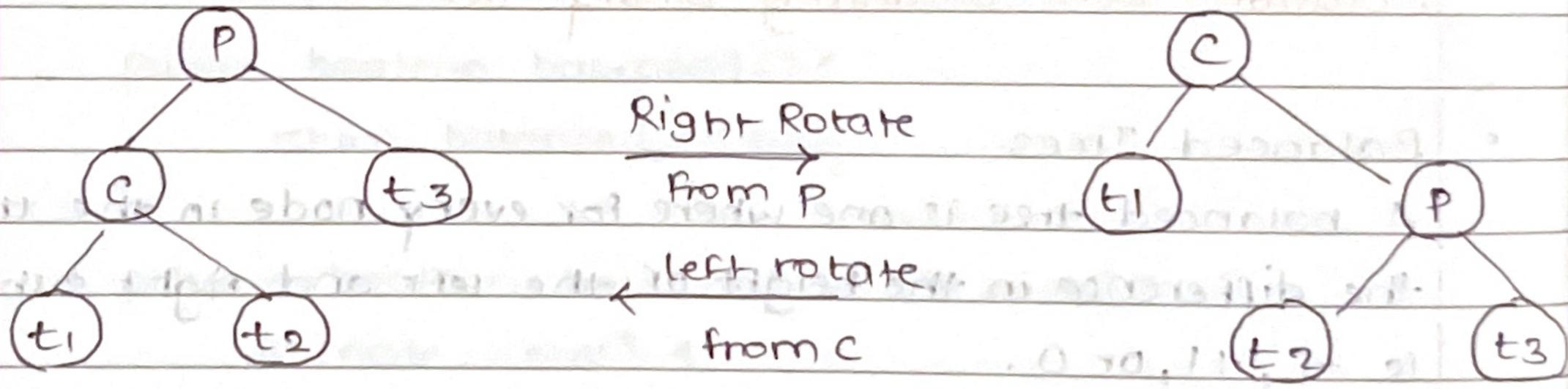
- AVL Tree

In AVL Tree defined as self-balancing Binary Search Tree where the difference between heights of left and right subtrees for any node cannot be more than one.
It is named after its inventors, Adelson-Velsky and Landis

Algorithm

1. Insert Normally: Insert the new node as you would in a normal BST.
2. Find unbalanced Node: After insertion, starting from the inserted node move upwards to find the first node that make tree unbalanced
3. Rotate to Balance: Apply one of the four rotation rules to rebalance the tree.

