

```

-- | 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 = undefined

{-

> 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
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 = undefined

{-

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

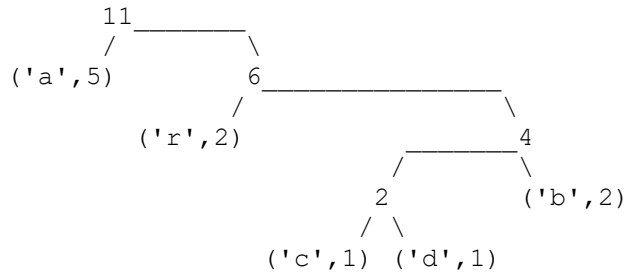
-}

-- 2.b
huffmanTree :: String -> WLeafTree Char
huffmanTree = undefined

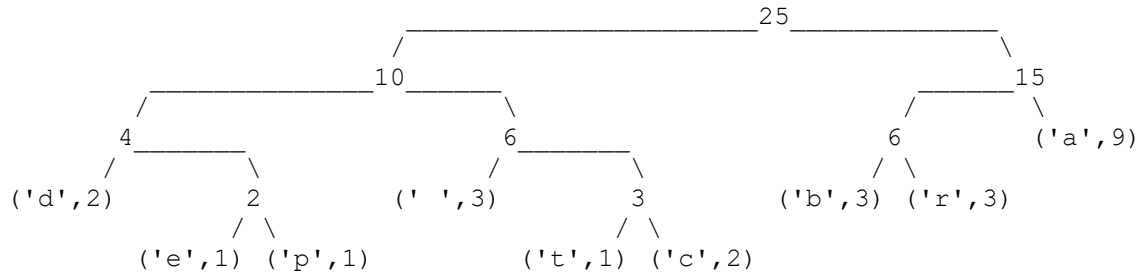
```

```
{-
```

```
> printWLeafTree $ huffmanTree "abracadabra"
```



```
> printWLeafTree $ huffmanTree "abracadabra pata de cabra"
```



```
> printWLeafTree $ huffmanTree "aaa"
```

```
*** Exception: huffmanTree: the string must have at least two different symbols
```

```
-}
```

```
-- | Exercise 3
```

```
-- 3.a
```

```
joinDics :: Ord a => D.Dictionary a b -> D.Dictionary a b -> D.Dictionary a b
```

```
joinDics = undefined
```

```
{-
```

```
> 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 = undefined
```

```
{-
```

```
> 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 = undefined
```

```
{-
```

```

> 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 = undefined

{-

> 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_
        / |   | \
      111 |   |   rr
         |   |   ...
         |   | rrrrrrrrrrr
-----
before any padding)      swl, swr (left/right string width (of this node)
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''

```