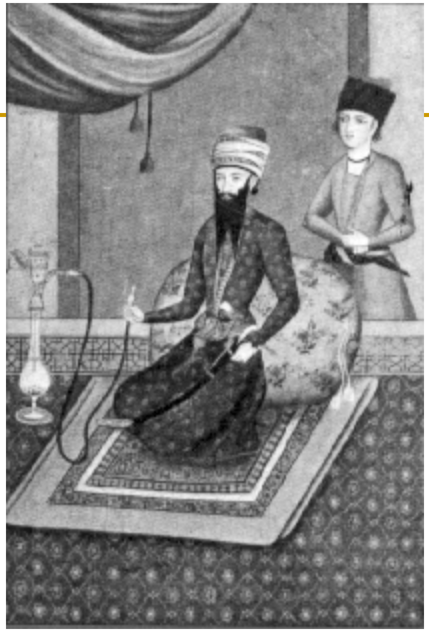
Diseño de Conjuntos y Diccionarios con Hashing



Representación de Conjuntos y Diccionarios

- **TAD Diccionario(clave, significado)**
- **Observadores básicos**
- def?: dicc(clave, significado) $\rightarrow$ bool
- obtener: clave c $\times$ dicc(clave, significado) d $\rightarrow$ significado
  (def?(c, d))
- **Generadores**
- vacío: $\rightarrow$ dicc(clave, significado)
- definir: clave $\times$ sign $\times$ dicc(clave, significado) $\rightarrow$ dicc(clave, significado)
- **Otras Operaciones**
- borrar: clave c $\times$ dicc(clave, significado) d $\rightarrow$ dicc(clave, significado) (def?(c, d))
- claves: dicc(clave, significado) $\rightarrow$ conj(clave)
- $\cdot$ =dicc $\cdot$: dicc(α) $\times$ dicc(α) $\rightarrow$ 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

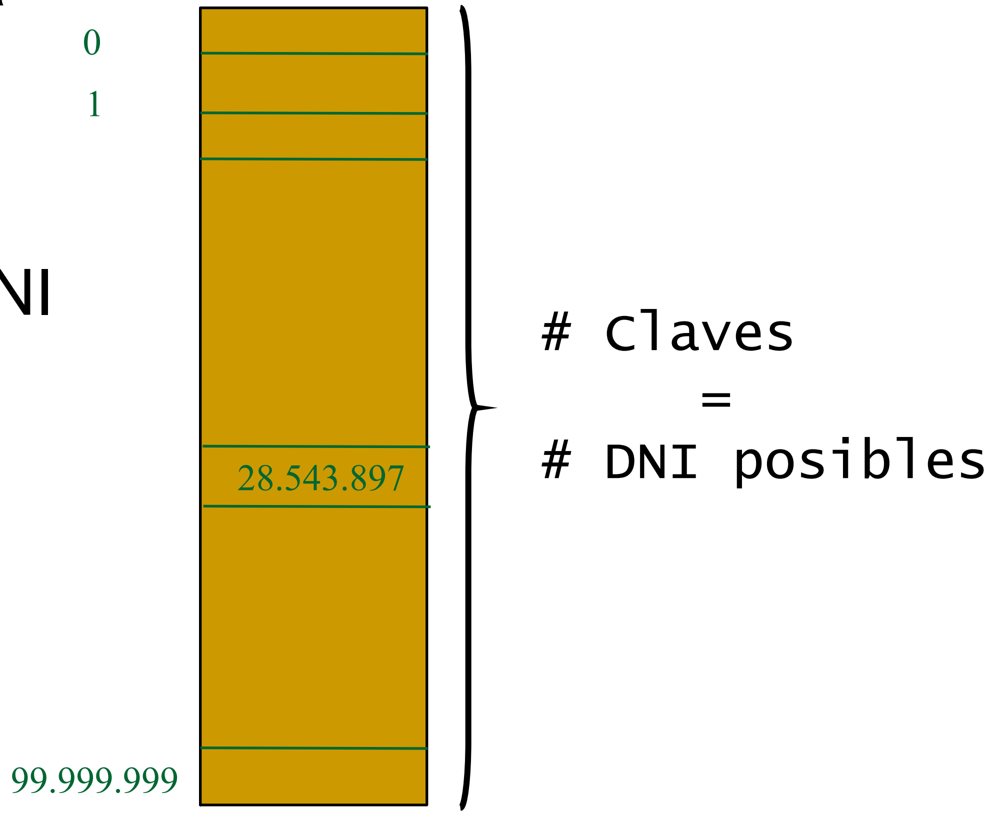
Direccionamiento directo

- 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)!

