

Hashing y tablas de *hash*

Estructuras de Datos y Algoritmos

Un **diccionario**
es una
estructura de
datos con las
siguientes
operaciones

Asociar un **valor** (p.ej., un archivo con la solución de la tarea 1)
a una **clave** (p.ej., un rut o número de alumno)

... o **actualizar** el valor asociado a la clave (p.ej., cambiar el
archivo)

Obtener el **valor** asociado a una **clave**

(... y en ciertos casos)

Eliminar del diccionario una **clave** y su **valor** asociado

Así, la idea de un diccionario es ...

... si me dan el rut (la clave), entonces yo quiero encontrar el archivo

.... si me dan el rut y me doy cuenta de que ese rut no está en mis registros (el diccionario), entonces ingresar el rut a mis registros

... si me dan el rut y me doy cuenta de que no hay un archivo asociado, entonces asociar un archivo al rut

... si me dan el rut y me doy cuenta de que tiene un archivo asociado, entonces cambiar el archivo por uno más actual

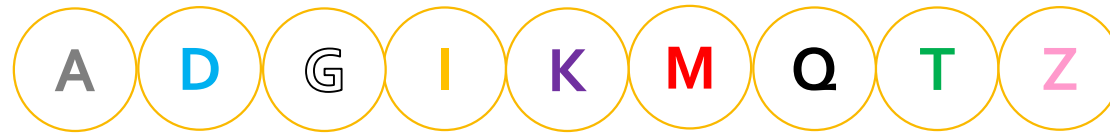
La búsqueda es
lo primero ... y,
si lo buscado
no está,
entonces
(posiblemente)
la inserción es
lo segundo

O sea, a partir de la clave, lo primero es buscarla (eficientemente) en el diccionario

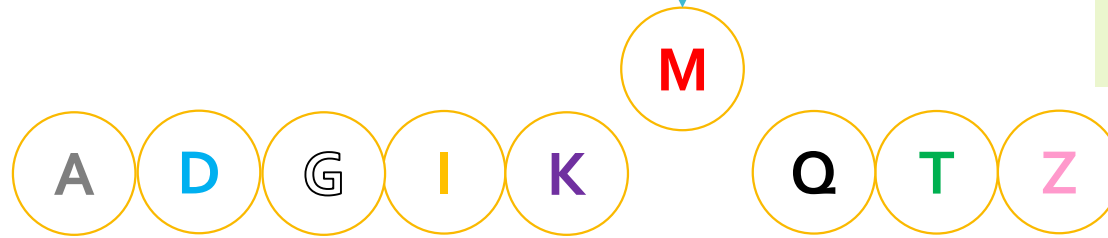
Si la encontramos, entonces ahora tenemos acceso a la información asociada a la clave

... si no la encontramos, entonces, tal vez, queremos ingresarla al diccionario junto al resto de la información correspondiente

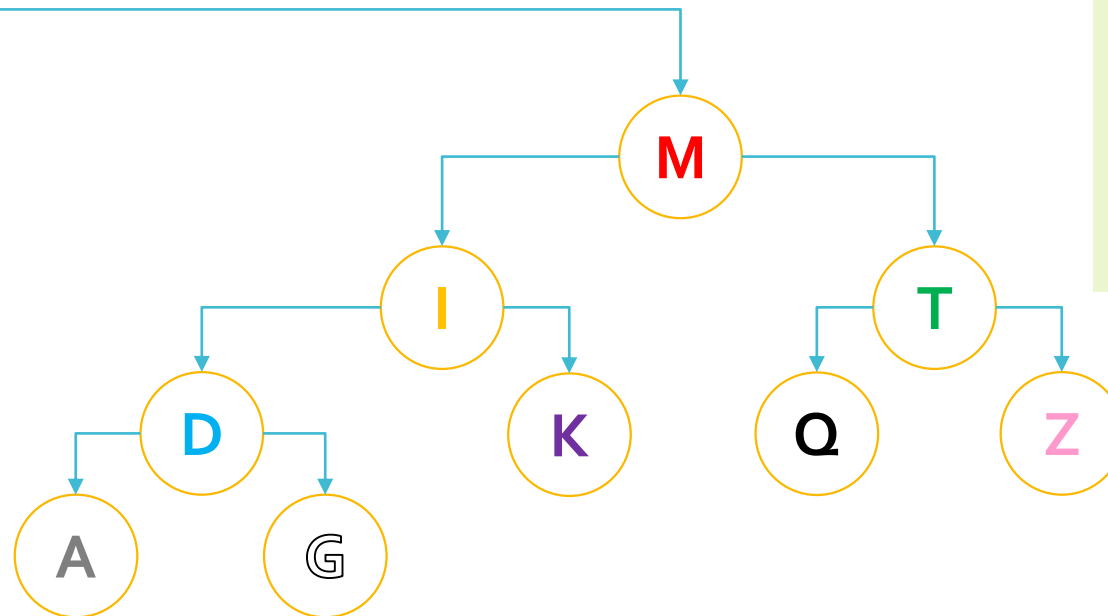
Lo que hicimos
anteriormente,
a partir de las
ventajas y
desventajas de
arreglos y listas
ligadas ...



