Computer Science Extended Essay

The Efficiency of Non-Cryptographic Hash Functions in Hash Tables

Research Question: How do the ***efficiency*** of non-cryptographic hash functions FNV-1a and MurmurHash3 compare in terms of ***collision resistance*** and ***time complexity*** for optimizing hash table implementations?

Word Count: 3993

Table of Contents

1. Introduction

One of the key concepts in computer science is *efficiency,* which refers to the effectiveness of a system in utilising time and space to perform its tasks. Especially in the modern era of technology, where data is being generated at an unprecedented rate and volume, there is an increasing demand for programs to *efficiently* manage large volumes of data (Hashem et al). One method that programmers commonly use to increase the efficiency of their programs is a hash table, a type of abstract data structure (ADT) responsible for fundamental operations such as searching, inserting, and deleting data.

According to Professor Myers in a Cornell lecture, hash tables are "one of the best and most useful data structures there is" (Myers). They use non-cryptographic hash functions (NCHFs) to produce a *deterministic* and *random* integer that maps to each value stored in the data structure (Pfenning and Simmons). Consequently, the efficiency of hash tables depends heavily on the NCHF used to produce the keys.

No NCHF is universally agreed upon to be the most efficient. In fact, each programming language uses a different hashing algorithm for its built-in hash table classes (Tapia-Fernández et al. 100). As such, this paper sets out to compare the performances of Fowler–Noll–Vo-1a (FNV-1a) and MurmurHash3, two of the most commonly used NCHFs, with the research question: *How do the* ***efficiency*** *of non-cryptographic hash functions FNV-1a and MurmurHash3 compare in terms of* ***collision resistance*** *and* ***time complexity*** *for optimizing hash table implementations?* Specifically, *collision resistance* refers to the hash table's ability to avoid collisions (Estébanez et al. 685), and *time complexity* refers to the relationship between the size of data and the NCHF's execution time ("Algorithms: Analysis"). This paper will quantify time complexity using Big-O notation, a mathematical concept that characterises an algorithm's time complexity in relation to the size of the input data based on its worst-case performance. Additionally, it will define execution time (or runtime) as the total time required for a computer program to complete a task.

A common misconception of hash tables is that they allow "all operations [to] be […] $O(1)$" (Myers). This belief assumes that the employed NCHF runs at $O(1)$, which is currently not accurate, as no NCHF

achieves that level of efficiency. However, by using a more efficient NCHF, a hash table can be made more efficient and approach $O(1)$.

This paper explores the hash table structure and the algorithms behind MurmurHash3 and FNV-1a, then details the hypothesis and methodology of the experiment. Through a thorough analysis of the experimental data, this paper concludes that both NCHFs have similar collision resistances, but MurmurHash3 is slightly more efficient due to its faster time complexity.

2. Theory

2.1 Hash Table

A hash table is an ADT that can lookup, insert, and delete key-value pairs. Although hash tables are constructed using an array, they don't store elements in the next available index (Black). Instead, they use a hash key to represent the *bucket*, the array index, that the value will be stored in (Sullivan). The hash keys are produced by NCHFs, mathematical functions that receive an object as input and return a fixed-sized integer that identifies the original object (Akoto-Adjepong et al.). Since there are 2^n possible hash values and limited buckets in a hash table, where *n* represents the number of bits in the hashes, the bucket that an element is stored in is the hash key modulus the number of buckets.

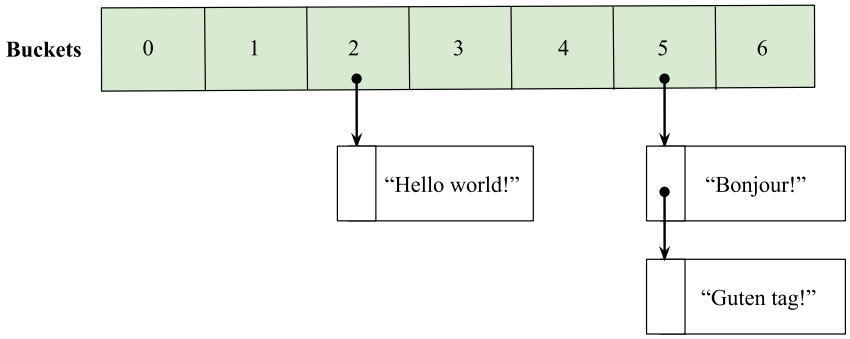

Figure 1. A visual representation of a hash table.

In Figure 1, the string “Hello world!” produces a hash key of 2, and is hence stored in the bucket 2. Similarly, both “Bonjour!” and “Guten tag!” lead to a hash of 5, and are hence stored in the bucket 5. There are multiple solutions to *collisions*, which occur when two or more unique elements generate the same hash key and are attempted to be stored in the same bucket. Common solutions include open addressing and chaining (Sanders et al. 92). For the sake of simplicity, this paper will implement the chaining method, which is when each bucket is a singly linked list that can store multiple variables.

A perfect hash table is described as when each of the elements stored in the hash table has a unique bucket and each of the buckets corresponds to a unique element (Sanders et al. 92-95). This one-to-one relationship ensures that the hash table does not waste any memory storage and can search/insert/remove each element directly from the bucket, resulting in those operations being $O(1)$, hence maximizing the efficiency of the data structure. In practice, it is impossible to maintain this one-to-one relationship since “the number of potential inputs is infinite and the number of potential outputs is finite” (Estébanez et al. 682). NCHFs are also more complex than $O(1)$, further increasing the complexity.

To achieve a close-to-perfect hash table by minimizing memory storage and collisions, the average number of elements in each bucket, also known as the *load factor* (Black), should be $\frac{n}{m} < 1$, where n represents the total number of elements, and m represents the total number of buckets (Myers).

Data Structure	**Operations**		
	contains()	**add()**	**remove()**
Singly-Linked List	O(n)	O(1)	O(n)
Hash Table	O(h + n)	O(h + 1)	O(h + n)

Table 1. The worst-case time complexity of a singly-linked list and a hash table.

Here, *n* represents the number of elements in the bucket, and *h* represents the time complexity of the NCHF.

As seen in Table 1, the time complexity of hash tables is determined by the time complexity of the NCHF and the singly linked list. As such, the efficiency of a hash table can be maximized if the hash function meets the following two conditions:

1. **Collision resistant:** The hash function can convert the time complexity of the linked list operations into $O(1)$ if there is only one element in the linked list. Using Figure 1 as an example, having only one element in bucket 2 allows accessing the element "Hello World!" to be $O(h + 1)$. As such, it is important to avoid collisions to maximize efficiency. Increasing a hash table's collision resistance, according to H. P. Luhn, the first known computer science researcher to write about hashing, is through disrupting the original data's patterns (Knuth). The hash function should produce *random* hashes for each unique element so that the elements can be uniformly distributed.

2. **Fast:** Since a hash table's main functions, *contains(), add(),* and *remove(),* require $O(h + k)$ in the worst-case scenario to execute, where k is the time complexity for the operation in a linked list, the time complexity of the NCHF is extremely important because it directly affects the efficiency of the hash table.

2.2 FNV-1a

The FNV hash algorithm was originally an idea between Glenn Fowler and Phong Vo in 1991 and was later improved by Landon Curt Noll (Noll). The function is commonly used in Domain Name Servers, web search engines, databases, and hash maps to this day due to its simplicity and effectiveness (Noll). FNV-1a is a slight alternative to the original FNV-1 hash algorithm and is preferred by Noll himself (Noll).

```
hash = offset_basis
for each octet_of_data to be hashed
      hash = hash * FNV_prime
      hash = hash xor octet_of_data
return hash
```

Figure 2. Official pseudocode of FNV-1a (Noll).

To start, the hash is initially set as a constant named *offset_basis*. Most people use the magic number $0x811C9DC5$ because it is a prime number that can produce 32-bit hashes (Karasek et al.). Next, for each of the bytes of the data to be hashed, the hash will go through two operations. One, the hash will equal to the hash XOR the byte of data. Two, the hash will equal itself multiplied by another constant named *FNV_prime*, most commonly as $0x01000193$. Note that the two constant integers were carefully computed and tested to optimize the distribution and collision resistance of hashes, but their derivations are out of this paper's scope (Karasek et al.).

2.3 MurmurHash3

MurmurHash is another commonly used NCHF. It was first created by Austin Appleby in 2008 and is currently implemented in many open-sourced projects ranging from common libraries, search engines, databases, and web servers (Appleby). This paper will discuss MurmurHash3, the most recent version of MurmurHash.

The process of MurmurHash3 can be separated into three sections.

