平衡二叉排序树－－－－红黑树（Red-Black Tree）

红黑树（Red Black Tree） 是一种自平衡二叉查找树，是一种高效的查找树。是在计算机科学中用到的一种数据结构，典型的用途是实现关联数组。

红黑树和AVL树类似，都是在进行插入和删除操作时通过特定操作保持二叉查找树的平衡，从而获得较高的查找性能。

它虽然是复杂的，但它的最坏情况运行时间也是非常良好的，并且在实践中是高效的： 它可以在O(log n)时间内做查找，插入和删除，这里的n 是树中元素的数目。

因为二叉树本身就是个递归的概念，所以在构建平衡二叉树的时候，应时刻记得递归这个概念。

了解平衡二叉树的平衡条件和调整策略有助于学习本节课的知识.

难点

- 对平衡条件的理解
- 对调整策略的理解

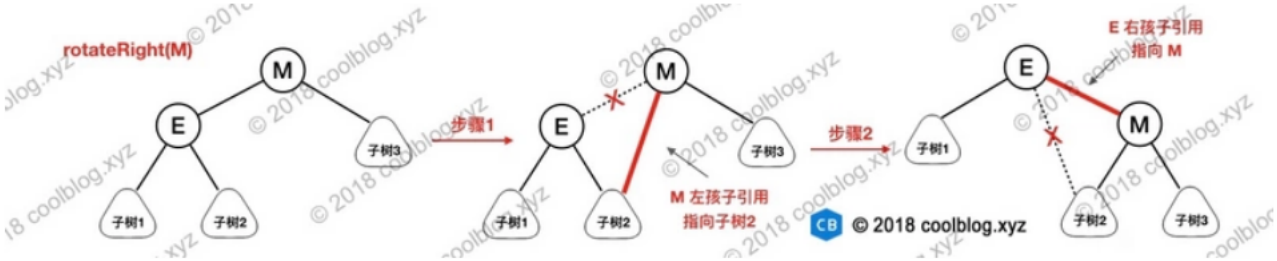
C++STL库中的map,set底层实现就是红黑树.红黑树的本质是平衡二叉树

树的旋转

在分析插入和删除操作前，这里需要插个队，先说明一下旋转操作，这个操作在后续操作中都会用得到。旋转操作分为左旋和右旋，左旋是将某个节点旋转为其右孩子的左孩子，而右旋是节点旋转为其左孩子的右孩子。这话听起来有点绕，所以还是请看下图：

上图包含了左旋和右旋的示意图，这里以右旋为例进行说明，右旋节点 M 的步骤如下：

1. 将节点 M 的左孩子引用指向节点 E 的右孩子
2. 将节点 E 的右孩子引用指向节点 M，完成旋转



上面分析了右旋操作，左旋操作与此类似，大家有兴趣自己画图试试吧，这里不再赘述了。旋转操作本身并不复杂，这里先分析到这吧。

红黑树的平衡条件:

1.每个节点非黑即红

2.根节点是黑色

3.叶节点是黑色

4.如果一个节点为红色，则它的两个子节点都是黑色.

5.从根节点出发到所有叶节点路径上，黑色节点数量相同.

红黑树的调整策略

１．插入调整站在祖父节点看

２．删除调整站在父节点看

３．插入删除的情况处理一共五种

插入调整

插入

红黑树的插入过程和二叉查找树插入过程基本类似，不同的地方在于，红黑树插入新节点后，需要进行调整，以满足红黑树的性质。性质1规定红黑树节点的颜色要么是红色要么是黑色，那么在插入新节点时，这个节点应该是红色还是黑色呢？答案是红色，原因也不难理解。如果插入的节点是黑色，那么这个节点所在路径比其他路径多出一个黑色节点，这个调整起来会比较麻烦。如果插入的节点是红色，此时所有路径上的黑色节点数量不变，仅可能会出现两个连续的红色节点的情况。

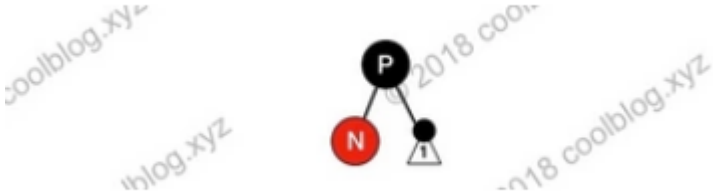
接下来，将分析插入红色节点后红黑树的情况。这里假设要插入的节点为 N，N 的父节点为 P，祖父节点为 G，叔叔节点为 U。插入红色节点后，会出现5种情况，分别如下：

