06讲链表（上）：如何实现LRU缓存淘汰算法

今天我们来聊聊“链表（Linked list）”这个数据结构。学习链表有什么用呢？为了回答这个问题，我们先来讨论一个经典的链表应用场景，那就是LRU缓存淘汰算法。

缓存是一种提高数据读取性能的技术，在硬件设计、软件开发中都有着非常广泛的应用，比如常见的CPU缓存、数据库缓存、浏览器缓存等等。

缓存的大小有限，当缓存被用满时，哪些数据应该被清理出去，哪些数据应该被保留？这就需要缓存淘汰策略来决定。常见的策略有三种：先进先出策略FIFO（First In，First Out）、最少使用策略LFU（Least Frequently Used）、最近最少使用策略LRU（Least Recently Used）。

这些策略你不用死记，我打个比方你很容易就明白了。假如说，你买了很多本技术书，但有一天你发现，这些书太多了，太占书房空间了，你要做个大扫除，扔掉一些书籍。那这个时候，你会选择扔掉哪些书呢？对应一下，你的选择标准是不是和上面的三种策略神似呢？

好了，回到正题，我们今天的开篇问题就是：**如何用链表来实现LRU缓存淘汰策略呢？** 带着这个问题，我们开始今天的内容吧！

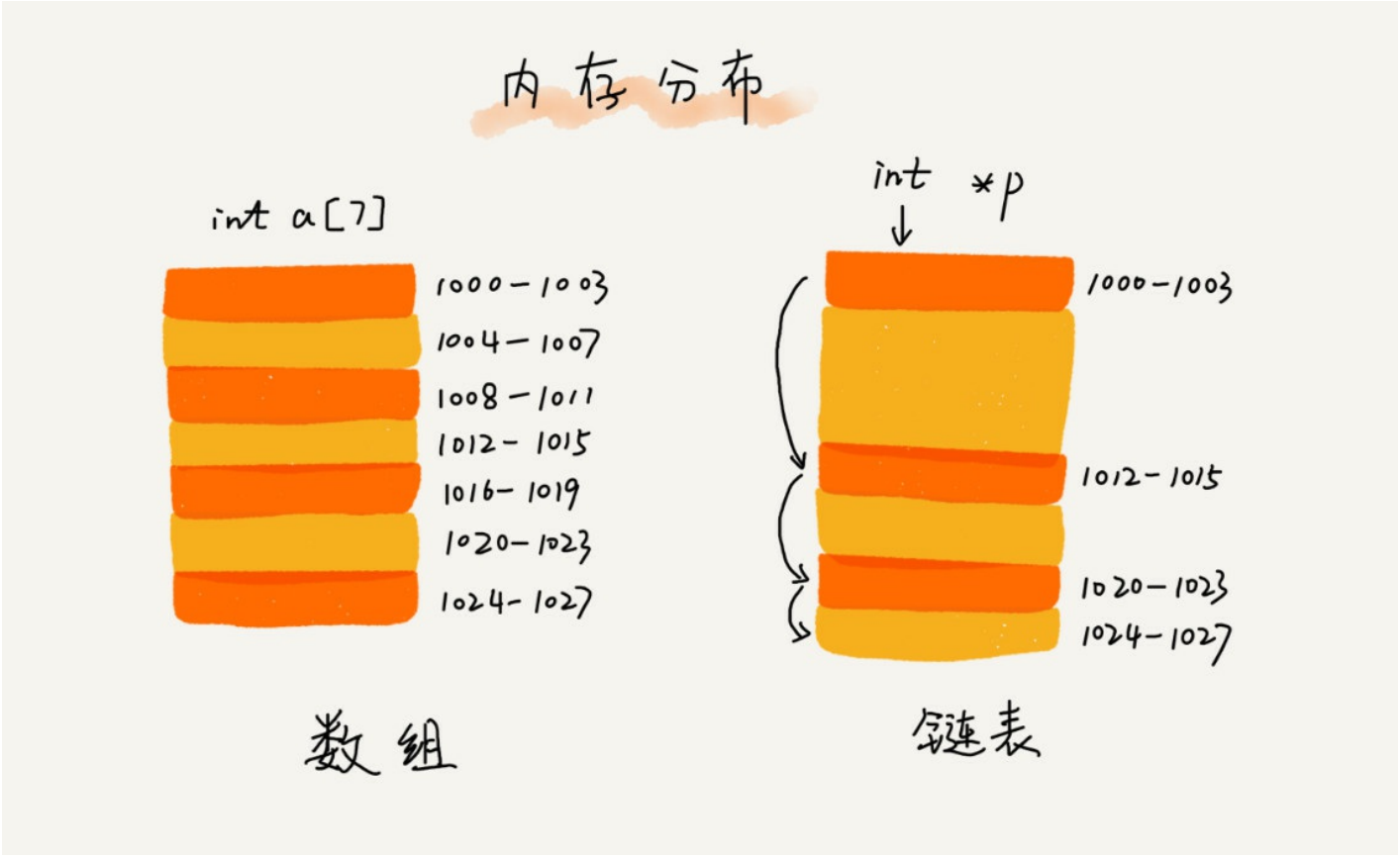
五花八门的链表结构

相比数组，链表是一种稍微复杂一点的数据结构。对于初学者来说，掌握起来也要比数组稍难一些。这两个非常基础、非常常用的数据结构，我们常常将会放到一块儿来比较。所以我们先来看，这两者有什么区别。

