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 1190202405

班　　级 1903005

学 生 姓 名 林逸灏

指 导 教 师 史先俊

实 验 地 点 G712

实 验 日 期 2021.06.09

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

Visual Studio 2010 64 位以上；CodeBlocks 64 位；vi/vim/gedit+gcc

1.3 实验预习

填写

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。

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。

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倍。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。

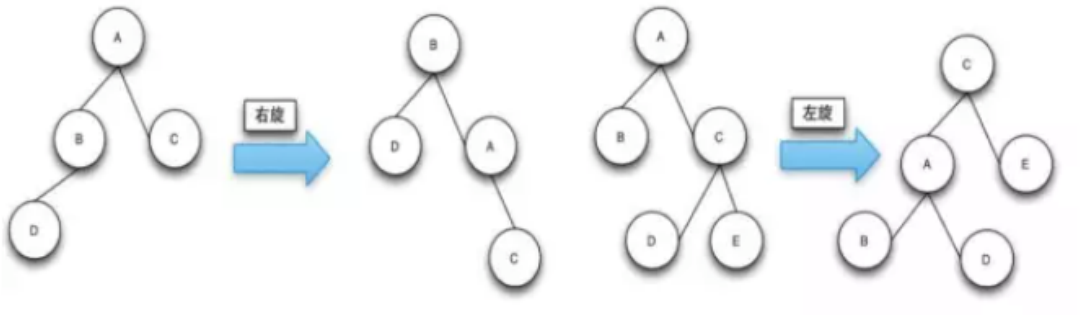
红黑树的查找：

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。

红黑树的更新：

RBTree 的旋转操作

旋转操作(Rotate)的目的是使节点颜色符合定义，让 RBTree 的高度达到平衡。Rotate 分为 left-rotate（左旋）和 right-rotate（右旋），区分左旋和右旋的方法是：待旋转的节点从左边上升到父节点就是右旋，待旋转的节点从右边上升到父节点就是左旋。



RBTree 的插入操作

RBTree 的插入与 BST 的插入方式是一致的，只不过是在插入过后，可能会导致树的不平衡，这时就需要对树进行旋转操作和颜色修复（在这里简称插入修复），使得它符合 RBTree 的定义。

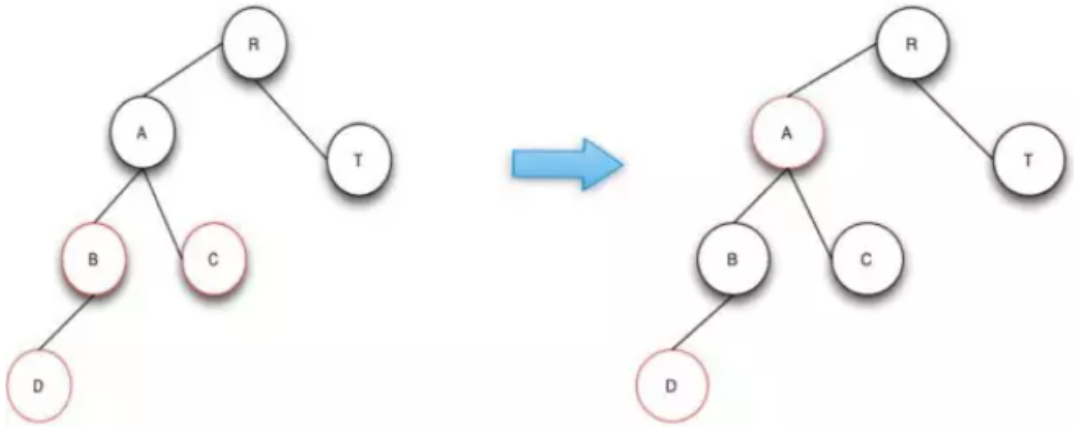
新插入的节点是红色的，插入修复操作如果遇到父节点的颜色为黑则修复操作结束。也就是说，只有在父节点为红色节点的时候是需要插入修复操作的。

