Laboratorio 4 – Persistent Tree (v1)

Este laboratorio tiene dos objetivos fundamentales:

1. Realizar una implementación de árboles binarios de búsqueda.
2. Implementar recorridos sobre árboles binarios.

El contenido de esta práctica forma parte del temario del segundo parcial y éste podrá contener preguntas sobre ella.

Apartado 1: Implementación de los árboles binarios de búsqueda inmutables

Inmutabilidad y persistencia

En este laboratorio trabajaremos con **árboles binarios de búsqueda** (ABB), concretamente, con implementaciones **inmutables (o persistentes)** de las mismas. ¿Qué queremos decir con ello?

- **Inmutable**: una vez creado un árbol, éste no puede ser modificado. Es por ello por lo que **las operaciones de modificación devuelven un nuevo árbol**, con las modificaciones necesarias sobre el árbol original.
- **Persistente**: al crear el árbol modificado, **no se destruye el árbol de partida**, por lo que las referencias al árbol original se refieren al árbol antes de las modificaciones.

Las estructuras de datos que hemos visto hasta el momento no son persistentes, por ejemplo:

```
List<Integer> li = new LinkedList<>();
li.add(2);
List<Integer> li2 = li;
li.add(3);
```

Aunque hayamos "capturado" la lista antes del segundo `add`, la referencia `li2` se refiere a la misma lista que `li` y, por tanto, los cambios realizados sobre `li` son visibles desde `li2`.

Si quisiéramos que `li2` se refiriera a la lista antes del segundo `add`, deberíamos **haber hecho una copia** de la lista, es decir:

```
List<Integer> li = new LinkedList<>();
li.add(2);
List<Integer> li2 = new LinkedList<>(li);
li.add(3);
```

Es decir, como trabajamos con estructuras de datos **mutables**, hemos de **hacer copias** si queremos poder volver a acceder al valor que tenía la estructura antes de hacer una modificación.

Fijaos en que si, por ejemplo, trabajáramos con `Strings`, al ser estos una estructura de datos inmutable, jamás podríamos modificar la cadena al añadir un carácter. En cambio, podríamos construir un nuevo `String`, a partir del original, añadiéndole nuevos caracteres (por lo que, si no perdemos su referencia, siempre podremos acceder al valor original del mismo):

```
String original = "patata";
String modified = original + "s";
```

Estas estructuras inmutables son utilizadas en el paradigma de **programación funcional** y también en **programación concurrente**, ya que, al no poderse modificar, pueden ser accedidas concurrentemente por varios hilos de ejecución[1].

Árboles binarios de búsqueda persistentes

Por lo tanto, en nuestro caso, un ABB **no podrá ser modificado** y, por ello, las operaciones que realicen una "modificación" sobre el ABB (inserción y eliminación) deberán **devolver un nuevo ABB**, y **no alterarán aquél sobre el que se apliquen**.

Tal y como se ha visto en el tema 4, los ABBs ofrecen una estructura de datos utilizada para asociar valores a claves para, posteriormente, recuperar el valor a partir de la clave. Se definen como:

- Un árbol binario vacío.
- Un árbol binario donde:
  - Cada nodo contiene una pareja (clave, valor).
  - No hay dos nodos con el mismo valor de la clave.

[1] Para que esto sea realmente cierto, no sólo el ABB debe ser inmutable, sino que los elementos que contenga también deben serlo. Este punto, cierto en el paradigma de programación funcional, no puede ser comprobado por el compilador de Java, y es una de las razones por las que la programación funcional cada día es más importante.

- La clave de la raíz es mayor[2] que todas las claves del subárbol (hijo) izquierdo.
- La clave de la raíz es menor que todas las claves del subárbol (hijo) derecho.
- El subárbol izquierdo es un ABB.
- El subárbol derecho es un ABB.

