



Tema 6 Estructures no lineals: Taules de hash

Maria Salamó Llorente
Estructura de Dades

Enginyeria Informàtica
Facultat de Matemàtiques i Informàtica,
Universitat de Barcelona



Contingut

6.0 Introducció

6.1 Conjunt

6.2 Mapa

6.3 Diccionari

6.4 Taules de hash (Taules de dispersió)

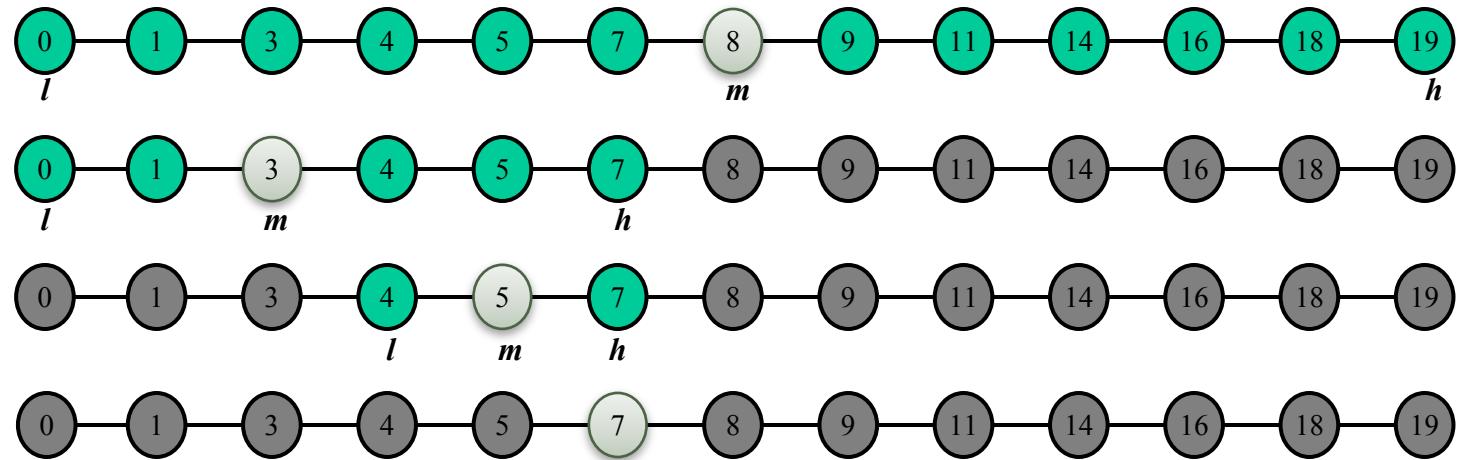
6.4.1 Introducció a Taules de Hash

6.4.2 Funcions de hash

6.4.3 Organització del hash

El tema s'explicarà en dues sessions de teoria
(Teo 12 i Teo13)

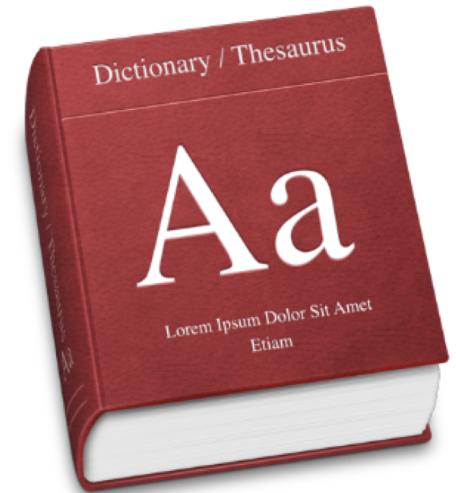
6.3 Diccionari





Diccionari

- Un diccionari està dissenyat per emmagatzemar ítems compostos per (*key*, *value*), on la *key* és utilitzada per trobar els *values* de l'element corresponent
- És molt similar a un *map*
- Aplicacions:
 - Llibreta d'adreces (nom → adreça)
 - Diccionari (paraula → definició)
 - Clients en un banc





Diccionari



- Un mapa modela la cerca en una col·lecció d'entrades (clau-valor)
- Les principals operacions d'un mapa són: **cercar, inserir i esborrar** ítems
- Múltiples entrades amb la mateixa clau **sí** es permeten
- Aplicacions:
 - Parells de (paraula – definició)
 - Autoritzacions de targetes de crèdit
 - Mapeig de DNS dels noms de host
 - (Ex. datastructures.net) a l'adreça IP (Ex., 128.148.34.101)



TAD Entry

- Una *entry* guarda un parell clau-valor (k, v)
- Mètodes:
 - `key()`: retorna la clau associada
 - `value()`: retorna el valor associat
 - `setKey(k)`: assigna la clau k
 - `setValue(v)`: assigna el valor v



TAD Diccionari: Operacions

- **find(k)** : si existeix una entry amb clau k, retorna un iterador a ella; sinó, retorna un iterador especial **end**
- **findAll(k)** : retorna iteradors b i e de tal manera que totes les entrades amb clau k estan en el rang de l'iterador [b, e) començant amb b i acabant just abans de e
- **put(k, o)** : inserta i retorna un iterador a ell
- **erase(k)** : elimina una entry amb clau k, error si no existeix
- **begin()** , **end()** : retorna iteradors a l'inici i al final del diccionari
- **size()** , **empty()**



Exemple

Operació	Sortida	Diccionari
put(5,A)	(5,A)	(5,A)
put(7,B)	(7,B)	(5,A),(7,B)
put(2,C)	(2,C)	(5,A),(7,B),(2,C)
put(8,D)	(8,D)	(5,A),(7,B),(2,C),(8,D)
put(2,E)	(2,E)	(5,A),(7,B),(2,C),(8,D),(2,E)
find(7)	(7,B)	(5,A),(7,B),(2,C),(8,D),(2,E)
find(4)	end	(5,A),(7,B),(2,C),(8,D),(2,E)
find(2)	(2,C)	(5,A),(7,B),(2,C),(8,D),(2,E)
findAll(2)	(2,C),(2,E)	(5,A),(7,B),(2,C),(8,D),(2,E)
size()	5	(5,A),(7,B),(2,C),(8,D),(2,E)
erase(5)	—	(7,B),(2,C),(8,D),(2,E)
find(5)	end	(7,B),(2,C),(8,D),(2,E)



Objectiu

- **put(key, val)**: afegeix un ítem (*key, value*) al diccionari
- **V find(key)**: retorna la *value* mapejada per la *key* corresponent
- **erase(key)**: elimina l'ítem corresponent a la *key* del diccionari
- **int size()**: retorna el nombre d'ítems que conté el diccionari
- **boolean isEmpty()**: comprova si el diccionari està buit

Implementar un map o un diccionari on totes aquestes funcions tinguin complexitat O(1)



Com implementar un diccionari

Tenim diferents estructures de dades que poden servir per implementar un diccionari.

- Array, LinkedList
- Arbre binari
- Arbre AVL
- **Taula de Hash**



Perquè volem fer hash?

Fer una cerca en aquestes estructures comporta un cost computacional de ...

- **Array, LinkedList** → $O(n)$
- **Arbre binari** → $O(\log n)$ però pot degenerar a $O(n)$
- **Arbre AVL** → $O(\log n)$ perquè balanceja l'arbre



Perquè volem fer hash?

