haskell.pdf

Luis_SX

Estructuras de Datos

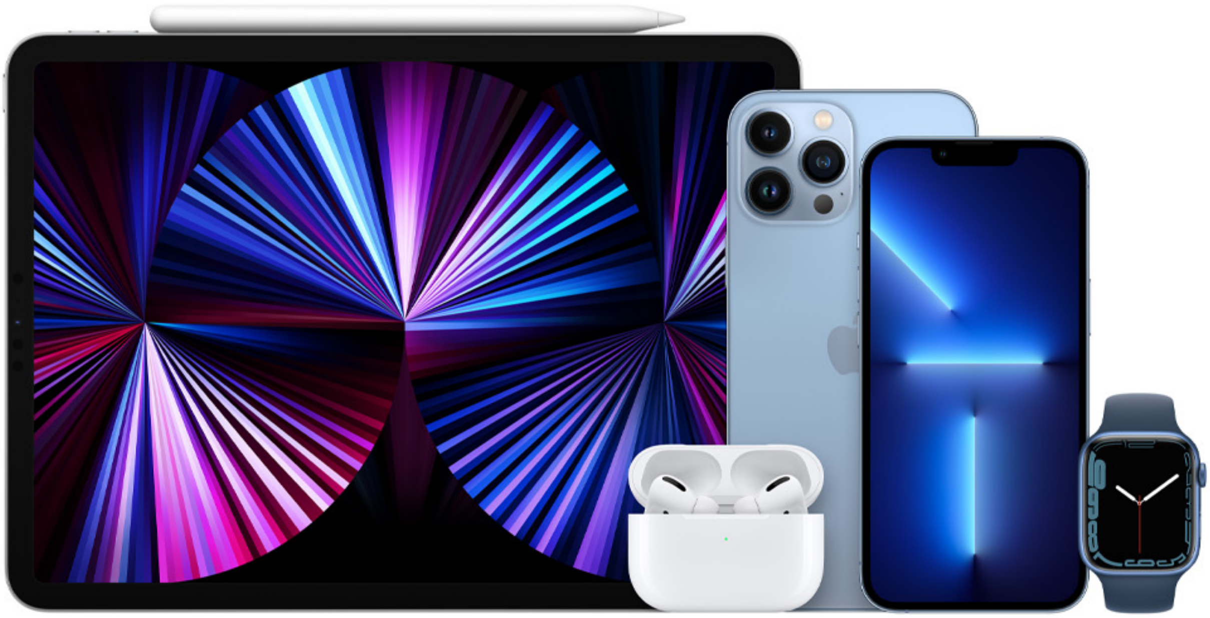
2º Grado en Ingeniería Informática

Escuela Técnica Superior de Ingeniería Informática
Universidad de Málaga

```
-- | Data Structures
-- | September, 2016
-- |
-- | Student's name:
-- | Student's group:

module Huffman where

import qualified DataStructures.Dictionary.AVLDictionary as D
import qualified
DataStructures.PriorityQueue.WBLeftistHeapPriorityQueue as PQ
import Data.List (nub)

-- | Exercise 1

weights :: Ord a => [a] -> D.Dictionary a Int
weights l = peso l D.empty



peso:: Ord a => [a] -> D.Dictionary a Int->D.Dictionary a Int
peso [] d = d
peso (x:xs) d= peso xs (D.updateOrInsert x (+1) 1 d)

sacaJust::Maybe a->a
sacaJust Nothing = error "Nothing found"
sacaJust (Just x) = x

suma1::Int->Int
suma1 x = x+1

{-

> weights "abracadabra"
AVLDictionary('a'->5,'b'->2,'c'->1,'d'->1,'r'->2)

> weights [1,2,9,2,0,1,6,1,5,5,8]
AVLDictionary(0->1,1->3,2->2,5->2,6->1,8->1,9->1)

> weights ""
AVLDictionary()

-}


-- Implementation of Huffman Trees
data WLeafTree a = WLeaf a Int  -- Stored value (type a) and weight
(type Int)
                 | WNode (WLeafTree a) (WLeafTree a) Int -- Left
child, right child and weight
                 deriving (Eq, Show)

weight :: WLeafTree a -> Int
weight (WLeaf _ n)   = n
weight (WNode _ _ n) = n

-- Define order on trees according to their weights
instance Eq a => Ord (WLeafTree a) where
  wlt <= wlt' =  weight wlt <= weight wlt'

-- Build a new tree by joining two existing trees
```

```haskell
merge :: WLeafTree a -> WLeafTree a -> WLeafTree a
merge wlt1 wlt2 = WNode wlt1 wlt2 (weight wlt1 + weight wlt2)


-- | Exercise 2

-- 2.a
huffmanLeaves :: String -> PQ.PQueue (WLeafTree Char)
huffmanLeaves s = creapq (D.keysValues (weights s)) PQ.empty

creapq:: [(Char,Int)]->PQ.PQueue (WLeafTree Char)->PQ.PQueue
(WLeafTree Char)
creapq [] p=p
creapq (x:xs) p= creapq xs (PQ.enqueue (WLeaf (key x) (value x)) p)


key::(a,w)->a
key (k,v)=k

value::(a,w)->w
value (k,v)=v
{-

> huffmanLeaves "abracadabra"
WBLeftistHeapPriorityQueue(WLeaf 'c' 1,WLeaf 'd' 1,WLeaf 'b' 2,WLeaf
'r' 2,WLeaf 'a' 5)

-}

-- 2.b
huffmanTree :: String -> WLeafTree Char
huffmanTree s = if PQ.size (huffmanLeaves s)<2 then error"the string
must have at least two different symbols"else  crearbol (huffmanLeaves
s)

crearbol:: PQ.PQueue (WLeafTree Char) ->WLeafTree Char
crearbol p
        |PQ.size p ==1 = PQ.first p
        |otherwise = crearbol (PQ.enqueue (merge a1 a2)
(PQ.dequeue(PQ.dequeue p) ))
          where
              a1= PQ.first p
              a2= PQ.first (PQ.dequeue p)


{-

> printWLeafTree $ huffmanTree "abracadabra"
      11_______
      /        \
('a',5)        6________________
              /                 \
        ('r',2)          _______4
                        /       \
                       2        ('b',2)
                      / \
                ('c',1) ('d',1)

> printWLeafTree $ huffmanTree "abracadabra pata de cabra"
                         ______________________25_____________
                        /                                     \
```

