Estructuras de datos y algoritmos

Tema 3 Map y Tabla Hash

Curso 2018-2019

El modelo Map

El modelo Map

Map es un diccionario o array asociativo:

- Guarda pares *(clave,valor)*.
- No se permiten claves repetidas.

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

- Permite buscar por la clave: dada una clave, queremos saber si está en el Map y recuperar su valor asociado.

Implementación en Java: la interfaz Map

```
package librerias.estructurasDeDatos.modelos;
public interface Map<C, V> {
  // inserta/actualiza la entrada (c,v) devuelve
  // valor anterior o null si no estaba
  V insertar(C c, V v);
  // elimina entrada con clave c, devuelve su
  // valor asociado, o null si no existe
  V eliminar(C c);
  // devuelve el valor asociado a la clave c
  // o null si no existia esa entrada
  V recuperar(C c);
  boolean esVacio();
  int talla();
  // devuelve una ListaConPI que contiene todas las claves
  ListaConPI<C> claves();
}
```

Ejemplos de uso

- Traducción automática:
  - Traducir palabra por palabra requeriría un diccionario.
  - Aunque traducir palabra por palabra funcionaría mal, muchos sistemas de traducción reales utilizan diccionarios (realmente enormes, pues suelen traducir grupos de palabras y además les asignan varias alternativas acompañadas de probabilidades).
- La tabla de símbolos de un compilador (va almacenando los nombres de variables, clases, métodos, etc.). Esta tabla se modifica a lo largo del proceso de compilación (algunos símbolos se introducen y luego se eliminan: las variables locales solamente existen en su método, etc.).

Ejemplos de uso

- Calcular la frecuencia de aparición de las palabras de un vocabulario en un documento puede ser muy útil en algunas tareas de procesamiento del lenguaje: podríamos crear un diccionario que utilice:
  - **Clave:** de tipo `String` para almacenar las palabras.
  - **Valor** de tipo `Integer` (la clase *envoltorio* de `int`) donde guardaremos, para cada palabra, el número de veces que ha aparecido.

Pos. 1: Implementación Map con lista enlazada

- Conocida como association list.
- Mantendríamos una lista de pares *(clave,valor)*.
- Coste de insertar, buscar y borrar es **lineal** con el número de entradas:
  - En el caso peor hay que recorrer toda la lista.
  - En un caso promedio, si cada valor de clave tuviese la misma probabilidad, el coste promedio seguiría siendo lineal.
- Podríamos tener la lista ordenada por claves ¿qué ventajas e inconvenientes tendría?
  - El caso peor y promedio sigue siendo lineal.

Pos. 2: Implementación Map con un vector

Si el conjunto de claves posibles fuesen los números del `0` al `N-1`, está claro que podríamos usar directamente las claves como índices de un vector de talla `N` donde guardaríamos los valores asociados:

- El coste de buscar, insertar o borrar es **constante**.
- Si una clave no está en el diccionario:
  - Si los valores son objetos, guardamos `null`.
  - Si los valores son tipos básicos, guardamos (de ser posible) un valor *especial*, una implementación genérica en Java no permite tipos básicos (usa clases envoltorio).
- Si el conjunto de valores no son los enteros entre `0` y `N-1`:
  - Si es un conjunto pequeño, usamos una función que mapee cada valor a un entero diferente entre `0` y `N-1`.
  - Si el conjunto de valores es muy grande, sería muy ineficiente en espacio (si guardas un pequeño porcentaje de entradas) o directamente imposible (ej: claves de tipo `String`).

