Prof. Dr. Peter Thiemann

Hannes Saffrich
saffrich@informatik.uni-freiburg.de

Winter Term 2024/25

**Functional Programming**
https://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2024/

# Exercise Sheet 5

**Exercise 1** (AVL Trees & Quickcheck)

In this exercise, we are implementing AVL search trees, i.e. binary search trees, where the `insert` operation ensure that the tree remains balanced, which allows it to have a `O(log(N))` runtime.

As the rebalancing operations are a bit more complicated, we are using `QuickCheck` to test for correctness.

1. Make yourself familiar with the basic ideas of how AVL trees work by searching the internet. The precise details are only relevant for the last subexercise.
2. We represent AVL trees similarly as we have represented binary trees in the lecture:

   ```
   data AVLTree a = Leaf | Branch Int (AVLTree a) a (AVLTree a)
   ```

   In contrast to the lecture, a `Branch` has an additional `Int` field, which represents the height of the branch. This allows computing the height of a tree in constant time, which in turn allows to implement the rebalancing `insert` function in `O(log(N))` time by using this cached height instead of recomputing it for all subtrees.

   Implement an `Arbitrary` instance to randomly generate AVL trees, i.e. `AVLTree`s, which are balanced, ordered, have unique elements, and correct height annotations.

   Define `QuickCheck` properties, which test that the generated trees satisfy the above mentioned properties.

   Hint: to randomly generate AVL trees, we recommend to first generate sorted lists with unique elements, and then repeatedly split them to generate an AVL tree.

   Hint: make sure that the balancing is also decided randomly, e.g. that for the list `[1,2]` it is possible to generate both

   - `Branch Leaf 1 (Branch Leaf 2 Leaf)`; and
   - `Branch (Branch Leaf 1 Leaf) 2 Leaf`

   For this purpose you might want to flip a coin by using the `Arbitrary` instance for `Bool`.

   Hint: You can think of `Gen a` very much like `IO a`: it describes some kind of computation, which when run will produce a value of type `a`, and you can use `do`-notation, `(>>=)`, and `return` to combine such computations.
3. In this exercise, we use a test-driven development approach, i.e. we are defining tests before we actually implement the functions for AVL trees.

   To keep our code typechecking during this phase, we define stubs for the functions on AVL trees, i.e. we specify their type signatures, but set their bodies to `undefined`.

   Generate property tests for the following function stubs:

```haskell
insert :: Ord a => a -> AVLTree a -> AVLTree a
insert = undefined

contains :: Ord a => a -> AVLTree a -> Bool
contains = undefined

merge :: Ord a => AVLTree a -> AVLTree a -> AVLTree a
merge = undefined

toList :: AVLTree a -> [a]
toList = undefined

-- Assumes the list is sorted and all elements are unique.
fromList :: [a] -> AVLTree a
fromList = undefined
```

Hint: Some examples of useful properties are:

a) returned `AVLTree`s should satisfy the `AVLTree` properties, i.e. they should be balanced, ordered, have unique elements, and correct height annotations;

b) some of those functions interact in a predictable way, e.g. after inserting an element, the tree should contain the element;

c) some of those functions are (partial) inverses of each other, e.g. converting a sorted list with unique elements to a tree and back should yield the same list.

4. Implement the function stubs from the previous exercise.