# <u>Investigating the Effects of Data Structure Implementation on Performance and Efficiency</u>

**To what extent does linear probing serve as a more efficient method for collision handling in Hash Tables in comparison to chaining?**

A Computer Science Extended Essay

---

3981 words

Mingchung Xia - Charles P. Allen High School

## Table of Contents

## 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. Software engineers, developers, programmers, and computer researchers seek to utilize the principles of computer science, ergo developing applications for such methods. 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 methods in which a computer is able to, but not limited to, store, allocate, and organize data. This hence results in techniques for which applicable problems can be resolved; data structures are operated by algorithms and operations.

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; as such, many programming languages have a natively implemented model of hash tables for ease of use. However, the flaws present during the operation of the hash function will fundamentally result in collisions, which occurs when multiple distinct keys hashed through the same hash function return the same integer hash code.

This paper seeks to compare and contrast two different methods of handling collisions (collision handling) - *linear probing* and *chaining* - and the extent of either method's varying efficiencies when implemented in hash tables. The experimental data retrieved would prove highly practical in the field of cryptography; the 2019 Facebook password leak of 419 million users was the consequence of the lack of hash function encryption, subsequently leaving private information unprotected. Therefore, as any hash function utilized, for applications not limited to cryptography, fundamentally results in collisions, the most efficient solution to a fundamentally present problem must be implemented.

To investigate and evaluate the efficiencies of these hash table collision handling methods, hash tables were implemented with differing controlled variables, to undermine results across various platforms. Hence for analysis, logical mathematical and computer science theory were utilized to lastly determine the extent to which linear probing serves as a more efficient method for collision handling in hash tables in comparison to chaining.

## Background Information

### 2.1 Asymptotic Notation

For analysis of algorithms in computer science, utilizing and implementing mathematical logic and tools are essential components. Any algorithm will take some amount of time and space to execute, which are mathematically expressed as an *asymptotic notation*, derived from functions to capture the essence of an algorithm when applied on a data structure.[Abhishek, Ref. 1] Thus, time complexities and space complexities are fundamental deviations of asymptotic notations.

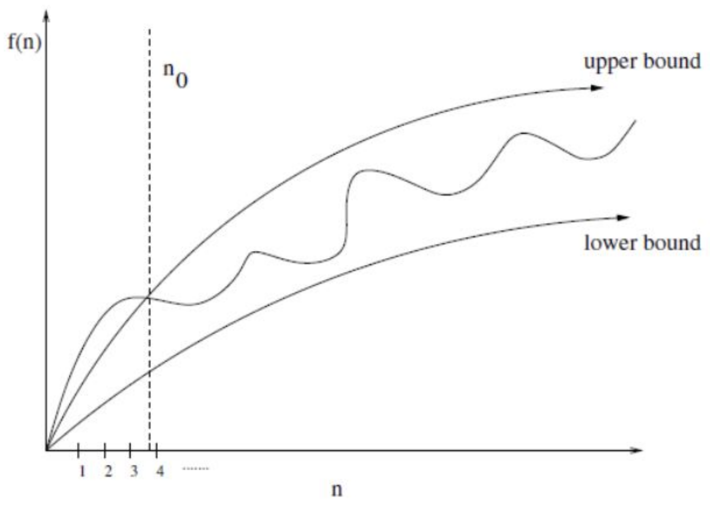

Figure 2.1.1

For valid bound-behaviour ranges $n \geq n_0$, the time complexity of an algorithm is shown to be graphed between lower and upper bounds as an average range. Big O notation refers to the upper bounds of an algorithm, indicating that some algorithm f(*n*) will never exceed the specified time complexity bounded by Big O notation for input *n*.[Abhishek, Ref. 2] In terms of data structures, time complexities measured by Big O analysis are dependent by algorithms and/or operations implemented on them. In contrast, *Space Complexity* analysis refers to the amount of memory an algorithm requires for execution, formulated by *Space Complexity = Auxiliary Space + Input Space*. Auxiliary space is defined to be the excess temporary space allocated by a system.[Abhishek, Ref. 3] For calculating input space, generally only the *data space* is concerned, which is the memory utilized by all variables and constants. Both time and space complexities should ideally be minimized.

## **2.2 Hash Tables**

The *hash table* is a complex data structure which utilizes an associative array data type in order to format values.[McDowell, Ref. 16] The basic concept is to associate a key and value by a hash function $H$ that maps items $K$ to index (hash code) by $H(K)$ in an array, known as a bucket, for high level programming languages,[McDowell, Ref. 16, Erickson, Ref. 21] shown in Figure 2.2.1. If $H(K)$ were to be equal to $K$, the data structure would simply be an array.[Erickson, Ref. 21]

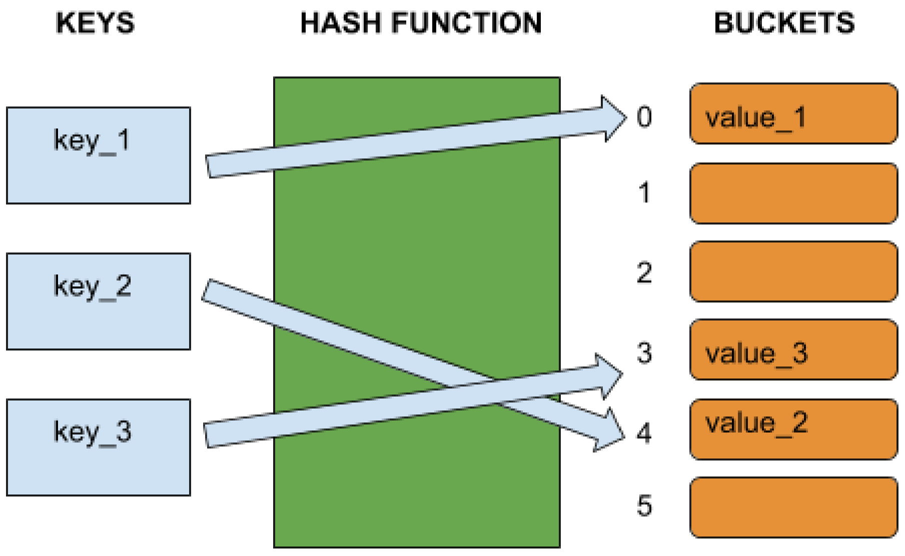

Figure 2.2.1 - **The Hash Table**

If $N$ items are to be stored in a fixed universal set $U$, containing all elements of $U$, and $M$ is the size of a hash table data structure where $M << U$, the hash function is computed as follows:[Erickson, Ref. 21]

$$H : U \rightarrow \{0,\ 1,\ \dots\ ,\ M - 1\}$$

The hash function $H$ computes any item $K \in U$ to a hash code or bucket index in the bucket array.[Erickson, Ref. 21] Consequently, *load factor* is a 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).[Javatpoint, Ref. 15] Load factor, α, is given by formula:

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

where $N$ is the number of expected occupied buckets, and $M$ is the hash table size.[CMU, Ref. 14] For hash table size $M$, the number of buckets (size) is optimally an array of prime number size that is at least 1.3 times $N$ given by the load factor.[USD, Ref. 20] Prime number length allows for less clustering of $N$ hash codes.