Imagina que volem inserir 10.000 estudiants (amb un ID de 5 díigits) en aquestes estructures i fer operacions d'inserir, cercar i eliminar, el cost computacional serà de ...

- **LinkedList** → $O(n)$
- **Arbre binari** → $O(\log n)$ però pot degenerar a $O(n)$
- **Arbre AVL** → $O(\log n)$ perquè balanceja l'arbre
- Usant **un array** de 100.000 posicions, podem tenir un accés de **$O(1)$** , però estarem ocupant molt d'espai que no usarem.



Perquè volem fer hash?

Hi ha alguna manera d'aconseguir un accés amb un cost computacional de $O(1)$ i que no malgastem tant d'espai?



Perquè volem fer hash?

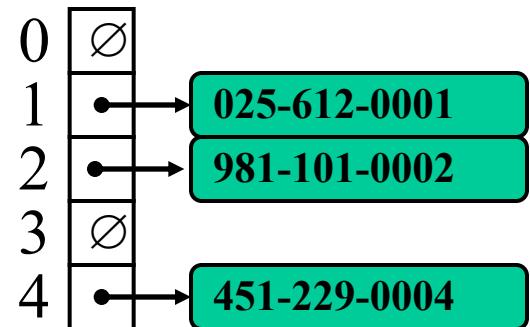
Hi ha alguna manera d'aconseguir un accés amb un cost computacional de $O(1)$ i que no malgastem tant d'espai?

La resposta és ... **TAULES DE HASH**



6.4 Taules de Hash

6.4.1 Introducció a Taules de Hash





Introducció a Taules de Hash

- **Permeten l'accés directe a un element de la seqüència, indicant la posició que ocupen**
- Les taules de hash tenen com a finalitat que la **cerca, la inserció i la eliminació** d'una entry es faci amb un **cost computacional constant**, és a dir, $O(1)$
- Una taula hash es sol usar per a la **implementació eficient d'un mapa o d'un diccionari**
- Les taules hash estan construïdes mitjançant arrays
 - L'organització ideal d'una taula de hash és que el camp clau de la entry es correspongui directament amb l'índex de l'array

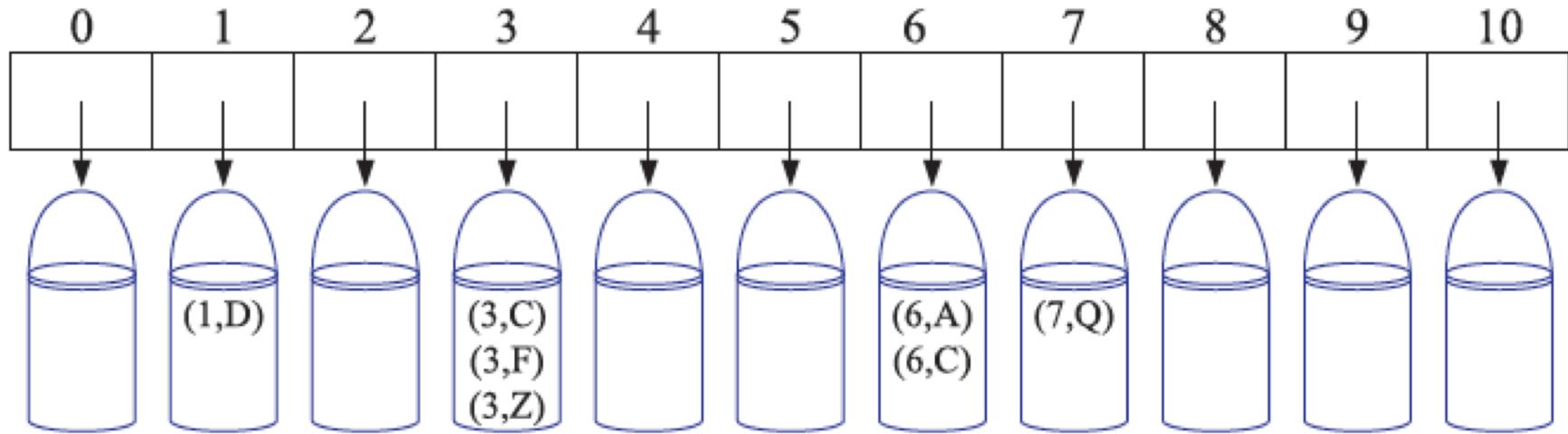


Introducció a Taules de Hash

- Per convertir el camp clau d'una entry en un índex dins del rang de definició de l'array s'utilitza una **funció de hash h**
- És possible que diferents keys vagin a parar al mateix índex de la taula
 - Dues claus diferents x_1, x_2 obtenen el mateix índex, és a dir que $h(x_1) = h(x_2)$
 - Això s'anomena **col·lisió**
 - **Com es pot guardar múltiples *values* de diferents key en un únic índex?**

Introducció a Taules de Hash

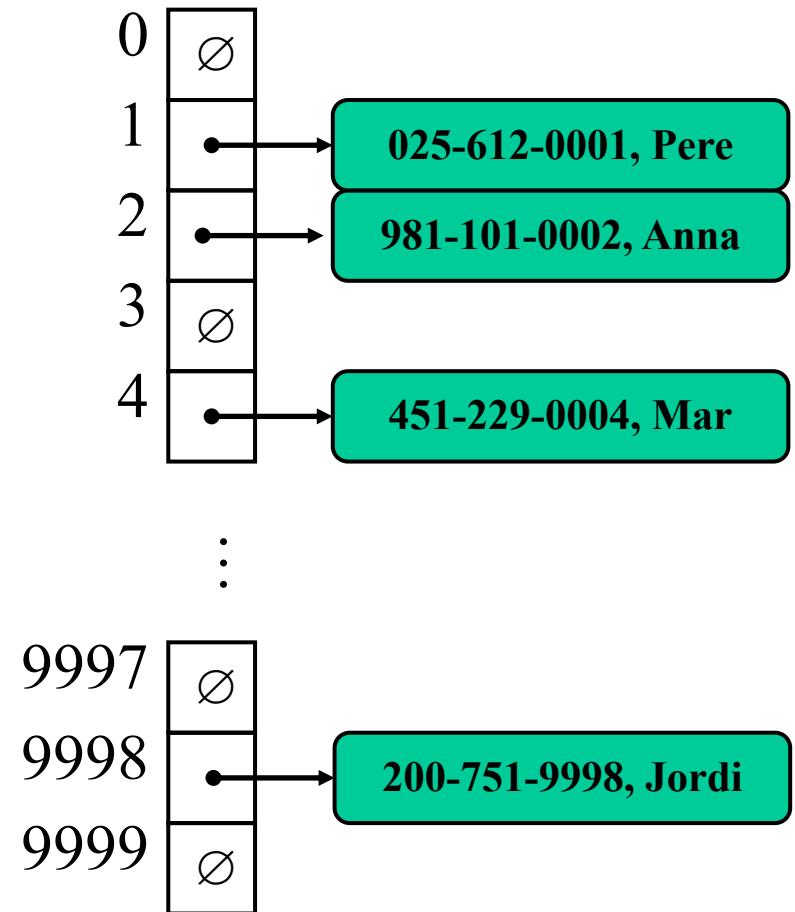
- Una taula de hash per a un tipus de clau consisteix en:
 - Una funció hash h
 - Un array (taula o *bucket array* en anglès) de mida N



