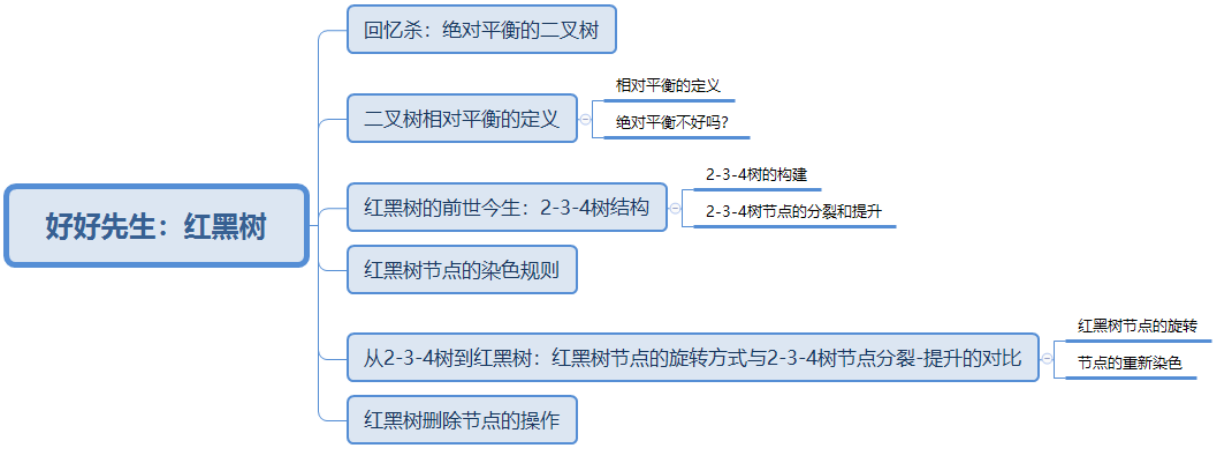


0.写在前面

红黑树（Red-Black Tree），同样是二叉树结构的典型应用，并且属于二叉排序树的分支、AVL树的升级版本

那么为什么我们要将红黑树部分单独提取出来写成一篇文章进行讨论呢？原因有二

1.红黑树是近年来笔试面试过程中非常常见的内容，从理论到实际应用，比比皆是，所以红黑树比较重要

2.红黑树结构相比起前面的二叉排序树和AVL树来讲，更为复杂，我们需要了解更多的铺垫内容，才能够更好的了解红黑树结构

所以，综上原因所述，我们将红黑树部分的内容单独提取出来进行讲解，方便各位同学的阅读和理解

1.好好先生：红黑树

①回忆杀：绝对平衡的二叉树

在之前我们学习AVL树的时候，曾经介绍过AVL树绝对平衡的概念

如果一个二叉排序树是绝对平衡的，那么就说明在这个二叉排序树中的任何节点，其左右子树的深度之差的绝对值不超过1

正是这种绝对平衡的概念，让原本可能退化成为单链表的二叉排序树，能够通过对不平衡节点的旋转操作，来保持平衡

而平衡之后的二叉排序树，其查找元素的效率大大上升，查询任何元素的时间复杂度，都很好的控制在O(logn)之内

但是，这种查询效率的提升，实际上是通过牺牲元素的插入和删除效率换来的。

②二叉树相对平衡的定义

那么，有没有办法在保证查询效率的基础上，还能够提升二叉排序树的增删效率呢？

那么答案就是：牺牲一部分查询效率，换取增删效率的提升

具有这种特性的二叉排序树结构，我们称之为相对平衡的二叉排序树

1.相对平衡的定义

二叉排序树的相对平衡，指的是这个二叉排序树中的任何节点的左右子树之间，其深度差不超过2倍的关系

例如：在一个相对平衡的二叉排序树中，某节点左子树的深度是4，那么其右子树的深度取值范围在[2, 8]之间

如果右子树的深度超过这个范围，那么就需要对这个树结构进行旋转，重新调整为相对平衡的状态

2.绝对平衡不好吗？

在之前的课程中我们讨论过，在AVL树中，我们每插入一个节点或者删除一个节点之后，都有可能引起3代至4代之中的某一节点不平衡

也就是说，我们在添加和删除节点之后，必须对节点3代至4代内的父节点的平衡性重新进行审查，如果出现不平衡的节点，就要通过旋转调整其平衡性

这样一来就是的AVL树的增删操作开销极大

所以我们不能说AVL树不好，我们只能说，频繁的对树结构进行旋转调整，在元素增删方面开销太大

所以正如前面对相对平衡结构的定义所说的，我们允许相对平衡的二叉排序树的左右孩子之间，深度差保持在2倍的关系之内

这样的话，在这个相对平衡的二叉排序树当中，如果同时保存了很多节点的话，那么在加入新节点或者删除原有节点的时候所引起的旋转调整的概率

是远低于绝对平衡的AVL树的旋转调整操作的概率的

但是同样是在保存大量节点的情况下，相对平衡二叉排序树的元素查找效率，略低于绝对平衡的AVL树的元素查找效率

但是整体的查找时间复杂度依然控制在O(logn)范围内，所以这种差异是相对较小的