Tablas de dispersión

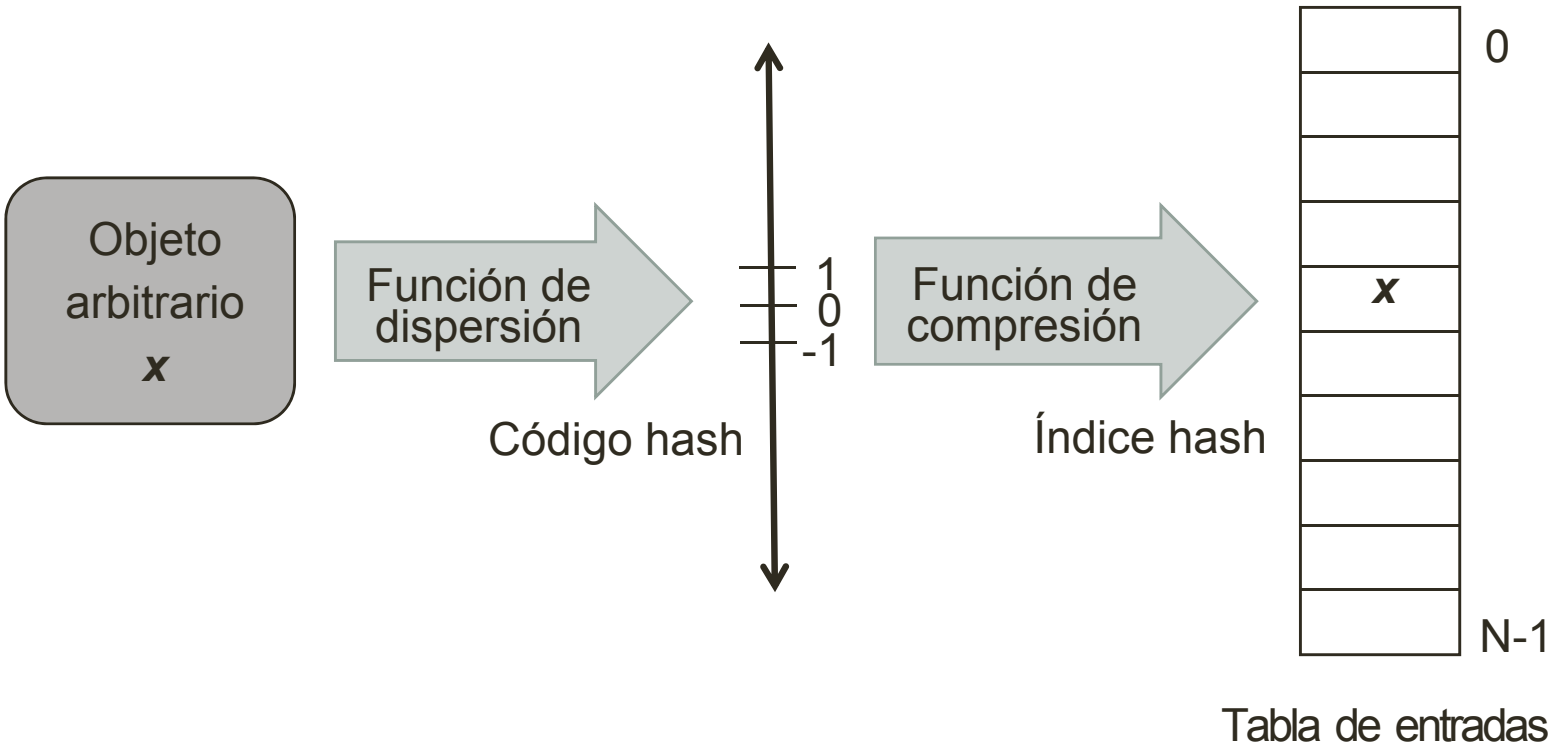
Tablas de dispersión

- Hemos visto que las **listas enlazadas** permiten implementar un diccionario para cualquier tipo de clave (basta que tenga el método `equals`), pero **son caras** (en este contexto, lineal es muy caro).
- Por otra parte, los **vectores** son muy **rápidos pero** están enormemente **limitados** por el tipo de claves soportadas.
- El objetivo de las **tablas de dispersión** es tener un Map al mismo tiempo **general y eficiente**.

Funciones de dispersión

Una característica principal de las tablas de dispersión es la combinación de vectores y de funciones que permiten convertir los elementos a buscar en valores numéricos que pueda ser utilizados **como índices en el vector**. Estas funciones se conocen como **funciones de dispersión** (en inglés, *hash functions*).

Esquema


Objeto
arbitrario
x
Función de
dispersión
1
0
-1
Código hash
Función de
compresión
Índice hash
0
x
N-1
Tabla de entradas

Tablas de dispersión

En general, una tabla de dispersión (*hash*) se caracteriza por:

- Uso de un vector para acceder a los datos. Las componentes del mismo se denominan cubetas (o *buckets*).
- Una función **de dispersión** para obtener un índice del vector a partir de la clave a buscar o a insertar.
- Un mecanismo para **resolver las colisiones.** Es posible que una función de dispersión genere un mismo índice de vector para elementos diferentes. Esto se conoce como **colisión** y es **inevitable** si hay más tipos de clave que posiciones en el vector.

