

# 《操作系统》课程大作业

## AVL 树→红黑树问题

无 42

ShowLo

2016/12/21

### 一、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树，在 Linux 2.6 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树替换为红黑树。

## 二、实验步骤

### 2.1 理解红黑树

#### 2.1.1 红黑树的定义

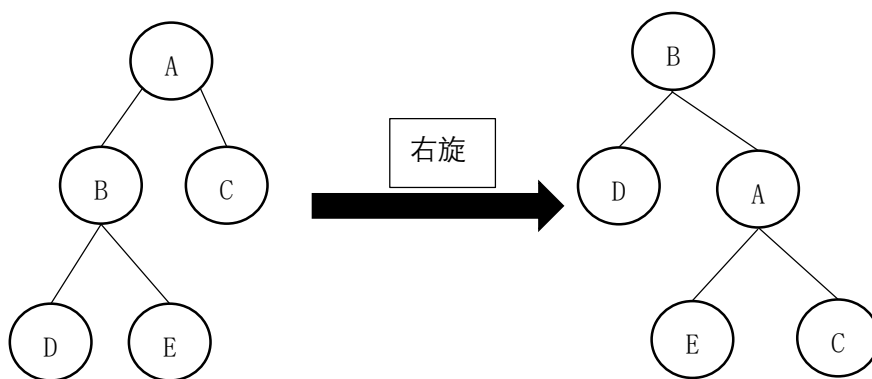
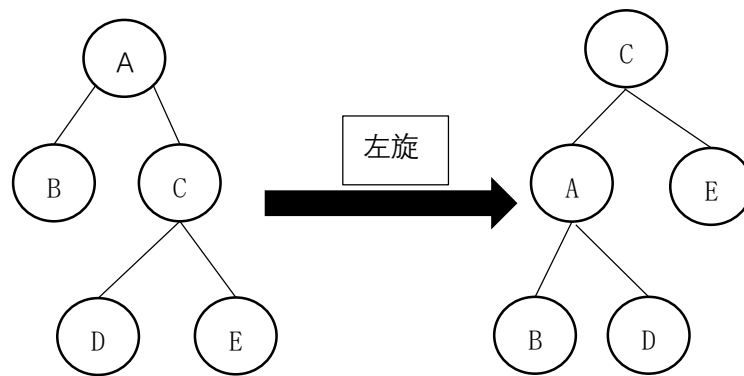
红黑树的定义如下：

- (1)任何一个节点都有颜色--红色抑或黑色
- (2)根节点是黑色的
- (3)父子节点之间不能出现连续两个红色节点
- (4)任何一个节点向下遍历到其子孙的叶子节点时所经过的黑色节点数必须一样
- (5)空节点认为是黑色的

红黑树在理论上还是一棵二叉查找树，但是因为它在执行插入和删除操作时会保持树的平衡，保证树的高度在 $[\log N, \log N + 1]$ （理论上极端情况下可以出现红黑树高度达到 $2\log N$ ，但实际上很难遇到），这样红黑树的查找时间复杂度始终维持在 $O(\log N)$ 从而接近理想的二叉查找树的性能。

#### 2.1.2 红黑树的旋转操作

旋转操作的目的是使得各节点的颜色符合红黑树的定义，从而使红黑树的高度达到平衡。旋转分为左旋和右旋，左旋即为待旋转的节点从右边上升到父节点，右旋即为待旋转的节点从左边上升到父节点，如下图所示。



