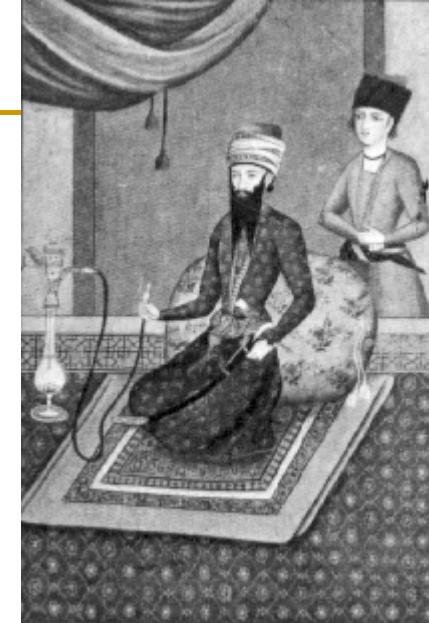
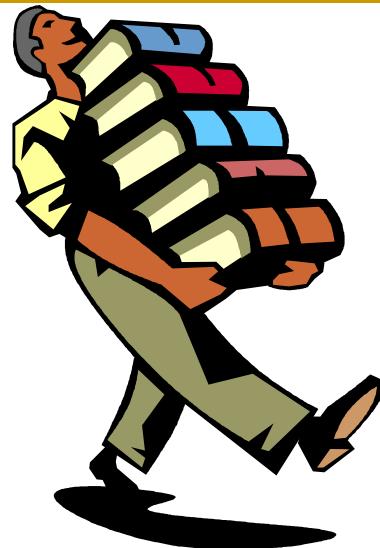


# Diseño de Conjuntos y Diccionarios con Hashing



# Representación de Conjuntos y Diccionarios

- **TAD Diccionario(clave, significado)**
- **Observadores básicos**
- def?: dicc(clave, significado) → bool
- obtener: clave c × dicc(clave, significado) d → significado  
( def?( c, d ) )
- **Generadores**
- vacío: → dicc(clave, significado)
- definir: clave × sign × dicc(clave, significado) → dicc(clave, significado)
- **Otras Operaciones**
- borrar: clave c × dicc(clave, significado) d → dicc(clave, significado)  
( def?( c, d ) )
- claves: dicc(clave, significado) → conj(clave)
- · =dicc · : dicc(α) × dicc(α) → bool



