Hashing
Collision Resolution

D.E ZEGOUR
École Supérieure d'Informatique
ESI

Hashing

Introduction

two ways to organize data that arrive
in any order into a table.

Keep the array ordered

Insertion with shifting (O(n))

Fast search (Binary search: O(Log(n))

Keep the array non ordered

Insertion at the end (O(1))

Slow search (Linear search: O(n))

To quickly search (O(Log(n)), one would need to sort slowly (O(n)).
To sort quickly (O(1)), one would need to search slowly (O(n)).

Hashing

Introduction

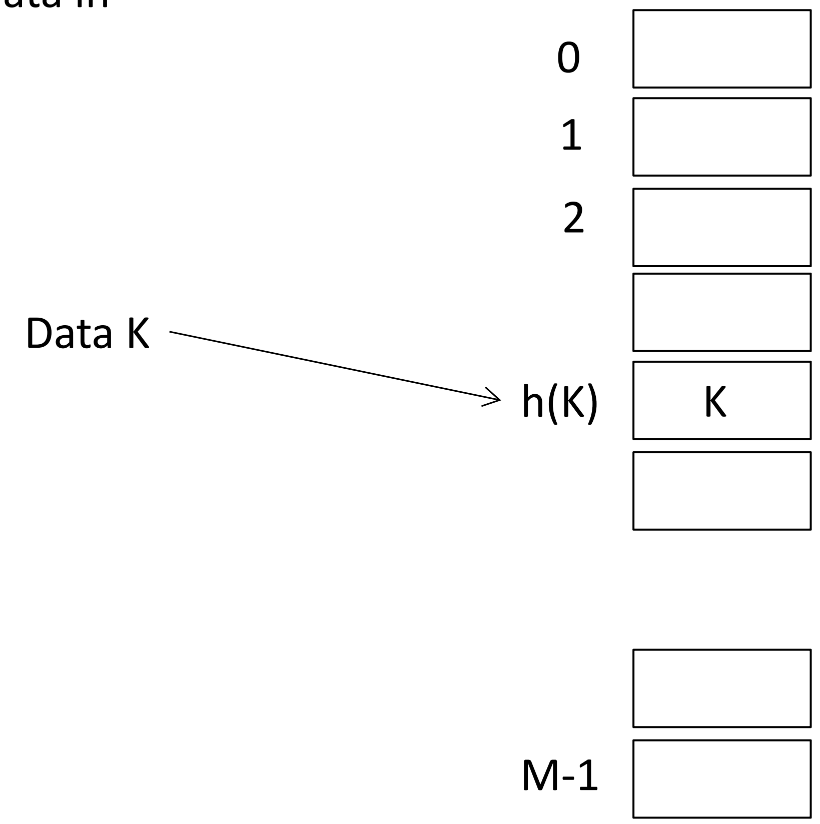
A third possibility for organizing data in an array

Data K is stored at a location calculated by a function h.

Fast search and insertion (O(1)).

Problem: Determination of a bijective function

Bijection: assigns a new location in the array to each data.



