NOMBRE:
APELLIDOS:
NIA:
GRUPO:

2ª Parte: Problemas (7 puntos sobre 10)

Duración: 180 minutos
Puntuación máxima: 7 puntos
Fecha: 19 Mayo 2015

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
- Utiliza el espacio de los recuadros en blanco para responder a cada uno de los apartados de los problemas
- **NO SE PERMITE DESGRAPAR HOJAS DEL EXAMEN**

Problema 1 (2 / 7 puntos)

Se quiere crear una aplicación en Java para la gestión de medicamentos en farmacias. Para ello, comenzaremos por modelar los medicamentos. Se dispone de una interfaz `Drug`, que proporciona métodos para conocer el nombre del medicamento y su principio activo. La interfaz permite a los diferentes laboratorios (`LabA` y `LabB`) tener sus propias implementaciones de los medicamentos bajo una interfaz común.

```
interface Drug {
   String getName();
   String getActiveIngredient();
}
```

Rellene las clases `LabADrug`, `GenericDrug`, `NonGenericDrug` y `LabBDrug`, teniendo en cuenta las siguientes indicaciones:

- La clase `LabADrug` implementa la interfaz `Drug` y:
  - Tiene un único atributo, `activeIng`, de tipo `String`, que representa el principio activo. El valor de dicho atributo debe ser accesible desde cualquier clase, pero no puede ser modificado desde ninguna clase.
  - Tiene un único constructor que recibe el parámetro necesario para inicializar su atributo.
  - El nombre del medicamento dependerá del tipo concreto de medicamento, y esta clase no dispone de información suficiente para proporcionarlo.
- La clase `GenericDrug` hereda de la clase `LabADrug` y:
  - No tiene ningún atributo adicional.
  - Tiene un único constructor que recibe el parámetro necesario para inicializar su atributo.
  - El nombre de un medicamento genérico es el nombre del ingrediente activo seguido de un espacio y `"LabA EFG"`.
- La clase `NonGenericDrug` hereda de la clase `LabADrug` y:
  - Tiene un atributo adicional, `name`, de tipo `String`, que almacena el nombre del medicamento. Este atributo no se puede modificar desde ninguna otra clase.
  - Contiene un constructor que recibe los parámetros necesarios para inicializar sus atributos.
- La clase `LabBDrug` implementa la interfaz `Drug` y:
  - Tiene dos atributos: `name`, de tipo `String`, para representar el nombre del medicamento; e `ingredients`, un array de elementos de tipo `String`, para almacenar la lista de ingredientes. El primer ingrediente de la lista es siempre el principio activo. Ninguno de los dos atributos se puede modificar desde ninguna otra clase.
  - Contiene un constructor que recibe los parámetros necesarios para inicializar sus atributos.

Ningún atributo puede recibir el valor `null`, y en el caso de los `Strings`, tampoco cadena vacía (`""`). Si se intenta asignar alguno de estos valores, se lanzará una excepción de tipo `IllegalArgumentException`. En el caso concreto del atributo `ingredients`, puede asumir que ninguna de las posiciones del array con el que se inicialice el atributo será nula, que no estará vacío y que el contenido de todas las posiciones será siempre válido.

Tenga en cuenta que los huecos que encontrará a continuación podrán O NO tener código.

```
public                    class LabADrug                                    {

              String activeIng;

   public LabADrug(                                                        ) {









   }









}
```

```
public                  class GenericDrug                                    {

   public GenericDrug(                                                     ) {









   }









}
```

```
public                  class NonGenericDrug                                    {

              String name;

   public NonGenericDrug(                                                    ) {









   }









}
```

```
public                  class LabBDrug                                          {

               String name;

               String[] ingredients;

   public LabBDrug(                                                          ) {









   }









}
```

Problema 2 (3,75 / 7 puntos)

Para vender un medicamento cuyo principio activo es A, se buscan los medicamentos disponibles en la farmacia cuyo principio activo sea A, y se vende el más apropiado según las reglas de la farmacia. El objetivo de nuestra aplicación es mejorar la gestión en dos aspectos:

- Buscar de forma rápida los medicamentos cuyo principio activo sea uno dado.
- Evitar que los medicamentos caduquen. Para ello, se deberán vender primero los medicamentos que llegaron antes a la farmacia.

