Hashing

Jordi Cortadella and Jordi Petit
Department of Computer Science

The parking lot

- We want to keep a database of the cars inside a parking lot. The database is automatically updated each time the cameras at the entry and exit points of the parking read the plate of a car.

- Each plate is represented by a free-format short string of alphanumeric characters (each country has a different system).

- The following operations are needed:
  - Add a plate to the database (when a car enters).
  - Remove a plate from the database (when a car exits).
  - Check whether a car is in the parking.

- **Constraint**: we want the previous operations to be very efficient, i.e., executed in ***constant time***.
  (*This constraint is overly artificial, since the activity in a parking lot is extremely slow compared to the speed of a computer.)*

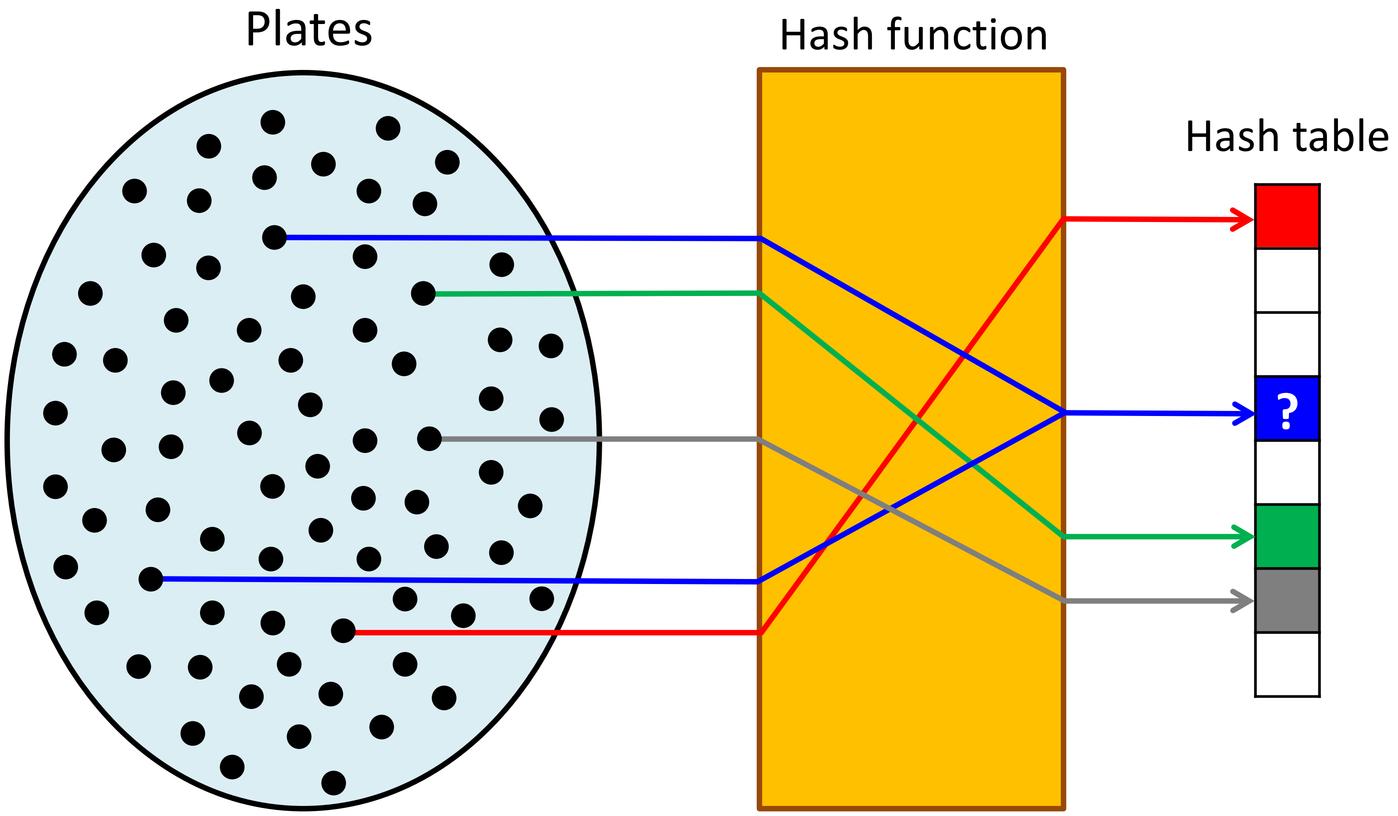
Naïve implementation options

- Lists, vectors or binary search trees are not valid options, since the operations take too long:
  - Unsorted lists: adding takes $\mathrm{O}(1)$. Removing/checking takes $\mathrm{O}(n)$.
  - Sorted vector: adding/removing takes $\mathrm{O}(n)$. Checking takes $\mathrm{O}(\log n)$.
  - AVL trees: adding/removing/checking takes $\mathrm{O}(\log n)$.

- A (Boolean) vector with one location for each possible plate:
  - The operations could be done in constant time!, but …
  - The vector would be extremely large (e.g., only the Spanish system can have 80,000,000 different plates).
  - We may not even know the size of the domain (all plates in the world).
  - Most of the vector locations would be "empty" (e.g. assume that the parking has 1,000 places).

- Can we use a data structure with size $\mathrm{O}(n)$, where n is the size of the parking?

Hashing



A hash function maps data of arbitrary size to a table of fixed size.

Important questions:

- How to design a good hash function?
- The hash function is not injective. How to handle collisions?

Hash function

- We can calculate the location for item x as

$$h(x) \mod S$$

  where h is the hash function and S is the size of the hash table.

- A good hash function must scatter items *uniformly* (to minimize the impact of collisions).

- A hash function must also be *consistent*, i.e., give the same result each time it is applied to the same item.

