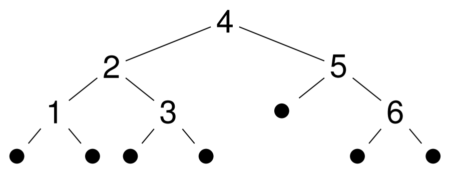
Tutorial 5: Trees

In this tutorial we will look at **trees**, constructed as **inductive data types**. It might be helpful to have pen and paper ready to draw the trees and their algorithms before implementing them. First we will look at the binary trees of the lectures, which store integers at the nodes:

```
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.



The line "`deriving Show`" tells Haskell to make the IntTree type an instance of the `Show` class by generating a default `show` function. It creates a literal representation of the data type: try it out with `ghci> 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) `member` : returns whether an integer occurs in the tree.

```
ghci> isEmpty t
False
ghci> rootValue t
4
ghci> height t
3
ghci> member 3 t
True
```

For better readability, we can make our own `Show` instance, with our own `show` function for trees. We print a tree sideways, with the root to the left, and use 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 `|`.

```
ghci> 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. the children of a node have similar height, the height of the tree is a **logarithmic** factor of the size.

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.)

Your file has the functions `insert`, `build`, `flatten`, and `treemap` from the lectures as examples and to aid with testing.

Exercise 2: Complete the following functions.

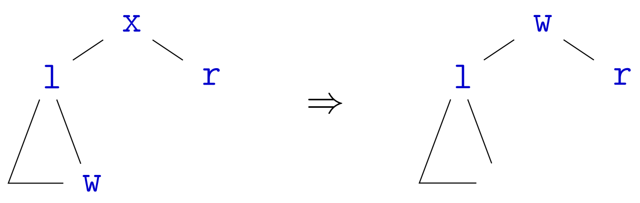
a) `present`: returns whether a given integer `i` is present in a tree. Make sure to avoid searching any subtree unnecessarily. (Compare this against the `member` function for unordered trees above.)

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:** use the `largest` and `smallest` functions and follow the definition: a (non-empty) tree is ordered when both subtrees are ordered, and when the value at the node is larger than the largest value of the left subtree, and smaller than the smallest value of the right subtree.

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 a given 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.



f) `sorted`: your `ordered` function is not as efficient as it could be, since it uses `smallest` and `largest` at every node. Complete the function `sorted` that tests if a tree is ordered using the following algorithm. First, define the auxiliary function `inRange` that tests whether a tree is ordered **and** all elements are (strictly) within the range given by two input numbers. Then a non-empty tree is `sorted` if all elements fall within the range of the smallest value minus one, and the largest value plus one.

```
ghci> member 3 t
True
ghci> largest t
6
ghci> ordered t
True
ghci> deleteLargest t
    +-1
  +-2
  | +-3
+-4
  +-5
```

```
ghci> delete 1 t
  +-2
  | +-3
+-4
  +-5
    +-6
ghci> delete 4 t
    +-1
  +-2
+-3
  +-5
    +-6
```

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. To fix this, first replace the `Show IntTree` instance with the new `Show (Tree a)` .

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

Second, un-comment the line:

```
type IntTree = Tree Int
```

This creates a **type alias**: the `IntTree` type now becomes a shorthand for `Tree Int` (your new `Tree a` type specialised to `Int`). The keyword `type` is used to create type aliases.

Give the functions in your file new type signatures to work with the type `Tree a` , using appropriate constraints `Eq a =>` , `Ord a =>` , or `Num a =>` .

```
ghci> 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 `IntTree` 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` .

Extra: AVL trees

Ordered trees are very effective when they are balanced (when the heights of the left and right subtrees of a node are similar), since then the height of the tree is logarithmic in the size. Unfortunately, inserting into an ordered tree does not automatically give a balanced tree. For example, consider the tree given by `build [0..9]`, which is unbalanced, and one given by building from an arbitrary list, say `build [3,5,9,1,6,2,4,0,8,7]`.

AVL trees [Adel'son-Velskii & Landis, 1962] are **self-balancing trees**: ordered trees that maintain balance. For each node, the difference in height between the left and right subtrees is kept at zero or one. This is recorded at each node as the **balance factor**, the height of the right subtree minus that of the left, which is -1, 0, or +1. When a value is inserted, the algorithm returns not only the result tree, but also a 0 or +1 whether the height of the tree has increased, so that balance factors can be updated and balance can be maintained.

When the tree threatens to lose balance, i.e. when a node would reach a balance factor of -2 or +2, balance is restored by reorganising the top two or three nodes, in operations called **rotations**. There are eight rotations in total, four each for -2 and for +2. Those for factor +2 are shown in Figure 1: the numbers give the balance factors of the rotating nodes; the colours and horizontal alignment are the same before and after so it can be seen which nodes end up where; and the vertical alignment illustrates the relative heights of each subtree.

Exercise 4: You are given a data type `Balance` for balance factors and one `AVLTree` for AVL trees, plus a mapping `forget` from AVL trees to regular trees that removes the balance factors, and a `Show` instance (un-comment these). Complete the function

