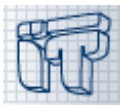


NOMBRE:
APELLIDOS:
NIA:
GRUPO:

Segundo parcial

2ª Parte: Problemas (7 puntos sobre 10)

Duración: 70 minutos
Puntuación máxima: 7 puntos
Fecha: 13 de mayo de 2021

Instrucciones para el examen:

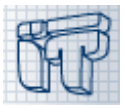
- No se permite el uso de libros o apuntes, ni tener teléfonos móviles u otros dispositivos electrónicos encendidos. Incumplir cualquiera de estas normas puede ser motivo de expulsión inmediata del examen.

Ejercicio 1 (2 / 7 puntos)

Suponiendo que tenemos un árbol con objetos de la clase Integer, se pide codificar el método `public int minTree(BTree<Integer> tree) throws BTreeException` que devuelva el **menor** de los elementos que está en el árbol. El árbol implementa la siguiente interfaz:

```
public interface BTree<E>
{
        static final int LEFT = 0;
        static final int RIGHT = 1;

        boolean isEmpty();

        E getInfo() throws BTreeException;
        BTree<E> getLeft() throws BTreeException;
        BTree<E> getRight() throws BTreeException;

        void insert(BTree<E> tree, int side) throws BTreeException;
        BTree<E> extract(int side) throws BTreeException;

        String toStringPreOrder();
        String toStringInOrder();
        String toStringPostOrder();
        String toString(); // pre-order

        int size();
        int height();

        boolean equals(BTree<E> tree);
        boolean find(BTree<E> tree);
}
```

Nota: La solución debe ser recursiva, no admitiendose soluciones que no lo sean.



Ejercicio 2 (3 / 7 Puntos)

Una empresa líder en el sector de la logística le ha pedido que desarrolle un programa para la gestión de un almacén. Se manejan, entre otras, estas clases (suponga que todas tienen implementadas los métodos `get` y `set` para los atributos):

- Producto almacenable (`StockableProduct`). Se trata de un producto que contiene un identificador numérico (`productID`) y el número de unidades de dicho producto disponibles en el almacén (`numUnits`). El resto de atributos no son relevantes para este problema.

- Pedido (`Order`). Representa el documento que contiene cada uno de los pedidos que se realizan en el almacén. Cada orden de pedido contiene la lista de productos (`productList`), que será un `ArrayList` de objetos `StockableProduct`. Tiene otro atributo que indica la prioridad (`priority`), que se representa por valores enteros, siendo los valores válidos únicamente los comprendidos entre el 1 y el 4. El resto de atributos no son relevantes para este problema.

- Gestor del almacén (`StoreManager`). Es el cerebro de la aplicación y contendrá toda la lógica del programa. Tiene una lista de productos llamada `stock` que guarde en su interior todos los productos del almacén. Este atributo pertenece a la clase `ProductList` (también ya programada), que tiene un método ya implementado `public StockableProduct search(int productID)` que busca el producto cuyo identificador coincida con el que recibe como parámetro y lo devuelve como resultado. En caso de que no haya ningún producto con ese identificador devuelve `null`.

  El almacén también tendrá:

  - Una cola de pedidos (`LinkedQueue<E>`) que almacene los pedidos (`Order`) pendientes de procesar (`ordersToProcess`).

  - Una lista enlazada (`LinkedList<E>`) que almacene los pedidos prioritarios (`Order`) pendientes de procesar (`prioritaryOrders`).

  - Una lista enlazada (`LinkedList<E>`) que almacene los pedidos (`Order`) que ya han sido procesados (`ordersProcessed`).

Las listas y la cola implementan las siguientes interfaces, respectivamente:

```
public interface LinkedListInterface<E> {
    boolean isEmpty();
    int size();
    void insert (E info);  // Inserta al principio
    void insertLast (E info);  // Inserta al final
    E extract();     // Extrae el primer elemento
    E get();  // Obtiene el primer elemento sin extraerlo
}
```

```
public interface Queue<E>
{
    boolean isEmpty();
    int size();
    void enqueue (E info);
    E dequeue();
    E front();
}
```

Apartado 1 (1 punto)

Codifique el método `public void setOrdersToProcess(ArrayList<Order> orders)` que para cada pedido del argumento que recibe debe colocarlo según su prioridad:

- Si es prioridad 1, lo debe colocar al inicio de la lista `prioritaryOrders`.
- Si es prioridad 2, lo debe colocar al final de la lista `prioritaryOrders`.

Cualquier otra prioridad válida se debe colocar en la cola de pedidos.

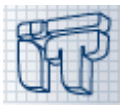
Apartado 2 (2 puntos)

Codifique el método `public void processPrioritaryOrders() throws Exception` en la clase `StoreManager` que procesa los pedidos prioritarios pendientes. Para cada pedido se deberá recorrer cada uno de los productos del pedido comprobando si el producto está en el stock y si se dispone de unidades suficientes del mismo para atender el pedido. En caso de no cumplirse ambas condiciones se debe lanzar una excepción de la clase `Exception`. Para saber si el producto está en stock hay que comparar los identificadores del producto. Si hay unidades suficientes de todos los productos, el pedido se insertará en la lista de pedidos procesados (`ordersProcessed`).

