高级树结构剖析

search tree

目录/CONTENTS

01

概述

s u m m a r y

Search trees的概述

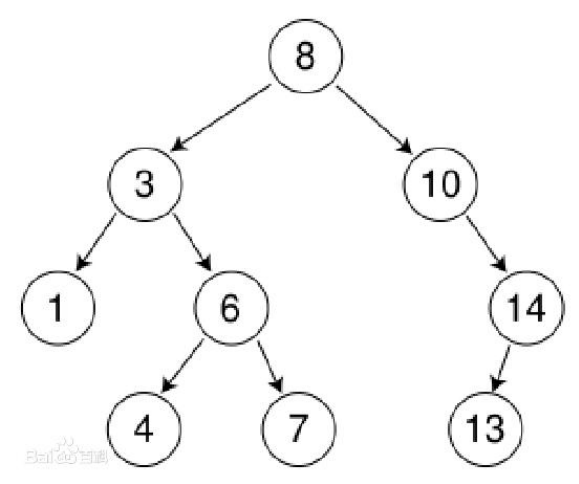
Search trees中的典型有**红黑树**和**AVL树**，红黑树是一种特化的AVL树，因此我们针对红黑树进行研究。红黑树是一种自平衡二叉查找树，它的操作有着良好的最坏情况运行时间，并且在实践中高效：它可以在O(logn)时间内完成查找、插入和删除，这里的n是树中元素的数目。想要说清楚红黑树，首先要说清楚**二叉查找树（BST）**，而BST具有以下特性：

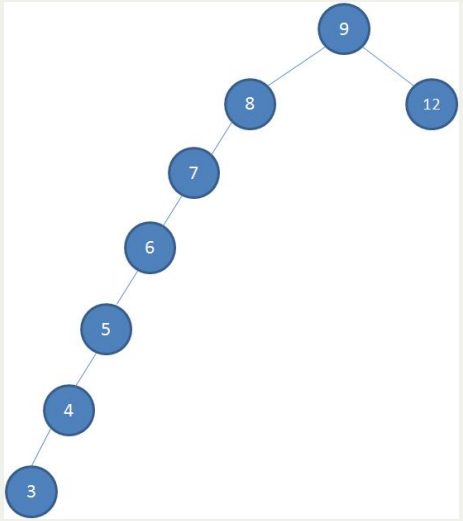
1.左子树上所有结点的值均小于它的根结点的值。

2.右子树上所有结点的值均大于它的根结点的值。

由于BST具有这种特性，所以，当我要查找某个数值x时，就可以通过不断比较子树大小， 从而在较短的时间内找到。如我要找6，由于6<8，因此遍历到8的左孩子3，由于6>3，所以遍历3的右孩子6，从而查找成功。（如下图1）

但也由于这种性质，在多次插入新结点，容易出现线性的情况，打破了平衡状态（如下图2）。因此在这基础上，发明了红黑树，红黑树能够一定程度上维持自平衡。





Search trees的概述

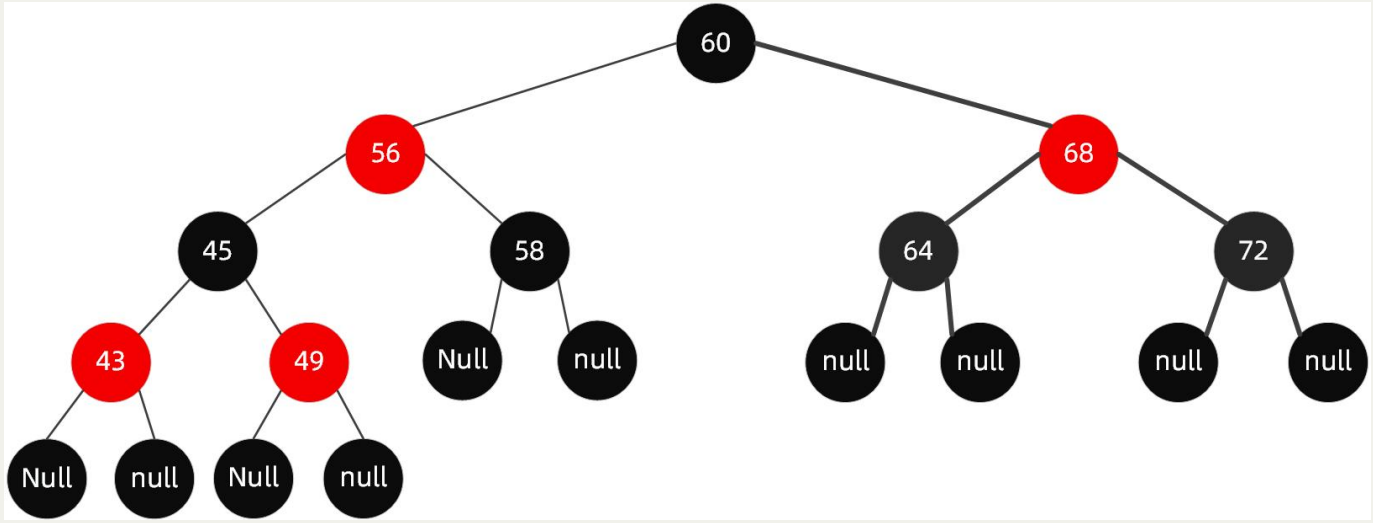
红黑树是每个节点都带有颜色属性的二叉查找树，颜色为红色或黑色。在二叉查找树强制一般要求以外，对于任何有效的红黑树我们增加了如下的额外要求：

