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 1190202107

班　　级 1936602

学 生 姓 名 姚舜宇

指 导 教 师 刘宏伟

实 验 地 点 G709

实 验 日 期 2021 6 10

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

Vmware 11；Ubuntu 18

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

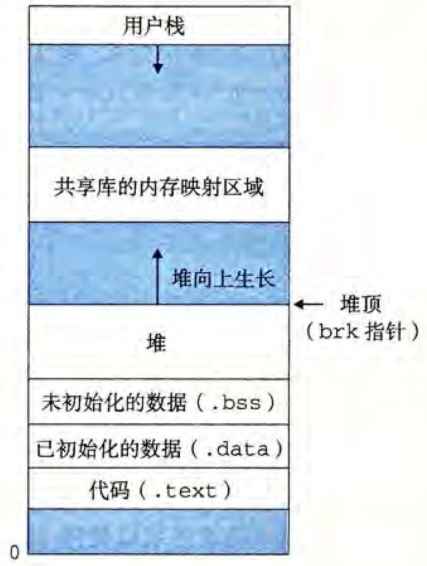
■ 熟知动态内存申请的内部实现机制：分配算法、释放合并算法等

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。对于每个进程，内核维护者一个变量 brk，指向堆的顶部。堆的示意图如下。



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。空闲块可以用来分配。一个已分配的块保持已分配状态，直到它被释放。分配器有两种基本风格，显式的和隐式的。

显式分配器，要求应用显式地释放任何已分配的块。如 C 标准库提供的 malloc 程序包显式分配器。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。

隐式分配器，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。

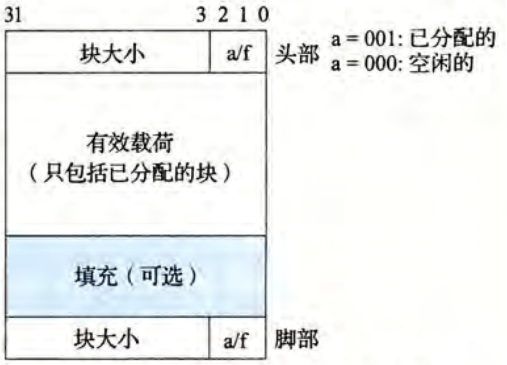
一般来说，一个分配器需要能够解决以下几个问题：

1. 空闲块组织：记录空闲块。
2. 放置：选择一个合适的空闲块来放置一个新分配的块。

3. 分割：在将一个新分配的块放置到某个空闲块之后，处理这个空闲块中的剩余部分。
4. 合并：处理一个刚刚被释放的块。

2.2 带边界标签的隐式空闲链表分配器原理（5 分）

带边界标签的隐式空闲链表的数据结构的示意图如下：



每一个堆块内有一些字，每个字有 4 个字节，即 32 位。第一个字，称为头部，记录这个堆块的大小，以及是已分配的还是空闲的。堆块的脚部有一个头部的副本，存储相同的信息。这里介绍的堆块是双字对齐的，所以块大小一定为 8 的倍数，二进制的低第三位是 0。所以用最低位来表示这个块是以分配的还是空闲的。有效载荷就是用户申请的空间，填充是不使用的，大小任意，填充可能是分配器策略的一部分，用来对付外部碎片，或者用它来满足对齐要求。

动态存储的分配管理主要包括以下几个操作：

1. 放置已分配的块

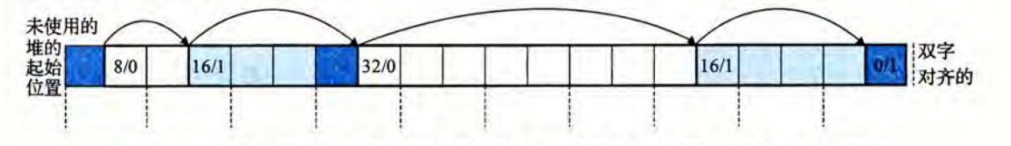
当应用请求一个 k 字节的块时，分配器搜索空闲链表，查找一个足够大的可以放置所请求块的空闲块。分配器执行这种搜索的方式是由放置策略确定的，有首次适配、下一次适配、最佳适配等。

首次适配：从头开始搜索空闲链表，选择第一个合适的空闲块。下一次适配：从上一次查询结束的地方开始搜索。最佳适配：检查每个空闲块，选择适合所需请求大小的最小空闲块。

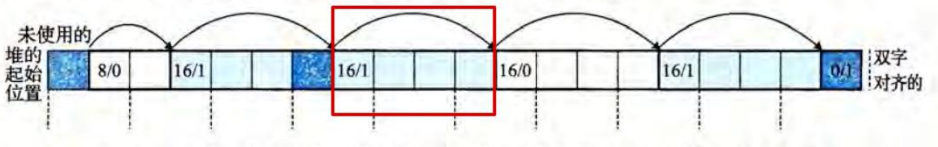
2. 分割空闲块

一旦分配器找到一个匹配的空闲块，就必须考虑分配这个空闲块中的多少空间。如果匹配不太友好，则分配器通常会选择将这个空闲块分割为两部分。第一部分变成分配块，剩下的部分变成一个新的空闲块。

