Tema 3 – S1

Map y Tabla de Dispersión (*Hash*)

Contenidos

1. El Modelo Map: definición y ejemplos de uso

1. El modelo Map

*¿Qué sabemos **ya** de la gestión de datos según un Diccionario?... Recuerda el Video -Ejemplo 2 del Tema 1*

Ejemplo 2 - Aplicación de gestión de una Colección de Entradas y su coste: **Modelo Diccionario**

1. El modelo Map

Definición

Un Map es un tipo de Diccionario...

- ➔ Modelo de gestión de datos: Búsqueda Dinámica de un Dato de Clave dada en una Colección

El método **`equals`** permite comprobar si dos claves son iguales o no

- ➔ Sus Datos, o **Entradas**, son Pares (Clave c, Valor v)

En realidad, se busca la Entrada de Clave c **para acceder a su Valor v, la información que se desea recuperar**

... **en el que NO se permiten claves repetidas**

1. El modelo Map

Definición en Java: la interfaz `Map`

```
package librerias.estructurasDeDatos.modelos;

public interface Map<C, V> {

    /** inserta/actualiza la Entrada(c, v) en un Map y devuelve
     *  el valor que estuviera ya asociado a su clave, null si
     *  no existe una Entrada con dicha clave */
    V insertar(C c, V v);

    /** elimina la Entrada con clave c de un Map y devuelve su
     *  valor asociado, null si no existe una Entrada con dicha clave
     */
    V eliminar(C c);

    /** devuelve el valor asociado a la clave c en un Map,
     *  null si no existe una Entrada con dicha clave */
    V recuperar(C c);

    /** comprueba si un Map está vacío */
    boolean esVacio();

    /** devuelve la talla, o número de Entradas, de un Map */
    int talla();

    /** devuelve una ListaConPI con las talla() claves de un Map */
    ListaConPI<C> claves();
}
```

1. El modelo Map

Ejemplo de uso: obtener el vocabulario de un texto

Existen múltiples aplicaciones que usan un Map: traducción de textos, agenda electrónica, cálculo de estadísticas sobre las palabras que aparecen en un texto (frecuencia, moda, etc.)...

El siguiente ejemplo pretende subrayar sus características principales: ¿por qué usar un Map en el problema planteado o, mejor dicho, qué Búsqueda por qué Clave hay que plantear en él? Establecida la Clave, y su tipo, ¿cuál es el Valor, y tipo, asociado a la Clave? Establecido el tipo de Entradas del Map de la aplicación, ¿cómo usar qué métodos de Map en esta aplicación?

Ejemplo de uso de un Map: obtener el vocabulario de un texto

Ejercicios (I)

Pensando siempre primero qué Map es más apropiado definir y qué tipo de Clave y Valor tiene cada una de sus Entradas, modifica convenientemente el programa `Test1Map` para...

Ejercicio 1: diseñar un nuevo programa `Test2Map` que lea un texto de teclado y muestre por pantalla el número de palabras distintas que contiene

Ejercicio 2: diseñar un nuevo programa `Test3Map` que lea un texto de teclado y muestre por pantalla un listado en el que cada línea contenga una palabra repetida del texto -que haya aparecido más de una vez en él- y el nº de veces que se repite en él –su frecuencia de aparición

Ejercicio 3: Número de palabras con frecuencia mayor que una dada

(clave: CCDHL00Z)

Diseñar un método `frecuenciaMayorQue` que devuelva el nº de palabras que aparecen en un texto con una frecuencia mayor que `n`. Por ejemplo, si `n=0` devolverá el nº de palabras distintas que aparecen en el texto; si `n=1` devolverá el nº de palabras repetidas que tiene el texto, etc.

Ejercicios (II)

Una de las más conocidas, útiles y sencillas aplicaciones que maneja un Map es la Traducción Bilingüe, Palabra a Palabra, de un texto: el traductor de la aplicación usa un Diccionario Bilingüe

Ejercicio 4: completa el método `traducir` del programa `Test4Map` para que dado un texto en Castellano, `textoC`, devuelva su traducción al Inglés, palabra a palabra, con ayuda de un Diccionario Bilingüe `d`

```
public static String traducir(String textoC, Map<String, String> d)
```

Para ello, dado que `Test4Map` ya tiene un método `cargarDiccionario` que crea `d` a partir de las Entradas del fichero *diccioSpaEng.txt*, SOLO debes tener en cuenta lo siguiente:

- Las Entradas que contiene `d` son pares de la forma

  (Clave = Palabra en Castellano p**, Valor =** Traducción al Inglés de p **)**

- Si `d` no contiene la traducción de una palabra de `textoC`, el `String` resultado de `traducir` debe contener el literal "<error>" en lugar de su traducción

Tema 3 – S2

Map y Tabla de Dispersión (Hash)

Contenidos

2. Tabla de Dispersión (Hash)
   - Concepto de dispersión: implementación de la Búsqueda Dinámica por Clave en tiempo constante
   - Función de Dispersión: valor hash (método `hashCode()`) e índice hash de una Clave (función de compresión)
   - Colisiones: origen y resolución mediante Encadenamiento Separado (o *Hashing* Enlazado)
   - Eficiencia: Factor de Carga y *Rehashing*

