Algorithms and Data Structures (ADS)

Written Report - Implementations of Hashing

Bioinformatics Degree, course 2020-2021
By Alain Arquerons & Núria Aleixandre

Index

Introduction: Overview, background knowledge. What is Hashing?

When initiating this investigation, the first question that arised was: what is hashing and what does it do? From the lecture slides we initially discover the act of hashing is an algorithmic procedure that involves computing a short string of integers and letters out of the bits of information that make up an object.

The conversion of the bits of information to the final output string, called a hash value, is done by using arithmetic-based functions. These are named hash functions, and is where the data bits are split up into blocks of a certain size. These blocks of information will then be reinterpreted and combined into the final string of text.

However, from the slides we also learn that not all data structures (objects) are hashable. Mutable objects such as python dictionaries and C++ maps, are not hashable because they can be subject to modifications after being created.

These types of objects or data structures aren't hashable because hashing makes usage of hash tables.
Hash tables are a data structure that maps keys to values. A hash table is what will be used together with hash functions to make our hash value. Hash functions will make an index for the input key in the hash table, index that will have a value given to it and stored in the hash table.
Later on, when you input the key, the hash function will search for the index corresponding to it and retrieve the value in said index.

Hence, the reason for mutable objects not being hashable. Dictionaries and similars, like other objects given to a hash function, must use a hash key for lookup in the hash table. If you were to change the key, which would be the dictionary itself, the value retrieval from the table would not work as intended.
This is similar to trying to retrieve the value entry(ies) for a C++ map or python dictionary for which the key you are looking for doesn not exist.

If you wanted to establish a map **m** in the manner you would normally do in C++:

map <**key**, **info**> m;

Because you will be updating and modifying **info**, as well as removing and adding **key**s, you will modify the instance of the map **m**. This is the reason why mutable objects such as maps are not hashable but immutable such as strings are; the instance of the C++ maps and python lists and sets does not remain the same as you add or remove entries from it, while this is does not hold for objects like integers and strings.

It is true that they can be modified, say, by doing things like these:

string = string + ‘roger’
int = int + 3

However, unlike in mutable objects, their instance has not been modified. The string “string” is still “string” and the integer “int” is still “int”.

To summarize what has been said up to this point: Hashing implies the use of a hash function that will use a hash table, and the process will give us an output of a fixed size depending on the algorithm/hash function used regardless of the size of the input, out of the bits that make up the internal representation of an object. This is why the act of hashing is thought of as a form of data structuring.
Even though there are many formulas that can be used to hash, there are a given set of properties these formulas will always have in common and satisfy:

- Each output will be unique. It should be impossible to obtain the same output using a different input, and it is in turn not possible to obtain a different output from a certain input
- Functions are be fast
- It should be nigh impossible to determine the input based on the output. This is also due to the fact that hash algorithms are one way only. Because they are non-reversible, you cannot obtain the input from an output. In this way, hashing algorithms guarantee the integrity and security of data, as it is very hard to obtain the initial state of the output. This will be discussed later

As said about hash functions, a small change in the input will produce a very different output. This would also be the case were we to use a different hash function/formula.
Generally, hashing formulas differ from one another in the way by which they resolve a Collision. A collision in hashing refers to the situation where different instances of data yield the same hash value when the hash index of the record that is to be inserted is already occupied.
A way in which this is dealt with is by using Linked hashing. Linked hashing behaves in such a way that each hash value is associated to a linked list of those values, and all of their keys map to a certain index in the hash table. This will be expanded upon when we talk about the relevance of hashing when it comes to cryptocurrency networks.

We also took our own approach on avoiding collisions in which, assuming the key to be numerical, by using the key and the table’s size, we are able to ensure that the hash is always within the limits of the table size as well as making sure that every entry key has its own unique value.