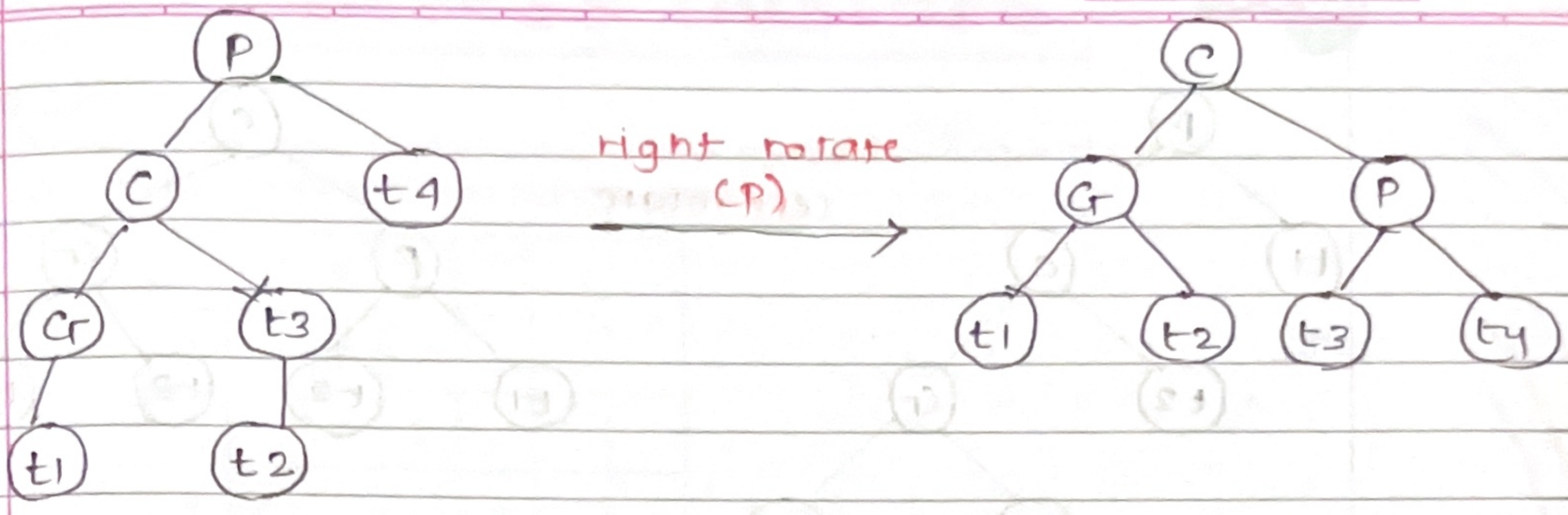
Right Rotate/left Rotate



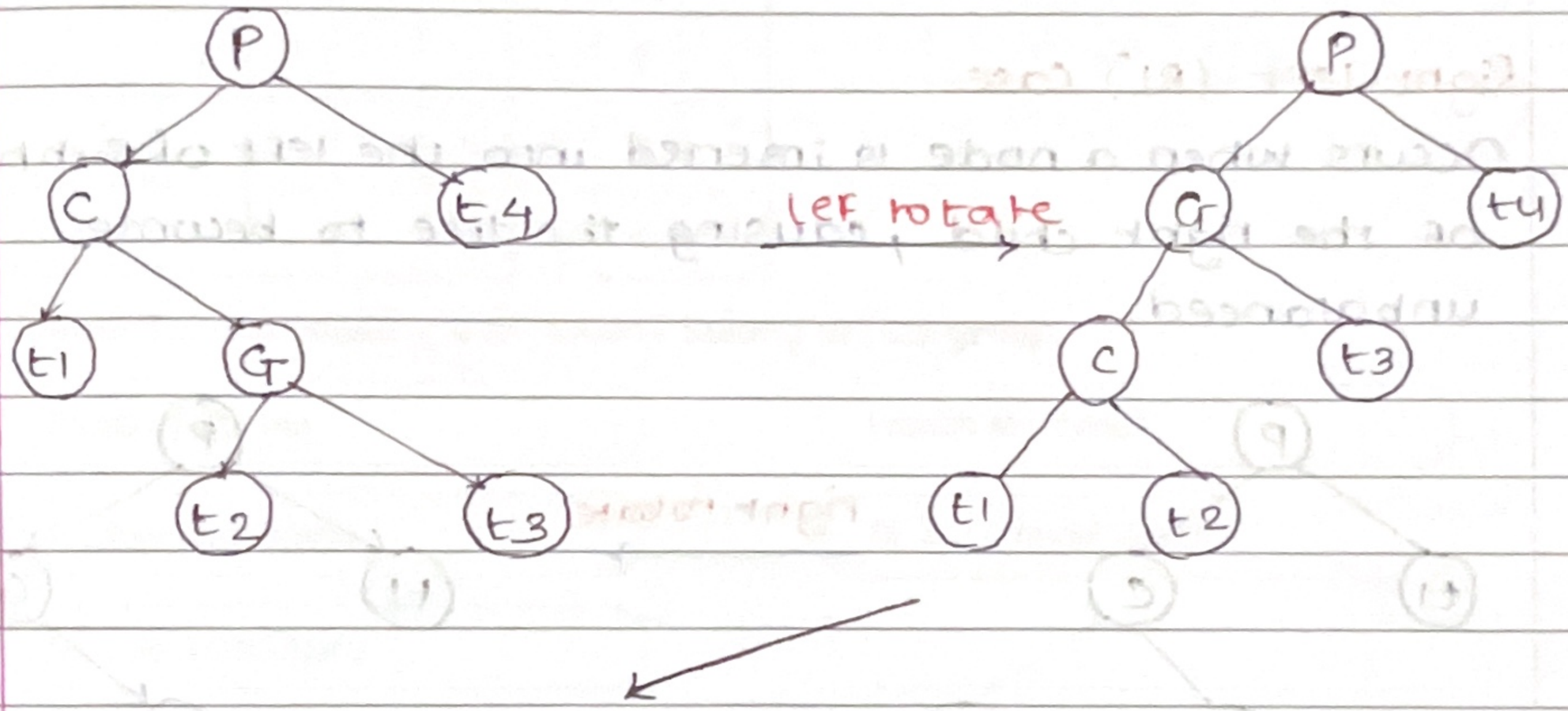
4 Rotation Rules

1. Left-Left (LL) case:

Occurs when a new node is inserted into the left subtree of the left child, causing the tree to become unbalanced.