我们先从**底层的存储结构**上来看一看。

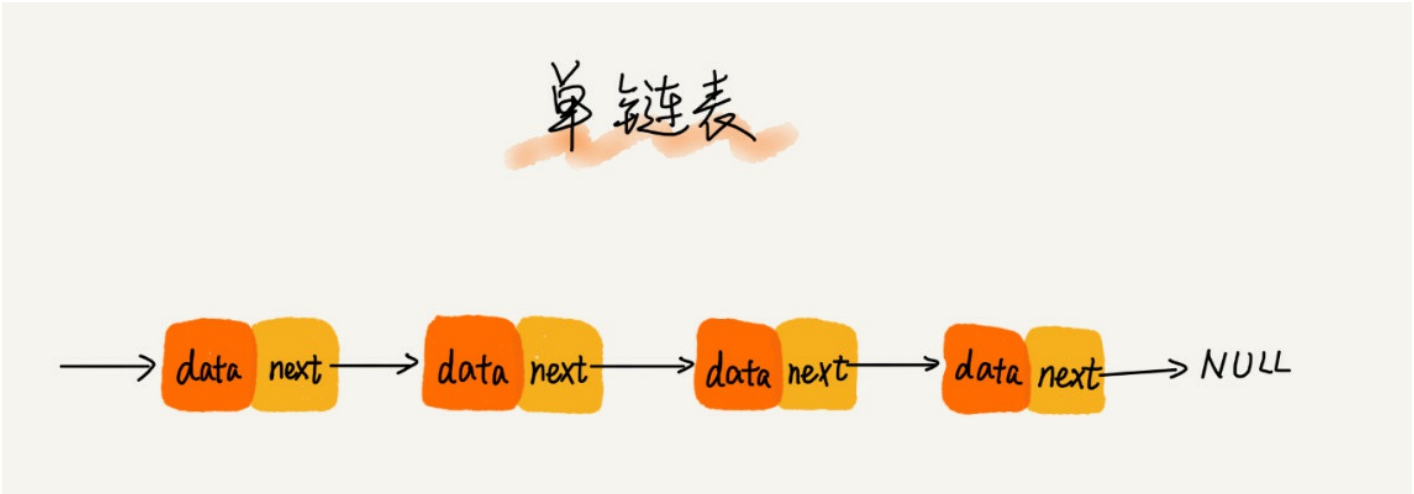
为了直观地对比，我画了一张图。从图中我们看到，数组需要一块**连续的内存空间**来存储，对内存的要求比较高。如果我们申请一个100MB大小的数组，当内存中没有连续的、足够大的存储空间时，即便内存的剩余总可用空间大于100MB，仍然会申请失败。

而链表恰恰相反，它并不需要一块连续的内存空间，它通过“指针”将一组**零散的内存块**串联起来使用，所以如果我们申请的是100MB大小的链表，根本不会有问题。



链表结构五花八门，今天我重点给你介绍三种最常见的链表结构，它们分别是：单链表、双向链表和循环链表。我们首先来看最简单、最常用的**单链表**。

我们刚刚讲到，链表通过指针将一组零散的内存块串联在一起。其中，我们把内存块称为链表的“**结点**”。为了将所有的结点串起来，每个链表的结点除了存储数据之外，还需要记录链上的下一个结点的地址。如图所示，我们把这个记录下个结点地址的指针叫作**后继指针next**。



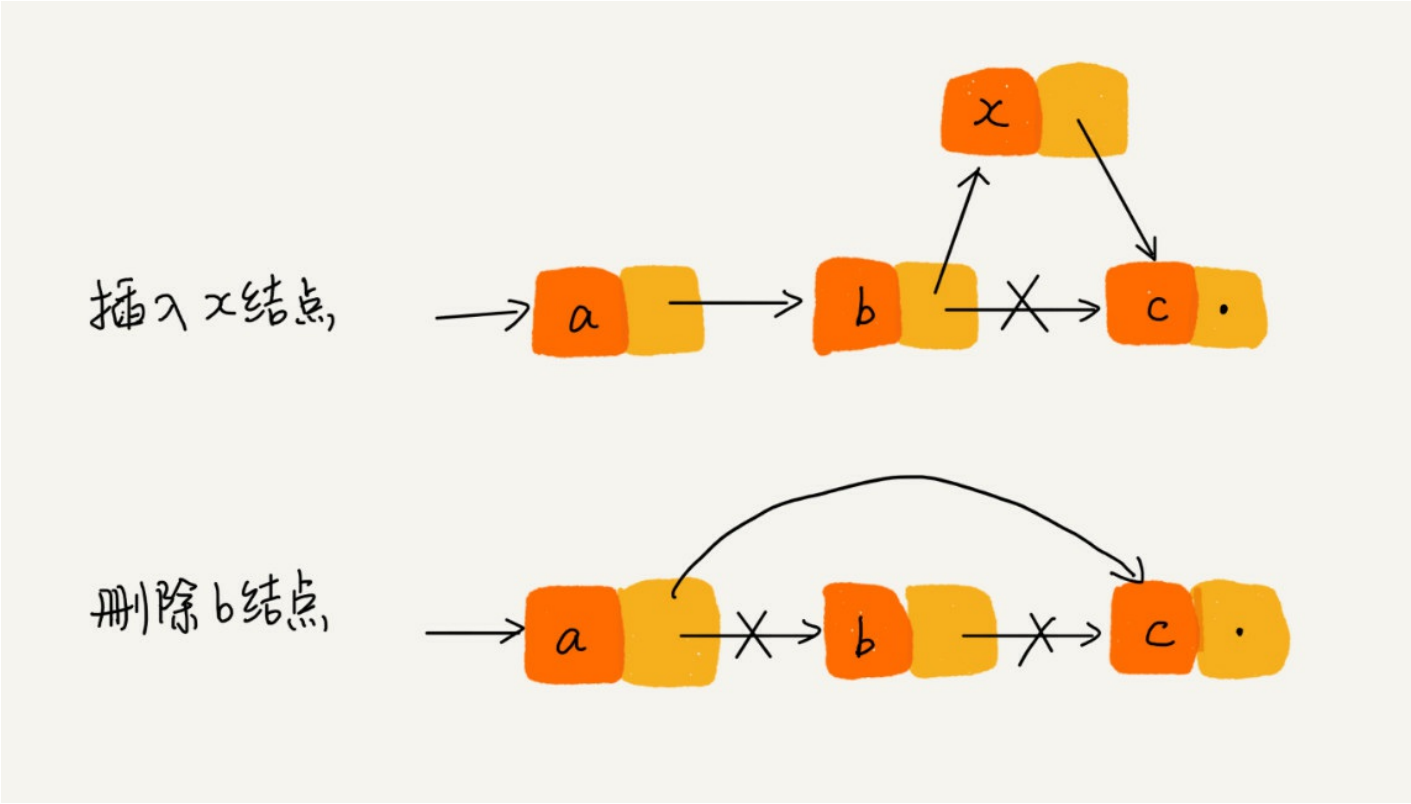
从我画的单链表图中，你应该可以发现，其中有两个结点是比较特殊的，它们分别是第一个结点和最后一个结点。我们习惯性地把第一个结点叫作**头结点**，把最后一个结点叫作**尾结点**。其中，头结点用来记录链表的基地址。有了它，我们就可以遍历得到整条链表。而尾结点特殊的地方是：指针不是指向下一个结点，而是指向一个**空地址NULL**，表示这是链表上最后一个结

点。

与数组一样，链表也支持数据的查找、插入和删除操作。

我们知道，在进行数组的插入、删除操作时，为了保持内存数据的连续性，需要做大量的数据搬移，所以时间复杂度是O(n)。而在链表中插入或者删除一个数据，我们并不需要为了保持内存的连续性而搬移结点，因为链表的存储空间本身就不是连续的。所以，在链表中插入和删除一个数据是非常快速的。