所以我们可以总结：相对平衡的二叉排序树，是使用较小查询代价的牺牲，换取了较大增删效率的提升

这波交易，不亏！

③红黑树的前世今生：2-3-4树结构

在上面的章节中，我们论证了相对平衡二叉排序树在效能方面的理论内容

那么实际在数据结构领域，有一种二叉树结构完美的实现了二叉排序树的相对平衡性，这种二叉树被我们称之为：红黑树

可能学习过红黑树结构的同学都有这样一种感觉：红黑树的限制实在是太多了，例如节点的染色、不同颜色节点之间的相连关系等等，十分复杂难懂

但是，如果我们在学习红黑树结构之前先搞明白红黑树结构的前世今生，那么实际上我们还是能够很容易找到红黑树结构的特征，并学习其中的原理的

红黑树的前世，叫做2-3-4树

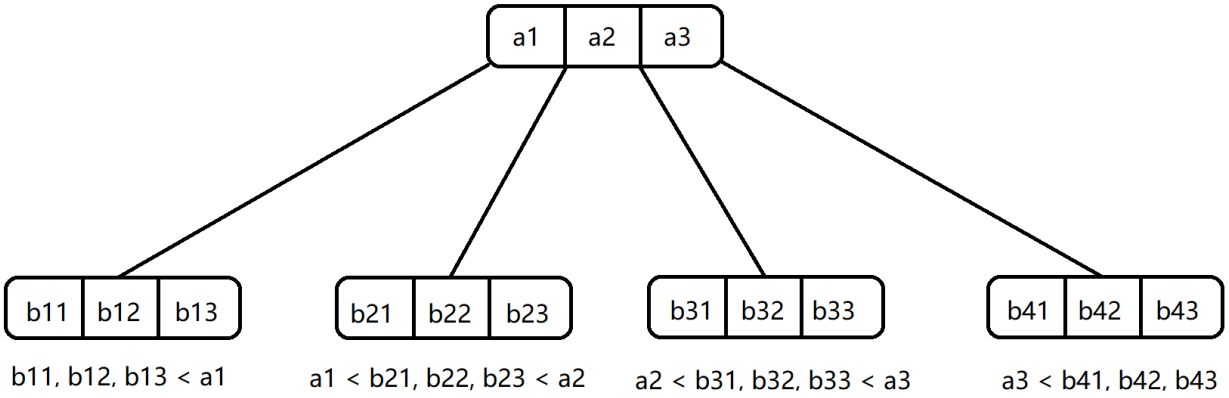
下面，我们就来一起学习一下，什么是2-3-4树

1.2-3-4树的构建

2-3-4树的结构和二叉树的结构具有很大的相似性：2-3-4树的每一个节点中最多能够存储3个数据，所以其每一个节点最多能够具有4个子节点

这4个子节点中保存的数据，与父节点中的3个数据之间的关系同样是是中序遍历有序的，也就是说：

假设存在如下2-3-4树节点之间的关系，则孩子节点中数据的取值范围与父节点中保存数据的关系是：



在明白了2-3-4树结构的特征之后，下面我们一起来研究一下2-3-4进行元素添加的流程

2.2-3-4树节点的分裂和提升

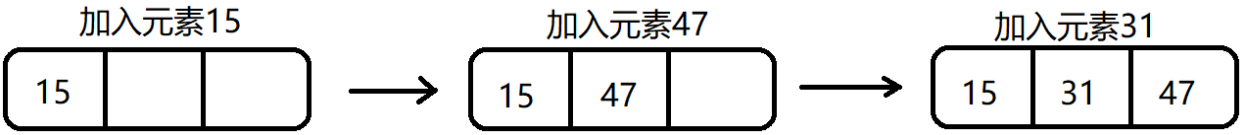
在2-3-4树结构中，我们只能够向叶子节点中添加元素，并且在添加元素的时候，每一个叶子节点内部的元素之间也是相对有序的

这表示在向2-3-4树中添加元素的时候，可能会在节点内部引起2-3个元素的位移

如果叶子节点的数据域添加满了，那么我们将会让这个叶子结点进行分裂和提升操作，进而得到更多的2-3-4树的层数和节点

下面我们将一起从2-3-4树只有一个根节点的情况说起，一起研究2-3-4树的构建过程

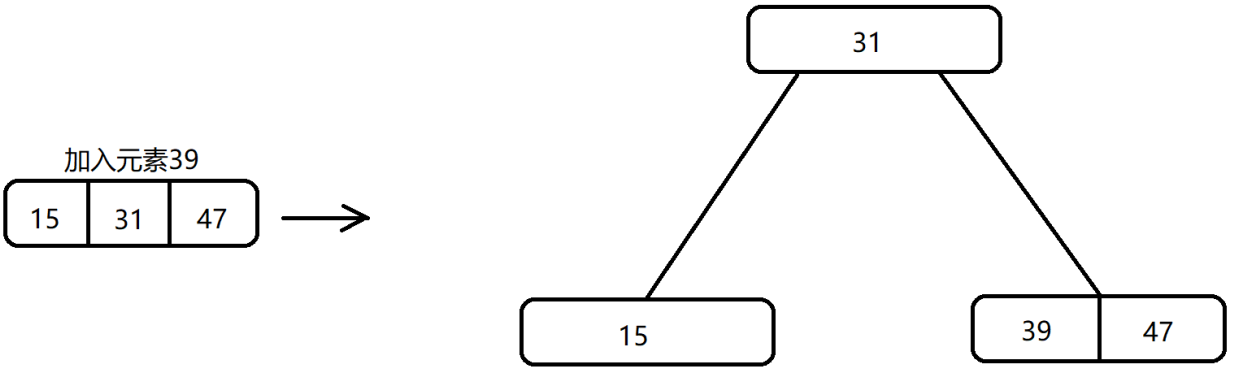
过程1：当前2-3-4树结构中只有一个根节点，向根节点中添加数据：



过程2：当根节点的数据域已经存放满之后，再次向其中添加元素的时候，将会导致节点的分裂

节点在分裂后，原始节点中间的元素向上提升，成为一个新的根节点，原始节点中左右两边的两个元素，分别成为新的根节点的左右孩子节点

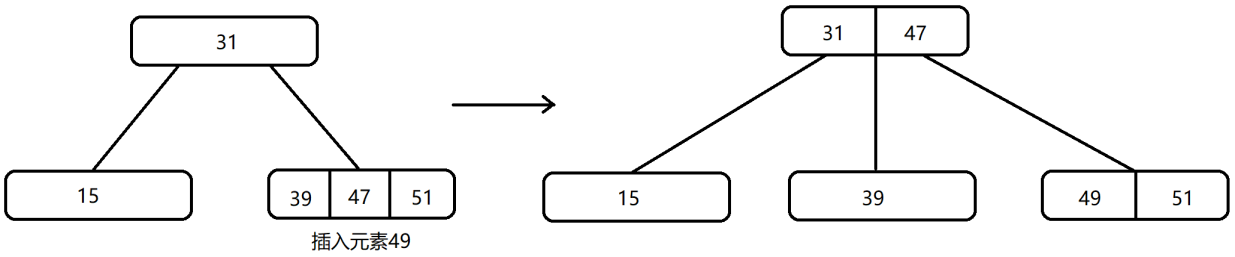
新加入其中的元素，根据插入位置，落在左右孩子节点中



过程3：当在向叶子节点中添加元素，并导致叶子节点元素添加满后，叶子结点将再次进行分裂，并将叶子结点的中间元素向上提升

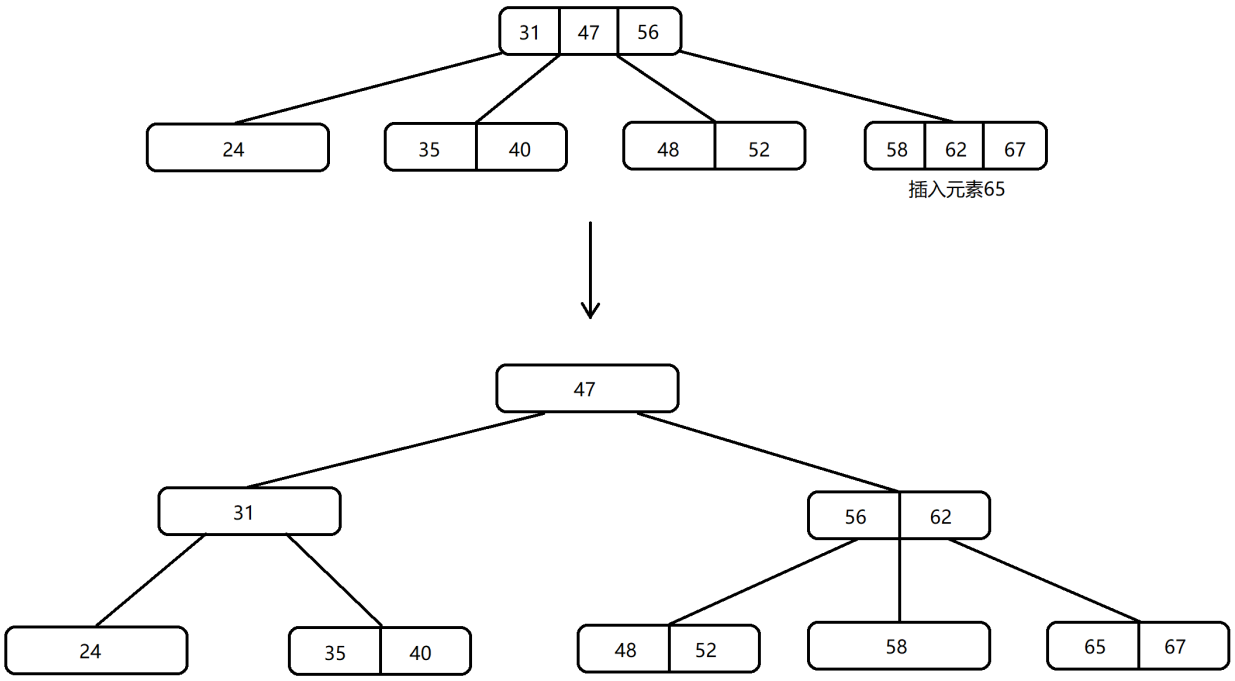
如果这个叶子结点具有父节点，并且父节点的数据域中尚有空位，那么提升的元素将插入父节点的数据域当中，这个叶子结点分裂出来的两个新的左右节点

