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。

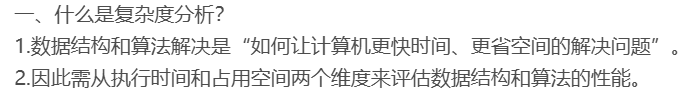
## 3方法

“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”**

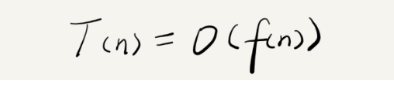
在学习的过程中，一定会碰到“拦路虎”。如果哪个知识点没有怎么学懂，不要着急，这是正常的。因为，想听一遍、看一遍就把所有知识掌握，这肯定是不可能的。**学习知识的过程是反复迭代、不断沉淀的过程。**

# 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？



## 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

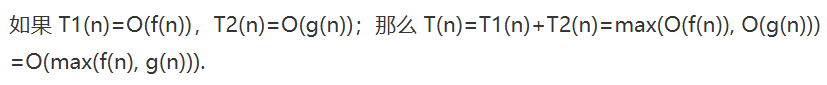
第一个例子中的 T(n) = O(2n+2)，第二个例子中的 T(n) = O(2n2+2n+3)。这就是**大O时间复杂度表示法**。

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：

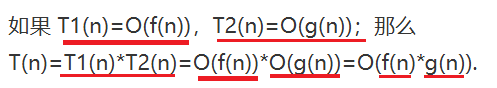
第一个例子：T(n) = O(n)； 第二个例子：T(n) = O(n2)。

## 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:**

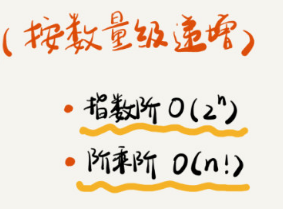
假设T1(n) = O(n)，T2(n) = O(n2)，则 T1(n) \* T2(n) = O(n3)。

## 几种常见时间复杂度

分为两类，**多项式量级**和**非多项式量级**。

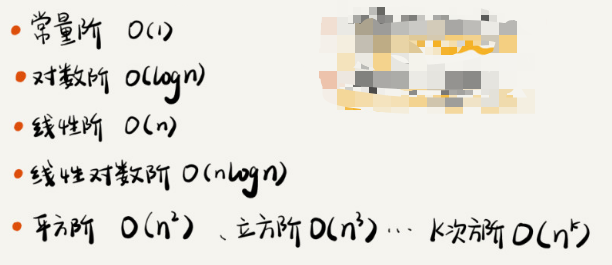
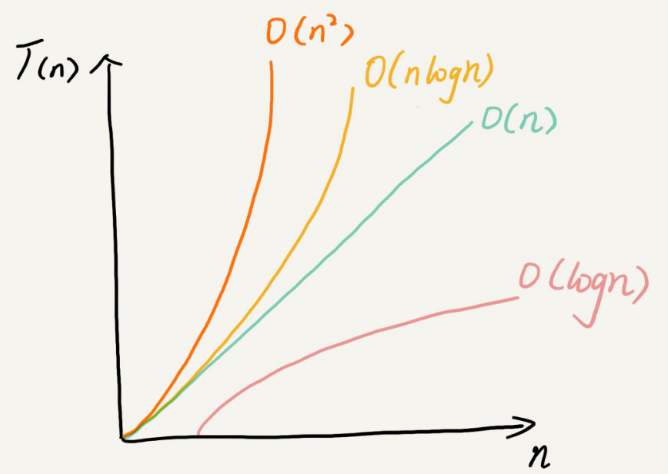
### **非多项式量级**

