Hashing y tablas de hash

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

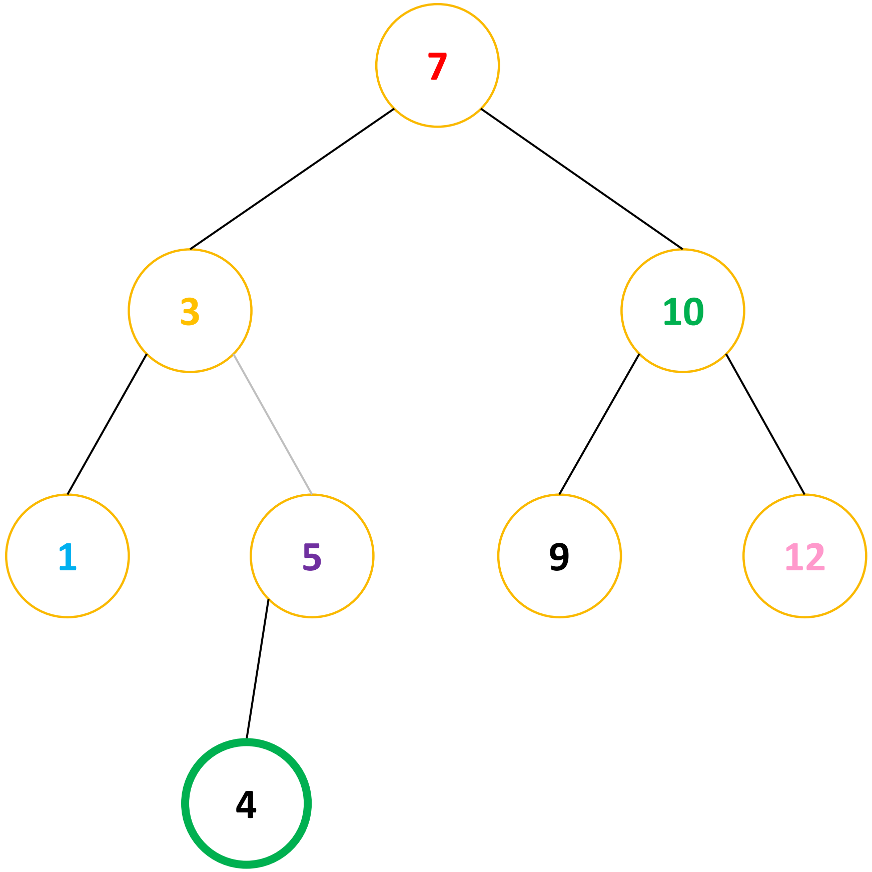
La búsqueda es lo primero … y, si no está, entonces (posiblemente) la inserción es lo segundo

O sea, a partir de la clave, lo primero es buscarla (eficiente-mente) 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

