universität innsbruck

Functional Programming	WS 2022	LVA 703025
Exercise Sheet 9, 10 points		Deadline: Wednesday, December 7, 2022, 6am

- Mark your completed exercises in the OLAT course of the PS.
- Start from `template_09.tgz` provided on the proseminar page.
- All your .hs-files should be compilable with `ghci` and be uploaded in OLAT. Please upload each .hs-file separately, and not an archive such as .zip, .tgz, .rar, etc.

Exercise 1 *Sets* **6 p.**

In programming one often has to store *sets* of elements, i.e., mathematical sets where $\{1,3\} = \{1,1,3\}$ since no duplicates are counted.
There are several ways to implement sets and in this exercise we consider two variants.

1. One possible implementation represents sets by *distinct lists*, i.e., lists where no element occurs more than once. For example, both `[1,3]` and `[3,1]` are valid representations of the set $\{1,3\}$, but `[1,1,3]` is not.

   Consider the following Haskell code in file `DList.hs` that implements sets as distinct lists. Here, the operations `empty`, `member`, `insert`, and `remove` correspond to the empty set, the membership test, insertion of a single element, and removal of a single element. For the latter, note that $A \setminus \{a\} = A$ if $a \notin A$.

   ```
   data DList a = DList [a]

   empty = DList []

   member x (DList xs) = x `elem` xs

   insert x a@(DList xs)
     | member x a = a
     | otherwise = DList $ x : xs

   remove x a@(DList xs) = case span (/= x) xs of
     (_, []) -> a
     (first, _ : last) -> DList $ first ++ last
   ```

   Identify those parts of the implementation that rely upon the lists being distinct, i.e., which can return wrong results on lists with duplicates.

   Furthermore, add a sensible module declaration such that an external user can access all set operations, but cannot create elements of type `DList a` which violate the distinctness condition. (1 point)

2. Make `DList` an instance of `Show`. Examples:

   ```
   show $ foldr insert empty [1..5] = "{1, 2, 3, 4, 5}"
   show $ foldr insert empty "abcba" = "{'c', 'b', 'a'}"
   show $ foldr insert empty [] = "{}"
   ```

   Your implementation must be based on a variant of `fold`, i.e., no explicit recursion is allowed. (1 point)

3. Make `DList` an instance of `Eq`. Example:

   ```
   foldr insert empty ([1..5] ++ [2..4]) == foldr insert empty [5,4..1]
   ```

   Try to exploit the distinctness condition in your implementation. (1 point)

4. Provide an alternative implementation of sets, which are now implemented as *ordered lists*, i.e., lists $[x_1, \ldots, x_n]$ such that $x_i < x_{i+1}$ for all $1 \leq i < n$. To this end, copy the current implementation into a new file `OList.hs` for ordered lists, replace the name `DList` by `OList`, and adjust the implementation so that it now takes the new invariant of being ordered into account. (2 points)

   Hint: Some operations get more complex, some can stay as they are, and some are getting simpler.

5. Write a file `Application.hs`. It should import both set implementations and contain a function `sanityCheck` of type `Int -> Bool`. The latter takes a parameter `n` and checks that

   ```
   foldr insert empty ([1..n] ++ [1..n]) == foldr insert empty (reverse [1..n])
   ```

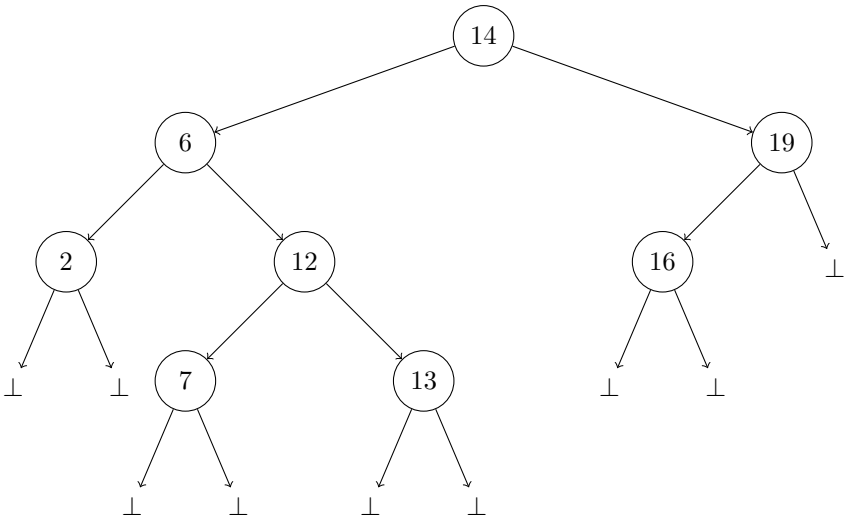
   is satisfied *for both set implementations*. (1 point)

