Project 3\_3

AVL树→红黑树问题

班级：无53

姓名：陈相宁

学号：2015011033

Email：[c.xiangning1997@gmail.com](mailto:c.xiangning1997@gmail.com)

**问题描述：**

在Windows的虚拟内存管理中，将VAD组织成AVL树。VAD树是一种平衡二叉树。红黑树也是一种自平衡二叉查找，在Linux2.6及其以后版本的内核中，采用红黑树来维护内存块。

请尝试参考Linux源代码将WRK源代码中的VAD树由AVL树替换成红黑树。

1. **实验思路：**
2. **红黑树（RB-Tree）学习：[[1]](#footnote-1)**

* 性质：

1. 每个节点为红色或黑色
2. 根节点是黑色的
3. 每个叶子节点（树尾端NULL节点）是黑色的
4. 如果一个节点是红色，那么其俩儿子都是黑色的
5. 对任一节点，其到叶子节点的每一条路径都包含相同数目的黑色节点

* 红黑树的插入和插入修复：

1. 插入根节点直接将根节点涂为黑色即可
2. 插入节点父节点是黑色，则什么都不用做
3. 插入节点父节点是红色，而祖父节点的另一儿子（叔叔节点）是红色，需插入修复
4. 插入节点父节点是红色，叔叔节点是黑色，需插入修复

