深入理解红黑树算法

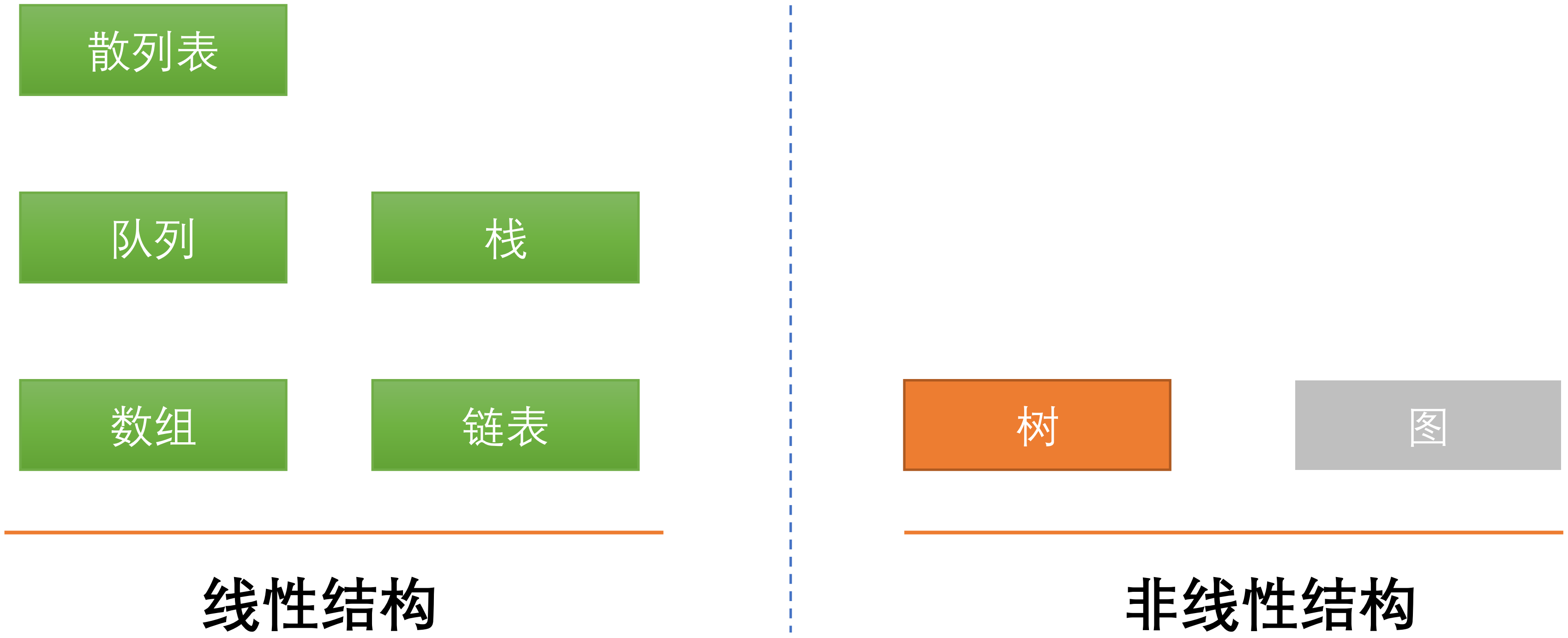
@远航君

主要内容

- 《数据结构与算法》基础
- 树的概念
- 二叉搜索树
- 树的旋转
- 学习红黑树
- Q & A

《数据结构与算法》基础

数据结构



算法

针对解决某一类问题的 **规律 or 套路**

举例：

1+2+3+4+5+ …… + 100 = ？

如何实现数组的快速排序

下棋怎么才能赢

算法的评判标准

时间复杂度

- 程序执行的 **耗时** 长短
- 程序基本操作执行次数
- 常用时间复杂度

$O(1)$

$O(\log_2 n)$

$O(n)$

$O(n^2)$

for

空间复杂度

- 算法存在中间数据
- 程序占用内存多少
- 常用空间复杂度

$O(1)$	int p = 0;
$O(n)$	new int[];
$O(n^2)$	new int[][];