We made said hash function using C++, and the code for it is as demonstrated:

key.cc

```cpp
#include <iostream>
#include <string>
#include <list>
using namespace std;
int hashkey(int key, int size) {
    return (key % size);
}

int main () {
    list <int> list1;
    for (int i=0; i<10; i++) list1.push_back(i); //we make a list of size 10
    int key = 35;
    int size = list1.size();
    int index = hashkey(key, size); //fits the key into the list size
    cout << ("The index for the key " + to_string(key) + " is " + to_string(index)) << endl;
}
```

Even though we used a list size of 10 and a key 35, we could use any list size and any numerical key we wanted to and this approach would work just the same. When running with our tested values, as expected, we obtain the index value 5.

```
~/Desktop/Algorithms and data structures/report+presentation hashing$ ./key.x
The index for the key 35 is 5
```

Now, believing we have laid out the basis of hashing and the procedures it follows, we can go on to the main task at hand: finding out about ways in which hashing is used and implemented.

Implementations of Hashing: uses for the Algorithmic Procedure

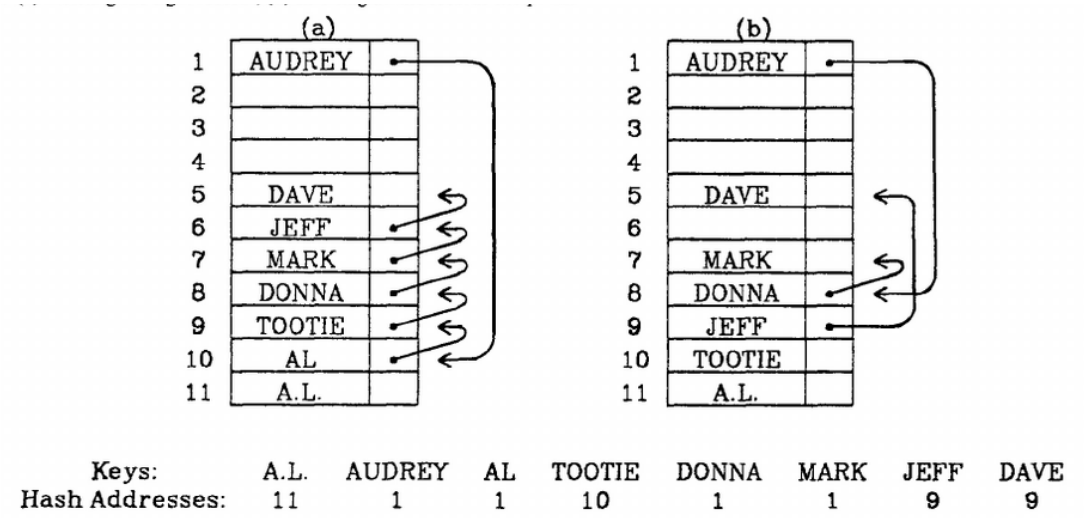
Coalesced hashing.

Since we just discussed collisions and how these want to be avoided, we have to mention Coalesced hashing.
Coalesced hashing is an interesting hashing procedure due to its particular way of collision solving that is given mostly by its structural design. The storage of a hash table produced by coalesced hashing is divided in two groups, the address region and the cellar region.
The address region corresponds to the range of the function as well as the storage for the records after the hash index is made. Meanwhile, the cellar region is a storage region created for solving collisions.
When a collision happens, the record that collides is stored in the cellar. When the cellar is full, the new colliding records are stored in empty slots in the address region.

So, how does coalescent hashing go about its insertions and deletions of records? It does so in the form of various algorithms that modify the records as needed.

Firstly we have deletion algorithms, which are used to get rid of records that don't belong to the set of objects the hash table is representing anymore. For example, orders in a takeaway service after the takeaway orders are delivered.

