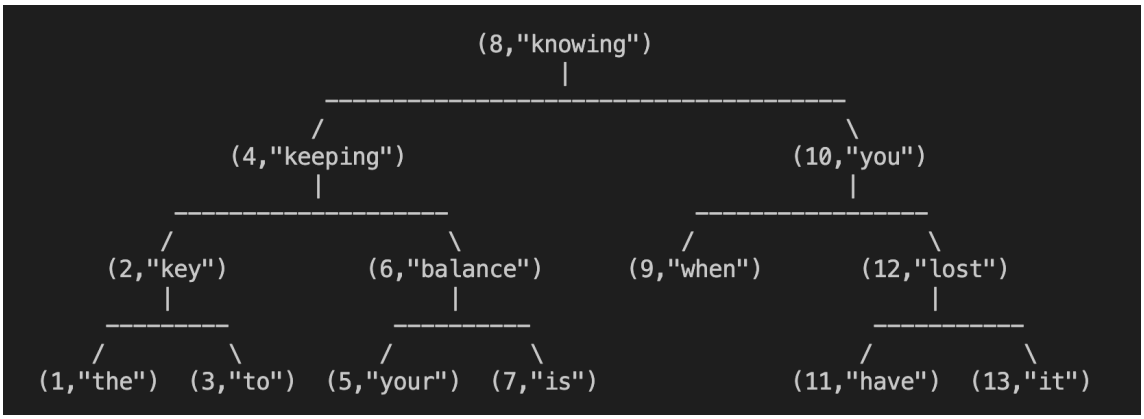
Homework assignment #3

Functional programming's emphasis on data immutability, pure functions, and recursion makes it a natural fit for implementing self-balancing trees. AVL trees (named after inventors Adelson-Velsky and Landis) were the first of such data structure to be invented, providing efficient search, insertion, and deletion operations with a worst-case time complexity of `O(log n)`, where `n` is the number of nodes in the tree. This makes them useful in many applications that require efficient searching.



Self-balancing trees should be fast

In class, we've implemented `search`, `insert` and `delete` functions over self-balancing trees. However, we implemented function `height` in a very naive way and `insert` does not run in `O(log n)`. The goal of this assignment is to improve the code for better performance.

Function `height` is an expensive recursive function that completes a tree traversal of the node and its sub-trees. A better way to implement a AVL tree is to store the height of each node and to update it every time a new node is inserted or deleted.

In GHCI, use command `:set +s` to check the runtime necessary to evaluate an expression. Try evaluating the expression `inorder $ foldl (flip insert) Leaf [1..5000]`.

Tasks:

1. Function `delete` does not preserve the balance property of an AVL tree. Your first task is to fix this.
2. Function `height` is very inefficent and requires a complete tree traversal. After fixing the code, you should be able to insert `5000` nodes into the self-balancing tree in less than `1` second.
   - Update data type `BTree a` to also store the height of each node, i.e. `data BTree a = Leaf | Node (BTree a) (BTree a) a Int`.
   - Update functions throughout to take into account the updated type. Do **not** change the type signature provided.
   - Make all changes necessary to ensure that the height stored with each node is correct.
   - Make function `height` run in `O(1)`.
3. Make `BTree a` an instance of typeclass `Eq` manually, without deriving it.
4. Write a function `isAVL` that determines whether a `BTree` is AVL.
5. (**extra credits**) The reference code of this homework was produced in class. Extra points will be given to students finding bugs or edge cases that might have been overlooked.

