Laboratorio 5 – Persistent Tree

Este laboratorio tiene dos objetivos fundamentales:

1. Trabajar con una implementación de árboles binarios de búsqueda.

2. Trabajar con recorridos sobre árboles binarios de búsqueda.

Su realización se llevará a cabo por medio de las Tareas 1 y 2 que conjuntamente valdrán 8 puntos y dos adicionales (Tareas 3 y 4) de 1 punto cada una.

Tarea 1: Implementación de los árboles binarios de búsqueda inmutables

En este laboratorio trabajaremos con árboles binarios de búsqueda (ABB) inmutables. Por lo tanto, un ABB generado no puede ser modificado. Para ello, las operaciones que realicen una "modificación" sobre el ABB, inserción y eliminación, deben devolver un nuevo ABB, y no alterarán aquel sobre el que se apliquen.

Además, hay que tener en cuenta que no sólo el ABB debe ser inmutable, sino que los elementos que contenga también deben serlo para que la implementación tenga sentido. Aunque este último punto no puede ser comprobado por el compilador de Java.

Finalmente, el ABB no deberá equilibrarse, con lo que las operaciones de búsqueda y eliminación no serán necesariamente logarítmicas.

Recuerda que los ABBs ofrecen una estructura de datos utilizada para asociar valores a claves y, posteriormente, recuperar el valor a partir de la clave. Se definen como:

- Un árbol binario vacío.
- Un árbol binario donde:
    - Cada nodo se identifica por una clave única. Se puede encontrar en cada nodo una pareja (clave, valor).
    - La clave de la raíz es más grande que todas las claves del subárbol (hijo) izquierdo.
    - La clave de la raíz es más pequeña 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:

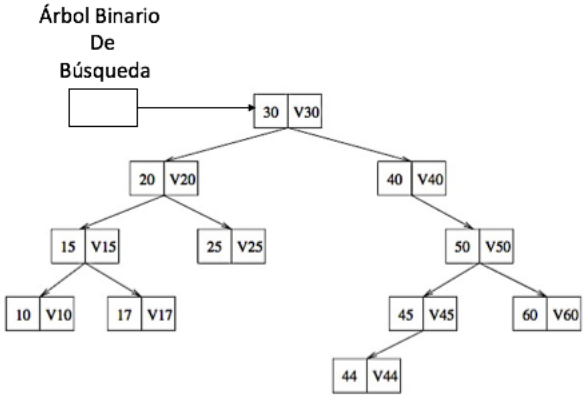


Figura 1. Ejemplo ABB

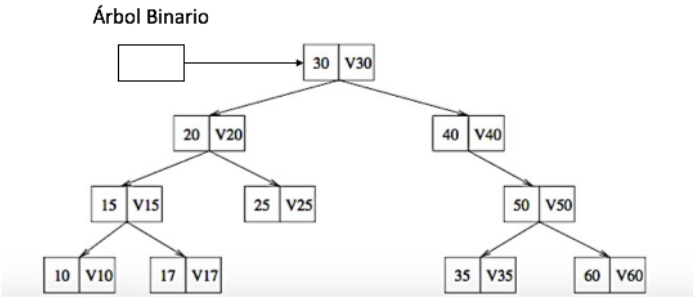


Figura 2. Ejemplo de árbol binario que no es de búsqueda

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);
}
```

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*.
- *get(K key)* devuelve el valor asociado a la clave *key*. Si no existe *key* en el ABB, devuelve *null*.

- *put(K key, V value)* devuelve un nuevo ABB basado en aquel desde el que se realiza la llamada, pero añadiendo un nuevo nodo con la pareja (*key*, *value*). Esto implica que 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.
  Si *key* o *value* son *null* devuelve *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*. 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.
  Si *key* es *null* devuelve *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 aspecto de la clase *LinkedBinarySearchTree<K, V>* podría ser el siguiente:

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

    private final Node<K, V> root;
    private final Comparator<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<K> comparator) {
        // ¿?
    }

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

}
```

