CS 2420 Program 2
Recursion Practice

Objective: The starter code contains the tests of each of the methods you are to write. Test them one at a time as you write them. Feel free to modify prototypes any way you like, but include the same tests as given in the starter code. All trees are trees of integers, so you don't need to retain the generic feature of the code. Some trees will be Binary Search Trees (BST) and some are not. Make sure you don't assume it is a BST unless specified. You will notice in the starter code I have public helper functions which make it possible to call a routine without knowing the root but have recursive "worker" routines that depend on knowing the current node. The code you write MUST BE your own work. Do not copy from anywhere.

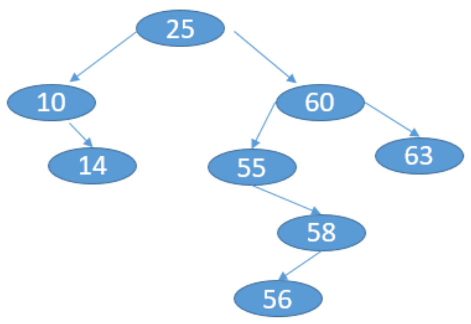
NOTE: In the comments to each function, provide a big-Oh expression for the complexity of the functions you write, assuming trees are roughly balanced (depth = log(n) for n nodes). Use recursion where appropriate, but if something isn't logically recursive, don't use recursion.

Identifying the big-Oh is worth two points.

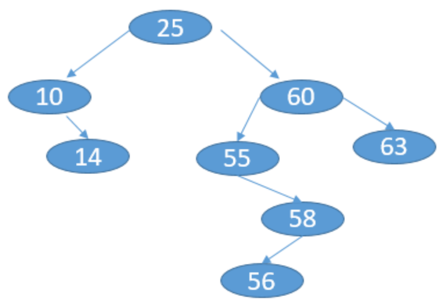
The starter code is designed to test all the methods. Get one working before moving on to the next. When testing, it is easiest if you comment out other methods.

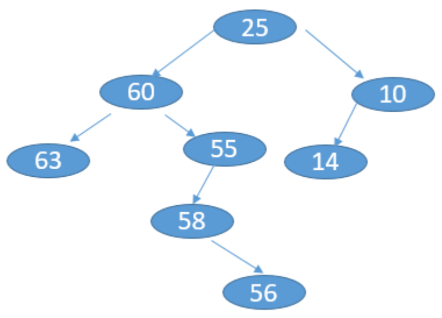
1. (3 points) Write a function, toString(), that returns a string containing: the tree name and the keys (in order) of a binary tree, given the root. Note, toString2 (which prints a flattened view of the tree) is provided.
   Tree1:

```
    63
  60
      58
        56
    55
25
    14
  10
```

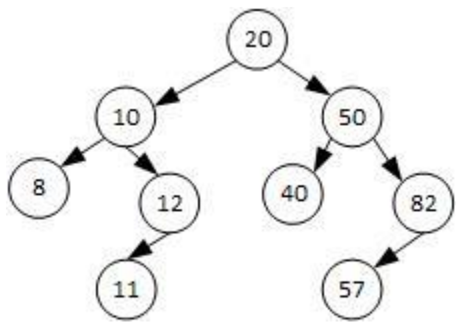


2. (3 point) Write the function flip() to swap left and right children recursively. The tree on the right is a flipped version of the tree on the left.





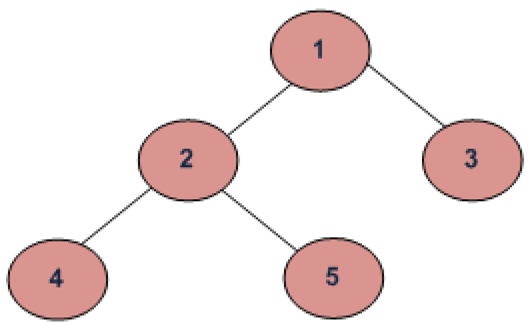
3. (3 points) Write a routine to find the deepestNode. The deepest node of a tree is a node at the maximum level. In the tree below, 11 (or 57) would be a deepest node. If a node has more than one deepest node, print one of them.



