Laboratorio 5 – Árboles Binarios y Recorridos

Este laboratorio tiene dos objetivos fundamentales:

1. Realizar una implementación de los árboles binarios.
2. Implementar los tres recorridos fundamentales sobre ellos (pre-, in- y post-orden) en sus versiones iterativas.
3. Razonar sobre las propiedades de los recorridos post-orden e in-orden de un árbol binario.

Tarea 1: Implementación de los árboles binarios (3 puntos)

Interfaz BinaryTree<E>

Esta será la interfaz que declarará las operaciones que podremos realizar sobre los árboles binarios:

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

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

- *elem(), left(), right()* devuelven el valor almacenado en el nodo raíz del árbol, el subárbol izquierdo y el subárbol derecho, respectivamente. Todas ellas lanzan *NoSuchElementException* en caso de ser aplicadas sobre un árbol vacío.
- *isEmpty()* devuelve *true* si el árbol está vacío y *false* en caso contrario.

Clase *LinkedBinaryTree<E>*

Esta será la clase que implementará la interfaz anterior (cumpliendo las especificaciones de las operaciones), usando nodos enlazados. Obviamente, además de las operaciones de la interfaz, tendrá diferentes constructores, para crear diferentes tipos de árboles:

- *public LinkedBinaryTree()*
- *public LinkedBinaryTree(E elem)*

- *public LinkedBinaryTree(E elem, LinkedBinaryTree<E> left, LinkedBinaryTree<E> right)*

Respectivamente, el primer constructor crea un un árbol vacío. El segundo crea un árbol con una hoja que contiene *elem* y, finalmente, el tercer constructor crea un árbol cuya raíz contiene el elemento *elem* y cuyos árboles izquierdo y derecho son *left* y *right*. En este último caso, si *left* y *right* son *null*, se tratarán como árboles vacíos.

Antes de continuar, el aspecto de la clase *LinkedBinaryTree<E>* podría ser el siguiente:

```
public class LinkedBinaryTree<E> implements BinaryTree<E> {

    private Node<E> root;

    private static class Node<E> {
        private final E elem;
        private final Node<E> left;
        private final Node<E> right;

        // ¿?

        @Override
        public boolean equals(Object obj) {
            // ¿?
        }

    }

    public LinkedBinaryTree() {
        // ¿?
    }

    public LinkedBinaryTree(E elem) {
        // ¿?
    }

    public LinkedBinaryTree(E e, LinkedBinaryTree<E> left,
                                 LinkedBinaryTree<E> right) {
        // ¿?
    }

    private LinkedBinaryTree(Node<E> root) {
        // ¿?
    }

    @Override
    public E elem() {
        // ¿?
    }

    @Override
    public BinaryTree<E> left() {
        ¿?
    }

    @Override
    public BinaryTree<E> right() {
```

```
        ¿?
    }

    @Override
    public boolean isEmpty() {
        return root == null;
    }

    @Override
    public boolean equals(Object obj) {
        ¿?
    }

}
```

Pista

- Puede ser conveniente añadir un constructor privado para construir un árbol a partir de un nodo (la clase interna que usáis para implementarlos), es decir:

  ```
  private LinkedBinaryTree(Node<E> root)
  ```

Junto con los constructores ya comentados, así como los que corresponde con la interfaz *BinaryTree<E>*, es interesante resaltar los siguientes puntos:

- Se deberá implementar *equals* para árboles binarios. Una implementación recursiva del *equals* en la clase interna Node puede simplificar la tarea.
- Cuando vayáis a realizar los test, podría ser de vuestro interés hacer uso del método estático *List.of* incluido en Java desde la versión 9 y que crea una lista inmutable con el comando *List.of(E…elements)*. Si lo que se necesita es una lista mutable, se puede pasar como parámetro del constructor de ArrayList: *new ArrayList<>(List.of(E…elements))*. Para más información: https://docs.oracle.com/javase/9/docs/api/java/util/List.html#of--

Tarea 2: Recorridos iterativos en árboles binarios (4 puntos)

Una vez implementado el árbol binario, os pedimos que implementéis los tres recorridos fundamentales sobre árboles binarios (pre-, in- y post-orden) en sus **versiones iterativas**, es decir, **sin usar recursividad**. La clave de ello consistirá en **usar una pila**, de manera semejante a lo que hace la máquina virtual de Java cuando ejecuta la versión recursiva. Esta pila deberá ser implementada dejando al alumno la elección de hacerlo usando memoria contigua o apuntadores.

