HashMap、红黑树与B树

WeGene移动开发团队—邓积艺

目录

- **HashMap源码解读**
- 红黑树与B树
- 红黑树特性解析
- 红黑树的实现

HashMap源码解读（JDK1.8）

- 里面是怎么存储数据的（使用到的数据结构）
- 怎么计算哈希值，怎么解决哈希冲突
- 初始化容量是多少？不断加入数据时，如何进行扩容
- 扩容后数据的存储位置是怎么样的
- 查找数据的时间复杂度
- 为什么要用红黑树？这里的红黑树实现有什么特点

- java.lang.Object
- ↳ java.util.Map<K,V>
- ↳ java.util.AbstractMap<K,V>
- ↳ java.util.HashMap<K,V>
- public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Cloneable, Serializable{}
- 构造函数

```java
public HashMap() {
    this.loadFactor = DEFAULT_LOAD_FACTOR; // all other fields defaulted
}
public HashMap(int initialCapacity) {
    this(initialCapacity, DEFAULT_LOAD_FACTOR);
}
public HashMap(int initialCapacity, float loadFactor) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal initial capacity: " +
                                           initialCapacity);
    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;
    if (loadFactor <= 0 || Float.isNaN(loadFactor))
        throw new IllegalArgumentException("Illegal load factor: " +
                                           loadFactor);
    this.loadFactor = loadFactor;
    this.threshold = tableSizeFor(initialCapacity);
}
```

- 关键常量和成员变量

static final int DEFAULT_INITIAL_CAPACITY = 1 << 4

static final int MAXIMUM_CAPACITY = 1 << 30

static final float DEFAULT_LOAD_FACTOR = 0.75f

static final int TREEIFY_THRESHOLD = 8

static final int UNTREEIFY_THRESHOLD = 6

static final int MIN_TREEIFY_CAPACITY = 64

transient Node<K,V>[] table

transient int size

transient int modCount

int threshold

存入数据关键方法

```
static final int hash(Object key) {
    int h;
    // 如果key为null，则hash值为0，否则调用key的hashCode()方法
    // 并让高16位与整个hash异或，这样做是为了使计算出的hash更分散
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

key的hashCode方法很重要（为什么一般用String做key?）

计算key在数组中的位置index
Index = (n - 1) & hash
这就是为什么数组的容量n必须是2的幂次方。与运算(&)比模运算(%)高效多了

Hash

step1:调用key.hashCode()并复制给h

11011111 01111110 10101111 11010101 [h = key.hashCode()]

----

step2:将h的高位与低位做异或(与无符号右移的h做异或)

11011111 01111110 10101111 11010101 [h]

00000000 00000000 11011111 01111110 [h >>> 16]

00000000 00000000 01110000 10101011 ^

----

step3:将结果与n(数组长度)-1做与

00000000 00000000 01110000 10101011

00000000 00000000 00000000 00001111 &

00000000 00000000 00000000 00001011

数组中存储的数据（为什么要这么设计数据结构？）

