CS 106B Section 7 Handout (Week 8)

Binary Search Trees

1. **BST insertion.** Draw the binary search tree (BST) 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.*

2. **`isBST`.** Write a member function `isBST` that returns whether or not a binary tree is arranged in valid binary search tree (BST) order. 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.

3. **`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 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
    8
   / \
  4   9
 / \
2   7
```

```
balanced
    4
   / \
  3   9
 /
1
```

```
not balanced
    8
   /
  4
 / \
2   7
```

```
not balanced
    4
   / \
  3   9
 /     \
1       5
       /
      2
```

4. **`toString`.** Write a member function `toString` that returns a string representation of the binary tree in a particular format. Return `"/"` for an empty tree. For a leaf node, it should return the data in the node as a string. For a branch node, it should return a parenthesized string that has three elements separated by commas: the data at the root followed by a string representation of the left subtree followed by a string representation of the right subtree. For example, if a variable `t` stores a reference to the following tree, then the call `t.toString()` should return: `"(2, (8, 0, /), (1, (7, 4, /), (6, /, 9)))"`

```
    2
   / \
  8   1
 /   / \
0   7   6
   /     \
  4       9
```

5. `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.