# An Imperial Investigation of Hash Functions

## Andrew Montgomery

# Introduction

Consider the problem of searching an array for a given value. If the array is unsorted, the worst-case scenario requires every value in the array to be examined in order to return the value being searched for resulting in O(n) time. If the array is sorted, then a binary search would speed up the process and the search would be completed in O(log n). While the binary search does speed up the process, it is limited by the need for a sorted array. However, by utilizing hashing the search time for a given value is reduced to O(1) because each value in the array is assigned a specific key that allows for instant access if the key of the value is known.

While there is clearly a huge advantage of using hashing to lookup values in an array, there are also certain challenges that hashing presents. The first challenge is that because hash tables use an array as the underlying data structure, the hash table must be large enough to contain all the entries that need to be stored. However, there are multiple strategies to dynamically increase the size of a hash table.

The other primary challenge is how to deal with collisions within the hash table. This report uses separate chaining as the method for collision resolution. When using separate chaining as the collision resolution scheme, when a new element is added to the chain, it is placed at the beginning of the link. This enables the time efficiency of the add function to be O(1), however because each key has a linked list of nodes, the retrieve and remove functions become less efficient as the length of the chain increases. Thus, the performance of the hash function is determined by the average length of chain generated when all the elements are added to the hash table; the longer the chain, the less efficient the hash function is.

Hashing is a technique used to assign a unique identifier (key) to a specific object from a group of similar objects. A hash function is a method used to generate a key for a specific object and each key and object pair is stored in a hash table. Hashing can be seen utilized in the real world for systems such as education records (assigning a student number to each student), libraries (assigning a unique number to each book), through the 911 call system (where the phone number is the search key), along with many more.

Given the specifications of this assignment; a hash function that uses separate chaining as the collision-resolution scheme, the ideal hash function would be if every index in the hash table had the same amount of chain lengths. In this example, because there are 46,332 names and telephone numbers in the file, and the hash table holds 4177 entries, then each index would have a chain length of 11 (46332/4177 = 11.09). The standard deviation can be used to determine how uniformly the data is spread from the average chain length. A low standard deviation means that most of the data (chain lengths) are close to the average, while a higher standard deviation means that the data is more spread out.

The goal of this report is to identify and describe three different hash functions, display the results of each hash function using a histogram, and discuss whether the given hash function was ideal.

# Hash Function #1

The first hash function was used to create a baseline using a simple modulus equation. The hash key was determined by adding the sum of the ascii value of the first character in a person’s first name and the first character in the last name to the sum of the person’s zip code and phone number.

## Pseudocode

Set hashKey value to 0

Set firstLetter equal to the ascii value of the first character in the person’s first name

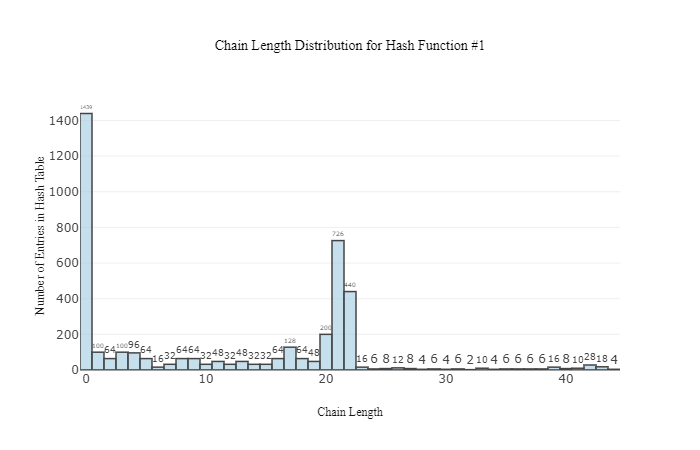
Set secondLetter equal to the ascii value of the first character in the person’s last name

Set hashKey equal to the sum of the person’s zip code and phone number added to the sum

of firstLetter and secondLetter

Return hashKey modulus the size of the hash table

## Results



## Explanation

Hash function #1 is clearly lacking in terms of performance. The results show that about 1/3 of the hash table has no entries (1439 indices have no entries). The remaining locations in the hash table are relatively even spread, except where there is a significant number of entries of chain lengths 20-22, making up roughly 1/3 of the entries as well. Also, the hash function is less than ideal because the hash function is determined only off the first character of the first and last name, which is limited by the number of possible first letters. The histogram clearly shows that this hash function is far from ideal because the hash function does not uniformly distribute the data across the entire sets of possible hash values. The standard deviation for Hash Function #1 is 10.75, meaning that the chain lengths were more spread out from the average (11).

# Hash Function #2

The second hash function chose was a further iteration of the first hash function. Rather than choosing only two values in the string that represented a person’s first and last name (the first character of the first name, and the first character of the last name), this hash function used the sum of all of the ascii values for each character in a person’s first and last name

## Pseudocode

Set hashKey value to 0

Set personString equal to the combination of the Person’s first and last name.

