Septiembre-2018-Haskell.pdf

Naxetee_

Estructuras de Datos

2º Grado en Ingeniería Informática

Escuela Técnica Superior de Ingeniería Informática
Universidad de Málaga

```haskell
-------------------------------------------------------------------------
-- Apellidos, Nombre: Avila Reyes, Ignacio
-- Titulacion, Grupo: Doble Grado Matemáticas + Ingeniería Informática
--
-- Estructuras de Datos. Grados en Informatica. UMA.
-------------------------------------------------------------------------

module AVLBiDictionary( BiDictionary
                      , empty
                      , isEmpty
                      , size
                      , insert
                      , valueOf
                      , keyOf
                      , deleteByKey
                      , deleteByValue
                      , toBiDictionary
                      , compose
                      , isPermutation
                      , orbitOf
                      , cyclesOf
                      ) where

import qualified DataStructures.Dictionary.AVLDictionary as D
import qualified DataStructures.Set.BSTSet               as S

import           Data.List                               (intercalate, nub,
                                                          (\\))
import           Data.Maybe                              (fromJust, fromMaybe,
                                                          isJust)
import           Test.QuickCheck


data BiDictionary a b = Bi (D.Dictionary a b) (D.Dictionary b a)

-- | Exercise a. empty, isEmpty, size

empty :: (Ord a, Ord b) => BiDictionary a b
empty = Bi (D.empty) (D.empty)

isEmpty :: (Ord a, Ord b) => BiDictionary a b -> Bool
isEmpty (Bi dk dv) = (D.isEmpty dk) && (D.isEmpty dv)

size :: (Ord a, Ord b) => BiDictionary a b -> Int
size (Bi dk dv) = D.size dk

-- | Exercise b. insert

insert :: (Ord a, Ord b) => a -> b -> BiDictionary a b -> BiDictionary a b
insert x y (Bi dk dv)
        | (D.isDefinedAt x dk) = (Bi (D.insert x y dk) (D.insert y x (D.delete
(fromJust(D.valueOf x dk)) dv)))
        | otherwise = (Bi (D.insert x y dk) (D.insert y x dv))

-- | Exercise c. valueOf

valueOf :: (Ord a, Ord b) => a -> BiDictionary a b -> Maybe b
```

```haskell
valueOf x bi@(Bi dk dv)
        | not(D.isDefinedAt x dk) = Nothing
        | otherwise = (D.valueOf x dk)

-- | Exercise d. keyOf

keyOf :: (Ord a, Ord b) => b -> BiDictionary a b -> Maybe a
keyOf y bi@(Bi dk dv)
        | not(D.isDefinedAt y dv) = Nothing
        | otherwise = (D.valueOf y dv)

-- | Exercise e. deleteByKey

deleteByKey :: (Ord a, Ord b) => a -> BiDictionary a b -> BiDictionary a b
deleteByKey x bi@(Bi dk dv)
        | D.isDefinedAt x dk = (Bi (D.delete x dk) (D.delete (fromJust (valueOf x bi)) dv))
        | otherwise = bi

-- | Exercise f. deleteByValue

deleteByValue :: (Ord a, Ord b) => b -> BiDictionary a b -> BiDictionary a b
deleteByValue y bi@(Bi dk dv)
      | D.isDefinedAt y dv = (Bi (D.delete (fromJust(keyOf y bi)) dk) (D.delete y dv))
      | otherwise = bi

-- | Exercise g. toBiDictionary

toBiDictionary :: (Ord a, Ord b) => D.Dictionary a b -> BiDictionary a b
toBiDictionary dic
        | esIny (D.values dic) = foldr (uncurry insert) empty (D.keysValues dic)
        | otherwise = error("toBiDictionary: No es inyectivo")
        where
                esIny [] = True
                esIny (x:xs) | elem x xs = False
                             | otherwise = (esIny xs)

-- | Exercise h. compose

compose :: (Ord a, Ord b, Ord c) => BiDictionary a b -> BiDictionary b c -> BiDictionary a c
compose (Bi dk1 dv1) (Bi dk2 dv2) = (foldr (uncurry insert) empty (list))
        where
          list = [(r,n) | r <- D.keys dk1, n <- D.values dk2, (D.valueOf r dk1)==(D.valueOf n dv2)]

-- | Exercise i. isPermutation

isPermutation :: Ord a => BiDictionary a a -> Bool
isPermutation (Bi dk dv) = (D.keys dk) == (D.keys dv)



-- |-----------------------------------------------------------------------



-- | Exercise j. orbitOf

orbitOf :: Ord a => a -> BiDictionary a a -> [a]
```

```haskell
orbitOf = undefined

-- | Exercise k. cyclesOf

cyclesOf :: Ord a => BiDictionary a a -> [[a]]
cyclesOf = undefined

-- |----------------------------------------------------------------------


instance (Show a, Show b) => Show (BiDictionary a b) where
  show (Bi dk dv)  = "BiDictionary(" ++ intercalate "," (aux (D.keysValues dk)) ++ ")"
                       ++ "(" ++ intercalate "," (aux (D.keysValues dv)) ++ ")"
    where
     aux kvs  = map (\(k,v) -> show k ++ "->" ++ show v) kvs

```