Implementa-mos un diccionario como un ABB



Los nodos del árbol contienen esencialmente las claves

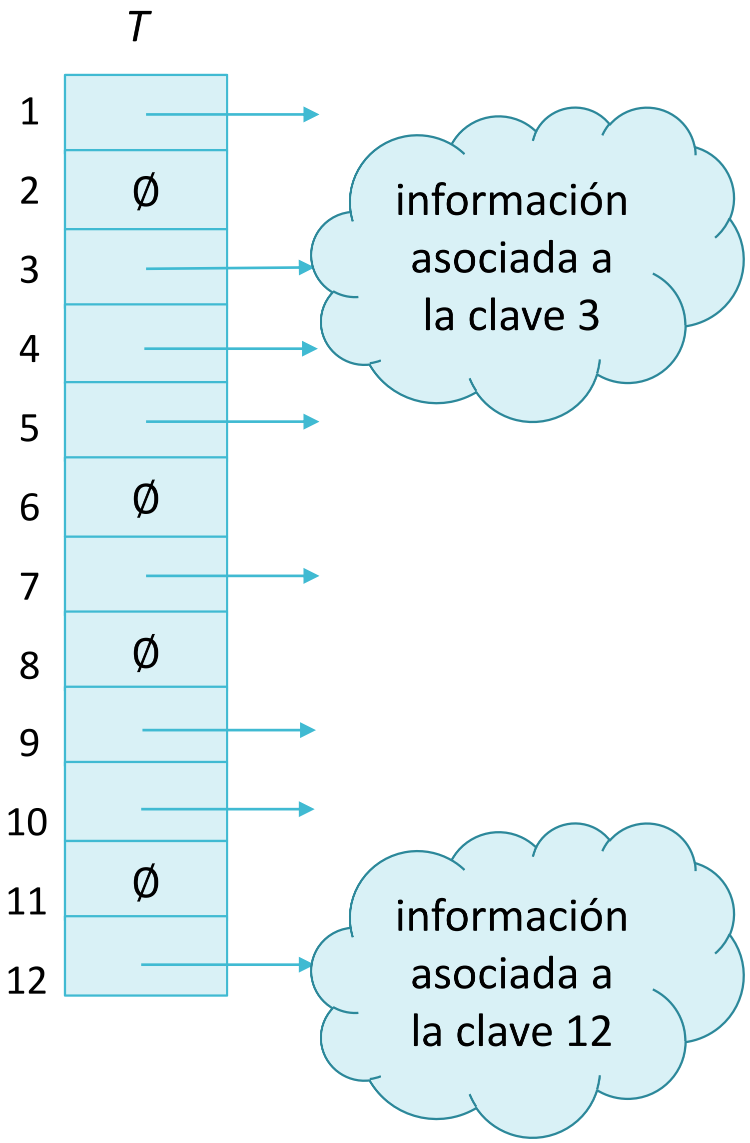
… que están totalmente ordenadas

(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 realmente fueran los números del 1 al 12, podría-mos 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

En principio, no es necesario almacenar las claves:

- son los índices del arreglo

… y no es necesario almacenar punteros a los hijos o al padre:

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

La realidad de las claves hace que usar las claves directamente como índice del arreglo no sea práctico

Incluso si las claves fueran números enteros mucho más grandes que 12, pero todavía en un rango razonable (p.ej., 0 a 65,535, es decir, 16 bits) podríamos usar un arreglo

Pero ¿qué pasa si las claves son los rut’s de las personas?

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

- el rango abarca hasta el número 25,000,000 (aprox.)

  … pero la universidad sólo tiene unos 25,000 estudiantes

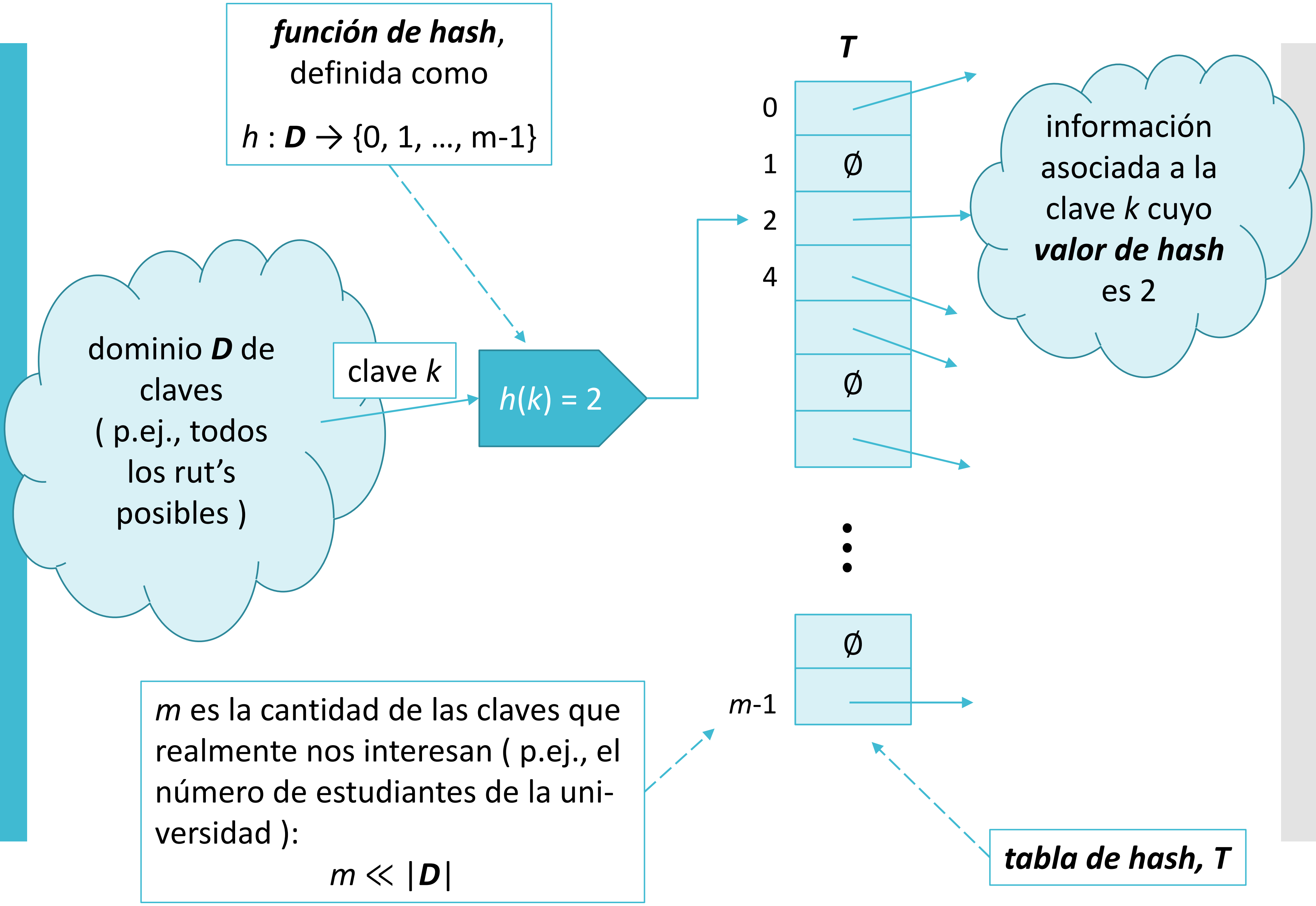
  … en promedio, sólo una de cada 1,000 casillas estaría ocupada