1.节点是红色或黑色。2.根是黑色。3.所有叶子都是黑色（叶子是NIL节点）。

4.每个红色节点必须有两个黑色的子节点。（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。）

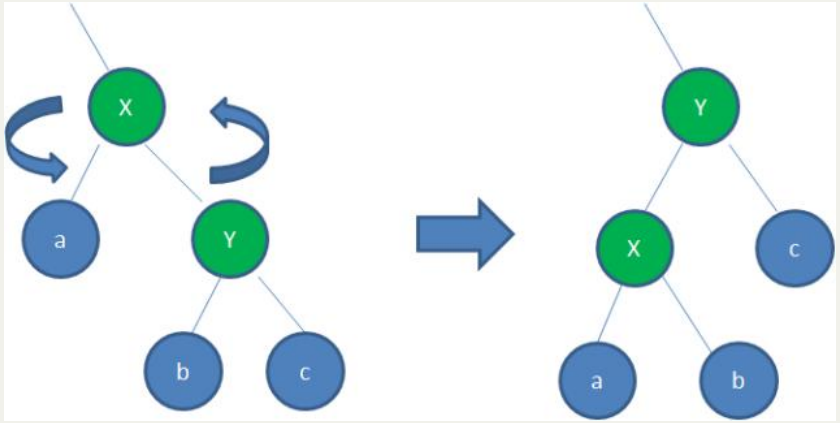
5.从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点。（图例如下图）

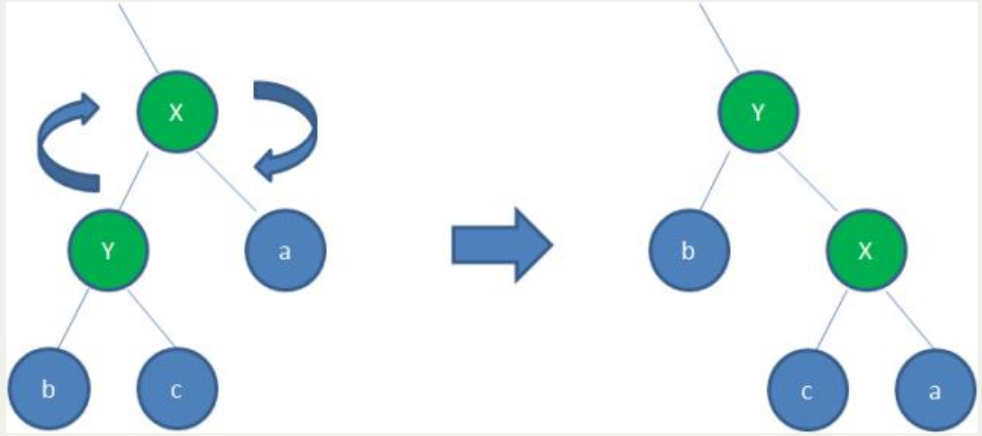
这些约束确保了红黑树的关键特性：**从根到叶子的最长的可能路径不多于最短的可能路径的两倍长。**结果是这个树大致上是平衡的。因为操作比如插入、删除和查找某个值的最坏情况时间都要求与树的高度成比例，这个在高度上的理论上限允许红黑树在最坏情况下都是高效的，而不同于普通的二叉查找树。要知道为什么这些性质确保了这个结果，注意到性质4导致了路径不能有两个毗连的红色节点就足够了。最短的可能路径都是黑色节点，最长的可能路径有交替的红色和黑色节点。因为根据性质5所有最长的路径都有相同数目的黑色节点，这就表明了没有路径能多于任何其他路径的两倍长。（引用自维基百科）



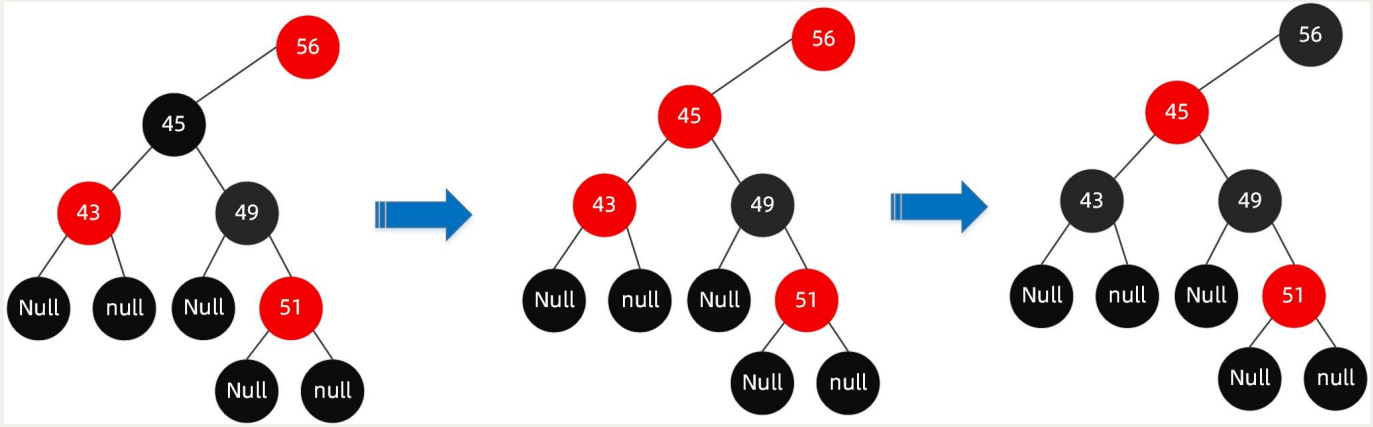
Search trees的概述

红黑树还可以进行**变色**和**旋转**。
先介绍左旋转，左旋转是逆时针旋转红黑树的两个节点，使得父节点被自己的右孩子取代，而自己成为自己的左孩子。如左图，图中，身为右孩子的Y取代了X的位置，而X变成了自己的左孩子。
而右旋转，是顺时针旋转红黑树的两个节点，使得父节点被自己的左孩子取代，而自己成为自己的右孩子。如右图，图中，身为左孩子的Y取代了X的位置，而X变成了自己的右孩子。（在BST中也有这种操作）





