**红黑树删除**

红黑树的删除操作也包括两部分工作：一查找目标结点；而删除后自平衡。查找目标结点显然可以复用查找操作，当不存在目标结点时，忽略本次操作；当存在目标结点时，删除后就得做自平衡处理了。删除了结点后我们还需要找结点来替代删除结点的位置，不然子树跟父辈结点断开了，除非删除结点刚好没子结点，那么就不需要替代。

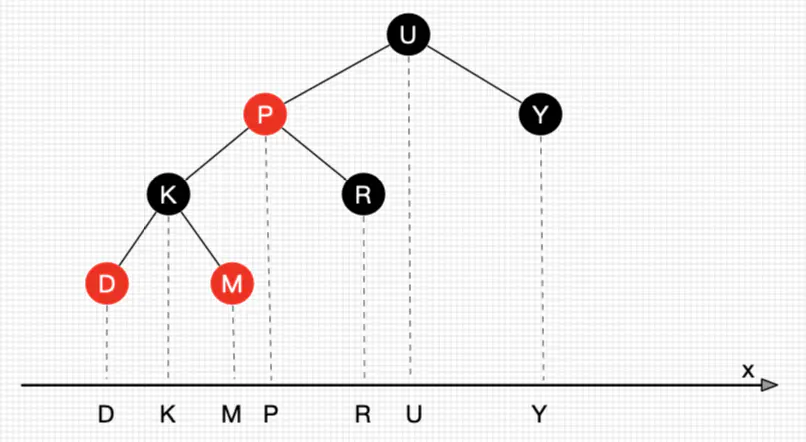
二叉树删除结点找替代结点有3种情情景：

情景1：若删除结点无子结点，直接删除

情景2：若删除结点只有一个子结点，用子结点替换删除结点

情景3：若删除结点有两个子结点，用后继结点（大于删除结点的最小结点）替换删除结点

补充说明下，情景3的后继结点是大于删除结点的最小结点，也是删除结点的右子树种最左结点。那么可以拿前继结点（删除结点的左子树最右结点）替代吗？可以的。但习惯上大多都是拿后继结点来替代，后文的讲解也是用后继结点来替代。另外告诉大家一种找前继和后继结点的直观的方法：把二叉树所有结点投射在X轴上，所有结点都是从左到右排好序的，所有目标结点的前后结点就是对应前继和后继结点。



接下来，讲一个重要的思路：删除结点被替代后，在不考虑结点的键值的情况下，对于树来说，可以认为删除的是替代结点！话很苍白，我们看图17。在不看键值对的情况下，图17的红黑树最终结果是删除了Q所在位置的结点！这种思路非常重要，大大简化了后文讲解红黑树删除的情景！

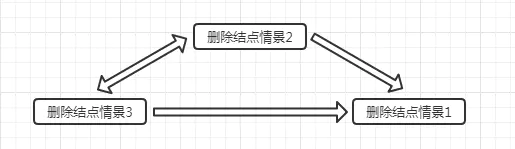


基于此，上面所说的3种二叉树的删除情景可以相互转换并且最终都是转换为情景1！

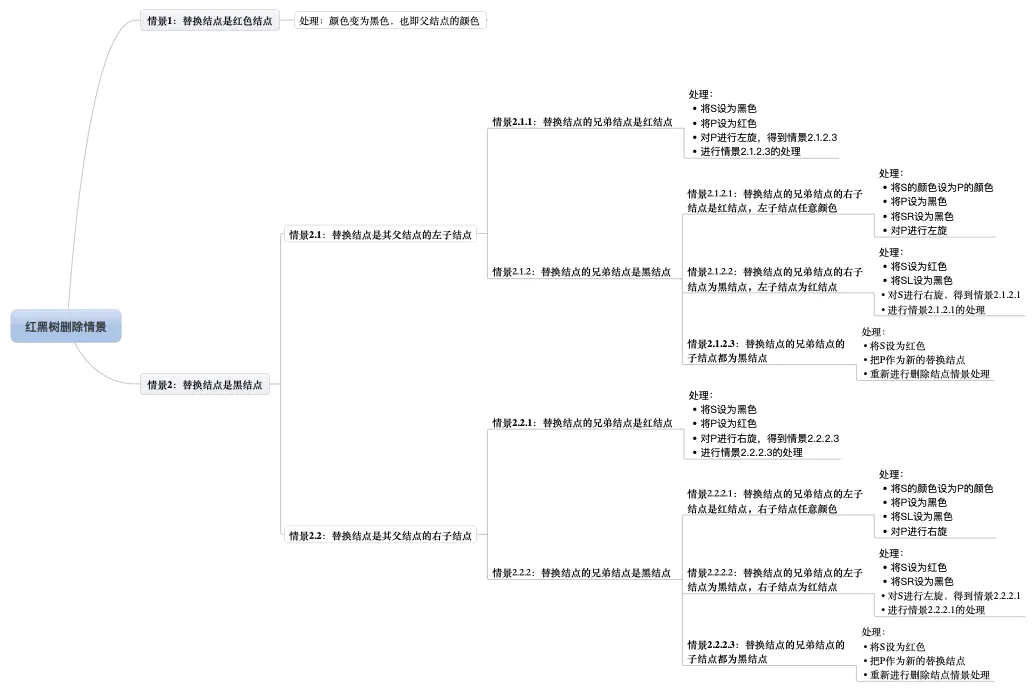
情景2：删除结点用其唯一的子结点替换，子结点替换为删除结点后，可以认为删除的是子结点，若子结点又有两个子结点，那么相当于转换为情景3，一直自顶向下转换，总是能转换为情景1。（对于红黑树来说，根据性质5.1，只存在一个子结点的结点肯定在树末了）

