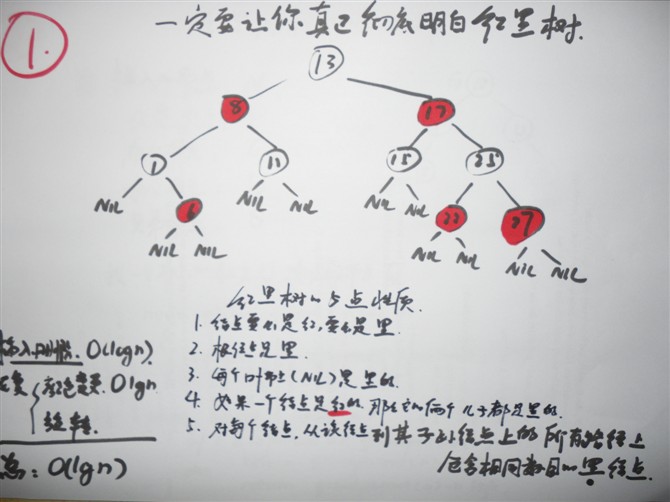
前言：  
1、有读者反应，说看了我的前几篇文章，对红黑树的了解还是不够透彻。  
2、我个人觉得，如果我一步一步，用图+代码来阐述各种插入、删除情况，可能会更直观易懂。  
3、既然写了红黑树，那么我就一定要把它真正写好，让读者真正彻底明白红黑树。

本文相对我前面红黑树相关的3篇文章，主要有以下几点改进：  
1.图、文字叙述、代码编写，彼此对应，明朗而清晰。  
2.宏观总结，红黑树的性质与插入、删除情况的认识。  
3.代码来的更直接，结合图，给你最直观的感受，彻底明白红黑树。

ok，首先，以下几点，你现在应该是要清楚明白了的：  
I、红黑树的五个性质：  
1）每个结点要么是红的，要么是黑的。  
2）根结点是黑的。  
3）每个叶结点，即空结点（NIL）是黑的。  
4）如果一个结点是红的，那么它的俩个儿子都是黑的。  
5）对每个结点，从该结点到其子孙结点的所有路径上包含相同数目的黑结点。  


II、红黑树插入的几种情况：  
情况1，z的叔叔y是红色的。  
情况2：z的叔叔y是黑色的，且z是右孩子  
情况3：z的叔叔y是黑色的，且z是左孩子

III、红黑树删除的几种情况。  
情况1：x的兄弟w是红色的。  
情况2：x的兄弟w是黑色的，且w的俩个孩子都是黑色的。  
情况3：x的兄弟w是黑色的，且w的左孩子是红色，w的右孩子是黑色。  
情况4：x的兄弟w是黑色的，且w的右孩子是红色的。

除此之外，还得明确一点：  
IV、我们知道，红黑树插入、或删除结点后，  
可能会违背、或破坏红黑树的原有的性质，  
所以为了使插入、或删除结点后的树依然维持为一棵新的红黑树，  
那就要做俩方面的工作：  
1、部分结点颜色，重新着色  
2、调整部分指针的指向，即左旋、右旋。

V、并区别以下俩种操作：  
1)红黑树插入、删除结点的操作，RB-INSERT(T, z)，RB-DELETE(T, z)  
2).红黑树已经插入、删除结点之后，  
为了保持红黑树原有的红黑性质而做的恢复与保持红黑性质的操作。  
如RB-INSERT-FIXUP(T, z)，RB-DELETE-FIXUP(T, x)

