Github代码：<https://github.com/wangzheng0822/algo>

# 02丨如何抓住重点，系统高效地学习数据结构与算法？

数据结构和算法有什么关系呢？为什么大部分书都把这两个东西放到一块儿来讲呢？

这是因为，数据结构和算法是相辅相成的。**数据结构是为算法服务的，算法要作用在特定的数据结构之上**

## 1.入门

想要学习数据结构与算法，**首先要掌握一个数据结构与算法中最重要的概念——复杂度分析。**

## **2.重点**

学习**20 个最常用的、最基础**数据结构与算法，**不管是应付面试还是工作需要，只要集中精力逐一攻克这 20 个知识点就足够了。**

10 个数据结构：数组、链表、栈、队列、散列表、二叉树、堆、跳表、图、Trie 树；

10 个算法：递归、排序、二分查找、搜索、哈希算法、贪心算法、分治算法、回溯算法、

动态规划、字符串匹配算法。

只要彻底掌握这个专栏的内容，就足以应对国内公司的技术面试，即便是 BAT 这样的公司。

## 3方法

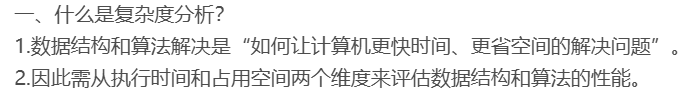
“边学边练”这一招非常有用。建议你每周花 1～2 个小时的时间，集中把这周的三节内容涉及的数据结构和算法，全都自己写出来，用代码实现一遍。

在学习数据结构和算法的过程中，你也要注意，不要只是死记硬背，不要为了学习而学习，而是**要学习它的“来历”“自身的特点”“适合解决的问题”以及“实际的应用场景”**

在学习的过程中，一定会碰到“拦路虎”。如果哪个知识点没有怎么学懂，不要着急，这是正常的。因为，想听一遍、看一遍就把所有知识掌握，这肯定是不可能的。**学习知识的过程是反复迭代、不断沉淀的过程。**

# **复杂度**

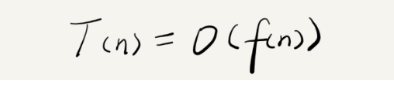
## 03 | 复杂度分析（上）：如何分析、统计算法的执行效率和资源消耗？



### 时间复杂度

#### 公式

**所有代码的执行时间 T(n) 与每行代码的执行次数 n 成正比**。把这个规律总结成一个公式



T(n)：表示代码执行的时间；

n : 表示数据规模的大小；

f(n) : 表示每行代码执**行的次数总和**。

公式中的 O，表示代码的执行时间 T(n) 与 f(n) 表达式成正比。大O 时间复杂度实际上并不具体表示代码真正的执行时间，而是表示**代码执行时间随数据规模增长的变化趋势**，所以，也叫作**渐进时间复杂度**（asymptotic time complexity），简称**时间复杂度**。

#### 案例

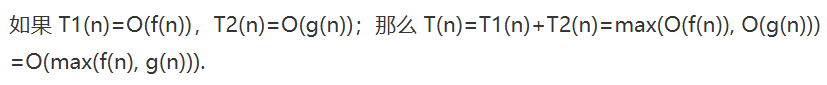
第一个例子中的 T(n) = O(2n+2)，第二个例子中的 T(n) = O(2n2+2n+3)。这就是**大O时间复杂度表示法**。

公式中的**低阶、常量、系数三部分**并不左右增长趋势，所以都可以忽略。我们只需要记录一个最大量级就可以了，用大O 表示法记为：

第一个例子：T(n) = O(n)； 第二个例子：T(n) = O(n2)。

### 时间复杂度分析

1. **只关注循环执行次数最多的一段代码**
2. **加法法则：总复杂度 等于 量级最大的那段代码的复杂度**



1. **乘法法则：嵌套代码的复杂度 等于 嵌套内外代码复杂度 的 乘积**

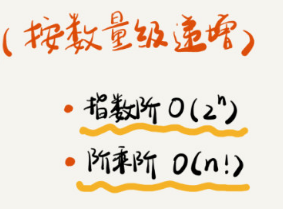


**eg:**

假设T1(n) = O(n)，T2(n) = O(n2)，则 T1(n) \* T2(n) = O(n3)。

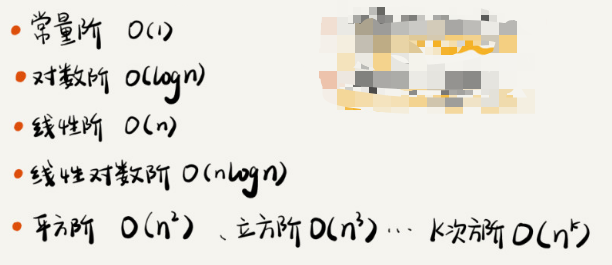
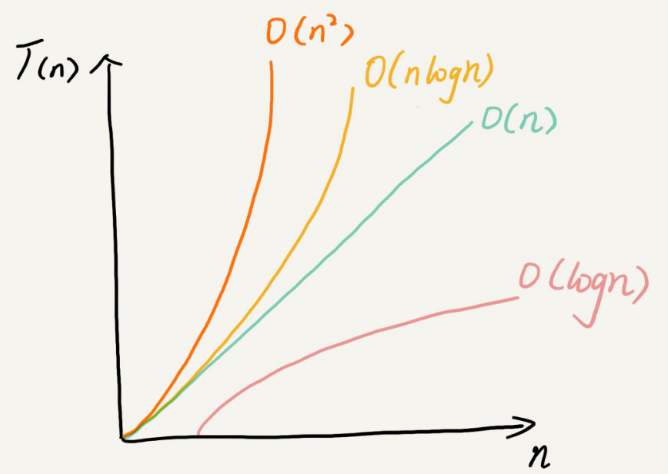
### 时间复杂度-**非多项式量级**

只有两个：O(2n) 和 O(n!)。

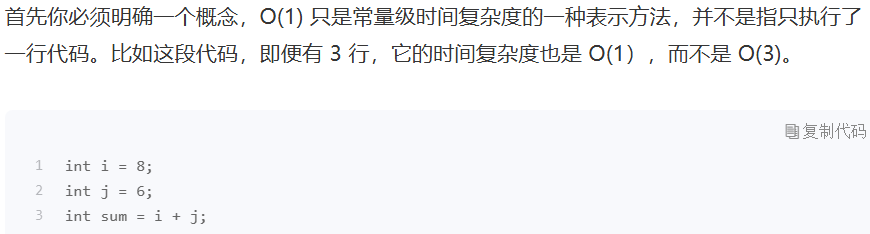


当数据规模 n 越来越大时，非多项式量级算法的执行时间会急剧增加，求解问题的执行时间会无限增长。所以，非多项式时间复杂度的算法其实是***非常低效的算法。***

### 时间复杂度-**多项式量级**

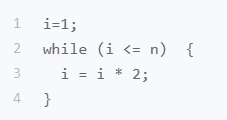
 

#### **1**.O(1)



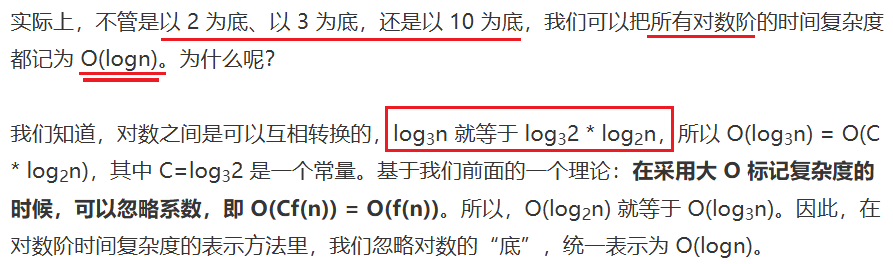
**一般情况下，只要算法中不存在循环语句、递归语句，即使有成千上万行的代码，其时间复杂度也是Ο(1)**。

#### ****2.O(logn)****

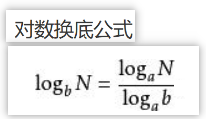
 

只要知道 x 值是多少，就知道这行代码执行的次数了。通过 2x=n 求解得出，x=log2n。所以，这段代码的时间复杂度就是 O(log2n)。

###### 为什么都记为 O(logn)？



证明：

，log3n 就等于 log32 \* log2n

#### **3.O(nlogn)**

还记得我们刚讲的乘法法则吗？

如果一段代码的时间复杂度是 O(logn)，我们循环执行 n 遍，时间复杂度就是 O(nlogn) 了。而且，O(nlogn) 也是一种非常常见的算法时间复杂度。比如，归并排序、快速排序的时间复杂度都是 O(nlogn)。

### 时间/空间复杂度

1.时间复杂度的

全称是**渐进时间复杂度**,**表示算法的执行时间与数据规模之间的增长关系**。

2.空间复杂度

全称是**渐进空间复杂度,表示算法的存储空间与数据规模之间的增长关系**。常见的空间复杂度就是 O(1)、O(n)、O(n2 )

## 04 | 复杂度分析（下）：浅析最好、最坏、平均、均摊时间复杂度

大部分情况下，我们并不需要区分最好、最坏、平均三种复杂度。平均复杂度只在某些特殊情况下才会用到，而**均摊时间复杂度**应用的场景比它更加特殊、更加有限。

最好情况时间复杂度（best case time complexity）

在最理想的情况下，执行这段代码的时间复杂度。

最坏情况时间复杂度（worst case time complexity）

在最糟糕的情况下，执行这段代码的时间复杂度。

平均情况时间复杂度（average case time complexity）

全称应该叫 **加权平均时间复杂度** 或者 **期望时间复杂度**。

### 均摊时间复杂度

**分析方法：**

在数组中插入数据的这个例子。每一次 O(n) 的插入操作，都会跟着 n-1 次 O(1) 的插入操作，所以把耗时多的那次操作均摊到接下来的 n-1 次耗时少的操作上，均摊下来，这一组连续的操作的均摊时间复杂度就是 O(1)。这就是均摊分析的大致思路。

