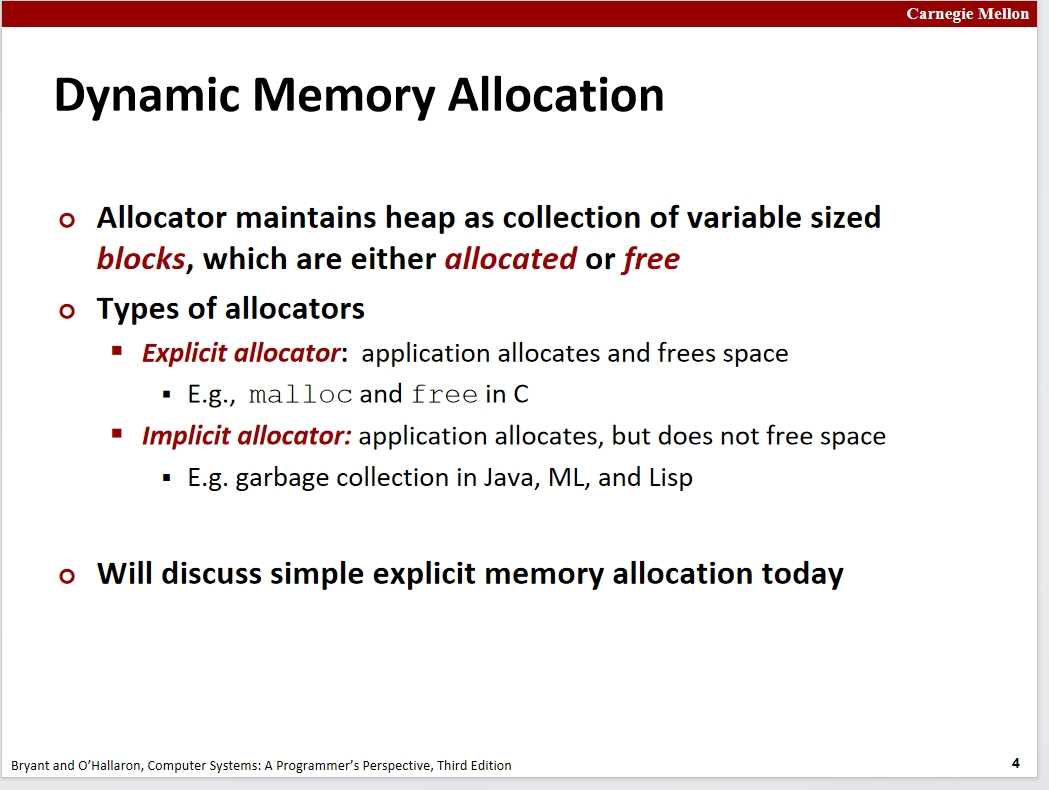
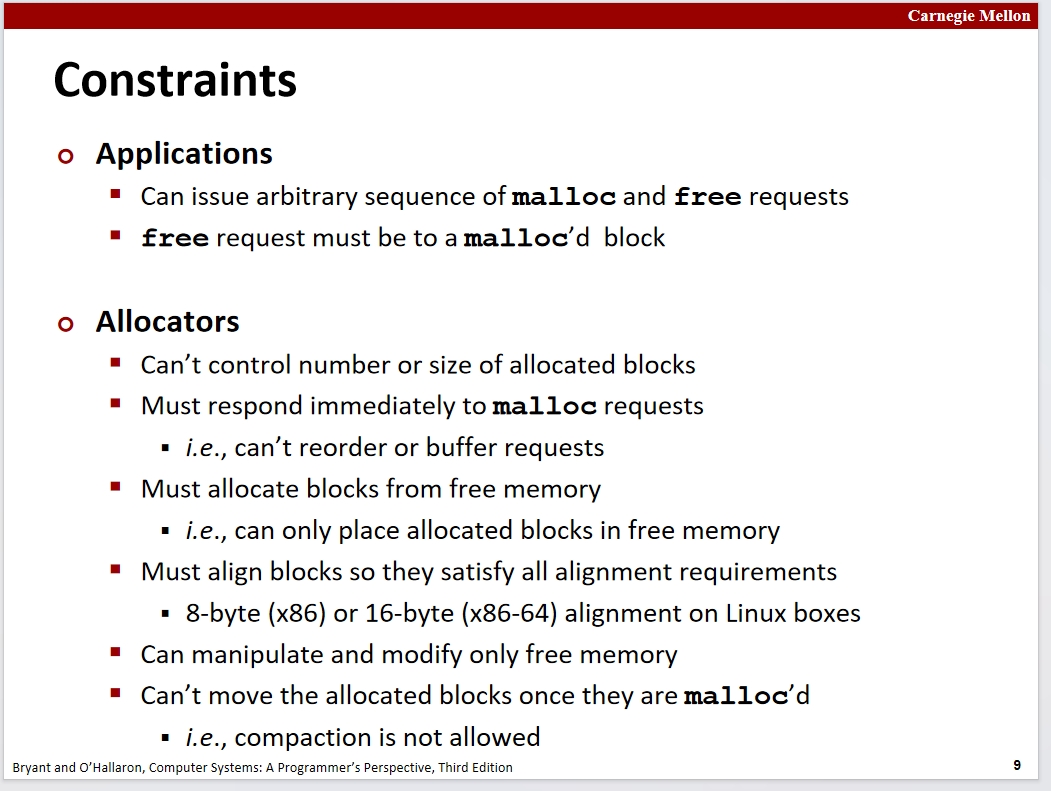
**动态内存管理，起动**

