Examen-Parcial-2020-Resuelto.pdf

pablofa02

Estructura de Datos

2º Doble Grado en Ingeniería Informática y Matemáticas

Escuela Técnica Superior de Ingeniería Informática
Universidad de Málaga

Práctica Evaluable - Noviembre 2020

Estructuras de Datos

Computadores A / Informática D / Matemáticas + Informática

Definición informal del TAD

El TAD SetMultiMap es una colección de pares `(K,Vs)` donde:

- `K` es la *clave* y
- `Vs` un *conjunto no vacío* de valores de tipo `V` asociado a la clave `K`.

En un SetMultiMap las claves son **únicas** (no puede haber dos pares con la misma clave).

Ejemplo. El siguiente es un SetMultiMap que asocia a cada clave (`String`) un conjunto de enteros (`Int`) (representamos los pares `(K,Vs)` por la notación gráfica `K -> { V1, ..., Vn }`):

```
"alfredo" -> { 9 }
"juan" -> { 8, 1, 0 }
"maria" -> { 4, -6, 8 }
```

Un SetMultiMap `xs` soporta las siguientes operaciones:

- `empty`: devuelve un SetMultiMap vacío.
- `isEmpty xs`: devuelve `True` si `xs` está vacío, `False` en caso contrario.
- `size xs`: devuelve el número de pares que contiene `xs`.
- `isDefinedAt k xs`: que devuelve `True` si la clave `k` aparece en `xs` y `False` en caso contrario.
- `insert xs k v`: si la clave `k` no está presente en `xs`, añade el par `(k, { v })`; si la clave `k` está presente en `xs`, añade `v` al conjunto de valores asociado a `k`.
- `valuesOf k xs`: devuelve, *si es posible*, el conjunto de valores asociado a `k` en `xs`.
- `deleteKey k xs`: si la clave `k` aparece en `xs` la elimina; de lo contrario no hace nada.
- `deleteKeyValue k v xs`: si la clave `k` aparece en `xs` y tiene asociado el valor `v`, elimina `v`; de lo contrario no hace nada.
- `filterValues p xs`: devuelve el SetMultimap que se obtiene al filtrar los valores asociados a cada clave con el predicado `p`.

Representación física

El TAD SetMultiMap se representará en Haskell por el siguiente tipo algebraico:

```haskell
data SetMultiMap a b = Empty
                     | Node a (S.Set b) (SetMultiMap a b)
                     deriving Eq
```

Además, se debe satisfacer el siguiente **invariante de representación**:

- Los nodos están ordenados por clave
- No hay claves repetidas
- No hay claves que tengan asociado un conjunto vacío

Ejercicios

El fichero `SetMultiMap.hs` contiene definiciones incompletas de la implementación y ejemplos de uso de cada función en los que se muestra la salida esperada para un SetMultimap `m1`.

Ejercicio 1. Completa en `SetMultiMap.hs` las definiciones de las funciones `empty`, `isEmpty`, `size`, `isDefinedAt`, `insert`, `valuesOf`, `deleteKey`, `deleteKeyValue`, y `filterValues`. Completa además el comentario indicando la clase de complejidad a la que pertenece cada función.

Ejercicio 2. Completa en `SetMultiMap.hs` los axiomas que definen la semántica de `deleteKeyValue`.

Ejercicio 3. Además de las operaciones anteriores, el TAD SetMultiMap tiene definido el siguiente plegado:

`fold f z xs`: pliega un SetMultiMap usando `z` para el caso base y `f` como función de plegado.

La función de plegado `f` recibe como parámetros la clave `k` y **cada uno** de los valores `v` asociados a `k` por separado; por ejemplo, para el SetMultiMap `m1` se `f` invocará 7 veces.

