# Trabajo Práctico # 4

Estructuras de Datos, Universidad Nacional de Quilmes

15 de diciembre de 2018

***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 temas que aborda esta práctica. Considere resolverlos si se encuentra practicando para una instancia de evaluación y ya resolvió todos los anteriores.*

## 1. 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.
- `intersectionS :: Eq a => Set a -> Set a -> Set a`
  Dados dos conjuntos devuelve un conjunto con todos los elementos en común entre ambos.
- `setToList :: Eq a => Set a -> [a]`
  Dado un conjunto devuelve una lista con todos los elementos distintos del conjunto.

1. 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.

2. 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. Costos en peor caso

Dar el orden de complejidad 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)
```

# 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.
- `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 String -> [String]`
     Dada una cola de nombres de personas, devuelve la lista de los mismos, donde el orden de la lista es el 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.

# 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.

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.
   - `treeToStack :: Tree a -> Stack a`
     Dado un árbol devuelve una pila con los elementos apilados inorder.
   - `balanceado :: String -> Bool` (desafío)
     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. **Pista**: recorra una stack pasada por parámetro a una subtarea que devuelve un booleano, que indifica si los parentesis están balanceados.
2. Implementar el tipo abstracto *Stack* utilizando listas.

# 5. Queue con longitud constante

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

- `lenQ :: Queue a -> Int -- O(1)`
  Devuelve la cantidad de elementos

# 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?

# 8. Conjunto con Máximo (hacer anexo)

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

- `maximoC :: Ord a => Set 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.

# Anexo 1

# 9. Maybe

1. Defina las siguientes operaciones (notar que son parciales si no devuelven Maybe):

   *a*) `headM :: [a] -> Maybe a`
   Versión de head que es total

*b*) `lastM :: [a] -> Maybe a`
Dada una lista, devuelve su último elemento.

*c*) `maximumM :: Ord a => [a] -> Maybe a`
Dada una lista de elementos devuelve el máximo.

*d*) `initM :: [a] -> Maybe [a]`
Dada una lista quita su último elemento.

*e*) `get :: Int -> [a] -> Maybe a`
Dado un índice devuelve el elemento de esa posición.

*f*) `indiceDe :: Eq a => a -> [a] -> Maybe Int`
Dado un elemento y una lista devuelve la posición de la lista en la que se encuentra dicho elemento.

*g*) `lookupM :: Eq k => [(k,v)] -> k -> Maybe v`
Dada una lista de pares (clave, valor) y una clave devuelve el valor asociado a la clave.

*h*) `fromJusts :: [Maybe a] -> [a]`
Devuelve los valores de los Maybe que no sean Nothing.

*i*) `minT :: Ord a => Tree a -> Maybe a`
Dado un árbol devuelve su elemento mínimo.

2. Indicar los ordenes de complejidad en peor caso de cada función implementada.