- Quan s'implementa un mapa amb una taula de hash, l'objectiu és guardar el ítem (k, o) en l'índex $i = h(k)$

Exemple

- Dissenyem una taula de hash per a un mapa que guarda les entrades com **(SSN, Nom)**, on SSN (número de la seguretat social) és un enter positiu de 9 díigits
- La nostra taula de hash usa un array de mida $N = 10,000$ i la funció de hash és:
$$h(x) = \text{últims quatre díigits of } x$$



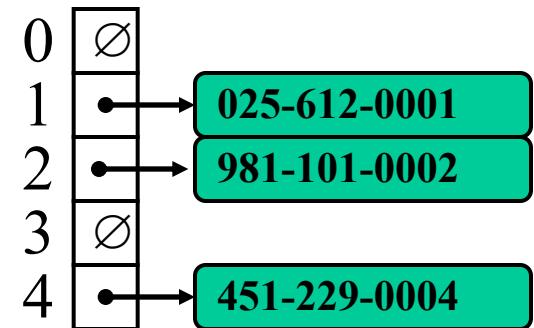


Introducció a Taules de Hash

- Les taules de hash es dissenyen considerant les col·lisions
- Es reserven més posicions de memòria, N, que elements a guardar, m.
- Quantes més posicions, menor risc de col·lisions si la funció de hash és bona, però més memòria estarà desaprofitada, ja que hi ha més forats buits a la taula.
- El **factor de càrrega**, λ mesura la fracció de la taula que està plena. El valor del factor de càrrega està en el rang [0 .. 1], on 0 indica que està buida i 1 que està completament plena.
 - $\lambda = \frac{m}{N}$, m és el número d'elements guardats i N la mida de la taula
 - Normalment es busca que el factor de càrrega sigui $\lambda \leq 0.8$



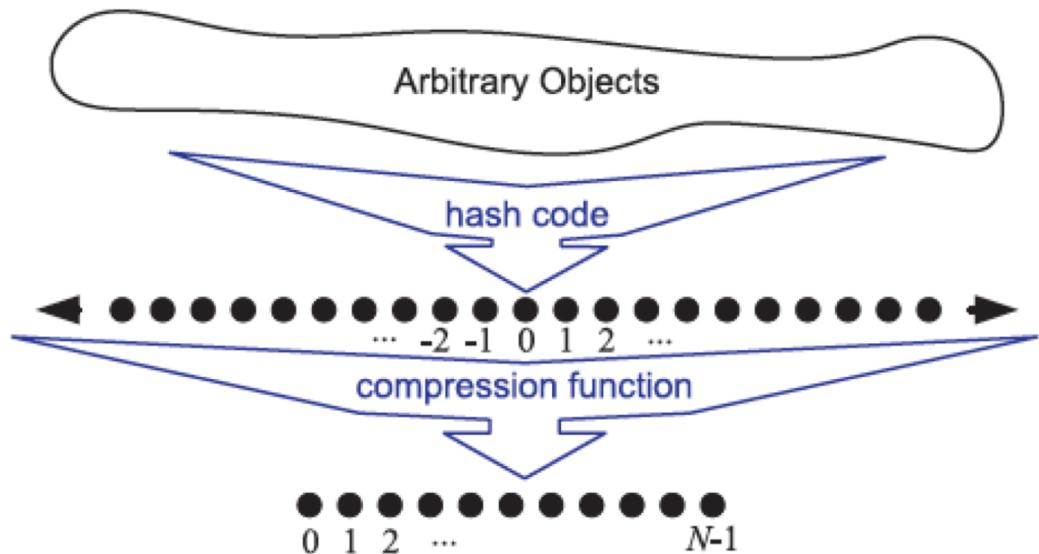
6.4.2 Funcions de Hash



Funcions de hash



- La tasca de la **funció de hash** $h(\text{key})$ és que a partir una clau (**key**) retorna un índex dins de l'array, on els valors (**values**) de la key corresponents estan emmagatzemats
- $h: K \rightarrow N$ on K és el conjunt de claus i N les posicions de memòria
- Per tota $x \rightarrow h(x)$, per tota clau x , $h(x)$ es denomina **valor hash** de la clau x . A més a més, és l'índex de la taula





Funcions de Hash

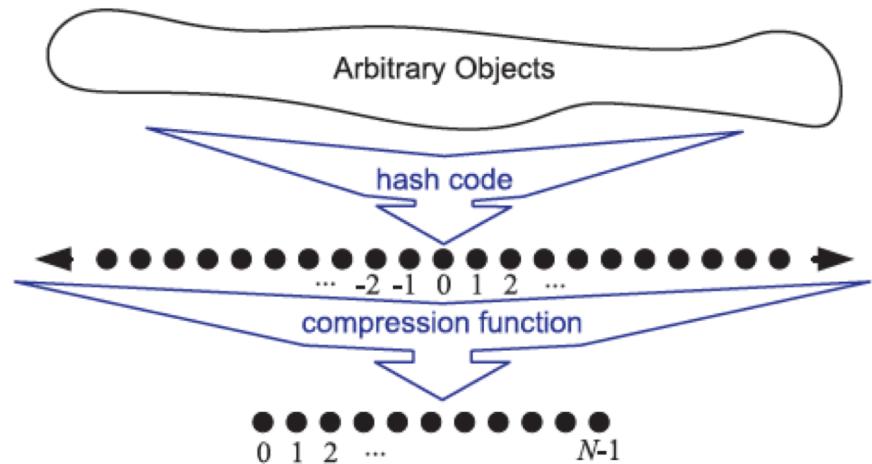
- Una funció de hash normalment s'especifica com la composició de dues funcions:

Codi Hash:

h_1 : keys \rightarrow integers

Funció de compressió:

h_2 : integers $\rightarrow [0, N - 1]$



- El codi hash s'aplica primer i la funció de compressió s'aplica després amb el resultat del codi hash,
i.e., $h(x) = h_2(h_1(x))$



Criteris per seleccionar una Funció de Hash



- L'objectiu de la funció de hash és “dispersar” les claus d'una manera aparentment random

Els **criteris** per seleccionar una funció de hash són:

1. $h(x)$ ha de ser **fàcil d'avaluar** i que el seu temps d'execució sigui mínim, de complexitat constant, $O(1)$
2. Ha de **distribuir uniformement els valors de hash** sobre el conjunt N , per així minimitzar les col·lisions



Codi hash (I)



□ Adreça de memòria

- Es reinterpreta l'adreça de memòria de la clau com un enter
- Bona en general, excepte per claus numèriques o strings

□ Conversió a enter

- Es reinterpreten els bits de la clau com un enter
 - S'assumeix que el nombre de bits de cada tipus és conegut
 - A C++ es pot fer un include <limits>
 - Existeix una classe `numeric_limits` que permet donat un tipus T (com un `char` o `int`) es pot conèixer quin és el nombre de bits d'una variable de tipus T a partir de `numeric_limits<T>.digits`
- Adequada per claus de mida menor o igual al nombre de bits del tipus enter (e.g., `byte`, `short`, `int` i `float` en C++)
- En el cas de `long` s'hauria de fer un cast-down a un enter
 - Això suposa una pèrdua d'informació del valor original de la clau
 - Millor en aquest cas fer una suma de la part alta i baixa del `long`.



