Tema 3

MAPS y Tablas de Dispersión (Hash)

Objetivos

- Presentar el modelo ***Map***, que se define para resolver problemas de búsqueda dinámica eficientemente
- Estudiar la ***Tabla Hash*** como una representación eficiente del modelo *Map*, prestando especial atención a los siguientes aspectos:
  - Los conceptos relacionados con su definición: función de dispersión (*hashing*), conflictos y su resolución
  - El análisis de su eficiencia, medida como su factor de carga
  - Implementación de la clase *TablaHash* con encadenamiento separado

Contenidos

1. El modelo ***Map***
2. Tablas de dispersión
   2.1. Concepto de dispersión
   2.2. Funciones de dispersión
   2.3. Funciones de compresión
   2.4. Colisiones
   2.5. Factor de carga y *rehashing*
   2.6. Histograma de ocupación
3. Implementación de una Tabla de dispersión con encadenamiento separado

1. El modelo *Map*

Introducción

- El modelo *Map* está diseñado para favorecer la búsqueda de un dato en la colección (no se permite, en general, datos repetidos)
- Los datos que se almacenan en un *Map* son pares clave-valor, donde:
  - La búsqueda se realiza en función de la clave: el método *equals* deberá permitir comprobar si dos claves son iguales o no
  - El valor es la información asociada a la clave que se desea recuperar
- La operación básica de un *Map* es la búsqueda por clave (o nombre) en una colección de entradas

1. El modelo *Map*

Métodos

- La funcionalidad del modelo *Map* podemos observarla mediante la siguiente interfaz Java:

```
public interface Map<C, V> {
  // Añade la entrada (c,v) y devuelve el antiguo valor que
  // tenía dicha clave (o null si no tenía ningún valor asociado)
  V insertar(C c, V v);

  // Elimina la entrada con clave c y devuelve su valor asociado
  // (o null si no hay ninguna entrada con dicha clave)
  V eliminar(C c);

  // Busca la clave c y devuelve su informacion asociada
  // o null si no hay una entrada con dicha clave
  V recuperar(C c);

  // Devuelve true si el Map está vacío
  boolean esVacio();

  // Devuelve el número de entradas que contiene el Map
  int talla();

  // Devuelve una lista con las claves de todas las entradas del Map
  ListaConPI<C> claves();
}
```

1. El modelo *Map*

Uso del modelo (I)

- Existen múltiples aplicaciones que manejan *Maps* y una de ellas es la traducción de textos. Un ejemplo sencillo es el diseño de un traductor palabra a palabra de castellano a inglés.

- Ejercicio: implementa el siguiente método:

```
public static String traducir(String textoCastellano,
   Map<String,String> map)
```

, teniendo en cuenta que la clave en el *map* es la palabra en castellano y el valor es su traducción al inglés. El método traducir devuelve una cadena con la traducción al inglés, palabra a palabra, de la cadena *textoCastellano*. Si una palabra no se encuentra en el *map* deberá sustituirla por “<error>” en la cadena resultante.

1. El modelo *Map*

Uso del modelo (II)

- Otra aplicación muy habitual es la del cálculo de la frecuencia de aparición de los elementos de una colección.

