

# CZ2001 Algorithm

Example Class 2 Report

SE1 | Group 2

**Members:**

U1620986A Lee Yi Zhuo

U1721978B Ngo Jun Hao Jason

U1722308D Thomas Stephen Felix

U1722841L Tan Yong Heng Kenrick

N1804770J Yung-Hsueh Lee

Background

In the new age of consumerism, where the sale of goods is incumbent on the speed and convenience of shopping on e-commerce sites, we are witnessing the exponential increase in the variety and stock of products that are available online.

Problem

Correspondingly, there is a pressing need for an appropriate implementation of the inventory database for e-commerce sites to keep the items and its relevant information neatly stored and easily accessible for employees.

Data Set

A collection of 20,000 products listed on Flipkart

Solution

Constrain: Use only closed address hashing

We have designed a program in Java, whose function is to store the product details in hash tables through the implementation of a hashing function.

By using this method, employees would quickly be able to retrieve the exact specified product by searching up for the key in the hash table, thus solving the accessibility problem.

Hashing Algorithm and Implementation

Analysis: Compare between hash tables of prime and non-prime sizes

We have chosen the folding method as our manner of implementation:

1. Product serial number is converted in String type
2. Each character in the String is multiplied by its position index
3. Each individual values are added up to form the key
4. Finally the key is put through a hash function to obtain the value

f(key) = key mod hashTableSize, where hashTableSize [20,500]

Testing

In each test scenario, we vary the the load factor in steps of 20, beginning from 20 and ending at 60.

In each instance when:

Load factor = 20

Hash table size = 20

Size of sample set = 400

Therefore, we choose 400 random items from a collection of 20,000 product serial numbers to form a sample set, and store it into the hash table.

We then proceed to select 1 random item from the sample set of 400 random items to test for successful searches.

This process is done 100 times, to capture the average number of comparison and search time.

In the next instance, the hash table size is increased and the random selection process is repeated again. This increment of the hash table size is done up to 500.

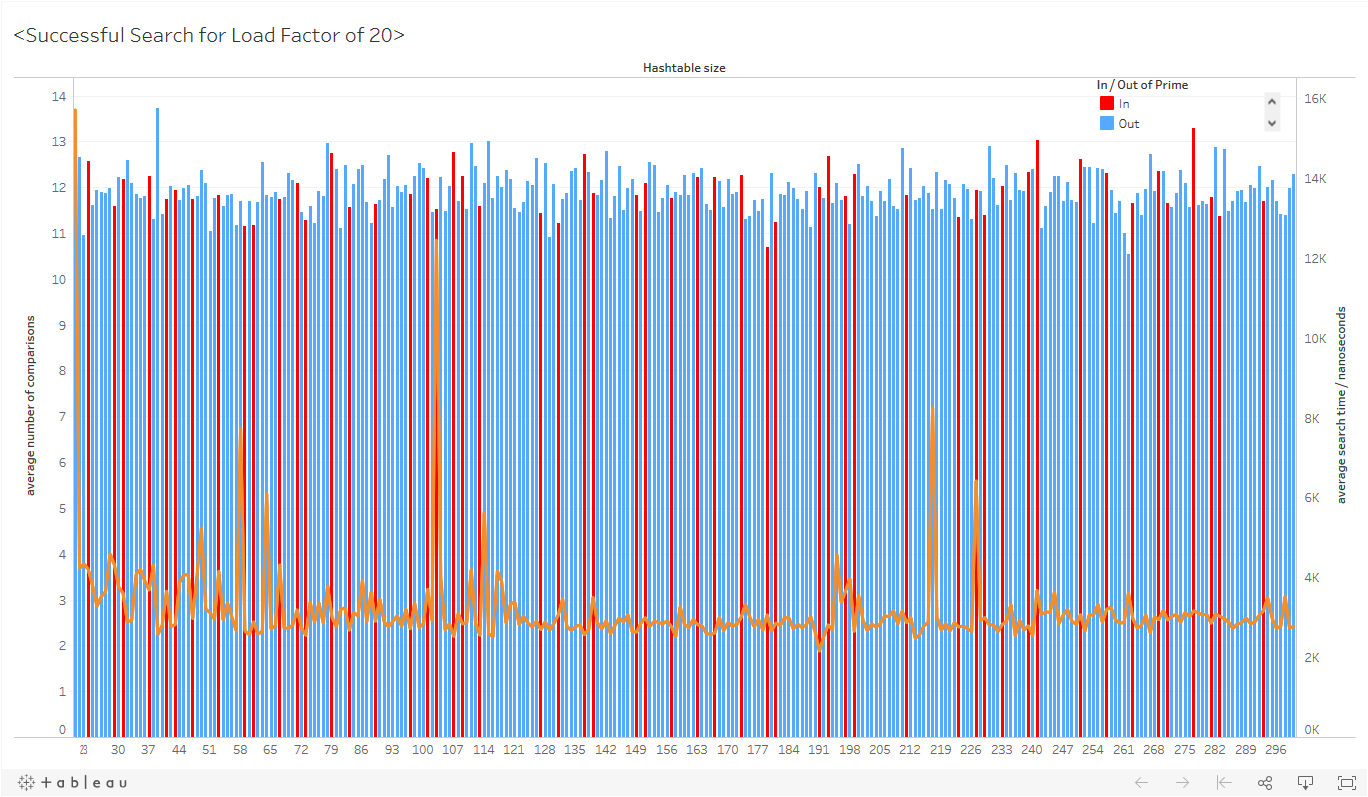
Following which, the load factor is then increased to 40, then 60 in the subsequent test scenario.

