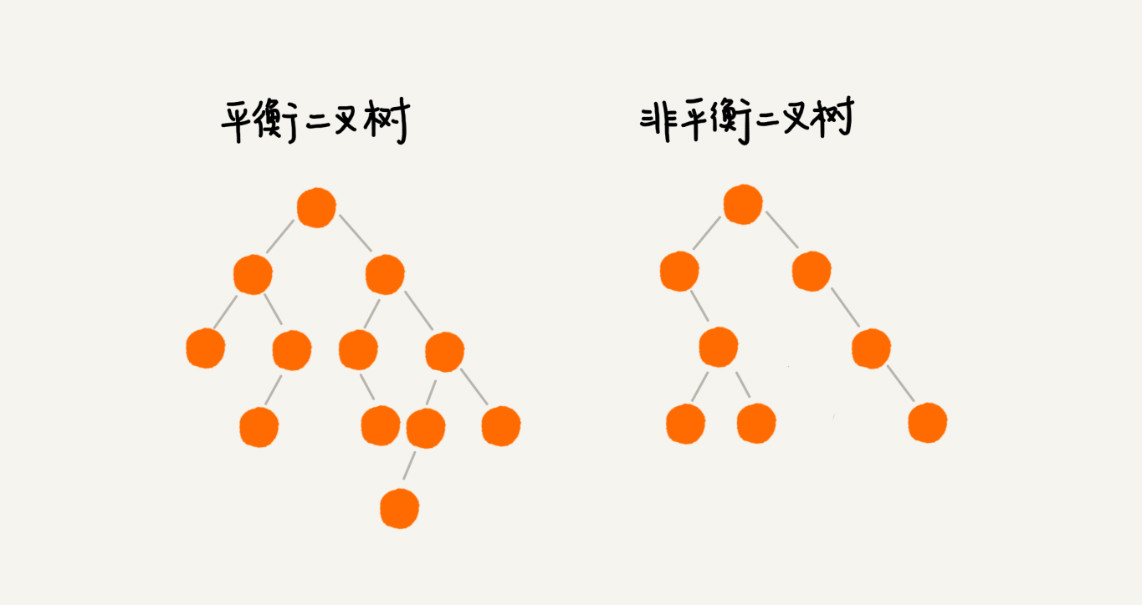
**一. 平衡二叉查找树**

要解决二叉查找树因为动态更新而造成复杂度退化的问题，我们需要设计一种平衡二叉查找树。

平衡二叉树的严格定义是这样的：二叉树中任意一个节点的左右子树的高度相差不能大于 1。从这个定义来看，上一节我们讲的完全二叉树、满二叉树其实都是平衡二叉树，但是非完全二叉树也有可能是平衡二叉树。



平衡二叉查找树不仅满足上面平衡二叉树的定义，还满足二叉查找树的特点。

平衡二叉查找树中“平衡”的意思，其实就是让整棵树左右看起来比较“对称”、比较“平衡”，不要出现左子树很高、右子树很矮的情况。这样就能让整棵树的高度相对来说低一些，相应的插入、删除、查找等操作的效率高一些。所以，如果我们现在设计一个新的平衡二叉查找树，只要树的高度不比 log2n 大很多（比如树的高度仍然是对数量级的），尽管它不符合我们前面讲的严格的平衡二叉查找树的定义，但我们仍然可以说，这是一个合格的平衡二叉查找树。

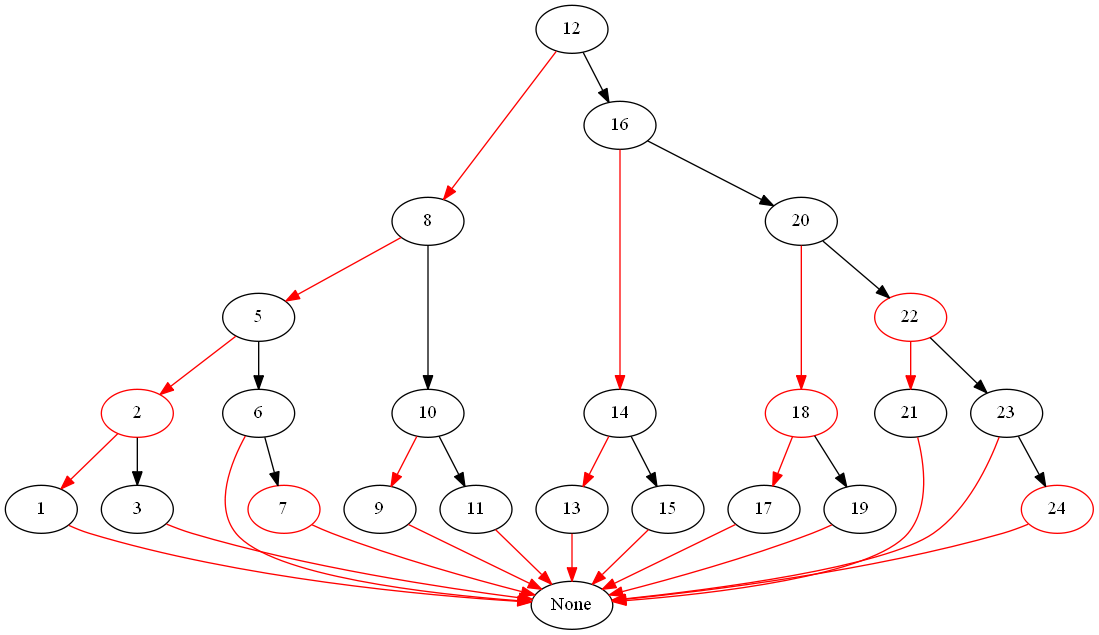
**二. 红黑树**

红黑树的英文是“Red-Black Tree”，简称 R-B Tree。它是一种不严格的平衡二叉查找树，它从根节点到各个叶子节点的最长路径，有可能会比最短路径大一倍。但他仍然是合格的平衡二叉查找树。

红黑树是为了解决普通二叉查找树在数据更新的过程中，复杂度退化的问题而产生的。红黑树的高度近似 log2n，所以它是近似平衡，插入、删除、查找操作的时间复杂度都是 O(logn)。

红黑树，顾名思义，其中的节点，一类被标记为黑色，一类被标记为红色。除此之外，一棵红黑树还需要满足这样几个要求：

* 根节点是黑色的；
* 每个叶子节点都是黑色的空节点（NIL），也就是说，叶子节点不存储数据；
* 任何相邻的节点都不能同时为红色，也就是说，红色节点是被黑色节点隔开的；
* 每个节点，从该节点到达其可达叶子节点的所有路径，都包含相同数目的黑色节点；



