

Main:

```
import Diccionario
import Data.Maybe
import Arbol23
import Test.HUnit
```

--Este módulo sirve para utilizar el diccionario sin acceder a su estructura interna. Pueden agregar otras funciones o casos de prueba.

{- Función a implementar. -}

```
búsquedaDelTesoro::Eq a=>a->(a->Bool)->Diccionario a a->Maybe a
búsquedaDelTesoro pista esTesoro d = (head (filter (esTesoroONothing esTesoro) (iterate (obtenerONothing d) (Just pista))))
```

```
esTesoroONothing::(a->Bool)->Maybe a->Bool
esTesoroONothing esTesoro = maybe True esTesoro
```

```
obtenerONothing::Eq a=>Diccionario a a -> Maybe a -> Maybe a
obtenerONothing d c = case c of
    Nothing -> Nothing
    Just c -> obtener c d
```

{- Arbol de prueba: -}

```
arbolDeg::Arbol23 Char Int
arbolDeg = Dos 1 (Hoja 'a') (Dos 2 (Hoja 'b') (Tres 3 4 (Hoja 'c') (Hoja 'd') (Dos 5 (Hoja 'e') (Hoja 'f'))))
arbolOps::Arbol23 Int (Int->Int->Int)
arbolOps = Tres (+) (*) (Dos (-) (Hoja 10) (Hoja 20)) (Dos (-) (Hoja 10) (Hoja 30)) (Tres (*) (*) (Hoja 2) (Hoja 2) (Hoja 2))
arbolOps2::Arbol23 Int (Int->Int->Int)
arbolOps2 = Tres (+) (*) (Dos (+) (Hoja 10) (Hoja 20)) (Dos (+) (Hoja 10) (Hoja 30)) (Tres (*) (*) (Hoja 2) (Hoja 2) (Hoja 2))
```

