HASHING

- WHY HASHING IS NEEDED?
- HASH FUNCTION
- HASH TABLE
- HASH COLLISION
- GOOD HASH FUNCTIONS
- STRING HASH
- CALCULATION OF THE HASH OF A STRING
- FAST HASH CALCULATION OF SUBSTRINGS
- PRACTICE: STRING HASH COMPUTATION

Author: prof. Yevhenii Borodavka

WHY HASHING IS NEEDED?

After storing a large amount of data, we need to perform various operations on these data. Lookups are inevitable for the datasets. Linear search and binary search perform lookups/search with time complexity of **O**(n) and **O**(log n) respectively. As the size of the dataset increases, these complexities also become significantly high which is not acceptable.

Hashing is a technique of mapping a large set of arbitrary data to tabular indexes using a hash function. It is a method for representing dictionaries for large datasets.

It allows lookups, updating and retrieval operation to occur in a constant time i.e. **O**(1).

HASH FUNCTION

A hash function is any function that can be used to map data of arbitrary size to fixed-size values, though there are some hash functions that support variable length output. The values returned by a hash function are called hash values, hash codes, hash digests, digests, or simply hashes. The values are usually used to index a fixed-size table called a hash table. Use of a hash function to index a hash table is called hashing or scatter storage addressing.

A hash function takes a key as an input, which is associated with a datum or record and used to identify it to the data storage and retrieval application. The keys may be fixed length, like an integer, or variable length, like a name. In some cases, the key is the datum itself. The output is a hash code used to index a hash table holding the data or records, or pointers to them.

HASH TABLE

The Hash table data structure stores elements in key-value pairs where

- Key — unique integer that is used for indexing the values
- Value — data that are associated with keys.

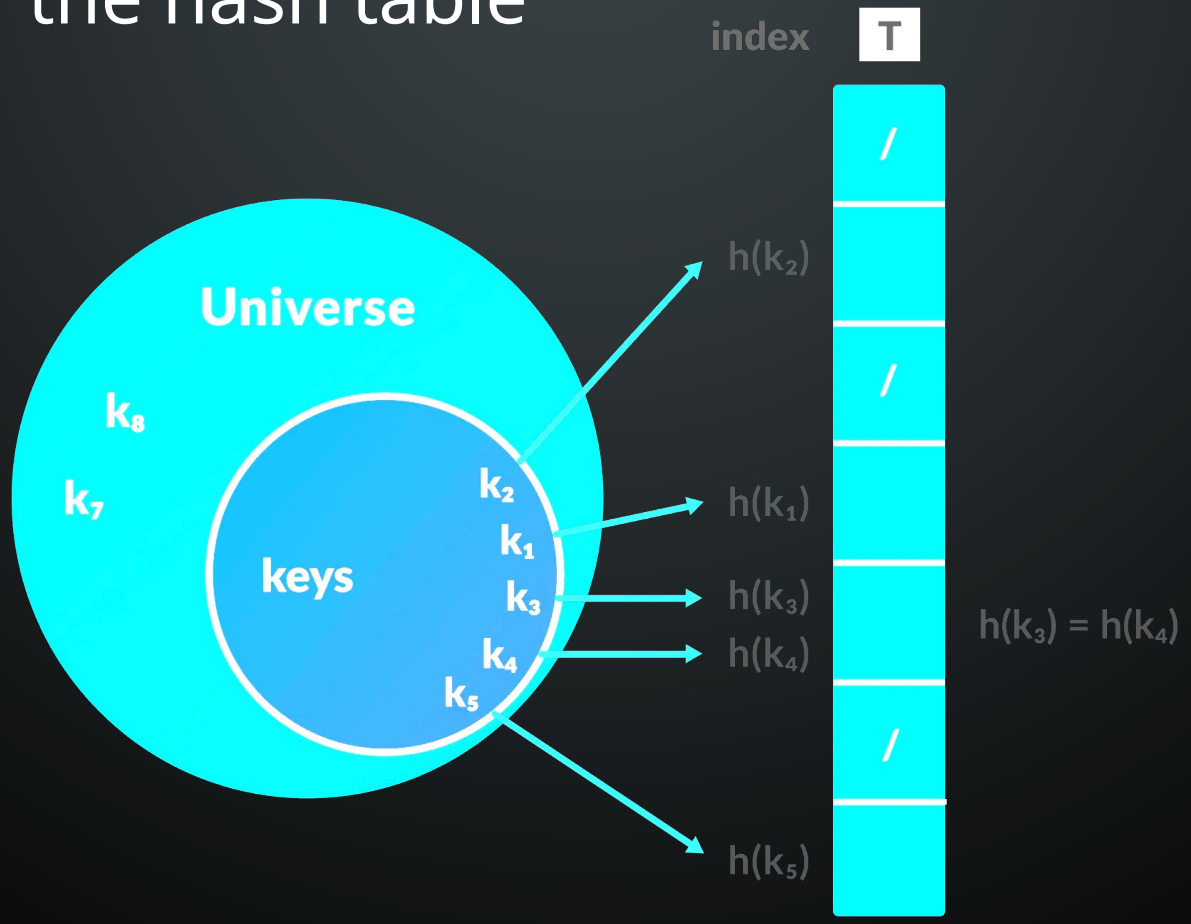
In a hash table, a new index is processed using the keys. And, the element corresponding to that key is stored in the index. This process is called hashing.

