NOMBRE:
APELLIDOS:
NIA:
GRUPO:

Convocatoria Ordinaria

2ª Parte: Problemas (7 puntos sobre 10)

Duración: 115 minutos
Puntuación máxima: 7 puntos
Fecha: 28 de mayo de 2021

Problema 1 (3 puntos)

Un centro comercial quiere implementar un sistema de control de aforo. Para ello ha colocado unos tornos en las puertas de entrada y de salida de manera que cada vez que entra una persona se incrementan dos indicadores del centro comercial: por una parte el contador de personas (`numPersonas`) de dicho centro comercial, y por otra el total de ocupación actual de todos los centros comerciales (`totalNumPersons`) que es un indicador que tiene el mismo valor en todos los centros comerciales y que indica el nivel de ocupación global. Del mismo modo, cada vez que sale una persona se decrementan ambos contadores.

Cada centro comercial, además de su aforo actual (`numPersons`) y la ocupación global (`totalNumPersons`) necesita almacenar información adicional como el nombre del centro comercial (`name`) y su capacidad máxima (`maxCapacity`). Además, las puertas de entrada y salida tendrán reconocedores faciales de personas, y cada vez que entre una persona se generará un objeto de la clase `Person` (suponga que ya está implementada con todos los constructores, métodos get, set, toString, etc.) y dicho objeto se guardará en un `ArrayList` (`people`) dentro del centro comercial. De la misma manera, cuando alguien salga por las puertas de salida el objeto `Person` debe darse de baja del `ArrayList`. Se pide:

Apartado 1.1 clase ShoppingCenter (0,75 puntos)

Declarar la clase `ShoppingCenter` con sus atributos y crear dos constructores, uno vacío (sin argumentos) y otro con todos los atributos teniendo en cuenta lo especificado anteriormente.

Apartado 1.2 método toString() (0,5 puntos)

Implementar el método `public String toString()` de la clase `ShoppingCenter`. Para el caso del atributo de tipo `ArrayList`, debe mostrarse el contenido de todos los objetos que contenga por medio del método `toString` de la clase de los objetos contenidos en él. Un ejemplo para un centro comercial llamado `sc1` con una capacidad de 3500 personas, una ocupación de 200 y un total de personas en todos los centros comerciales de 7500 sería:

```
ShoppingCenter sc1 [maxCapacity = 3500, numPersons= 200,
totalNumPersons = 7500, people = {...} ]
```

Nótese que la información del atributo `people` dependerá del número de personas que haya, y en todo caso dicha información debe ser obtenida por medio del `toString` de los objetos .

Apartado 1.3 método addPerson (0,75 puntos)

Codificar el método `public boolean addPerson(Person person)` que añadirá una persona a la lista de personas si el aforo lo permite. Si no se pudiera añadir, mostrará por pantalla un mensaje de error. Devolverá verdadero (`true`) o falso (`false`) dependiendo de si lo ha podido añadir o no.

Apartado 1.4 Método testAddPerson (1 punto)

Codificar un método de testing (`testAddPerson`) para el método `addPerson` que compruebe que se ha podido añadir correctamente un objeto Person. Sólo es necesario probar una clase de equivalencia.()

Problema 2 (1,5 puntos)

Dada la lista doblemente enlazada que se muestra en la figura y las clases `DNode` y `DLinkedList` se pide:


```java
class DNode<E> {
    DNode next, prev;
    E info;
    //constructors
    //getters and setters
}
```

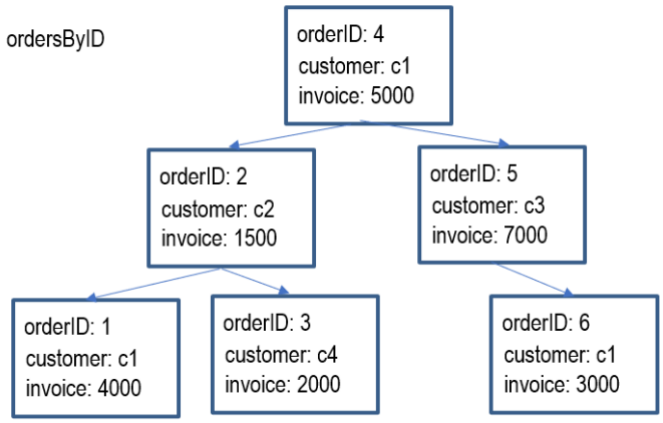
```java
public class DLinkedList<E> {
    DNode<E> first, last;
    int size;

    public DLinkedList() {
        first = new DNode<E>();
        last = new DNode<E>();
        first.setNext(last);
        last.setPrev(first);
        size = 0;
    }
    // constructors
    // getters and setters
}
```

Apartado 2.1. Método numOfOcurrences de la clase DoubleLinkedList (1,5 puntos)

programa el método `int numOfOccurrences (E info)` que devuelve el número de veces que se encuentra el elemento info dentro de la lista. En el ejemplo de la figura la llamada a `numOfOcurrences ("pear")` devolvería `2` como resultado.

