Laboratorio 3 – Recorridos iterativos en árboles binarios

En este proyecto os pedimos que implementéis los tres recorridos fundamentales sobre árboles binarios (pre-, in- y post-orden) pero 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.

Primera parte: implementación de los árboles binarios

Como solamente estamos interesados en resolver el problema de los recorridos, implementaremos, usando una estructura de nodos y referencias, una versión mínima de los árboles. Podéis basaros en la implementación que hay en los apuntes, simplificándola convenientemente.

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, crean un árbol vacío, uno consistente en una hoja que contiene *elem* y, finalmente, un árbol cuya raíz contiene el elemento *elem* y cuyos árboles izquierdo y derecho son *left* y *right*.

Pistas:

- 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)*

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.
- ¿De qué tipo son los elementos que hay en la pila? Si nos fijamos en lo que hemos comentado antes, hay tareas que necesitan un árbol y otras un elemento. Una posible solución es, en vez de guardar un elemento, guardar un árbol que solamente tiene un elemento. De esta manera, la pila que usamos es de árboles.
  - Inicialmente la pila contiene el árbol que se pasa como parámetro.
  - Mientras la pila no está vacía, se obtiene (y elimina) el árbol de la cima de la pila; se descompone y se trata; y, si es necesario, se añaden las subtareas a la pila, para ser tratadas en iteraciones posteriores.

Entrega

La entrega de este ejercicio consiste en un proyecto Netbeans con los ficheros que contienen vuestra implementación y un **informe en formato PDF** que contenga, como mínimo:

- Respuesta a la pregunta sobre la interfaz Traversals.
- Diagramas que muestren el funcionamiento de las operaciones sobre los árboles.

- Diagramas que muestren, para cada recorrido, la relación entre las llamadas recursivas del mismo y las manipulaciones que hacéis en la pila.

El resultado obtenido debe ser entregado por medio de un único **fichero comprimido en formato ZIP**, **uno por cada grupo,** con el nombre "Lab3_NombreIntegrantes.zip".

Consideraciones de Evaluación

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

- Desgraciadamente hemos de indicar que una solución que se detecte como copiada tendrá **una nota igual a 0 y no será recuperable**.
- Corrección de la solución (una solución incorrecta tendrá una **nota inferior a 4**).
  - Es conveniente que hagáis diferentes pruebas sobre diferentes árboles, de vuestra implementación
- Corrección del informe entregado y justificación de los diseños de las funciones, tanto de las correspondientes a la implementación de los árboles, como las de los recorridos iterativos.
  - No hace falta que hagáis los diagramas electrónicamente: podéis hacerlos a mano y después escanearlos.
- Calidad y limpieza del código (nombres de variables y métodos, descomposición funcional, etc., etc.).
- Diseño orientado a objetos y buen uso de los genéricos:
  - Recordaos de configurar `-Xlint:unchecked` (y de desconectar la compilación automática) para que se os muestren los errores en el uso de genéricos.
- Realización de tareas opcionales:
  - Re-estructurando el código, podéis usar esta misma idea para implementar iteradores sobre los árboles de manera que, a cada invocación de next, obtengan el siguiente elemento del recorrido. PISTA: para que funcione correctamente la función hasNext, es conveniente tenerlo siempre precalculado.

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;
    }
}
```