2.3.1 Blocks

1. The hash is first set to be the seed, which can be randomized by the user.
2. For each 32-bit block (4 bytes) in the data,

a. Multiply the block by $0xCC9E2D51$, bitwise rotate the block to the left by 15 positions, multiply the block by $0x1B873593$.

b. Set the hash to be itself XOR the block.

c. Bitwise rotate the hash to the left by 13 positions, multiply itself by 5, and add $0xE6546B64$.

2.3.2 Remaining Bytes

1. Combine the remaining bytes into one block.
2. Multiply the block by $0xCC9E2D51$, bitwise rotate the block to the left by 15 positions, multiply the block by $0x1B873593$.
3. The hash is then set to be itself XOR the block.

2.3.3 Finalizing Hash

1. Set the hash to be itself XOR the number of bytes in the data.
2. Set the hash to be itself XOR the hash shifted to the right by 16.
3. Multiply the hash by $0x85EBCA6B$.
4. Set the hash to be itself XOR the hash shifted to the right by 13.
5. Multiply the hash by $0xC2B2AE35$.
6. Repeat step 2.

Similar to FNV-1a, all of the constants here were determined via mathematical derivations and testing outside of this paper's scope (Appleby).

3. Hypothesis and Applied Theory

3.1 Hypothesis One

This paper hypothesizes that MurmurHash3 will be more collision-resistant than FNV-1a. MurmurHash3 includes multiple combinations of multiplications, rotations, and shiftings for each block to randomize the hashes, whereas FNV-1a only conducts one XOR and one multiplication for each byte. MurmurHash3 also randomizes its hashes through a finalization stage, which leads to a much more complex hashing process.

3.2 Hypothesis Two

This paper hypothesizes that both MurmurHash3 and FNV-1a will have a linear relationship between their execution time and their size of data, as represented by $O(n)$, but MurmurHash3 will be faster to compute than FNV-1a.

Both algorithms utilize operations such as shifts, XORs, and multiplications, all of which require $O(1)$ time to run. MurmurHash3 also utilizes left rotation in their algorithm, but since rotations are achieved via shifts, the rotation operations can be inferred to cost $O(1)$ time as well.

FNV-1a leverages a for-each loop that conducts the $O(1)$ operations with each byte in the data, which results in an $O(n)$ runtime in Big-O notation, where n represents the length of the data.

Although MurmurHash3 has more $O(1)$ operations within its for-loop, the algorithm groups every three bytes into one block, resulting in only $\frac{n}{4}$ iterations. This suggests that MurmurHash3 can run approximately four times faster than FNV-1a when the data contains more than three bytes.

4. Methodology

4.1 Independent Variable

The independent variable in this investigation is the type of NCHF. The two that will be investigated are FNV-1a and MurmurHash3. For testing, I programmed the 32-bit version NCHFs and their hash table implementations in Java and included them in ***Appendix Three and Four***.

4.2 Dependent Variables

4.2.1 Collision Resistance

To quantify the collision resistance, a *collision rate* will be computed for each data set inserted into the NCHFs' hash tables. The collision rate will be calculated as the ratio of the total number of collisions to the total number of buckets. The total number of collisions is measured by counting the number of elements added to filled buckets. The total number of buckets will be the number of elements hashed.

4.2.2 Time Complexity

This paper will measure the time required to compute each NCHF and the hash tables' *contains()*, *add()*, and *remove()* functions for different-sized data, using the difference between the start and end timestamps of the hashing operations obtained via *System.nanoTime()*. The time complexity for each NCHF is the relationship between the size of data in bytes and the execution time in nanoseconds.

4.3 Controlled Variables

4.3.1 Data and Data Sets

The data and data sets inputted to the two hash tables must be the same because factors such as the length of data and the size of the dataset will influence the collision rate and speed. The datasets for collision resistance tests are pre-existing to test the hash tables' performance in real-world applications.

- **words.txt** - a TXT file containing 466,550 English words, numbers, and symbols. It is useful for testing the hash tables' capability in randomly distributing shorter strings (Dwyl).
- **passwords.txt** - a TXT file containing the 100,000 most commonly-used passwords. It is useful for testing the NCHF's ability to hash short combinations of characters (Miessler).
- **bbc_news.csv** - CSV file containing 35,183 BBC news articles, including the titles, publishing dates, globally unique identifier, link, and description (Preda). All of the information will be combined into a string to test the hash tables' abilities to randomly distribute longer strings.

To evaluate the time complexity of the NCHFs, 10 random strings of varying lengths will be used as input. The lengths of the strings are 100000000, 130000000, 160000000, 190000000, 220000000, 250000000, 280000000, 310000000, 340000000, 370000000 bytes.

4.3.2 Computer and Operating System

The collision rate and speed of the hashing functions depends on the computer and operating system. To ensure the computer and operating system remain the same, all data will be collected on an Apple MacBook Air (M1, 2020) with 8 GB of memory. The computer runs on Sonoma version 14.2.1 macOS.

4.3.3 Programming Language and Integrated Development Environment (IDE)

The programming language and IDE must remain the same because the speed of the NCHFs depends on them. All data collection will be conducted using Java in IntelliJ IDEA 2023.1.2 (Community Edition), build #IC-231.9011.34, with a runtime version of 17.0.6+10-b829.9 aarch64 and an OpenJDK 64-Bit Server Virtual Machine.

4.3.4 Hash Table Implementation

The hash table implementations must remain the same because the algorithms within the implementations can influence the collision rate and speed. Both NCHFs utilize the same pieces of code for their hash table implementations, including the *insert()*, *remove()*, *contains()*, *getDistribution()*, and *getCollision()* functions, as seen in ***Appendix Three and Four***.

4.3.5 Internet Connection

The internet connection must remain the same because a stable connection minimizes variations in data transmission speed and latency, which could affect the execution time of the NCHFs. The computer will connect to the same Wi-Fi and complete the data collection within the same hour to maintain a consistent internet connection.

4.4 Procedure

1. Implement the FNV-1a and MurmurHash3 hash algorithms and their hash table implementations in Java.
2. Initialize an FNV-1a HashTable object and a MurmurHash HashTable object with the size of the first dataset, *words.txt*.
3. For each string in the dataset, insert into both hash tables.

4. Call both hash tables' *getCollision()* function, which will count the number of elements inserted into an occupied bucket. See ***Appendix Three and Four*** for the full code.
5. Call both hash tables' *getDistribution()* function, which will record the number of elements stored in each bucket. See ***Appendix Three and Four*** for the full code.
6. Repeat steps 2 - 5 two more times with *passwords.txt* and *bbc_news.csv*.
7. Generate five random strings with 100,000,000 characters.
8. For each string, record the start time, hash the string using FNV-1a, and record the end time. The execution time is the end time minus the start time.
9. Repeat step 8 with the MurmurHash3 hash function.
10. Repeat steps 7 - 9 nine more times, incrementing 30,000,000 characters to the length of the string for each iteration.
11. Repeat steps 8 to 12 three more times, but instead of measuring the runtime for hashing the string, measure the runtime for searching, inserting, and deleting data.

5. Data Analysis

5.1 Collision Resistance

Dataset	Number of Data/Buckets	FNV-1a		MurmurHash3	
		Number of Collisions	Collision Rate	Number of Collisions	Collision Rate
words.txt	466550	171820	0.3683	171437	0.3675
passwords.txt	100000	36575	0.3658	36792	0.3679
bbc_news.csv	37456	13830	0.3692	13836	0.3694

Table 2. Collision Rate of Each Dataset

In Table 2, the collision rate is calculated to be the number of collisions divided by the number of buckets, as mentioned in Section 4.2.1. The average collision rate for FNV-1a is 0.3678, while the average collision rate for MurmurHash3 is 0.3683.

However, upon close examination of the distribution of the hash tables, an interesting observation was made.

