<https://ib.compscihub.net/wp-content/uploads/2016/07/EE-for-CS-guidance-2016.pdf>

FOLLOW CRITERIA!

\*edit 2.1, expand on chi-squared distribution, concise word count, *move 4.1 to appendix*?

**Investigating the Effects of Data Structure Implementation on**

**Performance and Efficiency**

**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

---

X words **Word count will exclude redundant words - 4000 WORD LIMIT!!**

Mingchung Xia - Charles P. Allen High School

**Table of Contents**

**1. Introduction……………………………………………………………………………………p**

**2. Context…………………..……………………………………………………………………..p**

**2.1 Asymptotic Notation**…………………………………....……………………………p

Time Complexity…….….………………………..………………………………..

Space Complexity………………………………………………………………….

**2.2 Hash Tables**…………………………………………………………………………..p

Load Factor………………………………………………………………………..

Hash Functions………………………………………...…………….…………….

Chi-Squared Distribution…………………………………………………………..

**2.3 Collision Handling**…………………………………………………………………...p

Chaining………………………………………………………..….……………….

Linear Probing……………………………………………………………………..

**3. Experimental Methodology………………………………………………...………………...p**

**3.1 Methodology**…………..……………………………………………………………..p

**3.2 Controlled Variables**…….………………………….………………………………..p

**3.3 Dependent Variables**………………………………………………………………....p

**3.4 Datasets**…………………………………………………………….………………....p

**3.5 The Hash Tables Programmed**……………………………………………………...p

**3.6 Experimental Procedure**……………………………………………..……………...p

Implemented Hash Tables………………………………………………………….

**3.7 Hypothesis**…………………………………………………………………………….p

**4. Experimental Data….…………………………………………………………………………p**

**4.1 Tabular Data**…………….....………………………………………………………...p

**4.2 Graphical Data Visualization**……………………………………………………….p

**4.3 Data Analysis**……………………………………………………….………………..p

Analyzing Collisions and Non-Collisions………………………………………….

Analyzing Linear Probing and Chaining…………………………………………...

Justifying Experimental Accuracy………………………………………………….

**4.4 Errors**………………………………………………………….……………………...p

**5. Further Research……………………………………………………………………………...p**

**5.1 Examination in Alternate Programming Languages**……………………………..p

**5.2 Adjusting Collision Handling Equity**……………………………………………...p

**5.3 Exploring Additional Collision Handling Methods**……………………………….p  
**6. Conclusion…………………….……………………………………….......…………………..p**

**7. References………………………………………………………………………..……………p**

**8. Appendix………………………………………………………………………..……………..p**

**8.1 Hash Table Code**……………………………………………………………….…….p

Explanation of Code………………………………………………………………..

Specified Datasets…………………………………………………………………..

**8.2 Unused Graphical Data Visualization**………………………………………………p

**Red Background = Likely to remove**

**Yellow Background = Consider removing**

**Green Background = Possible Re-Write**

**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. The simple, yet highly-functional nature of hash tables 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 software engineering and generalistic programming as differentiating the more efficient method would evidently produce more favourable short term and long term outcomes. The utilization of hash tables is present in, but not limited to message digests, file systems, password verifications, pattern association, and developing the source of new programming languages; therefore, 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**

When it comes to any analysis of an algorithm in computer science, a commonly understood methodology is to utilize and implement mathematical tools to approach such an analysis. Given that algorithms directly affect data structures, applying varying algorithms can be used to measure efficiencies of varying data structures, including hash tables.

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.[1] Thus, time complexities and space complexities fundamental deviations of the value asymptotic notation.

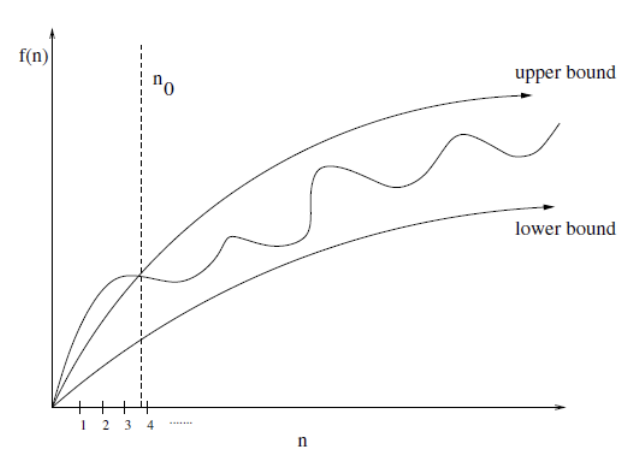


Figure 2.1.1

For valid bound-behaviour ranges , 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 function *f* (*n*) will never exceed the specified time complexity bounded by Big O notation for input *n*.[2] For example, given a 1-dimensional array comprised of *n* elements, array = [0, 1, 2, 3, 4 ... n-1, n], Big O analysis, for an iterative search algorithm, would be O(*n*) as the worst-case scenario would be an iteration of all *n* elements in the 1-dimensional array. In terms of data structures, time complexities measured by Big O analysis are dependent by algorithms and/or operations implemented on them.

When utilizing some algorithm and/or data structure, it is apparent that a system’s memory usage must be utilized; similarly, mathematics serves a major function for *Space Complexity* analysis, which refers to the amount of memory an algorithm requires for execution, formulated as *Space Complexity = Auxiliary Space + Input Space*, where auxiliary space is defined to be the excess temporary space allocated by a system.[3] The memory utilized may be the cause of instructional space, environmental stacks, or data space; however, when calculating space complexity, generally only the *data space* is concerned, which is the memory utilized by all variables and constants. As with time complexity, space complexity, likewise, should ideally be kept to a minimum.