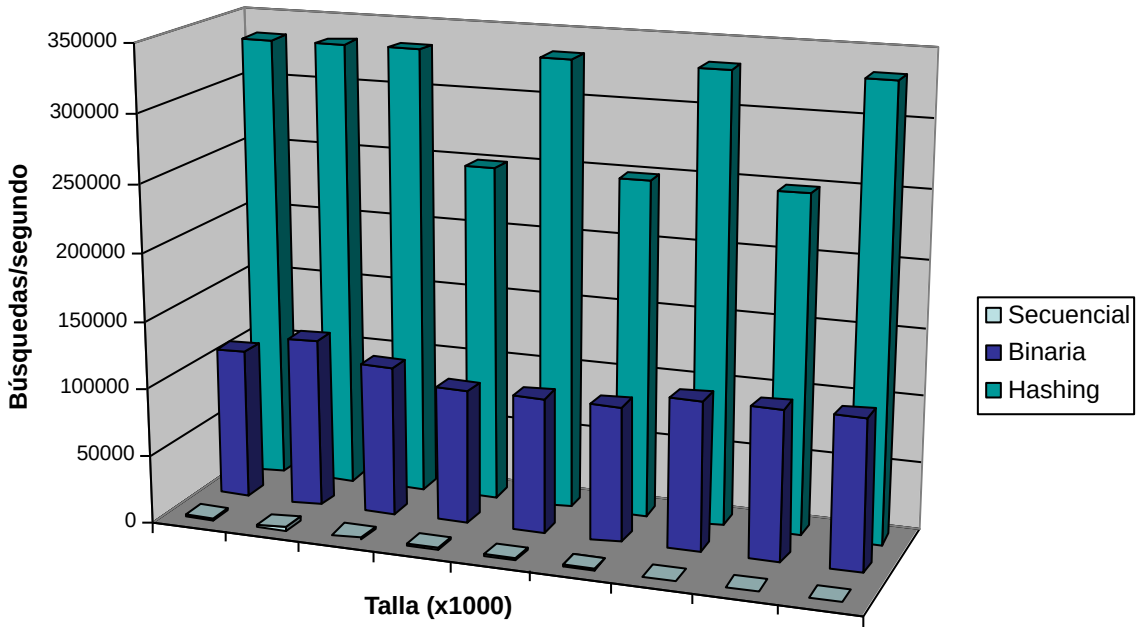
2. Tabla de Dispersión (*Hash*)

*¿Qué sabemos **ya** del coste de la gestión de un Diccionario?*

El coste promedio de `localizar` una clave en un Map será...

- Lineal si el Map se representa con un array no ordenado o una LEG, ordenada o no (Búsqueda Secuencial)
- Logarítmico si el Map se representa con un array ordenado (Búsqueda Binaria)
- **Constante si el Map se representa con una Tabla Hash**, un array "especial"

¿Cómo?



Secuencial	1471	2000	1351	1053	826	680	578	503	441
Binaria	111110	125002	111110	100000	100000	100000	111110	111110	111110
Hashing	333331	333357	333331	250003	333331	250003	333331	250003	333331

2. Tabla de Dispersión *(Hash)*

***Ejemplo de dispersión**, una forma de implementar la Búsqueda Dinámica por Clave en tiempo constante*

Se quiere digitalizar un **Diccionario Países Europeos - Capitales**, cuyas **Entradas** son de la forma **Clave** = País Europeo, **Valor** = Capital del País. Para ello, se ha pensado en usar un **Map** implementado mediante un array. Así, por ejemplo, **para buscar la capital de Bulgaria**, ...

+ Función de Dispersión(c) = orden(c) -1

Diccionario Países - Capitales

Implementación

`Entrada<String, String>[] elArray`

Albania	Tirana
Alemania	Berlín
Andorra	Andorra la Vieja
Austria	Viena
Bélgica	Bruselas
Bielorrusia	Minsk
Bosnia y Herzegovina	Sarajevo
8º Bulgaria	Sofía
Ciudad del Vaticano	Ciudad del Vaticano
Croacia	Zagreb
Dinamarca	Copenhague
Eslovaquia	Bratislava
Eslovenia	Liubliana
España	Madrid
...	

0	Albania	Tirana
1	Alemania	Berlín
2	Andorra	Andorra la Vieja
3	Austria	Viena
4	Bélgica	Bruselas
5	Bielorrusia	Minsk
6	Bosnia y Herzegovina	Sarajevo
7	Bulgaria	**Sofía**
8	Ciudad del Vaticano	Ciudad del Vaticano
9	Croacia	Zagreb
10	Dinamarca	Copenhague
11	Eslovaquia	Bratislava
12	Eslovenia	Liubliana
13	España	Madrid
...	...	

La idea básica para esta Implementación es INDEXAR el array por Claves

Función de dispersión de Claves, que asocia un **ÍNDICE** del array a cada clave

2. Tabla de Dispersión

Función de dispersión: ¿cómo implementarla "adecuadamente"?

Idea básica: **indexar** `elArray` **por la Clave** de la Entrada que se quiere insertar, eliminar o recuperar en él

Problema: definir una Función de Dispersión "adecuada", i.e. que indexa cada Clave `c` a un índice distinto del array en O(1)

Solución: como un `String c` es un array de `char` y un `char` su código ASCII, ...

$$\texttt{valorIntC} = \texttt{c.charAt(0)} * \texttt{base}^{\texttt{c.length()-1}} + \ldots + \texttt{c.charAt(c.length() - 1)} * \texttt{base}^{0}$$

```
String s1 = "hola", s2 = "teja";
int base = 1; // ¿Es 1 una base adecuada? Recuerda que 1 elevado a lo que sea es 1
int valorIntS1 = s1.charAt(0) * 1 + s1.charAt(1) * 1 + s1.charAt(2) * 1 + s1.charAt(3) * 1;
int valorIntS2 = s2.charAt(0) * 1 + s2.charAt(1) * 1 + s2.charAt(2) * 1 + s2.charAt(3) * 1;
valorIntS1
420  (int)
valorIntS2
420  (int)
```

Colisión

```
base = 2; // ¿Es 2 una base mejor? Recuerda: 2^3 = 8; 2^2 = 4; 2^1 = 2; 2^0 = 1
valorIntS1 = s1.charAt(0) * 8 + s1.charAt(1) * 4 + s1.charAt(2) * 2 + s1.charAt(3) * 1;
valorIntS2 = s2.charAt(0) * 8 + s2.charAt(1) * 4 + s2.charAt(2) * 2 + s2.charAt(3) * 1;
valorIntS1
1589  (int)
valorIntS2
1641  (int)
// Mejor valor de base = 2, porque "pesa" cada carácter de la clave por su posición
```

2. Tabla de Dispersión

Función de dispersión: ¿cómo implementarla en Java?

Ideas básicas:

- Indexar `elArray` por Clave
- Definir una Función de Dispersión "adecuada"

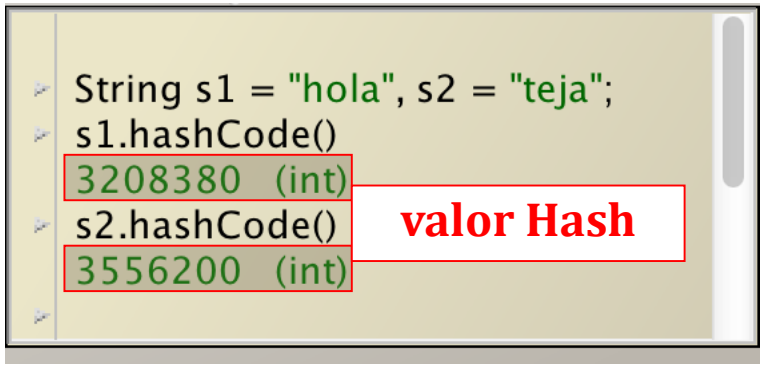
PERO, ¿cómo y dónde se hace esto en Java?