以上这5点，我已经在我前面的2篇文章，都已阐述过不少次了，希望，你现在已经透彻明了。

---------------------------------------------------------------------

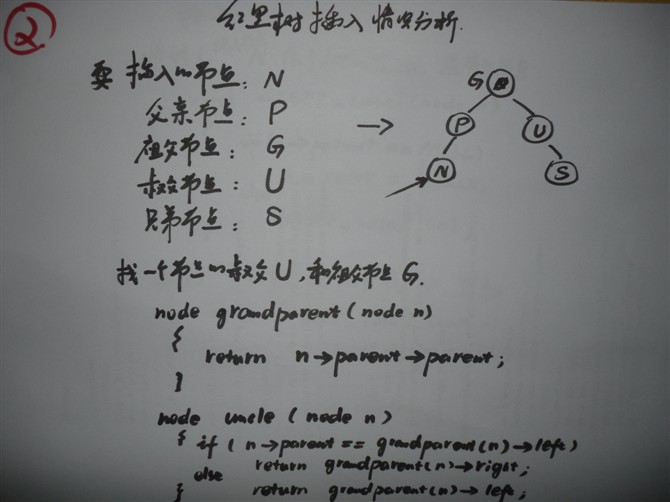
**本文，着重图解分析红黑树插入、删除结点后为了维持红黑性质而做修复工作的各种情况。**[下文各种插入、删除的情况，与我的第二篇文章，红黑树算法的实现与剖析相对应]

ok，开始。  
**一、在下面的分析中，我们约定：**要插入的节点为，N  
父亲节点，P  
祖父节点，G  
叔叔节点，U  
兄弟节点，S

如下图所示，找一个节点的祖父和叔叔节点:  
node grandparent(node n)     //祖父

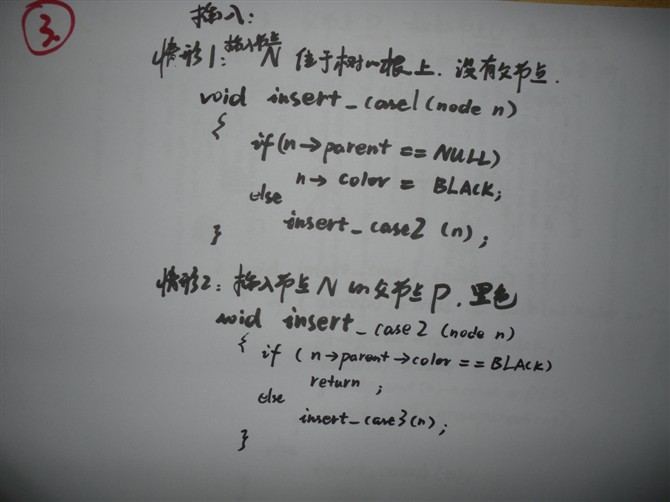
{  
     return n->parent->parent;  
 }  
   
 node uncle(node n)              //叔叔

{  
     if (n->parent == grandparent(n)->left)  
         return grandparent(n)->right;  
     else  
         return grandparent(n)->left;  
 }

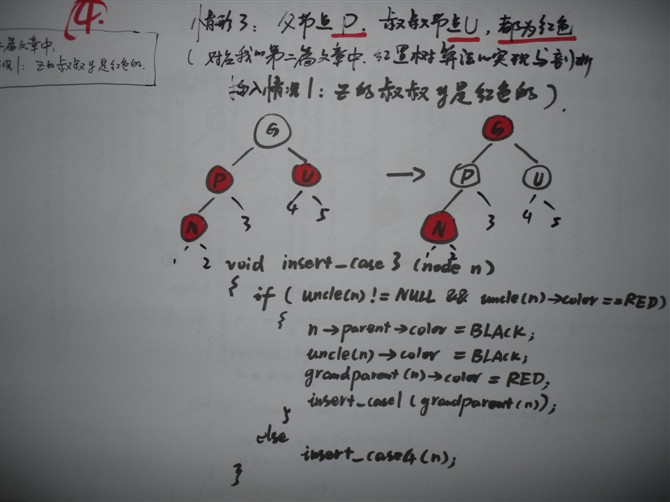


**二、红黑树插入的几种情况  
情形1: 新节点N位于树的根上，没有父节点**void insert\_case1(node n) {  
     if (n->parent == NULL)  
         n->color = BLACK;  
     else  
         insert\_case2(n);  
 }

