Relación de Ejercicios 4

Para realizar estos ejercicios necesitarás crear diferentes ficheros tanto en Haskell como en Java. En cada caso crea un nuevo fichero (con extensión hs para Haskell y java para Java). Añade al principio de tu fichero la siguiente cabecera, reemplazando los datos necesarios:

```
-------------------------------------------------------------------------------
-- Estructuras de Datos. 2º Curso. ETSI Informática. UMA
--
-- (completa y sustituye los siguientes datos)
-- Titulación: Grado en Ingeniería ..... [Informática | del Software | de Computadores].
-- Alumno: APELLIDOS, NOMBRE
-- Fecha de entrega:  DIA | MES | AÑO
--
-- Relación de Ejercicios 4. Ejercicios resueltos: ..........
--
-------------------------------------------------------------------------------
import Test.QuickCheck   (Cuando se trate de Haskell)
```

1. (Java) Implementa el TAD colas de prioridad (`LinkedPriorityQueue`) usando una lista enlazada de nodos (en orden ascendente). Este TAD debe Implementar la interfaz `PriorityQueue`. Una cola de prioridad es similar a una cola pero el método `enqueue` coloca los elementos en orden ascendente. Realiza tu implementación en el paquete `dataStructures.priorityQueue`.

```
public interface PriorityQueue<T> {
   void enqueue(T elem);
   void dequeue();
   T first();
   boolean isEmpty();
}

public class LinkedPriorityQueue <T implements Comparable<? super T>>
                implements PriorityQueue<T> {
   public void enqueue(T elem) {
          ...
   }
}
```

2. (Haskell) Implementa el TAD colas de prioridad usando una estructura recursiva lineal que no mantenga los elementos ordenados. La implementación insertará los elementos por la cabeza (como en un stack). `enqueue` y `first` deben localizar el elemento mínimo en una estructura no ordenada y, respectivamente, eliminarlo o devolverlo (crea un módulo `DataStructures.PriorityQueue.LinearPriorityQueue`). Analiza la complejidad de estas operaciones.

3. (Java) Resuelve el ejercicio anterior usando una lista enlazada no ordenada (`NonSortedLinkedPriorityQueue`). Este ejercicio se resolvería igual que el ejercicio 1 y en el mismo paquete.

4. (Java) Escribe un programa para determinar la altura media de un BST (árbol binario de búsqueda) con n valores aleatorios. Para generar un BST, usa `java.util.Random` para generar n elementos y después insértalos en un BST vacío.

5. (Java) Usando los métodos del ejercicio anterior, escriba un programa para generar 50 BSTs, todos con n elementos, y que calcule la media de sus alturas. Aplica este método para calcular la altura media de BSTs con 10, 100 y 1000 elementos aleatorios.

6. (Haskell). El módulo `Random` (disponible en `DataStructures.Util.Random`) proporciona la función `randomsR :: (Random a) => (a,a) -> Seed -> [a]` que puede ser usada para generar una lista infinita de valores aleatorios dentro de un rango. El segundo parámetro es un valor `Int` (`Seed` es un sinónimo de tipo de `Int`) de forma que, para diferentes valores de la semilla, la función devuelve diferentes secuencias. Por ejemplo, podemos crear una función que simule tirar 5 dados por

```
cincoDados :: Seed -> [Int]
cincoDados seed = take 5 (randomsR (1,6) seed)
```

Y usarlo con diferentes semillas para obtener diferentes experimentos

```
Main> cincoDados 0
[3,5,3,1,1]
Main> cincoDados 1
[4,3,4,1,5]
Main> cincoDados 2
[6,1,1,4,1]
Main> [ cincoDados s | s <- [0..2]]
[[3,5,3,1,1],[4,3,4,1,5],[6,1,1,4,1]]
```

Resuelve los problemas 4 y 5 en Haskell usando este módulo para generar 50 BST aleatorios (usa como valores para las semillas los elementos de la lista [0..49] y una lista por comprensión).

