数据结构

《数据结构》这门课是计算机专业的核心课程，但往往却让人头痛，因为比较抽象，当然了，也许你足够聪明，并不觉得它有多难，但对我而言，是有点难度，后来我仔细想了想，到底哪里难？我得出这么个结论：长篇大论，缺乏图表。现在的人都喜欢看电影，看电视剧，很少人还热衷于看小说吧，密密麻麻的文字不如一些图来得直观。

1.大圈表示法

面试时候如果让你写一个算法，要求复杂度为O(n)，你明白是什么意思吗？说起数据结构，就先提一下这个表示法吧，后面会用到。

“O”，其实不是英文的“O”，它是个希腊字母，发音大概是“欧麦克隆”，所以我们一般说“圈”而不是跟英文的O一样的发音。简单地说，大圈表示法是一种用于表示算法复杂度数量级的方法。要精确描述这个表示法，很难，不过我们不需要懂那么精确，只要八九不离十就可以了。下面我列个表，复杂度从低到高，大家就知道其意义：

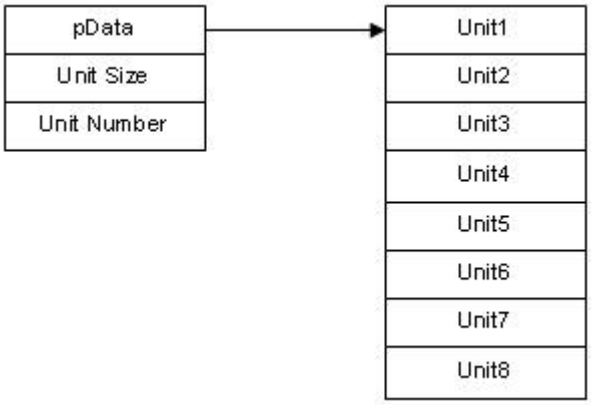
范例	解释
有10个数字，求它们的平均值	无论这10个数字是什么，电脑总是能在一个确定的时间内返回结果，所以复杂度为O(1)
有10000个数字，求它们的开方的和	虽然比上面的复杂，但电脑还是能在一个确定的时间内返回结果，所以复杂度还是O(1)
二分查找法（后面会提到）有一队从小到大排序的数字，用“对半法”查找出你想要的那个数的位置	假设有1000个数字，最坏的情况是要查找(log1000)/(log2)这么多次（注意：是大概，实际上要少一点，具体暂时不说），所以复杂度为O(logn)
有一队无序的数字，从头到尾查找你想要的一个数的位置	随着这一堆数数量的增多，电脑的工作量会呈线性递增，所以复杂度为O(n)
快速排序（后面会提到）	最坏的情况是O(n^2)，一般预期的情况是O(nlogn)，先不解释了。
一般平方矩阵乘法（两层循环）	2*2的方阵要算4次，4*4的方阵要算16次，呈平方递增，所以复杂度为O(n^2)
冒泡法排序（后面会提到）	对N个元素的数组，要比较和移动N*(N/2)次，所以复杂度为O(n^2)
一般立方矩阵乘法（三层循环）	2*2*2的立方阵要算8次，4*4*4的立方阵要算64次，呈立方递增，所以复杂度为O(n^3)

另外还有个叫指数复杂度，这里不提，因为见得实在太少，“指数级递增”本身就是一个很夸张的形容词，我们也要避免这种复杂度的出现。还需要说明的一点是大圈表示法是时间递增数量级的表示方法，注意“递增”两个字，所以并不是说复杂度为O(1)的算法消耗的时间一定比复杂度为O(n)的算法少。

如果你还是不太明白大圈表示法，不用担心，继续往下看，会慢慢明白的。

2.动态数组（Dynamic Array）

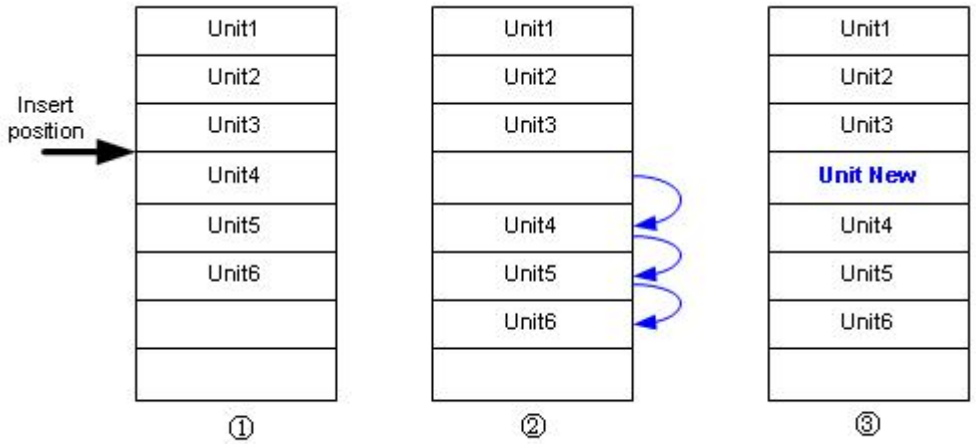
接下去介绍最最基本的两种数据结构，即动态数组和单向链表，其它数据结构其实都可以通过这两者衍生出来。BTW：如果算法太简单，我就不列出代码，只稍微描述一下。



这就是一个最基本的动态数组，pData 记录了数组第一个元素的位置，Unit Size 记录了每个元素的大小，（这样可以方便地找到第 N 个元素了）Unit Number 记录了元素的数目。

获取数组中第 N 个元素，是很简单的，无需多说。

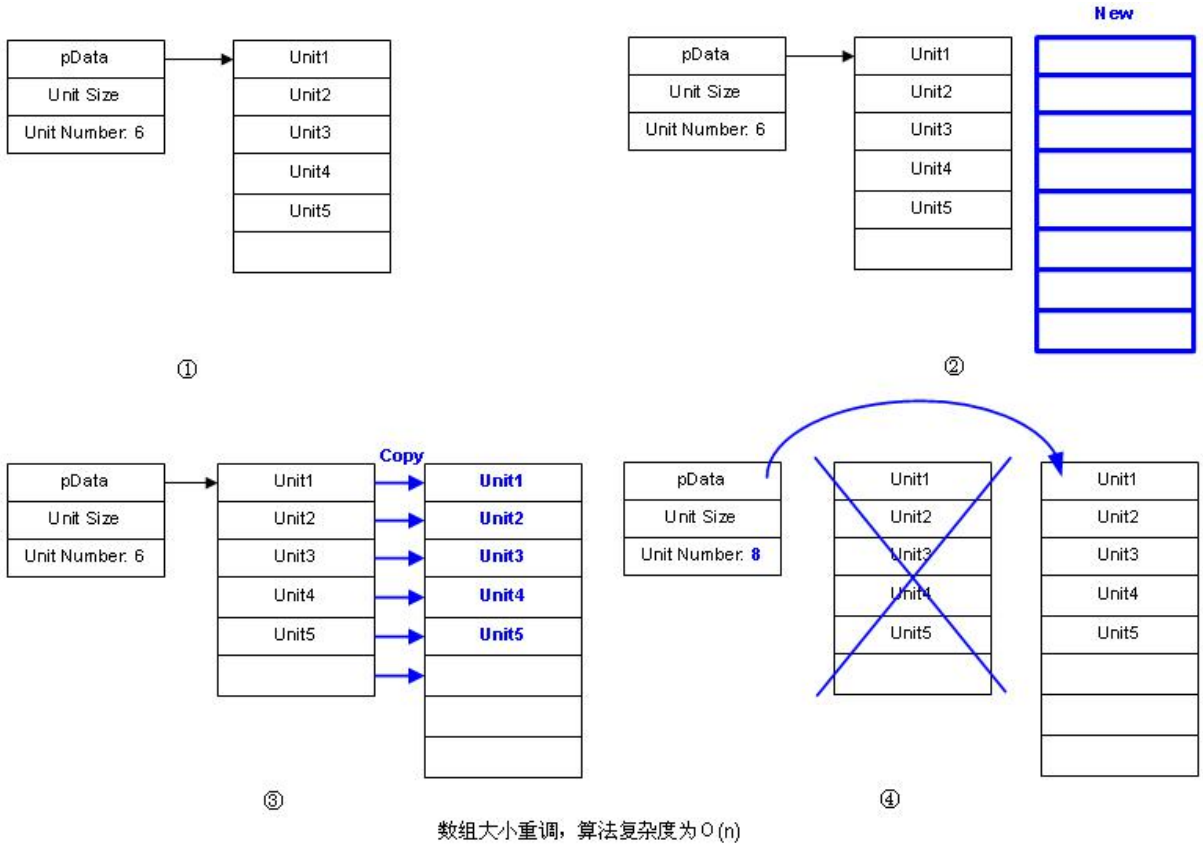
但已知某位置，要插入一个元素，就稍微有点难，因为要挪动一些元素，如图：



从已知位置插入一个元素，复杂度为 O(n)

删除元素跟这个也类似，也是需要挪一挪后面的元素，只不过是往前挪。

数组的大小不能很方便地调整，需要几个步骤，如下图所示：

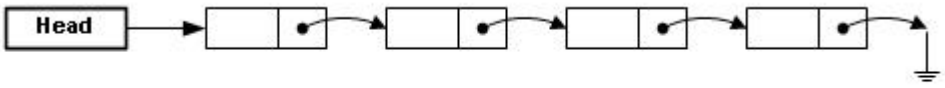


数组大小重调，算法复杂度为 O(n)

大体步骤就是 new/malloc，memcpy，delete/free 这几个步骤。

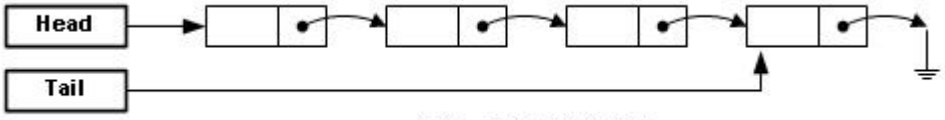
3、单向链表（Singly-linked List）

下图就是最简单最一般的单向链表：



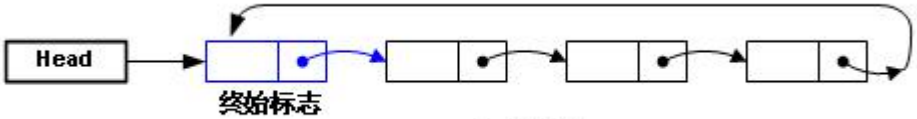
①最普通链表

还有这种：



②带末尾指针的链表

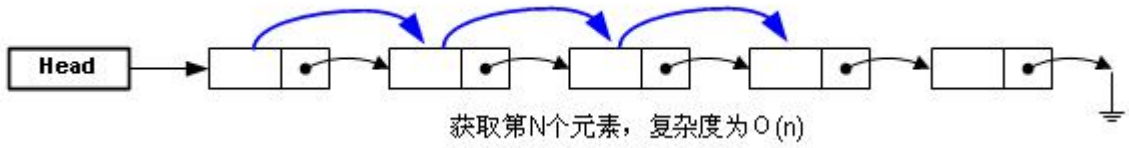
多一个 Tail 指针，好处就是能很方便地找到末尾，然后在末尾插入新的元素什么的。还有这种也比较常见：