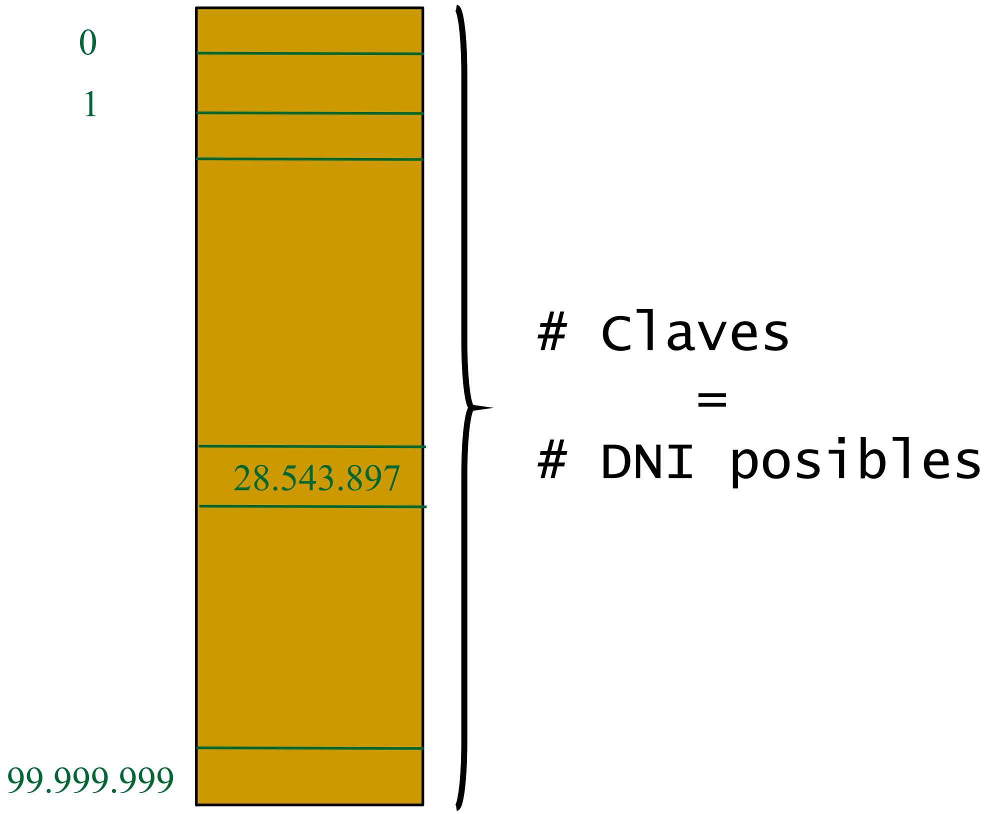
- ¿Problema? Mucho desperdicio de memoria!



Problemas

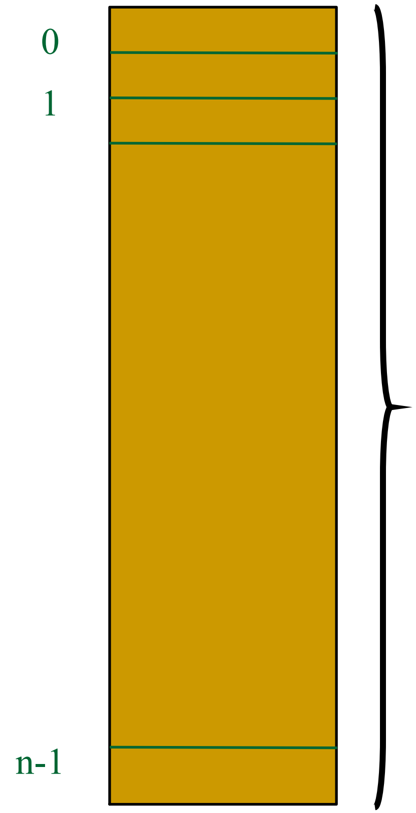
1. Sólo sirve para claves que son enteros no negativos.

2. Mucho desperdicio de memoria!



Objetivos

- Indexar otros tipos de datos (no sólo enteros).
- n = # claves efectivamente usadas
- Tiempo de búsqueda O(1)
- ¿Será posible?
- Nota: # claves posibles puede ser >> n




```
O(N)
=
O(#claves
efectivamente
usadas)
```

Pre-hashing

- Indexar otros tipos de datos (no sólo enteros).
- Necesidad de una función de correspondencia entre cualquier tipo de datos y un entero.

Pre-hashing

- Indexar otros tipos de datos (no sólo enteros).
- Necesidad de una función de correspondencia entre cualquier tipo de datos y un entero.

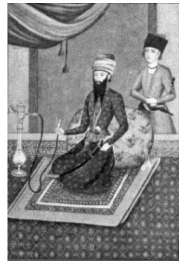
- Dos posibles alternativas:
  - En teoría:
  - ”hola” -> 10011101 -> 157
  - ”casa” -> 1100101 -> ….

Pre-hashing

- Indexar otros tipos de datos (no sólo enteros).
- Necesidad de una función de correspondencia entre cualquier tipo de datos y un entero.

- Dos posibles alternativas:
  - En teoría:
  - ”hola” -> 10011101 -> 157
  - ”casa” -> 1100101 -> ….
  - No es tan simple! En la práctica, depende de la implementación de cada lenguaje de programación.
  - hash(key) -> int
  - hash(x) = hash(y) <-> x=y (idealmente)

Tabla hash



- Representaremos un diccionario con una tupla <T, h>
- Donde T es un arreglo con N=tam(T) celdas
- $h: K \rightarrow \{0,\ldots,N-1\}$ es la función hash (o de hash, o de hashing)
  - K conjunto de claves posibles
  - $\{0,\ldots,N-1\}$ 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.

