版本号：v1.6 阶段：初稿

# LeetCode刷题宝典

MRL Liu

（未经许可，不得传播）

2022年02月17日

# 第1章 数组

数组很可能是最简单也最常见的数据结构之一，数组是**顺序存储结构**的代表，其是静态分配一段规定内存，数组初始化时必须指定其长度，在**查找**和**修改**时效率高。C++中存在字符和字符串的区别，字符串被看做是字符数组。

本节介绍数组中常见的LeetCode类型，部分题型可能会涉及到字符串的处理。

## 一、STL中数组的用法

本小节主要介绍C++中动态数组vector的使用，其在LeetCode题目中的使用也非常高，请务必掌握。其中包含了vector对象的5种初始化方式、三种遍历写法、插入、删除、获取、排序、反转和分割的操作。

|  |  |
| --- | --- |
| 初始化 |  |
| 遍历方式 |  |
| 插入方式 |  |
| 删除方式 |  |
| 获取方式 |  |
| 排序方式 |  |
| 反转方式 |  |
| 分割数组 |  |

## 双指针

双指针是LeetCode中经常使用的一种解题技巧，其经常不需要借助过多的空间就可以使得算法效率大大提升。

### 删除数组中的元素

|  |  |  |
| --- | --- | --- |
| **二分查找无重复元素的有序数组** | **情景一：删除数组中的某个（所有的）目标值。**  **方法：快慢指针法**  快指针负责遍历数组，判断当前元素是否和目标值相等；  慢指针负责划分[0,low),指向删除后的区间的最后一个元素的后一个索引  如果快指针等于目标值，就跳过；  如果快指针不等于目标值，就覆盖给慢指针 | **704 二分查找（简单难度）** |
| **情景二：删除有序数组中的重复项**  **方法：快慢指针法** | **35 搜索插入位置（简单难度）** |
|  | **情景二：删除有序数组中的重复项** |  |

### 移动数组中的元素

### 3、查找排序数组中的2个数

## 三、二分查找

当LeetCode题目要求**在一个有序数组中查找某些数**或者**设计一个时间复杂度为O(log n)的解决方案**时，那么题目大概率是希望你使用二分查找法来解决问题。

**二分查找法**也叫做**折半查找法**，其本质也是一种双指针法，由于其非常经典，我们将其单独列出来进行总结。可以快速使用二分查找法的条件如下：

|  |
| --- |
| * 数组为**有序数组**（**必须**具备，否则换思路） * 数组**无重复元素**（**最好**具备，否则查找的index无法确定是哪一个target） |

### 1、二分查找法的模板

二分查找法使用了left和right两个指针，left指针从左到右扫描数组[0,len)，right指针从右到左扫描数组[len-1,0)，mid是left和right索引的中间位置，每次通过比较nums[mid]和target来确定left和right的移动情况。这里我们介绍2种写法：

|  |  |
| --- | --- |
| **左闭右闭区间**  **[left, right]** |  |
| **左闭右开区间**  **[left, right)** |  |

观察以上模板，可以知道2个模板的逻辑思路是一样的，不同的在于区间的定义，如果区间定义不同，相应的细节需要修改。

### 二分查找法的简单变体

现在我们来列举一些二分查找法的简单变体。先介绍几道二分查找无重复元素的有序数组的题目，这是二分查找法的专长。

|  |  |  |
| --- | --- | --- |
| **二分查找无重复元素的有序数组** | **情景一：二分查找一个数，如果没找到，返回-1。**  **方法：套用一个二分查找模板即可** | **704 二分查找（简单难度）** |
| **情景二：二分查找一个数，如果没找到，返回该数可以顺序插入的索引**  **方法：先套用一个二分查找模板，如果没找到，返回right+1** | **35 搜索插入位置（简单难度）** |

35题和一般的二分查找法不一样的是，如果在数组中没找到，不能直接返回-1，返回其顺序插入的位置。这就需要笔者对二分查找的不同情况都烂熟于心，我们分析如下。

|  |
| --- |
| 1、如果找到target，自然返回其索引  2、如果没有找到target，则存在三种情况：  （1）target比数组元素最小元素还小，说明二分查找中一直移动right，直至right==0;  （2）target比数组元素最大元素还大，说明二分查找中一直移动left，直至left==len-1;  （3）target在数组元素范围内，但是就是不等于数组元素，说明left和right都在移动，最终right<left。  无论哪种情况，实际上，target应该顺序插入的元素位置都是right+1. |

### 二分查找算数平方根

有些题目暗示的非常不明显，但是却用二分查找法做。例如算数平方根和完全平方数，这种问题实际上就是要在[1,num]中查找一个数使其满足num的算数平方根或者判断num是否是完全平方数。

|  |  |  |
| --- | --- | --- |
| **二分查找算数平方根** | **情景一：不使用内置函数，求x的算数平方根。**  **方法：二分查找+除法比较**  x的算数平方根的范围只能是[1,x],只能遍历查找哪一个是平方根。[1,x]是一个有序数组，显然二分查找更加快捷。  这道题注意，不能使用mid\*mid和x比较，因为x的数值比较大，容易溢出，可以**x/mid和mid**比较。 | **69 x的平方根（简单难度）**  **剑指 Offer II 072. 求平方根（简单难度）** |
| **情景二：判断一个数是否是完全平方数**  **方法：先套用一个二分查找模板，如果没找到，返回right+1**  显然一个完全平方数必须可以被某个数整除，即在[1,num]间是否存在一个数可以被num整除，由于num非常大，所以查找必须更加快捷，只能用二分查找法。  **当一个数满足num/i==i&&num%i==0时才是完全平方数**，注意num/i==i不能保证num就是完全平方数，因为num/i可能会舍弃小数 | **367 有效的完全平方数（简单难度）** |

### 二分查找有重复元素的有序数组

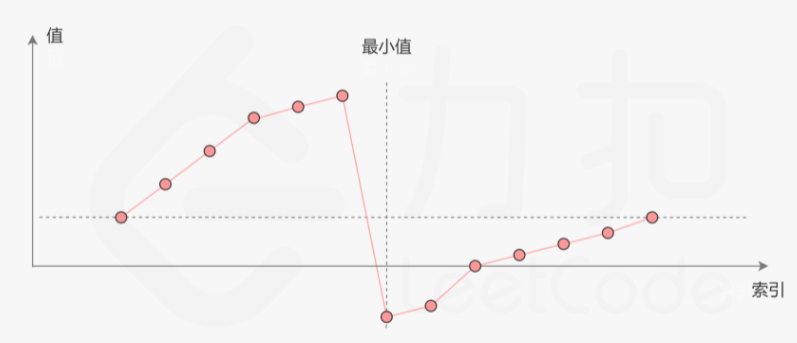
前面我们介绍二分查找法的条件时必须是有序数组，最好无重复元素，但是如果有重复元素还必须用二分查找法呢？接下来介绍二分查找有重复元素的有序数组。

|  |  |  |
| --- | --- | --- |
| **二分查找有重复元素的有序数组** | **情景一：在一个有序数组中查找某重复元素出现的第一个索引和最后一个索引。**  **方法：二分查找，然后根据找到的index尝试左移或者右移**  **二分查找一个有序数组的某重复元素，最后返回的index无法确定是重复元素中的第几个索引**。  这是可以从返回的index出发，尝试左移或者右移，来确定左右边界，只要和重复元素不一样就是边界。 | **34 在排序数组中查找元素的第一个和最后一个位置（中等难度）**  **剑指 Offer 53 - I. 在排序数组中查找数字 I（简单难度）** |

### 二分查找旋转排序数组

现在我们继续逐渐打破二分查找法的常规，二分查找法要求必须是有序数组，之前的题目这个有序数组一致是全局有序，即从小到大排列，现在如果有序数组是局部有序呢？

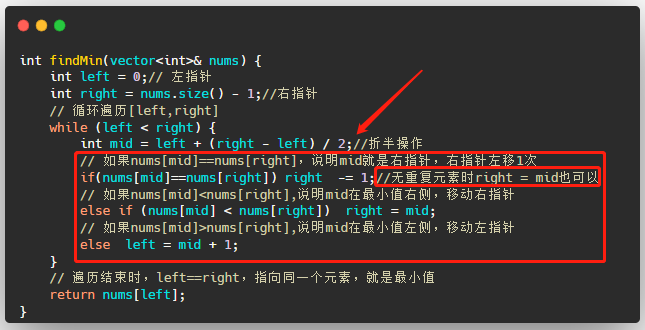
旋转数组的一部分，即将一个有序数组的[0,n)剪切到有序数组的末尾，此时继续查找有序数组中的最小值或者某个值。



#### （1）二分查找旋转排序数组的最小值

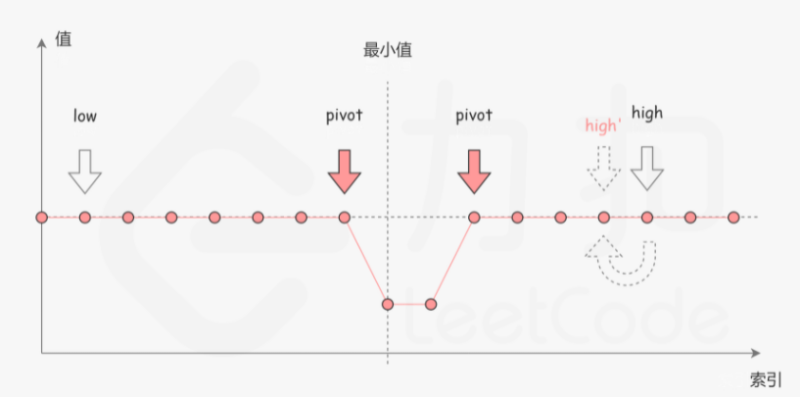
|  |  |  |
| --- | --- | --- |
| **二分查找旋转排序数组** | **情景一：查找有/无重复元素的旋转排序数组中的最小值**  **方法：nums[mid]可以和nums[right]比较进行二分查找**  假设数组最后1个元素为nums[right]，则最小值的右侧元素都<=nums[right];最小值的左侧元素都>=nums[right];我们的nums[mid]可以和nums[right]比较来进行二分查找 | **[153. 寻找旋转排序数组中的最小值（中等难度）](https://leetcode-cn.com/problems/find-minimum-in-rotated-sorted-array/)****[154. 寻找旋转排序数组中的最小值 II（困难难度）](https://leetcode-cn.com/problems/find-minimum-in-rotated-sorted-array-ii/)****[剑指 Offer 11. 旋转数组的最小数字（简单难度）](https://leetcode-cn.com/problems/xuan-zhuan-shu-zu-de-zui-xiao-shu-zi-lcof/)** |

我们分析下这类题的代码模板：二分查找模版是nums[mid]和target相比，存在三种情况，而且两者相等时就找到了target；该类题的模板是nums[right]和nums[mid]相比，两者相等时，不一定就找到了数组的最小值，只是让right指针左移一点，最后靠二分查找完left==right找到最小值。



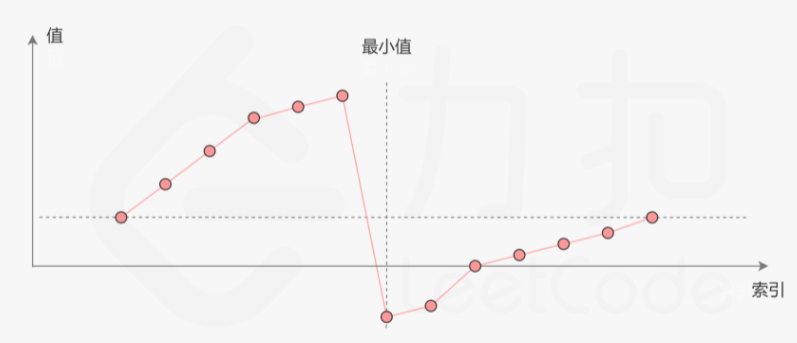
旋转排序数组有没有重复元素会影响nums[mid]==nums[right]的情况：

|  |
| --- |
| 没有重复元素时，mid一定就是right，right==mid即可，当然right-=1也对。  有重复元素时，mid可能在最小值右侧或者左侧。  索性采取保守策略，right统一左移1个元素。 |



#### 二分查找旋转排序数组的某个值

如果现在不找排序数组的最小值，就打算找其中某个值是否存在，如何设计算法呢？显然我们不能再比较nums[mid]和nums[right]，因为这是最小值才具有的性质，数组的随便一个数不具备这种性质。



|  |  |  |
| --- | --- | --- |
| **二分查找旋转排序数组** | **情景一：查找无重复元素的旋转排序数组中的某个值**  **方法：nums[mid]和target，nums[mid]和nums[0]比较**  先需要比较nums[mid]和target，确定nums[mid]是不是target，如果是直接返回mid；否则判断下nums[mid]和nums[0]，这是为了判断mid目前在第一段升序区间还是第二段升序区间；  如果在第一段升序区间，则判断target是否在区间[nums[0],nums[mid]]，如果在移动right指针即可，否则移动left指针；如果在第二段升序区间，则判断target是否在区间[nums[mid],nums[len-1]]，如果在移动left指针即可，否则移动right指针。 | **33. 搜索旋转排序数组（中等难度）** |
|  | **情景一：查找有重复元素的旋转排序数组中的某个值**  **方法：nums[mid]和target，nums[left]、nums[right]和nums[mid]比较** | **81. 搜索旋转排序数组 II（中等难度）** |

### 6、按值二分

|  |  |  |
| --- | --- | --- |
| **二分查找旋转排序数组** | **情景一：查找**  **方法：nums[mid]和target，nums[mid]和nums[0]比较**  先需要比较nums[mid]和target，确定nums[mid]是不是target，如果是直接返回mid；否则判断下nums[mid]和nums[0]，这是为了判断mid目前在第一段升序区间还是第二段升序区间； | **287. 寻找重复数（中等难度）** |

## 四、滑动窗口

### 1、什么是滑动窗口

滑动窗口是LeetCode中的一种解题技巧，它是在一个序列中使用一个长度可控的滑动窗口在序列的**合法区间**进行滑动，同时**动态地**记录一些有用数据，在很多情况下可以极大地提升算法效率。滑动窗口本质上也是一种双指针技巧。

对于滑动窗口区间的不同定义，也会决定不同的初始化逻辑和遍历逻辑，我们一般定义一个左闭右开的区间[left,right)来表示滑动窗口。

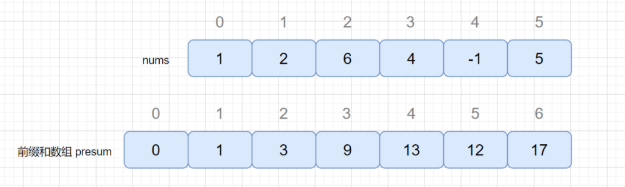
子数组是原数组中连续的元素，中间不可以删除或添加其他元素，每个元素的相对顺序和原数组相同

|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **LeetCode题目** |
| **寻找数组的中心索引** | **情景一：寻找数组中的中心索引，使中心索引左侧所有元素之和=右侧所有元素之和**  思路：  计算前缀和数组presum。  遍历数组，对每个索引进行如下判断：  第i个索引的左侧和=presum[i]  第i个索引的右侧和=presum[n]-num[i]-presum[i]  两者相等时i即为结果 | **724.寻找数组的中心下标（简单难度）**  **剑指 Offer II 012. 左右两边子数组的和相等（简单难度）**  **1991. 找到数组的中间位置（简单难度）** |

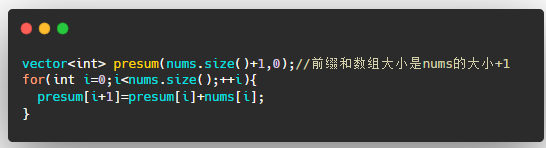
## 五、前缀和

#### 什么是前缀和

前缀和是LeetCode中的一种解题技巧，其本质就是数组里的前n项和。我们一般通过一个前缀和数组来保存数组中前n位的和，如下：



前缀和的作用就是可以帮助我们快速求解数组中每个小区间的和，例如我们要求nums[2]到nums[4]的和，只需要求presum[5]-presum[3]即可。前缀和数组的获取也非常简单，如下：



#### （2）寻找中心索引

|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **LeetCode题目** |
| **寻找数组的中心索引** | **情景一：寻找数组中的中心索引，使中心索引左侧所有元素之和=右侧所有元素之和**  思路：  计算前缀和数组presum。  遍历数组，对每个索引进行如下判断：  第i个索引的左侧和=presum[i]  第i个索引的右侧和=presum[n]-num[i]-presum[i]  两者相等时i即为结果 | **724.寻找数组的中心下标（简单难度）**  **剑指 Offer II 012. 左右两边子数组的和相等（简单难度）**  **1991. 找到数组的中间位置（简单难度）** |