插入修复操作分为以下的三种情况，而且新插入的节点的父节点都是红色的：

1. 叔叔节点也为红色。
2. 叔叔节点为空，且祖父节点、父节点和新节点处于一条斜线上。
3. 叔叔节点为空，且祖父节点、父节点和新节点不处于一条斜线上。

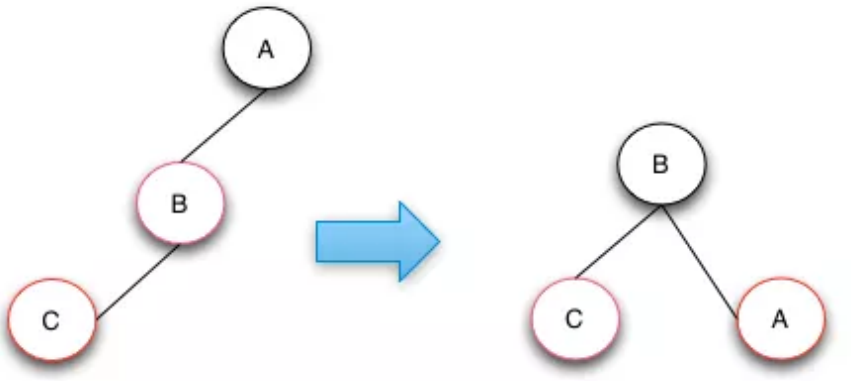
插入操作-Case 1

case 1 的操作是将父节点和叔叔节点与祖父节点的颜色互换，这样就符合了 RBTRee 的定义。即维持了高度的平衡，修复后颜色也符合 RBTree 定义的第三条和第四条。下图中，操作完成后 A 节点变成了新的节点。如果 A 节点的父节点不是黑色的话，则继续做修复操作。



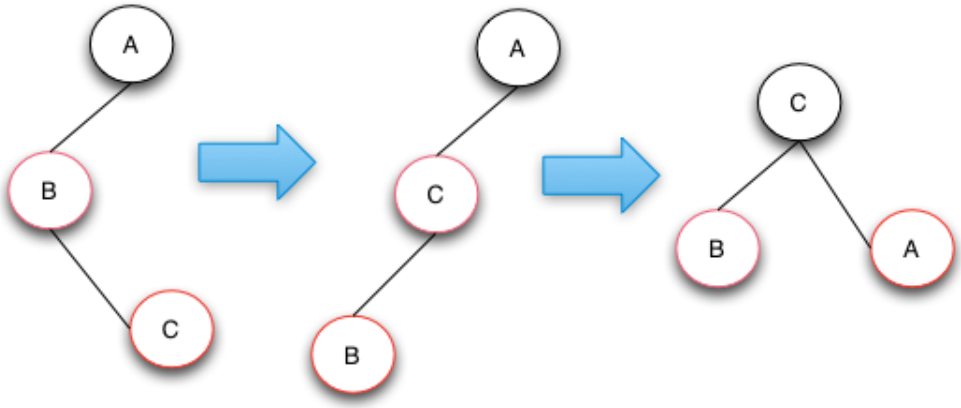
插入操作-Case 2

case 2 的操作是将 B 节点进行右旋操作，并且和父节点 A 互换颜色。通过该修复操作 RBTRee 的高度和颜色都符合红黑树的定义。如果 B 和 C 节点都是右节点的话，只要将操作变成左旋就可以了。



插入操作-Case 3

case 3 的操作是将 C 节点进行左旋，这样就从 case 3 转换成 case 2 了，然后针对 case 2 进行操作处理就行了。case 2 操作做了一个右旋操作和颜色互换来达到目的。如果树的结构是下图的镜像结构，则只需要将对应的左旋变成右旋，右旋变成左旋即可。



插入操作的总结

插入后的修复操作是一个向 root 节点回溯的操作，一旦牵涉的节点都符合了红黑树的定义，修复操作结束。之所以会向上回溯是由于 case 1 操作会将父节点，叔叔节点和祖父节点进行换颜色，有可能会导致祖父节点不平衡(红黑树定义 3)。这个时候需要对祖父节点为起点进行调节（向上回溯）。

祖父节点调节后如果还是遇到它的祖父颜色问题，操作就会继续向上回溯，

直到 root 节点为止，根据定义 root 节点永远是黑色的。在向上的追溯的过程中，针对插入的 3 中情况进行调节。直到符合红黑树的定义为止。直到牵涉的节点都符合了红黑树的定义，修复操作结束。

如果上面的 3 中情况如果对应的操作是在右子树上，做对应的镜像操作就是了。

RBTree 的删除操作

删除操作首先需要做的也是 BST 的删除操作，删除操作会删除对应的节点，如果是叶子节点就直接删除，如果是非叶子节点，会用对应的中序遍历的后继节点来顶替要删除节点的位置。删除后就需要做删除修复操作，使的树符合红黑树的定义，符合定义的红黑树高度是平衡的。