绝大多数，时间复杂度 更为重要一些，我们宁可多占一些内存，也要提升程序性能（速度）

拿 空间 换 时间

常见的算法

排序算法

时间复杂度 $O(n^2)$

冒泡排序

插入排序

选择排序

时间复杂度 $O(\log_2 n)$

快速排序

归并排序

希尔排序

时间复杂度 $O(n)$

计数排序

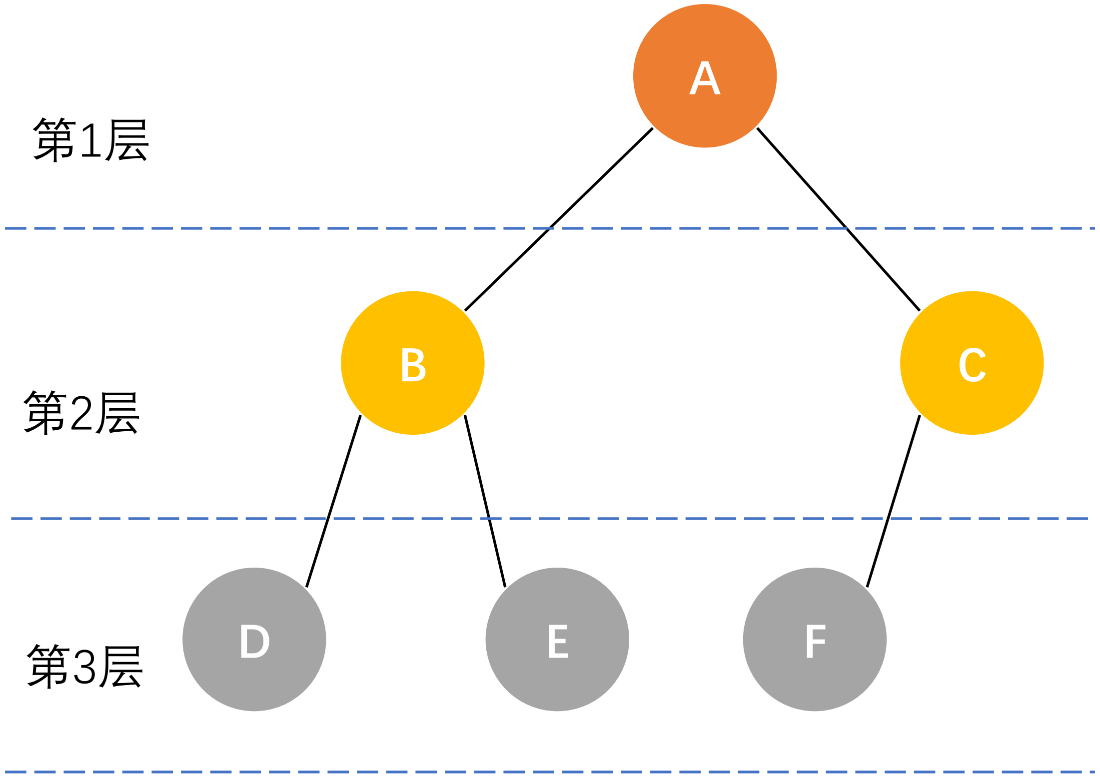
桶排序

查找算法

?