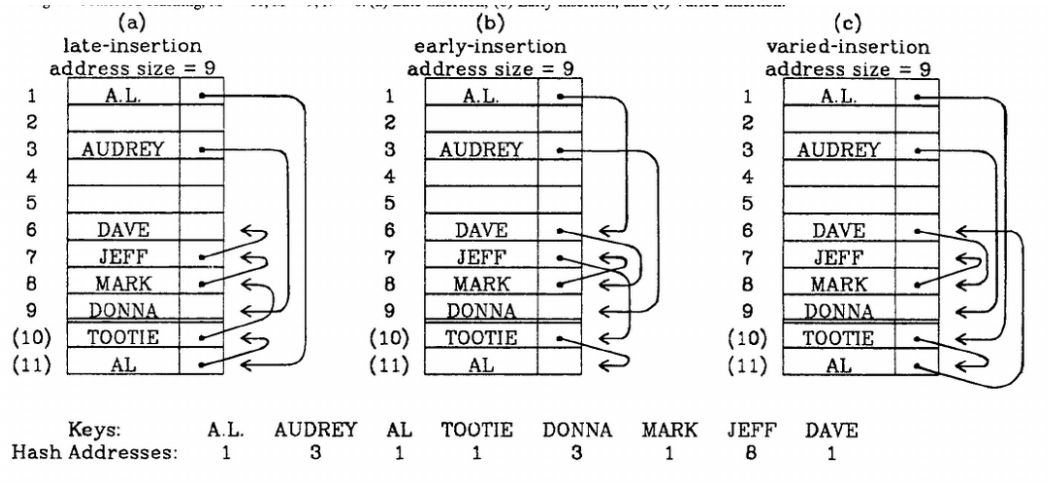
Coalesced hashing also keeps information about the state of a record, in this case, if the record has been deleted. The deleted table slot is now treated as if it were occupied by a null record.
This saves time, as not using this item means that every time that there is a deletion we do not need to move all records that follow the chain.
Rather than deleting the item, it is changed by its predecessor and its predecessor by its predecessor, this keeps on going until the last record, which is stored in the previous record and then deleted.



When deleting AL we create a hole in position 10, and TOOTIE goes to occupy the position, since it doesn't collide with anyone. However, when it comes to DONNA and MARK both of them collide with AUDREY due to having the same hash address so they remain in their slots. When it comes to JEFF and DAVE none of them collide, so JEFF goes to position 9 and DAVE will join the JEFF chain.

Having discussed how deletions work, we now move onto the insertion methods coalesced hashing presents.
When inserting an item, it will be inserted into its hash address and linked into the chain immediately after the item occupies its address. That's why it is called the early-insertion coalesced hashing variant. It is dubbed so because the item is linked "early" in the chain, rather than at the end, which causes the order of the records stored in the cellar to go from newest to oldest.
As there is early-insertion, we also have late-insertion. This methodology is the exact opposite of early-insertion, it keeps the order of the record chain from oldest to newest. This method is, however, rarely used in practice.

The other commonly used insertion method is varied-insertion. This approach behaves similarly to early-insertion, with the difference that if we have a full cellar, the colliding item is linked to the chain immediately after the last cellar slot in the chain.
A comparison of all the insertion methods:

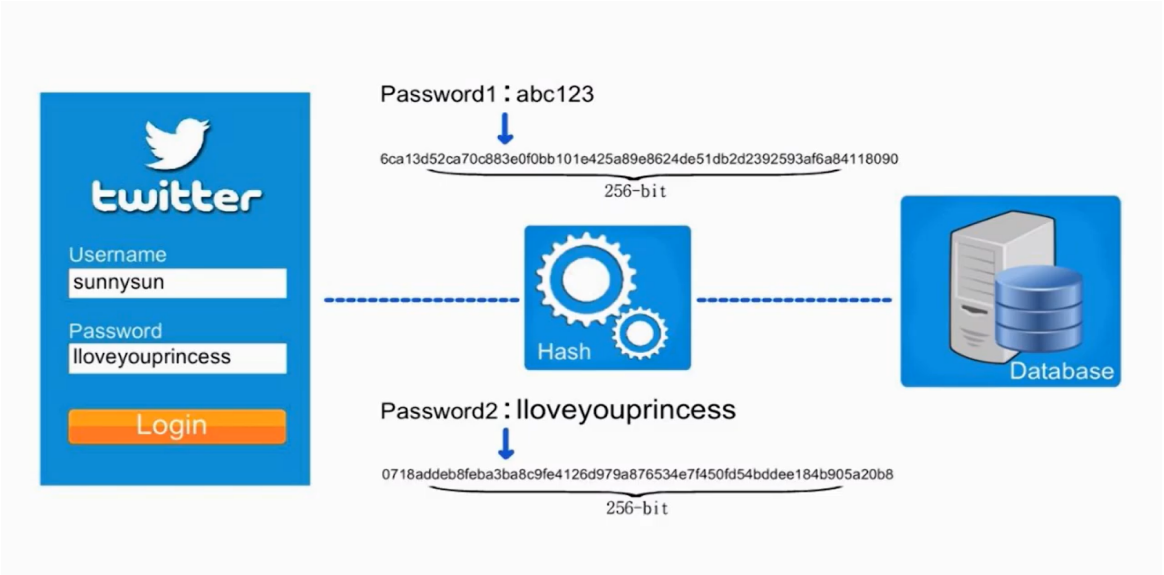


Usernames and Passwords: Database security & traceback.

Hashing is frequently used for data encryption: an area in which hashing algorithms excel is user account security.
Say, for instance, someone creates an account somewhere and logs into it.
The user inputs a string of text for a username and a password when making an account on a site. The hashing algorithm will receive the text and, making usage of its own algorithm and table, output a hash value.

Instead of storing the password itself, what is stored from passwords is their hash output. This is to avoid information theft if the database got breached, as the initial state of the passwords for their users will be very hard to obtain.

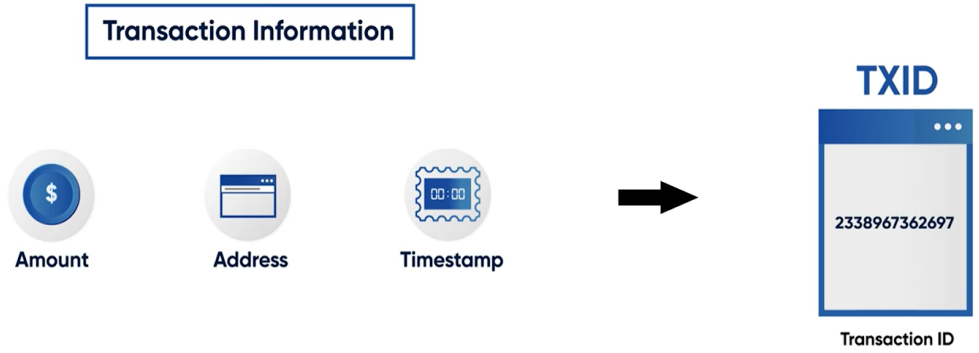
So, instead of checking your actual password the next time you log in, what the database will do is compare the hash value returned from the algorithm using the password you input with the hash value it has stored.
This is possible due to the workflow we mentioned earlier, in which a given input will always produce the same hash output. Different hashing algorithms that are popular when it comes to password protection include; MD5, SHA-1 and SHA-2.



As you can see, no matter the length of the password given, the hash output is always of 256 bits in size.

Money transfers.

Hashing algorithmic procedures also play a role in any kind of money movement or transfer. The algorithm takes into account the amount of money being transfered, the addresses of both the sender and the receiver, and the timestamp of when exactly the transaction happened. All of these different values are processed and computed into the hash value, which is what is commonly known as the Transaction ID. The hash output transaction ID can then be identified and used to confirm that the money movement successfully took place.