Problema 3 (2,5 puntos)

La aplicación que gestiona el almacén que has creado durante el proyecto se ha ampliado creando una nueva clase para representar los pedidos ya facturados (`InvoicedOrder`). Esta clase, además de los atributos de la clase `Order`, contiene un atributo adicional con el importe de la factura (`invoice`). Además se ha añadido un atributo adicional de tipo `BSTree` a la clase `StoreManager` llamado `InvoicedOrdersByID` para almacenar todos los pedidos (`InvoicedOrder`) ordenados en función de su identificador (`orderID`). Puedes ver un ejemplo en la figura. Dadas las clases `LBSTree, LBSNode, Order, InvoicedOrder y StoreManager`. Se pide:




```java
public class Order {
    private int orderID;
    private Person customer;
    private Person employee;
    // other attributes
    // constructors
    // setters and getters
    // other methods
}
```

```java
public class InvoicedOrder extends Order
                implements Comparable{
    private double invoice;
    //getters and setters

    public int compareTo(Object other) { ... }
}
```

```java
public class StoreManager {
    ArrayList<InvoicedOrder> list;
    BSTree<InvoicedOrder> invoicedOrdersById;
    // other attributes
    // constructors
    // setters and getters

    public void swap(ArrayList<InvoicedOrder> list, int i, int j) {
        InvoicedOrder aux = list.get(i); // is equivalent to aux = a[i]; when using arrays
        list.set(i, list.get(j)); // is equivalent to a[i]=a[j]; when using arrays
        list.set(j, aux); // is equivalent to a[j]=aux; when using arrays
    }

    public int numOfOrders() {
        return numOfOrders(invoicedOrdersById);
    }

    public int numOfOrders(BSTree<InvoicedOrder> tree) { ... }

    public void insertionSort() { ... }
}
```

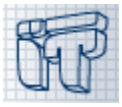
Apartado 3.1. Método numOfOrders de la clase Store Manager (1 punto)

Dada la interfaz `BSTree`, y la clase `LBSNode` programa el método **recursivo** `public int numOfOrders(BSTree<InvoicedOrder> tree)` de la clase `StoreManager` que calcula el número de pedidos almacenados en el árbol `ordersByID`. En el ejemplo de la figura la llamada al método devolvería 6. Nota: No puedes utilizar ningún método de la clase LBSTree que no aparezca en la interfaz BSTree)

```java
public interface BSTree<E> {
    boolean isEmpty();
    E getInfo();
    Comparable getKey();
    BSTree<E> getLeft();
    BSTree<E> getRight();
    String toStringPreOrder();
    String toStringInOrder();
    String toStringPostOrder();
    String toString(); // pre-order
    void insert(Comparable key, E info);
    BSTree<E> search(Comparable key);
    void exchangeInfoAndKey(LBSTree<Comparable> otherTree);
}
```

```java
public class LBSNode<E>{
    private E info;
    private Comparable key;
    private BSTree<E> right;
    private BSTree<E> left;
    // constructors
    // getters and setters
}
```

Apartado 3.2. Método insertionSort de la clase StoreManager (1,5 puntos)



Dadas las clases `InvoicedOrder` y `StoreManager`, Implementa el método `insertionSort()` que permita ordenar los pedidos facturados (`ArrayList<InvoicedOrder list`) del `ArrayList` de mayor a menor en función del importe de su factura (`invoice`) . NOTA: Puedes asumir que existe el método `swap (ArrayList<InvoicedOrder> int i, int j)` que permite intercambiar los elementos en las posiciones i y j dentro del `ArrayList` de pedidos facturados. Puedes ver el código del método swap implementado en la clase `StoreManager`. Puedes asumir también que existe el método `compareTo` en la clase `InvoicedOrder` que compara los pedidos en función del importe de su factura (`Invoice`).

SOLUCIÓN DE REFERENCIA (Varias soluciones son posibles)

Problema 1. OO + Testing (3 Puntos)

Apartado 1.1 clase ShoppingCenter (0,75 puntos)

Rúbrica:

- 0 si no tiene sentido.
- 0,3 por declarar la clase y los atributos correctamente (si no pone static máximo 0,1)
- 0,2 por tener correcto el constructor vacío.
- 0,25 por tener correcto el constructor con todos los argumentos

Solución:

```
public class ShoppingCenter {
    private String name;
    private int maxCapacity;
    private static int totalNumPersons = 0;
    private int numPersons;
    private ArrayList<Person> people;

    public ShoppingCenter() {
        people = new ArrayList<Person>();
        //this (null, 0, new ArrayList<Person>());
    }

    public ShoppingCenter(String name, int maxCapacity, ArrayList<Person> people) {
        this.name = name;
        this.maxCapacity = maxCapacity;
        this.people = people;
    }
}
```

