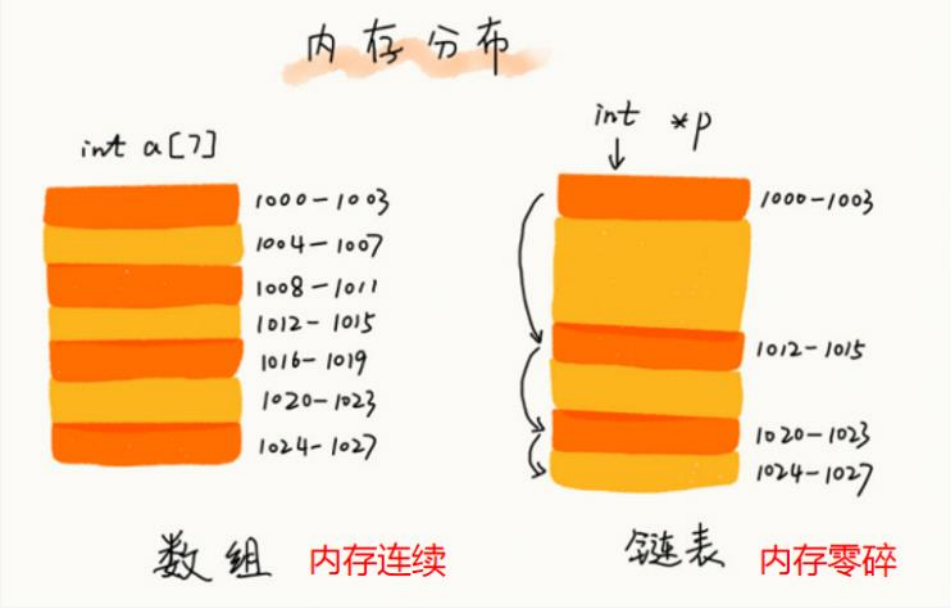
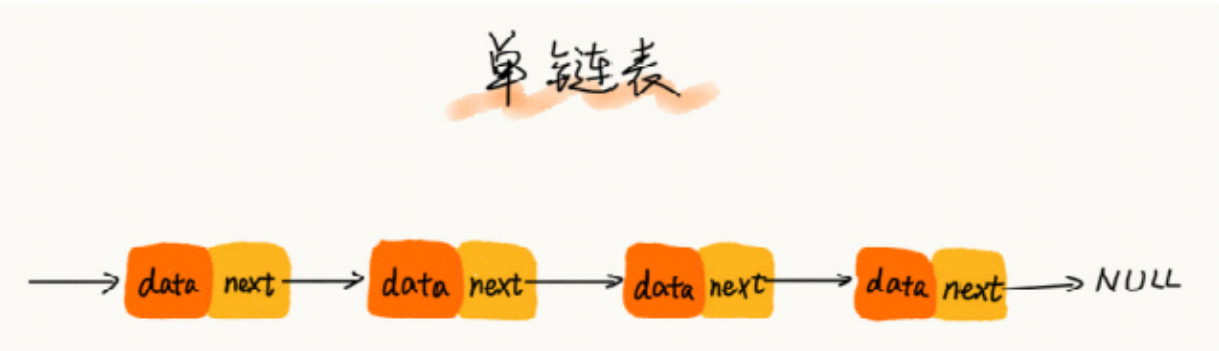
6-链表（上）

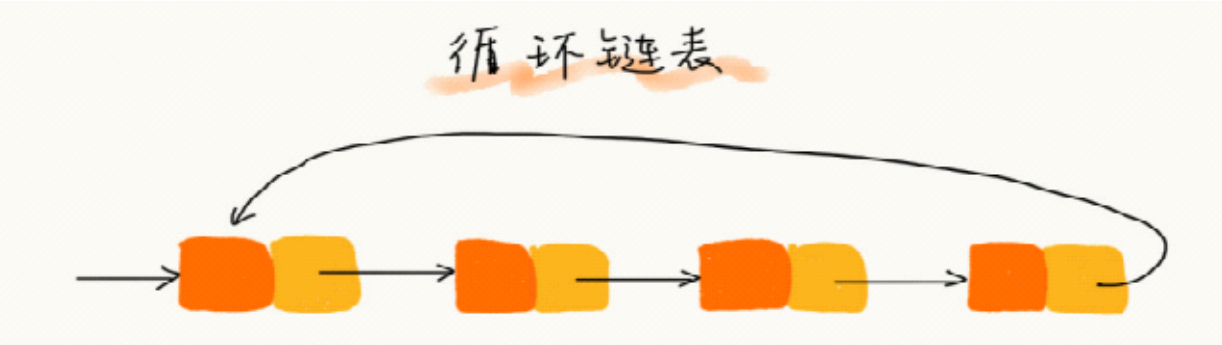
底层数据结构

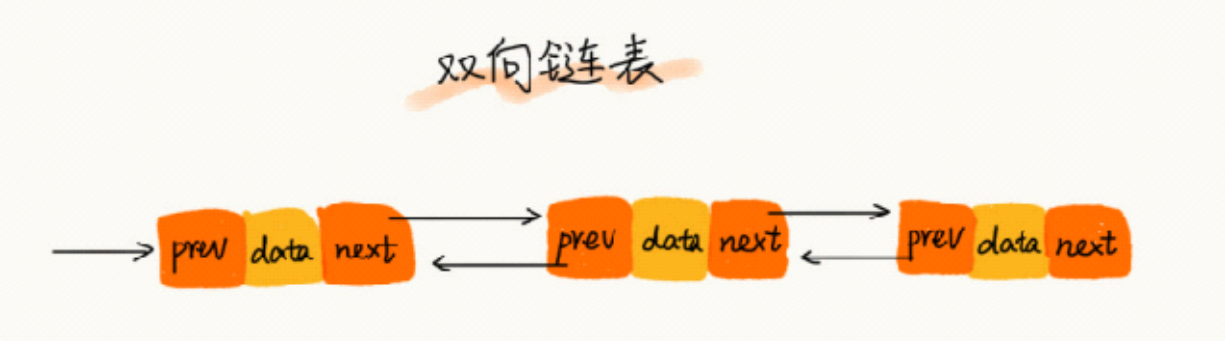


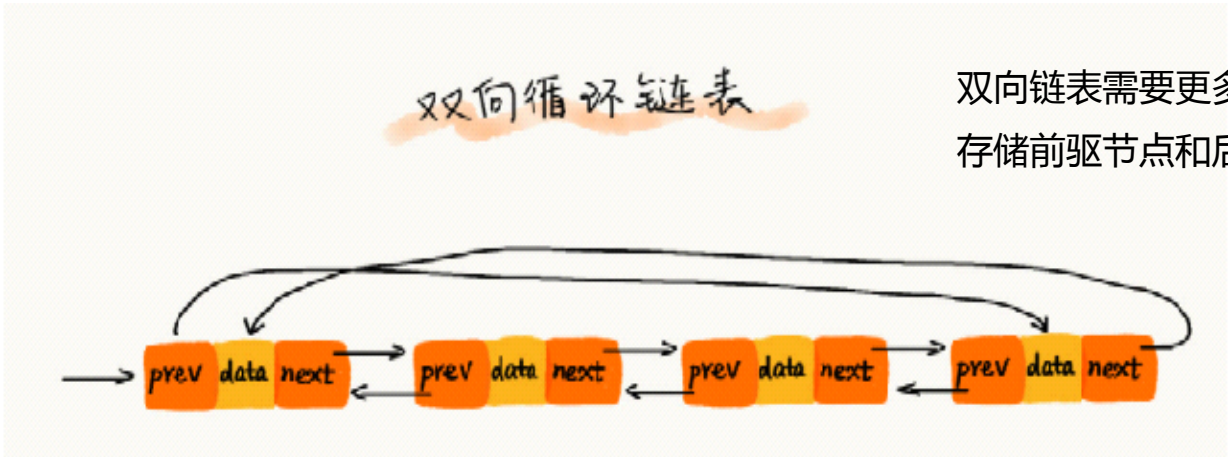
没有连续100MB内存，即使内存有100MB内存，数组申请内存会申请失败

链表结构

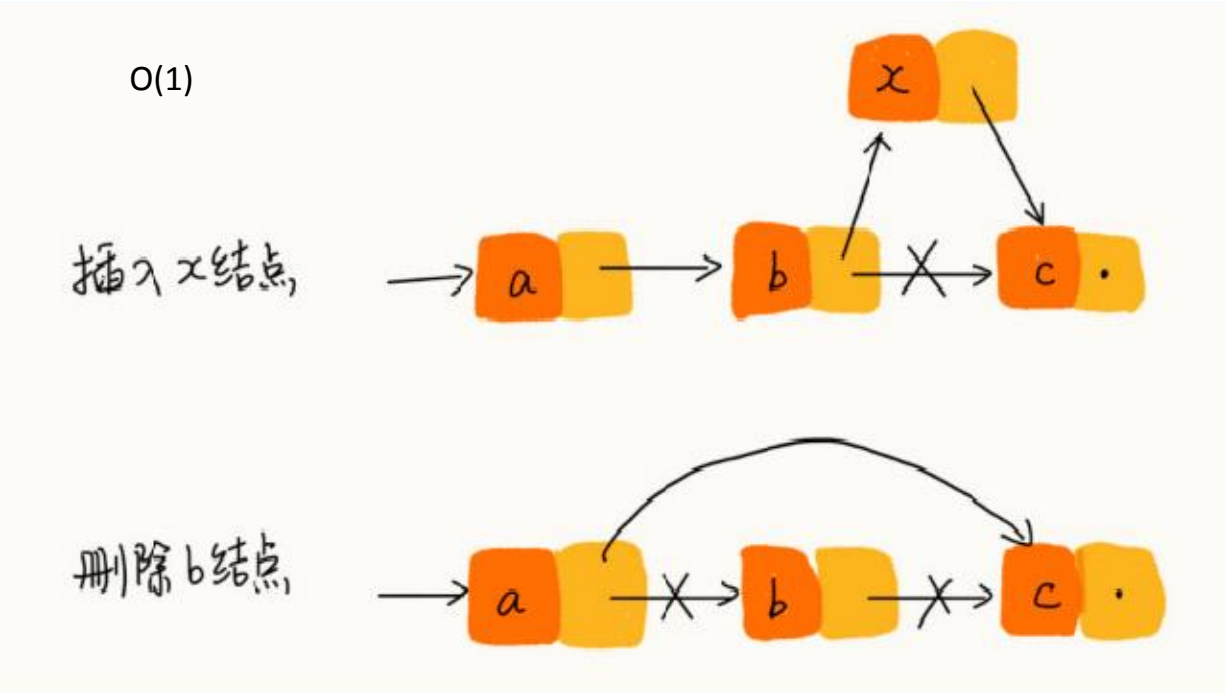








双向链表需要更多空间
存储前驱节点和后驱节点



	数组	链表
随机访问	根据K值寻址	遍历O(n)

单向链表 Vs 双向链表

	单向链表	双向链表
插入、删除等于给定值的某个结点	遍历 O(n) 删除 O(1) 加法法则 O(n)	遍历 O(n) 删除 O(1) 加法法则 O(n)
插入、删除给定指针的某个结点	遍历 O(n) 删除 O(1) 加法法则 O(n)	前驱结点 O(1) 删除 O(1) 加法法则 O(1)
对于有序链表，每次记录查询位置p，下次查寻根据p位置与值的大小关系，来判断是往前还是往后	全部数据	平均一半的数据
应用场景		LinkedHashMap

性能：链表 Vs 数组

	链表	数组
内存分配与访问	不需要连续内存	需要连续内存，支持CPU缓存，预先读取，访问效率高
	天然支持动态扩容	分配数组内存空间不足，数组或容器扩容会发生数据拷贝，耗时
插入、删除时间复杂度	虽为O(1)， 频繁的插入删除操作，容易造成内存碎片，导致频繁GC	O(n)
随机访问时间复杂度	O(n)	O(1)

缓存策略

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）

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？

链表实现LRU缓存淘汰算法

思路：

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。

   时间复杂度为 O(n)

链表实现LRU缓存淘汰代码实现

```
public class LRULinkedList {
    private int cap;
    private int len;
    private int count;
    private Node head;
    private Node temp;
    private Node tail;
    private List<Node> arrayList = new ArrayList<Node>();

    public LRULinkedList(int cap) {
        this.cap = cap;
    }

    public Node getNodeFromCache(int value) {
        if(head == null) {
            head = new Node(value, null);
            len++;
            return head;
        }
```

```
        temp = head;
        while(temp != null) {
            if (temp.getVal() == value) {
                return temp;
            }
            System.out.println("count " + count);
            if (count == cap - 2) {
                tail = temp;
            }
            temp = temp.getNext();
            count++;
        }
        count = 0;
        if (len < cap) {
            System.out.println("len < cap " + len);
            Node newHeader = new Node(value, head);
            head = newHeader;
            len++;
        } else {
            System.out.println("len > cap " + len);
            tail.setNext(null);
            len--;
            Node newHeader = new Node(value, head);
            head = newHeader;
            len++;
        }
        System.out.println("len " + len);
        return head;
    }
```

约瑟夫问题

约瑟夫是犹太军队的一个将军，在反抗罗马的起义中，他所率领的军队被击溃，只剩下残余的部队40余人，他们都是宁死不屈的人，所以不愿投降做叛徒。一群人表决说要死，所以用一种策略来先后杀死所有人。
于是约瑟夫建议：每次由其他两人一起杀死一个人，而被杀的人的先后顺序是由抽签决定的，约瑟夫有预谋地抽到了最后一签，在杀了除了他和剩余那个人之外

的最后一人，他劝服了另外一个没死的人投降了罗马。

假设问题是从n个人编号分别为0...n-1，取第k个，

则第k个人编号为k-1的淘汰，剩下的编号为 0,1,2,3...k-2,k,k+1,k+2...

此时因为从刚刚淘汰那个人的下一个开始数起，因此重新编号

把k号设置为0,则

k 0

k+1 1

...

0 n-k

1 n-k+1

假设已经求得了n-1个人情况下的最终胜利者保存在f[n-1]中，则毫无疑问，该胜利者还原到原来的真正编号即为 (f[n-1]+k)%n （因为第二轮重新编号的时候，相当于把每个人的编号都减了k，因此重新+k即可恢复到原来编号）。由此，我们可以想象，当最终只剩下一个人的时候，该人即为胜利者，此时重新编号，因为只有一个人，所以此时f[1]=0

LinkedHashMap 双向循环链表原理