Tema 13 - Implementación de un Diccionario según una Tabla de Dispersión

Germán Moltó
Escuela Técnica Superior de Ingeniería Informática
Universidad Politécnica de Valencia

Tema 13 - Implementación de un Diccionario según una Tabla de Dispersión

Índice general:

Objetivos

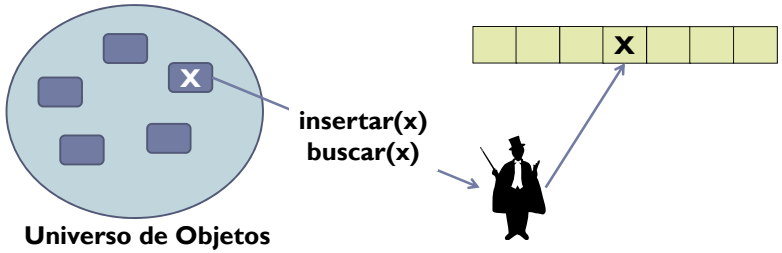
- Presentar la Tabla Hash como una Representación eficiente de la EDA Diccionario. En concreto, se estudiarán:
  - Los conceptos relacionados con su definición: Función de Dispersión (hashing), Conflictos y su Resolución mediante Encadenamiento Enlazado.
  - El Análisis de su Eficiencia, en función de las colisiones y su Factor de Carga.
- Desarrollar una implementación de Tabla Hash en Java.
- Estudiar las ventajas e inconvenientes que supone representar un Diccionario mediante una Tabla Hash o un ABB.

Bibliografía

- Libro de M.A. Weiss, "Estructuras de Datos en Java" (Adisson-Wesley, 2000).
  - Apartado 7 del capítulo 6 (Introducción a la Tabla de Dispersión).

Motivación

- Utilizar un array para almacenar un grupo de objetos requiere un coste lineal con el número de elementos (O(N)) para buscar uno de ellos.
  - Desconocer el punto exacto de la estructura dónde se almacena el objeto buscado implica realizar una búsqueda.
  - ¿Y si pudiera saber dónde guardar/buscar un objeto?



Motivación

- La operación básica del Diccionario es la búsqueda dinámica por nombre o clave en una colección de entradas.
- Una Implementación del Diccionario mediante un **ABB Equilibrado** (ABBDiccionario) consigue que sus operaciones se ejecuten en un **tiempo logarítmico** con su número de entradas.
- La Tabla Hash reducirá el coste de las operación de inserción, búsqueda y eliminación a **constante (O(1))**, sacrificando información relativa a la ordenación entre elementos.
  - La búsqueda del mínimo, el máximo u obtener la secuencia ordenada de elementos ya NO podrá obtenerse en tiempo lineal con el número de elementos.

Diccionario de Enteros de 16 bits (I)

- Implementar un diccionario de enteros de 16 bits (entre 0 y 65.535) mediante una representación que permita la búsqueda por nombre en **tiempo constante**.

0	0
0	1
0	2
.	.
.	.
.	.
.	.
.	.
.	.
.	.
.....	.
0	65.535

- Construimos un vector de tamaño máximo 65536 entradas.
- Cada posición del vector representa un Entero del Diccionario.
- Operaciones Principales:
  - Insertar el n° x: v[x]++;
  - Búsqueda del n° x: v[x] != 0
  - Eliminar el n° x: if (v[x] != 0) v[x]--;
- Realizamos las principales operaciones en tiempo constante.

Diccionario de Enteros de 16 bits (I)

- La solución anterior tiene varios problemas:
  - Si se quiere representar un Diccionario de Enteros de 32 bits, es necesaria una tabla que almacene 2^{32} (4 billones de elementos), lo que ocupa demasiada memoria RAM.
  - ¿Qué ocurre con los elementos que NO son enteros?, ¿Cómo se indexan esas Entradas?
- Interesa diseñar una solución genérica, que permita almacenar cualquier tipo de Objeto.
- Resulta necesario utilizar una función que, **dado un Objeto** cualquiera, **le asocie un número** que permita indexar la tabla.
  - Para inserción, búsqueda y borrado.

