

Tema 3.

Map y Tabla de Dispersión (*Hash*)

Contenidos

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

Tema 3.

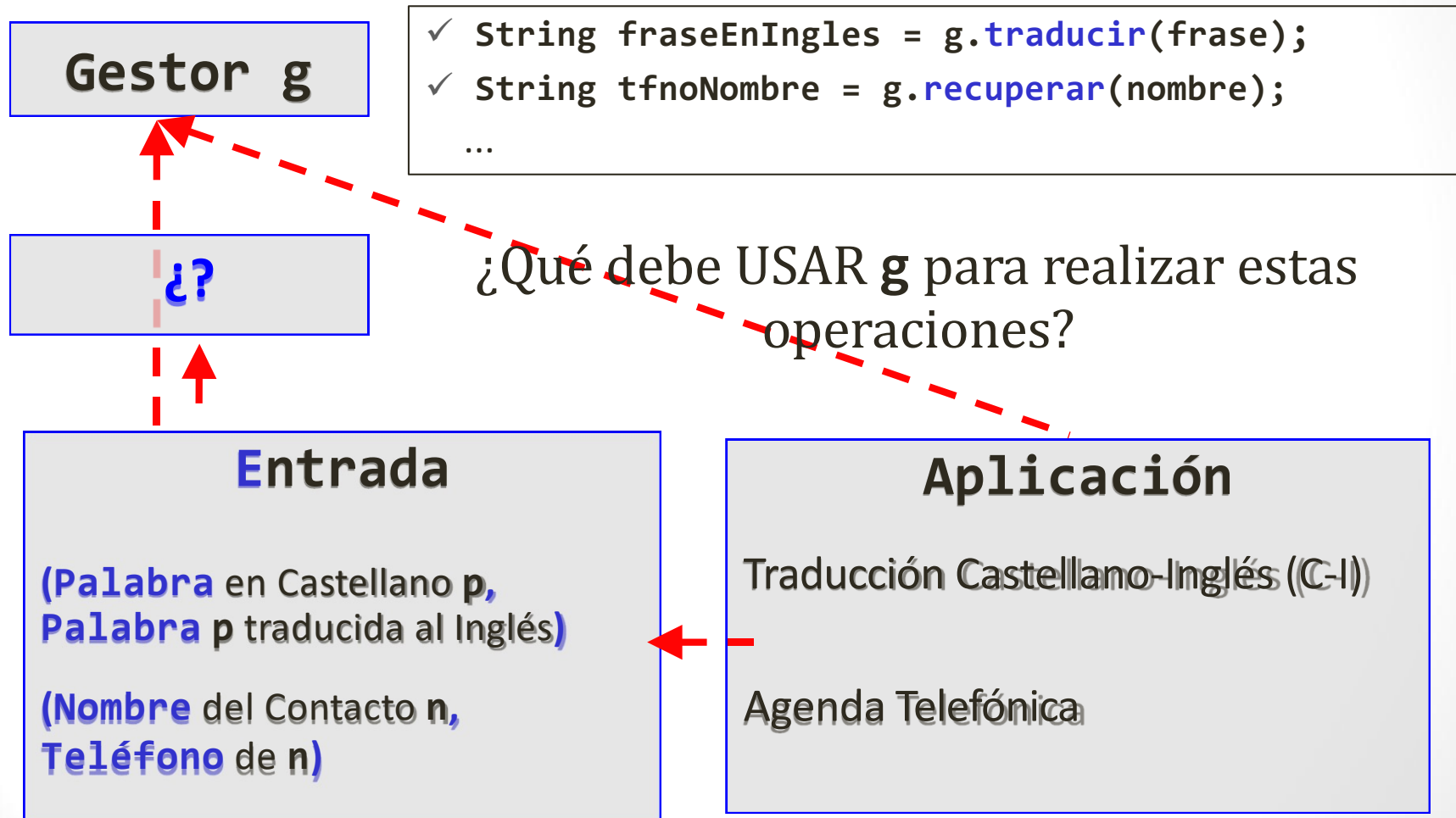
Map y Tabla de Dispersión (*Hash*)

s1

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

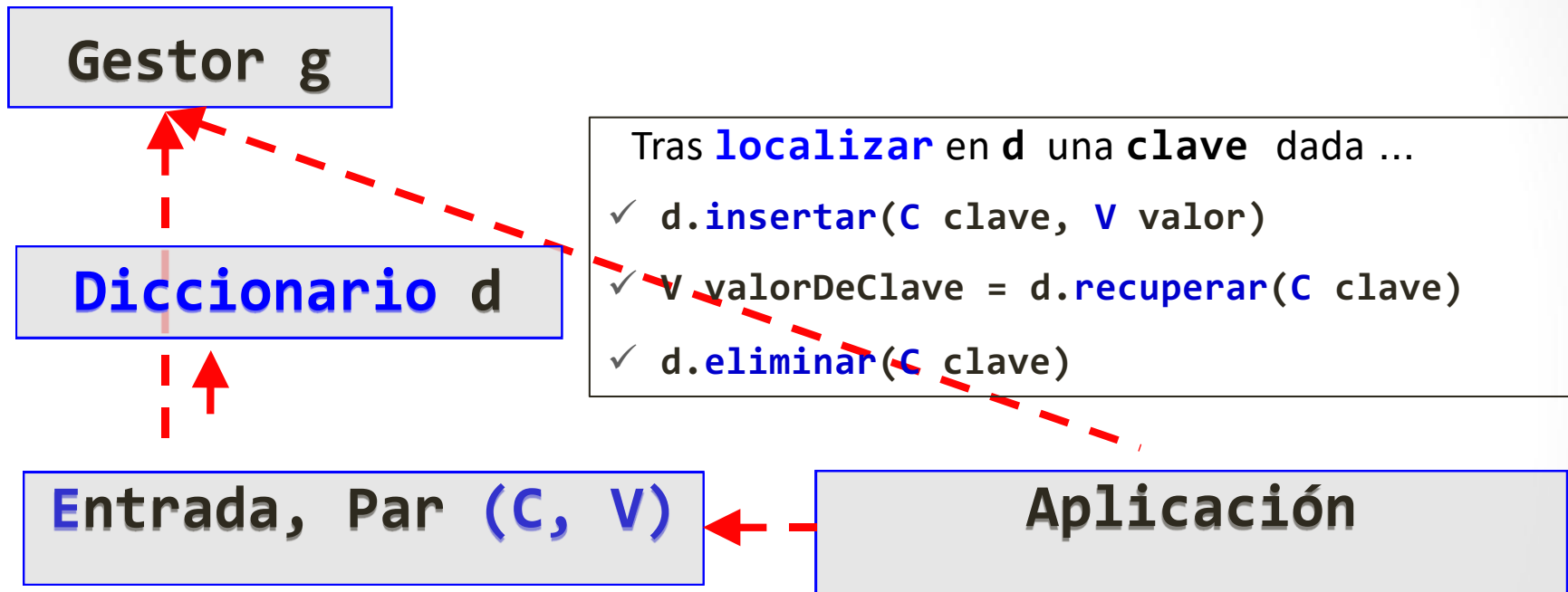
1. El modelo Map

*¿Qué sabemos ya de la gestión de datos según un Diccionario?
En el Tema 1, Ejemplo 2, decíamos ...*



1. El modelo Map

*¿Qué sabemos ya de la gestión de datos según un Diccionario?
En el Tema 1, Ejemplo 2, decíamos ...*



- Modelo de *Búsqueda Dinámica* **por nombre o clave**
(del valor de una Entrada)
- Dos Entradas son **igual**es si tienen la misma clave
- No hay Entradas repetidas

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,
     * o 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, o null si no existe una Entrada con dicha clave */
    V eliminar(C c);

    /** devuelve el valor asociado a la clave c en un Map,
     * o 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

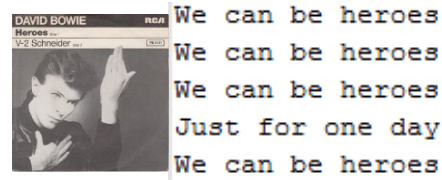
Ejemplos de uso

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 cuál 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: Se quiere mostrar por pantalla las palabras distintas, *tokens* separados por blancos (separadores estándar), que aparecen en una frase leída de teclado (o en un texto); es decir, se desea obtener el vocabulario que contiene tal frase (o texto)



Texto



[can, one, day, for, just, we, heroes, be]

Vocabulario

1. El modelo Map

Ejemplos de uso

1.- ¿Para qué se utiliza un Map en el problema planteado?

Para obtener **solo** las **palabras distintas**, antes hay que extraer todas las palabras de la frase y almacenar solo las que sean distintas en la memoria. Sino, ¿cómo se podría recordar más tarde cuáles son? **Como un Map es una EDA que NO permite insertar claves repetidas, el Map será el almacén de palabras distintas que necesitamos.**

2.- ¿Qué Clave y Valor, así como sus tipos, tiene una Entrada del Map?

Obviamente, las **Claves** de sus Entradas, son las palabras, de tipo String, que luego buscaremos en el Map para mostrarlas por pantalla.

Los **Valores** a ellas asociados NO nos importan en este caso, pues no intervienen en la solución. Así, por ejemplo, pueden ser String vacíos (OJO: vacío NO es null!!, sino "").

3.- ¿Cómo obtener a partir de una frase (String) todas las palabras que lo componen, o cómo establecer qué palabras hay sabiendo que las separan espacios en blanco (separadores)?

Usando el **método split de la clase String** (ver detalles en el programa ejemplo).

4.- ¿Tiene la frase el mismo número de palabras que el Map que se construye a partir de ella? ¿Por qué?

Salvo que no existan repetidas, el número de palabras distintas de la frase, que son las que se almacenan en el Map, es MENOR que el número de palabras de la frase.

En la actualización del proyecto **eda** de teoría disponible en PoliformaT, puedes analizar el código de la clase **Test1Map.java**, para ver los detalles de implementación de la solución del problema usando un Map.

Antes de los ejercicios

Descarga el fichero edaT3.zip de poliformat.

Los ficheros java incluidos en **edaT3.zip** deben copiarse en las carpetas correspondientes a sus paquetes (dentro del proyecto BlueJ **eda**, usado en las clases de teoría):

1.- En `package ejemplos.tema3` (previa creación de este paquete) los siguientes ficheros:

- AnalizadorDeTexto.java
- Matricula.java.java
- ModuloAutorizacion.java
- Par.java
- RadarApp.java
- Test1Map.java
- Test5Map.java
- TestModuloAutorizacion.java
- TestUnion.java
- Usuario.java

2.- En `package librerias.excepciones` el siguiente fichero:

- UsuarioExistente.java

3.- En `package librerias.estructurasDeDatos.modelos` el siguiente fichero:

- ListaConPIPlusMap.java

4.- En `package librerias.estructurasDeDatos.deDispersion` (previa creación de este paquete)

los siguientes ficheros:

- EntradaHash.java
- EntradaHashNodo.java
- TablaHash.java
- TablaHashNodos.java

5.- Por último, en el directorio raíz del proyecto **eda, el siguiente fichero:**

- diccioSpaEng.txt

Ejercicios (ojo no funcionarán porque tenemos implementaciones)

Teniendo siempre presente de antemano qué Mapa es más apropiado definir y qué tipo de Clave y Valor deben tener sus entradas, modifica el programa **Test1Map** (en el paquete ejemplos/tema3 del proyecto eda) como base para crear las clases de los siguientes ejercicios 1.1 y 1.2. Supón que la clase **TablaHash.java** ya está desarrollada (puedes incrustar los datos directamente en el código fuente del programa para probar tus clases).

1.1- Diseñar un nuevo programa **Test2Map** que lea un texto de teclado y muestre por pantalla el número de palabras distintas que contiene.

1.2.- 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 número de veces que se repite en él (su frecuencia de aparición).

