HW5

522031910213 朱涵

March 29, 2024

1 问题一

1.1

如下图是插入后的红黑树。

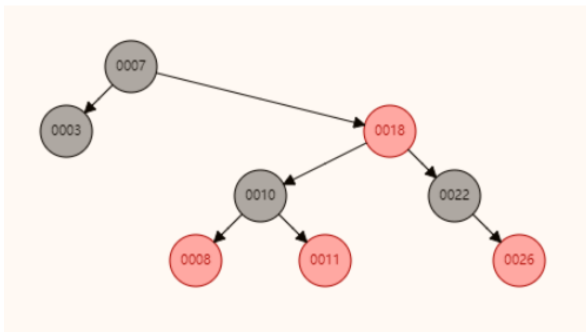

Figure 1: 插入后的红黑树

1.2

如下图是修改后的inorder代码以及编译后的结果

```
void RedBlackTree::inorderUtil(Node *root)
{
    // Inorder traversal logic
    if (root == nullptr)
        return;
    inorderUtil(root->left);
    std::cout << root->data << " (" << (root->color == RED ? "RED" : "BLACK") << ") ";
    inorderUtil(root->right);
}
```

Figure 2: inorder代码

```
Inorder traversal of the constructed tree:
3 (BLACK) 7 (BLACK) 8 (RED) 10 (BLACK) 11 (RED) 18 (RED) 22 (BLACK) 26 (RED)
```

Figure 3: 程序结果

可以看到画出的红黑树和中序遍历的结果是一致的。

2 问题二

删除操作的分类参考了课本上的根据颜色情况进行分类。 如下图，x的删除遵循了BST的删除模版，先和x关键码的后继节点进行交换位置之后再进行删除，因此实际被删除节点x的左子树必然为空（外部节点），并且交换前后红黑树的性质未遭到破坏。 接下来先按照x的颜色进行分类。若x为红色，删除x并用x的右子树r进行替代后，红黑树的性质仍保持，无需修改； 若x为黑色，继续依照r的颜色进行分类：若r为红色，发现删除x后会导致**局部子树的黑深度减少一**。为了恢复黑深度的性质，需要把r染为黑色，这样红黑树的性质就得到了保持；若r为黑色，则需要进一步分类讨论。

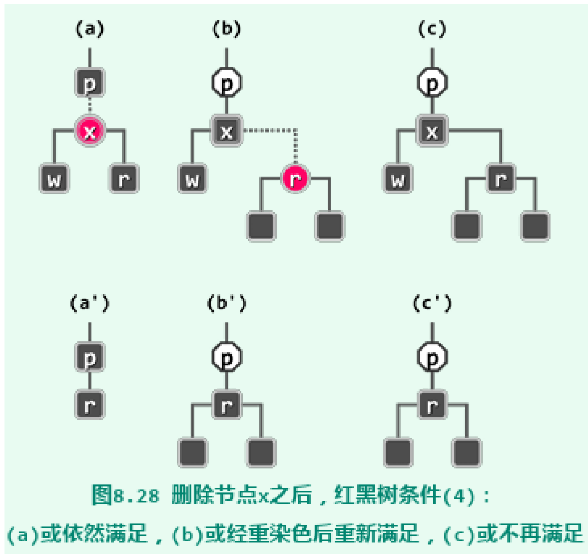

Figure 4: 初步分类

不难发现，p的子树x的黑深度至少为2，则x的兄弟子树的黑深度也至少为2，yinc想的兄弟s一定非空。 继续在x，r皆为黑色的情况（即书上所谓“双黑”）下进行分类，分类依据是x兄弟节点s，x父亲p以及s的子节点颜色情况。因为课本上主要通过转换为等价B树进行理解，我会尝试以**如何恢复红黑树的性质**的另一角度阐述自己的理解。 BB-1：s为黑色，且s至少有一个红色孩子节点。 此时发现删除x会导致**x子树的黑深度减少1**，而s子树不受影响。为了让x子树的黑深度恢复的同时又不影响s子树的黑深度，尝试使用已知的红色节点t进行修改。发现删除x后p子树有些**向左倾斜**，不妨先进行一个右旋恢复平衡。右旋之后，发现x所在子树的黑深度恢复了！那么只需要调整另一个子树即可。显然我们可以把t染黑，至此整个子树的黑深度性质恢复完成。

