Functional Programming | WS 2023/2024 | LVA 703025

Exercise Sheet 4, 5 points | Deadline: Tuesday, November 7, 2023, 8pm

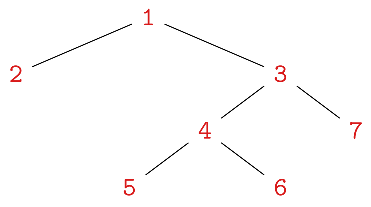
- Mark your completed exercises in the OLAT course of the PS.
- You can use a template .hs file that is provided on the proseminar page.
- Upload your modified .hs file in OLAT.
- Your .hs file must be compilable with ghci.

Exercise 1 *Polymorphism and Trees* **5 p.**

In this exercise we will consider a type for representing binary trees. To this end we consider the datatype `Tree` defined as

```haskell
data Tree a = Node a (Tree a) (Tree a) | Leaf a
```

For instance, `exampleTree` represents the following tree:



```haskell
exampleTree = Node 1 (Leaf 2) (Node 3 (Node 4 (Leaf 5) (Leaf 6)) (Leaf 7))
```

1. Write a function `height :: Tree a -> Integer` that calculates the height of a binary tree. The height is the number of edges on the longest path between the root and a leaf. (1 point)
   Hint: the Haskell function `max :: Ord a => a -> a -> a` *might be useful.*

   Examples:
   ```haskell
   height (Leaf 'a') == 0
   height exampleTree == 3
   ```

2. Write a function `flatten :: Tree a -> [a]` which takes a tree as an argument and returns a list containing exactly the elements in the tree from left to right. In particular, each node element should appear in the list after the elements in its left subtree and before the elements in its right subtree. (1 point)
   Hint: `(++) :: [a] -> [a] -> [a]` *is Haskell's predefined append-function for lists.*

   Examples:
   ```haskell
   flatten (Node 1 (Leaf 2) (Leaf 2)) == [2,1,2]
   flatten exampleTree == [2,1,5,4,6,3,7]
   ```

3. A binary tree `t` is said to be a binary search tree if `flatten t` is a list whose elements appear in strictly increasing order. Write a function `isSearchTree:: Ord a => Tree a -> Bool` that takes a tree as argument and returns `True` if and only if the tree is a binary search tree. (1 point)
   Hint: you may assume that `flatten` *is available even if you did not solve question 2. It might be useful to define an auxiliary function* `isStrictlySorted :: Ord a => [a] -> Bool` *to determine whether the elements in a list are strictly increasing.*

Examples:

```
isSearchTree (Leaf "hello") == True
isSearchTree exampleTree == False
isSearchTree (Node 3 (Leaf 1) (Node 6 (Leaf 4) (Leaf 11))) == True
```

4. Write a function `elemDepth :: Eq a => a -> Tree a -> Maybe Integer` which determines whether an element is in a tree. If the element is in the tree and `d` is the minimum depth at which the element appears, then `Just d` should be returned. Otherwise, `Nothing` should be returned.
   Is the restriction `Eq a` necessary? (2 points)
   Hint: it might be useful to define an auxiliary function of type `Maybe a -> Maybe a -> Maybe a` *to process results from recursive calls of* `elemDepth` *(see also slide 04/20).*
   The Haskell function `min :: Ord a => a -> a -> a` *might also be useful.*

   Examples:

```
elemDepth 7 exampleTree == Just 2
elemDepth 15 exampleTree == Nothing
elemDepth 'b' (Node 'a' (Leaf 'b') (Node 'c' (Leaf 'b') (Leaf 'd'))) == Just 1
```

Solution 1

```
-- Question 1
height :: Tree a -> Integer
height (Leaf _) = 0
height (Node _ t1 t2) = 1 + max (height t1) (height t2)

-- Question 2
flatten :: Tree a -> [a]
flatten (Leaf x) = [x]
flatten (Node x t1 t2) = flatten t1 ++ [x] ++ flatten t2

-- Question 3
isStrictlySorted :: Ord a => [a] -> Bool
isStrictlySorted (x : y : ys) = x < y && isStrictlySorted (y : ys)
isStrictlySorted _ = True

isSearchTree :: Ord a => Tree a -> Bool
isSearchTree t = isStrictlySorted (flatten t)

-- Question 4
elemDepthAux :: (Num a, Ord a) => Maybe a -> Maybe a -> Maybe a
elemDepthAux (Just x) (Just y) = Just (1 + min x y)
elemDepthAux (Just x) Nothing = Just (1 + x)
elemDepthAux Nothing (Just y) = Just (1 + y)
elemDepthAux _ _ = Nothing

elemDepth :: Eq a => a -> Tree a -> Maybe Integer
elemDepth x (Leaf y) = if x == y then Just 0 else Nothing
elemDepth x (Node y t1 t2) = if x == y then Just 0 else elemDepthAux (elemDepth x t1) (elemDepth x t2)
```