<http://www.jianshu.com/p/42f81846c0fb>

# 概念

## 算法

算法是指**解题方案的准确而完整的描述，是一系列解决问题的清晰指令，算法代表着用系统的方法描述解决问题的策略机制**。对于同一个问题的解决，可能会存在着不同的算法，为了衡量一个算法的优劣，提出了空间复杂度与时间复杂度这两个概念。

一个算法是由**控制结构**（顺序、分支和循环3种）和**原操作**（指固有数据类型的操作）构成的，则算法时间取决于两者的综合效果。为了便于比较同一个问题的不同算法，通常的做法是，**从算法中选取一种对于所研究的问题（或算法类型）来说是基本操作的原操作，以该基本操作的重复执行的次数作为算法的时间量度。**

## 时间复杂度

<http://blog.csdn.net/zolalad/article/details/11848739>

时间频度:一个**算法执行所耗费的时间**。

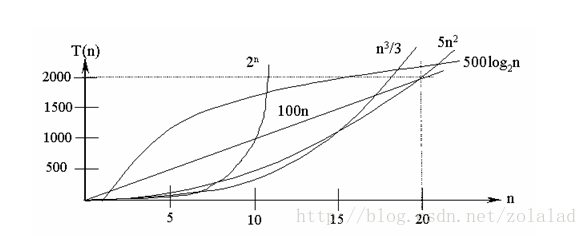
一个算法花费的时间与算法中语句的执行次数成正比例，哪个算法中语句执行次数多，它花费时间就多。一个算法中的语句执行次数称为语句频度或时间频度。记为T(n)

n称为**问题的规模**，当n不断变化时，时间频度T(n)也会不断变化。一般情况下，算法中基本操作**重复执行的次数**是**问题规模n**的某个函数，用T(n)表示，若有某个辅助函数f(n),使得当n趋近于无穷大时，T(n)/f(n)的极限值为不等于零的常数，则称f(n)是T(n)的同数量级函数。记作T(n)=Ｏ(f(n)),称Ｏ(f(n)) 为算法的渐进时间复杂度，简称时间复杂度。

简单来说，就是T(n)在n趋于正无穷时最大也就跟f(n)差不多大。

**算法中语句执行次数为一个常数，则时间复杂度为O(1)**。常见的时间复杂度有：**常数阶O(1),对数阶O(log2n),线性阶O(n), 线性对数阶O(n log2n),平方阶O(n2)，立方阶O(n3),...**。

Log28：2为底N的**对数**，即2的几次方等于8，结果为3



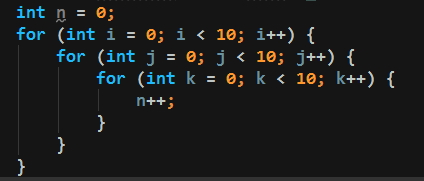
常见的算法时间复杂度由小到大依次为：Ο(1)＜Ο(log2n)＜Ο(n)＜Ο(n log2n)＜Ο(n2)＜Ο(n3)＜…＜Ο(2n)＜Ο(n!)

常数阶 < 对数阶 < 线性阶 < 线性对数阶 < 平方阶 < 立方阶 < … < 指数阶 < 阶乘

### 求解算法的时间复杂度的具体步骤

1. 找出算法中的基本语句；算法中**执行次数最多的那条语句**就是基本语句，通常是**最内层循环的循环体。**
2. 计算基本语句的**执行次数的数量级**；只需计算基本语句执行次数的数量级，这就意味着只要保证基本语句执行次数的函数中的**最高次幂正确**即可，可以忽略所有低次幂和最高次幂的系数。这样能够简化算法分析，并且使注意力集中在最重要的一点上：增长率。
3. 用大Ο记号表示算法的时间性能。

如：



第一个for循环的时间复杂度为Ο(n)，第二个for循环的时间复杂度为Ο(n2)，则整个算法的时间复杂度为Ο(n1+n2+n3)=Ο(n3)。

Ο(1)表示基本语句的执行次数是一个常数，一般来说，只要算法中不存在循环语句，其时间复杂度就是Ο(1)。其中**Ο(log2n)、Ο(n)、 Ο(nlog2n)、Ο(n2)和Ο(n3)称为多项式时间**，而Ο(2n)和Ο(n!)称为指数时间。计算机科学家普遍认为前者（即**多项式时间复杂度的算法）是有效算法。**

指数函数：y=ax，对数函数：y=logax，幂函数：y=xa

x为变量，a为常量

一个经验规则：其中c是一个常量，如果一个算法的复杂度为c 、 log2n 、n 、 n\*log2n ,那么这个算法时间效率比较高 ，如果是2n ,3n ,n!，那么稍微大一些的n就会令这个算法不能动了，居于中间的几个则差强人意。

## 空间复杂度

<http://blog.csdn.net/zolalad/article/details/11848739>

类似于时间复杂度的讨论，一个算法的空间复杂度(Space Complexity)S(n)定义为**该算法所耗费的存储空间**，它也是问题规模n的函数。

包括存储**算法本身所占用的存储空间**

算法的**输入输出数据所占用的存储空间**

算法在**运行过程中临时占用的存储空间**这三个方面。

当一个算法的空间复杂度为一个常量，即不随被处理数据量n的大小而改变时，可表示为O(1)；

# 数据结构

## 线性结构

一个线性表是n个数据元素的有限序列，线性表中的数据元素可以是各种各样的，但同一个线性表中的元素必须具有相同特性，即属于同一数据对象，相邻的数据元素间存在着序偶关系（直接前驱，直接后继）。

**线性表的顺序表示**：便于查找，但插入删除时需要移动大量元素。

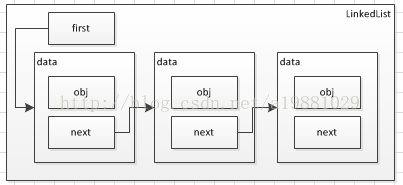
如：数组：指用一组地址连续的存储单元依次存储线性表的数据元素，逻辑上相邻的数据元素在屋里地址上也相邻。