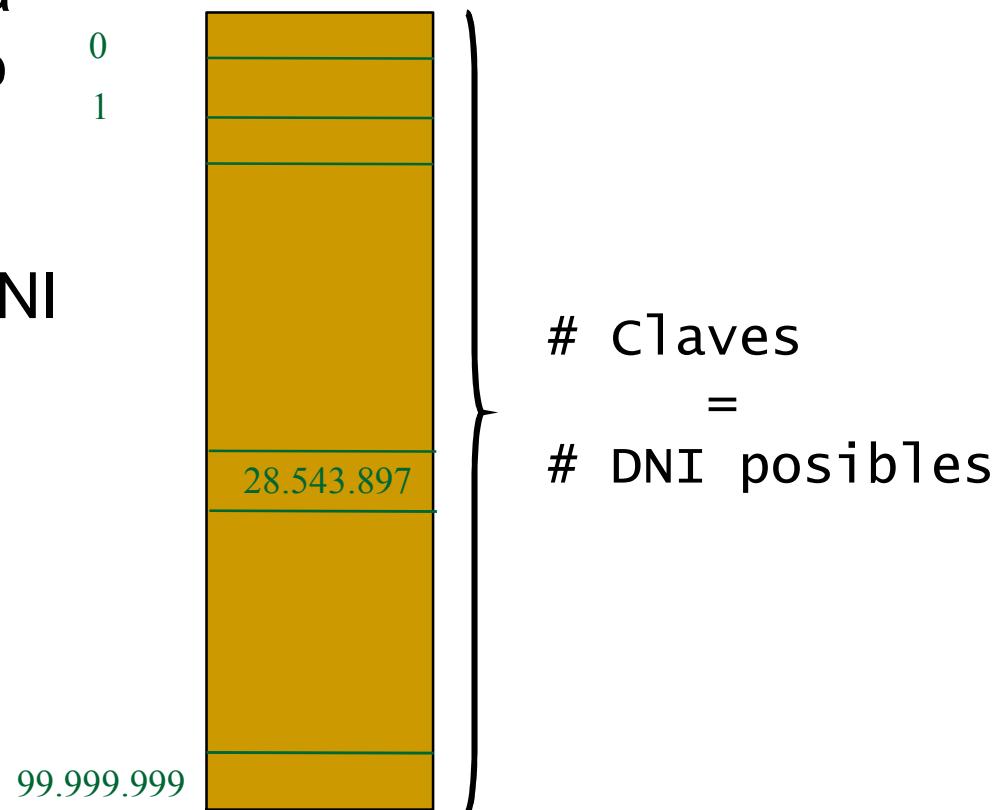
# Tablas hash



- Adecuadas para representar diccionarios
- Generalización del concepto de arreglo
- Importantes para el acceso a datos en memoria secundaria
  - Los accesos se dan en memoria secundaria
  - El costo de los accesos es el predominante
- Otras aplicaciones muy importantes: criptografía / firma digital

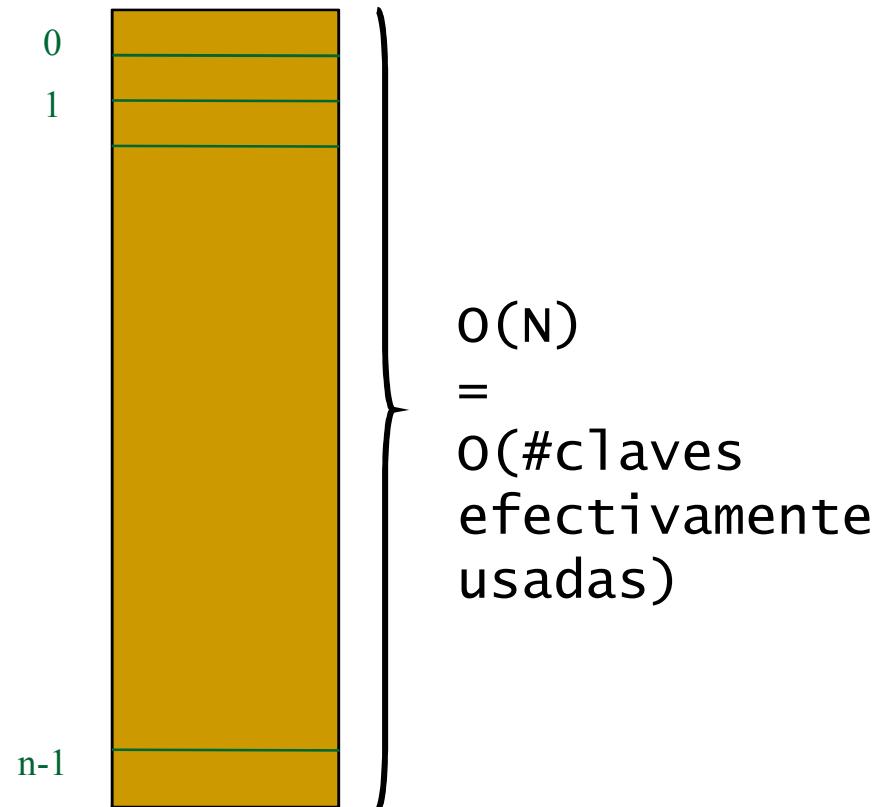
# Direccionamiento directo

- Se asocia a cada valor de la clave un índice de un arreglo
- Por ejemplo,  $10000=10^4$  clientes, clave=número de DNI (nro. entre 0 y  $10^{8-1}$ )
- Búsqueda en tiempo  $O(1)!$
- ¿Problema? Mucho desperdicio de memoria!



# Objetivos

- $n = \#$  claves efectivamente usadas
- Tiempo de búsqueda  $O(1)$
- ¿Será posible?
- Nota:  $\#$  claves posibles puede ser  $\gg n$



# Tabla hash



- Representaremos un diccionario con una tupla  $\langle T, h \rangle$   $\xrightarrow{\text{f. j.o}}$
- Donde  $T$  es un arreglo con  $N = \text{tam}(T)$  celdas
- $h: K \rightarrow \{0, \dots, N-1\}$  es la función hash (o de hash, o de hashing)
  - $K$  conjunto de claves posibles
  - $\{0, \dots, N-1\}$  conjunto de las posiciones de la tabla (a veces llamadas *pseudoclaves*)
  - La posición del elemento en el arreglo se calcula a través de la función  $h$ .