Para casa el 1.3 y 1.4

Ejercicio 1.1

```
// ONLY DIFFERENCE WITH RESPECT TO Test1Map:
```

```
// we are asked ONLY for the number of different words
```

```
System.out.println("Numero de palabras distintas en el texto, " + "o talla  
del Map " + m.talla());
```

Ejercicio 1.2

```
public class Test3Map { public static void main(String[] args) {
    Locale localEDA = new Locale("es", "US");
    Scanner teclado = new Scanner(System.in).useLocale(localEDA);
    System.out.println("Escriba palabras separadas por blancos:");
    String texto = teclado.nextLine();
    String[] palabrasDelTexto = texto.split(" ");
    Map m = new TablaHash(palabrasDelTexto.length);
    for (int i = 0; i < palabrasDelTexto.length; i++) {
        String palabra = palabrasDelTexto[i].toLowerCase();
        Integer contador = m.recuperar(palabra);
        if (contador == null) m.insertar(palabra, 1);
        else m.insertar(palabra, contador + 1);
    }
    System.out.println("\nListado de las palabras repetidas en el texto:");
    ListaConPI deClaves = m.claves();
    deClaves.inicio();
    while ( !deClaves.esFin() ) {
        String palabra = deClaves.recuperar();
        Integer contador = m.recuperar(palabra);
        if (contador >= 2) { System.out.println("(" + palabra + ", " + contador + ")"); }
        deClaves.siguiente(); }
    }
```

Ejercicio 1.3

```
public int frecuenciaMayorQue(int n) {  
    // COMPLETAR  
    int res = 0;  
    ListaConPI claves= m.claves();  
    //devuelve una ListaConPI que contiene todas  
    // las claves  
    for (claves.inicio(); !claves.esFin(); claves.siguiente()) {  
        Integer frec = m.recuperar(claves.recuperar());  
        if (frec > n) res++;  
    }  
    return res;  
}
```

Ejercicio 1.4

```
public static String traducir(String textoE, Map d) {  
    //COMPLETAR  
    String palabras[] = textoE.split(" ");  
    String res = "";  
    for (int i = 0; i < palabras.length; i++) {  
        String ingles = d.recuperar(palabras[i]);  
        if (ingles != null) res += ingles;  
        else res += "";  
        if (i < palabras.length - 1) res += " ";  
    }  
    return res; }  
}
```

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?

2. Tabla de Dispersión

Concepto de dispersión: una forma de implementar la Búsqueda Dinámica por Clave en tiempo constante

Supóngase un texto donde las capitales de distintos países del mundo aparecen codificadas mediante un número entero, el lugar que ocupa cada país (según el FMI) en el listado que figura en ...

[https://es.wikipedia.org/wiki/Lista_de_países_por_PIB_\(nominal\)_per_cápita](https://es.wikipedia.org/wiki/Lista_de_países_por_PIB_(nominal)_per_cápita)

Así por ejemplo, en lugar de **Lisboa** como capital de **Portugal** en el texto figura el **38**.

Si se quiere sustituir todos los códigos que aparecen en el texto por los nombres de las capitales y los países a los que corresponden, se puede emplear un traductor que use un Diccionario de 183 entradas.

- Las Entradas del Diccionario de Países son pares de la forma
(**Clave** **c** = Número de País, **Valor** **v** = Capital(Nombre País))
- El Diccionario de Países se representa mediante un
Map<Integer, String>

2. Tabla de Dispersión

Concepto de dispersión: una forma de implementar la Búsqueda Dinámica por Clave en tiempo constante

En **librerías.estructurasDeDatos. ...**

	null	0
1	"Luxemburgo (ídem)"	1
	null	2

15	"Ottawa (Canadá)"	15

26	"Roma (Italia)"	26

38	"Lisboa (Portugal)"	38

183	"Guitega (Burundi)"	183=elArray.length-1

Entrada<C,V>[] **elArray**

En **aplicaciones. ...**

La clase **Traductor** TIENE UN **Map<Integer, String> d** para representar el Diccionario de 183 Países.

Al crear un **Traductor t** se crea vacío el **Map d** y se **insertan** sus 183 Entradas ...

¿Qué sucede al **insertar** una Entrada en **d**? Por ejemplo,

d.insertar(38, "Lisboa (Portugal)");

Idea básica:
indexar **elArray** por la Clave

[16]

2. Tabla de Dispersión

Concepto de dispersión: una forma de implementar la Búsqueda Dinámica por Clave en tiempo constante

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

Problema: La Clave suele ser un **String**

Función de Dispersión

Solución: Función de conversión String - int

- Un **String** **s** es un array de **char** (método **charAt(i)**)
- Un **char** **i** se puede codificar con **n** bits en una cierta base $2^{n\text{bits}}$ como un entero pequeño entre 0 y $2^{n\text{bits}} - 1$

valorIntS = **s.charAt(0)** * **base**^{**s.length()-1**}

Valor de Dispersión,
o valor Hash de s

+ ... +

s.charAt(s.length()-1) * **base**⁰

2. Tabla de Dispersión

Función de Dispersión:

valor hash de una Clave: método `hashCode()`

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

Problema: La Clave suele ser un **String**

Solución: Aplicar una Función de Dispersión "adecuada" a la Clave

¿Cómo se hace eso en Java, para cualquier Clave?

Función de Dispersión estándar: `hashCode()` de `Object`

- La clase de la Clave **debe** sobrescribir `hashCode()` de `Object` y 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()]` (al **insertar**, **eliminar** o **recuperar**)

Volveremos más tarde sobre esta afirmación, para independizarla del tamaño de `elArray`

2. Tabla de Dispersión

*Función de Dispersión: el método **hashCode** de **Object***

<https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#hashCode-->

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

*Ejemplo de sobrescritura de hashCode de Object:
el método hashCode de String*

<https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#hashCode-->

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

The hash code for a String object is computed as

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

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;  
}
```

→ NO usa `Math.pow` ←

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

2. Tabla de Dispersión

Ejemplo de uso del método `hashCode` de `String`

En una aplicación de control del número de veces que se visita una dirección Web se requiere la Búsqueda Dinámica por Clave, i.e. el uso de un `Map<DireccionHTTP, Integer>`. Por tanto, 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(...) {...}  
    ...  
}
```

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

PISTA: Piensa cuándo 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?

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

2. Tabla de Dispersión

Ejemplo de uso del método `hashCode` de `String`

En una aplicación de control del número de veces que se visita una dirección Web se requiere la Búsqueda Dinámica por Clave, i.e. el uso de un `Map<DireccionHTTP, Integer>`. Por tanto, 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(...) {...}  
  
    public boolean equals(Object o) {  
        return o instanceof DireccionHTTP  
            && servidor.equals(((DireccionHTTP) o).servidor)  
            && puerto == ((DireccionHTTP) o).puerto  
            && dirRecurso.equals(((DireccionHTTP) o).dirRecurso);  
    }  
    public int hashCode() {  
        return servidor.hashCode() + puerto + dirRecurso.hashCode();  
    }  
}
```

`/* de String */`

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

PISTA: Piensa cuándo dos direcciones Web son iguales.

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

2. Tabla de Dispersión

Ejemplo de uso del método `hashCode` de `String`

En una aplicación de control del número de veces que se visita una dirección Web se requiere la Búsqueda Dinámica por Clave, i.e. el uso de un `Map<DireccionHTTP, Integer>`. Por tanto, 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(...) {...}  
  
    public boolean equals(Object o) {  
        return o instanceof DireccionHTTP  
            && servidor.equals(((DireccionHTTP) o).servidor)  
            && puerto == ((DireccionHTTP) o).puerto  
            && dirRecurso.equals(((DireccionHTTP) o).dirRecurso);  
    }  
    public int hashCode() {  
        return servidor.hashCode() + puerto + dirRecurso.hashCode();  
    }  
}
```

```
> "www.dsic.com".hashCode()  
208338547 (int)  
> "www.dsic.com".hashCode() + 80  
208338627 (int)  
> "/datos/fichero.txt".hashCode()  
-46620199 (int)  
> "www.dsic.com".hashCode() + 80 + "/datos/fichero.txt".hashCode()  
161718428 (int)
```

`/* de String */`

2. Tabla de Dispersión

Ejemplo de uso del método `hashCode` de `String`

En una aplicación de control del número de veces que se visita una dirección Web se requiere la Búsqueda Dinámica por Clave, i.e. el uso de un `Map<DireccionHTTP, Integer>`. Por tanto, si dicho Map se implementa mediante una Tabla Hash, `DireccionHTTP` es la clase de la Clave de las Entradas a situar sobre el Array que tiene dicha Tabla.

```
public class DireccionHTTP {  
    private String servidor, dirRecurso; private int puerto;  
    public DireccionHTTP(...) {...}  
  
    public boolean equals(Object o) {  
        return o instanceof DireccionHTTP  
            && servidor.equals(((DireccionHTTP) o).servidor)  
            && puerto == ((DireccionHTTP) o).puerto  
            && dirRecurso.equals(((DireccionHTTP) o).dirRecurso);  
    }  
    public int hashCode() {  
        // Mejor aún ...  
        return (servidor + puerto + dirRecurso).hashCode();  
    }  
}
```

/* de String */

```
("www.dsic.com" + 80 + "/datos/fichero.txt").hashCode()  
1481122244 (int)
```

2. Tabla de Dispersión

*Otros ejemplos de sobrescritura de **hashCode** de **Object**: distintos métodos en una clase con clave de tipo **String***

```
// el método "Weiss" (en capítulo 19 apartado 2, figura 19.2 del libro de Weiss)
public int hashCode() {
    int valorHash = 0;
    for (int i = 0; i < this.clave.length(); i++) {
        valorHash = 37 * valorHash + this.clave.charAt(i);
    }
    return valorHash;
}

// el método "Malo" (en capítulo 19 apartado 2, figura 19.3 del libro de Weiss)
public int hashCode() {
    int valorHash = 0;
    for (int i = 0; i < this.clave.length(); i++) {
        valorHash += this.clave.charAt(i);
    }
    return valorHash;
}

// el método "McKenzie" (sustituir 37 por 4 en el método "Weiss")
public int hashCode() { ... }
```

¿Cuál es la diferencia básica entre estos 3 métodos?

¿En qué se diferencian del método `hashCode()` de `String`?

2. Tabla de Dispersión

*Problemas al indexar **elArray** con el valor Hash de una Clave **c***

En **librerias.estructurasDeDatos. ...**

null	0
null	1
null	2
...	...
"a" "un"	97
...	...
null	...
...	...
null	...
...	...
null	46795=elArray.length-1

Entrada<C,V>[] **elArray**

En **aplicaciones. ...**

La clase **Traductor** TIENE UN **Map<String, String>** para representar un Diccionario Bilingüe Inglés-Español

Al crear un **Traductor t** se crea vacío el **Map d** y se insertan sus **46796** Entradas ...

¿Qué sucede al **insertar** una Entrada en **d**? Por ejemplo,

d.insertar("a", "un");

2. Tabla de Dispersión

*Problemas al indexar **elArray** con el valor Hash de una Clave **c***

En **librerias.estructurasDeDatos. ...**

null	0
null	1
null	2
...	...
"a" "un"	97
...	...
null	...
...	...
null	...
...	...
null	46795=elArray.length-1

Entrada<C,V>[] **elArray**

c.hashCode()=99.644
←.....

En **aplicaciones. ...**

La clase **Traductor** TIENE UN **Map<String, String>** para representar un Diccionario Bilingüe Inglés-Español

Al crear un **Traductor t** se crea vacío el **Map d** y se insertan sus **46796** Entradas ...

¿Qué sucede al **insertar** una Entrada en **d**? Por ejemplo,
d.insertar("dog", "perro");

CRASH

2. Tabla de Dispersión

*Función de Dispersión: índice hash de una Clave
(función de compresión)*

Problema: El valor Hash puede ser muy grande y ...

(1) Ni la memoria es infinita

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

c	99	s	115
l	3177	a	3662
i	98592	l	113630
m	3056461	t	3522646
a	94750388	o	109202137
-----		-----	
clima	94750388	salto	109202137

Función de Compresión

Solución: función de conversión valor Hash – posición válida de `elArray`

**Índice de Dispersión,
o índice Hash de c**

2. Tabla de Dispersión

Función de Dispersión: método de la división como función de compresión

Procedimiento para dispersar la Clave c en elArray:

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

Palabra	Valor Hash	Índice Hash (<i>elArray.Length</i> = 101)
clima	94750388	66
salto	109202137	28

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

Thomas Schürger, Why should the size of a hash table be a prime number?
<https://www.quora.com/Why-should-the-size-of-a-hash-table-be-a-prime-number>

2. Tabla de Dispersión

Función de Dispersión: método de la división como función de compresión

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

```
int valorHash = c.hashCode();  
// Método de la división: si  $\text{valorHash} \geq 0$  ...  
int indiceHash = valorHash % elArray.length;  
//  $0 \leq \text{indiceHash} < \text{elArray.length}$  ?
```

¿Y sino?

