# AVL树

# AVL树

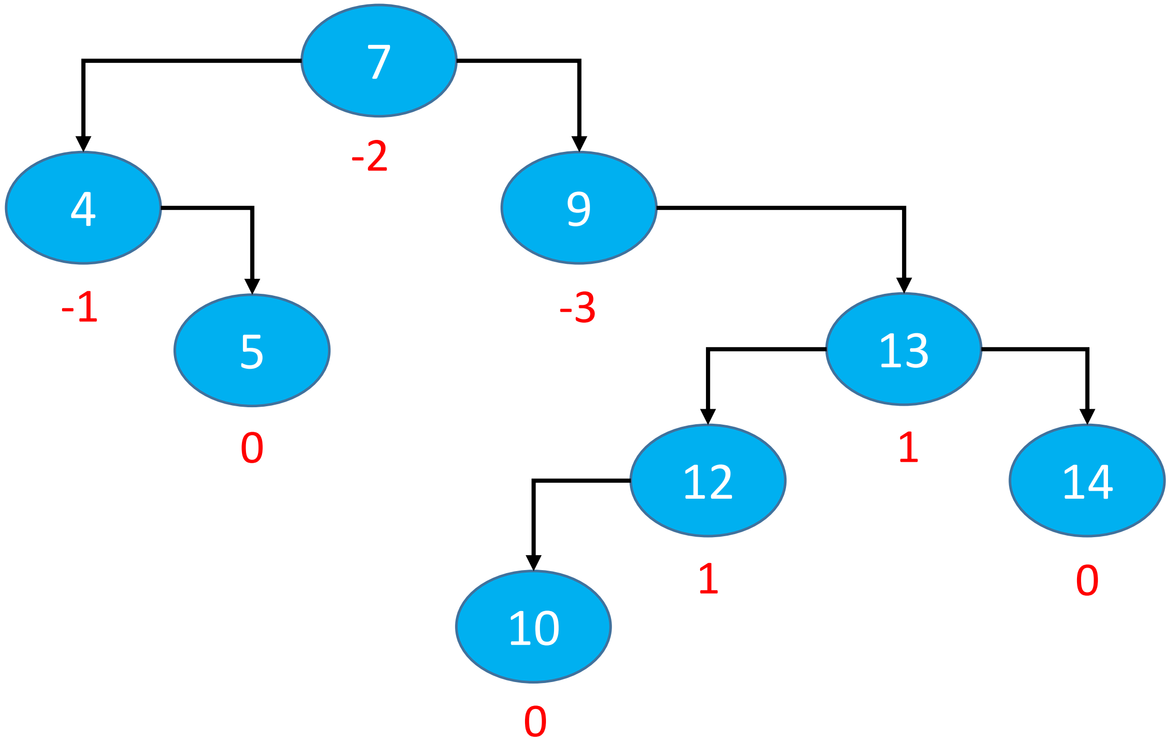
- AVL树是最早发明的自平衡二叉搜索树之一

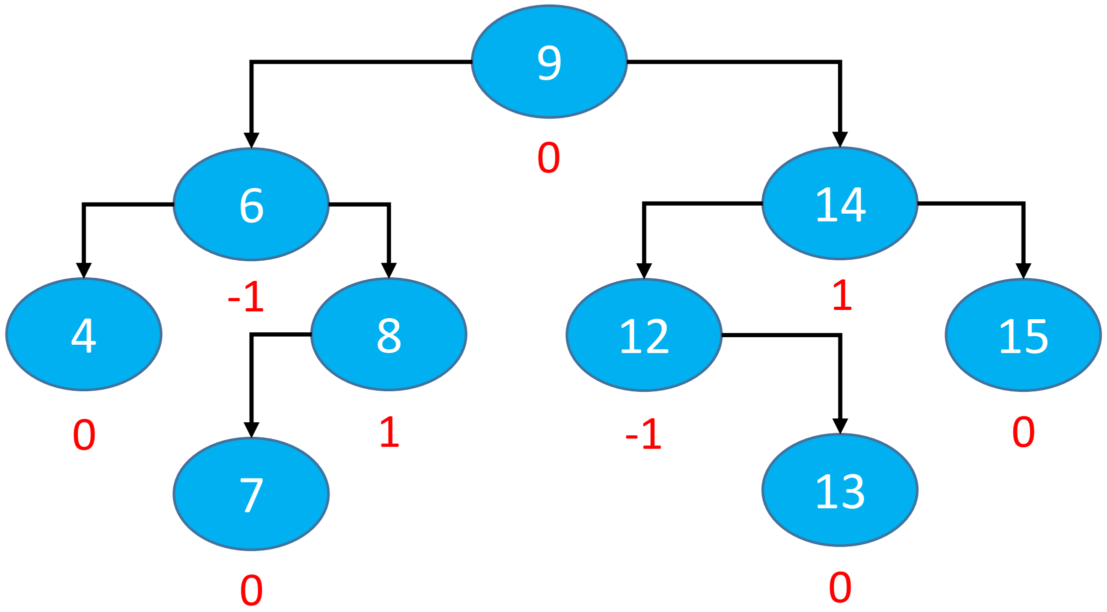
- AVL 取名于两位发明者的名字
  - G. M. Adelson-Velsky 和 E. M. Landis（来自苏联的科学家）

- Something interesting
  - 有人把AVL树念做“艾薇儿树”
  - 加拿大女歌手，几首不错的歌：《Complicated》、《When You're Gone》、《Innocence》

# AVL树

- 平衡因子（Balance Factor）：某结点的左右子树的高度差
- AVL树的特点
  - 每个节点的平衡因子只可能是 1、0、-1（绝对值 ≤ 1，如果超过 1，称之为“失衡”）
  - 每个节点的左右子树高度差不超过 1
  - 搜索、添加、删除的时间复杂度是 O(logn)

