

CSAPP : mallocclab 实验报告

姓名：施楚峰

学号：PB18000335

实验介绍

在这个实验中我们需要编写一个动态分配内存的函数，实现 `malloc` `free` `realloc` 等功能，并且做到空间利用率和效率之间的平衡。

实验准备

限制及条件

- `mm_init`: 在 driver 调用其它过程及函数前，会先调用 `mm_init` 进行初始化，例如分配初始空间
- `mm_malloc`: 函数返回一个负载至少为 `size` 的地址，并且分配的块应当在堆的限度内，并且不会与别的块重叠
- `mm_free`: 函数释放 `ptr` 处的空间，并且没有返回值
- `mm_realloc`: 函数返回一个至少 `size` 字节空间的地址，并且符合下述限制
 - `ptr` 为空等效于 `mm_malloc(size)`
 - `size` 为 0 等效于 `mm_free(ptr)`
 - `ptr` 不为空，并且是 `mm_malloc` 或 `mm_realloc` 返回的地址，则改变 `ptr` 指向的块的大小，并且返回新块的地址。应当对地址中存储的内容进行处理，使得新块中的内容与原先一致（变小截取，变大不用管）
- 不准改变 `mm.c` 中的接口
- 不允许使用任何有关内存管理的库函数或系统函数，因此程序中你不能使用 `malloc` `calloc` `free` `realloc` `sbrk` `brk` 等函数
- 你不能定义任何像数组、结构体、树、列表的全局或静态的复合数据结构，但是你可以定义全局标量变量，比如整形、浮点数和指针
- 为了与别的库保持一致，你的分配器应当返回八进制对齐的地址，driver 会强制要求你这么做

mm_check

用于 debug，检查堆的相容条件，我只是简单地把整个堆输出了一下，msg 由调用函数提供，输出调用函数信息及参数。

```
static void mm_check(char *msg)
{
    printf("%s\n", msg);
    for (char *bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKp(bp))
    {
        printf("%08x %d %d\n", HDRP(bp), GET_SIZE(HDRP(bp)),
GET_ALLOC(HDRP(bp)));
    }
}
```

实验过程

思路

为了获取更高的分数，达到高效率和高空间利用率，可以考虑使用显示空闲列表和分离适配的方法。而简单的分离适配需要遍历链表找到最佳适配复杂度为 $O(N)$ ，可以考虑利用平衡树中的 treap 来对链表进行维护，将复杂度下降为 $O(\log N)$ 。

隐式空闲链表的实现

代码基本和书本上一致，需要自己实现 realloc 功能，由于是在本地进行测试，所以更改了接口里的形参名。程序在没写 mm_check 之前一直没能通过测试，又不知道到底哪一块存在 bug，debug 了好久，最后发现 memcpy 函数中，两个 copy 的数组之间有重叠时，并不能达成复制的目的，最后自己写了一个 mm_memcpy 函数，将数值一个一个的进行复制。

下图是隐式空闲链表的测试结果

```
afool@ubuntu: ~/Documents/CS:APP_LAB/malloc
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: realloc2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.

Results for mm malloc:
trace  valid  util    ops    secs  Kops
0      yes   99%    5694   0.008512  669
1      yes   99%    5848   0.007762  753
2      yes   99%    6648   0.012343  539
3      yes  100%    5380   0.009189  585
4      yes   66%   14400   0.001734 8304
5      yes   92%    4800   0.008924  538
6      yes   92%    4800   0.008147  589
7      yes   55%   12000   0.149253   80
8      yes   51%   24000   0.297920   81
9      yes   44%   14401   0.095596  151
10     yes   45%   14401   0.016732  861
Total                77%  112372   0.616112  182

Perf index = 46 (util) + 12 (thru) = 58/100
afool@ubuntu:~/Documents/CS:APP_LAB/malloc$
```

对空间利用率较低的第 7、8、9、10 号数据观察，会发现可以对最后两个数据进行一个小优化，`place` 的策略可以改为将后一部分设为分配空间，不过意义好像并不大。

显式空闲链表+分离适配+平衡树

先考虑如何在数据块中存储所需要的信息，并计算最小块的大小。

由于有合并空闲块的操作，所以需要头部脚部信息，treap 需要维护 `rank` 值以及左右儿子，为了方便再把父节点信息存储下来，所以八进制对齐时最小块大小应为 24 字节共 6 个字。