Hashing the plates: some attempts

- Add the last three characters (e.g., ASCII codes) of plate:

  $$h(x) = x_{n-1} + x_{n-2} + x_{n-3}$$

  Bad choice: For the Spanish system, this would concentrate the values between 198 (BBB) and 270 (ZZZ).

- Multiply the last three characters:

  $$h(x) = x_{n-1} \cdot x_{n-2} \cdot x_{n-3}$$

  The values are distributed between 287,496 and 729,000. However the distribution is not uniform. The last three characters denote the age of the car. The population of new cars is larger than the one of old cars (e.g., about 15% of the cars are less than 1-year old).

  Moreover: consecutive plates would fall into the same slot. Some companies (e.g., car renting) have cars with consecutive plates and they could be located in the neighbourhood of the parking lot.

Hashing the plates: some attempts

- Multiply all characters of the plate:

  $$h(x) = x_0 \cdot x_1 \cdots x_{n-1}$$

  Better choice, but not fully random and uniform. Two plates with permutations of characters would fall into the same slot, e.g., 3812 DXF and 8321 FDX.

- The perfect hash function does not exist, but using ***prime numbers*** is a good option since most data have no structure related to prime numbers.

- Where can we use prime numbers?
  - In the size of the hash table
  - In the coefficients of the hash function

Example of hash function for strings

A usual hash function for a string with size n is the *polynomial rolling hash function*:

$$h(x) = \sum_{i=0}^{n-1} x_i \cdot p^i \mod m$$

where

- p is a prime number (e.g., 29791, 11111, …)
- m is a large prime (to reduce the probability of collision) $10^9 + 7$ and $10^9 + 9$ are widely used.
- and x_i is the character at location i

This function can be efficiently implemented using Horner's rule for the evaluation of a polynomial.

Polynomial rolling hash function

```python
def hash(s: str)-> int:
    """Hash function for strings"""
    p, m = 31, 10**9 + 7
    h = 0
    p_pow = 1
    for c in s:
        h = (h + ord(c) * p_pow) % m
        p_pow = (p_pow * p) % m
    return h


print(hash('hello'), hash('bye'), hash('hash'))
105835282 100910 3387231
```

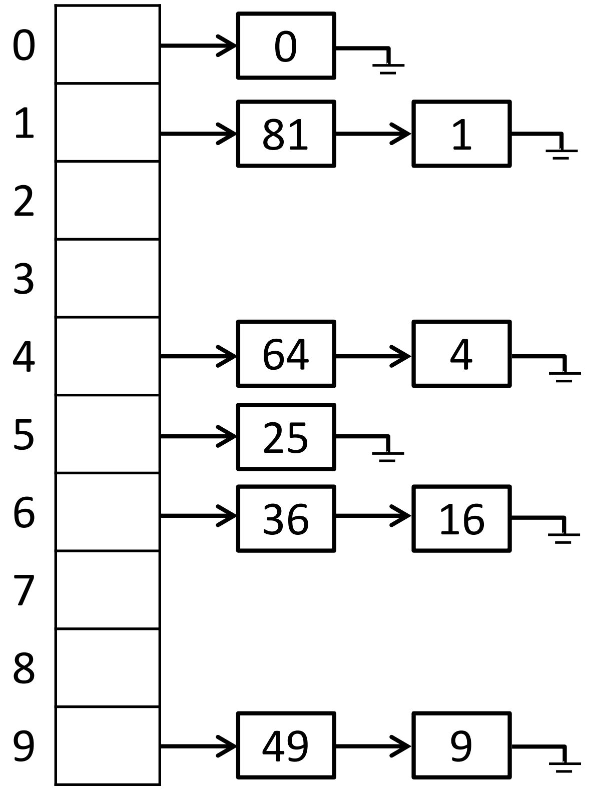
Handling collisions

- A collision is produced when

$$h(x_1) \equiv h(x_2) \mod S$$

- There are two main strategies to handle collisions:
  - Using lists of items with the same hash value (separate chaining)
  - Using alternative cells in the same hash table (linear probing, double hashing, …)

Handling collisions: separate chaining



(perfect squares mod 10)

Each slot is a list of the items that have the same hash value.

Load factor: $\lambda = \frac{\text{number of items}}{\text{table size}}$

λ is the average length of a list.

A successful search takes about $\lambda/2$ links to be traversed, on average.

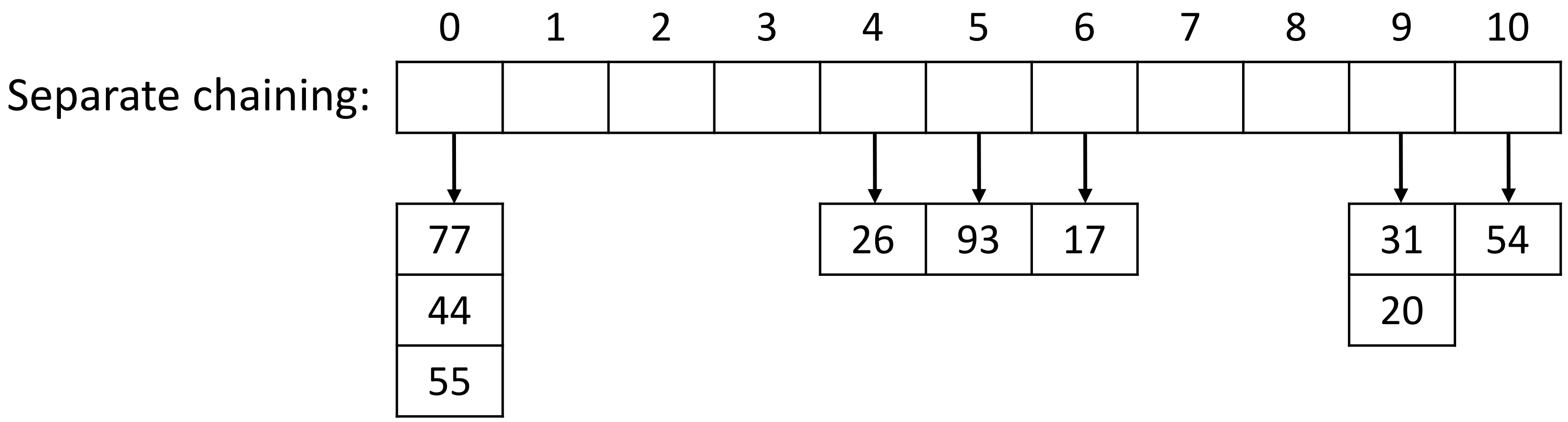
Table size: make it similar to the number of expected items.
Common strategy: when $\lambda > 1$, do rehashing.

