University of São Paulo
Institute of Mathematics and Statistics
Bachelor of Computer Science

Hash Functions and Hash Tables

Breno Helfstein Moura

Final Essay

mac0499 — Capstone Project

Program: Computer Science
Advisor: José Coelho de Pina Junior

São Paulo
December 10th, 2019

Resumo

Breno Helfstein Moura. **Funções e Tables de Hash**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2019.

Este trabalho de conclusão de curso trata de um dos mais fascinantes e úteis conceitos em ciência da computação: funções de hash e tabelas de hash. O texto está organizado em três partes principais:

- Funções de Hash
- Tabelas de Hash
- Aplicações

Funções de hash é uma ideia importante em ciência da computação e vai muito além de seu uso em tabelas de hash. Nesse texto são descritas algumas das ideias que Donald Knuth apresentou em seu livro inspirador, *The Art of Computer programming, Vol. 3* (Knuth, 1973). Estimamos a qualidade de funções de hash através de algumas métricas conhecidas.

Tabelas de hash é uma das mais usadas estruturas de dados em programação. Indicamos os componentes de tabelas de hash em que funções de hash têm um papel primordial. Descrevemos várias das implementações clássicas dessa estrutura; cada uma apropriada para um determinado cenário.

Por fim, descrevemos algumas aplicações de funções e tabelas de hash em problemas recorrentes em ciência da computação. Entre as aplicações está o algoritmo *Rabin-Karp* para busca de padrão em textos que utiliza hashing e um algoritmo eficiente para identificar isomorfismo em árvores utilizando funções de hash.

Espero que esse trabalho seja tão divertido de ler quanto foi para escrever!

Obs: O idioma escolhido para o trabalho foi o inglês devido a muitos termos que se referem a hashing estarem nesse idoma.

Palavras-chave: Hash. Hashing. Function. Maps. Dictionaries. Symbol Tables

Abstract

Breno Helfstein Moura. **Hash Functions and Hash Tables**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2019.

This text deals with one of the most fascinating and useful concepts in Computer Science, which are hash functions and hash tables. It is organized in three main topics:

- Hash functions
- Hash tables
- Applications

Hash functions is a key tool in Computer Science, its applications goes far beyond its use in hash tables. In this text it is presented some of the ideas Donald Knuth described in his inspiring book, *The Art of Computer programming, Vol. 3* (Knuth, 1973), and we apply some metrics in order to estimate the quality of a hash function.

Hash tables is one of the most used data structures in computer programming. We present the constituents parts of a hash table, in which hash functions have a prominent role, and show some of the classic implementations of this data structure; each one particularly useful in a specific scenario.

Finally, we describe some applications of hash functions and hash tables in every day computer science problems. Among the algorithms shown there are *Rabin-Karp*, a string search algorithm that uses hashing and a solution to identify isomorphism on trees using hashing functions.

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

Keywords: Hash. Hashing. Function. Maps. Dictionaries. Symbol Tables

Contents

1 Introduction **1**

2 Hash Functions **5**

2.1 Definition . . . 6
2.2 Division and Multiplicative Methods . . . 7
2.3 Hashing Strings . . . 8
2.4 Quality of Hash Functions . . . 9

3 Hash Tables **15**

3.1 No collision open addressing hash table . . . 17
3.2 Open addressing . . . 18
3.3 Linear Probing . . . 19
3.4 Quadratic Probing . . . 21
3.5 Double Hashing . . . 22
3.6 Robin Hood Hashing . . . 22
3.7 Cuckoo Hashing . . . 25
3.8 Coalesced Hashing . . . 27
3.9 Chaining hashing . . . 29
3.10 Simple Chaining Hashing Algorithm . . . 30
3.11 Move to front . . . 31
3.12 How to delete an entry . . . 32
3.13 When to resize an array . . . 33
3.14 Open Addressing vs Chaining Hashing . . . 33

4 Applications **35**

4.1 3-sum problem . . . 35
4.2 Rabin-Karp . . . 37
4.3 Complete tripartite graph . . . 40
4.4 Hashing trees to check for isomorphism . . . 40

5 Final Remarks 43

References 45

Chapter 1

Introduction

One of the most used data structures in computer science are dictionaries, which are also known as associative array, map or symbol table. Those are a collection or key-value pairs, where all pairs have different keys. The data structure supports the operations of inserting a new pair, finding the value associated to a given key, and deleting a pair. If one thinks about it, this is perhaps one of the most executed tasks in many software systems. For instance, the call history of numbers on a cell phone shows for each phone number, the owner of that number. In a dictionary we can insert a phone number and its owner, so that given a phone number one can retrieve the owner.

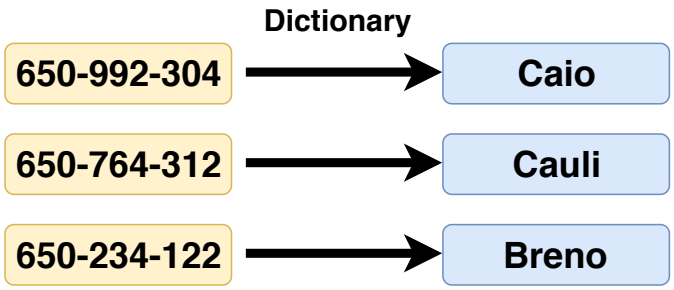

Figure 1.1: *Example of a dictionary that associates phone numbers to contact names.*

Other use of a dictionary that we can think is to count the frequency that a certain number was called. One of the most common and efficient implementations of dictionaries is with a hash table. A key component in the implementation of a hash table is a hash function. This function usually takes the key of the key-value pair we want to insert, find or delete in the table and "digest" it into a number. That number is then used to identify

the value, in this structure that we call hash table. In our example, the keys are phone numbers and the values are contact names.

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

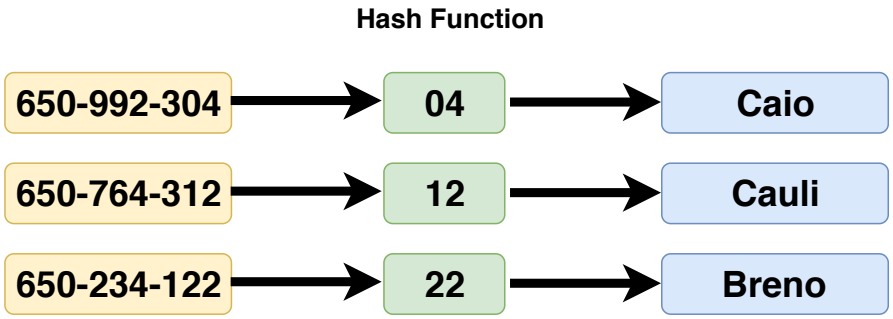

Figure 1.2: *Example of a hash function that just takes the last 2 digits of the phone number*

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, a collision in the table. Handling collisions in a hash table is a complete topic by itself, and it will be addressed in Chapter 3.

Handling collisions is actually a very important topic in hash tables as the vast majority of hash functions will have collisions. To illustrate this situation we recall the "Birthday Paradox", that states 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) (Knuth, 1973), 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. This shows that we will have collisions more often than not, so knowing how to deal with it is a major concern that can not be neglected.

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 the state of the art. While the hash table was widely discussed by many scientists, including 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 (Facebook, 2019).

An example of lack of consensus in this area are the different hash functions and hash tables built-in implementations in different languages. There is no clear consensus on how to decide the size of a hash table, what are the trade-offs 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 is presented throughout this undergraduate thesis.

It's important to notice that during this thesis we will present code snippets of implementations, all of them are in C++14 and can be compiled with the following command:

```
g++ -std=c++14 -Wall -Wshadow -O2 code.cpp -o code
```

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 fixed size (Wikipedia, 2019c).

They have wide applications in computer science, being used in information and data security, compilers, distributed systems and hardcore algorithms. In this chapter we first define and explain the basics of a hash function, then we give an intuition on some metrics that tries to capture the idea of what is expected of a good hash function, as discussed in the famous *"Red Dragon Book"* (Aho *et al.*, 1986) along with some reproduction of known results in the area.

The value extracted from the hash function for an object or key is usually called *hash value* or simply *value*. The hash value is in general, but not necessarily, smaller than the object that generated it (Figure 2.1). For example, we can have a hash function that takes Gigabytes or Terabytes files and returns an 8 bytes hash value.