树的概念

二叉树的概念



- 根节点
- 叶子节点
- 树的深度
- 节点数量（每层 --$2^{(i-1)}$）
- 完全二叉树
- 满二叉树

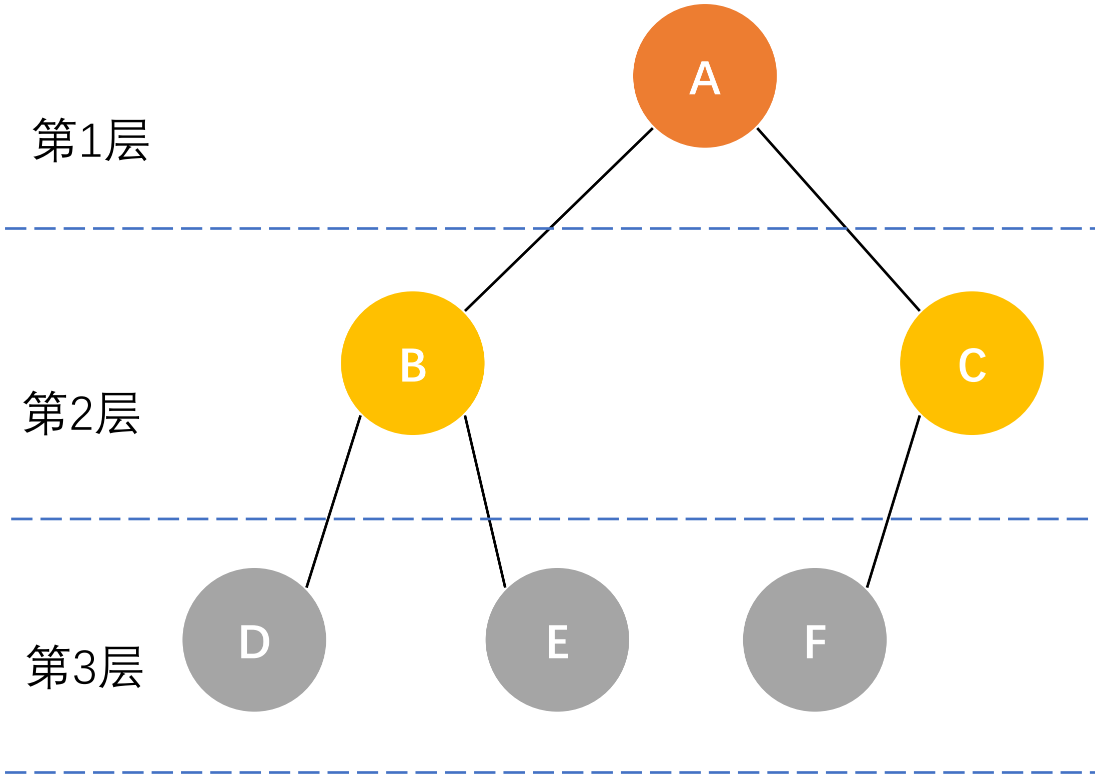
问题：有N个节点的 **完全二叉树** 的深度是多少

$\lfloor \log_2 N \rfloor + 1$

树节点

left	data	right

二叉树的遍历



前序遍历

根节点，左子树，右子树

ABDECF

中序遍历

左子树，根节点，右子树

DBEAFC

后序遍历

左子树，右子树，根节点

DEBFCA

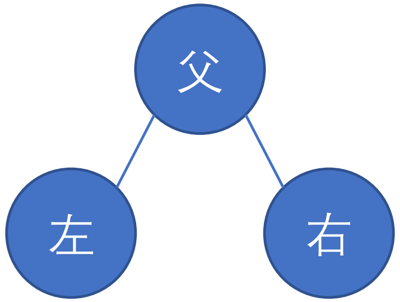
二叉搜索树

二叉搜索树

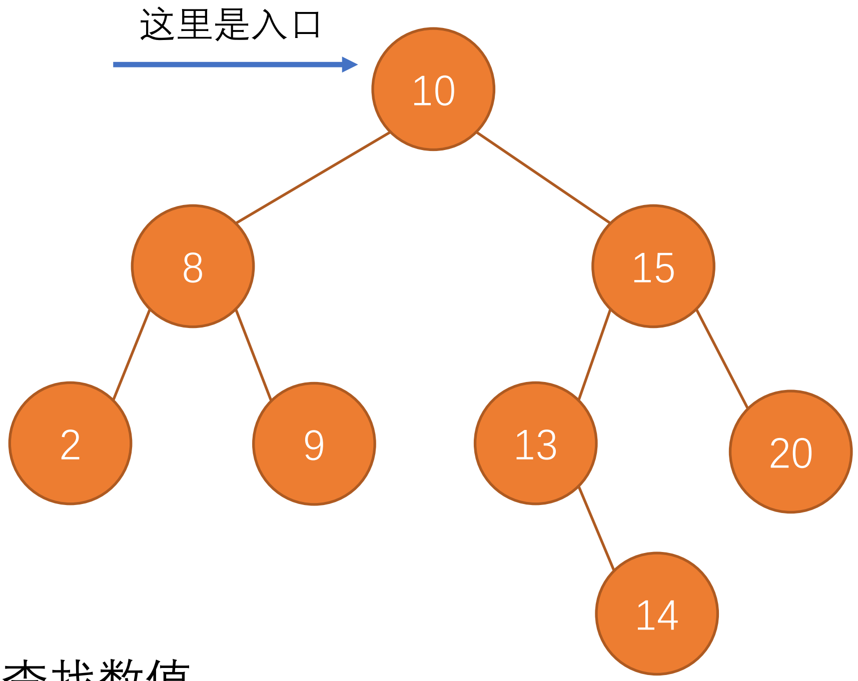
树节点

left	data	p	right

限制条件



左.data < 父.data < 右.data



查找数值

- 查找数值9
- 查找最大/小

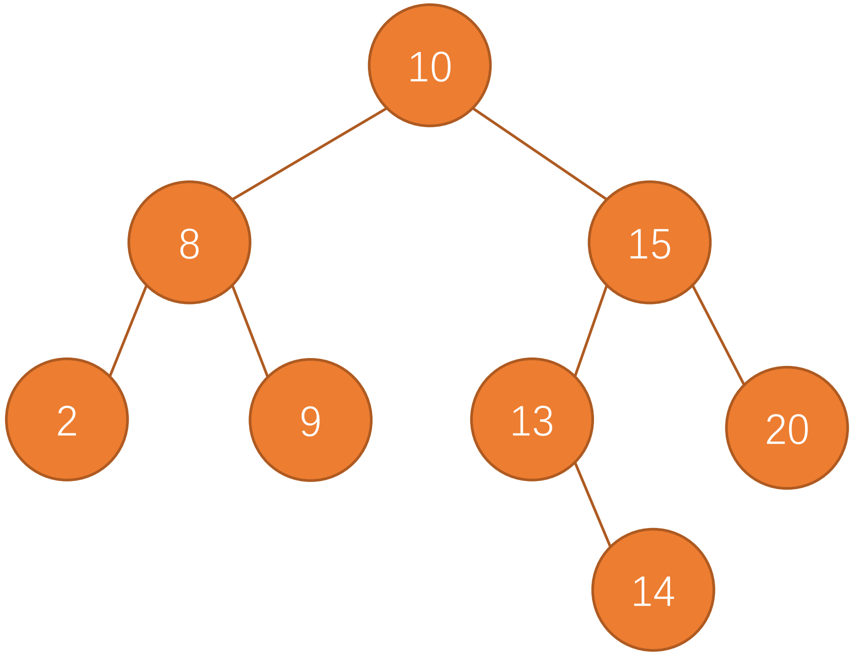
中序遍历

2, 8, 9, 10, 13, 14, 15, 20

▲

后继

二叉搜索树



查找后继

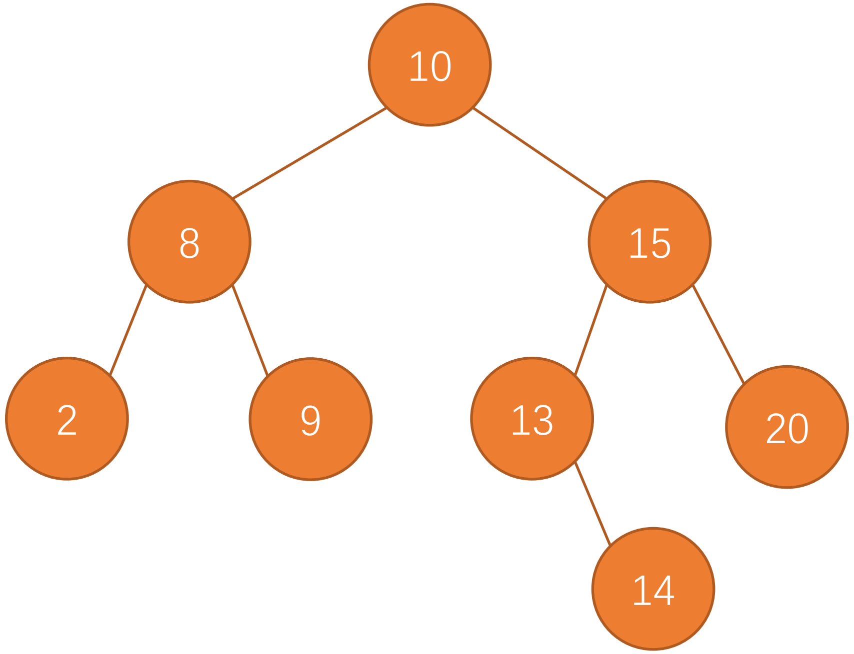
- 查找10的后继
- 查找14的后继

插入操作

- 新节点3

```
TREE-INSERT(T, z)
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z                // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```

二叉搜索树




