PROGRAMOWANIE FUNKCYJNE HASKELL

The Dance with Trees

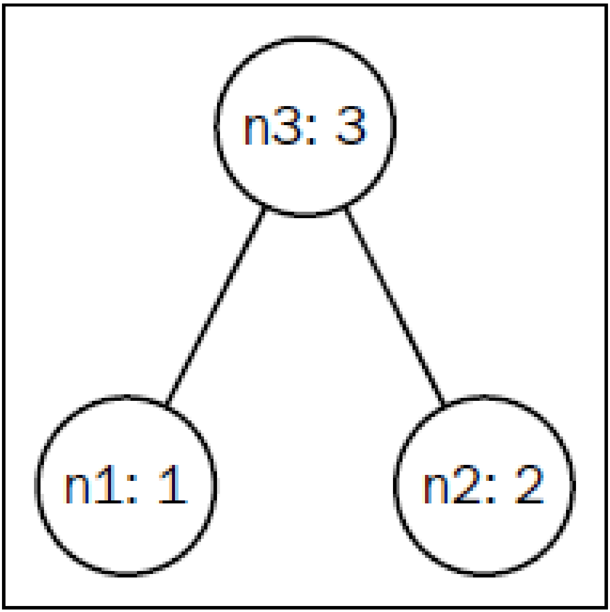
Bartosz Sroczyński, Piotr Szewerniak

Luty 2020

Politechnika Krakowska

PLAN PREZENTACJI

- Defining a binary tree data type
- Defining a rose tree (multiway tree) data type
- Traversing a tree depth-first
- Traversing a tree breadth-first
- Implementing a Foldable instance for a tree
- Calculating the height of a tree
- Implementing a binary search tree data structure
- Verifying the order property of a binary search tree
- Using a self-balancing tree
- Implementing a min-heap data structure
- Encoding a string using a Huffman tree
- Decoding a Huffman code


n3: 3
n1: 1
n2: 2

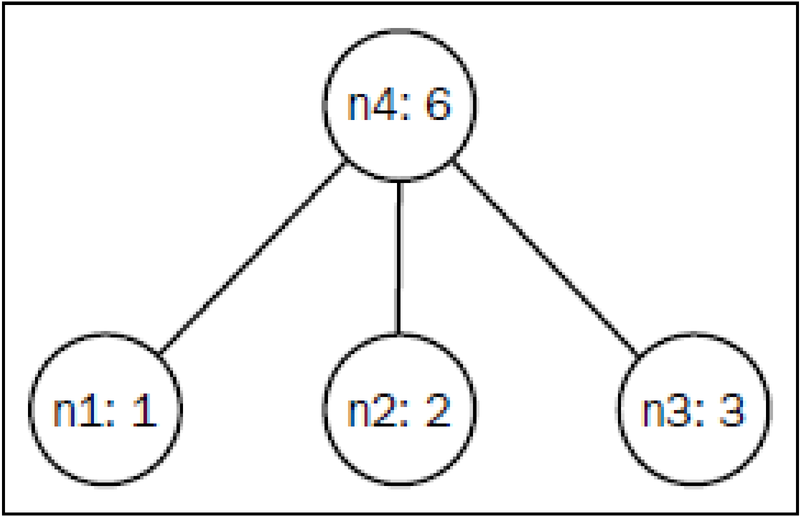
DEFINING A BINARY TREE DATA TYPE

```
data Tree a = Node { value :: a
                   , left  :: (Tree a)
                   , right:: (Tree a) }
            | Leaf
            deriving Show
```

```
main = do
  let n1 = Node { value = 1, left = Leaf, right = Leaf }
  let n2 = Node { value = 2, left = Leaf, right = Leaf }
  let n3 = Node { value = 3, left = n1,   right = n2 }
  print n3
```

```
$ runhaskell Main.hs


Node { value = 3
     , left = Node  { value = 1
                    , left = Leaf
                    , right = Leaf }
     , right = Node { value = 2
                    , left = Leaf
                    , right = Leaf }
     }
```