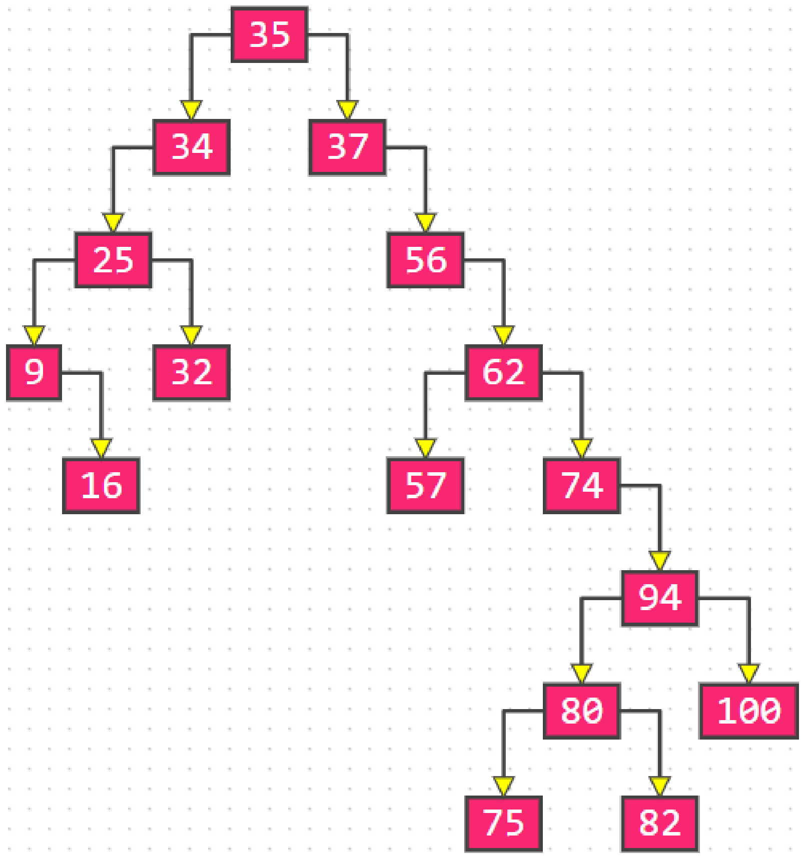


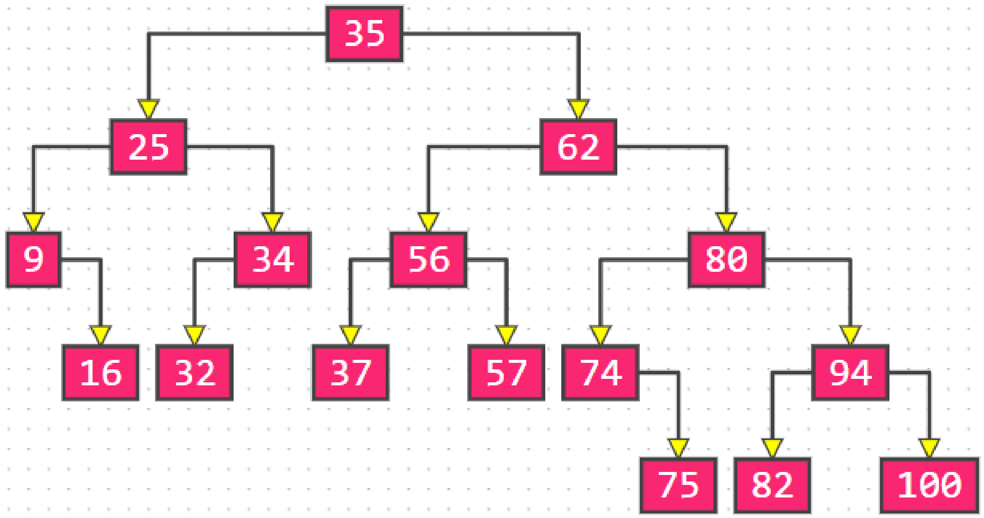
# 平衡对比

- 输入数据：35, 37, 34, 56, 25, 62, 57, 9, 74, 32, 94, 80, 75, 100, 16, 82

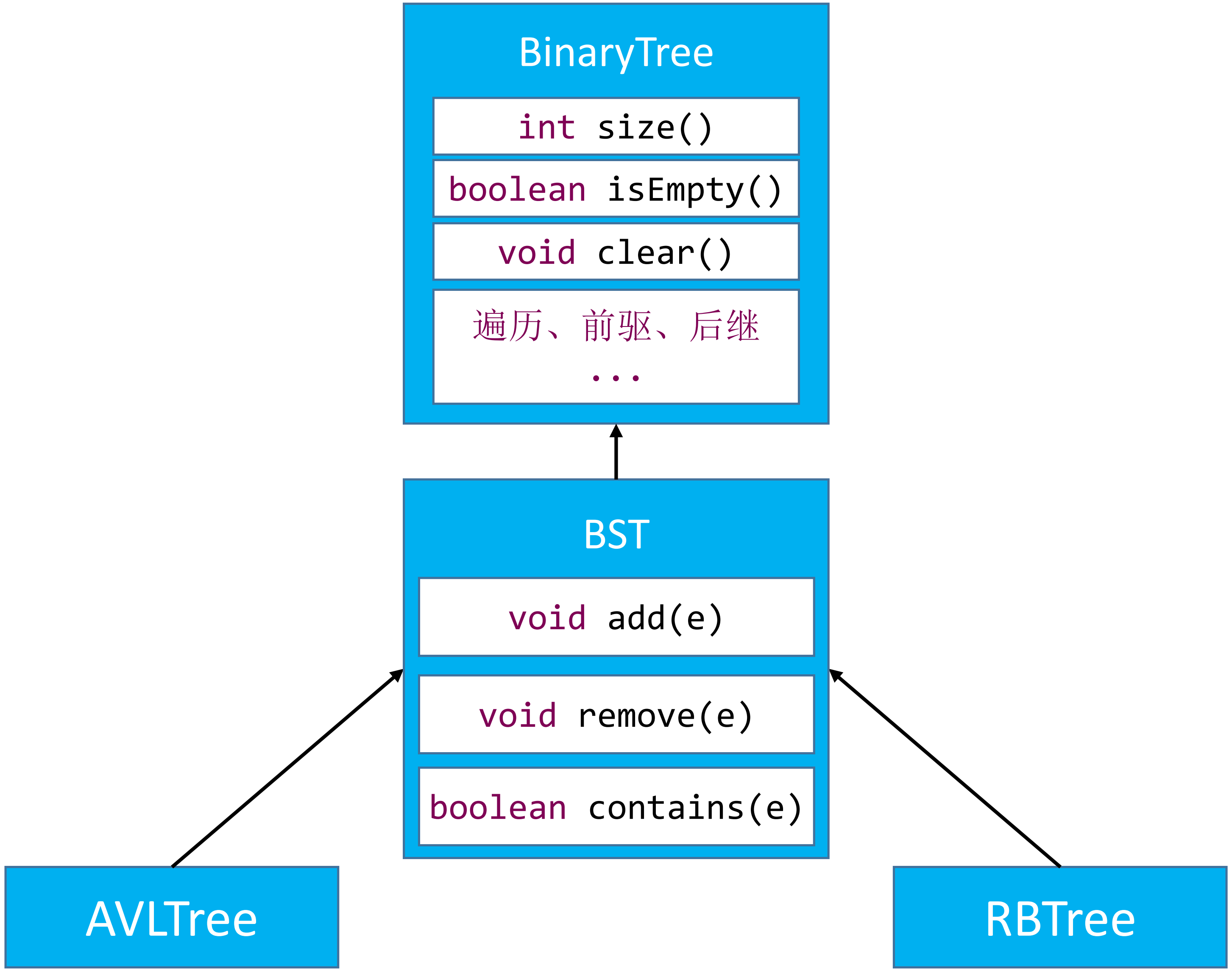
普通二叉搜索树




AVL树

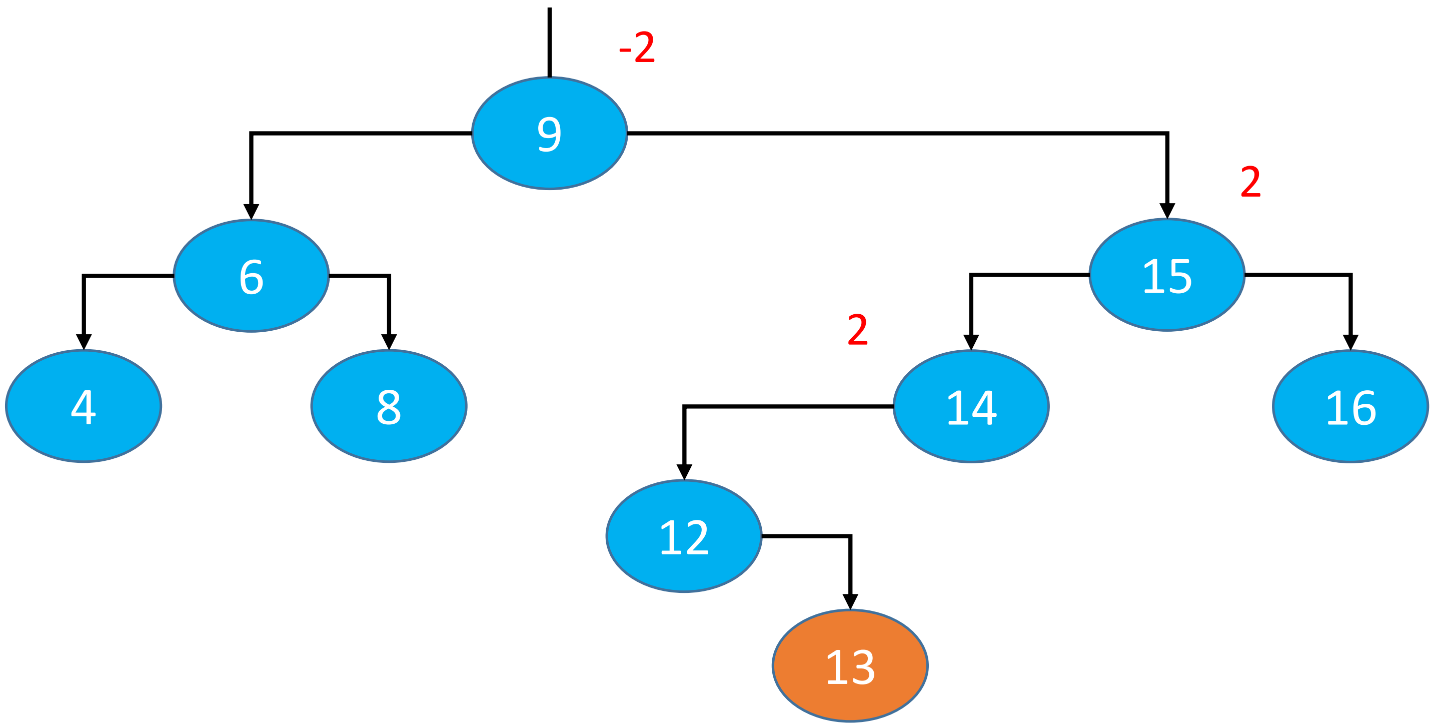


# 简单的继承结构


BinaryTree
int size()
boolean isEmpty()
void clear()
遍历、前驱、后继
...
BST
void add(e)
void remove(e)
