```haskell
-------------------------------------------------------------------------------
-- Student's name: Juan Díaz-Flores Merino
-- Student's group: 2°A
-- Identity number (DNI if Spanish/passport if Erasmus):
--
-- Data Structures. Grado en Informática. UMA.
-------------------------------------------------------------------------------


module DataStructures.Trie.DictionaryStringTrie(
    Trie()
  , empty
  , isEmpty
  , size
  , search
  , insert
  , strings
  , fromList
  , pretty
  , sampleTrie, sampleTrie1, sampleTrie2, sampleTrie3, sampleTrie4
  -- sizeValue, toTrie, childOf, update
  ) where

import qualified Control.DeepSeq as Deep
import Data.Maybe
import qualified DataStructures.Dictionary.AVLDictionary as D

data Trie a = Empty | Node (Maybe a) (D.Dictionary Char (Trie a)) deriving Show

-------------------------------------------------------------------------------
-- DO NOT WRITE ANY CODE ABOVE ------------------------------------------------
-------------------------------------------------------------------------------


-- | = Exercise a - empty
empty :: Trie a
empty = Empty

-- | = Exercise b - isEmpty
isEmpty :: Trie a -> Bool
isEmpty Empty = True
isEmpty _ = False

-- | = Exercise c - sizeValue
sizeValue :: Maybe a -> Int
sizeValue Nothing = 0
sizeValue _ = 1

-- | = Exercise d - size
```

```haskell
size :: Trie a -> Int
size Empty = 0
size (Node m dic) = aux (D.values dic) (sizeValue m)
   where
     aux [] cont = cont
     aux (x:xs) cont = aux xs (cont + size x)

-- | = Exercise e - toTrie
toTrie :: Maybe (Trie a) -> Trie a
toTrie may
   | isNothing may = Empty
   | otherwise = fromJust may

-- | = Exercise f - childOf
childOf :: Char -> Trie a -> Trie a
childOf c Empty = Empty
childOf c (Node m dic) = toTrie (D.valueOf c dic)

-- | = Exercise g - search
search :: String -> Trie a -> Maybe a
search _ Empty = Nothing
search [] (Node v _) = v
search (c:cs) t = search cs (childOf c t)

-- | = Exercise h - update
update :: Trie a -> Char -> Trie a -> Trie a
update Empty c child = (Node Nothing (D.insert c child D.empty))
update (Node v dic) c child = (Node v (D.insert c child dic))

-- | = Exercise i - insert
insert :: String -> a -> Trie a -> Trie a
insert [] v Empty = (Node (Just v) D.empty)
insert [] v (Node _ dic) = (Node (Just v) dic)
insert (c:cs) v t = update t c child
      where
        child = insert cs v (childOf c t)

--insert (c:cs) v t@Empty = (Node Nothing (D.insert c child D.empty))
--insert (c:cs) v t@(Node v' dic) = (Node v' (D.insert c child dic))
--     where
--        child = insert cs v (childOf c t)

--------------------------------------------------------------------------------
-- ONLY FOR PART TIME STUDENTS -------------------------------------------------
--------------------------------------------------------------------------------

-- | = Exercise e1 - strings
```

```
strings :: Trie a -> [String]
strings Empty = []
strings (Node mb dic)
   | isJust mb =  "" : aux   -- Incluye la palabra vacia si el nodo actual tiene un valor
asociado
   | otherwise = aux
       where
         aux = [c : s | (c, child) <- D.keysValues dic, s <- strings child]



-- | = Exercise e2 - fromList
fromList :: [String] -> Trie Int
fromList lista = foldl aux empty lista
  where
    aux :: Trie Int -> String -> Trie Int
    aux trie word = insert word w trie
      where
          v = search word trie
          w = if (isJust v) then v+1 else 1





-------------------------------------------------------------------------------
-- DO NOT WRITE ANY CODE BELOW ------------------------------------------------
-------------------------------------------------------------------------------


pretty :: (Show a) => Trie a -> IO ()
pretty t = putStrLn (showsTrie t "")

showsTrie :: (Show a) => Trie a -> ShowS
showsTrie Empty       = shows "Empty"
showsTrie (Node mb d) = showString "Node " . showValue mb . showChar '\n' . aux 1 d
  where
    aux n d =
      foldr (.) id [ showString (replicate (6*n) ' ')
                      . showChar c
                      . showString " -> "
                      . showString "Node "
                      . showValue mb
                      . showChar '\n'
                      . aux (n+1) d'
                    | (c, Node mb d') <- D.keysValues d
                    ]

    showValue mb = maybe (shows mb) (const (showChar '(' . shows mb . showChar ')')) mb
```

```haskell
      n1 = Node (Just 1) $ children [('b', n2), ('e', n5)]
      n2 = Node (Just 2) $ children [('c', n3), ('d', n4)]
      n3 = Node (Just 3) $ children []
      n4 = Node (Just 4) $ children []
      n5 = Node Nothing $ children [('d', n6)]
      n6 = Node Nothing $ children [('e', n7)]
      n7 = Node Nothing $ children [('f', n8)]
      n8 = Node (Just 5) $ children []

sampleTrie3 :: Trie Integer
sampleTrie3 = n0
   -- abcd -> 1
   where
      children = foldr (uncurry D.insert) D.empty
      n0 = Node Nothing $ children [('a', n1)]
      n1 = Node Nothing $ children [('b', n2)]
      n2 = Node Nothing $ children [('c', n3)]
      n3 = Node Nothing $ children [('d', n4)]
      n4 = Node (Just 1) $ children []

sampleTrie4 :: Trie Integer
sampleTrie4 = n0
   -- abcd -> 1  def -> 2
   where
      children = foldr (uncurry D.insert) D.empty
      n0 = Node Nothing $ children [('a', n1), ('d', n5)]
      n1 = Node Nothing $ children [('b', n2)]
      n2 = Node Nothing $ children [('c', n3)]
      n3 = Node Nothing $ children [('d', n4)]
      n4 = Node (Just 1) $ children []
      n5 = Node Nothing $ children [('e', n6)]
      n6 = Node Nothing $ children [('f', n7)]
      n7 = Node (Just 2) $ children []
```