Tabla hash

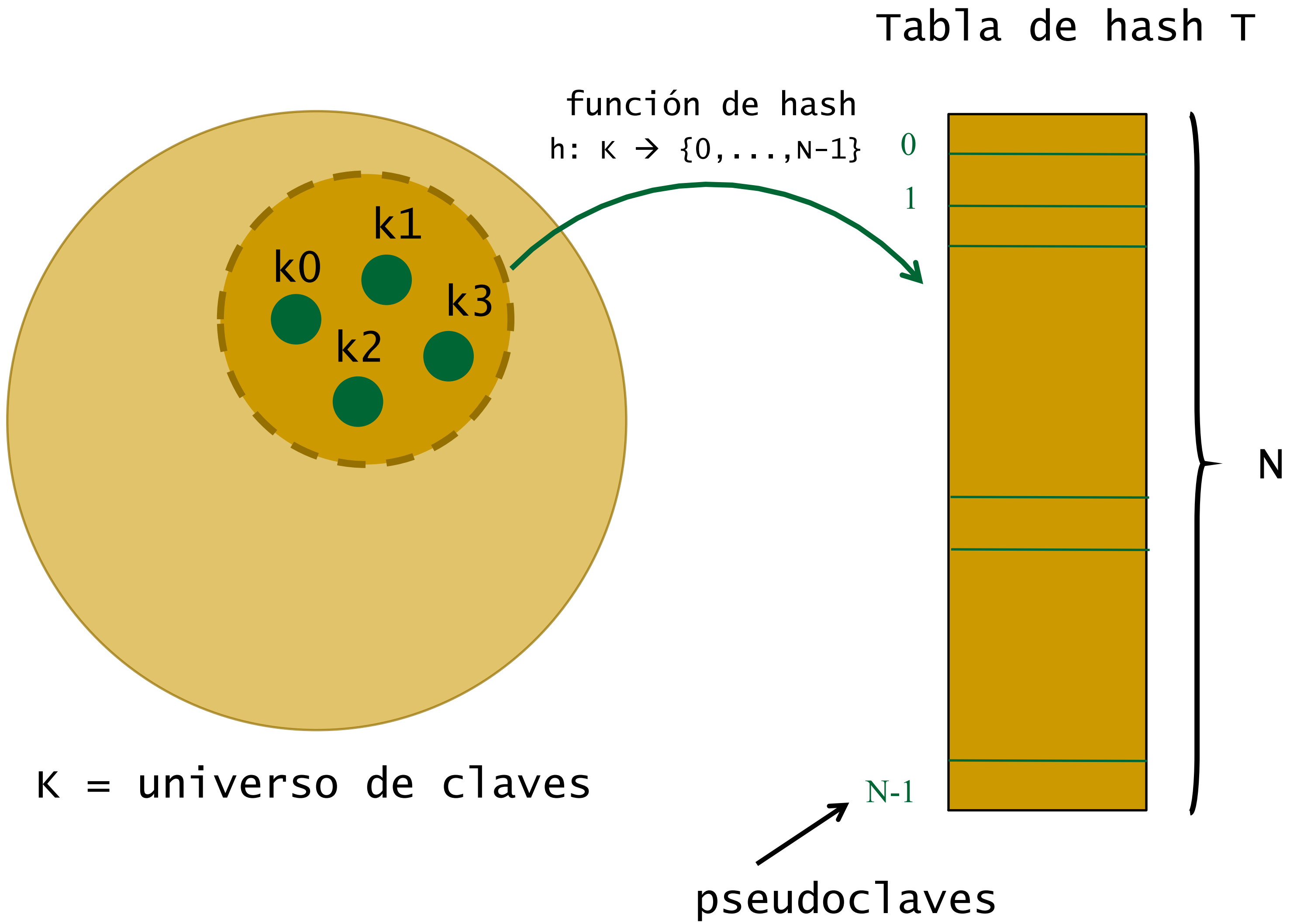


Tabla de hash T
función de hash
h: K → {0,...,N-1}
k1
k0
k3
k2
0
1
N
N-1
K = universo de claves
pseudoclaves

