AVL Trees

Sergios Anestis Kefalidis
Konstantinos Nikoletos
Kostas Plas &
Manolis Koubarakis

What is an AVL tree?

AVL (named after inventors **A**delson-**V**elsky and **L**andis) tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than **one** for all nodes.

Lookup, insertion, and deletion all take O(log *n*) time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.

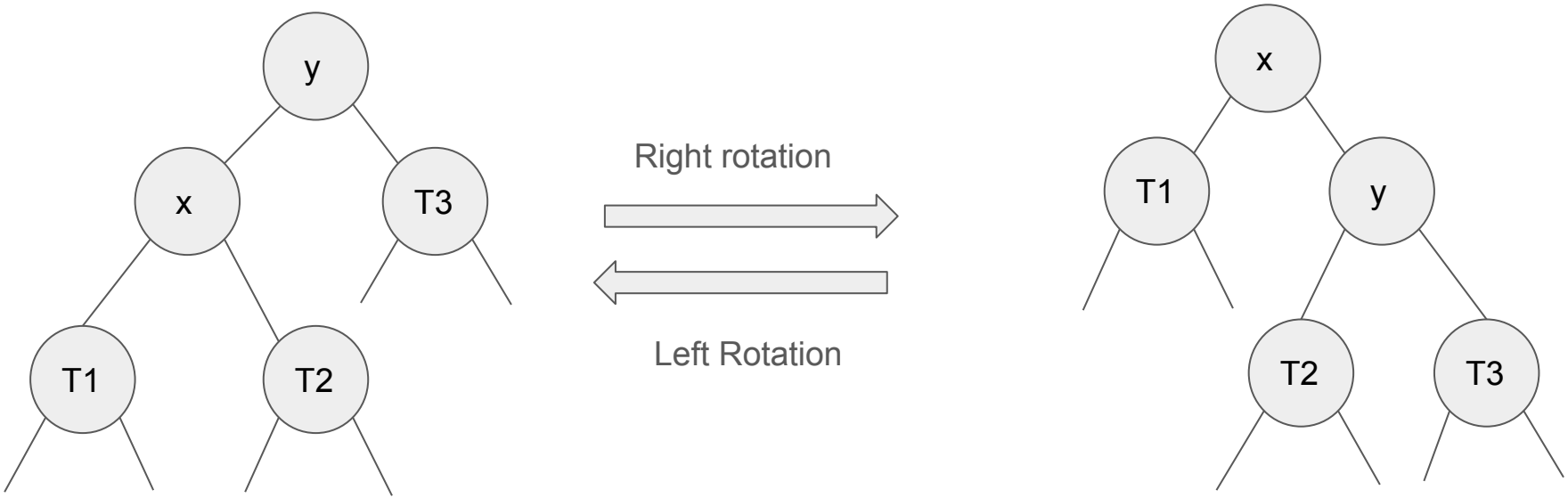
Tree Rotations (1)

Rotations are an operation performed on binary search trees to that rearrange the tree nodes, without violating the BST property. There are two types of rotations:

- Right Rotation
- Left Rotation

Tree Rotations (2)

T1, T2 and T3 are subtrees of the tree, rooted with y (on the left side) or x (on the right side)