**前置条件：**

对一个数据结构进行一组连续操作中，大部分情况下时间复杂度都很低，只有个别情况下时间复杂度比较高，而且这些操作之间存在前后连贯的时序关系。

**分析思路：**

只有在上述这种时候，我们就可以将这一组操作放在一块儿分析，看是否能将较高时间复杂度那次操作的耗时，平摊到其他那些时间复杂度比较低的操作上。

**得出结论：**

而且，在能够应用均摊时间复杂度分析的场合，一般**均摊时间复杂度**就等于**最好情况时间复杂度**。

只有在上述这种时

# 线性表

## 05 | 数组：为什么很多编程语言中数组都从0开始编号？

**线性表就**

是数据排成像一条线一样的结构。每个线性表上的数据最多只有前和后两个方向。其实除了数组，链表、队列、栈等也是线性表结构。

**非线性表**

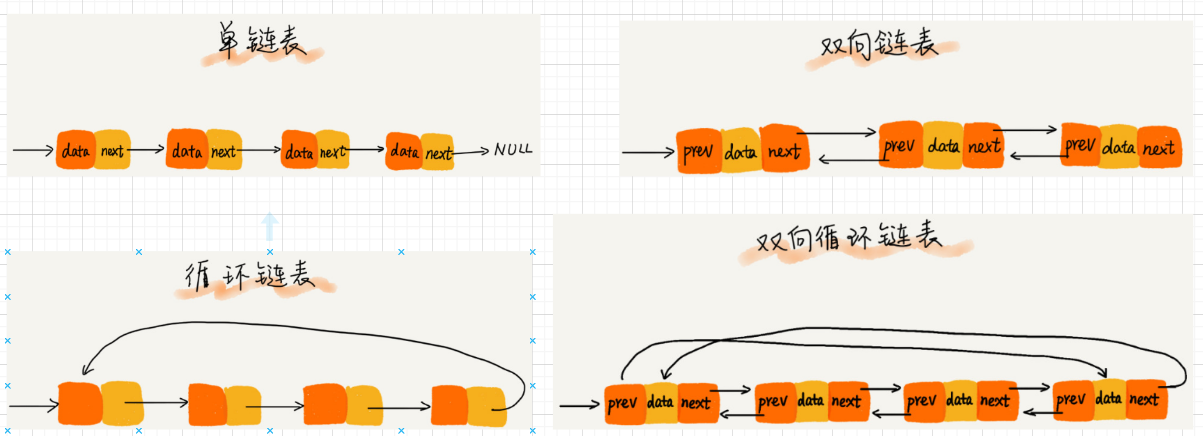
比如二叉树、堆、图等。之所以叫非线性，是因为，在非线性表中，数据之间并不是简单的前后关系。

数组（Array）是一种线性表数据结构。它用一组连续的内存空间，来存储一组具有相同类型的数据。

## 06丨链表（上）：如何实现LRU缓存淘汰算法？

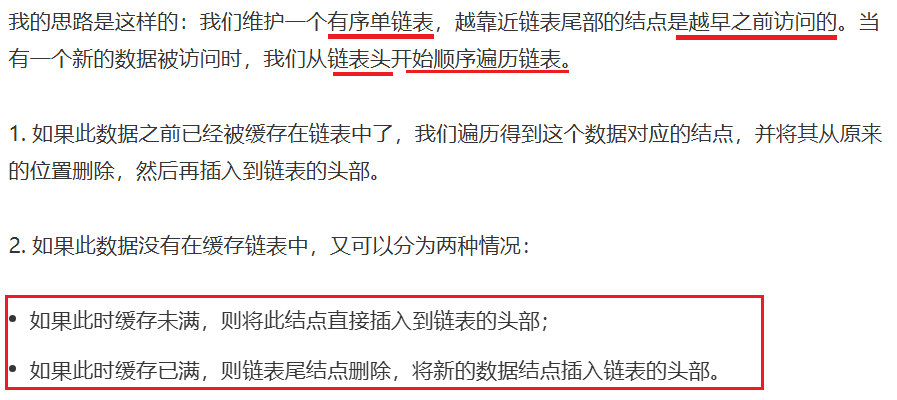
当缓存被用满时，哪些数据应该被清理出去，哪些数据应该被保留？常见的策略有三种：

1. **先进先出策略 FIFO（First In，First Out）。**
2. **最少使用策略 LFU（Least Frequently Used）**。【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”。】
3. **最近最少使用策略 LRU（Least Recently Used）**。【就像活在当下。比如在公司中，一个新员工做出新业绩，马上会得到重用。】



**如何基于链表实现 LRU 缓存淘汰算法？**

**链表头插入尾部删除**



**链表头删除 表尾部插入**

如何通过链表实现 LRU 缓存淘汰算法:

我们需要维护一个**按照访问时间**从大到小有序排列**的链表结构**。因为缓存大小有限，当缓存空间不够，**需要淘汰一个数据的时候**，我们就**直接将**链表头部的结点删除。

当要缓存某个数据的时候，**先在链表中查找这个数据**。如果没有找到，则直接将数据放到**链表的尾部**；如果找到了，我们就把它移动到**链表的尾部**。因为查找数据需要遍历链表，所以单纯用链表实现的 LRU 缓存淘汰算法的时间复杂很高，是 O(n)。

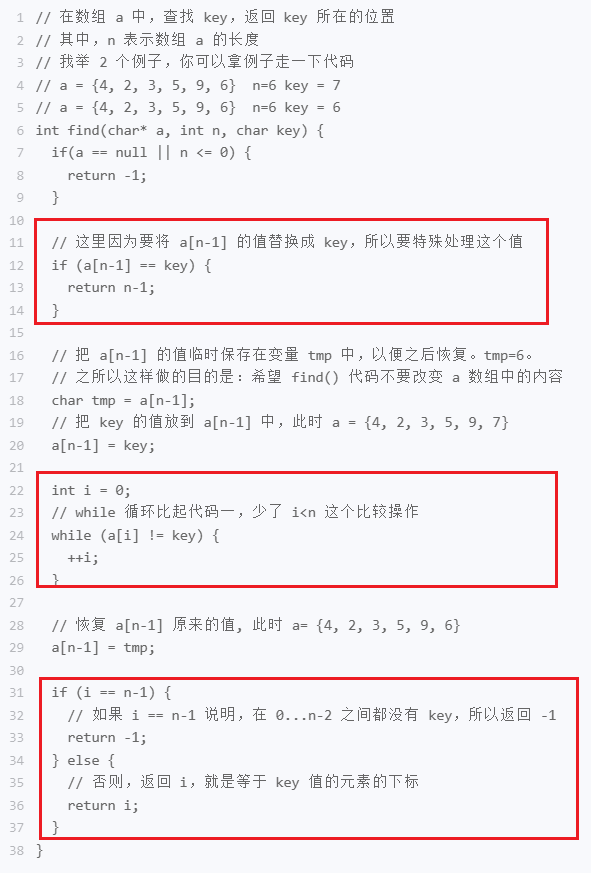
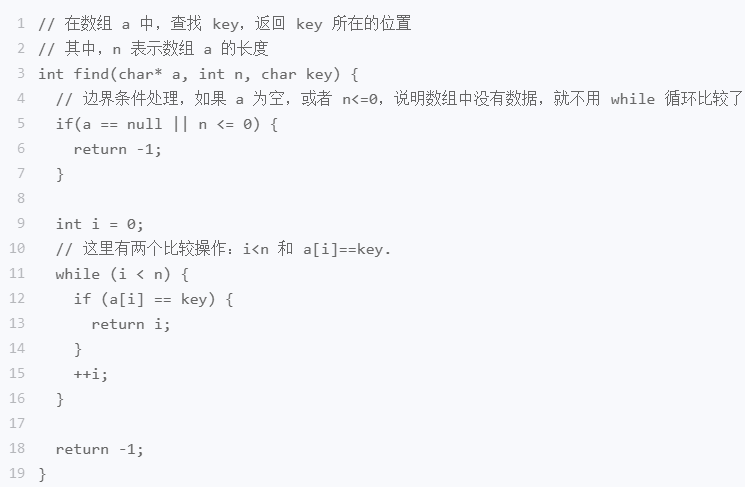
## 07丨链表（下）：如何轻松写出正确的链表代码？

**针对链表的插入、删除操作，需要对插入第一个结点和删除最后一个结点的情况进行特殊处理。**这样代码实现起来就会很繁琐，不简洁，而且也容易因为考虑不全而出错。如何来解决这个问题呢？

哨兵就要登场了。**哨兵，解决的是国家之间的边界问题**。同理，这里说的哨兵也是解决“边界问题”的，不直接参与业务逻辑。

还记得如何表示一个空链表吗？head=null 表示链表中没有结点了。其中 head 表示头结点指针，指向链表中的第一个结点。

如果我们引入哨兵结点，在任何时候，不管链表是不是空，head 指针都会一直指向这个哨兵结点。我们也把这种有哨兵结点的链表叫**带头链表**。相反，没有哨兵结点的链表就叫作**不带头链表**。



## 08丨栈：如何实现浏览器的前进和后退功能？

栈是一种“操作受限”的线性表。当某个数据集合只涉及在一端插入和删除数据，并且满足后进先出、先进后出的特性，我们就应该首选“栈”这种数据结构。

**如何实现一个表达式求值?**

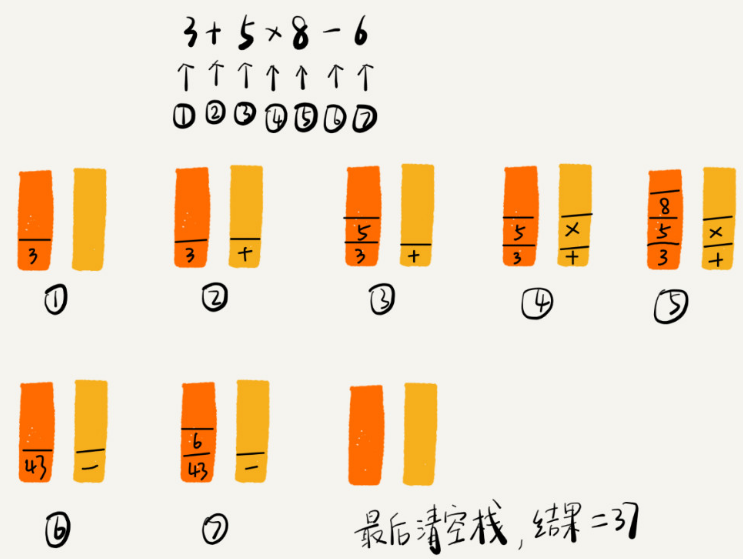
### 方案

通过两个栈来实现的。其中一个保存**操作数的栈**，另一个是保存**运算符的栈**。我们从左向右遍历表达式，当遇到数字，直接压入操作数栈；当遇到运算符，就与运算符栈的栈顶元素进行比较。

如果比运算符栈顶元素的**优先级高**，就将当前运算符压入栈；如果比运算符栈顶元素的优先级低或者相同，从运算符栈中取栈顶运算符，从操作数栈的栈顶取 2 个操作数，然后进行计算，再把计算完的结果压入操作数栈，继续比较。