For keys most *commonly* expressed as of string-type, hash functions are an imperative tool to convert strings to some integer value. A standard technique utilized to hash keys is *modular hashing*.[Cornell U, Ref. 10] Essentially, a modulo divisor $S$ is selected such that $S \approx M$ or $S = M$ where $M$ is the integer size of the hash table, or number of buckets, consequently applied to function:

$$H(K)\ =\ K\ mod\ S$$

For hash functions, each character in the given key are converted and summated by its respective unique integer ASCII value, equating to variable $K$ given in the formula.[USWISC, Ref.5, Geeksforgeeks, Ref. 7, Shaffer, Ref. 8] In contrast, *multiplicative hashing*, another hash function, can similarly be implemented to compute hash codes. Generally speaking, multiplication is computed at a faster execution than division operators,[Cornell U., Ref. 10] hence multiplicative hashing may be a preferable alternative. With multiplicative hashing, hash codes can be calculated by:

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

for constant $c \in \mathbb{R}$, $0 < c < 1$.[ODS, Ref. 12, Deepika, Sheng, Ref. 13] Note for operation $\{K * c\}$ the decimal of the fraction part is taken. To obtain constant *c*, *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.[Deepika, Sheng, Ref. 13] In either hash functions, the value *S* is optimized when *S* is a prime number.

Ideally, each key passed through a hash function returns a unique index value, where each index stores the key's respective value. This is known as the *perfect hash function* as it allows for constant time value searches at a time complexity of O(1).[Cornell U, Ref. 10, Henry, Ref. 11] Hash function distribution uniformity of hash codes can be evaluated by the *chi-squared test* which is a statistical hypothesis test.[Weisstein, Ref. 4, Pigeon, Ref. 9] However, when distributions are not perfectly uniform, and multiple keys passed through the same hash function yield the same integer hash code by $H(K_1) = H(K_2)$, and $K_1 \neq K_2$ , a problem known as a *collision* arises,[Erickson, Ref. 21] which can be resolved by various methods, known as *collision handling*.

## 2.3 Collision Handling

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.[USD, Ref 20] 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 $\alpha$, accumulatively being a mathematical proof derived from the formulae:[USD, Ref. 20]

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

Consequently, to handle collisions, two common methods - *separate chaining* and *open addressing*, are commonly implemented in hash tables to mitigate collisions.[MTU, Ref. 17] Separate chaining 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.[ODS, Ref. 12] 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 retrieving operations.[MTU. 17] The theory of chaining is illustrated in Figure 2.3.1.

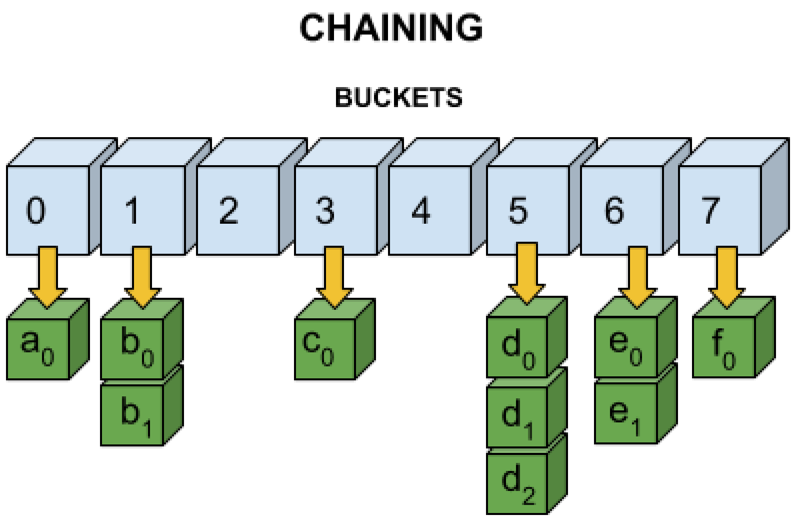

Figure 2.3.1

When assuming a hash table with load factor $\alpha = \frac{N}{M}$ and $0 \leq \alpha < 1$, execution time averages can be theoretically calculated based on average successful key searches.[USD, Ref. 20] As there is an average of $\frac{N-1}{M}$ keys within each bucket's array minus the targeted key, the average number of key comparisons for chaining is:[USD, Ref. 20]

$$S_\alpha = 1 + \frac{N-1}{2M} \approx 1 + \frac{\alpha}{2}$$

Unlike separate chaining, the open addressing collision handling method does not introduce a new data structure; instead, in the event of a collision, a search algorithm, known as *probing*, searches for the next available bucket (open address).[MTU, Ref. 17] The basic principle given by open addressing allows it to be applied in a wider variety of search algorithms; these include *linear probing*, *quadratic probing*, and *double hashing*.[MTU, Ref. 17]

Nonetheless, linear probing is the most common and relevant implementation of open addressing as it has the most efficient CPU cache utilization, thus resulting in high performance. As the iteration interval of linear probing is fixed to 1, more computational resources are conserved, allowing for easier computation compared to other open addressing methods.[Dunckelmann, Ref. 18]

Linear probing is performed starting at the position of the collision. Probing occurs until the last bucket, cycling back to the first, continuing until the bucket prior to the bucket of collision. Whenever an open bucket is detected, probing stops and the value is stored there, as shown in Figure 2.3.2.[Stanford, Ref.19]

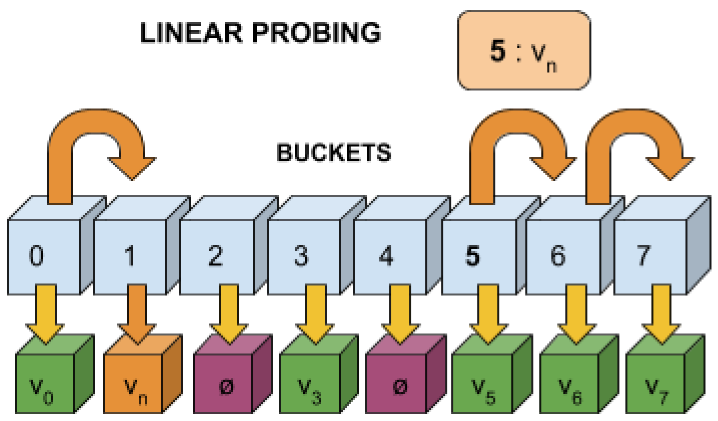

Figure 2.3.2

Hence, 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 $\alpha$ approaches 1, the number of collisions will simultaneously increase.[CMU, Ref. 14] For keys in a hash table, the amount of probe iterations is equal to the amount of probes required during initial insertion.[USD, Ref. 20] New key insertions will always increase load factor from $0 < i < \alpha$. During successful linear probing operations, the theoretical execution time can be similarly be calculated by the theoretical number of probes:[USD, Ref. 20]

$$S_\alpha \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)}\right)$$

## Experimental Methodology

### 3.1 Methodology

For purposes of this paper, datasets consisting of experimental data were obtained and analyzed. A modifiable hash table was programmed with varying hash functions, load factors and collision handling methods. This specific methodology was derived from the research question; by altering specific code within hash tables hence modified the dependent variables to exemplify data variations as a result of different collision handling methods. Simultaneously, aforementioned factors - hash functions and load factors - were imperative tools to examine consistencies of linear probing and chaining data across different platforms.