Además de la interfaz `Drug` previamente especificada, necesitaremos 3 clases adicionales para completar el programa que queremos implementar (puede consultar todas ellas en el apéndice):

- Una clase `Batch`, que representa un lote (conjunto de cajas) de medicamentos. Cada lote está compuesto por una referencia al medicamento (`Drug drug`) y la cantidad (>0) de unidades de dicho medicamento que contiene el lote (`int units`).
- Una clase `BatchQueue`, que representa una cola de lotes (`Batch`) almacenados por orden de llegada.
- Una clase `DrugStore`, que representa el almacén de la farmacia. La farmacia almacena los lotes de medicamentos que le llegan en diferentes colas de lotes (`BatchQueue`). Cada cola almacenará lotes de medicamentos con el mismo principio activo.

Para conseguir los objetivos de nuestra aplicación, se ha diseñado de la siguiente forma:

- Para evitar que los medicamentos caduquen, dado un principio activo, deberá venderse primero el medicamento del lote (`Batch`) que primero entró en el almacén. Para ello utilizaremos una cola (`BatchQueue`), que almacena los lotes (`Batch`) de medicamentos con un mismo principio activo, según su orden de llegada al almacén.
- Para poder buscar los medicamentos de forma eficiente usando como clave su principio activo, utilizaremos un árbol de búsqueda binario (`LBSTree<BatchQueue>`) donde los nodos del árbol están ordenados por principio activo.

Para poder conseguir simultáneamente los dos objetivos de gestión planteados, se ha creado en la clase `DrugStore` un atributo, `storage`, de tipo `LBSTree<BatchQueue>` que representa un árbol de búsqueda binaria, cuyos nodos almacenan como clave un principio activo, y como información una cola (`BatchQueue`) que almacenan lotes (`Batch`) de medicamentos con el principio activo de la clave.

Por **ejemplo**, suponga dos principios activos: PARACETAMOL e IBUPROFENO.

- Podemos encontrar dos medicamentos (objetos del tipo `Drug`) cuyo principio activo es PARACETAMOL: Frenadol y Gelocatil.
- Podemos encontrar dos medicamentos (objetos de tipo `Drug`) cuyo principio activo es IBUPROFENO: Espidifen y Neurofen.
- De cada uno de los medicamentos han llegado a la farmacia diferentes lotes (objetos de tipo `Batch`, representados mediante Nombre(número de cajas)) en el siguiente orden: Frenadol(5), Gelocatil(4), Espidifen(3), Frenadol(2), Neurofen(3).
- Para gestionar los lotes, habrá dos colas (una por cada principio activo), que contendrán:
  - Cola de PARACETAMOL: (front) Frenadol(5) → Gelocatil(4) → Frenadol(2)
  - Cola de IBUPROFENO: (front) Espidifen(3) → Neurofen(3)
- Cada una de estas colas estará almacenada en el nodo del árbol cuya clave de búsqueda sea el principio activo de los medicamentos que guarda la cola.

En el apéndice se muestran las clases que tiene a su disposición para desarrollar este programa. Escriba SÓLO el código de los métodos que se piden a continuación. Suponga que el resto de métodos están correctamente implementados. Puede utilizar los métodos que aparecen en el apéndice y los que se piden en los apartados del examen, NINGÚN OTRO.

Apartado 1 (1 / 7 puntos)

Para implementar una venta de un medicamento (con el método `public void sell(String ai)` de la clase `DrugStore` del apéndice), se necesita un método auxiliar. Dada la clase **`DrugStore`** del apéndice, implemente el método **`private static void sell(BatchQueue batchQueue)`**, que realiza una venta dada la cola de lotes de medicamentos `batchQueue`. Para ello, hay que disminuir una unidad la cantidad de unidades del lote que primero llegó a la cola (y que por tanto caducará antes). Si al finalizar esta operación no quedasen elementos en el lote, será necesario eliminar dicho lote de la cola. Si la cola está vacía o es `null`, se lanzará una excepción de tipo `IllegalArgumentException`.

Apartado 2 (1,5 / 7 puntos)

Dada la clase **`BatchQueue`** del apéndice, implemente el método **`public void clean()`**. Este método es necesario como medida de seguridad ante la posibilidad de que se se inserten lotes vacíos en la cola. Por tanto, el método eliminará de la cola todos aquellos lotes cuya cantidad de medicamentos sea cero. Preste especial atención al caso en el que la cola esté vacía. Tenga en cuenta también que puede ser necesario eliminar elementos del principio de la cola, del medio de la cola y/o del final de la cola.