#### （3）寻找和为K的连续子数组的个数

给定一个整数数组nums，给出数组nums的和为k的连续子数组的个数。

这里注意连续子数组的概念，例如nums=[1,2,3]，其长度为2的连续子数组就是[1,2]或[2,3]（注意[1,3]不是连续子数组）

我们可以通过双重遍历来搜索一个数组的连续子数组，例如：

|  |  |
| --- | --- |
| **双重遍历（会超时）** | **前缀和+双重遍历（会超时）** |
|  |  |

这里使用前缀和+哈希表的方案来提升算法的效率，其原理类似于两数之和：

|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **LeetCode题目** |
| **寻找数组的中心索引** | **情景一：给定一个整数数组和一个整数 k，你需要找到该数组中和为k的连续的子数组的个数。** | **560 和为 K 的子数组（中等难度）**  **剑指 Offer II 010. 和为 k 的子数组（中等难度）**  **930. 和相同的二元子数组（中等难度）** |
|  | **情景二：给定一个整数数组和一个整数k，你需要找到该数组中奇数个数为k的连续的子数组的个数。** | **1248. 统计「优美子数组」（中等难度）** |
|  | **情景三：给定一个整数数组和一个整数k，你需要找到该数组中和可以被k整除的连续的子数组的个数。**    （**该题的测试数据中存在元素为负数情况**） | **974. 和可被 K 整除的子数组（中等难度）** |
|  | **情景四：给定一个整数数组和一个整数k，你需要找到该数组中和可以被k整除的连续的子数组的个数。** | **523. 连续的子数组和（中等难度）** |

## 六、进制转换

本节介绍LeetCode数组相关题目中的进制转换类型的题目，其基础知识是**10进制转换为k进制**和**k进制转换为10进制**2种类型。在介绍进制转换前，先简单介绍下C++提供的字符串类string，可以将其看做是一个存储成员类型为char的vector，除了string.length()和vector.size()不同外，其他API的使用基本一致。

由于C++中默认的整数都是10进制，所以LeetCode题目中一般用vector或string存储转换后的k进制的数。所以其中可能会用到整数、字符、字符串的转换知识，如下：

|  |  |
| --- | --- |
| **整数和字符的相互转换** | **整数和字符串的相互转换** |
|  |  |

整数和字符可以通过加减进行相互转换，因为字符也属于整型，其参与四则运算时使用ASCII值，a’的ASCII值是97，‘A’的ASCII值是65，‘0’的ASCII值是48。

#### （1）进制转换的模板

|  |  |
| --- | --- |
| **理论依据** | **进制转换模板** |
|  |  |
|  |  |

#### （2）二进制转换的题目

|  |  |  |
| --- | --- | --- |
| **类型** | **解题思路** | **LeetCode题目** |
| **2进制转10进制** | 将链表添加到数组，遍历2进制数组转换为10进制 | **1290. 二进制链表转整数（简单难度）** |
| **10进制转2进制** | 10进制转换为2进制，然后遍历字符串统计字符1的个数 | 1. **位1的个数（简单难度）**   **剑指 Offer 15. 二进制中1的个数（简单难度）** |
| **10进制转2进制** | 遍历将n个数依次从10进制转换为2进制，然后遍历字符串统计字符1的个数 | 1. **比特位计数（简单难度）**   **剑指 Offer II 003. 前 n 个数字二进制中 1 的个数（简单难度）** |
| **10进制转2进制** | 10进制转换为2进制，然后遍历字符串查看相邻元素是否相同 | **693. 交替位二进制数（简单难度）** |
| **10进制转2进制2进制转10进制** | 10进制转换为2进制，然后遍历字符串将0和1字符替换，再从2进制转换为10进制。 | 1. **数字的补数（简单难度）**   **1009. 十进制整数的反码（简单难度）** |

#### （3）非二进制转换的题目

|  |  |  |
| --- | --- | --- |
| **类型** | **解题思路** | **LeetCode题目** |
| **10进制转7进制** | 套用模板，注意负数要先转换为正数，转换后加负号 | **504. 七进制数（简单难度）** |
| **10进制转16进制** | 套用模板，注意16进制中大于9的数要使用小写字母 | **405. 数字转换为十六进制数（简单难度）** |
| **26进制转10进制** | 套用模板，注意大写字母和数字转换 | **171. Excel 表列序号（简单难度）** |
| **10进制转26进制** | 套用模板，注意该题整数是1-26，每一位处理前需要-1 | **168. Excel表列名称（简单难度）** |

### 3、数位分离

数位分离就是将一个整数的各个位分离出来进行相关操作，常见的方法有两种，一种是直接将整数转换为string，然后对string进行操作；一种是通过10进制转（k=10）进制的模板拿到各个位。**判断一个数有多少位，只需要和10、100等比较即可（编程技巧）**。

|  |  |  |
| --- | --- | --- |
| **类型** | **解题思路** | **LeetCode题目** |
| **数位分离后累加** | 不断数位分离后累加直至一位数 | **1281. 整数的各位积和之差（简单难度）** |
| **数位分离后累加至1位数** | 不断数位分离后累加直至一位数（判断1位数可以通过和10比较） | **258. 各位相加（简单难度）** |
|  |  | **1945. 字符串转化后的各位数字之和（简单难度）** |
|  |  | **[1837. K 进制表示下的各位数字总和（简单难度）](https://leetcode-cn.com/problems/sum-of-digits-in-base-k/)** |
|  |  | **[9. 回文数（简单难度）](https://leetcode-cn.com/problems/palindrome-number/)** |

### 6、位运算和数学

位运算是算法中常见的对二进制的操作，数学更是与算法关系密切，可以用于求解很多算法题。

#### 整数在计算机中的表示方式

计算机中的整数分为有符号整数和无符号整数：

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 取值类型 | 数值范围（k字节,一般为4） | 本质区别 |
| 有符号整数 | 负整数、零与正整数 | -2k-1~2k-1-1 | 最高位为符号位，0表示零与正整数，1表示负整数 |
| 无符号整数 | 零与正整数 | 0~2k-1 | 无符号位 |

#### 原码、补码和反码

计算机中的数据都是以二进制形式表示的机器数，机器数是有符号数，即最高位是符号位。由于机器数存在符号位，所以机器数的形式值不一定等于其真值，例如10001010的形式值是138，真值是-10。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 表示 | 组成部分 | 优缺点 | 计算举例 | 正数例子（+10） | 负数例子（-10） |
| 原码 | 最高位为符号位，其余位表示数值 | 减法计算会出错；  同时存在+0和-0的错误表示 |  | 00001010 | 10001010 |
| 反码 | 0和正数与原码相同;负数的反码是将原码的除符号位外的每一位取反 | 解决了原码的减法计算问题，  但是存在+0和-0的错误表示 |  | 00001010 | 11110101 |
| 补码 | 0和正数与原码相同;负数的补码是在反码的基础上+1 | 解决了原码的减法计算问题;  解决了+0和-0的错误表示 |  | 00001010 | 11110110 |

由于补码解决了原码存在的2个问题，所以计算机最终内存中存储和参与运算的都是补码。



#### 位运算

计算机的一切数学运算本质都是位运算，位运算共有6种：与、或、异或、取反、左移、右移。

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 符号 | 记忆口诀 | 备注 |
| 与 |  | 全0为0，全1为1 | a&1==0/1经常用来取位操作，可以取数的二进制的最后一位，也可以用来判断奇偶性，等于0表示偶数，等于1表示奇数 |
| 或 |  | 有1则1，无1则0 |  |
| 取反 |  | 遇0变1，遇1变0 |  |
| 异或 |  | 相同为0，不同为1 | a^b可以得到判断两个数的二进制位数不同的个数；  a^0==a;  a^a==0; |
| 左移 | << | 高位丢弃，低位补0 | a<<k等价于a\*2k  （可能存在溢出的情况） |
| 右移 | >> | 低伟丢弃，  算数右移时，高位补最高位  逻辑右移时，高位补0. | a>>k等价于a/2k  （a不能为负数） |

1. 并查集
2. 智能指针

# 第2章 字符串

## 一、STL中字符串的用法

首先介绍下C++中STL内置的string使用方法，C++通过<string>文件来提供链表的数据结构操作，<string>本质就是一个字符数组，其大部分API和vector一样，常用的不一样的API是string.length()来获取长度，vector是通过vector.size()来获取长度。

数字转字符串：to\_string()

字符串的子串：string.substr(3,5);//表示索引3开始的5个字符，即[3,3+5)。

## 分割字符串

给定一个字符串句子"Hello how are you"，请获取其中的每个单词。

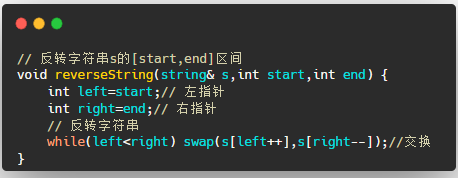
字符串中的单词用若干个空格间隔，字符串前端或结尾都可能存在若干个空格。

|  |  |  |  |
| --- | --- | --- | --- |
| **使用额外空间分割字符串** | 思路：  定义空的word字符串  一、遍历整个字符串的每个字符：  （1）如果当前字符不为空，添加到word中  （2）如果当前字符为空，判断word是否存在字符，添加进res中，重置word  二、遍历结束时，判断word是否存在字符，添加进res中 |  | **[1816. 截断句子](https://leetcode-cn.com/problems/truncate-sentence/)** **（简单难度）** |
| **原地指针分割字符串** | 思路：假设从左到右遍历，  当left<len时  （1）left从左到右找到第一个非空字符，标记第一个单词的起始位置  （2）right从此时left出发,找到第一个空字符，标记第一个单词的结尾位置  （3）找到[left,right-1],即为第一个单词所在区间  （4）left=right,再次寻找下一个单词 |  |  |

这两种方法都可以处理“字符串开头、中间、结尾存在若干空格”的情况，而指针法不需要借助额外空间，效率更高效，注意记忆体会。

## 三、反转字符串

假设给定一个字符串，让你反转整个字符串或者某一小段字符串区间。反转一段字符串区间使用双指针法即可实现：左右指针分别从开头和末尾出发，交换两者即可。接下来给出其模板：



接下来结合具体题目来体会其使用。

|  |  |  |
| --- | --- | --- |
| **反转字符串区间** | **情景一：反转整个字符串**  套用模板即可，reverseString(s,0,len-1) | **[344. 反转字符串](https://leetcode-cn.com/problems/reverse-string/)（简单难度）** |
| **情景二：间隔反转字符串区间**  反转一小段字符串中的固定长度，每隔2k个字符的就将前k个字符进行反转  **思路：**  先实现一个反转指定区间[start,end]的reverse()函数，接下来问题就变成了遍历指定区间即可，可以遍历字符串，i每次+2k  反转时判断[i,i+k)是否存在，不存在则反转[i,len) | **[541. 反转字符串 II（简单难度）](https://leetcode-cn.com/problems/reverse-string-ii/)** |
| **情景三：反转一个句子中的单词**  句子字符串中存在若干单词，单词以空格间隔，反转单词的顺序，每个单词内部的字母顺序不变。  **思路：**  方法一：分割+反转。将每个单词分割成单词数组，反转下整个数组即可。  方法二：从后向前分割出每个单词添加进结果中  方法三：2次反转。先对每个单词进行reverse，再对整个字符串reverse，即可实现翻转单词顺序，但是单词中字母顺序不变的效果 | **剑指 Offer 58 - I. 翻转单词顺序（简单难度）****[151. 翻转字符串里的单词（中等难度）](https://leetcode-cn.com/problems/reverse-words-in-a-string/)****[186. 翻转字符串里的单词 II（中等难度）](https://leetcode-cn.com/problems/reverse-words-in-a-string-ii/)** |

反转字符串，很可能和分割字符串的双指针法结合。

## 四、旋转字符串

在LeetCode中，将字符串的[0,k)移动到字符串的末尾称作字符串的左旋转操作，经常求一个字符串左旋转后的字符串。

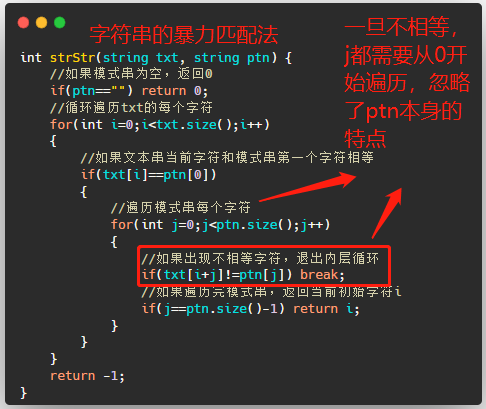
|  |  |  |
| --- | --- | --- |
| **旋转字符串** | **情景一：左旋转整个字符串**  **方法一：列表遍历拼接（借助额外内存空间）:**定义res，第一次遍历s的[n,s.length())加入res，第二次遍历s的[0,n)加入res。  **方法二：列表遍历拼接（取余操作简化）：**  定义res，遍历s的[n,s.length()+n)，res+=s[i%len]  **方法二：三次反转：**  反转s的[0,n)；  反转s的[n,len);  反转s的[0,len)； | **[剑指 Offer 58 - II. 左旋转字符串（简单难度）](https://leetcode-cn.com/problems/zuo-xuan-zhuan-zi-fu-chuan-lcof/)** |
| **情景二：判断s是否能够通过多次左旋转来得到goal**  **思路：**  s如果可以左旋转得到goal,其次数一定是有限的，假设每次左旋转一个字符，则左旋转的次数范围为[1，len]  那么遍历[1，len]，每次将s左旋转1个字符，并检查旋转后s是否等于goal即可。 | **[796. 旋转字符串（简单难度）](https://leetcode-cn.com/problems/rotate-string/)** |

## 五、匹配字符串

在匹配字符串中存在**模式串**（pat）和**文本串**（txt）2个概念，即“**在文本串（txt）中查找模式串（pat）是否存在，如果存在，返回这个字串的起始索引，如果不存在，返回-1。**”

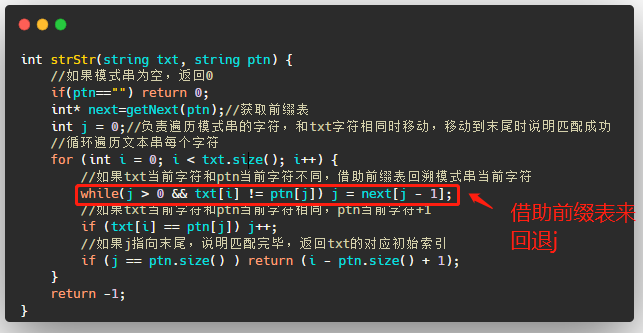
**KMP算法**（Knuth-Morris-Pratt 算法）是一个非常著名的字符串匹配算法，也是面试考察的高频点。为什么叫做KMP呢？因为该算法由Knuth，Morris和Pratt三位学者发明，所以取了三位学者名字的首字母，合称KMP。

在讲解KMP算法之前，我们先讲述下**传统的暴力匹配算法的不足之处**。



暴力匹配法采取简单直接的遍历，其时间复杂度是O（MN），即受文本串的长度N和模式串的长度M的影响，该算法的“愚蠢”之处在于处理**包含许多重复字符的模式串**时，**一旦txt[i+j]和ptn[j]不匹配，i都需要和模式串pat从头开始匹配，这就忽略了模式串本身的特点**。

KMP的主要思想是**当出现字符串不匹配时，可以知道一部分之前已经匹配的文本内容，可以利用这些信息避免从头再去做匹配了**。KMP算法借用额外的存储空间，先分析模式串ptn的特点来得到一个前缀表（prefix table），然后在文本串txt中进行匹配时，一旦不相同，就根据前缀表来获取j应该回退的位置，避免了效率浪费。



所以说KMP算法的核心就是**前缀表**，使用KMP算法就一定要用到前缀表。前缀表本质是分析**模式串ptn**的字符分布规律得来的一个数组，**其只和模式串ptn有关**，我们现在就分析下前缀表如何求得。

首先明确以下概念：

|  |
| --- |
| **前缀：**以首字符开头的不包含尾字符的**模式串（ptn）**的**连续子串**  **后缀：**以尾字符结尾的不包含首字符的**模式串（ptn）**的**连续子串**  **最长相等前后缀：**前缀和后缀中相等的最长长度 |

例如“abab”存在一个前缀“aba”和一个后缀“bab”，其相等的部分是“ab”，其长度为2.