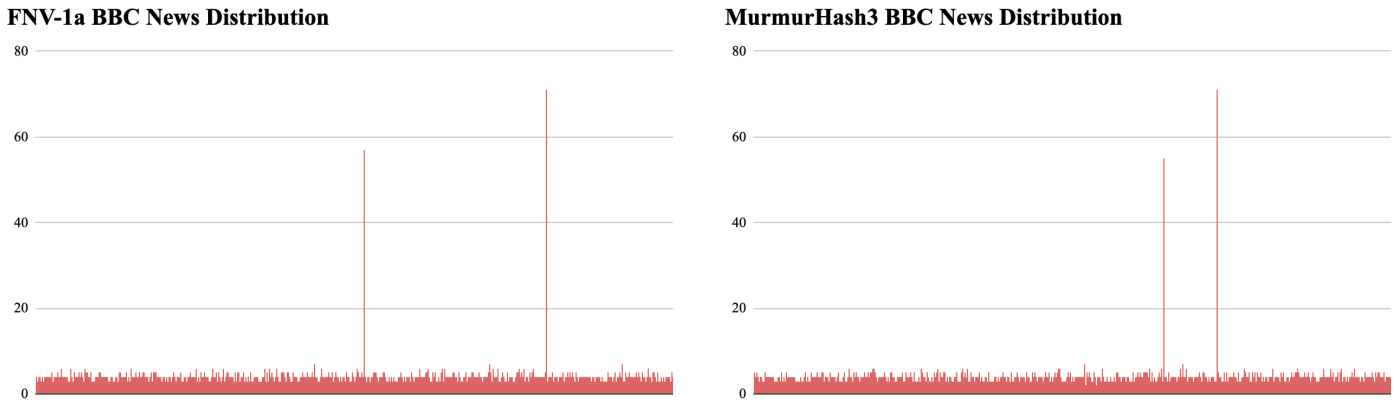

Figure 3 and 4. BBC News Distribution.

In both of the hash tables, two specific buckets respectively stored 57 and 71 strings. Since FNV-1a and MurmurHash3 are fundamentally different hashing algorithms, it was unlikely that both of the algorithms

caused this cluster. Instead, it was more likely that the dataset included the same BBC News articles multiple times.

To prevent the data repetitions from changing the collision rates, the code was slightly altered to only include unique strings in a dataset. This is important because having a large amount of the same data in one bucket would undesirably and significantly maximize the hash table operations' runtimes. The test was conducted once again, leading to the below results.

Dataset	Number of Data/Buckets	FNV-1a		MurmurHash3	
		Number of Collisions	Collision Rate	Number of Collisions	Collision Rate
words.txt	466550	171820	0.3683	171437	0.3675
passwords.txt	99943	37041	0.3706	36878	0.3690
bbc_news.csv	37330	13739	0.3680	13768	0.3688

Table 3. Updated Collision Rate of Each Dataset

The calculations in this table are the same as those in Table 2. The average collision rate for FNV-1a is 0.3690, while the average collision rate for MurmurHash3 is 0.3684.

To verify that there was no more significant clustering, the frequency distribution of the number of data per bucket was recorded.

Number of Data in a Bucket	Frequency					
	Dictionary		Passwords		BBC News	
	FNV-1a	MurmurHash3	FNV-1a	MurmurHash3	FNV-1a	MurmurHash3
0	171769	171397	36995	36838	13739	13768
1	171462	171885	36398	36593	13768	13734
2	85563	85700	18315	18385	6793	6791
3	28753	28679	6154	6080	2317	2306
4	7171	7050	1597	1552	576	577
5	1445	1466	318	330	108	138
6	215	215	53	56	23	15
7	47	34	9	6	5	0
8	6	5	1	0	1	1
9	1	1	0	0	0	0

Table 4. Frequency of Number of Hashes per Bucket

As seen in Table 4, the most significant clustering had only 9 hashes in the same bucket, which is a massive improvement from the previous results.

The below three graphs visually represent the above table.

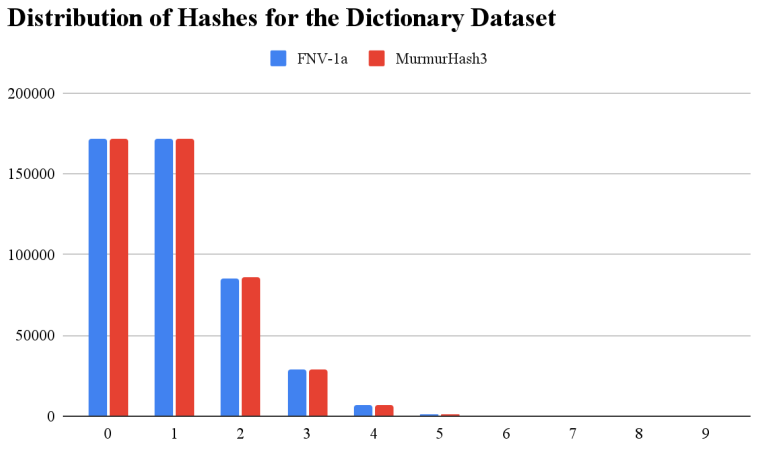

Figure 5. Distribution of Hashes for the Dictionary Dataset.

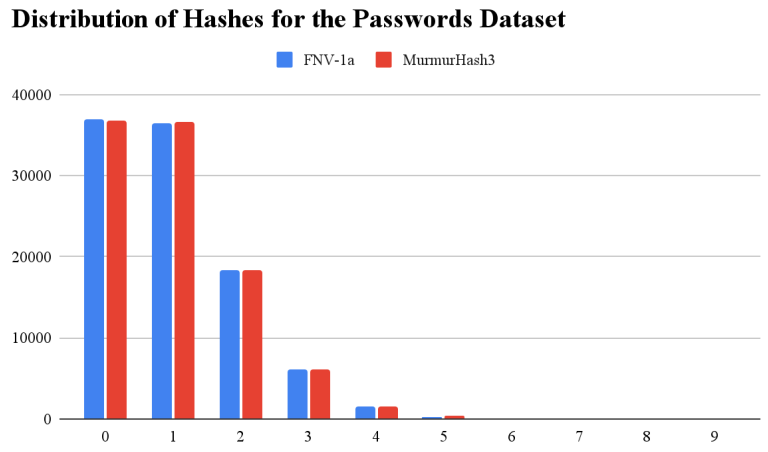

Figure 6. Distribution of Hashes for the Passwords Dataset.

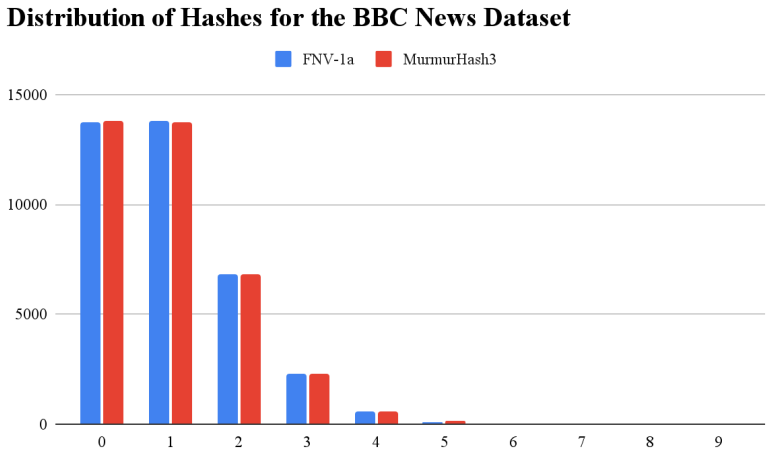

Figure 7. Distribution of Hashes for the BBC News Dataset.

5.2 Time Complexity

Note that all of the below runtimes are the means of five trials. The raw data can be found in ***Appendix One***.

Size of Data in Bytes	Average Time for Hashing /ns		Average Time for Searching /ns		Average Time for Insertion /ns		Average Time for Deletion /ns	
	FNV-1a	**MurmurHash3**	**FNV-1a**	**MurmurHash3**	**FNV-1a**	**MurmurHash3**	**FNV-1a**	**MurmurHash3**
100000000	172083825	107885458	165434092	108178158	166719458	106018250	329207575	213400242
130000000	212367833	136979400	217811216	140959167	214243433	138845958	426675550	277101175
160000000	264234633	169560742	266354033	182319341	263365017	171957242	524232975	340929342
190000000	310547392	201384358	315921100	203114975	312369200	202638484	624629800	405547533
220000000	360583858	233079208	378774600	234861050	368812891	235337725	723155617	469756067
250000000	486857850	266255192	459264242	294399883	441980933	266546108	822672467	533632908
280000000	508445808	320075483	504718667	300005425	491708508	303867550	919669258	597763200
310000000	510737392	329163258	510739758	331645500	508821208	330328933	1017864859	679416408
340000000	553633283	360989675	560422450	364012550	569527392	371163767	1115854642	731028400
370000000	609843200	397889483	616115750	394496583	663264942	394104925	1212311358	794755550

Table 5. Average Execution Time for Different Sizes of Data.

As seen in Table 5, the execution time for hashing, searching, inserting, and deleting strings of certain sizes with FNV-1a and MurmurHash3 was measured. To fully visualize the data, below are four graphs from Desmos representing the relationship between the size of data in bytes and execution time in nanoseconds.

