Programación Declarativa 2020-2
Facultad de Ciencias UNAM
Tarea 5: I'm Groot

Favio E. Miranda Perea Javier Enríquez Mendoza

Fecha de entrega: 15 de mayo de 2020

Árboles de Braun

En esta sección utilizaremos una clase especial de árboles binarios balanceados, para lograr una implementación de arreglos con complejidad logarítmica. El concepto particular de balanceo que utilizaremos es el siguiente:

- El árbol vacío está balanceado
- Un árbol no vacío está balanceado si y sólo si sus subárboles están balanceados y sucede alguna de las siguientes condiciones:
  - Los dos subárboles tienen el mismo número de nodos.
  - El subárbol izquierdo tiene un nodo más que el subárbol derecho.

  Estas dos condiciones son el invariante del balanceo.

A este tipo de árboles se les conoce como árboles de Braun. Para la implementación se utilizará el siguiente tipo de dato algebraico que define árboles binarios.

```
data BTree a = BVoid | BNode a (BTree a) (BTree a)
```

Observe que este es el tipo de árboles binarios ordinarios y por lo tanto no es posible capturar el invariante deseado estáticamente, es decir, un árbol de tipo `BTree a` no está necesariamente balanceado.

Para definir una implementación de arreglos funcionales se usa la clase `BArray` definida en el archivo barray.hs. Estos arreglos son bidireccionales, en el sentido de que es posible agregar elementos al inicio o al final de un arreglo dado.

1. Define una función `isBal` que verifica si un árbol dado está balanceado.
2. Define una instancia de la clase `BArray` para el tipo `BTree a` siguiendo las siguientes sugerencias y teniendo especial cuidando que cada arreglo `a` devuelto como resultado de alguna operación cumpla con la condición de balanceo, es decir, cumpla que `isBal a = True` para lograr complejidad logarítmica.
   - `size`: utilice el invariante para evitar calcular el tamaño de los dos subárboles recursivamente.
   - `(!)`: los arreglos deben implementarse de manera que el elemento 0 es la raiz del árbol; los elementos impares deben guardarse en el subárbol izquierdo y los elementos pares en el subárbol derecho.
   - `update`: es análoga a la definición de `(!)`
   - `lowExt`: se debe cumplir la siguiente especificación: si `t = lowExt x s` entonces

     $$\texttt{size t} = \texttt{size s} + 1 \land \texttt{t!0} = \texttt{x} \land \forall\ 0 \leq \texttt{j} < \texttt{size s}\ (\texttt{t!(j} + 1) = \texttt{s!j})$$

   - `lowRem`: se debe cumplir la siguiente especificación: si `t = lowRem s` con `s` no vacío, entonces

     $$\texttt{size t} = \texttt{size s} - 1 \land \forall\ 0 \leq \texttt{j} < \texttt{size s}\ (\texttt{t!j} = \texttt{s!(j} + 1))$$

   - `highExt`: se debe cumplir la siguiente especificación: si `t = highExt x s` entonces

     $$\texttt{size t} = \texttt{size s} + 1 \land \texttt{t!(size s)} = \texttt{x} \land \forall\ 0 \leq \texttt{j} < \texttt{size s}\ (\texttt{t!j} = \texttt{s!j})$$

   - `lowRem`: se debe cumplir la siguiente especificación: si `t = highRem s` con `s` no vacío, entonces

     $$\texttt{size t} = \texttt{size s} - 1 \land \forall\ 0 \leq \texttt{j} < \texttt{size s}\ (\texttt{t!j} = \texttt{s!j})$$

   - `copy`: utilice el invariante, observando que, como los elementos son todos el mismo no es necesario hacer dos llamadas recursivas
   - `fromList`: debe cumplirse que

     $$\texttt{length xs} = \texttt{size(fromList xs)} \land \forall\ 0 \leq \texttt{j} < \texttt{length xs}\ (\texttt{xs!!j} = \texttt{(fromList xs)!j})$$

   - `toList`: debe cumplirse que

     $$\texttt{size t} = \texttt{length(toList t)} \land \forall\ 0 \leq \texttt{j} < \texttt{size t}\ (\texttt{t!j} = \texttt{(toList t)!!j})$$

Árboles Cartesianos

Un árbol cartesiano es un árbol binario construido a partir de una lista preservando el orden en el que aparecen los elementos en ésta, es decir que para cada nodo todos los elementos de su subárbol izquierdo

aparecían antes que él en la lista y de la misma forma, todos los elementos del subrárbol derecho aparecían después en la lista. De tal forma que al hacer un recorrido *in-order* del árbol se obtiene la lista original.