动态储存分配器的基本思想是应用程序通过使用这个分配器去操控虚拟内存，去构造，分配以及释放虚拟内存器片。就比如malloc和new等。这个储存器在堆区中被维护，这也是为什么这个我们动态分配的内存会是在堆区的原因。  


首先。对于分配器，其一般将堆区维护为一些连续的块集合。这些块可以被分配和释放。被分配意味着当前块正在被使用，空闲意味着这个块可以被应用程序所使用。

存在俩种类型的分配器，一种是像c中的malloc的显示内存分配和释放，这种情况下系统不会释放任何用户分配的内存，必须对这些手动分配的内存进行严格的管理。还有一种就是像Java等语言的隐式内存处理。在这种处理模式下，用户负责显式的分配内存，而释放内存这方面的操作将会是系统自动完成的。

接下来介绍下分配的几个约束



分配器可能会将已分配的内存块与空闲块进行任意组合。因此无法预测这个分配器到底最后分配的是堆区的哪些内存。

想要释放一个内存，就必须使用之前分配内存时返回的指针进行定位。等等。像malloc这样的显示分配器会在很多不同的约束下运行，这些分配器无法控制分配块的大小和数量，这是因为malloc执行的必须得是之前程序所指定的，而不会是系统自动指定的。

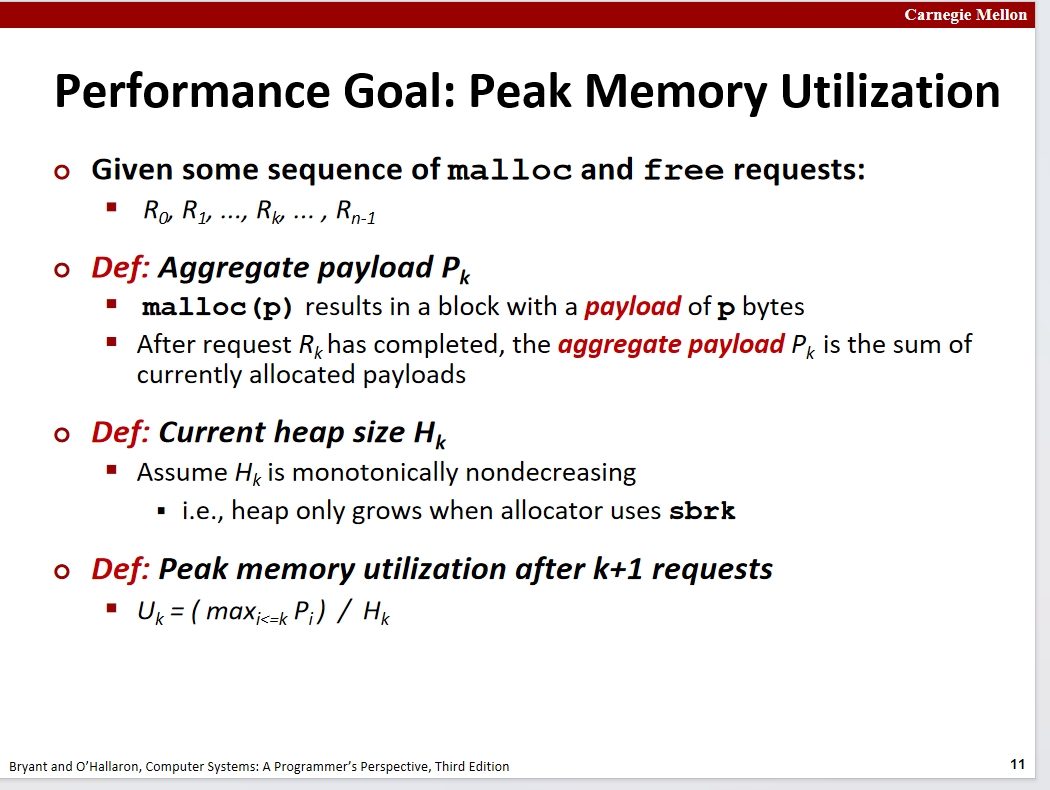
当我们的分配器被调用时，这个分配器必须立刻响应处理请求并返回。种种约束导致分配器其实是无法移动块的，也就无法通过移动块来实现碎片化内存的利用。

