目录

[第一章 算法排序 3](#_Toc17068)

[1.1算法基础 3](#_Toc187)

[1.1.1 时间复杂度 3](#_Toc8920)

[1.1.2 空间复杂度 5](#_Toc4311)

[1.1.3 稳定性 5](#_Toc20038)

[1.2 桶排序 5](#_Toc12898)

[1.2.2 什么是桶排序 6](#_Toc22723)

[1.2.3 桶排序的实现 6](#_Toc13746)

[1.2.4 桶排序的性能及特点 6](#_Toc5660)

[1.3 咕嘟咕嘟的冒泡排序 7](#_Toc6380)

[1.3.1 什么是冒泡排序 7](#_Toc18247)

[1.3.2 冒泡排序的原理 7](#_Toc28417)

[1.3.3冒泡排序的实现 8](#_Toc6623)

[1.3.4 冒泡排序的特点及性能 8](#_Toc17362)

[1.4最常用的快速排序 8](#_Toc21577)

[1.4.1 什么是快速排序 9](#_Toc3620)

[1.4.2 快速排序的原理 9](#_Toc7233)

[1.4.3 快速排序的实现 9](#_Toc1678)

[1.4.4 快速排序的特点及性能 9](#_Toc29351)

[1.5 简单的插入排序 10](#_Toc12941)

[1.5.1 什么是插入排序 10](#_Toc5816)

[1.5.2 插入排序原理 10](#_Toc11603)

[1.5.3 插入排序的实现 11](#_Toc21535)

[1.5.4 插入排序的特点性能 11](#_Toc17433)

[第二章 搜索 12](#_Toc6783)

[2.1 顺序查找 12](#_Toc27131)

[2.1.1 顺序查找的原理于实现 12](#_Toc29292)

[2.1.2 顺序查找的特点及性能分析 12](#_Toc26528)

[2.2 二分查找 13](#_Toc17119)

[2.2.1 二分查找的原理及实现 13](#_Toc3393)

[2.2.2 二分查找的特点及性能分析 14](#_Toc13233)

[第三章 树 14](#_Toc8651)

[3.1 二叉树 14](#_Toc7340)

[3.1.2 什么时二叉树 14](#_Toc24624)

[3.2.2 特殊的二叉树 15](#_Toc11803)

[3.2.3 二叉树的实现 15](#_Toc7205)

[3.2.4 二叉树的遍历 15](#_Toc10002)

[3.2.5 完全二叉树 17](#_Toc12763)

[3.3 二叉树的查找算法 17](#_Toc20162)

[3.3.1 二叉树查找 17](#_Toc16501)

[3.4 哈夫曼树 18](#_Toc20882)

[3.4.1 专业术语 18](#_Toc27983)

[3.4.2什么是哈夫曼树 19](#_Toc26577)

[3.5 堆 19](#_Toc7756)

[3.5.1 什么是堆 19](#_Toc6983)

[3.5.2 堆排序 20](#_Toc19398)

[3.6 红黑树 20](#_Toc27111)

[3.6.1 红黑树的自我修正 21](#_Toc20186)

[3.6.3、左旋和右旋代码 22](#_Toc14190)

[3.6.4 插入操作 24](#_Toc30493)

[3.6.5 删除操作 25](#_Toc19645)

[3.6.6 红黑树的效率 26](#_Toc19205)