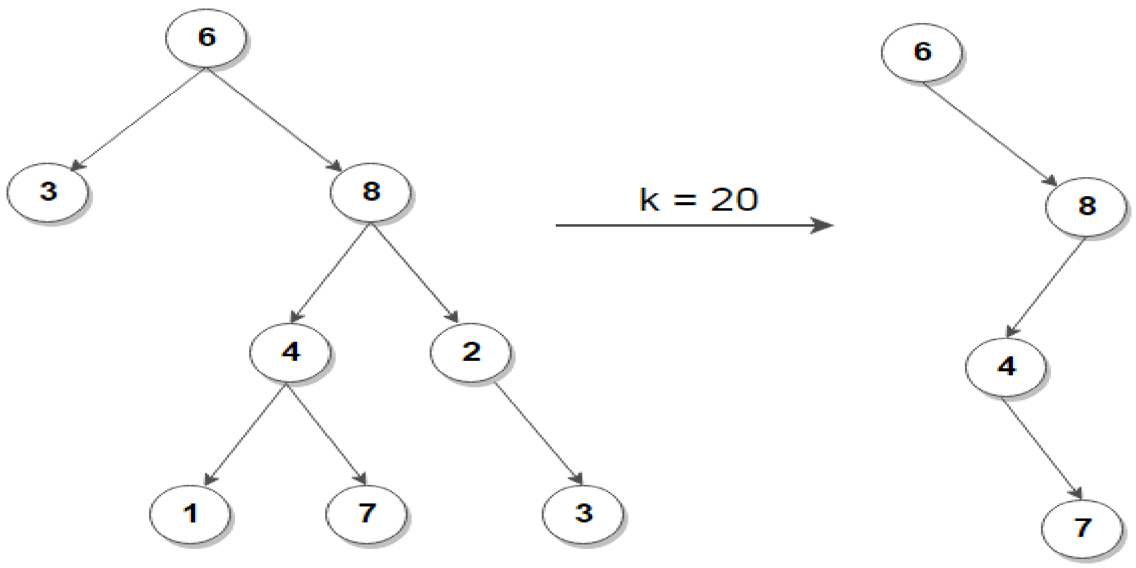
4. (3 points) Write the function *nodesInLevel(level)* that returns the total number of nodes on the specified level. For this problem, the root is at level zero, the root's children are at level one, and, for any node, the node's level is one more than its parent's level.

5. (3 points) Given a binary tree, print out all of its root-to-leaf paths one per line.
For the tree below, printAllPaths() would generate the following output:

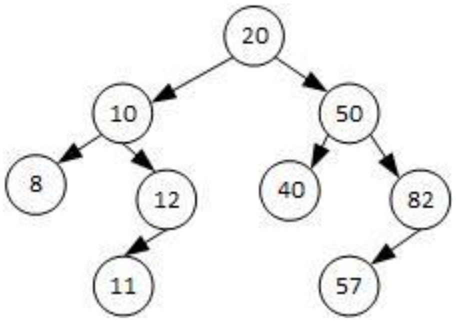
```
1 2 4
1 2 5
1 3
```



6. (3 points) Given a number k, pruneK(int k) removes nodes from the tree which are not part of a path having sum greater than or equal to k. For example.