# Hashing perfecto y colisiones

- función hash perfecta:
  - $k_1 \neq k_2 \rightarrow h(k_1) \neq h(k_2)$
  - Requiere  $N \geq |K|$
  - Raramente razonable en la práctica
- En general  $N < |K|$  (muy habitualmente  $N \ll |K|$ )
  - Consecuencia: Es posible que  $h(k_1) = h(k_2)$  aún con  $k_1 \neq k_2$ : colisión
  - Colisiones son más frecuentes que lo intuitivo, ver la paradoja del cumpleaños
- Ejercicio: proponer una función hash perfecta para el caso in que las claves sean strings de largo 3 en el alfabeto  $\{a, b, c\}$

Caso 1: Tenemos  $10^4$  ubicaciones y tenemos  
108 claves posibles  
 $\Rightarrow$  Colisión

# Resolución de colisiones

- Los métodos se diferencian por la forma de ubicar a los elementos que dan lugar a colisión. Dos familias principales:
- Direccionamiento cerrado o Concatenación : a la  $i$ -ésima posición de la tabla se le asocia la lista de los elementos tales que  $h(k)=i$ .
- Direccionamiento abierto: todos los elementos se guardan en la tabla (luego veremos cómo).

# Paradoja del cumpleaños

- Si elegimos 23 personas al azar, la probabilidad de que dos de ellos cumplan años el mismo día es  $> \frac{1}{2}$  (aprox. 50.7%)
  - ¡Demostrar!
- En términos de hashing, aún suponiendo una distribución uniforme entre las pseudoclaves, la probabilidad de que con 23 inserciones en una tabla de 365 posiciones se genere una colisión es mayor que  $\frac{1}{2}$ .

# Requisitos de una función hash

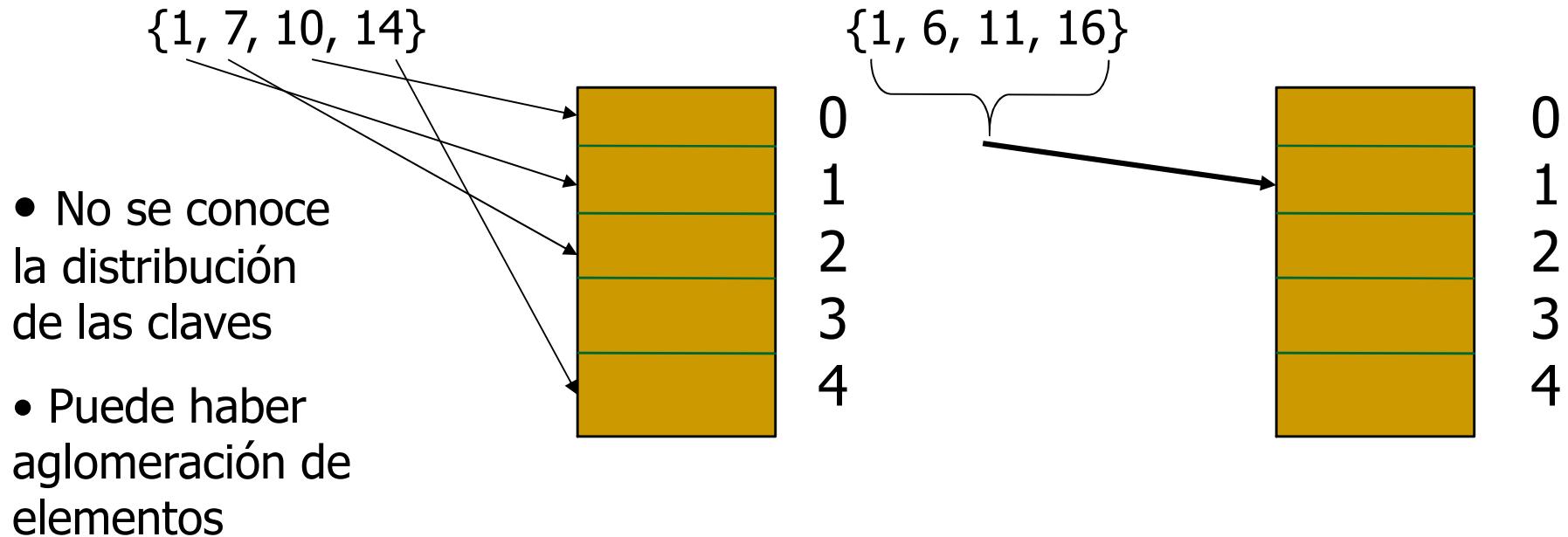
- Distribución de probabilidad de las claves:
  - $P(k)$  = probabilidad de la clave  $k$
- Uniformidad simple:

$$\forall j \sum_{k \in K : h(k)=j} P(k) \approx 1 / |N|$$

- Intuitivamente, se quiere que los elementos se distribuyan en el arreglo en manera uniforme (pensar en el caso  $P(k)=1/|K|$ )
- Difícil construir funciones que satisfagan la uniformidad simple:  $P$  generalmente es desconocida!