# 第一章 算法排序

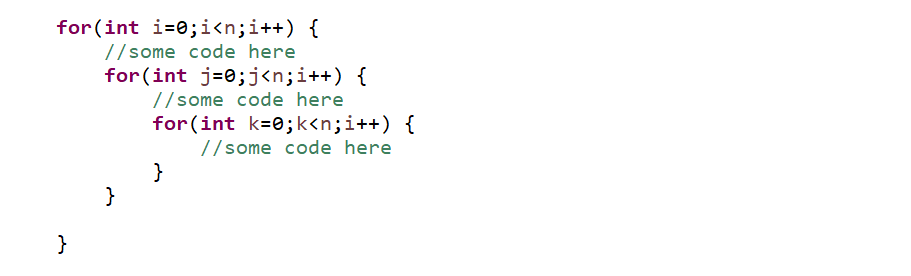
## 1.1算法基础

[算法复杂度](https://baike.baidu.com/item/%E7%AE%97%E6%B3%95%E5%A4%8D%E6%9D%82%E5%BA%A6" \t "https://baike.baidu.com/item/%E6%97%B6%E9%97%B4%E5%A4%8D%E6%9D%82%E6%80%A7/_blank)分为[时间复杂度](https://baike.baidu.com/item/%E6%97%B6%E9%97%B4%E5%A4%8D%E6%9D%82%E5%BA%A6" \t "https://baike.baidu.com/item/%E6%97%B6%E9%97%B4%E5%A4%8D%E6%9D%82%E6%80%A7/_blank)和[空间复杂度](https://baike.baidu.com/item/%E7%A9%BA%E9%97%B4%E5%A4%8D%E6%9D%82%E5%BA%A6" \t "https://baike.baidu.com/item/%E6%97%B6%E9%97%B4%E5%A4%8D%E6%9D%82%E6%80%A7/_blank)。其作用： [时间复杂度](https://baike.baidu.com/item/%E6%97%B6%E9%97%B4%E5%A4%8D%E6%9D%82%E5%BA%A6" \t "https://baike.baidu.com/item/%E6%97%B6%E9%97%B4%E5%A4%8D%E6%9D%82%E6%80%A7/_blank)是指执行算法所需要的计算工作量；而[空间复杂度](https://baike.baidu.com/item/%E7%A9%BA%E9%97%B4%E5%A4%8D%E6%9D%82%E5%BA%A6" \t "https://baike.baidu.com/item/%E6%97%B6%E9%97%B4%E5%A4%8D%E6%9D%82%E6%80%A7/_blank)是指执行这个算法所需要的[内存](https://baike.baidu.com/item/%E5%86%85%E5%AD%98" \t "https://baike.baidu.com/item/%E6%97%B6%E9%97%B4%E5%A4%8D%E6%9D%82%E6%80%A7/_blank)空间。（算法的复杂性体运行该算法时的计算机所需资源的多少上，[计算机](https://baike.baidu.com/item/%E8%AE%A1%E7%AE%97%E6%9C%BA" \t "https://baike.baidu.com/item/%E6%97%B6%E9%97%B4%E5%A4%8D%E6%9D%82%E6%80%A7/_blank)资源最重要的是时间和空间（即[寄存器](https://baike.baidu.com/item/%E5%AF%84%E5%AD%98%E5%99%A8" \t "https://baike.baidu.com/item/%E6%97%B6%E9%97%B4%E5%A4%8D%E6%9D%82%E6%80%A7/_blank)）资源，因此复杂度分为时间和空间复杂度。）

### 时间复杂度

一般的时间复杂度按照性能从差到好有这么几种：O (n3) 、O (n2) 、O(nlogn) 、O (n) 、O(logn) 、O(1)

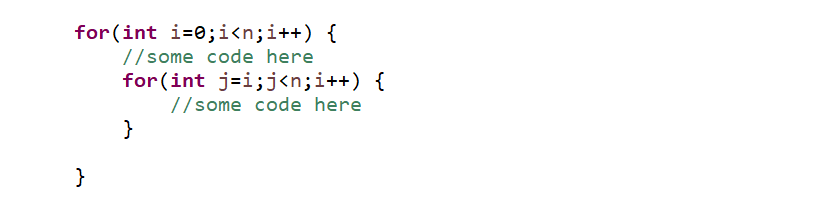
案例1：时间复杂度O (n3)



这段代码是个三重嵌套循环代码，且没重复循环都执行力完整的n编，n一般值算法的规模，很容易判断这段代码的时间复杂度为O (n3)

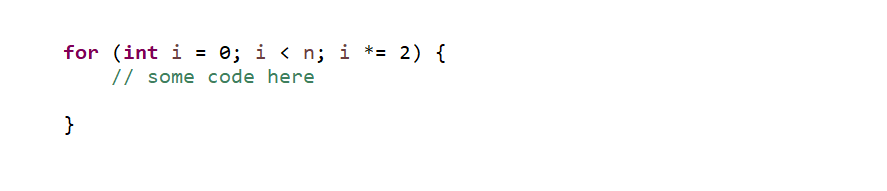
如果是两重嵌套循环代码，那么时间复杂度为O (n2)；如果为一重循环那么时间复杂度为O (n)。

案例2：时间复杂度O(nlogn)



在内层循环嵌套中的起始量是i，而随着每次外层嵌套循环i的增加，j的一层循环执行的次数将会越少。这种情况的时间复杂度为O(nlogn)。

案例3：时间复杂度O(logn)



这种情况称为对数阶，性能要优于O(n)。

案例4：时间复杂度O(1)



实际上，一个算法的执行时间是不可能通过计算得出的，必须到机器上真正执行才能知道，而且每次的运行时间不一样。但是我们没有必须讲每个算法都到机器上运行和测试，但是对于很多算法，我们通过简单的分析就能知道其性能的好坏，而且没有必要详细的写出来，所以时间复杂度的计算还是非常有用的。

### 空间复杂度

空间复杂度(Space Complexity)是对一个算法在运行过程中临时占用存储空间大小的量度，记做S(n)=O(f(n))。比如直接[插入排序](https://baike.baidu.com/item/%E6%8F%92%E5%85%A5%E6%8E%92%E5%BA%8F" \t "https://baike.baidu.com/item/%E7%A9%BA%E9%97%B4%E5%A4%8D%E6%9D%82%E5%BA%A6/_blank)的[时间复杂度](https://baike.baidu.com/item/%E6%97%B6%E9%97%B4%E5%A4%8D%E6%9D%82%E5%BA%A6/1894057" \t "https://baike.baidu.com/item/%E7%A9%BA%E9%97%B4%E5%A4%8D%E6%9D%82%E5%BA%A6/_blank)是O(n^2),空间复杂度是O(1) 。而一般的[递归](https://baike.baidu.com/item/%E9%80%92%E5%BD%92" \t "https://baike.baidu.com/item/%E7%A9%BA%E9%97%B4%E5%A4%8D%E6%9D%82%E5%BA%A6/_blank)算法就要有O(n)的空间复杂度了，因为每次递归都要存储返回信息。一个算法的优劣主要从算法的执行时间和所需要占用的存储空间两个方面[衡量](https://baike.baidu.com/item/%E8%A1%A1%E9%87%8F/483075" \t "https://baike.baidu.com/item/%E7%A9%BA%E9%97%B4%E5%A4%8D%E6%9D%82%E5%BA%A6/_blank)。