### 2.1.3 红黑树的查找操作

查找的时候，与当前节点进行比较并作处理如下：

- 如果相等的话直接返回当前节点
- 如果小于当前节点的话则继续查找当前节点的左节点
- 如果大于当前节点的话则继续查找当前节点的右节点

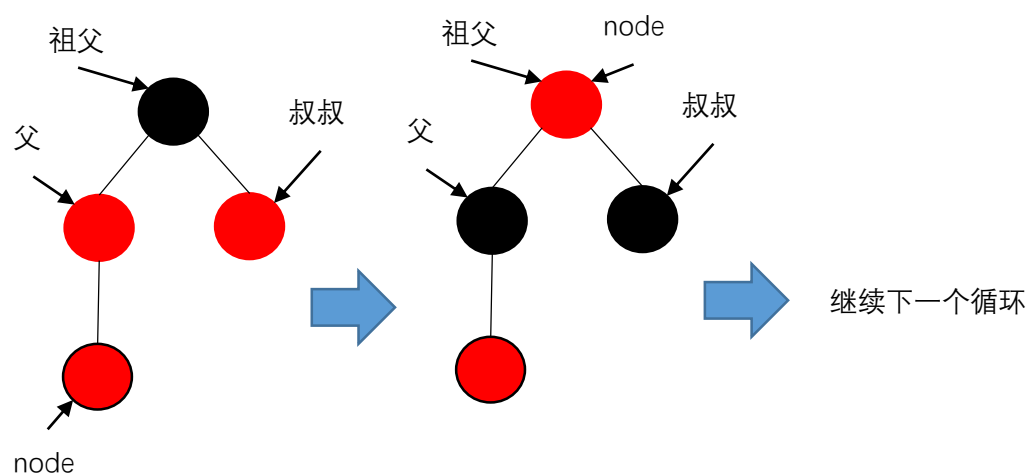
当当前节点指针为空或者查找到对应的节点时，查找结束。

### 2.1.4 红黑树的插入操作

红黑树的插入操作与二叉查找树是一样的，不过由于红黑树在插入之后还要维持树的平衡，所以需要对树进行插入修复（即旋转操作以及颜色修复），使得插入之后的树仍然满足红黑树的定义。

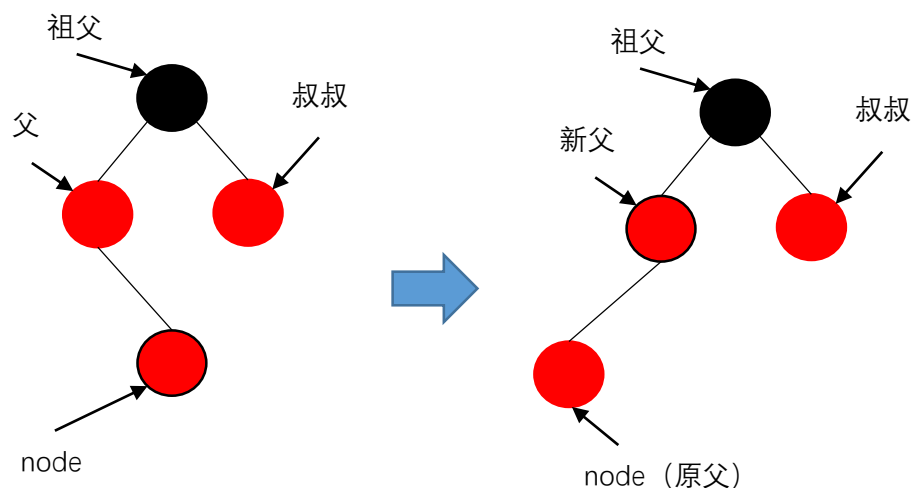
默认新插入的节点为红色的，所以如果父节点的颜色是黑色的则插入修复结束，也就是说只有父节点的颜色为红色的时候才需要进行插入修复。而插入修复又可以分为两种情况，即父节点是祖父节点的左子节点或者右子节点这两种情况，因为两种情况下的处理是极为类似的，所以只对父节点是祖父节点的左子节点的情况作讨论如下：

①如果新插入结点有叔叔节点且是红色的，那么将叔叔节点和父节点均改为黑色，然后将祖父节点改为红色，此时黑色节点的数目是平衡的，仍然符合红黑树定义。但是因为祖父节点变成红色的，所以此时不能确定是否还存在父子节点皆为红色的情况，故需要对祖父节点进行下一轮修复。第①步操作的示意图如下：