前缀表本身就是根据动态规划法就来的，为了叙述方便，我们用next[i]替代dp[i]。

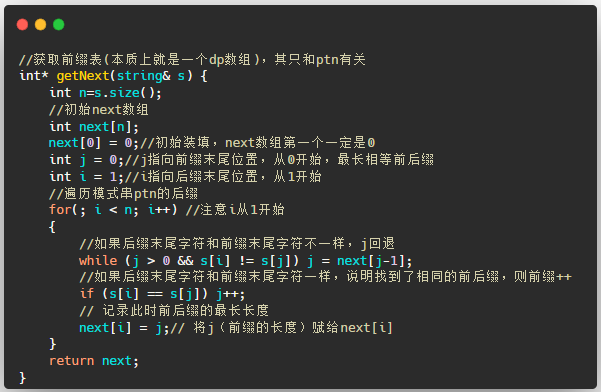
状态定义：next[i]表示模式串ptn的子串[0,i]的最长相等前后缀。

状态转移：

假设对于模式串”aabaaf”

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| i | 模式串的子串[0,i] | 最长前缀 | 最长后缀 | 最长相等前后缀 | Next[i] |
| 0 | “a” | “” | “” | 0 | 0 |
| 1 | “aa” | “a” | “a” | 1 | 1 |
| 2 | “aab” | “aa” | “ab” | 0 | 0 |
| 3 | “aaba” | “aab” | “aba” | 1 | 1 |
| 4 | “aabaa” | “aaba” | “abaa” | 2 | 2 |
| 5 | “aabaaf” | “aabaa” | “abaaf” | 0 | 0 |

next数组的实现：



|  |  |  |
| --- | --- | --- |
| **旋转字符串** | **情景一：实现strStr()**  **方法：KMP算法的实现** | **[28. 实现 strStr()（简单难度）](https://leetcode-cn.com/problems/implement-strstr/)** |
| **情景二：判断一个字符串是否是由某个连续子串重复构成**  **方法：通过前缀表来判断字符串的重复特点**  求得字符串的前缀表next  子串的长度为len-next[len-1]  想要为true满足如下条件：  （1）首先满足next[len-1]!=0  （2）len%(len-next[len-1])==0 | **[459. 重复的子字符串（简单难度）](https://leetcode-cn.com/problems/repeated-substring-pattern/)** |

## 六、字典树

字典树（Trie树）也叫前缀树、单词查找树，Trie来源于单词retrieve（检索），是一种树形数据结构，用于高效地存储和检索字符串数据集中的键。这一数据结构有相当多的应用情景，利用字符串前缀来查找指定的字符串，缩短查找时间提高查询效率，主要用于字符串的快速查找和匹配，例如自动补完和拼写检查。

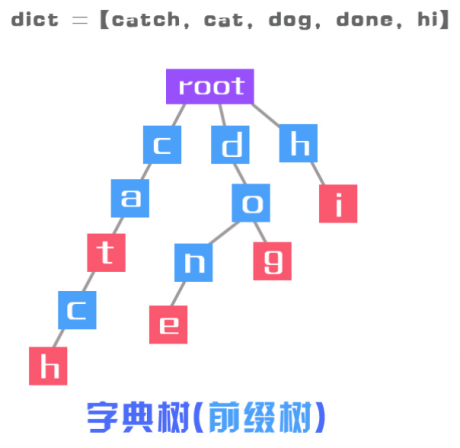
为什么要称其为字典树呢？因为Trie树的功能就像字典一样，想象一下查英文字典，我们会根据首字母找到对应的页码，接着根据第二、第三...个单词，逐步查找到目标单词，Trie树的组织思想和字典组织很像，字典树由此得名。

### 字典树的定义

字典树的核心思想是空间换时间，有3个基本性质：

|  |
| --- |
| 1. 根节点不包含字符，除根节点外每一个节点都只包含一个字符。 2. 从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串。 3. 每个节点的所有子节点包含的字符都不相同。 |

比如对单词序列lru,lua,mem,mcu建立字典树如下：

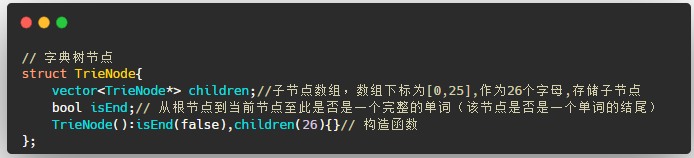


字典树建立和查询是可以同步进行的，可以在还没建立出完成的字典树之前就找到目标数据，而如果用Hash表等结构存储是需要先建立完成表才能开始查询，这也是字典树查询速度快的原因之一。

### 字典树的实现

#### （1）字典树节点

字典树是一种树形的数据结构，所以我们先定义字典树的树节点。一个字典树的树节点一般有26个子节点，所以需要一个子节点数组；另外一个节点还需要标记当前节点是否是某个单词的结尾，所以还需要一个bool变量。如下：

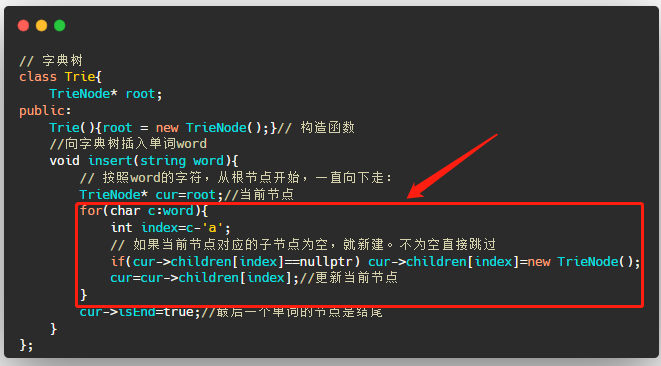


这里需要说明的是children的长度为26，是因为当前字典树计划存储字符串，所以计划每个节点存储一个英文字母，每个节点的子节点只能有26种不同值。但是这里不打算直接存储英文字符，而是巧妙地使用char-'a'的数组下标来表示不同节点对应的小写英文字母。

如果字典树不计划存储a-z的小写英文字母，则可能不是用长度26的数组，而是哈希表，所以字典树也是哈希树的一种。

#### （2）向字典树插入单词

字典树的每个节点存储一个字符，一个单词word有多少个字符就需要从根节点向下遍历几层。首先看字典树插入单词的代码：

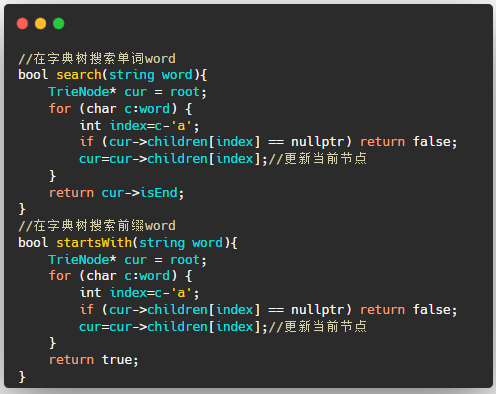


接下来简单介绍下上述代码。利用一个长度26的数组，就相当于每个节点都有26个分支，尝试状态每个分支都设为null。向字典树加入“cat”时，首先插入“c”，找到“c”对应的索引（“c”-‘a’=2），如果其为null就新建一个节点，然后继续插入“a”，找到“c”节点对应的0号索引插入，如此往复。



#### （3）搜索字典树的单词

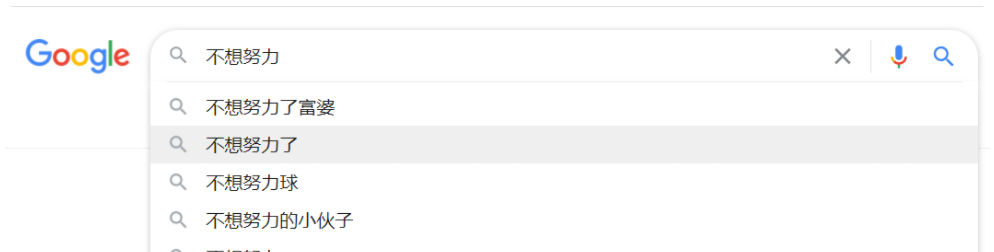
在上述插入单词的代码中，就已经包含了搜索单词的逻辑，下面直接给出代码实现。



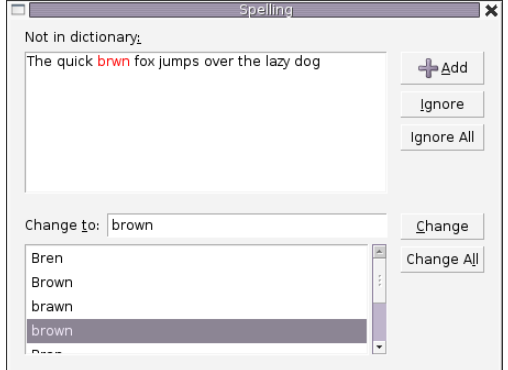
### 字典树的应用

字典树原理的应用场景如下：

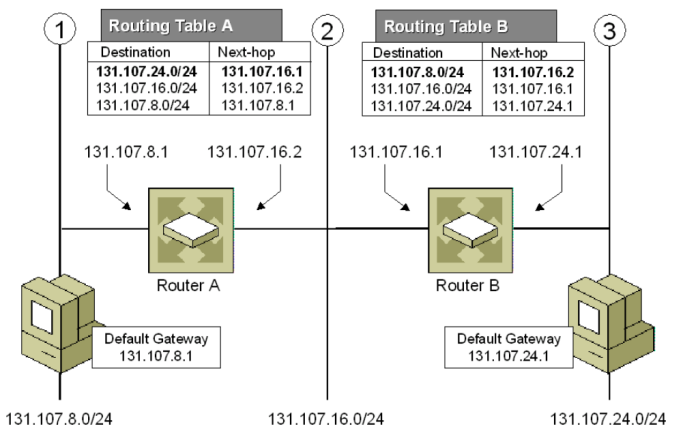
#### （1）搜索输入提示、输入法的自动补全等



#### （2）单词输入的拼写检查与修复



#### IP路由器中的最长前缀匹配算法



#### 搜索引擎的敏感词检测

面试/考试的时候很喜欢问一些关于搜索引擎的问题。这是一个经典问题，搜索引擎如何判断你搜索的内容是敏感词？

首先将搜索内容利用分词库进行分词，然后将分出的词去和敏感词库匹配。敏感词库是一个敏感词组成的Hash集合，但是匹配时是借助类似一个字典树的树形结构来匹配。

# 第3章 哈希表

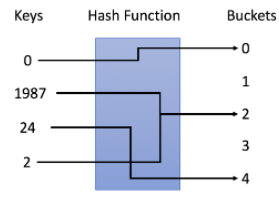
**哈希表**（hash table）也叫**散列表**，是一种通过**哈希函数**（hash function）来组织数据的数据结构，支持**快速插入**和**搜索**。本质上，哈希表就是根据**关键值key**来进行快速访问**value**的数据结构，即key哈希映射value，这种映射关系通过哈希函数来实现。

## 一、哈希表的原理是什么？

### 1、哈希表的内部原理

哈希函数的内部原理：通过**哈希函数**来实现keys和哈希表上索引数字Buckets的映射关系。编程语言内置的哈希表的**插入**和**查找**的平均时间复杂度为**O(1)**，最坏情况下是O(logN)。

假设哈希函数设为y=x%5，则可以使用如下例子来快速完成插入和搜索策略。



插入：将key输入哈希函数，得到要分配的桶号，例如1987分配给桶2，而24分配给桶4。

搜索：将key输入哈希函数得到存储的桶号，然后在特定存储桶中搜索。例如搜索1987，输入相同的哈希函数，得知1987映射到2，因此在桶2中搜索，如果在桶2中没有找到1987说明哈希表中不存在该值。

### 2、设计哈希表的2个基本因素

**哈希函数**（也叫散列函数）是哈希表中最重要的组件，x是输入的键值，y是输出的桶的索引，其选择取决于键值的范围和桶的数量。

哈希函数的设计是一个开放的问题，理想情况下，完美的哈希函数是键值和桶之间的一对一映射，然而实际情况中，哈希函数需要在桶的数量和桶的容量之间进行权衡。

**哈希冲突**是哈希表在多数情况下需要解决的问题，哈希冲突算法应该解决以下几个问题：

1. 如果组织在同一个桶内的值？
2. 如果为一个桶分配了太多的值，应该怎么办？
3. 如何在特定的桶中搜索目标值？

通常哈希表内部是通过**一种高度平衡的二叉树**来代替。

## 二、哈希集合和哈希映射的区别

|  |  |
| --- | --- |
| **[705. 设计哈希集合（简单难度）](https://leetcode-cn.com/problems/design-hashset)** | **[706. 设计哈希映射（简单难度）](https://leetcode-cn.com/problems/design-hashmap/)** |
|  |  |

## 三、C++的内置哈希结构

在LeetCode题目中，使用哈希法来解决问题时，一般会选择数组、集合（set）、映射（map）三种数据结构。在C++中，set和map分别提供以下三种数据结构，其底层实现以及优劣如下表所示：

| **集合** | **底层实现** | **是否有序** | **数值是否可以重复** | **能否更改数值** | **查询效率** | **增删效率** |
| --- | --- | --- | --- | --- | --- | --- |
| std::set | 红黑树 | 有序 | 否 | 否 | O(log n) | O(log n) |
| std::multiset | 红黑树 | 有序 | 是 | 否 | O(log n) | O(log n) |
| std::unordered\_set | 哈希表 | 无序 | 否 | 否 | O(1) | O(1) |

std::unordered\_set底层实现为哈希表，std::set 和std::multiset 的底层实现是红黑树，红黑树是一种平衡二叉搜索树，所以key值是有序的，但key不可以修改，改动key值会导致整棵树的错乱，所以只能删除和增加。虽然std::set、std::multiset 的底层实现是红黑树，不是哈希表，但是std::set、std::multiset 依然使用哈希函数来做映射，只不过底层的符号表使用了红黑树来存储数据，所以使用这些数据结构来解决映射问题的方法，我们依然称之为哈希法。

| **映射** | **底层实现** | **是否有序** | **数值是否可以重复** | **能否更改数值** | **查询效率** | **增删效率** |
| --- | --- | --- | --- | --- | --- | --- |
| std::map | 红黑树 | key有序 | key不可重复 | key不可修改 | O(log n) | O(log n) |
| std::multimap | 红黑树 | key有序 | key可重复 | key不可修改 | O(log n) | O(log n) |
| std::unordered\_map | 哈希表 | key无序 | key不可重复 | key不可修改 | O(1) | O(1) |

std::unordered\_map 底层实现为哈希表，std::map 和std::multimap 的底层实现是红黑树。同理，std::map 和std::multimap 的key也是有序的（这个问题也经常作为面试题，考察对语言容器底层的理解）。

当我们要使用集合来解决哈希问题的时候，优先使用unordered\_set，因为它的查询和增删效率是最优的，如果需要集合是有序的，那么就用set，如果要求不仅有序还要有重复数据的话，那么就用multiset。

C++中内置哈希结构的用法：

|  |  |
| --- | --- |
| unordered\_map的用法 | unordered\_set的用法 |
|  |  |

## 四、LRU算法

LRU算法（Least Recently Used），中文名称最近最少使用算法，它是一种缓存淘汰算法。该算法的主要思想是：**如果一个数据在最近一段时间没有被访问到，那么在将来它被访问的可能性也很小。所以，当指定的容量空间已存满数据时，应当把最久没有被访问到的数据淘汰**。该算法是面试中的高频考点，也有对应的Leetcode题目。

### 1、LRU算法的实现方案

现在假设让你实现一个LRU算法的缓存器，其主要由put(key,value)和get(key)两个方法构成，要求这2个**插入**和**查询**的方法的平均时间复杂度都为O(1)，那么应该如何设计？

根据LRU算法的思想，我们设计的数据结构应该具有以下特点：

|  |
| --- |
| 1.这个数据结构**必须是有时序的**，以区分最近使用的和很久没有使用的数据，当容量满了之后，要删除最久未使用的那个元素。  2.**查找**的效率必须足够快，每次找到元素时，要将设这个元素标记为最近使用的，然后返回其对应的value。  3.**插入**的效率必须足够块，每次插入元素时，要将设新的元素标记为最近使用的。如果插入后数据结构的元素数量超过最大容量，还要删除最久没有使用的元素。 |

