Recuperatorio trabajo práctico n.º 2

Programación II

Segundo cuatrimestre 2016

La fecha de entrega del recuperatorio es el **29 de noviembre de 2016**. Se debe incluir en la entrega la solución corregida a la consigna inicial,[1] más la solución a las dos ampliaciones que se proponen a continuación.

1 Métodos adicionales para *DequeEnlazada*

Se pide implementar en la clase *DequeEnlazada* los métodos estándar de Java *equals()* y *toString()*.

1.1 Igualdad

Para cualquier clase de Java, la signatura estándar del método *equals()* es:

```
public boolean equals(Object obj) { ... }
```

Es decir, siempre toma como parámetro una instancia genérica de *Object*.

Como recordatorio a lo visto en clase, los pasos a realizar son:

1. Comprobar que *obj* no sea `null`.
2. Comprobar si *obj* fuera en realidad una referencia a *this*.
3. Comprobar, usando el operador `instanceof`, que *obj* es una instancia de *DequeEnlazada*.
4. Realizar sobre *obj* una coerción al tipo `DequeEnlazada<?>`.

Una vez realizada la conversión de tipo, se procederá a la comparación de las estructuras, que deben tener exactamente los mismos elementos en el mismo orden.

La comparación elemento a elemento se debe delegar al completo en el método *equals()* del tipo paramétrico *T* de la cola. Para facilitar el manejo de *null* se recomienda usar el método *Objects.equals()*[2] de Java.

[1] https://github.com/ungs-prog2/codigo_aula/archive/tp2_2016_2.zip
[2] https://docs.oracle.com/javase/7/docs/api/java/util/Objects.html

1.2 Conversión a cadena

La firma del método *toString()* es:

```
public String toString() { ... }
```

Para *DequeEnlazada* se pide devolver exactamente el siguiente formato:

```
Deque{s₁ -> s₂ -> s₃}
```

O, si la cola está vacía o tiene un solo elemento:

```
Deque{}
Deque{s₁}
```

donde $s_1, s_2, \ldots, s_n$ corresponden a la representación como cadena de los n elementos almacenados y '->' es literalmente los dos caracteres ASCII *guión* y *mayor que*.

Adicionalmente, para evitar malgastar memoria, se debe implementar haciendo uso de la utilidad *StringBuilder* de Java.[3]

2 Métodos adicionales para la red social

Se pide implementar un mismo método *toString()* para las dos clases de red social, más la primitiva de red asimétrica que se describe a continuación.

2.1 Red inversa

A menudo en algoritmos de redes sociales (y grafos en general) se necesita obtener la inversa de una red asimétrica. Para este recuperatorio se pide implementar un método que realice precisamente esa tarea:

```
public RedSocialAsim inversa() { ... }
```

El algoritmo para calcular la red inversa es muy sencillo:

```
procedimiento "calcular red inversa":
    1. crear nueva red asimétrica
    2. para cada persona P de la red original:
       a. obtener la lista de relaciones de P
       b. para cada persona R de esa lista:
          => agregar a la nueva red la relación inversa R -> P
```

2.2 Lista de personas en la red

Una funcionalidad importante que no se incluyó en la versión inicial de la red, y que se necesita para el algoritmo de inversión, es poder acceder a la lista de personas que

[3] https://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html

la componen. La solución más elegante es hacer que *RedSocial* se pueda usar en un ciclo *foreach* directamente:

```
for (String p : original) {
    System.out.println("En la red está: " + p);
}
```

2.3 De *iterador* a *iterable*

En la consigna anterior se explicó en detalle el concepto de "objeto iterador" (interfaz *Iterator* de Java). Ahora definimos un objeto *O* como "iterable" cuando implementa la interfaz *Iterable* de Java y, por tanto:

1. incluye el método `iterator()` que define la interfaz, el cual devuelve un "objeto iterador" para la estructura.
2. se puede usar el objeto *O* en un ciclo *foreach*.

Así, si *RedSocial* implementa esa interfaz:

```
class RedSocial implements Iterable<String>
{
    @Override
    public Iterator<String> iterator() { ... }
}
```

será posible usarla dentro de un ciclo *foreach*.

2.4 Relaciones de una persona

En la consigna original la signatura del método *relaciones()* era:

```
public Iterator<String> relaciones(String p) { ... }
```

Esto permitía cumplir con los requisitos de complejidad y encapsulamiento exigidos, pero la iteración era tediosa porque se debía invocar de manera explícita los métodos de *Iterator*:

```
Iterator<String> it = red.relaciones("Spinoza");
while (it.hasNext())
    System.out.println("Spinoza influenció a: " + it.next());
```

Si, en cambio, modificamos la signatura para que devuelva un *iterable*:

```
public Iterable<String> relaciones(String p) { ... }
```

sería posible escribir directamente:

```
for (String f : red.relaciones("Spinoza"))
    System.out.println("Spinoza influenció a: " + f);
```

Fe de erratas: La versión inicial de la consigna mezclaba, inicialmente, el concepto de *iterador* e *iterable* en los ejemplos de código. Se corrigió y se aclara aquí..

2.5 Entrega

En definitiva, para el recuperatorio de la parte 2 se pide:

1. Implementar la interfaz `Iterable<String>` en el TAD red social.
2. Cambiar la firma del método *relaciones()* a:

   ```
   public Iterable<String> relaciones(String persona) { ... }
   ```

3. Incluir una implementación de *toString()* común para todas las redes sociales, y una primitiva `inversa()` para las redes asimétricas. Además:
   - al igual que en *DequeEnlazada*, `toString()` debe usar *StringBuilder* para no realizar construcciones de cadenas innecesarias
   - tanto en `inversa()` como en `toString()` solamente se permite usar métodos públicos de la red.

Instrucciones para la entrega

Las instrucciones de entrega son las mismas que en la consigna original. La copia en papel para la comisión 1 se debe dejar en el casillero de los profesores Simó o Bertaccini antes de las 20h del día de la entrega.