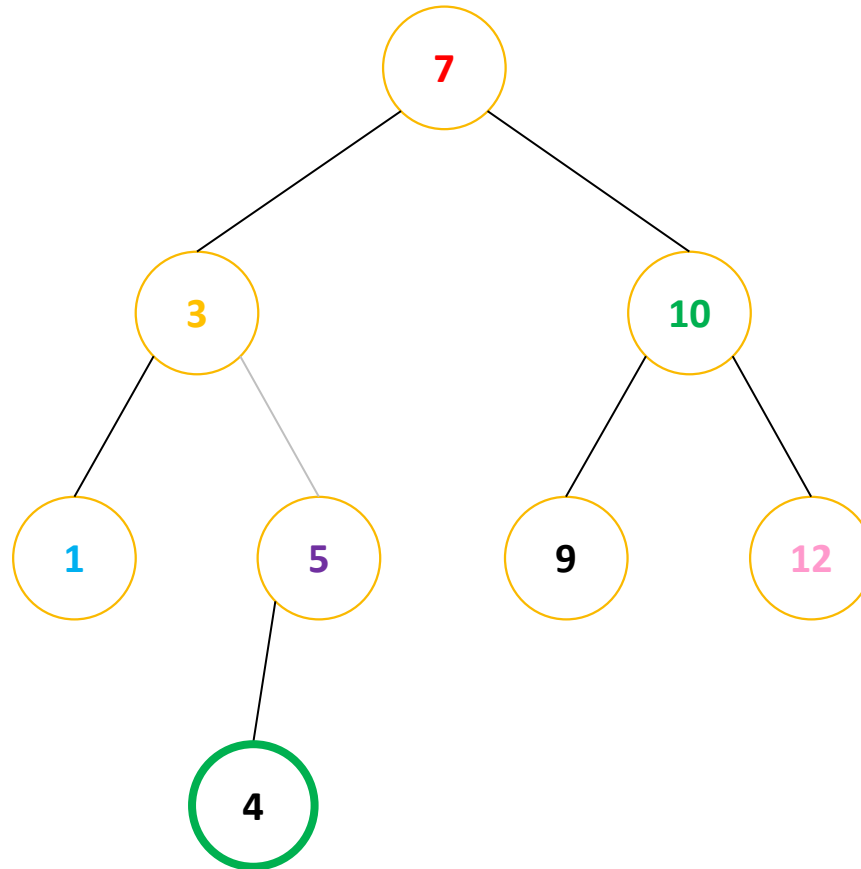
Podemos tener un puntero a un elemento más o menos en el centro de la lista



... y ese elemento puede tener punteros a elementos más o menos en el centro de cada una de las dos sublistas, a su izquierda y a su derecha; ... y así recursivamente



... es que
implemen-
tamos un
diccionario
como un ABB



Los nodos del árbol contienen
las claves

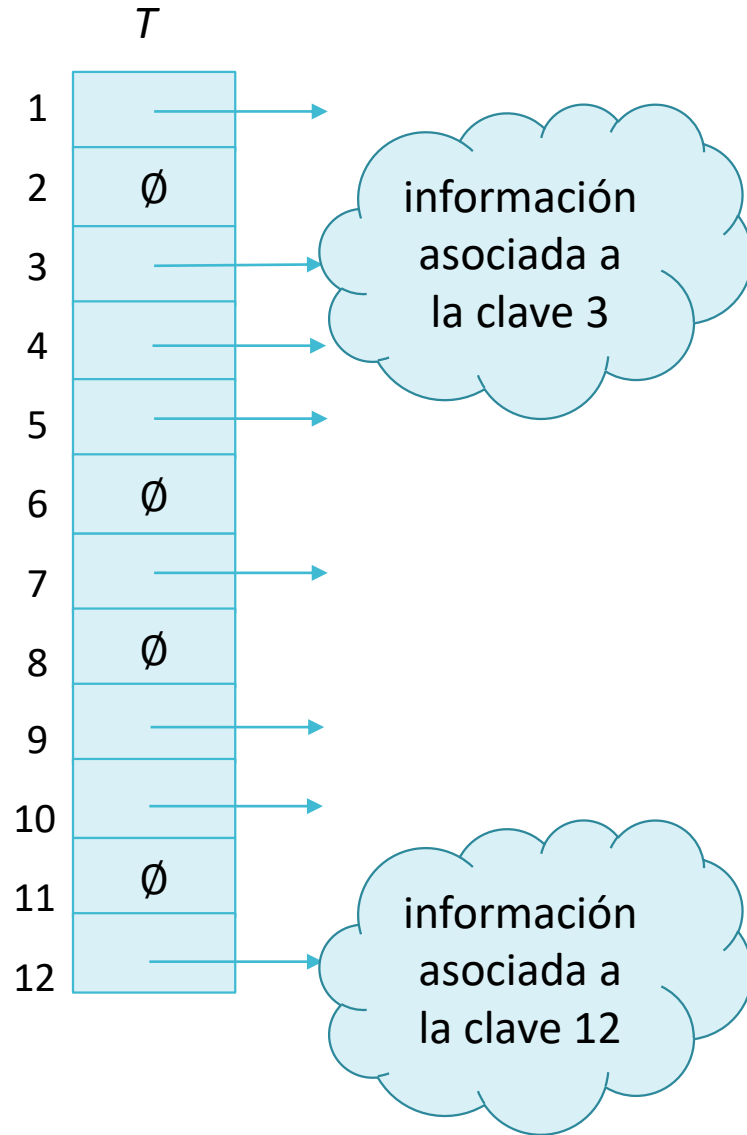
... que están ***totalmente
ordenadas***