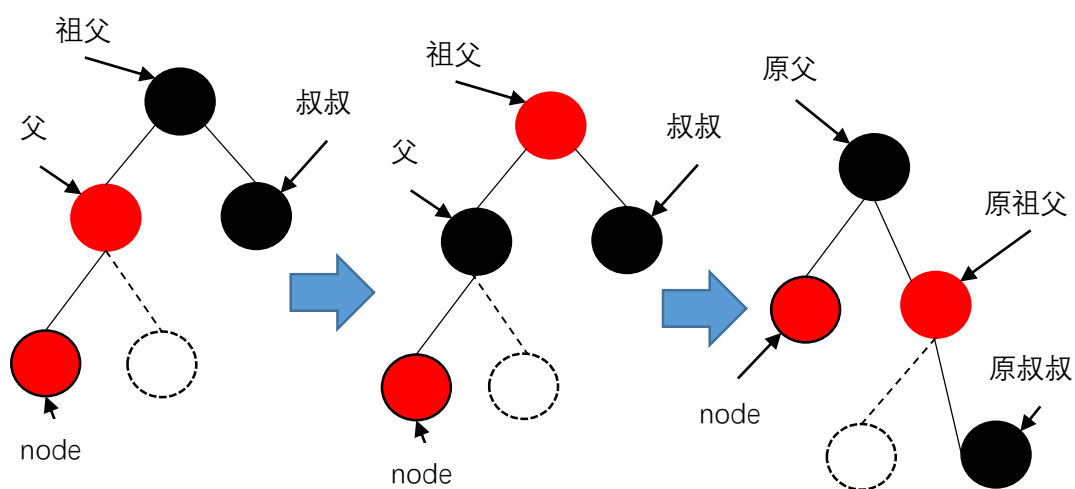


接下来是叔叔节点为空或者为黑色的情况。

②如果新插入结点是父节点的右子节点，则对父节点进行左旋操作，旋转后树仍然是平衡的，但是新插入节点占据了原父节点的位置，即插入的节点成了新的父节点而原父节点成了新的父节点的左子节点，而且这两个节点都是红色的，需要执行第③步的操作。第②步操作的示意图如下：



③当 **node** 是其父节点的左子节点时，此时父节点和左子节点都是红色的，虽然树是平衡的但是有连续红色节点，需要进行修复。先把父节点变成黑色，祖父节点变成红色，然后对祖父节点进行右旋，就可以将树调整为符合红黑树定义的状态。第③步操作的示意图如下：



总的来说，因为插入后的修复操作是一个向根节点回溯的操作，只要涉及到的节点都符合了红黑树的定义，那么插入修复就完成了。

#### 2.1.4 红黑树的删除操作

红黑树的删除操作也跟二叉查找树的删除操作一样，如果删除的是叶子节点的话节直接删除，如果不是叶子节点就用对应的中序遍历的后继节点顶替要删除

节点的位置。但是为了维持红黑树的平衡，所以在删除节点之后还需要做删除修复。但是因为删除红色节点不会导致红黑树失去平衡，所以删除修复都是针对删除黑色节点而言的。

为了修复由于删除黑色节点而可能导致的整棵树不满足红黑树的第(4)条定义，需要做的处理是从兄弟节点上借调黑色的节点过来，当兄弟节点没有黑色节点的时候就往上追溯，将每一级的黑色节点数减掉一个，最终使得整棵树满足红黑树的定义。

删除修复操作可以分为四种情况。第一种情况是兄弟节点为红色的情况，这个时候是没法从红色的兄弟节点借调出黑色节点的，只能令其上升到父节点，这时因为它的子节点是黑色的所以可以借调。其他的三种情况对应于兄弟节点为黑色的：其一是兄弟节点的所有子节点都是黑色的，这时可以直接将兄弟节点变为红色的，然后往上继续追溯调整（因为虽然局部的树的颜色符合红黑树的定义，但是整棵树却不一定满足）；剩下的就是兄弟节点的子节点为左红右黑或者（左红右红，左黑右红）的情况，前者可以转换为后者的情况。对于后者的情况，因为兄弟节点是黑色的且其右子节点为红色的，所以可以借调出两个节点出来做黑色节点，从而保证删除了黑色节点之后整棵树还是符合红黑树的定义的。

当遇到删除的节点为红色或者追溯到了 `root` 节点的时候删除修复操作就完成了。

本次实验所用的红黑树来自 `Linux-3.2.84`，有关红黑树的两个文件已放置于 `code` 文件夹中，分别为 `rbtree.h` 和 `rbtree.c`。

其中 `Linux` 的插入操作即为上面所讲到的插入操作，删除操作由于比较复杂，上面只讲了基本的操作思想，具体操作方法参见代码中我所写的注释。

## 2.2 理解 VAD 及 AVL 树

### 2.2.1 Windows 中的 VAD

VAD 即虚拟地址描述符，对于每一个进程，Windows 的内存管理器维护一组虚拟地址描述符来描述一段被分配的进程虚拟空间的状态，虚拟地址描述信息被构造成一棵自平衡二叉树（AVL 树）以使查找更有效率。这棵树的根节点的地址保存在进程结构 EPROCESS 中的一个 MM\_AVL\_TABLE 类型的变量 VadRoot 中。

其中 MM\_AVL\_TABLE 的定义如下：

```
typedef struct _MM_AVL_TABLE {
    MMADDRESS_NODE    BalancedRoot;
    ULONG_PTR    DepthOfTree: 5;
    ULONG_PTR    Unused: 3;
#ifdef _WIN64
    ULONG_PTR    NumberGenericTableElements: 56;
#else
    ULONG_PTR    NumberGenericTableElements: 24;
#endif
    PVOID    NodeHint;
    PVOID    NodeFreeHint;
} MM_AVL_TABLE, *PMM_AVL_TABLE;
```

这是一个对节点的封装，实际上 MMADDRESS\_NODE BalancedRoot 才是真正的根节点。而 MMADDRESS\_NODE 的定义如下：

```
typedef struct _MMADDRESS_NODE {
    union {
        LONG_PTR    Balance : 2;
        struct _MMADDRESS_NODE *Parent;
    } u1;
    struct _MMADDRESS_NODE *LeftChild;
    struct _MMADDRESS_NODE *RightChild;
    ULONG_PTR    StartingVpn;
    ULONG_PTR    EndingVpn;
} MMADDRESS_NODE, *PMMADDRESS_NODE;
```

Windows 利用 `MMADDRESS_NODE` 对 VAD 进行了抽象处理,这是因为利用 AVL 组织的不止 VAD, 还有其他一些与 VAD 较为类似的结构, 将这些结构的相同的部分抽象出来作为基础结构可以简化代码结构以及方便用统一的方法进行组织和操作。

### 2.2.1 AVL 树

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, 具有以下特点:

1. 本身首先是一棵二叉搜索树。
2. 带有平衡条件, 也即每个结点的左右子树的高度之差的绝对值(平衡因子)最多为 1。

AVL 树节点在 Windows 中是 `MMADDRESS_NODE` 结构体, 其中的 `Balance` 即为平衡因子。因为 AVL 树对外的接口只有两个--插入 `MiInsertNode` 和删除 `MiRemoveNode`, 所以这两个接口就是本次实验中需要进行修改的地方了。

## 2.3 进行代码移植

Windows 的 AVL 树的插入和删除函数均定义于 `addrsup.c` 中, 因此也只需要在这个文件中增加红黑树定义然后修改插入和删除函数即可。

首先看一下 Linux (版本为 3.2.84) 中红黑树的定义 (`rbtree.h` 中):



```

struct rb_node
{
    unsigned long rb_parent_color;
#define RB_RED    0
#define RB_BLACK  1
    struct rb_node *rb_right;
    struct rb_node *rb_left;
} __attribute__((aligned(sizeof(long))));
/* The alignment might seem pointless, but allegedly CRIS needs it */

struct rb_root
{
    struct rb_node *rb_node;
};

```

注意到 `rb_node` 指定了以 `long` 类型的大小对齐，即以 4 字节对齐（32 位系统），所以地址只能是 4 的倍数，也就是说其地址的最低两位永远是 0，Linux 的开发者于是用了其中一个空闲的来存储颜色信息，因此在取地址以及取颜色、判断颜色的时候只需简单地做逻辑运算即可，可以在 `rbtree.h` 中看到以下的操作均利用了这个特点而将各种操作用逻辑运算实现了。

```

#define rb_parent(r)    ((struct rb_node *)((r)->rb_parent_color & ~3))
#define rb_color(r)    ((r)->rb_parent_color & 1)
#define rb_is_red(r)    (!rb_color(r))
#define rb_is_black(r) rb_color(r)
#define rb_set_red(r)    do { (r)->rb_parent_color &= ~1; } while (0)
#define rb_set_black(r) do { (r)->rb_parent_color |= 1; } while (0)

static inline void rb_set_parent(struct rb_node *rb, struct rb_node *p)
{
    rb->rb_parent_color = (rb->rb_parent_color & 3) | (unsigned long)p;
}

static inline void rb_set_color(struct rb_node *rb, int color)
{
    rb->rb_parent_color = (rb->rb_parent_color & ~1) | color;
}

```

比如说第一个取父节点地址的将 `rb_parent_color` 和 `~3`（即 `0xFFFFFFFFC`）做与运算就相当于将 `rb_parent_color` 的最低两位又重新设置为 0，变回了地址，也即是取到了父节点的地址。再如取节点颜色的 `rb_color` 则是取出最低位的数，然后根据是 1 还是 0 就可以判断节点颜色（这其实是在 `rb_is_red` 和

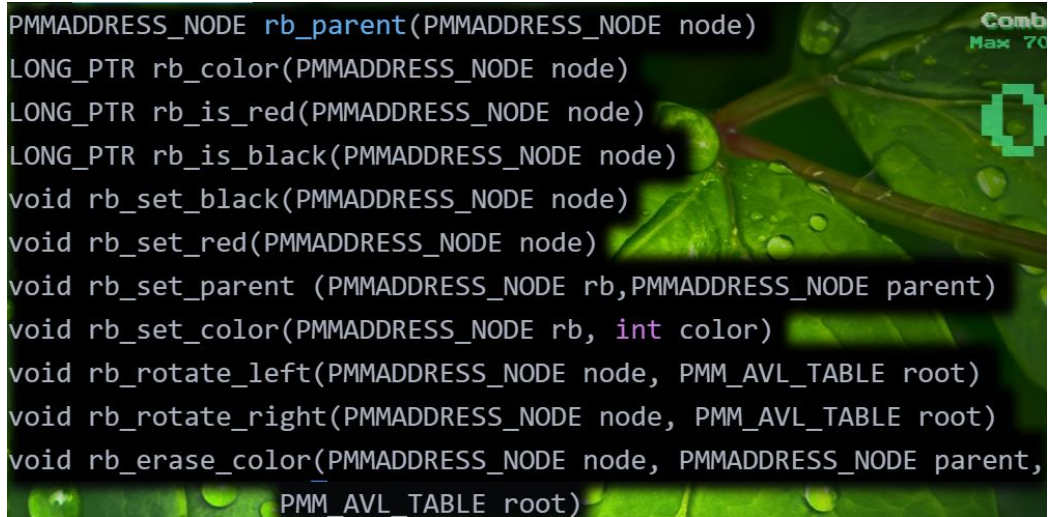
rb\_is\_black 中进行判断的，而且 Linux 设定 1 代表黑色，0 代表红色)。其他的操作都可以作类似的分析。

在 Linux 中跟红黑树维护有关的函数有旋转的 \_\_rb\_rotate\_left 和 \_\_rb\_rotate\_right，插入的 rb\_insert\_color，删除的 rb\_erase 和 \_\_rb\_erase\_color，替换的 rb\_replace\_node，寻找前驱后继的 rb\_next 和 rb\_prev 等。这些函数均实现于 rbtree.c 中。

有了以上的知识准备，就可以开始真正的代码移植了。

这里作移植的整体思路是保持 Windows 代码中原来的两个接口 MiInsertNode 和 MiRemoveNode 不变而在函数内部将 AVL 树的维护操作更改为红黑树的维护操作。而且注意到红黑树并不需要用到 AVL 树的平衡因子 Balance，所以可以利用 Balance 来存储节点颜色。

①首先要替换 Linux 中定义的函数，替换后的函数定义如下：

A screenshot of a code editor showing the definition of several functions for a red-black tree. The background of the code editor features a green, textured pattern resembling leaves or a microscopic view of a surface. The functions are defined with PMMADDRESS\_NODE as the primary data type. The functions include rb\_parent, rb\_color, rb\_is\_red, rb\_is\_black, rb\_set\_black, rb\_set\_red, rb\_set\_parent, rb\_set\_color, rb\_rotate\_left, rb\_rotate\_right, rb\_erase\_color, and a partially visible rb\_replace\_node.

```
PMMADDRESS_NODE rb_parent(PMMADDRESS_NODE node)
LONG_PTR rb_color(PMMADDRESS_NODE node)
LONG_PTR rb_is_red(PMMADDRESS_NODE node)
LONG_PTR rb_is_black(PMMADDRESS_NODE node)
void rb_set_black(PMMADDRESS_NODE node)
void rb_set_red(PMMADDRESS_NODE node)
void rb_set_parent (PMMADDRESS_NODE rb, PMMADDRESS_NODE parent)
void rb_set_color(PMMADDRESS_NODE rb, int color)
void rb_rotate_left(PMMADDRESS_NODE node, PMM_AVL_TABLE root)
void rb_rotate_right(PMMADDRESS_NODE node, PMM_AVL_TABLE root)
void rb_erase_color(PMMADDRESS_NODE node, PMMADDRESS_NODE parent,
PMM_AVL_TABLE root)
```

具体的实现见代码文件 addrsup.c。

②接着需要修改插入函数

首先看一下插入函数的定义：

```

VOID
FASTCALL
MiInsertNode (
    IN PMMADDRESS_NODE NodeToInsert,
    IN PMM_AVL_TABLE Table
)

```

注意 Windows 中 MiInsertNode 函数与 Linux 的 rb\_insert\_color 有所不同，Linux 的 rb\_insert\_color 是在已知插入位置的情况下被调用的，而 MiInsertNode 则要同时完成寻找插入位置和插入的两步操作。

所以对 MiInsertNode 进行修改的时候要保留原有的查找插入位置的操作，然后将维护操作改为 linux 里 rb\_insert\_color 函数的操作，当然也要作相应的修改。具体的操作在代码里已经作了注释。

上面用到的寻找插入位置函数定义如下：

```

TABLE_SEARCH_RESULT
MiFindNodeOrParent (
    IN PMM_AVL_TABLE Table,
    IN ULONG_PTR StartingVpn,
    OUT PMMADDRESS_NODE *NodeOrParent
);

```

这个函数的作用就是根据给定的根节点和数据内容寻找符合要求的节点，如果找到符合要求的节点则将 NodeOrParent 指向这个符合要求的节点，否则指向应该插入位置的父节点。函数返回一个枚举类型 TABLE\_SEARCH\_RESULT，可能的结果有树为空、找到节点、作为左子节点插入及作为右子节点插入。

在 Windows 源码里是先利用这个函数得到查询结果，然后根据查询结果决定在哪里插入节点。如果是树为空的情况，那么直接将节点作为根节点即可，且

其颜色必为黑色的。如果树不空，那就找到合适的位置插入，默认其颜色为红色的，在这时就需要进行调整了，因为新插入的节点是红色的有可能使得树不满足红黑树的性质，所以有必要进行调整。而调整操作就是需要作替换的部分，这里调整用到的就是 Linux 里维护红黑树性质用到的 `rb_insert_color`，基本上只需要修改变量类型使其适用于 Windows 即可，当然这里面有一个需要注意的地方，就是在 Windows 里树的根节点是 `BalancedRoot` 的右子节点。当然这只是一些小细节，并不本质，真正本质的是算法的适用性。

### ③最后需要修改删除函数

删除函数的定义如下：

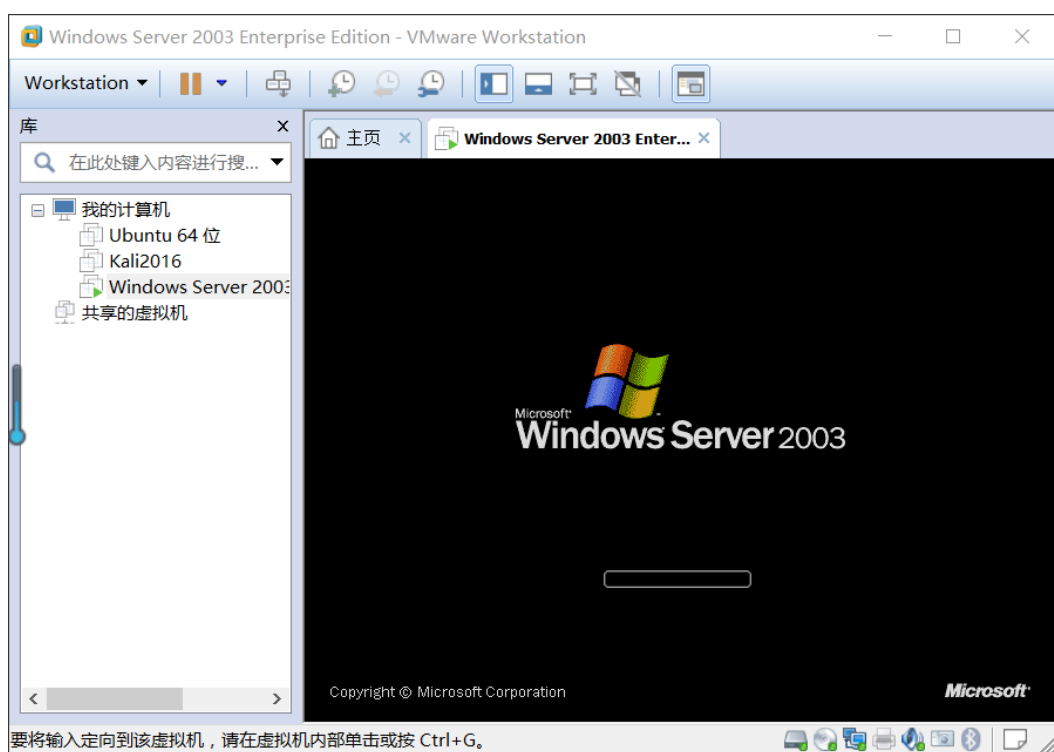
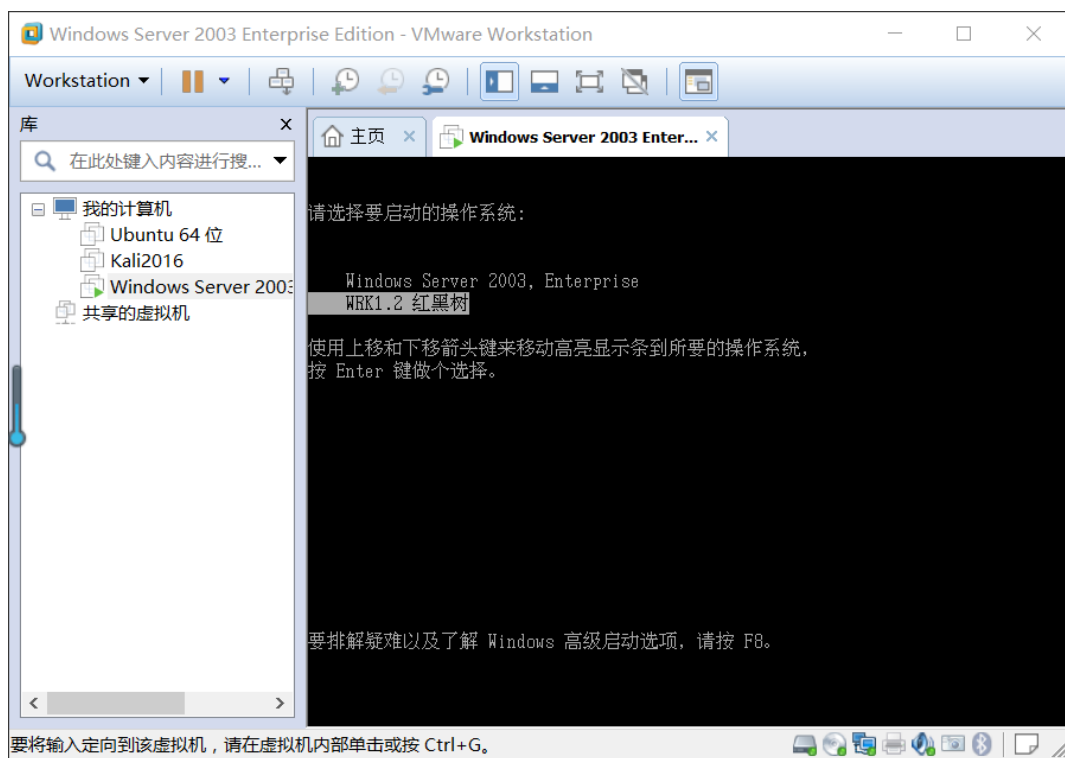
```
VOID
FASTCALL
MiRemoveNode(
    IN PMMADDRESS_NODE NodeToDelete,
    IN PMM_AVL_TABLE Table
)
```

