Recursive Types and Implementing Binary Search Trees

In this section we will consider recursively defined types, which allow us to describe structures of arbitrary size, like lists and trees. We will in particular build search trees using the mechanism of recursive types.

Recursive Types

A recursive type is a custom data type that refers to itself in one of its variants. In this way, a value of a particular type is either one of the basic options for that type, or a combination of simpler values, possibly of the same type.

For instance, we can use recursive types to implement binary search trees. Recall that a binary search tree is a binary tree, so each node has two (possibly empty) children, and each node also contains a value. In a binary search tree all values within the left child are less than the value at the node, and all values within the right child are greater than the value at the node.

Practice: Draw at least 3 different binary search trees consisting of the numbers 4, 6, 10, 23, 40.

We will represent a tree via a custom data type, with two variants: One representing the "empty" node and one representing an actual value node with a value and two children:

```
data Tree a = E | N (Tree a) a (Tree a)
```

Here the expression `N (Tree a) a (Tree a)` contains three values along with the `N` tag: The first is a `Tree a` value, which represents the left child. Then the `a` in the middle represents the value at the node, and the second `Tree a` represents the right child.

We can now build trees recursively by using these constructors `E` and `N`. For example, let us start with a simple function `leaf` that turns a single value into a node containing that value and with empty children (we usually call such nodes leaves):

```
leaf :: a -> Tree a
leaf v = N E v E
```

Let us also write a function that tests if a "tree" is a "leaf". A leaf is a tree both of whose children are empty:

```
isLeaf :: Tree a -> Bool
isLeaf (N E _ E) = True
isLeaf _         = False
```

Finally, let's write a function that turns a binary tree into a list:

```
toList :: Tree a -> [a]
toList E              = []
toList N left v right = (toList left) ++ (v :: toList right)
```

This provides what is typically called an in-order traversal of the tree.

Question: How would we obtain pre-order or post-order traversals instead?

Practice:

1. Write a method to count the number of non-empty nodes in a tree (do it directly with recursion, and NOT using toList).
2. Write a method that adds up all the values in all the (non-empty) nodes. The empty nodes should return 0 value.

Now let us proceed to write an insert method, that inserts a new element into the proper place in the tree. It would need to have type: Ord a => a -> Tree a -> Tree a, so it takes an element and a tree, and returns a new tree with the element inserted in the correct spot. This will have various cases:

- If we are dealing with a normal node, compare the value at the node with the given value. If they are equal, then the number is already there and does not need to be inserted again. If the searched value is smaller than the one in the node, then we try to insert in the left child, else we insert in the right child.
- If we are dealing with an empty node, then we just form a new element.

```
insert :: Ord a => Tree a -> a -> Tree a
insert v E           = leaf v
insert v (N l v' r) =
    case compare v v' of
        EQ -> N l v' r
        LT -> (insert v l) v' r
        GT -> N l v' (insert v r)
```

Practice

1. Write a function contains :: Ord a => a -> Tree a -> Bool that given a tree and an element searches for that element in the tree. Write two versions of this function: One version assumes that the tree is a binary search tree and uses this property to search only where it needs to; the other version does not make that assumption and therefore has to check every single location.
2. Write a function any :: (a -> Bool) -> Tree a -> Bool that given a tree and a predicate returns True if there is at least one element in the tree for which the predicate is True, and False otherwise (including empty trees). You may need to traverse all branches.
3. Write a function all :: (a -> Bool) -> Tree a -> Bool that given a tree and a predicate returns True if for all elements in the tree the predicate is True, and False otherwise. It should be True for empty trees (there is no element that can make the predicate False). You may need to traverse all branches.

4. Write a function `min :: Ord a => Tree a -> a` that given a binary search tree finds the smallest element. It should error on an empty tree. You would more or less have to traverse the left children. Draw some tree examples before attempting this.

5. Write a function `deleteMin :: Ord a => Tree a -> Tree a` that given a binary search tree removes the smallest element. It should error on an empty tree. You would more or less have to traverse the left children. Draw some tree examples before attempting this.

6. Write a function `delete :: Ord a => a -> Tree a -> Tree a` that given a binary tree removes the provided element (or errors if the element doesn't exist). The process for deleting an element once you have found its node would be as follows:

   - If the element to be deleted has no right child, then simply replace it with its left child.
   - If the element does have a right child, then: Find the smallest element of the right child, using `min`, remove that smallest element from the right child using `deleteMin`, then place that smallest element in the node of the element you are deleting, forming a new node with the same left child as before, the updated right child, and the new element as the value. Draw a picture of this to understand it first.

7. Write a function `randomTree :: Random a, RandomGen g => g -> n -> (Tree a, g)` which is given a random number generator and an integer `n` and returns a tree with `n` randomly generated nodes (along with the continuing generator).