(además, contienen punteros

... tanto para mantener el árbol
unido y poder recorrerlo —al
buscar—

... como para tener acceso a la
información asociada a la clave)

Si las claves que nos interesan realmente fueran los números del 1 al 12, podríamos usar un arreglo T (de punteros)



La búsqueda de (el objeto con) la clave k es ahora muy rápida:

- $T[k] = \emptyset \Rightarrow$ el objeto no está
- $T[k] \neq \emptyset \Rightarrow$ el objeto está

No es necesario almacenar las claves:

- son los índices del arreglo

... y no es necesario almacenar punteros a hijos o padres:

- sólo a la información asociada a la clave
- para recorrer el arreglo sólo hace falta cambiar el valor del índice

Esto funciona bien para ciertos tipos de claves

Números enteros no negativos en un rango razonable, tal que puedan ser usadas como índices de arreglos:

- p.ej., en IIC2343 hay 220 estudiantes este semestre
... si el profesor, los ayudantes y cada uno de los estudiantes supiera su posición en la lista del curso, entonces podríamos usar ese número como clave y como índice del arreglo T (de punteros)
- p.ej., en la universidad hay aprox. 25,000 estudiantes este año
... si a cada uno se le asignara un número único entre 1 y 25,000 y todos los involucrados supiéramos esos números, entonces podríamos usar esos números como claves y como índices del arreglo T

Pero la realidad
de las claves ...

¿Qué pasa si las claves son los rut's de las personas?

Pero la realidad
de las claves
hace que
usarlas
directamente
como índices
del arreglo ...

P.ej., en el caso de los estudiantes de la universidad:

- el rango de los rut's abarca hasta el número 25,000,000
... pero la universidad sólo tiene 25,000 estudiantes
... → en promedio, sólo una de cada 1,000 casillas del arreglo estaría ocupada

¿Y si las claves fueran los números de los teléfonos celulares

... o, en otro contexto, las patentes de los vehículos?

Pero la realidad
de las claves
hace que
usarlas
directamente
como índices
del arreglo no
sea práctico

