Analysis Assignment 07

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)

My BadHashFunctor generates a hash code by summing up the character values of the characters in the input string. For example, the hash code for “hello” would be 532 (‘h’ 104 + ‘e’ 101 + ‘l’ 108 + ‘l’ 108 + ‘o’ 111 = 532). This is a bad function because it’s highly probable that different strings, with different characters, will produce the same hash code, and when two keys have the same hash code that means more collisions. High collision rate leads to non-uniform distribution making it more likely for hash table degeneration (all the keys get mapped to the same bucket, and we have a linked list of n(size of the hash table) size from one single bucket). This is the worst case scenario for a hashtable, and would cause operations like contains, add, and remove O(N).

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).

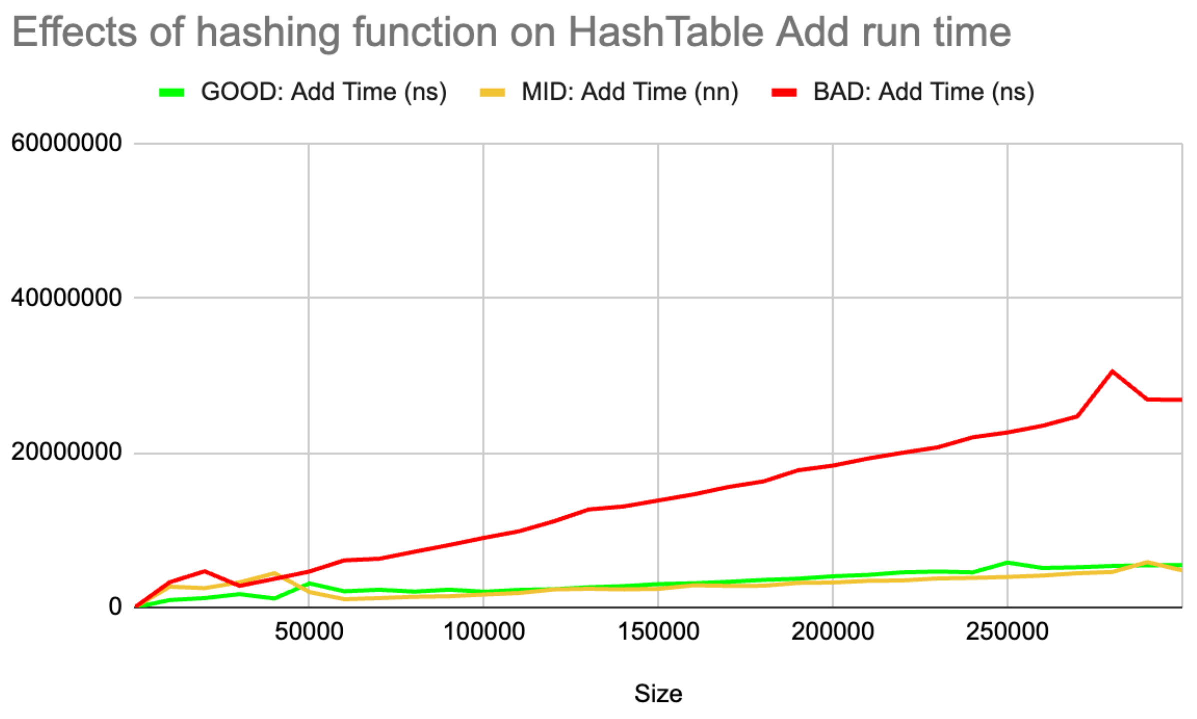
My ModerateHashFunctor generates a hash code by iterating through each character of the input string, then for each character in the string, it updates the hash value using the formula (hash * 32) + string.charAt(i). Multiplying by 32, an even number, results in an inherently less even distribution of hash codes. This can cause patterns in the resulting hash values, potentially leading to more collisions.While it can still produce patterns in hash codes due to its simplicity, the multiplication factor introduces some complexity that might reduce the amounts of keys producing the same hash code compared to the straightforward addition in the "BadHashFunctor."