情况一：

插入的新节点 N 是红黑树的根节点，这种情况下，我们把节点 N 的颜色由红色变为黑色，性质2满足。同时 N 被染成黑色后，红黑树所有路径上的黑色节点数量增加一个，性质5满足。

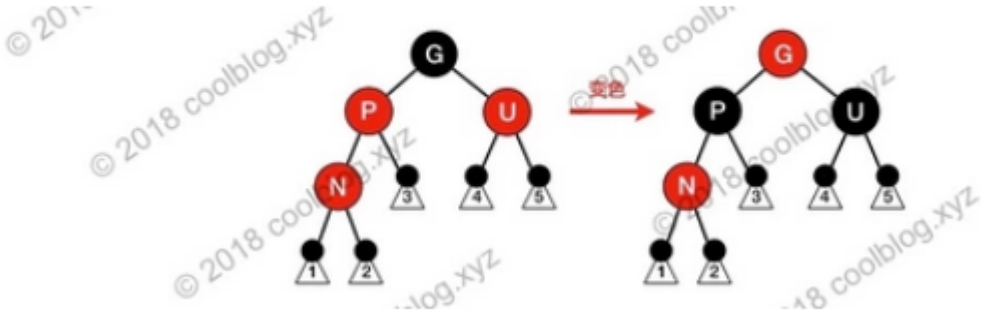
情况二：

N 的父节点是黑色，这种情况下，性质4和性质5没有受到影响，不需要调整。



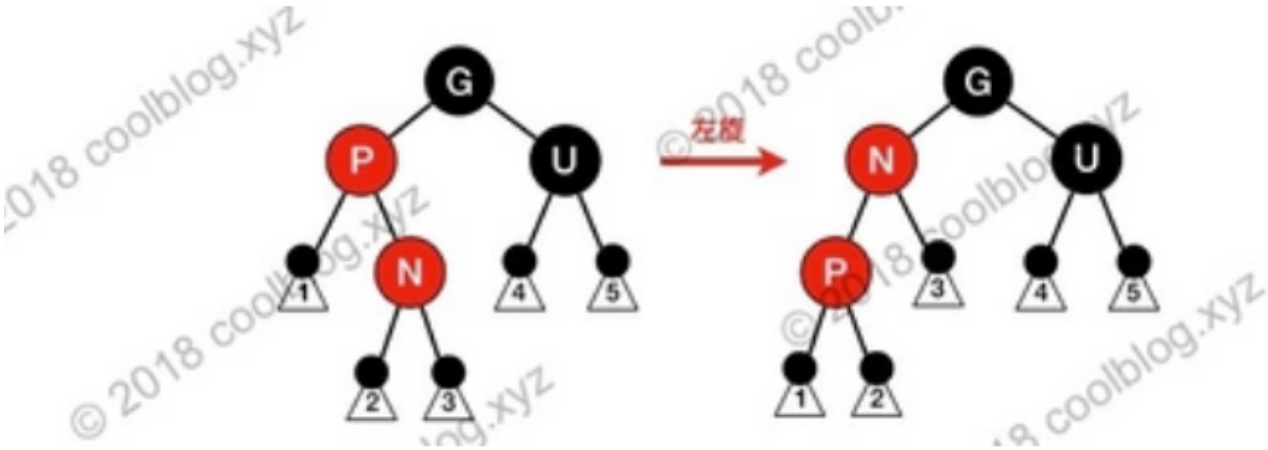
情况三：

N 的父节点是红色其祖父节点必然为黑色，叔叔节点 U 也是红色。由于 P 和 N 均为红色，所以递归插入到N时性质4不满足，此时需要回溯进行调整。这种情况下，先将 P 和 U 的颜色染成黑色，再将 G 的颜色染成红色。此时经过 G 的路径上的黑色节点数量不变，性质5仍然满足。但需要注意的是 G 被染成红色后，可能G和它的父节点形成连续的红色节点，此时需要向上继续调整。