接下来看几个对于分配器的效率衡量指标

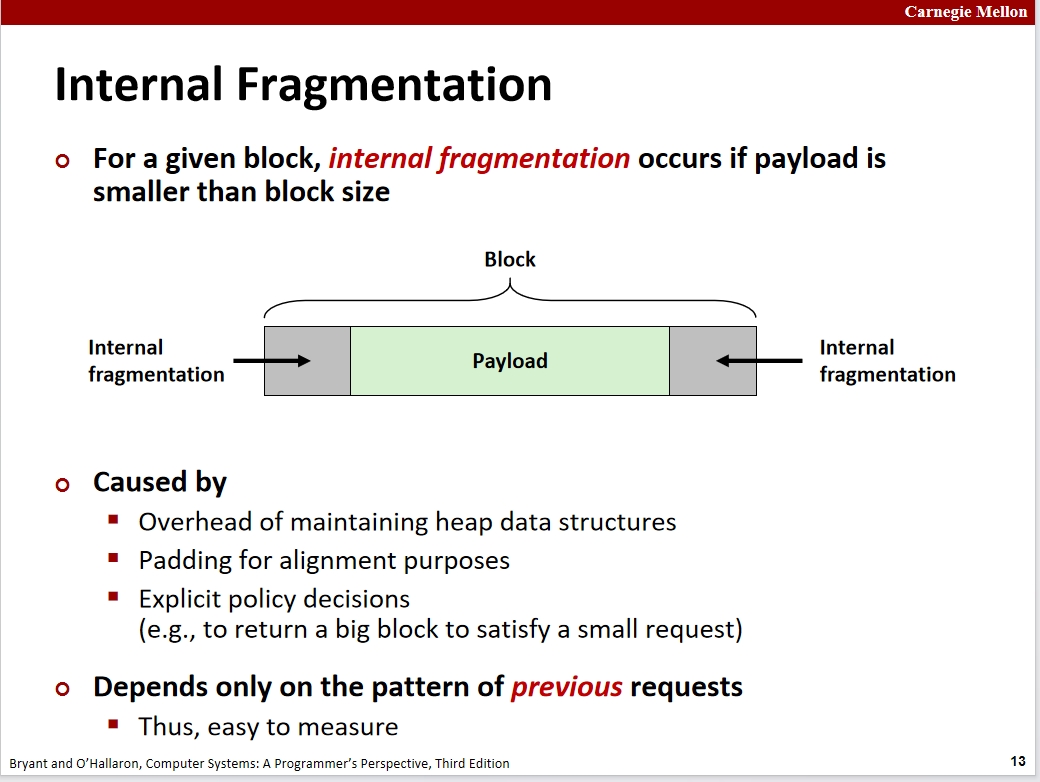
1~吞吐量(throughput)这个吞吐量就是计算一秒内分配器处理内存请求(malloc和new)等的数量，这个指标只是衡量malloc等对应的操作处理来自应用程序的请求的效率

