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 计算机科学与技术

学　　号 1180300315

班　　级 1836101

学 生 姓 名 周牧云

指 导 教 师

实 验 地 点

实 验 日 期

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

个人笔记本电脑

实验环境与工具所列明软件

参考手册: Linux 环境下的命令；GCC 手册；GDB 手册

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。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：显式分配器和隐式分配器。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。

1.显式分配器：要求应用显式地释放任何已分配的块。例如 C 程序通过调用 malloc 函数来分配一个块，通过调用 free 函数来释放一个块。其中 malloc 采用的总体策略是：先系统调用 sbrk 一次，会得到一段较大的并且是连续的空间。进程把系统内核分配给自己的这段空间留着慢慢用。之后调用 malloc 时就从这段空间中分配，free 回收时就再还回来（而不是还给系统内核）。只有当这段空间全部被分配掉时还不够用时，才再次系统调用 sbrk。当然，这一次调用 sbrk 后内核分配给进程的空间和刚才的那块空间一般不会是相邻的。

2.隐式分配器：也叫做垃圾收集器，例如，诸如 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）

对于带边界标签的隐式空闲链表分配器，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。

我们将对组织为一个连续的已分配块和空闲块的序列，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。

Knuth 提出了一种边界标记技术，允许在常数时间内进行对前面快的合并。这种思想是在每个块的结尾处添加一个脚部，其中脚部就是头部的一个副本。如果每个块包括这样一个脚部，那么分配器就可以通过检查它的脚部，判断前面一个块的起始位置和状态，这个脚部总是在距当前块开始位置一个字的距离。

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）

红黑树的结构：

一颗二叉搜索树，每个节点上增加了一个存储位来表示节点的颜色，通过对任意一条叶子节点到根节点的路径上的颜色进行约束，保证最长路径不超过最短路径的两倍，因此红黑树是近似平衡。

满足下述特性的二叉搜索树是红黑树：

1、每个节点不是红色就是黑色。

2、根节点为黑色。

3、每个叶子节点都是黑色的。

4、每个红色节点的子节点都是黑色。

5、任意节点，到其任意叶节点的所有路径都包含相同的黑色节点。

红黑树的查找：

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。红黑树的主要是想对 2-3 查找树进行编码，尤其是对 2-3 查找树中的 3-nodes 节点添加额外的信息。红黑树中将节点之间的链接分为两种不同类型，红色链接，他用来链接两个 2-nodes 节点来表示一个 3-nodes 节点。黑色链接用来链接普通的 2-3 节点。特别的，使用红色链接的两个 2-nodes 来表示一个 3-nodes 节点，并且向左倾斜，即一个 2-node 是另一个 2-node 的左子节点。这种做法的好处是查找的时候不用做任何修改，和普通的二叉查找树相同。

红黑树的插入：

在插入新节点的时候很显然，不会违背特性 1 和 3，如果插入的是根节点直接将根节点着色为黑色即可，这种情况可以忽略不计，所以插入节点时可能会违背了 4 和 5，又因为插入的是红色节点因此 5 也不会违背，最后在插入新节点的时候

我们只需要关注特性 4 就可以了。当父节点为红色的时候跟 4 有冲突，所以我们接下来讨论的就是这种情况。我们知道，在插入新节点之前整颗红黑树是平衡的，因此可以得出一个结论就是祖父节点肯定肯定是黑色的。我们现在只关注相关的节点即可，目前，我们知道了祖父的节点为黑色，父节点为红色，但是叔叔节点的颜色不知道，新节点的位置也不能确定，所以有 2x2 中情况，当叔叔节点为红色的时候，两种情况的处理方式是一致的，所以最后我们可以总结为 3 中情况：