Handling collisions: using the same hash table

- If the slot is occupied, find alternative cells in the same table. To avoid long trips finding empty slots, the load factor should be below $\lambda = 0.5$.
- Deletions must be “lazy” (slots must be invalidated but not deleted, thus avoiding truncated searches).
- **Linear probing:** if the slot is occupied, use the next empty slot in the table.
- **Double hashing:** if the slot is occupied using the first hash function h_1, use a second hash function h_2. The sequence of slots that is visited is $h_1(x)$, $h_1(x) + h_2(x)$, $h_1(x) + 2h_2(x)$, etc.

An example

Insertion of the elements 54, 26, 93, 17, 77, 31, 44, 55, 20.
Hash function: $h(x) = x \bmod 11$.



Linear probing:

0	1	2	3	4	5	6	7	8	9	10
77	44	55	20	26	93	17			31	54

What if we remove 55? Use lazy deletion!

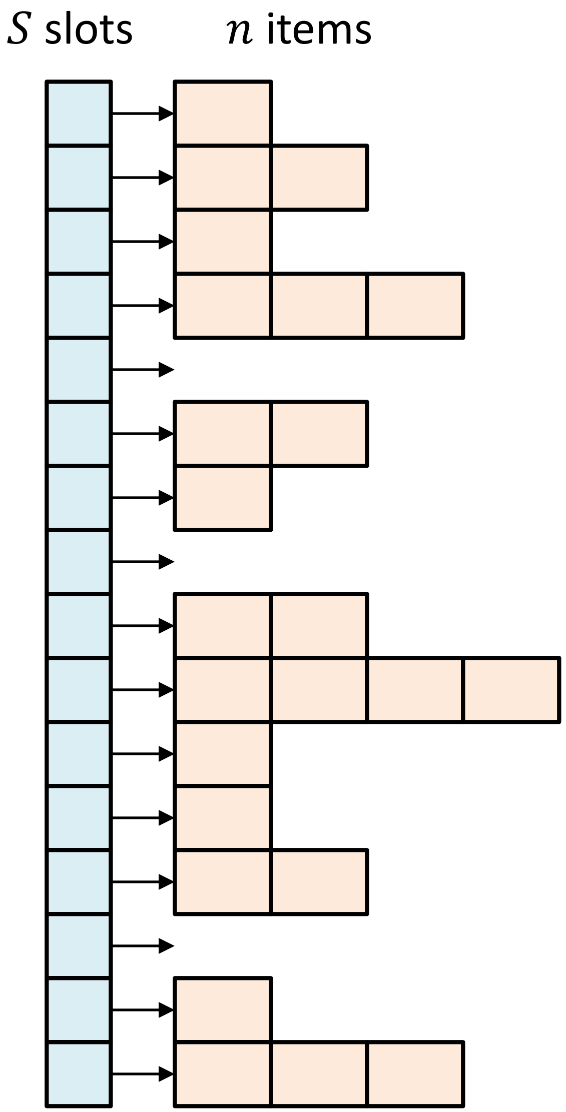
Rehashing

- When the table gets too full, the probability of collision increases (and the cost of each operation).

- Rehashing requires building another table with a larger size and rehash all the elements to the new table. Running time: $\mathrm{O}(n)$.

- New size: $2n$ (or a prime number close to it). Rehashing occurs very infrequently and the cost is amortized by all the insertions. The average cost remains constant.

Complexity analysis



The hash table occupies $\mathrm{O}(S+n)$ space.
Each slot has n/S items, on average.
The runtime to find an item is $\mathrm{O}(n/S)$, on average.

Cases	**Space: $\mathrm{O}(n+S)$**	**Time: $\mathrm{O}(n/S)$**
$S \gg n$	$\mathrm{O}(S)$	$\mathrm{O}(1)$
$n \gg S$	$\mathrm{O}(n)$	$\mathrm{O}(n)$
$S = \mathrm{O}(n)$	$\mathrm{O}(n)$	$\mathrm{O}(1)$

The best strategy is to have $S = \mathrm{O}(n)$ that allows to maintain a constant-time access without wasting too much memory.

Rehashing should be applied to maintain $S = \mathrm{O}(n)$.