Colisiones

Figura : Pigeonhole Principle

Funciones de dispersión

Aunque se estudiarán posteriormente con más detalle, avanzamos algunas ideas aquí. . . .

Una **función de dispersión** es una función que convierte una clave (no necesariamente numérica) en un entero adecuado para indexar en la tabla en la que dicho objeto se ha de almacenar.

- Un método simple: si la clave es una palabra, podemos sumar los códigos ASCII de los caracteres:

casa $\rightarrow 99 + 97 + 115 + 97 = 408$

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

Funciones de dispersión

- La suma de componentes no es una buena función de dispersión ya que es fácil que dos claves distintas tengan el mismo código y se genera una colisión:

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

teja $\rightarrow 116 + 101 + 106 + 97 = 420$

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

hola $\rightarrow 104 \cdot 2^3 + 111 \cdot 2^2 + 108 \cdot 2^1 + 97 \cdot 2^0 = 1589$

teja $\rightarrow 116 \cdot 2^3 + 101 \cdot 2^2 + 106 \cdot 2^1 + 97 \cdot 2^0 = 1641$

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 (si código hash negativo)

Resolución de colisiones

Existen diversos mecanismos para tratar las colisiones (ver enlace), pero en este tema nos limitamos a describir únicamente dos:

- **Direccionamiento abierto** *(open addressing)*: los valores se guardan en el propio vector. Las veremos muy por encima (no veremos su implementación ni las utilizaremos en el resto de la asignatura).
- **Encadenamiento** *(separate chaining)* o direccionamiento cerrado: cada cubeta contiene una lista con todos los pares *(clave,valor)* correspondientes a aquellas claves que han producido el índice correspondiente por medio de la función de dispersión.

Tablas de dispersión con direccionamiento abierto

- Los valores se guardan en el propio vector.
- Cada cubeta contiene a lo sumo un par *(clave,valor)*.
- Si al insertar un elemento la cubeta ya está ocupada, hay que buscar **otra posición** hasta encontrar una posición libre.
- Existen diversas formas de buscar otra posición (todas ellas implementando circularidad):
  - **Exploración lineal**: vamos avanzando el índice en intervalos constantes (normalmente de 1 en 1). El salto debe ser coprimo con el número de cubetas.
  - **Doble hashing**: a partir de la clave obtenemos tanto el índice inicial como el salto o incremento.
  - **Exploración cuadrática**: desde el primer índice vamos sumando $1^2, 2^2, 3^2, \ldots, i^2, \ldots$

Tablas de dispersión con direccionamiento abierto

Las tablas con direccionamiento abierto tienen algunos inconvenientes:

- No permiten almacenar un número de elementos mayor que la talla del vector, si bien podemos redispersar la tabla para aumentar su tamaño.
- El coste de insertar crece de manera dramática cuando la tabla empieza a llenarse.
- Al borrar un elemento hay que marcar la posición de manera especial para saber que al buscar hay que seguir buscando pero al insertar se puede considerar un hueco.

A pesar de todo, son muy utilizadas y pueden tener ventajas en casos concretos (ej: permite borrar algunas tablas en tiempo constante, etc.).

Tablas de dispersión con encadenamiento

- Una vez obtenida la cubeta correspondiente a una clave, no miramos en otras cubetas para buscar dicha clave.
- Resolución de colisiones: permitimos varios pares *(clave,valor)* en la misma cubeta (normalmente se utiliza una lista).

Ejemplo de tabla con encadenamiento

Veamos una tabla de dispersión con encadenamiento y 10 cubetas. Las claves son enteros (ignoramos los valores en este ejemplo). Supongamos que la función de dispersión devuelve el propio entero módulo la talla del vector y que insertamos los siguientes valores: 325, 12, 100, 30, 145, 89, 75, 237.