而变色，顾名思义，就是结点从红变黑，或者从黑变红。变色一般和旋转同时出现，以满足红黑树自身的规律。一般如右图。



02

物理结构的定义

physical structure

物 理 结 构 的 定 义

红黑树的物理结构本质上是链式存储结构，基本框架如下：

```
struct Node {
    int value;
    bool color;
    Node *leftTree, *rightTree, *parent;
    Node() : value(0), color(RED), leftTree(NULL), rightTree(NULL), parent(NULL) { }
    Node* grandparent() {
        if(parent == NULL){
            return NULL;
        }
        return parent->parent;
    }
    Node* uncle() {
        if(grandparent() == NULL) {
            return NULL;
        }
        if(parent == grandparent()->rightTree)
            return grandparent()->leftTree;
        else
            return grandparent()->rightTree;
    }
    Node* sibling() {
        if(parent->leftTree == this)
            return parent->rightTree;
        else
            return parent->leftTree;
    }
};
```

03

关键操作概述

key operation

插入

insert

我们首先以二叉查找树的方法增加节点并标记它为红色。（如果设为黑色，就会导致根到叶子的路径上有一条路上，多一个额外的黑节点，这个是很难调整的。但是设为红色节点后，可能会导致出现两个连续红色节点的冲突，那么可以通过颜色调换（color flips）和树旋转来调整。）下面要进行什么操作取决于其他临近节点的颜色。同人类的家族树中一样，我们将使用术语叔父节点来指一个节点的父节点的兄弟节点。注意：
性质1和性质3总是保持着。
性质4只在增加红色节点、重绘黑色节点为红色，或做旋转时受到威胁。
性质5只在增加黑色节点、重绘红色节点为黑色，或做旋转时受到威胁。
在下面的示意图中，将要插入的节点标为N，N的父节点标为P，N的祖父节点标为G，N的叔父节点标为U。
在图中展示的任何颜色要么是由它所处情形这些所作的假定，要么是假定所暗含（imply）的。通过下列函数，可以找到一个节点的叔父和祖父节点：

```
node* grandparent(node *n){
    return n->parent->parent;
}

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

关 键 操 作 概 述

我们首先以二叉查找树的方法增加节点并标记它为红色。（**如果设为黑色，就会导致根到叶子的路径上有一条路上，多一个额外的黑节点，这个是很难调整的。但是设为红色节点后，可能会导致出现两个连续红色节点的冲突，那么可以通过颜色调换（color flips）和树旋转来调整。**）下面要进行什么操作取决于其他临近节点的颜色。同人类的家族树中一样，我们将使用术语叔父节点来指一个节点的父节点的兄弟节点。注意：
性质1和性质3总是保持着。
性质4只在增加红色节点、重绘黑色节点为红色，或做旋转时受到威胁。
性质5只在增加黑色节点、重绘红色节点为黑色，或做旋转时受到威胁。
在下面的示意图中，将要插入的节点标为N，N的父节点标为P，N的祖父节点标为G，N的叔父节点标为U。
在图中展示的任何颜色要么是由它所处情形这些所作的假定，要么是假定所暗含（imply）的。通过下列函数，可以找到一个节点的叔父和祖父节点：

```
node* grandparent(node *n){
    return n->parent->parent;
}

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

注意插入实际上是原地算法，以下所有调用都使用了尾部递归。

关 键 操 作 概 述

情形1:新节点N位于树的根上，没有父节点。在这种情形下，我们把它重绘为黑色以满足性质2。因为它在每个路径上对黑节点数目增加一，性质5符合。

```
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，所以性质4没有失效（新节点是红色的）。在这种情形下，树仍是有效的。性质5也未受到威胁，尽管新节点N有两个黑色叶子子节点；但由于新节点N是红色，通过它的每个子节点的路径就都有同通过它所取代的黑色的叶子的路径同样数目的黑色节点，所以依然满足这个性质。

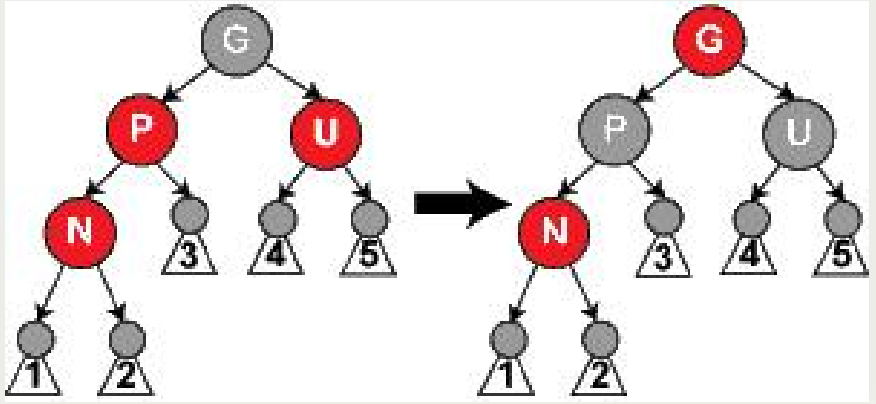
```
void insert_case2(node *n){
    if(n->parent->color == BLACK)
        return; /* 树仍旧有效*/
    else
        insert_case3 (n);
}
```

以上最简单的两种情况

关 键 操 作 概 述

情形3:如果父节点P和叔父节点U二者都是红色，（此时新插入节点N做为P的左子节点或右子节点都属于情形3，这里右图仅显示N做为P左子的情形）则我们可以将它们两个重绘为黑色并重绘祖父节点G为红色（用来保持性质5）。现在我们的新节点N有了一个黑色的父节点P。因为通过父节点P或叔父节点U的任何路径都必定通过祖父节点G，在这些路径上的黑节点数目没有改变。**但是，红色的祖父节点G可能是根节点，这就违反了性质2，也有可能祖父节点G的父节点是红色的，这就违反了性质4。**为了解决这个问题，我们在祖父节点G上递归地进行情形1的整个过程。（把G当成是新加入的节点进行各种情形的检查）