Las siguientes figuras, muestran un ejemplo de ABB (Figura 1) y otro que no lo es (Figura 2), ya que el nodo con clave 35 no puede pertenecer al subárbol derecho del nodo con clave 40:

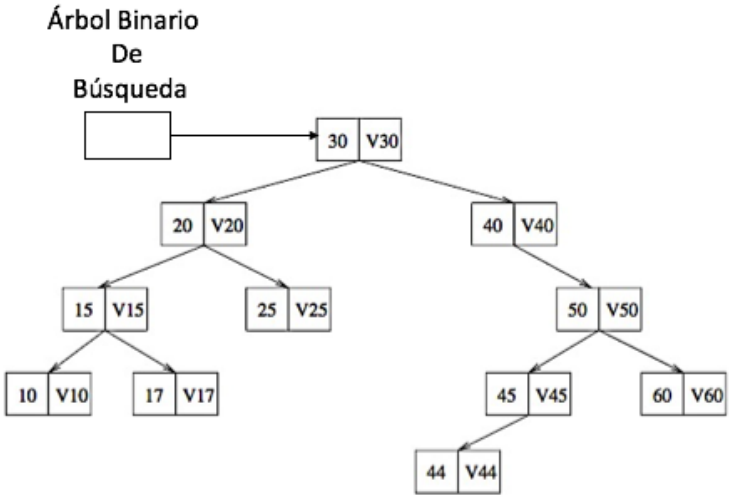

Figura 1. Ejemplo ABB

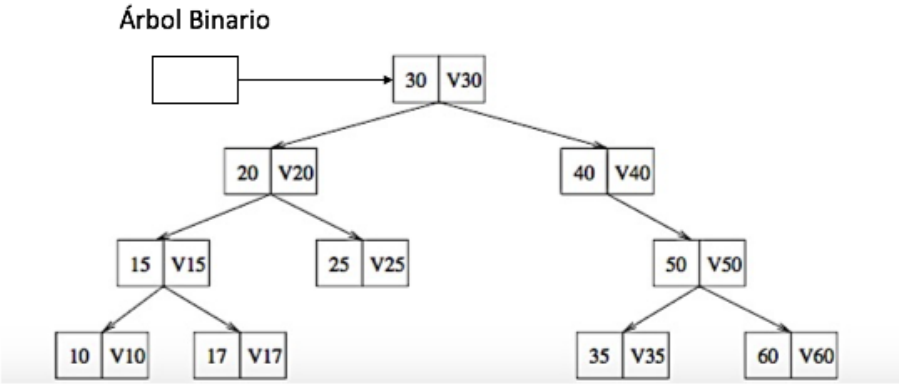

Figura 2. Ejemplo de árbol binario que no es de búsqueda

[2] Al no haber claves repetidas las comparaciones son estrictas.

BinarySearchTree<E>

Esta interfaz declarará las operaciones que podremos realizar sobre los árboles binarios de búsqueda inmutables:

```
public interface BinarySearchTree<K, V> {
    boolean isEmpty();
    boolean containsKey(K key);
    V get(K key);
    BinarySearchTree<K, V> put(K key, V value);
    BinarySearchTree<K, V> remove(K key);
}
```

Fijaos en que las operaciones `put` y `remove` devuelven un *`BinarySearchTree<K, V>`*, que es el árbol resultado de hacer la modificación.

Sus operaciones se comportan de la siguiente manera:

- *`isEmpty()`* devuelve *`true`* si el ABB está vacío, y *`false`* en caso contrario.
- *`containsKey(K key)`* devuelve *`true`* si el ABB contiene la clave *`key`*.
    - Si *`key`* es *`null`* lanza *`NullPointerException`* (excepción no comprobada predefinida).
- *`get(K key)`* devuelve el valor asociado a la clave *`key`*.
    - Si *`key`* es *`null`* lanza *`NullPointerException`* (excepción no comprobada predefinida).
    - Si no existe *`key`* en el ABB, lanza *`NoSuchElementException`* (excepción no comprobada predefinida).
- *`put(K key, V value)`* devuelve un nuevo ABB basado en aquel desde el que se realiza la llamada, pero añadiendo la asociación de *`key`* a *`value`*.
    - Si *`key`* o *`value`* son *`null`* lanza *`NullPointerException`* (excepción no comprobada predefinida).
- *`remove(K key)`* devuelve un nuevo ABB basado en aquel desde el que se realiza la llamada, pero eliminando la asociación establecida con la clave *`key`*.
    - Si *`key`* es *`null`* lanza *`NullPointerException`* (excepción no comprobada predefinida).