```
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    V value;
    Node<K,V> next;
}
// 位于HashMap中
static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V> {
    TreeNode<K,V> parent;  // 红黑树节点
    TreeNode<K,V> left;
    TreeNode<K,V> right;
    TreeNode<K,V> prev;    // 用于在删除元素的时候可以快速找到它的前置节点。
    boolean red;
}
// 位于LinkedHashMap中，双向链表节点
static class Entry<K,V> extends HashMap.Node<K,V> {
    Entry<K,V> before, after;
    Entry(int hash, K key, V value, Node<K,V> next) {
        super(hash, key, value, next);
    }
}
```

存入数据关键方法

- put—>hash—>putVal—>resize—>newNode—>TreeNode.putTreeVal—>treeifyBin—>afterNodeInsertion或afterNodeAccess
- 空数组有无初始化，没有的话初始化
- 如果通过 key 的 hash 能够直接找到值，跳转到 6，否则到 3；
- 如果 hash 冲突，两种解决方案：链表 or 红黑树；
- 如果是链表，递归循环，把新元素追加到队尾；
- 如果是红黑树，调用红黑树新增的方法；
- 通过 2、4、5 将新元素追加成功，再根据 onlyIfAbsent 判断是否需要覆盖；
- 判断是否需要扩容，需要扩容进行扩容，结束。

通过Key获取数据

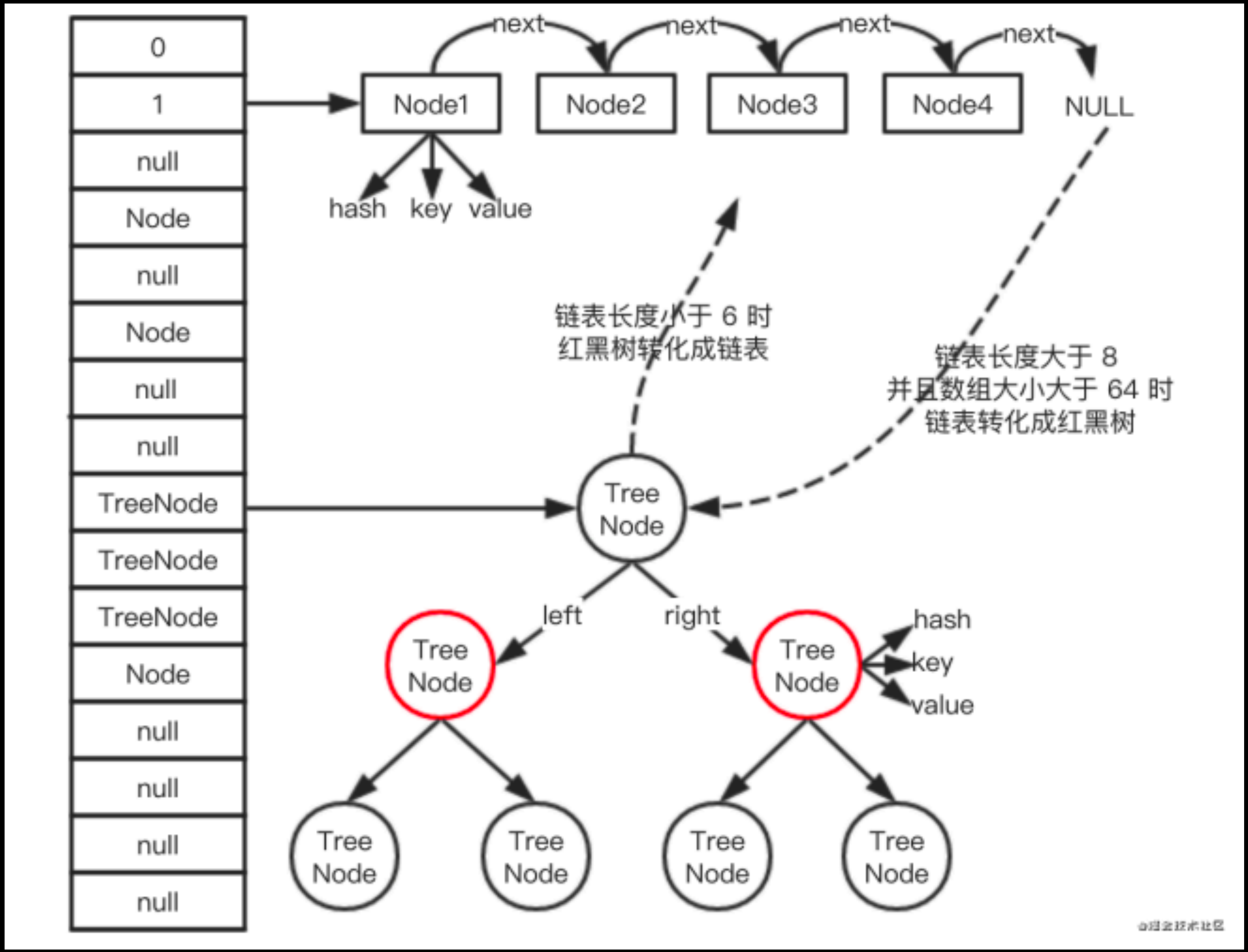
- get—>hash—>getNode—>TreeNode.getTreeNode
- 判断是否要找的数据：hash相等，key相等：(first.hash == hash && (k = first.key) == key || (key != null && key.equals(k))))
- 计算hash得到index, 从index取出数据，不为空，判断key是否相等。不等的情况看该数据是红黑树还是普通链表。链表遍历查找，红黑树查找。
