# EC330 Applied Algorithms and Data Structures for Engineers
# Spring 2022

## Homework 5

**Out:** March 27, 2022
**Due:** April 6, 2022

*This homework has a written part and a programming part. Both are due at 11:59 pm on April 6. You should submit both parts on Gradescope.*

*Problem 1, 2 and 3 should be completed individually, whereas Problem 4 can be done in pairs. See course syllabus for policy on collaboration.*

**1. AVL Tree [20 pt]**

a. In a binary tree, a child is called an "only-child" if it has a parent node but no sibling, i.e. it is the only child node of its parent. Prove that for any AVL tree with $n$ nodes ($n > 0$), the total number of only-children is at most $n/2$.

b. Is the statement "the order in which elements are inserted into an AVL tree does not matter, since the same AVL tree will be established following rotations" true? If yes, explain why. If not, give a counterexample.

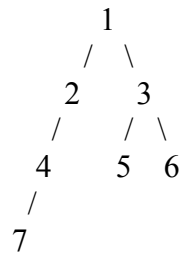
**2. Red-Black Tree [10 pt]**

An *in-order* traversal of a red-black tree containing the set of numbers 1 to 10 gives the following colors: R B R R R B R B R B. Produce the tree.

**3. Programming (Individual) [20 pt]**

Implement the method *isWeightBalanced(node* root, int k)* for checking if a binary tree is *weight balanced*. Similar to the notion of height balance in AVL trees, we define a binary tree to be *k*-weight balanced if for every node in the tree, the difference between the weight of the node's left subtree and the weight of the node's right subtree is *no more than k*. The weight of a (sub)tree is simply the number of nodes in that (sub)tree. Submit *balance.cpp* on Gradescope. **[20 pt]**

For example, in the binary tree below,
the weight of the subtree rooted at *node2* is *3*,
*isWeightBalanced*(*node1*, *0*) should return 0,
*isWeightBalanced*(*node2*, *1*) should return 0,
*isWeightBalanced*(*node3*, *0*) should return 1,
*isWeightBalanced*(*node4*, *1*) should return 1.




**4. Programming (Pair) [50 pt]**

Consider two binary search trees that contain the same set of *unique* keys, possibly in different orders. Design an efficient algorithm that will transform any given binary search tree into any other binary search tree (with the same keys) *using only ZIG and ZAG rotations*.

The provided *BST.h*, and *BST.cpp* files contain a BST class, implementing a binary search tree, and a Rotation class, which stores a rotation.

Implement a new derived class of BST, MyBST, which extends the binary search tree with the aforementioned *transform* method. You are allowed to modify the source BST in *transform*.
To get full credit your solution must work on any two binary search trees that contain exactly the same set of unique keys.

You may *not* modify *BST.h* or *BST.cpp*. You can add methods to but not remove any from *MyBST.cpp* (and modify *MyBST.h* accordingly) as you see fit. Submit both *MyBST.h* and *MyBST.cpp* on Gradescope.