An In-Depth Investigation of Hash Functions and Collision Handling Methods in Hash Tables

---

Mingchung Xia

June 18, 2021

Table of Contents

Abstract

This paper seeks to investigate the efficiencies of hash tables influenced by varying parameters with a focus on collision handling methods. An investigation was carried forward by reason that, despite a hash table's profound efficiency in terms of data structures for data search and mapping, it is evident that inappropriate conditions such as a poorly chosen collision handling method, hash function, or load factor for varying extents of weighted data distributions would greatly affect a hash table's performance. Subsequently, a conduction of a Python 3 program ran several trials to evaluate the insertion and retrieval of hash tables, adjusted based on nine degrees of weighted data distribution, four varied collision handling methods, two hash functions, at six different load factors. Data was experimentally retrieved by programming these differently implemented hash tables and graphically analyzed. Consequently, no collision handling method was evaluated to be the best in general, as each possessed their own flaws under certain parameters while other methods could excel. Hence, the data would give insight to optimizing the potential of a hash table's efficiency when correctly implemented under these known conditions.

1. Introduction

In modern age society, the study of computer science has served a key role in the foundations of the past, current, and future technological advancements. One of many contrivances present in computer design is the problem of data storage. Commonly, these are resolved by forms of data structures, which refer to a spectrum of applications for which a computer is able to store, allocate, and organize data. Hash tables are one of many forms of data structures which utilize associative arrays that seek to conjoin pairs of keys and values by hash functions. A hash table's distinctly fast operation time and usefulness makes it one of the essential data structures present in modern programming. However, the hash function present in hash tables fundamentally flaw the data structure as a whole, because the problem of collisions will indefinitely occur while operating; collisions can be resolved by various means of collision handling, and thus it is evident that different methods of collision handling would play a key role in a hash table's performance. Therefore, with knowing certain parameters of a hash table, an optimized implementation can be deduced for maximized efficiency. This would prove extremely beneficial to any application for which hash tables are utilized, not limited to message digests, file systems, password verification, pattern matching, and programming language standard library hash tables, and beyond. These applications of hash tables are essential components to modern day life, and thus, they should also operate at their maximal efficiency. The research conducted in this paper hopes to serve as a starting point to further the development of the hash table data structure.

2. Background Information

2.1 The Hash Table

The *hash table* is a complex data structure which utilizes an associative array data type in order to format keys and value pairs.[1] Essentially, a hash function *H* is chosen which maps items $K \in U$ where *U* is a universal set of items of positive integer hash codes to an *M* size array of a hash table's bucket, where typically $M << U$, as follows:[2]

$$H: U \rightarrow \langle\, 0,\ 1,\ \ldots\ ,\ M\ -\ 1\,\rangle$$

Consequently, *load factor* is defined to be the ratio of bucket occupation in a hash table. By maintaining an optimal hash table size, evaluated by load factor, operation complexities such as retrieving a value can be more likely to be minimized to O(1).[3] Load factor α is given by formulae

$$\alpha = \frac{N}{M}$$

where *N* is the expected occupation of buckets in the hash table bucket array, and *M*, as aforementioned, is the bucket size of the hash table.[4] Optimally, the *M* should be a prime number hence leading to less clustering of *N* hash codes in a hash table bucket, as further explained in 2.2.

2.2 Hash Functions

For common programming usages, hash table keys are generally expressed as of string-type, however, the indexes of an array are denoted by integer values. Hence, hash functions in this usage can convert these keys to a memory location in a given bucket array as aforementioned. Despite there exists a great extent of hash functions, not limited to application in

[1] McDowell, Gayle Laakmann. "Data Structures: Hash Tables." *YouTube*, Hackerrank, 27 Sept. 2016, www.youtube.com/watch?v=shs0KM3wKv8.

[2] Erickson, Jeff. "Lecture 12: Hash Tables." *Jeff Erickson CS*, University of Illinois, 2015, jeffe.cs.illinois.edu/teaching/datastructures/notes/12-hashing.pdf.

[3] "Load Factor in HashMap - Javatpoint." *Javaatpoint*, Javaatpoint, www.javatpoint.com/load-factor-in-hashmap.

[4] "Sets & Maps Hash Tables." *15-121 Introduction to Data Structures*, Carnegie Mellon University, www.cs.cmu.edu/~tcortina/15-121sp10/Unit08B.pdf.

hash tables, the *modular hash function* is a commonly implemented hash function for hash tables, given by the function:[5]

$$H(K) = K \bmod M$$

The variable K is a summation of all ASCII (ordinal) values of each character in a given string-type key.[6, 7] Here, it is evident that M is optimized when it is a prime number; consider if M is even, then modularly dividing the value K by M may only result in the hash table mapping only to even-indexed buckets.

M*ultiplicative hashing* can similarly be implemented to compute hash codes. Generally speaking, multiplication is computed at a faster execution than division operators,[5] hence multiplicative hashing may be a preferable alternative. With multiplicative hashing, hash codes can be calculated by

$$H(K) = \lfloor S * \{M * c\} \rfloor$$

for constant $c \in \mathbb{R},\ 0 < c < 1$.[8, 9] For operation $\{M * c\}$ the positive non-integer value of the fractional part is taken. Note that a fraction part truncation is defined strictly to be $\{x\} = x - \lfloor x \rfloor$hence it only yields the positive decimal value of real number x. Constant c is generally restricted to be a rational number where c can be written in form $c = \frac{x}{2^w}$for $0 < x < 2^w$ where w is the machine word size in bits.[10] Donald Knuth's *The Art of Computer Programming v3* states that a suitable optimum c constant can be chosen to be that of the golden ratio $\varphi = \frac{1+\sqrt{5}}{2}$as it allows for even distribution of hash codes, paired with a relatively fast execution time.[10]

---

[5] "CS 3110 Lecture 21 Hash Functions." *Lecture 21: Hash Functions*, Cornell University, www.cs.cornell.edu/courses/cs3110/2008fa/lectures/lec21.html.

[6] "Hashing Data Structure." *GeeksforGeeks*, GeeksforGeeks, www.geeksforgeeks.org/hashing-data-structure/.

[7] Shaffer, Clifford A. "Hashing Tutorial." *Hashing Tutorial: Section 2.4 - Hash Functions for Strings*, Virginia Tech Algorithm Visualization Research Group, research.cs.vt.edu/AVresearch/hashing/strings.php.

[8] "5.1 ChainedHashTable: Hashing with Chaining." *Open Data Structures*, www.opendatastructures.org/versions/edition-0.1f/ods-java/5_1_ChainedHashTable_Hashin.html

[9] "What Are Hash Functions and How to Choose a Good Hash Function?" Edited by Rana (@ranadeepika2409) Deepika and Shijie (@shijiesheng113) Sheng, *GeeksforGeeks*, GeeksforGeeks, 23 Nov. 2020, www.geeksforgeeks.org/what-are-hash-functions-and-how-to-choose-a-good-hash-function/.

[10] "'Sorting and Searching.'" *The Art of Computer Programming*, by Donald Ervin Knuth, 3rd ed., Addison-Wesley Publishing Company, 1968.