Statistics

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Load Factor (n/h) | Average Number of Comparisons | | | |
| Successful Lookup | | Unsuccessful Lookup | |
| Prime | Non-prime | Prime | Non-prime |
| 20 | 11.88 | 11.88 | 11.90 | 11.93 |
| 40 | 23.97 | 23.88 | 23.91 | 23.90 |
| 60 | 36.02 | 36 | 35.89 | 35.94 |

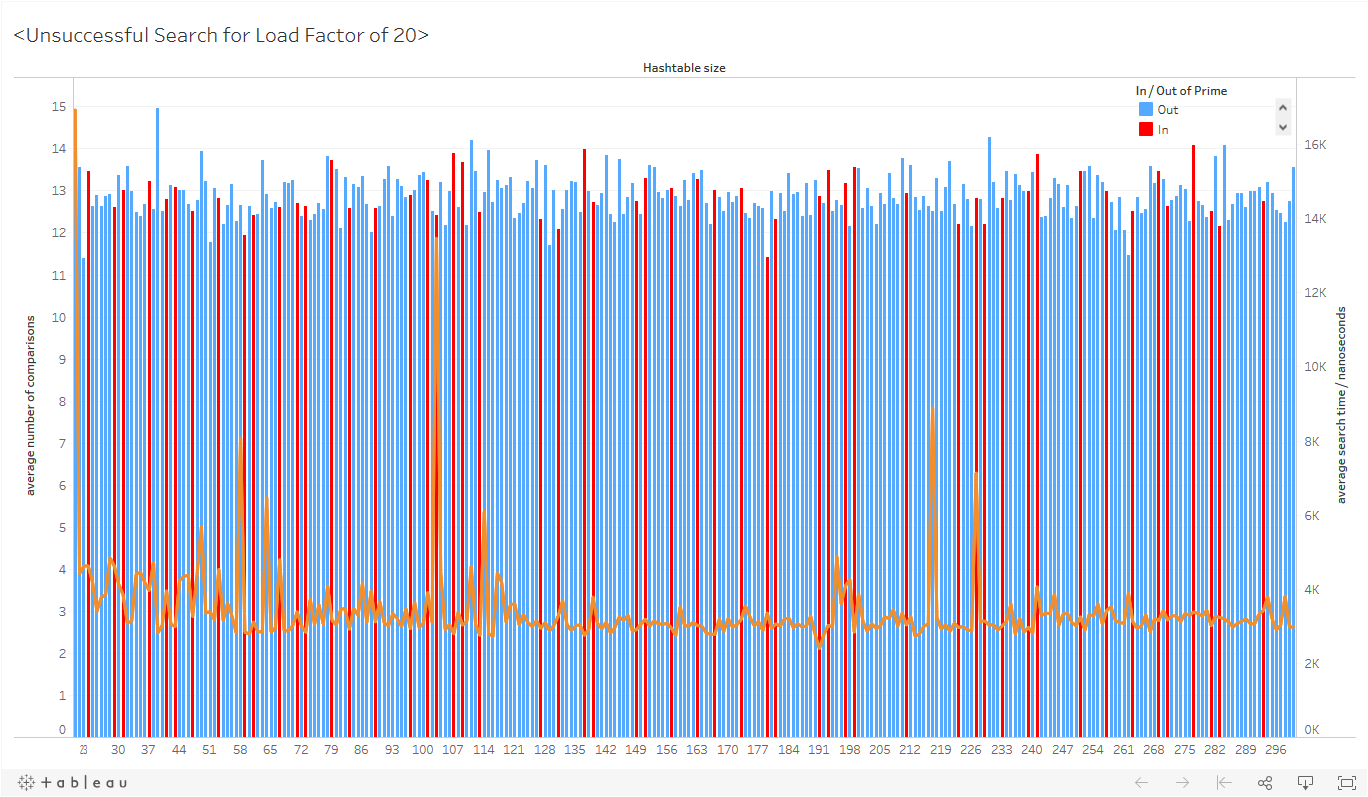
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Load Factor (n/h) | Average search time / nanoseconds | | | |
| Successful Lookup | | Unsuccessful Lookup | |
| Prime | Non-prime | Prime | Non-prime |
| 20 | 2890 | 2858 | 2697 | 2649 |
| 40 | 3237 | 3215 | 3137 | 3126 |
| 60 | 3837 | 3764 | 3700 | 3777 |

