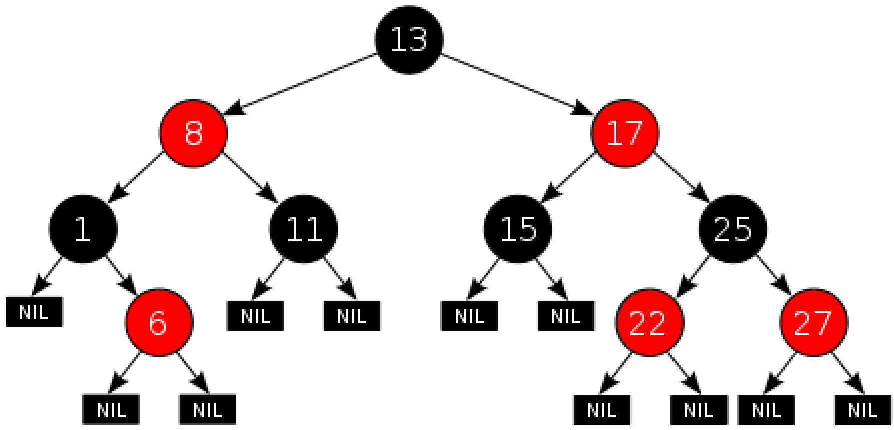
红黑树并没有我们想象的那么难(上)

红黑树并没有我们想象的那么难 上、下两篇已经完成, 希望能帮助到大家.

- 红黑树并没有我们想象的那么难(上): http://daoluan.net/blog/rbtree-is-not-difficult/
- 红黑树并没有我们想象的那么难(下): http://daoluan.net/blog/rbtree-is-not-difficult-2/

红黑树并没有想象的那么难, 初学者觉得晦涩难读可能是因为情况太多. 红黑树的情况可以通过归结, 通过合并来得到更少的情况, 如此可以加深对红黑树的理解. 网络上的大部分红黑树的讲解因为没有「合并」. 红黑树的五个性质:



性质1. 节点是红色或黑色。

性质2. 根是黑色。

性质3. 所有叶子都是黑色（叶子是NIL节点）。

性质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)

性质5. 从任一节点到其每个叶子的所有简单路径 都包含相同数目的黑色节点。

红黑树的数据结构

摘自 sgi stl 红黑树数据结构定义:

```
typedef bool _Rb_tree_Color_type;
const _Rb_tree_Color_type _S_rb_tree_red = false;
const _Rb_tree_Color_type _S_rb_tree_black = true;

struct _Rb_tree_node_base
{
  typedef _Rb_tree_Color_type _Color_type;
  typedef _Rb_tree_node_base* _Base_ptr;

  _Color_type _M_color;
```

```
  _Base_ptr _M_parent;
  _Base_ptr _M_left;
  _Base_ptr _M_right;

  static _Base_ptr _S_minimum(_Base_ptr __x)
  {
    while (__x->_M_left != 0) __x = __x->_M_left;
    return __x;
  }

  static _Base_ptr _S_maximum(_Base_ptr __x)
  {
    while (__x->_M_right != 0) __x = __x->_M_right;
    return __x;
  }
};

template <class _Value>
struct _Rb_tree_node : public _Rb_tree_node_base
{
  typedef _Rb_tree_node<_Value>* _Link_type;
  _Value _M_value_field;
};
```

二叉搜索树的插入删除操作

在展开红黑树之前, 首先来看看普通二叉搜索树的插入和删除. 插入很容易理解, 比当前值大就往右走, 比当前值小就往左走. 详细展开的是删除操作.