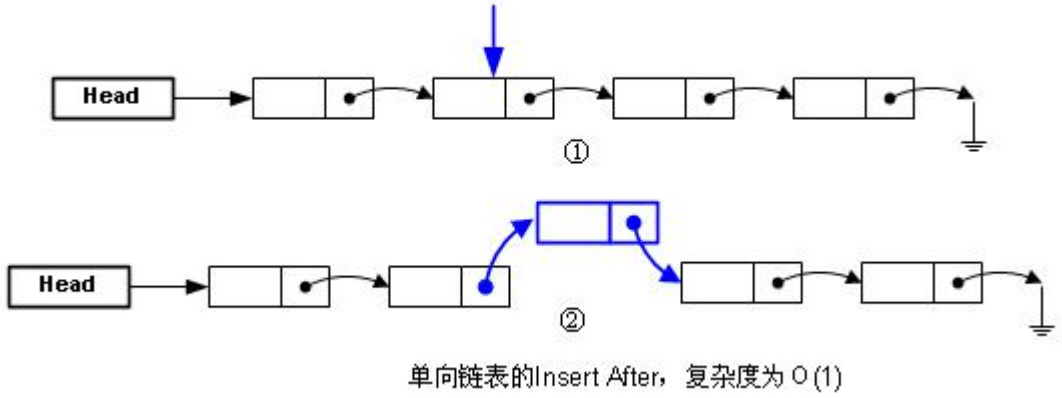
③环形链表

留一个终始标志，这个节点作为一个标志，不用于存储数据，链表末尾指向这个节点，形成一个“环形链表”，这样无论在链表的哪里插入新的元素，操作都一致了，不必判断头和尾的特殊性。

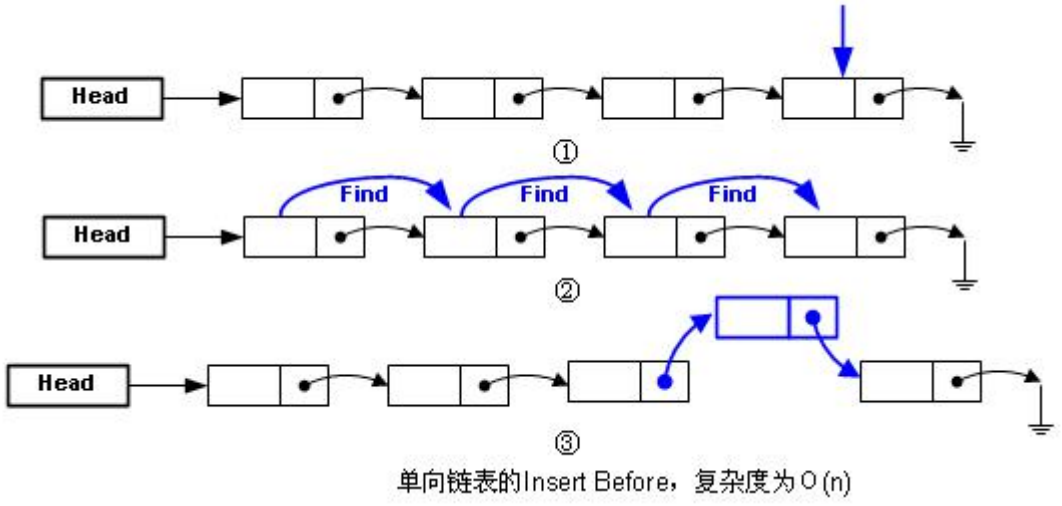
数组的好处就是链表的坏处，数组的坏处就是链表的好处，请看：



因为需要从头开始找，没办法像数组那样直接跳到那个地址。而插入元素，就比数组方便了，如果你已经得知了要插入的地址的话，不过还要注意哦，是“后插入”（Insert After）：



有“后插入”，那就有“前插入”（Insert Before），两者对单向链表来说真的不一样，下图描述了“前插入”：

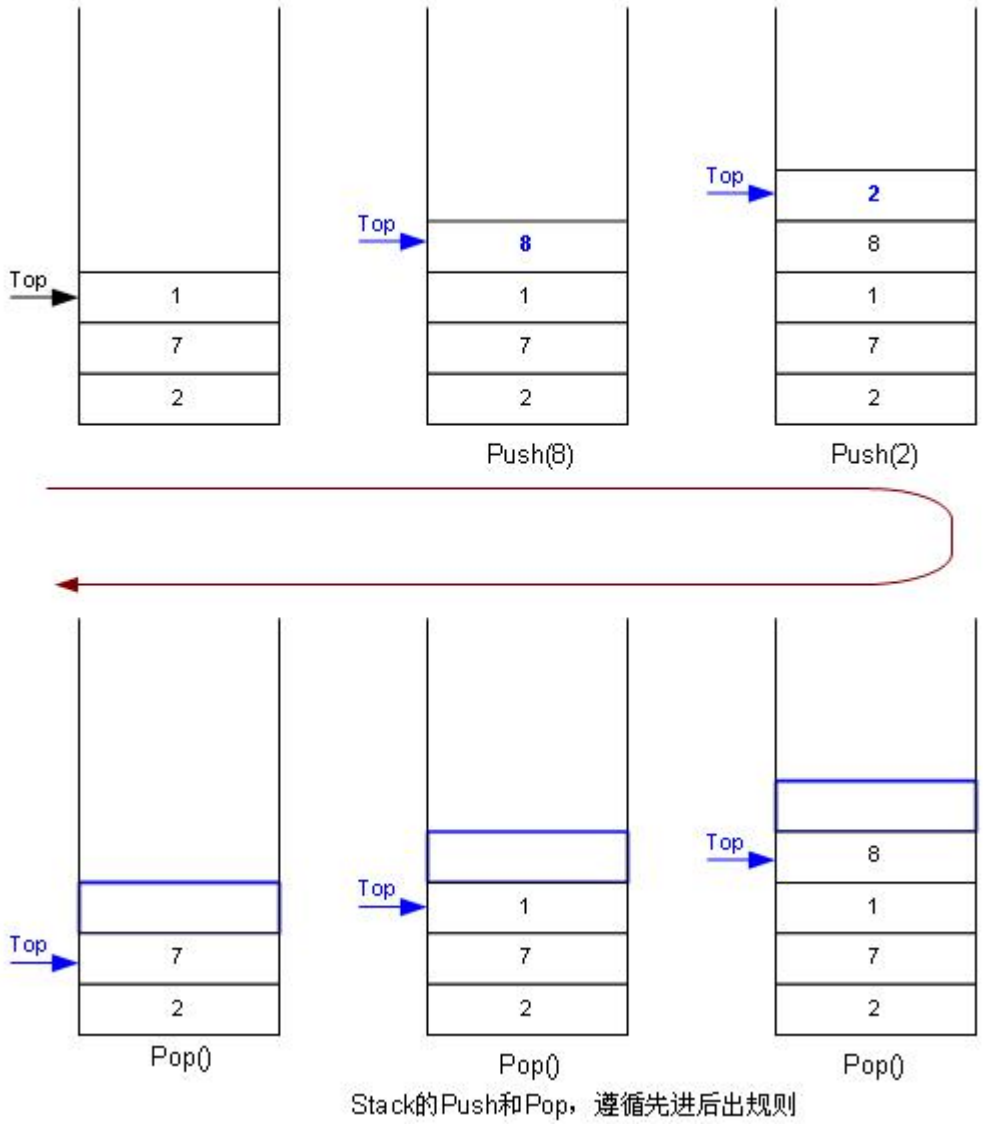


由于指针向后不向前，我们不知道要插入位置的前一个节点是什么，只能从头找，所以比较麻烦。

4、栈（Stack）

前一篇讲解了最基本的东西，这篇就稍微前进一点点，讲一下栈，栈在英文中叫 Stack，翻译成中文又叫“堆栈”，但决不能称为“堆”，这个要搞清楚，我们说的“栈”和“堆栈”指的都是 Stack 这种数据结构，但“堆”却是另外一个概念了，这里且不提。

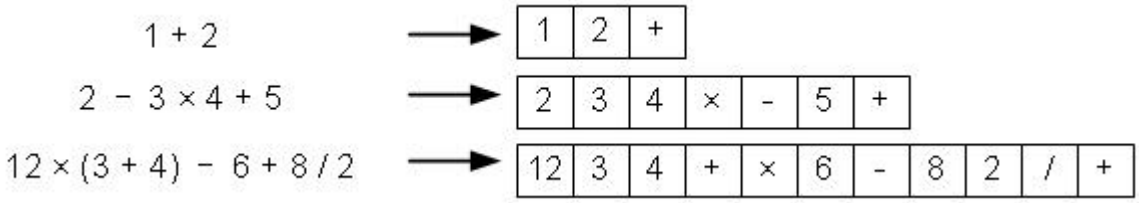
栈最大特点是先进后出，如图：



可以看出，栈有几个最常见的方法，或者说必备的方法，Push，Pop 和 Top，即进栈，出栈和取最顶元素。从代码上看，栈如何实现呢？用数组好还是用单向链表好呢？其实都可以，我下面的例子是用数组实现的。

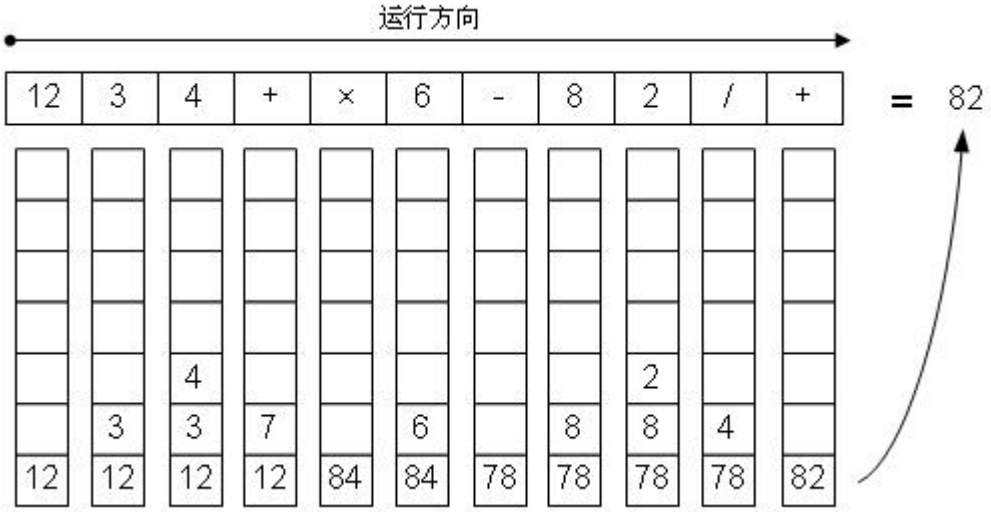
说了那么多，栈有什么用呢？下面就举一个最经典的例题——逆波兰表达式（RPN，Reversed Polish Notation）的求解。

什么是逆波兰表达式？我们表述一个算式通常是这样：X+Y，即：“操作数 1 操作符 操作数 2”，当然也有比较特别的，比如“sqrt(N)”，sqrt 是操作符，N 是操作数，而逆波兰表达式则很统一，先操作数，后操作符，为什么叫“逆波兰表达式”？因为有一个波兰人发明了波兰表达式，而逆的波兰表达式就叫“逆波兰表达式”了。看下图就能很好理解了：



所有的算式都可以用逆波兰表达式写出来，只是我这里的举例是为了方便起见，限制在整数的四则运算里。

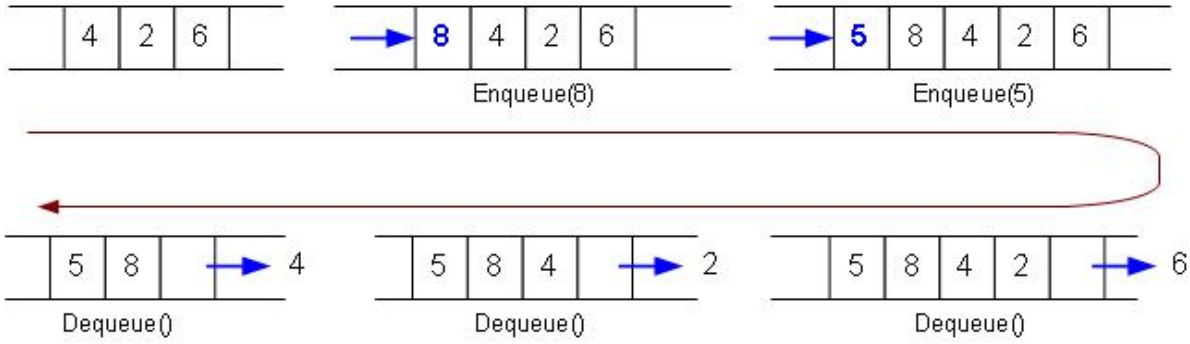
那假如现在我们有一个逆波兰表达式，那我们如何求出它的值呢？这里我们的“栈”就要派上用场了，由于操作数在操作符前面，所以我们按顺序遍历这个表达式，遇到操作数的时候进栈，遇到操作符时候让操作数出栈并运算，然后把运算结果进栈。过程如下图所示：



