Algoritmos y Estructuras de Datos

Maps y Tablas Hash

Guillermo Román Díez
groman@fi.upm.es

Lars-Åke Fredlund
lfredlund@fi.upm.es

Universidad Politécnica de Madrid

Curso 2021/2022

Motivación

- La información almacenada puede ser compleja y difícilmente "organizable"

Motivación

- La información almacenada puede ser compleja y difícilmente "organizable"
- Los **maps** permiten *organizar* la información usando una **clave** para *acceder* a la información
  - La inserción, la búsqueda y el borrado de información se harán mediante las claves

Motivación

- La información almacenada puede ser compleja y difícilmente "organizable"
- Los **maps** permiten *organizar* la información usando una **clave** para *acceder* a la información
  - La inserción, la búsqueda y el borrado de información se harán mediante las claves
- Un **map** es un conjunto finito de entradas (clave-valor) donde todas las entradas tienen distinta clave
- Un **map** es una *función parcial* que dado un tipo el tipo de las claves al tipo de los valores:

$$TipoClave \nrightarrow TipoValor$$

Motivación

- La información almacenada puede ser compleja y difícilmente "organizable"
- Los **maps** permiten *organizar* la información usando una **clave** para *acceder* a la información
  - La inserción, la búsqueda y el borrado de información se harán mediante las claves
- Un **map** es un conjunto finito de entradas (clave-valor) donde todas las entradas tienen distinta clave
- Un **map** es una *función parcial* que dado un tipo el tipo de las claves al tipo de los valores:

$$TipoClave \nrightarrow TipoValor$$

- A nivel de implementación, se podrá utilizar alguna propiedad de las claves que permitan *organizar* la información de forma eficiente

Maps

- 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

Maps

- 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
- Sería posible utilizar otros métodos, pero habitualemente no se usan
  - Si `K` implementa `Comparable<K>` se podría usar `compareTo`
  - Si se dispone de un `Comparator<K>` se podría usar el método `compare` del comparador

Maps

- 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
- Sería posible utilizar otros métodos, pero habitualemente no se usan
  - Si `K` implementa `Comparable<K>` se podría usar `compareTo`
  - Si se dispone de un `Comparator<K>` se podría usar el método `compare` del comparador
- Una implementación "sencilla" sería con una lista de entradas `<K,V>` (pares) sin claves repetidas

Interfaz `Map`

```
public interface Map<K,V> extends Iterable<Entry<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();
}
```

Interfaz `Map`

- 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`
  - 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
- `keys()` y `entries()` devuelven un objeto iterable para recorrer las claves o los pares clave-valor

Interfaz `Map`

- Los métodos `get`, `put` y `remove` lanzan `InvalidKeyException` si la clave no es válida: p.e. `null`
- Notad que si `map.get(k)` devuelve `null`, puede ser porque:
  - no hay ninguna entrada con la clave `k`
  - o, el valor asociado con la clave `k` es `null`. Es decir, hemos ejecutado `map.put(k,null)` antes
- 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 un `Pair<K,V>` excepto no hay "setters" `setKey` y `setValue`

Interfaz `Entry`

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

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

Pregunta

¿por qué no hay "setters"?

Interfaz `Entry`

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

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

Pregunta

¿por qué no hay "setters"?

- Cambiar la clave podría ser difícil si el map está ordenado según la clave – hay que borrar el entry del map y después reinsertarlo

Ejemplo de uso Map

```
Map<Character,Integer> map = new HashTableMap<...>();

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');
System.out.println(map.get('c')); // null

System.out.println(map.get('d')); // null
map.put('d', null);
System.out.println(map.get('d')); // null
```

Ejemplo de uso Map

```
// 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());
}
```

Implementaciones de `Map`

- Una posible implementación sería con una lista no ordenada de elementos de tipo `<K,V>`

Implementaciones de `Map`

- Una posible implementación sería con una lista no ordenada de elementos de tipo `<K,V>`
- Todos los métodos necesitan buscar si el elemento existe, por tanto la complejidad es $O(n)$

`get`	$O(n)$
`put`	$O(n)$
`remove`	$O(n)$

Implementaciones de `Map`

- Una posible implementación sería con una lista no ordenada de elementos de tipo `<K,V>`
- Todos los métodos necesitan buscar si el elemento existe, por tanto la complejidad es $O(n)$

`get`	$O(n)$
`put`	$O(n)$
`remove`	$O(n)$

Pregunta

¿y no se puede mejorar esto?

Implementaciones de `Map`

- Una posible implementación sería con una lista no ordenada de elementos de tipo `<K,V>`
- Todos los métodos necesitan buscar si el elemento existe, por tanto la complejidad es $O(n)$

`get`	$O(n)$
`put`	$O(n)$
`remove`	$O(n)$

Pregunta

¿y no se puede mejorar esto?

- Sí, si las claves son *comparables* – $O(log\ n)$

Implementaciones de `Map`

- Una posible implementación sería con una lista no ordenada de elementos de tipo `<K,V>`
- Todos los métodos necesitan buscar si el elemento existe, por tanto la complejidad es $O(n)$

`get`	$O(n)$
`put`	$O(n)$
`remove`	$O(n)$

Pregunta

¿y no se puede mejorar esto?