**线性表的链式表示**：用一组任意的存储单元存储线性表的数据元素（可以连续，也可以不连续），逻辑上相邻的两个元素其存储的物理位置不一定相邻。

### 数组

### 链表

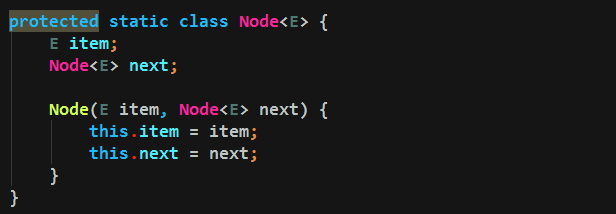
#### 单链表



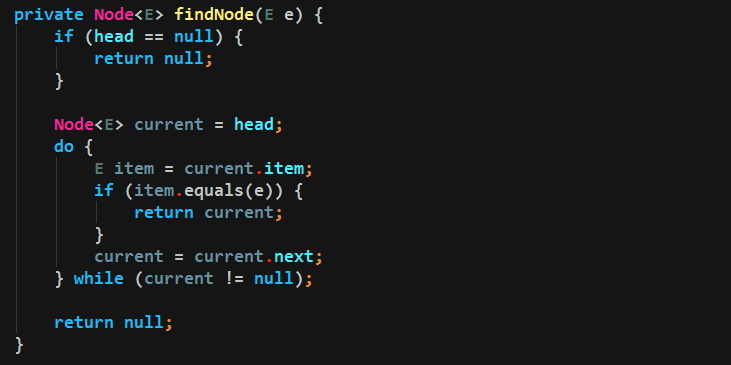
在表头插入或删除一个元素的时间复杂度为**常数**阶O(1)。

查找和移除链表中指定元素的时间复杂度为**线性**阶O(n)。

链表结点数据结构：



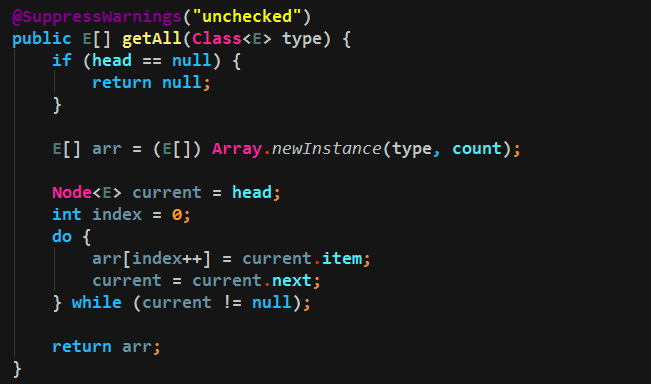
查找元素：



移除元素：



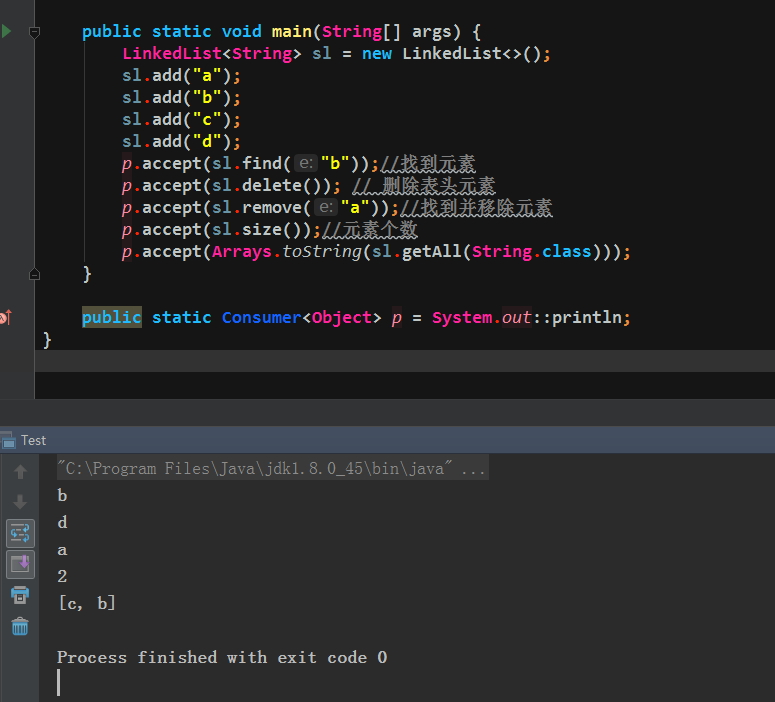
遍历所有元素：



头结点插入和移除：

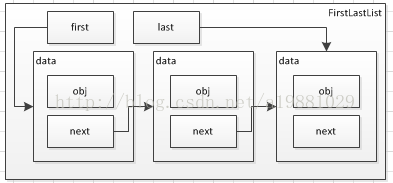


测试：



#### 双端链表

与单向链表的不同之处在保存有对最后一个链接点的引用(last)

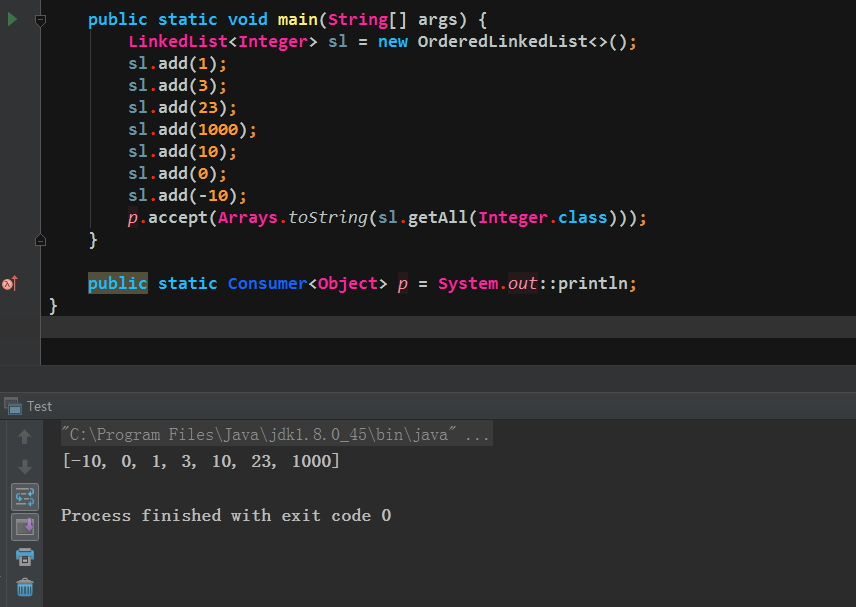


优势在于在表尾进行删除或插入操作。

#### 有序链表



测试：

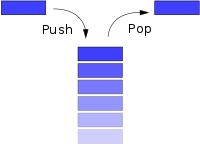


### 队列

### 堆栈

堆栈（英语：stack）又称为栈或堆叠，是计算机科学中一种特殊的串列形式的抽象资料型别，其特殊之处在于**只能允许在链接串列或阵列的一端（称为堆叠顶端指标，英语：top）进行加入数据（英语：push）和输出数据（英语：pop）的运算**。另外栈也可以用**一维数组**或**链表**的形式来完成。。