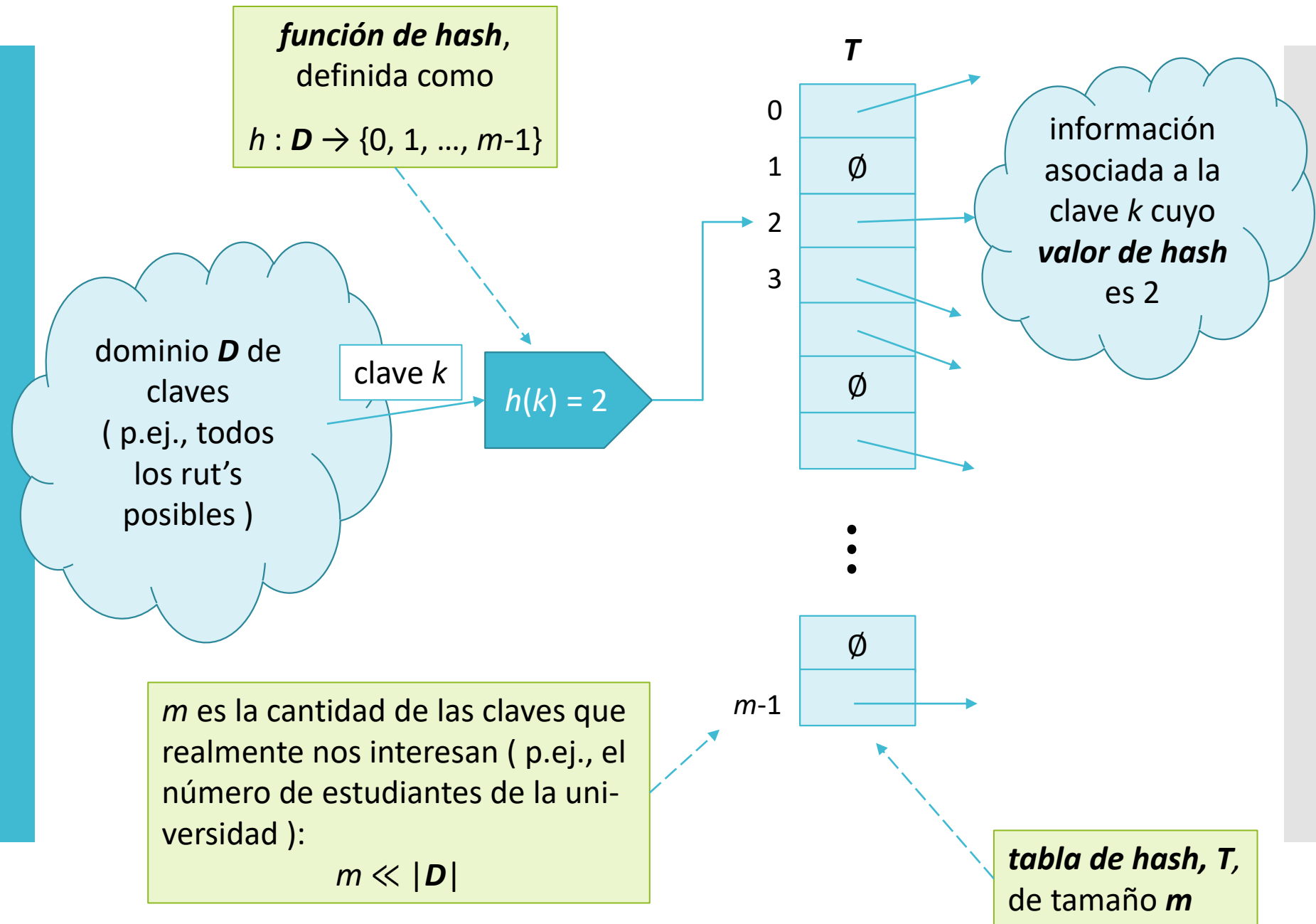
Según la naturaleza de los datos, las claves pueden pertenecer a dominios muy variados

... y ser de cardinalidades muy grandes en comparación a la cantidad de datos que nos interesan:

- p.ej., si las claves fueran strings de 10 letras mayúsculas
... entonces habría 26^{10} = algo más de 140 millones de millones (1.4×10^{14}) claves distintas

La solución es usar **hashing**:

- la clave no se usa directamente como índice
- el índice se **calcula** a partir de la clave



Propiedades de hashing: funciona “casi” como un arreglo

Hashing se comporta “casi” como un arreglo:

- en un arreglo, buscar el dato con clave k consiste simplemente en mirar $T[k]$ → es $O(1)$ (diap. 7)
- en hashing, buscar el dato con clave k consiste en mirar $T[h(k)]$ → es $O(1)$ pero sólo **en promedio**, como vamos a ver

Supongamos
que las claves k
son los RUT's y
que son de la
forma

$xx.yyy.zzz-v$

La función de hash h se calcula cada vez que se hace una consulta a la tabla

... → debe ser rápida de calcular

P.ej. (m es el tamaño de la tabla):

- si $m = 100$, podríamos usar $h(k) = xx$
- si $m = 1000$, podríamos usar $h(k) = yyy$
- si $m = 1000$, podríamos usar $h(k) = zzz$

Veamos ...

Propiedades de hashing: el orden de las claves no importa

En hashing el orden relativo de las claves **no importa**:

- comparar claves entre ellas (para determinar cuál es mayor)
... o, dada una clave, encontrar la clave predecesora o la sucesora
... **no son** operaciones de diccionario (diap. 2)

(En este sentido, los ABBs son en realidad diccionarios con operaciones adicionales:

- ABB = diccionario + *cola de prioridades*)

Una típica función de hash:

$$h(k) = k \bmod m$$

hashing modular

... es decir, **el resto de la división (entera) de k por m**

... que efectivamente es un valor entre 0 y $m-1$

En el ej. a la derecha, el dominio son los números entre 000 y 999

... y se muestran los valores de hash para algunas claves cuando $m = 100$ y cuando $m = 97$

k	$h(k)$ $m=100$	$h(k)$ $m=97$
212	12	18
618	18	36
302	2	11
940	40	67
702	2	23
704	4	25
612	12	30
606	6	24
772	72	93
304	4	13
423	23	35
650	50	68
317	17	26
907	7	34
507	7	22