```
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));
    }
    else
        insert_case4 (n);
}
```



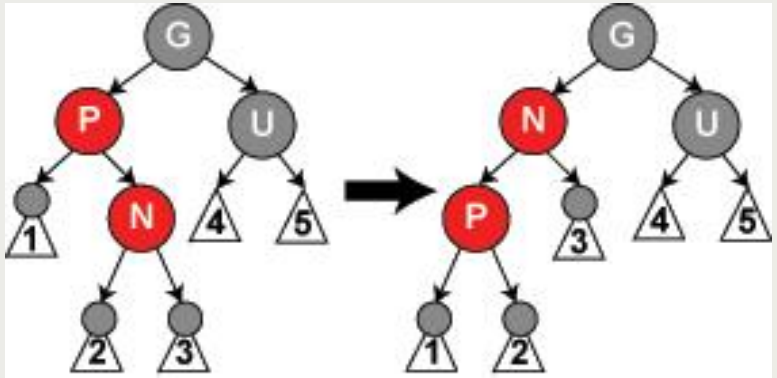
关 键 操 作 概 述

注意：在余下的情形下，我们假定父节点P是其祖父G的左子节点。如果它是右子节点，情形4和情形5中的左和右应当对调。

情形4是和情形5相连的

情形4:父节点P是红色而叔父节点U是黑色或缺少，并且新节点N是其父节点P的右子节点而父节点P又是其父节点的左子节点。在这种情形下，我们进行一次左旋转调换新节点和其父节点的角色;接着，我们按情形5处理以前的父节点P以解决仍然失效的性质4。注意这个改变会导致某些路径通过它们以前不通过的新节点N（比如图中1号叶子节点）或不通过节点P（比如图中3号叶子节点），但由于这两个节点都是红色的，所以性质5仍有效。

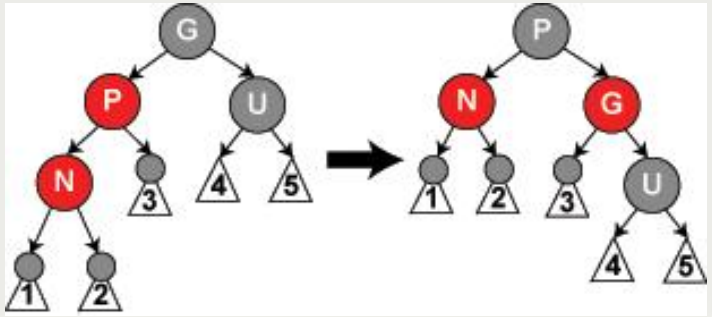
```
void insert_case4(node *n){
    if(n == n->parent->right && n->parent == grandparent(n)->left) {
        rotate_left(n);
        n = n->left;
    } else if(n == n->parent->left && n->parent == grandparent(n)->right) {
        rotate_right(n);
        n = n->right;
    }
    insert_case5 (n);
}
```



关 键 操 作 概 述

情形5：父节点P是红色而叔父节点U是黑色或缺少，新节点N是其父节点的左子节点，而父节点P又是其父节点G的左子节点。在这种情形下，我们进行针对祖父节点G的一次右旋转；在旋转产生的树中，以前的父节点P现在是新节点N和以前的祖父节点G的父节点。我们知道以前的祖父节点G是黑色，否则父节点P就不可能是红色（如果P和G都是红色就违反了性质4，所以G必须是黑色）。我们切换以前的父节点P和祖父节点G的颜色，结果的树满足性质4。性质5也仍然保持满足，因为通过这三个节点中任何一个的所有路径以前都通过祖父节点G，现在它们都通过以前的父节点P。在各自的情形下，这都是三个节点中唯一的黑色节点。