2~峰值内存利用效率。这是衡量有多少有效空间的标准。衡量在全部使用内存和有效内存上的效率差异。当我们使用malloc去请求一个类似10内存的块时，这个10内存的块就是我们的有效内存。这个10内存的块对应的大块中的其他内容都将会是开销。

当我们进行一系列的内存分配操作时，所有的有效内存的总和就是我们的有效负载。剩下的信息阅读下图和书。



事实上，在进行内存分配时，我们通常会导致一系列的碎片，毕竟分配器是按块来进行识别的。对于分配器所导致的碎片，一般有俩种情况，一个个来看。



这里先分析下**内部碎片**。一般来说，分配器在分配内存时，都是以块为单位的。当我们分配的有效负荷小于块大小时，会发生内部碎片。这可能是由块中的填充或者分配器给块分配的某种数据结构导致的。

首先对于一个数据的内存分配请求，系统会先将这个数据进行一波系统的字节对齐，这里对齐的内存损耗也会是我们的一块内部碎片。除此之外，一个块中还需要一些数据结构来储存这个块的大小等数据信息，这个也会占用一定的内存空间，由于这些块的元数据不是我们想要的但是无法避免的，故这里也被视为一种内部碎片。

内存分配器存在一系列的问题，接下来一个个来分析解决。

1~分配器是怎么知道一个free是要释放掉多少内存的

其中一种解决方法就是，增加一个用来标识当前正在使用的块的大小。就比如你想要分配一个4字长度大小的块，分配其会去查找一个大小为5的块，并将这个块的首位作为一个块大小的标识符。因此，对于一个由n个有效载荷字组成的块，至少有n个有效载荷字。块的头部字标识该块的大小。而且在这种情况下会返回的指针是一个指向真正有效负荷字的指针。

接下来了解分配器是怎么了解跟踪空闲块的。

最简单的方法是使用一个隐式列表。这种列表是指在堆区中每个无论是否被使用的块的前面都放置一个头部。由此，我们可以在堆区的入口处访问这个表，并以此来进行堆区中空闲区域地址及大小的计算，对各个位置的堆的块的访问使用等等。

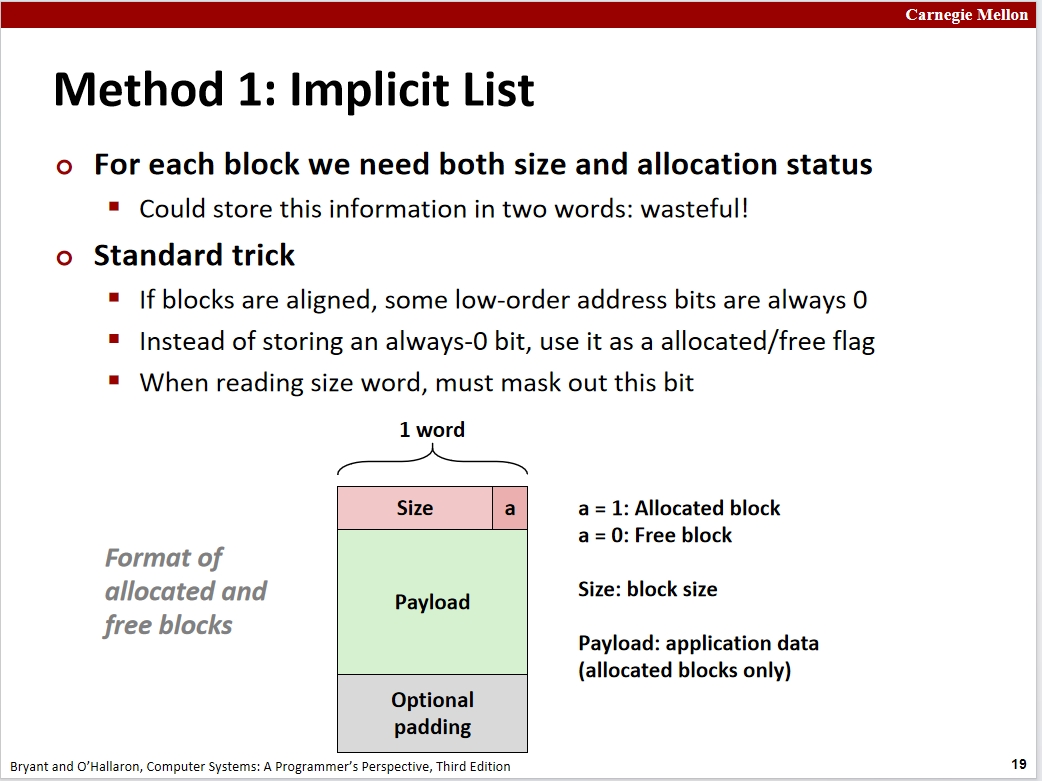
在这种情况下，我们并没有一个直接的空闲区域的列表，只是对于我们可以通过遍历整个链表来进行识别，只需要忽略那些已经识别的块就行了，这是其被称为隐式列表的原因。但问题就是这种方式需要遍历整个链表去计算和查找最合适的区域。