- 最好的情况O(1),链表情况O(n), 红黑树情况)(logn)

删除数据

- remove—>hash—>removeNode—>TreeNode.getTreeNode—>TreeNode.removeTreeNode—>afterNodeRemoval
- 计算key的hash, 得到index, 找到节点。按照链表删除或者红黑树删除

扩容resize

- 如果使用是默认构造方法，则第一次插入元素时初始化为默认值，容量为16，扩容门槛为12（懒初始化）
- 如果使用的是非默认构造方法，则第一次插入元素时初始化容量等于扩容门槛，扩容门槛在构造方法里等于传入容量向上最近的2的n次方
- 如果旧容量大于0，则新容量等于旧容量的2倍，但不超过最大容量2的30次方，新扩容门槛为旧扩容门槛的2倍
- 创建一个新容量的数组
- 搬移元素，普通节点直接搬移。原链表分化成两个链表，低位链表存储在原来桶的位置，高位链表搬移到原来桶的位置加旧容量的位置。红黑树会被拆分



此图关于红黑树的数据结构并不完全正确

目录

红黑树的由来

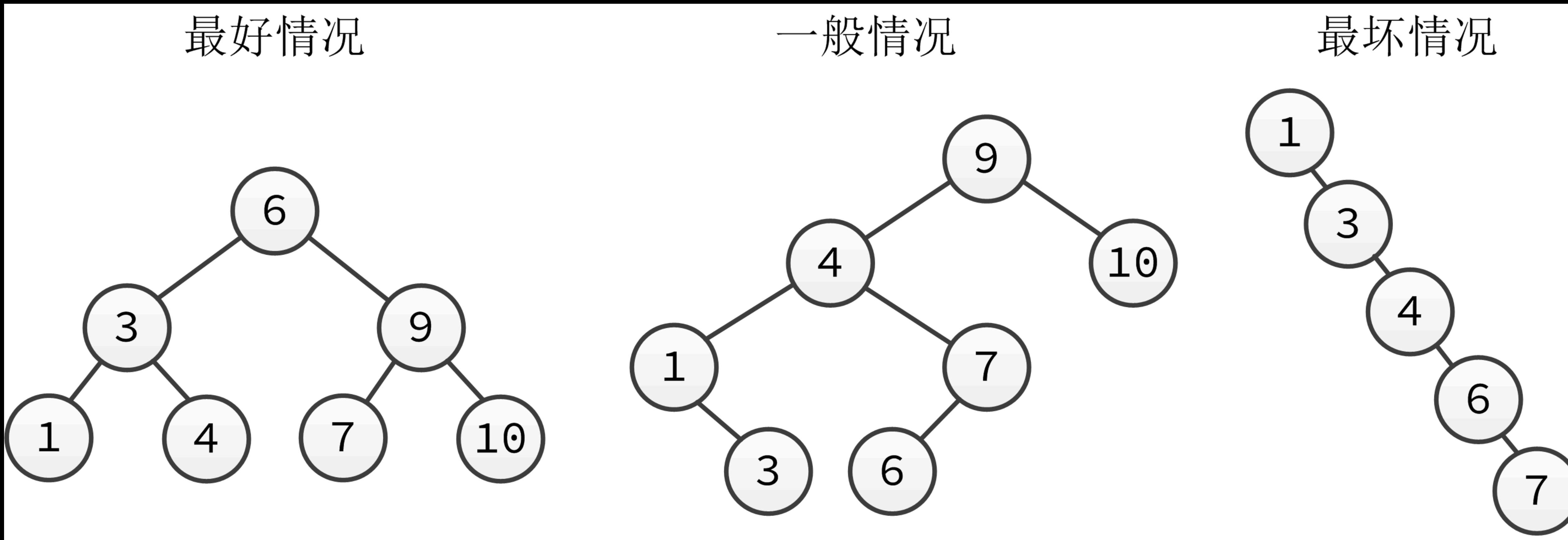
- 树》二叉树》二叉排序树》自平衡二叉树》2-3树和2-3-4树（B树）》红黑树

二叉排序树

- 二叉排序树（Binary Sort Tree），又称二叉查找树（Binary Search Tree），又称 二叉搜索树。

一棵空树，或者是具有下列性质的二叉树：

- （1）若左子树不空，则左子树上所有结点的值均小于它的根结点的值；
- （2）若右子树不空，则右子树上所有结点的值均大于它的根结点的值；
- （3）左、右子树也分别为二叉排序树；
- （4）没有键值相等的结点。



二叉查找树可能形状

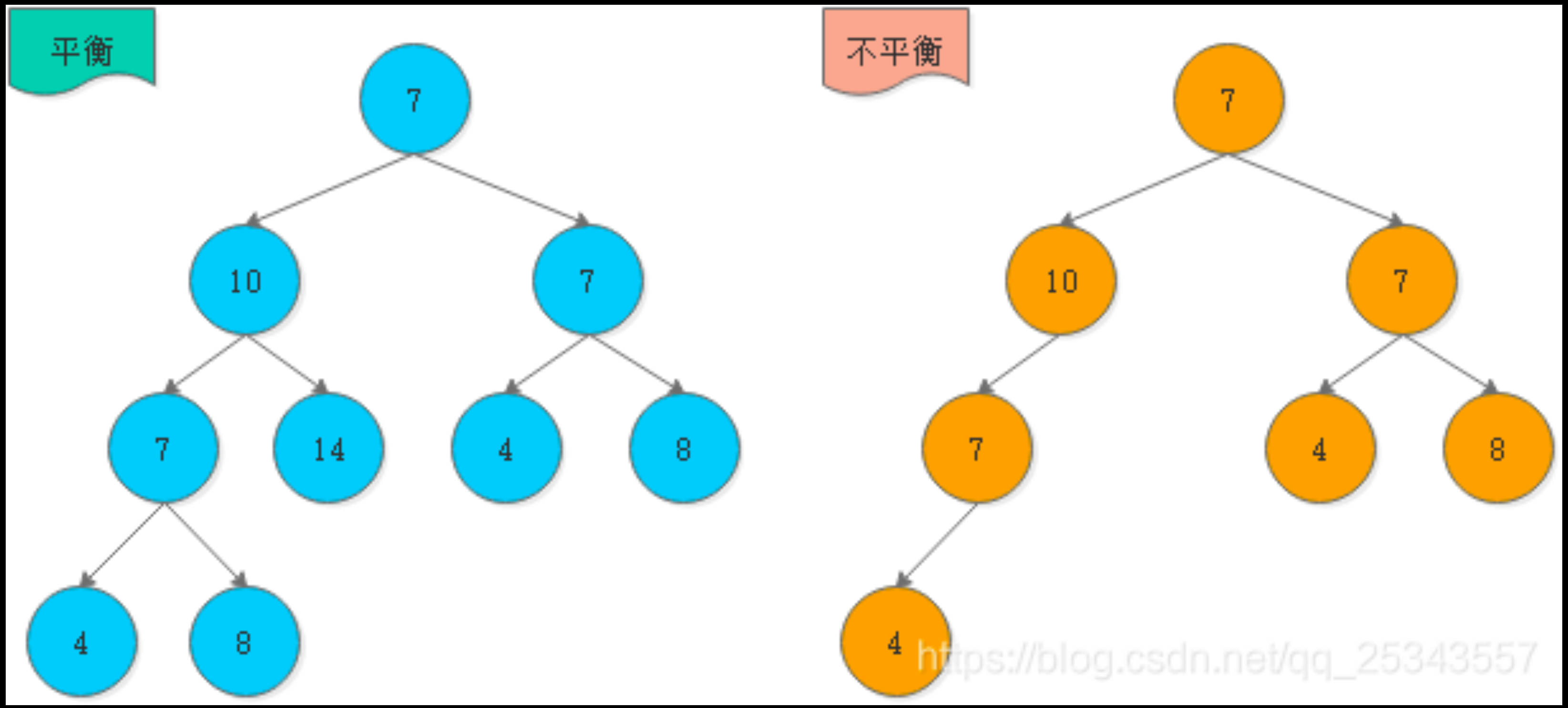
最好的情况下，查找效率O(logn)

最差的情况下，查找效率O(n)

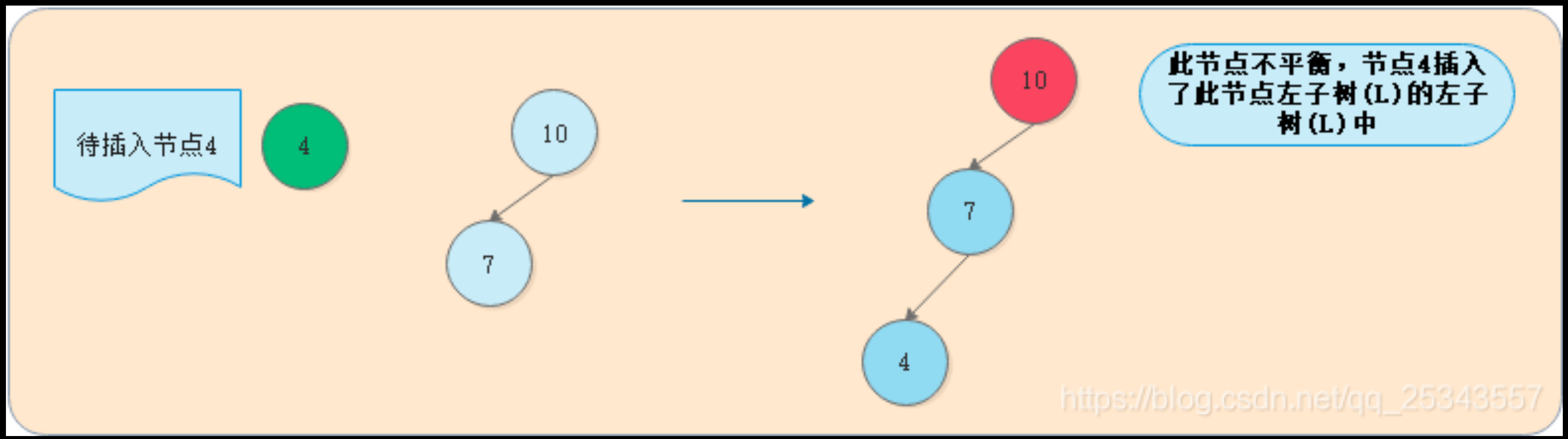
一般情况下，二叉排序树查询效率比链表结构要高

平衡二叉树

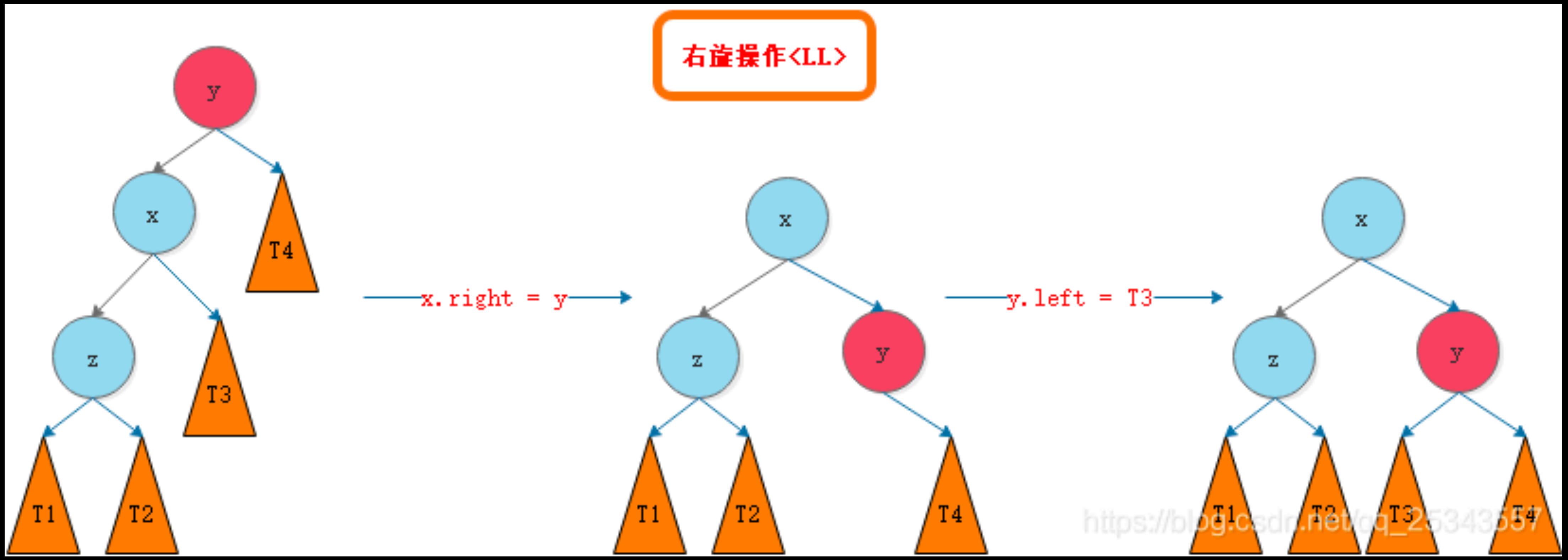
- 平衡二叉树，又叫AVL树。AVL是它的发明者的名字缩写（G. M. Adelson-Velsky和E. M. Landis）
- 首先是一棵二叉搜索树
- 每个结点的左右子树的高度之差的绝对值（平衡因子）最多为1
- 为了能让树保持平衡，插入或删除节点后经常要对树进行左旋或者右旋操作


平衡
7
10
7
7
14
4
8
4
8
不平衡