### 案例

运算级别：乘、除 高于 加、减



## 09丨队列：队列在线程池等有限资源池中的应用

用数组实现的栈叫作顺序栈，用链表实现的栈叫作链式栈。

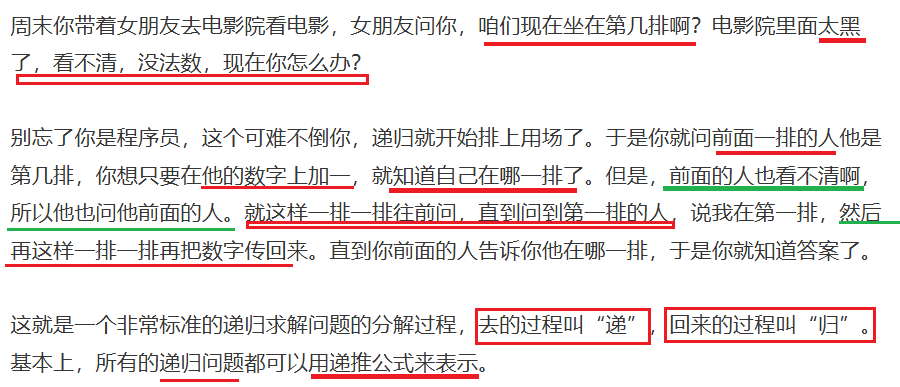
用数组实现的队列叫作**顺序队列**，用链表实现的队列叫作**链式队列**。

# 算法

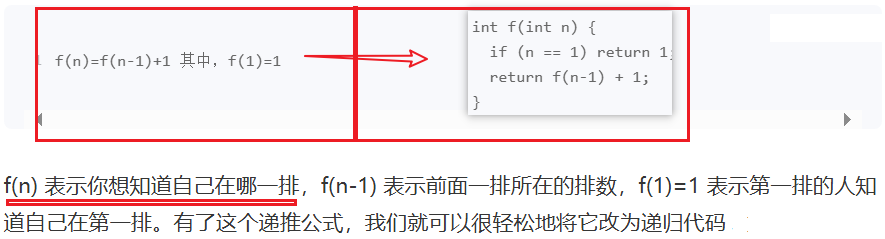
## 10丨递归：如何用三行代码找到“最终推荐人”？

递归：去的过程叫“递”，回来的过程叫“归”。

### 案例1



**公式**



### 递归需要满足的三个条件

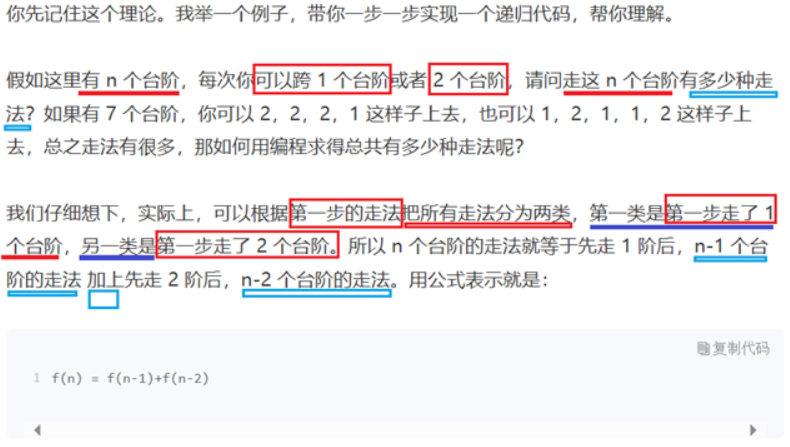
**只要同时满足以下三个条件，就可以用递归来解决。**

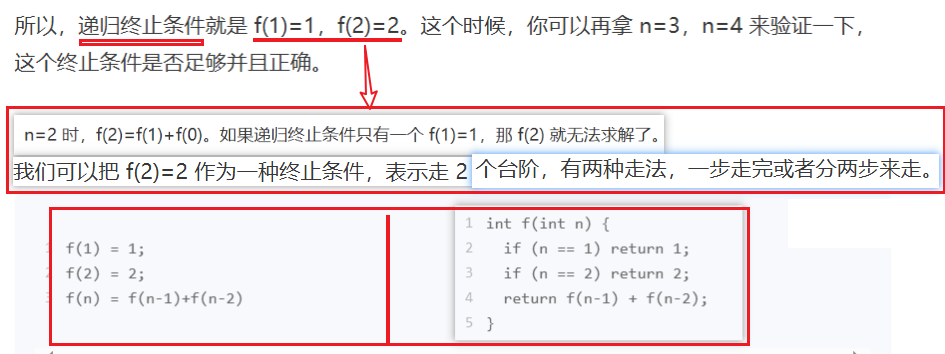


### 如何编写递归代码

**写递归代码的关键就是找到如何将大问题分解为小问题的规律，并且基于此写出递推公式，然后再推敲终止条件，最后将递推公式和终止条件翻译成代码**。

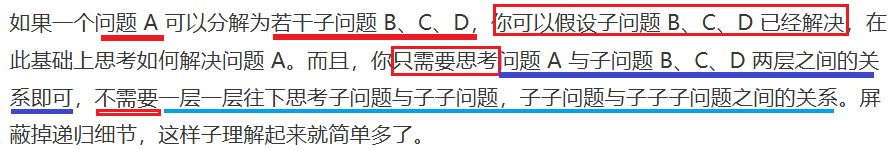
#### 案例2





### 如何已正确的思维方式去理解递归？

**编写递归代码的关键是，只要遇到递归，我们就把它抽象成一个递推公式，不用想一层层的调用关系，不要试图用人脑去分解递归的每个步骤**。



### 递归优缺点

利是：递归代码的表达力很强，写起来非常简洁；

弊是：空间复杂度高、有堆栈溢出的风险、存在重复计算、过多的函数调用会耗时较多等问题。

## 11丨排序（上）：数组实现-比较排序

### 排序算法分析方法

#### 1.执行效率

1. 最好情况、最坏情况、平均情况时间复杂度

2. 时间复杂度的系数、常数 、低阶

3. 比较次数和交换（或移动）次数

#### 2.内存消耗-原地排序

**原地排序（Sorted in place）**。就是特指空间复杂度是 O(1) 的排序算法.

#### 3.稳定性

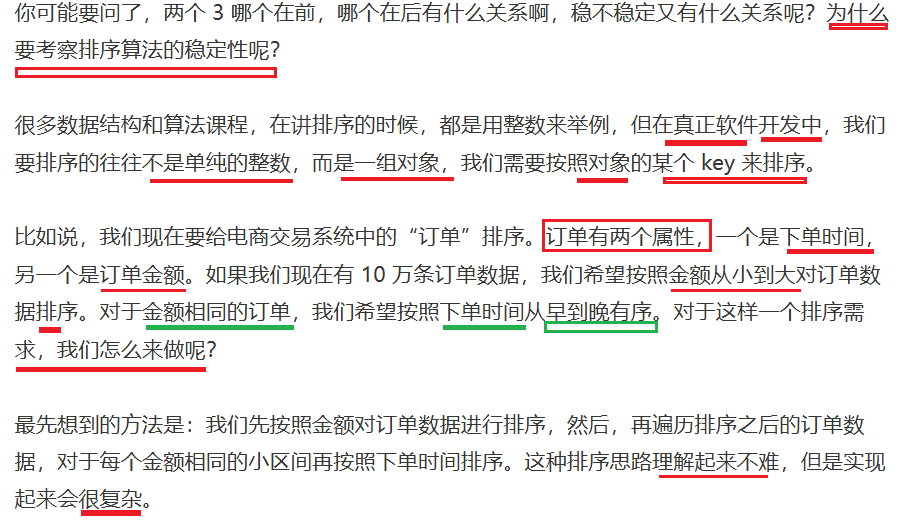
稳定性。这个概念是说，如果待排序的序列中存在值相等的元素，经过排序之后，相等元素之间原有的先后顺序不变。

eg: 我们有一组数据 2，9，3，4，8，3，按照大小排序之后就是 2，3，3，4，8，9。这组数据里有两个 3。经过某种排序算法排序之后，

如果**两个 3 的前后顺序没有改变**，那我们就把这种排序算法叫作稳定的排序算法；

如果前后顺序发生变化，那对应的排序算法就叫作不稳定的排序算法。

#### 4.排序算法为什么要有稳定性





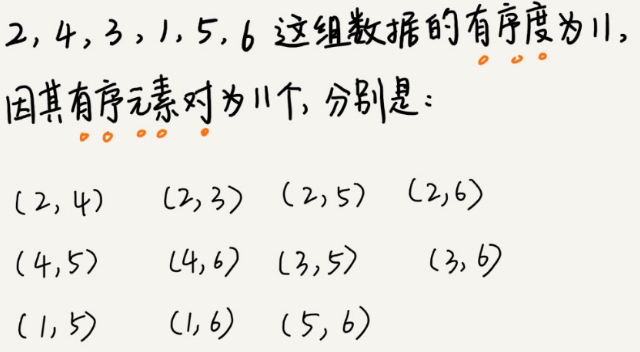
### 分析-平均情况下的时间复杂度

如果用**概率论方法**定量分析平均情况下的时间复杂度，涉及的数学推理和计算就会很复杂。还有一种思路，通过“有序度”和“逆序度”这两个概念来进行分析。

#### 1.有序度

是数组中具有**有序关系的**元素对的个数。

数学表达式：有序元素对：a[i] <= a[j], 如果 i < j。



对于一个**倒序排列的数组**，比如 6，5，4，3，2，1，**有序度是 0；**

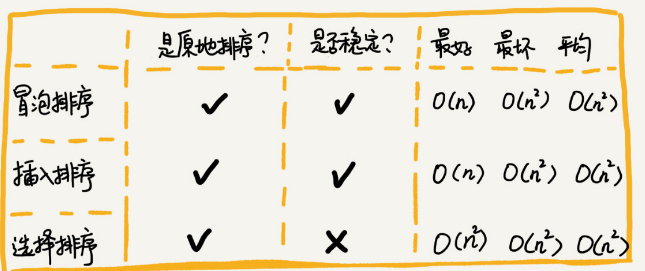
#### 2.满有序度

对于一个完全有序的数组，比如 1，2，3，4，5，6，有序度就是**n\*(n-1)/2**，也就是 15。我们把这种完全有序的数组的有序度叫作满有序度。

#### 3.逆序度

逆序度 = 满有序度 - 有序度。

### 总结



冒泡排序、插入排、选择排序 都是 基于数组实现的。

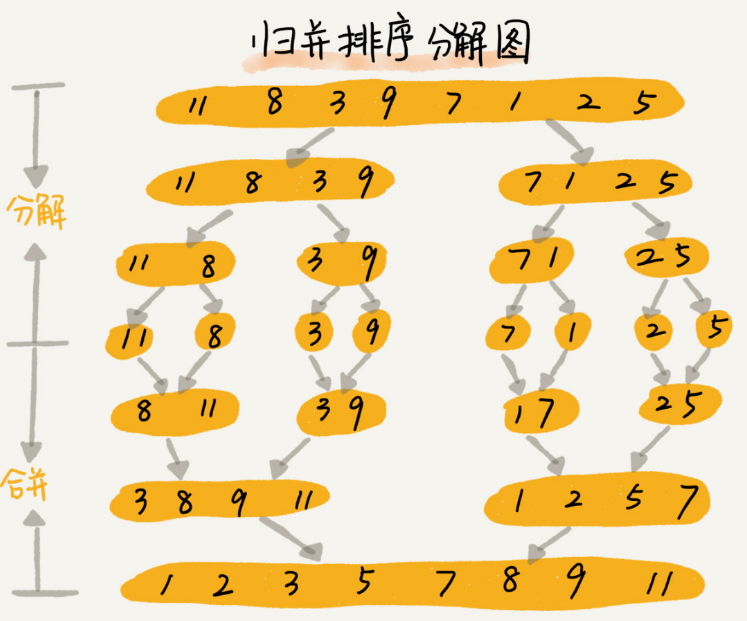
## 12丨排序（下）：递归实现-比较排序

### 归并排序（Merge Sort）

归并排序的核心思想还是蛮简单的。如果要排序一个数组，我们先把数组从中间分成前后两部分，然后对前后两部分分别排序，再将排好序的两部分合并在一起，这样整个数组就都有序了。

归并排序使用的就是分治思想。分治，顾名思义，就是分而治之，将一个大问题分解成小的子问题来解决。小的子问题解决了，大问题也就解决了。

分治思想跟我们前面讲的递归思想很像。是的，分治算法一般都是用递归来实现的。分治是一种解决问题的处理思想，递归是一种编程技巧，这两者并不冲突



### 归并排序的时间复杂度推到

#### 1.分析方法

如果我们定义求解问题 a 的时间是 T(a)，求解问题 b、c 的时间分别是 T(b) 和 T( c)，那我们就可以得到这样的递推关系式：**T(a) = T(b) + T(c) + K**

其中 **K 等于**将两个子问题 b、c 的结果合并成问题 a 的结果所消耗的时间。

#### 2.计算方法

我们假设对 n 个元素进行归并排序需要的时间是 T(n)，那分解成两个子数组排序的时间都是 T(n/2)。我们知道，merge() 函数合并两个有序子数组的时间复杂度是 O(n)。所以，套用前面的公式，归并排序的时间复杂度的计算公式就是：

T**(1) = C； n=1 时，只需要常量级的执行时间，所以表示为 C。**

**T(n) = 2\*T(n/2) + n； n>1**

#### 3.复杂度计算

T(n) = 2\*T(n/2) + n

= 2\*(2\*T(n/4) + n/2) + n = 4\*T(n/4) + 2\*n

= 4\*(2\*T(n/8) + n/4) + 2\*n = 8\*T(n/8) + 3\*n

= 8\*(2\*T(n/16) + n/8) + 3\*n = 16\*T(n/16) + 4\*n

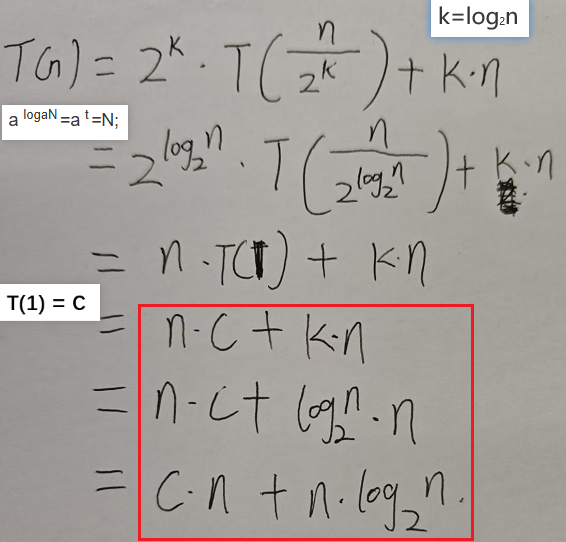
......

= **2^k \* T(n/2^k) + k \* n**

......

#### 4.推到过程

已知：当 T(n/2^k)=T(1) 时，也就是 n/2^k=1，我们得到 k=log2n 。



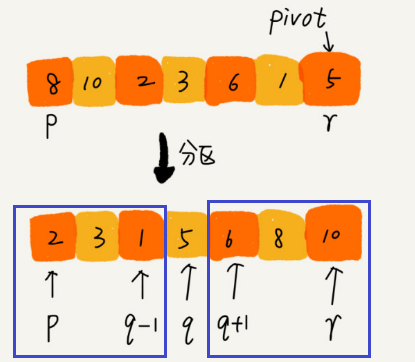
最终得到 **T(n)=Cn+nlog2n** 。如果我们用大 O 标记法来表示的话，T(n) 就等于 O(nlogn)。所以**归并排序的时间复杂度**是 O(nlogn)。

### 快速排序算法（Quicksort），

习惯性称为“快排”。快排利用的也是分治思想。

快排的思想是这样的：如果要排序数组中下标从 p 到 r 之间的一组数据，我们选择 p 到 r 之间的任意一个数据作为 pivot（分区点）。

我们遍历 p 到 r 之间的数据，将小于 pivot 的放到左边，将大于 pivot 的放到右边，将 pivot 放到中间。经过这一步骤之后，数组 p 到 r 之间的数据就被分成了三个部分，前面 p 到 q-1 之间都是小于 pivot 的，中间是 pivot，后面的 q+1 到 r 之间是大于 pivot 的。



#### 如何来选择分区点呢？

最理想的分区点是：被分区点分开的两个分区中，数据的数量差不多。

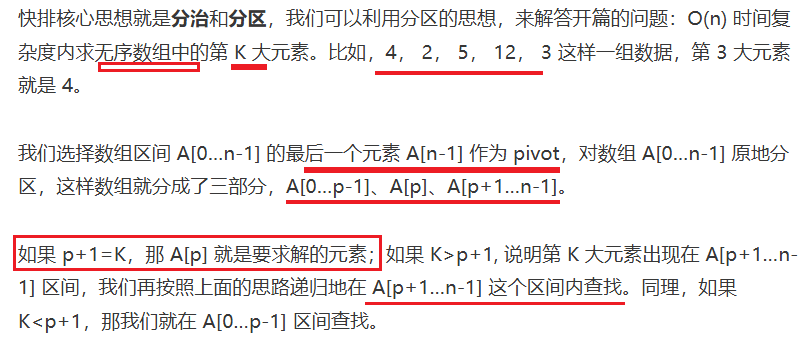
**1. 三数取中法**

从区间的首、尾、中间，分别取出一个数，然后对比大小，取这 3 个数的中间值作为分区点。如果要排序的数组比较大，那“三数取中”可能就不够了，可能要“五数取中”或者“十数取中”。

**2. 随机法**

随机法就是每次从要排序的区间中，随机选择一个元素作为分区点。

### 如何用快排思想在O(n)内查找第K大元素？



## 13丨线性排序：非比较排序-给100万用户数据排序？

三种时间复杂度是 O(n) 的排序算法：**桶排序、计数排序、基数排序。**因为这些排序算法的时间复杂度是线性的，所以我们把这类排序算法叫作线性排序（Linear sort）。

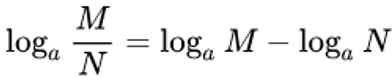
之所以能做到线性的时间复杂度，主要原因是，这三个算法是**非基于**【比较的排序算法】，都*不涉及元素之间的比较操作*。

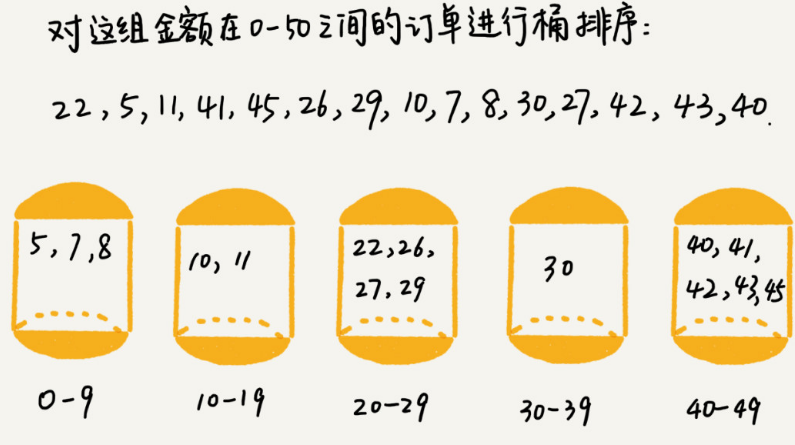
**重点的是掌握这些排序算法的适用场景。**

### 桶排序（Bucket sort）

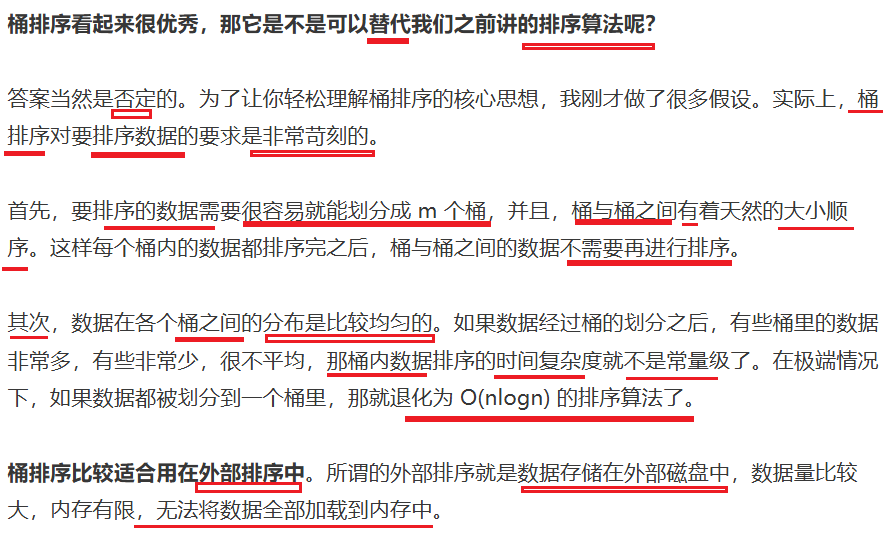
桶排序的时间复杂度为什么是 O(n) 呢？我们一块儿来分析一下。

如果要排序的数据有 n 个，我们把它们均匀地划分到 m 个桶内，每个桶里就有 k=n/m 个元素。**每个桶内部**使用**快速排序**，时间复杂度为 O(k \* logk)。m 个桶排序的时间复杂度就是 O(m \* k \* logk)，因为 k=n/m，所以整个桶排序的时间复杂度就是 O(n\*log(n/m))。

当桶的个数 m 接近数据个数 n 时，log(n/m) 就是一个非常小的常量，这个时候桶排序的时间复杂度接近 O(n)。



#### 使用场景



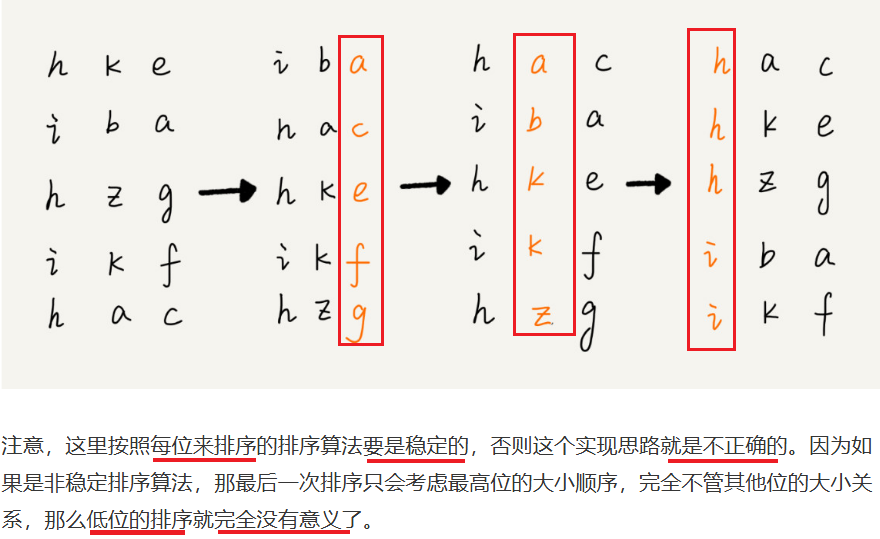
### 计数排序（Counting sort）

**计数排序其实是桶排序的一种特殊情况**。当要排序的 n 个数据，所处的**范围并不大**的时候，比如最大值是 k，我们就可以把数据划分成 k 个桶。每个桶内的数据值都是相同的，省掉了桶内排序的时间。



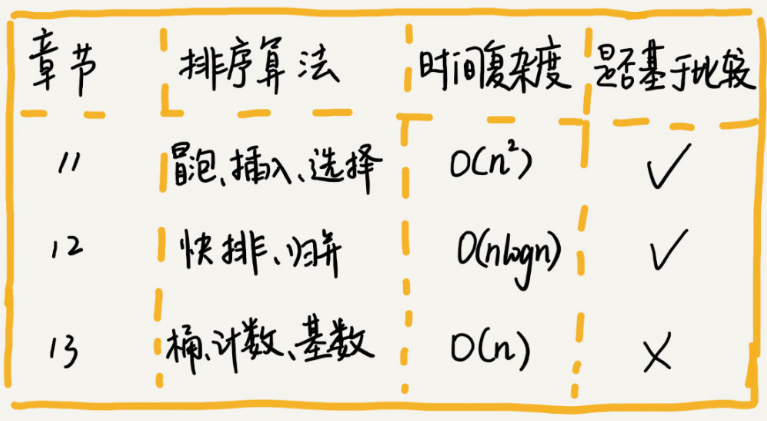
**计数排序只能用在数据范围不大的场景中，如果数据范围 k 比要排序的数据 n 大很多，就不适合用【计数排序】了。而且，计数排序只能给非负整数排序，如果要排序的数据是其他类型的，要将其在不改变相对大小的情况下，转化为非负整数。**

