FAQ – v.7
Ejercicio 3 – Preguntas frecuentes

josé a. mañas

17.3.2016

1 ¿Qué hay que hacer?

1. Implementar un diccionario usando tablas hash con desbordamiento en listas
2. Preparar pruebas y probar hasta poner a punto
3. Medir tiempos de la operación get() en dos circunstancias: diccionario con pocos y con muchos datos

2 Casos de prueba: HashListasTest

Se emplea JUnit.

Se puede partir de los casos de prueba del ejercicio 1.

Hay que añadir casos de prueba con cargas por debajo y por encima del número de ranuras.

Una estrategia puede ser, para un número de ranura NS,

- vamos metiendo pares (clave, valor) con claves distintas y valores diferentes para cada clave, hasta llegar a 2×NS
- tras cada put validamos que
    - la clave queda metida y asociada a su valor singular
    - el número de datos progresa adecuadamente
- repetimos la carga para asegurarnos de que los publicados se tratan adecuadamente, con nuevos valores asociados a las mismas claves
- vamos vaciando la tabla, asegurándonos de que las claves van desapareciendo y que la cuenta va bien

También hay que asegurarse de que clear() elimina una tabla con más datos que ranuras.

Lo que hay que evitar es que algún día alguien use nuestro diccionario en serio, tenga problemas y lo único que podamos hacer sea pedir perdón:

> *"Es que ese caso no lo habíamos probado."*

3 ¿Cómo se implementa HashListas?

Se necesita una tabla de ranuras inicializada con el número indicado en el constructor.

Opción 1. Se pueden poner listas vacías en todas las ranuras desde el principio. Es poco eficiente en espacio y ralentiza el constructor, pero funciona. En este caso, cree listas con reserva de espacio 0

```
new ArrayList<CV>(0);
```

Si no se indica la capacidad, java reserva espacio para 10, lo que es claramente costoso.

Opción 2. Se dejan las ranuras a null y se crean listas cuando hacen falta. Es más eficiente en espacio y el constructor va deprisa.

NOTA Usar ArrayList como listas de desbordamiento es costoso porque java reserva espacio al menos para 10 entradas y va creciendo en incrementos del 50% para ajustarse. Aparte de ese detalle, es una estructura de datos eficaz.

Como función hash puede usar la de java: hashCode().

3.1 Método get()

Primero determine en qué ranura estaría y luego busque en la lista.

Para buscar en la lista use el método getClave() de los objetos CV que se almacenan. Ese valor se compara con la clave buscada.

3.2 Método put()

Primero determine en qué ranura estaría y luego haga put() en la lista.

Ojo con los duplicados:

recorra la lista

- si encuentra una entrada con la clave, cámbiele el valor y acabe (return)
- si llega al final de la lista sin encontrar la clave, cree el objeto CV y añádalo a la lista

Para buscar en la lista use el método getClave() de los objetos CV que se almacenan. Ese valor se compara con la clave buscada.

3.3 Método remove()

Primero determine en qué ranura estaría. Luego busque en la lista y, si está, lo elimina.

Para buscar en la lista use el método getClave() de los objetos CV que se almacenan. Ese valor se compara con la clave buscada.

4 Medidas

4.1 Banco de pruebas

Necesita un banco de pruebas como el que se propone en

http://www.dit.upm.es/~pepe/doc/adsw/tema2/Meter1Ops.java

debiendo adaptarlo a su código y a los datos de medición que se recomiendan en el enunciado del ejercicio.

ND (número de datos): 5.000

NS (número de ranuras): 500, 1.000, 1.500, 2.000, 2.500, …, 20.000

O sea, de 500 en 500, desde 500 hasta 20.000.

4.2 Código preparado para medir

Debe tunear su código para que las llamadas a compareTo() en el método get() llamen a OpMeter.