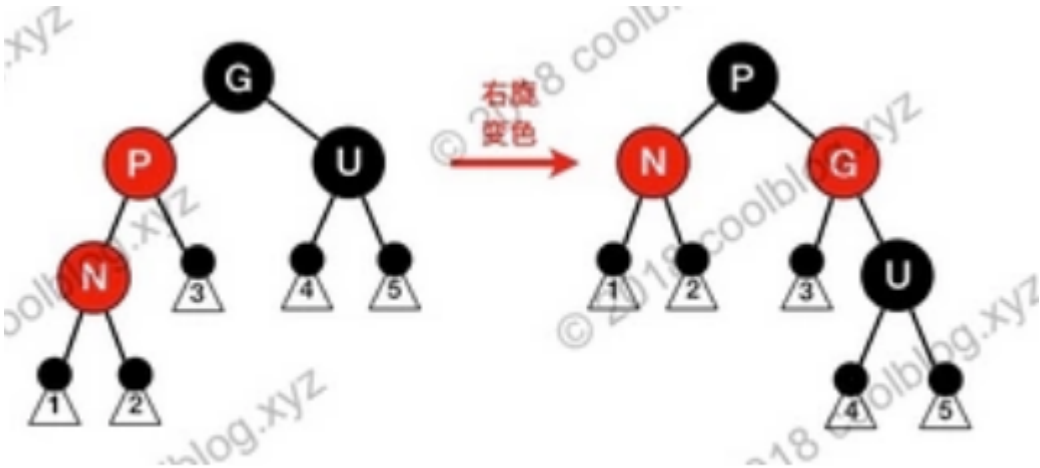


情况四：

N 的父节点为红色，叔叔节点为黑色。节点 N 是 P 的右孩子，且节点 P 是 G 的左孩子。此时先对节点 P 进行左旋，调整 N 与 P 的位置。接下来按照情况五进行处理，以恢复性质4。



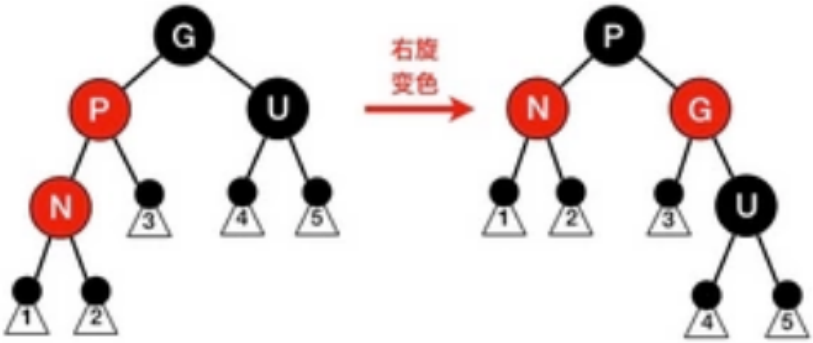
这里需要特别说明一下，上图中的节点 N 并非是新插入的节点。当 P 为红色时，P 有两个孩子节点，且孩子节点均为黑色，这样从 G 出发到各叶子节点路径上的黑色节点数量才能保持一致。既然 P 已经有两个孩子了，所以 N 不是新插入的节点。情况四是由以 N 为根节点的子树中插入了新节点，经过调整后，导致 N 被变为红色，进而导致了情况四的出现。考虑下面这种情况（PR 节点就是上图的 N 节点）：