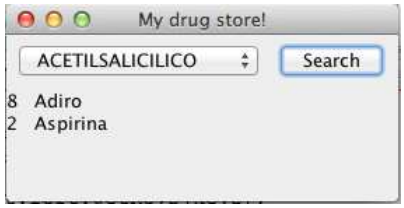
Apartado 3 (1,25 / 7 puntos)

Dada la clase **`LBSTree<E>`** del apéndice, implemente y use el método **`public void fillKeysList(ArrayList<String> list)`**. Este método rellena la lista `list` que recibe por parámetro con los Strings que se usan como claves de búsqueda del árbol **en orden alfabético**, o no hace nada si el árbol está vacío. Si `list` es `null`, este método lanzará una excepción de tipo `IllegalArgumentException`.

NOTA. Puede usar el método `public add(E e)` de la clase `ArrayList`, que inserta el elemento `e` al final de la lista.

Problema 3 (1,25 / 7 puntos)

La clase `DrugStoreGUI` gestiona la interfaz gráfica de la aplicación, que se muestra a continuación.



Rellene los huecos para conseguir la apariencia mostrada en la imagen superior y para que cuando se pulse el botón `Search` se muestren los 5 primeros lotes que llegaron al almacén cuyo principio activo sea el seleccionado en el desplegable. Si hay menos de 5 lotes, no se mostrará nada en los espacios reservados para tal fin. Tenga en cuenta que el desplegable y botón están colocados en la parte norte de la ventana, mientras que la información de los lotes está situada en la parte central de la ventana.

Deberá rellenar <u>todos</u> los huecos (espacios subrayados) ya que todos tienen código asociado.

```
public class DrugStoreGUI extends JFrame _____________________________________{
   private DrugStore storage;
   private JComboBox<String> activeIngredientList; //List active ingredients
   private JLabel[] labels;

   public DrugStoreGUI(DrugStore storage) {

      ____________________________________________________________________
      this.storage = storage;
      this.setDefaultCloseOperation(_____________________________________);

      JPanel contentPane = (JPanel) this.getContentPane();

      contentPane.setLayout(_____________________________________________);

      //Setting up the box for available active ingredients...
      setupActiveIngredientsList(storage.availableActiveIngredients());

      ____________________ searchButton =_____________________________

      searchButton._____________________________________________________

      JPanel upperPanel = new JPanel();
      upperPanel.add(activeIngredientList);
      upperPanel.add(searchButton);

      JPanel mediumPanel = new JPanel();
      mediumPanel.setLayout(new GridLayout(5,1));
      labels = new JLabel[5];
      for (int i=0; i<5; i++) {
         labels[i] = new JLabel("-");
         mediumPanel.add(labels[i]);
      }
      ____________________.add(upperPanel, _____________________________);

      ____________________.add(mediumPanel, _____________________________);

      this.pack();
      this.setVisible(true);
   }
```

```
   private void setupActiveIngredientsList
                              (ArrayList<String> availableActiveIngredients) {
      //Setting up active ingredients list...
   }

   public void ____________________________________________________________{
      BatchQueue batches =
         storage.searchDrugs((String)activeIngredientList.getSelectedItem());

      for(int i=0; i<5; i++) {
         labels[i].setText("");
         if(batches!=null && i<batches.size()) {
            Batch currentBatch = batches.batchAt(i);
            labels[i]._____________________(currentBatch.getUnits() + "  "+
                                           currentBatch.getDrug().getName());
         }
      }
   }
}
```