Completa en `SetMultiMapClient.hs` la definición de la función `compose`:

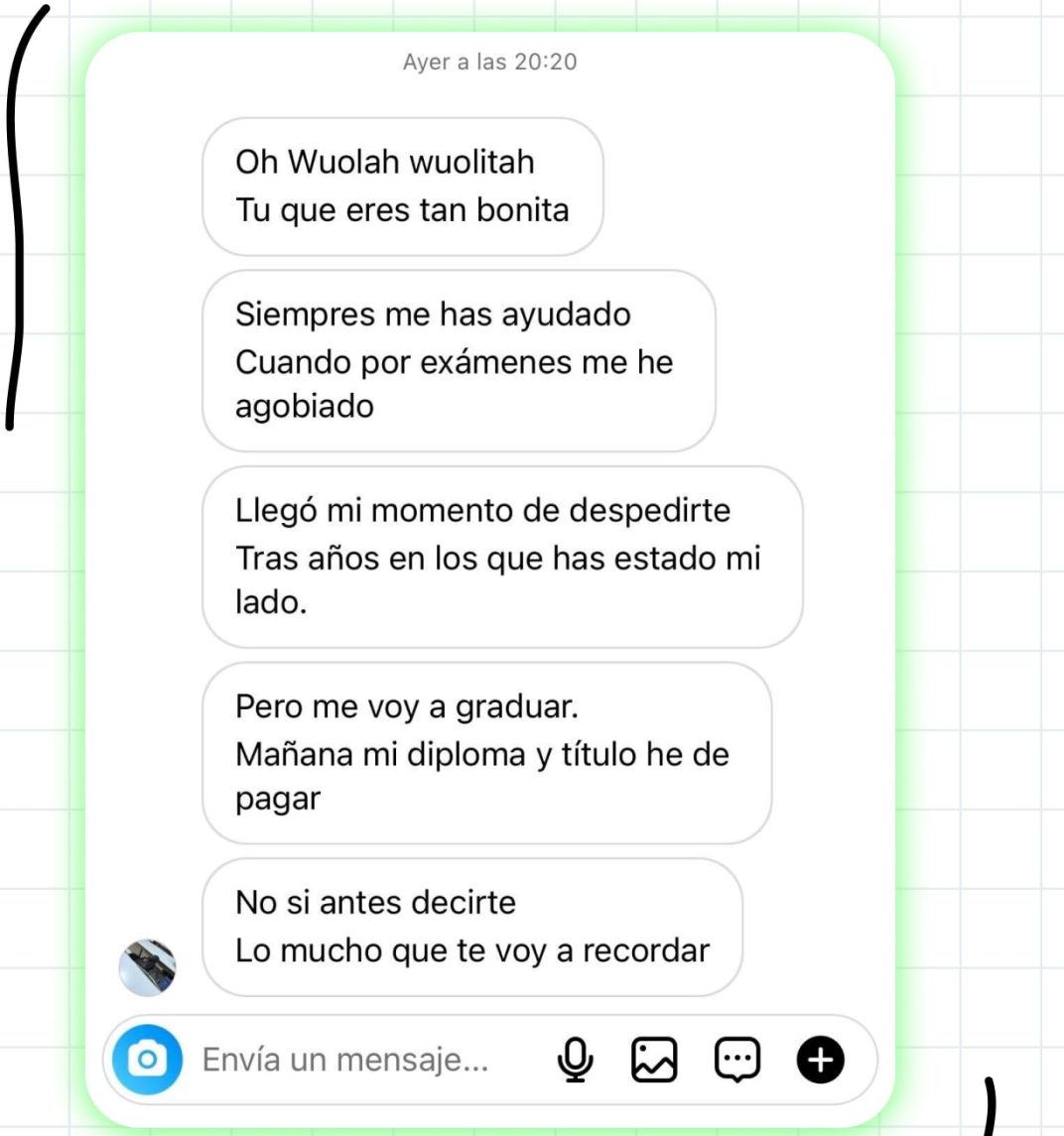
- `compose xs ys`: devuelve la composición de `xs` e `ys` (consultar en `SetMultiMapClient.hs` un ejemplo de uso).