A perfect hash function would distribute *N* hash codes to all *M* positions in a hash table bucket array, as it would concurrently allow for O(1) operations;[5, 11] the distribution uniformity can be evaluated by a statistical hypothesis test, the chi-squared test.[12, 13] However, if perfectly uniform distribution is not present and $H(K_1) = H(K_2) \mid K_1 \neq K_2$occurs, where two distinct keys passed through the same hash function yields the same hash code, a problem known as a collision arises.[2]

2.3 Collisions

Despite that a good hash function achieves relatively even hash code distribution, collisions must always be a consideration, as excessive hash table size is memory wastage, and perfect hash functions are only idealistic. The probability of a hash table collision amongst keys can be reasoned by the birthday paradox, which gives an analogy for which in 365 days in a year *M*, 60 randomly selected people *N* have an almost certain probability that at least one pair of people will have the same birthday, with *M* and *N* being variables with respect to load factor α, which can be calculated using the the formulae.

$$P_{N,M}(collision) = 1 - \prod_{i=1}^{N} P_{N,M}(i^{th}\ key\ non\ collision)$$

Hence collisions want to be resolved, by methods known as collision handling, as it is probably impossible to prevent them. Collision handling exists in a wide spectra of methods, including but not limited to *separate chaining, quadratic probing, double hashing,* and *linear probing*. Separate chaining is a collision handling method that refers to appending additional data structures from each bucket in the hash table. Such a principle is commonly utilized by implementing a linked list to chain value entries at a single index in the array bucket.[9] Nonetheless, arrays may be utilized in place of linked lists as the same purpose can be served despite linked lists being generally more memory efficient. At each value entry in the new data structure, multiple comparisons can be made to the requested key to return a correct value for

[11] Henry, Ed. "Hashing in Python." *Ed Henry's Blog*, © Ed Henry 2019, edhenry.github.io/2016/12/21/Hashing-in-Python/.

[12] Weisstein, Eric. "Chi-Squared Test." *Wolfram MathWorld*, Wolfram Research, mathworld.wolfram.com/Chi-SquaredTest.html.

[13] Pigeon, Steven. "Testing Hash Functions (Hash Functions, Part III)." *Harder, Better, Faster, Stronger*, WordPress.com, 13 Oct. 2015, hbfs.wordpress.com/2015/10/13/testing-hash-functions-hash-functions-part-iii/.

retrieving operations.[14] Theoretically then, the chaining collision handling method could hold an infinite amount of *N* values, however this is not computationally possible.

Double hashing is another technique belonging to the conjunction of *open addressing* that does not utilize the appenditures of additional data structures. Open addressing is a class of collision handling which generally refers to a search algorithm for the next empty bucket.[14] In double hashing, a hash value index is utilized to generate a repeatable probe sequence until a desired value is located, an empty bucket is found, or all *M* buckets have been searched. Firstly, a base hash function $H_1(K)$ is selected from a set of universal hash functions. Next, the implementation of the probe incrementation is determined by a secondary independent hash function $H_2(K)$, derived directly from the data. Evidently, it must be ensured that the probe value does not equate to 0. A basic formulation of double hashing is as follows:[15]

$$H_1(K) + i \times H_2(K) \, mod \, M$$

An alternate form of an open addressing scheme is quadratic probing, which comprises adding successive arbitrary values of quadratic polynomials as perfect squares which probes consecutively until a resultant open bucket in a hash table. The following mathematical sequence illustrates the incrementations taken by a quadratic probe.

$$\{H(K) + 1^2,\ H(K) + 2^2,\ H(K) + 3^2, \ldots, H(K) + n^2\}$$

Quadratic probing, however, does not guarantee a perfect success rate when operating in a hash table with a load factor of less than 0.5 with a prime numbered bucket size, and this statement has been mathematically proven.[16]

The last form of open addressing is linear probing. Linear probing scheme is displayed by an arithmetic sequence. A linear probe algorithm of probe incrementation 1 has the following search behaviour:[17]

$$S = \{H(K),\ H(K) + 1,\ H(K) + 2,\ \ldots,\ H(K) + n\}$$

[14] *Hash Table Collision Handling*. Michigan Technology University, www.csl.mtu.edu/cs2321/www/newLectures/17_Hash_Tables_Collisions.html.
[15] "Double Hashing in Data Structure" *Tutorialspoint*, Tutorialspoint, https://www.tutorialspoint.com/double-hashing-in-data-structure.

[16] de Greef, Lilian, *Lecture 7 - Hash Table Collisions*. University of Washington, https://courses.cs.washington.edu/courses/cse373/17su/lectures/Lecture%2007%20-%20Hash%20Table%20Collisions.pdf.

[17] "Why Is Quadratic Probing Better than Linear Probing?" Edited by Cisne Dunckelmann and Staff Editor2 EveryThingWhat, *Why Is Quadratic Probing Better than Linear Probing? | EveryThingWhat.com*, © EveryThingWhat.com LTD 2021, 22 June 2020, everythingwhat.com/why-is-quadratic-probing-better-than-linear-probing.

to find a next empty bucket in a cyclical manner, that is, cycling back to the beginning of the bucket array S [0] if any empty bucket is not found within the the index ranges $H(K)$ to $M - 1$.[18] The probe searches for the next empty bucket and returns to the beginning if an empty bucket is not found, until the bucket prior to the original hash code. This collision handling method is commonly implemented in hash tables as it conserves relatively more computational resources in most scenarios.[17] For all methods of open addressing, load factor α will always be in range $0 \leq \alpha \leq 1$ as any $\alpha > 1$ will result in an overload of the amount of buckets available for key value pairs; as α approaches 1, the number of collisions will simultaneously increase.[4] An alternate form of linear probing could be proposed by altering the probe incrementation (step) defined as s to be not limitedly set to the default 1, such that the algorithm simply probes between buckets at a rate with a greater gap of $s \mid s \geq 1$ as follows:

$$S = \{H(K),\ H(K) + s,\ H(K) + 2s,\ \dots,\ H(K) + ns\}$$

[18] "Linear Probing." *Sanford Class Archives*, Stanford University, web.stanford.edu/class/archive/cs/cs166/cs166.1166/lectures/12/Small12.pdf.

3. Experimental Methodology

Data obtained within this were conducted and altered based on a spectra of factors. The aim was to assess the efficiencies of all aforementioned collision handling methods paired with either the modular or multiplicative hash function, varied by hash code distribution and load factor. Thus, consistencies could be identified across various graphical presentation comparisons.

3.1 Controlled Variables

The Python 3.9.1 program ran 24 trials on a PC with the following specifications - CPU: Intel i5-2400 / RAM: 16GB / Windows 10 64-bit. The data results, saved to csv files, were processed by the Pandas Python library. The code for the programmed hash table remained consistent throughout experiments and can be found in the appendix.

10000 integer hash codes were randomly generated for each hash table trial. These 10000 hash codes were designed to fit nine degrees of "weighted distribution" - within the first 500 / 1000 / 2500 / 5000 / 7500 / 10000 / 20000 / 40000 / 20000000 buckets. The hash table is considered to be "extremely weighted" if the 10000 randomized hash codes fall forcefully into the first 500 buckets, and extremely unweighted if the 10000 randomized hash codes are distributed evenly to the first 20000000 buckets. Consequently, these hash codes would simultaneously guarantee the probability of collisions as validated in the birthday paradox.

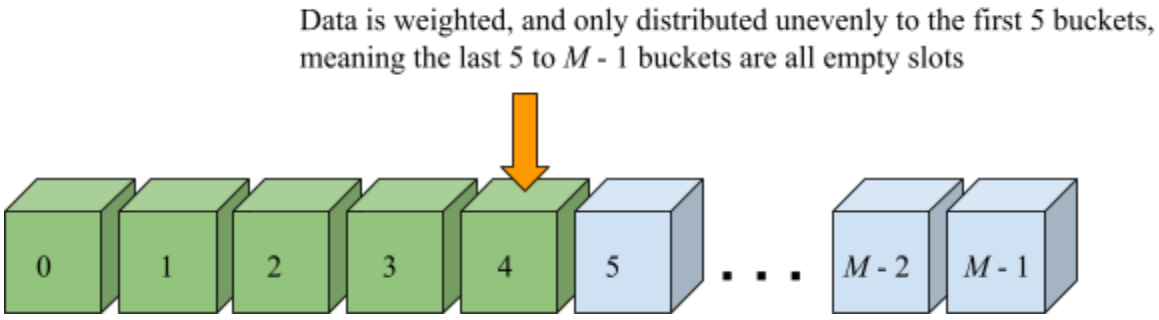

Figure 1: Illustration of weighted data distribution

Note that each data distributed were not key-value pairs of a string and some value. It was deemed unnecessary as the only difference would be the insertion or retrieval of a pair of items. Instead, each hash code inserted/retrieved had a key and value of itself, to firstly save execution time, and secondly to facilitate randomized generation for weighted distributions.