boolean contains(e)
AVLTree
RBTree

# 添加导致的失衡

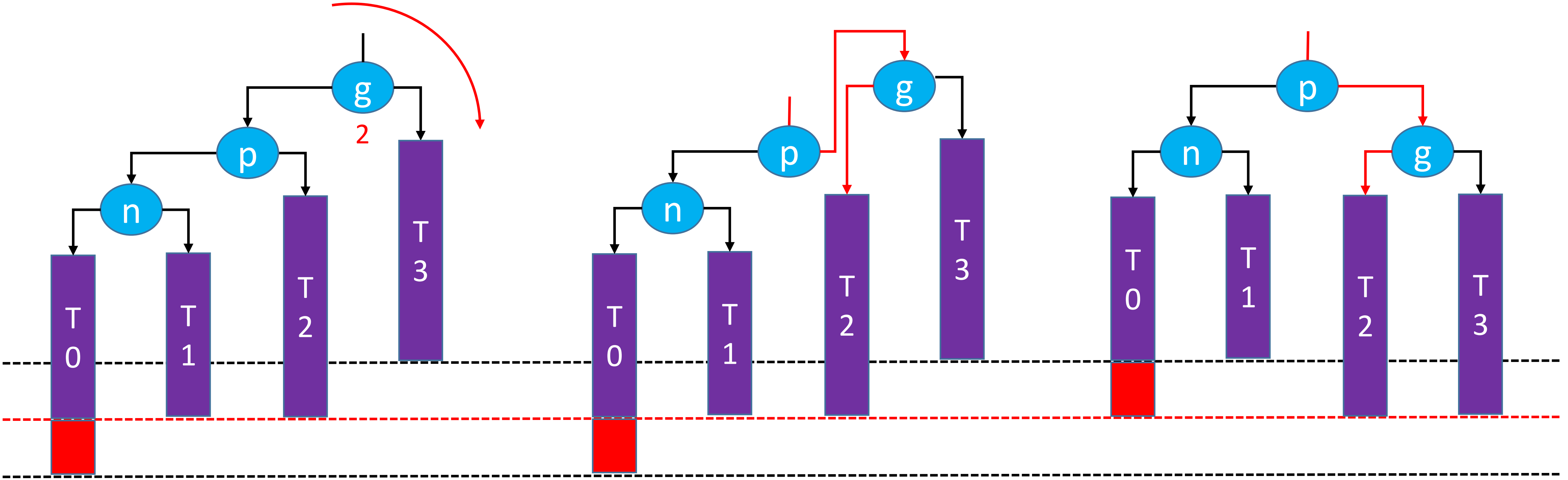
- 示例：往下面这棵子树中添加 13
- 最坏情况：可能会导致所有祖先节点都失衡
- 父节点、非祖先节点，都不可能失衡



# LL – 右旋转（单旋）

- g.left = p.right
- p.right = g
- 让p成为这棵子树的根节点
- 仍然是一棵二叉搜索树：T0 < n < T1 < p < T2 < g < T3
- 整棵树都达到平衡

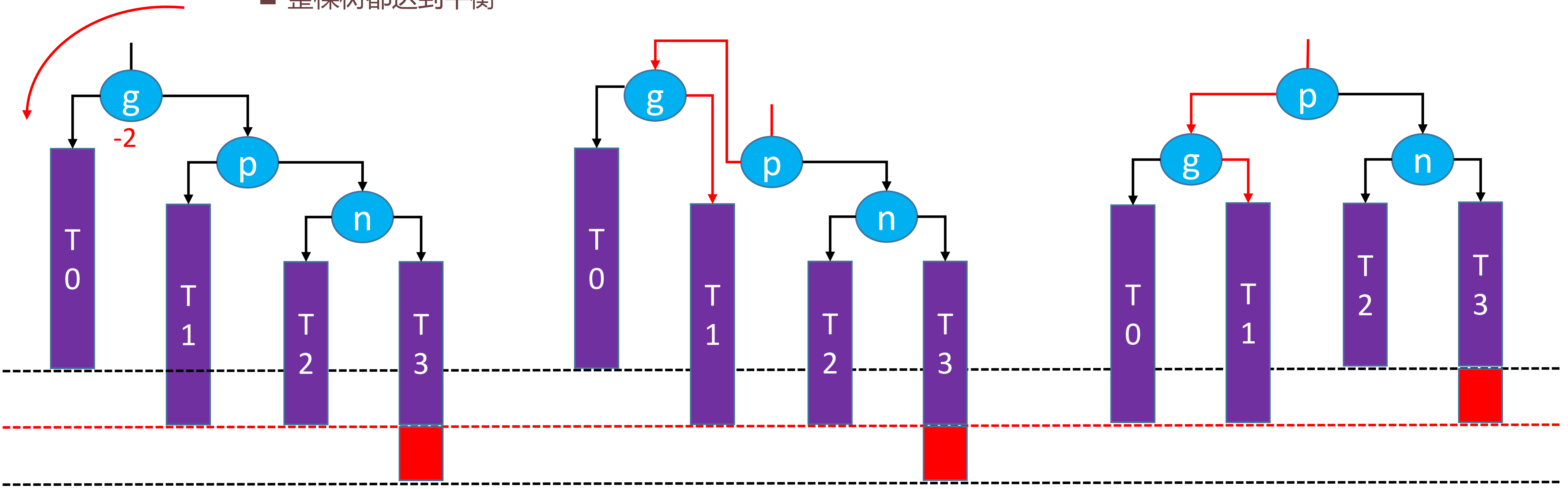
- 还需要注意维护的内容
  - T2、p、g 的 parent 属性
  - 先后更新 g、p 的高度