如上图，插入节点 N 并按情况三处理。此时 PR 被染成了红色，与 P 节点形成了连续的红色节点，这个时候就需按情况四再次进行调整。

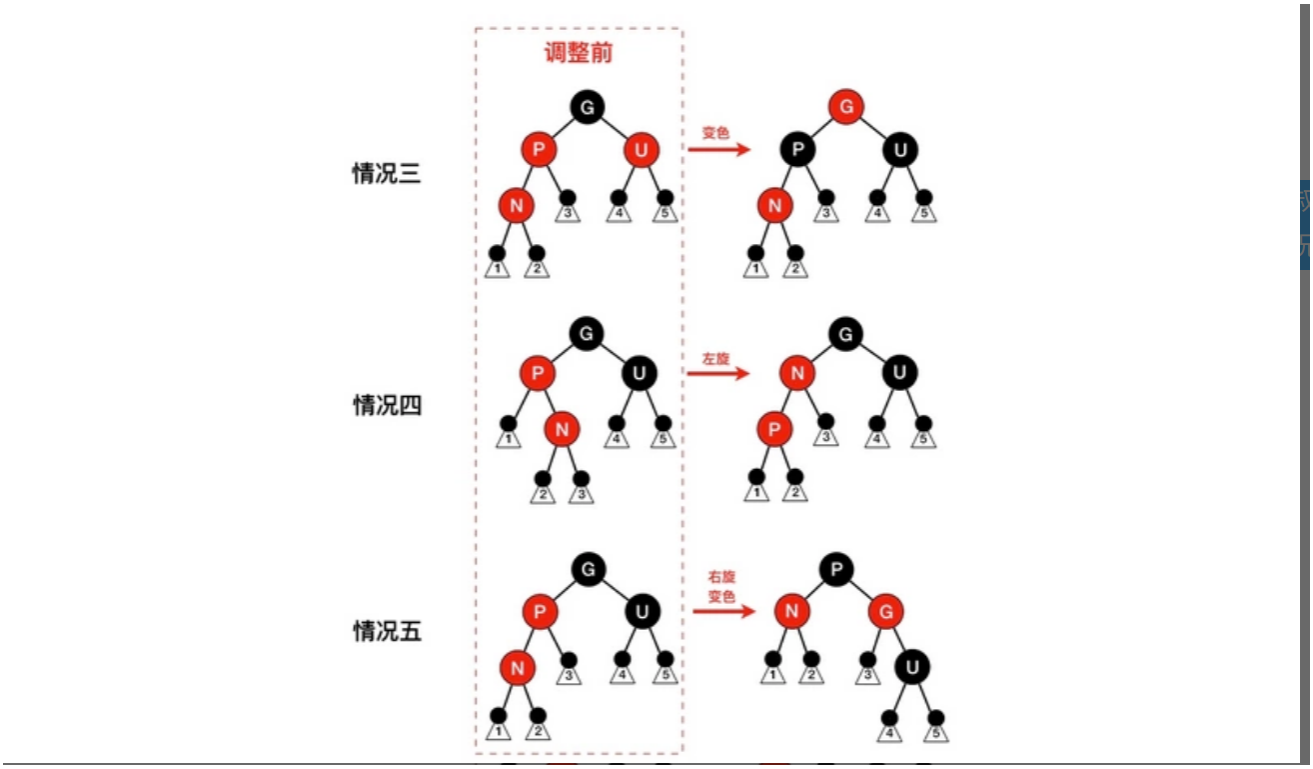
情况五：

N 的父节点为红色，叔叔节点为黑色。N 是 P 的左孩子，且节点 P 是 G 的左孩子。此时对 G 进行右旋，调整 P 和 G 的位置，并互换颜色。经过这样的调整后，性质4被恢复，同时也未破坏性质5。



插入总结

上面五种情况中，情况一和情况二比较简单，情况三、四、五稍复杂。但如果细心观察，会发现这三种情况的区别在于叔叔节点的颜色，如果叔叔节点为红色，直接变色即可。如果叔叔节点为黑色，则需要选选择，再交换颜色。当把这三种情况的图画在一起就区别就比较容易观察了，如下图：

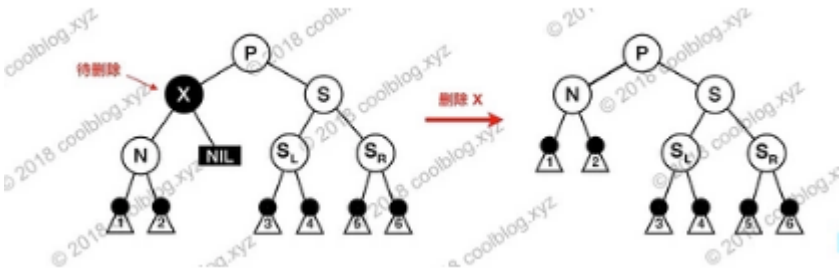


删除调整

红黑树的删除操作则要更为复杂一些。删除操作首先要确定待删除节点有几个孩子，如果有两个孩子，不能直接删除该节点。而是要先找到该节点的前驱（该节点左子树中最大的节点）或者后继（该节点右子树中最小的节点），然后将前驱或者后继的值复制到要删除的节点中，最后再将前驱或后继删除。由于前驱和后继至多只有一个孩子节点，这样我们就把原来要删除的节点有两个孩子的问题转化为只有一个孩子节点的问题，问题被简化了一些。我们并不关心最终被删除的节点是否是我们开始想要删除的那个节点，只要节点里的值最终被删除就行了。

红黑树删除操作的复杂度在于删除节点的颜色，当删除的节点是红色时，直接拿其孩子节点补空位即可。当删除的节点是黑色时，那么所有经过该节点的路径上的黑节点数量少了一个，破坏了性质5。如果该节点的孩子为红色，直接拿孩子节点替换被删除的节点，并将孩子节点染成黑色，即可恢复性质5。但如果孩子节点为黑色，处理起来就要复杂的多。分为6种情况，下面会展开说明。

在展开说明之前，我们先做一些假设，方便说明。这里假设最终被删除的节点为`X`（至多只有一个孩子节点），其孩子节点为`N`，`X`的兄弟节点为`S`，`S`的左节点为 SL，右节点为 SR。接下来讨论是建立在节点`X`被删除，节点`N`替换`X`的基础上进行的。这里说明把被删除的节点`X`特地拎出来说一下的原因是防止大家误以为节点`N`会被删除，不然后面就会看不明白。



