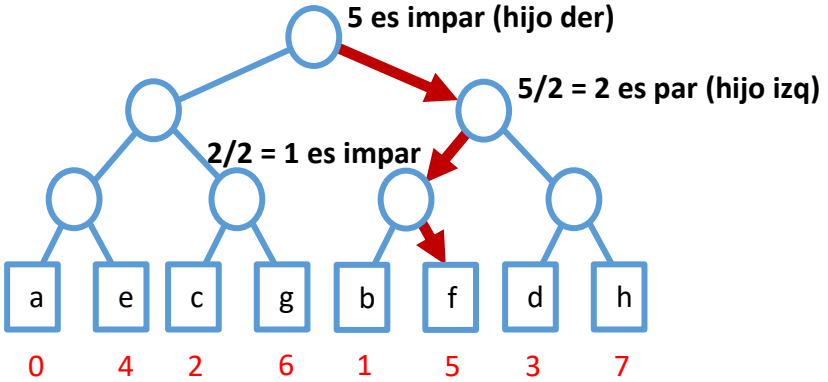
Estructuras de Datos
Grado en Ingeniería Informática, del Software y Computadores
Práctica 7 – Vectores

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 de elementos que almacena; para simplificar supondremos que el tamaño de un vector es siempre una potencia de 2 (es decir, el tamaño es de la forma 2^n, para $n \geq 0$). Observa que no existe el vector vacío. El índice 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 el correspondiente error (Haskell) o excepción (Java).

El objetivo de la práctica es implementar vectores mediante **árboles binarios perfectos con información solo en las hojas**. Por ejemplo, el vector de 8 caracteres:

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

se representa por el árbol:



Observa que elementos consecutivos en el vector no aparecen en hojas consecutivas en el árbol. En la figura, debajo de cada hoja, aparece el índice del componente del vector al que corresponde. La hoja que corresponde al elemento *i*-ésimo del vector se localiza partiendo de la raíz del árbol, descendiendo por la izquierda si *i* es par o por la derecha si *i* es impar. Este proceso se repite en el subárbol apropiado dividiendo en cada paso el índice por 2, hasta alcanzar la hoja. En la figura se ha anotado la búsqueda de la hoja correspondiente a la posición 5.

Haskell

Descarga del campus virtual el archivo comprimido que contiene el código fuente para resolver el problema y comprobar tu solución. Completa las definiciones de funciones del fichero **`TreeVector.hs`**. Este es el único fichero que tienes que modificar.

Los elementos de un vector se almacenan en valores del tipo **`Vector a`**, definido del siguiente modo:

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

donde la componente con tipo **`Int`** representa el tamaño o número de elementos del vector y la de tipo **`Tree a`** es el árbol binario perfecto que almacena sus elementos. Los árboles binarios se representan a su vez por el tipo **`Tree a`** del siguiente modo:

```
data BinTree a = Leaf a | Node (BinTree a) (BinTree a)
```

donde **`Leaf`** representa un nodo hoja y **`Node`** un nodo interno.

Resuelve los siguientes apartados **escribiendo**, además, **el tipo de cada función definida**:

a) 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) Define la función **`size`** que toma un **`Vector`** y devuelve su tamaño.

c) Define una función **`get`** que toma un entero (correspondiente a un índice) y un **`Vector`**, y devuelve, si es posible, el valor del elemento almacenado en esa posición del vector. En caso contrario debe elevar una excepción.

d) Define una función **`set`** que toma un entero (correspondiente a un índice), un valor **x** y un **`Vector`**, y devuelve, si es posible, 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**. En caso contrario debe elevar una excepción.

e) Define una función **`mapVector`** que toma una función **`f`** y un **`Vector`**, y devuelve el nuevo vector que se obtiene al aplicar **`f`** a todos los elementos del vector proporcionado.

f) Define una función **`intercalate`** que toma dos listas y devuelve una lista con los elementos de ambas listas intercalados. En la intercalación, los elementos situados en posiciones pares (consideramos que la posición de la cabeza es 0) provienen de la primera lista y los situados en las impares de la segunda lista. Si las listas no tienen el mismo número de elementos, se descartan los elementos sobrantes.