For each data set consisting of 10000 hash codes, four types of collision handling methods, mentioned in the background information, were evaluated as follows:

Collision Handling Method	**Description**
Separate Chaining	As stated.
Linear Probing	Probe incrementations from 1 through 99 inclusive.
Regular Quadratic Probing	Probe incrementations by i^2 --- $H(i) = H(0) + i^2$ $= H(0) + (i-1)^2 + 2i - 1$ $= H(i-1) + 2i - 1$
Altered Quadratic Probing	Probe incrementations by $\frac{i\,(i+1)}{2}$ --- $H(i) = H(0) + \frac{i\,(i+1)}{2}$ $= H(0) + \frac{i\,(i-1)}{2} + i$ $= H(i-1) + i$ --- The altered form of quadratic probing uses $H(i-1)$ instead of $H(0)$ to compute $H(i)$, theoretically being faster than regular quadratic probing.
Double Hashing 1	Modular hash function first.
Double Hashing 2	Multiplicative hash function first.

Then, for each of these cases of collision handling methods, twelve combinations of two hash functions, being either modular or multiplicative, paired with six load factors values α $(0.001 / 0.5 / 0.899 / 0.959 / 0.996 / 0.999)$ were evaluated. To achieve these distinguishable load factors stated, six specifically chosen prime numbers were chosen to be the bucket size M $(9999973 / 19993 / 11119 / 10429 / 10039 / 10009)$ respectively.

3.2 Dependent Variables

In these experiments, *execution time* was the dependent variable being measured in all sets. Execution time was measured for two hash table operations, being *insertion* and *retrieval* of an integer hash code, in seconds, by utilizing Python's `time` module and subtracting the current time `time.time()` by an initially set time, prior to the operation.

```
import time
inital_time = time.time()
execution_time = time.time() - inital_time
```

Figure 2: Python 3 Code Sample for *Execution Time*

3.3 Experimental Procedure

1. Program and implement the hash table with the stated specifications, or as found in Appendix
2. 10000 hash codes are to be randomly generated
3. The program is run on nested for-loops iterating through arrays of hash table specification that are implemented and evaluated
4. Data is saved to csv files while the loop is executed
5. Data is processed and displayed graphically using Python library Pandas, and Matplotlib

4. Graphical Data Visualization and Interpretation

4.1 Presentation of Data

The various sets of graphs plot the raw data obtained through experimental trials. Average retrieval times were calculated based on averages of 24 runs. Graphs are presented based on distribution of hash codes, load factor, and collision handling methods. More importantly, the following table outlines specifications present in the graphs.

Specification	**Relation**
Solid Lines	Modular Hash Function
Dotted Lines	Multiplicative Hash Function
Blue Lines	$\alpha = 0.001$
Cyan Lines	$\alpha = 0.5$
Green Lines	$\alpha = 0.899$
Magenta Lines	$\alpha = 0.959$
Red Lines	$\alpha = 0.996$
Black Lines	$\alpha = 0.999$

Furthermore, in Section 4.3 and Section 4.4, the *x*-axis indicates the extent of data distribution, thus simply showing a different viewpoint of the same data. The appending numbers following the worded distribution (500, 1000, 2500, ...) specify the degree of distribution weightedness of the hash table buckets, as mentioned in 3.1. The distribution henceforth determines the extent to which a hash function and a collision handling method affects the efficiency of a hash table. This is graphed as follows:

***x*-axis Value**	**Distribution**
0	Extremely Weighted Distribution - 500
1	Highly Weighted Distribution - 1000
2	Fairly Weighted Distribution - 2500
3	Somewhat Weighted Distribution - 5000
4	Weighted Distribution - 7500
5	Unweighted Distribution - 10000
6	Unweighted Distribution - 20000
7	Highly Unweighted Distribution - 40000
8	Extremely Unweighted Distribution - 20000000