例如，目前的堆情况如下图所示：



现在有一个 3 个字的分配请求，因为第一个空闲块空间不够，所以将第二个空闲块分割来分配。分配情况如下图，红色方框的位置即为新分配的块。第一个字保存这个分配块的信息，后三个字保存有效载荷。

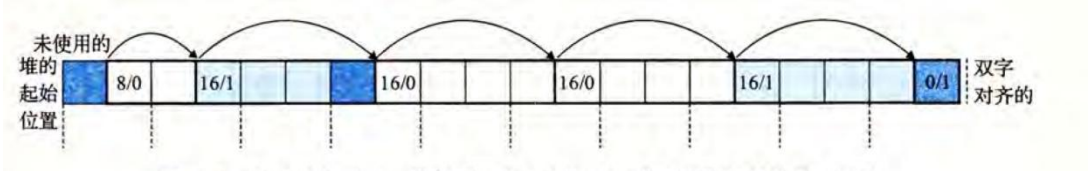


3. 获取额外的堆内存

如果分配器不能为请求块找到合适的空闲块，可以通过合并那些在内存中物理相邻的空闲块来创建一个更大的空闲块。如果这样还是不能生成一个足够大的块，则分配器会调用 sbrk 函数，向内核请求额外的堆内存。分配器将额外的内存转化为一个大的空闲块，将这个块插入到空闲链表中，然后将被请求的块放置在这个新的空闲块中。

4. 合并空闲块

当分配器释放一个已分配块石，可能有其他空闲块与这个新释放的空闲块相邻，如下图所示。



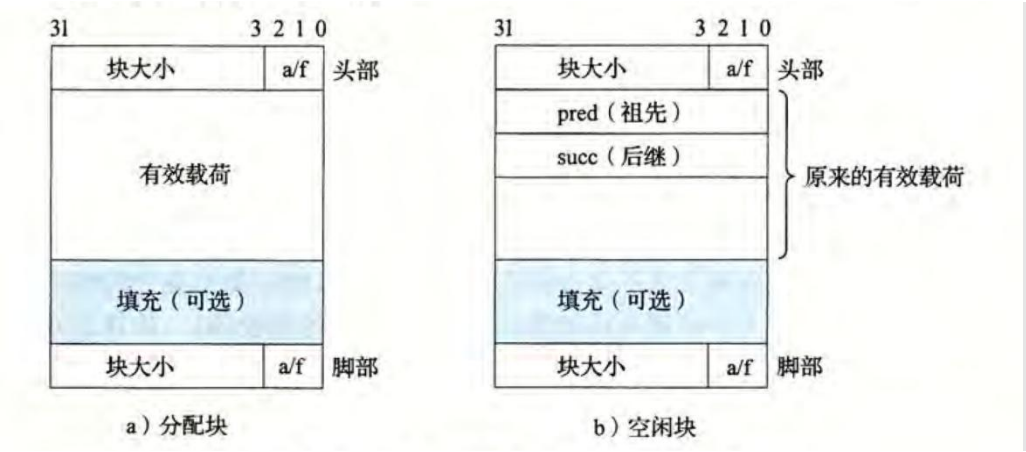
可以看到，有两个相邻的空闲块。此时如果请求一个 4 字的空闲块，分配器发现当前的空闲块无法满足要求，就会合并空闲块，将上图中两个相邻的空闲块合并成为一个大的空闲块。

对于这种带边界标签的隐式空闲链表，可以在常数时间内合并空闲块。因为头部和脚部都存储了本堆块的大小和是否空闲的信息，所以可以通过算术运算计算出当前堆块的前一个块和后一个块是否空闲。

2.3 显式空间链表的基本原理（5 分）

显式空闲链表是将空闲块组成为某种形式的显式数据结构。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，

其分配块和空闲块的结构示意图如下：



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。释放一个块的时间可以是线性的，也可能是一个常数，这取决于所选择的空闲链表中块的排序策略。维护链表的方法有两种，后进先出的顺序，和地址顺序。

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。使用 LIFO 的顺序和首次适配的放置策略，分配器会最先检查最近使用过的块。在这种情况下，释放一个块可以在常数时间内完成。如果使用了边界标记，那么合并也可以在常数时间内完成。

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。在这种情况下，释放一个块需要线性时间的搜索来定位合适的前驱。平衡点在于，按照地址排序的首次适配比 LIFO 排序的首次适配有更高的内存利用率，接近最佳适配的利用率。

2.4 红黑树的结构、查找、更新算法（5 分）

红黑树的结构

红黑树是一种二叉查找树，但每个节点上增加一个存储位表示节点的颜色，可以是红色或黑色。红黑树有五条特性。

1. 节点的颜色是红色或黑色。
2. 根节点一定是黑色。
3. 所有的叶节点一定是黑色。
4. 任意节点到叶节点经过的黑色节点数目相同。
5. 不会有连续的红色节点。