情况一：

> 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。

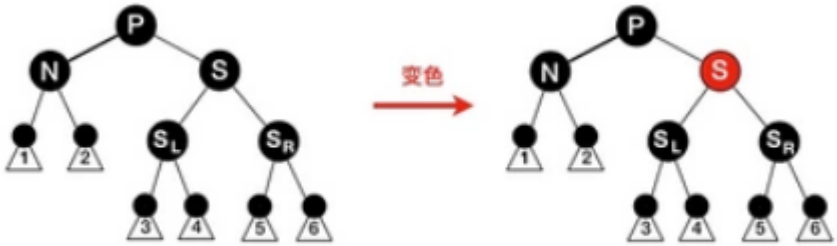
上面是维基百科中关于红黑树删除的情况一说明，由于没有配图，看的有点晕。经过思考，我觉得可能会是下面这种情形：

要删除的节点 X 是根节点，且左右孩子节点均为空节点，此时将节点 X 用空节点替换完成删除操作。

可能还有其他情形，大家如果知道，烦请告知。

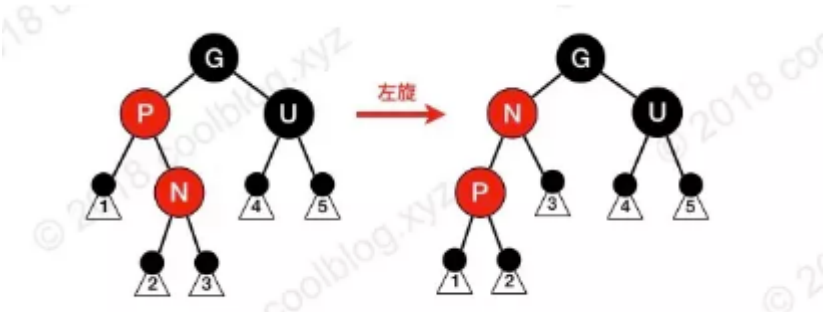
情况二：

S 为红色，其他节点为黑色。这种情况下可以对 N 的父节点进行左旋操作，然后互换 P 与 S 颜色。但这并未结束，经过节点 P 和 N 的路径删除前有3个黑色节点（`P -> X -> N`），现在只剩两个了（`P -> N`）。比未经过 N 的路径少一个黑色节点，性质5仍不满足，还需要继续调整。不过此时可以按照情况四、五、六进行调整。

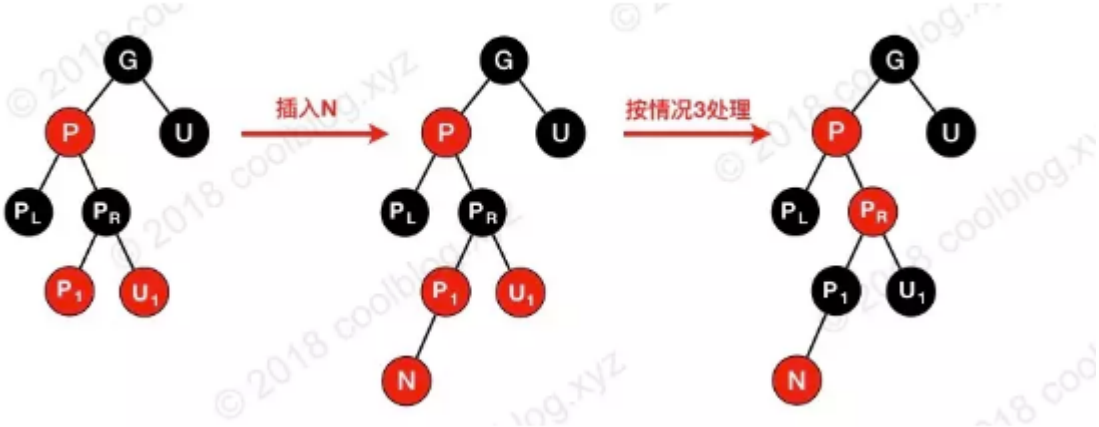


情况四：

N 的父节点是红色，S 和 S 孩子为黑色。这种情况比较简单，我们只需交换 P 和 S 颜色即可。这样所有通过 N 的路径上增加了一个黑色节点，所有通过 S 的节点的路径必然也通过 P 节点，由于 P 与 S 只是互换颜色，并不影响这些路径。