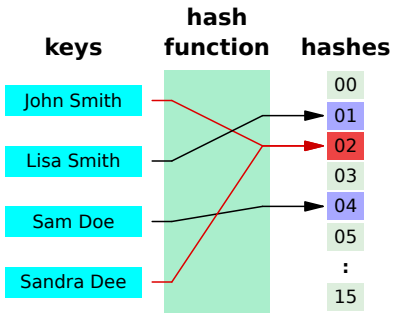

Figure 2.1: *Illustration of a hash function from string to 4 bit integer. Source: Wikipedia, Jorge Stolfi*

2.1 Definition

A hash function over a set X is a function H that takes an element x in X and associates to x an integer $H(x)$ in the interval $[0, M)$, for some integer M. In symbols

$$H : X \longrightarrow [0, M)$$

When dealing with hash tables, X is the set of possible keys and M is the size of the table, which is usually just an array. Moreover, we saw earlier that hash functions are usually more useful when $|X| > M$. This is the same definition used by Donald Knuth (Knuth, 1973) and some articles (Celis, 1986). This definition makes sense for our purpose because we will be talking mostly about hash functions used in hash tables. In that case we want integers that will be indexes in an array (as we will see later on). In other cases we may see hash values as strings, like for when we hash a string for password storage or when we use a hash function in files for checking integrity (for when we are checking if two files are the same). For the goal of this text we will not cover those functions, but it is important to notice that strings can also be easily associated to integers if we just look at their bytes.

For our purpose we are looking for a hash function that performs well for the construction of hash tables. Ideally, these functions should be fast to compute and minimize 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 changes, and for passwords one that is very hard to find its inverse, those are so called cryptographic functions. For some collision resolution techniques, as we will see later, we may also want that the hash function disperse the values well too. Intuitively, a hash function should look like a random function having a given key as seed.

As written in Knuth's book, we know that it is theoretically impossible to create a hash function 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 fewer collision than an uniformly random function. Knuth describes two specific methods for simple hash function, named *division hashing* and *multiplicative hashing* techniques. As the name suggests, the first is based on division operation and the latter on multiplication operation.

2.2 Division and Multiplicative Methods

The *division hashing* method maps an integer X associated to the data to its remainder modulo an integer M. Supposing we can represent the data as a non negative integer X, the division hashing would be to choose a value M and the hashing function would be X mod M. The code would look as following:

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

A good hash function combines number theory, statistics and engineering and in general, large prime numbers tend to be a good value to M, to avoid unwanted patterns or 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).

For *multiplicative hashing*, let's first suppose once more that we can represent the data as a non-negative integer X and we have chosen a constant S, where $0 < S < 1$. Then we multiply X by S and extract the fractional part of $X \times S$, that is $X \times S \mod 1$. We can calculate that by doing: $X \times S - \lfloor X \times S \rfloor$. Then we multiply that value, that will be between 0 and 1, by M. The code of what was described above would look as following:

```
1 unsigned int multiplicativeHashing(unsigned int X,
2                                    double S,
3                                    unsigned int M) {
4   double alpha = (double) X * S - floor((double) X * S);
5   return (unsigned int) floor(alpha * (double) M);
6 }
```

In Knuth's book he describes S as being an integer A divided by w, where w is the size of a "word" in our computer. He restricts A to be relatively prime to w. That definition is often useful if one wants to retrieve a value Y for a given hash value F. This can be done via Bezout theorem (Knuth, 1973). It is good to note here that if $H(X) = F$ and $H(Y) = F$, X is not necessarily equal to Y, as two different keys can have the same hash value.

2.3 Hashing Strings

We have many ways of converting non numerical data to non negative integers. In the end, it is all just a sequence of bytes, that when read in a specific way form another type of data, such as images or strings. For example, one way of transforming a string to a non-negative integer is summing the ASCII value of its characters. The C++14 code for that would look as following:

```
1 unsigned int stringToInteger(string str) {
2   unsigned int hashValue = 0;
3   for (char c : str) {
4     hashValue += (int) c;
5   }
6   return hashValue;
7 }
```

We always use unsigned integers for our non negative integer calculations due to the natural modulo operation of it on overflow cases. It is equivalent to having a $\mod 2^{32}$ every time it overflows, as we only look at the leading 32 bits. We can also use XOR function to mix numbers together.

There is also a very common type of hash functions that tend to work pretty well for strings (Kankowski, 2008). We can also think of a "superset", of generalization, of multiplicative hash functions. The C++14 code would look as following:

```
1 unsigned int hashForString(string str,
2                            unsigned int initialValue,
3                            unsigned int multiplier,
4                            unsigned int modulo) {
5   unsigned int hash = initialValue;
6   for (char c : str) {
7     hash = (multiplier * hash + (int)c);
8   }
9   return hash % modulo;
10 }
```

The above function is very common for string hashing, and by just choosing a different initial value and multiplier we can have completely different hash functions. Although using summing or using XOR to combine the previous hash value with the new character

usually don't provide much difference, with XOR operation we do not need to worry about overflow. Some values are of known hash functions, for example with `multiplier = 33` and `initialValue = 5381` generates *Bernstein hash djb2* (Bernstein, 1991) or `multiplier = 31` and `initialValue = 0` generates *Kernighan and Ritchie's hash* (Kernighan and Ritchie, 1988). Those are famous functions and their values are not chosen randomly, as there are some factors that maximize the chance of producing a good hash function, where good means low collision rate and fast computation. Those factors are:

- The multiplier should be bigger than the size of the alphabet, in our case usually 26 for English words or 256 for ASCII. That is the case because if it is smaller we can have wrong matches easier. For example, suppose that `multiplier = 10` and `initialValue = 0`, we have $H('ABA') = H('AAK') = 7225$ before taking the modulo operation.
- The multiplication by the multiplier should be easy to compute with simple operations, such as bitwise operations and adding. That is quite intuitive as we want a hash function that is fast to calculate.
- The multiplier should be coprime with the modulo. That is because otherwise we will "cycle" hashes at a greater rate than the modulo (We can use some modular arithmetic to prove that). Usually prime numbers tend to be good multipliers.

2.4 Quality of Hash Functions

Now that we know some good templates for producing hash functions let us try to find a concrete metric or formula that measures the quality of a hash function. Fortunately, the famous book Compilers: Principles, Techniques, and Tools, also known as "Red Dragon Book" (Aho *et al.*, 1986), has already proposed a quantity to measure the quality of a hash function. This quantity is given by:

$$\sum_{j=0}^{m-1} \frac{b_j(b_j+1)/2}{(n/2m)(n+2m-1)},$$

where n is the number of keys, m is the number of total slots and b_j is the number of keys in the j-th slot. The intuition for the numerator is that it represents the number of operations we will need to perform to find each element of the table. For example, we need 1 operation to find the first element, 2 to find the second, and so on. That means that we will end up with an arithmetic progression. We know that a hash function that distributes the keys according to an uniformly random distribution has expected bucket size of n/m,

so we can calculate that the expected value of the numerator formula is $(n/2m)(n+2m-1)$. So that gives us a ratio of collisions (thinking just about operations to access a value) of "our" hash function with an "ideal" function. That means that a value close to 1.00 of the above formula is good, and values below 1.00 means that we had less collisions than an uniformly distributed random function.

For common data as the ones in Dragon Book and Strchr website (Kankowski, 2008) some tests with the previously cited hash functions were performed.

