8. AVL tree.

1. AVL tree

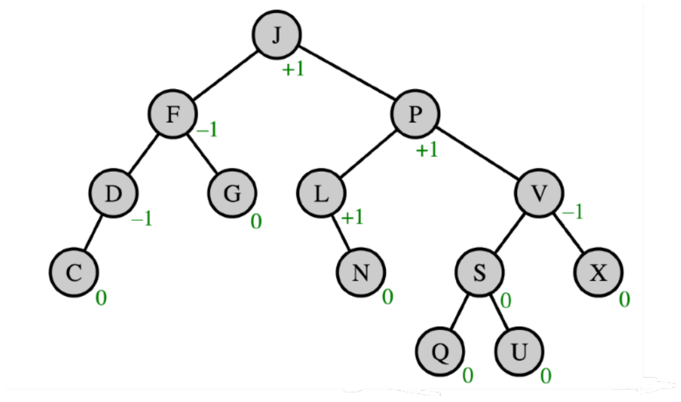
An AVL tree is a self-balancing binary search tree named after its inventors, Adelson-Velsky and Landis. It ensures that the height of the tree remains at the logarithmic level, providing optimal time complexity for search, insertion, and deletion operations.

Definition

In an AVL tree the balance factor of a node N is defined to be the height difference: BalanceFactor(N)=Height(RightSubtree(N))–Height(LeftSubtree(N)).

(Remark: The height of a tree with only one node is 0 by definition and the height of the empty tree is undefined, but the height of a tree with one node minus the height of the empty tree should be one, so in this case, we use -1 as the height of the empty tree, during the calculations of the balance factors.)

In an AVL tree, each node stores a **balance factor**, which is **the difference between the heights of its left and right subtrees**. If the absolute value of the balance factor is greater than 1, the node is unbalanced. To maintain balance, AVL trees use **rotation** operations to adjust the shape of the tree.



Specifically, **when a node's balance factor is greater than 1, we need to perform a rotation operation to re-balance its subtree**. If the balance factor is greater than 1 and the balance factor of the left subtree is greater than or equal to 0, we need to perform a right rotation. If the balance factor is greater than 1 and the balance factor of the left subtree is less than 0, we need to perform a left-right rotation. Similar rotation operations are required for balance factors less than -1 in the right subtree.

By using these rotation operations, AVL trees can automatically maintain balance to provide fast search, insertion, and deletion operations. However, due to the need to recalculate balance factors and perform rotation operations during insertion and deletion, AVL trees may be slightly slower than ordinary binary search trees.

2. AVL tree - rebalancing

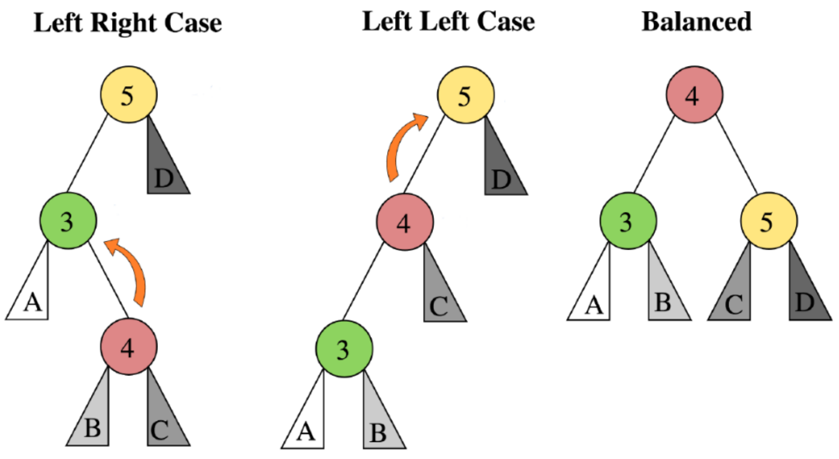
Rebalancing is not always necessary, only if during a modifying operation (e.g. insert, delete) a (temporary) height difference of more than one arises between two child subtrees. In this case, the parent subtree has to be rebalanced.

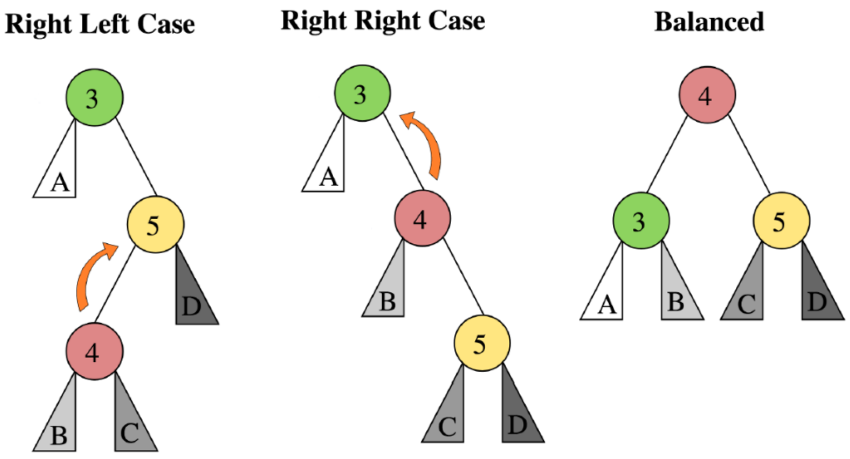
Specifically, when a node's balance factor is greater than 1, we need to perform a rotation operation to rebalance its subtrees. If the balance factor is greater than 1 and the balance factor of the left subtree is greater than or equal to 0, then we need to perform right rotation; if the balance factor is greater than 1 and the balance factor of the left subtree is less than 0, then we need to perform left and right rotation. For the case where the right subtree balance factor is less than -1, we need to perform a similar rotation operation.

AVL Tree Visualization: https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

There are 4 different cases, when rebalancing is necessary:

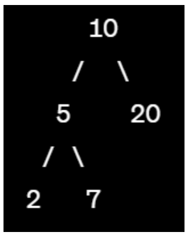
- Left-Left Case (LL)
- Left-Right Case (LR)
- Right-Left Case (RL)
- Right-Right Case (RR)

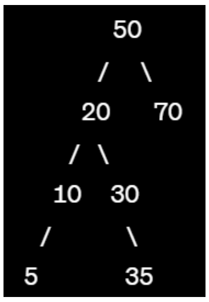


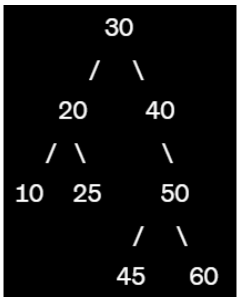


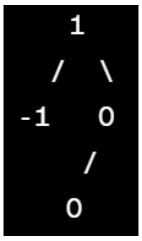
3. Exercises

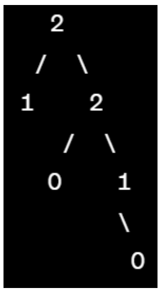
1. Determine if the following binary search tree is an AVL tree and calculate the balance factor for each node:

a. 

b. 

c. 

d. 

e. 

2. Build up an AVL tree: insert elements and use the LL, LR, RL and RR Cases when necessary.

a. Insert(5), Insert(7), Insert(11), Insert(3), Insert(1), Insert(6), Insert(20), Insert(10), Insert(2).

b. Insert(3), Insert(6), Insert(8), Insert(1), Insert(12), Insert(9), Insert(17), Insert(19), Insert(7), Insert(21).