**2.2 Hash Tables**

The *hash table* is a complex data structure which utilizes an associative array data type in order to format values such that a key returns its associated, assigned value.[16] For example, if a hash table’s key *key\_1*, associated to some value *value\_1*, *value\_1* will be returned when *key\_1* is called upon. This is completed by applying a *hash function*, which computes an *hash code*, to determine the memory location of the requested value in a *bucket*, visualized in Figure 2.2.1. For high-level programming languages, value objects stored in buckets are generally implemented by utilizing arrays.[16]



Figure 2.2.1

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).[15] Load factor, 𝛼, is given by formula 𝛼where *n* is the number of occupied buckets, and Hs is the hash table size.[14]

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*.[10] Essentially, a modulo divisor *S* is selected such that *S ≈ Hs* or *S* = *Hs* where *Hs*is the integer size of the hash table, or number of buckets, consequently applied to function . Often, for divisor *S*, optimal results are achieved when *S* is a prime number whose value is not close to 2n for positive integers *n*.[13] 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.[5,6,7,8]

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,[10] hence multiplicative hashing may be a preferable alternative. With multiplicative hashing, hash codes can be calculated by for constant .[12,13]Note for operation {*K \* c*} the decimal of the fraction part is taken. Unlike modular hashing, the selected *S* in H(*K*) is typically chosen to be of form 2n for positive integers *n*, nonetheless, this is not necessary.[13] To obtain constant *c*, *c* is generally restricted to be a rational number where *c* can be written in form for 0 < *x* < where *w* is the machine word size in bits.[12,13]

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).[10,11] Hence, to test hash functions, distribution uniformity of hash codes can be evaluated by the *chi-squared test* abbreviated by χ2, which is fundamentally a statistical hypothesis test.[4] For hash tables, the expected number of items in bucket, represented by *Ei* is for *M* items hashed to *N* buckets. The number of items in buckets *i* is denoted by *Oi* , which is dependent on the evaluated hash function. Hence, the formula is .[9,10] However, due to the minor instances, represented by outliers within the chi-squared distribution, of unique keys yielding the same integer hash code, issues within a hash table arise as it must be ensured that each key returns its correct associated value. This is known as a *collision*, 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. To handle collisions, two common methods - *separate chaining* and *open addressing*, are commonly implemented in hash tables to mitigate collisions.[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.[12,17] 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, comparisons can be made to the requested key to return a correct value for retrieving operations.[17]



Figure 2.3.1

In Figure 2.3.1, showing an example of chaining in hash tables, buckets with ≥ 2 values are buckets [1], [5], and [6] in which collisions have occurred. Buckets [2] and [4] which do not contain any values have simply not been hashed, and single value buckets [0], [3], and [7] have been hashed but have not been subject to collisions. Subsequently, in the occurrence of a collision, additional values are chained resulting in that bucket having multiple values which can be called upon given an associated key.

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).[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*.[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.[18,19]

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.[19]



Figure 2.3.2

Here, some key hashed to 5 attempts to store value vn at bucket [5]. However, as [5] is occupied by v5, linear probing cycles until bucket [1] which is an open bucket. Hence, despite the original hash being 5, the key is associated to the value in bucket [1] instead. With reference to load factor, load factor will always be in range 0 ≤ 𝛼 ≤ 1 as any 𝛼 > 1 will result in an overload of the amount of buckets available for key value pairs. Hence, as 𝛼 approaches 1, the number of collisions will simultaneously increase.[14]

**Experimental Methodology**

**3.1 Methodology**

For purposes of this paper, datasets consisting of experimental data were obtained and analyzed. Eight hash tables were programmed with varying hash functions, bucket sizes, and collision handling methods. This specific methodology was derived from the research question; by altering specific code within each hash table hence modified the dependent variables to exemplify data variations as a result of different collision handling methods. Simultaneously, aforementioned factors - hash functions and bucket sizes - 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  Processor: 3.1 GHz Dual-Core Intel Core i5  RAM: 16 GB 2133 MHz LPDDR3 |
| Integrated Development Environment (IDE) | Experiments conducted only using one IDE. | IDE: Visual Studio Code version 1.47.3  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. |
| Bucket Sizes | Same bucket sizes, *small* and *large* implemented for all respectively utilized hash tables. | Bucket Sizes:  Small ⇒ 10 000  Large ⇒ 1 000 000 |
| Key Value Pairs | Same key value pairs to control collisions and non collisions | See section 3.4 Datasets.  See section 8 Appendix. |

**3.3 Dependent Variables**

In these experiments, *execution time* was the main dependent variable being measured. Higher execution times would signify a less efficient operation in relation to a certain hash table, whereas lower executions times would contrarily signify a more efficient operation. Consequently, a measurement of efficiency based on execution time is **relative to** the execution time of other, similarly-purposed operations. Execution time was measured for three fundamental hash table operations, being *insertion*, *retrieving*, and *deletion* of a key value pair in seconds. The time was evaluated 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()  #Operation  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 Datasets**

*Exact strings and values can be found in section 8 Appendix.*

In order to stimulate collisions in a hash table, 30 string-type permutations of key "lvphcs" (inclusive) were operated with. When summated by all its character’s ASCII values, the resultant hash code is 656. Hence, all unique string permutations of "lvphcs" would similarly collide at this hash code value; this is one of the essential controlled variables discussed in section 3.2 and the implications of hash code 656 will be further discussed. Similarly, to evaluate non collision operations in a hash table, 30 string-type keys were operated with. These strings were ensured to have differing and unique hash codes to prevent any of these keys resulting in collision. All aforementioned keys were associated to a boolean value. For collisions, all except the operating key was set to False whereas the operating key "lvphcs" was set to True. As all non collision keys would result in unique hash codes, each key was simply associated with the value True.

**3.5 The Hash Tables Programmed**

To gain sufficient data, eight hash tables were programmed in order to observe data consistencies for uniquely implemented hash tables under constant controlled variables. The eight hash tables consisted of four hash tables that utilized chaining, and four hash tables that utilized linear probing. In each set, the hash tables were programmed to contain either modular or multiplicative hash functions, and either ‘small’ or ‘large’ bucket sizes (refer to 3.2). 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.



Figure 3.5.1 - Hash Table Structure

In constituent 1 of Figure 3.5.1, varying initializations per buckets were completed for three scenarios of hash table examinations. [] (an empty 1-dimensional array) was initialized for hash tables which implemented chaining as a means of collision handling. Both None and False were initialized for hash tables utilizing linear probing. When initializing False for all except bucket [655], as discussed in section 3.4, the bucket previous to the hash code 656 derived from "lvphcs" would simulate O(*n*) for collisions during linear probing operations as described in constituent 5.

