Examen de Febrero 2017

Un vector es una colección homogénea de elementos a los que se accede a través de su índice (un número natural). El tamaño (*size*) de un vector es el número total de elementos que almacena; para simplificar supondremos que el tamaño de un vector es siempre una potencia de dos (es decir, el tamaño es de la forma 2^n, para $n \geq 0$). Observa que no existe el vector vacío. El índice (*index*) del primer elemento de un vector es 0; el índice del último elemento es *size*-1. Todos los intentos de acceso a un vector con un índice fuera de rango deben señalarse con la correspondiente excepción (Java) o error (Haskell).

El objetivo del ejercicio es implementar vectores *esparcidos* (*sparse vectors*); se trata de una representación compacta de vectores que tienen muchos elementos consecutivos con el mismo valor.

Haskell

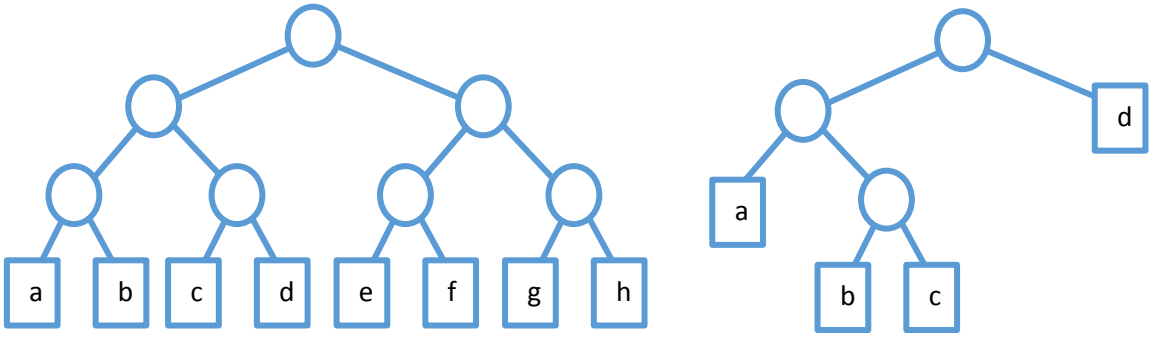
Descarga del campus virtual el archivo comprimido que contiene las fuentes para resolver el problema y comprobar tu solución. Completa las definiciones de funciones del fichero **`SparseVector.hs`**. Este es el único fichero que tienes que modificar y el único fichero que debes subir a través del enlace de entrega del campus virtual. Ten en cuenta que tu solución debe compilar sin errores para que se considere adecuada.

Los elementos que aparezcan en un vector esparcido deben almacenarse en valores del tipo **`Tree`**:

```
data Tree a = Unif a | Node (Tree a) (Tree a)
```

donde **a** es el tipo de los elementos almacenados en el vector. Un valor **`Unif x`** representa un vector con *todos* sus elementos *iguales* a **`x`**. Un valor **`Node lt rt`** representa un vector con elementos *diferentes*. Los elementos almacenados en la primera parte del vector (con índices entre 0 y (*size*/2) - 1) están almacenados en el vector **`lt`** mientras que el vector **`rt`** almacena los elementos de la segunda parte (con índices entre *size*/2 y *size*-1). Esta descomposición de vectores en dos mitades (**`Node`**) se aplica recursivamente, siempre que las mitades contengan elementos diferentes. Cuando todos los elementos del vector son iguales entre sí, ya no se descompone más (**`Unif`**). Nos referiremos a esta descomposición recursiva como el **Invariante del Árbol** de un vector esparcido. Puedes suponer que todas las funciones recibirán como argumentos árboles que satisfagan este invariante; y tendrás que *asegurar* que los árboles devueltos por estas funciones también verifican este invariante.

Por ejemplo, considera los siguientes dos árboles que representan dos vectores dispersos de 2^3 = 8 elementos cada uno:



En la figura, un círculo representa un **Node** mientras que un cuadrado representa un **Unif**. Para el árbol de la izquierda, los valores de los elementos del vector (desde el índice 0 hasta el índice 7) son **[a,b,c,d,e,f,g,h]**. Para el árbol de la derecha, los elementos son **[a,a,b,c,d,d,d,d]**.

Un vector esparcido se representará por el siguiente tipo:

```
data Vector a = V Int (Tree a)
```

donde el **Int** almacena el tamaño del vector y el **Tree** almacena sus elementos, de acuerdo con la descripción anterior.

a) (0.5 puntos) Define la función **vector** que toma un entero **n** y un valor **x** y devuelve un **Vector** de tamaño 2^n con todos sus elementos iguales a **x**. Si n es negativo, la función debe producir un error.
b) (0.5 puntos) Define la función **size**, que toma un **Vector** como parámetro y devuelve su tamaño.
c) (1 punto) Define una función (privada) **simplify**, que toma dos **Tree**s correspondientes a la primera y segunda mitad de un vector esparcido y devuelve un **Tree** correspondiente al vector completo. Ten en cuenta que si ambas mitades contienen el mismo valor, el árbol devuelto debe ser de la forma **Unif**.
d) (1.5 puntos) Define una función **get**, que toma un entero (correspondiente a un índice) y un **Vector**, y devuelve el valor del elemento almacenado en esa posición del vector.
e) (2 puntos) Define una función **set**, que toma un entero (correspondiente a un índice), un valor **x** y un **Vector**, y devuelve el nuevo vector que se obtiene al reemplazar en el vector de entrada el elemento almacenado en la posición indicada por el índice con el valor **x**.
f) (1 punto) ¿Cuál es la complejidad asintótica de las diferentes operaciones de esta estructura de datos?
g) (1 punto) Define una función **mapVector** que tome como parámetros una función y un vector disperso y que devuelva el nuevo vector disperso que se obtiene al aplicar la función argumento a todos los elementos del vector proporcionado.