红黑树可以看作是对概念模型 2-3 树的一种实现。为了降低红黑树调整过程中

的复杂度，在这里我们要求概念模型中的 3-节点在红黑树中必须用左倾的红节点来表示。对于 2-3 树中的 2-节点（有两个孩子），直接变为黑色。对于 3-节点（有 3 个孩子），变成左倾红黑树。即 3-节点中右边的数成为黑色节点，黑色节点的左儿子是内容是 3-节点中左边的数的红色节点。

红黑树的查找算法

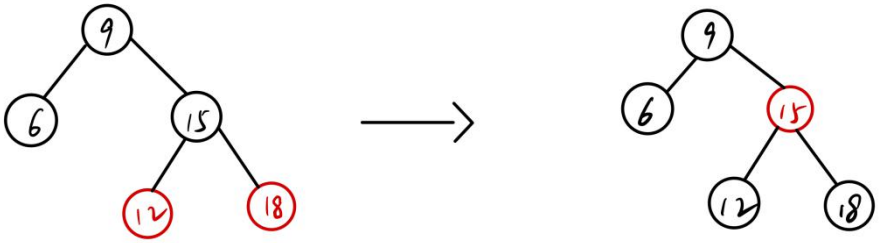
红黑树的查找和普通的二叉树是一样的。当想要查找某个数的时候，把这个数和根节点比较，如果比根节点大，就和根节点的右儿子比较，如果小，就和根节点的左儿子比较。以此类推。

红黑树的插入

红黑树的插入有三种情况。

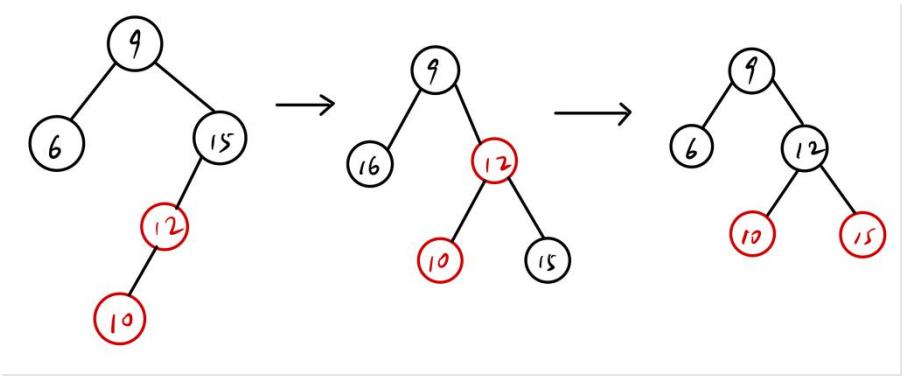
1. 待插入的元素比黑色父节点大，插入到黑色父节点的右边。而父节点的左儿子是红色，这样会导致在红黑树中出现右倾红节点。

   下例插入 18。这种情况对应了 2-3 树中出现了临时 4-节点，2-3 树中的处理是将这个临时 4-节点分裂，左右元素各自形成一个 2-节点，中间元素上升到上层和父节点结合。所以，在红黑树中，将原本红色的左右儿子染黑，将黑色父节点染红。如果染红的父节点的父节点是红色，会继续视情况处理。



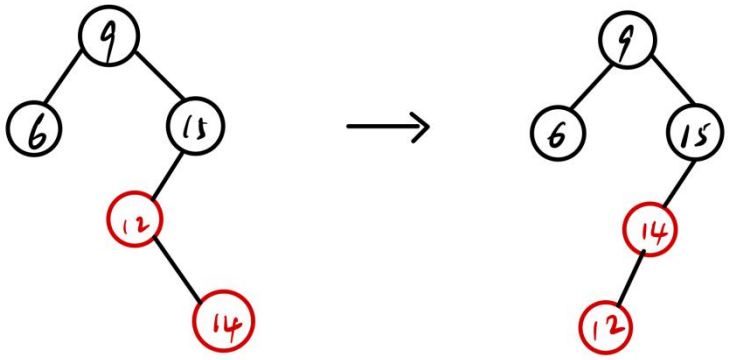
2. 待插入的元素比红色父节点小，且红色父节点自身是左倾，插入后会形成连续的红色节点。

   下例插入 10。这种情况需要分两步来调整。因为插入的是红色节点，所以不会破坏黑色的平衡，要注意的是在旋转和染色中继续保持黑色平衡。第一步对 15 所在的节点进行依次右旋，使得 12 占据 15 的位置。第二部将 12 和 15 互换颜色，消除连续红节点。



