## 红黑树

属于平衡二叉排序树

### 一、性质

1. 节点非黑即红
2. 根节点是黑色
3. 叶节点(虚拟空节点)(NIL)是黑色
4. **如果一个节点是红色那么他的两个子节点都是黑色的**
5. **从根节点出发到所有叶节点路径上,黑色节点数量相同**

#### 平衡条件的认识

4、5 是关键性质,共同决定了在红黑树中最长路径可以是最短路径的两倍(最长边一黑一红,最短边都是黑的,这里是不计算NIL节点的)红黑树本质上也是通过树高来控制平衡,只不过比起 AVL 树比起来说,红黑树树高控制条件更加松散一些,所以红黑树在插入和删除节点之后需要平衡调整的概率更小。

### 二、调整策略

1. 插入调整站在**祖父节点**向下看两层,**发现发生冲突则进行调整**
2. 删除调整站在**父节点**向下看
3. 插入和删除的情况处理一共五种情况
4. 把每一种情况想象成一颗大的红黑树当中的一颗子树
5. 局部调整时,为了不影响全局,要保证调整前后每条路径上的黑色节点数量相同

#### 插入的调整策略

**插入调整最重要的是干掉双红的情况**

插入的新节点一定是**红色**,因为如果插入新节点是黑色则就违反了性质 5,必然需要调整(**插入黑色必然调整,插入红色可能需要调整**)

**情况 1**(LL、LR、RR、RL)

1. 祖父节点是黑色
2. 祖父节点的两个节点都是红色节点
3. 插入新节点是红色

归纳之后可以得到相同的调整方法

调整策略:

1. 把父节点和叔叔节点调成黑色
2. 祖父节点调整成红色
3. 这样调整保证这个子树的所有路径上的黑色节点数量不变,不会影响整颗树路径上黑色节点的数量

简单表示:  
黑[红、红] -> 红[黑、黑]  
红色上顶(维持每条链路上红色节点数量不变)

**情况 2**(LL、LR、RR、RL)  
插入之前:  
黑[红、黑]  
在红色的节点下面又插入一个红色节点导致失衡,这就出现了 LL 型失衡(左孩子红色,左孩子的左孩子又是红色),调整方法就是抓着祖父节点来一个大右旋,右旋之后进行颜色调整,然后将祖父节点[父节点、父节点]改成 黑[红、红] 或者 红[黑、黑]都行  
LR 类型失衡先对父节点小左旋再对祖父节点进行大右旋  
另外两种情况也是对称的调整方式

**总结**

1. 叔父节点红色,修改小帽子,改为红黑黑
2. 叔叔节点是黑色, 参考 AVL 树的失衡情况,分成 LL、LR、RR、RL 先参考 AVL 树调整策略, 然后再调整三元组策略,然后再修改三元组的颜色,有两种调整策略:红色上浮、红色下沉
3. 两大类情况,包含 8 种小情况

**代码演示**

1. 插入调整,发生在递归的回溯阶段
2. 插入调整代码中,使用 goto 语句,8 行代码,变成了 4 行
3. 处理根节点一定是黑色,通过代码封装,insert -> \_\_insert

#### 删除的调整策略

**主要就是为了干掉双重黑**  
发生的前提:

1. 删除红色节点不会影响平衡
2. 度为 1 的黑色节点,唯一孩子,颜色一定是红色
3. 删除度为 1 的节点,不会产生平衡调整
4. 删除度为 0 的黑色节点,会产生一个双重黑的 NIL 节点
5. 删除调整,就是为了干掉双重黑

**情况 1**  
双重黑的兄弟是黑色,并且双重黑的兄弟的两个孩子都是黑色  
调整方法:

1. 双重黑节点的父节点加一重黑色
2. 双重黑节点和兄弟节点减一重黑色(即兄弟节点变成红色)

**情况 2**  
双重黑节点的兄弟是黑色,兄弟节点同方向的兄弟节点的孩子是红色(LL、RR)  
调整方法:

1. 抓着双重黑节点的父节点进行旋转(LL(右旋)、RR(左旋))
2. 选择之后把双重黑的父节点以及父节点的兄弟节点都改成黑色,双重黑的祖父节点(旋转之前的兄弟节点)改成原有的根节点的颜色
3. 双黑节点减一重黑色

**情况 3**  
双重黑节点的兄弟是黑色,兄弟节点同方向的兄弟节点的孩子节点是黑色,不同方向的兄弟节点的孩子节点是红色(LR、RL)  
调整方法:

1. 先抓着双重黑节点的兄弟节点进行左(右)旋
2. 把旋转后的兄弟节点变黑,再把原来的兄弟节点变红
3. 经过 2 的处理变成了**情况 2**去处理

**情况 4**  
双黑节点的兄弟节点是红色(这种情况下红色的兄弟一定有两层黑色的孩子)  
调整方法:

1. 将兄弟节点旋转成根节点
2. 将新根节点变成黑色
3. 原根节点变成红色
4. 这时站在原根节点向下看就变成双重黑的兄弟节点是黑色接下来就可以按**情况 1、2、3**去处理

### 三、代码注意要点

判断是 LL、RR 还是 LR、RL 时要判断子节点不是红色,因为 NIL 可能是双重黑

**红黑树源码如下：**

**attribute**可以设置函数属性(Function Attribute)、变量属性(Variable Attribute)和类型属性(Type Attribute)。

**attribute**前后都有两个下划线，并且后面会紧跟一对原括弧，括弧里面是相应的**attribute**参数  
**attribute**语法格式为：attribute ( ( attribute-list ) )

如果函数被设定为**constructor属性**，**则该函数会在main（）函数执行之前被自动的执行**；若函数被设定为**destructor属性**，**则该函数会在main（）函数执行之后或者exit（）被调用后被自动的执行。**

