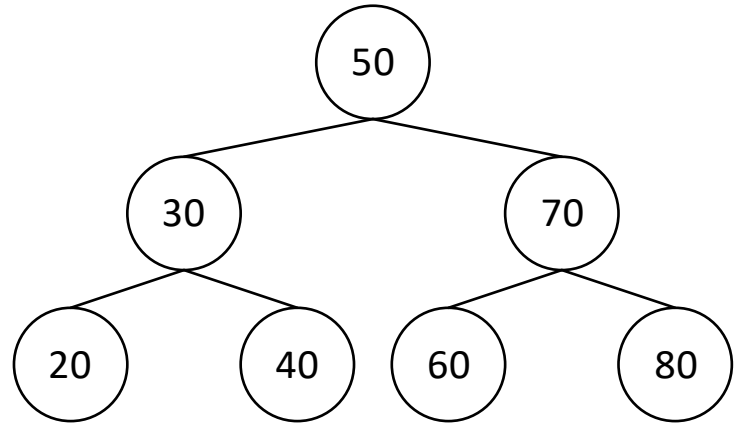
2021/2022(2)
IF184401 Design & Analysis of Algorithms
Binary Search & AVL Trees

Misbakhul Munir IRFAN SUBAKTI
司馬伊凡
Мисбакхул Мунир Ирфан Субакти

Binary Search Tree

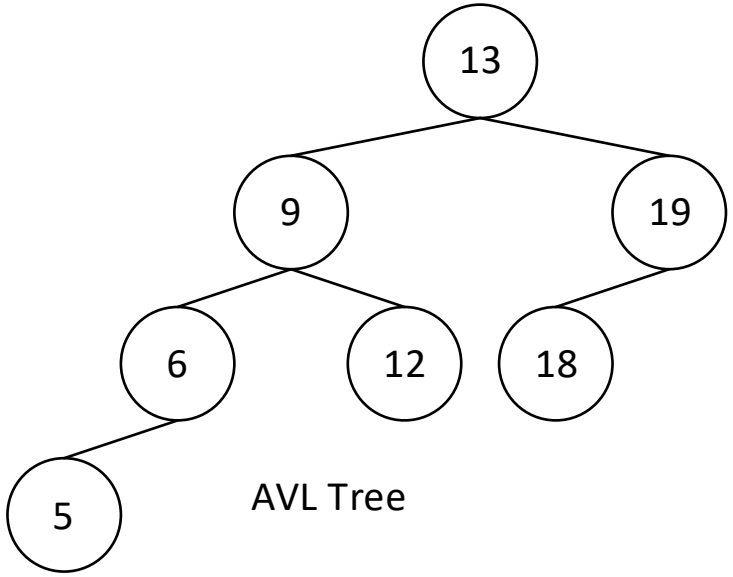
- Binary Search Tree (BST):
  - Node-based binary tree: data structure
    - Left branch(es) contains the nodes whose values < the value of the node
    - Right branch(es) contains the nodes whose values > the value of the node
    - Left and right branches should also be BST
    - There's no node has the same value



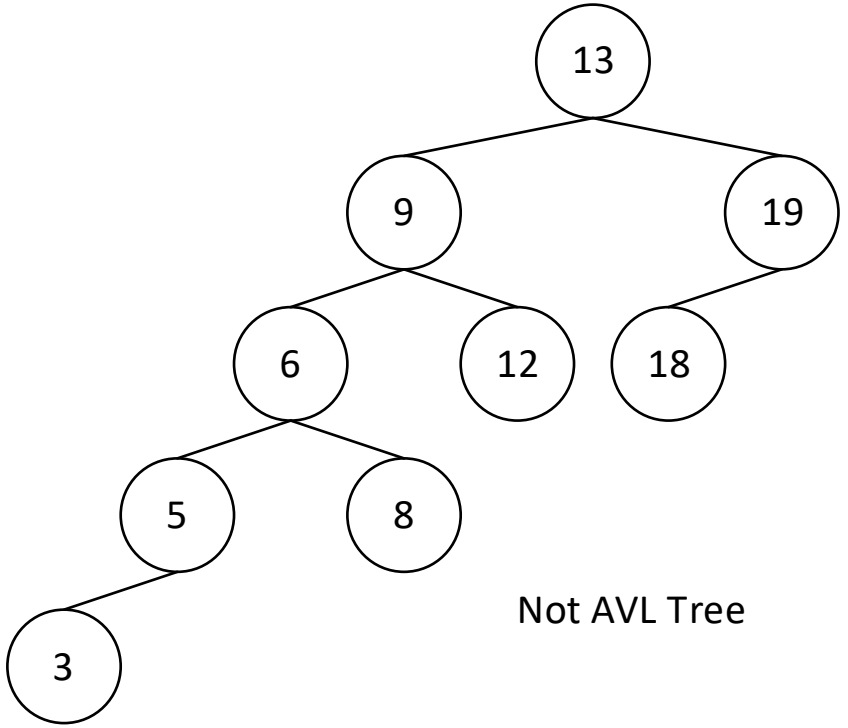
AVL Tree

- AVL = *A*delson-*V*elsky and *L*andis
- A type of BST which able to do self rebalance
- For each node, the height difference between the left and right branches is at most 1
- Insertion/deletion in AVL generally is allowed → causing the height imbalance property above → need the tree rotation for rebalancing the height
- Most operations in BST: find, max, min, insert, delete, etc., need the processing time $O(t)$ where t is BST's height → can be faster, $O(n)$ for the skewed binary tree, where n is the number of nodes
- If the tree's height can be kept to $O(\log n)$ after insertion & deletion → upper bound can be maintained at $O(\log n)$ for those operations, where n is the number of nodes
- The height of AVL tree is always $O(\log n)$ → upper bound for all of operations is $O(\log n)$

AVL Tree & Not AVL Tree



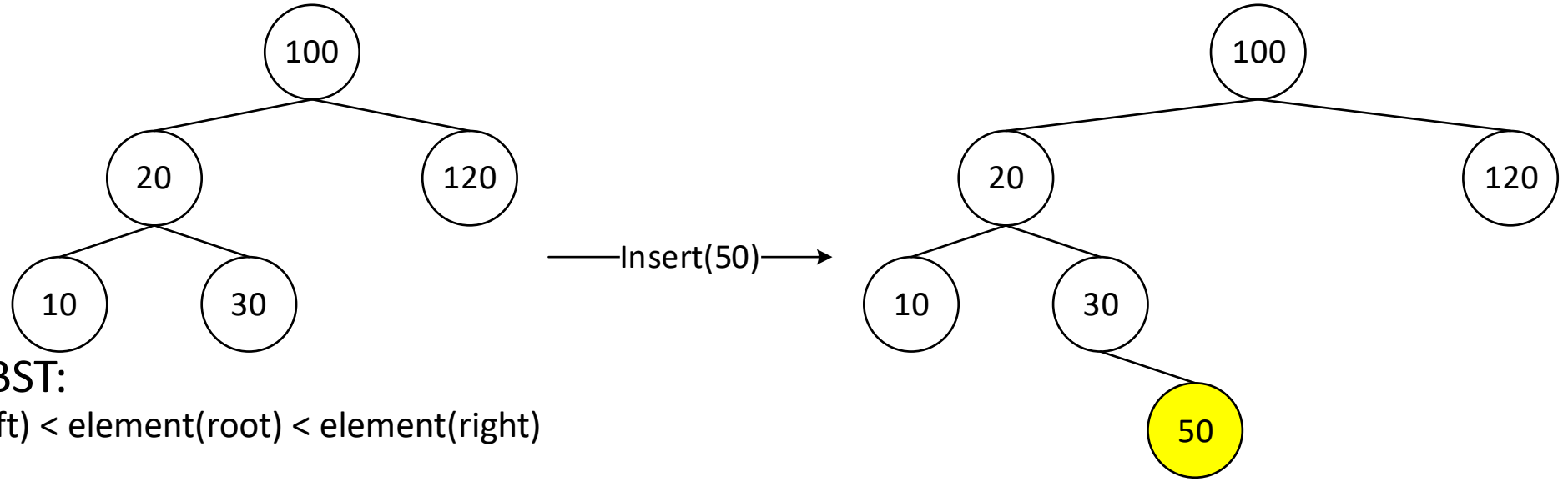
AVL Tree



Not AVL Tree

Insertion in BST

- Insertion of a new element (value) in a standard BST as below.
  - A new element always be inserted as a leaf node
  - Finding the place for the inserted node started from the root until a leaf node can be found
  - Once a leaf node has been found, then a new node will be added as a child of that leaf node



