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 计算学部

学　　号 1190200708

班　　级 1903008

学 生 姓 名 熊峰

指 导 教 师 吴锐

实 验 地 点 G709

实 验 日 期 2021.06.07

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

X86-64 CPU 3.60GHz; 16G RAM; 256G SSD; 1T SSD

1.2.2 软件环境

Win 10
Ubuntu 20.04.2 LTS
WSL2

1.2.3 开发工具

Visual Studio 2019; Vim; GCC; GDB; Code::Blocks; CLion 2020.3.1 x64; EDB

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。

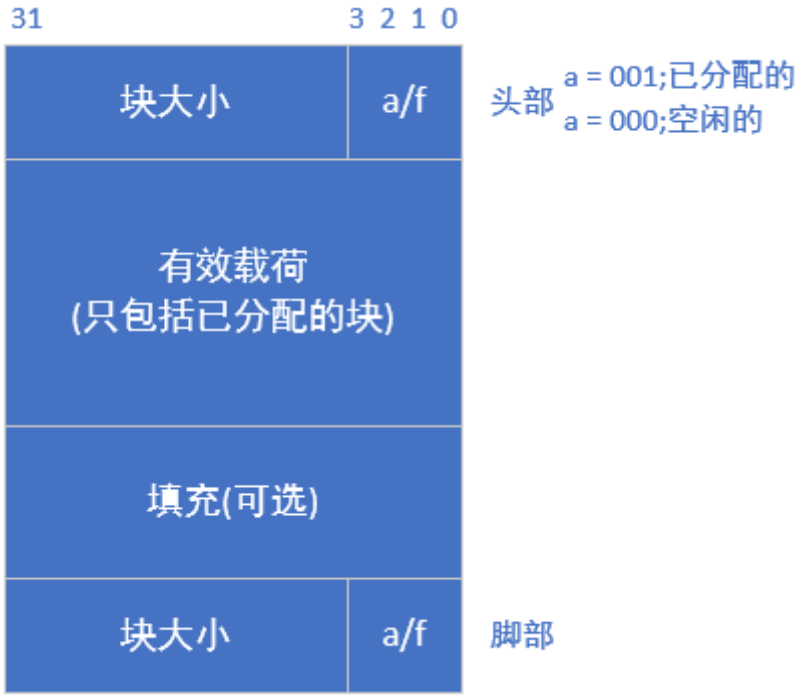
分配器将堆视为一组不同大小的块的集合来维护。每个块就是一个连续的虚拟内存片，要么是已分配的，要么是空闲的。已分配的块显式地保留为供应用程序使用。空闲块可用来分配。空闲块保持空闲，直到它显式地被应用所分配。一个已分配的块保持已分配状态，直到它被释放，这种释放要么是应用程序显式执行的，要么是内存分配器自身隐式执行的。

分配器有两种基本风格：显式分配器，隐式分配器。两种分隔偶要求应用显示地分配块。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，注入 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）

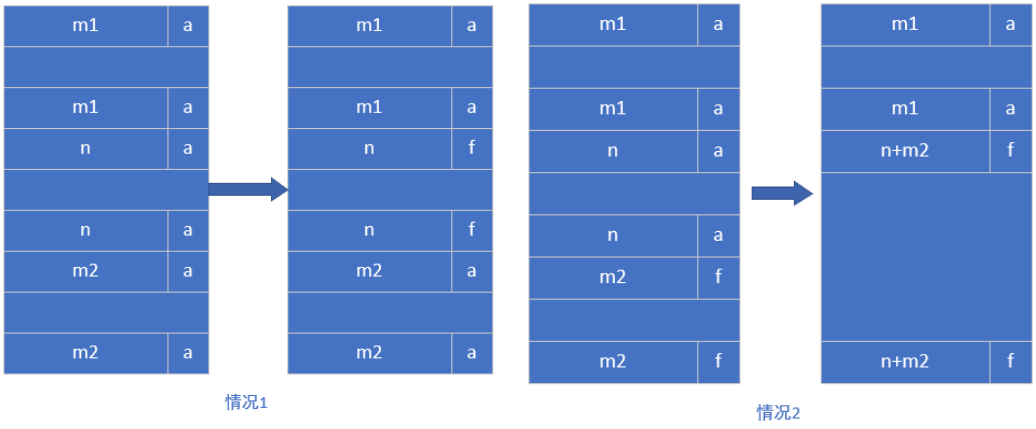


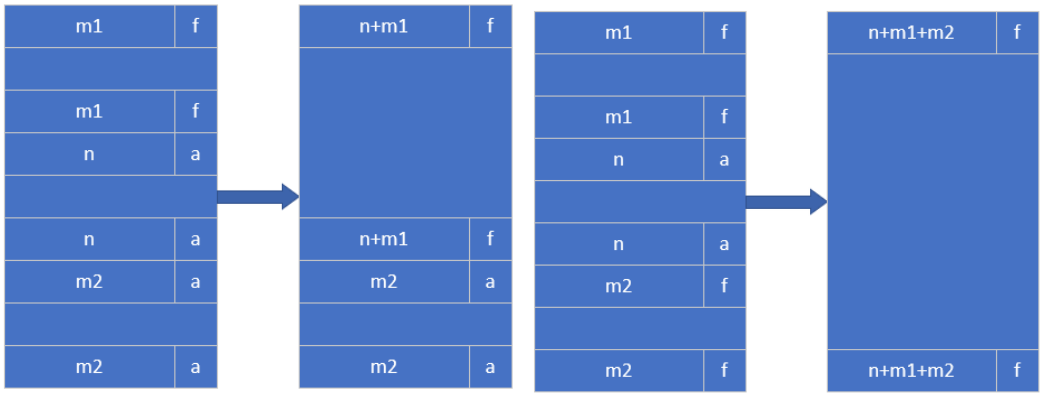
如图，在每个块的结尾处添加一个脚部，其中脚部就是头部的一个副本，如果每个块包括这样一个脚部，那么分配器就可以通过检查他的脚部，判断前面一个块的起始位置和状态，这个脚部总是在距当前块开始位置一个字的距离。

分配器释放当前块时所有可能存在的情况：

1) 前面的块和后面的块都是已分配的。
2) 前面的块是已分配的，后面的块是空闲的。
3) 前面的块是空闲的，而后面的块是已分配的。
4) 前面的和后面的块都是空闲的。

