120: Exercises for Finger Trees

The following exercises should help with understanding the paper of Hinze and Paterson on finger trees. All are supposed to be solved with pencil on paper (no programming).

These exercises **should not** be handed in. They are supposed to help you with understanding the paper, notation, etc. to make the implementation of Finger Trees easier. You may skip these exercises if you were comfortable with the notation.

Afterwards implement Finger Trees in Scala to deepen your understanding of how the data structure works, and to prepare better for the exam. Skeleton code is provide in the course repository to help you start faster.

Exercise 1. (somewhat trivial; skip if you know what a deque is)

Assume that we have a simple double-ended queue (a deque). We will make one diagram per each point below.

1. Draw how the dequeue looks after adding 1 on the left, and then adding 2 on the left, and then adding 3 on the left.
2. Now take the dequeue from the previous point and draw how it looks after adding 4 on the **right**, and then after adding 5 on the **right**.
3. Now take the dequeue resulting from the previous point and draw how it looks after popping two elements from the **left**, and one element from the **right**.

Exercise 2. Assume that we have an empty finger tree ready to store integer numbers. We perform the following operations on the tree. Draw how the tree looks after the series of operations in each point.

1. We add number 1, and then number 2 (both from the left).
2. Then add numbers 3, 4 and 5 from the left (in this order)
3. Then add number 6 from left
4. Then add numbers 7, 8 and 9 from the **left**.
5. Then add number 10 from the **right**.
6. Then remove a number from the **right** twice.
7. Then remove a number from the **right**.

Exercise 3. This exercise uses Haskell notation to stay close to the paper. So `(+)` denotes the binary plus operator and `(:)` denotes the cons operator for lists (in Scala denoted using double colon `(_::_)` or `Cons(_,_)`). Square brackets in Haskell are used to denote list literals, with empty square brackets representing the empty list (`Nil` or `List()` in Scala). The reduce right and reduce left functions are defined on page 3 in the paper.

What is the result of running:

1. `reducer (+) [1,2,3,4] 0`
2. `reducel (+) 0 [1,2,3,4]`
3. `reducer (+) (Node3 1 2 3) 0`
4. `reducel (+) 0 (Deep [1,2] (Single (Node3 3 4 5)) [0])`
5. `reducer (:) (Deep [1,2] (Single (Node3 3 4 5)) [0]) []`

6. `reducel (:) [] (Deep [1,2] (Single (Node3 3 4 5)) [0])`

Exercise 4. Write the following Haskell data terms using the corresponding Scala type constructors from `FingerTree.scala`

1. `[1,2,3]`
2. `[]`
3. `Node2 1 2`
   Observe that we work with `Node2[Int]` here (the type parameter is inferred).
4. `Node2 (Node2 1 2) (Node2 3 4)`
   After solving, make type parameters to `Node2` explicit, to see whether you understand the types.
5. `Deep [1,2] (FingerTree (Node3 3 4 5)) [0]`
   Also make the type parameters explicit.

Exercise 5. Translate to Scala the following Haskell type expressions. Use the algebraic data types for finger trees as defined in `FingerTree.scala`. Double colon in Haskell denotes a typing annotation, like a single colon in Scala. Haskell uses single arrows (`->`) as function type constructors, whereas Scala uses double arrows (`=>`). Small letters (free names) in Haskell types denote type variables. These can be both usual type parameters and higher kinded type parameters. Square brackets are also the list type constructor in Haskell, so `[a]` roughly corresponds to `List[A]` in Scala (adjusted for conventions).

NB. Some of these are solved in the Scala file, but please try to solve them yourself – then you will be able to work with the Scala file and with the paper more easily.

1. `addL :: a -> FingerTree a -> FingerTree a`
2. `addLprime :: f a -> FingerTree a -> FingerTree a`
3. `toTree :: f a -> FingerTree a`
4. `viewL :: FingerTree a -> ViewL FingerTree a`
5. `deepL :: [a] -> FingerTree a -> ViewL FingerTree a`

After solving the preparatory exercises, please continue work on the implementation.