### 基数排序（Radix sort）

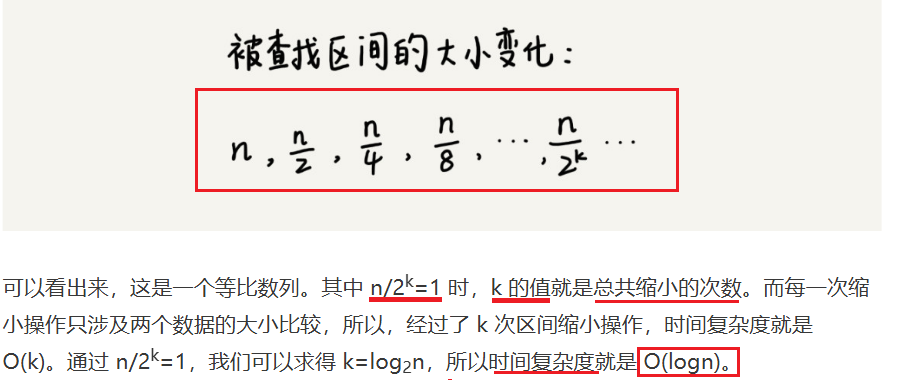


基数排序对要排序的数据是有要求的，**需要可以分割出独立的“位”来比较**，而且位之间**有递进的关系**，如果 a 数据的高位比 b 数据大，那剩下的低位就不用比较了。除此之外，**每一位的数据范围不能太大**，要可以用线性排序算法来排序，否则，基数排序的时间复杂度就无法做到 O(n) 了。

## 14丨排序优化：如何实现一个通用的、高性能的排序函数？



## 15丨二分查找（上）：如何用最省内存的方式实现快速查找功能？



### 二分查找应用场景的局限性

首先，二分查找依赖的是**顺序表结构，简单点说就是数组**。

其次，二分查找针对的是有序数据。

再次，数据量太小**不适合**二分查找

最后，数据量太大也不适合二分查找。

二分查找的底层需要依赖数组这种数据结构，而数组为了支持随机访问的特性，**要求内存空间连续，对内存的要求比较苛刻**。

大部分情况下，用二分查找可以解决的问题，用散列表、二叉树都可以解决。不管是散列表还是二叉树，都会需要比较多的额外的内存空间。如果用**散列表或者二叉树**来存储这 1000 万的数据，用 100MB 的内存肯定是存不下的。

而二分查找底层依赖的是数组，除了数据本身之外，**不需要额外存储其他信息**，是最省内存空间的存储方式，所以刚好能在限定的内存大小下解决这个问题。

## 16丨二分查找（下）：如何快速定位IP对应的省份地址？

凡是用二分查找能解决的，绝大部分我们更倾向于用散列表或者二叉查找树。即便是二分查找在内存使用上更节省，但是毕竟内存如此紧缺的情况并不多。那二分查找真的没什么用处了吗？

实际上，求“值等于给定值”的二分查找**确实不怎么会被用到**，**二分查找**更适合用在“近似”查找问题，在这类问题上，二分查找的优势更加明显。比如今天讲的这几种变体问题，用其他数据结构，比如散列表、二叉树，就比较难实现了。

变体一：查找第一个值等于给定值的元素

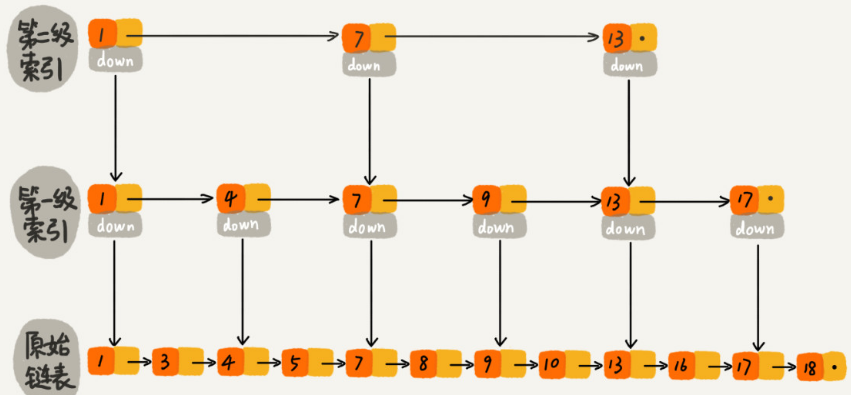
变体二：查找最后一个值等于给定值的元素

变体三：查找第一个大于等于给定值的元素

变体四：查找最后一个小于等于给定值的元素

## 17丨跳表：为什么Redis一定要用跳表来实现有序集合？

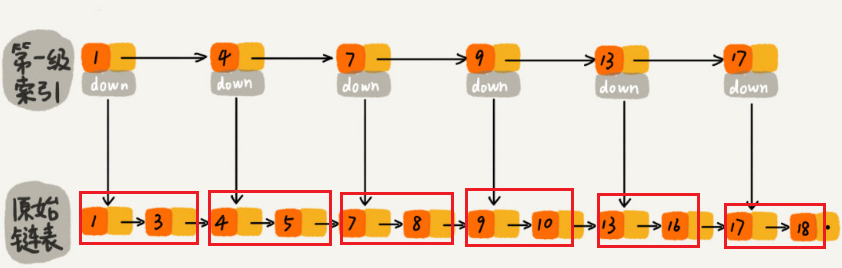
### **什么是跳表**

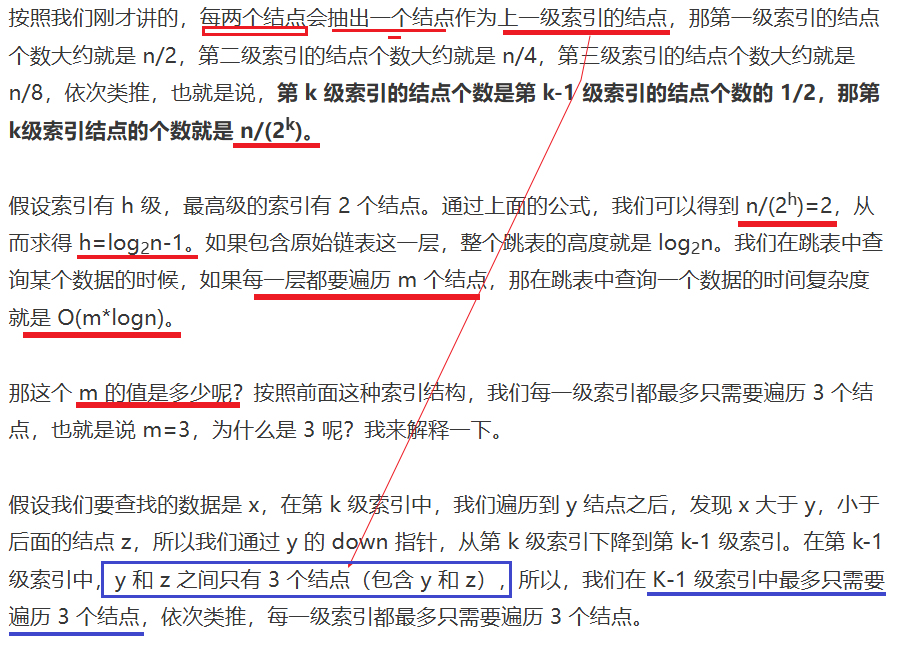


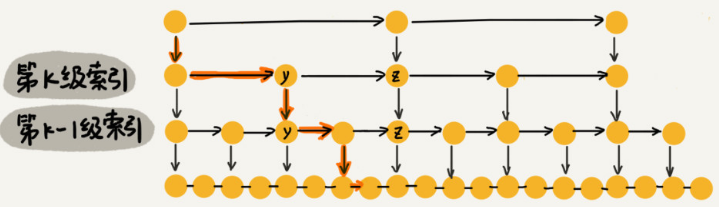
**这种链表加多级索引的结构，就是跳表.**

### 用跳表查询到底有多快？

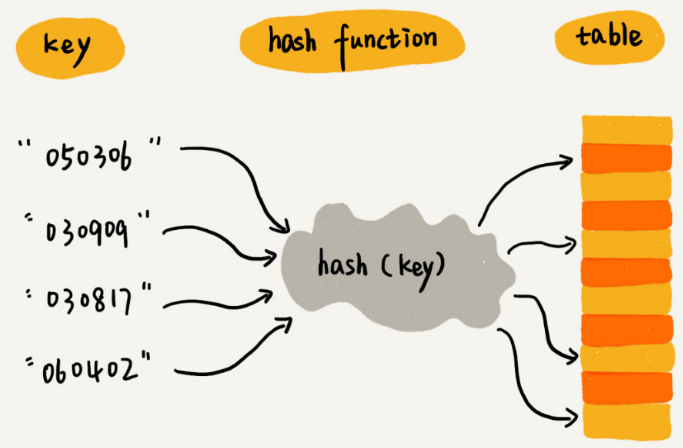
在跳表中查询任意数据的时间复杂度就是 **O(logn)**。这个查找的时间复杂度跟二分查找是一样的.