DEFINING A ROSE TREE (MULTIWAY TREE) DATA TYPE



DEFINING A ROSE TREE (MULTIWAY TREE) DATA TYPE

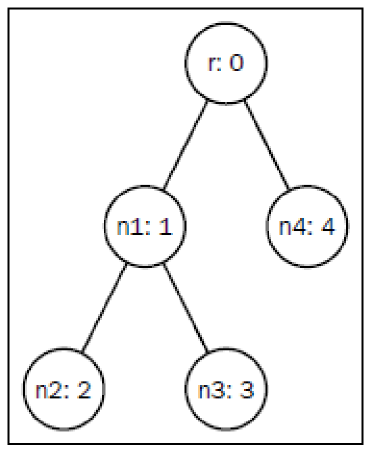
```
data Tree a = Node { value    :: a
                   , children  :: [Tree a] }
                   deriving Show
```

```
main = do
  let n1 = Node { value = 1, children = [] }
  let n2 = Node { value = 2, children = [] }
  let n3 = Node { value = 3, children = [] }
  let n4 = Node { value = 6, children = [n1, n2, n3] }
  print n4
```

DEFINING A ROSE TREE(MULTIWAY TREE) DATA TYPE

```
$ runhaskell Main.hs

Node { value = 6
     , children = [ Node { value = 1
                         , children = [] }
                  , Node { value = 2
                         , children = [] }
                  , Node { value = 3
                         , children = [] } ]
     }
```

TRAVERSING A TREE DEPTH-FIRST



```
import Data.Tree (rootLabel, subForest, Tree(..))
import Data.List (tails)
```

```
depthFirst :: Tree a -> [a]

depthFirst (Node r forest) =
  r : concat [depthFirst t | t <- forest]
```

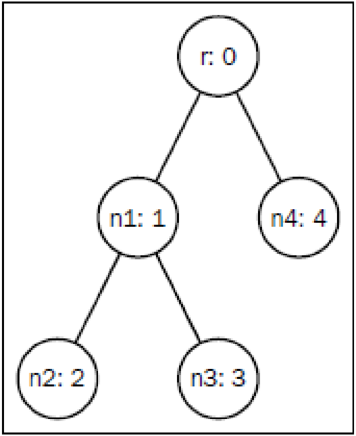
```
add :: Tree Int -> Int

add (Node r forest) = r + sum [add t | t <- forest]
```

```
someTree :: Tree Int

someTree = r
  where r  = Node { rootLabel = 0, subForest = [n1, n4] }
        n1 = Node { rootLabel = 1, subForest = [n2, n3] }
        n2 = Node { rootLabel = 2, subForest = [] }
        n3 = Node { rootLabel = 3, subForest = [] }
        n4 = Node { rootLabel = 4, subForest = [] }
```

```
main = do
  print $ depthFirst someTree
  print $ add someTree
```

```
$ runhaskell Main.hs

[0,1,2,3,4]
10
```

TRAVERSING A TREE BREADTH-FIRST



TRAVERSING A TREE BREADTH-FIRST

```
import Data.Tree (rootLabel, subForest, Tree(..))
import Data.List (tails)
```

```
breadthFirst :: Tree a -> [a]

breadthFirst t = bf [t]
  where bf forest | null forest = []
                  | otherwise   = map rootLabel forest ++
                                    bf (concat (map subForest forest))
```

```
add :: Tree Int -> Int

add t = sum $ breadthFirst t
```

