University of São Paulo
Institute of Mathematics and Statistics
Bachelor's in Computer Science

Breno Helfstein Moura

Hashing Functions and Hash Tables
A practical approach

São Paulo
November de 2019

Hashing Functions and Hash Tables
A practical approach

Final undergraduate thesis for subject
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: Prof. Dr. Jose Coelho de Pina Junior
[Cosupervisor: Prof. Dr. Nina S. T. Hirata]

São Paulo
November de 2019

Abstract

During this undergraduate thesis I will explain about two of the most fascinating and used ideas in Computer Science, hash functions and hash tables. I divided this thesis in three main parts:

- Hash functions
- Hash tables
- Applications

During the first part I explain why hash functions are an important idea in Computer Science, summarize some of the ideas Donald Knuth present on his book (The Art of Computer programming, Vol. 3) and use some metrics to evaluate what is a good hash function.

During the second part I talk about one of the most used data structures in computer programming, hash tables. I will explain what constitute a hash table, show some of the classic implementations of this data structure and explain some of the most used open addressing strategies. It is nice to observe here that although hash tables is a simple concept, there is still debates regarding this subject with no clear consensus on what is a state of the art hash table.

During the third, and last, part I will cite and explain some application of hash functions in computer science problems. I will explain Rabin-Karp, a string search algorithm that uses hashing and a solution to identify isomorphisms on trees using hashing functions.

I hope this is as fun to read for you as it was for me to write!

Keywords: hash functions, hash tables, collision-resolution, open-addressing.

Contents

1 Introduction **1**

2 Hash Functions **3**

3 Hash Tables **7**

4 Applications **9**

4.1 3-sum problem 9

5 Conclusions **13**

A Apendix **15**

Chapter 1

Introduction

One of the most used data structures in computer science are dictionaries. Those need to suport the operations of inserting, fiding and deleting an element. If you think about it, this is one of the most executed tasks in many softwares. For example, when you have the list of numbers you last called on your cell phone and you want to know for each phone number, what is the person associated with it. A dictionary perfoms the task of inserting for each phone number the name of the person, and then you can retrieve that information finding, for a phone number who is the person associated with it.

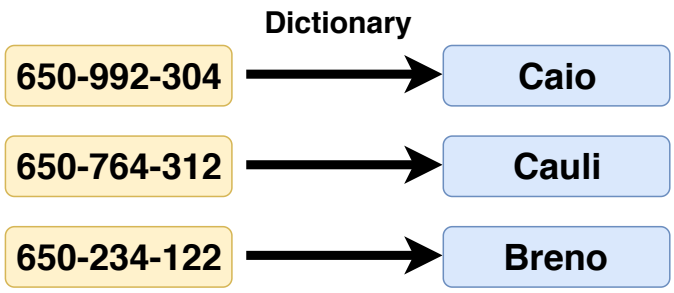

Figure 1.1: *Example of a dictionary that associates phone numbers to contact names.*

Other use of a dictonary that you can think is to count the number of times you called a certain number. One of the most used implementation of dictionaries is with a hash table.

The implementation of a hash table always requires a hash function. This function usually takes the element that you want to hash, or key as it is usually called (In our example, the phone numbers), and "digest" it into a number. That number is then used to indentify the value (In our eaxmple, the contact names), in this structure that we call hash table.

An example of a hash function, that "digest" the phone numbers is the following:

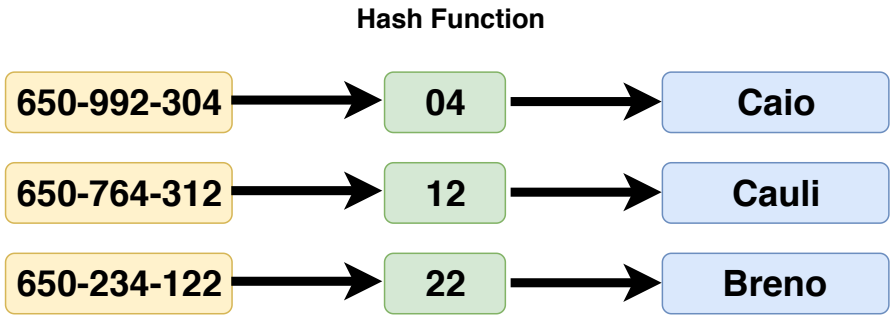

Figure 1.2: *Example of a hash function hash function that just take the last 2 digits of the phone number*

As you can see this is a pretty simple function, it simply take the last 2 digits of each phone number. In this specific case, this is enough to uniquely identify each phone. We can imagine a function that can't uniquely identify each phone number, like getting just the last digit (in this case, Cauli's and Breno's numbers would have the same hash value), this will cause a collision in the table. Solving collisions in a hash table is a complete topic by itself, and it will be addressed in Chapter 3, Hash Tables.

