

# Tablas de hash

Clase 11

IIC 2133 - Sección 1

Prof. Sebastián Buggedo

# Sumario

Obertura

Tablas de hash

Colisiones

Funciones de hash

Epílogo

# Diccionarios

## Definición

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

- **Asociar** un valor a una llave
- **Actualizar** el valor asociado a una llave
- **Obtener** el valor asociado a una llave
- En ciertos casos, **eliminar** de la estructura una asociación llave-valor

Los ABB fueron nuestra primera EDD para implementar diccionarios

# Diccionarios

- Los ABB efectivamente soportan las operaciones de diccionario
- La complejidad de las operaciones es  $\mathcal{O}(h)$
- Cuando están balanceados,  $h \in \mathcal{O}(\log(n))$  para  $n$  llaves almacenadas

## Ejemplo

Podemos mantener pares (llave, valor) de la forma (rut, archivo)

- Podemos saber si un rut está en el dic. haciendo búsqueda por rut
- Inserción usa rut para ubicar el nodo, balanceando si es necesario

Hay algo más que los ABB poseen y no es un requisito de los diccionarios

# Diccionarios

- Los ABB no solo soportan las operaciones de diccionario
- La propiedad de **árbol de búsqueda** garantiza que los datos están **ordenados**
- Si el orden es importante, esto es necesario

¿Qué es lo más importante en un diccionario?

# Diccionarios

- El principal objetivo de los diccionarios es **búsqueda eficiente** de llaves
- El objetivo secundario es **inserción/modificación eficiente** de pares llave-valor
- Por esto, el orden de las llaves deja de ser relevante

¿Podemos buscar e insertar más rápido  
si nos olvidamos de mantener el orden?

# Diccionarios

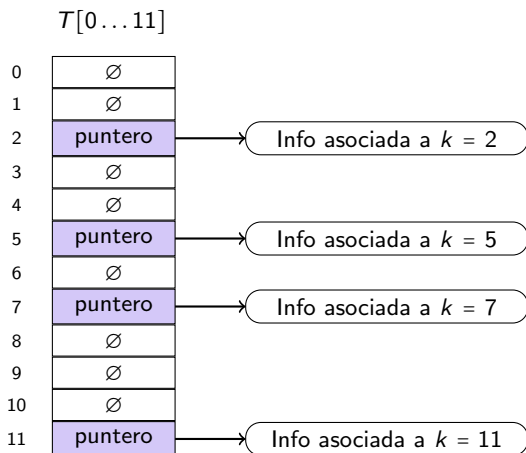
Para motivar nuestra siguiente estructura, consideremos un escenario ideal

- Conjunto de llaves posibles  $K = \{0, \dots, 11\}$  fijo y conocido
- Dada una llave  $k \in K$ , interesa saber si esta se encuentra asociada a un valor en la EDD

Este escenario se puede manejar con la siguiente EDD básica

- Arreglo  $T[0 \dots 11]$  iniciado con  $\emptyset$  en cada celda
- No almacenamos las llaves en las celdas, sino un puntero al valor asociado a la llave  $k$  en  $T[k]$
- Es decir,  $T[k] = \emptyset \Leftrightarrow$  no hay valor asociado a  $k$  en  $T$

# Diccionarios



¿Cuál es la complejidad de la búsqueda y la inserción en  $T$ ?



# Diccionarios

En la estructura  $T$  los accesos a  $T[k]$  son **accesos por índice**

- Verificar si  $T[k] = \emptyset$  es  $\mathcal{O}(1)$
- Insertar/modificar valor en  $T[k]$  es  $\mathcal{O}(1)$

No solo las operaciones deseadas son super eficientes

- A diferencia de un ABB, se almacena solo un puntero al valor guardado
- No se usan punteros a padres-hijos para mantener la estructura

¿Qué tan ideal es este escenario de llaves naturales  $K$ ?

# Diccionarios

El escenario de llaves  $K$  puede ocurrir en aplicaciones prácticas

- Rango de valores razonable
- Llaves siempre naturales (para ser usadas como índices de arreglos)

## Ejemplo

En la universidad hay aproximadamente 25.000 estudiantes este año.