# Requisitos de una función hash/2

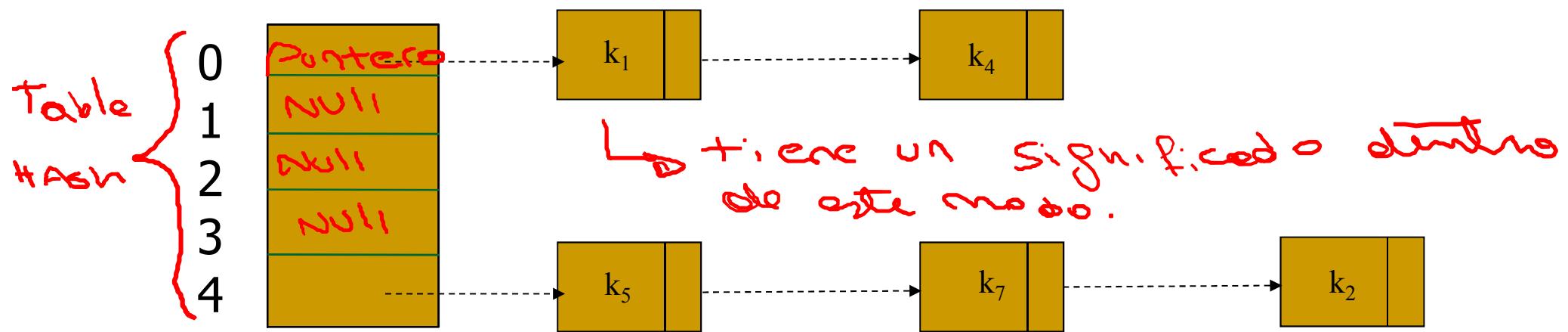
- Ejemplo: sea  $|T|=5$  y  $h(k)=k \bmod 5$



En la práctica: se trata de tener independencia de la distribución de los datos

# Concatenación

- $h(k_1) = h(k_4) = 0$
- $h(k_5) = h(k_7) = h(k_2) = 4$



- Ej.:  $h(k) = k \bmod 5$
- $k_1 = 0, k_4 = 10$
- $k_5 = 9, k_7 = 14, k_2 = 4$

# Concatenación: complejidad

- insert(el, k): inserción al principio de la lista asociada a la posición  $h(k)$ : costo  $O(1)$
- buscar(k): búsqueda linear en la lista asociada a la posición  $h(k)$ : costo  $O(\text{longitud de la lista asociada a } h(k))$
- delete(k): búsqueda en la lista asociada a la posición  $h(k)$ : costo  $O(\text{longitud de la lista asociada a } h(k))$
- Pero ¿cuánto miden las listas?

## Concatenación: ¿cuánto miden las listas?

- buscamos crear una buena función de hashing
- $n = \#$ elementos en la tabla ~~#Claves~~ y  $|T|$  es el largo del array fijo
  - $\alpha = n/|T|$ : factor de carga
  - Teorema: bajo la hipótesis de simplicidad uniforme de la función de hash, si las colisiones se resuelven por concatenación, en promedio
    - una búsqueda fallida requiere tiempo  $\Theta(1+\alpha)$
    - una búsqueda exitosa requiere tiempo  $\Theta(1+\alpha/2)$
  - No lo demostramos formalmente, pero debería ser intuitivo...
  - $O(1)$  si  $n \sim |T| \rightarrow$  dimensionar bien T es importante!

→ elegir bien el tamaño del array solo es chico.

~~más difícil es redimensionar tablas~~

## Direccionamiento abierto

neesition ITL > N

NO hay listas.