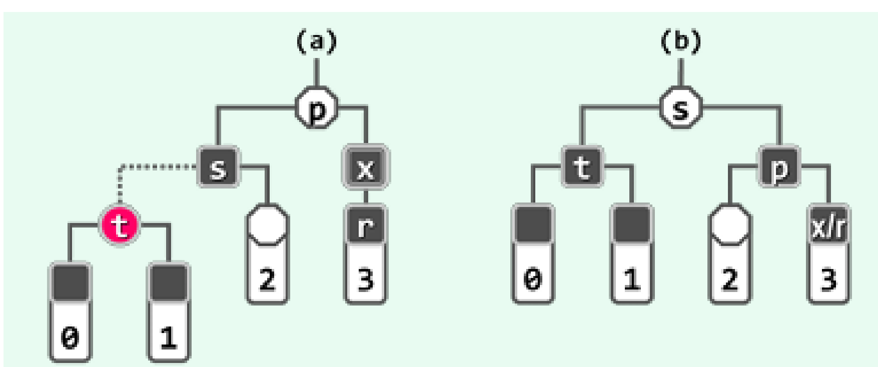

Figure 5: BB-1

BB-2-R：s为黑色，且s没有红色孩子节点，且p为红色。 同理，删除x后子树**黑深度减少**。我们还是尝试利用已知的红色节点进行调整，为了让x子树黑深度加1，显然把p染黑即可。不过这样就导致了**s子树黑深度加一**，不过现在s的父亲是黑色的了，不妨利用这点，把s染红，s子树的黑深度又恢复了，同时整棵子树的黑深度也未改变。

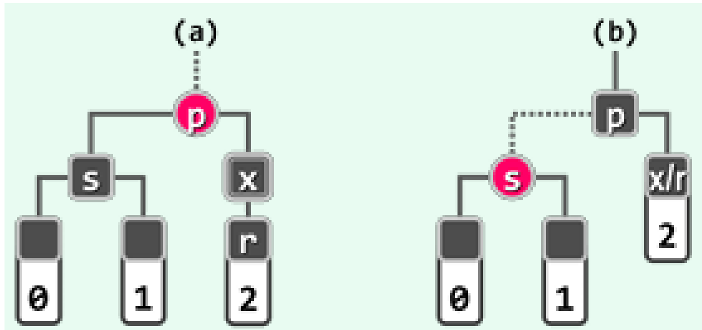

Figure 6: BB-2-R

BB-2-B：s为黑色，且s没有红色孩子节点，且p为黑色。 删除x后仍导致子树**黑深度减少**，但是我们没有可以利用的红色节点了。无奈之下，我们必须**选一个节点染红**。我们不能选未知颜色的子树节点，因为有可能它本来就是红色的；我们也不能选已知颜色的p，因为p的父亲有可能是红色而导致双红。因此我们只能选择**染红s**，而s是可以被染红的。染红之后左右子树的黑深度同时减少，恢复了平衡，但是同时导致了**p子树的黑深度减少**。这也就是B树中的下溢情况，需要递归的进行向上调整，继续向p的父亲进行恢复操作。