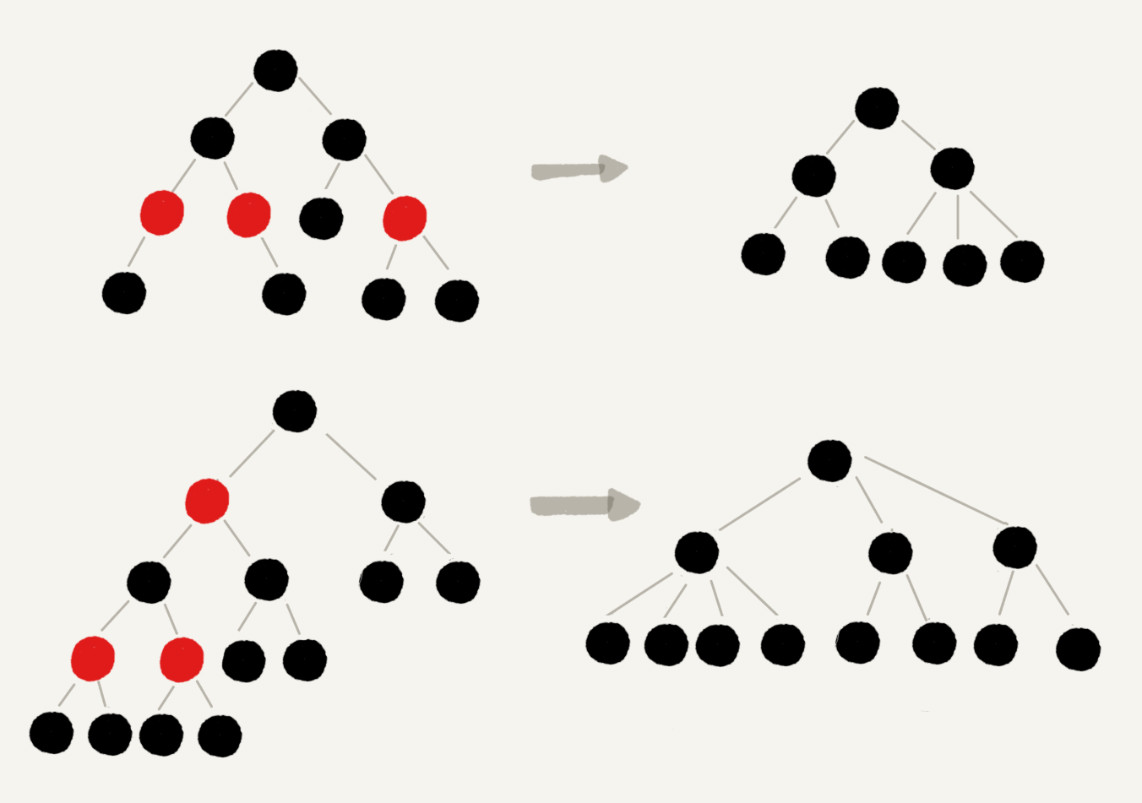
这里的第二点要求“叶子节点都是黑色的空节点”，稍微有些奇怪，它主要是为了简化红黑树的代码实现而设置的，下一节我们讲红黑树的实现的时候会讲到。这节我们暂时不考虑这一点，所以，在画图和讲解的时候，我将黑色的、空的叶子节点都省略掉了。

**三. 为什么说红黑树是“近似平衡“的？**

只需分析红黑树的高度是否比较稳定的接近log2n。

1. 首先，我们来看，如果我们将红色节点从红黑树中去掉，那单纯包含黑色节点的红黑树的高度是多少呢？

红色节点删除之后，有些节点就没有父节点了，它们会直接拿这些节点的祖父节点（父节点的父节点）作为父节点。所以，之前的二叉树就变成了四叉树。



前面红黑树的定义里有这么一条：从任意节点到可达的叶子节点的每个路径包含相同数目的黑色节点。我们从四叉树中取出某些节点，放到叶节点位置，四叉树就变成了完全二叉树。所以，仅包含黑色节点的四叉树的高度，比包含相同节点个数的完全二叉树的高度还要小。

完全二叉树的高度近似 log2n，这里的四叉“黑树”的高度要低于完全二叉树，所以去掉红色节点的“黑树”的高度也不会超过 log2n。

2. 我们现在知道只包含黑色节点的“黑树”的高度，那我们现在把红色节点加回去，高度会变成多少呢？

从上面我画的红黑树的例子和定义看，在红黑树中，红色节点不能相邻，也就是说，有一个红色节点就要至少有一个黑色节点，将它跟其他红色节点隔开。红黑树中包含最多黑色节点的路径不会超过 log2n，所以加入红色节点之后，最长路径不会超过 2log2n，也就是说，红黑树的高度近似 2log2n。

所以，红黑树的高度只比高度平衡的 AVL 树的高度（log2n）仅仅大了一倍，在性能上，下降得并不多。这样推导出来的结果不够精确，实际上红黑树的性能更好。

**四. 为什么工程中大家喜欢用红黑树这种平衡二叉查找树呢？**

我们前面提到 Treap、Splay Tree，绝大部分情况下，它们操作的效率都很高，但是也无法避免极端情况下时间复杂度的退化。尽管这种情况出现的概率不大，但是对于单次操作时间非常敏感的场景来说，它们并不适用。

AVL 树是一种高度平衡的二叉树，所以查找的效率非常高，但是，有利就有弊，AVL 树为了维持这种高度的平衡，就要付出更多的代价。每次插入、删除都要做调整，就比较复杂、耗时。所以，对于有频繁的插入、删除操作的数据集合，使用 AVL 树的代价就有点高了。

红黑树只是做到了近似平衡，并不是严格的平衡，所以在维护平衡的成本上，要比 AVL 树要低。所以，红黑树的插入、删除、查找各种操作性能都比较稳定。对于工程应用来说，要面对各种异常情况，为了支撑这种工业级的应用，我们更倾向于这种性能稳定的平衡二叉查找树。

