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.)*

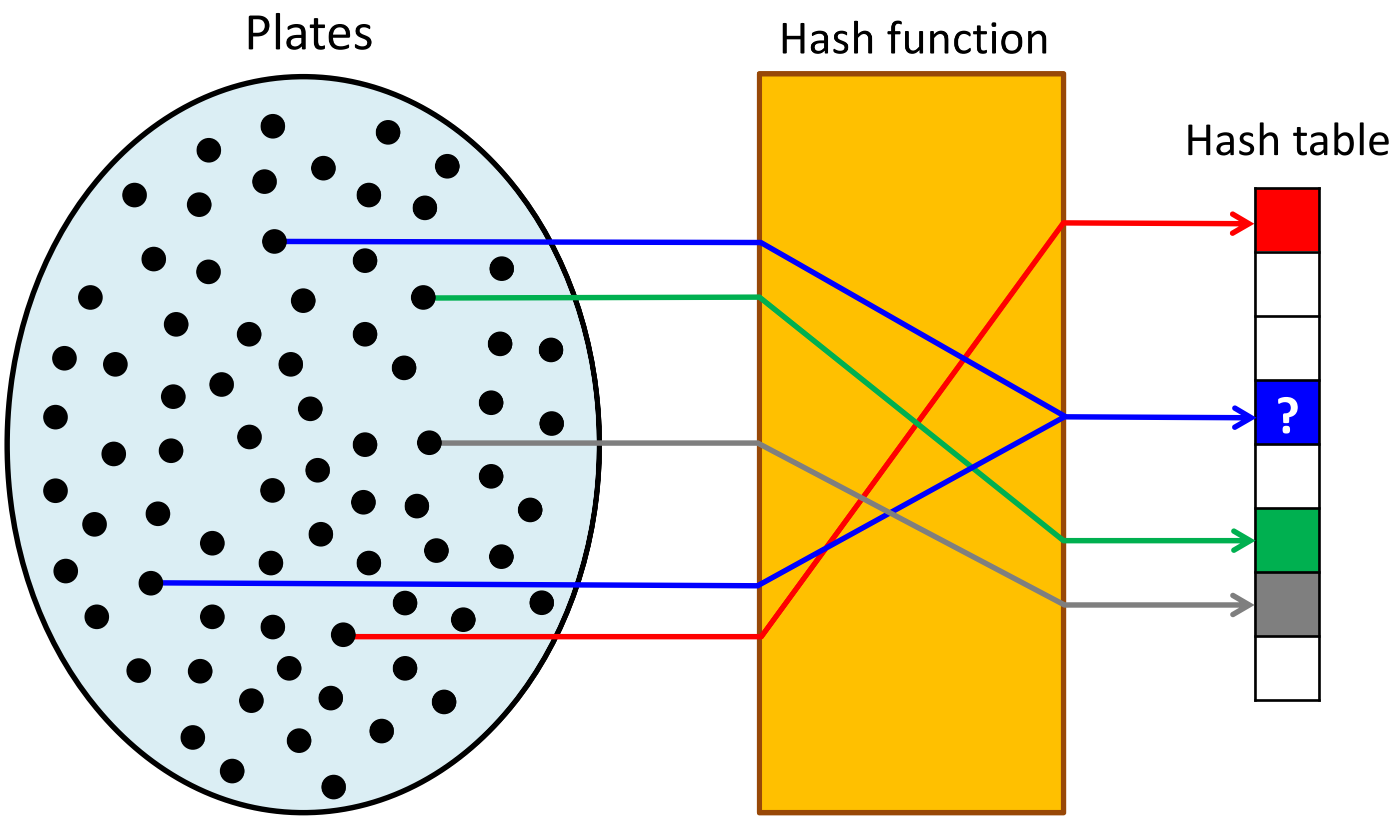
Naïve implementation options

- 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

$$h(x) \mod m$$

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

- A good hash function must scatter items *randomly* and *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 as follows:

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

  where p is a prime number 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.