TRAVERSING A TREE BREADTH-FIRST

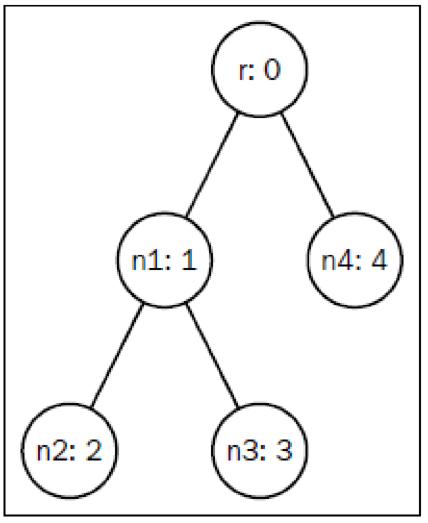
```haskell
someTree :: Tree Int

someTree = root
  where root = Node { rootLabel = 0, subForest = [n1, n4] }
        n1   = Node { rootLabel = 1, subForest = [n2, n3] }
        n2   = Node { rootLabel = 2, subForest = [] }
        n3   = Node { rootLabel = 3, subForest = [] }
        n4   = Node { rootLabel = 4, subForest = [] }
```

```haskell
main = do
  print $ breadthFirst someTree
  print $ add someTree
```

TRAVERSING A TREE BREADTH-FIRST

```
$ runhaskell Main.hs


[0,1,4,2,3]
10
```

IMPLEMENTING A FOLDABLE INSTANCE FOR A TREE



IMPLEMENTING A FOLDABLE INSTANCE FOR A TREE

```
import Data.Monoid (mempty, mappend)
import qualified Data.Foldable as F
import Data.Foldable (Foldable, foldMap)
```

```
data Tree a = Node { value :: a
                   , children :: [Tree a] }
                   deriving Show
```

```
instance Foldable Tree where
  foldMap f Null = mempty
  foldMap f (Node val xs) = foldr mappend (f val)
                             [foldMap f x | x <- xs]
```

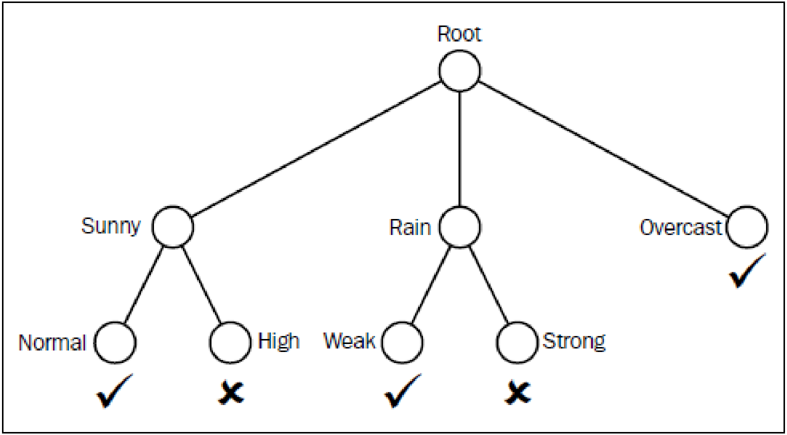
```
add :: Tree Integer -> Integer

add = F.foldr1 (+)
```

```
someTree :: Tree Integer

someTree = root
  where root = Node { value = 0, children = [n1, n4] }
        n1   = Node { value = 1, children = [n2, n3] }
        n2   = Node { value = 2, children = [] }
        n3   = Node { value = 3, children = [] }
        n4   = Node { value = 4, children = [] }
```

```haskell
main :: IO ()
main = print $ add someTree
```

```
$ runhaskell Main.hs


10
```

CALCULATING THE HEIGHT OF A TREE



CALCULATING THE HEIGHT OF A TREE

```
import Data.List (maximum)
import Data.Tree
```

```
height :: Tree a -> Int

height (Node val []) = 1
height (Node val xs) = 1 + maximum (map height xs)
```

```
someTree :: Tree Integer

someTree = root
  where root = 0 [n1, n4]
        n1   = 1 [n2, n3]
        n2   = 2 []
        n3   = 3 []
        n4   = 4 []
```

```
main = print $ height someTree
```

```
$ runhaskell Main.hs


3
```

```
module BSTree (insert, find, single) where
```

```
data Tree a = Node  {value :: a
                    , left   :: (Tree a)
                    , right  :: (Tree a)}
               | Null
               deriving (Eq, Show)
```

```
single :: a -> Tree a

single n = Node n Null Null
```

```
insert :: Ord a => Tree a -> a -> Tree a

insert (Node v l r) v'
    | v' < v        = Node v (insert l v') r
    | v' > v        = Node v l (insert r v')
    | otherwise     = Node v l r

  insert _ v' = Node v' Null Null
```