## 18丨散列表（上）：Word文档中的单词拼写检查功能是如何实现的？



**散列表，用的是[数组支持下标随机访问数据]的特性，所以散列表其实就是数组的一种扩展，由数组演化而来。可以说，如果没有数组，就没有散列表。**

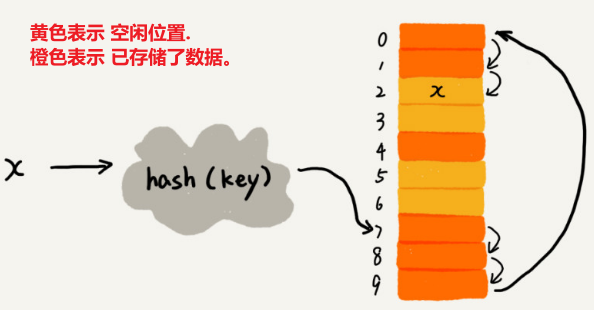
**散列函数，顾名思义，它是一个函数。我们可以把它定义成hash(key)，其中 key 表示元素的键值，hash(key) 的值表示经过散列函数计算得到的散列值。**

**散列冲突解决方法有两类，开放寻址法（open addressing）和链表法（chaining）。**

### 1. 开放寻址法

**开放寻址法的核心思想是**，如果出现了散列冲突，我们就重新探测一个空闲位置，将其插入。例如：**线性探测**（Linear Probing）。

**线性探测**（Linear Probing）。当往散列表中插入数据时，如果某个数据经过散列函数散列之后，存储位置已经被占用了，我们就从当前位置开始，依次往后查找，看是否有空闲位置，直到找到为止。



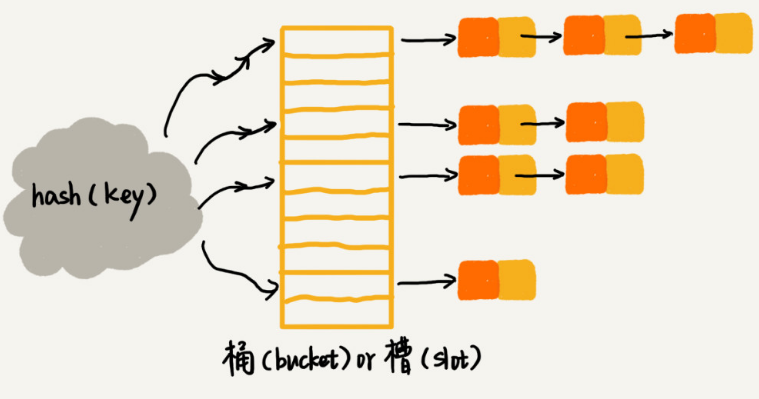
**装载因子**（load factor）来表示空位的多少。计算公式是：

散列表的装载因子 = **填入表中的元素个数 / 散列表的长度**

装载因子越大，说明空闲位置越少，冲突越多，散列表的性能会下降。

### 2. 链表法

链表法是一种更加常用的散列冲突解决办法，相比开放寻址法，它要简单很多。在散列表中，每个“桶（bucket）”或者“槽（slot）”会对应一条链表，所有散列值相同的元素我们都放到相同槽位**对应的链表中**。



### 总结

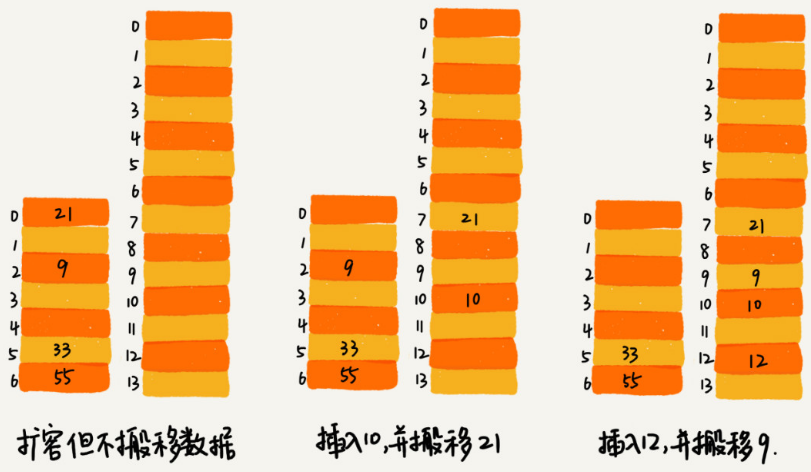
散列表来源于数组，它借助散列函数对数组这种数据结构**进行扩展**，利用的是数组支持按照下标随机访问元素的特性。散列表两个核心问题是**散列函数设计**和**散列冲突解决**。散列冲突有两种常用的解决方法，开放寻址法和链表法。散列函数设计的好坏决定了散列冲突的概率，也就决定散列表的性能。

|  |  |
| --- | --- |
|  |  |

## 19丨散列表（中）：如何打造一个工业级水平的散列表？

### 如何避免低效地扩容？

当有新数据要插入时，我们将新数据插入新散列表中，**并且**从老的散列表中拿出一个数据放入到**新散列表**。每次插入一个数据到散列表，我们都重复上面的过程。经过多次插入操作之后，老的散列表中的数据就一点一点全部搬移到新散列表中了。这样没有了集中的一次性数据搬移，插入操作就都变得很快了。



Java 中 LinkedHashMap 就采用了链表法解决冲突，ThreadLocalMap 是通过线性探测的开放寻址法来解决冲突。这两种冲突解决方法各有什么优势和劣势，又各自适用哪些场景吗？

#### 1. 开放寻址法

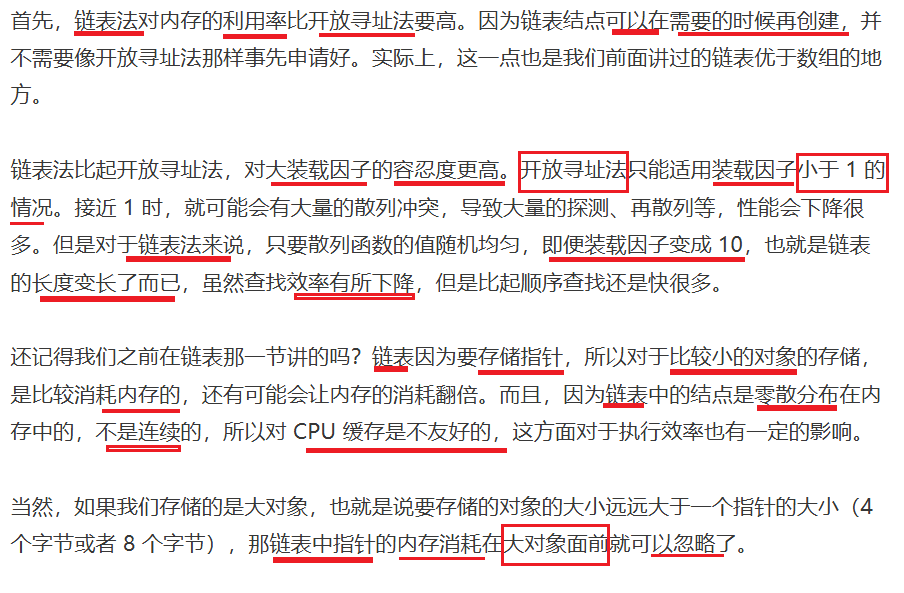
散列表中的数据都存储在数组中，可以有**效地利用 CPU 缓存**加快查询速度 .

开放寻址法解决冲突的散列表，删除数据的时候比较麻烦，需要特殊标记已经删除掉的数据。而且，在开放寻址法中，所有的数据都存储在一个数组中，比起链表法来说，冲突的代价更高。

所以，使用开放寻址法解决冲突的散列表，装载因子的上限不能太大。这也导致这种方法比链表法更浪费内存空间。

**总结一下，当数据量比较小、装载因子小的时候，适合采用开放寻址法。这也是 Java 中的ThreadLocalMap使用开放寻址法解决散列冲突的原因**。

#### 2. 链表法



**总结一下，基于链表的散列冲突处理方法比较适合存储大对象、大数据量的散列表，而且，比起开放寻址法，它更加灵活，支持更多的优化策略，比如用红黑树代替链表**

### **如何实现这样一个散列表呢？**

* 设计一个合适的散列函数；
* 定义装载因子阈值，并且设计动态扩容策略；
* 选择合适的散列冲突解决方法。

## 20丨散列表（下）：为什么散列表和链表经常会一起使用？

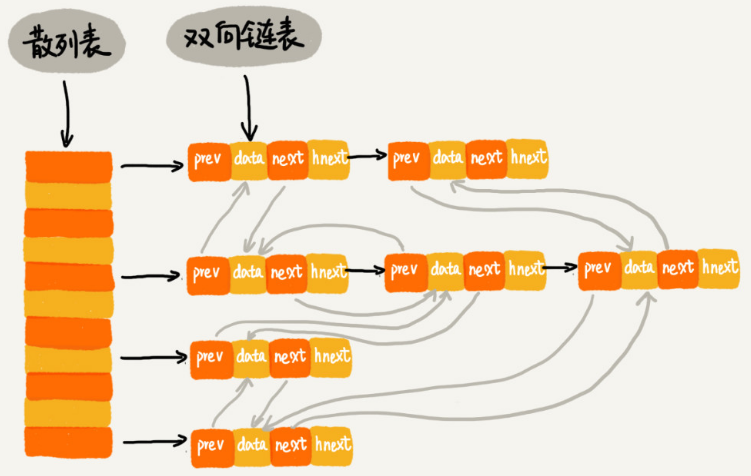
链表那一节，讲到如何用链表来实现 LRU 缓存淘汰算法，但是链表实现的 LRU 缓存淘汰算法的时间复杂度是 O(n)，通过散列表可以将这个时间复杂度降低到 O(1)。

跳表那一节， Redis 的有序集合是使用跳表来实现的，跳表可以看作一种改进版的链表。也提到过，Redis 有序集合不仅使用了跳表，还用到了散列表。

Java 编程语言中， LinkedHashMap 这样容器，**也用到了**散列表和链表两种数据结构。

### 散列表和链表结合(LRU缓存淘汰算法)

将散列表和链表两种数据结构组合使用，可以将这三个操作的**时间复杂度都降低到 O(1)**。具体的结构如下：



1. 因为**散列表**是通过**链表法**解决散列冲突的，所以每个结点会在**两条链中**。
2. 一个链是刚刚我们提到的**双向链表**，另一个链是散列表中的**拉链**。
3. **前驱(prev)和后继(next)指针是为了将结点串在双向链表中，**
4. **hnext 指针是为了将结点串在散列表的拉链中**。

#### 缓存的三个操作，是如何做到时间复杂度是 O(1) 的？

总结一下，一个缓存（cache）系统主要包含下面这几个操作：

1. 往缓存中**添加**一个数据；
2. 从缓存中**删除**一个数据；
3. 在缓存中**查找**一个数据。

首先，**如何查找一个数据**。在**散列表**中查找数据的**时间复杂度接近 O(1)**，所以通过散列表，可以很快地在缓存中找到一个数据。当找到数据之后，我们还需要将它**移动到**双向链表的**尾部**。

其次，**如何删除一个数据**。我们需要找到数据所在的结点，然后将结点删除。借助散列表，我们可以在 O(1) 时间复杂度里找到**要删除的结点**。因为我们的链表是双向链表，双向链表可以通过**前驱指针 O(1) 时间复杂度**获取**前驱结点**，所以在双向链表中，删除结点只需要 O(1) 的时间复杂度。

最后，**如何添加一个数据**。添加数据到缓存稍微有点麻烦，我们需要先看这个数据是否已经在缓存中。如果已经在其中，需要将其移动到双向链表的尾部；如果不在其中，还要看缓存有没有满。如果满了，则将**双向链表头部的结点**删除，然后再将数据放到链表的尾部；如果没有满，就直接将数据放到**链表的尾部**。

整个过程涉及的查找操作都可以通过散列表来完成。其他的操作，比如删除头结点、链表尾部插入数据等，都可以在 O(1) 的时间复杂度内完成。所以，这三个操作的时间复杂度都是 O(1)。至此，我们就通过散列表和双向链表的组合使用，实现了一个高效的、支持 LRU 缓存淘汰算法的缓存系统原型。

如何将数据移动倒链表的尾部：

在链表中，表头、尾指针已经在链表中被定义，且被记录。当**移动数据**到**尾部**的操作就是：**先删除**，**再添加**的过程。

### Redis 有序集合

跳表，在有序集合中，**每个成员对象**有两个重要的属性，**key（键值）和score（分值）**。我们不仅会通过 score 来查找数据，还会**通过 key 来查找数据**。

举个例子，比如**用户积分排行榜**有这样一个功能：我们可以通过**用户的ID** 来查找**积分信息**，也可以通过**积分区间**来查找**用户ID**或者姓名信息。

这里包含**ID、姓名和积分的用户信息，**就是**成员对象**，**用户ID**就是**key（键值）**，**积分** 就是**score（分值）**。

所以，如果我们细化一下 Redis 有序集合的操作，那就是下面这样：

1. 添加**一个成员对象**；
2. 按照**键值(key)**来删除一个成员对象；
3. 按照**键值(key)**来**查找**一个成员对象；
4. 按照**分值(score)区间**查找数据，比如查找积分在 [100, 356] 之间的成员对象；
5. 按照**分值(score)**从小到大排序成员变量；

如果**仅按照分值(score)**将成员对象组织成跳表的结构，那按照**键值(key)**来删除、查询成员对象**就会很慢**。

解决方法与 LRU 缓存淘汰算法的解决方法类似。再按照 **键值(key)** **构建一个散列表**，这样按照 key 来删除、查找一个成员对象的时间复杂度就变成了 O(1)。同时，借助跳表结构，其他操作也非常高效。

## Java LinkedHashMap

1. 照访问时间排序的 LinkedHashMap **本身就是**一个**支持LRU缓存淘汰策略的缓存系统。**
2. LinkedHashMap 是通过双向链表和散列表这两种数据结构组合实现的。LinkedHashMap 中的“Linked”实际上指的是**双向链表**，**并非**指用链表法解决**散列冲突**。



这段代码打印的结果是 1，2，3，5。

**在第 8 行代码中**，**再次将键值为 3 的数据放入到 LinkedHashMap 的时候**，会**先查找**这个键值是否已经有了，然后，**再将已经存在的 (3,11) 删除**，并且将**新的 (3,26)** **放到链表的尾部**。

**在第 9 行代码**访问到 key 为 5 的数据的时候，我们将被访问到的数据**移动到**链表的尾部。

## 21丨哈希算法（上）：如何防止数据库中的用户信息被脱库？

实际上，不管是“**散列”还是“哈希**”，这都是中文翻译的差别，英文**其实就是“Hash”**。

所以，我们常听到有人把“散列表”叫作“哈希表”“Hash 表”，把**“哈希算法”叫作“Hash 算法”或者“散列算法”。**

### 哈希算法的定义

将任意长度的二进制值串映射为固定长度的二进制值串，这个映射的规则就是哈希算法，而通过原始数据映射之后得到的二进制值串就是哈希值。

**设计一个优秀的哈希算法并不容易，需要满足的几点要求：**

1.从哈希值不能反向推导出原始数据（所以哈希算法也叫单向哈希算法）；

2.对输入数据非常敏感，哪怕原始数据只修改了一个 Bit，最后得到的哈希值也大不相同；

3.散列冲突的概率要很小，对于不同的原始数据，哈希值相同的概率非常小；

4.哈希算法的执行效率要尽量高效，针对较长的文本，也能快速地计算出哈希值。

哈希算法的应用非常多，我选了最常见的七个，分别是: 安全加密、唯一标识、数据校验、散列函数、负载均衡、数据分片、分布式存储。

**安全加密、**

常用于加密的哈希算法是

**MD5**（MD5 Message-Digest Algorithm，MD5 消息摘要算法）。

**SHA**（Secure Hash Algorithm，安全散列算法）。

**DES**（Data Encryption Standard，数据加密标准）。

**AES**（Advanced Encryption Standard，高级加密标准）。

**注意：哈希算法无法做到零冲突**

**唯一标识**、

要在海量的图库中，搜索一张图是否存在，我们**不能单纯地用图片的元信息**（比如图片

名称）来比对，因为有可能存在**名称相同但图片内容不同**，或者**名称不同图片内容相同**

**的情况**。那我们该如何搜索呢？

**可以给每一个图片取一个唯一标识**。【比如，我们可以从图片的二进制码串**开头取 100**

**个字节**，从**中间取 100 个字节**，从**最后再取 100 个字节**，然后将**这 300 个字节放到**

**一块**，】通过哈希算法（比如 MD5），**得到一个哈希字符串**，用它**作为图片的唯一标识**。

通过这个唯一标识来判定图片是否在图库中，这样就可以减少很多工作量。

**数据校验、**

当BT从多个机器上并行下载一个 2GB 的电影，这个电影文件可能会被分割成很多文件块（比如可以分成 100 块，每块大约 20MB）。等**所有的文件块**都下载完成之后，再组装成一个完整的电影文件就行了。

**由于网络传输是不安全的**，所以现在的问题是**，如何来校验文件块的安全、正确、完整呢？**

解决方法是：通过哈希算法，对 100 个文件块分别取哈希值，并且保存在种子文件中。

**散列函数、**

实际上，**散列函数也是哈希算法的一种应用。**

散列函数是设计一个散列表的关键。它直接决定了散列冲突的概率和散列表的性能。不过，相对哈希算法的其他应用**，散列函数对于散列算法冲突的要求要低很多。**即便出现个别散列冲突，**只要不是过于严重，我们都可以通过开放寻址法或者链表法解决。**

不仅如此，**散列函数**对于散列算法计算得到的值，是否能**反向解密也并不关心**。散列函数中用到的散列算法，更加关注**散列后的值是否能平均分布**，也就是，一组数据是否能均匀地散列在各个槽中。除此之外，散列函数执行的快慢，也会影响散列表的性能，所以，散列函数用的散列算法一般都比较简单，比较追求效率。

## 22丨哈希算法（下）：哈希算法在分布式系统中有哪些应用？

**负载均衡、**

会话保持 又被叫做 **粘滞会话（Sticky Sessions）。**

是负载均衡器上的一种机制，可以识别**客户端**与**服务器**之间交互过程的关联性，即便

做负载均衡的同时，**还保证**一系列相关联的访问请求**会保持分配到一台服务器上**。

**数据分片、**

通过哈希函数计算哈希值，然后再跟 n 取模，来决定将　数据　分配到那台机器。

**分布式存储**

**场景：**

如何决定将数据放到哪个机器上呢？我们可以借用前面数据分片的思想，即通过哈希算法对数据取哈希值，然后对机器个数取模，这个值就是应该存储的 缓存机器编号。

如果数据增多，原来的10台机器已经无法承受了，我们**就需要扩容**了，比如扩到 11 台机器，这时候麻烦就来了. **因为计算规则变了,所有的数据都要重新计算哈希值，然后重新搬移到正确的机器上。**

**一致性哈希算法：**

1.很好地解决了分布式系统**在扩容或者缩容时**，**发生**过多的数据迁移的问题。

2.原理：假设我们有 k 个机器，数据的哈希值的范围是 [0, MAX]。我们将整个范围划分成 m 个小区间（m 远大于 k），每个机器负责 m/k 个小区间。

　　　　当有新机器加入的时候，我们就将某几个小区间的数据，从原来的机器中**搬移到**新的机器中。这样，既不用**全部重新哈希**、搬移数据，也保持了各个机器上数据数量的均衡。

**参考文章：**

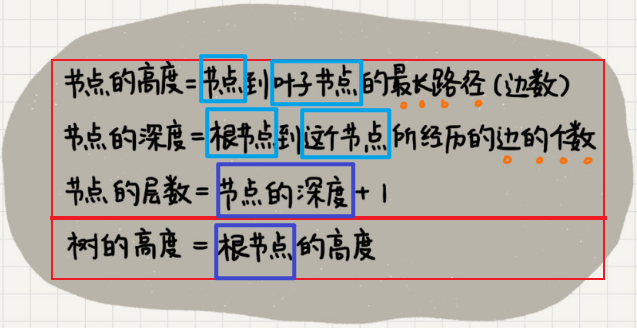
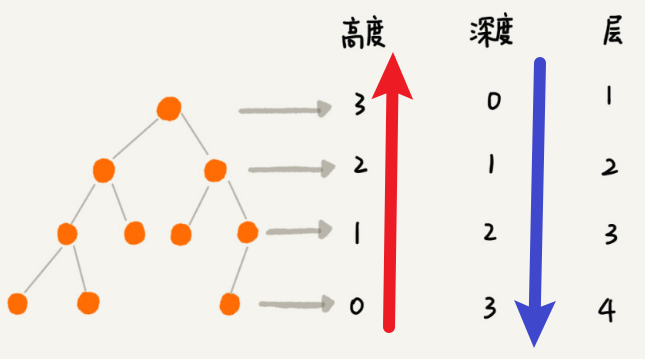
1. [一致性哈希是什么，使用场景，以及它解决了什么问题？](https://blog.csdn.net/ThinPikachu/article/details/123300298)
2. <http://www.zsythink.net/archives/1182>

## 23丨二叉树基础（上）：什么样的二叉树适合用数组来存储？

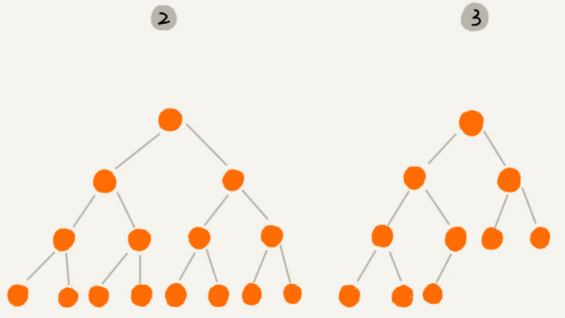
### 节点类型

根节点、父节点、子节点、兄弟节点、叶子节点。

**没有子节点的节点** 叫作 **叶子节点**或者**叶节点**

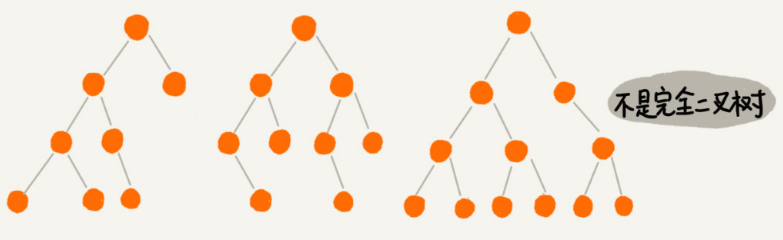
### 满二叉树与完全二叉树



编号2的二叉树中，叶子节点全都在最底层，除了叶子节点之外，每个节点都有左右两个子节点，这种二叉树就叫作**满二叉树**。

编号3的二叉树中，**叶子节点都在最底下两层**，**最后一层的叶子节点都靠左排列**，并且**除了最后一层**，**其他层的节点个数都要达到最大**，这种二叉树叫作**完全二叉树**。