也将成为父节点的两个孩子节点，此时2-3-4树结构的层数不会增加



过程4：如果在叶子节点分类并提升之后，导致父节点同样发生元素插满的现象发生，那么父节点同样需要进行分裂和提升

直到分裂出来的节点插入一个没有插满的父节点中，或者根节点分裂提升，导致2-3-4树结构的层数+1



从上面的过程中我们不难发现，不管2-3-4树结构中存在多少节点，不论2-3-4树的节点结构如何分裂和提升

2-3-4树结构本身都是一种完美平衡的树结构，这是因为2-3-4树的构建过程是自顶向上、从两边向中间完成的，

所以2-3-4树才能够保证任何一个父节点下，要么不存在子节点，要是存在子节点，则必然能够填满父节点下的所有分支（实际上父节点是因为子节点的分裂“挤”出来的）

完美平衡是一种比绝对平衡还要更加平衡的结构：完美平衡的树结构中的任何一个节点的任何一个分支，其深度都是相同的

虽然2-3-4树具有完美的平衡性，并且也不会在插入节点和删除节点的时候频繁的发生分裂和提升

但是因为其在编码方面的难度比较高，所以在真实开发环境中，2-3-4树结构基本用不到

我们一般使用的，都是2-3-4树结构的等价结构：红黑树结构

那么，2-3-4树结构和红黑树结构之间具有什么样的等价关系呢？下面就让我们开始进行红黑树结构的学习

④红黑树节点的染色规则

在学习红黑树结构之前，我们先来认识一下红黑树结构节点的染色规则：

规则1：节点要么是红色要么是黑色（废话……要不为啥叫你红黑树……）

规则2：根节点是黑色的

规则3：每个叶子节点是黑色的。注意：这里的叶子结点并不是指没有左右孩子的节点，而是指这种节点的左右为null的部分

规则4：如果一个节点是红色的，那么他的左右孩子必须是黑色的

规则5：从一个节点出发，到其任何一代子节点的路径上，所经历的黑节点的数量是相同的

怎么样？看完上面的规则，有没有一种想要掀桌子的冲动？没错，这就是红黑树！

红黑树就是建立在上述极其复杂的规则之上的，也就是拜这些规则所赐，红黑树才能够保持近似平衡性

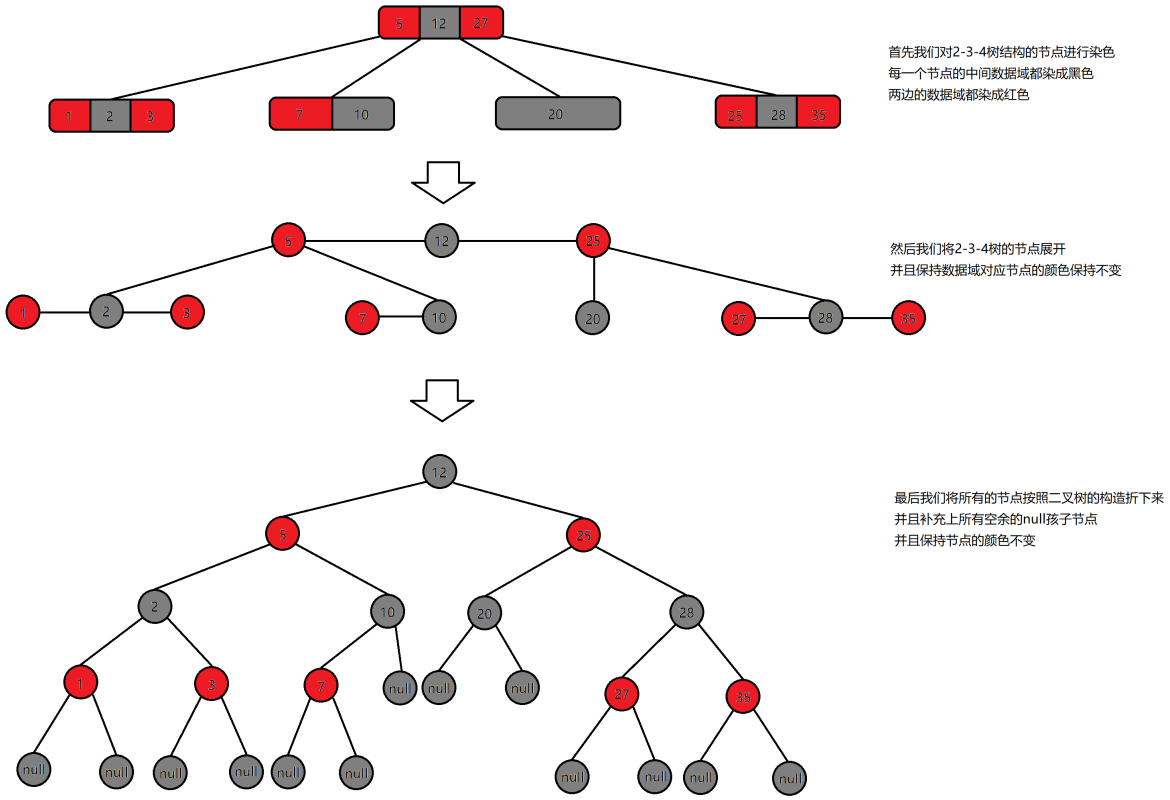
如果在构建红黑树，或者删除其节点的时候导致破坏了上述规则，那么我们必须对红黑树的不平衡节点进行旋转，已达到重新平衡的目的

（关于节点旋转的内容，详见上一篇内容的对平衡二叉树结构的说明）

⑤从2-3-4树到红黑树：红黑树节点的旋转方式与2-3-4树节点分裂-提升的比较

上面我们所说的红黑树的构建规则乍一看上去十分复杂，不便于理解，但是我们之前曾经说过

2-3-4树结构和红黑树结构是等价的，所以二者之间必然存在相关联的地方，下面我们就通过一张图片，来解释二者之间的关联关系



怎么样？一个红黑树是不是就出现了？！

如果你比对一下上面所说的所有规则，你会发现：这样的一个经由2-3-4树结构扩展而来的二叉排序树结构

是完全符合红黑树的所有节点染色规则的！

所以这也就印证了刚才我们说过的一句话：红黑树结构就是一种等价的2-3-4树结构

那么，在2-3-4树中，节点的分裂和提升动作，与红黑树中节点的旋转操作，又具有怎样的对应呢?