7
10
7
7
4
8
4

右旋

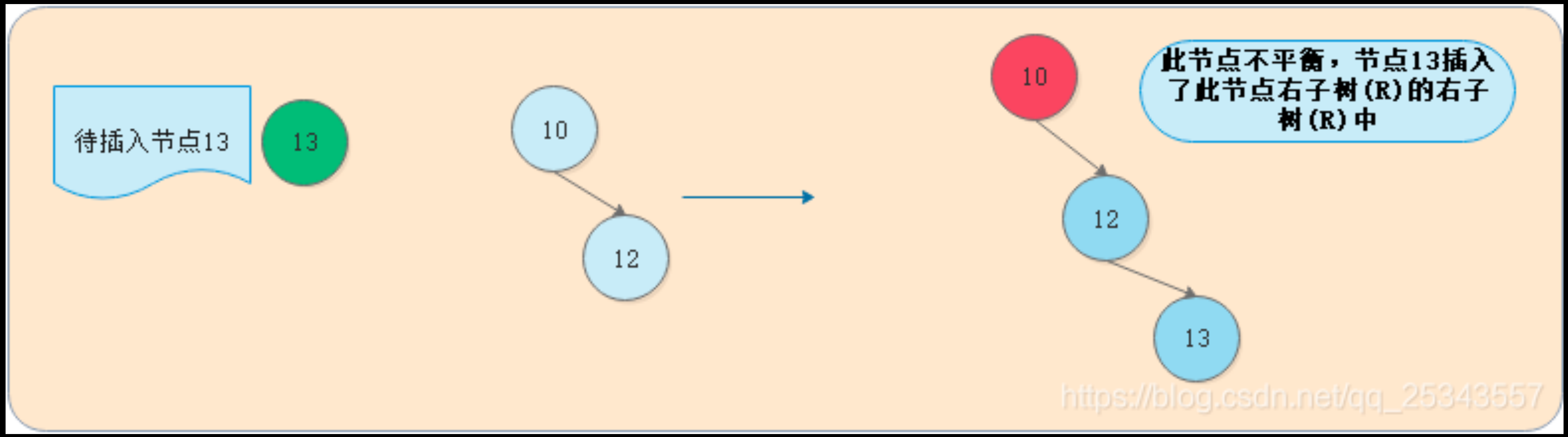


待插入节点4
4
10
7
10
7
4
此节点不平衡，节点4插入了此节点左子树(L)的左子树(L)中
https://blog.csdn.net/qq_25343557

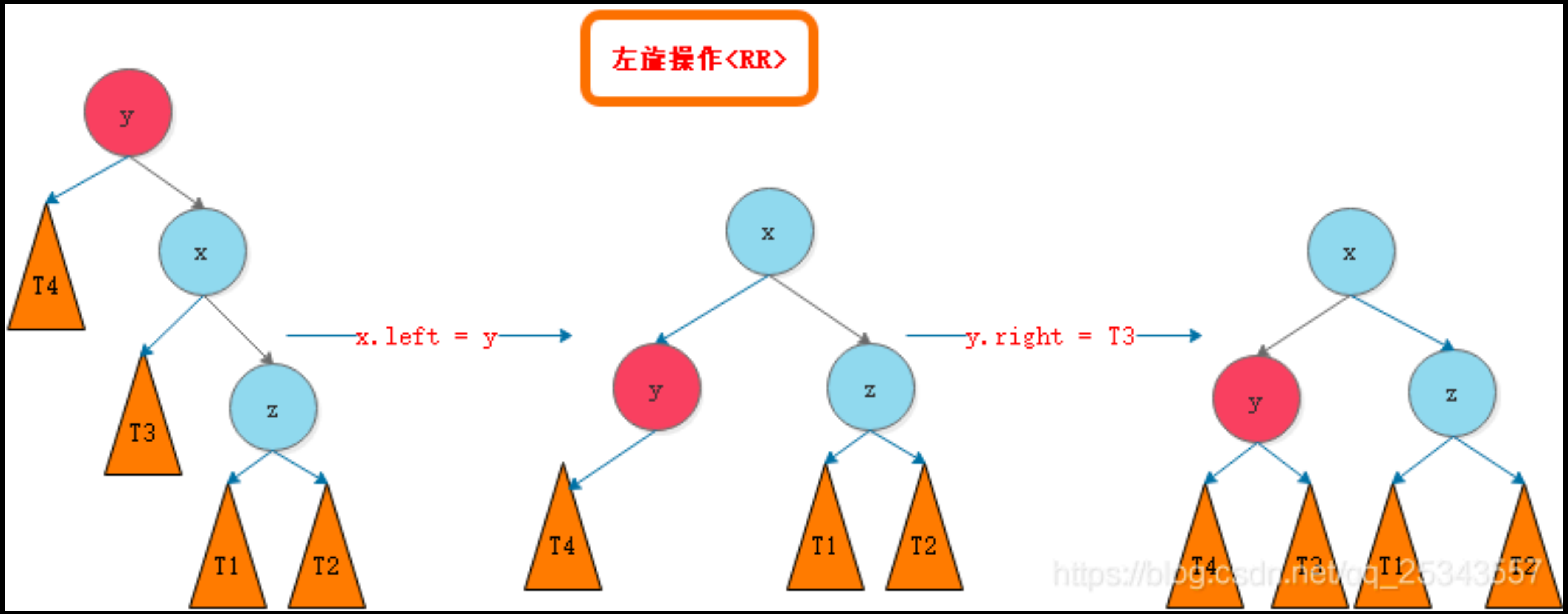


右旋操作<LL>
y
x
T4
z
T3
T1
T2
x.right = y
x
z
y
T1
T2
T4
y.left = T3
x
z
y
T1
T2
T3
T4
https://blog.csdn.net/qq_25343557

左旋



待插入节点13
13
10
12
10
12
13
此节点不平衡，节点13插入了此节点右子树(R)的右子树(R)中
https://blog.csdn.net/qq_25343557



左旋操作<RR>
y
x
z
T4
T3
T1
T2
x.left = y
y.right = T3
https://blog.csdn.net/qq_25343557

B树

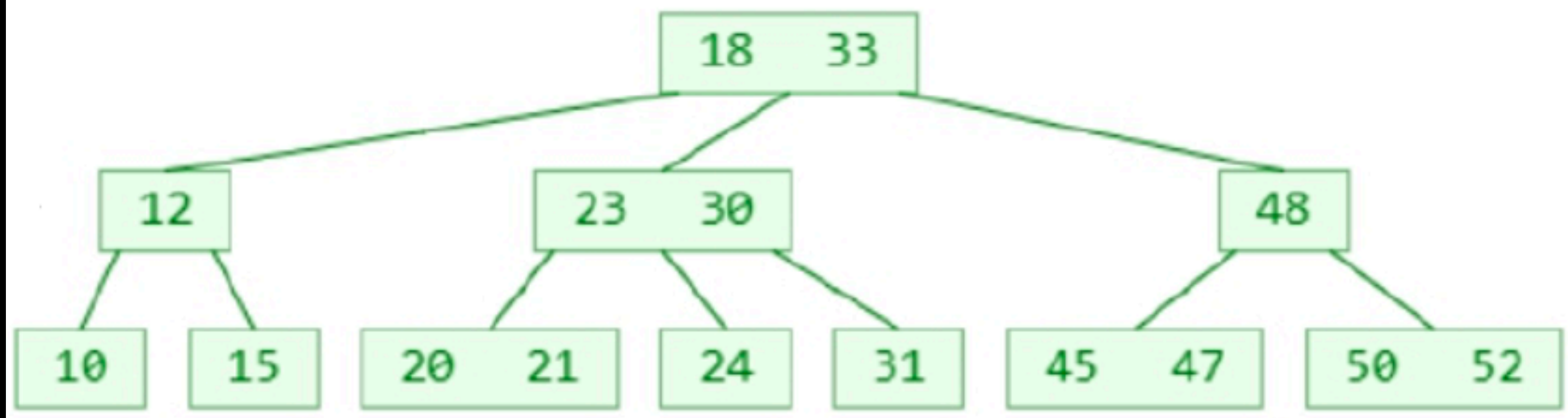
- B树是一种平衡的多路搜索树

- 1 个节点可以存储超过 2 个元素（多个元素从小到大排列）、可以拥有超过 2 个子节点。一个节点存储的元素个数是它的儿子个数减1

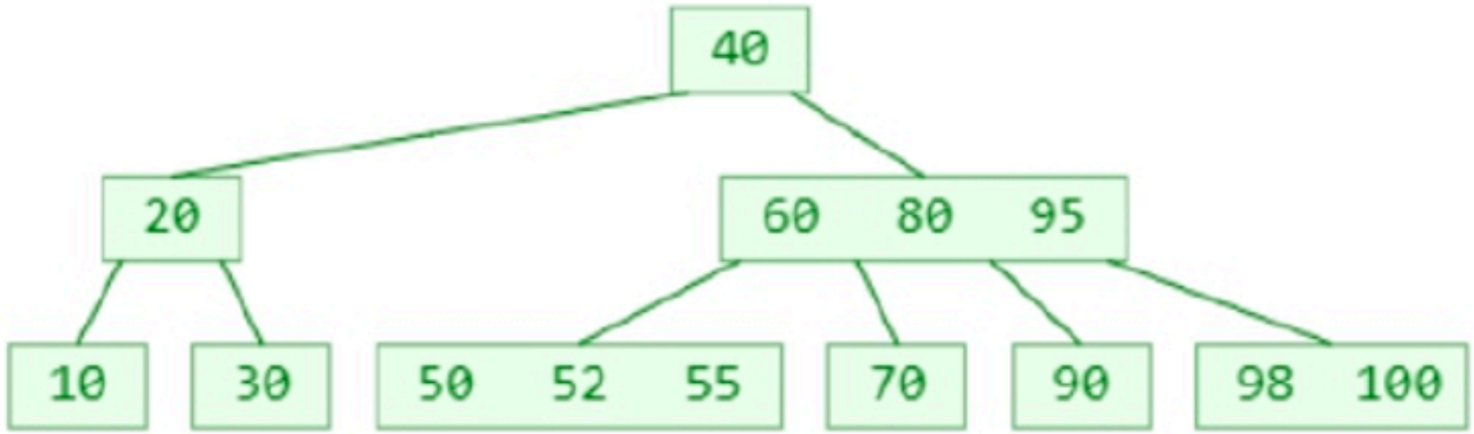
- 拥有二叉搜索树的一些性质

- 绝对平衡：每个节点的所有子树高度一致

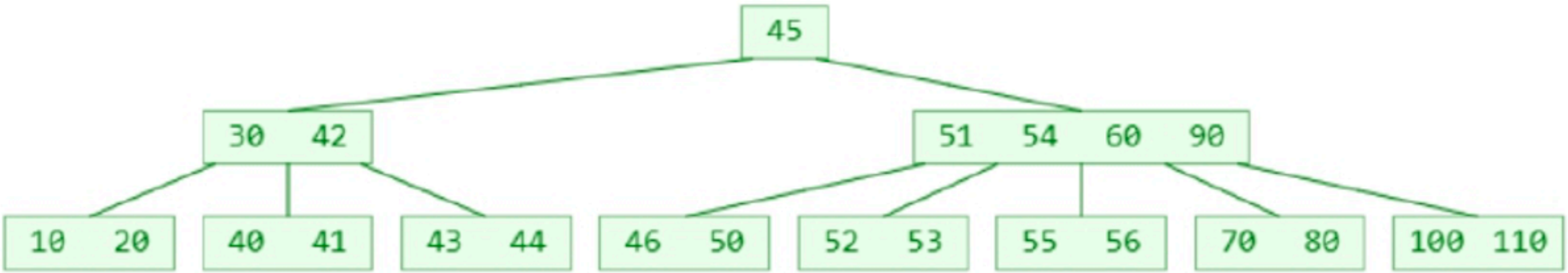
- 2-3树和2-3-4树是B树的特例，2-3树是最简单的B树



3阶B树

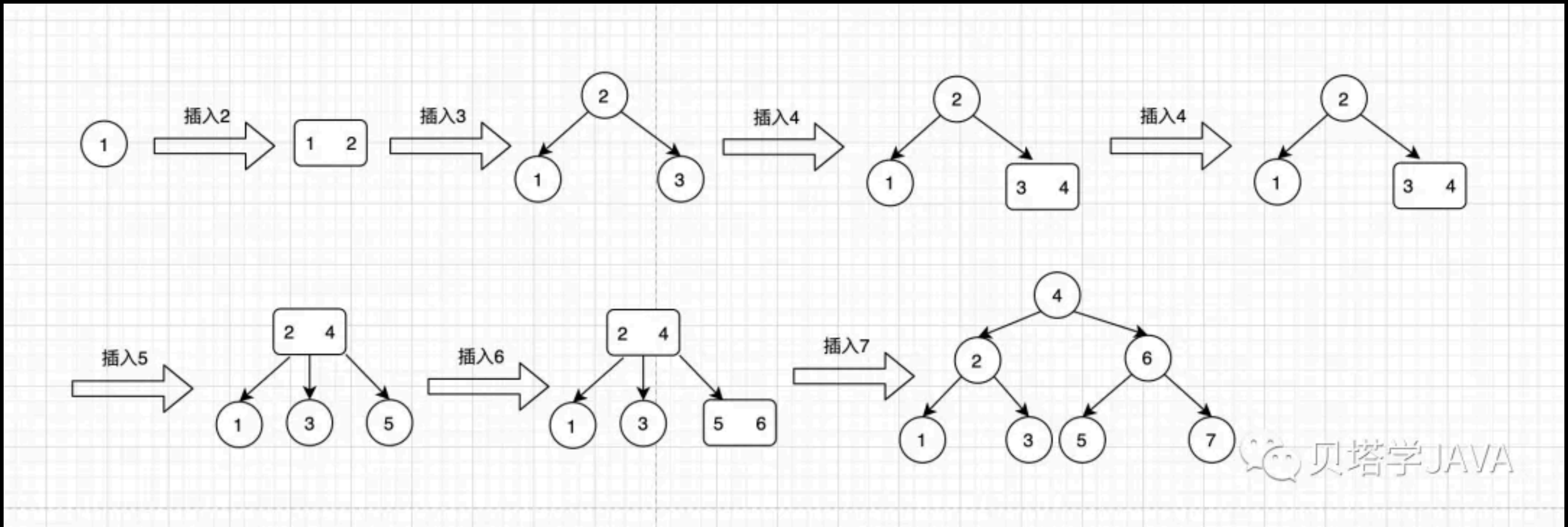


4阶B树



5阶B树

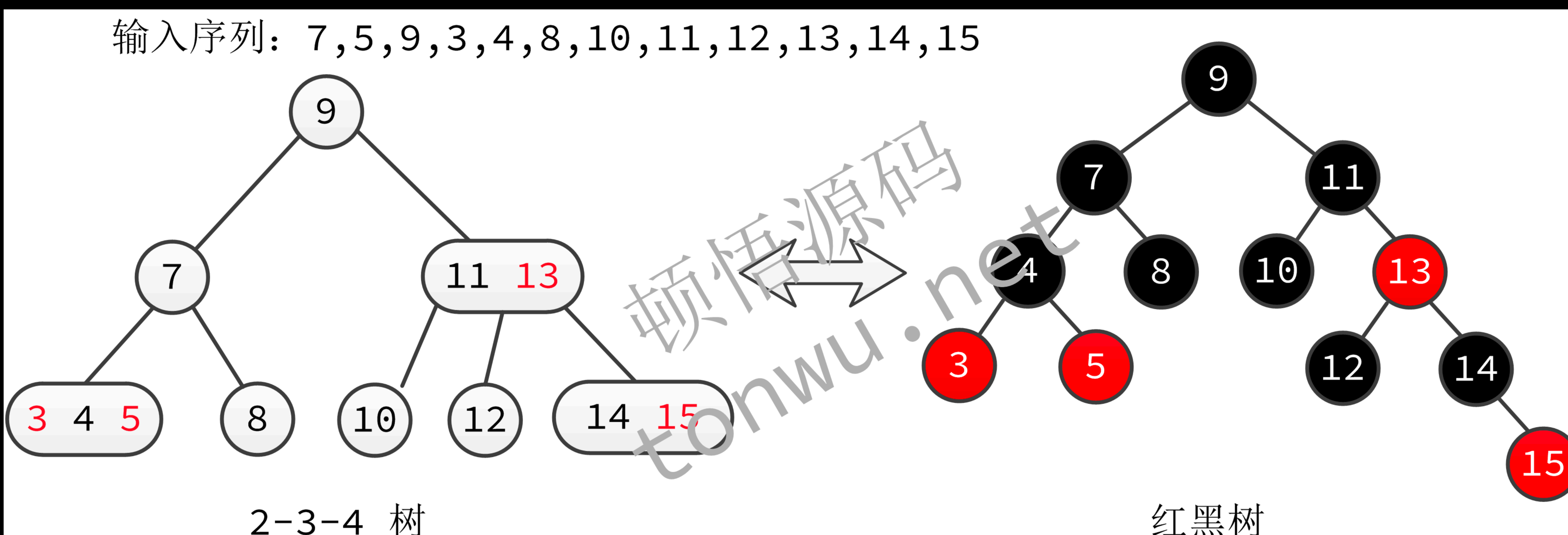
2-3树的插入



- 2-3树的插入会导致树向上生长，并保持绝对平衡
- 2-3树的删除节点会导致节点合并，让树保持绝对平衡

2-3树、2-3-4树与红黑树

- 2-3树、2-3-4树虽然能保持平衡，但是计算机不好实现
- 红黑树（Red Black Tree） 是一种自平衡二叉查找树。红黑树是近似平衡的二叉树，左右子树高差有可能大于 1，但是他的平均性能要好于AVL树
- 红黑树对应的理论模型可以是2-3树，也可以是2-3-4树。普遍红黑树的实现是采用2-3-4树模型



目录