Solving collisions is actually a very important topic in hash tables, and that is because the vast majority of hash functions will have collisions. To picture that we can remember the "Birthday Paradox", that is the conclusion that we only need 23 people in a room to have a chance greater than 50% of 2 or more people having the same birthday. In Donald Knuth's famous book, The Art of Computer Programming (Vol. 3, Chapter 6.4) [**?**], he uses as an example a function from a 31-element set to a 41-element set, and from about 10^{50} functions only about 10^{43} give distinct values for each argument, that is about 1 in every 10 million functions. That shows that we will have collisions more often than not, so knowing how to deal with it is a major problem.

Hash functions and hash tables are among the most classic topics within computer science, yet is still one of the topics with most debate about what is state of the art. While the hash table was invented in 1953, widely discussed by Donald Knuth in his book, there are still many tweaks that can be made to boost its performance for specific use cases. One great example is F14, an open-source memory efficient hash table by Facebook [1].

An example of lack of consensus in this area are the different hash functions and hash table implementations in different languages. There is no clear consensus on how to decide the size of a hash table, what are the tradeoffs of the collision-resolution algorithms or even what defines a good hash function. Hopefully, we got years of research on the topic to study and present a view on the subject, and that is what I am presenting thoughout this undegraduate thesis.

[1] F14 is open sourced: https://engineering.fb.com/developer-tools/f14/

Chapter 2

Hash Functions

Outside computer science, the word *"hash"* in the english language means to "chop" or to "mix" something. This meaning is entirely related to what hash functions are supposed to do. hash functions are functions that are used to map data of an arbitrary size to data of a fized size [**?**].

They have wide applications in computer science, being used in information and data scurity, compilers, distributed systems and hardcore algorithms. During this chapter I first define and explain the basics of a hash function, then I give an intiution in some metrics of what is a good hash function, as discussed in the famous *"Red Dragon Book"* [**?**] along with some reproduction of known results in the area.

The value extracted from the hash function for an object is usually called *Hash Value*. The hash value is usually, but not necessarily, smaller than the object that generated it. For example, we can have a hash function that takes Gigabytes or Terabytes files and return an 8 bytes hash value.

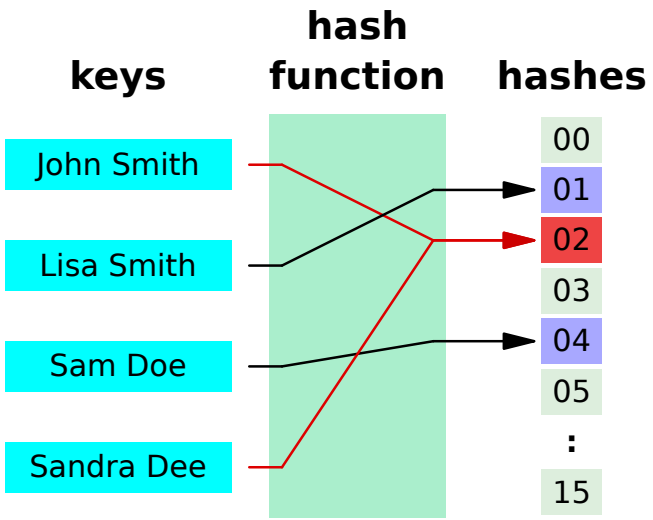

Figure 2.1: *Example of a hash function from string to 4 bit integer.*

To formalize a little, lets define a hash function as a function H that takes an element $x \in X$ and has $[0, M)$ as a codomain.

This is the same definition used by Donald Knuth [**?**] and some articles [**?**]. This definition makes sense for our case because we will be talking mostly about hash functions used in hash tables, and in that case we want integers that will be indexes in an array (as we will se later on). In other cases will may see hash functions value as strings, like for when we hash an string for password storage or when we use a hash function in files for check-sum (for when we are checking if two files are the same). For the goal of this thesis we will not be focusing on those functions, but it is important to notice that strings can also be abstracted to integers if we just look at the bytes.

For our specific case we are looking at a hash function that is good for the construction of hash tables, that is that is fast to calculate and minimizes the number of collisions. Depending on our goals we might want a different metric, for check-sums for example we may want a function that is very sensible to chages, and for passwords one that is very hard to find its inverse.

As said in Donald Knuth's book, we know that it is theoricatically impossible to create a hash functio that generates true random data from non random data in actual file, but we can do pretty close to that (or in some cases, even better). Donald Knuth describes 2 specific methods for simple hash function, named *division hashing* and *multiplicative hashing* techniques. As the name sugests, the first is based on division and the former on multiplication.

The divison hashing method simply to represent the data as a number take the remainder of that number modulo a value. Supposing that we can represent the data as an integer X the division hashing would be to choose a value M and the hashing function would be $XmodM$. The C++ code would look as following:

```
1 int divisionHashing(int X, int M) {
2   return X % M;
3 }
```

In general large prime numbers tend to be a good value to M, because if not we may have repetitions. One great example of this is if M is even, then the parity of hash value of X will match the parity of X (which will cause a bad distribution). The same pattern will happen in different intervals for different powers of 2.

For the multiplicative hashing, we can furst imagine that the overflow is like a "natural" modulo operation (We also have methods to take the modulo without overflowing, to know more about that). Supposing that we can represent the data as an integer X, the multiplication hashing would be to choose a value A that we mulitply by X and then take the value module M (That M is described as a power of 2 by Donald Knuth, so it would be a more "General" multiplicative method). The C++ code would look as follwing:

```
1 int multiplicativeHashing(int X, int A, int M) {
2   return (A * X) % M;
3 }
```

As you can see from above multiplicative hashing is just a more general division hashing. In knuth's book he restricts A to be relatively prime to w, being w the size of a "word" in the machine (which is MAX_INT in our case). That definition is often useful if you can retrieve a value Y which you have $H(Y) = H(X)$. It is good to note here that X is not necessarily equal to Y, as we can have two keys having the same hash value.

Chapter 3

Hash Tables

- Define hash table and its operations
- Open Addressing Strategies (Linear Probing, Quadratic Probing, ...)
- Chaining Strategy (Simple Chaining, Move-to-front ...)
- Load factor and resizing/rehashing the table

Hash tables or hash maps is one of the most used applications of hash functions. It is actually so used in computer science that is almost impossible to talk about one without mentioning the other. This data structure consists in associating a *key* to a *value* in a table. That is, given a *key*, it can retrieve the correct *value* for it.

This data structure is usually considered very useful among software engineers and computer scientists, although it usually has a linear worst case cost for retrieving, inserting and deleting a key a value pair. That is because it has a constant average and amortized cost for those operations.

Moreover, when talking about hash tables we have the problem of key collision, that is when two keys maps to the same hash value. To solve that problem, we have several techniques that envolves different tradeoffs. Those techniques are usually divided into two main categories, open addressing and separate chaining. Other problem to consider regarding this data structure is when to resize the hash table, to minimize the chance of collision and the use o memory.

It is also important to notice that hash tables have applications in different areas of computer science also, like compilers, caches and database indexing.