- Asignamos un  $k \in \{0, \dots, 24.999\}$  a cada estudiante
- Usamos cada natural como índice del arreglo  $T$

# Diccionarios

## Ejemplo

Cada estudiante ya posee un rut único

- Rango de rut's abarca hasta el 25.000.000
- Cantidad de estudiantes mucho menor (25.000)
- **Problema:** solo 1/1000 celdas del arreglo  $T$  indexado por ruts estarán ocupadas

No solo los ruts son llaves posibles

- Números de teléfono
- Patentes de vehículos
- ...

¿Cómo acercarnos a un conjunto de llaves  $K$  razonable?

# Objetivos de la clase

- ☐ Comprender el concepto de función de hash
- ☐ Identificar limitaciones en el almacenamiento a través de arreglos indexados
- ☐ Comprender concepto de tabla de hash
- ☐ Comprender concepto de colisión y sus posibles manejos
- ☐ Distinguir diferencias entre encadenamiento y direccionamiento abierto
- ☐ Identificar buenas propiedades de una función de hash
- ☐ Comprender algunas familias de funciones de hash comunes

# PRIMER DATICONCURSO!



Link al concurso: [DatiConcurso #1](#)

Plazo de recepción de creaciones: 28 de septiembre 23:59

# Sumario

Obertura

**Tablas de hash**

Colisiones

Funciones de hash

Epílogo

# Funciones de hash

## Definición

Dado un espacio de llaves  $K$  y un natural  $m > 0$ , una **función de hash** se define como

$$h : K \rightarrow \{0, \dots, m-1\}$$

Dado  $k \in K$ , llamaremos **valor de hash de  $k$**  a la evaluación  $h(k)$ .

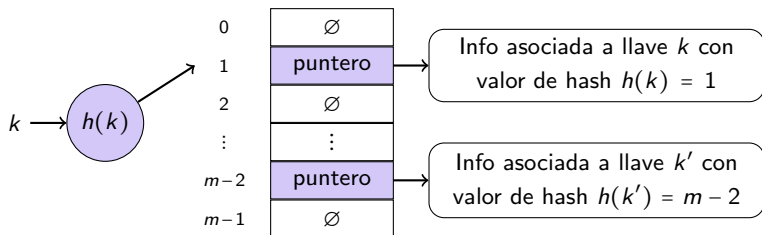
Notemos que

- Una función de hash nos permite mapear un espacio de llaves a otro más pequeño (con  $m$  razonable)
- Una función de hash no necesariamente es **inyectiva**
- Si  $m < |K|$ , no puede ser inyectiva
- En la práctica,  $m \ll |K|$

# Tablas de hash

## Definición

Dado  $m > 0$  y un conjunto de llaves  $K$ , una **tabla de hash**  $T$  es una EDD que asocia valores a llaves indexadas usando una función de hash  $h: K \rightarrow \{0, \dots, m-1\}$ . Diremos que tal  $T$  es de tamaño  $m$ .





# Tabla de hash

- El ejemplo *ideal* que estudiamos es una tabla de hash
- La función de hash es  $h : K \rightarrow K$  dada por

$$h(k) = k$$

- Las operaciones de diccionario son sencillas

`IdentityHashSearch (T, k):`

**return** `T[k]`

`IdentityHashInsert (T, k, v):`

`T[k] = v`

`IdentityHashDelete (T, k):`

`T[k] = ∅`

0	∅
1	∅
2	puntero
3	∅
4	∅
5	puntero
6	∅
7	puntero
8	∅
9	∅
10	∅
11	puntero

# Tabla de hash

- Usar la misma estrategia para hashing general sería

HashSearch ( $T, k$ ):

**return**  $T[h(k)]$

HashInsert ( $T, k, v$ ):

$T[h(k)] = v$

HashDelete ( $T, k$ ):

$T[h(k)] = \emptyset$

- Pero sabemos que  $h$  no necesariamente es inyectiva
- Es decir, puede ocurrir una **colisión**  $h(k_1) = h(k_2)$  para  $k_1 \neq k_2$

Veremos formas de manejar las colisiones

# Sumario

Obertura

Tablas de hash

**Colisiones**

Funciones de hash

Epílogo

# Una función de hash típica

Para ejemplificar el problema de las colisiones, consideremos la siguiente función de hash

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

Se le conoce como **hashing modular** y corresponde al resto al dividir  $k$  entre  $m$