**五. 总结一下支持插入、删除、查找操作的动态数据结构**

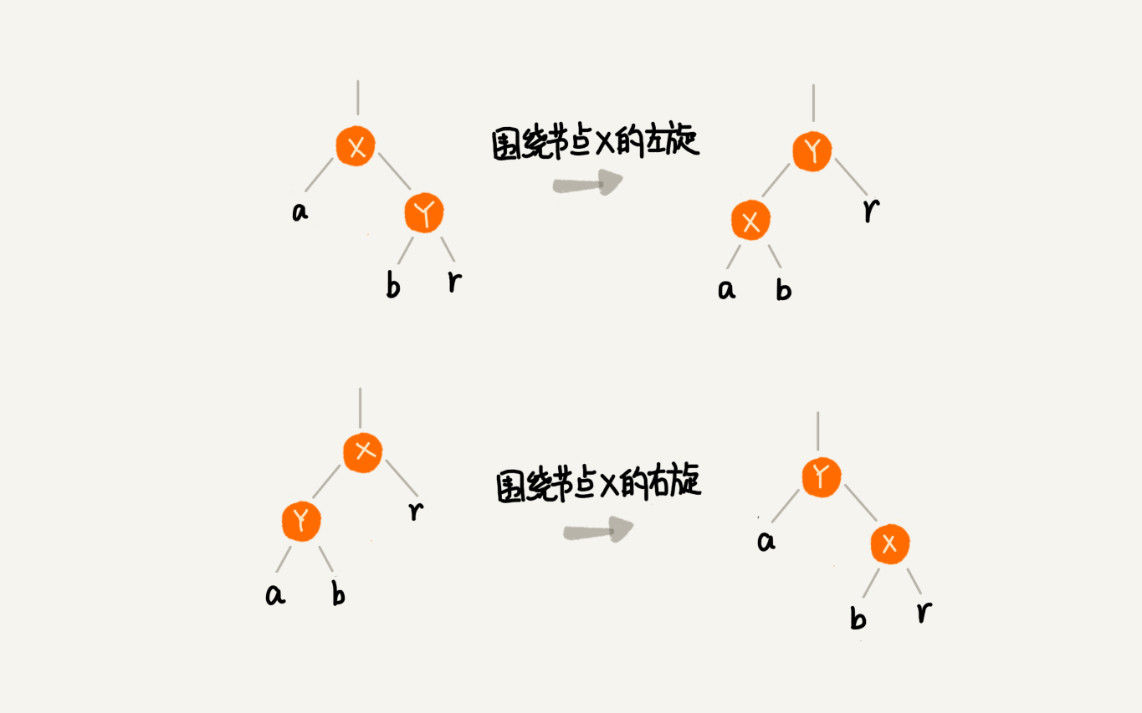
散列表：插入删除查找都是O(1), 是最常用的，但其缺点是不能顺序遍历以及扩容缩容的性能损耗。适用于那些不需要顺序遍历，数据更新不那么频繁的。

跳表：插入删除查找都是O(logn), 并且能顺序遍历。缺点是空间复杂度O(n)。适用于不那么在意内存空间的，其顺序遍历和区间查找非常方便。

红黑树：插入删除查找都是O(logn), 中序遍历即是顺序遍历，稳定。缺点是难以实现，去查找不方便。其实跳表更佳，但红黑树已经用于很多地方了。

**六. 红黑树平衡调整**

1. 左旋和右旋



2. 红黑树插入操作的平衡调整

红黑树规定，插入的节点必须是红色的。而且，二叉查找树中新插入的节点都是放在叶子节点上

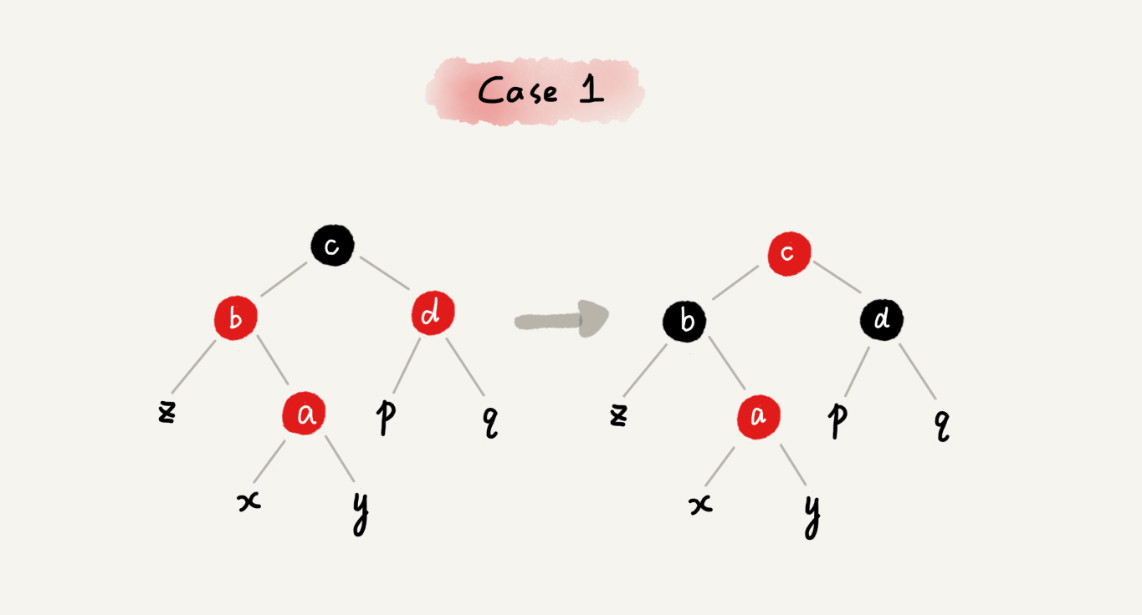
* 如果插入节点的父节点是黑色的，那我们什么都不用做，它仍然满足红黑树的定义。
* 如果插入的节点是根节点，那我们直接改变它的颜色，把它变成黑色就可以了。

除此以外，其他情况需要通过左右旋转和改变颜色来调整

Ps: 在实现代码中还要考虑关注节点的父节点是关注节点的祖父节点的左节点还是右节点，删除操作的第二次调整也同理分两类。

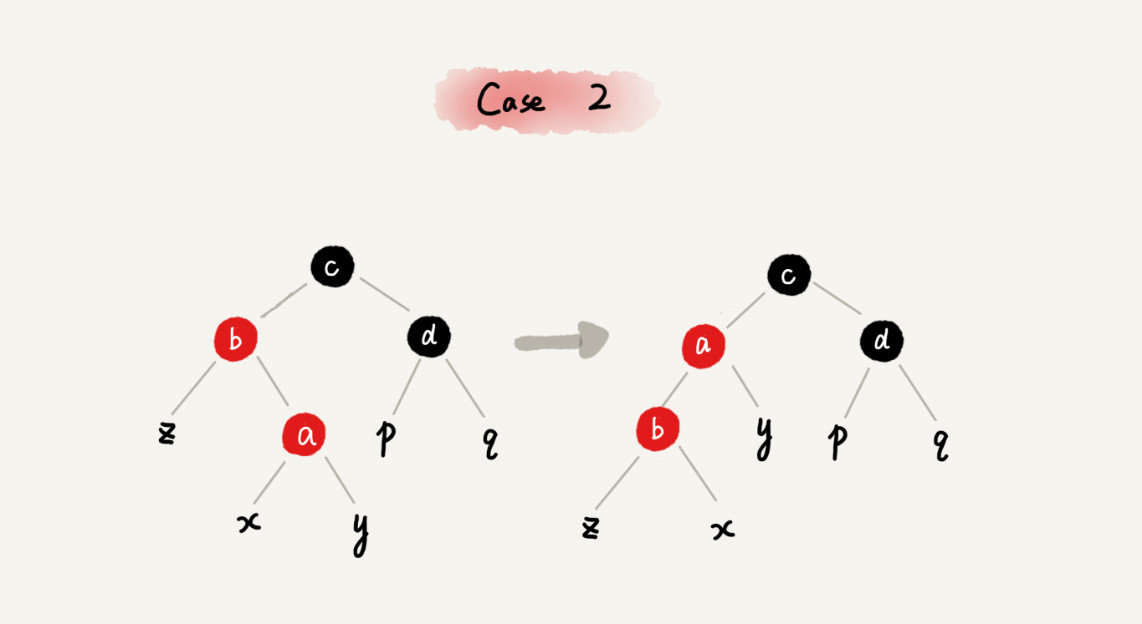
**CASE 1：如果关注节点是 a，它的叔叔节点 d 是红色，**我们就依次执行下面的操作：