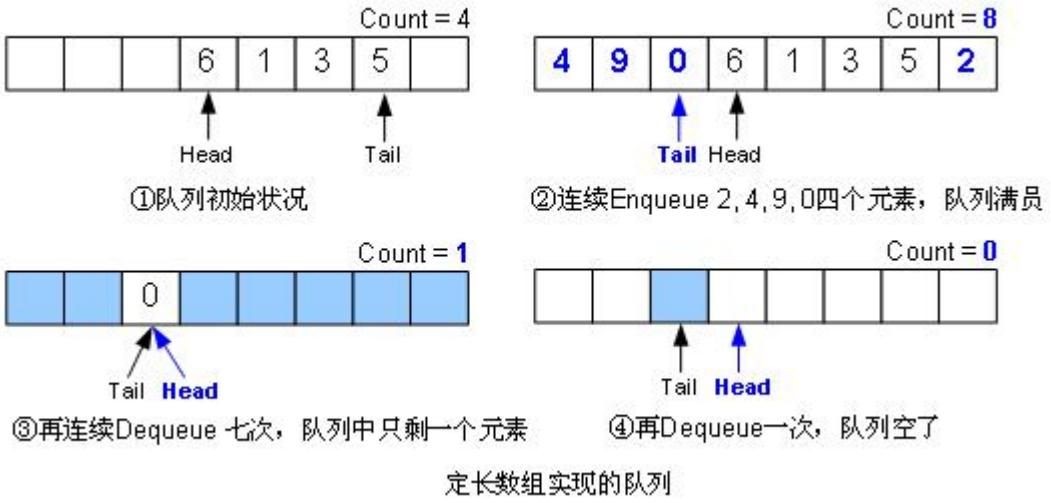
遇到第一个操作符，“+”的时候，由于需要两个操作数，所以出栈两次，4 和 3 出栈，执行加法运算，结果是 7，7 进栈……依此类推。

5. 队列（Queue）

前一篇讲了栈（Stack），队和栈其实只有一个差别，栈是先进后出，队是先进先出，如图：



从图中可以看出，队有两个常用的方法，Enqueue 和 Dequeue，顾名思义，就是进队和出队了。队和栈一样，既可以用数组实现，也可以用链表实现，我还是偏向于用数组，我的实现示意图如下：

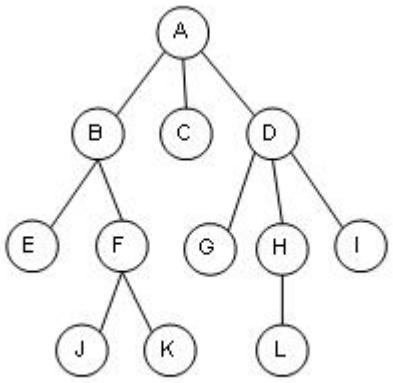


定长数组实现的队列

队有啥用呢？一个最常用的用途就是“buffer”，即缓冲区，比如有一批从网络来的数据，处理需要挺长的时间，而数据抵达的间隔并不均匀，有时快，有时慢，先来的先处理，后来的后处理，于是你创建了一个队，用来缓存这些数据，出队一笔，处理一笔，直到队列为空。当然队的作用远不止于此，下面的例子也是一个很经典的例子，希望读者能举一反三。

例子：使用队对树进行广度优先遍历。

广度优先区别于深度优先，即优先遍历最靠近根节点的各个节点：



广度优先，遍历顺序应该是：A B C D E F G H I J K L

我们的算法是：

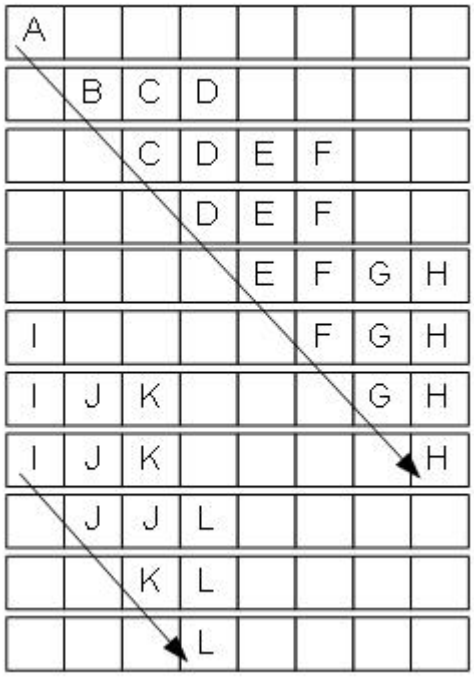
1，根节点入队

2，出队一个节点，算一次遍历，直到队列为空

3，将刚出队的节点的子节点入队

4，转到 2

队列的状况如下图：



树的遍历一般习惯使用递归，理论上所有的递归都可以转变为迭代，如何实现这个转变？队就是其中一种有效的办法。

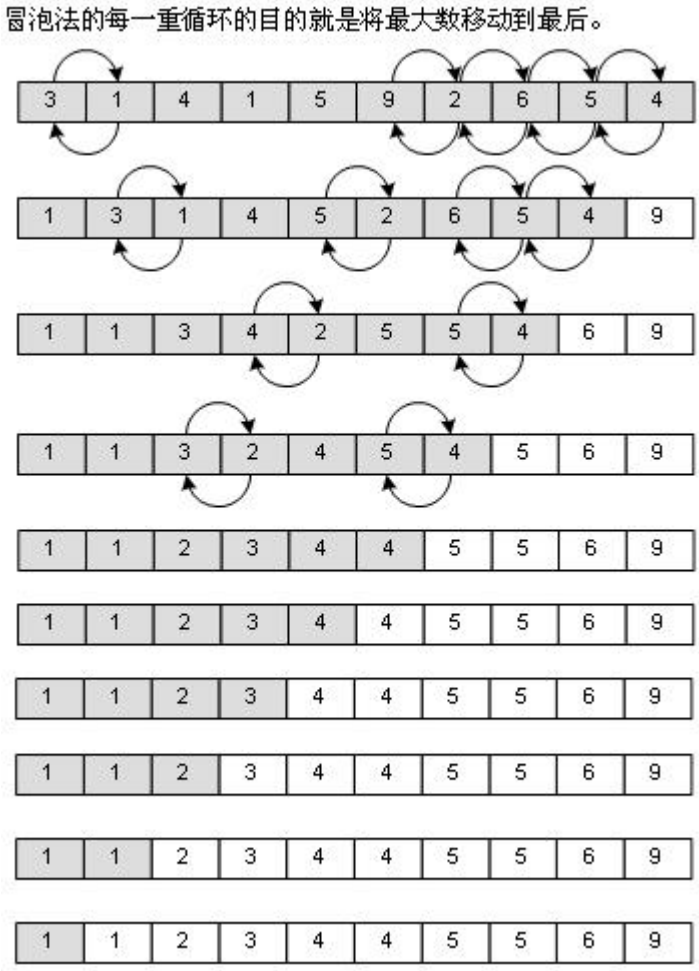
6、排序（Sort）

排序的考题，在各大公司的笔试里最喜欢出了，但我看多数考得都很简单，通常懂得冒泡排序就差不多了，确实，我在刚学数据机构时候，觉得冒泡排序真的很“精妙”，我怎么就想不出呢？呵呵，其实冒泡通常是效率最差的排序算法，差多少？请看本文，你一定不会后悔的。

1、冒泡排序（Bubbler Sort）

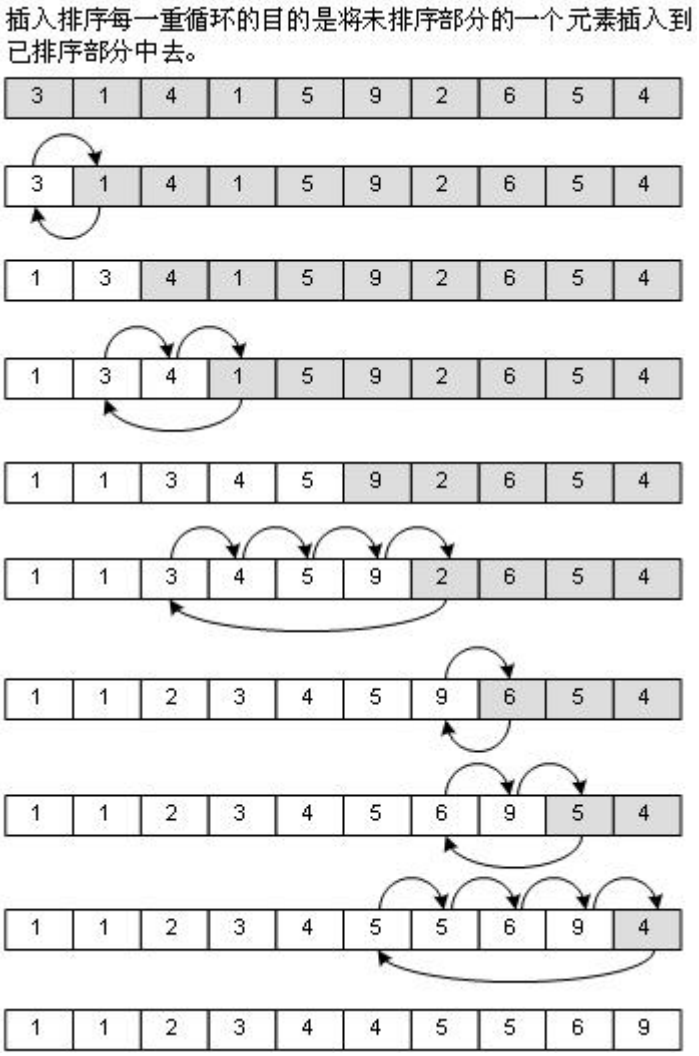
前面刚说了冒泡排序的坏话，但冒泡排序也有其优点，那就是好理解，稳定，再就是空间复杂度低，不需要额外开辟数组元素的临时保存控件，当然了，编写起来也容易。

其算法很简单，就是比较数组相邻的两个值，把大的像泡泡一样“冒”到数组后面去，一共要执行 N 的平方除以 2 这么多次的比较和交换的操作（N 为数组元素），其复杂度为 $O(n^2)$，如图：



2、直接插入排序（Straight Insertion Sort）

冒泡法对于已经排好序的部分（上图中，数组显示为白色底色的部分）是不再访问的，插入排序却要，因为它的方法就是从未排序的部分中取出一个元素，插入到已经排好序的部分去，插入的位置我是从后往前找的，这样可以使得如果数组本身是有序（顺序）的话，速度会非常之快，不过反过来，数组本身是逆序的话，速度也就非常之慢了，如图：

