National Undergraduate Programme in Mathematical Sciences

National Graduate Programme in Computer Science

Functional Programming in Haskell

Assignment 5

- **Due date**: November 21, 2023, 2359 hours
- There are two problems, with the second problem having several parts. A file `template.hs` is uploaded to the Moodle page. Replace the `undefined` with your definitions. Copy this file to `login.hs` and upload **a single file**. For example, my submission would be named `spsuresh.hs`.
- You can use any auxiliary functions and library **`import`**s in your files. You can use the `Queue` module presented in earlier lectures. If you do so, you should just use the functions exactly as presented in the `Queue.hs` uploaded as part of Lecture 21.
- Please compile and check that the sample cases are satisfied before submitting!

---

1. Write a program to solve the **Vanadurga riddle**. A set of people, numbered 1 to n, are seated clockwise in a circle. Durga starts at number 1 and eliminates every m^{th} person (not counting the ones who have been eliminated). After r people are eliminated, your program has to list all the eliminated people in the order of elimination, and all the survivors. We can assume that $m \geqslant 2$ and $0 \leqslant r < n$.

   Here are a few sample cases to illustrate the working of the program.

```
vanadurga :: (Int, Int, Int) → ([Int],[Int])
{- vanadurga (m,r,n) eliminates every m-th person in [1..n].
    This goes on for r rounds, and finally the eliminated ones and survivors are listed.
-}
vanadurga (3,10,15) = ([3,6,9,12,15,4,8,13,2,10],[1,5,7,11,14])
vanadurga (2,19,20) = ([2,4,6,8,10,12,14,16,18,20,3,7,11,15,19,5,13,1,17],[9])
vanadurga (2,0,20)  = ([],[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20])
vanadurga (9,8,10)  = ([9,8,10,2,5,3,4,1],[6,7])
```

2. In this problem, we define a binary search tree data structure where all data is stored in the leaves. More precisely, a tree is either empty or nonempty, and a nonempty tree is either a leaf or a branch with two subtrees (both of which are nonempty). A search tree is one where all the elements on the left subtree of a branch are strictly less than those on the right subtree. We would like to implement the `tSearch`, `tInsert`, `tDelete`, and `inorder` functions on trees. To be able to perform these easily, it is convenient to store at each branch the maximum element of the left subtree and the maximum element in the right subtree.

Given below are the datatype declaration, signatures for the functions, and a few test cases, where we use `it` to refer to the result of the previous expression. (The trees are displayed using the default **`show`** with some minimal formatting. But I have provided a custom **`show`** that will make it easier for you to run test cases.)

```
data Tree   = E | NE T
data T      = Leaf Int | Branch (Int,Int) T T
instance Show Tree where show = {- code already provided in file -}

tMax :: T → Int
{- tMax gives the maximum value in the nonempty tree -}

tSearch :: Int → Tree → Bool
{- tSearch x t == True iff x is an element of t -}

tInsert :: Int → Tree → Tree
{- tInsert x t adds x to t, if it is not already in the tree -}

tDelete :: Int → Tree → Tree
{- tDelete x t removes x from the tree -}

fromList :: [Int] → Tree
fromList    = foldr tInsert E
{- fromList l forms a tree consisting of all elements of l -}

inorder :: Tree → [Int]
{-  inorder t lists the leaves from leftmost to rightmost.
    We would like to satisfy the following equation.
                inorder . fromList = nub . sort
-}
fromList []                 = E
fromList [0..4]             = NE    (Branch (3,4)
                                        (Branch (2,3)
                                            (Branch (1,2)
                                                (Branch (0,1)
                                                    (Leaf 0)
                                                    (Leaf 1)
                                                )
                                                (Leaf 2)
                                            )
                                            (Leaf 3)
                                        )
                                        (Leaf 4)
                                    )
```

```
fromList [-6,2,10,-15,2]    = NE    (Branch (-15,10)
                                        (Leaf (-15))
                                        (Branch (2,10)
                                            (Branch (-6,2) (Leaf (-6)) (Leaf 2))
                                            (Leaf 10)
                                        )
                                    )
tDelete (-15) (tInsert 22 (tInsert 2 (tDelete 2 (tInsert (-3) it))))
                            = NE    (Branch (-3,22)
                                        (Branch (-6,-3) (Leaf (-6)) (Leaf (-3)))
                                        (Branch (2,22)
                                            (Leaf 2)
                                            (Branch (10,22) (Leaf 10) (Leaf 22))
                                        )
                                    )
inorder it                  = [-6,-3,2,10,22]
tDelete 5 (NE (Leaf 5))     = E
tDelete 6 (NE (Leaf 5))     = NE (Leaf 5)
tInsert 6 (NE (Leaf 5))     = NE (Branch (5,6) (Leaf 5) (Leaf 6))
tDelete 5 it                = NE (Leaf 6)
```