Si $k_1 \neq k_2$, pero
 $h(k_1) = h(k_2)$,
entonces
tenemos una
colisión (en la
tabla de hash)

Como vemos en el ej. anterior, en particular en la columna para $m = 100$, en hashing pueden ocurrir **colisiones**:

$$h(212) = 12 \text{ y } h(612) = 12$$

$$h(907) = 7 \text{ y } h(507) = 7$$

Es decir, para datos con distintas claves, su ubicación en la tabla es (o debería ser) la misma:

- ¿cómo podemos guardar ambos datos en la tabla?
- nos interesa poder buscarlos en el futuro

Como la cardinalidad del dominio, $|D|$, es mucho mayor que el tamaño m de la tabla, las colisiones potenciales son inevitables

En los ejemplos
que siguen:

- las claves son
números enteros
pequeños
- la tabla tiene
 $m = 7$ casillas
(índices 0 a 6)

Insertemos la clave 15:

$$h(15) = 15 \bmod 7 = 1$$

nil	nil	nil	nil	nil	nil	nil
0	1	2	3	4	5	6

(el dato con) la
clave 15 queda
en la casilla con
índice 1

Insertemos la clave 15:

$$h(15) = 15 \bmod 7 = 1$$

nil	15	nil	nil	nil	nil	nil
0	1	2	3	4	5	6

Similarmente
para la clave 37
y la casilla con
índice 2

Insertemos la clave 37:

$$h(37) = 37 \bmod 7 = 2$$

nil	15	37	nil	nil	nil	nil
0	1	2	3	4	5	6

¿Qué hacemos
si una nueva
clave debería
quedar en una
casilla que ya
está ocupada?
→ **Colisión**: dos
posibilidades

Insertemos la clave 51:

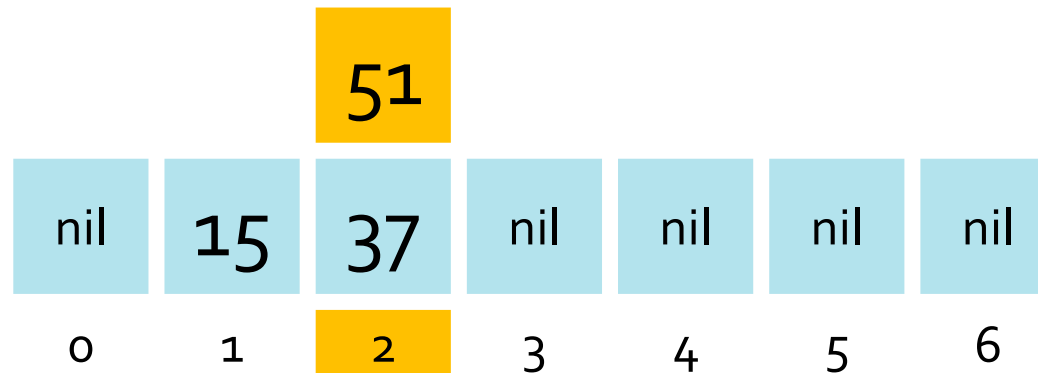
$$h(51) = 51 \bmod 7 = 2$$

nil	15	37	nil	nil	nil	nil
0	1	2	3	4	5	6

Una posibilidad
es usar **enca-**
denamiento:
hacer una lista
con las claves
que van a una
misma casilla

Insertemos la clave 51:

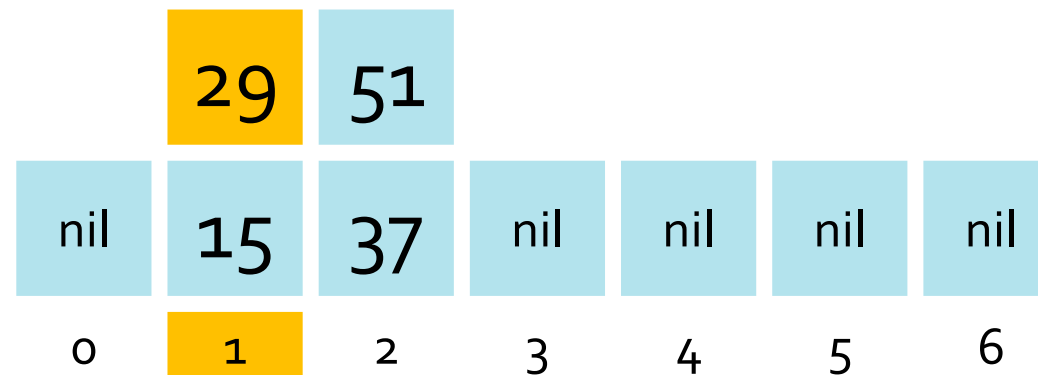
$$h(51) = 51 \bmod 7 = 2$$



Similarmente
para la clave 29
y la casilla con
índice 1

Insertemos la clave 29:

