



UNIVERSITAT DE BARCELONA



Estructura de datos

Hash

Santi Seguí | 2014-15

Índice

1. Conjunto
2. Diccionario
3. Tabla Hash
 1. Hash Abierto
 2. Hash Cerrado
4. Funciones Hash

Conjunto (set)

- Un conjunto es una colección de elementos no repetidos
- A diferencia que la lista o array, un conjunto **no mantiene ningún orden** de sus elementos.



Conjunto ADT

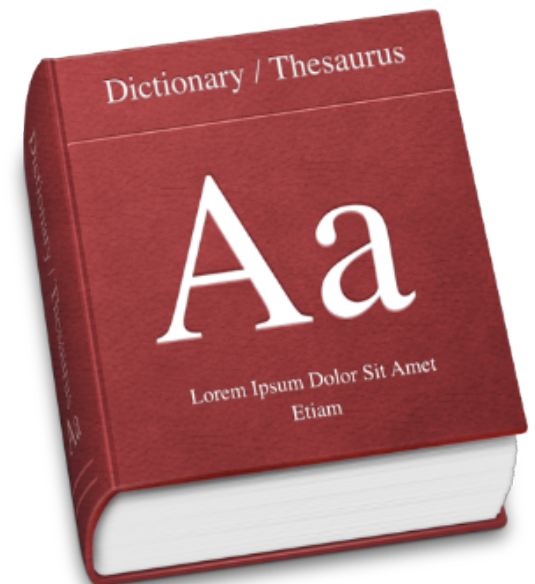
- **add(obj)**: añade un elemento en el conjunto, si aún no existe.
- **remove(obj)**: elimina un elemento del conjunto, si existe
- **boolean contains(obj)**: comprueba si un objeto existe en el conjunto
- **int size()**: devuelvo el número de elemento que hay en el conjunto
- **boolean isEmpty()**: comprueba si el conjunto esta vacío
- **list enumerate()**: devuelve una lista con todos los elemento en algún orden arbitrario

Implementation de un conjunto

- Podemos utilizar un array
 - **add**: añadir un elemento en el array $O(1)$
 - **contains**: recorrer el array $O(n)$
 - **remove**: encontrar y eliminar $O(n)$
- ¿Lo podemos hacer mejor?
 - ¿qué estructura tenemos en python?

Diccionario

- Un diccionario está diseñado para almacenar ítems compuestos de (*key*, *value*), donde la *key* es utilizada para encontrar los *values* del elemento correspondiente
- También conocido como *map*
- Aplicaciones:
 - Libreta de direcciones (name → address)
 - ...diccionario(word → definition)



Diccionario ADT

- **add(key, val)**: añade un ítem (key, value) al diccionario
- **V get(key)**: devuelve el value mapeado por la key correspondiente
- **remove(key)**: elimina la el ítem correspondiente a la key del diccionario
- **int size()**: devuelve el número de ítems que contiene el diccionario
- **boolean isEmpty()**: comprueba si el diccionario esta vacío.

Objetivo: Implementar un diccionario donde todos estos elementos tengan complejidad $O(1)$

Tablas Hash

- Una tabla hash es la **implementación de un diccionario**
- Las tablas de has están contruidos mediante arrays
- Tenemos una función $h(key)$ llamada “funcion hash” que obtiene una clave (key) y devuelve un índice dentro del array, donde los *values* de la key correspondiente están almacenados
- Es posible que distintas keys vayan a para al mismo índice.

¿cómo podemos guardar múltiples *values* de distintas key en un único indice?

Tipos de Tablas Hash

- **Hash Abierto**

- Consiste en tener en cada posición de la tabla, una lista de los elementos $\langle \text{key}, \text{values} \rangle$ (también llamado array de "buckets") que, de acuerdo a la función de hash, correspondan a dicha posición.
- En el peor caso, el hashing abierto nos conduce a una lista con todas las claves en una única lista.
 - El peor caso para búsqueda es así $O(n)$.

Tabla Hash Abierta (2)

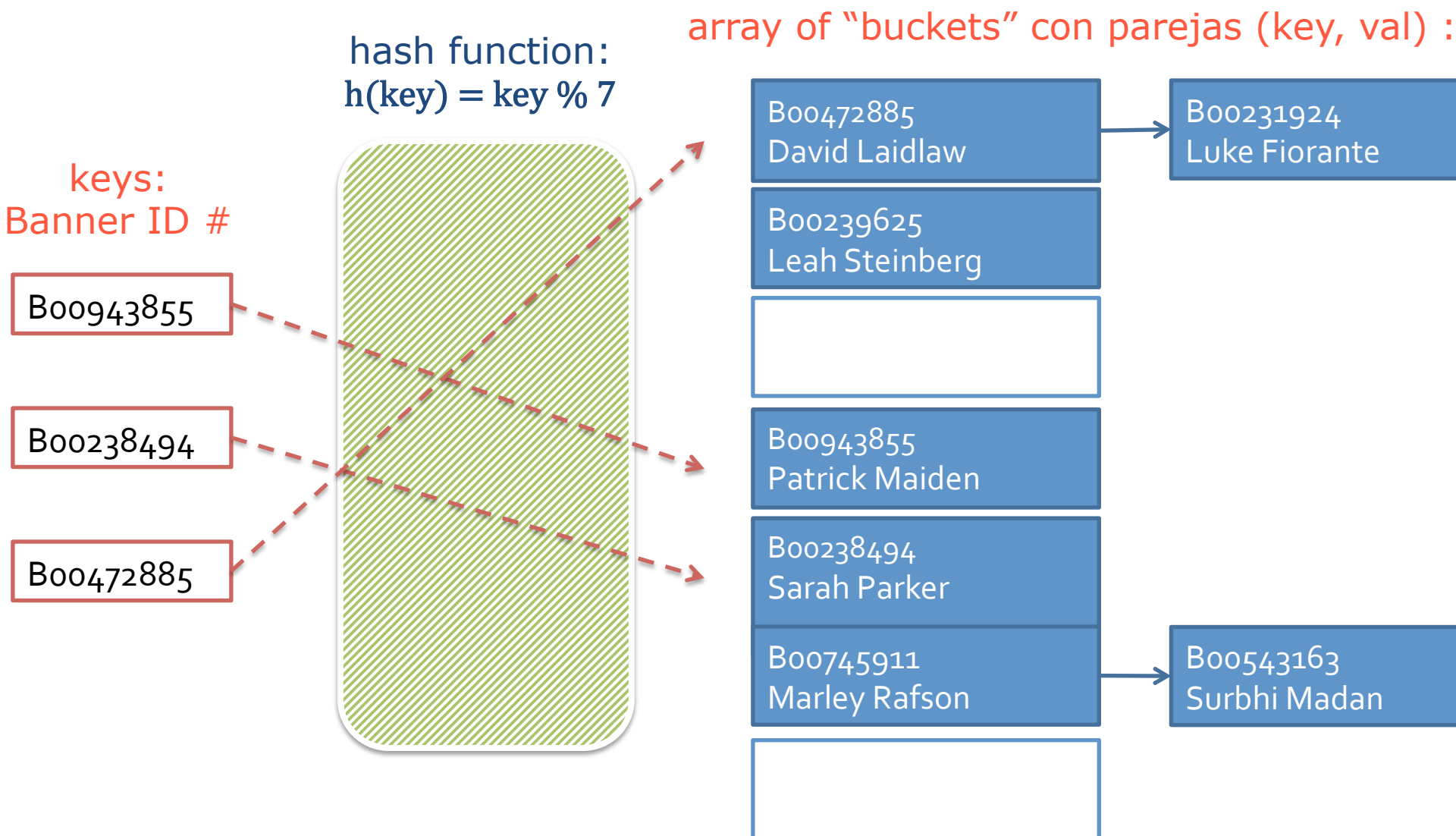


Tabla Hash Abierta (3)


```
table = array of some size  
h = some hash function
```

```
function add(key, val):  
    index = h(key)  
    table[index].append(key, val)
```

```
function get(key):  
    index = h(key)  
    for (k, v) in table[index]:  
        if k = key:  
            return v  
    error("key not found")
```

$O(1)$, mientras
que $h()$ sea
constante

Depende del
tamaño del
"bucket"!



Tipos de Tablas Hash

- **Hash Cerrado**

- El hash cerrado soluciona las colisiones buscando celdas alternativas hasta encontrar una vacía (dentro de la misma tabla). Se va buscando en las celdas: $d0(k)$, $d1(k)$, $d2(k)$, ..., donde:

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

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

- Cuando se busca una clave, se examinan varias celdas de la tabla hasta que se encuentra la clave buscada, o es claro que está no está almacenada.
- La inserción se efectúa probando la tabla hasta encontrar un espacio vacío.

Tabla Hash Cerrada (2)

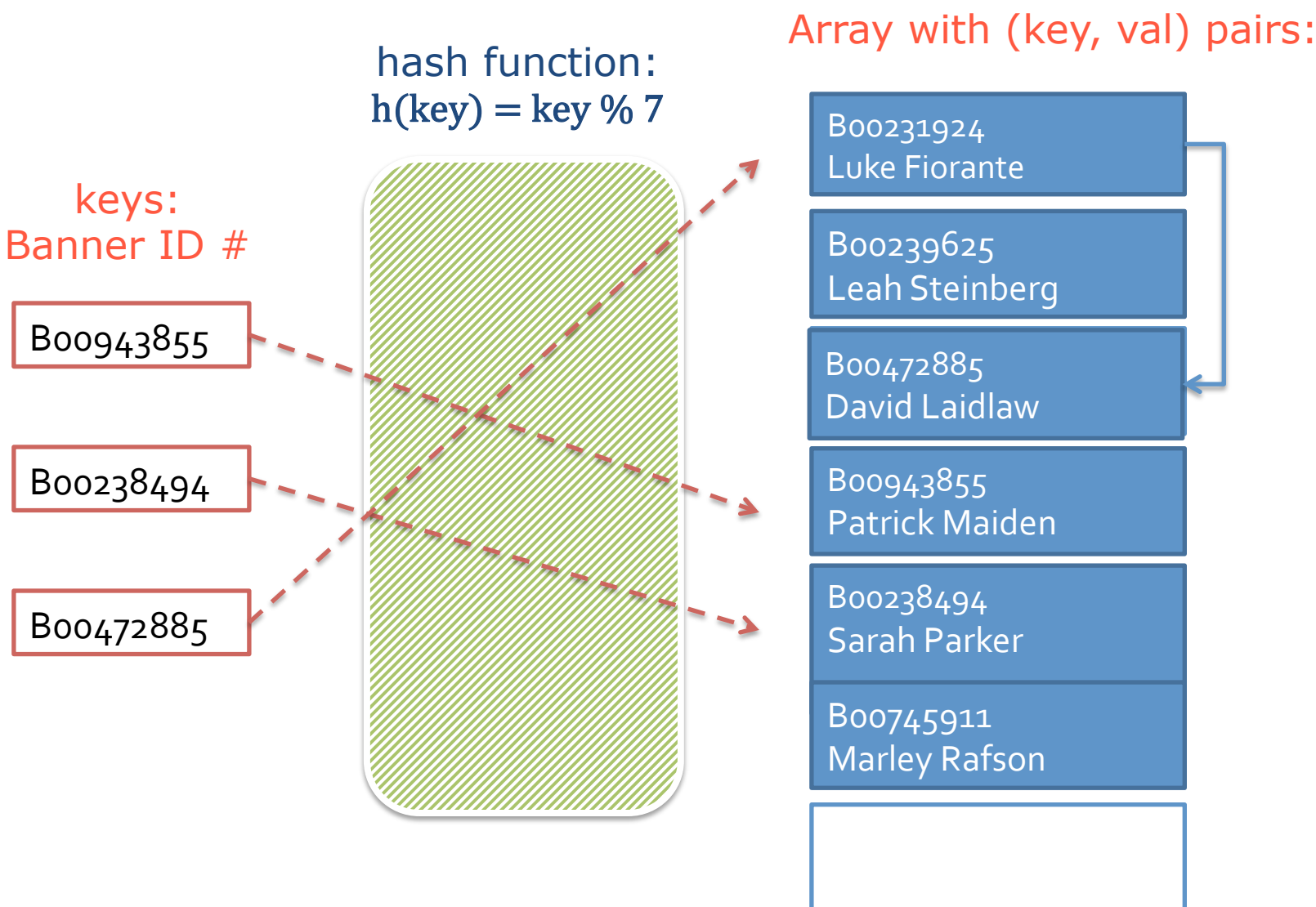


Tabla Hash Cerrada (3)

- **Ventaja:** Elimina totalmente los apuntadores. Se libera así espacio de memoria, el que puede ser usado en más entradas de la tabla.
- **Desventaja:** Si la aplicación realiza eliminaciones frecuentes, puede degradarse el rendimiento de la misma. Se requiere una tabla más grande. Para garantizar el funcionamiento correcto, se requiere que la tabla de hash tenga, por lo menos, el 50% del espacio disponible.

Tabla Hash Cerrada: add()

```
table = array of some size  
h = some hash function  
rehash = some rehash function given an index which is full  
  
function add(key, val):
```

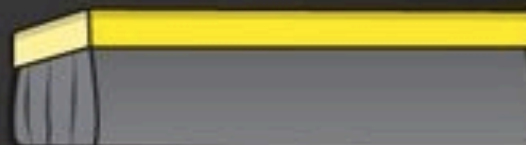


Tabla Hash Cerrada: add()

```
table = array of some size  
h = some hash function  
rehash = some rehash function given an index which is full
```

```
function add(key, val):  
    index = h(key)  
    if(table[index]==None):  
        table[index]=(key, val)  
    else:  
        index = rehash(index)  
        while(table[index]!=None):  
            index = rehash(index, key)  
        table[index]=(key, val)
```

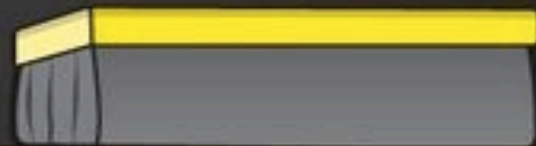
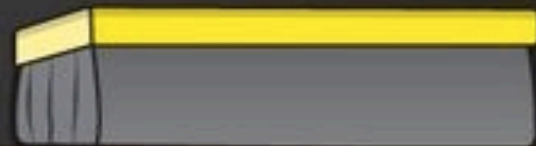


Tabla Hash Cerrada: get()

```
table = array of some size  
h = some hash function  
rehash = some rehash function given an index which is full  
  
function get(key):
```



Funciones Hash

- Es importante que la función $h(\text{key})$:
 - Tenga complejidad constante
 - Provoque el mínimo de colisiones posible

