EXAMEN FEBRERO.2021

Examen de Febrero 2021

Códigos para el ejercicio 1:
Tren.hs

Ejercicio 1 - Haskell (16:05-16:30)

Un tren está compuesto por una máquina y vagones. Los vagones pueden transportar objetos identificados por su peso (`Int`). Cada vagón puede transportar varios objetos hasta un peso máximo (dado por la función `tope = 10`). Para representar un tren usaremos el tipo:

```
data Tren = Maquina
          | Vagon Int [Int] Tren
```

donde

- `Maquina` es un tren sin vagones
- `Vagon cr xs tren` es un vagón con:
  - `cr` capacidad restante del vagón (peso que aún cabe en el vagón)
  - `xs` lista de objetos (pesos) que ya lleva el vagón (siempre se cumple `cr + sum xs == tope`)
  - `tren` el resto del tren

Cuando se crea un vagón, su capacidad restante es `tope` y la lista de objetos que ya lleva está vacía.

Define la función:

```
del :: Int -> Tren -> Tren
```

que borra el **primer objeto** de un peso dado del tren (el primer objeto con dicho peso que se encuentre en el primer vagón que lo contenga, al recorrer el tren de izquierda a derecha) y devuelve el tren sin ese objeto. Si en el tren no hay ningún objeto de ese peso, se debe elevar un error. Si como resultado de eliminar ese objeto el vagón que lo contenía queda vacío, el vagón también se debe desconectar del tren.

Por ejemplo:

```
ejemplo1 = Vagon 2 [5,3] (Vagon 3 [3,2,2] (Vagon 5 [5] Maquina))
ejemplo2 = Vagon 1 [2,1,1,3,2] (Vagon 4 [4,2] Maquina)

> del 5 ejemplo1
(7,[3])-(3,[3,2,2])-(5,[5])-XxIx>

> del 5 (del 5 ejemplo1)
(7,[3])-(3,[3,2,2])-XxIx>

> foldr del ejemplo1 [5,2,5,3,3]
(8,[2])-XxIx>

> del 1 ejemplo2
(2,[2,1,3,2])-(4,[4,2])-XxIx>

> del 5 ejemplo2
(1,[2,1,1,3,2])-(4,[4,2])-*** Exception: No se ha encontrado ese peso en el tren
```

Códigos para el ejercicio 2:
reverse,src,zip

Ejercicio 2 - Java (16:30-16:55)

Consideremos una estructura lista genérica implementada linealmente mediante nodos enlazados, de forma que cada nodo contiene un elemento y una referencia al nodo siguiente (o un valor `null` en caso de ser el último nodo de la estructura). La clase que implementa la lista, `LinkedList` (en el paquete `reverse`), contiene un único atributo, `first`, que es una referencia al primer nodo de la lista (o un valor `null` en caso de que la lista esté vacía.)

```
public class LinkedList<T> {
  private static class Node<E> {
    E elem;
    Node<E> next;
```

```
    Node(E elem) {
      this.elem = elem;
      this.next = null;
    }
  }
```

```
  private Node<T> first;
}
```

Define en esta clase un método `reverse` para la clase que actúe sobre una lista e invierta el orden de sus elementos (el primero debe acabar siendo el último, el segundo será el penúltimo, etc.). A la hora de implementar este método, no deberán crearse nuevos nodos sino que deberán re-enlazarse en orden inverso los nodos que ya forman parte de la lista a invertir. Debes implementar este método manipulando directamente la lista (sin usar otra estructura de datos auxiliar) y de forma que su complejidad sea O(n).

Hay un ejemplo de prueba en los códigos proporcionados para el ejercicio (clase `LinkedListTest` del paquete `reverse`). La salida que debes obtener al ejecutar dicho ejemplo es:

```
List before reverse
LinkedList(0,1,2,3,4,5,6,7,8,9)
List after reverse
LinkedList(9,8,7,6,5,4,3,2,1,0)
```

Árboles binarios

Los árboles binarios pueden representarse en Java mediante la clase genérica `BinTree<T>`:

```
public class BinTree<T> {

    private static class Tree<E> {
        private E elem;
        private Tree<E> left;
        private Tree<E> right;

        public Tree(E e, Tree<E> l, Tree<E> r) {
            elem = e;
            left = l;
            right = r;
        }
    }

    private Tree<T> root;
    ...
}
```

La clase `BinTree` dispone de hasta 3 constructores (para árboles vacíos, hoja y con dos hijos) y del método `toString`.

En el caso de Haskell, los árboles binarios pueden representarse por el tipo polimórfico `BinTree a`:

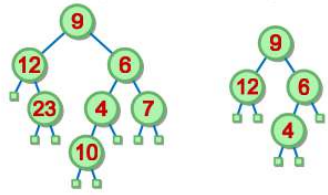
```
data BinTree a = Empty
               | Node a (BinTree a) (BinTree a)
               deriving Show
```

Códigos para los ejercicios 3 y 4:
trim.src.zip

Ejercicios 3 y 4 - Poda de árboles (16:55-17:30)

La **poda** de un árbol binario consiste en eliminar todos sus nodos hoja. Por ejemplo, podando el árbol de la izquierda obtenemos el árbol de la derecha:

árbol original **árbol podado**



1. Completa en la clase `BinTree` del paquete `dataStructures.tree.trimmedTrees` la definición del método java

```
public void trim()
```

que poda el árbol.

2. Completa en el módulo `TrimmedTrees` la definición de la función Haskell

```
trim :: BinTree a -> BinTree a
```

que dado un árbol binario devuelve el árbol podado.

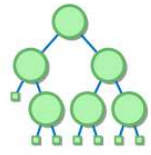
Códigos para los ejercicios 5 y 6:
isomorphic.src.zip

Ejercicio 5 y 6 - Isomorfismo de árboles (17:30-18:05)

Dos árboles binarios son **isomorfos** si son iguales prescindiendo del valor de sus nodos; es decir, tienen la misma forma o estructura. Por ejemplo, los siguientes 3 árboles binarios son isomorfos entre sí:

árbol 1 **árbol 2** **árbol 3**

Observa que los 3 árboles anteriores tienen, en efecto, la misma forma:



1. Completa en la clase `BinTree` del paquete `dataStructures.tree.isomorphicTrees` la definición del método java

```
public <E> boolean isomorphic(BinTree<E> that)
```

que devuelve `true` si los árboles `this` y `that` son isomorfos y `false` en caso contrario. El método público `isomorphic` debe basarse en el método privado:

```
private static <A, B> boolean isomorphicRec(Tree<A> t1, Tree<B> t2)
```

2. Completa en el módulo `IsomorphicTrees` la definición de la función Haskell

```
isomorphic :: BinTree a -> BinTree b -> Bool
```

que dados dos árboles binarios devuelve `True` si los árboles son isomorfos y `False` en caso contrario.