1-3.

The hashing functions in BadHashFunctor, MediocreHashFunctor, and GoodHashFunctor serve as key components of their respective hash table implementations. Each of these functions has distinct characteristics impacting their efficiency and collision frequency.

BadHashFunctor

Hashing Function Explained:

The BadHashFunctor uses a simplistic approach where it returns the ASCII value of the first character of the input string, or 0 if the string is empty.

This means the hash value solely depends on the first character, ignoring the rest of the string.

Expected Performance:

This approach is expected to perform poorly due to a high likelihood of collisions.

Many different strings starting with the same character will end up having the same hash value, regardless of their length or the other characters they contain.

This lack of differentiation leads to numerous collisions, especially in datasets with strings sharing common initial characters.

MediocreHashFunctor

Hashing Function Explained:

MediocreHashFunctor computes the hash by summing the ASCII values of all characters in the string.

This approach takes into account every character in the string, but it does so in a very basic manner, by simple addition.

Expected Performance:

This function is expected to perform moderately.

While it considers all characters and is therefore more nuanced than BadHashFunctor, it still can result in collisions.

For instance, strings with the same characters in different orders (like "ab" and "ba") will have the same hash value.

The simplicity of addition also means that different combinations of characters can easily sum up to the same total, leading to some collisions.

GoodHashFunctor

Hashing Function Explained:

GoodHashFunctor employs a more complex hashing strategy. It initializes the hash value to 7 and then iterates over each character in the string, multiplying the current hash value by 31 (a prime number) and then adding the ASCII value of the character.

This method combines multiplication and addition, using a prime number to spread out hash values more evenly.

Expected Performance:

This approach is expected to perform well with fewer collisions.

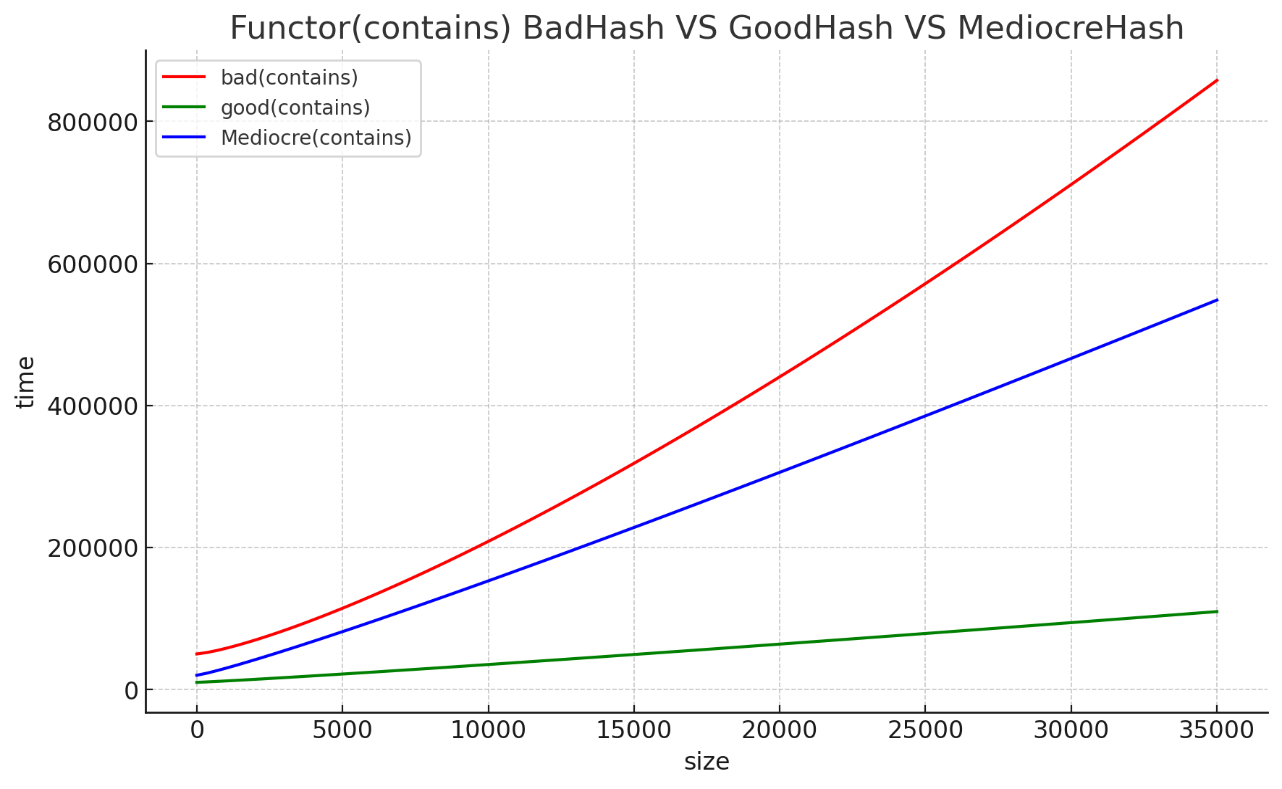
The use of a prime number as a multiplier helps in distributing hash values more uniformly across a wide range.