With reference to section 2.2, constant *c* in the multiplicative hash function , *c* ∈ ℝ, 0 < *c* < 1 was chosen to be as this constant concurrently agrees to its form restriction for 0 < *x* < where *w* = 64 as experiments are conducted on Python 3.8.3 64-bit. This hash function is completed as shown in Figure 3.5.2. The modular hash function code is not covered in this section as there are no special constants.

|  |
| --- |
| from math import floor, modf  c = (3^40)/(2^64)  floor(S \* modf(K \* c)) |

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

**3.6 Experimental Procedure**

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

1. *Create 7 csv files for each hash table - 6 for execution time, 1 for data space*
2. *Set a hash table object*
3. *Apply prior key value insertions*
4. *Complete operation - insertion, retrieving, and deletion*
5. *Write execution time to csv file*
6. *Run program for execution time 100 times using an external executor file*
7. *Write data space for both collisions and non collisions in data space csv file*

These steps were completed for all implemented hash tables displayed with their abbreviations in Figure 3.6.1.

|  |
| --- |
| **Implemented Hash Tables** |
| 1. Small Bucket Size, Modular Hash Function, Chaining (SMC) |
| 1. Large Bucket Size, Modular Hash Function, Chaining (LMC) |
| 1. Small Bucket Size, Multiplicative Hash Function, Chaining (SMuC) |
| 1. Large Bucket Size, Multiplicative Hash Function, Chaining (LMuC) |
| 1. Small Bucket Size, Modular Hash Function, Linear Probing (SMLP) |
| 1. Large Bucket Size, Modular Hash Function, Linear Probing (LMLP) |
| 1. Small Bucket Size, Multiplicative Hash Function, Linear Probing (SMuLP) |
| 1. Large Bucket Size, Multiplicative Hash Function, Linear Probing (LMuLP) |

Figure 3.6.1

**3.7 Hypothesis**

Given that chaining operates only at a specifically hashed bucket, chaining should be of greater efficiency when compared to linear probing, exclusively for hash tables of greater bucket sizes, as the operations would occur in a smaller array within the collided bucket at O(*n*) in contrast to linearly probing at O(*n*) for the entire large-size bucket’s array. Additionally, for insertions, it is probable that linear probing will be of higher efficiency for non collision insertions; unlike chaining, linear probing would not need to manipulate any other data structures other than the bucket’s array itself, and would henceforth be directly inserted at the correct hash code. Furthermore, it must be considered that, according to the chi-squared distribution (refer to section 2.2) **FIND A VALUE TO PROVE HERE** stating the unlikelihood of many collisions occurring at a single bucket, thus increasing the probability of O(1) constant operations rather than O(*n*) for linear probing. Therefore, it is hypothesized that linear probing will serve as a more efficient method for collision handling in hash tables when assuming a quality hash function.

**Experimental Data**

**4.1 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** | **Average Execution Time (s)**  Collisions | **Average Execution Time (s)**  Non Collisions |
| SMC | 9.10E-04 | 9.20E-04 |
| LMC | 8.99E-02 | 8.76E-02 |
| SMuC | 1.11E-03 | 9.96E-04 |
| LMuC | 8.94E-02 | 9.33E-02 |
| SMLP | 3.26E-03 | 5.90E-06 |
| LMLP | 2.96E-01 | 1.49E-05 |
| SMuLP | 2.01E-03 | 2.41E-06 |
| LMuLP | 1.86E-01 | 2.07E-05 |

Figure 4.1.1 - **Insertion Data**

|  |  |  |
| --- | --- | --- |
| **Hash Table** | **Average Execution Time (s)**  Collisions | **Average Execution Time (s)**  Non Collisions |
| SMC | 3.89E-05 | 4.40E-05 |
| LMC | 3.86E-05 | 3.85E-05 |
| SMuC | 4.63E-05 | 4.60E-05 |
| LMuC | 4.11E-05 | 3.86E-05 |
| SMLP | 3.31E-03 | 5.63E-04 |
| LMLP | 2.82E-01 | 4.58E-02 |
| SMuLP | 2.78E-03 | 4.14E-04 |
| LMuLP | 1.84E-01 | 4.40E-02 |

Figure 4.1.2 - **Retrieving Data**

|  |  |  |
| --- | --- | --- |
| **Hash Table** | **Average Execution Time (s)**  Collisions | **Average Execution Time (s)**  Non Collisions |
| SMC | 1.26E-05 | 5.14E-06 |
| LMC | 1.53E-05 | 1.07E-05 |
| SMuC | 9.48E-06 | 5.69E-06 |
| LMuC | 1.49E-05 | 1.23E-05 |
| SMLP | 3.25E-03 | 2.59E-04 |
| LMLP | 2.79E-01 | 4.46E-02 |
| SMuLP | 1.95E-03 | 2.41E-04 |
| LMuLP | 1.87E-01 | 4.29E-02 |

Figure 4.1.3 - **Deletion Data**

|  |  |
| --- | --- |
| **Hash Table** | **Data Space / Memory Usage (bytes)** |
| SMC | 87616 |
| LMC | 8697456 |
| SMuC | 87616 |
| LMuC | 8697456 |
| SMLP | 87616 |
| LMLP | 8697456 |
| SMuLP | 87616 |
| LMuLP | 8697456 |

Figure 4.1.4 - **Memory Usage**

**4.2 Graphical Data Visualization**

Execution time performance of the various hash tables has been represented by the following line charts. As depicted in the legends, red-lines are representative of linear probing, whereas blue-lines are representative of chaining. Each circular point on the line graph displays a single data point, or a single execution of the hash table program. Whilst there were an accumulated 24 graphs adapted from raw data, only 12 most relevant graphs are shown in this section, the remaining graphs can be found in section 8 Appendix. Unlike for execution time, memory usage has been displayed in a horizontal bar-chart. As shown in Figure 4.1.4, memory usage for all hash tables of size small and/or size large were equivalent, hence being conjoined into a single bar respectively.

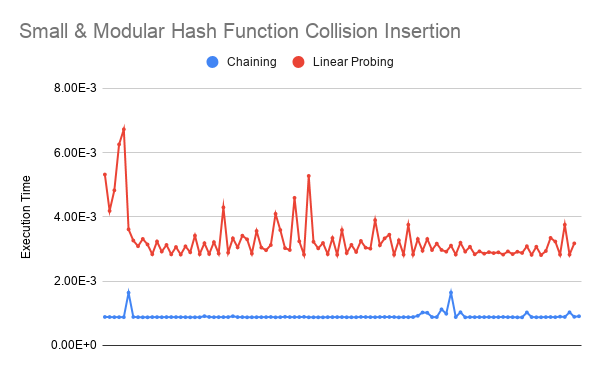
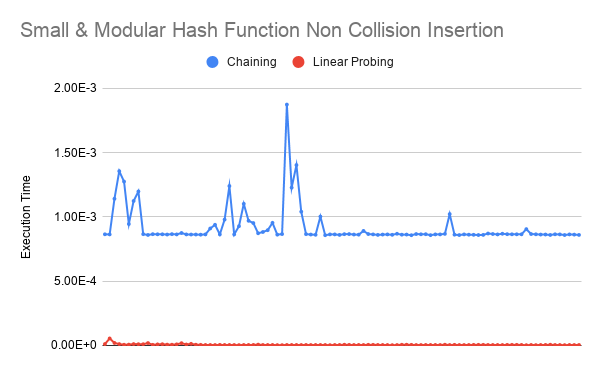


