Mejor un array con número primo

\+ dispersión es mejor

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();
}
```

2. Tabla de Dispersión

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

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

Si es negativo debemos transformarlo a positivo

Procedimiento para dispersar la Clave c en elArray:

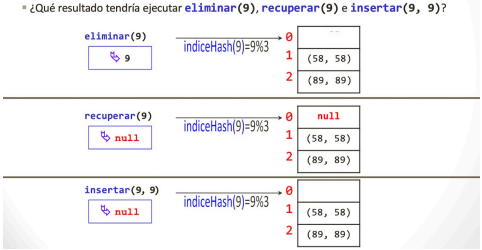
```
public int indiceHash(C c) {
   int valorHash = c.hashCode();
   // Método de la división: si valorHash ≥ 0 …
   int indiceHash = valorHash % elArray.length;
   // Overflow de enteros
   if (indiceHash < 0) { indiceHash += elArray.length; }
   // 0 ≤ indiceHash < elArray.length
   return indiceHash;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

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)**?



Elección cuidadosa de un número primo para `elArray.length`

Para evitar correlación entre números de hash se usan largos primos.

Para evitar colisiones, elijo una buena función.

Mala -> función que genera números que dispersan poco

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;
}
```

más pequeño -> 0*28
más grande -> 127*28

109580 | 3557

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]** ...

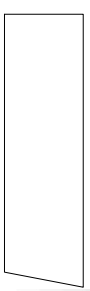
¿Por qué se producen **siempre** el mismo número de colisiones, independientemente del valor de **`elArray.length`** que tenga la Tabla?

Funciones de dispersión

Map

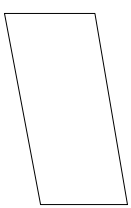
insertar(c,v)
recuperar(c)
eliminar(c)
talla()

clave → hashcode
valor



compresión

%elArray.length

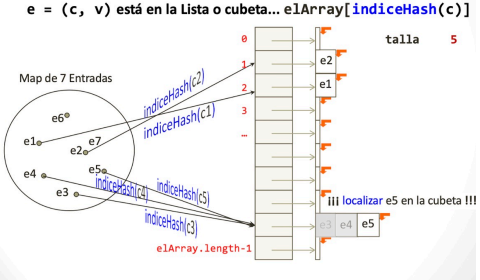


2. Tabla de Dispersión

Colisiones: resolución por Hashing Enlazado

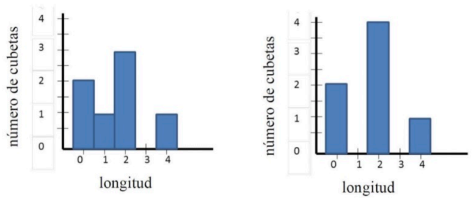
Todas las Entradas que colisionan en una misma posición de **`elArray`** se almacenan en la **Lista de Colisiones**, o **Cubeta**, asociada a dicha posición:

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



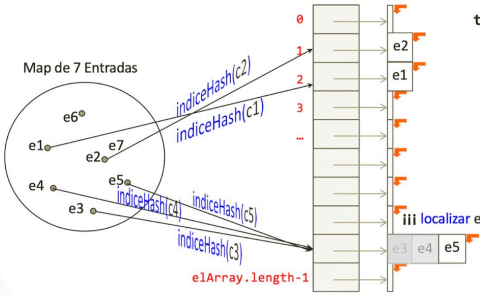
Ejercicio 2.4

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.



7 cubetas
11 elementos

Todas las Entradas que colisionan en una misma posición de e
almacenan en la **Lista de Colisiones**, o **Cubeta**, asociada a dic
e = (c, v) está en la Lista o cubeta... elArray[indic



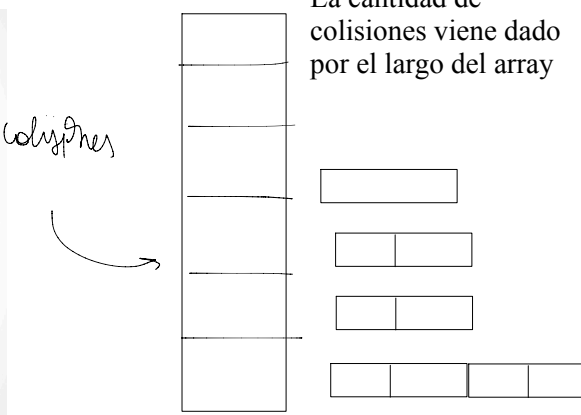
ijado **hashCode** y suponiendo constante su tiempo de ejecución, ...
Una Tabla Hash de **n** = `talla` Entradas será eficiente siempre que el tiempo romedio ($T^{\square}_{localizarC}$**(n)**) que se tarda en localizar en ella cualquier clave **c** ea constante. Dado que ...