### 稳定性

什么是稳定性？在排序算法中，可能在一个列表中存在多个相等的元素，而经过排序只会，这些元素的相对次序保持不变，这时我们称这个算法是稳定的，若经过排序只会次序变了，那么就是不稳定的。

稳定性有什么用呢？如果算法是稳定的，那么第1个元素排序的结果可以被第2个相同值额元素排序所用，也就是说算法是稳定的，那么可以避免多余的比较。

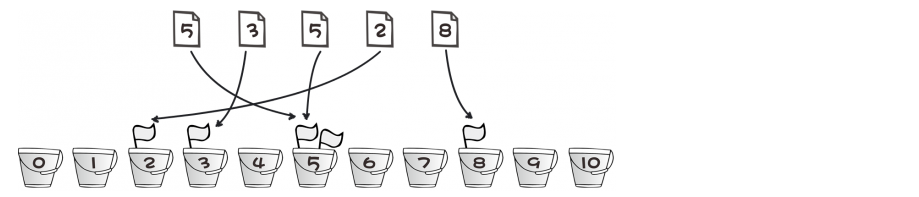
在某些情况下，若是值一样的元素也要保持于原有的相对次序不变，那么这时必须用一个稳定的算法。

## 桶排序

在我们生活的这个世界中到处都是被排序过的。站队的时候会按照身高排序，考试的名次需要按照分数排序，网上购物的时候会按照价格排序，电子邮箱中的邮件按照时间排序……总之很多东西都需要排序，可以说排序是无处不在。现在我们举个具体的例子来介绍一下排序算法。

案例1：某学期期末考试，老师把大家的分数排序，比如有5个学生分别考5、3、5、2、8（满分10分），从大到小的排序应该是8、5、5、3、2。

解题思路：方案有很多，这里用桶排序解决，找到11个桶，分别编号为0-10，对应0-10分，接着把这些分数按照桶的编号放入桶内，接着从最大的编号的桶到最小编号的桶依次输出每个桶中的分数，分别是8、5、5、3、2，这就是桶排序的思想。



### 1.2.2 什么是桶排序

桶排序也叫箱排序,是一个排序算法。其中的思想是我们首先需要知道所有待排序的范围，然后需要有在这个范围内的同样数量的桶，接着把元素放到对应的桶中，最后按顺序输出。上面是一个简单的案例，如果考试分数的范围是0-100万呢？

实际上一个桶中并不是宗方同一个元素在很多时候一个同理可能放多个元素，这就跟散列表有点相似，桶排序和散列表的有一样的原理。

除了对一个桶内的元素做链表存储，我们也有可能对每个同种的元素继续使用其他的排序算法进行排序，所有更多时候，桶排序会结合其他算法排序一起使用。

### 1.2.3 桶排序的实现

使用数组对桶排序进行处理，比如有11个桶，我们只需要声明一个长度为11的数组，然后把每一个元素往桶中放是，就把数组指定位置的值家1，最终按倒叙输出数组的下表，数组每个位置的值为几，就输出几次下标，这样就可以实现桶排序了。

### 1.2.4 桶排序的性能及特点

桶排序实际上只需要遍历一遍所有的待排序的元素，然后一次放入指定的位置。如果加上输出排序的时间，那么需要遍历所有的桶，时间复杂度就是O（n+m），其中n为元素的个数，m为桶的个数，这是相当快速的排序算法，但是对于空间的消耗太大了。比如对1、10、100、1000、10000这四个元素排序，那么我们需要的长度为10001的数组用来排序，我们发现，当元素的跨段范围越大是，空间的浪费就越大，即使只有几个元素，所以桶排序的空间复杂度是O(m),其中m为桶的个数，待排序元素分布越均匀，这个空间的利用率越好。

通过上面的分析，我们可以知道桶排序的特点：速度快，简单，但是空间利用率低，如果数据跨度大，则空间可能无法承受，或者说这些元素并不适合使用桶排序。

## 咕嘟咕嘟的冒泡排序

虽然桶排序又简单又快速，但是空间利用率低，即算法的性能很受桶的分数影响，无论是时间复杂度还是空间复杂度，都需要考虑桶的个数,所以我们继续接触另一个排序算法-冒泡排序。

### 什么是冒泡排序

重复的走访要排序的数列，一次比较两个数据元素，如果顺序不对则进行交换，并一直重复这样的走访操作，知道没有要交换的元素为止。

### 冒泡排序的原理

首先肯定有一个数组，里面存放着待排序的元素列表，我们如果需要把比较大的元素排在前卖，把小的元素排在后面，那么需要从尾到头开始下名的比较操作。

1. 从尾部开始比较相邻的两个元素，如果尾部的元素比前卖的大，就交换两个元素的位置。
2. 往前对两个相邻元素都做这样的比较，交换操作，这样到数组头部时，第一个元素会成为最大的元素
3. 重新从尾部开始第1，2步的操作，除了在这之前，头部已经拍好的元素。
4. 继续对越来越少的数据进行比较，交换操作，直到没有可比较的数据为止，排序完成

假如我们要把12，44，66，33，22这5个数从大到小进行排序，那么数越大，越需要把它放到前面。冒泡排序的思想就是每次遍历一遍未排序的数列只会，将一个数据元素浮上去（也就拍好了一个数据）。

