Exercise 1 – testing whether a string is a palindrome

§1.a. Write a Haskell function `reverseList` of type

```
reverseList :: [a] -> [a]
```

which takes a list as input and reverses it in the following way:

```
ghci> reverseList [2,3,7,5]
[5,7,3,2]
```

§1.b. Write a Haskell function `isPalindrome` of type

```
isPalindrome :: String -> Bool
```

which takes a character string and returns a boolean indicating whether the string is a palindrome or not:

```
ghci> isPalindrome "apple"
False
ghci> isPalindrome "neveroddoreven"
True
```

§1.c. The two Haskell functions

```
isLetter :: Char -> Bool
toLower :: Char -> Char
```

are implemented in the library `Data.Char`. The purpose of the `isLetter` function is to indicate whether a given character is a letter, while the purpose of the `toLower` function is to turn every uppercase letter into the corresponding lowercase letter. Import the two Haskell functions using the instruction

```
import Data.Char (isLetter, toLower)
```

and then define a `isPalindromeStr` function
isPalindromeStr :: String -> Bool

which improves the isPalindrome function by accepting spaces and upper case letters in palindromes. The isPalindromeStr should behave in the following way:

ghci> isPalindromeStr "never odd or even"
True
ghci> isPalindromeStr "Madam I’m Adam"
True

§1.c. Explain why the Haskell function isPalindrome is of the more general type

isPalindrome :: (Eq a) => [a] -> Bool

while this is not the case of the improved function isPalindromeStr.

Exercise 2 – the Quicksort algorithm in Haskell

The problem of sorting a list from its lowest value to its highest one has been very much studied in computer science, and there are literally dozens of algorithms for doing it. One well-known example of the “divide-and-conquer” algorithm known as Quicksort and designed in 1959 by the famous British computer scientist Tony Hoare. In order to sort a list of ordered values, the Quicksort algorithm works as follows:

1. If the list is empty, it is already sorted.
2. Otherwise, take the first element of the list and call it pivot.
3. From the rest of the list, create two lists, the first list containing all of the elements strictly less than pivot and the second list containing all of the elements greater than or equal to pivot.
4. Then sort (using the algorithm recursively) the two new lists.
5. Finally, concatenate the two sorted lists together, sticking pivot in the middle.

§2.a. Using pattern matching, write a Haskell function quicksort of type

quicksort :: [Int] -> [Int]

which implements the Quicksort algorithm on lists of integers.

§2.b. Show that the resulting quicksort function can be given the more general type

quicksort :: (Ord a) => [a] -> [a]

where Ord is the class type of ordered types. Explain why in a few words.
Exercise 3 – the type Tree of binary trees

We are interested in the set of binary trees whose nodes are labelled with an integer, in the following way:

\[
\text{illustration} = \begin{array}{c}
\text{2} \\
\text{1} & \text{6} \\
\text{4} & \text{7} \\
\text{3} & \text{5} \\
\end{array}
\]

To that purpose, we define the Haskell type

\[
data Tree = \text{Nil} \mid \text{Node Int Tree Tree}
\]

§3.a. Represent the binary tree illustration depicted above as a value of the Haskell type Tree just defined.

§3.b. Write the Haskell function

\[
\text{treeTraversal} = \text{Tree} \to [\text{Int}]
\]

which transforms a binary tree in the list of integers obtained by left-to-right in-order traversal. Typically, the treeTraversal function should behave in the following way when one applies it on the binary tree illustration:

\[
\text{ghci} > \text{treeTraversal illustration} \\
[1,2,3,4,5,6,7]
\]

§3.c. Write the Haskell function

\[
\text{treeLeaves} = \text{Tree} \to [\text{Int}]
\]

which computes the list of leaves of the binary tree, presented from left to right. Typically, the treeLeaves function should behave in the following way when one applies it on the binary tree illustration:

\[
\text{ghci} > \text{treeLeaves illustration} \\
[1,3,5,7]
\]

§3.d. Write the Haskell function
showTree = Tree -> 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> showTree illustration
"(()1())2(((()3())4(()5()))6(()7()))"

§3.d. Write the Haskell function

```
showTree = Tree -> 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> showTree illustration
"(()1())2(((()3())4(()5()))6(()7()))"

§3.e. Taking inspiration from the showTree function just implemented in the previous question §3.d., turn the type Tree into an instance of the type class Show. As a result, the interactive ghci should automatically print the value of every binary tree, in the following way:

ghci> illustration
(()1())2(((()3())4(()5()))6(()7()))

§3.f. Implement a function

```
sumToBranch :: Tree -> Tree
```

which computes for every node of the input tree the sum of all the nodes on the branch to the root. Typically, the binary tree illustration is transformed into the binary tree

```
sumToBranch illustration =
```

```
    1
   /|
  2 3 8
 /    |
12 15 15
    /|
   15 17
```

Here the number 17 appears in the pointed node because 17 = 2 + 6 + 4 + 5 is the sum of the numbers appearing on the branch from the node to the root of the original binary tree illustration, as depicted below:

![Binary Tree Diagram](image)

The sumToBranch function should behave in the following way:

```
ghci> illustration
(()1())2(((()3())4((()5()))6(()7())))
ghci> sumToBranch illustration
(()3())2(((15())12((17()))8())15())
```

### Exercise 4 [optional] – 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

```
data 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 =

[2]
[2,1]
[2,6]
[2,6,4]
[2,6,4,3]
[2,6,4,5]
[2,6,7]

```

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]()())
```

**Bibliographic note:** the second exercise is adapted from Paul Hudak’s excellent book *The Haskell School of Expression*, published at Cambridge University Press, 2000.