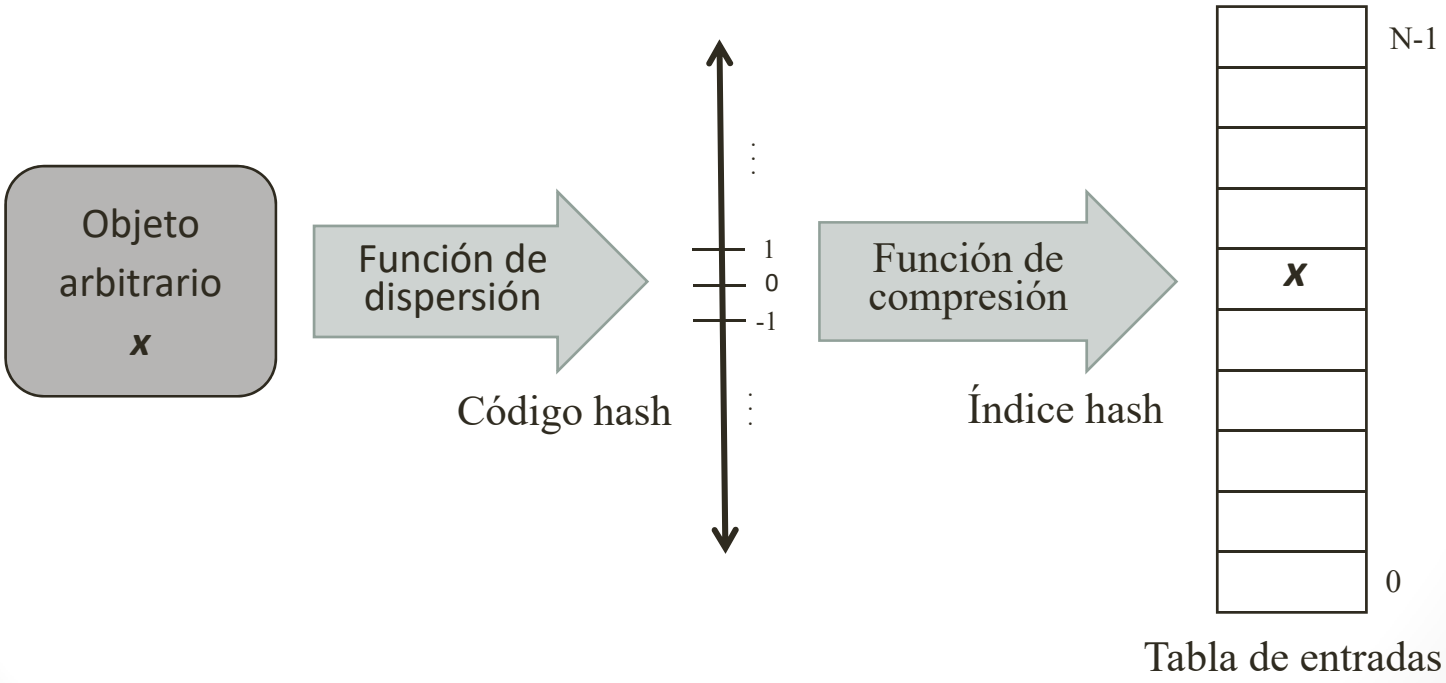
- Ejercicio: usando la interfaz *Map,* y disponiendo de la clase *TablaHash* que la implementa, diseña un programa que lea un texto por teclado y devuelva el número de palabras distintas que hay en dicho texto junto con la frecuencia de aparición de cada una de ellas.

  Nota: la frecuencia se calcula como el número de veces que aparece la palabra en el texto dividido entre el número total de palabras del texto.

2. Tablas de dispersión

2.1. Concepto de dispersión

- Estructura de datos especialmente diseñada para la implementación de *Maps* (operaciones *recuperar*, *insertar* y *eliminar* en un tiempo esperado O(1))



2. Tablas de dispersión

2.2. Funciones de dispersión – Método simple

- Definición: función que convierte una entrada en un entero (*código hash*) adecuado para indexar la tabla en la que dicha entrada se ha de almacenar
- Método simple: suma de componentes. Ejemplo:
  - *U*na entrada es una palabra en castellano (clave) junto con su traducción al inglés (valor)
  - Queremos almacenar la colección de entradas en un array
  - Para saber en qué posición del array guardar cada entrada podemos sumar los códigos ASCII de los caracteres de su clave:

clave de la entrada		**código *hash***
casa	→	99 + 97 + 115 + 97 = 408
hola	→	104 + 111 + 108 + 97 = 420

2. Tablas de dispersión

2.2. Funciones de dispersión polinomiales

- La suma de componentes no es una buena función de dispersión ya que es fácil que dos entradas distintas tengan el mismo código *hash* (**colisión**):

hola $\longrightarrow$ $104 + 111 + 108 + 97 = 420$

teja $\longrightarrow$ $116 + 101 + 106 + 97 = 420$ **colisión**

- Funciones polinomiales: para mejorar la calidad de la función de dispersión se puede ponderar la posición de cada carácter dentro de la clave:

$f(c) = c_0 \cdot a^{k-1} + c_1 \cdot a^{k-2} + \ldots + c_{k-2} \cdot a^1 + c_{k-1}$, con $a>1$.

