Module 6 AVL Trees

💡 AVL Tree: a "self-balancing" BST - after the tree is mutated (via insertion or deletion), they can rebalance themselves as needed by "rotating" nodes to optimize their height.

AVL Trees and Balancing

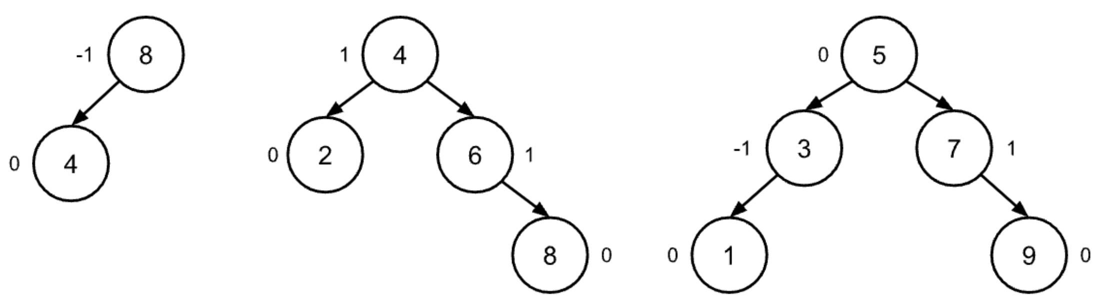
When we discuss balance in the context of BSTs, we're referring to trees with a height of approximately *log n. The reason why this is important is that the primary operations on BSTs all have O(h) runtime complexity.*

A BST is height-balanced if, at every node in the tree, the heights of the node's left and right subtrees differ by at most 1.

The balance factor of a node N is the difference in height between N's right subtree and its left subtree:

`balanceFactor(N) = height(N.right) - height(N.left)`

- If a node has a negative balance factor, we can call it left-heavy; otherwise, right-heavy



Various levels of heights in a balanced tree

For all root nodes: Tree 1: Right 0 - Left 1 = -1; Tree 2: Right 2 - Left 1 = 1; Tree 3: Right 2 - Left 2 = 0

AVL Tree Rotations

Each rotation has a center and a direction. The center is the node at which the rotation is performed (要移动的是center node), and we can perform either a left rotation or a right rotation around this center node.

A left rotation moves nodes in a “counterclockwise” direction, with the center moving downwards and the nodes to its right moving upwards.

- Also known as R-R rotation since the node’s right child is right-heavy

Conversely, a right rotation moves nodes in a “clockwise” direction, with the center moving downwards and the nodes to its left moving upwards.

💡 a rotation is needed when **height balance is lost at a specific node** in the tree

> If N has a balance factor of –2, this means N is left-heavy. If N has a balance factor of 2, this means N is right-heavy. Regardless of the direction of N’s heaviness, let’s refer to the heavier of N’s children as C.
>
> The node C itself will have a balance factor of –1, or 1, which, respectively, means that C itself is left-heavy or right-heavy.
>
> If N and C are heavy in the same direction (i.e., if the balance factor has the same sign at both N and C), then a single rotation is needed around N in the opposite direction as N’s heaviness

L-L	Single Rotation (Right)
L-R	Double Rotation (Left, then Right)
R-L	Double Rotation (Right, then Left)
R-R	Single Rotation (Left)