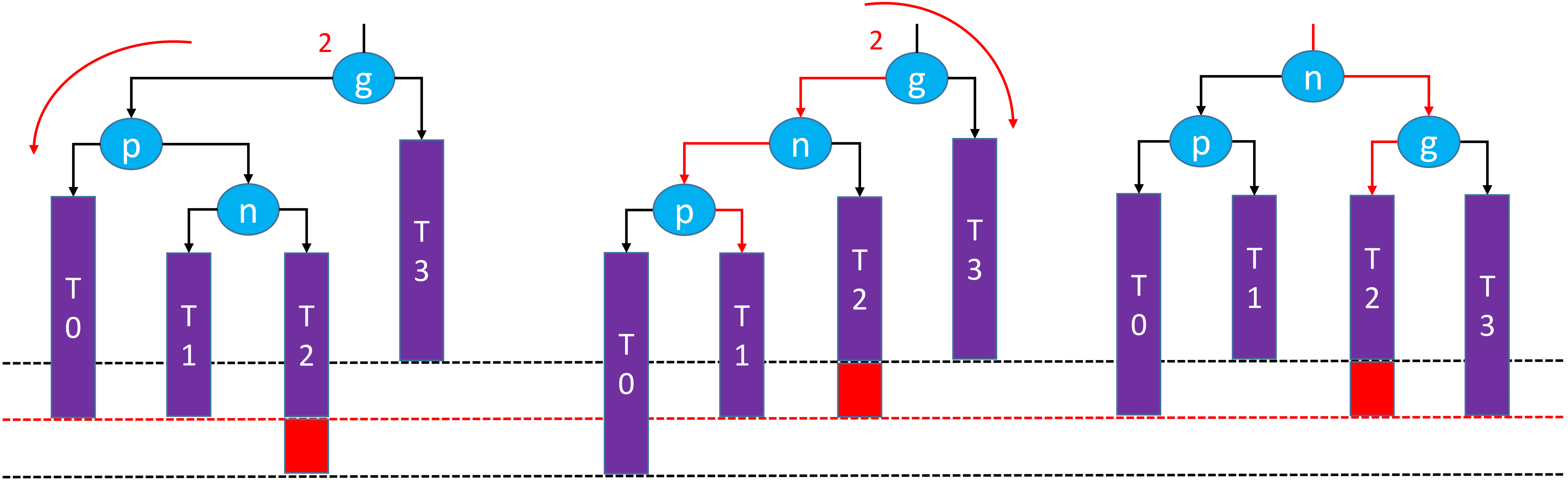
# RR – 左旋转（单旋）

- g.right = p.left
- p.left = g
- 让p成为这棵子树的根节点
- 仍然是一棵二叉搜索树：T0 < g < T1 < p < T2 < n < T3
- 整棵树都达到平衡

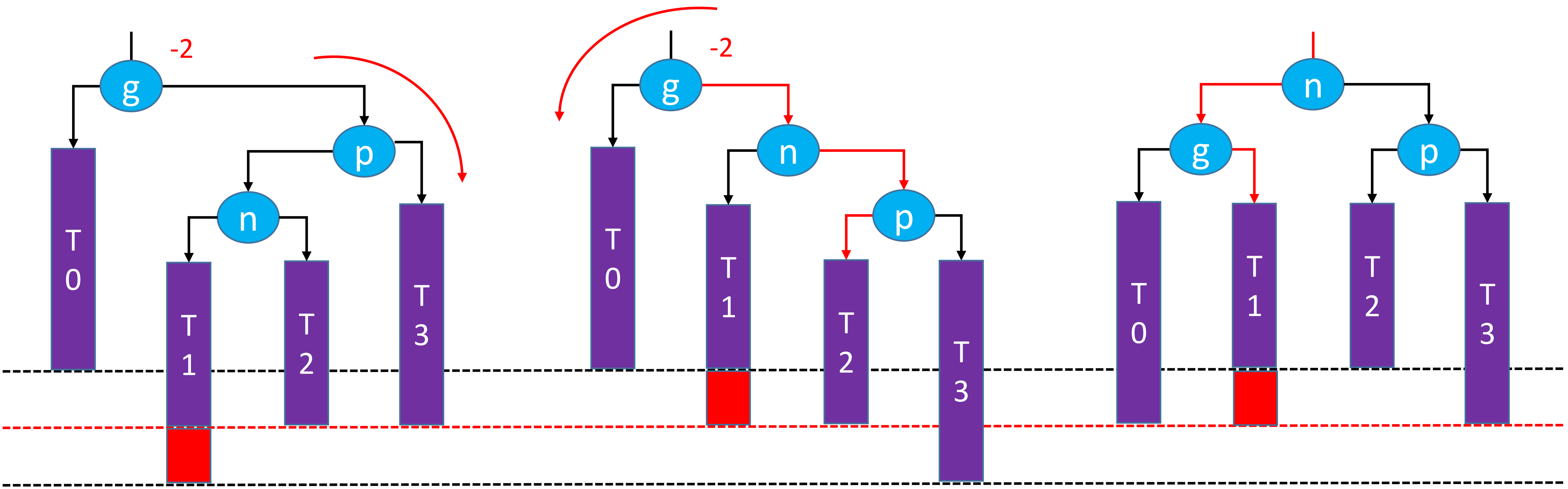
- 还需要注意维护的内容
  - T1、p、g 的 parent 属性
  - 先后更新 g、p 的高度



# LR – RR左旋转，LL右旋转（双旋）



# RL – LL右旋转，RR左旋转（双旋）



# zig、zag

- 有些教程里面
  - 把右旋转叫做zig，旋转之后的状态叫做zigged
  - 把左旋转叫做zag，旋转之后的状态叫做zagged

# 添加之后的修复

```java
@Override
protected void afterAdd(Node<E> node) {
    while ((node = node.parent) != null) {
        if (isBalanced(node)) {
            // 更新高度
            updateHeight(node);
        } else {
            // 恢复平衡
            rebalance(node);
            break;
        }
    }
}
```

```java
/**
 * 重新恢复平衡
 * @param grand 高度最低的不平衡节点
 */
protected void rebalance(Node<E> grand) {
    Node<E> parent = grand.tallerChild();
    Node<E> node = parent.tallerChild();
    if (parent.isLeftChild()) { // L
        if (node.isLeftChild()) { // LL
            rotateRight(grand);
        } else { // LR
            rotateLeft(parent);
            rotateRight(grand);
        }
    } else { // R
        if (node.isLeftChild()) { // RL
            rotateRight(parent);
            rotateLeft(grand);
        } else { // RR
            rotateLeft(grand);
        }
    }
}
```