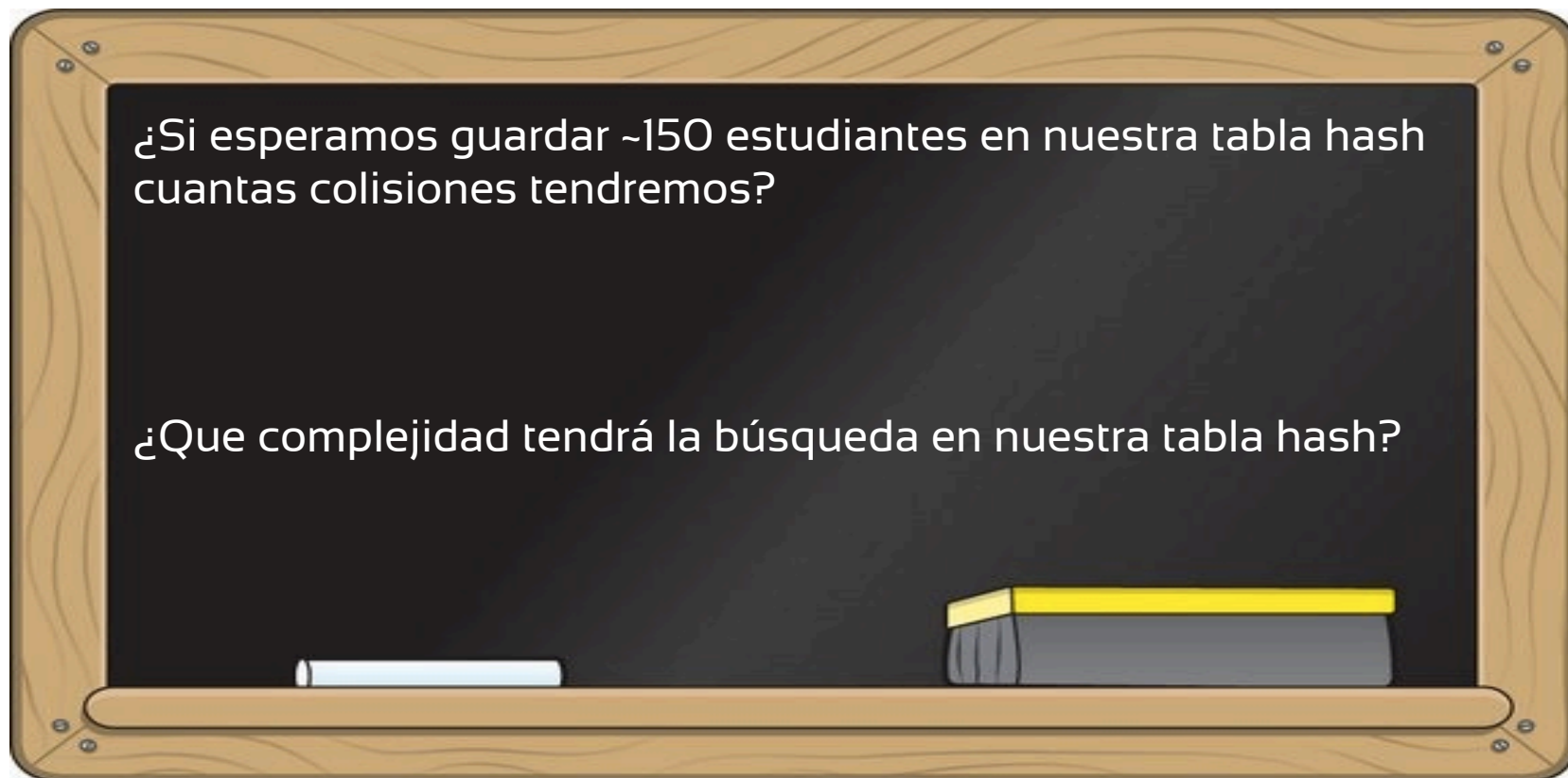
Funciones Hash

- En el ejemplo de la diapositiva #13, la tabla hash tenía tamaño 7, y la función hash utilizada era:

$$h(\text{key}) = \text{key} \% 7$$

¿Si esperamos guardar ~150 estudiantes en nuestra tabla hash cuantas colisiones tendremos?

¿Que complejidad tendrá la búsqueda en nuestra tabla hash?



Funciones Hash

- En el ejemplo de la diapositiva #13, la tabla hash tenía tamaño 7, y la función hash utilizada era:

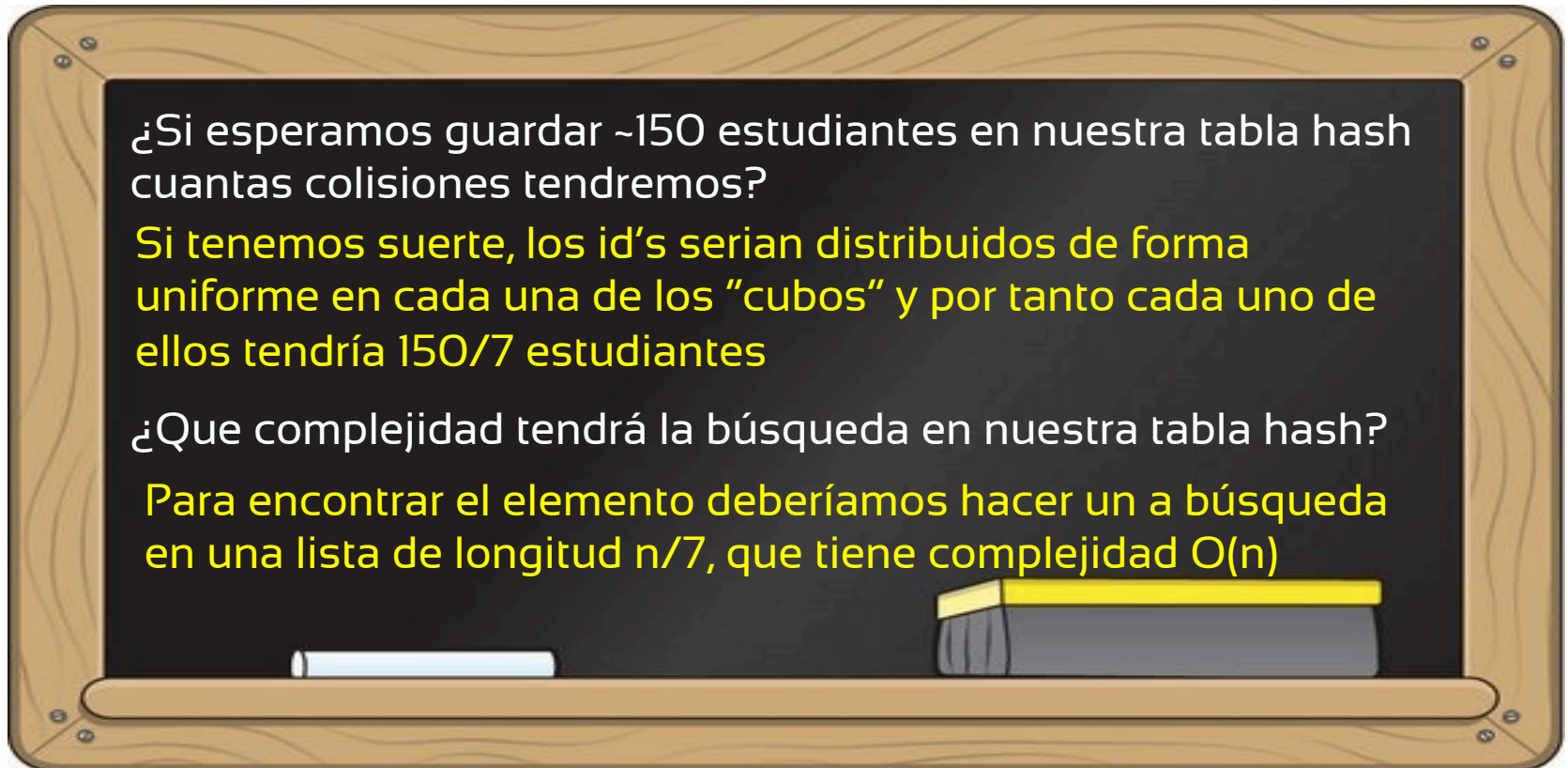
$$h(\text{key}) = \text{key} \% 7$$

¿Si esperamos guardar ~150 estudiantes en nuestra tabla hash cuantas colisiones tendremos?