```
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(n->parent);
    } else {
        /* Here, n == n->parent->right && n->parent == grandparent (n)->right */
        rotate_left(n->parent);
    }
}
```



删除

delete

如果需要删除的节点有两个儿子，那么问题可以被转化成删除另一个只有一个儿子的节点的问题（为了表述方便，这里所指的儿子，为非叶子节点的儿子）。对于二叉查找树，在删除带有两个非叶子儿子的节点的时候，**我们要么找到它左子树中的最大元素、要么找到它右子树中的最小元素，**并把它的值转移到要删除的节点中（如在这里所展示的那样）。我们接着删除我们从中复制出值的那个节点，它必定有少于两个非叶子的儿子。因为只是复制了一个值（没有复制颜色），不违反任何性质，这就把问题简化为如何删除最多有一个儿子的节点的问题。它不关心这个节点是最初要删除的节点还是我们从中复制出值的那个节点。

在本文余下的部分中，我们只需要讨论删除只有一个儿子的节点（如果它两个儿子都为空，即均为叶子，我们任意将其中一个看作它的儿子）。如果我们删除一个红色节点（此时该节点的儿子将都为叶子节点），它的父亲和儿子一定是黑色的。所以我们可以简单的用它的黑色儿子替换它，并不会破坏性质3和性质4。通过被删除节点的所有路径只是少了一个红色节点，这样可以继续保证性质5。另一种简单情况是在被删除节点是黑色而它的儿子是红色的时候。如果只是去除这个黑色节点，用它的红色儿子顶替上来的话，会破坏性质5，但是如果我们重绘它的儿子为黑色，则曾经通过它的所有路径将通过它的黑色儿子，这样可以继续保持性质5。需要进一步讨论的是在要删除的节点和它的儿子二者都是黑色的时候，这是一种复杂的情况（这种情况下该结点的两个儿子都是叶子结点，否则若其中一个儿子是黑色非叶子结点，另一个儿子是叶子结点，那么从该结点通过非叶子结点儿子的路径上的黑色结点数最小为2，而从该结点到另一个叶子结点儿子的路径上的黑色结点数为1，违反了性质5）。我们首先把要删除的节点替换为它的儿子。出于方便，称呼这个儿子为N（在新的位置上），称呼它的兄弟（它父亲的另一个儿子）为S。在下面的示意图中，我们还是使用P称呼N的父亲，SL称呼S的左儿子，SR称呼S的右儿子。我们将使用下述函数找到兄弟节点：

关 键 操 作 概 述

```
struct node *
sibling(struct node *n)
{
        if(n == n->parent->left)
                return n->parent->right;
        else
                return n->parent->left;
}
```

我们可以使用下列代码进行上述的概要步骤，这里的函数replace_node替换child到n在树中的位置。出于方便，在本章节中的代码将假定空叶子被用不是NULL的实际节点对象来表示（在插入章节中的代码可以同任何一种表示一起工作）。

```
delete_one_child(struct node *n)//删除只有一个孩子的结点
{/* * Precondition: n has at most one non-null child. */
        struct node *child = is_leaf(n->right)? n->left : n->right;
         replace_node(n, child);
        if(n->color == BLACK){
                if(child->color == RED)
                        child->color = BLACK;
                else
                        delete_case1 (child);
        }
        free (n);
}
```

关 键 操 作 概 述

如果N和它初始的父亲是黑色，则删除它的父亲导致通过N的路径都比不通过它的路径少了一个黑色节点。因为这违反了性质5，树需要被重新平衡。有几种情形需要考虑：

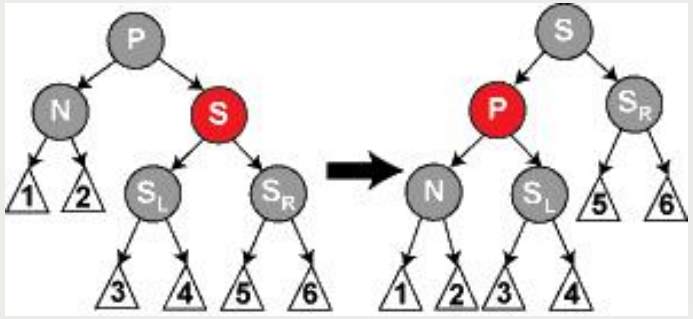
情形1: N是新的根。在这种情形下，我们就做完了。我们从所有路径去除了一个黑色节点，而新根是黑色的，所以性质都保持着。

```
void delete_case1(struct node *n)
{
        if(n->parent != NULL)
                delete_case2 (n);}
```

注意：在情形2、5和6下，我们假定N是它父亲的左儿子。如果它是右儿子，则在这些情形下的左和右应当对调。

情形2： S是红色。在这种情形下我们在N的父亲上做左旋转，把红色兄弟转换成N的祖父，我们接着对调N的父亲和祖父的颜色。完成这两个操作后，尽管所有路径上黑色节点的数目没有改变，但现在N有了一个黑色的兄弟和一个红色的父亲（它的新兄弟是黑色因为它是红色S的一个儿子），所以我们可以接下去按情形4、情形5或情形6来处理。