* 将关注节点 a 的父节点 b、叔叔节点 d 的颜色都设置成黑色；
* 将关注节点 a 的祖父节点 c 的颜色设置成红色；
* 关注节点变成 a 的祖父节点 c；跳到 CASE 2 或者 CASE 3。



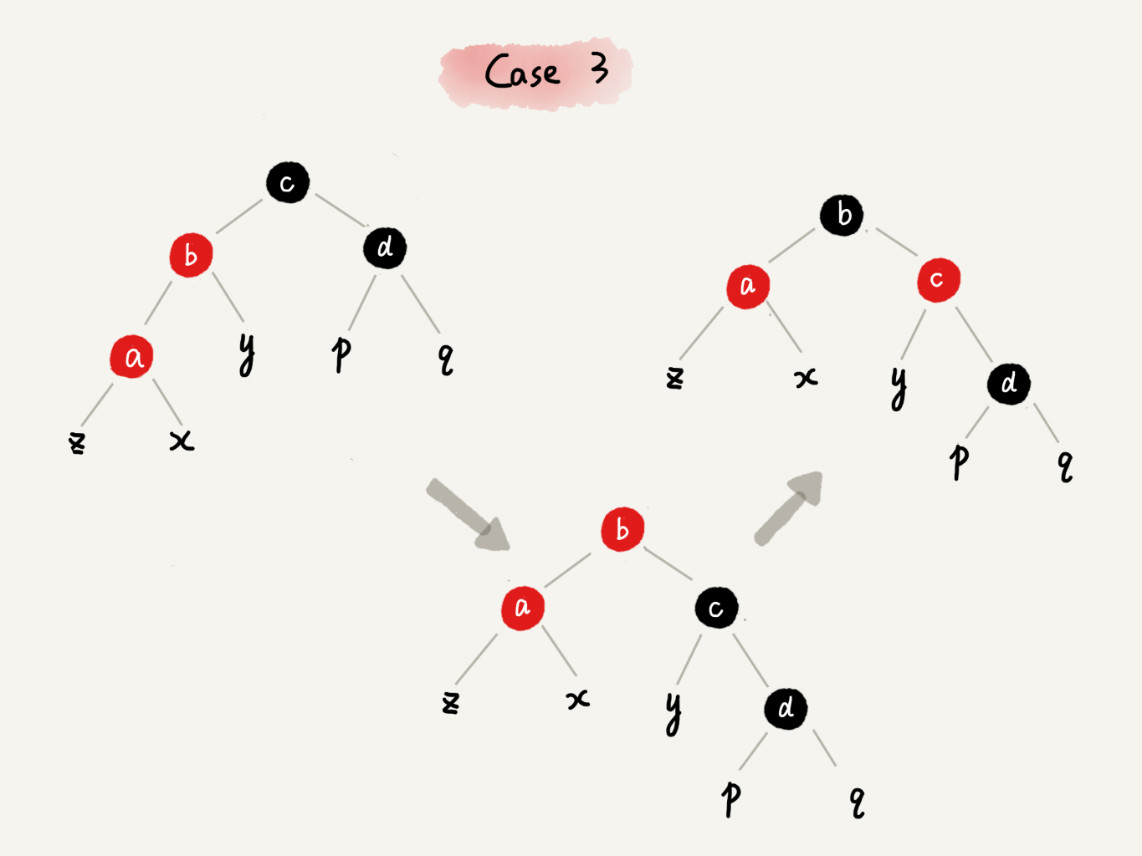
**CASE 2：如果关注节点是 a，它的叔叔节点 d 是黑色，关注节点 a 是其父节点 b 的右子节点，**我们就依次执行下面的操作：

* 关注节点变成节点 a 的父节点 b；
* 围绕新的关注节点b 左旋；
* 跳到 CASE 3。



**CASE 3：如果关注节点是 a，它的叔叔节点 d 是黑色，关注节点 a 是其父节点 b 的左子节点，**我们就依次执行下面的操作：

* 围绕关注节点 a 的祖父节点 c 右旋；
* 将关注节点 a 的父节点 b、兄弟节点 c 的颜色互换。
* 调整结束。



3. 红黑树删除操作的平衡调整

删除操作的平衡调整分为两步，第一步是针对删除节点初步调整。初步调整只是保证整棵红黑树在一个节点删除之后，仍然满足最后一条定义的要求，也就是说，每个节点，从该节点到达其可达叶子节点的所有路径，都包含相同数目的黑色节点；第二步是针对关注节点进行二次调整，让它满足红黑树的第三条定义，即不存在相邻的两个红色节点。