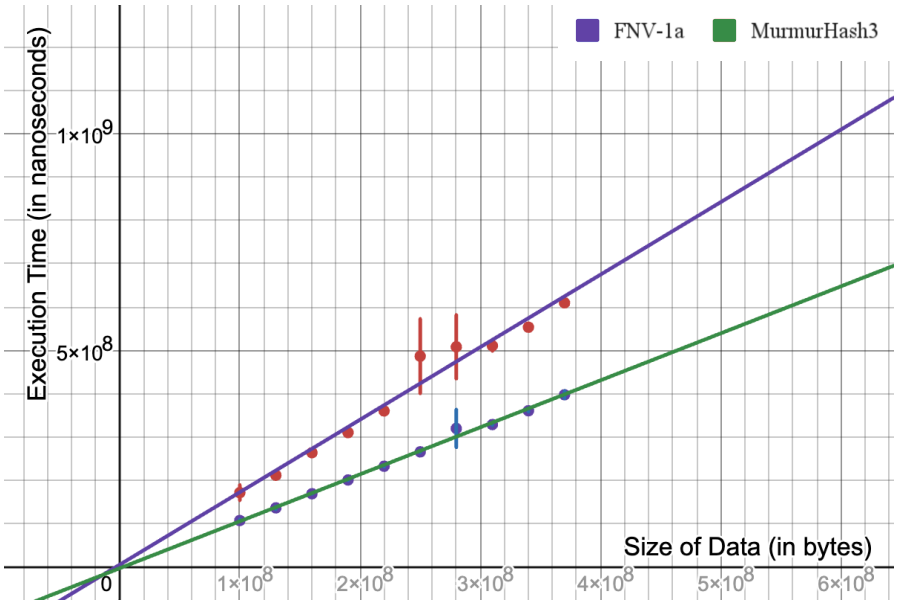

Figure 8. Time Complexity of Hashing

In Figure 8, the error bars represent the standard deviation of the measured execution times throughout the five trials.

According to Desmos, the equation representing FNV-1a's time complexity is

$$t = 1.67296n + 5.7868 \times 10^6 \quad (1)$$

where t represents the runtime in nanoseconds and n represents the size of data in bytes.

The equation representing MurmurHash3's time complexity is

$$t = 1.08391n - 2.3934 \times 10^6 \quad (2)$$

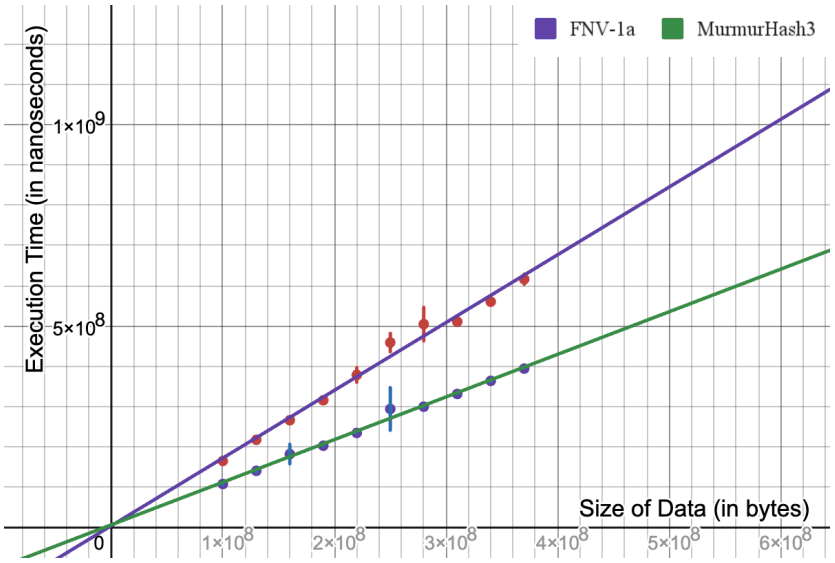

Figure 9. Time Complexity of Searching

In Figure 9, the line representing the estimated time complexity of FNV-1a is once again above the MurmurHash3 line.

According to Desmos, the equation representing FNV-1a's time complexity when searching is

$$t = 1.68146n + 4.4126 \times 10^6 \quad (3)$$

The equation representing MurmurHash3's time complexity for searching is

$$t = 1.05759n + 6.8651 \times 10^6 \quad (4)$$

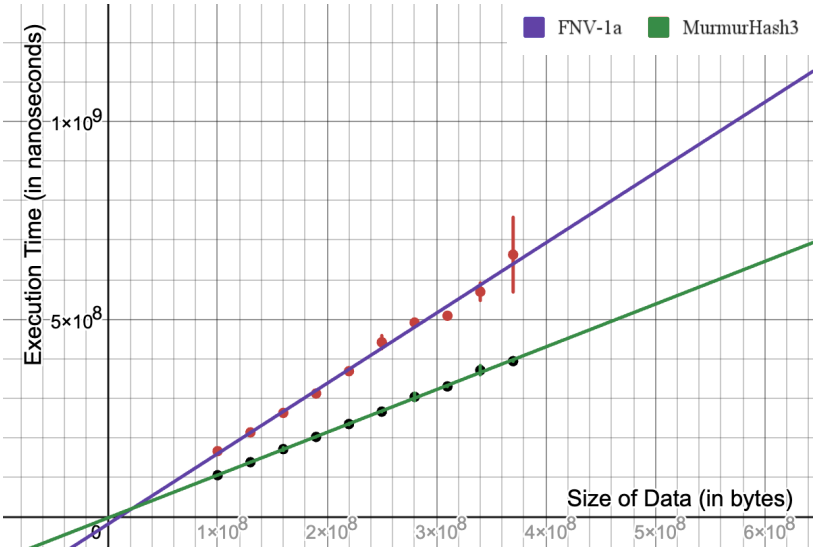

Figure 10. Time Complexity of Inserting

The equation representing the time complexity of inserting data into an FNV-1a hash table implementation is

$$t = 1.77664n - 1.7429 \times 10^{7} \quad (5)$$

The equation representing a MurmurHash3 hash table's time complexity for inserting data is

$$t = 1.07995n - 1.7077 \times 10^{6} \quad (6)$$

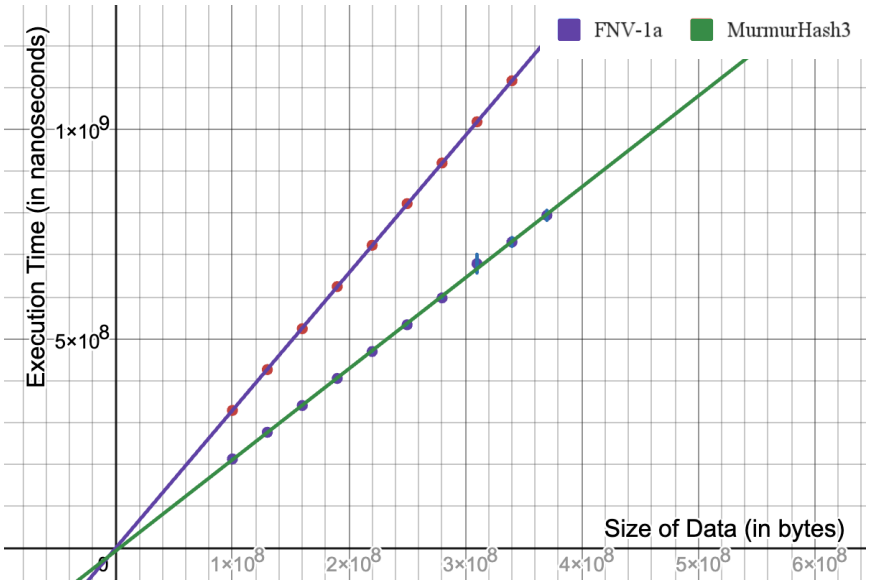

Figure 11. Time Complexity of Removing

The equation representing the time complexity of removing data into an FNV-1a hash table implementation is

$$t = 3.27777n + 1.3505 \times 10^{6} \quad (7)$$

The equation representing a MurmurHash3 hash table's time complexity for removing data is

$$t = 2.17023n - 5.6714 \times 10^{6} \quad (8)$$

6. Conclusion

FNV-1a and MurmurHash3 have very similar collision resistances. As seen in Tables 2 and 3, depending on the specific dataset, sometimes FNV-1a has a lower collision rate, while other times MurmurHash3 would be more collision resistant. This is also reflected in Figures 5 to 7, as the distribution of the hashes is almost identical for all three datasets. In all cases, the collision rates are around 0.37, which suggests that around 37% of the strings inserted into the hash tables will cause a collision, a significant proportion of the strings. This suggests that the time complexity of hash table operations would require $O(n)$ to access the data in the linked list.

Curiously, the collision rate of both FNV-1a and MurmurHash3 hash tables is slightly larger after the repeated strings are removed. This was likely a coincidence caused by a slightly larger decrease in the number of available buckets compared to the number of collisions. The difference is only approximately 0.001, which is insignificant.

