Trabajo Práctico # 4

Estructuras de Datos, Universidad Nacional de Quilmes

11 de septiembre de 2014

Aclaraciones:

- *Los ejercicios fueron pensados para ser resueltos en el orden en que son presentados. No se saltee ejercicios sin consultar antes a un docente.*
- *Recuerde que puede aprovechar en todo momento las funciones que ha definido, tanto las de esta misma práctica como las de prácticas anteriores.*
- *Pruebe todas sus implementaciones, al menos en una consola interactiva.*
- *Es sumamente aconsejable resolver los ejercicios utilizando primordialmente los conceptos y metodologías vistos en clase, dado que los exámenes de la materia evaluarán principalmente este aspecto. Si se encuentra utilizando formas alternativas al resolver los ejercicios consulte a los docentes.*
- *No dude en manifestar observaciones y críticas sobre los ejercicios de esta práctica, que con gusto serán recibidas por los docentes.*
- *Los ejercicios del anexo pueden obviarse, pero recuerde que aportan una compresión más profunda sobre los temás que aborda esta práctica. Considere resolverlos si se encuentra practicando para una instancia de evaluación y ya resolvió todos los anteriores.*

1. Queue (cola)

Una *Queue* es un tipo abstracto de datos de naturaleza FIFO (*first in, first out*). Esto significa que los elementos que se agregan primero a la estructura, son los primeros 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.
   - `listToQueue :: [a] -> Queue a`
     Convierte una lista en una cola. Los elementos se encolan en el orden en que aparecen en la lista.
   - `queueToList :: Queue a -> [a]`
     Convierte una cola en una lista.

- `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.

2. 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:
   - `reverseS :: Stack a -> Stack a`
     Dada una pila devuelve otra con el orden de los elementos invertidos.
   - `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.

3. Set (conjunto)

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

- `emptySet :: Set a`
  Crea un conjunto vacío.
- `add :: Eq a => a -> Set a -> Set a`
  Dados un elemento y un conjunto, agrega el elemento al conjunto.
- `belongs :: Eq a => a -> Set a -> Bool`
  Dados un elemento y un conjunto indica si el elemento pertenece al conjunto.

- `union :: Eq a => Set a -> Set a -> Set a`
  Dados dos conjuntos devuelve un conjunto con todos los elementos de ambos conjuntos.
- `intersection :: Eq a => Set a -> Set a -> Set a`
  Dados dos conjuntos devuelve el conjunto de elementos que ambos conjuntos tienen en común.
- `setToList :: Set a -> [a]`
  Dado un conjunto devuelve una lista con todos los elementos del conjunto.

1. Como *usuario* del tipo abstracto *Set* implementar las siguientes funciones:
   - `sinRepetidos :: [a] -> [a]`
     Quita todos los elementos repetidos de la lista dada utilizando un conjunto como estructura auxiliar.
   - `losQuePertenecen :: Eq a => Queue a -> Set a -> Queue a`
     Dados una cola y un conjunto, devuelve una cola con todos los elementos que pertenecen al conjunto.
   - `unirTodos :: Stack (Set a) -> Set a`
     Dada una stack de conjuntos devuelve un conjunto con la union de todos los conjuntos de la stack.
   - `intersectTree :: Eq a => Tree a -> Tree a -> Set a`
     Dados dos árboles en los que sus elementos no se repiten devuelve un conjunto con los elementos que ambos árboles tienen en común. Utilizar el tipo abstracto conjunto como estructura auxiliar para calcular el conjunto de elementos en común de ambos árboles.
2. Implementar el tipo abstracto *Set* indicando los invariantes de representación correspondientes.

4. Lista con longitud en tiempo constante

Implementaremos el tipo abstracto *List* con la siguiente interfaz:

- `emptyL :: List a`
  Crea una lista vacía.
- `isEmptyL :: List a -> Bool`
  Dada una lista indica si está vacía.
- `consL :: a -> List a -> List a`
  Dados un elemento y una lista agrega el elemento a la lista.
- `tailL :: List a -> List a`
  Dada una lista devuelve la lista sin su primer elemento.
- `headL :: List a -> a`
  Dada una lista devuelve su primer elemento.
- `lenL :: List a -> Int`
  Dada una lista devuelve su cantidad de elementos.

Elegir una representación que permita implementar la operación *lenL* en tiempo constante (sin recorrer toda la estructura para contar la cantidad de elementos). Indicar los invariantes de representación correspondientes.

5. 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.

6. 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?