Hashing

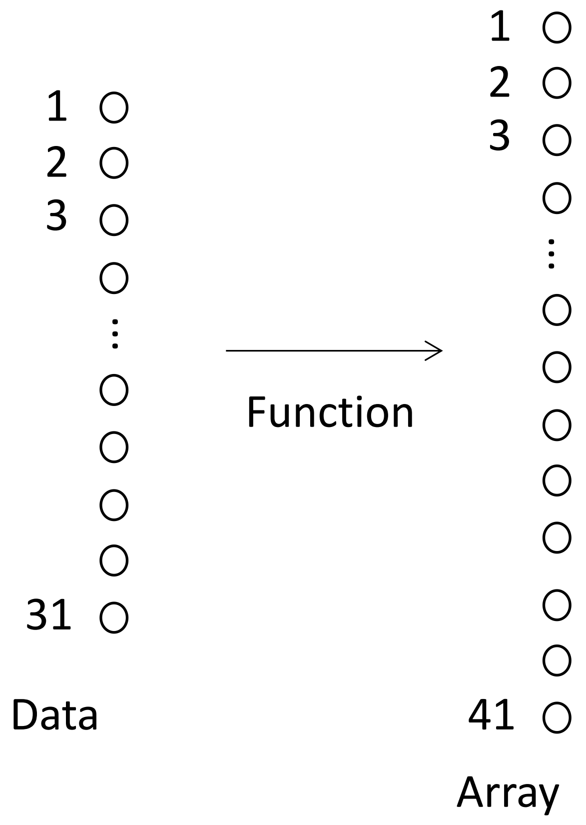
Introduction

Not easy to discover a bijective function

There are $41^{31} \approx 10^{50}$ possible functions mapping a set of 31 elements to a set of 41 elements.

Out of these, only $41! / 10! = 10^{43}$ functions have distinct values for each argument. (bijective functions)

Approximately one function in 10 million meets this criterion. ($10^{43} / 10^{50}$)



Hashing

Introduction

Not easy to discover a bijective function

Birthday Paradox:

<< If there are 23 or more people present in a room, the chances are high that at least two of them share the same day and month of birth.>

Functions from a set of 23 elements to a set of 365 elements

Calculate ...

Probability = 0.4927

Hashing

Introduction

This class of algorithms is referred to as Hashing or the scatter arrangement technique.

There will always be distinct data K1 and K2 for which f(K1) = f(K2).
This scenario is known as a collision.

In conclusion, to use a hashing technique, we must define the following:
- A hash function
- A collision resolution method.

Hashing

Terminology

Data that share the same image under the hash function are referred to as synonyms.
K1, K2, ..., Kn are synonyms if $h(K1) = h(K2) = ... = h(Kn)$

The primary address of a data is determined by the function f(data).

Any data not located at its primary address is referred to as overflow.
It is also described as being stored at a secondary address.

Hashing

Hashing Functions

The goal is to discover a function f such that:
$0 \leq f(K) < M$
to minimize the occurrence of collisions
(K is the data to hash and M is the table size)

Ideally, we aim for f to be bijective.
The worst-case scenario arises when all data are hashed to a single address.

An acceptable solution is one where some data share the same address (f is surjective).

Hashing

Example of Hashing function

1. Represent the data in numerical form.

2. Concatenate and sum (Folk and Add).

3. Divide the result by a prime number and use the remainder as the address.

Transformation of the word ALGORITHM

a) 65 76 71 79 82 73 84 72 77 32

b)

6576	+	7179	= 13,755 Mod 20000	= 13,755
13755	+	8273	= 22,028 Mod 20000	= 2,028
2028	+	8472	= 10,500 Mod 20000	= 10,500
10500	+	7732	= 18,232 Mod 20000	= 18,232

c) a= 18,232 mod 101 = 52.

Modulo 20,000 is employed to prevent any overflow.

Hashing

Examples of Hashing functions

Modulo
h(K)= K mod M

M : table size
Good choice : M prime number

Example:
h(453) = 53

table size : 101

Middle square

Square the data and extract the middle numbers.

Example:

$(453)^2 = 205209$
h(453) = 52

table size : 100

favorable results when there are no zeros in the squared number

Radix
The data is converted into a specific number base, and we calculate the remainder of the division of the transformed data by the size of the table.

Example :

$453 = (382)_{11}$ (en base 11)
382 Mod 100 = 85
h(453) = 85

table size : 100

Hashing

Collision Resolution

There are various methods for handling collisions

The most classical methods:

1. Linear probing
2. Double hashing
3. Internal chaining
4. External chaining or separated chaining

Hashing function used : Modulo

Hashing

Linear probing

If a collision occurs in cell I of the array T[0..M-1], we insert the data into the first available cell within the cyclic sequence:

I-1, ..., 0, M-1, M-2, ..., I+1

In essence, we perform a linear search for an available cell within the mentioned sequence, hence the name of the method.

