Trabajo Práctico # 4

Estructuras de Datos, Universidad Nacional de Quilmes

8 de abril de 2016

1. Conjunto

Un *Conjunto* es un tipo abstracto de datos que consta de las siguientes operaciones:

- `vacioC :: Conjunto a`
  Crea un conjunto vacío.
- `agregarC :: Eq a => a -> Conjunto a -> Conjunto a`
  Dados un elemento y un conjunto, agrega el elemento al conjunto.
- `perteneceC :: Eq a => a -> Conjunto a -> Bool`
  Dados un elemento y un conjunto indica si el elemento pertenece al conjunto.
- `cantidadC :: Eq a => Conjunto a -> Int`
  Devuelve la cantidad de elementos distintos de un conjunto
- `borrarC :: Eq a => a -> Conjunto a -> Conjunto a`
  Devuelve la cantidad de elementos distintos de un conjunto
- `unionC :: Eq a => Conjunto a -> Conjunto a -> Conjunto a`
  Dados dos conjuntos devuelve un conjunto con todos los elementos de ambos conjuntos.
- `listaC :: Eq a => Conjunto a -> [a]`
  Dado un conjunto devuelve una lista con todos los elementos distintos del conjunto.

1. Como *usuario* del tipo abstracto *Conjunto* implementar las siguientes funciones:
   - `losQuePertenecen :: Eq a => [a] -> Conjunto a -> [a]`
     Dados una lista y un conjunto, devuelve una lista con todos los elementos que pertenecen al conjunto.
   - `sinRepetidos :: [a] -> [a]`
     Quita todos los elementos repetidos de la lista dada utilizando un conjunto como estructura auxiliar.
   - `unirTodos :: Arbol (Conjunto a) -> Conjunto a`
     Dado un arbol de conjuntos devuelve un conjunto con la union de todos los conjuntos del arbol.

- `interseccionArbol :: Eq a => Tree a -> Tree a -> Conjunto a`
  Dados dos árboles devuelve un conjunto con los elementos que ambos árboles tienen en común.

2. Implementar la variante del tipo abstracto *Conjunto* con una lista que no tiene repetidos y guarda la cantidad de elementos en la estructura.
   Nota: la restricción *Eq* aparece en toda la interfaz se utilice o no en todas las operaciones de esta implementación, pero para mantener una interfaz común entre distintas posibles implementaciones estamos obligados a escribir así los tipos.

2. Queue (cola)

Una *Queue* es un tipo abstracto de datos de naturaleza FIFO (*first in, first out*). Esto significa que los elementos salen en el orden con el que entraron, es decir, los que se agrega primero es el primero en salir (como la cola de un banco). Su interfaz es la siguiente:

- `emptyQ :: Queue a`
  Crea una cola vacía.
- `isEmptyQ :: Queue a -> Bool`
  Dada una cola indica si la cola está vacía.
- `queue :: a -> Queue a -> Queue a`
  Dados un elemento y una cola, agrega ese elemento a la cola.
- `firstQ :: Queue a -> a`
  Dada una cola devuelve el primer elemento de la cola.
- `dequeue :: Queue a -> Queue a`
  Dada una cola la devuelve sin su primer elemento.

1. Como *usuario* del tipo abstracto *Queue* implementar las siguientes funciones:
   - `largoQ :: Queue a -> Int`
     Cuenta la cantidad de elementos de la cola.
   - `atender :: Queue Persona -> [Persona]`
     Dada una cola de personas, devuelve la lista de las mismas, donde el orden de la lista es de la cola.
   - `unirQ :: Queue a -> Queue a -> Queue a`
     Inserta todos los elementos de la segunda cola en la primera.
2. Implemente el tipo abstracto *Queue* utilizando listas. Los elementos deben encolarse por el final de la lista y desencolarse por delante.

3. Stack (pila)

Una *Stack* es un tipo abstracto de datos de naturaleza LIFO (*last in, first out*). Esto significa que los últimos elementos agregados a la estructura son los primeros en salir (como en una pila de platos). Su interfaz es la siguiente:

- `emptyS :: Stack a`
  Crea una pila vacía.
- `isEmptyS :: Stack a -> Bool`
  Dada una pila indica si está vacía.
- `push :: a -> Stack a -> Stack a`
  Dados un elemento y una pila, agrega el elemento a la pila.
- `top :: Stack a -> a`
  Dada un pila devuelve el elemento del tope de la pila.
- `pop :: Stack a -> Stack a`
  Dada una pila devuelve la pila sin el primer elemento.

1. Como *usuario* del tipo abstracto *Stack* implementar las siguientes funciones:
   - `apilar :: [a] -> Stack a`
     Dada una lista devuelve una pila sin alterar el orden de los elementos.
   - `balanceado :: String -> Bool`
     Toma un string que representa una expresión aritmética, por ejemplo "$(2+3)*2$", y verifica que la cantidad de paréntesis que abren se corresponda con los que cierran. Para hacer esto utilice una stack. Cada vez que encuentra un paréntesis que abre, lo apila. Si encuentra un paréntesis que cierra desapila un elemento. Si al terminar de recorrer el string se desapilaron tantos elementos como los que se apilaron, ni más ni menos, entonces los paréntesis están balaceados.
2. Implementar el tipo abstracto *Stack* utilizando listas.

4. Queue con longitud constante

Extender la interfaz de *Queue* con la siguiente operación:

- `lenQ :: Queue a -> Int`
  Devuelve la cantidad de elementos

5. Conjunto con Máximo

Extender la interfaz de *Conjunto* con la siguiente operación:

- `maximoC :: Ord a => Conjunto a -> a`
  Devuelve el máximo elemento en un conjunto

1. Implementar la variante que recorre la estructura buscando el máximo
2. Implementar otra variante que no tenga que hacer un recorrido

6. Stack con máximo en tiempo constante

Una pila con máximo es un tipo abstracto que posee la misma interfaz que una pila, pero agregando la operación *maxS* y modificando el tipo de *push*:

- `maxS :: Ord a => Stack a -> a`
  Devuelve el elemento máximo de la pila.
- `push :: Ord a => a -> Stack a -> Stack a`
  Dados un elemento y una pila agrega ese elemento a la pila.

Implementaremos este tipo abstracto de tal manera que la operación de máximo opere en tiempo constante. Una forma de hacer esto es mantener mediante una lista adicional el maximo elemento conocido al momento de agregar cada elemento en la pila. Indicar los invariantes de representación correspondientes.

7. Queue con dos stack

Implemente la interfaz de *Queue* pero en lugar de una lista utilice dos stack.

La estructura funciona de la siguiente manera. Llamemos a una de las stack *fs* (front stack) y a la otra *bs* (back stack). Quitaremos elementos a través de *fs* y agregaremos a través de *bs*, pero todas las operaciones deben garantizar el siguiente invariante de representación: Si *fs* se encuentra vacía, entonces la cola se encuentra vacía.

¿Qué ventaja tiene esta representación de *Queue* con respecto a la de listas?