# 7- Hashing

Bruno MARTIN,
University of Nice - Sophia Antipolis
mailto:Bruno.Martin@unice.fr
http://www.i3s.unice.fr/~bmartin/mathmods.html



## Hashing

The steps in hashing:

1. compute a **hash function** which maps keys in table addresses

   Since there are more records ($N$) than indexes ($M$) in the table, two or more keys may hash to the same table address : it's the **collision** problem
2. the **collision resolution** process

Good hash functions should uniformly distribute entries in the table

Since, if the function uniformly distributes the keys, the complexity of searching is approx. divided by the table's size



## Hashing

Hashing is a **completely different** method of searching

**The idea is to access directly the record in a table using its key - the same way an index accesses an entry in an array -**

We use a hash function that computes a table index from the key

**Basic operations:** insert, remove, search



## Transform Keys into Integers in $[\![0, \; M-1]\!]$

If the key is already a large integer

- choose $M$ to be a prime and compute $key \mod M$

If the key is an uppercase character string

- encode each char in a 5-bit code (5 bits ($2^5$) are required to encode 26 items): each letter is encoded by the binary value of its rank in the alphabet
- compute the modulo of the corresponding decimal value

**Example**

$ABC \Rightarrow$ 00001 00010 00011 $\Rightarrow$
$1 * (2^5)^2 + 2 * (2^5)^1 + 3 * (2^5)^0 = 1091 \Rightarrow 1091 \mod M \Rightarrow$
*index table*

## Why does $M$ have to be prime ?

An example of hash function is

$$hash(key) = \left(key[0] \times (2^k)^0 + key[1] \times (2^k)^1 + \ldots + key[n] \times (2^k)^n\right) \mod M$$

Suppose you choose $M = 2^k$ then

- $XXX \ mod \ M$ is unaffected by adding to $XXX$ multiples of $2^k$
- $hash(key) = key[0]$ : *hash* only depends on the $1^{st}$ char of *key*

The simplest way to ensure that the *hash* function takes all the characters of a key into account is to take $M$ **prime**



## Example

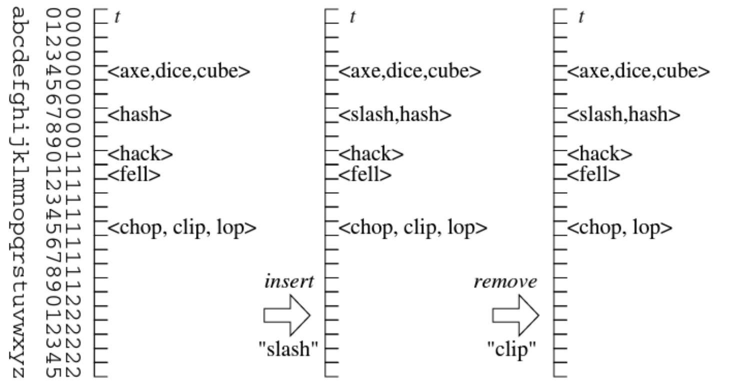


**Fig. 4.1.** Hashing with chaining. We have a table $t$ of sequences. The figure shows an example where a set of words (short synonyms of "hash") is stored using a hash function that maps the last character to the integers 0..25. We see that this hash function is not very good



## How to Handle the Collision Process

We have an array of size $M$ - called the hash table - and a *hash* function which gives for any key a possible entry in this array

**Problem**: decide what to do when 2 keys hash to the same address

A first simple method is to build for each table entry a **linked list** of records whose keys hash to the same entry
Colliding records are chained together we call it **separate chaining**
At the initialization, the hash table will be an array of $M$ pointers to empty linked lists



## Searching a record in a Hash Table with linked lists

Main operation on a *HashTable*: **search** a record with its *key*:

- compute the *hash* value of the *key* : $hash(key) = i$
- access to the linked list at position $i$ : $HashTable[i]$
- if there's more than your record in the list you have collisions
- searching becomes a search in a list: iterate on each record comparing the keys
- unsuccessful search: you iterate down the list without finding your record
- Operations of insertion and removal of records in a Hash Table become linked list operations

## Searching Performances

Good hash functions uniformly distribute $N$ entries over the $M$ positions of the table