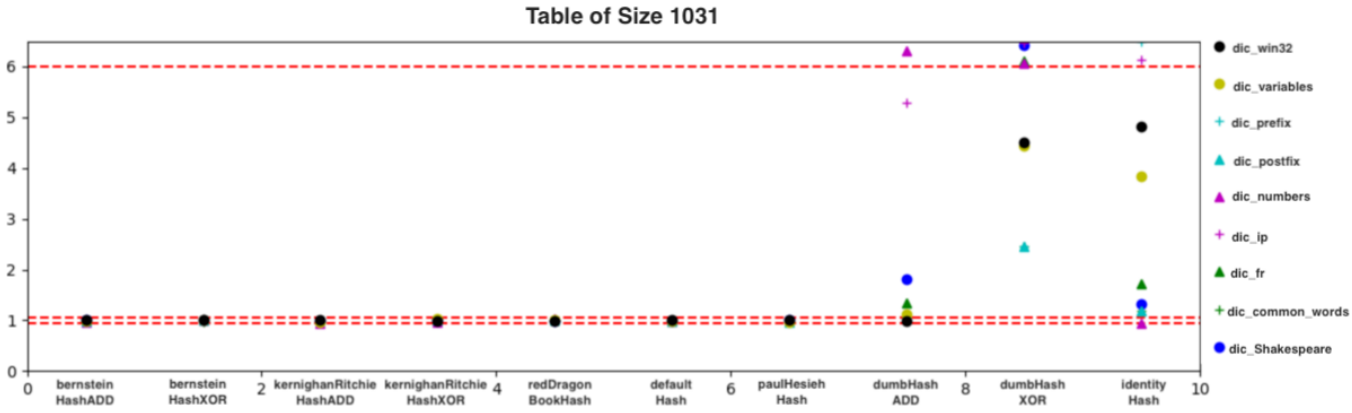

Figure 2.2: *Functions tested against a "small" table*

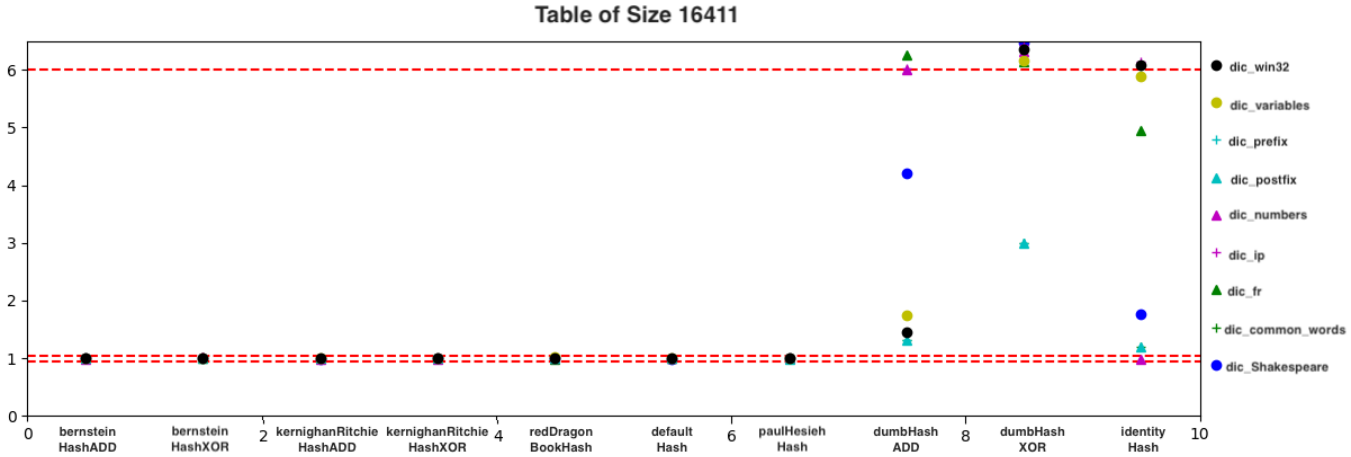

Figure 2.3: *Functions tested against a "medium" sized table*

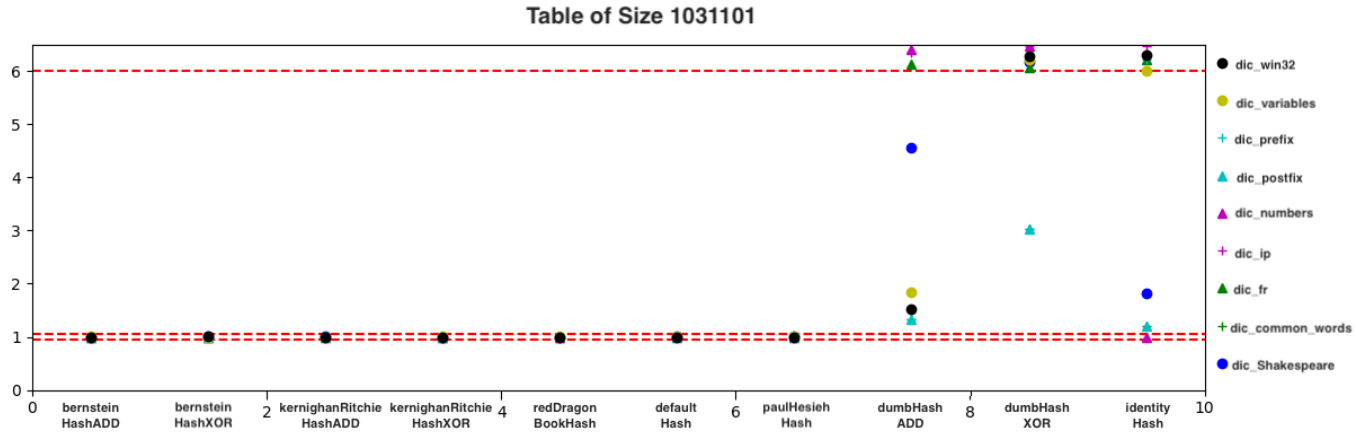

Figure 2.4: *Functions tested against a "large" table*

The results are shown in the same way displayed in the Red Dragon Book, with hash functions in the x axis and the collision ratio displayed in the y axis, with different identification for each file. We consider three sizes of tables to count collisions, a "small", "medium" and a "large" table, the small table having a load factor (The percentage of the table occupied) of approximately ~ 0.5, the medium with ~ 0.05 and large with ~ 0.005. It's assumed that the modulo is not the responsibility of the hash function, so all hash functions return values from 0 to $2^{32} - 1$, and the modulo is taken depending on the size of the table. From that we can already see that the load factor doesn't make a good hash function bad, but expose problems of "bad" hash functions in some cases.

Another fact that it is important to notice from the graph is the red dotted lines. The top one is the "Upper" threshold, which results greater than 6 are just considered "Big", as in some cases the ratio exploded to values up to 200. The lower 2 red lines are in $y = 1.05$ and $y = 0.95$ which is the interval that we consider a hash function to have "Good" values.

The tests were made with 10 different hash functions, tested against 9 different files (which can be found in strchr website (Kankowski, 2008)). All of the code used to test this can be found in the github repo (Moura, 2019). The 10 hash functions are the following:

- **bernsteinHashADD**: The Bernstein hash function described earlier. We use the hash template adding the elements showed earlier. The multiplier is 33 and the initial value is 5381. In the end we XOR the bits of the hash with itself shifted 16 to the right (That is half of the bits with our implementation).
- **bernsteinHashXOR**: The same as above but substituting the first adding operator

by the XOR operation.

- **kernighanRitchieHashADD** The Kernighan and Ritchie Hash function described earlier. We use the given hash template adding the elements. The multiplier is 31 and the initial value is 0.
- **kernighanRitchieHashXOR** The same as above but substituting the first adding operator by the XOR operation.
- **redDragonBookHash** The hash function tested in the Red Dragon Book. It is described as x65599 in the book.
- **defaultHash** The default hash function of C++ standard template library.
- **paulHesiehHash** A fast hash function described by Paul Hesieh (Hesieh, 2004). It is fast to calculate and more complex than Knuth multiplicative or division Hashing.
- **dumbHashADD** A hash function that simply add all characters.
- **dumbHashXOR** A hash function that simply XOR all characters.
- **identityHash** A hash function that takes the first 4 bytes of the string.

We have a variety of hash functions, with all being considered "fast" hash functions. The files tested include common words in English and french, strings of some IP values, numbers, common variable names and words with common prefix and suffix.

First, we can notice from those graphs that changing the ADD function to XOR doesn't make a good multiplicative hash function bad. Both are actually "Equivalent" given that we are also multiplying the values. For `dumbHashADD` and `dumbHashXOR` we can see clear differences, with `dumbHashXOR` being clearly worse. This can be explained by the cancellation property of XOR. We can see this example on the hash of this IP below:

```
dumbHashXOR('168.1.1.0') = dumbHashXOR('168.2.2.0') = dumbHashXOR('124.6.8.0')
```

We can see that many different IPs have the same hash value. More than that, XOR don't increase the number of bits, so all the hashes will be of just 1 byte.

We can also notice that "identity" hash is good or perfect in some cases. One obvious case that "identity" function works perfectly is for numbers as we will have 0 collisions. Some languages, like Python 3, use the identity function to calculate hash for integers, as it is very fast and produces no collisions. But we can see that for other cases, such as common prefix, it works terrible as we just get the first 4 bytes.

The most common multiplicative hash functions tend to work similarly well, being reasonably close to an uniformly random function (that is our "ideal" hash function) in all

cases.

As we can see, we don't need a lot of complexity to make a good hash function for a hash table. We have some functions working better for some specific case, like identity function working well for numbers, but general functions already work well enough.

It is important to note here that hash function is a very vast topic, and here we just covered hash functions related to hash tables. Hash functions have applications in distributed systems (consistent hashing), database indexing, caching, compilers (Red Dragon Book) and cryptography. Each application has different requirements and make some hash functions better than others.

Chapter 3

Hash Tables

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 *value* for it.

It is one of the possible, and many times considered the best, implementations of a dictionary. It has to implement the `insert`, `find` and `remove` operations, that can be accessed from outside the dictionary. It usually implements a lot of other private methods.

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-value pair. That is because hash tables usually have a constant average cost for those operations.

Moreover, when talking about hash tables we have the problem of key collision, that is when two keys map to the same hash value. As we saw in the previous chapter, collisions are more common than not, so collision resolution is a critical problem. To solve that problem, we have several techniques that involve different trade-offs. 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. For this last one we usually consider a load factor, α, that is the ratio of keys with the available slots in the table.

Also, hash tables can be easily abstracted to hash sets, commonly used to store a set of elements and check whether an element is in the set. We can abstract hash sets to a hash table always with an empty value. Hash sets are one of the common ways to implement sets in programming languages, like unordered_set from C++14.