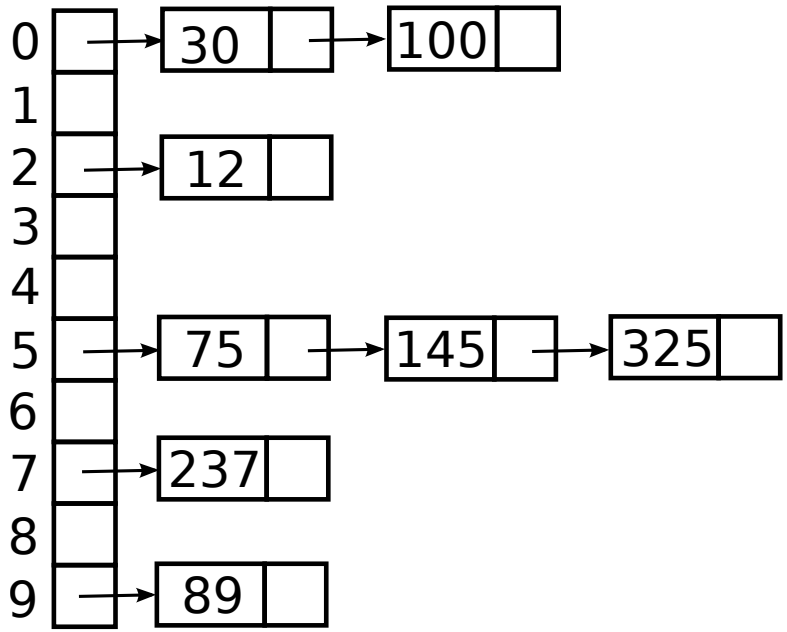


Figura : Ejemplo de tabla de dispersión

Factor de carga

- El **factor de carga** *(load factor)* se define como el número de elementos dividido entre el número de cubetas.
- La interpretación del factor de carga depende del tipo de tabla:
  - **Encadenamiento**: el factor de carga corresponde a la longitud media de las listas asociadas a cada cubeta.

Implementación mediante listas directas

- La implementación básica que veremos en este tema se basa en **listas directas**. Definimos una clase genérica que almacene conjuntamente la clave y el valor de una entrada.
- En una tabla con resolución de colisiones por encadenamiento los pares *(clave,valor)* se guardan en una lista enlazada que puede ser simplemente enlazada:

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

Implementación: constructor

- El constructor recibe el número estimado de elementos a almacenar y reserva espacio para guardarlos con un factor de carga de 0,75 (imitando `HashMap` de `java.util.Collections`).
- Para muchas funciones de dispersión funciona mejor un número primo de cubetas.

```
public class TablaHash<C, V> implements Map<C, V> {
  // Array de listas de entradas
  private EntradaHash<C,V> elArray[];
  // Numero de datos almacenados en la tabla
  private int talla;
  @SuppressWarnings("unchecked")
  public TablaHash(int tallaMaximaEstimada) {
   int capacidad=siguientePrimo((int)(tallaMaximaEstimada
    /0.75));
   elArray = new EntradaHash[capacidad];
   talla = 0;
  }
```

Implementación: inserción

```
// Inserta la entrada (c,v) y devuelve el antiguo valor
// que tenia dicha clave (o null si no tenia valor)
public V insertar(C c, V v) {
  V antiguoValor = null;
  int pos = indiceHash(c);
  EntradaHash<C, V> e = elArray[pos];
  while (e != null && !e.clave.equals(c))
    e = e.siguiente;
  if (e == null) { // Nueva entrada
    elArray[pos] = new EntradaHash<C,V>(c, v,
    elArray[pos]);
    talla++;
  } else { // Entrada existente
    antiguoValor = e.valor;
    e.valor = v;
  }
  return antiguoValor;
}
```

Implementación: borrado

```
// Elimina la entrada con clave c y devuelve su valor
// asociado (o null si no esta esa clave)
public V eliminar(C c) {
  int pos = indiceHash(c);
  EntradaHash<C, V> e = elArray[pos], ant = null;
  while (e != null && !e.clave.equals(c)) {
    ant = e;
    e = e.siguiente;
  }
  if (e == null) return null; // No encontrado
  if (ant == null)
    elArray[pos]  = e.siguiente;
  else
    ant.siguiente = e.siguiente;
  talla--;
  return e.valor;
}
```

Implementación: recuperar, esVacio, 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) {
  EntradaHash<C, V> e = elArray[indiceHash(c)];
  while (e != null && !e.clave.equals(c))
    e = e.siguiente;
  if (e == null) return null;
  return e.valor;
}

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

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

Implementación alternativa: listas con PI

Hemos visto una implementación basada en listas directas. También se pueden utilizar listas con punto de interés:

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

La clase `EntradaHash` ya no lleva una referencia a siguiente. Esta versión también sirve para tablas con direccionamiento abierto.

