Anti-AVL

AVL trees use left and right rotations to ensure that the binary search tree ends up balanced after insertion and deletion. Here is insertion for a variant of AVL trees that we call *Anti-AVL*.

```
Input: The root node v of an Anti-AVL tree B and an
       element x
Output: An Anti-AVL tree with x inserted
Procedure InsertAntiAVL(v, x)
    if v = null then
        v ← new Node(x)
    else if x < v.element then
        v.left ← InsertAntiAVL(v.left, x)
    else if x > v.element then
        v.right ← InsertAntiAVL(v.right, x)

    if v.left ≠ null then
        return RightRotate(v)

    return v
```

```
Input: A node z
Output: The root of the right-rotated
        tree
Procedure RightRotate(z)
    y ← z.left
    T2 ← y.right

    y.right ← z
    z.left ← T2

    return y
```

(a) Insert the numbers $3, 1, 4, 2, 5$ with `InsertAntiAVL`.

- Briefly describe what the resulting Anti-AVL tree looks like.
- State the total number of rotations that occur during the insertions.

(b) Describe more generally what Anti-AVL trees look like.

(c) A binary search tree is a binary tree where the element in each node is:

- greater than all elements in the left subtree, and
- smaller than all elements in the right subtree.

Do Anti-AVL trees have the same property? Explain.

(d) What is the runtime complexity of lookup in Anti-AVL trees? You may assume that lookup is implemented identically to ordinary binary search trees and AVL trees. Briefly justify your answer.

(e) Briefly describe how insertion in Anti-AVL trees differs from insertion in AVL trees.