- Todos los elementos se almacenan en la tabla  $\Rightarrow \alpha > 1$
- Las colisiones se resuelven dentro de la tabla
  - Si la posición calculada está ocupada, hay que buscar una posición libre.  $\rightarrow$  Iterativo .
  - Los distintos métodos con direccionamiento abierto se distinguen por el método de barrido que utilizan.
  - La función hash pasa a depender también del número de intentos realizados
  - Dirección= $h(k, i)$  para el  $i$ -ésimo intento
  - $h(k,i)$  debe generar todas las posiciones de T
  - Problemas con el borrado! (Ya van a ver...)

ver combinando  
a la func. de  
hash con  
el indice i

# Algoritmos: Inserción

si } clave → Table

insertar (el, k, T) es

$i \leftarrow 0;$

mientras ( $T[h(k, i)]$  está ocupada e  $(i < |T|)$ )

incrementar  $i;$

si ( $i < |T|$ ), hacer  $T[h(k, i)] \leftarrow (el, k)$

en caso contrario <overflow>

tuple

clave con este  
se llenó la table clave

$$h_i(n) = (n + i) \bmod x$$

- Ojo: ¡Podemos tener overflow!

Ins: {7, 10, 2, 22}

$$\text{Ins}(7) = h_0(7) = 7 \bmod 5 = 2$$

$$\text{Ins}(2) = h_0(2) = 2 \Rightarrow h_1(2) = (2+2) \bmod 4 = 0$$



# Algoritmos: búsqueda

buscar ( $k, T$ )

$i=0;$

mientras ( $(k \neq T[h(k, i)].clave) \wedge T[h(k, i)] \neq \text{null}$   
 $\wedge (i < |T|)$ ) incrementar  $i$ ;

Si ( $i < |T|$ ) y  $T[h(k, i)] \neq \text{null}$  entonces  $T[h(k, i)] \rightarrow$   
en caso contrario <no está>

→  $i$  es null cortamos el bucle  
∴ es un problema por si borramos un elemento

- Ojo:  $T[h(k, i)] \neq \text{null}$  ¿Entonces, cómo borramos?
- Podemos marcar los elementos como “borrados”, en lugar de “null”, pero....

$$h'(x) = x \bmod 5$$

$$\text{Búsqueda}(12) \Rightarrow h_0(12) = 2 \quad \text{No}$$

$$h_1(12) = 3 \quad \text{No}$$

$$h_2(12) = 4 \quad \text{Si}$$



# Barrido

- La función  $h(k, i)$  debe recorrer todas las posiciones de la tabla
- Varias formas típicas para la función  $h(k,i)$ 
  - Barrido linear
  - Barrido cuadrático
  - Hashing doble
- Se diferencian entre sí por su complejidad de cálculo y por el comportamiento respecto a los fenómenos de aglomeración.

# Barrido linear

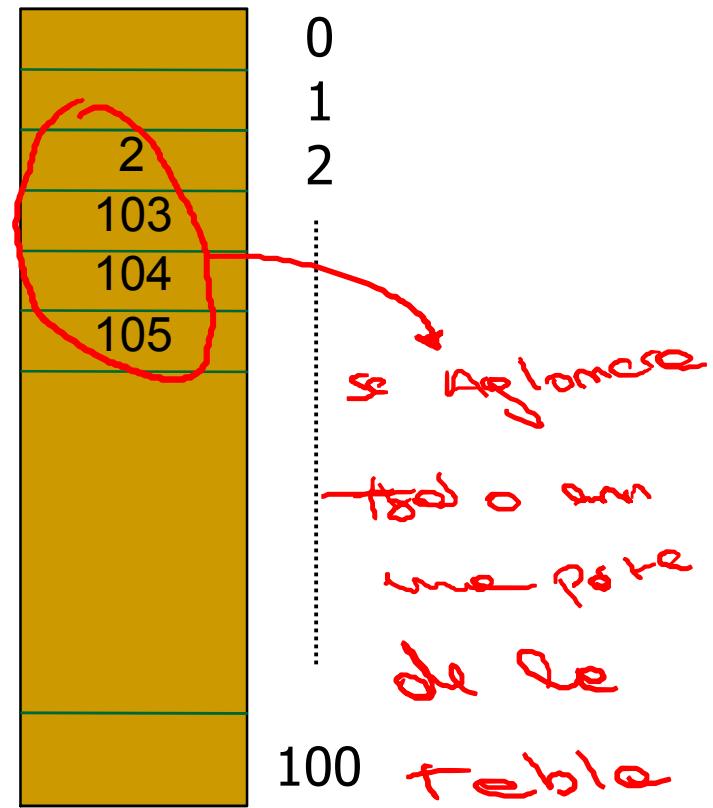
Caso que vimos arriba

- $h(k, i) = (h'(k)+i) \bmod |T|$ , donde  $h'(k)$  es una función de hashing
  - Se recorren todas las posiciones en la secuencia  $T[h'(k)], T[h'(k)+1], \dots, T[|T|], 0, 1, \dots, T[h'(k)-1]$  Pero caso
  - Posibilidad de aglomeración primaria: dos secuencias de barrido que tienen una colisión, siguen colisionando \rightarrow x \in 3, todos los mun mod 2
  - Los elementos se aglomeran por largos tramos
- ↓  
nos dan los  
mismos t-cont
- es decir el 7 y el 2 tienen la misma persona  
por el mismo recorrido de  
Indices.