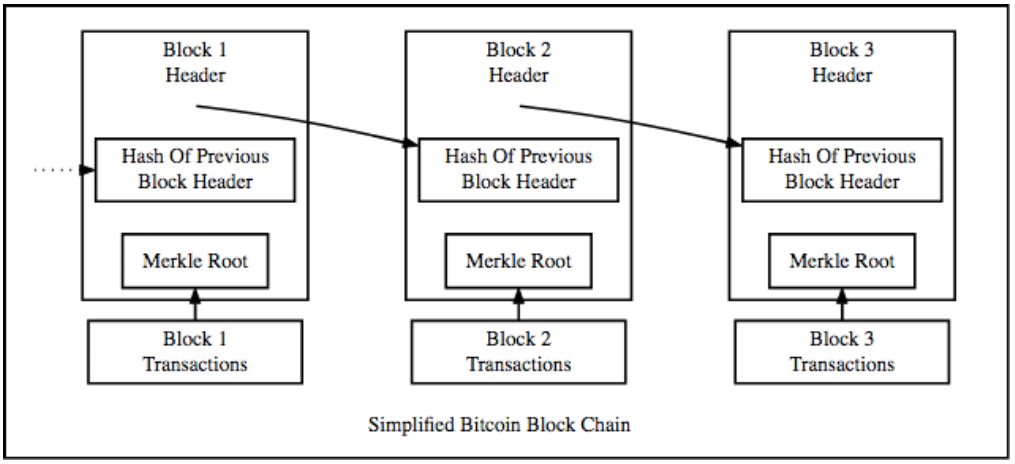


Cryptocurrency.

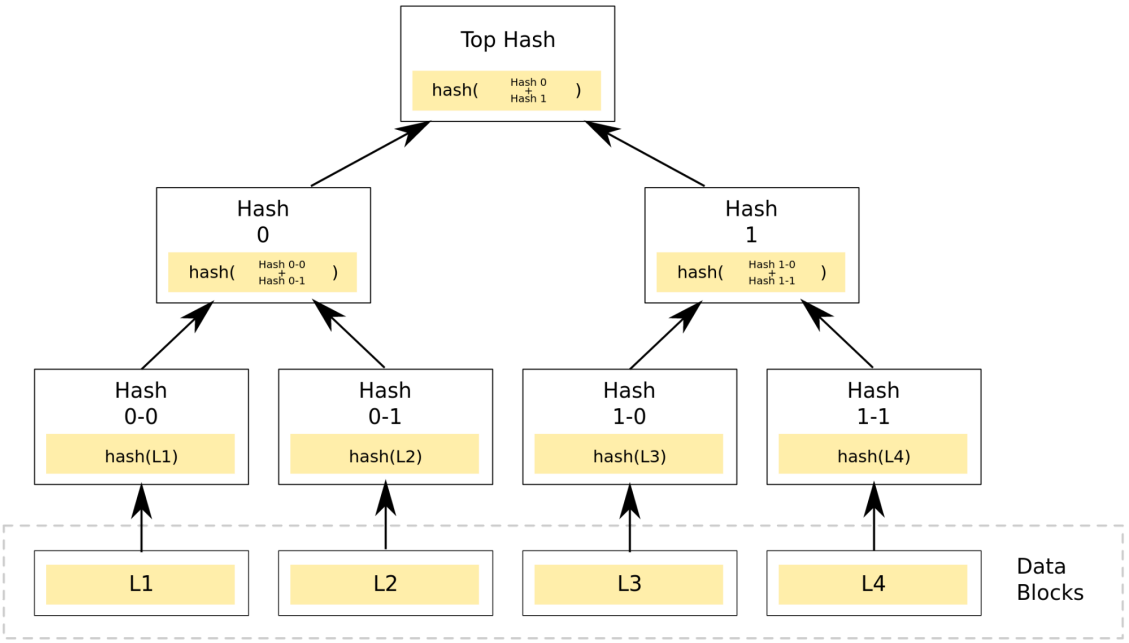
Earlier we mentioned the importance of hashing regarding cryptocurrency-based networks.
In bitcoin for example, the SHA-2 algorithm that is used for password protection is also used here.
Cryptocurrency relates back to our previous point, as what is output in bitcoin networks, for example, is the transaction ID. These are in turn taken into account as one of the inputs so as to make the next transaction ID and are themselves run through a hashing algorithm which will give a new hash value.

