Universidad Nacional Autónoma de México
Facultad de Ciencias

Estructuras Discretas

Práctica 5

Javier Enríquez Mendoza **Mauricio E. Hernández Olvera**

26 de Octubre de 2018
Fecha de entrega: 9 de Noviembre de 2018

Instrucciones generales

La práctica debe resolverse en los archivos `Snoc.hs` y `Tree.hs` conservando las firmas de las funciones idénticas a las que se muestran en cada ejercicio. Cada función y definición debe estar debidamente comentada con la especificación de ésta.

Se tomará en cuenta la legibilidad y el estilo del código.

Estructuras de Datos Recursivas.

1 Listas Snoc

Utilizando la siguiente definición para**Listas Snoc** en Haskell, resolver los siguientes ejercicios en el archivo `Snoc.hs`.

```
-- Tipo de dato Algebraico para construir Lisas Snoc.

data SnocList a = Mt
                | Snoc (SnocList a) a
```

Ejercicio 1.1 (1 pt.) Definir la función `addSnoc` que recibe una lista Snoc, un elemento e y agrega a e como último elemento de la lista.

```
addSnoc :: SnocList a -> a -> SnocList a
```

```
> addSnoc Mt 1
(Snoc Mt 1)
> addSnoc (Snoc (Snoc Mt 1) 2) 3
(Snoc (Snoc (Snoc Mt 1) 2) 3)
```

Ejercicio 1.2 (0.5 pts.) Definir la función `ultimo` que regresa el último elemento de una lista Snoc.

```
ultimo :: SnocList a -> a
```

```
> ultimo (Snoc (Snoc (Snoc Mt 1) 2) 3)
3
> ultimo (Snoc (Snoc (Snoc (Snoc Mt ‘‘Hola’’)‘‘Mundo’’)‘‘Hello’’)‘‘World’’)
‘‘World’’
```

Ejercicio 1.3 (0.5 pts.) Definir la función `resto` que regresa todos los elementos a excepción del último de una lista Snoc.

```
resto :: SnocList a -> SnocList a
```

```
> resto (Snoc (Snoc (Snoc Mt 1) 2) 3)
(Snoc (Snoc Mt 1) 2)
> resto (Snoc (Snoc (Snoc (Snoc Mt ‘‘Hola’’)‘‘Mundo’’)‘‘Hello’’)‘‘World’’)
(Snoc (Snoc (Snoc Mt ‘‘Hola’’)‘‘Mundo’’)‘‘Hello’’)
```

Ejercicio 1.4 (1 pt.) Definir la función `cabeza` que regresa el primer elemento de una lista Snoc.

```
cabeza :: SnocList a -> a
```

```
> cabeza (Snoc (Snoc (Snoc Mt 1) 2) 3)
1
> cabeza (Snoc (Snoc (Snoc (Snoc Mt ‘‘Hola’’)‘‘Mundo’’)‘‘Hello’’)‘‘World’’)
‘‘Hola’’
```

Ejercicio 1.5 (1 pt.) Definir la función `cola` que regresa todos los elementos a excepción del primero de una lista Snoc.

```
cola :: SnocList a -> SnocList a
```

```
> cola (Snoc (Snoc (Snoc Mt 1) 2) 3)
(Snoc (Snoc Mt 2) 3)
> cola (Snoc (Snoc (Snoc (Snoc Mt ‘‘Hola’’)‘‘Mundo’’)‘‘Hello’’)‘‘World’’)
(Snoc (Snoc (Snoc Mt ‘‘Mundo’’)‘‘Hello’’)‘‘World’’)
```

Ejercicio 1.6 (1 pt.) Definir la función `longitud` que regresa la cantidad de elementos de una lista Snoc.

```
longitud :: SnocList a -> Int
```

```
> longitud (Snoc (Snoc (Snoc Mt 1) 2) 3)
3
> longitud (Snoc (Snoc (Snoc (Snoc Mt ‘‘Hola’’)‘‘Mundo’’)‘‘Hello’’)‘‘World’’)
4
```

2 Árboles Binarios Ordenados

Utilizando la siguiente definición para**Árboles Binarios** en Haskell, resolver los siguientes ejercicios en el archivo `Tree.hs`.

```
-- Tipo de dato Algebraico  para construir Árboles Binarios.

data BinaryTree a = Void
                  | Node (BinaryTree a) a (BinaryTree a)
```

Ejercicio 2.1 (1 pt) Definir la función `addTree` que agrega un elemento a un árbol binario ordenado, preservando el orden.

```
addTree :: (Ord a) => BinaryTree a -> a -> BinaryTree a
```

```
> addTree Void 1
(Node Void 1 Void)
> addTree (Node (Node Void 1 Void) 2 (Node Void 3 Void)) 4
Node (Node Void 1 Void) 2 (Node Void 3 (Node Void 4 Void))
```

Ejercicio 2.2 (0.5 pts.) Definir la función `inorder` que recibe un `BinaryTree` y regresa una lista con los elementos del árbol recorriéndolo inorder.

```
inorder :: BinaryTree a -> [a]
```

```
> inorder (Node (Node (Node Void 1 Void) 2 (Node Void 3 Void)) 4 (Node (Node
Void 5 Void) 6 (Node Void 7 Void)))
[1,2,3,4,5,6,7]
> inorder (Node (Node Void 1 Void) 2 (Node Void 3 Void))
[1,2,3]
```