O sea, donde pone

```
int cmp = clave.compareTo(datos[m].getClave());
```

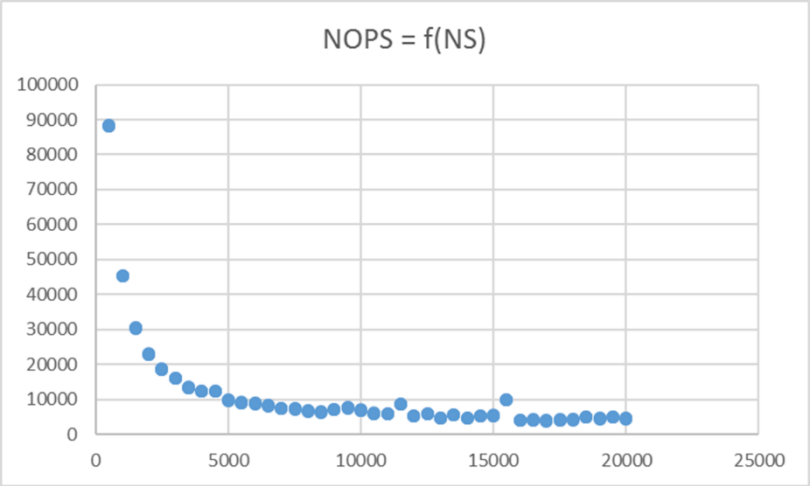
pasar a

```
int cmp = OpMeter.compareTo(clave, datos[m].getClave());
```

4.3 Medidas

Dibuje el número de operaciones de comparación en función del número de ranuras.