2. Left-Right (LR) Case

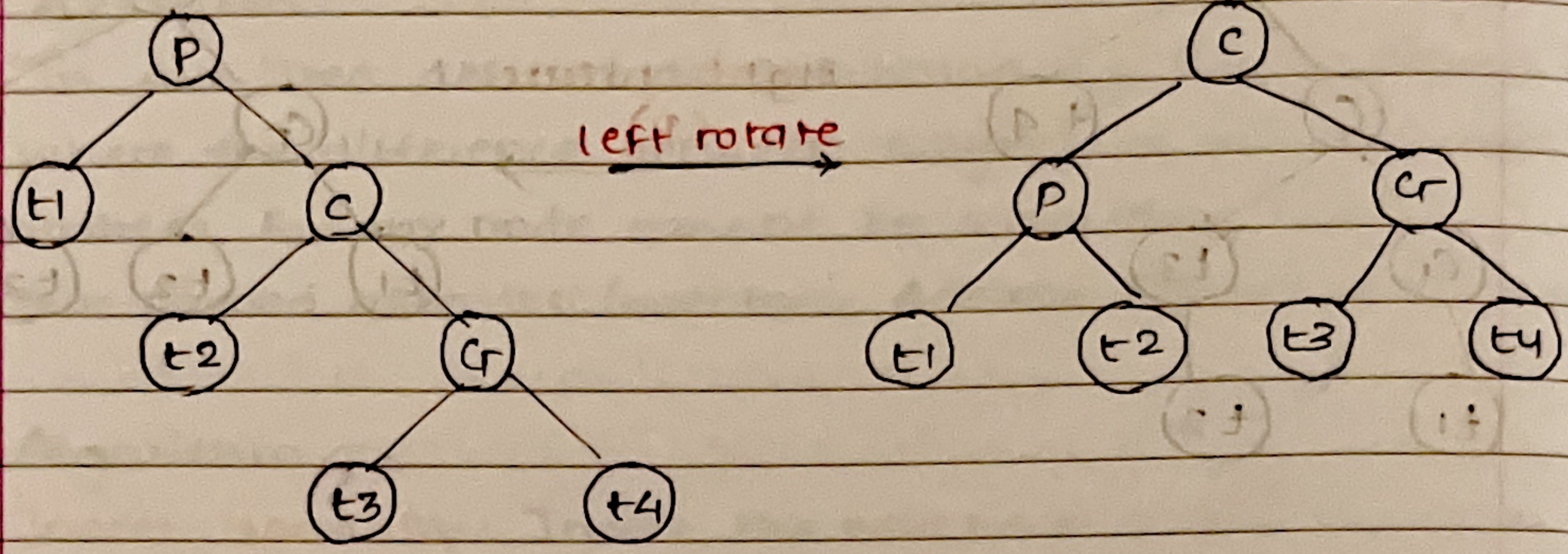


right rotate
same as 1st rule

∴ Occurs when a new node inserted into the right subtree of the left child, causing the tree to become unbalanced.