Let **k** be a key and **h**(**x**) be a hash function.

Here, **h**(**k**) will give us a new index to store the element linked with **k**.

HASH TABLE

An example of the hash table



HASH COLLISION

When the hash function generates the same index for multiple keys, there will be a conflict (what value to be stored in that index). This is called a hash collision.

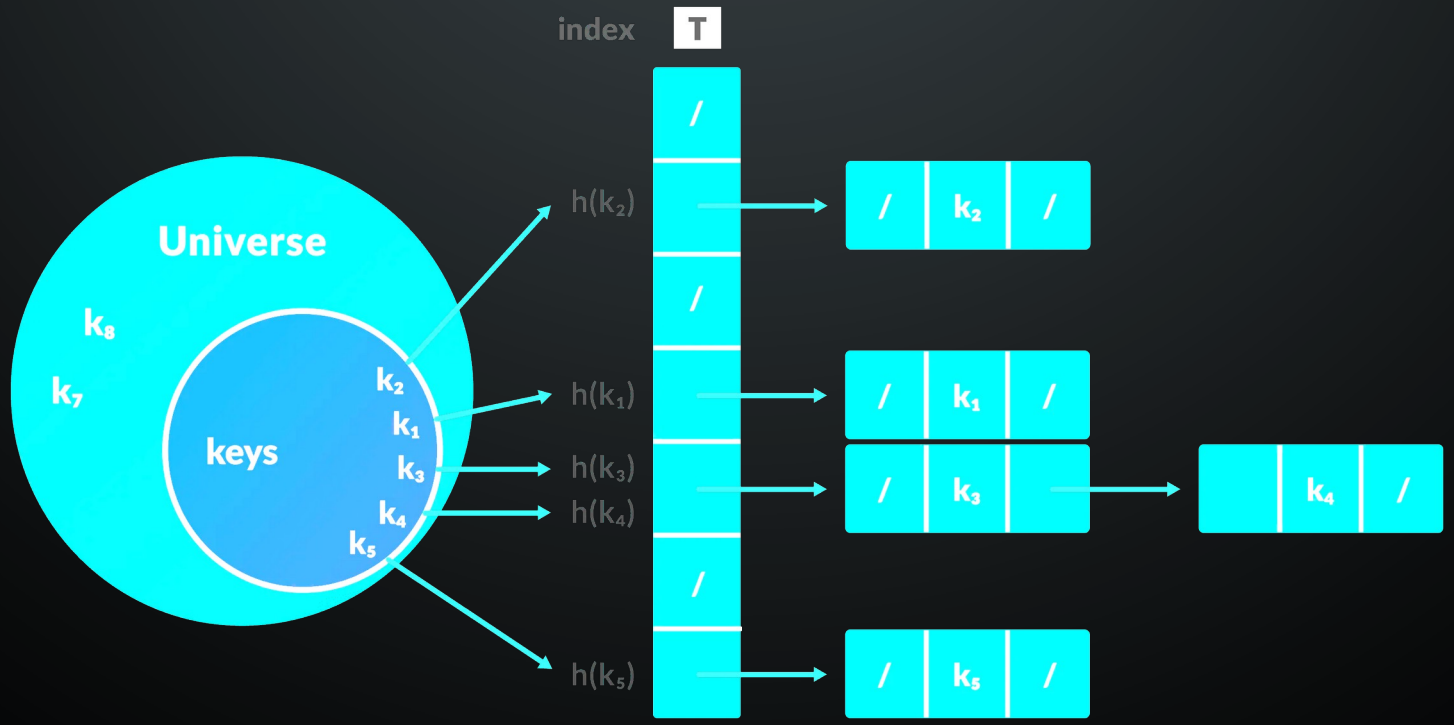
We can resolve the hash collision using one of the following techniques.