Ejemplo con a=2

hola $\longrightarrow$ $104 \cdot 2^3 + 111 \cdot 2^2 + 108 \cdot 2 + 97 = 1589$

teja $\longrightarrow$ $116 \cdot 2^3 + 101 \cdot 2^2 + 106 \cdot 2 + 97 = 1641$

2. Tablas de dispersión

2.2. El método hashCode de Java

```
public int hashCode();     // definido en la clase Object
```

- Cualquier clase que vaya a ser utilizada como clave en un *Map* debe reescribir adecuadamente este método.
- La clase *String* implementa una función de dispersión polinomial con base 31:

$$codigoHash = \sum_{i=0}^{length-1} charAt(i) \bullet base^{length-1-i}$$

2. Tablas de dispersión

2.3. Funciones de compresión

- El código *hash* puede ser un valor mayor que el tamaño del *array*. Puede ser también un número negativo.
- Función de compresión: convierte un código *hash* en un **índice hash** entre 0 y la capacidad del *array* menos uno.
- Método de la división:

indiceHash = codigoHash % capacidadDelArray

if (indiceHash < 0) indiceHash += capacidadDelArray;

Para el caso de que el código hash sea negativo

2. Tablas de dispersión

2.4. Colisiones

- La función de dispersión devuelve siempre el mismo valor para una misma entrada (o para dos entradas que son iguales de acuerdo con el método *equals*)
- Si dos entradas son diferentes, es conveniente que la función de dispersión devuelva dos valores diferentes. Aunque esto no es estrictamente necesario, esta característica mejora la eficiencia de las tablas hash
- Aún con una buena función de dispersión, las colisiones son posibles $\Rightarrow$ métodos para la resolución de colisiones:
  - Direccionamiento abierto
  - Encadenamiento separado

2. Tablas de dispersión

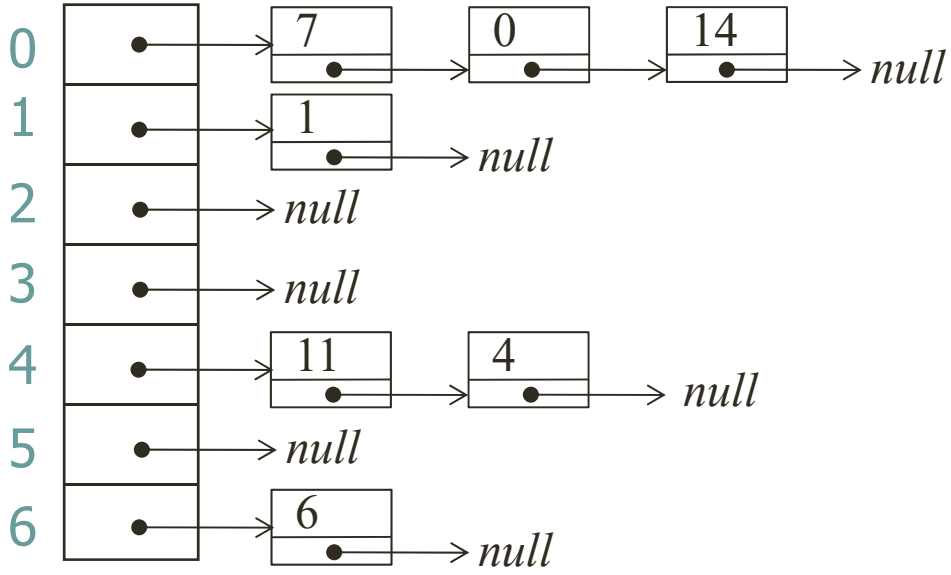
2.4. Colisiones – Direccionamiento abierto