Figure 4.2.1 Figure 4.2.2

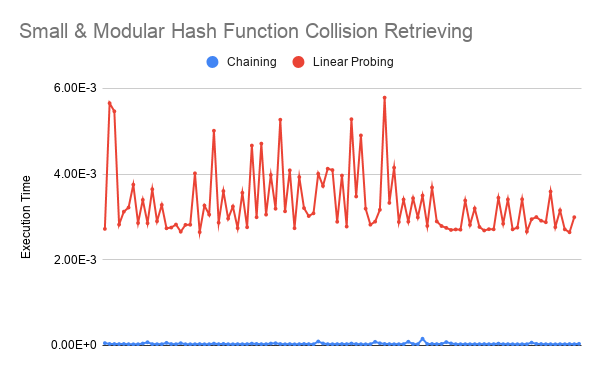
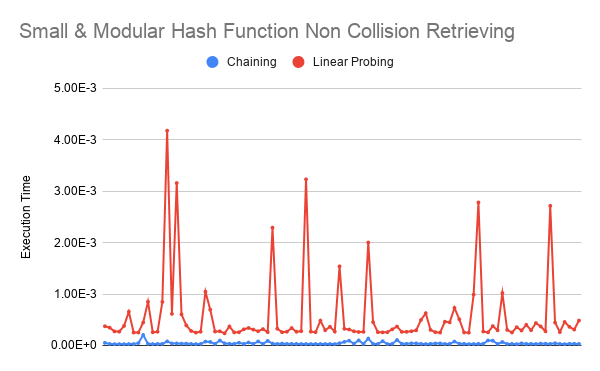


Figure 4.2.3 Figure 4.2.4

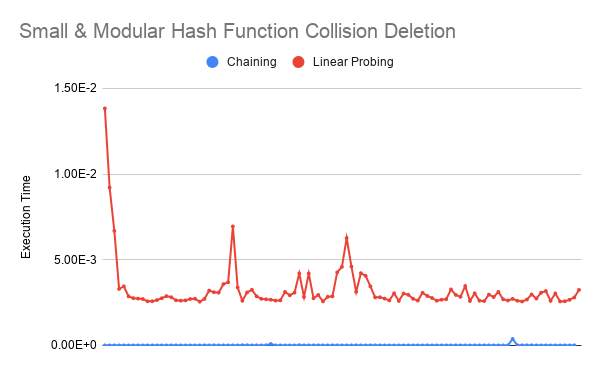
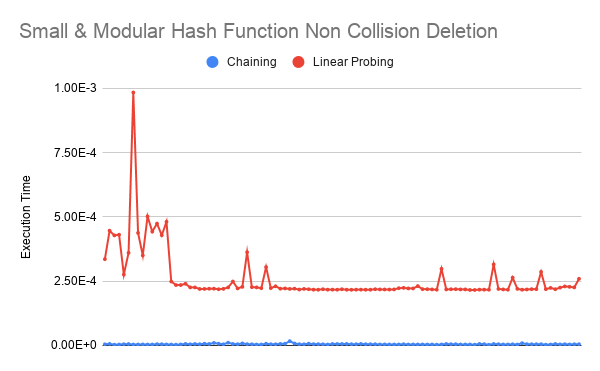


Figure 4.2.5 Figure 4.2.6

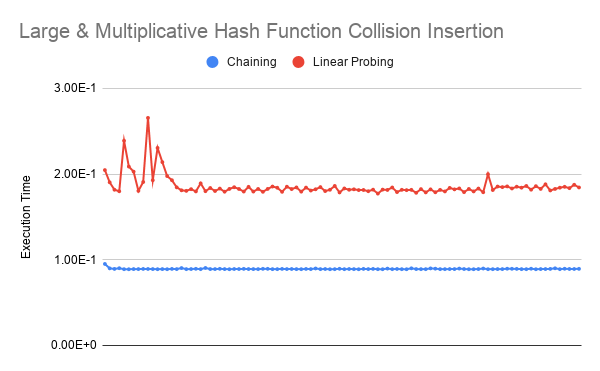
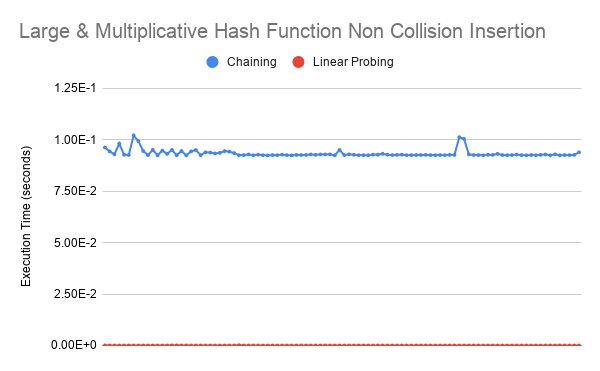


Figure 4.2.7 Figure 4.2.8

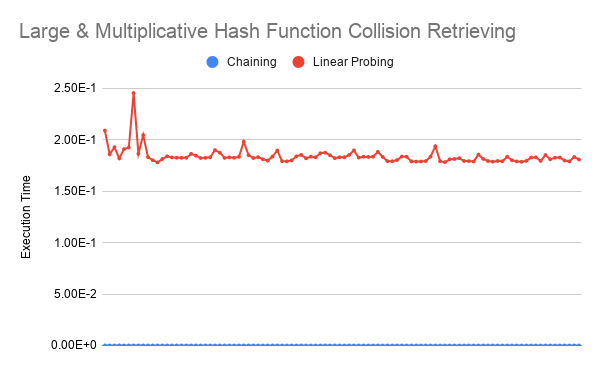
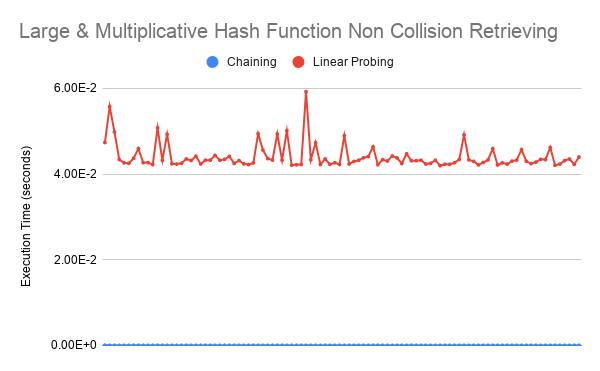


