Ruth Tilahun
CMSC 337
Lab 5

Hash Tables and Hash Functions – Collision Analysis

One of the most common and apparent operations that follow data collection and organization is data retrieval. We want to access and use the data we collected in the most efficient way possible. There are several searching methods to retrieve data. One way to access data is by going through the entire dataset until we come across the one we are looking for. If we use this method, in the worst case, we might have to go through the entire list, which isn't ideal for large datasets. If we are looking for some item in a list and we know where exactly it is located, then it is much easier to access it. Hashing lets us do this by mapping data to a unique index in a hash table using hash functions. In this lab we will compare performances of Java's built-in hash function with our own hash function which we will implement using Horner's method.

Hash Functions

Hash functions convert general keys, to corresponding indices in the range [0, N-1], where N is the capacity of the hash table. The main goal is to use the value of the hash function as in index into the array instead of directly using the key as it may not be appropriate for direct use as an index. In many cases, the mapped indices will hold some value that is associated with the key. As an example, a social media site relies on usernames which aren't typically numeric, as keys that can be efficiently mapped to users' associated information(values). In essence, we are using the key like an "index" to access the value associated with it.

The process of turning a key into an index involves two steps. The first step converts keys into a numeric code and the second step maps the code into an integer within the range of the size of the hash table. A problem arises when two or more different keys map to the same index. This is called collision. Collisions occur mainly due to bad hash functions that don't distribute keys well. A good hash function maps the keys so as to sufficiently minimize collisions. If our hash function is good, then we expect the indices to be uniformly distributed in the N cells of the table. Thus to store n items, the expected number of keys in a table would be n/N, which is $\Theta(1)$ if n is $\Theta(N)$. In the worst case, a poor hash function could map every key to the same index. This would result in linear-time performance for the operations that deal with collision (Goodrich et al., 2014). Two of the most widely used methods for dealing with collisions are open addressing and separate chaining. In open addressing, upon a collision, the

algorithm searches for the next available slot in the hash table until an empty slot is found. In separate chaining, each cell in the hash table contains a separate data structure to store values whose keys hash to the same index.

Collision Performance of Java's Hash Function

For this experiment, we used a dataset of approximately 69000 unique words and mapped each word to an index in array to see how many times distinct words are hashed to the same index. We used Java's built-in method, hashCode() to generate a hash code for each word, then mapped the hash code to an entry in an array of size N, where N is the number of words in the dataset. The value at an index will be the number of words the were mapped to that index.
The outcome of this resulted in approximately 25000 words being mapped into the same location. The percentage of spaces used in the array, known as load factor, was around 60%.
To extend the experiment, we increased the size of the array, N, by 70,000 until we achieved collisions less than 100. The relationship between N and the number of collisions looks like the following:

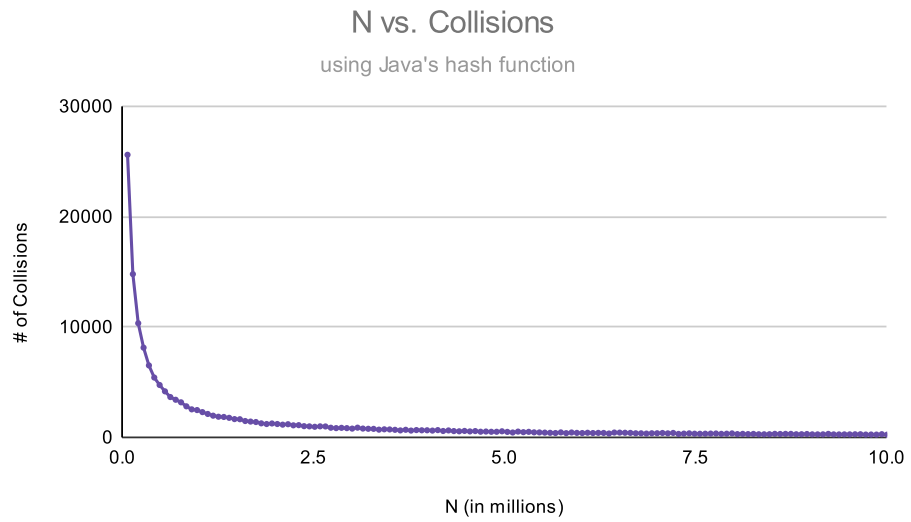

Figure 1: relationship between size of the array and the number of collisions

In the figure above, we can observe a big difference in the number of collisions by only doubling the size of N. With consequent increases in N, the number of collisions decrease at a diminishing rate. Therefore, after some increase in N, the effect of the size of the array on the number of collisions isn't significant. Therefore, just increasing the size of the array isn't going to fix the problem of collisions as they are essentially unavoidable.