* 红黑树的删除和删除修复：

涉及步骤较为复杂，此处不列举详细情况。

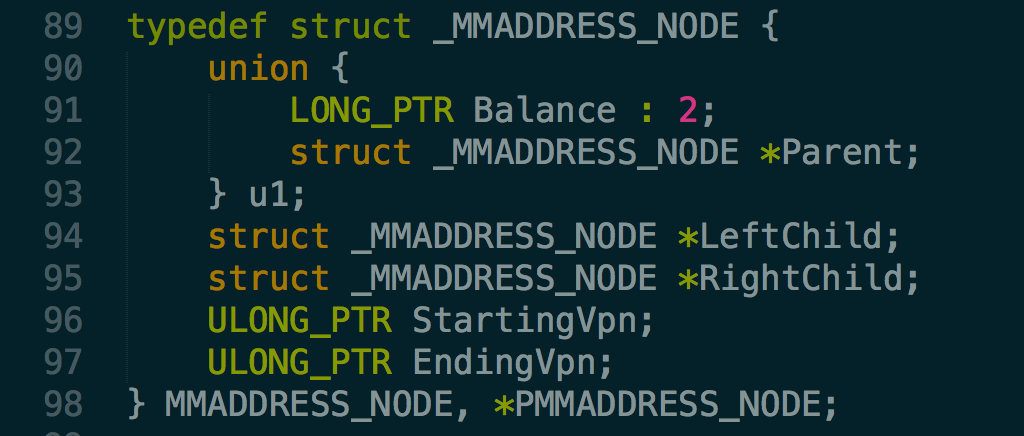
1. **AVL树与红黑树（RB-Tree）对比：**

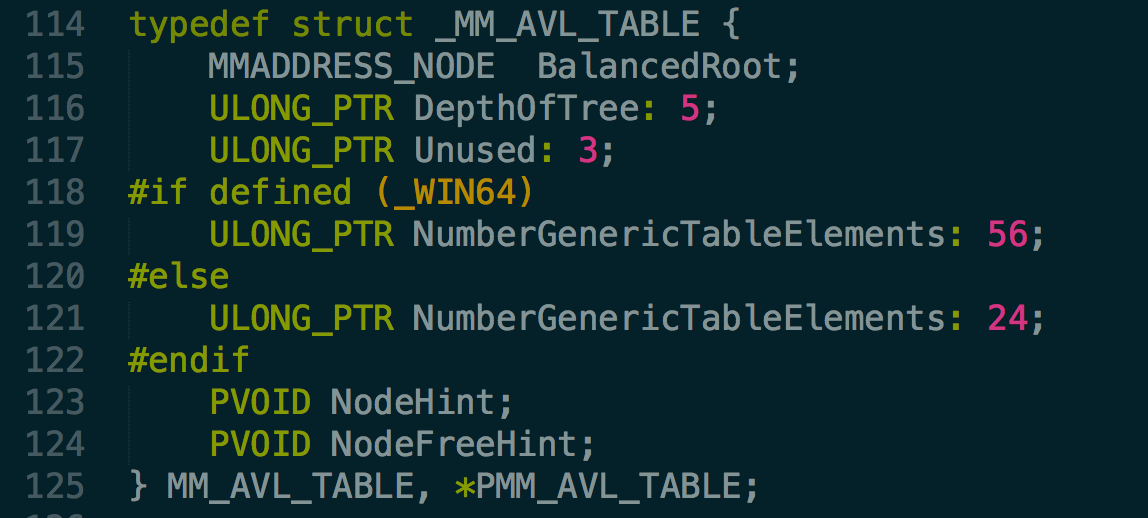
* 如果插入一个node引起了树的不平衡，AVL和RB-Tree都是最多只需要2次旋转操作，即两者都是O(1)；但是在删除node引起树的不平衡时，最坏情况下，AVL需要维护从被删node到root这条路径上所有node的平衡性，因此需要旋转的量级O(logN)，而RB-Tree最多只需3次旋转，只需要O(1)的复杂度。
* AVL的结构相较RB-Tree来说更为平衡，在插入和删除node更容易引起Tree的unbalance，因此在大量数据需要插入或者删除时，AVL需要rebalance的频率会更高。