情景3：删除结点用后继结点（肯定不存在左结点），如果后继结点有右子结点，那么相当于转换为情景2，否则转为为情景1。

二叉树删除结点情景关系图如图18所示。



综上所述，删除操作删除的结点可以看作删除替代结点，而替代结点最后总是在树末。有了这结论，我们讨论的删除红黑树的情景就少了很多，因为我们只考虑删除树末结点的情景了。



删除情景1：替换结点是红色结点

我们把替换结点换到了删除结点的位置时，由于替换结点时红色，删除也了不会影响红黑树的平衡，只要把替换结点的颜色设为删除的结点的颜色即可重新平衡。

处理：颜色变为删除结点的颜色

删除情景2：替换结点是黑结点

当替换结点是黑色时，我们就不得不进行自平衡处理了。我们必须还得考虑替换结点是其父结点的左子结点还是右子结点，来做不同的旋转操作，使树重新平衡。

删除情景2.1：替换结点是其父结点的左子结点

删除情景2.1.1：替换结点的兄弟结点是红结点

若兄弟结点是红结点，那么根据性质4，兄弟结点的父结点和子结点肯定为黑色，不会有其他子情景，我们按图21处理，得到删除情景2.1.2.3（后续讲解，这里先记住，此时R仍然是替代结点，它的新的兄弟结点SL和兄弟结点的子结点都是黑色）。

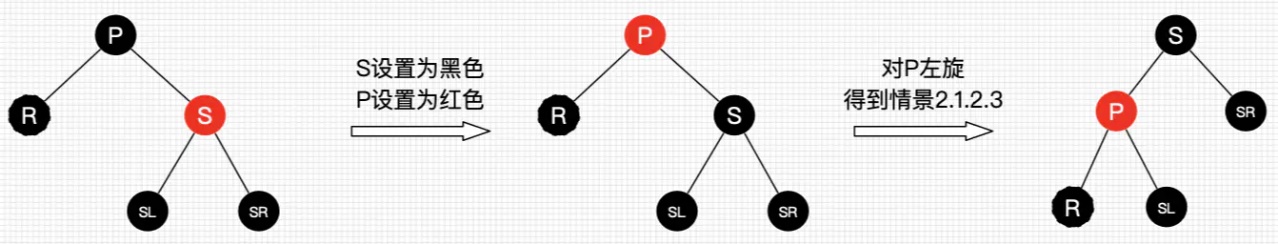
处理：

将S设为黑色

将P设为红色

对P进行左旋，得到情景2.1.2.3

进行情景2.1.2.3的处理



删除情景2.1.2：替换结点的兄弟结点是黑结点

当兄弟结点为黑时，其父结点和子结点的具体颜色也无法确定（如果也不考虑自底向上的情况，子结点非红即为叶子结点Nil，Nil结点为黑结点），此时又得考虑多种子情景。

删除情景2.1.2.1：替换结点的兄弟结点的右子结点是红结点，左子结点任意颜色

即将删除的左子树的一个黑色结点，显然左子树的黑色结点少1了，然而右子树又又红色结点，那么我们直接向右子树“借”个红结点来补充黑结点就好啦，此时肯定需要用旋转处理了。如图22所示。