```
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);
                else
                        rotate_right(n->parent);}
        delete_case3 (n);}
```

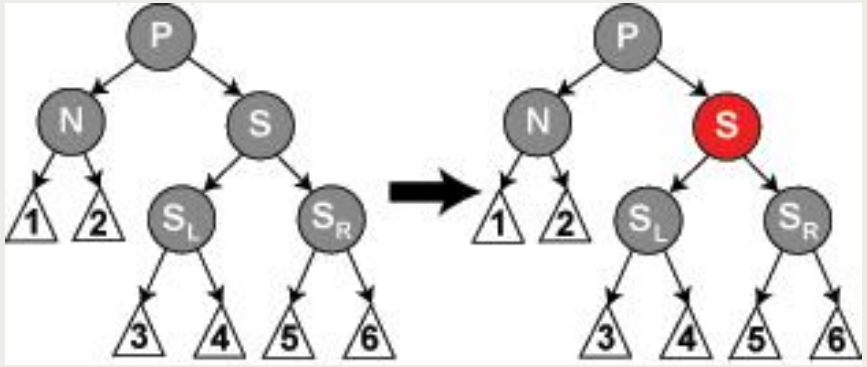


此处还需后面进行处理

关 键 操 作 概 述

情形3：N的父亲、S和S的儿子都是黑色的。在这种情形下，我们简单的重绘S为红色。结果是通过S的所有路径，它们就是以前不通过N的那些路径，都少了一个黑色节点。因为删除N的初始的父亲使通过N的所有路径少了一个黑色节点，这使事情都平衡了起来。但是，通过P的所有路径现在比不通过P的路径少了一个黑色节点，所以仍然违反性质5。要修正这个问题，我们要从情形1开始，在P上做重新平衡处理。

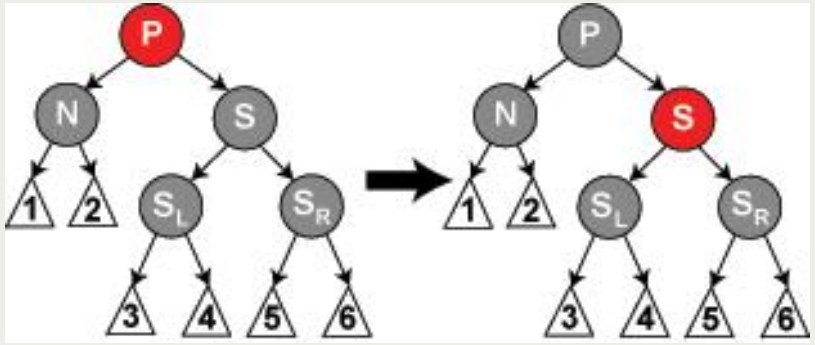
```
delete_case3(struct node *n)
{
        struct node *s = sibling (n);
        if((n->parent->color == BLACK)&&
(s->color == BLACK)&&
(s->left->color == BLACK)&&
(s->right->color == BLACK)) {
                s->color = RED;
                delete_case1(n->parent);
        } else
                delete_case4 (n);
}
```



关 键 操 作 概 述

情形4： S和S的儿子都是黑色，但是N的父亲是红色。在这种情形下，我们简单的交换N的兄弟和父亲的颜色。这不影响不通过N的路径的黑色节点的数目，但是它在通过N的路径上对黑色节点数目增加了一，添补了在这些路径上删除的黑色节点。

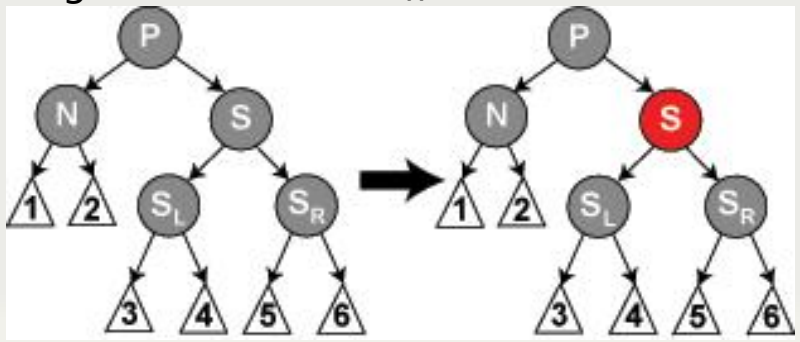
```
void
delete_case4(struct node *n)
{
        struct node *s = sibling (n);
        if  ((n->parent->color == RED)&&
(s->color == BLACK)&&
(s->left->color == BLACK)&&
(s->right->color == BLACK)) {
                s->color = RED;
                n->parent->color = BLACK;
        } else
                delete_case5 (n);
}
```



关 键 操 作 概 述

情形5： S是黑色，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。在这种情形下我们在S上做右旋转，这样S的左儿子成为S的父亲和N的新兄弟。我们接着交换S和它的新父亲的颜色。所有路径仍有同样数目的黑色节点，但是现在N有了一个黑色兄弟，他的右儿子是红色的，所以我们进入了情形6。N和它的父亲都不受这个变换的影响。

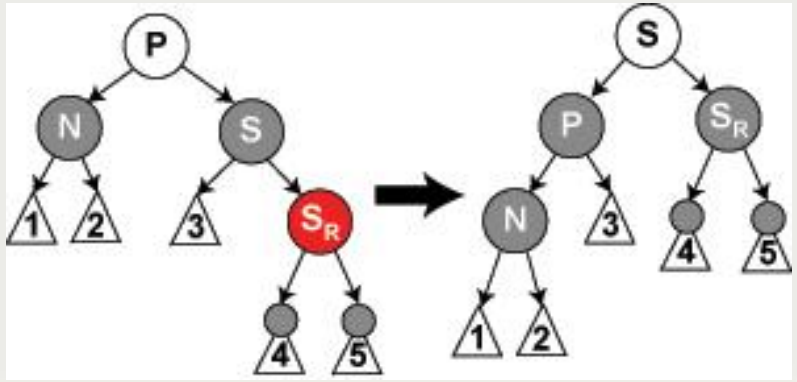
```
void delete_case5(struct node *n)
{
        struct node *s = sibling (n);
        if  (s->color == BLACK){ /* this if statement is trivial, due to Case 2(even though Case two changed the sibling to a
sibling's child, the sibling's child can't be red, since no red parent can have a red child). */
// the following statements just force the red to be on the left of the left of the parent,
// or right of the right, so case six will rotate correctly.
          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);}
```



关 键 操 作 概 述

情形6： S是黑色，S的右儿子是红色，而N是它父亲的左儿子。在这种情形下我们在N的父亲上做左旋转，这样S成为N的父亲（P）和S的右儿子的父亲。我们接着交换N的父亲和S的颜色，并使S的右儿子为黑色。子树在它的根上的仍是同样的颜色，所以性质3没有被违反。但是，N现在增加了一个黑色祖先：要么N的父亲变成黑色，要么它是黑色而S被增加为一个黑色祖父。所以，通过N的路径都增加了一个黑色节点。
此时，如果一个路径不通过N，则有两种可能性：
1.它通过N的新兄弟。那么它以前和现在都必定通过S和N的父亲，而它们只是交换了颜色。所以路径保持了同样数目的黑色节点。
2.它通过N的新叔父，S的右儿子。那么它以前通过S、S的父亲和S的右儿子，但是现在只通过S，它被假定为它以前的父亲的颜色，和S的右儿子，它被从红色改变为黑色。合成效果是这个路径通过了同样数目的黑色节点。
在任何情况下，在这些路径上的黑色节点数目都没有改变。所以我们恢复了性质4。在示意图中的白色节点可以是红色或黑色，但是在变换前后都必须指定相同的颜色。