IMPLEMENTING A BINARY SEARCH TREE DATA STRUCTURE

```
find :: Ord a => Tree a -> a -> Bool

find (Node v l r) v'
  | v' < v        = find l v'
  | v' > v        = find r v'
  | otherwise     = True

find Null v' = False
```

```
import BSTree
```

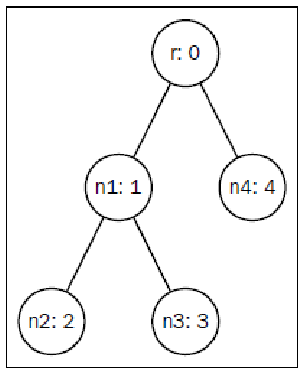
```
main = do
  let tree = single 5
  let nodes = [6,4,8,2,9]
  let bst = foldl insert tree nodes
```

```
print bst
print $ find bst 1
print $ find bst 2
```

```
$ runhaskell Main.hs

Node { value = 5
     , left = Node { value = 4
                   , left = Node { value = 2
                                 , left = Null
                                 , right = Null }
                   , right = Null }
     , right = Node { value = 6
```

```
                         , left = Null
                         , right = Node { value = 8
                                        , left = Null
                                        , right = Node { value = 9
                                                       , left = Null
                                                       , right = Null }
                                        }
                         }
        }

   False

   True
```

VERIFYING THE ORDER PROPERTY OF A BINARY SEARCH TREE



```
data Tree a = Node { value  :: a
                   , left   :: (Tree a)
                   , right  :: (Tree a)}
            | Null
     deriving (Eq, Show)
```

```
someTree :: Tree Int

someTree = root
  where root = Node 0 n1 n4
        n1   = Node 1 n2 n3
        n2   = Node 2 Null Null
        n3   = Node 3 Null Null
        n4   = Node 4 Null Null
```

VERIFYING THE ORDER PROPERTY OF A BINARY SEARCH TREE

```
valid :: Ord t => Tree t -> Bool

valid (Node v l r) = leftValid && rightValid
  where leftValid  = if notNull l
```

```
                        then valid l && value l <= v
                        else True
        rightValid = if notNull r
                        then valid r && v <= value r
                        else True
        notNull t  =  t /= Null
```

VERIFYING THE ORDER PROPERTY OF A BINARY SEARCH TREE

```
main = print $ valid someTree
```

```
$ runhaskell Main.hs

False
```

We will be using the `AvlTree` package to use `Data.Tree.AVL`:

```
$ cabal install AvlTree
```

```
import Data.Tree.AVL
import Data.COrdering
```

