Trabajo Práctico # 5 - Set, Stack y Queue

Estructuras de Datos, Universidad Nacional de Quilmes

12 de mayo de 2022

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 videos publicados o clases presenciales****, 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.*

1. Cálculo de costos

Especificar el costo operacional de las siguientes funciones:

```
head' :: [a] -> a
head' (x:xs) = x

sumar :: Int -> Int
sumar x = x + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1

factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n-1)

longitud :: [a] -> Int
longitud [] = 0
longitud (x:xs) = 1 + longitud xs

factoriales :: [Int] -> [Int]
factoriales [] = []
factoriales (x:xs) = factorial x : factoriales xs

pertenece :: Eq a => a -> [a] -> Bool
pertenece n [] = False
pertenece n (x:xs) = n == x || pertenece n xs

sinRepetidos :: Eq a => [a] -> [a]
sinRepetidos [] = []
sinRepetidos (x:xs) =
    if pertenece x xs
        then sinRepetidos xs
        else x : sinRepetidos xs
```

```
-- equivalente a (++)
append :: [a] -> [a] -> [a]
append [] ys = ys
append (x:xs) ys = x : append xs ys

concatenar :: [String] -> String
concatenar [] = []
concatenar (x:xs) = x ++ concatenar xs

takeN :: Int -> [a] -> [a]
takeN 0 xs = []
takeN n [] = []
takeN n (x:xs) = x : takeN (n-1) xs

dropN :: Int -> [a] -> [a]
dropN 0 xs = xs
dropN n [] = []
dropN n (x:xs) = dropN (n-1) xs

partir :: Int -> [a] -> ([a], [a])
partir n xs = (takeN n xs, dropN n xs)

minimo :: Ord a => [a] -> a
minimo [x] = x
minimo (x:xs) = min x (minimo xs)

sacar :: Eq a => a -> [a] -> [a]
sacar n [] = []
sacar n (x:xs) =
    if n == x
        then xs
        else x : sacar n xs

ordenar :: Ord a => [a] -> [a]
ordenar [] = []
orderar xs =
    let m = minimo xs
        in m : ordenar (sacar m xs)
```

2. Set (conjunto)

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

- `emptyS :: Set a`
  Crea un conjunto vacío.
- `addS :: 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.
- `sizeS :: Eq a => Set a -> Int`
  Devuelve la cantidad de elementos distintos de un conjunto.

- `removeS :: Eq a => a -> Set a -> Set a`
  Borra un elemento del conjunto.
- `unionS :: Eq a => Set a -> Set a -> Set a`
  Dados dos conjuntos devuelve un conjunto con todos los elementos de ambos. conjuntos.
- `setToList :: Eq a => Set a -> [a]`
  Dado un conjunto devuelve una lista con todos los elementos distintos del conjunto.

1. Implementar la variante del tipo abstracto *Set* 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. Como *usuario* del tipo abstracto Set implementar las siguientes funciones:
   - `losQuePertenecen :: Eq a => [a] -> Set a -> [a]`
     Dados una lista y un conjunto, devuelve una lista con todos los elementos que pertenecen al conjunto.
   - `sinRepetidos :: Eq a => [a] -> [a]`
     Quita todos los elementos repetidos de la lista dada utilizando un conjunto como estructura auxiliar.
   - `unirTodos :: Eq a => Tree (Set a) -> Set a`
     Dado un arbol de conjuntos devuelve un conjunto con la union de todos los conjuntos del arbol.
3. Implementar la variante del tipo abstracto *Set* que posee una lista y admite repetidos. En otras palabras, al agregar no va a chequear que si el elemento ya se encuentra en la lista, pero sí debe comportarse como *Set* ante el usuario (quitando los elementos repetidos al pedirlos, por ejemplo). Contrastar la eficiencia obtenida en esta implementación con la anterior.

3. 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, el 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.
- `enqueue :: 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. Implemente el tipo abstracto *Queue* utilizando listas. Los elementos deben encolarse por el final de la lista y desencolarse por delante.

2. Implemente ahora la versión que agrega por delante y quita por el final de la lista. Compare la eficiencia entre ambas implementaciones.

3. Como *usuario* del tipo abstracto *Queue* implementar las siguientes funciones:

   - `lengthQ :: Queue a -> Int`
     Cuenta la cantidad de elementos de la cola.
   - `queueToList :: Queue a -> [a]`
     Dada una cola devuelve la lista con los mismos elementos,
     donde el orden de la lista es el de la cola.
     Nota: chequear que los elementos queden en el orden correcto.
   - `unionQ :: Queue a -> Queue a -> Queue a`
     Inserta todos los elementos de la segunda cola en la primera.

4. 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.
- `lenS :: Stack a -> Int`
  Dada la cantidad de elementos en la pila.
  Costo: constante.

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.
   - `desapilar :: Stack a -> [a]`
     Dada una pila devuelve una lista sin alterar el orden de los elementos.
   - `insertarEnPos :: Int -> a -> Stack a -> Stack a`
     Dada una posicion válida en la stack y un elemento, ubica dicho elemento en dicha posición (se desapilan elementos hasta dicha posición y se inserta en ese lugar).

2. Implementar el tipo abstracto *Stack* utilizando una lista.

5. Queue con dos listas

Implemente la interfaz de *Queue* pero en lugar de una lista utilice dos listas. Esto permitirá que todas las operaciones sean constantes (aunque alguna/s de forma amortizada).

La estructura funciona de la siguiente manera. Llamemos a una de las listas *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 que usa una sola lista?