```
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);}
        }
```



04

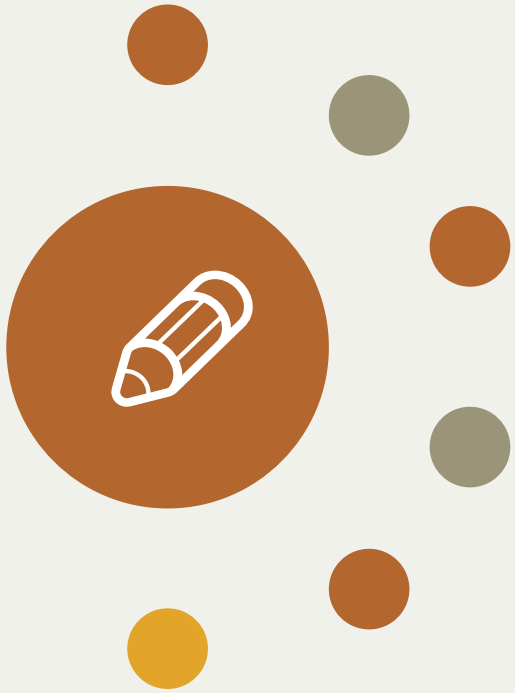
具体应用

a p p l i c a t i o n E

具　体　应　用

P1071 潜伏者

由于map是基于红黑树实现的STL，因此通过map的例题体会红黑树



题目描述：R国和S国正陷入战火之中，双方都互派间谍，潜入对方内部，伺机行动。历尽艰险后，潜伏于S国的R国间谍小C终于摸清了S国军用密码的编码规则：

1．S国军方内部欲发送的原信息经过加密后在网络上发送，原信息的内容与加密后所得的内容均由大写字母‘A’-‘Z’构成（无空格等其他字符）。

2．S国对于每个字母规定了对应的“密字”。加密的过程就是将原信息中的所有字母替换为其对应的“密字”。

3．每个字母只对应一个唯一的“密字”，不同的字母对应不同的“密字”。“密字”可以和原字母相同。

例如，若规定‘A’的密字为‘A’，‘B’的密字为‘C’（其他字母及密字略），则原信息“ABA”被加密为“ACA”。

现在，小C通过内线掌握了S国网络上发送的一条加密信息及其对应的原信息。小C希望能通过这条信息，破译S国的军用密码。小 C的破译过程是这样的：扫描原信息，对于原信息中的字母x（代表任一大写字母），找到其在加密信息中的对应大写字母y，并认为在密码里 y是x的密字。如此进行下去直到停止于如下的某个状态：

1．所有信息扫描完毕，‘A’-‘Z’所有 2626个字母在原信息中均出现过并获得了相应的“密字”。

2．所有信息扫描完毕，但发现存在某个（或某些）字母在原信息中没有出现。

3．扫描中发现掌握的信息里有明显的自相矛盾或错误（违反S国密码的编码规则）。例

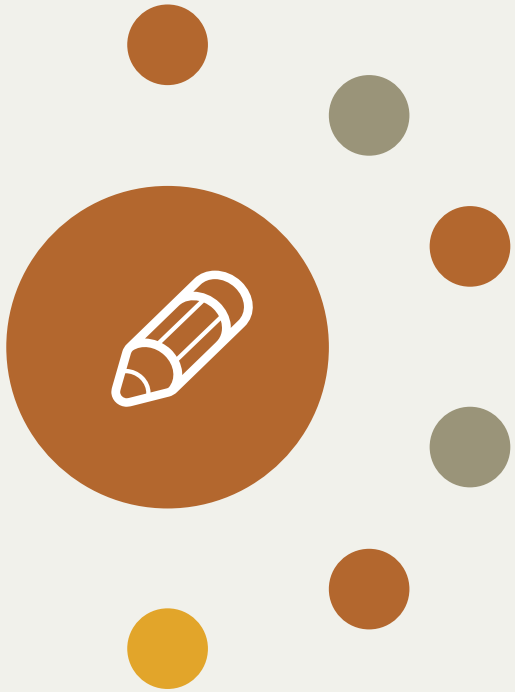
如某条信息“XYZ”被翻译为“ABA”就违反了“不同字母对应不同密字”的规则。

在小 C忙得头昏脑涨之际，R国司令部又发来电报，要求他翻译另外一条从S国刚刚截取到的加密信息。现在请你帮助小C：通过内线掌握的信息，尝试破译密码。然后利用破译的密码，翻译电报中的加密信息。

具　体　应　用

P1071 潜伏者

由于map是基于红黑树实现的STL，因此通过map的例题体会红黑树



输入格式：共 3行，每行为一个长度在1到100之间的字符串。

第1行为小C掌握的一条加密信息。

第2行第1行的加密信息所对应的原信息。

第3行为R国司令部要求小C翻译的加密信息。

输入数据保证所有字符串仅由大写字母‘A’-‘Z’构成，且第 1行长度与第 2行相等。

输出格式：共1 行。

若破译密码停止时出现 2,3两种情况，请你输出“Failed”（不含引号，注意首字母大写，其它小写）。

否则请输出利用密码翻译电报中加密信息后得到的原信息。

例如：

AA　　Failed

AB

EOWIE

或者

QWERTYUIOPLKJHGFDSAZXCVBN　　Failed

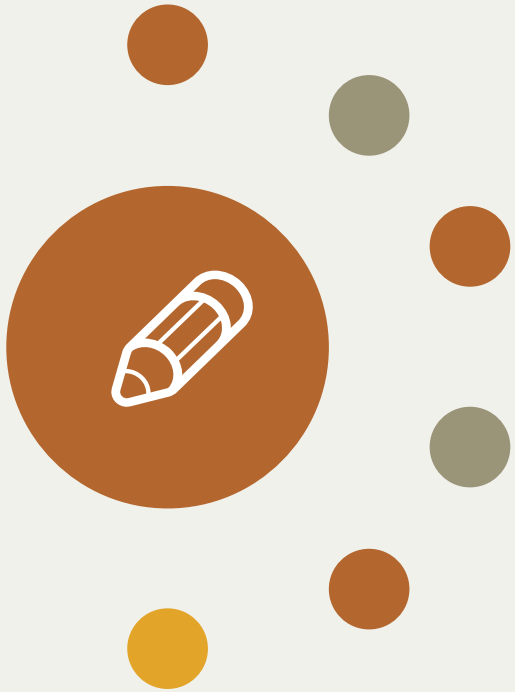
ABCDEFGHIJKLMNOPQRSTUVWXY

DSLIEWO

具　体　应　用

P1071 潜伏者

由于map是基于红黑树实现的STL，因此通过map的例题体会红黑树



输入格式：共 3行，每行为一个长度在1到100之间的字符串。

第1行为小C掌握的一条加密信息。

第2行第1行的加密信息所对应的原信息。

第3行为R国司令部要求小C翻译的加密信息。

输入数据保证所有字符串仅由大写字母‘A’-‘Z’构成，且第 1行长度与第 2行相等。

输出格式：共1 行。

若破译密码停止时出现 2,3两种情况，请你输出“Failed”（不含引号，注意首字母大写，其它小写）。

否则请输出利用密码翻译电报中加密信息后得到的原信息。

例如：

AA　　Failed

AB

EOWIE

或者

QWERTYUIOPLKJHGFDSAZXCVBN　　Failed

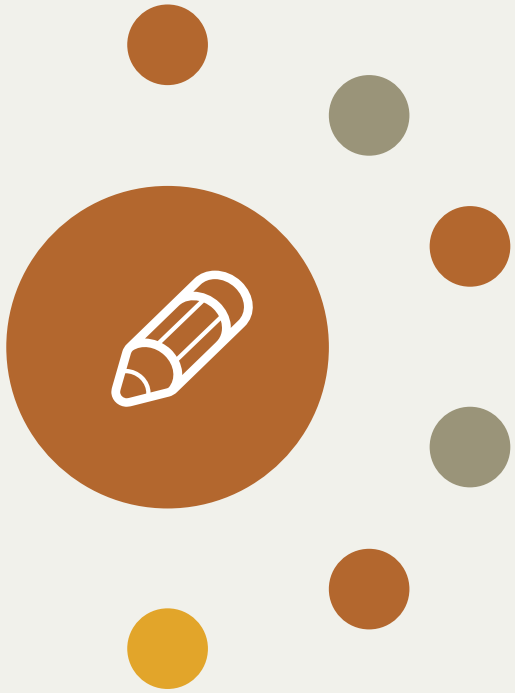
ABCDEFGHIJKLMNOPQRSTUVWXY

DSLIEWO

具　体　应　用

P1071 潜伏者

由于map是基于红黑树实现的STL，因此通过map的例题体会红黑树