Codi hash (II)



- Suma de components

- En el cas del long la suma de components és millor opció que només fer el cast de la part baixa per convertir-lo a int.
- La suma de components es pot aplicar a qualsevol objecte que la seva representació es pugui dividir en parts (x_0, x_1, \dots, x_{k-1}) d'enters
- Es parteixen els bits en components de mida fixa (e.g., 16 or 32 bits) i es sumen els components (ignorant overflows)

$$\sum_{i=0}^{k-1} x_i$$

- Adequada per claus numèriques de mida fixa major o igual al nombre de bits del tipus enter (e.g., long i double en C++)
- No és adequada per cadenes de caràcters o strings



Codi hash (II)



- Suma de components
 - No és adequada per cadenes de caràcters o strings
 - Per exemple, les paraules:
 - "STOP"
 - "POTS"
 - "TOPS"
 - "SPOT"
 - La solució passa per usar un polinomi

Totes coincideixen en la
suma de components



Codi hash (III)

- Acumulació polinomial:

- Es partitionen els bits de la clau en una seqüència de components de mida fixa (e.g., 8, 16 or 32 bits)

$$a_0 \ a_1 \ \dots \ a_{n-1}$$

- S'avalua el polinomi

$$p(z) = a_0 + a_1 z + a_2 z^2 + \dots + a_{n-1} z^{n-1}$$

ó

$$p(z) = a_0 z^{n-1} + a_1 z^{n-2} + \dots + a_{n-2} z + a_{n-1}$$

amb un valor fixe z , ignorant overflows

- Bons valors per a z són 33, 37, 29 o 41
 - Especialment adequada per strings o cadenes de caràcters
 - Per exemple, escollint $z = 33$, en un conjunt de 50.000 paraules en anglès, es generen com a molt 6 col·lisions



Codi hash (III)

- Acumulació polinomial:

$$p(z) = a_0 + a_1 z + a_2 z^2 + \dots + a_{n-1} z^{n-1}$$

$$p(z) = a_0 z^{n-1} + a_1 z^{n-2} + \dots + a_{n-2} z + a_{n-1}$$

- El polinomi $p(z)$ es pot avaluar en un temps $O(n)$ usant la **regla de Horner**:
 - Els polinomis es calculen successivament, cadascú a partir de l'anterior en un temps $O(1)$
 - Tenim $p(z) = p_{n-1}(z)$

$$p_0(z) = a_{n-1}$$

$$p_i(z) = a_{n-i-1} + z p_{i-1}(z) \quad (i = 1, 2, \dots, n-1)$$



Funcions de compressió

- Divisió o Aritmètica modular:

$$h_2(y) = y \bmod N$$

- La mida N de la taula de hash normalment s'escull un nombre primer
 - N primer més gran però més proper al nombre d'elements que es volen guardar a la taula
- No és recomanable escollir una mida N que sigui múltiple de 2 o de 10
 - Si $N = 2^j$ o 10^j la distribució de $h(x)$ es basa únicament en els j díigits menys significatius
 - Per ex., $N=100 = 10^2$, $h(628) = h(228) = h(128) = 28$



Funcions de compressió



- Divisió o Aritmètica modular:

$$h_2(y) = y \bmod N$$

- **Exemple:**

- Volem guardar $N=900$ registres
 - Escolliu la mida de la taula i calcular la posició que ocupa cadascuna d'aquestes claus:

- 245643 245981 257135

- Solució:

- Un bon valor és $N = \underline{\hspace{2cm}}$

- $h(245643) = 245643 \bmod \underline{\hspace{2cm}} = \underline{\hspace{2cm}}$

- $h(245981) = 245981 \bmod \underline{\hspace{2cm}} = \underline{\hspace{2cm}}$

- $h(257135) = 257135 \bmod \underline{\hspace{2cm}} = \underline{\hspace{2cm}}$



Funcions de compressió



- Divisió o Aritmètica modular

$$h_2(y) = y \bmod N$$

- Exemple:

- Volem guardar $N=900$ registres
 - Escolliu la mida de la taula i calcular la posició que ocupa cadascuna d'aquestes claus:

- 245643 245981 257135

- Solució:

- Un bon valor és $N = 997$
 - $h(245643) = 245643 \bmod 997 = 381$
 - $h(245981) = 245981 \bmod 997 = 719$
 - $h(257135) = 257135 \bmod 997 = 906$



Funcions de compressió



- Multiplica, Afegeix i Divideix (MAD)

$$h_2(y) = (ay + b) \bmod N$$

- a and b són enters no negatius tal que $a \bmod N \neq 0$
- Altrament, cada enter mapejaria al mateix valor b

- Plegament

- Consisteix en partir les claus en parts $x_1, x_2, x_3, \dots, x_n$ y combinar les parts (per exemple sumant)
- Per exemple: 245643 245981 257135 si els dividim en dos parts

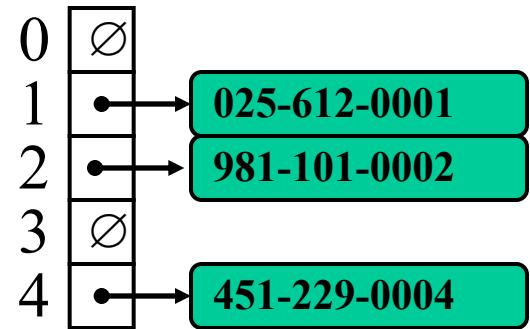
- $h(245643) = 245 + 643 = 888$

- $h(245981) = 245 + 981 = 1226 = 226$ (s'ignora overflow)

- $h(257135) = 257 + 135 = 392$



6.4.3 Organització del hash





Resolució de col·lisions



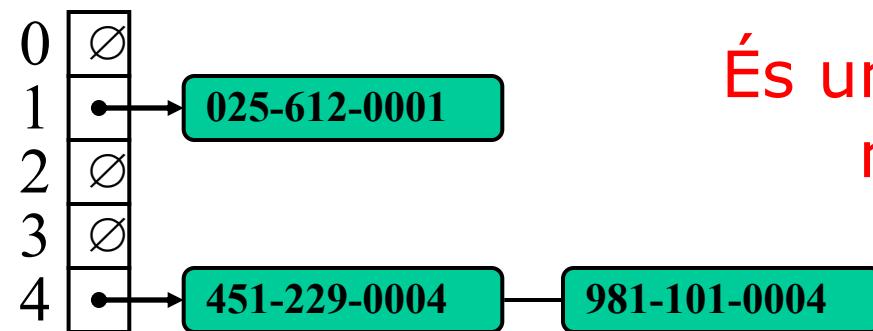
- Una col·lisió ocorre quan diferents elements es mapegen a la mateixa cel·la

ESTRATÈGIES:

- **Hash obert** o direcccionament enllaçat: es dissenya una estructura de llista enllaçada que permet incloure varíes claus en una mateixa posició
- **Hash tancat** o direcccionament obert: Quan es produeix una col·lisió hi ha caselles lliures a la taula, es busca una nova posició per la clau que ha col·lisionat