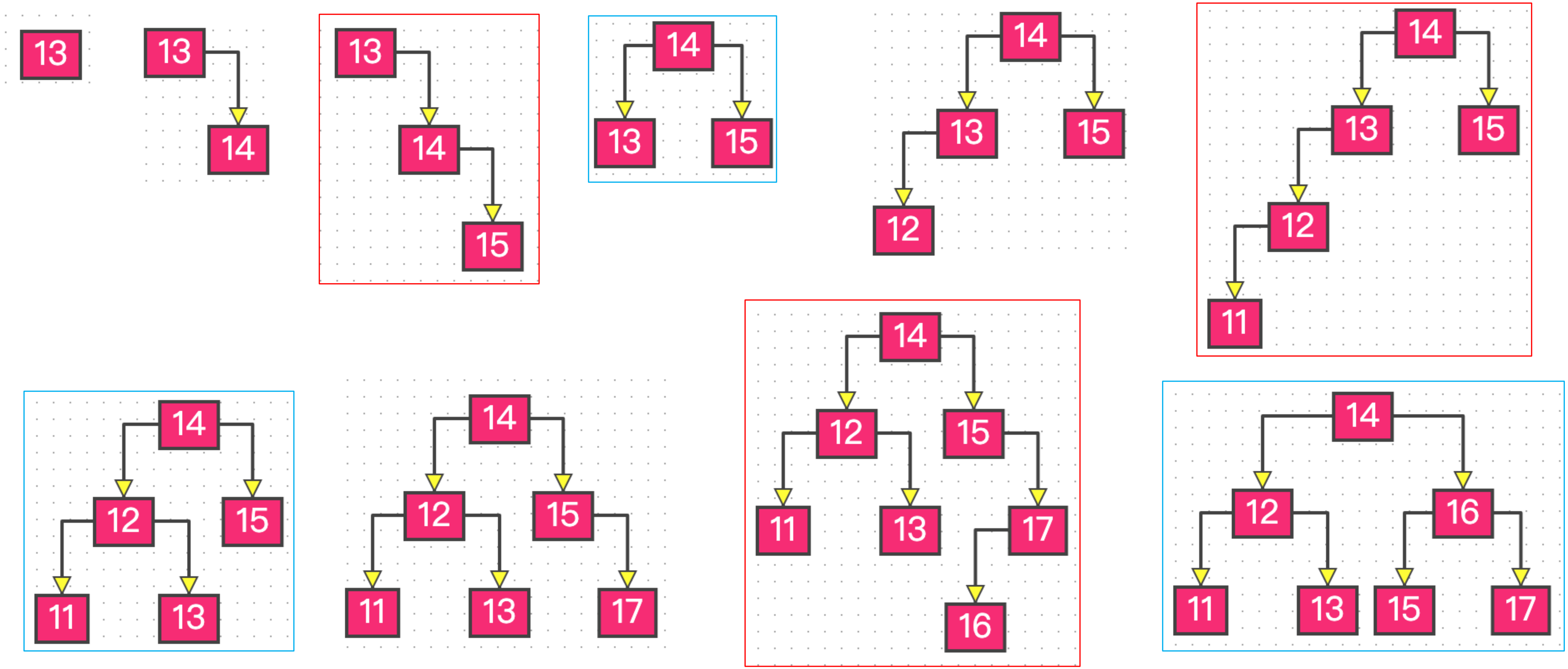
# 旋转

```java
protected void rotateLeft(Node<E> grand) {
    // 交换子树
    Node<E> parent = grand.right;
    Node<E> child = parent.left;
    grand.right = child;
    parent.left = grand;
    // 维护parent和height
    afterRotate(grand, parent, child);
}

protected void rotateRight(Node<E> grand) {
    // 交换子树
    Node<E> parent = grand.left;
    Node<E> child = parent.right;
    grand.left = child;
    parent.right = grand;
    // 维护parent和height
    afterRotate(grand, parent, child);
}
```

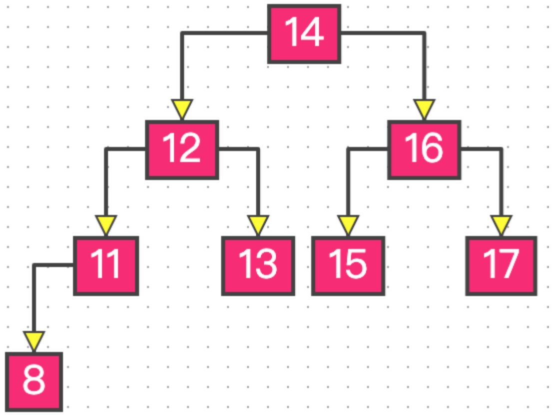
```java
/**
 * 公共代码：不管是左旋转、右旋转，都要执行的
 * @param grand 失衡节点
 * @param parent 失衡节点的tallerChild
 * @param child g和p需要交换的子树（本来是p的子树，后面会变成g的子树）
 */
protected void afterRotate(Node<E> grand, Node<E> parent, Node<E> child) {
    // 子树的根节点嫁接到原树中
    if (grand.isLeftChild()) {
        grand.parent.left = parent;
    } else if (grand.isRightChild()) {
        grand.parent.right = parent;
    } else {
        root = parent;
    }

    // parent维护
    if (child != null) {
        child.parent = grand;
    }
    parent.parent = grand.parent;
    grand.parent = parent;

    // 更新高度（先更新比较矮的grand，再更新比较高的parent）
    updateHeight(grand);
    updateHeight(parent);
}
```

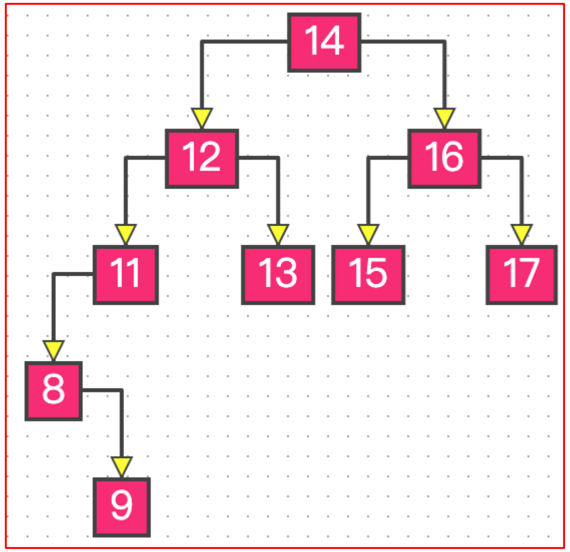
# 示例

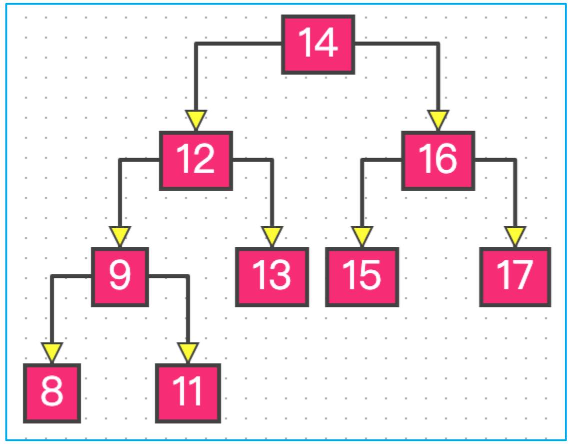
- 输入数据：13, 14, 15, 12, 11, 17, 16, 8, 9, 1