为了方便你理解，我画了一张图，从图中我们可以看出，针对链表的插入和删除操作，我们只需要考虑相邻结点的指针改变，所以对应的时间复杂度是O(1)。

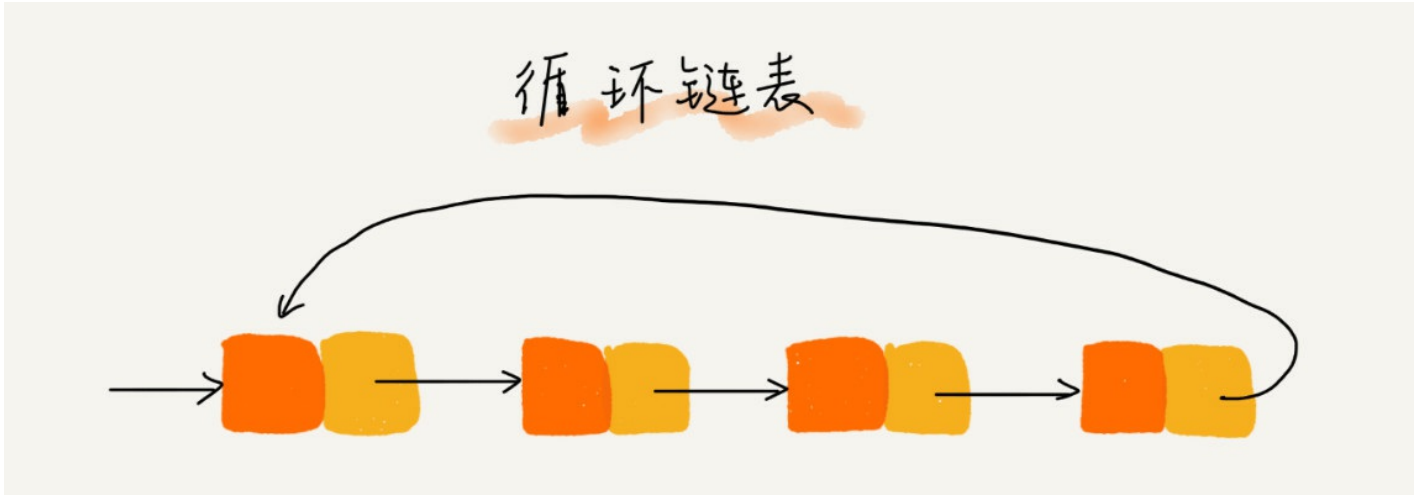


但是，有利就有弊。链表要想随机访问第k个元素，就没有数组那么高效了。因为链表中的数据并非连续存储的，所以无法像数组那样，根据首地址和下标，通过寻址公式就能直接计算出对应的内存地址，而是需要根据指针一个结点一个结点地依次遍历，直到找到相应的结点。

你可以把链表想象成一个队伍，队伍中的每个人都只知道自己后面的人是谁，所以当我们希望知道排在第k位的人是谁的时候，我们就需要从第一个人开始，一个一个地往下数。所以，链表随机访问的性能没有数组好，需要O(n)的时间复杂度。

好了，单链表我们就简单介绍完了，接着来看另外两个复杂的升级版，**循环链表**和**双向链表**。

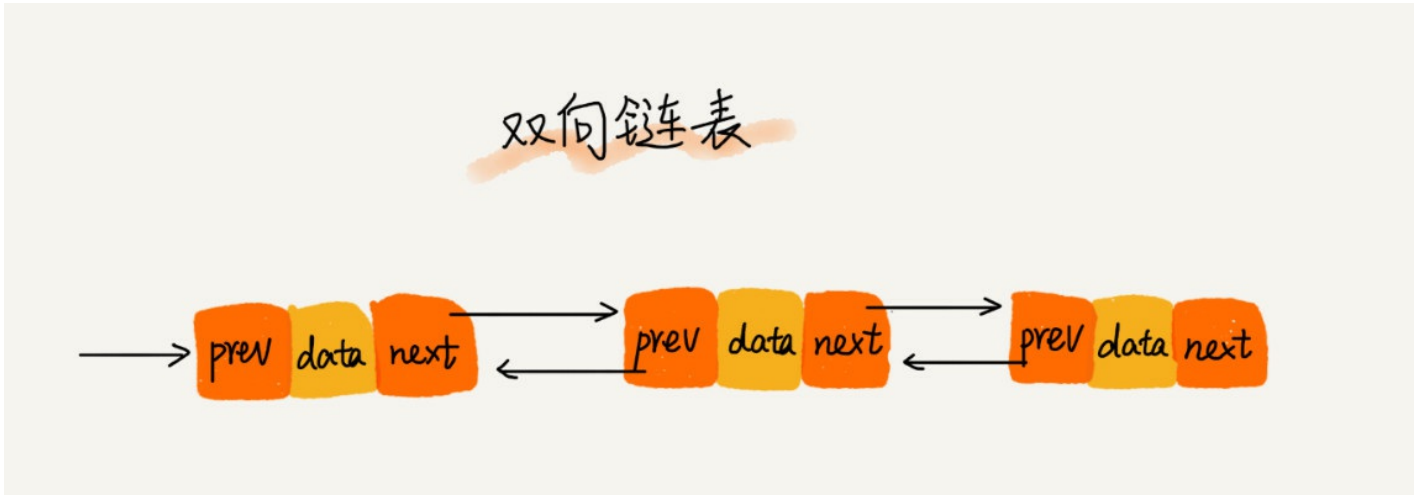
循环链表是一种特殊的单链表。实际上，循环链表也很简单。它跟单链表唯一的区别就在尾结点。我们知道，单链表的尾结点指针指向空地址，表示这就是最后的结点了。而循环链表的尾结点指针是指向链表的头结点。从我画的循环链表图中，你应该可以看出来，它像一个环一样首尾相连，所以叫作“循环”链表。



和单链表相比，**循环链表**的优点是从链尾到链头比较方便。当要处理的数据具有环型结构特点时，就特别适合采用循环链表。比如著名的约瑟夫问题。尽管用单链表也可以实现，但是用循环链表实现的话，代码就会简洁很多。

单链表和循环链表是不是都不难？接下来我们再来看一个稍微复杂的，在实际的软件开发中，也更加常用的链表结构：**双向链表**。

单向链表只有一个方向，结点只有一个后继指针next指向后面的结点。而双向链表，顾名思义，它支持两个方向，每个结点不止有一个后继指针next指向后面的结点，还有一个前驱指针prev指向前面的结点。