下面我们一起来看一下他们之间的对应关系

1.红黑树节点的旋转

红黑树节点的旋转方式和AVL树是相同的，依然遵循：左左不平右旋，右右不平左旋，左右不平左右旋，右左不平右左旋的规则

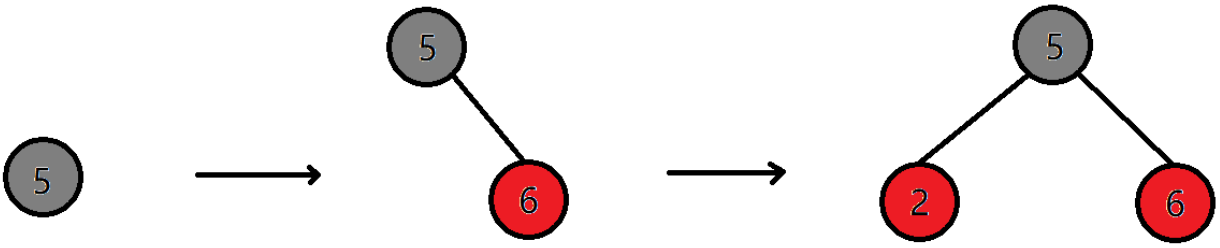
只不过，在红黑树中对节点不平衡的判断标准是：该节点左右子树之间的深度差超过2倍的关系

下面我们来比较一下红黑树中引发旋转的操作，与在2-3-4树节点中对应的情况：

首先，我们假设在2-3-4树结构中，每一个首先加入节点的元素，都占据了最中间的数据域位置，即在红黑树中的黑色节点

当后序元素加入这个节点的时候，如果恰巧落在原有元素的左右两边，即红黑树中红色节点的位置，则不会引发原有元素数据域位置的改变

此时这种情况在红黑树中同样不会引发节点的旋转



但是如果新元素在加入节点的过程中，导致原有节点占有的数据域的移动，则会对应的在红黑树中导致节点的旋转

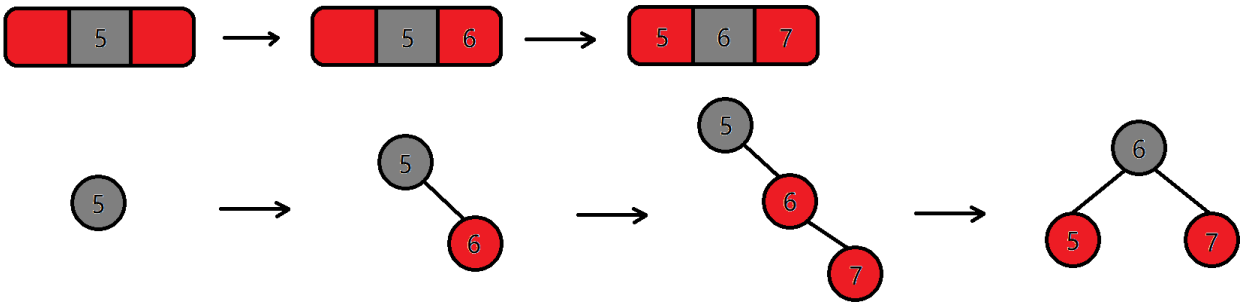
左旋:

步骤1：第一个新加入元素比原始元素大

步骤2：第二个新加入元素比第一个新加入元素还大

步骤3：原始元素左移（红黑树节点左旋）

步骤4：重新染色



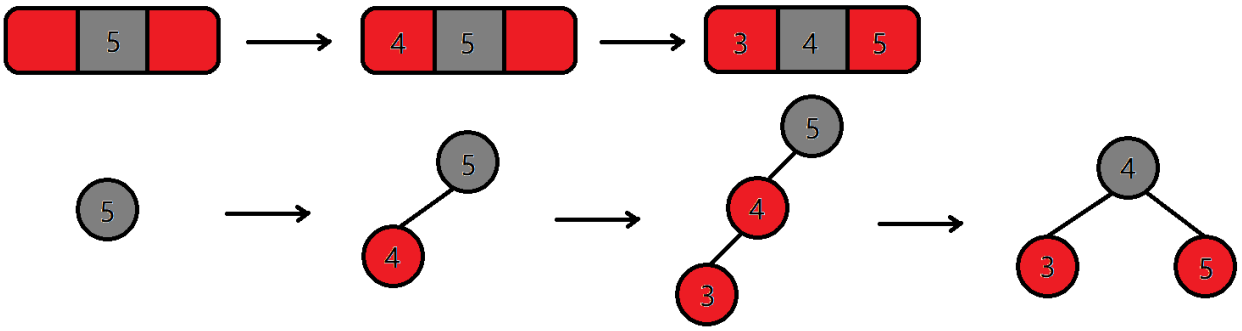
右旋：

步骤1：第一个新加入元素比原始元素小

步骤2：第二个先加入元素比第一个新加入元素还小

步骤3：原始元素右移（红黑树节点右旋）

步骤4：重新染色



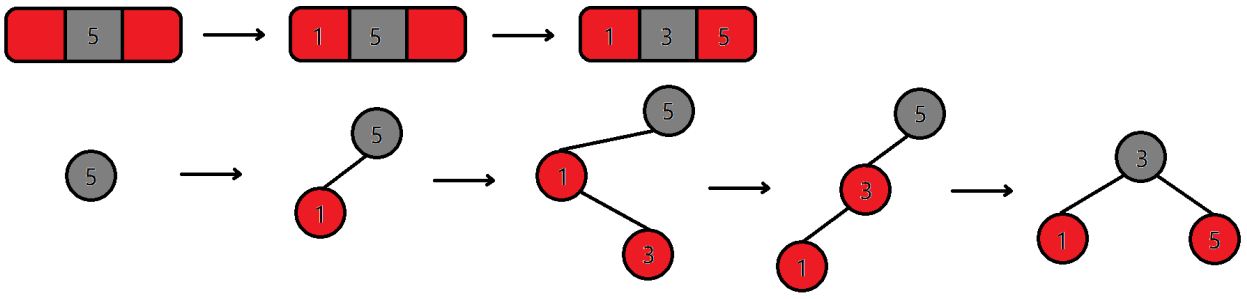
左右旋：

步骤1：第一个新加入元素比原始元素小

步骤2：第二个先加入元素的大小介于第一个新加入元素和原始元素之间

步骤3：原始元素右移（红黑树节点左右旋）

步骤4：重新染色



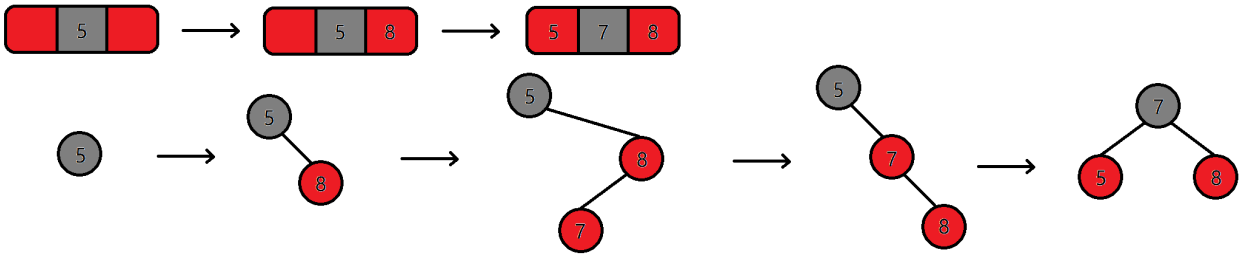
右左旋：

步骤1：第一个新加入元素比原始元素大

步骤2：第二个新加入元素的大小介于第一个新加入元素和原始元素之间

步骤3：原始元素左移（红黑树节点右左旋）

步骤4：重新染色



上面我们只是演示了在红黑树中，3代以内出现不平衡节点的旋转情况。和AVL树相似的是，红黑树中也有可能因为节点的添加和删除而引发

在4代（或以上）节点中出现不平衡节点的情况。这些情况在2-3-4树中对应的，就是因为叶子结点分裂和提升，而导致父级节点中元素发生未知变化的情况

而这些情况对应的不平衡状态，同样适用于上述描述的旋转方式，故在此不做进一步的描述。

从上面对于节点旋转的操作描述我们可以看出，在红黑树中，所有节点的旋转都是伴随着节点重新染色而发生的

实际上我们应该将这个流程倒过来描述：因为两个红色节点相遇了，所以才导致了节点的不平衡和旋转的发生

但是在红黑树结构中，实际上还存在着节点重新染色，但是并不会引发旋转的情况发生，因为此时虽然是两个红色节点相遇，但是并未引起任何节点的不平衡

也就是说：只有导致红黑树对应2-3-4树结构中，某一节点下最先插入其中的元素存储位置发生变化的情况，才会导致旋转的发生

这种情况我们将在下一话题中进行讨论

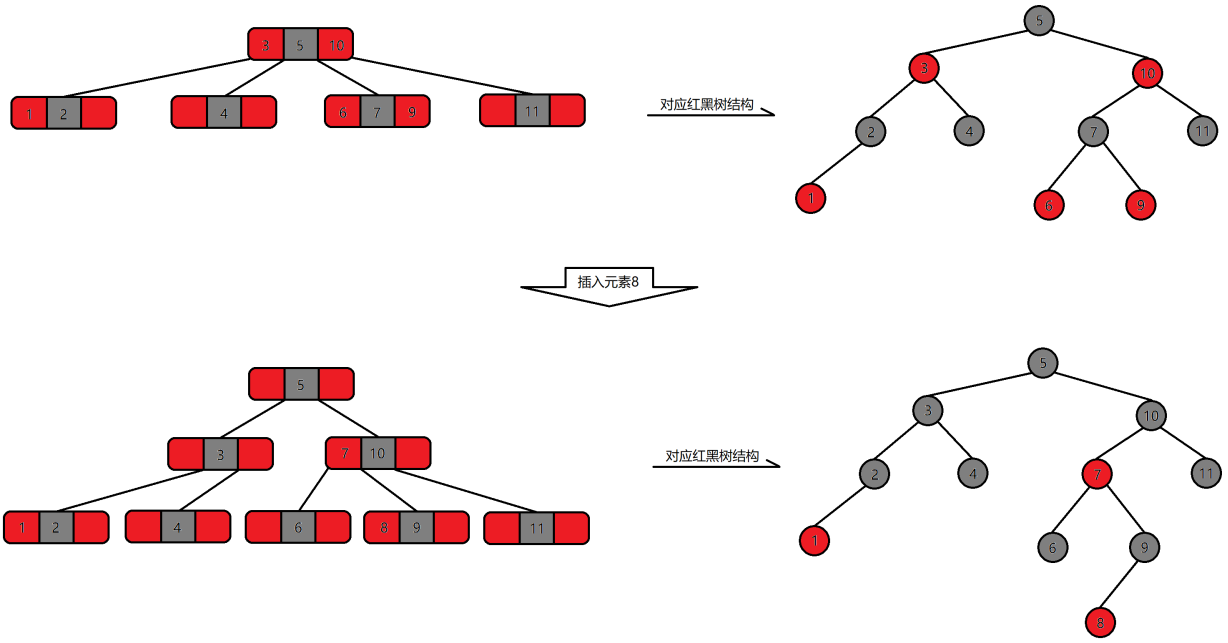
2. 节点的重新染色

在上面我们说过，只有在插入元素的时候，引起2-3-4树结构中，某一节点中最先插入元素的位移的情况下，才会导致对应红黑树节点旋转的发生

那么反过来说就是：如果在插入元素的时候，不会导致2-3-4树中节点最先插入元素的位移，那么即使发生了2-3-4树节点的分裂和提升，其对应的红黑树结构也不会发生旋转

