TEMA 5: MAPAS

Dado que la información almacenada puede llegar a ser muy compleja y difícilmente "organizable" surgen los mapas, que nos permiten organizar la información usando una clave para acceder a la información, donde la inserción, búsqueda y borrado de la información se hace a través de las claves.

Un mapa es un conjunto finito de entradas (clave-valor) donde todas las entradas tienen **distinta clave**. El requisito para poder implementar un **Map<K, V>** es poder establecer una relación de igualdad entre dos elementos de tipo K. Lo habitual es utilizar el método `equals` de K para comparar las claves. Una implementación "sencilla" sería con una lista de entradas <K, V> (pares) sin claves repetidas. Su **interfaz** es la que sigue:

```
public interface Map <K,V> {
  public int size();
  public boolean isEmpty();
  public V put(K key, V value) throws InvalidKeyException;
  public V get(K key) throws InvalidKeyException;
  public boolean containsKey(Object key) throws InvalidKeyException;
  public V remove(K key) throws InvalidKeyException;
  public Iterable<K> keys();
  public Iterable<Entry <K,V>> entries();
}
```

- El método `put(key,value)` añade una nueva entrada en la tabla con clave `key`. En caso de que la entrada ya exista en el Map, se reemplaza el valor almacenado para la entrada key por value y devuelve el valor que hubiera almacenado para key si ya existía previamente o null en caso de que se haya creado una nueva entrada.
- El método `get(key)` devuelve el valor almacenado en la tabla para la entrada key (si la entrada no existe, devuelve null)
- El método `containsKey(key)` devuelve true si la clave key tiene entrada en el Map y false en caso contrario
- El método `remove(key)` elimina de la tabla la información y la entrada key. Devuelve el valor almacenado en key o null si no existía
- Los métodos `keys()` y `entries()` devuelven un objeto iterable para recorrer las claves o los pares clave-valor

NOTA1: Los métodos get, put y remove lanzan InvalidKeyException si la clave no es válida: p.e. null
NOTA2: Si map.get(k) devuelve null, puede ser porque no hay ninguna entrada con la clave k o porque el valor asociado con la clave k es null (es decir, hemos ejecutado map.put(k,null) antes)
NOTA3: Se puede usar map.containsKey(k) para determinar si una entrada con la clave k realmente existe

Interfaz `Entry`

```
public interface Entry<K,V> {
  public K getKey();
  public V getValue();
}
```

Básicamente es un `Pair<K,V>` excepto no hay "setters" `setKey` y `setValue`

Ejemplos de uso de Maps:

```
Map <Character,Integer> map = new HashTableMap <...>() ; // key = char, value = int
map.put('a', 5); // El map tiene [<a ,5 >]
map.put('b', 8); // El map tiene [<a,5>, <b ,8 >]
map.put('c', 7); // El map tiene [<a,5>, <b,8>, <c ,7 >]
map.put('a', 10); // [<a ,10>, <b,8>, <c ,7 >]
System.out.println(map.get('a')); // 10
System.out.println(map.get('b')); // 8
System.out.println(map.get('c')); // 7
map.remove('c'); // El map tiene [<a,5>, <b,8>]
System.out.println(map.get('c')); // null
System.out.println(map.get('d')); // null
map.put('d', null); // El map tiene [<a,5>, <b,8>,<d,null>]
System.out.println(map.get('d')); // null

// Recorremos Claves
Iterator<Character> itk = map.keys().iterator();
while (itk.hasNext())
  System.out.println(k + " " + map.get(itk.next()));

for(Character c: map.keys())
System.out.println(k + " " + map.get(c));

// Recorremos entradas
Iterator<Entry<Character,Integer>> ite = map.entries().iterator();
while(ite.hasNext()) {
  Entry<Character,Integer> e = it.next();
  System.out.println(e.getKey() + "-" + e.getValue());
}

for(Entry<Character,Integer> e: map.entries())
  System.out.println(e.getKey() + "-" + e.getValue());
```

Tablas Hash

El objetivo es implementar un Map cuyas operaciones tengan complejidad **O(1)**. Para ello, las claves deben ser dispersables. Se utiliza una función de codificación o función hash cuyo objetivo es, dada una clave, devolver un valor numérico dentro del rango [0 … K-1] utilizando una función de compresión/dispersión para que todos los posibles hash codes se compriman y distribuyan al rango [0 … N-1] usando un array de tamaño N como tabla de dispersión.

Dada una función hash que codifica en un rango [0..K - 1] y un objeto key como clave y dada una función de compresión/dispersión que trabaja en un rango [0 … N - 1], usamos un array arr de tamaño N para almacenar la tabla. La secuencia de operaciones sería:

1. Dado un objeto o obtenemos su hash code (p.e. m etodo hashCode de Object)
2. Aplicamos la función de compresión/dispersión (comprimir) sobre el hash code obtenido
3. La entrada arr[comprimir(o.hashCode)] almacenará la información referente al objeto o

La obtención de hashCode debe tener complejidad O(1)
La aplicación de la función comprimir también debe ser O(1)
Los accesos a arr[i] tienen complejidad O(1)

Implementación de los métodos `equals` y `hashCode`. Es necesario que haya coherencia entre equals y hashCode
Dados o1 y o2 usados como claves puede ocurrir:

- o1.hashCode()!=o2.hashCode() && !o1.equals(o2)
  OK. Son dos objetos distintos, dos entradas distintas en el map
- o1.hashCode()==o2.hashCode() && o1.equals(o2)
  OK. Una unica entrada en el map
- o1.hashCode()==o2.hashCode() && !o1.equals(o2)
  Colisión. No hay problema, dos entradas distinta en el map
- o1.hashCode()!=o2.hashCode() && o1.equals(o2)
  Inconsistente. Si los objetos son iguales deberían tener el mismo hashCode
  Se producen dos entradas distintas en el map, teniendo dos objetos iguales (de acuerdo al resultado de equals)

Si el objeto implementa Comparable, también se debe conservar la coherencia con compareTo

Por ejemplo, si tenemos un array de tamaño 2 → map.put(9,"Hola"); map.put(5,"Adios");

Colisiones

Cuando obtenemos la misma posición del array para dos objetos diferentes (ya sea porque los hashcode coinciden, o porque la función de compresión devuelve la misma dirección aunque tengamos dos hashcode distintos), decimos que se ha producido una colisión. Esto se puede solucionar de varias formas diferentes:

- **Separate Chaining:** Cada uno de los elementos del array son listas y usar equals para buscar el elemento. Se puede usar un Map implementado con una lista.

- Técnicas de **open addressing:**
  - **Linear Probing:** Si la posición del array está ocupada, uso la siguiente posición (circularmente)
  - **Quadratic Probing:** Si hay colisión, calcula la siguiente opción usando una función $(i + j^2)$ mod N con j = 0, 1, 2…
  - **Double Hashing:** Si hay colisión, calcula la siguiente opción usando una función (i + otrohash(j)) mod N con j = 0, 1, 2…