NOMBRE:

APELLIDOS:

NIA:

GRUPO:

Segundo examen parcial

2ª Parte: Problemas (7 puntos sobre 10)

Duración: 80 minutos
Puntuación máxima: 7 puntos
Fecha: 4 mayo 2018

Instrucciones para el examen:

- No se permite el uso de libros o apuntes, ni tener teléfonos móviles u otros dispositivos electrónicos encendidos. Incumplir cualquiera de estas normas puede ser motivo de expulsión inmediata del examen.
- Rellena tus datos personales antes de comenzar a realizar el examen.

Problema 1: 1 punto

Queremos sumar los elementos de un array de enteros, desde la posición 0 hasta la última, pero parando la suma (y mostrando un mensaje de error) en cuanto se supere un tope dado. Para ello se pide desarrollar dos métodos.

Apartado 1.1: 0.8 puntos

Programe un método basado en **recursividad por la cola** que reciba 4 argumentos (el array, el tope, la posición que toca sumar en ese momento y el resultado ya acumulado hasta ese momento). Si el tope es alcanzado en algún momento por el resultado acumulado hasta ese momento, se debe dejar de sumar elementos, imprimir el mensaje de error *"Límite alcanzado"* y devolver un -1 como resultado. Este método será private (o auxiliar), pues el punto de entrada será el del apartado 1.2. **No será válido el uso de elementos iterativos (bucles).**

```
private static int sumaArray(int a[], int tope, int i, int resul) {
        /* IMPLEMENTAR AQUÍ */




}
```

Apartado 1.2: 0.2 puntos

Programe un segundo método que realice la **llamada inicial al método anterior** con los valores adecuados y que será public (el que proporciona la funcionalidad hacia fuera).

```
public static int sumaArray(int a[], int tope) {
        /* IMPLEMENTAR AQUÍ */



}
```

Problema 2: 3 puntos

Dadas las siguientes clases:

```
public class LinkedQueue<E> implements Queue<E> {
    private Node<E> top;
    private Node<E> tail;
    private int size;
    public LinkedQueue() { ... }
    public boolean isEmpty() { ... }
    public int size() { ... }
    public E front() { ... }
    public void enqueue(E info) { ... }
    public E dequeue() { ... }
}
```

```
public class Node<E> {
    private E info;
    private Node<E> next;
    public Node(E info, Node<E> next) { ... }
    public E getInfo() { ... }
    public Node<E> getNext() { ... }
    public void setInfo(E info) { ... }
    public void setNext(Node<E> next) { ... }
}
```

Apartado 2.1: 1 punto

Programar en la clase *LinkedQueue<E>* un método iterativo que **concatene uno a uno** los elementos de la cola que se recibe por argumento (cola2) en la cola actual. El resultado final consistirá en que:

1. La cola actual mantendrá sus elementos iniciales al principio, seguidos a continuación de los elementos que anteriormente estaban en la cola2 (y manteniendo su orden).
2. **La cola2 quedará vacía** (como consecuencia del trasvase de sus elementos a la cola actual)

```
public void concatIter(LinkedQueue<E> cola2) {
        /* IMPLEMENTAR AQUÍ */



}
```

Apartado 2.2: 1 punto

Programar la versión recursiva del apartado 2.1.

```
public void concatRecur(LinkedQueue<E> cola2) {
        /* IMPLEMENTAR AQUÍ */




}
```

Apartado 2.3: 1 punto

Programar la versión eficiente del apartado 2.1 **(sin bucles ni recursividad)**. Para ello, bastará con **concatenar de golpe (y no uno a uno)** todos los elementos de la cola2.

```
public void concatEficiente(LinkedQueue<E> cola2) {
        /* IMPLEMENTAR AQUÍ */









}
```

Problema 3: 3 puntos

Dadas (ya implementadas) las clases *LBSTree<E>* (árbol binario de búsqueda) y *LBSNode<E>* descritas al final de este problema.