Like in user account databases, these hashes are what is remembered by the system. However cryptocurrency networks differ from what we previously saw in that they make use of a blockchain.

What a blockchain is essentially is a linked list. A linked list is a type of data structure that contains data and a hash pointer that points to its previous block and this is what makes the chain. The difference between a hash pointer and a regular pointer we are used to seeing is that instead of just containing the address of the previous block it also contains the hash of the data from that block.



If a change happens somewhere in the blockchain, the rest of the network notices this and handles it accordingly.
All parts of the network must agree to make changes for them to be made in the network, such as the removal, addition or modification of data.
The bitcoin network makes use of the blockchain in order to keep track of new movements. Every new hash block is made by using the previous hash block in the blockchain, as well as a merkle root hash. A merkle tree is a tree such that the root is made of two hashes, and those hashes are made of two other hashes and so on until you reach the leaves of the tree which is unhashed data, like this:



These are used because every block contains many many movements so in terms of time it’s nefficient to store all of the data inside each block. By using a merkle tree you cut down the time required to find out if a particular transaction belongs to that block or not. As an example, take this tree. If you want to know if the data belongs to a given block, you can just follow the trail of hashes.

And that is how bitcoin “miners" operate. They use how bitcoin networks work to search for a new block that contains a bitcoin. Miners use some prior knowledge such as the fact that the hash that bitcoin hashes have to be of a certain difficulty, so that you know, bitcoin hashes currently have 18 zeros at the beginning, then followed by more output. So, whenever a new block arrives, it is hashed together with an arbitrary string to see its number of zeros. If the hash has less than the amount wanted it is added to the blockchain. If it does not, the string is changed for another one and the process is repeated.

Miners aim to solve this input-output problem by slowly chipping away and obtaining the hash value they want, obtaining a bitcoin when they get their desired output.

Here we show a simplistic idea of this, as seen in the following code, where our hash block is “give me bitcoin” and our arbitrary string hashed with it are exclamation marks. Because we do not get a string with a leading zero, we keep changing our string (by adding exclamation marks) until we solve the block and get our desired outpt.

```
~/Desktop/Algorithms and data structures/report+presentation hashing$ ./mining.x
```

```
give me bitcoin!!
66925f1da83c54354da73d81e013974d
give me bitcoin!!!
c8de96b4cf781a6373766c668ceac0f0
give me bitcoin!!!!
9ea367cea6a2cc4a6f5a1d9a334d0d9e
give me bitcoin!!!!!
b8d43387d98f035e2f0ac49740a5af38
give me bitcoin!!!!!!
0fe46518541f4739613b9ce29ecea6b6 -> SOLVED
```

Wrap-up: Thoughts.

All in all, these findings have been interesting to see and explore. Whether you're interested in how exactly people get rich by mining bitcoin to do so yourself or you're simply curious to know about database security structure and how exactly you are identified on your accounts, the answer to that is to understand the various concepts that hashing revolves around.

Finding out these things and seeing how hashing and them are tied together was interesting and dynamic to explore and learn about, as many of these are things we observe and interact with every other day and we never really stop to give them any thought.
By doing so, we learnt some more about hashing and in turn about the world we live in, and that which surrounds us.

Resources. Sources of information.

Implementations for Coalesced Hashing; Vitter, Jeffrey, December 1982

What Is Hashing?; Rosic, Ameer

Bitcoin Hash Functions Explained; Faife, Corin, May 2017

What is Hashing? Hash Functions Explained Simply; Lisk Academy, August 2018

How do hash functions work?; Sunny Classroom, June 2017

How to Hash Passwords: One-Way Road to Enhanced Security; Arias, Dan, September 2019