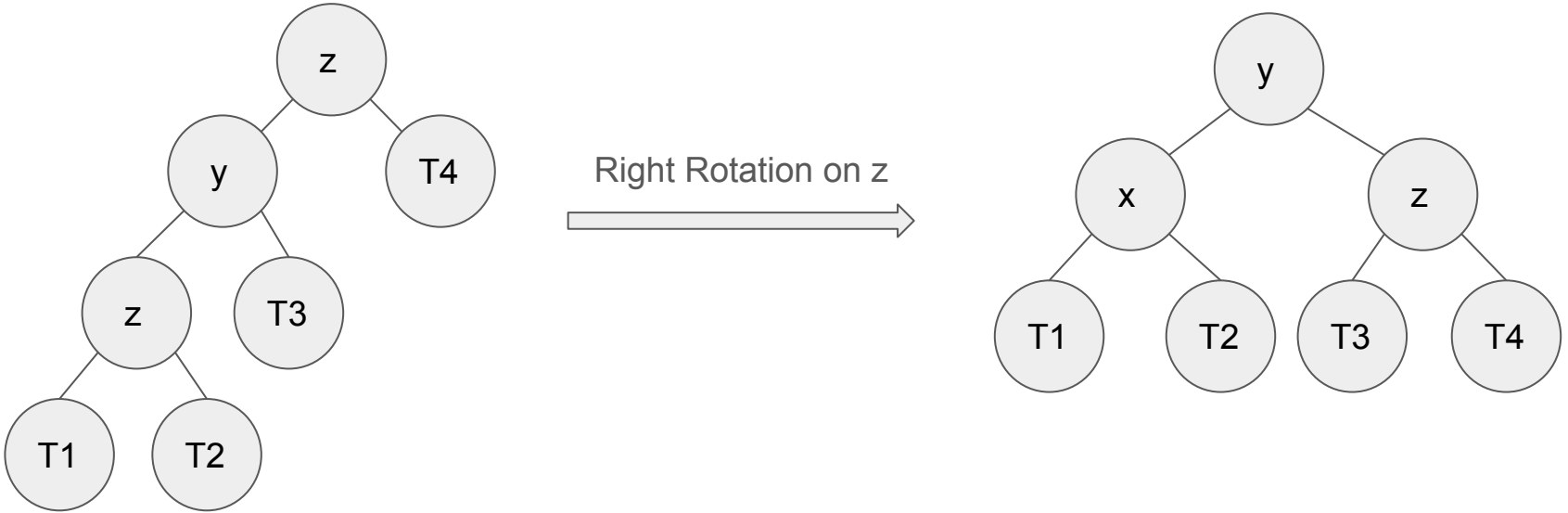


AVL Insertion: Algorithm

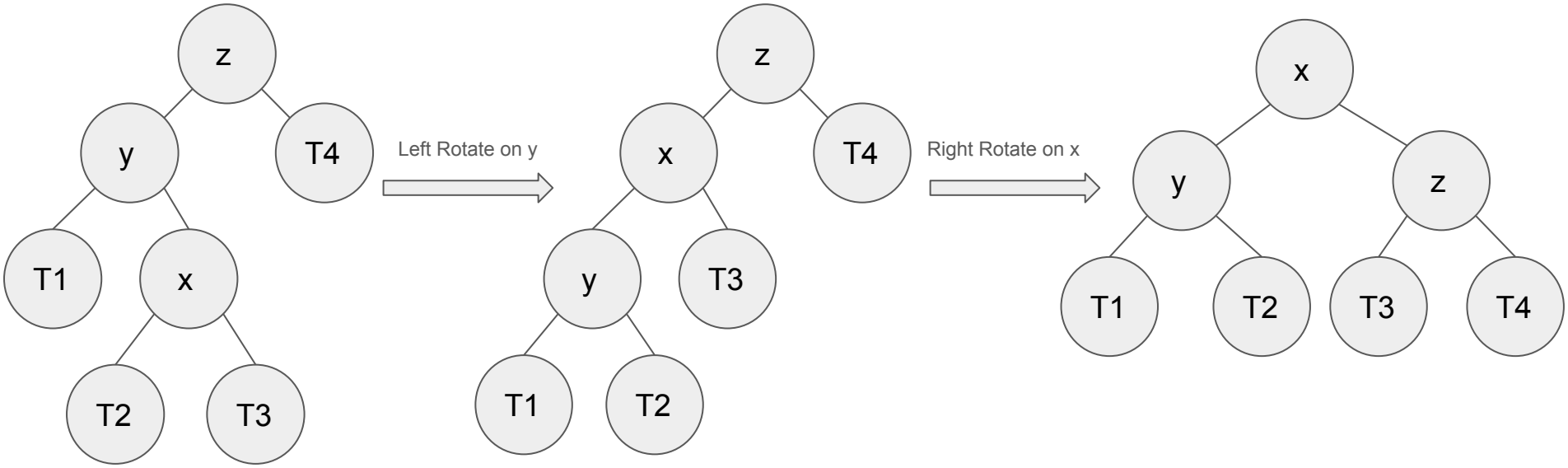
Let the newly inserted node be **w**

- Perform standard **BST** insert for **w**.
- Starting from **w**, travel up and find the first **unbalanced node**. Let **z** be the first unbalanced node, **y** be the **child** of **z** that comes on the path from **w** to **z** and **x** be the **grandchild** of **z** that comes on the path from **w** to **z**.
- Re-balance the tree by performing appropriate rotations on the subtree rooted with **z.** There can be 4 possible cases that need to be handled as **x, y** and **z** can be arranged in 4 ways.
- Following are the possible 4 arrangements:
    - y is the left child of z and x is the left child of y (Left Left Case)
    - y is the left child of z and x is the right child of y (Left Right Case)
    - y is the right child of z and x is the right child of y (Right Right Case)
    - y is the right child of z and x is the left child of y (Right Left Case)