因此，RB-Tree在需要大量插入和删除node的场景下，效率更高。但是，由于AVL高度平衡，因此AVL的search效率更高。

所以总体来说，AVL树与红黑树（RB-Tree）各有利弊，但日常使用下总体统计性能RB-Tree应该是略高于AVL树的。

1. **WRK中的AVL树：**

在wrk-v1.2/base/inc/ps.h中，AVL树及节点定义如下：



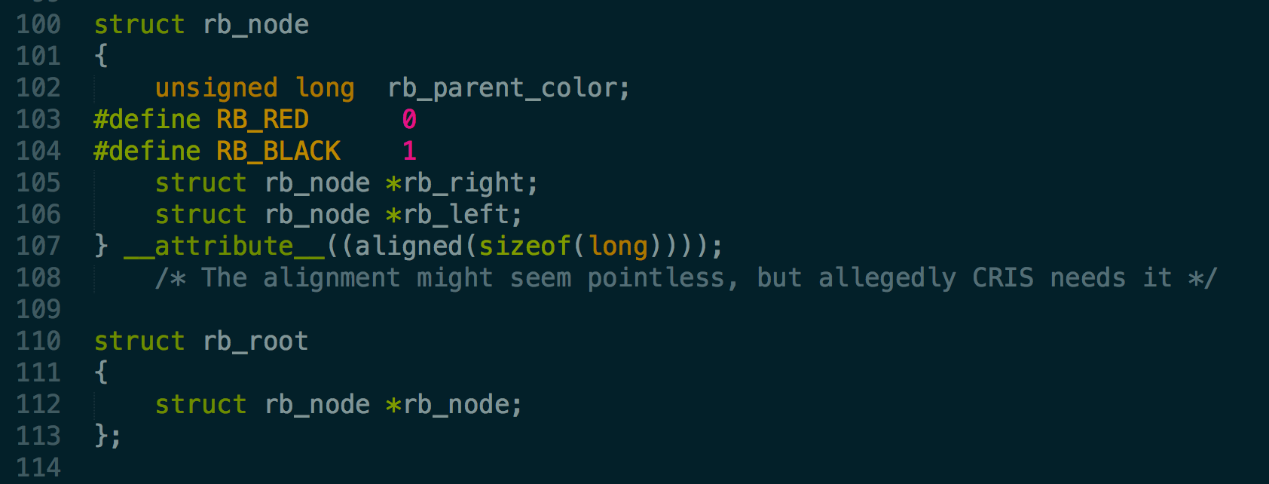


函数具体实现在wrk-v1.2/base/ntos/mm/addrsup.c中。其中需要注意的是，AVL树真正的root是BalanceRoot的RightChild，而LeftChild是NULL。