```cpp
//题目要求一个明文对应一个密文，一个密文对应一个明文
using namespace std;
map<char,char> m1,m2;//m1记录密文对明文，m2记录明文对密文
char miwen[101],mingwen[101],translate[101];
int main(){
    cin>>miwen>>mingwen>>translate;
    for(int i=0;i<strlen(miwen);i++){//记录明文和密文的关系
    //判断密文多对一或者明文多对一的情况，如果存在，则Failed
    //count是返回指定元素的出现次数，也就是若一个元素出现不止一次，且对应不只一个元素，则Failed
        if((m1.count(miwen[i])&&m1[miwen[i]]!=mingwen[i])||(m2.count(mingwen[i])&&m2[mingwen[i]]!=miwen[i])){
            cout<<"Failed"<<endl;
            return 0;
        }
        //如果符合规则，则记录下来
        else{
            m1[miwen[i]]=mingwen[i];
            m2[mingwen[i]]=miwen[i];
        }
    }
    //再判断A`Z是否全部都有对应密文
    for(int i=0;i<26;i++){
        }
    }
    //再判断A`Z是否全部都有对应密文
    for(int i=0;i<26;i++){
    if(!m2.count('A'+i)){
        cout<<"Failed"<<endl;
        return 0;
    }
    }
    for(int i=0;i<strlen(translate);i++)
        cout<<m1[translate[i]];
        return 0;
}
```

THANKS FOR WATCHING