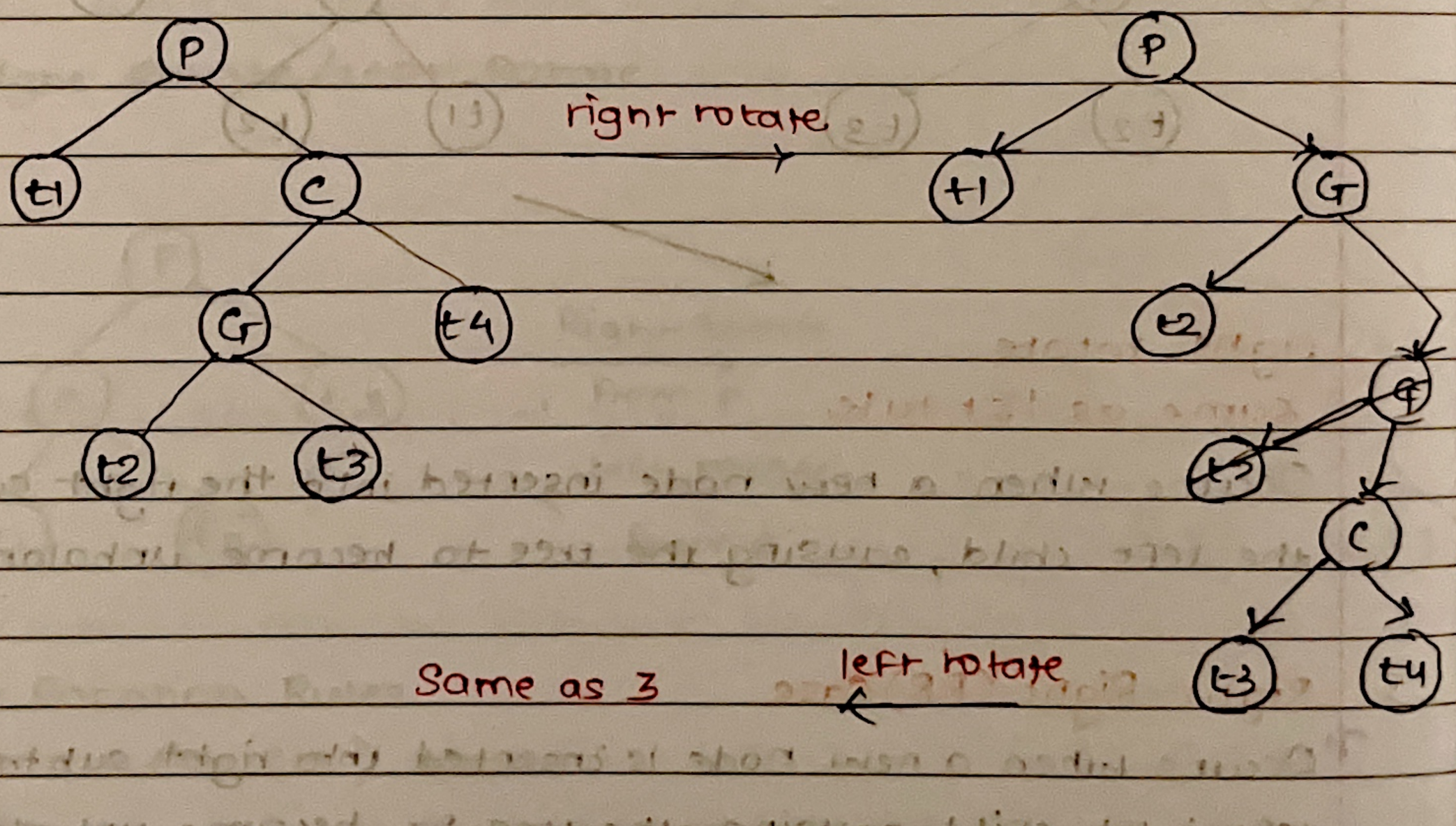
3. Right-Right (RR) Case

Occurs when a new node is inserted into right subtree of the ~~left~~ right child causing the tree to become unbalanced.



4. Right-left (RL) case.

Occurs when a node is inserted into the left of subtree of the right child, causing the tree to become unbalanced.



Implementation:

```
class AVL {
    public class Node {
        private int value;
        private Node left;
        private Node right;
        private int height;

        public Node(int value) {
            this.value = value;
        }

        public int getValue() {
            return value;
        }
    }
    private Node root;

    public AVL() { }
    public int height(Node node) {
        if(node == null) {
            return -1;
        }
        return node.height;
    }
    public Node insert(int value) {
        if (node == null) {
            node = new Node(value);
            return Node;
        }
```

```
if (value < node.value) {
        node.left = insert(value, node.left);
    }
    if (value > node..value) {
        node.right = insert(value, node.right);
    }
    node.height = Math.max(height(node.left),
                           height(node.right)) +1;
    return rotate(node);
}

private Node rotate(Node node) {
    if(height(node.left) - height(node.right)>0) {
        //left heavy
        if(height(node.left.left) - height(node.left.right)>0)
        { // left left case
            Noderetrurn rightRotate(node);
        }
        if(height(node.left.left) - height(node.left.right)<0){
            // left right case
            node.left = leftRotate(node.left);
            return rightRotate(node);
        }
    }
    if(height(Node.left) - height(node.right) <-1) {
        // right heavy
        if(height(node.right.left) - height(node.right.right)
        >0) {
            // right right case
            return leftRotate(node);
        }
        if(height(node.right.left) - height(node.right.
        right >0) {
            // left right case
```

```
            node.right = rightRotate(node.right);
            return leftRotate(node);
        }
    }
    return node;
}

public Node RightRotate(Node p) {
        Node c = p.left;
        Node t = c.right;
        c.right = p;
        p.left = t;
        p.height = Math.max(height(p.left), height(p.right)
            +1);
        c.height = Math.max(height(c.left), height(c.right)+1
        return c;
}

public Node leftRotate(Node c) {
        Node p = c.right;
        Node t = p.right;
        p.left = c;
        c.right = t;
        p.height = Math.max(height(p.left), height(p.right)
            +1);
        c.height = Math.max(height(c.left), height(c.right)
            +1);
        return p;
}
```