Laboratorio 4 – Árboles Binarios y Recorridos (v2)

Este laboratorio tiene cuatro objetivos fundamentales:

1. Añadir un constructor de copia sobre los árboles binarios para poder evitar, cuando sea necesario, los problemas provocados por la compartición de referencias a objetos mutables (como se ha comentado en el documento que está enlazado en el campus).
2. Implementar la estrategia de enhebrar en inorden los nodos de cara a poder implementar un iterador en inorden, sin necesidad de usar una pila.
3. Implementar los iteradores en inorden, usando la enhebración de los nodos del apartado anterior.
4. Implementar los iteradores por niveles, usando una cola de nodos.

Estructura del proyecto

Para esta actividad os proporcionaremos un proyecto de partida. con un conjunto de test, que deberéis ampliar para comprobar el buen funcionamiento de vuestra solución.

Las clases de test del proyecto que os proporcionamos son:

- `AbstractLinkedBinaryTreeTest`: define dos instancias de árbol (que se recrearán antes de ejecutar cada test, ya que son variables de instancia) y un método para simplificar los test de los iteradores.
- `LinkedBinaryTreeTest`: define los test de los métodos que os damos implementados. Vuestra solución no debería romper ninguno de estos test.
- `CopyTest`: clase de test vacía en la que colocaréis los test del primer apartado sobre la copia de árboles.
- `InOrderIteratorTest`: clase de test vacía en la que colocaréis los test sobre los iteradores en inorden.
- `LevelOderIteratorTest`: clase de test vacía en la que colocaréis los test sobre los iteradores por niveles.
- `ThreadingEffectTest`: test que pone de manifiesto los efectos de la enhebración sobre el hijo izquierdo de un árbol. Estos test deberían pasar cuando se hayan solucionado los tres primeros apartados de la práctica.

El proyecto tiene líneas marcadas con comentarios de la forma:

```
// TODO: Exercise 2
```

Estos comentarios sirven pues IntelliJ tiene una ventana en la que aparecen todos los TODOs (y FIXMEs) que hay en el proyecto.

Para ver esta ventana, podéis pulsar dos veces la tecla de mayúsculas, entrar todo en el buscador y os aparecerá la acción que os abre esta ventana. Conforme vayáis resolviendo la prácticas iréis borrando estos comentarios de manera que, al acabar un apartado, ya no deberían quedar tareas pendientes sobre él.

Obviamente, esta funcionalidad también os puede ser de mucha utilidad en vuestros proyectos.

Tarea 1: Implementación de un constructor de copia

Partiremos de una simplificación, para que así solamente os tengáis que preocupar de los métodos más fundamentales, de la interfaz BinaryTree<E> y de su implementación usando nodos enlazados.

La interfaz BinaryTree<E> sobre la que trabajaréis será:

```
public interface BinaryTree<E> {
    BinaryTree<E> left();
    BinaryTree<E> right();
    E root();
    int size();
    boolean isEmpty();
    BinaryTreeIterator<E> inOrderIterator();
    BinaryTreeIterator<E> levelOrderIterator();
}
```

Y de su implementación en forma de nodos enlazados LinkedBinaryTree<E>.

La primera tarea que realizar será añadir un nuevo constructor que nos haga una copia del árbol que le pasaremos como parámetro. De esta manera, los dos árboles no compartirán nodos, pero sí que compartirán referencias a los valores a los que apuntan dichos nodos. La signatura de dicho constructor será:

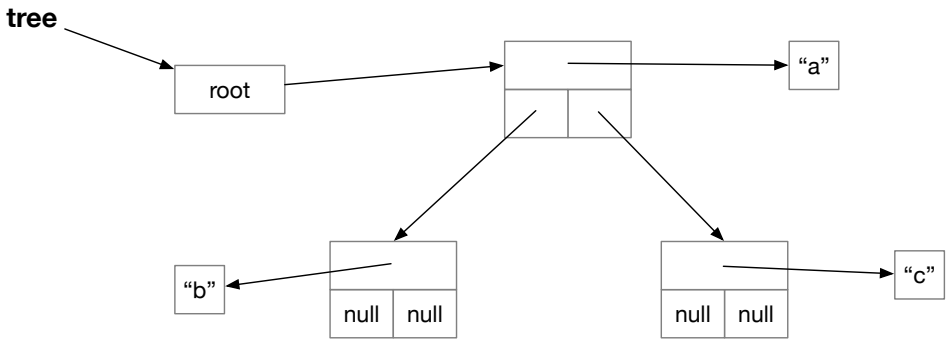
```
public LinkedBinaryTree(LinkedBinaryTree<E> tree) { ¿? }
```