Tabla hash

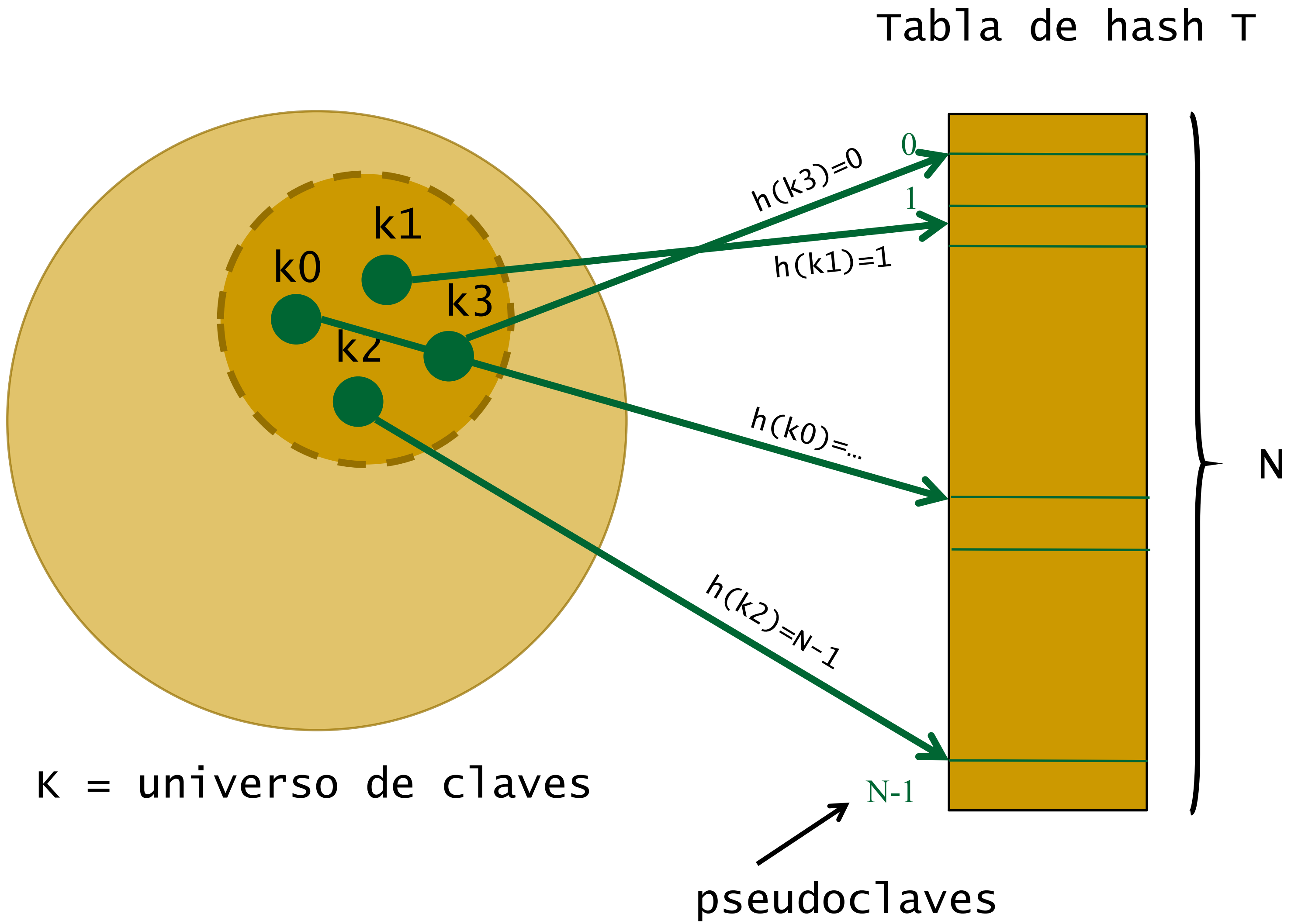


Tabla de hash T
k1
k0
k3
k2
h(k3)=0
h(k1)=1
h(k0)=...
h(k2)=N-1
0
1
N-1
N
K = universo de claves
pseudoclaves

Tabla hash

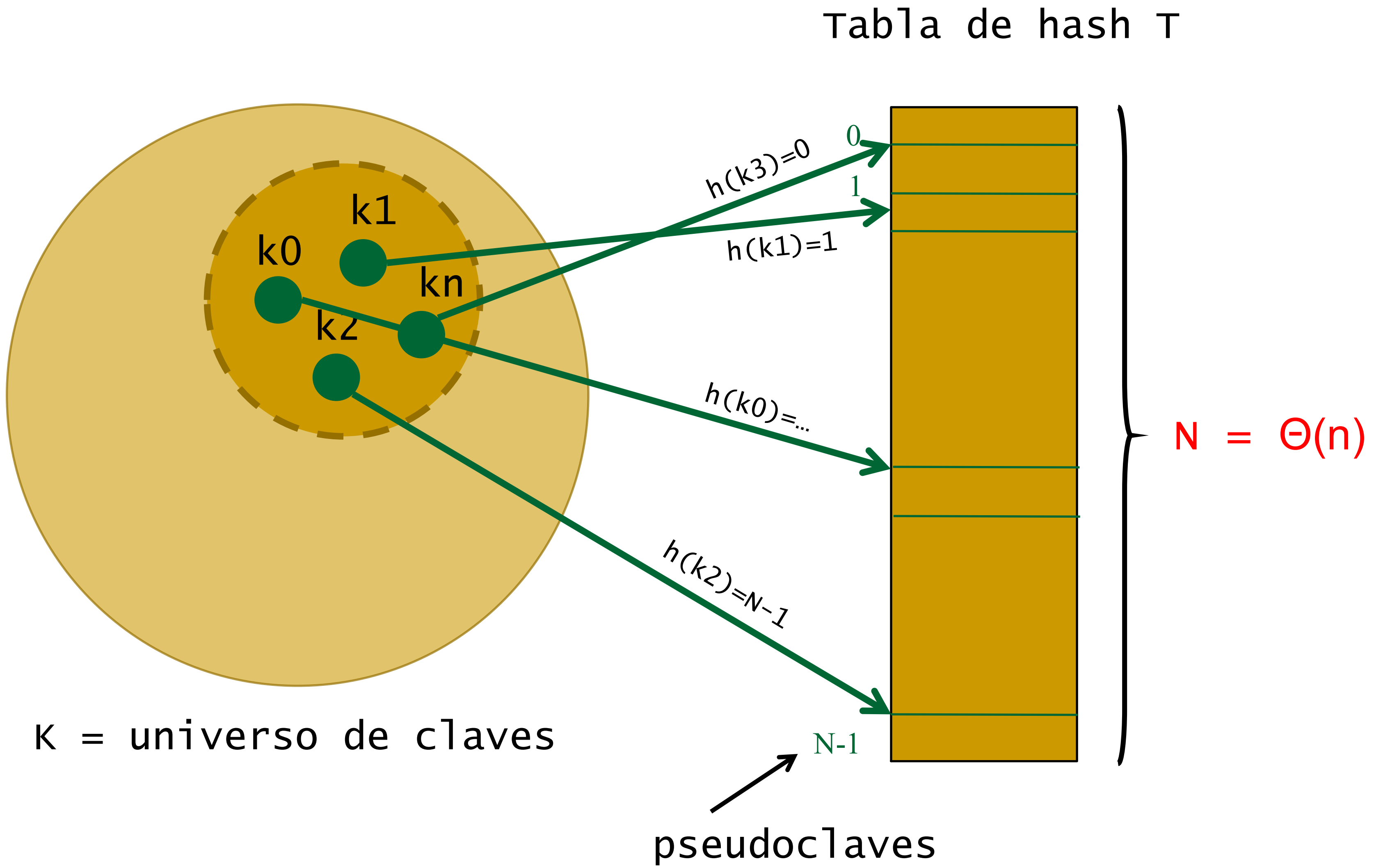


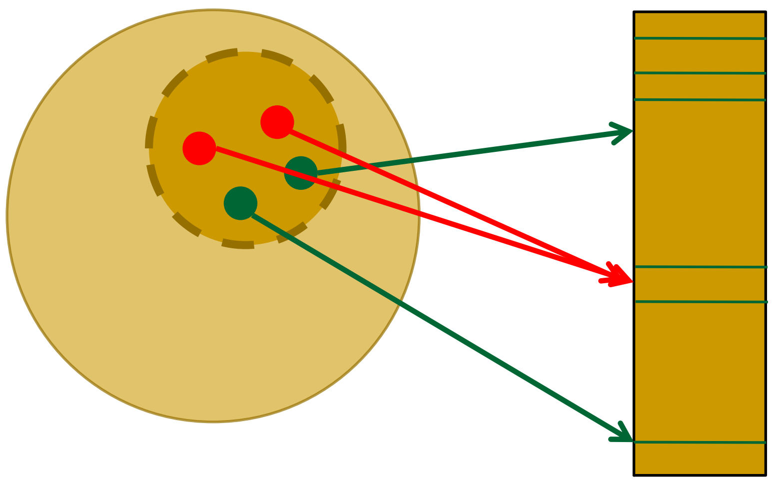
Tabla de hash T
k1
k0
kn
k2
h(k3)=0
h(k1)=1
h(k0)=...
h(k2)=N-1
0
1
N-1
N = Θ(n)
K = universo de claves
pseudoclaves