Another observation is that the number of empty buckets in both hash tables, as seen in Figures 5 to 7, is very similar to the number of buckets with exactly one data. This suggests that the hash table buckets are not very efficiently used as there is a lot of wasted memory storage.

Hence, this paper's hypothesis regarding MurmurHash3 having a better collision rate than FNV-1a has been disproved. Instead, both of the NCHFs are very similarly collision-resistant, despite MurmurHash3 having a much more complex hashing process.

In regards to the time complexities, this paper's hypothesis was correct in that both FNV-1a and MurmurHash3 have a linear relationship between the execution time and the size of the data being hashed. This is evident in Table 5 and Figures 8 to 11, where all operations could be represented as a positive straight line. This paper prompted equations 1 to 8 to include y-intercepts in case they would reflect the time required to hash null, but some of the equations that Desmos created had negative y-intercepts. This suggests that these

equations cannot accurately represent the runtime of small data. Indeed, a concept in computer science called overhead explains that there will be some unwanted computer processes that slow down execution time, not allowing the relationship between execution time and size of data to be directly proportionate when the size of data is close to zero (Wilper et al).

Another correct hypothesis was that MurmurHash3 would be faster than FNV-1a. This is evident in Figures 8 to 11 and their respective equations because the best-fit line of MurmurHash3 was always below FNV-1a. However, the hypothesis was wrong that the MurmurHash3 hash table would be four times faster than the FNV-1a hash table, as the time complexity equations of FNV-1a have slopes that are approximately **1.6** times larger than that of MurmurHash3, suggesting that MurmurHash3 is not even double the speed of FNV-1a. One possible explanation is that MurmurHash3 incorporates more hashing processes than FNV-1a, including crucial steps such as the finalization stage. Another possible explanation is that the aforementioned overhead processes created a systematic error in the measured times (Lange).

Interestingly, the *remove()* function has the highest time complexity for both FNV-1a and MurmurHash3. This can be explained by the necessity for the function to call *contains()* to verify if the data already exists within the hash table, which would require more time to execute.

Overall, both of the NCHFs create hash tables with very similar collision rates. However, MurmurHash3 has a better time complexity than FNV-1a for all of the operations. As such, in response to the research question, MurmurHash3 is a slightly more efficient NCHF than FNV-1a in regards to their hash table implementations' collision resistance and time complexity.

7. Evaluation

There are some limitations to this investigation.

For example, all of the tested data were strings. The original intent was to test the hash tables with all types of objects, such as ArrayList and self-designed objects. However, the *toByteArray()* function from the ByteArrayOutputStream class was noticed to heavily influence the execution time depending on the data type. This distorted the experimental results, leading to the decision to only hash strings as they can be directly converted into a byte array via their characters. This solution unfortunately removed the opportunity to investigate how different data types react with the NCHFs.

Next, collision resistance and time complexity are not the only factors that determine a hashing function's efficiency. Some papers on NCHFs emphasize the importance of the avalanche effect, which is the "ability to produce a large change in output under a minimum change in the input" (Estébanez et al.). This would be a great topic for research as an extension of this paper.

Lastly, as mentioned in Section 6, there are likely systematic errors in the time complexity due to overheads. Although there are no methods to stop every other CPU activity during a test, it is still important to acknowledge the error (Lange).

Despite the above limitations, this paper provides insightful conclusions.

First of all, the conclusion that MurmurHash3 has a faster time complexity than FNV-1a agrees with pre-existing sources. For example, the below table comes from research conducted by Peter Kankowski who tested NCHFs' ability to hash all the words in Wikipedia.

	Wikipedia	Avg
iSCSI CRC	5725944 [2077725]	1.00 [1.00]
Meiyan	5829105 [2111271]	1.02 [1.02]
Murmur2	6313466 [2081476]	1.10 [1.00]
Larson	6403975 [2080111]	1.12 [1.00]
Murmur3	6492620 [2082084]	1.13 [1.00]
x65599	6479417 [2102893]	1.13 [1.01]
FNV-1a	6599423 [2081195]	1.15 [1.00]
SBox	6964673 [2084018]	1.22 [1.00]
Hanson	7007689 [2129832]	1.22 [1.03]
CRC-32	7016147 [2075088]	1.23 [1.00]
Sedgewick	7060691 [2080640]	1.23 [1.00]
XXHfast32	7078804 [2084164]	1.24 [1.00]
K&R	7109841 [2083145]	1.24 [1.00]
XXHstrong32	7168788 [2084514]	1.25 [1.00]
Bernstein	7247096 [2074237]	1.27 [1.00]
lookup3	7342986 [2084889]	1.28 [1.01]
Murmur2A	7376650 [2081370]	1.29 [1.00]
Paul Hsieh	7387317 [2180206]	1.29 [1.05]
x17 unrolled	7410443 [2410605]	1.29 [1.16]
Ramakrishna	8172670 [2093253]	1.43 [1.01]
One At Time	8338799 [2087861]	1.46 [1.01]
MaPrime2c	8428492 [2084467]	1.47 [1.00]
Arash Partow	8503299 [2084572]	1.49 [1.00]
Weinberger	9416340 [3541181]	1.64 [1.71]
Novak unrolled	21289919 [6318611]	3.72 [3.05]
Fletcher	22235133 [9063797]	3.88 [4.37]

Table 6. Execution Time and Number of Collisions of Different NCHFs (Kankowski).

In the above table, the black numbers represent the execution time in "thousands of clock cycles", while the squared bracket numbers represent the number of collisions in the hash table (Kankowski). It can be seen that while MurmurHash3 is slightly faster than FNV-1a, their number of collisions are very similar. Interestingly, most of the other commonly-used NCHFs in the table above exhibit very similar collision rates with a few inefficient exceptions.

Overall, this paper corrects a common misconception that hash tables are a flawless data structure capable of performing all operations in $O(1)$ time. In reality, the collision rate for both commonly-used NCHFs are significant, and the time complexity can also be costly, revealing several inefficiencies associated with hash tables.

Nevertheless, hash tables maintain the potential to be an extremely efficient ADT. With ongoing research and optimization, there are opportunities to reduce collision rates, minimize time complexity and improve overall performance, making hash tables a valuable choice for efficient applications in computer science.

Bibliography

Akoto-Adjepong, Vivian, et al. "An Enhanced Non-Cryptographic Hash Function." *International Journal of Computer Applications*, vol. 176, no. 15, 15 Apr. 2020, pp. 10-17, https://doi.org/10.5120/ijca2020920014. Accessed 11 Aug. 2024.

"Algorithms: Analysis, Complexity." 2010. *MIT OpenCourseWare*, Massachusetts Institute of Technology, 2010, ocw.mit.edu/courses/1-204-computer-algorithms-in-systems-engineering-spring-2010/8ee75d49f1cb9a947f1d3f15a2aa9e00_MIT1_204S10_lec05.pdf. Accessed 11 Aug. 2024. Lecture.

Appleby, Austin. "SMHasher." *Github*, 25 May 2022, github.com/aappleby/smhasher/wiki. Accessed 11 Aug. 2024.

Black, Paul E. "Dictionary of Algorithms and Data Structures." *National Institute of Standards and Technology*, U.S. Department of Commerce, 20 Apr. 2022, xlinux.nist.gov/dads/HTML/hashtab.html. Accessed 11 Aug. 2024.

Dwyl. *english-words*. *Github*, 14 July 2014, github.com/dwyl/english-words/tree/master?tab=readme-ov-file. Accessed 11 Aug. 2024.

Estébanez, César, et al. "Performance of the Most Common Non-cryptographic Hash functions." *Software: Practice and Experience*, vol. 44, no. 6, 28 Jan. 2013, pp. 681-98, https://doi.org/10.1002/spe.2179. Accessed 11 Aug. 2024.

Hashem, Ibrahim Abaker Targio, et al. "The Rise of 'big Data' on Cloud Computing: Review and Open Research Issues." *Information Systems*, vol. 47, Jan. 2015, pp. 98-115, https://doi.org/10.1016/j.is.2014.07.006. Accessed 11 Aug. 2024.

Kankowski, Peter. "Hash functions: An empirical comparison." Edited by Nils et al. *strchr*, 2008, www.strchr.com/hash_functions. Accessed 11 Aug. 2024.

Karasek, Jan, et al. "Towards an Automatic Design of Non-cryptographic Hash Function." *The 34th International Conference on Telecommunications and Signal Processing*, https://doi.org/10.1109/tsp.2011.6043785. Accessed 11 Aug. 2024.