```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```

删除操作

- 没有孩子：2
- 一个孩子：13
- 两个孩子：8，10

二叉搜索树

对于一棵深度为h的二叉搜索树（节点数N）

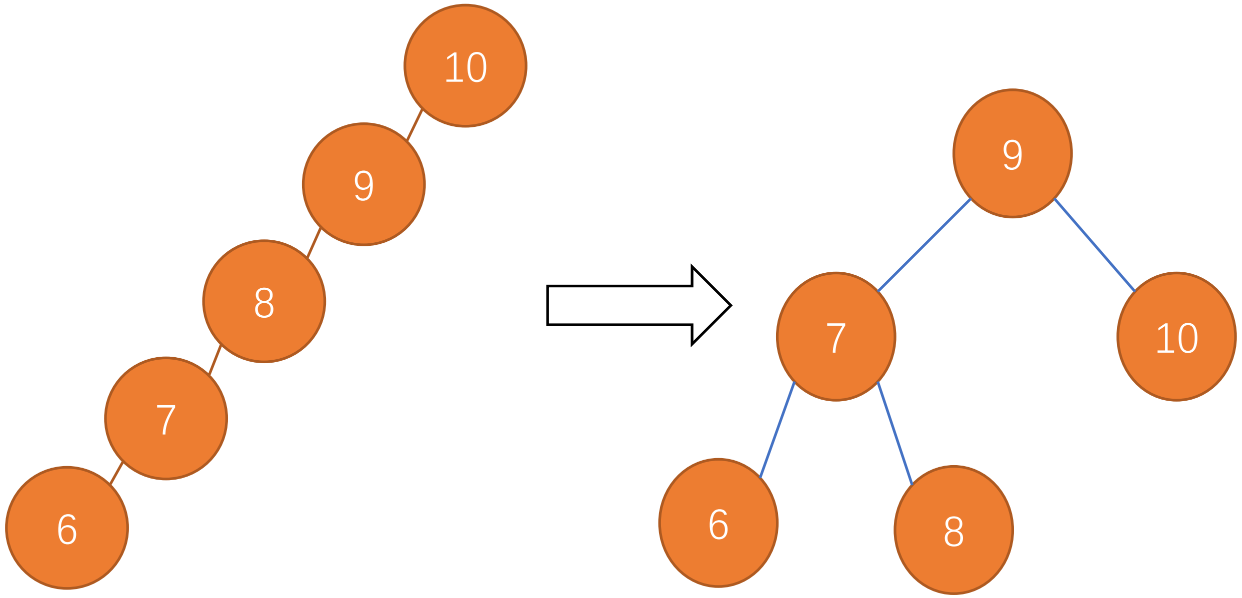
查找数值	O(h)
查找极值	O(h)
查找后继	O(h)
遍历操作	O(N)
插入操作	O(h)
删除操作	O(h)