Hashing

Linear probing

Inserting the following data along with their transformations (in parentheses): a(3), b(2), c(3), d(2), e(1) into a table T with 6 elements

- Inserting a(3)
- Inserting b(2)
- Inserting c (3)
- Inserting d (2)
- Inserting e(1)

0	d
1	c
2	b
3	a
4	
5	e

Hashing

Linear probing

Algorithm :
- Search for data K in the table T[0..M-1] of M elements.
- If K is not found and the table is not fulll, data k is inserted

A static variable is used : N
Number of data inserted.

Table is considered filled when N = M - 1, not when N = M

L1. [Hash]
i := h(K) { 0 ≤ i < M }

L2. [Compare]
IF Data(i) = K, the algorithm terminates successfully.
Otherwise, IF T(i) is empty, go to L4.

L3. [Advance to next]
i := i - 1
IF i < 0: i := i + M
GO TO L2.

L4. [Insert] {search is unsuccessful}
IF N = M - 1
  The algorithm ends with overflow
ELSE
  N := N + 1
  Mark T(i) as occupied
  Data(i) := K

Hashing

Double hashing

This method is quite similar to the previous one

In other words, when a collision occurs at cell I, a step p is calculated using another hash function, and the cyclic sequence to be consulted would be I-p, I-2p, and so on.

Two hashing functions are used h(K) et h'(K). Hence the name of the method.

The choice of M holds significant importance, as an incorrect choice can result in the incomplete coverage of the set of possible addresses

We demonstrate that when M is a prime number, and the hash function is random, it provides full coverage of the entire set of addresses.

Hashing

Double hashing

Inserting a(3), b(2), c(3), d(2), e(1)
with h'(c) = 3 ; h'(d) = 1 ; h'(e) = 3 (h' is the second hashing function) into a table T of 6 elements

- Inserting a(3)
- Inserting b(2)
- Inserting c (3)
- Inserting d (2)
- Inserting e(1)

0	c
1	d
2	b
3	a
4	e
5	

Hashing

Double hashing

Algorithm :
- Search for data K in the table T[0..M-1] of M elements.
- If K is not found and the table is not fulll, data k is inserted

A static variable is used : N
Number of data inserted.

Table T is considered filled when N = M - 1, not when N = M

D1. [First hashing]
i := h(K)

D2. [First test]
IF T(i) is empty THEN GOTO D6.
IF Data(i) = K, the algorithm ends successfully.

D3. [Second hashing]
c := h'(K)

D4. [Advance to next]
i := i - c ; IF i < 0 THEN i := i + M

D5. [Compare]
IF T(i) is empty THEN GOTO D6.
IF Data(i) = K, the algorithm ends successfully.
OTHERWISE GOTO D4

D6. [Insert]
IF N = M - 1 THEN "overflow".
OTHERWISE
  N := N + 1
  Make T(i) occupied
  Data(i) := K

Hashing

Internal Chaining

Synonyms are organized into a linked list represented within the table. This method is aptly named.

When a collision occurs at cell K, we navigate through the linked list that starts at K

If the data is not found, search for an empty location in the table. This location will be added to the linked list.

Strategy : search for an empty position from the end

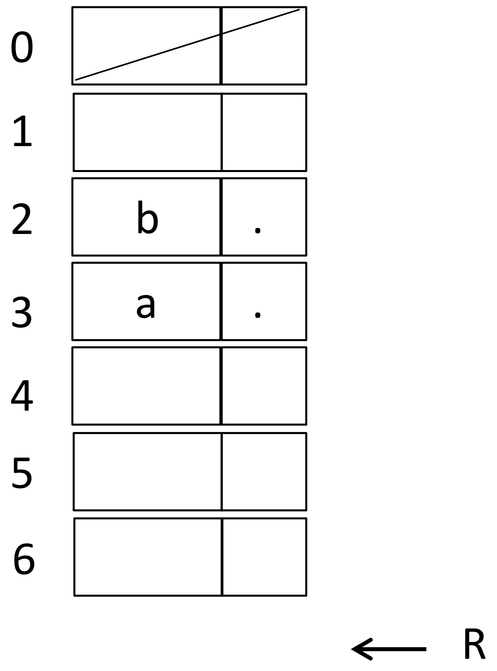
Importante Remark :
A linked list contains groups of synonyms.

