Assignment 6 Hash Table – Analysis Document

U1426365 Jin-Ching Jeng

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).

For the BadHashFunctor, it returns the ASCII value of the first letter of the input string. After getting the value from BadHashFunctor, the index of hash table is [ASCII value % capacity]. It is quite bad because the ASCII value has only 127 values in total (even less for letters), so all elements will cluster in the first 127 indexes of an array with lots of collisions.

```
public int hash(String item) {
    // compute ASCII value of the first letter
    int ascii = item.charAt(0);
    return ascii;
}
```

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).

For the MediocreHashFunctor, it returns the accumulated ASCII value of the string. After getting the value from MediocreHashFunctor, the index of the hash table is [accumulated ASCII value % capacity]. It is better than the BadHashFunctor because the elements will be allocated in more than 127 indexes of an array.

```
public int hash(String item) {
    // compute accumulated ASCII value of the string
    int sum = 0;
    for( int i = 0; i < item.length(); i++ ){
        sum += item.charAt(i);
    }
    return sum;
}
```

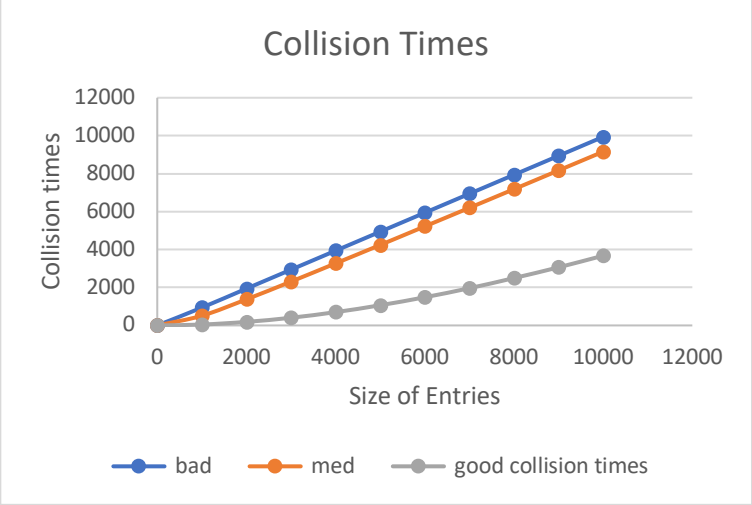
3. Explain the hashing function you used for GoodHashFunctor. Be sure to discuss why you expected it to perform well (i.e., result in few or no collisions).

For the GoodHashFunctor, it returns the computing value of the string from djb2 algorithm. It iterates the given array of characters and performs the following operations for each character: Multiply the hash variable by 33, and add the ASCII value of the current character to it. Multiplying by 33, which is a magic number that hasn't been adequately explained, is been represented using bit shift in code. This is the best hash functor of these three because it has more variation for different strings. After getting the value from GoodHashFunctor, the index of the hash table is [return value % capacity].

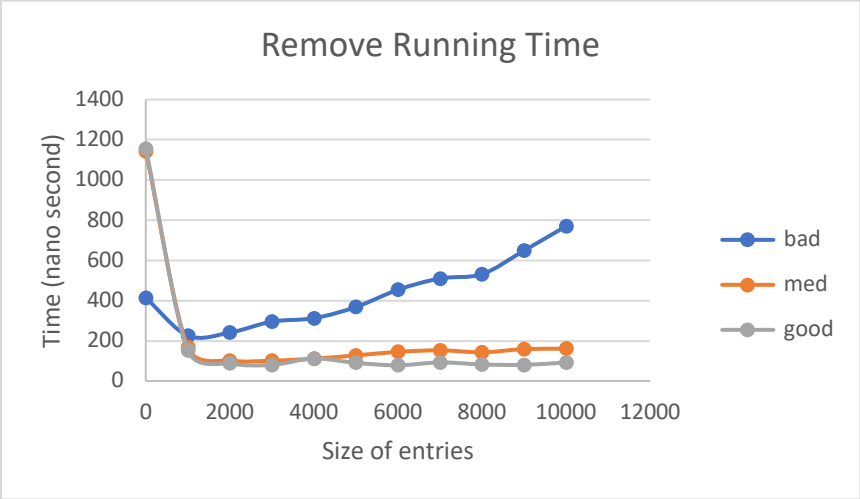
```
public int hash(String item) {
    int hash = 0;
    for (int i = 0; i < item.length(); i++) {
        hash = item.charAt(i) + ((hash << 5) + hash);
    }
    return Math.abs(hash);
}
```