1. **Linux中的红黑树：**

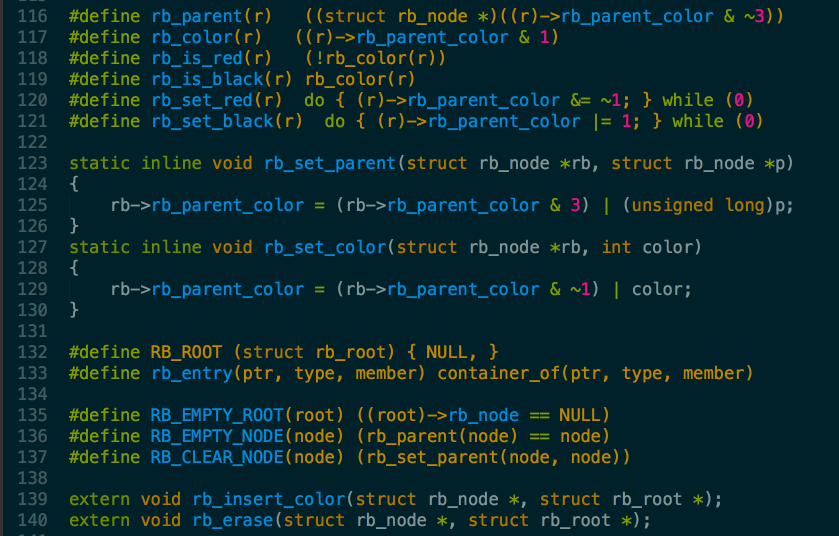
选取Linux-2.6版本。在rbtree.c以及rbtree.h中，红黑树有关操作如下：

红黑树节点定义如下：



其中，rb\_root较特殊，该指针在移植时应特别注意。

一些基本操作接口如下：



供外界使用的主要有获取父节点（rb\_parent）、颜色（rb\_color），判断节点颜色（rb\_is\_red、rb\_is\_black），设置节点颜色（rb\_set\_color、rb\_set\_black）以及对本次实验最为重要的2个外部函数rb\_insert\_color用于插入修复以及rb\_erase用于删除修复。