由于堆叠数据结构只允许在一端进行操作，因而**按照后进先出（LIFO, Last In First Out）的原理运作。**



两种基本操作：

推入：将数据放入堆叠的顶端（栈顶）

弹出：将顶端数据资料输出（回传）

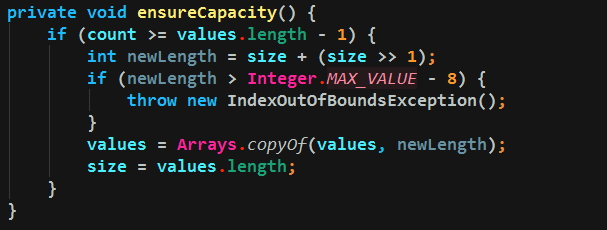
特定：

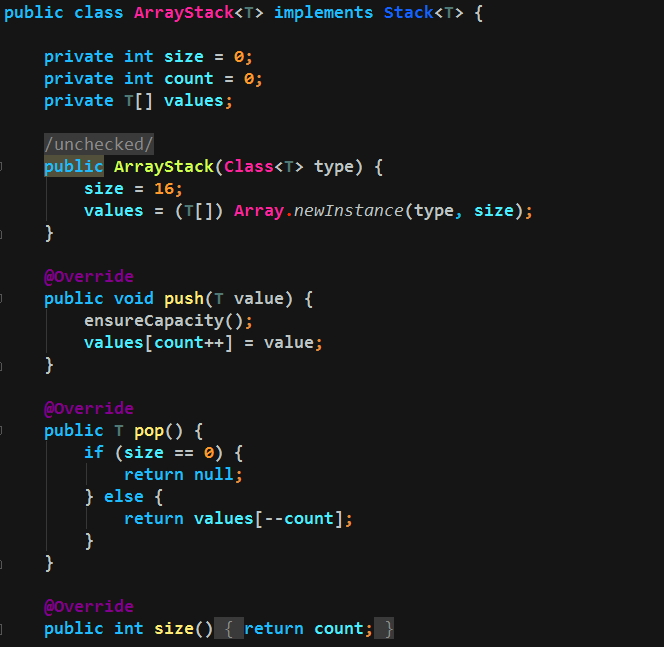
**先入后出，后入先出。**

**除头尾节点之外，每个元素有一个前驱，一个后继**。

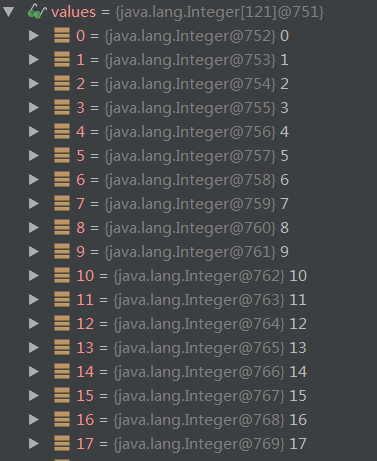
#### 数组实现

数组扩容函数（参考ArrayList）



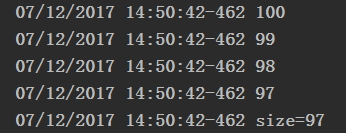


测试：



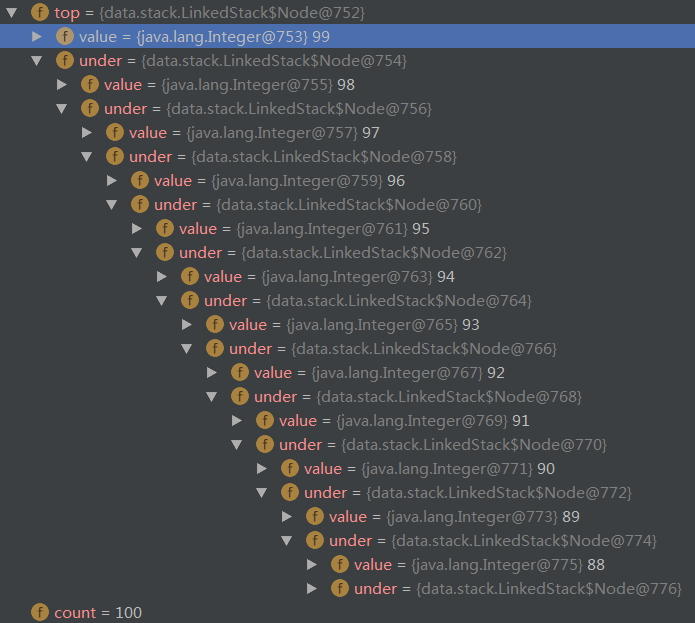


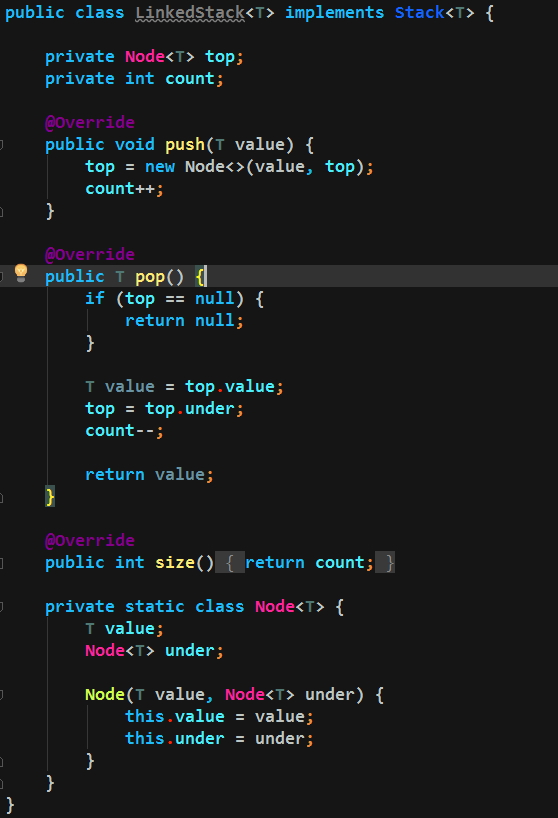
输出：



#### 链表实现

测试可参考数组实现时的测试。



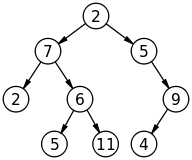


## 非线性结构

### 树

#### 二叉树

<https://zh.wikipedia.org/wiki/%E4%BA%8C%E5%8F%89%E6%A0%91>



二叉树（英语：Binary tree）是每个节点最多只有两个分支(不存在分支度大于2的节点)的树结构。通常分支被称作“左子树”和“右子树”。二叉树的分支具有左右次序，不能颠倒。

##### 性质

1. 二叉树的第i层至多有2^(i-1)个节点
2. 深度为k的二叉树，至多有2^(k+1) – 1个节点（定义根节点所在深度为0）
3. 对任何一棵非空的二叉树T，如果其叶片(终端节点)数为n0，分支度为2的节点数为n2，则n2 = n0 - 1。

##### 用途

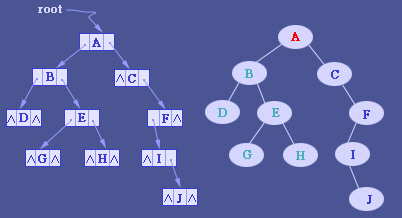
典型用法是对节点定义一个标记函数，将一些值与每个节点相关系。这样标记的二叉树就可以实现二叉查找树和二元堆积，并应用于**高效率的搜索和排序**。

##### 存储二叉树的方法

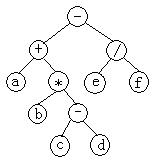
基于数组的存储结构

https://upload.wikimedia.org/wikipedia/commons/thumb/8/86/Binary_tree_in_array.svg/220px-Binary_tree_in_array.svg.png