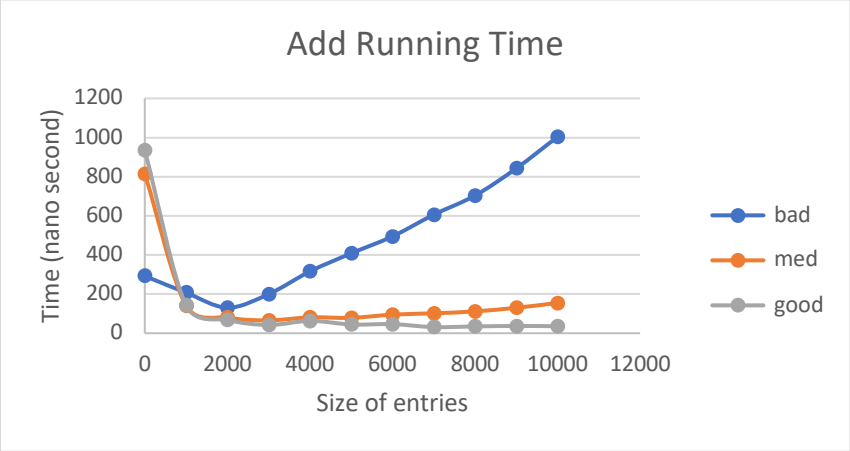
4. Design and conduct an experiment to assess the quality and efficiency of each of your three hash functions. Briefly explain the design of your experiment. Plot the results of your experiment. Since the organization of your plot(s) is not specified here, the labels and titles of your plot(s), as well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

The first plot tests the number of collisions incurred by each hash function for a variety of lengths of the string. Fixed variables are the capacity of the table = 10007(prime number), the length of strings = 50, and the iteration time = 1000.



The following three plots test the running time for three operations by each hash function for various lengths of the string. Fixed variables are the capacity of the table = 10007(prime number), the length of strings = 30, and the iteration time = 1000. The first data (size = 1) can be ignored because the computer might not be stable in the beginning.





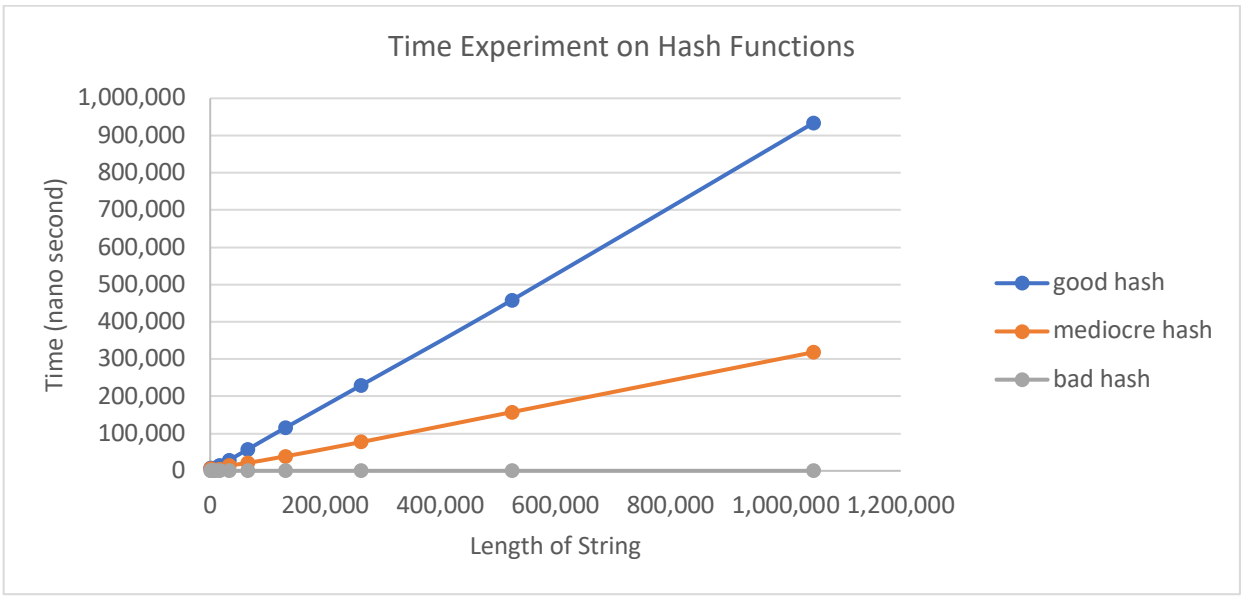
5. What is the cost of each of your three hash functions (in Big-O notation)? Note that the problem size (N) for your hash functions is the length of the String, and has nothing to do with the hash table itself. Did each of your hash functions perform as you expected (i.e., do they result in the expected number of collisions)?

For good hash function: O(N)

For mediocre hash function: O(N)

For bad hash function: O(1)

This is as I expected. For bad hash function, it only takes the first letter of the string to compute the hash key no matter how big the length of the string is. For good and mediocre hash functions, they use for loop to examine every character in the string.


Time Experiment on Hash Functions
1,000,000
900,000
800,000
700,000
600,000
500,000
400,000
300,000
200,000
100,000
0
Time (nano second)
0
200,000
400,000
600,000
800,000
1,000,000
1,200,000
Length of String
good hash
mediocre hash
bad hash