Successful Search for Load Factor of 20



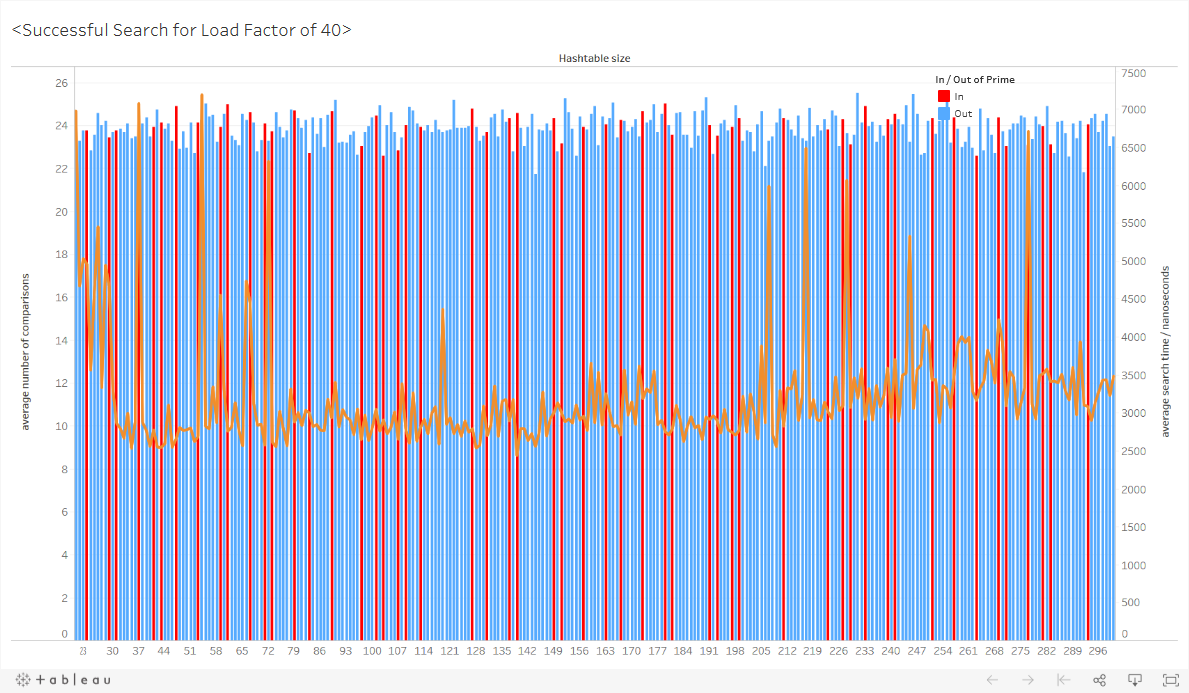
[click here to explore the interactive graph](https://public.tableau.com/profile/lee.yi.zhuo#!/vizhome/SuccessfulSearchforLoadFactorof20/Dashboard1)