- HashMap源码解读
- 红黑树与B树
- **红黑树特性解析**
- 红黑树的实现

红黑树5个特性（Why）

- 1） 节点是红色或黑色
- 2） 根节点是黑色
- 3） 空节点（NIL节点，有些文章也叫叶子节点）是黑色的
- 4） 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)
- 5 ）从任一节点到其每个叶子的路径上包含的黑色节点数量都相同
- 附加特性：插入的节点设为红色

节点插入与删除涉及操作

- 为了保持红黑树的平衡（即符合红黑树的5个特性），节点插入或删除需要做一些操作：

- 变色

- 左旋转或右旋转，或者将两种旋转组合

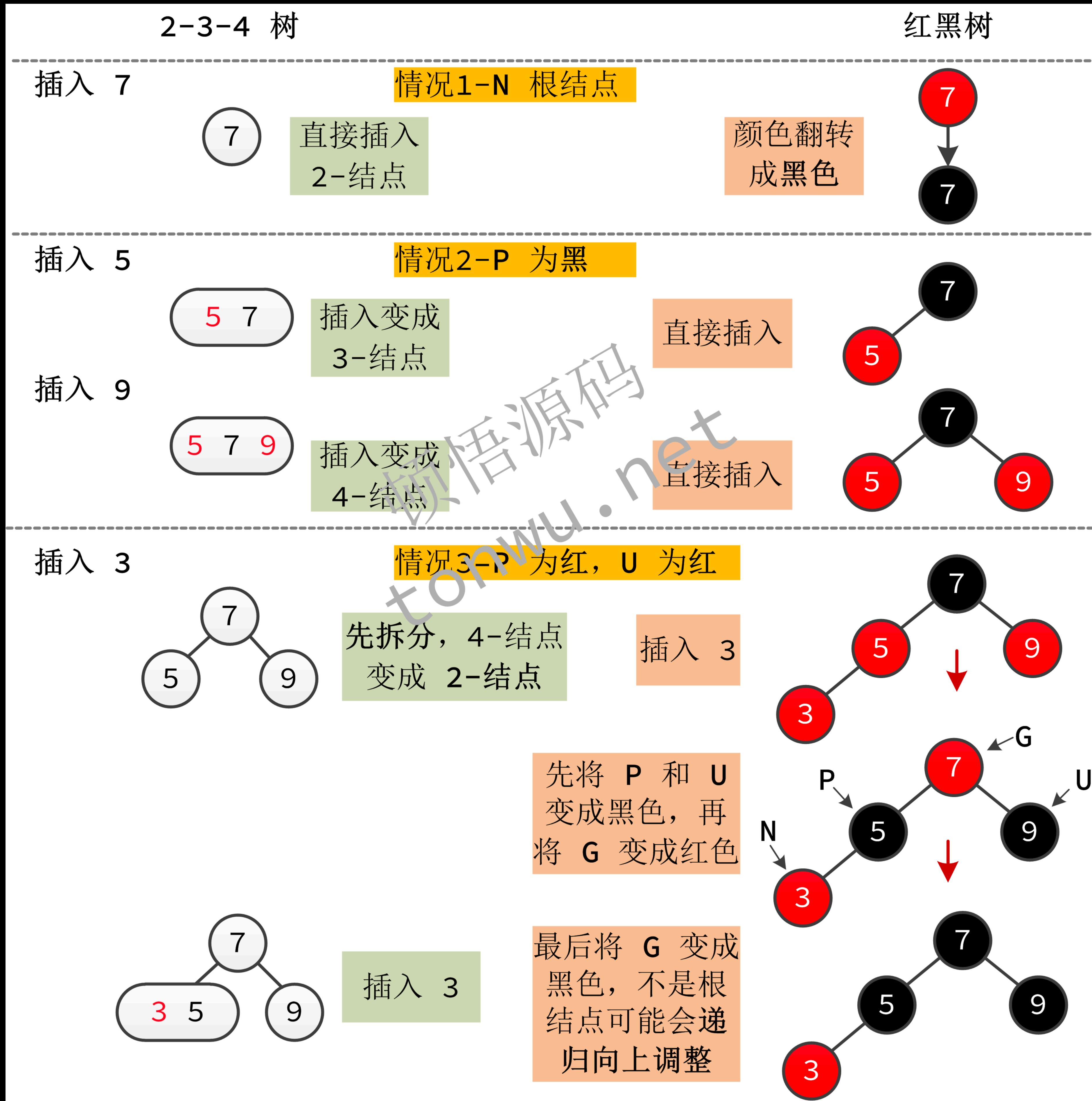
- 涉及到的节点：当前插入的节点，当前节点的父节点，爷爷节点，叔叔节点

插入节点

- 先查找到位置，插入，再进行相关操作

这里假设待插入结点为 **N(node)**，**P** (parent)是 **N** 的父结点，**G** (grandparent)是 **N** 的祖父结点，**U** (uncle)是 **N** 的叔叔结点（即父结点的兄弟结点），那么红黑树有以下几种插入情况：

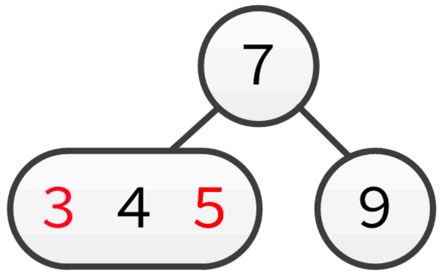
- (1)N 是根结点，即红黑树的第一个结点
- (2)N 的父结点（P）为黑色
- (3)P 是红色的（不是根结点），它的兄弟结点 U 也是红色的
- (4)P 为红色，而 U 为黑色(注意空节点是黑色的)


2-3-4 树
红黑树
插入 7
情况1-N 根结点
直接插入
2-结点
颜色翻转
成黑色
插入 5
情况2-P 为黑
插入变成
3-结点
直接插入
插入 9
插入变成
4-结点
直接插入
插入 3
情况3-P 为红，U 为红
先拆分，4-结点
变成 2-结点
插入 3
先将 P 和 U
变成黑色，再
将 G 变成红色
G
P
U
N
插入 3
最后将 G 变成
黑色，不是根
结点可能会递
归向上调整

- 情况1，直接插入，根据规则2，将节点变黑即可
- 情况 2，不影响红黑树的性质，不会打破平衡，直接插入即可
- 情况 3，P 和 U 变成黑色，G 变成红色（即将他们的颜色翻转）。若 G 是根结点，直接变黑，否则递归向上检查是否造成不平衡

2-3-4 树 红黑树

插入 4



情况4-P 为红，U 为黑

直接插入变成 4-结点

U 为叶子节点黑色

P 是 G 的左孩子，N 是 P 的右孩子，那么 P 先左旋转，然后再将祖父结点 G 右旋转