注意：

h 不等于 $\lfloor \log_2 N \rfloor + 1$

想一想

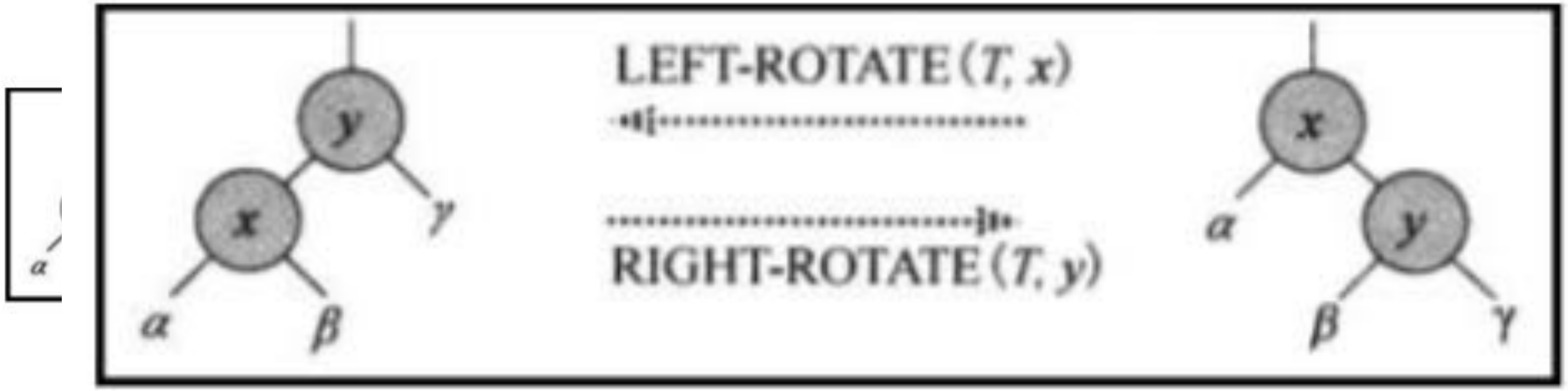
序列 10, 9, 8, 7, 6 构建二叉搜索树



树的旋转

树的旋转

树结构变化：是通过 **旋转** 来完成的

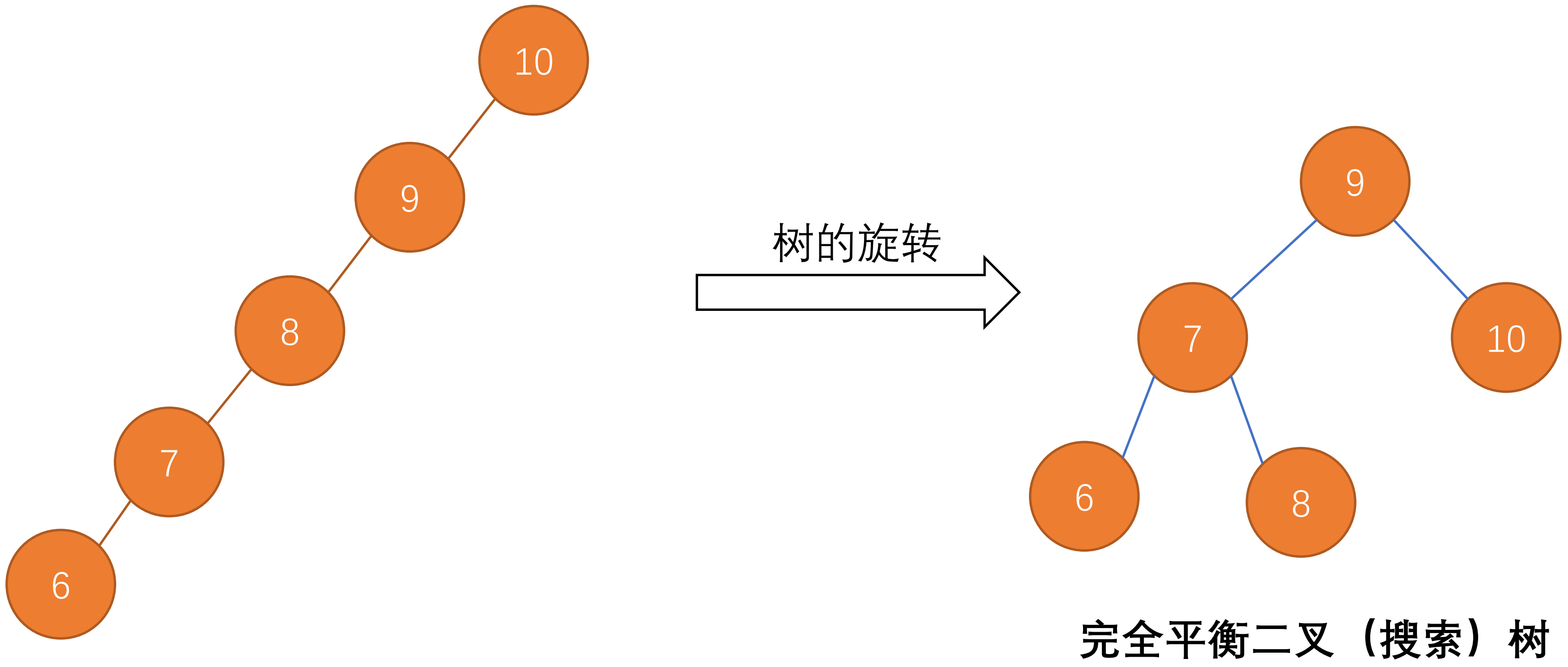



```
LEFT-ROTATE(T, x)
    y = x.right                     // set y
                                    // turn y's left subtree
                                       into x's right subtree

                                    // link x's parent to y

    elseif x == x.p.left
        x.p.left = y
    else x.p.right = y
    y.left = x                      // put x on y's left
    x.p = y
```

树的旋转



优点：查找时速度很快

缺点：插入速度慢，牵一发而动全身，需要不断旋转处理

学习红黑树

红黑树

声明

红黑树并不是完全平衡二叉搜索树，它能够确保没有一条路径是其他路径的2倍长度，是近似平衡的。

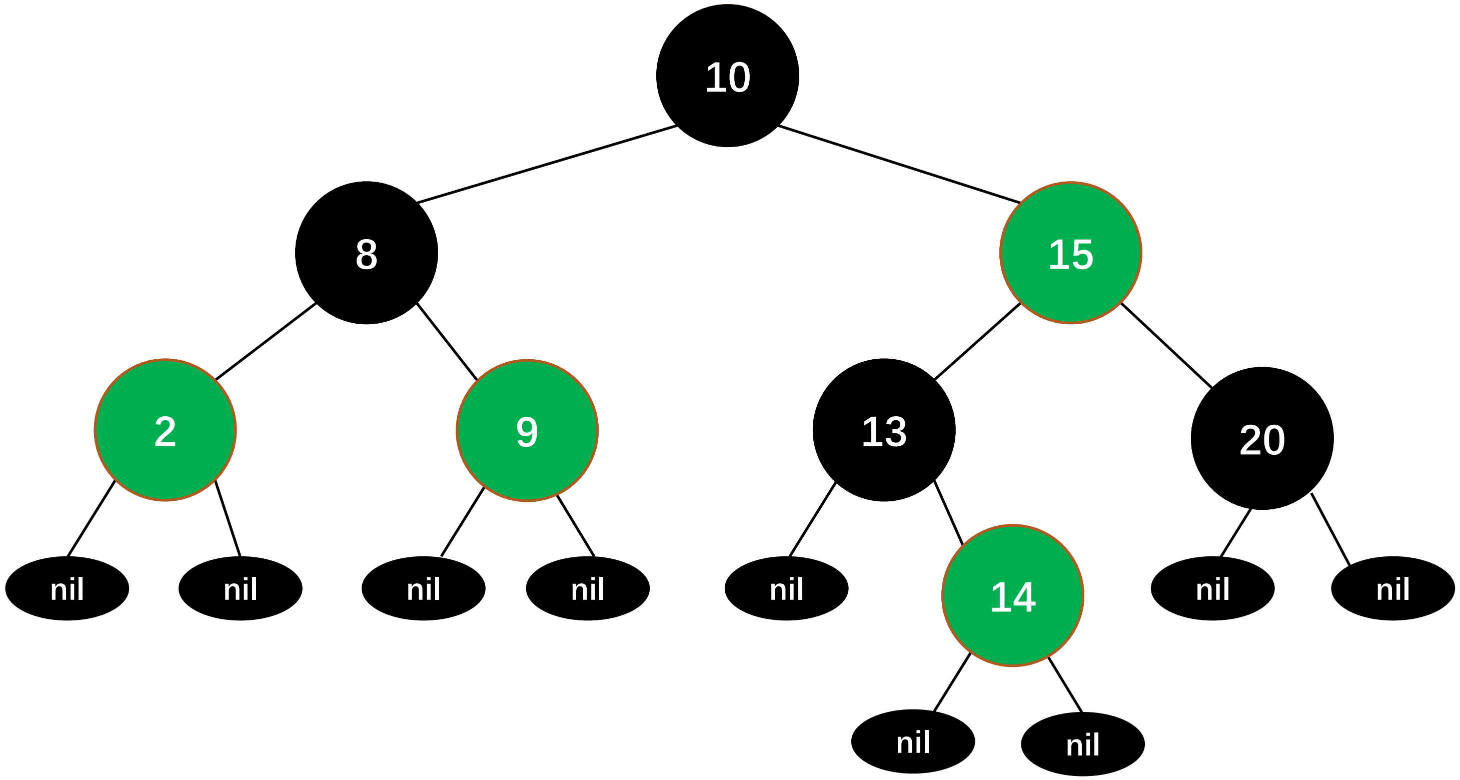
限制条件

1. 每个结点或是红色的，或是黑色的。
2. 根结点是黑色的。
3. 每个叶结点(NIL)是黑色的。
4. 如果一个结点是红色的，则它的两个子结点都是黑色的。
5. 对每个结点，从该结点到其所有后代叶结点的简单路径上，均包含相同数目的黑色结点。