USING A SELF-BALANCING TREE

```
main = do
  let avl  = asTree fstCC [4,2,1,5,3,6]
  let min  = tryReadL avl
  let max  = tryReadR avl
  print min
  print max
```

```
$ runhaskell Main.hs


Just 1
Just 6
```

```
$ cabal install lens
```

```haskell
module MinHeap (empty, insert, deleteMin, weights) where

import Control.Lens (element, set)
import Data.Maybe (isJust, fromJust)
```

```
data Heap v = Heap { items :: [Node v] }
              deriving Show

data Node v = Node { value :: v, weight :: Int }
              deriving Show
```

```
empty = Heap []
```

```
insert v w (Heap xs) = percolateUp position items'
  where items'    = xs ++ [Node v w]
        position = length items' - 1
```

```
deleteMin (Heap xs) = percolateDown 1 items'
  where items' = set (element 1) (last xs) (init xs)
```

```
viewMin heap@(Heap (_:y:_)) =
  Just (value y, weight y, deleteMin heap)
viewMin _                   = Nothing
```

```
percolateDown i items
  | isJust left && isJust right = percolateDown i'
                                      (swap i i' items)
  | isJust left = percolateDown l (swap i l items)
  | otherwise = Heap items
```

```
where left   = if l >= length items
                 then Nothing
                 else Just $ items !! l
      right  = if r >= length items
                 then Nothing
                 else Just $ items !! r
      i'     = if (weight (fromJust left)) <
                    (weight (fromJust right))
               then l else r
      l      = 2*i
      r      = 2*i + 1
```

```
percolateUp i items
  | i == 1 = Heap items
  | w < w' = percolateUp c (swap i c items)
  | otherwise = Heap items
   where  w  = weight $ items !! i
          w' = weight $ items !! c
          c  = i `div` 2
```

```
swap i j xs = set (element j) vi (set (element i) vj xs)
  where vi = xs !! i
        vj = xs !! j
```

```
weights heap = map weight ((tail.items) heap)
```

```
import MinHeap

main = do
  let heap = foldr (\x -> insert x x)
                empty [11, 5, 3, 4, 8]
  print $ weights heap
  print $ weights $ iterate deleteMin heap !! 1
  print $ weights $ iterate deleteMin heap !! 2
  print $ weights $ iterate deleteMin heap !! 3
  print $ weights $ iterate deleteMin heap !! 4
```

```
$ runhaskell Main.hs


[3,5,4,8,11]

[4,5,11,8]

[5,8,11]

[8,11]

[11]
```

ENCODING A STRING USING A HUFFMAN TREE

```
import Data.List (group, sort)
import MinHeap
import Network.HTTP ( getRequest, getResponseBody
                    , simpleHTTP )
import Data.Char (isAscii)
import Data.Maybe (fromJust)
import Data.Map (fromList, (!))
```

```
freq xs = map (\x -> (head x, length x))
            group . sort $ xs
```

```
data HTree  =  HTree { value :: Char
                     , left  :: HTree
                     , right :: HTree }
            | Null
            deriving (Eq, Show)
```

```
single v = HTree v Null Null
```

```
htree heap = if length (items heap) == 2
             then case fromJust (viewMin heap) of
                    (a,b,c) -> a
             else htree $ insert newNode (w1 + w2) heap3

  where (min1, w1, heap2) = fromJust $ viewMin heap
        (min2, w2, heap3) = fromJust $ viewMin heap2
        newNode           = HTree { value  = ' '
                                  , left   = min1
                                  , right  = min2 }
```

ENCODING A STRING USING A HUFFMAN TREE

```
codes htree = codes' htree ""

  where codes' (HTree v l r) str
             | l==Null && r==Null = [(v, str)]
             | r==Null            = leftCodes
             | l==Null            = rightCodes
             | otherwise          = leftCodes ++ rightCodes
             where  leftCodes  = codes' l ('0':str)
                    rightCodes = codes' r ('1':str)
```

```
encode str m = concat $ map (m !) str
```

ENCODING A STRING USING A HUFFMAN TREE

```
main = do
  rsp <- simpleHTTP (getRequest
           "http://norvig.com/big.txt")
  html <- fmap (takeWhile isAscii) (getResponseBody rsp)
  let freqs = freq html
  let heap = foldr (\(v,w) -> insert (single v) w)
               empty freqs
  let m = fromList $ codes $ htree heap
  print $ encode "hello world" m
```

```
$ runhaskell Main.hs

"01000111001111011111000101110100010001101101111110010"
```

DECODING A HUFFMAN CODE

```
decode :: String -> HTree -> String
decode str htree = decode' str htree
  where  decode' "" _ = ""
      decode' ('0':str) (HTree _ l _)
        | leaf l     = value l : decode' str htree
        | otherwise = decode' str l
      decode' ('1':str) (HTree v _ r)
        | leaf r     = value r : decode' str htree
        | otherwise = decode' str r
      leaf tree = left tree == Null && right tree == Null
```

Dziękujemy za uwagę!