Apéndice

```
class Batch {
    private Drug drug;
    private int  units;

    Batch(Drug drug, int units) { ... }
    Drug getDrug() { ... }
    int  getUnits() { ... }
    void sell() { ... } //Sell (decrement) 1 unit from the batch
    public String toString() { ... }
}
```

```
class BatchQueue {

   protected class Node {
      Batch info;
      Node next;
      Node(Batch datum, Node next) { ... }
   }

   private Node top; // if empty, first is null.
   private Node tail;  // if empty, last is null.
   private int size; // if empty, size is 0.

   public BatchQueue() { ... }
   public boolean isEmpty() { ... }
   public int size() { return this.size; }
   public void enqueue(Batch e) throws IllegalArgumentException { ... }
   public Batch dequeue() { ... } //return null if queue empty
   public Batch front() { ... } //return null if queue empty
   public void clean() { //Apartado 2 }
   public Batch batchAt(int index) { ... } //return null if index<0 >=size
}
```

```
public class LBSTree<E> {

   protected class LBSNode {
      E info;
      String key;
      LBSTree<E> right;
      LBSTree<E> left
      public LBSNode(String key, E info,
                     LBSTree<E> left, LBSTree<E> right) { ... }
   }

   private LBSNode root;

   public LBSTree(){ ... }
   public boolean isEmpty() { ... }
   public void insert(String key, E info) { ... }
   public E find(String key) { ... } //return null if key not found
   public void fillKeysList(ArrayList<String> list)
                            throws IllegalArgumentException { //Apartado 3  }
}
```

```
public class DrugStore {
   private LBSTree<BatchQueue> storage;

   public DrugStore() {
      storage = new LBSTree<BatchQueue>();
   }
   public void addBatch(Batch batch) { ... }
   public void sell(String ai) {
      BatchQueue queue = storage.find(ai);
      if (queue != null) { sell(queue); }
   }
   private static void sell(BatchQueue drugBatch) { //Apartado 1 }
   public ArrayList<String> availableActiveIngredients() {
      ArrayList<String> availableActiveIngredients = new ArrayList<String>();
      storage.fillKeysList(availableActiveIngredients);
      return availableActiveIngredients;
   }
   public BatchQueue searchDrugs(String activeIngredient) { ... }
}
```

Soluciones y criterios de corrección

Problema 1 (2 puntos)

No había necesidad de crear ningún método adicional a los mencionados en el enunciado, ya que incluso para cumplir con los requisitos de control de acceso bastaba con implementar la interfaz `Drug`.

Los errores derivados de errores anteriores también cuentan como errores (por ejemplo, si el atributo se declaró como protected y se accede a él desde una clase hija con super.nombreAtributo, pero se pedía acceder desde getNombreAtributo(), se contará como error).

- Clase LabADrug (0,55)

```
public abstract class LabADrug implements Drug {
    private String activeIngredient;

    public LabADrug(String activeIngredient) {
        if(activeIngredient == null || activeIngredient.equals(""))
            throw new IllegalArgumentException();
        this.activeIngredient = activeIngredient;
    }
    public String getActiveIngredient() {
        return this.activeIngredient;
    }
}
```

[A001] (+0,1) Clase abstracta.

[A002] (+0,1) Implementa interfaz Drug

[A003] (+0,05) Atributo privado

[A004] (+0,2) Constructor:

- (-0,1) si parámetros no adecuados
- (-0,1) si la comprobación de que activeIng no sea null o cadena vacía es inexistente o incorrecta (activeIng ==””)
- (-0,1) si no lanzan la excepción correctamente
- (-0,1) si no inicializan el atributo correctamente
- Si queda puntuación negativa, dejarla a 0.
- Si “throws IllegalArgumentException” aparece en la declaración del constructor, no se penaliza ya que, aunque no es necesario, el programa funcionaría correctamente.

[A005] (+0,1) Método getActiveIngredient correcto

[A006] (-0,1) Si aparece el método getName() sin que sea abstract. Dado que la clase implementa la interfaz Drug, pueden no poner getName(), pero si lo ponen tiene que ser abstract.

[A007] (-0,2) Si hay métodos o atributos adicionales (setters, atributo name…)

[A099] (-0,1) Si hay mal código: “throws IllegalArgumentException” en la declaración de clase, variables de métodos public o private, return en un constructor…)

- Clase GenericDrug (0,4)

```
public class GenericDrug extends LabADrug {
    public GenericDrug(String activeIngredient) {
        super(activeIngredient);
    }
    public String getName() {
        return this.getActiveIngredient() + " LabA EFG";
    }
}
```

[A008] (+0,1) extends LabADrug y no duplicar código de la clase padre

[A009] (+0,15) Constructor:

- (-0,1) si parámetros no adecuados
- (-0,1) si no usa super, -0.1
- (-0,05) si usa super, pero sin parámetro
- (-0.05) si no coloca super como primera línea del constructor
- Si queda puntuación negativa, dejarla a 0.

