Analysis Doc Assignment 07

Aiden Pratt

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.charAt(0);
}
```

This hash functor I made just returns the item.charAt(0). This will return the first letter of the String that is passed in, organizing my HashTable in ascii order of the first characer. This can result in a lot of collisions if many words with the same starting letter are passed in, causing more time to search in the LinkList portion of the HashTable. In my random String generator, I allow for 62 possible characters. This means the greatest size of the array of LinkedLists can only be 62. So when the test is run 250,000 times, many of the randomized strings passed in, and quickly all, will be placed in the same LinkedList based on this hash functor.

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 c = 1;
    int ch = item.length();
    int sum = 1;
    while (c++ != ch) {
        sum *= item.charAt(c-1);
    }
    return Math.abs(sum);
    }
```

My Medicore Hash Functor declares three ints which I use to return a value for the index in the array of LinkedLists. This loops through each character in the String being passed in. Each loop, the ascii value of that character is multiplied together. After looping through the full String, the value is returned as the index for the LinkedList array. This Hash Functor is quite a bit better than the Bad Hash Functor I created. The chances of multiple Strings of length 10 to create the same value after this equation is very low.

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 hash = 5381;
    int c = 1;
    int ch = item.length();
    while (c++ != ch) {
        hash = ((hash << 5) + hash) + item.charAt(c - 1); /* hash * 33 + c */
    }
    return Math.abs(hash);
}
```

My GoodHash Functor initializes three ints which will be used to calculate a return index for the array of LinkedLists. This hash function is a well known good function called djb2 by Dan Bernstein. The while loop is condition is quite similar to my MediocreHashFunctor, however the way in which the hash is returned is different. The algorithm within the while loop multiplies the hash by 33 with the binary left shift operation by 5. Then, the ascii value of the character is added. This function then returns the final hash value which will indicate the index within the array of LinkedLists. This is a great hash function because the return index that is returned will be very unique based on the formula provided. It will do a good job of distributing the hash values evenly across the range and avoid collisions.

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.