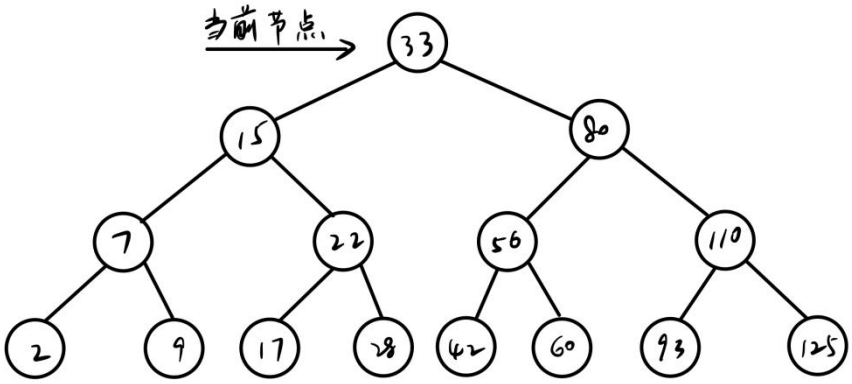
3. 待插入的元素比红色父节点大，且红色父节点自身是左倾。

也就是说插入的这个节点形成了一个右倾的红节点。对右倾的处理很简单，将红色父节点进行一次左旋，就能使得右倾红节点变为左倾。下例插入 14。现在出现了连续的左倾红节点，直接按照情况 2 处理即可。

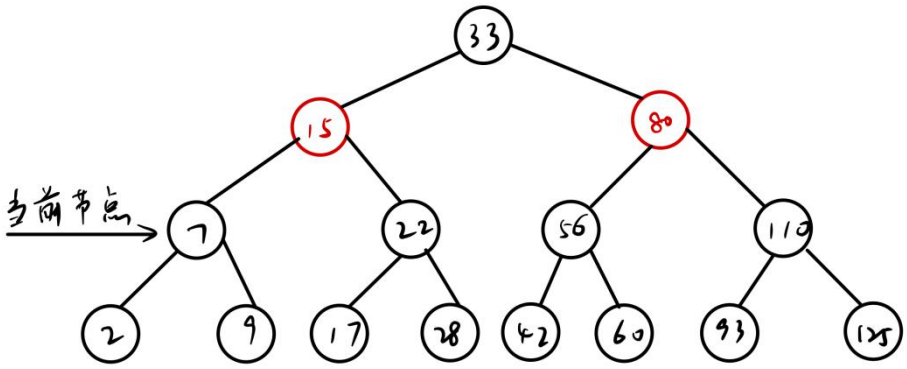


红黑树的删除

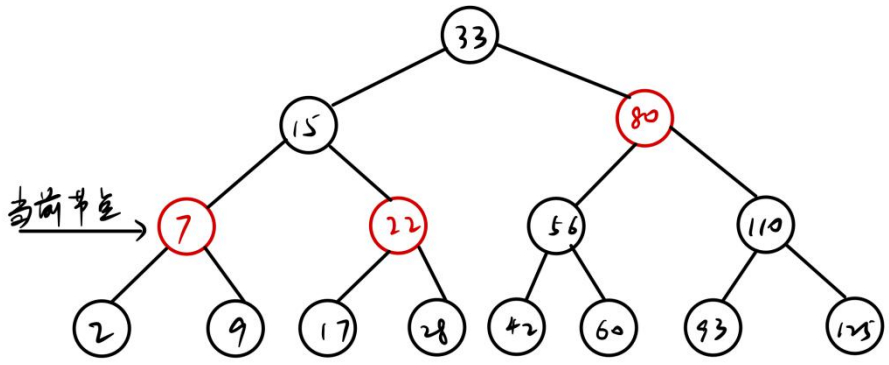
左倾红黑树的删除需要借鉴二叉查找树通用的删除策略。当要删除某个节点的时候，选择它的前驱节点或后继节点元素来替代它，转而删除它的前驱或后继节点。在这里我们选择用后继节点来替代被删除节点。假如需要删除的节点它的右子树如图，那么对该节点的删除实际上就是对 2 的删除。从当前的根节点触发，利用 2-3 树中的预合并策略逐层对红黑树进行调整。具体做法是：每次都保证当前的节点是 2-3 树中的非 2-节点，如果当前节点已经是非 2-节点，那么直接跳过。如果是 2-节点，那么根据兄弟节点的状况进行调整。如果兄弟是 2-节点，那么从父节点借一个元素给当前节点，然后与兄弟节点一起形成一个临时 4-节点。如果兄弟是非 2-节点，那么兄弟上升一个元素到父节点，同时父节点下降一个元素到当前节点，使得当前节点成为一个 3-节点。这种策略能保证最后走到待删除节点的时候，它一定是一个非 2-节点，可以直接将其元素删除。



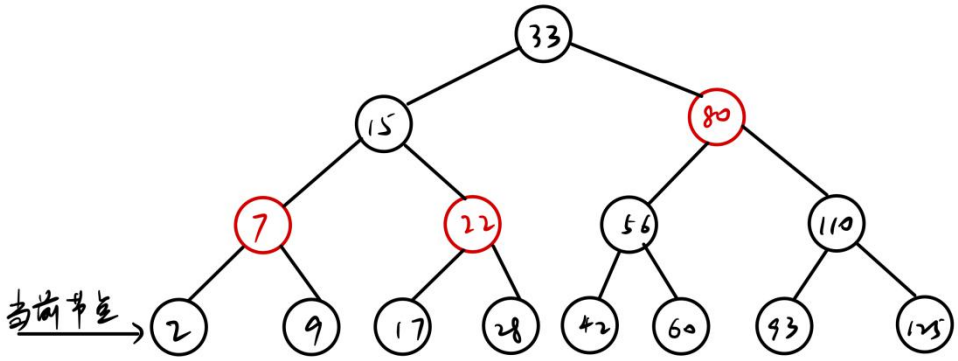
上图中，当前节点是 2-节点，需要合并，由于它是根节点，不存在父节点，所以直接把它与两个儿子合并成临时 4-节点，之后下移一层。



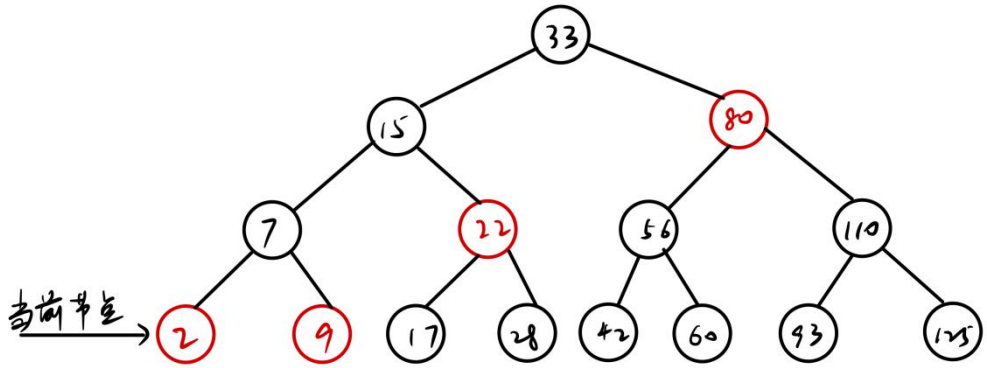
本来当前节点应该指向 15，但是由于 15 已经与 33 合并成了临时 4-节点，因此 15 其实和 33 在同一高度，那么下降会来到 7 所在的节点。当前节点是 2-节点，而它的父节点是 4-节点，兄弟是 2-节点，因此父节点下调一个元素，与当前节点和兄弟一起形成一个临时 4-节点。33 现在是一个临时 4-节点了。第二层的双红节点，要想到 2-3 树中的临时 4-节点。



当前节点现在是一个临时 4-节点了，继续下移，将会去到元素 2 所在的节点。

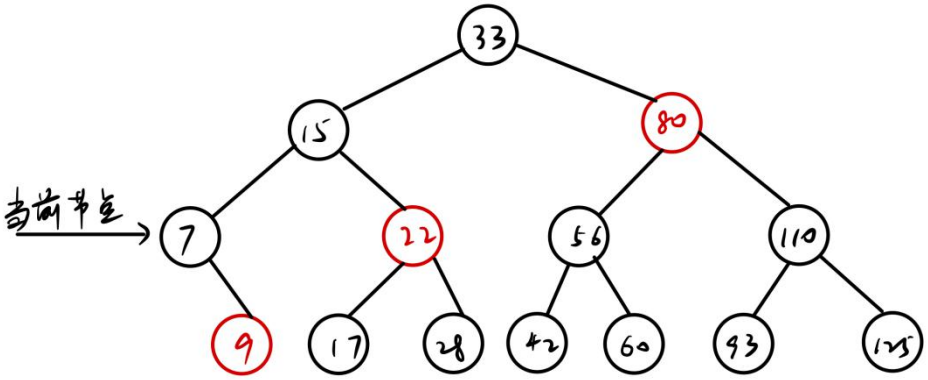


当前节点是一个 2-节点，它的父节点是临时 4-节点，兄弟是 2-节点，将父节点下调一个元素，与它和兄弟合并成一个临时 4-节点。



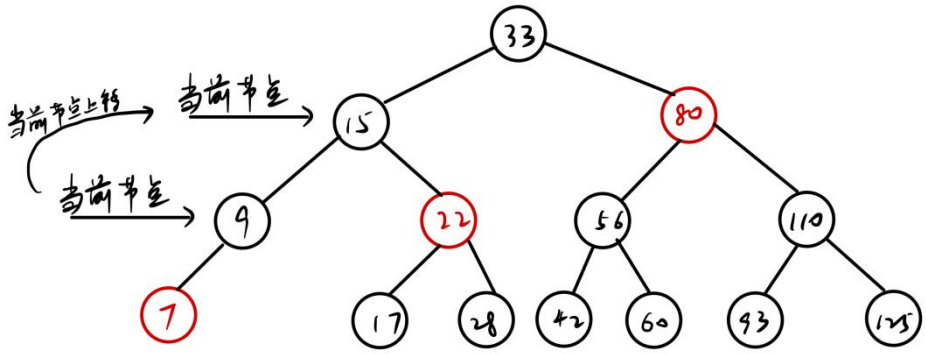
现在当前节点是一个临时 4-节点，如果想删除 2，可以直接删除。

接下来开始修复工作。由于红黑树定义的限制，在调整过程中出现了一些本不该存在的红色右倾节点，于是顺着搜索的方向向上回溯，如果遇到当前节点具有红色的右儿子，那么对当前节点进行一次左旋，这时原本的右儿子会来到当前节点的位置，然后将右儿子和当前节点交换颜色，就将右倾红色节点修复成了左倾红色节点，同时保证了黑色节点的平衡。