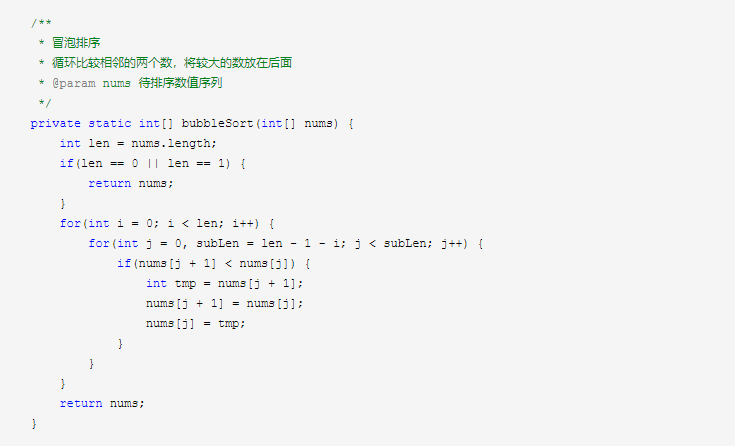
如图：。。。

经过第一躺排序，我们已经找到了最大的元素，接下来的第2汤排序就只对剩下的4个元素排序。第2躺排序的过程

如图：。。。

### 1.3.3冒泡排序的实现

代码：



### 1.3.4 冒泡排序的特点及性能

通过冒泡排序的思想，我们发现冒泡排序在算法的每轮排序中会使一个元素排到一端，也就是最终需要n-1轮这样的排序（n为待排序的数列的长度），而在每轮排序中都需要对相邻的两个元素进行比较，在最坏的情况下，每次比较只会都需要交换位置，所以这里的时间复杂度是o(n2)，其实冒泡排序在最好的情况下，时间复杂度可以达到o(n)，这当然是在待排序有序的情况下，在待排序的数列本身就是我们想要的排序结果时，时间复杂度为o(n)，因为只需要一轮排序，且不用交换，但实际上这种情况很少，所以冒泡排序的平均时间复杂度为o(n2)。

对于空间复杂度来说，冒泡排序用到的额外的村粗空间只有一个，那就是用于交换位置的临时变量，其他所有操作都是在原有待排序列上处理的，所以空间复杂度为o(1)。

冒泡排序时稳定的，因为在比较过程中，只有后一个元素比前面的元素大时，才会对它们交换位置并向上冒出，对于同样大小的元素，是不需要交换位置的，所以对于同样大小的元素来说，相对位置是不会改变的。

冒泡排序算法的时间复杂度其实比较高。性能不高，所以在实际工作中并不会用到，但是在面试时还是有可能会用到的。

## 1.4最常用的快速排序

冒泡排序时间复杂度是O(n2)，如果计算机每秒运算速度是10亿次，排序1亿个数字，那么桶排序只需要0.1秒，冒泡排序需要1千万秒（115天），有没有一种排序既节省时间有节省空间呢？

### 1.4.1 什么是快速排序

快速排序其实就是对冒泡排序的一种改进，它的基本思想是：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据比另一部分的所有数据要小，再按这种方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，使整个数据变成有序的序列。

### 1.4.2 快速排序的原理

在待排序的数列中，首先找到一个书句子作为基准数，为了方便，一般选择第一个数据为基准数，接下来把这个待排序的数列中小于基准数的元素移动到待排序的数列的左边，把大于基准数的元素移动到待排序的数列的右边，这时左右两个分区的元素就相对有序了，接着把两个分区的元素分别按照上面两种方法继续对每个分区找出基准数，然后移动，直到各个分区只有一个数时为止。

### 1.4.3 快速排序的实现

快速排序有个简单的思想就是递归，对于每一躺排序都是一样的思想，只不过需要进行排序的数组范围越来越小了，下面使用递归实现这种排序。

代码：。。。

### 1.4.4 快速排序的特点及性能

快速排序就是在冒泡排序的基础上改进而来的，冒泡排序每次只能交换相邻的两个元素，而快速排序时跳跃式的交换，交换的距离很大，因此总的比较和交换次数少了很多，快速也快了不少。

但是快速排序在最坏的情况下，时间复杂度和冒泡排序一样，是O(n2),实际上每次比较都需要交换，但是这种情况并不常见。最好的情况是每次比较都不需要交换，那么数列的平均时间复杂度是O(nlogn)。

快速排序只是使用数组原本的空间进行排序，所以占用的空间应该是常量级别的，但是由于每次划分只会是递归调用，所以递归调用在运行的过程中会消耗一定的空间，在一般情况下空间复杂度是O(logn)，在最差的情况下，若每次只完成一个元素，那么空间复杂度为O(n)。

快速排序是一个不稳定的算法，在经过排序只会，可能会对相同值得元素得对应位置造成改变。

快速排序基本上被认为是相同数量级得所有排序算法中，平均性能最好的。

## 1.5 简单的插入排序

### 1.5.1 什么是插入排序

插入排序分为两种，一直是直接插入排序，一种是二分插入排序。直接插入排序就是往数列中插入数据元素，一般认为是往一个已经排好序的待排序的数列中插入一个数，使得插入这个数只会，数列仍然有序。

