

**SEGUNDO EXAMEN PARCIAL**  
**ESTRUCTURA DE DATOS Y ALGORITMOS**  
**27 MARZO 2014**  
**(Grupo 81M – Leganés Mañana)**

<b>Apellidos y Nombre:</b>	
<b>Grupo</b>	

**Algunas reglas:**

- Antes de comenzar el examen, escribe tu nombre y grupo.
- Lee atentamente el enunciado de cada ejercicio.
- Utiliza las hojas de examen (hojas de cuadros) para preparar tu respuesta. Una vez que estés segur@ de la solución de un ejercicio, escribe **la respuesta en estas hojas de examen (cuadernillo)** el hueco que le corresponde. Los profesores solo recogerán este cuadernillo.
- Cuando finalice la prueba, se deben entregar el enunciado del examen y cualquier hoja que haya empleado.
- No está permitido salir del aula por ningún motivo hasta la finalización del examen.
- Desconecten los móviles durante el examen.

**Características:**

- El examen durará **2 horas**.
- La puntuación de cada pregunta está indicada (sobre 10 puntos).
- Este examen supone un **20% de la calificación final** de la asignatura.
- **Todos los métodos necesarios de las librerías de EDA son públicos y los atributos de los TADs son públicos (para facilitar el acceso a los mismos en el examen).**

1. (2 puntos) Dada el siguiente código relacionado con la interfaz que define las operaciones que se pueden realizar con Conjuntos (*ISet*)

```
public interface ISet<E> {
    boolean isEmpty();
    boolean addElement(E e);
    boolean deleteElement(E e);
    E getElement(int index);
    int size();
    boolean equivalence(ISet<E> other);
    boolean isSubset(ISet<E> other);
    boolean isSuperset(ISet<E> other);
    ISet<E> union(ISet<E> other);
    ISet<E> intersection(ISet<E> other);
    ISet<E> difference(ISet<E> other);
}
```

y dada la clase *DSet* que implementa los conjuntos mediante una Lista Doblemente Enlazada (*DList*):

```
public class DSet<E> implements ISet<E> {  
    DList<E> listset;  
    public DSet(){  
        listset = new DList<E>();  
    }  
}
```

Implementa el método *union* definido en la interfaz *ISet*.

Solución:

```
@Override  
public ISet<E> union(ISet<E> other) {  
    DSet<E> unionlist = new DSet<E>();  
  
    // First, it includes the elements at listset  
    for (int i = 0; i < listset.getSize(); i++) {  
        unionlist.addElement(listset.getAt(i));  
    }  
  
    // Then it includes the elements at other  
    for (int i = 0; i < other.size(); i++) {  
        unionlist.addElement(other.getAt(i));  
    }  
  
    return unionlist;  
}
```

2. (2 puntos) El coste de la operación *addLast* en listas simples (*SList*) es muy alto, como bien sabemos. ¿Podríamos mejorar la implementación de *SList* de alguna forma para optimizar la operación de insertar un elemento al final de la lista? Crear una nueva clase *SListTail* cuyo principal objetivo sea implementar todos los métodos de la lista, optimizando la operación *addLast*.

a) (0,5 puntos) Escribe la cabecera de la clase *SListTail*, sus atributos y su constructor:

```
// NOTA: supongo que SList<E> tiene el atributo "size" indicando el tamaño  
public class SListTail<E> extends SList<E> {  
    SNode<E> tailNode;  
  
    SListTail() {  
        super();  
        tailNode = null;  
    }  
}
```

b) (0,5 puntos) ¿Sería necesario cambiar la implementación del método *addFirst* heredado de *SList*? Si es así, escribe el código del mismo

```
@Override
public void addFirst(E elem) {
    SNode<E> newNode = new SNode<E>(elem);

    //Es necesario añadir el siguiente if
    if (this.isEmpty()){
        tailNode = newNode;
    }

    newNode.nextNode = firstNode;
    firstNode = newNode;
    size++;
}
```

c) (0,5 puntos) ¿Sería necesario cambiar la implementación del método *addLast* heredado de *SList*? Si es así, escribe el código del mismo

```
@Override
public void addLast(E elem) {
    SNode<E> newNode = new SNode<E>(elem);
    if (this.isEmpty()) {
        firstNode = newNode;
    } else {
        tailNode.nextNode = newNode;
    }
    tailNode = newNode;
    size++;
}
```