删除修复操作在遇到被删除的节点是红色节点或者到达 root 节点时，修复操作完毕。

删除修复操作是针对删除黑色节点才有的，当黑色节点被删除后会让整个树不符合 RBTree 的定义的第四条。需要做的处理是从兄弟节点上借调黑色的节点过来，如果兄弟节点没有黑节点可以借调的话，就只能往上追溯，将每一级的黑节点数减去一个，使得整棵树符合红黑树的定义。

删除操作的总体思想是从兄弟节点借调黑色节点使树保持局部的平衡，如果局部的平衡达到了，就看整体的树是否是平衡的，如果不平衡就接着向上追溯调整。

删除修复操作分为四种情况(删除黑节点后)：

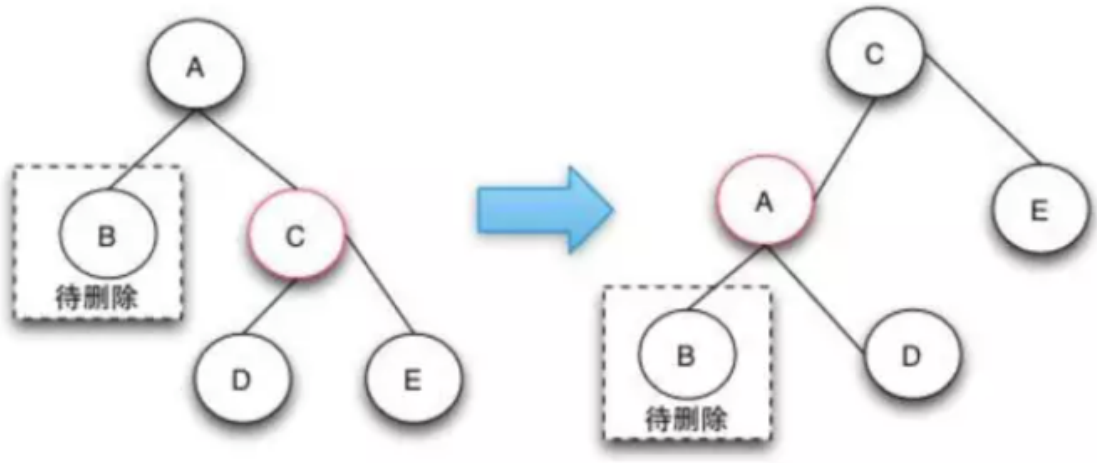
1. 待删除的节点的兄弟节点是红色的节点。
2. 待删除的节点的兄弟节点是黑色的节点，且兄弟节点的子节点都是黑色的。
3. 待调整的节点的兄弟节点是黑色的节点，且兄弟节点的左子节点是红色的，右节点是黑色的(兄弟节点在右边)，如果兄弟节点在左边的话，就是兄弟节点的右子节点是红色的，左节点是黑色的。
4. 待调整的节点的兄弟节点是黑色的节点，且右子节点是是红色的(兄弟节点在右边)，如果兄弟节点在左边，则就是对应的就是左节点是红色的。

删除操作-Case 1

由于兄弟节点是红色节点的时候，无法借调黑节点，所以需要将兄弟节点提升到父节点，由于兄弟节点是红色的，根据 RBTree 的定义，兄弟节点的子节点是黑色的，就可以从它的子节点借调了。

case 1 这样转换之后就会变成后面的 case 2，case 3，或者 case 4 进行处理了。上升操作需要对 C 做一个左旋操作，如果是镜像结构的树只需要做对应的右旋操作即可。