二分插入排序也是用了分治法的思想去排序，实际上就是使用了二分查找来找到这个插入的位置，剩余的插入额思想其实和直接插入排序一样。

### 1.5.2 插入排序原理

插入排序实际上把待排序的数列分为两个部分，一部分以排好序，另一部分待排序。

案例1：假设待排序的序列为63，88，34，99，38，55，9。

如图：。。。

这时全部数列为待排序部分，我们开始一点点的进行插入排序。

首先取出63，这是第一个元素，不需要排序，这时已排好序的部分有一个元素了，就是63，而剩下的元素为待排序部分。

接着取出88，与前面的元素比，发现比63大，符合把数列小到大的排序规则，无需交换，这时排好序的部分又多了一个元素，而待排序的部分相应的减少了一个元素。

接着取34，比前面的元素88小，把34拿出来，让它在外面等一下，把88向后移动一位。此时，数组的情况如图：

接下来继续向前比较，直到比较到第一个元素为止，发现34仍然比63小，继续把63想后移动，这时数组状态如图：

现在发现已经比较到第一个元素了，第一个位置的63需要移动，所以第一个位置空出来了，把34放到第一个位置上，这时数组的状态为34，63，88，99，38，55，9。

现在已排好序的部分又34，63，88，剩下的部分为待排序部分。

接着取出99，与前一个元素比较，发现比88大，由于前面的已经排好序，所以88是前面排好序部分中的最大元素，99比88大，肯定也比前面的元素都大，不用继续比较了，可以直接把99加入到前面的排好序的部分。

接着取出38，与前一个元素99比，比99小，于是把38拿出来，继续将99向后移动一位，这时待排序的数列状态如图：

接着继续用38与88比较，发现比88小，88继续后移动一位，继续与63比，发现比63小，63也后移一位，这时数组的状态如图：

接着继续用38与34比，发现比34大，把38放到那个空位上此时数组的状态为34，38，63，88，99，后面就是处理55和9了，处理方案同上。

### 1.5.3 插入排序的实现

代码：

### 1.5.4 插入排序的特点性能

插入排序的时间复杂度为o(n2),我们发现这个实现实际是个双重嵌套循环，外层执行n遍，内层最坏的情况下执行n遍，而且除了比较操作外还有移动操作。最好的情况是数列近似有序，这时一部分内存循环只需要比较及移动较少的次数即可完成排序，如果序列本身已经排好序，那么插入排序也可以达到线性时间复杂度及O(n)。

插入排序的空间复杂度是O(1),是常量级的，由于在采用插入排序时，我们只需要使用一个额外的空间来存储这个”拿出来的元素”，所以插入排序只需要额外的一个空间取做排序，这是常量级的空间消耗。

插入排序是稳定的，由于数组内部自己排序，把后面的部分按前面顺序一定的的比较，移动，可以保持相对顺序不变，所以插入排序是最稳定的排序算法。

# 搜索

## 2.1 顺序查找

一提到查找，比如从一个数列中查找第一个为K的数，最先想到的是一个一个的去找，从数列的第一个数开始对比，直到找到K的数为止。

顺序查找的定义：在一个已知的队列中找到与给定的关键字相同的数的具体位置。原理是让关键字与队列中的数从开始一个一个的往后逐个比较，直到找到与给定的关键字相同的数。

当然，顺序查找不仅仅限于数字，字符的查找，也使用于前缀，对象信息的关键信息的匹配。

### 2.1.1 顺序查找的原理于实现

顺序查找对数列是否有序没有要求，一般是从数列的一段开始查找，直到找到为止，找到则返回元素，没有找到则返回一个无意义的结果。

代码。。。。

### 2.1.2 顺序查找的特点及性能分析

顺序查询就是一个一个的找，当然也可以结合并发来处理，比如说两个人去查找一副牌中的某张牌，可以把牌分成两份，一人找一半，这样速度肯定会快些。

顺序查找的性能：平均时间复杂度为O(n),n是待查找数列的长度，因为顺序查找是从头到尾查找，最好的情况是第一个元素就是我们需要查找的元素，最坏的情况就是需求查找整个数列。

并发情况下的顺序查找，实际上我们在并发查找的元素可能更多，比如两个线程把待查找数列分成两个部分进行查找，如果元素恰巧在第1个线程要查找的列中，那么第2个线程的查找就白做了，但是通常并发还是能够更快的查找。元素如果在后面的线程，则会快很多，尤其是在大数列，更多的线程时。

顺序查找对数列顺序的比较，没有额外的空间，所以空间复杂度是常数O(1)。

## 2.2 二分查找

查找也有特殊情况时，比如数列本身时有序的，这个有序数列是怎么产生的呢？有时它可能本身就是有序的，也有可能是通过之前的排序算法得到的。

不管怎么说，如果我们得到的是一个有序的数列，那么查找某个元素的时候，就可以采用二分查找的方法。

二分查找也叫折半查找，它有两个要求：一个是数列有序，一个是数列使用顺序存储结构

### 2.2.1 二分查找的原理及实现

以升序数列为列，比较一个元素与序列中间位置元素的大小，如果比中间按位置的元素大，则继续在后半部分的数列中进行二分查找；如果比中间位置的元素小，则在数列的前半部分进行比较；如果相等，则找到了元素的位置。每次比较序列的长度都会是之前数列的一半，直到找到相等元素的位置或者最终都没有找到要找的元素。

案例1：数列中有3个数，用二分法进行查找

先与中间的数（刚好是第二个数）比较，如果比第2个数大，则与第2个数后面的数列进行二分查找，这时，右边只剩下一个数了，直接比较是否相等即可。所以3个数的时候最多比较两次。