{- Diccionarios de prueba: -}

```
dicc1::Diccionario Int String
dicc1 = definirVarías [(0,"Hola"),(-10,"Chau"),(15,"Felicidades"),(2,"etc."),(9,"a")] (vacío (<))
```

```
dicc2::Diccionario String String
dicc2 = definirVarías [("inicio","casa"),("auto","flores"),("calle","auto"),("casa","escalera"),("ropero","alfajor"),("escalera","ropero")] (vacío (<))
```

```
dicc3::Diccionario Int String
dicc3 = definirVarías [(0,"Hola"),(-10,"Chau"),(15,"Felicidades"),(2,"etc."),(9,"a")] (vacío (\x y->x `mod` 5 < y `mod` 5))
```

```
diccOrdenado::Diccionario Int Int
diccOrdenado = definirVarías [(0,1),(1,2),(2,3),(3,4),(4,5)] (vacío (<))
```

```
diccVacio::Diccionario String String
diccVacio = definirVarías [] (vacío (<))
```

```
diccA1::Diccionario Int Int
diccA1 = definir 0 0 (vacío (<))
diccA2::Diccionario Int Int
```

```

diccA2 = definir 1 0 (diccA1)
diccA3::Diccionario Int Int
diccA3 = definir 2 0 (diccA2)
diccB1::Diccionario Int Int
diccB1 = definir 2 0 (vacio (<))
diccB2::Diccionario Int Int
diccB2 = definir 1 0 (diccB1)
diccB3::Diccionario Int Int
diccB3 = definir 0 0 (diccB2)

--Ejecución de los tests
main :: IO Counts
main = do runTestTT allTests

allTests = test [
  "ejercicio2" ~: testsEj2,
  "ejercicio3" ~: testsEj3,
  "ejercicio4" ~: testsEj4,
  "ejercicio5" ~: testsEj5,
  "ejercicio6" ~: testsEj6,
  "ejercicio7" ~: testsEj7,
  "ejercicio8" ~: testsEj8,
  "ejercicio9" ~: testsEj9,
  "ejercicio10" ~: testsEj10
]

testsEj2 = test [
  [0,1,2,3,4,5,6,7] ~=? internos arbolito1,
  "abcdefghi" ~=? hojas arbolito1,
  [True,False,True] ~=? internos arbolito2,
  [1,2,3,2,3,4,3,4,5,4] ~=? take 10 (hojas arbolito3),
  [5,2,0,1,12,(-3),4,9,20,7] ~=? hojas (arbolito4),
  ['p','l','g','r','a','p','n','d','e'] ~=? internos (arbolito4),
  [True, False, True] ~=? internos (arbolito2),
  True ~=? esHoja (Hoja 'a'),
  False ~=? esHoja (Dos 3 (Hoja 'a') (Hoja 'a')),
  False ~=? esHoja (Tres 3 5 (Hoja 'a') (Hoja 'a') (Hoja 'a')),
  ['a','b','c','d','e','f'] ~=? hojas (arbolDeg),
  [1,2,3,4,5] ~=? internos (arbolDeg)
]

testsEj3 = test [
  [0,1,-1,5] ~=? hojas (incrementarHojas arbolito2),
  [-4,0,-8,16] ~=? hojas (mapA23 (\x-> x*4) id arbolito2),
  [False,True,False] ~=? internos (mapA23 id (\x-> not x) arbolito2),
  ['w','w','w','w','w','w'] ~=? hojas (mapA23 (\x-> 'w') id arbolDeg),
  [0,0,0,0,0] ~=? internos (mapA23 id (\x-> 0) arbolDeg)
]

testsEj4 = test [
  [1,2,3,2,3,4,3,4,5,4,5,6,0,0,0,0] ~=? hojas (truncar 0 6 arbolito3),
  [0] ~=? hojas (truncar 0 0 arbolito5),
  ['a'] ~=? internos (truncar 0 1 arbolito5),
  ['a','b','c','a'] ~=? internos (truncar 0 2 arbolito5),
  [0,0,0,0,0] ~=? hojas (truncar 0 2 arbolito5),
  [0,0,0,0,0,0,0,0,0,0] ~=? hojas (truncar 0 3 arbolito4),
  [1,1,1,1] ~=? hojas (truncar 1 2 arbolito4),
  ['w','w','w','w','w','w','w'] ~=? hojas (truncar 'w' 2 arbolito1),

```

```

['w'] ~=? hojas (truncar 'w' 0 arbolito1),
[99] ~=? hojas (truncar 99 0 arbolito4),
[99,99] ~=? hojas (truncar 99 1 arbolito4),
['p'] ~=? internos (truncar 99 1 arbolito4),
[] ~=? internos (truncar 99 0 arbolito4),
[1] ~=? internos (truncar 'a' 1 arbolDeg),
[1,2,3,4] ~=? internos (truncar 'a' 3 arbolDeg),
['a','b','a','a','a'] ~=? hojas (truncar 'a' 3 arbolDeg)
]

testsEj5 = test [
  45 ~=? evaluar (truncar 0 7 arbolito3),
  22 ~=? evaluar (truncar 0 6 arbolito3),
  8 ~=? evaluar (truncar 0 5 arbolito3),
  1 ~=? evaluar (truncar 0 4 arbolito3),
  (-1) ~=? evaluar (truncar 0 3 arbolito3),
  (-240) ~=? evaluar arbolOps,
  (560) ~=? evaluar arbolOps2
]

testsEj6 = test [
  [] ~=? claves (diccVacio)
]

testsEj7 = test [
  [0] ~=? claves(diccA1),
  [0,1] ~=? claves(diccA2),
  [0,1,2] ~=? claves(diccA3),
  [2] ~=? claves(diccB1),
  [1,2] ~=? claves(diccB2),
  [0,1,2] ~=? claves(diccB3),
  claves(diccA3) ~=? claves(diccB3)
]

testsEj8 = test [
  Nothing ~=? obtener "algo" diccVacio,
  Nothing ~=? obtener "algo" dicc2,
  Just "Felicidades" ~=? obtener 15 dicc1,
  Just "cartones" ~=? obtener "flores" (definir "flores" "cartones" dicc2)
]

testsEj9 = test [
  [] ~=? claves diccVacio,
  [0,1,2,3,4] ~=? claves diccOrdenado,
  ["auto","calle","casa","escalera","inicio","ropero"] ~=? claves dicc2,
  claves dicc1 ++ [999] ~=? claves (definir 999 "etc" dicc1)
]

testsEj10 = test [
  Just "alfajor" ~=? búsquedaDelTesoro "inicio" ((=='a').head) dicc2,
  Just 5 ~=? búsquedaDelTesoro 0 (==5) diccOrdenado,
  Nothing ~=? búsquedaDelTesoro 0 (==6) diccOrdenado,
  Just "oro" ~=? búsquedaDelTesoro "inicio" (=="oro") (definir "alfajor" "oro" dicc2)
]