进一步的，有一种专门用来管理空闲块的链表，这种链表只使用这些块中空闲块的头部来构建链表，使得分配器可以高效的去查询需要的空闲块。

除此之外，我们对于堆区的管理还可以不只局限于一个列表，我们可以使用多个空闲列表。每个空闲列表专门指定一种特殊信息的块，就比如指定一个特定大小的块或者特定大小范围内的块。对于这种情况，就比如我们有一个管理0-32大小块的链表1,32-64大小的链表2。我们如果请求一个50字节的块，分配其回去链表2中进行查找，假设存在一个60大小的块，那么这个块中的50字大小的空间将被分配。而且接下来的10字节的大小的块将被重新归入到链表1中进行管理。

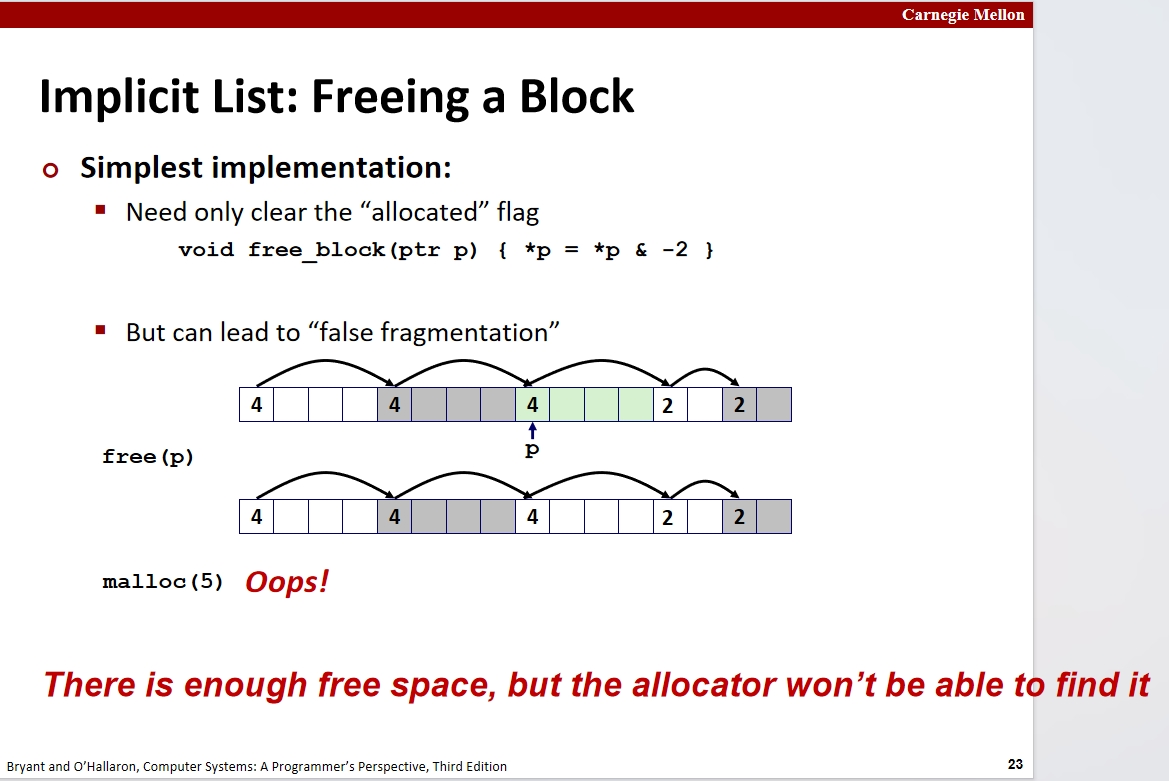
对于这种情况的处理，我们一般可能会使用某种树来进行管理，就比如红黑树能够进行高效的插入删除操作，并且查询效率也不低。

