哈爾濱工業大學

实验报告

实 验（八）

题　　目 Dynamic Storage Allocator

动态内存分配器

专　　业 计算机类

学　　号 1180300829

班　　级 1803008

学 生 姓 名 余　涛

指 导 教 师 吴　锐

实 验 地 点 G709

实 验 日 期 2019.12.14

计算机科学与技术学院

目 录

第 1 章 实验基本信息

1.1 实验目的

理解现代计算机系统虚拟存储的基本知识
掌握 C 语言指针相关的基本操作
深入理解动态存储申请、释放的基本原理和相关系统函数
用 C 语言实现动态存储分配器，并进行测试分析
培养 Linux 下的软件系统开发与测试能力

1.2 实验环境与工具

1.2.1 硬件环境

X64 CPU；2GHz；2G RAM；256GHD Disk 以上

1.2.2 软件环境

Windows7 64 位以上；VirtualBox/Vmware 11 以上；Ubuntu 16.04 LTS 64 位/优麒麟 64 位

1.2.3 开发工具

Visual Studio 2010 64 位以上；TestStudio；Gprof;Valgrind 等

1.3 实验预习

- **上实验课前，必须认真预习实验指导书（PPT 或 PDF）**
- **了解实验的目的、实验环境与软硬件工具、实验操作步骤，复习与实验有关的理论知识。**
- **熟知 C 语言指针的概念、原理和使用方法**
- **了解虚拟存储的基本原理**
- **熟知动态内存申请、释放的方法和相关函数**
- **熟知动态内存申请的内部实现机制：分配算法、释放合并算法等**

第 2 章 实验预习

总分 20 分

2.1 动态内存分配器的基本原理（5 分）

动态内存分配器维护着一个进程的虚拟内存区域，称为堆。系统之间细节不同，但是不失通用性，假设堆是一个请求二进制零的区域，它紧接在未初始化的数据区域后开始，并向上生长。对于每个进程，内核维护着一个变量 brk，它指向堆的顶部。

分配器将堆视为一组不同大小的块的集合来维护。每个块就是一个连续的虚拟内存片，要么是已分配的，要么是空闲的。已分配的块显式地保留为供应用程序使用。空闲块可用来分配。空闲块保持空闲，直到它显式地被应用所分配。一个已分配的块保持已分配状态，直到它被释放，这种释放要么是应用程序显式执行的，要么是内存分配器自身隐式执行的。

分配器有两种基本风格，两种风格都要求应用显式地分配块，它们的不同之处在于由哪个实体来负责释放已分配的块。

显式分配器：要求应用显式地释放任何已分配的块。例如，c 标准库提供一种叫做 malloc 程序包的显式分配器。c 程序通过调用 malloc 函数来分配一个块，并通过调用 free 函数来释放一个块。c++中的 new 和 delete 操作符与 c 中的 malloc 和 free 相当。

隐式分配器：另一方面，要求分配器检测一个已分配块何时不再被程序所使用，那么就释放这个块。隐式分配器也叫做垃圾收集器，而自动释放未使用的已分配的块的过程叫做垃圾收集，例如 Lisp,ML 以及 Java 之类的高级语言就依赖垃圾收集来释放已分配的块。

2.2 带边界标签的隐式空闲链表分配器原理（5 分）

一个块是由一个字的头部、有效载荷、可能的一些额外的填充，以及在块的结尾处的一个字的脚部组成的。头部编码了这个块的大小（包括头部和所有的填充），以及这个块是已分配的还是空闲的。如果我们强加一个双字的对齐约束条件，那么块大小就总是 8 的倍数，且块大小的最低 3 位总是 0。因此，我们只需要内存大小的 29 个高位，释放剩余的 3 位来编码其他信息。在这种情况中，我们用其中的最低位（已分配位）来指明这个块是已分配的还是空闲的。

头部后面就是应用调用 malloc 时请求的有效载荷。有效载荷后面是一片不使用的填充块，其大小可以是任意的。需要填充有很多原因。比如，填充可能是分配器策略的一部分，用来对付外部碎片。或者也需要用它来满足对齐要求。

我们称这种结构称为隐式空闲链表，是因为空闲块是通过头部中的大小字段

隐含地连接着的。分配器可以通过遍历堆中所有的块，从而间接地遍历整个空闲块的集合。注意：此时我们需要某种特殊标记的结束块，可以是一个设置了已分配位而大小为零的终止头部。

2.3 显示空间链表的基本原理（5 分）

因为根据定义，程序不需要一个空闲块的主体，所以实现空闲链表数据结构的指针可以存放在这些空闲块的主体里面。

显式空闲链表结构将堆组织成一个双向空闲链表，在每个空闲块的主体中，都包含一个 pred（前驱）和 succ（后继）指针。

使用双向链表而不是隐式空闲链表，使首次适配的分配时间从块总数的线性时间减少到了空闲块数量的线性时间。不过，释放一个块的时间可以是线性的，也可能是个常数，这取决于空闲链表中块的排序策略。

一种方法是用后进先出（LIFO）的顺序维护链表，将新释放的块放置在链表的开始处。另一种方法是按照地址顺序来维护链表，其中链表中每个块的地址都小于它后继的地址。

2.4 红黑树的结构、查找、更新算法（5 分）

1.结构：

R-B Tree，全称是 Red-Black Tree，又称为“红黑树”，它一种特殊的二叉查找树。红黑树的每个节点上都有存储位表示节点的颜色，可以是红(Red)或黑(Black)。