只有两个：O(2n) 和 O(n!)。

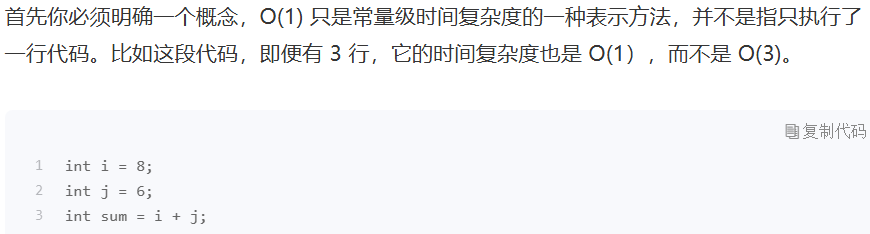


当数据规模 n 越来越大时，非多项式量级算法的执行时间会急剧增加，求解问题的执行时间会无限增长。所以，非多项式时间复杂度的算法其实是***非常低效的算法。***

### **多项式量级**

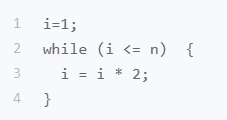
 

#### **1.O(1)**



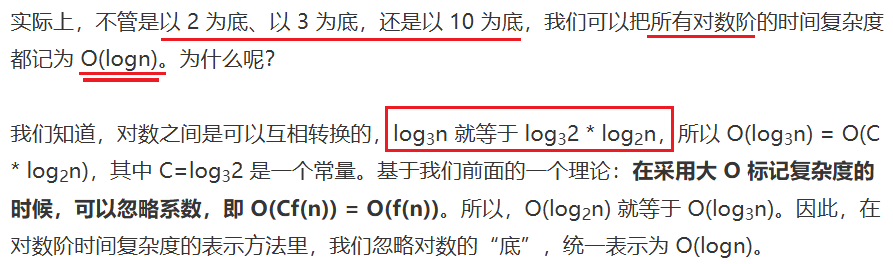
**一般情况下，只要算法中不存在循环语句、递归语句，即使有成千上万行的代码，其时间复杂度也是Ο(1)**。

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

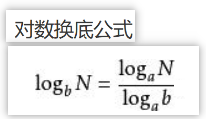
 

只要知道 x 值是多少，就知道这行代码执行的次数了。通过 2x=n 求解得出，x=log2n。所以，这段代码的时间复杂度就是 O(log2n)。

##### 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（amortized time complexity）

**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**

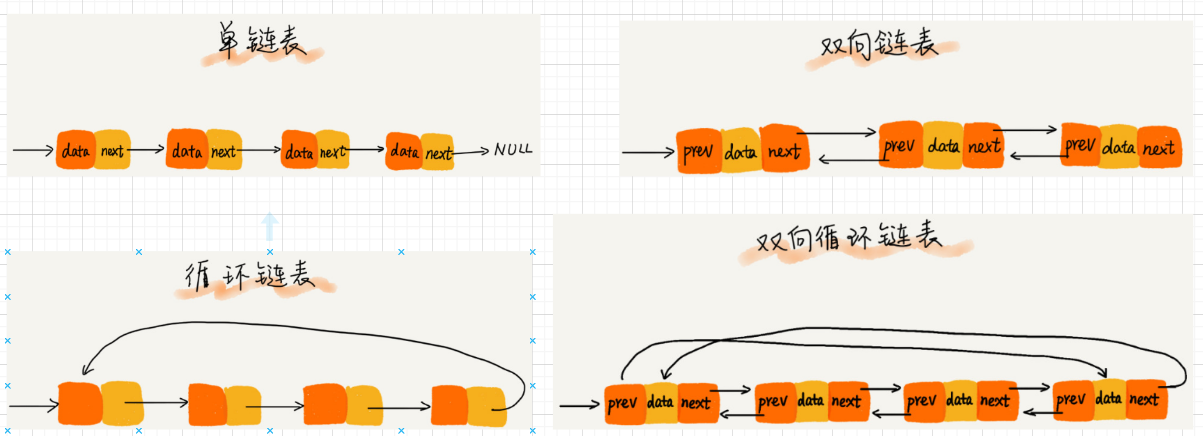
比如二叉树、堆、图等。之所以叫非线性，是因为，在非线性表中，数据之间并不是简单的前后关系。