下图分别展示了如何对这四种情况合并。





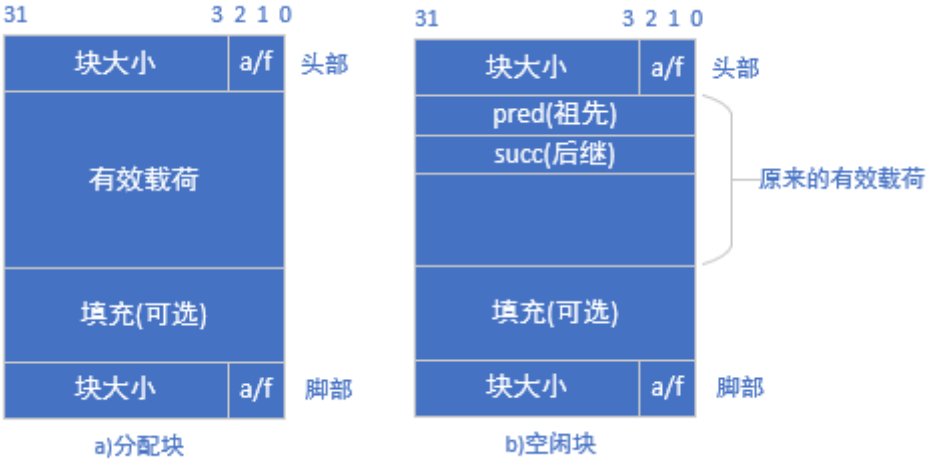
情况3　　情况4

在情况 1 中，两个临接的块都是已分配的，因此不可能进行合并。所以当前块的状态只是简单地从已分配变成空闲。在情况 2 中，当前块与后面的块合并。用当前块和后面块的大小的和来更新当前块的头部和后面块的脚部。在情况 3 中，前面的块和当前块合并。用两个块大小的和来更新前面块的头部和当前块的脚部。在情况 4 中，要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。在每种情况种，合并都是在常数时间内完成的。

边界标记的概念是简单优雅的，它对许多不同类型的分配器和空闲链表组织都是通用的。然而，他也存在一个潜在的缺陷。他要求每个块都保持一个头部和一个脚部，在应用程序操作许多个小块时，会产生显著的内存开销。例如，如果一个图形应用通过反复调试 malloc 和 free 来动态地创建和销毁图形节点，并且每个图形节点都只要求两个内存字，那么头部和脚部将占用每个已分配块的一半的空间。

2.3 显式空间链表的基本原理（5 分）

显式空闲链表将空闲块组织为某种形式的显示数据结构。因为根据定义，程序不需要一个空闲块的主体，所以实现这个数据结构的指针可以存放在这些空闲块的主体里面。例如，堆可以组织成一个双向空闲链表，在每个空闲块中，都包含一个 pred(前驱)和 succ(后继)指针，如图所示。



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，这取决于我们所选择的空闲链表中块的排序策略。

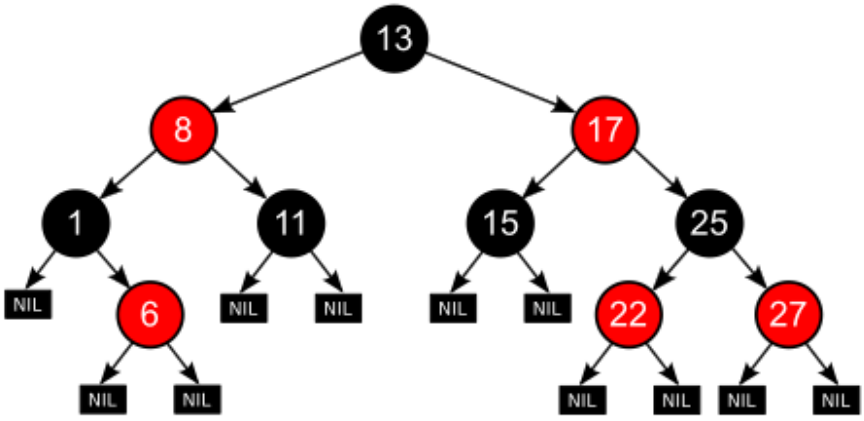
2.4 红黑树的结构、查找、更新算法（5 分）

红黑树的结构特点：

1. 每个节点要么是红色，要么是黑色。
2. 所有 NIL 叶子都被认为是黑色的。
3. 根节点是黑色。
4. 如果一个节点是红色的，那么它的两个子节点都是黑色的。
5. 从给定节点到其任何后代 NIL 叶的每条路径都经过相同数量的黑色节点。

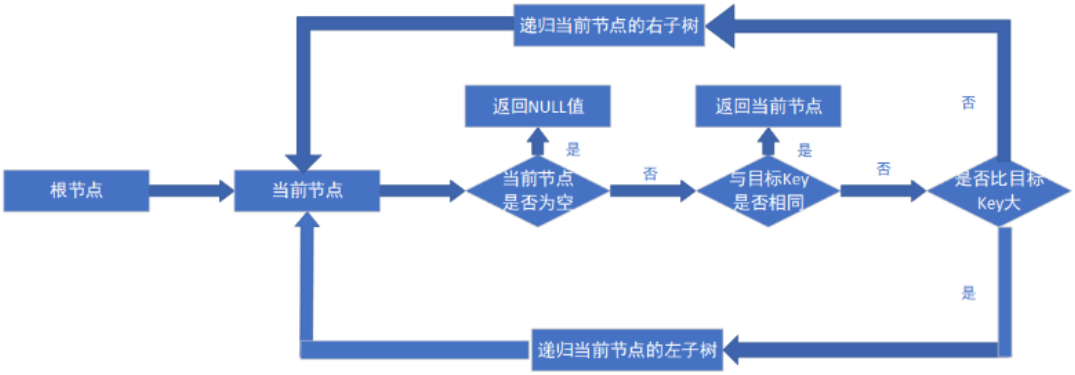
红黑树的优点：红黑树和 AVL 树一样都对插入时间、删除时间和查找时间提供了最好可能的最坏情况担保。

下图为一个红黑树的示例：



查找算法：

1. 从根结点开始查找，把根结点设置为当前结点。
2. 如果当前结点为空时，则以 null 作为返回。
3. 如果当前结点不为空，则将当前结点的 key 同要查找的 key 作比较。
4. 如果当前结点的 key 等于查找的 key，那么该 key 就是查找的目标，则返回前结点的值。
5. 如果当前结点的 key 大于查找的 key，那么就把当前结点的左子节点设置为当前结点，重复步骤 2。
6. 如果当前结点的 key 小于查找的 key，那么就把当前结点的右子结点设置为当前结点，重复步骤 2。



删除算法：

如果需要删除的节点有两个儿子，那么问题可以被转化成删除另一个只有一个儿子的节点的问题。对于二叉查找树，在删除带有两个非叶子儿子的节点的时候，我们要么找到它左子树中的最大元素、要么找到它右子树中的最小元素，并

把它的值转移到要删除的节点中。我们接着删除我们从中复制出值的那个节点，它必定有少于两个非叶子的儿子。因为只是复制了一个值，不违反任何性质，这就把问题简化为如何删除最多有一个儿子的节点的问题。它不关心这个节点是最初要删除的节点还是我们从中复制出值的那个节点。

我们只需要讨论删除只有一个儿子的节点（如果它两个儿子都为空，即均为叶子，我们任意将其中一个看作它的儿子）。如果我们删除一个红色节点（此时该节点的儿子将都为叶子节点），它的父亲和儿子一定是黑色的。所以我们可以简单的用它的黑色儿子替换它。通过被删除节点的所有路径只是少了一个红色节点。另一种简单情况是在被删除节点是黑色而它的儿子是红色的时候。如果只是去除这个黑色节点，用它的红色儿子顶替上来的话，会破坏本身的性质，但是如果我们重绘它的儿子为黑色，则曾经通过它的所有路径将通过它的黑色儿子，这样可以继续保持性质。