- Una Entrada de clave **c** SOLO puede ubicarse en la cubeta **elArray[indiceHash(c)]**, $T^{\square}_{localizarC}$**(n)** será directamente proporcional a la longitud MEDIA de las cubetas de la Tabla.
- La longitud de cualquier cubeta **elArray[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),
$T^{\square}_{localizarC}$**(n) es constante**

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

talla/elArray.length



La cantidad de colisiones viene dado por el largo del array

4 colisiones

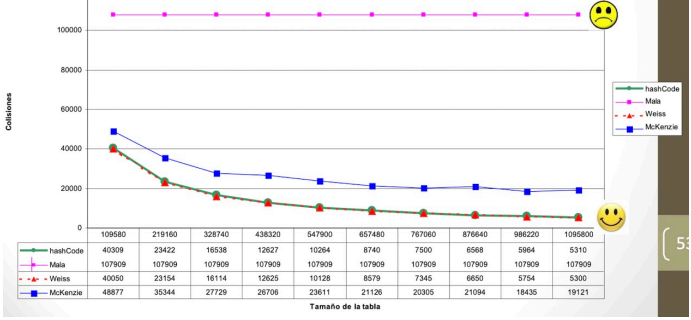
Ejemplo de análisis del comportamiento de una Tabla Hash en el Experimento 1.

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.



	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

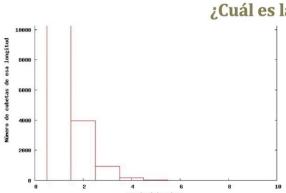
evitar cubetas grandes

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

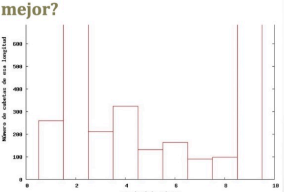
Calcular y mostrar su histograma de ocupación y la desviación típica.

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

¿Cuál es la mejor?



Desviación típica < 0.86



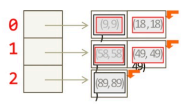
Desviación típica > 12

2. Tabla de Dispersión

Eficiencia: Rehashing

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

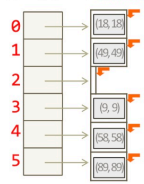
FC = 5/3 → $T_{localizarC}(n) = 7/5 \in O(FC)$



➔ ***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 → $T_{localizarC}(n) = 5/5 \in O(FC)$



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

Map

insertar(c,v)
recuperar(c) -> v o null
eliminar(c) -> v o null

ListaConPI

insertar(e)
recuperar()
eliminar()
siguiente()
inicio()
fin()
esFin()
esVacia()

```
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;
        }}
```

```
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 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.

```java
public class RadarApp {

    private Map<Matricula, Integer> map;

    public RadarApp() {
        map = new TablaHash<Matricula, Integer>(1000);
    }

    public void registrar(Matricula mat) {
        // COMPLETAR
        Integer freq = map.recuperar(mat);
        if(freq==null){
            map.insertar(mat, 1);
        }
        else{
            map.insertar(mat, freq+1);
        }
    }
}
```

considerar que así se cambia

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.

```java
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();
}
```

```java
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 los tallas de **l1** y **l2**

Coste entre n y n^2

talla_array != tallahash (número de valores que de vdd tengo)

Operación de rehashing

Total cubetas -> array.length