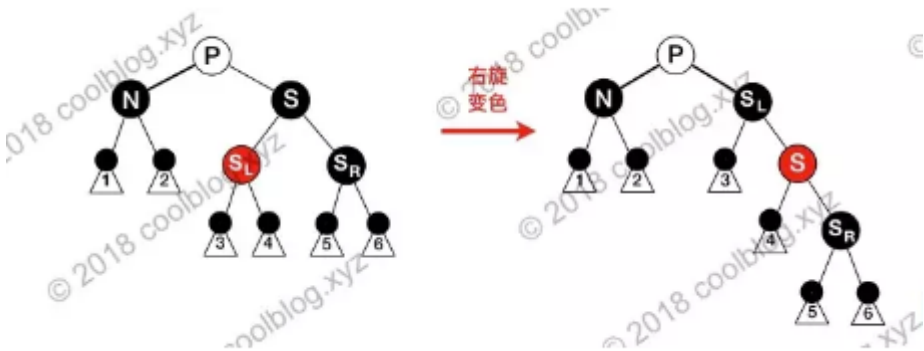


这里需要特别说明一下，上图中的节点 N 并非是新插入的节点。当 P 为红色时，P 有两个孩子节点，且孩子节点均为黑色，这样从 G 出发到各叶子节点路径上的黑色节点数量才能保持一致。既然 P 已经有两个孩子了，所以 N 不是新插入的节点。情况四是由以 N 为根节点的子树中插入了新节点，经过调整后，导致 N 被变为红色，进而导致了情况四的出现。考虑下面这种情况（PR 节点就是上图的 N 节点）：



情况五：

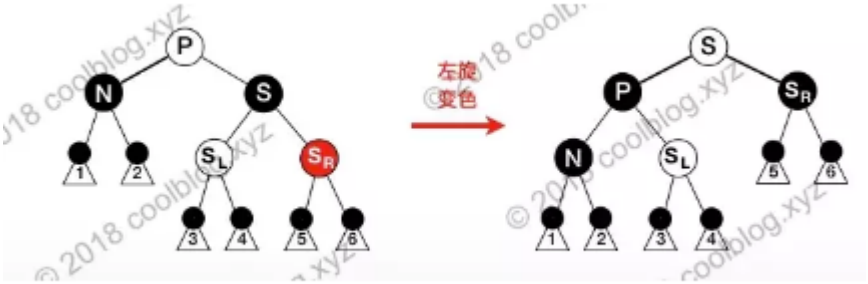
S 为黑色，S 的左孩子为红色，右孩子为黑色。N 的父节点颜色可红可黑，且 N 是 P 左孩子。这种情况下对 S 进行右旋操作，并互换 S 和 SL 的颜色。此时，所有路径上的黑色数量仍然相等，N 兄弟节点的由 S 变为了 SL，而 SL 的右孩子变为红色。接下来我们到情况六继续分析。