- Collision resolution by chaining
- Open Addressing: Linear/Quadratic Probing and Double Hashing

Let’s considering both methods more detail.

HASH COLLISION: RESOLUTION BY CHAINING

In chaining, if a hash function produces the same index for multiple elements, these elements are stored in the same index by using a doubly-linked list.



HASH COLLISION: OPEN ADDRESSING

Unlike chaining, open addressing doesn't store multiple elements into the same slot. Here, each slot is either filled with a single key or left **NIL**.

Different techniques used in open addressing are:

- Linear Probing
- Quadratic Probing
- Double hashing

HASH COLLISION: OPEN ADDRESSING

In linear probing, collision is resolved by checking the next slot.

$h(k, i) = (h'(k) + i) \bmod m$

where i = {0, 1, ...} and h'(k) is a new hash function.

If a collision occurs at h(k, 0), then h(k, 1) is checked. In this way, the value of **i** is incremented linearly.

The problem with linear probing is that a cluster of adjacent slots is filled. When inserting a new element, the entire cluster must be traversed. This adds to the time required to perform operations on the hash table.

HASH COLLISION: OPEN ADDRESSING

Quadratic probing works similar to linear probing but the spacing between the slots is increased (greater than one) by using the following relation.

$h(k, i) = (h'(k) + c_1*i + c_2*i^2) \bmod m$

where,

c_1 and c_2 are positive auxiliary constants,

$i = \{0, 1, ...\}$

HASH COLLISION: OPEN ADDRESSING

Double hashing

If a collision occurs after applying a hash function h(k), then another hash function is calculated for finding the next slot.

$h(k, i) = (h_1(k) + i*h_2(k)) \bmod m$

GOOD HASH FUNCTIONS

A good hash function may not prevent the collisions completely however it can reduce the number of collisions.

Here, we will look into different methods to find a good hash function:

- Division Method
- Multiplication Method
- Universal Hashing

GOOD HASH FUNCTIONS: DIVISION METHOD

If k is a key and m is the size of the hash table, the hash function h() is calculated as:

h(k) = k mod m

For example, If the size of a hash table is 10 and k = 112 then h(k) = 112 mod 10 = 2. The value of **m** must not be the powers of 2. This is because the powers of 2 in binary format are 10, 100, 1000, …. When we find **k** mod **m**, we will always get the lower order **p**-bits.

GOOD HASH FUNCTIONS: MULTIPLICATION METHOD

h(k) = ⌊m(k*A mod 1)⌋

where,

K*A mod 1 gives the fractional part k*A,

⌊ ⌋ gives the floor value

A is any constant. The value of A lies between 0 and 1. But, an optimal choice will be ≈ (√5-1)/2 suggested by Knuth.

In Universal hashing, the hash function is chosen at random independent of keys.

STRING HASH

Hashing algorithms are helpful in solving a lot of problems.

We want to solve the problem of comparing strings efficiently. The brute force way of doing so is just to compare the letters of both strings, which has a time complexity of $\mathbf{O}(\min(n_1, n_2))$ if n_1 and n_2 are the sizes of the two strings. We want to do better. The idea behind the string hashing is the following: we map each string into an integer and compare those instead of the strings. Doing this allows us to reduce the execution time of the string comparison to $\mathbf{O}(1)$.

For the conversion, we need a hash function.