**情形2: 新节点的父节点P是黑色**void insert\_case2(node n) {  
     if (n->parent->color == BLACK)  
         return; /\* 树仍旧有效 \*/  
     else  
         insert\_case3(n);  
 }

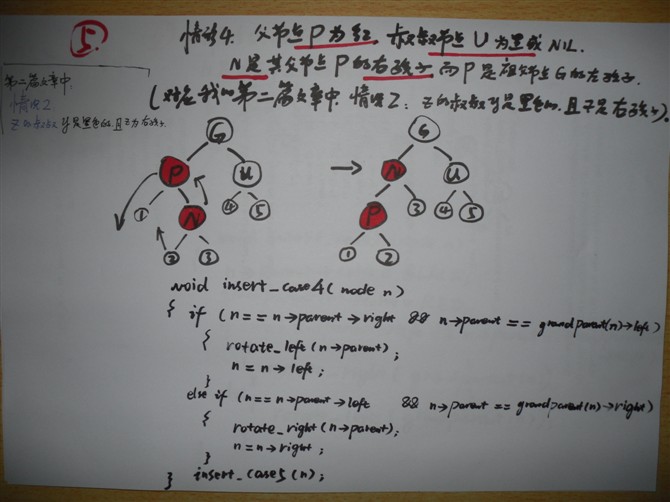


**情形3:父节点P、叔叔节点U，都为红色，**[对应第二篇文章中，的情况1：z的叔叔是红色的。]  
void insert\_case3(node n) {  
     if (uncle(n) != NULL && uncle(n)->color == RED) {  
         n->parent->color = BLACK;  
         uncle(n)->color = BLACK;  
         grandparent(n)->color = RED;  
         insert\_case1(grandparent(n));   //因为祖父节点可能是红色的，违反性质4，递归情形1.  
     }  
     else  
         insert\_case4(n);   //否则，叔叔是黑色的，转到下述情形4处理。

