链表（上）

数据结构与算法

链表的经典应用场景： LRU 缓存淘汰算法。

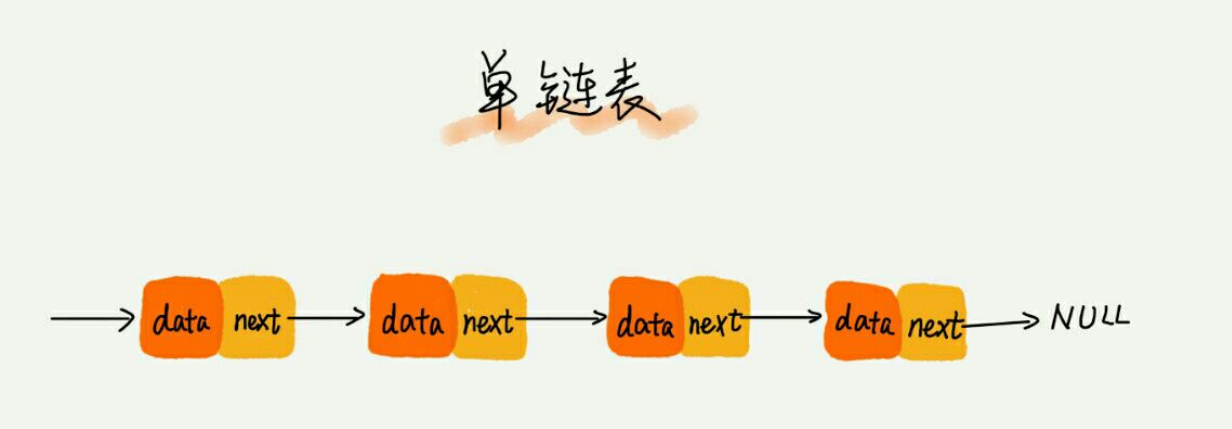
缓存是一种提高数据读取性能的计数，如常见的：CPU 缓存，数据库缓存，浏览器缓存等。

缓存的大小有限，当缓存被用满时，那些数据应该被清理出去，那些数据应该保留，这就需要缓存淘汰策略算法来决定。常见得策略有三种：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）。

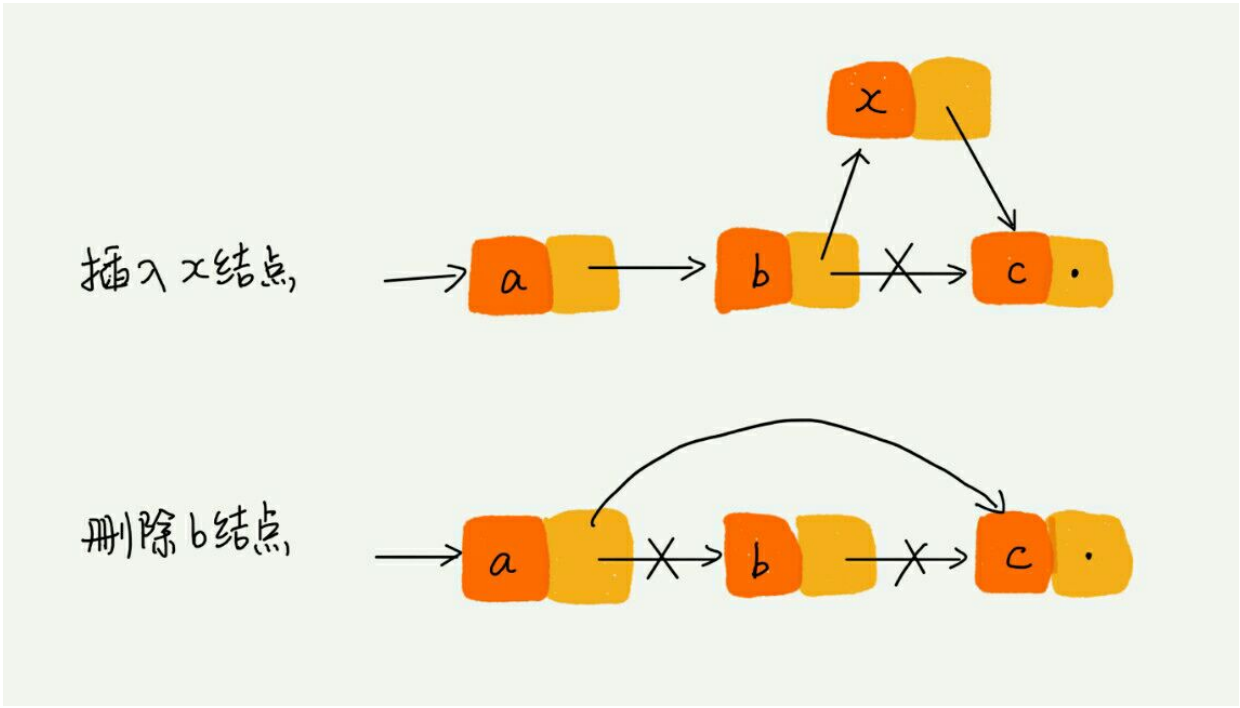
链表的结构

从底层的存储数据结构上看，数组需要连续的内存空间来存储，对内存的要求比较高，而链表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。

单链表有两个节点是特殊的，他们分别是第一个节点和最后一个节点，习惯性称之为头结点和尾节点，头结点用来记录链表的基地址，而尾节点特殊的地方是：指针不是指向下一个节点，而是指向一个空地址 NULL。

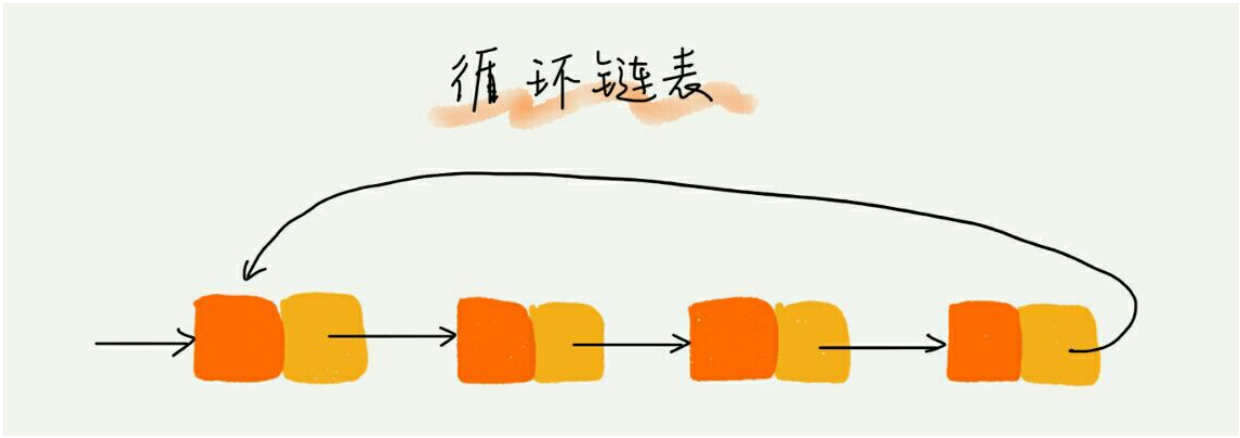


在链表中插入或者删除一个数据不需要像数组那样为了保存内存的连续性而搬移结点，所以在链表中插入和删除操作的时间复杂度为 $O(1)$。

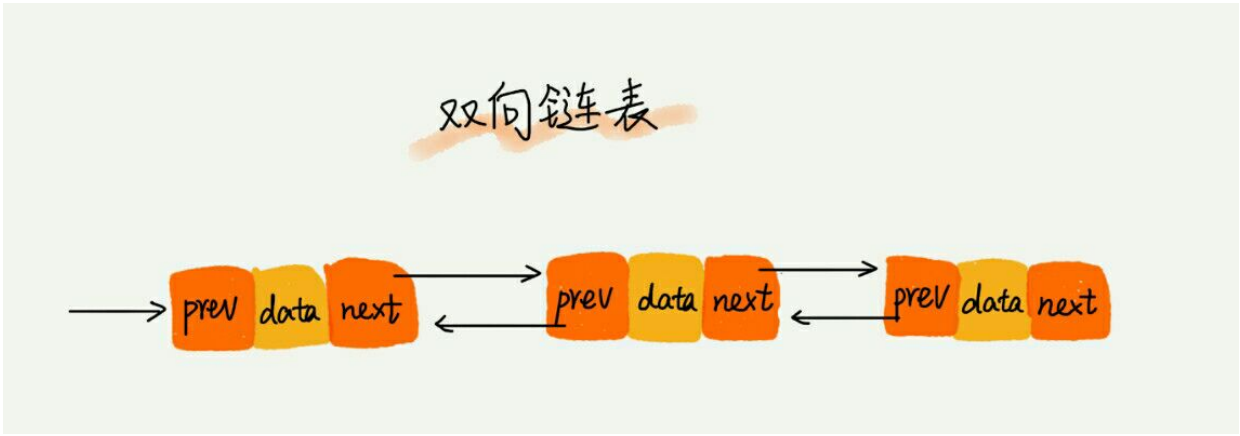


但是有利就有弊，链表药性随机访问第 k 个元素怒，就没有数组那么高效了，因为链表中的数据并非连续存储，所以无法像数组那样，根据首地址和下标，通过寻址公式就能直接计算出对应的内存地址，而是需要根据指针一个接一个结点的依次遍历，直到找到相应的结点。时间复杂度为$O(n)$。

循环链表是一种特殊的单链表，唯一区别就在为节点指针指向链表的头结点。



双向链表，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，显然需要更占用内存。



在实际的删除操作中，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

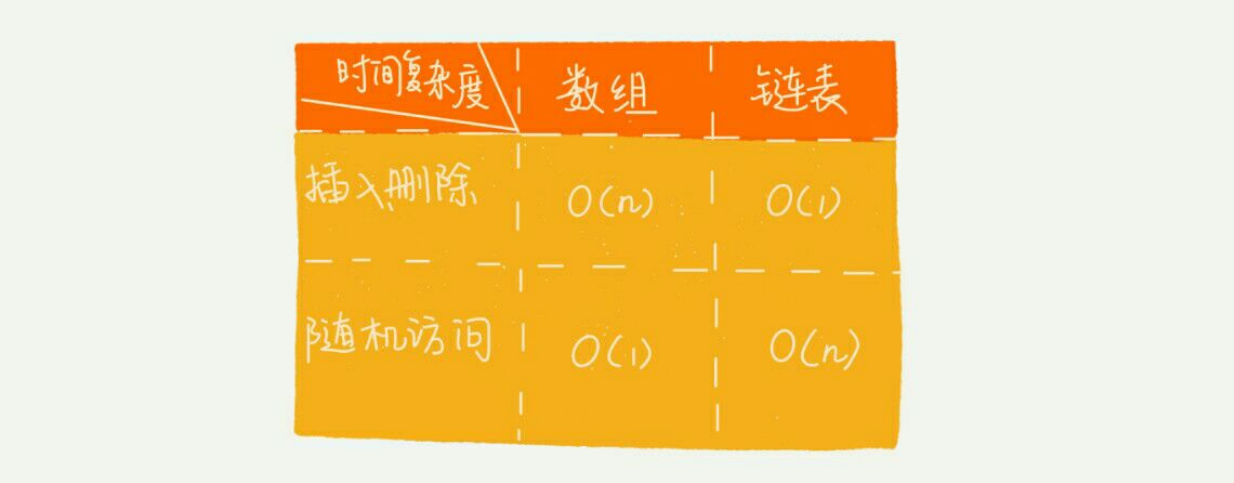
对于第一种情况，无论单链表还是双链表，都需要从头结点遍历一遍整个链表，直到找到值等于给定值的结点，将其删除，尽管删除的时间复杂度为 $O(1)$，但查找的时间复杂度为 $O(n)$，所以总的时间复杂度为 $O(n)$。

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，所以还是需要从头结点遍历，直到找到 p->next = q，才可进行删除，时间复杂度为 $O(n)$，而对于双向链表，泽科直接进行删除，时间复杂度为 $O(1)$。

同理如果我们希望在链表的某个指定结点前插入一个结点插入操作，双向链表可以在 $O(1)$ 时间复杂度内搞定，而单链表则需要 $O(n)$ 时间复杂度。

双向链表的重要思想是空间换时间，当内存空间充足时，如果更加追求代码的执行速度，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，相反同理。而缓存正是利用了空间换时间的设计思想。

链表和数组的比较



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，在实现上使用的连续的内存空间，可以借助 CPU 的缓存机制，预读数据，所以访问效率更高。而链表在内存中并不是连续存储的，所以不支持预读。

数组声明需要预先分配内存大小，而链表天然支持动态扩容。除此之外，如果代码对内存的使用非常苛刻，数组更加适合，因为链表中的每个结点都需要消耗额外的存储空间，而且，对链表进行频繁的掺入、删除操作，还会导致频繁的内存申请和释放，容易造成内存内存碎片。

链表实现 LRU 缓存淘汰算法

维护一个有序的单链表，越靠近链表的尾部的结点越早之前访问，当有一个新的数据被访问时，我们从链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，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，两种情况
    - 此时缓存未满，则将此节点直接插入到链表的头部。
    - 此时缓存已满，则链表尾结点删除，将新的数据节点插入链表头部。