Knuth, Donald Ervin. *The Art of Computer Programming*. 2nd ed., e-book ed., vol. 3, Reading, Addison-Wesley, 1998. PDF.

Lange, Fabian. "Measure Java Performance - Sampling or Instrumentation." *Codecentric*, 5 Oct. 2011, www.codecentric.de/wissens-hub/blog/measure-java-performance-sampling-or-instrumentation. Accessed 11 Aug. 2024.

Miessler, Daniel. *SecLists*. *Github*, 8 May 2019, github.com/danielmiessler/SecLists/blob/master/Passwords/Common-Credentials/100k-most-used-passwords-NCSC.txt. Accessed 11 Aug. 2024.

Myers, Andrew. "Hash Tables and Amortized Analysis." 2014. *Cornell Bowers College of Computing and Information Science*, Cornell University, 2014, www.cs.cornell.edu/courses/cs3110/2014fa/lectures/13/lec13.html. Accessed 11 Aug. 2024. Lecture.

Noll, Landon Curt. "FNV Hash." *Landon Curt Noll*, 28 May 2023, www.isthe.com/chongo/tech/comp/fnv/index.html#history. Accessed 11 Aug. 2024.

Pfenning, Frank, and Rob Simmons. "Lecture 12 Hash Tables." 2023. *Carnegie Mellon University Qatar*, Carnegie Mellon University, 2023, web2.qatar.cmu.edu/~mhhammou/15122-s23/lectures/13-hashing/writeup/pdf/main.pdf. Accessed 11 Aug. 2024. Lecture.

Preda, Gabriel. "BBC News." 2024. *Kaggle*, https://doi.org/10.34740/kaggle/dsv/9148879. Accessed 11 Aug. 2024. Data set.

Sanders, Peter, et al. "Hash Tables and Associative Arrays." *Sequential and Parallel Algorithms and Data Structures*, 2019, pp. 81-98, https://doi.org/10.1007/978-3-030-25209-0_4. Accessed 11 Aug. 2024.

Sullivan, David G. "Hash Tables." 19 June 2020. *Computer Science S-111 Intensive Introduction to Computer Science and Data Structures*, Harvard University, 19 June 2020, cscis111.sites.fas.harvard.edu/files/lectures/unit9-4.pdf. Accessed 11 Aug. 2024. Lecture.

Tapia-Fernández, Santiago, et al. "Key Concepts, Weakness and Benchmark on Hash Table Data Structures." *Algorithms*, vol. 15, no. 3, 21 Mar. 2022, p. 100, https://doi.org/10.3390/a15030100. Accessed 11 Aug. 2024.

Wilper, Holly, et al. "Understanding the Visualization of Overhead and Latency in NVIDIA Nsight Systems." *NVDIA Developer*, 18 Sept. 2020, developer.nvidia.com/blog/understanding-the-visualization-of-overhead-and-latency-in-nsight-systems/. Accessed 11 Aug. 2024.

Appendix

Appendix One - Raw Data of Execution Time

Size of Data	Time of Hashing (in nanoseconds)						
	Trial 1	**Trial 2**	**Trial 3**	**Trial 4**	**Trial 5**	**Average**	**Absolute Uncertainty**
100000000	207195667	161552792	164633250	163317708	163719709	172083825	17584416
130000000	211994125	211766625	213240917	213847041	210990459	212367833	1034273
160000000	281025917	259625542	260735667	259836583	259949458	264234633	8404038
190000000	312772375	310540042	308752667	312573292	308098583	310547392	1911679
220000000	357274917	357116458	373734125	356983458	357810333	360583858	6581137
250000000	650143208	456702916	486413250	405713792	435316084	486857850	85805055
280000000	653873375	465018291	478128208	459158083	486051083	508445808	73328149
310000000	503920167	504747709	502664958	534116959	508237166	510737392	11835289
340000000	550968000	551445916	551424708	553338583	560989209	553633283	3767296
370000000	612089625	613305000	600172209	616319208	607329959	609843200	5635843

Table 7. Raw Data for FNV-1a Hashing

Size of Data	Time of Hashing (in nanoseconds)						
	Trial 1	**Trial 2**	**Trial 3**	**Trial 4**	**Trial 5**	**Average**	**Absolute Uncertainty**
100000000	108777666	106222334	105733208	112192917	106501167	107885458	2394354
130000000	130044291	138632667	138849667	138395084	138975292	136979400	3473174
160000000	170407709	168924834	168660334	169987959	169822875	169560742	660862
190000000	199977750	199908625	202401375	201849875	202784166	201384358	1213823
220000000	230196458	232277000	234328583	234622333	233971666	233079208	1654838
250000000	264174417	266392250	268072792	266028375	266608125	266255192	1251246
280000000	300614958	407073791	296715125	297474916	298498625	320075483	43518893
310000000	326994292	327167375	330797291	329972333	330885000	329163258	1730724
340000000	358186583	363232292	358536042	362654834	362338625	360989675	2167920
370000000	390582250	414885042	394556750	393953333	395470042	397889483	8656775

Table 8. Raw Data for MurmurHash3 Hashing

Size of Data	Time of Searching (in nanoseconds)						
	Trial 1	**Trial 2**	**Trial 3**	**Trial 4**	**Trial 5**	**Average**	**Absolute Uncertainty**
100000000	164165833	163950167	163992000	170940917	164121541	165434092	2754562
130000000	221665583	213037291	226131375	214743875	213477958	217811216	5197254
160000000	277370125	263999166	263014625	264340250	263046000	266354033	5532626
190000000	328820542	312973458	312931333	312327209	312552959	315921100	6454198
220000000	411901500	378362917	379991750	361863875	361752959	378774600	18301646
250000000	481829500	448743459	435217208	491452500	439078541	459264242	22985578
280000000	487094042	469466458	466856000	521936250	578240583	504718667	41686555
310000000	510286833	509347417	509268208	514485250	510311083	510739758	1924666
340000000	558080292	559258292	558487250	566877041	559409375	560422450	3264170
370000000	609158458	608313208	608460167	612942500	641704416	616115750	12905354

Table 9. Raw Data for FNV-1a Searching

Size of Data	Time of Searching (in nanoseconds)						
	Trial 1	**Trial 2**	**Trial 3**	**Trial 4**	**Trial 5**	**Average**	**Absolute Uncertainty**
100000000	108813250	106397542	106483500	112934000	106262500	108178158	2558655
130000000	139692833	139233625	148635958	138504458	138728959	140959167	3860405
160000000	231109833	169897875	170301291	170490250	169797458	182319341	24396571
190000000	205679250	202630375	202175042	202674167	202416042	203114975	1294344
220000000	236492708	234324125	234512875	234532417	234443125	234861050	819083
250000000	267713291	266480416	267650208	400368083	269787417	294399883	52994809
280000000	299368334	298081750	298384834	304611917	299580292	300005425	2372055
310000000	332593375	330395209	330405500	334819584	330013833	331645500	1829560
340000000	363842375	361715333	363086083	366668917	364750042	364012550	1659198
370000000	394371459	394566750	394216791	394305750	395022167	394496583	286886

Table 10. Raw Data for MurmurHash3 Searching

Size of Data	Time of Insertion (in nanoseconds)						
	Trial 1	**Trial 2**	**Trial 3**	**Trial 4**	**Trial 5**	**Average**	**Absolute Uncertainty**
100000000	164524500	163448041	164005833	166987708	174631209	166719458	4136461
130000000	213398917	213305292	213676750	217008625	213827583	214243433	1395254
160000000	263820708	263317459	263061583	263308083	263317250	263365017	247997
190000000	315899542	309277250	311816416	312337750	312515042	312369200	2114299
220000000	362613833	362457708	364006791	370413041	384573083	368812891	8400707
250000000	418974750	436008667	437019542	470201375	447700333	441980933	16844422
280000000	490405041	500093958	486168000	492289500	489586042	491708508	4637983
310000000	505971500	509189125	508716875	512309875	507918667	508821208	2061672
340000000	614155125	558141459	557223625	557543333	560573417	569527392	22344789
370000000	607120917	850803667	638760458	612061708	607577959	663264942	94494662

Table 11. Raw Data for FNV-1a Inserting

Size of Data	Time of Insertion (in nanoseconds)						
	Trial 1	**Trial 2**	**Trial 3**	**Trial 4**	**Trial 5**	**Average**	**Absolute Uncertainty**
100000000	106063042	105883250	105904667	105930250	106310042	106018250	158757
130000000	138926667	138714083	138476833	138808792	139303416	138845958	272326
160000000	169868209	170419208	169784166	179553625	170161000	171957242	3804823
190000000	203375625	202573334	202218125	202755959	202269375	202638484	418178
220000000	235351791	239094042	233982125	234019208	234241458	235337725	1943608
250000000	266581541	266460541	265874208	266285000	267529250	266546108	546767
280000000	298994417	323519291	298024500	299314042	299485500	303867550	9838866
310000000	330735958	330449375	329579125	331175041	329705167	330328933	607844
340000000	392225667	362836334	362096250	362183541	376477042	371163767	11866297
370000000	393696416	393929667	393924500	394135833	394838208	394104925	392124