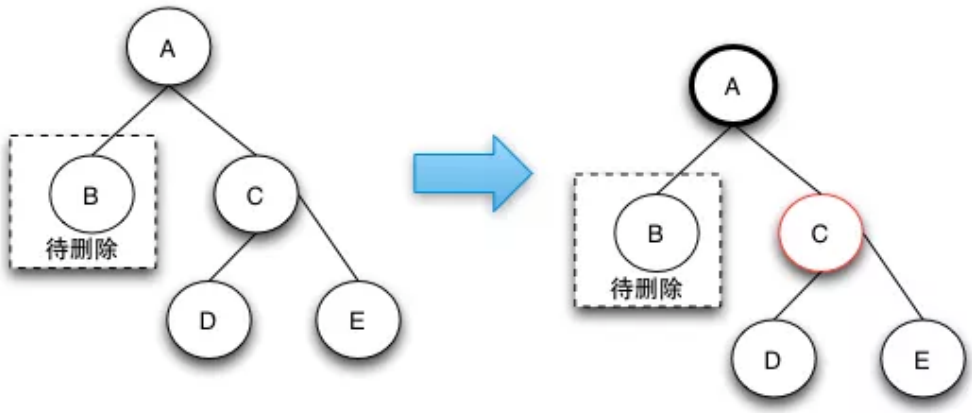
之所以要做 case 1 操作是因为兄弟节点是红色的，无法借到一个黑节点来填补删除的黑节点。



删除操作-Case 2

case 2 的删除操作是由于兄弟节点可以消除一个黑色节点，因为兄弟节点和兄弟节点的子节点都是黑色的，所以可以将兄弟节点变红，这样就可以保证树的局部的颜色符合定义了。这个时候需要将父节点 A 变成新的节点，继续向上调整，直到整颗树的颜色符合 RBTree 的定义为止。

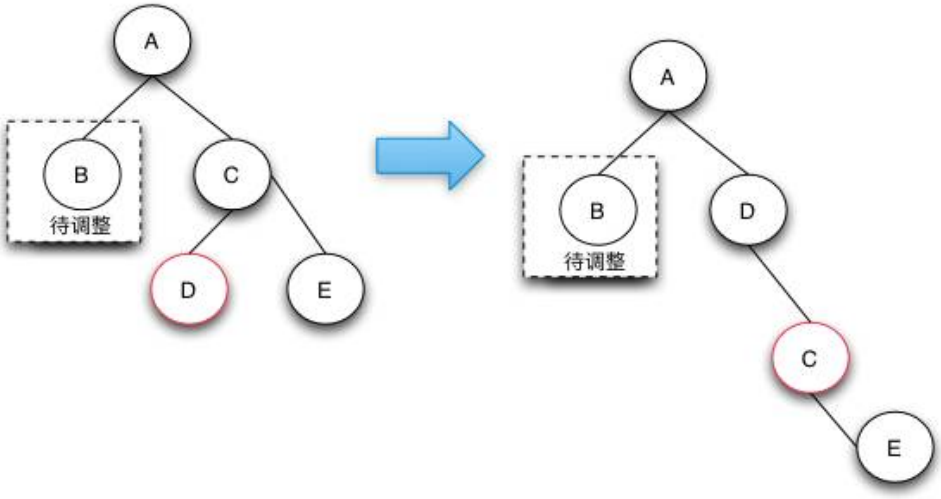
case 2 这种情况下之所以要将兄弟节点变红，是因为如果把兄弟节点借调过来，会导致兄弟的结构不符合 RBTree 的定义，这样的情况下只能是将兄弟节点也变成红色来达到颜色的平衡。当将兄弟节点也变红之后，达到了局部的平衡了，但是对于祖父节点来说是不符合定义 4 的。这样就需要回溯到父节点，接着进行修复操作。



删除操作-Case 3

case 3 的删除操作是一个中间步骤，它的目的是将左边的红色节点借调过来，这样就可以转换成 case 4 状态了，在 case 4 状态下可以将 D，E 节点都阶段过来，通过将两个节点变成黑色来保证红黑树的整体平衡。