Implementación listas con PI: constructor

- El constructor recibe el número estimado de elementos a almacenar y reserva espacio para guradarlos con un FC=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> {
  private ListaConPI<EntradaHash<C,V>> elArray[];
  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;
}
```

Implementación listas con PI: búsqueda posición

```
// Calcula la cubeta en la que debe estar un elemento
// con clave c. Para ello primero obtiene el valor de
// hash (hashCode) y a continuacion su indice 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;
}
```

Implementación listas con PI: inserción

```
// Anyade la entrada (c,v) y devuelve el antiguo valor que
// tenia dicha clave (o null si no tenia 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 esta, insertamos entrada
    cubeta.insertar(new EntradaHash<C,V>(c, v));
    talla++; // Haria falta rehashing si se excede el FC
  } else { // Si ya estaba actualizamos el valor
    antiguoValor = cubeta.recuperar().valor;
    cubeta.recuperar().valor = v;
  }
  return antiguoValor;
}
```

Implementación listas con PI: borrado

```
// Elimina la entrada con clave c y devuelve su valor
// asociado o null si no hay ninguna entrada con esa clave
public V eliminar(C c) {
  int pos = indiceHash(c);
  ListaConPI<EntradaHash<C,V>> cubeta = elArray[pos];
  V valor = null;
  // 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 la encontramos la borramos
    valor = cubeta.recuperar().valor;
    cubeta.eliminar();
    talla--;
  }
  return valor;
}
```

Implementación listas con PI: recuperar, esVacio,

```
// 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];
  // Busqueda 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 esta vacio
public boolean esVacio() { return talla == 0; }

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

Funciones de dispersión

Función hash o de dispersión

- El objetivo de la función de dispersión es generar un entero entre `0` y `N-1` que cumpla ciertas propiedades.
- En Java existe una función estándar en la clase `Object` que se denomina `hashCode`:

```
public int hashCode() { ... }
```

- `hashCode` **no conoce el número de cubetas** `N`, por lo que devuelve un valor en el rango de los enteros (4 bytes) incluyendo potencialmente los negativos.
- El mètodo `hashCode` heredado de `Object` no siempre funciona o es adecuado en todas las subclases, por lo que normalmente se redefine y se utiliza para implementar la función que obtiene el índice del vector o la cubeta asociada al elemento.

Función de compresión

Para obtener el índice del vector asociado a un elemento a partir del valor calculado por `hashCode` aplicamos una función que convierte este valor al rango de `0` a `N-1` y que se conoce como **función de comprensión**.

Normalmente se basa en el módulo de la división. Es importante recordar que `hashCode` puede devolver negativos y que el operador `%` devuelve, en Java, el valor con el signo del dividendo, de ahí el `if` de esta implementación del método `indiceHash` de la clase `TablaHash`:

```
public int indiceHash(C c) {
  int valorHash  = c.hashCode();
  int indiceHash = valorHash % elArray.length;
  if (indiceHash<0) indiceHash += elArray.length;
  // 0 <= indiceHash < elArray.length
  return indiceHash;
}
```

Función hash o de dispersión

La función de dispersión debe cumplir algunas propiedades:

- **Obligatorio:** que sea determinista y que dé el mismo valor para claves que sean iguales de acuerdo con `equals`.
- **Inevitable:** es posible que dos valores diferentes terminen cayendo en la misma cubeta, esto se denomina una **colisión** y es inevitable si el número de claves posibles es mayor que el número de cubetas.
- **Deseable:** que sea rápida de evaluar y que todas las cubetas reciban aproximadamente el mismo número de elementos para equilibrarlas lo más posible: los índices de las claves que vamos insertando se debe parecer a sacar numeros entre `0` y `N-1` de manera equiprobable y con reemplazamiento.

Funciones de dispersión habituales

Algunas funciones de dispersión habituales son:

- Multiplicativa.
- Divisiva.
- Polinómica.

En Java, la función `hashCode` para la clase `String` se calcula como si los caracteres de la cadena fuesen los coeficientes de una función de dispersión polinomial en base 31 (se ha escogido este valor porque es primo). Para una cadena `s` de talla `n` se calcularía así (con la restricción de que tiene que caber en un entero):

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

¿Cómo está funcionando la tabla de dispersión?

¿Cuántos elementos hay de media por cubeta? o ¿cuántas colisiones se han producido?

- La media o **factor de carga** nos contesta esta pregunta, y se define como el número de elementos dividido entre el número de cubetas.

