```haskell
-- Copyright 2002 Manuel M T Chakravarty
--

module Lab13
where

-- 1nd Exercise ----------------------------------------------------------

-- NOTE: You need to put the definition for AVL trees into a separate module
--       to execute them, as the data constructors of Tree and AVLTree collide.

type Balance = Int

data AVLTree a = Node Balance a (AVLTree a) (AVLTree a)
               | Leaf
               deriving(Show)

-- Questions
-- ==========
--
--  depth = n -> 2^n - 1 elements max, n min
--
--  depth = 3 = 2^3 - 1 = 8 elems max, 4 min
--
--  AVL trees do not 'degenerate'. Since they are
--  almost balanced, searching, inserting and also
--  deleting are always O(log n) (for n elements)
--

-- Returns 'True' iff a tree is a ordered AVL tree
checkAVL:: Ord a => AVLTree a -> Bool
checkAVL t = (checkBalance t) && (isSortedAVL t)

-- checks if balance if correct and in the AVL range
checkBalance Leaf = True
checkBalance (Node b _ t1 t2) =
    inRange           &&
    correctBalance    &&
    (checkBalance t1) &&
    (checkBalance t2)
    where
      inRange        = ((-2) < b) && (b < 2)
      correctBalance =  b == (depth t1 - depth t2)

-- checks whether the node values are sorted,
-- similar to lasts weeks definition
isSortedAVL t = listIsOrdered (flatten t)
  where
    listIsOrdered []  = True
    listIsOrdered [x] = True
    listIsOrdered (x1:x2:xs) = (x1 < x2) && listIsOrdered (x2:xs)

flatten Leaf = []
flatten (Node _ x t1 t2) =
    (flatten t1) ++ [x] ++ (flatten t2)

depth :: AVLTree a -> Int
depth Leaf = 0
depth (Node _ x t1 t2) =
  1 + max (depth t1) (depth t2)

-- 2nd Exercise ----------------------------------------------------------

data RTree a = NodeRT a [RTree a]

mapRTree :: (a -> b) -> RTree a -> RTree b
mapRTree f (NodeRT x trees) = NodeRT (f x) (map (mapRTree f) trees)
```