Sets and Dictionaries

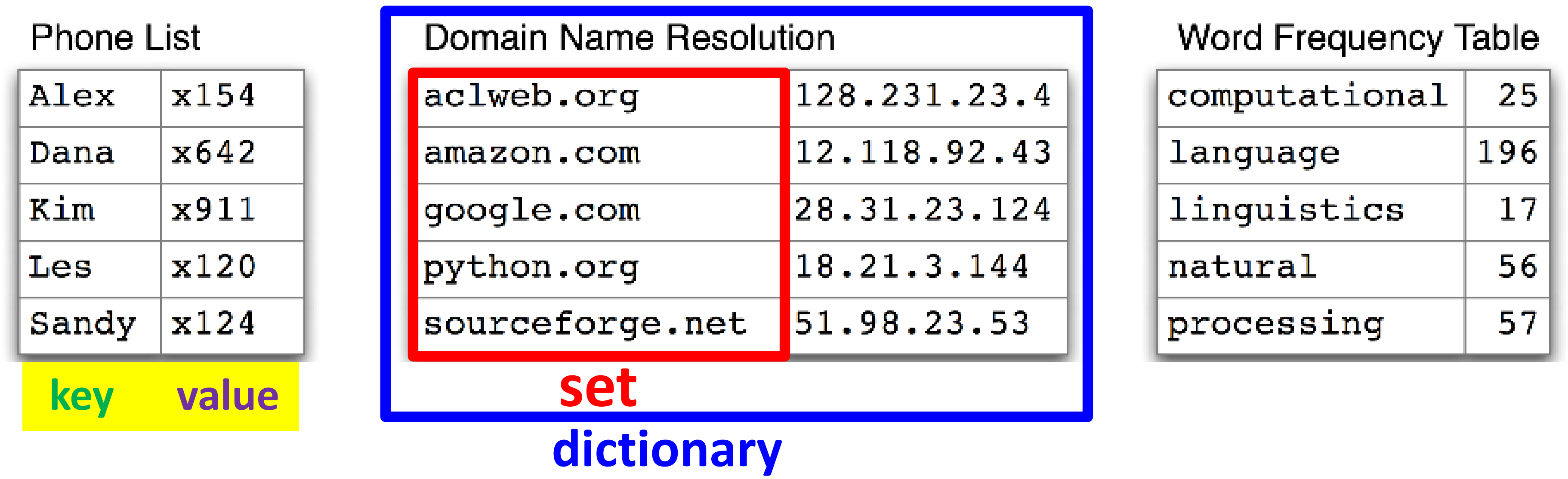
- A set: a collection of items. The typical operations are:
  - Add/remove one element
  - Does it contain an element?
  - Size?, Is it empty?
  - Visit all items

- A dictionary (map): a collection of key-value pairs. The typical operations are:
  - Put a new key-value pair
  - Remove a key-value pair with a specific key
  - Get the value associated to a key
  - Does it contain a key?
  - Visit all key-value pairs

Sets and Dictionaries

- A dictionary can be treated as a set of keys, each key having an associated value.

- We will focus on the implementation of sets.


key
value
set
dictionary

Phone List

Alex	x154
Dana	x642
Kim	x911
Les	x120
Sandy	x124

Domain Name Resolution

aclweb.org	128.231.23.4
amazon.com	12.118.92.43
google.com	28.31.23.124
python.org	18.21.3.144
sourceforge.net	51.98.23.53

Word Frequency Table

computational	25
language	196
linguistics	17
natural	56
processing	57

Source: Natural Language Processing with Python, by Steven Bird, Ewan Klein and Edward Loper

Binary Search Trees vs. Hash Tables

Operation	Binary Search Tree	Hash Table
Insertion/Deletion/Lookup	O$(\log n)$	O(1)
Sorted Iteration	In-order traversal: O(n)	Needs an extra sorted vector: O$(n \log n)$
Hash function	Not required	Required
Total order	Required	Not required
Min/max search	O$(\log n)$	O(n)
Range search	O$(\log n)$	O(n)

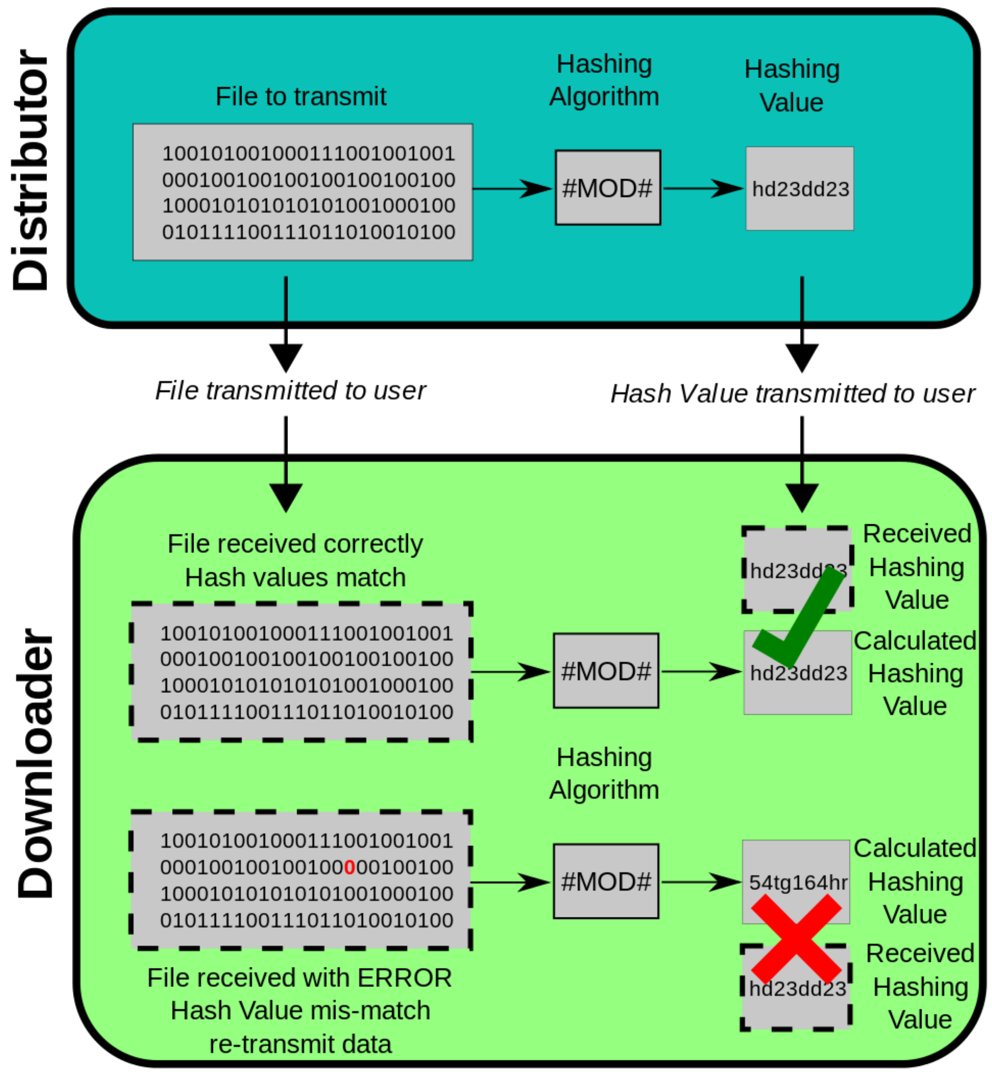
Python:

- **`set`** and **`dict`** are implemented with hash tables

C++ STL:

- **`set`** and **`map`** are implemented with BSTs
- **`unordered_set`** and **`unordered_map`** are implemented with hash tables

Application: data integrity check

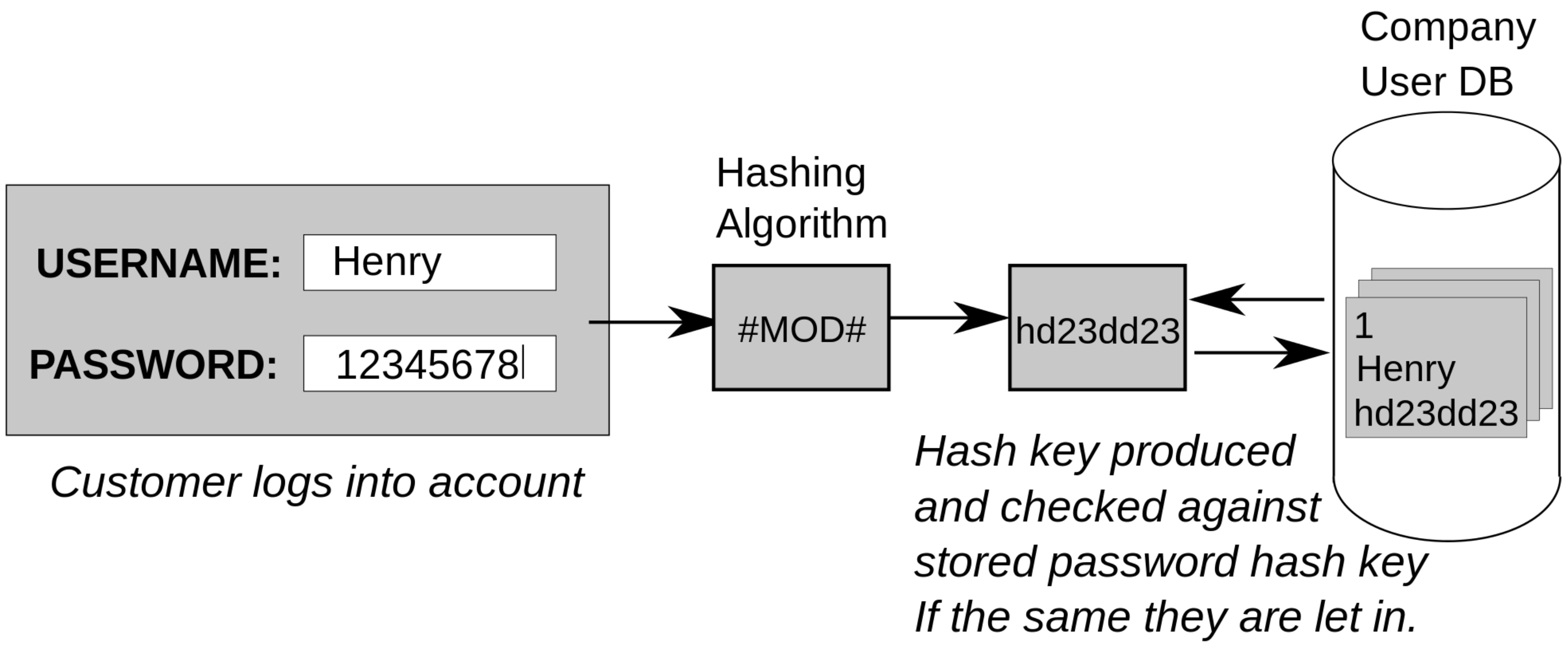


Hash functions are used to guarantee the integrity of data (files, messages, etc) when distributed between different locations.

Different hashing algorithms exist: SHA1, SHA255, …

The probability of collision is extremely low.

Application: password verification



Security is based on the fact that hashing functions are cryptographic (not reversible).

Be careful: there are databases of hash values for “popular” passwords
(e.g., 123456, qwerty, password, barcelona, android,…).

Bloom filters

- Let us assume that we have an internet router that wants to filter the packets from a set of blacklisted IPs. The router has an speed of 100 Gbits/s. New IPs can be added to the blacklist dynamically.