All the tech you need at your fingertips



Grover

SCAN ME

Get your **15% Student Discount** on grover.com

```
           ______________10______                          ______15
          /                      \                        /       \
         4_______                6_______                6
('a',9)
      /         \              /         \              / \
('d',2)          2        (' ',3)         3        ('b',3) ('r',3)
               / \                       / \
        ('e',1) ('p',1)          ('t',1) ('c',2)

> printWLeafTree $ huffmanTree "aaa"
*** Exception: huffmanTree: the string must have at least two
different symbols

-}


-- | Exercise 3

-- 3.a
joinDics :: (Ord a,Ord b) => D.Dictionary a b -> D.Dictionary a b ->
D.Dictionary a b
joinDics d1 d2= junta (D.keysValues d1) (D.keysValues d2) D.empty



junta:: (Ord a,Ord b) => [(a,b)]->[(a,b)]->D.Dictionary a b-
>D.Dictionary a b
junta [] [] d= d
junta [] (x:xs) d= junta [] xs (D.insert (key x) (value x) d)
junta (y:ys) [] d = junta ys [] (D.insert (key y) (value y) d)
junta (y:ys) (x:xs) d
          |value y >= value x = junta ys (x:xs) (D.insert (key y)
(value y) d)
          |otherwise = junta (y:ys) xs (D.insert (key x) (value x) d)
{-

> joinDics (D.insert 'a' 1 $ D.insert 'c' 3 $ D.empty) D.empty
AVLDictionary('a'->1,'c'->3)

> joinDics (D.insert 'a' 1 $ D.insert 'c' 3 $ D.empty) (D.insert 'b' 2
$ D.insert 'd' 4 $ D.insert 'e' 5 $ D.empty)
AVLDictionary('a'->1,'b'->2,'c'->3,'d'->4,'e'->5)

-}

-- 3.b
prefixWith :: Ord a => b -> D.Dictionary a [b] -> D.Dictionary a [b]
prefixWith x d= concatena d (D.keys d) x


concatena::(Ord a)=> D.Dictionary a [b]->[a]->b->D.Dictionary a [b]
concatena d [] e = d
concatena d (x:xs) e = concatena (D.updateOrInsert x ([e]++) [e] d) xs
e

{-

> prefixWith 0 (D.insert 'a' [0,0,1] $ D.insert 'b' [1,0,0] $ D.empty)
AVLDictionary('a'->[0,0,0,1],'b'->[0,1,0,0])

> prefixWith 'h' (D.insert 1 "asta" $ D.insert 2 "echo" $ D.empty)
```

```
AVLDictionary(1->"hasta",2->"hecho")

-}

-- 3.c
huffmanCode :: WLeafTree Char -> D.Dictionary Char [Integer]
huffmanCode t = code t [] D.empty

code::WLeafTree Char ->[Integer]-> D.Dictionary Char [Integer]-
>D.Dictionary Char [Integer]
code (WLeaf x _) l d =D.insert x l d
code (WNode lt rt _) l d=joinDics (code lt (l++[0]) d) (code rt
(l++[1]) d)


{-

> huffmanCode (huffmanTree "abracadabra")
AVLDictionary('a'->[0],'b'->[1,1,1],'c'->[1,1,0,0],'d'->[1,1,0,1],'r'-
>[1,0])

-}

-- ONLY for students not taking continuous assessment

-- | Exercise 4

encode :: String -> D.Dictionary Char [Integer] -> [Integer]
encode [] _ = []
encode (x:xs) d =  sacaJust(D.valueOf x d) ++encode xs d

{-

> encode "abracadabra" (huffmanCode (huffmanTree "abracadabra"))
[0,1,1,1,1,0,0,1,1,0,0,0,1,1,0,1,0,1,1,1,1,0,0]

-}

-- | Exercise 5

-- 5.a
takeSymbol :: [Integer] -> WLeafTree Char -> (Char, [Integer])
takeSymbol = undefined

{-

> takeSymbol [0,1,1,1,1,0,0,1,1,0,0,0,1,1,0,1,0,1,1,1,1,0,0]
(huffmanTree "abracadabra")
('a',[1,1,1,1,0,0,1,1,0,0,0,1,1,0,1,0,1,1,1,1,0,0])

> takeSymbol [1,1,1,1,0,0,1,1,0,0,0,1,1,0,1,0,1,1,1,1,0,0]
(huffmanTree "abracadabra")
('b',[1,0,0,1,1,0,0,0,1,1,0,1,0,1,1,1,1,0,0])

-}

-- 5.b
