练习 13

参考地点

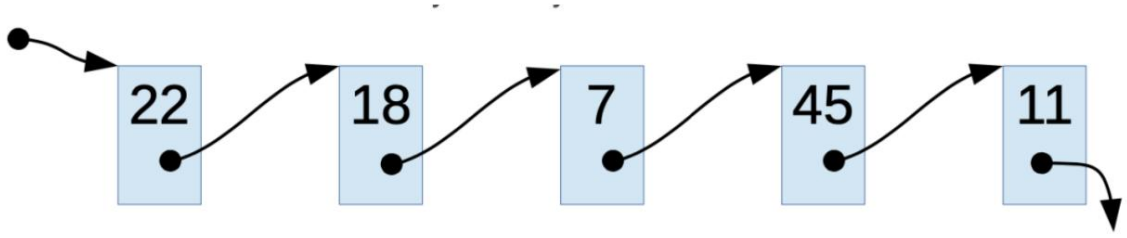
数组往往表现出良好的局部性,尤其是当您按顺序访问元素或依次访问附近的元素时
另一个（即,数组中的附近元素及时访问附近）。像链表这样的链接结构可能不会表现出那么多的局部性。从一个节点移动到
下一个节点可能涉及移动到一个完全不同的位置
记忆。这可能会损害各种类型的缓存（包括 TLB)的性能,并且如果程序不能全部放入主内存,则可能会真正损害虚拟内
存的性能。

在本练习中,我们将比较快速排序的数组和链表实现。这是一个很好的算法
考虑到,由于它的基本操作,根据选定的枢轴划分一系列值,主要是访问
元素有序。

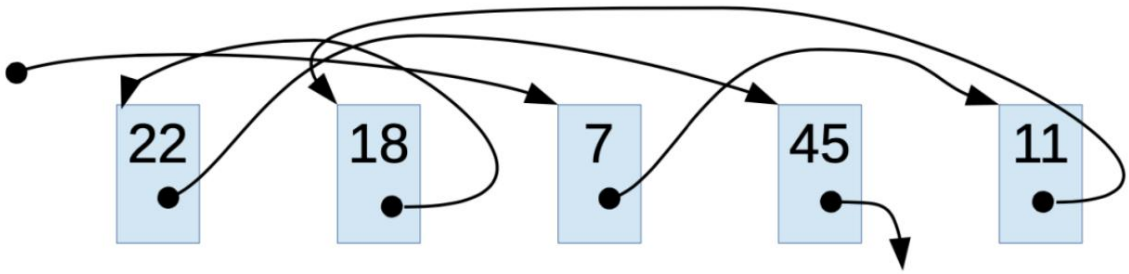
对于数组实现,当排序在数组中移动值时,我们仍然应该访问内存
命令。如果列表开始时乱序：

当排序重新排列项目时,它会在内存中四处移动它们,因此我们列表中附近的项目总是存储在
附近的内存位置：

这不是链表的工作方式。即使列表的节点开始在内存中有序,列表中有附近的节点
在附近的内存位置分配的顺序：



当列表被排序时,链接结构将改变以重新排列值,但节点实际上并没有在内存中移动,
因此访问列表中附近的项目可能不会表现出太多的地方性。



让我们看看这对性能有何影响。我写了一个基于数组的快速排序，array.c。它需要一个命令行

参数 n，给出要排序的项目数。然后，它生成一个包含 n 个项目的随机列表，并递归地对它们进行排序

使用快速排序，始终使用列表中的第一项作为基准。排序后，它打印出排序后的值列表（所以

您可以验证排序是否有效）。

$ ./数组 10
424238335
596516649
719885386
846930886
1189641421
1649760492
1681692777
1714636915
1804289383
1957747793