Exercise 2 *Binary Trees* **4 p.**

In this exercise we will be implementing an (unbalanced) binary search tree. A binary search tree is a tree with the following properties:

- A tree is either empty ($\perp$ in the example diagram) or a node (a circle in the example diagram.)
- Each node holds exactly one value
- Each node has exactly two sub-trees.
- Each value in the left sub-tree of a node is smaller than the value in the node.
- Each value in the right sub-tree of a node is greater than the value in the node.

Example:



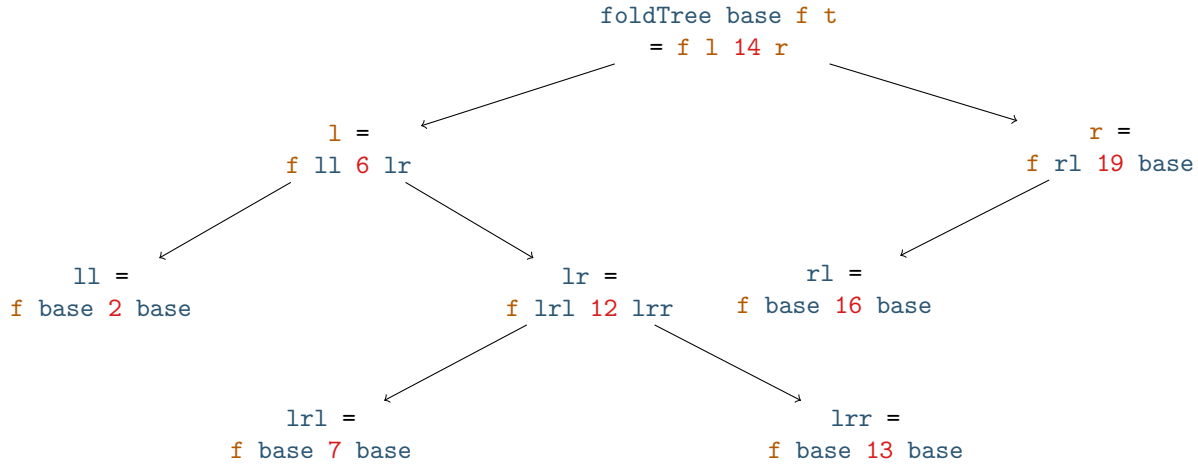
For this exercise, create a new Haskell file `BinarySearchTree.hs` and add `module BinarySearchTree where` at the top. If you want, try to modify this statement so that it only exposes necessary components, similarly to Exercise 1.1. The file `BinarySearchTreeTests.hs` contains tests for tasks 2–4. Note that for the tests to compile and work, your datatype needs to be an instance of `Eq` (deriving is enough, no need for a manual implementation), and you need to at least add the function declarations and stubs (`fun :: Type` and `fun = undefined`) to your Haskell file.

1. Define a recursive data type `BinaryTree a` to represent binary trees. Remember: a node stores one element and has exactly two sub-trees. Additionally implement a function `empty :: BinaryTree a` which returns an empty binary tree. (1 point)
   Reminder: `BinaryTree a` needs to derive `Eq` for the tests to compile. Deriving `Show` may also be useful for debugging.

2. Implement a function `insert :: Ord a => a -> BinaryTree a -> BinaryTree a`, which inserts an element into a binary search tree in such a way, that the properties of the binary search tree are conserved. If the element is already contained in the tree, the tree is not changed. (1 point)
   Examples (with the binary search tree from above as starting point):

- Inserting 12 does not change the tree
- Inserting 5 adds it as the right sub-tree of the node containing 2
- Inserting 15 adds it as the left sub-tree of the node containing 16
- The example tree could have been obtained via `foldr insert empty [6,7,13,7,12,2,16,19,6,14]`.

3. Implement a function `foldTree :: b -> (b -> a -> b -> b) -> BinaryTree a -> b`. If `t` represents the example tree from above, `foldTree base f t` is evaluated as follows: (1 point)



4. Use `foldTree` from the previous task to implement the functions `height :: BinaryTree a -> Int`, and `isMember :: Ord a => a -> BinaryTree a -> Bool`.
The height of a tree is the number of edges on the longest path from the root to any node. Note that an empty tree has height -1.
`isMember` returns `True` if the given element is contained in the tree, and `False` otherwise. (1 point)