**For** (loop from the start of personString until the end)

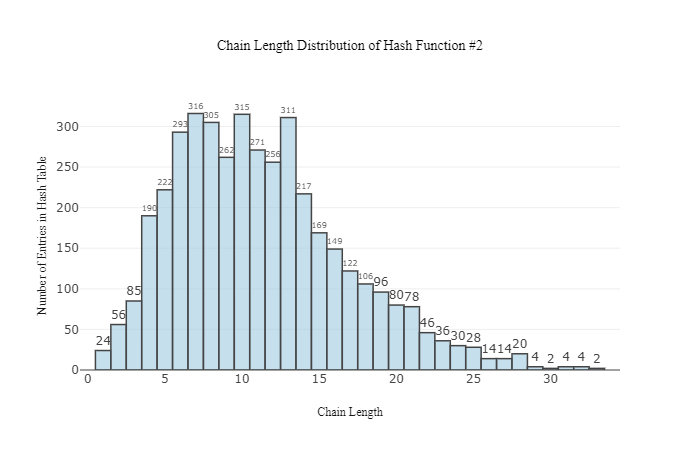
Add the ascii value of char in personString at position i to hashKey

**End For Loop**

Add (zipCode + phoneNumber) to hashKey

Return hashKey modulus TABLESIZE

## Results



## Explanation

The results of Hash Function #2 are an improvement over Hash Function #1 as the chain lengths are more uniformly spread across the hash table. The hash table does not have any indices with a chain length of 1, and most chain lengths are centered around a chain length of 11, as was determined earlier to be the value a hash function is aiming for to have perfect distribution of chain lengths. This hash function also uses every character or integer provided by the input, which greatly expands the bounds of the possible hash keys. However, the hash function can be improved upon because while the distribution is better than Hash Function #1 it is still far from ideal because the histogram clearly shows that the chain lengths are not distributed evenly. The standard deviation for Hash Function #2 is 5.5, which is a clear improvement from Hash Function #1, but still shows that the chain lengths are relatively spread out from the mean.

# Hash Function #3

The third hash function used was once again aimed to be an improved iteration of the first and second hash functions. For this hash function, the same strategy of utilizing every value provided by the input was maintained, but also went a step further by modifying the values in each iteration of the for loop. Specifically, the hash function multiplied each ascii value of the string containing the person’s first and last name by the prime number 31, and then at each point, the bits of the current hash key value are shifted. The goal of doing these extra steps was to further minimize the number of collisions and bring the overall result of chain lengths closer to the expected average.

## Pseudocode

Set hashKey value to 0

Set personString equal to the combination of the Person’s first and last name.

**For** (loop from the start of personString until the end)

Add absolute value of the ascii value of char in personString at position i multiplied by 31 to hashKey

Shift the bits in the hashKey using the Shift() function

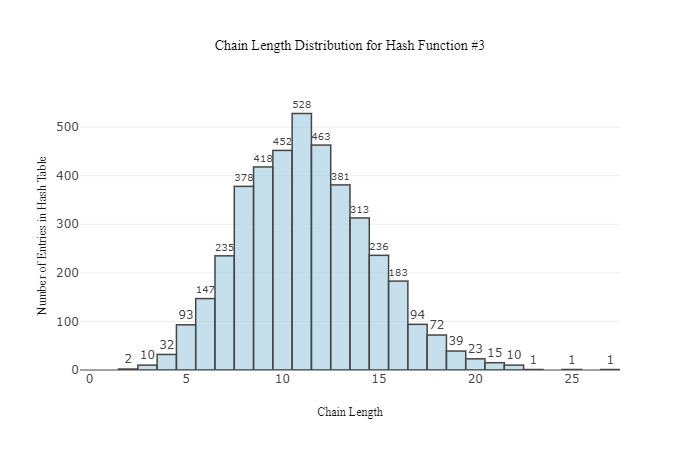
**End For Loop**

Add (zipCode + phoneNumber) to hashKey

Shift the bits in the hashKey using the Shift() function

Return hashKey modulus TABLESIZE

## Results



## Explanation

The results of Hash Function #3 show a clear improvement over the previous hash functions. Based upon the histogram above, the distribution of the chain lengths is clearly centered around the expected average of 11 and show a vast improvement over the lengths of each tail. The standard deviation for Hash Function #3 was 3.35, further showing that this hash function is superior to the other hash functions.

# Conclusion

The results from the three different hash functions show that as the complexity of the hash function increases, the performance of the hash function also increases. Hash Function #1 was the simplest, and didn’t utilize every value from the input, and thus resulted in the highest deviation from the mean. As the amount of data from the input was increased and the number of operations on that data was also increased, the deviation became closer to the mean. The report clearly shows why Hash Function #3 would be the most optimal choice out of the three hash functions explored because it has the lowest deviation from the expected mean.