这种情况下，我们往往只需要对红黑树中的节点进行重新染色，就能够让合格红黑树结构重新符合之前所说的标准

下面我们用一张图片来说明这种情况：



从上图中我们不难看出：在添加节点的时候，虽然导致了2-3-4树结构节点的分裂和提升，但是在对应的红黑树结构中，并没有出现任何旋转的节点

因为在上图的例子中，2-3-4树的节点在分裂和提升之后，提升的节点在插入父节点中的时候，并没有导致父节点中原有元素的位移

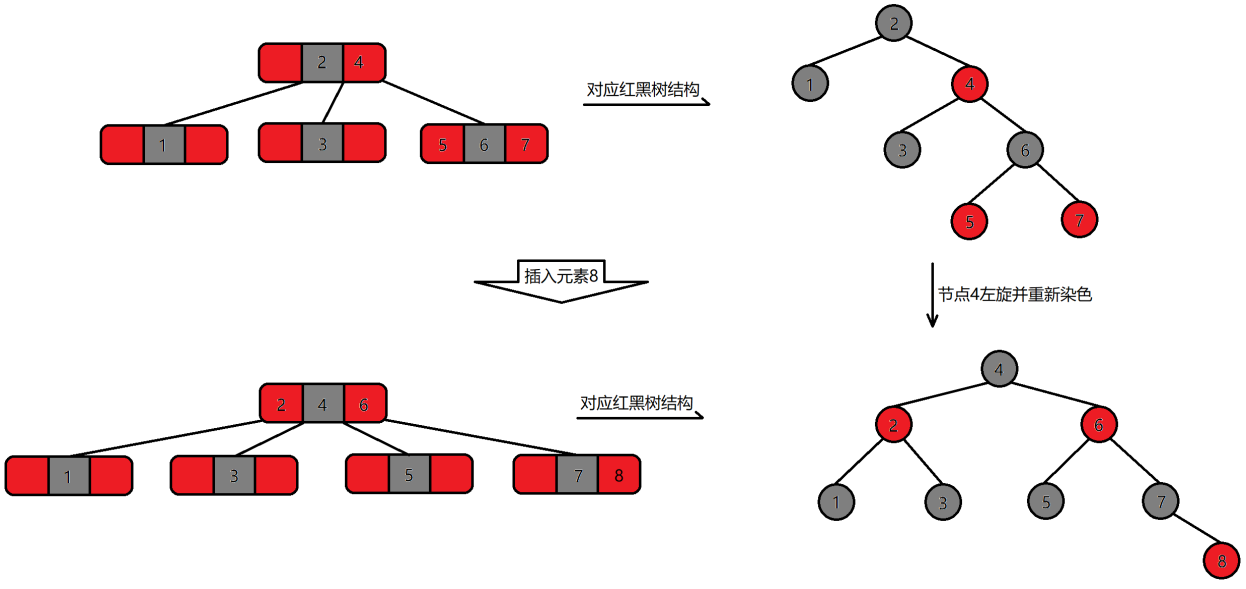
所以在对应的红黑树结构中，也就没有出现需要通过旋转进行保持平衡的节点

但是在2-3-4树结构中，提升的节点从原来的，位于节点中间的位置，变成了位于其父节点两边的位置，所以此时这个元素对应的红黑树节点的颜色也就发生了变化

那么此时，在通过对应结构的2-3-4树绘制出来的红黑树当中，理所当然的就要重新对这些节点进行染色操作

也就是说此时的红核数结构，只要改变几个节点的颜色，就能够重新符合所有的红黑树构建规则了

注意：如果在2-3-4树结构中，节点的分裂与提升导致父节点中原有元素的位移，那么同样需要对这个父节点进行旋转，才能够保证对应红黑树结构的平衡性



上图中，在插入节点8之后，2-3-4树结构的高度并没有发生变化，但是在节点(5,6,7)发生分裂和提升的时候，元素7上升至节点(2,4)中

并且导致其父节点中的原有元素进行了左移，相应的，这也会导致在其对应结构的红黑树中，节点4发生左旋

⑥红黑树删除节点的操作

在研究红黑树删除节点的操作之前，我们先来看一下在2-3-4树中删除节点的操作

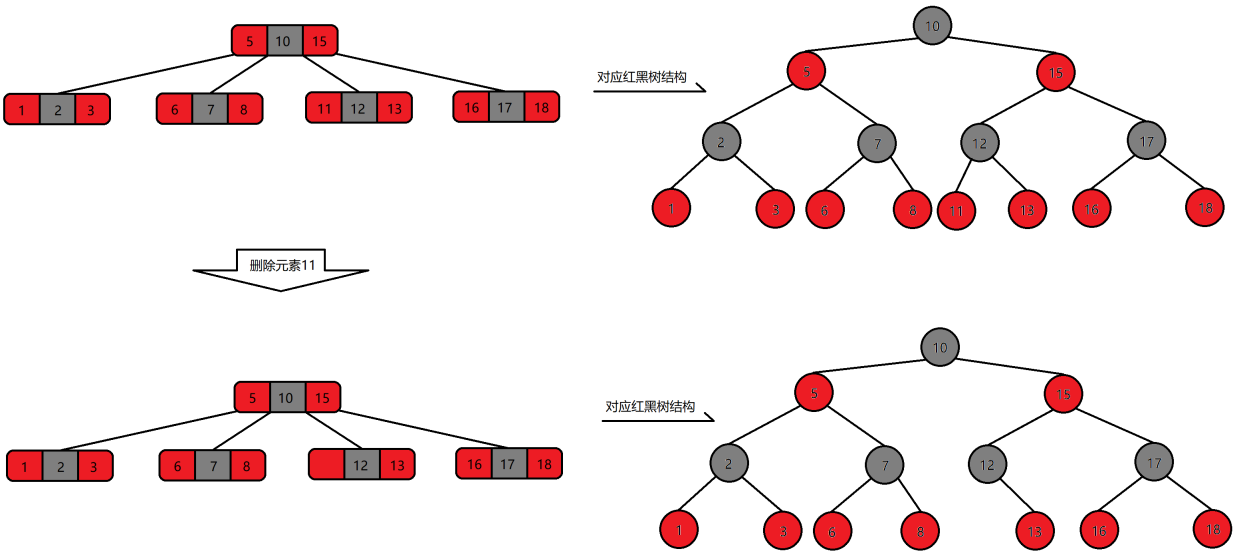
在2-3-4树结构中进行删除元素的操作，我们大致上可以分为如下两种情况：删除叶子结点中的元素和删除父节点中的元素

在上述两种情况中，删除2-3-4树结构叶子节点中的元素情况比较复杂，我们优先讨论；而删除父节点中的元素的操作相对比较简单，我们放在后面讨论

1.删除2-3-4树叶子结点中元素的操作

情况1：如果删除的是2-3-4树叶子结点中，左右两侧的元素，并且这个叶子节点中的元素没有被删除空，那么将不会对2-3-4树整体结构产生影响

所以，在对应的红黑树结构中也不会影响平衡性，并且节点颜色不需要重新染色



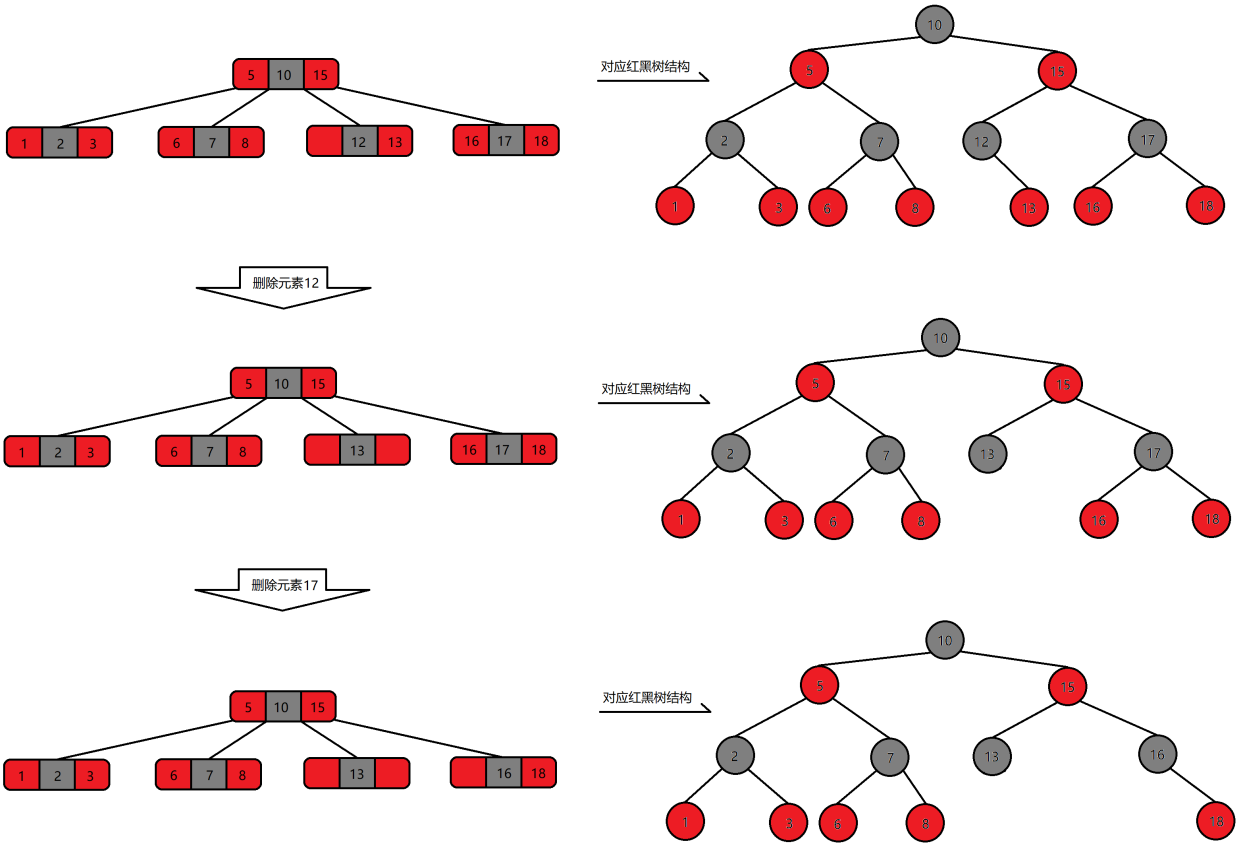
情况2：在2-3-4树结构中，某叶子结点只有中间元素和左侧元素，或者只有中间元素和右侧元素，那么如果此时删除这个叶子结点的中间元素