Figure 4.2.9 Figure 4.2.10

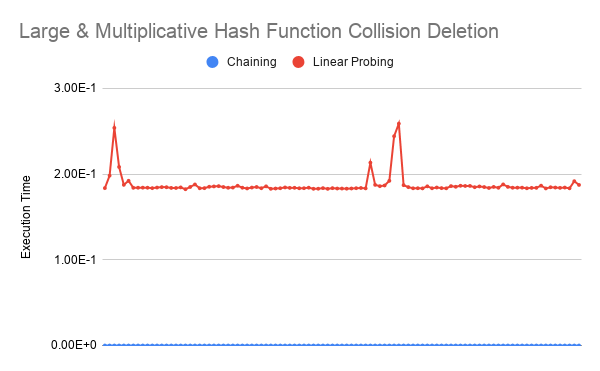
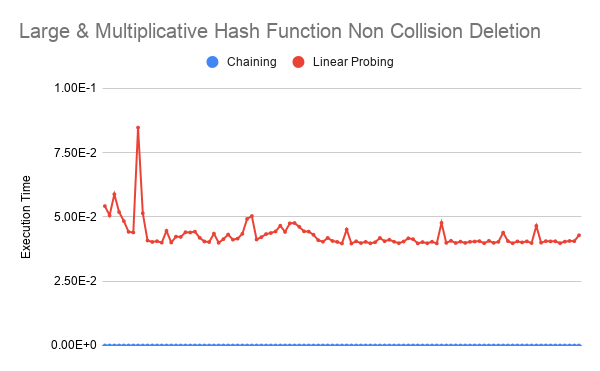


Figure 4.2.11 Figure 4.2.12

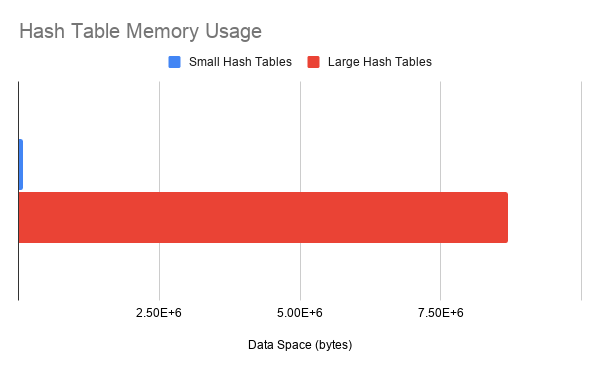
****

Figure 4.2.13 - **Hash Table Memory Usage**

**4.3 Data Analysis**

Indicatively, in terms of linear probing, the execution time for non collision operations are shown to have always been executed at a faster rate under all variability. This result is of logical reasoning as collision related operations inevitably required additional proceedings to execute; linear probing required O(*n*) iteration time compared to its non collision equivalent of O(1) constant operation time as reflected in Figure 4.2.1, Figure 4.2.2, Figure 4.2.7 and Figure 4.2.8. In contrast, the chaining collision handling method demonstrated consistent execution time for both non collision and collision for all operations - this result is explained further in the following paragraphs.

Given the experimental data retrieved, chaining is observed to have demonstrated more efficiency in comparison to linear probing for all operations except non collision insertions. The insertion of chaining required appending data to a supplementary data structure implemented at each hashed bucket, whereas linear probing O(1) insertion was applied directly to the hashed bucket, theoretically and experimentally yielding a justifiable result, as direct insertion should be of faster execution in comparison to indirect insertion of data.

Nonetheless, the results in which display chaining as a more efficient collision handling method may have been the outcome of a fundamentally flawed experiment which posed biased towards chaining. All linear probing experimentations were done in a simulation of O(*n*) as explained, however, this was not done for chaining. In all scenarios of varying hash table sizes, chaining possessed only 30 key value pairs at each collided bucket. Therefore, the iteration of linear probing at O(*n*) would have always been significantly greater; and . It would have been impossible to have simulated O(*n*) for chaining operations as each bucket could theoretically contain infinite appenditures of key value pairs, resulting in O(∞). Ergo non collision and collision operations for chaining showed insignificant difference relative to non collision and collision operations for linear probing, as iteration of a constant 30 elements within a bucket’s array is considerably smaller than 10 000 and 1 000 000 iterations. Consequently, all operations for chaining hence showed similar, if not equal execution times. However, this unfair bias for chaining may be reasoned by the chi-squared distribution **FIND NUMBER TO PROVE HERE** thus fundamentally proving that it would have been unlikely for 30 collisions to have incipiently occurred at a single bucket. Additionally, it can be noted that following the O(*n*) insertion of linear probing, the hash table would have a load factor value of 1; subsequent insertions would always guarantee collision overfilling the hash table buckets. The chaining collision handling method does not have this limitation present.

Finally, in terms of memory usage, 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.

**4.4 Errors**

|  |  |
| --- | --- |
| **Random Errors** | **Systematic Errors** |
| Varying laptop runtimes. | Utilizing arrays in place of linked lists for chaining hash tables. |
| Writing in csv file execution time. | Fundamental advantage for chaining, see section 4.3 Data Analysis. |
| Random computer executions in background. | / |

Figure 4.4.1 - **Error Chart**

**Further Research**

**5.1 Examination in Alternative Programming Languages**

In addition to the variability present by differing hash functions and bucket sizes 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 Adjusting Collision Handling Equity**

Although mentioned in section 4.3, the natural yet improbable advantage for the chaining collision handling method may have been a point for further research. It could have been a possibility to have made each subarray within a bucket’s array the same length as the array linear probing operated with, thus producing a more comparable difference in data variability.

**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; open addressing specifically notwithstandingly contains two other methods to have been explored. Further research into other forms of open addressing 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.

**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 tables, evaluated based on time complexity and memory usage.

Summatively, data obtained following the engineering of an experimental methodology displays results in which the chaining collision handling method generally showed greater performance in comparison to linear probing, contradicting that of hypothesized, which predicted chaining to be exclusively more efficient of larger bucket sizes. In contrast, the execution time of chaining generally showed faster results through all programmed variations of each hash table. Nonetheless, these results may have been a result of an unfair experiment in which nullified major differences in data variations, despite this scenario being improbable proven by the chi-squared distribution.

