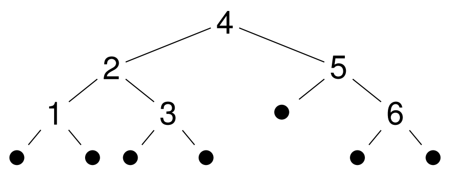
Tutorial 4: Trees

In this tutorial we will look at trees, constructed as **recursive data types**. First we will look at the binary trees of the lectures, 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> isLeaf 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 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 2: Complete the following functions.

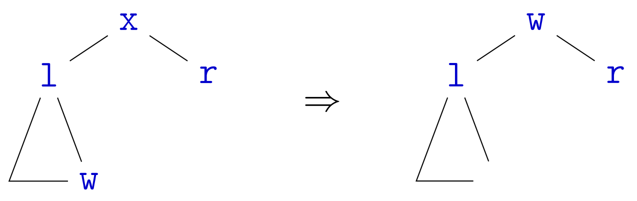
a) member: returns whether a given integer i occurs in a tree. Compare this against your earlier find function for unordered trees.

b) largest: find the largest element in a tree. **Hint:** the corresponding smallest function is in the 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 3: 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 the functions for the previous exercises, 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 "For" Empty (Node "The" Empty (Node "Watch" Empty Empty))
+-"For"
  +-"The"
    +-"Watch"

*Main> member "The" it
True
```

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.

Dialogues

We will use trees to create dialogue in your adventure game. In typical fashion, your conversation partner will say something, you will be given one ore more options how to respond, and the conversation continues depending on your choice. This is a natural tree structure.

Exercise 4:

a) Complete the data type `Dialogue` with two constructors:

- `End`, with a `String` to give a closing message;
- `Choice`, with a `String` for the initial message, and a list containing pairs of your answer (a `String`) and the subsequent conversation (a `Dialogue`).

b) Complete the `instance Show Dialogue` in the following way.

- At the `End` of a dialogue, just the final message should be returned, with a newline character (`'\n'`) appended at the end for better layout.
- At a `Choice` in the dialogue, display the initial line, then a line break (`'\n'`), and a list of your possible replies. Use the `enumerate` function from last tutorial (provided) to list these options. Note that your function is **not** recursive: we're not displaying the whole tree, just the current options.

c) Complete the function `choose` that, given a dialogue `Choice` and an integer `i`, chooses the option corresponding to `i` in the displayed list of choices.

Un-comment the sample dialogue `bridgeOfDeath` to test your functions:

```
*Main> bridgeOfDeath
Stop. Who would cross the Bridge of Death must answer me
these questions three, ere the other side he see.
  1. Ask me the questions, bridgekeeper. I am not afraid.

*Main> choose it 1
What... is your name?
  1. My name is Sir Lancelot of Camelot.
  2. Sir Robin of Camelot.
  3. Sir Galahad of Camelot.
  4. It is Arthur, King of the Britons.

*Main> choose it 2
What... is your quest?
  1. To seek the Holy Grail.

*Main> choose it 1
What... is the capital of Assyria?
  1. I don't know that!

*Main> choose it 1
[Thunk] WAAAAaaaaaauuuuggh
```