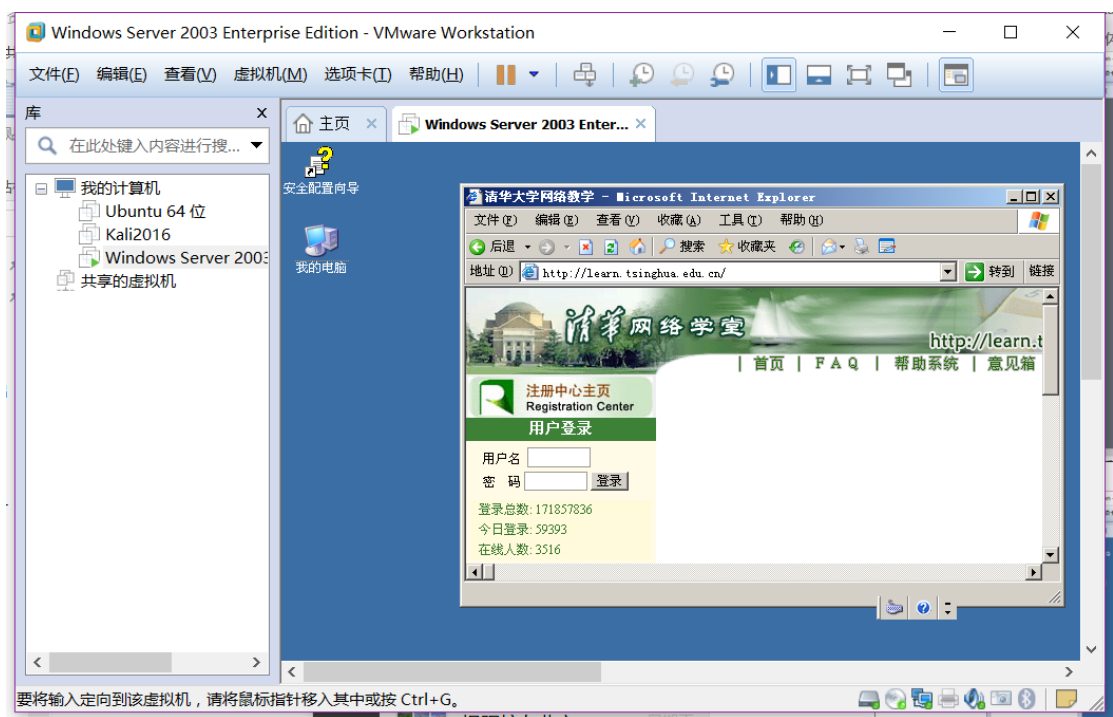
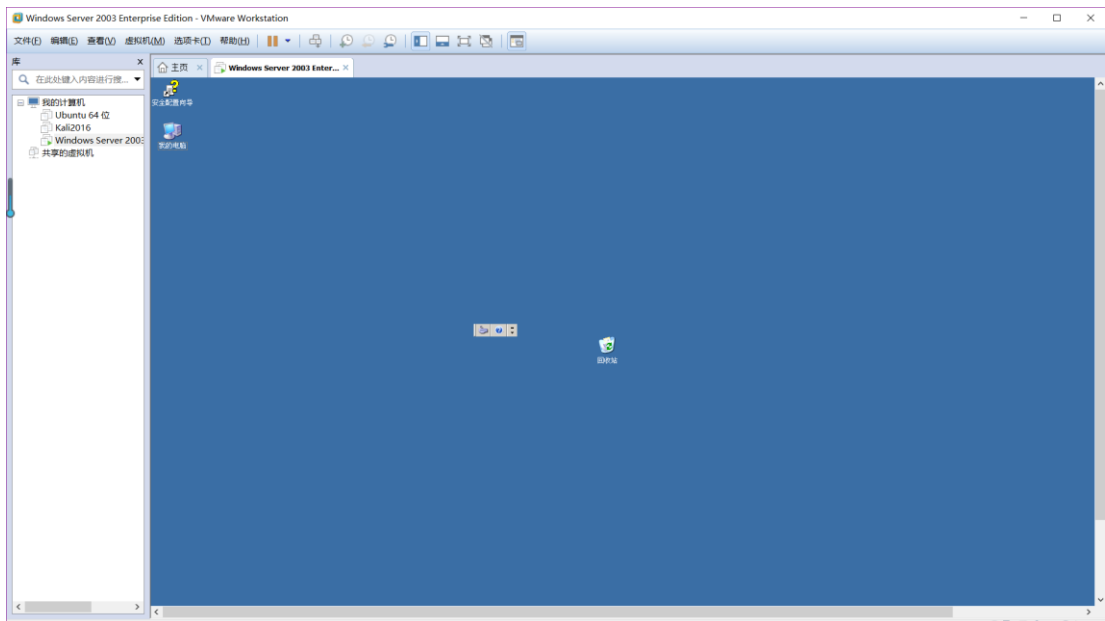
因为红黑树算法的适用性，可以直接将 Linux 里面的删除函数 `rb_erase` 函数稍作变量类型修改，然后同样将根节点表示方法改为适用于 Windows 的即可。具体的操作已经在代码里作了注释，详见 `addrsup.c`。