- Here is a slightly different implementation (reversed string):

```
/** Hash function for strings */
unsigned int hash(const string& key, int tableSize) {
  unsigned int hval = 0;
  for (char c: key) hval = 37*hval + c;
  return hval%tableSize;
}
```

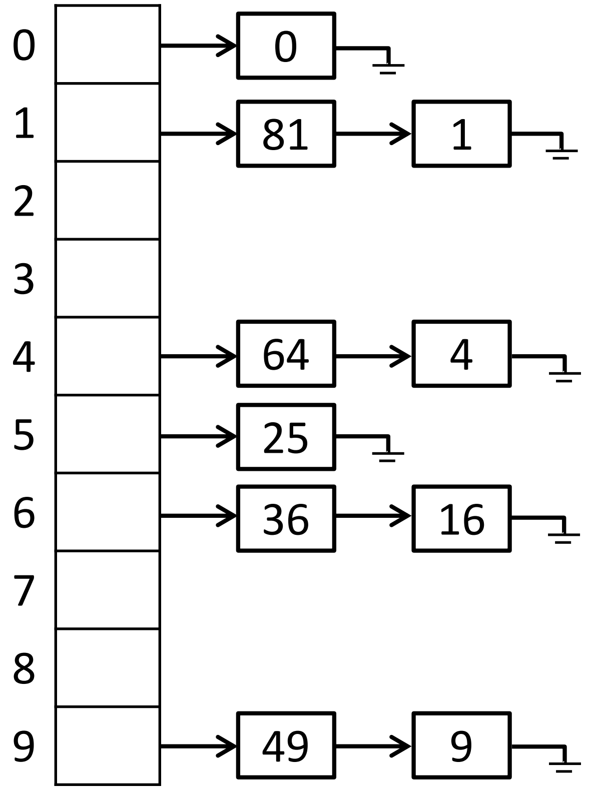
Handling collisions

- A collision is produced when

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

- 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: $\lambda = \dfrac{\text{number of items}}{\text{table size}}$

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).

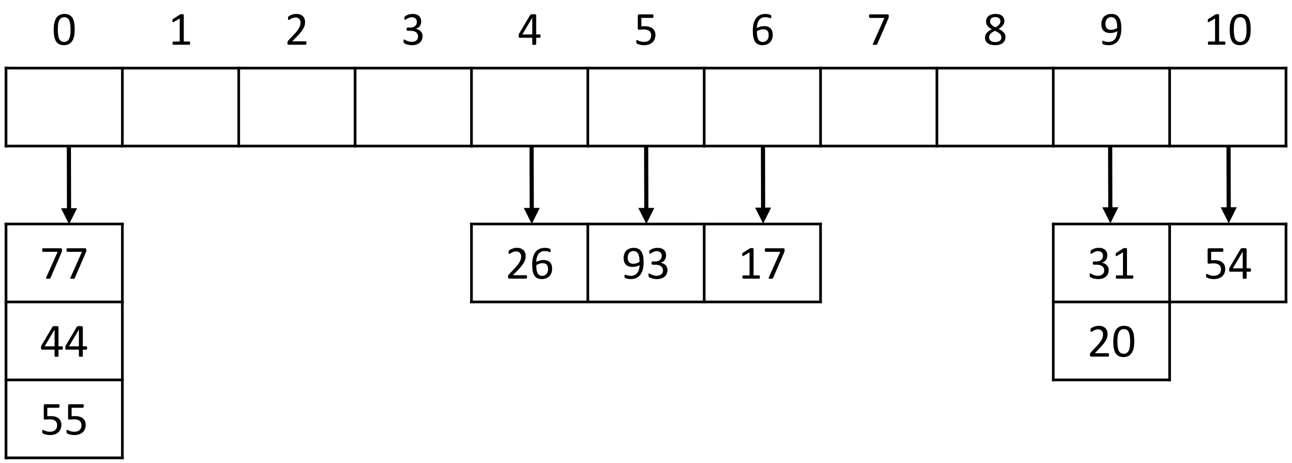
- **Linear probing:** if the slot is occupied, use the next empty slot in the table.

- **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: $h(x) = x \bmod 11$.

Separate chaining:



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!