Como siempre, siguiendo la estrategia de implementación que hemos utilizado en la implementación de la clase, dicho constructor usará una operación (recursiva) que tendremos en la clase interna nodo:

```
static <E> Node<E> copy(Node<E> node) { ¿? }
```

Operación que devolverá una copia del árbol de nodos cuya raíz es el nodo pasado (y que puede ser null cuando el árbol está vacío).

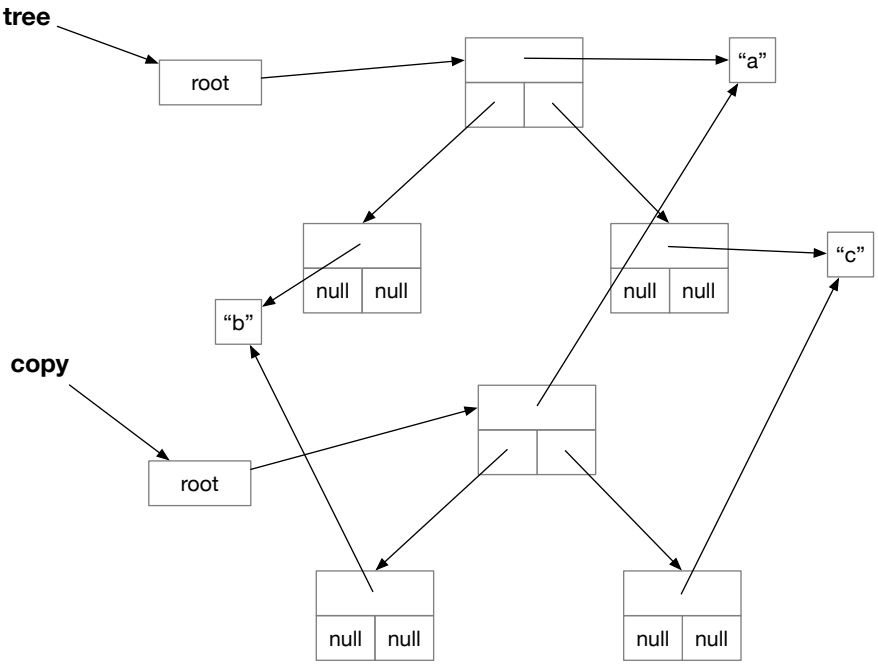
Por ejemplo, dado el árbol de strings referenciado por tree (en el que he omitido atributos no relevantes para lo que nos ocupa como el size):



Si hacemos su copia, es decir:

```
var copy = new LinkedBinaryTree<>(tree);
```

El árbol que obtenemos será:



Consideraciones sobre el testing

Como ya se ha comentado otras veces, los test deben comprobar el funcionamiento de las operaciones dada su especificación, no cómo éstas están implementadas interiormente.

En el caso de la copia, tenemos un problema: ¿cómo detectamos que lo que se nos devuelve es realmente una copia y que los nodos no están compartidos?

Con las operaciones que tenemos implementadas, podemos comprobar que la copia que recibimos es igual (equals) al árbol de partida, pero eso no nos garantiza que se nos haya devuelto una copia independiente (a nivel de nodos).

De hecho, esto es perfectamente normal: si no hay posibilidad de mutar un árbol, no hay manera de construir un comportamiento en el que el original y la copia se comporten de forma diferente.

Para ello, hemos añadido a la interfaz BinaryTree<E> un método que modifique el valor contenido en la raíz del árbol (y que lanzará NoSuchElementException en caso de que el árbol esté vacío).

En el informe, además de explicar el diseño de la operación de copia (ayudándoos de diagramas), deberéis explicar los test que habéis realizado para comprobar que este apartado funcione correctamente.