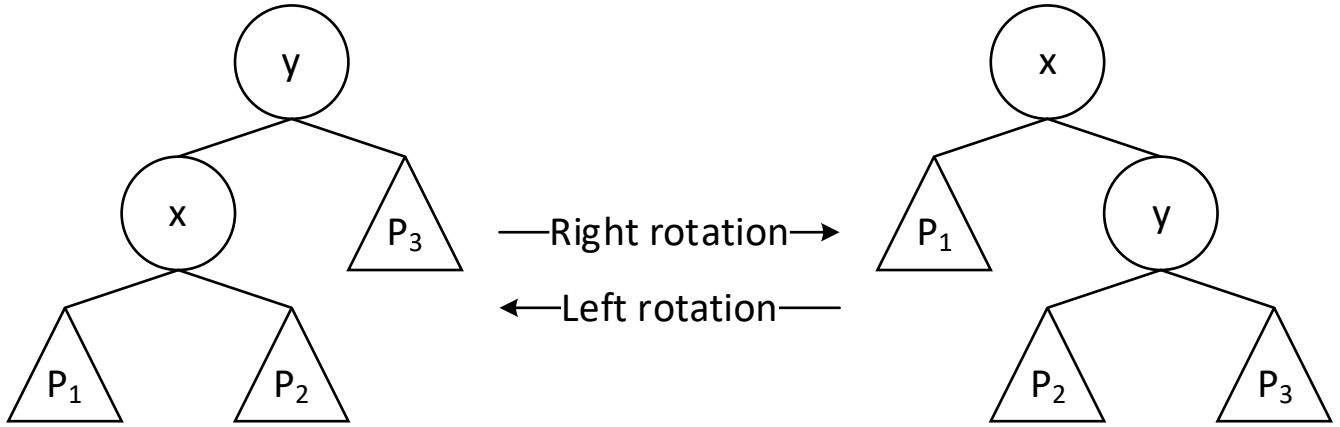
  - Property of BST:
    - element(left) < element(root) < element(right)

Insertion in AVL Tree

- To maintain the AVL tree's condition after insertion, the aforementioned standard BST insertion needs to be modified to regain its balance
- Two basic operations for BST rebalancing without breaching the BST's property: (element(left) < element(root) < element(right)
  - Left rotation
  - Right rotation

Left and Right Rotations

- P_1, P_2 and P_3 are the branches of a tree with the root *y* (left picture) and the root *x* (right picture)



- Elements of the trees above are following the rule:

  elements(P_1) < element(x) < elements(P_2) < element(y) < elements(P_3)

- So, it's still following the BST property compliance

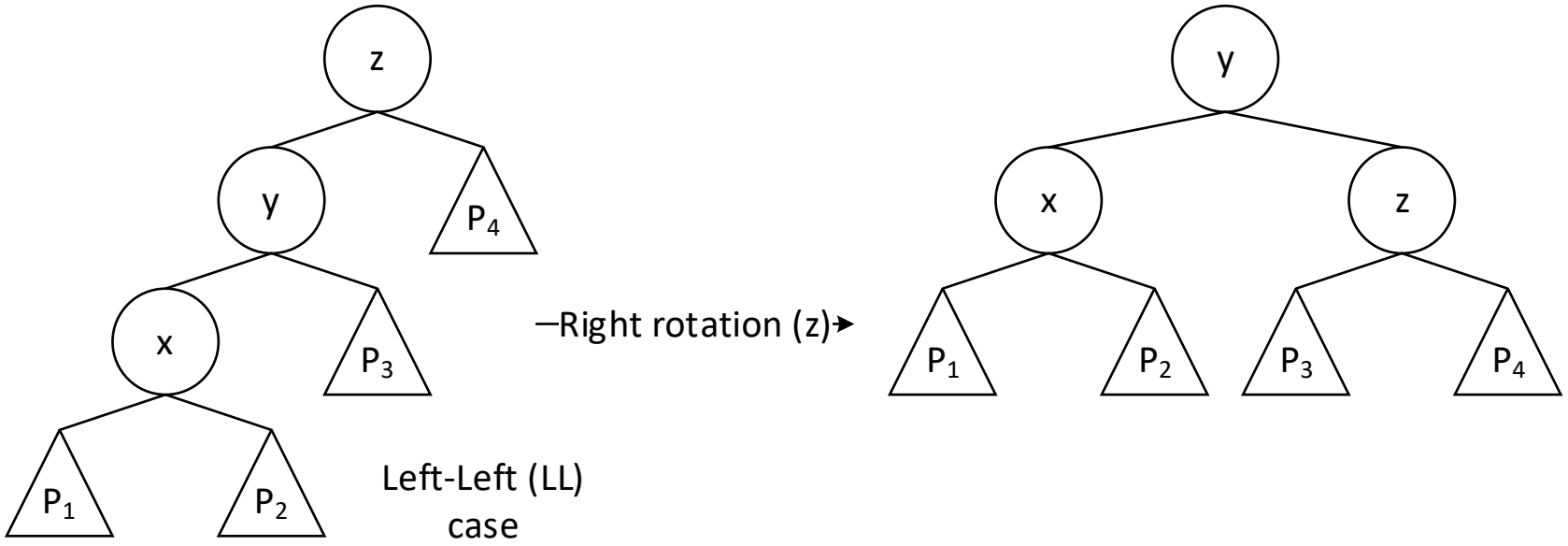
AVL: Insertion

The node will be inserted is *s*.

1. Do the standard BST insertion for *s*.
2. Starting from *s*, go up and find the first imbalanced node. Assumed that *z* is the first imbalanced node, *y* is the child of *z* which came from the path of *s* to *z*, and *x* is the grandchild of *z* which came from the path of *s* to *z*
3. Rebalance the tree by necessary rotation(s) on the branch(es) rooted in *z*. There are 4 possible cases, because *x*, *y* and *z* can be arranged in 4 positions:
   a) *y* is left branch of *z* and *x* is left branch of *y* (Left-Left case)
   b) *y* is left branch of *z* and *x* is right branch of *y* (Left-Right case)
   c) *y* is right branch of *z* and *x* is right branch of *y* (Right-Right case)
   d) *y* is right branch of *z* and *x* is left branch of *y* (Right-Left case)

