Lab 4 CS1: Rebalancing a Binary Search Tree (total 100pts)

In class, you were exposed to Binary Search Trees (BST) and saw the benefits of such structure, particularly during searching. However, you also saw that a tree works best when it is balanced; otherwise its behavior ends up being the same as a LinkedList. For example, a balanced BST can find if an element is contained within it in $\log_2(n)$ in contrast of n comparisons required with the LinkedList. This is not a fundamental difference when we talk about a small number of elements, for instance 4 elements would only take 2 comparisons in a BST, which is not a huge difference. However, when considering a large number like 1024, it would only take 10 comparisons.

Rebalancing a Tree

There are many ways to rebalance a BST, to name a few Red-Black Trees (http://en.wikipedia.org/wiki/Red%E2%80%93black_tree) and AVL Trees (http://en.wikipedia.org/wiki/AVL_tree). This lab asks for a different approach, not as effective as a self-balance tree, but a good way to learn your way around trees. It is still advice that you go and at least understand how the pseudocode of these two trees work, as they a popular way of implementing a tree.

What you need to build

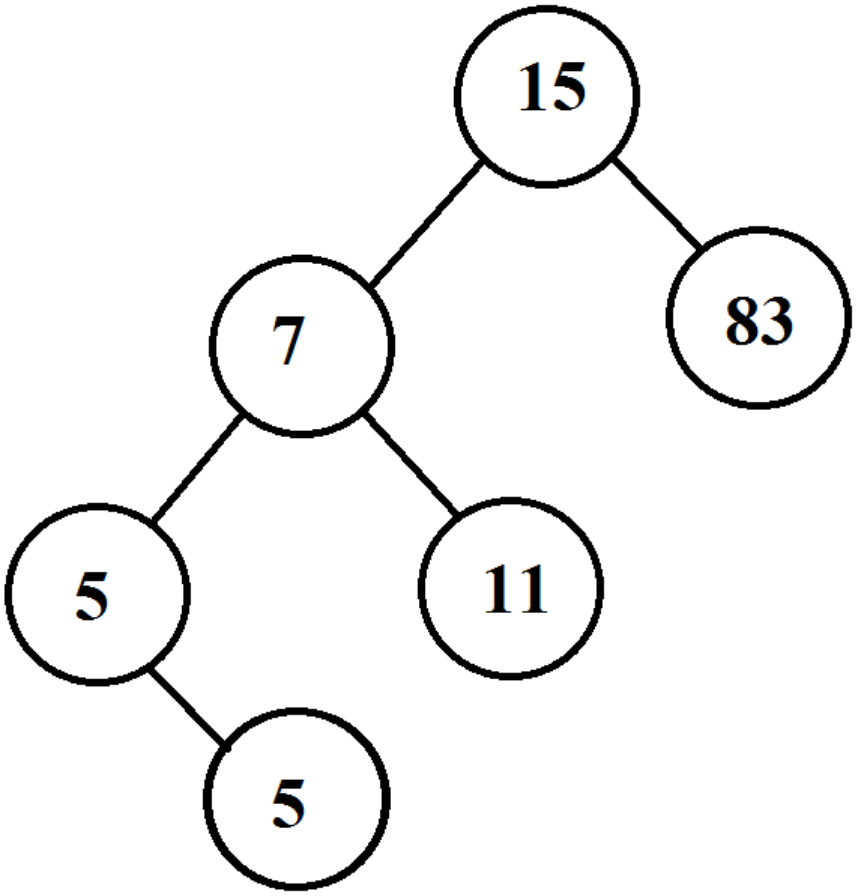
Create a function that balances a BST of type BSTree as shown in Trees.h library. The function prototype is as follows:

```
/*
        @param tree the tree that will be rebalanced
        @return the rebalanced tree is actually returned as a rebalanced copy, so the tree parameter is
                kept intact. If user has no need for this unbalanced tree, it is responsible for releasing it.
*/
BSTree* rebalanceBST(BSTree* tree);
```

To complete the lab you will need the following **(the items in bold need to use recursion)**:

1. 10 pts – Your tree must be generic (hold any data type with the same code). Write comments a header comment with the people who worked on the lab (it can be up to 4 as long as the names are written).
2. 20 pts – bool isBalanced(BSTree* tree); returns true if the tree is balanced (the difference in height from left and right is -1, 0, or 1).
3. 20 pts – ArrayList* BST2Array(BSTree* tree); get all elements in the tree IN ORDER and store them in an array
4. **20 pts – BSTree* allocBSTBalanced(ArrayList* sortedElements); Returns a balanced tree containing all elements in the sortedElements array.**
5. 10 pts – Release the temp ArrayList* and return the tree created at step 2. This step is the merge of steps 2 and 3, ultimately the body of **rebalanceBST**.
6. **20 pts – void releaseBST(BSTree* tree); a function for releasing the nodes in a tree … HINT: use POST ORDER traversal.**

Example of Execution



Unbalanced tree (height = 3). You can get this one by inserting {15, 7, 83, 5, 11, 5}

Calling BST2Array returns {5, 5, 7, 11, 15, 83}

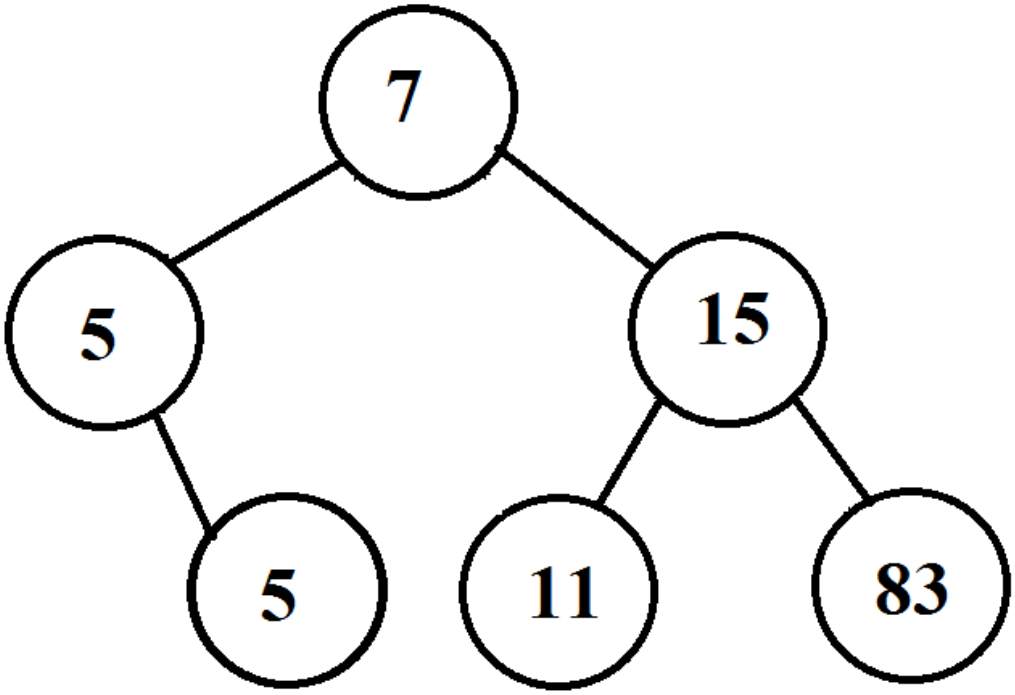
Then inserting the elements in a new BSTree can make it balanced if they are inserted as follows:

- Cut the array in half insert the mid-point into the tree
- Do the same thing with the two half arrays
- Repeat for each half that gets created (left and right)

Example of mid-point insertion:

List	Mid-point (floor(n/2))	Left	Right
{5, 5, 7, 11, 15, 83}	7	{5, 5}	{11, 15, 83}
{5, 5}	5	{5}	NULL
{11, 15, 83}	15	{11}	{83}
{5}	5		
{11}	11		
{83}	83		

If you insert the mid-points in the order that you find them ({7, 5, 15, 5, 11, 83}, you get:



Balanced tree (height = 2)