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: método de la división como función de compresión

Procedimiento para dispersar la Clave c en elArray:

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

Palabra	Valor Hash	Indice Hash (elArray.length=101) con overflow	Indice Hash (elArray.length=101)
clima	94750388	66	66
salto	109202137	28	28
poesía	-982864703	-70	31
novela	-1039634011	-5	96

2. Tabla de Dispersión

Función de Dispersión, finalmente

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

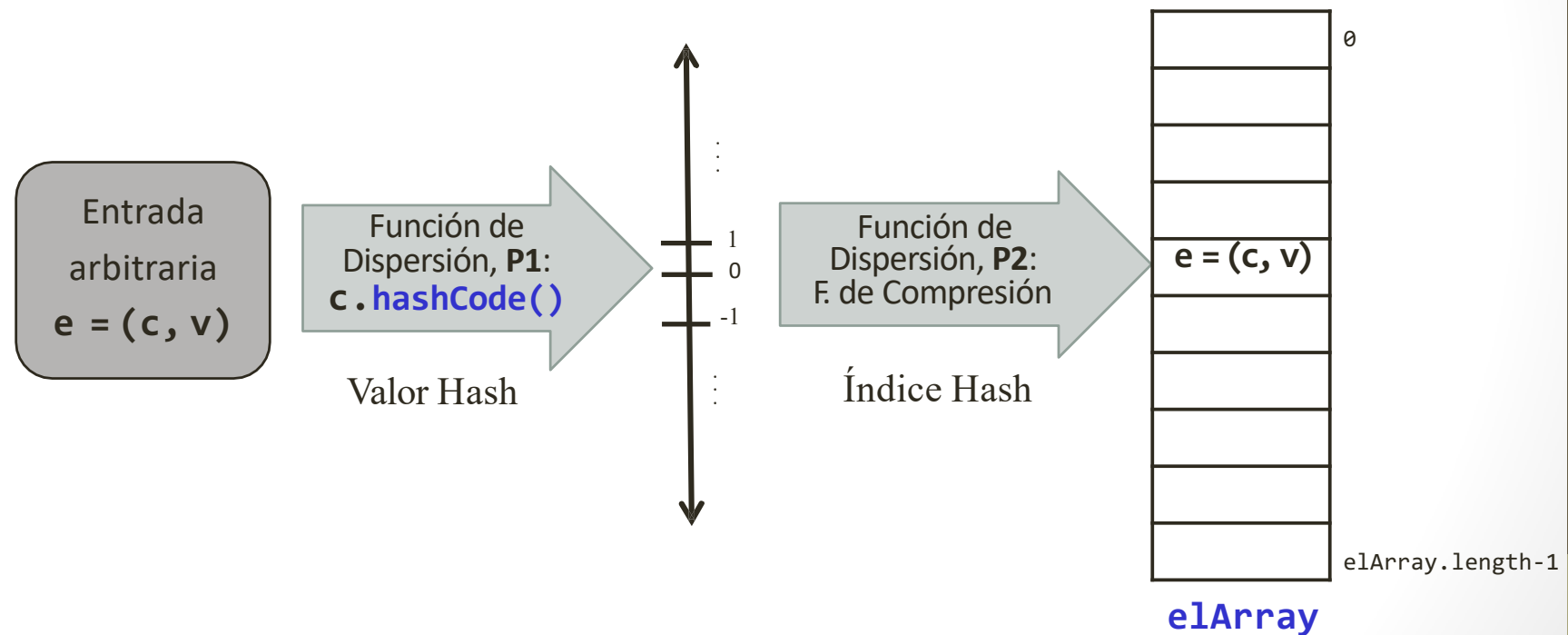
```
public int indiceHash(C c) {  
    int valorHash = c.hashCode();  
    // Método de la división: si  $\text{valorHash} \geq 0$  ...  
    int indiceHash = valorHash % elArray.length;  
    // Overflow de enteros  
    if (indiceHash < 0) { indiceHash += elArray.length; }  
    //  $0 \leq \text{indiceHash} < \text{elArray.length}$   
    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 (coste constante, **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!



Tema 3.

Map y Tabla de Dispersión (*Hash*)

s2

2. Tabla de Dispersión (*Hash*)

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

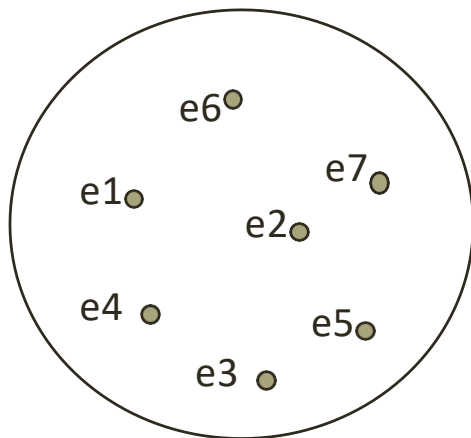
2. Tabla de Dispersión

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

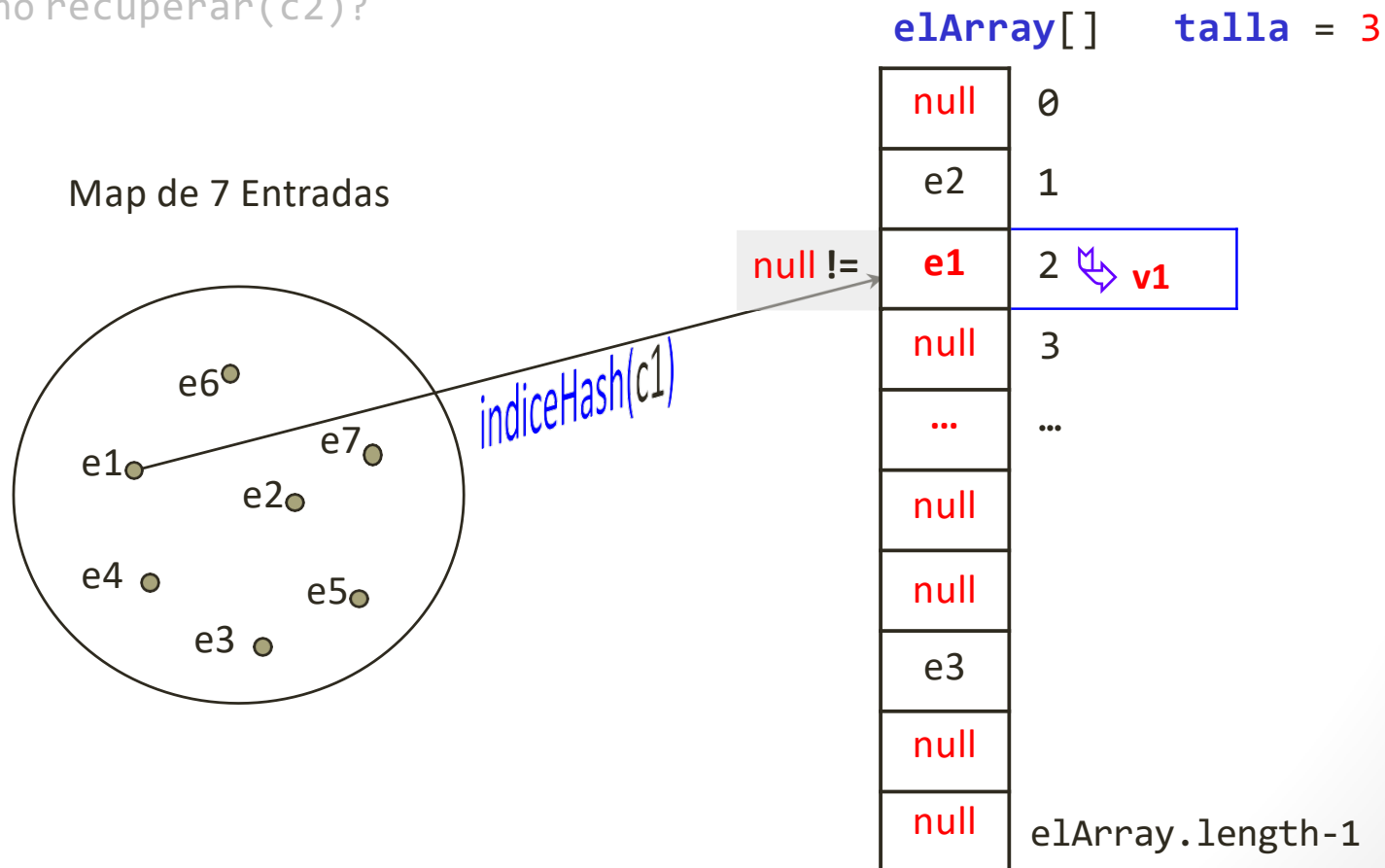
null	0
e2	1
e1	2
null	3
...	...
null	
null	
e3	
null	
null	elArray.length-1

2. Tabla de Dispersión

Colisiones: origen - ¿es siempre inyectiva la función de dispersión?

Ejemplo 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)**?

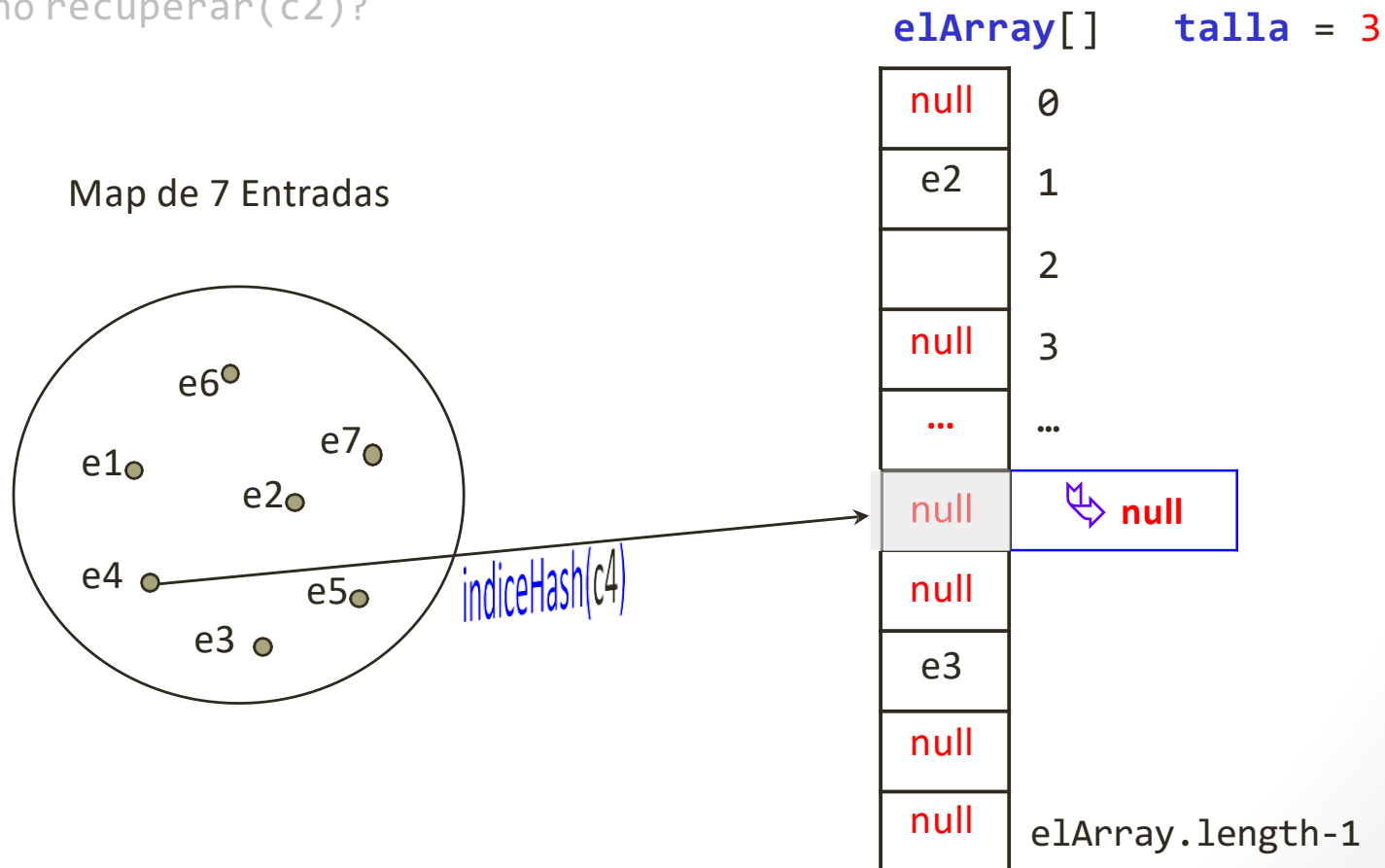


2. Tabla de Dispersión

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)?

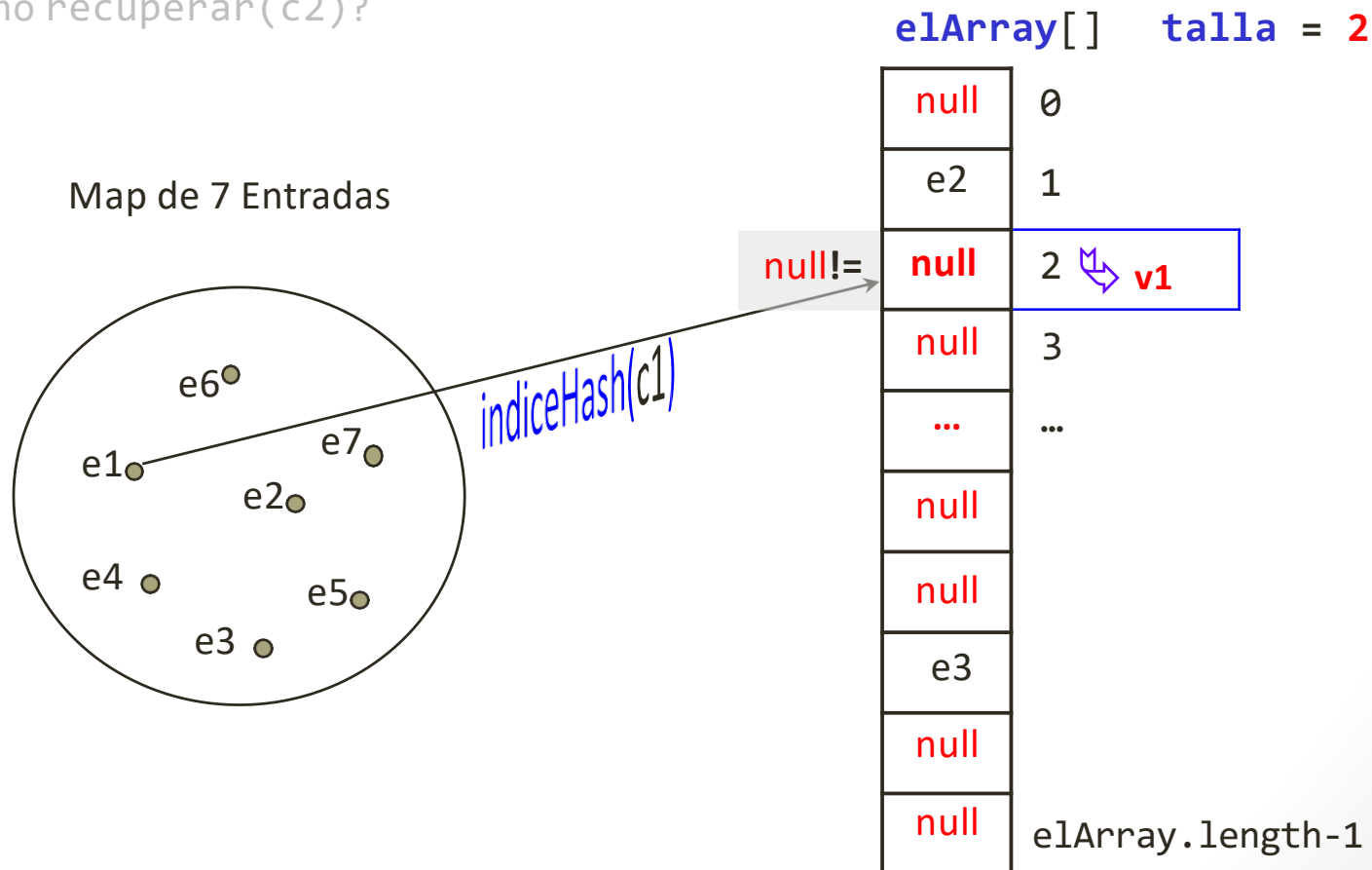


2. Tabla de Dispersión

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)?

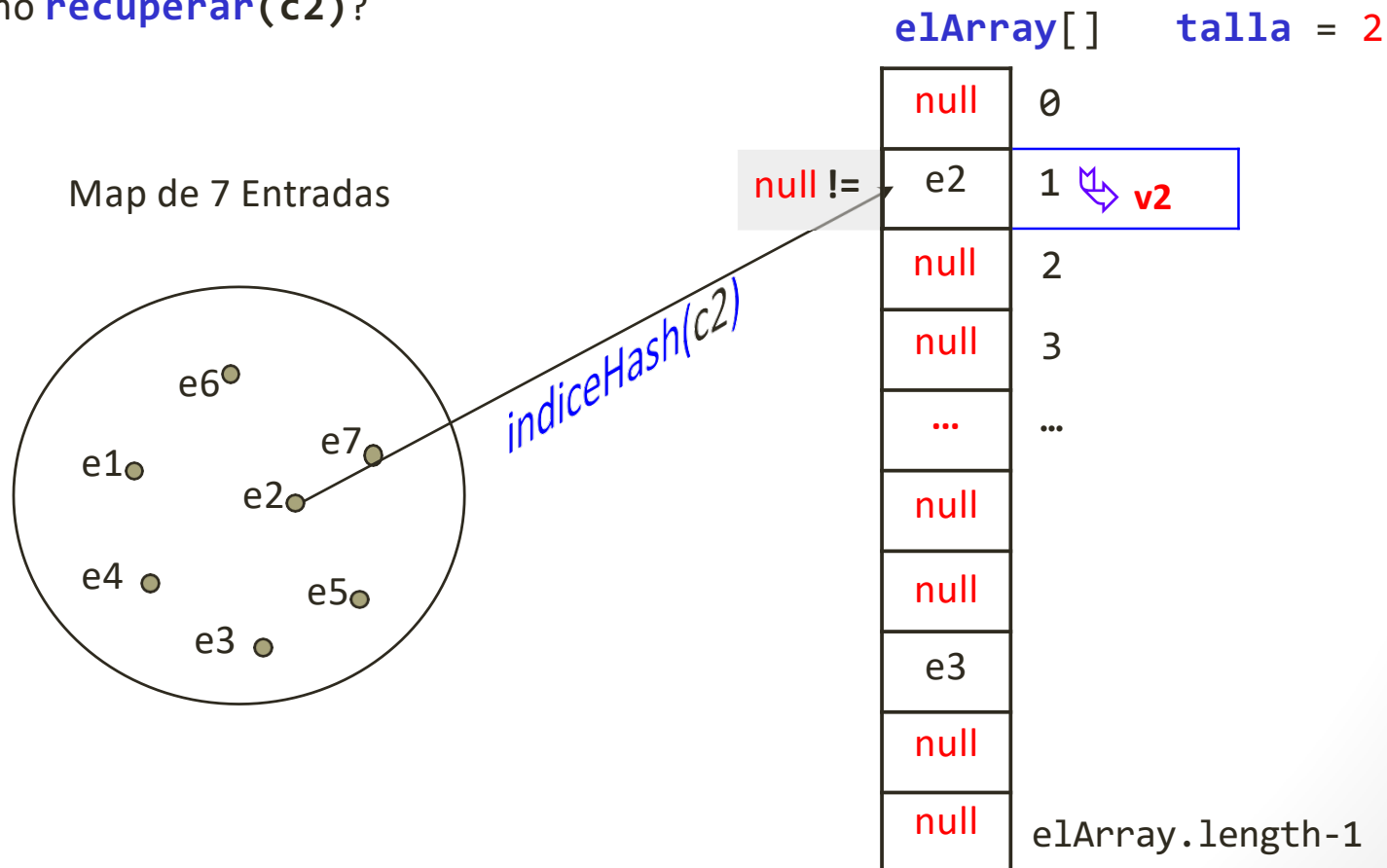


2. Tabla de Dispersión

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)**?



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 de la misma talla y en la que **c.hashCode() = c.intValue()**

- ¿Qué resultado tiene **insertar** en la Tabla **vacía** las siguientes 3 entradas?
(9, 9), (58, 58), (89, 89)
- ¿Qué resultado tendría ejecutar **eliminar(9)**, **recuperar(9)** e **insertar(9, 9)**?
- ¿Qué sucede al **insertar(18, 18)**? ¿Y al **insertar(49, 49)**? ¿Y al **insertar(3, 3)**?