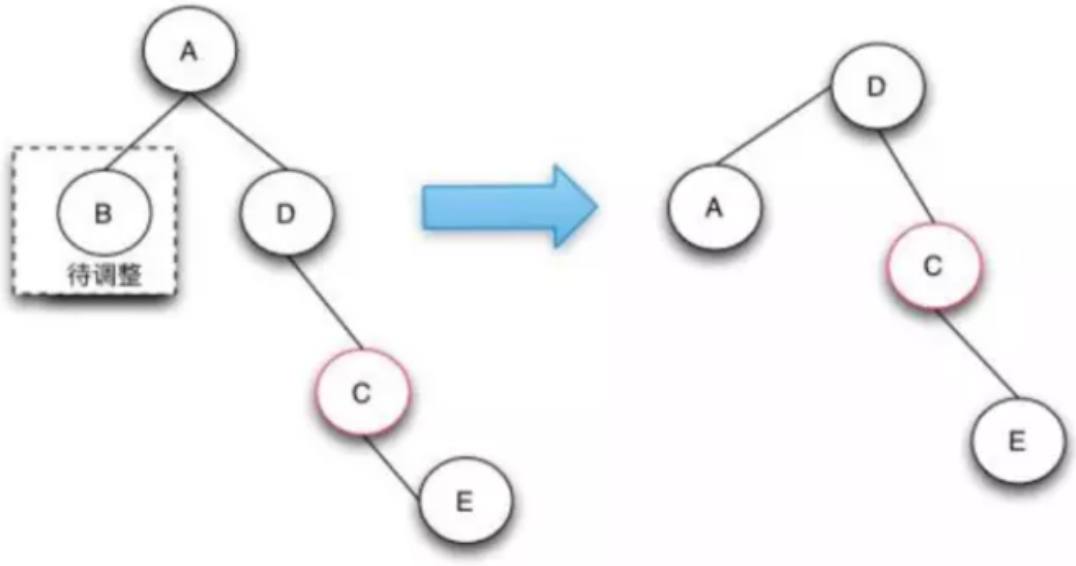
之所以说 case-3 是一个中间状态，是因为根据红黑树的定义来说，下图并不是平衡的，他是通过 case 2 操作完后向上回溯出现的状态。之所以会出现 case 3 和后面的 case 4 的情况，是因为可以通过借用侄子节点的红色，变成黑色来符合红黑树定义 4。



删除操作-Case 4

Case 4 的操作是真正的节点借调操作，通过将兄弟节点以及兄弟节点的右节点借调过来，并将兄弟节点的右子节点变成红色来达到借调两个黑节点的目的，这样的话，整棵树还是符合 RBTree 的定义的。

Case 4 这种情况的发生只有在待删除的节点的兄弟节点为黑，且子节点不全部为黑，才有可能借调到两个节点来做黑节点使用，从而保持整棵树都符合红黑树的定义。



删除操作的总结

红黑树的删除操作是最复杂的操作，复杂的地方就在于当删除了黑色节点的时候，如何从兄弟节点去借调节点，以保证树的颜色符合定义。由于红色的兄弟节点是没法借调出黑节点的，这样只能通过选择操作让他上升到父节点，而由于它是红节点，所以它的子节点就是黑的，可以借调。

对于兄弟节点是黑色节点的可以分成 3 种情况来处理，当所以的兄弟节点的子节点都是黑色节点时，可以直接将兄弟节点变红，这样局部的红黑树颜色是符合定义的。但是整颗树不一定是符合红黑树定义的，需要往上追溯继续调整。

对于兄弟节点的子节点为左红右黑或者 (全部为红，右红左黑)这两种情况，可以先将前面的情况通过选择转换为后一种情况，在后一种情况下，因为兄弟节点为黑，兄弟节点的右节点为红，可以借调出两个节点出来做黑节点，这样就可以保证删除了黑节点，整棵树还是符合红黑树的定义的，因为黑色节点的个数没有改变。

红黑树的删除操作是遇到删除的节点为红色，或者追溯调整到了 root 节点，这时删除的修复操作完毕。

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。
分配块、结构块使用隐式链表。空闲块使用分离的空闲链表。在链表中，各块按照大小进行递增排序，使用每次插入和删除的节点来维护链表的递增顺序。

4. 相应算法：

使用显示分离链表、维护大小递增与首次适配的算法。其中首次适配为每次都选择第一个合适的空闲块进行分配，因为块是按从小到大递增排列的，所以每次选择第一个合适的空闲块能够使得 malloc 操作更加的高效并且能够合理的分配内存空间，减少内存碎片。
在链表操作中，每次添加节点时，都根据新加入的块的 size 进行比较，直到找到合适的位置插入，以满足链表从小到大的递增顺序。

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：应用程序（例如轨迹驱动的测试程序 mdriver）在使用 mm_malloc、mm_realloc 或 mm_free 之前，首先要调用该函数进行初始化。用于初始化内存系统模型，例如申请初始堆区域、初始化分离空闲链表。