- Notemos que  $h(k) \in \{0, \dots, m-1\}$  para todo  $k$
- Todas las llaves con el mismo resto al dividir entre  $m$  generan una colisión, i.e.

$$h(k_1) = h(k_2) \Leftrightarrow k_1 \equiv_m k_2$$

## Ejemplo

Tomando  $m = 100$  y  $K = \{0, \dots, 999\}$ , el hashing modular cumple

- $h(12) = h(112) = \dots = h(912) = 12$
- $h(18) = \dots = h(918) = 18$

# Inserción

Usaremos la función de hashing modular para experimentar con **inserciones**

Consideremos  $m = 7$ . Insertemos la llave 15 en la siguiente tabla de hash

- Su valor de hash es  $h(15) = 15 \bmod 7 = 1$

0	Ø
1	Ø
2	Ø
3	Ø
4	Ø
5	Ø
6	Ø

# Inserción

La posición  $h(15) = 1$  está libre y guardamos la llave

0	∅
1	∅
2	∅
3	∅
4	∅
5	∅
6	∅

0	∅
1	∅
2	∅
3	∅
4	∅
5	∅
6	∅

0	∅
1	15
2	∅
3	∅
4	∅
5	∅
6	∅

# Inserción

Ahora insertamos la llave 37

- Su valor de hash es  $h(37) = 15 \bmod 7 = 2$

0	∅
1	15
2	∅
3	∅
4	∅
5	∅
6	∅

0	∅
1	15
2	∅
3	∅
4	∅
5	∅
6	∅

0	∅
1	15
2	37
3	∅
4	∅
5	∅
6	∅

# Inserción

Ahora insertamos la llave 51

- Su valor de hash es  $h(51) = 15 \bmod 7 = 2$

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

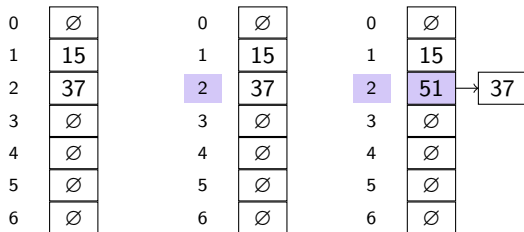
¿Qué hacemos con la colisión?



# Inserción con encadenamiento

Primera propuesta: **encadenamiento**

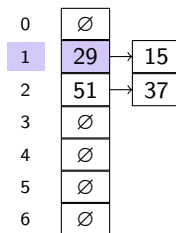
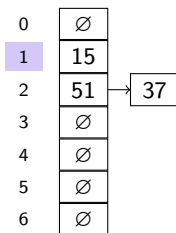
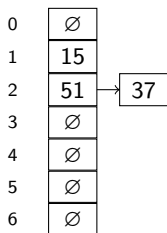
- Cada valor guardado es un nodo de una lista ligada
- Cada colisión agrega un nodo al principio de la lista



# Inserción con encadenamiento

Al insertar la llave 29 seguimos la misma idea

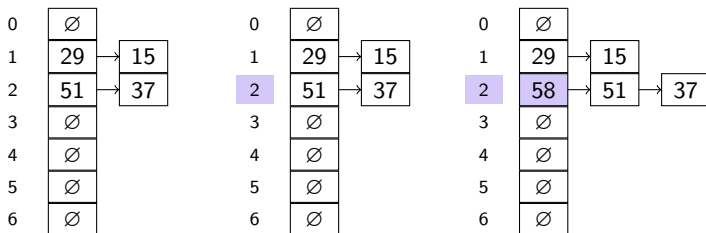
- Su valor de hash es  $h(29) = 29 \bmod 7 = 1$



# Inserción con encadenamiento

Al insertar la llave 58 seguimos la misma idea

- Su valor de hash es  $h(58) = 58 \bmod 7 = 2$



# Encadenamiento

- Las operaciones de diccionario involucran la lista ligada  $T[h(k)]$

**ChainedHashSearch** ( $T, k$ ):

    Buscar llave  $k$  en  $T[h(k)]$

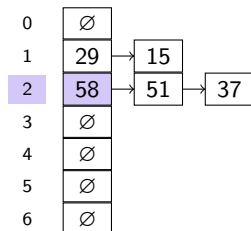
**ChainedHashInsert** ( $T, k, v$ ):

    Insertar  $(k, v)$  como cabeza de  $T[h(k)]$

**ChainedHashDelete** ( $T, k$ ):

    Eliminar llave  $k$  de  $T[h(k)]$

- La complejidad de estas operaciones depende de qué tan largas sean las listas
- Una *buena* función de hash repartiría las llaves de manera más o menos homogénea



# Otra estrategia para colisiones

Volvamos a la inserción con colisión del 51

- Su valor de hash es  $h(51) = 15 \bmod 7 = 2$

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

¿Alguna alternativa al encadenamiento?

