**CS 2420 Program 2 – 28 points  
Spring 2020**Recursion Practice

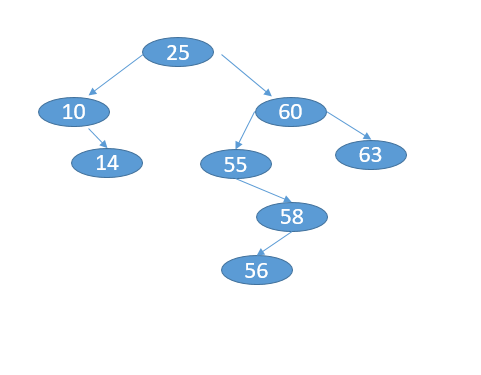
**Objective:** The starter code contains the tests of each of the methods you are to write. **T**est them one at a time as you write them.Feel free to modify prototypes any way you like, but include the same tests as shown. All trees are trees of integers, so you don’t need to retain the generic feature of the code. Some trees will be 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.

Documentation of code and identifying the big-Oh are worth four points. Consult the style guidelines.

Build the trees as shown in the starter code.

1. (2 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. The parent of each node is indicated in square brackets.

Tree1:

63 [60]

60 [25]

58 [55]

56 [58]

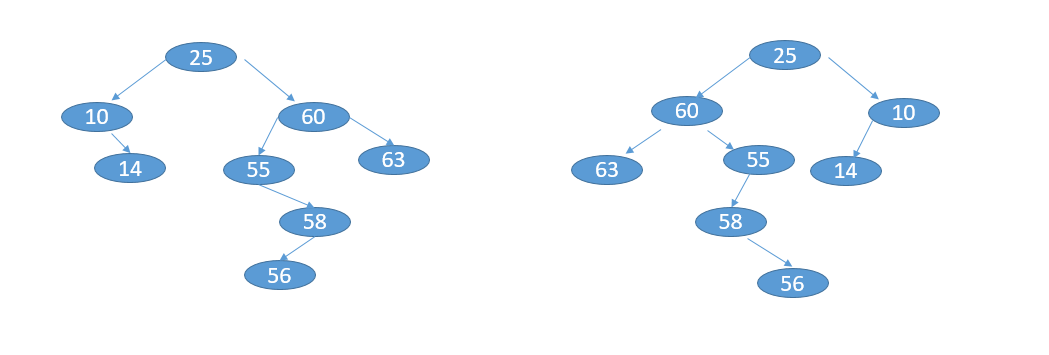
55 [60]

25 [no parent]

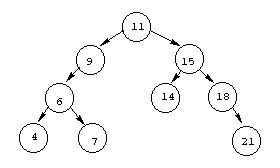
14 [10]

10 [25]

1. (2 point) Write the function flip() to swap left and right children recursively.



1. (2 points) Write a function to find the inorder successor() of a node (given the last node accessed in the BST tree). You should use the right child to find the successor, but an inorder successor is not the same thing as a child. This function can be written in fewer than 10 lines of code – but that's a guideline, not a requirement. Note, for each call, we find the successor of given a node. This needs to use a routine that can find a successor from a specific node (without having to traverse the tree from the root). In the tree below, the successor(11) is 14. The successor(7) is 9. The successor(18) is 21.



1. (2 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.
2. (2 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



1. (2 points) Given a Binary Tree, the task is to print the nodes byLevelZigZag(int level) up to level. Nodes on even levels are printed right to left. Nodes on odd levels are printed left to r ight.

Input:

1

/ \

2 3

/ \ \

4 5 6

Output byLevelZigZag(2)= : 6 5 4 2 3 1

Output byLevelZigZag(1) = 2 3 1

Input:

5

/ \

9 3

/ \

6 4

/ \

8 7

Output byLevelZigZag(3): 8 7 4 6 9 3 5

1. (2 points) countBST() counts the number of Binary Search Trees present in a Binary Tree

*11*

*/ \*

*8 10*

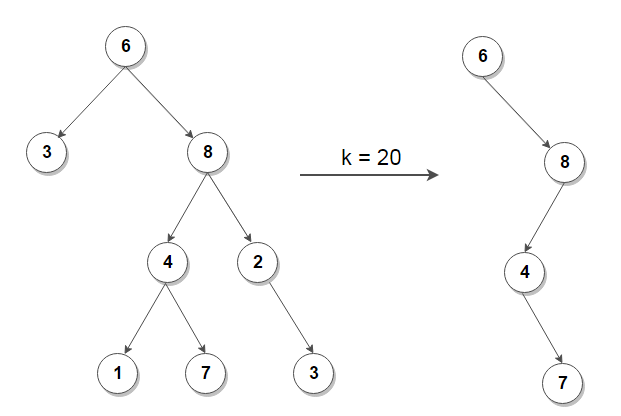
*/ / \*

*5 9 3*

*/ \*

*4 6*

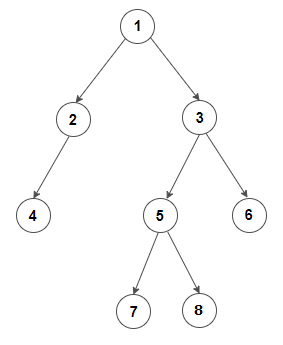
countBST: 6 (Trees rooted at 4,6,5,8,9,3)

1. (2 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.
2. (2 points) Create a tree from its inorder and preorder listing. The tree is not necessarily a BST. You may assume all elements are unique. Be sure to store parent references. You notice that having parent pointers may be helpful for some of the exercises, but it comes with the cost of having to always keep them current. For example, buildTreeTraversals(E[] inorder, E[] preorder) for the following tree:

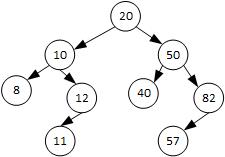
Inorder: 4 2 1 7 5 8 3 6

Preorder: 1 2 4 3 5 7 8 6

Output:

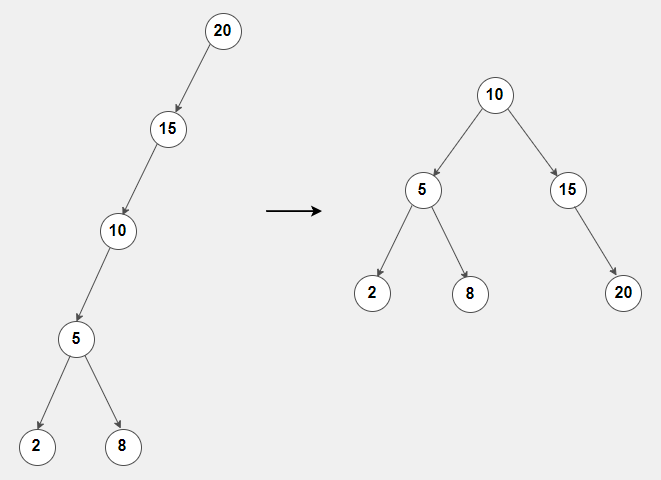


1. (2 points) Write a function lca( int a, int b) which returns the least common ancestor of two nodes in a binary search tree. A least common ancestor is an ancestor of both nodes and is closest to the nodes. A node is considered to be an ancestor of itself. In the tree below, the least common ancestor of 82 and 8 is 20. The least common ancestor of 42 and 50 is 50 (even though 42 doesn’t exist). The least common ancestor of 57 and 40 is 50.



1. (2 points) balanceTree() constructs a height balanced BST from an unbalanced BS. Notice we are not balancing via rotations, but just starting over.

For example:



1. (2 points) keepRange(int first, int last) removes nodes from a BST which have keys outside a valid range.

