Trabajo práctico n.º 2

Programación II

Segundo cuatrimestre 2016

La fecha de entrega de este TP es el **17 de noviembre de 2016**. Se puede realizar de manera individual, o en grupos de a lo sumo dos personas.

El TP consta de dos ejercicios que se entregarán de manera conjunta en un único ZIP con el código Java. Se deben seguir las instrucciones especificadas al final de esta consigna.

El código de apoyo para los ejercicios se puede descargar aquí:

`https://github.com/ungs-prog2/codigo_aula/archive/tp2_2016_2.zip`

Contenidos

1 Generics

1.1 Consigna

Se pide implementar **la interfaz Deque de Java** usando nodos doblemente enlazados. Dicha interfaz representa una cola doblemente terminada.

Lineamientos de la entrega:

- La única estructura auxiliar que se permite es *Nodo*.
- El método `removeLast()` debe tener complejidad computacional $O(1)$.
- El nombre de la clase debe ser *DequeEnlazada* y debe implementar correctamente la interfaz *Deque* de Java.[1]
- Los siguientes métodos no forman parte del trabajo práctico, esto es, no se deben implementar; para cumplir la interfaz, simplemente devolverán *null* (o *false*, según corresponda):
  - `iterator()`
  - `descendingIterator()`
  - `toArray()`
  - `removeAll()`
  - `retainAll()`

El resto de la sección describe en mayor detalle la tarea a realizar.

1.2 Cola doblemente terminada

Una cola doblemente terminada[2] es una estructura de datos normalmente enlazada con primitivas de inserción y borrado para cada extremo (cabeza y cola).

Las primitivas resultantes son:

	Inserción	Examen	Borrado
Cabeza	`insertarPrimero`	`verPrimero`	`eliminarPrimero`
Cola	`insertarUltimo`	`verUltimo`	`eliminarUltimo`

De estas seis primitivas, las primeras cinco son $O(1)$ de manera trivial al combinar una estructura enlazada simple con una referencia al último elemento. Por el contrario, para implementar `eliminarUltimo()` en tiempo constante es imprescindible usar nodos doblemente enlazados.

En los nodos doblemente enlazados cada nodo mantiene dos referencias: una a su nodo siguiente y una segunda a su nodo anterior. Para mantener esta invariante, todas las primitivas de la estructura deberán actualizar ambas referencias cuando corresponda.

[1] https://docs.oracle.com/javase/7/docs/api/java/util/Deque.html
[2] https://es.wikipedia.org/wiki/Cola_doblemente_terminada

1.3 La interfaz *Deque* de Java

Por los siguientes motivos, la interfaz *Deque* de Java incluye más primitivas de las que se listaron arriba:

1. Las primitivas de examen y borrado ofrecen dos variantes según su comportamiento cuando la cola está vacía:
   - `removeFirst()` y `getFirst()` lanzan *NoSuchElementException*.
   - `pollFirst()` y `peekFirst()`, por el contrario, simplemente devuelven *null*.
2. Debido a interfaces como *Stack* y *Queue*, anteriores a Java 1.5, *Deque* ofrece más de un nombre para algunos de sus métodos, así:

Alias	Canónico
`push()`	`addFirst()`
`add()`	`offerLast()`
`offer()`	`offerLast()`
`pop()`	`removeFirst()`
`remove()`	`removeFirst()`
`poll()`	`pollFirst()`
`peek()`	`peekFirst()`
`element()`	`getFirst()`

3. *Deque*, además, hereda de *Collection*, lo cual añade (entre otros métodos):
   - `clear()`
   - `addAll()`
   - `contains()`
   - `containsAll()`
4. *Deque* viola (o extiende) la interfaz estándar de cola doblemente terminada ofreciendo borrado de nodos intermedios:
   - `removeFirstOccurrence()`
   - `removeLastOccurrence()`

1.4 Materiales y ayuda para la implementación

Para aliviar la cantidad de "trabajo burocrático" que conlleva implementar la interfaz *Deque* al completo, se incluye en el código de apoyo una clase abstracta *DequeBase*.

Esta clase marca como *abstract* las diez o doce primitivas principales de la clase a implementar por el alumno, e incluye definiciones para las demás.[3]

De usarse esta clase base, la declaración final de *DequeEnlazada* quedaría:

```
public class DequeEnlazada<T>
                extends DequeBase<T> implements Deque<T> { ... }
```

[3] El uso de esta clase base, aunque recomendado, es enteramente opcional; verdaderamente, los lineamientos de la entrega al principio de la consigna son suficientes para implementar el ejercicio.

Otras indicaciones:

- se puede arrancar a partir de las clases *PilaInt* y *ColaInt* vistas en la cursada y/o en materias anteriores. El código de apoyo incluye sendas implementaciones como referencia.
- también se incluye una clase *TestDeque* con pruebas para la implementación a entregar. El código entregado *debe* pasar estas pruebas para ser admisible. No obstante, durante la corrección se podrá aplicar un conjunto más extenso de pruebas.

