Malloc lab实验报告

10215501412 彭一珅

1. 实验目的

本实验目的是实现一个分配器，通过改变块格式、空闲链表格式、放置、分割、合并策略，提升分配器的吞吐率throughput和空间利用率utility，使分配器的性能趋于最优。

1. 实验分析

分配器的空间利用率，主要由内部碎片和外部碎片影响。内部碎片在一个已分配块比有效载荷大时发生。比如为满足对齐条件，会将原本申请的size扩大成满足8倍数的asize；同时，隐式空闲链表的头部和脚部，显示空闲链表的后继和前驱指针，简单分离存储中不被分割的块等也会造成内部碎片，引起空间利用率的下降。外部碎片是在空闲内存可以分配一个块，却没有一个单独的空闲块可以用来分配而造成。主要原因是链表中有大量的小空闲块。

分配器的吞吐率，主要由malloc时查找find，放置place合适的空闲块这两个动作的时间复杂度，以及free时对块的合并策略的时间复杂度决定。

为了提高程序的性能，课本中有多种策略可供选择。

在块的格式方面，可以选择只有一个头部的块、具有头部和脚部的块，以及包含头部、脚部、前驱后继指针的块，还有使得已分配的块不含脚部，未分配的块含有脚部的方法。

在合并策略方面，为了解决假碎片问题，可以实施在free时立即合并和推迟到在malloc时合并两种策略。

在分割策略方面，可以直接利用特定大小的块，也可以分割出块的一部分，另一部分划分为新的空闲块。

在放置策略方面，可以使用后进先出LIFO的顺序，用首次适配维护链表，也可以按照地址顺序维护链表，运用首次适配、下次适配。

在空闲链表格式方面，有隐式空闲链表、显式空闲链表、分离空闲链表等方式可以实现。

1. 实验过程
2. 实现基本功能（隐式空闲链表+立即合并）

先根据课本上简单分配器实现的例子，基于隐式空闲链表，使用立即边界标记合并方式，实现一个分配器的基本功能。

Coalesce函数使用立即合并的策略，每次释放时都检查前后块是否可合并。

Find\_fit函数首次适配+按地址顺序遍历。

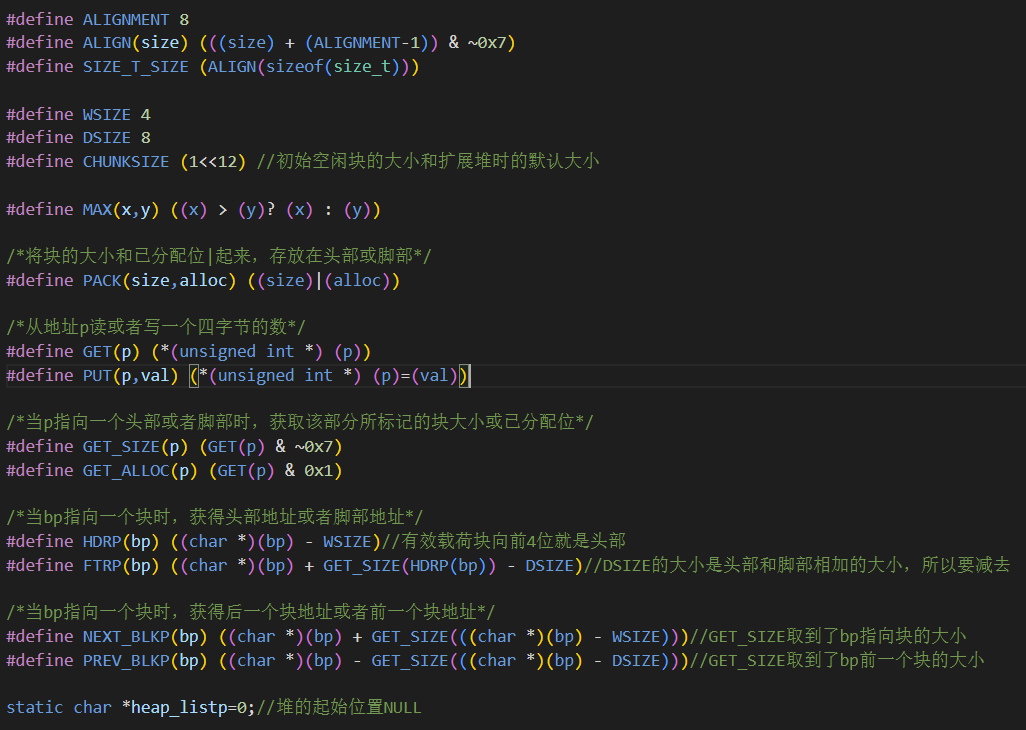
Place函数在切割出的空闲块大于最小块是选择分割，否则使用整个空闲块。

Init函数设置序言块，结尾块，并且把heap指针设置在要分配的空闲块中间，以便于extendheap。

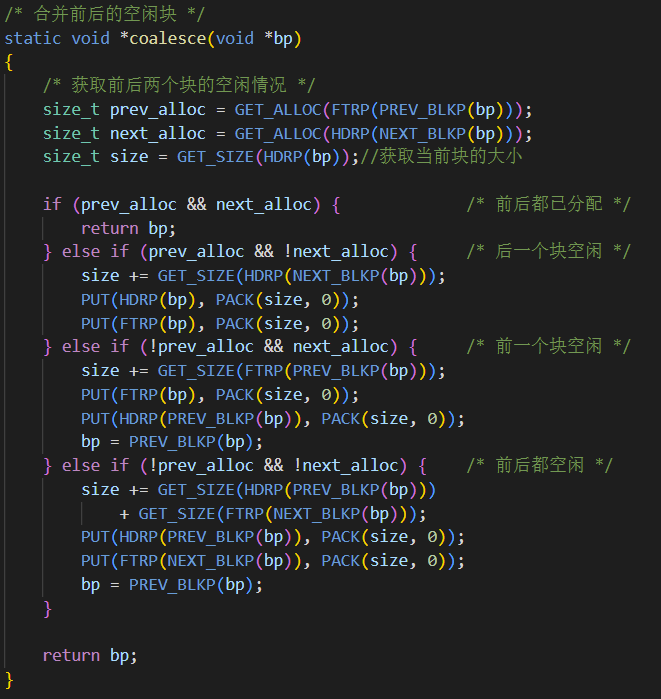
Realloc分为几种情况。第一，其中一个参数为0，则直接调用malloc或free，第二，分配的大小小于或等于指针指向块的大小，则直接place，第三，分配的大小大于指针指向块的大小，这里考虑后一个块是否空闲，如果空闲，则合并两个块，如果不空闲，则malloc并把旧的内容copy到新分配的块中，释放原块。