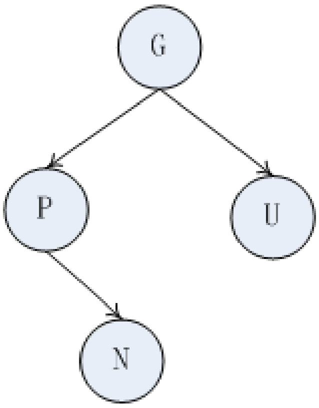
二叉树的删除操作有一个技巧, 即在查找到需要删除的节点 X; 接着我们找到要么在它的左子树中的最大元素节点 M、要么在它的右子树中的最小元素节点 M, 并交换(M,X). 此时, M 节点必然至多只有一个孩子; 最后一个步骤就是用 M 的子节点代替 M 节点就完成了. 所以, 所有的删除操作最后都会归结为删除一个至多只有一个孩子的节点, 而我们删除这个节点后, 用它的孩子替换就好了. 将会看到 sgi stl map 就是这样的策略.

在红黑树删除操作讲解中, 我们假设代替 M 的节点是 N(下面的讲述不再出现 M).

红黑树的插入

插入新节点总是红色节点, 因为不会破坏性质 5, 尽可能维持所有性质.

假设, 新插入的节点为 N, N 节点的父节点为 P, P 的兄弟(N 的叔父)节点为 U, P 的父亲(N 的爷爷)节点为 G. 所以有如下的印象图:



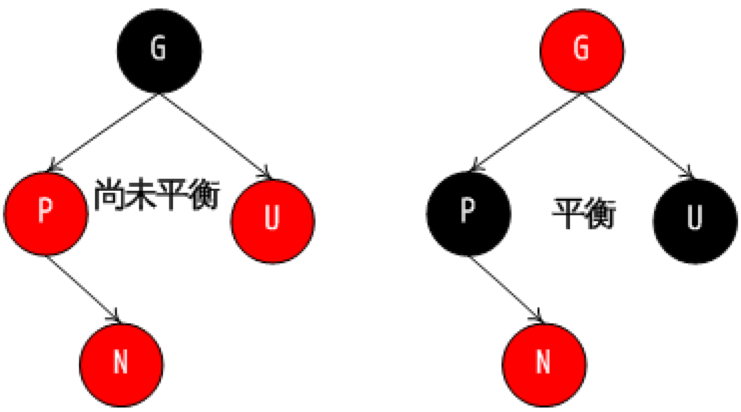
插入节点的关键是:

1. 插入新节点总是红色节点
2. 如果插入节点的父节点是黑色, 能维持性质
3. 如果插入节点的父节点是红色, 破坏了性质. 故插入算法就是通过重新着色或旋转, 来维持性质

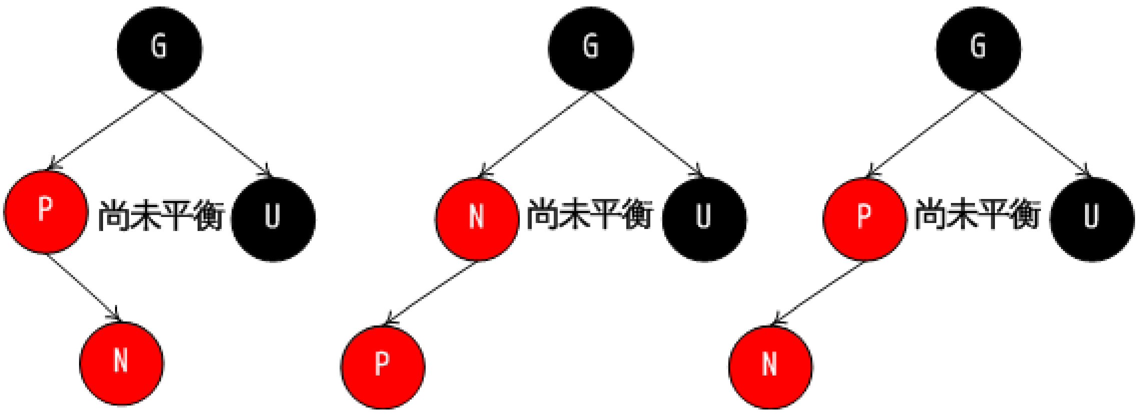
插入算法详解如下, 走一遍红黑树维持其性质的过程:

第 0.0 种情况, N 为根节点, 直接 N->黑. over 第 0.1 种情况, N 的父节点为黑色, 这不违反红黑树的五种性质. over

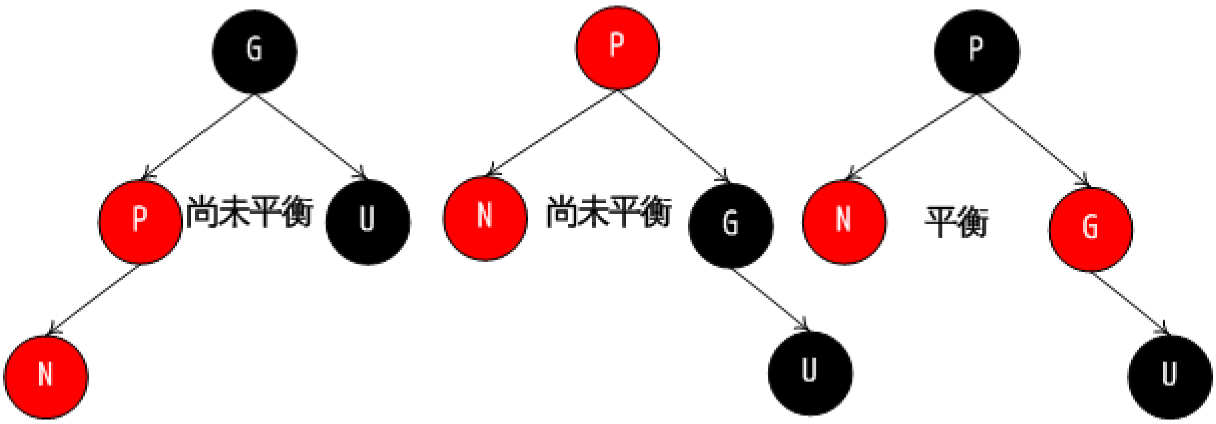
第 1 种情况, N,P,U 都红(G 肯定黑). 策略: G->红, N,P->黑. 此时, G 红, 如果 G 的父亲也是红, 性质又被破坏了, HACK: 可以将 GPUN 看成一个新的红色 N 节点, 如此递归调整下去; 特俗的, 如果碰巧将根节点染成了红色, 可以在算法的最后强制 root->黑.



第 2 种情况, P 为红, N 为 P 右孩子, N 为红, U 为黑或缺少. 策略: 旋转变换, 从而进入下一种情况:



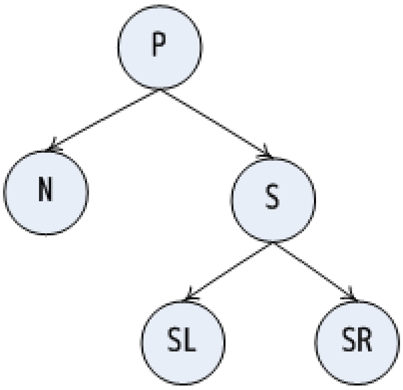
第 3 种情况, 可能由第二种变化而来, 但不是一定: P 为红, N 为 P 左孩子, N 为红.
策略: 旋转, 交换 P,G 颜色, 调整后, 因为 P 为黑色, 所以不怕 P 的父节点是红色的情况. over



红黑树的插入就为上面的三种情况. 你可以做镜像变换从而得到其他的情况.

红黑树的删除

假设 N 节点见上面普通二叉树删除中的定义, P 为 N 父节点, S 为 N 的兄弟节点, SL,SR 分别是 S 的左右子节点. 有如下印象图:



N 没有任何的孩子!