Unsuccessful Search for Load Factor of 20



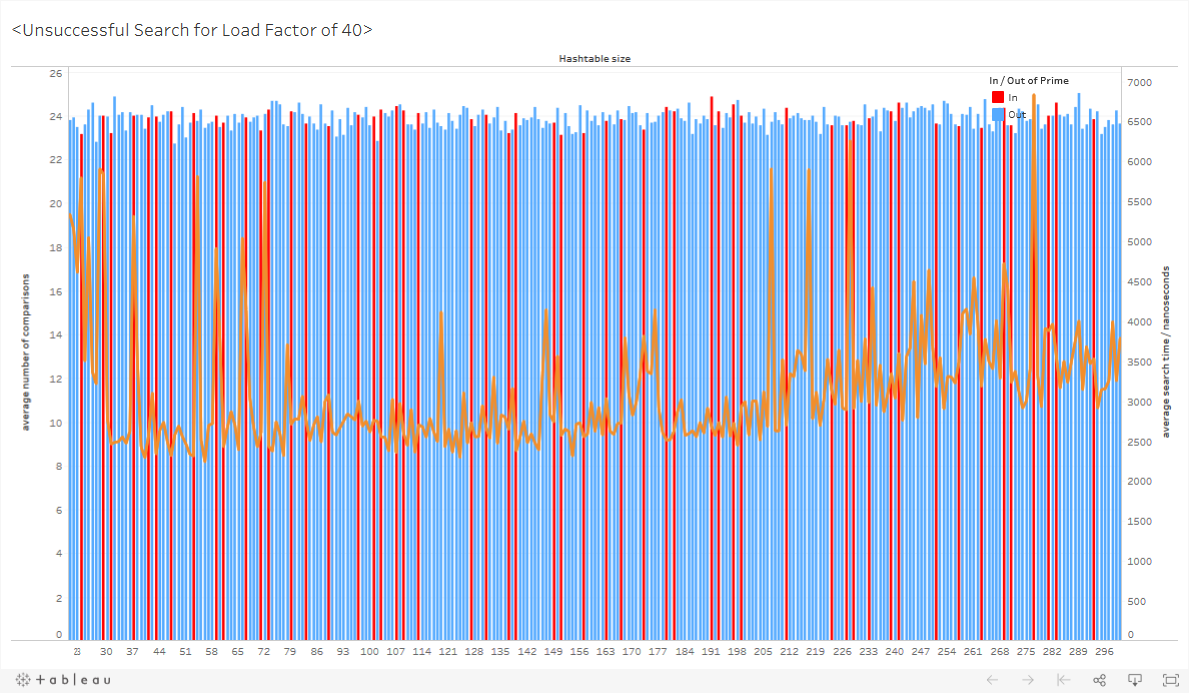
[click here to explore the interactive graph](https://public.tableau.com/profile/lee.yi.zhuo#!/vizhome/UnsuccessfulSearchforLoadFactorof20/Dashboard1?publish=yes)

Successful Search for Load Factor of 40



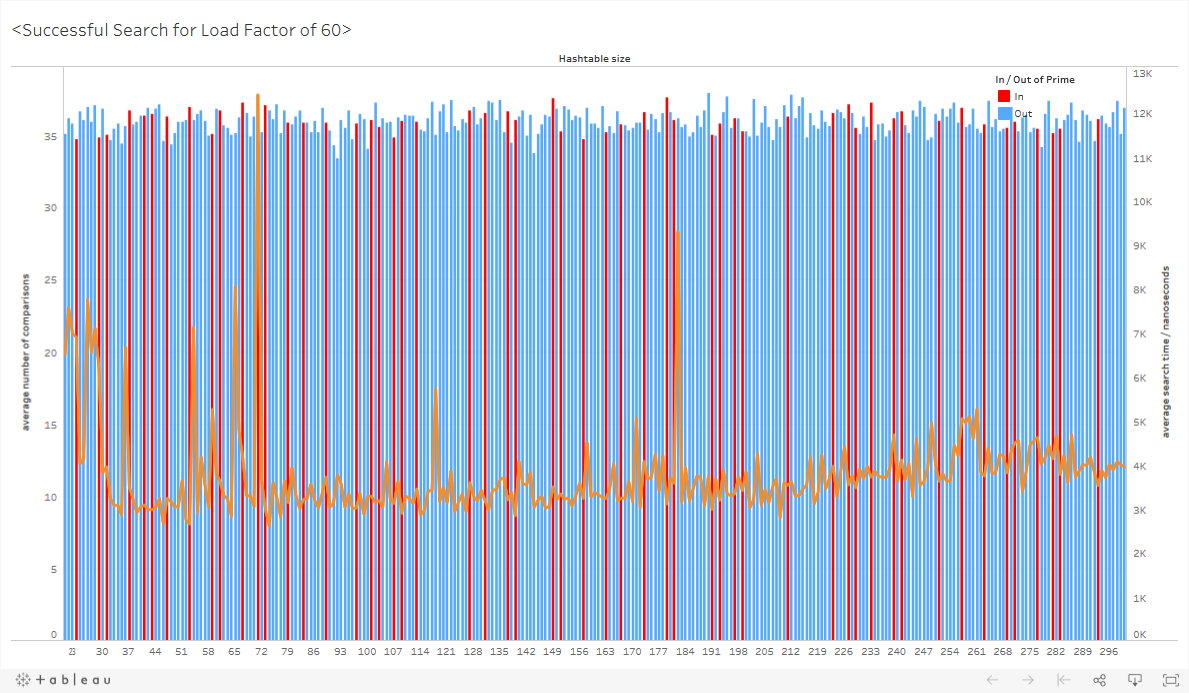
[click here to explore the interactive graph](https://public.tableau.com/profile/lee.yi.zhuo#!/vizhome/SuccessfulSearchforLoadFactorof40/Dashboard1?publish=yes)

