Exercise 1 – implement a generator of binary trees

A binary tree is either a leaf or it is composed of a root element and two successors, which are binary trees themselves. The type of binary trees is defined in Haskell as

```haskell
data Tree = Leaf | Node Tree Tree deriving (Show)
```

§1.a. Define a `countNode` function

```haskell
countNode :: Tree -> Int
```

which counts the number of nodes in a binary tree. Typically, the `countNode` function should behave as follows:

```haskell
ghci> countNode (Node Leaf Leaf)
1
```

§1.b. The size of a binary tree is defined as its number of nodes. Define a function

```haskell
genTree :: Int -> [Tree]
```

which generates all the binary trees of a given size. The `genTree` function should typically behave in this way:

```haskell
ghci> genTree 2
[Node (Node Leaf Leaf) Leaf, Node Leaf (Node Leaf Leaf)]
```

This two binary trees of size 2 may be represented as follows:

```
     *     *
      |     |
     *---*---*
```

where we only draw the nodes and not the leaves in the diagrams.

§1.c Define a function

```haskell
mirrorTree :: Tree -> Tree
```
which computes the mirror image of a binary tree, defined at the image under the mirror-image symmetry below:

![mirror-image symmetry](image.png)

§1.d. Define a function

```
genSymTree :: Int -> [Tree]
```

which generates all the symmetric binary trees of a given size. Here, by symmetric binary tree, we mean a binary tree which is equal to its own mirror image. The function should typically behave in this way:

```
ghci> genSymTree 2
[
]
ghci> genSymTree 3
[Node (Node Leaf Leaf) (Node Leaf Leaf)]
```

since there are no symmetric binary tree of size 2, and the unique symmetric binary tree of size 3 is the binary tree

```
Node (Node Leaf Leaf) (Node Leaf Leaf)
```

represented as

![binary tree](image.png)

Exercise 2 – from binary trees to Dyck words

A Dyck word is a string consisting of \( n \) \( X \)’s and \( n \) \( Y \)’s such that no initial segment of the string has more \( Y \)’s than \( X \)’s. For example, the following are the Dyck words of length 6:

```
XXXYYY  XYXXYY  XYXYXY  XXXYXY  XXXXYY
```

§2.a. Show by induction that there is a one-to-one correspondence between:

- the binary trees of size \( n \) (which contain \( n \) nodes) in the sense of §1,
• the Dyck words of length $2n$.

§2.b. Define a `fromTreeToWord` function

```
fromTreeToWord :: Tree -> String
```

which associates to every binary tree the corresponding Dyck word.

§2.c. Deduce from §1.d a function

```
genDyckWord :: Int -> [String]
```

which generates the list of all Dyck words.

§2.d. Do you see a direct way to define the `genDyckWord` function, without using the one-to-one correspondence between Dyck words and binary trees?

§2.e. Define the type `Tree` of binary trees

```
data Tree = Leaf | Node Tree Tree
```

and instantiate it in the type class `Show` in such a way that the `show` function of the type class is defined as the `fromTreeToWord` function.

**Exercise 4 of HW3 – binary trees and polymorphism**

We are interested in the binary trees without making any assumption on the type of the values stored on each of their node. We define the polymorphic type of binary trees in the following way

```
data BinTree a = Empty | Node a (BinTree a) (BinTree a)
```

§4.a. Define the function

```
treeToBinTree Tree -> BinTree Int
```

which transports every binary tree input to the same binary tree output seen this time as a value of type `BinTree Int`. Take this opportunity to define `binIllustration` as the binary tree

```
ghci> let binIllustration = fromTreeToBinTree illustration
```

§4.b. Define the function

```
showBinTree :: (Show a) => BinTree a -> String
```
which prints the binary tree as a sequence of integers and brackets, obtained by in-order left-to-right exploration of the tree. Typically, the `showTree` function should behave in the following way on the binary tree `illustration`:

```
ghci> showBinTree binillustration
"(()1())2(((3())4((5()))6(()7())))"
```

§4.c. Proceed in the same way as in §3.e. and turn the type `(BinTree a)` into an instance of the type class `(Show a)` for a type variable `a`.

§4.d. Define the function

```
listToBranch BinTree a -> BinTree [a]
```

which transports every binary tree `input` to the binary tree `output` where every node is labelled by the list of elements appearing from the root to the node in the original tree `input`. Typically, the binary tree `binillustration` is transformed in the following way:

```
listToBranch binillustration =

```

The `listToBranch` function should behave in the following way:

```
ghci> binillustration
(()1())2(((3())4((5()))6(()7())))
ghci> listToBranch binillustration
(()[2,1])2(((([2,6,4,3]())[2,6,4]())[2,6,4,5]())[2,6](()[2,6,7]()))
```