STRING HASH

The goal of it is to convert a string into an integer — the hash of the string. The following condition has to hold: if two strings **s** and **t** are equal (**s**=**t**), then also their hashes have to be equal (hash(**s**)=hash(**t**)). Otherwise, we will not be able to compare strings.

Notice, the opposite direction doesn't have to hold. If the hashes are equal (hash(**s**)=hash(**t**)), then the strings do not necessarily have to be equal. E.g. a valid hash function would be simply hash(**s**)=0 for each **s**. Now, this is just a stupid example, because this function will be completely useless, but it is a valid hash function. The reason why the opposite direction doesn't have to hold, is because there are

STRING HASH

So usually we want the hash function to map strings onto numbers of a fixed range [0, m), then comparing strings is just a comparison of two integers with a fixed length. And of course, we want hash(**s**)≠hash(**t**) to be very likely if **s**≠**t**.

That's the important part that you have to keep in mind. Using hashing will not be 100% deterministically correct, because two complete different strings might have the same hash (the hashes collide). However, in a wide majority of tasks, this can be safely ignored as the probability of the hashes of two different strings colliding is still very small.

CALCULATION OF THE HASH OF A STRING

The good and widely used way to define the hash of a string **s** of length **n** is: hash(s) = s[0] + s[1]*p + s[2]*p^2 + ... + s[n-1]*p^{n-1} mod m

where **p** and **m** are some chosen, positive numbers. It is called a polynomial rolling hash function.

It is reasonable to make **p** a prime number roughly equal to the number of characters in the input alphabet. For example, if the input is composed of only lowercase letters of the English alphabet, **p**=31 is a good choice. If the input may contain both uppercase and lowercase letters, then

p=53 is a possible choice.

CALCULATION OF THE HASH OF A STRING

Obviously m should be a large number since the probability of two random strings colliding is about $\approx 1/m$. Sometimes $m=2^{64}$ is chosen, since then the integer overflows of 64-bit integers work exactly like the modulo operation. However, there exists a method, which generates colliding strings (which work independently from the choice of p). So in practice, $m=2^{64}$ is not recommended. A good choice for m is some large prime number. The recommendation is to use $m=10^9+9$. This is a large number, but still small enough so that we can perform multiplication of two values using 64-bit integers.

FAST HASH CALCULATION OF SUBSTRINGS

Given a string **s** and indices **i** and **j**, find the hash of the substring **s**[i...j].

By definition, we have:

$hash(s[i...j]) = s[i]*p^0 + s[i+1]*p^1 + ... + s[j]*p^{j-i} \bmod m.$

Multiplying both by p^i gives:

$hash(s[i...j])*p^i = hash(s[0...j]) - hash(s[0...i-1]) \bmod m.$

So by knowing the hash value of each prefix of the string **s**, we can compute the hash of any substring directly using this formula. The only problem that we face in calculating it is that we must be able to divide

FAST HASH CALCULATION OF SUBSTRINGS

However, there does exist an easier way. In most cases, rather than calculating the hashes of substring exactly, it is enough to compute the hash multiplied by some power of p. Suppose we have two hashes of two substrings, one multiplied by p^i and the other by p^j. If **i<j** then we multiply the first hash by p^{j-i}, otherwise, we multiply the second hash by p^{i-j}. By doing this, we get both the hashes multiplied by the same power of p (which is the maximum of **i** and **j**) and now these hashes can be compared easily with no need for any division.

PRACTICE: STRING HASH COMPUTATION

Problem. You have a string **S** of **N** symbols (1<=**N**<=1000) and you need to compute the hash of this string using polynomial rolling hash function: **Hash** = **S**[0]*P^0 + **S**[1]*P^1 + **S**[2]*P^2 + … + **S**[n-1]*P^{N-1} mod **M**. Use the values for **P**=1009 and for **M**=10^9+7.

Task. Create a program using C/C++/Python to solve this problem.

Input. A single string of **N** symbols.

Output. The hash of input string.

Example. Input string '**Samsung**' has hash **628380962**.

PRACTICE: STRING HASH COMPUTATION

Code example:

```
#include <iostream>
#define MAX 1000
#define P   1009
#define M   1000000007
char S[MAX];
long long A;
int main()
{
    cin >> S;

    // Your code here

    cout << A;

    return 0;
}
```

THANK YOU!