### 3.2 Controlled Variables

| **Variable** | **Description** | **Specification** |
|---|---|---|
| Computer and Operating System | Experiments conducted on a MacBook Pro 2017 13-inch. | Version: 11.01 macOS Big Sur<br>Processor: 3.1 GHz Dual-Core Intel Core i5<br>RAM: 16 GB 2133 MHz LPDDR3 |
| Integrated Development Environment (IDE) | Experiments conducted only using one IDE. | IDE: Visual Studio Code version 1.47.3<br>Commit: 91899dcef7b8110878ea59626991a18c8 |
| Programming Language | Experiments conducted only using one programming language. | Programming Language: Python 3.8.3 64-bit |
| Linear Probing | Same code snippet for linear probing implemented for all linear probing hash tables. | See section 8 Appendix. |

| | | |
|---|---|---|
| Chaining | Same code snippet for chaining implemented for all chaining hash tables. | See section 8 Appendix. |
| Hash Functions | Same code snippets for *modular hash functions* and *multiplicative hash functions* implemented for all respectively utilized hash tables. | See section 8 Appendix. |
| Load Factors | Same load factor variations for all respectively utilized hash tables, hence same hash table sizes respectively. *N was set to a constant 10000.* | Load Factor Values:<br>*Small* $\alpha \approx 0.1$, *M = 100003*<br>*Half* $\alpha \approx 0.5$, *M = 19991*<br>*Large* $\alpha \approx 0.9$, *M = 11113* |
| Randomization | Same randomization of hash code insertions utilized. | Randomization by utilizing the `random` Python module. See section 3.4 Randomized Datasets. |

Figure 3.2.1 - **Controlled Variables**

## 3.3 Dependent Variables

In these experiments, *execution time* was the main dependent variable being measured. A measurement of execution time efficiency is based **relative to** the execution time of other, similarly-purposed operations. Execution time was measured for two hash table operations, being *insertion* and *retrieval* of a key value pair, 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
#Prior insertions
inital_time = time.time()
#Insertion
execution_time = time.time() - inital_time
```

Figure 3.3.1 - Python 3.8.x Code Sample for *Execution Time*

A secondary dependent variable measured was bucket-size memory usage in bytes. Data space (refer to 2.1) occupied by a measured hash table's buckets was evaluated using Python's `getsizeof` function from the `sys` module. Likewise, less memory usage would achieve a more ideal hash table as it would conserve **relatively** more computational resources.

```
from sys import getsizeof
data_space = getsizeof(bucket)
```

Figure 3.3.2 - Python 3.8.x Code Sample for *Data Space*

## 3.4 Randomized Datasets

**Code details can be found in section 8 Appendix*

When examining collision insertion and retrieval operations in a hash table, firstly, a methodology to distribute random key value pairs into a hash table must be present. In this experimentation, *N* as a constant of 10000 (refer to load factor) pseudo-randomly generated string-type keys associated to a boolean value `False` from random length ranges 1 to 26 were inserted into hash tables. Dependent variables were measured for the insertion and retrieval of these *N* pseudo-random keys. Simultaneously, this methodology would guarantee in the product of well over one collision given by probabilities validated in the birthday paradox (section 2.3 Collision Handling).

## 3.5 The Hash Tables Programmed

To obtain data, a single dynamically adjustable hash table containing all relevant attributes under specific circumstances was programmed in order to observe data consistencies under constant controlled variables. The hash table consisted of modifications allowing it to have a multiplicative or modular hash function, varying load factors by adjusting *M* relative to

$N = 10000$, and finally having a collision handling method, all specified by initialization input. It is important to note that the optimal $M$ value was not implemented by $M = 1.3N$'s nearest prime, as load factor $\alpha$ was a controlled changing variable, relative to $N$ to match a specific load factor value, 0.1, 0.5, or 0.9. For hash tables which implemented chaining, arrays were used as the appended data structure. Figure 3.5.1 illustrates the procedure and structure in which these hash tables were programmed. The full code and details can be found in section 8 Appendix.

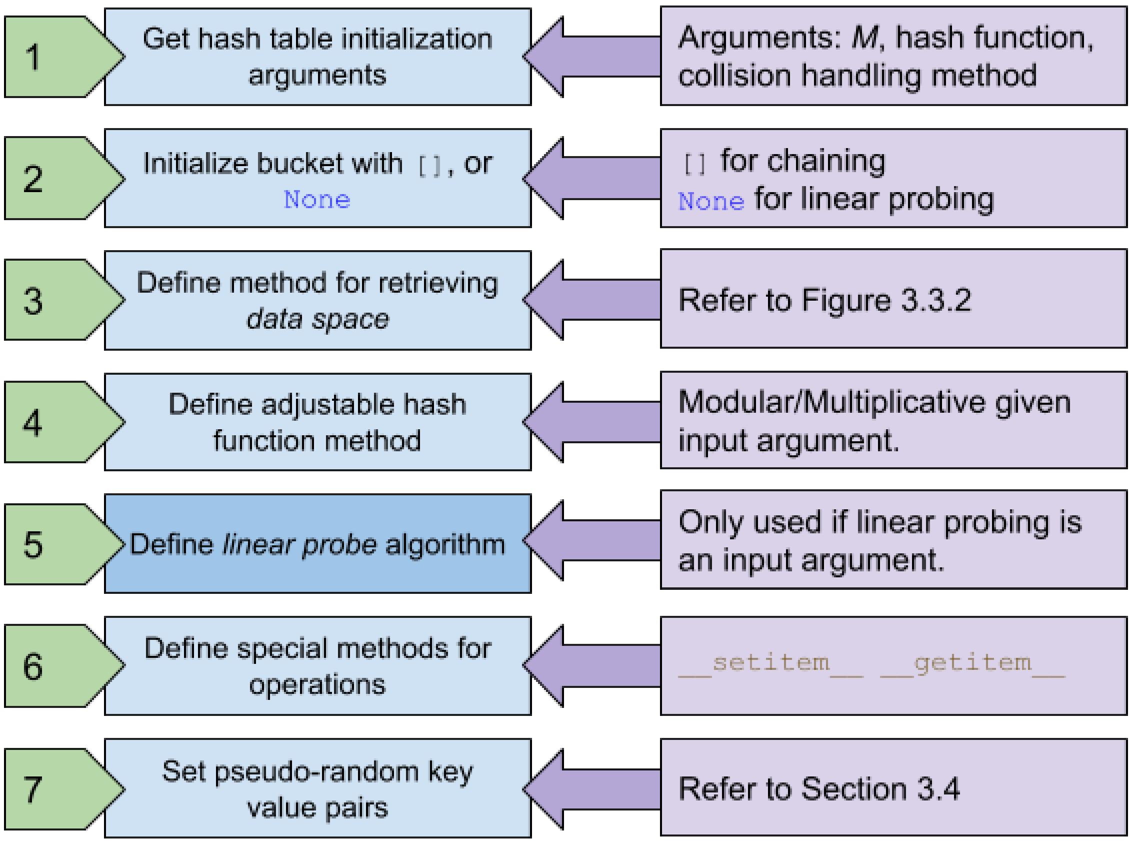
1
Get hash table initialization arguments
Arguments: *M*, hash function, collision handling method
2
Initialize bucket with [], or None
[] for chaining
None for linear probing
3
Define method for retrieving *data space*
Refer to Figure 3.3.2
4
Define adjustable hash function method
Modular/Multiplicative given input argument.
5
Define *linear probe* algorithm
Only used if linear probing is an input argument.
6
Define special methods for operations
__setitem__ __getitem__
7
Set pseudo-random key value pairs
Refer to Section 3.4

Figure 3.5.1 - Hash Table Structure

With reference to section 2.2, constant $c$ in the multiplicative hash function $H(K) = \lfloor S * \{K * c\} \rfloor$, $c \in \mathbb{R}$, $0 < c < 1$ was chosen to be $\frac{3^{40}}{2^{64}}$ as this constant concurrently agrees to its form restriction $c = \frac{x}{2^w}$ for $0 < x < 2^w$ where $w = 64$ as experiments are conducted on Python 3.8.3 64-bit. The multiplicative hash function is completed as shown in Figure 3.5.2.