红黑树的特性:

（1）每个节点或者是黑色，或者是红色。

（2）根节点是黑色。

（3）每个叶子节点（NIL）是黑色。[注意：这里叶子节点，是指为空(NIL 或 NULL)的叶子节点！]

（4）如果一个节点是红色的，则它的子节点必须是黑色的。

（5）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

注意：

(01) 特性(3)中的叶子节点，是只为空(NIL 或 null)的节点。

(02) 特性(5)，确保没有一条路径会比其他路径长出俩倍。因而，红黑树是相对是接近平衡的二叉树。

2.查找：

红黑树是一种特殊的二叉查找树，他的查找方法也和二叉查找树一样，不需要做太多更改。

但是由于红黑树比一般的二叉查找树具有更好的平衡，所以查找起来更快。

```
//查找获取指定的值
public override TValue Get(TKey key)
{
    return GetValue(root, key);
}
```

```
private TValue GetValue(Node node, TKey key)
{
    if (node == null) return default(TValue);
    int cmp = key.CompareTo(node.Key);
    if (cmp == 0) return node.Value;
    else if (cmp > 0) return GetValue(node.Right, key);
    else return GetValue(node.Left, key);
}
```

3.更新：

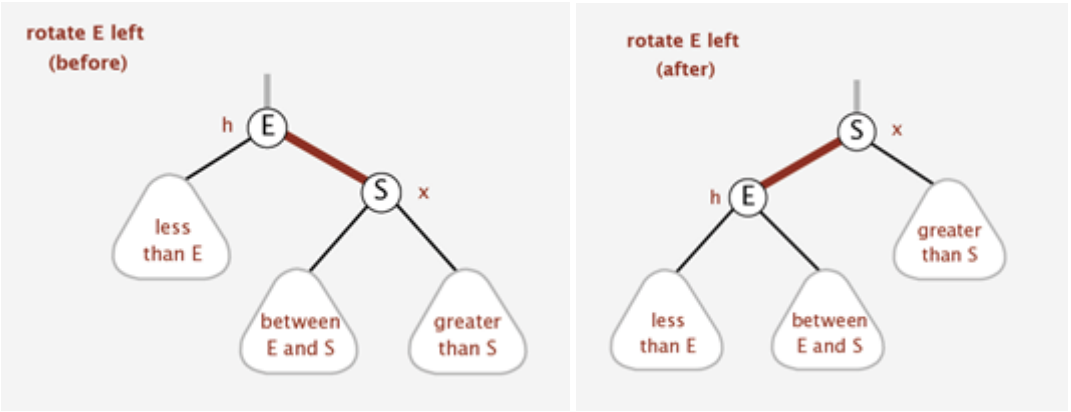
平衡化

在介绍插入之前，我们先介绍如何让红黑树保持平衡，因为一般的，我们插入完成之后，需要对树进行平衡化操作以使其满足平衡化。

旋转

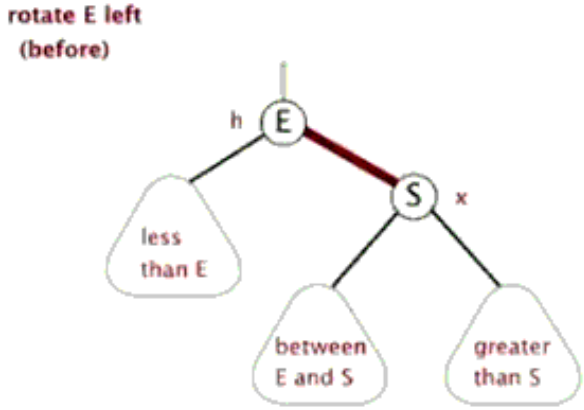
旋转又分为**左旋**和**右旋**。通常左旋操作用于将一个向右倾斜的红色链接旋转为向左链接。对比操作前后，可以看出，该操作实际上是将红线链接的两个节点中的一个较大的节点移动到根节点上。

左旋操作如下图：

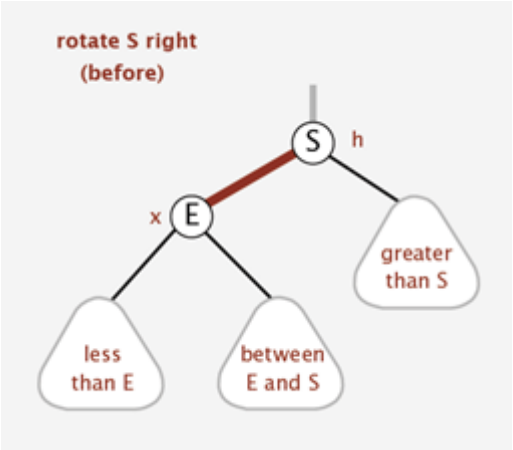


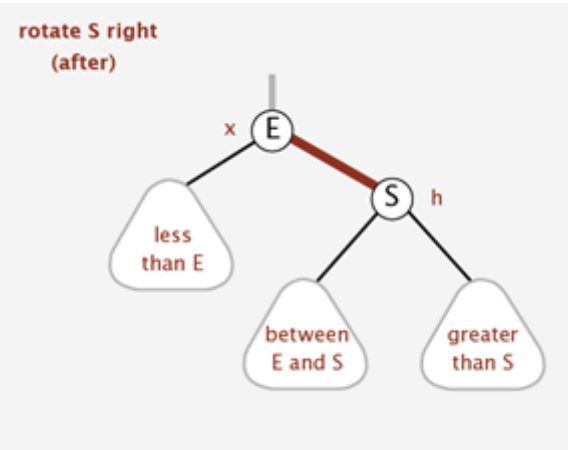

```
//左旋转
private Node RotateLeft(Node h)
{
    Node x = h.Right;
    //将 x 的左节点复制给 h 右节点
    h.Right = x.Left;
    //将 h 复制给 x 右节点
    x.Left = h;
    x.Color = h.Color;
    h.Color = RED;
    return x;
}
```