1. **代码实现：**

本次实验将AVL树转变为红黑树，因此只需将rbtree.c以及rbtree.h中的方法移植到addrsup.c中即可。

为了便于移植做出的几处更改：

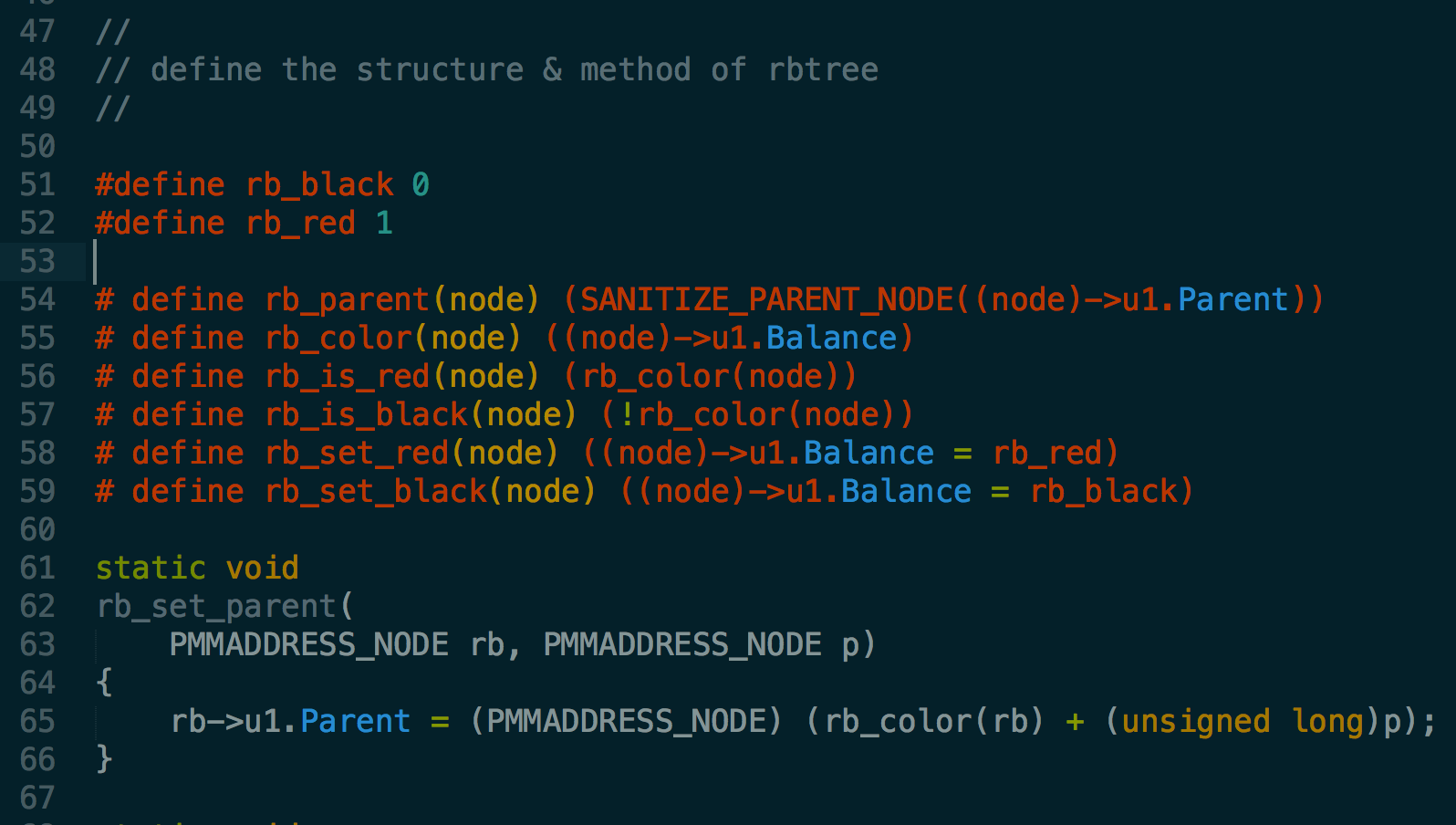
\\Mac\Home\Desktop\屏幕快照 2017-12-06 17.19.40.png

1. 由rb\_color可见，rb\_node最低位表示颜色，为了避免指针带来的不必要错误，用PMMADDRERSS的Balance表示节点颜色。
2. AVL树初始化时，Balance为0，但是红黑树初始化时根节点为黑色。因此为了尽量降低出错概率，将红色定义为1，黑色定义为0，这与rb\_tree.h中的定义正好相反。

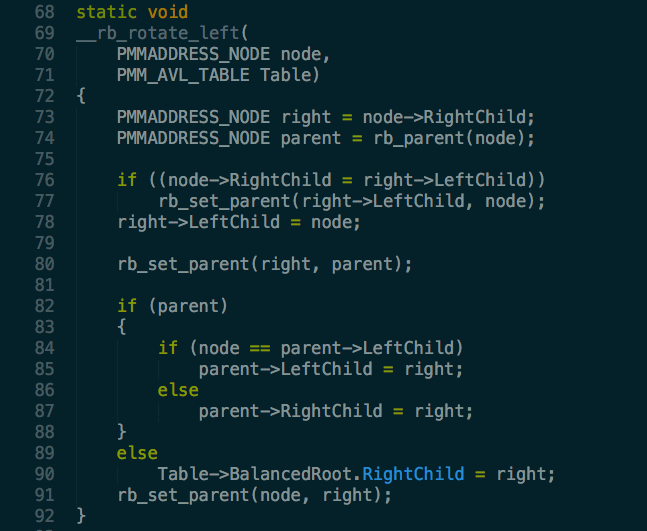
针对若干简单函数，只需将：

1. rb\_node \*替换为PMMADDRESS\_NODE
2. rb\_root \*替换为PMM\_AVL\_TABLE
3. rb\_left替换为LeftChild
4. rb\_right替换为RightChild
5. root替换为root→BalancedRoot.RightChild即可