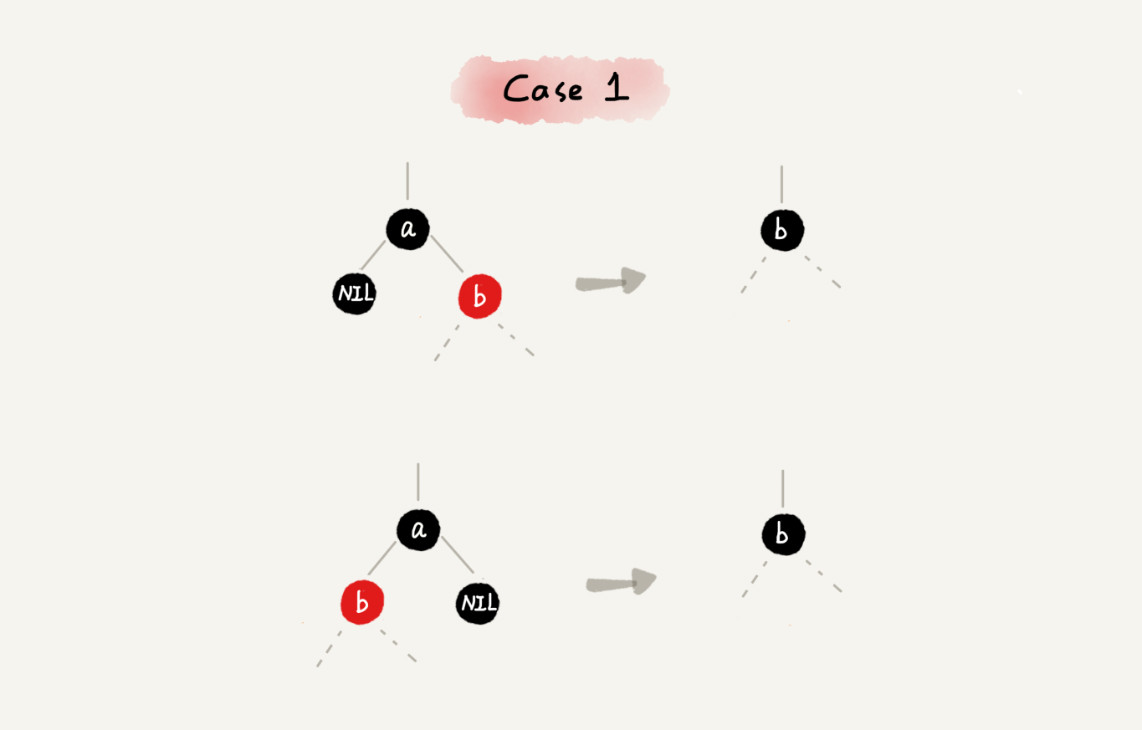
***针对删除节点初步调整：***

这里需要注意一下，红黑树的定义中“只包含红色节点和黑色节点”，经过初步调整之后，为了保证满足红黑树定义的最后一条要求，有些节点会被标记成两种颜色，“红 - 黑”或者“黑 - 黑”。如果一个节点被标记为了“黑 - 黑”，那在计算黑色节点个数的时候，要算成两个黑色节点。

在下面的讲解中，如果一个节点既可以是红色，也可以是黑色，在画图的时候，我会用一半红色一半黑色来表示。如果一个节点是“红 - 黑”或者“黑 - 黑”，我会用左上角的一个小黑点来表示额外的黑色。

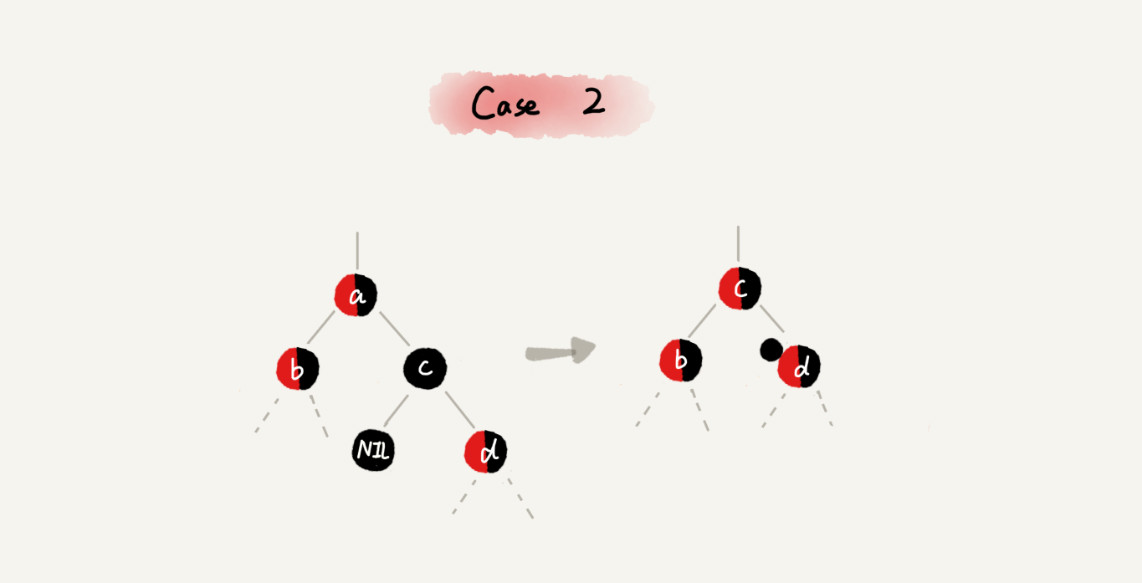
**CASE 1：如果要删除的节点是 a，它只有一个子节点 b**，那我们就依次进行下面的操作：

* 删除节点 a，并且把节点 b 替换到节点 a 的位置，这一部分操作跟普通的二叉查找树的删除操作一样；
* 节点 a 只能是黑色，节点 b 也只能是红色，其他情况均不符合红黑树的定义。这种情况下，我们把节点 b 改为黑色；
* 调整结束，不需要进行二次调整。



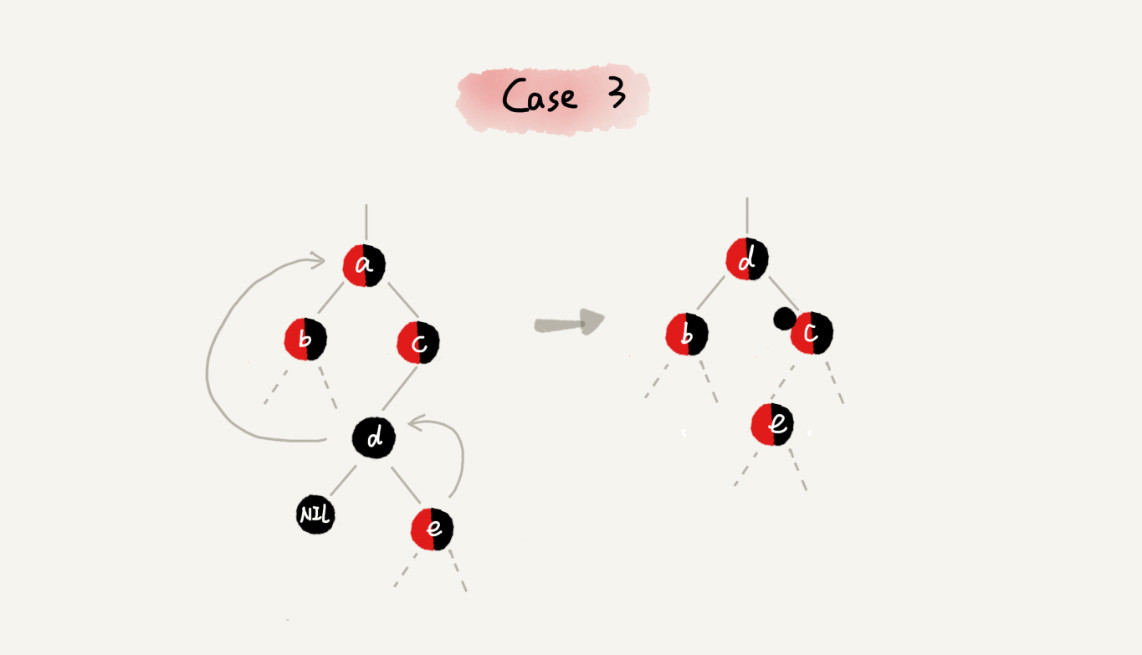
**CASE 2：如果要删除的节点 a 有两个非空子节点，并且它的后继节点就是节点 a 的右子节点 c。**我们就依次进行下面的操作：