¿Por qué?

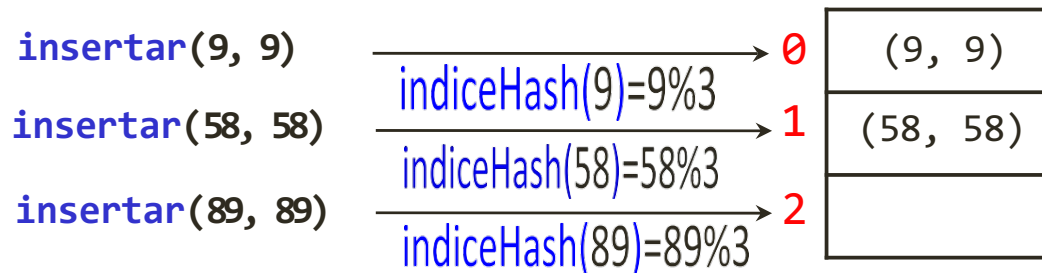
0	null
1	null
2	null

2. Tabla de Dispersión

Colisiones: origen - ¿es siempre inyectiva la función de dispersión?

Ejemplo 2: Se quiere representar un Map de 3 Integer (clave = valor) mediante una Tabla Hash de la misma talla y en la que `c.hashCode() = c.intValue()`

- ¿Qué resultado tiene **insertar** en la Tabla **vacía** las siguientes 3 entradas?
(9, 9), (58, 58), (89, 89)
- ¿Qué resultado tendría ejecutar `eliminar(9)`, `recuperar(9)` e `insertar(9, 9)`?
- ¿Qué sucede al `insertar(18, 18)`? ¿Y al `insertar(49, 49)`? ¿Y al `insertar(3, 3)`?
¿Por qué?



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 de la misma talla y en la que **c.hashCode() = c.intValue()**

- ¿Qué resultado tiene insertar en la Tabla vacía las siguientes 3 entradas?

(9, 9), (58, 58), (89, 89)

- ¿Qué resultado tendría ejecutar **eliminar(9)**, **recuperar(9)** e **insertar(9, 9)**?

eliminar(9)



$\text{indiceHash}(9) = 9 \% 3$

0

1

2

--
(58, 58)
(89, 89)

recuperar(9)



$\text{indiceHash}(9) = 9 \% 3$

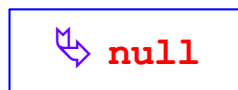
0

1

2

null
(58, 58)
(89, 89)

insertar(9, 9)



$\text{indiceHash}(9) = 9 \% 3$

0

1

2

(58, 58)
(89, 89)

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 de la misma talla y en la que **c.hashCode() = c.intValue()**

- ¿Qué resultado tendría ejecutar **eliminar(9)**, **recuperar(9)** e **insertar(9, 9)**?
- ¿Qué sucede al **insertar(18, 18)**? ¿Y al **insertar(49, 49)**? ¿Y al **insertar(3, 3)**?
¿Por qué?

insertar(18, 18)

$\text{indiceHash}(18) = 18 \% 3$

0

1

2

(9, 9)

(58, 58)

(89, 89)

!=null

CRASH

COLISIÓN de 9 y 18, al tener igual índice hash

insertar(49, 49)

$\text{indiceHash}(49) = 49 \% 3$

0

1

2

(9, 9)

(58, 58)

(89, 89)

!=null

CRASH

COLISIÓN de 58 y 49, al tener igual índice hash

insertar(3, 3)

$\text{indiceHash}(3) = 3 \% 3$

0

1

2

(9, 9)

(58, 58)

(89, 89)

!=null

CRASH

COLISIÓN de 9, 18 y 3, al tener igual índice hash

(43)

2. Tabla de Dispersión

Colisiones: origen - ¿es siempre inyectiva la función de dispersión?

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 número 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?

2. Tabla de Dispersión

Colisiones: origen - ¿es siempre inyectiva la función de dispersión?

Cuestión: Anteriormente se definió el siguiente método **hashCode** “Malo”:

```
public int hashCode() {  
    /* Malo: en Weiss, capítulo 19 apartado 2, figura 19.3 */  
    int valorHash = 0;  
    for (int i = 0; i < this.clave.length(); i++) {  
        valorHash += this.clave.charAt(i);  
    }  
    return valorHash;  
}
```

Si se usa este método en una Tabla Hash donde se insertan 109.580 Entradas con Claves de tipo **String** de longitud máxima 28, sabiendo que $\text{clave.charAt}(i) \in [0..127] \dots$

¿Por qué se producen **siempre** el mismo número de colisiones, independientemente del valor de **elArray.length** que tenga la Tabla?

2. Tabla de Dispersión

Colisiones: origen - ¿es siempre inyectiva la función de dispersión?

Resolución de la Cuestión: Método `hashCode` “Malo”

```
public int hashCode() {  
    /* Malo: en Weiss, capítulo 19 apartado 2, figura 19.3 */  
    int valorHash = 0;  
    for (int i = 0; i < this.clave.length(); i++) {  
        valorHash += this.clave.charAt(i);  
    }  
    return valorHash;  
}
```

Si se usa este método en una Tabla Hash donde se insertan 109.580 Entradas con Claves de tipo **String** de longitud máxima 28, sabiendo que `clave.charAt(i) ∈ [0..127]` ... ¿Por qué se producen **siempre** el mismo número de colisiones, independientemente del valor de `elArray.length` que tenga la Tabla?

`valorHash ∈ [0, 3556]`

- ➔ Solo se pueden indexar las **3.557** primeras posiciones de `elArray`
- ➔ Como mínimo, habrán **109.580 – 3.557 = 106.023 colisiones**

2. Tabla de Dispersión

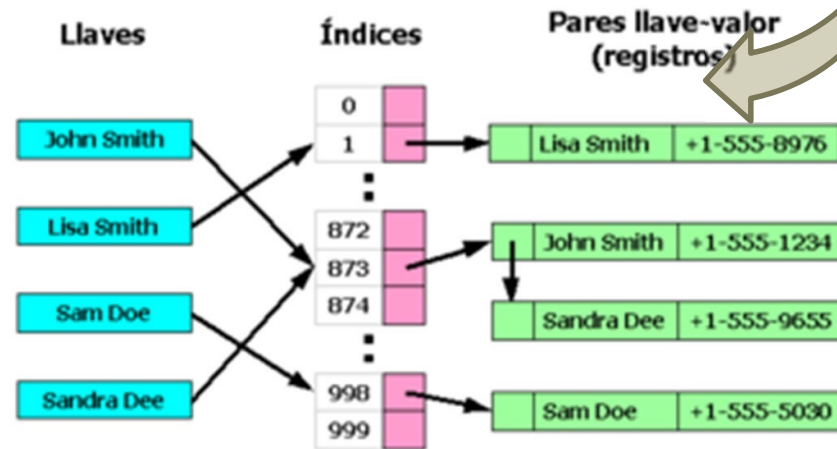
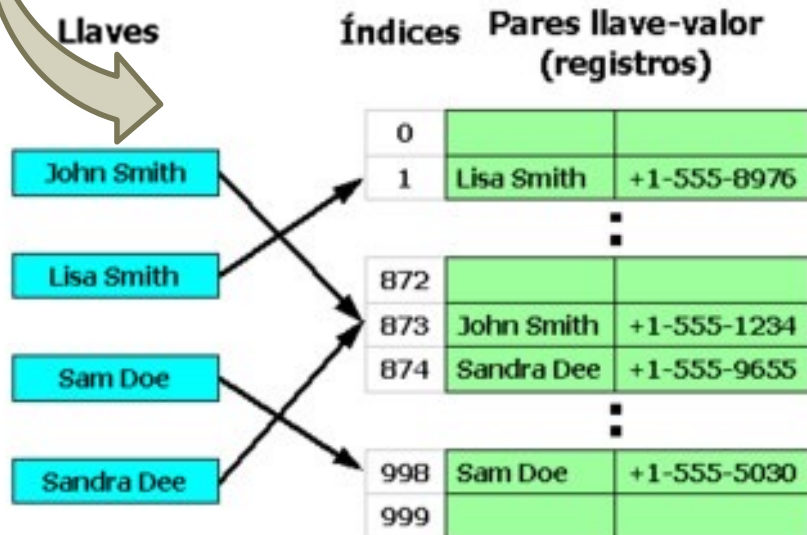
Colisiones: métodos de resolución

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

Métodos para la resolución de colisiones:

- Direcccionamiento abierto

- Encadenamiento separado, o *Hashing Enlazado*



2. Tabla de Dispersión

Colisiones: resolución por Direcccionamiento abierto

Si vamos a insertar una Entrada en una posición y esa posición está ocupada, se busca la primera posición libre alternativa:

- **Exploración lineal:** Resuelve una colisión buscando **secuencialmente** la siguiente posición libre de la Tabla **a partir de** $\text{indiceHash} + 1$
 - Problema de **agrupación primaria**
- **Exploración cuadrática:** Resuelve una colisión buscando la siguiente posición libre de la Tabla, sucesivamente, **en las posiciones** $\text{indiceHash} + 1^2$, $\text{indiceHash} + 2^2$, ..., $\text{indiceHash} + i^2$, implementando circularidad.
 - No hay **agrupación** primaria, pero sí **secundaria**

Direcccionamiento abierto o Hashing Cerrado en:

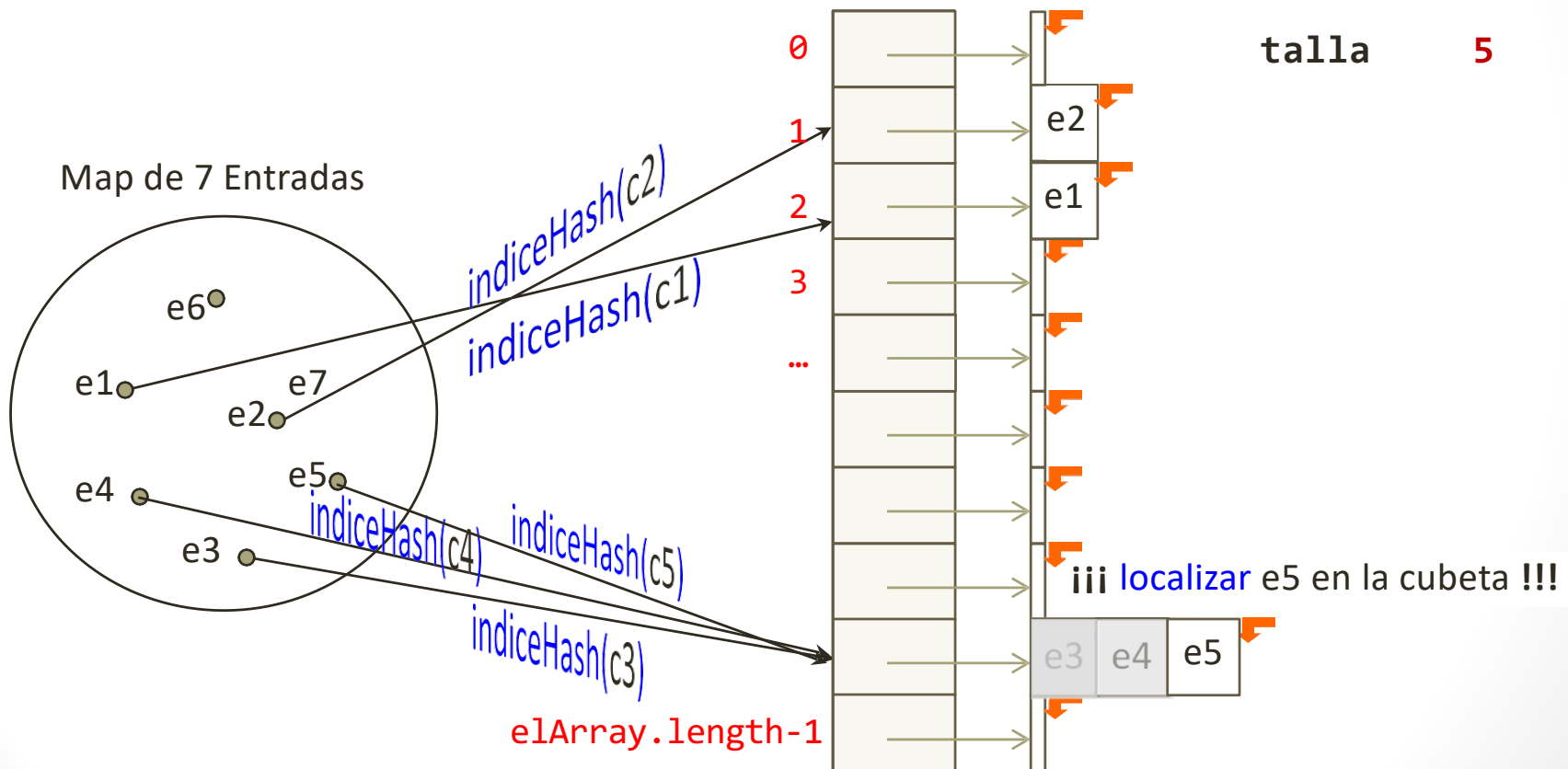
<http://decsai.ugr.es/~jfv/ed1/tedi/cdrom/docs/tablash.html>

2. Tabla de Dispersión

Colisiones: resolución por Hashing Enlazado

Todas las Entradas que colisionan en una misma posición de **e1Array** se almacenan en la **Lista de Colisiones**, o **Cubeta**, asociada a dicha posición:

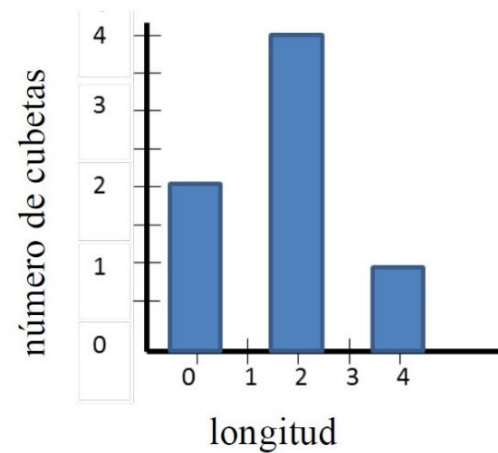
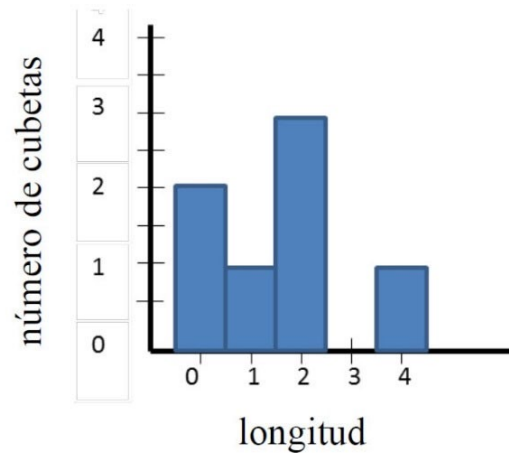
e = (c, v) está en la Lista o cubeta... **e1Array[indiceHash(c)]**



Ejercicio 2.4

Sea el histograma de ocupación de una Tabla Hash, mostrado en la figura de la izquierda.

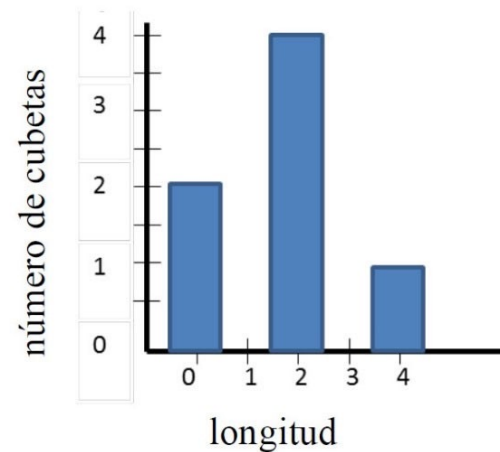
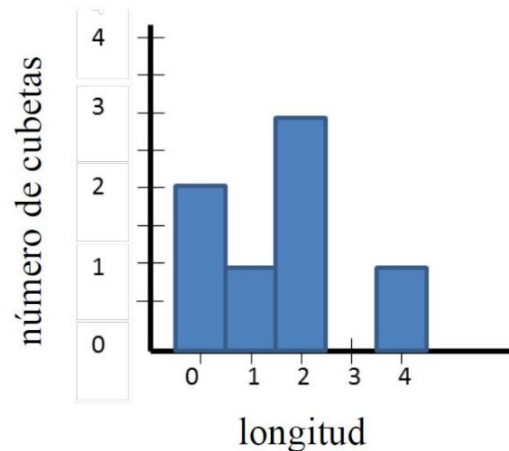
- a) Indica el número de elementos y el de cubetas que tiene la Tabla.
- b) Indica en qué cubeta de la Tabla se debe insertar un nuevo elemento para que su histograma de ocupación pase a ser el mostrado en la figura de la derecha.



Ejercicio 2.4

Sea el histograma de ocupación de una Tabla Hash, mostrado en la figura de la izquierda.

- a) Indica el número de elementos y el de cubetas que tiene la Tabla.
- b) Indica en qué cubeta de la Tabla se debe insertar un nuevo elemento para que su histograma de ocupación pase a ser el mostrado en la figura de la derecha.



- a) **11 elementos y 7 cubetas.**
- b) **Se debe insertar en la (única) cubeta de longitud 1. Así, el número de cubetas de longitud 2 de la Tabla pasará a ser 4, y el de cubetas de longitud 1 pasará a ser cero.**

2. Tabla de Dispersión

Eficiencia

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

Una Tabla Hash de $n = \text{talla Entradas}$ será eficiente siempre que el tiempo promedio ($T_{\text{localizarC}}(n)$) que se tarda en localizar en ella cualquier clave c sea constante. Dado que ...

- Una Entrada de clave c SOLO puede ubicarse en la cubeta $\text{elArray}[\text{indiceHash}(c)]$, $T_{\text{localizarC}}(n)$ será directamente proporcional a la longitud MEDIA de las cubetas de la Tabla.
- La longitud de cualquier cubeta $\text{elArray}[\text{indiceHash}(c)]$ es exactamente el número de colisiones que provoca la clave c , o número de Entradas que colisionan con la de clave c .

SII el número medio de colisiones es muy pequeño
(cada clave provoca 1 o 2 colisiones en promedio),
 $T_{\text{localizarC}}(n)$ **es constante**

¿Existe algún parámetro de la Tabla Hash que represente el número medio de colisiones?

Ejercicio 2.2

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, suponer que no existe en la Tabla ninguna Entrada con clave **c**.

¿Cómo se representan las colisiones en TablaHash?
Listas Con PI

Ejercicio 2.2

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, suponer que no existe en la Tabla ninguna Entrada con clave **c**.

¿Cómo se representan las colisiones en **TablaHash**?

Listas Con PI