[A010] (+0,15) Método getName correcto.

- (-0,1) si acceden el ingrediente activo sin usar el método getActiveIngredient()
- (-0,1) si no se concatenan Strings
- Si queda puntuación negativa, dejarla a 0

[A011] (-0,2) Si hay métodos o atributos adicionales (setters, atributo name…)

[A099] (-0,1) Si hay mal código, como por ejemplo "throws IllegalArgumentException" en la declaración de clase, variables de métodos public o private, return en un constructor… o cualquier error similar.

- Clase NonGenericDrug (0,45)

```
public class NonGenericDrug extends LabADrug {
    private String name;
    public NonGenericDrug(String activeIngredient, String name) {
        super(activeIngredient);
        if (name == null || name.equals(""))
            throw new IllegalArgumentException();
        this.name = name;
    }
    public String getName() {
        return this.name;
    }
}
```

[A012] (+0,1) extends LabADrug y no duplicar código de la clase padre

[A013] (+0,05) Atributo privado

[A014] (+0,2) Constructor:

- (-0,1) si no usan super, -0.1
- (-0,05) si usa super, pero sin parámetro
- (-0,05) si no coloca super como primera línea del constructor
- (-0,1) si la comprobación de que name no sea null o cadena vacía es inexistente o incorrecta (name==””)
- (-0,1) si no lanzan excepción correctamente
- (-0,1) si no inicializan el atributo correctamente
- Si queda puntuación negativa, dejarla a 0.
- Si “throws IllegalArgumentException” aparece en la declaración del constructor, no se penaliza ya que, aunque no es necesario, el programa funcionaría correctamente.

[A015] (+0,1) Método getName correcto.

[A016] (-0,2) Si hay métodos o atributos adicionales (setters, atributo name…)

[A099] (-0,1) Si hay mal código, como por ejemplo “throws IllegalArgumentException” en la declaración de clase, variables de métodos public o private, return en un constructor… o cualquier error similar.

- Clase LabBDrug (0,6)

```
public class LabBDrug implements Drug {
    private String name;
    private String [] ingredients;

    public LabBDrug(String name, String[] ingredients) {
        if(name == null || ingredients == null || name.equals(""))
          throw new IllegalArgumentException();
        this.name = name;
        this.ingredients = ingredients;
    }
    public String getName() {
        return this.name;
    }
    public String getActiveIngredient() {
        return this.ingredients[0];
    }
}
```

[A017] (+0,1) implements Drug

[A018] (+0,05) Atributo privado

[A019] (+0,2) Constructor:

- (-0,1) si comprobación de si name es null o cadena vacía o si ingredients es null inexistente o incorrecta (name==””)
- (-0,1) si no lanzan excepción correctamente
- (-0,1) si no inicializan los atributos correctamente
- si queda negativo, dejarlo a 0.
- Si “throws IllegalArgumentException” aparece en la declaración del constructor, no se penaliza ya que, aunque no es necesario, el programa funcionaría correctamente.

[A020] (+0,1) Método getName

[A021] (+0,15) Método getActiveIngredient:

- (-0,1) si no retornan el contenido de la primera posición del array

[A022] (-0,2) Si hay métodos o atributos adicionales (setters, atributo name…)

[A099] (-0,2) Si hay mal código, como por ejemplo “throws IllegalArgumentException” en la declaración de clase, variables de métodos public o private, return en un constructor… o cualquier error similar.

Problema 2 (3,75 puntos)

Apartado 1 (1 punto)

```
//Solucion 1
private static void sell(BatchQueue batchQueue) {
   if (batchQueue == null || batchQueue.isEmpty()) {
      throw new IllegalArgumentException();
   }
   Batch oldestBatch = batchQueue.front();
   oldestBatch.sell();
   if (oldestBatch.getUnits() == 0) {
      batchQueue.dequeue();
   }
}

//Solucion 2
private static void sell(BatchQueue batchQueue) {
   if (batchQueue == null || batchQueue.isEmpty()) {
      throw new IllegalArgumentException();
   }
   if(batchQueue.front().getUnits() == 1) {
      batchQueue.dequeue();
   } else {
      batchQueue.front().sell();
   }
}
```

[B001] (+0,1) Lanzar excepción si cola null o vacía

- (-0,05) si sólo comprueban una de las dos cosas
- (-0,05) si no lanzan la excepción correctamente
- Si "throws IllegalArgumentException" aparece en la declaración del constructor, no se penaliza ya que, aunque no es necesario, el programa funcionaría correctamente.