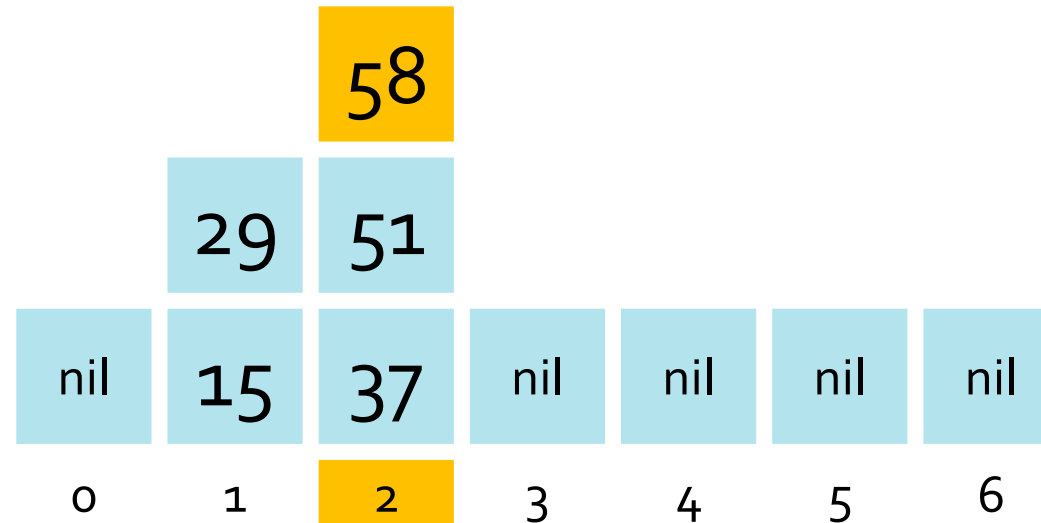
$$h(29) = 29 \bmod 7 = 1$$



Y similarmente
para la clave 58
y nuevamente
la casilla con
índice 2

Insertemos la clave 58:

$$h(58) = 58 \bmod 7 = 2$$



¿Cómo buscamos una clave en una tabla de hash con encade- namiento?

Para buscar (el dato con) la clave k , primero calculamos el valor de hash $h(k)$ y miramos la casilla $T[h(k)]$:

- si $T[h(k)]$ está vacía, entonces la clave k no está en T
- si $T[h(k)]$ no está vacía, entonces apunta a una lista ligada de una o más claves, todas distintas entre ellas, pero que tienen el mismo valor de hash que la clave k

... por lo tanto, buscamos k en esta lista, p.ej., secuencialmente (k puede estar o no en la lista)

¿Cómo
eliminamos
una clave de
una tabla de
hash con enca-
denamiento?

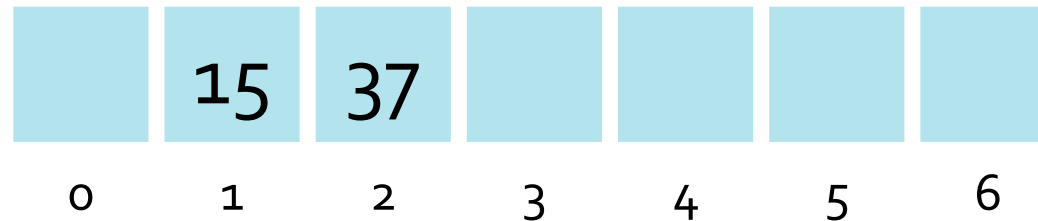
Para eliminar la clave k —suponiendo que la buscamos y la encontramos— simplemente la sacamos de la lista ligada

Otra forma de manejar colisiones es **direccionamiento abierto**: buscar sistemáticamente una casilla vacía

Insertemos las claves 15 y 37:

$$h(15) = 15 \bmod 7 = 1$$

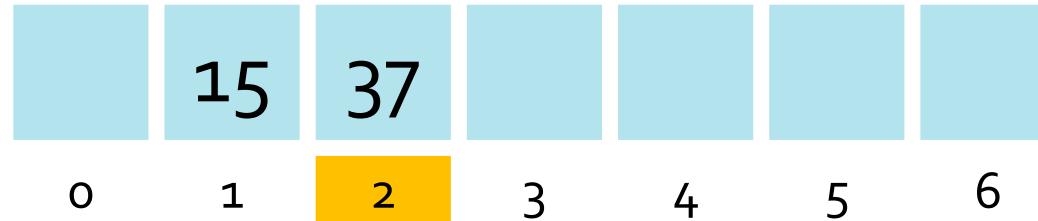
$$h(37) = 37 \bmod 7 = 2$$



La clave 51
debería ir a
parar a la
casilla con
índice 2, que
ya está
ocupada

Insertemos la clave 51:

$$h(51) = 51 \bmod 7 = 2$$



... entonces la
ponemos en la
*primera casilla
desocupada a
la derecha:*
sondeo lineal