左旋的动画效果如下：



右旋是左旋的逆操作，过程如下：

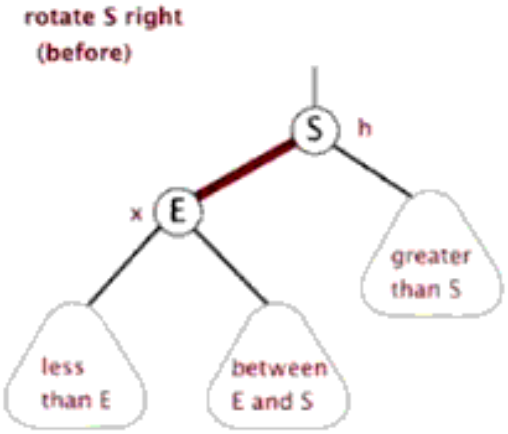




代码如下：

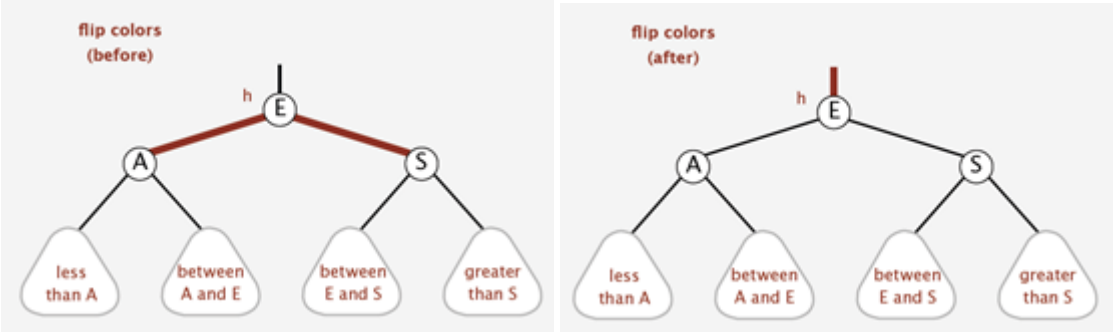
```
//右旋转
private Node RotateRight(Node h)
{
    Node x = h.Left;
    h.Left = x.Right;
    x.Right = h;

    x.Color = h.Color;
    h.Color = RED;
    return x;
}
```

右旋的动画效果如下：



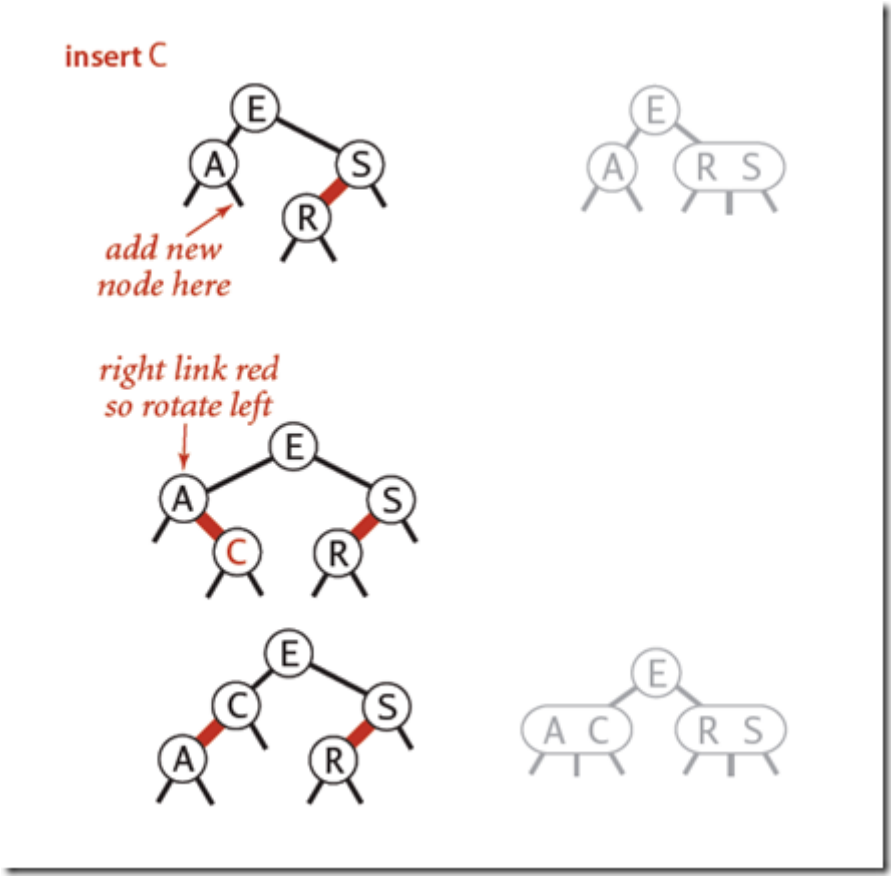
颜色反转

当出现一个临时的 4-node 的时候，即一个节点的两个子节点均为红色，如下图：



这其实是个 A，E，S 4-node 连接，我们需要将 E 提升至父节点，操作方法很简单，就是把 E 对子节点的连线设置为黑色，自己的颜色设置为红色。

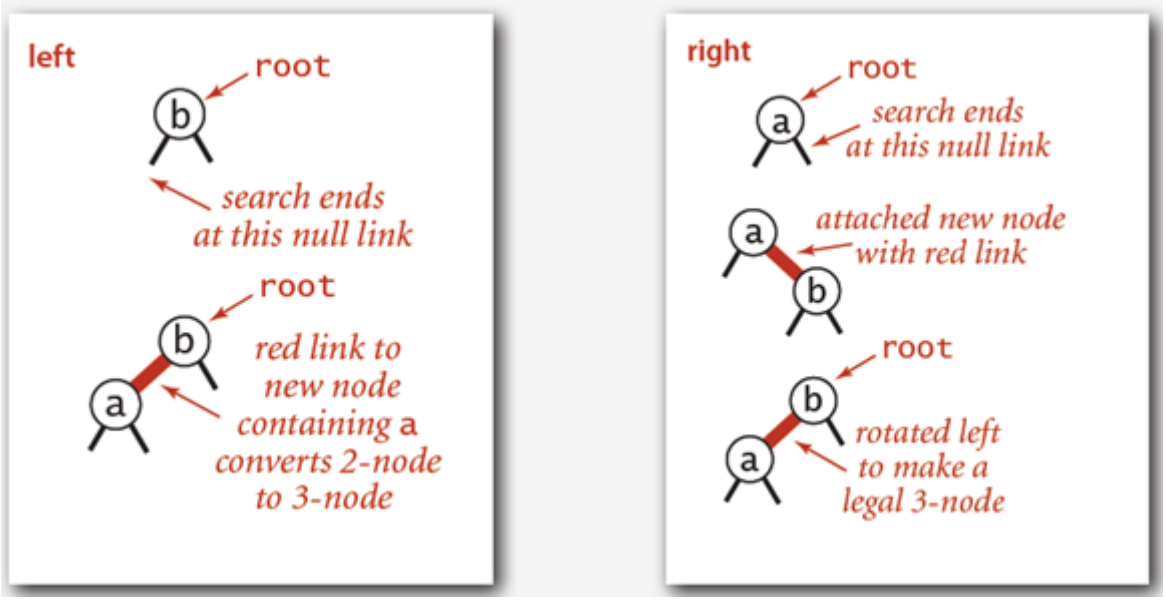
有了以上基本操作方法之后，我们现在对应之前对 **2-3 树的平衡操作来对红黑树进行平衡操作，**这两者是可以一一对应的，如下图：



现在来讨论各种情况：

Case 1 往一个 2-node 节点底部插入新的节点

先热身一下，首先我们看对于只有一个节点的红黑树，插入一个新的节点的操作：



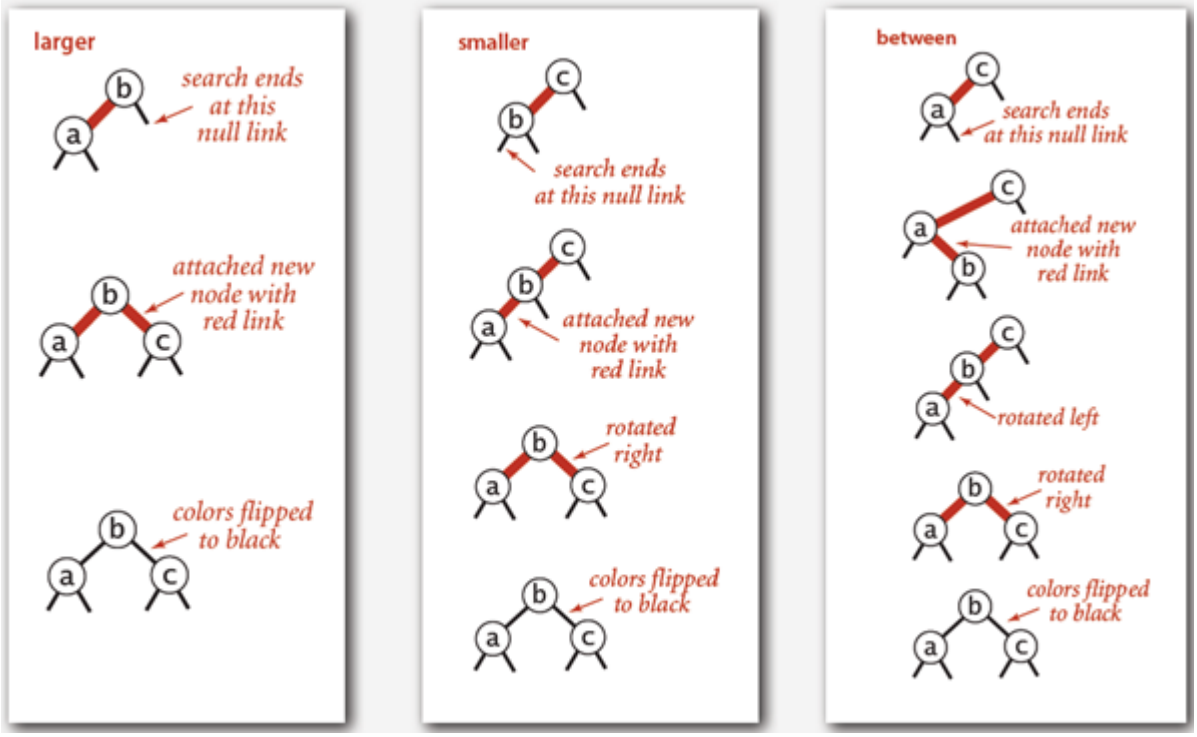
这种情况很简单，只需要：

标准的二叉查找树遍历即可。新插入的节点标记为红色

如果新插入的节点在父节点的右子节点，则需要进行左旋操作

Case 2 往一个 3-node 节点底部插入新的节点

先热身一下，假设我们往一个只有两个节点的树中插入元素，如下图，根据待插入元素与已有元素的大小，又可以分为如下三种情况：

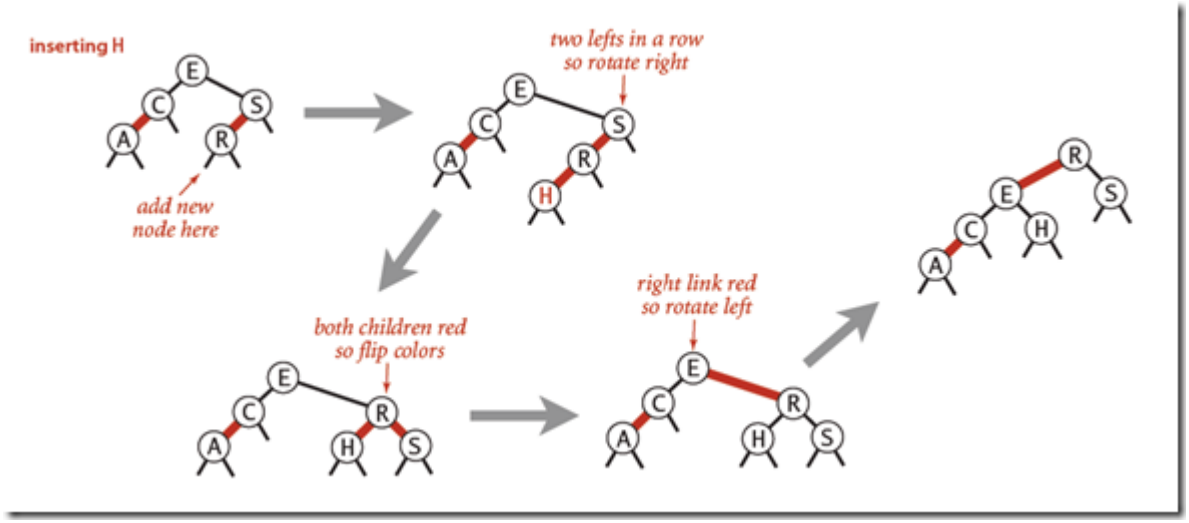


如果带插入的节点比现有的两个节点都大，这种情况最简单。我们只需要将新插入的节点连接到右边子树上即可，然后将中间的元素提升至根节点。这样根节点的左右子树都是红色的节点了，我们只需要调研 FlipColor 方法即可。其他情况经过反转操作后都会和这一样。

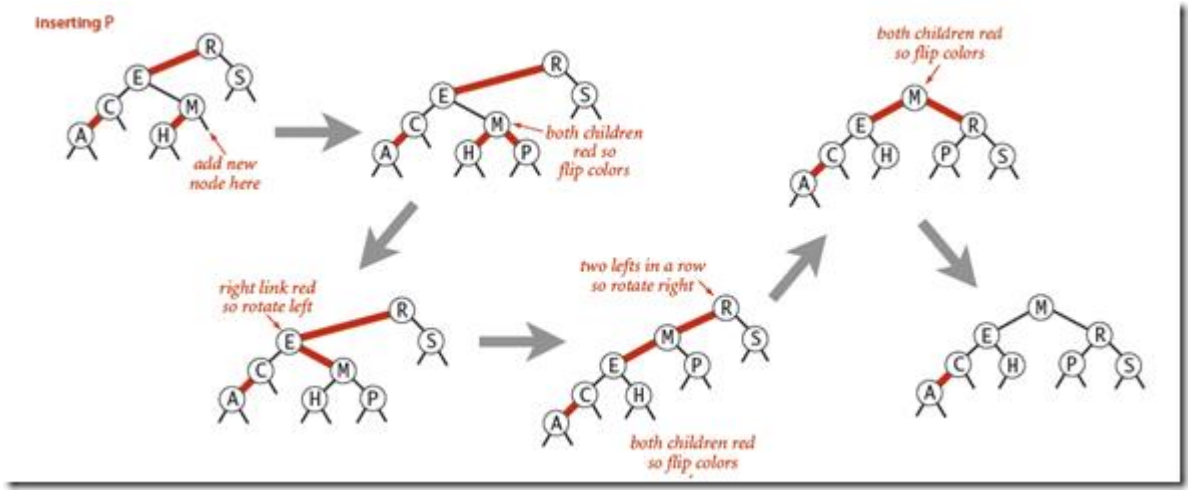
如果插入的节点比最小的元素要小，那么将新节点添加到最左侧，这样就有两个连接红色的节点了，这是对中间节点进行右旋操作，使中间结点成为根节点。这是就转换到了第一种情况，这时候只需要再进行一次 FlipColor 操作即可。

如果插入的节点的值位于两个节点之间，那么将新节点插入到左侧节点的右子节点。因为该节点的右子节点是红色的，所以需要进行左旋操作。操作完之后就变成第二种情况了，再进行一次右旋，然后再调用 FlipColor 操作即可完成平衡操作。

有了以上基础，我们现在来总结一下往一个 3-node 节点底部插入新的节点的操作步骤，下面是一个典型的操作过程图：

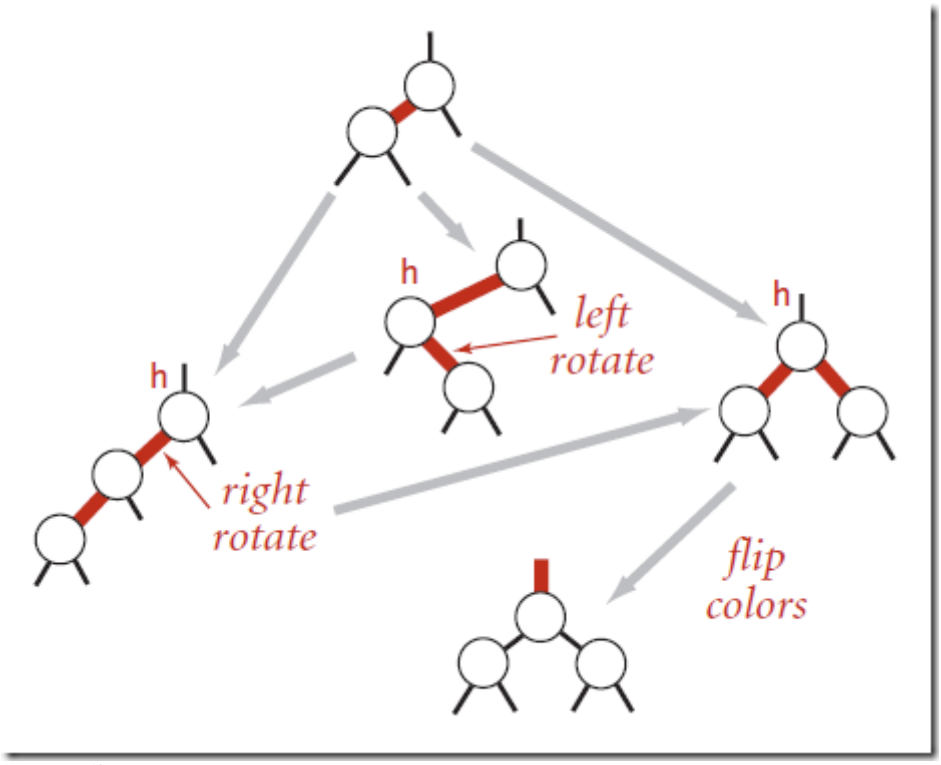


可以看出，操作步骤如下：
执行标准的二叉查找树插入操作，新插入的节点元素用红色标识。
如果需要对 4-node 节点进行旋转操作
如果需要，调用 FlipColor 方法将红色节点提升
如果需要，左旋操作使红色节点左倾。
在有些情况下，需要递归调用 Case1 Case2，来进行递归操作。如下：



代码实现

经过上面的平衡化讨论，现在就来实现插入操作，一般地插入操作就是先执行标准的二叉查找树插入，然后再进行平衡化。对照 2-3 树，我们可以通过前面讨论的，左旋，右旋，FlipColor 这三种操作来完成平衡化。



具体操作方式如下：
如果节点的右子节点为红色，且左子节点位黑色，则进行左旋操作
如果节点的左子节点为红色，并且左子节点的左子节点也为红色，则进行右旋操作
如果节点的左右子节点均为红色，则执行 FlipColor 操作，提升中间结点。
根据这一逻辑，我们就可以实现插入的 Put 方法了。

```
public override void Put(TKey key, TValue value)
{
    root = Put(root, key, value);
    root.Color = BLACK;
}

private Node Put(Node h, TKey key, TValue value)
{
    if (h == null) return new Node(key, value, 1, RED);
    int cmp = key.CompareTo(h.Key);
    if (cmp < 0) h.Left = Put(h.Left, key, value);
    else if (cmp > 0) h.Right = Put(h.Right, key, value);
    else h.Value = value;

    //平衡化操作
    if (IsRed(h.Right) && !IsRed(h.Left)) h = RotateLeft(h);
    if (IsRed(h.Right) && IsRed(h.Left.Left)) h = RotateRight(h);
    if (IsRed(h.Left) && IsRed(h.Right)) h = FlipColor(h);

    h.Number = Size(h.Left) + Size(h.Right) + 1;
    return h;
}

private int Size(Node node)
{
```

```
    if (node == null) return 0;
    return node.Number;
}
```

第 3 章 分配器的设计与实现

总分 50 分

3.1 总体设计（10 分）

介绍堆、堆中内存块的组织结构，采用的空闲块、分配块链表/树结构和相应算法等内容。

1.堆：

动态内存分配器维护着一个进程的虚拟内存区域，称为堆。系统之间细节不同，但是不失通用性，假设堆是一个请求二进制零的区域，它紧接在未初始化的数据区域后开始，并向上生长。对于每个进程，内核维护着一个变量 brk，它指向堆的顶部。