Additionally, increasing the size of the array until we reach zero collisions isn't space efficient. When dealing with large datasets it is very costly to make the size of the array a hundred or more times bigger than the dataset.

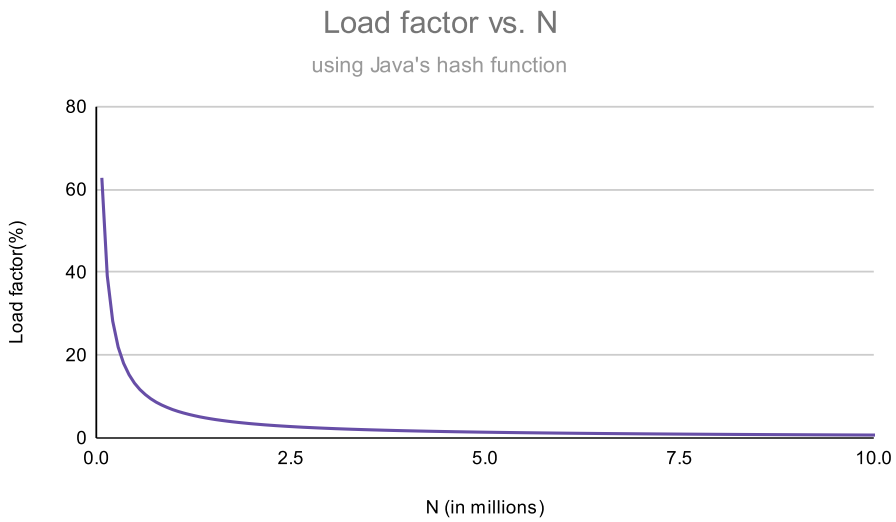

Figure 2: relationship between load factor and the size of the array using Java's hash function

Figure 2 further supports the idea that there is a point where increasing the size of the array wouldn't make a big difference in collisions. We can see here that, as N increases the percentage of used space in the array decreases to the point where additional increases in N don't affect the load factor as much.

Collision Performance of Our Hash Function

We used Horner's method of hashing to implement our own hash function. Horner's method calculates hash codes for each character in a string and aggregates their value as a hash code for the entire string.

```
Given S : a string
Result: an integer in the full range of integers
Let:  hash=0
      mul= a prime number > 50
With each ch in the characters of the string
      Let vch = integer value of the character
      hash = hash*mul + vch
      return hash
```

We first choose a prime number as the base for the hash computation. Using a prime number helps to get hash codes that are unique which minimizes collisions and promotes a more distributed set of hash codes. Particularly, prime numbers

are useful because the product of a prime number with any other number has a better chance of being unique since prime numbers are themselves unique.
Then for each character of the string, we calculate the hash code by multiplying the current hash value by the prime number.
Using this hash function we repeated the experiment we did with Java's hash function, to find result which are more or less identical.

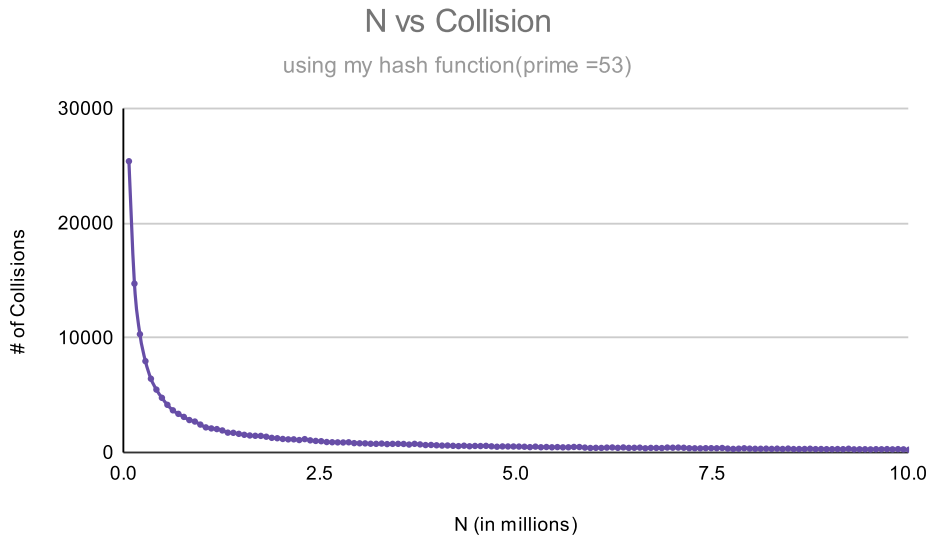

Figure 3: relationship between size of the array and the number of collisions using Horner's method as our hash function