Ejercicio 2.3 (0.5 pts.) Definir la función `preorder` que recibe un `BinaryTree` y regresa una lista con los elementos del árbol recorriéndolo preorder.

```
preorder :: BinaryTree a -> [a]
```

```
> preorder (Node (Node (Node Void 1 Void) 2 (Node Void 3 Void)) 4 (Node (Node
Void 5 Void) 6 (Node Void 7 Void)))
[4,2,1,3,6,5,7]
> preorder (Node (Node Void 1 Void) 2 (Node Void 3 Void))
[2,1,3]
```

Ejercicio 2.4 (0.5 pts.) Definir la función `postorder` que recibe un `BinaryTree` y regresa una lista con los elementos del árbol recorriéndolo postorder.

```
postorder :: BinaryTree a -> [a]
```

```
> postorder (Node (Node (Node Void 1 Void) 2 (Node Void 3 Void)) 4 (Node (Node
Void 5 Void) 6 (Node Void 7 Void)))
[1,3,2,5,7,6,4]
> postorder (Node (Node Void 1 Void) 2 (Node Void 3 Void))
[1,3,2]
```

Ejercicio 2.5 (0.75 pts.) Definir la función `maximo` que regresa el elemento más grande de un `BinaryTree` ordenado.

```
maximo :: (Ord a) => BinaryTree a -> a
```

```
>maximo (Node Void 1 Void)
1
> maximo (Node (Node Void 1 Void) 2 (Node Void 3 Void))
3
```

Ejercicio 2.6 (0.75 pts.) Definir la función `minimo` que regresa el elemento más pequeño de un `BinaryTree` ordenado.

```
minimo :: (Ord a) => BinaryTree a -> a
```

```
> minimo (Node Void 1 Void)
1
> minimo (Node (Node Void 1 Void) 2 (Node Void 3 Void))
1
```

Ejercicio 2.7 (1 pt.) Definir la función `busca` que recibe un elemento y un `BinaryTree` ordenado y nos dice si el elemento pertenece o no al `BinaryTree` utilizando el algoritmo de búsqueda visto en clase.

```
busca :: (Ord a) => a -> BinaryTree a -> Bool
```

```
> busca 6 (Node Void 1 Void)
False
> busca 3 (Node (Node Void 1 Void) 2 (Node Void 3 Void))
True
```

Extras

Ejercicio 3.1 (0.5 pts.) Definir la función `mapSnoc` que recibe una función, una Lista Snoc y aplica la función a cada elemento de la lista.

```
mapSnoc :: (a -> b) -> SnocList a -> SnocList b
```

```
> mapSnoc succ (Snoc (Snoc (Snoc (Mt 1) 2) 3)
(Snoc (Snoc (Snoc Mt 2) 3) 4)
> mapSnoc (\x -> x ++"!!") (Snoc (Snoc Mt"Hello")"World")
(Snoc (Snoc Mt"Hello!!")"World!!")
```

Ejercicio 3.2 (0.5 pts.) Definir la función `mapTree` que recibe una función, un `BinaryTree` y aplica la función a cada elemento del árbol.

```
mapTree :: (a -> b) -> BinaryTree a -> BinaryTree b
```

```
> mapTree (\x -> [x]) (Node Void 1 Void)
Node Void [1] Void
> mapTree even (Node (Node Void 1 Void) 2 (Node Void 3 Void))
Node (Node Void False Void) True (Node Void False Void)
```

Pruebas Unitarias

En el archivo `testP05.hs` se agregaron una serie de pruebas que verifican el correcto funcionamiento de cada una de las funciones de esta práctica.

Para poder correr estas pruebas, se tiene que copiar el archivo en el mismo directorio en el que se encuentran `Snoc.hs` `yTree.hs`, y desde la terminal ejecutar los siguientes comandos para compilar y ejecutar las pruebas respectivamente.

```
> ghc testP05.hs
> ./testP05
```

Se mostrará en la consola los resultados de cada una de las pruebas.

Se recomienda no modificar el archivo `testP05.hs`.

Entrega

- La entrega se realiza mediante correo electrónico a la dirección de los ayudantes de laboratorio (javierem_94@ciencias.unam.mx y mauriciohdez08@ciencias.unam.mx).
- Es **necesario** que el correo se envíe a ambos ayudantes.
- La practica deberá ser entregada en equipos de máximo 3 personas.
- Se debe entregar un directorio `numeroCuenta_P05`, dónde `numeroCuenta` es el número de cuenta de un integrante del equipo. Dentro del directorio se debe incluir:
  - Un archivo readme.txt con los nombres y números de cuenta de los alumnos, comentarios, opiniones, críticas o ideas sobre la práctica.
  - Los archivos requeridos en la práctica. Debe enviarse código lo más limpio posible.
- Los archivos requeridos para esta práctica son:`Snoc.hs` y `Tree.hs`.

- El directorio se deberá comprimir en un archivo con nombre `numeroCuenta_P05.tar.gz`, dónde `numeroCuenta` es el número de cuenta de un integrante del equipo.
- Únicamente **un integrante** del equipo deberá enviar el correo con la práctica.
- El asunto del correo debe ser [ED-20191-P05].
- Se recibirá la práctica hasta las 23:59:59 horas del día fijado como fecha de entrega, cualquier práctica recibida después no será tomada en cuenta.
- **Cualquier práctica total o parcialmente plagiada, será calificada automáticamente con cero y no se aceptarán más prácticas durante el semestre.**