Tipus de Taules Hash

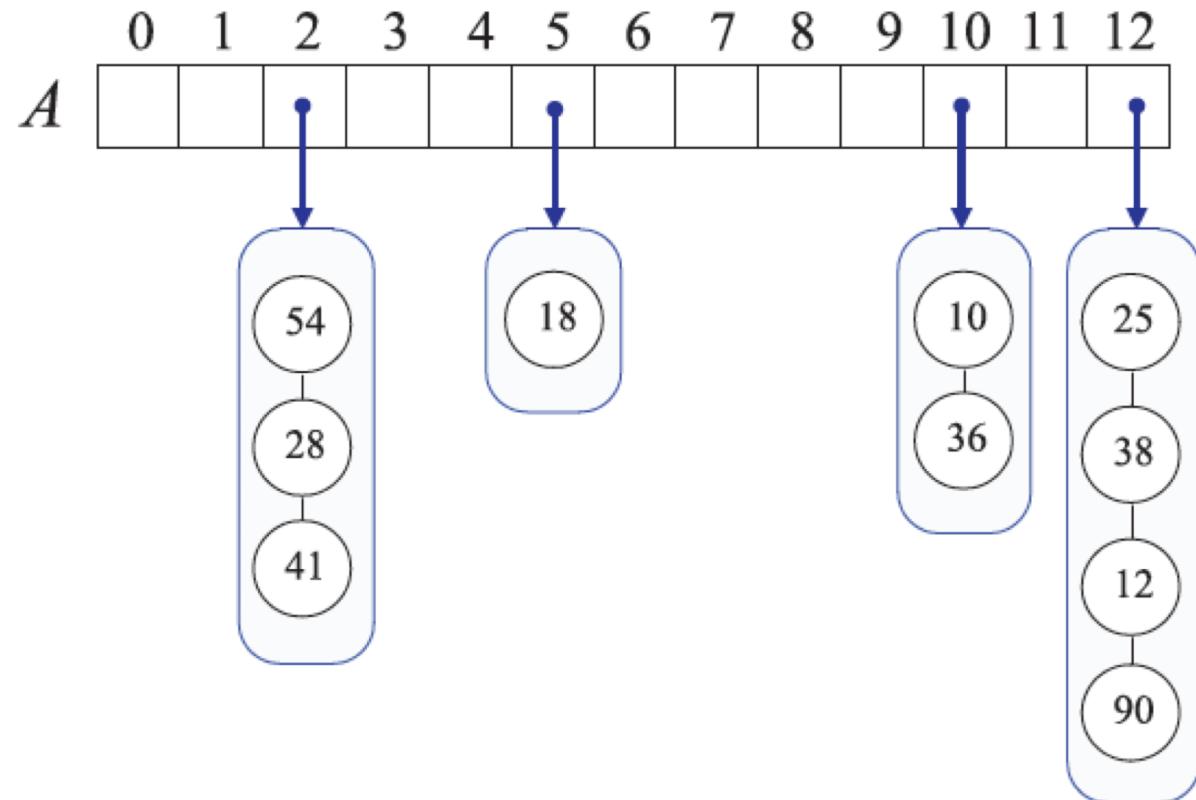
- **Hash Obert o direcccionament enllaçat**
 - Consisteix en tenir a cada posició de la taula, una llista encadenada dels elements `<key,values>` (també anomenat array de “buckets”) que, d’acord a la funció de hash, corresponguin a dita posició
 - En el pitjor cas, el hashing obert ens porta a una llista on totes les claus estan en única llista
 - El pitjor cas en una cerca és $O(n)$



És un mètode simple però requereix més memòria externa a la taula

Tipus de Taules Hash

- **Hash Obert**
- **Exemple:** Taula de hash de mida 13, que guarda 10 entrades amb una clau de tipus enter, amb col·lisions resoltes amb un hash obert



La funció de compressió és:
 $h(k)=k \bmod 13$

Els valors associats a les keys no es mostren en el dibuix.

Taula Hash Oberta

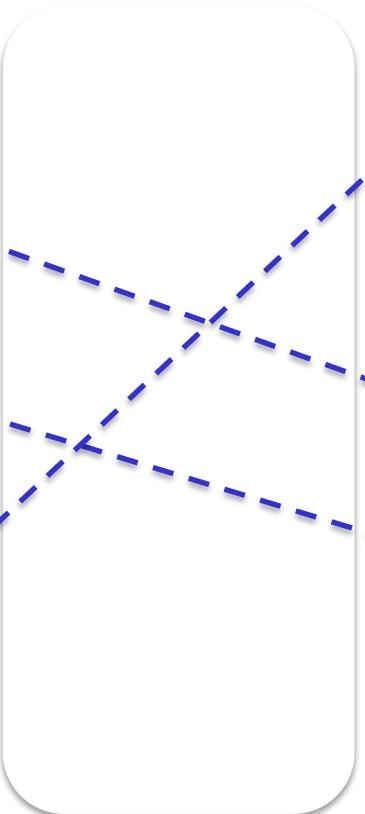
funció de hash:
 $h(key) = key \% 7$

keys:
Banner ID #

B00943855

B00238494

B00472885



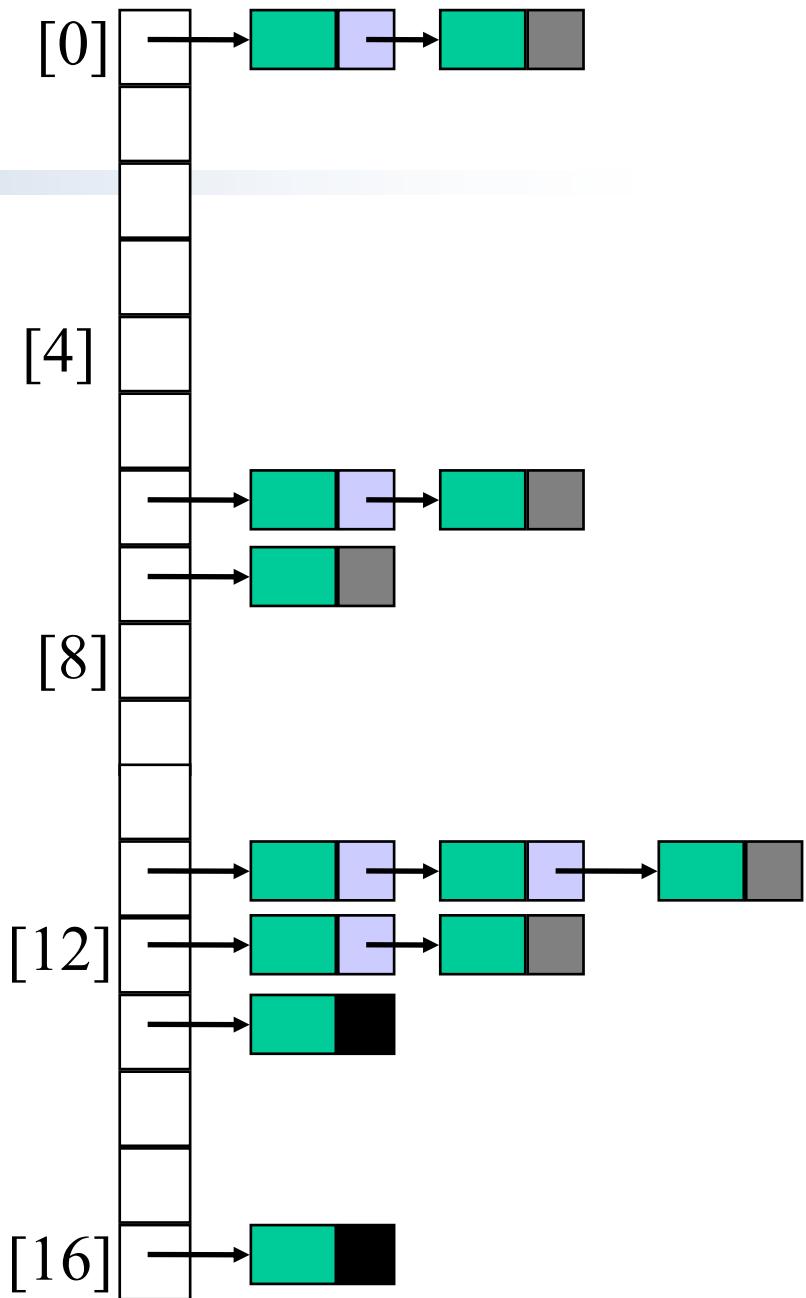
array de “buckets” amb parelles (key, value) :