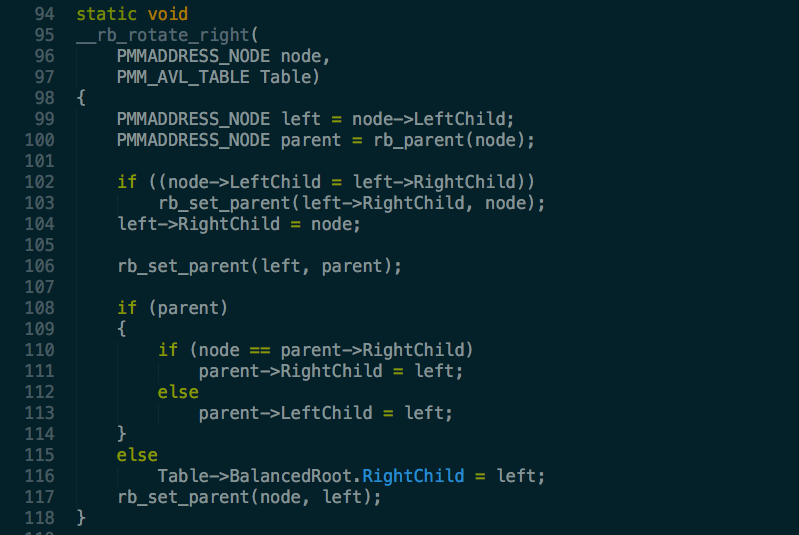
具体修改代码如下：



rb\_tree左旋函数\_\_rb\_rotate\_left：

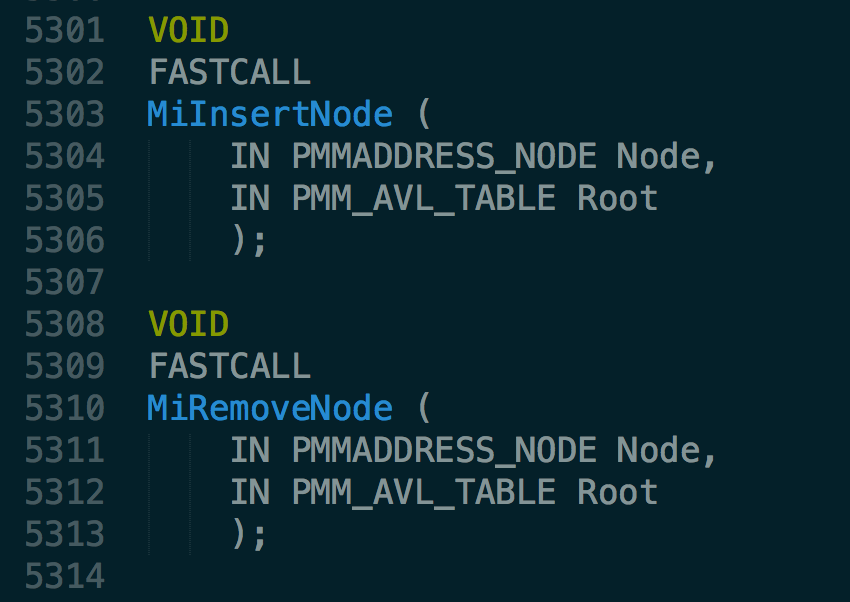


rb\_tree右旋函数\_\_rb\_rotate\_right：



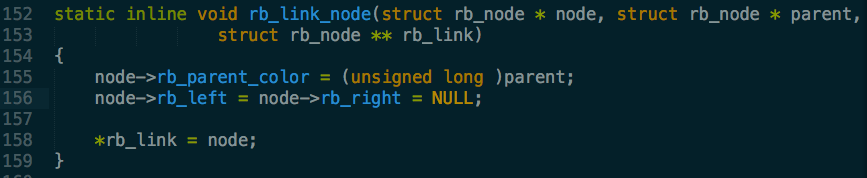
移植时，只需更改源码中的MiInsertNode以及MiRemoveNode函数。

