# Exercise 3

### 1. Representation & Representation Invariant

Representation : `acBST<A>` = <A: array>

Representation Invariant :

`RI` = A is sorted(increasing) **&&** m = length(A)/2, `A[m]` is the root

**&&** In the tree representation, levels are filled left to right

**&&** `A[(m+1)/2]` is the left child of the root

**&&** `A[:m]` is either empty or can be represented as an `acBST`

**&&** `A[n-(n-m)/2]` is the right child of the root, n = length(A)

**&&** `A[m+1:]` is either empty or can be represented as an `acBST`

### 2. acBST is an AVL tree

> Recall: In an **AVL** tree, for every node, the heights of the two child subtrees differ by at most one.

- As we said in `RI`, both left and right subtrees of each node are acBST;
- In each acBST, we figure its root as length/2, which means we separate its two subtrees almost-evenly(binarily). More precisely, the number of elements in the left subtree may one more bigger than the one of right subtree;
- Thus, all branches will reach its leaf almost the same time, perhaps one level differ(the branches on the left may one node longer). Each level is filled as evenly as possible, so the difference of height is at most one.

$\Rightarrow$ All acBST are AVL trees.

### 3. Naive AVL Construction Algorithm

#### (a). Count the Number of Rotations

[code in the wet part : `NaiveAVL.java`]

Output: `Total number of rotations: 7`

#### (b). Justify

**False**. 'Almost-complete AVL' requires the last level filled from left to right, as we shown in RI. But the `build(A)` function in the Naive AVL structure may cause the tree to be right-skewed. Thus, not all AVL created by build(A) is almost-complete.

**4. Rotation-Free AVL Construction**

```
AVLNode build2(int[] A):
    n = length(A)
    if (n = 0) :
        return null
    else :
        m = n / 2
        AVLNode node = new AVLNode(A[m])
        node.left = build2(A[:m])
        node.right = build2(A[m+1:])
    return node
```

**5. Complexity Analysis for the Worst Case**

Under this rotation-free strategy, the worst case is the same as its best case(or the general cases). Because the algorithm balances the left and right subtrees recursively, the tree is at most possibly evenly at every step. In any case, each node is created exactly once, and the conditional check repeats $2 \cdot \log_2 n$ times. Thus, the total time complexity is $n \cdot O(1) + 2 \cdot \log_2 n \cdot O(1) = O(n)$.

**6. Implementation and Testing**

[code in the wet part : `AutoAVL`]

**7. Almost-Completeness AVL**

Yes, it always generate an almost-complete AVL. Under our definition, the left subtree has `m` elements (from A[0] to A[m-1]), and the right subtree has `n-m-1` elements (from A[m+1] to A[n-1]). So the tree we built is either even or left-skewed, with at most one element heavier. Thus, our algorithm always generate an almost-complete AVL.