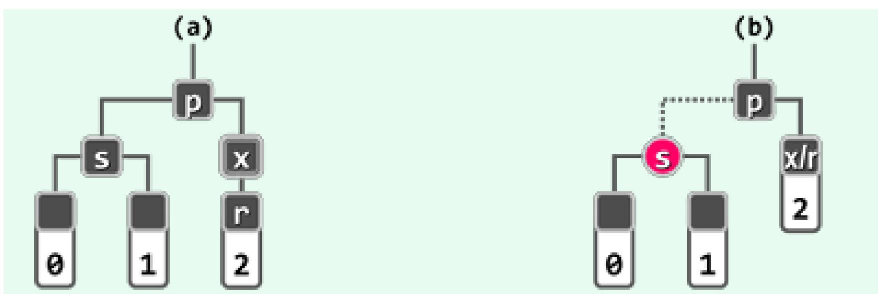

Figure 7: BB-2-B

BB-3：s为红色，因此p必须为黑色，s的孩子节点也为黑色。 此时删除x导致子树黑深度减少。因为x的兄弟是红色的，现在很难进行调整，不过我们可以把子树进行右旋一下恢复平衡性，同时把s和p的颜色进行交换，如此处理之后发现x的兄弟节点此时必然是**黑色**的。这就意味着我们可以使用**BB-1或者BB-2**的方法处理此时的p子树（左子树的黑深度未改变，无需处理），且p是红色的，也就是说唯一需要进行递归处理的BB-2-B是不可能出现的。这样我们就完成了红黑树性质的恢复。

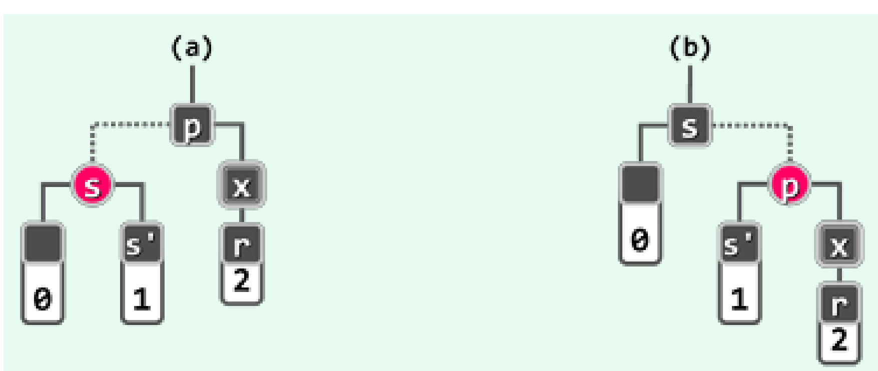

Figure 8: BB-3

3 问题三

在补充了fixViolation函数之后，我用test函数向红黑树中分别以顺序和乱序插入了0-9999共10000个元素。其中乱序插入执行了3000次以计算平均值。其中失衡次数以双红修复发生的次数为准，一次双红递归导致的向上多次修复算作一次失衡。程序运行结果如下图：

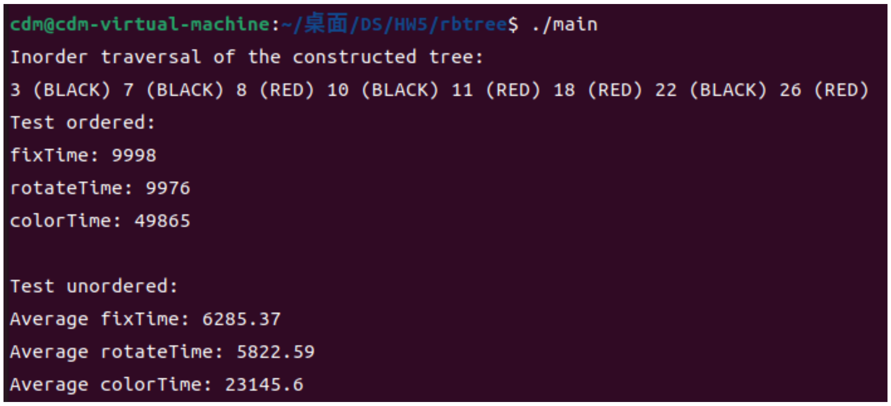

Figure 9: 测试结果

统计表如下图：

	失衡次数	染色次数	旋转次数
顺序插入	9998	49865	9976
乱序插入（平均值）	6285	23146	5823

Table 1: 实验结果统计表

可以得到结论：**乱序插入的失衡次数、染色次数以及旋转次数明显比顺序插入要低。**具体原因可能是顺序插入会导致红黑树的结构经常失衡，且红黑树的性质经常被破坏，需要频繁进行旋转和染色操作来恢复。这与我们的预先设想是符合的。