¿Cómo están repartidos los elementos entre las cubetas?

- La **varianza** nos dice como de próxima está la ocupación de todas las cubetas a la media.

Histograma de ocupación

Es una manera gráfica de ver lo bien dispersada que está una tabla. Indica cuántas cubetas tienen 0 elementos, 1 elemento, etc.

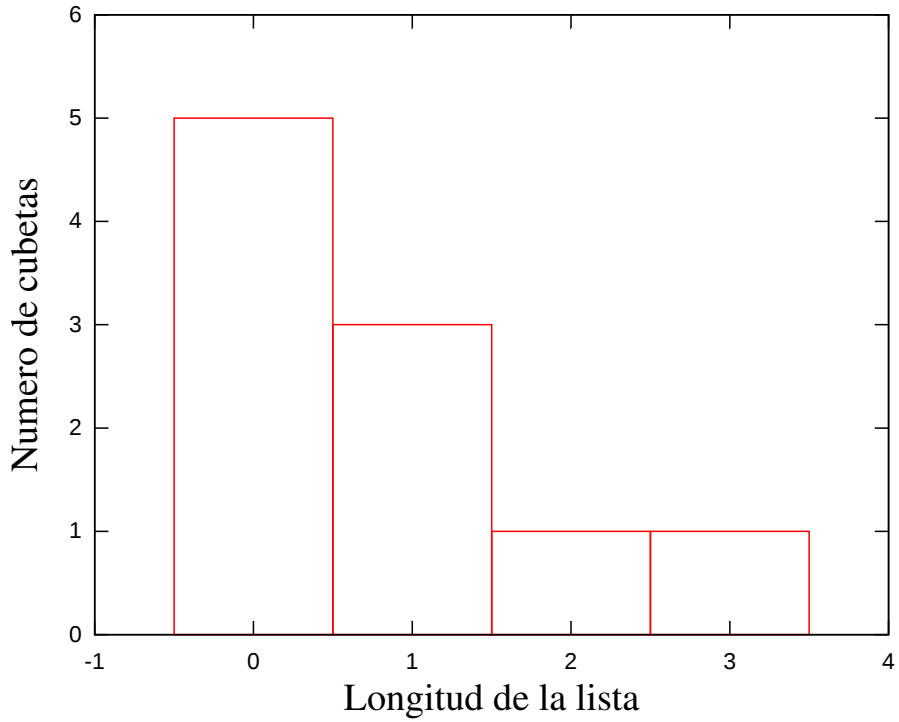


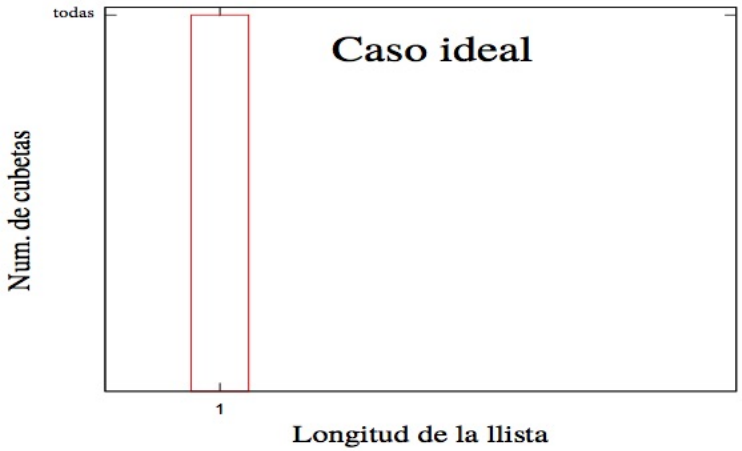
Figura : Histograma de ocupación del ejemplo anterior