# Inserción con sondeo lineal

Segunda propuesta: **direccionamiento abierto**

- Buscamos sistemáticamente una celda vacía
- Puede producir nuevas colisiones no previstas por  $h$

Una forma de buscar: el **sondeo lineal** inserta en la primera celda vacía a la derecha de la colisión

0	∅
1	15
2	37
3	∅
4	∅
5	∅
6	∅

0	∅
1	15
2	37
3	∅
4	∅
5	∅
6	∅

0	∅
1	15
2	37
3	51
4	∅
5	∅
6	∅

# Insertión con sondeo lineal

Al insertar la llave 29 seguimos la misma idea del sondeo lineal

- Su valor de hash es  $h(29) = 29 \bmod 7 = 1$

0	∅
1	15
2	37
3	51
4	∅
5	∅
6	∅

0	∅
1	15
2	37
3	51
4	∅
5	∅
6	∅

0	∅
1	15
2	37
3	51
4	∅
5	∅
6	∅

0	∅
1	15
2	37
3	51
4	29
5	∅
6	∅

# Búsqueda con sondeo lineal

Si las inserciones son con sondeo lineal, la búsqueda debe tenerlo en cuenta

- No necesariamente  $k$  está guardado en  $T[h(k)]$
- Debemos revisar esa celda, y si no corresponde, buscar **hacia la derecha**

Por ejemplo, al buscar la llave 29 comenzamos la búsqueda en la pos. 1

0	∅
1	15
2	37
3	51
4	29
5	∅
6	∅

0	∅
1	15
2	37
3	51
4	29
5	∅
6	∅

0	∅
1	15
2	37
3	51
4	29
5	∅
6	∅

0	∅
1	15
2	37
3	51
4	29
5	∅
6	∅

La búsqueda sigue la misma secuencia que la inserción



# Búsqueda con sondeo lineal

¿Cómo detectamos si la llave no está?

- Comenzamos la búsqueda en  $T[h(k)]$
- Si al buscar a la derecha llegamos a un  $\emptyset$  significa que no está

Por ejemplo, al buscar la llave 10, tal que  $h(10) = 10 \bmod 7 = 3$

0	$\emptyset$
1	15
2	37
3	51
4	29
5	$\emptyset$
6	$\emptyset$

0	$\emptyset$
1	15
2	37
3	51
4	29
5	$\emptyset$
6	$\emptyset$

0	$\emptyset$
1	15
2	37
3	51
4	29
5	$\emptyset$
6	$\emptyset$

Concluimos que 10 no está almacenada

# Eliminación con sondeo lineal

Para eliminar llaves guardadas tenemos un problema

- Si borramos una llave, la reemplazamos por  $\emptyset$

Por ejemplo, si borramos el 51 y buscamos el 29 con  $h(29) = 29 \bmod 7 = 1$

0	$\emptyset$
1	15
2	37
3	51
4	29
5	$\emptyset$
6	$\emptyset$

0	$\emptyset$
1	15
2	37
3	$\emptyset$
4	29
5	$\emptyset$
6	$\emptyset$

0	$\emptyset$
1	15
2	37
3	$\emptyset$
4	29
5	$\emptyset$
6	$\emptyset$

0	$\emptyset$
1	15
2	37
3	$\emptyset$
4	29
5	$\emptyset$
6	$\emptyset$

Concluimos que 29 no está almacenado...

