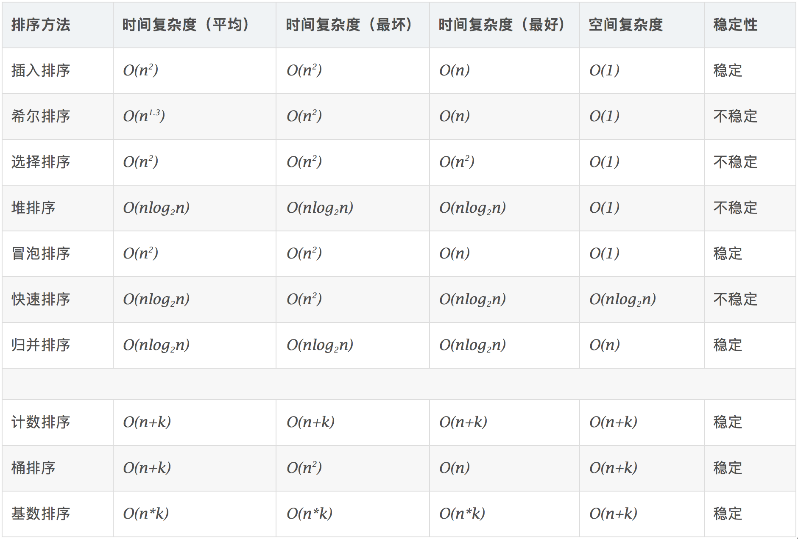
<https://www.cnblogs.com/onepixel/articles/7674659.html>



# 排序算法的比较

# 插入排序和冒泡排序的区别

1、冒泡排序得到一个元素的位置的先后是由它在新序列中的位置决定的，插入排序是由它在旧序列中的位置决定的。

2、冒泡有个小优化，就是某次循环没有发生交换就说明已经是有序的了。但是复杂度两个都是o(n^2)的，区别不大。

1.稳定性比较

插入排序、冒泡排序、二叉树排序、二路归并排序及其他线形排序是稳定的

选择排序、希尔排序、快速排序、堆排序是不稳定的

2.时间复杂性比较

插入排序、冒泡排序、选择排序的时间复杂性为O(n2)

其它非线形排序的时间复杂性为O(nlog2n)

线形排序的时间复杂性为O(n);

3.辅助空间的比较

线形排序、二路归并排序的辅助空间为O(n),其它排序的辅助空间为O(1);

4.其它比较

插入、冒泡排序的速度较慢，但参加排序的序列局部或整体有序时，这种排序能达到较快的速度。 反而在这种情况下，快速排序反而慢了。

当n较小时，对稳定性不作要求时宜用选择排序，对稳定性有要求时宜用插入或冒泡排序。

若待排序的记录的关键字在一个明显有限范围内时,且空间允许是用桶排序。

当n较大时，关键字元素比较随机，对稳定性没要求宜用快速排序。

当n较大时，关键字元素可能出现本身是有序的，对稳定性有要求时，空间允许的情况下。宜用归并排序。

当n较大时，关键字元素可能出现本身是有序的，对稳定性没有要求时宜用堆排序。

# 如何实现链表快排

# 快排：

## 为什么堆排序比快排慢

回顾一下堆排的过程：

1. 建立最大堆（堆顶的元素大于其两个儿子，两个儿子又分别大于它们各自下属的两个儿子... 以此类推）

2. 将堆顶的元素和最后一个元素对调（相当于将堆顶元素（最大值）拿走，然后将堆底的那个元素补上它的空缺），然后让那最后一个元素从顶上往下滑到恰当的位置（重新使堆最大化）。

3. 重复第2步。

这里的关键问题就在于第2步，堆底的元素肯定很小，将它拿到堆顶和原本属于最大元素的两个子节点比较，它比它们大的可能性是微乎其微的。实际上它肯定小于其中的一个儿子。而大于另一个儿子的可能性非常小。于是，这一次比较的结果就是概率不均等的，根据前面的分析，概率不均等的比较是不明智的，因为它并不能保证在糟糕情况下也能将问题的可能性削减到原本的1/2。可以想像一种极端情况，如果a肯定小于b，那么比较a和b就会什么信息也得不到——原本剩下多少可能性还是剩下多少可能性。

在堆排里面有大量这种近乎无效的比较，因为被拿到堆顶的那个元素几乎肯定是很小的，而靠近堆顶的元素又几乎肯定是很大的，将一个很小的数和一个很大的数比较，结果几乎肯定是“小于”的，这就意味着问题的可能性只被排除掉了很小一部分。