Si tenemos suerte, los id's serían distribuidos de forma uniforme en cada una de los "cubos" y por tanto cada uno de ellos tendría $150/7$ estudiantes

¿Que complejidad tendrá la búsqueda en nuestra tabla hash?

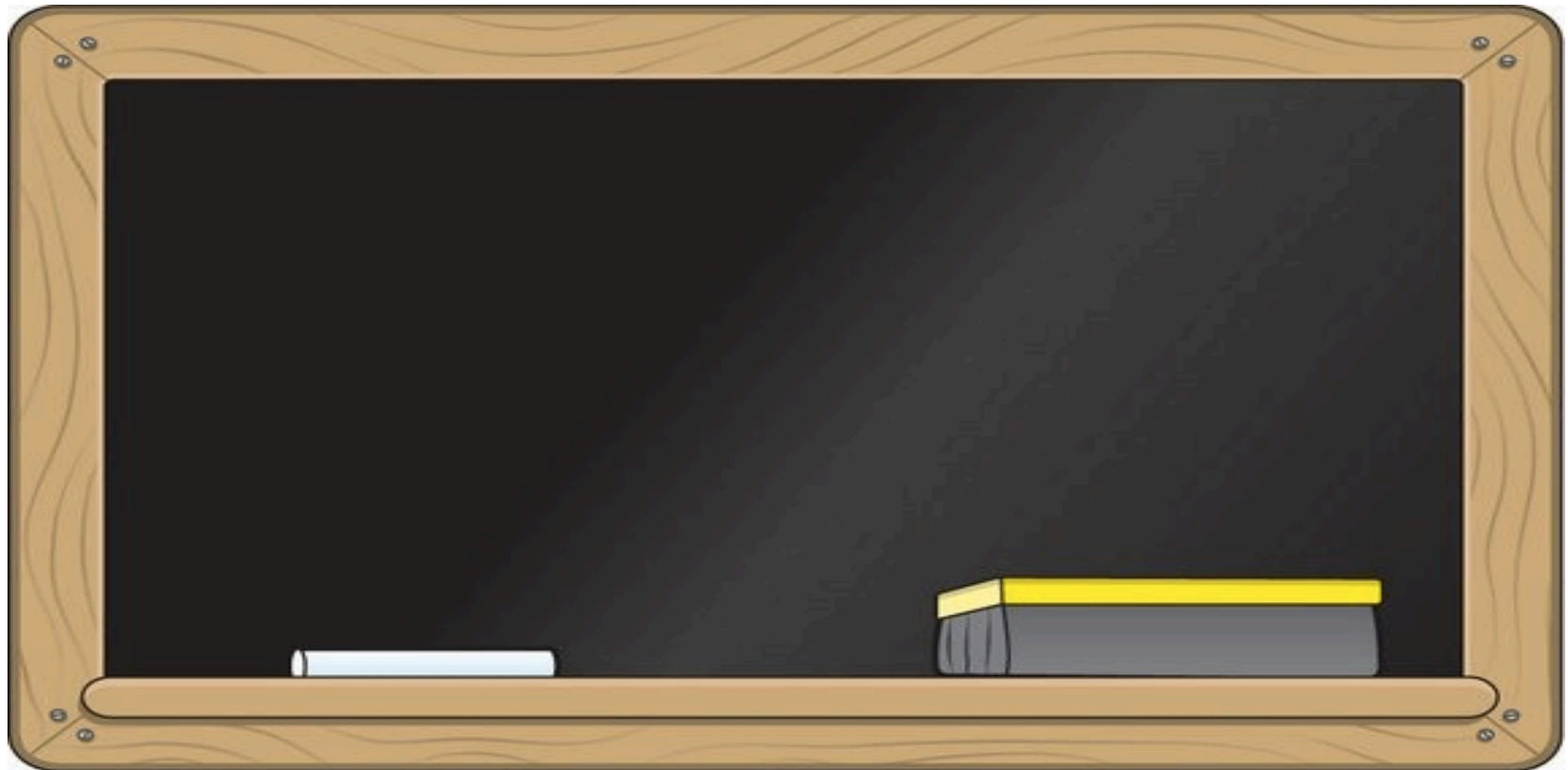
Para encontrar el elemento deberíamos hacer una búsqueda en una lista de longitud $n/7$, que tiene complejidad $O(n)$



Funciones Hash

Sabemos que los id's tiene 8 bits, por tanto, el Id mas grande será 99,999,999

¿Como lo podemos hacer mejor?



Funciones Hash

Sabemos que los id's tiene 8 bits, por tanto, el Id mas grande será 99,999,999

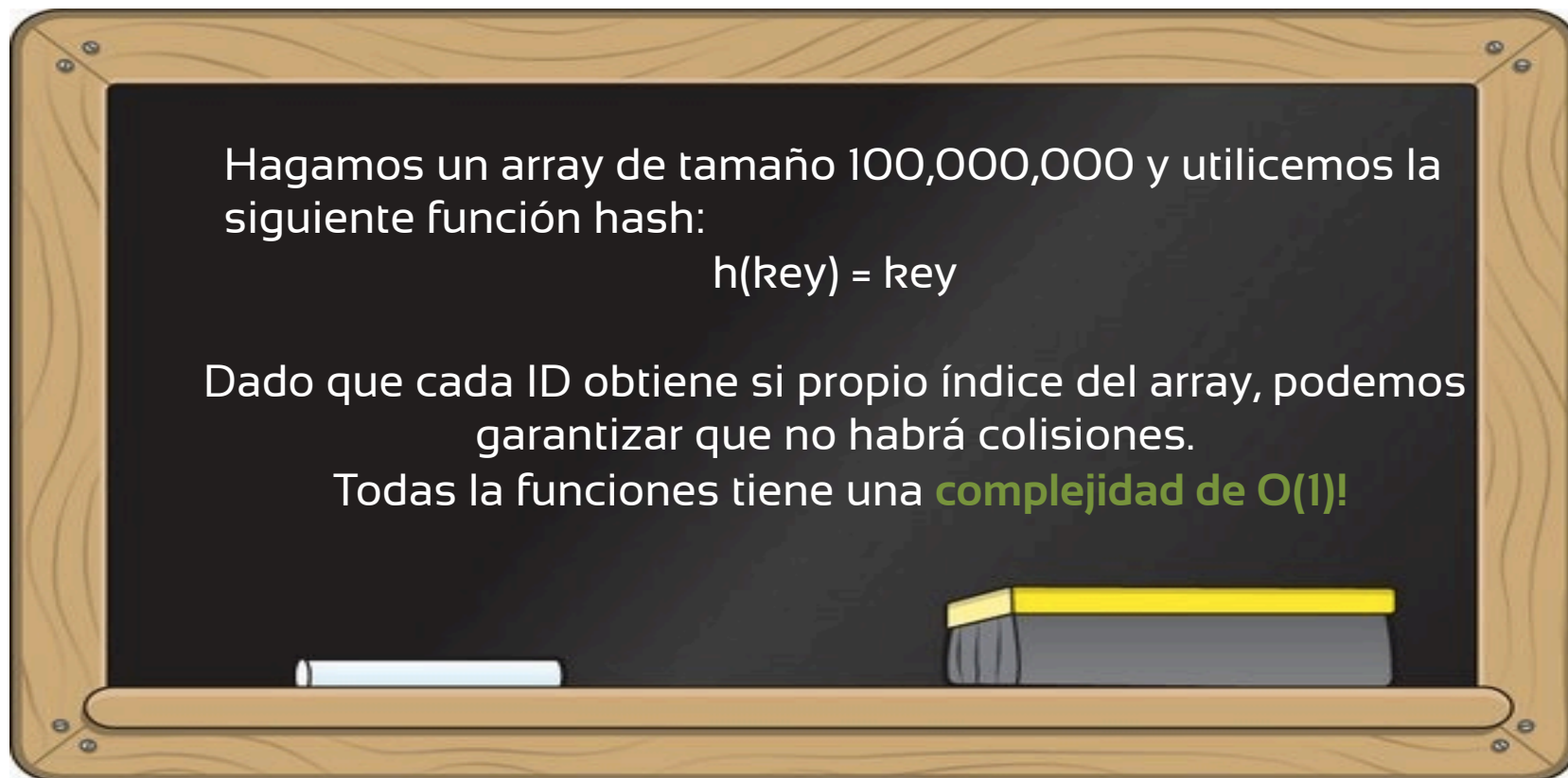
¿Como lo podemos hacer mejor?

