Analysis Document for Assignment07

Tailang (Terry) CAO   u1480633

1. BadHashFunctor

   The bad hash functor will return the same integer every time so that every string will have the same index to insure the maximum collision.

2. MedHashFunctor

   The mediocre hash functor will return the sum of the ASCII value of each of the characters in the string, decreasing the possibility of different strings having the same index in the hash table. However, as the sum of ASCII values of different characters can be equal (Eg. 10+20 = 15+15), collisions are still likely to happen.

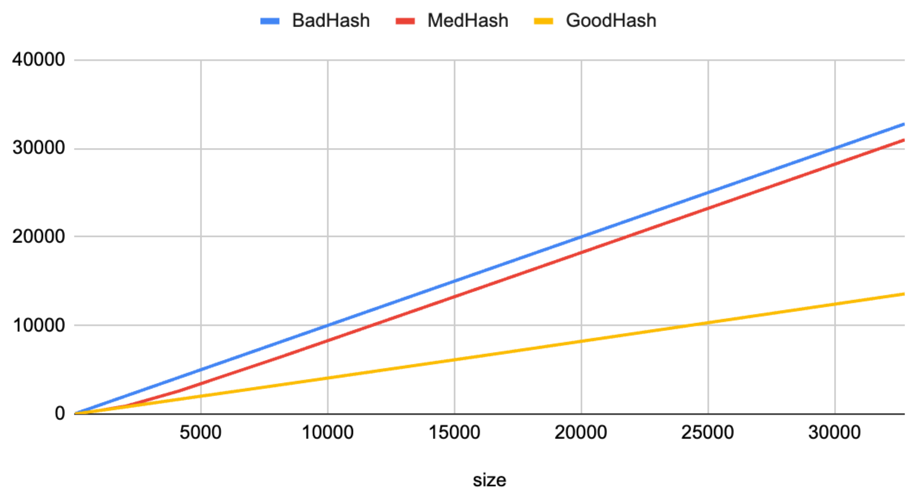
3. GoodHashFunctor

   The good hash functor uses a simple polynomial rolling hash algorithm to determine the index of each string. The hash value is updated by multiplying the current hash by a prime number (hereby 31) and adding the ASCII value of the next character in the string. This helps to produce a good distribution of hash values and minimizes collisions.

4.1 Experiment

(1) Collision Experiment

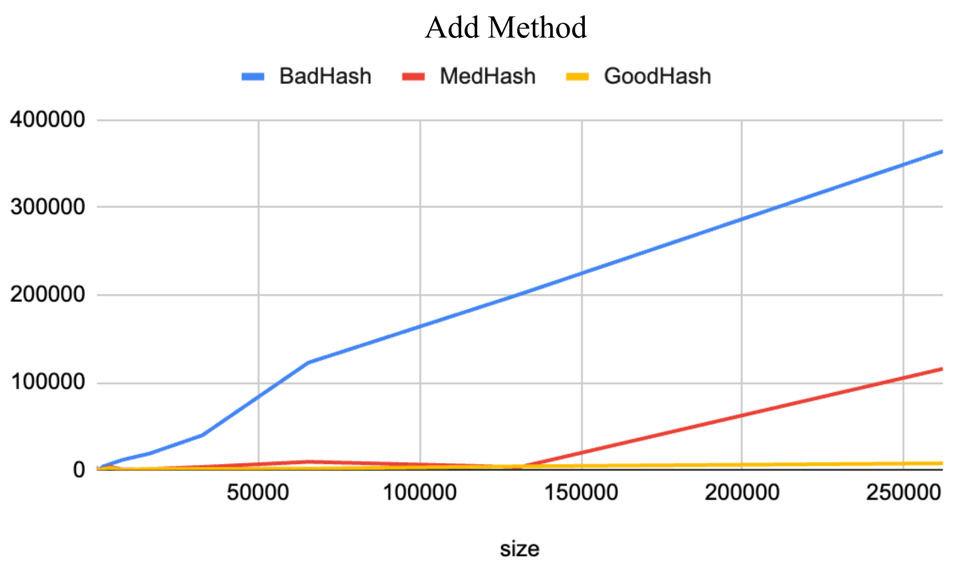
I created three hash table using the three functors I had and add the same number of randomly generated strings (using a generateRandomString method to generate) into each hash table. I then print out the recorded collision times (which is recorded in the add method).