Clase LinkedBinarySearchTree<E>

Esta será la clase que implementará el árbol binario de búsqueda por medio de la interfaz anterior (cumpliendo las especificaciones de las operaciones) **usando nodos enlazados**.

El ABB **no deberá equilibrarse**, con lo que las operaciones de búsqueda, inserción y eliminación no serán necesariamente logarítmicas.

La estructura general de la clase *LinkedBinarySearchTree<K, V>* podría ser la siguiente[3]:

```
public class LinkedBinarySearchTree<K, V>
        implements BinarySearchTree<K, V> {

    private final Node<K, V> root;
    private final Comparator<? super K> comparator;

    private static class Node<K, V> {
        private final K key;
        private final V value;
        private final Node<K, V> left;
        private final Node<K, V> right;
        // ¿?
    }

    public LinkedBinarySearchTree(
           Comparator<? super K> comparator) {
        // ¿?
    }

    private LinkedBinarySearchTree(
            Comparator<? super K> comparator,
            Node<K, V> root) {
        // ¿?
    }

    @Override
    public boolean isEmpty() {
        // ¿?
    }

    @Override
    public boolean containsKey(K key) {
        // ¿?
    }

    @Override
    public V get(K key) {
        // ¿?
    }

    @Override
    public LinkedBinarySearchTree<K, V> put(K key, V value) {
        // ¿?
    }
```

[3] Fijaos en que todos los atributos de instancia son finales para "ayudar", pues no puede garantizarse si hacen referencia a objetos mutables, a la inmutabilidad de la estructura.

```
    @Override
    public LinkedBinarySearchTree<K, V> remove(K key) {
        // ¿?
    }

}
```

Fijaos en que una implementación sencilla, consistiría en copiar todo el árbol antes de realizar un *put* o *remove*. Ello sería válido, pero ocuparía demasiada memoria (además de ser ineficiente en tiempo ya que copiar todo el árbol tendría un coste lineal).

La implementación que os pedimos solamente copia el camino desde la raíz hasta el nodo que se ha modificado. Si el árbol estuviera equilibrado, este camino sería de longitud logarítmica.

Por ejemplo, en el caso de un `put` (Figura 3):

- El árbol origen y el que es devuelto, **compartan aquellos nodos que no se han visto modificados**.
- Aquellos nodos que hayan sido modificados como resultado de añadir la pareja (`key`, `value`) serán duplicados para el nuevo ABB.
- Si `key` existe en el ABB de origen se modificará su actual valor por `value` generando un nuevo nodo que represente esta modificación para el árbol devuelto.