Función de Dispersión (Hashing)

- La **Función de Dispersión** convierte un Objeto en un Entero (valor Hash) adecuado para indexar la Tabla en la que se guardará el Objeto.
- Java dispone de mecanismos estándar para obtener valores hash a partir de cualquier Objeto:
  - En la clase Object está definido el método:
    public int **hashCode**()
  - Este método devuelve un valor entero a partir del objeto sobre el que se invoca (obtenido a partir de la dirección de memoria del objeto sobre el que se invoca).
- Dado que el método está implementado en la raíz de la jerarquía de herencia, cualquier clase dispone de una implementación por defecto.

Método hashCode() de la clase Object

- Obtenido a partir del API de Java Platform SE 6.0
- Returns a hash code value for the object. This method is supported for the benefit of hashtables such as those provided by java.util.Hashtable. 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 hashtables.

Método hashCode() de la clase Object (II)

- Ideas Traducidas al Español:
  - Dos invocaciones al método hashCode de un **mismo objeto** durante una ejecución de la aplicación deben devolver el **mismo valor** entero.
  - Si **dos objetos son iguales** (de acuerdo a la definición de **equals**) el método hashCode debe devolver el **mismo valor** entero.
  - No es necesario que dos **objetos diferentes** devuelvan **valores diferentes** cuando les sea invocado el método hashCode. Sin embargo, devolver valores diferentes puede **mejorar las prestaciones** de la tabla hash en la que se empleen.
- Además, los objetos no tienen porqué ser Comparables.

Ejemplo de Implementación de hashCode

- Documentación del método hashCode() de la clase String (extraído del API de Java Platform SE 6.0)
- 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) + ... + 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 (sobreescribe): hashCode in class Object
- La clase String redefine el método hashCode de Object:
  - Al invocar el método sobre dos objetos iguales (de acuerdo al criterio de *equals*) se obtiene el mismo valor.
    - Esto no ocurre si únicamente se consideran direcciones de memoria.

Problemas en el uso de valores hash para String

- Si convertimos un String a un tipo primitivo int que represente un valor hash, ¿Éste valor serviría como índice para indexar la tabla hash?
- Ejemplo: Asumimos que un carácter puede ser representado por 7 bits como un número en [0, 127].
- Representamos la cadena 'hola' como un entero:
  'h'*128^3 + 'o'*128^2 + 'l'*128^1 + 'a'*128^0 → 219936353
- El valor obtenido es muy grande, y cadenas más grandes harían obtener índices todavía mayores, lo que implicaría disponer de un vector excesivamente grande.

¿Cómo evitar el uso de índices demasiado grandes que requieren un vector de gran tamaño?

Función de dispersión apropiada para Tabla Hash

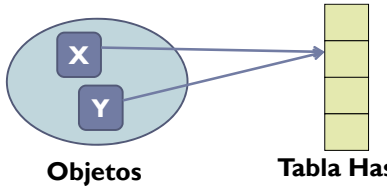
- Utilizamos el operador *módulo* (resto de la división entera): %
  - Para obtener un índice adecuado para una Tabla Hash.
- Si valorHash es un entero no negativo arbitrario, entonces

indiceHash = valorHash % capacidadDelArray es un entero en **[0, capacidadDelArray-1]**

- Ejemplo de uso de la Función de Dispersión (capacidadDelArray = 100):
  - valorHash = 50 → 50%100=50; valorHash = 100 → 100%100 = 0
  - valorHash = 125→125%100=25; valorHash=225 → 225%100 = 25

Colisiones provocadas por la Función de Dispersión

- Importante: Hay más valores Hash que entradas en la Tabla y, por lo tanto, es de esperar que se produzcan **colisiones**
  - Cuando a **dos valores diferentes** (en base al criterio de equals) les corresponde **la misma posición** en la tabla (el mismo valor hash).
    - **!**o1.**equals**(o2) **&&** indiceHashO1 **==** indiceHashO2



Objetos diferentes reciben la misma entrada en la Tabla Hash. Es una colisión.

Paradoja del cumpleaños

Sobre la Función Hash (I)

- Una buena función Hash debe:
  - Distribuir uniformemente la asignación de las casillas de la tabla para reducir el número de colisiones.
  - Tener un coste computacional relativamente bajo.
- ¿Dónde se implementa la Función Hash en Java?
  - Sobreescribiendo el método *hashCode* en la clase de los objetos a almacenar en la Tabla Hash.
- ¿Qué implicaciones tiene sobreescribir el método *hashCode?*
  - Se debe sobreescribir también el método *equals* de la clase de los objetos a almacenar en la Tabla Hash.

¿Cuál es la peor función Hash posible?

