Advanced Programming
Homework Assignment 3

Due Wednesday, February 6, at 3PM

Warm-up

1. Read chapters 9 and 12 of SOE.
2. Work problems 9.4, 9.5, 9.7, 12.2, and 12.3 from SOE. Note that 12.2 was not previously announced.
3. Review the lecture handout on persistent red-black trees.
4. Look up how AVL trees work, either in your favorite algorithms text or on the web. (The Wikipedia article is pretty good.)

Main assignment

The goal of this assignment is to write your own implementation of AVL trees in Haskell. A sample implementation of part of the functionality is provided on the course web page, but you are encouraged *not* to look at this implementation unless you get stuck: You'll get the most out of the assignment if you try each step yourself, from scratch, and peek at the sample implementation only as necessary.

1. Create a Haskell module Main with the following type declaration:

```
data AVL e = E           -- empty tree
           | N           -- non-empty tree
               Int           -- balance factor (right size - left size)
               (AVL e)       -- left subtree
               e             -- value
               (AVL e)       -- right subtree
  deriving Show
```

2. Build a few particular trees that you can use as test cases later—some that obey all of the AVL invariants...

```
t1 = ...
t2 = ...
t3 = ...
```

... and some others that do not...

```
bad1 = ...
bad2 = ...
bad3 = ...
```

3. Write a function

```
check :: (Ord e, Show e) => AVL e -> IO ()
```

that, given an `AVL` tree `e`, verifies that `e` satisfies the following properties:

(a) The branching factor at each node is correctly calculated. (You'll need a helper function `height` for this.)

(b) The branching factor at each node is between -1 and +1.

(c) The items stored in `e` are in strictly increasing order. (One easy way to do this is to define a helper function `avlToList` that returns a list of values encountered during an inorder traversal of `e` and another function that checks whether a list is in strictly increasing order.)

If any of these conditions fail, the `check` function should print out a message saying what is wrong.

4. Write a `main` action that uses `check` to test all of your example trees. Make sure that the good trees are accepted silently and that the bad ones are correctly identified.

   You can delete the calls to failing tests from `main` once this step is finished, so that your whole program prints nothing at all when it runs.

5. Write a function `rebalance` that takes a tree `e` whose root node has branching factor -2 or +2 and rearranges it to an equivalent tree with -1/0/+1 branching factors.

   For this step, you will probably find it helpful to have a good diagram to refer to. (I found the one on Wikipedia useful.) Note, though, that most explanations of AVL trees will talk about "rotating" the nodes near the root, which implies some sort of pointer manipulation. Here, we're simply rebuilding a completely new tree out of the pieces of the old one, so the notion of rotating doesn't really apply. In particular, you may find it easier (I did) to implement the "double rotations" that standard presentations of the algorithm talk about in a single step.

   Even so, a diagram that shows the effect such rotations are tryoing to achieve is a useful guide to implementing your rearrangement. I actually named the variables in my patterns to match the labels in the diagram I was looking at, and this made it very much easier to write the rearranged trees correctly.

   (For me, the only hard part of writing this function was figuring out the balance factor annotations for the rearranged tree. You may find a piece of paper helpful for figuring this out.)

6. Add testing code to your `main` function that uses `check` to verify that applying `rebalance` to all the different shapes of unbalanced trees yields correct results in each case.

7. Write a testing function that checks whether rebalancing a tree yields a tree that is not only well formed but also contains the same elements. (I am intentionally not telling you exactly how to do this, but one way is to extract the list of elements in each and compare these.)

8. Write a *stub function*

```
avlInsert :: Ord e => e -> AVL e -> AVL e
```

that simply returns its second argument unchanged.

9. Write a testing function

```
checkInsert :: Ord e => e -> AVL e -> IO (AVL e)
```

that uses `avlInsert` to insert a given element into a given tree, checks that the result is correct (i.e., that the new tree is well formed and that it contains the same set of elements as the old tree, plus the new element), prints a message if it finds anything wrong, and returns the new tree.

10. Now write a real implementation of `avlInsert` that inserts a new element into a tree and calls `rebalance` to rebalance the resulting tree if necessary. Add more checks until you are sure that it is working correctly.

11. *(Optional)* Write a function

    ```
    avlDelete :: Ord e => e -> AVL e -> AVL e
    ```

    that removes an element from a tree and rebalances the resulting tree as necessary.

    Write a testing function and use it to verify that this function works as expected.

Submission instructions

- Your `main` function should execute all of the tests that you've written for the various parts of the assignment. Since these should all succeed, the effect of running your program should be to print nothing.
- Include your solutions to the warm-up exercises in the same file.
- Put your name in a comment at the top of the file.

  Also, please put the approximate number of hours that you spent on this assignment (including reading).
- Email the file to both `jschorr@seas.upenn.edu` and `bcpierce@cis.upenn.edu`.