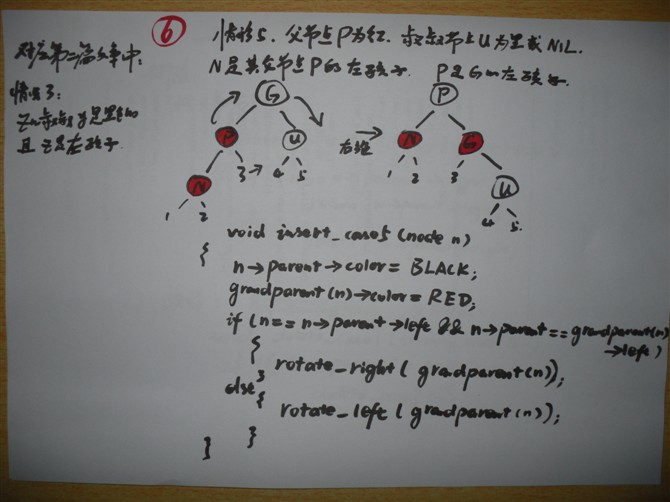


此时新插入节点N做为P的左子节点或右子节点都属于上述**情形3**,上图仅显示N做为P左子的情形。

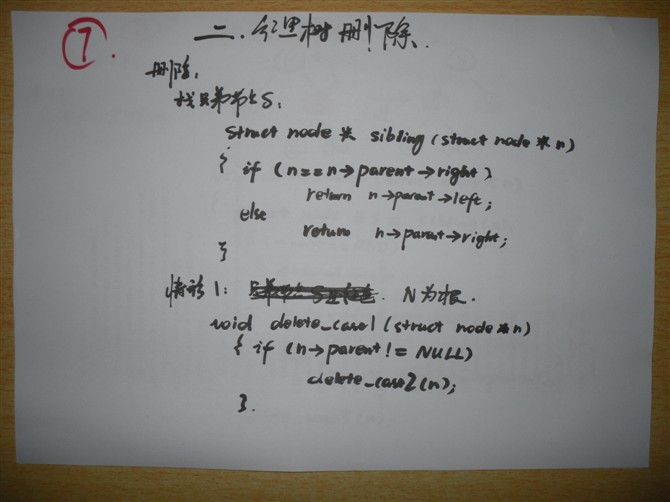
**情形4: 父节点P是红色，叔叔节点U是黑色或NIL;**插入节点**N是其父节点P的右孩子**，而父节点P又是其父节点的左孩子。  
[对应我第二篇文章中，的情况2：z的叔叔是黑色的，且z是右孩子]  
void insert\_case4(node n) {  
     if (n == n->parent->right && n->parent == grandparent(n)->left) {  
         rotate\_left(n->parent);  
         n = n->left;  
     } else if (n == n->parent->left && n->parent == grandparent(n)->right) {  
         rotate\_right(n->parent);  
         n = n->right;  
     }  
     insert\_case5(n);    //转到下述情形5处理。



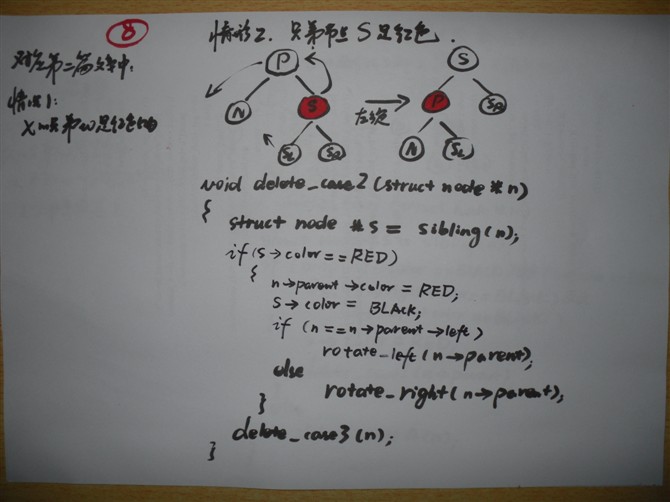
**情形5: 父节点P是红色，而叔父节点U 是黑色或NIL，**要插入的节点**N 是其父节点的左孩子**，而父节点P又是其父G的左孩子。  
[对应我第二篇文章中，情况3：z的叔叔是黑色的，且z是左孩子。]  
void insert\_case5(node n) {  
     n->parent->color = BLACK;  
     grandparent(n)->color = RED;  
     if (n == n->parent->left && n->parent == grandparent(n)->left) {  
         rotate\_right(grandparent(n));  
     } else {  
         /\* 反情况，N 是其父节点的右孩子，而父节点P又是其父G的右孩子 \*/  
         rotate\_left(grandparent(n));  
     }  
 }



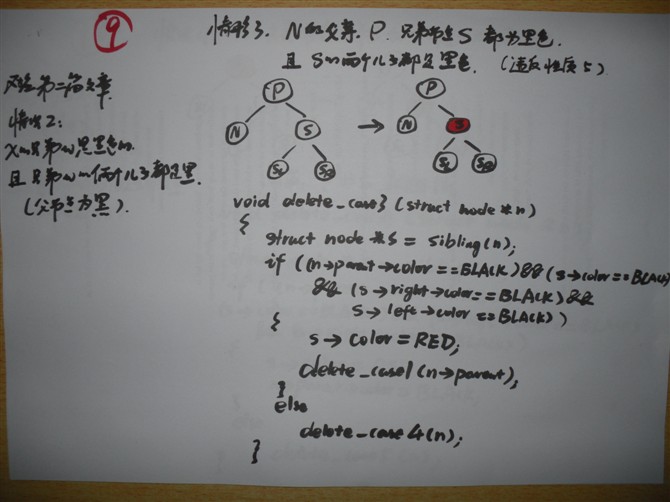
**三、红黑树删除的几种情况**上文我们约定，兄弟节点设为S，我们使用下述函数找到兄弟节点:  
struct node \* sibling(struct node \*n)  //找兄弟节点  
{  
        if (n == n->parent->left)  
                return n->parent->right;  
        else  
                return n->parent->left;  
}

