Recitation Session, October 11th, 2017

Please do not write on this sheet of paper

And do not use laptops during the session

This week we will play with genericity and OO concepts.
A binary search tree is a binary tree such that, for a node, all elements in the left sub-tree are smaller than the element at the node, and all elements in the right sub-tree are greater than the element at the node. As such, there are therefore no two elements of a binary search tree that are equal.

Because we want to build a generic tree structure, we also need the notion of a comparator, or a less-than-or-equal operator (denoted as `leq`) for two generic elements which satisfies the following properties:

- Transitivity: `leq(a, b) && leq(b, c) ⇒ leq(a, c)`
- Reflexivity: `leq(a, a)` is `true`.
- Anti-symmetry: `leq(a, b) && leq(b, a) ⇒ a == b`
- Totality: either `leq(a, b)` or `leq(b, a)` is `true` (or both)

Note that the above defines a total order.

Here is the structure we will be using for implementing these trees:

```
abstract class Tree[T] { … }
case class EmptyTree[T](leq: (T, T) => Boolean) extends Tree[T] { … }
case class Node[T](left: Tree[T], elem: T, right: Tree[T],
    leq: (T, T) => Boolean) extends Tree[T] { … }
```

For consistency, all subtrees must contain the same `leq` parameter.
Creating an empty binary tree for integers can be then done as follows:

```
val intLeq = (x: Int, y: Int) => x <= y
val emptyIntTree: Tree[Int] = new EmptyTree(intLeq)
```

Ex 0: Given only `leq` for comparison, how can you test for equality? How about strictly-less-than?

Ex 1: Define the `size` method on `Tree[T]`, which returns its size, i.e. the number of `Nodes` in the tree.

```
abstract class Tree[T] {
  def size: Int
  ...
}
```

Implement it in two ways:

- within `Tree[T]`, using pattern matching,
- in the subclasses of Tree[T].

Ex 2: Define the `add` method, that adds an element to a `Tree[T]`, and returns the resulting tree

```
def add(t: T): Tree[T] = ???
```

Remember that trees do not have duplicate values. If `t` is already in the tree, the result should be unchanged.

Ex 3: Define the function `toList`, which returns the sorted list representation for a tree. For example, `emptyIntTree.add(2).add(1).add(3).toList` should return `List(1, 2, 3)`

```
def toList: List[T] = ???
```

You can use the `Nil` operator for creating an empty list, and the `::` operator for adding a new element to the head of a list: `1 :: List(2,3) == List(1,2,3)`. You are naturally free to define any auxiliary functions as necessary.

Ex 4: Define the function `sortedList`, which takes an unsorted list where no two elements are equal, and returns a new list that contains all the elements of the previous list (and only those), but in increasing order.

```
def sortedList[T](leq: (T, T) => Boolean, ls: List[T]): List[T] = ???
```

Hint: you might need to define some auxiliary functions to help you with this.