# 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.

```
                              (8,"knowing")
                                    |
                ------------------------------------------
                /                                         \
          (4,"keeping")                                (10,"you")
                |                                           |
      --------------------                         ----------------
      /                  \                         /              \
  (2,"key")          (6,"balance")           (9,"when")       (12,"lost")
      |                   |                                        |
  ---------          ----------                               -----------
  /       \          /        \                               /          \
(1,"the")  (3,"to")  (5,"your")  (7,"is")               (11,"have")  (13,"it")
```

## 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.
    - Update functions throughout to take into account the updated type.
    - 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 [ ]:
```
module BTree where

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

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

-- pretty print BTree
instance Show a => Show (BTree a) where
    show :: Show a => BTree a -> String
    show = drawVerticalTree . treeconverter

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

-- in its current form, delete does not preserve balance
delete :: Ord a => a -> BTree a -> BTree a
delete a Leaf = error "Element not in the tree"
delete a (Node b left right) | a > b = Node b left (delete a right)
                             | a < b = Node b (delete a left) right
                             | left == Leaf && right == Leaf = Leaf
                             | left == Leaf = right
                             | right == Leaf = left
                             | otherwise = let z = rightmost left
                                           in Node z (delete z left) right
  where
    rightmost :: BTree a -> a
    rightmost Leaf = error "Tree is empty. There's no rightmost node."
    rightmost (Node x Leaf Leaf) = x
    rightmost (Node _ _ right) = rightmost right

-- examples
t1 :: BTree Int
t1 = foldl (flip insert) Leaf [5,4,3,7,1,10,8]

t2 :: BTree Int
t2 = foldl (flip insert) Leaf [1..10]

-- tree traversals
inorder :: BTree a -> [a]
inorder Leaf = []
inorder (Node val left right)= inorder left ++ [val] ++ inorder right

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

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

-- conver BTree T.Tree
treeconverter :: Show a => BTree a -> T.Tree String
treeconverter Leaf = T.Node {T.rootLabel = ".", T.subForest =[]}
treeconverter (Node nodeName tL tR) = T.Node {T.rootLabel = (show nodeName), T.subForest = [(treeconve

-- height & balance factor
height :: BTree a -> Int
height Leaf = 0
height (Node v l r) = 1 + max (height l) (height r)

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

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

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

-- rebalance tree
rebalance :: BTree a -> BTree a
rebalance t@(Node x left right)
    | balanceFactor t == 2  && balanceFactor left  == -1 = rotateRight $ Node x (rotateLeft left) righ
    | balanceFactor t == 2  && balanceFactor left  == 0  = rotateRight t
    | balanceFactor t == 2  && balanceFactor left  == 1  = rotateRight t
    | balanceFactor t == -2 && balanceFactor right == -1 = rotateLeft t
    | balanceFactor t == -2 && balanceFactor right == 0  = rotateLeft t
    | balanceFactor t == -2 && balanceFactor right == 1  = rotateLeft $ Node x left (rotateRight right
    | otherwise = t

-- insert preserving balance
insert :: Ord a => a -> BTree a -> BTree a
insert x Leaf = Node x Leaf Leaf
insert x t@(Node y left right) | x > y = rebalance $ Node y left (insert x right)
                               | x < y = rebalance $ Node y (insert x left) right
                               | otherwise = t
```

Good luck! 👾