需要进一步讨论的是在要删除的节点和它的儿子二者都是黑色的时候，这是一种复杂的情况。我们首先把要删除的节点替换为它的儿子。出于方便，称呼这个儿子为 N（在新的位置上），称呼它的兄弟（它父亲的另一个儿子）为 S。在下面的示意图中，我们还是使用 P 称呼 N 的父亲，SL 称呼 S 的左儿子，SR 称呼 S 的右儿子。我们将使用下述函数找到兄弟节点：

```
struct node *sibling(struct node *n)
{
    if(n == n->parent->left)
        return n->parent->right;
    else
        return n->parent->left;
}
```

我们可以使用下列代码进行上述的概要步骤，这里的函数 replace_node 替换 child 到 n 在树中的位置。出于方便，在本章节中的代码将假定空叶子被用不是 NULL 的实际节点对象来表示。

```
void delete_one_child(struct node *n)
{
    // Precondition: n has at most one non-null child.
    struct node *child = is_leaf(n->right)? n->left : n->right;
    replace_node(n, child);
    if(n->color == BLACK){
        if(child->color == RED)
```

```
                child->color = BLACK;
            else
                delete_case1 (child);
        }
        free (n);
    }
```

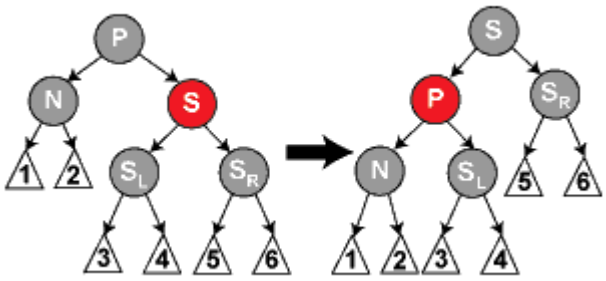
如果 N 和它初始的父亲是黑色，则删除它的父亲导致通过 N 的路径都比不通过它的路径少了一个黑色节点，树需要被重新平衡。有几种情形需要考虑：

1. N 是新的根。在这种情形下，我们就做完了。我们从所有路径去除了一个黑色节点，而新根是黑色的，所以性质都保持着。

```
    void delete_case1(struct node *n)
    {
        if(n->parent != NULL)
            delete_case2 (n);
    }
```

在情形 2、5 和 6 下，我们假定 N 是它父亲的左儿子。如果它是右儿子，则在这些情形下的左和右应当对调。

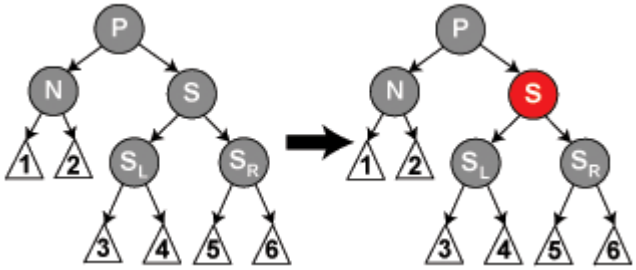
2. S 是红色。在这种情形下我们在 N 的父亲上做左旋转，把红色兄弟转换成 N 的祖父，我们接着对调 N 的父亲和祖父的颜色。完成这两个操作后，尽管所有路径上黑色节点的数目没有改变，但现在 N 有了一个黑色的兄弟和一个红色的父亲（它的新兄弟是黑色因为它是红色 S 的一个儿子），所以我们可以接下去按情形 4、情形 5 或情形 6 来处理。




```
    void delete_case2(struct node *n)
    {
        struct node *s = sibling (n);
        if(s->color == RED){
            n->parent->color = RED;
            s->color = BLACK;
```

```
            if(n == n->parent->left)
                rotate_left(n->parent);
            else
                rotate_right(n->parent);
        }
        delete_case3 (n);
}
```

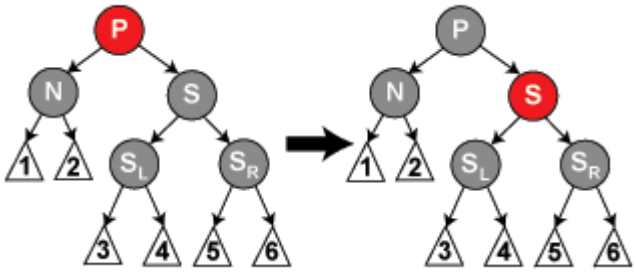
3. N 的父亲、S 和 S 的儿子都是黑色的。在这种情形下，我们简单的重绘 S 为红色。结果是通过 S 的所有路径，它们就是以前不通过 N 的那些路径，都少了一个黑色节点。因为删除 N 的初始的父亲使通过 N 的所有路径少了一个黑色节点，这使事情都平衡了起来。但是，通过 P 的所有路径现在比不通过 P 的路径少了一个黑色节点，所以仍然违反性质。要修正这个问题，我们要从情形 1 开始，在 P 上做重新平衡处理。




```
void delete_case3(struct node *n)
{
        struct node *s = sibling (n);
        if((n->parent->color == BLACK)&&
                (s->color == BLACK)&&
                (s->left->color == BLACK)&&
                (s->right->color == BLACK)) {
                    s->color = RED;
                    delete_case1(n->parent);
        }
        else
            delete_case4 (n);
}
```

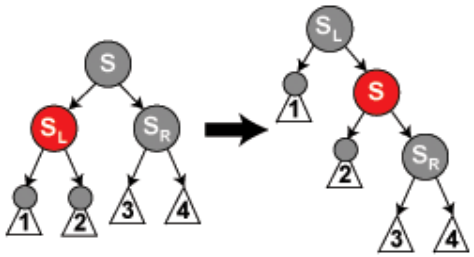
4. S 和 S 的儿子都是黑色，但是 N 的父亲是红色。在这种情形下，我们简单

的交换 N 的兄弟和父亲的颜色。这不影响不通过 N 的路径的黑色节点的数目，但是它在通过 N 的路径上对黑色节点数目增加了一，添补了在这些路径上删除的黑色节点。




```
void delete_case4(struct node *n)
{
    struct node *s = sibling (n);
    if（(n->parent->color == RED)&&
        (s->color == BLACK)&&
        (s->left->color == BLACK)&&
        (s->right->color == BLACK)) {
        s->color = RED;
        n->parent->color = BLACK;
    } else
        delete_case5 (n);
}
```

5. S 是黑色，S 的左儿子是红色，S 的右儿子是黑色，而 N 是它父亲的左儿子。在这种情形下我们在 S 上做右旋转，这样 S 的左儿子成为 S 的父亲和 N 的新兄弟。我们接着交换 S 和它的新父亲的颜色。所有路径仍有同样数目的黑色节点，但是现在 N 有了一个黑色兄弟，他的右儿子是红色的，所以我们进入了情形 6。N 和它的父亲都不受这个变换的影响。




```
void delete_case5(struct node *n)
{
    struct node *s = sibling (n);
```

```
        if（s->color == BLACK){
        // this if statement is trivial,
        // due to Case 2(even though Case two changed the sibling to a sibling's child,
        // the sibling's child can't be red, since no red parent can have a red child).
        // the following statements just force the red to be on the left of the left of the
parent,
        // or right of the right, so case six will rotate correctly.
            if((n == n->parent->left)&&
                (s->right->color == BLACK)&&
                (s->left->color == RED)) {
                // this last test is trivial too due to cases 2-4.
                s->color = RED;
                s->left->color = BLACK;
                rotate_right (s);
            }
            else if((n == n->parent->right)&&
                    (s->left->color == BLACK)&&
                    (s->right->color == RED)) {
                    // this last test is trivial too due to cases 2-4.
                  s->color = RED;
                  s->right->color = BLACK;
                  rotate_left (s);
            }
        }
        delete_case6 (n);
    }
```

6. S 是黑色，S 的右儿子是红色，而 N 是它父亲的左儿子。在这种情形下我们在 N 的父亲上做左旋转，这样 S 成为 N 的父亲（P）和 S 的右儿子的父亲。我们接着交换 N 的父亲和 S 的颜色，并使 S 的右儿子为黑色。子树在它的根上的仍是同样的颜色，所以性质 3 没有被违反。但是，N 现在增加了一个黑色祖先：要么 N 的父亲变成黑色，要么它是黑色而 S 被增加为一个黑色祖父。所以，通过 N 的路径都增加了一个黑色节点。

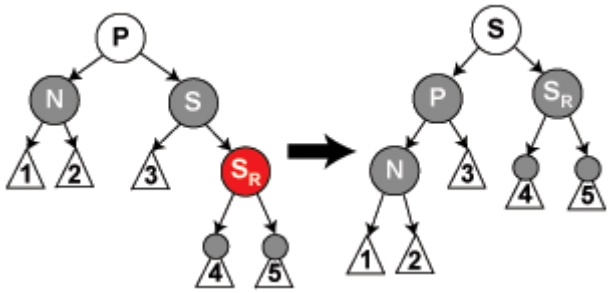
此时，如果一个路径不通过 N，则有两种可能性：

它通过 N 的新兄弟。那么它以前和现在都必定通过 S 和 N 的父亲，而它们只是交换了颜色。所以路径保持了同样数目的黑色节点。

它通过 N 的新叔父，S 的右儿子。那么它以前通过 S、S 的父亲和 S 的右儿子，

但是现在只通过 S，它被假定为它以前的父亲的颜色，和 S 的右儿子，它被从红色改变为黑色。合成效果是这个路径通过了同样数目的黑色节点。

在任何情况下，在这些路径上的黑色节点数目都没有改变。在示意图中的白色节点可以是红色或黑色，但是在变换前后都必须指定相同的颜色。




```
void delete_case6(struct node *n)
{
    struct node *s = sibling (n);
    s->color = n->parent->color;
    n->parent->color = BLACK;
    if(n == n->parent->left){
        s->right->color = BLACK;
        rotate_left(n->parent);
    }
    else {
        s->left->color = BLACK;
        rotate_right(n->parent);
    }
}
```

第 3 章 分配器的设计与实现

总分 50 分

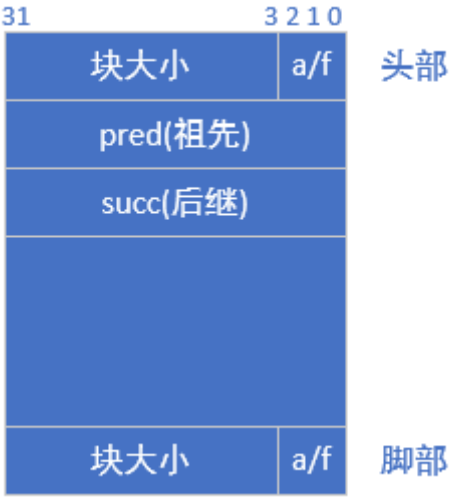
3.1 总体设计（10 分）

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。

组织结构如下：

堆的结构：如图所示，第一个字是一个双字边界对齐的不使用的填充字。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，只由一个头部和一个脚部组成。序言块是在初始时创建的，并且永不释放，在序言块后紧跟的是零个或者多个由 malloc 或者 free 调用创建的普通块，堆总是以一个特殊的结尾块来结束，这个结块是一个大小为零的已分配块，只由一个头部组成。

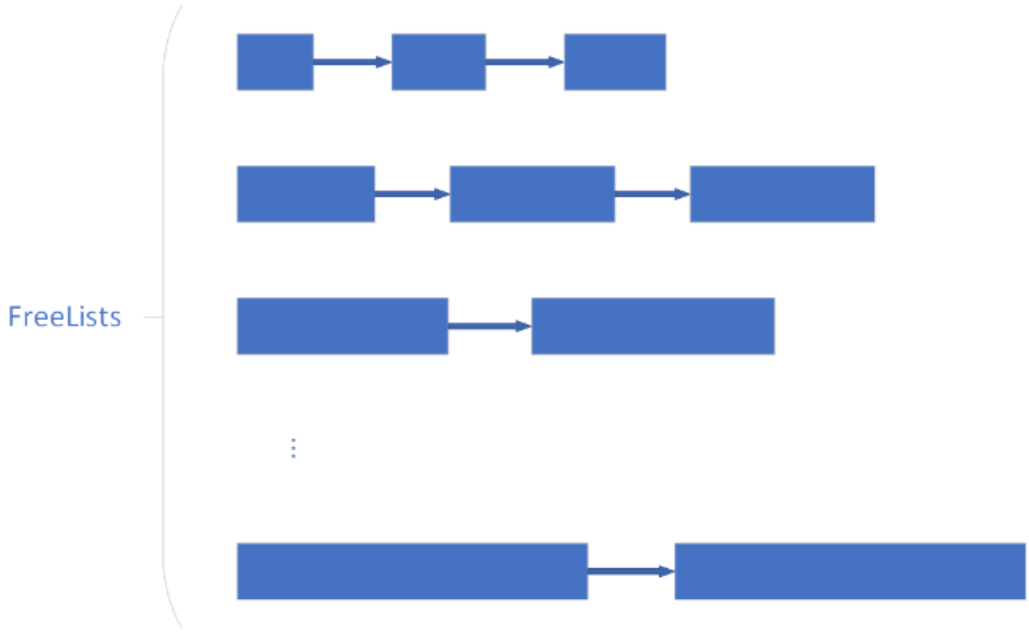
块的结构：如图所示，将块组织为显式数据结构，组织成一个双向空闲链表，在每个空闲块中，都包含一个 pred(前驱)和 succ(后继)指针。每个块的 pred 和 succ 指针指向的是空闲链表数组中实际上的前驱和后继块。使用双向链表而不是隐式空闲链表，使首次适配的分配时间从块总数的线性时间减少到空闲块数量的线性时间，同时采用分离适配的方法，使时间开销大大降低。



分配块链表：采用分离适配的方法，分配器维护着一个空闲链表的数组，每个空闲链表是和一个大小类相关联的，并且被组织成某种类型的显式或隐式链表。每个链表包含潜在的大小不同的块，并按照由小到大的顺序排列，方便执行向链表添加或删除元素的操作。

为了分配一个块，首先要确定请求的大小类，并对适当的空闲链表做首次适配，查找到一个合适的块，如果找到了则分割他，并将剩余部分插入到空闲链表中。否则就搜索下一个更大的大小类的空闲链表。直到找到一个合适的块，若不存在合适的块，则调用 extend_heap 函数，扩展堆顶，分配出一块新的内存，并分割，将剩余部分添加到空闲链表数组中，并执行合并操作。

空闲链表结构如下所示：



FreeLists 中每条链表表示不同的大小类，FreeLists[0]可以索引容量在[1,2]之间的内存块，FreeLists[1]可以索引容量在[3,4]之间的内存块，FreeLists[2]可以索引容量在[5,8]的内存块，以此类推。

要完成以上操作，还需以下宏定义：

```
// 满足对齐要求
#define ALIGNMENT 8
#define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~0x7)

/* single word (4) or double word (8) alignment */
// 定义字的大小，由于 32 位，字长 4 个字节
#define WSIZE 4
// 定义双字的大小，由于要求双字对齐，因此为 8 个字节
#define DSIZE 8

// 每次扩展堆的大小
```

```
#define CHUNKSIZE (1<<12)

// 定义分离表的大小
#define ListMax 16
void *FreeLists[ListMax];

// 求 x、y 之间的较大值和较小值
#define Max(x,y) ((x)>(y) ? (x):(y))
#define Min(x,y) ((x)<(t) ? (x):(y))

// 将 size 和 alloc 位合并
#define PACK(size,alloc) ((size)|(alloc))

// Get 取 P 处的值
#define GET(p)                  (*(unsigned int *)(p))
// PUT 将 val 写入 p 处
#define PUT(p, val)             (*(unsigned int *)(p) = (val))
// TODO:
// 将 val 写入到 p 处
#define SET_PTR(p, ptr) (*(unsigned int *)(p) = (unsigned int)(ptr))

// 获取 p 头部的块的大小
#define GET_SIZE(p)    (GET(p)& ~0x7)
// 获取 p 头部的分配位，判断是否已经分配
#define GET_ALLOC(p) (GET(p) & 0x1)

// 获取 p 块的头部指针
#define HDRP(bp) ((char *)(bp) - WSIZE)
// 获取 p 块的脚部指针
#define FTRP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)

// 获取 p 块的下一个块的指针
#define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE((char *)(bp) - WSIZE))
// 获取 p 块的上一个块的指针
#define PREV_BLKP(bp) ((char *)(bp) - GET_SIZE((char *)(bp) - DSIZE))

// 获取 p 块的祖先
#define PRED(bp) (*(char **)(bp))
// 获取 p 块的后继
#define SUCC(bp) (*(char **)(SUCC_PTR(bp)))
// 获取 bp 的祖先的块的指针
#define PRED_PTR(p) ((char *)(p))
// 读取 bp 的后继的块的指针
#define SUCC_PTR(p) ((char *)(p) + WSIZE)
#define SIZE_T_SIZE (ALIGN(sizeof(size_t)))
```

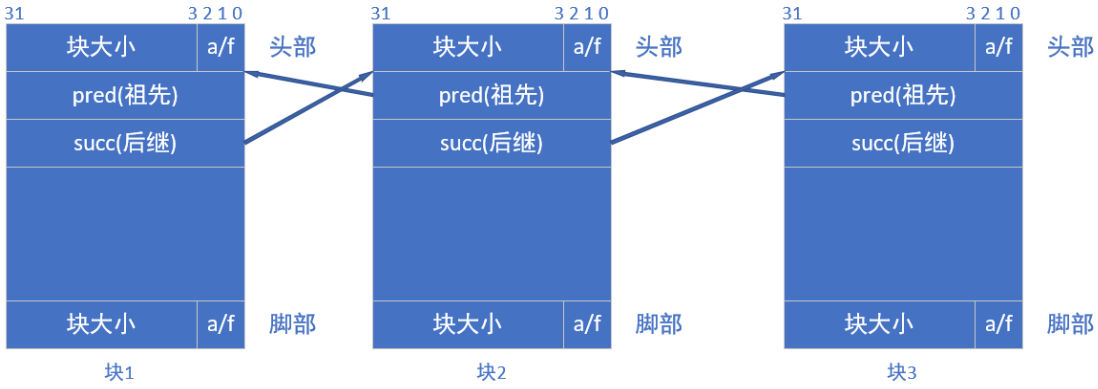
此处主要解释以下 PRED 和 PRED_PTR，SUCC 和 SUCC_PTR，其他定义注释已经较为详细，此处不再赘述。

PRED(bp)用来表示 bp 块的祖先，即将指针移动到 bp 块的祖先处，而 PRED_PTR(bp)则用来表示 bp 块中 pred 的部分的指针。

如图所示：

如图，块 2 的 pred 处保存的内容为指向块 1 的 pred 处的指针，当调用 PRED(bp)，若此时 bp 为块 2，此时返回值为指向块 1，而 PRED_PTR(bp)表示，块 2 中 pred 部分处的指针。SUCC 与 SUCC_PTR 同理。

因此，若我们需要执行 addBlock 或 deleteBlock 等操作时，我们需要利用这两个函数来实现。



除此之外，还需要以下辅助函数：

static void *extend_heap(size_t words);	扩展堆
static void *place(void *bp, size_t size);	将请求块放置在空闲块的起始位置 当剩余部分的空间大于 2*DSIZE 时，分离并保存到分离表中
static void addBlock(void *bp, size_t size);	将 bp 的空闲块添加到空闲表中
static void deleteBlock(void *bp);	将 bp 指向的已分配块从空闲表删除

下面为对辅助函数的说明：

1. extend_heap

函数功能：

扩展堆，通过请求大小向上舍入为最接近两字的倍数，保证扩展后的堆双字对齐。extend_heap 函数的使用存在以下两种情况：1.堆初始化时候 2.当 mm_malloc 函数不能找到一个合适的匹配块的时候。

函数具体实现：

判断 words 是否为单数，若为单数则分配(words+1)*WSIZE，否则分配 words*WSIZE 个字节。分配后，设置扩展的堆上的块的头部和脚部，并设置结尾块，再将此时的 bp 添加到空闲块中。由于原来的结尾块，此时由于已经被设置为新的块的头部，因此此处要重新设置结尾块。再将此时新分配的块进行合并操作。