## 三、实验结果

按照马老师之前给的一个文件“WRK 实验环境设置”配置实验环境，并将原本的 `addrsup.c` 替换为自己修改了的 `addrsup.c`（即 `code` 文件夹里的）。然后启动虚拟机中的 Windows Server 2003，选择自己编译好的 WRK 内核启动系统，可以看到以下的实验结果：







可以看到正确完成了开机操作，并且可以正常完成上网功能，另外文件的打开也是正常的，说明虚拟内存管理没有出现错误，因此本次实验成功地用红黑树替换了 AVL 树。

## 四、实验总结

首先，做完本次实验的第一感觉就是老师说的没错，这个实验确实需要很多

的时间，甚至需要的时间比选择做三个其他的实验的时间还要多。一方面是理解 Linux 源码需要比较多的时间，这是因为 Linux 的开发者对内核速度的狂热追求导致了代码比较费解，所以我在理解 Linux 源码的时候也遇到了不少的麻烦，耗费了不少的时间。当然这也说明了 Linux 开发者的过人之处和我自己的不足，学无止境，他们应该是我学习的榜样。另一方面是因为实验是在操作系统内核层面直接做的修改，导致调试起来比一般的程序要困难很多，一旦出错就很容易会出现蓝屏的情况，我在实验中就遇到了几十次蓝屏，也确实考验着我的耐心。

但是，花费这么多时间在这个实验上也收获了很多。最直接的就是弥补了数据与算法课上没学红黑树的遗憾，对于红黑树和 AVL 树有了更多的理解和认识。同时也在实验中切身体会到了编写和维护一个操作系统的不易，对于 Windows 和 Linux 的开发维护人员有一种由然而发的敬佩感，编写和维护一个操作系统无疑是不易的，但这也恰恰显示了人类智慧的伟大非凡。

## 五、参考文献

[1]Cormen T H, Leiserson C E, Rivest R L, 等. 算法导论 (第 3 版). 殷建平, 等. 机械工业出版社, 2012.

[2]Sedgewick R, Wayne K. 算法 (第 4 版). 谢路云 译. 人民邮电出版社, 2012.

[3]Weiss M A. 数据结构与算法分析 (第 2 版). 冯舜玺 译. 机械工业出版社, 2004.

[4]Knuth D E. 计算机程序设计艺术 卷 3: 排序与查找 (英文版 第 2 版). 人民邮电出版社, 2010.

[5] Linux 内核源码之红黑树注释