Java define una Función de Dispersión **estándar: el método `hashCode()` de la clase `Object`**

- La clase de la Clave **debe sobrescribir** `hashCode()` de `Object` y, por tanto, también su método `equals`
- En la clase que implementa el `Map` **siempre se localiza** la Entrada de Clave `c` mediante la instrucción `elArray[c.hashCode()]`

Volveremos luego a esta afirmación

Ejemplo de sobrescritura de `hashCode()` de la clase `String`:



2. Tabla de Dispersión

Función de dispersión: el método `hashCode` *de* `Object`

`public int hashCode()` returns a hash code value for the object...

The general contract of `hashCode` is:

- Whenever it is invoked **on the same object** more than once during an execution of a Java application, **the `hashCode` method must consistently return the same integer**, provided no information used in equals comparisons on the object is modified...

- **If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.**

- **It is not required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results.** However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables...

2. Tabla de Dispersión

Función de dispersión: el método `hashCode` *de* `String`

`public int` **`hashCode`**`()` returns a hash code for this string...

The hash code for a String object is computed as

$$s[0] * 31^{(n-1)} + s[1] * 31^{(n-2)} + \ldots + s[n-1]$$

using **`int`** arithmetic, where s[i] is the i-th character of the string, n is the length of the string, and ^ indicates exponentiation. (The hash value of the empty string is zero.)

Overrides: `hashCode` in class `Object` ...

¿Cómo está implementado?

```
public int hashCode() {
    int valorHash = 0;
    for (int i = 0 ; i < this.length(); i++) {
        valorHash = 31 * valorHash + this.charAt(i);
    }
    return valorHash;
}
```

→ **NI usa** `Math.pow` **NI usa** `Math.exp` ←

```
"hola".hashCode()
3208380 (int)
"a".hashCode()
97 (int)
"eda".hashCode()
100258 (int)
"".hashCode()
0 (int)
```

2. Tabla de Dispersión

***Función de dispersión**: otro ejemplo de uso del método `hashCode` de `String`*

En una aplicación de control del número de veces que una dirección Web ha sido visitada se requiere el uso de un `Map<DireccionHTTP, Integer>`. Si dicho Map se implementa mediante una Tabla Hash, `DireccionHTTP` es **la clase de la Clave** de las Entradas a situar sobre `elArray` que tiene dicha Tabla.

```
public class DireccionHTTP {
    private String servidor, dirRecurso;
    private int puerto;
    public DireccionHTTP(...){…}
    …
}
```

NOTA: una dirección http, como por ejemplo *http://www.dsic.com:80/datos/fichero.txt*, consta de: un servidor (*www.host.com*), un puerto (*80*) y una dirección al recurso solicitado (*/datos/fichero.txt*)

1.- ¿Cuál sería la Clave que permitiría localizar una dirección Web?

PISTA: piensa qué te lleva a decir que dos direcciones Web son iguales

2.- ¿Qué dos métodos tiene que definir `DireccionHTTP` para poder ser la clase de la Clave de un Map implementado con una Tabla Hash?

2. Tabla de Dispersión

***Función de dispersión**: índice hash de una Clave - Función de compresión*

Problema: el valor Hash puede ser muy grande AND...

(1) NI la memoria es Infinita

(2) NI la longitud de una Clave se puede reducir

```
c        99
l        3177
i        98592
m        3056461
a        94750388
----------------
clima    94750388
```

```
s        115
a        3662
l        113630
t        3522646
o        109202137
------------------
salto    109202137
```

Función de Compresión

Solución: función de conversión valor Hash – **Índice Hash**

[0, elArray.length[

2. Tabla de Dispersión

***Función de dispersión**: método de la división para implementar la Función de Compresión*

Procedimiento para dispersar una Clave `c` en `elArray` :

```
int valorHash = c.hashCode();
// Método de la división: si valorHash ≥ 0…
int  indiceHash = valorHash % elArray.length;
// 0 ≤ indiceHash < elArray.length
```

Clave `c`	Valor Hash	Índice Hash (`elArray.length`=101)
`clima`	94750388	66
`salto`	109202137	28

¡¡ Para que la dispersión sea efectiva el tamaño de `elArray` debe de ser un número primo !!

2. Tabla de Dispersión

Función de dispersión: overflow de enteros

Procedimiento para dispersar una Clave `c` en `elArray` :

```
int valorHash = c.hashCode();
// Método de la división: si valorHash □ 0… ¿ Y sino?
int  indiceHash = valorHash % elArray.length;
```

Overflow de enteros: cuando `valorHash` es lo bastante grande como para **NO** poder almacenarse en una variable de **4** bytes se pueden descartar algunos bits de orden superior y, siendo la representación de un número en complemento a 2, se podría llegar a obtener un número negativo. Como la operación módulo respeta el signo hay que hacer la comprobación…

```
if (indiceHash < 0) { indiceHash += elArray.length; }
```

2. Tabla de Dispersión

Función de dispersión: solución al overflow de enteros

Procedimiento para dispersar una Clave `c` en `elArray`:

```
int valorHash = c.hashCode();
// Método de la división: si valorHash ≥ 0…
int  indiceHash = valorHash % elArray.length;
// Overflow de enteros: corrección
if (indiceHash < 0) {
    indiceHash += elArray.length;
}
// 0 ≤ indiceHash < elArray.length
```

Clave `c`	Valor Hash	Índice Hash (`elArray.lenght`=101) **CON** overflow	Índice Hash (`elArray.lenght`=101) **SIN** overflow
clima	94750388	66	66
salto	109202137	28	28
poesía	-982864703	-70	31
novela	-1039634011	-5	96

2. Tabla de Dispersión

Función de dispersión, finalmente!!!