```
from math import floor, modf
c = (3^40)/(2^64)
floor(S * modf(K * c)[0])
```

Figure 3.5.2 - Python 3.8.x Code Sample for a Multiplicative Hash Function

For divisor *S* selected in both modular and multiplicative hash functions, *S* was simply equal to *M* (hash table size), achieving optimal products for additional reason that *S* would be a prime number given that the initialization of *M* was incipiently prime. The modular hash function code is not covered in this section as it requires no special constants.

## 3.6 Experimental Procedure

The following steps describe the process in which the experiments were conducted in.

1. *Create 13 csv files, 12 for execution time, 1 for data space*
2. *Execute code shown in section 8 Appendix*
3. *Complete insertion and/or retrieving operation*
4. *Write execution times to csv file*
5. *Run program for execution time 100 times using an external executor file*
6. *Get data space with defined method*

These steps were completed for all implemented hash tables displayed with their abbreviations in Figure 3.6.1, the *X* describes the load factor being *S* (small), *H* (half), or *L* (large).

| **Implemented Hash Tables** |
|---|
| 1. Modular Hash Function, Chaining (*X*ModCH) |
| 2. Modular Hash Function, Linear Probing (*X*ModLP) |
| 3. Multiplicative Hash Function, Chaining (*X*MuCH) |
| 4. Multiplicative Hash Function, Linear Probing (*X*MuLP) |

Figure 3.6.1

## 3.7 Hypothesis

With recalling to the mathematical theory for average searches for both linear probing and chaining with respect to load factor, it is hypothesized that linear probing serves as a more efficient method of collision handling in hash tables in comparison to chaining, exclusively when load factor $\alpha < 0.5$. The reasoning by this is that chaining is able to store *N* values where $N > M$, therefore, $\alpha > 1$. However, this is not possible with linear probing as it operates specifically with the bucket array. Thus, when comparing linear probing to chaining, it must be assumed that $\alpha < 1$ and hence $N < M$. When this is true, for averages, $\alpha$ in chaining would simply be 1, thus fixing the average amount of searches in chaining to be $1 + \frac{[1]}{2} = 1.5$, a constant. Contrarily, if a linear probe search operation is of O(1) and thus $\alpha = 0$, the average $\frac{1}{2}(1 + \frac{1}{(1-[0])})$ would thus equate to 1 probe. However, if the load factor $\alpha = 0.5$, substituting $\alpha$ in $\frac{1}{2}(1 + \frac{1}{(1-[0.5])})$ equates to 1.5; thus, at $\alpha = 0.5$, it is assumed that search probe increases proportionally to load factor, thus, any load factor greater than 0.5 should require more searches than the 1.5 constant when $\alpha < 1$. Additionally, if $\alpha = 1$, the time complexity would be O(*N*), significantly less efficient than chaining.

## <u>**Experimental Data**</u>

### <u>**4.1 Graphical Data Visualization**</u>

For execution time performance, as depicted in the legends, red-lines are representative of linear probing, whereas blue-lines are representative of chaining; lighter variants of each colour signify hash tables with modular hash functions, whereas darker variants signify multiplicative hash functions. Each circular point on the line graph depicts a single execution of a hash table program, or a singular data point, adapted from quantitative data which can be found in section 8.2 Tabular Data. In contrast, memory usage is displayed in a horizontal bar-chart; memory usage for all hash tables of same load factors were equivalent, hence being conjoined into single bars respectively.

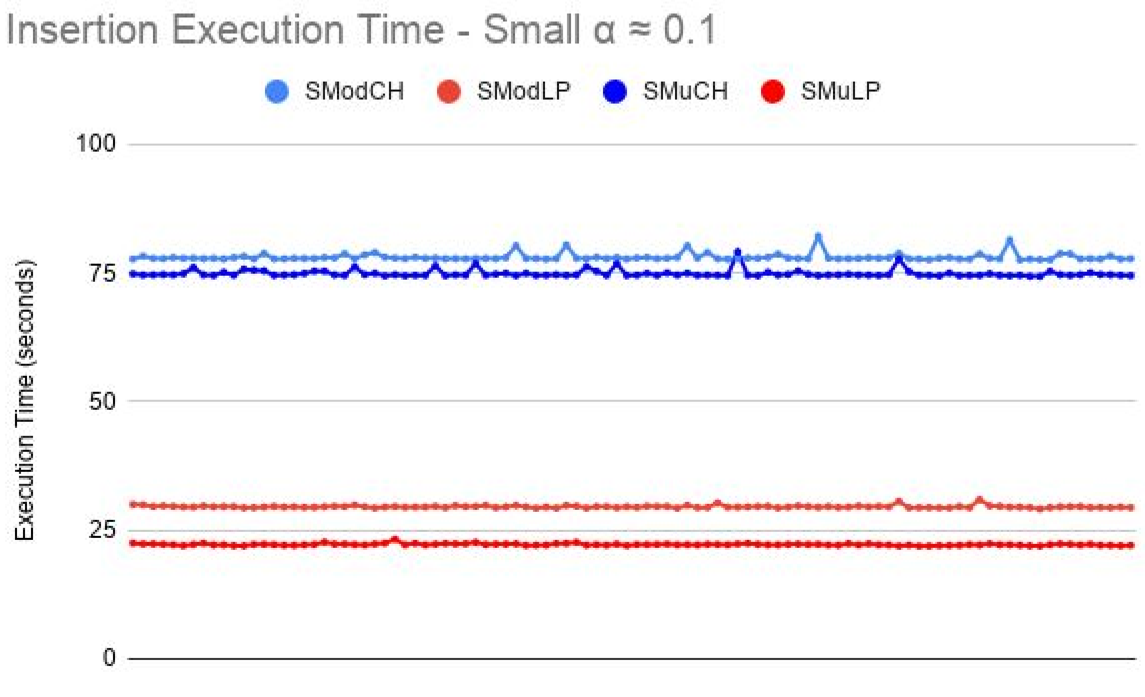

Figure 4.1.1