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

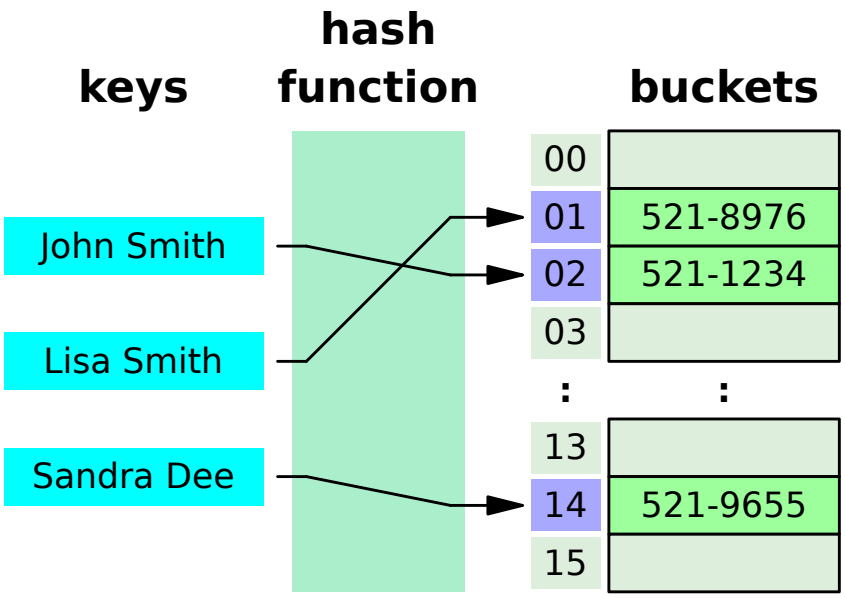

Figure 3.1: *Example of a hash table from string to string, more specifically name to phone number. Source: Wikipedia, Jorge Stolfi*

In the above figure (3.1), we can see an example of a hash function that matches names of people to phone numbers, as we saw in the first chapter. This table has no collisions, and we can see that, for example, “Jhon Smith” has a hash value of 2 and his value in the table is “521-1234”. That is we associated a names with phone numbers.

3.1 No collision open addressing hash table

To start let's give an example of a hash table that has a perfect hash function, that is a function from X to $[0, M)$ with no collisions from the used keys. For that example we use open addressing, that basically means that all data will be contained in an array (that is, the whole table). The operations `insert`, `find` and `remove` would be very easy to implement. For the sake of simplicity, we will assume all the keys are strings and values are integers. To start lets look at this simple class with dummy methods:

```
1 class HashTable {
2    vector< pair<string, int> > table;
3    int m, n;
4
5    HashTable() {
6       m = 16;
7       table.resize(m);
8       n = 0;
9    }
10
11   unsigned int hashFunction(string s) {}
12
13   void insert(string key, int value) {}
14
15   int find(string key) {}
16
17   void remove(string key) {}
18
19 private:
20   double alpha = 1;
21   void resizeIfNecessary() {}
22 }
```

As we can see it is pretty simple. The constructor builds a table of size 16, and we can assume a dynamic resizing every time the table is full. Later on we will see that this means that we resize every time the load factor, α, is equal to 1.00. We also can note that at the table part we are storing a pair of key and value, not just value. This is because we may want to retrieve all pairs of the table (like in a regular dictionary). The pairs are usually unordered (If they are not ordered by chance ...). Actually, if one needs the set of keys sorted by total order, very likely hash tables are not the appropriate data structure. We will skip the implementation of hashFunction, as we already saw plenty of it in the last

chapter, so we will go right in for the implementation of `insert`:

```
1 void insert(string key, int value) {
2   resizeIfNecessary();
3   unsigned int idx = hashFunction(key);
4   table[idx] = pair<string, int>(key, value);
5   n++;
6 }
```

That is pretty simple, that is mostly because we will assume that we will never have a collision, so we just put the key on the position returned by the hash function. The method `find` is implemented as following:

```
1 int find(string key) {
2   unsigned int idx = hashFunction(key);
3   if (table[idx].first == key)
4     return table[idx].second;
5   return 0;
6 }
```

Also very simple, we always know the value will be in position returned by idx. The `remove` will be of the same simplicity, as following:

```
1 void remove(string key) {
2   unsigned int idx = hashFunction(key);
3   table[idx].first = pair<string, int>("", 0);
4   n--;
5 }
```

Here we make the assumption that an empty position has an empty string. We could also carry a boolean, usually called a tombstone, to check if the position is occupied or not. If the hash function is perfect, than the insert, find and delete operations can be performed in constant time and linear space.

3.2 Open addressing

We can define open addressing in a general way as a hash table algorithm where the data always stay within the same vector. So, in the case of a collision, we need to define a

systematic way to traverse the table. The sequence of elements we need to traverse when we have a collision is called *"Probe sequence"*. With that our hash function would change to the following:

$$h(x, i)$$

, where x is our key and i is the probe sequence number. So every time we have a collision in $h(x, i)$ we can simply go to $h(x, i + 1)$. Given that, lets look into some different probe sequences.

At the end of each section there will be a 👍 and a 👎 to indicate a summary of pros and cons.

3.3 Linear Probing

Linear probing is one of the most simple an practical probe sequences known. The probe sequence is basically:

$$h(x, i) = (H(x) + i) \bmod M$$

, where M is the table size. That is a very simple probe sequence with not much secret on it. To implement the `insert` we can do the following:

```
1 void insert(string key, int value) {
2   resizeIfNecessary();
3   unsigned int idx = hashFunction(key);
4   while (table[idx] != pair<string, int>("", 0))
5     idx = (idx + 1) % m;
6   table[idx] = pair<string, int>(key, value);
7   n++;
8 }
```

That assumes that `pair<string, int>("", 0)` is the empty position, and performs a linear search until it finds one empty position to put the new (key, value) pair. The implementation of `find` is very similar:

```
int find(string key) {
  unsigned int idx = hashFunction(key);
  while (table[idx] != pair<string, int>("", 0)) {
    if (table[idx].first == key)
      return table[idx].second;
    idx = (idx + 1) % m;
  }
  return 0; // Default value
}
```

It performs a linear search until it finds the element. If the key is not found the function returns a default value, that in our case is 0.

For removal, we have the problem that we cannot leave "holes" in our table. That will be discussed more in depth later on the section "How to delete an entry". For now we will remove the element, and reinsert the key-value pairs that come right after it:

```
void remove(string key) {
  unsigned int idx = hashFunction(key);
  while (table[idx] != pair<string, int>("", 0)) {
    if (table[idx].first == key)
      break;
      idx = (idx + 1) % m;
  }

  if (table[idx].first == key) {
    table[idx] = pair<string, int>("", 0);
    vector< pair<string, int> > toRehash;
    int j = (idx + 1) % m;
    while (table[j] != pair<string, int>("", 0)) {
      toRehash.push_back(table[j]);
      table[j] = pair<string, int>("", 0);
      j = (j + 1) % m;
    }
    for (auto p : toRehash) {
      insert(p.first, p.second);
    }
    n--;
  }
}
```

With that, the three operations have linear time complexity in the worst case. As we

saw, we know hash functions that are considered good, with a rate of collision very close to an uniformly random function. Those functions will leave us with an expected constant time complexity for those operations, and we will see later on that in practice it is much faster than linear access. Another great benefit that linear probing has, specially when compared to other collision resolution strategies, is locality and cache friendliness.

However linear probing has the problem of clustering, that is long chains of occupied positions. This generates a greater problem, as long chains are only expected to get longer and longer. This can get worse if the hash function is not too sensitive to changes, having a lot of sequential hash values.

- 👍 It is easy to implement and cache friedly.
- 👎 It has problems with clustering.

3.4 Quadratic Probing

Another strategy for resolving collisions is quadratic probing. In this case, the probe sequence can be defined as:

$$h(x, i) = (H(x) + i^2) \mod M.$$

That solves the problem of having sequential hash values. The implementation of quadratic probing is very similar to the implementation of linear probing, with the exception that instead of adding one for each step we can keep the initial value and add the square of a counter.

However, quadratic probing doesn't solve the problem of clustering. Long chains are still expected to get longer and longer, the only difference is that the positions that lead to a longer chain are better distributed in the table. Another thing to notice is that quadratic probing has a worse locality and cache friendliness than linear probing.

- 👍 It is easy to implement and has less problems with clustering than linear probing.
- 👎 It is less cache friendly than linear probing and still has some clustering problems.

3.5 Double Hashing

As we saw the two strategies above have the problem of clustering, due to the fact that the sequences are the same for all keys. For that reason double hashing is a very good approach for open addressing. Double hashing probe sequence can be defined as following:

$$h(x, i) = (H_1(x) + i * H_2(x)) \mod M,$$

where H_1 and H_2 are two distinct hash functions. In that way not only the initial hash value will depend on x, but it's probe sequence offset (that is, the number of slots between $h(x, i)$ and $h(x, i + 1)$) will too. The implementation of double hashing is very similar to both quadratic and linear probing, except that we sum a different offset.