案例2：数列中有4个数，用二分法进行查找

先与中间的数比较，中间的数一般都是首末相加除以2算出，(1+4)/2等于2，所以先与第2个数比较，如果比第2个数小，则直接与第1个数比较，否则在后面两个数中进行二分查找，这时中间的数是(3+4)/2等于3，也就是后半部分的第一个数比较，相等则找到这个元素，小于则没有这个数（因为左边都比较过了），大于则继续向右二分查找，所以4个数的时候以此类推。

代码实现：

### 2.2.2 二分查找的特点及性能分析

二分查找有个很重要的特点，就是不会查找数列的全部元素，而查找的数据量其实正好符合元素的对数，正常情况下每次查找的元素都是在一半一半的减少。所以二分查找的时间复杂度为O(log2n)。当然最好的情况时只查找一次就能找到，但是最坏的情况和一般情况下比顺序查找要好很多。

# 树

## 3.1 二叉树

### 3.1.2 什么时二叉树

二叉树是有限个节点的集合，这个集合可以是空集，也可以是一个根节点和两个不相交的子二叉树组合的集合，其中一颗树叫做根的左子树，另一根树叫做跟的右子树。

所以，二叉树的定义是个递归的定义，二叉树规定自己可以是空集，而且很明确的区分了一个根节点的两个子树，分别是左子树，右子树。如下图所有的两棵树并不是同样的二叉树，但是i如果认为它们是树，则应该是同样的树，所以二叉树绝对不是一颗特殊的树那么简单。

对于上图中的两颗二叉树，我们根据定义很容易理解，左边一颗二叉树的根节点是A，而左子树的根节点是B，右子树是空集。而右边一颗二叉树的根节点也是A，而右子树的根节点是B，左子树是空集。

### 3.2.2 特殊的二叉树

二叉树其实就是一棵树中的每个节点至多两个子节点，而且子节点的是左右的。

1. 满二叉树

满二叉树就是一颗二叉树的高度为k,且拥有2的k次方个节点的二叉树。其实一颗满二叉树就是一颗二叉树，只是每个节点要么左右子树都有值，要么左右子树都没值。而且每一层的所有节点之间必须要么都有两颗子树，要么都没有子树。

如下图：

1. 完全二叉树

完全二叉树有一些特殊的规则，假设一颗高度为k的二叉树，则完全二叉树需要满足以下特点：

1. 所有叶子节点都处在在k或者k-1层，而且从1到k-1层必须达到最大节点树。
2. 第k层可以不是满的，但是第k层的所有节点必须集中在最左边。

如图：一颗高度为4的完全二叉树

### 3.2.3 二叉树的实现

二叉树的每个节点最多只有两个子节点，二叉树的每个左右节点仍是一颗二叉树，所以我们可以用递归的方式来定义二叉树，二叉树的递归节点的实现代码如下：

### 3.2.4 二叉树的遍历

二叉树的遍历就是按照一定规律来顺序遍历二叉树的各个节点，使得每个节点都会被访问到且仅有一次。

下面介绍三种遍历方式，这三种遍历方式都采用了递归的定义方式。

1. 先根遍历

若二叉树为空树，则推出，否则进行下面的操作。

1)访问根节点

2)遍历左子树

3)再遍历右子树

4)退出

按照先根遍历的方式，如果要遍历下图的二叉树，则访问的顺序是：。。。。

代码实现如下：

。。。

1. 中根遍历

若二叉树为空树，则推出，否则进行下面的操作。

1)中根遍历左子树

2)访问根节点

3)中根遍历右子树

4)退出

按照先根遍历的方式，如果要遍历下图的二叉树，则访问的顺序是：。。。。

代码实现如下：

。。。

1. 后跟遍历

若二叉树为空树，则推出，否则进行下面的操作。

1)后根遍历左子树

2)后根遍历右子树

3)访问根节点

4)退出

按照先根遍历的方式，如果要遍历下图的二叉树，则访问的顺序是：。。。。

代码实现如下：

。。。

### 3.2.5 完全二叉树

如下图，每个元素按照从上到下，从左到右的顺序进行标号，即按照节点顺序依次放入数组里，对于这样的完全二叉树，我们可以很方便的得到每个节点的父节点，左右孩子节点的位置。

我们按照上面的顺序进行编号为i，(1<i<n,其中n为节点个数)的元素有下面的特性。

1. 如果i不为1，则其父节点的编号必然为1/2；如果i为1，那么i节点为二叉树的根节点
2. 如果2i<=1,则i节点的左孩子节点为2i；如果2i>=n,则i节点没有左孩子。
3. 如果2i+1<=n,则右孩子节点为2i+1；如果2i+1>n,则i节点没有右孩子。

这个规律对于完全二叉树来说很实用，但是对于普通的二叉树来说没什么用。

## 3.3 二叉树的查找算法

为什么学习树及二叉树呢？为了更好的查找性能

### 3.3.1 二叉树查找

在之前学习的查找算法中，性能最好的就是二分查找了，但是二分查找要求数列有序，这样每次进行查找时必须对数列排序，然后进行查找。所以二分查找适合数据元素很少增减的情况，如果数列经常变动，则不合适采用二分查找。

什么时二叉树查找呢？

二叉查找树或许是空树，或者满足下面几个特性：

1. 子左数不为空，那么它子左树上的任意节点的值都小于根节点的值。
2. 如果它的右子树不为空，那么他的右子树上的任意节点的值都大于根节点的值。
3. 同样，他的左子树和右子树也都是二叉查找树。

