Estructuras de datos

Aclaraciones:

- *Esta evaluación es a libro abierto. Se pueden usar todas las funciones y propiedades vistas en la práctica y en la teórica, aclarando la referencia. Cualquier otra función o propiedad que se utilice* ***debe ser definida o demostrada****.*
- *No se olvide de poner nombre, nro. de hoja y cantidad total de hojas en cada una de las hojas. Tampoco olvide poner un título al principio del examen y dejar espacio para que se pueda ver como un trabajo y no como una hoja llena de borrones...*
- *Le recomendamos leer el enunciado en su totalidad y organizar sus ideas antes de comenzar la resolución.*
- *¿Sabía que reusar código es una forma muy eficiente de disminuir el tiempo necesario para programar..., no?*
- *Recuerde que la intención es medir cuánto comprende usted del tema. Por ello, no dude en escribir todo lo que sabe, en explicar lo que se propone antes de escribir código, en probar sus funciones con ejemplos, etc.*

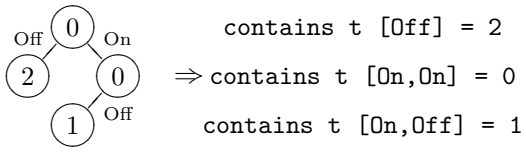
Ejercicio 1

Para representar conjuntos múltiples[1] de *strings* (cadenas) de bits, se puede utilizar el siguiente tipo de datos de árboles binarios de bits (o `BitTree`s), donde cada una de las ramas del árbol codifica una cadena de `Bit`s (`Off` o `On`):

```
data BitTree = Nil | Node Int BitTree BitTree
data Bit     = Off | On
type Chain   = [Bit]
```

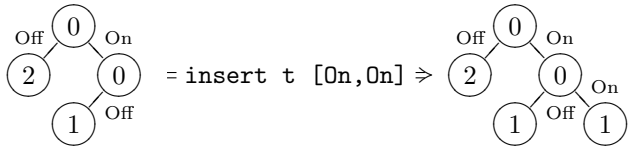
En un `BitTree` los enlaces a hijos izquierdos representan un bit en `Off`, mientras que los enlaces a hijos derechos representan un bit en `On`. Esto significa que una cadena de bits está en el conjunto si en el árbol si existe una rama que codifica la cadena. Además en los nodos se almacena un entero que indica la cantidad de repeticiones de esa cadena. La estructura mantiene un invariante de representación que prohíbe que una hoja del árbol tenga asociado una cantidad de repeticiones igual a cero.
Implementar las siguientes operaciones indicando la complejidad de cada una en peor caso:

a) `contains :: BitTree -> Chain -> Int`
Que retorna la cantidad de repeticiones de una cadena en un árbol de bits (0 si la cadena no está definida).

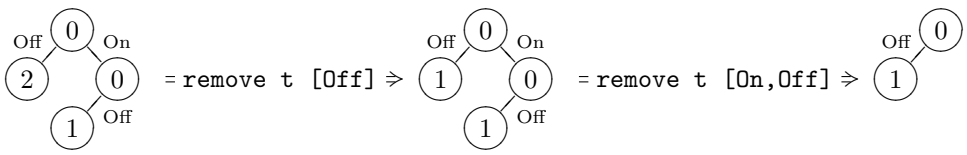



```
contains t [Off] = 2
⇒ contains t [On,On] = 0
contains t [On,Off] = 1
```

b) `insert :: BitTree -> Chain -> BitTree`
Que inserta una ocurrencia de la cadena en un árbol de bits.



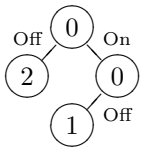
c) `remove :: BitTree -> Chain -> BitTree`
Que remueve una ocurrencia de una cadena en un árbol de bits.



d) `elements :: BitTree -> [(Int,Chain)]`
Que devuelve una lista de las cadenas definidas en el árbol asociadas a sus repeticiones.

[1] conjuntos que pueden tener varias repeticiones de un elemento

Ayuda: Considere utilizar una función auxiliar que tome como parámetro la rama recorrida hasta el momento.



⇒ `elements t = [(2,[Off]),(1,[On,Off])]`

Ejercicio 2
Considere, ahora, el tipo de los árboles de Huffman (HTree):

```
data HTree = HLeaf Int | HBin Int HTree HTree
```

Los árboles de Huffman son árboles binarios que almacenan en sus nodos un peso, tanto en las hojas (`HLeaf`) como en los nodos internos (`HBin`). Los árboles de Huffman son conocidos por su utilidad a la hora de programar algoritmos de compresión. Estos árboles respetan el invariante de representación que el peso de los nodos internos es la suma de los pesos de cada uno de sus dos hijos. Observe que el invariante no restringe de ninguna manera el peso de las hojas.
Implementar las siguientes operaciones útiles en el procedimiento de compresión:

a) `weigth :: HTree -> Int`
Que retorna el peso de árbol de Huffman.

b) `weigthedSize :: HTree -> Int`
Que calcula el "tamaño ponderado" de un árbol de Huffman, que se calcula como la suma de los pesos de cada hoja multiplicada por la altura en la que se encuentra (considere que la raíz está a altura 1 y cada hijo a un nivel de altura más que el padre).

c) `assemble :: PriortyQueue pq => pq HTree -> HTree`
Que genera un `HTree` a partir de una cola de prioridad de `HTree` (donde se considera que el árbol de menor peso tiene mayor prioridad en la cola). **Nota:** `assemble` trabaja combinando los dos nodos de menor peso (mayor prioridad) en un nodo binario con la suma de ambos y así sucesivamente hasta quedarse con un único nodo (que debe retornarse).

Ejercicio 3
Implementar una función `compress :: [Chain] -> [Chain]` que dada una lista de cadenas, genera la lista de cadenas en su codificación comprimida.
Para realizar esta función, asuma (sin necesidad de codificarlas) la existencia de las siguientes funciones

- una función `genHTree:: [Chain] -> HTree`, que dada una lista de cadenas sin codificar, obtiene el `HTree` que codifica dicha lista como conjunto múltiple de cadenas.
- una función `genTable :: Map m => HTree -> m Chain Chain`, que dado un `HTree` representando un conjunto múltiple de cadenas, obtiene un diccionario que a cada cadena sin comprimir le hace corresponder su codificación comprimida.

Ayuda: Considere definir una función auxiliar `compressWith :: Map m => m Chain Chain -> [Chain] -> [Chain]` que dado un diccionario que asocia cadenas con su codificación comprimida, y una lista de cadenas, genera la lista de cadenas en su codificación comprimida, y combínela con las funciones provistas.

Referencia

Considere las siguiente interfaces para los tipos abstractos de datos `PriorityQueue` y `Map`:

```
class PriorityQueue q where
  emptyPQ   :: q a
  isEmptyPQ :: q a -> Bool
  enqueuePQ :: Ord a => q a -> a -> q a
  dequeuePQ :: Ord a => q a -> (a, q a)
  sizePQ    :: q a -> Int

class Map m where
```

```
emptyM  :: m k v
defineM :: Ord k => m k v -> k -> v -> m k v
removeM :: Ord k => m k v -> k -> v -> m k v
lookupM :: Ord k => m k v -> k -> Maybe v
domM    :: Eq k  => m k v -> [k]
```