```
public int numeroColisiones(C c) {  
    return elArray[indiceHash(c)].talla();  
}
```

2. Tabla de Dispersión

Eficiencia: Factor de carga

Fijado **hashCode(c)** y suponiendo constante su tiempo de ejecución, ...

La eficiencia de una Tabla de Dispersión de **n = talla** Entradas se mide en términos de su **Factor de Carga (FC)**, o su grado de ocupación

- **FC = talla / elArray.length** (= 1 en el caso ideal)
- En una Tabla con *Hashing* Enlazado es la **longitud MEDIA de las cubetas** y, por tanto, depende de...
 - ➔ La calidad de la Función de Dispersión, pues cuanto mejor disperse menor será el número de colisiones.
 - ➔ La capacidad de la Tabla, o `elArray.length`, pues, cuanto mayor sea, menor será el número de colisiones.

¿Cómo elegir `elArray.length` para favorecer la dispersión?

- ① **Debe ser un número primo**
- ② **Debe asegurar un FC < 1**, por ejemplo **0.75** (en promedio, 3/4 de las cubetas tendrán una longitud 1 ó 2 y el resto 0)

2. Tabla de Dispersión

Eficiencia: análisis experimental

Para analizar el comportamiento de una Tabla Hash de $n = \text{talla}$ Entradas y método **hashCode** dado, se pueden realizar dos tipos de experimentos:

- **Experimento 1:** Calcular y representar el número de colisiones que se producen conforme la capacidad de la Tabla (`elArray.length`) aumenta.

Para una buena función de dispersión, se espera que el número de colisiones decrezca conforme la capacidad aumente, o disminuya FC

- **Experimento 2:** Calcular y representar el histograma de ocupación de la Tabla (i.e. cuántas cubetas hay de longitud 0, 1, 2, etc.), junto con la desviación típica de las longitudes de las cubetas (con respecto a la media, o FC).

Para una buena función de dispersión, se espera un gran % de cubetas con longitudes próximas a la media (FC) y una desviación típica baja

Para ilustrarlos, a continuación se presenta un caso concreto de análisis de cada uno de ellos.

2. Tabla de Dispersión

Eficiencia: análisis experimental

Ejemplo de análisis del comportamiento de una Tabla Hash en el Experimento 1.

Calcular y representar el número de colisiones que se producen conforme la capacidad de la Tabla (`elArray.length`) aumenta.

Sea 109.580 la talla de una Colección de Palabras (`String`) representada por una Tabla Hash, y sea 28 la longitud máxima de Palabra. Se ha calculado el número de colisiones que se producen...

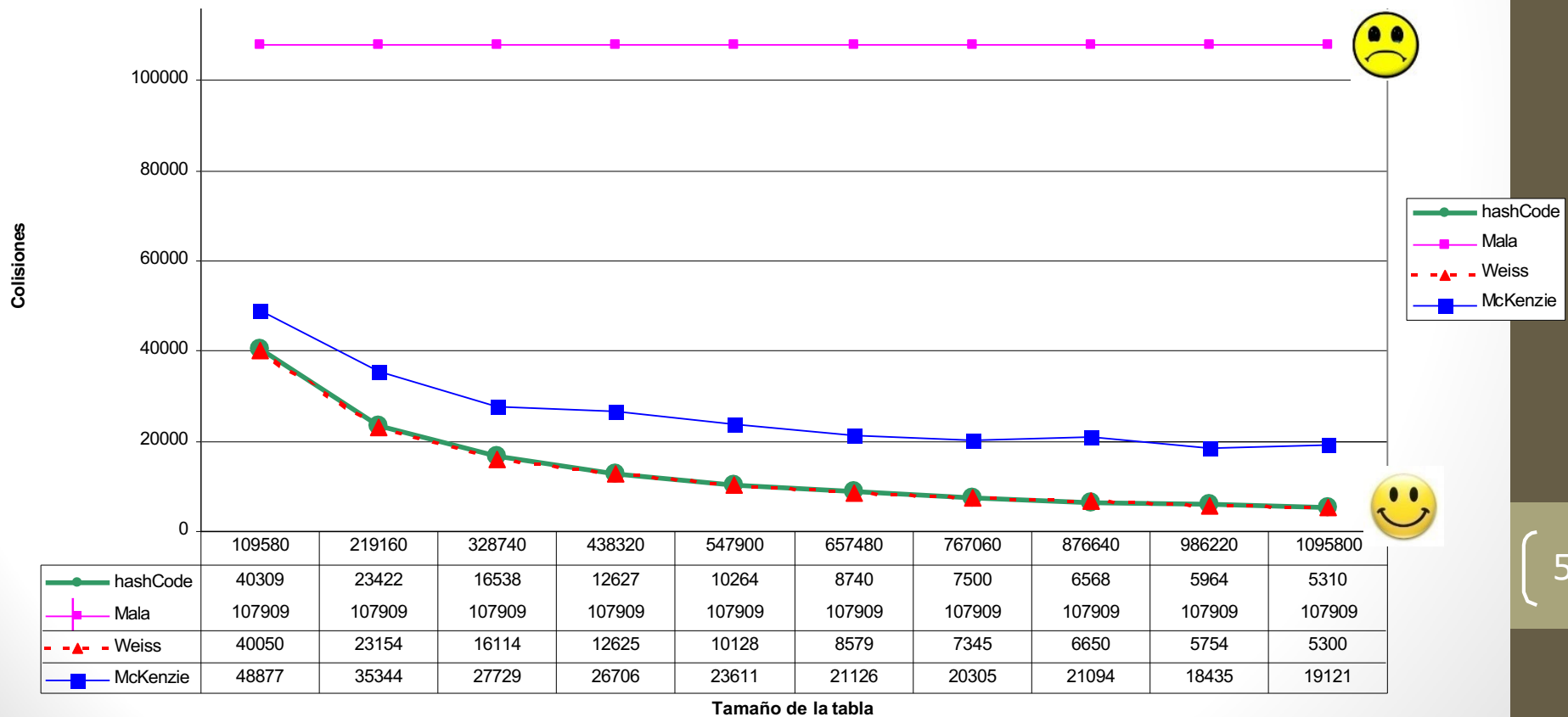
- Al utilizar, para un `elArray.length` dado, cada uno de los 4 métodos `hashCode()` definidos en páginas anteriores (el método de `String` y los métodos de Weiss, “Malo” y McKenzie)
- Al incrementar `elArray.length` en cada paso de la simulación en 109.580 unidades, desde 109.580 hasta 1.095.800

La siguiente gráfica permite analizar los resultados obtenidos, y extraer conclusiones sobre la calidad de las cuatro funciones de dispersión utilizadas, y cuál es la capacidad óptima de la Tabla para cada una de ellas.

2. Tabla de Dispersión

Eficiencia: análisis experimental

- Calidad y Eficiencia se deben combinar en el diseño de la Función de Dispersión, **PERO** debe prevalecer el de Calidad
- **Dado un Factor de Carga**, cuanto mejor es la Función de Dispersión menor número de Colisiones provoca
- El número de Colisiones disminuye conforme el FC disminuye, o aumenta elArray.length



2. Tabla de Dispersión

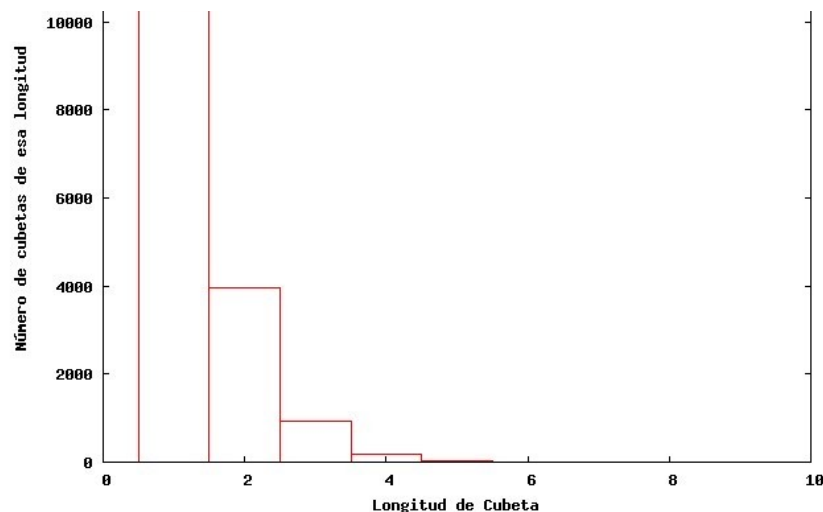
Eficiencia: análisis experimental

Ejemplo de análisis del comportamiento de una Tabla Hash en el Experimento 2.

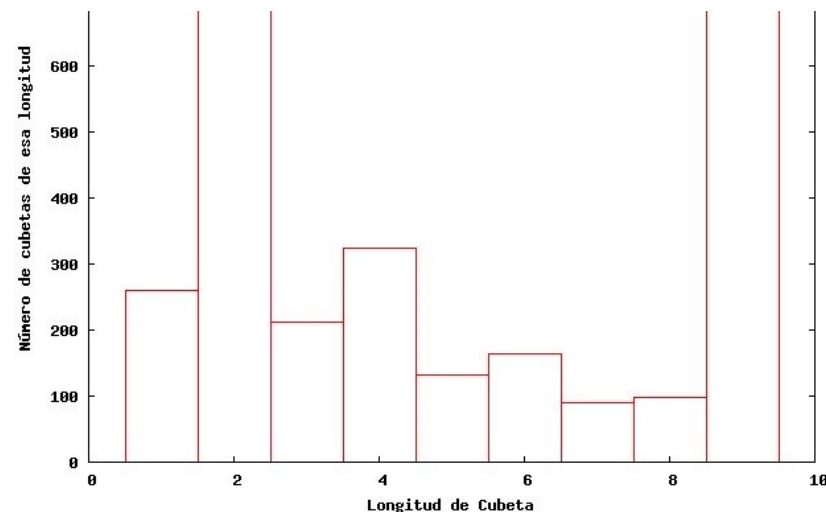
Calcular y mostrar su histograma de ocupación y la desviación típica.

Las siguientes gráficas muestran los histogramas de ocupación de una Tabla Hash con $FC = 0.75$, que contiene 22.000 imágenes de 11×3 píxeles según dos funciones de dispersión....

¿Cuál es la mejor?



Desviación típica < 0.86



Desviación típica > 12

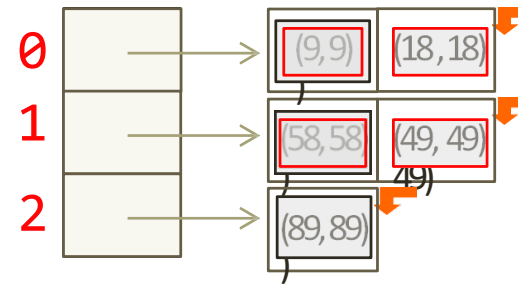


2. Tabla de Dispersión

Eficiencia: Rehashing

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

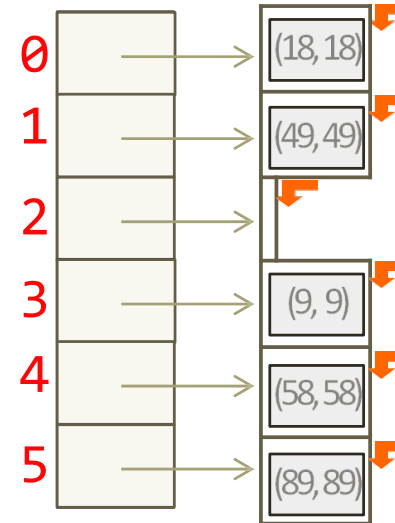
$$FC = 5/3 \rightarrow T_{\text{localizarC}}^{\square}(n) = 7/5 \in O(FC)$$



→ *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

$$FC = 5/6 \rightarrow T_{\text{localizarC}}^{\square}(n) = 5/5 \in O(FC)$$



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

2. Tabla de Dispersión

Eficiencia: Rehashing

El método **rehashing()**, que se implementará en la práctica 3, tiene un coste \propto 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 número de operaciones de *rehashing* y, a la vez, no malgastar memoria es:

1. Estimar la talla máxima de la Tabla Hash Y fijar un $FC < 1$ ($FC = 0.75$)
→ **Determinar una “buena” capacidad inicial de la Tabla**
(`elArray.length`)
2. Hacer *rehashing* SII la talla ACTUAL de la Tabla supera su talla máxima estimada
→ **Hacer *rehashing* SII el factor de carga REAL de la Tabla supera FC**

Sí. Pero si se llega a hacer *rehashing* ...
¿Sigue siendo **$O(1)$** el coste de las operaciones básicas?

2. Tabla de Dispersión

Eficiencia: Rehashing

Sí. 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

Alternativamente ...

Se puede demostrar que el coste total de **M** inserciones, incluidas las del *Rehashing*, es **$O(M)$**

Tema 3.

Map y Tabla de Dispersión (*Hash*)

s3

3. Implementación de una Tabla de Dispersión con *Hashing* Enlazado: las clases **TablaHash** y **EntradaHash**

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> {  
    protected C clave;  
    protected V valor;  
    public EntradaHash(C c, V v) {  
        clave = c; valor = v;  
    }  
    ...  
}  
  
public class TablaHash<C, V> implements Map<C, V> {  
    protected ListaConPI<EntradaHash<C,V>>[] elArray;  
    protected int talla;  
    protected int indiceHash(C c) {...}  
    ...  
}
```