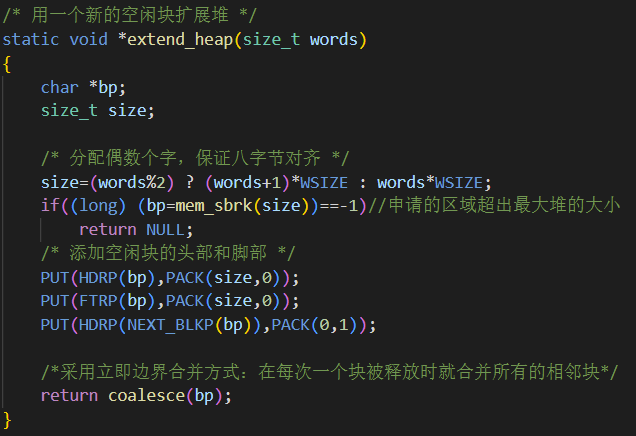
这段代码是参考课本上的实现过程，具体如下：

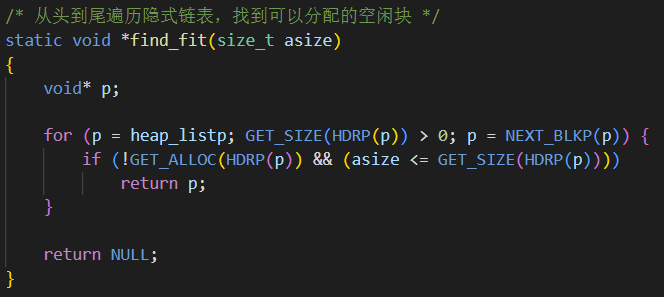
宏定义部分



静态函数部分

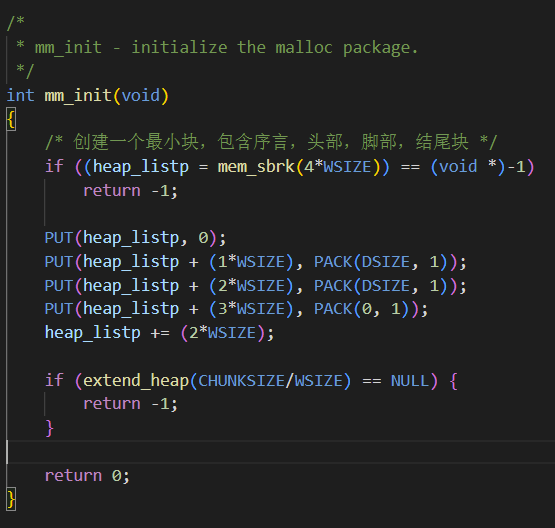


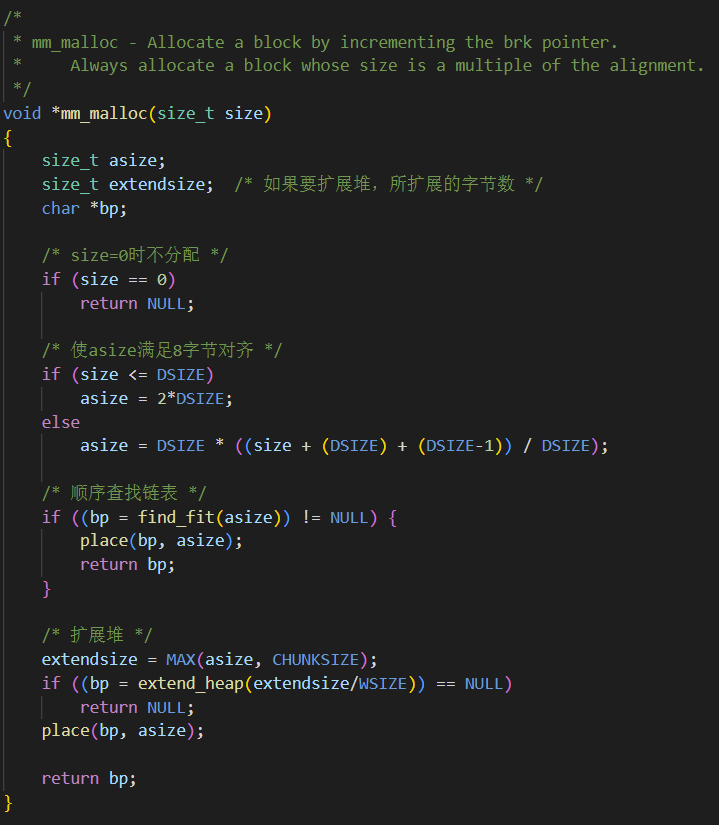


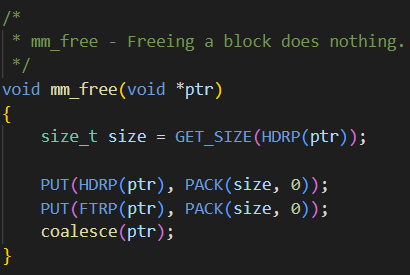


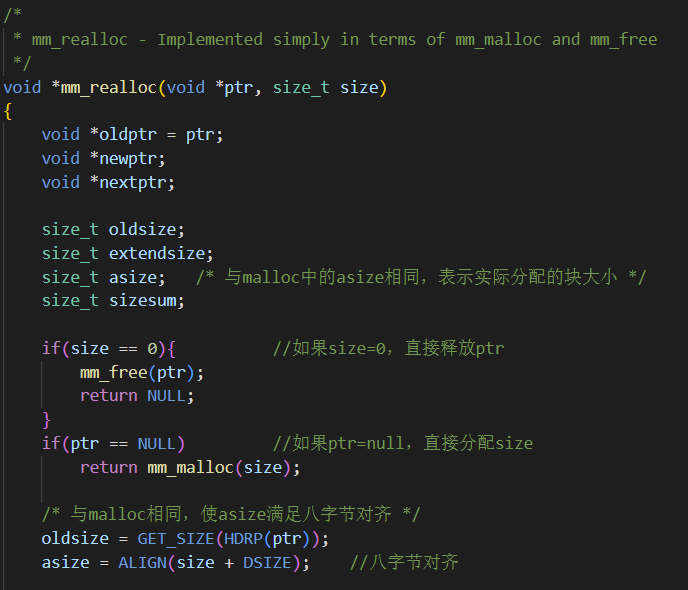


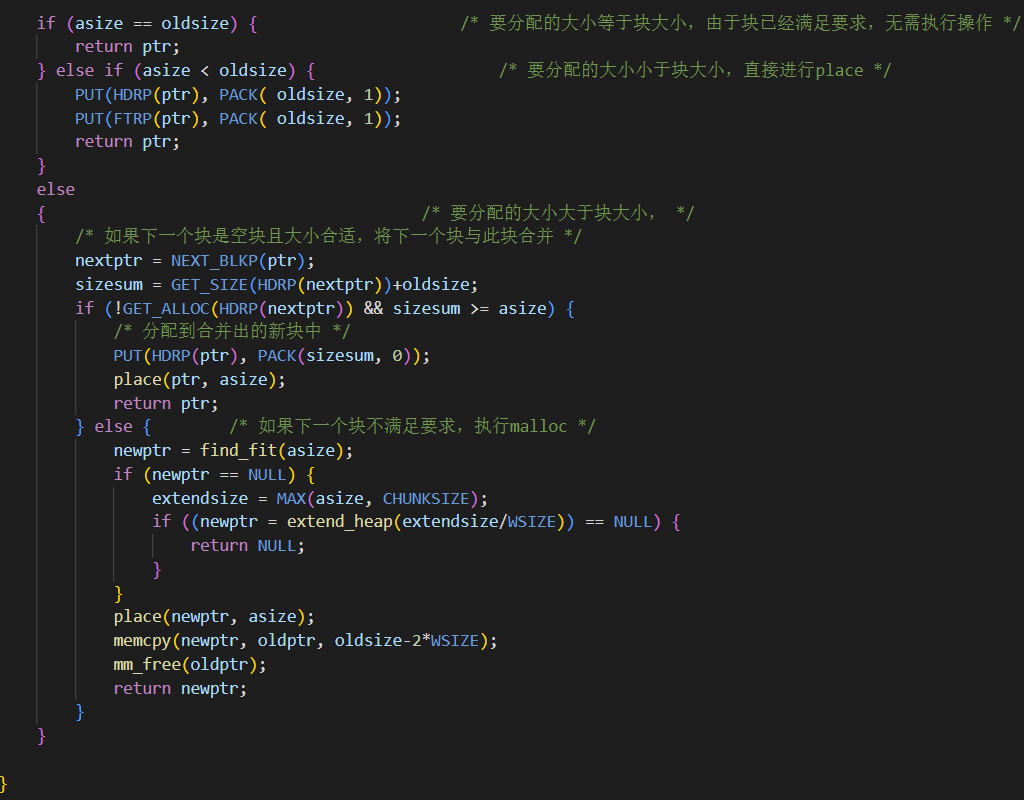
正文部分



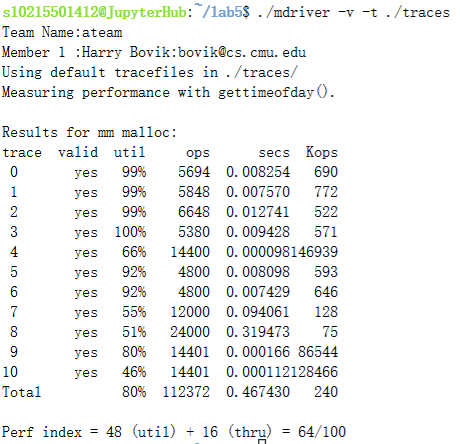








这段代码的运行结果如下，显然，这只是一个简单的实现，还没有进行优化。



可以看出，时间效率方面，这种实现不够快，主要原因在于遍历隐式链表需要很长时间，而且首次适配也使得每次查找空闲块需要查找得越来越远。

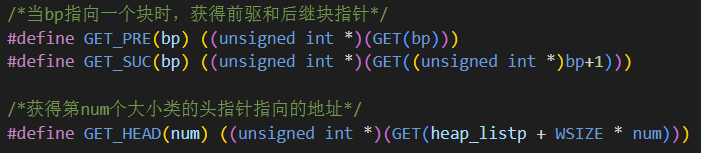
空间利用率方面，有的测试用例可以实现的比较完美，有的测试用例很差。

1. 对实验进行改进（分离链表）

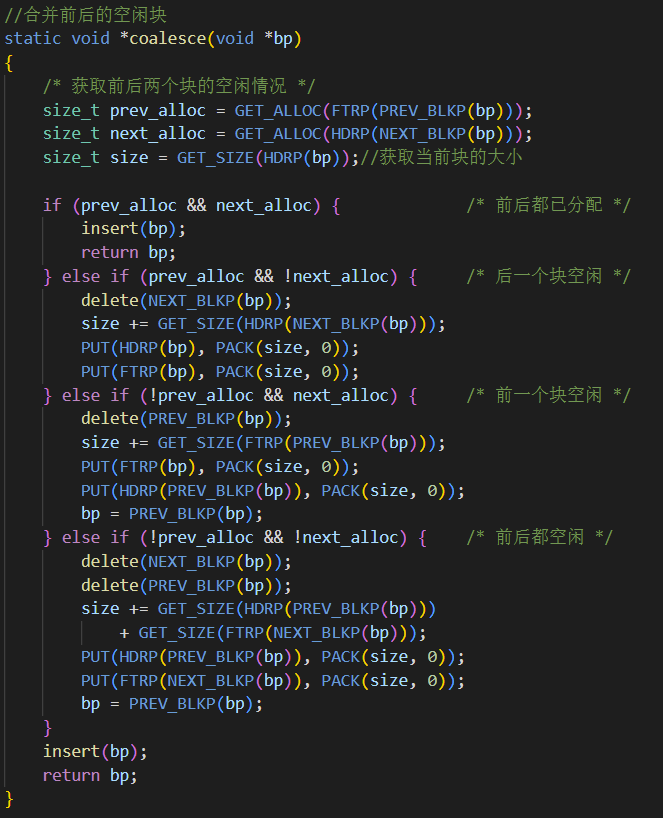
根据课本中的说明，分离链表具有较高的效率，于是当机立断改用分离链表，把原本的隐式空闲链表修改为数个大小类链表，增加了宏定义get\_pre和get\_suc，而get\_head是获得大小类的头指针。