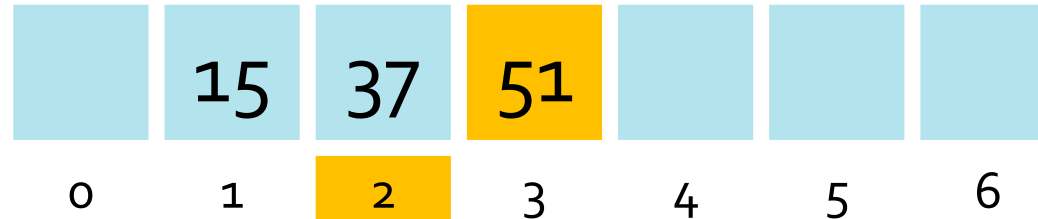
Insertemos la clave 51:

$$h(51) = 51 \bmod 7 = 2$$

Sondeo lineal: a partir de la casilla identificada por el valor de hash de la clave (en este ej., la casilla 2)

... vamos mirando una por una las casillas a la derecha

... hasta que encontremos la primera casilla desocupada, y ahí ponemos la clave (en este ej., la casilla 3)



Ahora, la clave 29 debería ir a la casilla con índice 1, que también ya está ocupada

Insertemos la clave 29:

$$h(29) = 29 \bmod 7 = 1$$

	15	37	51			
0	1	2	3	4	5	6

... empleando
sondeo lineal,
ponemos la
clave 29 en la
primera casilla
desocupada a
la derecha

Insertemos la clave 29:

$$h(29) = 29 \bmod 7 = 1$$

Empleando sondeo lineal, después de mirar la casilla 1, que está ocupada, miramos las casillas 2, que también está ocupada, y 3, lo mismo

... así que llegamos hasta la casilla 4, que está vacía, y ponemos ahí la clave 29

	15	37	51	29		
0	1	2	3	4	5	6

La búsqueda
bajo sondeo
lineal sigue la
misma
secuencia de
comparaciones
que al insertar

	15	37	51	29		
0	1	2	3	4	5	6

Buscamos la clave 29:

$$h(29) = 29 \bmod 7 = 1$$

Por lo tanto, miramos primero la casilla 1, y de ahí hacia la derecha las casilla 2, 3 y 4, donde finalmente encontramos la clave 29

	15	37	51	29		
0	1	2	3	4	5	6

Si durante una búsqueda llegamos a una casilla vacía, significa que la clave que buscamos no está en la tabla

	15	37	51	29		
0	1	2	3	4	5	6

Buscamos la clave 10:

$$h(10) = 10 \bmod 7 = 3$$

Por lo tanto, miramos primero la casilla 3, y de ahí hacia la derecha las casillas 4 y 5; como la casilla 5 está vacía, deducimos que la clave 10 **no está almacenada** en la tabla, porque de lo contrario, tendría que haber estado en esta casilla

	15	37	51	29		
0	1	2	3	4	5	6

La eliminación en este caso es problemática: si es necesario poder eliminar claves, entonces es mejor usar encadenamiento

	15	37	51	29		
0	1	2	3	4	5	6

Eliminamos primero la clave 51

... y buscamos luego la clave 29:

$$h(29) = 29 \bmod 7 = 1$$

Miramos primero la casilla 1, y luego las casillas 2 y 3; como la casilla 3 está ahora vacía, deducimos **erróneamente** que la clave 29 no está almacenada en la tabla

	15	37		29		
0	1	2	3	4	5	6

Con
direcciona-
miento abierto,
podemos
emplear otras
políticas de
sondeo

Sondeo lineal (el que vimos recién): si $h(k) = H$, entonces

- buscamos en $H, H + 1, H + 2, H + 3, \dots$
- también puede ser $H, H + d, H + 2d, H + 3d, \dots$, en que d no tiene factores comunes con m

Sondeo cuadrático: si $h(k) = H$, entonces

- buscamos en $H, H + 1, H + 4, H + 9, \dots$

Doble hashing:

- ocupamos dos funciones de hash, $h_1(k)$ y $h_2(k)$
- buscamos en $h_1(k), h_1(k) + h_2(k), h_1(k) + 2h_2(k), h_1(k) + 3h_2(k), \dots$
- p.ej., $h_1(k) = k \bmod m$ y $h_2(k) = 1 + k \bmod (m-2)$, en que m y $m-2$ son *primos gemelos*

En cualquiera de estos casos, el problema al eliminar claves se manifiesta igual

¿Qué tan llena está la tabla?