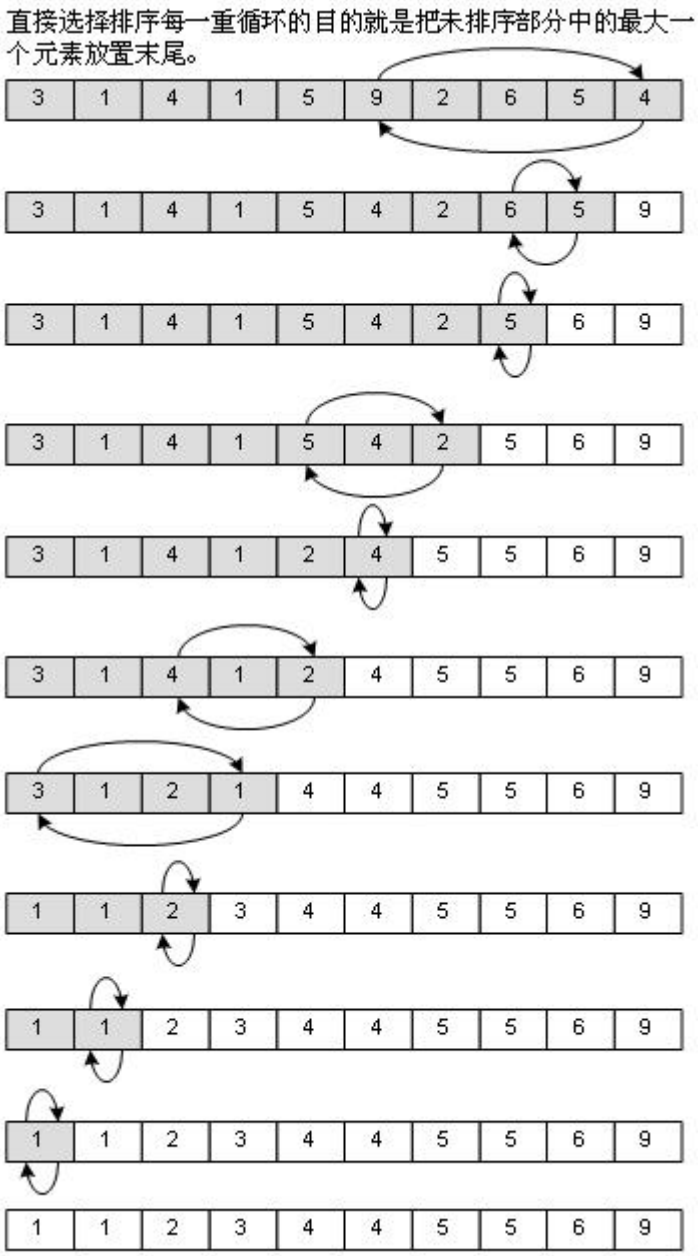


3、二分插入排序（Binary Insertion Sort）

这是对直接插入排序的改进，由于已排好序的部分是有序的，所以我们就能使用二分查找法确定我们的插入位置，而不是一个个找，除了这点，它跟插入排序没什么区别，至于二分查找法见我前面的文章（本系列文章的第四篇）。图跟上图没什么差别，差别在于插入位置的确定而已，性能却能因此得到不少改善。（性能分析后面会提到）

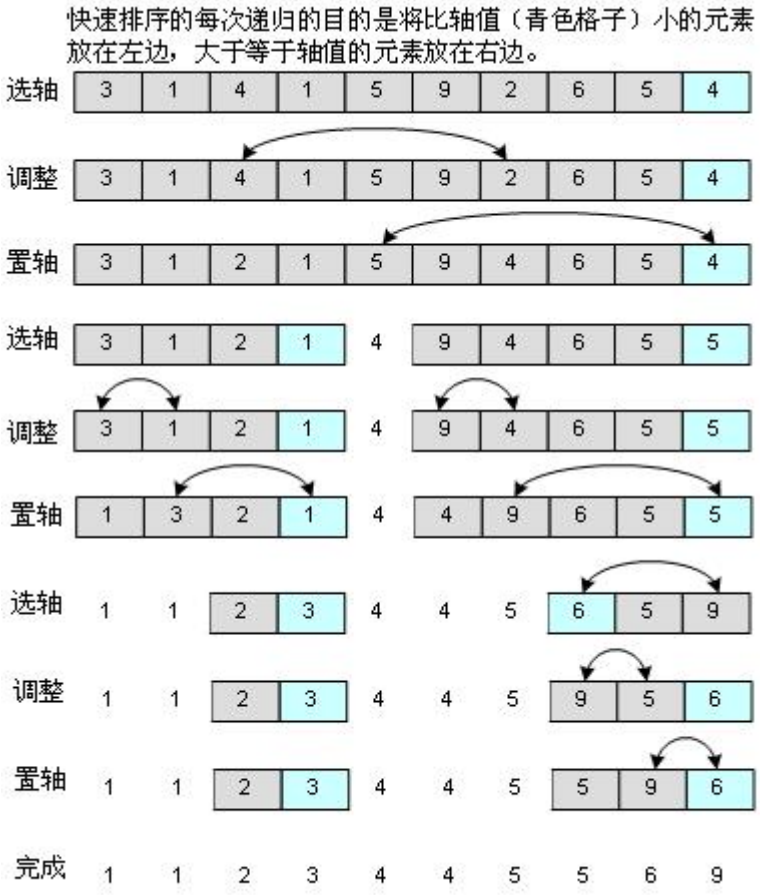
4、直接选择排序（Straight Selection Sort）

这是我在学数据结构前，自己能够想得出来的排序法，思路很简单，用打擂台的方式，找出最大的一个元素，和末尾的元素交换，然后再从头开始，查找第 1 个到第 N-1 个元素中最大的一个，和第 N-1 个元素交换……其实差不多就是冒泡法的思想，但整个过程中需要移动的元素比冒泡法要少，因此性能是比冒泡法优秀的。看图：



5、快速排序（Quick Sort）

快速排序是非常优秀的排序算法，初学者可能觉得有点难理解，其实它是一种“分而治之”的思想，把大的拆分为小的，小的再拆分为更小的，所以你一会儿从代码中就能很清楚地看到，用了递归。如图：



其中要选择一个轴值，这个轴值在理想的情况下就是中轴，中轴起的作用就是让其左边的元素比它小，它右边的元素不小于它。（我用了“不小于”而不是“大于”是考虑到元素数值会有重复的情况，在代码中也能看出来，如果把“>=”运算符换成“>”，将会出问题）当然，如果中轴选得不好，选了个最大元素或者最小元素，那情况就比较糟糕，我选轴值的办法是取出第一个元素，中间的元素和最后一个元素，然后从这三个元素中选中间值，这已经可以应付绝大多数情况。

6、改进型快速排序（Improved Quick Sort）

快速排序的缺点是使用了递归，如果数据量很大，大量的递归调用会不会导致性能下降呢？我想应该会的，所以我打算作这么种优化，考虑到数据量很小的情况下，直接选择排序和快速排序的性能相差无几，那当递归到子数组元素数目小于 30 的时候，我就是用直接选择排序，这样会不会提高一点性能呢？我后面分析。排序过程可以参考前面两个图，我就不另外画了。

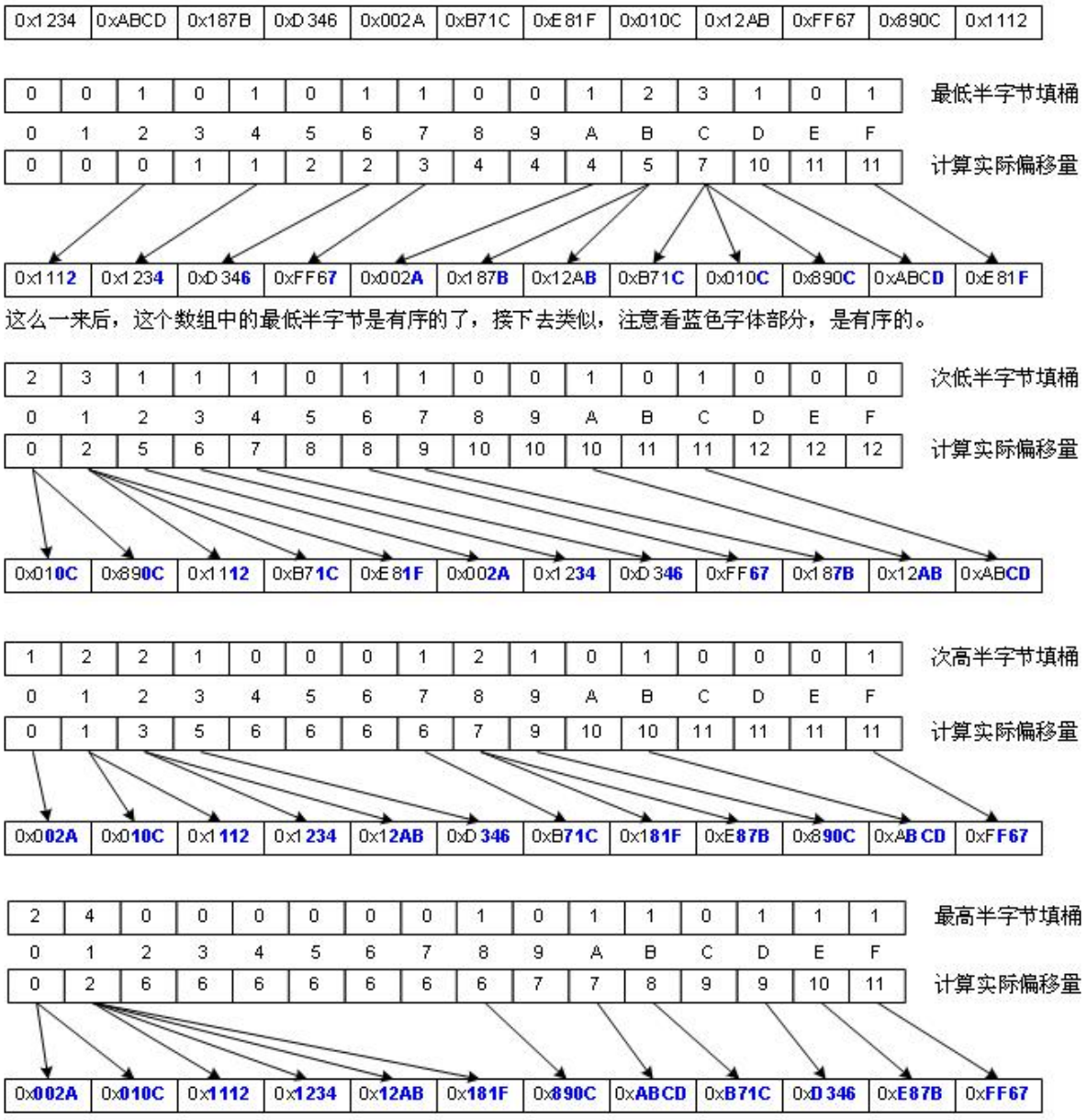
7、桶排序（Bucket Sort）

这是迄今为止最快的一种排序法，其时间复杂度仅为O(n)，也就是线性复杂度！不可思议吧？但它是有条件的。举个例子：一年的全国高考考生人数为 500 万，分数使用标准分，最低 100，最高 900，没有小数，你把这 500 万元素的数组排个序。我们抓住了这么个非常特殊的条件，就能在毫秒级内完成这 500 万的排序，那就是：最低 100，最高 900，没有小数，那一共可出现的分数可能有多少种呢？一共有 900-100+1=801，那么多种，想想看，有没有什么“投机取巧”的办法？方法就是创建 801 个“桶”，从头到尾遍历一次数组，对不同的分数给不同的“桶”加料，比如有个考生考了 500 分，那么就给 500 分的那个桶（下标为 500-100）加 1，完成后遍历一下这个桶数组，按照桶值，填充原数组，100 分的有 1000 人，于是从 0 填到 999，都填 1000，101 分的有 1200 人，于是从 1000 到 2019，都填入 101……如图：

很显然，如果分数不是从 100 到 900 的整数，而是从 0 到 2 亿，那就要分配 2 亿个桶了，这是不可能的，所以桶排序有其局限性，适合元素值集合并不大的情况。

8、基数排序（Radix Sort）

基数排序是对桶排序的一种改进，这种改进是让“桶排序”适合于更大的元素值集合的情况，而不是提高性能。它的思想是这样的，比如数值的集合是 8 位整数，我们很难创建一亿个桶，于是我们先对这些数的个位进行类似桶排序的排序（下文且称作“类桶排序”吧），然后再对这些数的十位进行类桶排序，再就是百位……一共做 8 次，当然，我说的是思路，实际上我们通常并不这么干，因为 C++的位移运算速度是比较快，所以我们通常以“字节”为单位进行桶排序。但下图为了画图方便，我是以半字节（4 bit）为单位进行类桶排序的，因为字节为单位进行桶排得画 256 个桶，有点难画，如图：