Se espera algo así

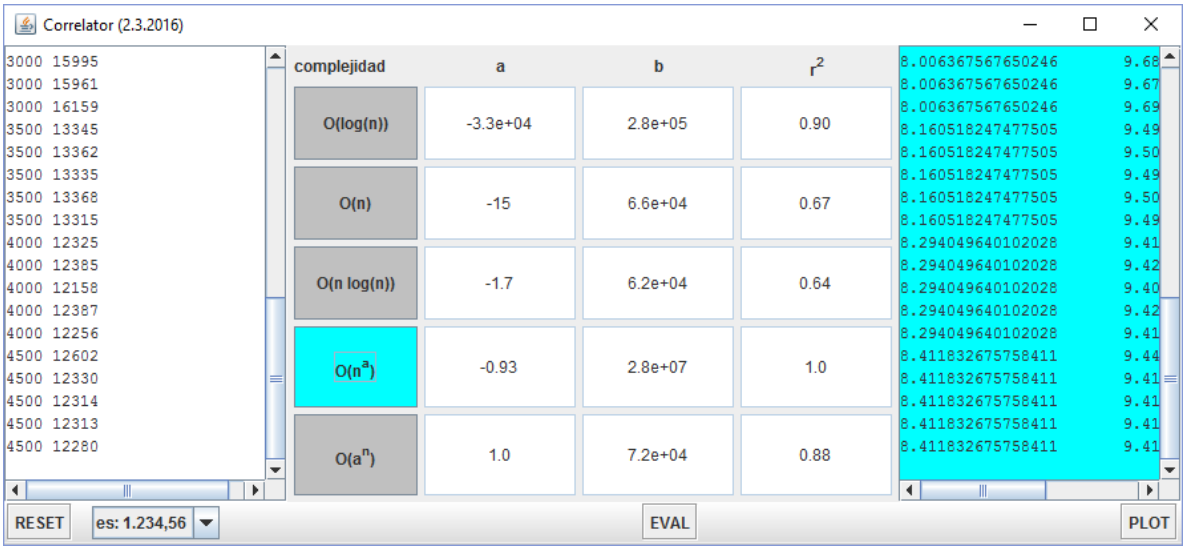


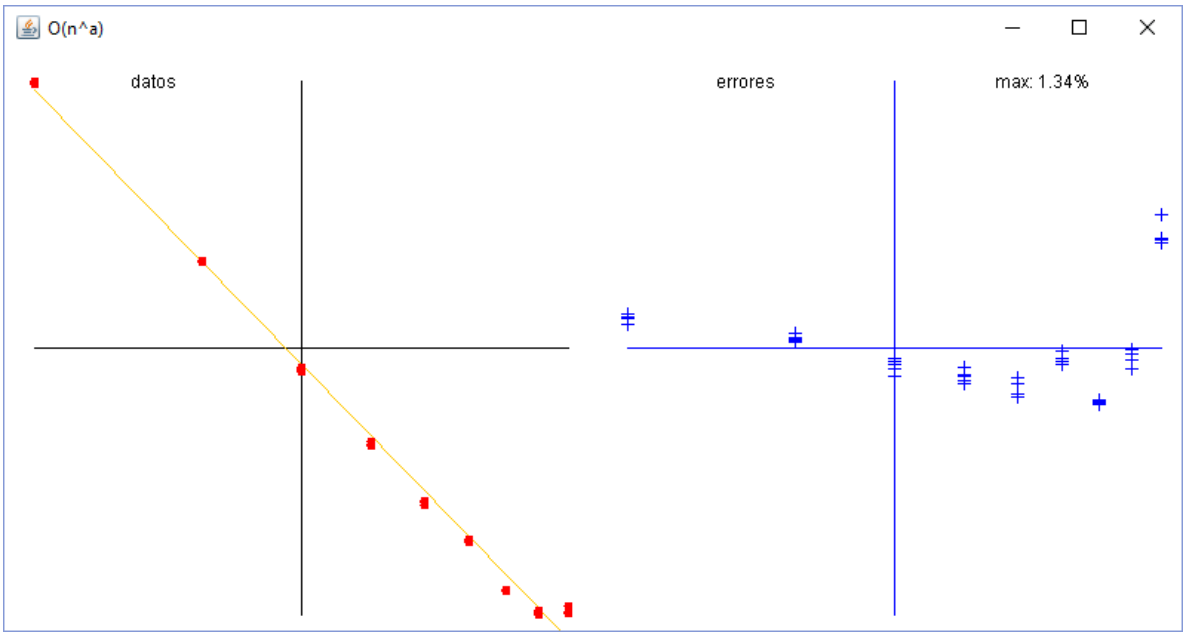
4.4 Medidas en la zona de alta carga

Cuando hay muchos más datos que ranuras: NS << ND.

Use el correlator para calcular la relación entre el número de operaciones y el número de ranuras.

Se espera algo así:





4.5 Medidas en la zona de baja carga

Mida para NS >> ND.

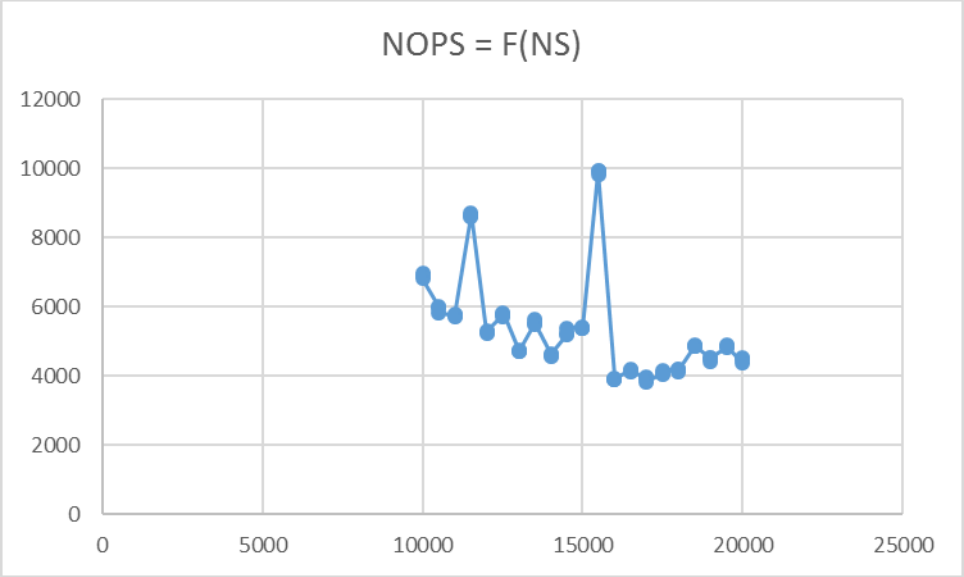
Dibuje el número de operaciones en función del número de ranuras. Se espera algo uniforme, O(1).

5 Entrega

- HashListas.java
- HashListasTest.java
- Gráficas en PDF

6 ¿por qué hay un pico en 15.500?

