Tutorial 7A: Ordered trees

In the previous Haskell tutorial, we looked at the following binary trees of integers:

```
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))
```

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 lecture 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 1: Complete the following functions.

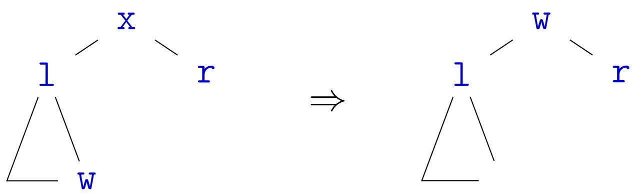
a) `member` : returns whether a given integer `i` occurs in a tree. Make sure to avoid searching any subtree unnecessarily. (Compare this against the `member` function for unordered trees from the previous tutorial.)

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

c) `ordered` : returns whether a tree is ordered. **Hint:** An easy way to do this is to use `flatten` from the lecture and (an adaptation of) `sorted` from Tutorial 2. Another way is to use `largest` above, add a similar `smallest` function, and then use Boolean operators to test for every node with value `x` that: if the left subtree `l` is non-empty, its largest value is smaller than `x` , and `l` is itself ordered; if the right subtree `r` is non-empty, its smallest value is larger than `x` , and `r` is itself ordered.

d) `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.

e) `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` by 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
```

(Optional challenge) The suggested implementations of `ordered` are inefficient (the first because `flatten` is inefficient, the second because `largest` and `smallest` traverse the tree more often than necessary). Can you find an efficient (linear-time) implementation?

Exercise 2: Change your `IntTree` data type so that it can carry any type `a` instead of `Int`. It should start like this:

```
data Tree a = ...
```

Your file will no longer type-check at this point, so comment out the type signatures of any function on `IntTree`, and replace the `Show IntTree` instance with the given

```
instance Show a => Show (Tree a) ...
```

The tree `t` should get the type `Tree Int` — this is now what the `IntTree` type used to be. Your file should type-check again. Give your functions new type signatures to work with the type `Tree a`, using appropriate constraints `Eq a =>`, `Ord a =>`, or `Num a =>`.

```
*Main> Node "Hello" (Node "Darkness my old friend" Empty Empty)
 (Node "My name is" (Node "Inigo Montoya" Empty Empty)
   (Node "Slim Shady" Empty Empty))

  +-"Darkness my old friend"
+-"Hello"
  | +-"Inigo Montoya"
  +-"My name is"
    +-"Slim Shady"
```

Hints: your new `Node` constructor stores a value of type `a` and two children of type `Tree a`. An occurrence of `Tree` in a type declaration should become `Tree a`. A type `Int` should sometimes change to `a`, but not always! Use any error messages you may be getting to find the right constraint. The type class `Num a` is for any type representing numbers, such as `Int` and `Integer`, but also `Float`.

(Optional challenge) the `sort` function from the lecture has good average complexity ($n \cdot \log n$), but quadratic worst-case complexity (when the list is already sorted). For good worst-case performance you can use **self-balancing** trees, like **AVL**-trees or **red-black**-trees. You can find these on Wikipedia. Implement a data type and `member` and `insert` functions for one or both of them.

Lambda-calculus: capture-avoiding substitution

We will continue implementing the λ-calculus. Your Haskell file contains the solutions to the previous Haskell tutorial, which includes the data type `Term` for λ-terms, various example terms, and the functions `used` and `free` to find the used variables and free variables in a term. First, we add a lot more examples.

Exercise 3: Complete the function `numeral` which given a number i, returns the corresponding Church numeral N_i as a `Term`. Recall that the Church numerals are:

$$N_0 = \lambda f.\lambda x.\,x \qquad N_1 = \lambda f.\lambda x.\,f\,x \qquad N_2 = \lambda f.\lambda x.\,f\,(f\,x) \qquad \dots$$

You may find the following recursive definition of the numeral N_i helpful.

$$N_i \;=\; \lambda f.\lambda x.\,N'_i \qquad \begin{array}{ll} N'_0 = x & \\ N'_i = f\,(N'_{i-1}) & (\text{if } i \neq 0) \end{array}$$

```
*Main> numeral 2
\f. \x. f (f x)
```

In the next part of the tutorial we will add **variable renaming** and **capture-avoiding substitution**. Recall the renaming operation $M[y/x]$ (M with x renamed to y) from the lectures, slightly paraphrased:

$$z[y/x] \;=\; \begin{cases} y & \text{if } z = x \\ z & \text{otherwise} \end{cases}$$

$$(\lambda z.M)[y/x] \;=\; \begin{cases} \lambda z.M & \text{if } z = x \\ \lambda z.(M[y/x]) & \text{otherwise} \end{cases}$$

$$(M N)[y/x] \;=\; (M[y/x])\,(N[y/x])$$

The definition of capture-avoiding substitution, similarly paraphrased, is:

$$y[N/x] \;=\; \begin{cases} N & \text{if } y = x \\ y & \text{otherwise} \end{cases}$$

$$(\lambda y.M)[N/x] \;=\; \begin{cases} \lambda y.M & \text{if } y = x \\ \lambda z.(M[z/y][N/x]) & \text{otherwise} \end{cases}$$

where z is **fresh**: not used in M or N, and $z \neq x, y$

$$(M_1 M_2)[N/x] \;=\; (M_1[N/x])(M_2[N/x])$$

Note that both definitions now give a direct template for the corresponding Haskell function.

Exercise 4:

a) Complete `variables` as an infinite list of variables, first `"a"` through `"z"`, then repeating these suffixed with 1, `"a1",...,"z1"`, then 2, `"a2",...,"z2"`, etc.

b) Complete the function `fresh` which given a list of used variables, generates a fresh variable not occurring in the list. **Hint:** use the `removeAll` function from previous tutorials to remove all the used variables from your list `variables`, and then take the first remaining variable.

c) Complete the function `rename x y m` that renames `x` to `y` in the term `m`, i.e. $M[y/x]$.

d) Complete the function `substitute` that implements capture-avoiding substitution, i.e. `substitute x n m` corresponds to $M[N/x]$. Use `fresh` to generate the fresh variable z as above; it must not be used in `n` and `m`, and not be `x`.

```
*Main> take 10 variables
["a","b","c","d","e","f","g","h","i","j"]
*Main> [variables !! i | i <- [0,1,25,26,27,100,3039]]
["a","b","z","a1","b1","w3","x116"]
*Main> fresh ["a","b","x"]
"c"
*Main> fresh (used example)
"d"
*Main> rename "b" "z" example
\a. \x. (\y. a c) x z
*Main> substitute "z" (Variable "HERE") n2
\a. (\b. a) HERE
*Main> substitute "z" (Apply (Variable "a") (Variable "b")) n2
\c. (\d. c) (a b)
*Main> substitute "c" (Variable "HERE") example
\d. \a. (\a. d HERE) a b
*Main> substitute "c" (numeral 2) example
\d. \a. (\a. d (\f. \x. f (f x)) a b
```