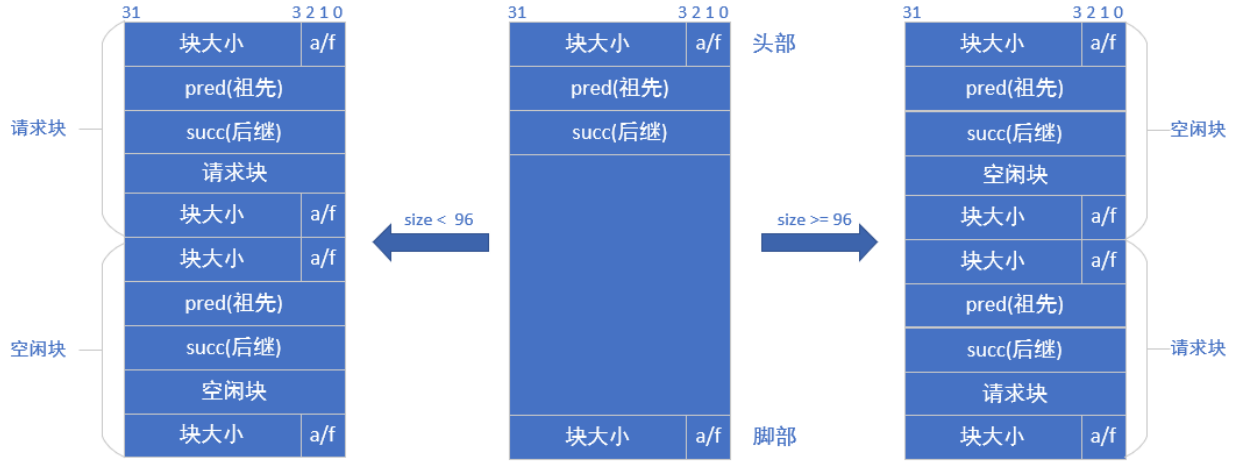
2. place

函数功能：

将请求块放置在空闲块的起始位置，当剩余部分的空间大于 2*DSIZE 时，分离并保存到分离表中，为了进一步优化，此处将返回值设置为 void *类型，通过此操作，可以修改 place 的修改策略，通过一定的选择策略，使其决定将其保存在 bp 的前一部分或保存在 bp 的后一部分。

函数具体实现：

首先获得 bp 块的 size，计算此时的空闲块，被填充后，剩下的块的大小。若剩下的块的大小小于单个的最小块，即四个字的大小的时候，此时不分离 bp 块。此处存在一处优化：通过尝试不同的 size，可得当 size>=96 时候，将前半段，即剩余的部分作为分离出的空闲块为最佳分割，否则，将后半段作为分割出的空闲块，效率更高。



3. addBlock

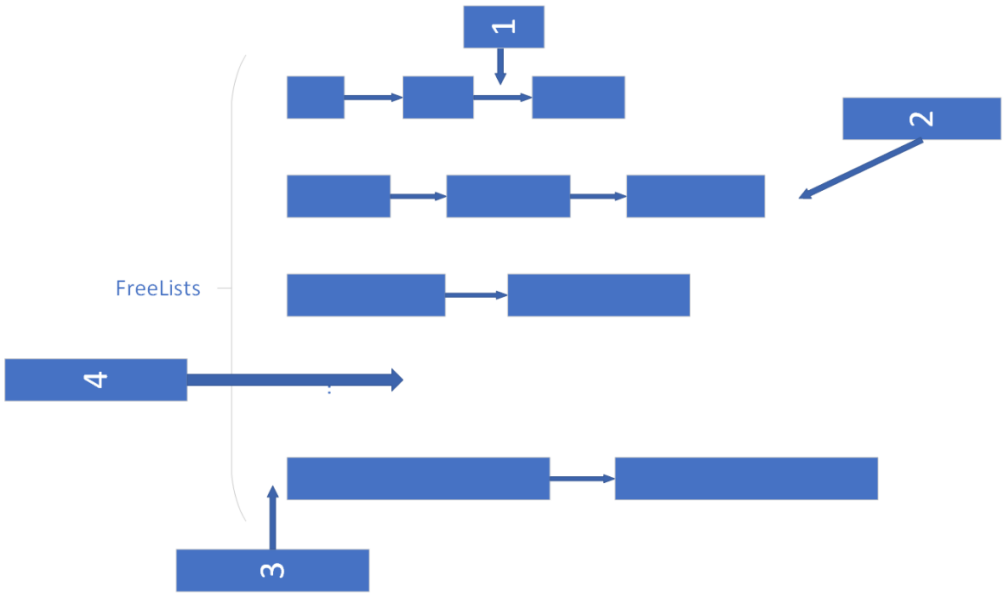
函数功能：

将 bp 空闲块添加到 FreeLists 中，并使其在每条链中保持从小到大的顺序。

函数具体实现：

首先通过 FreeLists 中的不同的大小类，区分不同的区域，找到 bp 所对应的小小类。按照从小到大的顺序，进行查询，若找到合适的块，即 GET_SIZE(FTRP(addNextptr)) >= size 的时候，由于空闲链表按照从小到大的顺序，因此此时只需要插在 addNextptr 前即可。分为以下四种情况：1.插入位置的祖先和后继块均不为空，此时需要将 bp 的祖先块的 succ 处的指针修改为 bp，bp 的 pred 处的指针修改为 bp 的祖先，再将 bp 的后继块中的 pred 处的指针修改为 bp，bp 的 succ 处的指针修改为 bp 的后继；2. 插入位置的祖先存在且后继不存在，此时只需将 bp 的祖先块的 succ 处的指针修改为 bp，bp 的 pred 处的指针修改为 bp 的祖先，将 bp 的 pred 处的指针修改为 null；3. 插入位置祖先不存在且后继存在，此时只需要将 bp 的后继块中的 pred 处的指针修改为 bp，bp 的 succ 处的指针修改为 bp 的后继，再将 FreeLists 中对应的大小类改为指向 bp4. 插入位置祖先和后继均不存在，此时只需要将 FreeLists 中对应的大小类指向 bp，且将 bp 的 succ 和 pred 处的指针都修改为 NULL 即可。

以下依次为四种情况的图示：



4. deleteBlock

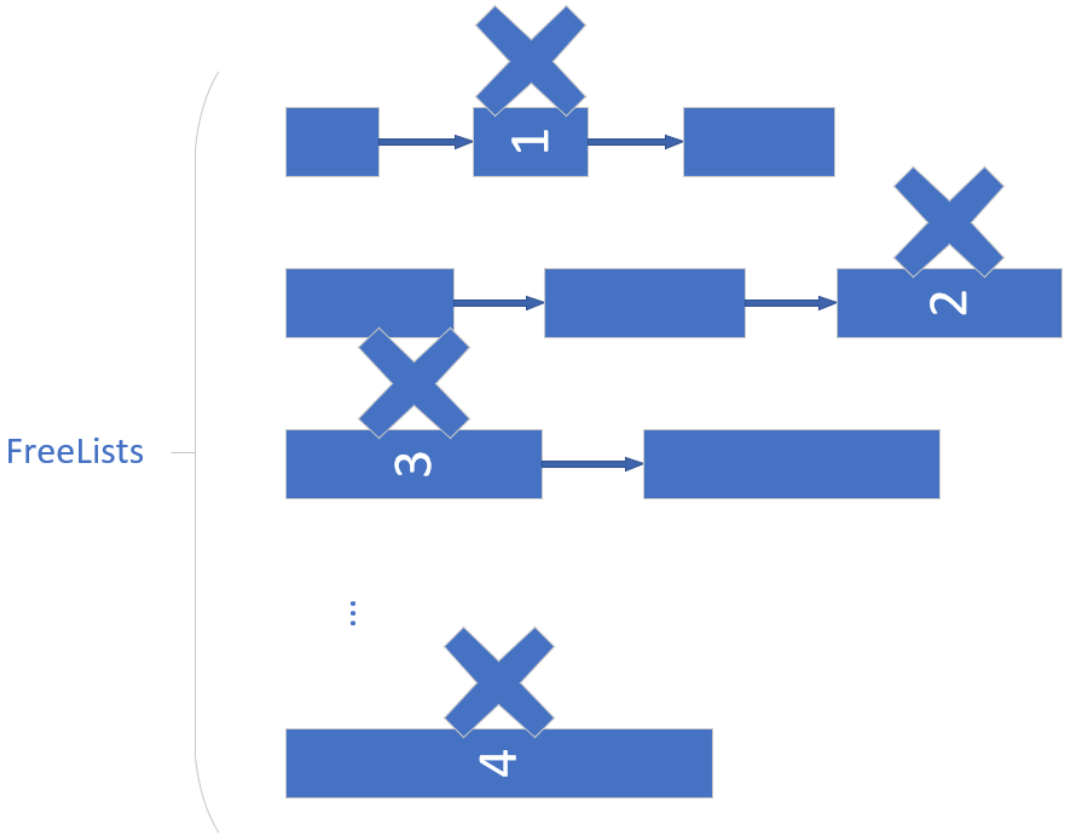
函数功能：

将 bp 空闲块，从 FreeLists 中删除。

函数具体实现：

通过 FreeLists 中的不同的大小类，找到其所对应的类，然后根据每条链的顺序，找到对应的 FreeLists 中所对应的空闲块。则此时也存在以下四种情况：1.待删除的块祖先和后继都存在，此时只需要将 bp 祖先块的 succ 处设置为 bp 的后继块，再将 bp 的后继块的 pred 处设置为 bp 的祖先块即可；2.待删除的块的祖先存在，后继不存在，此时只需将 bp 的祖先块的 succ 处设置为 NULL 即可；3.待删除的块的祖先不存在，后继存在，此时只需将 bp 的后几块的 pred 处设置为 NULL，并将 FreeLists 对应的大小类指向 bp 的后继块即可；4.若待删除块的祖先后继均不存在，此时说明，该链表只有 bp 一个元素，此时需要将 FreeLists 对应的大小类指向 NULL 即可。

以下依次为四种情况的图示：



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：

1.对空闲表 ListMax 初始化

2.分配堆上的内存，并对其分配出的四个字初始化，第一个字使初始化出的块对齐，第二个字作为序言块的头部，第三个字作为序言块的脚部，第四个字作为结尾块的头部，同时将以上的 alloc 位置都赋值为 1。

处理流程：

首先将空闲表初始化为 NULL，即利用分离适配，初始化一个空闲链表的数组，每个大小类包含潜在的大小不同的块，并按照从小到大的顺序排列，然后在从内存系统中得到 4 个字，并将他们初始化，创建一个新的链表，然后通过调用 extend_heap 函数，将堆扩展 CHUNKSIZE 字节，并创建初始的空闲块。

要点分析：

以上分配出的四个字的 alloc 位置都赋值为 1，可以方便在合并时候判断边界条件，避免在每次合并时候，都需要先判断是否为边界。

初始要分配出四个字，需要满足双字对齐，因此将第一个字赋值为 0 即可，只需要保证双字对齐。

FreeLists 空闲链表数组按照从小到大的顺序排列，并在数组中任一条链表中仍然按照从小到大的顺序排列。否则会影响之后的 addBlock、deleteBlock 等操作。

3.2.2 void mm_free(void *ptr)函数（5 分）

函数功能：

释放 ptr 处内存，将 ptr 位置处的空闲块，放入空闲表中，并将 ptr 处的前面的块，与后面的块合并。

参　　数：

ptr 为 malloc 函数申请的内存地址。

处理流程：

首先通过 GET_SIZE(HDRP(ptr))的宏定义，得到 ptr 对应的块的大小，并通过 PUT 的宏定义，将块的状态修改为空闲，并将该块保存到空闲表中，在对 ptr 对应的块进行前后的合并空闲块的操作。

要点分析：

ptr 处对应的块应该为由 mm_malloc 申请的内存。

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

函数功能：

1. 若 ptr 为空指针 NULL，则 realloc 等价于 mm_malloc(size)
2. 若参数 size 为 0，则等价于 mm_free(ptr)
3. 若 ptr 非空，则 ptr 应该为 mm_malloc 申请的内存，将 ptr 所指向内存块的大小变为 size，并返回新内存块的地址，若 ptr 所指向内存块后存在空闲块，则试图合并，并检查大小是否大于 size；或检查 ptr 所指向内存块后是否为结尾块，若为结尾块，则调用 extend_heap 函数，将堆扩充。否则，调用 mm_malloc 函数，重新为 ptr 分配大小为 size 的内存块，并调用 mmcpy 函数，保护原内存块的数据。

参　　数：

ptr 为 malloc 函数申请的内存地址，size 为扩充后内存块的大小。

处理流程：

1. 检查 ptr 是否为空，若 ptr 为空，则调用 mm_malloc 函数，为 ptr 分配大小为 size 的内存块，并返回。
2. 检查 size 是否为 0，若 size 为 0，则调用 mm_free 函数，释放 ptr 处的内存，并返回 NULL。
3. 计算 size 所对应的使其满足双字对齐的大小。
4. 判断 size 是否小于当前块的 size 的大小，若小于等于，则直接返回 ptr
5. 判断 ptr 的后继的块是否为空闲，并计算 ptr 的后继的块的大小，加上 ptr 的块的大小，是否大于 size，若大于 size，则将 ptr 和 ptr 后的块合并，调用 deleteBlock 函数，将 ptr 后的块，从空闲链表数组中删除，在将 ptr 的头部、脚部，用新计算出的两块的内存的大小的和重新写入，并返回 ptr。
6. 若 ptr 后继的块不为空闲，则判断后继块是否为结尾块，若为结尾块，则计算此时需要扩展的内存的大小，通过 Max 函数，我们可以很容易获得，通过 extend_heap 函数，将内存堆扩展，并将计算出的内存的大小和重新写入，并返回此时的 ptr。
7. 若非以上情况，则调用 mm_malloc 函数，获得一块内存大小为 size 的块，并将原来块的信息写入到新申请的块，并将原来的块释放，返回此时新申请的块。

要点分析：

1. ptr 若不为空，则 ptr 必须由 malloc 申请。

3.2.4 int mm_check(void)函数（5 分）

函数功能：

检查堆的一致性：

1. 检查堆的序言块是否为八字节已分配
2. 检查每块是否双字对齐，且每块的头部和脚部是否相同
3. 检查所有空闲块是否被标记为 0，且保存在空闲表中，非空闲块是否被标记为 1
4. 检查结尾是否为 PACK(0,1)

处理流程：

通过循环，遍历空闲表中所有元素，检查是否存在已分配的块，若此时存在已分配的块，则此时 GET_ALLOC(HDRP(FreeList))为 1，则返回 0，表明此时不满足一致性。再检查序言快的头部和脚部是否匹配，若不匹配则返回 0。再检查每个块是否为双字对齐，且头部和脚部是否相同，若不相同则返回 0。再将检查是否所有

空闲块都在空闲表中，若存在空闲块不在空闲表中，返回 0；最后检查结尾块是否为 PACK(0,1)。若不为 PACK(0,1)则返回 0，否则返回 1.