Entrega de la práctica evaluable

- Se deben entregar los ficheros:
  - `SetMultiMap.hs` y
  - `SetMultiMapClient.hs`
- No olvides completar tu nombre, apellidos y grupo
- Para que una función puntúe es necesario que compile

Que no te escriban poemas de amor cuando terminen la carrera 🚩🚩🚩🚩🚩🚩

(a nosotros por suerte nos pasa)



WUOLAH

```haskell
{- |

Estructuras de Datos
2.º A Computadores, 2.º D Informática, 2.º Matemáticas + Informática

Práctica Evaluable - noviembre 2020

Apellidos, Nombre: Fazio Arrabal, Pablo

Grupo: 2.º Matemáticas + Informática


-}

module SetMultiMap( SetMultiMap
                  , empty
                  , isEmpty
                  , size
                  , isDefinedAt
                  , insert
                  , deleteKey
                  , deleteKeyValue
                  , valuesOf
                  , filterValues
                  , fold
                  ) where

import           Data.List                    (intercalate)
import           Test.QuickCheck

import qualified DataStructures.Set.LinearSet as S

-- Invariante de representación:
--   - Los nodos están ordenados por clave
--   - No hay claves repetidas
--   - No hay claves que tengan asociado un conjunto vacío

data SetMultiMap a b = Empty
                     | Node a (S.Set b) (SetMultiMap a b)
                     deriving Eq

-- ejemplo de SetMultiMap para probar las funciones

m1 :: SetMultiMap String Int
m1 = Node "alfredo" (mkSet [9]) (
     Node "juan"    (mkSet [0,1,8]) (
     Node "maria"   (mkSet [4,-6,8])
     Empty))

mkSet :: Eq a => [a] -> S.Set a
mkSet = foldr S.insert S.empty

-- | Ejercicio 1 - Definición de operaciones y complejidad
-----------------------------------------------------------------------

-- 0,25 ptos.
-- |
-- >>> empty
-- {}
```

```haskell

-- Complejidad: O(1)
empty :: SetMultiMap a b
empty = Empty

-- 0,25 ptos.
-- |
-- >>> isEmpty m1
-- False

-- Complejidad: O(1)
isEmpty :: SetMultiMap a b -> Bool
isEmpty Empty = True
isEmpty _ = False

-- 1 pto.
-- |
-- >>> size m1
-- 3

-- Complejidad: O(n)
size :: SetMultiMap a b -> Integer
size Empty = 0
size (Node x xs ys) = 1 + size ys

-- 1 pto.
-- |
-- >>> isDefinedAt "maria" m1
-- True
--
-- >>> isDefinedAt "eva" m1
-- False

-- Complejidad: O(n)
isDefinedAt :: (Ord a, Eq a) => a -> SetMultiMap a b -> Bool
isDefinedAt k Empty = False
isDefinedAt k (Node x xs ys) = (k==x) || isDefinedAt k ys

-- 1 pto.
-- |
-- >>> insert "alfredo" 5 m1
-- "alfredo" --> LinearSet(9,5)
-- "juan" --> LinearSet(8,1,0)
-- "maria" --> LinearSet(8,-6,4)
--
-- >>> insert "carmen" 20 m1
-- "alfredo" --> LinearSet(9)
-- "carmen" --> LinearSet(20)
-- "juan" --> LinearSet(8,1,0)
-- "maria" --> LinearSet(8,-6,4)

-- Complejidad: O(n)
insert :: (Ord a, Eq b) => a -> b -> SetMultiMap a b -> SetMultiMap a b
insert k v Empty = Node k (S.insert v S.empty) Empty
insert k v (Node x xs ys) | k > x && not(isDefinedAt k (Node x xs ys)) = Node x xs
(Node k (S.insert v S.empty) ys)
                          | k == x = Node x (S.insert v xs) ys
                          | otherwise = Node x xs (insert k v ys)

-- 1 pto.
```

```haskell
-- |
-- >>> deleteKey "juan" m1