#### （1）固定数组方案

数组中存储的每个元素是<key,value>，通过额外的时间标记变量（可以是自增的数字）来区分最近使用和很久未使用数据。

插入元素时，插入的新元素的时间标记置为0，其他元素的时间标记都自增1，数组满了就将时间标记最大的元素删除；

查找元素时，需要遍历找到对应的key，将其时间标记置为0，其他元素的时间标记都自增1。

这种固定数组方案查询和插入都需要遍历数组，所以平均时间复杂度为O(n)。

#### （2）单链表方案

单链表中存储的每个元素是<key,value>，通过链表的元素顺序区分最近使用和很久未使用数据，越靠后的元素使用时间越近。

插入元素时，新的元素必须插入到单链表末尾，如果元素个数超出最大容量，删除头结点。

查找元素时，找到的元素必须调整到单链表末尾。

这种单链表方案查询和插入都需要遍历单链表，所以平均时间复杂度也为O(n)。

#### （3）双向链表+哈希表方案

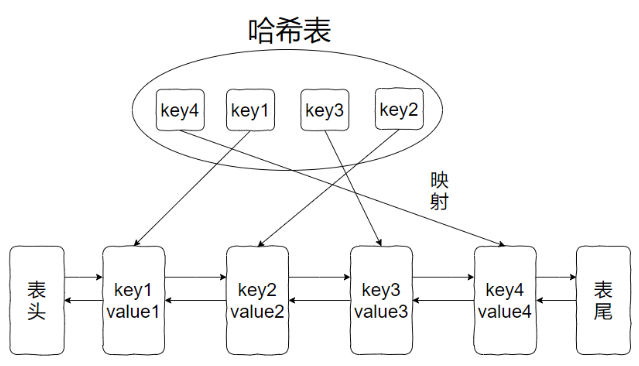
前2种方案虽然都能实现LRU算法的思想，但是其插入和查找的效率太低。如果要求**查询算法必须为O(1)**，则可以用**哈希表**。如果让链表在头部和尾部插入和删除都很快，则可以用**双向链表**，所以可以使用双向链表+哈希表的方案，即**哈希链表**（LinkedHashMap）。

哈希链表的数据结构存在一个双向链表和一个哈希映射。双向链表中的存储的每个元素是<key,value>，通过链表的元素顺序区分最近使用和很久未使用数据，越靠前的元素使用时间越近。

插入元素时，新的元素必须插入到双向链表头部，如果元素个数超出最大容量，删除尾结点，其效率都是O(1)。

查找元素时，通过哈希表的key来查找链表元素，其效率是O(1)，将查找到的元素放在双向链表的头部，只更改2个元素的指向，其效率也是O(1)。

哈希链表整体逻辑如下:



**问题1、哈希链表为什么必须得使用双向链表而不能使用单链表？**

哈希链表插入元素时，需要在链表的一端插入元素，如果超过最大容量需要在另一端删除元素，为了保证效率都为O(1)，所以必须使用双向链表。

**问题2、哈希链表中的哈希表已经保存了key，为什么双向链表的每个元素还要保存<key,value>,不能只保存value吗？**

哈希链表删除一个元素时，都是删除双向链表的一端元素，如果这个要删除的元素不是<key,value>而是<value>，那么程序就只知道删除某个value，不知道删除某个key，就无法同步删除哈希表中的key。

### 2、哈希链表的实现+LeetCode题目

哈希链表的C++实现和相关Leetcode题目如下：

|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **LeetCode题目** |
| **LRU缓存器** |  | **146 LRU缓存（中等难度）** |
| **剑指 Offer II 031. 最近最少使用缓存（中等难度）** |
| **面试题 16.25. LRU 缓存（中等难度）** |

### 3、LRU的应用

#### （1）LRU在MySQL中的应用

LRU在MySQL的应用就是缓冲池（Buffer Pool），也就是一块默认大小128M的可以读写的连续内存，被划分为若干默认大小为16KB的页，当缓存池满了的时候，就将内存中很久没有被访问过的页删除，这样就能减少磁盘IO。

不过MySQL里面并不是简单的使用了LRU算法，因为MySQL里面有一个预读功能还有全表扫描的SQL语句。预读的出发点是好的，但是有可能预读到并不需要被使用的页，这些页也被放到了链表的头部，容量不够，导致尾部元素被淘汰；全局SQL语句有可能直接访问了很多经常未使用的缓冲页。

总之，MySQL就是使用的LRU算法的变种。

#### （2）LRU在Redis中的应用

Redis就是一个key-value存储系统，是一个跨平台的非关系型数据库。其内存淘汰策略之一就是近似的LRU算法，从已设置过期时间的数据集中挑选最近最少使用的数据淘汰。没有设置过期时间的key不会被淘汰。

## 五、哈希表的简单题型

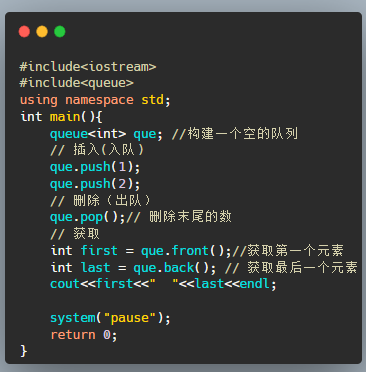
|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **LeetCode题目** |
| **找出一个序列中某个特殊的数** | 情景一：找出数组中出现次数超过一半的数字  遍历序列，统计数组中每个元素频率；  遍历哈希表，找出频率超过n/2的数 | **[169. 多数元素（简单难度）](https://leetcode-cn.com/problems/majority-element/)**  **剑指 Offer 39. 数组中出现次数超过一半的数字（简单难度）** |
| 情景二：找出数组中（第一个）只出现一次的数字  遍历序列，统计数组中每个元素频率；  遍历哈希表，找出频率==1的数  /遍历序列，找出其第一个出现一次的字符 | **[136. 只出现一次的数字（简单难度）](https://leetcode-cn.com/problems/single-number/)**  **[137. 只出现一次的数字 II（中等难](https://leetcode-cn.com/problems/single-number-ii/)[剑指 Offer II 004. 只出现一次的数字 （中等难度）](https://leetcode-cn.com/problems/WGki4K/)**  **[度）](https://leetcode-cn.com/problems/single-number-ii/)**  **[剑指 Offer II 004. 只出现一次的数字 （中等难度）](https://leetcode-cn.com/problems/WGki4K/)**  **[剑指 Offer II 070. 排序数组中只出现一次的数字（中等难度）](https://leetcode-cn.com/problems/skFtm2/)**  **387. 字符串中的第一个唯一字符（简单难度）**  **剑指 Offer 50. 第一个只出现一次的字符（简单难度）** |
| 情景三：找出数组中重复的数  遍历序列，如果哈希表中已经存在就返回，否则添加进哈希表 | **[217. 存在重复元素（简单难度）](https://leetcode-cn.com/problems/contains-duplicate/)**  **[剑指 Offer 03. 数组中重复的数字（简单难度）](https://leetcode-cn.com/problems/shu-zu-zhong-zhong-fu-de-shu-zi-lcof/)**  **[219. 存在重复元素 II（简单难度）](https://leetcode-cn.com/problems/contains-duplicate-ii/)** |
| **找出两个序列中不一样的地方** | 情景一：给定1个数组，找出其中在[0,n]中消失的数。  遍历第1个序列，统计每个元素的频率；  遍历第2个序列，查找哪个数不存在。 | **[448. 找到所有数组中消失的数字（简单难度）](https://leetcode-cn.com/problems/find-all-numbers-disappeared-in-an-array/)**  **[剑指 Offer 53 - II. 0～n-1中缺失的数字（简单难度）](https://leetcode-cn.com/problems/que-shi-de-shu-zi-lcof/)**  **面试题 17.04. 消失的数字（简单难度）** |
| 情景二：给定2个字符串，找出不一样的字符。  遍历第1个序列，统计每个元素的频率；  遍历第2个序列，字符存在则频率-1，不存在则找到。 | **[389. 找不同（简单难度）](https://leetcode-cn.com/problems/find-the-difference/)** |
| 情景三：给定2个数组，找出其交集。  遍历第1个序列，统计每个元素的频率；  遍历第2个序列，字符存在则加入结果并从哈希表中删除/频率-1。 | **[349. 两个数组的交集（简单难度）](https://leetcode-cn.com/problems/intersection-of-two-arrays/)**  **350 两个数组的交集II（简单难度）** |
| 情景四：给定2个字符串，判断a的字符是否都被b包含。  遍历第1个序列a，每个存在的元素的频率+1；  遍历第2个序列b，每个存在的字符的频率-1；  遍历哈希表，如果某个key的频率小于0则返回false； | **[383. 赎金信（简单难度）](https://leetcode-cn.com/problems/ransom-note/)** |
| 情景五：给定2个字符串，判断a和b是否是字母异位词。  遍历第1个序列a，每个存在的元素的频率+1；  遍历第2个序列b，每个存在的字符的频率-1；  遍历哈希表，如果某个key的频率！=0则返回false； | **[242. 有效的字母异位词（简单难度）](https://leetcode-cn.com/problems/valid-anagram/)**  **[剑指 Offer II 032. 有效的变位词（简单难度）](https://leetcode-cn.com/problems/dKk3P7/)** |

## 六、哈希表的中等题型

|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **LeetCode题目** |
| **双哈希法** | **情景一：判断2个字符串中的字母是否一对一唯一映射。**  思路：s字符和t字符建立一对一唯一对应的双射，需要双哈希表。  同时遍历字符串s和t，进行如下判断：  如果s字母存在映射，检查值是否与t字母一致  如果t字母存在映射，检查值是否与s字母一致  如果s字母和t字母没有映射，建立双映射  **情景二：判断1个字符串中的单词是否按照某种模式排列。**  思路：单词和字符建立一对一唯一对应的双射，需要双哈希表。  同时遍历字符串和模式串，进行如下判断：  如果当前字母存在映射，检查值是否与当前单词一致  如果当前单词存在映射，检查值已有映射和当前字母是否一致；  如果当前字母和当前单词没有映射，建立双映射 | **[205. 同构字符串（简单难度）](https://leetcode-cn.com/problems/isomorphic-strings/)**  **290. 单词规律（简单难度）** |
| **给一个字符串求可以构成的最长回文串长度** | 遍历字符串，统计每个元素的频率；  遍历哈希表，处理不同频率的元素，偶数频率字符全部可以使用，奇数频率舍弃一个其余均可使用，最长回文串长度=可使用部分长度+中心字符 | **409. 最长回文串（简单难度）** |
| **判断是否是快乐数** | 求数字N的各位的平方和sum；  如果sum==1，说明为真；  如果sum在哈希表中，说明已陷入无限循环，为假；  如果sum不在哈希表中，加入哈希表  N=sum | **202 快乐数（简单难度）** |
| **字符串数组找交集** | **情景一：找出多个字符串的共用字符（包括重复字符）**  多个字符串中频次最小的字母就是共用字符  首先获取第一个字符串中26字符的频率，设为最小频率；  循环遍历其他字符串，获取其26字符频率，和最小频率比较取最小；  将不为0的字符加入结果其最小频率次， | **[1002. 查找共用字符（简单难度）](https://leetcode-cn.com/problems/find-common-characters/)** |
| **情景二：将多个字符串按照字母异构词分类**  初始化<排序后的字符串，乱序的字符串数组>的哈希表；  遍历整个字符串数组，将每个字符串排序下，然后加入哈希表，遍历完遍得到哈希表  将哈希表的值放入结果数组中返回即可。 | **49 字母异位词分组（中等难度）**  **剑指 Offer II 033. 变位词组（中等难度）** |
| **情景三：找两个字符串数组中的交集，并且交集字符串所在的索引和最小**  建立字符串和字符串索引的哈希表；  建立当前索引和变量（初始化为最大值）  遍历第1个字符串数组，全部加入哈希表；  遍历第2个字符串数组，对每个元素做如下判断：  如果哈希表中存在当前元素，计算新的索引和：  如果新的索引和<当前索引和，清空结果，添加新索引和  如果新的索引和=当前索引和，添加新索引和 | **599. 两个列表的最小索引总和（简单难度）** |

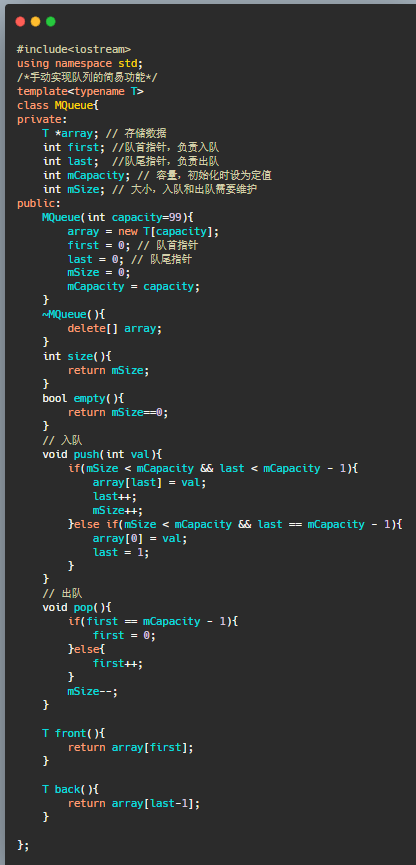
# 第4章 队列

本小节介绍首先介绍队列的使用，C++中可以直接使用已有的类：



## 用数组实现队列功能

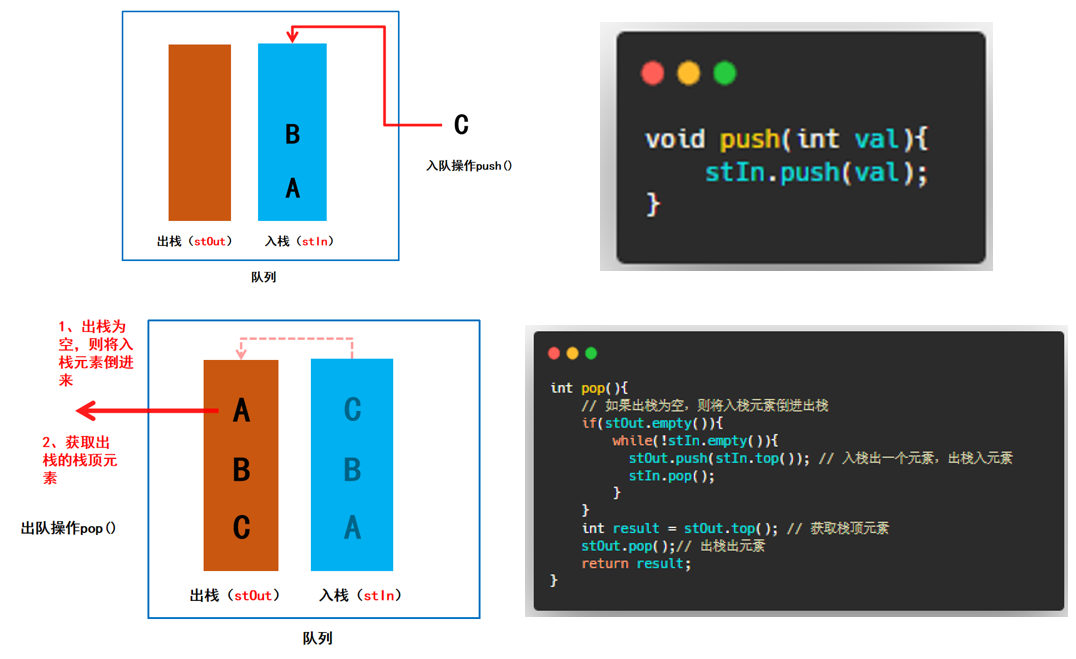
我们可以使用一个固定数组和指针来实现一个队列的入队和出队操作，其内部需要两个指针：队首指针和队尾指针，分别辅助实现出队和入队操作。



在实际做LeetCode题的时候，可以结合vector来快速实现队列的功能，但是结合vector等数组的时间复杂度较高，不适合频繁的插入操作，这个时候就需要结合链表来实现。

## 用栈实现队列功能

在leetCode中经常有用2个栈实现队列的题目，本小节介绍用栈模拟队列的思路。栈的“先进后出”的性质类似于一杯水，这里介绍用栈模拟的队列的入队、出队。假设队列内部有两个栈，分别叫做入栈stIn和出栈stOut。



全部的代码如下，注意获取队首元素和队尾元素的方法：



|  |  |  |
| --- | --- | --- |
| **LeetCode题目** | **相关链接** | **备注** |
| 232 用栈实现队列（简单难度） | [232. 用栈实现队列 - 力扣（LeetCode） (leetcode-cn.com)](https://leetcode-cn.com/problems/implement-queue-using-stacks/) |  |
| 面试题 03.04. 化栈为队（简单难度） | [面试题 03.04. 化栈为队 - 力扣（LeetCode） (leetcode-cn.com)](https://leetcode-cn.com/problems/implement-queue-using-stacks-lcci/) |  |
| 剑指 Offer 09. 用两个栈实现队列（简单难度） | [剑指 Offer 09. 用两个栈实现队列 - 力扣（LeetCode） (leetcode-cn.com)](https://leetcode-cn.com/problems/yong-liang-ge-zhan-shi-xian-dui-lie-lcof/submissions/) | 出队操作主要注意判断队列是否为空 |