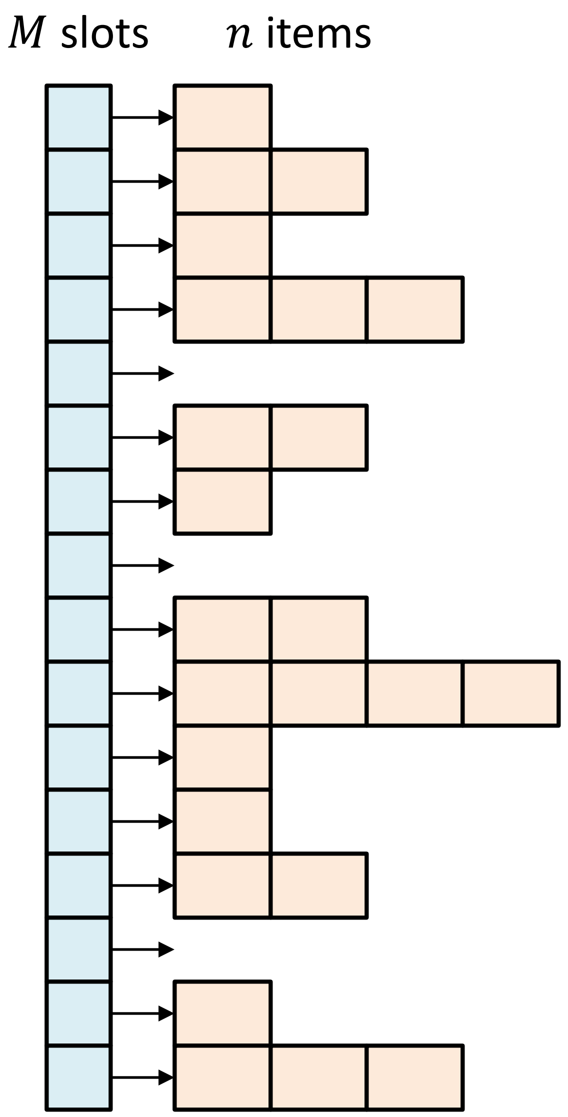
Rehashing

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

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

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

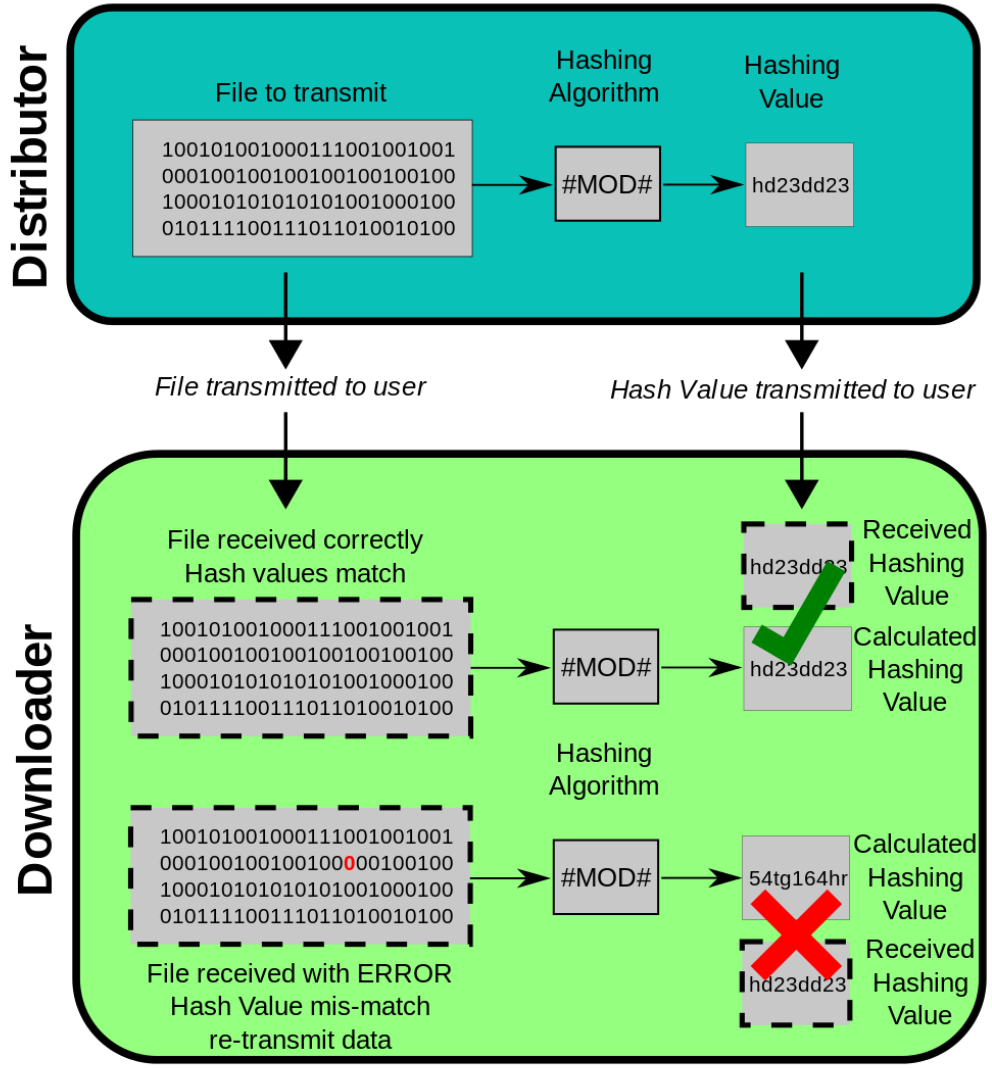
Rehashing should be applied to maintain $M = \mathrm{O}(n)$.