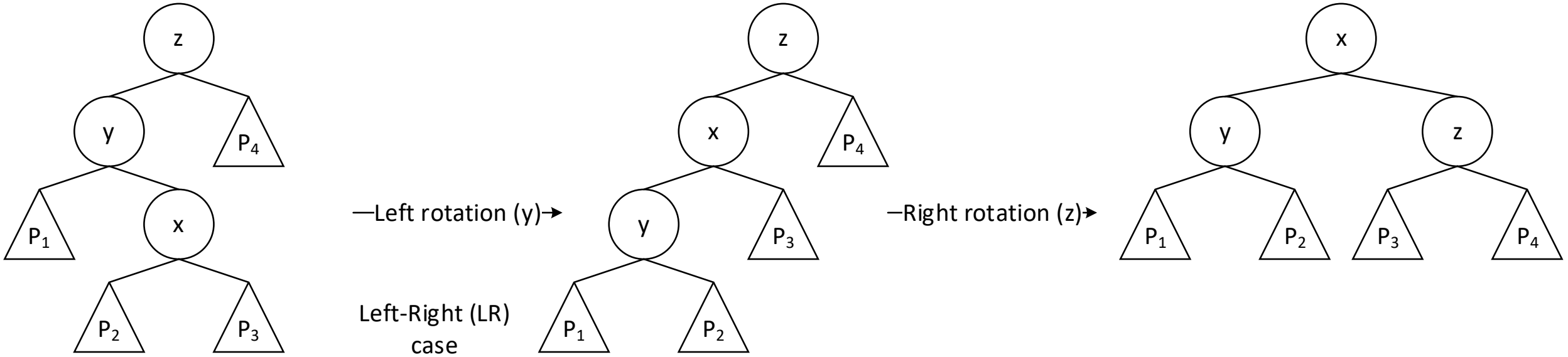
AVL Insertion: Left-Left Case

P_1, P_2, P_3 and P_4 are the tree branches



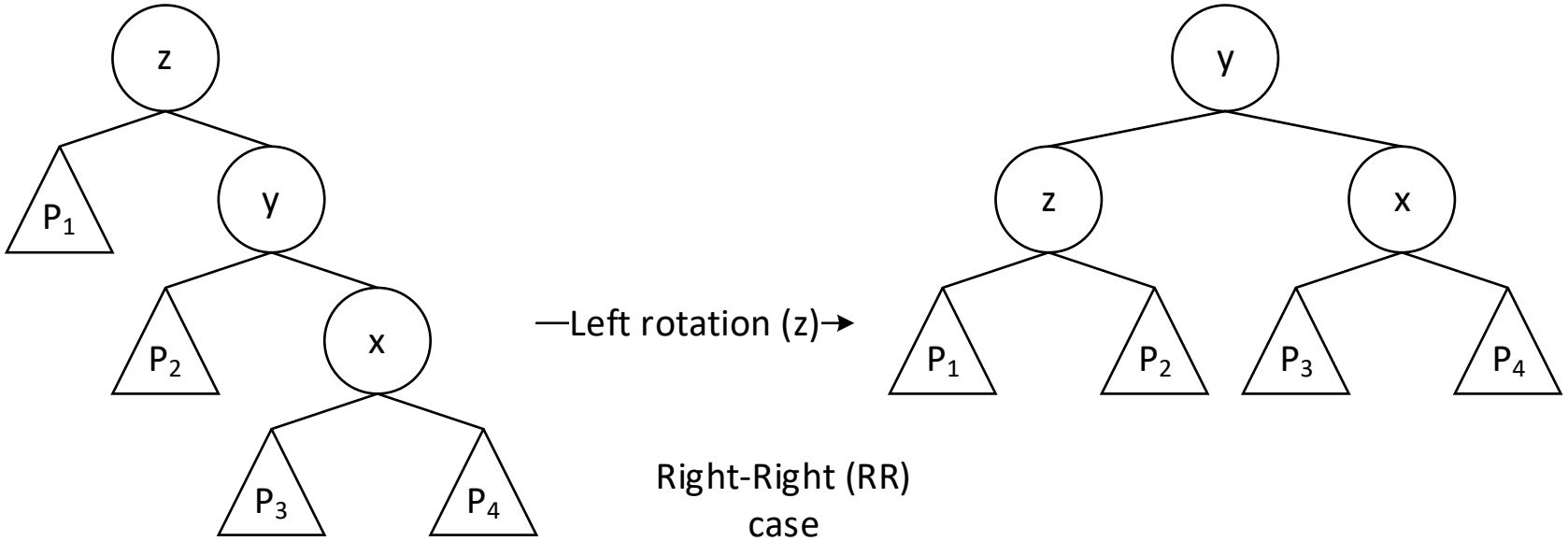
AVL Insertion: Left-Right Case

P_1, P_2, P_3 and P_4 are the tree branches



AVL Insertion: Right-Right Case

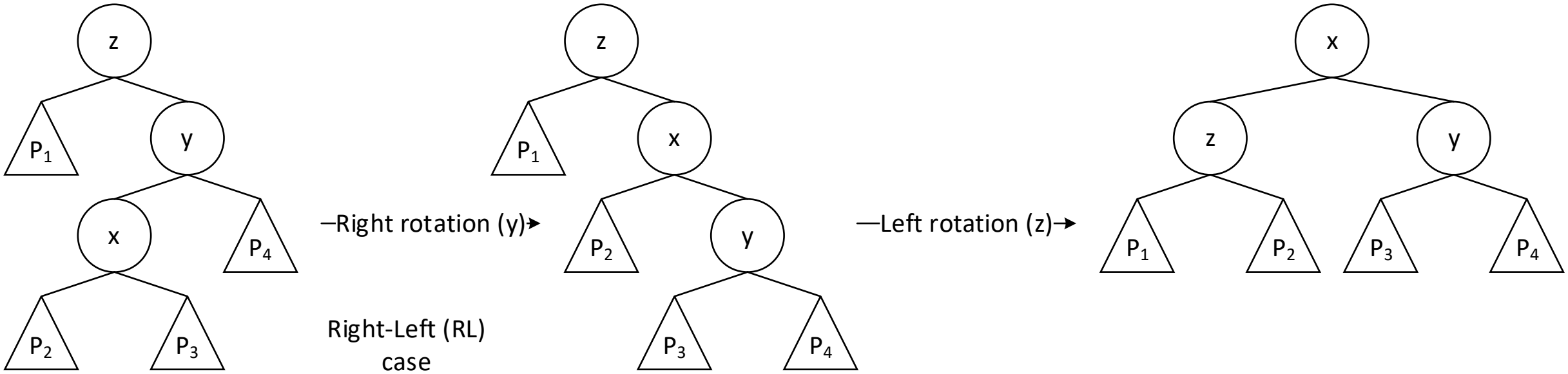
P_1, P_2, P_3 and P_4 are the tree branches



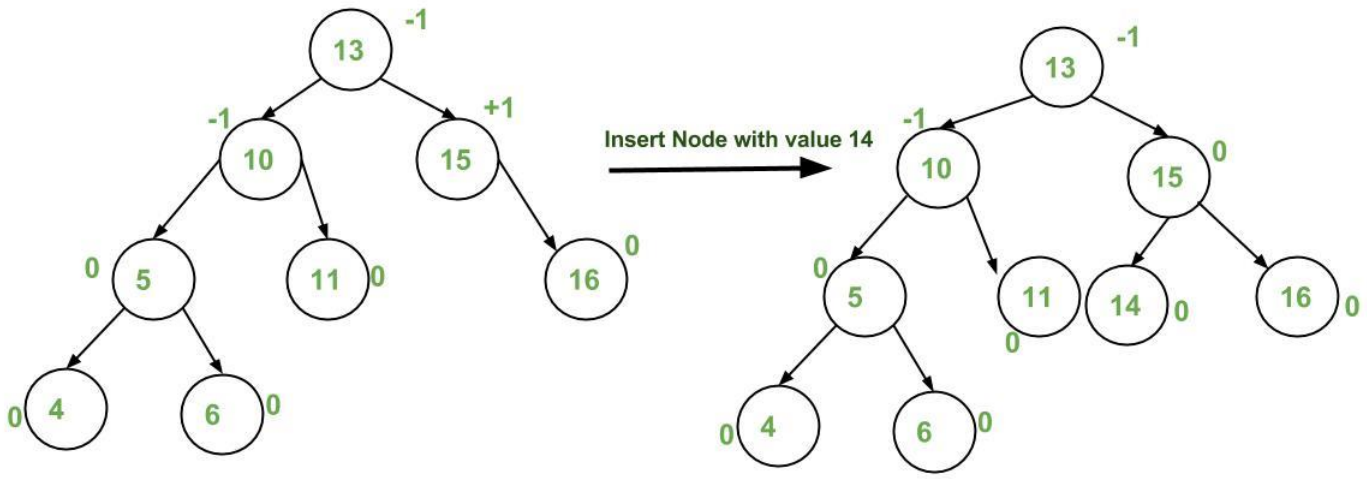
Right-Right (RR) case

AVL Insertion: Right-Left Case

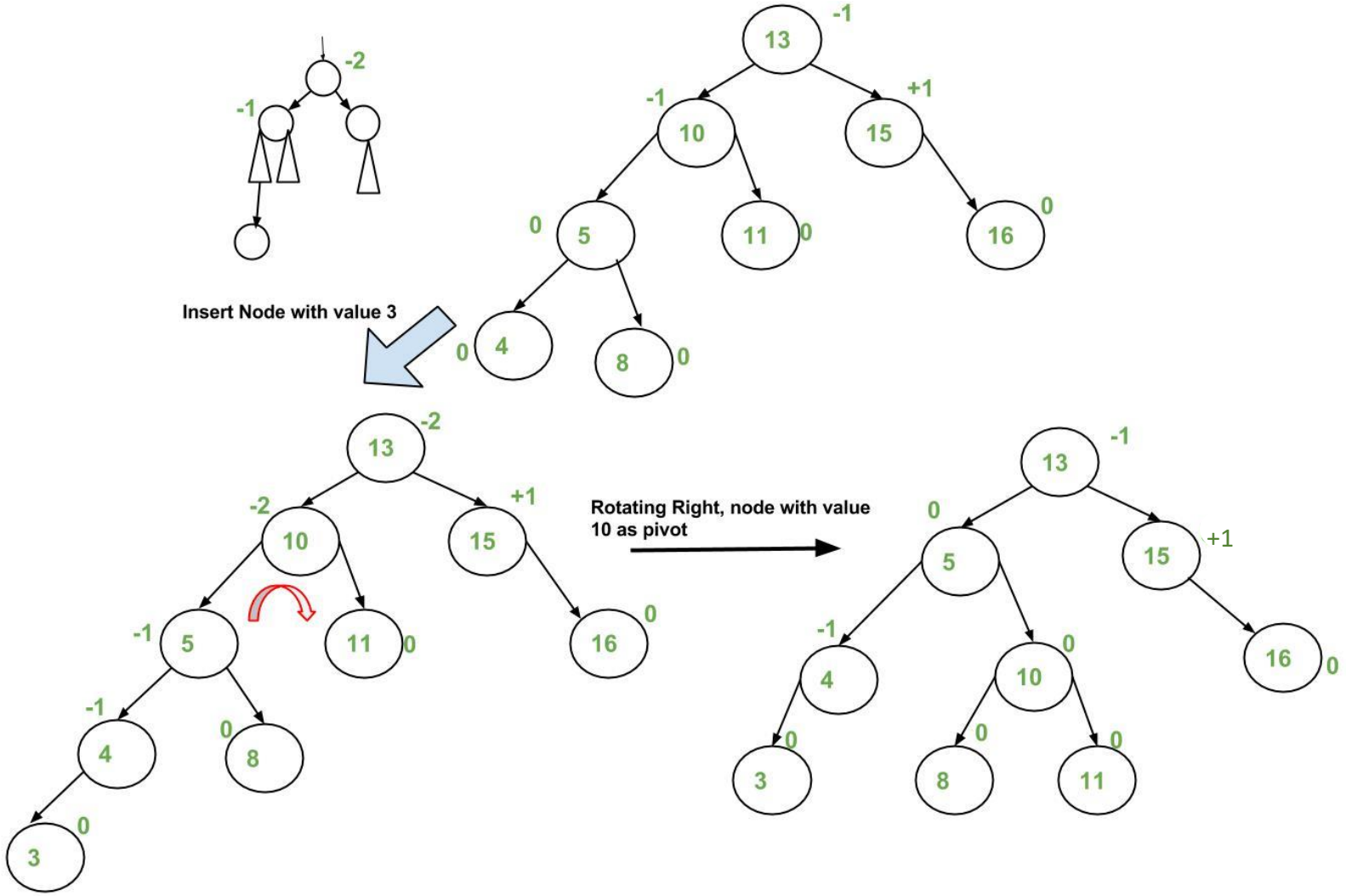
P_1, P_2, P_3 and P_4 are the tree branches