0	B00472885 David Laidlaw	→	B00231924 Luke Fiorante
1	B00239625 Leah Steinberg		
2			
3	B00943855 Patrick Maiden		
4	B00238494 Sarah Parker		
5	B00745911 Marley Rafson	→	B00543163 Surbhi Madan
6			



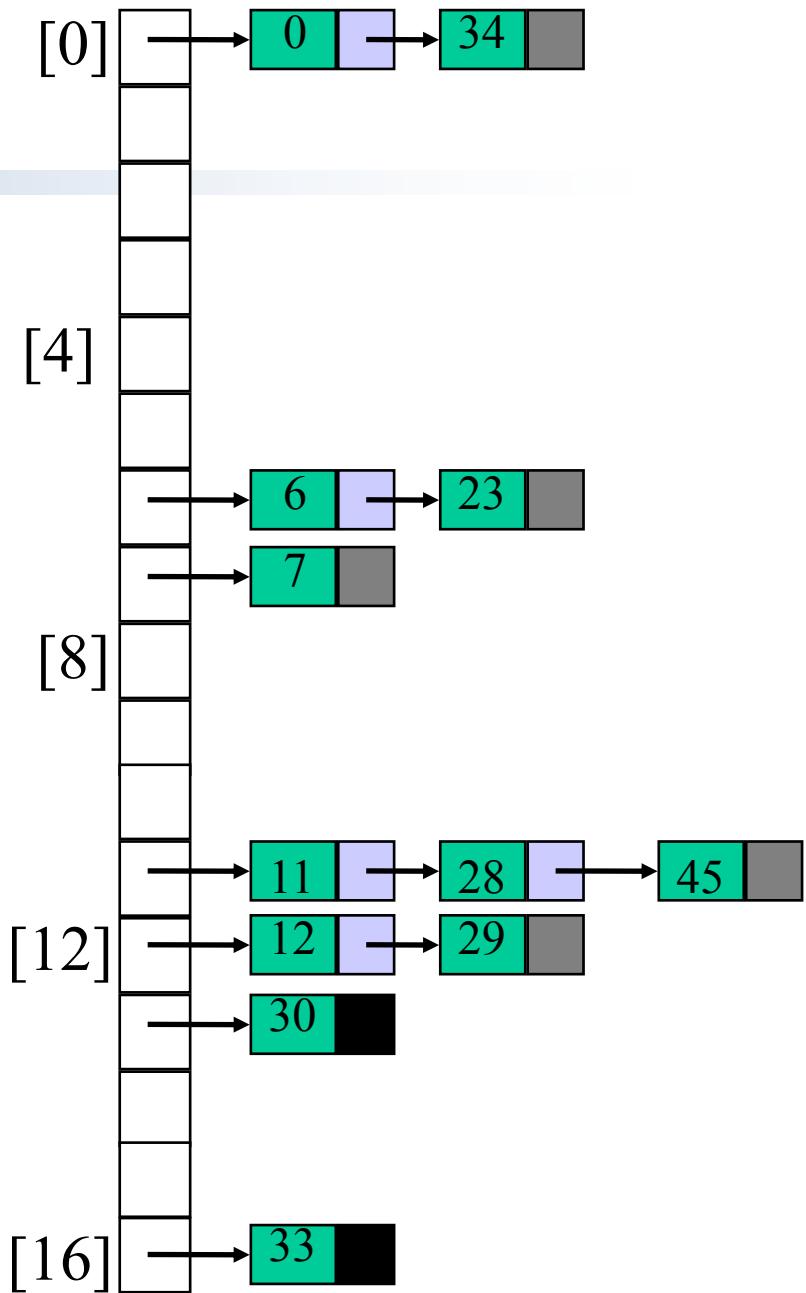
Exemple: Taula Hash Oberta

- Claus a inserir en aquest ordre
 $6, 12, 34, 29, 28, 11, 23, 7, 0,$
 $33, 30, 45$
- Bucket = key % 17



Exemple: Taula Hash Oberta

- Claus a inserir en aquest ordre 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45
- Bucket = key % 17





Taula Hash Oberta

- **Avantatges**

- La gestió de les col·lisions és simple.
- S'insereix sempre a la primera posició de la llista enllaçada per garantir un cost $O(1)$

- **Desavantatges**

- El factor de càrrega pot ser superior a 1. A mesura que augmenta el factor de càrrega, l'eficiència de la cerca disminueix.
- Requereix l'ús de punters
- Ocupa més espai de memòria



Tipus de Taules Hash

- **Hash Tancat o direcccionament obert**

- El hash tancat soluciona les col·lisions cercant cel·les alternatives fins a trobar una buida (dins de la mateixa taula). Es va buscant en les cel·les: $d_0(k)$, $d_1(k)$, $d_2(k)$, ..., on:

$$d_i(k) = (h(k) + f(i)) \bmod \text{MAX_TAULA}$$

$$h(k, i) = (h'(k) + f(i)) \bmod \text{MAX_TAULA}$$

- Quan es fa una **cerca** d'una clau, s'examinen vàries cel·les de la taula fins que es troba la clau buscada, o està clar que aquesta no està emmagatzemada
- La **inserció** s'efectua provant la taula fins a trobar un espai buit

Taula Hash Tancada

funció de hash
 $h(key) = key \% 7$

keys:
Banner ID #

B00943855

B00238494

B00472885

Array amb parelles (key, value):

B00231924
Luke Fiorante

B00239625
Leah Steinberg

B00472885
David Laidlaw

B00943855
Patrick Maiden

B00238494
Sarah Parker

B00745911
Marley Rafson



Taula Hash Tancada

- **Avantatges**

- Elimina totalment els apuntadors. S'allibera així espai de memòria, el que pot ser usat en més entrades de la taula

- **Desavantatges**

- Si l'aplicació realitza eliminacions freqüents, pot degradar-se el rendiment de la mateixa. Es requereix una taula més gran.
 - Per garantir el funcionament correcte, es requereix que la taula de hash tingui, almenys, el 50% de l'espai disponible.