这是一个递归的定义，是不是发现这棵树基本上与递归分不开。下面两棵查找二叉树，如果按照中根遍历，则得到的序列是一样的：。。。。

通过这个例子可以发现，对于同一序列，按照不同的方式去插入，则会得到不同的二叉查找树，而随着元素的增多，则会得到更多的不同的二叉查找树。

## 3.4 哈夫曼树

### 3.4.1 专业术语

1.路径和路径长度

在一棵树中，从一个节点到达孩子或者孙子的通路叫做路径；通路中分支的数目是路径的长度。可以理解为：如果根节点为第1层，那么根节点到达第2层的节点的路径长度都是1，达到第n层的路径长度为n-1。

2.节点的权力及带权路径的长度

如果将树中的节点赋予某种有意义的值，那么这个值就是这个节点的权。其带权路径长度就是从根节点到这个节点的路径的长度乘以权值。

3.树的带权路径长度

树的带权路径长度是树的所有叶子节点的带权路径长度之和，也叫做WPL。

### 3.4.2什么是哈夫曼树

如下图：

这两棵树的带权路径长度分别如下：

左边:1\*2+2\*2+3\*2+4\*2=20

右边:1\*3+2\*3+3\*2+$\*1=19

从上诉结果可以看出，右边的树的带权路径长度最小。

哈夫曼树的定义：

给定n个权值作为一棵树的n个叶子节点，构造一棵二叉树，如果带权路径的长度达到最小，那么这棵树叫做最优二叉树，也就是哈夫曼树。哈夫曼树要求带权路劲的长度最小，所以权值越大的点离根节点越近，这一点非常重要。

## 堆

### 什么是堆

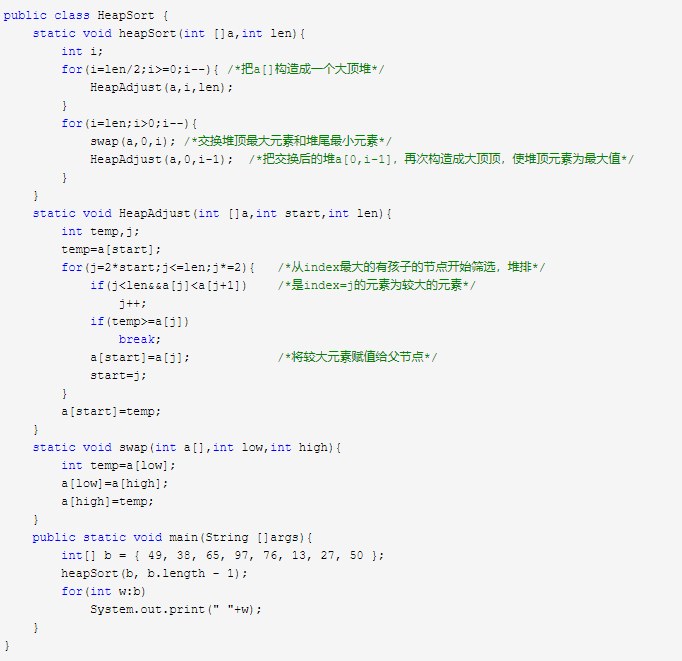
堆是一种特殊的数据结构，是一种特殊形式的完全二叉树。堆分为两种，一种是大顶堆，一种是小顶堆，除父节点外，大顶堆中每个节点的值都不大于其父节点的值。小顶堆中每个节点的值都小于其父节点的值。

由上可知：大顶堆中跟节点元素的值一定是最大的，而且元素的值是从上到下越来越小的；小顶推中根节点元素的值一定是最小的额，而且元素的值是从上到小越来越大的。

假设用数组来定义堆，给堆的每个元素编号为1~n（1<=i<=n,0位置可以用来存储元素的个数），则对于大顶堆来说，第i个元素一定大于等于第2i个元素（如果有），并且大于等于第2i+1个元素（如果有），同样，对于小顶堆来说，第i个元素一定小于等于第2i个元素（如果有），并且小于等于第2i+1个元素（如果有）。

### 堆排序

堆排序就是堆数组进行初始化堆，依次插入元素和删除元素的操作。首先初始化一个堆，然后把无序待排数组中的每个值一次放入堆中，接下来一直执行删除操作，并把被删除的元素放到数组中的最后一个有效元素之后的一个位置，最后的数组就是一个有序的数列了。



## 3.6 红黑树

红黑树本质上是一可二叉查找树。但是它在普通的二叉查找树上有增加了一个标记（颜色），同时有一些特定的规则，它的这些规则保证了红黑树的插入，删除，查找操作的最坏时间复杂度都为O(logn)。

红黑树的规则：

1、每个节点要么是红色，要么是黑色

2、根节点都是黑色节点

3、每个叶节点是黑色节点

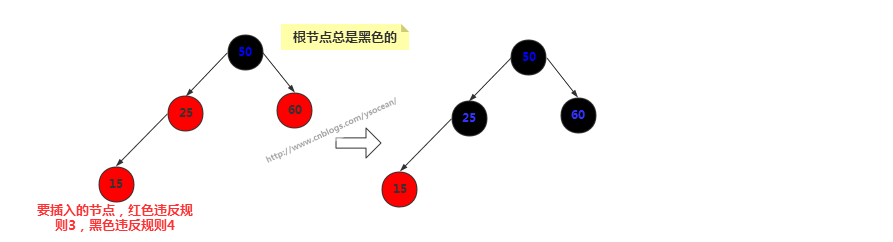
4、每个红色节点的两个子节点都是黑色节点，反之，不做要求，换句话说就是不能有连续两个红色节点