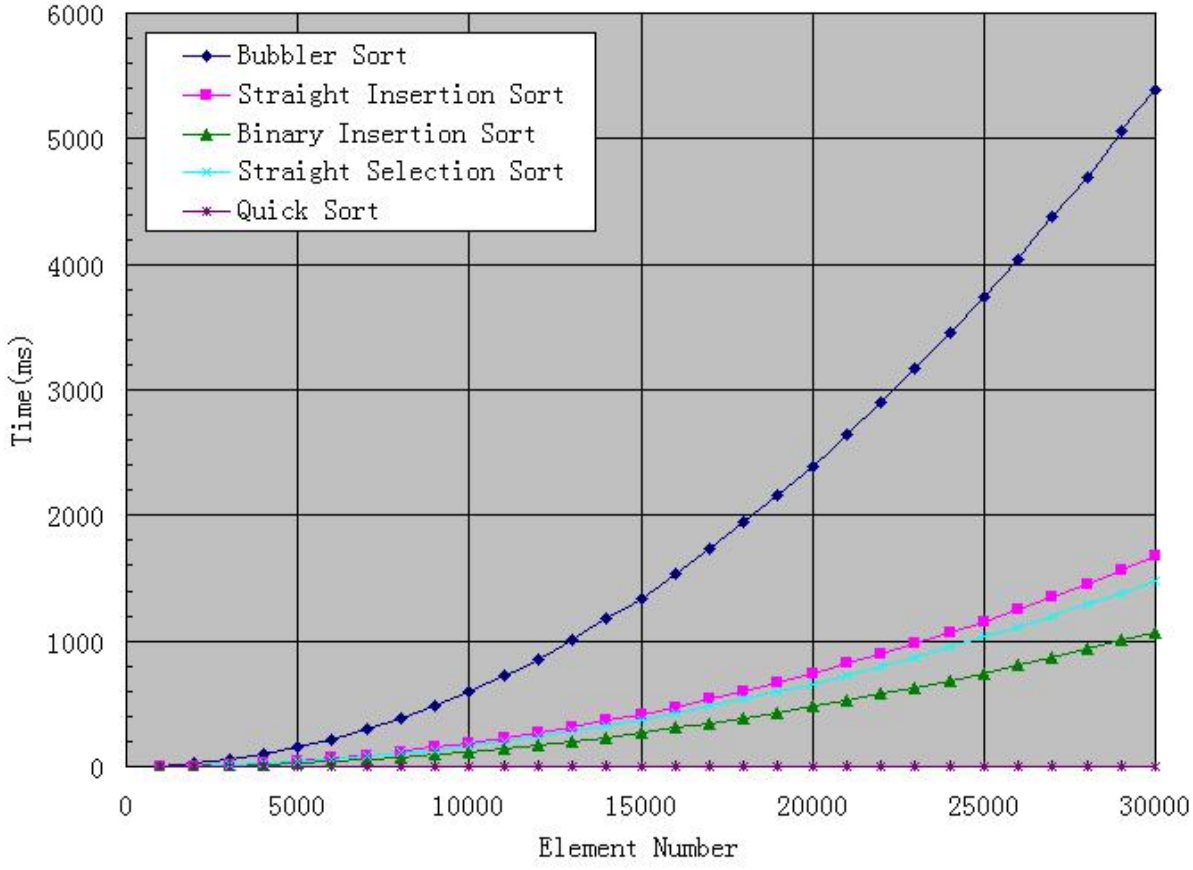
基数排序适合数值分布较广的情况，但由于需要额外分配一个跟原始数组一样大的暂存空间，它的处理也是有局限性的，对于元素数量巨大的原始数组而言，空间开销较大。性能上由于要多次“类桶排序”，所以不如桶排序。但它的复杂度跟桶排序一样，也是O(n)，虽然它用了多次循环，但却没有循环嵌套。

9、性能分析和总结

先不分析复杂度为O(n)的算法，因为速度太快，而且有些条件限制，我们先分析前六种算法，即：冒泡，直接插入，二分插入，直接选择，快速排序和改进型快速排序。

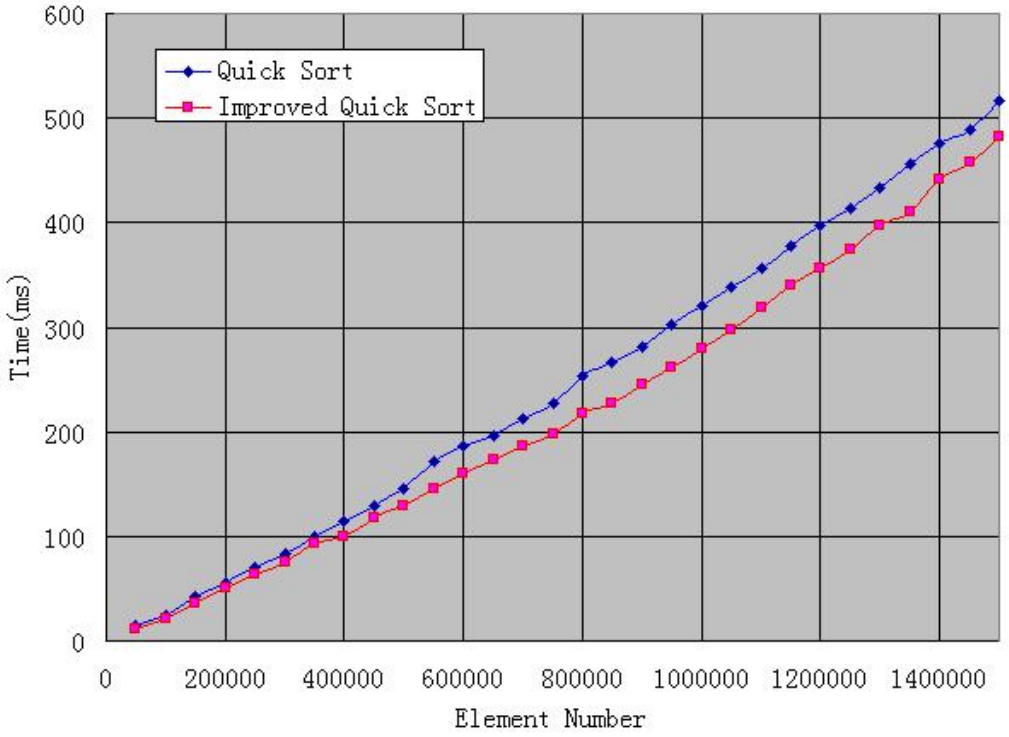
我的分析过程并不复杂，尝试产生一个随机数数组，数值范围是 0 到 7FFF，这正好可以用 C++的随机函数 rand()产生随机数来填充数组，然后尝试不同长度的数组，同一种长度的数组尝试 10 次，

以此得出平均值，避免过多波动，最后用 Excel 对结果进行分析，OK，上图了。



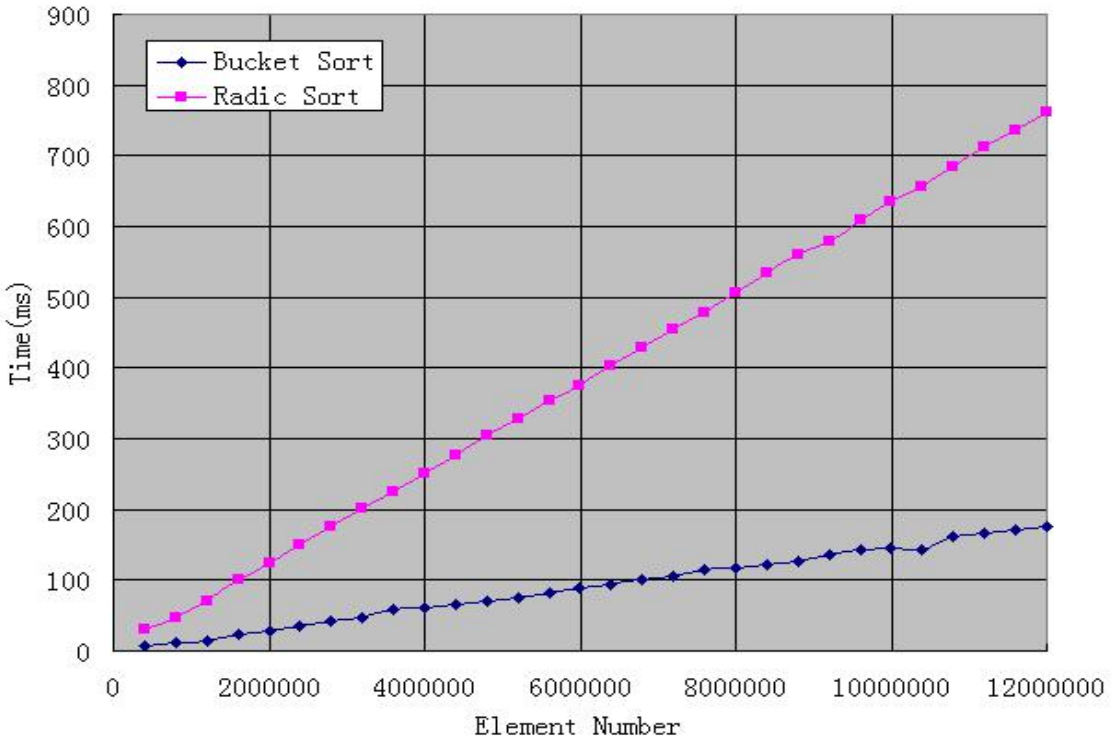
最差的一眼就看出来了，是冒泡，直接插入和直接选择旗鼓相当，但我更偏向于使用直接选择，因为思路简单，需要移动的元素相对较少，况且速度还稍微快一点呢，从图中看，二分插入的速度比直接插入有了较大的提升，但代码稍微长了一点点。

令人感到比较意外的是快速排序，3 万点以内的快速排序所消耗的时间几乎可以忽略不计，速度之快，令人振奋，而改进型快速排序的线跟快速排序重合，因此不画出来。看来要对快速排序进行单独分析，我加大了数组元素的数目，从 5 万到 150 万，画出下图：



可以看到，即便到了 150 万点，两种快速排序也仅需差不多半秒钟就完成了，实在快，改进型快速排序性能确实有微略提高，但并不明显，从图中也能看出来，是不是我设置的最小快速排序元素数目不太合适？但我尝试了好几个值都相差无几。

最后看线性复杂度的排序，速度非常惊人，我从 40 万测试到 1200 万，结果如图：



可见稍微调整下算法，速度可以得到质的飞升，而不是我们以前所认为的那样：再快也不会比冒泡法快多少啊？

最后制作一张表，比较一下这些排序法：

	时间复杂度	速度	额外空间消耗	其它特点
冒泡法 Bubbler	O(n^2)	★	基本没有	代码简单，易理解，但速度太慢，不推荐使用。
直接插入法 Straight Insertion	O(n^2)	★★	基本没有	如果数组本身有序，那效率非常高，但如果数组逆序，效率就非常低了。
二分插入法 Binary Insertion	O(n^2)	★★☆	基本没有	对直接插入法的改进，是O(n^2)时间复杂度排序算法中性能最好的。
直接选择法 Straight Selection	O(n^2)	★★	基本没有	代码简单并不比冒泡法复杂，性能却不错，可取代冒泡法。推荐使用。
快速排序法 Quick	O(nlogn)	★★★☆	递归，栈空间占用	性能非常好，但如果环境不支持递归，就很难用，需要额外消耗一些空间。
改进型快速排序法 Improved Quick	O(nlogn)	★★★☆	(同上)	除了快速排序的特点之外，能减少一些递归的调用，防止栈溢出。
桶排序 Bucket	O(n)	★★★★★	桶空间占用	元素值集合小的情况下适用，性能最为优秀。
基数排序 Radix	O(n)	★★★★☆	桶空间占用 临时数组空间占用	元素数目不是太多的情况下适用，性能非常好。

补充：

简化版快速排序

快速排序是 C.R.A.Hoare 于 1962 年提出的一种划分交换排序。它采用了一种分治的策略，通常称其为分治法(Divide-and-ConquerMethod)。