Table 12. Raw Data for MurmurHash3 Inserting

Size of Data	Time of Deletion (in nanoseconds)						
	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average	Absolute Uncertainty
100000000	329240959	328947709	329123208	330394542	328331458	329207575	671153
130000000	426825583	427137209	427586042	426857084	424971833	426675550	894454
160000000	525185875	525800834	524025541	525178791	520973833	524232975	1727762
190000000	625318500	625226708	623950542	624184958	624468291	624629800	550630
220000000	723763209	725327334	721811208	722237125	722639208	723155617	1265169
250000000	822375959	822723333	819795583	828183834	820283625	822672467	2981567
280000000	920953708	921290625	919410417	925182167	911509375	919669258	4501589
310000000	1027164084	1018988792	1012830250	1013968250	1016372917	1017864859	5108729
340000000	1118364500	1118898417	1112083000	1116715750	1113211542	1115854642	2739223
370000000	1220648459	1213845750	1211698791	1209234958	1206128834	1212311358	4897442

Table 13. Raw Data for FNV-1a Deleting

Size of Data	Time of Deletion (in nanoseconds)						
	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average	Absolute Uncertainty
100000000	213083667	213599583	213101541	214353334	212863083	213400242	534060
130000000	277564167	277556500	277088125	276791167	276505916	277101175	417682
160000000	341135791	341009292	341623584	340538750	340339292	340929342	454248
190000000	405722708	405808083	405256667	406086792	404863417	405547533	434049
220000000	469399125	471459959	469733250	468802417	469385583	469756067	903093
250000000	533841708	534039500	534689417	532706750	532887167	533632908	740138
280000000	598499667	597785833	598049042	597461791	597019667	597763200	503645
310000000	672924708	679037875	662477291	660964750	721677417	679416408	22161569
340000000	727158792	725437291	752014750	725075083	725456083	731028400	10518117
370000000	788955666	789107416	818294167	788809958	788610541	794755550	11770453

Table 14. Raw Data for MurmurHash3 Deleting

Appendix Two - Unprocessed Distribution of Buckets

The raw distribution of the buckets and execution time can be found in the below link:

https://github.com/sherrys2025/NCHF_Tests/tree/master/unprocessedData

Appendix Three - FNV-1a and Hash Table Code

```
import static java.lang.Integer.toHexString;
import static java.lang.Math.abs;

public class FNV1a {
   public static int fnv1aHash(String element) {
       return abs(fnv1aHashHelper(element.toCharArray()));
   }
   private static int fnv1aHashHelper(char[] input) {
       final int FNV_PRIME = 0x01000193;
       final int FNV_OFFSET_BASIS = 0x811c9dc5;

       int hash = FNV_OFFSET_BASIS;
       for (char b: input) {
           hash ^= (b & 0xff);
           hash *= FNV_PRIME;
           hash &= 0xFFFFFFFF;
       }

       return hash;
   }
}

import java.util.LinkedList;

public class FNV1aHashTable <String> {
   private LinkedList<String>[] hashTable;
   private int load;
   private int size;

   public FNV1aHashTable(int load) {
       size = 0;
       this.load = load;
       hashTable = new LinkedList[load];
   }

   public void add(String value){
       size++;
       int hash = FNV1a.fnv1aHash((java.lang.String) value) % load;
       if (hashTable[hash] == null) {
           hashTable[hash] = new LinkedList<>();
```

```
        }
        hashTable[hash].add(value);
    }

    public boolean contains(String value){
        int hash = FNV1a.fnv1aHash((java.lang.String) value) % load;
        if (hashTable[hash] == null) {return false;}
        return hashTable[hash].contains(value);
    }

    public void remove(String value){
        if (!contains(value)){
            throw new RuntimeException("Object not found.");
        }
        int hash = FNV1a.fnv1aHash((java.lang.String) value) % load;
        hashTable[hash].remove(value);
        size--;
    }

    public int[] getDistribution(){
        int[] buckets = new int[load];
        for (int i = 0; i < load; i++) {
            if (hashTable[i] == null) {
                buckets[i] = 0;
            } else {
                buckets[i] = hashTable[i].size();
            }
        }
        return buckets;
    }

    public int getCollision(){
        int count = 0;
        for (int i = 0; i < load; i++) {
            if (hashTable[i] != null) {
                if (hashTable[i].size() > 1) {
                    count += hashTable[i].size() - 1;
                }
            }
        }
        return count;
    }

    public int getSize() {
        return size;
    }
}
```

Appendix Four - MurmurHash3 and Hash Table Code

```
import static java.lang.Integer.toHexString;
import static java.lang.Math.abs;

public class MurmurHash3 {
   private static final int SEED = 256;
   public static int murmurHash(String element) {
       return abs(murmurHash3Helper(element.toCharArray()));
   }

   private static int murmurHash3Helper(char[] input) {
       final int c1 = 0xcc9e2d51;
       final int c2 = 0x1b873593;

       int hash = SEED;
       int len = input.length;
       int k;

       final int remainingBytes = len & 3;
       final int totalBlocks = len >>> 2;

       for (int i = 0; i < totalBlocks; i++) {
           k = getBlock(input, i * 4);
           k *= c1;
           k = rotl(k, 15);
           k *= c2;

           hash ^= k;
           hash = rotl(hash, 13);
           hash = hash * 5 + 0xe6546b64;
       }

       k = 0;
       switch (remainingBytes) {
           case 3:
               k ^= (input[len - 1] & 0xff) << 16;
               k ^= (input[len - 2] & 0xff) << 8;
               k ^= (input[len - 3] & 0xff);
               break;
           case 2:
               k ^= (input[len - 1] & 0xff) << 8;
               k ^= (input[len - 2] & 0xff);
               break;
           case 1:
               k ^= (input[len - 1] & 0xff);
               break;
       }

       if (remainingBytes > 0) {
```

```
            k *= c1;
            k = rotl(k, 15);
            k *= c2;
            hash ^= k;
        }

        hash ^= len;
        return finalizeHashValue(hash);
    }

    private static int finalizeHashValue(int hash) {
        hash ^= hash >>> 16;
        hash *= 0x85ebca6b;
        hash ^= hash >>> 13;
        hash *= 0xc2b2ae35;
        hash ^= hash >>> 16;
        return hash;
    }

    private static int rotl(int hash, int i) {
        return (hash << i) | (hash >>> (32 - i));
    }

    private static int getBlock(char[] input, int i) {
        return ((input[i] & 0xff)) |
                ((input[i + 1] & 0xff) << 8) |
                ((input[i + 2] & 0xff) << 16) |
                ((input[i + 3] & 0xff) << 24);
    }
}

import java.util.LinkedList;

public class MurmurHashTable <String> {
    private LinkedList<String>[] hashTable;
    private int load;
    private int size;

    public MurmurHashTable(int load) {
        this.size = 0;
        this.load = load;
        hashTable = new LinkedList[load];
    }

    public void add(String value){
        size++;
        int hash = MurmurHash3.murmurHash((java.lang.String) value) % load;
        if (hashTable[hash] == null) {
            hashTable[hash] = new LinkedList<>();
        }
        hashTable[hash].add(value);
```

```
    }

    public boolean contains(String value){
        int hash = MurmurHash3.murmurHash((java.lang.String) value) % load;
        if (hashTable[hash] == null) {return false;}
        return hashTable[hash].contains(value);
    }

    public void remove(String value){
        if (!contains(value)){
            throw new RuntimeException("Object not found.");
        }
        int hash = MurmurHash3.murmurHash((java.lang.String) value) % load;
        hashTable[hash].remove(value);
        size--;
    }

    public int[] getDistribution(){
        int[] buckets = new int[load];
        for (int i = 0; i < load; i++) {
            if (hashTable[i] == null) {
                buckets[i] = 0;
            } else {
                buckets[i] = hashTable[i].size();
            }
        }
        return buckets;
    }

    public int getCollision(){
        int count = 0;
        for (int i = 0; i < load; i++) {
            if (hashTable[i] != null) {
                if (hashTable[i].size() > 1) {
                    count += hashTable[i].size() - 1;
                }
            }
        }
        return count;
    }

    public int getSize() {
        return size;
    }
}
```