These last three implementations of hash table give us linear time worst case complexity for all operations and constant time expected time complexity under the simple uniform hashing assumption.

- 👍 It is harder to get collisions than linear probing and chaining hashing.
- 👎 It is less cache friendly than linear probing and requires 2 hash functions.

3.6 Robin Hood Hashing

Robin Hood Hashing is an optimization technique regarding collision resolution with open addressing. It should be paired with Linear Probing, Quadratic Probing or Double Hashing, but usually it is paired with linear probing due to it is good locality and cache friendliness. It is basic is that it minimizes the distance of each key from its "home Slot", that is, its initial hash value position. (Celis, 1986). Also, Robin Hood hashing is one of the few open addressing strategies to be built-in in hash tables of some languages, such as Rust.

In order to minimize the distance from each key from its home slot, Robin Hood hashing uses a concept that is called Probe Sequence Length, or PSL, of a key-value pair. The PSL of a key-value pair is the number of probes required to find that pair. For that reason we need to define a new class to store in our table, that we will call `Node`:

```
1 class Node {
2 public:
3    string key;
4    int value;
5    unsigned int PSL;
6    Node(string K = "", int V = 0, unsigned int P = 0):
7       key(K), value(V), PSL(P) {}
8
9    bool operator == (const Node& ot) {
10      return key == ot.key && value == ot.value && PSL == ot.PSL;
11   }
12   bool operator != (const Node& ot) {
13      return key != ot.key || value != ot.value || PSL != ot.PSL;
14   }
15 };
16
17 const Node defaultNode = Node();
```

`Node` is a very simple class that stores a key, a value and an unsigned integer that is the PSL. The main idea around Robin Hood hashing it to move the Nodes with a low PSL in favor of Nodes with a high PSL. We can think of Nodes with a high PSL as poor, because we take longer to find it, and Nodes with a low PSL as rich because we can find them faster. For that reason the algorithm is called Robin Hood Hashing (Celis, 1986).

As explained when inserting an element we first look for an empty position. While searching for it, we check whether the `Node` that is in the way has a lower PSL than the Node that we are inserting, and if that is the case we swap them, securing a position for the current Node and move the other node forward. The implementation of this algorithm using Linear Probing would be the following:

```
1 void insert(string key, int value) {
2   resizeIfNecessary();
3   unsigned int idx = hashFunction(key);
4   Node toInsert = Node(key, value, 0);
5   while (table[idx] != defaultNode) {
6     if (toInsert.PSL > table[idx].PSL)
7       swap(toInsert, table[idx]);
8     idx = (idx + 1) % m;
9     toInsert.PSL++;
10  }
11  table[idx] = toInsert;
12  n++;
13 }
```

For the find method we can use different lookup techniques. Here we will focus on the lookup that is most similar with linear probing, but with a tweak that will make finding that keys are not present faster. While searching for a key, we can calculate what the PSL of that key would be if were inserted, and if we find a Node with a greater PSL that means the pair is not present. That is because all Nodes after it will also have a PSL greater than the current PSL. The implementation of what was described above would be the following:

```
1 int find(string key) {
2   unsigned int idx = hashFunction(key);
3   unsigned int curPSL = 0;
4   while (table[idx] != defaultNode) {
5     if (table[idx].key == key)
6       return table[idx].value;
7     // If the key were inserted it would be before this Node.
8     if (table[idx].PSL > curPSL)
9       break;
10    idx = (idx + 1) % m;
11    curPSL++;
12  }
13  return 0; // Default value
14 }
```

For the removal we can apply *backward shifting*. Although this will be discussed more in depth in the “How to delete an entry” section, this approach is unique to robin hood hashing and has a better performance than rehashing.

Backward shifting consists in first clearing out the slot that contains the key to be removed, then shifting the following keys one step back until a Node with 0 PSL or an empty slot is encountered. The code for that would be the following:

```
1 void remove(string key) {
2   unsigned int idx = hashFunction(key);
3   while (table[idx] != defaultNode) {
4     if (table[idx].key == key)
5       break;
6     idx = (idx + 1) % m;
7   }
8   if (table[idx].key == key) {
9     table[idx] = defaultNode;
10    while (table[(idx + 1) % m] != defaultNode &&
11           table[(idx + 1) % m].PSL != 0) {
12      swap(table[idx], table[(idx + 1) % m]);
13      table[idx].PSL--;
14      idx = (idx + 1) % m;
15    }
16    n--;
17  }
18 }
```

We can always do that because the keys are always sorted according to the their home slot (That is, the first Node with PSL that is 0 that comes before them).

The worst time complexity of all operations is linear and the expected time complexity is constant. The expected length of the longest PSL in a full table is log n.

👍 It is easy to implement and it is cache friendly. Also has a better performance than linear probing.

👎 Uses more memory than linear probing and also more complex to implement

3.7 Cuckoo Hashing

Another well known strategy for collision resolution in open addressing is Cuckoo Hashing. It is a different strategy regarding the previous ones because it uses more than one array, usually two, but up to any number of arrays, to perform collision resolution. It is usually classified as open addressing because each slot can hold up to one key-value pair.

For this explanation let's assume that we are using two arrays. Cuckoo hashing requires also one hash function per array used, in our case two hash functions.

For the insertion of cuckoo hashing we try to insert the key in the first table and if a collision occurs we swap the key value pair that we are trying to insert with the element that is currently on the table and then try to insert it on the next array. If a collision occurs in the other array we swap the pairs and try again on the next one, until we find an empty position or we reach a certain threshold. The threshold is important because we can have cycles.

The code for the algorithm described above is the following:

```
1 void insert(string key, int value) {
2   resizeIfNecessary();
3   unsigned int j = 0, it = 0;
4   unsigned int idx = hashFunction(key, j), lim = maxLoop();
5   pair<string, int> toInsert = pair<string, int>(key, value);
6   while (table[j][idx] != pair<string, int>("", 0) && it < lim) {
7     swap(table[j][idx], toInsert);
8     j = (j + 1) % numTables;
9     idx = hashFunction(toInsert.first, j);
10    it++;
11  }
12  if (it == lim)
13    resize();
14  table[j][idx] = toInsert;
15  n++;
16 }
```

This gives a very strong property to this collision resolution approach, that is every key value pair will be in its corresponding position in exactly one of the arrays. And this will give constant Lookup and Removal time.

In order to find a key value pair we just need to look if the key value pair is present in one of the tables. The code is the following:

```
1 int find(string key) {
2   for (unsigned int j = 0; j < numTables; j++) {
3     unsigned int idx = hashFunction(key, j);
4     if (table[j][idx].first == key)
5       return table[j][idx].second;
6   }
7   return 0;
8 }
```

For removal we can simply erase the key value pair from the table, as no key value pair affect the lookup of any other pair. The code is the following:

```
1 void remove(string key) {
2   for (unsigned int j = 0; j < numTables; j++) {
3     unsigned int idx = hashFunction(key, j);
4     if (table[j][idx].first == key) {
5       table[j][idx] = pair<string, int>("", 0);
6       n--;
7     }
8   }
9 }
```

Besides the amazing property of guaranteed constant lookup and removal, Cuckoo hashing has the problem of cycles during insertion, which can cause unwanted rehashes. To deal with that, many implementations also use a stash to keep a constant amount of elements in case the threshold is reached. A stash is a sort of "bin" of fixed size that we put key-value pairs that failed insertion, and during lookup we would also need to look at the stash.

👍 Has guaranteed constant lookup and deletion

👎 Complex to implement, insertion can be very slow

3.8 Coalesced Hashing

Another well known strategy, described in Donald Knuth book (Knuth, 1973), is Coalesced Hashing. Although without much advantages in contrast with previous strategies, coalesced hashing condenses the hash table well in memory and is very similar to Chaining Hashing, our next topic.

The main idea of coalesced hashing is to add a new parameter to our key value pairs in the table, called next. That would create linked lists in the table in case we have a collision. To find the next element in case a collision we can find the first free bucket looking to the array in reverse order. The function to find the next free bucket is the following:

```
1 int nextFreeBucket() {
2   for (int i = m - 1; i >= 0; i--) {
3     if (table[i].isDefaultNode())
4       return (unsigned int)i;
5     }
6   return -1; // error
7 }
```

To insert an element, in case of a collision, we need to traverse the linked list beginning on the bucket that the key hashes to until the end. Then we add a new Node to the end of the linked list, pointing to the next free bucket. The complete code of the insertion algorithm will be shown later on.