* 如果节点 a 的后继节点就是右子节点 c，那右子节点 c 肯定没有左子树。我们把节点 a 删除，并且将节点 c 替换到节点 a 的位置。这一部分操作跟普通的二叉查找树的删除操作无异；
* 然后把节点 c 的颜色设置为跟节点 a 相同的颜色；
* 如果节点 c 是黑色，为了不违反红黑树的最后一条定义，我们给节点 c 的右子节点 d 多加一个黑色，这个时候节点 d 就成了“红 - 黑”或者“黑 - 黑”；
* 这个时候，关注节点变成了节点 d，第二步的调整操作就会针对关注节点来做。



**CASE 3：如果要删除的是节点 a，它有两个非空子节点，并且节点 a 的后继节点不是右子节点，**我们就依次进行下面的操作：

* 找到后继节点 d，并将它删除，删除后继节点 d 的过程参照 CASE 1；
* 将节点 a 替换成后继节点 d；把节点 d 的颜色设置为跟节点 a 相同的颜色；
* 如果节点 d 是黑色，为了不违反红黑树的最后一条定义，我们给节点 d 的右子节点 c 多加一个黑色，这个时候节点 c 就成了“红 - 黑”或者“黑 - 黑”；
* 这个时候，关注节点变成了节点 c，第二步的调整操作就会针对关注节点来做

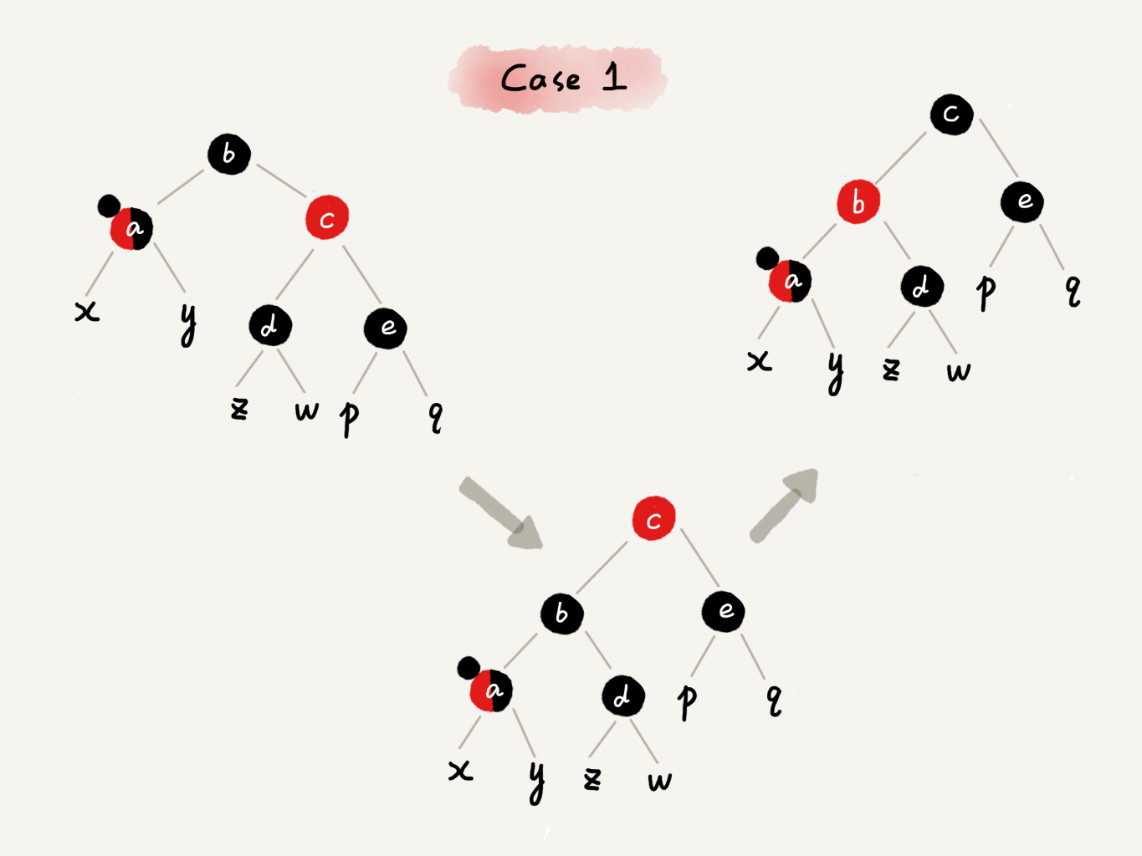


***针对关注节点进行二次调整：***

经过初步调整之后，关注节点变成了“红 - 黑”或者“黑 - 黑”节点。针对这个关注节点，我们再分四种情况来进行二次调整。二次调整是为了让红黑树中不存在相邻的红色节点。

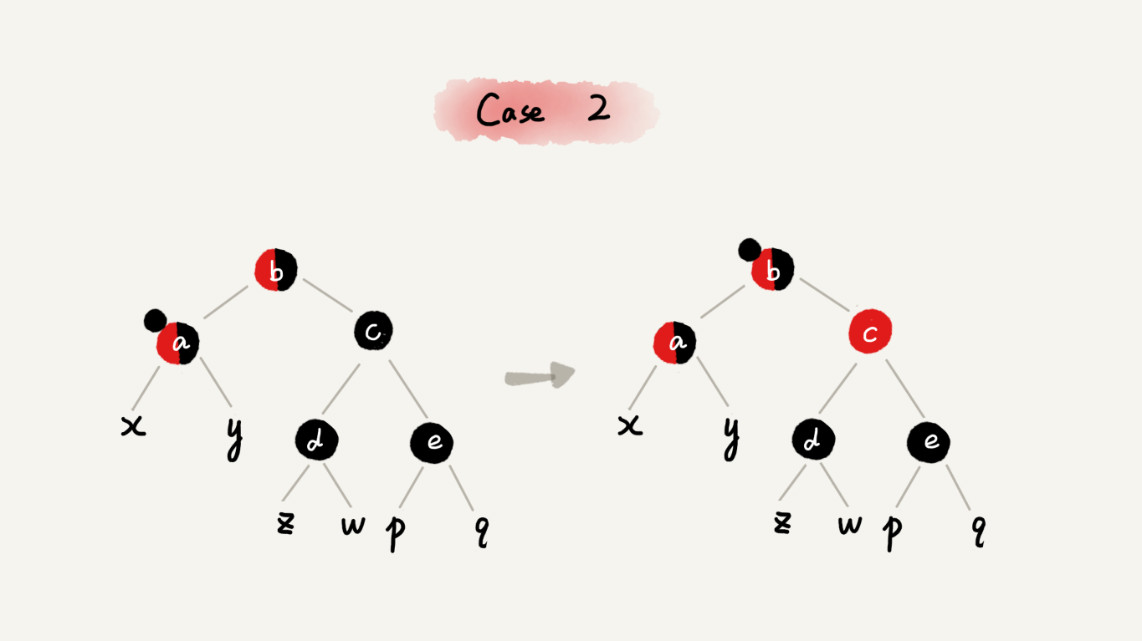
**CASE 1：如果关注节点是 a，它的兄弟节点 c 是红色的，**我们就依次进行下面的操作：

