COMP 212 Spring 2015
Lab 4

The goal for the this lab is to make you more comfortable writing functions that operate on trees.

1 Tree Recursion

Recall the definition of trees from lecture:

> A tree is either
>
> - `Empty`, or
> - `Node(l,x,r)` where `l` is a tree, `x` is an `int`, and `r` is a `tree`.
>
> and that's it!

This allows us to `case` on a `tree` like so:

```
case t of
    Empty => ...
  | Node (l, x, r) => ...
```

and in the `Node` case, we will usually make recursive calls on `l` and `r`.

1.1 Depth

Intuitively, the depth of a tree is the length of the longest path from the root to a leaf. More precisely, we define the depth of a tree inductively: the depth of `Empty` is 0; the depth of `Node`($\mathtt{l}, \mathtt{x}, \mathtt{r}$) is one more than the larger of the depths of its two children `l` and `r`.

Task 1.1 Define the function

```
depth : tree -> int
```

that computes the depth of a tree.

Hint: You will probably find the function `max : int * int -> int`, which we have provided for you, useful.

1.2 Tree to List

Task 1.2 Define a function `treeToList`, which converts a tree to a list "in order". This means that the contents of the left subtree should come before the middle data, which should come before the contents of the right subtree. For example:

```
treeToList (Node(Node(Empty,1,Empty),
                 2,
                 Node(Empty,3,Empty)))
== [1,2,3]
```

2 Lists to Trees

For testing, it is useful to be able to create a tree from a list of integers. To make things interesting, we will ask you to return a *balanced* tree: one where the depths of any two leaves differ by no more than 1.

Task 2.1 Define the function

```
listToTree : int list -> tree
```

that transforms the input `list` into a balanced tree. *Hint:* You may use the `split` function provided in the support code, whose spec is as follows:

```
   If l is non-empty, then there exist l1,x,l2 such that
      split l == (l1,x,l2) and
      l == l1 @ x::l2 and
      length(l1) and length(l2) differ by no more than 1
```

3 Reverse

In this problem, you will define a function to reverse a tree, so that the in-order traversal of the reverse comes out backwards:

$$\texttt{treeToList (revT t)} \cong \texttt{reverse (treeToList t)}$$

Code

Task 3.1 Define the function

```
     revT : tree -> tree
```

according to the above spec.

Task 3.2 Explain why `revT` is total.

Have the TAs check your code for reverse before proceeding!

Analysis

Task 3.3 Determine the recurrence for the work of your `revT` function, in terms of the size (number of elements) of the tree. You may assume the tree is balanced.

Task 3.4 Use the tree method to write a closed form for the recurrence, in terms of a sum.

Task 3.5 Solve the sum (it should be one we have discussed previously in the course).

Task 3.6 Use the closed form to determine the big-O of $W_{\texttt{revT}}$.

Task 3.7 Determine the recurrence for the span of your `revT` function, in terms of the size of the tree. You may assume the tree is balanced.

Task 3.8 Use the tree method to give a closed form for this recurrence.

Task 3.9 Use the closed form to give a big-O for $S_{\texttt{revT}}$.

Correctness

Prove the following:

Theorem 1. *For all values* `t : tree`*,* `treeToList (revT t)` $\cong$ `reverse (treeToList t)`*.*

You may use the following lemmas about `reverse` on lists:

- `reverse []` $\cong$ `[]`
- For all valuable expressions l and r of type `int list`,

  `reverse (l @ (x::r))` $\cong$ `(reverse r) @ (x::(reverse l))`

In your justifications, be careful to prove that expressions are valuable when this is necessary. Follow the template on the following page.

Case for `Empty`
To show:

Case for `Node(l,x,r)`
Two Inductive hypothes**es**:

To show: