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 1171000410

班　　级 1703005

学 生 姓 名 强文杰

指 导 教 师 吴锐

实 验 地 点 G712

实 验 日 期 2018.12.16

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

codeblocks

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：

红黑树是一种近似平衡的二叉查找树，它能够确保任何一个节点的左右子树的高度差不会超过二者中较低那个的一陪。具体来说，红黑树是满足如下条件的二叉查找树（binary search tree）：

1. 每个节点要么是红色，要么是黑色。
2. 根节点必须是黑色
3. 红色节点不能连续（也即是，红色节点的孩子和父亲都不能是红色）。
4. 对于每个节点，从该点至 null（树尾端）的任何路径，都含有相同个数的黑色节点。

在树的结构发生改变时（插入或者删除操作），往往会破坏上述条件 3 或条件 4，需要通过调整使得查找树重新满足红黑树的条件。

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。

```
//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);
}
```

红黑树的更新：

与 2-3 树对应关系

如果将一棵红黑树中的红链接画平，那么所有的空链接到根结点的距离都将是相同的。如果我们将由红链接相连的节点合并，得到的就是一棵 2-3 树。图 3.3.13

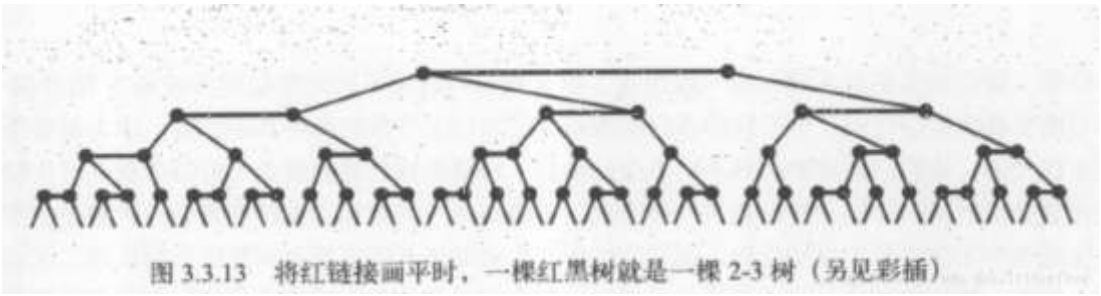

图 3.3.13 将红链接画平时，一棵红黑树就是一棵 2-3 树（另见彩插）

旋转

修复红黑树，使得红黑树中不存在红色右链接或两条连续的红链接。

左旋：将红色的右链接转化为红色的左链接

右旋：将红色的左链接转化为红色的右链接，代码与左旋完全相同，只要将 left 换成 right 即可。

插入结点

在插入新的键时，我们可以使用旋转操作帮助我们保证 2-3 树和红黑树之间的一一对应关系，因为旋转操作可以保持红黑树的两个重要性质：有序性和完美平衡性。也就是说，我们在红黑树中进行旋转时无需为树的有序性或者完美平衡性担心。下面我们来看看应该如何使用旋转操作来保持红黑树的另外两个重要性质：不存在两条连续的红链接和不存在红色的右链接。我们先用一些简单的情况热热身。

1.向树底部的 2-结点插入新键

一棵只含有一个键的红黑树只含有一个 2-结点。插入另一个键之后，我们马上就需要将他们旋转。如果新键小于老键，我们只需要新增一个红色的节点即可，新的红黑树和单个 3-结点完全等价。如果新键大于老键，那么新增的红色节点将会产生一条红色的右链接。我们需要使用 parent = rotateLeft(parent);来将其旋转为红色左链接并修正根结点的链接，插入才算完成。两种情况均把一个 2-结点转换为一个 3-结点，树的黑链接高度不变，如图 3.3.18 和 3.3.19

2.向一棵双键树（即一个 3-结点）中插入新键

这种情况又可分为三种子情况：新键小于树中的两个键，在两者之间，或是大于树中的两个键。每种情况中都会产生一个同时链接到两条红链接的结点，而我们的目标就是修正这一点。

三者中最简单的情况是新键大于原树中的两个键，因此它被链接到 3-结点的右链接。此时树是平衡的，根结点为中间大小的键，它有两条红链接分别和较小和较大的结点相连。如果我们将两条链接的颜色都由红变黑，那么我们就得到了一棵由三个结点组成，高为 2 的平衡树。它正好能够对应一棵 2-3 树，如图 3.3.20(左)。其他两种情况最终也会转化为这两种情况。

如果新键小于原书中的两个键，它会被链接到最左边的空链接，这样就产生了两条连续的红链接，如果 3.3.20（中）。此时我们只需要将上层的红链接右旋转即可得到第一种情况。

如果新键介于原书中的两个键之间，这又会产生两条连续的红链接，一条红色左链接接一条红色右链接，如果 3.3.20（右）。此时我们只需要将下层的红链接左旋即可看得到第二种情况。

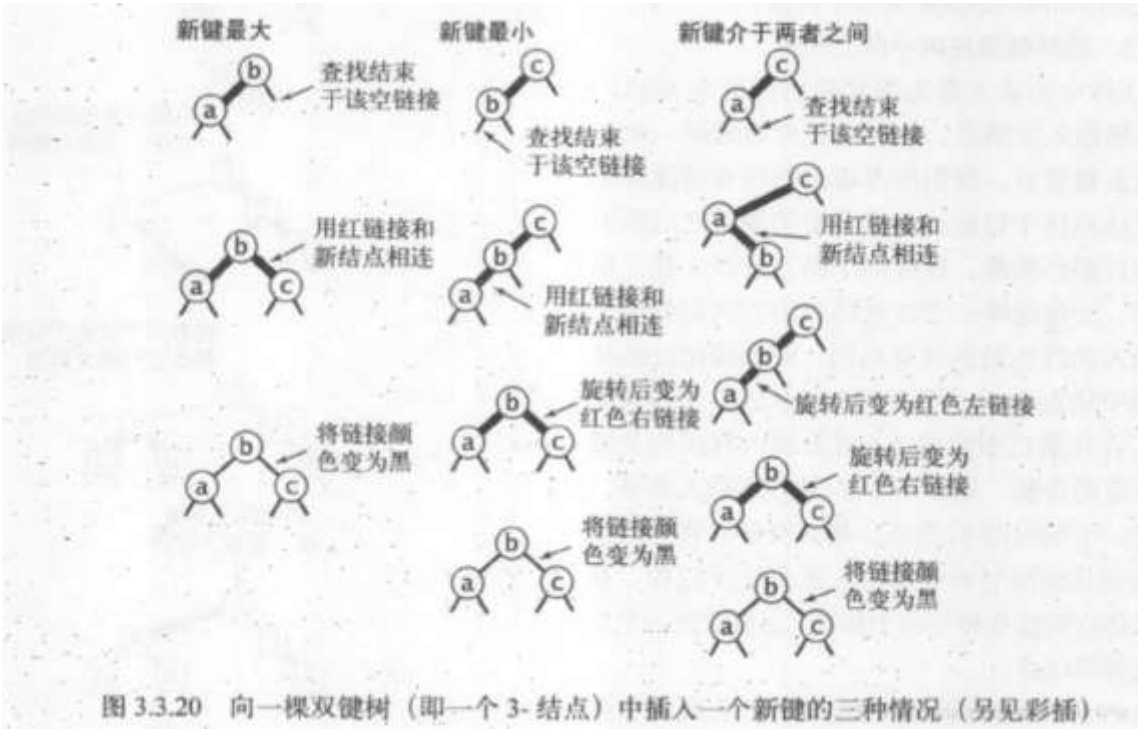

图 3.3.20 向一棵双键树（即一个 3-结点）中插入一个新键的三种情况（另见彩插）

4.根结点总是黑色

颜色转换会使根结点变为红色，我们在每次插入操作后都会将根结点设为黑色。

5.向树底部的 3-结点插入新键

现在假设我们需要在树的底部的一个 3-结点下加入一个新结点。前面讨论过的三种情况都会出现，如图 3.3.22 所示。颜色转换会使指向中结点的链接变红，相当于将它送入了父结点。这意味着在父结点中继续插入一个新键，我们也会继续用相同的办法解决这个问题。

6.将红链接在树中向上传递

2-3 树中的插入算法需要我们分解 3-结点，将中间键插入父结点，如此这般知道遇到一个 2-结点或是根结点。总之，只要谨慎地使用左旋，右旋，颜色转换这三种简单的操作，我们就能保证插入操作后红黑树和 2-3 树的一一对应关系。在沿着插入点到根结点的路径向上移动时在所经过的每个结点中顺序完成以下操作，我们就能完成插入操作：

如果右子结点是红色的而左子结点是黑色的，进行左旋转

如果左子结点是红色的且她的左子结点也是红色的，进行右旋

如果左右子结点均为红色，进行颜色转换。

删除操作

要描述删除算法，首先要回到 2-3 树。和插入操作一样，我们也可以定义一系列局部变换来在删除一个结点的同时保持树的完美平衡性。这个过程比插入一个结点更加复杂，因为我们不仅要在（为了删除一个结点而）构造临时 4-结点时沿着查找路径向下进行变换，还要在分解遗留的 4-结点时沿着查找路径向上进行变换（同插入操作）。

1.自顶向下的 2-3-4 树

作为第一轮热身，我们先学习一个沿着查找路径既能向上也能向下进行变换的稍简单的算法：2-3-4 树的插入算法，2-3-4 树中允许存在我们以前见过的 4-结点。它的插入算法沿着查找路径向下进行变换是为了保证当前结点不是 4-结点（这样树底才有空间来插入新的键），沿着查找路径向上进行变换是为了将之前创建的

4-结点配平，如图 3.3.25 所示。

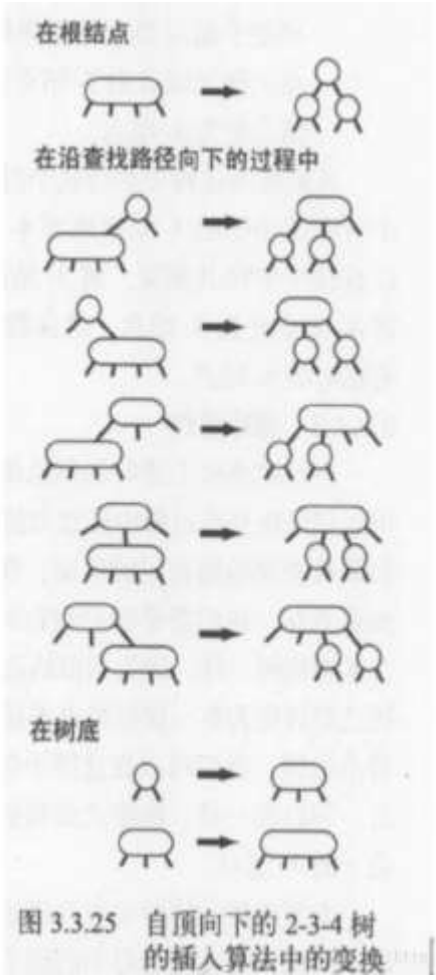

图 3.3.25 自顶向下的 2-3-4 树的插入算法中的变换

向下的变换和我们在 2-3 树中分解 4-结点所进行的变换完全相同。如果根结点是 4-结点，我们就将它分解成三个 2-结点，使得树高加 1。在向下查找的过程中，如果遇到一个父结点为 2-结点的 4-结点，我们将 4-结点分解为两个 2-结点并将中间键传递给他的父结点，使得父结点变为一个 3-结点；如果遇到一个父结点为 3-结点的 4-结点，我们将 4-结点分解为两个 2-结点并将中间键传递给它的父结点，使得父结点变为一个 4-结点；我们不必担心会遇到父结点为 4-结点的 4-结点，因为插入算法本身就保证了这种情况不会出现。到达树的底部之后，我们也只会遇到 2-结点或者 3-结点，所以我们可以插入新的键。要用红黑树实现这个算法，我们需要：

将 4-结点表示为由三个 2-结点组成的一颗平衡的子树，根结点和两个子结点都用红链接相连；

在向下的过程中分解所有 4-结点并进行颜色转换；

和插入操作一样，在向上的过程中用旋转将 4-结点配平。（因为 4-结点可以存在，所以可以允许一个结点同时链接两条红链接）。

令人惊讶的是，你只需要移动上面算法的 put()方法中的一行代码就能实现 2-3-4 树中的插入操作：将 colorFlip()语句（及其 if 语句）移动到递归调用之前（null 测试和比较操作之间）。在多个进程可以同时访问同一棵树的应用中这个算法优于 2-3 树。

2.删除最小键

在第二轮热身中我们要学习 2-3 树中删除最小键的操作。我们注意到从树底部的 3-结点中删除键是很简单的，但 2-结点则不然。从 2-结点中删除一个键会留下一个空结点，一般我们会将它替换为一个空链接，但这样会破坏树的完美平衡。所以我们需要这样做：为了保证我们不会删除一个 2-结点，我们沿着左链接向下进行变换，确保当前结点不是 2-结点（可能是 3-结点，也可能是临时的 4-结点）。首先根结点可能有两种情况。如果根是 2-结点且它的两个子结点都是 2-结点，我们可以直接将这三个结点变为一个 4-结点；否则我们需要保证根结点的左子结点不是 2-结点，如有必要可以从它右侧的兄弟结点“借”一个键来。

在沿着左链接向下的过程中，保证以下情况之一成立：

如果当前结点的左子结点不是 2-结点，完成；

如果当前结点的左子结点是 2-结点而它的亲兄弟结点不是 2-结点，将左子结点的兄弟结点中的一个键移动到左子结点中；

如果当前结点的左子结点和它的亲兄弟结点都是 2-结点，将左子结点，父结点中的最小键和左子结点最近的兄弟结点合并为一个 4-结点，使父结点由 3-结点变为 2-结点或由 4-结点变为 3-结点。

3.删除操作

在查找路径上进行和删除最小键相同的变换同样可以保证在查找过程中任意当前结点均不是 2-结点。如果被查找的键在树的底部，我们可以直接删除它。如果不在，我们需要将它和它的后继结点交换，就和二叉树一样。因为当前结点必然不是 2-结点，问题已经转化为在一颗根结点不是 2-结点子树中删除最小键，我们可以在这个子树中使用前问所述的算法。和以前一样，删除之后我们需要向上回溯并分解余下的 4-结点。

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。

```
while((i<MAX_LEN-1)&&(size>1))  // 根据size的大小找到对应的分离空闲链表
{
    size >>= 1;
    i++;
}
/* 找到分离空闲链表，在该链中寻找对应的插入位置，并且保持链中块由小到大分布 */
search_bp = Lists[i];
while ((search_bp != NULL) && (size > GET_SIZE(HDRP(search_bp))))
{
    insert_bp = search_bp;
    search_bp = PRED(search_bp);
}
```

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。

```
// 根据size的大小找到对应的分离空闲链表
while ((i < MAX_LEN - 1) && (size > 1))
{
    size >>= 1;
    i++;
}
/* 四种可能性 */
if (PRED(bp) != NULL)
{
```

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。

#end　以上就是对堆整体设计的全部分析

3.2 关键函数设计（40 分）

3.2.1 int mm_init(void)函数（5 分）

```
/*初始化内存分配器*/
int mm_init(void)
{
    int i;
    /* 初始化分离空闲链表 */
    for (i = 0; i < MAX_LEN; i++)
    {
        Lists[i] = NULL;
    }
    if ((heap_listp = mem_sbrk(4*WSIZE)) == NULL)
        return -1;
    PUT(heap_listp, 0);   //对齐填充
    PUT(heap_listp + (1 * WSIZE), PACK(DSIZE, 1));  //序言块
    PUT(heap_listp + (2 * WSIZE), PACK(DSIZE, 1));
    PUT(heap_listp + (3 * WSIZE), PACK(0, 1));  //结尾块

    /* Extend the empty heap with a free block of INITCHUNKSIZE bytes */
    if (extend_heap(INITCHUNKSIZE) == NULL)
    return -1;
    return 0;
}
```

函数功能：初始化内存系统模型（包括初始化分离空闲链表和初始化堆）

处理流程：

因为采用隐式空闲链表来组织堆，因此 init 函数比较简单。

```
/*初始化内存分配器*/
int mm_init(void)
{
    int i;
    /* 初始化分离空闲链表 */
    for (i = 0; i < MAX_LEN; i++)
    {
        Lists[i] = NULL;
    }
    if ((heap_listp = mem_sbrk(4*WSIZE)) == NULL)
        return -1;
    PUT(heap_listp, 0);    //对齐填充
    PUT(heap_listp + (1 * WSIZE), PACK(DSIZE, 1));  //序言块
    PUT(heap_listp + (2 * WSIZE), PACK(DSIZE, 1));
    PUT(heap_listp + (3 * WSIZE), PACK(0, 1));  //结尾块

    /* Extend the empty heap with a free block of INITCHUNKSIZE bytes */
    if (extend_heap(INITCHUNKSIZE) == NULL)
    return -1;
    return 0;
}
```

第一步：初始化分离空闲链表。根据申请的链表数组，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。其中创建一个空的空闲链表分为以下 3 步：

1.第一个字是一个双字边界对齐不使用的填充字。

2.填充后面紧跟着一个特殊的序言块，，这是一个 8 字节的已分配块，只有一个头部和一个脚部组成，创建的时候使用 PUT(heap_listp + (1 * WSIZE), PACK(DSIZE, 1)); PUT(heap_listp + (2 * WSIZE), PACK(DSIZE, 1)); 两个语句将头部和脚部指向的字中，填充大小并且标记为已分配。

3.堆的结尾以一个特殊的结尾块来结束，使用 PUT(heap_listp + (3 * WSIZE), PACK(0, 1)); 语句来表示这个块是一个大小为零的已分配块。

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）

```
/*mm_free - Free a block*/
void mm_free(void *bp)
{
    size_t size = GET_SIZE(HDRP(bp));

    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
    InsertNode(bp, size);
    coalesce(bp);
}
```

函数功能：释放一个块

参　　数：指向请求块首字的指针 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：调用上文介绍的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。关于 coalesce 函数在后面会有分析。

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：最初的步骤和 mm_malloc 函数一样。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：

```
if (size <= DSIZE)
    size = 2 * DSIZE;
else
    size = ALIGN(size + DSIZE);  //内存对齐

/* 如果size小于原来块的大小，直接返回原来的块 */
if ((remaining = GET_SIZE(HDRP(bp)) - size) >= 0)
    return bp;

/* 否则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 */
else if (!GET_ALLOC(HDRP(NEXT_BLKP(bp))) || !GET_SIZE(HDRP(NEXT_BLKP(bp))))
{
    /* 如果加上后面连续地址上的未分配块空间也不够，那么需要扩展块 */
    if ((remaining =GET_SIZE(HDRP(bp))+GET_SIZE(HDRP(NEXT_BLKP(bp)))-size)<0)
    {
        if (extend_heap(MAX(-remaining, CHUNKSIZE)) == NULL)
            return NULL;
        remaining +=MAX(-remaining,CHUNKSIZE);
    }

    /* 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 */
    DeleteNode(NEXT_BLKP(bp));
    PUT(HDRP(bp), PACK(size + remaining, 1));
    PUT(FTRP(bp), PACK(size + remaining, 1));
}
/* 如果没有可以利用的连续未分配块，只能申请新的不连续的未分配块 */
else
{
    new_p = mm_malloc(size);
    memcpy(new_p, bp, GET_SIZE(HDRP(bp)));
    mm_free(bp);
}
```

1.如果 size 小于原来块的大小，直接返回原来的块。
2.否则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：和 mm_malloc 函数开头一致，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）

（备注：此分析只是根据 mm_implicit.c 中提供的 checkheap 函数进行分析，其中 checkheap 函数的原型是 **void mm_checkheap(int verbose)**）

函数功能：检查堆的一致性

参数：在 mdriver.c 中定义的全局详细信息输出 verbose

```
/*检查堆的一致性*/
void mm_checkheap(int verbose)
{
    char *bp = heap_listp;

    if (verbose)
        printf("Heap (%p):\n", heap_listp);

    if ((GET_SIZE(HDRP(heap_listp)) != DSIZE) || !GET_ALLOC(HDRP(heap_listp)))
        printf("Bad prologue header\n");
    checkblock(heap_listp);

    for (bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)) {
        if (verbose)
            printblock(bp);
        checkblock(bp);
    }

    if (verbose)
        printblock(bp);
    if ((GET_SIZE(HDRP(bp)) != 0) || !(GET_ALLOC(HDRP(bp))))
        printf("Bad epilogue header\n");
}
```

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。

```
static void checkblock(void *bp)
{
    if ((size_t)bp % 8)
        printf("Error: %p is not doubleword aligned\n", bp);
    if (GET(HDRP(bp)) != GET(FTRP(bp)))
        printf("Error: header does not match footer\n");
}
```

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，

```
static void printblock(void *bp)
{
    size_t hsize, halloc, fsize, falloc;

    hsize = GET_SIZE(HDRP(bp));
    halloc = GET_ALLOC(HDRP(bp));
    fsize = GET_SIZE(FTRP(bp));
    falloc = GET_ALLOC(FTRP(bp));

    if (hsize == 0) {
        printf("%p: EOL\n", bp);
        return;
    }

    printf("%p: header: [%d:%c] footer: [%d:%c]\n", bp,
           hsize, (halloc ? 'a' : 'f'),
           fsize, (falloc ? 'a' : 'f'));
}
```

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。

```
while (i < MAX_LEN)
{
    /* 先找合适的空闲链表 */
    if (((asize <= 1) && (Lists[i] != NULL)))
    {
        bp = Lists[i];
        /* 找到链表，在该链寻找大小合适的未分配块 */
        while ((bp != NULL) && ((size > GET_SIZE(HDRP(bp)))))
            bp = PRED(bp);

        /* 找到对应的未分配的块 */
        if (bp != NULL)
            break;
    }
    asize >>= 1;
    i++;
}
```

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：

1.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。

2.下面简单分析 place 函数如下：

size_t remaining = csize - asize;　用变量 remaining 表示分配空间之后剩余的大小

如果剩余的大小小于最小块，则不分离原块。否则将原块分离。但是此时存在一个问题，如果我们每次分配块，把大小间隔着分配，那么会可能会出现大块全部被释放，小块仍然被分配，导致很多外部碎片，如果下次来一个更大块的请求，我们还要重新去 free 块的问题。因此我们可以根据分配块的大小将小块放在连续的地方，大块放在连续的地方。函数具体实现如下：

```
/*将大小字节的块放在空闲块bp的开始处，并且如果余数至少是最小块大小则拆分*/
static void *place(void *bp, size_t asize)
{
    size_t csize = GET_SIZE(HDRP(bp));
    size_t remaining = csize - asize; /* allocate size大小的空间后剩余的大小 */

    DeleteNode(bp);

    /* 如果剩余的大小小于最小块，则不分离原块 */
    if (remaining < DSIZE * 2)
    {
        PUT(HDRP(bp), PACK(csize, 1));
        PUT(FTRP(bp), PACK(csize, 1));
    }

    else if (asize >= 96)
    {
        PUT(HDRP(bp), PACK(remaining, 0));
        PUT(FTRP(bp), PACK(remaining, 0));
        PUT(HDRP(NEXT_BLKP(bp)), PACK(asize, 1));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(asize, 1));
        InsertNode(bp, remaining);
        return NEXT_BLKP(bp);
    }

    else
    {
        PUT(HDRP(bp), PACK(asize, 1));
        PUT(FTRP(bp), PACK(asize, 1));
        PUT(HDRP(NEXT_BLKP(bp)), PACK(remaining, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(remaining, 0));
        InsertNode(NEXT_BLKP(bp), remaining);
    }
    return bp;
}
```

注：以上对 asize>=96 的选择是根据测试得出的，取 96 为最佳的情况。

以下是我分别取 128 和 64 时测试结果：

```
 8       yes    51%    24000  0.002063 11631
 9       yes    99%    14401  0.000476 30254
10       yes    98%    14401  0.000363 39716
Total            93%   112372  0.010609 10592

Perf index = 56 (util) + 40 (thru) = 96/100
```

```
 7       yes    61%    12000  0.000837 14345
 8       yes    88%    24000  0.006415  3741
 9       yes    99%    14401  0.000339 42443
10       yes    98%    14401  0.000375 38444
Total            93%   112372  0.012918  8699

Perf index = 56 (util) + 40 (thru) = 96/100
```

分别会在 trace8 和 trace7 出现不佳的测试结果。

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

```
/*合并块*/
static void *coalesce(void *bp)
{
    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));
    /*四种情况*/
    if (prev_alloc && next_alloc)   /*case1*/
    {
        return bp;
    }

    else if (prev_alloc && !next_alloc)   /*case2*/
    {
```

函数功能：边界标记合并。将指针返回到合并块。

参　　数：指向请求块首字的指针 bp

处理流程：coalesce 函数中的代码是对书上介绍的四种情况的一种简单直接的实现。因为我们选择的空闲链表格式（它的序言块和结尾块总是标记为已分配）允许我们忽略潜在的麻烦边界情况。下面介绍主要的步骤：

第一步：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));

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。具体操作为 size += GET_SIZE(HDRP(NEXT_BLKP(bp)));

PUT(HDRP(bp), PACK(size, 0));

PUT(FTRP(bp), PACK(size, 0));

3.前面块空闲，后面块已分配。这时和上面类似，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。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);　　此时 bp 指向前面块

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。

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);　　此时 bp 之前前面块

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

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

总体有个不错的测试结果，已经根据 trace 文件将内存率和吞吐率几乎最大化。因为其中利用了很多减少碎片的方式，比如在 place 函数中根据分配块的大小，将小块放在连续的地方，大块放在连续的地方，减少了很多外部碎片。并且大小类链表实现了块由小到大的变化，这样即使是首次适配，也能几乎达到最佳适配一样的效果。

外部碎片是当内存合计起来足够满足一个分配请求，但是没有一个单独的空闲块足够大可以来处理这个请求时发生的，因此任何分配器都可能有外部碎片，少量的外部碎片几乎是无法避免的。

4.3 自测试结果

```
qwj@qwj-virtual-machine:~/hitics/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   97%    5694  0.001222  4658
 1       yes   99%    5848  0.000630  9277
 2       yes   99%    6648  0.000546 12171
 3       yes   99%    5380  0.000379 14195
 4       yes   99%   14400  0.000670 21496
 5       yes   94%    4800  0.000570  8427
 6       yes   91%    4800  0.000786  6108
 7       yes   95%   12000  0.000788 15225
 8       yes   88%   24000  0.006236  3848
 9       yes   99%   14401  0.000420 34296
10       yes   98%   14401  0.000280 51395
Total          96%  112372  0.012528  8970

Perf index = 58 (util) + 40 (thru) = 98/100
```

第 5 章 总结

5.1 请总结本次实验的收获

1.最大的收获是实现了一个简单的分配器，并且有着不错的内存利用率和吞吐率。
2.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
3.学会了针对 trace 测试文件优化自己的分配器

5.2 请给出对本次实验内容的建议

因为能力有限，很难用红黑树对此分配器的设计进行优化，期待能多给出红黑树优化的提示信息。

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.