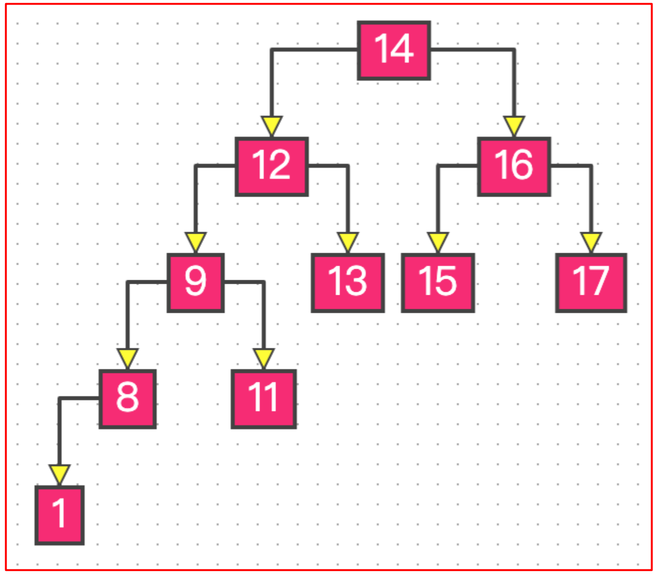


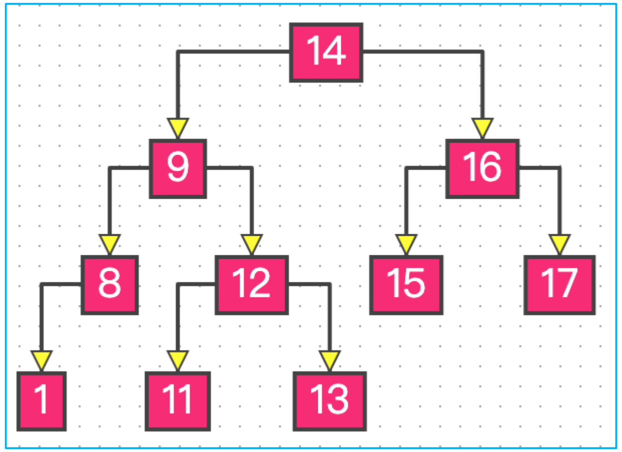
# 示例


14
12
16
11
13
15
17
8


14
12
16
11
13
15
17
8
9

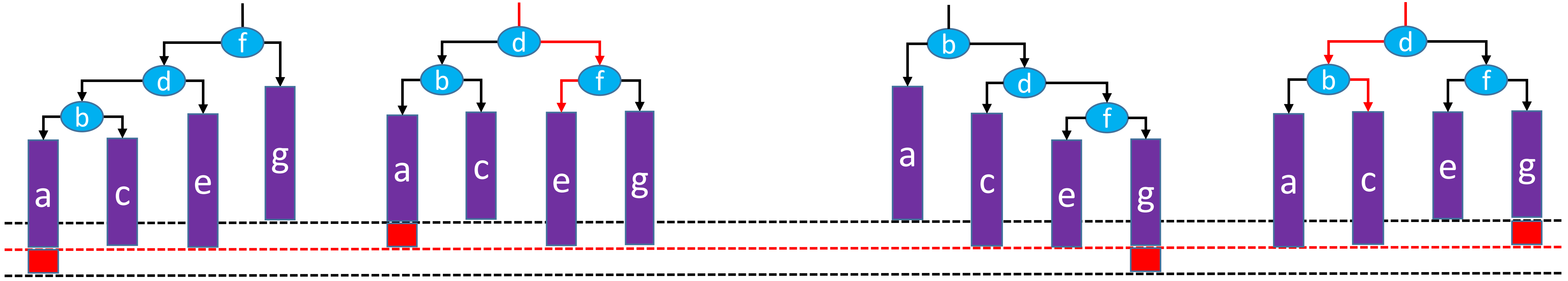

14
12
16
9
13
15
17
8
11


14
12
16
9
13
15
17
8
11
1

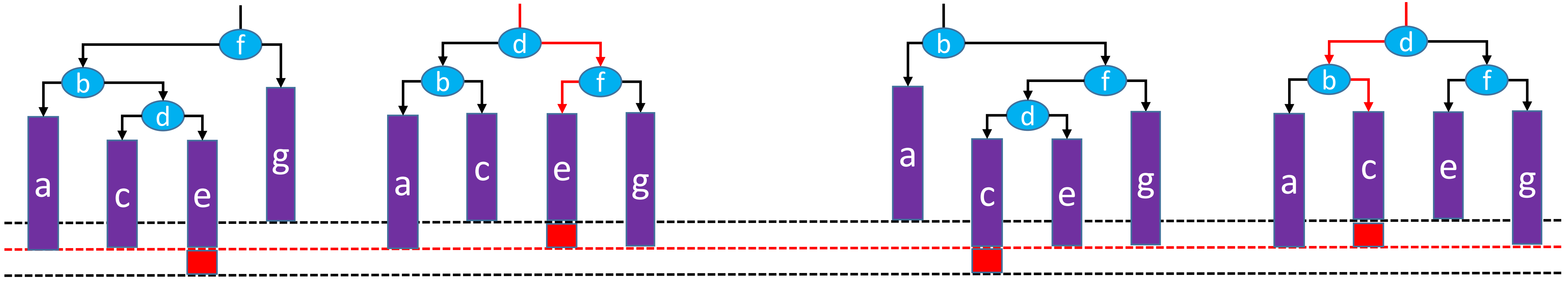

14
9
16
8
12
15
17
1
11
13

# 统一所有旋转操作

LL　　RR




LR　　RL

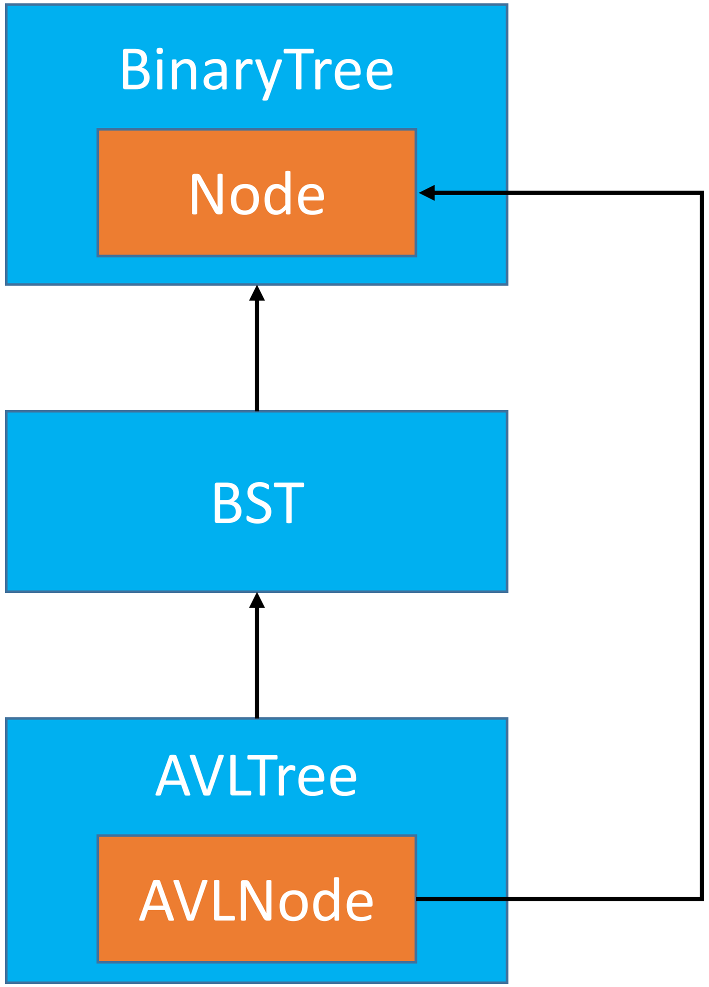


# 统一所有旋转操作

```java
private void rotate(
        Node<E> r, // 子树的根节点
        Node<E> b, Node<E> c, Node<E> d, Node<E> e, Node<E> f) {
    // 让d成为这棵子树的根节点
    d.parent = r.parent;
    if (r.isLeftChild()) {
        r.parent.left = d;
    } else if (r.isRightChild()) {
        r.parent.right = d;
    } else {
        root = d;
    }

    // b-c
    b.right = c;
    if (c != null) c.parent = b;
    updateHeight(b);

    // e-f
    f.left = e;
    if (e != null) e.parent = f;
    updateHeight(f);

    // b-d-f
    d.left = b;
    d.right = f;
    b.parent = d;
    f.parent = d;
    updateHeight(d);
}
```