这次改进与上次相对比，每个函数的改动不大，因为只是改成了分离链表，并没有对其他策略进行改动。

增加了宏定义：

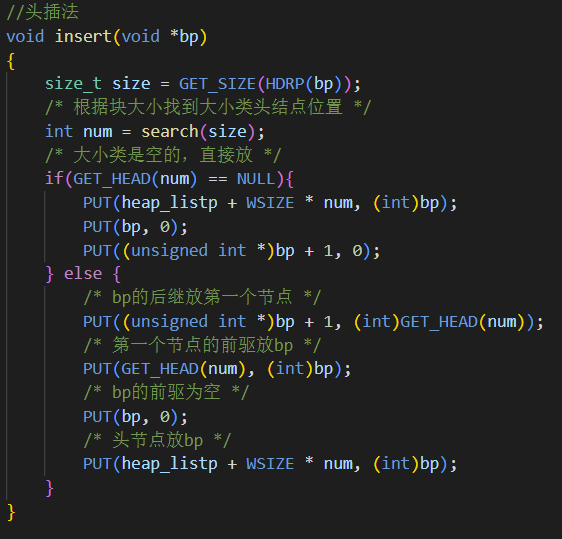


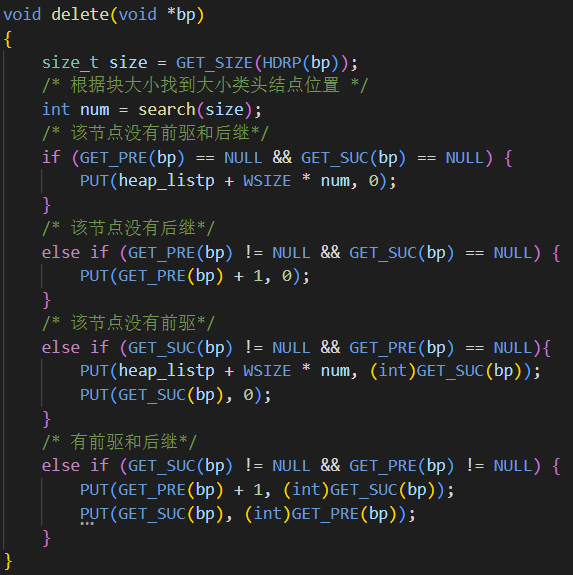
Coalesce函数：在原有的基础上加上了插入、删除节点的函数。



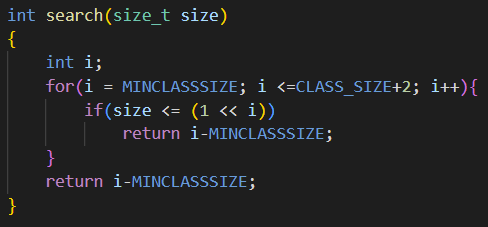
Extend\_heap函数没有改变，在此就不放代码了。

Insert、delete函数：此次修改最重要的两个函数，用来维护分离链表。Insert函数将指向的块插入到相应大小类的第一个位置，并更新指针。insert插入链表的都是空闲块，这种方式可以让首次适配的效率更高。Delete函数分为四种情况，没有前驱和后继，意味着这个节点是这个大小类惟一的节点，没有后继，意味着是大小类的最后一个节点，没有前驱，意味着是第一个节点，有前驱和后继，意味着它是中间的节点。

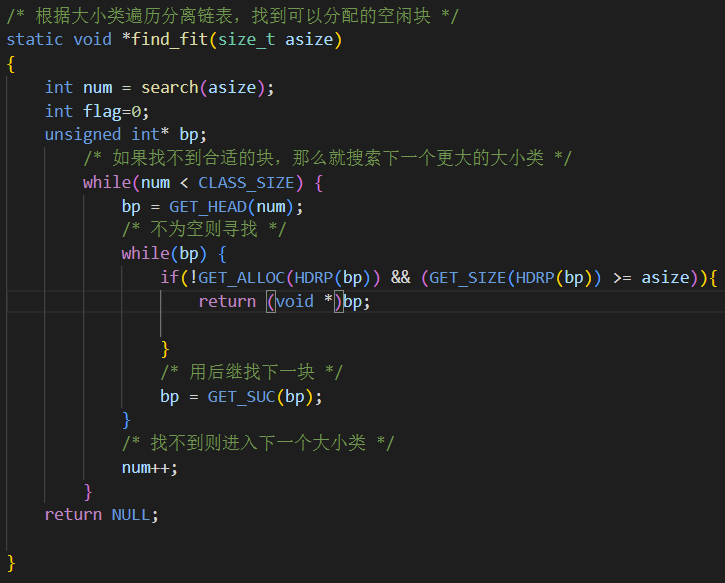




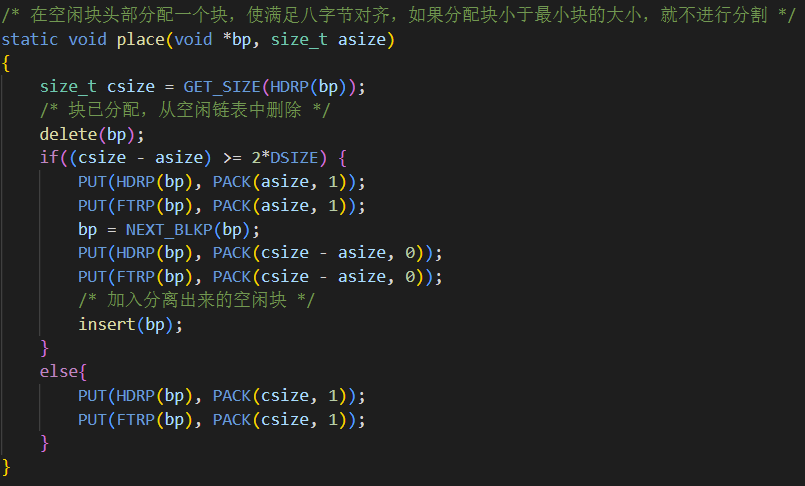
此外，还新增了一个search函数，这个函数可以通过给定的块大小找到它所属的大小类的编号，通过这个编号就可以定位到大小类的头指针。



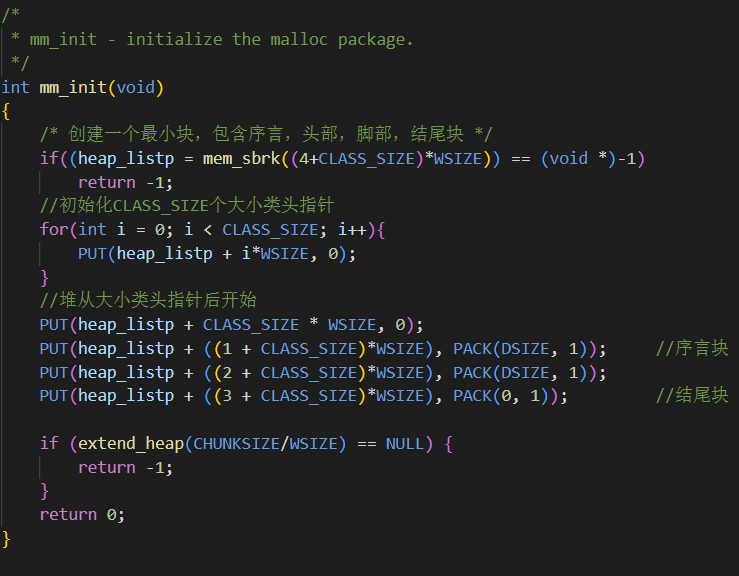
Find\_fit函数也经过了较大的修改，运用首次适配的方法，从最符合的大小类开始搜索，大大提高了找到空闲块的效率。



Place函数，几乎没有改动



Init函数，初始化了大小类头指针，把这些指针的位置设置在堆的前面。

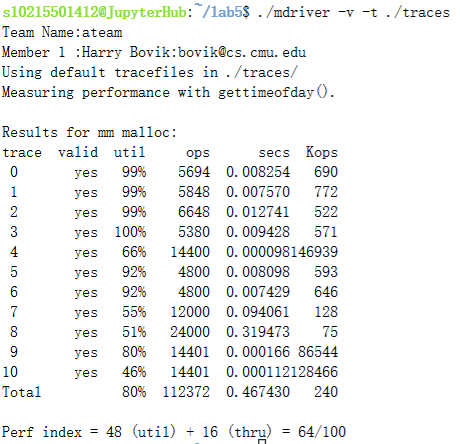
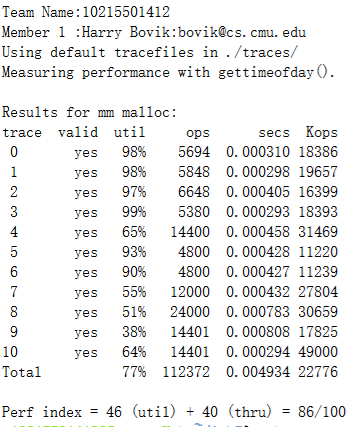