- Si vamos a insertar un elemento en una posición y esa posición está ocupada se busca una posición alternativa
- La exploración lineal resuelve una colisión buscando secuencialmente a partir de *indiceHash* la siguiente posición libre de la tabla
  - Problema de agrupación primaria
- La exploración cuadrática la resuelve consultando sucesivamente las posiciones $indiceHash+1^2$, $indiceHash+2^2$, ..., $indiceHash+i^2$, implementando circularidad
  - No hay agrupación primaria, pero sí secundaria

2. Tablas de dispersión

2.4. Colisiones – Encadenamiento separado

- Todas las entradas que colisionan en una misma posición se almacenan en una lista enlazada
  - A cada una de estas listas se las denomina **cubeta**



2. Tablas de dispersión

2.5. Factor de carga

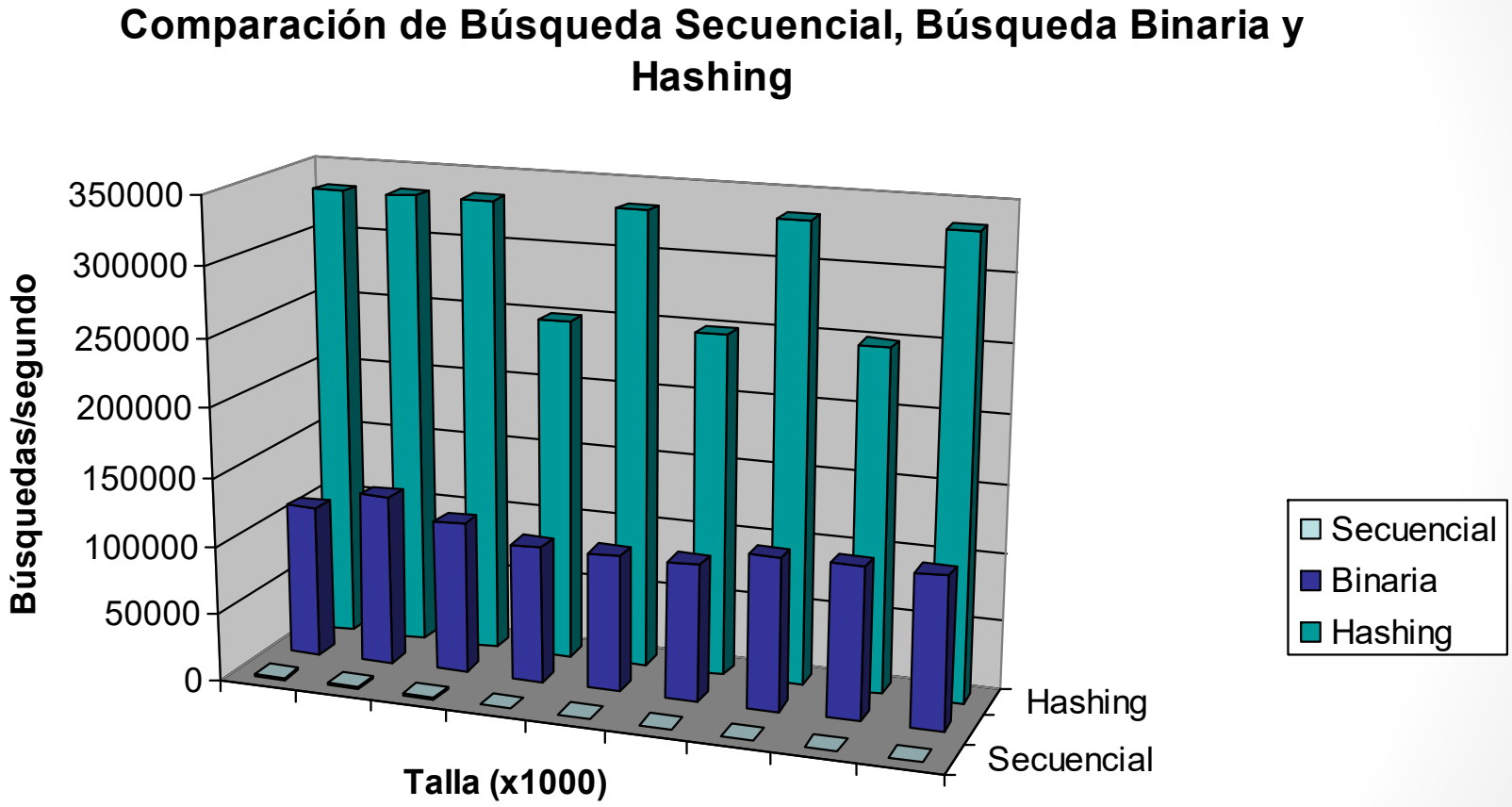
- El rendimiento de una tabla de dispersión se mide en términos de su factor de carga, que se define como la longitud media de las cubetas:
  - FC = tallaActualTabla / capacidadDelArray
- La eficiencia de una tabla de dispersión depende, por tanto, de:
  - La calidad de su función de dispersión:

    Mejor dispersión → menos colisiones
  - Su grado de ocupación:

    Tabla más llena → más colisiones
  - Su método de resolución de colisiones

2. Tablas de dispersión

2.5. Comparativa de eficiencia



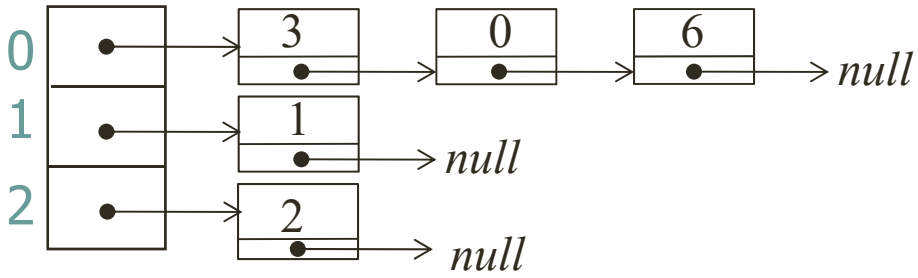
Secuencial	1471	2000	1351	1053	826	680	578	503	441
Binaria	111110	125002	111110	100000	100000	100000	111110	111110	111110
Hashing	333331	333357	333331	250003	333331	250003	333331	250003	333331

2. Tablas de dispersión

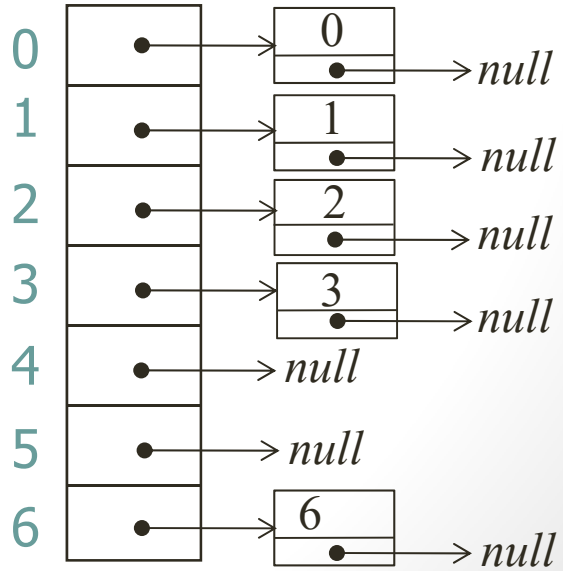
2.5. Rehashing

- El número de colisiones puede crecer excesivamente si el Factor de Carga (o grado de ocupación) es demasiado alto
- El ***rehashing*** consiste en incrementar el tamaño de la tabla de dispersión, reduciendo así su grado de ocupación