7. (Haskell) Escribe una función:

```
levels :: Tree a -> [a]
```

para atravesar un árbol general por niveles. El primer elemento sería el correspondiente al nivel 0 (raíz), seguidos por los elementos del nivel 1, seguidos por los elementos del nivel 2, etc. El algoritmo debe ser eficiente (y no debe explorar el árbol tantas veces como niveles). Para implementar este algoritmo necesitarás usar una cola que mantiene los subárboles no atravesados. Comenzamos por añadir el árbol a explorar en una cola vacía, para obtener una cola `queueTrees`. Ahora repetimos el siguiente proceso hasta que la cola `queueTrees` sea vacía cola: extraemos un árbol de la cola, visitamos el nodo raíz, y añadimos todos sus hijos a la cola.

8. (Haskell) Este ejercicio está tomado de http://programmingpraxis.com.

Los montículos Maxifóbicos (*Maxophobics*) son una variante de montículos zurdos, y ambos permiten una implementación eficiente de una cola de prioridad.

En los zurdos, el peso del hijo izquierdo es siempre mayor o igual que el del derecho.

Como los zurdos, los maxifóbicos se representan por medio de árboles binarios aumentados, que mantienen la propiedad de Heap Order, y por tanto, en cada nodo la clave es menor que las claves de sus dos hijos. Por tanto, el `minElem` se encuentra en la raíz del árbol. `delMin` descarta la raíz y mezcla sus dos hijos e `insert` mezcla el árbol existente con un nuevo árbol con el elemento a insertar como único elemento (`singleton`).

Los árboles zurdos son simples de utilizar y de codificar; es fácil demostrar que la mezcla de dos montículos zurdos (resuelto en las transparencias) es zurdo, y de esta forma la longitud de su espina

derecha no supera al logaritmo del total de nodos de la mezcla, siendo ésta, salvo un factor de proporcionalidad, una cota del número de operaciones de la mezcla. De aquí se deduce que las operaciones de inserción y eliminación tienen complejidad logarítmica, ya que éstas se reducen a una mezcla.

Los montículos maxifóbicos son una alternativa a los zurdos, pero no necesitan estar balanceados a la izquierda (lo que libera del uso del invariante de los zurdos), y sin embargo las operaciones de inserción y borrado tienen complejidad logarítmica. La diferencia de los montículos zurdos con los maxifóbicos se encuentra en la operación de mezclar. En los maxifóbicos, la operación de mezcla se implementa comparando los valores de las raíces de los dos árboles. La menor se mantiene como valor de la raíz del montículo *hm* combinado resultado de la mezcla; el resto de la información se distribuye en tres árboles: el montículo ganador de la comparación (que contendrá la clave mayor), y los hijos del perdedor. De estos tres árboles se toma el mayor (con más nodos) y se coloca en la rama derecha del montículo mezcla *hm*, colocándose en la rama izquierda la mezcla de los dos restantes. De esta forma el montículo combinado contiene todas las claves de los originales y además verifica la propiedad HO (heap order). Sorprendentemente un sencillo razonamiento permite demostrar que esta mezcla tiene complejidad logarítmica (véase el artículo de Chris Okasaki, "Alternatives to Two Classic Data Structures", *SIGCSE'05,* February 23–27, 2005).

El nombre maxifóbico quiere decir "esquivando (o descartando) el mayor": se mezclan los dos menores subárboles (los dos con menos elementos).

a) Completa la implementación de las operaciones de estos montículos maxifóbicos, modificando lo necesario en el código de los árboles zurdos (usa un módulo `DataStructures.Heap.MaxiphobicHeap`).

b) (Difícil) La operación de mezcla de montículos maxifóbicos normalmente se implementa con complejidad logarítmica por lo que es una implementación eficiente. Probar que efectivamente la operación tiene complejidad logarítmica.

Ayuda: Sea T(N) el número de invocaciones a la función de mezcla para mezclar dos árboles maxifóbicos, donde el número total de nodos en ambos árboles es N. Intenta escribir una relación de recurrencia para T(N) teniendo en cuenta que el mayor de los subárboles se ha evitado en cada paso. Para calcular el tamaño total de los dos subárboles que se mezclarán (los dos más pequeños), se debería primeramente determinar cuantos elementos tiene el subárbol evitado (el mayor).

9. (Java) Implementa los montículos maxifóbicos en Java. (usa una clase `dataStructures.heap.MaxiphobicHeap.java`)

10. (Haskell) Implementa un módulo para diccionarios de claves y valores usando árboles AVL (usa un módulo `DataStructures.Dictionary.AVLDictionary`). Cada nodo deberá contener la clave y el valor asociado, y el árbol debería estar ordenado por las claves.