Hashing perfecto y colisiones

- función hash perfecta:
  - $k_1 != k_2 \rightarrow h(k_1) != h(k_2)$
  - Requiere N >= |K|
  - Raramente razonable en la práctica
- En general N < |K| (muy habitualmente N << |K|)

Hashing perfecto y colisiones

- función hash perfecta:
  - $k_1 != k_2 \rightarrow h(k_1) != h(k_2)$
  - Requiere N >= |K|
  - Raramente razonable en la práctica
- En general N < |K| (muy habitualmente N << |K|)



$h(k_1) == h(k_2)$ aún si $k_1 != k_2$:

Hashing perfecto y colisiones

- función hash perfecta:
  - $k_1 != k_2 \rightarrow h(k_1) != h(k_2)$
  - Requiere $N >= |K|$
  - Raramente razonable en la práctica
- En general $N < |K|$ (muy habitualmente $N << |K|$)
  - Consecuencia: Es posible que $h(k_1) == h(k_2)$ aún con $k_1 != k_2$: 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}

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 > ½ (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 ½.

Requisitos de una función hash

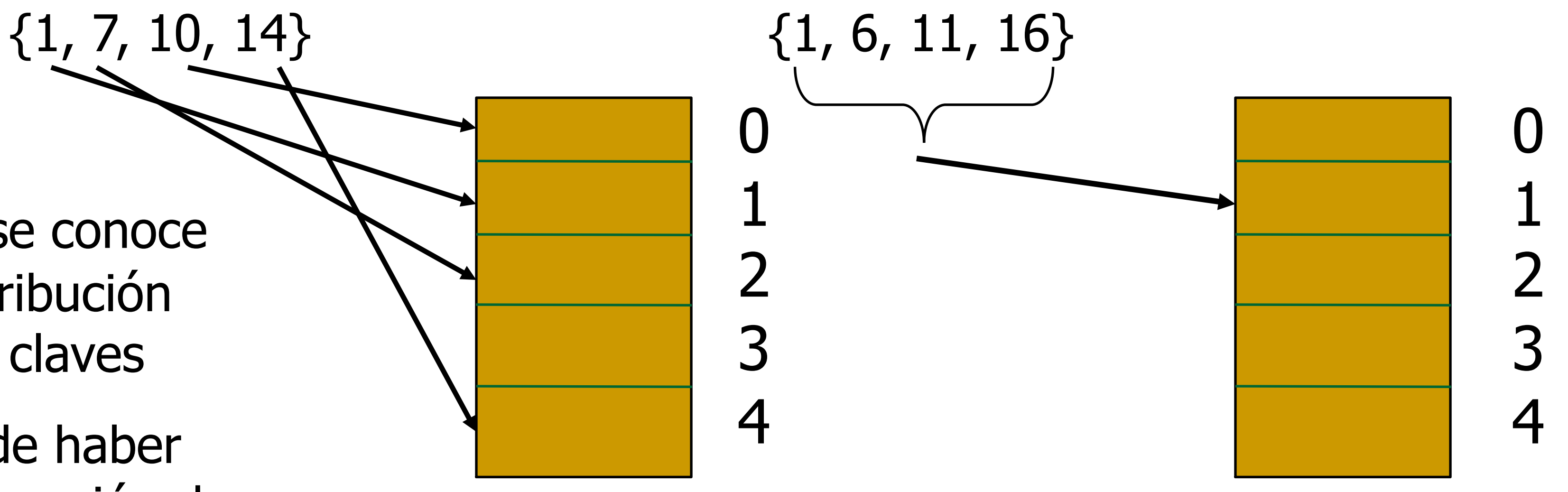
- Distribución de probabilidad de las claves:
  - P(k) = probabilidad de la clave k
- Uniformidad simple:

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

- 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 mod 5

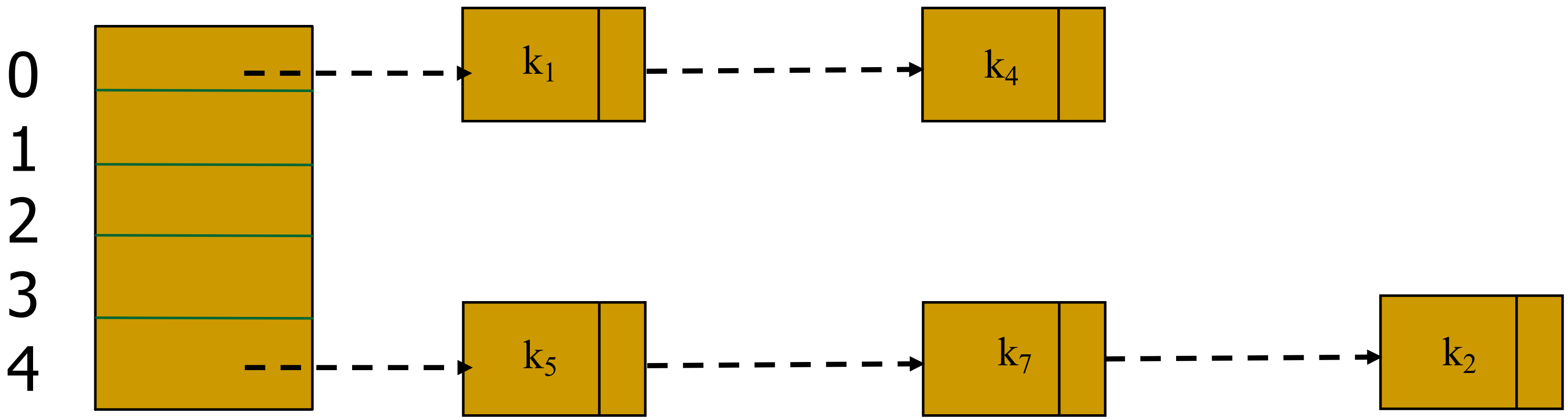


- No se conoce la distribución de las claves
- Puede haber aglomeración de elementos