Hagamos un array de tamaño 100,000,000 y utilicemos la siguiente función hash:

$$h(\text{key}) = \text{key}$$

Dado que cada ID obtiene si propio índice del array, podemos garantizar que no habrá colisiones.

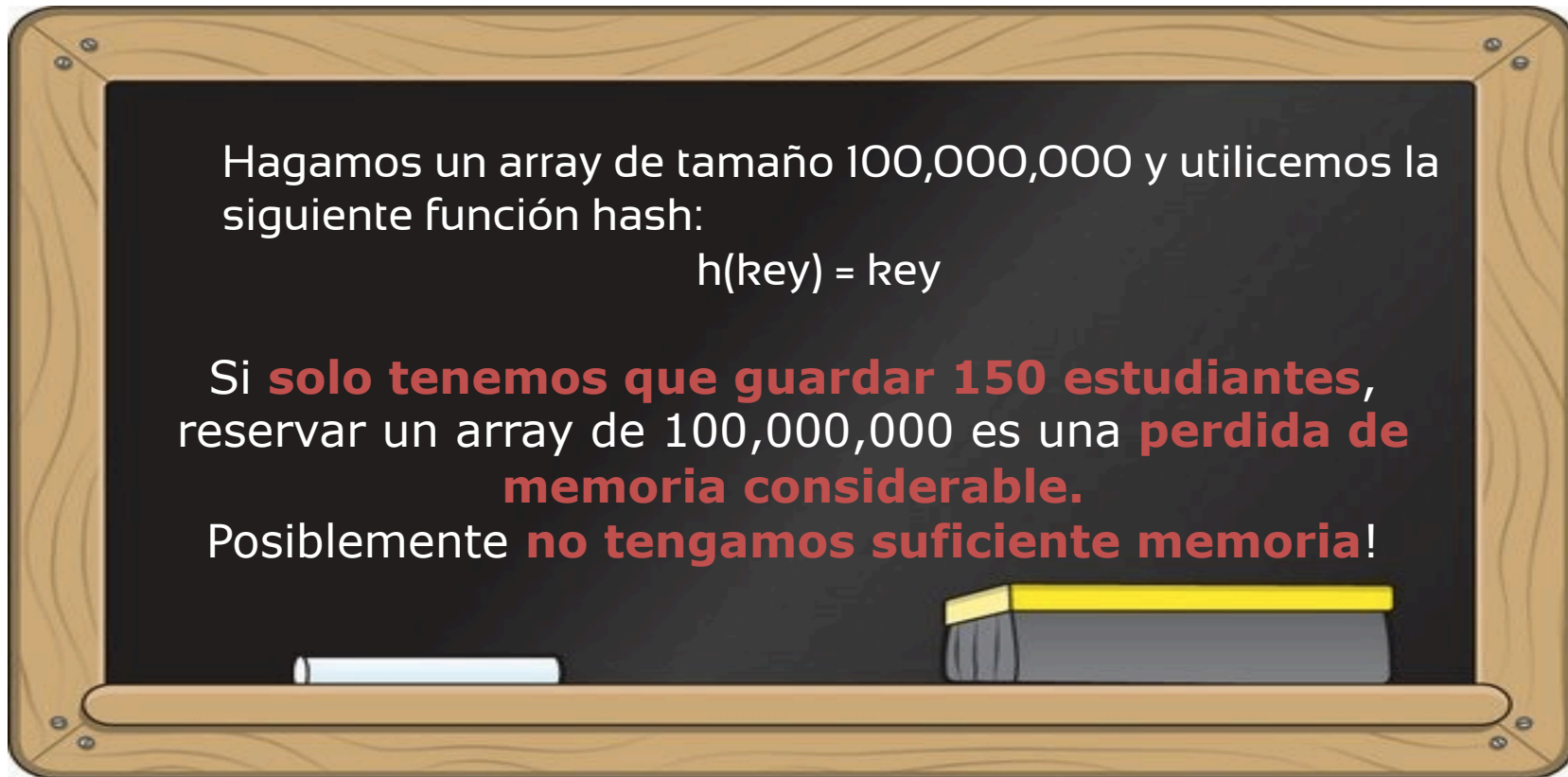
Todas la funciones tiene una **complejidad de $O(1)$** !



Funciones Hash

Sabemos que los id's tiene 8 bits, por tanto, el Id mas grande será 99,999,999

¿Como lo podemos hacer mejor?



Hagamos un array de tamaño 100,000,000 y utilicemos la siguiente función hash:

$$h(\text{key}) = \text{key}$$

Si **solo tenemos que guardar 150 estudiantes**, reservar un array de 100,000,000 es una **perdida de memoria considerable**.

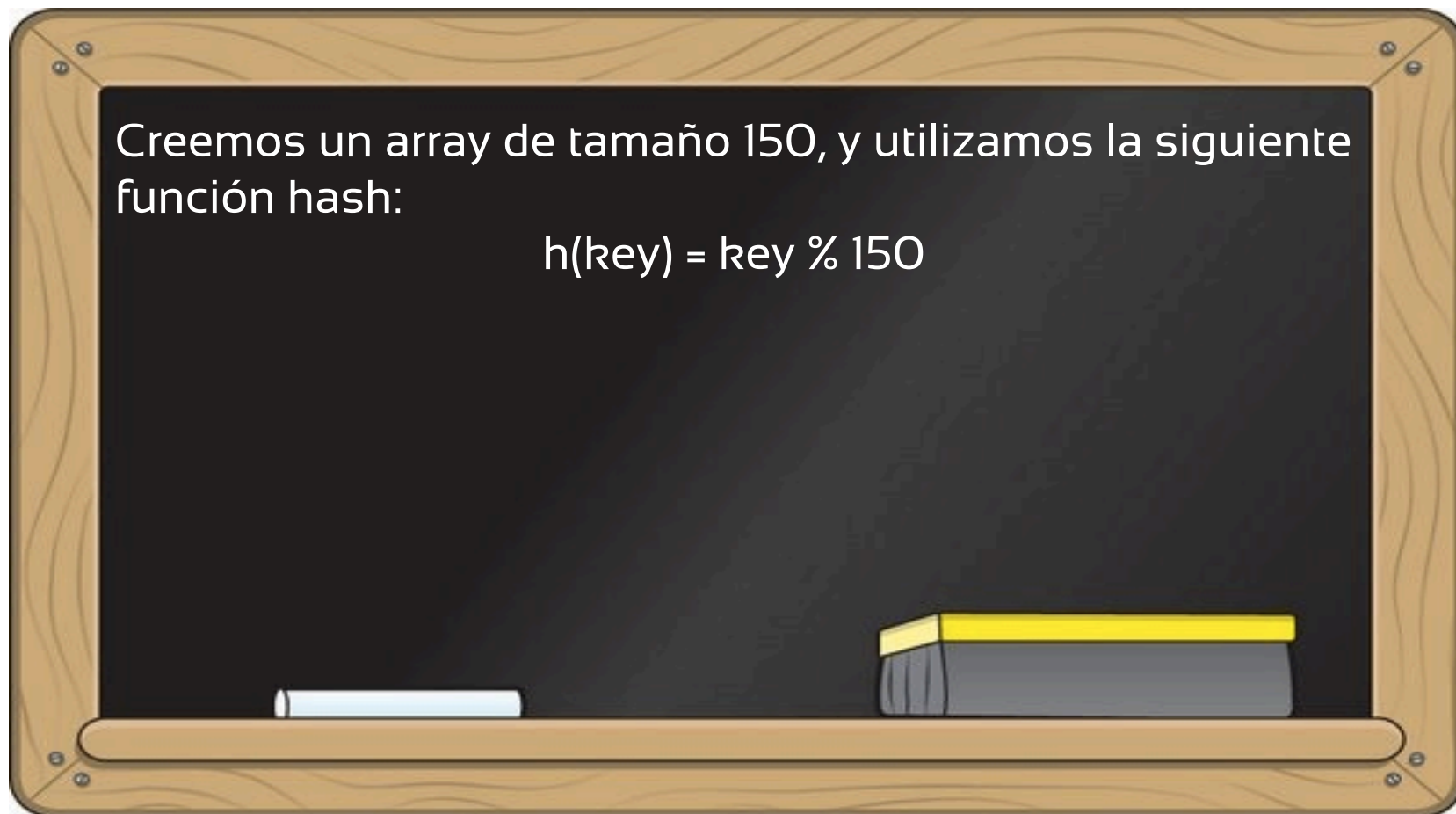
Posiblemente **no tengamos suficiente memoria!**