4.2 Hash Tables with Data Distribution and Linear Probing Steps

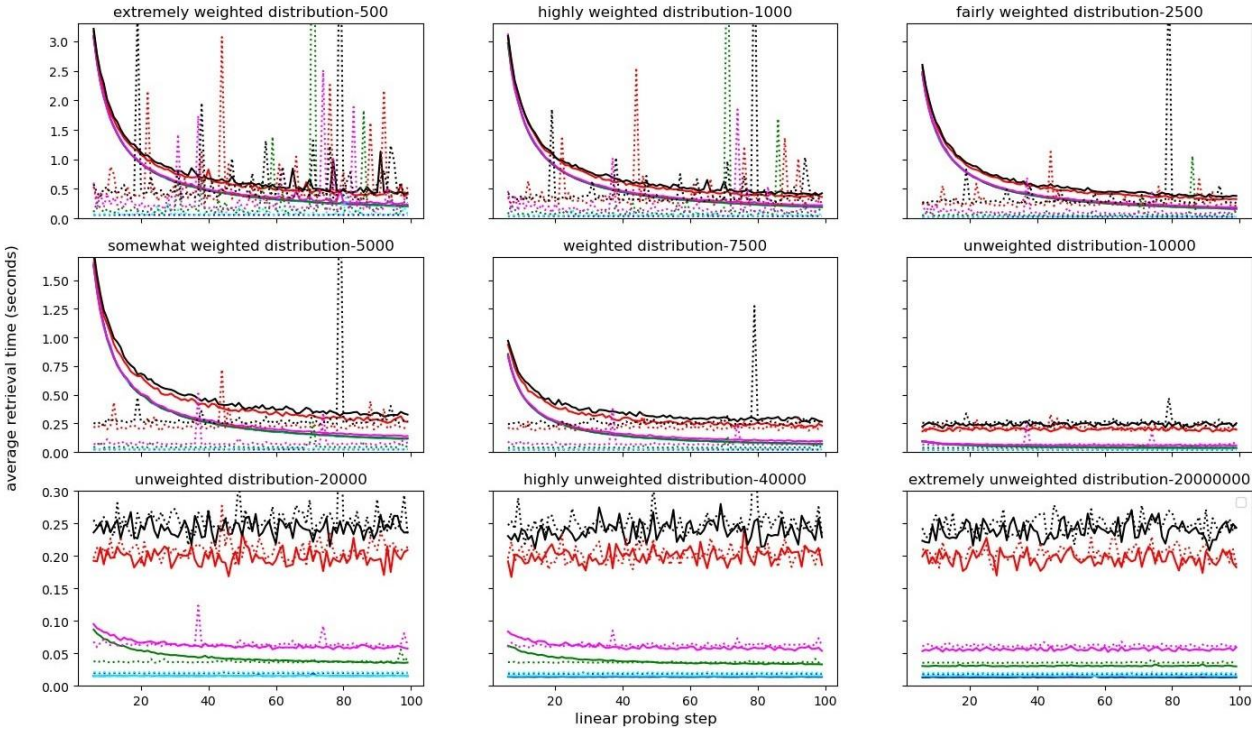

Figure 3

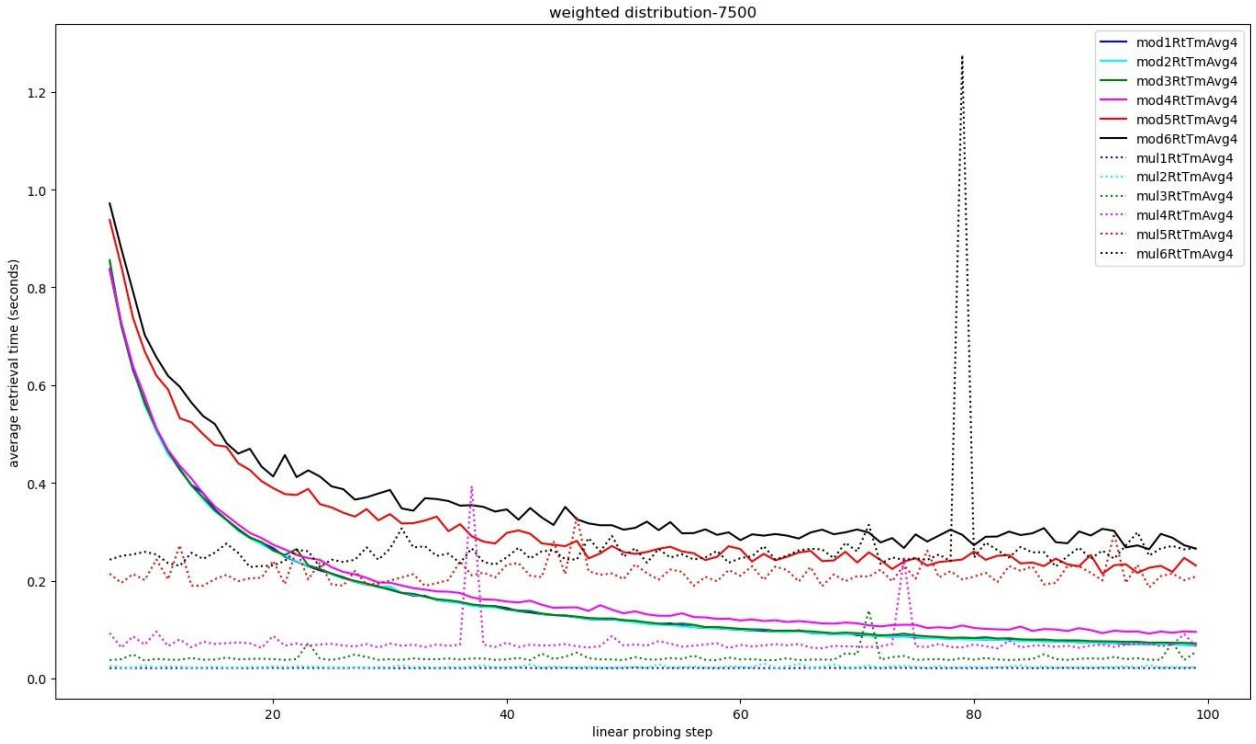

Figure 4: Weighted Distribution - First 7500 Buckets (Enhanced)

From Figure 3, it can be generally seen how the dataset distribution affects execution time efficiency. When the data is distributed unevenly, evidently some possible form of inverse relationship exists. From figure 4, a zoomed and enhanced depiction of one of these graphs, this inverse relationship discernibly only occurs with hash tables using the modular hash functions. Considering that our hash table bucket of size $M = 10000$, and the distribution weight of the hash codes fall into the first 7500 buckets, the hash codes will remain hashed towards this area of the bucket through modular division. Thus, in these 7500 buckets, the 10000 hash codes would collide frequently and thereby resulting in higher execution times. The spectrum of graphs shown in Figure 3 of consecutively less weighted hash tables show that when the dataset is less weighted, the hash table performs more efficiently, thus justifying this conclusion.

Visually, for hash tables implemented with the multiplicative hash function, the inverse relationship does not exist. The multiplicative hash function distributes hash codes to the whole bucket rather than the intended weight for the first 7500 in Figure 4. This can be logically concluded by returning to the theory of the multiplicative hash function, and its multiplying constant $\{c\} = \{\varphi\}$ which distributes hash codes evenly throughout the hash table bucket regardless, and thus simultaneously justifying the effectiveness of that value. Therefore, no inverse relationship could exist as the multiplicative hash function would always exist as the relative equivalent of being “extremely unweighted”, and this can be seen for higher linear probing step values even when paired with the modular hash function. Thus, in general, the multiplicative hash function should generally be preferred as it is independent of the weight of a dataset.

For when linear probing was implemented with the modular hash function, it appeared that as the step decreased, the clustering of hash codes increased, and noticeably consumed exponentially more execution time in searching the next available bucket. One of the main inefficiencies of the one step linear probing method is the overclustering of hash codes in an area of a bucket in the event of a collision, and it is likely that the greater steps while probing reduces the clusters and thus enhances performance. Furthermore, it is additionally evident that a lower load factor ratio decreased the execution time of the hash table operations (insertion/retrieval).