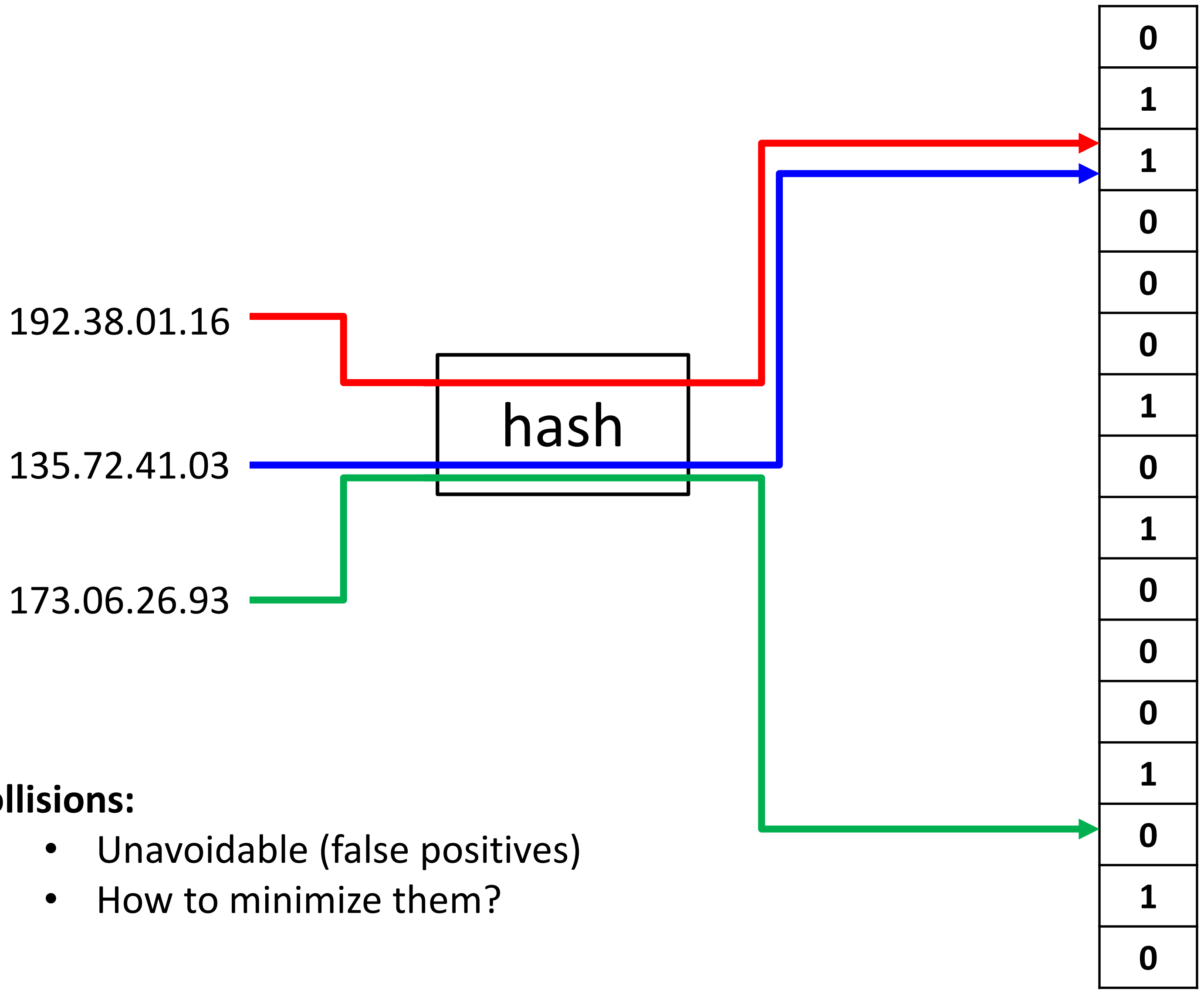
- IPv4 addresses have 32 bits (4,294,967,296 possible addresses).

- Each time the router receives a packet, it must perform a search to check whether the IP is blocked. In most cases, the IP will not be blocked.

- We would like to have a *fast* and *small* data structure to avoid the packets waiting for the search of their IP.

- Bloom filters give a space-efficient solution to the membership testing problem. They do not store the keys of the set. But the answer is probabilistic (small probability of false positives).

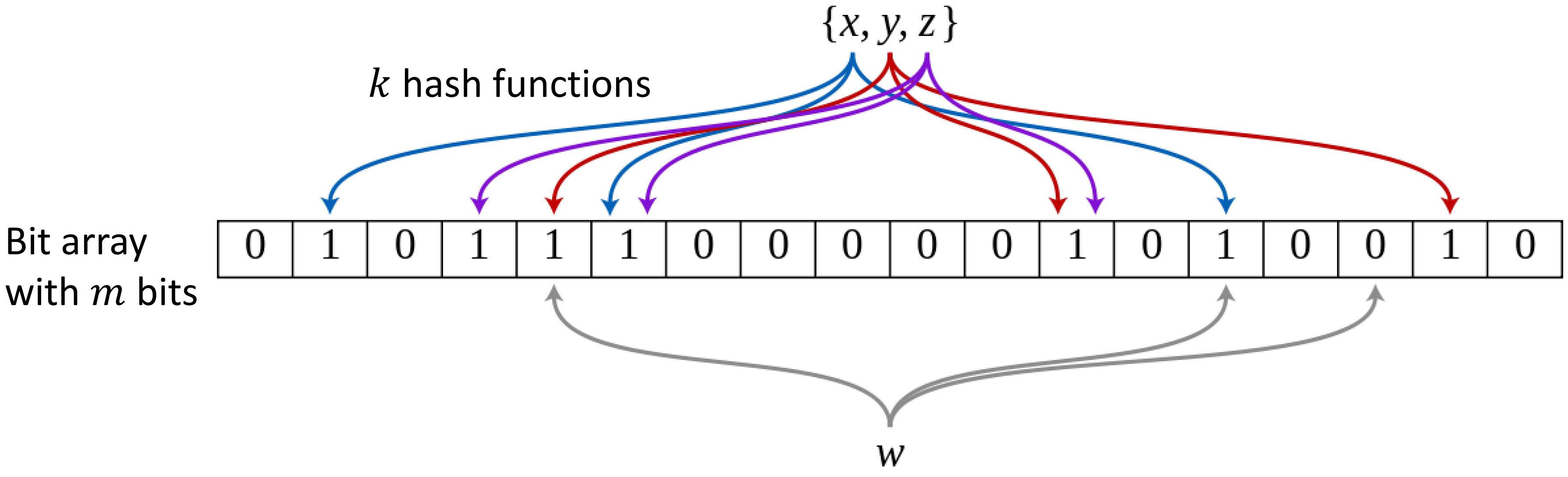
Simple Bloom filter



Collisions:

- Unavoidable (false positives)
- How to minimize them?