Hashing

Internal Chaining

Inserting a(3), b(2), c(3), d(2), e(1), f(6) into
a table T of 6 elements

- Inserting a(3)
- Inserting b(2)

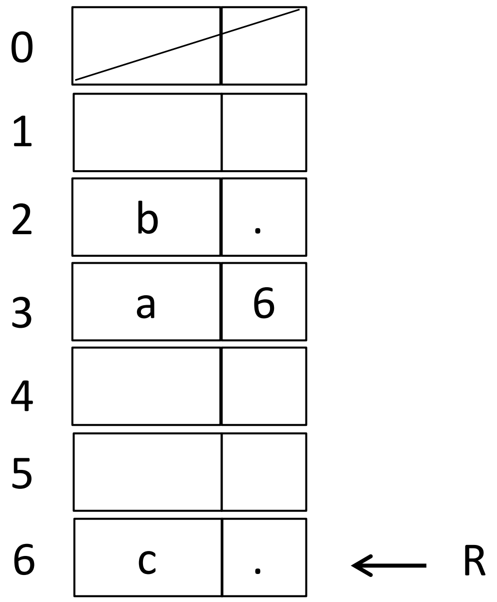


Hashing

Internal Chaining

Inserting a(3), b(2), c(3), d(2), e(1), f(6)
into a table T of 6 elements

- Inserting a(3)
- Inserting b(2)
- Inserting c (3)

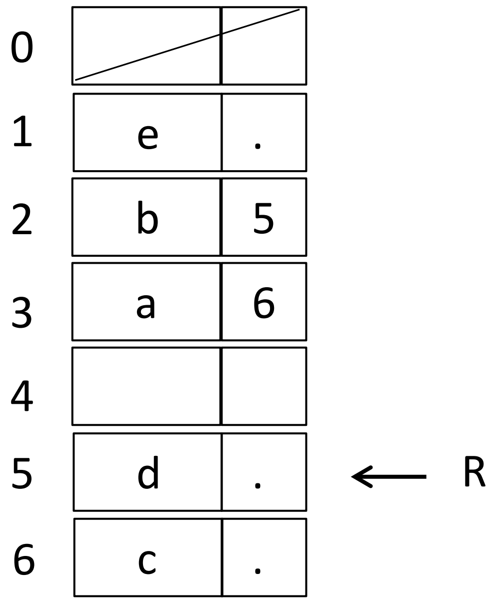


Hashing

Internal Chaining

Inserting a(3), b(2), c(3), d(2), e(1), f(6) into
a table T of 6 elements

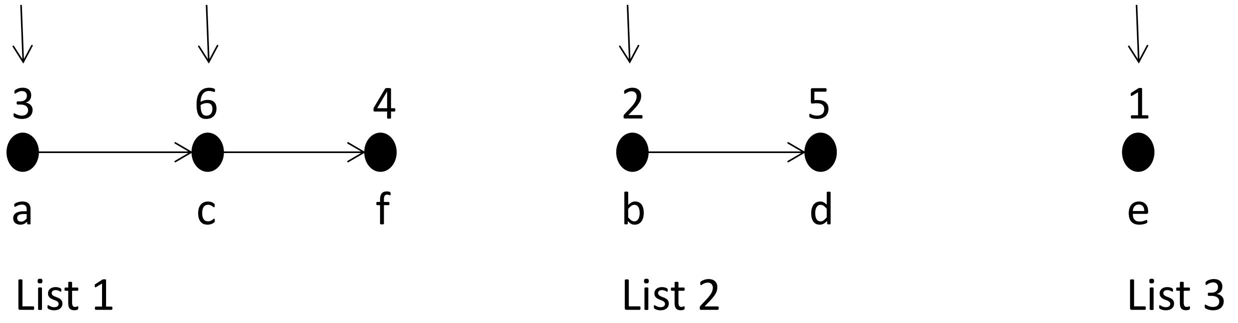
- Inserting   a(3)
- Inserting   b(2)
- Inserting   c (3)
- Inserting   d (2)
- Inserting   e(1)