Ejemplo de Implementación de hashCode

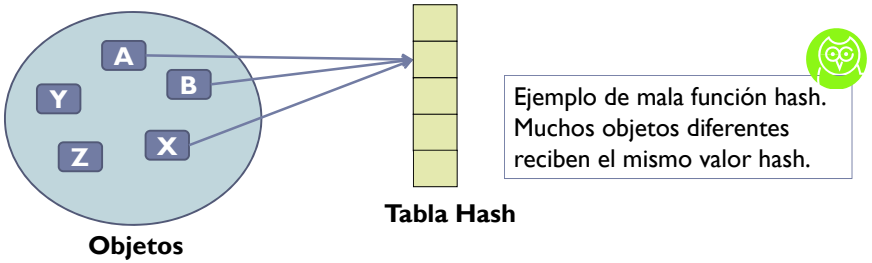
```
public class Persona{
   private String nif;   private String nombre;   private int edad;

   public Persona(String nif, String nombre, int edad){
     this.nif = nif;  this.nombre = nombre; this.edad = edad;
    }
   public int hashCode(){  return this.nif.hashCode();  }

    public boolean equals(Object x){
        Persona o = (Persona) x;
        return this.nif.equals(o.nif);
    }
}
```

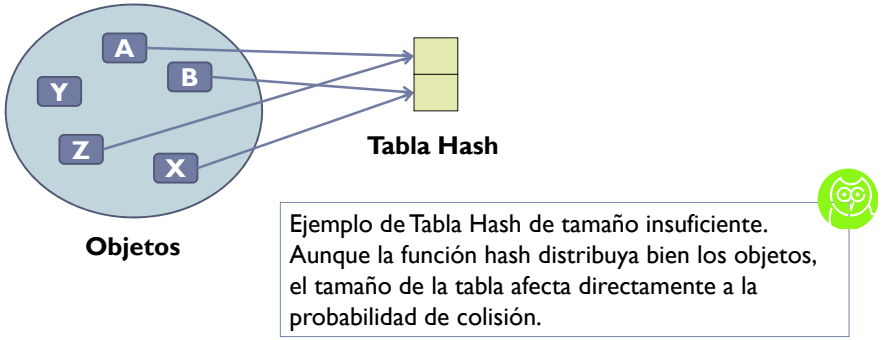

Sobre la Función Hash (II)

- La función de dispersión (hash(x)) debe calcularse en tiempo constate y distribuir de forma uniforme los objetos a lo largo de la tabla.



Sobre la Tabla Hash

- La Tabla Hash debe tener un tamaño relativamente grande para reducir el número de colisiones.
  - Tabla Hash de una sola entrada: ¡N-1 colisiones para N datos!

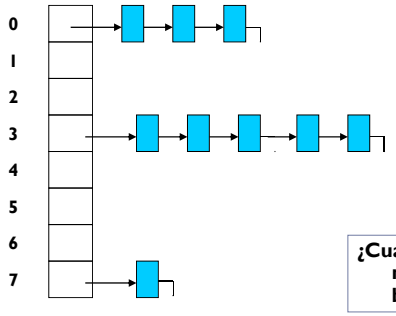


Formas de Resolver las Colisiones

- Resolución Mediante Exploración:
  - Trata de buscar otra posición libre en la tabla para albergar la inserción del elemento.
    - Exploración Lineal
      - Visita la siguiente casilla. Si está libre, realiza la inserción. Si no, visita la siguiente casilla.
    - Exploración Cuadrática
      - Visita la casilla i^2 posiciones más allá de la que causó la colisión. Si está libre, realiza la inserción. Si no, repite el proceso hasta encontrar una libre.
- Resolución Mediante Encadenamiento Enlazado:
  - En cada posición de la tabla se mantiene una Lista Enlazada en la que se van insertando los elementos cuyo valor de hash les asigna la misma posición.

Hashing Enlazado

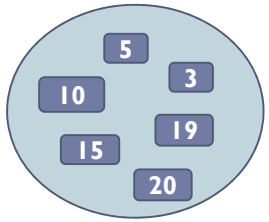
- El Hashing Enlazado usa un vector de Listas Enlazadas.
  - Aquellos objetos que reciban un determinado valor de Hash, se insertarán en la lista enlazada correspondiente.