decode :: [Integer] -> WLeafTree Char -> String
decode = undefined

{-
```

```
> decode [0,1,1,1,1,0,0,1,1,0,0,0,1,1,0,1,0,1,1,1,1,0,0] (huffmanTree
"abracadabra")
"abracadabra"

-}

----------------------------------------------------------------------
---------
-- Pretty Printing a WLeafTree
-- (adapted from http://stackoverflow.com/questions/1733311/pretty-
print-a-tree)
----------------------------------------------------------------------
---------

printWLeafTree :: (Show a) => WLeafTree a -> IO ()
printWLeafTree t = putStrLn (unlines xss)
 where
   (xss,_,_,_) = pprint t

pprint :: Show a => WLeafTree a -> ([String], Int, Int, Int)
pprint (WLeaf x we)             =  ([s], ls, 0, ls-1)
  where
    s = show (x,we)
    ls = length s
pprint (WNode lt rt we)         =  (resultLines, w, lw'-swl,
totLW+1+swr)
  where
    nSpaces n = replicate n ' '
    nBars n = replicate n '_'
    -- compute info for string of this node's data
    s = show we
    sw = length s
    swl = div sw 2
    swr = div (sw-1) 2
    (lp,lw,_,lc) = pprint lt
    (rp,rw,rc,_) = pprint rt
    -- recurse
    (lw',lb) = if lw==0 then (1," ") else (lw,"/")
    (rw',rb) = if rw==0 then (1," ") else (rw,"\\")
    -- compute full width of this tree
    totLW = maximum [lw', swl,  1]
    totRW = maximum [rw', swr, 1]
    w = totLW + 1 + totRW
{-
A suggestive example:
     dddd | d | dddd__
        / |   |       \
      lll |   |       rr
          |   |      ...
          |   | rrrrrrrrrrr
     ----       ----           swl, swr (left/right string width (of
this node) before any padding)
      ---       -----------    lw, rw   (left/right width (of subtree)
before any padding)
     ----                      totLW
                -----------    totRW
     ----   -   -----------    w (total width)
-}
    -- get right column info that accounts for left side
    rc2 = totLW + 1 + rc
```

```
    -- make left and right tree same height
    llp = length lp
    lrp = length rp
    lp' = if llp < lrp then lp ++ replicate (lrp - llp) "" else lp
    rp' = if lrp < llp then rp ++ replicate (llp - lrp) "" else rp
    -- widen left and right trees if necessary (in case parent node is
wider, and also to fix the 'added height')
    lp'' = map (\s -> if length s < totLW then nSpaces (totLW - length
s) ++ s else s) lp'
    rp'' = map (\s -> if length s < totRW then s ++ nSpaces (totRW -
length s) else s) rp'
    -- first part of line1
    line1 = if swl < lw' - lc - 1 then
                nSpaces (lc + 1) ++ nBars (lw' - lc - swl) ++ s
            else
                nSpaces (totLW - swl) ++ s
    -- line1 right bars
    lline1 = length line1
    line1' = if rc2 > lline1 then
                line1 ++ nBars (rc2 - lline1)
             else
                line1
    -- line1 right padding
    line1'' = line1' ++ nSpaces (w - length line1')
    -- first part of line2
    line2 = nSpaces (totLW - lw' + lc) ++ lb
    -- pad rest of left half
    line2' = line2 ++ nSpaces (totLW - length line2)
    -- add right content
    line2'' = line2' ++ " " ++ nSpaces rc ++ rb
    -- add right padding
    line2''' = line2'' ++ nSpaces (w - length line2'')
    resultLines = line1'' : line2''' : zipWith (\lt rt -> lt ++ " " ++
rt) lp'' rp''
```