**情况1: N 是新的根。**void  
delete\_case1(struct node \*n)  
{  
        if (n->parent != NULL)  
                delete\_case2(n);  
}  


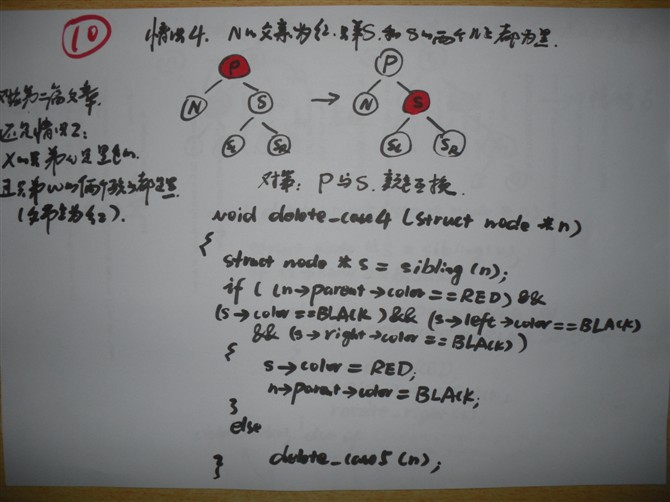
**情形2：兄弟节点S是红色**[对应我第二篇文章中，情况1：x的兄弟w是红色的。]  
void delete\_case2(struct node \*n)  
{  
        struct node \*s = sibling(n);  
   
        if (s->color == RED) {  
                n->parent->color = RED;  
                s->color = BLACK;  
                if (n == n->parent->left)  
                        rotate\_left(n->parent);  //左旋  
                else  
                        rotate\_right(n->parent);  
        }   
        delete\_case3(n);  
}



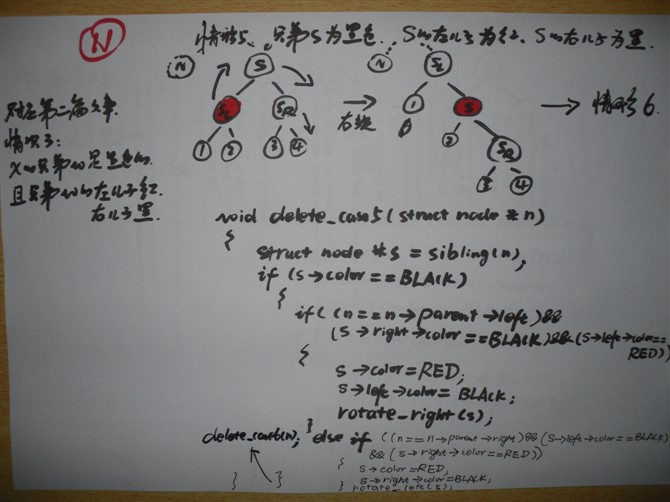
**情况 3: 兄弟节点S是黑色的，且S的俩个儿子都是黑色的。但N的父节点P，是黑色。**[对应我第二篇文章中，情况2：x的兄弟w是黑色的，且兄弟w的俩个儿子都是黑色的。  
(这里，父节点P为黑)]  
void delete\_case3(struct node \*n)  
{  
        struct node \*s = sibling(n);  
   
        if ((n->parent->color == BLACK) &&  
            (s->color == BLACK) &&  
            (s->left->color == BLACK) &&  
            (s->right->color == BLACK)) {  
                s->color = RED;  
                delete\_case1(n->parent);  
        } else  
                delete\_case4(n);  
}