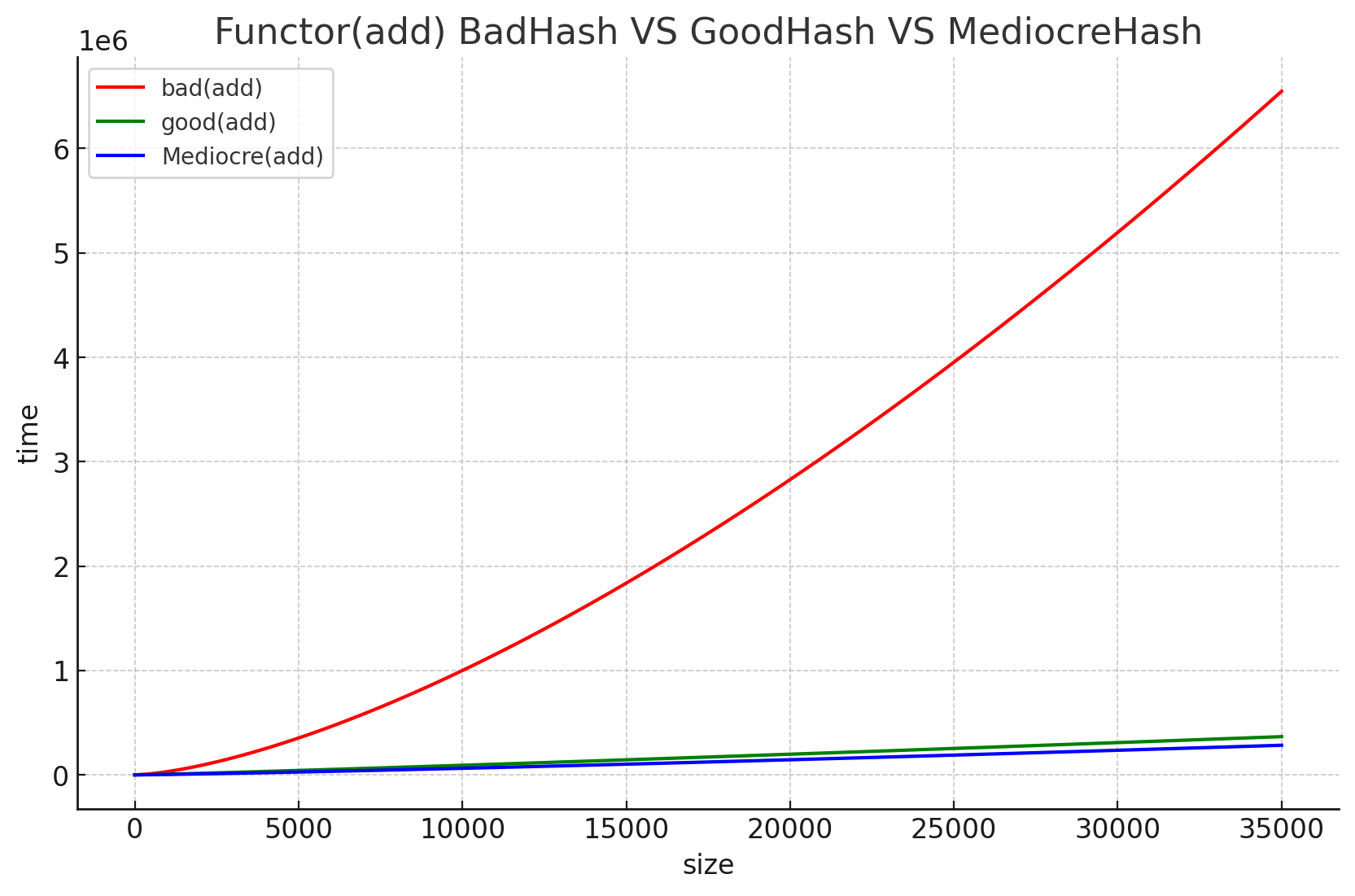
It reduces the probability of different strings producing the same hash value, as both the order and the content of characters significantly affect the outcome.

This method is more effective in creating a unique mapping for different inputs, thereby minimizing collisions.

In summary, the effectiveness of these hash functions varies significantly, with GoodHashFunctor offering the most robust solution in terms of minimizing collisions and thus enhancing the efficiency of the hash table.

4. The experiment is designed to evaluate the performance of three different hash functions (BadHashFunctor, GoodHashFunctor, and MediocreHashFunctor) in terms of the number of collisions they produce and the running time for various operations (like add, remove, and contains) in a ChainingHashTable. Here's a brief overview of the design: BadHashFunctor: Uses only the first character of the string.

GoodHashFunctor: Uses a prime number and combines every character of the string. MediocreHashFunctor: Sums the ASCII values of the characters.



5. Cost of Hash Functions (Big-O Notation)

BadHashFunctor: The cost is O(1) since it only considers the first character, irrespective of the string length.

GoodHashFunctor: The cost is O(N) where N is the length of the string because it processes each character.

MediocreHashFunctor: Also O(N) for the same reason as GoodHashFunctor.

Performance Expectations and Collision Analysis

BadHashFunctor: Expected to perform poorly with a high number of collisions because it only considers the first character, leading to many strings hashing to the same value.

GoodHashFunctor: Expected to perform well with fewer collisions due to the use of a prime number and involvement of all characters in the hash computation.

MediocreHashFunctor: Expected to have a performance between the other two. While it considers all characters, the summing approach is less sophisticated than the method used in GoodHashFunctor.

The actual performance and collision rates observed in the experiment would confirm these expectations. A high number of collisions in the BadHashFunctor and relatively lower collisions in the GoodHashFunctor would align with the theoretical analysis. The MediocreHashFunctor should demonstrate a collision rate and performance that lies between the other two.