要在更大的列表上测量性能，我们可以使用 time 命令运行程序，并将输出重定向到

/dev/null（因此打印输出不会支配运行时）。这是我对 10,000,000 个值进行排序时得到的结果

remote.eos.ncsu.edu 的其中一台机器：

$时间./array 10000000 >/dev/null

真实的 0m3.762s
用户 0m3.695s
系统 0m0.059s

让我们将这次与排序链表进行比较。我已经编写了链表快速排序的部分实现。它创建一个包含与数组程序相同的值序列的列表，调用 sort() 函数递归快速排序

列表，然后打印出结果列表。您可以填写快速排序实现。一旦你这样做了，你应该能够像数组程序一样运行它，看看它与快速排序数组有何不同。 （顺便说一句，我的解决方案

是

在我上面使用的同一系统上，对 10,000,000 个随机值的链表进行快速排序需要 17 秒。告诉你这种情况会泄露测试的结果，但它会帮助你知道你是否正确地实现了快速

排序）。

就像数组程序一样,您的快速排序将使用列表中的第一个值作为基准对列表进行分区。您将制作两个临时列表,将所有值小于主元的节点放在一个列表中（我们称之为较小)和所有

值大于或等于另一个列表上的枢轴的节点（我们称之为更大）。然后,您将递归排序

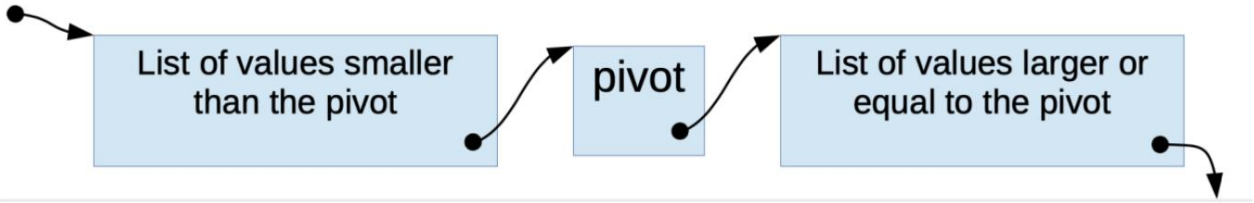
两个较小的列表,然后通过更改一些指针将它们组合成一个大的排序列表。的头

组合列表将是较小列表中的第一个节点。较小列表中的最后一个节点将需要指向枢轴作为其

下一个节点,而枢轴外的下一个指针将需要指向更大列表中的第一个节点。记住，

较小或较大的列表可能为空（例如,如果没有任何值小于主元）。对待空和

单节点列表作为特殊情况（不需要任何排序）,因此总会有一个枢轴节点。



为了更容易地组合这样的列表,我们将一个列表表示为一个小结构,并带有一个指向头节点的指针

列表和尾部的不同指针（列表的最后一个节点）。部分实现使用如下所示的结构。

```
// 列表的表示,带有头指针和尾指针。
类型定义结构{
    // 列表的头指针。
    节点*头;

    // 指向列表中最后一个节点的指针,如果列表 // 为空,则为 null。

    节点*尾巴;
} 列表;
```

跟上列表中的最后一个节点可以在恒定时间内将排序列表与枢轴组合起来，

而不必每次都去寻找最后一个节点。这样做可以将链表的排序时间缩短一半

（与仅跟上头指针并必须遍历列表以找到其最后一个节点相比）。

或者,您可以像下面这样表示您的列表,尾指针存储为列表末尾的空指针的地址（即使那是头指针）。如果您代表您的,则需要修改列表创建代码

以这种方式列出（你应该仍然得到相同的随机生成的列表,但你需要对

main 中的列表构建代码）,但是,总的来说,像这样表示列表为我节省了大约十几行代码,因为我

不必对空列表给予太多特殊待遇。

```
// 列表的表示,带有头指针和尾指针。 typedef struct { // 列表的头指针。

    节点*头;

    // 指向列表末尾的空指针。
    节点**尾巴;
} 列表;
```

一旦你的快速排序开始工作,它应该打印一个排序列表,就像基于数组的解决方案一样。实际上,
渐近地（大 O 表示法）,您的解决方案应该与基于数组的解决方案一样快。然而,随着
表示为链接列表的值序列,在排序之后或排序期间,引用位置将不会那么好。这
（以及其他一些事情)应该会导致更高的运行时间。尝试对 10,000,000 个值进行排序并发送
/dev/null 的输出,就像我们在上面对基于数组的实现所做的那样。

完成后,将已完成的 linked.c 程序的源代码提交给名为
EX13。