4.3 Hash Tables with Load Factor and Linear Probing Steps

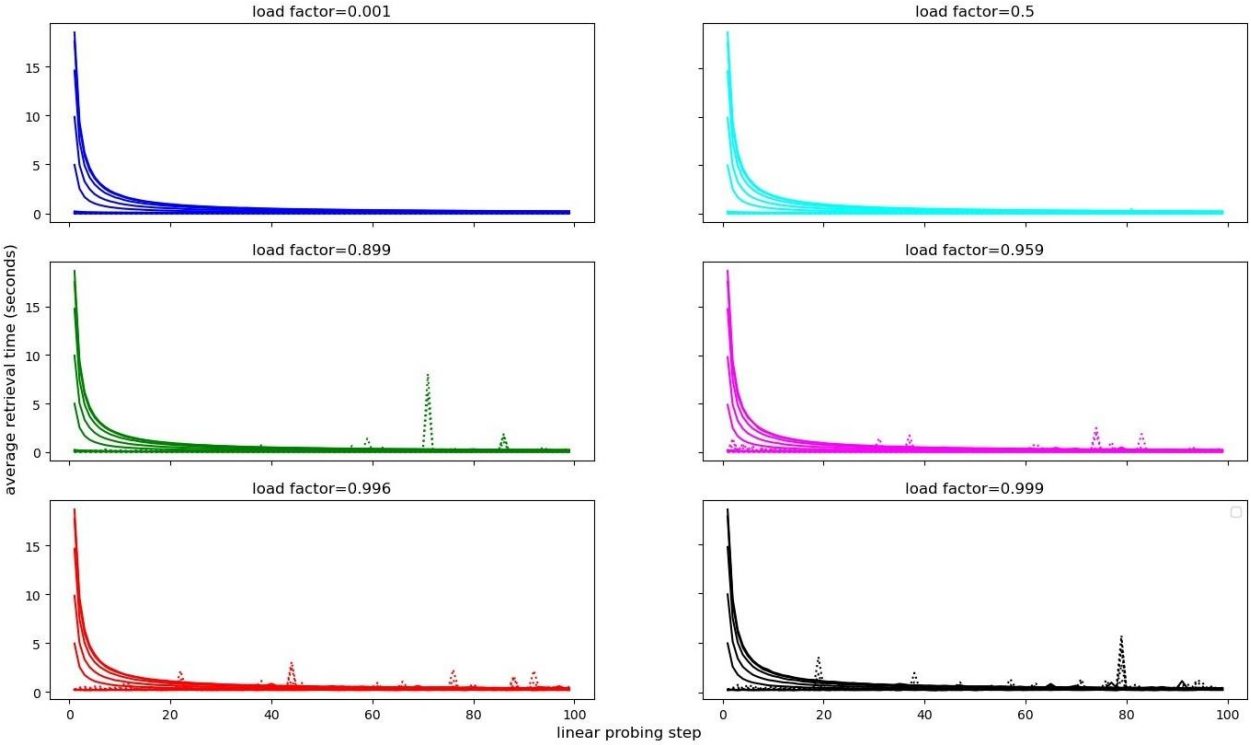

Figure 5

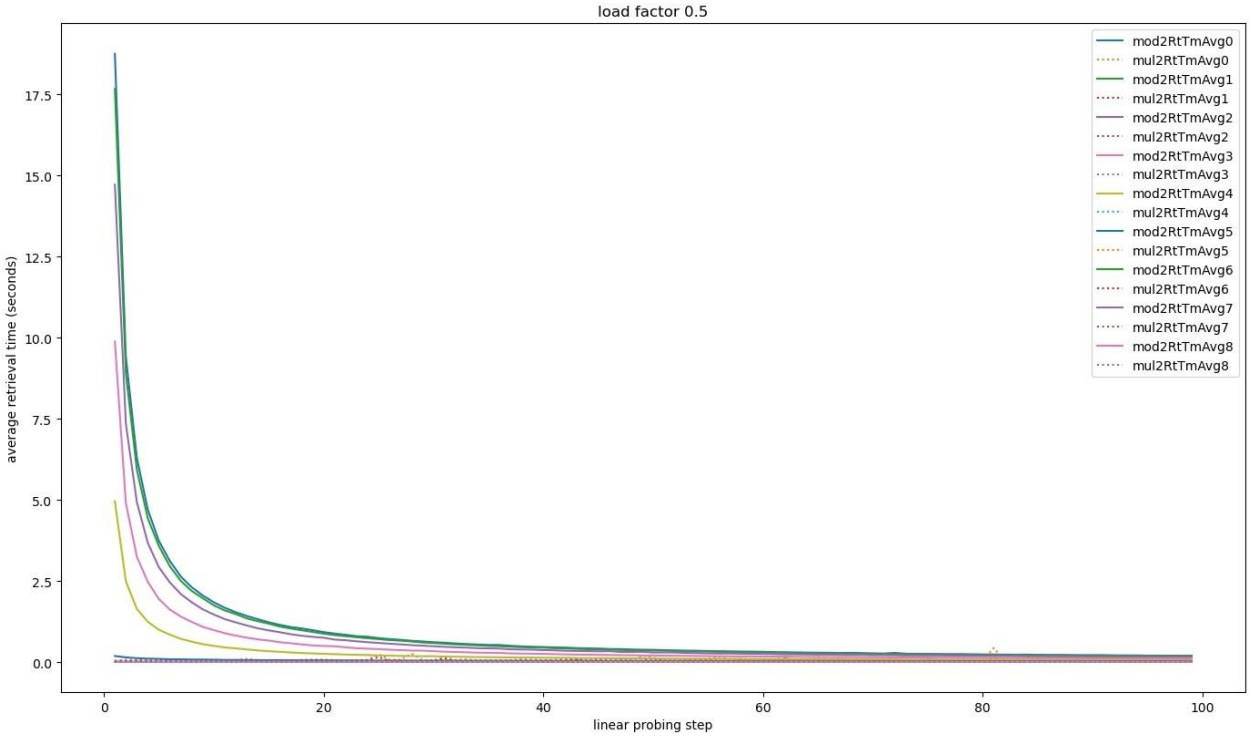

Figure 6: Hash Tables with Load Factor of 0.5 (Enhanced)

Figure 5 displays the effect of hash table load factor on its execution time efficiency. This is clearly shown in Figure 6. The trend is consistent for all load factor values. Interestingly, unlike what occurred in varying data distributions, the values of the data points in each hash table of varying load factors simultaneously align, with the trend occurring at identical values. This implies that there exists a function which relates the execution time to the linear probing step, that is not influenced by the load factor, however, is influenced by data distribution, when compared to the data displayed from Section 4.1.

4.4 Collision Handling Methods and Data Distribution

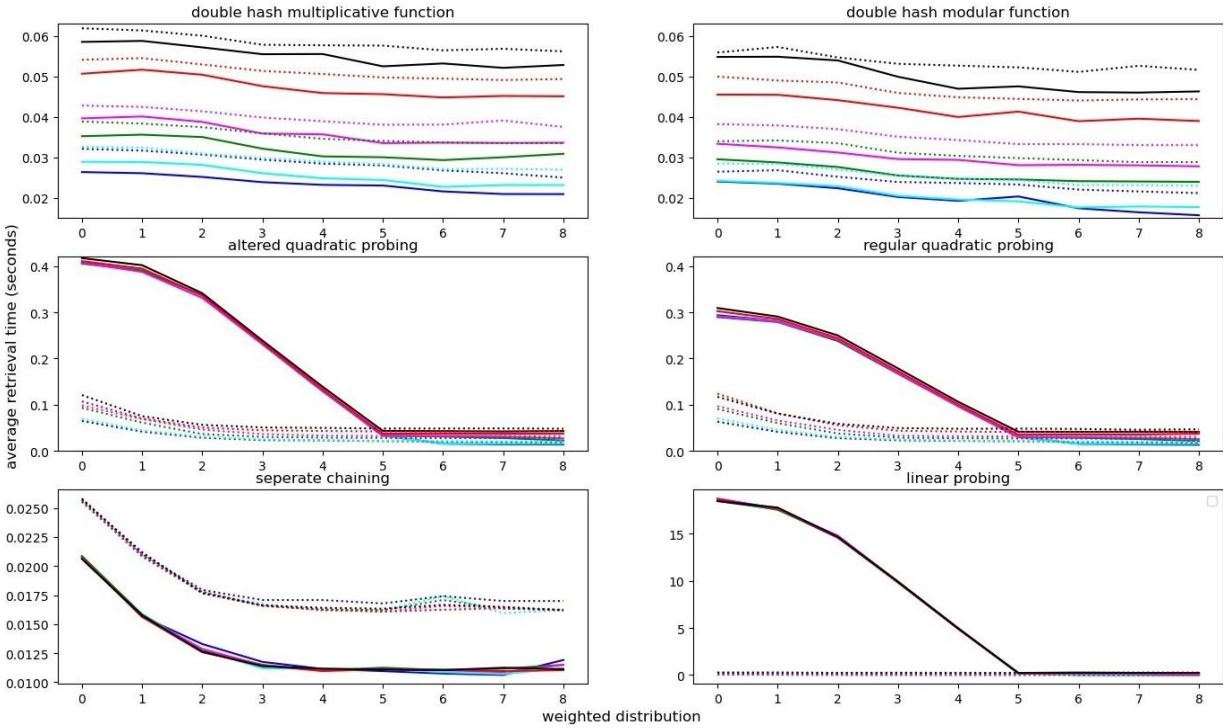

Figure 7

Figure 7 displays the effect of the varying collision handling methods on hash table efficiency. Note that here the linear probing step is 1. In every case of distribution degree, separate chaining performed the most efficiently in terms of execution time; furthermore, its modular hash function performed better than its multiplicative hash function counterpart. Unlike open addressing techniques, separate chaining does not search the whole bucket array as it only operates at its respective hash code. Its time complexity can be defined as O(N) in the worst, improbable case, where all N hash codes collide at the same hash code, although probabilistically

this is unlikely. Yet, this is still significantly better than, for example, one-step linear probing at O(M) time complexity, as recall that $N < M$ for open addressing because the load factor must not exceed 1. Due to separate chaining being the most unique collision handling method relative to the other methods presented, it is thus difficult to draw fair comparisons. It is the only form of collision handling that allows for a load factor greater than 1, that is, $N > M$. However, this does result in an extremely memory-inefficient collision handling method due to extraneous unused dynamic arrays/linked lists data structures appended to each bucket of the hash table.

Comparatively, the double hash collision handling method also excelled in execution time efficiency. More importantly, the data retrieved from the double hash method shows that this collision handling technique is independent from the degree of data distribution weight compared to other methods of collision handling, that is, its efficiency is generally uninfluenced by distribution; moreover, using the modular hash function first when double hashing simultaneously produced marginally faster runtimes.