在wrk-v1.2/base/ntos/mm/mi.h中定义了这两个函数的外部接口：



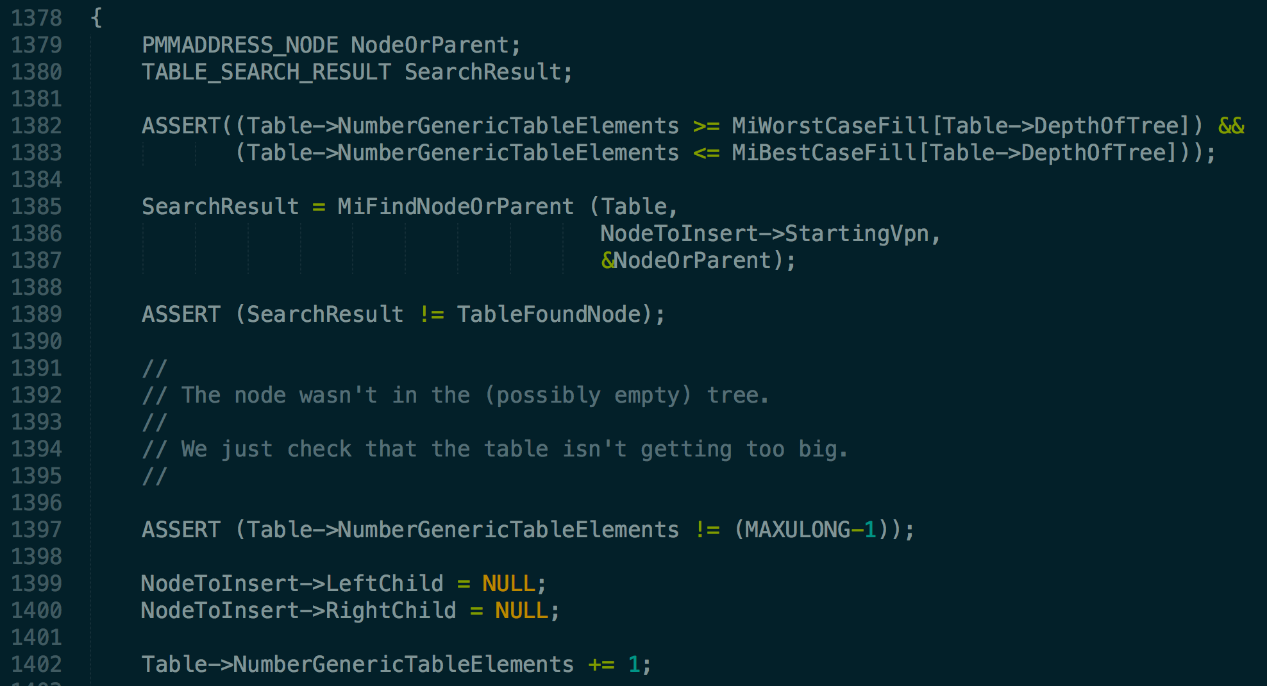
1. MiInsertNode函数：

Linux rbtree.h源码中用于插入节点的函数是rb\_link\_node：

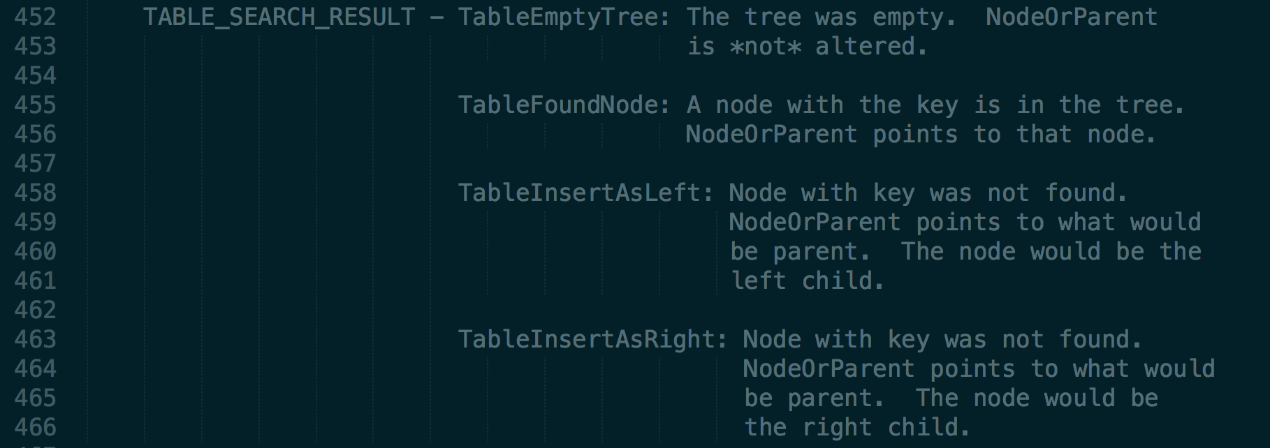


但是如图所示，该函数输入为待删除节点、该节点的父节点以及根节点；而MiInsertNode函数输入只有待删除节点以及根节点，所以不能使用该函数。正确的做法应该是利用MiInsertNode中本来的插入节点部分，只是需要在插入节点后使用rb\_insert\_color以保持rbtree的结构。

判断节点插入位置代码如下：



首先利用源码中的寻找符合节点函数MiFindNodeOrParent，返回值为TableEmptyTree、TableFoundNode、TableInsertAsLeft、TableInsertAsRight。具体意义如下：