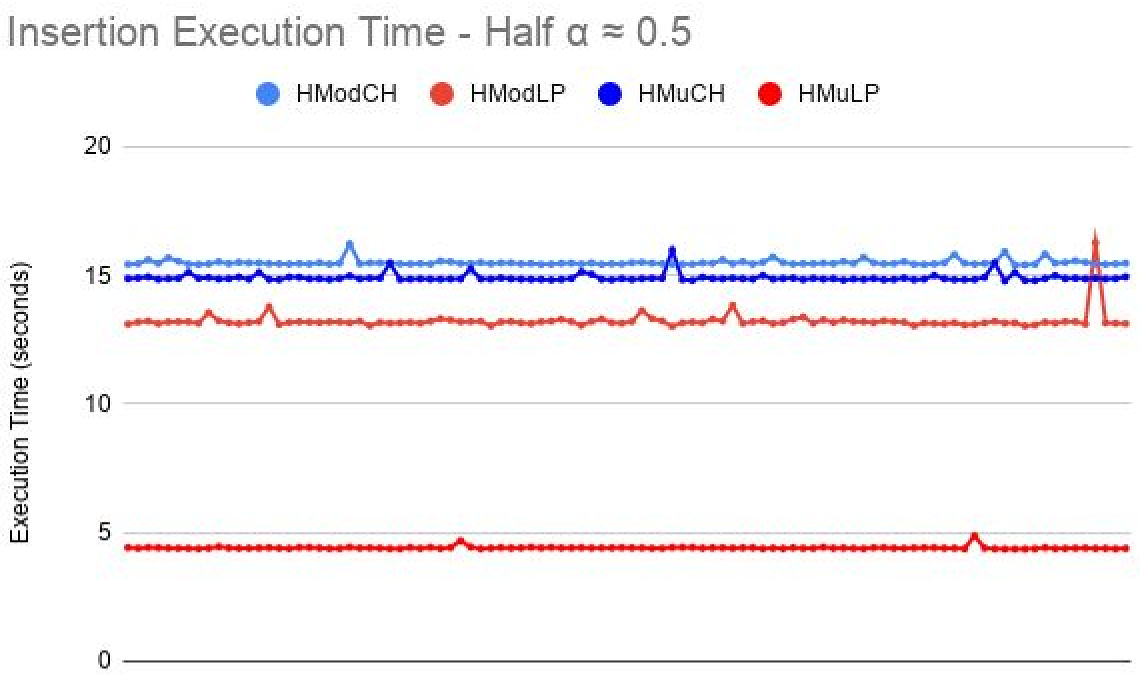

Figure 4.1.2

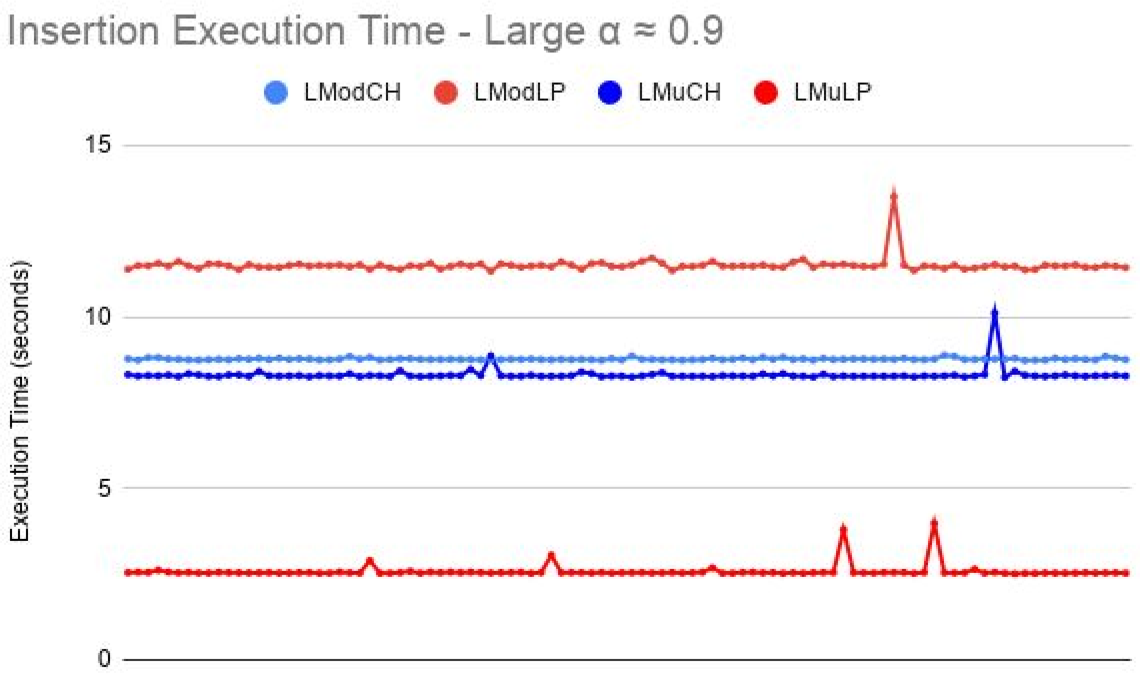

Figure 4.1.3

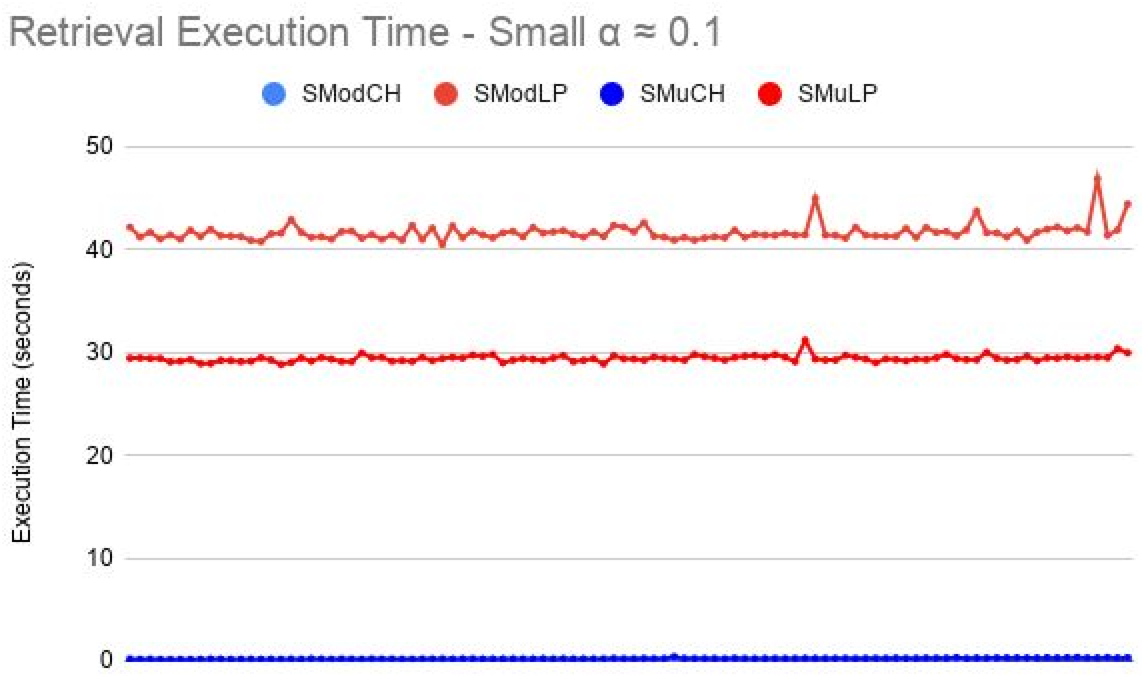

Figure 4.1.4

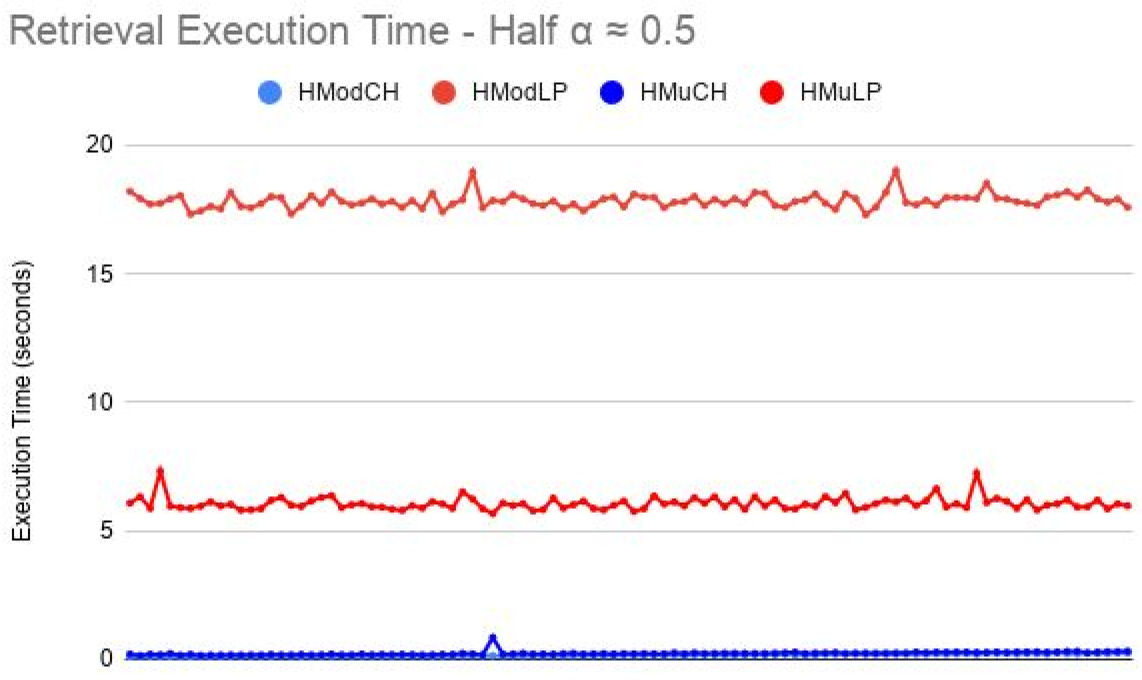

Figure 4.1.5

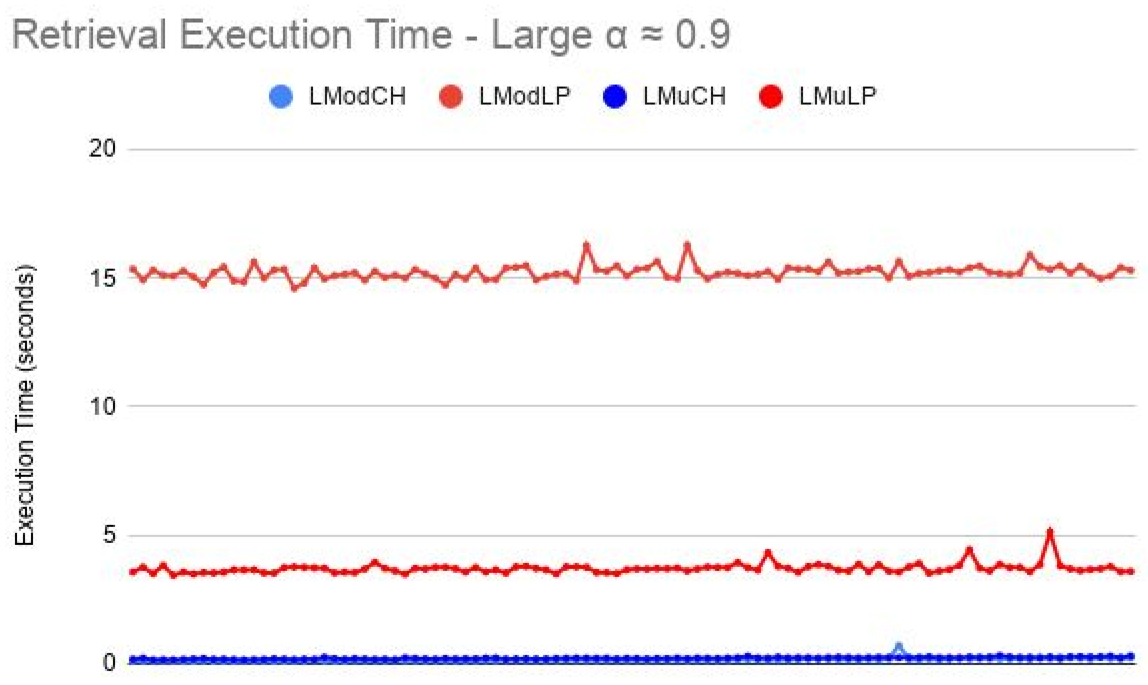

Figure 4.1.6

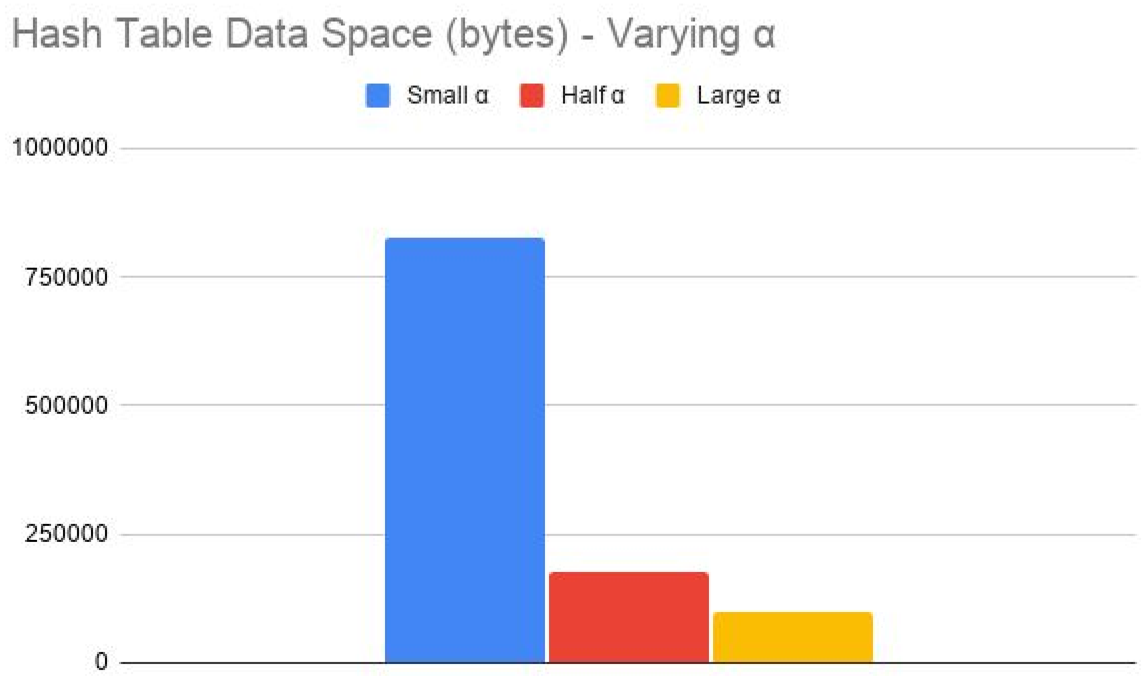

Figure 4.1.7

## 4.2 Data Analysis

Indicatively, in terms of modular hash function, the outcome experimental data for insertion show complete similarity to that of mathematically predicted in section 3.7 Hypothesis. As the modular hash function is conventionally selected to be the closest to a constant O(1) perfect hash function, it is justified that the modular hash function produced the perfectly predictable results. Focusing exclusively on the modular hash function data, when $\alpha \approx 0.1$ and thus $\alpha < 0.5$, linear probing evidently served as a more time efficient collision handling method, yet when $\alpha = 0.5$, data displays that performance of linear probing and chaining was approximately equal. The opposite held true for $\alpha > 0.5$ as hypothesized. Nonetheless, for analysis of experimental yields of execution time, an non-linear relationship between load factor and time efficiency was produced, differing that of hypothesized. Although load factors $\alpha \approx 0.1$ and $\alpha \approx 0.9$ are equidistant to $\alpha \approx 0.5$, the execution time difference did not scale proportionally. This is likely due to the variations in the $M$ value of the load factors; the numerator $N$ of the load factor did not maintain linear ratios to $M$ for varied hash table load factors, thus the execution time must have not been directly proportional, as shown by differences of dependent variables in Figures 4.1.1, 4.1.2, and 4.1.3. Furthermore, in contrast, the multiplicative hash function always produced more time efficient results, similarly reasoned as multiplication can be computed at a faster rate than division. However, unpredicted results were shown by the graph as the multiplicative hash function did not follow the same pattern as the modular hash function. Not only may this have been the cause of a poorly or too excellently chosen constant $c = 3^{40}$, for hash tables with a load factor $\alpha \to 1$, even perfect hash functions can expect worse-case time complexity at a high probability.[Erickson, Ref. 21]

In retrieving operations, evidently chaining was vastly more efficient at O(1) time complexity for execution time; relatively instantaneous. Accordingly, this result is logically deductible; as aforementioned in section 3.7 Hypothesis, given that the load factor $\alpha$ of chaining hash tables was always $\alpha < 1$ for averages, retrieving operations must have originated from buckets with chains of greater but roughly equal to $1$, or 0 key value pairs. However, as only buckets with greater but roughly equal to 1 lengthened chains will contain any key value pairs, the significantly lower average execution time advantage, in comparison to linear probing, is demonstrated by Figures 4.1.4, 4.1.5, and 4.1.6.

Finally, in terms of memory usage given by Figure 4.1.7, the data space utilized by all small and large hash tables displayed the same usage respectively. Hence, it is difficult to undermine a conclusion of memory usage efficiency comparing linear probing and chaining, as data displays absolute equality. Nonetheless, the memory usage does not account for initialized auxiliary space. From a theoretical perspective, linear probing would minimize this as the arrays initialized in chaining collision handling hash tables will foundationally allocate auxiliary space as each subarray is a dynamic