基于链表的存储结构

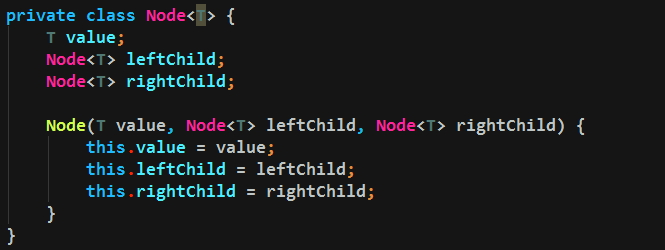


##### 遍历



* 先序dlr遍历的序列是：-+a\*b-cd/ef
* 中序ldr遍历的序列是：a+b\*c-d-e/f
* 后序lrd遍历的序列是：abcd-\*+ef/-

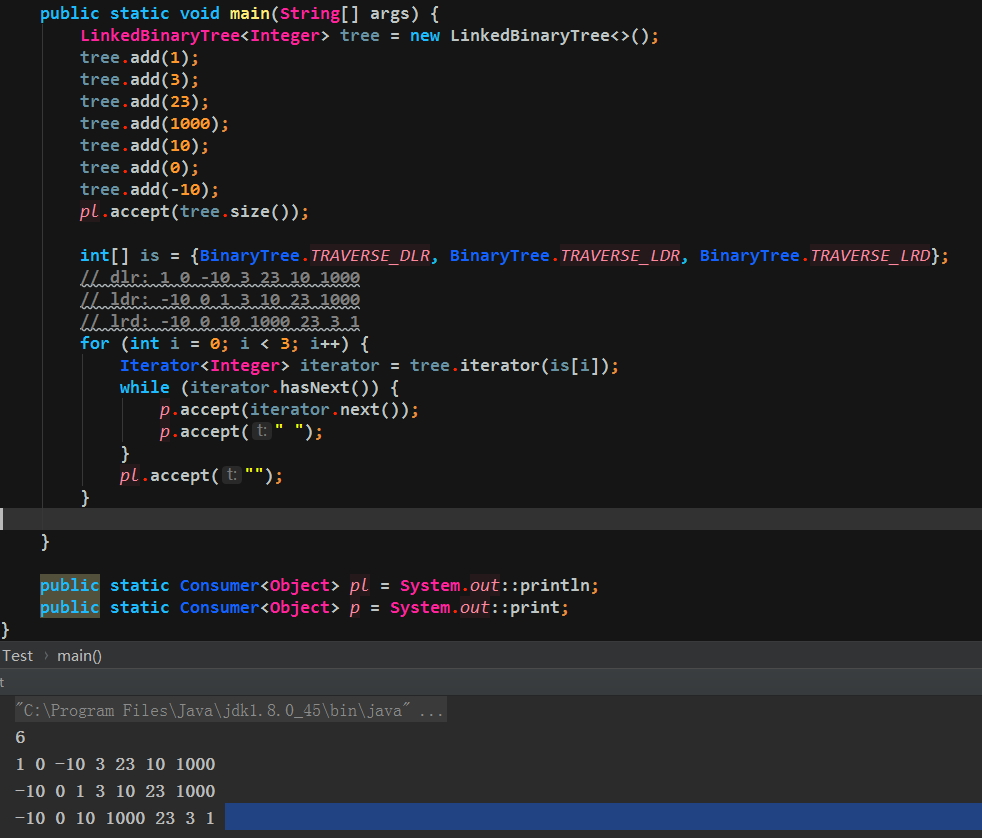
链表结构数据节点定义：



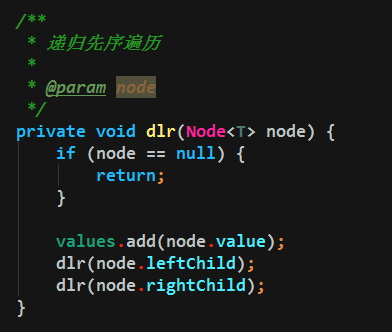
按照元素比较顺序（Comparable）插入（二叉查找树的规则）：



测试



###### 前(先)序遍历，先根遍历-DLR



###### 中序遍历，中根遍历-LDR



###### 后序遍历，后跟遍历-LRD



#### 二叉查找树

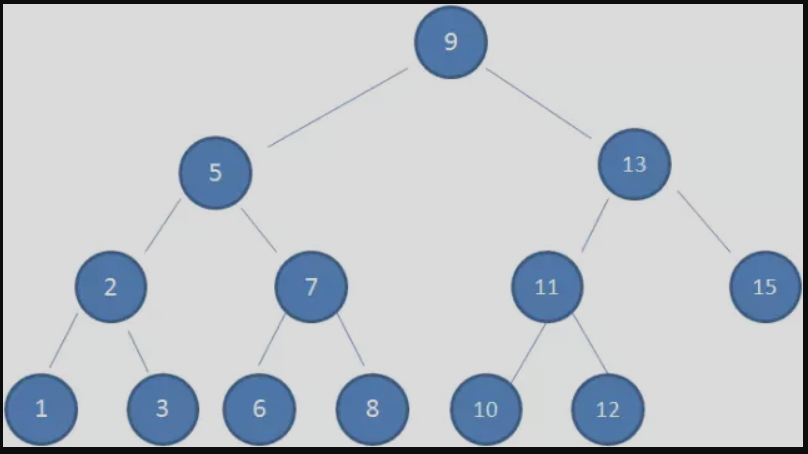
<http://mp.weixin.qq.com/s?__biz=MzIxNTIwOTgxNw==&mid=2650612655&idx=1&sn=09bd44c8bf9978c8beedcdf5a07d1047&chksm=8f924199b8e5c88f45a0ffc8571c820d830c4a332a2c16f162d48295e70ea796ebabc4773fdf&mpshare=1&scene=23&srcid=1125u1TfchKscPuptKpdLlMC#rd>