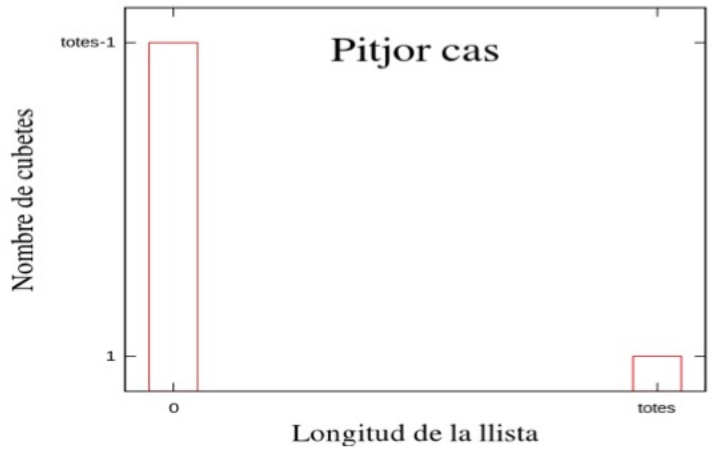
Histograma de ocupación

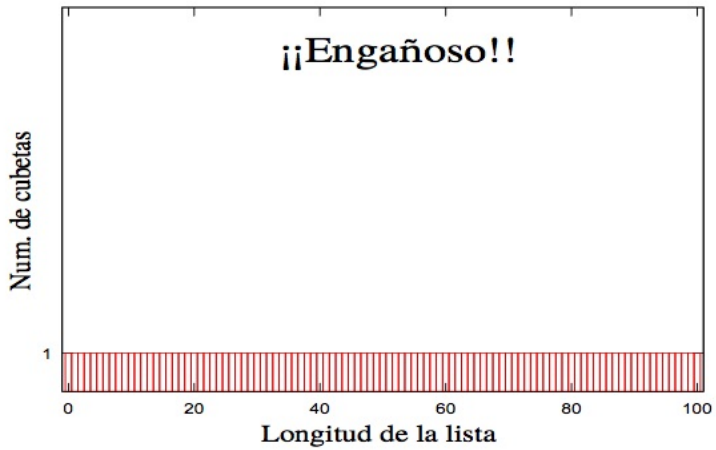
El histograma de ocupación permite ver "a golpe de vista" lo bien distribuidas que están las claves en una tabla de dispersión:

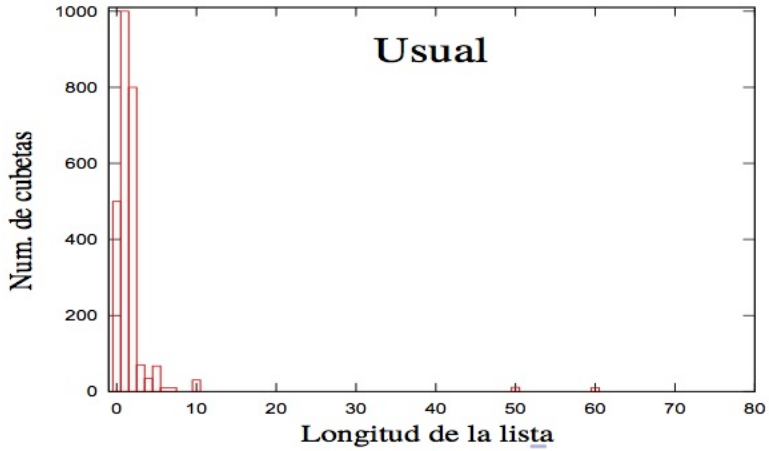
- **Ideal:** una tabla de dispersión ideal tendría la mayor parte de las listas de longitud cercana a la media (factor de carga). Su histograma de ocupación tendría un pico en la longitud de lista igual al factor de carga.
- **Malo:** una tabla de dispersión mal distribuida tendría listas de longitud mucho mayor que la media o factor de carga. Su histograma de ocupación tendría columnas mayor que cero en longitudes mucho mayor que el factor de carga, cosa que no necesariamente corresponde a columnas de altura elevada.

Histograma de ocupación









Dispersión perfecta (perfect hashing)

Algunas aplicaciones utilizan **diccionarios que no se modifican**:

> *Un GPS lleva un mapa con un diccionario que asocia a cada ciudad sus coordenadas geográficas.*

No se trata del interfaz Map visto en la introducción, ya que un diccionario estático no permite inserciones ni borrados.

Es posible diseñar una función de dispersión *ad hoc* para el conjunto de claves conocido a priori de modo que no se produzcan colisiones.

Esto se conoce como perfect hashing y permite implementaciones muy eficientes de diccionarios estáticos.