Funciones Hash

¿Alguna otra solución utilizando teniendo en cuenta que solo tenemos 150 ejemplos?

Creemos un array de tamaño 150, y utilizamos la siguiente función hash:

$$h(\text{key}) = \text{key} \% 150$$



Funciones Hash

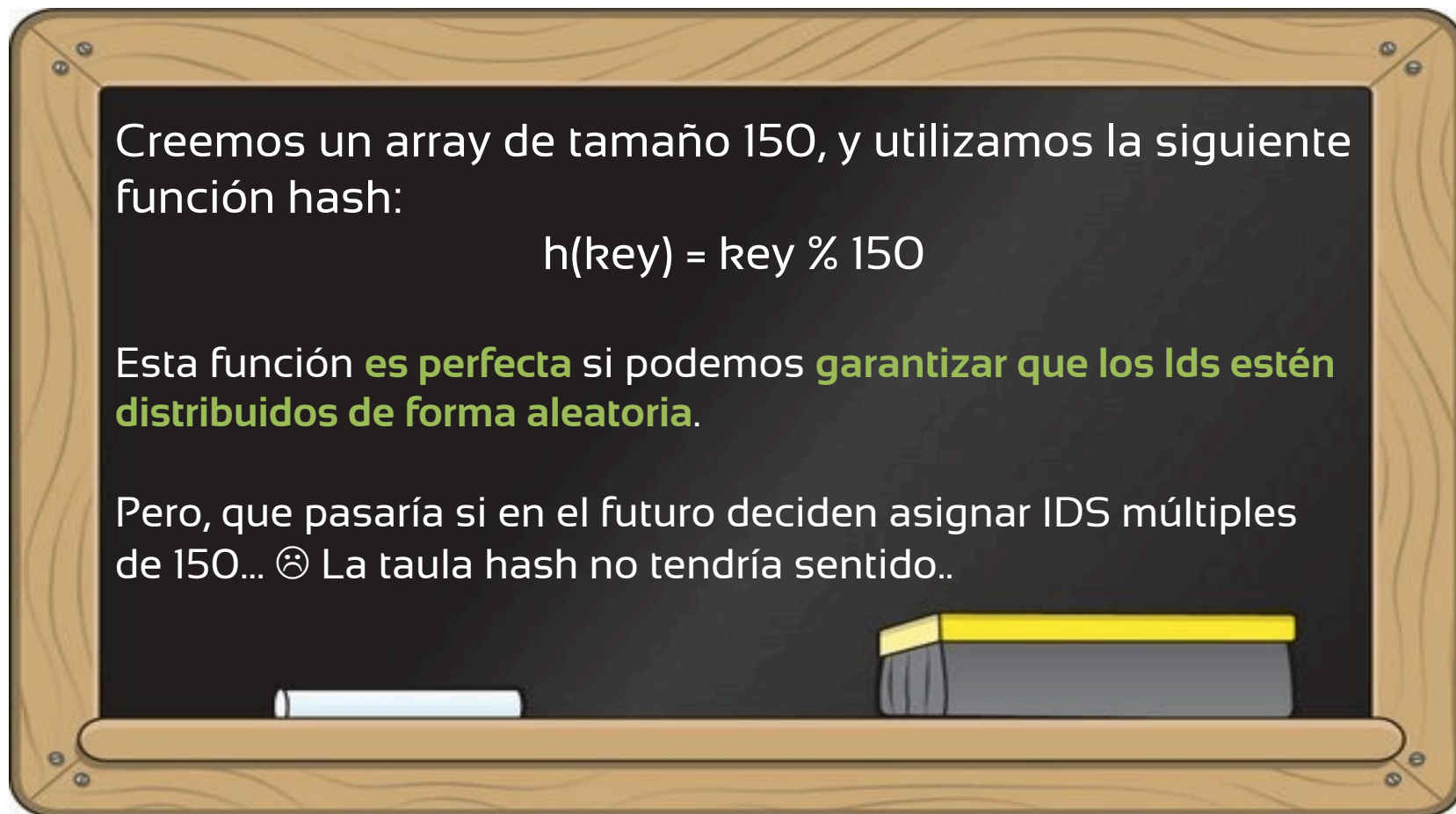
¿Alguna otra solución utilizando teniendo en cuenta que solo tenemos 150 ejemplos?

Creemos un array de tamaño 150, y utilizamos la siguiente función hash:

$$h(\text{key}) = \text{key} \% 150$$

Esta función **es perfecta** si podemos **garantizar que los Ids estén distribuidos de forma aleatoria**.

Pero, que pasaría si en el futuro deciden asignar IDS múltiples de 150... ☹ La taula hash no tendría sentido..