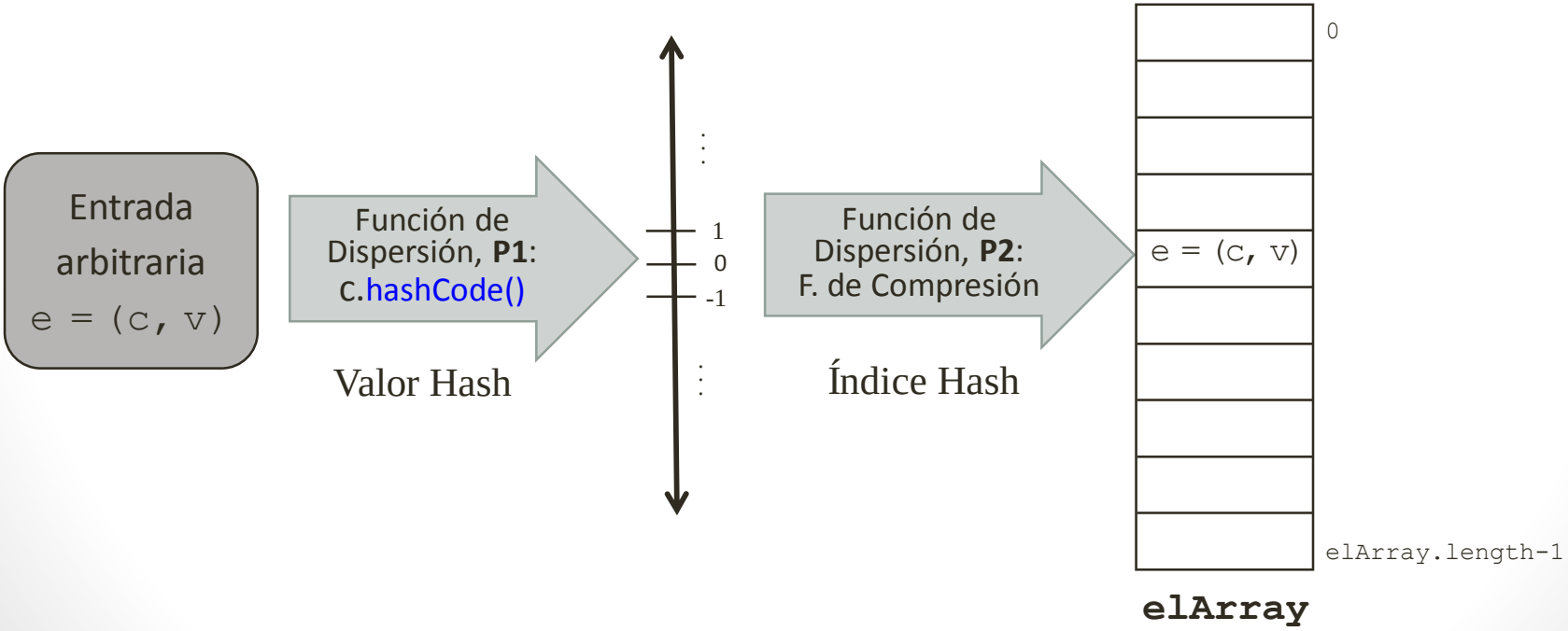
```
public int indiceHash(C c) {

    int valorHash = c.hashCode();
     // Método de la división: si valorHash □ 0…
     int  indiceHash = valorHash % elArray.length;
     // Overflow de enteros: corrección
     if (indiceHash < 0) {
         indiceHash += elArray.length;
     }
     // 0 □ indiceHash < elArray.length
     return indiceHash;
}
```

2. Tabla de Dispersión

***Resumen**: de un array a una Tabla de Dispersión*

Al indexar `elArray` por Clave, la Función de Dispersión hace de un array un soporte válido y eficaz (O(1)) para una Colección de Entradas sobre las que se quiere efectuar la Búsqueda Dinámica

¡Es la Función de Dispersión la que convierte un simple array en una Tabla Hash!



2. Tabla de Dispersión

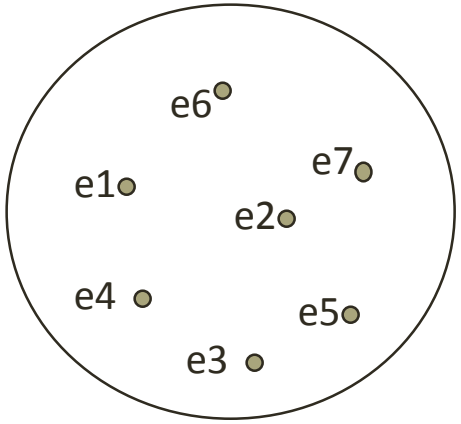
Colisiones: origen

¿Es siempre inyectiva la función de dispersión?

Ejemplo 1: se quiere representar un Map de 7 Entradas mediante la Tabla Hash con función de dispersión **Inyectiva** (caso **Ideal**); en el dibujo se ve su estado tras insertar e1, e2 y e3

- ¿Qué sucede al volver a `insertar e1 = (c1, v1)`?
- ¿Cómo `eliminar(c4)`? ¿y `eliminar(c1)`?
- ¿Cómo `recuperar(c2)`?

Map de 7 Entradas



`elArray[]` `talla = 3`

null	0
e2	1
e1	2
null	3
…	…
null	
null	
e3	
null	
null	`elArray.length-1`

2. Tabla de Dispersión

***Colisiones**: origen*

¿Es siempre inyectiva la función de dispersión?

Ejemplo 2: se quiere representar un Map de 3 `Integer` (clave = valor) mediante una Tabla Hash del mismo tamaño y `c.hashCode() = c.intValue()`

- ¿Qué sucede al `insertar` en la Tabla vacía `(9, 9)`, `(58, 58)` y `(89, 89)`?
- ¿Qué sucede al ejecutar `eliminar(9); recuperar(9); insertar(9, 9);`?
- ¿Qué sucede al `insertar(18, 18)`? ¿Y al `insertar(49, 49)`? ¿Y al `insertar(3, 3)`? ¿Por qué?