En las series se aprecia un pico de operaciones de comparación para un número de 15.500 ranuras. Eso indica que para ese valor los datos no se distribuyen uniformemente; o, en otras palabras, que hay muchas colisiones. Esta anomalía se repite en todos los ejercicios entregados, así que hay algo más que una casualidad.



La pregunta es ¿qué tiene 15.500 de especial?

La distribución se decide usando la fórmula

```
Math.abs(clave.hashCode()) % slots.length
```

Puede que la causa sea el uso de números demasiado 'redondos' como tamaño de la tabla. Veamos la descomposición en factores primos de los tamaños utilizados

N	factores
500	[2, 2, 5, 5, 5]
1000	[2, 2, 2, 5, 5, 5]
1500	[2, 2, 3, 5, 5, 5]
2000	[2, 2, 2, 2, 5, 5, 5]
2500	[2, 2, 5, 5, 5, 5]
3000	[2, 2, 2, 3, 5, 5, 5]
3500	[2, 2, 5, 5, 5, 7]
4000	[2, 2, 2, 2, 2, 5, 5, 5]
4500	[2, 2, 3, 3, 5, 5, 5]
5000	[2, 2, 2, 5, 5, 5, 5]
5500	[2, 2, 5, 5, 5, 11]
6000	[2, 2, 2, 2, 3, 5, 5, 5]
6500	[2, 2, 5, 5, 5, 13]
7000	[2, 2, 2, 5, 5, 5, 7]
7500	[2, 2, 3, 5, 5, 5, 5]

8000 [2, 2, 2, 2, 2, 2, 5, 5, 5]
8500 [2, 2, 5, 5, 5, 17]
9000 [2, 2, 2, 3, 3, 5, 5, 5]
9500 [2, 2, 5, 5, 5, 19]
10000 [2, 2, 2, 2, 5, 5, 5, 5]
10500 [2, 2, 3, 5, 5, 5, 7]
11000 [2, 2, 2, 5, 5, 5, 11]
11500 [2, 2, 5, 5, 5, 23]
12000 [2, 2, 2, 2, 2, 3, 5, 5, 5]
12500 [2, 2, 5, 5, 5, 5, 5]
13000 [2, 2, 2, 5, 5, 5, 13]
13500 [2, 2, 3, 3, 3, 5, 5, 5]
14000 [2, 2, 2, 2, 5, 5, 5, 7]
14500 [2, 2, 5, 5, 5, 29]
15000 [2, 2, 2, 3, 5, 5, 5, 5]
15500 [2, 2, 5, 5, 5, 31]
16000 [2, 2, 2, 2, 2, 2, 2, 5, 5, 5]
16500 [2, 2, 3, 5, 5, 5, 11]
17000 [2, 2, 2, 5, 5, 5, 17]
17500 [2, 2, 5, 5, 5, 5, 7]
18000 [2, 2, 2, 2, 3, 3, 5, 5, 5]
18500 [2, 2, 5, 5, 5, 37]
19000 [2, 2, 2, 5, 5, 5, 19]
19500 [2, 2, 3, 5, 5, 5, 13]
20000 [2, 2, 2, 2, 2, 5, 5, 5, 5]

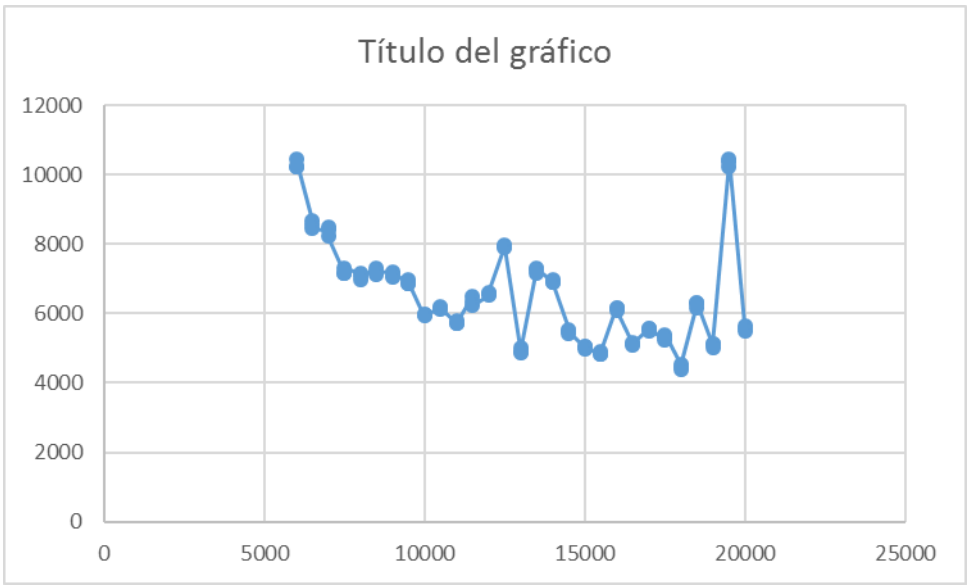
No parece que 15.500 tenga nada especial. Antes, al contrario: tiene pocos factores y uno de ellos es un primo elevado: 31.