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 mod 5
- k1=0, k4=10
- k5=9, k7=14, k2=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*(longitud de la lista asociada a h(k))
- delete(k): búsqueda en la lista asociada a la posición h(k): costo *O*(longitud de la lista asociada a h(k))
- Pero ¿cuánto miden las listas?

Concatenación: ¿cuánto miden las listas?

- n = #elementos en la tabla
- α = 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~|T| $\rightarrow$ ¡dimensionar bien T es importante!

Direccionamiento abierto

- Todos los elementos se almacenan en la tabla
- Las colisiones se resuelven dentro de la tabla
  - Si la posición calculada está ocupada, hay que buscar una posición libre.
  - Los distintos métodos con direccionamiento abierto se distinguen por el método de barrido que utilzan.
  - 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...)

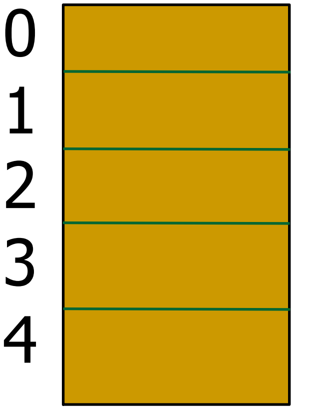
Algoritmos: Inserción

```
insertar (el, k, T) es
  i ← 0;
  mientras (T[h(k, i)] está ocupada e (i<|T|))
      incrementar i;
  si (i < |T|), hacer T[h(k, i)] ← (el,k)
  en caso contrario <overflow>
```

- Ojo: ¡Podemos tener overflow!