0	null
1	null
2	null

2. Tabla de Dispersión

Colisiones: origen, definición y resolución

Origen: la función de dispersión (`indiceHash`) **NO es Inyectiva** (caso **Real**), asocia el mismo índice Hash (posición de `elArray`) a Claves distintas

- Porque el método `hashCode` elegido no dispersa bien, i.e. no proporciona un rango de valores Hash suficientemente amplio para discriminar las distintas Entradas de una Colección - por rápido que sea, no pesa bien los distintos bits de las Claves

  Elección cuidadosa de `hashCode`

- Porque el grado de correlación que muestran las Claves de una aplicación impide que la Función de Compresión `valorHash % elArray.length` sea uniforme en la práctica

  Elección cuidadosa de un nº primo para `elArray.length`

- Por el overflow que causa la limitación de la representación en memoria

Definición: dos claves `c1` y `c2`, **colisionan** cuando

```
!c1.equals(c2) & indiceHash(c1)= indiceHash(c2)
```

¿Cómo minimizar las colisiones?

¿Cómo resolverlas?

Ejercicios de la Sesión 2:

Para que te familiarices con los conceptos básicos sobre la función de dispersión de una Tabla Hash y su implementación en Java, hemos preparado los siguientes ejercicios:

Ejercicio 5: Examen poli formaT

Tema 3 - S2: Cuestiones sobre *Hashing* I

Ejercicio 6: en una aplicación que controla el número de veces que ha pasado un determinado coche por un radar superando el límite de velocidad se usa un `Map<Matricula, Integer>` implementado mediante una Tabla Hash.

La clase Matricula (clave: CCDHN00Z)

Sabiendo que una matrícula es igual a otra si los números y letras que tiene una coinciden con los que tiene la otra, añade a la clase `Matricula` que se te proporciona aquellos métodos que consideres necesarios para que pueda ser la clase de las clave del Map de la aplicación

Tema 3 – S3

Map y Tabla de Dispersión (*Hash*)

Contenidos

2. Tabla de Dispersión
   - Concepto de dispersión: implementación de la Búsqueda Dinámica por Clave en tiempo constante
   - Función de Dispersión: valor hash (método `hashCode()`) e índice hash de una Clave (función de compresión)
   - Colisiones: origen y resolución mediante Encadenamiento Separado (o Hashing Enlazado)
   - Eficiencia: Factor de Carga y Rehashing
3. Implementación de una Tabla de Dispersión con Hashing Enlazado: las clases `TablaHash` y `EntradaHash`

2. Tabla de Dispersión

Colisiones: *métodos de resolución*

Aún con una buena función de dispersión, las colisiones son posibles... ¿Cómo resolverlas?

Métodos para la resolución de colisiones:

- **Direccionamiento abierto:** si vamos a insertar una Entrada en una posición y esa posición está ocupada, se busca la primera posición libre alternativa (Exploración Lineal y Cuadrática)
- **Encadenamiento separado, o *Hashing* Enlazado**

26

2. Tabla de Dispersión

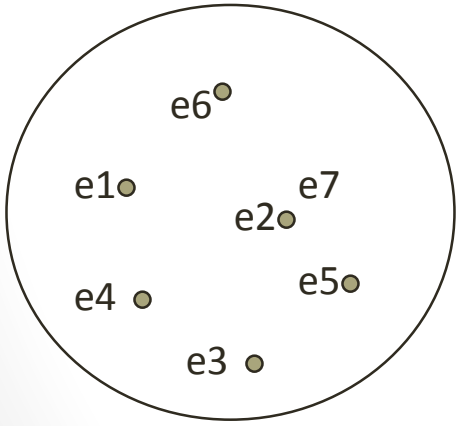
Colisiones: resolución por Hashing Enlazado

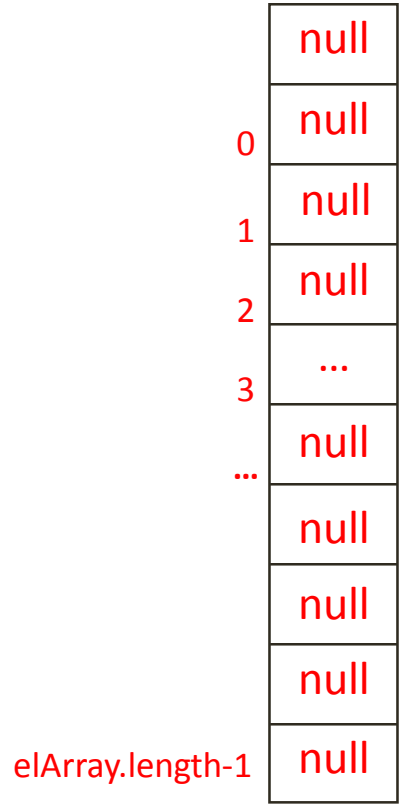
Todas las Entradas que colisionan en una misma posición de `elArray` se almacenan en la **Lista de Colisiones**, o **Cubeta**, asociada a tal posición

`e = (c, v)` está en la Cubeta `elArray[indiceHash(c)]`

Ejemplo 3: se quiere representar un Map de 7 Entradas mediante una Tabla con Hashing Enlazado...

Map de 7 Entradas





talla = 0

2. Tabla de Dispersión
Eficiencia

Fijado `hashCode` y suponiendo constante su tiempo de ejecución, ...

Una Tabla Hash de x = `talla` Entradas será eficiente siempre que el tiempo promedio que se tarda en localizar en ella cualquier clave c ($T^{\mu}_{localizarC}(x)$) sea constante

Dado que...

- Una Entrada de clave `c` SOLO puede ubicarse en la cubeta `elArray[indiceHash(c)]`, **$T^{\mu}_{localizarC}(x)$ será directamente proporcional a la longitud MEDIA de las cubetas de la Tabla**
- **La longitud de la cubeta `elArray[indiceHash(c)]` es exactamente el número de colisiones que provoca la clave `c`**, o nº de Entradas que colisionan con la de clave `c`

SII el nº medio de colisiones, o longitud media de las cubetas, es muy pequeño (cada clave provoca 1 o 2 colisiones en promedio), $T^{\mu}_{localizarC}(x)$ **es constante**

¿Existe algún parámetro de una Tabla con *Hashing* Enlazado que represente el nº medio de colisiones o longitud media de las cubetas?