La implementación debería definir las siguiente funciones:

- `empty :: Dict a b`. Devuelve un diccionario vacío.
- `isEmpty :: Dict a b -> Bool`. Test para diccionarios vacíos
- `insert :: a -> b -> Dict a b -> Dict a b`. Inserta una clave y su valor en un diccionario. Si la clave ya existía, el nuevo valor reemplazará al antiguo.
- `delete :: a -> Dict a b -> Dict a b`. Elimina la clave y su valor de un diccionario
- `get :: a -> Dict a b -> Maybe b`. Devuelve (como `Maybe`) el valor asociado a una clave.
- `keys :: Dict a b -> [a]`. Devuelve una lista ordenada con las claves del diccionario.
- `values :: Dict a b -> [b]`. Devuelve una lista con todos los valores del diccionario.

- `keysValues :: Dict a b -> [(a,b)]`. Devuelve una lista de pares clave y valor del diccionario ordenadas por clave.
- `mapValues :: (b -> c) -> Dict a b -> Dict a c`. Aplica la función dada a todos los valores del diccionario, sin alterar las claves.

11. (Java) Implementa un clase genérica para definir tuplas de dos componentes `Tuple2<A,B>`, donde `A` corresponde al tipo de la primera componente y `B` al tipo de la segunda (usa una clase `dataStructures.tuple.Tuple2.java`). Además del constructor, que recibe un dato de cada tipo, define los siguiente métodos:

```
public A _1();
public B _2();
```

que devuelven la primera y la segunda componente de la tupla y el método `toString`, que devuelve la cadena "Tuple2(x,y)", donde x e y son las strings correspondientes a las componentes.

12. (Java) Implementa los diccionarios en Java usando árboles AVL (usa una clase `dataStructures.dictionary.AVLDictionary.java`). Define todos los métodos de la interfaz `Dictionary<K,V>`.

```
public interface Dictionary<K, V> {
   boolean isEmpty();
   int size();
   void insert(K k, V v);
   V valueOf(K k);
   boolean isDefinedAt(K k);
   void delete(K k);
   Iterable<K> keys();
   Iterable<V> values();
   Iterable<Tuple2<K,V>> keysValues();
}
```

13. (Java) Implementa sacos usando árboles AVL (en `dataStructures.bag.AVLBag.java`). Las claves serán los elementos contenidos en el saco y los valores representarán las ocurrencias de la clave. Compara la eficiencia de esta implementación con la lineal desarrollada en el tema 3 y comprobueba experimentalmente los resultados.

14. (Haskell) Las expresiones aritméticas pueden representarse en forma arbórea con los operadores en los nodos internos del árbol y los valores (por ejemplo, números enteros) en las hojas. Así, consideremos la siguiente declaración de tipo:

```
data Expr = Value Integer
          | Add Expr Expr
          | Diff Expr Expr
          | Mult Expr Expr
          deriving Show
```

Por ejemplo, la expresión (1+2)*3 se representa como:

```
e1 :: Expr
e1 = Mult (Add (Value 1) (Value 2)) (Value 3)
```

a) Define la siguiente función para evaluar tales expresiones:

```
evaluate :: Expr -> Integer
```

b) El recorrido en post-orden de un árbol de tipo `Expr` proporciona su representación en la notación polaca inversa (RPN: Reverse Polish Notation). Define la función `toRPN :: Expr -> String`, que devuelve la lista de caracteres de la representación en RPN del argumento. Por ejemplo:

```
Main> toRPN e1
"1 2 + 3 *"
```

c) Consideremos ahora la siguiente función de plegado de expresiones:

```
foldExpr :: (Integer -> a) ->
            (a -> a -> a) ->
            (a -> a -> a) ->
            (a -> a -> a) ->
            Expr -> a
foldExpr ifValue ifAdd ifDiff ifMult e = fun e
  where
    fun (Value x)    = ifValue x
    fun (Add e1 e2)  = ifAdd (fun e1) (fun e2)
    fun (Diff e1 e2) = ifDiff (fun e1) (fun e2)
    fun (Mult e1 e2) = ifMult (fun e1) (fun e2)
```