Appendix Five - Tests

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestFactory;

import java.io.*;
import java.util.Arrays;
import java.util.Random;
import java.util.Scanner;
import java.util.Stack;
import java.util.stream.Collectors;

public class HashTableTest {
   private static FNV1aHashTable<Object> fnvHashTable;
   private static MurmurHashTable<Object> murmurHashTable;
   private Stack<String> englishWords;
   private Stack<String> passwords;
   private Stack<String> news;

   private void printFNVCollisions(){
       int collisions = fnvHashTable.getCollision();
       int numBuckets = fnvHashTable.getSize();
       System.out.println("Number of collisions: " + collisions);
       System.out.println("Number of buckets: " + numBuckets);
       System.out.println("Collision Rate: " + collisions*1.0/numBuckets);
   }

   private void printMurmurCollisions(){
       int collisions = murmurHashTable.getCollision();
       int numBuckets = murmurHashTable.getSize();
       System.out.println("Number of collisions: " + collisions);
       System.out.println("Number of buckets: " + numBuckets);
       System.out.println("Collision Rate: " + collisions*1.0/numBuckets);
   }

   @TestFactory
   static void reset(int size){
       fnvHashTable = new FNV1aHashTable<>(size);
       murmurHashTable = new MurmurHashTable<>(size);
   }

   @TestFactory
   void resetDictionary(){
       try (BufferedReader reader = new BufferedReader(new FileReader("src/words.txt"))) {
           englishWords = new Stack<>();
           String line;
           while ((line = reader.readLine()) != null) {
               if (!englishWords.contains(line)){
                   englishWords.push(line);
               }
```

```
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        reset(englishWords.size());
    }

    @TestFactory
    void resetPasswords(){
        try (BufferedReader reader = new BufferedReader(new
FileReader("src/passwords.txt"))) {
            passwords = new Stack<>();
            String line;
            while ((line = reader.readLine()) != null) {
                if (!passwords.contains(line)){
                    passwords.push(line);
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        reset(passwords.size());
    }

    @TestFactory
    void resetNews(){
        try (BufferedReader reader = new BufferedReader(new
FileReader("src/bbc_news.csv"))) {
            news = new Stack<>();
            String line;
            while ((line = reader.readLine()) != null) {
                if(!news.contains(line)){
                    news.push(line);
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        reset(news.size());
    }

    @TestFactory
    static String createRandomString(int size){
        return new Random().ints(48, 123)
                .filter(i -> (i <= 57 || i >= 65) && (i <= 90 || i >= 97))
                .limit(size)
                .mapToObj(i -> String.valueOf((char)i))
                .collect(Collectors.joining());
    }

    @TestFactory
```

```
    void collisionResistanceDistributionFile(String dataset, int[] distribution){
        FileWriter myWriter;
        try {
            myWriter = new FileWriter("unprocessedData/distributionOf" + dataset + ".txt");
            for (int i: distribution) {
                myWriter.write("" + i + "\n");
            }
            for (int i = 0; i < 100; i++){
                myWriter.write("\n");
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Test
    void collisionResistanceDictionaryTest(){
        resetDictionary();
        System.out.println("FNV-1a: ");
        while (!englishWords.isEmpty()) {
            fnvHashTable.add(englishWords.pop());
        }
        printFNVCollisions();
        collisionResistanceDistributionFile("DictionaryFNV",
fnvHashTable.getDistribution());

        resetDictionary();
        System.out.println("MurmurHash3: ");
        while (!englishWords.isEmpty()) {
            murmurHashTable.add(englishWords.pop());
        }
        printMurmurCollisions();
        collisionResistanceDistributionFile("DictionaryMurmur",
murmurHashTable.getDistribution());
    }

    @Test
    void collisionResistancePasswordsTest(){
        resetPasswords();
        System.out.println("FNV-1a: ");
        while (!passwords.isEmpty()) {
            fnvHashTable.add(passwords.pop());
        }
        printFNVCollisions();
        collisionResistanceDistributionFile("PasswordsFNV",
fnvHashTable.getDistribution());

        resetPasswords();
        System.out.println("MurmurHash3: ");
        while (!passwords.isEmpty()) {
            murmurHashTable.add(passwords.pop());
```

```
        }
        printMurmurCollisions();
        collisionResistanceDistributionFile("PasswordsMurmur",
murmurHashTable.getDistribution());
    }

    @Test
    void collisionResistanceBBCNewsTest(){
        resetNews();
        System.out.println("FNV-1a: ");
        while (!news.isEmpty()) {
            fnvHashTable.add(news.pop());
        }
        printFNVCollisions();
        collisionResistanceDistributionFile("BBCNewsFNV", fnvHashTable.getDistribution());

        resetNews();
        System.out.println("MurmurHash3: ");
        while (!news.isEmpty()) {
            murmurHashTable.add(news.pop());
        }
        printMurmurCollisions();
        collisionResistanceDistributionFile("BBCNewsMurmur",
murmurHashTable.getDistribution());
    }

    @Test
    void hashingTimeComplexityTest() {
        FileWriter myWriter;
        try {
            myWriter = new FileWriter("hashingTimeComplexity.txt");
            for(int i = 0; i < 6; i++){
                for (int j = 100000000; j < 400000000; j += 30000000) {
                    String randomString = createRandomString(j);
                    long startTime = System.nanoTime();
                    FNV1a.fnv1aHash(randomString);
                    long endTime = System.nanoTime();
                    myWriter.write(j + ": \n" + (endTime - startTime) + " nanoseconds\n");

                    startTime = System.nanoTime();
                    MurmurHash3.murmurHash(randomString);
                    endTime = System.nanoTime();
                    myWriter.write((endTime - startTime) + " nanoseconds\n\n");
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Test
```

```
void insertTimeComplexityTest() {
    FileWriter myWriter;
    try {
        myWriter = new FileWriter("insertionTimeComplexity.txt");
        for(int i = 0; i < 6; i++){
            for (int j = 100000000; j < 400000000; j += 30000000) {
                reset(10);
                String randomString = createRandomString(j);
                long startTime = System.nanoTime();
                fnvHashTable.add(randomString);
                long endTime = System.nanoTime();
                myWriter.write(j + ": \n" + (endTime - startTime) + " nanoseconds\n");

                startTime = System.nanoTime();
                murmurHashTable.add(randomString);
                endTime = System.nanoTime();
                myWriter.write((endTime - startTime) + " nanoseconds\n\n");
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

@Test
void containsTimeComplexityTest() {
    FileWriter myWriter;
    try {
        myWriter = new FileWriter("containsTimeComplexity.txt");
        for(int i = 0; i < 6; i++){
            for (int j = 100000000; j < 400000000; j += 30000000) {
                reset(10);
                String randomString = createRandomString(j);
                long startTime = System.nanoTime();
                fnvHashTable.contains(randomString);
                long endTime = System.nanoTime();
                myWriter.write(j + ": \n" + (endTime - startTime) + " nanoseconds\n");

                startTime = System.nanoTime();
                murmurHashTable.contains(randomString);
                endTime = System.nanoTime();
                myWriter.write((endTime - startTime) + " nanoseconds\n\n");
            }
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

@Test
void removeTimeComplexityTest() {
```

```
        FileWriter myWriter;
        try {
            myWriter = new FileWriter("removeTimeComplexity.txt");
            for(int i = 0; i < 6; i++){
                for (int j = 100000000; j < 400000000; j += 30000000) {
                    reset(10);
                    String randomString = createRandomString(j);
                    fnvHashTable.add(randomString);
                    murmurHashTable.add(randomString);

                    long startTime = System.nanoTime();
                    fnvHashTable.remove(randomString);
                    long endTime = System.nanoTime();
                    myWriter.write(j + ": \n" + (endTime - startTime) + " nanoseconds\n");

                    startTime = System.nanoTime();
                    murmurHashTable.remove(randomString);
                    endTime = System.nanoTime();
                    myWriter.write((endTime - startTime) + " nanoseconds\n\n");
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Test
    void countDistributions(){
        String[] datasets = {
                "DictionaryFNV", "DictionaryMurmur",
                "PasswordsFNV", "PasswordsMurmur",
                "BBCNewsFNV", "BBCNewsMurmur"
        };
        for (String dataset: datasets) {
            int[] count = new int[10];
            try (Scanner scanner = new Scanner(new File("unprocessedData/distributionOf" +
dataset + ".txt"))) {
                while (scanner.hasNextLine()) {
                    String num = scanner.nextLine();
                    count[Integer.parseInt(num)]++;
                }
                System.out.println(dataset + ":");
                System.out.println(Arrays.toString(count));
            } catch (FileNotFoundException e) {
                System.out.println("File not found: ");
                e.printStackTrace();
            }
        }
    }

}
```