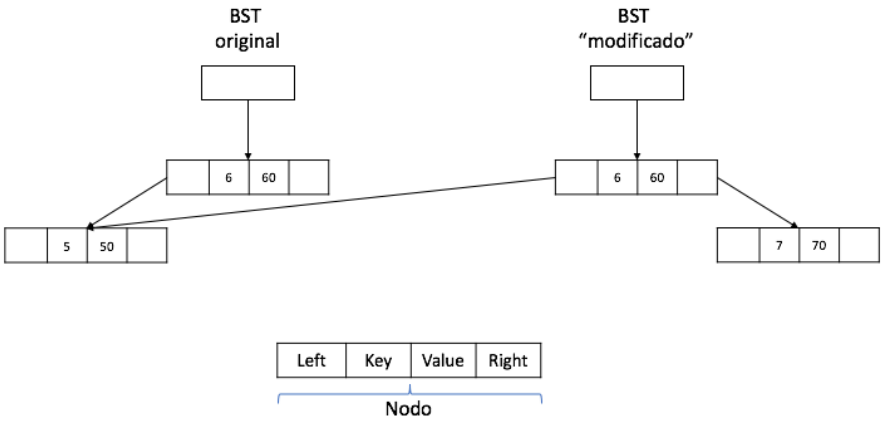


Figura 3. Ejemplo de la operación *put* sobre el árbol de búsqueda original. El resultado será el árbol de búsqueda modificado.

En base a la implementación de *LinkedBinarySearchTree<K, V>*, ahora podemos mostrar dos ejemplos sobre cómo funcionan las operaciones *put* y *remove*, a través de la Figura 3 y Figura 4, respectivamente.

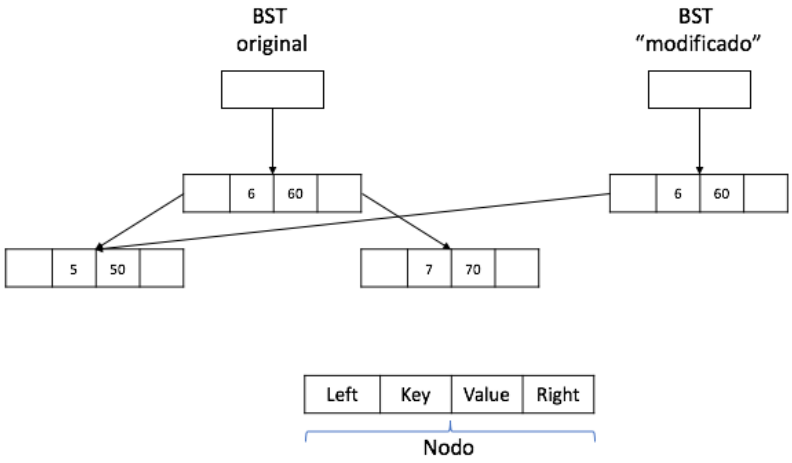


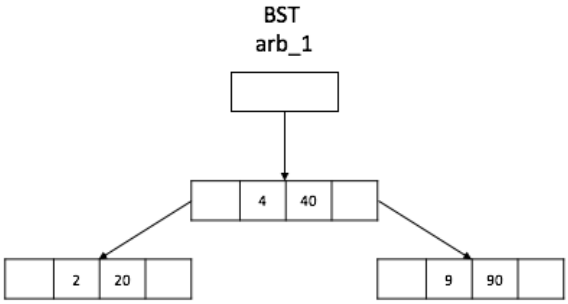
Figura 4. Ejemplo de la operación *remove* sobre el árbol de búsqueda original. El resultado será el árbol de búsqueda modificado.

Consideraciones para la realización de los test de la tarea 1

A la hora de construir un árbol para hacer un test, **se pide explícitamente que lo construyáis solamente en base al constructor de la clase LinkedBinarySeachTree<K, V> (que os crea un árbol vacío) y de la operación put de la interfaz *BinarySearchTree<K, V>, que os asocia un valor a una clave en el árbol***. A partir del árbol creado, podréis comprobar el resto de las operaciones de la interfaz. Esto implica conocer cuál será el orden necesario para generarlo. **Por lo tanto, no se podrá hacer uso de *left()* o *rigth()* que aparecerán en la Tarea 2.**

Por ejemplo, en la Figura 5, para llegar al árbol binario de búsqueda *arb_1*, debemos realizar la sucesión de operaciones que se muestra en la misma Figura 5 en el orden concreto que se indica. En este sentido, primero se construye *arb_1* y posteriormente, haciendo uso de *put,* se añaden los nodos con clave 40, 20 y 60.

En el ejemplo de la Figura 5, se puede comprobar que se enlazan operaciones en la creación de *arb_1*. Esto puede servir de ayuda para facilitar la realización de los test.




```
LinkedBinaryTree<Integer, Integer> arb_1 = new LinkedBinaryTree<>() arb1;
arb_1 = abr_1
        .put(4, 40)
        .put(2, 20)
        .put(9, 90);
```

Figura 5. Ejemplo del orden de las operaciones a la hora de llegar a un árbol binario de búsqueda concreto desde otro original

En la elaboración de los test de esta tarea, se recomienda que, al ejecutar el código de test, se realice con la opción de cobertura de código para que nos indique aquellas líneas que han sido cubiertas con el test. Para ello, tenemos la posibilidad de elegir que se aplique sobre todos los test (Figura 6) o sobre un test concreto (Figura 7).