array. However, this may have only been due to the usage of arrays; assuming the implementation of linked lists in place of arrays, memory usage of linear probing and chaining would logically be incomparably similar in terms of memory usage efficiency. Further analysis shows that data space is directly proportional to the $M$ value of the load factor, justified by the fixed memory usage of each bucket. Nonetheless, the experiments conducted vary $M$ to be of a value not set optimally by the load factor relationship $M = 1.3N$'s nearest prime. It is likely that utilizing the ideal $M$ value could make best usage of memory.

Lastly, the various errors discussed within this section are separated into random and systematic errors, outlined in Figure 4.2.1.

| **Random Errors** | **Systematic Errors** |
|---|---|
| Computer background executions. | Utilizing arrays in place of linked lists for chaining hash tables. |
| / | Multiplicative hash function *c* constant. |
| / | Unideal $M$ value for $M = 1.3N$ |

Figure 4.2.1 - **Error Chart**

## Further Research

### 5.1 Examination in Alternative Programming Languages

In addition to the variability present by differing hash functions and load factors of hash tables modified in these experiments, Python 3.8.x was notably the only programming language utilized. Consistencies or variabilities of linear probing and chaining collision handling data in hash tables coded in alternate programming languages could be a point of interest, especially low level programming languages distincting that of Python.

### 5.2 Alternate Hash Table Operations

Despite insertion and retrieval being primary operations in a hash table, deletion could be an operation of value to explore as chaining and linear probing handle deletion within a bucket array vastly differently. Linear probing cannot simply delete a value as size $M$ is fixed, thus it must tombstone, essentially resetting a bucket to its original null value, whereas chaining can simply delete a part of a chain. Thus, deletion operations could serve as an important distinction of collision handling method efficiency.

### 5.3 Exploring Additional Collision Handling Methods

For experiments conducted, examining collision handling in hash tables was exclusively limited to linear probing and chaining. However, aforementioned in section 2.2 Background Information, there exists a vast spectrum of collision handling methods; this includes other forms of open addressing, double hashing, and including, but also not limited to random hashing. Further research into other forms of collision handling could be of valuable comparison as it presents the likelihood of being a method of collision handling more efficient than both linear probing and chaining, under certain circumstances.

## **Conclusion**

In this paper, the effects of data structure implementation on efficiency and performance were analyzed for hash table collision handling methods. Linear probing and chaining were implemented in varying hash table functions, evaluated based on time complexity and memory usage. Summatively, data obtained following the engineering of an experimental methodology displays results for which exactly match that of hypothesized, exclusively for modular hash functions implemented in hash tables for reason that the modular hash function approximately reflects that of a O(1) perfect hash function. Assuming load factor is less than or equal to 1, linear probing serves as a more efficient method for collision handling when the load factor of the hash table is less than 0.5. Contrarily, if the load factor of the hash table is greater than 0.5, chaining displays greater time efficiency, proven mathematically by average searches, and experimentally. In addition, if the load factor of the hash table is ever set to be greater than 1, only chaining will function.