这就是为什么堆排比较慢（堆排虽然和快排一样复杂度都是O(NlogN)但堆排复杂度的常系数更大）。

MacKay也提供了一个修改版的堆排：每次不是将堆底的元素拿到上面去，而是直接比较堆顶（最大）元素的两个儿子，即选出次大的元素。由于这两个儿子之间的大小关系是很不确定的，两者都很大，说不好哪个更大哪个更小，所以这次比较的两个结果就是概率均等的了。

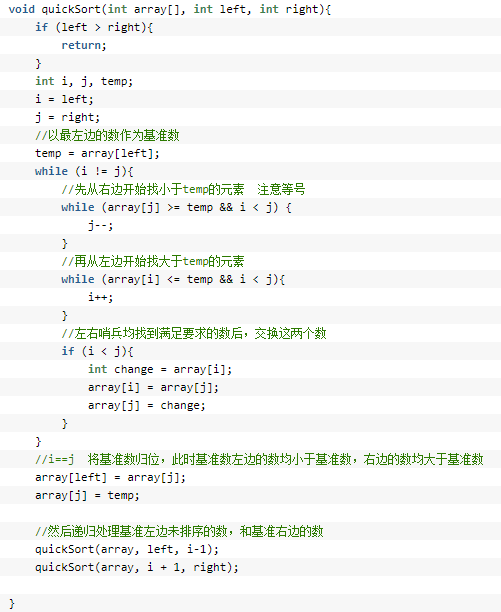
## **为什么快排其实也不是那么快**

我们考虑快排的过程：随机选择一个元素做“轴元素”，将所有大于轴元素的移到左边，其余移到右边。根据这个过程，快排的第一次比较就是将一个元素和轴元素比较，这个时候显而易见的是，“大于”和“小于”的可能性各占一半。这是一次漂亮的比较。

然而，快排的第二次比较就不那么高明了：我们不妨令轴元素为pivot，第一次比较结果是a1<pivot，那么可以证明第二次比较a2也小于pivot的可能性是2/3！这容易证明：如果a2>pivot的话，那么a1，a2，pivot这三个元素之间的关系就完全确定了——a1<pivot<a2，剩下来的元素排列的可能性我们不妨记为P（不需要具体算出来）。而如果a2<pivot呢？那么a1和a2的关系就仍然是不确定的，也就是说，这个分支里面含有两种情况：a1<a2<pivot，以及a2<a1<pivot。对于其中任一种情况，剩下的元素排列的可能性都是P，于是这个分支里面剩下的排列可能性就是2P。所以当a2<pivot的时候，还剩下2/3的可能性需要排查。

再进一步，如果第二步比较果真发现a2<pivot的话，第三步比较就更不妙了，模仿上面的推理，a3<pivot的概率将会是3/4！

这就是快排也不那么快的原因，因为它也没有做到每次比较都能将剩下的可能性砍掉一半。



另一种方法：

1. **int** partition(vector<**int**> &vi, **int** low, **int** up)
2. {
3. **int** pivot = vi[up];
4. **int** i = low;
5. **for** (**int** j = low; j < up; j++)
6. {
7. **if**(vi[j] <= pivot)
8. {
9. swap(vi[i++], vi[j]);
10. }
11. }
12. swap(vi[i], vi[up]);
13. **return** i+1;
14. }
15. **void** quickSort(vector<**int**> &vi, **int** low, **int** up)
16. {
17. **if**(low < up)
18. {
19. **int** mid = partition(vi, low, up);
20. //Watch out! The mid position is on the place, so we don't need to consider it again.
21. //That's why below is mid-1, not mid! Otherwise it will occur overflow error!!!
22. quickSort(vi, low, mid-1);
23. quickSort(vi, mid+1, up);
24. }
25. }