## 实现双端队列功能

|  |  |  |
| --- | --- | --- |
| **LeetCode题目** | **相关链接** | **备注** |
| 232 用栈实现队列（简单难度） | [232. 用栈实现队列 - 力扣（LeetCode） (leetcode-cn.com)](https://leetcode-cn.com/problems/implement-queue-using-stacks/) |  |
| 面试题 03.04. 化栈为队（简单难度） | [面试题 03.04. 化栈为队 - 力扣（LeetCode） (leetcode-cn.com)](https://leetcode-cn.com/problems/implement-queue-using-stacks-lcci/) |  |
| 剑指 Offer 09. 用两个栈实现队列（简单难度） | [剑指 Offer 09. 用两个栈实现队列 - 力扣（LeetCode） (leetcode-cn.com)](https://leetcode-cn.com/problems/yong-liang-ge-zhan-shi-xian-dui-lie-lcof/submissions/) | 出队操作主要注意判断队列是否为空 |

# 第5章 栈

## 一、设计栈的数据结构

### 1、实现最小栈功能

这种类型题的难点在于要实现一个min函数来获取整个数据结构的最小值，并且要求时间复杂度是O(1)，请注意min函数可能连续多次调用，所以仅仅靠一个指针指向最小值是无法实现要求的。

|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **LeetCode题目** |
| **最小栈** | 思路：  使用两个栈，一个正常栈，实现栈的正常功能；一个辅助栈，存储最小的数。  push():如果新加入的元素较小就加入辅助栈；  pop():如果出栈元素和辅助栈栈顶元素一样就出栈。 | [155. 最小栈](https://leetcode-cn.com/problems/min-stack/)**（简单难度）** |
| [剑指 Offer 30. 包含min函数的栈](https://leetcode-cn.com/problems/bao-han-minhan-shu-de-zhan-lcof/)**（简单难度）** |
| [面试题 03.02. 栈的最小值](https://leetcode-cn.com/problems/min-stack-lcci/)**（简单难度）** |

### 2、验证出栈序列

这种类型题的难点在于不知道什么时候出栈什么时候入栈，出入栈的顺序多种多样的。采用模拟的思路，要出栈肯定得先入一次栈，所以一开始先入一个栈，然后如果栈顶元素和出栈序列第一个元素相同就出栈，不相同就继续入栈，实现要用一个双重循环。

|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **LeetCode题目** |
| **验证栈序列** |  | [946. 验证栈序列](https://leetcode-cn.com/problems/validate-stack-sequences/)**（中等难度）** |
| [剑指 Offer 31. 栈的压入、弹出序列](https://leetcode-cn.com/problems/zhan-de-ya-ru-dan-chu-xu-lie-lcof/)**（中等难度）** |

# 第6章 链表

链表可能是面试中被提及的最频繁的数据结构之一，其实现的代码量较少，而且需要对指针进行操作，可以很好地考察应聘者的编程功底。

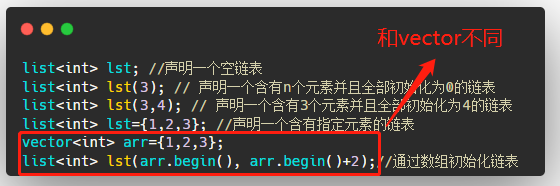
【**链表的特点**】链表是一种**链式存储结构**，在**插入**和**删除**效率上比顺序存储结构要高得多，但在**查找**和**修改**效率上不如**顺序存储结构**。链表创建时无需知道链表的长度，可以在每次插入新节点动态分配内存。

【**链表的种类**】链表大致可以分为单链表和双链表2种，单链表即单向链表，双链表即双向链表。在C++的STL中默认提供的是**双链表**<list>，LeetCode题目中考察最多的是**单链表**（因为单链表操作不方便，更容易考察编程功底）。

## 一、STL中链表的用法

首先介绍下C++中STL的链表使用方法，C++通过<list>文件来提供链表的数据结构操作，<list>是一个双向链表，其标准的用法如下：

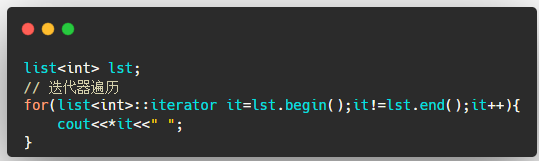
### 1、list的5种初始化方式



以上5种初始化方式中，第5中通过迭代器iterator的初始化方式和vector不同，其只能使用vector的迭代器进行初始化，请注意，list.begin()+2或者list.end()-1是违法的，会报错。（list.begin()指向）

### 2、list的遍历方式

list几乎只能通过迭代器遍历，以下是其遍历方式，注意练习：



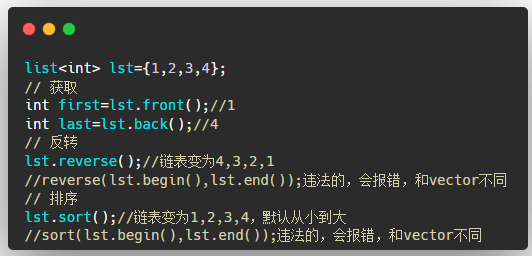
其中lst.begin()指向链表的第一个元素，lst.end()指向链表的最后一个元素的下一个，为空。

### 3、list的3种插入和删除方式

|  |
| --- |
|  |
|  |

请注意迭代器it的操作，只能it++,不能it+5，这点和vector的迭代器很不一样。

### 4、list的获取、反转、排序方式



## 二、单链表和双链表的实现

本小节介绍下单链表和双链表的实现方法，以表格的形式进行对比。在面试中很可能让手撕单链表或双链表的实现，**所以请读者一定在业余时间勤加练习**。

|  |  |  |
| --- | --- | --- |
|  | 单链表 | 双向链表 |
| 节点类型 |  |  |
| 插入操作 |  |  |
| 删除操作 |  |  |
| 获取操作 |  |  |

在上述插入、删除和获取的一般操作中，最关键的代码其实是**如何遍历找到第index个元素，这个编程技巧非常重要**，上述红框中的代码其实就是**计数法**，其在LeetCode题目中也会经常用到。**在链表中如何找到第index个元素，**有2种方法：**计数法**和**双指针法**：

计数法适用于index容易求得的情形，例如正数/倒数第k个，中间节点等。

双指针法使用于index不容易求解的情形，可以通过设置快慢指针的不同速率来达到找指定区间、判断环等复杂问题。以下将会在leetcode题目中进入深刻体会。

## 三、删除链表中的值

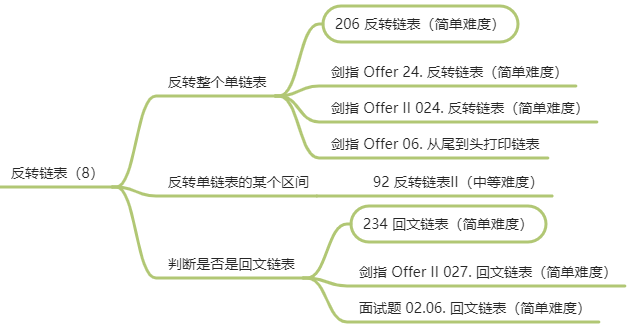
LeetCode中删除链表的题目一般是操作单链表，要删除某个元素就必须定位到待删除元素的上一个节点，修改上一个节点的next指针。



|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **相关LeetCode题目** |
| **删除单链表中的某个特定值元素/相同元素（双指针）** |  | **203 移除链表元素（简单难度）** |
| **剑指 Offer 18. 删除链表的节点（简单难度）** |
| [83. 删除排序链表中的重复元素](https://leetcode-cn.com/problems/remove-duplicates-from-sorted-list/)（简单难度） |
| **删除链表中的一段特殊序列（动态指针区间）** |  | **82. 删除排序链表中的重复元素II（中等难度）** |
|  |  | **1171. 从链表中删去总和值为零的连续节点（中等难度）** |
| **删除单链表中的倒数第N个节点/正中间节点（遍历到指定位置）** |  | **剑指 Offer 22. 链表中倒数第k个节点（简单难度）** |
| **19 删除链表的倒数第N个节点（中等难度）** |
| **剑指 Offer II 021. 删除链表的倒数第 n 个结点（中等难度）** |
| **2095. 删除链表的中间节点（中等难度）** |
| **只给出待删除元素，实现删除效果（类似于脑筋急转弯）** |  | **237 删除链表中的节点（简单难度）** |
| **面试题 02.03. 删除中间节点（简单难度）** |

## 四、反转链表

反转链表是LeetCode题目中的高频知识点，请务必掌握，由于LeetCode题目中的链表题目节点是自定义的，**无法直接调用反转函数，常常需要自己手写**。

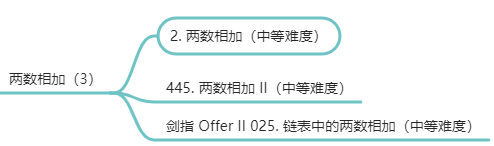


相关LeetCode题目有：

|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **LeetCode题目** |
| **反转整个单链表** |  | **206 反转链表（简单难度）** |
| **剑指 Offer 24. 反转链表（简单难度）** |
| **剑指 Offer II 024. 反转链表（简单难度）** |
| **剑指 Offer 06. 从尾到头打印链表** |
| **反转单链表的某个区间[left,right]** |  | **92 反转链表II（中等难度）** |
| **判断是否是回文链表（反转后半部分，比较前后两半部分）** |  | **234 回文链表（简单难度）**  **剑指 Offer II 027. 回文链表（简单难度）**  **面试题 02.06. 回文链表（简单难度）** |

## 五、链表元素相加

**用链表元素相加不能将链表中的元素提取出来组成一个数再相加**，因为链表的元素数量可能是非常大的，转换成数字相加可能会超过int或long的大小限制。所以推荐的思路是：同时遍历2个链表来相加，这个模板也可以用来**解决大数相加问题**。



|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **LeetCode题目** |
| **2个链表的元素相加** |  | **2. 两数相加（中等难度）** |
| **445. 两数相加 II（中等难度）** |
| **剑指 Offer II 025. 链表中的两数相加（中等难度）** |

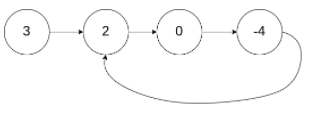
## 六、链表相交和环形链表

### 1、找到2个单链表的的相交节点

|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **LeetCode题目** |
| **2个单链表的相交节点** | 如下图，找到A和B两个节点的第一个相交节点（c1）：    思路：分别求出lenA和lenB，如果lenA>lenB，则让遍历指针pA先走lenA-lenB个节点，然后和pB一起走，当pA==pB时则找到了相交节点。 | **[160. 相交链表（简单难度）](https://leetcode-cn.com/problems/intersection-of-two-linked-lists/)** |
| **[面试题 02.07. 链表相交（简单难度）](https://leetcode-cn.com/problems/intersection-of-two-linked-lists-lcci/)** |
| **[剑指 Offer 52. 两个链表的第一个公共节点（简单难度）](https://leetcode-cn.com/problems/liang-ge-lian-biao-de-di-yi-ge-gong-gong-jie-dian-lcof/)** |
| **[剑指 Offer II 023. 两个链表的第一个重合节点（简单难度）](https://leetcode-cn.com/problems/3u1WK4/)** |

### 2、判断1个单链表的环和环的入口

如下图，找到单链表的环的入口（2）：



如果链表中有某个节点，可以通过连续跟踪 next 指针再次到达，则链表中存在环。使用快慢指针法，分别定义fast和slow指针，

（1）都从头结点出发，fast指针每次移动两个节点，slow指针每次移动一个节点，如果出现fast==slow，说明这个链表有环。

（2）当fast==slow，此时两个指针分别从head和相遇点以相同速度同时出发，再次相遇的节点即是环的入口。

|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **LeetCode题目** |
| **判断1个单链表的环和环的入口** |  | **[141. 环形链表（简单难度）](https://leetcode-cn.com/problems/linked-list-cycle/)** |
| **[142. 环形链表 II（中等难度）](https://leetcode-cn.com/problems/linked-list-cycle-ii/)** |
| **[剑指 Offer II 022. 链表中环的入口节点（中等难度）](https://leetcode-cn.com/problems/c32eOV/)** |

## 七、合并链表

|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **LeetCode题目** |
| **拼接链表** | 使用计数法找到单链表的特定节点 | **1669. 合并两个链表（中等难度）** |
| **合并2个有序链表** |  | **21 合并两个有序链表（简单难度）** |
| **剑指 Offer 25. 合并两个排序的链表（简单难度）** |
| **88 合并两个有序数组（简单难度）** |
| **面试题 10.01. 合并排序的数组（简单难度）** |

## 八、循环链表

循环链表就是在单链表的尾元素指向头元素，其一般保存头指针和尾指针。

|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **LeetCode题目** |
| **将单链表中的每个节点向右移动k个位置。** | 将单链表的元素向右移动k个元素,如下图：    思路：考虑k和单链表长度len的关系：   1. k==len时，单链表不变； 2. K!=len时，如果k>len,k=k%len，将单链表变为循环链表，找到第len-k个节点，即尾指针所在节点（如上图中的rotate1的4、rotate2的3），其next即新的头指针，最终解除循环链表。 | [61. 旋转链表](https://leetcode-cn.com/problems/rotate-list/)（中等难度） |
| **在一个有序的循环链表中插入一个新的节点** | 在一个有序的循环链表中插入一个新节点：    分析：  （1）如果插入值大于循环链表的最大值或者小于循环链表的最小值，插入tail之后即可  （2）如果插入值介于最小值和最大值之间，则遍历循环链表，找到一个满足**p指针的值<=插入值<=p->next的值**的位置  所以解题首先需要遍历循环链表找到链表中的最大值所在节点，将其设为tail，tail的next必定是最小值 | [剑指 Offer II 029. 排序的循环链表](https://leetcode-cn.com/problems/4ueAj6/)（中等难度） |

## 九、分隔链表

|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **LeetCode题目** |
| **将单链表中的不同节点分隔开来** | 将单链表的奇数节点和偶数节点分割开来,如下图：    思路：不要在一个单链表中操作，新建一个front的链表头结点用来存放奇数节点，新建一个back的链表头结点用来存放偶数节点。遍历一次整个链表，是奇数时就放入front链表，是偶数就放入back链表。最终将back链表的next设为空，将front链表的next设为back链表的头结点，返回front链表的头结点。 | **[328. 奇偶链表](https://leetcode-cn.com/problems/odd-even-linked-list/)（中等难度）** |
| 将单链表中比特定值（3）小的数排到比特定值（3）大的数之前，不同节点的相对顺序不变：    思路：不要在一个单链表中操作，新建一个front的链表头结点用来存放比特定值小的数，新建一个back的链表头结点用来存放比特定值大或相等的数。遍历一次整个链表，比特定值小时就放入front链表，比特定值大或相等就放入back链表。最终将back链表的next设为空，将front链表的next设为back链表的头结点，返回front链表的头结点。 | [86. 分隔链表](https://leetcode-cn.com/problems/partition-list/)（中等难度） |
| **面试题 02.04. 分割链表（中等难度）** |

## 十、复制链表

本小节介绍下复制链表的题型，这也是链表题目的高频题型，也不难，目前只有一种类型：复制带随机指针的单链表。我们接下来分析下这种题的难点。



复制一个单链表的思路很简单，就是直接遍历原始链表，每次新建节点，然后让其指向下一个节点即可，其简单之处在于，每个节点都只有确定next指针，next指针是明确的，一定指向下一个节点。

复制带随机指针的单链表，其每个节点除了next指针还有一个random指针，这个random指针的指向是不确定的，所以其无法在一次遍历原始链表中就同时指定其random指针，因为random可能指向后面未创建的节点。

我们的思路是遍历两次原始链表：第一次遍历原始链表来创建其对应的新节点；第二次遍历原始链表来修改每个新节点的next指针和random指针。

由于两次遍历是分开的，所以我们需要用哈希表建立原始链表节点与新建链表节点的一对一映射。

|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **LeetCode题目** |
| **复制带随机指针的单链表** | 思路：  （1）第一次遍历原始链表，每次创建一个新节点，和老节点一起存入哈希表。  （2）第二次遍历原始链表，每次获取一个新节点，根据老节点映射修改每个新节点的next指针和random指针。 | **138. 复制带随机指针的链表（中等难度）** |
| **剑指 Offer 35. 复杂链表的复制（中等难度）** |

## 十一、链表与二叉树

## 十二、排序链表

### 1、调整链表特定节点的顺序

|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **LeetCode题目** |
| **分隔链表+合并链表** | 将单链表按照第1个和倒数第1个放在一起的形式排列,如下图：    思路：  （1）获取链表中间节点将链表分为前半部分和后半部分  （2）反转链表的后半部分。  （3）将链表的前半部分和后半部分合并为新链表 | **143 重排链表（中等难度）** [剑指 Offer II 026. 重排链表](https://leetcode-cn.com/problems/LGjMqU/)（中等难度） |
| 两两交换相邻节点：    思路一：   1. 分割链表，奇数节点和偶数节点各自存放在一个链表中。 2. 合并链表，按照偶数节点先放的顺序合并为一个新链表。   思路二：  使用指针cur，每次交换cur->next和cur->next->next。 | **24 两两交换链表中的节点（中等难度）（中等难度）** |