When given any hash function, the importance of collision handling is unconditionally imperative, not limited to applications within hash tables. Most things alike, hash functions similarly cannot be ideal, minimizing operations to O(1), while simultaneously perfectly distributing hash codes. 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 not limitedly the implementation of hash tables for computer science, but in additional fields comprising cryptography, mathematics, and beyond.

## References

1. Ahlawat, Abhishek. "Asymptotic Notations." *Studytonight.com*, © 2020 Studytonight Technologies Pvt. Ltd., www.studytonight.com/data-structures/aysmptotic-notations.
2. Ahlawat, Abhishek. "Time Complexity of Algorithms." *Studytonight.com*, © 2020 Studytonight Technologies Pvt. Ltd., www.studytonight.com/data-structures/time-complexity-of-algorithms.
3. Ahlawat, Abhishek. "Space Complexity of Algorithms." *Studytonight.com*, © 2020 Studytonight Technologies Pvt. Ltd., www.studytonight.com/data-structures/space-complexity-of-algorithms.
4. Weisstein, Eric. "Chi-Squared Test." *Wolfram MathWorld*, Wolfram Research, mathworld.wolfram.com/Chi-SquaredTest.html.
5. CS 367-3. "Notes on Hashing." *CS 367-3 - Hashing*, University of Wisconsin, pages.cs.wisc.edu/~siff/CS367/Notes/hash.html.
6. "Data Structure and Algorithms - Hash Table." *Tutorialspoint*, Tutorialspoint, www.tutorialspoint.com/data_structures_algorithms/hash_data_structure.htm.
7. "Hashing Data Structure." *GeeksforGeeks*, GeeksforGeeks, www.geeksforgeeks.org/hashing-data-structure/.
8. 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.
9. 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/.

10. "CS 3110 Lecture 21 Hash Functions." *Lecture 21: Hash Functions*, Cornell University, www.cs.cornell.edu/courses/cs3110/2008fa/lectures/lec21.html.
11. Henry, Ed. "Hashing in Python." *Ed Henry's Blog*, © Ed Henry 2019, edhenry.github.io/2016/12/21/Hashing-in-Python/.
12. "5.1 ChainedHashTable: Hashing with Chaining." *Open Data Structures*, www.opendatastructures.org/versions/edition-0.1f/ods-java/5_1_ChainedHashTable_Hashin.html.
13. "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/.
14. "Sets & Maps Hash Tables." *15-121 Introduction to Data Structures*, Carnegie Mellon University, www.cs.cmu.edu/~tcortina/15-121sp10/Unit08B.pdf.
15. "Load Factor in HashMap - Javatpoint." *Javaatpoint*, Javaatpoint, www.javatpoint.com/load-factor-in-hashmap.
16. McDowell, Gayle Laakmann. "Data Structures: Hash Tables." *YouTube*, Hackerrank, 27 Sept. 2016, www.youtube.com/watch?v=shs0KM3wKv8.
17. *Hash Table Collision Handling*. Michigan Technology University, www.csl.mtu.edu/cs2321/www/newLectures/17_Hash_Tables_Collisions.html.
18. "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.

19. "Linear Probing." *Sanford Class Archives*, Stanford University, web.stanford.edu/class/archive/cs/cs166/cs166.1166/lectures/12/Small12.pdf.
20. "Lecture 16 Hashing." *University of San Diego*, University of San Diego, http://cseweb.ucsd.edu/~kube/cls/100/Lectures/lec16/lec16.pdf.
21. Erickson, Jeff. "Lecture 12: Hash Tables." *Jeff Erickson CS*, University of Illinois, 2015, jeffe.cs.illinois.edu/teaching/datastructures/notes/12-hashing.pdf.

## Appendix

### 8.1 Hash Table Code

The following is the code programmed in Python 3.8.3 64-bit utilized in the experiments conducted, and obtained data was written to csv files. The majority of section 8.1 will be in reference to section 3 Experimental Methodology.