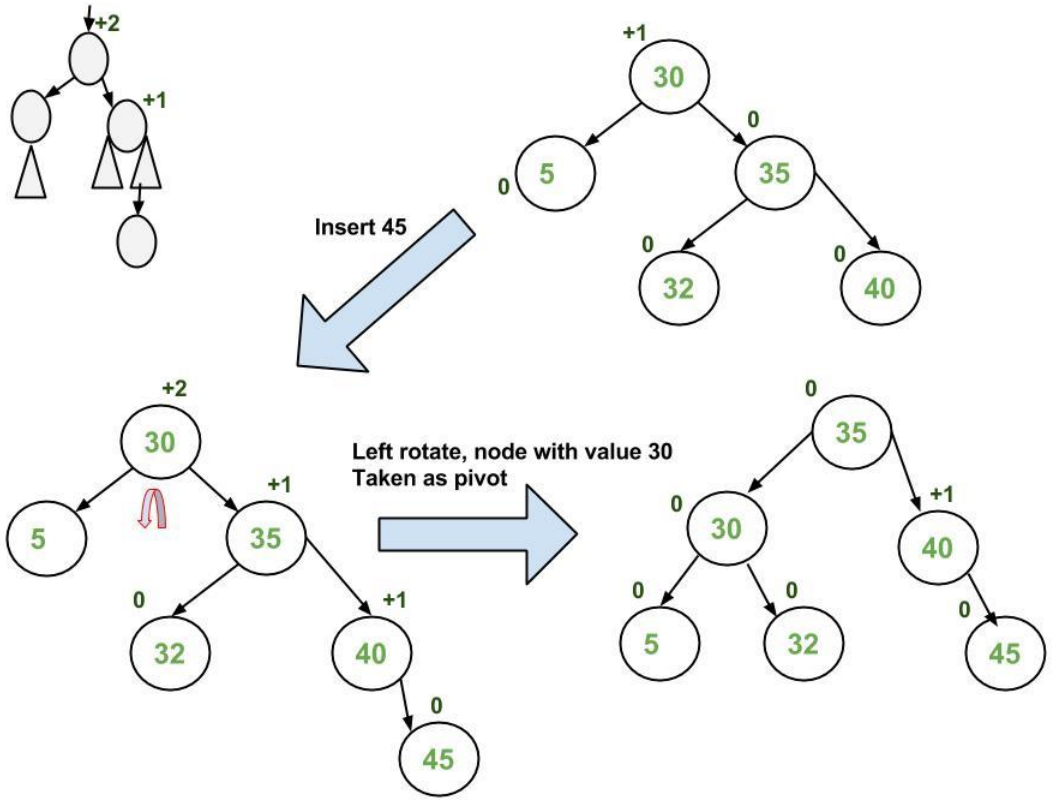
AVL Insertion: Example



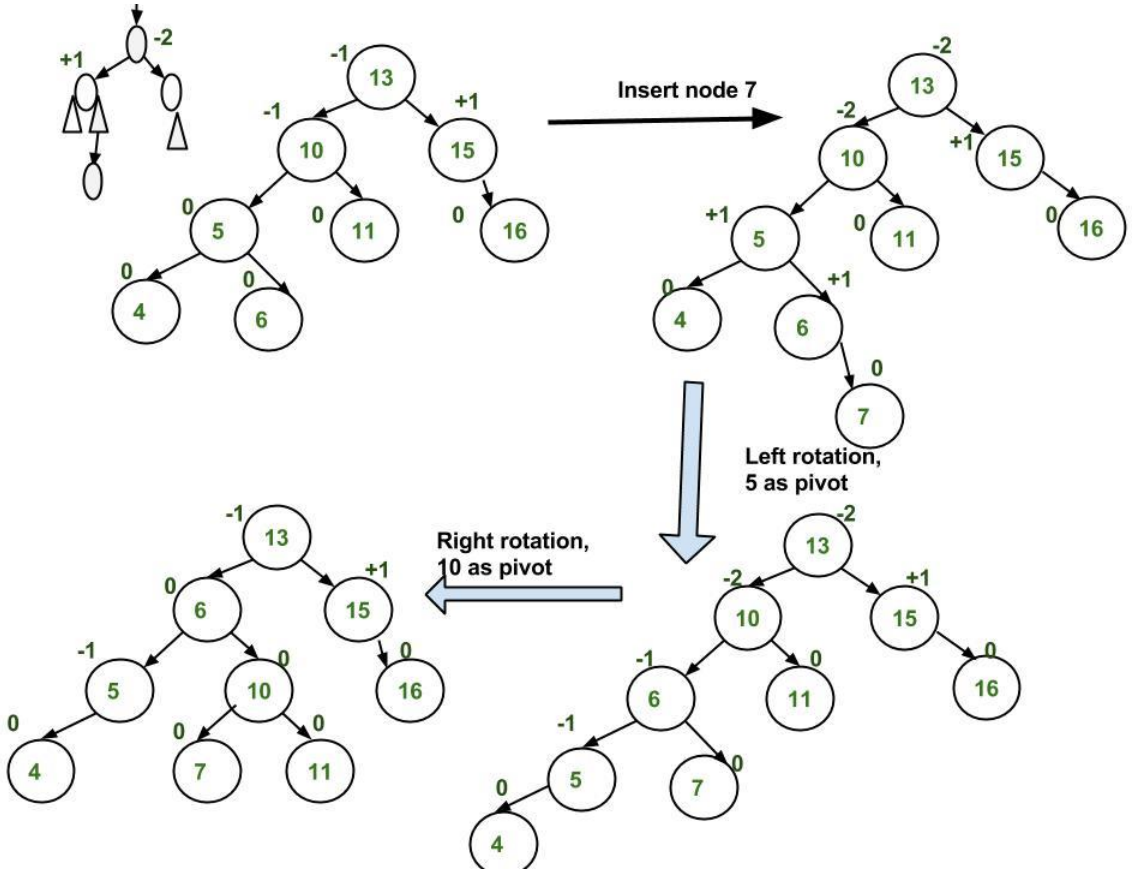
AVL Insertion: Example (continued)