Apartado 3.1: 1 punto

Programar en la clase *LBSTree<E>* un método *boolean isSorted()* que confirme que **está de verdad ordenado**, es decir, si es cierto que **en todos los nodos del árbol** si tiene hijo a su izquierda, éste es menor y si tiene hijo a su derecha, éste es mayor.

Apartado 3.2: 1 punto

Programar en la clase *LBSTree<E>* un método: que reciba un *ArrayList<E>* por argumento y le añada los elementos del árbol recorridos en inorden.

Apartado 3.3: 1 punto

Programar en la misma clase *LBSTree<E>* un método *ArrayList<E> toArrayListOrdenados()* que devuelva *null* si el árbol no estuviera ordenado. En caso contrario devolverá un *ArrayList<E>* con todos elementos del árbol **ordenados de menor a mayor**.

NOTA IMPORTANTE 1: Algunos de los métodos de la clase *ArrayList<E>*, que podrían serle útiles para resolver **este apartado** son:

- boolean add(E e)
- void add(int index, E element)
- void clear()
- E get(int index)
- int indexOf(Object o)
- boolean isEmpty()
- E remove(int index)
- boolean remove(Object o)
- E set(int index, E element)
- int size()

NOTA IMPORTANTE 2: Las clases de apoyo para **todo este problema** son:

```
public class LBSTree<E> implements BSTree<E> {
      private LBSNode<E> root;

      public LBSTree() { ... }
      public LBSTree(Comparable key, E info) { ... }
      public boolean isEmpty() { ... }
      public E getInfo() { ... }
      public Comparable getKey() { ... }
      public LBSTree<E> getLeft() { ... }
      public LBSTree<E> getRight() { ... }
      public void insert(Comparable key, E info) { ... }
      public BSTree<E> search(Comparable key) { ...
}
}
```

```
public class LBSNode<E> {
      private E info;
      private Comparable key;
      private LBSTree<E> left;
      private LBSTree<E> right;

      public LBSNode(Comparable key, E info, LBSTree<E> left,
                        LBSTree<E> right) { ... }
      public E getInfo() { ... }
      public void setInfo(E info) { ... }
      public Comparable getKey() { ... }
      public void setKey(Comparable key) { ... }
      public LBSTree<E> getLeft() { ... }
      public void setLeft(LBSTree<E> left) { ... }
      public LBSTree<E> getRight() { ... }
      public void setRight(LBSTree<E> right) { ... }
}
```

SOLUCIONES DE REFERENCIA (Varias soluciones a cada uno de los problemas son posibles)

Solución 1.1: 0.8 puntos

```
private static int sumaArray(int a[], int tope, int i, int resul) {
        if (resul > tope) {
                System.err.println("Límite alcanzado");
                return -1;
        } else if (i >= a.length) {
                return resul;
        } else {
                return sumaArray(a, tope, i+1, resul+a[i]);
        }
}
```

Rúbrica

- 0.1: Caso base de superación de tope
- 0.1: Caso base de fin de array
- 0.6: Caso recursivo por la cola
    - 0.1: Que el índice avance en +1
    - 0.1: Que el resultado acumulado se actualice antes de la llamada recursiva
    - 0.3: Que se haga la llamada recursiva correctamente
    - 0.1: Que no se altere el resultado de la llamada recursiva
- Los errores significativos están sujetos a sanciones adicionales

Solución 1.2: 0.2 puntos

```
public static int sumaArray(int a[], int tope) {
        return sumaArray(a, tope, 0, 0);
}
```

Rúbrica

- 0.2: Llamada correcta al método recursivo, pasando 0 como tercer y cuarto argumento
- Los errores significativos están sujetos a sanciones adicionales

Solución 2.1: 1 punto

```
public void concatIter(LinkedQueue<E> cola2) {
        while (cola2.size() > 0)
                enqueue(cola2.dequeue());
}
```