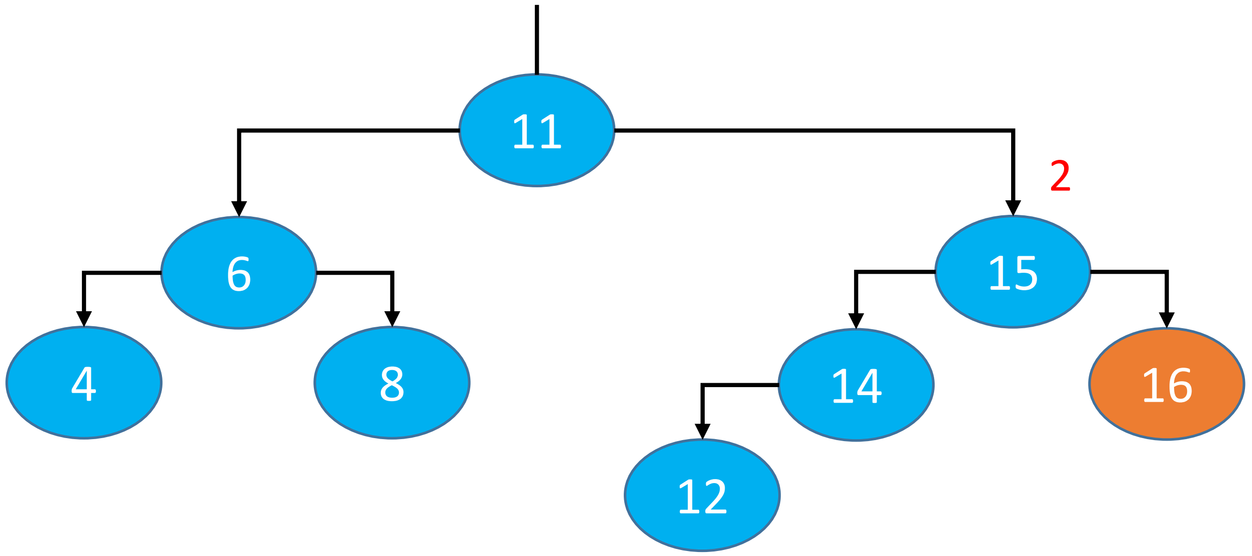
```java
private void rebalance(Node<E> grand) {
    Node<E> parent = ((AVLNode<E>)grand).tallerChild();
    Node<E> node = ((AVLNode<E>)parent).tallerChild();
    if (parent.isLeftChild()) { // L
        if (node.isLeftChild()) { // LL
            rotate(grand, node, node.right, parent, parent.right, grand);
        } else { // LR
            rotate(grand, parent, node.left, node, node.right, grand);
        }
    } else { // R
        if (node.isLeftChild()) { // RL
            rotate(grand, grand, node.left, node, node.right, parent);
        } else { // RR
            rotate(grand, grand, parent.left, parent, node.left, node);
        }
    }
}
```

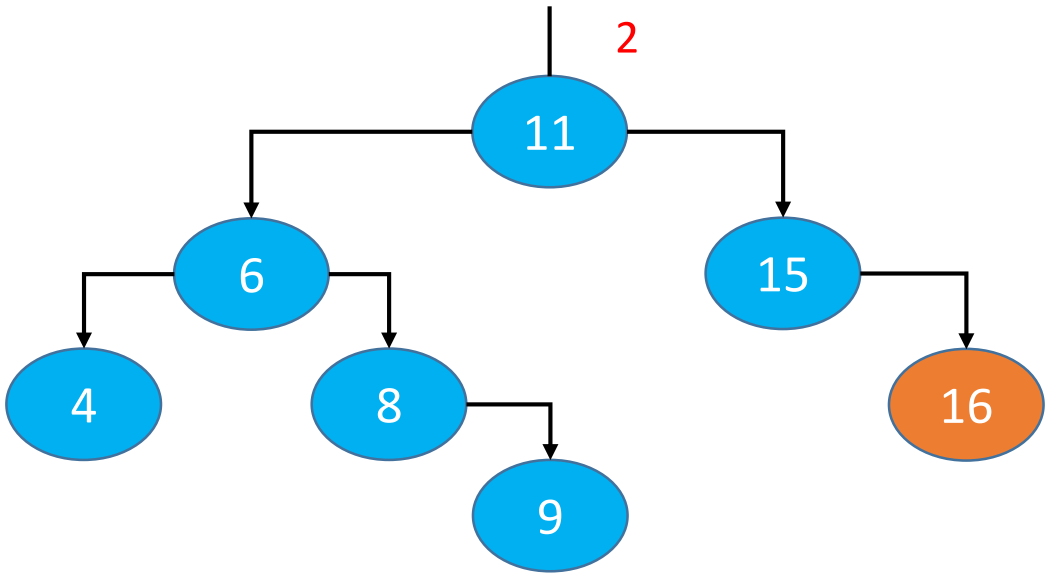
# 独立出AVLNode



# 删除导致的失衡

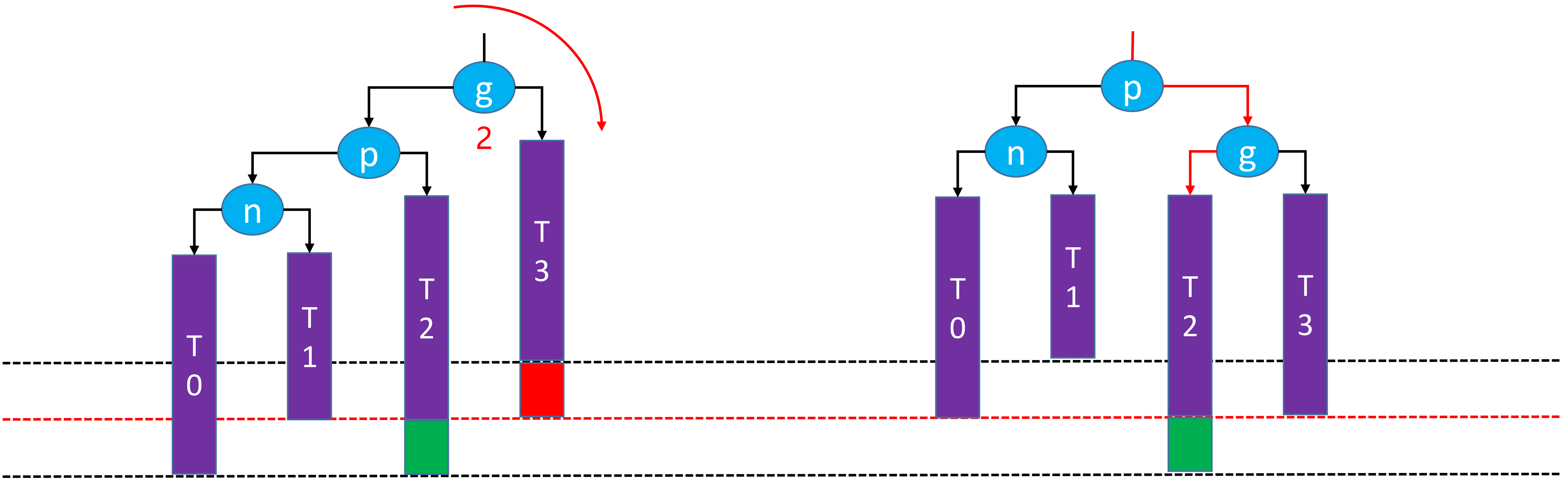
- 示例：删除子树中的 16
- 可能会导致父节点或祖先节点失衡（只有1个节点会失衡），其他节点，都不可能失衡



