COMP 212 Spring 2025
Lab 5

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

1 Tree Recursion

In class, we saw that mergesorting lists doesn't have a very good span ($O(n)$), because splitting and merging lists are entirely sequential operations. This motivates representing sequences of data as trees instead of as lists.

We represent a tree as follows:

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!

Task 1.1 Draw a diagram of the tree

```
val t0 = Node( Node(Empty,3,Node(Empty,1,Empty)),
               7,
               Node(Node(Empty, 12, Empty), 2, Empty))
```

Task 1.2 The *root* of a tree is the value at the top. What is the root of `t0`?

Task 1.3 The *size* of a tree is the number of numbers in it. What is the size of `t0`?

Task 1.4 The *depth* of a tree is greatest number of non-Empty Nodes on a path between the root and an Empty (including the root but not counting the Empty). What is the depth of `t0`?

Task 1.5 A tree is *balanced* (roughly) when its depth is as small as possible given its size. More precisely, we can say that a tree is balanced when the depths of any two Emptys differ by no more than 1. Is `t0` balanced? Draw some other trees with the same elements that are less balanced.

Writing functions on trees We can use `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`.

For example, a function that computes the size of a tree is defined like this:

```
(* Purpose: compute the number of elements in the tree *)
fun size (t : tree) : int =
    case t of
        Empty => 0
      | Node (L, x, R) => 1 + size L + size R
```

Here is a function for adding one to each element of a tree, producing a new tree:

```
(* Purpose: add one to each element in a tree *)
fun addone (t : tree) : int =
    case t of
        Empty => Empty
      | Node (L, x, R) => Node( addone L, x+1, addone R)
```

1.1 Depth

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

Task 1.6 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 1.7 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]
```

Hint: remember that the `append` function on lists is built-in and called `@`.

Have us check your code before proceeding!

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 Emptys 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
```

Have us check your code before proceeding!

3 Reverse

In this problem, you will define a function to reverse a tree, so that the to-list 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.

Have us check your code for reverse before proceeding!

Analysis

Task 3.2 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 3.3 Give a closed form and big-O bound for $W_{\texttt{revT}}$.

Task 3.4 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 3.5 Give a closed form and a big-O bound for $S_{\texttt{revT}}$.

Have us check your analysis before proceeding!

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))`

When we do induction on a tree, we do a case for `Empty` and a case for `Node(l,x,r)`. In the `Node` case, we can assume **two** inductive hypotheses, which say that the theorem holds for `l` **and** that the theorem holds for `r`.

Case for `Empty`
To show:

Case for `Node(l,x,r)`
Inductive hypothesis for `l`:
Inductive hypothesis for `r`:

To show:

Have us check your proof!