该方法的基本思想是：

1．先从数列中取出一个数作为基准数。

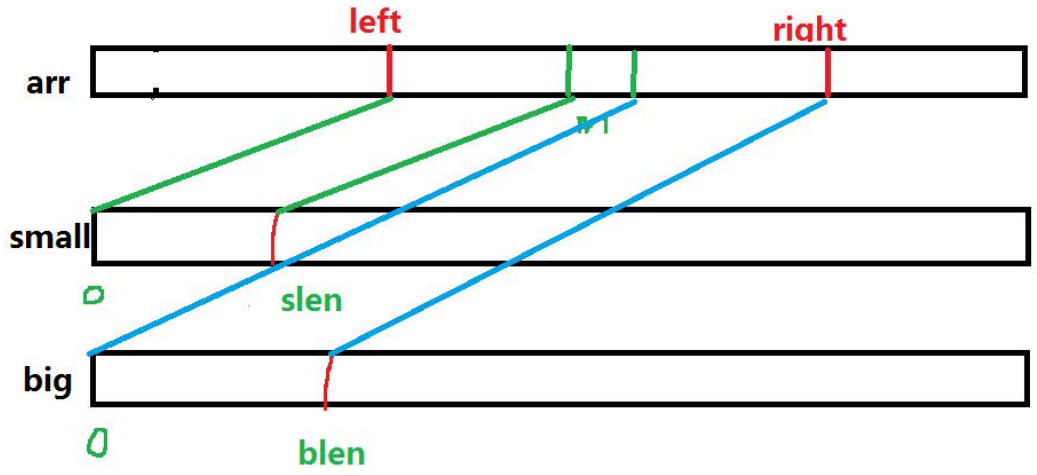
2．分区过程，将比这个数大的数全放到它的右边，小于或等于它的数全放到它的左边。

3．再对左右区间重复第二步，直到各区间只有一个数。

如图所示

	small										
	big										
6	arr	6	2	1	7	9	4	3	5	10	8
		left									right
1、选择中间元素 int middle = arr[left]											
2、根据中间元素分割为两部分，临时保存small、big							si				
	small	2	1	4	3	5					
	big	7	9	10	8						
						bi					
3、将small、middle、big的回写到arr相应位置											
	arr	2	1	4	3	5	6	7	9	10	8
		left					m				right
4、将arr[left...m-1]看成一个数组，递归											
5、将arr[m+1...right]看成一个数组，递归											

注意往 arr 回拷贝的时候下标的变化



归并排序

归并（Merge）排序法是将两个（或两个以上）有序表合并成一个新的有序表，即把待排序序列分为若干个子序列，每个子序列是有序的。然后再把有序子序列合并为整体有序序列。

按照我们常规的方法，我们每次从两个列表开头元素选取较小的一个，直到某一个列表到达底部，再将另一个剩下部分顺序取出，例如

1 3 5 7

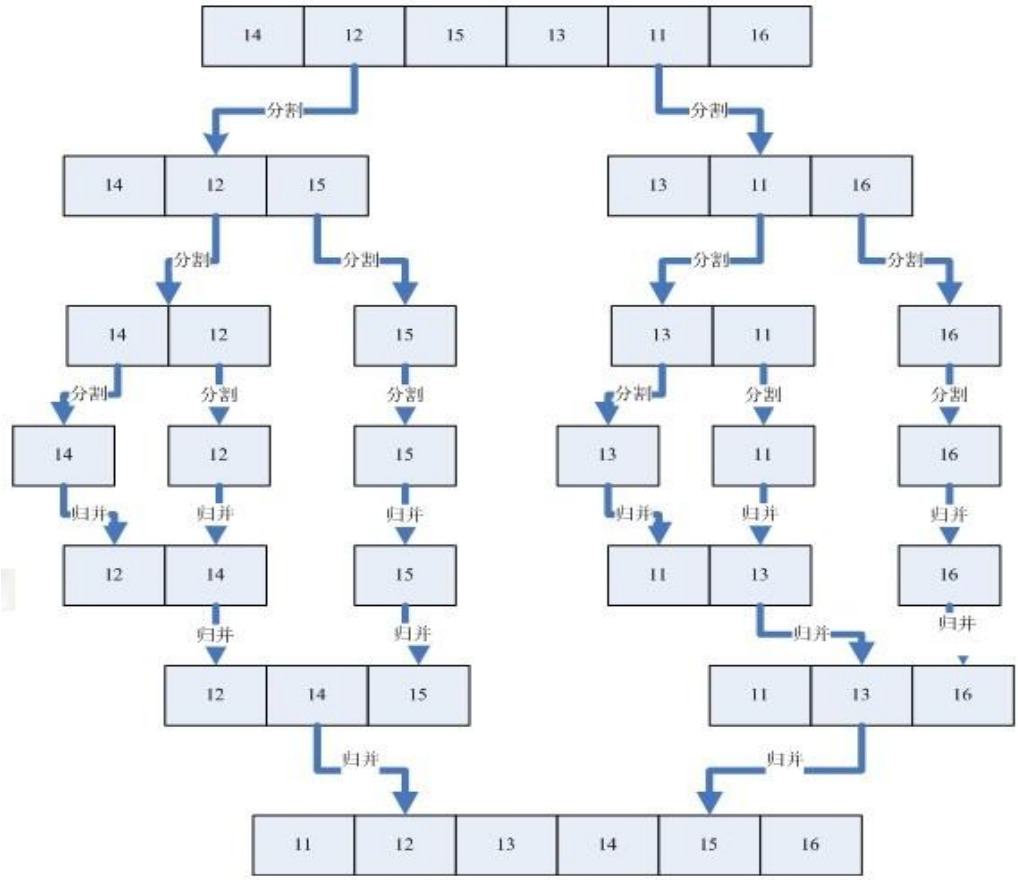
2 4 6 8 10 12 14

该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。

或者说将 n 个元素的序列划分为两个序列，再将两个序列划分为 4 个序列，直到每个序列只有一个元素，最后，再将两个有序序列归并成一个有序的序列。

通过直观图分析归并排序过程

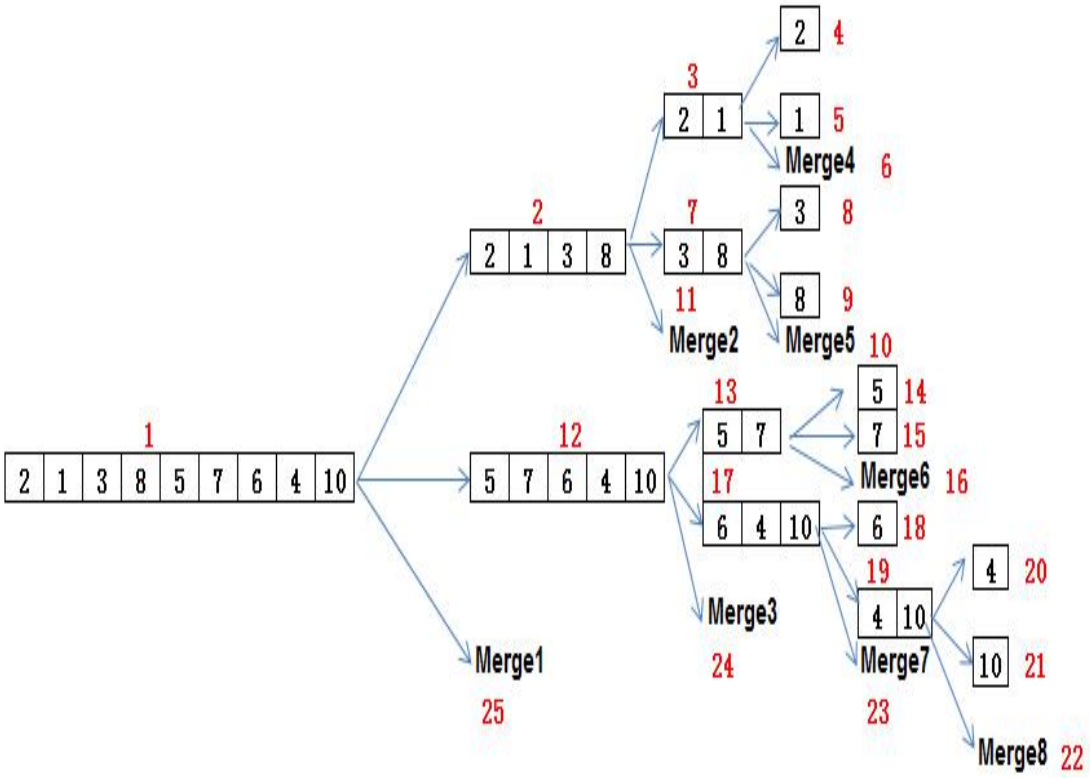
假设我们有一个没有排好序的序列，那么首先我们使用分割的办法将这个序列分割成一个一个已经排好序的子序列，然后再利用归并的方法将一个个的子序列合并成排序好的序列。分割和归并的过程可以看下面的图例： 14 12 15 13 11 16



从上图可以看出，我们首先把一个未排序的序列从中间分割成 2 部分，再把 2 部分分成 4 部分，依次分割下去，直到分割成一个一个的数据，再把这些数据两两归并到一起，使之有序，不停的归并，最后成为一个排好序的序列。

归并执行过程分析

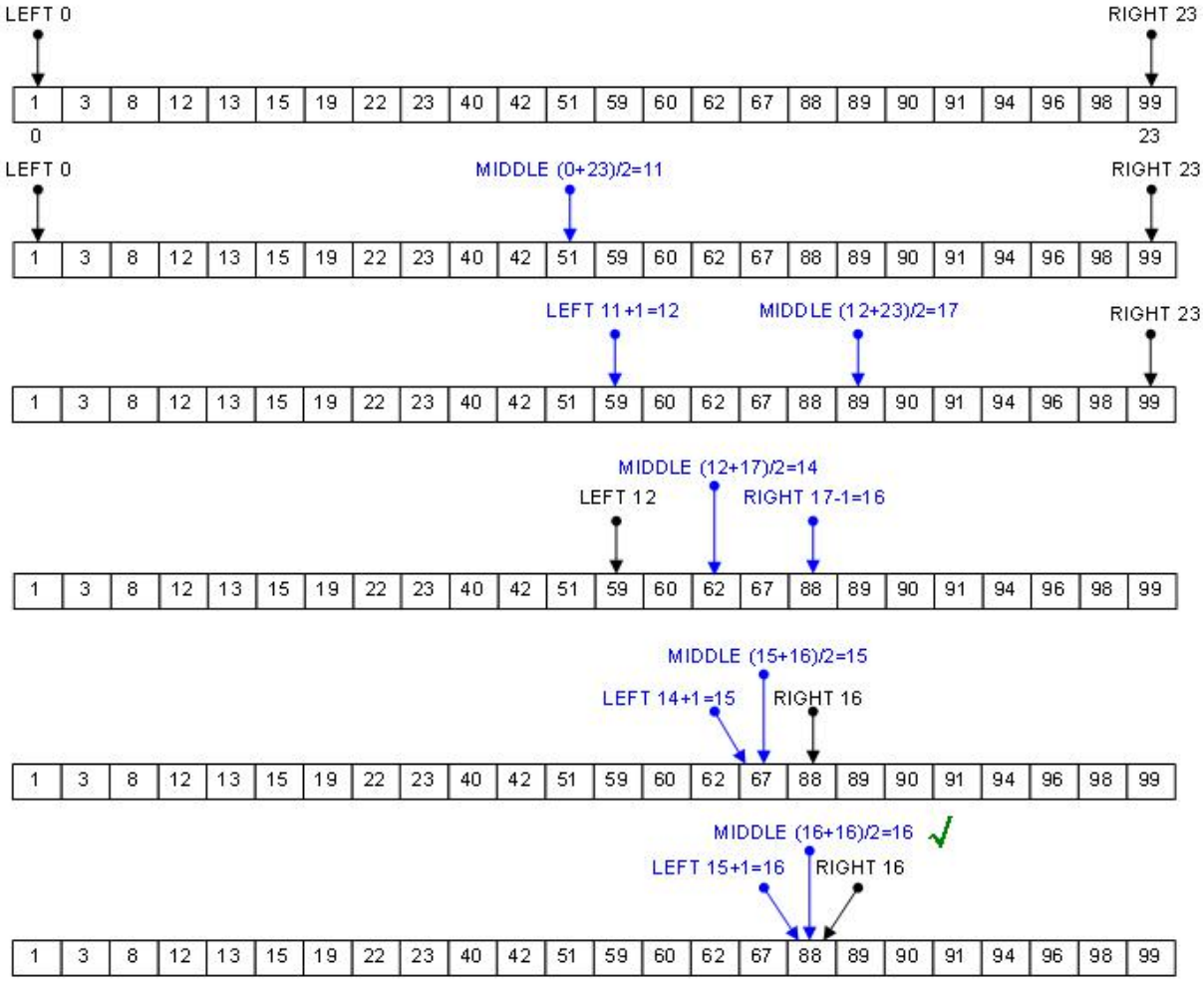
对于原始的数组 2,1,3,8,5,7,6,4,10，在整个过程执行的是顺序是途中红色编号 1-20。虽然我们描述中说的是程序先分解，再归并，但实际过程是一边分解一边归并，前半部分分先排好序，后半部分再拍好，最后整个归并为一个完整的序列



