CS 106B Section 7 (Week 8)

This week is all about binary trees. Remember that the recursive structure of trees makes writing recursive methods for them very natural.

Recommended problems: #3, #5, #6, #7

1. Traversals.
Write the elements of each tree below in the order they would be seen by a pre-order, in-order, and post-order traversal

a)

```
        3
      /   \
     5     2
    /     / \
   1     4   6
```

b)

```
          19
         /  \
       47    63
      /  \     \
    23   -2     94
         /        \
       55          28
```

c)

```
         2
          \
           1
          / \
         7   6
        /     \
       4       9
      / \     /
     3   5   8
```

2. BST Insertion
Draw the binary search tree that would result from inserting the following elements in the given order.

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!

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


class BinaryTree {
public:
    member functions;
private:
    TreeNode* root; // NULL if empty
};
```

*For each coding problem, you are to write a new public member function for the `BinaryTree` class from lecture that performs the given operation. You may define additional private functions to implement your public members. For functions that remove nodes, remember that you **must not leak memory**. You can assume that there is a helper function `deleteTree` that frees all memory associated with a given subtree.*

3. height.

Write a member function called **height** that returns the height of a 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.

4. countLeftNodes.

Write a member function **countLeftNodes** that returns the number of left children in the tree. A left child is a node that appears as the root of the left-hand subtree of another node. For example, the tree in Problem 1 (a) above has 3 left children (the nodes storing the values 5, 1, and 4).

5. isBalanced.

Write a member function **isBalanced** that 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

      8               4                 8                4
    /   \           /   \              /               /   \
   4     9         3     9            4               3     9
        / \       /                  / \             /       \
       4   6     1                  2   7           1         5
                                                             /
                                                            2
```

6. removeLeaves.

Write a member function **removeLeaves** that removes the leaf nodes from a tree. A leaf is a node that has empty left and right subtrees. If a variable t refers to the tree below at left, the call of t.removeLeaves(); 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). Free the memory for any removed nodes.

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

`      7`
`    /   \`
`  3       9`
` / \     / \`
`1   4   6   8`
`              \`
`               0`
	`    7`
	`  /   \`
	`3       9`
	`         \`
	`          8`
		`7`
		`  \`
		`   9`
			7	NULL

7. completeToLevel.

Write a member function **completeToLevel** that accepts an integer k as a parameter and adds nodes with value -1 to a tree so that the first k levels are complete. A level is complete if every possible node at that level is non-NULL. We will use the convention that the overall root is at level 1, its children are at level 2, and so on. Preserve any existing nodes in the tree. For example, if a variable called t refers to the tree below and you make the call of t.completeToLevel(3); you should fill in nodes to ensure that the first 3 levels are complete. Notice that level 4 of this tree is not complete. Keep in mind that you might need to fill in several different levels. You should throw an integer exception if passed a value for *k* that is less than 1.

Before call	After call

`         17`
`       /    \`
`     83       6`
`    /          \`
`  19            87`
`    \          /`
`     48      75`
	`          17`
	`        /    \`
	`      83       6`
	`     /  \     / \`
	`   19   -1  -1   87`
	`    \             /`
	`     48         75`

CS 106B Section 7 (Week 8)

8. tighten.

Write a member function **tighten** that eliminates branch nodes that have only one child. For example, if a variable t stores the tree below at left, the call of t.tighten(); should leave t storing the tree at right. The nodes that stored 28, 19, 32, and -8 have been eliminated because each had one child. When a node is removed, it is replaced by its child. This can lead to multiple replacements because the child might itself be replaced (as in 19 which is replaced by 32, replaced by 72). Free memory as needed

```
              Before call                              After call

                  12                                       12
                 /  \                                     /  \
              28      19                               94      72
              /       /                               /  \    /  \
           94       32                              65    10 42    50
          /  \        \
        65    -8       72
                \     /  \
                10   42   50
```

9. limitPathSum.

Write a member function limitPathSum that accepts an integer value representing a maximum, and removes tree nodes to guarantee that the sum of values on any path from the root to a node does not exceed that maximum. For example, if variable t refers to the tree below at left, the call of t.limitPathSum(50); will require removing node 12 because the sum from the root down to that node is more than 50 (29 + 17 + -7 + 12 = 51). Similarly, we have to remove node 37 because its sum is (29 + 17 + 37 = 83). When you remove a node, you remove anything under it, so removing 37 also removes 16. We also remove the node with 14 because its sum is (29 + 15 + 14 = 58). If the data stored at the root is greater than the given maximum, remove all nodes, leaving an empty (NULL) tree. Free memory as needed, but only remove the nodes that are necessary to remove.

```
              Before call                              After call

                  29                                       29
                 /  \                                     /  \
              17      15                               17      15
             /  \    /  \                              /       /
           -7   37  4    14                          -7       4
          /  \    \     /  \                         /
        11    12   16  -9   19                     11
```