树节点

left	data	p	color	right

红黑树



红黑树插入操作

规定

所有新插入的节点，都标记为 **红色**

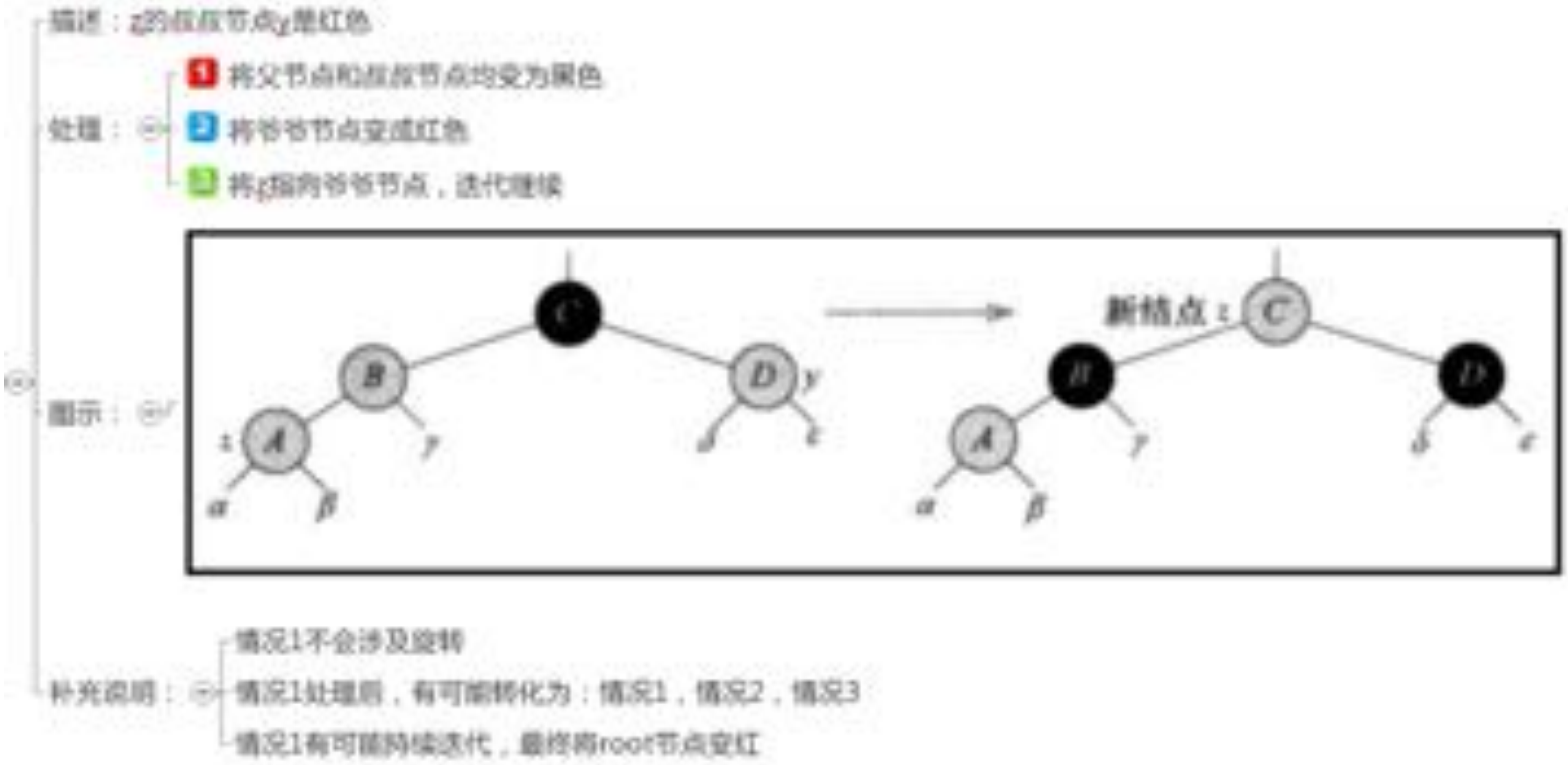
不满足的性质

1. 每个结点或是红色的，或是黑色的。
2. 根结点是黑色的。
3. 每个叶结点(NIL)是黑色的。
4. 如果一个结点是红色的，则它的两个子结点都是黑色的。
5. 对每个结点，从该结点到其所有后代叶结点的简单路径上，均包含相同数目的黑色结点。