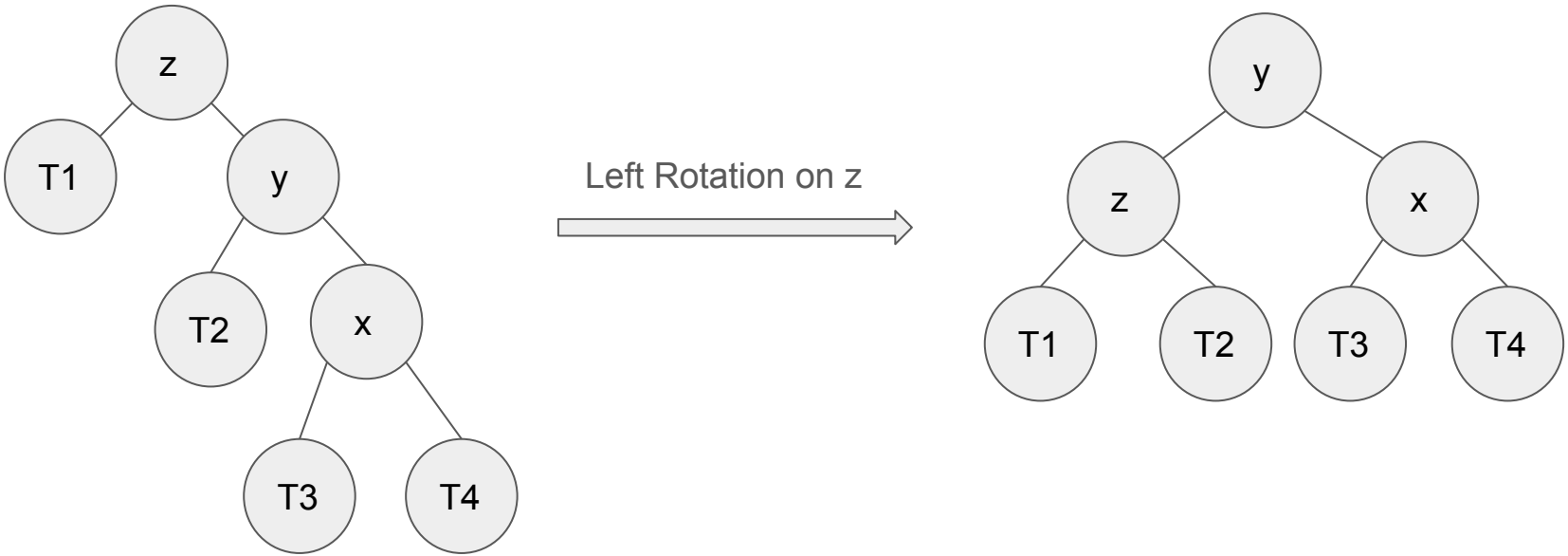
AVL Insertion: Left Left Case



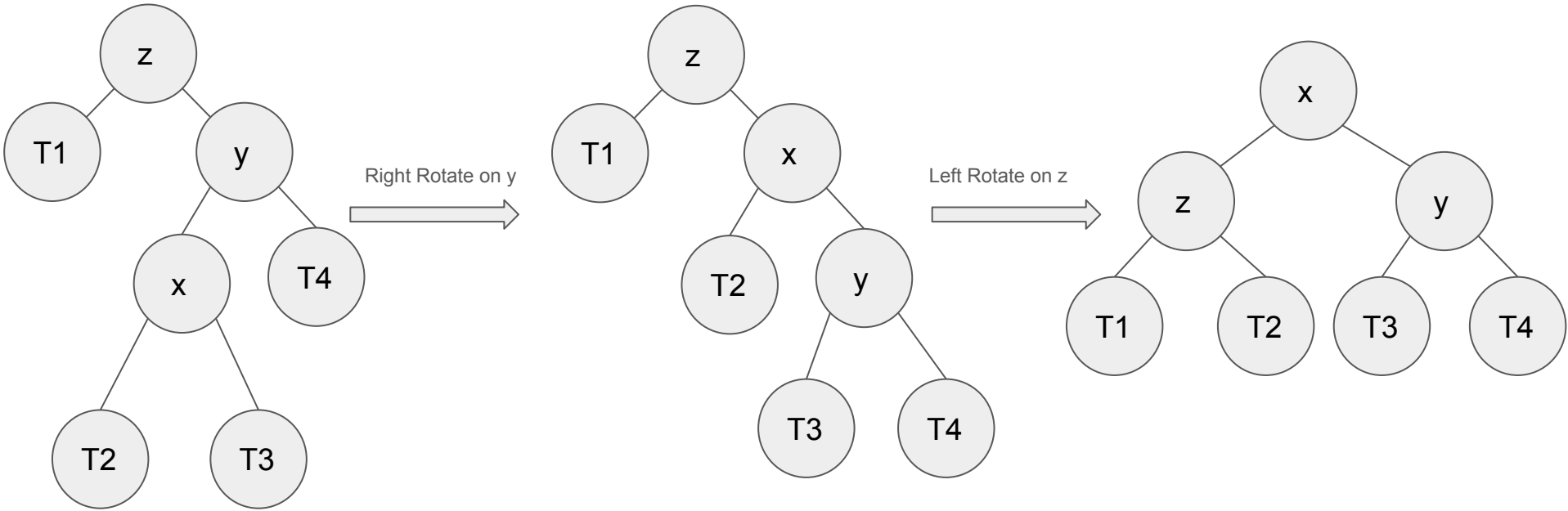
AVL Insertion: Left Right Case



AVL Insertion: Right Right Case



AVL Insertion: Right Left Case



AVL Insertion: Implementation

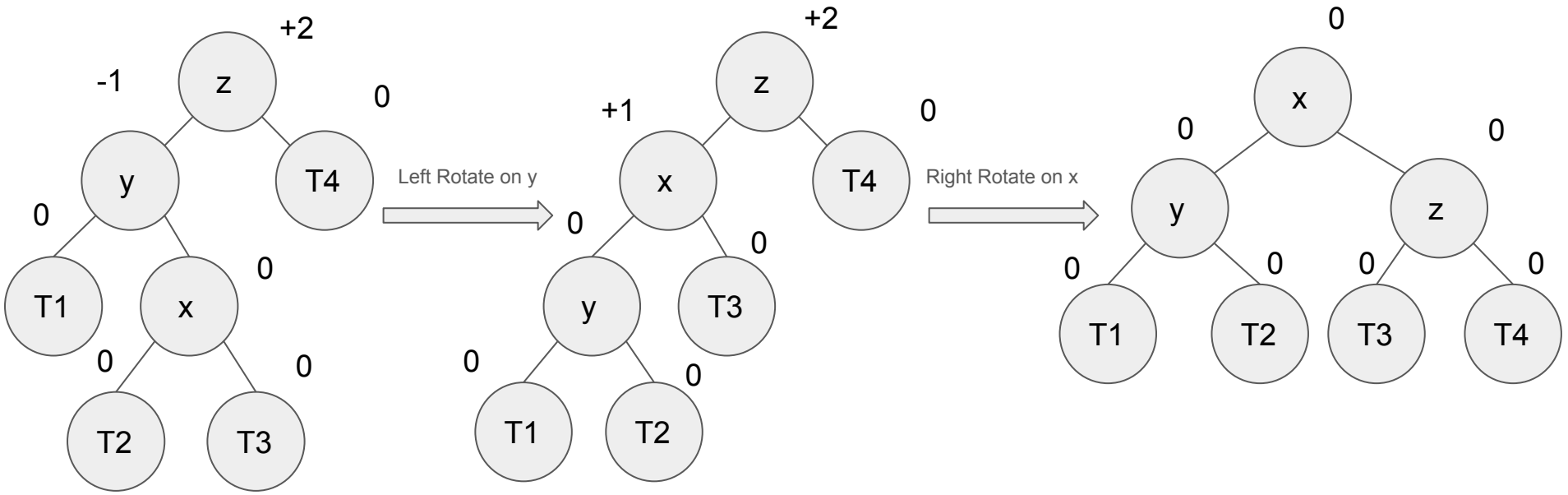
How can we implement this algorithm?

- Perform the normal BST insertion.
- The current node must be one of the ancestors of the newly inserted node. Update the **height** of the current node.
- Get the balance factor **(left subtree height – right subtree height)** of the current node.
- If the balance factor is greater than **1,** then the current node is unbalanced and we are either in the **Left Left case** or **left Right case**. To check whether it is **left left case** or not, compare the newly inserted key with the key in the **left subtree root**.
- If the balance factor is less than **-1**, then the current node is unbalanced and we are either in the Right Right case or Right-Left case. To check whether it is the Right Right case or not, compare the newly inserted key with the key in the right subtree root.

AVL Insertion : Balancing Factor

- When using the balancing factor we can easily decide if a subtree is balanced or not. The balancing factor should hold the values {-1,0,1}, else the subtree is unbalanced.
- The balancing factor can easily be calculated from each point of the tree (especially if we store the height of each subtree in our implementation)
- The balancing factor as mentioned above is
  - BF = Height(node->left) - Height(node->right)

AVL Insertion: Left Right Case Using Balancing Factor



Why should we use AVL Trees

AVL trees maintain the height of the tree at O(logn), which is very useful when performing searches on very large trees. Therefore, it is a better choice than using other self-balancing trees like Red-Black trees, when the need for search operations is high. Red-Black trees are better used when more insertion and deletion operations are required, since Red-Black trees generally perform less rotations.

Interesting Links

Original AVL paper

AVL Visualization