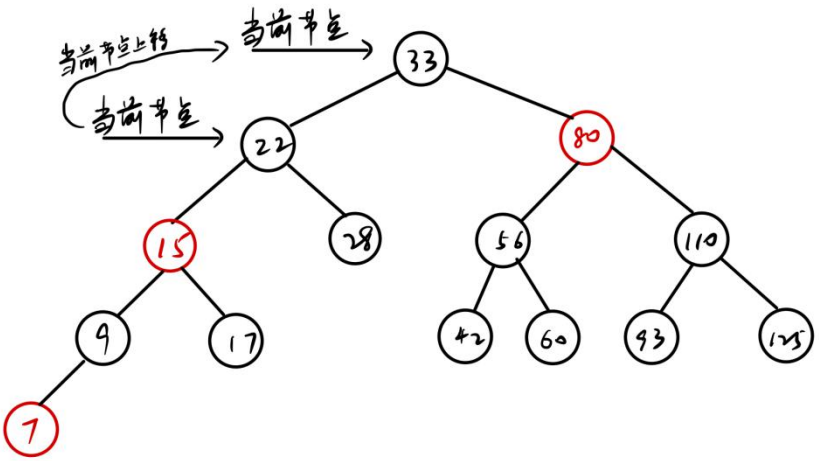


将元素 2 从临时 4-节点中删除，接下来修复一些在生成临时 4-节点的过程中出现的右倾红色节点，当前节点来到 7 的位置，开始向上调整。对 7 所在节点进

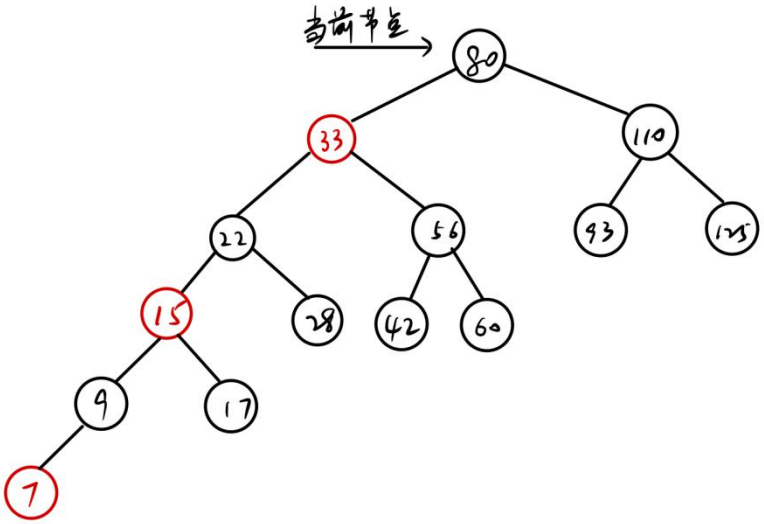
行一次左旋以及重新染色即可修复右倾红色节点。



9 没有右倾红色节点，直接上升到 15 的节点，发现 15 存在右倾的红色节点 22，需要进行修复，同意是对 15 所在节点进行一次左旋以及重新染色。



22 没有右倾红色节点，直接上升到 33 的节点，发现 33 存在右倾的红色节点 80，需要进行修复，对 33 所在的节点进行一次左旋以及重新染色。



至此，本次针对 2 所在节点的删除和修复工作已经全部结束。

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。

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。对于每个进程，内核维护者一个变量 brk，指向堆的顶部。堆的示意图如下。

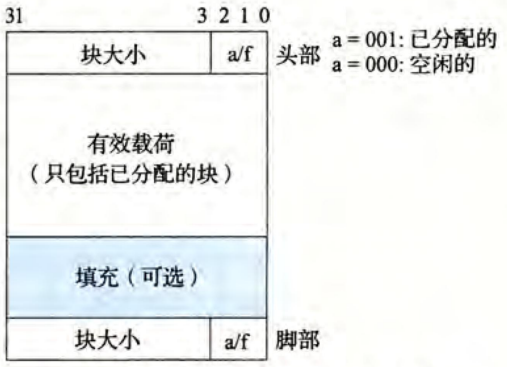
堆中内存块的组织结构：

一种较为简单的叫隐式空闲链表的数据结构，其结构示意图如下：

每一个堆块内有一些字，每个字有 4 个字节。第一个字记录这个堆块的大小，以及是已分配的还是空闲的。这里介绍的堆块是双字对齐的，所以块大小一定为 8 的倍数，二进制的低第三位是 0。所以用最低位来表示这个块是以分配的还是空闲