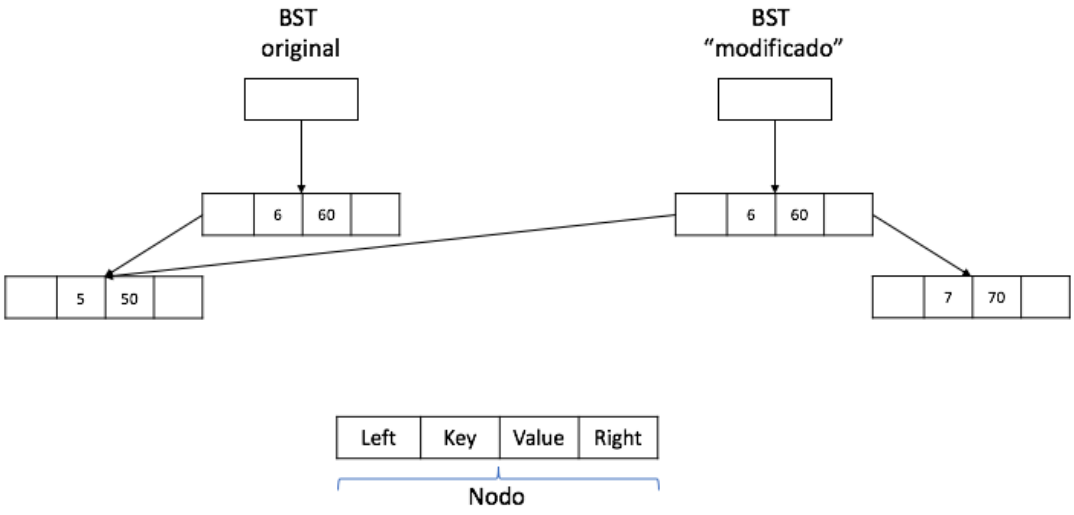

Figura 3. Ejemplo de la operación `put` sobre el árbol de búsqueda original. El resultado será el árbol de búsqueda modificado.

Y en el de un *remove* (Figura 4):

- Aquellos nodos que se han recorrido en la eliminación de (*key*, *value*) serán duplicados para el nuevo ABB.
- Esto implica que el árbol de origen y el que es devuelto, **compartan aquellos nodos que no se han recorrido.**
- Si existe *key* en el ABB, devuelve el nuevo ABB generado. En caso contrario, también se devolverá un nuevo ABB.
    - Esto simplifica la programación (de hecho, la estructura del código es muy parecida a la del caso de la inserción), pero ocupa más espacio del necesario.
    - Por ello, una de las propuestas de la parte optativa consiste, precisamente, en mejorarlo.

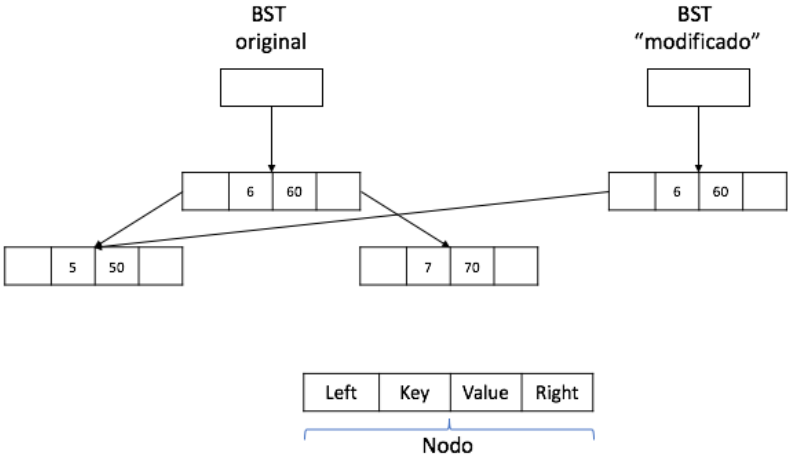

Figura 4. Ejemplo de la operación `remove` sobre el árbol de búsqueda original. El resultado será el árbol de búsqueda modificado.

Consideraciones para la realización de los test del apartado 1

Los tests de este apartado han de comprobar que la implementación de la clase cumple con la especificación de las operaciones de la interfaz y, para ello, las operaciones que disponéis son las de la interfaz más el constructor de un árbol de búsqueda vacío.

No comprobaréis si el árbol tiene la estructura, sino si se comporta como se ha de comportar.

Es decir, las únicas comprobaciones que podéis hacer son si podéis encontrar las claves (sus valores asociados) que habéis añadido; si ya no las encontráis si las habéis eliminado; y que encontráis los nuevos valores si las habéis modificado; y que el árbol inicial no se ha modificado (por ejemplo, al eliminar una clave, ésta se está eliminado en el árbol resultado, no en el de partida).