1	2	3	8	5	7	6	4	10	Merge4
1	2	3	8	5	7	6	4	10	Merge5
1	2	3	8	5	7	6	4	10	Merge2
1	2	3	8	5	7	6	4	10	Merge6
1	2	3	8	5	7	6	4	10	Merge8
1	2	3	8	5	7	4	7	10	Merge7
1	2	3	8	4	5	7	7	10	Merge3
1	2	3	4	5	7	7	8	10	Merge1

7.二分法查找（Binary Search）

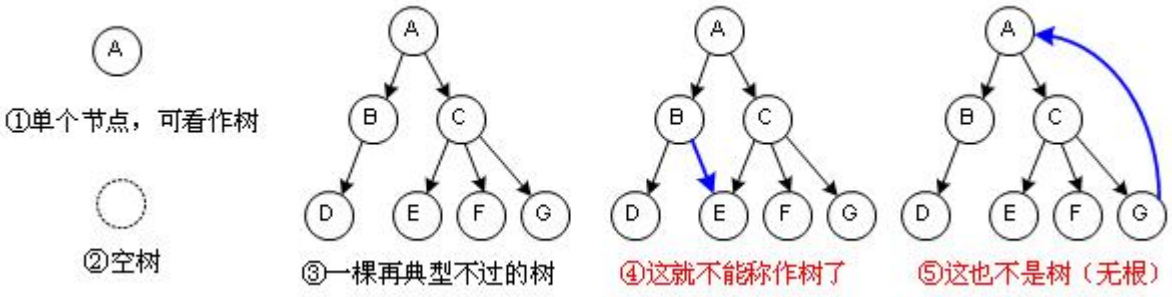
如何从数组里找一个元素的位置？如果排列是无序的，我们只能从头到尾找，但如果排列是有序的，我们则可以用别的更好的方法，二分查找法就类似我们在英汉词典里找一个单词的方法。如下图所示（假如我们要查找的数字是“88”）：



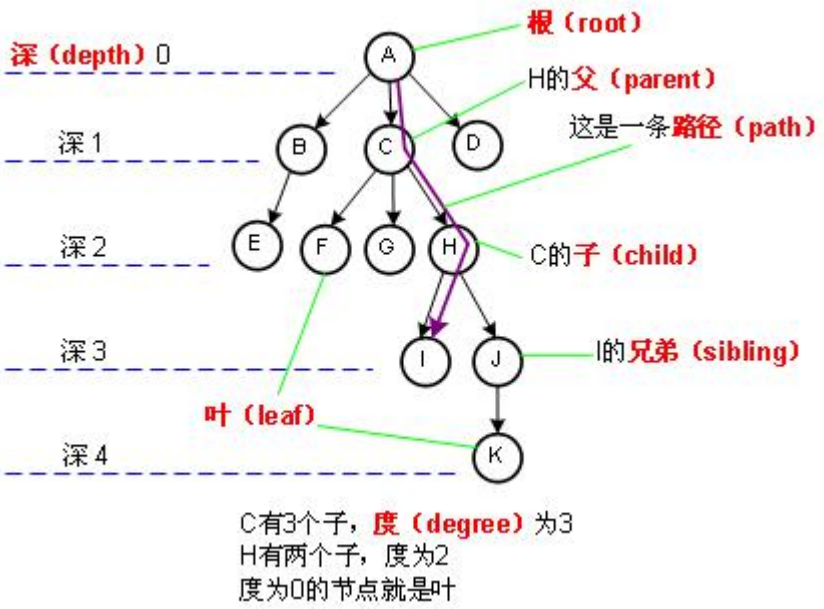
二分查找法比顺序查找快多少，为了方便起见，初始化有序表的时候填入的数字都是均匀的，而事实上数字可以不均匀。你可以调整一下代码中 TABLE_SIZE 的值，从 500，调到 5000，再调到 10000，再调到 30000……你会发觉两者差距越来越明显。我在第一篇的地方提到二分查找法的复杂度为 O (logn)，而顺序查找的复杂度为 O (n)，当 n 越来越大时候， O (logn)的优势也就越来越明显，当然了，前提是“有序”，才可用二分查找法。

8、树（Tree）

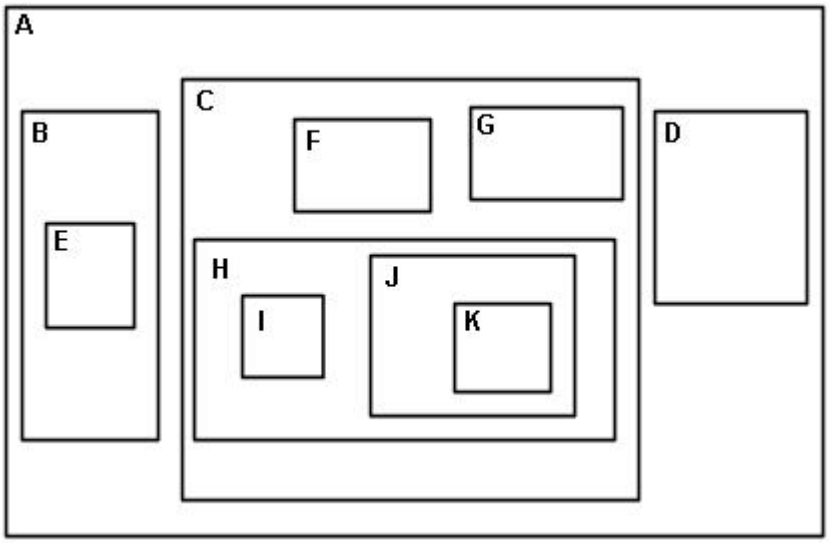
树，顾名思义，长得像一棵树，不过通常我们画成一棵倒过来的树，根在上，叶在下。不说那么多了，图一看就懂：



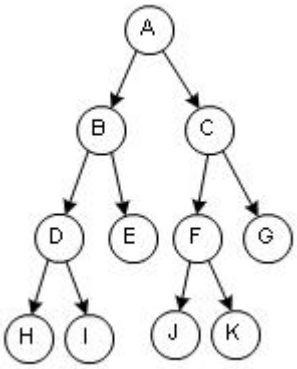
当然了，引入了树之后，就不得不引入树的一些概念，这些概念我照样尽量用图，谁会记那么多文字？



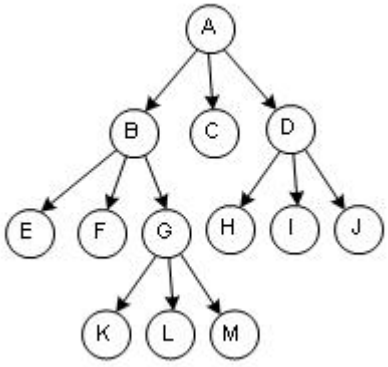
树这种结构还可以表示成下面这种方式，可见树用来描述包含关系是很不错的，但这种包含关系不得出现交叉重叠区域，否则就不能用树描述了，看图：



面试的时候我们经常被考到的是一种叫“二叉树”的结构，二叉树当然也是树的一种了，它的特点是除了叶以外的节点都有两个子，图：



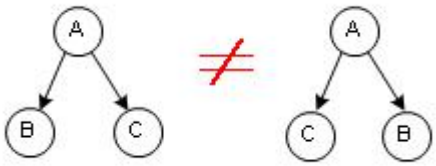
由此我们还可以推出“三叉树”：



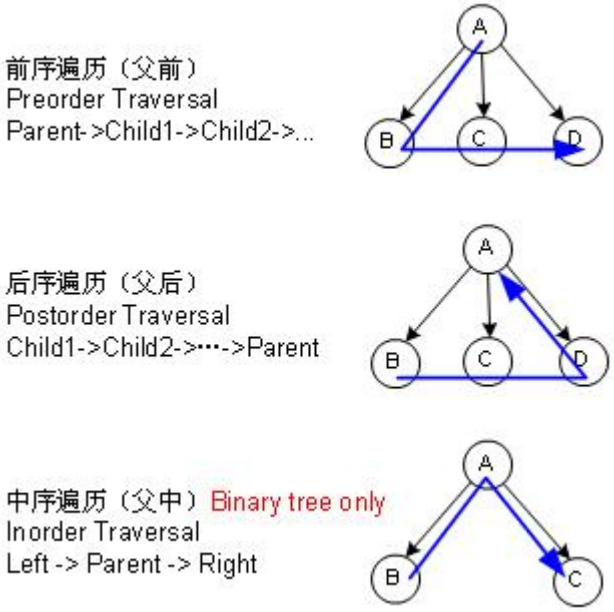
当然还有“四叉树”，“五叉树”，“六叉树”……但太难画了，节点太多，略过。

树的遍历（Traversal）

值得再提一下的是二叉树，因为它确实比较特别，节点有两个子，这两个子是有左右之分的，颠倒一下左右，就是不一样的二叉树了，所以左右是不能随便颠倒的。



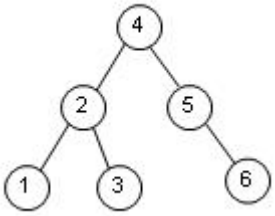
在第三篇讲到“队”的时候，提及到了广度优先遍历（Breadth-first traversal），除了广度优先遍历之外，还有深度优先遍历（Depth-first Traversal），深度优先遍历又可分为：前序遍历（Preorder Traversal），后序遍历（Postorder Traversal）和中序遍历（Inorder Traversal），其中中序遍历只有对二叉树才有意义，下图解释这几种遍历：



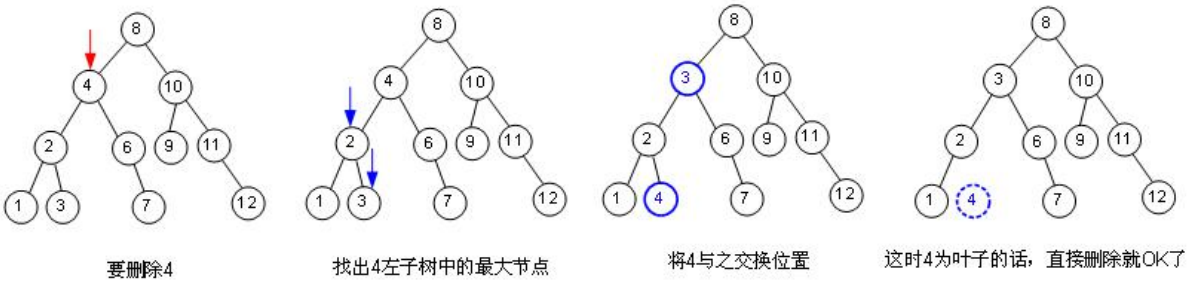
好了，又到代码阶段，写点代码。我看过许多数据结构的教材，二叉树遍历都是必不可少的内容，而且我知道的全部都是用递归实现的，现在，我要求你不用递归，实现对二叉树的中序遍历。怎么办？我给个提示：广度优先遍历时候我们用了队，中序遍历，我们使用*栈*。