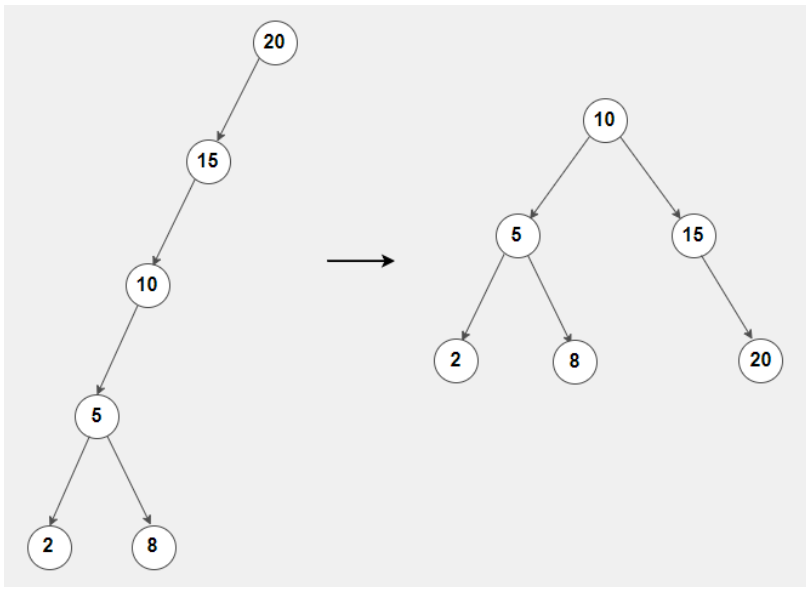


7. (3 points) Write a function lca(int a, int b) which returns the least common ancestor of two nodes in a binary search tree. Both nodes must exist in the tree. A least common ancestor is an ancestor of both nodes and is closest to the nodes. A node is considered to be an ancestor of itself. In the tree below, the least common ancestor of 82 and 8 is 20. The least common ancestor of 57 and 50 is 50. The least common ancestor of 8 and 11 is 10.

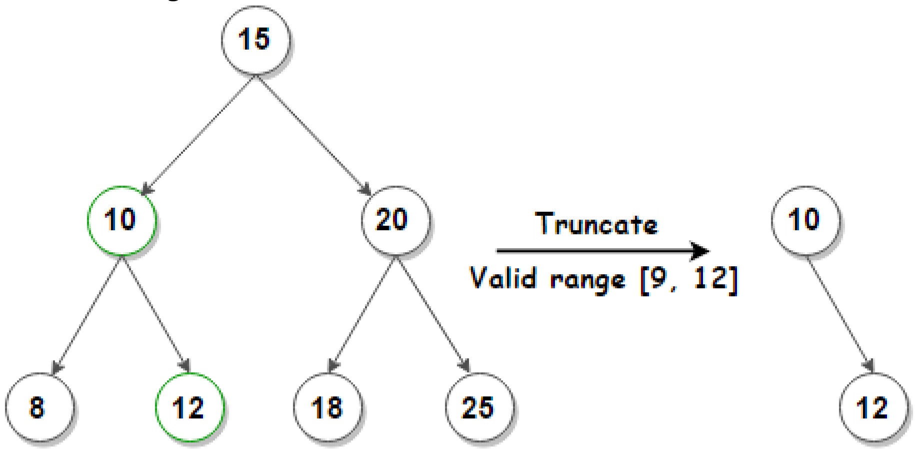


8. (3 points) balanceTree() constructs a height balanced binary search tree (BST) from an unbalanced BST. Hint, we are not balancing via rotations, but just starting over.

For example:



9. (3 points) keepRange(int first, int last) removes nodes from a BST which have keys outside a valid range.



10. (3 points) countBST() counts the number of Binary Search Trees present in a Binary Tree. Hint, you may need more passed back than just the number of BST trees in the subtree.

```
    11
   /  \
  8     10
 /     /  \
5    9     3
/ \
4   6
```

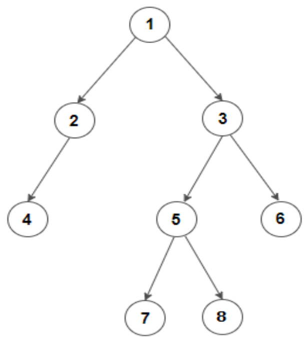
countBST: 6 (Note, you only need to output the number of BST. This is counting the trees rooted at 4,6,5,8,9,3, which I have circled in purple).

Bonus Problem

11. (3 points) Create a tree from its inorder and preorder listing. The tree is not necessarily a BST. You may assume all elements are unique. For example, buildTreeFromTraversals(Integer[] inorder, Integer[] preorder) for the following tree:

Inorder: 4 2 1 7 5 8 3 6
Preorder: 1 2 4 3 5 7 8 6

Output:



Hints: You have been given the public prototypes for each function. Feel free to adjust the private prototypes as you see fit. For example, my private version of countBST looked something like:

```
      private BSTInfo countBST (BinaryNode node)
Where BSTInfo contains all the information I need to know about the BST.
```