Hashing

Internal Chaining

Inserting a(3), b(2), c(3), d(2), e(1), f(6)
into a table T of 6 éléments



- Inserting a(3)
- Inserting b(2)
- Inserting c (3)
- Inserting d (2)
- Inserting e(1)
- Inserting f(6)

0			
1	e	.	
2	b	5	
3	a	6	
4	f	.	⟵ R
5	d	.	
6	c	4	

Hashing

Internal Chaining

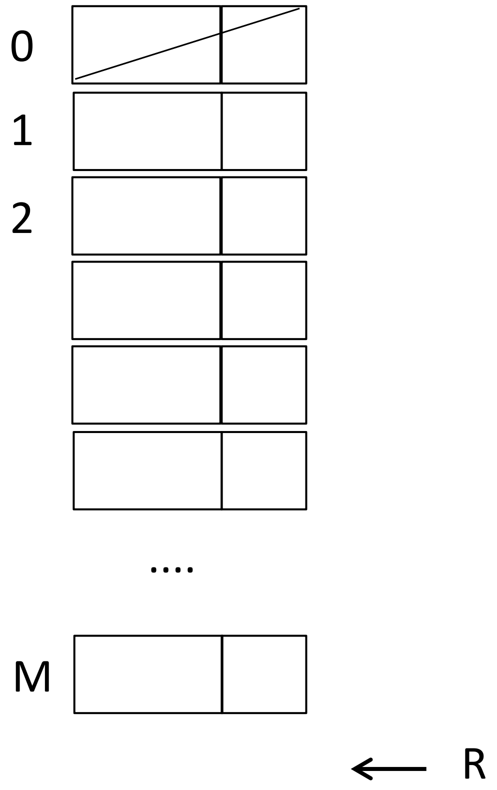
Algorithm :
- Search for data K in the table T[0..M] of M elements.
- If K is not found and the table is not fulll, data k is inserted

Element = 2 fields : Data and Link

An auxiliary variable R is utilized to aid in identifying empty spaces. When the table is empty, R equals M

After several insertions, we have : T(j) occupied for all j such that R ≤ j ≤ M.

By convention T(0) is not used (always empty)



Hashing

Internal Chaining

Algorithm :
- Search for data K in the table T[0..M] of M elements.
- If K is not found and the table is not fulll, data k is inserted

Element = 2 fields : Data and Link

An auxiliary variable R is utilized to aid in identifying empty spaces. When the table is empty, R equals M

After several insertions, we have : T(j) occupied for all j such that R ≤ j ≤ M.

By convention T(0) is not used (always empty

C1. [Hash]
i := h(K) + 1 { so 1 ≤ i ≤ M }
C2. [Does a list exist?]
IF T(i) is empty THEN GOTO C6
{ otherwise T(i) is occupied; then we consult the list of occupied chains }
C3. [Compare]
IF K = DATA(i), the algorithm ends successfully.
C4. [Advance to next]
IF LINK(i) <> 0 THEN
 i := LINK(i) ; GOTO C3
C5. [Find an empty cell]
{ The search is unsuccessful, and we want to find an empty position in the table }
Decrement R one or more times until T(R) is empty.
IF R = 0 THEN the algorithm terminates with overflow.
Otherwise, do:
 LINK(i) := R ; i := R

C6. [Insert the new data]
Make T(i) Occupied with:
DATA(i) := K
LINK(i) := 0

Hashing

Separate Chaining

Synonyms are stored in a separate linked list, which is why this method is named as such.