从我画的图中可以看出来，双向链表需要额外的两个空间来存储后继结点和前驱结点的地址。所以，如果存储同样多的数据，双向链表要比单链表占用更多的内存空间。虽然两个指针比较浪费存储空间，但可以支持双向遍历，这样也带来了双向链表操作的灵活性。那相比单链表，双向链表适合解决哪种问题呢？

从结构上来看，双向链表可以支持O(1)时间复杂度的情况下找到前驱结点，正是这样的特点，也使双向链表在某些情况下的插入、删除等操作都要比单链表简单、高效。

你可能会说，我刚讲到单链表的插入、删除操作的时间复杂度已经是O(1)了，双向链表还能再怎么高效呢？别着急，刚刚的分析比较偏理论，很多数据结构和算法书籍中都会这么讲，但是这种说法实际上是不准确的，或者说是有先决条件的。我再来带你分析一下链表的两个操作。

我们先来看**删除操作**。

在实际的软件开发中，从链表中删除一个数据无外乎这两种情况：

- 删除结点中“值等于某个给定值”的结点；
- 删除给定指针指向的结点。

对于第一种情况，不管是单链表还是双向链表，为了查找到值等于给定值的结点，都需要从头结点开始一个一个依次遍历对比，直到找到值等于给定值的结点，然后再通过我前面讲的指针操作将其删除。

尽管单纯的删除操作时间复杂度是O(1)，但遍历查找的时间是主要的耗时点，对应的时间复杂度为O(n)。根据时间复杂度分析中的加法法则，删除值等于给定值的结点对应的链表操作的总时间复杂度为O(n)。

对于第二种情况，我们已经找到了要删除的结点，但是删除某个结点q需要知道其前驱结点，而单链表并不支持直接获取前驱结点，所以，为了找到前驱结点，我们还是要从头结点开始遍历链表，直到p->next=q，说明p是q的前驱结点。

但是对于双向链表来说，这种情况就比较有优势了。因为双向链表中的结点已经保存了前驱结点的指针，不需要像单链表那样遍历。所以，针对第二种情况，单链表删除操作需要O(n)的时间复杂度，而双向链表只需要在O(1)的时间复杂度内就搞定了！

同理，如果我们希望在链表的某个指定结点前面插入一个结点，双向链表比单链表有很大的优势。双向链表可以在O(1)时间复杂度搞定，而单向链表需要O(n)的时间复杂度。你可以参照我刚刚讲过的删除操作自己分析一下。

除了插入、删除操作有优势之外，对于一个有序链表，双向链表的按值查询的效率也要比单链表高一些。因为，我们可以记录上次查找的位置p，每次查询时，根据要查找的值与p的大小关系，决定是往前还是往后查找，所以平均只需要查找一半的数据。

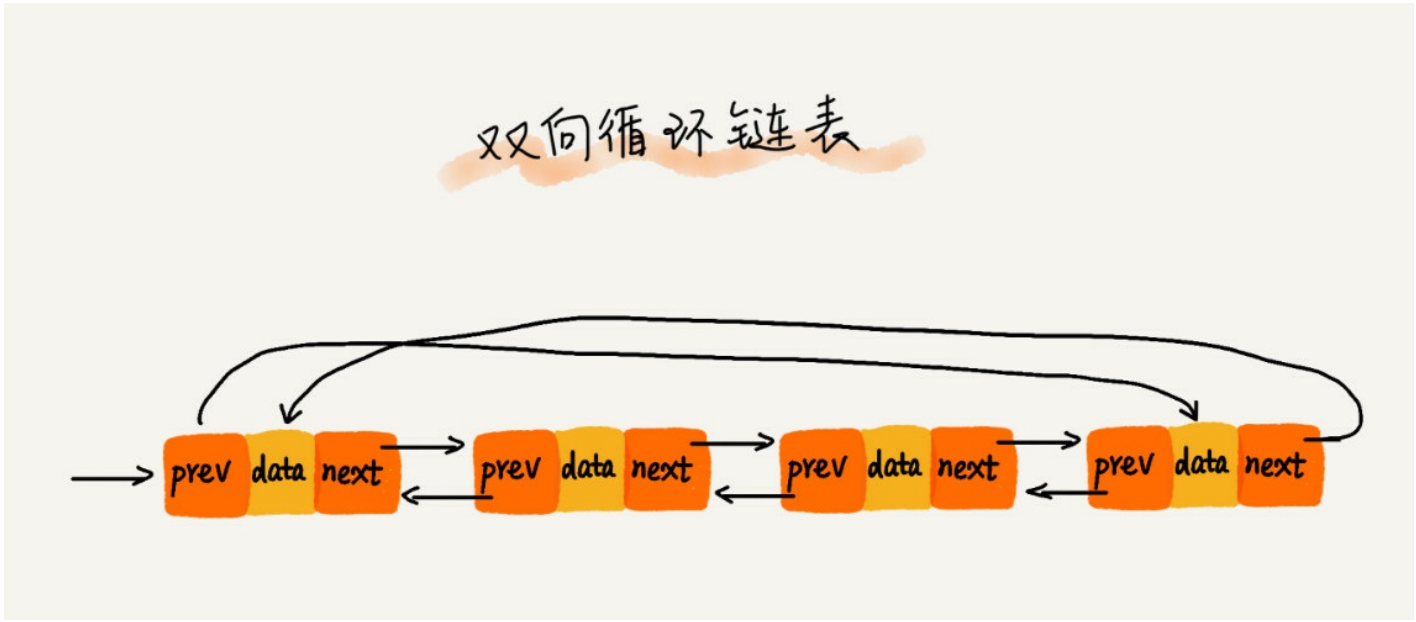
现在，你有没有觉得双向链表要比单链表更加高效呢？这就是为什么在实际的软件开发中，双向链表尽管比较费内存，但还是比单链表的应用更加广泛的原因。如果你熟悉Java语言，你肯定用过LinkedHashMap这个容器。如果你深入研究LinkedHashMap的实现原理，就会发现其中就用到了双向链表这种数据结构。

实际上，这里有一个更加重要的知识点需要你掌握，那就是**用空间换时间**的设计思想。当内存空间充足的时候，如果我们更加追求代码的执行速度，我们就可以选择空间复杂度相对较高、但时间复杂度相对很低的算法或者数据结构。相反，如果内存比较紧缺，比如代码跑在手机或者单片机上，这个时候，就要反过来用时间换空间的设计思路。