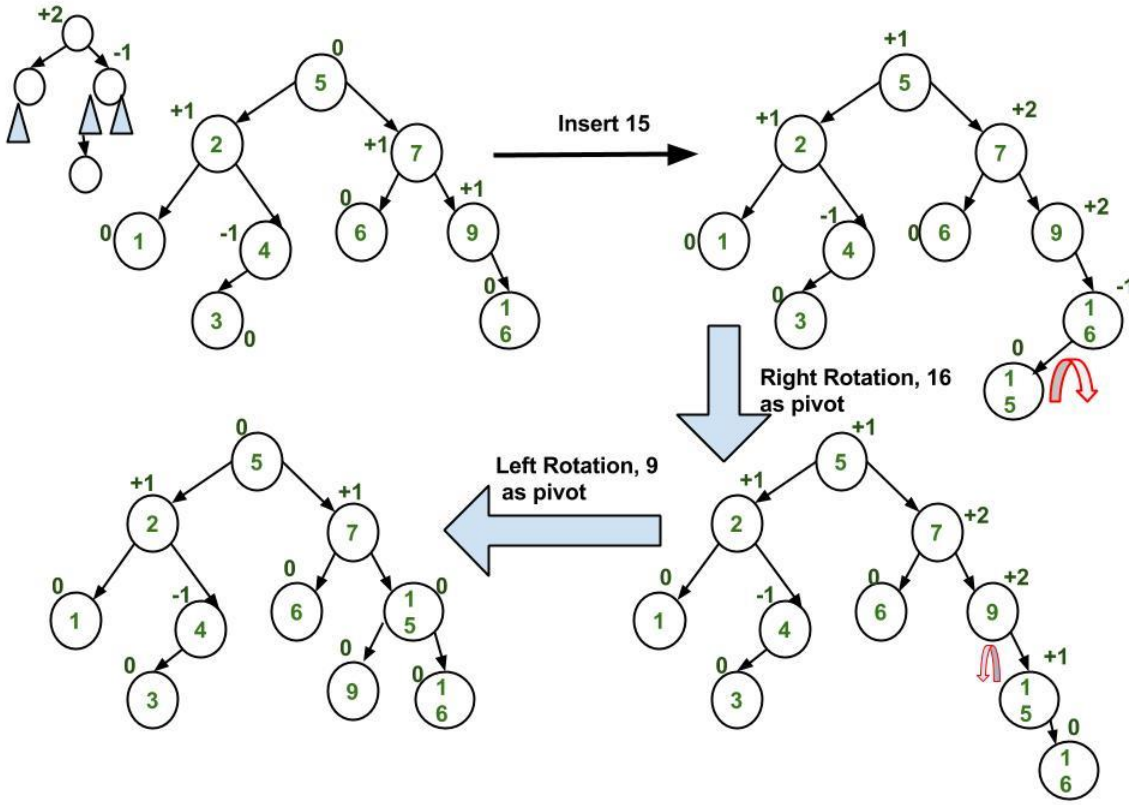
AVL Insertion: Example (continued)



AVL Insertion: Example (continued)



AVL Insertion: Example (continued)



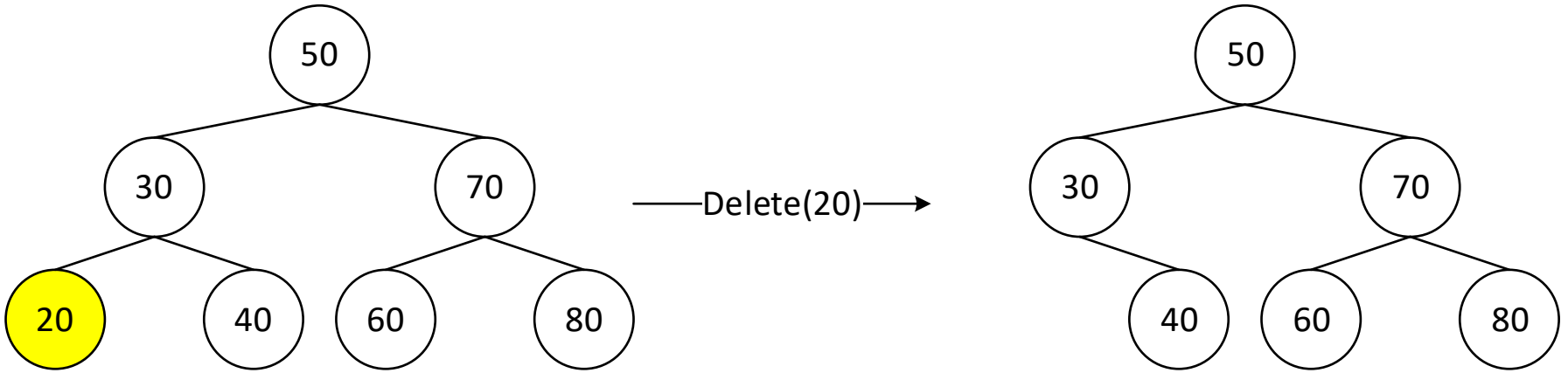
AVL Insertion: Implementation

1. Do the BST standard insertion
2. The current node should be any ancestors of the new inserted node. Update the height of the current node.
3. Get the balance factor (the height of right branch – the height of left branch) of the current node
4. If the balance factor less than -1 (< -1), then the current node is imbalance and we have Left-Left or Left-Right cases. To check whether it's Left-Left case or Left-Right, compare the sign of balance factors of path *z* → *y* → *x* → *s*. LL: *z* to *y* → (-) to (-), LR: *z* to *y* → (-) to (+)
5. If the balance factor greater than 1 (> 1), then the current node is imbalance and we have Right-Right or Right-Left cases. To check whether it's Right-Right case or Right-Left, compare the sign of balance factors of path *z* → *y* → *x* → *s*. RR: *z* to *y* → (+) to (+), RL: *z* to *y* → (+) to (-)

Deletion in BST

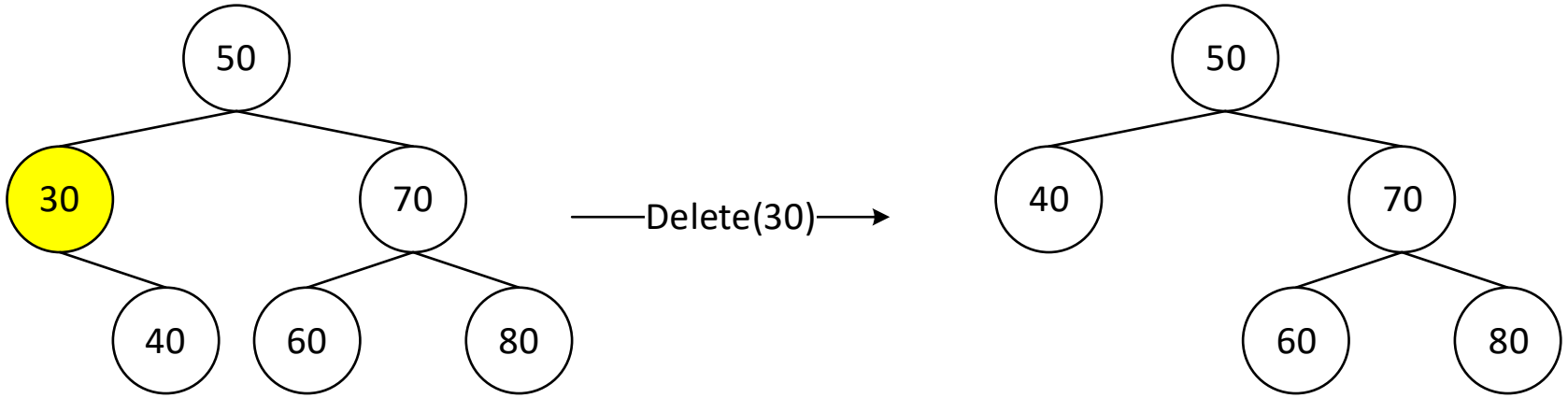
Three possible cases in deletion of an element (value) in standard BST are below.

1. The node will be deleted is a leaf node → just delete that node from the tree



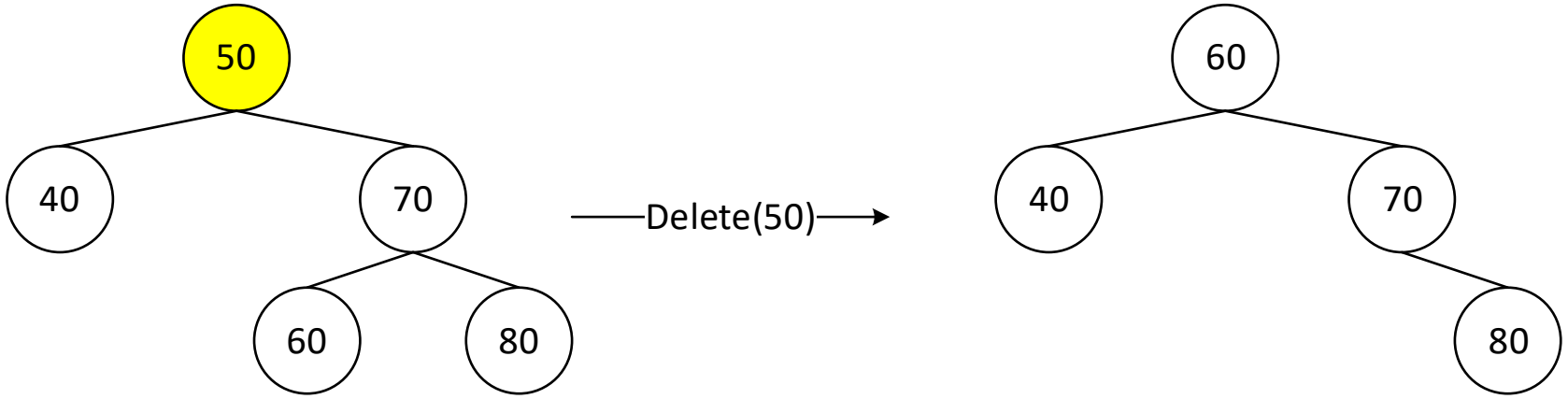
Deletion in BST (continued)

