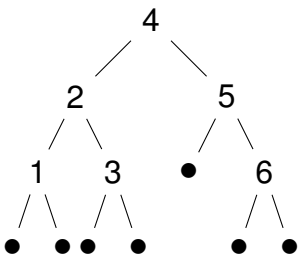
Problem Sheet 3.2: Lists in Haskell

In this tutorial we will look at trees, constructed as **recursive data types**. First we will look at the binary trees as we introduced in the unit content, which store integers at the nodes (but not the leaves):

```
data IntTree = Empty | Node Int IntTree IntTree
  deriving Show


t :: IntTree
t = Node 4 (Node 2 (Node 1 Empty Empty) (Node 3 Empty Empty))
           (Node 5 Empty (Node 6 Empty Empty))
```

We can draw the tree `t` as follows, using • for `Empty`. Note that each internal (i.e., non-leaf) node has three attributes: an integer, and two children, each itself a (sub)tree. Also, this is a computer-science tree: these grow from the ceiling down, as opposed to mathematical trees which grow from the ground up.



"`Deriving Show`" tells Haskell to make a default instance of the `Show` class for the `IntTree` type. It creates a literal representation of the data type: try it out with `*Main> t`.

Exercise 1: Complete the following functions.

a) `isEmpty`: determines whether a tree is `Empty` or not.

b) `rootValue`: returns the integer at the root of the tree, or zero for an empty tree.

c) `height`: returns the height of the tree. A leaf has height zero, and a node is one higher than its highest subtree. The function `max` will be helpful.

d) `find`: finds whether an integer occurs in the tree.

```
*Main> isEmpty t
False
*Main> rootValue t
```

```
4
*Main> height t
3
*Main> find 3 t
True
```

As in the last tutorial, we can make our own `Show` instance, with our own `show` function for trees. To get a readable layout we print a tree sideways, with the root to the left, and using indentation to indicate the parent-child relation. Browsing directories on Windows uses this, for instance. Comment out the line `deriving Show`, and un-comment the given `Show` instance. Try it out: a `+` indicates the root of a (sub)tree, connected to its parent with `|`.

```
*Main> t
    +-1
  +-2
  | +-3
+-4
  +-5
    +-6
```

Ordered trees

Note that the tree `t` is **ordered**: for every node, the values in the left subtree are all smaller than the value of the node, and those in the right subtree are all larger. Ordered trees are extremely useful, since (for instance) to find or insert an item you only need to traverse a single path from the root to a leaf. The longest such path is the **height** of the tree, and if the tree is **balanced**, i.e., all paths have similar length, the height is only a **logarithmic** factor of the size. For an ordered tree, the `flatten` function from the content returns an ordered list.

From here on, we will assume our `IntTree` type is **ordered**. But there is nothing we can do with the Haskell type system to enforce that. Here, we've found a limit to what safety guarantees the type system can give. (However, there are languages with stronger type systems which can enforce such constraints.)

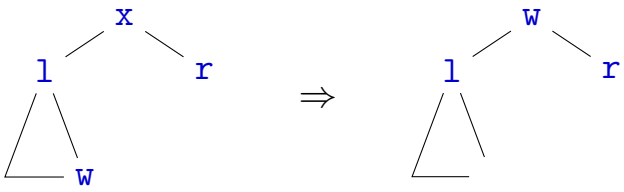
Exercise 2: Complete the following functions.

a) `member`: returns whether a given integer `i` occurs in a tree. Compare this against your earlier `find` function for unordered trees.

b) `largest`: find the largest element in a tree. **Hint:** the corresponding `smallest` function is in the content.

c) `deleteLargest`: delete the largest element from a tree. **Hint:** this is similar to `largest`, except when you find the element you delete it. The relevant case should have a simple way of returning a tree not containing the element.

d) `delete`: delete an element `x` from a tree (or return the original tree if it doesn't contain `x`). This is a bit of a puzzler, so take some time to think it through. There are four cases, given by the guards in the tutorial file. First, if `x` is in the left or right subtree, delete it there recursively. Otherwise, `x` is the element to delete. First, if it happens to be the smallest element, it can be deleted easily (similar to `deleteLargest`). Otherwise, to maintain the ordering, you can replace `x` with the element `w` that is immediately smaller. This is the largest element in the left subtree `l`; use your `largest` and `deleteLargest` to replace `x` with it. The following schematic illustrates the idea.




```
*Main> member 3 t
True
*Main> largest t
6
*Main> deleteLargest t
    +-1
  +-2
  | +-3
+-4
  +-5
*Main> delete 1 t
  +-2
  | +-3
+-4
  +-5
    +-6
*Main> delete 4 t
    +-1
  +-2
+-3
  +-5
    +-6
```