2. Tabla de Dispersión

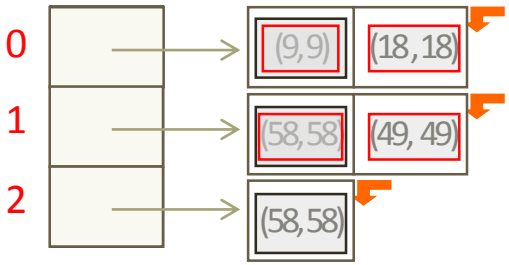
Eficiencia y Factor de Carga (FC)

¿Existe algún parámetro de la Tabla con *Hashing* Enlazado que represente el número medio de colisiones o longitud media de las cubetas?

Su Factor de Carga (FC), ya que…

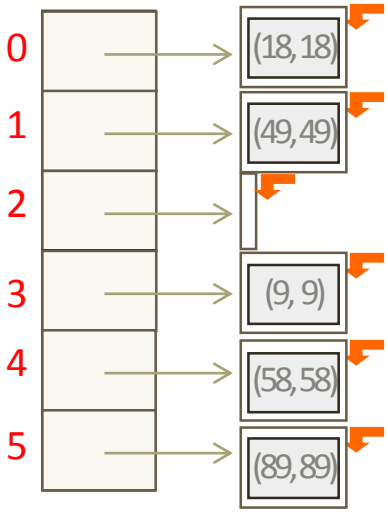
- Por definición, FC = `talla / elArray.length` (=1 en el caso ideal)
- En una Tabla con *Hashing* Enlazado el FC coincide, como se observa en los siguientes ejemplos, con la longitud MEDIA de las cubetas

FC = 5 / **3**



$T^{\mu}_{localizarC}(x) = 7 / 5 \in O(FC)$

FC = 5 / **6**



$T^{\mu}_{localizarC}(x) = 5 / 5 \in O(FC)$

La eficiencia de una Tabla con *Hashing* Enlazado de `talla` Entradas se mide en términos de su FC

¿Asegurar un FC < 1? p. ej. 0.75 (en promedio, 3/4 de las cubetas con una longitud 1 ó 2 y el resto 0)

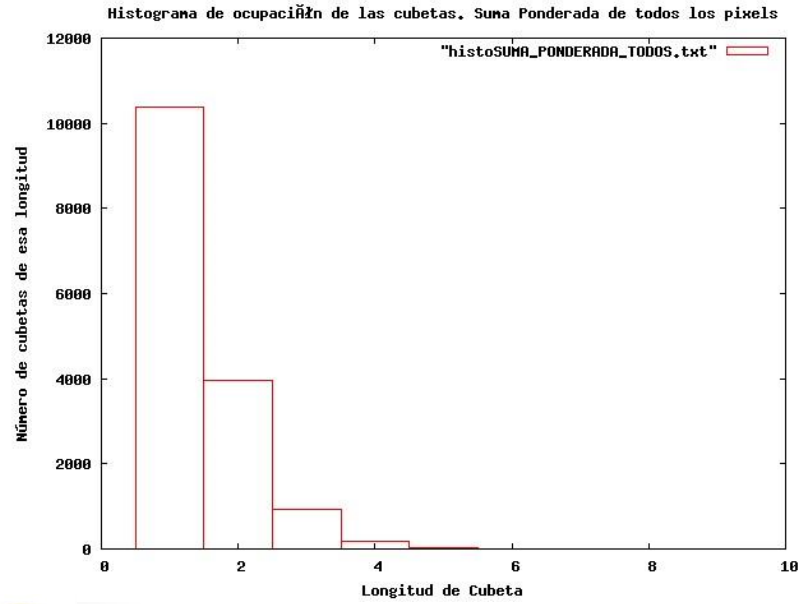
Elegir un "buen" `elArray.length`: la talla MÁXIMA estimada del Map convertida en un nº primo, si no lo es

2. Tabla de Dispersión

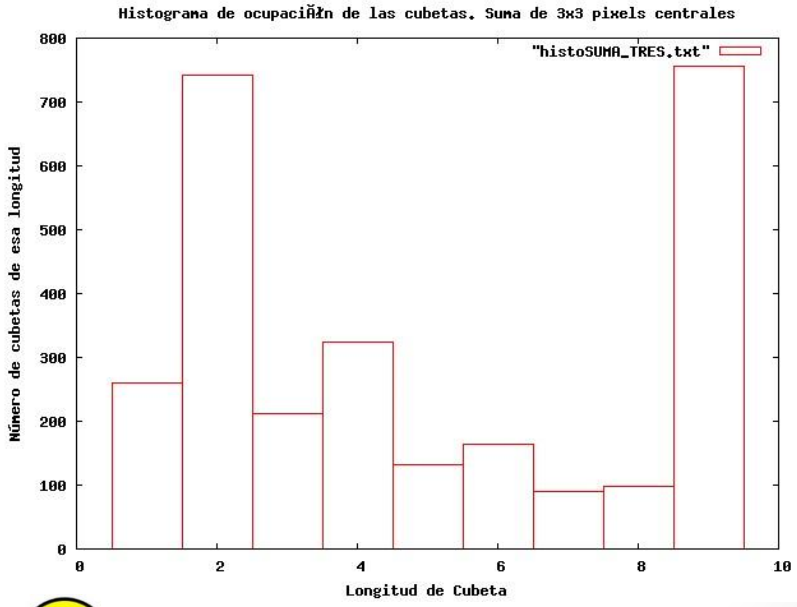
Eficiencia: análisis experimental

Ejemplo de análisis del comportamiento de una Tabla Hash mediante el cálculo y análisis de su histograma de ocupación y la desviación típica de las longitudes de sus cubetas

Las siguientes gráficas muestran los histogramas de ocupación y desviaciones típicas de una Tabla Hash con FC=0.75, que contiene 22.000 imágenes de 11 x 3 píxeles según dos funciones de dispersión... **¿Cuál es la mejor?**



Desviación típica < 0.86



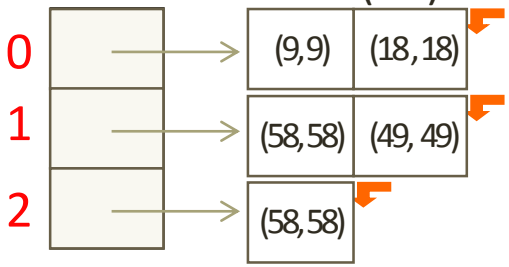
Desviación típica > 12

2. Tabla de Dispersión

Eficiencia: motivación y definición de Rehashing

- Si por cualquier motivo, como por ejemplo una mala estimación de su capacidad, FC→1 o FC>1, i.e. aumenta la longitud **media** de las cubetas...