### 2、用插入排序给链表排序

|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **LeetCode题目** |
| **分隔链表+合并链表** | 思路：新建虚拟头结点，定义最后一个排序好的节点，遍历整个链表  （1）如果当前值>最后一个排序好的节点，+1  （2）如果当前值<最后一个排序好的节点，从头遍历寻找插入位置 | **[147. 对链表进行插入排序（中等难度）](https://leetcode-cn.com/problems/insertion-sort-list/)** |

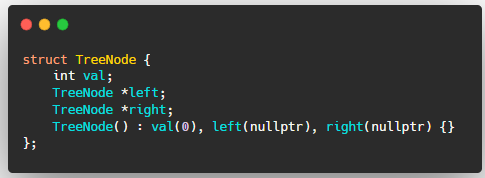
### 3、用归并排序给链表排序

|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **LeetCode题目** |
| **链表排序** | 思路：  （1）获取链表中间节点将链表分为前半部分和后半部分  （2）递归排序左右两半部分。  （3）合并2个有序链表 | [148. 排序链表（中等难度）](https://leetcode-cn.com/problems/sort-list/)[剑指 Offer II 077. 链表排序（中等难度）](https://leetcode-cn.com/problems/7WHec2/) |

# 二叉树

## 普通二叉树

本小节二叉树节点定义如下：



在介绍二叉树的算法前请先回忆下容器、栈和队列的API用法：

#include<vector>  
#include<stack>  
#include<queue>

|  |  |
| --- | --- |
| 容器的使用 | 创建对象：vector<Type> M  是否为空：M.empty()  查看大小：M.size()  添加：M.push\_back()  反向：reverse(result.begin(), result.end()); |
| 栈的使用 | 创建对象：stack<Type> M  是否为空：M.empty()  查看大小：M.size()  入栈：M.push()  出栈：M.top()（获取）；M.pop()（删除） |
| 队列的使用 | 创建对象：queue<Type> M  是否为空：M.empty()  查看大小：M.size()  入队：M.push()  出队：M.front()（获取）；M.pop()（删除） |

### 1、基础遍历

以下顺口溜可以帮助读者记忆二叉树基础遍历的迭代法流程：

|  |
| --- |
| **先序遍历先入栈，栈不为空就遍历；**  **出栈一定要处理，先右后左慢入栈；**  **中序遍历不先入，指示与栈全不空；**  **存在入栈换左边，不存出栈换右边；**  **后序遍历同先序，先左后右记反向；**  **层序遍历同先序，借用队列先左右。** |

#### （1）深度优先遍历

##### A、递归法（先序遍历为例）



|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **LeetCode题目** |
| **二叉树的先序遍历** | 迭代法思路：  先序遍历先入栈，栈不为空就遍历；  出栈一定要处理，先右后左慢入栈； | **[144. 二叉树的前序遍历](https://leetcode-cn.com/problems/binary-tree-preorder-traversal/)（简单难度）****589. N 叉树的前序遍历（简单难度）****114. 二叉树展开为链表（中等难度）** |
| **二叉树的中序遍历** | 中序遍历不先入，  指示与栈全不空；  存在入栈换左边，  不存出栈换右边； | **[94. 二叉树的中序遍历](https://leetcode-cn.com/problems/binary-tree-inorder-traversal/)（简单难度）** |
| **二叉树的后序遍历** | 后序遍历同先序，  先左后右记反向； | **[145. 二叉树的后序遍历](https://leetcode-cn.com/problems/binary-tree-postorder-traversal/)（简单难度）****590. N 叉树的后序遍历（简单难度）** |

##### 迭代法

#### （2）广度优先遍历

|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **LeetCode题目** |
| **二叉树的层序遍历（一维输出）** | 迭代法思路：  **层序遍历同先序，借用队列先左右**。 | **面试题32 - I. 从上到下打印二叉树（中等难度）** |
| **二叉树的层序遍历（二维输出）** | 迭代法思路： | **情景1：按层输出**  **102 二叉树的层序遍历（中等难度）**  **剑指 Offer 32 - II. 从上到下打印二叉树 II（简单难度）**  **429 N叉树的层序遍历（中等难度）**  **情景2：某层反转**  **107 二叉树的层序遍历II（中等难度）**  **剑指 Offer 32 - III. 从上到下打印二叉树 III（中等难度）**  **情景3：按层输出最后一个**  **199 二叉树的右视图（中等难度）**  **剑指 Offer II 046. 二叉树的右侧视图（中等难度）**  **情景4：寻找某层的特殊值**  **515 在每个树行中找最大值（中等难度）**  **剑指 Offer II 044. 二叉树每层的最大值（中等难度）**  **513 找树左下角的值（中等难度）**  **剑指 Offer II 045. 二叉树最底层最左边的值（中等难度）**  **637 二叉树的层平均值（简单难度）**  **情景5：修改每层的指针**  **116 填充每个节点的下一个右侧结点指针（中等难度）**  **117 填充每个节点的下一个右侧结点指针II（中等难度）** |

### **2、从数组中构造二叉树**

二叉树的遍历算法是输入二叉树，输出数组；二叉树的数组转换是输入数组，输出二叉树。

#### 主要模板

输入数组，使用递归法构建一个二叉树的主要思路可以归纳为：

|  |
| --- |
| 第一步：找数组的切割点，确定中间节点；  第二步：用切割点将数组切割为左右子数组；  第三步：分别递归调用左右子数组。 |

算法流程简单总结为：2个终止条件，1次切割数组，2次递归调用。

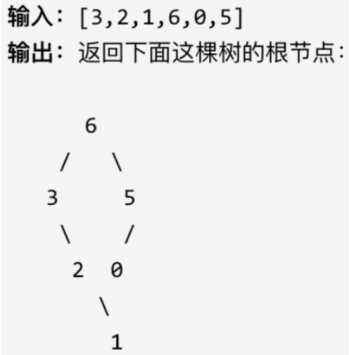


#### 数组转换为最大二叉树

给定⼀个不含重复元素的整数数组，构建一个最⼤⼆叉树，并且输出其根节点。

⼀个以此数组构建的最⼤⼆叉树定义如下：

* ⼆叉树的根是数组中的最⼤元素。
* 左⼦树是通过数组中最⼤值左边部分构造出的最⼤⼆叉树。
* 右⼦树是通过数组中最⼤值右边部分构造出的最⼤⼆叉树。



|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **LeetCode题目** |
| **数组转换为最⼤⼆叉树** | 思路：  第一步：找数组的最大值作为切割点，确定中间节点；  第二步：用切割点将数组切割为左右子数组  第三步：分别递归调用左右子数组 | **654 最大二叉树（中等难度）** |

#### （3）遍历数组确定唯一二叉树

在二叉树算法题型中中最常见的一类题型是根据遍历结果来推断二叉树结构，一般分两类：

给定先序遍历和中序遍历，确定唯一一个二叉树；

给定中序遍历和后序遍历，确定唯一一个二叉树。

可以发现，给定的遍历结果中**一定有中序遍历**，请注意以下三条规律是设计该算法的依据：

|  |
| --- |
| （1）中序遍历结果负责判断左右子树，在中间节点左侧一定属于其对应左子树，在右侧一定属于其右子树；  （2）前序遍历结果第一个一定是中间节点，其后依次是其左子树节点和右子树节点；  （3）后序遍历结果最后一个一定是中间节点，其前依次是其右子树节点和左子树节点。 |

相关题目

|  |  |  |
| --- | --- | --- |
| **类型** | **解题模板** | **LeetCode题目** |
| **前序遍历和中序遍历** | 思路：  **切割中序数组**  第一步：遍历中序数组，找到和先序数组第一个值相等的点的索引作为中序数组的切割点；  第二步：用切割点将中序数组切割为左右子数组。  **切割先序数组**  第三步：找先序数组的切割点，确定中间节点（先序数组的切割点等于去除第一个元素后的中序数组左子数组的长度）；  第四步：用切割点将先序数组切割为左右子数组  第五步：分别递归调用左右子数组 | **105 从前序与中序遍历序列构造二叉树（中等难度）**  **剑指 Offer 07. 重建二叉树（中等难度）** |
| **后序遍历和中序遍历** | 思路：  区别仅仅在于中间节点由第一个元素变成了最后一个元素确定： | **106 从中序与后序遍历序列构造二叉树（中等难度）** |

### 3、自顶向下遍历树

二叉树的先序遍历很可能是最常见的遍历方式，因为它按照中左右的顺序遍历，符合二叉树自顶向下的查找逻辑。

#### 递归遍历1个二叉树

|  |  |  |
| --- | --- | --- |
| **水平翻转二叉树** | 思路：  “翻转二叉树只需要将树中的每⼀个节点的左右子节点交换⼀下即可” | **226 翻转二叉树（简单难度）**  **剑指 Offer 27. 二叉树的镜像（简单难度）** |

#### （2）同时递归遍历2个二叉树

|  |  |  |
| --- | --- | --- |
| **判断两个树是否相同** | 思路：  先判断中间节点是否相等，再判断中间节点的左节点是否相等，再判断右节点是否相等，直至遍历完整个二叉树，全部相等才返回true。 | **100 相同的树（简单难度）** |
| **合并二叉树（对应节点相加）** |  | **617 合并二叉树（简单难度）** |
| **判断二叉树是否对称** |  | **101 对称二叉树（简单难度）**  **剑指 Offer 28. 对称的二叉树（简单难度）** |
| **判断一个树是否是另一个数的子树** | 思路：  subRoot是root的子树，只有以下三种情况：   * root和subRoot相等； * subRoot是root的左子树的子树； * subRoot是root的右子树的子树；   所以最终还是回归到判断两个树是否相等 | **572 另一个树的子树（简单难度）** |
| **判断一个树是否是另一个数的子结构** | 思路：  和判断子树的代码几乎一模一样，注意子树与子结构的区别 | **剑指 Offer 26. 树的子结构（中等难度）** |

### 自顶向下计算属性

二叉树的先序遍历很可能是最常见的遍历方式，因为它按照中左右的顺序遍历，符合二叉树自顶向下的查找逻辑。

#### 求二叉树的节点个数

|  |  |  |
| --- | --- | --- |
| **求二叉树的节点个数** | 思路：  求出一棵 完全二叉树 的节点个数。  “中间节点为根的树的节点数”=“左节点为根的树的节点数”+“左节点为根的树的节点数”+1 | **222 完全二叉树的节点个数（中等难度）** |

#### 求树的深度

在LeetCode中我们还会遇到一类求解二叉树的最大深度和最小深度，两者的区别：

二叉树的最大深度：从根节点到最远叶子节点的最长路径上的节点数量。

二叉树的最小深度：从根节点到最近叶子节点的最短路径上的节点数量。

这两者的重点在于**最远叶子节点**和**最近叶子节点**的判断上。

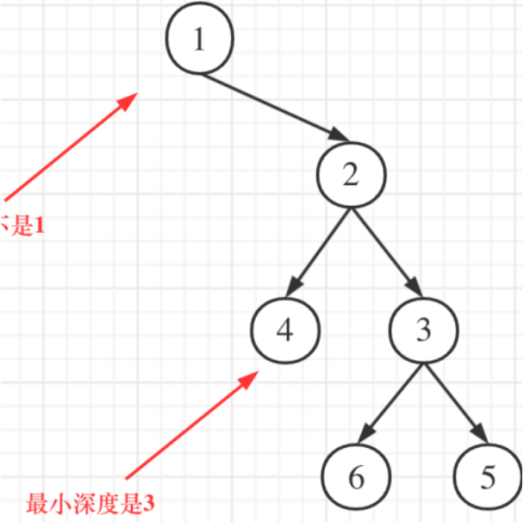
最远叶子节点的终止条件是该叶子节点的左右子节点为空

最近叶子节点的终止条件则更复杂，需要判断最近的情况，有两种特殊情况：

（1）当左⼦树为空，右⼦树不为空时，“中间节点的最小深度”！=min（“左节点的深度”，“左节点的深度”）+1；

（2）当左⼦树不为空，右⼦树为空时，“中间节点的最小深度”！=min（“左节点的深度”，“左节点的深度”）+1；

举个例子：



|  |  |  |
| --- | --- | --- |
| **求二叉树的最大深度** | 思路：  “中间节点为根的树的最大深度”=“左节点的最大深度”+“右节点的最大深度”+1 | **104 二叉树的最大深度（简单难度）**  **剑指 Offer 55 - I. 二叉树的深度（简单难度）** |
| **求N叉树的最大深度** |  | **559 N叉树的最大深度（简单难度）** |
| **求二叉树的最小深度** |  | **[111. 二叉树的最小深度（简单难度）](https://leetcode-cn.com/problems/minimum-depth-of-binary-tree/)** |

#### 求树的直径

二叉树的直径等价于二叉树的每个节点的左右子树的最大深度的最大值。

|  |  |  |
| --- | --- | --- |
| **求树的直径** |  | **543. 二叉树的直径（简单难度）** |

#### 平衡二叉树

在LeetCode中与求高度相关的一类问题是判断是不是平衡二叉树。高度平衡二叉树的定义：一个二叉树每个节点的左右两个子树的高度差的绝对值不超过1。

现在我们用递归法来思考这个问题，首先将问题规模缩小到“根节点及其左子树和右子树”上：

|  |
| --- |
| 如果是一个空树，其满足平衡二叉树要求；  如果只有一个根节点，其满足平衡二叉树要求；  如果根节点的左子树不是平衡二叉树，则二叉树不满足平衡二叉树要求；  如果根节点的右子树不是平衡二叉树，则二叉树不满足平衡二叉树要求；  如果根节点的左右子树都是平衡二叉树，但是左右子树的高度绝对值超过1，则二叉树不满足平衡二叉树要求。 |

综上，我们可以发现，我们将判断二叉树的问题变得与判断两个左右子树是否是二叉树有关，而判断一个左右子树是否是二叉树还与其高度绝对值有关。

|  |  |  |
| --- | --- | --- |
| **判断是否是平衡二叉树** |  | **110 平衡二叉树（简单难度）**  **剑指 Offer 55 - II. 平衡二叉树（简单难度）** |

### 自底向上寻找祖先

一个节点也可以自己的祖先，2个节点p、q的最近公共祖先x是指x是p、q的祖先并且深度尽可能大，最近公共祖先也可能是自身。如何判断两个节点的最近公共祖先？

如果当前节点的左子树出现过结点p或q且右子树出现过节点p或q，则当前节点最可能是最近公共祖先；

如果当前节点的左子树出现过节点p或q但右子树没出现过节点p或q，则当前节点的左子树中最可能有最近公共祖先；

如果当前节点的左子树没出现过节点p或q但右子树出现过节点p或q，则当前节点的右子树中最可能有最近公共祖先；

如果当前节点的左子树没出现过节点p或q且右子树没出现过节点p或q，则当前节点的子树中不可能有最近公共祖先；

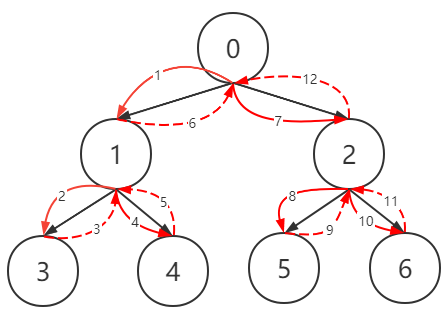
|  |  |  |
| --- | --- | --- |
| **寻找最近公共节点** |  | **236二叉树的最近公共祖先（中等难度）**  **剑指 Offer 68 - II. 二叉树的最近公共祖先（简单难度）** |

### 自顶向下回溯路径

二叉树的先序遍历很可能是最常见的遍历方式，因为它按照中左右的顺序遍历，符合二叉树自顶向下的查找逻辑。

#### 查找所有路径

二叉树的路径是从根节点出发到叶节点结束。二叉树中查找路径需要借助回溯思想，在下图中，实线部分为递归，虚线部分为回溯。



观察上图，我们可以发现以下规律：

(1)每次递归在Path中添加一个节点值，每次回溯在Path中去除一个节点值

(2)当遇到叶子节点时，说明得到一条路径，返回该路径；

(3)当遇到内部节点时，判断其是否左节点，存在则递归左节点，递归结束后回溯左节点；判断其是否右节点，存在则递归右节点，递归结束后回溯右节点。

|  |  |  |
| --- | --- | --- |
| **查找二叉树的所有路径** |  | **257 二叉树的所有路径（简单难度）** |

#### 查找目标和为定值的一条路径

给定一个二叉树和一个目标和，判断该树中是否存在根节点到叶子节点的路径，这条路径上所有节点值相加等于目标和。

|  |  |  |
| --- | --- | --- |
| **查找二叉树的某条路径总和** |  | **112 路径总和（简单难度）** |

#### 查找目标和为定值的所有路径

给定一个二叉树和一个目标和，找到所有从根节点到叶子节点路径总和等于给定目标和的路径。

|  |  |  |
| --- | --- | --- |
| **查找二叉树的所有路径总和** |  | **113 路径总和II（中等难度）**  **剑指 Offer 34. 二叉树中和为某一值的路径（中等难度）** |

#### 查找节点和为定值的数目

该题目中的难点在于路径不需要从根节点开始，也不需要在叶子节点结束，但是路径方向必须是向下的（只能从父节点到子节点）。

|  |  |  |
| --- | --- | --- |
| **查找二叉树的所有“节点和”的数目** |  | [437. 路径总和 III](https://leetcode-cn.com/problems/path-sum-iii/)（中等难度） |

#### 计算左叶子节点和

|  |  |  |
| --- | --- | --- |
| **计算二叉树的所有左叶子节点的和** |  | **404 左叶子之和（简单难度）** |

## 二叉搜索树

二叉搜索树是一种特殊的二叉树，它是一个有序树：

|  |
| --- |
| 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值；  若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值；  它的左、右子树也分别为二叉搜索树。 |

请特别注意，二叉搜索树的最大特点就是**它是有序树**。二叉搜索树的题目整体上比普通二叉树的题目要简单。

### 有序查找和插入

因为是有序树，所以不需要全部遍历，需要有条件的递归。

|  |  |  |
| --- | --- | --- |
| **寻找最近公共节点** |  | **700 二叉搜索树中的搜索（简单难度）** |

# 第8章 排序算法

排序算法是最基本的算法之一。排序即对一系列对象根据某个关键字进行排序。首先介绍一下排序术语：5a,6，2,3，5b

|  |
| --- |
| **稳定**：a=b，排序前a在b之前，排序后a仍然在b之前  **不稳定**：a=b，排序前a在b之前，排序后a可能在b之后  **内排序**：所有排序操作都可以在内存中完成  **外排序**：当数据太大时需要将数据放在磁盘中  **时间复杂度**：一个算法执行所耗费时间的估计值  **空间复杂度**：一个算法执行所耗费空间的估计值 |

排序算法分类如下：

根据排序方式，排序算法整体上可以划分为内部排序和外部排序。内部排序是指数据记录在内存中进行排序，不需要申请额外的存储空间。外部排序是指在排序过程中需要申请额外的内存空间。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **排序方式** | **排序算法** | **最好情况** | **最坏情况** | **平均时间复杂度** | **空间复杂度** | **稳定性** |
| In-place | 选择排序 |  |  |  |  | 不稳定 |
| 冒泡排序 |  |  |  |  | 稳定 |
| 插入排序 |  |  |  |  | 稳定 |
| 快速排序 |  |  |  |  | 不稳定 |
| 希尔排序 |  |  |  |  | 不稳定 |
| 堆排序 |  |  |  |  | 不稳定 |
| Out-place | 归并排序 |  |  |  |  | 稳定 |
| 计数排序 |  |  |  |  | 稳定 |
| 桶排序 |  |  |  |  | 稳定 |
| 基数排序 |  |  |  |  | 稳定 |

（注意：n指的是数据规模；k指的是“桶”的个数；In-place指的是占用常数内存，不占用额外内存；Out-place指的是占用额外内存）

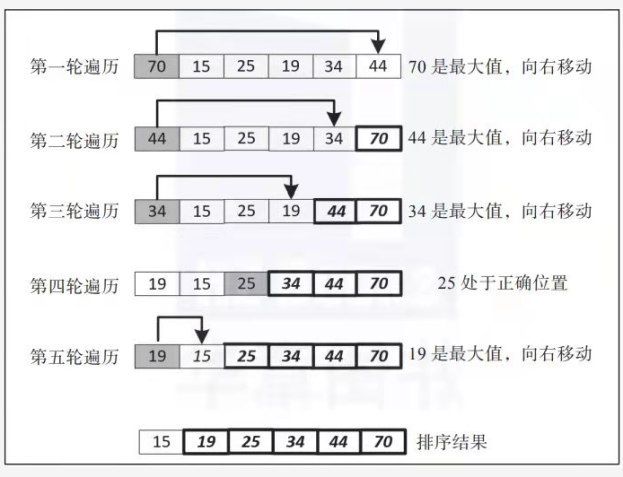
本章节接下来的测试代码可以在Leetcode 912题上编译运行，其中希尔排序（三分之一增量版本）、堆排序、归并排序不会超时。

## 一、内排序

### 1、选择排序

选择排序（selection-sort）是性能鲁棒性最好的排序算法之一，任何情况下耗费时间都为，而且不占用额外内存，适合于数据规模n较小的情景。

**【算法思路】**：从未排序序列中找到最小（大）元素，存放到排序序列的末尾位置，直至未排序序列为空。



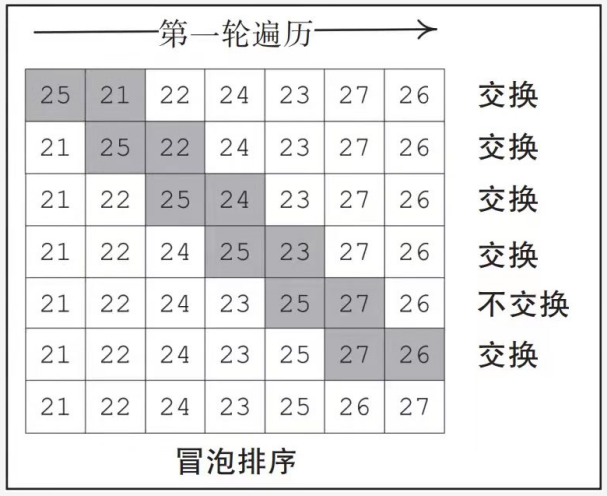
**【编码思路】：**双层循环，外层循环负责从左至右确定已排序序列，内层循环负责从左至右在未排序序列中查找最值，找到后交换两数即可。



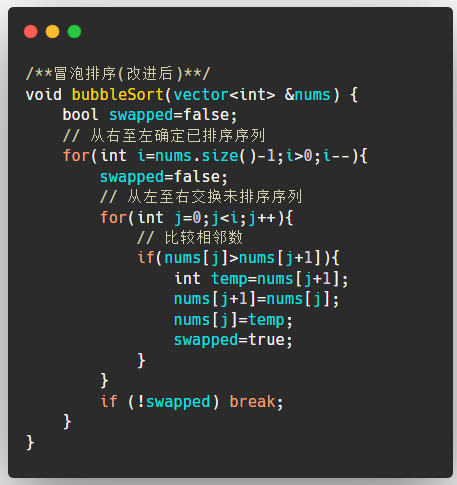
### 2、冒泡排序

冒泡排序（bubble-sort）比选择排序性能稍好一定，而且稳定，排序不影响相同数的位置，平均时间复杂度为，同样不占用额外内存，适合于数据规模n较小的情景。

**【算法思路】**：连续进行n轮遍历，每一轮遍历都让相邻数两两交换，确定一个已排序的数，n轮遍历后就会确定所有的数。



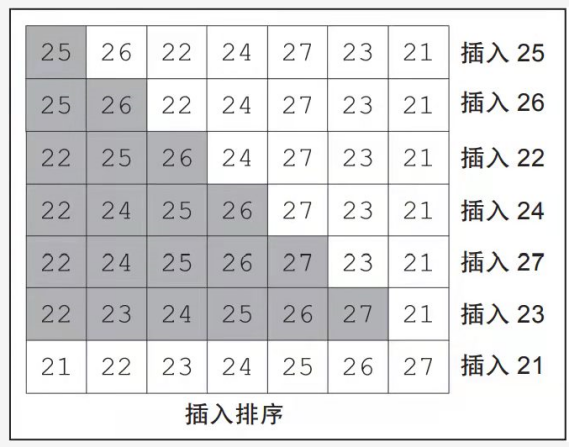
**【编码思路】：**双层循环，外层循环负责从右至左确定已排序序列，内存循环负责从左至右交换未排序序列中的相邻数。此处有个改进小技巧，内层循环维护一个bool值，如果当前一轮遍历没有交换过说明已完成排序，直接退出外层循环。



### 3、插入排序

插入排序（insert-sort）的性能和冒泡排序差不多，平均时间复杂度为，同样不占用额外内存，适合于数据规模n较小的情景。

**【算法思路】**：序列左侧视为已排序序列，序列右侧视为未排序序列，每次从未排序序列中移除一个元素到已排序序列，已排序序列每次加入新的元素就要重新交换一遍来确定顺序。



**【编码思路】：**双层循环，外层循环负责从左至右扩展已排序序列，内层循环负责从右至左在已排序序列中交换完成排序。



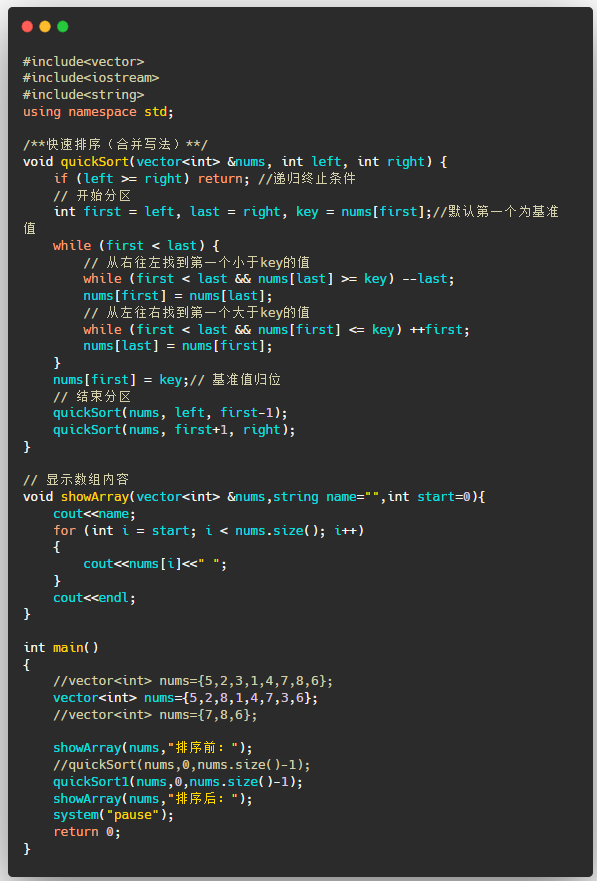
### 4、快速排序

快速排序（quick-sort）是性能鲁棒性最好的排序算法之一，任何情况下耗费时间都为，而且不占用额外内存，适合于数据规模n较小的情景。

**【算法思路】**：将一个数组分成两个子数组，将两部分独立地排序。切分地位置取决于数组的内容。先随意取第一个元素作为切分元素，从数组左端开始向右扫描直到找到一个大于等于它的元素，在从数组右端开始向左扫描找到一个小于等于它的元素，交换这两个元素的位置，直到两个指针相遇，最后和切分元素交换位置即可。

**【编码思路】：**双层循环，外层循环负责划分已排序序列和未排序序列，内存循环负责从未排序序列中查找最值，找到后交换两数即可。

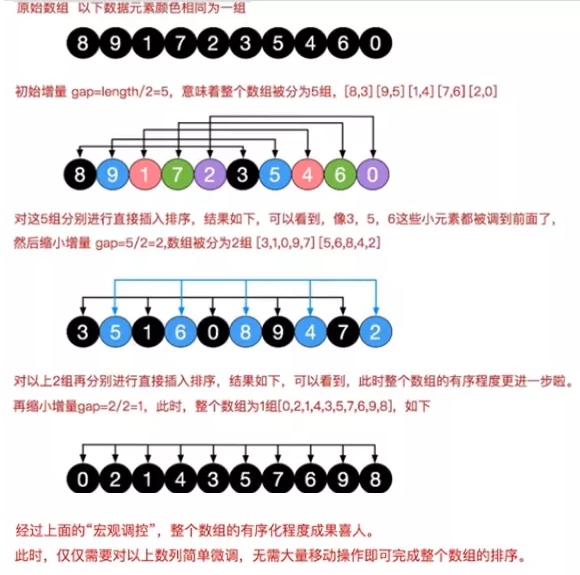




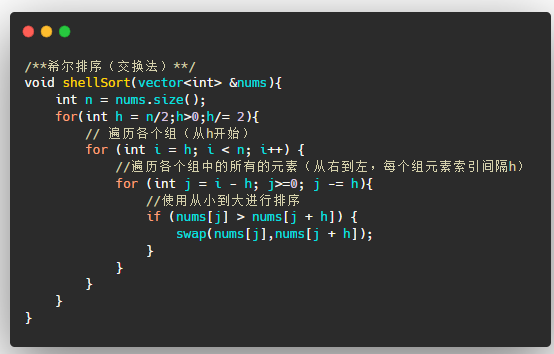
### 5、希尔排序

希尔排序（shell-sort）是1959年希尔（Donald Shell）提出的一种排序算法，可以看做是插入算法的改进版本，也成为了**缩小增量排序**。假设从小到大排序，简单插入算法插入一个较小的数时需要后移的次数明显增多，

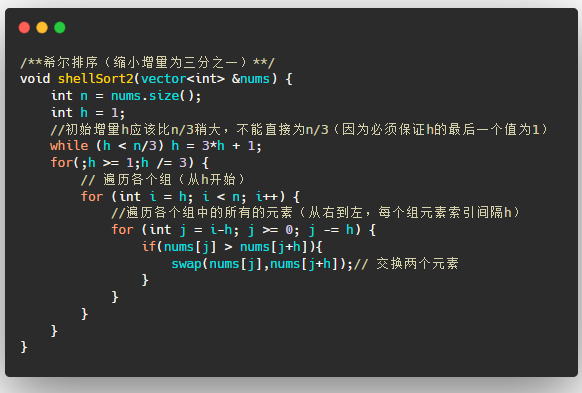
**【算法思路】**：整个数组按照增量h分成h个分组，各个分组独立进行插入排序；然后不断缩小增量h，当h等于1的时候整个数组就完成了排序。。增量h的取值也叫做Shell增量序列，如最常见的折半增量序列{n,n/2,...,1}。



**【编码思路】：三**层循环，第一层循环负责增量h的取值，第二层循环负责切换不同h分组最后一个元素索引的取值，第三层负责遍历当前h分组中所有的元素，把顺序不对的相邻元素交换过来。以折半增量序列为代表的交换式希尔算法代码实现如下：



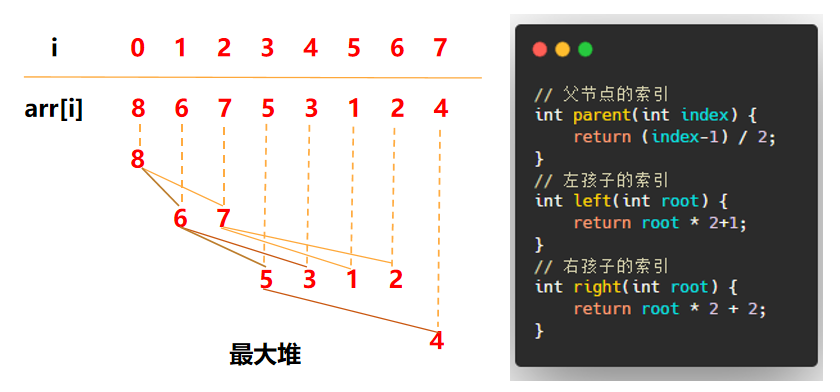
有时候我们简单换一下增量h的取值序列，就可以显著提升希尔排序的算法效率，如我们将h的取值切换为三分之一来看一下：



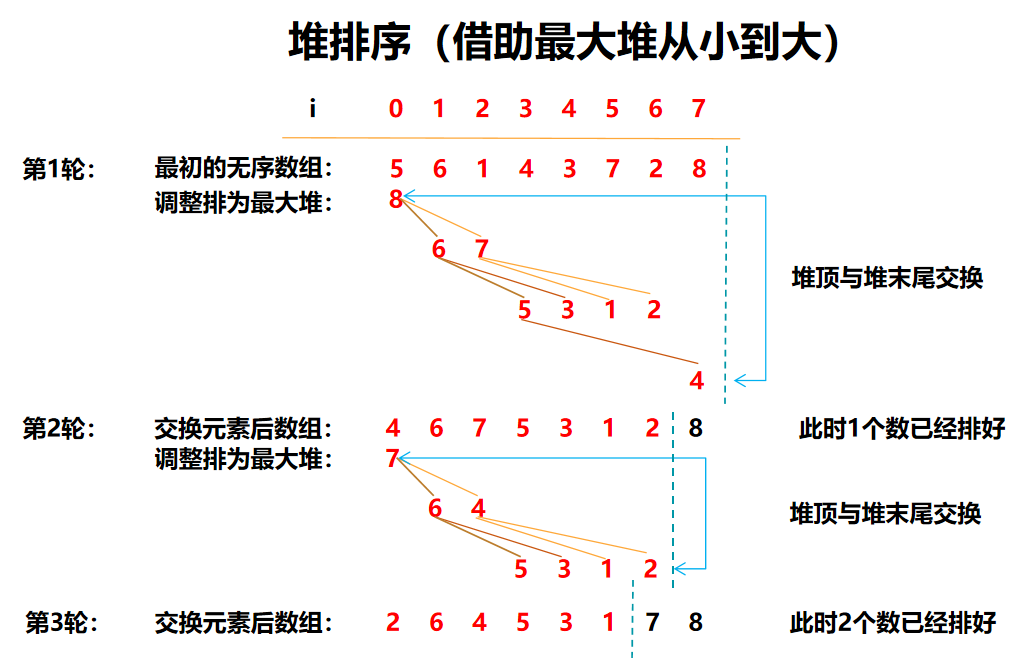
该h的取值比折半Shell增量的效率要高（力扣912题该h取值的希尔算法可以通过测试。）