g) Usando la función **`intercalate`** del apartado anterior y sin usar **get** o **set**, define una función **`toList`** que toma un **`Vector`** y devuelve una lista con los elementos del vector. El elemento almacenado en la posición *i*-ésima del vector debe aparecer en la posición *i*-ésima de la lista.

h) ¿Cuál es la complejidad asintótica de las diferentes operaciones de esta estructura de datos implementadas hasta ahora?

i) Define una función booleana **`isPowerOfTwo`** que dado un entero no negativo indique si es potencia de 2.

j) Sin usar **get** o **set** (es decir, construyendo directamente el árbol), define una función **`fromList`** que toma una lista cuya longitud debe ser potencia de 2 y devuelve un **`Vector`** de manera que el elemento *i*-ésimo de la lista aparezca en la posición *i*-ésima del vector. Si la longitud de la lista no es potencia de 2 debe devolver error.

Java

Implementaremos los vectores en Java usando la misma representación que en Haskell. Descarga del campus virtual el archivo comprimido que contiene el código fuente para resolver el problema y comprobar tu solución. Completa las definiciones de métodos del fichero **`dataStructures\vector\TreeVector.java`**. Este es el único fichero que tienes que modificar.

Los elementos del vector se almacenan en clases que implementan la interfaz:

```
private interface Tree<T> {
    T get(int i);
    void set(int i, T x);
    List<T> toList();
```

```
}
```

donde **T** es el tipo de los elementos del vector. El método **get** toma como parámetro un índice **i** y devuelve el elemento almacenado en la posición **i** del vector. El método **set** toma como parámetros un índice **i** y un valor **x**, y reemplaza el elemento almacenado en la posición **i** por el valor **x**. El método **toList** devuelve una lista con los elementos del vector.

Hay dos clases que implementan esta interfaz. La primera corresponde a un nodo hoja:

```
private static class Leaf<T> implements Tree<T> {
     private T value;
     public Leaf(T e) { value = e; }
     public T get(int i) { ... }
     public void set(int i, T x) { ... }
     public List<T> toList() { ... }
}
```

La segunda clase se utiliza para representar nodos internos:

```
private static class Node<T> implements Tree<T> {
  private Tree<T> left, right;
  public Node(Tree<T> l, Tree<T> r) { left = l; right = r; }
  public T get(int i) { ... }
  public void set(int i, T x) { ... }
  public List<T> toList() { ... }
}
```

a) Completa las implementaciones de **get** y **set** de la clase **Leaf**. Puedes suponer que los índices pasados como parámetros son válidos (es decir, pertenecen al rango 0 a **size**-1).
b) Completa las implementaciones de **get** y **set** de la clase **Node**. Puedes suponer que los índices pasados como parámetros son válidos (es decir, pertenecen al rango 0 a **size**-1).

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

```
public class TreeVector<T> {
     private int size;
     private Tree<T> root;
     public TreeVector(int n, T value) { ... }
     public int size() { ... }
     public T get(int i) { ...}
     public void set(int i, T x) { ...}
     public List<T> toList() { ... }
     private static <E> List<E> intercalate(List<E> xs, List<E> ys)
         {...}
     }
```

c) Completa la implementación del constructor, que inicializa el vector de manera que su tamaño es 2^n y sus valores son todos iguales a **`value`**. Si **`n`** es negativo, debes elevar una excepción del tipo **`VectorException`**.

d) Completa la implementación del método **`size`**, que devuelve el número de elementos almacenados en el vector.

e) 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) 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) Completa la definición del método estático **`intercalate`** que dadas dos listas devuelve su intercalación, tal como se ha descrito anteriormente.

h) Usando el método **`intercalate`** y sin usar **get** o **set**, completa las definiciones de los métodos **`toList`** de las clases **`Leaf`**, **`Node`** y **`Tree`**, que dado un vector devuelve una lista con sus elementos, tal como se ha descrito anteriormente.

i) Define un método estático **`isPowerOfTwo`** que dado un entero no negativo devuelva **`true`** si es potencia de 2 y **`false`** en caso contrario.

j) Sin usar **get** o **set** (es decir, construyendo directamente el árbol), define un método estático **fromList** que toma una lista cuya longitud debe ser potencia de 2 y devuelve un **Vector** de manera que el elemento *i*-ésimo de la lista aparezca en la posición *i*-ésima del vector. Si la longitud de la lista no es potencia de 2 debe elevar una excepción **`VectorException`**.