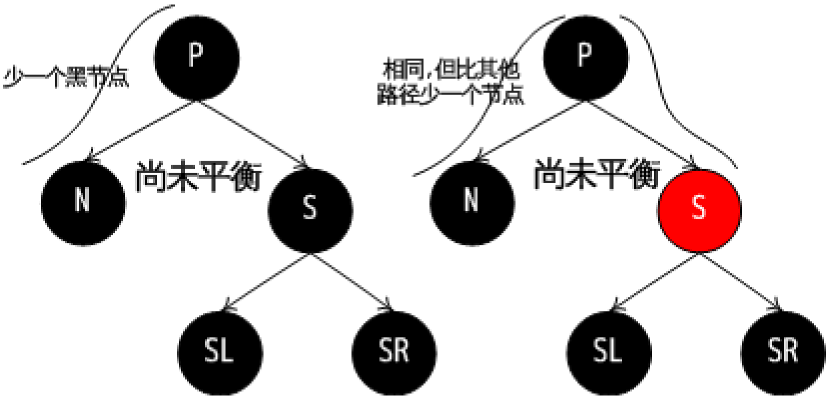
删除节点的关键是:

1. 如果删除的是红色节点, 不破坏性质
2. 如果删除的是黑色节点, 那么这个路径上就会少一个黑色节点, 破坏了性质. 故删除算法就是通过重新着色或旋转, 来维持性质

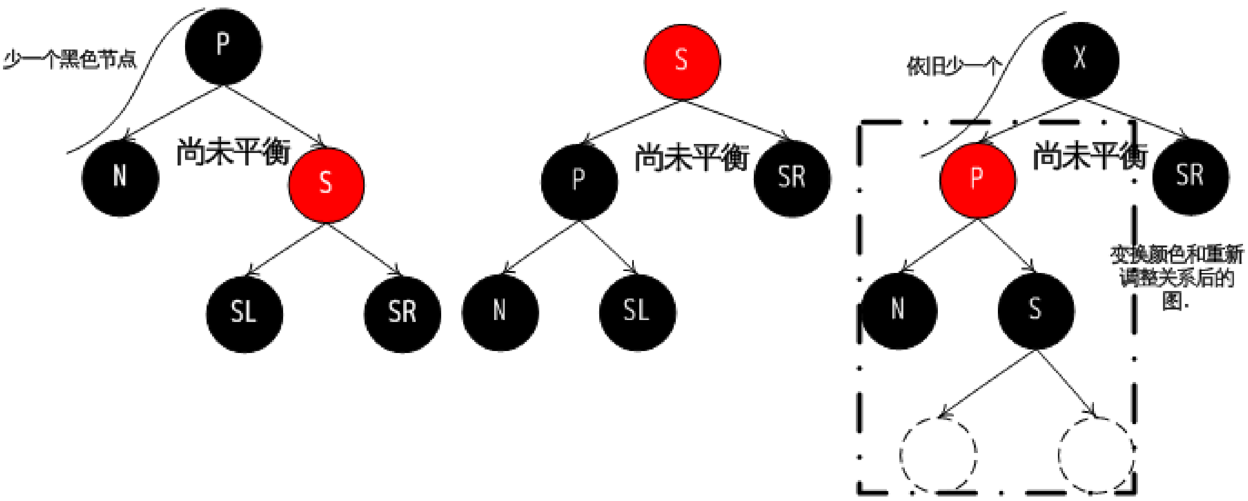
删除算法详解如下, 走一遍红黑树维持其性质的过程:

第 0.0 情况, N 为根节点. over 第 0.1 情况, 删除的节点为红. over 第 0.2 情况, 删除节点为黑, N 为红. **策略: N->黑, 重新平衡.** over

