CS106B
Spring 2016 Cynthia Lee

Section 5 (Week 6) Handout

Section problems by Marty Stepp, with edits by Cynthia Lee

Binary Tree Reference:

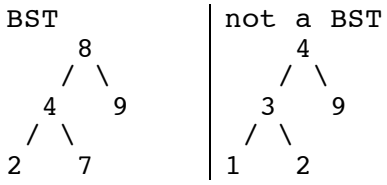
```
struct TreeNode {
    int data;
    TreeNode* left;
    TreeNode* right;
    ...
};

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

Binary Tree Member Functions

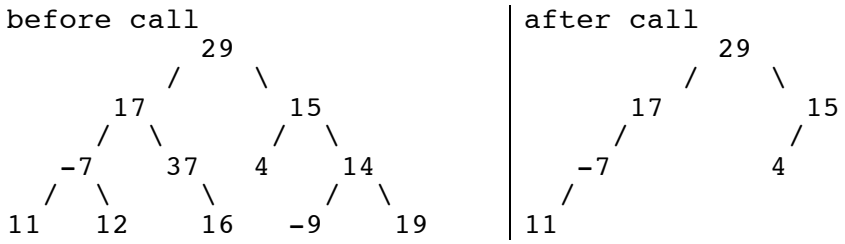
Each of the following problems (except #7) asks you to add a member function to the `BinaryTree` class from lecture. In all cases, if your function deletes a node from the tree, free the associated memory for the node.

1. **height.** Write a member function `height` that returns the height of a tree. The height is defined to be the number of edges 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 is a tree of height 2, etc.

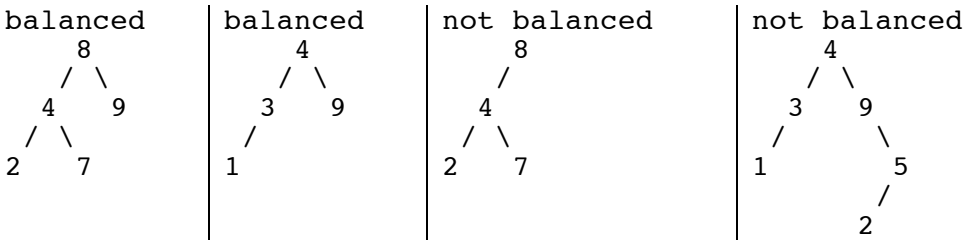
2. **isBST.** Write a member function `isBST` that returns whether or not a binary tree is arranged in valid binary search tree (BST) order. Remember that a BST is a tree in which every node *n*'s left subtree is a BST that contains only values less than *n*'s data, and its right subtree is a BST that contains only values greater than *n*'s data.



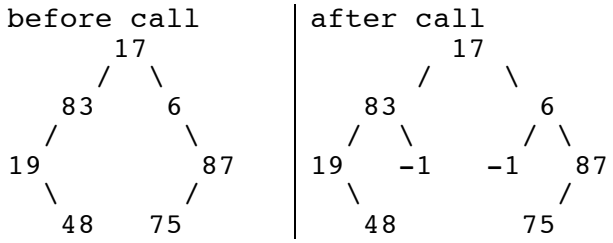
3. **limitPathSum.** Write a member function `limitPathSum` that accepts an integer value representing a maximum, and removes tree nodes to guarantee that the sum of values on any path from the root to a node does not exceed that maximum. For example, if variable `t` refers to the tree below at left, 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 nodes when necessary.



4. **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.



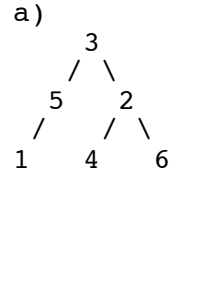
5. **completeToLevel.** Write a member function `completeToLevel` that accepts an integer *k* as a parameter and adds nodes with value -1 to a tree so that the first *k* levels are complete. A level is complete if every possible node at that level is non-`NULL`. We will use the convention that the overall 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.



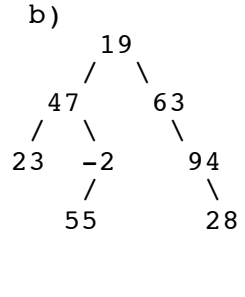
6. **countLeftNodes.** Write a member function `countLeftNodes` that returns the number of left children in the tree. A left child is a node that appears as the root of the left-hand subtree of another node. For example, the tree in Problem 7 (a) below has 3 left children (the nodes storing the values 5, 1, and 4).

7. **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)



b)



c)