Malloc函数与free函数，没有改动，在此不做展示。

Realloc函数，仅有一处改动：在合并该块和下一个块时，要用delete将下一个块从其原本的链表中删除。

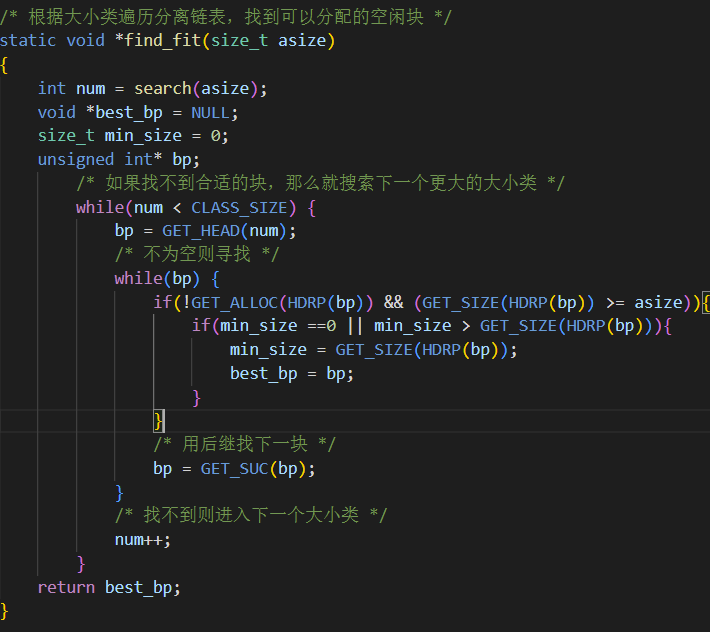


这段代码的运行结果如下：

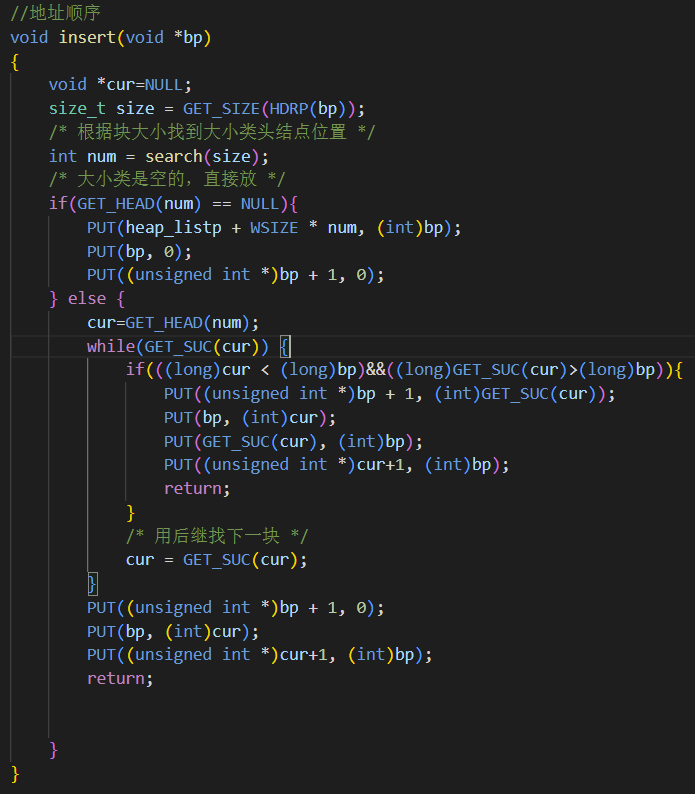


可以看到，分数有了很大提升，吞吐率由于find\_fit的改进，有了极为显著的提高，但是空间利用率甚至还不如原来的实现，第九个用例竟然有38%之少，这可能是因为分离链表产生了大量的外部碎片所致。为了牺牲一点吞吐率来提高空间利用率，我在这个基础上思考了很多小的改进，在此一一列举。

1. 下一次适配：结果有几个用例直接out of memory了。
2. 最佳适配：分数没有变化。效率虽然降低了，但是吞吐率没扣分，可是空间利用率也没有想象中的提升。



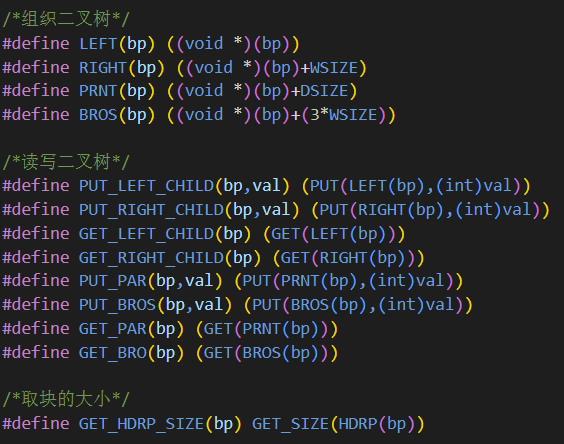
1. 按地址顺序维护链表：使每个块的前驱都是比它地址小的，后继都是比它地址大的块，虽然理论上可以增加空闲块相邻的概率，增加可合并的可能性，但是很遗憾，空间利用率也没有提升。后来经过思考这些策略只有在显示链表中应用才能发挥相应的作用。