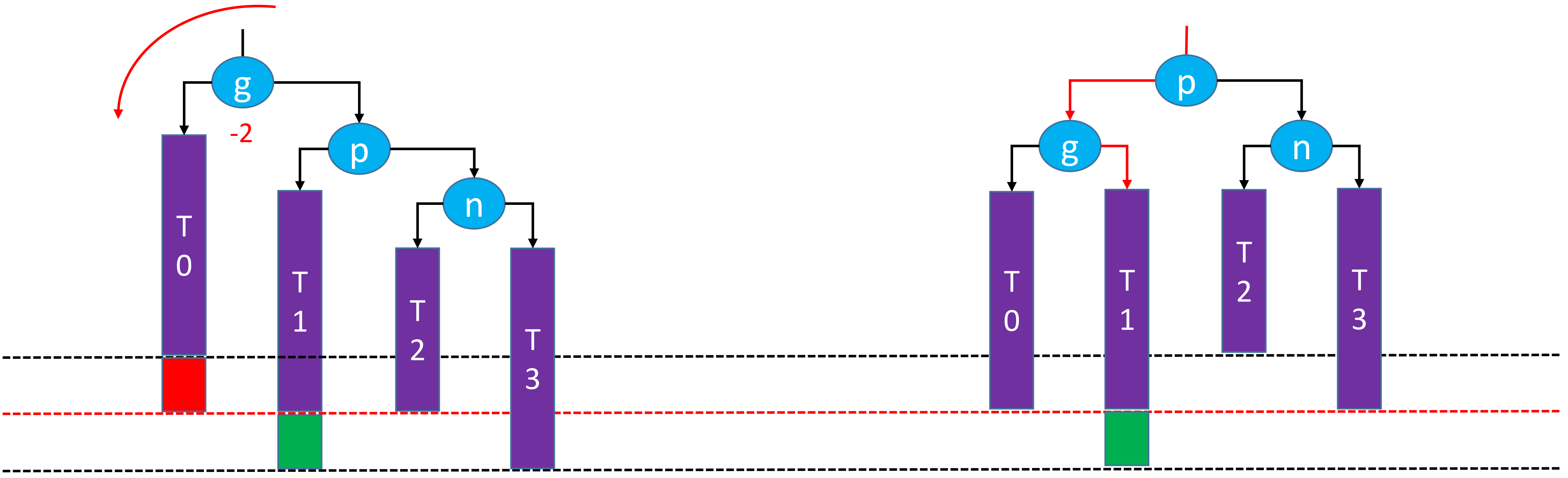



# LL – 右旋转（单旋）

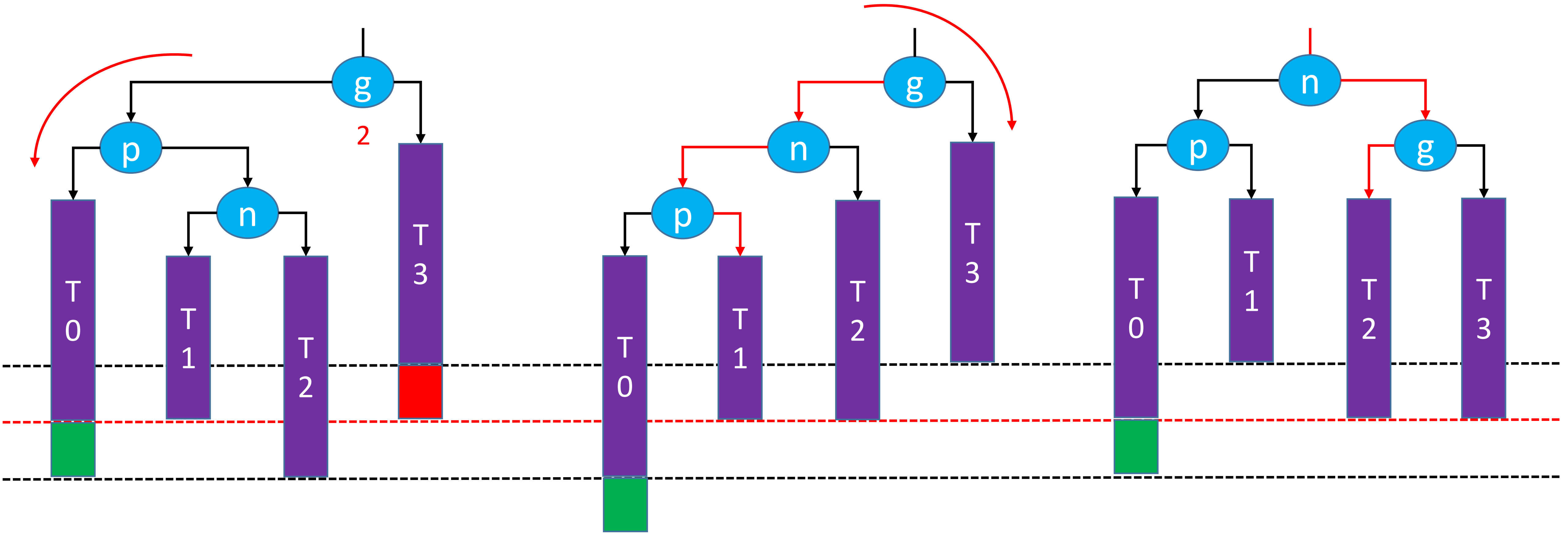
- 如果绿色节点不存在，更高层的祖先节点可能也会失衡，需要再次恢复平衡，然后又可能导致更高层的祖先节点失衡...
- 极端情况下，所有祖先节点都需要进行恢复平衡的操作，共 O(logn) 次调整



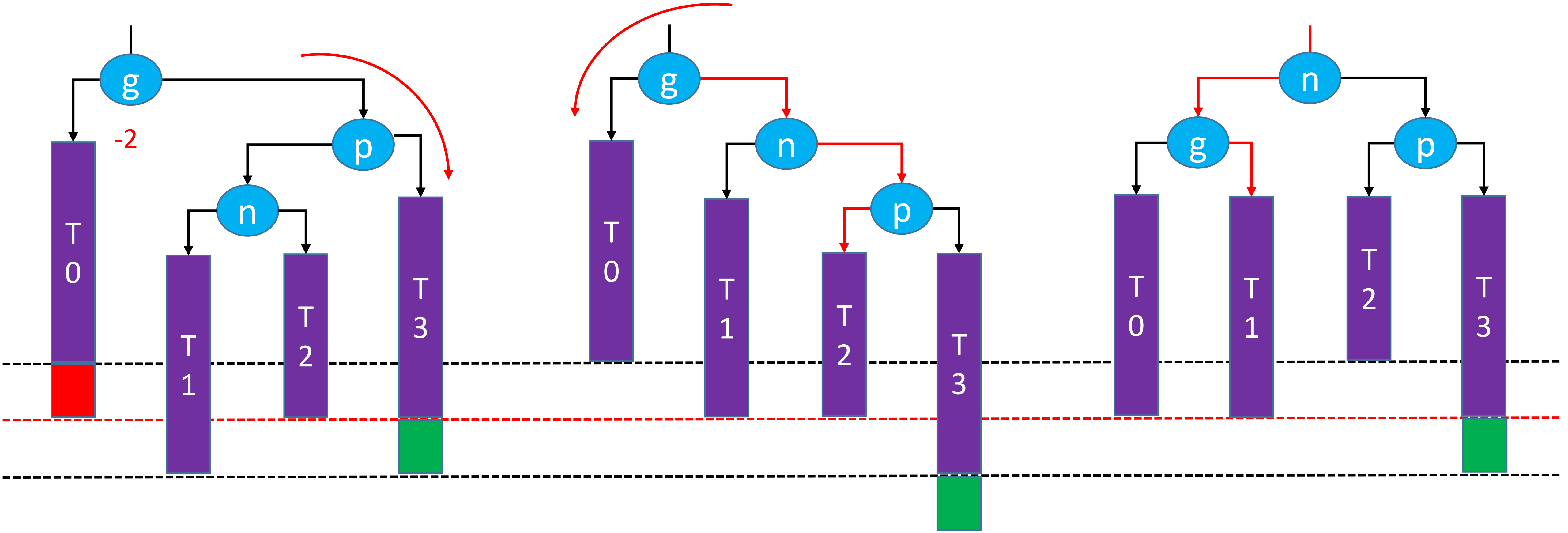
# RR – 左旋转（单旋）



# LR – RR左旋转，LL右旋转（双旋）



# RL – LL右旋转，RR左旋转（双旋）



# 删除之后的修复

```java
@Override
protected void afterRemove(Node<E> node) {
    while ((node = node.parent) != null) {
        if (isBalanced(node)) {
            updateHeight(node);
        } else {
            rebalance(node);
        }
    }
}
```

# 总结

- 添加
  - 可能会导致所有祖先节点都失衡
  - 只要让高度最低的失衡节点恢复平衡，整棵树就恢复平衡【仅需 O(1) 次调整】

- 删除
  - 可能会导致父节点或祖先节点失衡（只有1个节点会失衡）
  - 恢复平衡后，可能会导致更高层的祖先节点失衡【最多需要 O(logn) 次调整】

- 平均时间复杂度
  - 搜索：O(logn)
  - 添加：O(logn)，仅需 O(1) 次的旋转操作
  - 删除：O(logn)，最多需要 O(logn) 次的旋转操作

# 作业

- 平衡二叉树：https://leetcode-cn.com/problems/balanced-binary-tree/