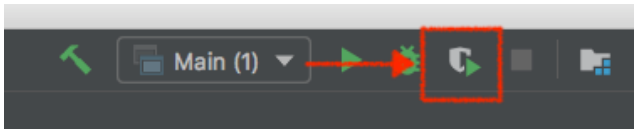


Figura 6. Ejecución de todos los test con cobertura

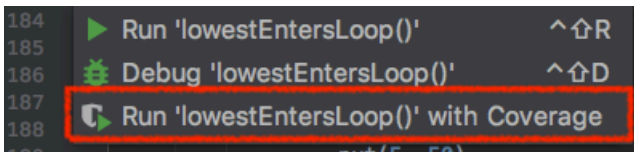


Figura 7. Ejecución de un test concreto con cobertura

Una vez ha sido ejecutado de esta manera, el entorno IntelliJ nos informará qué parte del código ha sido cubierto con el test. 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 el test (Figura 8) y en rojo lo que no (Figura 9). Podéis comprobar que el entorno ofrece 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, es mejor acceder al código testeado y comprobar visualmente (indicaciones visuales rojas y verdes) para saber qué código ha sido cubierto por el test ejecutado.

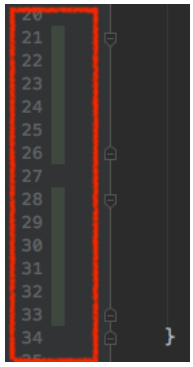


Figura 8. Marcas visuales de IntelliJ sobre aquellas líneas que han sido cubiertas por el test ejecutado

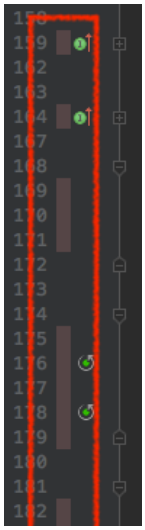


Figura 9. Marcas visuales de IntelliJ sobre aquellas líneas que no han sido cubiertas por el test ejecutado

Tarea 2: Recorrido en inorden sobre árboles binarios de búsqueda inmutables

En esta segunda tarea 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>* en su **versión iterativa**.

Para esta tarea será necesario trabajar con una pila, que podrá llamarse *Stack<E>* (cuyas operaciones ya son conocidas):

```
public interface Stack<E> {
    boolean isEmpty();
    E top();
    void pop();
    void push(E e);
}
```

La implementación de la pila podría tener como aspecto el siguiente:

```
public class LinkedStack<E> implements Stack<E> {
    private Node<E> top;

    private static class Node<E> {
        // ¿?
    }

    // ¿?
}
```

¿Qué información contendrá la pila? Por lo menos algo que tenga que ver con los árboles binarios, ya que éstos son los elementos que una implementación recursiva se pasaría como parámetro. Sin embargo, y esto es una pista, junto a estos árboles será necesario almacenar información adicional. Para poder hacer esto de forma sencilla, podéis usar clase *Pair<S, T>* (utilizando los tipos adecuados):

```
public class Pair<S, T> {

    private final S first;
    private final T second;

    public Pair(S first, T second) {
        // ¿?
    }

    public S first() {
        // ¿?
    }

    public T second() {
        // ¿?
    }

}
```

Ahora, debemos retocar la clase *LinkedBinarySearchTree<K, V>* para que implemente la interfaz *BinaryTree<Pair<K, V>>* y así, trabajar en la obtención del recorrido en inorden con las operaciones que esta interfaz nos ofrece, la cual es la siguiente:

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

Las operaciones de esta interfaz se comportan de la siguiente manera:

- *isEmpty()* devuelve *true* si el árbol binario está vacío y *false* en caso contrario.
- *root*: devuelve la raíz del árbol binario y lanza NullPointerException si el árbol está vacío.
- *left()*: devuelve subárbol izquierdo o árbol vacío en caso de que este no exista y lanza NullPointerException si el árbol está vacío.
- *right()*: devuelve subárbol derecho o árbol vacío en caso de que este no exista y lanza NullPointerException si el árbol está vacío.