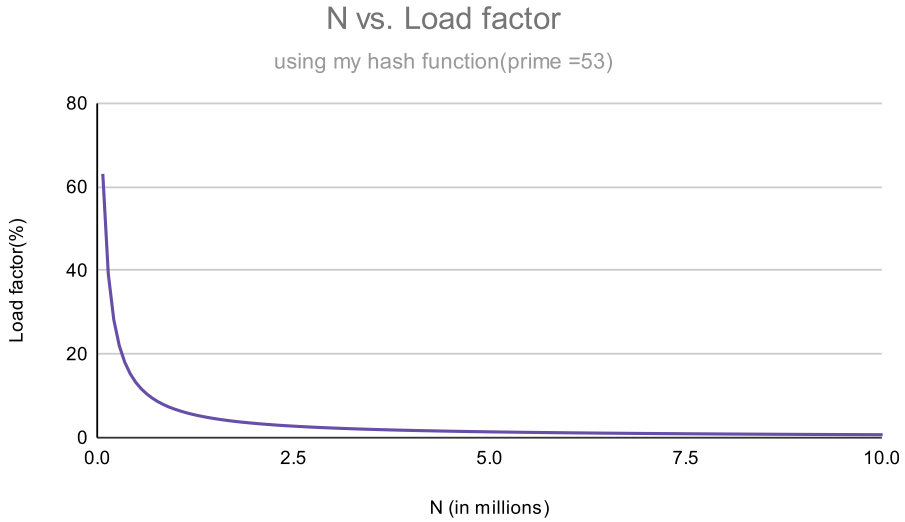

Figure 4: relationship between load factor and the size of the array using Horner's method as our hash function

The conclusion from this experiment is that increasing the size of the hash table reduces the number of collisions. However, to avoid collisions completely using such technique comes with a very high cost of memory. Instead, when we reach a certain reasonable size, we should opt for other methods such as separate chaining and open addressing, to efficiently cope with collisions.

References

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data Structures and Algorithms in Java*. Wiley.

Appendix A: Sample Runs

Task 1

```
The concordance has 69568 unique words
Query the Concordence
Word to ask about (hit return to quit): headphones
        headphones is not in the concordance.
Word to ask about (hit return to quit): vscode
        vscode is not in the concordance.
Word to ask about (hit return to quit): the
        the appears 265962 times
Word to ask about (hit return to quit): battle
        battle appears 774 times
Word to ask about (hit return to quit): julius
        julius appears 60 times
Word to ask about (hit return to quit):
```

Task 2

```
Bryn Mawr-> 978918218
k-Cass-> -1187996800
Haverford-> 612402085
Swarthmore-> -698967618
```

Task 4

```
Bryn Mawr -> -1892612766
k-Cass -> -215128534
Haverford -> -651867119
Swarthmore -> -1269718988
```

Appendix B: Code

```
public static void main(String[] args) {
        int N = 70111;
        int totalCol = 0;
        for (int i = 0; i < args.length; i++) {
            readFile(args[i]);
        }
        for (int j = 0; j < 100; j++) {
            int[] collision= new int[N];
            totalCol = collisionCount(collision, N);
            System.out.print(totalCol + ",");
            int usedSpace = 0;
            for (int i = 0; i < collision.length; i++) {
                if (collision[i] >= 1) {
                    usedSpace++;
                }
            }
            System.out.println(usedSpace);
            N += 70069;
        }
    }
    public static int collisionCount(int[] collision, int N) {
        int index;
        int colCount = 0;
        int totalCol = 0;
        for (String key : concordance.keySet()) {
            //System.out.print(key + " ");
            index = myHash(key, N);
            if (collision[index] == 0) {
                colCount = 1;
                collision[index] = colCount;
            } else {
                colCount = collision[index] + 1;
                collision[index] = colCount;
                totalCol++;
            }
        }
        return totalCol;
    }

    public static int javaHash(String word, int N) {
        int hashCode = Math.abs(word.hashCode()) % N;
        return hashCode;
    }
```

```
public static int myHash(String key, int N) {
    int index = 0;
    int  hash = 0;
    int mul = 31;
    for (int i = 0; i < key.length(); i++) {
        hash = (hash * mul) + key.charAt(i);
    }
    index = Math.abs(hash) % N;
    return index;
}
```