Algoritmos: Inserción

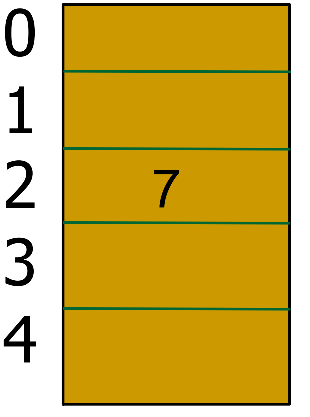
Insertar: 7,10,2,12



Algoritmos: Inserción

Insertar: 7,10,2,12

Insertar (7): $h_0(7) = 7 \ mod \ 5 = 2$

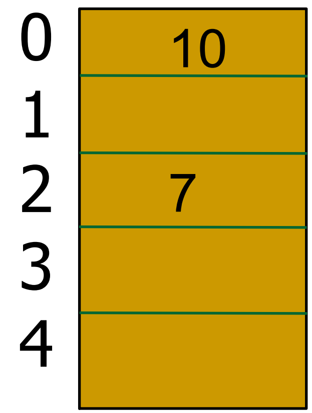


Algoritmos: Inserción

Insertar: 7,10,2,12

Insertar (7): $h_0(7) = 7\ mod\ 5 = 2$

Insertar (10): $h_0(10) = 10\ mod\ 5 = 0$



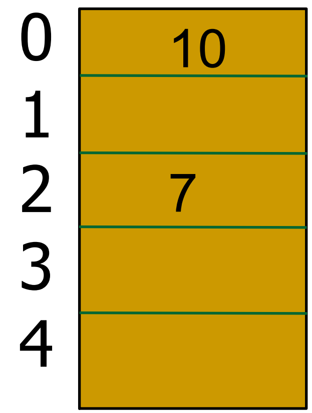
Algoritmos: Inserción

Insertar: 7,10,2,12

Insertar (7): $h_0(7) = 7 \, mod \, 5 = 2$

Insertar (10): $h_0(10) = 10 \, mod \, 5 = 0$

Insertar (2): $h_0(2) = 2 \, mod \, 5 = 2$



Algoritmos: Inserción

Insertar: 7,10,2,12

Insertar (7): $h_0(7) = 7 \; mod \; 5 = 2$

Insertar (10): $h_0(10) = 10 \; mod \; 5 = 0$

Insertar (2): $h_0(2) = 2 \; mod \; 5 = 2$

$h_1(2) = (2 + 1) \; mod \; 5 = 3$

0	10
1	
2	7
3	2
4	

Algoritmos: Inserción

Insertar: 7,10,2,12

Insertar (7): $h_0(7) = 7\ mod\ 5 = 2$

Insertar (10): $h_0(10) = 10\ mod\ 5 = 0$

Insertar (2): $h_0(2) = 2\ mod\ 5 = 2$

$h_1(2) = (2 + 1)\ mod\ 5 = 3$

Insertar (12): $h_0(12) = 12\ mod\ 5 = 2$

0	10
1	
2	7
3	2
4	

Algoritmos: Inserción

Insertar: 7,10,2,12

Insertar (7): $h_0(7) = 7\,mod\,5 = 2$

Insertar (10): $h_0(10) = 10\,mod\,5 = 0$

Insertar (2): $h_0(2) = 2\,mod\,5 = 2$

$h_1(2) = (2+1)\,mod\,5 = 3$

Insertar (12): $h_0(12) = 12\,mod\,5 = 2$

$h_1(12) = (12+1)\,mod\,5 = 3$

0	10
1	
2	7
3	2
4	

Algoritmos: Inserción

Insertar: 7,10,2,12

Insertar (7): $h_0(7) = 7 \, mod \, 5 = 2$

Insertar (10): $h_0(10) = 10 \, mod \, 5 = 0$

Insertar (2): $h_0(2) = 2 \, mod \, 5 = 2$

$h_1(2) = (2 + 1) \, mod \, 5 = 3$

Insertar (12): $h_0(12) = 12 \, mod \, 5 = 2$

$h_1(12) = (12 + 1) \, mod \, 5 = 3$

$h_2(12) = (12 + 2) \, mod \, 5 = 4$

0	10
1	
2	7
3	2
4	12

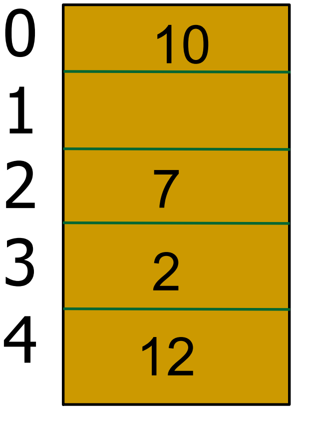
Algoritmos: búsqueda

```
buscar (k,T)
   i=0;
   mientras ((k!= T[h(k, i)].clave) y T[h(k,i)]!=null
   e (i<|T|)) incrementar i;
   Si (i < |T|) y T[h(k, i)]!=null entonces T[h(k,i)]
   en casocontrario <no está>
```

- Ojo: T[h(k, i)]!=null ¿Entonces, cómo borramos?
- Podemos marcar los elementos como “borrados, en lugar de “null”, pero....