My experiment times the performance of the add methods for each of my three different hash functors, and calculates the number of collisions that occur in each hash functor. I run a loop to test this performance with varying sizes, from 2000 until 270,000, double at each iteration. As the size increases, I measure the performance of the add method and plot the data on a linear graph to compare results for each of the hash functors.

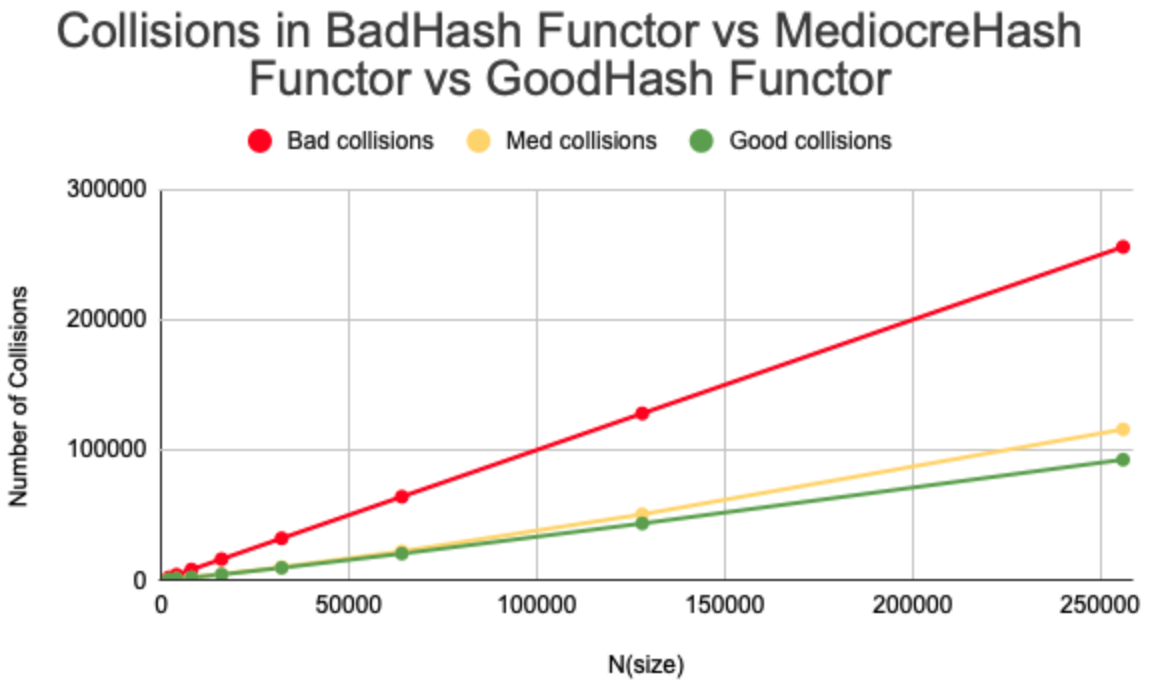
What is the cost of each of your three hash functions (in Big-O notation)? Note that the problem size (N) for your hash functions is the length of the String, and has nothing to do with the hash table itself. Did each of your hash functions perform as you expected (i.e., do they result in the expected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

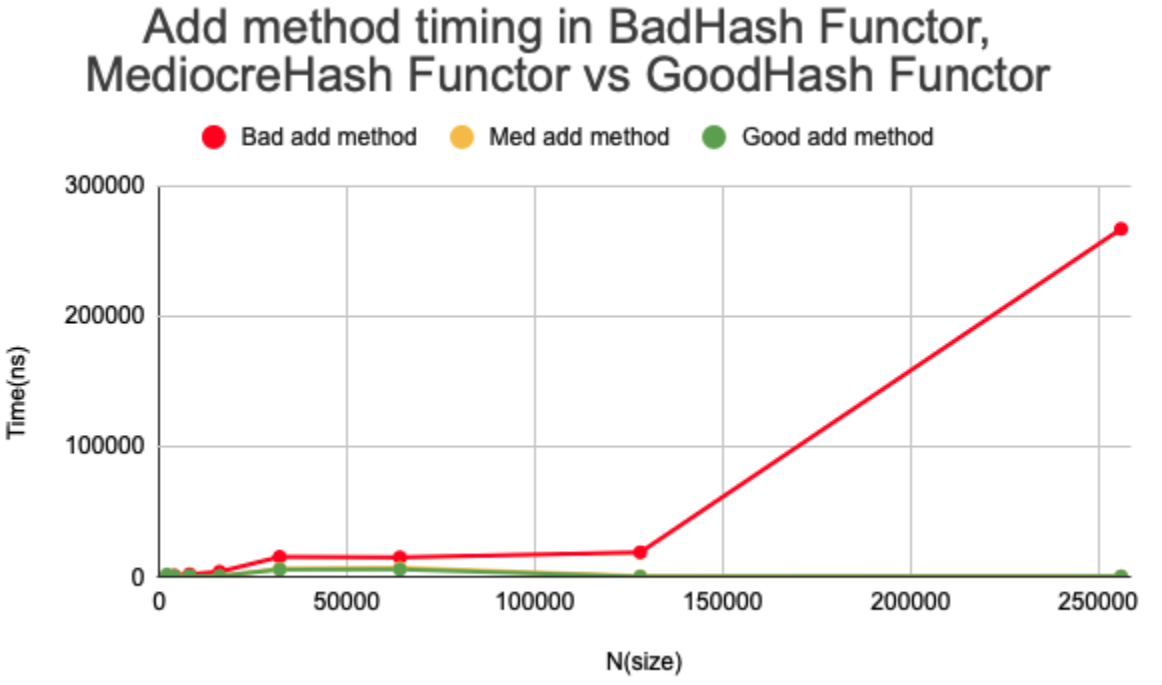
Each of my hash functions do perform as I expected.

The bad function add method performs much slower than my mediocre and good add methods. This is because the functor I wrote only allows for a small number of possible indexes in the array of LinkedLists to be filled (based on the first character in the given string). The data shows the time to greatly increase as the size of Strings to add increases. The primary reason the add method takes a long time later into the experiment is because it must call my contains method. As more Strings are added to the HashTable, the contains method continues to run slower as it must search through the LinkedList for the String to confirm it is not there before adding it. This add method begins as O(1) but can become O(LinkList size) in the worst case for this reason. The number of collisions according to this data is consistently and always N-62. Meaning, there are only 62 possible locations in the array that will be able to hold the String.

The mediocre add method performs as expected. It is consistently O(1) and operates very fast. Because I am timing in nanoseconds, the time calculated is near instant and can have some variability due to the computer's operation time. The number of collisions are also much lower when compared to the bad hash function as the way the index location is being calculated will result in a very unique index amongst the other random strings.

Similar to the mediocre, the good add method performs at O(1). The collisions are much lower than the bad hash function, and even lower than the mediocre hash function consistently by a noticeable margin within the line graph. When there are collisions, the big O time complexity can get to O(LinkedList size). However this is rare.


Collisions in BadHash Functor vs MediocreHash Functor vs GoodHash Functor
Bad collisions
Med collisions
Good collisions
Number of Collisions
300000
200000
100000
0
0
50000
100000
150000
200000
250000
N(size)


Add method timing in BadHash Functor, MediocreHash Functor vs GoodHash Functor
Bad add method
Med add method
Good add method
Time(ns)
300000
200000
100000
0
0
50000
100000
150000
200000
250000
N(size)