1、叔叔节点为红色

2、新节点为右节点，叔叔节点为黑色

3、新节点为左节点，叔叔节点为黑色

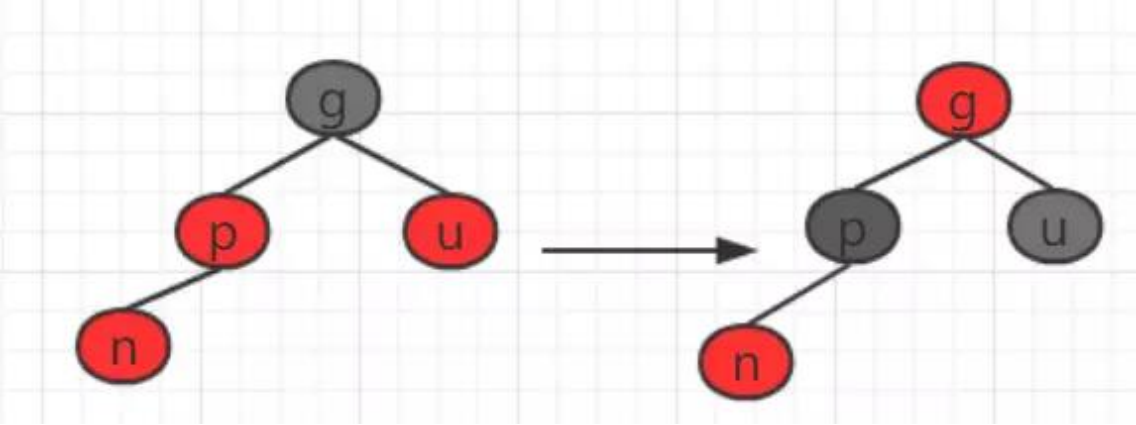
情况 1 叔叔节点为红色

1、父节点设为黑色

2、叔叔节点设为黑色

3、祖父节点设为红色

4、把祖父节点设置为新节点（当前节点）

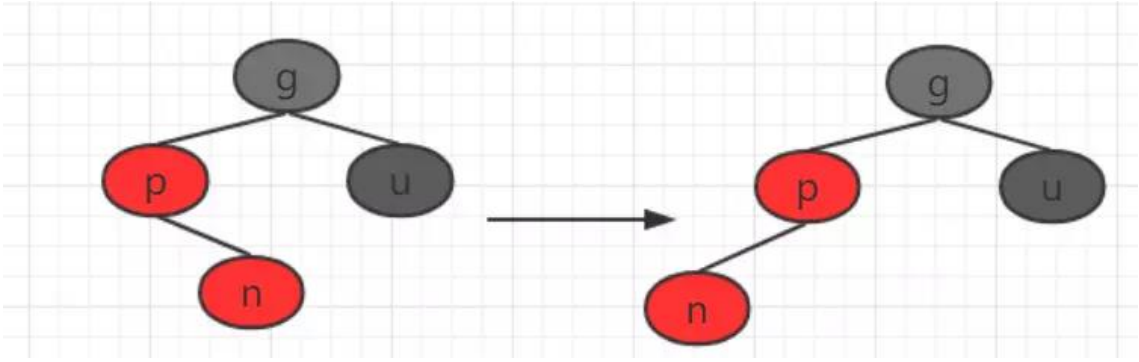


情况 2 新节点为右节点，叔叔节点为黑色

1、以父节点为支点左旋

2、父节点和新节点互换位置

3、把父节点设为当前节点

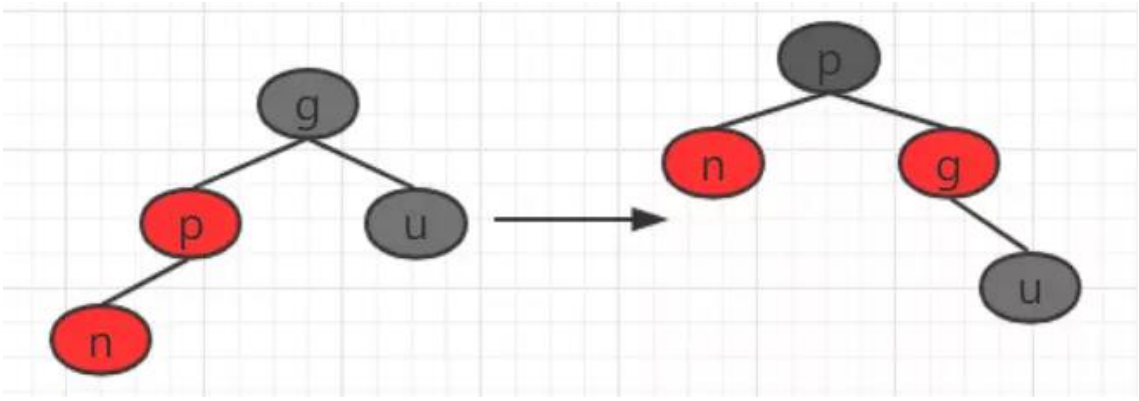


情况 3 新节点为左节点，叔叔节点为黑色

1、父节点设为黑色

2、祖父节点设为红色

3、以祖父节点为支点右旋



红黑树删除：

删除一个节点的时候有 3 中情况：

1、删除节点没有子节点

2、删除节点只有一个子节点

3、删除节点有两个子节点

首先，我们逐个来分析每种情况删除节点后对整颗红黑树的平衡性的影响。在删除节点时候红黑树的特性 1，2，3 肯定不会违背，所以只需要考虑特性 4，5 即可。

对于情况 1，肯定不会违背特性 4，如果删除节点为红色，那么对整颗红黑树的平衡性都不会影响，如果是黑色则违背了特性 5，我们先将这种情况记录下来，稍后再进一步讨论。

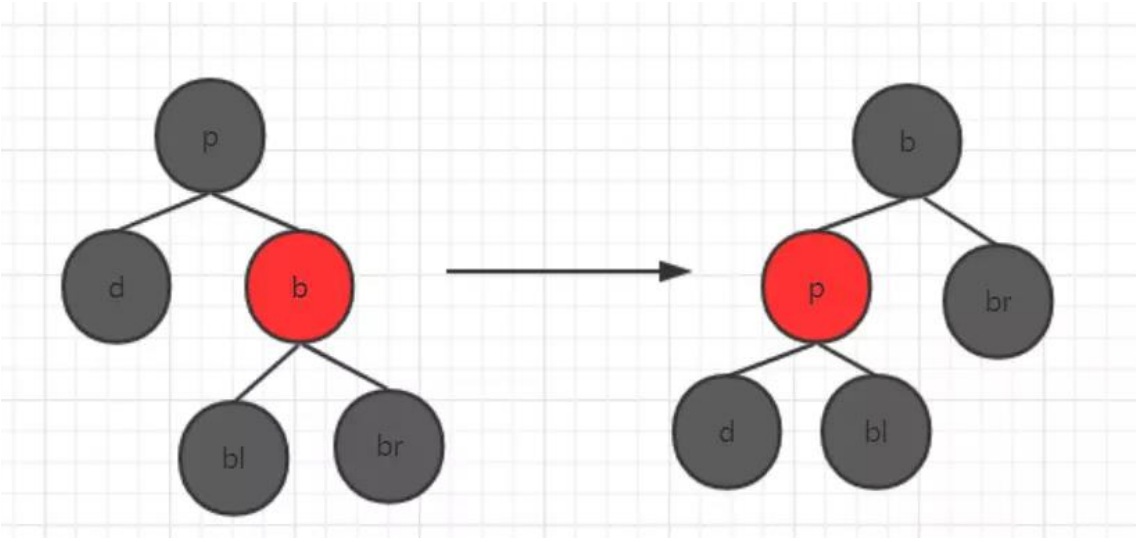
对于情况 2，有可能删除的是左子树或右子树，暂且不讨论。如果删除的节点为红色，不影响平衡性，如果删除的是黑色，那么肯定会和特性 5 有冲突，当删除节点的父节点为红色，子节点为红色是也和特性 4 有冲突。

对于情况 3，其实最后删除的是它的替代节点，根据替代节点的特点，最终其实是回到了 1 这种情况或者情况 2。

总结上面的 3 种情况可得到一个结论，只有删除节点为黑色时才会破坏红黑树原来的平衡，因在删除节点之前红黑树是出于平衡状态的，删除之后很明显的其兄弟节点分支必然比删除节点的分支多了一个黑色的节点，因此我们只需要改变兄弟节点的颜色即可，我们只讨论左节点，右节点对称。

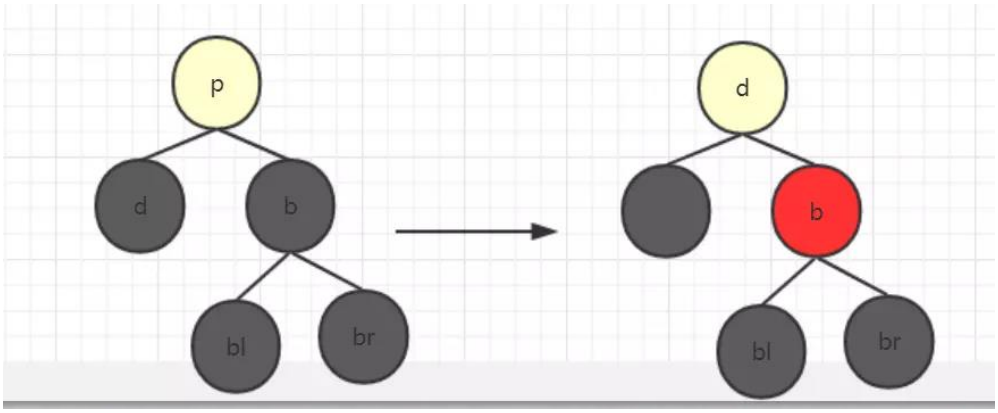
一、删除节点的兄弟节点是红色

将兄弟节点设为黑色，父节点设为红色，以父节点为支点左旋转，然后将父节点的右节点放到兄弟节点上：



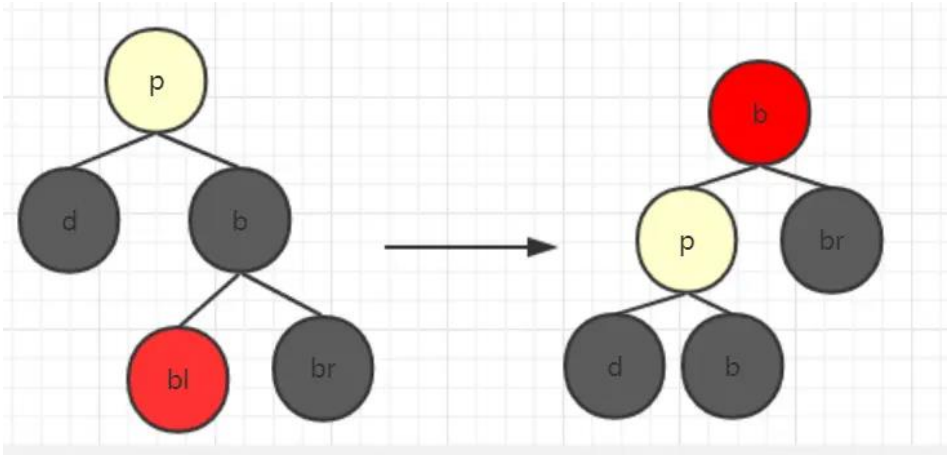
二、兄弟节点是黑色的，兄弟的两个子节点也都是黑色的

兄弟节点设为红色，把父节点设置为新的删除节点：



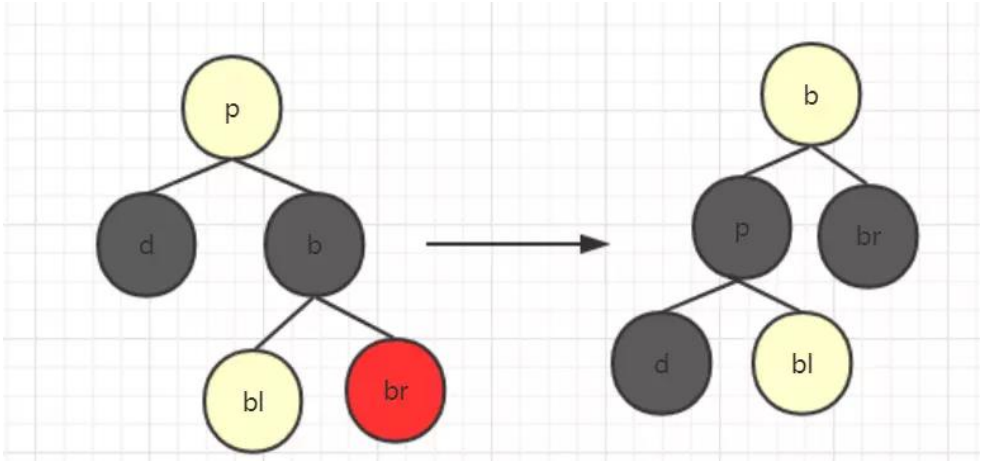
三、兄弟节点是黑色的，且兄弟节点的左子节点是红色，右子节点是黑色

将兄弟节点的左子节点设为黑色，兄弟节点设为红色，以兄弟节点为支点右旋，把父节点的右节点设置为兄弟节点



四、兄弟节点是黑色的，且兄弟节点的右子节点是红色，左子节点任意颜色

把兄弟节点的设为父节点的颜色，父节点设为黑色，父节点的右节点设为黑色，父节点为支点左旋



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。简单来说，动态分配器就是我们平时在 C 语言上用的 malloc 和 free,realloc，通过分配堆上的内存给程序，我们通过向堆申请一块连续的内存，然后将堆中连续的内存按 malloc 所需要的块来分配，不够了，就继续向堆申请新的内存，也就是扩展堆，这里设定，堆顶指针想上伸展（堆的大小变大）。

2.堆中内存块的组织结构：用隐式空闲链表来组织堆，具体组织的算法在 mm_init 函数中。对于带边界标签的隐式空闲链表分配器，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.对于空闲块和分配块链表：采用分离的空闲链表。全局变量： void *Lists[MAX_LEN]; 因为一个使用单向空闲块链表的分配器需要与空闲块数量呈线性关系的时间来分配块，而此堆的设计采用分离存储的来减少分配时间，就是维护多个空闲链表，每个链表中的块有大致相等的大小。将所有可能的块大小根据 2 的幂划分。

4.放置策略（适配方式）：首次适配（其实有着最佳适配的效果）。malloc 搜索块的时间从所有空的空闲块降低到局部链表的空闲块中，当分到对应的大小类链表的时候，它的空间也会在大小类链表的范围里面，这样使得即使是首次适配也可以是空间利用率接近最佳适配。进一步解释：当空闲链表按照块大小递增的顺序排序时，首次适配是选择第一个合适的空闲块，最佳适配是选择所需请求大小最小的空闲块，也是会选择第一个合适的空闲块，后面的块大小递增，不再选择。因此两种适配算法效率近似。

5.关于链表操作主要函数和算法：
static void InsertNode(void *bp, size_t size)
static void DeleteNode(void *bp)
程序中大部分函数在后面都会介绍到，因此在这里只简单分析 InsertNode 和

DeleteNode 函数，分别用来插入分离的空闲链表，和从分离的空闲链表中删除。
在介绍这两个函数之前，先阐明新的宏定义：
#define SET_PTR(p, bp) (*(unsigned int *)(p) = (unsigned int)(bp))
/*将 bp 写入参数 p 指的字中*/
#define PRED_PTR(bp) ((char *)(bp)) /*祖先节点*/
#define SUCC_PTR(bp) ((char *)(bp) + WSIZE) /*后继节点*/
#define PRED(bp) (*(char **)(bp))
#define SUCC(bp) (*(char **)(SUCC_PTR(bp)))
根据分配器的设计，后面两个宏定义分别表示 size 更大块的指针和 size 更小块的指针。

InsertNode(void *bp, size_t size)函数：
1.将 free 块插入分离空闲链表，首先要在链表数组中，找到块的大小类，从而找到对应的分离空闲链表；其次，找到链表后，需要根据 size 的比较一直循环，直到链中的下一个块比 bp 所指的块大为止，以保持链表中的块由小到大排列，方便之后的适配。
2.insert_bp 表示的是待插入的位置，search_bp 表示的是比 bp 所指块更大的块的指针，找到对应位置，有四种情况：
如果 search_bp != NULL，那么可能是在中间插入，或者在 List[i]首地址之后插入（并且此时 List[i]后面不是空）
否则 search_bp = NULL，那么可能是在结尾插入，或者该 List[i]链表原本就为空，在其首地址插入即可。

DeleteNode(void *bp)函数：
将块从分离空闲链表中删除，其实和插入的操作类似。
1. 首先要在链表数组中，找到块的大小类，从而找到对应的分离空闲链表
2.从链表中删除块的时候，也分四种情况：在链表的中间删除；在表头删除，并且删除后 List[i]不是空表；在链表的结尾删除；在 List[i]表头删除，并且原本 bp 所指的块就是表中最后一个块。

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：初始化内存系统模型

处理流程：
第一步：初始化分离空闲链表，将分离空闲链表全部初始化为 NULL。
第二步：mm_init 函数从内存中得到四个字，并且将堆初始化，创建一个空的空闲

链表。
第三步：调用 extend_heap 函数，这个函数将堆扩展 INITCHUNKSIZE 字节，并且创建初始的空闲块。

要点分析：初始化分离空闲链表和初始化堆的过程主要是要了解 mm_init 函数初始化分配器时，分配器使用最小块的大小是 16 字节，空闲链表组织成一个隐式空闲链表，它的恒定形式便是一个双字边界对齐不使用的填充字+8 字节的序言块+4 字节的结尾块。另外空闲链表创建之后需要使用 extend_heap 函数来扩展堆。

3.2.2 void mm_free(void *ptr)函数（5 分）

函数功能：释放一个块

参　　数：指针 ptr

处理流程：
第一步：通过 GET_SIZE(HDRP(bp))来获得请求块的大小，并且使用 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 。

第二步：调用插入分离空闲链表函数 InsertNode(bp, size); 将 free 块插入到分离空闲链表中。

第三步：调用 coalesce(bp); 使用边界标记合并技术将释放的块 bp 与相邻的空闲块合并起来。

要点分析：此函数的要点在于将请求块 bp 标记为 free 后，需要将它插入到分离的空闲链表中，并且注意 free 块需要和与之相邻的空闲块使用边界标记合并技术进行合并。

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：待处理的块第一个字的指针 ptr，需要分配的字节 size

处理流程：
第一步：在检查完请求的真假之后，分配器必须表征请求块的大小，并且满足双字对齐的要求。操作强制了最小块的大小是 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 8 的整数倍。

第二步：
1.如果 size 小于原来块的大小，直接返回原来的块。

2.如果 size 大于原来块的大小，先检查地址连续下一个块是否为未分配块或者该块是堆的结束块，因为我们要尽可能利用相邻的 free 块，以此减小外部碎片。如果加上后面连续地址上的未分配块空间也不够，那么需要 extend_heap(MAX(-remaining, CHUNKSIZE)) 来扩展块。这时从分离空闲链表中删除刚刚利用的未分配块并设置新块的头尾。

3.如果此时没有可以利用的连续未分配块，那么只能申请新的不连续的未分配块，使用 memcpy(new_p, bp, GET_SIZE(HDRP(bp)));复制原块内容并且释放原块。

要点分析：先要实现内存对齐，调整 size 的大小。后面是一个找到合适块的过程，其中为了减少外部碎片，需要尽可能利用相邻的块，如果没有可以利用的连续未分配的块，此时只能申请新的而不连续的未分配块。

3.2.4 int mm_check(void)函数（5 分）

函数功能：检查堆的一致性

处理流程：
第一步：先定义指针 bp，初始化为指向序言块的全局变量 heap_listp。后面的操作大多数都是在 verbose 不为零时执行的。最初是检查序言块，如果序言块不是 8 字节的已分配块，则会打印 Bad prologue header。

第二步：checkblock 函数。
checkblock 函数的主要功能就是检查是否双字对齐，并且通过获得 bp 所指块的头头部和脚部指针，判断二者是否匹配，如果不匹配，则返回错误信息。

第三步：检查所有 size 大于 0 的块，如果 verbose 部位零，则执行 printblock 函数，对于 printblock 函数，先获得从 bp 所指的块的头部和脚部分别返回的大小和已分

配位，然后打印信息，如果头部返回的大小为 0，则 printf("%p: EOL\n", bp); 之后再分别打印头部和脚部的信息，其中'a' 和 'f' 分别表示 allocated 和 free，对应的是已分配位的信息。

第四步：最后检查结尾块。如果结尾块不是一个大小位零的已分配块，则会打印出 Bad epilogue header 。

要点分析：总结来说，checkheap 函数主要检查了堆序言块和结尾块，每个 size 大于 0 的块是否双字对齐和头部脚部 match，并且打印了块的头部和脚部的信息。