### 四、红黑树源码

#include <stdlib.h>  
#include <stdio.h>  
  
#声明节点结构  
typedef struct Node {  
 int key;  
 int color; // 0 red, 1 black, 2 double black  
 struct Node \*lchild, \*rchild;  
} Node;  
  
Node \_\_NIL;  
#define NIL (&\_\_NIL)  
\_\_attribute\_\_((constructor))  
void init\_NIL() {  
 NIL->key = 0;  
 NIL->color = 1;  
 NIL->lchild = NIL->rchild = NIL;  
 return ;  
}  
  
#申请新节点，返回指针  
Node \*getNewNode(int key) {  
 Node \*p = (Node \*)malloc(sizeof(Node));  
 p->key = key;  
 p->color = 0;  
 p->lchild = p->rchild = NIL;  
 return p;  
}  
  
#判断子节点是否为红  
int has\_red\_child(Node \*root) {  
 return root->lchild->color == 0 || root->rchild->color == 0;  
}  
  
#左旋  
Node \*left\_rotate(Node \*root) {  
 Node \*temp = root->rchild;  
 root->rchild = temp->lchild;  
 temp->lchild = root;  
 return temp;  
}  
  
#右旋  
Node \*right\_rotate(Node \*root) {  
 Node \*temp = root->lchild;  
 root->lchild = temp->rchild;  
 temp->rchild = root;  
 return temp;  
}  
  
Node \*insert\_maintain(Node \*root) {  
 if (!has\_red\_child(root)) return root;  
 int flag = 0;  
 if (root->lchild->color == 0 && root->rchild->color == 0) goto insert\_end;  
 if (root->lchild->color == 0 && has\_red\_child(root->lchild)) flag = 1;  
 if (root->rchild->color == 0 && has\_red\_child(root->rchild)) flag = 2;  
 if (flag == 0) return root;  
 if (flag == 1) {  
 if (root->lchild->rchild->color == 0) {  
 root->lchild = left\_rotate(root->lchild);  
 }  
 root = right\_rotate(root);  
 } else {  
 if (root->rchild->lchild->color == 0) {  
 root->rchild = right\_rotate(root->rchild);  
 }  
 root = left\_rotate(root);  
 }  
insert\_end:  
 root->color = 0;  
 root->lchild->color = root->rchild->color = 1;  
 return root;  
}  
  
Node \*\_\_insert(Node \*root, int key) {  
 if (root == NIL) return getNewNode(key);  
 if (root->key == key) return root;  
 if (key < root->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 = 1;  
 return root;  
}  
  
Node \*predecessor(Node \*root) {  
 Node \*temp = root->lchild;  
 while (temp->rchild != NIL) temp = temp->rchild;  
 return temp;  
}  
  
Node \*erase\_maintain(Node \*root) {  
 if (root->lchild->color != 2 && root->rchild->color != 2) return root;  
 if (has\_red\_child(root)) {  
 int flag = 0;  
 root->color = 0;  
 if (root->lchild->color == 0) {  
 root = right\_rotate(root); flag = 1;  
 } else {  
 root = left\_rotate(root); flag = 2;  
 }  
 root->color = 1;  
 if (flag == 1) root->rchild = erase\_maintain(root->rchild);  
 else root->lchild = erase\_maintain(root->lchild);  
 return root;  
 }  
 if ((root->lchild->color == 2 && !has\_red\_child(root->rchild)) ||  
 (root->rchild->color == 2 && !has\_red\_child(root->lchild))) {  
 root->lchild->color -= 1;  
 root->rchild->color -= 1;  
 root->color += 1;  
 return root;  
 }  
 if (root->lchild->color == 2) {  
 if (root->rchild->rchild->color != 0) {  
 root->rchild->color = 0;  
 root->rchild = right\_rotate(root->rchild);  
 root->rchild->color = 1;  
 }  
 root = left\_rotate(root);  
 root->color = root->lchild->color;  
 } else {  
 if (root->lchild->lchild->color != 0) {  
 root->lchild->color = 0;  
 root->lchild = left\_rotate(root->lchild);  
 root->lchild->color = 1;  
 }  
 root = right\_rotate(root);  
 root->color = root->rchild->color;  
 }  
 root->lchild->color = root->rchild->color = 1;  
 return root;  
}  
  
Node \*\_\_erase(Node \*root, int key) {  
 if (root == NIL) return NIL;  
 if (key < root->key) {  
 root->lchild = \_\_erase(root->lchild, key);  
 } else if (key > root->key) {  
 root->rchild = \_\_erase(root->rchild, key);  
 } else {  
 if (root->lchild == NIL || root->rchild == NIL) {  
 Node \*temp = root->lchild != NIL ? root->lchild : root->rchild;  
 temp->color += root->color;  
 free(root);  
 return temp;  
 } else {  
 Node \*temp = predecessor(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 = 1;  
 return root;  
}  
  
void clear(Node \*root) {  
 if (root == NIL) return ;  
 clear(root->lchild);  
 clear(root->rchild);  
 free(root);  
 return ;  
}  
  
void print(Node \*root) {  
 printf("(%d| %d, %d, %d)\n",  
 root->color, root->key,  
 root->lchild->key,  
 root->rchild->key  
 );  
 return ;  
}  
  
void output(Node \*root) {  
 if (root == NIL) return ;  
 print(root);  
 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);  
 printf("------------\n");  
 }  
 return 0;  
}