```
 insertAVL :: Ord a => a -> AVLTree a -> (Bool,AVLTree a)
```

which inserts a value into an AVL tree, and returns a `Bool` to indicate if the insertion caused the tree to grow. The auxiliary functions `balanceLeft` and `balanceRight` implement the cases where the inserted value caused the left subtree, respectively right subtree, to grow. Their arguments are: the old balance factor, the value, and the new subtrees. The given case below corresponds to the first rotation in Figure 1:

```
 balanceRight Positive x a (AVLNode Positive y b c) =
         (False, AVLNode Neutral y (AVLNode Neutral x a b) c)
```

The right subtree has grown, so the old balance `Positive` (+1) would become (+2), triggering the rotation. The pattern is that of the first illustrated rotation: the (orange) root node "`x`", the (blue) right child node "`y`" with balance factor +1, and the three subtrees `a`, `b`, and `c` shown in the picture as the three (red, green, and violet) solid circles. The resulting tree takes `y` as the root, `x` as the left child, and keeps the three subtrees in order. In the illustration, the height of the tree has decreased, which means the tree no longer grows (i.e. the first element of the result is `False`).

There are six cases to complete for each of `balanceLeft` and `balanceRight`, four of which are rotations.

```
ghci> Leaf

ghci> snd (insertAVL 1 it)
+-1

ghci> snd (insertAVL 2 it)
+-1
  +-2

ghci> snd (insertAVL 3 it)
  +-1
+-2
  +-3

ghci> snd (insertAVL 4 it)
  +-1
+-2
  +-3
    +-4

ghci> snd (insertAVL 5 it)
  +-1
+-2
  | +-3
  +-4
    +-5

ghci> snd (insertAVL 6 it)
    +-1
  +-2
  | +-3
+-4
  +-5
    +-6
```

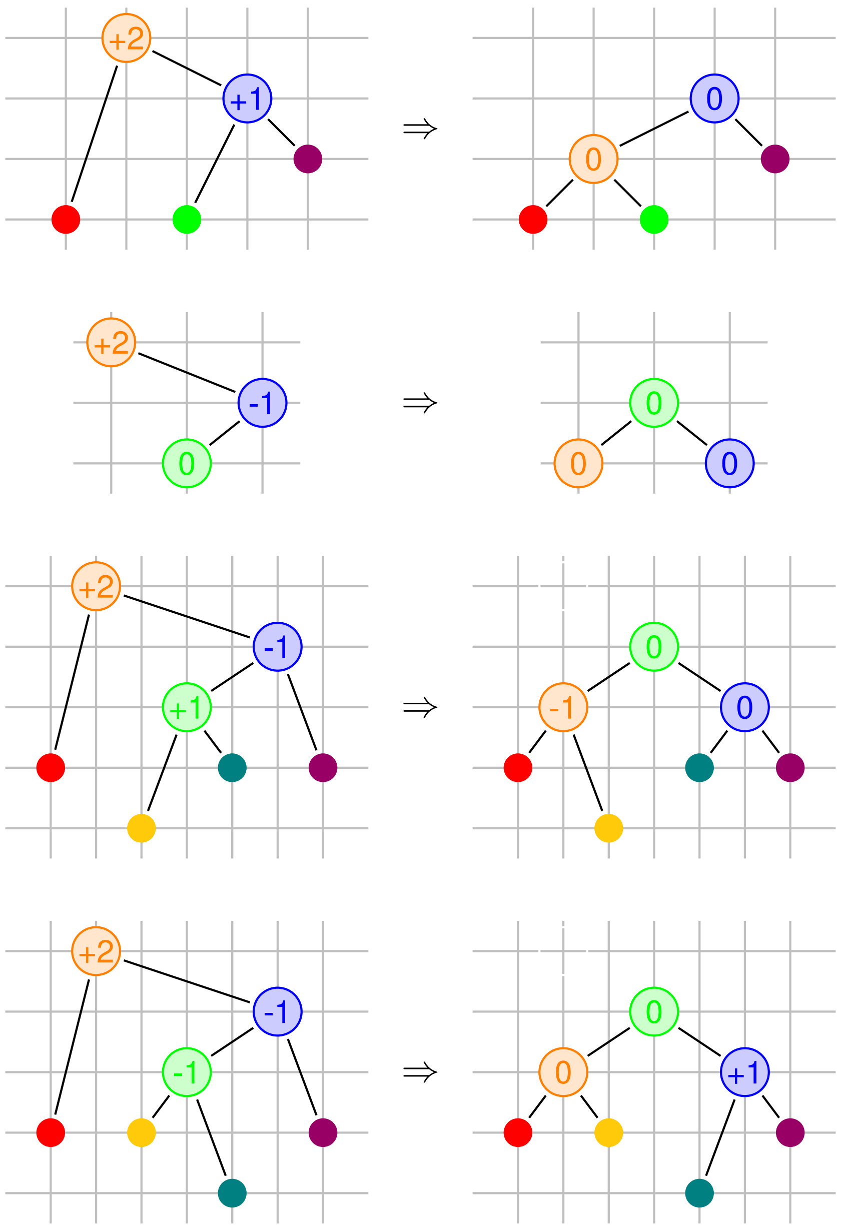


Figure 1: Rotations for balance factor (+2)