Assignment 7 - Analysis

Lindsay Haslam

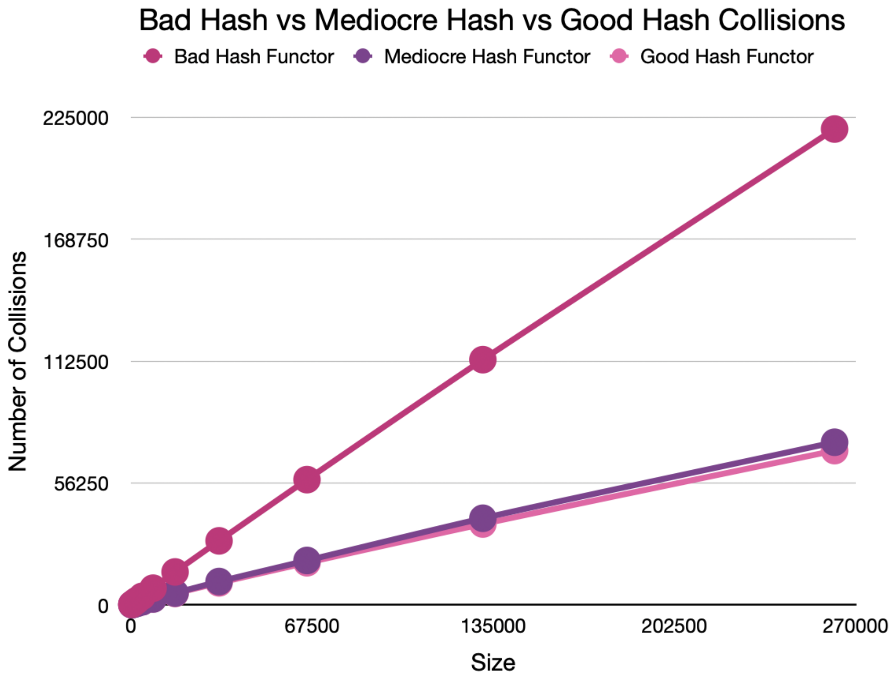
1) In the BadHashFunctor, the hash method will always return 0. This means that no matter the input size, the hash code will always be zero. This will lead to poor distribution of elements across the array, so they will always be put in the same bucket. This will cause both collisions and a degradation in the performance.

2) For MediocreHashFunctor, I initialize the hash code to 7. I then use a for loop to iterate over each character within the string. I then multiply hash by 31 to get a good distribution of hash code to avoid collisions. The ASCII value of the character in the string is also added to the multiplication result. And then I return the hashcode of that result. I expected this to perform moderately because we are distributing elements much better than simply returning 0, but it isn't as optimal.

3) In GoodHashFunctor, the performance is improved by bit shifting to left by seven, and then multiplying that by a large prime number (5381). These are better distributed than the way it was done in my MediocreHashFunctor. As I said previously, this is the best hash functor because there is greater distribution of elements.

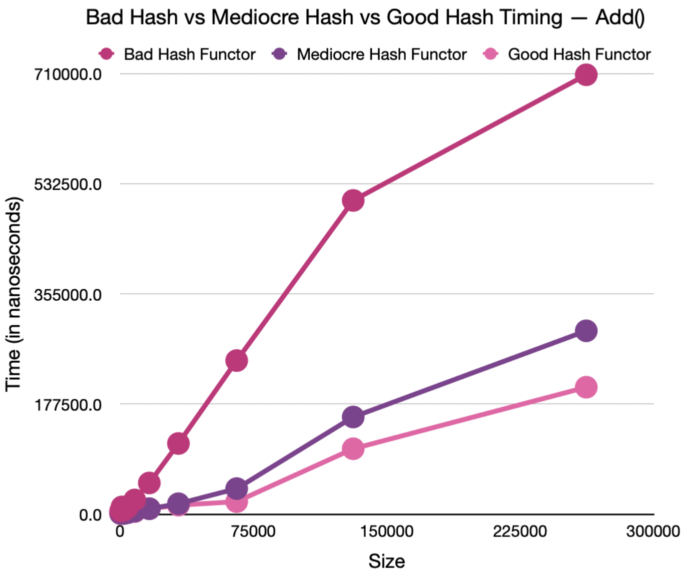
4) *See the charts below.*

5) In general, the Big O notation for BadHashFunctor resembled that of O(1), going at a rate that is consistent with the input size. But as for my Mediocre Hash and Good Hash, they resemble O(N) a little bit more. I could have made my Mediocre Hash distribute elements a little bit less so that it was more dramatic and in between Bad and Good, but in general they performed at a better rate. I was expecting O(1) for a Bad Hash because I was returning 0 and not distributing the elements.