- Sí, si las claves son *comparables* – $O(log\ n)$
- Sí, usando funciones finitas y tablas de dispersión – $O(1)$ (coste medio)

Tablas Hash

- El objetivo es implementar un `Map` cuyas operaciones tengan en complejidad $O(1)$
- Para ello, las claves tienen que ser *dispersables*

Tablas Hash

- El objetivo es implementar un `Map` cuyas operaciones tengan en complejidad $O(1)$
- Para ello, las claves tienen que ser *dispersables*
- Utilizaremos una **función de codificación** o **función hash**
  - El objetivo de la *función hash* es, dada una clave, devolver un valor numérico dentro de un determinado rango $[0..K-1]$

Tablas Hash

- El objetivo es implementar un `Map` cuyas operaciones tengan en complejidad $O(1)$
- Para ello, las claves tienen que ser *dispersables*
- Utilizaremos una **función de codificación** o **función hash**
  - El objetivo de la *función hash* es, dada una clave, devolver un valor numérico dentro de un determinado rango $[0..K-1]$
- Se utilizará una **función de compresión/dispersión** para que todos los posibles *hash codes* se compriman y distribuyan al rango $[0..N-1]$

Tablas Hash

- El objetivo es implementar un `Map` cuyas operaciones tengan en complejidad $O(1)$
- Para ello, las claves tienen que ser *dispersables*
- Utilizaremos una **función de codificación** o **función hash**
  - El objetivo de la *función hash* es, dada una clave, devolver un valor numérico dentro de un determinado rango $[0..K-1]$
- Se utilizará una **función de compresión/dispersión** para que todos los posibles *hash codes* se compriman y distribuyan al rango $[0..N-1]$
- Usaremos un array de tamaño N como **tabla de dispersión**

Tablas Hash

- Dada una función hash que codifica en un rango $[0..K-1]$ y un objeto `key` como clave
- Dada una función de compresión/dispersión que trabaja en un rango $[0..N-1]$
- Usaremos 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. método `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

Tablas Hash

- Dada una función hash que codifica en un rango $[0..K-1]$ y un objeto `key` como clave
- Dada una función de compresión/dispersión que trabaja en un rango $[0..N-1]$
- Usaremos 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. método `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 $O(1)$

Colisiones

Pregunta

¿podemos obtener la misma posición del array para dos objetos distintos?

Colisiones

Pregunta

¿podemos obtener la misma posición del array para dos objetos distintos?

- Sí. Decimos que se ha producido una **colisión**

Colisiones

Pregunta

¿podemos obtener la misma posición del array para dos objetos distintos?

- Sí. Decimos que se ha producido una **colisión**

Pregunta

¿cómo se puede producir una colisión?

Colisiones

Pregunta

¿podemos obtener la misma posición del array para dos objetos distintos?

- Sí. Decimos que se ha producido una **colisión**

Pregunta

¿cómo se puede producir una colisión?

- Si el `hashCode` de dos objetos es el mismo
- Si la función de compresión devuelve la misma dirección aunque tengamos dos *hash code* distintos

Implementando `equals` y `hashCode`

- Es necesario que haya *coherencia* entre `equals` y `hashCode`
- Dados `o1` y `o2` usados como claves puede ocurrir:

Implementando `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)`

Implementando `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

Implementando `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)`

Implementando `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 única entrada en el map

Implementando `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 única entrada en el map
  - `o1.hashCode()==o2.hashCode() && !o1.equals(o2)`

Implementando `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 única entrada en el map
  - `o1.hashCode()==o2.hashCode() && !o1.equals(o2)`
    - ★ Colisión. No hay problema, dos entradas distinta en el map

Implementando `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 única 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)`

Implementando `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 única 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`)

Implementando `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 única 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`

Algunas funciones de codificación

NOTA!!

El objetivo fundamental de la *función de codificación* es que se generen las menores colisiones posibles

- El método `hashCode` de `Object` devuelve el `int` que representa la dirección de memoria de un objeto
- Si el tamaño del dato es $\leq 32 bits$ (`byte`, `char`,...) podemos hacer un casting a `int`
- Si el tamaño es $> 32 bits$ (`long`, `double`) se puede hacer la "suma binaria de las dos palabras que lo componen"
- Se puede hacer un *Polynomial hash code* (p.e. `String` de Java):

$$s[0] * 31^{n-1} + s[1] * 31^{n-2} + ... + s[n-1]$$

  donde $s[i]$ es la letra i del String s, n es la longitud de s, y a^b es exponenciación
- Desplazamiento binario cíclico *(cyclic shift)*

Algunas Funciones de compresión/dispersión

NOTA!!

El objetivo fundamental de una *función de compresión/dispersión* es que se generen las menores colisiones posibles

- La función de compresión debe pasar del rango $i \in [0..K]$ devuelto por `hashCode` a un rango $[0..N]$ que es el tamaño del array
- La más sencilla sería el *"division method"*:

$$i \ mod \ N$$

- El método *MAD* (*multiply-add-and-divide*):

$$[(a \cdot i + b) mod \ p] \ mod \ N$$

  - $p > N$ y p es primo
  - a y b son números aleatorios en $[0..p-1]$ y $a > 0$

Solucionando las colisiones

- **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 $j = 0, 1, 2 \ldots$
  - **Double Hashing**: Si hay colisión, calcula la siguiente opción usando una función $(i + otrohash(j))\ mod\ N$ con $j = 0, 1, 2 \ldots$