[B002] (+0,5) Vender una unidad del primer lote de la cola (front):

- (-0.25) si no lo hacen sobre el lote que devuelve front() o batchAt(0) (es decir, si usan first, o xxx.info, o si no lo hacen sobre el primer elemento de la cola…)
- (-0.2) si no venden la unidad con el método sell() (units es atributo privado de Batch)
- (-0.3) si desencolan en lugar de usar front().
- Si queda negativo, dejar a 0.

[B003] (+0,4) si quedan 0 unidades del lote después de vender, o si queda 1 unidad del lote antes de vender, eliminar lote de la cola:

- (-0.1) si el if es incorrecto (usan batchQueue.info.units, batch.units, si consultan el size de la cola… en lugar de hacer batchQueue.front().getUnits() == 0 ó ==1, dependiendo del caso)
- (-0.3) si no desencolan con dequeue().

[B099] (-0,3) Si hay mal código, como por ejemplo "trhows IllegalArgumentException" en la declaración de clase, variables de métodos public o private, return en un constructor, condiciones mal construidas (getUnits() == true, isEmpty() != null), variables sin tipo o del tipo equivocado para el objeto o tipo primitivo que almacenan, llamadas a métodos que no son de la clase del objeto que los llama… o cualquier error similar.

Apartado 2 (1,5 puntos)

```
//Solucion 1
public void clean() {
   Node prev = null;
   Node current = top;
   while (current != null ) {
      if (current.info.getUnits() == 0) {
         if (prev == null) {
            top = current.next;
         } else {
            prev.next = current.next;
         }
         if (tail == current) {
            tail = prev;
         }
         size--;
      } else {
         prev = current;
      }
      current = current.next;
   }
}

//Solucion 2
public void clean() {
   int originalSize = this.size;
   for(int index=0; index < originalSize; i++) {
      Batch auxBatch = dequeue();
      if (auxBatch.getUnits() > 0) {
         enqueue(auxBatch);
      }
   }
}
```

[B004] (+0,25) Para listas vacías, no hacer nada.

[B005] (+0,25) Encontrar lotes vacíos (comprobar que info.getUnits() == 0).

[B006] (+0,25) Eliminar correctamente si el/los nodos están al principio de la cola (prestando atención a la actualización de top)

[B007] (+0,25) Eliminar correctamente si el/los nodos están en el medio de la cola.

[B008] (+0,25) Eliminar correctamente si el/los nodos están al final de la cola (prestando atención a la actualización de tail)

[B009] (+0,25) Decrementar size por cada elemento que se elimine.

[B010] Pueden hacerlo tanto recorriendo la cola como una lista enlazada, o desencolando y encolando, siempre y cuando el orden final de los lotes que queden en la cola sea el mismo que existía al principio (es decir, si se desencolan y vuelven a encolar todos los elementos)

- (-0,2) en cada uno de los apartados que se implementen de [B006],[B007],[B008] si no dejan el mismo orden de la cola original

[B099] (-0,5) Si hay mal código, como por ejemplo “trhows IllegalArgumentException” en la declaración de clase, variables de métodos public o private, return en un constructor, condiciones mal construidas (getUnits() == true, isEmpty() != null), variables sin tipo o del tipo equivocado para el objeto o tipo primitivo que almacenan, llamadas a métodos que no son de la clase del objeto que los llama… o cualquier error similar.

Apartado 3 (1,25 puntos)

```
//LBSTree<E> class
public void fillKeysList(ArrayList<String> list)
                                                  throws IllegalArgumentException {
   if (list == null){
      throw new IllegalArgumentException();
   }
   if(isEmpty()) { return; }
   root.left.fillKeysList(list);
   list.add(root.key);
   root.right.fillKeysList(list);
}
```

[B011] (+0,1) Lanzar excepción si list es null

[B012] (+0,25) Caso base: no hacer nada si el árbol está vacío (prestar atención a que esta condición puede ser literal como en la solución de arriba, o puede estar incluida en otras aproximaciones como comprobar que el subárbol derecho o izquierdo son vacíos y por tanto no se aplica recursión sobre ellos).