Estratègies en Hash tancat

- **Exploració Lineal:** compara una a una totes les posicions de la taula
- **Exploració quadràtica:** compara les caselles però saltant algunes d'elles
- **Hashing doble:** consisteix en definir dues funcions de hash
 - La primera per obtenir la posició inicial
 - La segona en usar en cas de col·lisió



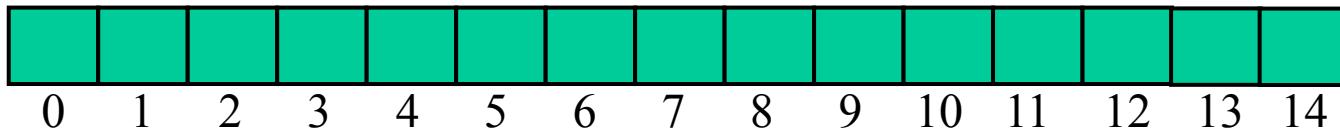
Exploració Lineal

- **Hash tancat:** l'ítem que col·lisiona es situa en una posició diferent de la taula
- **Exploració lineal:**
 - Es considera que la taula és circular
 - Manega les col·lisions situant l'element que col·lisionava a la següent (circular) posició disponible a la taula
$$p, p+1, p+2, p+3, p+4, \dots$$
 - Cada cel·la de la taula inspeccionada es coneix com a “prova”
 - **Avantatge:** que és molt fàcil d’implementar
 - **Desavantatge:** que els elements que col·lisionen s’ajunten en posicions adjacents, causant a les futures col·lisions una seqüència més llarga de proves

Exploració Lineal

- Exemple:

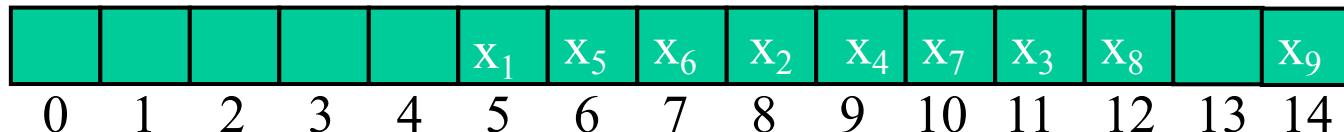
- Si tenim 9 elements amb les claus $x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8$ i x_9
- Element: $x_1 \ x_2 \ x_3 \ x_4 \ x_5 \ x_6 \ x_7 \ x_8 \ x_9$
- $H(x)$: 5 8 11 9 5 7 8 6 14
- Aplicant exploració lineal a on quedarán finalment els elements?



Exploració Lineal

- Exemple:

- Si tenim 9 elements amb les claus $x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8$ i x_9
- Element: $x_1 \ x_2 \ x_3 \ x_4 \ x_5 \ x_6 \ x_7 \ x_8 \ x_9$
- $H(x)$: 5 8 11 9 5 7 8 6 14
- Aplicant exploració lineal a on quedarán finalment els elements?

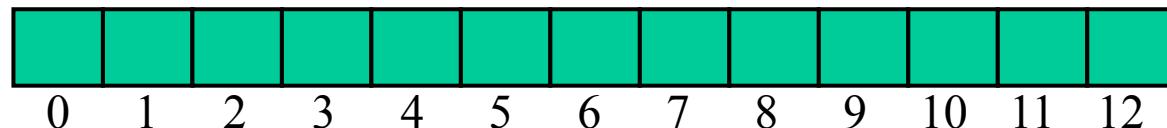


- Element: $x_1 \ x_2 \ x_3 \ x_4 \ x_5 \ x_6 \ x_7 \ x_8 \ x_9$
- $H(x)$: 5 8 11 9 6 7 10 12 14

Exploració Lineal

- Exemple:

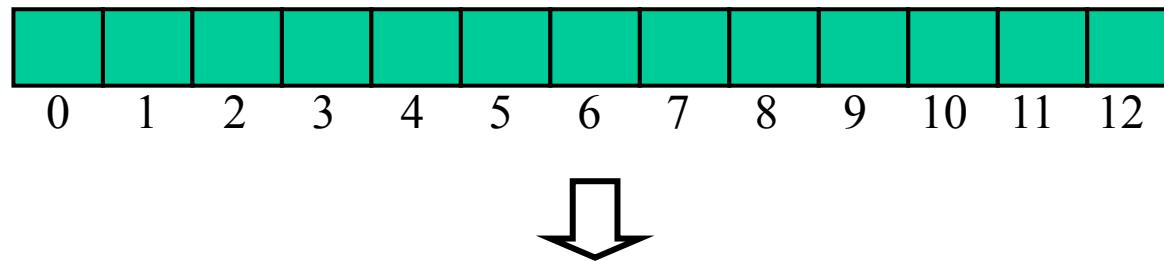
- $h(x) = x \bmod 13$
- Inserta claus 18, 41, 22, 44, 59, 32, 31, 73, en aquest ordre



Exploració Lineal

- Exemple:

- $h(x) = x \bmod 13$
- Inserta claus 18, 41, 22, 44, 59, 32, 31, 73, en aquest ordre



Solució

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12



Cerca amb Exploració lineal



- Considerem una taula hash A que usa exploració lineal
- $\text{find}(k)$
 - Es comença a la cel·la $h(k)$
 - Es proven posicions consecutives fins a que una de les següents condicions ocorre
 - Un ítem amb clau k es troba, o
 - Una cel·la buida es troba, o
 - N cel·les han estat provades sense èxit

Algorithm $\text{find}(k)$

$i \leftarrow h(k)$

$p \leftarrow 0$

repeat

$c \leftarrow A[i]$

if $c = \emptyset$

return null

else if $c.\text{key}() = k$

return $c.\text{value}()$

else

$i \leftarrow (i + 1) \bmod N$

$p \leftarrow p + 1$

until $p = N$

return null



Modificació amb Exploració lineal

- Per manegar insercions i esborrats,
 - introduïm un objecte especial, anomenat *AVAILABLE*, el qual reemplaça els elements esborrats
- **erase(*k*)**
 - Es busca una entry amb clau *k*
 - Si la entry (k, o) es troba, es reemplaça amb un ítem especial *AVAILABLE* i es retorna l'element *o*
 - Sinó, es retorna *null*
- **put(*k, o*)**
 - Es llença una excepció si la taula està plena
 - Es comença a la cel·la $h(k)$
 - Es proven cel·les fins a que ocorre una de les següents condicions
 - Una cel·la *i* es troba que està “buida” o guarda *AVAILABLE*, o
 - *N* cel·les han sigut provades sense èxit
 - Es guarda (k, o) a la cel·la *i*



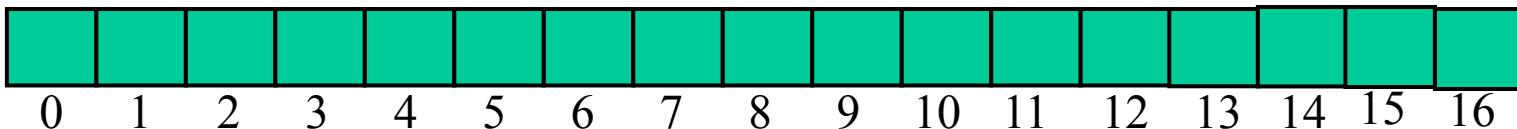
Exploració quadràtica

- **Hash tancat:** l'ítem que col·lisiona es situa en una posició diferent de la taula
- **Exploració quadràtica:**
 - Es considera que la taula és circular
 - Maneja les col·lisions situant l'element segons la fórmula
$$p, p+1, p+4, p+9, \dots, p+i^2 \quad (i = 1, 2, 3, \dots)$$
 - Cada cel·la de la taula inspeccionada es coneix com a “prova”
 - **Avantatge:** Redueix l'agrupament de l'exploració lineal
 - **Desavantatge:** si no s'escull bé la mida de la taula, no es pot garantir que es proven totes les posicions de la taula
 - Es pot demostrar que si la mida de la taula és un nombre primer i el factor de càrrega no arriba al 50%, totes les proves que es realitzen amb la seqüència $p+i^2$ es fa sobre posicions de la taula diferents i sempre es podrà inserir