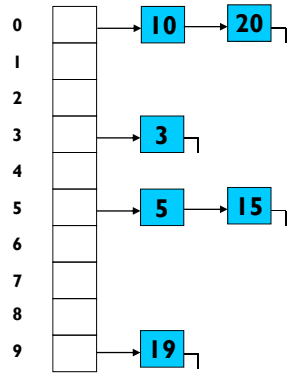
¿Cuál es el tamaño óptimo de las listas para mantener una buena eficiencia en las búsquedas de elementos de la Tabla Hash?

Ejemplo de Uso de Tabla Hash

- Insertar los elementos {5, 10, 3, 15, 20, 19} en una Tabla Hash de tamaño 10 y con función hash:
  - hash(x) = x % 10



Objetos



- Factor de Carga
  - Elementos / Capacidad = 6/10

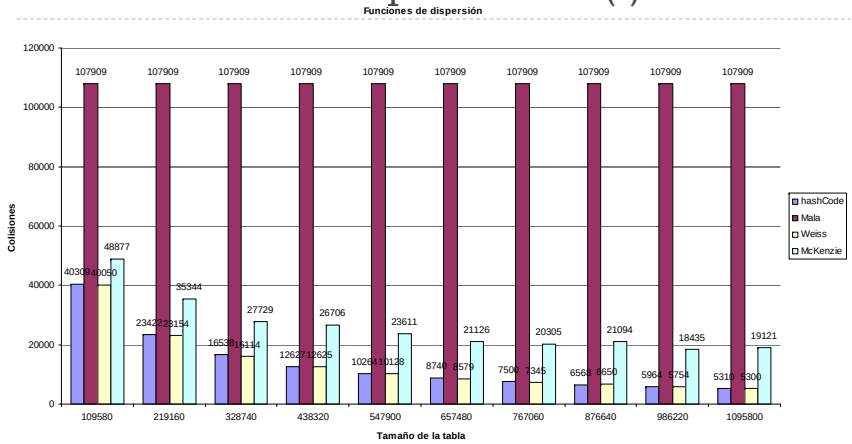
El Factor de Carga de una Tabla Hash

- Factor de Carga de una Tabla Hash:
  - Relación entre el número de posiciones ocupadas de la Tabla y su dimensión (valor en [0,1]).
- Para examinar los efectos que tiene la **Calidad de la Función de Dispersión** y el **Factor de Carga** de una Tabla Hash sobre el **Número de Colisiones** se propone el siguiente experimento de simulación:
  - Sea 109.580 el número de cadenas y sea la longitud de clave máxima 28. Se calculará el número de colisiones que se producen cuando se insertan esas entradas:
    - Al utilizar las 4 funciones de dispersión descritas en la página siguiente: hashCode, Weiss, Mala y McKenzie
    - Al incrementar en cada paso de la simulación el tamaño del array en 109.580 unidades, desde 109.580 hasta 1.095.800

Funciones Hash utilizadas en el Experimento

1. `public int hashCode(){ return this.clave.hashCode();}`
2.
```
public int hashCode(){
    /** Weiss: en Weiss, capítulo 19 apartado 2, figura 19.2 */
    int valorHash = 0;
    for ( int i = 0 ; i < clave.length() ; i++ )
      valorHash = 37 * valorHash + clave.charAt(i);
    return valorHash;
  }
```
3.
```
public int hashCode(){
   /** Mala: en Weiss, capítulo 19 apartado 2, figura 19.3 */
    int valorHash = 0;
    for ( int i = 0 ; i < clave.length() ; i++ )  valorHash += clave.charAt(i);
    return valorHash;
  }
```
4. McKenzie: substituir 37 por 4 en Weiss

Resultados del Experimento (I)

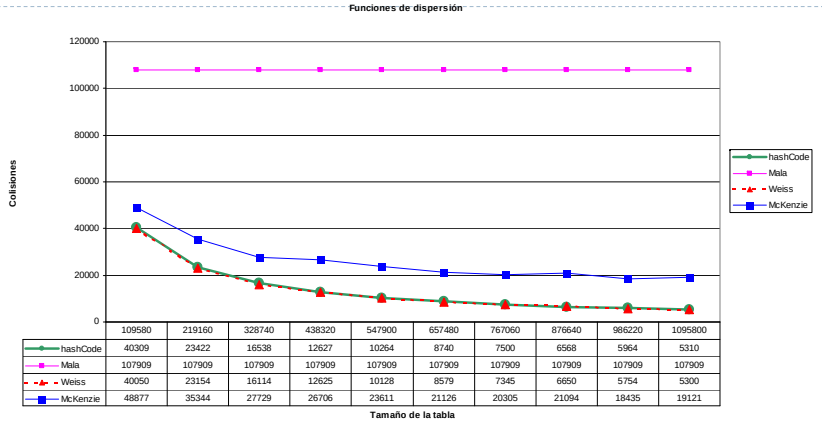


- Aumentar el tamaño de la tabla (reducir el factor de carga), reduce las colisiones.
- Cuanto mejor es la función de dispersión, menor número de colisiones provoca.