- (-0,2) si no comprueban correctamente si el árbol está vacío (usando, por ejemplo, root.info==null)

[B013] (+0,25) Recursión por la izquierda del árbol: -0.1 si no llaman al método recursivo sobre root.left o si lo hacen sobre root.getLeft(); -0.1 si no le pasan list como argumento.

[B014] (+0,2) Añadir la key del árbol actual a list: -0.1 si no llaman a add sobre list; -0.1 si no introducen root.key. Si queda negativo, dejar a 0.

[B015] (+0,25) Recursión por la derecha del árbol: -0.1 si no llaman al método recursivo sobre root.right o si lo hacen sobre root.getRight(); -0.1 si no le pasan list como argumento.

[B016] (+0,2) Si lo imprimen en inorder (bien sea left-arbol-right o right-arbol-left, ya que no se especifica nada en el enunciado).

[B099] (-0,4) Si hay mal código, como por ejemplo “trhows IllegalArgumentException” en la declaración de clase, variables de métodos public o private, return en un constructor, condiciones mal construidas (getUnits() == true, isEmpty() != null), variables sin tipo o del tipo equivocado para el objeto o tipo primitivo que almacenan, llamadas a métodos que no son de la clase del objeto que los llama… o cualquier error similar.

Si comprueban si left o right de root son o no null, no se penaliza, ya que en el enunciado no se indica explícitamente en ningún sitio si pueden ser null o van a ser siempre árboles vacíos o llenos. La idea era seguir la implementación de los laboratorios donde left y right sólo pueden ser árboles vacíos o llenos, nunca null. Pero si comprueban null lo único que ocurre es que nunca lo van a ser, por lo que el código funcionaría igualmente.

Problema 3 (1,25 puntos)

```
public class DrugStoreGUI extends JFrame implements ActionListener {

   private DrugStore storage;
   JComboBox<String> activeIngredientList;
   JLabel[] labels;

   public DrugStoreGUI(DrugStore storage) {
      super("My drug store!");
      this.storage = storage;

      this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      JPanel contentPane = (JPanel) this.getContentPane();
      contentPane.setLayout(new BorderLayout());

      setupActiveIngredientsList(storage.availableActiveIngredients());

      JButton searchButton = new JButton("Search");
      searchButton.addActionListener(this);

      JPanel upperPanel = new JPanel();
      upperPanel.add(activeIngredientList);
      upperPanel.add(searchButton);

      JPanel mediumPanel = new JPanel();
      mediumPanel.setLayout(new GridLayout(5,1));
      labels = new JLabel[5];
      for (int i=0; i<5; i++) {
         labels[i] = new JLabel("-");
         mediumPanel.add(labels[i]);
      }

      contentPane.add(upperPanel, BorderLayout.NORTH);
      contentPane.add(mediumPanel, BorderLayout.CENTER);

      this.pack();
      this.setVisible(true);
   }

   private void setupActiveIngredientsList
                           (ArrayList<String> availableActiveIngredients) {
      //Setting up active ingredients list...   }

   public void actionPerformed(ActionEvent e) {
      BatchQueue batches =
         storage.searchDrugs((String)activeIngredientList.getSelectedItem());
      for(int i=0; i<5; i++) {
         labels[i].setText("");
         if(batches!=null && i<batches.size()) {
            Batch currentBatch = batches.batchAt(i);
            labels[i].setText(currentBatch.getUnits()+"   "+
                              currentBatch.getDrug().getName());
         }
      }
   }
}
```

[C001] (+0,15) implements ActionListener

[C002] (+0,15) super(“My drug store!”);

[C003] (+0,125) JFrame.EXIT_ON_CLOSE

[C004] (+0,125) new BorderLayout()

[C005] (+0,05) JButton

[C006] (+0,1)new JButton(“Search”)

[C007] (+0) addActionListener(this) (este hueco no estaba en el examen de inglés debido a un error en la traducción, por lo que no se puntúa en ninguno)

[C008] (+0,075) contentPane (o this.getContentPane())

[C009] (+0,075) contentPane (o this.getContentPane())

[C010] (+0,075) BorderLayout.NORTH

[C011] (+0,075) BorderLayout.CENTER

[C012] (+0,125) actionPerformed(ActionEvent e)

[C013] (+0,125) setText