性质2：新节点是根节点
性质4：新节点的父节点是红色的

红黑树插入操作

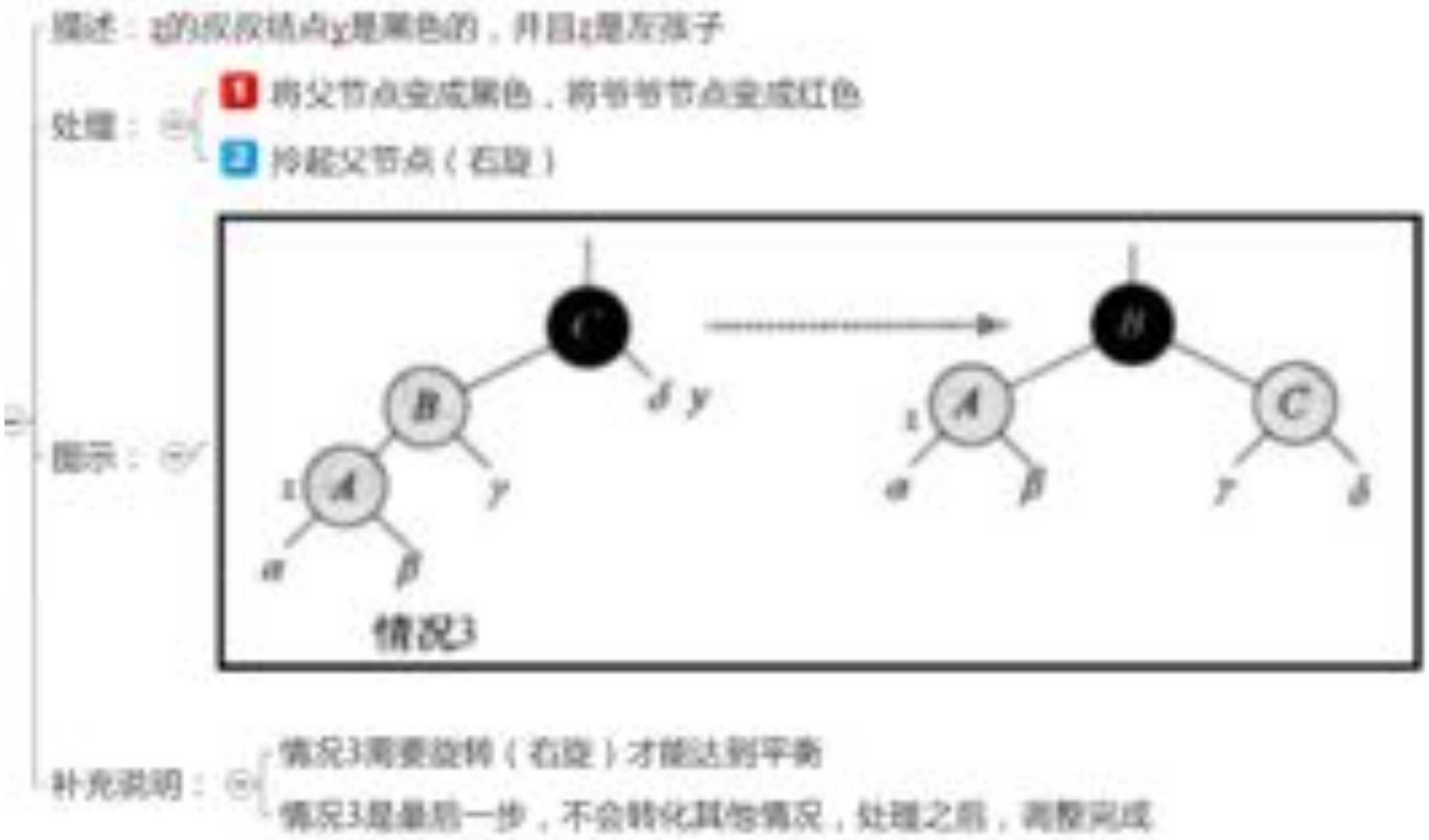
情况1



描述：z的叔叔节点y是红色
处理：
将父节点和叔叔节点均变为黑色
将爷爷节点变成红色
将z指向爷爷节点，迭代继续
图示：
新结点 z
补充说明：
情况1不会涉及旋转
情况1处理后，有可能转化为：情况1，情况2，情况3
情况1有可能持续迭代，最终将root节点变红