1. 在分离链表上无论如何也无法提升分数了，痛定思痛，决定换用二叉树组织块。

该二叉树的基本思路是，相同大小的块形成兄弟链表，其他块根据大小组成简单二叉树。这样可以简化

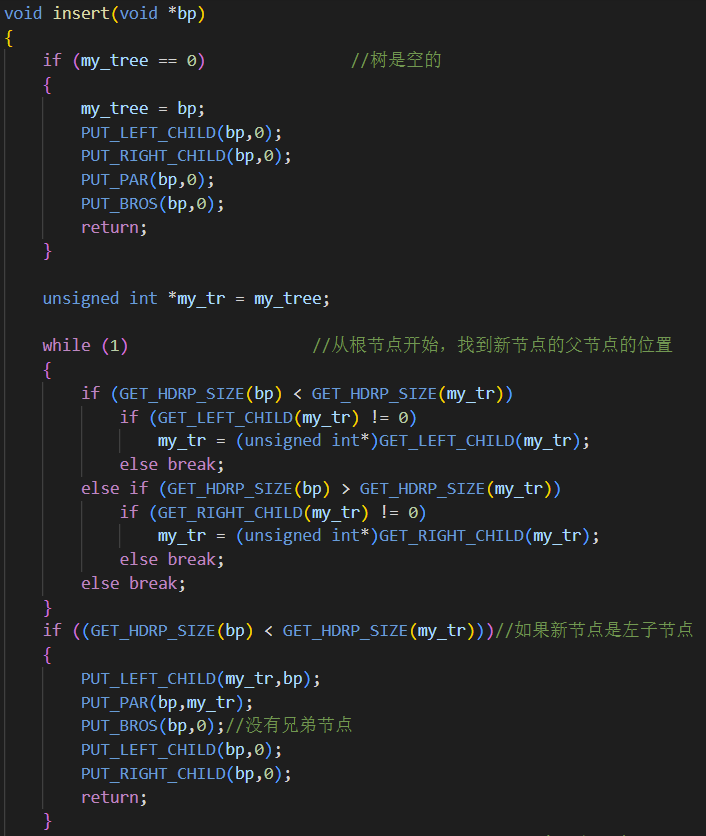
加入了宏定义：

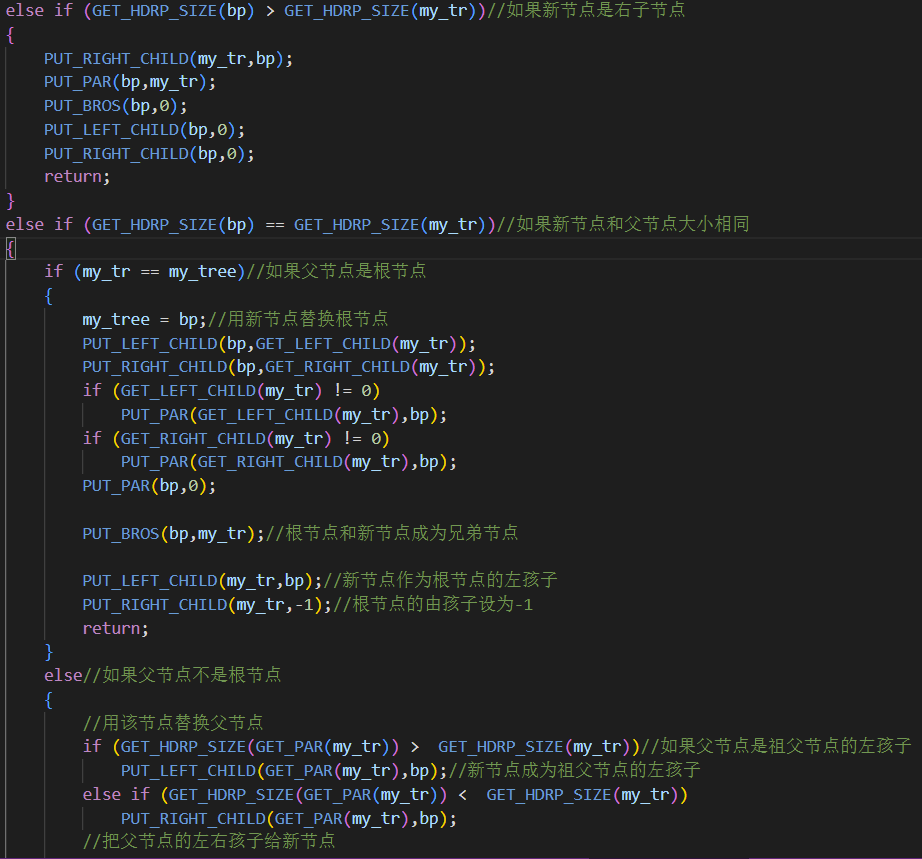


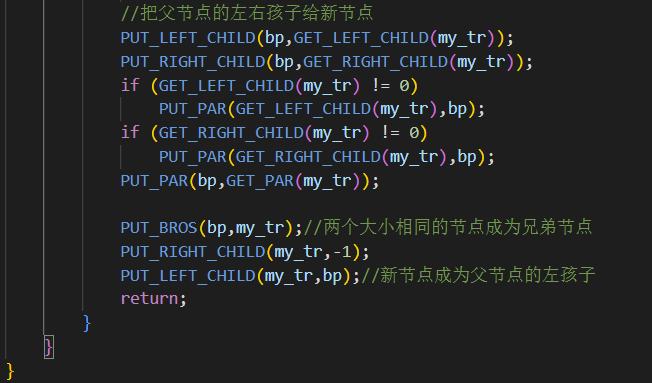
Coalesce函数在分离链表基础上没有改动。

重写了Insert、delete函数，实现二叉树节点的插入与删除。

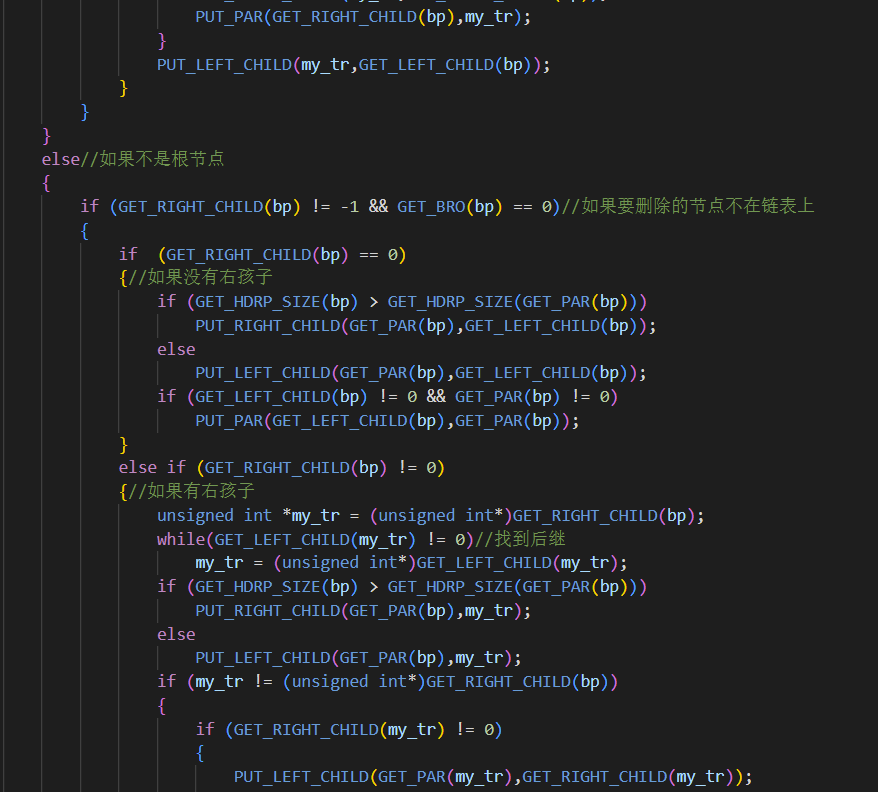
具体的实现过程在注释中已经写清，在此不多赘述。

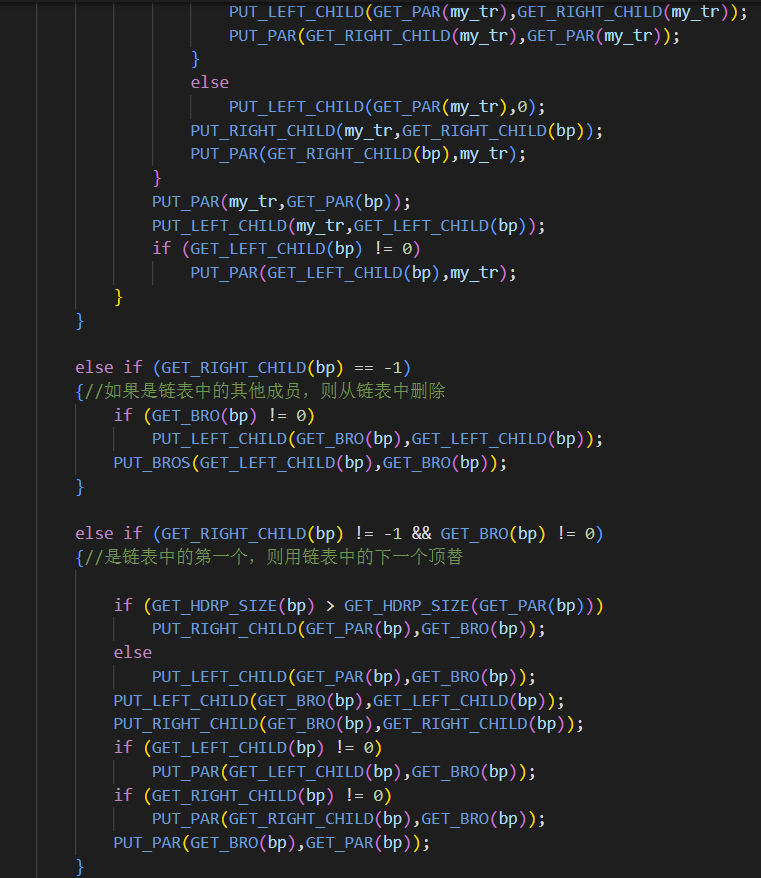




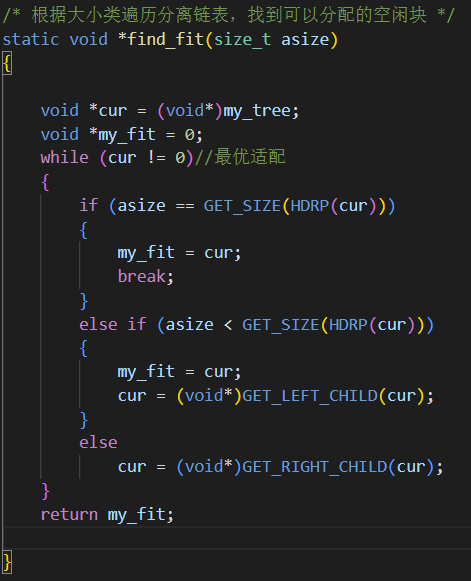






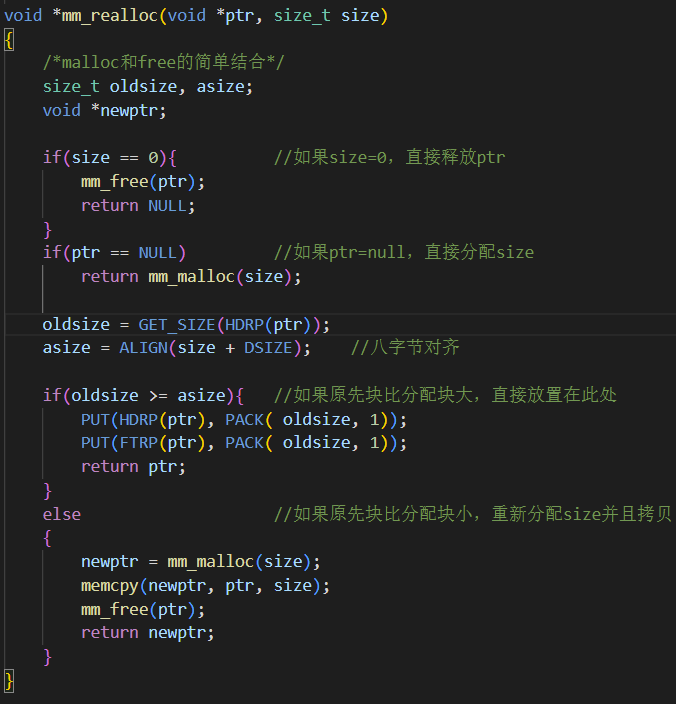


Find\_fit函数使用了最优适配，用以配合二叉树的查找功能，在较短的时间内实现。

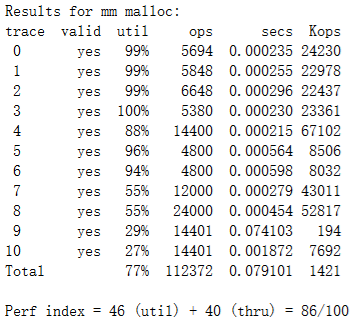


Place函数无改变，删除了search函数，init函数改动也很小，只把原先分配给头指针的空间省去了。另外，把extend\_heap当中words\*WSIZE的部分删去了，仅取距离words最近的8的倍数，这么一来可以减少每次堆扩充的大小，有利于空间利用率的提升。Malloc也没有改变。

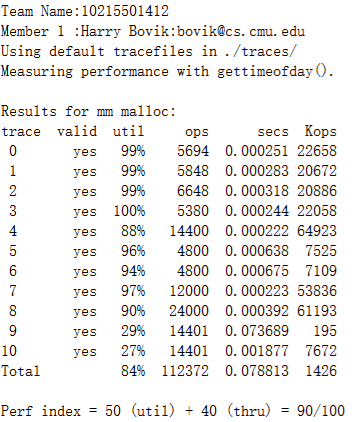
这次，realloc图省事运用了一种比较简单的方法，忽略下一个块的合并，如果指向的块大小不足以分配，直接进行malloc。



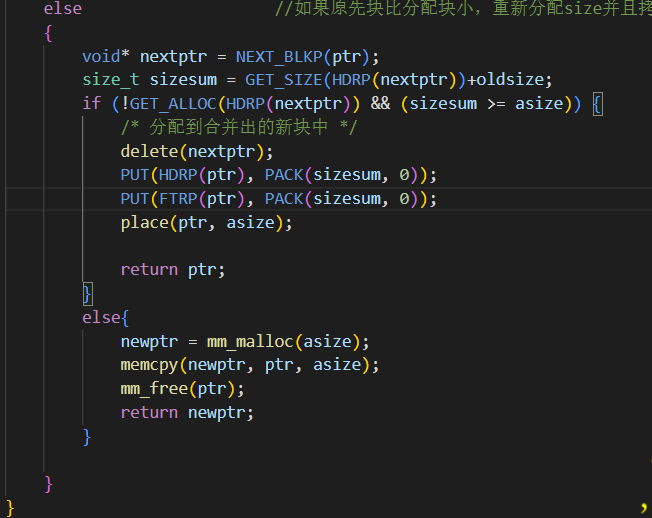
结果执行还是86分，非常头疼。

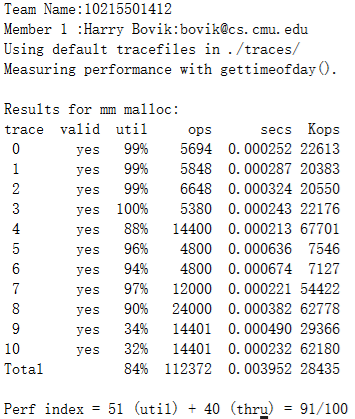


针对用例67进行了一番面向测试用例编程，最终得分90分，虽然胜之不武，但是这给了我继续前进的动力。



而对realloc改进之后，reallloc像之前的两步一样可以与下一个块合并了，此时分数达到了91分，提升了最后两个测试用例的空间利用率。





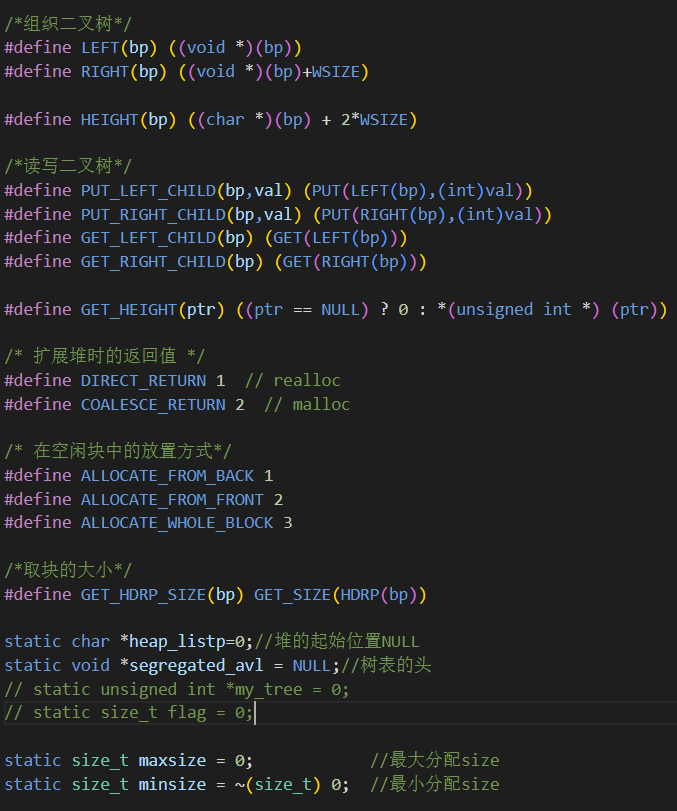
1. 二叉平衡树+分离链表

后来我又思考，如果将二叉搜索树和分离链表结合起来，是否能进一步提高分数呢？

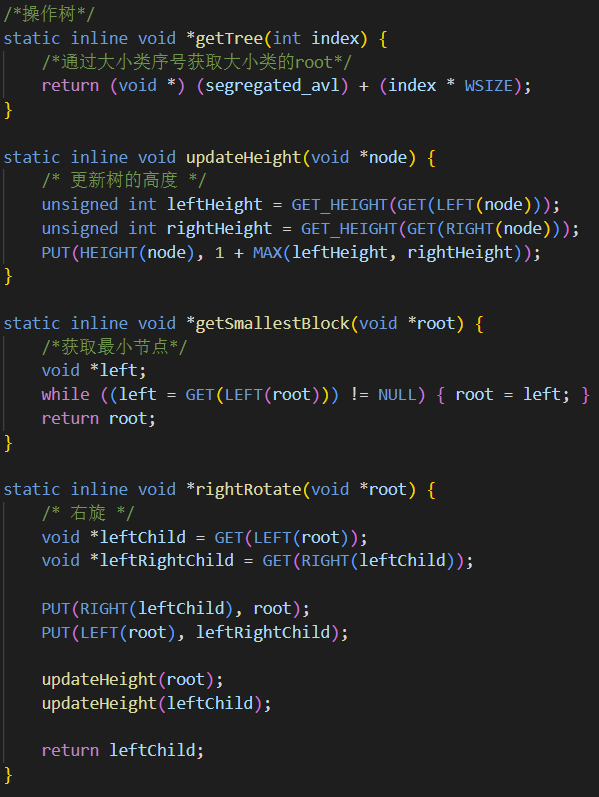
然后在网上看到类似答案得了很高的分数，参考了高手的代码，决定尝试一下这种实现。这次推倒重来，虽然保留了原先的几个基本函数名称，但为了实现avl树，代码被改的面目全非。

增加了宏定义和静态变量如下：

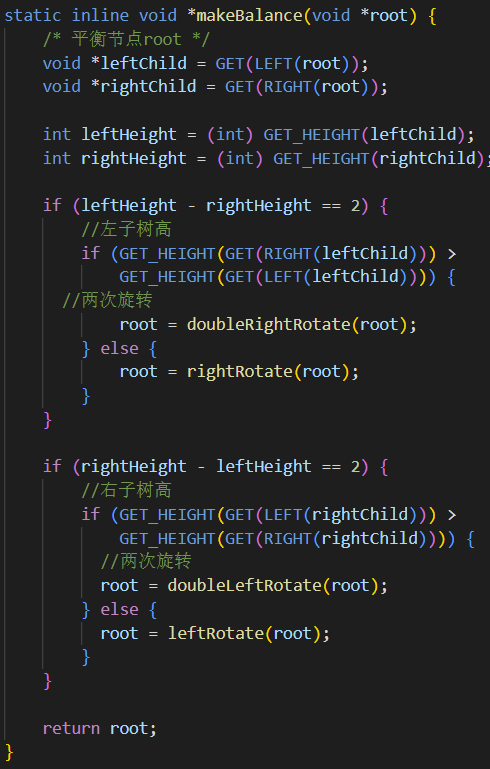
首先，给malloc和realloc实现了不同的堆的扩展方式。其次，在空闲块中的放置方式被分为从头开始、从尾部开始、放置整个块三种情况，增加了从尾放置的方式，有利于空闲块的合并。

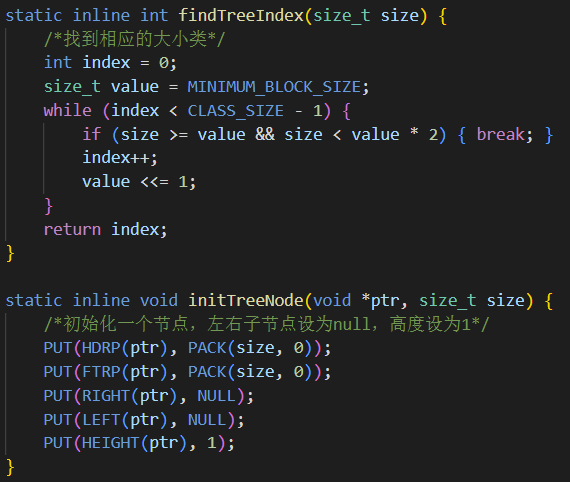


操作树的函数：

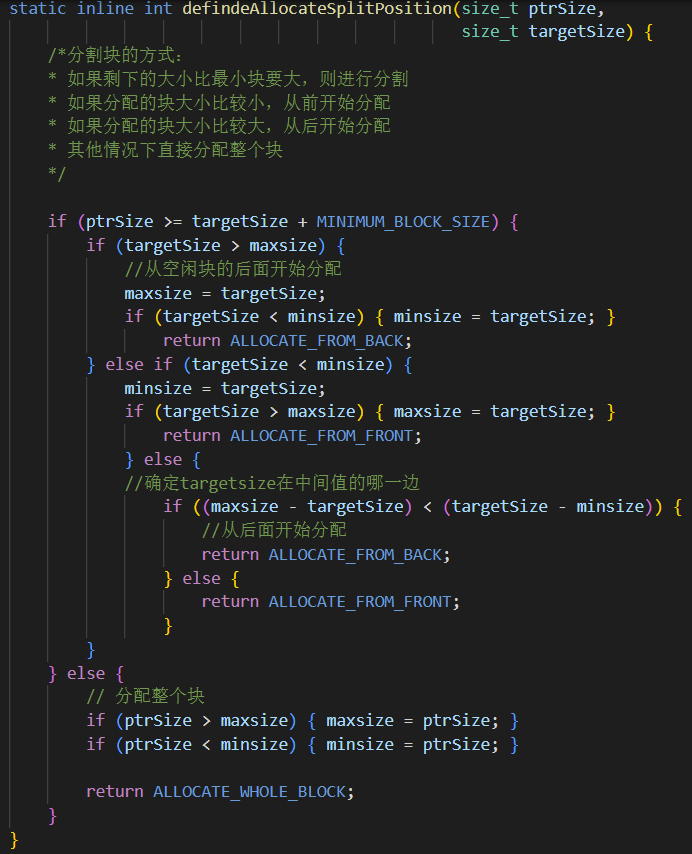




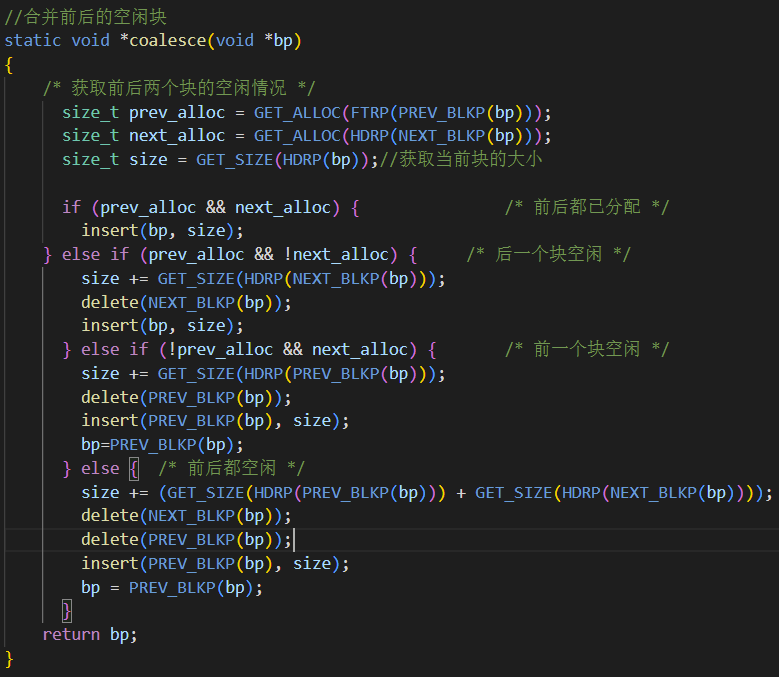




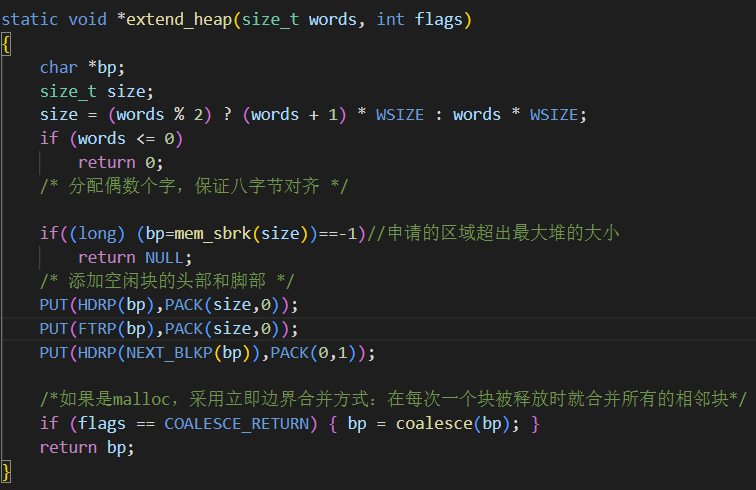
分割块的方式：



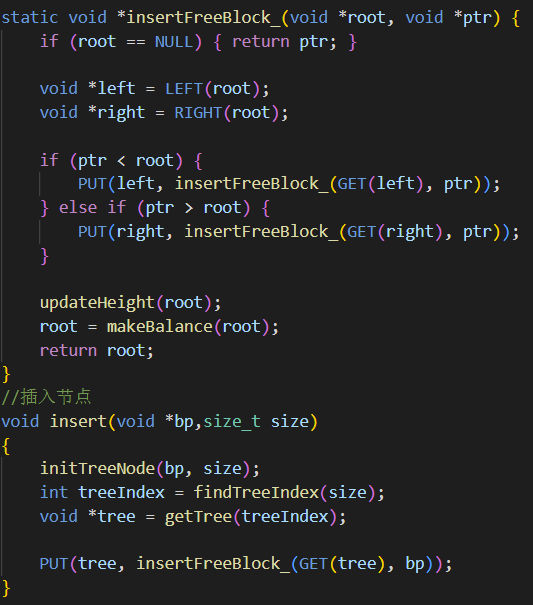
Coalesce函数还是相同的思路：



Extend\_heap也是没有太大改变：

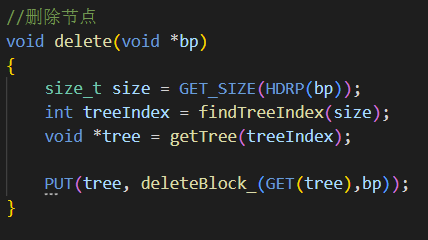


Insert这边用了一个递归实现，就是简单的对二叉树的插入以及makebalance来保持树两边高度差不超过2。

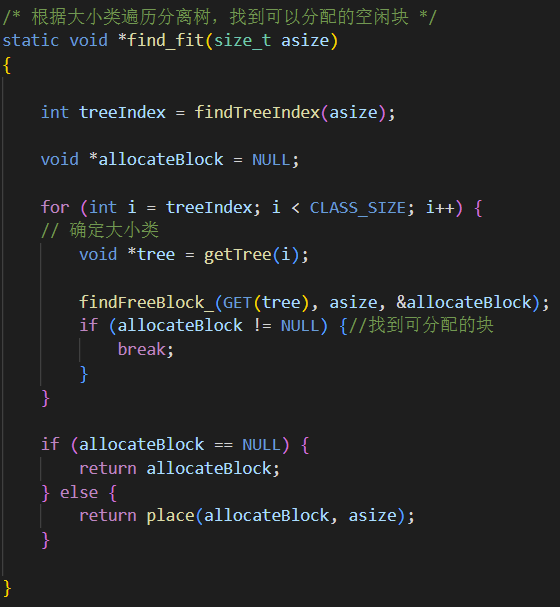


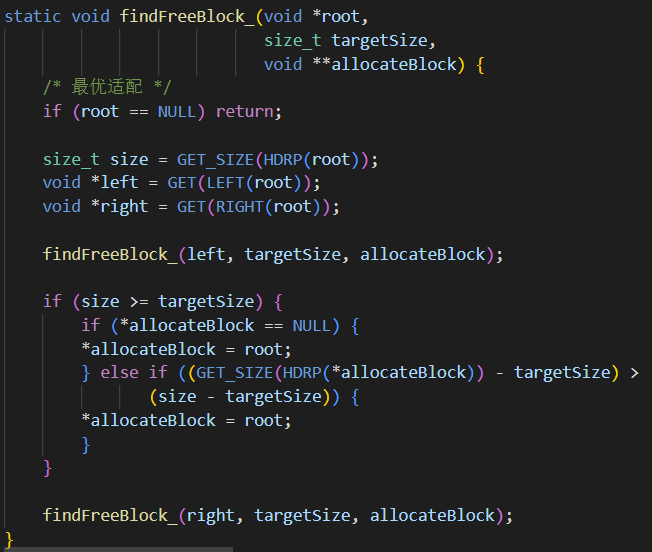
Delete函数也是普通的avl树插入＋平衡方法。



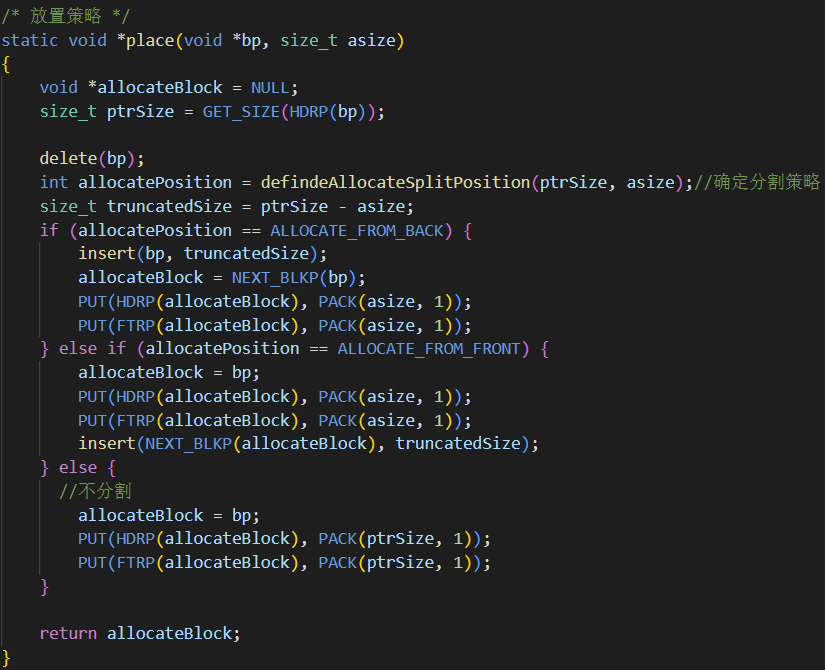


Find\_fit函数的方法，是把分离链表搜索大小类和二叉树的最优适配两者结合起来。





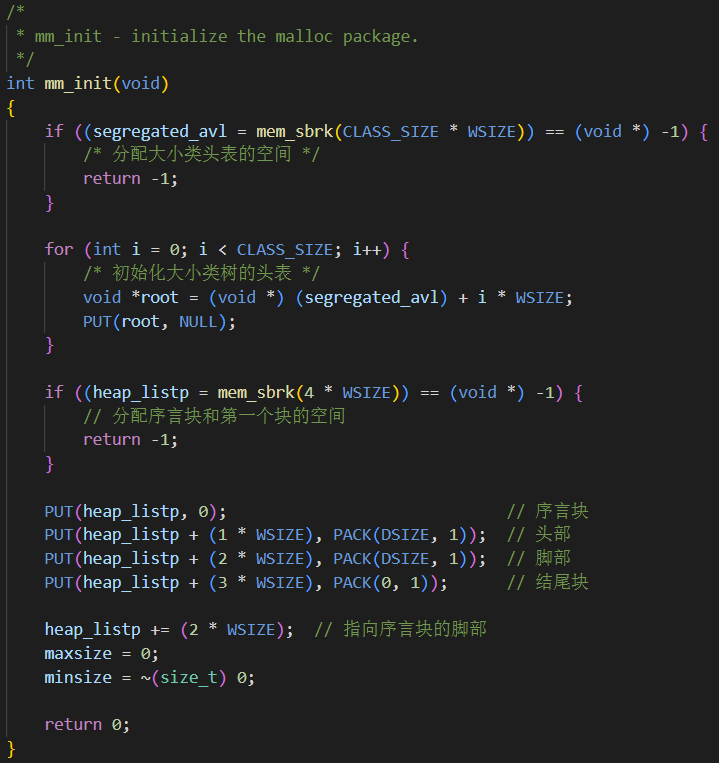
Place函数，由三种分割策略的情况组成。

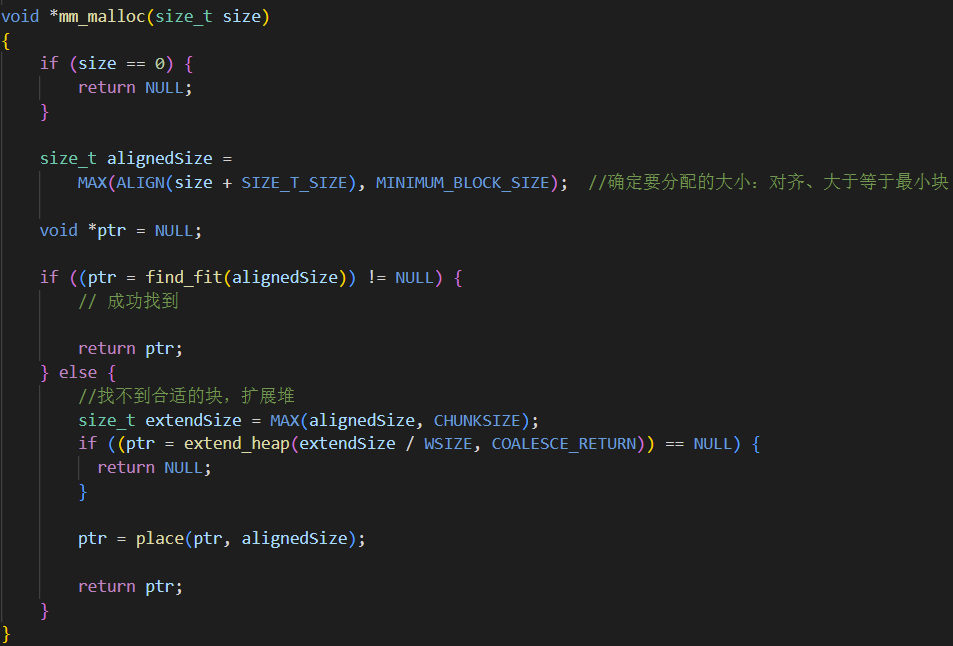


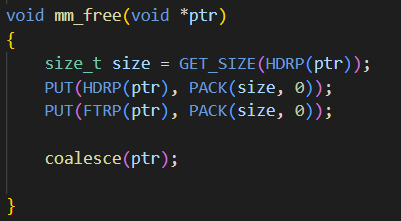
Realloc的放置策略略有不同，首先去掉了delete的步骤，因为realloc时指向的块是已分配块，不需要从树中删去。另外增加了memmove的环节，以便于与前方块合并。