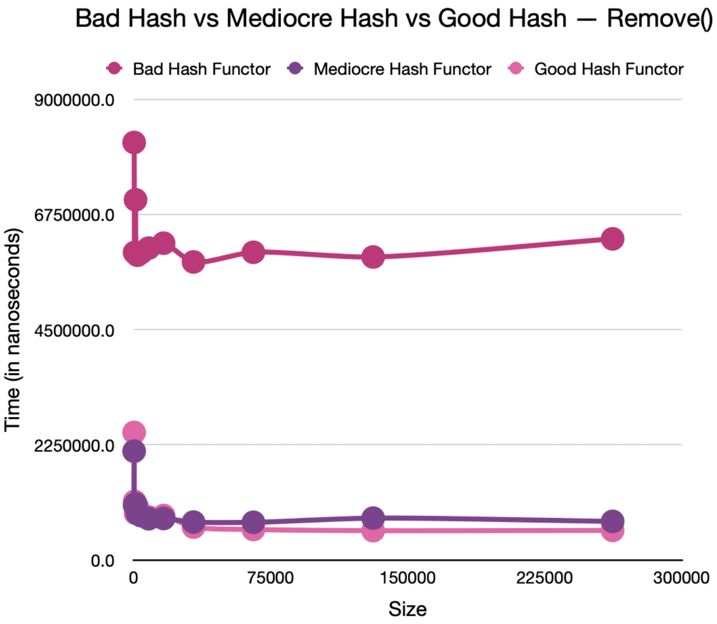
FuncCollisionAddTimeFinal

256	5250.0	1334.0	875.0
512	6333.0	917.0	834.0
1024	11792.0	1625.0	1709.0
2048	6417.0	1291.0	1292.0
4096	11208.0	2459.0	2275
8192	23000.0	5084.0	4275
16384	50375.0	8791.0	8959.0
32768	114042.0	16917.0	14583.0
65536	247666.0	41292	20292
131072	505833.0	156709.0	105542
262144	708417.0	295666	204717



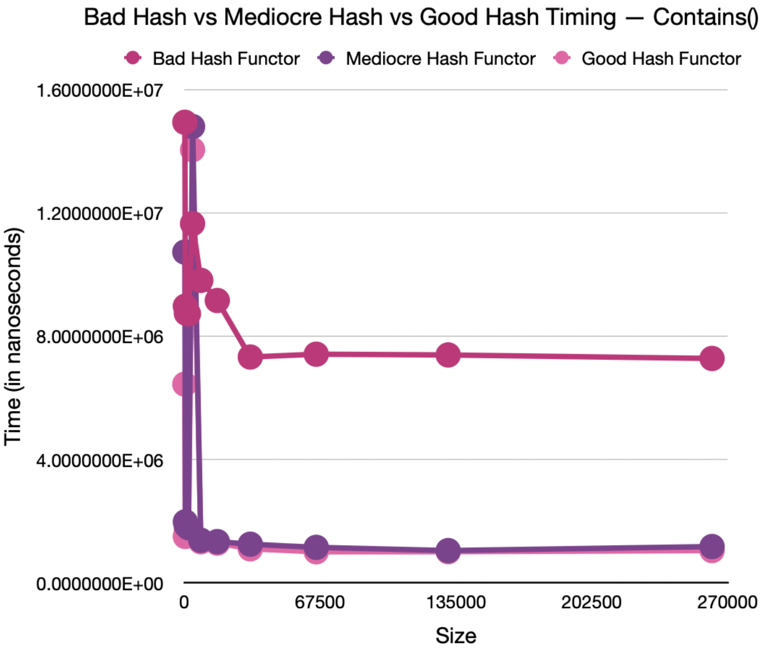
DictRemoveCollisionTimeFINAL

256	8159166.0	2124542.0	2490792.0
512	6009000.0	1065958.0	1143375.0
1024	7038584.0	1094041	911291
2048	5954958.0	905000.0	1024167.0
4096	6006334.0	863334.0	904958.0
8192	6094417.0	804292.0	837625.0
16384	6190667.0	809125.0	866041.0
32768	5824042.0	738250	640416
65536	6012459.0	733917	593250
131072	5920625.0	815375	570667
262144	6275166.0	751833	575500.0



DictContainsCollisionTimeFINAL

256	1.4940833E+07	1.0729875E+07	6446417
512	8974250.0	1980458	1505625
1024	8743125.0	1852917	1723792
2048	8728375.0	1790625	1597541
4096	1.1653583E+07	14801208	14052667
8192	9813167.0	1382958	1334209
16384	9159250.0	1338167.0	1290292.0
32768	7326333.0	1253875	1103833
65536	7417458.0	1152208.0	1008000
131072	7393167.0	1047500.0	1010958
262144	7279083.0	1178292	1048292


Bad Hash vs Mediocre Hash vs Good Hash Timing — Contains()
● Bad Hash Functor
● Mediocre Hash Functor
● Good Hash Functor
1.6000000E+07
1.2000000E+07
8.0000000E+06
4.0000000E+06
0.0000000E+00
Time (in nanoseconds)
Size