红黑树插入操作

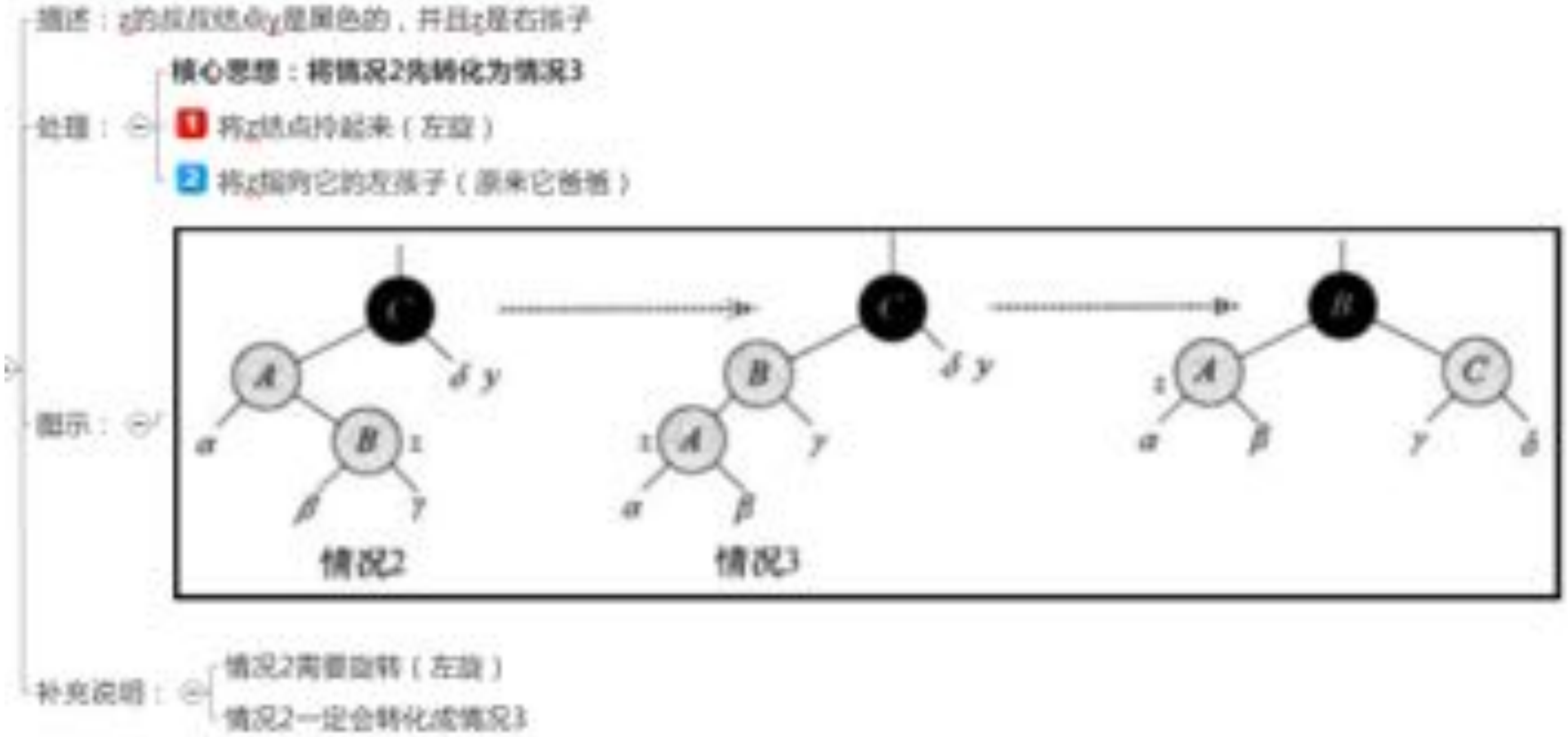
情况3



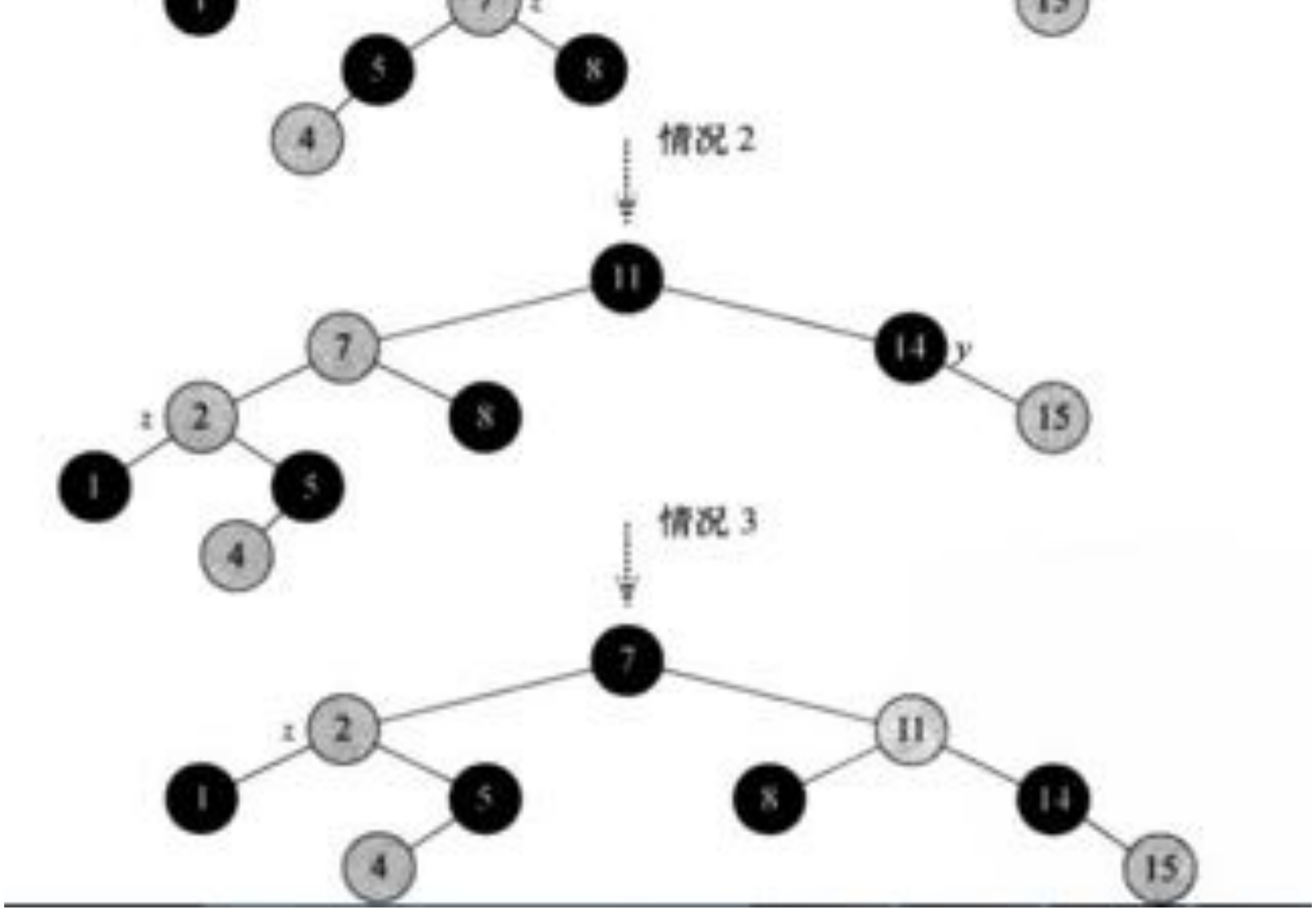
描述：z的叔叔结点y是黑色的，并且z是左孩子
处理：
1 将父节点变成黑色，将爷爷节点变成红色
2 拎起父节点（右旋）
图示：
情况3
补充说明：
情况3需要旋转（右旋）才能达到平衡
情况3是最后一步，不会转化其他情况，处理之后，调整完成

红黑树插入操作

情况2


描述：z的叔叔结点y是黑色的，并且z是右孩子
处理：
核心思想：将情况2先转化为情况3
1 将z结点拎起来（左旋）
2 将z指向它的左孩子（原来它爸爸）
图示：
情况2
情况3
补充说明：
情况2需要旋转（左旋）
情况2一定会转化成情况3

红黑树插入操作



Q & A