历尽千辛万苦，总算将程序调出来之后，却尴尬的发现分数非但没有提升，反而下降了。

```
afool@ubuntu: ~/Documents/CS:APP_LAB/new
文件(E) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
Reading tracefile: binary2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: realloc-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: realloc2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.

Results for mm malloc:
trace  valid  util    ops      secs  Kops
0      yes   38%    5694  0.000889  6406
1      yes   28%    5848  0.001142  5123
2      yes   42%    6648  0.000781  8510
3      yes   72%    5380  0.000891  6036
4      yes   66%   14400  0.000475 30303
5      yes   96%    4800  0.001063  4516
6      yes   94%    4800  0.000873  5496
7      yes   55%   12000  0.000482 24896
8      yes   51%   24000  0.001010 23769
9      yes   40%   14401  0.529441    27
10     yes   45%   14401  0.014650   983
Total                57%  112372  0.551698   204

Perf index = 34 (util) + 14 (thru) = 48/100
afool@ubuntu:~/Documents/CS:APP_LAB/new$
```

空间利用率下降倒是可以理解，毕竟最小块的大小变成了 6 个字，但效率好像也没比原来高多少。仔细一看发现时间主要浪费在了第九个点上，而其它的点速度倒是比原本块不少，于是选择对第九个点进行针对性优化。观察数据发现程序大部分时间都花在了 realloc 上，而其中又反反复复地调用 malloc，根据对数据的观察，我们发现对于最后一个块可以直接进行延长，于是优化之后程序跑分就上去了。

```
afool@ubuntu: ~/Documents/CS:APP_LAB/new
文件(E) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
Reading tracefile: binary2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: realloc-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: realloc2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.

Results for mm malloc:
trace  valid  util    ops      secs  Kops
0      yes   99%    5694  0.000338 16821
1      yes   99%    5848  0.000494 11845
2      yes   99%    6648  0.002039  3260
3      yes   99%    5380  0.001184  4546
4      yes   66%   14400  0.000700 20586
5      yes   96%    4800  0.000792  6058
6      yes   95%    4800  0.000829  5793
7      yes   55%   12000  0.002163  5548
8      yes   51%   24000  0.003173  7564
9      yes   40%   14401  0.003961  3636
10     yes   45%   14401  0.000727 19820
Total                77%  112372  0.016399  6852

Perf index = 46 (util) + 40 (thru) = 86/100
afool@ubuntu:~/Documents/CS:APP_LAB/new$
```

不过貌似还没人家网上不用平衡树来的分高，可能是我空间使用的姿势不对吧。

代码分析

宏和书上基本保持一致。

```
#define DEBUG
#undef DEBUG

#define WSIZE          4
#define DSIZE          8
#define CHUNKSIZE      (1 << 12)

#define MAX(x, y)       ((x) > (y)? (x) : (y))
#define PACK(size, alloc) ((size) | (alloc))
#define GET(p)          (*(unsigned int *) (p))
#define PUT(p, val)     (*(unsigned int *) (p) = (unsigned int) (val))
#define GET_SIZE(p)     (GET(p) & ~0x7)
#define GET_ALLOC(p)    (GET(p) & 0x1)
#define HDRP(bp)        ((char *) (bp) - WSIZE)
#define FTRP(bp)        ((char *) (bp) + GET_SIZE((HDRP(bp))) - DSIZE)
#define SIZE(bp)        ( GET_SIZE(HDRP(bp)))
#define ALLOC(bp)       (GET_ALLOC(HDRP(bp)))
#define NEXT_BLKP(bp)   ((char *) (bp) + GET_SIZE(((char *) (bp) - WSIZE
E)))
#define PREV_BLKP(bp)   ((char *) (bp) - GET_SIZE(((char *) (bp) - DSIZ
E)))
```

为了方便平衡树操作，用了一个结构体，一开始的时候直接用了指针变量，但是可以只用 int 来存储地址，这样可以省下不少空间。

```
typedef struct Node
{
    unsigned int rank;
    int par;
    int lch;
    int rch;
}node;
```

平衡树的相关实现，由于 c 并不支持引用，所以很多函数用了返回值做到引用的效果。

```
// 随机数生成
static unsigned int mrand()
{
    static unsigned int seed = 12345;
    seed = seed * 482711UL % 4294967295UL;
    return seed;
}
// 获取节点的size
```

```

static unsigned int get_size(node *rt)
{
    return SIZE((void *)rt);
}

static void new_node(node *bp)
{
    bp->rank = mrand();
    bp->par = bp->lch = bp->rch = 0;
}

static void clear_node(node *rt)
{
    rt->rank = 0;
    rt->par = rt->lch = rt->rch = 0;
}

// rt的左子树或右子树必定存在

// 左旋
static int lturn(int _rt)
{
    node *rt = (node *)_rt;
    int _rchd = rt->rch;
    node *rchd = (node *)_rchd;

    rt->rch = rchd->lch;
    rchd->lch = _rt;

    if (rt->rch != 0)
        ((node *)(rt->rch))->par = _rt;
    rchd->par = rt->par;
    rt->par = _rchd;
    if (rchd->par != 0)
    {
        node *par = (node *)(rchd->par);
        if (par->lch == _rt)
            par->lch = _rchd;
        else if (par->rch == _rt)
            par->rch = _rchd;
    }

    return _rt = _rchd;
}

// 右旋
static int rturn(int _rt)
{
    node *rt = (node *)_rt;
    int _lchd = rt->lch;
    node *lchd = (node *)_lchd;

```



```

    rt->lch = lchd->rch;
    lchd->rch = _rt;

    if (rt->lch != 0)
        ((node *)(rt->lch))->par = _rt;
    lchd->par = rt->par;
    rt->par = _lchd;
    if (lchd->par != 0)
    {
        node *par = (node *)(lchd->par);
        if (par->lch == _rt)
            par->lch = _lchd;
        else if (par->rch == _rt)
            par->rch = _lchd;
    }

    return _rt = _lchd;
}
// 插入节点
static int insert_node(int _rt, int _chd)
{
    if (_rt == 0)
    {
        _rt = _chd;
        return _rt;
    }

    node *rt = (node *)_rt;
    node *chd = (node *)_chd;

    if (get_size(rt) >= get_size(chd))
    {
        rt->lch = insert_node(rt->lch, _chd);
        node *suc = (node *)(rt->lch);
        suc->par = _rt;
        if (rt->rank > suc->rank)
            _rt = rturn(_rt);
    }
    else {
        rt->rch = insert_node(rt->rch, _chd);
        node *suc = (node *)(rt->rch);
        suc->par = _rt;
        if (rt->rank > suc->rank)
            _rt = lturn(_rt);
    }

    return _rt;
}
// 在以_rt为根的树中找到大小为size的节点，没有返回 0
static int find_node(int _rt, size_t size)
{

```

```

node *rt = (node *)_rt;
if (_rt == 0) return 0;
if (get_size(rt) == size)
    return _rt;

int res = 0;

if (get_size(rt) >= size)
{
    res = find_node(rt->lch, size);
    if (res == 0)
        res = _rt;
}
else res = find_node(rt->rch, size);

return res;
}
// 删除节点
static int delete_node(int _rt)
{
    node *rt = (node *)_rt;
    if (_rt == 0)
        return 0;

    int _lchd = rt->lch;
    int _rchd = rt->rch;
    node *lchd = (node *)(_lchd);
    node *rchd = (node *)(_rchd);

    if ((lchd == NULL) || (rchd == NULL))
    {
        if (lchd == NULL)
        {
            if (rchd != NULL)
                rchd->par = rt->par;
            _rt = _rchd;
        }
        else {
            lchd->par = rt->par;
            _rt = _lchd;
        }
    }
    else {
        if (lchd->rank < rchd->rank)
        {
            _rt = rturn(_rt);
            node *temp = (node *)_rt;
            temp->rch = delete_node(temp->rch);
        }
        else {
            _rt = lturn(_rt);
            node *temp = (node *)_rt;

```



```

        temp->lch = delete_node(temp->lch);
    }
}

return _rt;
}

```

关于 malloc 部分的实现。

```

// 找到大小为size的节点所属的类别
static void *get_root(size_t size)
{
    int i = 0, lim = 32;
    while ((size > lim) && i < 8)
    {
        i++;
        lim <<= 1;
    }
    return (void *)(heap_start + i * WSIZE);
}
// 插入节点
static void push_list(void *bp)
{
#ifdef DEBUG
    char ch[100];
    printf("\n====push_list(%d)====\n", SIZE(bp));
#endif

    if (bp == NULL)
        return;

    void *cat = get_root(SIZE(bp));
    int ptr = GET(cat);
    int chd = (int)(long long)bp;

    new_node((node *)chd);
    ptr = insert_node(ptr, chd);
    int root = ptr;
    PUT(cat, root);

    sprintf(ch, "-----push_list-----\n");
#ifdef DEBUG
    mm_check(ch);
#endif // DEBUG

    // original root's parent is NULL
}
// 删除节点

```

```

static void erase_list(void *bp)
{

#ifdef DEBUG
    char ch[100];
    printf("\n====erase_list(%d)====\n", SIZE(bp));
#endif

    if (bp == NULL)
        return;

    void *cat = get_root(SIZE(bp));
    int ptr = GET(cat);

    if (ptr == 0)
        return;

    int rt = (unsigned int)(long long)bp;
    int temp = rt;
    int tpar = ((node *)rt)->par;

    rt = delete_node(rt);
    clear_node(temp);

    if (ptr == temp)
    {
        PUT(cat, (unsigned int)(long long)rt);
    }
    else {
        if (tpar != 0)
        {
            node *par = (node *)tpar;
            if (par->lch == temp)
            {
                par->lch = rt;
                if (rt != 0)
                    ((node *)rt)->par = tpar;
            }
            else if (par->rch == temp)
            {
                par->rch = rt;
                if (rt != 0)
                    ((node *)rt)->par = tpar;
            }
        }
    }
}

```

```

#ifdef DEBUG
    sprintf(ch, "-----erase_list-----\n");
    mm_check(ch);
#endif // DEBUG

}
// 初始化
int mm_init(void)
{

#ifdef DEBUG
    char ch[100];
    printf("\n====mm_init====\n");
#endif

    if ((heap_start = mem_sbrk(12 * WSIZE)) == (void *)-1)
        return -1;

    // generate free list links, stores pointer to treap's root
    for (int i = 0; i < 9; ++i)
        PUT(heap_start + i * WSIZE, 0);
    /*
        i = 0 size <= 32
        ...
        i = 7 size <= 4096
        i = 8 size > 4096

    */

    PUT(heap_start + 9 * WSIZE, PACK(DSIZE, 1));
    PUT(heap_start + 10 * WSIZE, PACK(DSIZE, 1));
    PUT(heap_start + 11 * WSIZE, PACK(0, 1));

    heap_listp = heap_start + 10 * WSIZE;

#ifdef DEBUG
    sprintf(ch, "-----mm_init-----\n");
    mm_check(ch);
#endif

    if (extend_heap(CHUNKSIZE / WSIZE) == NULL)
        return -1;
    return 0;
}
// 拓展堆的大小
static void *extend_heap(size_t words)
{

#ifdef DEBUG
    char ch[100];

```

```

        printf("\n====extend_heap(%d)====\n", words);
#endif // DEBUG

    char *bp;
    size_t size;

    size = words + (words & 1);
    size = size * WSIZE;
    if ((long)(bp = mem_sbrk(size)) == -1)
        return NULL;

    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1));

    push_list(bp);

#ifdef DEBUG
    sprintf(ch, "-----extend_heap-----\n");
    mm_check(ch);
#endif // DEBUG

    return coalesce(bp);
}
// 合并块
static void *coalesce(void *bp)
{
#ifdef DEBUG
    char ch[100];
    printf("\n====coalesce(%d)====\n", SIZE(bp));
#endif

    size_t prev_alloc = ALLOC(PREV_BLKP(bp));
    size_t next_alloc = ALLOC(NEXT_BLKP(bp));
    size_t size = SIZE(bp);

    if (prev_alloc && next_alloc)
    {
        return bp;
    }
    else if (prev_alloc && !next_alloc)
    {
        erase_list(NEXT_BLKP(bp));
        erase_list(bp);
    }

```

```

        size += GET_SIZE(HDRP(NEXT_BLK(b)));
        PUT(HDRP(b), PACK(size, 0));
        PUT(FTRP(b), PACK(size, 0));

        push_list(b);
    }
    else if (!prev_alloc && next_alloc)
    {
        erase_list(PREV_BLK(b));
        erase_list(b);

        size += GET_SIZE(HDRP(PREV_BLK(b)));
        PUT(FTRP(b), PACK(size, 0));
        PUT(HDRP(PREV_BLK(b)), PACK(size, 0));
        b = PREV_BLK(b);

        push_list(b);
    }
    else {
        erase_list(PREV_BLK(b));
        erase_list(NEXT_BLK(b));
        erase_list(b);

        size += GET_SIZE(HDRP(PREV_BLK(b)))
            + GET_SIZE(HDRP(NEXT_BLK(b)));
        PUT(HDRP(PREV_BLK(b)), PACK(size, 0));
        PUT(FTRP(NEXT_BLK(b)), PACK(size, 0));
        b = PREV_BLK(b);

        push_list(b);
    }

#ifdef DEBUG
    sprintf(ch, "-----coalesce-----\n");
    mm_check(ch);
#endif // DEBUG

    return b;
}
// 合并块并标记为已占用
static void *coalesce1(void *b)
{
#ifdef DEBUG
    char ch[100];
    printf("\n=====coalesce(%d)===== \n", SIZE(b));
#endif

    size_t prev_alloc = ALLOC(PREV_BLK(b));
    size_t next_alloc = ALLOC(NEXT_BLK(b));

```

```

size_t size = SIZE(bp);

if (prev_alloc && next_alloc)
{
    return bp;
}
else if (prev_alloc && !next_alloc)
{
    erase_list(NEXT_BLKPTR(bp));

    size += GET_SIZE(HDRP(NEXT_BLKPTR(bp)));
    PUT(HDRP(bp), PACK(size, 1));
    PUT(FTRP(bp), PACK(size, 1));

}
else if (!prev_alloc && next_alloc)
{
    erase_list(PREV_BLKPTR(bp));

    size += GET_SIZE(HDRP(PREV_BLKPTR(bp)));
    PUT(FTRP(bp), PACK(size, 1));
    PUT(HDRP(PREV_BLKPTR(bp)), PACK(size, 1));
    bp = PREV_BLKPTR(bp);

}
else {
    erase_list(PREV_BLKPTR(bp));
    erase_list(NEXT_BLKPTR(bp));

    size += GET_SIZE(HDRP(PREV_BLKPTR(bp)))
        + GET_SIZE(HDRP(NEXT_BLKPTR(bp)));
    PUT(HDRP(PREV_BLKPTR(bp)), PACK(size, 1));
    PUT(FTRP(NEXT_BLKPTR(bp)), PACK(size, 1));
    bp = PREV_BLKPTR(bp);

}

#ifdef DEBUG
    sprintf(ch, "-----coalesce-----\n");
    mm_check(ch);
#endif // DEBUG

    return bp;
}
// 找可能存在未分配合适大小块的位置
static int find_place(int i, size_t asize)
{
    int cat = 0;
    int size = asize;
    int lim = 1 << (i+5);
    while (i < 8 && (size > lim || GET(heap_start + i * WSIZE) == 0))

```

```

    {
        i++;
        lim <= 1;
    }
    if (GET(heap_start + i * WSIZE) == 0) ++i;
    return i;
}
// 分割
static void place(int bp, size_t asize)
{
#ifdef DEBUG
    char ch[100];
    printf("\n====place(%d)====\n", asize);
#endif // DEBUG

    size_t csize = GET_SIZE(HDRP(bp));

    if ((csize - asize) >= (3 * DSIZE))
    {
        erase_list(bp);

        PUT(HDRP(bp), PACK(asize, 1));
        PUT(FTRP(bp), PACK(asize, 1));
        bp = NEXT_BLK(P(bp));
        PUT(HDRP(bp), PACK(csize - asize, 0));
        PUT(FTRP(bp), PACK(csize - asize, 0));

        push_list(bp);
    }
    else
    {
        erase_list(bp);

        PUT(HDRP(bp), PACK(csize, 1));
        PUT(FTRP(bp), PACK(csize, 1));
    }

#ifdef DEBUG
    sprintf(ch, "-----place-----\n");
    mm_check(ch);
#endif // DEBUG
}

void *mm_malloc(size_t size)
{
#ifdef DEBUG
    char ch[100];