Binary Search Trees vs. Hash Tables

Not a clear winner

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
Range search	$O(\log n)$	$O(n)$

Application: data integrity check

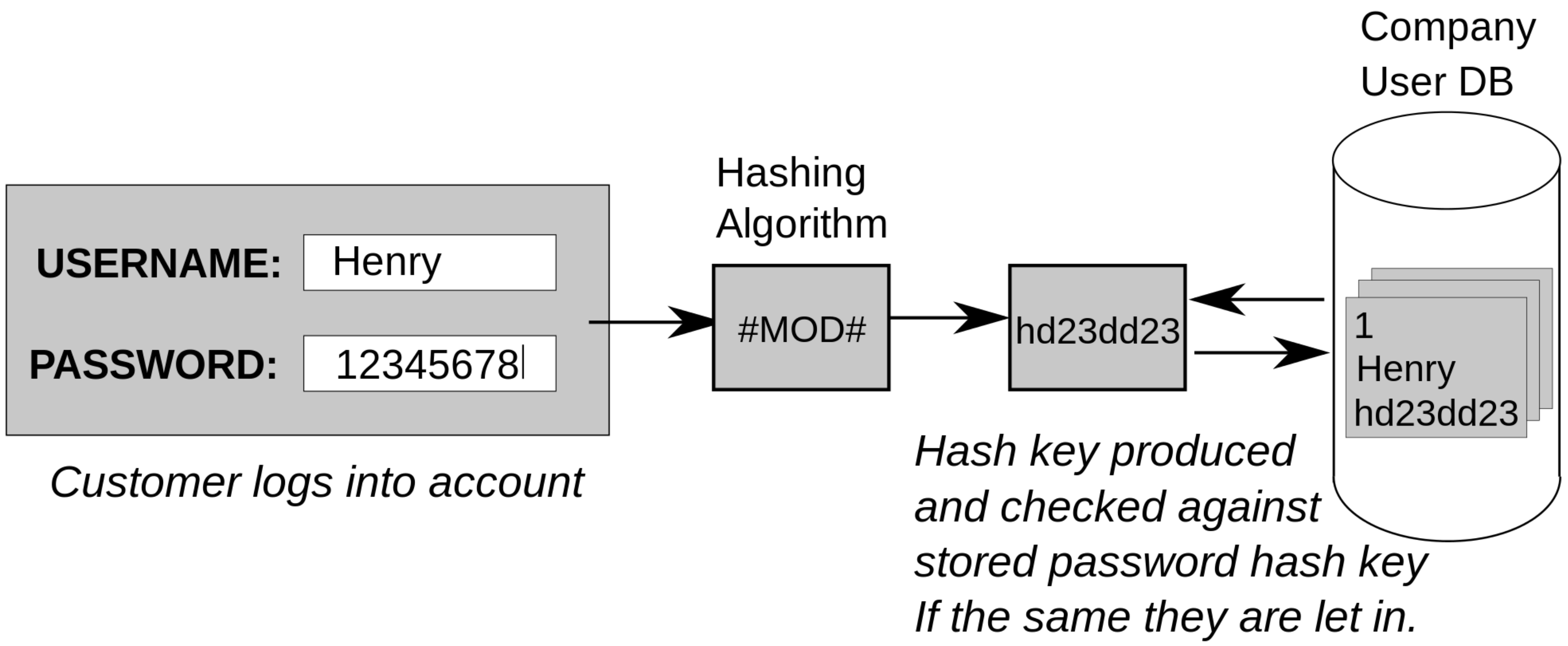


Hash functions are used to guarantee the integrity of data (files, messages, etc) when distributed between different locations.

Different hashing algorithms exist:
MD5, 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., 1234, qwert, Messi10, Barcelona92,…).

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 AVLs.
- Using hash tables.