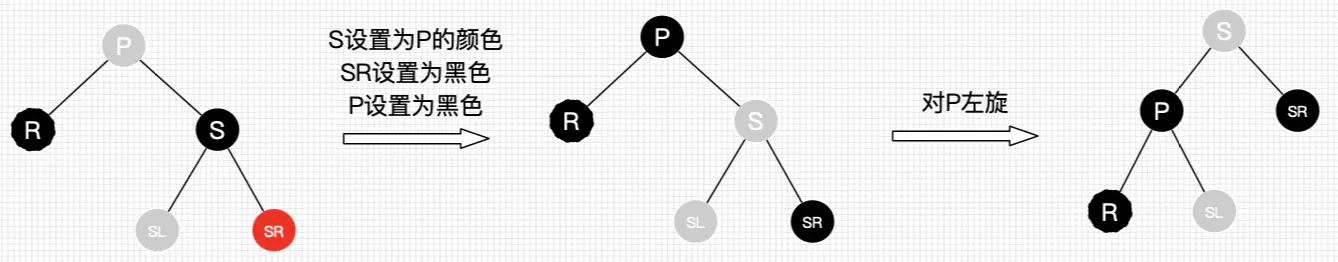
处理：

将S的颜色设为P的颜色

将P设为黑色

将SR设为黑色

对P进行左旋



平衡后的图怎么不满足红黑树的性质？前文提醒过，R是即将替换的，它还参与树的自平衡，平衡后再替换到删除结点的位置，所以R最终可以看作是删除的。另外图2.1.2.1是考虑到第一次替换和自底向上处理的情况，如果只考虑第一次替换的情况，根据红黑树性质，SL肯定是红色或为Nil，所以最终结果树是平衡的。如果是自底向上处理的情况，同样，每棵子树都保持平衡状态，最终整棵树肯定是平衡的。后续的情景同理，不做过多说明了。

删除情景2.1.2.2：替换结点的兄弟结点的右子结点为黑结点，左子结点为红结点

兄弟结点所在的子树有红结点，我们总是可以向兄弟子树借个红结点过来，显然该情景可以转换为情景2.1.2.1。图如23所示。

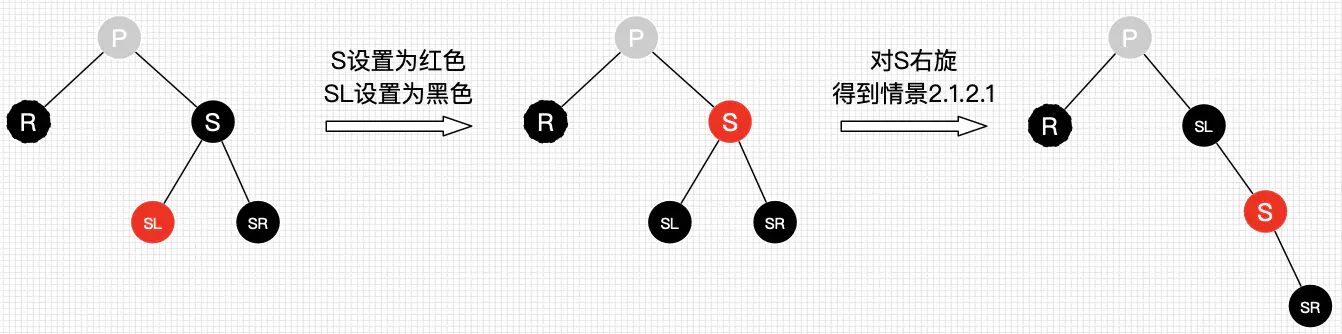
处理：

将S设为红色

将SL设为黑色

对S进行右旋，得到情景2.1.2.1

进行情景2.1.2.1的处理



删除情景2.1.2.3：替换结点的兄弟结点的子结点都为黑结点

好了，此次兄弟子树都没红结点“借”了，兄弟帮忙不了，找父母呗，这种情景我们把兄弟结点设为红色，再把父结点当作替代结点，自底向上处理，去找父结点的兄弟结点去“借”。但为什么需要把兄弟结点设为红色呢？显然是为了在P所在的子树中保证平衡（R即将删除，少了一个黑色结点，子树也需要少一个），后续的平衡工作交给父辈们考虑了，还是那句，当每棵子树都保持平衡时，最终整棵总是平衡的。