1.5 Ejemplo de uso

Un ejemplo inicial del uso de *Deque* sería:

```
Deque<String> deque = new DequeEnlazada<>();

deque.addFirst("b");
deque.addFirst("a");
deque.addLast("c");

deque.size();        // 3
deque.isEmpty();     // false

deque.getFirst();    // "a"
deque.removeLast();  // "c"
deque.getLast();     // "b"
deque.size();        //  2


// Excepciones sobre un deque vacío:

Deque<String> vacío = new DequeEnlazada<>();

deque.isEmpty();    // true
vacío.peekFirst();  // devuelve null
vacío.getFirst();   // lanza NoSuchElementException
```

2 Polimorfismo

2.1 Introducción

Como parte de un proyecto de investigación, en la UNGS se está diseñando una nueva red social y nos han pedido un diseño de clases para representar las relaciones entre distintas personas.

En esta primera versión solo guardaremos el nombre de las personas de la red y sus relaciones, aunque quizá en el futuro se haga necesario almacenar más información.

Lo que sí quedó establecido es que se debe poder representar dos tipos distintos de red social:

- en una *red simétrica* se considera que las relaciones son recíprocas, esto es: si A aparece en la lista de relaciones de B, B también debe aparecer en la lista de A.
- en una *red asimétrica* no se exige que las relaciones sean recíprocas: A puede tener a B en su lista de relaciones sin que B necesariamente tenga a A en la suya.

2.2 Objetivos

Las siguientes objetivos deben primar en la versión inicialmente entregada a los investigadores:

- Minimizar la cantidad repetida de código.
- Diseñar de manera que los futuros algoritmos puedan funcionar sobre cualquiera de los dos tipos de red.
- Usar las estructuras de datos adecuadas para una implementación eficiente tanto en espacio como en tiempo.

2.3 Operaciones sobre una red social

Las operaciones más importantes sobre la clase-TAD *RedSocial* son dos:

1. agregar una relación entre dos personas, A y B.[4]
2. obtener el conjunto de relaciones de una persona A.[5]

Como primitivas de consulta auxiliares se debe poder obtener:

3. el número personas en la red.
4. el número total de relaciones en la red.
5. el número de relaciones de una persona en concreto.

[4] La API debe indicar si la operación se llevó a cabo o no, esto es: si efectivamente se añadió la relación porque no existía previamente o si, por el contrario, no se realizó acción alguna porque la relación ya existía.

[5] No se exige ningún orden específico; queda gobernado por la implementación.

Al crear la red no se conoce cuántas personas formarán parte de ella. Tampoco hay una primitiva explícita para agregar personas: estas se incorporan a la red de manera gradual, según se crean relaciones que las involucran.[6]

2.4 Complejidad asintótica

Se establece que el tiempo de ejecución de las siguientes cuatro primitivas debe estar en $O(1)$:

- `totalPersonas()`
- `totalRelaciones()`
- `numRelaciones(a)`
- `agregarRelacion(a, b)`

También se establece un límite al uso de memoria: la complejidad espacial de la red, esto es, el espacio que ocupa el objeto en memoria, debe quedar en $O(n \times m)$ —donde n y m son, respectivamente, el número de personas y el número total de relaciones en la red.

Finalmente, se desea que la lista de relaciones de una persona se pueda obtener con complejidad espacial $O(1)$ y se pueda recorrer con complejidad temporal $O(m_a)$, donde m_a es el número de relaciones de esa persona.

2.5 Informe

Para este segundo ejercicio (no así para el primero) se pide un informe de como mucho *una* carilla con la siguiente información:

- uno o dos párrafos sobre cómo se alcanzó un diseño polimórfico: si mediante interfaces o mediante herencia, y por qué de ese modo frente al otro.
- uno o dos párrafos explicando cómo se evitó la repetición de código, y en qué casos no se pudo evitar (si hubo alguno).
- uno o dos párrafos describiendo las dificultades encontradas en el uso de estructuras de datos, o cualquier otro aspecto de la implementación.

2.6 Guía de implementación

Muchas de las primitivas se pueden implementar en $O(1)$ simplemente usando de modo adecuado las implementaciones de diccionario y/o conjunto de Java: *HashMap*, *HashSet*, etc. No se pide, en ningún caso, que estas estructuras sean implementadas como parte del trabajo práctico. Lo mismo ocurre para la complejidad espacial de la red.

El caso más complejo es la obtención de las relaciones de una persona:

- al pedirse complejidad espacial $O(1)$, no se debe realizar una copia de esa lista o estructura.

[6] Por esta razón, el único error posible resulta de consultar el número de relaciones de una persona que no está presente en la red; en ese caso, es suficiente con devolver cualquier valor negativo. En el caso de las relaciones de una persona que no está en la red, se *sugiere* devolver el conjunto vacío.