Si necesitamos eliminación, es mejor usar encadenamiento

# Otros sondeos

## Sondeo lineal

- Si  $h(k) = H$ , para alguna constante  $d$  buscamos en

$$H, H + d, H + 2d, \dots$$

- Se debe cumplir  $d = 1$  o que  $d$  y  $m$  son primos relativos

## Sondeo cuadrático

- Si  $h(k) = H$ , buscamos en

$$H, H + 1, H + 4, H + 9, \dots$$

## Doble hashing

- Usamos dos funciones de hash  $h_1$  y  $h_2$  y buscamos en

$$h_1(k), h_1(k) + h_2(k), h_1(k) + 2h_2(k), \dots$$

Todos ellos presentan el problema de la eliminación

# Factor de carga

Dado que las colisiones impactan la tabla, nos interesa medir cuántos datos tenemos almacenados

## Definición

Dada una tabla de hash  $T$  de tamaño  $m$  con  $n$  valores almacenados, se define su **factor de carga** como

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

El factor de carga es una medida de *qué tan llena está la tabla*

Según la estrategia de resolución de colisiones

- Encadenamiento: es aceptable  $\lambda \approx 1$
- Direcccionamiento abierto:  $\lambda > 0.5$  resulta en inserciones y búsquedas muy lentas

# Rehashing

Si  $\lambda$  es grande y ya no es aceptable, las operaciones se vuelven costosas

Una solución es hacer **rehashing**

- Se crea una nueva tabla más grande
- Aproximadamente del doble del tamaño original
- Como el espacio de índices ya no es de tamaño  $m$ , se define una nueva función de hash
- Mover los datos a la nueva tabla

Esta es una operación costosa para tablas de hash

- Es  $\mathcal{O}(n)$  para  $n$  datos insertados
- No obstante, es **infrecuente**

# Sumario

Obertura

Tablas de hash

Colisiones

**Funciones de hash**

Epílogo

# Funciones de hash

## Definición

Dado un espacio de llaves  $K$  y un natural  $m > 0$ , una **función de hash** se define como

$$h : K \rightarrow \{0, \dots, m-1\}$$

Dado  $k \in K$ , llamaremos **valor de hash de  $k$**  a la evaluación  $h(k)$ .

¿Qué caracteriza al espacio de llaves  $K$ ?

- En principio podemos aceptar cualquier tipo de conjunto  $K$
- Haremos la distinción entre llaves **numéricas** y **no numéricas**
- En cualquier caso, nos interesa llevar esas llaves a  $\{0, \dots, m-1\}$
- En general, las no numéricas primero se convierten a entero

Nos centraremos en las llaves numéricas

# Funciones de hash

Nos interesa tener *buenas* funciones de hash

- Esperamos que el mapeo de llaves sea uniforme en el rango  $\{0, \dots, m-1\}$
- Debe ser fácil/barata de calcular

¿Cómo lograr esto? ¿o al menos acercarnos?

Sabemos que las llaves de  $K$ , en última instancia, se representan en binario

- Hay operaciones en binario muy eficientes
- Idea: que el valor  $h(k)$  dependa de todos/muchos bits de  $k$



# Funciones de hash

## Ejemplo

Consideremos  $m = 16$  (tamaño de la tabla) y  $k = 1234567$ . El cuadrado de la llave es  $k^2 = 1524155677489$ , cuya representación binaria tiene  $n$  bits y es

$$(k^2)_2 = (010110001011011110110000011111101100110001)_2$$

Nos centraremos en los  $m = 16$  bits intermedios

$$0101100010110\underline{1111011000001111}1101100110001$$

Consideremos el valor de hash  $h(k)$  dado por

$$(\underline{1111011000001111})_2 = (62991)_{10}$$

de manera que  $h(1234567) = 62991$

# Funciones de hash

## Ejemplo

Generalicemos este resultado para definir una función de hash que considere los  $m$  bits centrales de  $k^2$

Recordemos que en representación binaria

- Dividir  $a$  por  $2^b$  **borra** los  $b$  bits de la derecha de  $(a)_2$
- $(2^b)_2$  es un 1 seguido de  $b$  ceros
- Como  $[a \bmod 2^b]$  es el resto al dividir entre  $2^b$  y  $[a \bmod 2^b] < 2^b$ , entonces  $(a \bmod 2^b)_2$  son los  $b$  bits de la derecha en  $(a/2^b)_2$

# Funciones de hash

## Ejemplo

La función de hash que definimos es

$$h_m(k) = \left( \frac{k^2}{2^r} \right) \bmod 2^m$$

donde  $r = n - m/2$  es la cant. de bits ignorados en el lado derecho de  $(k^2)_2$

