Aclaraciones trabajo práctico n.º 2

Programación II

Segundo cuatrimestre 2016

1 Añadir JUnit a un proyecto en Eclipse

Los casos de prueba proporcionados para la clase *Deque* están en formato JUnit[1], una herramienta estándar de Java para automatizar la validación de software.

El archivo *TestDeque.java* se puede ejecutar normalmente una vez se añade la biblioteca JUnit4 al proyecto de Eclipse.

Todas las instalaciones de Eclipse incluyen una copia de la biblioteca, y el menú para agregarla se encuentra en: `Project -> Build Path -> Add libraries -> JUnit -> Version 4`.

2 Introducción a diccionarios en Java

Un diccionario es un tipo abstracto de datos que permite asociar parejas de elementos, esto es: dado un elemento a_1, podemos asociarle un valor b_1. Después podremos realizar la pregunta: ¿qué valor quedó asociado con a_1?—y obtener b_1 como respuesta.

Conceptualmente, se puede concebir el diccionario como un conjunto de tuplas (a_i, b_i) con las siguientes dos restricciones:

1. no puede haber dos tuplas t_i y t_j tal que $a_i = a_j$.
2. a la hora de averiguar valores asociados, la pregunta se realiza por el *primer* elemento de la tupla; no es posible realizarla por el segundo.

En otras palabras, se pregunta solo sobre el primer elemento, y este primer elemento no puede estar repetido.

2.1 Clave y significado

Usando la terminología propia de diccionarios, al primer elemento de la tupla se le denomina *clave* y al segundo *valor* o *significado*.

[1] https://es.wikipedia.org/wiki/JUnit

Por tanto, un diccionario genérico usa dos tipos paramétricos K y V: uno para las claves, K, y otro para los significados, V.

2.2 Interfaz de Java

El diccionario genérico se representa en Java mediante la interfaz *Map<C, S>* (clave, significado) o *Map<K, V>* (*key* y *value* en inglés).

Por ejemplo, `Map<String, Double>` podría servir para asociar a cada estudiante por su nombre una la nota númerica; y `Map<Integer, Persona>` para asociar al DNI como entero los datos de una persona.

La biblioteca estándar de Java incluye dos implementaciones de la interfaz *Map*: *HashMap* y *TreeMap*.

2.3 Primitivas

Las primitivas más importantes del TAD diccionario son:

- `insertar(c, s)`: asociar a la clave *c* el significado _s
- `obtener(c)`: obtener el valor asociado con una clave.
- `pertenece(c)`: preguntar si está presente la clave en el diccionario.
- `borrar(c)`: eliminar una pareja del diccionario, buscando por su clave.

El resto de métodos de los diccionarios de Java se pueden consultar en la documentación de Map[2]. Las cuatro primitivas anteriores se traducen así:

- `V put(K key, V value)` (guarda un elemento y devuelve el valor previamente asociado con la clave, o *null* si no existía antes de la inserción)
- `V get(Key k)`: devuelve el valor asociado con una clave
- `boolean containsKey(Key k)`: devuelve *true* si la clave está presente en el diccionario.
- `V remove(Key k)`: elimina la pareja clave-valor del diccionario, y devuelve este último.

2.4 Estructuras y complejidad

El TAD diccionario puede estar implementado sobre cualquier estructura, pero por razones de eficiencia las más comunes son dos: un árbol binario de búsqueda, o una tabla hash.

La clase *TreeMap* de Java implementa un diccionario sobre ABB. Las cuatro primitivas arriba mencionadas tienen complejidad $O(\log n)$. Java mantiene la invariante del ABB comparando los elementos entre sí al insertar.

[2] https://docs.oracle.com/javase/7/docs/api/java/util/Map.html

La implementación de diccionario de Java sobre tabla hash se llama *HashMap*. En general, las primitivas de diccionario sobre tabla hash tienen complejidad $O(1)$. Esto se consigue con el uso de una *función hash* sobre K (no hace falta función hash para V).

Java ya proporciona una función hash para todos sus tipos primitivos, incluyendo *String*. Por tanto, en caso de tener un diccionario *Map<String, Persona>* no sería necesario que el usuario proporcione ninguna función de hash. Lo mismo ocurriría con, por ejemplo, *Map<Integer, String>*.

Por el contrario, si K es un tipo proporcionado por el usuario, por ejemplo en *Map<Persona, Integer>* el usuario *sí* tendría que proporcionar una función hash para el tipo *Persona*.