接下来查看对于块的空间分配策略优化。



对于一个块的头部，我们通常需要一些信息来进行这个块的属性的标识。在对于块的分配中，系统一般是采取一种字节对齐的规范的。也就是说，对于每一个块的起始位置，都会是这个规范的大小的整数倍。就比如我们如果使用8字节对齐的规范。8的二进制数为1000,16的二进制数为10000等等。可以看到，对于这些数，其本身有一定的特性，就是它的01串尾部一般都是一个不会改变的。这里就都是0的情况。在这种情况下，我们考虑用这些位来进行一些属性的标识。就比如我们使用这个01串的最后一个位来进行这个块是否已经被分配的标识。假设用1来标识已经被使用，那么这个头部01串可能就被设置为1001。

相应的，由于我们修改了这个位来进行一些属性的标识，所以在之后的可能对于这个快的信息的读取中，就比如对于这个块的大小的读取，我们需要屏蔽掉我们修改的这个位的影响，这个不难想到用位操作来进行。使用一个合适的掩码将末位的位设置为0，就比如末位&0等操作可以实现对这种属性位的屏蔽并读取想要的信息。



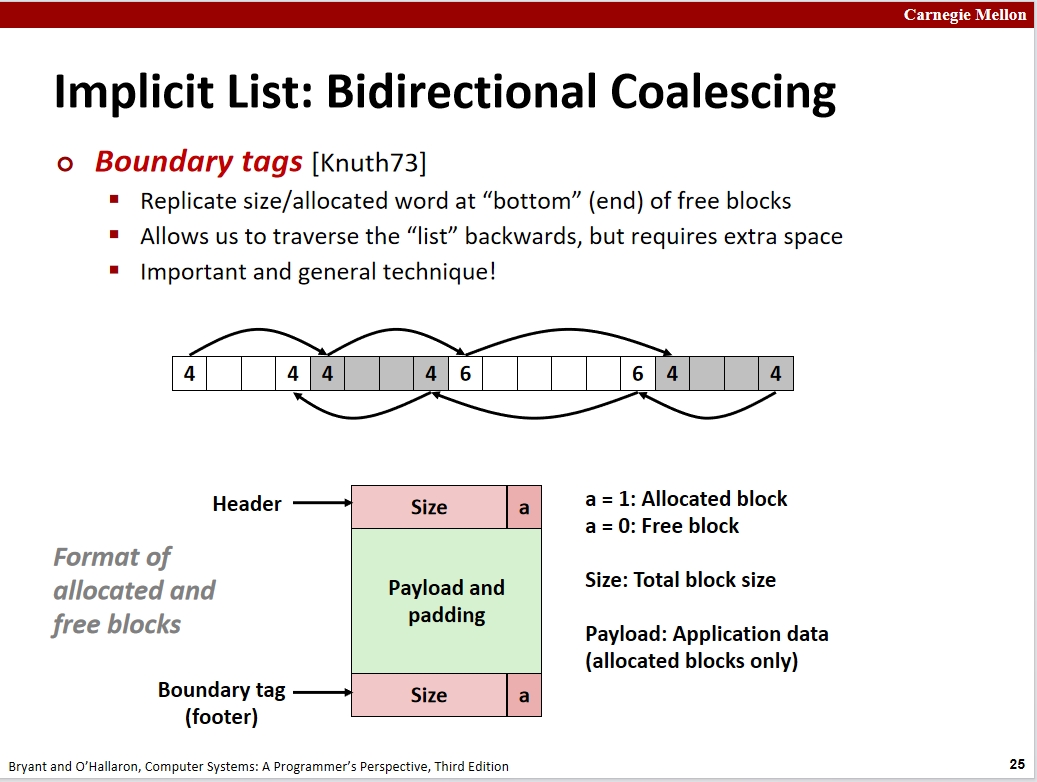
跳过一些东西，来看一些释放块需要注意的

就比如这里的，如果我们已经有了如上图的内存分配情况，且此时我们需要去把那块绿的内存给释放掉，最简单的自然是直接将这个块的有效位置为0.但是这会导致一些问题，如上图，最明显的就是即使释放后的存在连续的俩个空块，但是分配器并不会将他们视为一个大块，当我们想要分配一个大小为5的块时，此时堆中是没有符合大小的。这种情况下会导致严重的内存浪费。

对于一个优秀的分配器，我们不能允许连续空闲块这种情况的出现。因此我们需要一定的措施来进行补救。最简单的自然就是对于这些内存块的合并。当我们释放掉一个内存块时，我们去检查下一个内存块的状态，如果是一个空闲的块，就对我们这前面的块大小属性进行修改，并修改对应的链表属性等。

但我们也需要注意前一块是空闲块的情况。这种就需要我们在遍历的时候去记录上一个块的位置了。如果使用双向链表，这个开销相对于记录一个遍历过程数据来说会相当的大，毕竟你一般不会进行频繁的堆区内存操作。