- Modificando  $m$  podemos alterar la cantidad de bits que *importan*
- Si  $m = n$ , entonces  $r = 0$  y

$$h_n(k) = \left( \frac{k^2}{2^0} \right) \bmod 2^n = k^2 \bmod 2^n = k^2$$

que corresponde a la función  $h_m$  que más bits de  $k^2$  incorpora

# Hashing modular

Una familia simple de funciones razonable, que ya estudiamos, es de la forma

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

y se conoce como **hashing modular** o **método de la división**

- Es fácil de calcular
- Distribuye uniformemente las llaves si estas son *aleatorias*

# Hashing modular

Como se vio en el ejemplo, no siempre calculamos el módulo de un *k puro*... puede ser resultado de operaciones entre otros números

Recordemos las propiedades de la aritmética modular

- $(a + b) \bmod c = ((a \bmod c) + (b \bmod c)) \bmod c$
- $(a \cdot b) \bmod c = ((a \bmod c) \cdot (b \bmod c)) \bmod c$

## Ejemplo

Calculemos  $29 \cdot 72 \bmod 13$  sin efectuar el producto  $29 \cdot 72$

$$\begin{aligned} 29 \cdot 72 \bmod 13 &= ((29 \bmod 13) \cdot (72 \bmod 13)) \bmod 13 \\ &= (3 \cdot 7) \bmod 13 \\ &= 21 \bmod 13 \\ &= 8 \end{aligned}$$

# Hashing modular

## Algunas propiedades del hashing modular

- Si  $m$  es muy pequeño, habrán muchas colisiones
- Si  $m = 2^b$ , solo serán relevantes los  $b$  bits menos significativos... pueden repetirse mucho ciertos patrones
- Además, no queremos que  $m$  tenga muchos divisores
  - Si  $m$  es par y las llaves  $k$  son pares,  $k \bmod m$  es par
  - ¡La mitad de la tabla va a estar vacía!

Es conveniente tomar  $m$  como primo, cercano al tamaño ideal de la tabla

# Método de la multiplicación

Otra familia razonable considera un real  $0 < A < 1$  y es de la forma

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

y se conoce como **método de la multiplicación**

- Se extrae la parte decimal de  $A \cdot k$
- A diferencia del hashing modular, el valor de  $m$  no es crítico
- Tomar  $m$  como potencia de 2 simplifica los cálculos

Es conveniente tomar  $m$  como primo, cercano al tamaño ideal de la tabla

# Llaves strings

Si las llaves son strings debemos convertirlas a enteros

Una forma de realizar la conversión a entero es usar códigos ASCII y concatenarlos

## Ejemplo

Si sabemos que

$$\text{ASCII}("A") = 64 = (01000001)_2 \quad \text{ASCII}("B") = 66 = (01000010)_2$$

entonces una posible interpretación numérica de  $s = "AB"$  es

$$(0100000101000010)_2 = 16706$$

Ahora bien, si

$s' = "PROXIMA SEMANA ELEGIMOS GANADORE DEL DATICONCURSO N UNO"$

¿Qué tan grande sería esta representación para  $s'$ ?



# Llaves strings

Una forma común es interpretar  $s$  como entero en una **base apropiada**

Si  $s = s_0s_1 \dots s_p$ , una forma de convertirlo a un entero  $k$  es

$$k = \#(s_p) + \#(s_{p-1}) \cdot R + \dots + \#(s_0) \cdot R^p = \sum_{i=0}^p \#(s_{p-i}) \cdot R^i$$

donde

- $\#(s_i) = \text{ASCII}(s_i)$
- $R \in \{31, 37\}$ , i.e. un número primo que permite que los bits de todos los  $s_i$  jueguen un rol

Con esta estrategia estamos interpretando a  $s$  como un número de  $p + 1$  dígitos en base  $R$

Finalmente, convertimos  $k$  al rango  $\{0, \dots, m - 1\}$  con alguna función de hash

# Sumario

Obertura

Tablas de hash

Colisiones

Funciones de hash

**Epílogo**

# Epílogo

Ve a

**www.menti.com**

Introduce el código

**4233 8475**



O usa el código QR