分配器将堆视为一组不同大小的块的集合来维护。每个块就是一个连续的虚拟内存片，要么是已分配的，要么是空闲的。已分配的块显式地保留为供应用程序使用。空闲块可用来分配。空闲块保持空闲，直到它显式地被应用所分配。一个已分配的块保持已分配状态，直到它被释放，这种释放要么是应用程序显式执行的，要么是内存分配器自身隐式执行的。

2.堆中内存块的组织结构：

一个块是由一个字的头部、有效载荷、可能的一些额外的填充，以及在块的结尾处的一个字的脚部组成的。头部编码了这个块的大小（包括头部和所有的填充），以及这个块是已分配的还是空闲的。如果我们强加一个双字的对齐约束条件，那么块大小就总是 8 的倍数，且块大小的最低 3 位总是 0。因此，我们只需要内存大小的 29 个高位，释放剩余的 3 位来编码其他信息。在这种情况中，我们用其中的最低位（已分配位）来指明这个块是已分配的还是空闲的。

3.采用的空闲块、分配块链表：

使用隐式的空闲链表，使用立即边界标记合并方式，最大的块为2^{32} =4GB，代码是64位干净的，即代码能不加修改地运行在32位或64位的进程中。

4. 相关算法：

int mm_init(void)函数；

void mm_free(void *ptr)函数；

void *mm_realloc(void *ptr, size_t size)函数；

int mm_check(void)函数；

void *mm_malloc(size_t size)函数。

3.2 关键函数设计（40 分）

3.2.1 int mm_init(void)函数（5 分）

函数功能：创建一个带初始空闲块的堆

处理流程：

1.在调用 mm_malloc 和 mm_free 之前，必须调用 mm_init 函数来初始化分离空闲链表和堆。
2. mm_init 函数从内存中得到四个字，并将它们初始化，创建一个空的空闲链表。
3.调用 extend_heap 函数，将堆扩展 CHUNKSIZE 字节，并且创建初始的空闲块。此刻，分配器已初始化了，并且准备好接受来自应用的分配和释放请求。
要点分析：
1.注意初始化分离空闲链表和初始化堆要知道分配器使用最小块的大小为 16 字节。创建空闲链表之后需要使用 extend_heap 来扩展堆。
2.为了保持对齐，extend_heap 将请求大小向上舍入为最接近 2 字（8 字节）的倍数，然后向内存系统请求额外的堆空间。

3.2.2 void mm_free(void *ptr)函数（5 分）

函数功能：释放所请求的块
参　　数：指向请求块首字的指针 ptr
处理流程：
1.调用 GET_SIZE(HDRP(bp))来获得请求块的大小。
2.调用 PUT(HDRP(bp), PACK(size, 0)); PUT(FTRP(bp), PACK(size, 0));将请求块的头部和脚部的已分配位置为 0，表示为 free。
3.调用 coalesce(bp);将释放的块 bp 与相邻的空闲块合并起来。
要点分析：
将请求块的 bp 标记位 free 之后，将它插入到分离的空闲链表中，注意 free 块需要和与之相邻的空闲块使用边界标记合并。

3.2.3 void *mm_realloc(void *ptr, size_t size)函数（5 分）

函数功能：向 ptr 所指的块重新分配一个具有至少 size 字节的有效负载的块
参　　数：指向请求块首字的指针 ptr，需要分配的字节 size
处理流程：
1. 首先检查请求的真假，在检查完请求的真假后，分配器必须调整请求块的大小。从而为头部和脚部留有空间，并满足双字对齐的要求。第 12~13 行强制了最小块大小是 16 字节；8 字节用来满足对齐要求，而另外 8 个用来放头部和脚部。对于超过 8 字节的请求，一般的规则是加上开销字节，然后向上舍入到最接近的 8 的整数倍。
2.,执行 copySize = GET_SIZE(HDRP(ptr));得到 ptr 的块大小,如果 size 比 copy_size 小，则更新 copy_size 为 size，然后释放 ptr。
要点分析：
当需要重新分配的大小 size 小于原来的 ptr 指向的块的大小时,注意更新 copy_sized 的值。

3.2.4 int mm_check(void)函数（5 分）

函数功能：检查堆是否一致
处理流程：
1.首先定义指针 bp，将其初始化为指向序言块的全局变量 heap_listp。其后的操作大多数都是在 verbose 不为零时才执行的。最开始检查序言块，当序言块不是 8 字节的已分配块，就会打印 Bad prologue header。
2.对于 checkblock 函数：作用为检查是都为双字对齐，然后通过获得 bp 所指块的头部和脚部指针，判断两者是否匹配，不匹配的话就返回错误信息。
3.检查所有 size 大于 0 的块，如果 verbose 不为 0，就执行 printblock 函数。
4.检查结尾块。当结尾块不是一个大小为 0 的已分配块，就会打印 Bad epilogue header 。
要点分析：
checkheap 主要检查了堆序言块和结尾块，所有 size 大于 0 的块都需要检查是否双字对齐和头部脚部 match，并且打印块的头部和脚部的信息。