Init，malloc，free三个函数，与先前无大差别。







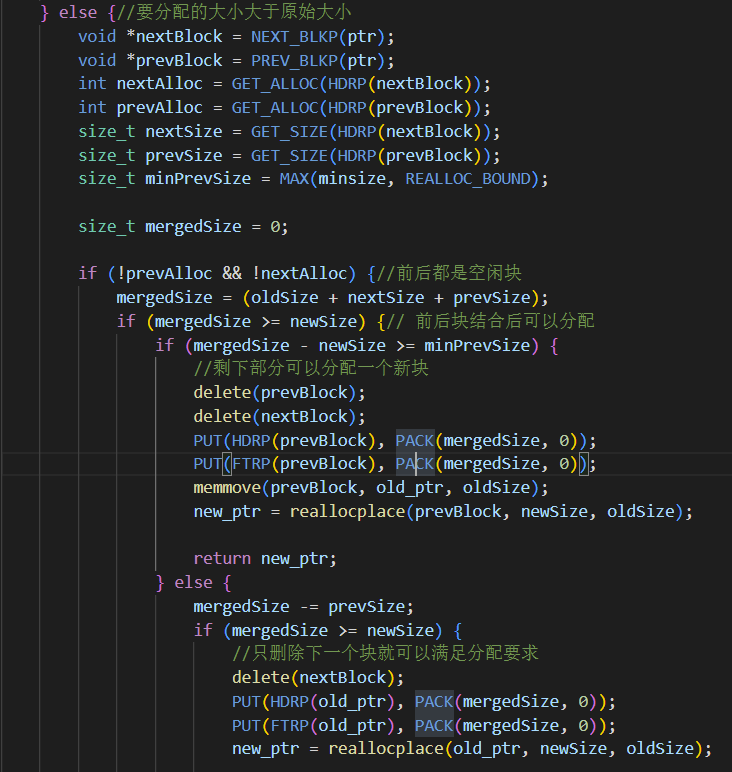
Realloc函数，前面都和之前一样，就不再展示。但是在要分配的大小大于原始大小这一部分，进行了进一步的优化。

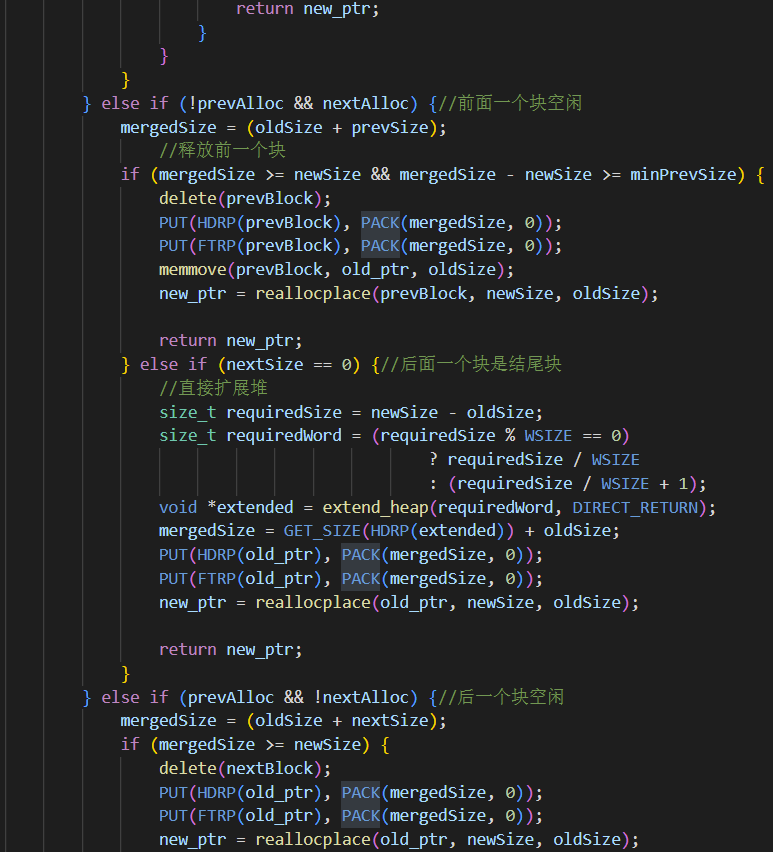
它分为四种情况：1.前后都是空闲块。这又分为三种情况，第一，只结合后方块就可以满足条件。第二，要结合前和后才能满足条件。第三，即使前后都结合，也不能满足分配条件。

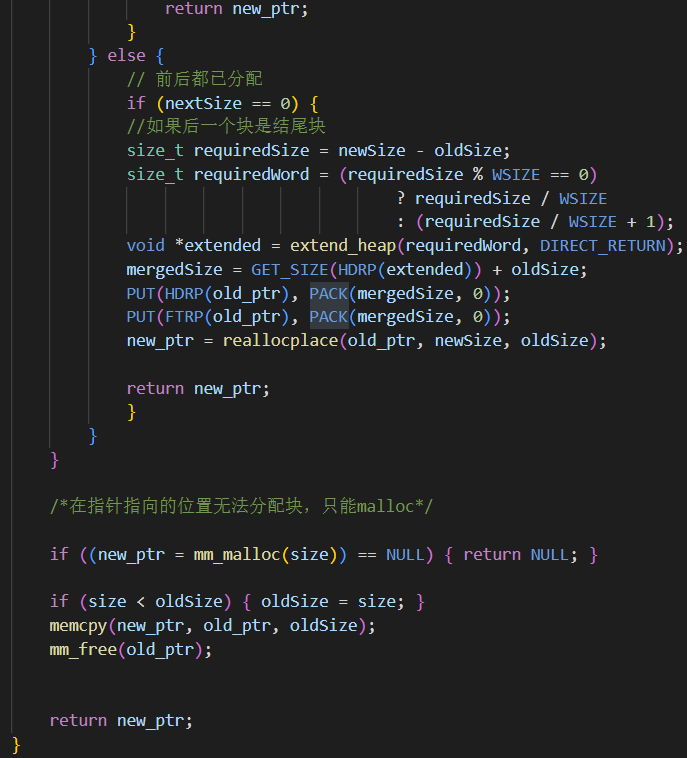
2.前一个块空闲。这又分为三种情况。第一，结合前方块可以满足条件。第二，后方块是结尾块。第三，结合前方块不能满足分配条件，而且后方块不是结尾块。

3.后一个块空闲。这分为两种情况。第一，结合后方块能满足分配条件。第二，结合后方块不能满足分配条件。

4.前后都不空闲。这分为两种情况。第一，后一个块是结尾块。第二，后一个块不是结尾块。

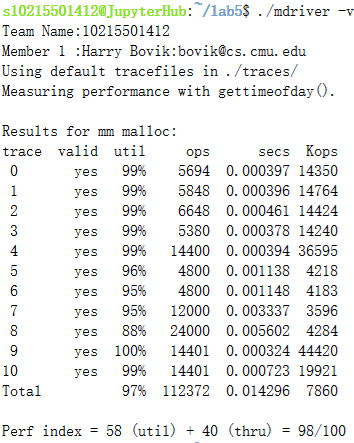






这四种情况中，每个的最后一种分情况，都将会引发malloc的执行，其他情况中，遇到结尾块可以直接extend\_heap，而如果空间足以分配会调用reallocplace函数。

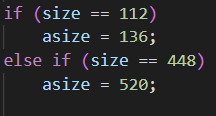
最终运行结果：



1. 实验总结

本实验耗时三天时间，期间遇到许多segmentation fault，调试这些bug，虽然需要大量耐心，但是也促使了我对指针的更深刻理解。由于没有尝试显式空闲链表，无法对比其优劣。隐式空闲链表最重要的缺点是顺序查找效率低，但是空间利用率相比平凡的二叉树和分离链表，其实是相当的。

关于测试用例的理解，可以看出，0123是很容易满足的，56容易达到90%，但是几乎不能更高了。4、7、8、9、10不容易满足。对比其他方式都相同的隐式链表、分离链表和二叉树，值得注意的是，隐式链表在9有比较好的实现，空间利用率有80%；二叉树在4有比较好的实现，空间利用率有88%，但是9和10都很低；分离链表在10有比较好的实现，空间利用率有64%。根据推测和观察，4应该是malloc和free了许多大小相同的块，7、8、9、10都是大小块的抖动，7和8是恒定的大小，所以可以根据特定的测试用例数字进行分配，例如：



而经过我最后一次对realloc的调试，发现realloc的优化可以很好的增加9和10的空间利用率。

总而言之，realloc的优化、放置块的策略、最优适配都可以很好的提高空间利用率，由于题目对时间的要求不高，就算运行时间总和0.08也可以达到40分满分，所以可以牺牲一点时间来换取空间利用率，从而提升这个实验的分数。

稍显遗憾的是没有很好的控制变量，从普通二叉树逐渐加入各种策略，看分数一点点升高，从而对每个优化策略的性能有更深刻的理解。不过这次实验我也收获颇丰，尤其是加深了对指针的认识，以及对分配器工作原理的了解。