Por ello, de cara a que comprobéis que tenéis test que cubren todas las posibilidades, deberéis ser capaces de:

- Diseñar (dibujar) el árbol sobre el que queréis hacer pruebas
- Encontrar la secuencia de inserciones que lo construyen
- Definir el test

Por ejemplo, si queréis hacer pruebas sobre el árbol `arb1` de la Figura 5:

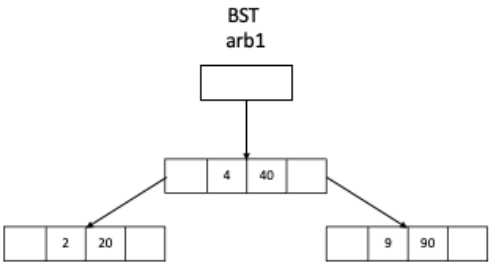

Figura 5. Un ejemplo de árbol de pruebas

Deberéis diseñar una secuencia de operaciones que lo construyan, por ejemplo:

```
Comparator<Integer> cmp = Comparator.<Integer>naturalComparator();
LinkedBinaryTree<Integer, Integer> arb1 = new LinkedBinaryTree<>(cmp);
arb1 = arb1
        .put(4, 40)
        .put(2, 20)
        .put(9, 90);
```

Por ello, para ver si aún quedan casos por comprobar, en la elaboración de las pruebas de esta tarea, se recomienda que, al ejecutar el código de prueba, se realice con la **opción de cobertura de código**[4] para que IntelliJ indique aquellas líneas que han sido cubiertas con la prueba. Ello no es garantía de haber cubierto todos los casos, pero es una ayuda.

Dicha ejecución puede realizarse para todas las pruebas (Figura 5) o sobre una en concreto (Figura 6).

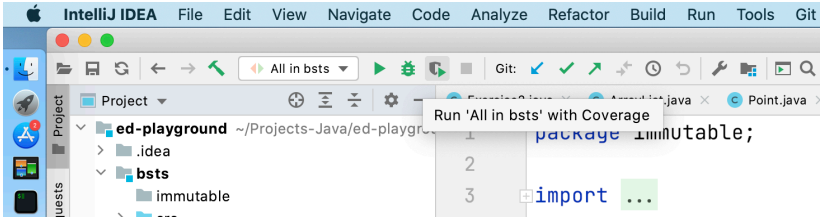

Figura 5. Ejecución de todos los test con cobertura

[4] https://www.jetbrains.com/help/idea/code-coverage.html

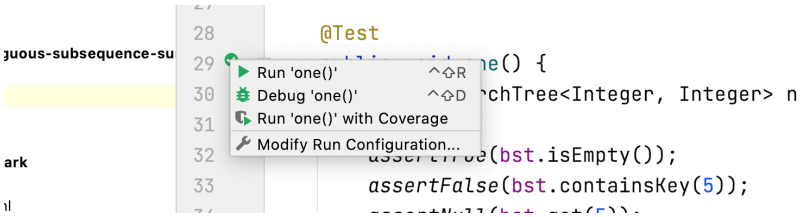

Figura 6. Ejecución de un test concreto con cobertura

Una vez ha sido ejecutado de esta manera, el entorno IntelliJ nos informará de qué líneas de código han sido ejecutadas con la prueba. La mejor manera de comprobarlo es por medio de la información visual que nos ofrece en el código que ha sido testeado. Aquello marcado con verde representa código cubierto por la prueba y en rojo el que no (Figura 7).

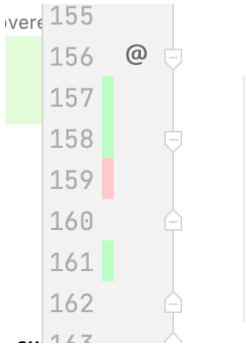

Figura 7. Marcas visuales de IntelliJ

Podéis comprobar que el entorno ofrece también información sobre el porcentaje de líneas cubiertas y no cubiertas. Sin embargo, esta información contiene cierto porcentaje relativo a interfaces (en el caso de este laboratorio) que no nos es útil. Por lo tanto, en nuestro caso es mejor acceder al código testeado y comprobar visualmente (indicaciones visuales rojas y verdes) para saber qué código ha sido cubierto por la ejecución.