FC = 5/3 = 1.67



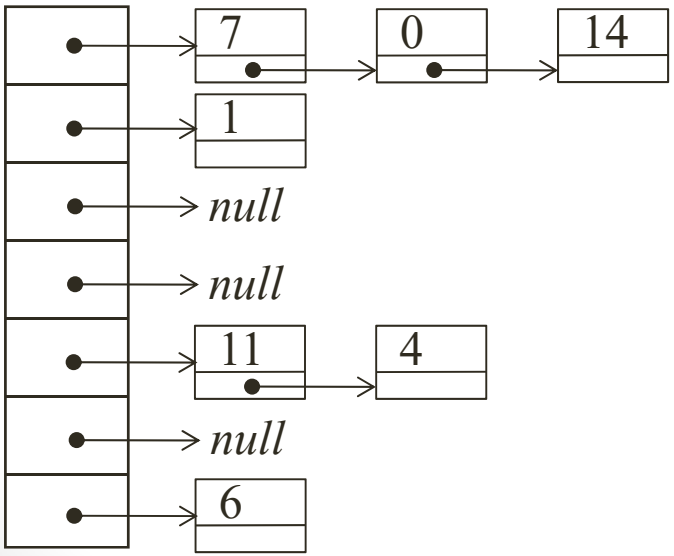
FC = 5/7 = 0.71

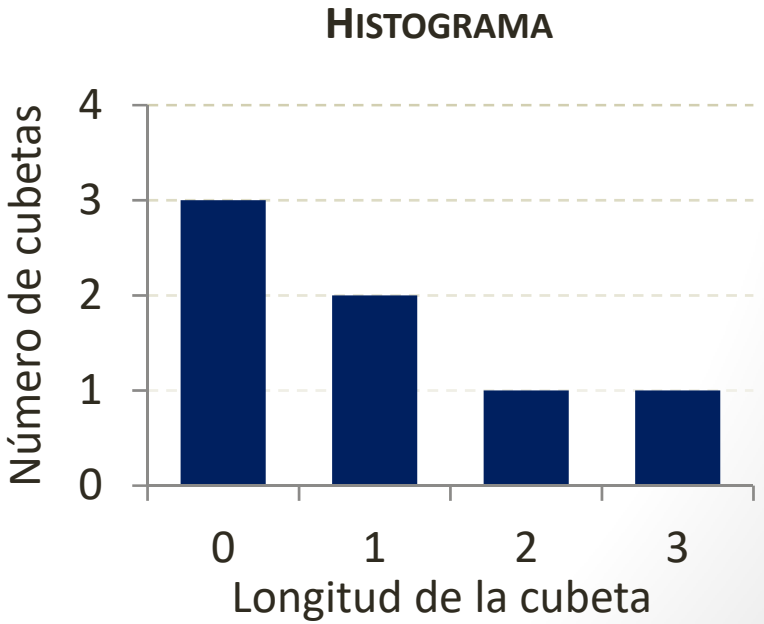


2. Tablas de dispersión

2.6. Histograma de ocupación

- Existe una manera gráfica de ver lo bien que dispersa una *TablaHash*: el histograma de ocupación
  - La altura de cada columna indica cuántas cubetas hay con una longitud dada
  - Ejemplo:





3. Implementación

Entradas para la Tabla de dispersión

- Es necesario definir una clase genérica que almacene conjuntamente la *clave* y el *valor* de una entrada:

```
class EntradaHash<C, V> {
  C clave;                          // Clave de la entrada
  V valor;                          // Valor de la entrada

  public EntradaHash(C clave, V valor) {
    this.clave = clave;
    this.valor = valor;
  }
}
```