De este modo, *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() {
        // ¿?
    }


    //...
}
```

Tarea 3 (Adicional): Mejora de la operación remove

Como tarea complementaria, para obtener una valoración extra en este laboratorio 5, se propone mejorar la operación *remove* de la clase *LinkedBinarySearchTree<K, V>*. En concreto, la mejora que se propone es en la eliminación de un nodo del árbol que no se encuentra en él. En este caso, se ha implementado duplicando nodos a pesar de que ninguno se ha visto modificado (Figura 10). Por lo tanto, lo que se propone es no duplicar los nodos que no hayan sido modificados (Figura 11).

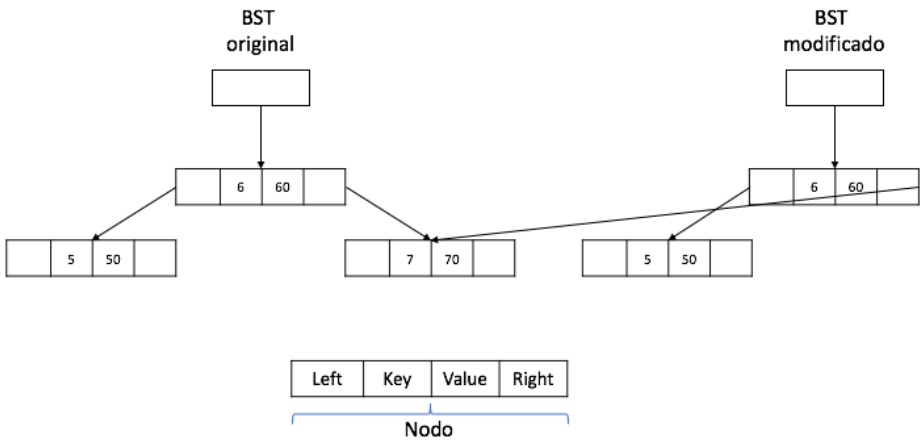


Figura 10. Ejemplo de *remove* sin mejorar.
La operación realizada es *remove(1, 10)* sobre BST original.

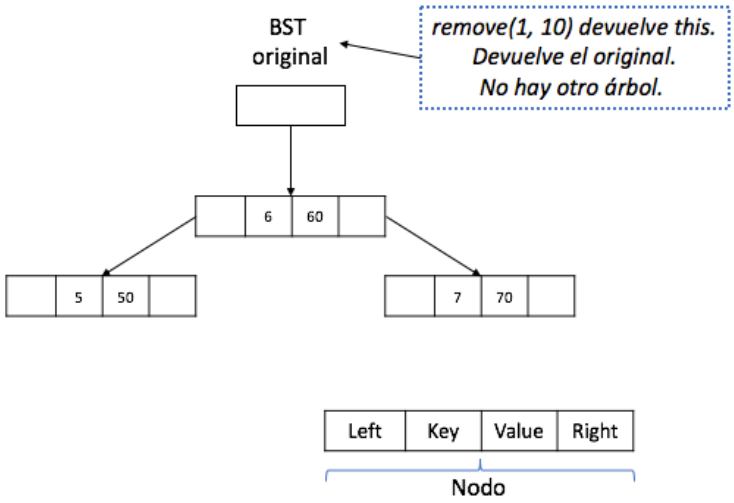


Figura 11. Ejemplo de la versión mejorada de *remove*.
La operación realizada es *remove(1, 10)* sobre BST original.

Tarea 4 (Adicional): Iterador sobre árboles binarios de búsqueda

Como tarea complementaria, para obtener una valoración extra en este laboratorio 5, se propone implementar un **iterador** (**recorrido en inorden**) sobre el ABB inmutable *LinkedBinarySearchTree<K, V>* obtenido en la tarea 2 (o la 3).

Para ello *LinkedBinarySearchTree<K, V>* también implementará la interfaz *Iterable<Pair<K, V>>* y existirá, como en el caso de las listas enlazadas, una clase privada dentro de ella que implementará *Iterator<Pair<K, V>>*.

Otras consideraciones

No olvidéis compilar usando la opción –Xlint:unchecked para que el compilador os avise de errores en el uso de genéricos.

Universitat de Lleida
Departament d'Informàtica
i Enginyeria Industrial

Entrega

El resultado obtenido debe ser entregado por medio de un proyecto IntelliJ por cada grupo, comprimido y con el nombre "Lab5_NombreIntegrantes". En el proyecto deberán estar presentes los test realizados con JUnit 4.

Además, se debe entregar un documento de texto, máximo de cuatro páginas, en el cual se explique la solución a cada una de las tareas.

Consideraciones de Evaluación

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

- El código de cada proyecto ofrece una solución a cada una de las tareas planteadas.
- Realización de test con JUnit 4.0.
- Explicación adecuada del trabajo realizado, es decir, esfuerzo aplicado al documento de texto solicitado.
- Calidad y limpieza del código.