Apartado 1.2 método toString() (0,5 puntos)

Rúbrica:

- 0 si no tiene sentido.
- 0,2 por recorrer correctamente el ArrayList
- 0,2 por llamar al toString de los objetos del ArrayList
- 0,1 por devolver correctamente el valor

Solución:

```
public String toString() {

            String result = "ShoppingCenter " + name + " [maxCapacity =" +
maxCapacity + ", numPersons = " + numPersons + ", totalNumPersons = " +
totalNumPersons + ", people=";
            for (int i = 0; i < people.size(); i++) {
                  result = result + people.get(i).toString() + "\n";
            }
```

```
            result = result + "]";
            return result;
        }
```

Apartado 1.3 método addPerson (0,75 puntos)

Rúbrica:

- 0 si no tiene sentido.
- 0,2 por comparar maxCapacity con el tamaño del ArrayList
- 0,1 por imprimir el mensaje cuando no se puede añadir. Si lo hacen lanzando una excepción y tanto la sentencia de lanzar la excepción como la declaración del método son correctas considerar correcto también.
- 0,2 por añadir correctamente el objeto al ArrayList
- 0,15 por incrementar numPersons y totalNumPersons
- 0,1 por devolver correctamente el valor

Solución:

```
public boolean addPerson(Person person){

    boolean result = true;

    if (maxCapacity == people.size()){
        System.out.println("Maximum capacity reached");
        result=false;
    }
    else{
        people.add(person);
        numPersons++;
        totalNumPersons++;
    }

    return result;

}
```

Apartado 1.4 Método testAddPerson (1 punto)

Rúbrica:

- 0 si no tiene sentido.
- 0,2 por la anotación @Test
- 0,2 por instanciar un objeto Person
- 0,2 por instanciar un objeto ShoppingCenter con capacidad suficiente
- 0,4 por tener el assertEquals correcto.

Solución:

```java
class ShoppingCenterTest {
    @Test
    public void testAddPerson() {
        Person person = new Person("Pedro", "Perez", "0000");
        ShoppingCenter shoppingCenter = new ShoppingCenter("Shopping center", 100, new ArrayList<Person>());
        assertEquals(shoppingCenter.addPerson(person), true);
    }
}
```

Problema 2. (1,5 puntos)

Apartado 2.1. Método numOfOcurrences de la clase DLinkedList (1,5 puntos)

Rúbrica

- 0 si no tiene sentido
- 0,25 inicialización y devolución del tipo de retorno
- 0,5 inicialización y actualización de current para recorrer la lista
- 0,5 condiciones correctas del bucle while
- 0,25 if correcto
    - 0,15 condición
    - 0,1 incrementar el numero de ocurrences.

Solución

```java
public int numOfOcurrences(E info) {
    int result = 0;
    DNode<E> current = first.getNext();
    while (current != last) {
        if (current.getInfo().equals(info) && current.getInfo() != null) {
            result++;
        }
        current = current.getNext();
    }
    return result;
}
```

Problema 3 (2,5)

Apartado 3.1. Método numOfOrders de la clase StoreManager (1 punto)

Rúbrica

- 0 si no se hace recursivo
- 0,25 inicialización y devolución del tipo de retorno
- 0,25 caso árbol vacío
- 0,5 caso recursivo
    - 0,2 cuenta el nodo actual
    - 0,3 llamadas recursivas

Solución

```java
public int numOfOrders(BSTree<InvoicedOrder> tree) {
    int result = 0;

    if (tree.isEmpty()) {
        result = 0;
    } else {
        result = 1 + numOfOrders(tree.getLeft())
                   + numOfOrders(tree.getRight());
    }

    return result;
}
```

Apartado 3. 2. Método insertionSort de la clase StoreManager (1,5 puntos)

Rúbrica

- 0 si no tiene sentido
  - 0,1 en total si han memorizado el método de las transparencias pero no lo adaptan al ejercicio.
- 0,25 Bucle externo (inicialización, condición y actualización)
- 0,25 guardar elemento i en tmp
- 0,75 Bucle interno
  - 0,25 inicialización, condición y actualización de j
  - 0,25 Llamada correcta a compareTo con < en el orden adecuado y llamada a get
  - 0,25 llamada a set dentro del bucle
- 0,25 actualización del valor de j con set.

Solución

```java
public void insertionSort() {

    for (int i = 0; i < list.size(); i++) {
        InvoicedOrder tmp = list.get(i);
        int j = i;
        while (j > 0 && tmp.compareTo(list.get(j - 1)) > 0) {
            list.set(j, list.get(j - 1));
            j--;
        }
        list.set(j, tmp);
    }
}
```