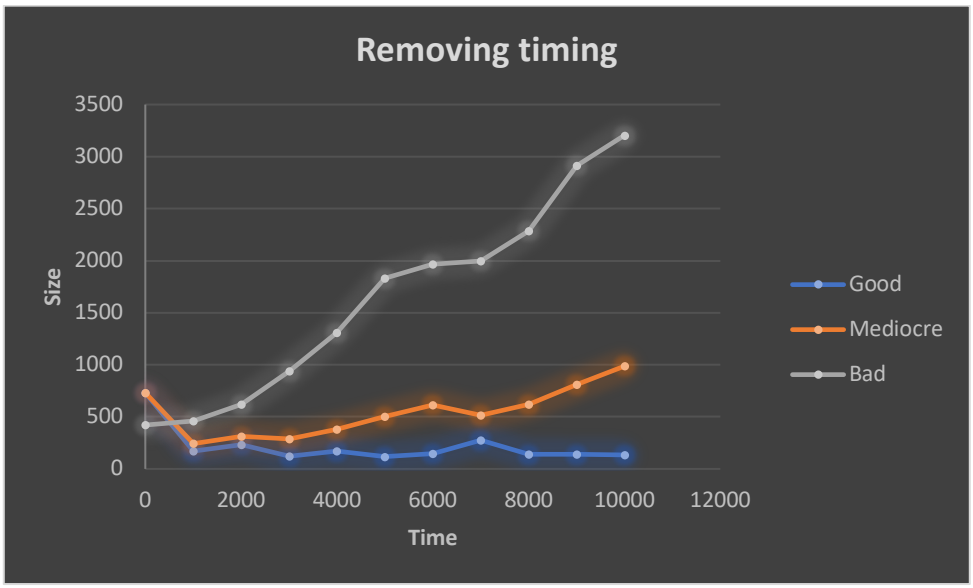
```
for (int exp = 1; exp <= capacity; exp+=1000) {
    int size = exp;
    int totalCollision = 0;

    for (int iter = 1; iter < iterCount; iter++) {

        ChainingHashTable goodHashTable = new ChainingHashTable(capacity,
mediocreHashFunctor);
        var temp = buildStringList(size);
        for (String t : temp
        ) {
            goodHashTable.add(t);

        }
        int collision = goodHashTable.get_collision();
        totalCollision += collision;
```

```
    }
    int avgCollision = totalCollision/iterCount;
    System.out.println(size + "\t" + avgCollision);

}
```



I add the string to the test array and using the random function to find the element that contains the item.

```
for (int iter = 1; iter < iterCount; iter++) {


                String findElement = test.get(random.nextInt(size));
                startTime = System.nanoTime();
//                goodHashTable.contains(findElement);
                mediocreHashTable.contains(findElement);
//                badHashTable.contains(findElement);
                stopTime = System.nanoTime();
                totalTime += stopTime - startTime;


            }
```



Same method as I am testing the containing timing. I first add the string to the test array and using random function to find that element and remove it. But different thing is I add it back after record the timing, so the size of that array would not be smaller. If we not adding that element that we removed, then the timing of removing element would be faster.

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)?**

Upload your solution (.pdf only) through Canvas.

For the bad hash, its O(1) bc it take every first character of the string.
For the mediocre hash, its O(N), bc it is based on the string length itself, so it takes O(N) times
Good hash is same as mediocre hash O(N).
At first, the mediocre and bad hash are close to each other but after trying few times to mess up with the value, it started to separate a little bit.