$FC = 5/3 \rightarrow T^{\mu}_{localizarC}(x) = 7/5 \in O(FC)$

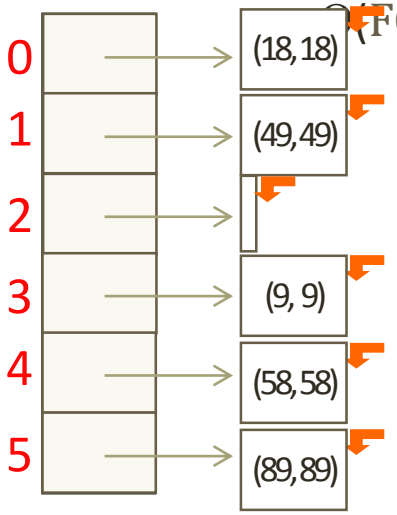


➔ ***Rehashing***:

1) Duplicar la capacidad de la Tabla

2) Volver a dispersar las Entradas, lo que reduce el FC y, por tanto, mejora la eficiencia

$FC = 5/6 \rightarrow T^{\mu}_{localizarC}(x) = 5/5 \in O(FC)$



Pero, al hacer *Rehashing*, **¿sigue siendo O(1) el coste de las op. básicas?**

2. Tabla de Dispersión

Eficiencia: conclusiones sobre el Rehashing

El método `rehashing()`, que se implementará en la práctica 3, tiene un coste ¡¡¡lineal!!! con la talla de una Tabla. Dado que el método será invocado, siempre que sea necesario, desde el método `insertar`…

Una solución de compromiso para hacer el mínimo nº de operaciones de *rehashing* y, a la vez, no despilfarrar memoria es:

1. Fijar un FC < 1 (FC = 0.75) y determinar una "buena" capacidad inicial de la Tabla (`elArray.length`)
2. Hacer *rehashing* **SII** el factor de carga ACTUAL de la Tabla supera a FC

Sí, vale. Pero si se llega a hacer *rehashing*…

¿Sigue siendo O(1) el coste de las operaciones básicas?

Sí

Se puede demostrar que el coste *promedio* por inserción es constante o, alternativamente, que el coste total de M inserciones, incluidas las del *Rehashing*, es O(M)

3. Implementación

*Clases para la implementación de una Tabla Hash como un **array de Listas con PI de Entradas***

```
package librerias.estructurasDeDatos.deDispersion;
class EntradaHash<C, V> {
    //Una EntradaHash TIENE...
    C clave; V valor;
    public EntradaHash(C clave, V valor) {
        this.clave = clave; this.valor = valor;
    }
}
public class TablaHash<C, V> implements Map<C, V> {
    protected ListaConPI<EntradaHash<C,V>>[] elArray;
    protected int talla;
    protected int indiceHash(C c) {…}
   ...
}
```

Analiza el resto del código de la clase `TablaHash`

Ejercicios de la Sesión 3:

Para que te familiarices con los conceptos básicos sobre la implementación eficiente en Java del *Hashing* Enlazado, hemos preparado un cuestionario en PoliformaT

Ejercicio 7: Examen poli formaT

Tema 3 - S3: Cuestiones sobre *Hashing* II

Tema 3 – S4

Map y Tabla de Dispersión (Hash)

Contenidos

- Ejercicios a resolver
- Ejercicios propuestos

Ejercicios a resolver

Ejercicio 8: añade a la clase `TablaHash` un método que dada una Entrada de clave `c` devuelva el número de colisiones que provoca su localización; por simplificar, supón que no existe en la Tabla ninguna Entrada con clave `c`

PISTA: en la clase `TablaHash` solo hay una `ListaConPI` que representa las colisiones que provoca la clave `c`

Ejercicio 9: Barajar

(clave: CCDHM00Z)

Utilizando única y exclusivamente los métodos de la clase `TablaHash` y con coste lineal, diseña un método `barajar` que dado un array `v` devuelva un `String` con las componentes de `v` barajadas, i.e. desordenadas

PISTA: una Tabla Hash permite barajar, o desordenar, los elementos de `v` gracias a su método `indiceHash(C c)`, que implementa su Función de Dispersión

OTRA PISTA: las claves de la Tabla serán del mismo tipo que las componentes del array a barajar: `Integer`

Ejercicios a resolver

Ejercicio 10: en la aplicación de radares de tráfico del **Ejercicio 6**, que usa un `Map<Matricula, Integer>` implementado mediante una Tabla Hash, se pide:

Registrar matrícula (clave: CCDHO00Z)

Sabiendo que contabilizar las veces que un coche ha excedido el límite de velocidad al pasar por un radar determinado requiere actualizar *convenientemente* el Map de la aplicación, diseña el método que se encarga de hacerlo, con perfil...

```
public static void registrarM(Map<Matricula, Integer> dApp,
                              Matricula m)
```

¿Qué operación de un Map permite registrar una matrícula?

Piensa que, igual, la matrícula YA está registrada, por lo que el radar simplemente habrá "pillado" otra vez al mismo conductor

Ejercicios a resolver

Ejercicio 11: dadas dos Listas Con Punto de Interés `l1` y `l2`, el siguiente método obtiene un `String` en el que aparecen cada uno de los elementos que conforman su unión seguido del número de veces que éste aparece repetido en las listas

```
public static <E> String union(ListaConPI<E> l1, ListaConPI<E> l2) {
    ListaConPI<Par> aux = new LEGListaConPI<Par>();
    for (l1.inicio(); !l1.esFin(); l1.siguiente()) {
        E e = l1.recuperar();
        for (aux.inicio(); !aux.esFin() && !e.equals.aux.recuperar().dato); aux.siguiente()) { ; }
        if (aux.esFin()) { aux.insertar(new Par(e, 1)); }
        else { aux.recuperar().frec++; }
    }
    for (l2.inicio(); !l2.esFin(); l2.siguiente()) {
        E e = l2.recuperar();
        for (aux.inicio();!aux.esFin() && !e.equals(aux.recuperar().dato); aux.siguiente()) { ; }
        if (aux.esFin()) { aux.insertar(new Par(e, 1)); }
        else { aux.recuperar().frec++; }
    }
    return aux.toString();
}
```

donde la clase `Par` se define como sigue:

```
class Par<E> {
   E dato; int frec;
   Par(E d, int f) { dato = d; frec = f; }
   String toString() { return (dato.toString() + "-" + frec + " " ); }
}
```

Tras contestar a las preguntas que aparecen en la próxima transparencia, rediseña el método `union` para que su coste sea lineal con la suma de las tallas de `l1` y `l2`

Ejercicios a resolver

Preguntas para resolver el Ejercicio 11: si queremos rediseñar el método `union` es porque es ineficiente, pero...