-- "alfredo" --> LinearSet(9)
-- "maria" --> LinearSet(8,-6,4)

-- Complejidad: O(n)
deleteKey :: (Ord a, Eq b) => a -> SetMultiMap a b -> SetMultiMap a b
deleteKey k Empty = Empty
deleteKey k (Node x xs ys) | k == x = ys
                           | otherwise = Node x xs (deleteKey k ys)

-- 1 pto.
-- |
-- >>> deleteKeyValue "maria" 4 m1
-- "alfredo" --> LinearSet(9)
-- "juan" --> LinearSet(8,1,0)
-- "maria" --> LinearSet(8,-6)

-- Complejidad: O(n)
deleteKeyValue :: (Ord a, Eq b) => a -> b -> SetMultiMap a b -> SetMultiMap a b
deleteKeyValue k v Empty = Empty
deleteKeyValue k v (Node x xs ys) | k == x = if(S.size(S.delete v xs) == 0) then ys
else Node x (S.delete v xs) ys
                                  | otherwise = Node x xs (deleteKeyValue k v ys)

-- 1 pto.
-- |
-- >>> valuesOf "maria" m1
-- Just LinearSet(8,-6,4)
--
-- >>> valuesOf "paco" m1
-- Nothing

-- Complejidad: O(n)
valuesOf :: (Ord a, Eq b) => a -> SetMultiMap a b -> Maybe(S.Set b)
valuesOf k Empty = Nothing
valuesOf k (Node x xs ys) | k == x = Just xs
                          | otherwise = valuesOf k ys

-- 1,25 ptos.
-- |
-- >>> filterValues (> 0) m1
-- "alfredo" --> LinearSet(9)
-- "juan" --> LinearSet(8,1)
-- "maria" --> LinearSet(8,4)

-- Complejidad: O(n)
filterValues :: (Ord a, Eq b) => (b -> Bool)-> SetMultiMap a b -> SetMultiMap a b
filterValues p Empty = Empty
filterValues p (Node x xs ys) | funcional == S.empty = filterValues p ys
                              | otherwise =  Node x funcional (filterValues p ys)
    where
       funcional = (S.fold (\s q -> if p s then S.insert s q else q ) S.empty xs)
 -- S.insert devuelve el Set con los valores de las claves ordenados.

-- | Ejercicio 2 - Axiomas del TAD
-----------------------------------------------------------------------

-- 1 pto.
-- | completa los axiomas que definen deleteKeyValue
```

```haskell

ax_deleteKeyValue_empty x y = deleteKeyValue x y empty == empty
ax_deleteKeyValue_1 k v q = not(isDefinedAt k q) ==> deleteKeyValue k v q == q
-- ax_deleteKeyValue_2 k v q = isDefinedAt k q && S.isElem v (valuesOf k q) ==>
not(S.isElem v (valuesOf k (deleteKeyValue k v q))) -- Sin Tipo Maybe en ValuesOf


------------------------------ NO EDITAR EL CÓDIGO DE ABAJO -----------------------
------

fold :: (Ord a, Eq b) => (a -> b -> c -> c) -> c -> SetMultiMap a b -> c
fold f z ms = recSetMultiMap ms
  where
    recSetMultiMap Empty = z
    recSetMultiMap (Node k s ms)
      | S.isEmpty s = recSetMultiMap ms
      | otherwise = f k v (recSetMultiMap (Node k s' ms))
      where
        (v, s') = pickOne s
    pickOne s = (v, S.delete v s)
      where v = head $ S.fold (:) [] s

instance (Show a, Show b) => Show(SetMultiMap a b) where
  show Empty          = "{}"
  show ms             = intercalate "\n" (showKeyValues ms)
    where
      showKeyValues Empty = []
      showKeyValues (Node k s ms) = (show k ++ " --> " ++ show s) : showKeyValues ms

instance  (Ord a, Arbitrary a, Eq b, Arbitrary b) => Arbitrary (SetMultiMap a b)
where
    arbitrary = do
      xs <- listOf arbitrary
      ys <- listOf arbitrary
      return (foldr (uncurry insert) empty (zip xs ys))

```