```

Arbol23:

module Arbol23 where

```
data Arbol23 a b = Hoja a | Dos b (Arbol23 a b) (Arbol23 a b) | Tres b b (Arbol23 a b) (Arbol23 a b)
(Arbol23 a b)
```

{- Funciones para mostrar el árbol. -}

```
instance (Show a, Show b) => Show (Arbol23 a b) where
  show = ("\n" ++) . (padTree 0 0 False)
```

padlength = 5

```
padTree:: (Show a, Show b) => Int -> Int -> Bool -> (Arbol23 a b) -> String
```

```
padTree nivel acum doPad t = case t of
  (Hoja x) -> initialPad ++ stuff x
  (Dos x i d) -> initialPad ++ stuff x ++
    pad padlength ++ rec x False i ++ "\n" ++
    rec x True d ++ "\n"
  (Tres x y i m d) -> initialPad ++ stuff x ++ --(' ':tail (stuff y)) ++
    pad padlength ++ rec x False i ++ "\n" ++
    pad levelPad ++ stuff y ++ pad padlength ++ rec x False m ++ "\n" ++
    rec x True d ++ "\n"
```

```
where l = length . stuff
      levelPad = (padlength*nivel + acum)
      initialPad = (if doPad then pad levelPad else "")
      rec x = padTree (nivel+1) (acum+l x)
```

```
stuff:: Show a => a -> String
stuff x = if n > l then pad (n-l) ++ s else s
where s = show x
      l = length s
      n = padlength
```

```
pad:: Int -> String
pad i = replicate i ' '
```

{- Funciones pedidas. -}

--foldA23::

```
foldA23 :: (a->c)->(b->c->c->c)->(b->b->c->c->c->c)->Arbol23 a b->c
```

```
foldA23 f g h (Hoja a) = f a
```

```
foldA23 f g h (Dos b ar1 ar2) = g b (fold ar1) (fold ar2) where fold = foldA23 f g h
```

```
foldA23 f g h (Tres b c ar1 ar2 ar3) = h b c (fold ar1) (fold ar2) (fold ar3) where fold = foldA23 f g h
```

--Lista en preorden de los internos del árbol.

```
internos::Arbol23 a b->[b]
```

```
internos = foldA23 (const []) (\b x y -> b : x ++ y) (\b c x y z -> b : c : (x ++ y ++ z))
```