2. The node will be deleted only has 1 child → copy the child to the node will be deleted, then delete the child



Deletion in BST (continued)

3. The node will be deleted has 2 children → find the next (*successor*) node from the node will be deleted by using *inorder* traversing (*inorder successor*) → min(right branch). Copy *inorder successor* to the node will be deleted, then delete *inorder successor*



Deletion in BST (continued)

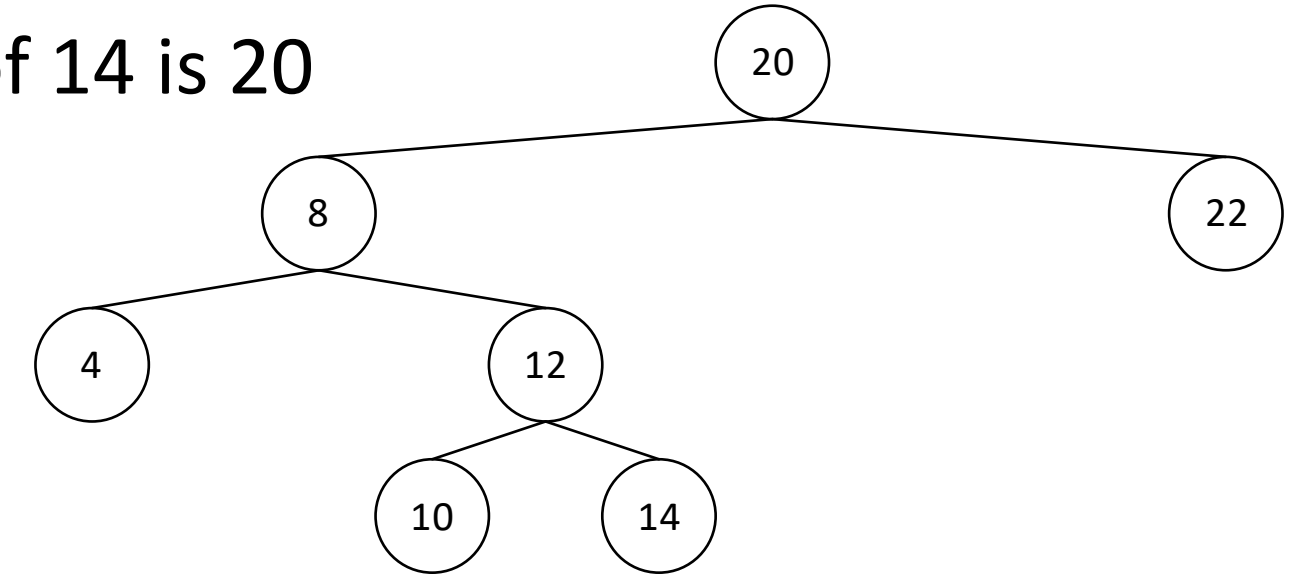
- Alternatively, the previous (*predecessor*) node of the node will be deleted by using *inorder* traversing (*inorder predecessor*) can also be used as a copy before node deletion → max(left branch)
- Note: *inorder successor* can be used if there exists children in the right branch. In this case, *inorder successor* can be obtained by finding the minimum value of children in the right branch (min(right branch)) from the node will be deleted.

Inorder Successor in BST

- In BST, *inorder successor* of a node is the next (*successor*) node which can be obtained by inorder traversing
- *Inorder successor* has *null* value for the last node in *inorder* traversing
- In BST, *inorder successor* from a given *input node* defined as a node with the least value, which is greater than the value of *input node* → sometimes it's important to find *inorder successor* in ordered tree

Inorder Successor in BST (continued)

- *Inorder successor* of 8 is 10
- *Inorder successor* of 10 is 12
- *Inorder successor* of 14 is 20

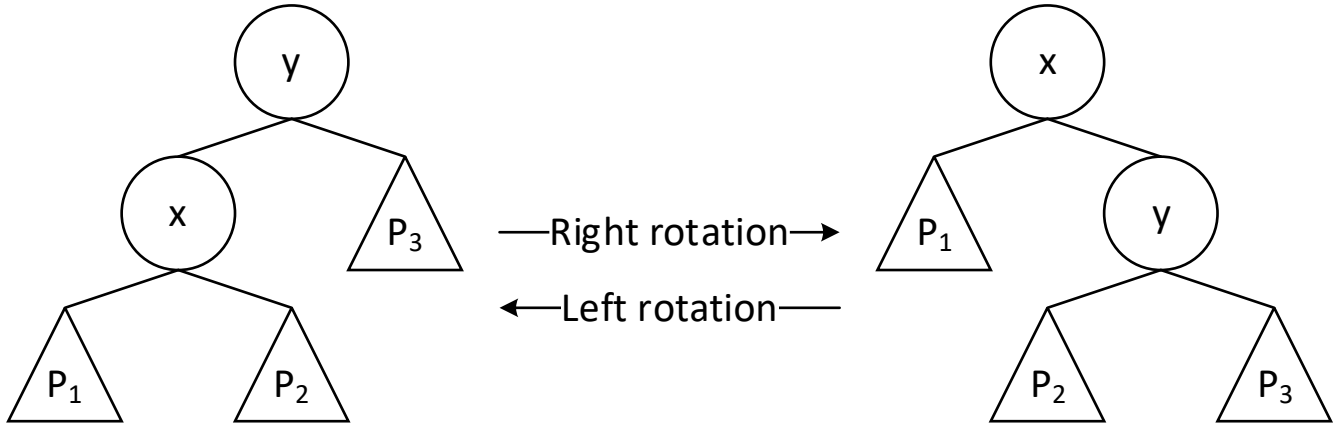


Deletion in AVL Tree

- To maintain the AVL tree's condition after deletion, the aforementioned standard BST deletion needs to be modified to regain its balance
- Two basic operations for BST rebalancing without breaching the BST's property: (element(left) < element(root) < element(right)
  - Left rotation
  - Right rotation

Left and Right Rotations

- P_1, P_2 and P_3 are the branches of a tree with the root y (left picture) and the root x (right picture)



- Elements of the trees above are following the rule:

  elements(P_1) < element(x) < elements(P_2) < element(y) < elements(P_3)

- So, it's still following the BST property compliance

AVL: Deletion

The node will be deleted is *s*.

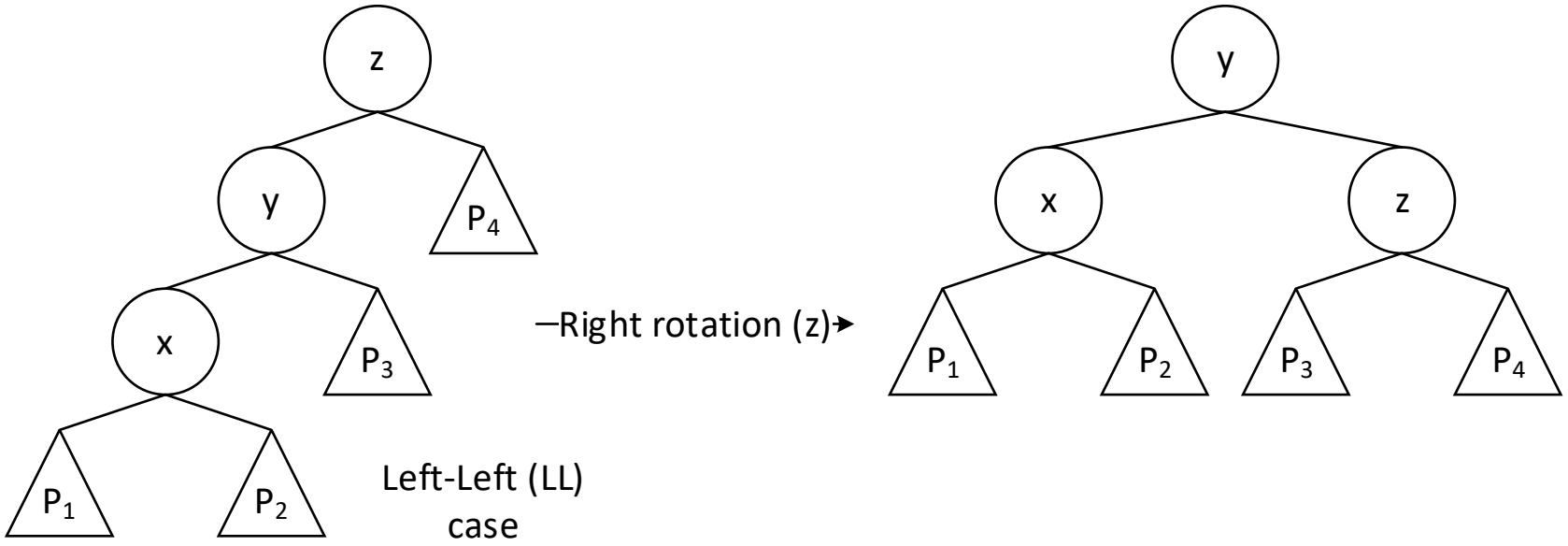
1. Do the standard BST deletion for *s*.
2. Starting from *s*, go up and find the first imbalanced node. Assumed that *z* is the first imbalanced node, *y* is the child of *z* whose highest balance factor, and *x* is the child of *y* whose highest balance factor. (Note: the definitions of *x* and *y* are different from *x* and *y* in the previous AVL's insertion)
3. Rebalance the tree by necessary rotation(s) on the branch(es) rooted in z. There are 4 possible cases, because x, y and z can be arranged in 4 positions:
   a) *y* is left branch of *z* and *x* is left branch of *y* (Left-Left case)
   b) *y* is left branch of *z* and *x* is right branch of *y* (Left-Right case)
   c) *y* is right branch of *z* and *x* is right branch of *y* (Right-Right case)
   d) *y* is right branch of *z* and *x* is left branch of *y* (Right-Left case)