还是开篇缓存的例子。缓存实际上就是利用了空间换时间的设计思想。如果我们把数据存储在硬盘上，会比较节省内存，但每次查找数据都要询问一次硬盘，会比较慢。但如果我们通过缓存技术，事先将数据加载在内存中，虽然会比较耗费内存空间，但是每次数据查询的速度就大大提高了。

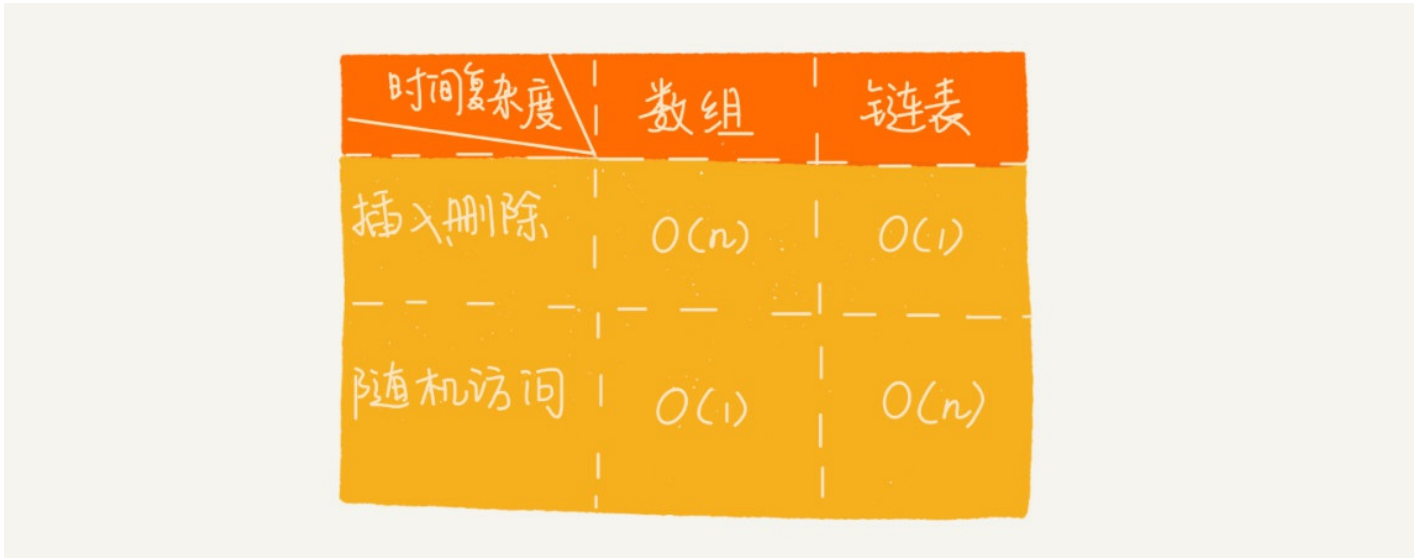
所以我总结一下，对于执行较慢的程序，可以通过消耗更多的内存（空间换时间）来进行优化；而消耗过多内存的程序，可以通过消耗更多的时间（时间换空间）来降低内存的消耗。你还能想到其他时间换空间或者空间换时间的例子吗？

了解了循环链表和双向链表，如果把这两种链表整合在一起就是一个新的版本：**双向循环链表**。我想不用我多讲，你应该知道双向循环链表长什么样子了吧？你可以自己试着在纸上画一画。



链表VS数组性能大比拼

通过前面内容的学习，你应该已经知道，数组和链表是两种截然不同的内存组织方式。正是因为内存存储的区别，它们插入、删除、随机访问操作的时间复杂度正好相反。



不过，数组和链表的对比，并不能局限于时间复杂度。而且，在实际的软件开发中，不能仅仅利用复杂度分析就决定使用哪个数据结构来存储数据。

数组简单易用，在实现上使用的是连续的内存空间，可以借助CPU的缓存机制，预读数组中的数据，所以访问效率更高。而链表在内存中并不是连续存储，所以对CPU缓存不友好，没办法有效预读。

数组的缺点是大小固定，一经声明就要占用整块连续内存空间。如果声明的数组过大，系统可能没有足够的连续内存空间分配给它，导致“内存不足（out of memory）”。如果声明的数组过小，则可能出现不够用的情况。这时只能再申请一个更大的内存空间，把原数组拷贝进去，非常费时。链表本身没有大小的限制，天然地支持动态扩容，我觉得这也是它与数组最大的区别。

你可能会说，我们Java中的ArrayList容器，也可以支持动态扩容啊？我们上一节课讲过，当我们往支持动态扩容的数组中插入一个数据时，如果数组中没有空闲空间了，就会申请一个更大的空间，将数据拷贝过去，而数据拷贝的操作是非常耗时的。

我举一个稍微极端的例子。如果我们用ArrayList存储了了1GB大小的数据，这个时候已经没有空闲空间了，当我们再插入数据的时候，ArrayList会申请一个1.5GB大小的存储空间，并且把原来那1GB的数据拷贝到新申请的空间上。听起来是不是就很耗时?

除此之外，如果你的代码对内存的使用非常苛刻，那数组就更适合你。因为链表中的每个结点都需要消耗额外的存储空间去存储一份指向下一个结点的指针，所以内存消耗会翻倍。而且，对链表进行频繁的插入、删除操作，还会导致频繁的内存申请和释放，容易造成内存碎片，如果是Java语言，就有可能会导致频繁的GC（Garbage Collection，垃圾回收）。

所以，在我们实际的开发中，针对不同类型的项目，要根据具体情况，权衡究竟是选择数组还是链表。

解答开篇

好了，关于链表的知识我们就讲完了。我们现在回过头来看下开篇留给你的思考题。如何基于链表实现LRU缓存淘汰算法?

我的思路是这样的：我们维护一个有序单链表，越靠近链表尾部的结点是越早之前访问的。当有一个新的数据被访问时，我们从链表头开始顺序遍历链表。

1.如果此数据之前已经被缓存在链表中了，我们遍历得到这个数据对应的结点，并将其从原来的位置删除，然后再插入到链表的头部。

2.如果此数据没有在缓存链表中，又可以分为两种情况：

- 如果此时缓存未满，则将此结点直接插入到链表的头部；
- 如果此时缓存已满，则链表尾结点删除，将新的数据结点插入链表的头部。

这样我们就用链表实现了一个LRU缓存，是不是很简单?