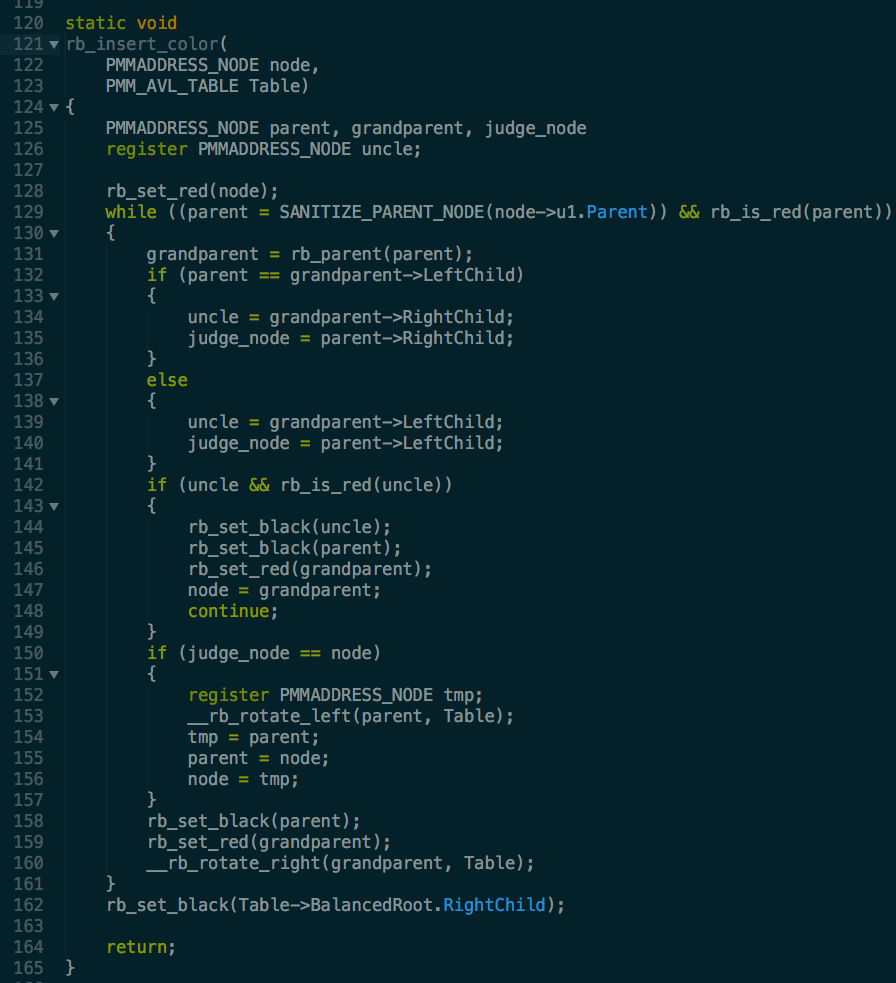
依据这个函数便得到了节点的插入位置。

之后，根据SearchResult的结果分别处理：



* 如为TableEmptyTree，则将将待插入点置于Table→BalancedRoot.RightChild（rb\_tree实际root节点）
* 如为TableInsertAsLeft或TableInsertAsRight，则根据MiFindNodeOrParent返回的NodeOrParent值插入到相应位置。之后核心的一步便是调用rb\_insert\_color函数调整颜色以保持红黑树的性质。

rb\_insert\_color代码如下：



该函数实现逻辑与linux-2.6中逻辑一致，并分别对应实验思路红黑树学习插入和插入修复中的几种情况。

1. MiRemoveNode函数：

Linux rbtree.h源码中用于插入节点的函数是rb\_erase：

\\Mac\Home\Desktop\屏幕快照 2017-12-06 19.21.25.png

函数输入为待删除节点以及根节点；相应的，MiRemoveNode函数输入为待删除节点以及根节点。这两个函数输入完全一致，因此可以直接利用rb\_erase。只不过需要调整用于删除节点后保持rbtree性质的\_\_rb\_erase\_color函数。

修改后\_\_rb\_erase\_color函数代码如下：





该函数实现逻辑与rbtree.c中完全一致，只是替换为相应的数据结构。

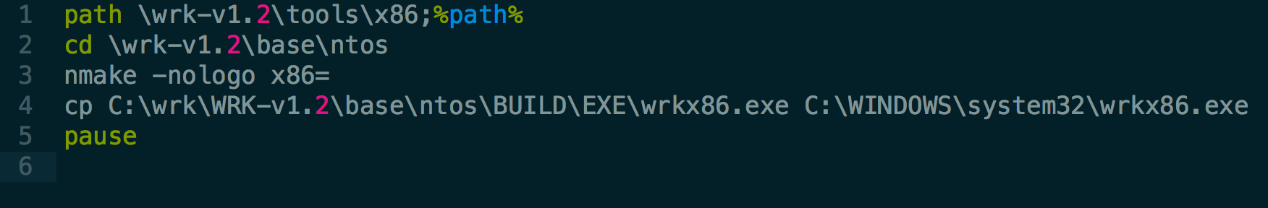
而MiRemoveNode函数可以直接按照rb\_erase的方式实现：





至此，代码移植工作完成。

接下来，在虚拟机上使用脚本将新的代码移植到操作系统内核即可。



1. **实验感想：**

本次实验我阅读了大量WRK及linux源码，并成功将AVL树改为红黑树。实验原理其实不难，难度主要在找到并理解windows和linux文件系统的相关接口。本次实验收获主要在两方面：其一，这是我第一次接触操作系统编程及内核调试，算是打开一扇新世界的大门；其二，在阅读操作系统源码的过程中，我感受到了专业工程师们严谨的代码风格以及详实的注释，对于debug及他人阅读带来了很大的便利性。所以，我在以后的编程中也应自我鞭策，多加注意。

1. 参考“The Art Of Programming” (https://github.com/julycoding/The-Art-Of-Programming-By-July/blob/master/ebook/zh/03.01.md) [↑](#footnote-ref-1)