Rúbrica

- 0.1: No alterar nada cuando cola2 está vacía
- 0.3: Ir desencolando por orden los elementos de la cola2
- 0.3: Ir encolando esos elementos en la cola actual
- 0.1: Que cola2 quede vacía al terminar
- 0.2: Cumplir que this.size() sumará el antiguo valor de cola2.size()
- Si no se hace de forma iterativa, el ejercicio vale 0
- Los errores significativos están sujetos a sanciones adicionales

Solución 2.2: 1 punto

```
public void concatRecur(LinkedQueue<E> cola2) {
        if (cola2.size() > 0) {
                enqueue(cola2.dequeue());
                concatRecur(cola2);
        }
}
```

Rúbrica

- 0.1: No alterar nada cuando cola2 está vacía
- 0.3: Ir desencolando por orden los elementos de la cola2
- 0.3: Ir encolando esos elementos en la cola actual
- 0.1: Que cola2 quede vacía al terminar
- 0.2: Cumplir que this.size() sumará el antiguo valor de cola2.size()
- Si no se hace de forma recursiva, el ejercicio vale 0
- Los errores significativos están sujetos a sanciones adicionales

Solución 2.3: 1 punto

```
public void concatEficiente(LinkedQueue<E> cola2) {
        if (isEmpty()) {
                top = cola2.top;
                tail = cola2.tail;
        } else if (!cola2.isEmpty()) {
                tail.setNext(cola2.top);
                tail = cola2.tail;
        }
        size += cola2.size;
        cola2.top = cola2.tail = null;
        cola2.size = 0;
}
```

Rúbrica

- 0.1: No alterar nada cuando cola2 está vacía
- 0.2: Tratar el caso de que la cola actual esté vacía
- 0.4: Tratar el caso de que la cola actual no esté vacía
- 0.1: Que cola2 quede vacía al terminar
- 0.2: Cumplir que this.size() sumará el antiguo valor de cola2.size()
- Si se hace de forma recursiva o iterativa, el ejercicio vale 0
- Los errores significativos están sujetos a sanciones adicionales

Solución 3.1: 1 punto

```
public boolean isSorted() {
        if (isEmpty()) {
                return true;
        } else if (!root.getLeft().isEmpty() && root.getKey().compareTo(root.getLeft().getKey()) <= 0) {
                return false;
```

```
        } else if (!root.getRight().isEmpty() && root.getKey().compareTo(root.getRight().getKey()) >= 0) {
                return false;
        } else {
                return root.getLeft().isSorted() && root.getRight().isSorted();
        }
}
```

Rúbrica

- 0.1: Caso base árbol vacío
- 0.4: Casos base nodos en desorden
- 0.3: Casos recursivos
- 0.2: Conjunción lógica (and) de los casos recursivos
- Los errores significativos están sujetos a sanciones adicionales

Solución 3.2: 1 punto

```
public void addInOrder(ArrayList<E> a) {
        if (!isEmpty()) {
                root.getLeft().addInOrder(a);
                a.add(root.getInfo());
                root.getRight().addInOrder(a);
        }
}
```

Rúbrica

- 0.2: No alterar nada si el árbol está vacío
- 0.3: Añadir correctamente en el arrayList el elemento de la raíz
- 0.4: Recorrer el árbol en inOrden con las llamadas recursivas correctas
- 0.1: Pasar el mismo arrayList a ambas llamadas recursivas
- Los errores significativos están sujetos a sanciones adicionales

Solución 3.3: 1 punto

```
public ArrayList<E> toArrayListOrdenados() {
        if (!isSorted()) return null;
        ArrayList<E> a = new ArrayList<E>();
        addInOrder(a);
        return a;
}
```

Rúbrica

- 0.25: Devolver null si el árbol no está ordenado
- 0.25: Construir un arrayList vacío
- 0.25: Rellenarlo recorriendo el árbol en inOrden
- 0.25: Retornar el arrayList rellenado
- Los errores significativos están sujetos a sanciones adicionales