En el informe correspondiente a esta parte explicaréis el diseño de cada una de las operaciones de la clase `LinkedBinarySeachTree<K, V>` así como una explicación de cómo habéis decidido los casos de prueba. Ayudaos de diagramas para que la explicación resulte entendible.

Apartado 2: Recorridos sobre árboles binarios de búsqueda inmutables

En este segundo apartado se deberá implementar el **recorrido en inorden** sobre el ABB implementado en la tarea anterior.

Deberá ser implementado **fuera de la clase** `LinkedBinarySearchTree<K, V>` y en su **versión iterativa**, usando el método explicado en el laboratorio 2 con las

posteriores simplificaciones que consideréis oportunas (como siempre, explicando todo en el informe).

Para poder implementar el recorrido, deberemos poder acceder a los subárboles derecho e izquierdo de un árbol y a su raíz, para lo que definiremos la interfaz *BinaryTree<E>*:

```
public interface BinaryTree<E> {
    boolean isEmpty();
    E root();
    BinaryTree<E> left();
    BinaryTree<E> right();
}
```

con el significado habitual de las operaciones.

Y partiremos de esta implementación del recorrido en inorden:

```
public class Traversals {

    public static <E> List<E> inorder(BinaryTree<E> tree) {
        List<E> result = new ArrayList<>();
        inorderRec(result, tree);
        return result;
    }

    private static <E> void inorderRec(List<E> result,
                                       BinaryTree<E> tree) {
        if (!tree.isEmpty()) {
            inorderRec(result, tree.left());
            result.add(tree.root());
            inorderRec(result, tree.right());
        }
    }
}
```

Como implementación de la pila, tal y como sugiere la documentación de Java, en vez de usar la clase *Stack*[5], usaremos la interfaz **java.util.Deque**[6] y su implementación **java.util.ArrayDeque**[7], en concreto, los métodos *push, pop* y *element* (para obtener el elemento en la cima de la pila) que son los que lanzan *NoSuchElementException* en los casos erróneos, así como *isEmpty* (que se hereda de *Collection*).

[5] Javadoc Stack: link
[6] Javadoc Deque: link
[7] Javadoc ArrayDeque: link

Por tanto, deberemos retocar la clase *LinkedBinarySearchTree<K, V>* para que implemente la interfaz *BinaryTree<Pair<K, V>>* y así poder realizar recorridos en inorden de un árbol binario de búsqueda.

La definición de *Pair<F, S>* es tan simple como:

```
public record Pair<F, S>(F first, S second) {}
```

Así pues, *LinkedBinarySearchTree<K, V>* tendrá el siguiente aspecto:

```
public class LinkedBinarySearchTree<K, V>
        implements BinarySearchTree <K, V>,
                   BinaryTree<Pair<K, V>> {

    //...

    @Override
    public Pair<K, V> root(){
        // ¿?
    }

    @Override
    public LinkedBinarySearchTree<K, V> left() {
        // ¿?
    }

    @Override
    public LinkedBinarySearchTree<K, V> right() {
        // ¿?
    }

    //...
}
```

Y, en la clase `Traversals`, deberemos añadir e implementar el método:

```
public static <E> List<E> inorderIterative(BinaryTree<E> tree) { ¿? }
```

Que, como ya se ha indicado, deberá **obtenerse aplicando estrictamente el mecanismo de transformación a iterativo presentado en el Laboratorio 2** a la implementación recursiva del recorrido presentado más arriba, con simplificaciones si las hubiere.