Unsuccessful Search for Load Factor of 40



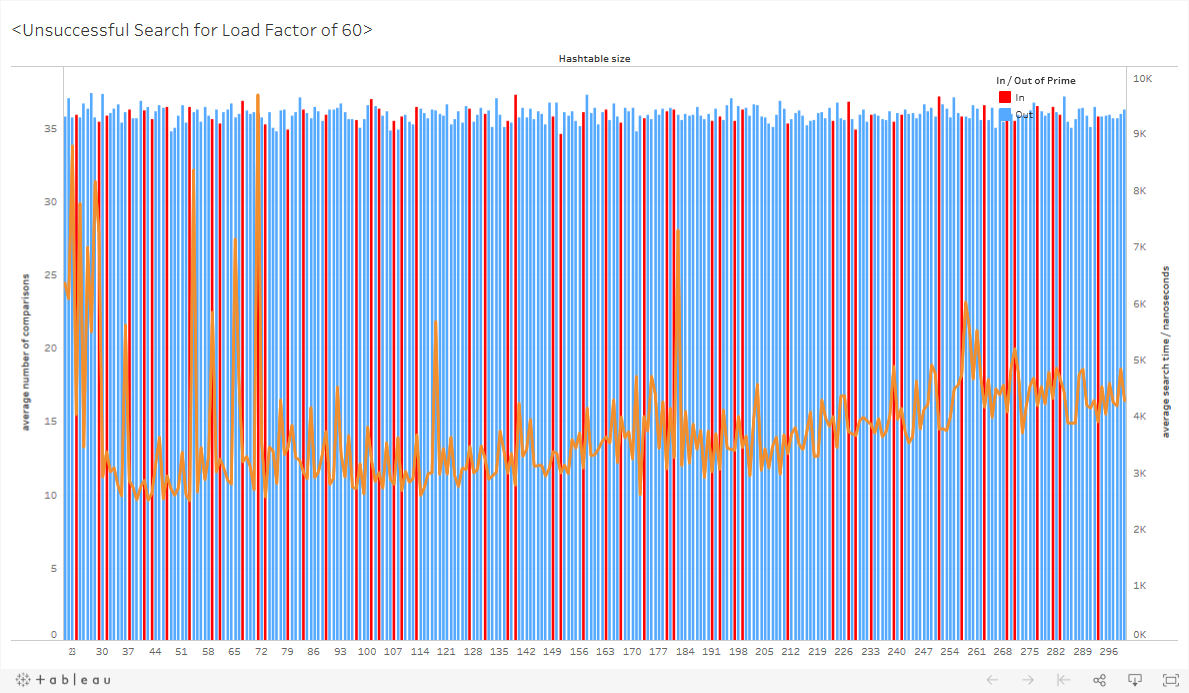
[click here to explore the interactive graph](https://public.tableau.com/profile/lee.yi.zhuo#!/vizhome/UnsuccessfulSearchforLoadFactorof40/Dashboard1?publish=yes)

Successful Search for Load Factor of 60



[click here to explore the interactive graph](https://public.tableau.com/profile/lee.yi.zhuo#!/vizhome/SuccessfulSearchforLoadFactorof60/Dashboard1?publish=yes)