From the graph depicting linear probing data results, evidently linear probing performed best when paired with the multiplicative hash function for all forms of distribution. However, the performance of these hash functions converge to be almost instantaneously similar when the distribution of the data falls into the first 10000 buckets, marked by number 5 on the x-axis representing “unweighted distribution”. When the distribution was unweighted (for any value on the x-axis past or equal to 5), the linear probing collision handling method performed more efficiently than the double hash function in all cases. Lastly, despite quadratic not achieving runtimes faster than the double hash function, it was still more time efficient than linear probing in most scenarios.

4.5 Collision Handling Methods and Load Factors

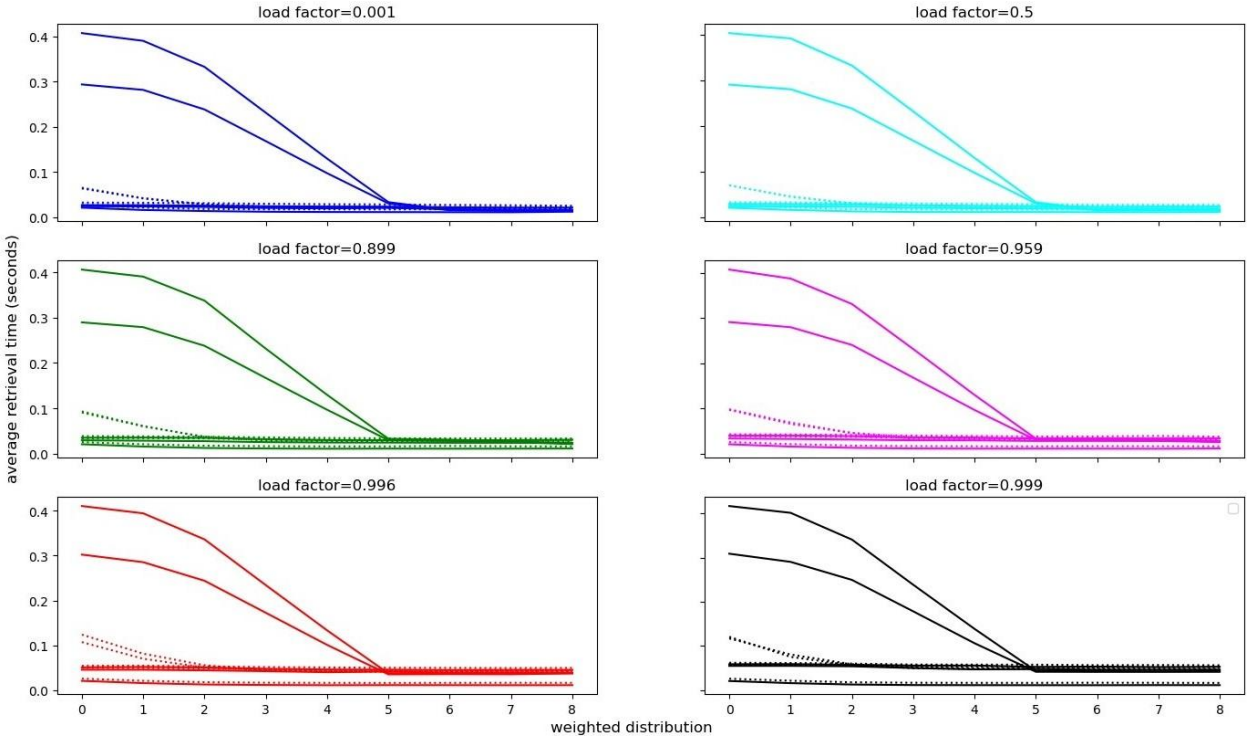

Figure 8

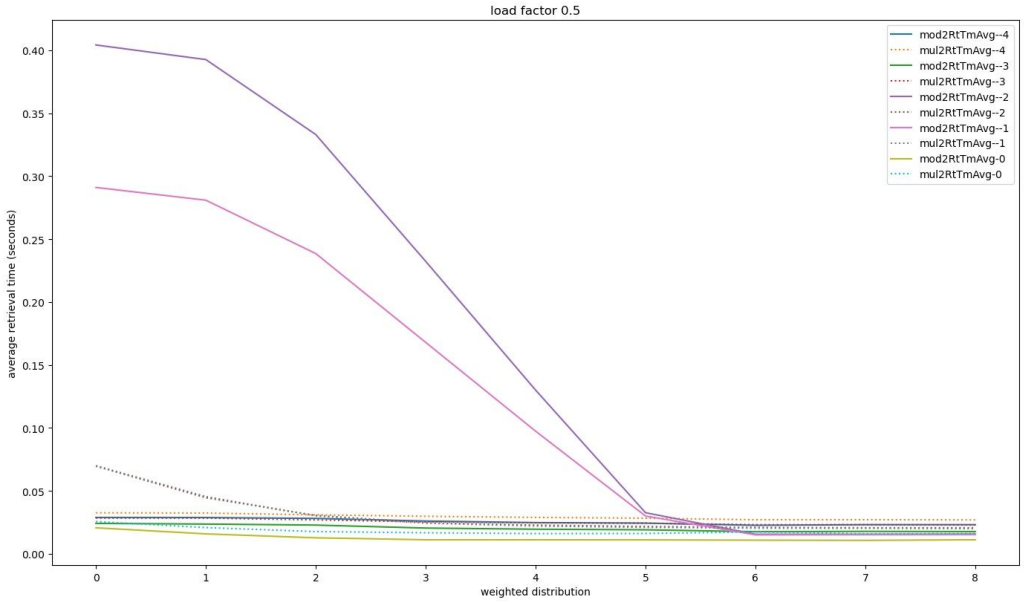

Figure 9: Collision Handling Hash Tables with Load Factor of 0.5

Figure 8 depicts the relationship of load factor to the efficiency of varying collision handling methods. The quadratic probing technique showed the most performance variations for different weighted data distributions, especially with the modular hash function. Relative to other collision handling methods, it appeared to be the only method that was influenced by load factors, for all degrees of data distribution weight. In addition, the altered quadratic probing collision handling method was not as efficient as predicted, as the lower rate of probing step increase resulted in greater clustering, therefore overtaking the benefits of less computing.

The final graph, Figure 9, depicts a clearer representation of the effect of load factor and data distribution weight. In terms of execution time, it is logical that lower load factor ratios in turn results in a faster hash table as, probability-wise, less collisions would occur upon the insertion of data, and thus less iteration upon the retrieval of data. Nonetheless, to formulate a smaller load factor ratio α, which recall to be defined by $\alpha = \frac{N}{M}$, the hash table bucket size *M* must be greater, or *significantly* greater than the *N* number of hash codes to be distributed. Accordingly, it is unideal to choose an unreasonably large *M* value relative to *N* as it would use an unnecessary amount of computational resources and thus reduce efficiency. Hence, when considering the dataset size, lower load factor values are optimal, however, the memory usage should also be considered to find a balance that is suitable to the program.

5. Conclusion

When computing a significant amount of data in hash tables, the way in which the hash codes are distributed within the bucket significantly influences its efficiency. Consequently, the comparison of the modular and multiplicative hash function shows that the multiplicative hash function is significantly more independent, if not completely independent from the extent to which the data distribution is weighted in the hash table bucket, whereas the modular hash function's efficiency is almost entirely based around this factor. Therefore, and in general for all distributions, it is more suited that the multiplicative hash function should be implemented over the modular hash function, specifically for linear and quadratic probing collision handling methods. In contrast, for separate chaining and double hashing, the modular hash function is the preferred hash function.

Evidently amongst all forms of collision handling analyzed, it can be seen that certain collision handling methods were more efficient when paired with different aforementioned hash functions. Each of these techniques were shown to excel or flaw under certain conditions - double hashing was uninfluenced by data distribution weight, quadratic probing does not guarantee successful searches, separate chaining could have the lowest time complexity at the risk of the highest space complexity, and the inverse relationship produced by the modular hash function in linear probing schemes.