Searching expected values in $O(\alpha)$ ($\alpha = \frac{N}{M}$ table's filling rate):

- Unsuccessfull: $\frac{1}{M}\sum_M(1+\sharp L_i)$ since the element $\notin L_i$

$$\overline{Q^-}(M,N) = \alpha + 1 \qquad \text{since } \sum \sharp L_i = N$$

- Successful: searching for an element in the table equals the cost of inserting it when only the inserted elements before it were already in the table:

$$\overline{Q^+}(M,N) = \frac{1}{N}\sum_{i=0}^{N-1}\overline{Q^-}(M,i) = \frac{1}{N}\sum_{i=0}^{N-1} 1 + \frac{i}{M} = 1 + \frac{\alpha}{2} - \frac{1}{2M}$$

**The interest of hashing is that it is efficient and easy to program**



## Expected cost – interpretation

- if $N = O(M)$, then $\alpha = N/M = O(M)/M = O(1)$
- searching takes constant time on the average
- insertion is $O(1)$ in the worst case
- deletion takes $O(1)$ worst-case time for doubly linked lists
- hence, all dictionary operations take $O(1)$ time on average with hash tables with chaining



## Alternative proof for successful search

- $x_i$ is the $i^{\text{th}}$ element inserted into the table and $k_i = key[x_i]$
- $X_{ij} = \mathbf{1}\{h(k_i) = h(k_j)\}$ for all $i, j$ (indicator R.V.)
- simple uniform hashing: $Pr\{h(k_i) = h(k_j)\} = 1/M \Rightarrow E[X_{ij}] = 1/M$
- expected number of elements examined in a successful search:

$$E\left[\frac{1}{N}\sum_{i=1}^{N}\left(1 + \sum_{j=i+1}^{N} X_{ij}\right)\right] \qquad (1)$$

$\sum_{j=i+1}^{N} X_{ij} = \sharp$ of elements inserted after $x_i$ into the same slot as $x_i$.

$(1) = \frac{1}{N}\sum_{i=1}^{N}\left(1 + \sum_{j=i+1}^{N} E[X_{ij}]\right) = \frac{1}{N}\sum_{i=1}^{N}\left(1 + \sum_{j=i+1}^{N}\frac{1}{M}\right) =$
$1 + \frac{1}{NM}\sum_{i=1}^{N}(N-i) = 1 + \frac{1}{NM}\left(\sum_{i=1}^{N} N - \sum_{i=1}^{N} i\right) =$
$1 + \frac{1}{NM}\left(N^2 - \frac{N(N+1)}{2}\right) = 1 + \frac{N-1}{2M}$



## Another structure for Hash Table: Linear Probing

When the **number** of elements $N$ can be **estimated** in advance
You can **avoid** using any **linked list**
You **store** $N$ records in a table of size $M > N$
**Empty places** in the table **help you** for collision resolution
It is called the **linear probing**

## Searching and Inserting in Linear Probing

If the place $HashTable[hash(key)]$ is already busy

- If the keys match, the search is **successful**
- Else there is a **collision**

You search at the next place $i + 1$

- If the place is free, the search is **unsuccessful** and you have found **a place to insert** your record
- Else if the keys match, the search is **successful**
- If the keys differ **try the next position** $i + 2$
- But be careful the position after $i$ is $i + 1 \ mod \ M$
- And check that the table is not full otherwise the iteration won't terminate



## Problem with Linear Probing

Suppose you like to perform the operation of **suppression**
To **suppress** an element in the Hash Table, you search it, you remove it from the array and the place is free again. Is it so simple?
Suppose $key1$ and $key2$ (different) hash to the same address $i$

- you insert $key1$ first at position $i$
- you try to insert $key2$ at position $i$, you find it busy, and you finally insert it at position $i + 1$
- now you suppress $key1$. The place $i$ becomes free
- you search $key2$: it hashes at a free position $i$: its search is unsuccessful **but** $key2$ is in the table