d) (0,5 puntos) Dada la siguiente implementación del método `removeFirst`, ¿crees que hay algún error en el código? En caso afirmativo, indica por qué modifica el código para solucionarlo

```
/**
 * Borra el primer nodo de la lista
 */
@Override
public void removeFirst() {
    if (firstNode != null) {
        //Basta con pasar a que firstNode apunte a su sucesor
        firstNode = firstNode.nextNode;
        if (tailNode==null) tailNode=null;
        //decrementamos el tamaño de la lista
        size--;
    }
}
```

```
@Override
public void removeFirst() {
    if (firstNode != null) {
        firstNode = firstNode.nextNode;

        // En esta línea estaba el error, cuando la lista tenía
        // un solo elemento (ahora se comprueba firstNode = null)
        if (firstNode == null) {
            tailNode = null;
        }
        size--;
    }
}
```

3. (3,5 puntos) Dada la clase *sumaColas* que se define como se indica en el código, implementa el método *sumarColas* que dadas dos colas de números enteros del mismo tamaño ( $C1[e_1, e_2, \dots, e_n]$  y  $C2[f_1, f_2, \dots, f_n]$ ) devuelve una nueva cola  $C3$  donde se ha realizado la suma de los elementos de las anteriores 2 a 2, pero en la  $C2$  comenzando en sentido inverso:  $C3[e_1+f_n, e_2+f_{n-1}, \dots, e_n+f_1]$ .

PISTA: utiliza una pila auxiliar para hacer los cálculos

Ejemplo:

Cola1:	1	2	3
--------	---	---	---

Cola2:	4	5	6
--------	---	---	---

ColaResultado:	1+6	2+5	3+4
----------------	-----	-----	-----

```

/** Clase que añade un método SumaColas a la implementación de SQueue para
 * números enteros */
public class SumaColas extends SQueue<Integer> {

    public SQueue<Integer> sumarColas (SQueue<Integer> cola){
        //comprobamos que las colas sean del mismo tamaño
        if (cola.getSize()!=this.getSize()){
            System.out.println("Error: las colas no son del mismo
tamaño");
            return null;
        }

        //si son del mismo tamaño
        SStack<Integer> pila = new SStack<Integer>();
        SQueue<Integer> colaFin = new SQueue<Integer>();

        // cada elemento de una de las colas se introduce en una pila
        while (!cola.isEmpty()){
            pila.push(cola.dequeue());
        }

        // se suman los números de la pila + cola
        // e insertan en la nueva cola
        while (!pila.isEmpty()){
            colaFin.enqueue(pila.pop()+this.dequeue());
        }

        return colaFin;
    }
}

```

4. (2,5 puntos) Marca verdadero o falso en las siguientes sentencias y justifica tu respuesta.

a) (0,5 puntos) Invocar a un método estático de una clase requiere la creación previa de un objeto de la clase

V [ ] F [X]

Justificación:

Los métodos estáticos se pueden invocar directamente con:

<nombre de la clase>.< nombre del método>

P.e. suponiendo la clase *Clase* con el método estático *método()*, se puede realizar directamente la invocación de:

*Clase.metodo();*

Sin necesidad de crear un objeto de tipo *Clase* previamente

b) (0,5 puntos) Una de las diferencias entre las Interfaces y las Clases Abstractas es que las primeras pueden tener variables en su definición.

V [ ] F [X]

Justificación:

Las Interfaces no pueden definir variables en su implementación, solo constantes y métodos.

c) (0,5 puntos) ¿Se puede utilizar tipos de datos primitivos al instanciar clases parametrizables? Por ejemplo, ¿el siguiente código sería correcto?

```
public class ListOfChars extends DList<char>
```

V [ ] F [X]

Justificación:

No, nunca, solo tipos de datos no primitivos. P.e. no se puede utilizar "char", pero sí "Character" para instanciar la clase anterior.

d) (0,5 puntos) El TAD Cola (Queue) se basa en la estrategia FIFO (First-IN, First-OUT). Esto significa que el último elemento insertado es el primero en ser devuelto.

V [ ] F [X]

Justificación:

Primer elemento en insertar, primero en ser devuelto (FIFO)

- e) (0,5 puntos) Solo hay una forma de implementar la interfaz IStack (relativa al TAD Pilas), mediante Listas Simplemente Enlazadas (SList).

V [] F [X]

Justificación:

Existen muchas formas de implementar la interfaz IStack, mediante arrays, listas dobles, etc.

- f) (0,5 puntos) En el TAD *DList* la operación *addFirst()* es más costosa que la operación *addLast()*.

V [] F [X]

Justificación:

No, al tratarse de una lista doblemente enlazada (DList), ambas tienen la misma complejidad