Si una tabla de m casillas tiene almacenados n datos

... entonces definimos el **factor de carga** λ como

$$\lambda = \frac{n}{m}$$

- con encadenamiento, es aceptable $\lambda \approx 1$ (o incluso un número entero pequeño mayor que 1)
- con direccionamiento abierto, $\lambda > 0.5$ resulta en inserciones y búsquedas muy lentas

Rehashing

Cuando la tabla se empieza a llenar demasiado —las búsquedas e inserciones se empiezan a demorar más de lo aceptable—

... hay que construir otra tabla —aproximadamente el doble de grande— y definir una nueva función de hash para esta tabla

... y pasar todos los datos de la tabla original a la nueva tabla —calculando el nuevo valor de hash para cada uno

Esta es una operación cara — $O(n)$ — pero infrecuente:

- tienen que haber habido $O(n)$ inserciones en la tabla original

... por lo que en esencia estamos agregando un costo constante a cada inserción (por eso la nueva tabla es el doble de grande)

Funciones de hash

Claves numéricas:

- la función de hash debe convertir la clave a un número entero entre 0 y $m-1$
- las claves pueden ser números enteros o bien números reales

Claves no numéricas:

- la función de hash debe, primero, convertir la clave a un número (entero)
- ... y, luego, convertir este número a un número entero entre 0 y $m-1$

La función de hash debe ser fácil de calcular y debe distribuir las claves uniformemente en la tabla

Si las claves k son número enteros, entonces

$$h(k) = k \bmod m$$

—llamado *hashing modular*— en que m es el tamaño de la tabla, es generalmente una función razonable:

- es conveniente que m sea un número primo (próx. diap.)

Si las claves son (idealmente) números aleatorios

... entonces esta función es simple de calcular

... y distribuye las claves uniformemente a lo largo de la tabla

Para mejorar la distribución, conviene que la función dependa de todos los dígitos de la clave

Como vemos en el ej., si $m = 100 (= 10^2)$, entonces el valor de la función depende sólo de (y es igual a) los dos dígitos menos significativos de la clave

... y si, p.ej., $m = 128 (= 2^7)$, entonces el valor de la función va a depender sólo de los 7 bits menos significativos de la clave

En general, si m es un múltiplo de 2, 3, 5 o cualquiera otra constante pequeña

... entonces la distribución de las claves en la tabla no va a ser muy uniforme (al menos para ciertos conjuntos de claves)

m debería ser un número primo cercano al tamaño que queríamos para la tabla

k	$h(k)$ $m=100$	$h(k)$ $m=97$
2 12	12	18
618	18	36
3 02	2	11
940	40	67
7 02	2	23
7 04	4	25
6 12	12	30
606	6	24
772	72	93
3 04	4	13
423	23	35
650	50	68
317	17	26
9 07	7	34
5 07	7	22

Otra
posibilidad
para números
enteros: el
método de la
multiplicación

Sea A un número entre 0 y 1:

$$h(k) = \lfloor m \cdot (A \cdot k \bmod 1) \rfloor$$

Es decir, multiplicamos k por A y extraemos la parte fraccional del producto (lo que queda a la derecha del punto decimal)

... éste valor lo multiplicamos por m y finalmente tomamos el piso del resultado

En este caso (a diferencia del anterior), el valor de m no es crítico

... y, de hecho, una forma de simplificar el cálculo es que m sea una potencia de 2

Precaución: la intuición nos puede jugar malas pasadas

Si las claves son números reales entre 0 y 1, intuitivamente podríamos multiplicar la clave por m y redondear al entero más cercano para obtener un índice entre 0 y $m-1$:

- $h(k) = \lfloor km \rfloor$
- el problema es que esto da más peso a los dígitos más significativos de la clave
 - ... los dígitos menos significativos (los que están más a la derecha) no influyen

Para resolver esto, podemos usar hashing modular sobre la representación binaria de la clave:

- (bit de signo &) 8 bits de exponente & 23 bits de fracción

Las claves no
siempre son
números
enteros

Si las claves son strings s , entonces

... primero hay que convertir s a un número entero k

... y luego ajustar k al tamaño m de la tabla

Una forma común es interpretar s como un número entero escrito en una base apropiada

Si $s = ch_0ch_1...ch_p$

... entonces una forma común de convertirlo a un número k (llamado *el valor de hash de s*) es

$$k = \#(ch_p) + \#(ch_{p-1}) \times R + \#(ch_{p-2}) \times R^2 + \dots + \#(ch_0) \times R^p$$

en que $\#(ch)$ es el valor ASCII del carácter ch

... y R es 31 o 37 (un número primo permite que los bits de todos los caracteres jueguen un rol):

- interpretamos s como un número entero de $p+1$ dígitos en base R

Finalmente, hay que convertir k a un índice i válido para la tabla, es decir, un número entre 0 y $m-1$: $i = k \bmod m$

Otros temas,
que no vamos
a ver

hashing universal

hashing perfecto