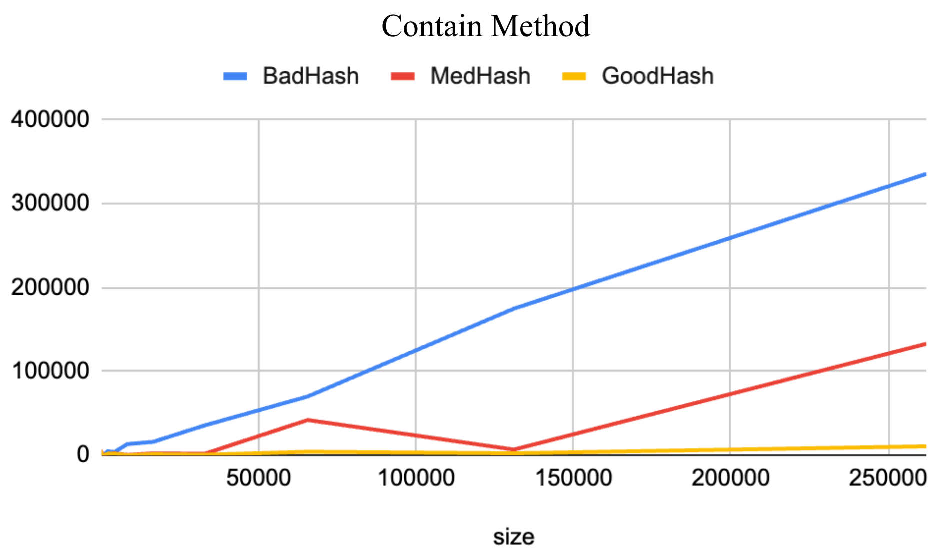


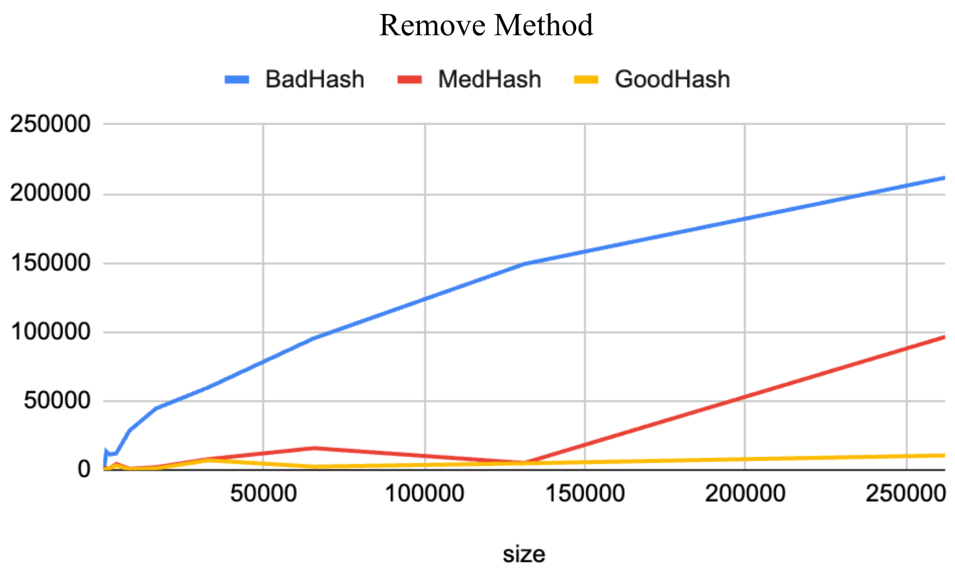
The size is determined to be from 2^5 to 2^15, taking the average of 10 iterations. As is shown in the plot, the collision occurred by using goodHashFunctor to create the hash table is significantly lower comparing with the other two functors, especially when the size of the table is getting larger.

(2) Runtime Experiment

I created three hash table with the corresponding functor first, then test the runtime of add, remove, contain method separately. A list named aL is used for storing the same number of randomly generated strings. For the add method, I had three timers to time the three functors respectively. For the contain and remove method, the table is completed beforehand, and three timers will test the method time respectively for 10 iterations and take the average time.


Add Method
BadHash
MedHash
GoodHash
400000
300000
200000
100000
0
50000
100000
150000
200000
250000
size


Contain Method
BadHash
MedHash
GoodHash
400000
300000
200000
100000
0
50000
100000
150000
200000
250000
size


Remove Method
BadHash
MedHash
GoodHash
250000
200000
150000
100000
50000
0
50000
100000
150000
200000
250000
size

As is seen in the plot, in all three plots, the method from the good hash functor has a run time significantly lower than the methods from the other two functors. For the add method, the three methods reached a peak and the runtime of the mediocre method and the good method starts decreasing after the peak while the bad method remains around the peak.

For the contain and remove method, the plots look similar. While the runtime of the bad method keeps increasing, the runtime of the good method doesn't increase significantly and the mediocre method is increasing slower than the bad method as the size of the table grows.

4.2 Time Complexity

(1) BadHashFunctor

- Add method: O(N)
- Remove method: O(N)
- Contains method: O(N)

(2) MedHashFunctor

- Add method: O(1) when size within 20000, O(N) when size exceeding 20000
- Remove method: O(1) when size within 20000, O(N) when size exceeding 20000
- Contains method: O(1) when size within 20000, O(N) when size exceeding 20000

(3) GoodHashFunctor

- Add method: O(1)

- Remove method: O(1)
- Contains method: O(1)

5. Cost of hash functors

(1) Bad: O(1)

(2) Med: O(N)

(3) Good: O(N)

The expectation of collision number and runtime meet the experiment result.