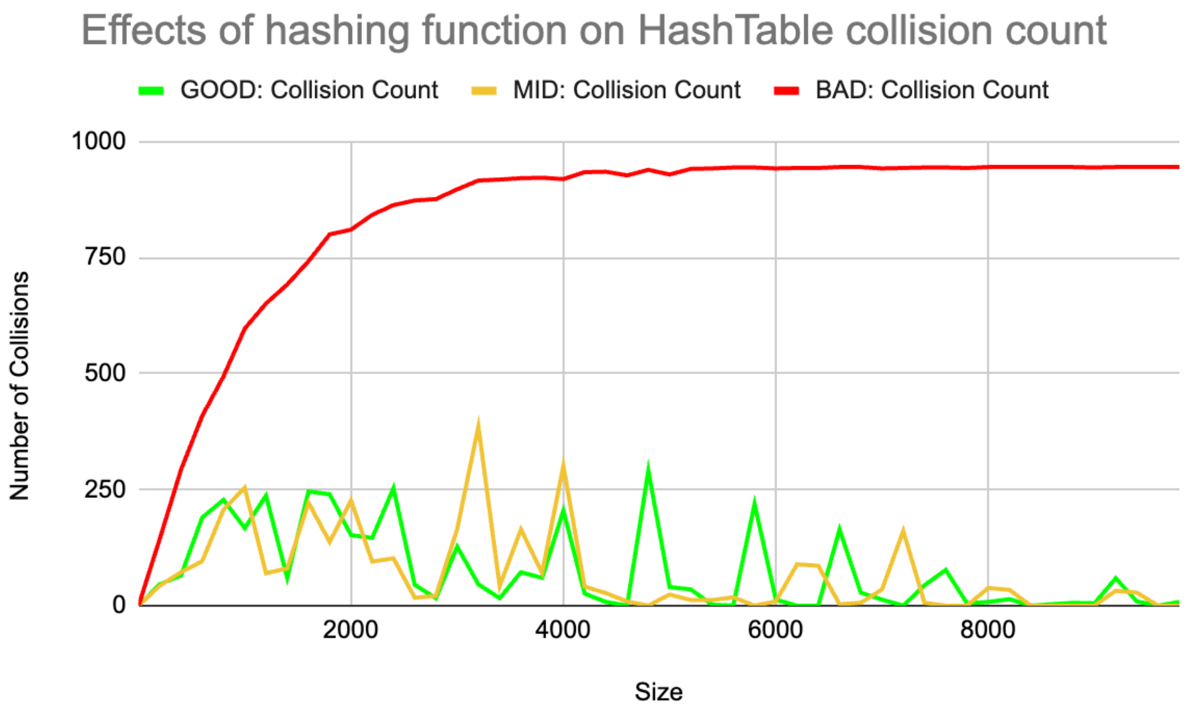
Explain the hashing function you used for oodHashFunctor. Be sure to discuss why you expected it to perform well (i.e., result in few or no collisions).

My GoodHashFunctor generates a hash code by iteratively multiplying the current hash by 31 and adding the Unicode value of each character. Multiplying by 31, an odd prime number, helps to create more diversity in the resulting hash code and introduces a bit-shift subtraction in the bit representation which aids in scattering the values across the hash space. This provide a non-linear manipulation of the hash, enhancing its ability to generate distinct hash codes for different strings which results in more even distribution across the available buckets in the hash table and little to possibly even no collisions.

Design and conduct an experiment to assess the quality and efficiency of each of your three hash functions.

My experiment is designed to evaluate the efficiency and collision occurrences of three distinct hash functions—Good, Mediocre, and Bad—within a chaining hash table structure. To gauge their performance, I vary table sizes from 1 to 10,000 with intervals of 200, ensuring a diverse range of load factors for evaluation. For each run, I generate test data consisting of randomly created strings with varying lengths and characters. To measure collision counts accurately, I iterate through the hash table's buckets, assessing non-empty buckets for collisions. Moreover, I track the execution time for addition, removal, and search operations using System.nanoTime() for precise time measurements. This approach enables a comparative analysis of hash function performance across different table sizes, aiming to uncover how various hash functions impact collision occurrences and operational efficiency within a chaining hash table structure.





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)?

The cost of the good and moderate hash functions is O(1) and the cost of the bad hash function is O(N). The results of this experiment are as expected. A good hash function generally distributes elements evenly across the hash table, minimizing collisions. As long as there are no significant collisions, its processes are in constant time. A mediocre hash function might have slightly more collisions but it still tends to keep the number of collisions manageable for most practical cases, resulting in nearly constant time complexity for insertion. In the case of a bad hash function, the distribution of elements into hash table slots is highly uneven, causing a large number of elements to collide and form longer chains. As the number of collisions increases, the time taken to run its processes grows linearly with the length of the collision chain, resulting in O(N) complexity for insertion due to the linear traversal of these chains.