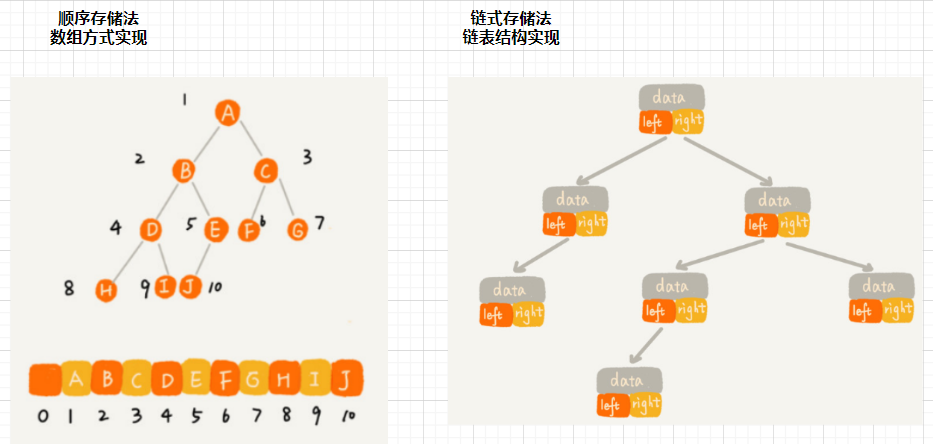
**不是完全二叉树案例**



### 完全二叉树的由来？

为什么偏偏把最后一层的叶子节点靠左排列的叫完全二叉树？

**主要为了节省空间。降低树的高度，**



基于数组的**顺序存储法**。

把根节点存储在下标 i = 1 的位置，那左子节点存储在下标 2 \* i = 2 的位置，右子节点存储在 2 \* i + 1 = 3 的位置。

以此类推，B 节点的左子节点存储在 2 \* i = 2 \* 2 = 4 的位置，右子节点存储在 2 \* i + 1 = 2 \* 2 + 1 = 5 的位置。

总结一下，如果节点 X 存储在数组中下标为 i 的位置，下标为 2 \* i 的位置存储的就是左子节点，下标为 2 \* i + 1 的位置存储的就是右子节点。反过来，下标为 i/2 的位置存储就是它的父节点。

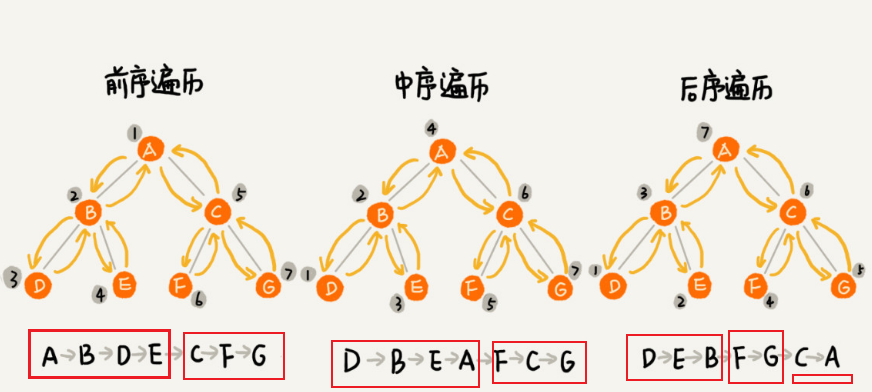
通过这种方式，我们只要知道根节点存储的位置（一般情况下，为了方便计算子节点，根节点会存储在下标为 1 的位置），这样就可以通过下标计算，把整棵树都串起来。

**堆其实就是一种完全二叉树，最常用的存储方式就是数组。**

### 二叉树的遍历

如何将所有节点都遍历打印出来呢？经典的方法有三种，**前序遍历、中序遍历和后序遍历**。其中，**前、中、后序，表示的是节点**与它的**左右子树节点**遍历打印的**先后顺序**。

1. 前序遍历是指，对于树中的任意节点来说，先打印这个节点，然后再打印它的左子树，最后打印它的右子树。
2. 中序遍历是指，对于树中的任意节点来说，先打印它的左子树，然后再打印它本身，最后打印它的右子树。
3. 后序遍历是指，对于树中的任意节点来说，先打印它的左子树，然后再打印它的右子树，最后打印这个节点本身。



前序遍历的递推公式：

preOrder(r) = print r->preOrder(r->left)->preOrder(r->right)

伪代码：

void preOrder(Node\* root) {

if (root == null) return;

print root // 此处为伪代码，表示打印 root 节点

preOrder(root->left);

preOrder(root->right);

}

中序遍历的递推公式：

inOrder(r) = inOrder(r->left)->print r->inOrder(r->right)

伪代码：

void inOrder(Node\* root) {

if (root == null) return;

inOrder(root->left);

print root // 此处为伪代码，表示打印 root 节点

inOrder(root->right);

}

后序遍历的递推公式：

postOrder(r) = postOrder(r->left)->postOrder(r->right)->print r

伪代码：

void postOrder(Node\* root) {

if (root == null) return;

postOrder(root->left);

postOrder(root->right);

print root // 此处为伪代码，表示打印 root 节点

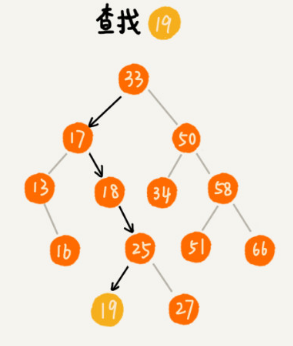
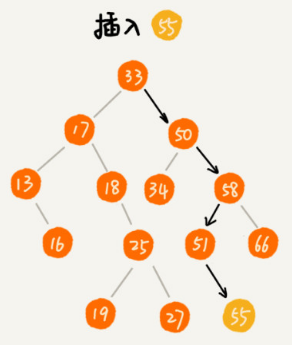
}

## 24丨二叉树基础（下）：有了如此高效的散列表，为什么还需要二叉树？

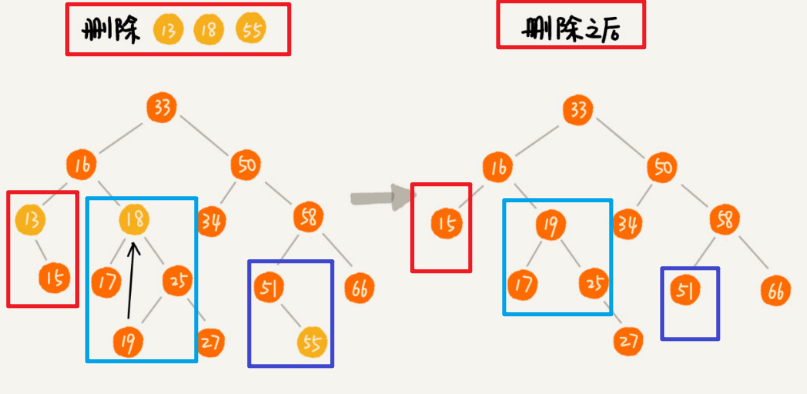
二叉查找树也叫二叉搜索树。顾名思义，二叉查找树是为了实现快速查找而生的。最大的特点就是，支持动态数据集合的快速插入、删除、查找操作。

### 定义

二叉查找树要求，在树中的任意一个节点，其左子树中的每个节点的值，都要小于这个节点的值，而右子树节点的值都大于这个节点的值。

### 删除



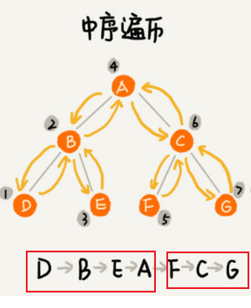
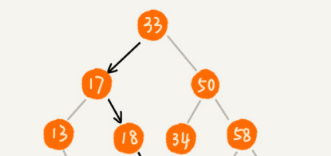
**第一种情况是**，如果要删除的节点没有子节点，我们只需要直接将父节点中，指向要删除节点的指针置为 null。比如图中的删除节点 55。

**第二种情况是**，如果要删除的节点只有一个子节点（只有左子节点或者右子节点），我们只需要更新父节点中，指向要删除节点的指针，让它指向要删除节点的子节点就可以了。比如图中的删除节点 13。

**第三种情况是**，如果要删除的节点有两个子节点，这就比较复杂了。我们需要找到这个节点的右子树中的最小节点，把它替换到要删除的节点上。然后再删除掉这个最小节点，因为最小节点肯定没有左子节点（如果有左子结点，那就不是最小节点了），所以，我们可以应用上面两条规则来删除这个最小节点。比如图中的删除节点 18。

### 有序输出数据

中序遍历**二叉查找树**，可以输出**有序的数据序列**，时间复杂度是 O(n)，非常高效。因此，二叉查找树**也叫作二叉排序树。**

**输出结果为： 13、17、18、33、34、50、58**

## 二叉查找树的时间复杂度分析



**最坏情况：O(n) 退化为链表形式**

**二叉查找树是一棵完全二叉树（或满二叉树）。这个时候，插入、删除、查找的时间复杂度是多少呢？**

**时间复杂度其实都跟树的高度成正比，也就是 O(height)**。既然这样，现在问题就转变成另外一个了，也就是，如何求一棵包含 n 个节点的完全二叉树的高度？

**树的高度**就等于最大层数**减一**，为了方便计算，我们转换成层来表示。从图中可以看出，包含 n 个节点的完全二叉树中，第一层包含 1 个节点，第二层包含 2 个节点，第三层包含 4 个节点，依次类推，下面一层节点个数是上一层的 2 倍，

**第 K 层包含的节点个数就是 2^(K-1)。**

平衡二叉查找树的高度接近 logn，所以插入、删除、查找操作的时间复杂度也比较稳定，是 O(logn)。

25丨红黑树（上）：为什么工程中都用红黑树这种二叉树？

26丨红黑树（下）：掌握这些技巧，你也可以实现一个红黑树

27丨递归树：如何借助树来求解递归算法的时间复杂度？

28丨堆和堆排序：为什么说堆排序没有快速排序快？

29丨堆的应用：如何快速获取到Top10最热门的搜索关键词？

30丨图的表示：如何存储微博、微信等社交网络中的好友关系？

31丨深度和广度优先搜索：如何找出社交网络中的三度好友关系？

32丨字符串匹配基础（上）：如何借助哈希算法实现高效字符串匹配？

33丨字符串匹配基础（中）：如何实现文本编辑器中的查找功能？

34丨字符串匹配基础（下）：如何借助BM算法轻松理解KMP算法？

35丨Trie树：如何实现搜索引擎的搜索关键词提示功能？

36丨AC自动机：如何用多模式串匹配实现敏感词过滤功能？

37丨贪心算法：如何用贪心算法实现Huffman压缩编码？

38丨分治算法：谈一谈大规模计算框架MapReduce中的分治思想

39丨回溯算法：从电影《蝴蝶效应》中学习回溯算法的核心思想

40丨初识动态规划：如何巧妙解决“双十一”购物时的凑单问题？

41丨动态规划理论：一篇文章带你彻底搞懂最优子结构、无后效性和重复子问题

42丨动态规划实战：如何实现搜索引擎中的拼写纠错功能？

# 总结

1. 操作受限的线性表数据结构：队列、栈。
2. 跳表：时间复杂度是 O(logn)、空间复杂度是 O(n)。
3. 二分查找 时间复杂度是 O(logn)。

## 数学公式

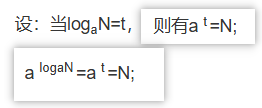
### 公式一



### 公式二



证明：



# 案例