Sobre Resultados del Experimento (I)

- La función Mala siempre provoca el mismo número de colisiones porque únicamente proporciona valores en [0, 3556].
  - Asumiendo que charAt(i) ∈ [0, 127], como la longitud de la palabra es de 28 caracteres, el máximo valor que puede devolver esa función hash es 28*127 = 3556
- Dado que el número de elementos a insertar es mucho mayor, en cuanto esas posiciones hayan sido ocupadas, el resto de entradas provocarán irremediablemente una colisión.
- Nótese que, por lo tanto, interesa que la función hash distribuya uniformemente los valores por toda la tabla.
  - De esta manera, es posible reducir el número de colisiones al encontrar, en primera instancia, una posición libre.
- Calidad y Eficiencia se deben combinar en el diseño de la función de dispersión, pero debe prevalecer el de calidad.

Resultados del Experimento (II)



	109580	219160	328740	438320	547900	657480	767060	876640	986220	1095800
hashCode	40309	23422	16538	12627	10264	8740	7500	6568	5964	5310
Mala	107909	107909	107909	107909	107909	107909	107909	107909	107909	107909
Weiss	40050	23154	16114	12625	10128	8579	7345	6650	5754	5300
McKenzie	48877	35344	27729	26706	23611	21126	20305	21094	18435	19121

- El número de colisiones disminuye conforme aumenta el tamaño de la tabla, ya que se reduce la probabilidad de obtener dos índices iguales.
- El número de colisiones se reduce conforme disminuye el Factor de Carga.

Rehashing en una Tabla Hash

- Si el Factor de Carga de una Tabla Hash es alto el número de colisiones aumenta rápidamente.
- Solución: Rehashing
  - Incrementar el tamaño de la tabla hash para reducir el factor de carga (grado de ocupación de la tabla).
- Ventajas:
  - Permite mantener un reducido factor de carga para que las principales operaciones (insertar, buscar, eliminar) se realicen en tiempo constante.
- Inconvenientes:
  - Requiere construir un nuevo array e insertar nuevamente todos los datos.

La clase TablaHash (I)

```
package librerias.estructurasDeDatos.hash;
Import librerias.estructurasDeDatos.modelos.*;
import librerias.estructurasDeDatos.lineales.*;
import librerias.excepciones.*;
public class TablaHash<E> {
    private ListaConPI<E> elArray[];
    private int talla;
    private static final int TAMANYO_DEFECTO = 50;

public TablaHash() {
        this(TAMANYO_DEFECTO);
    }
```


La Clase TablaHash (II)

```
@SuppressWarnings("unchecked")
 public TablaHash(int tallaMaximaEstimada){
    int capacidad = siguientePrimo((int)(tallaMaximaEstimada/0.75));
    this.elArray = new LEGListaConPI[capacidad];
    for (int i=0; i< elArray.length; i++ ) elArray[i] = new LEGListaConPI<E>();
    talla = 0;
 }
 protected static final int siguientePrimo(int n){
        if ( n % 2 == 0 ) n++;  for ( ; !esPrimo(n); n += 2 ) ; return n;
    }
protected static final boolean esPrimo(int n){
        for (int i = 3 ; i*i <= n; i += 2 )
          if ( n % i == 0 ) return false;  return true; }
```


La Clase TablaHash (III)

```
protected int posTabla(E x) {
        int indiceHashX =  x.hashCode() % this.elArray.length;
        if (indiceHashX < 0) indiceHashX += this.elArray.length;
        return indiceHashX;
    }
 public final boolean esVacia(){ return ( this.talla == 0 );  }
 public final int talla() { return this.talla; }
 public final double factorCarga() {
        return (double)(talla)/elArray.length;
    }
```


La Clase TablaHash (IV)

```
public final void insertar(E x) {
        ListaConPI<E> lpi = elArray[posTabla(x)];
        for (lpi.inicio(); !lpi.esFin() && !lpi.recuperar().equals(x); lpi.siguiente());
        lpi.insertar(x);
        if (lpi.esFin()) {     // Insertar nuevo
            talla++;
            if (factorCarga() > 1.5) rehashing();
        } else {               // Actualizar
            lpi.eliminar();
        }
    }
```

- Si el elemento *x* no existía, lo añade al final. Si ya existía, entonces inserta el nuevo elemento y reemplaza el antiguo.