En el informe correspondiente a esta parte deberéis explicar cómo habéis aplicado la transformación a iterativo. Si habéis hecho alguna simplificación de ésta, explicad tanto la solución sin simplificar como la simplificación realizada.

Tareas complementarias

Como primera tarea complementaria, se propone mejorar la operación `remove` de la clase `LinkedBinarySearchTree<K, V>`. En concreto, la mejora se aplica al caso de la eliminación de una clave del árbol que no se encuentra en él.

En este caso, tal y como se ha implementado, se han ido duplicando nodos, a pesar de que ninguno se ha visto modificado (Figura 9) y dicha duplicación de nodos debería ser innecesaria.

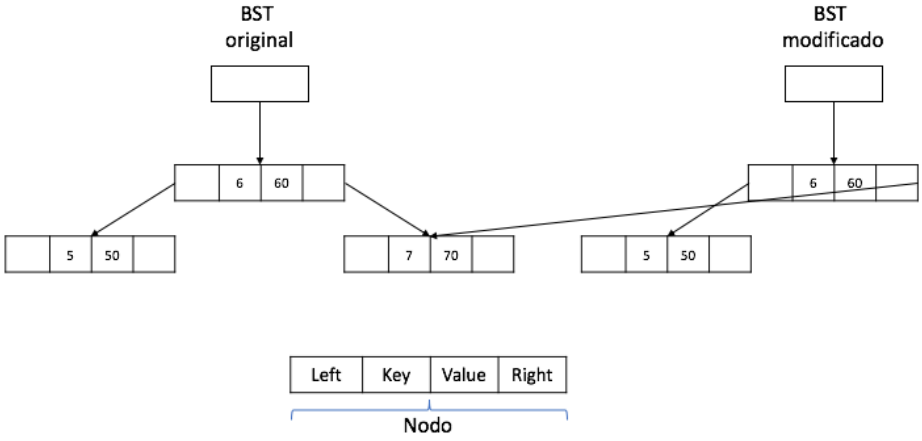

Figura 9. Ejemplo de `remove` sin mejorar.
La operación realizada es `remove(1)` sobre BST original.

De hecho, lo mismo afecta cuando hacemos un `put` y asociamos a una clave un valor que ya era igual al asociado previamente en el ABB.

Otra ampliación muy interesante consiste en implementar iteradores sobre los árboles binarios, recordad, en preorden, inorden y postorden, mediante el uso de una pila. La idea sería aplicar el esquema de transformación iterativo visto en el Laboratorio 2 (y aplicado a la segunda parte de esta práctica) y estructurarlo de manera que, a cada invocación de `next` sobre el iterador, se ejecute el código necesario para obtener el siguiente elemento en la iteración (la pila formaría parte del estado del iterador en cada momento).

Para implementar la operación `hasNext` es conveniente que el iterador en todo momento precalcule el siguiente elemento que devolverá `next`: si éste ha podido precalcularse, `hasNext` es cierto; si no, falso.

Otras consideraciones

No olvidéis compilar usando `-Xlint:unchecked y -Xlint:rawtypes` como opciones de compilación y eliminar la opción de compilación al guardar para que el compilador os avise del máximo de errores en el uso de genéricos.

Entrega

El resultado obtenido debe ser entregado como un archivo comprimido en **formato ZIP** y con el nombre "Lab4_NombreIntegrantes". En el fichero se incluirá:

- El proyecto IntelliJ con el código y las pruebas con JUnit 5.
- El informe, **en formato PDF**, en el cual se explique la solución a cada una de las tareas.

Consideraciones de Evaluación

Cada apartado es el 50% de la nota total y, a su vez, se divide en un 60% código; 40% informe.

A la hora de evaluar cada apartado se tendrán en cuenta los siguientes aspectos:

- El código ofrece una solución a cada una de las tareas planteadas.
- Calidad y limpieza del código.
- Realización de pruebas con JUnit 5.
- La explicación es entendible y describe correctamente vuestra solución del problema.
- Uso de diagramas para ayudarse en las explicaciones.