3.2.5 void *mm_malloc(size_t size)函数（10 分）

函数功能：向内存请求大小为 size 字节的块
参　　数：size
处理流程：
1.首先检查请求的真假，在检查完请求的真假后，分配器必须调整请求块的大小。从而为头部和脚部留有空间，并满足双字对齐的要求。第 12~13 行强制了最小块大小是 16 字节；8 字节用来满足对齐要求，而另外 8 个用来放头部和脚部。对于超过 8 字节的请求，一般的规则是加上开销字节，然后向上舍入到最接近的 8 的整数倍。
2.分配器调整了请求的大小后，就会搜索空闲链表，寻找一个合适的空闲块。如果有合适的，那么分配器就放置这个请求块，并可选址地分割出多余的部分，然后返回新分配块的地址。
要点分析：
mm_malloc 函数是为了更新 size 来满足要求的大小，然后在分离空闲链表数组里面找到合适的请求块，找不到的话就使用一个新的空闲块来扩展堆。

3.2.6 static void *coalesce(void *bp)函数（10 分）

函数功能：使用边界标记合并技术将之与邻接的空闲块合并起来。
处理流程：
1.首先参考书上的代码，获得前一块和后一块的已分配位，并且调用 GET_SIZE(HDRP(bp))获得 bp 所指向的块的大小。

```
size_t    prev_alloc = GET_ALLOC(HDRP(PREV_BLKP(bp)));
size_t    next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
size_t size = GET_SIZE(HDRP(bp));
```

2.然后根据 bp 所指向的相邻块，可以得到四种情况：

(1).前后均为 allocated 块，不做合并，直接返回

```
if (prev_alloc && next_alloc)     /*case1*/
{
    return bp;
}
```

(2).前面的块是 allocated，但是后面的块是 free 的，这时将两个 free 块合并

```
else if (prev_alloc && !next_alloc)     /*case2*/
{
    size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
}
```

(3).后面的块是 allocated，但是前面的块是 free 的，这时将两个 free 块合并

```
else if (!prev_alloc && next_alloc)     /*case3*/
{
    size += GET_SIZE(HDRP(PREV_BLKP(bp)));
    PUT(FTRP(bp), PACK(size, 0));
    PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
    bp = PREV_BLKP(bp);
}
```

(4).前后两个块都是 free 块，这时将三个块同时合并

```
else     /*case4*/
{
    size += GET_SIZE(HDRP(PREV_BLKP(bp))) + GET_SIZE(HDRP(NEXT_BLKP(bp)));
    PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
    PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
    bp = PREV_BLKP(bp);
}
```

3.四种操作后更新的 bp 所指向的块插入分离空闲链表

要点分析：

使用的空闲链表格式允许我们忽略潜在的麻烦边界情况，也就是请求块 bp 在堆的起始处或者堆的结尾处，如果没有这些特殊块，代码将混乱的多，更加容易出错，并且更慢，因为我们将不得不在每次释放请求时，都去检查这些并不常见的边界情况。

第 4 章测试

总分 10 分

4.1 测试方法

生成可执行评测程序文件的方法：

linux>make

评测方法:

mdriver [-hvVa] [-f <file>]

选项：

-a 不检查分组信息

-f <file> 使用 <file>作为单个的测试轨迹文件

-h 显示帮助信息

-l 也运行 C 库的 malloc

-v 输出每个轨迹文件性能

-V 输出额外的调试信息

轨迹文件：指示测试驱动程序 mdriver 以一定顺序调用

性能分 pindex 是空间利用率和吞吐率的线性组合

获得测试总分 linux>./mdriver -av -t traces/

4.2 测试结果评价

由于没有使用分离的空闲链表，所以测试结果不理想

4.3 自测试结果

```
yt1180300829@ubuntu:~/hitics/lab8/malloclab-handout$ ./mdriver -av -t traces
Using default tracefiles in traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.009007   632
 1       yes   99%    5848  0.008121   720
 2       yes   99%    6648  0.013957   476
 3       yes  100%    5380  0.009772   551
 4       yes   66%   14400  0.000105137536
 5       yes   92%    4800  0.009602   500
 6       yes   92%    4800  0.008922   538
 7       yes   55%   12000  0.178974    67
 8       yes   51%   24000  0.345570    69
 9       yes   27%   14401  0.078181   184
10       yes   34%   14401  0.002870  5019
Total          74%  112372  0.665079   169

Perf index = 44 (util) + 11 (thru) = 56/100
```

第 5 章 总结

5.1 请总结本次实验的收获

1.明白了动态内存分配的原理。
2.知道的堆的运行规则。

5.2 请给出对本次实验内容的建议

注：本章为酌情加分项。

参考文献

为完成本次实验你翻阅的书籍与网站等

[1] 林来兴. 空间控制技术[M]. 北京：中国宇航出版社，1992：25-42.

[2] 辛希孟. 信息技术与信息服务国际研讨会论文集：A 集[C]. 北京：中国科学出版社，1999.

[3] 赵耀东. 新时代的工业工程师[M/OL]. 台北：天下文化出版社，1998 [1998-09-26]. http://www.ie.nthu.edu.tw/info/ie.newie.htm（Big5）.

[4] 谌颖. 空间交会控制理论与方法研究[D]. 哈尔滨：哈尔滨工业大学，1992：8-13.

[5] KANAMORI H. Shaking Without Quaking[J]. Science，1998，279（5359）：2063-2064.

[6] CHRISTINE M. Plant Physiology: Plant Biology in the Genome Era[J/OL]. Science，1998，281：331-332[1998-09-23]. http://www.sciencemag.org/cgi/collection/anatmorp.