AVL: Deletion (continued)

- As in AVL *insertion*, each cases will be described in the following.
- Note: not as in *insertion*, rebalancing on node *z* does not rebalance a whole AVL tree. After rebalancing *z*, most probably the *ancestors* of *z* should also be rebalanced.

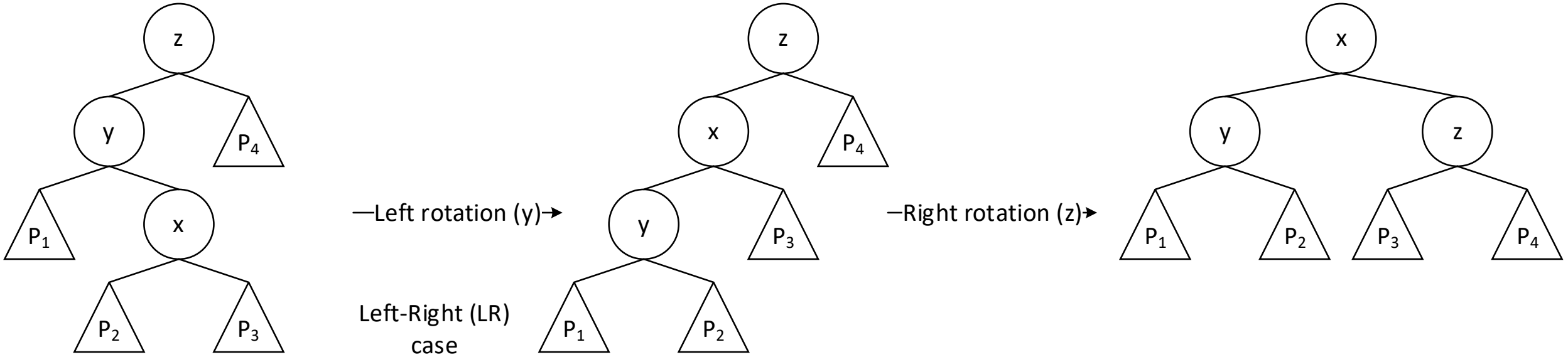
AVL Deletion: Left-Left Case

P_1, P_2, P_3 and P_4 are the tree branches



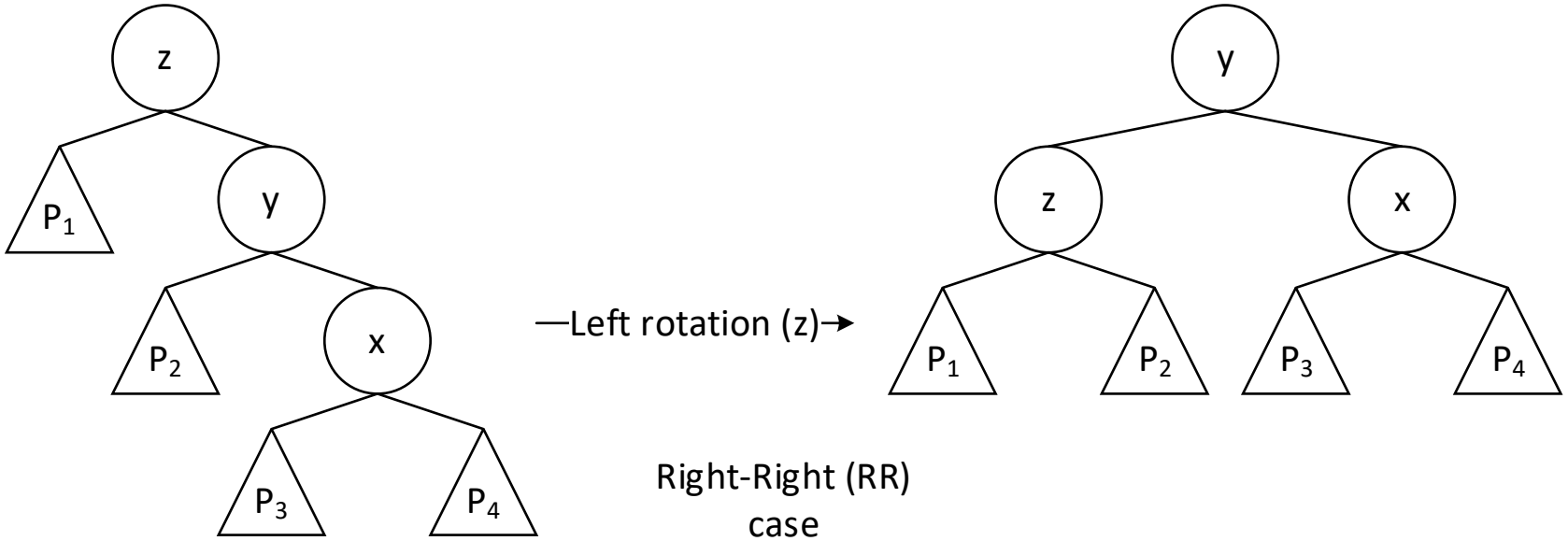
AVL Deletion: Left-Right Case

P_1, P_2, P_3 and P_4 are the tree branches



AVL Deletion: Right-Right Case

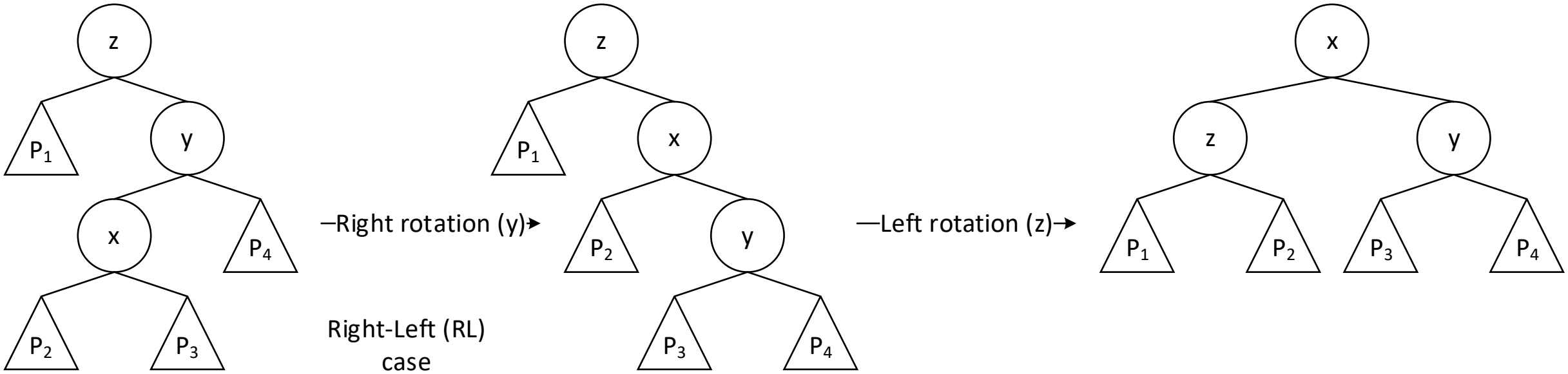
P_1, P_2, P_3 and P_4 are the tree branches