Podemos comprobar qué pasa si alteramos un poco los valores. Concretamente, vamos a usar una entrada menos:

N factores
13999 [13999]
14499 [3, 3, 3, 3, 179]
14999 [53, 283]
15499 [11, 1409]
15999 [3, 5333]
16499 [7, 2357]
16999 [89, 191]
17499 [3, 19, 307]
17999 [41, 439]
18499 [13, 1423]
18999 [3, 3, 2111]
19499 [17, 31, 37]

19999 [7, 2857]

Volvemos a medir:

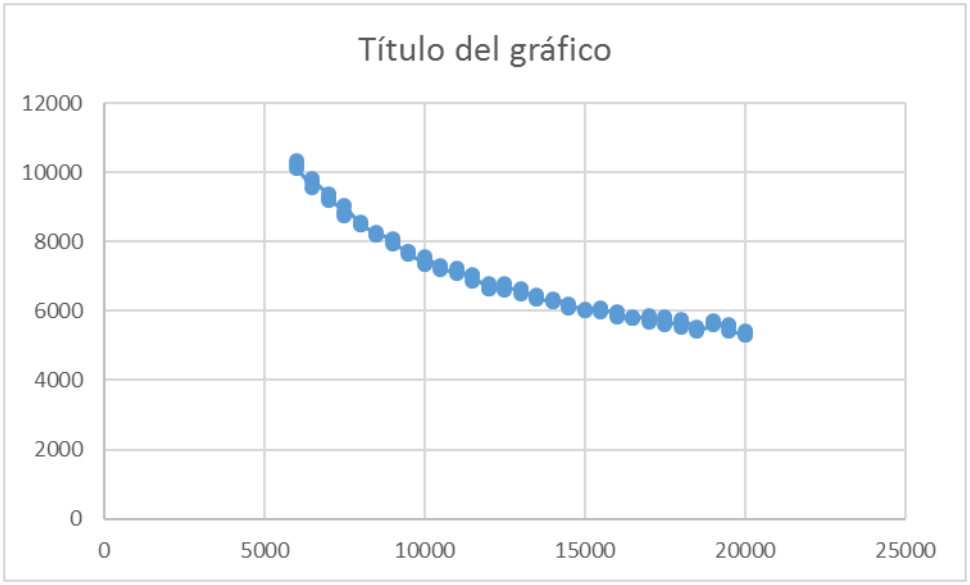


Parece que hemos desplazado el pico a otro sitio; pero sigue siendo un resultado anómalo: con tablas a las que les “tiene manía”.

Probemos con otra función hash:

```
Math.abs(MurmurHash.hash32(clave)) % slots.length
```

el resultado es:



Podemos concluir que la función hash murmur tiene un comportamiento más uniforme que el hashCode() nativo de java. En este escenario de pruebas.

6.1 ¿Por qué?

El problema reside en la función que usa java para calcular el hash.

```
int hashCode(String s) {
    int h = 0;
    char val[] = s.toCharArray();
    for (int i = 0; i < val.length; i++) {
        h = 31 * h + val[i];
    }
    return h;
}
```

en general

$$\text{hashCode}(s) = c_0 + c_1\ 31 + c_2\ 31^2 + \ldots + c_n\ 31^n$$

siendo c_0 el último carácter, c_1 el anterior, etc.