则在2-3-4树的这个叶子节点中，使用仅有的左侧元素或者右侧元素去替代这个中间元素的位置即可；

如果在这个叶子节点中，左右两侧的元素都存在，并且删除的是中间节点，则从理论上来说使用左侧或者右侧的元素来替换这个中间节点都是可以的

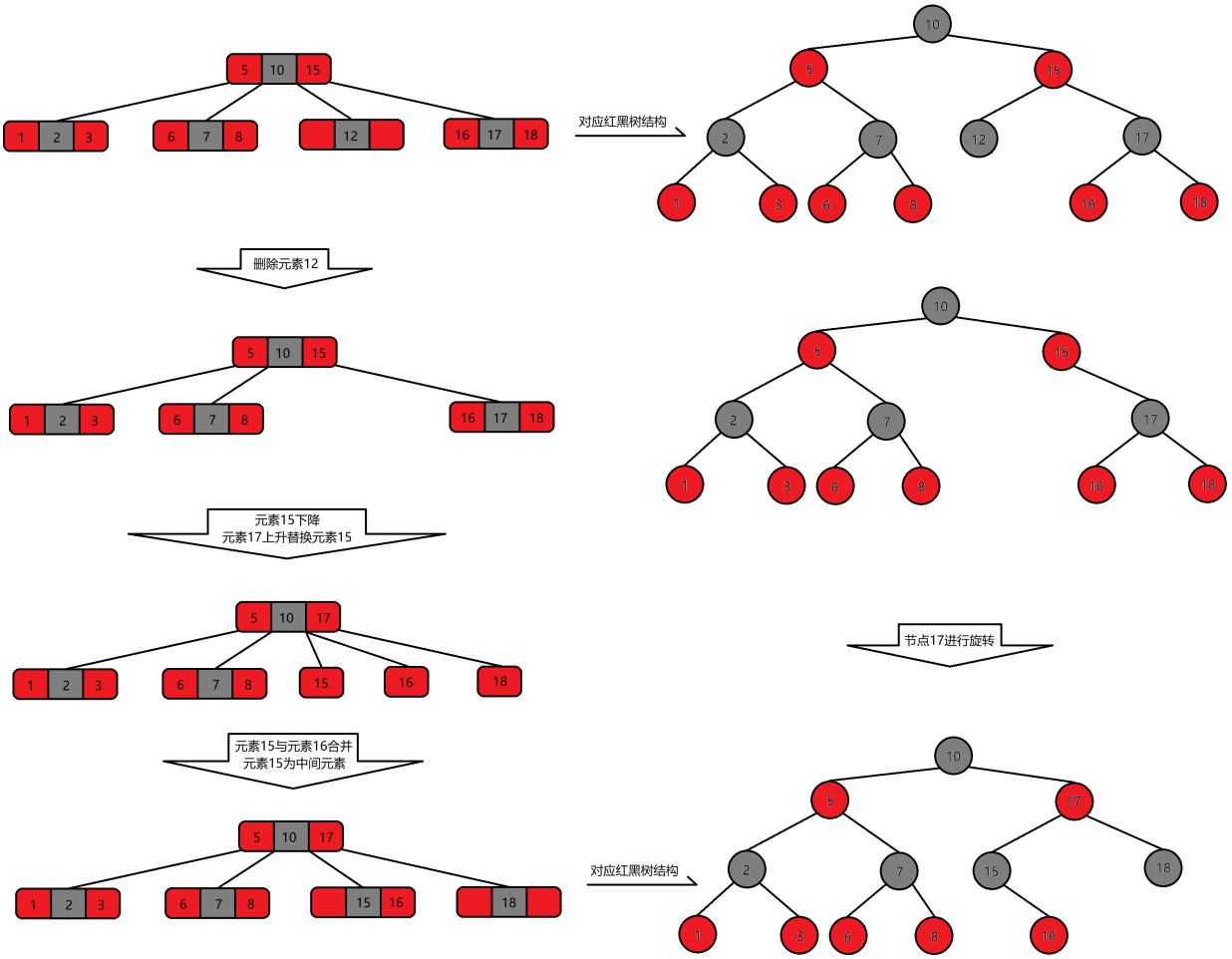
但是在红黑树结构中，选择的是使用左孩子，也就是2-3-4树节点中的左侧元素来替代这个中间元素