A place may have three status: **free**, **busy** and **suppress**



## Example

*insert* : axe, chop, clip, cube, dice, fell, hack, hash, lop, slash

| an | bo | cp | dq | er | fs | gt | hu | iv | jw | kx | ly | mz |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *t* 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| ⊥ | ⊥ | ⊥ | ⊥ | axe | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ |
| ⊥ | ⊥ | chop | ⊥ | axe | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ |
| ⊥ | ⊥ | chop | clip | axe | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ |
| ⊥ | ⊥ | chop | clip | axe | cube | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ |
| ⊥ | ⊥ | chop | clip | axe | cube | dice | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ | ⊥ |
| ⊥ | ⊥ | chop | clip | axe | cube | dice | ⊥ | ⊥ | ⊥ | ⊥ | fell | ⊥ |
| ⊥ | ⊥ | chop | clip | axe | cube | dice | ⊥ | ⊥ | ⊥ | hack | fell | ⊥ |
| ⊥ | ⊥ | chop | clip | axe | cube | dice | hash | ⊥ | ⊥ | ⊥ | fell | ⊥ |
| ⊥ | ⊥ | chop | clip | axe | cube | dice | hash | lop | ⊥ | hack | fell | ⊥ |
| ⊥ | ⊥ | chop | clip | axe | cube | dice | hash | lop | slash | hack | fell | ⊥ |

*remove* clip

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ⊥ | ⊥ | chop | ~~clip~~ | axe | cube | dice | hash | lop | slash | hack | fell | ⊥ |
| ⊥ | ⊥ | chop | lop | axe | cube | dice | hash | ~~lop~~ | slash | hack | fell | ⊥ |
| ⊥ | ⊥ | chop | lop | axe | cube | dice | hash | slash | ~~slash~~ | hack | fell | ⊥ |
| ⊥ | ⊥ | chop | lop | axe | cube | dice | hash | slash | ⊥ | hack | fell | ⊥ |

**Fig. 4.2.** Hashing with linear probing. We have a table *t* with 13 entries storing synonyms of "(to) hash". The hash function maps the last character of the word to the integers 0..12 as indicated above the table: a and n are mapped to 0, b and o are mapped to 1, and so on. First, the words are inserted in alphabetical order. Then "clip" is removed. The figure shows the state changes of the table. Gray areas show the range that is scanned between the state changes



## Performances in Hash Table with linear probing

This hashing works because it guarantees that when you search for a particular key you look at every key that hashes to the same table address
In linear probing when the table begins to fill up, you also look to other keys: 2 different collision sets may be stuck together: **clustering problem**
Linear probing is very slow when tables are almost full because of the clustering problem
And when the table is full you cannot continue to use it

## Eliminating the Clustering Problem

Instead of examining each successive entry, we use a **second hash function** to compute a fixed increment to use for the sequence (instead of using 1 in linear probing)

Depending on the choice of the second hash function, the program may not work : obviously 0 leads to an infinite loop



## Hashing in Ruby

```
zip=Hash.new
zip={"06000" => "Nice", "06100" => "Nice", "06110" => "Le Cannet",
"06130" => "Grasse", "06140" => "Coursegoules", "06140" => "Tourrettes
sur Loup", "06140" => "Vence", "06190" => "Rocquebrune Cap Martin",
"06200" => "Nice", "06230" => "Saint Jean Cap Ferrat", "06230" =>
"Villefranche sur Mer"}

zip["06300"]="Nice" # adds a new entry
zip.keys=>["06140", "06130", "06230", "06110", "06000", "06100",
"06200", "06300", "06190"]

zip.values=>["Vence", "Grasse", "Villefranche sur Mer", "Le Cannet",
"Nice", "Nice", "Nice", "Nice", "Rocquebrune Cap Martin"]

zip.select { |key,val| val="Nice"}=>[["06000", "Nice"], ["06100",
"Nice"], ["06200", "Nice"], ["06300", "Nice"]]

zip.index "Nice" => "06000"

zip.each {|k,v| puts "#{k}/#{v}"}=>
06140/Vence
06130/Grasse
06230/Villefranche sur Mer
06110/Le Cannet
06000/Nice
06100/Nice
06200/Nice
06300/Nice
06190/Rocquebrune Cap Martin
```



## Conclusion on Hashing

Hashing is a classical problem in CS: various algorithms have been studied and are widely used

There are many empirical and analytic results that make utility of Hashing evident for a broad variety of applications

**Hashing is prefered to binary tree searches** for many applications because it is **simple to implement** and can provide **very fast constant searching times** when space is available for a large enough table