二叉查找树（BST）

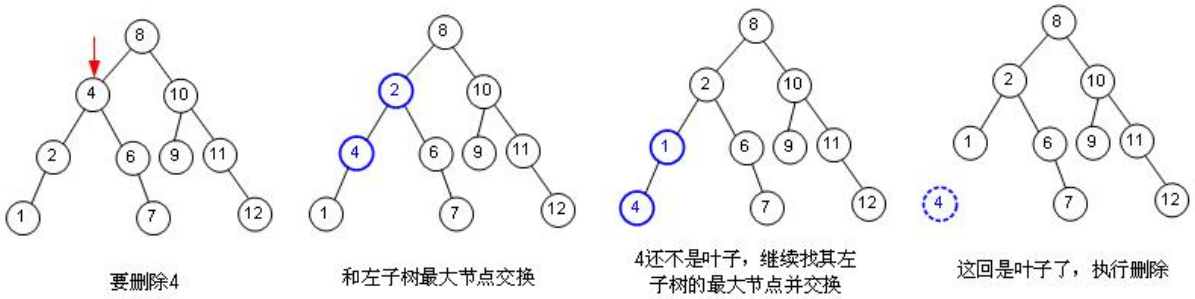
前一篇介绍了树，却未介绍树有什么用。但就算我不说，你也能想得到，看我们 Windows 的目录结构，其实就是树形的，一个典型的分类应用。当然除了分类，树还有别的作用，我们可以利用树建立一个非常便于查找取值又非常便于插入删除的数据结构，这就是马上要提到的二叉查找树（Binary Search Tree），这种二叉树有个特点：对任意节点而言，左子（当然了，存在的话）的值总是小于本身，而右子（存在的话）的值总是大于本身。



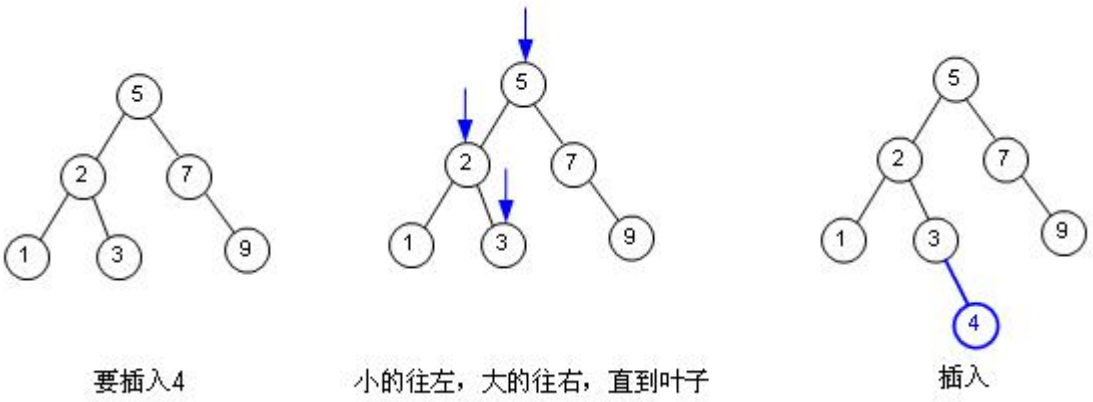
这种特性使得我们要查找其中的某个值都很容易，从根开始，小的往左找，大的往右找，不大不小的就是这个节点了；插入一样的道理，从根开始，小的往左，大的往右，直到叶子，就插入，算法比较简单，不一一列了，它们的时间复杂度期望为 O(logn)。（为什么是“期望”，后面会讲）



删除则稍微麻烦点，因为我们删的不一定是叶子，如果只是叶子，那就好办，如果不是呢？我们最通常的做法就是把这个节点往下挪，直到它变为叶子为止，看图。



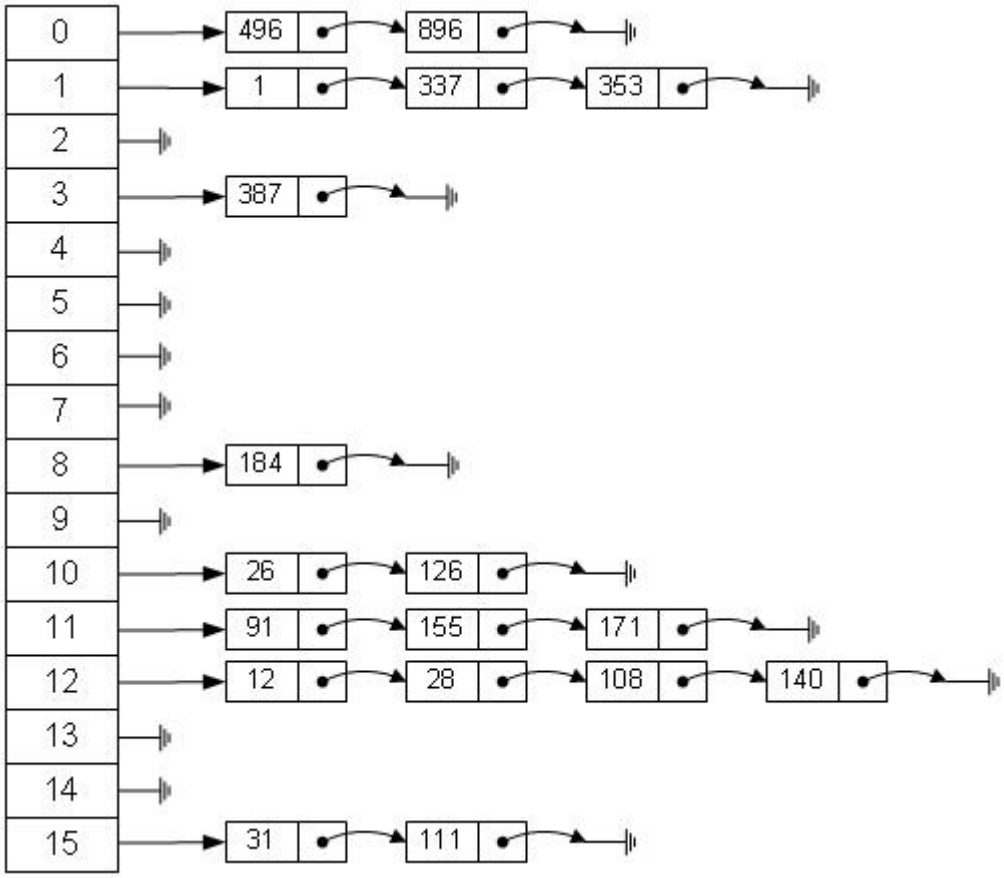
也许你要问，如果和左子树最大节点交换后，要删除的节点依然不是叶子，那怎么办呢？那继续呗，看图：



那左子树不存在的情况下呢？你可以查找右子树的最小节点，和上面是类似的，图我就不画了。

9、哈希表（Hash Table）及散列法（Hashing）

数组的特点是：寻址容易，插入和删除困难；而链表的特点是：寻址困难，插入和删除容易。那么我们能不能综合两者的特性，做出一种寻址容易，插入删除也容易的数据结构？答案是肯定的，这就是我们要提起的哈希表，哈希表有多种不同的实现方法，我接下来解释的是最常用的一种方法——拉链法，我们可以理解为“链表的数组”，如图：



左边很明显是个数组，数组的每个成员包括一个指针，指向一个链表的头，当然这个链表可能为空，也可能元素很多。我们根据元素的一些特征把元素分配到不同的链表中去，也是根据这些特征，找到正确的链表，再从链表中找出这个元素。

元素特征转变为数组下标的方法就是散列法。散列法当然不止一种，我下面列出三种比较常用的。

1，除法散列法

最直观的一种，上图使用的就是这种散列法，公式：

index = value % 16

学过汇编的都知道，求模数其实是通过一个除法运算得到的，所以叫“除法散列法”。

2，平方散列法

求 index 是非常频繁的操作，而乘法的运算要比除法来得省时（对现在的 CPU 来说，估计我们感觉不出来），所以我们考虑把除法换成乘法和一个位移操作。公式：

index = (value * value) >> 28

如果数值分配比较均匀的话这种方法能得到不错的结果，但我上面画的那个图的各个元素的值算出来的 index 都是 0——非常失败。也许你还有个问题，value 如果很大，value * value 不会溢出吗？答案是会的，但我们这个乘法不关心溢出，因为我们根本不是为了获取相乘结果，而是为了获取 index。

3，斐波那契（Fibonacci）散列法

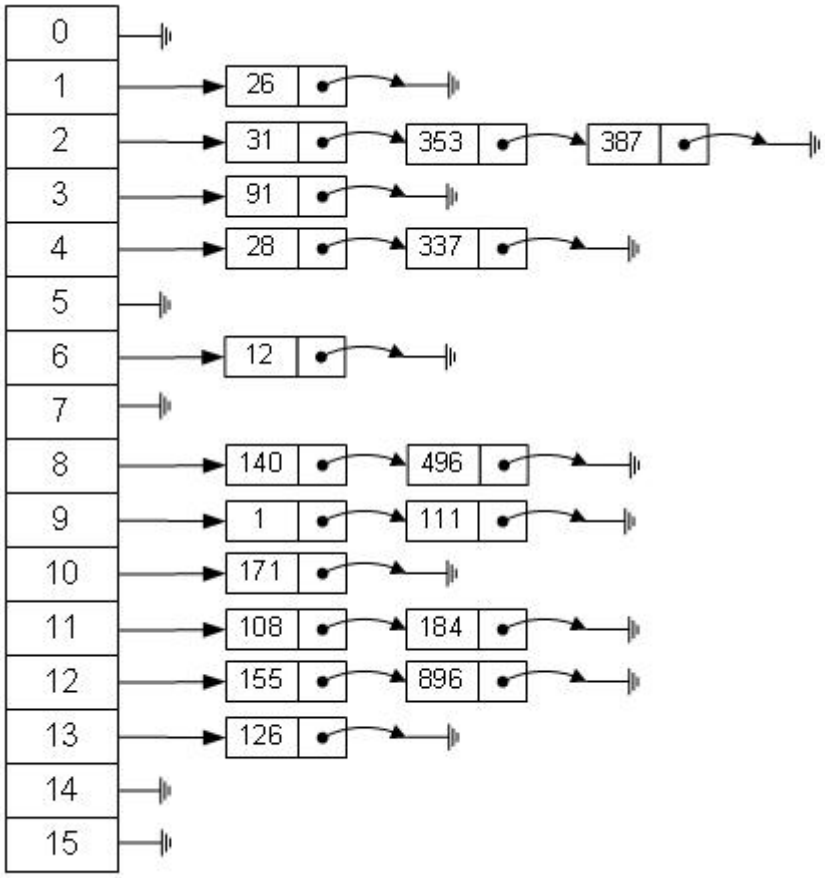
平方散列法的缺点是显而易见的，所以我们能不能找出一个理想的乘数，而不是拿 value 本身当作乘数呢？答案是肯定的。

1，对于 16 位整数而言，这个乘数是 40503
2，对于 32 位整数而言，这个乘数是 2654435769
3，对于 64 位整数而言，这个乘数是 11400714819323198485

这几个“理想乘数”是如何得出来的呢？这跟一个法则有关，叫黄金分割法则，而描述黄金分割法则的最经典表达式无疑就是著名的斐波那契数列，如果你还有兴趣，就到网上查找一下“斐波那契数列”等关键字，我数学水平有限，不知道怎么描述清楚为什么，另外斐波那契数列的值居然和太阳系八大行星的轨道半径的比例出奇吻合，很神奇，对么？

对我们常见的 32 位整数而言，公式：
index = (value * 2654435769) >> 28

如果用这种斐波那契散列法的话，那我上面的图就变成这样了：



用斐波那契散列法重新调整过的哈希表

看起来不错，以后就用斐波那契散列法吧。