要点分析：

若堆是一致的，则返回 1，否则返回 0.

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 的字节的块，分配器调整块的大小，使其满足头部和脚部的空间，且保持双字对齐。

参　　数：

size 为待申请的空间的字节数。

处理流程：

1. 首先判断 size 是否为 0，若 size 为 0，则直接返回 NULL。
2. 计算可以使 size 满足双字对齐的 size 的大小。
3. 查找空闲表数组中，是否存在合适的空闲块，能够存放 size 大小的空闲块
4. 查找策略：通过循环遍历数组，通过对 size 的移位操作，判断是否满足属于对应于 FreeLists[i]的空闲链表。并对当前空闲链表的线性遍历，由于链表按照从小到大排列，因此只需让空闲块刚大于 size 时，所对应的空闲块，即为所需要的空闲块。并调用 place 函数分离空闲块，将多余部分保存在空闲表数组中。
5. 若不存在，则此时调用 extend_heap 函数，重新分配一块大小为 Max(size/WSIZE, CHUNKSIZE/WSIZE)的内存块，并将通过 place 函数将请求块防止在空闲块的起始位置，并通过 place 函数当剩余块大于 2*DSIZE 时，分离并保存到空闲表数组中。返回此时的 ptr。

要点分析：

若空闲块中没有查找到对应的空闲块可以满足其大小大于 size，则需要在堆上重新分配大小为使 size 满足双字对齐的大小的块。

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

函数功能：

合并 bp 处相邻的空闲块，查看 bp 的前面的块和后面的块的 alloc 位，查看其是否已经被分配，并根据分配情况，判断是否要合并。

处理流程：

通过两个布尔型变量，记录 bp 前面一个块和后面一个块的是否分配，分别根据以下四种情况，作出不同的合并情况。

1. 当前块的前一块为已分配块，后一块为已分配块，对于这种情况，只需要将 ptr 返回即可，此时不存在可合并的块。
2. 当前块的前一块为已分配块，后一块为空闲块，此时需要将 bp 和 bp 后面的块合并，删除空闲表数组中 bp 后面的块，通过计算两个块的 size，利用 PUT 重新在 bp 的头部和 bp 后面块的脚部写入 size、alloc 等信息，在将其保存到空闲表数组中，返回 bp。
3. 当前块的前一块为空闲块，后一块为已分配块，此时需要将 bp 和 bp 前面的块合并，删除空闲表数组中 bp 前一块，通过计算两个块合并后的大小，利用 PUT 重新在 bp 的脚部和 bp 前面块的头部写入，在重新保存到空闲表中，在将 bp 前一个块的地址返回。
4. 当前块的前一块为空闲块，后一块为空闲块，此时需要将 bp 与 bp 的前一块，和 bp 的后一块合并，此时需要将 bp 的前一块与 bp 的后一块，从空闲表数组中分别删除，重新计算新的块的大小，并通过 PUT 重新写入到 bp 前一个块的头部，和 bp 后一个块的脚部，在将 bp 前一个块的地址返回。

要点分析：

在情况 2 中，如果此时先修改 bp 的头部，则在修改脚部时，只需通过调用 FTRP 即可，因为此时 bp 的头部的 size 已经修改为合并后的 size 了，因此直接对 FTRP(bp) 赋 PACK(size,0)即可。

第 4 章测试

总分 10 分

4.1 测试方法与测试结果(3 分)

实验目标：能正确、高效、快速地运行

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 mm_malloc, mm_realloc 和 mm_free.

轨迹文件如下：:amptjp-bal.rep、cccp-bal.rep、cp-decl-bal.rep、expr-bal.rep、coalescing-bal.rep、random-bal.rep、random2-bal.rep、binary-bal.rep、binary2-bal.rep、realloc-bal.rep、realloc2-bal.rep.

性能分 pindex 是空间利用率和吞吐率的线性组合。

编译程序后，使用./mdriver -av -t traces/命令，测试所有轨迹文件，测试结果如下：

```
baileys@xf1190200708:/mnt/c/Users/Alienware/Desktop/share/lab8/malloclab-handout$ make
gcc -Wall -O2 -m32   -c -o mm.o mm.c
gcc -Wall -O2 -m32 -o mdriver mdriver.o mm.o memlib.o fsecs.o fcyc.o clock.o ftimer.o
baileys@xf1190200708:/mnt/c/Users/Alienware/Desktop/share/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   97%    5694  0.000247 23062
 1       yes   98%    5848  0.000345 16936
 2       yes   99%    6648  0.000385 17254
 3       yes   99%    5380  0.000314 17156
 4       yes   66%   14400  0.000682 21117
 5       yes   94%    4800  0.000322 14930
 6       yes   91%    4800  0.000317 15123
 7       yes   95%   12000  0.000456 26298
 8       yes   88%   24000  0.002737  8768
 9       yes   99%   14401  0.000245 58684
10       yes   86%   14401  0.000218 66120
Total          92%  112372  0.006269 17926

Perf index = 55 (util) + 40 (thru) = 95/100
```

4.2 测试结果分析与评价（3 分）

由于采用了显式链表、分离适配的方法，并进行大幅度的优化，在实际测试中，可以看到平均吞吐率较高，并在测试中获得满分，基本符合编写程序时的预期。

但是由于此时对堆的扩展，以及 realloc 合并操作，边界标记等未考虑最优解决方案，仍然有待提高空间利用率。

4.4 性能瓶颈与改进方法分析（4 分）

性能瓶颈：

此时的测试代码的平均吞吐率已经足够高，在测试中获得满分，目前未想到更加完善的优化方案。

此时的平均空间利用率仍然有待提高，在扩展堆的时候，应该根据实际需求，扩展堆，每次扩展 CHUNKSIZE/WSIZE 可能会导致一定程度上的使平均空间利用率下降。

除此之外，在 mm_realloc 函数中，可以考虑与前面的空闲的块合并，目前由于要保证其在与其前面块的空闲块合并后，仍要保持块内数据不变，暂未实现该功能。

由于实际采用的是显式空闲链表，因此此时每个块中需要包含空闲块的祖先和后继，因此会影响到实际的平均的空间利用率，可以通过边界标记的优化方法，使在已分配的块中不在需要脚部。

改进方法分析：

经过简要分析，仍然存在以下提升性能的方案：

扩展堆的时候，针对不同的情况，扩展的大小应该具体分配。

在 realloc 函数中，添加考虑与前面的空闲块合并的情况。

采用边界标记的优化方案。

第 5 章 总结

5.1 请总结本次实验的收获

1. 对 C 语言的指针有了更深刻的体会
2. 熟悉了虚拟存储的基本原理
3. 掌握并能够自己实现动态内存申请、释放的方法和相关函数
4. 掌握并能够自己实现动态内存申请的内部实现机制：分配算法、释放合并算法等
5. 掌握显式隐式两种不同方式实现的动态内存分配器
6. 了解了红黑树对实际应用的帮助

5.2 请给出对本次实验内容的建议

本次内容较为充实，收获丰富。

注：本章为酌情加分项。

参考文献

[1] RANDALE.BRYANT, DAVIDR.O‘HALLARON. 深入理解计算机系统[M]. 机械工业出版社, 2011.