1. **¿Cuánto?** Analiza su complejidad Temporal (talla, casos Peor y Mejor si los hubiera, costes con notación asintótica, etc.) para saber de qué hablamos

2. **¿Dónde?** Indica aquellas "zonas" de su código donde creas que reside la carga temporal a rebajar y razona tu elección

3. **¿Quién?** Indica qué EDA conoces que puede hacer lo mismo que en las zonas señaladas en la pregunta anterior pero con menor coste: Modelo, Implementación eficiente y, en su caso, la(s) instancia(s) de su tipo(s) genérico(s) que te sirve(n) en este problema concreto. Razona tu elección en términos de la mejora de coste que, a priori, supone

4. **¿Cómo?** Indica cómo usarías dicha EDA en el problema planteado (métodos de su Modelo o Implementación y orden de aplicación)

Hecho esto, finalmente, rediseña el método `union` para que su coste sea lineal con la suma de las tallas de `l1` y `l2`

Soluciones de los ejercicios 8 y 11

Ejercicios propuestos

Ejercicio 12: usando un Map, diseña un método (estático) `modaDe` que obtenga la moda de un array genérico `v`, i.e. que devuelva el primer elemento de `v` que se repite más veces

¿Por qué no usar un array de contadores, como en IIP?

Ejercicio 13: usando un Map, diseña un método (estático) `eliminarRepetidos` que, como su nombre indica, borre todos los elementos repetidos de `l`, una Lista Con PI de tipo genérico `E` en un tiempo Lineal con la talla de `l`

Por ejemplo, si `l` es la Lista de `Character` `{A, A, B, C, B, D}`, tras ejecutar `eliminarRepetidos l` será `{A B C D}`

¿Qué "pinta" un Map en este método?

Piensa que la Búsqueda es su operación básica y que, precisamente, lo mejor que se puede hacer con un Map es buscar muy rápido

Ejercicios propuestos

Ejercicio 14: como parte de una aplicación de control de acceso a un sistema de reservas, se desea desarrollar un módulo de autorización que permita:

- Conceder autorización a un usuario dado su nombre y su contraseña
- Conocer si un usuario está autorizado en el sistema a partir de su nombre y su contraseña. Se considera que un usuario está autorizado si su nombre se encuentra registrado y la contraseña que proporciona para acceder al sistema de reservas coincide con la que suministró al registrarse

Diseña completamente el sistema de acuerdo a las siguientes indicaciones:

(a) Clase `Usuario`: tiene como atributos `nombre` y `password`

¿Qué métodos hay que definir en la clase para que sea la clave de una Tabla Hash?
Piensa en el ejercicio "La clase Matricula" que has hecho ya en CAP

(b) Clase `ModuloAutorizacion`: tiene un `Map` de `Usuarios` con los métodos

```
public void registrarUsuario(String nombre, String pwd)
    throws UsuarioExistente
public boolean estaAutorizado(String nombre, String pwd)
```

(c) Programa `TestModuloAutorizacion`: concede, o no, la autorización a un usuario tras comprobar que, efectivamente, está registrado

Ejercicios propuestos

Ejercicio 15: diseña una clase `GestorNotas` que, haciendo uso de un Map, permita almacenar y consultar las notas de los alumnos en cada una de las asignaturas en las que están matriculados. En concreto, dicha clase deberá proporcionar los siguientes métodos:

```
public void guardarNota(String alumno, String asignatura, double nota)

public double consultarNota(String alumno, String asignatura)
    throws NotaNoEncontrada
```

Nota:

- Si un alumno ya tiene nota en una asignatura, al `guardarNota` se modificará su nota
- La excepción `NotaNoEncontrada` se lanzará cuando no se encuentre la nota del alumno en la asignatura
- Si fuera necesario, se pueden crear clases adicionales para resolver el ejercicio

Ejercicio 16: diseña un método en la clase `TablaHash` que, dado un determinado valor, devuelva una `ListaConPI` con todas las claves que tienen asociado dicho valor

Ejercicio 17: diseña un método en la clase `TablaHash` tal que devuelva una `ListaConPI` con las claves de la primera cubeta de longitud máxima de una Tabla

la primera "Peor " cubeta

¿Y si nos pidieran las claves de la primera "Mejor " cubeta?

Ejercicios propuestos

Ejercicio 18: recordando que en una Tabla Hash Enlazada si dos Entradas están en la misma cubeta es porque han colisionado, diseña un método consultor en la clase `TablaHash` que devuelva el número total de colisiones que se han producido tras insertar un cierto número de Entradas distintas, con claves distintas

Ejercicio 19: supón que se ha modificado la clase genérica `TablaHash` para permitir la inserción de Entradas con la misma Clave. Diseña en esa clase un método `recuperarIguales` que, con coste mínimo, obtenga una Lista con Punto de Interés con los Valores de todas las Entradas de una Tabla Hash cuya Clave sea `c`

Ejercicio 20: se dispone de dos `Map<Clave, Valor>` `m1` y `m2` implementados con sendas Tablas Hash. Diseña un método estático `diferencia` que devuelva el Map diferencia de `m1` y `m2`, es decir un Map que contenga solo aquellas Entradas de `m1` que no estén también en `m2`. Por ejemplo, si las Entradas de `m1` son

{(“uno”, 1) (“dos”, 2) (“tres”, 3) (“cuatro”, 4) (“seis”, 6)}

y las de `m2` son

{(“tres”, 3) (“cuatro”, 4) (“siete”, 7) (“ocho”, 8)},

las Entradas del Map diferencia serán {(“uno”, 1) (“dos”, 2) (“seis”, 6)}

Ten en cuenta que sólo podrás utilizar los métodos definidos en la interfaz `Map` y el método constructor de `TablaHash`