处理：

将S设为红色

把P作为新的替换结点

重新进行删除结点情景处理



删除情景2.2：替换结点是其父结点的右子结点

好啦，右边的操作也是方向相反，不做过多说明了，相信理解了删除情景2.1后，肯定可以理解2.2。

删除情景2.2.1：替换结点的兄弟结点是红结点

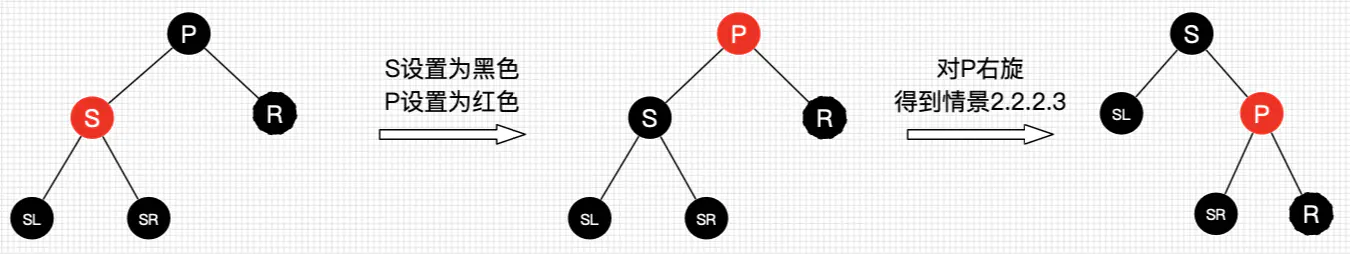
处理：

将S设为黑色

将P设为红色

对P进行右旋，得到情景2.2.2.3

进行情景2.2.2.3的处理



删除情景2.2.2：替换结点的兄弟结点是黑结点

删除情景2.2.2.1：替换结点的兄弟结点的左子结点是红结点，右子结点任意颜色

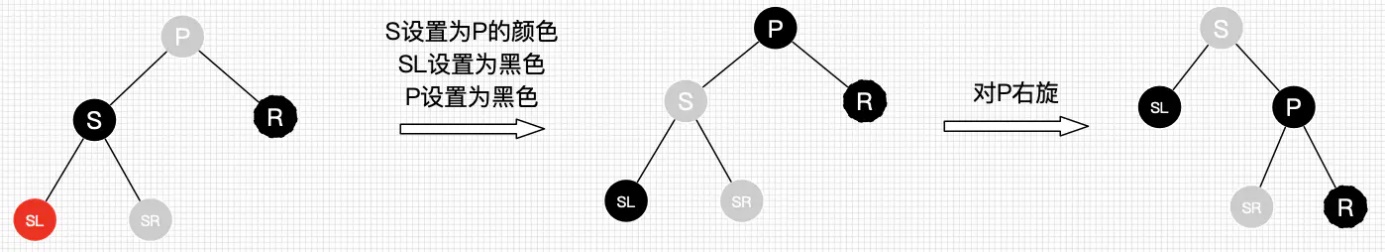
处理：

将S的颜色设为P的颜色

将P设为黑色

将SL设为黑色

对P进行右旋



删除情景2.2.2.2：替换结点的兄弟结点的左子结点为黑结点，右子结点为红结点

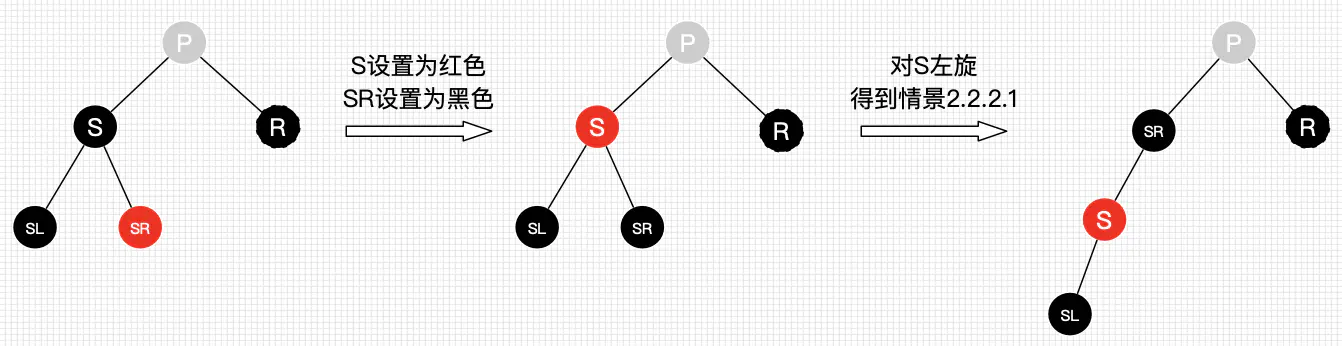
处理：

将S设为红色

将SR设为黑色

对S进行左旋，得到情景2.2.2.1

进行情景2.2.2.1的处理



删除情景2.2.2.3：替换结点的兄弟结点的子结点都为黑结点

处理：

将S设为红色

把P作为新的替换结点

重新进行删除结点情景处理

