Estructura de Datos. Práctica 5

Cola de Prioridades

Configuración

Para la implementación de las operaciones de una cola de prioridad se proporciona una plantilla y un fichero con axiomas:

- `LinearPriorityQueue.hs` plantilla donde se tendrá que implementar la cola de prioridad. ***Aquí es donde debéis trabajar y será el único fichero a entregar***.
- `PriorityQueueAxioms.hs` proporciona axiomas de la cola de prioridad. ***No hay que modificarlo, solo usarlo***.

Para la configuración, se debe crear el directorio `DataStructures/PriorityQueue` y ***colocar en él los dos ficheros***.

Situados en ese directorio, debemos lanzar el intérprete de Haskell `ghci` y una vez dentro ejecutar `:set -i../..` para indicar el directorio donde se encuentra la raíz del módulo. Para trabajar habrá que leer el fichero ejecutando `:l LinearPriorityQueue.hs`.

Cuestiones

El ejercicio a resolver es la implementación de una cola de prioridades. Una cola de prioridad es una cola que mantiene los elementos ordenados (el primero es el menor) de manera que al introducir un elemento se colocará justo delante de los que sean mayores que él. El tipo para representar la cola de prioridades será:

```haskell
data PQueue a = Empty | Node a (PQueue a)
```

La cabecera del módulo es la siguiente:

```haskell
module DataStructures.PriorityQueue.LinearPriorityQueue
  ( PQueue
-- basicas
  , empty
  , isEmpty
  , first
  , dequeue
  , enqueue
-- extras
  , mkPQ
  , mapPQ
  , filterPQ
  , foldrPQ
  , fromPQ
  , toPQ
  , toList
  ) where
```

donde se ve que hay funcionalidad básica y funcionalidad extra. Salvo `enqueue`, el resto de funciones básicas son similares a las de una cola. Aquí se presenta un ejemplo de uso de la funcionalidad básica del módulo:

```
> q = enqueue 3 (enqueue 5 (enqueue 2 (enqueue 1 (enqueue 2 empty))))
LinearPriorityQueue(1,2,2,3,5)
```

```
> first q
1
> dequeue q
LinearPriorityQueue(2,2,3,5)
```

La funcionalidad extra tiene el siguiente comportamiento:

`mkPQ :: (Ord a) => [a] -> PQueue a` crea una cola de prioridades a partir de una lista de elementos.

`mapPQ :: (Ord a, Ord b) => (a -> b) -> PQueue a -> PQueue b` aplica una función a cada elemento de la cola creando una cola de prioridades con los resultados. Hay que tener en cuenta que al aplicar la función el orden puede alterarse.

`filterPQ :: Ord a => (a -> Bool) -> PQueue a -> PQueue a` filtra los elementos de una cola por medio de un predicado.

`foldrPQ :: (Ord a) => (a -> b -> b) -> b -> PQueue a -> b` realiza un plegado de la cola partiendo del final de la misma.

`fromPQ :: (Ord a) => a -> PQueue a -> PQueue a` devuelve la sección de la cola a partir de los elementos que son mayores o iguales a un elemento dado.

`toPQ :: (Ord a) => a -> PQueue a -> PQueue a` devuelve la sección de la cola hasta los elementos menores a un elemento dado.

`toList :: (Ord a) => PQueue a -> [a]` crea una lista con los elementos de la cola.

`p_pqinversa :: Ord a => PQueue a -> Bool` crea en la plantilla la propiedad `p_pqinversa` que compruebe que `mkPQ` y `toList` son inversas una de la otra.

Complejidad completa en la plantilla la siguiente tabla con las complejidades de las operaciones indicadas.

Operación	Complejidad
empty	O(1)
isEmpty	
enqueue	
first	
dequeue	
mkPQ	
mapPQ	
filterPQ	
foldrPQ	
fromPQ	
toPQ	
toList	

En la plantilla se indican las operaciones que deben implementarse. Por defecto, todas están implementadas con `undefined`. En cada función se pueden utilizar las ecuaciones y patrones que se desee. Las funciones básicas, `empty`, `isEmpty`, `first`, `enqueue` y `dequeue` deben implementarse obligatoriamente para verificar los axiomas. Esto se consigue leyendo el fichero `PriorityQueueAxioms.hs` y ejecutando la función `priorityQueueAxioms` que pasa automáticamente todos los test.

En la plantilla hay algunos ejemplos comentados que se proporcionan para verificar el correcto funcionamiento de las funciones básicas y extras.