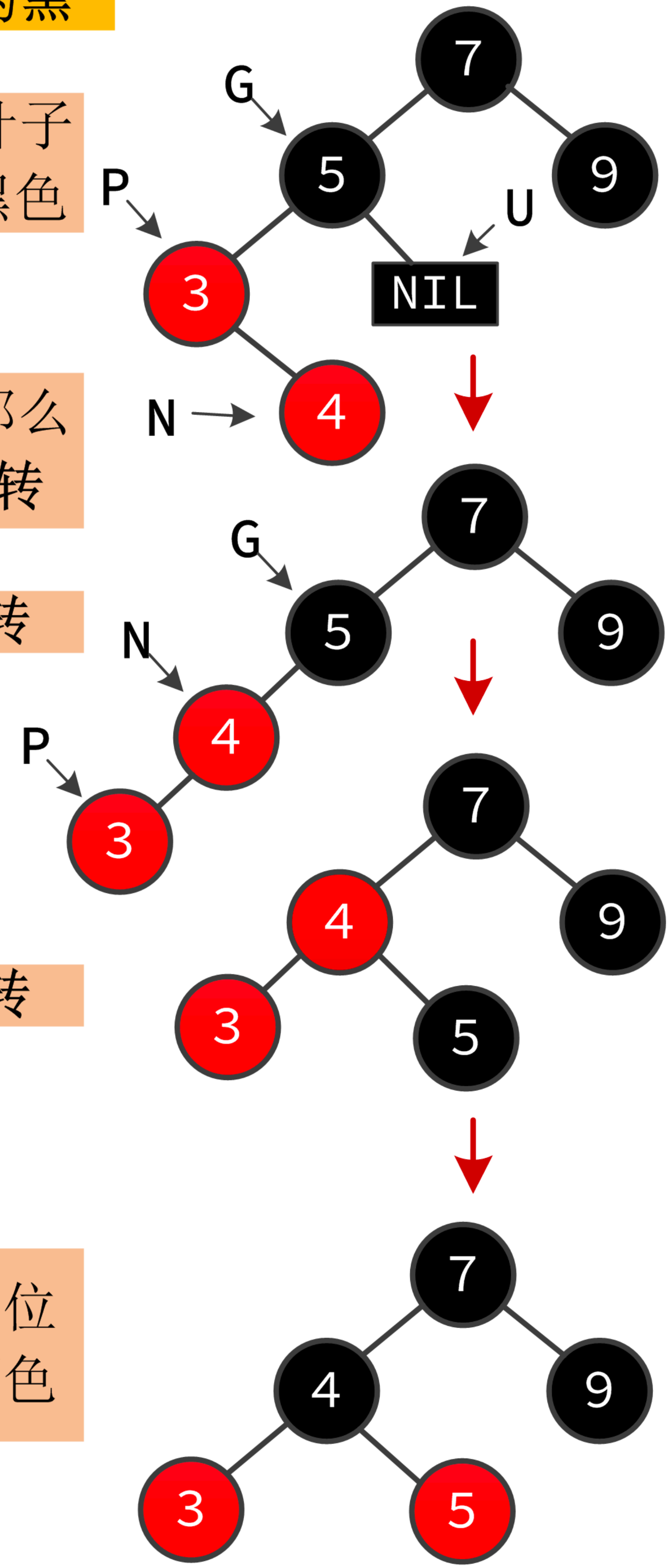
如果 N 是左孩子，直接将 G 右旋转即可

P 先左旋转

对应情况的旋转都是固定的，因为最终都会像 2-4 树那样变成一个 4-结点 的结构

G 再右旋转

N 和 G 调整位置，并互换颜色



情况 4，P 为红色，而 U 为黑色

- P 是 G 的左孩子，若 N 是 P 的左孩子，那么将祖父结点 G 右旋转 即可（/)
- P 是 G 的左孩子，若 N 是 P 的右孩子，那么 P 先左旋转，然后再将祖父结点 G 右旋转 (< To /)
- P 是 G 的右孩子，若 N 是 P 的右孩子，那么将祖父结点 G 左旋转 即可 (\)
- P 是 G 的右孩子，若 N 是 P 的左孩子，那么 P 先右旋转，然后再将祖父结点 G 左旋转(> to \)

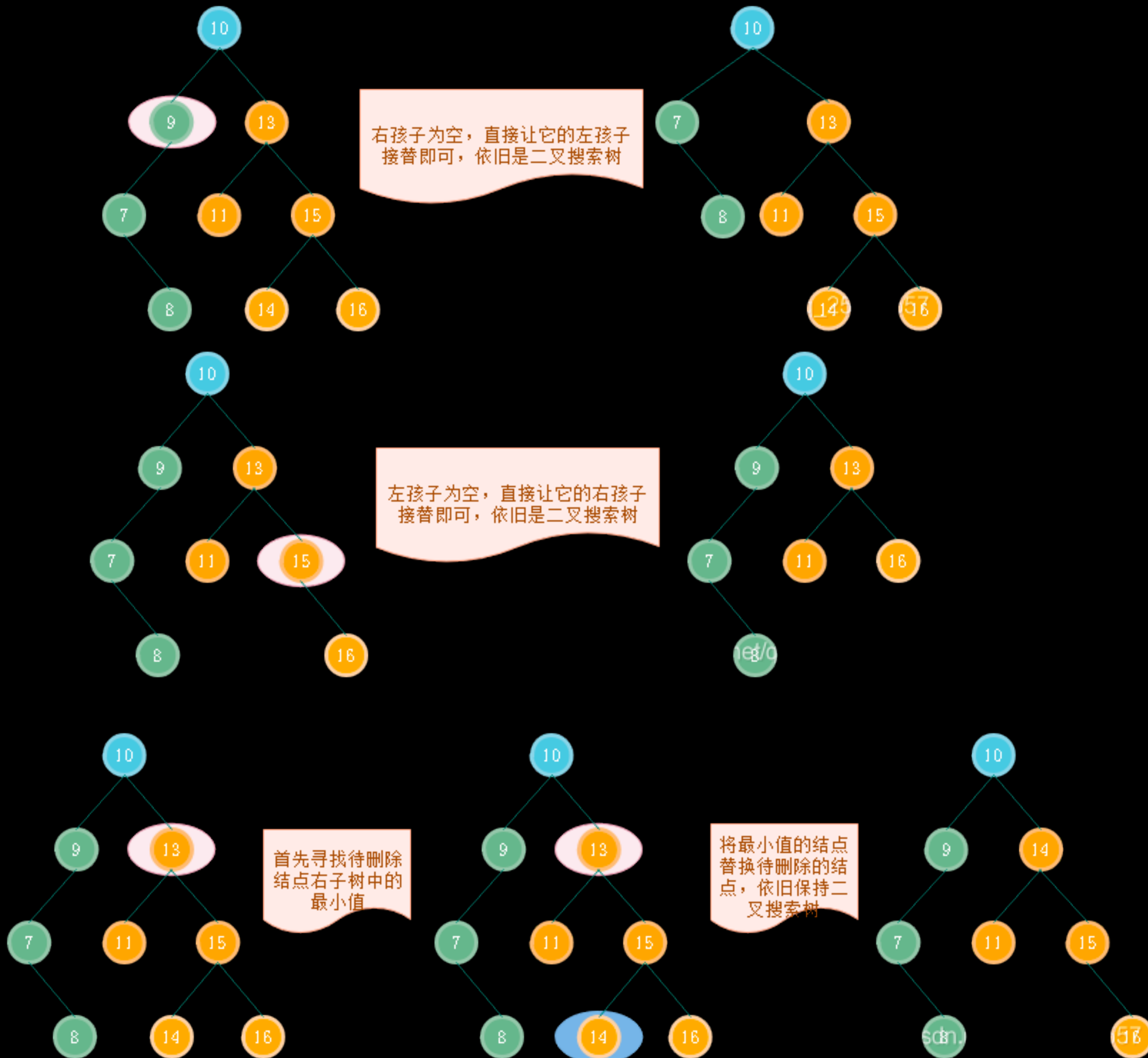
插入总结

- 先看父，再看兄，然后看爷爷。若爷孙三代不在直线，先父转，再变色, 再爷转

二叉查找树删除节点

- 二叉查找树的结点无非是有两个子结点，有一个子结点和叶子结点三种，其中有两个子结点的 M 结点的删除逻辑是：
- 首先寻找 M 结点左子树最大或右子树最小的结点 X
- 然后把 X 结点的值复制到 M 结点
- 最后删除 X 结点，而这个结点要么是叶子结点，要么就只有一个孩子

所以，删除任一结点的问题就简化成了：删除一个最多只有一个孩子的结点的情况（要么没有孩子，要么只有一个孩子）


右孩子为空，直接让它的左孩子接替即可，依旧是二叉搜索树
左孩子为空，直接让它的右孩子接替即可，依旧是二叉搜索树
首先寻找待删除结点右子树中的最小值
将最小值的结点替换待删除的结点，依旧保持二叉搜索树

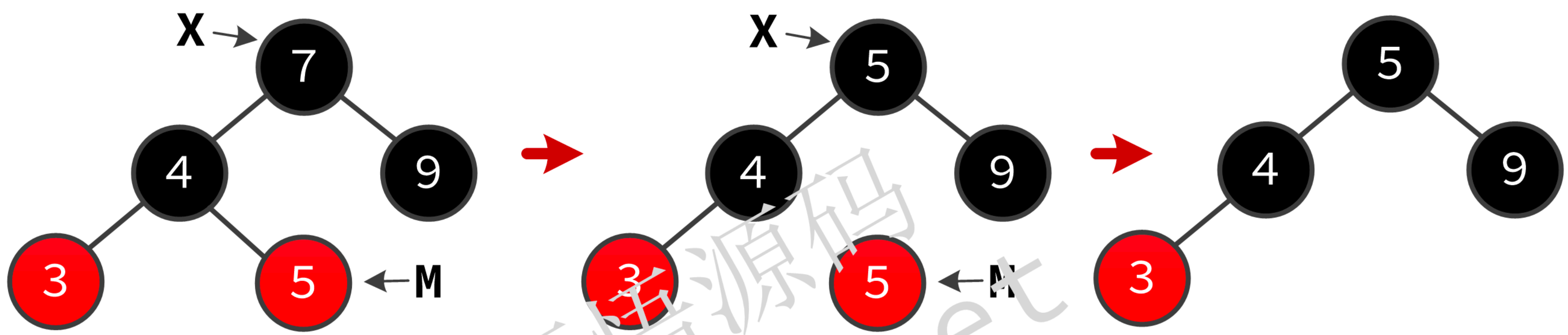
红黑树删除节点要领

- 删除红色叶子节点不影响平衡
- 删除黑色节点会影响树的平衡，所以想办法从孩子节点，或者兄弟节点，或者父节点借一个红色节点过来，并把它变黑，这样树就恢复平衡了。
- 如果没办法借到红色节点，只能将平衡交给父节点处理

删除结点 **7**

M 是一个红色结点

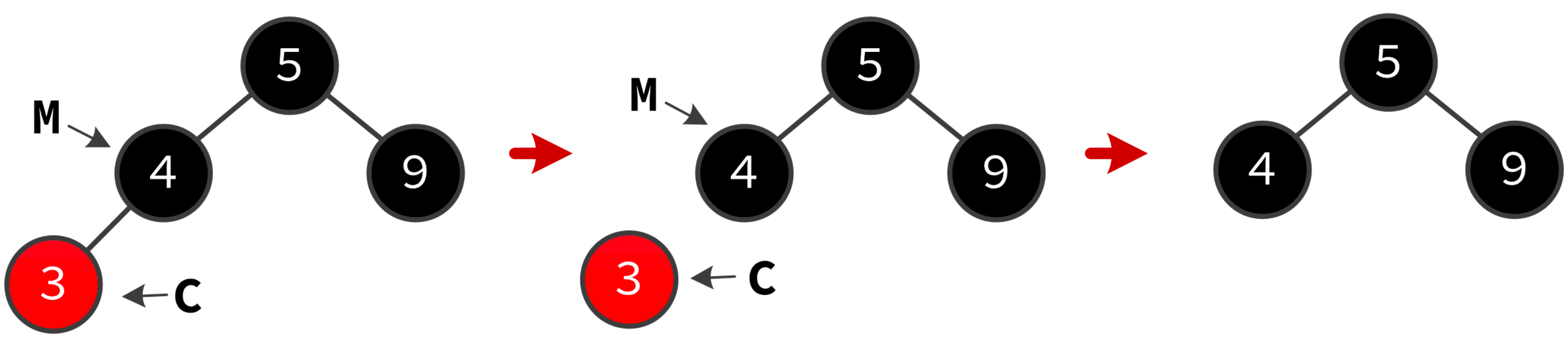
首先找到要删除的结点 **X**，然后查找到左子树值最大的结点 **M** → 将 **M** 的值复制给 **X**，最后删除 **M** 结点 → 实际删除的其实是一个红色叶子结点



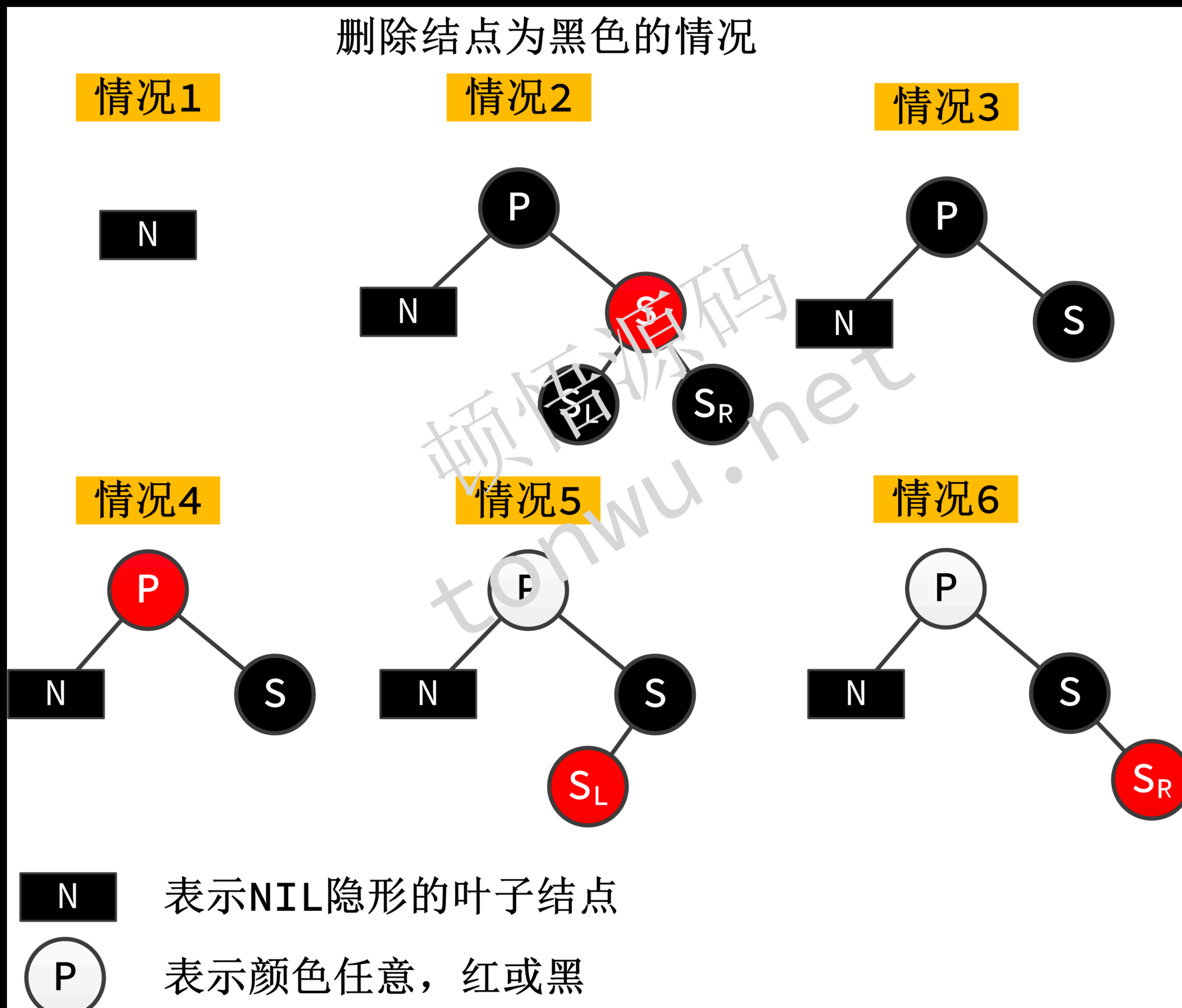