--Lista las hojas de izquierda a derecha.

```
hojas::Arbol23 a b->[a]
```

```
hojas = foldA23 (\x -> [x]) (const(++)) (\_ _ x y z -> x ++ y ++ z)
```

```

esHoja::Arbol23 a b->Bool
esHoja ar = case ar of
    Hoja _ -> True
    otherwise -> False

mapA23::(a->c)->(b->d)->Arbol23 a b->Arbol23 c d
mapA23 f g = foldA23 (\val -> (Hoja (f val)))
    (\val ai ad -> (Dos (g val) ai ad))
    (\val val2 ai ac ad -> (Tres (g val) (g val2) ai ac ad))

--Ejemplo de uso de mapA23.
--Incrementa en 1 el valor de las hojas.
incrementarHojas::Num a =>Arbol23 a b->Arbol23 a b
incrementarHojas = mapA23 (+1) id

--Trunca el árbol hasta un determinado nivel. Cuando llega a 0, reemplaza el resto del árbol por una hoja con
el valor indicado.
--Funciona para árboles infinitos.

truncar::a->Integer->Arbol23 a b->Arbol23 a b
truncar e lim arb = foldA23
    (\val -> (\i -> hojaORama i hojaE (Hoja val)))
    (\val ai ad -> (\i->let iDec=(i-1) in hojaORama i hojaE (Dos val (ai iDec) (ad iDec)) ))
    (\val val2 ai ac ad -> (\i->let iDec=(i-1) in hojaORama i hojaE (Tres val val2 (ai iDec) (ac
iDec) (ad iDec)) )) arb lim
    where hojaE = Hoja e

hojaORama:: Integer-> Arbol23 a b ->Arbol23 a b ->Arbol23 a b
hojaORama i h a = if (i==0) then h else a

--Evalúa las funciones tomando los valores de los hijos como argumentos.
--En el caso de que haya 3 hijos, asocia a izquierda.
evaluar::Arbol23 a (a->a->a)->a
evaluar = foldA23 (id)
    (id)
    (\val val2 ai ac ad -> (val2 (val ai ac) ad))

--Ejemplo:
--evaluar (truncar 0 6 arbolito3) = 22 = (1*2-3)+(2*3-4)+(3*4-5)+(4*5-6)

{- Árboles de ejemplo. -}

arbolito1::Arbol23 Char Int
arbolito1 = Tres 0 1
    (Dos 2 (Hoja 'a') (Hoja 'b'))
    (Tres 3 4 (Hoja 'c') (Hoja 'd') (Dos 5 (Hoja 'e') (Hoja 'f'))))
    (Dos 6 (Hoja 'g') (Dos 7 (Hoja 'h') (Hoja 'i'))))

arbolito2::Arbol23 Int Bool
arbolito2 = Dos True (Hoja (-1)) (Tres False True (Hoja 0) (Hoja (-2)) (Hoja 4))

arbolito3::Arbol23 Int (Int->Int->Int)
arbolito3 = Dos (+) (Tres (*) (-) (Hoja 1) (Hoja 2) (Hoja 3)) (incrementarHojas arbolito3)

arbolito5::Arbol23 Int (Char)
arbolito5 = Dos ('a') (Tres ('b') ('c') (Hoja 1) (Hoja 2) (Hoja 3)) (incrementarHojas arbolito5)

arbolito4::Arbol23 Int Char

```

```
arbolito4 = Dos 'p' (Dos 'l' (Dos 'g' (Hoja 5) (Hoja 2)) (Tres 'r' 'a' (Hoja 0)(Hoja 1)(Hoja 12)))
            (Dos 'p' (Tres 'n' 'd' (Hoja (-3))(Hoja 4)(Hoja 9)) (Dos 'e' (Hoja 20)(Hoja 7)))
```

Diccionario:

module Diccionario (Diccionario, vacio, definir, definirVarias, obtener, claves) where

```
import Data.Maybe
import Data.List
import Arbol23
```

```
{- Definiciones de tipos. -}
```

```
type Comp clave = clave->clave->Bool
type Estr clave valor = Arbol23 (clave,valor) clave
```

```
data Diccionario clave valor = Dicc {cmp :: Comp clave, estructura :: Maybe (Estr clave valor)}
--El comparador es por menor.
```

```
{- Funciones provistas por la cátedra. -}
```

```
--Inserta un nuevo par clave, valor en una estructura que ya tiene al menos un dato.
```

```
insertar::clave->valor->Comp clave->Estr clave valor-> Estr clave valor
```

```
insertar c v comp a23 = interceptar (insertarYPropagar c v comp a23) id (\s1 (c1, s2)->Dos c1 s1 s2)
```

```
--Maneja el caso de que la segunda componente sea Nothing.
```

```
interceptar::(a,Maybe b)->(a->c)->(a->b->c)->c
```

```
interceptar (x,y) f1 f2 = case y of
```

```
    Nothing -> f1 x
```

```
    Just z -> f2 x z
```

```
{- Inserta una clave con su valor correspondiente. Si se actualiza el índice, el cambio se propaga hacia arriba para mantener balanceado el árbol.
```

```
    Usamos recursión explícita porque este tipo de recursión no es estructural (no se aplica a todos los hijos). -}
```

```
insertarYPropagar::clave->valor->Comp clave->Estr clave valor-> (Estr clave valor, Maybe (clave, Estr clave valor))
```

```
insertarYPropagar c v comp a23 = let rec = insertarYPropagar c v comp in case a23 of
```

```
    --Si es hoja, elegimos la máxima de las claves y propagamos el balanceo hacia arriba.
```

```
    Hoja (ch,vh) -> if comp c ch
```

```
        then (Hoja (c,v), Just (ch, Hoja (ch,vh)))
```

```
        else (Hoja (ch, vh), Just (c, Hoja (c,v)))
```

```
{- Si el actual es Nodo-Dos, o se queda en forma Nodo-Dos o se transforma en
```

```
    Nodo-Tres; no puede ocurrir que haya propagación hacia arriba (retornamos Nothing). -}
```

```
Dos c1 a1 a2 -> (if comp c c1
```

```
    then
```

```
        -- La clave c va del lado izquierdo.
```

```
        interceptar (rec a1)
```

```
            (\s1 -> Dos c1 s1 a2)
```

```
            (\s1 (c3, s2) -> Tres c3 c1 s1 s2 a2)
```

```
    else
```

```
        -- La clave c va del lado derecho.
```

```
        interceptar (rec a2)
```

```
            (\s1 -> Dos c1 a1 s1)
```

```
            (\s1 (c3, s2) -> Tres c1 c3 a1 s1 s2), Nothing)
```

```
{- Nodo-tres sólo propaga si de abajo propagan, los tres casos son muy similares
```

```
    Sólo cambia en que árbol se inserta. -}
```

```
Tres c1 c2 a1 a2 a3 -> if comp c c1
```

```

then
  -- La clave debe ir en el primer árbol.
  interceptar (rec a1)
    (\s1 -> (Tres c1 c2 s1 a2 a3, Nothing))
    (\s1 (c3, s2) -> (Dos c3 s1 s2, Just(c1, Dos c2 a2 a3)))
else if comp c c2
then
  -- La clave debe ir en el árbol del medio.
  interceptar (rec a2)
    (\s1 -> (Tres c1 c2 a1 s1 a3, Nothing))
    (\s1 (c3, s2) -> (Dos c1 a1 s1, Just(c3, Dos c2 s2 a3)))
else
  --La clave debe ir en el último árbol.
  interceptar (rec a3)
    (\s1 -> (Tres c1 c2 a1 a2 s1, Nothing))
    (\s1 (c3, s2) -> (Dos c1 a1 a2, Just(c2, Dos c3 s1 s2)))

```

--Se asume que la lista no tiene claves repetidas.

definirVarias::[(clave,valor)]->Diccionario clave valor->Diccionario clave valor

definirVarias = (flip.foldr.uncurry) definir

{- Funciones a implementar. -}

vacio::Comp clave->Diccionario clave valor

vacio c = Dicc c Nothing

definir::clave->valor->Diccionario clave valor->Diccionario clave valor

```

definir c v d = case (estructura d) of
  Nothing -> Dicc (cmp d) (Just (Hoja (c, v)))
  Just arbol -> Dicc (cmp d) (Just (insertar c v (cmp d) (fromJust (estructura d))))

```

obtener::Eq clave=>clave->Diccionario clave valor->Maybe valor

```

obtener c d = case (estructura d) of
  Nothing -> Nothing
  Just arbol -> buscar c (cmp d) arbol

```

{- La función buscar no recorre más de una rama del árbol23 dado.