```
import time #Measures execution time in seconds
from sys import getsizeof #Measures memory usage in bytes
from math import floor, modf #For multiplicative Hashing only
import random #For randomization of strings
import string #For random strings

class HashTable: #Hash Table Specification
    def __init__(self, M, hf, ch):
        self.size = M #Prime number at large size
        self.ch = ch
        self.hf = hf
        if ch == "lp": #Check the argument passed in
            self.bucket = [None for generation in range(self.size)] #[]
for chaining, None for linear probing
        elif ch == "ch":
            self.bucket = [[] for generation in range(self.size)]

    def print_bucket(self):
        print(self.bucket)

    def memory_usage(self):
        return getsizeof(self.bucket)

    def hash_function(self, key):
        hash_code = 0
        for character in key:
            hash_code += ord(character)
        if self.hf == "mod": #Argument check
```

```
            return hash_code % self.size
        elif self.hf == "mu":
            constant = (3^40)/(2^64)
            return floor(self.size * (modf(hash_code * constant)[0]))

    def probe_range(self, index):
        return [*range(index, len(self.bucket))] + [*range(0, index)]

    def linear_probe(self, key, index):
        for index in self.probe_range(index):
            if self.bucket[index] is None:
                return index
            if type(self.bucket[index]) != bool:
                if self.bucket[index][0] == key:
                    return index
        raise Exception("Hash Table is full")

    def __setitem__(self, key, value):
        hash_code = self.hash_function(key)
        if self.ch == "ch": #Chaining insertion
            found = False
            for index, element in enumerate(self.bucket):
                if len(element) == 2 and element[0] == key:
                    self.bucket[hash_code][index] = (key, value)
                    break
            if not found:
                self.bucket[hash_code].append((key, value))
        elif self.ch == "lp": #Linear probing insertion
            if self.bucket[hash_code] is None:
                self.bucket[hash_code] = (key, value)
            else:
                self.bucket[self.linear_probe(key, hash_code)] = (key,
value)

    def __getitem__(self, key):
        hash_code = self.hash_function(key)
        if self.ch == "ch": #Chaining retrieval
            for elements in self.bucket[hash_code]:
                if elements[0] == key:
                    return elements[1]
            raise Exception("Key does not exist in Hash Table")
        elif self.ch == "lp": #Linear probing retrieval
            if self.bucket[hash_code] is None:
                return None
            for index in self.probe_range(hash_code):
                if self.bucket[index] is None:
                    return None
                if type(self.bucket[index]) != bool:
                    if self.bucket[index][0] == key:
                        return self.bucket[index][1]

N = 10000 #Fixed constant

letters = string.ascii_lowercase
keys = [] #Keeps track what keys there are
for i in range(N):
    keys.append("".join(random.choice(letters) for i in
range(random.randint(1,26)))) #N Random keys

def insert(hash_table):
    global keys
    for key in keys:
        hash_table[key] = False #Sets N random keys associated to False

def retrieve(hash_table):
    global keys
    for key in keys:
        print(hash_table[key])

operations = (
    [100003,"mod","ch"],
    [19991,"mod","ch"],
    [11113,"mod","ch"],
    [100003,"mod","lp"],
    [19991,"mod","lp"],
    [11113,"mod","lp"],
    [100003,"mu","ch"],
    [19991,"mu","ch"],
    [11113,"mu","ch"],
    [100003,"mu","lp"],
    [19991,"mu","lp"],
    [11113,"mu","lp"]
    ) #Specified hash table settings

directories = (
    r"/Users/mingchungxia/Desktop/IB Extended Essay Data/Version 3/ModCH 
Execution Time/SModCH.csv",
    r"/Users/mingchungxia/Desktop/IB Extended Essay Data/Version 3/ModCH 
Execution Time/HModCH.csv",
    r"/Users/mingchungxia/Desktop/IB Extended Essay Data/Version 3/ModCH 
Execution Time/LModCH.csv",
    r"/Users/mingchungxia/Desktop/IB Extended Essay Data/Version 3/ModLP 
Execution Time/SModLP.csv",
    r"/Users/mingchungxia/Desktop/IB Extended Essay Data/Version 3/ModLP 
Execution Time/HModLP.csv",
    r"/Users/mingchungxia/Desktop/IB Extended Essay Data/Version 3/ModLP 
Execution Time/LModLP.csv",
    r"/Users/mingchungxia/Desktop/IB Extended Essay Data/Version 3/MuCH 
Execution Time/SMuCH.csv",
    r"/Users/mingchungxia/Desktop/IB Extended Essay Data/Version 3/MuCH 
Execution Time/HMuCH.csv",
    r"/Users/mingchungxia/Desktop/IB Extended Essay Data/Version 3/MuCH 
Execution Time/LMuCH.csv",
    r"/Users/mingchungxia/Desktop/IB Extended Essay Data/Version 3/MuLP 
Execution Time/SMuLP.csv",
    r"/Users/mingchungxia/Desktop/IB Extended Essay Data/Version 3/MuLP 
Execution Time/HMuLP.csv",
    r"/Users/mingchungxia/Desktop/IB Extended Essay Data/Version 3/MuLP 
Execution Time/LMuLP.csv",
)

#Execution Time
for i, op in enumerate(operations):

    hash_table = HashTable(op[0],op[1],op[2]) #Hash table settings

    with open(directories[i], "a") as file:
        #Insertion
        initial_time = time.time()
        insert(hash_table)
        insertion_time = time.time() - initial_time

        #Retrieval
        initial_time = time.time()
        retrieve(hash_table)
        retrieval_time = time.time() - initial_time
        file.write(f"\n{insertion_time:.20f},{retrieval_time:.20f}")

    print(f"Completed {i} operation.")

#Memory Usage
for i, op in enumerate(operations):

    hash_table = HashTable(op[0],op[1],op[2])

    insert(hash_table)

    print(hash_table.memory_usage())
```

Figure 8.1.1 - **Dynamically Adjustable Hash Table**

## **8.2 Tabular Data**

The following tables display tabular data of average execution times and memory usage. Execution times are averages based on 100 data points.

| **Hash Table** | Small α | Half α | Large α |
|---|---|---|---|
| ModCH | 78.07 | 15.48 | 8.76 |
| ModLP | 29.58 | 13.22 | 11.50 |
| MuCH | 74.88 | 14.90 | 8.31 |
| MuLP | 22.23 | 4.41 | 2.58 |

Figure 8.2.1 - **Tabular Data of Insertion** (seconds)

| **Hash Table** | Small α | Half α | Large α |
|---|---|---|---|
| ModCH | 0.15 | 0.16 | 0.16 |
| ModLP | 41.66 | 17.84 | 15.21 |
| MuCH | 0.22 | 0.23 | 0.22 |
| MuLP | 29.40 | 6.06 | 3.71 |

Figure 8.2.2 - **Tabular Data of Retrieval** (seconds)

| **Load Factor** α | **Data Space** |
|---|---|
| Small α | 824456 |
| Half α | 178016 |
| Large α | 98616 |

Figure 8.2.3 - **Tabular Data of Memory Usage** (bytes)