Section #7

Tyler Conklin and Kate Rydberg (Based on handouts by previous CS106B instructors and TAs, especially those of Marty Stepp and Julie Zelenski.)

Trees

*This week is all about **binary trees**. The recursive structure of trees makes writing recursive functions very natural. Each function we write will accept a pointer to the root of the tree, along with any other needed parameters. You may define additional helper functions as needed to implement your behavior. Remember that you must not leak memory. In this handout, the TreeNode structures we will use will look like this:*

```
struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;
};
```

```
struct TreeNodeChar {
    char data;
    TreeNodeChar* left;
    TreeNodeChar* right;
};
```

```
struct TreeNodeString {
    char data;
    TreeNodeString* left;
    TreeNodeString* right;
};
```

1. Binary Search Tree Insertion

Draw the binary search tree that would result from inserting the following elements in the given order. Here's the alphabet in case you need it :) `ABCDEFGHIJKLMNOPQRSTUVWXYZ`

a) Leia, Boba, Darth, R2D2, Han, Luke, Chewy, Jabba

b) Meg, Stewie, Peter, Joe, Lois, Brian, Quagmire, Cleveland

c) Kirk, Spock, Scotty, McCoy, Chekov, Uhuru, Sulu, Khaaaan!

2. Height

Write a function called height that accepts a pointer to a `TreeNode` and returns the height of the tree. The height is defined to be the number of levels (i.e., the number of nodes along the longest path from the root to a leaf). For example, an empty tree has height 0. A tree of one node has height 1. A node with one or two leaves as children has height 2, etc.

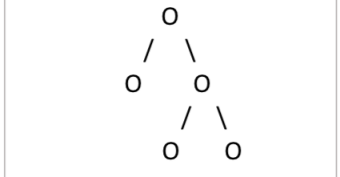
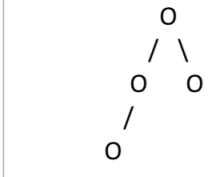
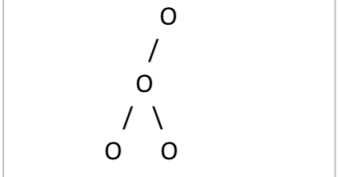
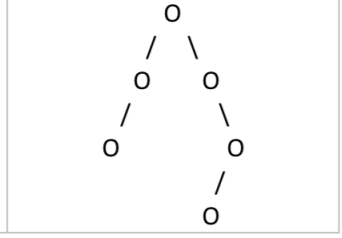
3. Find Min

Write a function called `findMin` that accepts a pointer to a `TreeNode` (which is the root of a binary search tree) and returns the minimum value in the tree. You may assume the tree passed to your function is not empty.

Follow-up question: where would you have to look for the minimum value if the tree was just a binary tree, not a binary search tree?

4. Is Balanced

Write a function `isBalanced` that accepts a pointer to a `TreeNode` and returns whether or not a binary tree is balanced. A tree is balanced if its left and right subtrees are also balanced trees whose heights differ by at most 1. The empty (null) tree is balanced by definition. You may call solutions to other section exercises to help you.

balanced	balanced	not balanced	not balanced
			

5. Is BST

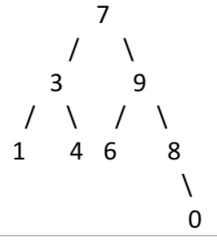
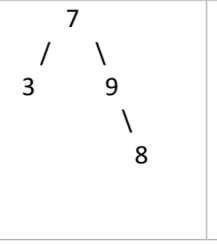
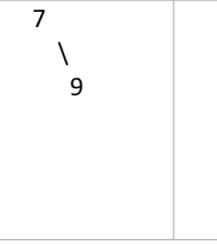
Write a function `isBST` that accepts a pointer to a `TreeNode` and returns whether or not a binary tree is arranged in valid binary search tree (BST) order. Remember that a BST is a tree in which every node `n`'s left subtree is a BST that contains only values less than `n`'s data, and its right subtree is a BST that contains only values greater than `n`'s data. You may assume that the BST does not have any duplicate values.

This is tricky! Hint: if a tree is a BST and you traverse all of its nodes in-order, their values should be in sorted order.

6. Remove Leaves

Write a function `removeLeaves` that accepts a reference to a pointer to a `TreeNode` and removes the leaf nodes from a tree. A leaf is a node that has empty left and right subtrees.

If `t` is the tree on the left, `removeLeaves(t)` should remove the four leaves from the tree (the nodes with data 1, 4, 6, and 0). A second call would eliminate the two new leaves in the tree (the ones with data values 3 and 8). A third call would eliminate the one leaf with data value 9, and a fourth call would leave an empty tree because the previous tree was exactly one leaf node. If your function is called on an empty tree, it does not change the tree because there are no nodes of any kind (leaf or not). You must free the memory for any removed nodes.

Before call	After 1st call	After 2nd call	After 3rd call	After 4th call
			7	nullptr