的。有效载荷就是用户申请的空间，填充是不使用的，大小任意，填充可能是分配器策略的一部分，用来对付外部碎片，或者用它来满足对齐要求。

有一中更加优化的数据结构能够在常数时间内进行合并。



相比于前面的隐式空闲链表结构，这种结构在块的脚部有一个头部的副本，这样无论是从当前的块向前还是向后合并，都可以检查前一个块或者是后一个块是否是空闲的。如果是，就将它的大小简单地加到当前块头部的大小上，使得这两个块在常数时间内被合并。

采用的空闲块、分配块链表：

隐式空闲链表，边界标记结构的方式以便于合并。用分离适配的方法进行合并。一个链表中，块按照大小进行升序排序，每次插入和删除节点是都会检查维护其顺序。

相应算法：

一种减少分配时间的方法，称为分离存储，就是维护多个空闲链表，其中每个链表中的块有大致相等的大小。将所有可能的块大小分成一些等价类。分配器维护着一个空闲链表数组，每个大小类一个空闲链表，按照大小的升序配列。当分配器需要一个大小为 n 的块时，它就搜索相应的空闲链表。如果不能找到合适的块与之匹配，就搜索下一个链表。有关动态内存分配有两种基本的方法，简单分离存储和分离适配。

简单分离存储：每个大小类的空闲链表包含大小相等的块，每个块的大小就是这个大小类中最大元素的大小。为了分配一个给定大小的块，检查相应的空闲链表。如果链表非空，就分配其中第一块的全部。空闲块不会分割以满足分配请求。

分离适配：分配器维护者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，这些块的大小是大小类的成员。一种简单的分离适配分配器如下。为了分配一个块，必须确定请求的大小类，并且对适当的空闲链表做首次适配，查找一个合适的块。如果找到了一个，那么就(可选地)分割它，并将剩余的部分插入到适当的空闲链表中。如果找不到合适的块，那么就搜索下一个更大的大小类的空闲链表。如此重复，直到找到一个合适的块。如果空闲链表中没有合适的块，那么就向操作系统请求额外的堆内存，从这个新的堆内存中分配出一个块，将剩余部分放置在适当的大小类中。要释放一个块，我们执行合并，并将结果放置到

相应的空闲链表中。

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：初始化分配器，如果成功则返回 0，否则返回-1。
处理流程：
1. 从内存中得到 4 个字，创建一个序言块。
2. 设置对齐格式和初始化头部和脚部。
3. 调用 extend_heap 函数，扩展堆的容量大小。

要点分析：
为了使头部和脚部存储的信息能够正常运作，需要设置对齐格式。设置完之后才调用 extend_heap 函数进行堆容量的扩展。
初始化空闲块的大小为 16 字节，表示分配器使用的最小块的大小为 4 字。

3.2.2 void mm_free(void *ptr)函数（5 分）

函数功能：释放一个以前分配的块。
参　　数：想要释放的分配块的首地址指针。
处理流程：
1. 使用 GET_SIZE 获得想要释放的块的大小。
2. 将该块的头部和脚部的表示是否分配的位修改为 0，表示空闲块。
3. 调用 colaecse 函数，合并空闲块。

要点分析：
　　释放一个块，就是修改标记位，并且为了减少碎片的产生，方便以后更好地服用堆的内存空间，选哟合并空闲块，恢复成原本的状态。

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

函数功能：将已分配的块重新分配一个块。
参　　数：ptr，是需要修改的块的首地址指针。size，表示需要分配的字节数。
处理流程：
1. 用 mm_malloc 函数向堆中申请一块大小为 size 字节的空间。如果有，则获得这段空间的指针。
2. 获取原本块的大小，如果新空间大小小于旧空间大小，则用新空间大小更新旧空间大小。然后将已更新或不需要更新的旧空间大小的旧空间内容复制到新空间中。
3. 使用 mm_free 函数释放旧的分配块。

要点分析：
　　如果新空间大小小于旧空间大小，需要用新空间大小更新旧空间大小后再进行内容的复制，否则可能会把超出新空间大小的内容复制到新空间，引起不可预料的后果。而且内容复制结束后需要将旧空间释放，避免占用堆空间。

3.2.4 int mm_check(void)函数（5 分）

函数功能：检查堆的一致性。
处理流程：
1. 检查序言块，如果不存在或不是 8 字节则报错。
2. 依次遍历每一个已分配块或空闲块，调用 checkblock 函数，检查是否为双字对齐且头部和脚部信息一致，如果不是双字对齐或头部和脚部信息不一致，则报出相应的错误。
3. 检查结尾块，是否是一个大小为 0 的已分配块，如果不是，则报出相应错误。