How a Bloom filter works



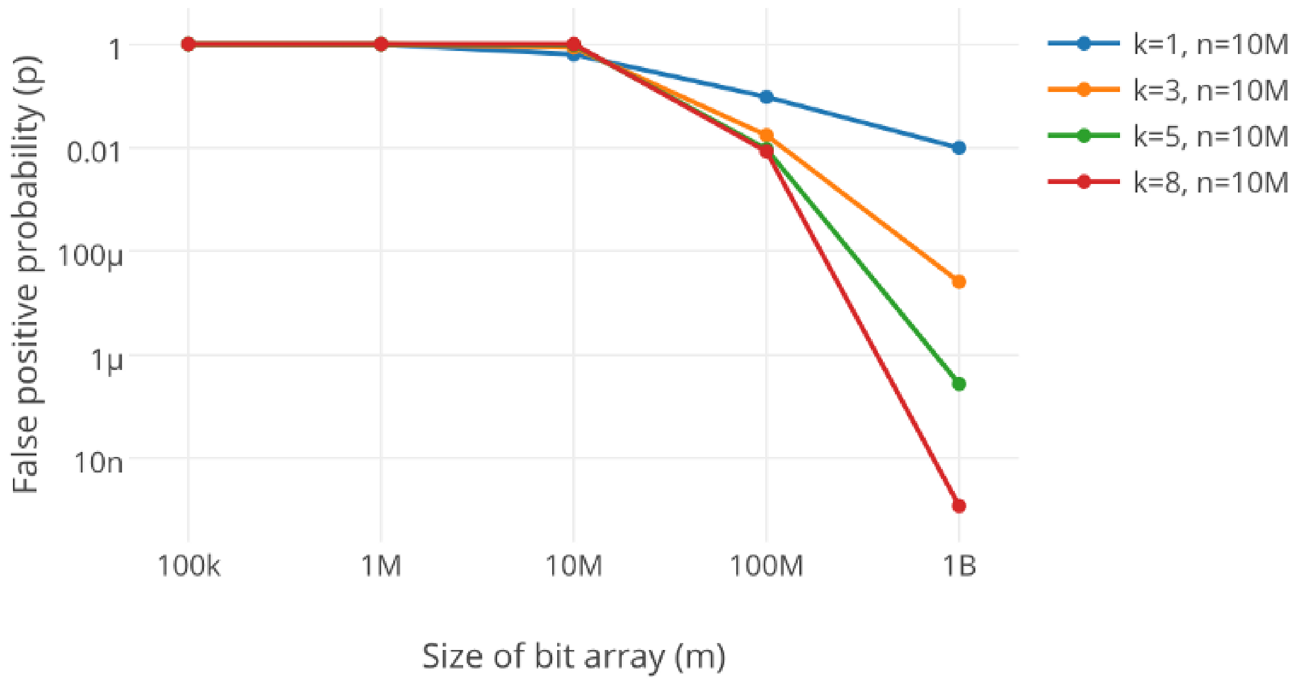
- Initially, all array positions are at 0.
- To add an element, feed it to each of the k hash functions to get k array positions. Set the bits at all these positions to 1.
- To test whether an element is in the set, feed it to each of the k hash functions to get k array positions. If any of the bits at these positions is 0, the element is not in the set. If all are 1, then either the element is in the set, or the bits have by chance been set to 1 during the insertion of other elements, resulting in a *false positive*.
- Note: "basic" Bloom filters do not allow to delete elements.

Bloom filter: probabilistic analysis

- Goal: given a set of elements, what is the size of the bit array and the number of hash functions required to achieve a false positive rate ε?

- Assumptions for the probabilistic analysis:
  - Hash functions select each position of the array with equal probability
  - Different hash functions are independent

Bloom filter probabilistic analysis



Source: Abishek Bhat, "Use the bloom filter, Luke!"
https://medium.com/engineering-semantics3/use-the-bloom-filter-luke-b59fd0839fc4

Intuition:

- As we increase the size of the array (m), false positives decrease
- As we increase the number of hash functions (k), … (not clear)

Bloom filter: probabilistic analysis

- Probability that a bit is not set to 1 by any of the hash functions:

$$\left(1-\frac{1}{m}\right)^{k}$$

- A well-known identity:

$$\lim_{m\to\infty}\left(1-\frac{1}{m}\right)^{m}=\frac{1}{e}$$

- Thus, for large m:

$$\left(1-\frac{1}{m}\right)^{k}=\left(\left(1-\frac{1}{m}\right)^{m}\right)^{\frac{k}{m}}\approx e^{-\frac{k}{m}}$$

- After inserting n elements, the probability that a certain bit is still 0 is

$$e^{-\frac{kn}{m}}$$

- And the probability that is 1 is

$$1-e^{-\frac{kn}{m}}$$

Bloom filter: probabilistic analysis

- The probability of a false positive (ε) is the probability that each of the k positions computed by the hash functions are at 1, i.e.,

$$\varepsilon = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \approx \left(1 - e^{-\frac{kn}{m}}\right)^{k}$$