现在我们来看下m缓存访问的时间复杂度是多少。因为不管缓存有没有满，我们都需要遍历一遍链表，所以这种基于链表的实现思路，缓存访问的时间复杂度为O(n)。

实际上，我们可以继续优化这个实现思路，比如引入**散列表**（Hash table）来记录每个数据的位置，将缓存访问的时间复杂度降到O(1)。因为要涉及我们还没有讲到的数据结构，所以这个优化方案，我现在就不详细说了，等讲到散列表的时候，我会再拿出来讲。

除了基于链表的实现思路，实际上还可以用数组来实现LRU缓存淘汰策略。如何利用数组实现LRU缓存淘汰策略呢？我把这个问题留给你思考。

内容小结

今天我们讲了一种跟数组“相反”的数据结构，链表。它跟数组一样，也是非常基础、非常常用的数据结构。不过链表要比数组稍微复杂，从普通的单链表衍生出来好几种链表结构，比如双向链表、循环链表、双向循环链表。

和数组相比，链表更适合插入、删除操作频繁的场景，查询的时间复杂度较高。不过，在具体软件开发中，要对数组和链表的各种性能进行对比，综合来选择使用两者中的哪一个。

课后思考

如何判断一个字符串是否是回文字符串的问题，我想你应该听过，我们今天的思题目就是基于这个问题的改造版本。如果字符串是通过单链表来存储的，那该如何来判断是一个回文串呢？你有什么好的解决思路呢？相应的时间空间复杂度又是多少呢?

欢迎留言和我分享，我会第一时间给你反馈。

我已将本节内容相关的详细代码更新到GitHub，戳此即可查看。

精选留言

Rain
Re Ydyhm:

“数组简单易用，在实现上使用的是连续的内存空间，可以借助 CPU 的缓存机制，预读数组中的数据，所以访问效率更高。而链表在内存中并不是连续存储，所以对 CPU 缓存不友好，没办法有效预读。” 这里的CPU缓存机制指的是什么？为什么就数组更好了？

我没有百度也没有Google。之前开发时遇到过，我斗胆说下。
CPU在从内存读取数据的时候，会先把读取到的数据加载到CPU的缓存中。而CPU每次从内存读取数据并不是只读取那个特定要访问的地址，而是读取一个数据块(这个大小我不太确定。。)并保存到CPU缓存中，然后下次访问内存数据的时候就会先从CPU缓存开始查找，如果找到就不需要再从内存中取。这样就实现了比内存访问速度更快的机制，也就是CPU缓存存在的意义:为了弥补内存访问速度过慢与CPU执行速度快之间的差异而引入。

对于数组来说，存储空间是连续的，所以在加载某个下标的时候可以把以后的几个下标元素也加载到CPU缓存这样执行速度会快于存储空间不连续的链表存储。

大牛请指正哈！
2018-10-04 07:25

作者回复

同学，太爱你了。写的太好了！就喜欢你这样的，减轻了我很多回复留言的工作量。
2018-10-04 21:09

JK David

思考题：

使用快慢两个指针找到链表中点，慢指针每次前进一步，快指针每次前进两步。在慢指针前进的过程中，同时修改其 next 指针，使得链表前半部分反序。最后比较中点两侧的链表是否相等。

时间复杂度：O(n)
空间复杂度：O(1)

https://github.com/andavid/leetcode-java/blob/master/note/234/README.md

2018-10-03 23:04

作者回复

思路正确，不过空间复杂度计算的不对，应该是O(1)，不是O(n)。我们要看额外的内存消耗，不是看链表本身存储需要多少空间。

2018-10-04 21:29

Liam

1 快慢指针定位中间节点
2 从中间节点对后半部分逆序
3 前后半部分比较，判断是否为回文
4 后半部分逆序复原

时间复杂度On, 空间复杂度O1
把LRU和回文都实现了一遍~~

如果是双向链表，时间效率更高，看了下LinkedList，底层也是用双向链表实现

2018-10-03 16:31

作者回复

回答的很好！

2018-10-03 23:45

glbfor.gtw

1 快慢指针定位中间节点（这里要区分奇偶情况）
1.1 奇数情况，中点位置不需要矫正
1.2 偶数情况，使用偶数定位中点策略，要确定是返回上中位数或下中位数
1.2.1 如果是返回上中位数，后半部分串头取next
1.2.2 如果是返回下中位数，后半部分串头既是当前节点位置，但前半部分串尾要删除掉当前节点
2 从中间节点对后半部分逆序，或者将前半部分逆序
3 一次循环比较，判断是否为回文
4 恢复现场

题外话，这种操作有点BT啊？实际运用场景中，也真的直接改变引用值吗？至少在多线程情况，要加N多锁（Read Write都要加锁），这个时间成本就不能简单用时间复杂度来衡量了。如果是用copy 机制，不论是倒置前半段还是后半段，至少有一段是需要n/2个节点副本的空间消耗的，那么空间复杂度就是O（n）？？啦~跑题了，跑题了~~

老师，你给我评价被~ 我比较容易钻牛角尖。。

2018-10-11 14:31

作者回复

回答的非常好

2018-10-11 19:11

sky

用快慢指针先找到中点，然后把后半段链表reversed，然后一个指针在头部，一个指针再中点，开始逐个比较，时间复杂度是O（n）

2018-10-09 08:22

作者回复

对的！

2018-10-09 09:32

姜威

五、应用

1.如何分别用链表和数组实现LRU缓冲淘汰策略?

1）什么是缓存?

缓存是一种提高数据读取性能的技术，在硬件设计、软件开发中都有着非广泛的应用，比如常见的CPU缓存、数据库缓存、浏览器缓存等等。

2）为什么使用缓存？即缓存的特点

缓存的大小是有限的，当缓存被用满时，哪些数据应该被清理出去，哪些数据应该被保留？就需要用到缓存淘汰策略。

3）什么是缓存淘汰策略?

指的是当缓存被用满时清理数据的优先顺序。

4）有哪些缓存淘汰策略?