Sólo para alumnos a tiempo parcial

h) (1.25 puntos) Sin usar **get** y trabajando directamente con la estructura **Tree**, define la función **filterVector** que tome un predicado (con tipo **a -> Bool**) y un vector disperso, y devuelva una lista con los elementos del vector que verifican el predicado. El número de repeticiones de cada elemento en la lista de salida debe coincidir con el nº de elementos en el vector con dicho valor.
i) (1.25 puntos) Define una función **depthOf** que tome un índice **i** y un vector disperso y que devuelva la profundidad dentro del árbol a la que se encuentra almacenado el dato **i**-ésimo del vector.

Java

Implementaremos los vectores esparcidos en Java usando la misma representación que en Haskell, respetando el **Invariante del Árbol** de un vector esparcido anteriormente descrito.

Descarga del campus virtual el archivo comprimido que contiene el proyecto Eclipse para resolver el problema y comprobar tu solución. Completa las definiciones de métodos del fichero **`dataStructures\vector\SparseVector.java`**. Este es el único fichero que tienes que modificar y el único fichero que debes subir a través del enlace de entrega del campus virtual. Ten en cuenta que tu solución debe compilar sin errores para que se considere adecuada.

Los elementos que aparezcan en un vector esparcido deben almacenarse en clases que implementen la interfaz:

```
protected interface Tree<T> {
    T get(int sz, int i);
    Tree<T> set(int sz, int i, T x);
}
```

donde **`T`** es el tipo de los elementos del vector. El método **`get`** toma como parámetros el tamaño del vector **`sz`** y un índice **`i`** y devuelve el elemento almacenado en la posición **`i`** del vector. El método **`set`** toma los mismos parámetros y un valor **`x`**, y devuelve el **`Tree`** que se obtiene tras reemplazar en el vector de entrada el elemento almacenado en la posición **`i`** por el valor **`x`**.

Hay dos clases que implementan esta interfaz. La primera corresponde a un vector con todos los elementos iguales:

```
protected static class Unif<T> implements Tree<T> {
    T elem;
    public Unif(T e) { elem = e; }
    public T get(int sz, int i) { to be completed }
    public Tree<T> set(int sz, int i, T x) { to be completed }
}
```

La segunda clase se utiliza para representar vectores que almacenan valores *diferentes*, donde **`left`** (**`right`**) almacenan la primera (segunda) mitad del vector, respectivamente:

```
protected static class Node<T> implements Tree<T> {
    Tree<T> left, right;
    public Node(Tree<T> l, Tree<T> r) { left = l; right = r; }
    public T get(int sz, int i) { to be completed }
    public Tree<T> set(int sz, int i, T x) { to be completed }
    public Tree<T> simplify() { to be completed }
}
```

a) (0.75 + 1 puntos) Completa las implementaciones de **`get`** y **`set`** de la clase **`Unif`**. Puedes suponer que los índices pasados como parámetros son válidos (es decir, pertenecen al rango de 0 a **`sz`**-1).

b) (1+1+1 puntos) Completa las implementaciones de **simplify**, **get** y **set** de la clase **Node**. Puedes suponer que los índices pasados como parámetros son válidos (es decir, pertenecen al rango de 0 a **sz**-1).
Recuerda que puedes comprobar si una variable **t** de tipo **Tree** es un nodo **Unif** usando el operador **instanceof** de Java (**t instanceof Unif<?>**) y que, si el resultado de esta comprobación es **true**, puedes realizar el correspondiente *casting* (**Unif<T> u = (Unif<T>) t**).

Un vector esparcido se representa mediante un **int** correspondiente a su tamaño y un **Tree**, que almacena sus elementos según se ha descrito anteriormente:

```
public class SparseVector<T> {
      private int size;
      private Tree<T> root;
      public SparseVector(int n, T elem) { to be completed }
      public int size() { to be completed }
      public T get(int i) { to be completed }
      public void set(int i, T x) { to be completed }
}
```

c) (0.5 puntos) Completa la implementación del constructor, que inicializa el vector de manera que su tamaño es 2^n y sus valores son todos iguales a **elem**. Si n es negativo, debes elevar una excepción del tipo **VectorException**.
d) (0.25 puntos) Completa la implementación del método **size**, que devuelve el número de elementos almacenados en el vector.
e) (0.5 puntos) Completa la implementación del método **get**, que devuelve el elemento situado en la posición **i** del vector. Ten en cuenta que el índice facilitado puede no ser válido (eleva una excepción del tipo **VectorException** en dicho caso)
f) (0.5 puntos) Completa la implementación del método **set**, que asigna el valor **x** al elemento del vector almacenado en la posición **i**. Ten en cuenta que el índice facilitado puede no ser válido (eleva una excepción del tipo **VectorException** en dicho caso).
g) (1 punto) Define un iterador que devuelve todos los elementos de un vector (comenzando por el elemento almacenado en la posición 0 y siguiendo por orden creciente del índice).

Sólo para alumnos a tiempo parcial

h) (1.25 puntos) Define un constructor de copia para la clase **SparseVector**. Este constructor debe tomar como argumento un vector disperso y devolver otro idéntico, pero sin compartir memoria con el vector dado.
i) (1.25 puntos) Define un método **depthOf** que tome un índice **i** y que devuelva la profundidad dentro del árbol a la que se encuentra almacenado el dato **i**-ésimo del vector.