From this information, future implementations of hash tables seeking to achieve maximum efficiency could greatly benefit from known prior knowledge regarding the data being inserted, such as its possible distribution weight and load factor, and thus can utilize the most optimal combination of hash function and collision handling method. Ergo the importance of collision handling, and the value various collision handling methods have under varying circumstances, serves as a functional and applicable tool by presenting maximized efficient collision handling solutions subsequent to the usage of a hash function, in all applications in the modern world that use hash tables.

6. Works Cited

1. McDowell, Gayle Laakmann. “Data Structures: Hash Tables.” *YouTube*, Hackerrank, 27 Sept. 2016, www.youtube.com/watch?v=shs0KM3wKv8.
2. Erickson, Jeff. “Lecture 12: Hash Tables.” *Jeff Erickson CS*, University of Illinois, 2015, jeffe.cs.illinois.edu/teaching/datastructures/notes/12-hashing.pdf.
3. “Load Factor in HashMap - Javatpoint.” *Javaatpoint*, Javaatpoint, www.javatpoint.com/load-factor-in-hashmap.
4. “Sets & Maps Hash Tables.” *15-121 Introduction to Data Structures*, Carnegie Mellon University, www.cs.cmu.edu/~tcortina/15-121sp10/Unit08B.pdf.
5. “CS 3110 Lecture 21 Hash Functions.” *Lecture 21: Hash Functions*, Cornell University, www.cs.cornell.edu/courses/cs3110/2008fa/lectures/lec21.html.
6. “Hashing Data Structure.” *GeeksforGeeks*, GeeksforGeeks, www.geeksforgeeks.org/hashing-data-structure/.
7. Shaffer, Clifford A. “Hashing Tutorial.” *Hashing Tutorial: Section 2.4 - Hash Functions for Strings*, Virginia Tech Algorithm Visualization Research Group, research.cs.vt.edu/AVresearch/hashing/strings.php.
8. “5.1 ChainedHashTable: Hashing with Chaining.” *Open Data Structures*, www.opendatastructures.org/versions/edition-0.1f/ods-java/5_1_ChainedHashTable_Hashin.html.
9. “What Are Hash Functions and How to Choose a Good Hash Function?” Edited by Rana (@ranadeepika2409) Deepika and Shijie (@shijiesheng113) Sheng, *GeeksforGeeks*, GeeksforGeeks, 23 Nov. 2020, www.geeksforgeeks.org/what-are-hash-functions-and-how-to-choose-a-good-hash-function/.
10. “‘Sorting and Searching.’” *The Art of Computer Programming*, by Donald Ervin Knuth, 3rd ed., Addison-Wesley Publishing Company, 1968.
11. Henry, Ed. “Hashing in Python.” *Ed Henry's Blog*, © Ed Henry 2019, edhenry.github.io/2016/12/21/Hashing-in-Python/.

12. Weisstein, Eric. “Chi-Squared Test.” *Wolfram MathWorld*, Wolfram Research, mathworld.wolfram.com/Chi-SquaredTest.html.
13. Pigeon, Steven. “Testing Hash Functions (Hash Functions, Part III).” *Harder, Better, Faster, Stronger*, WordPress.com, 13 Oct. 2015, hbfs.wordpress.com/2015/10/13/testing-hash-functions-hash-functions-part-iii/.
14. *Hash Table Collision Handling*. Michigan Technology University, www.csl.mtu.edu/cs2321/www/newLectures/17_Hash_Tables_Collisions.html.
15. “Double Hashing in Data Structure” *Tutorialspoint*, Tutorialspoint, https://www.tutorialspoint.com/double-hashing-in-data-structure.
16. de Greef, Lilian, *Lecture 7 - Hash Table Collisions*. University of Washington, https://courses.cs.washington.edu/courses/cse373/17su/lectures/Lecture%2007%20-%20Hash%20Table%20Collisions.pdf.
17. “Why Is Quadratic Probing Better than Linear Probing?” Edited by Cisne Dunckelmann and Staff Editor2 EveryThingWhat, *Why Is Quadratic Probing Better than Linear Probing? | EveryThingWhat.com*, © EveryThingWhat.com LTD 2021, 22 June 2020, everythingwhat.com/why-is-quadratic-probing-better-than-linear-probing.
18. “Linear Probing.” *Sanford Class Archives*, Stanford University, web.stanford.edu/class/archive/cs/cs166/cs166.1166/lectures/12/Small12.pdf.