**特性**

**1. 左子树上所有结点的值均小于或等于它的根结点的值。**

**2. 右子树上所有结点的值均大于或等于它的根结点的值。**

**3. 左、右子树也分别为二叉排序树。**



问：找出值为10的节点

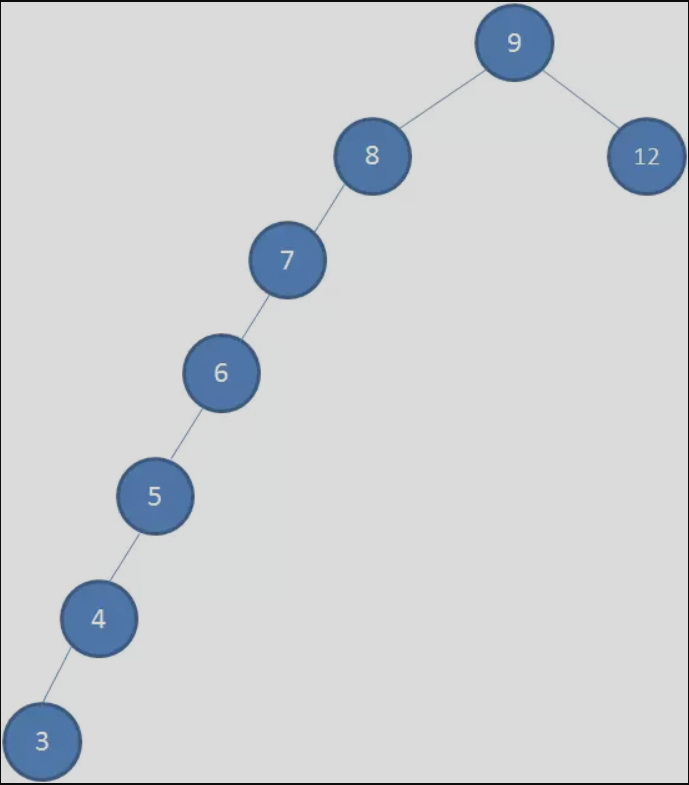
先看9,9小于10，根据二叉查找树的特性，10应该在节点9的右节点子树上。

依次查看9-13-11->9，可找到9所在节点。

这种方式正是二分查找的思想，查找所需的最大次数等于二叉树的高度。插入节点时也是基于该特性。

**缺陷**

依次插入如下五个节点：7,6,5,4,3。依照二叉查找树的特性，结果会变成什么样呢？



虽然符合二叉查找数的特性，但查找效率大打折扣，几乎变成线性。

为了**解决二叉查找树多次插入新节点导致的不平衡**，红黑树应运而生。

#### 红黑树

自平衡的二叉树，符合二叉查找树的基本特性，此外还具有如下特性：

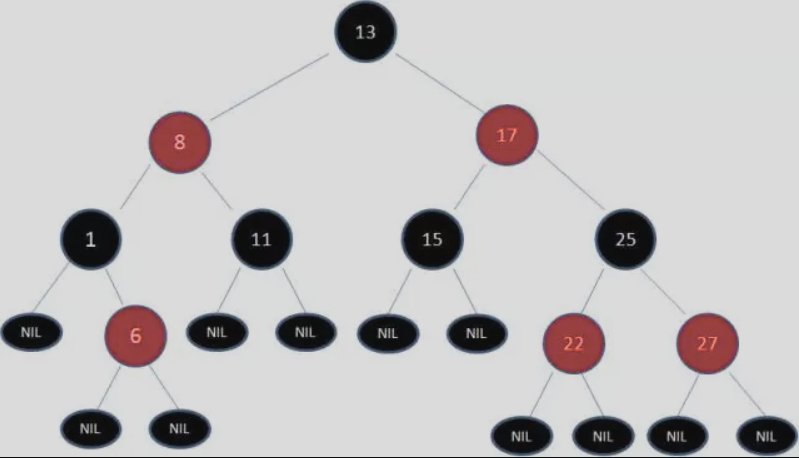
**1. 节点是红色或黑色。**

**2. 根节点是黑色。**

**3. 每个叶子节点都是黑色的空节点（NIL节点）。**

**4. 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)**

**5. 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。**

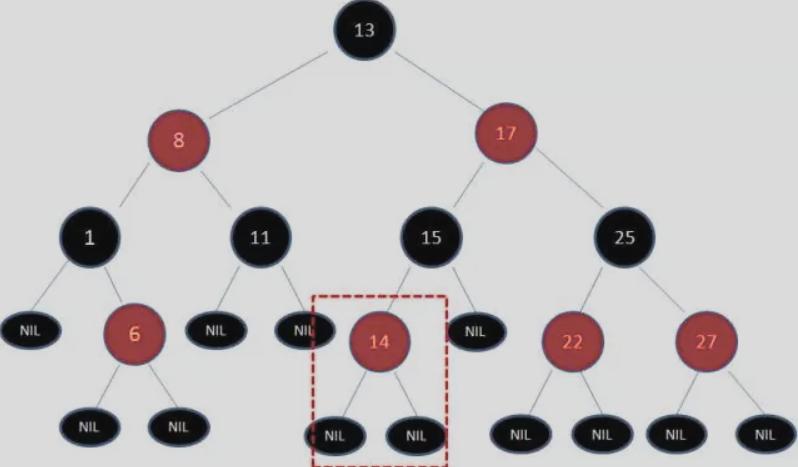


查找值为10的节点：13-8-11-NULL

红黑树从根到叶子的最长路径不会超过最短路径的2倍。

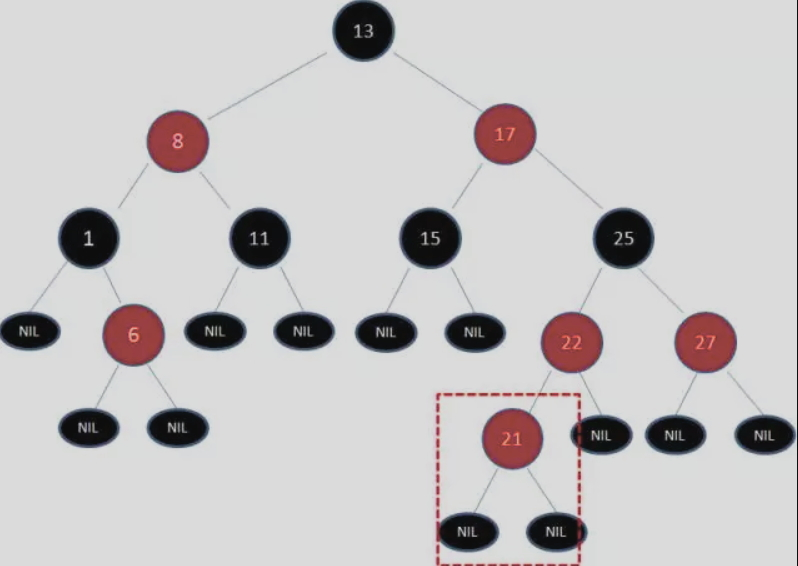
当插入或删除节点时，红黑树的规则有可能被打破，此时需要做出调整，来继续维持红黑树的规则。

1. 向原红黑树插入值为14的新节点：



由于父节点15是黑色节点，因此这种情况并不会破坏红黑树的规则，无需做任何调整。

1. 向原红黑树插入值为21的新节点：



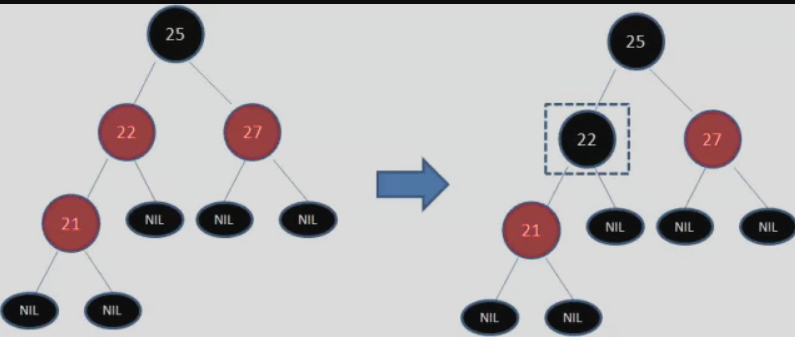
由于父节点22是红色节点，因此这种情况打破了红黑树的规则4（每个红色节点的两个子节点都是黑色），必须进行调整，使之重新符合红黑树的规则。

调整有两种方法：**变色**和**旋转**（旋转又分左旋转和右旋转）。

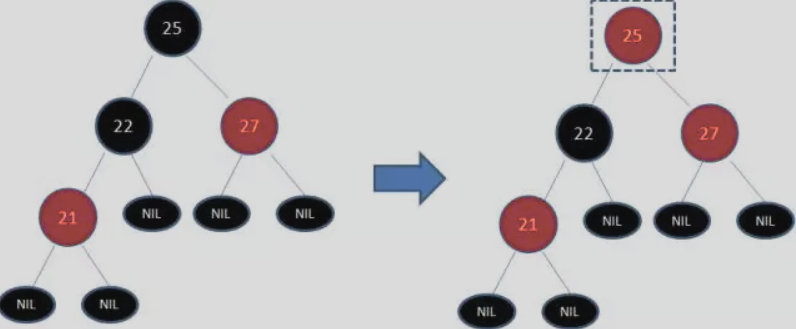
* 变色

为了重新符合红黑树的规则，尝试把红色节点变为黑色，或者把黑色节点变为红色。

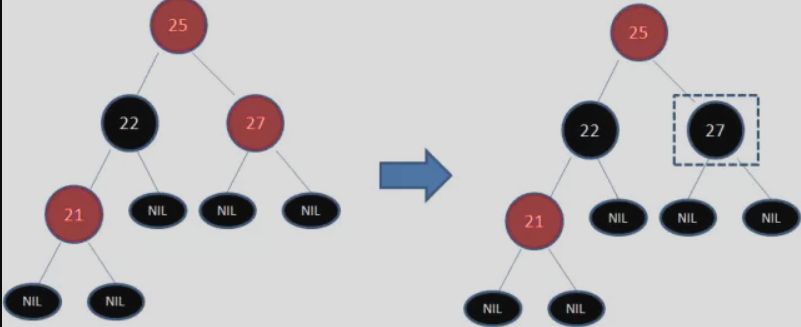
下图所表示的是红黑树的一部分，需要注意节点25并非根节点。因为节点21和节点22连续出现了红色，不符合规则4，所以把节点22从红色变成黑色：



但这样并不算完，因为凭空多出的黑色节点打破了规则5，所以发生连锁反应，需要继续把节点25从黑色变成红色：



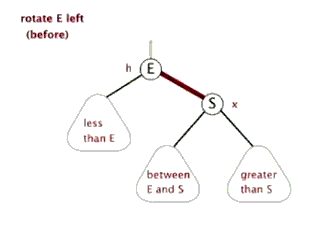
此时仍然没有结束，因为节点25和节点27又形成了两个连续的红色节点，需要继续把节点27从红色变成黑色：