La Clase TablaHash (V)

```
public final E recuperar(E x) throws ElementoNoEncontrado {
        ListaConPI<E> lpi = elArray[posTabla(x)];
        for (lpi.inicio(); !lpi.esFin() && !lpi.recuperar().equals(x); lpi.siguiente());
        if (lpi.esFin()) throw new ElementoNoEncontrado(x + " no está");
        return lpi.recuperar();
    }
```

- Para buscar el elemento, el método posTabla indica en qué lista se debe buscar por si está el elemento (a partir del código hash del objeto). Si no está en dicha lista, se notifica con la excepción.

La Clase TablaHash (VI)

```
public final void eliminar(E x) throws ElementoNoEncontrado {
        ListaConPI<E> lpi = elArray[posTabla(x)];
        for (lpi.inicio(); !lpi.esFin() && !lpi.recuperar().equals(x); lpi.siguiente());
        if (lpi.esFin()) throw new ElementoNoEncontrado(x + " no está");
        lpi.eliminar();
        talla--;
    }
```

- La búsqueda en la ListaConPI termina bien con el PI tras el último elemento (lpi.esFin() vale true) o con el PI apuntando al elemento encontrado. En ese último caso, se debe eliminar el elemento.

La Clase TablaHash (VII)

```
public final String toString(){
        String res = "";
        for (int i = 0; i < this.elArray.length; i++) {
            ListaConPI<E> lpi = elArray[i];
            for (lpi.inicio(); !lpi.esFin(); lpi.siguiente())
              res += lpi.recuperar() + "\n";
        }
        return res;
    }
```


Tabla Hash vs Árbol Binario de Búsqueda

- Las Tablas Hash presentan mejores tiempos de búsqueda, inserción y eliminado ($\Theta(1)$) que los ABBs ($\Theta(\log_2(N))$). Sin embargo, hay que tener en consideración:
  - La Tabla Hash tiene que tener un factor de carga moderado para evitar realizar búsquedas costosas en las listas enlazadas.
  - Baja localidad de referencia de las Tablas Hash (accesos aleatorios a memoria) que provocan numerosos (y costosos) fallos de página.
- El ABB:
  - Permite mantener la ordenación de los elementos (recorrido inorden).
  - La Tabla Hash no mantiene el orden. Es muy costoso obtener una secuencia ordenada de sus elementos, así como obtener el sucesor de un dato (o el mínimo, máximo, etc.).

Sobre la Implementación de un Diccionario Mediante una Tabla Hash

```
package librerias.estructurasDeDatos.modelos;
public interface Diccionario<C,V> {
        void insertar(C c,V v);
        V recuperar(C c) throws ElementoNoEncontrado;
        void eliminar(C c) throws ElementoNoEncontrado;
        boolean esVacio();
        int talla();
        ListaConPI<C> toLPIClaves();
}
```

- Es necesaria una clase intermedia, EntradaDic, para establecer la correspondencia entre los pares clave, valor.
  - La clase debe sobreescribir los métodos equals y hashCode (necesarios para ser utilizada en TablaHash). Los implementa delegando en los correspondientes métodos de la clave.

La Clase EntradaDic

- Esta clase permite almacenar los pares clave,valor.

```
package librerias.estructurasDeDatos.hash;
class EntradaDic<C,V> {
   C clave;
   V valor;
   EntradaDic(C c,V v) { clave = c; valor = v; }
   EntradaDic(C c) { this(c, null); }
   @SuppressWarnings("unchecked")
   public boolean equals(Object x) {
      return ((EntradaDic<C,V>)x).clave.equals(this.clave);
   }
   public String toString() {
       return "(" + this.clave + "," + this.valor + ")";
   }
   public int hashCode() { return clave.hashCode(); }}
```