事实上 checkheap 函数只是对堆一致性的简单检查，如空闲块是否都在空闲链表等方面并没有展开检查。

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

函数功能：向内存请求分配一个具有至少 size 字节的有效负载的块。

参　　数：向内存请求块大小 size 字节

处理流程：
第一步：在检查完请求的真假之后，分配器必须表征请求块的大小，从而为头部和脚部留有空间，并且满足双字对齐的要求。操作强制了最小块的大小是 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 8 的整数倍。

第二步：当分配其调整了请求的大小，它就会搜索空闲链表，寻找一个合适的空闲块。寻找合适空闲块的过程为下列的循环语句：（步骤带有注释）
其中 MAX_LEN 为分离空闲链表数组的大小。

第三步：如果有合适的，那么分配器就用 place 函数放置这个请求块，并且分割出多余的部分，然后返回新分配块的地址。如果分配器不能够发现一个匹配的块，那么就用一个新的空闲块来扩展堆，同样把请求块放置在这个新的空闲块里，可选地分割这个块，然后返回一个指向这个新分配块的指针。

要点分析：

mm_malloc 函数主要是更新 size 为满足要求的大小，然后在分离空闲链表数组中找合适的请求块，如果找不到则用一个新的空闲块来扩展堆。注意每次都要使用 place 函数放置请求块，并可选地分割出多余的部分。

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

函数功能：边界标记合并。将指针返回到合并块。

处理流程：
第一步：获得前一块和后一块的已分配位，并且获得 bp 所指块的大小。

第二步：根据 bp 所指块相邻块的情况，可以得到以下四种可能性：
1.前面的和后面的块都已分配。此时不进行合并，所以当前块直接返回就可以了。
2.前面块已分配，后面块空闲。先把当前块和后面块从分离空闲链表中删除。然后此时当前块和后面块合并，用当前块和后面块的大小的和来更新当前块的头部和脚部。
3.前面块空闲，后面块已分配。先把当前块和前面块从分离链表中删除。然后此时将前面块和当前块合并，用两个块大小的和来更新前面块的头部和当前块的脚部。
4.前面块和后面块都空闲。先把前面块、当前块和后面块从分离链表中删除。然后合并所有的三个块形成一个单独的空闲块，用三个块大小的和来更新前面块的头部和后面块的脚部。

第三步：将上述四种操作后更新的 bp 所指的块插入分离空闲链表。

要点分析：获得了当前块相邻块的情况之后，主要是处理不同的四种情况。合并前首先要把待合并的块从分离空闲链表中删除，合并后注意更新总合并块的头部和脚部，大小为总合并块之和。最后需要把更新的 bp 所指的块插入到分离空闲链表中即可。

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

linux>./mdriver -av -t traces/

4.2 测试结果评价

Trace 7、8、9、10 表现欠佳

4.3 自测试结果

```
gcc -Wall -O2 -m32   -c -o memlib.o memlib.c
gcc -Wall -O2 -m32   -c -o fsecs.o fsecs.c
gcc -Wall -O2 -m32   -c -o fcyc.o fcyc.c
gcc -Wall -O2 -m32   -c -o clock.o clock.c
gcc -Wall -O2 -m32   -c -o ftimer.o ftimer.c
gcc -Wall -O2 -m32 -o mdriver mdriver.o mm.o memlib.o fsecs.o fcyc.o clock.o ft
imer.o
zhoumuyun@ubuntu:~/Downloads/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  0.006773   841
 1       yes   99%    5848  0.005744  1018
 2       yes   99%    6648  0.009778   680
 3       yes  100%    5380  0.007180   749
 4       yes   66%   14400  0.000078183908
 5       yes   92%    4800  0.006366   754
 6       yes   92%    4800  0.005883   816
 7       yes   55%   12000  0.135512    89
 8       yes   51%   24000  0.251980    95
 9       yes   27%   14401  0.057641   250
10       yes   34%   14401  0.002065  6975
Total          74%  112372  0.488999   230

Perf index = 44 (util) + 15 (thru) = 60/100
```

第 5 章 总结

5.1 请总结本次实验的收获

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.