Consecuencia de ello es que para cualquier String

$$h = \text{hashCode}(s)\ \%\ 31 = c_0\ \%\ 31$$

Ahora, si calculamos el máximo común divisor de NS = 15.500

$$g = \text{mcd}(15500, 31) = 31$$

y cuando calculamos la posición en la tabla

$$pos = h\ \%\ NS$$

introducimos un patrón y es que siempre es cierto que

$$(h\ \%\ NS)\ \%\ g = h\ \%\ g$$

En la práctica, usamos Strings que son números. Eso implica que el carácter c_0 puede ser uno de los 10 dígitos: '0' .. '9'. En consecuencia, h%31 sólo puede tomar 10 de los 31 valores posibles, lo que deja 21 valores sin opción.

Cuando mcd(NS, 31) = 31, h%NS es un múltiplo de 31 y en una tabla de tamaño NS sólo vamos a usar 10 de cada 31 valores posibles, el 32%, condenando al 68% restante de las entradas a no ser usado nunca. O sea, que de una tabla de 15.500 entradas, sólo podemos usar 15.500 * 32% = 5.000.

Las medidas del ejercicio muestran esta situación.

6.2 Conclusiones

Si la función hash sigue algún patrón del tipo

$$hash\ \%\ x = cte$$

hay que buscar el máximo valor para NS/mcd(NS, x)

Una forma fácil de conseguirlo es que NS sea un número primo, de forma que

NS/mcd(NS, x) = NS / 1 = NS

independientemente de x.

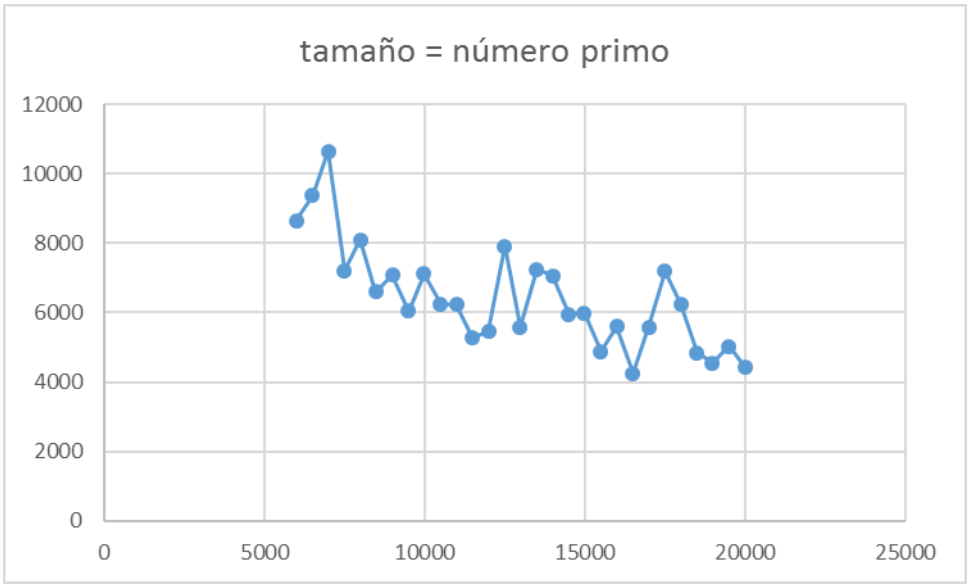
De esta forma, la tabla hash es inmune a ciertos patrones que pudiera tener un hash desafortunado.

Probando sobre nuestro ejercicio, cambiamos el número de ranuras para que use el número primo inmediato inferior a los números propuestos de medir

O sea

5987, 6491, 6997, 7499, 7993, 8467, 8999, 9497, 9973, 10499, 10993, 11497, 11987, 12497, …

El resultado es mejor, pero no perfecto



El problema remanente es que la función hashCode() es manifiestamente mejorable, al menos la empleada para Strings.

java.util.HashMap, consciente de este problema, manipula un poco el hashCode() nativo. Hay varias versiones; en java 8u40, hace esto

```
private int getIdx(String clave) {
    int h = clave.hashCode();
    int hh = h ^ (h >>> 16);
    return Math.abs(hh) % slots.length;
}
```

con este resultado (trabajando con talas de tamaño un número primo):