Ejercicio 3 (2 / 7 Puntos)

Dada la clase del siguiente código:

```
public class Point {

	private double x;
	private double y;

	public Point(double x, double y) {
		this.x = x;
		this.y = y;
	}

	public String toString() {
		return "(" + x + ", " + y + ")";
	}

	public double getX() {
		return x;
	}

	public double getY() {
		return y;
	}
```



Implemente el método `public static void bubbleSortPoint (Point [] a)` (que no debe devolver nada y debe ser estático) para que ordene el array según el método de la burbuja en orden **descendente, teniendo en cuenta solamente la primera coordenada del punto.**

Ejemplo:

```
Elements before sorting:
(4.0, 5.0)
(3.0, 7.0)
(5.0, 4.0)
(6.0, 8.0)
(7.0, 1.0)

Elements After sorting

(7.0, 1.0)
(6.0, 8.0)
(5.0, 4.0)
(4.0, 5.0)
(3.0, 7.0)
```

SOLUCIÓN DE REFERENCIA (Varias soluciones son posibles)

Ejercicio 1 (2 / 7 puntos)

```
public int minTree(BTree<Integer> tree) throws BTreeException {

		int min = Integer.MAX_VALUE;

		if (!tree.isEmpty()) {

			min = Math.min(minTree(tree.getLeft()), tree.getInfo());
			min = Math.min(minTree(tree.getRight()), min);

		}

		return min;
	}
```

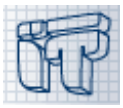
Rúbrica:

- 0 si no tiene sentido.
- 0,25 por comprobar si el árbol está vacío.
- 0,5 por cada llamada recursiva correcta
- 0,5 por comparar correctamente (izquierda y nodo, y de esos, el menor con la derecha).
- 0,25 por devolver el valor correcto (también devolver algo si está vacío).
- Los errores significativos están sometidos a penalizaciones.

Ejercicio 2 (3 / 7 puntos)

Apartado 1 (1 punto)

```
public void setOrdersToProcess(ArrayList<Order> orders) {

		for (int i = 0; i < orders.size(); i++) {
			Order o = orders.get(i);

			switch (o.getPriority()) {
			case 1:
				prioritaryOrders.insert(o);
				break;
			case 2:
				prioritaryOrders.insertLast(o);
				break;
			case 3,4:
```



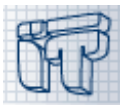
```
                        ordersToProcess.enqueue(o);
                        break;
                default:
                        System.out.println("Invalid priority");
                }
        }
}
```

Rúbrica:

- 0 si no tiene sentido.
- 0,25 por recorrer correctamente el arrayList obteniendo el pedido.
- 0,25 por cada inserción correcta (al principio y al final de la list o en la cola).
- -0,1 si no contempla que la prioridad tiene un valor correcto, o no accede al atributo prioridad por medio del getter.
- Los errores significativos están sometidos a penalizaciones.

Apartado 2 (2 puntos)

```
public void processPrioritaryOrders() throws Exception {

        for (int i = 0; i < prioritaryOrders.size(); i++) {
                Order o = (Order) prioritaryOrders.extract();

                ArrayList<StockableProduct> al = o.getProductList();

                for (int j = 0; j < al.size(); j++) {
                        StockableProduct s = al.get(i);
                        boolean ok = false;

                        StockableProduct s1 = stock.search(s.getProductID());
                        if (s1 != null) {
                                if (s1.getNumUnits() >= s.getNumUnits()) {
                                        ok = true;
                                }
                        }

                        if (!ok)
                                throw new Exception("Error in processing order");
                }
                ordersProcessed.insert(o);
        }

}
```



Rúbrica:

- 0 si no tiene sentido.
- 0,3 por recorrer la lista de órdenes
- 0,3 por extraer el elemento (0,1 si no hace el casting)
- 0,3 por extraer la lista de productos
- 0,3 por recorrer la lista de productos
- 0,3 por comprobar que el producto existe y hay unidades (ojo si es null).
- 0,25 por insertar en la cola de órdenes procesadas.
- 0,25 por lanzar la excepción.
- Los errores significativos están sometidos a penalizaciones.

Ejercicio 3 (2 / 7 puntos)

```
public static void  bubbleSortPoint (Point [] a) {
       for (int i = 0; i < a.length; i++) {
       for (int j = 0; j < a.length - i - 1; j++) {
              if (a[j].getX() < a[j+1].getX()) {

                  Point temp = a[j];
                     a[j] = a[j+1];
                     a[j+1] = temp;
              }
       }
       }
}
```

Rúbrica:

- 0 si no tiene sentido.
- 0,2 : Declaración correcta del método.
  - Penalizar 0,1 si no ponen void y/o static o sino ponen el argumento o si se equivocan en el tipo del array
- 0,4 : Primer bucle for
  - Penalizar 0,1 si ponen size() en lugar de `length`
  - Si los límites no están bien definidos, entonces 0
- 0,4 : Segundo bucle for
  - Penalizar 0,1 si ponen size() en lugar de `length`
  - Si los límites no están bien definidos, entonces 0
- 0,6 . Condicional if
  - Penalizar 0,3 si ponen en el if la ordenación ascendente
  - Penalizar 0,2 si ponen mal los índices
- 0,4 : Líneas dentro del if
  - Penalizar 0.1 por la no declaración de la variable tipo Point
- Los errores significativos están sometidos a penalizaciones.