```



```

        printf("\n====mm_malloc(%d)====\n", size);
#endif // DEBUG

    size_t asize;
    size_t extendsize;
    int bp=0;

    if (size == 0)
        return NULL;

    if (size <= 2 * DSIZE)
        asize = 3 * DSIZE;
    else
        asize = DSIZE * ((size + (DSIZE)+(DSIZE - 1)) / DSIZE);

    int i = 0;
    while ((i = find_place(i, asize)) < 9)
    {
        void *cat = (unsigned int)(long long)(heap_start + i * WSIZE);
        int res = find_node(GET(cat), asize);
        if (res != 0)
        {
            bp = res;
            break;
        }
        ++i;
    }

    if (bp != 0)
    {
        place(bp, asize);
        return (void *)bp;
    }

    extendsize = MAX(asize, CHUNKSIZE);
    if ((bp = (int)(long long)extend_heap(extendsize / WSIZE)) == 0)
        return NULL;
    place(bp, asize);

#ifdef DEBUG
    sprintf(ch, "-----mm_malloc-----\n");
    mm_check(ch);
#endif DEBUG

    return (void *)bp;
}

void mm_free(void *ptr)

```

```

{

#ifdef DEBUG
    char ch[100];
    printf("\n====free(%d)====\n", SIZE(ptr));
#endif // DEBUG


    size_t size = GET_SIZE(HDRP(ptr));

    PUT(HDRP(ptr), PACK(size, 0));
    PUT(FTRP(ptr), PACK(size, 0));

    push_list(ptr);

    coalesce(ptr);

#ifdef DEBUG
    sprintf(ch, "-----mm_free-----\n");
    mm_check(ch);
#endif // DEBUG
}

/*
 * mm_realloc - Implemented simply in terms of mm_malloc and mm_free
 */
void *mm_realloc(void *ptr, size_t size)
{
    size_t asize;

    if (ptr == NULL)
    {
        return mm_malloc(size);
    }

    if (size == 0)
    {
        mm_free(ptr);
        return ptr;
    }

    if (size <= 2 * DSIZE)
        asize = 3 * DSIZE;
    else
        asize = DSIZE * ((size + (DSIZE)+(DSIZE - 1)) / DSIZE);

    if (asize <= SIZE(ptr))
    {
        place(ptr, asize);
    }
}

```

```

        return ptr;
    }
    else
    {

        int csize = SIZE(ptr) - DSIZE;
        void *p1 = coalesce1(ptr);
        int psize = SIZE(p1);
        if (psize >= asize)
        {
            if (p1 != ptr)
                mm_memcpy(p1, ptr, csize);
            place(p1, asize);
            return p1;
        }
        else {
            if (NEXT_BLKPTR(ptr) == NULL)
            {
                void *res = extend_heap((asize - psize)/WSIZE);
                size += SIZE(res);
                PUT(HDRP(p1), PACK(size, 1));
                PUT(FTRP(p1), PACK(size, 1));
                if (p1 != ptr)
                    mm_memcpy(p1, ptr, csize);
            }
            void *res = mm_malloc(size);
            mm_memcpy(res, ptr, csize);
            mm_free(p1);
            return res;
        }

    }

    return NULL;
}

```

实验总结

这个实验的代码从三月一直撸到了五月，一方面程序总是推倒重来不是这里没想对，就是那里变量错了，期间还有个比较困扰的问题就是如何把指针类型转存为整型，不过好在也不难解决，这之中又穿插了期中考，能完成实验简直就是万幸了哈哈。这次实验虽然使用了平衡树的做法，但还是没有达到最高的分数，原因在于对于中间的数据，空间使用率并不高，可以针对特定的数据再进行优化，不过鉴于写代码加调程耗费了这么多的时间精力，剩下的也没啥心思写了。不过虽说花的时间很长，但也能够很好的巩固动态分配内存的知识，也是很值得的！

其它

本实验所有资源在 [CSAPP-Labs/malloclab at master · Afool1999/CSAPP-Labs · GitHub](#)