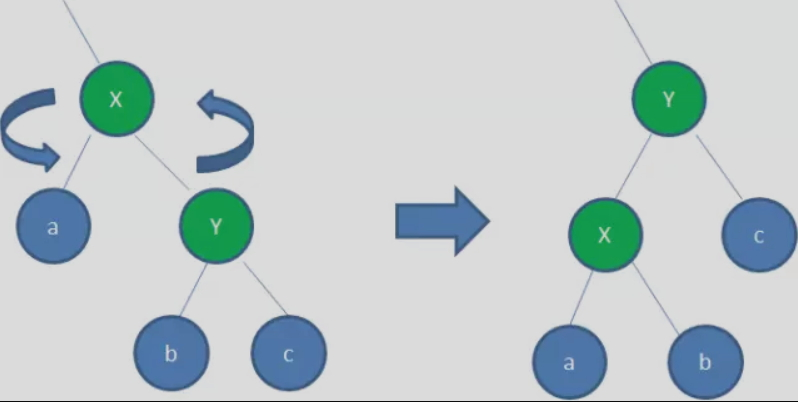


* 旋转

**左旋转**

逆时针旋转红黑树的两个节点，使得父节点被自己的右孩子取代，而自己成为自己的左孩子。说起来很怪异，大家看下图：

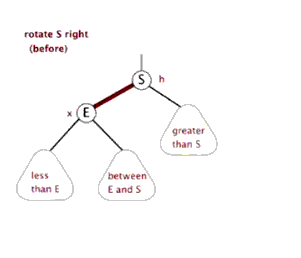


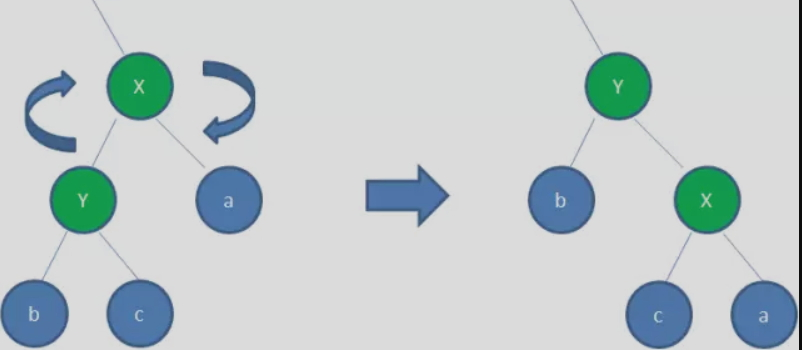


图中，身为右孩子的Y取代了X的位置，而X变成了自己的左孩子。此为左旋转。

**右旋转**

顺时针旋转红黑树的两个节点，使得父节点被自己的左孩子取代，而自己成为自己的右孩子。大家看下图：





图中，身为左孩子的Y取代了X的位置，而X变成了自己的右孩子。此为右旋转。

实际应用

JDK集合类，TreeMap和TreeSet底层用的是红黑树，java8中HashMap也用到了红黑树。

### 图

### 表

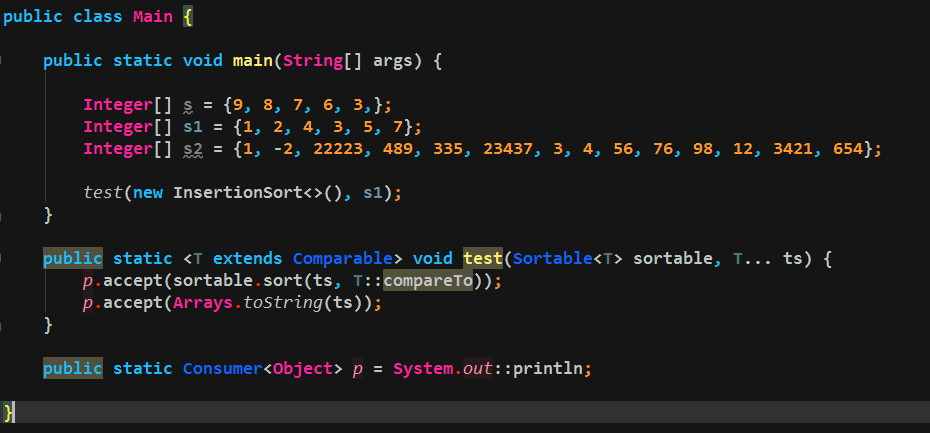
## 哈希结构

# 算法

## 排序

<http://www.jianshu.com/p/42f81846c0fb>

测试方法



### 冒泡排序 – 逐个比较相邻元素

<https://zh.wikipedia.org/wiki/%E5%86%92%E6%B3%A1%E6%8E%92%E5%BA%8F>

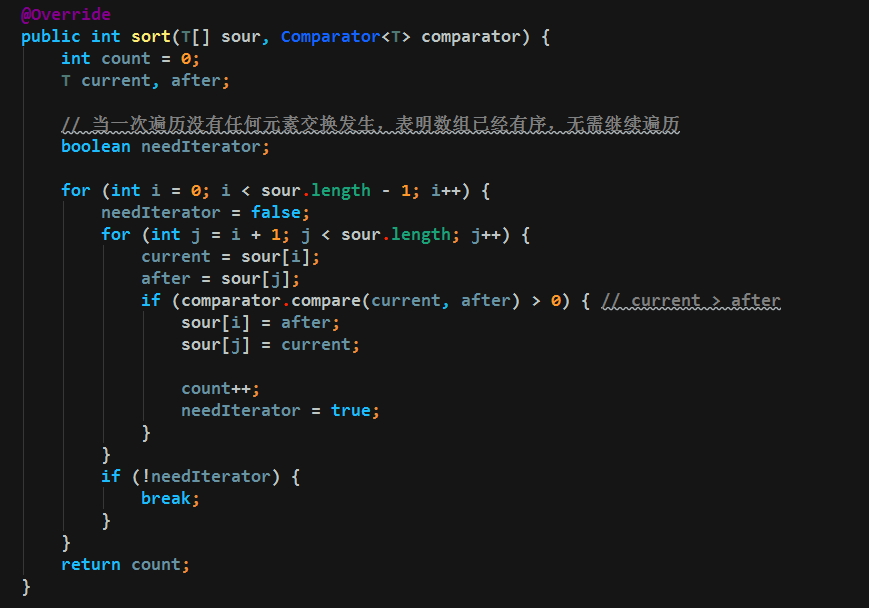
时间复杂度O(n2)，最好情况O(n)，最坏情况O(n2)，稳定。

**适合数据规模很小的时候，而且它的效率也比较低。**

通过重复地走访要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。

**算法描述如下：**

1. 比较相邻的元素。如果第一个比第二个大，就交换他们两个。
2. 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。这步做完后，最后的元素会是最大的数。
3. 针对所有的元素重复以上的步骤，除了最后一个。
4. 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。



### 直接选择排序 – 选择剩下的元素里最大(小)放到最后(前)

时间复杂度O(n2)，最好情况O(n2)，最坏情况O(n2)，不稳定。

选择排序（Selection sort）是一种简单直观的排序算法。**元素交换次数少，比较占多数。**

它的工作原理如下：

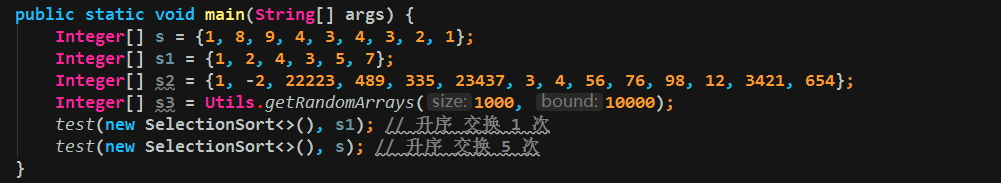
首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置

然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾

以此类推，直到所有元素均排序完毕。

选择排序的主要优点与数据移动有关。如果某个元素位于正确的最终位置上，则它不会被移动。选择排序每次交换一对元素，它们当中至少有一个将被移到其最终位置上，因此**对 n个元素的表进行排序总共进行至多 n-1次交换。**在所有的完全依靠交换去移动元素的排序方法中，选择排序属于非常好的一种。

示例：



交换次数比冒泡排序较少，由于交换所需CPU时间比比较所需的CPU时间多，**选择排序比冒泡排序快。**



### 插入排序 – 将元素比较插入到前面的位置

<https://zh.wikipedia.org/wiki/%E6%8F%92%E5%85%A5%E6%8E%92%E5%BA%8F>

时间复杂度O(n2)，最好情况O(n)，最坏情况O(n2)，稳定。（同冒泡）

是一种简单直观的排序算法。适用于**量级小于千，或者若已知输入元素大致上按照顺序排列**。

插入排序不适合对于数据量比较大的排序应用。

工作原理是**通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。**

**算法描述如下：**

1. 从第一个元素开始，该元素可以认为已经被排序
2. 取出下一个元素（第二个元素）作为目标元素，在已经排序的元素序列中**从后向前**扫描
3. 如果当前元素（扫描到的元素）大于目标元素，将当前元素与目标元素交换
4. 重复步骤3，直到找到目标元素的位置（当前元素小余或等于目标元素）
5. 重复步骤2-4

先对数组前两个元素进行排序，之后将第三个元素与前两个比较，按排序规则插入正确的位置，之后将第四个元素与前三个，比较插入正确的位置，以此类推直至最后一个元素

如果比较操作的代价比交换操作大的话，可以采用二分查找法来减少比较操作的数目。该算法可以认为是插入排序的一个变种，称为**二分查找插入排序**。



### 归并排序

### 希尔（Shell）排序

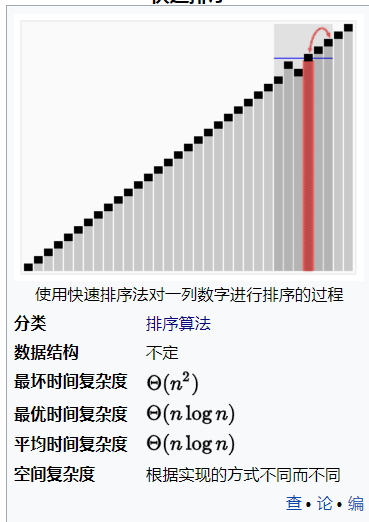
### 快速排序 – 将“基准”前，后不符合规则的元素交换

<https://zh.wikipedia.org/wiki/%E5%BF%AB%E9%80%9F%E6%8E%92%E5%BA%8F>

快速排序（英语：Quicksort），又称划分交换排序（partition-exchange sort）。

时间复杂度O(nlog2n)，最坏状况下O(n2)，但这种状况并不常见。

事实上，快速排序通常明显比其他O(nlog2n)算法更快，因为它的内部循环（inner loop）可以在大部分的架构上很有效率地被实现出来。

快速排序使用**分治法**策略来把一个序列分为两个子序列。

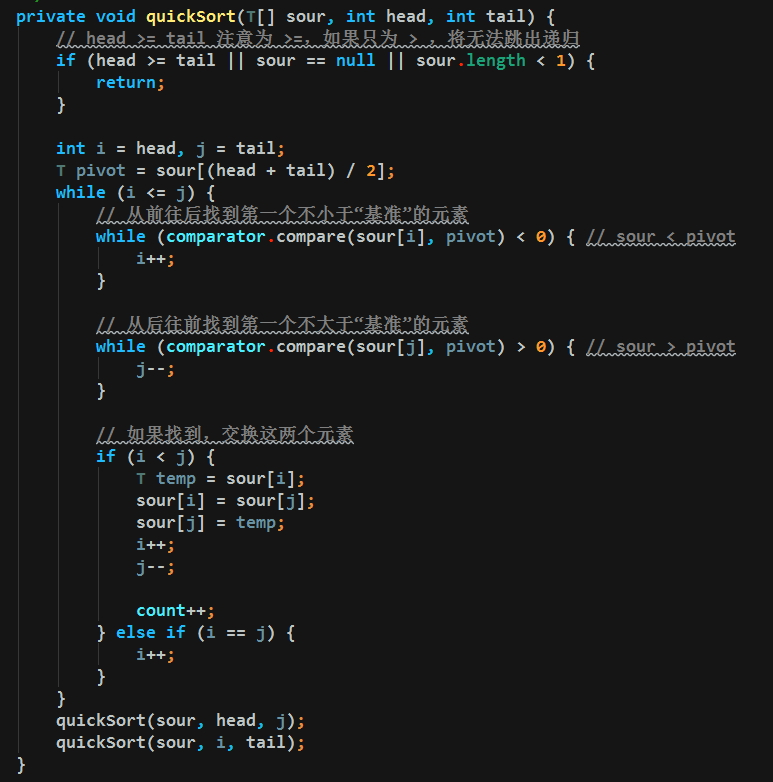
步骤为：

1. 选取“基准”：从数列中挑出一个元素，称为"基准"（pivot）
2. 分区（partition）操作：重新排序数列，**所有比基准值小的元素摆放在基准前面**，**所有比基准值大的元素摆在基准后面**（相同的数可以到任何一边）。在这个分区结束之后，该基准就处于数列的中间位置。
3. 递归地把小于基准值元素的子数列和大于基准值元素的子数列排序。

递归到最底部时，数列的大小是零或一，也就是已经排序好了。

这个算法一定会结束，因为在每次的迭代（iteration）中，它**至少会把一个元素摆到它最后的位置去。（最终排好序时其应该在的位置）**





取中间位置元素为“基准”

取“基准”以左最左元素为“头”

取“基准”以右最右元素为“尾”

从两端向中间遍历：

找到“头”往后第一个不小于“基准”的元素a及其下标i

找到“尾”往前第一个不大于“基准”的元素b及其下标j

如果i<j，交换a和b，继续循环；

如果i==j，此次排序结束，开始下一次递归

如果i>j，结束循环，此次排序结束，开始下一次递归

### 堆排序



## 查找

### 深度优先搜索

### 广度优先搜索

### 顺序查找

### 二分查找

时间复杂度：最坏O(Logn) 最优O(1) 平均O(Logn)

二分搜索（英语：binary search），也称折半搜索（英语：half-interval search）。

是一种在**有序**数组中查找某一特定元素的搜索算法。

可用于查找外，还可用在插入排序中。

搜索过程从数组的中间元素开始，如果中间元素正好是要查找的元素，则搜索过程结束；

如果某一特定元素大于或者小于中间元素，则在数组大于或小于中间元素的那一半中查找，且跟开始一样从中间元素开始比较。

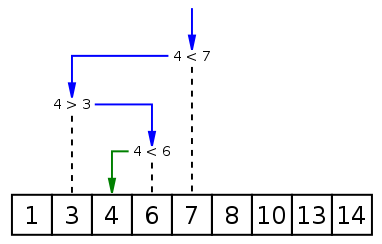
如果在某一步骤数组为空，则代表找不到。

这种搜索算法每一次比较都使搜索范围缩小一半。

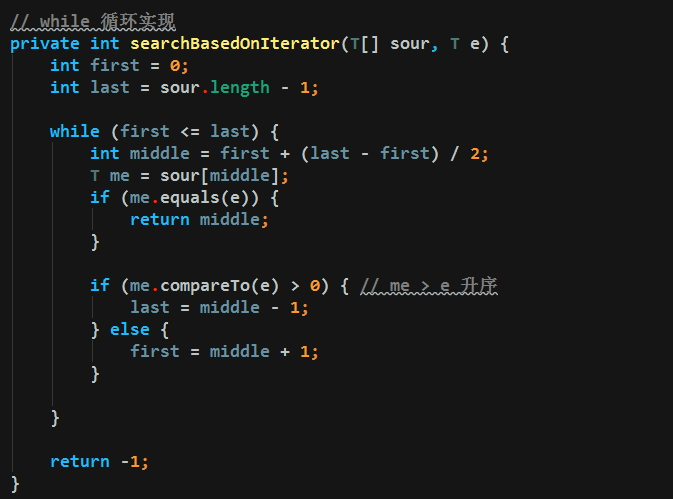
**步骤：**

给予一个包含n个带值元素的数组 A或是记录A0, … ,An-1，使 A0<= … An-1，以及目标值 T，还有下列用来搜索 T在 A中位置的子程序

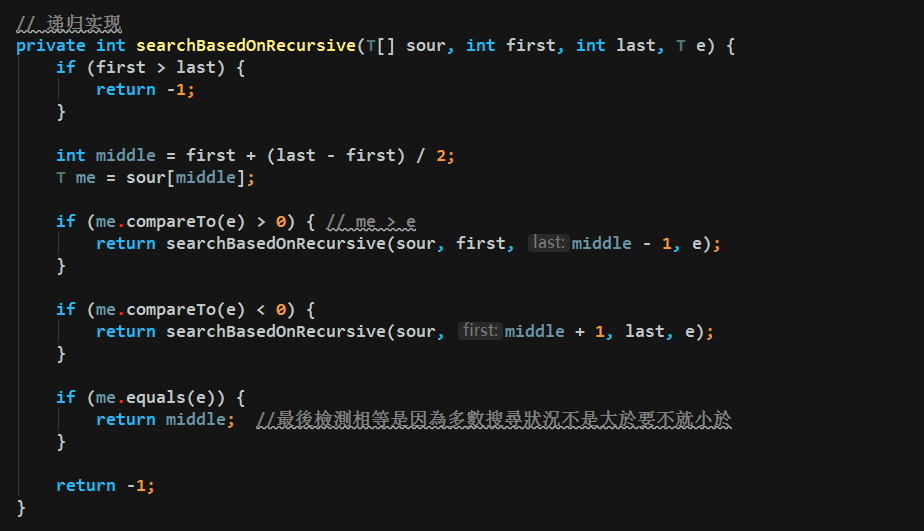
1. 令F为0，L为n-1
2. 如果F>L，则搜索以失败告终
3. 令m（中间值元素下标）为(F+L)/2
4. 如果Am < T，令F为m+1并回到步骤2
5. 如果Am > T，令L为m-1并回到步骤2
6. 当Am=T，搜索结束，回传值m（或Am）



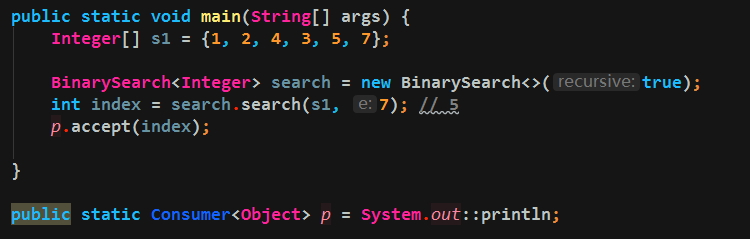
循环实现：



递归实现：



测试：



## Dijkstra （最短路径算法）

## 递归

## 分治算法

## 动态规划

## 贪心算法

## 回溯算法

## 匹配算法

## 正则表达式和字符串匹配



# 习题

<http://www.codeceo.com/article/15-algorithms-question.html>

<http://dongxicheng.org/structure/structure-algorithm-summary/>