To lookup for an element, we can traverse the linked list until we find a matching Node. The code for the find method would be the following:

```
1 int find(string key) {
2   unsigned int idx = hashFunction(key);
3   while (idx != -1) {
4     if (table[idx].key == key)
5       return table[idx].value;
6     idx = table[idx].next;
7   }
8   return 0;
9 }
```

Removing a node in coalesced hashing is very difficult, as many other nodes can depend on it. For this reason the best way to delete an element in coalesced hashing is by using a strategy that is known as tombstoning. The idea of this strategy is to put a placeholder value, that will be considered as occupied by the find method but as free by the insertion method. The code for that would be the following:

```
1 void remove(string key) {
2   unsigned int idx = hashFunction(key);
3   while (idx != -1) {
4     if (table[idx].key == key)
5       break;
6     idx = table[idx].next;
7   }
8   if (table[idx].key == key) {
9     table[idx].transformTombstone();
10    n--;
11  }
12 }
```

For this reason, the insertion method explained earlier on would have to be a little bit different, considering tombstones. The code would look like the following:

```
1 void insert(string key, int value) {
2   resizeIfNecessary();
3   unsigned int idx = hashFunction(key);
4   Node toInsert = Node(key, value, -1);
5   if (!table[idx].isDefaultNode()) {
6     while (table[idx].next != -1 && !table[idx].isTombstone())
7       idx = table[idx].next;
8     if (!table[idx].isTombstone()) {
9       table[idx].next = nextFreeBucket();
10      idx = table[idx].next;
11    }
12  }
13  table[idx] = toInsert;
14  n++;
15 }
```

- It uses little memory
- Has complex deletion and usually very slow

3.9 Chaining hashing

Chaining hashing, also known as closed addressing, is the implementation of a hash table using a container, usually called bucket, to store the (key, value) pairs with a given

hash. On this implementation, each bucket of the table is a linked list, that will carry the key value pair in our case. We deal with collisions with this implementation by adding a new node to the start of the list.

This implementation is considered simpler than open addressing, usually because the way of dealing with collisions is clearer. Also it is less system dependent if we consider performance (as we saw one of the key advantages of open addressing is that it is cache friendly). That is one of the key reasons that C++ uses chaining hashing for its default implementation of unordered_hash (Austern, 2003a).

Below we will discuss an implementation of chaining hashing and just like in open addressing at the end of each section there will be a 👍 and a 👎 to indicate a summary of pros and cons.

3.10 Simple Chaining Hashing Algorithm

For this chaining hashing implementation we will use C++14 STL data structure list as our container. list is a doubly linked list. For our `insert` we can implement it in the following way:

```
1 void insert(string key, int value) {
2   resizeIfNecessary();
3   unsigned int idx = hashFunction(key);
4   table[idx].emplace_front(key, value);
5   n++;
6 }
```

As we can see it is a very simple implementation, we just push a new element in the front of the list pointed in the idx. As before we add the counter of elements in the list and call resizeIfNcessary().

For `find` we can implement in the following way:

```
1 int find(string key) {
2   unsigned int idx = hashFunction(key);
3   auto it = find_if(table[idx].begin(), table[idx].end(),
4                     [&key](auto& kv) { return kv.first == key; });
5   if (it != table[idx].end())
6     return it->second;
7   return 0;
8 }
```

That implementation is very succinct but uses some of the features of C++14 (such as generic lambdas). For erase we can implement in a very similar fashion:

```
1 void remove(string key) {
2   unsigned int idx = hashFunction(key);
3   auto it = find_if(table[idx].begin(), table[idx].end(),
4                     [&key](auto& kv) { return kv.first == key; });
5   if (it != table[idx].end()) {
6     table[idx].erase(it);
7     n--;
8   }
9 }
```

As we can see, with linked list it is clearly easier to erase an element.

The naive algorithm of chaining hashing with a linked list gives linear worst time complexity for all operations and constant expected time complexity under the assumption of simple uniform hashing.

👍 Very consistent implementation, which makes it a good choice for default and built in hash tables.

👎 Not as fast as open addressing variants, usually due to not being as cache friendly.

3.11 Move to front

One great optimization to chaining hashing is every time you execute the find method to move to the beginning of the container the element that was found. That will keep in the beginning of the container the elements that are searched the most. As in many applications we can apply the 80 / 20 rule this greatly helps in time performance. The 80 /

20 rule is basically the idea that usually, 20% of the keys will represent 80% of the searches, this rule is also cited by Knuth (Knuth, 1973).

If our container is a linked list we can easily adapt the above implementation to move to front every time we search an element, with const time complexity cost. The implementation of `find` would be the following:

```
1 int find(string key) {
2   unsigned int idx = hashFunction(key);
3   auto it = find_if(table[idx].begin(), table[idx].end(),
4                     [&key](auto& kv) { return kv.first == key; });
5   if (it != table[idx].end()) {
6     if (it != table[idx].begin()) {
7       table[idx].splice(table[idx].begin(), table[idx],
8                         it, next(it));
9     }
10    return it->second;
11  }
12  return 0;
13 }
```

Here we are using the `splice` method of list C++ standard library to move an element inside a list. This still keeps the complexity of `find` in linear worst time and constant expected time. It is important to notice here that if our container wasn't a linked list we could take longer than constant time to move it to front.

- Faster than normal chaining hashing if you have some keys more requested by lookup than others.
- Slower than normal chaining hashing if lookups are usually for different keys.

3.12 How to delete an entry

In open addressing deleting an entry is considered hard by many of the collision resolution methods. Between clearing the entry and rehashing, clearing the entry and shifting the elements back or using tombstone, tombstone is usually considered the fastest approach due to its laziness. The problem with tombstones is that it can make the table "dirty" if we have a high number of deletions, making lookups or insertions slower. So one suggestion is to rehash your table in the case of a high number of tombstones.

In contrast, deleting an entry in chaining hashing is delegated to the container that contains the key. That is, if we have a linked list as our container we just delegate the deletion to it. This is much easier is create less problems than open addressing deletion. That is one of the reasons why chaining hashing is usually chosen for default hash table implementation in many languages, like in C++ (Austern, 2003b).

3.13 When to resize an array

In open addressing the load factor to resize a hash table can't be greater than 1.0, because the table can't have more elements than its capacity. That is not true for chaining hashing as we will see later on. A good load factor depends on several factors, such as the strategy used. Some strategies are more "permissive" of a load factor closer to 1, Robin Hood for example can still work well with load factors close to 0.9 and doesn't lose much performance with load factors greater than that (Sylvan, 2013). On the other hand, Cuckoo Hashing doesn't work well with load factors greater than 0.5. Higher load factors means a better use o memory, which is an advantage of Open Addressing, where lower load factors means more memory used but greater efficiency when using the data structure. For that reason we try to always use the greater load factor possible without degrading much performance when using open addressing. In general this value ranges from 0.3 for cuckoo hashing up to .9 for Robin Hood hashing.

In contrast to open addressing, chaining hashing can have max load factors greater than 1.0, although many times those are not used, and when they are used they are not far from 1.0. Default hash tables of C++ and Java use chaining hashing, and the max load factor for a hash table in C++ is 1.0, while for java is 0.75. (C++, 2019). It can be easily proven that the expected time complexity for operations in chaining hashing is $O(1 + \alpha)$ where α is the max load factor. For that reason an big alphas still works reasonably well with chaining hashing. Golang for example has 6.5 as max load factor. Although chaining hashing can still work well with bigger load factors it ends up using more memory and also has a worse locality for cache purposes.

3.14 Open Addressing vs Chaining Hashing

When comparing Open Addressing vs Chaining hashing we can cite many pros and cons. Let us start with the open addressing pros. Among the pros of open addressing we can see that open addressing techniques such as linear probing tend to be more cache

friendly. That is because as the key value pairs are stored in the memory in a sequential way with the vector, when loading a key value pair we will load a chunck of memory that is around it (that will have other key value pairs). Related to it is the 80 / 20 rule, that when applied to hash tables means that "in practice" 80% of the keys will be accessed 20% of the time (and 20% of the keys will be accessed 80% of the time). This is only for illustration purposes, obviously this is not valid for every application, as we can artificially create one that does not follow the rule. Another advantage of open addressing is that all the memory will be in a single and sequential "Block" of memory.

Chapter 4

Applications

Hash functions and hash tables have a great number of applications in computer science. In this chapter we present applications of hash functions in algorithms, and other areas (like cryptography, data deduplication and caching).

We focus in two applications: Rabin-Karp (Wikipedia, 2019e) 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. Hashing of rooted tree for isomorphism checking (rng_58, 2017) is an interesting application sometimes used in competitive programming.

We start by presenting the so called 3-sum problem as a motivation.