- al mismo tiempo, la interfaz no debe exponer ni la implementación particular de la estructura que almacena las relaciones, ni desde luego permitir su modificación desde fuera.

Por esto, se aconseja el uso de iteradores de manera que se pueda escribir:

```
static void imprimirRelaciones(RedSocial red, String persona)
{
    for (String r : red.relaciones(persona))
        System.out.println(r);
}
```

donde la primitiva *relaciones()* tendría la siguiente signatura:

```
Iterator<String> relaciones(String persona)
{
    // ...
}
```

El uso de iteradores permite cumplir con los tres requisitos: complejidad espacial $O(1)$, complejidad temporal $O(m_a)$ y no exponer la estructura subyacente ni permitir su modificación.

Cabe notar que, por lo general, *no* corresponde implementar un iterador propio para estructuras estándar del lenguaje. En particular, todas las colecciones de Java tienen un método *iterator()* en el que se puede delegar la primitiva:

```
Iterator<String> relaciones(String a)
{
    // Obtener una referencia a la estructura de 'a' y devolver
    // su iterador:
    return estructura_A.iterator();
}
```

2.7 Caso de prueba

Junto con la implementación de las clases necesarias, se pide un archivo de ejemplo donde:

1. se cree una red de cada tipo
2. se invoque a los distintos métodos de cada red, comprobando que el resultado sea correcto.

Como *mínimo*, en el caso de prueba se tiene que construir la red que se representa en la figura 1.

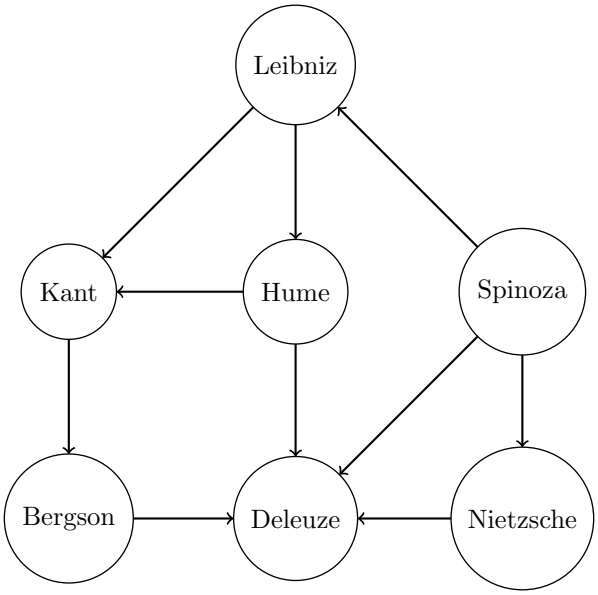

Figura 1: ¿Quién influenció a quién? (7 nodos y 10 relaciones)

Instrucciones para la entrega

Formato

La entrega se realiza por correo electrónico en un mensaje que debe contener dos archivos adjuntos:

- primer adjunto: un archivo PDF con el informe (una carilla) del ejercicio 2.
- segundo adjunto: un archivo ZIP con los archivos fuente del proyecto.[7] En particular, se espera:
  - la solución al ejercicio 1 en un un archivo llamado *DequeEnlazada.java*.

    Los archivos *DequeBase.java* y *TestDeque.java* **no** deben ser modificados y no forman parte de la entrega.

    Si, por iniciativa propia, el alumno añade sus propios casos de prueba, los puede entregar en un archivo *TestDequeAlumno.java*.
  - los archivos de código para el ejercicio 2, que serán al menos tres: las implementaciones de red simétrica y asimétrica más sus casos de prueba. Obviamente, se permite incluir más clases y archivos de código según necesite el diseño propuesto.

Las direcciones de entrega electrónica son:

- Comisión 1: `entregas.prog2@gmail.com`
- Comisión 2: `ungsmpprogramacion2@hotmail.com.ar`

[7] Al exportar desde Eclipse se debe seleccionar solamente el directorio 'src'. No debe incluirse ningún archivo en formato JAR y, preferentemente, tampoco archivos *.class* (bytecode de Java).

El asunto debe incluir "TP2 2016/2", la comisión (COM1 o COM2), el número de grupo (p.ej. G3 o G17) y los apellidos de sus integrantes. Por ejemplo:

```
Asunto: TP2 2016/2 - COM1 - Bertaccini Simó G9
```

Requisitos

Todo el código entregado debe compilar para considerarse una entrega válida.

Además, la implementación de *DequeEnlazada* debe pasar los casos de prueba que se adjuntaron a la consigna (*TestDeque.java*).

Copia en papel

Para la comisión 1, además, se debe entregar una copia impresa de código e informe. Se debe entregar en mano en el laboratorio el día de la entrega o, alternativamente, en el casillero de los profesores en el ICI hasta las 19h del mismo día.

Consideración final

De haber múltiples entregas, se considera como definitiva la *última* enviada. Caso de no ser la correcta, se debe enviar de nuevo la versión que sí lo es.