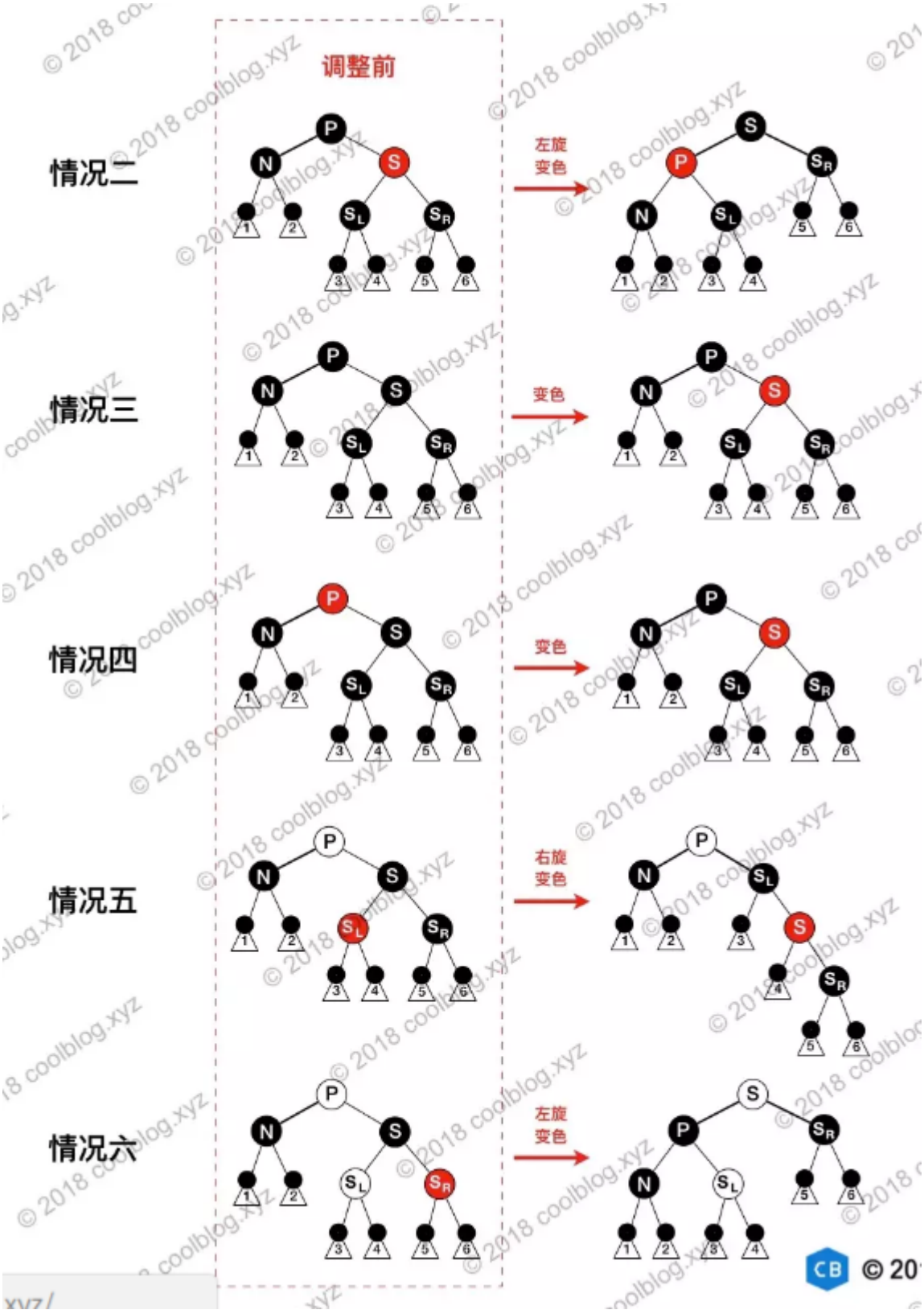
情况六：

S 为黑色，S 的右孩子为红色。N 的父节点颜色可红可黑，且 N 是其父节点左孩子。这种情况下，我们对 P 进行左旋操作，并互换 P 和 S 的颜色，并将 SR 变为黑色。因为 P 变为黑色，所以经过 N 的路径多了一个黑色节点，经过 N 的路径上的黑色节点与删除前的数量一致。对于不经过 N 的路径，则有以下两种情况：

1. 该路径经过 N 新的兄弟节点 SL ，那它之前必然经过 S 和 P。而 S 和 P 现在只是交换颜色，对于经过 SL 的路径不影响。
2. 该路径经过 N 新的叔叔节点 SR，那它之前必然经过 P、 S 和 SR，而现在它只经过 S 和 SR。在对 P 进行左旋，并与 S 换色后，经过 SR的路径少了一个黑色节点，性质5被打破。另外，由于 S 的颜色可红可黑，如果 S 是红色的话，会与 SR 形成连续的红色节点，打破性质4（每个红色节点必须有两个黑色的子节点）。此时仅需将 SR 由红色变为黑色即可同时恢复性质4和性质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。）。



删除总结



总结

红黑树是一种重要的二叉树，应用广泛，但在很多数据结构相关的书本中出现的次数并不多。很多书中要么不说，要么就一笔带过，并不会进行详细的分析，这可能是因为红黑树比较复杂的缘故。

代码演示

```c
#include <stdio.h>
#include <stdlib.h>
#define RED 0
#define BLACK 1
#define DOUBLE_BLACK 2

typedef struct Node {
    int key, color; // 0 red, 1 black, 2 double black
    struct Node *lchild, *rchild;
} Node;

Node _NIL, * const NIL = &_NIL;

__attribute__((constructor))
void init_NIL() {
    NIL->key = 0;
    NIL->lchild = NIL->rchild = NIL;
    NIL->color = BLACK;
    return ;
}

Node *getNewNode(int key) {
    Node *p = (Node *)malloc(sizeof(Node));
    p->key = key;
    p->lchild = p->rchild = NIL;
    p->color = RED;
    return p;
}

int hasRedChild(Node *root) {
    return root->lchild->color == RED || root->rchild->color == RED;
}

Node *left_rotate(Node *root) {
    Node *temp = root->rchild;
    root->rchild = temp->lchild;
    temp->lchild = root;
    return temp;
}

Node *right_rotate(Node *root) {
    Node *temp = root->lchild;
    root->lchild = temp->rchild;
    temp->rchild = root;
    return temp;
}