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 we present the brute force solution:

```
1 bool threeSumWithoutHashTable(vector<int>& v, int S) {
2    for (int i = 0; i < v.size(); i++)
3       for (int j = i + 1; j < v.size(); j++)
4          for (int k = j + 1; k < v.size(); k++)
5             if (v[i] + v[j] + v[k] == S) return true;
6    return false;
7 }
```

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 doesn'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:

```
1 bool threeSumWithHashTable(vector<int>& v, int S) {
2    unordered_map<int, int> hashTable;
3    for (int i = 0; i < v.size(); i++)
4       hashTable[v[i]]++;
5    for (int i = 0; i < v.size(); i++)
6       for (int j = i + 1; j < v.size(); j++) {
7          hashTable[v[i]]--;
8          hashTable[v[j]]--;
9          if (hashTable.find(S - v[i] - v[j]) != hashTable.end() &&
10             hashTable[S - v[i] - v[j]] > 0) return true;
11         hashTable[v[i]]++;
12         hashTable[v[j]]++;
13      }
14   return false;
15 }
```

The above solution solves the problem in $O(n^2)$ time complexity (average and expected) 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 we did some simulations with different array sizes. The arrays were generated randomly and 100 arrays were generated for each test case, 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 in the table above, the three sum solution using hash table quickly surpasses the brute force implementation. To learn more about how the tests were made, you can check the GithubRepo (Moura, 2019).

4.2 Rabin-Karp

Rabin Karp is a famous pattern matching on string algorithm. Differently than other classic solutions to pattern matching, such as Knuth-Morris-Pratt (Wikipedia, 2019d) algorithm or Boyer Moore (Wikipedia, 2019a), Rabin Karp is based on hashing. It relies on the property that if the hashes of two strings are not equal, they are certainly different strings, and if they are equal, they can be the same string. The definition of the pattern matching problem is the following:

"*Make a function that given two strings, one string t and one string p, it returns the index of the first occurrence of p in t, or -1 if p is not present in t. It is guaranteed that the length of t is greater than the length of p.*"

So given two strings, we need to find the first occurrence of p in t. To first solve this problem, we use the naive, brute force solution:

```
1 int findPatternBruteForce(string t, string p) {
2    for (int i = 0; i <= t.size() - p.size(); i++) {
3       bool match = true;
4       for (int j = 0; j < p.size(); j++)
5          if (t[i + j] != p[j]) {
6             match = false;
7             break;
8          }
9       if (match) return i;
10   }
11   return -1;
12 }
```

We can see that the brute force solution has worst case scenario of $O(nm)$ being $n = |t|$, the size of the string t, and $m = |p|$, the size of the string p. One possible optimization for this solution is if we could check a text interval against the pattern quicker than $O(m)$. If we had the hash of the pattern and the hash of the text interval, we could easily do that. The hash of the pattern is constant, but we have $O(n)$ intervals to check, and given that each interval has $O(m)$ size, if we calculated each of them alone this would take $O(nm)$ again. However, for some hash functions, given the hash of an interval we could calculate the next hash faster. One example of a hash function with this property is the $dumbHashXOR$ hash function presented in chapter 1. Lets test it with intervals in "abracadabra" with intervals of size 4:

$$dumbHashXOR('\texttt{brac}') = dumbHashXOR('\texttt{abra}') \oplus '\texttt{a}' \oplus '\texttt{c}'$$

So given the hash of `'abra'` we could easily move to `'brac'`. Functions with this "shifting" property are called rolling hash functions. As we saw in the hash function chapter, "dumbHashXOR" is, generally, a not so good hash function. Hopefully, we have better rolling hash functions for that, one example is polynomial hashing. The polynomial hashing of a string s with prime P would be:

$$\sum_{i=0}^{m-1} s[i] \times P^i$$

So we know that given hash of $s[0\,..\,m-1]$ we can calculate the hash of $s[1\,..\,m]$ in $O(1)$ in the following way:

$$PolynomialHash(s[1\,..\,m]) = \sum_{i=1}^{m} s[i] \times P^{i-1} = \sum_{i=0}^{m-1} s[i] \times P^i - s[0] + s[m] \times P^{m-1}$$

We would just need to store P^{m-1} for recalculating the hash. So we can check if the pattern is matched on the text quicker with hashing. As just hashing may return a match where we don't have a match, we need to double check to have 100% accuracy. So the algorithm will be:

```
1 const int PRIME = 33;
2 const int MOD = 1000033;
3 int findPatternRabinKarp(string t, string p) {
4    int textHash = 0, patternHash = 0;
5    int pot = 1;
6    // pot will be PRIME^{p.size() - 1}
7    for (int i = 0; i < p.size() - 1; i++)
8       pot = (pot * PRIME) % MOD;
9    for (int i = 0; i < p.size(); i++) {
10      textHash = (textHash * PRIME + t[i]) % MOD;
11      patternHash = (patternHash * PRIME + p[i]) % MOD;
12   }
13   for (int i = 0; i <= t.size() - p.size(); i++) {
14      if (textHash == patternHash) {
15         bool match = true;
16         for (int j = 0; j < p.size(); j++)
17            if (t[i + j] != p[j]) {
18               match = false;
19               break;
20            }
21         if (match) return i;
22      }
23      textHash = (PRIME * (textHash - pot * t[i]) + t[i+p.size()])
           % MOD;
24      if (textHash < 0) textHash += MOD;
25   }
26   return -1;
27 }
```

The expected time complexity of this algorithm is $O(n)$, because the number of string collisions on line 17 on the code above is expected to be low. One interesting fact is that when testing both algorithms shown against each other, for random strings, generated with random lowercase alphabetic characters, the first algorithm is actually faster. That is because in most cases we would exit the brute force early on (we have actually $(1/26)^j$ chance of getting to the next step for each check for an alphabetical random string), making it "expected linear" for this case. And as Rabin Karp has an overhead for calculating the hash, that makes it slower for that case. But that doesn't mean that the algorithm is actually worse, for real text and for random strings where each character is repeated 100 times the algorithm show its strength.

All the code and tests made for this algorithm can be find in github (Moura, 2019).

4.3 Complete tripartite graph

We can also use hashing for graph problems. This is a pretty specific problem but with a very interesting application from the hashing point of view. The problem statement goes as following:

"*Given an undirected graph, decide if the vertices can be partitioned in three groups, such that no two vertices of the same group are connected and every two vertices of different groups are connected .*"

That is, given a graph decide if the graph is a tripartite complete graph.

We can notice that all the vertices from a partition will have the same adjacency list (They will be adjacent all the other vertices). From that we can hash every adjacency list into an integer, and divide the nodes in groups. If we have 3 groups, we have a potential tripartition. Then we can double check if all the vertices in a partition have an adjacency list of size $n - partitionSize$, where n is the number of vertices in the graph and $partitionSize$ is the size of the partition. That will guarantee that they are pointing to all vertices outside the partition.

For the hashing algorithm in the solution of this problem, we can use any of the hashing algorithms described in the first chapter, as we can see an array of integers as a string. If we use a hashing algorithm that is not $dumbXOR$ or $dumbADD$, we need to first sort the array to ensure that all equivalent lists will have the same array.

If we use $dumbXOR$ or $dumbADD$, one thing that can help to minimize collisions is to give each vertex a random value in a greater set. For example if we have $n = 10^5$ for the size of our graph, we can map the nodes to a random value between 0 and $10^9 + 7$. That would increase the size of our potential hashes, minimizing the possible collisions. This problem can be seen in codeforces: https://codeforces.com/contest/1228/problem/D

4.4 Hashing trees to check for isomorphism

For this last algorithmic application of hash functions and hash tables we describe how to decide if two rooted trees are isomorphic. We say that two trees, T_1 and T_2 are isomorphic if there is a bijection ϕ between the set of vertices V_1 and V_2 of the trees, such that:

$$\forall u, v \in V_1 \;\; u \; adj_{T_1} \; v \iff \phi(u) \; adj_{T_2} \; \phi(v)$$

That is, if u is adjacent to v in the first tree, $\phi(u)$ must be adjacent to $\phi(v)$ in the second tree and vice-versa.