Exploració Quadràtica

- Exemple:

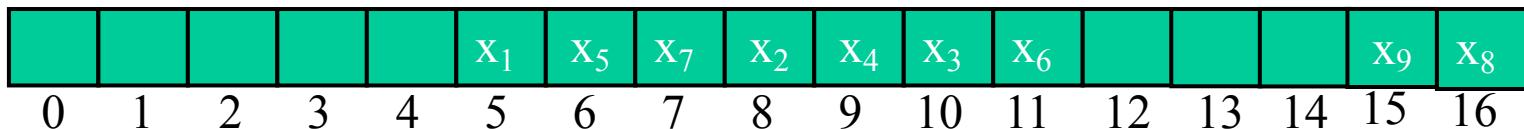
- Si tenim 9 elements amb les claus $x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8$ i x_9
- Element: $x_1 \ x_2 \ x_3 \ x_4 \ x_5 \ x_6 \ x_7 \ x_8 \ x_9$
- $H(x)$: 5 8 10 8 5 11 6 7 7
- Aplicant exploració quadràtica a on quedaran finalment els elements?



Exploració Quadràtica

- Exemple:

- Si tenim 9 elements amb les claus $x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8$ i x_9
- Element: $x_1 \ x_2 \ x_3 \ x_4 \ x_5 \ x_6 \ x_7 \ x_8 \ x_9$
- $H(x)$: 5 8 10 8 5 11 6 7 7
- Aplicant exploració quadràtica a on quedarán finalment els elements?



- Element: $x_1 \ x_2 \ x_3 \ x_4 \ x_5 \ x_6 \ x_7 \ x_8 \ x_9$
- $H(x)$: 5 8 10 9 6 11 7 16 15



Hashing doble



- El hashing doble usa una funció de hash secundària $d(k)$ i manega les col·lisions situant un ítem en la primera cel·la disponible de la sèrie
$$(i + j d(k)) \bmod N \quad (j = 0, 1, \dots, N-1)$$
- La **funció de hash secundària** $d(k)$ no pot retornar un valor zero
- La mida de la taula de hash N ha de ser un nombre primer per permetre provar totes les cel·les
- S'escull la mateixa funció de compressió per la funció de hash secundària:

$$d_2(k) = q - k \bmod q$$

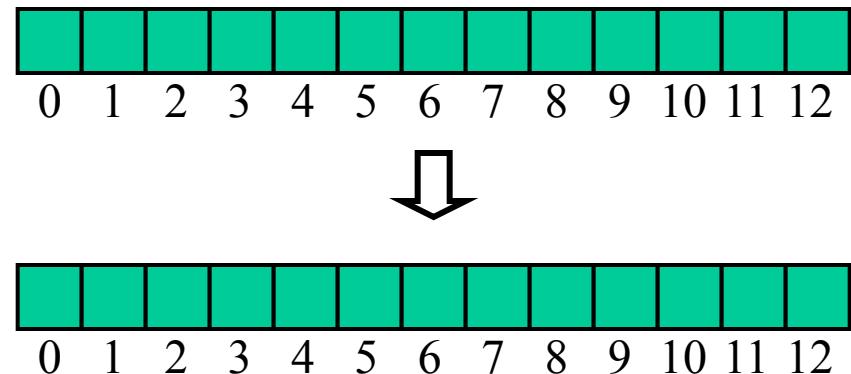
on $q < N$ i q és un nombre primer

- Els possibles valors per $d_2(k)$ són: $1, 2, \dots, q$

Exemple de Hashing doble

- Considera una taula de hash que guarda claus enteres que resol les col·lisions amb un doble hashing
 - $N = 13$
 - $h(k) = k \bmod 13$
 - $d(k) = 7 - k \bmod 7$
- Inserta les claus 18, 41, 22, 44, 59, 32, 31, 73, en aquest ordre

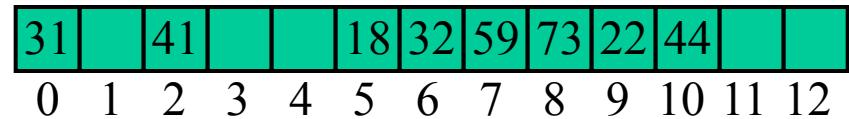
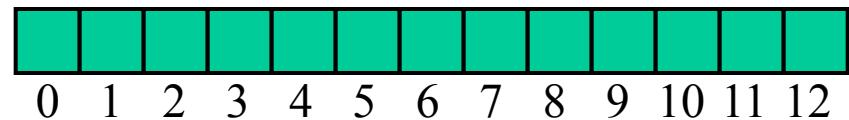
k	$h(k)$	$d(k)$	Proves
18	5	2	
41	11	3	
22	9	1	
44	10	6	
59	12	5	
32	6	7	
31	11	4	
73	12	3	



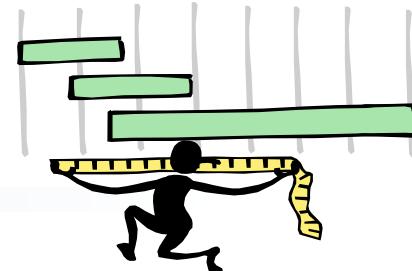
Exemple de Hashing doble

- Considera una taula de hash que guarda claus enteres que resol les col·lisions amb un doble hashing
 - $N = 13$
 - $h(k) = k \bmod 13$
 - $d(k) = 7 - k \bmod 7$
- Inserta les claus 18, 41, 22, 44, 59, 32, 31, 73, en aquest ordre

k	$h(k)$	$d(k)$	Proves
18	5	3	5
41	2	1	2
22	9	6	9
44	5	5	5 10
59	7	4	7
32	6	3	6
31	5	4	5 9 0
73	8	4	8



Rendiment del Hashing



- En el pitjor cas, busca, insereix i elimina en una taula de hash costa $O(n)$ en temps
- El pitjor cas ocorre quan totes les claus inserides en el map col·lisionen
- El factor de càrrega $\lambda = m/N$ afecta al rendiment de la taula de hash
- Assumint que els valors de hash són com nombres randoms, es pot demostrar que el nombre esperat de proves per una inserció amb hash obert és
$$1 / (1 - \lambda)$$
- El temps esperat d'execució per totes les operacions del TAD diccionari en una taula de hash és $O(1)$
- A la pràctica, el hashing és molt ràpid tenint en compte que el factor de càrrega no estigui proper al 100%
- Aplicacions de les taules hash:
 - Petites bases de dades
 - Compiladors
 - La caché dels browsers



Tema 6 Estructures no lineals: Taules de hash

Maria Salamó Llorente
Estructura de Dades

Enginyeria Informàtica
Facultat de Matemàtiques i Informàtica,
Universitat de Barcelona