处理流程：

1. 先初始化分离空闲链表。
2. 使用 heap_listp = mem_sbrk(4 * WSIZE)函数从内存中得到 4 个字，并将 heap 初始化。
3. 新建空闲链表。使用 PUT 函数对 heap 进行填充，补充相应的内容。
   a) 先 PUT(heap_listp, 0);填充 0，使这块区域边界对齐；
   b) PUT(heap_listp + (1 * WSIZE), PACK(DSIZE, 1)); PUT(heap_listp + (2 * WSIZE), PACK(DSIZE, 1));这两行代码创建并分配了一个 8 字节的 Prologue 块，其中 header 占 4 个字节，footer 占 4 个字节，这一块被标记为已分配
   c) PUT(heap_listp + (3 * WSIZE), PACK(0, 1));这段代码的作用是表示一个 Epilogue header，表示堆的结尾，这个块是大小为零的已分配块。
4. 最后调用 extend_heap 函数，堆拓展并创建空闲块。

要点分析：
mm_init 初始化分配器使用的最小空间为 4 * WSIZE，即 16 字节。使用 PUT 函数来新建并初始化隐式空闲链表，该链表由左端的对齐填充块与 4 个字节的 Prologue header、4 个字节的 Prologue footer 和 4 个字节的 Epilogue 组成。最后，在空闲链表创建之后使用 extend_heap 函数来扩展堆，extend_heap 函数中使用 ALIGN 函数，每次都将请求扩展的堆大小对齐为 8 字节的倍数。

3.2.2 void mm_free(void *ptr)函数（5 分）

函数功能：释放参数 “ptr”指向的已分配内存块，没有返回值。指针值 ptr 应该是之前调用 mm_malloc 或 mm_realloc 返回的值，并且没有释放过。

参　　数：void *ptr
处理流程：
1. 调用 GET_SIZE(HDRP(ptr))函数来获得 ptr 指针指向的需要释放的块的大小。
2. 调用 PUT(HDRP(ptr), PACK(size, 0)); PUT(FTRP(ptr), PACK(size, 0));将请求块的头部和脚部的已分配位置全部为 0。
3. 调用 InsertNode(ptr, size)将该块插入到分离空闲链表
4. 调用 coalesce(ptr), 将释放的块 ptr 与相邻的空闲块合并起来。

要点分析：
为了能够正确地释放内存，需要调用 GET_SIZE 函数来获取需要释放块的大小，这样才不会错误释放本不应该释放的内存。在将需要释放的块置零，标记为空闲之后，需要调用 coalesce 函数将该块与周围的空闲块合并，否则可能会产生大量内存碎片。

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

函数功能：向 ptr 指针所指的块重新分配一个具有至少 size 字节的有效负载的块。
- 如 ptr 是空指针 NULL,等价于 mm_malloc(size)
- 如果参数 size 为 0，等价于 mm_free(ptr)
- 如 ptr 非空, 它应该是之前调用 mm_malloc 或 mm_realloc 返回的数值，指向一个已分配的内存块。

参　　数：void *ptr, size_t size
处理流程：
1. 先判断 size 的大小，若 size 为 0，等价于 mm_free(ptr);若 size<=DSIZE，则调用 ALIGN 函数对 size 进行对齐；若 size 小于原来块的大小，则直接返回原来的块；若 size 大于原来块的大小，则需要扩展内存空间，realloc。
2. 扩展时，先判断原来的块的地址的下一个块是否未分配，若是，则将原来的块空间扩展至下一个空闲块；若下一个空闲块的空间不足，需要调用 extend_heap 函数来扩展堆空间。
3. 若原来的块的下一个块空间已分配，无法扩展，则申请新的未分配块，使用 memcpy 函数将原块内容复制到新的块中，再调用 mm_free 释放掉原块的内存。

要点分析：
realloc 函数首先需要根据输入的 size 的大小实现内存对齐。为了节省内存空间，减少外部碎片，应尽量合并周围的空闲块。若周围的空闲块内存空间不足，则需要扩展堆空间。若周围没有可以利用的空闲块，则需要申请新的不连续的未分配块，将原块的内容复制到新的块中。

3.2.4 int mm_check(void)函数（5 分）

函数功能：检查堆的一致性。检查重要的不变量和一致性条件。当且仅当堆是一致的，才能返回非 0 值。

处理流程：

1. 定义一个指向全局变量 head_listp 的指针 bp，便于遍历指向各个块头部的链表。
2. 先检查 Prologue block，若它不是 8 字节的已分配块，则输出错误提示信息
3. 然后从 head_listp 开始，利用前面定义的 bp 指针，遍历所有的链表，检查每个块是否被占用，检查每个块的头部和脚部，判断是否匹配，检查是否双字对齐。
4. 最后再检查 Epilogue block，若它不是大小为零的已分配块，则提示错误信息。