Esta función *reemplaza* los constructores `Value`, `Add`, `Diff` y `Mult` por las funciones `ifValue`, `ifAdd`, `ifDiff` y `ifMult` respectivamente. Da una definición alternativa de `evaluate` usando `foldExpr`:

```
evaluate' :: Expr -> Integer
evaluate' e = foldExpr ...
```

15. (Haskell) La imagen especular de un árbol.

a) Define una función `mirrorB` que tome un árbol binario y devuelva su imagen especular:

```
data TreeB a = EmptyB | NodeB a (TreeB a) (TreeB a) deriving Show
mirror :: TreeB a -> TreeB a
```

Por ejemplo:

```
Main> mirror (NodeB 1 (NodeB 2 EmptyB (Node 3 EmptyB EmptyB)) Empty
NodeB 1 Empty (NodeB 2 (Node 3 EmptyB EmptyB) Empty)
```

b) Prueba por inducción estructural que `mirrorB . mirrorB = id`

Para ello, prueba el caso base y el paso inductivo siguientes:

Caso Base: `(mirrorB . mirrorB) EmptyB = EmptyB`

Paso Inductivo: Si `(mirrorB . mirrorB) lt = lt`, y además `(mirrorB . mirrorB) rt = rt`

entonces `(mirrorB . mirrorB) (NodeB x lt rt) = NodeB x lt rt`

16. (Haskell) Un árbol binario se dice simétrico si el hijo derecho es la imagen especular del hijo izquierdo. Escribe una función `isSymmetricB` que compruebe si un árbol binario es simétrico.

17. (Haskell) Recuerda que una hoja con valor `z` para un `TreeB` es el nodo sin hijos `NodeB z EmptyB EmptyB`, y que un nodo que no sea una hoja se llama *nodo interno*.

a) Define la función `leafsB` que toma un `TreeB` y devuelve una lista con los valores de las hojas.

b) Define una función `internalsB` que toma un `TreeB` y devuelve una lista con los valores de los nodos internos.

c) Define las funciones `leafs` e `internals` similares a las anteriores, pero para árboles generales.

18. (Haskell) A veces interesa representar árboles binarios de caracteres con una notación como:

```
a(b(d,e),c(,f(g,)))
```

Escribe funciones para convertir un `TreeB String` en la correspondiente `String`, y viceversa; por ejemplo:

```
Main> stringToTreeB "x(y,a(,b))"
NodeB 'x' (NodeB 'y' EmptyB EmptyB) (NodeB 'a' EmptyB (NodeB 'b' EmptyB EmptyB))

Main> treeToString $ NodeB 'x' (NodeB 'y' EmptyB EmptyB) (NodeB 'a' EmptyB (NodeB
'b' EmptyB EmptyB))
"x(y,a(,b))"
```

19. (Haskell) Definimos la *longitud interna* (*internal path length*) de un árbol general como la suma total de las longitudes de los caminos desde la raíz al resto de nodos. Según esta definición, para el siguiente árbol la longitud interna es 9:

```
data Tree a = Empty | Node a [Tree a] deriving Show
tree :: Tree Char
tree = Node 'a' [ Node 'f' [Node 'g' []]
                , Node 'c' []
                , Node 'b' [Node 'd' [], Node 'e' []]
                ]
```

Define la función `internalPL` que calcula la longitud interna de su argumento.

20. (Haskell) Un camino final o camino raíz-hoja, o camino maximal, es una secuencia de nodos recorridos desde la raíz hasta una hoja. Escribe la función `allMaxPaths` que devuelve la lista de todos estos caminos para un árbol general.