* 围绕关注节点 a 的父节点 b 左旋；
* 关注节点 a 的父节点 b 和祖父节点 c 交换颜色；
* 关注节点不变；
* 继续从四种情况中选择适合的规则来调整。



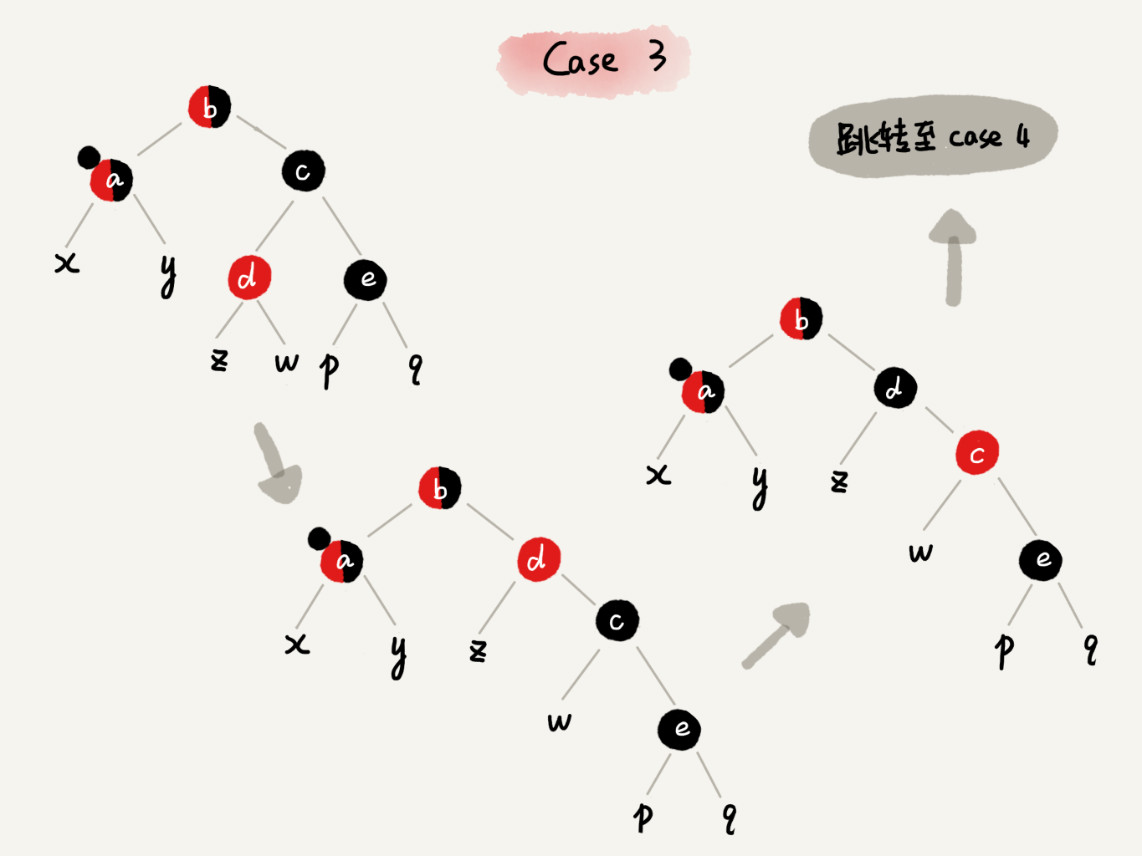
**CASE 2：如果关注节点是 a，它的兄弟节点 c 是黑色的，并且节点 c 的左右子节点 d、e 都是黑色的，**我们就依次进行下面的操作：

* 将关注节点 a 的兄弟节点 c 的颜色变成红色；
* 从关注节点 a 中去掉一个黑色，这个时候节点 a 就是单纯的红色或者黑色；
* 给关注节点 a 的父节点 b 添加一个黑色，这个时候节点 b 就变成了“红 - 黑”或者“黑 - 黑”；
* 关注节点从 a 变成其父节点 b；
* 继续从四种情况中选择符合的规则来调整。



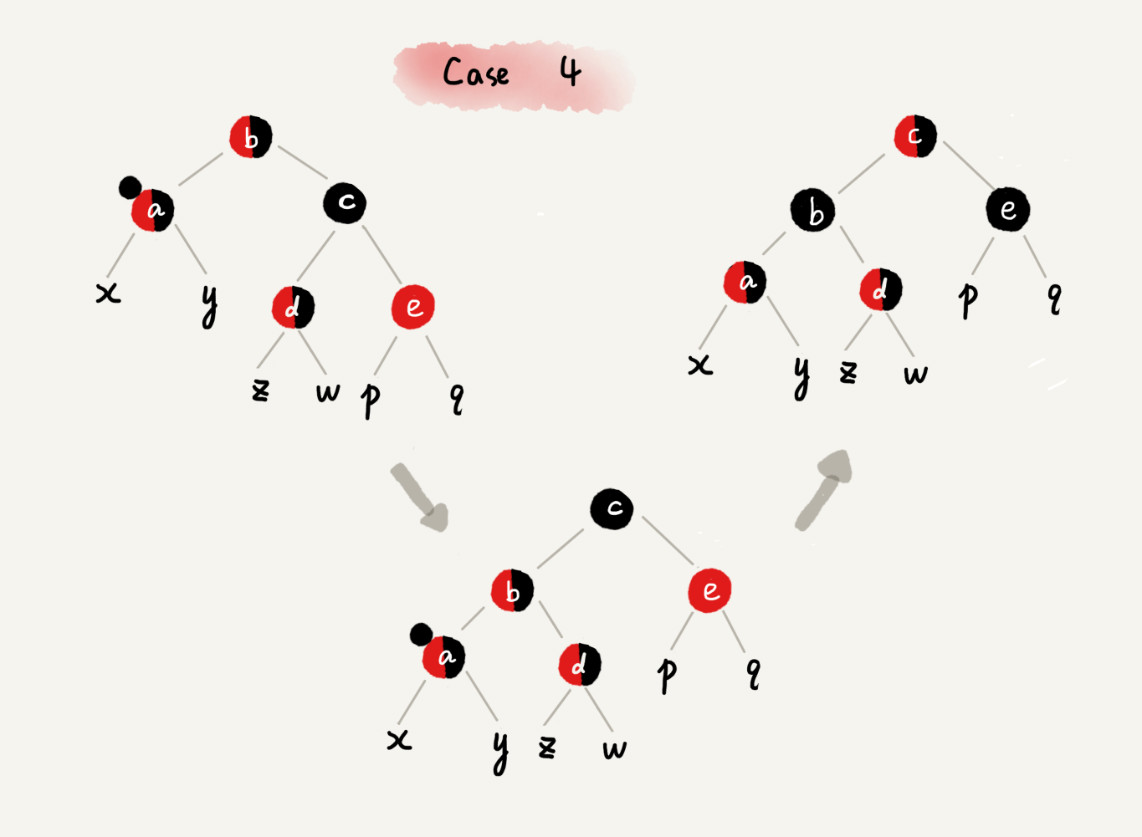
**CASE 3：如果关注节点是 a，它的兄弟节点 c 是黑色，c 的左子节点 d 是红色，c 的右子节点 e 是黑色，**我们就依次进行下面的操作：