A linked list holds only one group of Synonyms.

When a collision occurs at position i (i = h(k)) in the array T[0..M-1], we traverse the list starting at h(k). If the data is not found, we insert the data into the list (at the beginning or at the end).

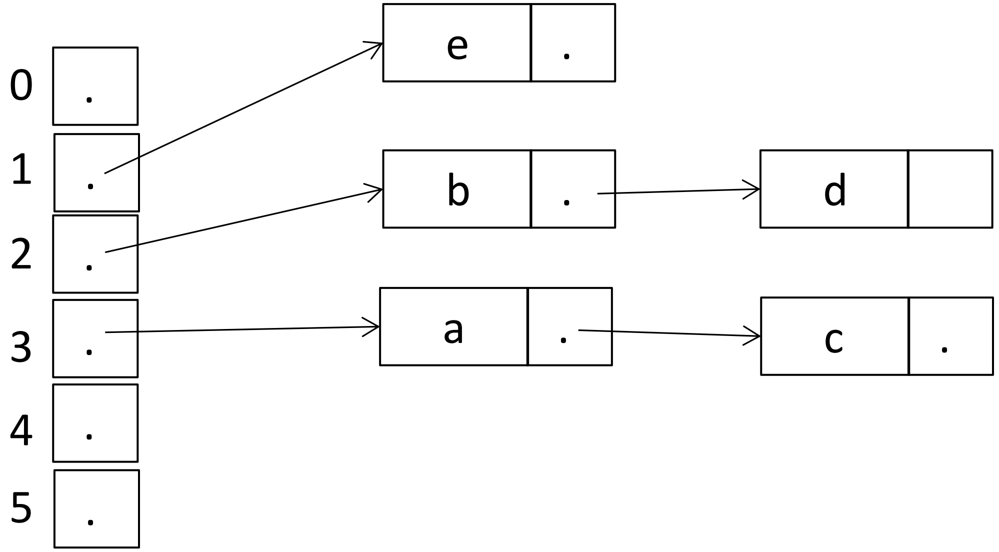
It is possible to store more elements than the size of the array.

Hashing

Separate Chaining

Inserting a(3), b(2), c(3), d(2), e(1) into a table T of 6 elements

- Inserting a(3)
- Inserting b(2)
- Inserting c(3)
- Inserting d 2)
- Inserting e(1)



Hashing

Separate Chaining

Algorithm:

- Search for a data K in the table T[1..M]. If K is not found in the corresponding linked list, the data is inserted.

- An element T(i) holds the list of synonyms.

- Initially, T(i) := Nil
for all i in the interval [0..M-1].

S1. [Hash]
i := h(K)

S2. [Is there a list?]
IF T(i) is empty THEN GOTO S5
{ in other cases, T(i) is occupied, and we then consult the list of occupied chains }
P := T(i)

S3. [Compare]
IF K = DATA(p) THEN
  the algorithm ends successfully.

S4. [Advance to next]
IF LINK(p) <> Nil THEN
  P := LINK(P)
  GOTO S3

S5. [Insert new data]
Allocate a cell, denoted as Q.
DATA(Q) := K
LINK(Q) := T(i)
T(i) := Q