Appendix

The following was the Python 3.9.1 code used to conduct the experiments in this paper.

```
import time #Measures execution time in seconds
import random #For randomization of strings
import string #For random strings
import pandas #Process bucket slot and chain length
from datetime import timedelta #Display elapsed time
from sys import getsizeof #Measures memory usage in bytes
from math import floor, ceil, modf, sqrt #For multiplicative Hashing only

class HashTable: #Hash Table Specification
    def __init__(self, bs, hf, step):
        self.size = bs # Prime number of bucket size
        self.hf = hf # Hash function: modular or multiplicative
        self.step =  step # -4,-3: double hash / -2,-1: quadratic probing / 0:
chaining / > 0: linear probing
        self.max_probing = bs # max probing to quit

        self.ch = "ch" if step == 0 else "lp" # Collision handling strategy
        self.mf = (sqrt(5)-1)/2 # Use golden ratio as multiplicative factor

        # [] for chaining, None for linear probing
        self.bucket = [[] for generation in range(self.size)] if self.ch ==
"ch" else [None for generation in range(self.size)]

    #print every key and its slot/chain
    def print_bucket(self):
        if self.ch == "ch":
            for slot in range(self.size):
                for chain in range(len(self.bucket[slot])):
                    print(slot,chain,self.bucket[slot][chain])
```

```
        else:
            for slot in range(self.size):
                if self.bucket[slot] is not None:
                    print(slot,self.bucket[slot])

    #measure memory usage in bytes
    def memory_usage(self):
        return getsizeof(self.bucket)

    # hash key: convert key string to a integer for hash function input
    def hash_key(self, key):
        #hash_key = 0
        #for i in range(len(key)):
        #    hash_key += ord(key[i]) * (10**i)
        #return hash_key
        return key

    # hash function, return slot
    def hash_function(self, key):
        hash_key = self.hash_key(key)
        slot = hash_key % self.size if self.hf == "mod" else floor( self.size *
modf(hash_key*self.mf)[0] )
        return slot

    # double hash function: slot = slot0 + j * hf2(key,j)
    # must > 0 for open addressing / closed hashing
    def double_hash_function(self, key, j):
        hash_key = self.hash_key(key)

        if self.step > 0 :     # linear probing
            return self.step
        elif self.step == -1 : # classic quadratic probing
            return j
        elif self.step == -2 : # faster quadratic probing
```

```
            return j
        elif self.step == -3 : # double hash modular function   ###    better
if self.size and self.size-2 are twin primes
            return 1 + hash_key % (self.size-2)
        elif self.step == -4 : # double hash multiplicative function
            return ceil((self.size-2) * modf(hash_key * self.mf)[0])
        elif self.step == -5 : # classic quadratic probing with a large
starting step
            return j
        elif self.step == -6 : # faster quadratic probing with a large starting
step
            return j
        elif self.step == -7 : # double hash modular function with quadratic
probing
            return 1 + hash_key % (self.size-2)
        elif self.step == -8 : # double hash multiplicative function with
quadratic probing
            return ceil((self.size-2) * modf(hash_key * self.mf)[0])
        else:
            raise Exception(f"Invalid collision handling strategy! Check the
step input.")

    # find next slot based on current slot and j-th probing
    # Warning: fixed-step (-4,-3,>0) always success if it is coprime to bucket
size; however, j-varied-step (-8,-7,-6,-5,-2,-1) may fail after many probings
in small bucket
    def next_slot(self, slot_now, j, step0):
        if self.step > 0 :                      # linear probing
            slot_next = slot_now + self.step    # hf2(key,j) = step    =>
step_j = (slot0 + j*step) - (slot0 + (j-1)*step) = step
        elif self.step == -1 :                  # classic quadratic probing
            slot_next = slot_now + 2*j-1        # hf2(key,j) = j       =>
step_j = (slot0 + j^2) - (slot0 + (j-1)^2) = j^2 - (j-1)^2 = 2*j-1
        elif self.step == -2 :                  # faster quadratic probing
```

```
            slot_next = slot_now + j            # hf2(key,j) = (j+1)/2 =>
step_j = (slot0 + j*(j+1)/2) - (slot0 + (j-1)*(j-1+1)/2) = j
        elif self.step in [-3,-4] :             # double hash function
            slot_next = slot_now + step0        # double hash without j in
hf2(key,j) = step0: it is just a key-varied-step linear probing
        elif self.step == -5 :                  # classic quadratic probing
with a large starting step
            slot_next = slot_now + 2*j + 999    # hf2(key,j) = j + stepK
        elif self.step == -6 :                  # faster quadratic probing with
a large starting step
            slot_next = slot_now + j + 1000     # hf2(key,j) = (j+1)/2 + stepK
        elif self.step in [-7,-8] :             # double hash function with
quadratic probing
            slot_next = slot_now + step0 + j    # double hash with j in
hf2(key,j) = step0 + (j+1)/2

        return slot_next % self.size

    # insert key, return bucket slot
    def __setitem__(self, key, hash_key):
        slot = self.hash_function(hash_key)

        if self.ch == "ch":
            self.bucket[slot].append(key)
            return slot
        else:
            step0 = self.double_hash_function(key,0) # key-varied-step for
double hash

            for j in range(1,self.max_probing): # chain counting
                if self.bucket[slot] is None:
                    self.bucket[slot] = key
                    return slot
```

```
                slot = self.next_slot(slot, j, step0)

            raise Exception(f"Insertion failed after {j} probing for (key,
hash_key) = ({key}, {hash_key}).")


    # query key, return chain location
    def __getitem__(self, key, hash_key):
        slot = self.hash_function(hash_key)

        if self.ch == "ch":
            for j in range(len(self.bucket[slot])):
                if self.bucket[slot][j] == key:
                    return j
        else:
            step0 = self.double_hash_function(key,0) # key-varied-step for
double hash

            for j in range(1,self.max_probing): # chain counting
                if self.bucket[slot] == key:
                    return j

                slot = self.next_slot(slot, j, step0)

            raise Exception(f"Retrieval failed after {j} probing for (key,
hash_key) = ({key}, {hash_key}).")

    # return slot for each key
    def insert(self, keys):
        slot_list = []
        for i in range(len(keys)):
            slot_list.append(self.__setitem__(i,keys[i]))
        return slot_list
```

```
    # return collision / chain length for each key
    def retrieve(self, keys):
        chain_list = []
        for i in range(len(keys)):
            chain_list.append(self.__getitem__(i,keys[i]))
        return chain_list

def main():
    starting_time = time.time()
    N             = 10000 # raw dataset size
    roundN        = 25    # repeat round for average statistics
    hk_range      = [500, 1000, 2500, 5000, 7500, 10000, 20000, 40000,
20000000] # create hash_key from extremely uneven to even distribution
    datasetN      = len(hk_range)     # total datasets to test
    stepN         = 100   # total steps to test
    stepS         = -4    # starting step
    hf_list       = ["mod", "mul"] # hash function list
    bs_list       = [9999973, 19993, 11119, 10429, 10039, 10009] # bucket size
list

    # repeat rounds for average statistics
    for r in range(roundN):
        # reset round initial time
        round_initial_time = time.time()

        # test on different distribution datasets
        for ds in range(datasetN):
            # reset dataset initial time
            dataset_initial_time = time.time()

            # create a new raw dataset for each new round
            keys = [random.randint(1, hk_range[ds]) for i in range(N)]

            # create a new output csv file for each new round
```

```
            file = open(f"Hash_D{str(ds).zfill(2)}-R{str(r).zfill(2)}.csv","a")

            # first row of output csv file
            file_header = "step,"
            for hf in hf_list:
                for bs in range(len(bs_list)):
                    col = hf + str(bs+1)
                    file_header +=
col+"InTm,"+col+"RtTm,"+col+"CMed,"+col+"CMax,"+col+"CAvg,"+col+"CStd,"

            # remove the last comma
            file.write(file_header[:-1])

            # step = -8: open addressing / closed hashing: double hash
multiplicative function with quadratic probing
            # step = -7: open addressing / closed hashing: double hash modular
function with quadratic probing
            # step = -6: open addressing / closed hashing: faster quadratic
probing with a large starting step
            # step = -5: open addressing / closed hashing: classic quadratic
probing with a large starting step
            # step = -4: open addressing / closed hashing: double hash
multiplicative function
            # step = -3: open addressing / closed hashing: double hash modular
function
            # step = -2: open addressing / closed hashing: faster quadratic
probing
            # step = -1: open addressing / closed hashing: classic quadratic
probing
            # step =  0: seperate chaining / open hashing
            # step >  0: open addressing / closed hashing: linear probing
            for step in range(stepS, stepN):
                # start with a return and step for each new row
                file.write(f"\n{step}")
```

```
                # test different operations in one row
                for hf in hf_list: # hash function
                    for bs in bs_list: # change bucket size (prime number) to
have different load factor
                        hash_table = HashTable(bs, hf, step)

                        # Insertion
                        initial_time   = time.time()
                        slot_list      = hash_table.insert(keys)
                        insertion_time = time.time() - initial_time

                        # Retrieval
                        initial_time   = time.time()
                        chain_list     = hash_table.retrieve(keys)
                        retrieval_time = time.time() - initial_time

                        # collision / chain length on every bucket slot
                        df = pandas.DataFrame({'slot':slot_list,
'chain':chain_list})
                        cdf = df.groupby(['slot']).max()

                        # one operation completed

file.write(f",{insertion_time:.10f},{retrieval_time:.10f},{cdf.chain.median()},
{cdf.chain.max()},{cdf.chain.mean()},{cdf.chain.std()}")

                # one step row calculation completed
                dataset_percent = 100*(step-stepS+1)/(stepN-stepS)
                round_percent   =
100*((step-stepS+1)+ds*(stepN-stepS))/((stepN-stepS)*datasetN)
                total_percent   =
100*((step-stepS+1)+ds*(stepN-stepS)+r*(stepN-stepS)*datasetN)/((stepN-stepS)*d
atasetN*roundN)
```

```
                dataset_time    = round( time.time() - dataset_initial_time )
                round_time      = round( time.time() - round_initial_time )
                total_time      = round( time.time() - starting_time )

                print(f"{step:4d} step in dataset({ds+1:2d}/{datasetN})
{dataset_percent:6.2f}% completed in {timedelta(seconds=dataset_time)}. " +
                      f"Round({r+1:2d}/{roundN}) {round_percent:6.2f}%
completed in {timedelta(seconds=round_time)}. " +
                      f"Total {total_percent:8.4f}% completed in
{timedelta(seconds=total_time)}")

            # one dataset file output completed
            file.close()

            #round_percent = 100*(ds+1)/datasetN
            #total_percent = 100*(ds+1+r*datasetN)/(datasetN*roundN)
            #round_time    = round( time.time() - round_initial_time )
            #total_time    = round( time.time() - starting_time )

            #print(f"{ds+1:2d}/{datasetN} dataset file output in
round({r+1:2d}/{roundN}) {round_percent:6.2f}% completed in
{timedelta(seconds=round_time)}. " +
            #      f"Total {total_percent:8.4f}% completed in
{timedelta(seconds=total_time)}")

if __name__ == '__main__':
    main()
```