¿Y si las claves son los números de los teléfonos celulares?

Según la naturaleza de los datos, las claves pueden pertenecer a dominios muy variados y de cardinalidades muy grandes en comparación a la cantidad de datos que nos interesan

La solución es usar ***hashing***:

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



Algunas propiedades de hashing

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) (ver diap. # 6)
- 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

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

  … **no son** operaciones de diccionario (ver diap. # 2)

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

- ABB = diccionario + cola de prioridades **)**

Una típica fun-ción de hash:

$h(k) = k \bmod m$

… 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 observamos en el ej. anterior, en particular en la columna correspondiente a m = 100, en hashing pueden ocurrir colisiones:

$h(212) = 12$ y $h(612) = 12$

$h(907) = 7$ y $h(507) = 7$

Es decir, para datos con distintas claves, su ubicación en la tabla es la misma:

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

Como la cardinalidad del dominio, $|\boldsymbol{D}|$, 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 < 100 y la tabla tiene 7 casillas

$$m = 7$$

Insertemos la clave 15:

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

0	1	2	3	4	5	6

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

$$m = 7$$

Insertemos la clave 15:

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

	15					
0	1	2	3	4	5	6

…

$$m = 7$$

Insertemos la clave 37:

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

	15					
0	1	2	3	4	5	6

Similarmente para la clave 37 y la casilla con índice 2

$$m = 7$$

Insertemos la clave 37:

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

	15	37				
0	1	2	3	4	5	6

Ahora resulta que la clave 51 también debería ir a la casilla con índice 2

$$m = 7$$

Insertemos la clave 51:

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

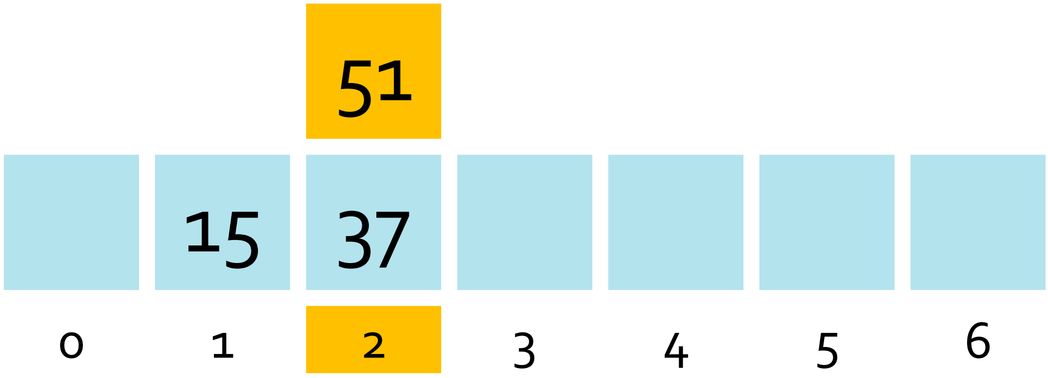
	15	37				
0	1	2	3	4	5	6

Usamos **encadena-miento**: formamos una lista con las claves que van a parar a la misma casilla

$$m = 7$$

Insertemos la clave 51:

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

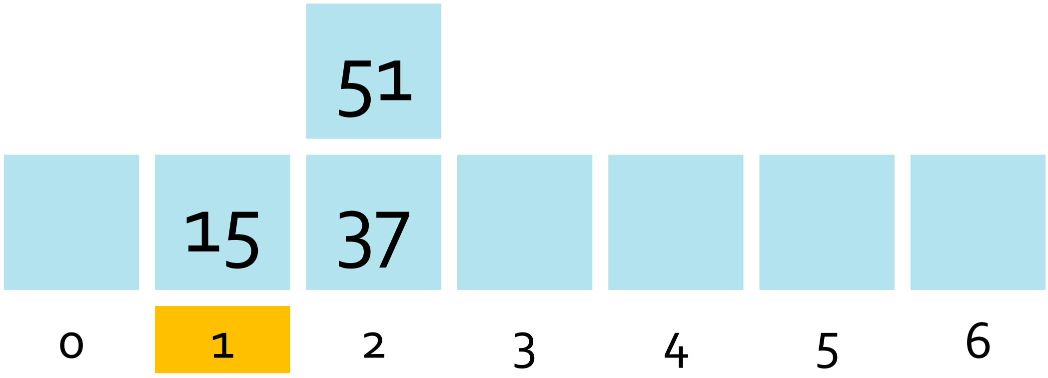


…

$$m = 7$$

Insertemos la clave 29:

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

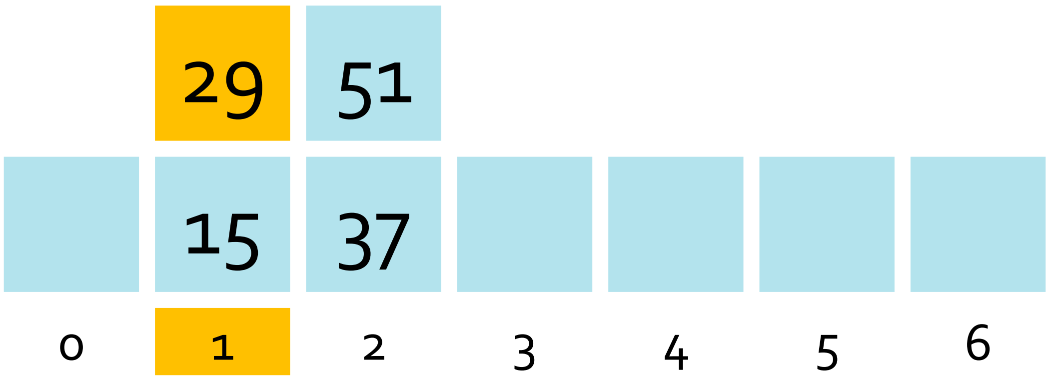


Similarmente para la clave 29 y la casilla con índice 1

$$m = 7$$

Insertemos la clave 29:

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

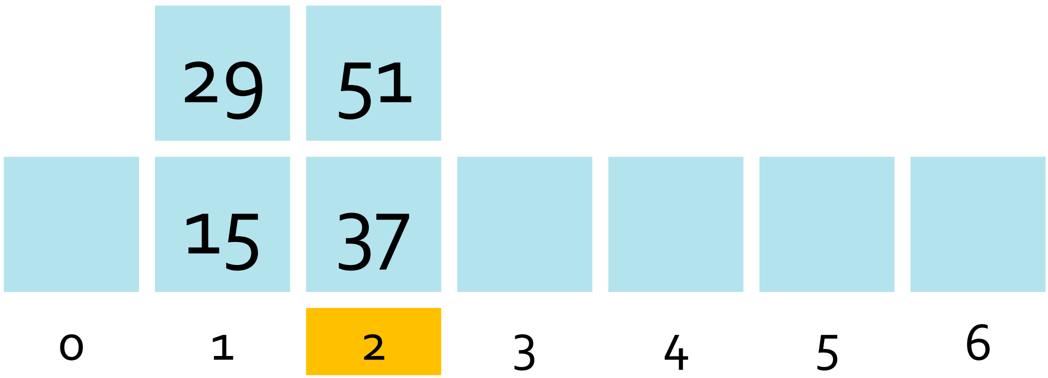


…

$$m = 7$$

Insertemos la clave 58:

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

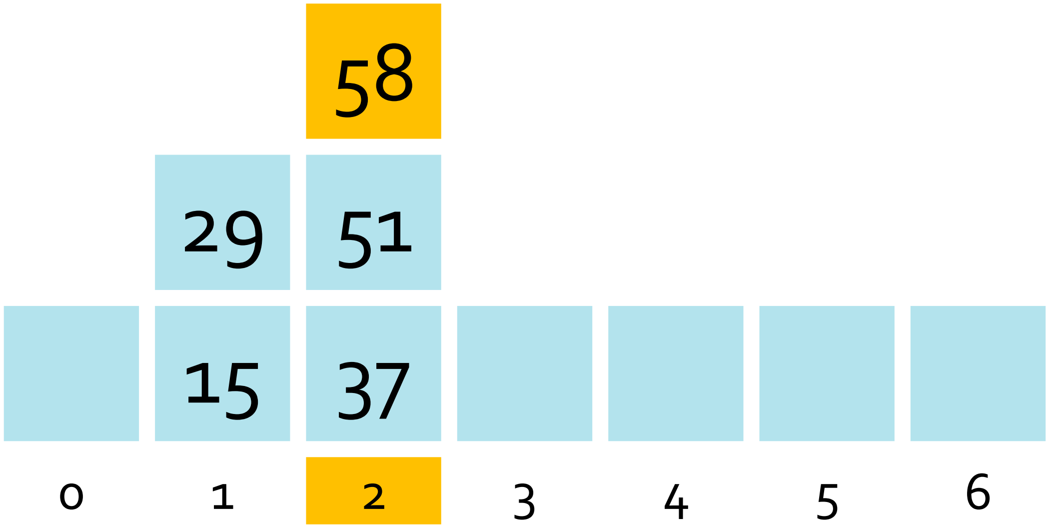


Y similarmente para la clave 58 y la casilla con índice 2

$$m = 7$$

Insertemos la clave 58:

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



Direcciona-miento abierto es otra posibilidad para resolver colisiones

$$m = 7$$

Insertemos la clave 15:

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

	15					
0	1	2	3	4	5	6

...

$$m = 7$$

Insertemos la clave 37:

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

	15					
0	1	2	3	4	5	6

…

$$m = 7$$

Insertemos la clave 37:

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

	15	37				
0	1	2	3	4	5	6

La clave 51 también debería ir a parar a la casilla con índice 2

$$m = 7$$

Insertemos la clave 51:

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

	15	37				
0	1	2	3	4	5	6

… pero la ponemos en la **primera casilla desocupada a la derecha:** ***sondeo lineal***

$$m = 7$$

Insertemos la clave 51:

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

	15	37	51			
0	1	2	3	4	5	6

La clave 29 debería ir a parar a la casilla con índice 1, ya ocupada con la clave 15

$$m = 7$$

Insertemos la clave 29:

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

	15	37	51			
0	1	2	3	4	5	6

... así que también la ponemos en la primera casilla desocupada a la derecha

$$m = 7$$

Insertemos la clave 29:

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

	15	37	51	29		
0	1	2	3	4	5	6

Búsqueda exitosa bajo (direcciona-miento abierto con) sondeo lineal

	15	37	51	29		
0	1	2	3	4	5	6

$$m = 7$$

¿Cómo buscamos la clave 29 con sondeo lineal?

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

	15	37	51	29		
0	1	2	3	4	5	6

Búsqueda de un dato que no está

	15	37	51	29		
0	1	2	3	4	5	6

$$m = 7$$

¿Cómo buscamos la clave 10 con sondeo lineal?

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

	15	37	51	29		
0	1	2	3	4	5	6

La eliminación es problemática

	15	37	51	29		
0	1	2	3	4	5	6

$$m = 7$$

Eliminemos la clave 51. ¿Cómo buscamos la clave 29 con sondeo lineal?

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

	15	37		29		
0	1	2	3	4	5	6