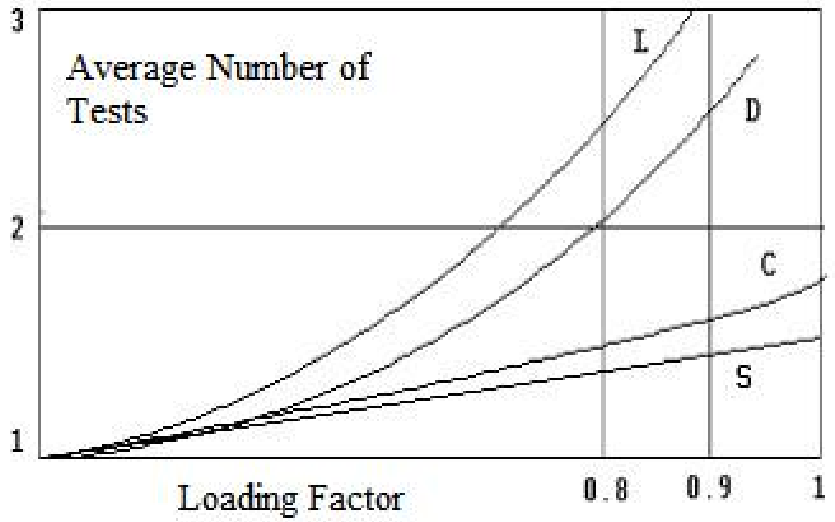
Hashing

Comparison between the Different Method

Curves representing the average number of tests for data search compared to the array loading factor .

(Loading factor = N/M, where N is the number of elements present in the array, and M is the size of the array)

- L denotes linear probing,
- D denotes double hashing,
- C denotes internal chaining, and
- S denotes separate chainin



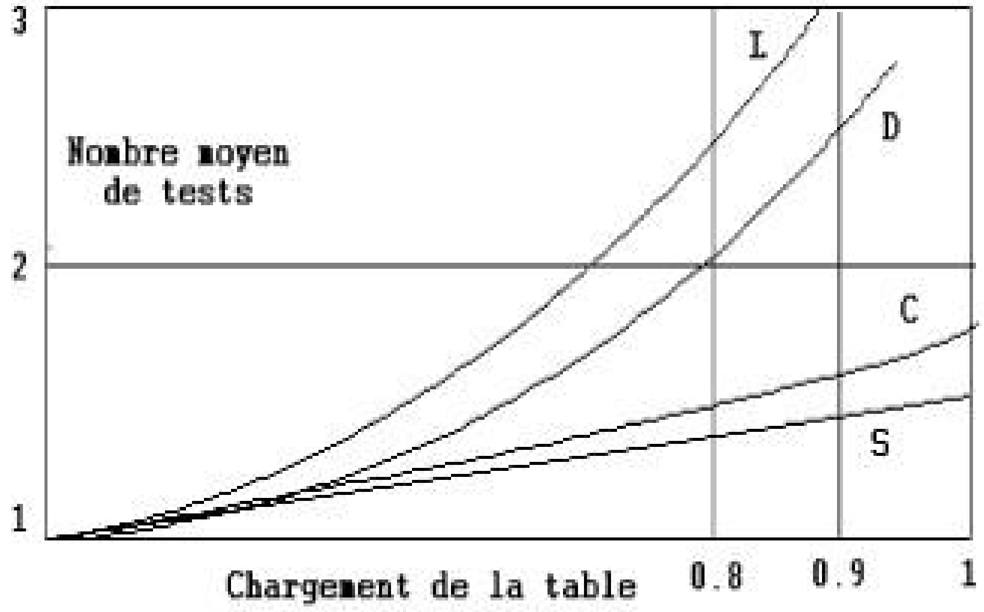
Hashing

Comparison between the Different Method

S > C > D > L

Chaining methods appear to be the most efficient

For a 70-80% loading factor --> O(1).



Hashing

Synthesis

Advantage : Very fast access to the l'information (O(1))

Drawbacks:
- Lack of order
- Limitation to static data

Usage
- Data insertion with load factor control (setting a threshold).
- Good compromise: loading from 70 to 80%.

Re hashing
- In case of table overload (Size increases to 2M).
- In case of table underload (Size decreases to M/2).

Application : dictionnary

Generalisation: More than one data per table cell.