Given that definition, we can state the problem of deciding if two trees are isomorphic:

"*Write a function that given two rooted trees, decide if they are isomorphic.*"

We could solve this problem using hash functions if we knew how to hash trees to integers. As we saw in the first chapter, everything is bits in the end and we just need a smart way of representing our data. We can represent a rooted tree as a node that points to its child nodes and so on. So we could define a hash function that always collide when we have isomorphic trees. The following hash function was described by competitive programmer *rng_58* in his blog (RNG_58, 2017).

$$Hash(\mathbf{N}) = \begin{cases} x_0 \text{ if N is a leaf (has no childs)} \\ \Pi_{i=1}^{k}(x_d + Hash(C_i)) \; modM \text{ where } C_i \text{ is a child node, and d is the height of N} \end{cases}$$

For that function we need an array x of size at least the maximum height between both trees. That function is actually a polynomial value of our tree. We can visualize that on the following image:

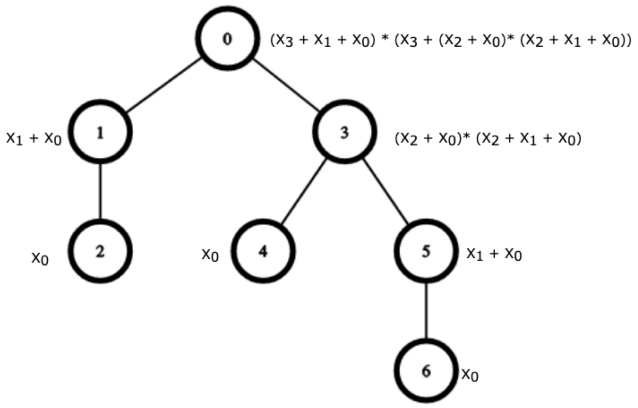

Figure 4.1: *Example of a tree with the hash of each node.*

On the above picture we can see that the hash function is cumulative, and it forms a polynomial function in the end. The code for the above algorithm can be written as following in C++14:

```
vector< vector<int> > adj; // Adjacency list of the graph
vector<long long> x; // X values for hash calculation
vector<int> d; // Height of each node
const int MOD = 1e9 + 7;
long long hashTree(int u, int p) {
   long long myHash = 1;
   for (int v : adj[u])
      if (v != p)
         myHash = (myHash * (d[h[u]] + hashTree(v, u))) % MOD;
   return myHash;
}
```

The hash of a tree can be defined recursively by the hashes of its subtrees. We can define the values of the array X as random values between 0 and `MOD - 1`. We can also trivially calculate the height of each node recursively. The algorithm described is linear.

One interesting caveat of this problem is that, although hashing trees to check for isomorphism only works with rooted trees, we can make it work for the problem of checking tree isomorphism for arbitrary trees. The basic idea is that we can root the tree by the center or centroid of tree (Because we know that we only have at most 2 centers or centroids on a tree). If we have more than one center or centroid, we can calculate the hash of the tree rooted by both and check if at least one of the hashed match. As this is not on the scope of this text I will limit the explanation here, but one can learn more about it in the bibliography (Carpanese, 2018).

It is also important to cite here that there is another algorithm, the Aho-Hopcroft-Ulman algorithm, that runs in worst case linear time complexity that also solves the tree isomorphism problem. This algorithm is based on comparing both trees in a bottom up fashion A. V. Aho and Ullman, 1974.

SPOJ Problem: https://www.spoj.com/problems/TREEISO

Chapter 5

Final Remarks

As we can see by previous chapters, hash functions and hash tables are very wide topics, with several details that we can have pages and pages with explanations. This text aimed to provide a glance on how to implement a hash table data structure, with some other applications of the hash function itself.

It is important to notice that almost every programming language has it's implementation of a hash table, with some having a built-in implementation of a hash function for external usage. The most famous that we can cite here is Java, C++ (which have an implementation on STL), Golang, Python, Ruby, C# and Scala. Its implementation may differ among paradigms as well, for example although in Scala Mutable hash maps uses chaining hashing, it also uses a Hash Trie for immutable hash maps, which is a complete different implementation with it's own specificalities and benefits for functional programming languages. (Cohen, 2017)

Another interesting fact about hash tables is that it is an example about how memory locality is important in modern data structures. Memory locality is the proximity of the data accessed, which means that the data of your data structure is close to each other and it is "cache friendly". Although not accounted by usual complexity analysis, it is an important factor for regular used data structures, as one of the key performance factors of modern day processors.

During this text I also realized how deep we can go in each hash related topic. For that reason there are many contents that are not included here but are interesting to learn about. The main topics that would be included if I had more time were:

- Consistent Hashing
- Distributed Hash Table

- A deep analysis of the hash tables implemented during this text

Consistent hashing is a very interesting topic, because it is a special kind of hash function commonly used to implement sharded databases. It is special because when the table is resized, only $\frac{K}{n}$ keys need to be remapped on average, where K is the number of keys and N is the number of slots.(Wikipedia, 2019b) This is very interesting and very useful in the context of databases because it easily allows horizontal scaling (that is, adding more machines).

Another interesting topic that I would have liked to discuss is distributed hash tables. A very used application in modern day, distributed hash table is a service that simulated a hash table lookup in a distributed environment. Common examples of this is modern services such as Memcached and Redis, that allow fast responses and scaling in modern web applications.

Lastly, it would be interesting to do a deep analysis of the hash tables implemented during this text. This would be different because we would see in practice many of the trade-offs discussed during this text, such as memory locality. There are many different factors in a deep analysis like this, like testing with different load factors, different deleting strategies, and different queries (random queries or queries closer to the 80/20 rule).

References

[Aho *et al.* 1986] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, Inc, 1986 (cit. on pp. 5, 9).

[A. V. Aho and Ullman 1974] J. E. Hopcroft A. V. Aho and J. D. Ullman. "The Design and Analysis of Computer Algorithms". In: *Addison-Wesley* (1974) (cit. on p. 42).

[Austern 2003a] Matthew Austern. *A Proposal to Add Hash Tables to the Standard Library (revision 4)*. 2003. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1456.html (cit. on p. 30).

[Austern 2003b] Matthew Austern. *A Proposal to Add Hash Tables to the Standard Library (revision 4)*. 2003. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1456.html (cit. on p. 33).

[Bernstein 1991] Bernstein. *djb2*. 1991. URL: http://www.cse.yorku.ca/~oz/hash.html (cit. on p. 9).

[C++ 2019] C++. *C++ Reference max_load_factor*. 2019. URL: http://www.cplusplus.com/reference/unordered_set/unordered_set/max_load_factor/ (cit. on p. 33).

[Carpanese 2018] Igor Carpanese. *An illustrated introduction to centroid decomposition*. 2018. URL: https://medium.com/carpanese/an-illustrated-introduction-to-centroid-decomposition-8c1989d53308 (cit. on p. 42).

[Celis 1986] Pedro Celis. "Robin Hood Hashing". In: *Waterloo PhD Research* (1986). URL: https://cs.uwaterloo.ca/research/tr/1986/CS-86-14.pdf (cit. on pp. 6, 22, 23).

[Cohen 2017] Russel Cohen. *An Analysis of Hash Map Implementations in Popular Languages*. 2017. URL: https://rcoh.me/posts/hash-map-analysis/ (cit. on p. 43).

[Facebook 2019] Facebook. "F14 is Open Sourced". In: *Facebook Blog* (2019). URL: https://engineering.fb.com/developer-tools/f14/ (cit. on p. 3).

[Hesieh 2004] Paul Hesieh. *Hash functions*. 2004. URL: http://www.azillionmonkeys.com/qed/hash.html (cit. on p. 12).

[Kankowski 2008] Peter Kankowski. *Hash functions: An empirical comparison*. 2008. URL: https://www.strchr.com/hash_functions (cit. on pp. 8, 10, 11).

[Knuth 1973] Donald Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1973 (cit. on pp. i, iii, 2, 6, 7, 27, 32).

[Kernighan and Ritchie 1988] Brian W Kernighan and Dennis M. Ritchie. *The C programming language, second edition*. Prentice Hall Software Series, 1988 (cit. on p. 9).

[Moura 2019] Breno Helfstein Moura. *Code for Undergraduate thesis*. 2019. URL: https://github.com/breno-helf/TCC/tree/master/monografia/code (cit. on pp. 11, 37, 39).

[rng_58 2017] rng_58. *Hashing and Probability of Collision*. 2017. URL: http://rng-58.blogspot.com/2017/02/hashing-and-probability-of-collision.html (cit. on pp. 35, 41).

[Sylvan 2013] Sebastian Sylvan. *Robin Hood Hashing should be your default Hash Table implementation*. 2013. URL: https://www.sebastiansylvan.com/post/robin-hood-hashing-should-be-your-default-hash-table-implementation/ (cit. on p. 33).

[Wikipedia 2019a] Wikipedia. *Boyer-Moore string-search Algorithm*. 2019. URL: https://en.wikipedia.org/wiki/Boyer-Moore_string-search_algorithm (cit. on p. 37).

[Wikipedia 2019b] Wikipedia. *Consistent Hashing*. 2019. URL: https://en.wikipedia.org/wiki/Consistent_hashing (cit. on p. 44).

[Wikipedia 2019c] Wikipedia. *Hash Function*. 2019. URL: https://en.wikipedia.org/wiki/Hash_function (cit. on p. 5).

[Wikipedia 2019d] Wikipedia. *Knuth-Morris-Pratt Algorithm*. 2019. URL: https://en.wikipedia.org/wiki/Knuth-Morris-Pratt_algorithm (cit. on p. 37).

[Wikipedia 2019e] Wikipedia. *Rabin Karp*. 2019. URL: https://en.wikipedia.org/wiki/Rabin-Karp_algorithm (cit. on p. 35).