3. Implementación

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

```
package librerias.estructurasDeDatos.deDispersion;  
public class TablaHash<C, V> implements Map<C, V> {  
    protected ListaConPI<EntradaHash<C,V>>[] elArray;  
    protected int talla;  
    public TablaHash(int tallaMaximaEstimada) {  
        int capacidad = siguientePrimo((int)  
            (tallaMaximaEstimada / FACTOR_DE_CARGA));  
        elArray = new LEGListaConPI[capacidad];  
        for (int i = 0; i < elArray.length; i++)  
            elArray[i] = new  
            LEGListaConPI<EntradaHash<C,V>>();  
        talla = 0;  
    }  
}
```

Ejercicio 2.1

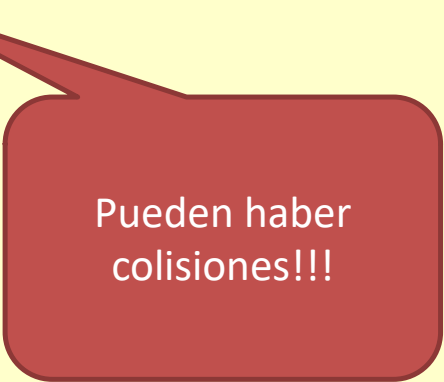
Completar el código de la clase **TablaHash<C, V>**.

```
public class TablaHash<C, V> implements Map<C, V> {  
  
    public V recuperar(C c) {  
        ...  
    }  
  
    public V eliminar(C c) {  
        ....  
    }  
  
    public V insertar(C c, V v) {  
        ....  
    }  
  
    public ListaConPI<C> claves() {  
        ....  
    }  
  
    public final String toString{  
    }  
  
    public final double factorDeCarga(){  
  
    }  
}
```

Ejercicio 5 (parte I).

Completar el código de la clase **TablaHash<C, V>**.

```
public V recuperar(C c) {  
    int pos = indiceHash(c);  
    ListaConPI<EntradaHash<C,V>> cubeta = elArray[pos];  
    V valor = null;  
    if(!cubeta.esFin()) valor=cubeta.recuperar().valor;  
    return valor;  
}
```



Pueden haber
colisiones!!!

Ejercicio 5 (parte I).

Completar el código de la clase **TablaHash<C, V>**.

```
public V recuperar(C c) {  
    int pos = indiceHash(c);  
    ListaConPI<EntradaHash<C,V>> cubeta = elArray[pos];  
    for (cubeta.inicio(); !cubeta.esFin() && !cubeta.recuperar().clave.equals(c); cubeta.siguiente());  
    V valor = null;  
    if(!cubeta.esFin()) valor=cubeta.recuperar().valor;  
    return valor;  
}
```

Ejercicio 5 (parte I).

Completar el código de la clase **TablaHash<C, V>**.

```
private ListaConPI<EntradaHash<C, V>> localizar(C c) {
    int pos = indiceHash(c);
    ListaConPI<EntradaHash<C, V>> cubeta = elArray[pos];
    for (cubeta.inicio(); !cubeta.esFin() && !cubeta.recuperar().clave.equals(c);
cubeta.siguiente());
    return cubeta;
}

public V recuperar(C c) {
    V valor = null;
    // Búsqueda en cubeta de la Entrada de clave c cuyo valor se quiere recuperar
    ListaConPI<EntradaHash<C, V>> cubeta = localizar(c);
    // Resolución de la Búsqueda: Si esta la Entrada se recupera su valor
    if (!cubeta.esFin()) { valor = cubeta.recuperar().valor; }
    return valor;
}

public ListaConPI<C> claves() {
    ListaConPI<C> l = new LEGListaConPI<C>();
    for (ListaConPI<EntradaHash<C, V>> cubeta : elArray)
        for (cubeta.inicio(); !cubeta.esFin(); cubeta.siguiente())
            l.insertar(cubeta.recuperar().clave);
    return l; }
```

Ejercicio 5 (parte I).

Completar el código de la clase **TablaHash<C, V>**.

```
public V eliminar(C c) {
    int pos = indiceHash(c);
    ListaConPI<EntradaHash<C,V>> cubeta = localizar(c);
    V valor = null;
    if(!cubeta.esFin()) {
        valor=cubeta.recuperar().valor;
        cubeta.eliminar();
        talla--;
    }
    return valor;
}

public V insertar(C c, V v) {
    V antiguoValor = null;
    ListaConPI<EntradaHash<C, V>> cubeta = localizar(c);
    if (cubeta.esFin()) { // si no esta, insercion efectiva de la Entrada (c, v)
        cubeta.insertar(new EntradaHash<C, V>(c, v));
        talla++;
    } else {
        antiguoValor = cubeta.recuperar().valor;
        cubeta.recuperar().valor = v;
    }
    return antiguoValor;
}
```

Ejercicio 5 (parte I).

Completar el código de la clase **TablaHash<C, V>**.

```
public final double factorCarga() {  
    return (double) talla / elArray.length;  
}  
  
public final String toString() {  
    StringBuilder res = new StringBuilder();  
    for (ListaConPI<EntradaHash<C, V>> cubeta : elArray)  
        for (cubeta.inicio(); !cubeta.esFin(); cubeta.siguiente())  
            res.append(cubeta.recuperar() + "\n");  
    return res.toString();  
}
```


Ejercicio 4.1. ejemplos/tema3/Matricula

Se dispone de una aplicación de radares de tráfico que permite saber las veces que un coche de matrícula dada ha pasado por cierto radar superando el límite de velocidad. Para ello, la aplicación consulta un diccionario representado mediante un **Map<Matricula, Integer>**

(a) La clase Matricula

Sabiendo que una matrícula es igual a otra si los números y letras que tiene una coinciden con los de la otra, añade a la clase los métodos necesarios para que pueda ser la clase de las claves del Map que usa la aplicación de radares de tráfico.

(b) Registrar matrícula.

- a) Sabiendo que contabilizar las veces que un coche ha excedido el límite de velocidad al pasar por un radar determinado requiere actualizar *convenientemente* el diccionario de la aplicación, diseña, en la clase **RadarApp** (en **ejemplos/tema3**) el método que se encarga de hacerlo, con perfil:

```
public void registrar(Matricula mat)
```

¿Qué operación de un Map permite registrar una matrícula? Piensa que la matrícula podría ya estar registrada, i.e. el radar habría detectado otra vez al mismo coche.