Reference code:

In [ ]:

```haskell
module AVLTree where

import qualified Data.Tree as T (Tree(..), drawTree)
import Data.Tree.Pretty (drawVerticalTree)

{-
data Tree a = Node {
        rootLabel :: a,         -- ^ label value
        subForest :: Forest a   -- ^ zero or more child trees
    }

type Forest a = [Tree a]
-}

data BTree a = Leaf | Node (BTree a) (BTree a) a deriving (Eq)

-- make BTree an instance of typeclass Show
instance Show a => Show (BTree a) where
    show :: BTree a -> String
    show tree = drawVerticalTree $ convertTree tree

-- search an element in a BTree
search :: Ord a => a -> BTree a -> Bool
search _ Leaf = False
search x (Node left right y) | x == y    = True
                             | x < y     = search x left
                             | otherwise = search x right

-- insert *without* rebalancing
insert :: Ord a => a -> BTree a -> BTree a
insert x Leaf = Node Leaf Leaf x
insert x t@(Node left right y) | x > y = Node left (insert x right) y
                               | x < y = Node (insert x left) right y
                               | otherwise = t

-- delete *without* rebalancing
delete :: Ord a => a -> BTree a -> BTree a
delete _ Leaf                = Leaf
delete x (Node l r y)
    | x > y                  = Node l (delete x r) y
    | x < y                  = Node (delete x l) r y
    | l == Leaf && r == Leaf = Leaf -- from this line and below: x == y
    | l == Leaf              = r
    | r == Leaf              = l
    | otherwise              = let z = leftmost r
                               in Node l (delete z r) (leftmost r)
                               -- we replace y with z, and delete z from right banch
  where
    leftmost :: BTree a -> a
    leftmost Leaf = error "Tree is empty. We shouldn't get here!"
    leftmost (Node Leaf _ y) = y
    leftmost (Node left _ _) = leftmost left

-- inorder traversal
inorder :: BTree a -> [a]
inorder Leaf = []
inorder (Node left right x) = inorder left ++ [x] ++ inorder right

-- convert BTree a to T.Tree String
convertTree :: Show a => BTree a -> T.Tree String
convertTree Leaf = T.Node { T.rootLabel = ".", T.subForest = []}
convertTree (Node left right x) =
    T.Node { T.rootLabel = show x
           , T.subForest = [convertTree left, convertTree right]
           }
```

```haskell
-- auxiliary functions to get height and balance factor of a node
-- height
height :: BTree a -> Int
height Leaf = 0
height (Node l r _) = 1 + max (height l) (height r)

-- balance factor
bf :: BTree a -> Int
bf Leaf = 0
bf (Node l r _) = height l - height r

-- rotations preserving order
rotateLeft :: BTree a -> BTree a
rotateLeft (Node t1 (Node t2 t3 y) x) = Node (Node t1 t2 x) t3 y
rotateLeft t = t

rotateRight :: BTree a -> BTree a
rotateRight (Node (Node t1 t2 x) t3 y) = Node t1 (Node t2 t3 y) x
rotateRight t = t

-- rebalance: if the tree is balanced,
-- adding or removing a single node can only slightly unbalance it
rebalance :: BTree a -> BTree a
rebalance Leaf = Leaf
rebalance t@(Node l r x)
    | bf t == 2  && bf l == -1 = rotateRight $ Node (rotateLeft l) r x
    | bf t == 2  && bf l == 0  = rotateRight t
    | bf t == 2  && bf l == 1  = rotateRight t
    | bf t == -2 && bf r == -1 = rotateLeft t
    | bf t == -2 && bf r == 0  = rotateLeft t
    | bf t == -2 && bf r == 1  = rotateLeft $ Node l (rotateRight r) x
    | otherwise                = t

-- insert preserving balance
insert' :: Ord a => a -> BTree a -> BTree a
insert' x Leaf = Node Leaf Leaf x
insert' x t@(Node left right y)
    | x > y = rebalance $ Node left (insert' x right) y
    | x < y = rebalance $ Node (insert' x left) right y
    | otherwise = t

-- delete preserving balance
-- TO DO
delete' :: Ord a => a -> BTree a -> BTree a
delete' = undefined

-- examples of BTree
t1 :: BTree Int
t1 = foldl (flip insert) Leaf [1,5,10,3,4,2,6]

t2 :: BTree Int
t2 = foldl (flip insert) Leaf [1..5]

t3 :: BTree Int
t3 = foldl (flip insert') Leaf [1..5] -- insert' preserves balance
```

Good luck! 👾