* 围绕关注节点 a 的兄弟节点 c 右旋；
* 节点 c 和节点 d 交换颜色；
* 关注节点不变；
* 跳转到 CASE 4，继续调整。



**CASE 4：如果关注节点 a 的兄弟节点 c 是黑色的，并且 c 的右子节点是红色的，**我们就依次进行下面的操作：

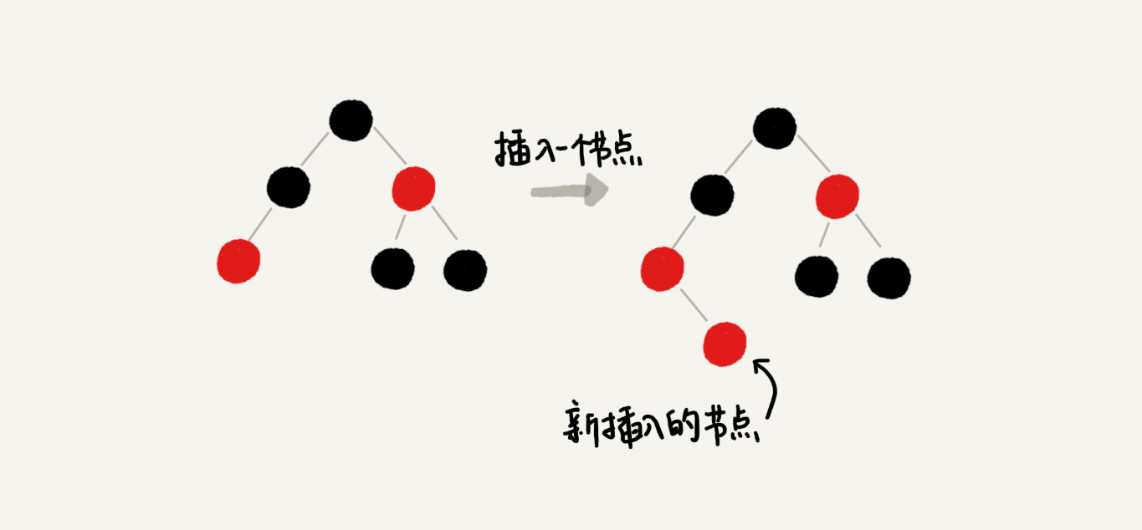
* 围绕关注节点 a 的父节点 b 左旋；
* 将关注节点 a 的兄弟节点 c 的颜色，跟关注节点 a 的父节点 b 设置成相同的颜色；
* 将关注节点 a 的父节点 b 的颜色设置为黑色；
* 从关注节点 a 中去掉一个黑色，节点 a 就变成了单纯的红色或者黑色；
* 将关注节点 a 的叔叔节点 e 设置为黑色；
* 调整结束。



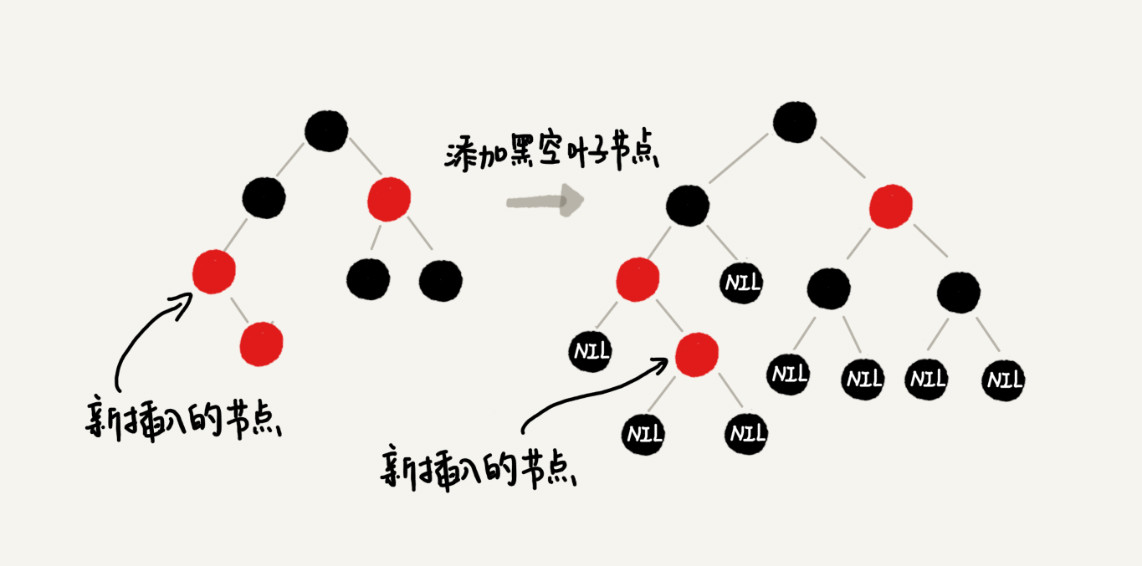
**六. 为什么红黑树的定义中，要求叶子节点是黑色的空节点**

只要满足这一条要求，那在任何时刻，红黑树的平衡操作都可以归结为我们刚刚讲的那几种情况。

假设红黑树的定义中不包含刚刚提到的那一条“叶子节点必须是黑色的空节点”，我们往一棵红黑树中插入一个数据，新插入节点的父节点也是红色的，两个红色的节点相邻，这个时候，红黑树的定义就被破坏了。那我们应该如何调整呢？



你会发现，这个时候，我们前面在讲插入时，三种情况下的平衡调整规则，没有一种是适用的。但是，如果我们把黑色的空节点都给它加上，变成下面这样，你会发现，它满足 CASE 2 了。



你可能还会说，这样给红黑树添加黑色的空的叶子节点，会不会比较浪费存储空间呢？答案是不会的。虽然我们在讲解或者画图的时候，每个黑色的、空的叶子节点都是独立画出来的。实际上，在具体实现的时候，我们只需要像下面这样，共用一个黑色的、空的叶子节点就行了。