La Clase TablaHashDiccionario (I)

```
public class TablaHashDiccionario<C,V> implements
   Diccionario<C,V>{
    private TablaHash<EntradaDic<C,V>> th;

    public TablaHashDiccionario() {
        th = new TablaHash<EntradaDic<C,V>>();
    }
    public TablaHashDiccionario(int tallaMaximaEstimada) {
        th = new TablaHash<EntradaDic<C,V>>(tallaMaximaEstimada);
    }
 public void insertar(C c,V v) {
        th.insertar(new EntradaDic<C,V>(c, v));
    }
```


La Clase TablaHashDiccionario (II)

```
public void eliminar(C c) throws ElementoNoEncontrado {
        th.eliminar(new EntradaDic<C,V>(c));
    }
public V recuperar(C c) throws ElementoNoEncontrado {
        return th.recuperar(new EntradaDic<C,V>(c)).valor;
    }
 public boolean esVacio() {
       return th.esVacia();
 }
 public int talla() {
      return th.talla();
}
```

La Clase TablaHashDiccionario (III)

```
  @SuppressWarnings("unchecked")
   public ListaConPI<C> toLPIClaves() {
       EntradaDic<C,V> v[] = new EntradaDic[th.talla()];
       th.toArray(v);
       ListaConPI<C> lpi = new LEGListaConPI<C>();
       for (int i = 0; i < v.length; i++) {
           lpi.insertar(v[i].clave);
       }
       return lpi;
   }

} /* Fin de la clase TablaHashDiccionario */
```


Ejemplo de Uso de Diccionario y TablaHashDiccionario (I)

- Objetivo:
  - Uso de Diccionario y TablaHashDiccionario para una sencilla aplicación de gestión de un garaje.
  - Es necesario establecer la correspondencia entre Personas y Coches.
- Importante:
  - Establecer qué clase será la clave del Diccionario (clase Persona).
  - La clase de la clave (Persona) debe implementar obligatoriamente los métodos equals y hashCode para poder trabajar con un Diccionario implementado mediante una TablaHash.

Ejemplo de Uso de Diccionario y TablaHashDiccionario (II)

```
public class Persona{
   private String nif;   private String nombre;   private int edad;
   public Persona(String nif, String nombre, int edad){
     this.nif = nif;  this.nombre = nombre; this.edad = edad;
    }
    public Persona(String nif){  this.nif = nif;  }
    public int hashCode(){  return this.nif.hashCode();  }
    public boolean equals(Object x){
        Persona o = (Persona) x;
        return this.nif.equals(o.nif);
    }
  public String toString(){return this.nif + " " + this.nombre + " " + this.edad; }
}
```


Ejemplo de Uso de Diccionario y TablaHashDiccionario (III)

```
public class Coche{
  private String matricula;
  private String modelo;
  public Coche(String mat, String mod){
      this.matricula = mat; this.modelo = mod;
    }
   public String toString(){return this.matricula + " " + this.modelo;}
}
```

Ejemplo de Uso de Diccionario y TablaHashDiccionario (IV)

```
public class TestGaraje{
  public static void main(String args[]){
  Diccionario<Persona,Coche> garaje = new TablaHashDiccionario<Persona,Coche>();
    Persona p = new Persona("647262929F","Paco Ruiz",55);
    Coche c = new Coche("8987FGG","Alfa Romeo");
    garaje.insertar(p,c);
    Persona p2 = new Persona("647262929F");
     try{
        System.out.println("El coche de " + p2 + " es " + garaje.recuperar(p2));
     }catch(ElementoNoEncontrado ex){
         System.err.println("No está el coche de " + p2 );
        }
    }
}
```


Comentarios sobre el Ejemplo

- El método *equals* de la clase Persona utiliza exclusivamente el NIF para determinar la igualdad entre objetos de tipo Persona.
  - Por eso es posible recuperar del Diccionario a través del objeto p2, que únicamente incluye información sobre el NIF.
- Por supuesto, sería posible también recuperar del Diccionario usando el mismo objeto *p* que previamente se ha insertado.

Tablas de Dispersión: Conclusiones

- Las Tablas Hash permite que el coste medio de las operaciones *insertar, buscar* y *eliminar* sea constante.
- El Factor de Carga es un parámetro importante de cara a mantener las buenas prestaciones de la Tabla.
- Hay que elegir correctamente la función de hash:
  - Debe ser fácilmente calculable, sin involucrar costosas operaciones.
  - Debe tener una buena distribución de valores entre todas las componentes de la tabla.
- Ejemplos de Aplicación de Tablas de Dispersión:
  - Corrector Ortográfico, Tablas de Símbolos de un Compilador, etc.