3 Cola doblemente terminada

3.1 Comparación

En principio, el TAD "cola doblemente terminada" es genérico sobre un T arbitrario y no realiza operaciones sobre los elementos: simplemente los almacena sin examinarlos.

La interfaz *Deque* de Java, no obstante, añade algunos métodos sobre el TAD que sí "miran" a los elementos, en particular para realizar comparaciones de igualdad.

Las cuatro primitivas de *Deque* que realizan comparaciones de igualdad son:

- `removeFirstOccurrence()`
- `removeLastOccurrence()`
- `contains()`
- `containsAll()`

Estas comparaciones se deben realizar con el método *equals()*, que está presente en todos los objetos de Java. Su signatura es:

```
public boolean equals(Object obj);
```

Esto es, recibe un objeto anónimo de tipo *Object*, y no un T.[3]

Por esta razón, las cuatro primitivas que realizan comparaciones incluyen *Object* en su firma:

- `removeFirstOccurrence(Object obj)`
- `removeLastOccurrence(Object obj)`
- `contains(Object obj)`
- `containsAll(Collection<?> col)`

En el último de ellos, `Collection<?>` es, a efectos prácticos, equivalente a `Collection<Object>`. La iteración se debe realizar con una variable de tipo *Object*:

[3] Esto es así porque el método *equals()* existe desde Java 1.1, mucho antes de que se introdujeran los tipos paramétricos en Java 1.5.

```
for (Object elem : col) {
    // ...
}
```

3.2 Excepciones

Solo hay cuatro primitivas que lanzan excepciones, y solo la lanzan cuando el deque está vacío:

- `getFirst()`
- `getLast()`
- `removeFirst()`
- `removeLast()`

Todas lanzan la misma excepción *NoSuchElementException:*

```
if (isEmpty())
    throw new NoSuchElementException();
```

4 Ciclo *foreach*

En general, estamos acostumbrados a iterar sobre listas mediante un contador:

```
int suma(List<Integer> lista) {
    int suma = 0;

    for (int i=0; i < lista.size(); i++)
        suma += lista.get(i);

    return suma;
}
```

Sin embargo, hay tipos de colecciones como *Stack* (pila) o *Set* (conjunto) que no ofrecen el método `get(i)`. **Para ellos, y también para listas,** Java ofrece el siguiente tipo de iteración no basada en contador (comúnmente denominado *"foreach loop"):*

```
int suma(List<Integer> lista) {
    int suma = 0;

    for (Integer x : lista)
        suma += x;

    return suma;
}
```

Además, ahora en lugar de *List* podemos tomar cualquier colección de Java como parámetro: `Set<Integer>`, `Stack<Float>`, etc.[4]

[4] Hay, además, una sutileza importante: la implementación de *suma()* con contador es $O(n)$ para *ArrayList*, pero $O(n^2)$ para *LinkedList* (nota: ¿por qué?). Por el contrario, la segunda versión es $O(n)$ para las clases.

4.1 Iteradores

El el ciclo *for* con contador guarda su estado entre una ejecución y la siguiente en, precisamente, la variable contadora. En cierto modo, esa variable actúa como la "memoria" del ciclo para saber por dónde va.

El ciclo foreach funciona parecido gracias a una abstracción llamada *iterador*. En este caso, no obstante, ese pseudo-contador/iterador que mantiene el estado entre ejecución y ejecución queda oculto en la implementación del ciclo *foreach*, y el programador no tiene acceso directo.

Sin embargo, sí es posible trabajar con iteradores directamente desde fuera de un ciclo foreach. Cada colección tiene un método `iterator()` que devuelve un "objeto para iterar" apuntando al primer elemento de la colección:

```
List<String> lista = new ArrayList<>();

lista.add("uno");
lista.add("dos");

Iterator<String> iterador = lista.iterator();

iterador.next();     // Devuelve "uno"

iterador.hasNext();  // Devuelve true: todavía quedan elementos.
iterador.next();     // Devuelve "dos"

iterador.hasNext();  // Devuelve false.
```

Mirando el comportamiento de este objeto se puede ver, por tanto, que una manera equivalente de implementar *suma()* que todavía funciona para todas las colecciones es:

```
int suma(List<Integer> lista) {
    int suma = 0;
    Iterator<Integer> iterador = lista.iterator();

    while (iterador.hasNext())
        suma += iterador.next();

    return suma;
}
```