但是事实上这种在遍历时去记录数据的情况的开销其实也很大，现在的话有一种相当高明的技术。就是我们将一个块的头部复制到块的尾部。需要注意的是，我们的头部属性的指针跨越的长度其实就是我们这个块的大小。那么对于我们尾部的指针会跨越的长度也会是块的大小，这也就意味着，这个尾部指针将会像头部指针指向下一个块一样去指向前一个块的末位置。在这种架构下，我们就可以在常数时间下去访问前后节点，且这种开销相对于前俩种事可以接受的。



除此之外，还有一种很有趣的策略，我们可以通过一些方法来减少这些尾部标签的使用，试想我们在哪些区域不需要使用这些标签呢。对于那些已经分配的块，我们是不需要去分配标签的。但是我们还是需要一些信息去对这种状态进行标识的。有没有一种方法对于前面的块的这种状态进行标识呢？

别忘了我们前面用来标识这个块的有效性用的技术，我们利用了由于字节对齐一个块的大小表示字中的未位0.一般来说，这种末位0至少有3-4位，我们完全可以利用这种来进行标识我们的前一个块的状态。就比如我们使用1来代表前一个块正在被使用，那么我们就不必去考虑这前一个块。但是，当我们这个标识位标识前一个块时空闲的时候，我们就需要去合并了。此时这个空闲块是一定含有一个尾部信息块的，我们只需要直接去偏移查找这个尾部块的信息即可获取这个块的大小进行合并。

这种技术使用的技术基础是位运算远远快于其他的运算。毕竟相对于这种方法，其他几种也只是偏移去查找一定的信息实现的。

在这种技术中，我们减少了对于尾部标签的使用，使得一个已分配的内存可以使用原本需要用来标识信息的尾部块。但在碰到这个的时候我在想有没有可能同时也对下一个块进行标识。

但是我忽略了一些问题。本质上，我们这里对前一个块进行标识是为了减少一些尾部标签页的使用。如果我们使用一个位来进行标识，那么这个最明显的难道就能优化掉我们的尾信息了吗。不能，为什么，因为有了这个尾标签页我们才能去有效的去找到我们前一个空闲块的起始位置。如果我们使用位来优化，即使我们能够优化掉空闲块的尾标签，但是我们还需要一些额外的技术来进行对于空闲块的起始位置的寻找，但是又有多少技术可以快过通过对块其实位置偏移来找到前一个空闲块的大小来计算合并呢？除此之外，如果我们需要记录后一个块的状态，这个可能是一个连锁的块，对一个块的状态进行修改时，很可能导致一系列的块的状态都需要进行修改，这就需要极其复杂的实现逻辑。

总的来说，这种同时对下一个块的标识其实目前来说还是一个吃力不讨好的实现。目前来看，尾部标签配合前块状态标识的技术，仍是当下内存分配器中的**黄金组合**。