Trabajo Práctico # 5

Estructuras de Datos, Universidad Nacional de Quilmes

4 de octubre 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. Maybe

1. Defina las siguientes operaciones parciales y las precondiciones necesarias para poder utilizarlas.

   a) `init :: [a] -> [a]`
   Dada una lista quita su último elemento.

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

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

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

   e) `maximum :: Ord a => [a] -> a`
   Dada una lista de elementos devuelve el máximo.

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

2. Defina las versiones totales de las operaciones parciales definidas en el punto anterior, empleando el tipo de dato `Maybe`.

2. Map (diccionario)

Ejercicio 1

La interfaz del tipo abstracto Map es la siguiente:

- `emptyM  :: Map k v`
- `assocM  :: Eq k =>  Map k v -> k -> v -> Map k v`
- `lookupM :: Eq k =>  Map k v -> k -> Maybe v`
- `deleteM :: Eq k =>  Map k v -> k -> Map k v`
- `domM    :: Map k v -> Set k`

Implemente las distintas variantes del tipo Map vistas en clase:

1. Como una lista de pares-clave valor con claves repetidas
2. Como una lista de pares-clave valor sin claves repetidas
3. Como un árbol binario de búsqueda (BST)

Ejercicio 2

Implementar como usuario del tipo abstracto Map las siguientes funciones:

1. `indexar :: [a] -> Map Int a`
   Dada una lista de elementos construye un Map que relaciona cada elemento con su posición en la lista.
2. `pedirTelefonos :: [String] -> Map String Int -> [Maybe Int]`
   Dada una lista de nombres de personas y un Map que relaciona nombres con números de teléfonos, devuelve una lista con los números de las personas de la lista o *Nothing* en caso de que no posea número.
3. `ocurrencias :: String -> Map Char Int`
   Dado un string cuenta las ocurrencias de cada caracter utilizando un Map.

Ejercicio 3

El tipo abstracto Celda posee la siguiente interfaz:

- `celdaVacia :: Celda`
  Crea una celda con cero bolitas de cada color
- `poner :: Color -> Celda -> Celda`
  Agrega una bolita de ese color a la celda
- `sacar :: Color -> Celda -> Celda`
  Saca una bolita de ese color, parcial cuando no hay bolitas de ese color
- `nroBolitas :: Color -> Celda -> Int`
  Devuelve la cantidad de bolitas de ese color
- `hayBolitas :: Color -> Celda -> Bool`
  Indica si hay bolitas de ese color

Defina este tipo abstracto utilizando la siguiente representación:

```
data Color = Azul | Negro | Rojo | Verde
data Celda = MkCelda (Map Color Int)

{- Inv. Rep.:
- Existe una clave para cada color existente
- El valor asociado a un color es un número positivo
-}
```

Luego como usuario de este tipo abstracto implemente las siguientes operaciones:

- `nroBolitasMayorA :: Color -> Int -> Celda -> Bool`
  Devuelve True si hay mas de "n"bolitas de ese color
- `ponerN :: Int -> Color -> Celda -> Celda`
  Agrega "n"bolitas de ese color a la celda
- `hayBolitasDeCadaColor :: Celda -> Bool`
  Indica si existe al menos una bolita de cada color posible

Ejercicio 4

Definir el tipo abstracto *MiniTablero* con la siguiente interfaz:

- `crearFila :: Int -> MiniTablero`
  Crea una fila de celdas vacias de tamaño "n", apuntando a la primera celda
- `mover :: Dir -> MiniTablero -> MiniTablero`
  Dada una dirección d, mueve el cabezal hacia d. Esta función es parcial cuando no existe una celda en esa dirección.
- `puedeMover :: Dir -> MiniTablero -> Bool`
  Dada una dirección indica si existe una celda en esa dirección.
- `poner :: Color -> MiniTablero -> MiniTablero`
  Poner una bolita en la celda actual del color indicado.
- `sacar :: Color -> MiniTablero -> MiniTablero`
  Saca una bolita de la celda actual del color indicado.
- `nroBolitas :: Color -> MiniTablero -> Int`
  Devuelve el número de bolitas de un color en la celda actual.
- `hayBolitas :: Color -> MiniTablero -> Bool`
  Indica si hay bolitas de un color en la celda actual.