要点分析：

需要获取链表的开始指针，便于遍历所有的链表，本函数检查了序言块、结尾块，以及他们之间的块是否匹配，是否符合一致性的要求。

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

函数功能：申请有效载荷至少是参数“size”指定大小的内存块，返回该内存块地址首地址（可以使用的区域首地址）。申请的整个块应该在对齐的区间内，并且不能与其他已经分配的块重叠。返回的地址应该是 8 字节对齐的（地址%8==0）。

参　　数：size_t size

处理流程：

1. 先判断 size 的大小，若 size = 0，malloc 请求无效，返回 NULL 空指针；若 size<=DSIZE 则需要进行对齐，以满足双字对齐的要求。最小块的大小为 16 字节，其中 8 个字节用来存储头部与脚部，8 个字节的有效载荷用来满足对齐要求，若这一部分超过了八字节，需要向上对齐到 8 的整数倍。
2. 通过对齐处理之后 size 的大小，在分离空闲链表中进行搜索，直到找到一个合适的空闲块；如果没有找到合适的空闲块，则调用 extend_heap 函数，扩展堆大小。
3. 在能够分配内存空间的空闲块中，分配器调用 place 函数来放置这个请求块，分配指定大小的空间，分割出多余的内存空间，返回指向这个新分配块的指针。

要点分析：

先需要将 size 大小调整为满足内存对齐的大小，然后在分离空闲链表中寻找合适的空闲块，若找不到合适的空闲块，则调用 extend_heap 函数来扩展堆空间。在合适的块中，使用 place 函数来放置请求块，每次放置后都需要分割出多余的部分，以减小请求块的内部碎片，节省内存资源。

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

函数功能：将要回收的空闲块和临近的空闲块（如果有的话）合并成一个大的空闲块，并返回合并后的空闲块指针。

处理流程：按书中所示，分配共分为四种情况：

1. 如果前面的块与后面的块都是已分配的，不可能进行合并，所以直接返回当前块。
2. 如果前面的块是已分配的，后面的块是空闲的，则在显式分离链表中删除当前块与后面的块，将两个块合并，用当前块和后面的块的大小之和来更新合并后的块的头部与脚部。
3. 如果前面的块是空闲的,而后面的块是已分配的，和上一个情况类似，在显式分离链表中删除当前块与前面的块，将两个块合并，用当前块和前面的块的大小之和来更新合并后的块的头部与脚部。
4. 如果前面的和后面的块都是空闲的，则在显式分离链表中将这三个块都删除，然后将这三个块合并成一个单独的空闲块，用这三个块的大小之和来更新合并后的块的头部与脚部。
5. 合并完之后，将上述得到的新的空闲块插入到显式分离链表中。

要点分析：

需要分清合并的四种情况，对每一种不一样的情况进行单独处理。需要在显式分离链表中删除需要的合并块，合并完之后需要更新对应的头部与脚部，最后再把合并的块插入到显式分离链表中。

第 4 章测试

总分 10 分

4.1 测试方法

- 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

性能分 pindex 是空间利用率和吞吐率的线性组合：

4.2 自测试结果

```
→ 1190202405-malloclab-handout-hit ./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   98%    5694  0.000306 18590
 1       yes   97%    5848  0.000327 17906
 2       yes   98%    6648  0.000382 17403
 3       yes   99%    5380  0.000302 17826
 4       yes   99%   14400  0.000507 28430
 5       yes   93%    4800  0.000532  9014
 6       yes   91%    4800  0.000533  8999
 7       yes   55%   12000  0.000461 26030
 8       yes   51%   24000  0.000929 25843
 9       yes  100%   14401  0.000312 46231
10       yes   87%   14401  0.000260 55346
Total          88%  112372  0.004851 23167

Perf index = 53 (util) + 40 (thru) = 93/100
```

4.3 测试结果评价

使用了显式分离空闲链表，又使用了维护大小递增与首次适配的算法，减少了很多不必要的内存碎片，使得内存利用率与吞吐率达到了较好的效果。

第 5 章 总结

5.1 请总结本次实验的收获

了解了虚拟内存的分配，合并等过程，对虚拟内存分配器的实验原理有了更深的理解。

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.