5、从任意节点到所有叶子节点上的黑色节点数量是相同的

### 3.6.1 红黑树的自我修正

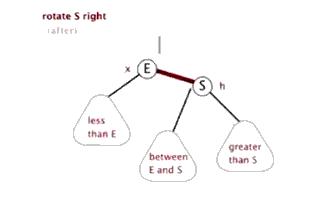
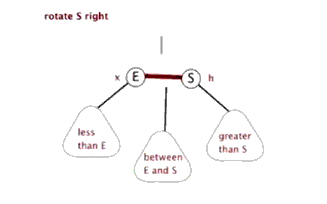
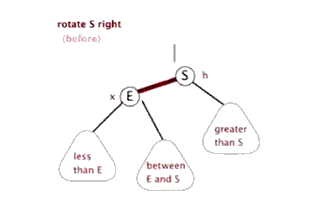
红-黑树主要通过三种方式对平衡进行修正，改变节点颜色、左旋和右旋。

1. 改变节点颜色



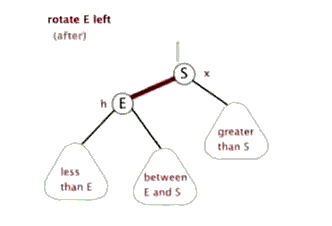
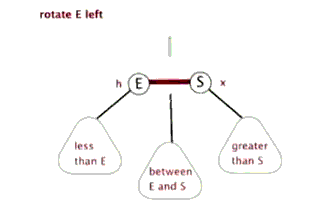
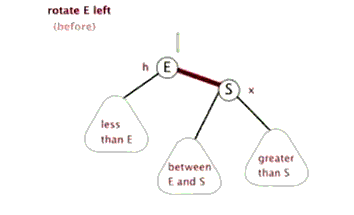
1. 右旋

首先要说明的是节点本身是不会旋转的，旋转改变的是节点之间的关系，选择一个节点作为旋转的顶端，如果做一次右旋，这个顶端节点会向下和向右移动到它右子节点的位置，它的左子节点会上移到它原来的位置。右旋的顶端节点必须要有左子节点。



1. 左旋

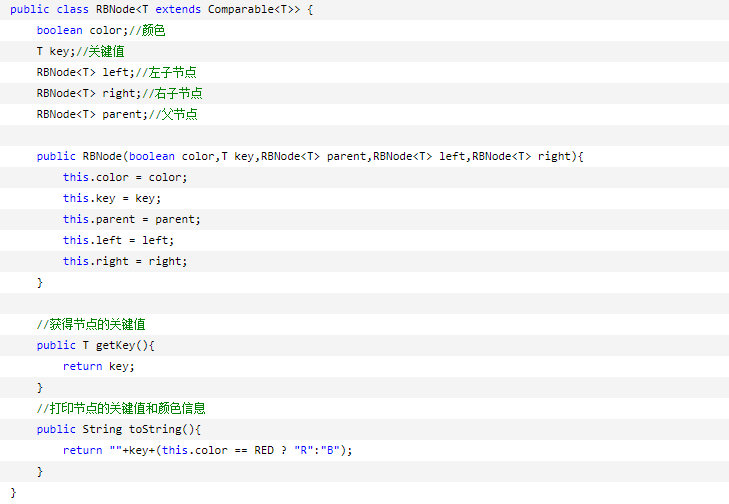
左旋的顶端节点必须要有右子节点。



### 3.6.3、左旋和右旋代码

1.节点类

节点类和二叉树的节点类差不多，只不过在其基础上增加了一个 boolean 类型的变量来表示节点的颜色。



2.左旋的具体实现



3.右旋的具体实现



### 3.6.4 插入操作

和二叉树的插入操作一样，都是得先找到插入的位置，然后再将节点插入。先看看插入的前段代码：



### 删除操作

上面探讨完了红-黑树的插入操作，接下来讨论删除，红-黑树的删除和二叉查找树的删除是一样的，只不过删除后多了个平衡的修复而已。我们先来回忆一下二叉搜索树的删除：

1. 如果待删除的节点没有子节点，那么直接删除即可。
2. 如果待删除的节点只有一个子节点，那么直接删掉，并用其子节点去顶替它。
3. 如果待删除的节点有两个子节点，这种情况比较复杂：首先找出它的后继节点，然后处理“后继节点”和“被删除节点的父节点”之间的关系，最后处理“后继节点的子节点”和“被删除节点的子节点”之间的关系。每一步中也会有不同的情况。

### 红黑树的效率

红黑树的查找、插入和删除时间复杂度都为O(log2N)，额外的开销是每个节点的存储空间都稍微增加了一点，因为一个存储红黑树节点的颜色变量。插入和删除的时间要增加一个常数因子，因为要进行旋转，平均一次插入大约需要一次旋转，因此插入的时间复杂度还是O(log2N),(时间复杂度的计算要省略常数)，但实际上比普通的二叉树是要慢的。

大多数应用中，查找的次数比插入和删除的次数多，所以应用红黑树取代普通的二叉搜索树总体上不会有太多的时间开销。而且红黑树的优点是对于有序数据的操作不会慢到O(N)的时间复杂度。