Estos test se implementarán en la clase `CopyTest` y podrán usar toda la infraestructura de `AbstractLinkedBinaryTreeTest` ya que la extienden.

Tarea 2: Enhebración simple en inorden

Como paso previo a la implementación de los iteradores en inorden, modificaréis la implementación de los árboles, para hacer que los nodos estén enhebreados simplemente en inorden.

Esta estrategia consiste en que aprovecharemos el apuntador al hijo derecho para, en caso de que el nodo no tenga hijo derecho, apuntar al nodo siguiente en inorden. De esa manera podremos implementar el recorrido en inorden siguiendo esos apuntadores (y la estructura normal del árbol), sin necesidad de recursividad ni de la ayuda de una pila.

Por ello, la definición de la clase Node<E> cambiará a:

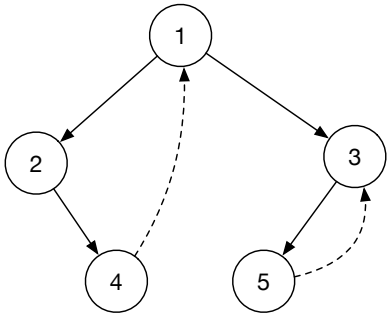
```
private static class Node<E> {
    Node<E> left;
    E element;
    Node<E> rightOrNext;
    int size;
    boolean isRightChild;

    // ...
}
```

El significado de los campos es el siguiente:

- `left`: apuntador al hijo izquierdo (null en caso de que sea vacío)
- `element`: apuntador al valor almacenado en el nodo
- `isRightChild`: booleano que nos indica si el campo rightOrNext se corresponde o no con el hijo derecho
- `rightOrNext`: apuntador al hijo derecho (cuando el booleano vale true) o bien, porque su hijo derecho es vacío, al siguiente nodo en inorden
    - NOTA: de hecho, en ambos casos, el nodo apuntado es el siguiente nodo en inorden
- `size`: tamaño del árbol enraizado en el nodo

Si consideramos el siguiente árbol, cuyo recorrido en inorden es 2, 4, 1, 5, 3:



tenemos:

- el nodo 1 que mantiene apuntadores a sus dos hijos (como en un árbol "normal")
- el nodo 2 que tiene un hijo izquierdo vacío y cuyo hijo derecho es el árbol enraizado en 4
- el nodo 3 que tiene como hijo izquierdo el árbol enrazado en 5 y su hijo derecho es vacío (ya que no tiene un siguiente en inorden)
- el nodo 4, que es una hoja, pero cuyo enlace derecho apunta al siguiente nodo en inorden.
- El nodo 5, que es también una hoja, y cuyo enlace derecho apunta al siguiente nodo en inorden.

Para conseguir esta funcionalidad, deberéis modificar convenientemente el constructor de la clase Node<E>:

```
Node(Node<E> left, E element, Node<E> right) { }
```

Y, como ahora el enlace al hijo derecho a veces no apunta al hijo derecho, para acceder a dicho hijo es conveniente usar el método de acceso:

```
Node<E> right() {
    return isRightChild ? rightOrNext : null;
}
```

Sobre el diseño

En este apartado se valorará especialmente la descripción del diseño de la solución y cómo explicáis el proceso de construcción y enhebración del árbol usando diagramas. Y el comentario de las modificaciones que habéis realizado del código existente para que continúe funcionando.

Sobre el testing

El comportamiento externo de la clase será el mismo, por lo que en este apartado solamente nos deberemos de preocupar de no haber roto nada de lo que antes funcionaba.

Como los iteradores en inorden usarán la enhebración, serán estos los que nos permitirán comprobar que ésta se ha realizado correctamente.

Tarea 3: Iteradores en inorden

En este apartado implementaremos los iteradores en inorden, de forma iterativa, usando la enhebración que se ha hecho en la tarea anterior.

