

# Trees

Andres Löh

Difficulty: medium

The goal of this task is to work with binary trees. Start a new module and define a datatype of binary trees as follows:

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

The `deriving Show` instructs Haskell to automatically generate a function that allows GHCi (and you) to show and print trees on screen.

## 1. Write a function

```
single :: a -> Tree a
```

that builds a tree with a single element.

Example:

```
> single 3
Node Leaf 3 Leaf
```

## 2. Write a function

```
size :: Tree a -> Int
```

that counts the number of nodes in a tree.

Example:

```
> size (Node (single 4) 7 (single 5))
3
```

3. Write a function

```
height :: Tree a -> Int
```

that counts the height of a tree (the maximal distance of the root to a leaf).

Example:

```
> height (Node (single 4) 7 (Node (single 1) 3 (single 7)))
4
```

4. Write a function that flattens a tree into a list

```
flatten :: Tree a -> [a]
```

Example:

```
> flatten (Node (single 4) 7 (single 5))
[4, 7, 5]
```

5. Write a function that reverses a tree

```
reverse :: Tree a -> Tree a
```

Example:

```
> flatten (Node (single 4) 7 (single 5))
Node (Node Leaf 5 Leaf) 7 (Node Leaf 4 Leaf)
```

6 (medium). Write a function that implements tree-sort, an algorithm that is similar to quicksort: given a nonempty list, take the first element, partition the list in all element smaller of equal and all elements larger, and create a node with the first element in the root, and recursively invoke the function build the left and right subtrees. The resulting tree should be a binary search tree.

```
treesort :: Ord a => [a] -> Tree a
```

Calling `flatten` on the result of `treesort` should yield a sorted list.

7. Write a function that checks if a tree is a binary search tree.

```
bst :: Ord a => Tree a -> Bool
```

8 (medium). Write a function that labels a tree, each node differently, by traversing the tree, maintaining a state, and assigning a new number to every node.

```
labelTree' :: Tree a -> State Int (Tree Int)
```

Remember to import `Control.Monad.State` to access the state monad. There is a function

```
runState :: State s a -> s -> (a, s)
```

you can use to pass in an initial state and get at the final state and result in the end:

```
labelTree :: Tree a -> Tree Int
labelTree t = fst (runState (labelTree t))
```