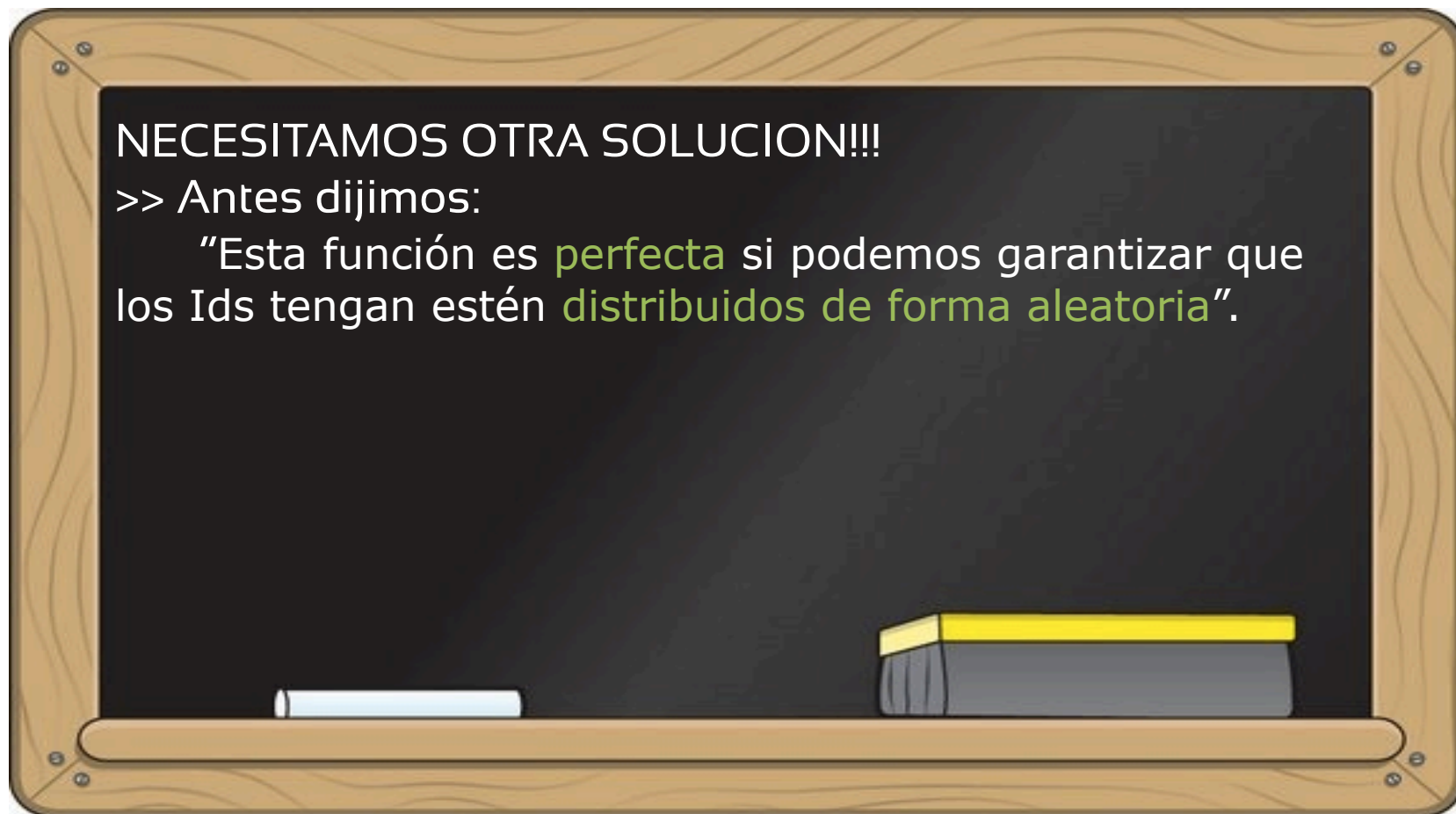
Funciones Hash

¿Alguna otra solución utilizando teniendo en cuenta que solo tenemos 150 ejemplos?

NECESITAMOS OTRA SOLUCION!!!

>> Antes dijimos:

“Esta función es **perfecta** si podemos garantizar que los Ids tengan estén **distribuidos de forma aleatoria**”.



Funciones Hash

¿Alguna otra solución utilizando teniendo en cuenta que solo tenemos 150 ejemplos?

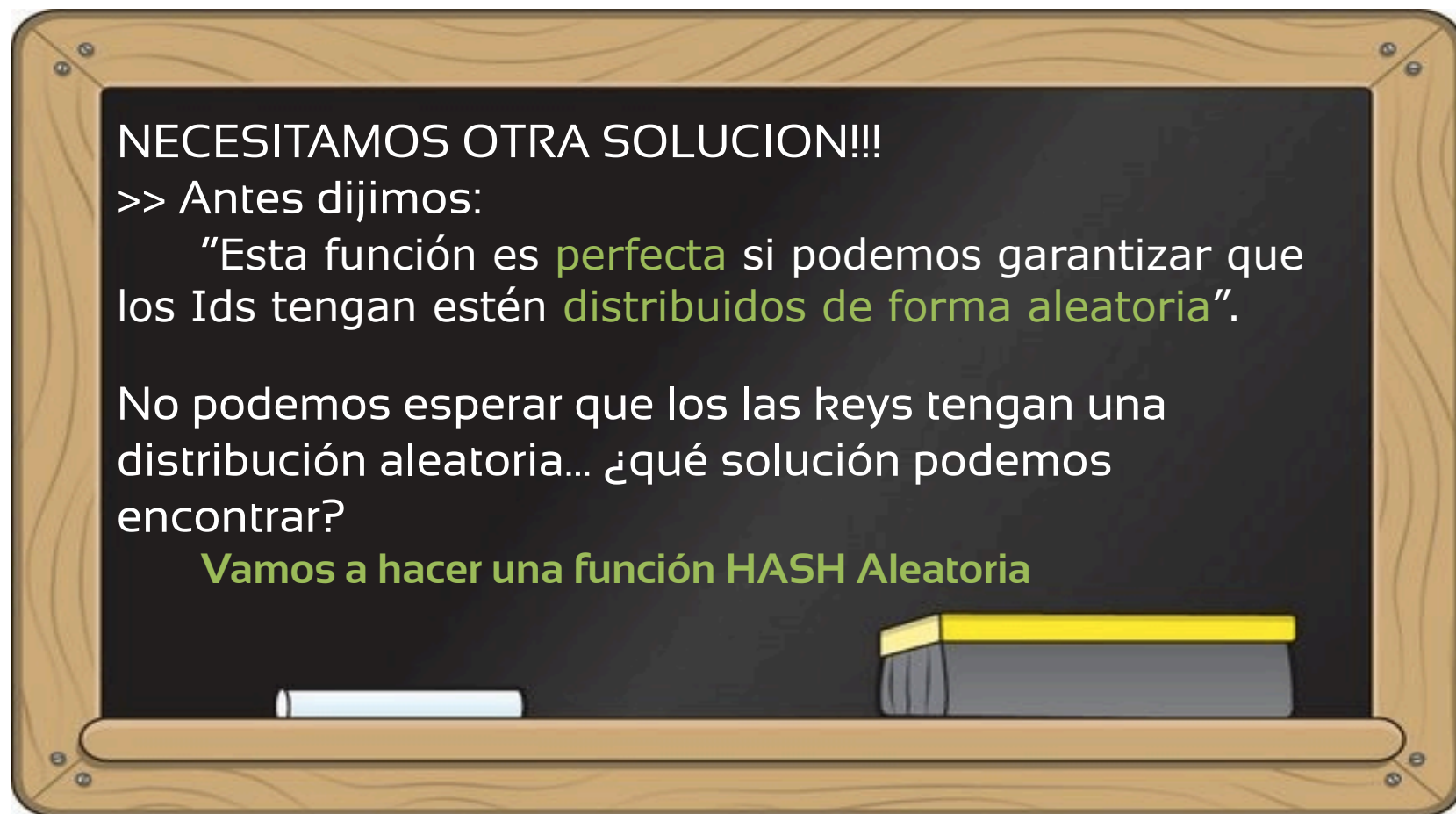
NECESITAMOS OTRA SOLUCION!!!

>> Antes dijimos:

“Esta función es **perfecta** si podemos garantizar que los Ids tengan estén **distribuidos de forma aleatoria**”.

No podemos esperar que los las keys tengan una distribución aleatoria... ¿qué solución podemos encontrar?