Unsuccessful Search for Load Factor of 60



[click here to explore the interactive graph](https://public.tableau.com/profile/lee.yi.zhuo#!/vizhome/UnsuccessfulSearchforLoadFactorof60/Dashboard1?publish=yes)

Time Complexity

Let n be number of keys, and h be number of slots(linked lists) in the hash table

For closed address hashing, the worst case scenario occurs when the keys are all placed in the same slot in the hash table, forming one linked list of length n. An unsuccessful search will iterate through the entire linked list to do n number of comparisons, resulting in a time complexity of θ(n). For a successful search, assuming an equal probability of a key being stored anywhere in the linked list, the number of comparisons required to find a key on average is given by (sum of time taken to reach each node)/n, which will result in a time complexity of θ(n).

The average case occurs when any key has equal probability of being hashed into any of the slots in the hash table. An unsuccessful search still involves iterating through an entire linked list. However, since the keys are assumed to be evenly distributed across the hash table, the linked lists would have a length of n/h on average which means doing n/h comparisons on average, resulting in a time complexity of θ(n/h).

For a successful search, the key would be found 1 node after the previous node in the linked list where it was inserted. If the key to be found is the ith item to be inserted, then the length of the linked list from head to that previous node is given by (i-1)/h. Since, the key is 1 node after that previous node, its length is 1 node longer, so the number of comparisons required to reach it is 1+(i-1)/h.

The average number of comparisons for n number of keys can be obtained by summing up the number of comparisons for each key, 1+(i-1)/h, and dividing the sum by the number of keys. This results in a time complexity of θ(1+n/h). If n is proportional to h, the time complexity is in fact O(1).

For this task, we varied the number of keys to be stored in the hash table based on the hash table size and predetermined load factors, so n is always proportional to h. For hash tables of prime sizes, regardless of the distribution of key values, the keys should be evenly distributed over the table, so searching for a key tends to be an average case. However, for hash table of non-prime sizes, if the distribution of key values is not random, there would to be more collision, resulting in longer linked lists and more worst cases.

Therefore, as long as n is proportional to h, searching in a hash table of prime size tends to have a time complexity of O(1), whereas searching in a hash table of non-prime size tends to have a time complexity of O(n).

Conclusion

For an implementation of an efficient hash table, collisions should be minimized as much as possible. Using primes for hash tables is a good idea because it minimizes clustering in the hash table. It is also the most convenient for growing a hash table in the face of expanding data.

However, the test scenarios in this example case have shown that hash table size of prime numbers have returned mixed results. This could be largely attributed to the fact that the selection of items have been made wholly random, when in reality, the distribution of data tends to exhibit a pattern which would lead to clustering and collision in the hash table.

A blind understanding of the theory of using hash tables of prime sizes may also lead one to conclude that the larger the value of the prime number, the better the hash table would perform.

However, the results of the experiment have once again reminded us of the imperative need to apply the theory in context. For instance, with a collection of 20,000 product serial numbers, it would be counter-productive to use 20,011 (which is also a prime number) as the size of the hash table, as much as it is inapplicable.

Additional Comments

The function searchProduct() has a boolean parameter that controls whether to print the starting time, ending time, execution time, as well as whether a product’s key can be found in a hash table. Normally, these print statements are unnecessary in a searching function since its primary purpose is to search. However, we find the without these print statements, the execution times obtained in the result csv files generated are mostly 0: the computer searches through the hash tables so fast that it seems to take no time to search, even when using a measurement scale of nanoseconds. As such, we needed these print statements to slow down the search function significantly enough for us to capture the results.

For the case of unsuccessful searches, we generate random strings of 32 characters (to match the format of the products’ serial numbers), of which each character has an equal probability of an alphanumeric value (1-9, a-z, A-Z), to be used as the target keys to search for. For the search to be unsuccessful, these target keys generated have to not match any of the keys in the hash table. The random string generator function does not account for the existing keys stored in the hash table, so there is a possibility that a target key generated end up being the same as a key stored in the hash table. However, we deemed this possibility to be insignificant, as there are (10+26+26)^32 = 62^32 combinations of strings that can be generated, of which only 20 000 would already exist in the data set. Even if keys of the entire data set are stored in the hash table, the probability of getting 2 strings that match is 20000/62^32 = 8.8e-54, which is a very, very small probability. We ran the searches multiple times, for various load factors, for hash tables of various hash size, so even if we have one anomalous data, the rest of the data will average the error out.