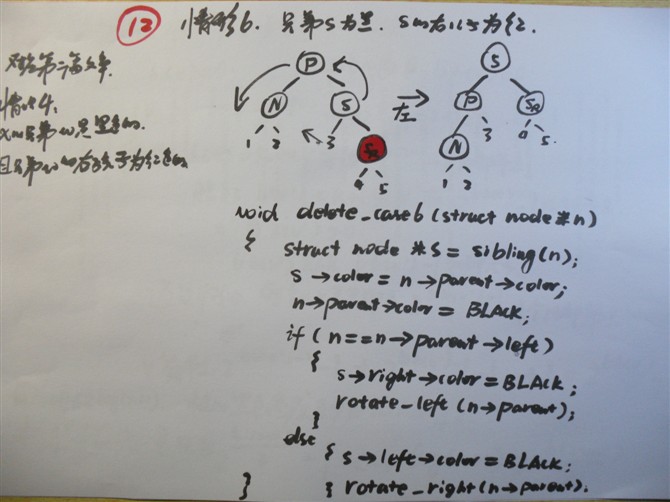


**情况4:** 兄弟节点**S 是黑色的、S 的儿子也都是黑色的，但是 N 的父亲P，是红色。**[还是对应我第二篇文章中，情况2：x的兄弟w是黑色的，且w的俩个孩子都是黑色的。  
(这里，父节点P为红)]  
void delete\_case4(struct node \*n)  
{  
        struct node \*s = sibling(n);  
   
        if ((n->parent->color == RED) &&  
            (s->color == BLACK) &&  
            (s->left->color == BLACK) &&  
            (s->right->color == BLACK)) {  
                s->color = RED;  
                n->parent->color = BLACK;  
        } else  
                delete\_case5(n);  
}



**情况5: 兄弟S为黑色，S 的左儿子是红色，S 的右儿子是黑色，而N是它父亲的左儿子。**//此种情况，最后转化到下面的情况6。  
[对应我第二篇文章中，情况3：x的兄弟w是黑色的，w的左孩子是红色，w的右孩子是黑色。]  
void delete\_case5(struct node \*n)  
{  
        struct node \*s = sibling(n);  
   
        if  (s->color == BLACK)   
                if ((n == n->parent->left) &&  
                    (s->right->color == BLACK) &&  
                    (s->left->color == RED)) {   
                        // this last test is trivial too due to cases 2-4.  
                        s->color = RED;  
                        s->left->color = BLACK;  
                        rotate\_right(s);  
                } else if ((n == n->parent->right) &&  
                           (s->left->color == BLACK) &&  
                           (s->right->color == RED)) {  
                       // this last test is trivial too due to cases 2-4.  
                        s->color = RED;  
                        s->right->color = BLACK;  
                        rotate\_left(s);  
                }  
        }  
        delete\_case6(n);  //转到情况6。



**情况6: 兄弟节点S是黑色，S的右儿子是红色，而 N 是它父亲的左儿子。**[对应我第二篇文章中，情况4:x的兄弟w是黑色的，且w的右孩子时红色的。]  
void delete\_case6(struct node \*n)  
{  
        struct node \*s = sibling(n);  
   
        s->color = n->parent->color;  
        n->parent->color = BLACK;  
   
        if (n == n->parent->left) {  
                s->right->color = BLACK;  
                rotate\_left(n->parent);  
        } else {  
                s->left->color = BLACK;  
                rotate\_right(n->parent);  
        }  
}  


//呵呵，画这12张图，直接从中午画到了晚上。希望，此文能让你明白。

**四、红黑树的插入、删除情况时间复杂度的分析**因为每一个红黑树也是一个特化的二叉查找树，  
因此红黑树上的只读操作与普通二叉查找树上的只读操作相同。  
然而，在红黑树上进行插入操作和删除操作会导致不再符合红黑树的性质。

恢复红黑树的属性需要少量(O(log n))的颜色变更(实际是非常快速的)和  
不超过三次树旋转(对于插入操作是两次)。  
虽然插入和删除很复杂，但操作时间仍可以保持为 O(log n) 次。