Vamos a hacer una función HASH Aleatoria



Funciones Hashing Universales

- Funciones Hash Universales:
 1. Escogemos un número primo mayor que la capacidad máxima esperada: 151
 1. Este va a ser el tamaño de nuestro array
 2. Escogemos 4 valores aleatorios entre 0 i 151
$$a_1, a_2, a_3, a_4$$
 1. Estos valores serán constantes durante la vida de nuestra función hash
 3. Dividimos la clave (key) en cuatro partes:
$$x_1, x_2, x_3, x_4$$
 1. e.g. B00238918 \rightarrow 00, 23, 89, 18
 4. $h(\text{key}) = (a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4) \% 151$

Funciones Hashing Universales

- ¿Porque la función de Hashing Universal funciona?
 - No es trivial... pero **muy interesante**.
- Para las funciones universales, podemos demostrar que el tamaño de cada "bucket" es menor que 2 lo que significa una complejidad del método `get()` es constante!

Demostración Hashing Universal: Background

- Recordemos algunos términos: Fracciones e inversos de fracciones:
 - El inverso de $3/4$ es $4/3$, porque $(3/4)*(4/3) = 1$
 - Algunas veces lo veremos escrito como: $(3/4)^{-1} = 4/3$
- Normalmente, los enteros no tienen inversos, ya que no podemos multiplicar un entero i por otro entero y obtener el valor 1. (solo funciona para $i = 1....$)
- Cuando entramos en el **mundo del módulo**, los enteros **empiezan a tener inversos !!!**
- Cojamos los enteros mod 7:
 - El inverso de 2 es 4, porque $2*4 = 8 \cong 1 \text{ mod } 7$
 - El inverso de 5 es 3, porque $5*3 = 15 \cong 1 \text{ mod } 7$

Demostración Hashing Universal: Background

- Pero, ¿todos los enteros tienen siempre un inverso bajo cualquier módulo?
 - ¿Que pasa con los enteros en mod 4? ¿Tiene el 2 un inverso?
 - $2*0 = 0 \cong 0 \pmod{4}$
 - $2*1 = 2 \cong 2 \pmod{4}$
 - $2*2 = 4 \cong 0 \pmod{4}$
 - $2*3 = 6 \cong 2 \pmod{4}$

No hay inverso!!
- Un entero $i \pmod{n}$ solo tiene **inverso si i y n son primos entre ellos**, lo que significa que únicamente el entero positivo que divide ambos i y n es el número 1.
- Por tanto, hay un inverso, y el inverso y el inverso es único
 - Teoría de Algebra!!
- Por tanto, si estamos hablando de enteros mod n , donde n es un número primo, entonces todo entero tiene inverso, ya que todos ellos son primos entre n .
 - A excepción del 0. $0 \times Y = 0$

Demostración Hashing Universal

Para la demostración:

- Llamemos n al número primero igual (o mayor) al tamaño del array
- Escojamos 2 ID's cualquiera, i dividámoslas en 4 partes:

$$(x_1, x_2, x_3, x_4) \text{ y } (y_1, y_2, y_3, y_4)$$

- Ya que los IDs son distintos entre ellos, sabemos que al menos una de sus partes es diferente. Sin perder generalización, podemos asumir que las partes distintas son las últimas:

$$x_4 \neq y_4$$

- Fijamos 4 números aleatorios para nuestra función hash, h :

$$a_1, a_2, a_3, a_4$$

- La probabilidad que las 2 IDs vayan a parar a misma celda hash es la probabilidad de que :

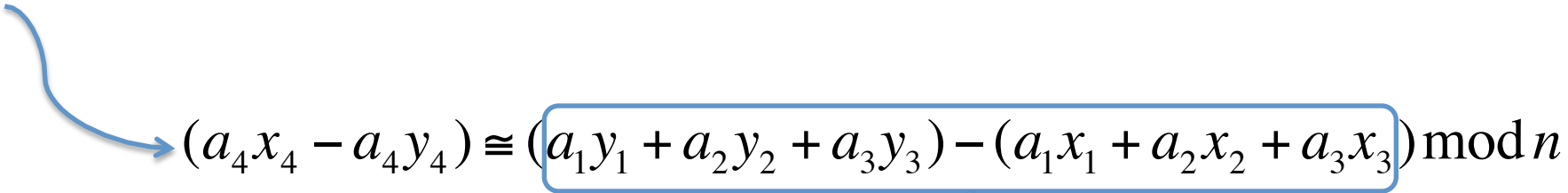
$$h(x_1, x_2, x_3, x_4) = h(y_1, y_2, y_3, y_4)$$

Demostración Hashing Universal (2)

Simplifiquemos la expresion:

$$h(x_1, x_2, x_3, x_4) = h(y_1, y_2, y_3, y_4)$$

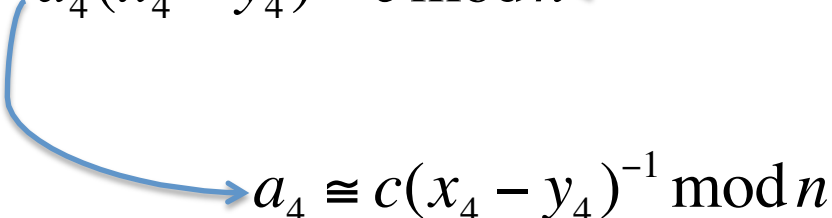
$$(a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4) \equiv (a_1y_1 + a_2y_2 + a_3y_3 + a_4y_4) \pmod n$$


$$(a_4x_4 - a_4y_4) \equiv (a_1y_1 + a_2y_2 + a_3y_3) - (a_1x_1 + a_2x_2 + a_3x_3) \pmod n$$

Es solo un número, c

$$a_4(x_4 - y_4) \equiv c \pmod n$$

Multiplicamos
ambos lados por
 $(x_4 - y_4)^{-1}$


$$a_4 \equiv c(x_4 - y_4)^{-1} \pmod n$$

Demstración Hashing Universal (3)

- ...por tanto, la probabilidad que 2 IDs distintos colisionen es la probabilidad que:

$$a_4 \cong c(x_4 - y_4)^{-1} \text{ mod } n$$

- Ya que $x_4 \neq y_4$, sabemos que:

$$(x_4 - y_4) \neq 0$$

- Como n es un número primo, podemos garantizar que $(x_4 - y_4)$ tiene un único inverso $\text{mod } n$.
- Por tanto, hay un único valor posible que para $c(x_4 - y_4)^{-1}$, y un único valor de a_4 que satisfaga esta congruencia.
- Dado que a_4 ha sido escogido aleatoriamente de un conjunto de n valores posibles, la probabilidad que a_4 haya sido seleccionado "correctamente" es $1/n$.
- Por tanto, la probabilidad, x , que un ID cualquiera colisione con otro ID es $1/n = 1/151$. Esto significa, que el número esperado de colisiones mediante x y los otros IDs es igual a $149/151 \approx 1$.
- Por tanto el tamaño esperado del "bucket" ≈ 2

Volvamos a los conjuntos

- ¿Podemos usar hashing para implementar conjuntos?
 - No tenemos parejas <key,value>, solo elementos.

```
function add(obj):  
    index = h(obj)  
    table[index].append(obj)
```

```
function contains(obj):  
    index = h(obj)  
    for elt in table[index]:  
        if elt == obj:  
            return true  
    return false
```

- Es llamado HashSet

Keys alfanúmericas?

- Si se trata de **secuencia de caracteres o strings**, se puede interpretar cada carácter como un número en base 128 (los números ASCII van del 0 al 127) y el string completo como un número en base 128.
 - Por ejemplo la clave "pt" puede ser transformada a $(112 * 128 + 116) = 14452$. (ASCII(p)=112 y ASCII(t)=116)

HashMap vs HashSet

Hash Map	Hash Set
<ul style="list-style-type: none">• Mapea claves (keys) a valores• No tienen orden	<ul style="list-style-type: none">• No tenemos claves(keys), solo valores.<ul style="list-style-type: none">• Es como el HashMap, donde la clave (key) es el mismo valor.• No tienen orden