Para ello implementaréis la clase interna InOrderIterator

```
private class InOrderIterator implements
BinaryTreeIterator<E> {

    private Node<E> next;
    private Node<E> lastReturned;

    ¿?
}
```

Una pequeña pista sobre la implementación: la clase guardará dos referencias a nodo:

- `next`: nodo que contiene el siguiente elemento a retornar por el iterador
- `lastReturned`: último retornado por next

No consideraremos en este caso ningún tipo de modCount (eso sí, podéis pensar qué operación, u operaciones, tiene sentido que invaliden la iteración realizada sobre los árboles que habéis implementado).

Sobre el diseño

En este apartado el diseño también deberá ser explicado en base a diagramas que muestren todos los aspectos de creación y manejo de los iteradores definidos.

Sobre el testing

Al hacer pruebas sobre el funcionamiento correcto del recorrido en inorden, estaréis comprobando también que la enhebración de la tarea anterior se ha realizado correctamente. Eso sí, ante un test que falle, como la enhebración no se ha testeado hasta este momento, deberéis considerar que:

- puede estar fallando el diseño/implementación de la operación con iteradores
- puede estar fallando el diseño/implementación de la enhebración
- pueden fallar las dos cosas a la vez

Como en la práctica anterior, os puede ser de utilidad ejecutar los test con cobertura para detectar qué casos no han quedado cubiertos por ninguna prueba.

Estos test se implementarán en la clase `InOrderIteratorTest` y podrán usar toda la infraestructura de `AbstractLinkedBinaryTreeTest` ya que la extienden.

Además, una vez implementadas las tres primeras tareas, los test existentes en la clase `ThreadingEffectTest` también deberían pasar.

Tarea 4: iteradores por niveles

La cuarta tarea consistirá en implementar otro recorrido del árbol con iteradores: la iteración por niveles. Esta iteración, de forma natural, se implementa de forma iterativa usando una **cola de nodos no nulos** que se usa para, por un lado, obtener el siguiente nodo del recorrido y, por otro, para añadir a sus hijos (no nulos) a la cola de nodos pendientes.

Para representar esta cola, usaremos la estructura Deque, ya en la librería estándar de Java, cuya documentación es:

https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/Deque.html

Como implementación, podéis usar tanto:

- ArrayDeque (implementación en forma de array circular ampliable[1])
  https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/ArrayDeque.html
- LinkedList (implementación en forma de nodos doblemente enlazados, como ya sabéis)
  https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/LinkedList.html

Los métodos que usaremos serán los que lanzan excepciones en caso de error, es decir:

- `addLast`, para añadir un elemento a la cola
- `removeFirst`, para eliminar el primer elemento de la cola
- `getFirst`, para acceder al primer elemento de la cola

Es decir:

```
private class LevelOrderIterator
        implements BinaryTreeIterator<E> {

    private final Deque<Node<E>> queue;
    private Node<E> lastReturned;

    ¿?
}
```

Sobre el diseño

Como siempre, se valorará una correcta explicación del diseño, ayudándoos de diagramas, fragmentos de código (que es texto, no imagen), etc.

Sobre el testing

Estos test se implementarán en la clase `LevelOrderIteratorTest` y podrán usar toda la infraestructura de `AbstractLinkedBinaryTreeTest` ya que la extienden.

Otras consideraciones

No olvidéis compilar usando la opción –Xlint:unchecked, para que el compilador os avise de errores en el uso de genéricos. Prestad a atención a los mensajes que aparezcan al ejecutar.

[1] La implementación de esta clase es muy interesante como ejemplo de manipulación y manejo de subarrays (prefijos, infijos, sufijos).

Entrega

El resultado obtenido debe ser entregado por medio de un **proyecto IntelliJ** por cada grupo, comprimido (en formato **ZIP**) y con el nombre "**Lab4_NombreIntegrantes.zip**".

Además, se debe entregar un documento de texto (en formato **PDF**) en que se documenten los diseños e implementaciones de los apartados realizados.

Consideraciones de Evaluación

A la hora de evaluar el laboratorio se tendrán en cuenta los siguientes aspectos:

- El código ofrece una solución al problema planteado (3 puntos)
- Realización de test con JUnit 5 (2,5 puntos)
- Explicación adecuada del trabajo realizado, es decir, esfuerzo aplicado al documento de texto solicitado (3 puntos)
- Calidad y limpieza del código, formato de las entregas, etc. (1,5 puntos)