3. Implementación

La clase TablaHash: atributos y constructor

- El constructor recibe el número estimado de elementos a almacenar y reserva espacio para guardarlos con un FC del 75%
- Es altamente recomendable que el tamaño del *array* sea un número primo, pues mejora la dispersión de los datos

```
public class TablaHash<C, V> implements Map<C, V> {
  // Array de listas (con PI) de entradas
  private ListaConPI<EntradaHash<C,V>> elArray[];
  // Número de datos almacenados en la tabla
  private int talla;

  @SuppressWarnings("unchecked")
  public TablaHash(int tallaMaximaEstimada) {
    int capacidad = siguientePrimo((int)
                    (tallaMaximaEstimada/0.75));
    elArray = new LEGListaConPI[capacidad];
    for (int i = 0; i < elArray.length; i++)
       elArray[i] = new LEGListaConPI<EntradaHash<C,V>>();
    talla = 0;
  }
```

3. Implementación

Búsqueda de la posición de un dato en la tabla

```
/** Calcula la cubeta en la que debe estar un elemento
 *  con clave c. Para ello primero obtiene el valor de
 *  hash (hashCode) y a continuación su índice hash
 *  @param c   Clave del dato a localizar
 *  @return    Cubeta en la que se encuentra el dato
 */
protected int indiceHash(C c) {
  int indiceHash = c.hashCode() % this.elArray.length;
  if (indiceHash < 0)
      indiceHash += this.elArray.length;
  return indiceHash;
}
```

3. Implementación

Inserción de una entrada en la tabla

```
// Añade la entrada (c,v) y devuelve el antiguo valor
// que tenía dicha clave (o null si no tenía ningún
// valor asociado)
public V insertar(C c, V v) {
  V antiguoValor = null;
  int pos = indiceHash(c);
  ListaConPI<EntradaHash<C,V>> cubeta = elArray[pos];
  //Busqueda en cubeta de la entrada de clave c
  for (cubeta.inicio(); !cubeta.esFin() &&
      !cubeta.recuperar().clave.equals(c); cubeta.siguiente());
  if (cubeta.esFin()) { // Si no está insertamos la entrada
      cubeta.insertar(new EntradaHash<C,V>(c, v));
      talla++;       // Haría falta rehashing si se excede el FC
  } else {           // Si ya estaba actualizamos el valor
      antiguoValor = cubeta.recuperar().valor;
      cubeta.recuperar().valor = v;
  }
  return antiguoValor;
}
```

3. Implementación

Borrado de una entrada de la tabla

```
// Elimina la entrada con clave c y devuelve su valor
// asociado (o null si no hay ninguna entrada con dicha
// clave)
public V eliminar(C c) {
  int pos = indiceHash(c);
  ListaConPI<EntradaHash<C,V>> cubeta = elArray[pos];
  V valor = null;
  // Búsqueda en cubeta de la entrada de clave c
  for (cubeta.inicio(); !cubeta.esFin() &&
     !cubeta.recuperar().clave.equals(c); cubeta.siguiente());
  if (!cubeta.esFin()) {     // Si la encontramos la borramos
      valor = cubeta.recuperar().valor;
      cubeta.eliminar();
      talla--;
  }
  return valor;
}
```

3. Implementación

Búsqueda de entradas, esVacio y talla

```
// Busca la clave c y devuelve su informacion asociada
// o null si no hay una entrada con dicha clave
public V recuperar(C c) {
  int pos = indiceHash(c);
  ListaConPI<EntradaHash<C,V>> cubeta = elArray[pos];
  // Búsqueda en la cubeta de la entrada de clave c
  for (cubeta.inicio(); !cubeta.esFin() &&
     !cubeta.recuperar().clave.equals(c); cubeta.siguiente());
  if (cubeta.esFin()) return null;          // No encontrado
  else return cubeta.recuperar().valor;     // Encontrado
}

// Devuelve true si el Map está vacío
public boolean esVacio() { return talla == 0; }

// Devuelve el número de entradas que contiene el Map
public int talla() { return talla; }
```

Bibliografía

- Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java (4th edition)*. John Wiley & Sons, Inc., 2005.
  - Capítulo 9, apartados 1 y 2.
- Weiss, M.A. *Estructuras de Datos en Java*. Adisson-Wesley, 2000.
  - Capítulo 6, apartado 7, y capítulo 19.