Therefore, data indicates that linear probing serves as a more efficient method of collision handling in hash tables only during non collision insertion operations, and furthermore is not necessarily a more memory efficient alternative, concurrently shown to be consistent through hash tables with varying hash functions and bucket sizes.

The guaranteed issue of collisions occurring as a byproduct of hash functions in hash tables subsequently forces a solution for this issue. This paper seeks to serve as a guide for software engineers, developers, programmers, and computer researchers by presenting maximized efficient collision handling solutions in the implementation of hash tables.

**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.

**Appendix**

Code and raw data utilized and obtained can be digitally found at: <https://drive.google.com/drive/folders/1PnrCWHsEtI2hMCd2Wxt0nQ4b2IlI8e2b?usp=sharing>

**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  class HashTablesXXXX: #Hash Table Specification  def \_\_init\_\_(self):  self.size = XXXX #10000 or 1000000 = Small / Large  self.bucket = [XXXX for generation in range(self.size)] #[] for chaining, False for linear probing collisions, None for linear probing non collisions  self.bucket[655] = None #Only For Linear Probing Collisions! This simulates O(n) collisions as "lpvsch" with hash code 656  def print\_bucket(self):  print(self.bucket)  def memory\_usage(self):  print(getsizeof(self.bucket)) |

Figure 8.1.1 - **Initialization**

|  |
| --- |
| def modular\_hash(self, key):  hash\_code = 0  for character in key:  hash\_code += ord(character)  return hash\_code % self.size  def \_\_setitem\_\_(self, key, value):  hash\_code = self.modular\_hash(key)  found = False  for index, element in enumerate(self.bucket): #Iterate through array at hashed bucket  if len(element) == 2 and element[0] == key: #If it already exists, replace key value pair  self.bucket[hash\_code][index] = (key, value)  break  if not found:  self.bucket[hash\_code].append((key, value))    def \_\_getitem\_\_(self, key):  hash\_code = self.modular\_hash(key)  for elements in self.bucket[hash\_code]: #Iterate through array at hashed bucket  if elements[0] == key: #elements[0] because each elements are tuples of (key,value)  return elements[1]  raise Exception("Key does not exist in Hash Table")  def \_\_delitem\_\_(self, key):  hash\_code = self.modular\_hash(key)  for index, element in enumerate(self.bucket[hash\_code]):  if element[0] == key: #Iterate through array at hashed bucket until key is found  del self.bucket[hash\_code][index]  return  raise Exception("Key does not exist in Hash Table") |

Figure 8.1.2 - **Modular Hash Function & Chaining**

|  |
| --- |
| def multiplicative\_hash(self, key):  constant = (3^40)/(2^64) #Real number for c = s/2^64, 0<s<2^w  hash\_code = 0  for character in key:  hash\_code += ord(character)  return floor(self.size \* (modf(hash\_code \* constant)[0]))  def \_\_setitem\_\_(self, key, value):  hash\_code = self.multiplicative\_hash(key)  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))    def \_\_getitem\_\_(self, key):  hash\_code = self.multiplicative\_hash(key)  for elements in self.bucket[hash\_code]:  if elements[0] == key:  return elements[1]  raise Exception("Key does not exist in Hash Table")  def \_\_delitem\_\_(self, key):  hash\_code = self.multiplicative\_hash(key)  for index, element in enumerate(self.bucket[hash\_code]):  if element[0] == key:  del self.bucket[hash\_code][index]  return  raise Exception("Key does not exist in Hash Table") |

Figure 8.1.3 - **Multiplicative Hash Function & Chaining**

|  |
| --- |
| def probe\_range(self, index):  return [\*range(index, len(self.bucket))] + [\*range(0, index)] #Returns an array of bucket indexes in order of linear probe cycle  def linear\_probe(self, key, index): #Goal is to find an index which is None (empty)  for index in self.probe\_range(index):  if self.bucket[index] is None:  return index  if type(self.bucket[index]) != bool: #This needs to be checked in case of collision simulation of False filled self.bucket  if self.bucket[index][0] == key:  return index  raise Exception("Hash Table is full")  def modular\_hash(self, key):  hash\_code = 0  for character in key:  hash\_code += ord(character)  return hash\_code % self.size  def \_\_setitem\_\_(self, key, value):  hash\_code = self.modular\_hash(key) #Simply use the linear probe function to get an empty index  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.modular\_hash(key)  if self.bucket[hash\_code] is None: #If hashed bucket 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: #Also a tuple hence double indexing  return self.bucket[index][1]  def \_\_delitem\_\_(self, key):  hash\_code = self.modular\_hash(key)  if self.bucket[hash\_code] is None:  return None  for index in self.probe\_range(hash\_code):  if self.bucket[index] is None:  raise Exception("Key does not exist in Hash Table")  if self.bucket[index][0] == key: #If key is found...  self.bucket[index] = None #The index is simply replaced by None (tombstoning)  break |

Figure 8.1.4 - **Modular Hash Function & Linear Probing**

|  |
| --- |
| 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 multiplicative\_hash(self, key):  constant = (3^40)/(2^64)  hash\_code = 0  for character in key:  hash\_code += ord(character)  return floor(self.size \* (modf(hash\_code \* constant)[0]))  def \_\_setitem\_\_(self, key, value):  hash\_code = self.multiplicative\_hash(key)  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.multiplicative\_hash(key)  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]  def \_\_delitem\_\_(self, key):  hash\_code = self.multiplicative\_hash(key)  if self.bucket[hash\_code] is None:  return None  for index in self.probe\_range(hash\_code):  if self.bucket[index] is None:  raise Exception("Key does not exist in Hash Table")  if type(self.bucket[index]) != bool:  if self.bucket[index][0] == key:  self.bucket[index] = None  break |

Figure 8.1.5 - **Multiplicative Hash Function & Linear Probing**

|  |
| --- |
| hash\_table["lpvshc"] = False  hash\_table["lpvcsh"] = False  hash\_table["lpvchs"] = False  hash\_table["lpvhsc"] = False  hash\_table["lpvhcs"] = False  hash\_table["lpsvch"] = False  hash\_table["lpsvhc"] = False  hash\_table["lpscvh"] = False  hash\_table["lpschv"] = False  hash\_table["lpshvc"] = False  hash\_table["lpshcv"] = False  hash\_table["lpcvsh"] = False  hash\_table["lpcvhs"] = False  hash\_table["lpcsvh"] = False  hash\_table["lpcshv"] = False  hash\_table["lpchvs"] = False  hash\_table["lpchsv"] = False  hash\_table["lphvsc"] = False  hash\_table["lphvcs"] = False  hash\_table["lphsvc"] = False  hash\_table["lphscv"] = False  hash\_table["lphcvs"] = False  hash\_table["lphcsv"] = False  hash\_table["lvpsch"] = False  hash\_table["lvpshc"] = False  hash\_table["lvpcsh"] = False  hash\_table["lvpchs"] = False  hash\_table["lvphsc"] = False  hash\_table["lvphcs"] = False  hash\_table["lpvsch"] = True |