删除结点 **4**

M 是黑色而 **C** 是红色(左或右孩子)

首先找到要删除的结点 **M**，然后查找左子树值最大的结点 **C** → 将 **C** 的值复制给 **M**，最后删除 **C** 结点 → 实际删除的其实是一个红色结点



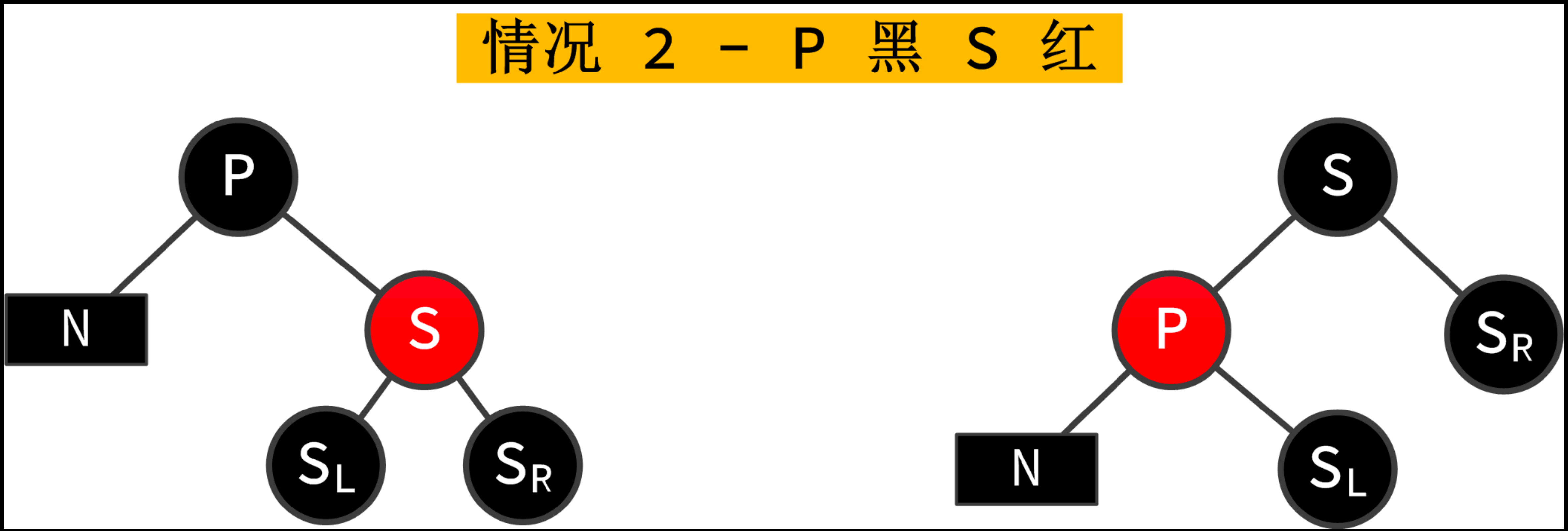
待删除结点为 M，**且为黑色，且为叶子节点（没有非空子节点）。一个黑色节点被删除，平衡被打破，**需要找一个结点填补这个空缺。M被删除后，它的位置上就变成了 **NIL** 结点，为了方便描述，这个结点记为 **N**，**P** 表示 **N** 的父结点，**S** 表示 **N** 兄弟结点，**S** 如果存在左右孩子，分别使用 **SL** 和 **SR** 表示，那么删除就有以下几种情况：



情况 2 - P 黑 S 红

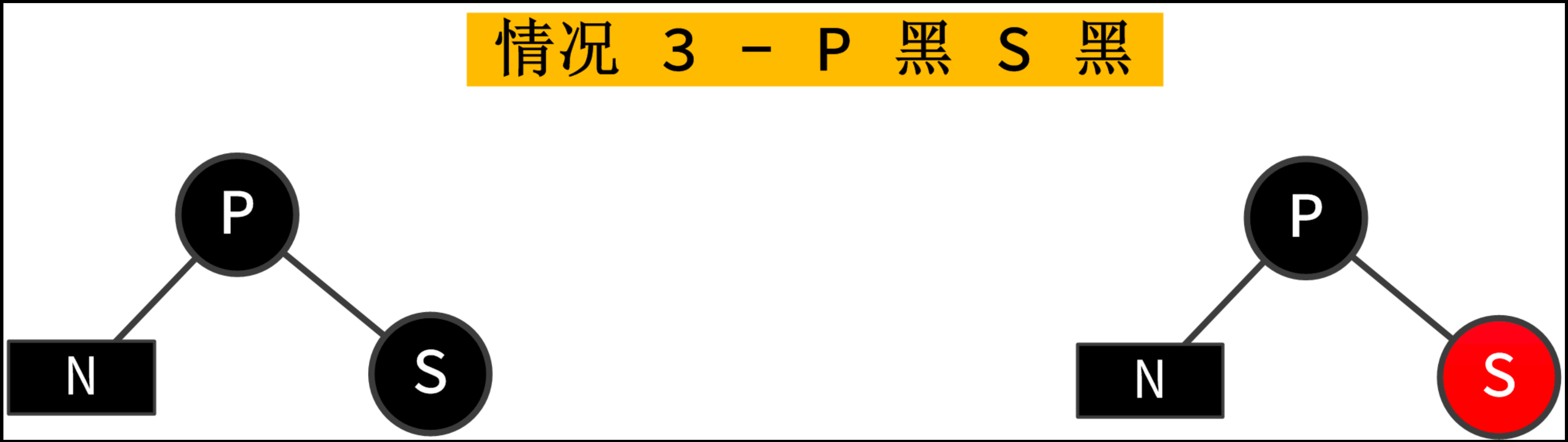
- P 黑 S 红。**S** 是红色，那么它必有两个孩子结点，且都为黑色，而且 **P** 也肯定是黑色。此时，交换 **P** 和 **S** 的颜色，然后对 **P** 左旋转

- 结点 **N** 的父结点变成了红色，兄弟结点变成了 **SL**，此时就可以按照**情况 4**



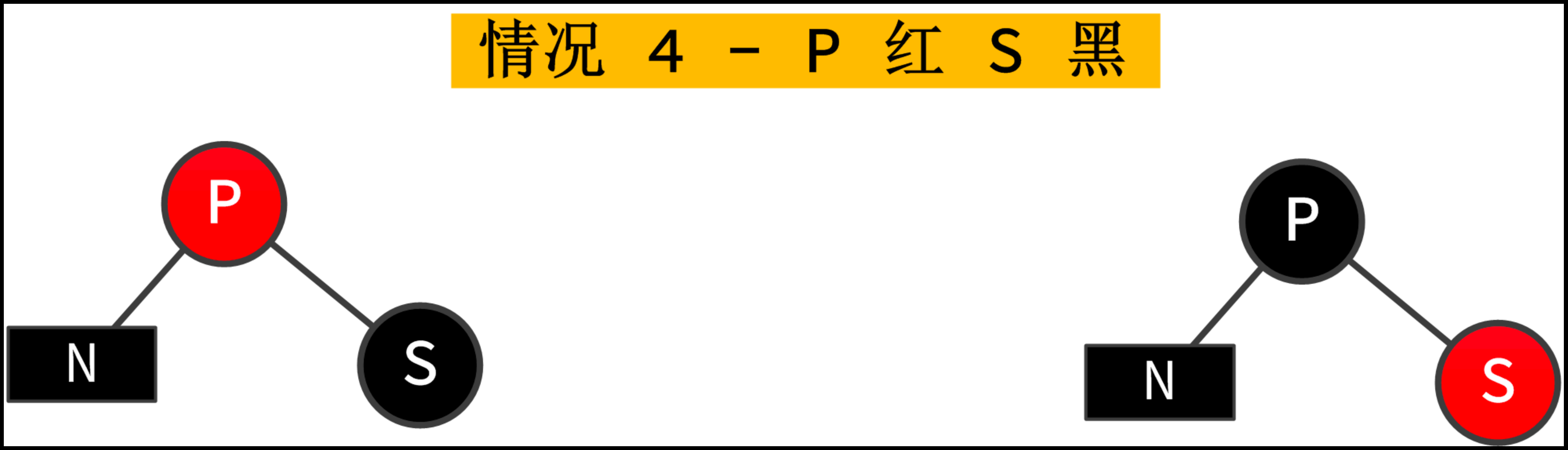
情况 3 - P 黑 S 黑

- P 是黑色，S 也是黑色，并且 S 也没有非空的孩子结点
- 直接将 S 变成**红色**，那么经过 S 的路径也就少了一个黑色结点，不平衡状态从结点 N 转移到了结点 P。**把P当做好平衡的节点，向上做平衡**



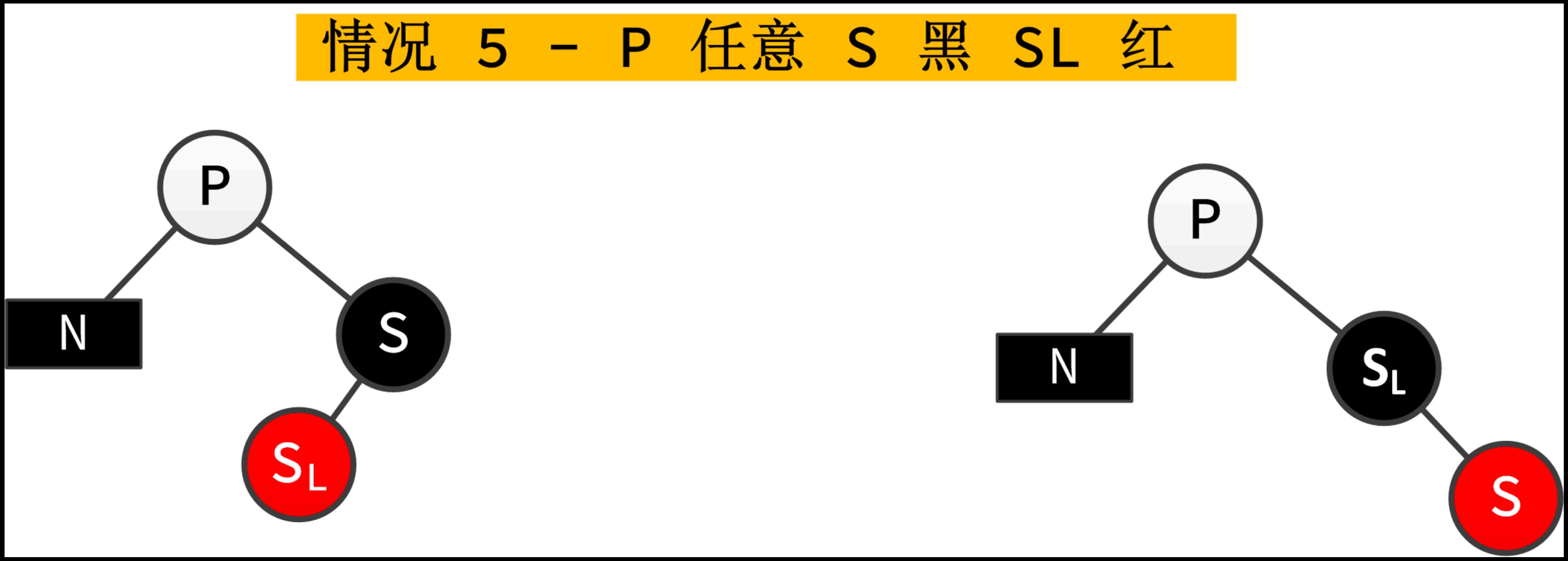
情况 4 - P 红 S 黑

- **P** 是红色，**S** 是黑色，并且 **S** 也没有非空的孩子结点
- 交换 **P** 和 **S** 的颜色，正好填补了少一个黑色结点的空缺，也就是恢复了平衡的状态



情况 5 - P 任意 S 黑 SL 红

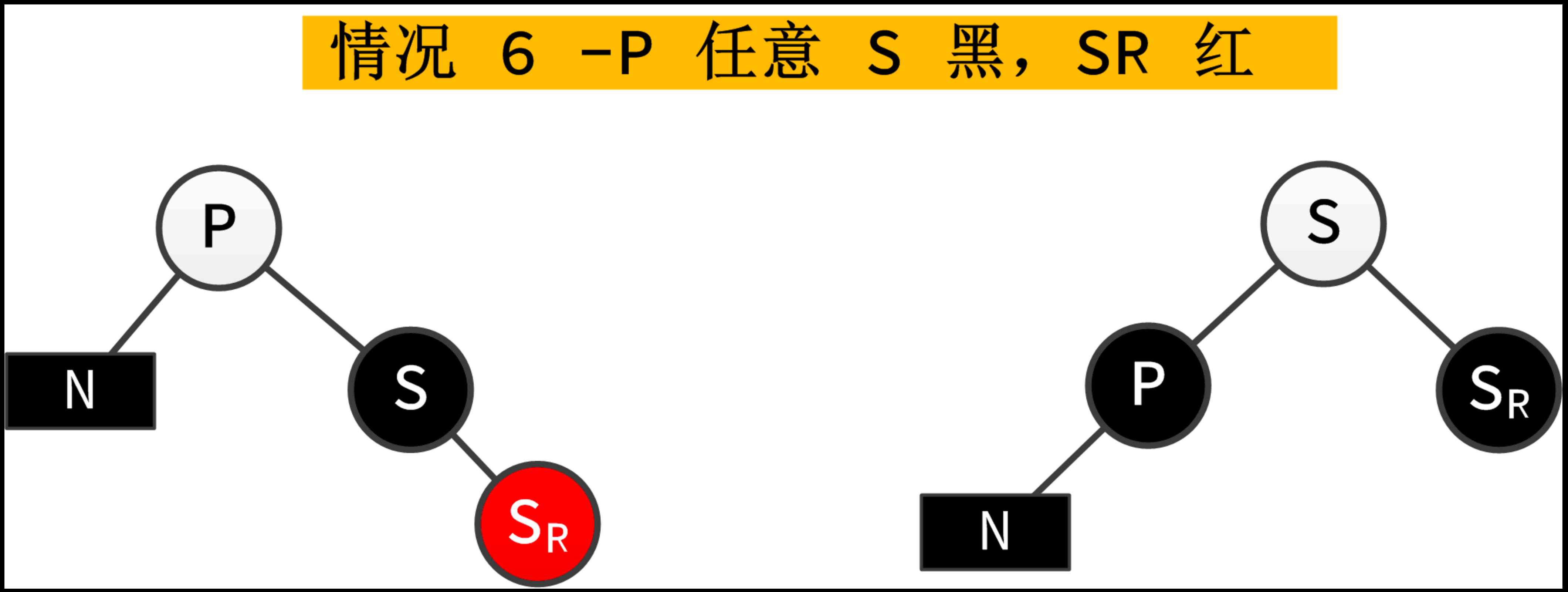
- **P** 任意颜色，**S** 黑色，**S** 的左孩子红色，S只有左孩子
- 对 **S** 右旋转，并交换 **S** 和 **SL** 的颜色，转成了 **情况 6** 进行处理



情况 6 - P 任意 S 黑，SR 红

- **P** 任意颜色，**S** 黑色，**S** 的右孩子红色，（**S** 有左孩子也是**必然是红色**，并且不影响处理）