### 6、堆排序

堆排序（heap-sort）是利用一种叫做堆（heap）的数据结构所设计的排序算法。堆本质上就是用数组实现的二叉树，所以也叫做二叉堆。堆有一个性质，叫做堆有序。根据这种堆有序的性质，堆分为2种，最大堆（大顶堆）和最小堆（小顶堆）。在最大堆中，父节点的值都比每一个子节点的值要大；在最小堆中，父节点的值都比每一个子节点的值要小。（注意和二叉搜索树的区别，二叉搜索树是左子树的值比父节点的值小，右子树的值比父节点的值大）。下图是一个最大堆的内部数据的逻辑结构，根据这种特点，给定任意一个数所在的索引index，我们就可以知道它的父节点索引或者左右子节点的索引。

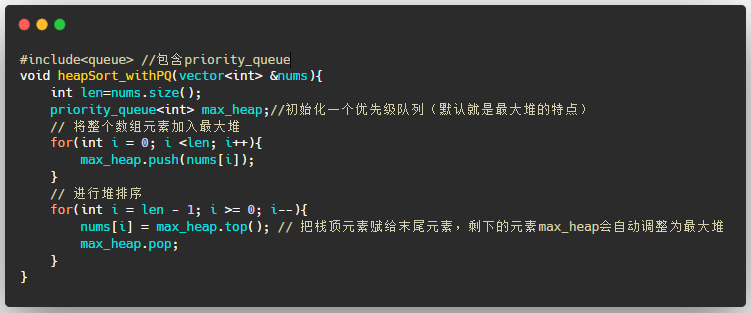


**【算法思路】**：假设从小到大排序，堆排序的基本思路是先将整个无序数组构造成一个最大堆，那么最大堆的根节点一定就是全局最大的数，将其取出来放入已排序序列；删除掉根节点的最大堆在重新构建一个最大堆，从而又找到了一个局部最大值，接着将其放入已排序序列，当无序序列中构建最大堆的数只剩下一个时，排序也就完成。过程如下：



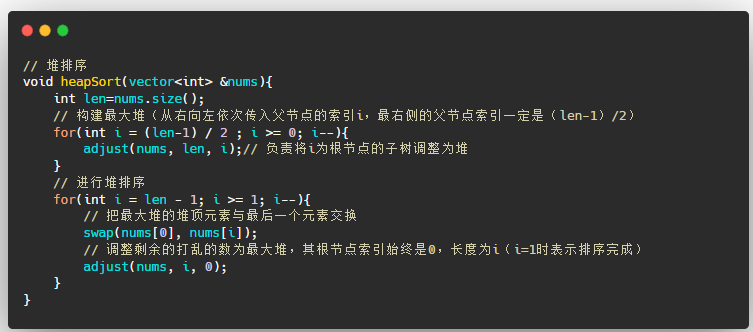
**【编码思路】：**在上述思路分析中，可知将一个无序序列转换为堆是最关键的步骤。最大堆是一个父节点都大于子节点的二叉树，在编码上可以采取递归的方式来实现。我们首先借助C++的优先级队列来实现排序。

在大多数编程语言中，优先级队列（Priority Queue）就可以看做是一个堆结构（其底层可能就是借助堆实现的），优先级队列的特点就是其队首元素始终是整个队列中最大或者最小的数，这和最大堆或最小堆的特点一样。所以，我们给出如下代码即可完成排序：

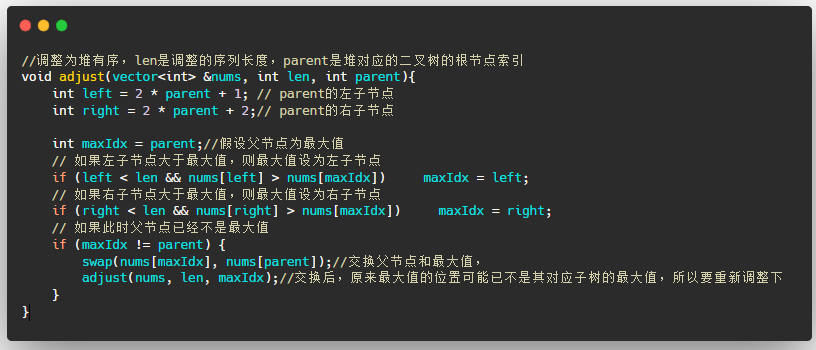


上述编码其实就是堆排序的思路，借助一个最大堆或最小堆，把无序序列始终调整为堆有序，其堆顶元素一定是全局最大或最小。现在假设面试官不让我们继续使用Priority Queue，让我们手写堆的调整结构，不要怕，难度也不会增加很多。

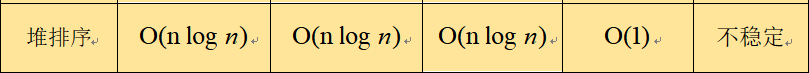
假设我们有一个函数adjust()，其功能就是负责将i为根节点的子树调整为堆有序，即满足最大堆或最小堆，那么我们借助这个adjust就可以写出如下堆排序的代码：



现在我们的任务就是写出adjust函数，其也利用了二叉树递归的思路，只要保证每个小子树中父节点比孩子节点都大或者都小即可，如下：



**【**性能分析**】**：



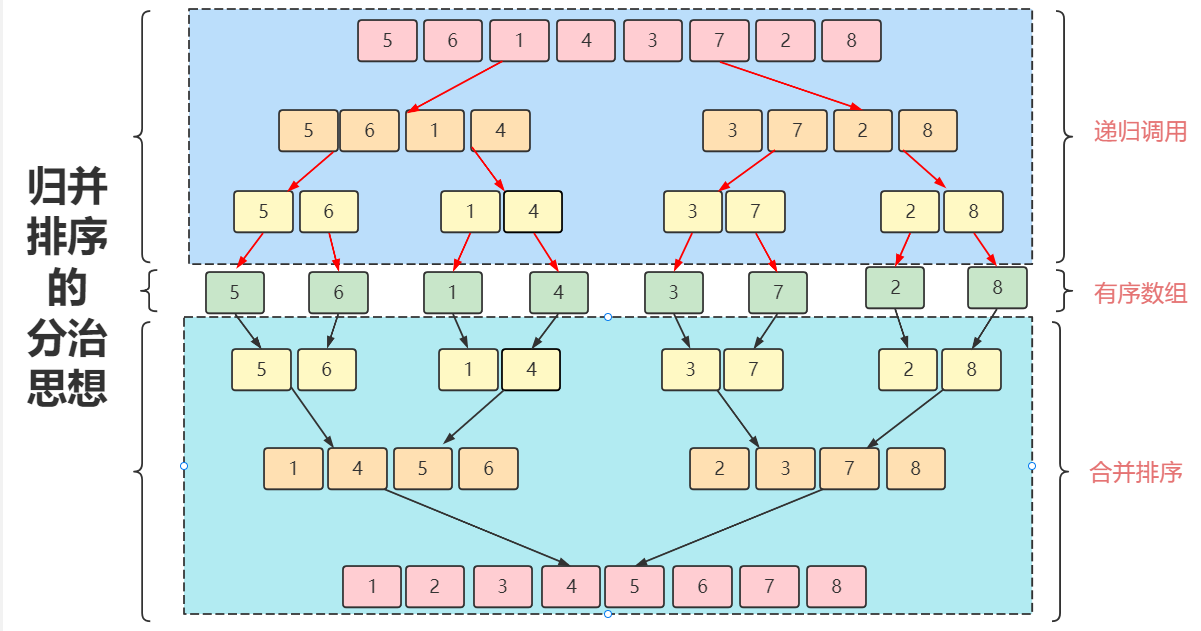
此时我们已经完成了整个堆排序的任务，最重要的是adjust的实现，读者要多默写几遍。学习堆排序后，我们实际也同时掌握了堆、优先级队列的核心知识，一石三鸟。

## 二、外排序

### 1、归并排序

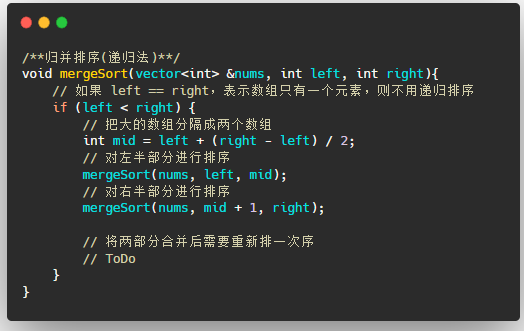
归并排序（merge-sort）采用了分治思想（即将一个大问题分解成若干个小问题来解决），分治思想在编码上一般容易使用递归法。

【算法思路】：归并排序认为如果要给一个数组排序，可以把整个大数组对等分成2个小数组，2个小数组继续分割，直到分出的每个小数组大小都是1，只有1个元素的数组一定是有序数组了，然后再把2个有序数组合并成新的更大数组，合并的时候需要重新排下序，合并后的数组就又有序了。

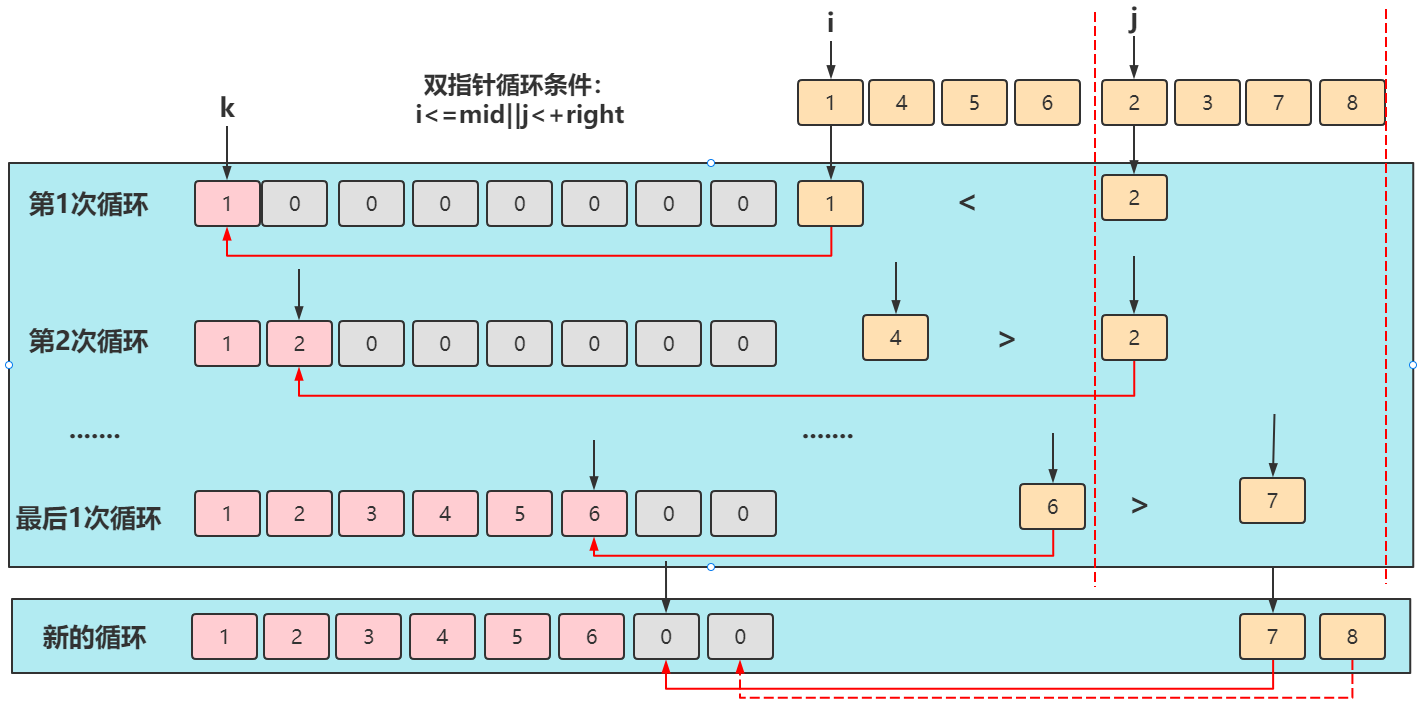


给2个小的有序数组（长度=N/2）重新排序通常比直接排序一个大的无序数组（长度=N）要快，这也是归并排序要先划分后合并的原因，先划分直至每个数组都是有序数组，然后合并排序成一个新的有序数组。

【编码思路】：在上述的思路分析中，首先递归调用来不断分割数组的编码很简单了，如下：



接下来就是考虑如何合并2个有序数组了，归并排序使用了申请新数组的双指针法来合并排序。首先直接申请一个足够容纳2个有序数组的临时数组，然后i指针指向数组的左半部分nums[left,mid]，j指针指向数组的右半部分nums[mid+1,right]。接下来依次比较2个数组中的每个元素，哪个小数组的数更小就放入临时数组中，如下：



最后双指针循环结束后，原数组中还可能会有剩余的没有复制，因为双指针的循环条件是有一个指针达到其终点即可，所以循环结束时至少有一个数组的元素全部被复制完（图中是i指针指向的左数组），另一个数组则需要一个新的循环来复制，最后直接上代码。



【性能分析】：先给出归并排序的性能分析结论：



归并排序分为两部分：递归分组和合并排序。通过递归方式把整个大数组一层一层的折半分组，则由完全二叉树的深度可知，整个排序过程需要进行logn（向上取整）次分组,最后合并排序的时候需要将每个有序小数组的数都扫描一次，整个排序过程一共扫描了整个大数组的数，所以每轮合并排序的时间复杂度为O(n)，则总的时间复杂度为 O(nlogn)。

归并排序的执行效率与要排序的原始数组的有序程度无关，所以在最好，最坏，平均情况下时间复杂度均为 O(nlogn) 。

归并排序所创建的临时数组都会在方法结束时释放，单次归并排序的最大空间是n,所以归并排序的空间复杂度为O(n).

归并排序的稳定性，取决于合并排序的时候如何处理相等的两个数。在归并排序的代码中，设置了nums[i]<=nums[j]，当两个元素相同时，先放nums[i] 的值到大数组中，所以两个相同元素的相对位置没有发生改变，此处归并排序是稳定的排序算法。

虽然归并排序时间复杂度很稳定，但是他的应用范围却不如快速排序广泛，这是因为归并排序不是原地排序算法，空间复杂度不为O(1)。

## 附加资料

本章节的主要参考资料：

[（四）排序【C++刷题】-Caoer199-博客园(cnblogs.com)](https://www.cnblogs.com/caoer/p/15722363.html)

[十大排序算法(背诵版+动图) - 力扣（LeetCode）](https://leetcode-cn.com/circle/article/0akb5U/)

借助**快速排序**的算法思想可以快速解决如下LeetCode题目：

[912. 排序数组 - 力扣（LeetCode）](https://leetcode-cn.com/problems/sort-an-array/)

[75. 颜色分类 - 力扣（LeetCode）](https://leetcode-cn.com/problems/sort-colors/submissions/)

借助**快速排序**的算法思想可以快速解决如下LeetCode题目：

[215. 数组中的第K个最大元素 - 力扣（LeetCode）](https://leetcode-cn.com/problems/kth-largest-element-in-an-array/submissions/)

[剑指 Offer II 076.数组中的第k大的数字 - 力扣（LeetCode）](https://leetcode-cn.com/problems/xx4gT2/)



# 第9章 经典算法问题

## 一、龟兔赛跑算法

## 二、跑道问题

# 附录

# 课后习题

## 链表部分

1. 请简述下顺序存储结构和链式存储结构的特点/优缺点。
2. 请手动调用C++的STL中list的初始化、遍历、插入、删除、获取、反向、排序的方法。
3. 请手动实现下单链表的以下功能的代码：