常见的3种包括先进先出策略FIFO（First In，First Out）、最少使用策略LFU（Least Frenquently Used）、最近最少使用策略LRU（Least Recently Used）。

5）链表实现LRU缓存淘汰策略

当访问的数据没有存储在缓存的链表中时，直接将数据插入链表表头，时间复杂度为O(1)；当访问的数据存在于存储的链表中时，将该数据对应的节点，插入到链表表头,时间复杂度为O(n)。如果缓存被占满，则从链表尾部的数据开始清理，时间复杂度为O(1)。

6）数组实现LRU缓存淘汰策略

方式一：首位置保存最新访问数据，末尾位置优先清理

当访问的数据未存在于缓存的数组中时，直接将数据插入数组第一个元素位置，此时数组所有元素需要向后移动1个位置，时间复杂度为O(n)；当访问的数据存在于缓存的数组中时，查找到数据并将其插入数组的第一个位置，此时亦需移动数组元素，时间复杂度为O(n)。缓存用满时，则清理掉末尾的数据，时间复杂度为O(1)。

方式二：首位置优先清理，末尾位置保存最新访问数据

当访问的数据未存在于缓存的数组中时，直接将数据添加进数组作为当前最有一个元素时间复杂度为O(1)；当访问的数据存在于缓存的数组中时，查找到数据并将其插入当前数组最后一个元素的位置，此时亦需移动数组元素，时间复杂度为O(n)。缓存用满时，则清理掉数组首位置的元素，且剩余数组元素需整体前移一位，时间复杂度为O(n)。（优化：清理的时候可以考虑一次性清理一定数量，从而降低清理次数，提高性能。）

2.如何通过单链表实现“判断某个字符串是否为水仙花字符串”？（比如 上海自来水来自海上）

1）前提：字符串以单个字符的形式存储在单链表中。

2）遍历链表，判断字符个数是否为奇数，若为偶数，则不是。

3）将链表中的字符倒序存储一份在另一个链表中。

4）同步遍历2个链表，比较对应的字符是否相等，若相等，则是水仙花字串，否则，不是。

六、设计思想

时空替换思想：“用空间换时间” 与 “用时间换空间”

当内存空间充足的时候，如果我们更加追求代码的执行速度，我们就可以选择空间复杂度相对较高，时间复杂度小相对较低的算法和数据结构，缓存就是空间换时间的例子。如果内存比较紧缺，比如代码跑在手机或者单片机上，这时，就要反过来用时间换空间的思路。

2018-10-03 10:39

作者回复

2018-10-03 23:57

molybdenum

看了大家的评论学习到了快慢指针法，看语言描述没太懂，自己用代码写了下才明白。

大致思路如下

由于回文串最重要的就是对称，那么最重要的问题就是找到那个中心，用快指针每步两格走，当他到达链表末端的时候，慢指针刚好到达中心，慢指针在过来的这趟路上还做了一件事，他把走过的节点反向了，在中心点再开辟一个新的指针用于往回走，而慢指针继续向前，当慢指针扫完整个链表，就可以判断这是回文串，否则就提前退出，总的来说时间复杂度按慢指针遍历一遍来算是O(n),空间复杂度因为只开辟了3个额外的辅助，所以是o(1)

2018-10-07 00:31

_stuView

双向链表存储，两个指针分别从头节点和尾节点开始遍历，依次比较节点value，判断是否为回文序列

2018-10-03 08:48

姜威

总结

一、什么是链表?

1.和数组一样，链表也是一种线性表。

2.从内存结构来看，链表的内存结构是不连续的内存空间，是将一组零散的内存块串联起来，从而进行数据存储的数据结构。

3.链表中的每一个内存块被称为节点Node。节点除了存储数据外，还需记录链上下一个节点的地址，即后继指针next。

二、为什么使用链表？即链表的特点

1.插入、删除数据效率高O(1)级别（只需更改指针指向即可），随机访问效率低O(n)级别（需要从链头至链尾进行遍历）。

2.和数组相比，内存空间消耗更大，因为每个存储数据的节点都需要额外的空间存储后继指针。

三、常用链表：单链表、循环链表和双向链表

1.单链表

1）每个节点只包含一个指针，即后继指针。

2）单链表有两个特殊的节点，即首节点和尾节点。为什么特殊？用首节点地址表示整条链表，尾节点的后继指针指向空地址null。

3）性能特点：插入和删除节点的时间复杂度为O（1），查找的时间复杂度为O(n)。

2.循环链表

1）除了尾节点的后继指针指向首节点的地址外均与单链表一致。

2）适用于存储有循环特点的数据，比如约瑟夫问题。

3.双向链表

1）节点除了存储数据外，还有两个指针分别指向前一个节点地址（前驱指针prev）和下一个节点地址（后继指针next）。

2）首节点的前驱指针prev和尾节点的后继指针均指向空地址。

3）性能特点：

和单链表相比，存储相同的数据，需要消耗更多的存储空间。

插入、删除操作比单链表效率更高O(1)级别。以删除操作为例，删除操作分为2种情况：给定数据值删除对应节点和给定节点地址删除节点。对于前一种情况，单链表和双向链表都需要从头到尾进行遍历从而找到对应节点进行删除，时间复杂度为O(n)。对于第二种情况，要进行删除操作必须找到前驱节点，单链表需要从头到尾进行遍历直到p->next = q，时间复杂度为O(n)，而双向链表可以直接找到前驱节点，时间复杂度为O(1)。

对于一个有序链表，双向链表的按值查询效率要比单链表高一些。因为我们可以记录上次查找的位置p，每一次查询时，根据要查找的值与p的大小关系，决定是往前还是往后查找，所以平均只需要查找一半的数据。

4.双向循环链表：首节点的前驱指针指向尾节点，尾节点的后继指针指向首节点。

四、选择数组还是链表?

1.插入、删除和随机访问的时间复杂度

数组：插入、删除的时间复杂度是O(n)，随机访问的时间复杂度是O(1)。

链表：插入、删除的时间复杂度是O(1)，随机访问的时间复杂端是O(n)。

2.数组缺点

1）若申请内存空间很大，比如100M，但若内存空间没有100M的连续空间时，则会申请失败，尽管内存可用空间超过100M。

2）大小固定，若存储空间不足，需进行扩容，一旦扩容就要进行数据复制，而这时非常费时的。

3.链表缺点

1）内存空间消耗更大，因为需要额外的空间存储指针信息。

2）对链表进行频繁的插入和删除操作，会导致频繁的内存申请和释放，容易造成内存碎片，如果是Java语言，还可能会造成频繁的GC（自动垃圾回收器）操作。

4.如何选择?

数组简单易用，在实现上使用连续的内存空间，可以借助CPU的缓冲机制预读数组中的数据，所以访问效率更高，而链表在内存中并不是连续存储，所以对CPU缓存不友好，没办法预读。

如果代码对内存的使用非常苛刻，那数组就更适合。

2018-10-03 10:37

作者回复

2018-10-03 23:57

Joshua 兆甲

习题解答

1.快进慢进法[两组指针，从头开始，a组一次进一，b组一次进二，b组到终点时，a组位置即为链表中间结点，循环次数为链表除去中间结点后前后两组的长度] 求得单向链表“中间”节点。并计算遍历次数，经过验证，遍历次数为‘’半链表‘’长度

2.从中间结点开始，以动态步长[每第i次步长是半链表长度-i+1]遍历链表，同时，从头节点开始，以1步长遍历。比较两组对应元素是否相同，相同继续，不同退出，返回不是回文字符串的结论。

3.返回是回文字符串的结论，退出.