Costes y redispersión

Costes

- El **coste espacial** de una tabla con M elementos y N cubetas es $O(N + M)$.
- El **coste temporal** de insertar y de buscar viene dado por el coste de la función de dispersión seguida por el coste de buscar en la lista de la cubeta correspondiente. Este coste depende de la longitud de la lista ponderado por el coste del método `equals`:
  - El coste *promedio* de buscar en la tabla se corresponde con el factor de carga.
  - El *caso peor* depende de la longitud más larga de las listas. Este valor puede ser mucho mayor que el valor medio si la longitud de las listas tiene mucha varianza.
- **Para un número de cubetas fijo, el coste asintótico** de insertar/buscar/borrar crece de manera **lineal** con el número de elementos, si bien con una constante $1/N$ muy bajita.

Costes y dispersión

El **coste amortizado** es el coste medio de un conjunto de operaciones.

> ***Ejemplo:*** *si solamente vas a realizar un trayecto en metro, el coste de un billete sencillo es menor que el coste de un bono, pero si tienes en cuenta que vas a realizar muchos viajes, llega un momento en que compensa comprar el bono.*

De manera similar, queremos que el coste promedio de una serie de inserciones en una tabla de dispersión sea en promedio constante (que el coste total de M inserciones sea $O(M)$). Para ello el número de cubetas ha de aumentar cuando el factor de carga supere un límite, aunque ese aumento sea **puntualmente** caro.

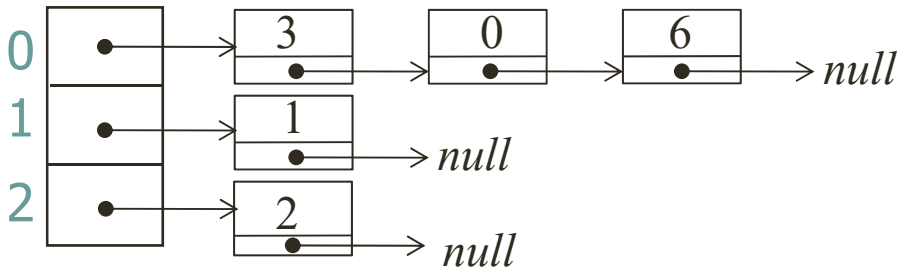
La redispersión (o "rehashing")

Consiste en modificar el número de cubetas. Este proceso, para pasar de N1 a N2 cubetas, se divide en 3 fases *(este método es un ejercicio de prácticas)*:

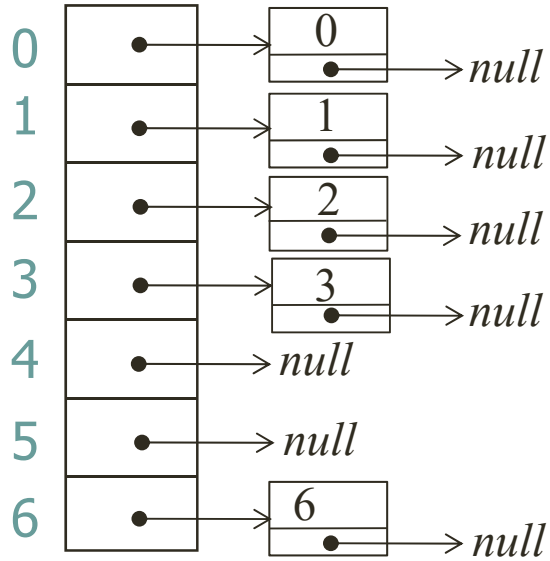
1. Crear una tabla nueva de talla N2 sin perder (todavía) la referencia a la tabla anterior.
2. Trasladar cada uno de los M nodos de las listas enlazadas de las cubetas antiguas a las nuevas. En general, los nodos van a cambiar de cubeta y hay que volver a calcular el hashCode. Al insertar en la cubeta nueva **no hace falta** comprobar que el elemento a insertar ya existe, como ocurre con una inserción convencional.
3. La tabla original se ha quedado sin cubetas, dejamos de referenciarla para que el recolector de basura la pueda eliminar.

La redispersión o "rehashing"

FC = 5/3 = 1.67



FC = 5/7 = 0.71



Bibliografía

Bibliografía

- *Data structures, algorithms, and applications in Java* (Sahni, Sartaj) Universities Press.
  - Capítulo 11, apartados 1 y 5 (excepto 11.5.3)
- *Estructuras de Datos en Java.* Weiss, M.A. Adisson-Wesley, 2000.
  - Capítulo 6, apartado 7, y capítulo 19.
- *Data structures and algorithms in Java* (Goodrich, Michael T.). John Wiley & Sons, Inc.
  - Capítulo 9, apartados 1 y 2.