Esta afirmación se puede asegurar teniendo en cuenta el Orden Normal de Elección de Subexpresiones en el procedimiento de Reducción de Expresiones: tanto la función obtenerADos como obtenerATres se resuelven recorriendo el subárbol correspondiente según el dato buscado, observando los datos alojados en el nodo correspondiente. Pero esto se hace con sentencias del tipo IF-THEN-ELSE, las cuales se resuelven (por el Orden Normal de Haskell) antes que las recursiones a los subárboles, dejando al reducirse la expresión únicamente la llamada recursiva al subárbol por el cual se debe proseguir la búsqueda.

Extendiendo este razonamiento a cada paso recursivo, se llegará a una (y sólo una) hoja del árbol, momento en el cual la función tiene retorno (se halle la clave buscada o no).

-}

buscar::Eq clave=>clave->Comp clave->Arbol23 (clave,valor) clave->Maybe valor

buscar c cmp = foldA23 (obtenerHoja c) (obtenerADos c cmp) (obtenerATres c cmp)

obtenerHoja::Eq clave=>clave->(clave,valor)->Maybe valor

obtenerHoja c1 (c2,v) = if c1==c2 then Just v else Nothing

obtenerADos::Eq clave=>clave->Comp clave->clave->Maybe valor->Maybe valor->Maybe valor

obtenerADos c1 comp c2 a23Izq a23Der = if comp c1 c2 then a23Izq else a23Der

```

obtenerATres::Eq clave=>clave->Comp clave->clave->clave->Maybe valor->Maybe valor->Maybe valor-
>Maybe valor
obtenerATres c1 comp c2 c3 a23Izq a23Med a23Der = if comp c1 c2 then a23Izq else if comp c1 c3 then
a23Med else a23Der

--claves::Diccionario clave valor->[clave]

claves::Diccionario clave valor -> [clave]
claves d = case (estructura d) of
    Nothing -> []
    (Just z) -> clavesAux z

--si el diccionario tiene un arbol no vacio en su estructura, esta funcion es la que realmente
-- devuelve las claves (primera componente de cada hoja)
clavesAux::Arbol23 (clave,valor) clave -> [clave]
clavesAux arb = foldA23 (\x -> [fst x]) (\b x y -> x ++ y) (\b c x y z -> x ++ y ++ z) arb

{- Diccionarios de prueba: -}

arbolitoTest::Arbol23 (Int, Char) Int
arbolitoTest = Tres 0 1
    (Dos 2 (Hoja (7,'a')) (Hoja (3,'b'))))
    (Tres 3 4 (Hoja (54,'c')) (Hoja (65,'d')) (Dos 5 (Hoja (6,'e')) (Hoja (7,'f'))))
    (Dos 6 (Hoja (3,'g')) (Dos 7 (Hoja (4,'h')) (Hoja (5,'i'))))
diceTest=Dicc (<) (Just arbolitoTest)

arbolitoTest2::Arbol23 (Int, Char) Int
arbolitoTest2 = (Tres 5 8 (Hoja (4,'a')) (Hoja (7,'e')) (Dos 10 (Hoja (9,'s')) (Hoja (17,'w'))))
diceTest2=Dicc (<) (Just arbolitoTest2)

dice1::Diccionario Int String
dice1 = definirVarias [(0,"Hola"),(-10,"Chau"),(15,"Felicidades"),(2,"etc."), (9,"a")] (vacio (<))

dice2::Diccionario String String
dice2 = definirVarias [("inicio","casa"),("auto","flores"),("calle","auto"),("casa","escalera"),
    ("ropero","alfajor"),("escalera","ropero")] (vacio (<))

dice3::Diccionario Int String
dice3 = definirVarias [(0,"Hola"),(-10,"Chau"),(15,"Felicidades"),(2,"etc."), (9,"a")] (vacio (\x y->x `mod` 5 <
y `mod` 5))

```