空间复杂度O(n). 不用连续内存，可以磁盘操作

时间复杂度度O(n). 主要费时操作遍历

算了，不够直观，不易别人看懂。还是先把单项链表转存为线性表。

1.单向遍历，获得对应的线性表Arr，求线性表长度为L

2.运用线性表可以任意访问的性质，遍历Arr，令下标i从0。比较Arr[i]和Arr[L－i]是否相等 相等继续，不等报告不是回文字符串结论，退出

3.报告是回文字符串结论，结束。

空间复杂度O(n)

时间复杂度O(n)

看起来一样，这个就需要字符串不太大，有足够的连续内存可以分配，而且，预先不知道链表多长，可能还会遇到扩容问题。

2018-10-03 07:54

雨山

果然有程序员风格，放假还更新，昨天临睡前就看完了，但是没有评价，总之这个课绝对物有所值。

2018-10-03 08:22

无崖子

用数组解决Lru缓存问题：

维护一个有序的数组，越靠近数组首位置的数据越是最早访问的。

1.如果这个数据已经存在于数组中，把对应位置的数据删掉，直接把这个数据加到数组的最后一位。时间复杂度为o(n)

2.如果数据不存在这个数组中，数据还有空间的话，就把数据直接插到最后一位。没有的话，就把第一个数据删掉，然后把数据插入到数组最后一个。这样的时间复杂度为o(n）。

第一个小伙伴的留言有点问题。判断是否为回文串和奇偶数没关系吧，偶数个字符串也可以是哈，比如abccba。

2018-10-05 00:08

落叶飞逝的恋

老师，关于解答开篇那边，能不能附加一些代码示例，这样配合代码跟思路讲解，可能更好的理解呢。

2018-10-03 08:33

徐凯

通过一个栈 遍历整个链表 然后再从栈中弹出 如果元素都匹配则为回文

2018-10-03 08:44

null

老师，您回复 JK David 说到：

空间复杂度计算的不对，应该是O(1)，不是O(n)。我们要看额外的内存消耗，不是看链表本身存储需要多少空间。

在《复杂度分析（上）》提到：

第 3 行申请了一个大小为 n 的 int 类型数组，所以整段代码的空间复杂度就是 O(n)。

疑问一：
为什么两处计算空间复杂度的方法不一致，回复评论是以额外消耗为准，而文章中是以分配存储的空间为准?

疑问二：
《复杂度分析（上）》中介绍空间复杂度没有时间复杂度详细且带例子，看到老师您回复 JK David 以额外消耗为分析对象，就更懵圈了。
如果说：
1. 额外消耗常量值内存的空间复杂度是 O(1)；
2. 额外消耗 n 内存的空间复杂度是 O(n)；
那么空间复杂度是 O(n 的平方)，如何才能额外消耗 n 的平方空间呀?
老师能否针对空间复杂度，有一个更详细的说明和举例呢?

谢谢老师
2018-10-07 23:38


六六六
判断单链表是否是回文，只想到了这种low一些的做法，时间复杂度为O(n^2)：
public static boolean isHuiwen(LinkedList linkedList) {
Node first = linkedList.getFirst();
int size = linkedList.getSize();
Node head = null;
Node foot = null;
for (int a = 0; a < size / 2; a++) {
head = head == null ? first : head.next;
foot = head;
for (int i = a; i < size - 1 - a; i++) {
foot = foot.next;
}
if (!head.getData().equals(foot.getData())) {
return false;
}
}
return true;
}

2018-10-03 21:19


JStFs
LRU：活在当下。比如在公司中，一个新员工做出新业绩，马上会得到重用。

LFU：以史为镜。还是比如在公司中，新员工必须做出比那些功勋卓著的老员工更多更好的业绩才可以受到老板重视，这样的方式比较尊重“前辈”。
2018-10-04 09:20

作者回复
哈哈 形象!
2018-10-04 21:07


阳仔
学习反馈：
链表也是一种基础的线性表结构。由于它的很多特点跟数组是相反的，因此可以与数组一起对比着学习。
数组的存储空间是连续，而链表不是；数组可以通过寻址公式计算通过下标来访问，而链表访问元素需要遍历。
常见的链表有：

单链表、双向链表、循环链表、双向循环链表。
链表擅长插入、删除操作，时间复杂度为O(1)；查询的效率不高，时间复杂度为O(n)。
数组擅长通过下标随机访问元素，时间复杂度为O(1)；插入、删除的效率不高，时间复杂度为O(n)。
在实际项目开发中，选择数组或者链表不能只关注时间复杂度，还需要考虑具体业务，综合考虑选择数组还是链表。
了解了链表的数据结构，那么实现一个机遇链表数据结构的LRU算法就比较简单了：
从链表中查询此缓存数据是否存在：
1、如果存在，则删除该缓存数据节点，并把数据插入到链表头部的位置；
1、如果不存在，则也考虑两种情况：
1、如果缓存充足，则把数据插入到链表头部的位置；
2、如果缓存不足，则把链表中的末尾节点删除，再把缓存数据插入到头部。
思考题：
如果是只使用单链表的话，假设存储回文的链表是L1，再用一个链表L2来存储逆文；
我的思路是这样：
1、循环这个回文链表L1，在遍历到一半之前把逆文存在一个L2中；
例如L1 为A->B->C->B->A，那么遍历到一半时，L2为：B->A；
偶数和奇数的区别在与中间的节点要不要放在L2中。
2、继续遍历比较L1,L2两个链表各个元素是否相等，如果不相等则立即返回；如果比较到最后遍历结束，则说明是回文；
因此通过一次遍历就知道这个链表是否为回文。时间复杂度为O(n)。

2018-10-03 17:04

作者回复

2018-10-03 23:43

Smallfly

思考题：根据原有单链表回文创建一个逆向的单链表回文，while 循环遍历比较，复杂度为 O(N)。

2018-10-03 06:31

A漩

最近最少使用策略 LRU（Least Recently Used）
老师此处的例子是着重体现了”最近“的场景吧！
如果想体现”最少“是不是还得为LinkedList加个”访问次数“的属性？

2018-10-03 14:22

作者回复

字面上是你的理解 不过请百度百科一下LRU 我讲的没错呢

2018-10-03 23:54