Figure 8.1.6 - **String Permutations of** "lvphcs" **for Collisions**

|  |
| --- |
| hash\_table["%$gD"] = True  hash\_table["]Sz@#"] = True  hash\_table["!"] = True  hash\_table["~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"] = True  hash\_table["uvjolx"] = True  hash\_table["hajtnulkajsx"] = True  hash\_table["djbaffcgtipyyfswqnu"] = True  hash\_table["aph"] = True  hash\_table["wszajlikf"] = True  hash\_table["cnnvsvltndqddogr"] = True  hash\_table["lrihcznaxwxshs"] = True  hash\_table["kqnzfswojjjhmveoyfqrngitubppwtertuuvxprdywpsha"] = True  hash\_table["wngnadicsxqfblkrqfygtrfrix"] = True  hash\_table["valnrmpew"] = True  hash\_table["fqvsojexjkqlqjtlmzweq"] = True  hash\_table["gfkapbqgza"] = True  hash\_table["facab"] = True  hash\_table["ilffibzd"] = True  hash\_table["pdpnykngegzqfz"] = True  hash\_table["ltcsbmlgi"] = True  hash\_table["zfgcdviookjdefi"] = True  hash\_table["prfgquaazjliwsobpdpogbhmhhnhaioivwhzrrfydrngwvlglgjrhlbhrt"] = True  hash\_table["gdvub"] = True  hash\_table["aqsuwiw n g v a"] = True  hash\_table["cwafkhh"] = True  hash\_table["fwwmizcazhaapq"] = True  hash\_table["zobrutuanrttiwlwlgpdwtjshvnp"] = True  hash\_table["taezifxqrmwmun"] = True  hash\_table["lmsjwtnqsqwchdushxzauqwkbybpnawfsrvurowzptvhvdwzusizpcbbuzfonqgqhhyfnovtsnaogitluacvx"] = True  hash\_table["mxibee"] = True |

Figure 8.1.7 - **Random Strings for Non Collisions**

|  |
| --- |
| hash\_table = HashTableXXXX()  #Prior key insertions here...  initial\_time = time.time()  #Operation is set here...  For insertions:  hash\_table["XXXX"] = True  For retrieving:  print(hash\_table[XXXX])  For deletion:  del hash\_table[XXXX]  execution\_time = time.time() - initial\_time  directory = r"XXXX" #XXXX = File Directory  with open(directory,"a") as dataset:  dataset.write(f"\n{execution\_time:.20f}")  hash\_table.memory\_usage() |

Figure 8.1.8 - **Dependent Variable Measurement**

|  |
| --- |
| for execution in range(100):  exec(open(f"XXXX").read()) #XXXX = File Directory |

Figure 8.1.9 - **File Executor**

**8.2 Unused Graphical Data Visualization**

The following figures show the remaining unused graphical data not presented in section 4.2 Graphical Data Visualization however still display the same consistencies.

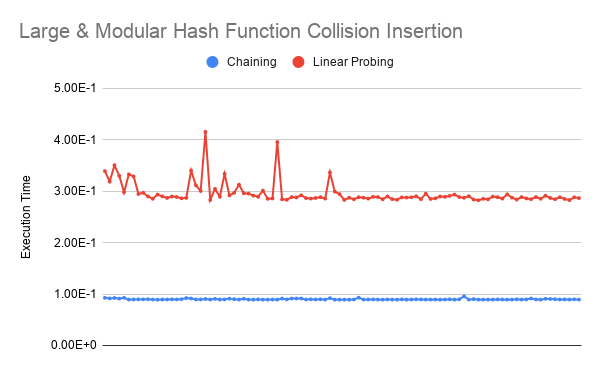
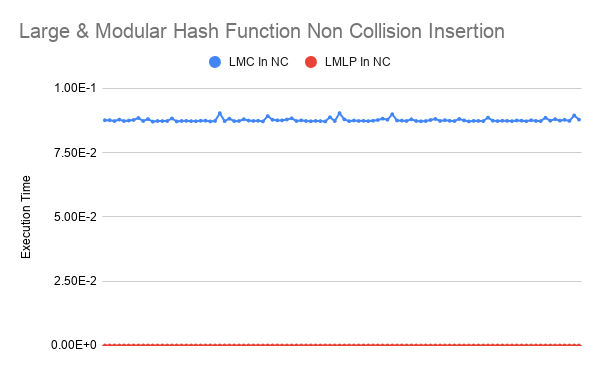


Figure 8.2.1 Figure 8.2.2

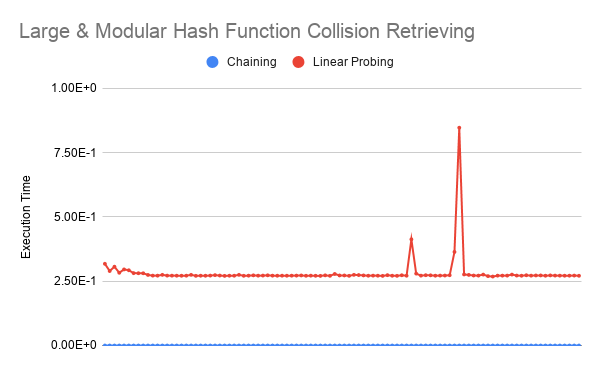
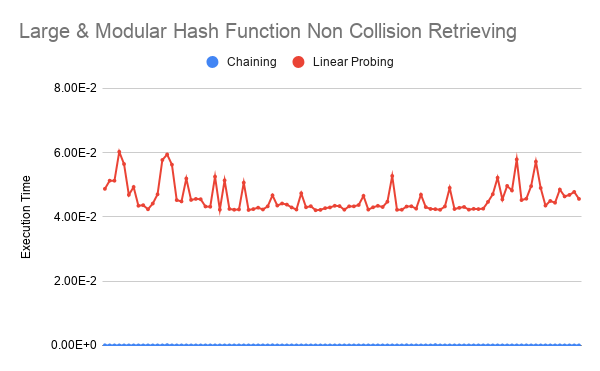