# Aglomeración primaria

- $h(k, i) = (h'(k)+i) \bmod 101$
- $h'(k)=k \bmod 101$
- Secuencia de inserciones  
 $\{2, 103, 104, 105, \dots\}$
- Caso extremo, pero el problema existe!

Nunca encontrarás los primero;



# Barrido cuadrático

- $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod |T|$ , donde  $h'(k)$  es una función de hashing,  $c_1$  y  $c_2$  son constantes
- Ejemplos:
  - $h(k, i) = h'(k) + i^2$ ,  $h(k, i+1) = h'(k) + (i+1)^2$ ,  $i=1, \dots, (|T|-1)/2$
  - $h(k, i) = h'(k) + i/2 + i^2/2$ ,  $|T|=2^x$  *hay que dejar un i sobrante*
- Posibilidad de aglomeración secundaria: si hay colisión en el primer intento... sigue habiendo colisiones ( $h'(k_1) = h'(k_2) \rightarrow h'(k_1, i) = h'(k_2, i)$ )  
*por la secuencia + algo*
- Describir  $h(k, i)$  cuando  $h'(k) = k \bmod |5|$

*Estos 3 dígitos serán en 2 dígitos luego el mismo valor en el paso 0 → tanto considerar en el Paso 1.*

# Hashing doble

- Idea: que el barrido también dependa de la clave
- $h(k, i) = (h_1(k) + ih_2(k)) \text{ mod } |T|$ , donde  $h_1(k)$  y  $h_2(k)$  son funciones de hashing *→ mato 2 factos*
- El hashing doble reduce los fenómenos de aglomeración secundaria
- Y no tiene aglomeración primaria

# Construcción de funciones de Hash

1º Forma una func. Pre Hashing luego hasheo

- Recordemos que las claves no son necesariamente números naturales
- Por ejemplo, las claves podrían ser strings
- Solución: asociar a cada clave un entero
- ¿Cómo? Depende de la aplicación, de las claves, etc.

# Ejemplo: strings

- Possible método: asociar a cada carácter su código ASCII y a la cadena el número entero obtenido en una determinada base
- Ejemplo: base 2, posición menos significativa a la derecha

*Pre-metodología*

String = "ppt" → pseudoclave =  $112*2^2+112*2^1+116*2^0=788$

Ascii('p')=112

Ascii('t')=116

*Única escritura*

# Funciones hash

- Múchos métodos
  - División
  - Partición
  - Mid-square
  - Extracción
  - .....
- Objetivo: distribución lo más uniforme posible...
- Diferencias:
  - Complejidad
  - Tratamiento de los fenómenos de aglomeración

# División

- $h(k) = k \bmod |T|$
- Baja complejidad
- Aglomeraciones
  - No potencias de 2: si  $|T| = 2^p$  entonces todas las claves con los p bits menos significativos iguales, colisionan
  - No potencias de 10 si las claves son números decimales (mismo motivo)
  - En general, la función debería depender de todas las cifras de la clave, cualquiera sea la representación
  - Una buena elección en la práctica: un número primo no demasiado cercano a una potencia de 2 (ejemplo:  $h(k) = k \bmod 701$  para  $|K|=2048$  valores posibles)

# Partición

- Partitionar la clave  $k$  en  $k_1, k_2, \dots, k_n$
- $h(k) = f(k_1, k_2, \dots, k_n)$
- Ejemplo: la clave es un No. de tarjeta de crédito. Posible función hash:

*→ Punto de clave → 3 bandas  
Algunas → de Clu .*

4772 6453 7348 → {477, 264, 537, 348}

$$\begin{aligned}f(477, 264, 537, 348) &= (477 + 264 + 537 + 348) \bmod 701 \\&= 224\end{aligned}$$

# Extracción

- Se usa solamente una parte de la clave para calcular la dirección
- Ejemplo: Las 6 cifras centrales del número de tarjeta de crédito
  - 4772 6453 7348 → 264537

*nosotros seleccionamos algunas cifras porque en el medio no*

- El número obtenido puede ser manipulado ulteriormente *este*
- La dirección puede depender de una parte de la clave *como por ej. el DNS* *este* *partido*

---