Interfaz *Traversals*

Esta interfaz declarará los métodos que implementarán cada uno de los recorridos. Fijaos que se trata de una interfaz no genérica que contiene tres métodos que sí lo son.

```
public interface Traversals {
    <E> List<E> preOrder(BinaryTree<E> tree);
    <E> List<E> inOrder(BinaryTree<E> tree);
    <E> List<E> postOrder(BinaryTree<E> tree);
}
```

¿Qué diferencias provocaría que la interfaz fuera genérica y los métodos no?

Clase *IterativeTraversals*

Esta es la clase que contendrá la implementación iterativa de los tres recorridos y la parte central de este laboratorio.

Pistas

- Recordad: en los tres recorridos, visitar la raíz de un árbol es lo mismo que hacer el recorrido de un árbol que solamente tiene un nodo y dos árboles vacíos como hijos.
- Conceptualmente pueden verse los recorridos como funciones que hacen dos operaciones de "tratamiento":
    - tratar(árbol)
        - si no es vacío, descompone en las partes y hace las llamadas (algunas recursivas) a las operaciones de tratamiento.
    - tratar(elemento)
        - añade la información del elemento al resultado.
- Para diseñar las funciones de forma iterativa, es conveniente dibujar cómo cada recorrido va guardando en la pila las diferentes subtareas a realizar.
- Recordad que las pilas son LIFO (o FILO), por lo que, al meter las subtareas en una pila, estas se realizarán en orden inverso.
- Podéis usar tanto la clase Stack<E> definida en la JCF como una implementación propia.

Tarea 3: Reconstrucción del árbol binario (3 puntos)

Una vez implementado el árbol binario con nodos enlazados (*LinkedBinaryTree<E>*), el siguiente paso consistirá en poder reconstruir un árbol, **si sus nodos no contienen elementos repetidos**, dados sus recorridos *postorden* e *inorden*.

Es decir, se pedirá el método:

```
public class Regenerator {

    public static <E> LinkedBinaryTree<E> postAndIn(
                            List<? extends E> postorder,
                            List<? extends E> inorder)
    { ¿? }

}
```

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.

Entrega

El resultado obtenido debe ser entregado por medio de un proyecto IntelliJ por cada grupo, comprimido y con el nombre "Lab5_NombreIntegrantes".

Además, se debe entregar un documento de texto, máximo de dos páginas, en el cual se explique el funcionamiento de la implementación realizada.

Ayudaos de diagramas para explicar cómo habéis diseñado las clases y funciones.

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.
- Realización de test con JUnit 5
- Explicación adecuada del trabajo realizado, es decir, esfuerzo aplicado al documento de texto solicitado.
- Calidad y limpieza del código.

Apéndice: La clase *RecursiveTraversals*

De cara a hacer pruebas, os puede resultar útil esta clase que implementa, de forma recursiva, los tres recorridos que os pedimos:

```
public class RecursiveTraversals implements Traversals {

    @Override
    public <E> List<E> preOrder(BinaryTree<E> tree) {
        List<E> result = new ArrayList<>();
        preOrder(tree, result);
        return result;
    }

    private <E> void preOrder(BinaryTree<E> tree, List<E> result) {
        if (!tree.isEmpty()) {
            result.add(tree.elem());
            preOrder(tree.left(), result);
            preOrder(tree.right(), result);
        }
    }

    @Override
    public <E> List<E> inOrder(BinaryTree<E> tree) {
        List<E> result = new ArrayList<>();
        inOrder(tree, result);
        return result;
    }

    private <E> void inOrder(BinaryTree<E> tree, List<E> result) {
        if (!tree.isEmpty()) {
            inOrder(tree.left(), result);
            result.add(tree.elem());
            inOrder(tree.right(), result);
        }
    }

    @Override
    public <E> List<E> postOrder(BinaryTree<E> tree) {
        List<E> result = new ArrayList<>();
        postOrder(tree, result);
        return result;
    }

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

Apéndice: Versión iterativa del Fibonacci

De cara a entender mejor el tipo de transformación que debéis realizar, os mostramos el siguiente ejemplo: la transformación a iterativo del cálculo de la secuencia de Fibonacci.

```
public class Fibonacci {

    public static int recursiveFibonacci(int n) {
        if (n <= 1)
            return n;
        else
            return recursiveFibonacci(n - 1)
                    + recursiveFibonacci(n - 2);
    }

    public static int iterativeFibonacci(int n) {
        int result = 0;
        Stack<Integer> pendingWork = new LinkedStack<>();
        pendingWork.push(n);
        while (!pendingWork.isEmpty()) {
            int current = pendingWork.pop();
            if (current <= 1) {
                result += current;
            } else {
                pendingWork.push(current - 2);
                pendingWork.push(current - 1);
            }
        }
        return result;
    }
}
```