Utilice la siguiente representación para implementar este tipo abstracto (indicando los invariantes de representación que correspondan):

```
data MiniTablero = MkT (Map Coord Celda) Coord
type Coord = Int
data Dir = Este | Oeste
```

Luego como usuario de este tipo abstracto implementar las siguientes funciones:

- `irAlExtremo :: Dir -> MiniTablero -> MiniTablero`
  Dada una dirección d, mueve el cabezal hacia el borde d.
- `llenarHasta :: Int -> MiniTablero -> MiniTablero`
  Dados un número n y una fila, pone n bolitas de cada color en todas las celdas de la fila.
- `contarCeldas :: MiniTablero -> Int`
  Dado un tablero indica cuántas celdas posee.
- `noHayBolitas :: MiniTablero -> Bool`
  Dado un tablero indica si todas sus celdas se encuentran vacías.
- `cantidadesDeBolitas :: MiniTablero -> Map Color Int`
  Dado un tablero arma un Map en donde indica para cada color cuántas bolitas hay en total en el tablero.

Ejercicio 5

Modifique los tipos abstractos Celda y Minitablero para que sus operaciones parciales pasen a ser totales, utilizando el tipo *Maybe* en donde corresponda.

3. MultiSet (MultiConjunto)

Un *MultiSet* (multiconjunto) es un tipo abstracto de datos similar a un *Set* (conjunto). A diferencia del último, cada elemento puede aparecer más de una vez, y es posible saber la cantidad de ocurrencias para un determinado elemento. Su interfaz es la siguiente:

- `emptyMSet :: MultiSet a`
  Crea un multiconjunto vacío.
- `addMS :: Ord a => a -> MultiSet a -> MultiSet a`
  Dados un elemento y un multiconjunto, agrega una ocurrencia de ese elemento al multiconjunto.
- `ocurrencesMS :: Ord a => a -> MultiSet a -> Int`
  Dados un elemento y un multiconjunto indica la cantidad de apariciones de ese elemento en el multiconjunto.
- `unionMS :: Ord a => MultiSet a -> MultiSet a -> MultiSet a`
  Dados dos multiconjuntos devuelve un multiconjunto con todos los elementos de ambos multiconjuntos.
- `intersectionMS :: Ord a => MultiSet a -> MultiSet a -> MultiSet a`
  Dados dos multiconjuntos devuelve el multiconjunto de elementos que ambos multiconjuntos tienen en común.
- `multiSetToList :: MultiSet a -> [(Int,a)]`
  Dado un multiconjunto devuelve una lista con todos los elementos del conjunto y su cantidad de ocurrencias.

1. Implementar el tipo abstracto *MultiSet* utilizando como representación un Map.
2. Reimplementar como usuario de *MultiSet* la función `ocurrencias` de ejercicios anteriores, que dado un string cuenta la cantidad de ocurrencias de cada caracter en el string. En este caso el resultado será un multiconjunto de caracteres.

Anexo con ejercicios adicionales

Ejercicio 6

Implemente el tipo abstracto *Quizas* (representación abstracta de Maybe), que consta de la siguiente interfaz:

- `nothing :: Quizas a`
- `just :: a -> Quizas a`
- `fromJust :: Quizas a -> a`
- `isNothing :: Quizas a -> Bool`

Ejercicio 7

Se desea desarrollar el sistema de administración de un comercio. En el mismo se quieren registrar empleados nuevos (con su nombre completo y DNI), los artículos (dados por un código, descripción y precio) y las ventas (dadas por un número identificador único de venta, el DNI del vendedor y el código de artículo vendido). También se desea conocer el artículo más vendido, el vendedor que más articulos vendio y el que vendio por mayor cantidad de plata (sumando los costos de todos sus artículos vendidos).

Se pide lo siguiente.

a) Definir las funciones que debería tener el tipo *Comercio* para resolver el problema, indicando precondiciones y propósitos.

b) Dar una estructura y un invariante de representación para el tipo Comercio. Programar y probar las funciones definidas en el ítem anterior.

c) Suponga que se pide, en peor caso, $O(\log n)$ para registrar una venta (donde n es la cantidad de artículos vendidos hasta el momento). ¿Debería cambiar sustancialmente la representación dada en el ítem anterior? (Si la respuesta es sí, consulte a un docente).