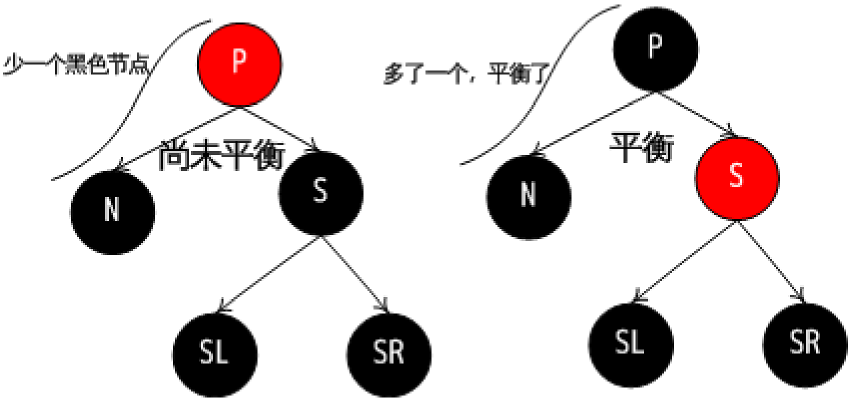
第 1 情况, N,P,S,SR,SL 都黑. **策略: S->红.** 通过 PN,PS 的黑色节点数量相同了, 但会比其他路径多一个, 解决的方法是在 P 上从情况 0 开始继续调整. 为什么要这样呢? HANKS: 因为既然 PN,PS 路径上的黑节点数量相同而且比其他路径会少一个黑节点, 那何不将其整体看成了一个 N 节点! 这是递归原理.



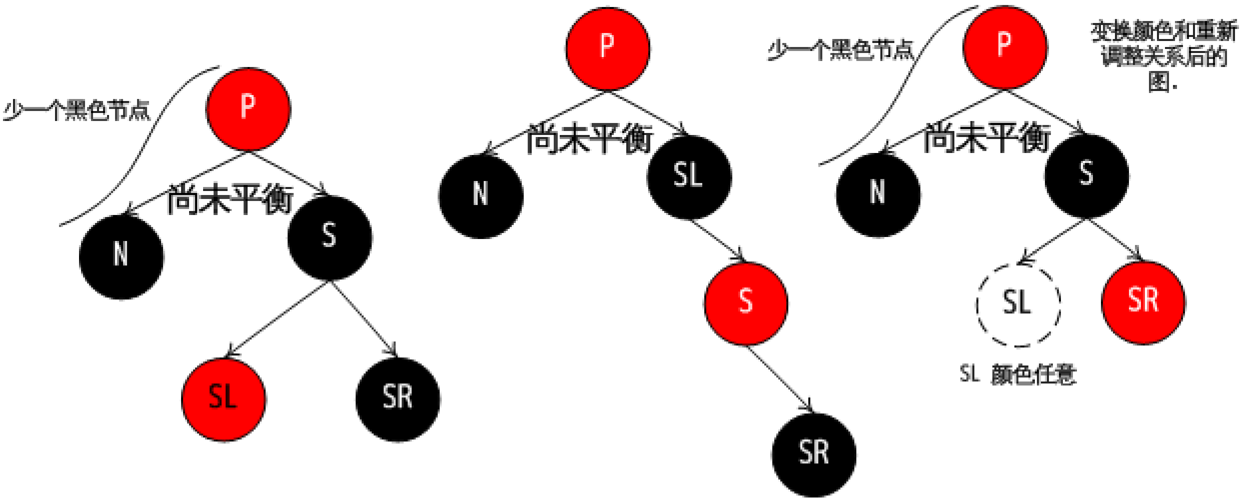
第 2 情况, S 红, 根据红黑树性质 P,SL,SR 一定黑. **策略: 旋转, 交换 P,S 颜色.** 处理后关注的范围缩小, 下面的情况对应下面的框图, 算法从框图重新开始, 进入下一个情况:



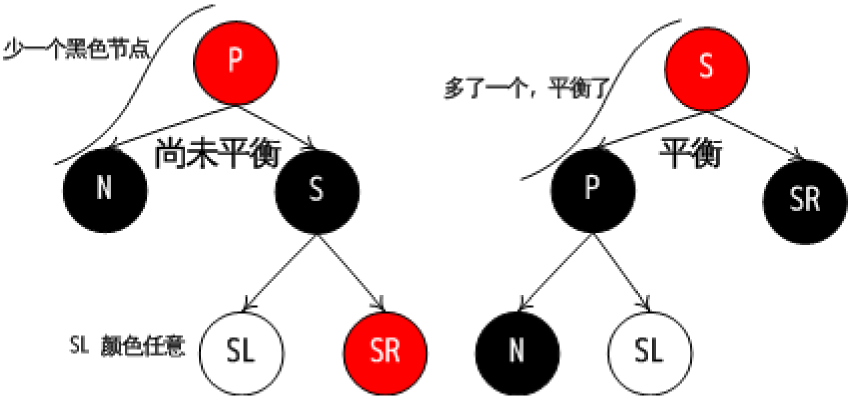
第 2.1 情况, S,SL,SR 都黑. **策略: P->黑. S->红,** 因为通过 N 的路径多了一个黑节点, 通过 S 的黑节点个数不变, 所以维持了性质 5. over. 将看到, sgi stl map 源代码中将第 2.1 和第 1 情况合并成一种情况, 下节展开.



第 2.2.1 情况, S,SR 黑, SL 红. **策略: 旋转, 变换 SL,S 颜色.** 从而又进入下一种情况:



第 2.2.2 情况, S 黑, SR 红. **策略: 旋转, 交换 S,P 颜色, SR->黑色, 重新获得平衡.**



上面情况标号 X.X.X 并不是说这些关系是嵌套的, 只是这样展开容易理解. 此时, 解释三个地方:

1. 通过 N 的黑色节点数量多了一个
2. 通过 SL 的黑色节点数量不变
3. 通过 SR 的黑色节点数量不变

红黑树删除重新调整伪代码如下:

```
// 第 0.0 情况, N 为根节点. over
if N.parent == NULL:
```

```
    return;

// 第 0.1 情况, 删除的节点为红. over
if color == RED:
    return;

// 第 0.2 情况, 删除节点为黑, N 为红, 简单变换: N->黑, 重新平衡. over
if color == BLACK && N.color == RED:
    N.color = BLACK;

// 第 1 种情况, N,P,S,SR,SL 都黑. 策略: S->红. 通过 N,S 的黑色节点数量相同了, 但会比其他路径多一个, 解决的方法是在 P 上从情况 0 开始继续调整.
if N,P,S,SR,SL.color == BLACK:
    S.color = RED;

    // 调整节点关系
    N = P
    N.parent = P.parent
    S = P.paernt.another_child
    SL = S.left_child
    SR = S.right_child
    continue;

// 第 2 情况, S 红, 根据红黑树性质 P,SR,SL 一定黑. 旋转, 交换 P,S 颜色. 此时关注的范围缩小, 下面的情况对应下面的框图, 算法从框图重新开始.
if S.color == RED:
    rotate(P);
    swap(P.color,S.color);

    // 调整节点关系
    S = P.another_child
    SL = S.left_child
    SR = S.right_child

// 第 2.1 情况, S,SL,SR 都黑. 策略: P->黑. S->红, 因为通过 N 的路径多了一个黑节点, 通过 S 的黑节点个数不变, 所以维持了性质 5. over. 将看到, sgi stl map 源代码中将第 2.1 和第 1 情况合并成一种情况, 下节展开.
if S,SL,SR.color == BLACK:
    P.color = BLACK;
    S.color = RED;
    return
```

```
// 第 2.2.1 情况, S,SR 黑, SL 红. 策略: 旋转, 变换 SL,S 颜色. 从而又进入下一种情况:
if  S,SR.color == BLACK && SL.color == RED:
    rotate(P);
    swap(S.color,SL.color);

    // 调整节点关系
    S = SL
    SL = S.left_child
    SR = S.right_child

// 第 2.2.2 情况, S 黑, SR 红. 策略: 旋转, 交换 S,P 颜色.
if S.color == BLACK && SR.color == RED:
    rotate(P);
    swap(P.color,S.color);
    return;
```

总结

所以, 插入的情况只有三种, 删除的情况只有两种. 上面的分析, 做镜像处理, 就能得到插入删除的全部算法, 脑补吧. 从上面的分析来看, 红黑树具有以下特性: 插入删除操作都是 0(lnN), 且最多旋转三次.