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 1190200128

班　　级 1903001

学 生 姓 名 詹先佑

指 导 教 师 郑贵滨

实 验 地 点 格物楼 213

实 验 日 期 2021 年 6 月 11 日

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

无

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 位总是零。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。我们将堆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。

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）

结构：

红黑树的英文是“Red-Black Tree”，简称 R-B Tree，它是一种不严格的平衡

二叉查找树

红黑树中的节点，一类被标记为黑色，一类被标记为红色。除此之外，一棵红黑树还需要满足这样几个要求：

根节点是黑色的；

每个叶子节点都是黑色的空节点（NIL），也就是说，叶子节点不存储数据（图中将黑色的、空的叶子节点都省略掉了）；

任何相邻的节点都不能同时为红色，也就是说，红色节点是被黑色节点隔开的；

每个节点，从该节点到达其可达叶子节点的所有路径，都包含相同数目的黑色节点。

查找算法：

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 - nodes 是另一个 2 - nodes 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。

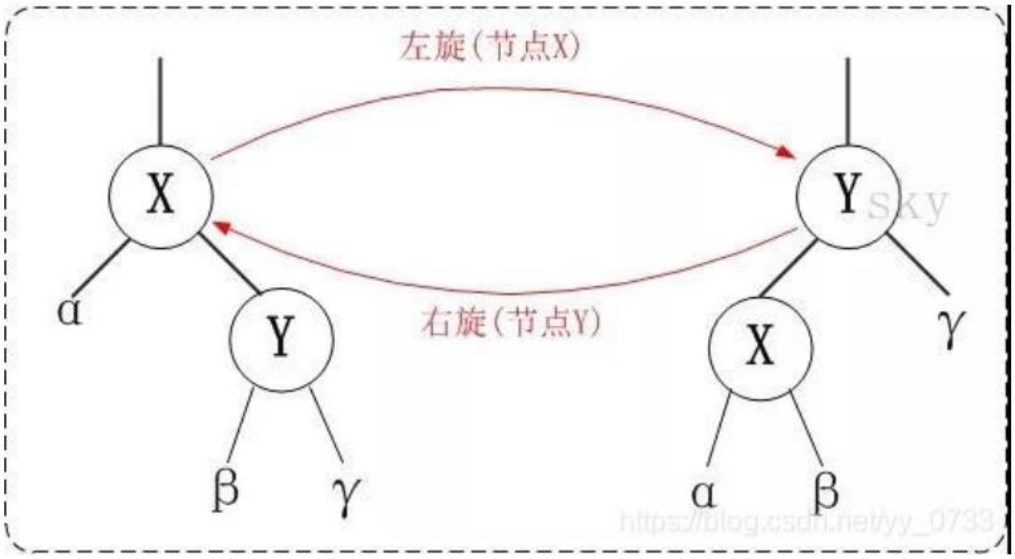
更新算法：

（1） 左旋和右旋：

左旋：对 x 进行左旋，意味着，将“x 的右孩子”设为“x 的父亲节点”；即，将 x 变成了一个左节点(x 成了为 z 的左孩子)！因此，左旋中的“左”，意味着“被旋转的节点将变成一个左节点”

右旋：对 x 进行右旋，意味着，将“x 的左孩子”设为“x 的父亲节点”；即，将 x 变成了一个右节点(x 成了为 y 的右孩子)！因此，右旋中的“右”，意味着“被旋转的节点将变成一个右节点”。

无论是左旋还是右旋，被旋转的树，在旋转前是二叉查找树，并且旋转之后仍然是一颗二叉查找树。



（2） 添加：

步骤：首先，将红黑树当作一颗二叉查找树，将节点插入；然后，将节点着色为红色；最后，通过旋转和重新着色等方法来修正该树，使之重新成为一颗红黑树。详细描述如下：

第一步: 将红黑树当作一颗二叉查找树，将节点插入。

红黑树本身就是一颗二叉查找树，将节点插入后，该树仍然是一颗二叉查找树。也就意味着，树的键值仍然是有序的。此外，无论是左旋还是右旋，若旋转之前这棵树是二叉查找树，旋转之后它一定还是二叉查找树。这也就意味着，任何的旋转和重新着色操作，都不会改变它仍然是一颗二叉查找树的事实。

好吧？那接下来，我们就来想方设法的旋转以及重新着色，使这颗树重新成为红黑树！

第二步：将插入的节点着色为"红色"。

为什么着色成红色，而不是黑色呢？为什么呢？在回答之前，我们需要重新温习一下红黑树的特性：

(1) 每个节点或者是黑色，或者是红色。

(2) 根节点是黑色。

(3) 每个叶子节点是黑色。[注意：这里叶子节点，是指为空的叶子节点！]

(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。

将插入的节点着色为红色，不会违背"特性(5)"！少违背一条特性，就意味着我们需要处理的情况越少。接下来，就要努力的让这棵树满足其它性质即可；满足了的话，它就又是一颗红黑树了。

第三步: 通过一系列的旋转或着色等操作，使之重新成为一颗红黑树。

第二步中，将插入节点着色为"红色"之后，不会违背"特性(5)"。那它到底会违背哪些特性呢？

对于"特性(1)"，显然不会违背了。因为我们已经将它涂成红色了。

对于"特性(2)"，显然也不会违背。在第一步中，我们是将红黑树当作二叉查找树，然后执行的插入操作。而根据二叉查找数的特点，插入操作不会改变根节点。所以，根节点仍然是黑色。

对于"特性(3)"，显然不会违背了。这里的叶子节点是指的空叶子节点，插入非空节点并不会对它们造成影响。

对于"特性(4)"，是有可能违背的！

那接下来，想办法使之"满足特性(4)"，就可以将树重新构造成红黑树了。

根据被插入节点的父节点的情况，可以将“当节点 z 被着色为红色节点，并插入二叉树”划分为三种情况来处理。

(1)情况说明：被插入的节点是根节点。

处理方法：直接把此节点涂为黑色。

(2)情况说明：被插入的节点的父节点是黑色。

处理方法：什么也不需要做。节点被插入后，仍然是红黑树。

(3)情况说明：被插入的节点的父节点是红色。

处理方法：那么，该情况与红黑树的“特性(5)”相冲突。这种情况下，被插入节点是一定存在非空祖父节点的；进一步的讲，被插入节点也一定存在叔叔节点(即使叔叔节点为空，我们也视之为存在，空节点本身就是黑色节点)。理解这点之后，我们依据"叔叔节点的情况"，将这种情况进一步划分为 3 种情况(Case):

Case 1

当前节点的父节点是红色，且当前节点的祖父节点的另一个子节点（叔叔节点）也是红色。

(01) 将“父节点”设为黑色。

(02) 将“叔叔节点”设为黑色。

(03) 将“祖父节点”设为“红色”。

(04) 将“祖父节点”设为“当前节点”(红色节点)；即，之后继续对“当前节点”进行操作。

Case 2

当前节点的父节点是红色，叔叔节点是黑色，且当前节点是其父节点的右孩子

(01) 将“父节点”作为“新的当前节点”。

(02) 以“新的当前节点”为支点进行左旋。

Case 3

当前节点的父节点是红色，叔叔节点是黑色，且当前节点是其父节点的左孩子

(01) 将“父节点”设为“黑色”。

(02) 将“祖父节点”设为“红色”。

(03) 以“祖父节点”为支点进行右旋。

上面三种情况(Case)处理问题的核心思路都是：将红色的节点移到根节点；然后，将根节点设为黑色。下面对它们详细进行介绍。

1.(Case 1)叔叔是红色

1.1 现象说明

当前节点(即，被插入节点)的父节点是红色，且当前节点的祖父节点的另一个子节点（叔叔节点）也是红色。

1.2 处理策略

(01) 将“父节点”设为黑色。

(02) 将“叔叔节点”设为黑色。

(03) 将“祖父节点”设为“红色”。

(04) 将“祖父节点”设为“当前节点”(红色节点)；即，之后继续对“当前节点”进行操作。

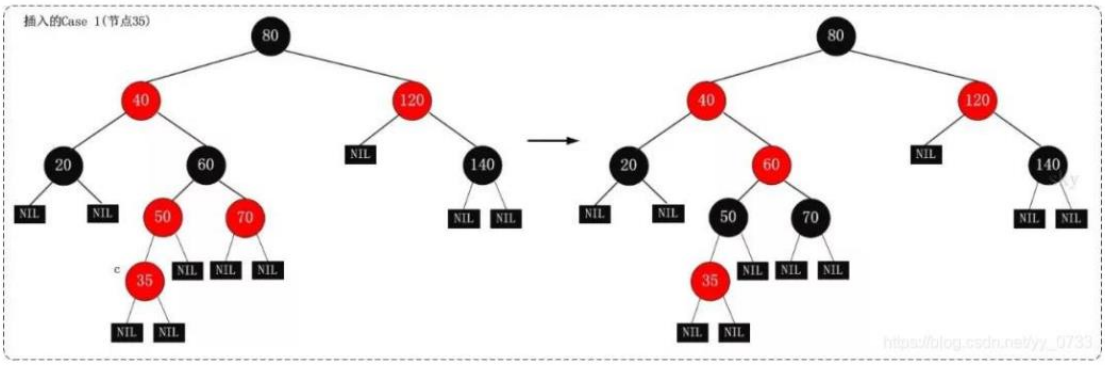
下面谈谈为什么要这样处理。(建议理解的时候，通过下面的图进行对比)

“当前节点”和“父节点”都是红色，违背“特性(4)”。所以，将“父节点”设置“黑色”以解决这个问题。

但是，将“父节点”由“红色”变成“黑色”之后，违背了“特性(5)”：因为，包含“父节点”的分支的黑色节点的总数增加了 1。 解决这个问题的办法是：将“祖父节点”由“黑色”变成红色，同时，将“叔叔节点”由“红

色”变成“黑色”。关于这里，说明几点：第一，为什么“祖父节点”之前是黑色？这个应该很容易想明白，因为在变换操作之前，该树是红黑树，“父节点”是红色，那么“祖父节点”一定是黑色。 第二，为什么将“祖父节点”由“黑色”变成红色，同时，将“叔叔节点”由“红色”变成“黑色”；能解决“包含‘父节点’的分支的黑色节点的总数增加了 1”的问题。这个道理也很简单。“包含‘父节点’的分支的黑色节点的总数增加了 1” 同时也意味着“包含‘祖父节点’的分支的黑色节点的总数增加了 1”，既然这样，我们通过将“祖父节点”由“黑色”变成“红色”以解决“包含‘祖父节点’的分支的黑色节点的总数增加了 1”的问题； 但是，这样处理之后又会引起另一个问题“包含‘叔叔’节点的分支的黑色节点的总数减少了 1”，现在我们已知“叔叔节点”是“红色”，将“叔叔节点”设为“黑色”就能解决这个问题。所以，将“祖父节点”由“黑色”变成红色，同时，将“叔叔节点”由“红色”变成“黑色”；就解决了该问题。

按照上面的步骤处理之后：当前节点、父节点、叔叔节点之间都不会违背红黑树特性，但祖父节点却不一定。若此时，祖父节点是根节点，直接将祖父节点设为“黑色”，那就完全解决这个问题了；若祖父节点不是根节点，那我们需要将“祖父节点”设为“新的当前节点”，接着对“新的当前节点”进行分析。



2.(Case 2)叔叔是黑色，且当前节点是右孩子

2.1 现象说明

当前节点(即，被插入节点)的父节点是红色，叔叔节点是黑色，且当前节点是其父节点的右孩子

2.2 处理策略

(01) 将“父节点”作为“新的当前节点”。

(02) 以“新的当前节点”为支点进行左旋。

下面谈谈为什么要这样处理。(建议理解的时候，通过下面的图进行对比)

首先，将“父节点”作为“新的当前节点”；接着，以“新的当前节点”为支点进行左旋。 为了便于理解，我们先说明第(02)步，再说明第(01)步；为了便于说明，我们设置“父节点”的代号为 F(Father)，“当前节点”的代号为 S(Son)。

为什么要“以 F 为支点进行左旋”呢？根据已知条件可知：S 是 F 的右孩子。而之前我们说过，我们处理红黑树的核心思想：将红色的节点移到根节点；然后，将根节点设为黑色。既然是“将红色的节点移到根节点”，那就是说要不断的将破坏红黑树特性的红色节点上移(即向根方向移动)。 而 S 又是一个右孩子，因此，我们可以通过“左旋”来将 S 上移！

按照上面的步骤(以 F 为支点进行左旋)处理之后：若 S 变成了根节点，那么直接将其设为“黑色”，就完全解决问题了；若 S 不是根节点，那我们需要执行步骤(01)，即“将 F 设为‘新的当前节点’”。那为什么不继续以 S 为新的当前节点继续处理，而需要以 F 为新的当前节点来进行处理呢？这是因为“左旋”之后，F 变成了 S 的“子节点”，即 S 变成了 F 的父节点；而我们处理问题的时候，需要从下至上(由叶到根)方向进行处理；也就是说，必须先解决“孩子”的问题，再解决“父亲”的问题；所以，我们执行步骤(01)：将“父节点”作为“新的当前节点”。

3.(Case 3)叔叔是黑色，且当前节点是左孩子

3.1 现象说明

当前节点(即，被插入节点)的父节点是红色，叔叔节点是黑色，且当前节点是其父节点的左孩子

3.2 处理策略

(01) 将“父节点”设为“黑色”。

(02) 将“祖父节点”设为“红色”。

(03) 以“祖父节点”为支点进行右旋。

下面谈谈为什么要这样处理。(建议理解的时候，通过下面的图进行对比)

为了便于说明，我们设置“当前节点”为 S(Original Son)，“兄弟节点”为 B(Brother)，“叔叔节点”为 U(Uncle)，“父节点”为 F(Father)，祖父节点为 G(Grand-Father)。

S 和 F 都是红色，违背了红黑树的“特性(4)”，我们可以将 F 由“红色”变为“黑色”，就解决了“违背‘特性(4)’”的问题；但却引起了其它问题：违背特性(5)，因为将 F 由红色改为黑色之后，所有经过 F 的分支的黑色节点的个数增加了 1。那我们如何解决“所有经过 F 的分支的黑色节点的个数增加了 1”的问题呢？ 我们可以通过“将 G 由黑色变成红色，同时“以 G 为支点进行右旋”来解

决。

（3） 删除：

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。

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：堆由开发人员分配和释放， 若开发人员不释放，程序结束时由 OS 回收，分配方式类似于链表。参考代码如下：

```
int main() {
  // C 中用 malloc() 函数申请
  char* p1 = (char *)malloc(10);
  cout<<(int*)p1<<endl;          //输出：00000000003BA0C0

  // 用 free() 函数释放
  free(p1);

  // C++ 中用 new 运算符申请
  char* p2 = new char[10];
  cout << (int*)p2 << endl;      //输出：00000000003BA0C0

  // 用 delete 运算符释放
  delete[] p2;
}
```

（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：

采用分离的空闲链表，使用立即边界标记合并方式。

（4）算法：

int mm_init(void)函数；

void *mm_malloc(size_t size)函数
void mm_free(void *ptr)函数；
void *mm_realloc(void *ptr, size_t size)函数；
int mm_checkheap (void)函数；
static void *extend_heap(size_t words)函数；
static void place(void *bp, size_t asize)函数；
static void *find_fit(size_t asize)函数
static void *coalesce(void *bp)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：初始化内存管理器
处理流程：
（1） 创建一个空闲的分离链表
（2） 通过设置相应块的 header 和 footer 来申请堆空间
（3） 调用 extend_heap（）函数拓展堆空间，扩大一个大小为 CHUNKSIZE 的块
要点分析：
（1） 创建空闲链表之后需要使用 extend_heap 来扩展堆。
（2） 需要保持对齐的要求

3.2.2 void mm_free(void *ptr)函数（5 分）

函数功能：释放一个内存块
参　　数：指向需要释放的内存块的名为 ptr 的指针
处理流程：
（1） 得到需要释放的块的大小
（2） 通过调用 PUT(p, val)将需要释放的块的头部和脚部设置为空闲状态
（3） 再通过调用 coalesce(bp)函数合并空闲的块
要点分析：
将内存块释放后，空闲的内存块可能可以与相邻的内存块进行合并，我们不能忽视这个情况