实际上上述所讨论的内容，在红黑树结构中对应的情景近视删除一个带有左右孩子节点的倒数第2层节点的情况，这种情况不会导致红黑树节点的旋转，但是会导致节点的重新染色



情况3：如果在2-3-4树结构中，我们将一个叶子结点整体删除干净，那么我们需要将父节点中的元素下移，形成新的叶子节点

但是父节点中空缺的位置由谁来进行补充呢？答案是使用这个被删除节点的兄弟节点中的中间元素来进行替补



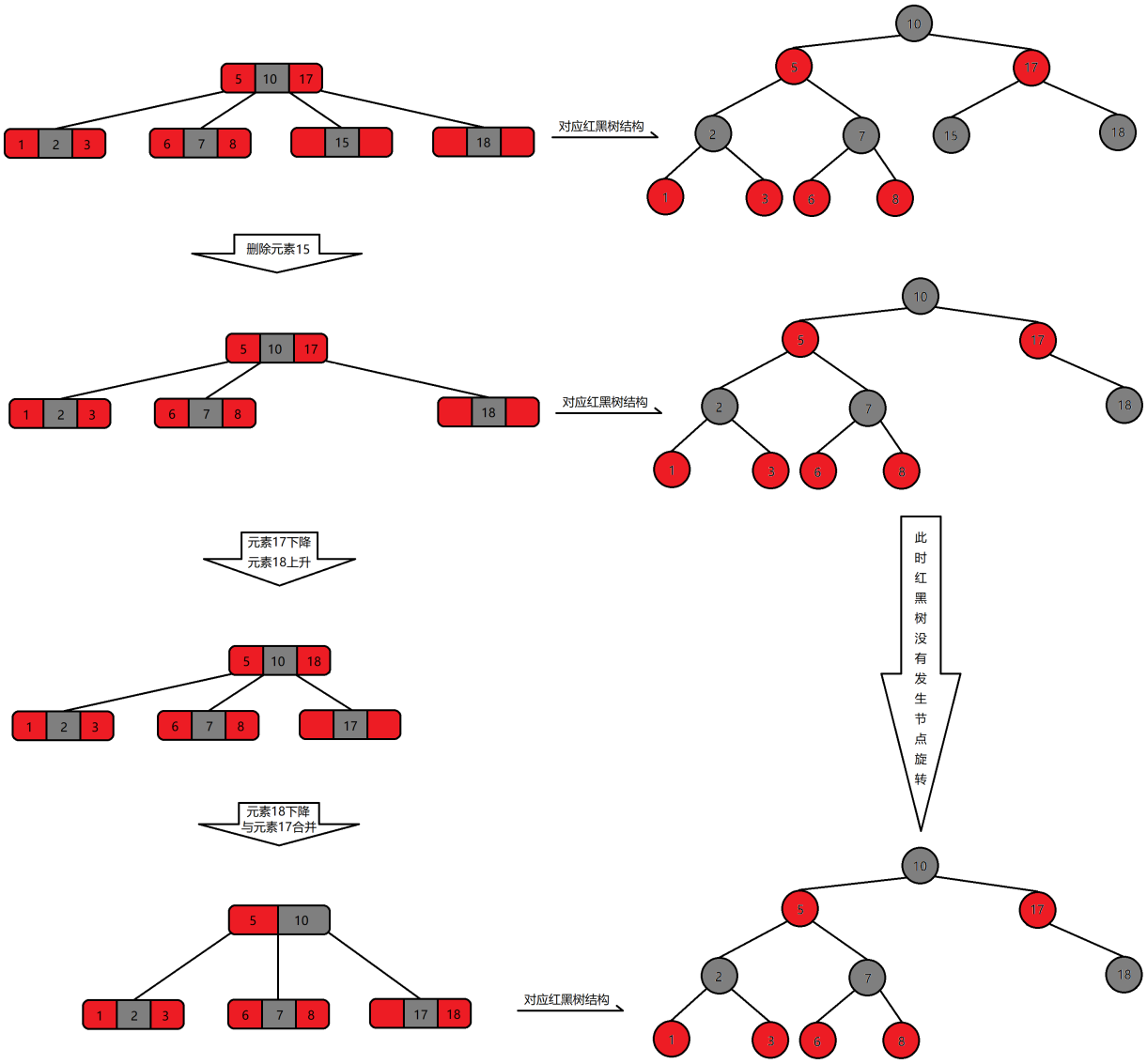
但是如果删除节点的兄弟节点中，仅存在一个中间节点怎么办？那岂不是这个兄弟节点相当于被删空了？

没错，也就是说，如果被删空的节点的兄弟节点只有一个中间元素，那么在使用这个中间元素替补下降的父节点元素之后

这个替补的元素同样要下降下来，否则其左右两边就会产生空位。下降下来的替补元素和之前的父节点元素融合，形成新的叶子节点

但是在这个过程中，红黑树的节点结构却没有发生任何变化，也就是说，在这个过程中，是不会伴随着红黑树节点的旋转和重新染色的

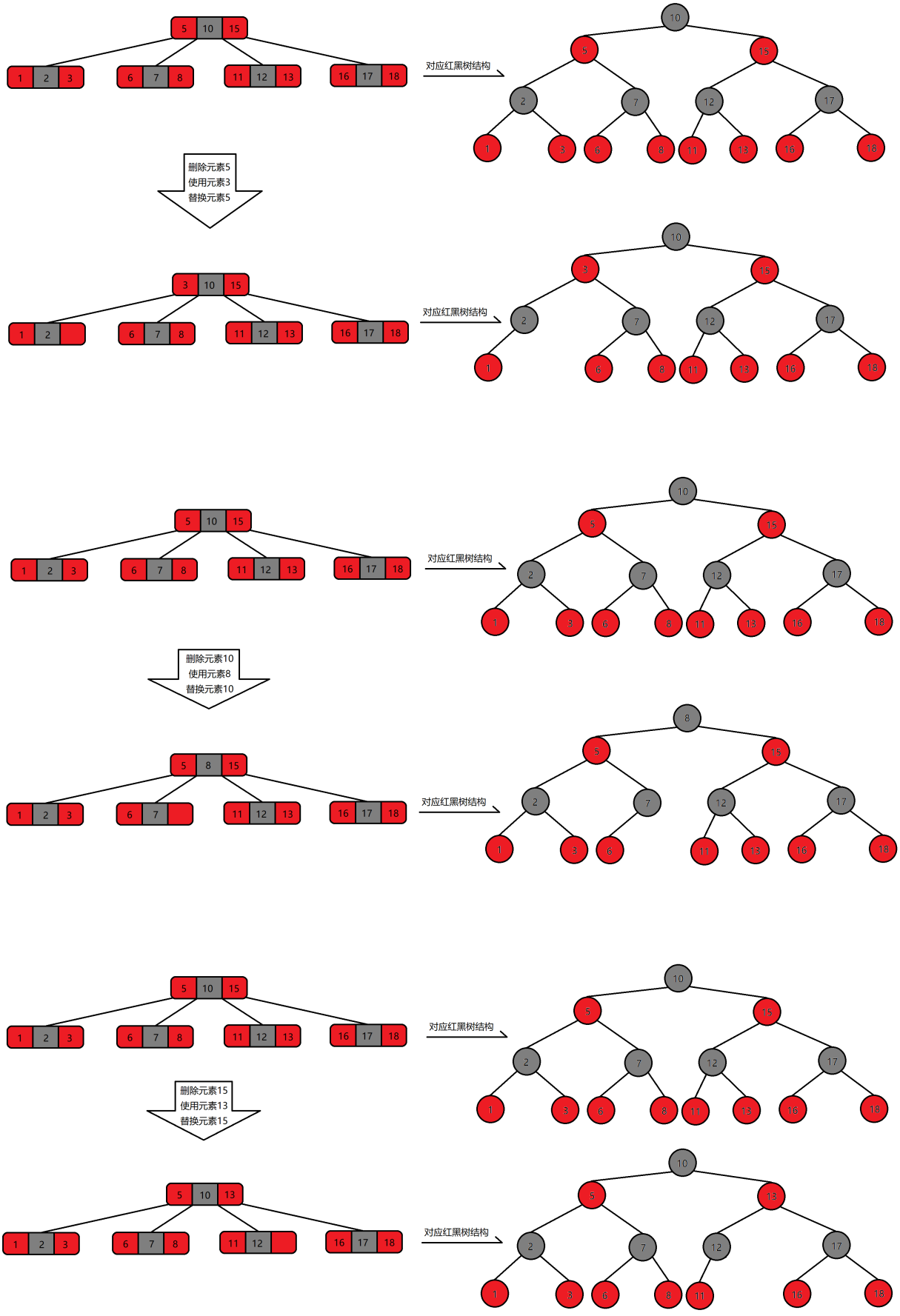
因为这种操作相当于删除一个倒数第二层红黑树节点的左孩子或者右孩子，正如前面所说的，这种操作并不会引起红黑树结构上的变化



2.删除2-3-4树父节点中元素的操作

删除2-3-4树父节点中元素的操作相对简单一些，我们只要将这个元素对应左子树中的最大值替换这个删除元素即可

2-3-4树左子树的最大值，对应红黑树中删除节点的左子树中最靠右下角的那个元素



但是需要注意的是，在具有祖父节点的情况下，删除2-3-4树结构中父节点的元素，同样需要考虑叶子结点元素的上升和重新排序问题

也就是说，删除节点，也同样可能导致红黑树节点的不平衡，进而导致旋转和重新染色的问题

最后，附赠大家一个能够通过图像化节点来验证红黑树的网址：

https://www.cs.usfca.edu/~galles/visualization/RedBlack.html（国外网址，有点儿卡，你们懂的……）