Trabajo Práctico # 5

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

1. 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 -> [k]`

Implementar como usuario del tipo abstracto Map las siguientes funciones:

1. `buscarClaves :: Eq k => [k] -> Map k v -> [Maybe v]`
   Busca todas las claves dadas en la lista.
2. `estanTodas :: Eq k => [k] -> Map k v -> Bool`
   Indica si en el map se encuenrtan todas las claves dadas.
3. `actualizarClaves :: Eq k => [(k, v)] -> Map k v -> Map k v`
   Actualiza todas las claves dadas por las primeras componentes con los valores de las segundas componentes de cada par.
4. `unirDoms :: Eq k => [Map k v] -> [k]`
   Une los doms de una lista de maps. En el resultado no hay claves repetidas.

5. `mapSuccM :: Eq k => [k] -> Map k Int -> Map k Int`
   Dada una lista de claves de tipo k y un mapa que va de k a int, le suma uno a cada número asociado con dichas claves.

6. `agregarMap:: Eq k => Map k v -> Map k v -> Map k v`
   Dado dos maps se agregan las claves y valores del primer map en el segundo. Si una clave del primero existe en el segundo, es reemplazada por la del primero.

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

Ejercicio 2

Implemente las distintas variantes del tipo Map vistas en clase:

1. Como una lista de pares-clave valor sin claves repetidas
2. Como una lista de pares-clave valor con claves repetidas
3. Como dos listas, una de claves y otra de valores, donde en la posición i de la lista de claves esta la clave del valor i de la lista de valores.
4. Una lista con los k de los pares ordenados de menor a mayor

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

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

Implemente estas otras funciones como usuario de Map:

- `indexar :: [a] -> Map Int a`
  Dada una lista de elementos construye un Map que relaciona cada elemento con su posición en la lista.
- `ocurrencias :: String -> Map Char Int`
  Dado un string cuenta las ocurrencias de cada caracter utilizando un Map.

Indicar los ordenes de complejidad en peor caso de cada función de la interfaz y del usuario.

2. MultiSet (MultiConjunto)

Ejercicio 5

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:

- `emptyMS :: 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. Indicar los ordenes de complejidad en peor caso de cada función de la interfaz.
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.

3. Árboles de Búsqueda

Ejercicio 6

Implementar las siguientes funciones suponiendo que reciben un árbol binario que cumple los invariantes de BST. Todas deben ser implementadas en O(log n).

1. `perteneceBST :: Ord a => a -> Tree a -> Bool`
   Dado un BST dice si el elemento pertenece o no al árbol.
2. `insertBST :: Ord a => a -> Tree a -> Tree a`
   Dado un BST inserta un elemento en el árbol.
3. `minBST :: Ord a => Tree a -> a`
   Dado un BST devuelve el mínimo elemento.
4. `deleteMinBST :: Ord a => Tree a -> Tree a`
   Dado un BST devuelve el árbol sin el mínimo elemento
5. `maxBST :: Ord a => Tree a -> a`
   Dado un BST devuelve el máximo elemento.
6. `deleteMaxBST :: Ord a => Tree a -> Tree a`
   Dado un BST devuelve el árbol sin el máximo elemento
7. `deleteBST :: Ord a => a -> Tree a -> Tree a`
   Dado un BST borra un elemento en el árbol.
8. `splitMinBST :: Ord a => Tree a -> (a, Tree a)`
   Dado un BST devuelve un par con el mínimo elemento y el árbol sin el mismo.
9. `splitMaxBST :: Ord a => Tree a -> (a, Tree a)`
   Dado un BST devuelve un par con el máximo elemento y el árbol sin el mismo.
10. `esBST :: Ord a => Tree a -> Bool`
    Dado un árbol cualquiera indica si cumple con las condiciones de `BST`. Nota: está función no se suele usar en ninguna implementación porque es costosa.
11. `elMaximoMenorA :: Ord a => a -> Tree a -> Maybe a`
    Dado un BST y un elemento, devuelve el máximo elemento que sea menor o igual al elemento dado.
12. `elMinimoMayorA :: Ord a => a -> Tree a -> Maybe a`
    Dado un BST y un elemento, devuelve el mínimo elemento que sea mayor o igual al elemento dado.

Ejercicio 7

Implementar usando árboles binarios con invariantes de BST:

1. `Map`
2. `Set`

Ejercicio 8

Revise las secciones anteriores en las que se definieron funciones usando `Map` o `Set` y vuelva a calcular el costo de las mismas.

4. Colas de Prioridad (Heaps)

Ejercicio 9

Implementar la interfaz de Heap utilizando un BST. Indicar los costos de cada operación. La interfaz de Heap es la siguiente:

- `emptyH :: Heap a`
- `isEmptyH :: Heap a -> Bool`
- `insertH :: Ord a => a -> Heap a -> Heap a`
- `findMin :: Ord a => Heap a -> a -- Parcial en emptyH`
- `deleteMin :: Ord a => Heap a -> Heap a -- Parcial en emptyH`
- `splitMin :: Ord a => Heap a -> (a,Heap a) -- Parcial en emptyH`

Ejercicio 10

Implementar la función `heapSort :: Ord a => [a] -> [a]`, que dada una lista la ordena de menor a mayor utilizando una Heap como estructura auxiliar.

Ejercicio 11

Implementar la interfaz de Heap utilizando un árbol binario con invariantes de BinaryHeap.

Anexo con ejercicios adicionales

Ejercicio 12

(Desafío) Implementar `Red Black Trees`, una forma de árboles BST balanceados.

Ejercicio 13

(Desafío) Resolver el problema de `Subset Sum`, que dado un conjunto de números, indica si existe un subconjunto cuyos elementos suman un determinado número.

En Haskell puede ser una función como

```
subsetSum :: [Int] -> Int -> Bool
```

¿Cuál es el costo de esta función?