```

```
Node *insert_maintain(Node *root) {
    if (!hasRedChild(root)) return root;
    if (root->lchild->color == RED && root->rchild->color == RED) {
        if (!hasRedChild(root->lchild) && !hasRedChild(root->rchild)) return root;
        goto insert_end;
    }
    if (root->lchild->color == RED) {
        if (!hasRedChild(root->lchild)) return root;
        if (root->lchild->rchild->color == RED) {
            root->lchild = left_rotate(root->lchild);
        }
        root = right_rotate(root);
    } else {
        if (!hasRedChild(root->rchild)) return root;
        if (root->rchild->lchild->color == RED) {
            root->rchild = right_rotate(root->rchild);
        }
        root = left_rotate(root);
    }
    insert_end:
    root->color = RED;
    root->lchild->color = root->rchild->color = BLACK;
    return root;
}

Node *__insert(Node *root, int key) {
    if (root == NIL) return getNewNode(key);
    if (root->key == key) return root;
    if (root->key > key) root->lchild = __insert(root->lchild, key);
    else root->rchild = __insert(root->rchild, key);
    return insert_maintain(root);
}

Node *insert(Node *root, int key) {
    root = __insert(root, key);
    root->color = BLACK;
    return root;
}

Node *predeccessor(Node *root) {
    Node *temp = root->lchild;
    while (temp->rchild != NIL) temp = temp->rchild;
    return temp;
}

Node *erase_maintain(Node *root) {
    if (root->lchild->color != DOUBLE_BLACK && root->rchild->color != DOUBLE_BLACK)
return root;
    if (root->rchild->color == DOUBLE_BLACK) {
        if (root->lchild->color == RED) {
            root->color = RED;
            root->lchild->color = BLACK;
            root = right_rotate(root);
```

```
            root->rchild = erase_maintain(root->rchild);
            return root;
        }
        if (!hasRedChild(root->lchild)) {
            root->color += 1;
            root->lchild->color -= 1;
            root->rchild->color -= 1;
            return root;
        }
        if (root->lchild->lchild->color != RED) {
            root->lchild->rchild->color = BLACK;
            root->lchild->color = RED;
            root->lchild = left_rotate(root->lchild);
        }
        root->lchild->color = root->color;
        root->rchild->color -= 1;
        root = right_rotate(root);
        root->lchild->color = root->rchild->color = BLACK;
    } else {
        if (root->rchild->color == RED) {
            root->color = RED;
            root->rchild->color = BLACK;
            root = left_rotate(root);
            root->lchild = erase_maintain(root->lchild);
            return root;
        }
        if (!hasRedChild(root->rchild)) {
            root->color += 1;
            root->lchild->color -= 1;
            root->rchild->color -= 1;
            return root;
        }
        if (root->rchild->rchild->color != RED) {
            root->rchild->lchild->color = BLACK;
            root->rchild->color = RED;
            root->rchild = right_rotate(root->rchild);
        }
        root->rchild->color = root->color;
        root->lchild->color -= 1;
        root = left_rotate(root);
        root->lchild->color = root->rchild->color = BLACK;
    }
    return root;
}

Node *__erase(Node *root, int key) {
    if (root == NIL) return root;
    if (root->key > key) {
        root->lchild = __erase(root->lchild, key);
    } else if (root->key < key) {
        root->rchild = __erase(root->rchild, key);
    } else {
        if (root->lchild == NIL || root->rchild == NIL) {
```

```
            Node *temp = root->lchild == NIL ? root->rchild : root->lchild;
            temp->color += root->color;
            free(root);
            return temp;
        } else {
            Node *temp = predeccessor(root);
            root->key = temp->key;
            root->lchild = __erase(root->lchild, temp->key);
        }
    }
    return erase_maintain(root);
}

Node *erase(Node *root, int key) {
    root = __erase(root, key);
    root->color = BLACK;
    return root;
}

Node *erase (Node *root, int key) {
    root = __ erase(root, key) ;
    root ->color = BLACK;
    return root;

}

void clear (Node *root ,int key ) {
    root = __ erase (root ,key ) {
        root->color =BLACK;
        return root;
    }
}



void clear(Node *root) {
    if (root == NIL) return ;
    clear(root->lchild);
    clear(root->rchild);
    free(root);
    return ;
}

void output(Node *root) {
    if (root == NIL) return ;
    printf("%d [%d, %d] %s\n",
           root->key,
           root->lchild->key,
           root->rchild->key,
           root->color ? "BLACK" : "RED"
          );
    output(root->lchild);
    output(root->rchild);
```

```
    return ;
}

int main() {
    int op, val;
    Node *root = NIL;
    while (~scanf("%d%d", &op, &val)) {
        switch (op) {
            case 1: root = insert(root, val); break;
            case 2: root = erase(root, val); break;
        }
        output(root);
    }
    return 0;
}

```