Right-Right (RR) case

AVL Deletion: Right-Left Case

P_1, P_2, P_3 and P_4 are the tree branches



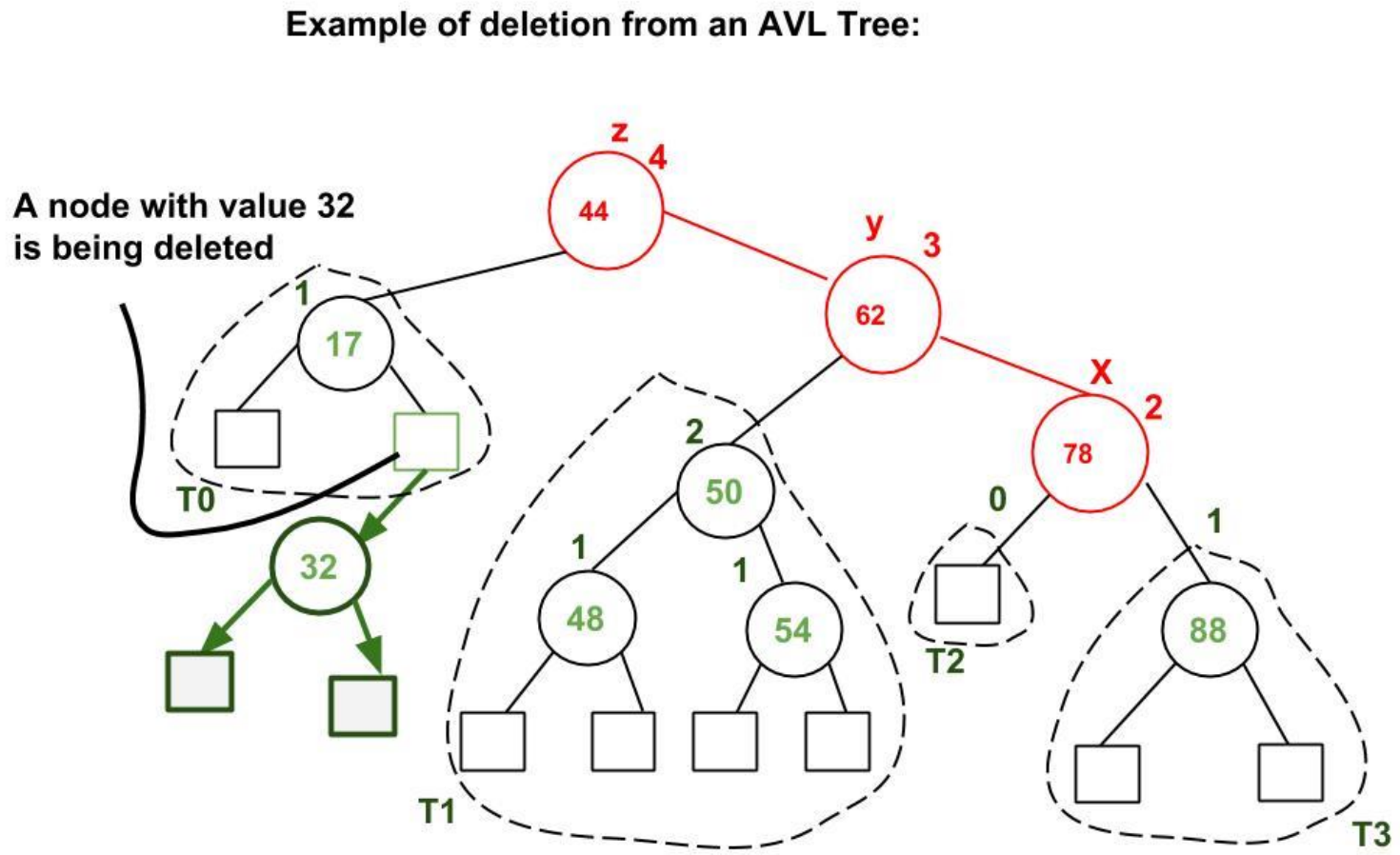
AVL: Deletion (continued)

- Not as in AVL *insertion*, in *deletion* after we did rotation on *z*, then we may do the rotation on the *ancestors* of *z* as well.
- That's why we have to continue our traversing up to the top root.

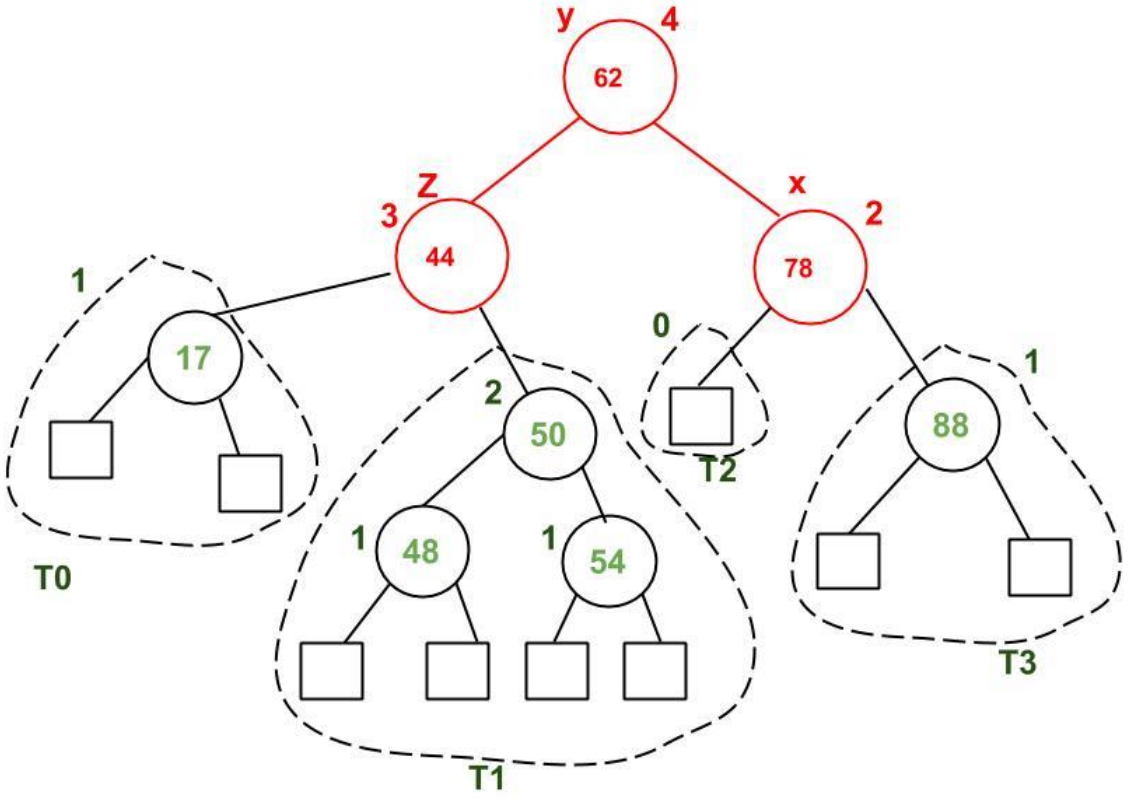
AVL Deletion: Example

- Node whose value 32 will be deleted
- After deleting 32, then go up and find the first imbalanced node, i.e., 44
- We marked 44 by *z*, then *z*'s highest child branch as *y*, i.e., 62, and *y*'s highest child branch as *x*, i.e., it can be 78 or 50 because both of them have the same height. Arbitrarily, 78 will be chosen.
- Now, the case is Right-Right one, so the left rotation will be done

AVL Deletion: Example (continued)



AVL Deletion: Example (continued)



AVLTree.java

```java
package AVLTree;

/**
 *
 * AVLTree class extended from Tree class.
 * Added a variable "height" for performing height-balanced efficiently
 */

public class AVLTree extends Tree {

    protected final boolean empty;
    protected final int value;
    protected final AVLTree left, right;
    protected final int height;

    /**
     * Creates a new Tree whose root value is x and left and right subtrees are r
     * and l
     */
    public AVLTree(int value, AVLTree left, AVLTree right) {
        this.empty = false;
        this.value = value;
        this.left = left;
        this.right = right;
        this.height = 1 + Math.max(left.getHeight(), right.getHeight());
    }

    /**
     * Creates an empty tree
     */
    public AVLTree() {
        this.empty = true;
        this.value = 0;
        this.left = null;
        this.right = null;
        this.height = 0;
    }

    /**
     * Creates a tree with a single node
     */
    public AVLTree(int value) {
        this.empty = false;
        this.value = value;
        this.left = new AVLTree();
        this.right = new AVLTree();
        this.height = 1;
    }

    /**
     * Gets the height
     */
    private int getHeight() {
        return height;
    }

}
```