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

函数功能：将指针 ptr 所指向的内存块重新拓展一个大小为 size 的内存块

参　　数：指向需要拓展的内存块的名为 ptr 的指针、需要分配的内存大小 size
处理流程：
（1） 当申请一个大小为 size 的内存块失败时，会有报错信息，并且结束程序；
（2） 获得需要拓展的内存块的大小 copySize，如果 copySize 大于 size，那么就要把 size 的值赋给 copySize
（3） 在调用 memcpy(newp, ptr, copySize)函数对内存块进行复制，以达到拓展内存块的目的
（4） 最后释放原有的 ptr 指针所指向的内存块，返回指向新的拓展的内存块的指针 newp；
要点分析：
当需要拓展的内存块的大小 copySize 大于 size 时，需要将 size 的值赋给 copySize，不然会拓展失败

3.2.4 int mm_check(void)函数（5 分）

函数功能：查看堆的一致性
处理流程：
（1） 先定义指针 bp，初始化为指向序言块的全局变量 heap_listp。如果 verbose 不为 0，则打印堆的信息
（2） 如果堆的大小没有双字对齐，就会有报错信息
（3） 如果 bp 所指的内存块的头部和脚部不相等的话就会有报错信息
（4） 通过 for 循环遍历出所以正常的内存块，并且打印它们
（5） 检查最后一个内存块，并且打印它的信息，如果最后一个内存块没有被分配就会有报错信息
要点分析：
检查的条件应该考虑全面，包括大小是否对齐，脚部和头部是否匹配等等

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

函数功能：申请一个大小为 size 的内存块
参　　数：内存大小 size
处理流程：
（1） 首先确定该内存申请是否合法，如果不合法，就会报错
（2） 然后调整内存块的大小，以保证开销和对齐的要求
（3） 再寻找链表中空闲的地方，来放置空闲的内存块。如果找到了就直接放，如果没找到就要请求更多的内存空间
要点分析：
每一次申请内存块空闲链表不一定都可以满足要求，所以需要设置有请求更多内存空间的措施。

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

函数功能：合并空闲的内存块
处理流程：

（1） 当要回收的块前后的两个块都是被分配的块时，不需要进行合并，直接返回 bp
（2） 当要回收的块后面的块是被分配的块，前面的块是空闲块时，将回收的块和前面的块合并
（3） 当要回收的块前面的块是被分配的块，后面的块是空闲块时，将回收的块和后面的块合并
（4） 当要回收的块后前后两个块都是被分配的块时，将三个块进行合并

要点分析：
合并内存块总共有四种情况，需要一一进行判断，然后合并。合并后更新总合并块的头部和脚部，内存块的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。

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

输入./mdriver -v -t traces/ 命令测试 traces 文件，并输出结果

4.2 自测试结果

```
zxy@ubuntu:~/code/csapp/lab8/malloclab-handout-hit$ ./mdriver -v -t traces/
Team Name:1190200128
Member 1 :Zhan Xianyou:2209376427@qq.com
Using default tracefiles in traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.009479   601
 1       yes   99%    5848  0.008228   711
 2       yes   99%    6648  0.014290   465
 3       yes  100%    5380  0.010709   502
 4       yes   66%   14400  0.000228 63020
 5       yes   92%    4800  0.010015   479
 6       yes   92%    4800  0.009048   531
 7       yes   55%   12000  0.193769    62
 8       yes   51%   24000  0.349399    69
 9       yes   27%   14401  0.090406   159
10       yes   34%   14401  0.003550  4057
Total          74%  112372  0.699123   161

Perf index = 44 (util) + 11 (thru) = 55/100
```

4.3 测试结果评价

没有进行相应的优化，导致测试分数不是很高。

第 5 章 总结

5.1 请总结本次实验的收获

更加熟悉动态存储申请、释放的基本原理和相关系统函数，了解了文件测试的步骤

5.2 请给出对本次实验内容的建议

无。

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.