21. (Haskell) Recordemos la función vista en clase (transparencias del tema 4) para comprobar si un BST es realmente un árbol binario de búsqueda:

```
isBST :: (Ord a) => BST a -> Bool
isBST Empty          =  True
isBST (Node x lt rt) =  forAll (<x) lt && forAll (>x) rt
                        && isBST lt && isBST rt
 where
   forAll :: (a -> Bool) -> BST a -> Bool
   forAll p Empty          =  True
   forAll p (Node x lt rt) =  p x && forAll p lt && forAll p rt
```

que está definida en `DataStructures.SearchTree.BST`. La función anterior es simple porque expresa exactamente la definición de un BST. Sin embargo es ineficiente porque los nodos son visitados varias veces. Este problema puede resolverse eficientemente a través de una función auxiliar con dos parámetros adicionales:

```
inRange :: (Ord a) => a -> a -> BST a -> Bool
```

de forma que la llamada `inRange min max t` comprueba si todas las claves del árbol están en el intervalo [min,max]. Los valores iniciales para min y max serán las cotas correspondientes al tipo base del árbol. En Haskell estos valores extremos pueden determinarse por las funciones `minBound` y `maxBound` de la clase `Bounded`, y de esta forma la llamada inicial será:

```
isBST' :: (Bounded a, Ord a) => BST a -> Bool
isBST' t = inRange minBound maxBound t
```

Define la función `inRange` para que `isBST'` compruebe eficientemente si su argumento es un BST.

22. (Haskell) Escribe una función `findTreeB` que reconstruya un árbol binario a partir de las dos listas correspondientes a los recorridos *pre-orden* y *en-orden* (transparencias del tema 4). Puedes asumir que no existen elementos repetidos:

```
Main> findTreeB [2,1,3,4,6] [3,1,4,2,6]
NodeB 2 (NodeB 1 (NodeB 3 EmptyB EmptyB) (NodeB 4 EmptyB EmptyB)) (NodeB
6 EmptyB EmptyB)
```

23. (Java) Resuelve el problema anterior en Java.

24. Explica cómo puede usarse una cola con prioridad (priority queue) para implementar una cola (queue) o una pila (stack).

25. (Tomado de Sedgewick and Wayne, Computational number theory, 2010). Escribe un programa que muestre en orden creciente los enteros de la forma $a^3 + b^3$, donde a y b son enteros entre 0 y N, de forma que el programa no haga un uso excesivo de espacio. Es decir, en lugar de calcular el array de las $(N+1)^2$ sumas ($a^3 + b^3$) y después ordenarla, usa una cola con prioridad que inicialmente contenga las ternas (0^3, 0, 0), (1^3, 1, 0), (2^3, 2, 0), . . . , (N^3, N, 0); el orden considerado será el orden lexicográfico. A partir de esta cola inicial repetid las siguientes acciones mientras la cola no sea vacía: extraed la menor terna ($i^3 + j^3$, i, j) y visualizarla; seguidamente, si fuera $j < i$, insertamos la terna ($i^3 + (j+1)^3$,i, j+1). De esta forma la cola solo contendrá ternas ($i^3 + j^3$, i, j) con $j<=i$, y las soluciones simétricas triviales ($i^3 + j^3$, i, j) con $j>i$ pueden obviarse.

Usa este programa para encontrar los enteros a, b, c, d entre 0 y 10^6 tales que $a^3 + b^3 = c^3 + d^3$.

26. (Java) Escribe un programa para contar las ocurrencias de las palabras de un fichero de texto usando un árbol AVL cuyas claves sean palabras y cuyos valores sean las ocurrencias de éstas. El programa contendrá probablemente código tal como:

```
if (!avl.isElem(word)) avl.insert(word, 1);
else avl.insert(word, avl.search(word) + 1);
```

Esto es ineficiente ya se recorre varias veces el árbol. La técnica llamada *Software Caching* puede utilizarse para mejorarlo: salvamos la posición de la búsqueda de la clave más reciente en una variable de instancia de la clase árbol, de forma que en la siguiente búsqueda para la misma clave se conoce la posición. Añade esta optimización en la implementación de los árboles AVL.

27. (Java) Añade implementaciones eficientes de las siguiente operaciones sobre BSTs:

- `K floor(K key)`: la mayor clave menor o igual al argumento.
- `K ceiling(K key)`: la menor clave mayor o igual al argumento.
- `int rank(K key)`: número de claves menores que el argumento.
- `K select(int i)`: la i-ésima clave en orden de menor a mayor (i=0 para la mínima).
- `void deleteMin()`: elimina la menor clave.
- `void deleteMax()`: elimina la mayor clave.
- `int size(Key lo, Key hi)`: número de claves en el intervalo `[lo,hi]`.

Nota: con objeto de realizar una implementación eficiente de estas operaciones, el BST deberá representarse como un árbol aumentado.