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。

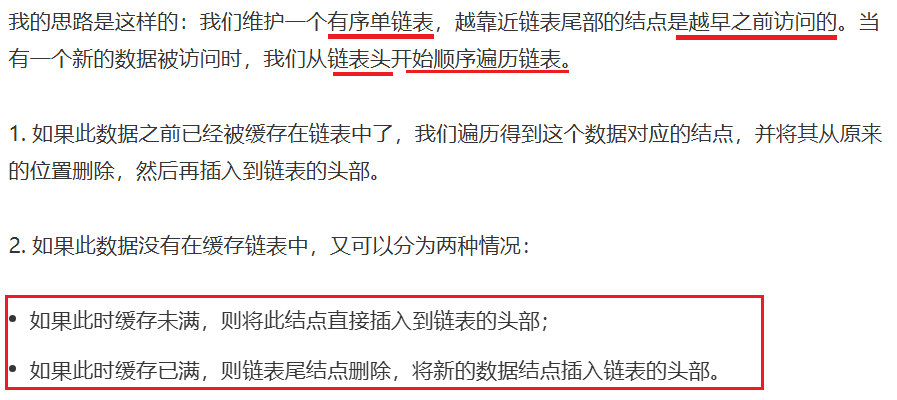
# 06丨链表（上）：如何实现LRU缓存淘汰算法？

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？**



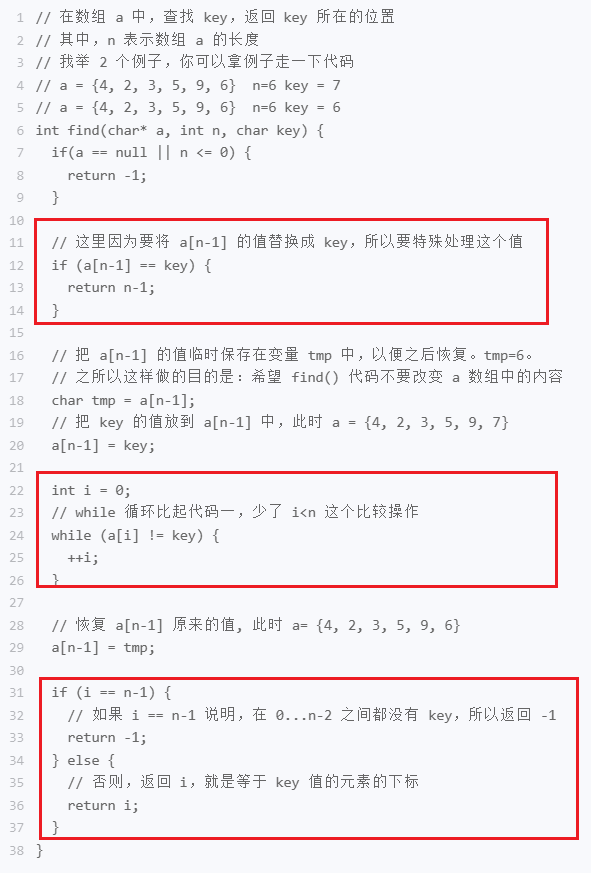
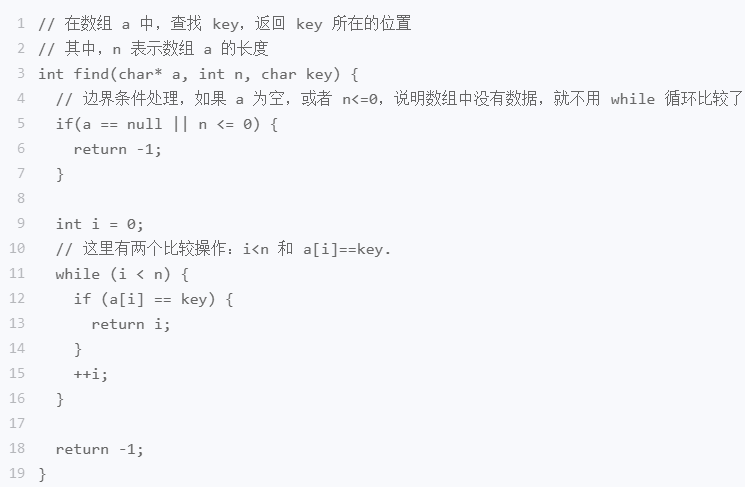
# 07丨链表（下）：如何轻松写出正确的链表代码？

**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?**

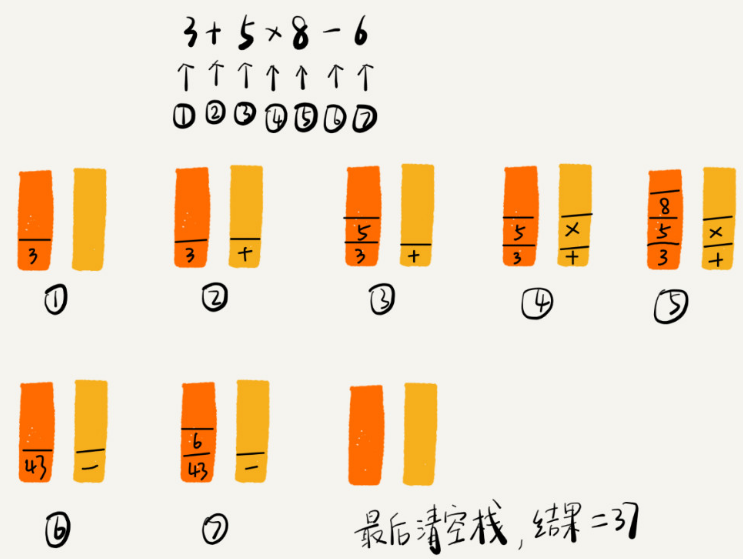
## 方案

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

10丨递归：如何用三行代码找到“最终推荐人”？