- 对 **P** 左旋转，交换 **P** 和 **S** 的颜色，并将 **SR** 变成黑色。此时已恢复平衡的状态



目录

- HashMap源码解读
- 红黑树与B树
- 红黑树特性解析
- **红黑树的实现**

红黑树的实现

- HashMap中的红黑树特性：代码复杂，成员变量多，包含双向链表结构，空间冗余

- TreeMap：按照2-3-4树模型实现，代码可读性强

- 重复造轮子：手动实现一个红黑树

终极疑问

- 红黑树最差情况是怎样的?
- 基于2-3树和基于2-3-4树模型实现的红黑树有什么区别?
- HashMap里的红黑树什么搞那么复杂? （包含双向链表，空间冗余）

参考资料

- https://juejin.cn/post/6959100025423003684 全网最硬核的源码分析之——HashMap源码分析
- https://juejin.cn/post/6933491739651112967 HashMap源码分析
- https://juejin.cn/post/6844904154385629198 漫画：什么是红黑树?
- https://www.cnblogs.com/yinbiao/p/10732600.html 目前最详细的红黑树原理分析（大量图片+过程推导！！！）
- **https://www.cnblogs.com/chuonye/p/11236136.html 红黑树这个数据结构，让你又爱又恨？看了这篇，妥妥的征服它**
- https://blog.csdn.net/qq_25343557/article/details/89110319 详细图文——AVL树 （递归法）
- https://juejin.cn/post/6956589890062516237 B树
- https://blog.csdn.net/m0_46864744/article/details/113924234 红黑树与TreeMap详细解析
- https://mp.weixin.qq.com/s/HdSuV8jzMwKvxfaiTPUMPQ 硬核图解红黑树并手写实现
- https://www.cs.usfca.edu/~galles/visualization/RedBlack.html 红黑树插入删除节点可视化

对网上参考资料吐槽

- 没有把红黑树规则3说清楚（说叶子节点是黑色的很容易让人误解）
- 大部分没有讲红黑树怎么来的，有什么用
- 没有讲解红黑树的删除操作（删除比插入更复杂）
- 对代码的解读只是简单的代码注释，估计作者也是半懂不懂的