Rotation Mechanics

Single Rotations

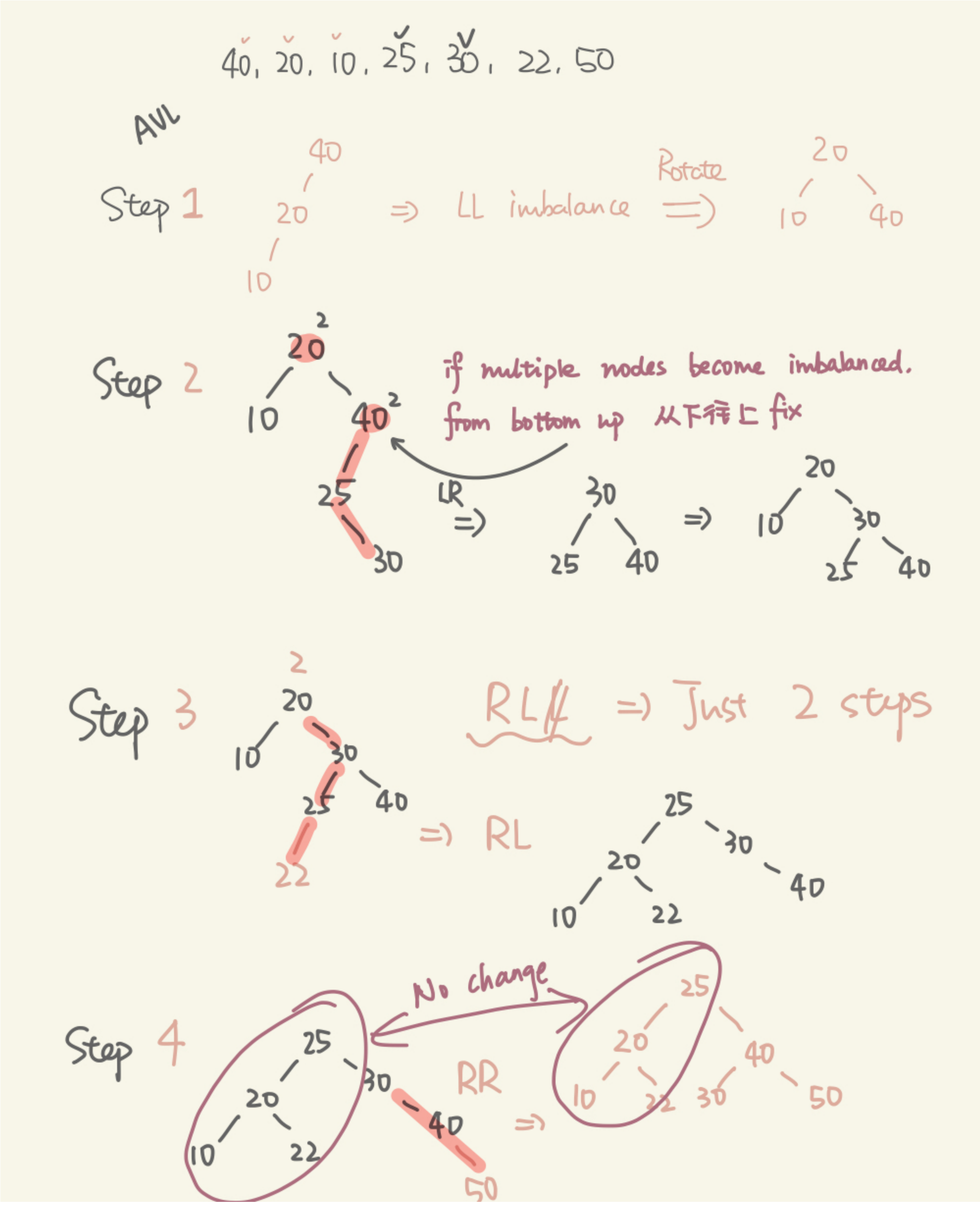
- same direction imbalance

Double Rotations

- **Left-right imbalance** – N is left-heavy and N’s left child C is right-heavy
    - To fix a left-right imbalance, we apply a left rotation around C followed by a right rotation around N.
- **Right-left imbalance** – N is right-heavy and N’s right child C is left-heavy
    - To fix a right-left imbalance, we apply a right rotation around C followed by a left rotation around N.

Rotation Implementation

> 💡 DO NOT WAIT UNTIL ALL THE NODES HAVE BEEN INSERTED/REMOVED! CHECK FOR IMBALANCE EVERYTIME YOU DO INSERT/REMOVE.



If we wrote a Python **Node** class to represent each node, we would need the following properties:

- **key** - an integer that the nodes are sorted by
- **value** - the value held by the node
- **height** - an integer representing the height of the node
- **left** - the left child *Node*
- **right** - the right child *Node*
- **parent** - the parent *Node*