Ademas un árbol cartesiano tiene la propiedad de ser un minheap, es decir que para todos los subárboles del árbol se cumple que el valor de la raíz es más pequeño que el valor de sus hijo, también podría ser un maxheap dependiendo de la implementación, nosotros haremos un minheap.

El tipo de dato algebraico para definir árboles cartesianos el es siguiente:

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

Parece conocido ¿no?

1. Define la función `cart` que construye un árbol cartesiano a partir de una lista siguiendo el algoritmo visto anteriormente.

   ```
   cart :: (Ord a) => [a] -> CartT a
   ```

2. Define la función `inorder` que regresa una lista con los elementos de un árbol haciendo un recorrido *in-order*.

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

   Verifica que se cumpla `inorder (cart xs)= xs`, de no ser así corrige tus implementaciones.

Árboles Generales

Un árbol general es un árbol con una estructura de múltiples ramas, es decir un árbol en el que el número de hijos de cada nodo no está acotado. Considera el siguiente tipo de dato algebraico:

```
data Gtree a = Node a [Gtree a]
```

Para representar el múltiple ramaje de los árboles se define un nodo con una lista de hijos, podríamos incluso crear nodos con una cantidad infinita de hijos.

1. Define las funciones `size` y `depth` que calculan el tamaño (número de elementos) y la profundidad (número de niveles) de un árbol general.

   ```
   size :: Gtree a -> Int

   depth :: Gtree a -> Int
   ```

2. Define la función `tran` que tranforma un árbol general en un árbol binario, utilizando la definición de árbol binario que se vio en las secciones anteriores (**No** es necesario que este ordenado).

   ```
   tran :: Gtree a -> BT a
   ```

3. Define las funciones de orden superior siguientes en donde el comportamiento de éstas está basado en las equivalentes para listas.

```
mapg :: (a -> b) -> Gtree a -> Gtree b

foldg :: (a -> [b] -> b) -> Gtree a -> b
```

   ¿Cuál es el patrón que encapsula la función `foldg`?

4. Define la función `searchg` que verifica si un elemento pertenece al árbol general.

```
searchg :: (Eq a) => a -> Gtree a -> Bool
```

Extra

Este ejercicio tiene un valor de hasta 2 puntos extras sobre la calificación de esta tarea.

1. Diseña un tipo `Time` que sea capaz de representar el tiempo en formato de 24 horas, por ejemplo 1518hrs, o en formato de 12 horas, por ejemplo 6:34PM. A partir de su definición implemente las siguientes funciones:

   (a) `to24` que transforma una hora de 24 a 12 horas.

   (b) `to12` que transforma una hora de 12 a 24 horas.

   (c) `eqTime` que decide si dos horas son iguales, sin importar su formato. Por ejemplo, 19:05 y 7:05PM son iguales.

   (d) `showTime` que muestra en pantalla las horas de acuerdo al siguiente formato: para 24 horas "xxxxHRS" y para 12 horas "hh:mmAM" o "hh:mmPM" con excepción de las horas 12:00AM que debe mostrarse como "Medianoche" y 12:00PM que debe mostrarse como "Mediodia".

   Hint : para este punto es más facil definir la función

```
showsPrec ::  Int -> Time -> String -> String
```

   (e) Instancia el tipo `Time` a las clases `Eq` y `Show` utilizando las funciones de los incisos anteriores

   (f) Define una función `ltTime` que decida si una hora es menor que otra, independientemente del formato.

   (g) Instancie el tipo `Time` a la clase `Ord`.

Entrega

- La tarea puede entregarse de forma individual o en parejas.
- Se tomará en cuenta tanto el estilo de programación como la complejidad de las soluciones para la calificación de la sección práctica.

- La tarea se entrega a través de Slack:
  - Todos los archivos deben tener al principio como comentario los nombres de los alumnos.
  - Si la tarea se realizo en parejas crear un canal privado con el ayudante y los miembros del equipo para la entrega.
  - Si se realizo de forma individual enviarla en un mensaje directo al ayudante.
  - **No** deben comprimir los archivos, en caso de tener mas de uno enviarlos por separado.