- For a given n and m, the optimal number of hash funcions to minimize ε is:

$$k = \frac{m}{n} \ln 2 \approx 0.693 \frac{m}{n}$$

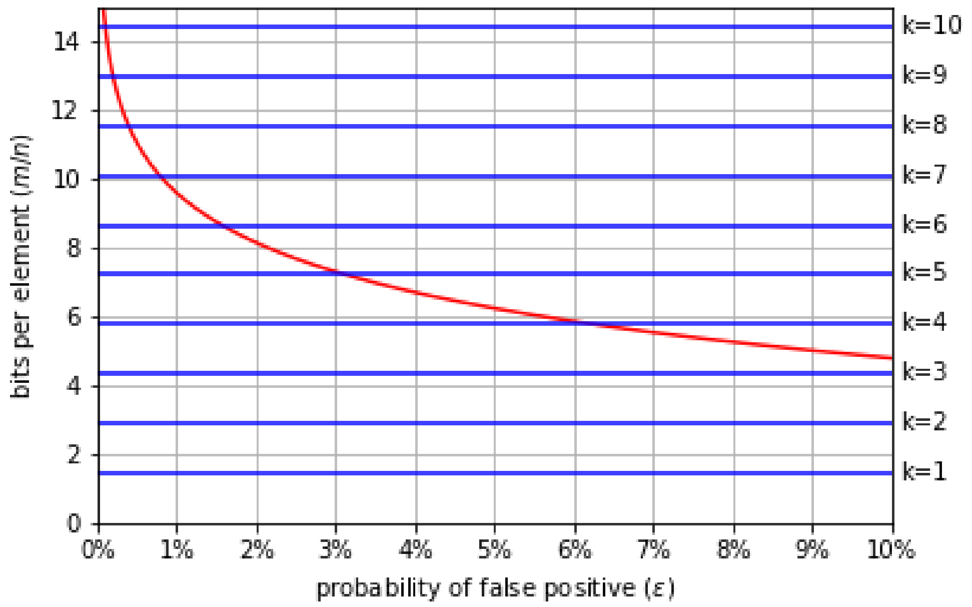
- Given ε and n, the optimal size of the bit array and number of hash funcions is:

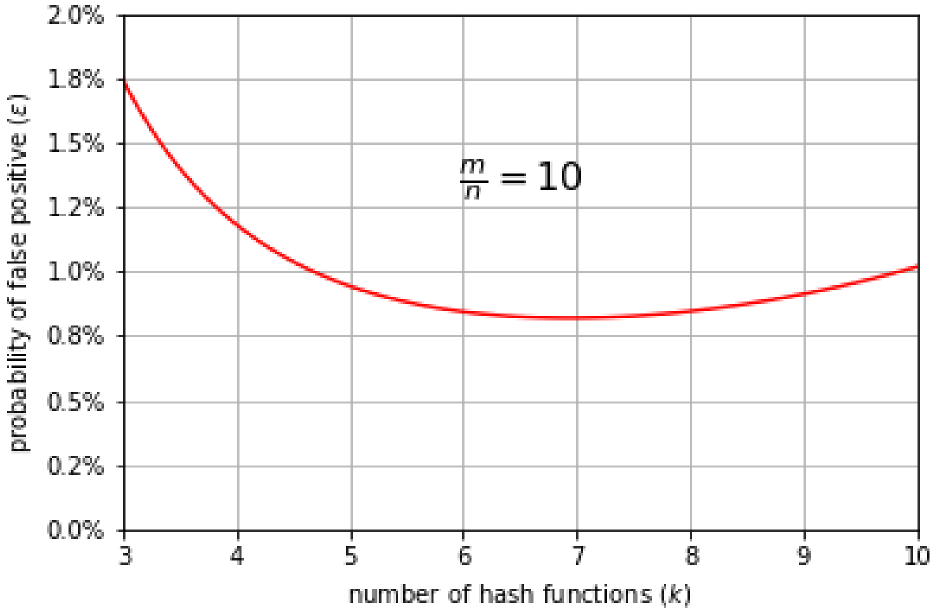
$$m = -\frac{n \log_2 \varepsilon}{\ln 2} \qquad\qquad k = -\log_2 \varepsilon$$

- Equivalently, the number of bits per element must be

$$\frac{m}{n} \approx -1.44 \log_2 \varepsilon$$

Bloom filter: probabilistic analysis





EXERCISES

Hash function

Given the values {2341, 4234, 2839, 430, 22, 397, 3920}, a hash table of size 7, and hash function $h(x) = x \bmod 7$, show the resulting tables after inserting the values in the given order with each of these collision strategies:

- Separate chaining
- Linear probing

All elements different

Let us assume that we have a list with n elements. Design an algorithm that can check that all elements are different. Analyze the complexity of the algorithm considering different data structures:

- Checking the elements without any additional data structure, i.e., using the same list
- Using balanced BSTs
- Using hash tables

String matching

- Let us assume that we have two strings, a pattern s and a text t. Propose an algorithm to find all occurrences of the pattern in the text.

- Constraint: we want to do it in $\mathrm{O}(|s| + |t|)$ time.

- Hint: Use the polynomial rolling hash function. Propose an efficient (incremental) method to calculate $\mathrm{hash}(t[i \dots j])$ from $\mathrm{hash}(t[i-1 \dots j-1])$.

Bloom filter parameters

- Let us consider a Bloom filter with $n = 10^6$ elements.
  - Give the size of the bit array and the number of hash functions to achieve a 1% false positive rate

- Let us consider a Bloom filter with $m = 10^7$.
  - What is the maximum number of elements it must contain to guarantee a false positive rate smaller than 2%?
  - How many hash functions would be required?

Bloom filter

Let us consider a Bloom filter with $m = 17$ and three hash functions:

$$h_1(x) = 19x, \;\; h_2(x) = 23x, \;\; h_3(x) = 31x$$

Insert the values 1 and 2, and give the smallest positive integer than produces a false positive