CS3200: Programming Languages
Homework 9: data types and functors in Haskell

BST module: For this assignment, you'll be implementing your own binary search tree data structure in Haskell. If you're a bit fuzzy about binary search trees, go review your data structures! Please make sure you understand this data structure before you start the assignment!!

However, be careful with google on this one - I will NOT accept code that is obviously a cut and past of posted code on the internet! You're allowed to look at online references, but that isn't the same thing as copying and pasting code and claiming it is your own. Come see me if you have any questions on this point, but be aware that the last time I assigned a data structure like this, almost half of my students verbatim copy and pasted (slightly incorrect) solutions from the internet, and got a 0 for the entire HW as a result.

Our start will be the following code, based on the book's binary tree class from your reading:

```
data  BSTree a = Empty
               | Node a (BSTree a) (BSTree a)
                      deriving (Show, Read)

singleton :: a -> BSTree a
singleton x = Node x Empty Empty

treeInsert :: (Ord a) => a -> BSTree a -> BSTree a
treeInsert x Empty = singleton x
treeInsert x (Node a left right)
    | x == a = Node x left right
    | x < a  = Node a (treeInsert x left) right
    | x > a  = Node a left (treeInsert x right)

treeElem :: (Ord a) => a -> BSTree a -> Bool
treeElem x Empty = False
treeElem x (Node a left right)
    | x == a = True
    | x < a  = treeElem x left
    | x > a  = treeElem x right

instance Functor BSTree where
    fmap f Empty = Empty
    fmap f (Node x leftsub rightsub) = Node (f x) (fmap f leftsub) (fmap f rightsub)
```

Once you load and play with this a bit, modify it and add the following (with proper signatures for each function):

1. Add the function `empty`, which returns True if the tree is empty and False if it has at least one value in it.

2. Add the function `size`, which returns the number of Nodes in the tree (not counting Empty).

3. Add the function `height`, which computes the height of the tree.

4. Make the BSTree class an instance of Show, but (instead of just deriving and printing the big messy string) have it print an inorder traversal of the tree (so that it shows up in sorted order, either as a list or a string).

5. Add the function `printLevel`, which takes an Int k and a BSTree and returns a list that contains every node which has height k in the tree, in sorted order. If k is larger than the height of the tree, print an appropriate error message.

6. Add the functions `pivotLeft` and `pivotRight`, which take a BSTree and pivot at the root so that the root moves down (to the left or the right, respectively) and is replaced by its other child. (See here for a snazzy animation: https://en.wikipedia.org/wiki/Tree_rotation, if you need a reminder of how this works.) Your function should return a new BSTree with the root and children rearranged appropriately. If the funcitons are called on an empty tree or a tree with no appropriate child to rotate up to the root's spot, raise an appropriate error message.

7. Modify insert so that your BST is actually an AVL tree, where no node in the tree is unbalanced after inserting. (See https://en.wikipedia.org/wiki/AVL_tree, or https://www.cs.usfca.edu/~galles/visualization/AVLtree.html for an interactive visualization of how insert would work for this.) Note that you do not have to implement deletion - just modify the insert to work as shown, probably using your pivot functions.

8. Now write a main to test your function; accept inputs from a prompt, create some trees and print them, and add/pivot/whatever interactively to demo your data structure.

9. Extra credit: : Modify your BSTree class so it is also of type Applicative, and define $<*>$ appropriately.