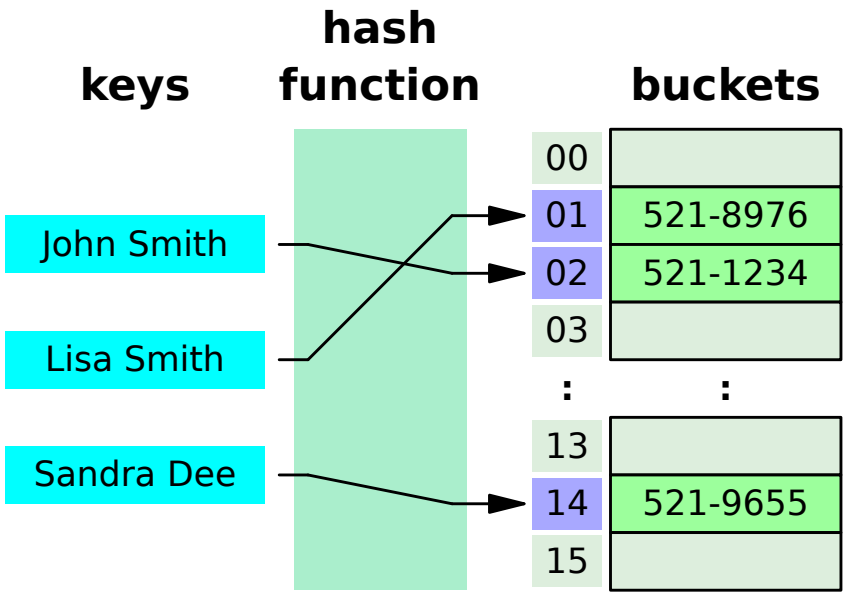

Figure 3.1: *Example of a hash table from string to string, more specifically name to number*

Chapter 4

Applications

- Give a glance at what type of application we have, focus on 2 algorithimic
- Rabin Karp
- Hashing trees

Hash functions and hash tables have a great number of applications in computer science. During this last section I focus on applications of hash functions in algorithms, but citing superficially applications in other areas (like criptography, data deduplication and caching).

Among the applications that I explain in this section there is a focus in two applications: Rabin-Karp string matching algorithm and hashing of a rooted tree for isomorphism checking. Rabin-karp string matching algorithm is one of the main application of a technique called rolling hashing. Hahsing of rooted tree for isomorphism checking is an interesting application sometimes used in competitive programming.

To motivate the start of this thesis I will start using hash tables to solve a very simple, yet famous, problem called 3-sum.

4.1 3-sum problem

The problem is stated as following:

Make a function that given an array of integer numbers and an integer S, it returns if there are any 3 different elements in this array that its sum equals S. Assume that there are no three different elements in the array that overflow a 32-bit integer when summed together.

This a very interesting problem that has many different solutions. To start I will show and explain to you the brute force solution:

```cpp
bool threeSumWithoutHashTable(vector<int> v, int S) {
    for (int i = 0; i < v.size(); i++) {
        for (int j = i + 1; j < v.size(); j++) {
            for (int k = j + 1; k < v.size(); k++) {
                if (v[i] + v[j] + v[k] == S) return true;
            }
        }
    }
    return false;
}
```

The above solution solves the problem in $O(n^3)$ time complexity and $O(1)$ memory complexity, being n the size of the array. It don't allocate any memory but checks every triple to find if one satisfy the condition. The question is, can we do better in time complexity using hash tables? The answer is yes:

```cpp
bool threeSumWithHashTable(vector<int> v, int S) {
    unordered_map<int, int> hashTable;
    for (int i = 0; i < v.size(); i++) {
        hashTable[v[i]]++;
    }
    for (int i = 0; i < v.size(); i++) {
        for (int j = i + 1; j < v.size(); j++) {
            hashTable[v[i]]--;
            hashTable[v[j]]--;
            if (hashTable.find(S - v[i] - v[j]) != hashTable.end() &&
                hashTable[S - v[i] - v[j]] > 0) return true;
            hashTable[v[i]]++;
            hashTable[v[j]]++;
        }
    }
    return false;
}
```

The above solution solves the problem in $O(n^2)$, time complexity (average) and $O(n)$ memory complexity. Although the worst case scenario is $O(n^3)$ and it uses more memory, this solution is way faster in practice for large input cases. To showcase this I made simmulations with the codes shown (that can be found in bibliography), the results are:

ArraySize	Time Without Hash Table	Time with Hash Table	Increase in Performance
128	4.231ms	6.494ms	-53.4%
256	34.223ms	26.665ms	22.0%
512	267.499ms	99.130ms	62.9%
1024	1742.688ms	302.453ms	82.6%
2048	7345.126ms	683.197ms	90.6%
4096	25029.888ms	761.363ms	96.9%

As we can see for the table above, the three sum solution using hash table quickly surpasses the brute force implementation. As we will see later, hash tables (or *unordered_map*) are not the fastest hash tables possible [**?**]. That means we can have an even greater performance than what is shown now.

Chapter 5

Conclusions

Texto texto[1].

[1]Exemplo de referência para página Web: www.vision.ime.usp.br/~jmena/stuff/tese-exemplo

Appendix A

Apendix

Texto texto.