Algoritmos: búsqueda

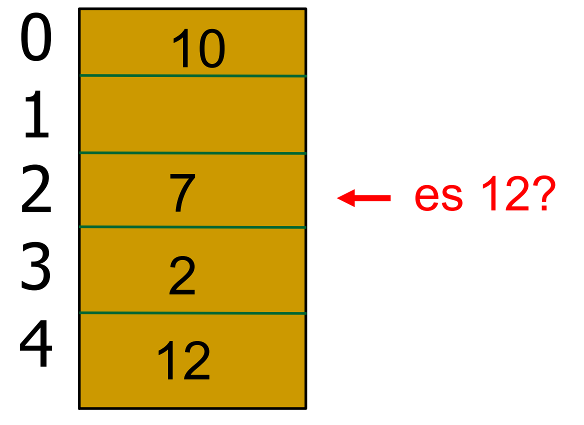
Buscar: 12,30



Algoritmos: búsqueda

Buscar: 12,30

Buscar (12): $h_0(12) = 12\ mod\ 5 = 2$

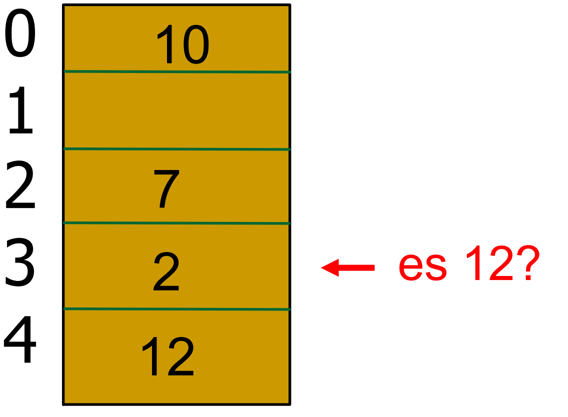


Algoritmos: búsqueda

Buscar: 12,30

Buscar (12): $h_0(12) = 12\,mod\,5 = 2$

$h_1(12) = (12 + 1)\,mod\,5 = 3$



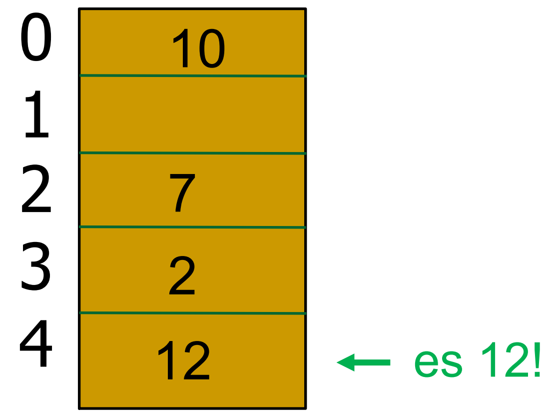
Algoritmos: búsqueda

Buscar: 12,30

Buscar (12): $h_0(12) = 12\ mod\ 5 = 2$

$h_1(12) = (12 + 1)\ mod\ 5 = 3$

$h_2(12) = (12 + 2)\ mod\ 5 = 4$



Algoritmos: búsqueda

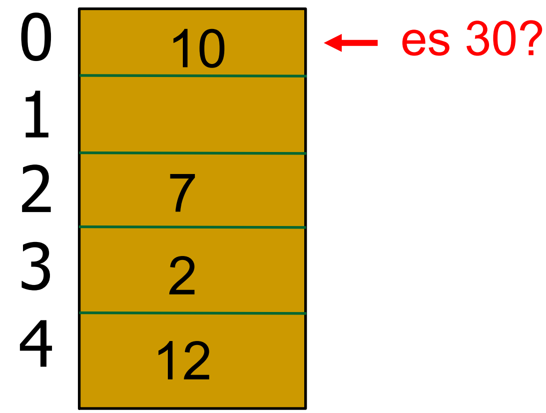
Buscar: 12,30

Buscar (12): $h_0(12) = 12\ mod\ 5 = 2$

$h_1(12) = (12 + 1)\ mod\ 5 = 3$

$h_2(12) = (12 + 2)\ mod\ 5 = 4$

Buscar (30): $h_0(30) = 30\ mod\ 5 = 0$



Algoritmos: búsqueda

Buscar: 12,30

Buscar (12): $h_0(12) = 12 \; mod \; 5 = 2$

$h_1(12) = (12 + 1) \; mod \; 5 = 3$

$h_2(12) = (12 + 2) \; mod \; 5 = 4$

Buscar (30): $h_0(30) = 30 \; mod \; 5 = 0$

$h_1(30) = (30 + 1) \; mod \; 5 = 1$

0	10	
1		← es NULL!
2	7	
3	2	
4	12	

Algoritmos: búsqueda

Buscar: 12,30

Buscar (12): $h_0(12) = 12\ mod\ 5 = 2$

$h_1(12) = (12 + 1)\ mod\ 5 = 3$

$h_2(12) = (12 + 2)\ mod\ 5 = 4$

0	10
1	
2	7
3	2
4	12

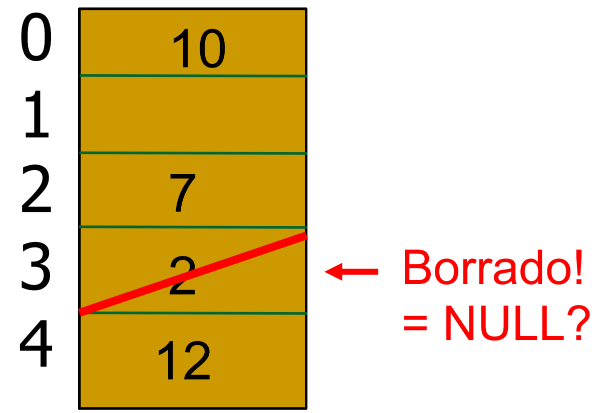
Algoritmos: búsqueda

Buscar: 12,30

Buscar (12): $h_0(12) = 12\,mod\,5 = 2$

$h_1(12) = (12 + 1)\,mod\,5 = 3$

$h_2(12) = (12 + 2)\,mod\,5 = 4$



Algoritmos: búsqueda

Buscar: 12,30

Buscar (12): $h_0(12) = 12 \; mod \; 5 = 2$

$h_1(12) = (12 + 1) \; mod \; 5 = 3$

$h_2(12) = (12 + 2) \; mod \; 5 = 4$

0	10
1	
2	7
3	~~2~~
4	12

Algoritmos: búsqueda

```
buscar (k,T)
  i=0;
  mientras ((k!= T[h(k, i)].clave) y T[h(k,i)]!=null
  e (i<|T|)) incrementar i;
  Si (i < |T|) y T[h(k, i)]!=null entonces T[h(k,i)]
  en casocontrario <no está>
```

- Ojo: T[h(k, i)]!=null ¿Entonces, cómo borramos?
- Podemos marcar los elementos como “borrados, en lugar de “null”, pero....

Barrido

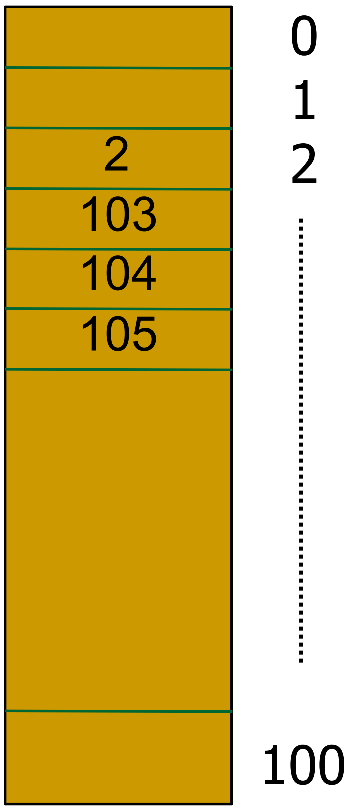
- La función h(k, i) debe recorrer todas las posiciones de la tabla
- Varias formas típicas para la función h(k,i)
  - Barrido linear
  - Barrido cuadrático
  - Hashing doble
- Se diferencian entre sí por su complejidad de cálculo y por el comportamiento respecto a los fenómenos de aglomeración.

Barrido linear

- $h(k, i) = (h'(k)+i) \bmod |T|$, donde $h'(k)$ es una función de hashing
- Se recorren todas las posiciones en la secuencia $T[h'(k)]$, $T[h'(k)+1]$, ... $T[|T|]$, 0, 1, ..., $T[h'(k)-1]$
- Posibilidad de aglomeración <u>primaria</u>: dos secuencias de barrido que tienen una colisión, siguen colisionando
- Los elementos se aglomeran por largos tramos

Aglomeración primaria

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



Barrido cuadrático

- $h(k, i) = (h'(k)+c_1i+c_2i^2) \bmod |T|$, donde $h'(k)$ es una función de hashing, c_1 y c_2 son constantes
- Ejemplos:
  - $h(k, i) = h'(k)+i^2$, $h(k, i+1) = h'(k)+(i+1)^2$, $i=1,..., (|T|-1)/2$
  - $h(k, i) = h'(k)+i/2+i^2/2$, $|T|=2^x$
- Posibilidad de aglomeración <u>secundaria</u>: si hay colisión en el primer intento....sigue habiendo colisiones ($h'(k_1)= h'(k_2) \rightarrow h'(k_1,i)= h'(k_2,i)$)
- Describir $h(k, i)$ cuando $h'(k)=k \bmod |5|$

Hashing doble

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

Construcción de funciones de Hash

- 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

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

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

Ascii('t')=116

Ascii('p')=112

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

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

$$\underbrace{477}\underbrace{2\ 64}\underbrace{53\ 7}\underbrace{348} \rightarrow \{477, 264, 537, 348\}$$

$$f(477,264,537,348) = (477+264+537+348) \bmod 701$$
$$= 224$$

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

- El número obtenido puede ser manipulado ulteriormente
- La dirección puede depender de una parte de la clave