Figure 8.2.3 Figure 8.2.4

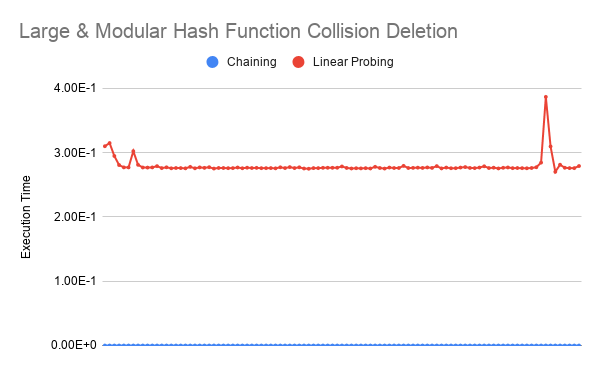
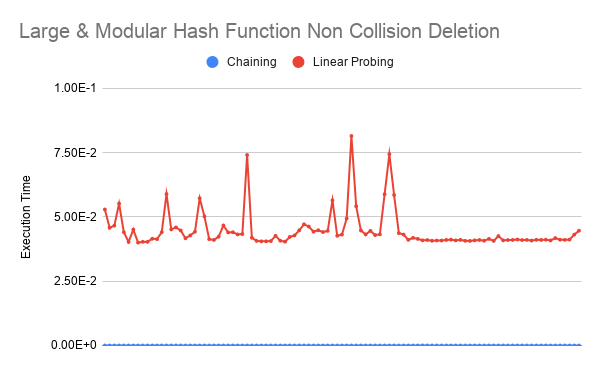


Figure 8.2.5 Figure 8.2.6

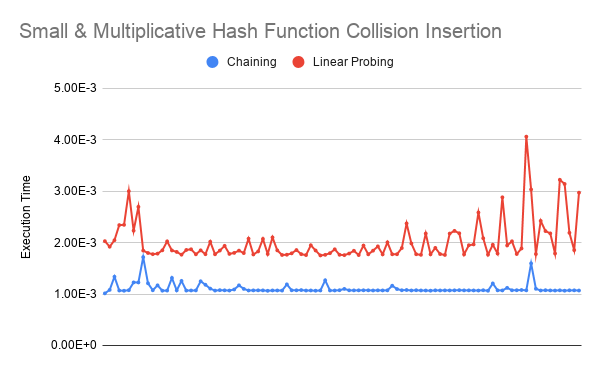
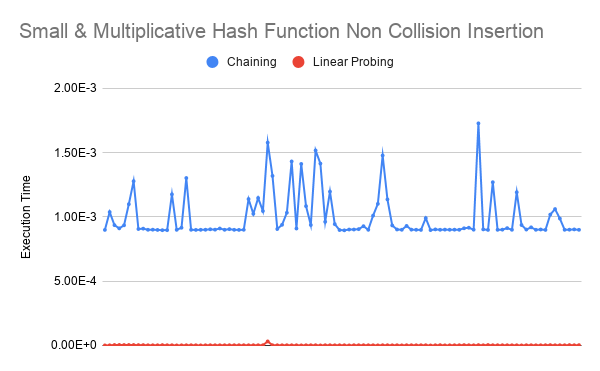


Figure 8.2.7 Figure 8.2.8

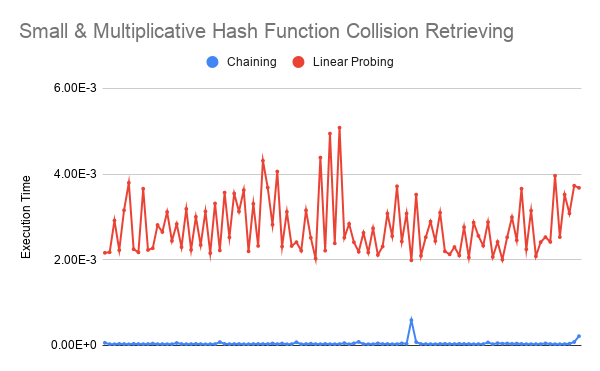
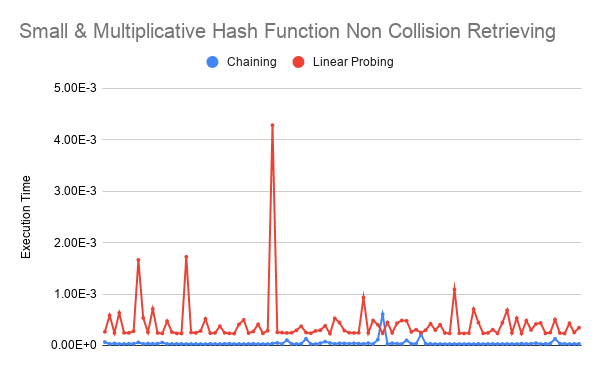


Figure 8.2.9 Figure 8.2.10

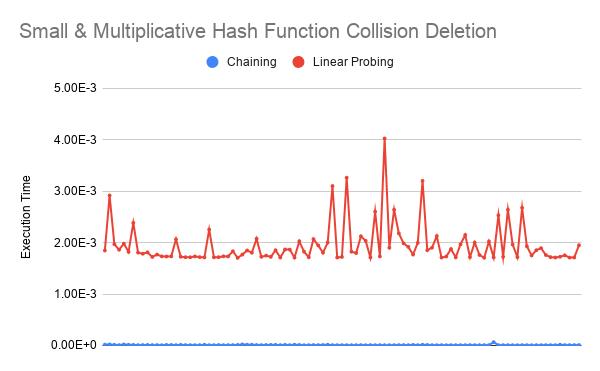
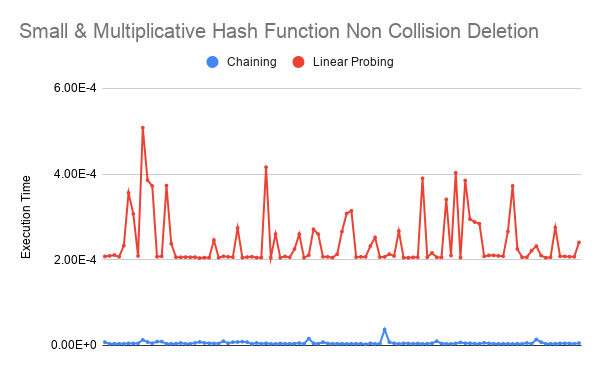


Figure 8.2.11 Figure 8.2.12

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_