要点分析：
除了检查头部和脚部信息是否符合外，还要检查是否是双字对齐。如果不是双字对齐，头部和脚部的信息存储方案就会发生错误。

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

函数功能：向堆中申请一个大小为 size 字节的块。
参　　数：size，表示要申请的空间的大小。
处理流程：
1. 判断参数是否合法。如果 size 小于或等于 0，直接返回 NULL。
2. 判断申请空间的大小是否小于或等于 8 字节，保证最小分配块的大小为 16 字节。其中 8 字节存放头部和脚部，8 字节满足对齐要求。如果申请的空间超出 8 字节，则计算分配块的大小，同时满足申请的空间和 8 字节对齐的最小大小。
3. 计算完大小之后，开始在堆中寻找一个大小适合的空闲块，找到之后就使用这个空闲块，返回首地址指针。如果没有找到适合的空闲块，就申请扩展堆的空间，扩展结束之后使用刚刚增加的空间来分配给请求的块。

要点分析：
需要计算好分配块的大小，充分考虑到申请的空间大小，是否满足 8 字对齐，头部和脚部是否能够被存放。计算之后进行查找。如果不存在合适的空闲块需要对堆进行扩展后再进行分配。

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

函数功能：将相邻的几个空闲块合并成一个完整的空闲块。
处理流程：
1. 因为是带边界标签的隐式空闲链表，所以可以以同样的效率向前和向后查找。可以分为四种情况，前后都已分配；向后有空闲块而向前没有；向前有空闲块而向后没有；前后都有空闲块。
2. 第一种情况，前后都是已分配的块。则直接返回 bp。
3. 第二种情况，向后有空闲块而向前没有。将 size 更新为此空闲块的大小和向后空闲块的大小之和，重新设置空闲块的头部和脚部信息。
4. 第三种情况，向前有空闲块而向后没有。将 size 更新为此空闲块的大小和向前空闲块的大小之和，重新设置空闲块的头部和脚部信息，并且使空闲块指针指

向向前的空闲块首地址。

5. 第四种情况，前后都有空闲块。将 size 更新为此空闲块的大小和前后两个空闲块的大小之和，重新设置空闲块的头部和脚部信息，并且使空闲块指针指向向前的空闲块首地址。

要点分析：

合并空闲块时除了需要更新空闲块的大小和头部和脚部存储的信息之外，如果当前块之前有空闲块还需要更新指向空闲块的指针。为了保证能够减少碎片的产生，在每次释放一个已分配块之后都需要调用 coalesce 函数进行合并空闲块。

第 4 章测试

总分 10 分

4.1 测试方法与测试结果(3 分)

生成可执行评测程序文件：
linux>make
评测方法：
./mdriver -a v -t traces
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 显式帮助信息
-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
测试结果：

```
yaoshunyu@yaoshunyu-virtual-machine:~/hitics/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   99%    5694  0.006938   821
 1       yes   99%    5848  0.005932   986
 2       yes   99%    6648  0.009995   665
 3       yes  100%    5380  0.007951   677
 4       yes   66%   14400  0.000077185806
 5       yes   92%    4800  0.006301   762
 6       yes   92%    4800  0.005776   831
 7       yes   55%   12000  0.135820    88
 8       yes   51%   24000  0.256733    93
 9       yes   27%   14401  0.058377   247
10       yes   34%   14401  0.002027  7104
Total          74%  112372  0.495926   227

Perf index = 44 (util) + 15 (thru) = 60/100
```

4.2 测试结果分析与评价（3 分）

使用了隐式空闲链表和首次适配的方法，用例可以全部通过，但性能不如显式空闲链表或红黑树。使用隐式链表，其首次适配的事件是块总数的线性时间，包括已分配的块和空闲块。显然已分配的块可以不用加入适配的计算，因为它已经被分配过了，只会造成额外的时间开销。并且使用的合并策略是立即合并，每次

释放一个块时都会调用合并的函数，这也拉低了程序的性能。而显式链表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，当大量内存被占用时块得多。

4.4 性能瓶颈与改进方法分析（4 分）

使用隐式空闲链表，首次适配的分配时间是块总数的线性时间，会收到已分配块的影响。如果使用显式空闲链表，它将所有的空闲块显式地连接起来，这样首次适配的分配时间就降到空闲块的线性时间。如果已分配块的个数较大，分配速度会有较大的差别。如果使用分离的空闲链表，即将大小相近的空闲块装入一个链表内，这样分配的时候大小会更适合，可以在一定程度上减少碎片的产生。另外如果使用红黑树来组织空闲块，首次适配会以关于空闲块的 log 时间进行分配，效率上又有了较大的优化。其次如果使用延时合并，如为 malloc 扫描空闲链表时合并或外部碎片达到阈值时合并，都会对释放的性能有所提高。

第 5 章 总结

5.1 请总结本次实验的收获

明白了动态内存分配的原理，了解了一些优化方法。

5.2 请给出对本次实验内容的建议

优化方案感觉难度较大，临近期末，很难抽出足够的时间进行优化

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.