5 Recursive definitions on trees

The datatype declaration

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

defines binary trees. Use pattern-matching to give recursive definitions of:

1. a function *size* :: *Tree a* → *Integer* that calculates the number of elements in a tree;

   There is one element in each *Node*, so the size of the tree is also the number of *Node*s in the tree. The *Empty* tree is clearly not a *Node*; and a tree of the form *Node l a r* has one *Node* at the root, plus however many *Node*s are in *l* and *r*.

   ```
   size :: Tree a → Integer
   size Empty        = 0
   size (Node l a r) = size l + 1 + size r
   ```

2. a function *tree* :: [*a*] → *Tree a* that converts a list into a tree;

   The naive way to implement *tree* is to construct a right-leaning tree where all left subtrees are empty and the right subtree contains the the rest of the tree; but this results in a very unbalanced tree. To get a balanced tree, it is possible to split the list in half, use the middle element as the root node, and then use the left half of the list to create the left subtree, and the right half to create the right subtree.

   ```
   tree :: [a] → Tree a
   tree [ ] = Empty
   tree xs = Node (tree right) x (tree left)
     where
       (right, x : left) = splitAt (length xs `div` 2) xs
   ```

3. a function *memberT* :: *Eq a* ⇒ *a* → *Tree a* → *Bool* that determines whether a given tree contains a specified element;

It is clear that a value cannot be in the *Empty* tree. A value can be in a non-empty tree in three ways: at the root, in the left subtree, or in the right subtree.

```
memberT :: (Eq a) ⇒ a → Tree a → Bool
memberT x Empty        = False
memberT x (Node l a r) = (x == a) || memberT x l || memberT x r
```

Note that because (||) is non-strict in its right argument, when a value is found in some *Node* it is not necessary to continue searching in the subtrees of that *Node*.

4. a function *searchTree* :: *Ord a* ⇒ [*a*] → *Tree a* that converts a list into a search tree (a search tree is a tree in which, for a given node, all the values in the left subtree are smaller and all the values in the right subtree are larger);

One way to create a search tree is to write an insertion function that inserts an element into an existing search tree, and use it to insert elements of a list one by one into an initially *Empty* search tree.

```
searchTree :: (Ord a) ⇒ [a] → Tree a
searchTree [ ]      = Empty
searchTree (x : xs) = insert x (searchTree xs)
  where
    insert x Empty = Node Empty x Empty
    insert x (Node l a r)
      | x < a      = Node (insert x l) a r
      | x > a      = Node l a (insert x r)
      | otherwise  = Node l a r
```

This can result in a highly unbalanced tree, however, if the elements are in sorted or almost sorted order; how would you avoid this?

5. a function *memberS* :: *Ord a* ⇒ *a* → *Tree a* → *Bool* that determines whether a given search tree contains a specified element (this should be more efficient than your definition of *memberT*);

```
memberS :: (Ord a) ⇒ a → Tree a → Bool
memberS x Empty = False
memberS x (Node l a r)
   | x < a       = memberS x l
   | x > a       = memberS x r
   | otherwise = True
```

6. a function *inOrder* :: *Tree a* → [*a*] that produces the list of elements from an in-order tree traversal (an in-order tree traversal is one in which the left subtree is traversed, then the root node, and then the right subtree).

```
inOrder :: Tree a → [a]
inOrder Empty          = [ ]
inOrder (Node l a r) = inOrder l ++ [a] ++ inOrder r
```

5.1 More exercises on algebraic datatypes

There are some more exercises on various kinds of tree datatype in the optional practicals on Huffman Coding and the MILan programming language, which you can find towards the end of this handout. I consider them to be optional exercises; I expect you'll have time for them only if you're flying through the other exercises. I may work through some parts as 'live coding' during the course.