Ejercicio 4.1. ejemplos/tema3/Matricula

Se dispone de una aplicación de radares de tráfico que permite saber las veces que un coche de matrícula dada ha pasado por cierto radar superando el límite de velocidad. Para ello, la aplicación consulta un diccionario representado mediante un **Map<Matricula, Integer>**

(a) La clase Matricula

Sabiendo que una matrícula es igual a otra si los números y letras que tiene coinciden con los de la otra, añade a la clase los métodos necesarios para que pueda ser la clase de las claves del Map que usa la aplicación de radares de tráfico.

```
public class Matricula {
    private int numeros;
    private String letras;
    public Matricula(int n, String l) { numeros = n; letras = l; }
    public int getNumeros() { return numeros; }
    public String getLetras() { return letras; }
    public String toString() { return "Matricula "+numeros+" "+letras; }

    public boolean equals(Object o) {
        return o instanceof Matricula
            && letras.equals(((Matricula) o).letras)
            && numeros == ((Matricula) o).numeros;
    }
    public int hashCode() {
        return (letras + numeros).hashCode();
    }
}
```

Ejercicio 4.1. ejemplos/tema3/RadarApp

Se dispone de una aplicación de radares de tráfico que permite saber las veces que un coche de matrícula dada ha pasado por cierto radar superando el límite de velocidad. Para ello, la aplicación consulta un diccionario representado mediante un **Map<Matricula, Integer>**

(b) Registrar matrícula

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

```
public void registrar(Matricula mat)
```

```
public class RadarApp {  
    private Map<Matricula, Integer> map;  
    public RadarApp() {  
        map = new TablaHash<Matricula, Integer>(1000);  
    }  
  
    public void registrar(Matricula mat) {  
        Integer frec = map.recuperar(mat);  
        if (frec != null) {  
            map.insertar(mat, frec + 1);  
        } else {  
            map.insertar(mat, 1);  
        }  
    }  
}
```

Ejercicio 3.1. ejemplos/tema3/TestUnion

Dadas dos Listas Con PI, **l1** y **l2**, el siguiente método obtiene un String en el que aparecen cada uno de los elementos de 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> a = new LEGListaConPI<Par>();
    for (l1.inicio(); !l1.esFin(); l1.siguiente()) {
        E e = l1.recuperar();
        for (a.inicio(); !a.esFin() && !e.equals(a.recuperar().dato); a.siguiente());
        if (a.esFin()) a.insertar(new Par(e, 1));
        else a.recuperar().frec++;
    }
    for (l2.inicio(); !l2.esFin(); l2.siguiente()) {
        E e = l2.recuperar();
        for (a.inicio(); !a.esFin() && !e.equals(a.recuperar().dato); a.siguiente());
        if (a.esFin()) a.insertar(new Par(e, 1));
        else a.recuperar().frec++;
    }
    return a.toString();
}
```

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

Rediseña el método **union** para que su coste sea lineal con la suma de las tallas de **l1** y **l2**

Ejercicio 3.1. ejemplos/tema3/TestUnion

Queremos rediseñar **union** porque es ineficiente, pero ...

1. **¿Cuánto?** Calcula su coste
2. **¿Dónde?** Indica aquellas “zonas” del código más costosas
3. **¿Quién?** Indica la EDA a usar para hacer lo mismo que en esas zonas pero con menor coste

```
public static <E> String union(ListaConPI<E> l1, ListaConPI<E> l2)
```

```
{  ListaConPI<Par> a = new LEGListaConPI<Par>();  
  for (l1.inicio(); !l1.esFin(); l1.siguiente()) {
```

```
    E e = l1.recuperar();
```

```
    for (a.inicio(); !a.esFin() && !e.equals(a.recuperar().dato); a.siguiente());
```

```
    if (a.esFin()) a.insertar(new Par(e, 1));
```

```
    else a.recuperar().frec++;
```

```
}
```

```
for (l2.inicio(); !l2.esFin(); l2.siguiente()) {
```

```
  E e = l2.recuperar();
```

```
  for (a.inicio(); !a.esFin() && !e.equals(a.recuperar().dato); a.siguiente());
```

```
  if (a.esFin()) a.insertar(new Par(e, 1));
```

```
  else a.recuperar().frec++;
```

```
}
```

```
return a.toString();
```

```
}
```

En vez de una ListaConPI<Par>, un **Map<E, Integer>** de talla `l1.talla()+ l2.talla()` implementado con una **TablaHash**

Sustituir por una búsqueda más eficiente

$$T_{\text{union}}(n = \max(l1.talla(), l2.talla())) \in \Omega(n)$$

$$T_{\text{union}}(n = \max(l1.talla(), l2.talla())) \in O(n^2)$$

Ejercicio 3.1 ejemplos/tema3/TestUnion

Queremos rediseñar **union** porque es ineficiente, pero ...

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

```
public static <E> String union(ListaConPI<E> l1, ListaConPI<E> l2)
{
    Map<E,Integer> m = new TablaHash<E,Integer>(l1.talla()+l2.talla());
    for (l1.inicio(); !l1.esFin(); l1.siguiente()) {
        E e = l1.recuperar();
        Integer frec = m.recuperar(e);
        if (frec == null) m.insertar(e, 1);
        else m.insertar(e, ++frec);
    }
    for (l2.inicio(); !l2.esFin(); l2.siguiente()) {
        E e = l2.recuperar();
        Integer frec = m.recuperar(e);
        if (frec == null) m.insertar(e, 1);
        else m.insertar(e, ++frec);
    }
    return m.toString();
}
```

¿Solo?

¿Solo?

Tema 3.

Map y Tabla de Dispersión (*Hash*)

s4

Ejercicios

Ejercicio 3.2.

Usando un Map, diseña un método **modaDe** que obtenga la moda de un array genérico **v** (i.e. que devuelva el primer elemento de **v** que se repite más veces), con perfil:

```
public static <E> E modaDe(E[] v)
```


Ejercicio 3.2.

Usando un Map, diseña un método **modaDe** que obtenga la moda de un array genérico **v** (i.e. que devuelva el primer elemento de **v** que se repite más veces), con perfil:

```
public static <E> E modaDe(E[] v)
```

```
public static <E> E modaDe(E[] v) {  
    E moda = null;  
    int frecModa = 0;  
    Map<E, Integer> d = new TablaHash<E, Integer>(v.length);  
    for (int i = 0; i < v.length; i++) {  
        Integer frec = d.recuperar(v[i]);  
        if (frec != null) { frec++; }  
        else { frec = 1; }  
        d.insertar(v[i], frec);  
        if (frec > frecModa) {  
            frecModa = frec;  
            moda = v[i];  
        }  
    }  
    return moda;  
}
```

Ejercicio 3.3.

Se dispone de dos **Map<Clave,Valor>**, **m1** y **m2**, implementados con 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**.

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 del Map diferencia serán: {"uno", 1} {"dos", 2} {"seis", 6}

Tener en cuenta que solo se podrán utilizar los métodos definidos en las interfaces **Map** y **ListaConPI** y el método constructor de **TablaHash**.

Ejercicio 3.3.

Se dispone de dos **Map<Clave,Valor>**, **m1** y **m2**, implementados con 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**.

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 del Map diferencia serán: {"uno", 1} {"dos", 2} {"seis", 6}

Tener en cuenta que solo se podrán utilizar los métodos definidos en las interfaces **Map** y **ListaConPI** y el método constructor de **TablaHash**.

```
public static <C,V> Map<C,V> diferencia(Map<C,V> m1, Map<C,V> m2) {  
    Map<C,V> res = new TablaHash<C,V>(m1.talla());  
    ListaConPI<C> lpi = m1.claves();  
    for (lpi.inicio(); !lpi.esFin(); lpi.siguiente()) {  
        C clave = lpi.recuperar();  
        if (m2.recuperar(clave) == null) {  
            res.insertar(clave, m1.recuperar(clave));  
        }  
    }  
    return res;  
}
```

Ejercicio 3.4. subinterfaz ListaConPIPlusMap

En el paquete librerias.estructurasDeDatos.modelos, se ha definido **ListaConPIPlusMap**, subinterfaz de **ListaConPI**.

```
public interface ListaConPIPlusMap<E> extends ListaConPI<E> {  
    /** elimina los elementos repetidos de una ListaConPI,  
    * dejando únicamente su primera aparición */  
    void eliminarRepetidos();  
    /** elimina los elementos de una ListaConPI  
    * que están en otra */  
    void diferencia(ListaConPI<E> otra);  
}
```

Diseña una clase **LEGListaConPIPlusMap** que la implemente. La implementación de sus métodos se debe realizar usando, únicamente, los métodos de **ListaConPI** y **Map**.

La cabecera de la clase será:

```
public class LEGListaConPIPlusMap<E>  
    extends LEGListaConPI<E>  
    implements ListaConPIPlusMap<E> { ... }
```

Ejercicio 3.4. subinterfaz ListaConPIPlusMap

La implementación del método `eliminarRepetidos`, en la clase `LEGListaConPIPlusMap`:

```
public void eliminarRepetidos() {
    Map<E, E> m = new TablaHash<E, E>(this.talla());
    this.inicio();
    while (!this.esFin()) {
        E e = this.recuperar();
        E f = m.recuperar(e);
        if (f == null) {
            m.insertar(e, e);
            this.siguiente();
        }
        else {
            this.eliminar();
        }
    }
}
```

Ejercicio 3.4. subinterfaz ListaConPIPlusMap

La implementación del método **diferencia**, en la clase **LEGListaConPIPlusMap**:

```
public void diferencia(ListaConPI<E> otra) {
    Map<E, E> m = new TablaHash<E, E>(otra.talla());
    for (otra.inicio(); !otra.esFin(); otra.siguiente()) {
        E e = otra.recuperar();
        m.insertar(e, e);
    }
    this.inicio();
    while (!this.esFin()) {
        E e = this.recuperar();
        E f = m.recuperar(e);
        if (f == null) this.siguiente();
        else          this.eliminar();
    }
}
```

Ejercicio 4.2

Sea una aplicación de gestión de notas en la que se usa un **Map<Alumno, Double>**, cada una de cuyas Entradas representa a un alumno y la nota que este ha obtenido en una asignatura.

a) Implementa un método cuyo perfil sea:

```
public static Map<Alumno, Double> obtenerAprobados(Map<Alumno, Double> m)
```

donde el parámetro **m** es el Map de las notas de todos los alumnos de una asignatura. El método debe realizar las siguientes acciones:

- ☐ Devolver un (nuevo) Map que contenga únicamente las Entradas de **m** que corresponden a alumnos aprobados (con nota mayor o igual que 5.0).
- ☐ Eliminar del Map **m** todas las Entradas que corresponden a alumnos aprobados. Es decir, al terminar la ejecución del método, **m** debe contener únicamente las Entradas correspondientes a alumnos suspendidos.

b) Suponiendo que el Map **m** se ha implementado eficientemente mediante una **TablaHash**, indica para el método diseñado: la talla del problema **n** que resuelve, en función de sus parámetros; las instancias significativas que presenta, si las hubiera; su coste Temporal en notación asintótica (O y Ω o bien Θ).

Ejercicio 4.2

a) Implementa el método ...

```
public static Map<Alumno, Double> obtenerAprobados(Map<Alumno,
Double> m) {
    Map<Alumno, Double> aprobados
        = new TablaHash<Alumno, Double>(m.talla());

    ListaConPI<Alumno> l = m.claves();

    for (l.inicio(); !l.esFin(); l.siguiente()) {
        Alumno alumno = l.recuperar();
        Double nota = m.recuperar(alumno);
        if (nota >= 5.0) {
            aprobados.insertar(alumno, nota);
            m.eliminar(alumno);
        }
    }

    return aprobados;
}
```


Ejercicio 4.2

b) Analiza el coste del método ...

- Talla del problema, en función de los parámetros del método:

`n = m.talla()`

- Instancias significativas:

No hay, pues se trata de un método de Recorrido.

- Coste, utilizando la notación asintótica:

$T_{\text{obtenerAprobados}}(n) \in \Theta(n)$

Ejercicio 4.3

Diseñar un método estático que, dados un array **v** de **Integer** y un **Integer e**, devuelva un **String** en el que se indique, con el formato que se muestra en el siguiente ejemplo, los pares de elementos de **v** que sumen **e**.

Por ejemplo, si **v = {1, 4, 6, 3, 8, 9, 5, 2}** y **e = 10**, el resultado del método es el **String** siguiente:

$$v[1] + v[2] = 4 + 6 = 10$$

$$v[0] + v[5] = 1 + 9 = 10$$

$$v[4] + v[7] = 8 + 2 = 10$$

Para diseñar este método se deben tener en cuenta las siguientes consideraciones:

- ☐ El array no debe contener elementos repetidos. En cuanto se detecte un repetido debe devolver como resultado el String: **"Error: hay elementos repetidos"**.
- ☐ Si $v[i] + v[j]$ está en el resultado, no hay que incluir también $v[j] + v[i]$ (por la propiedad conmutativa).
- ☐ El método debe tener el menor coste temporal posible, evitando recorrer el array **v** más de una vez. Considerad el uso de un **Map** como estructura auxiliar.

Ejercicio 4.3

```
public static String metodoArrays(Integer[] v,
Integer e) {
    String res = "";
    // Map: clave = element v, valor = position in v
    Map<Integer,Integer> map
        = new TablaHash<Integer,Integer>(v.length);

    for (int j = 0; j < v.length; j++) {
        Integer i = map.recuperar(e - v[j]);
        if (i != null) {
            res += "v["+i+"] + v["+j+"] = "
                +v[i]+" + "+v[j]+" = " +e+"\n";
        }
        Integer jRepetido = map.insertar(v[j], j);
        if (jRepetido != null) {
            return "Error: hay elementos repetidos";
        }
    }
    return res;
}
```