快速排序的基本思想：通过一趟排序将待排记录分隔成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。

6.1 算法描述

快速排序使用分治法来把一个串（list）分为两个子串（sub-lists）。具体算法描述如下：

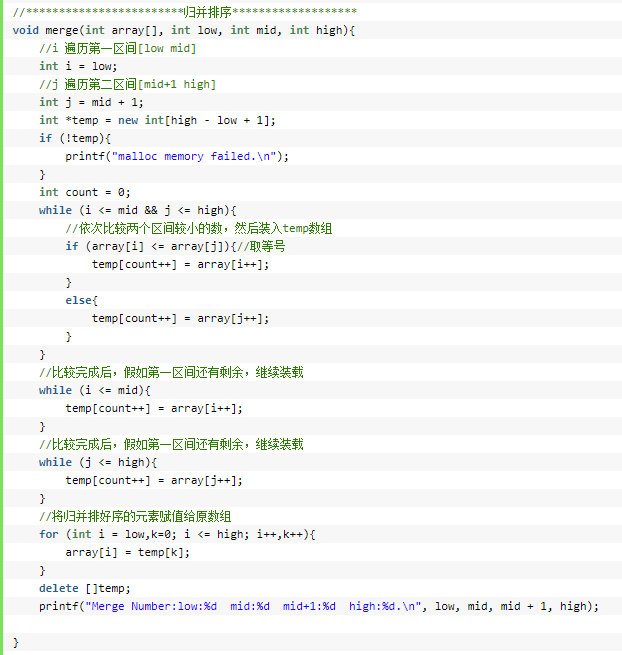
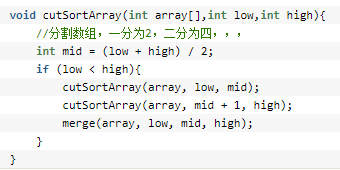
* 从数列中挑出一个元素，称为 “基准”（pivot）；
* 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；
* 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序。

# 归并：

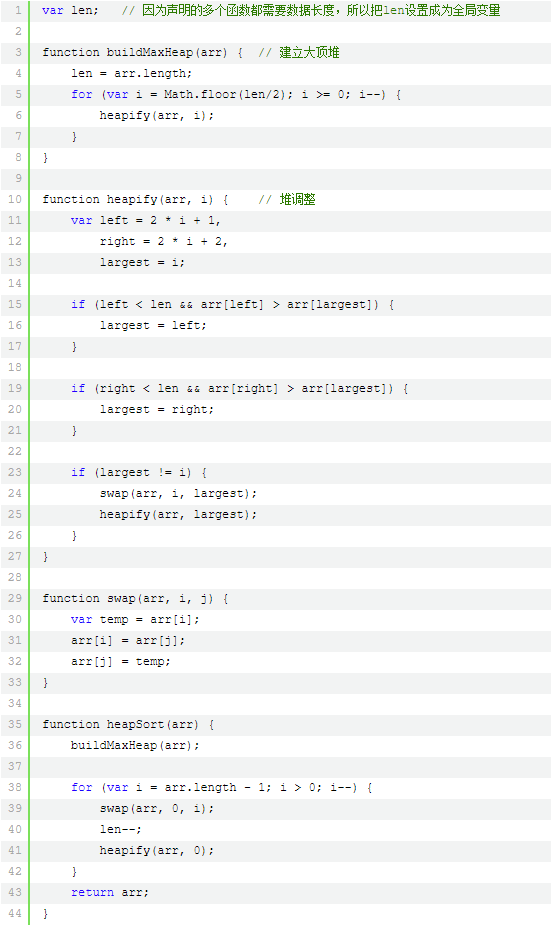
归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为2-路归并。

5.1 算法描述

* 把长度为n的输入序列分成两个长度为n/2的子序列；
* 对这两个子序列分别采用归并排序；
* 将两个排序好的子序列合并成一个最终的排序序列。



# 堆排序：

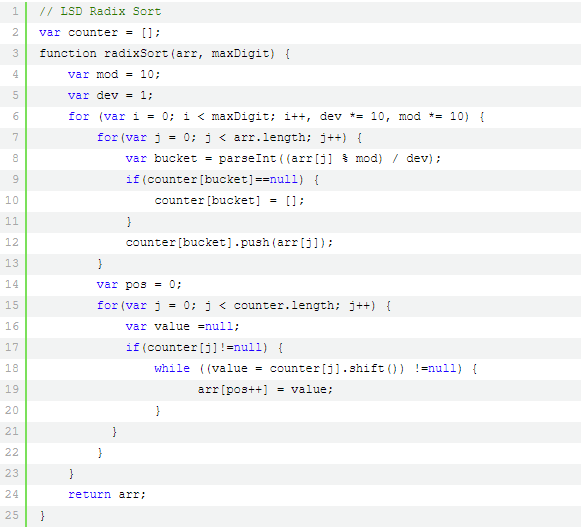


堆排序（Heapsort）是指利用堆这种数据结构所设计的一种排序算法。堆积是一个近似完全二叉树的结构，并同时满足堆积的性质：即子结点的键值或索引总是小于（或者大于）它的父节点。

7.1 算法描述

* 将初始待排序关键字序列(R1,R2….Rn)构建成大顶堆，此堆为初始的无序区；
* 将堆顶元素R[1]与最后一个元素R[n]交换，此时得到新的无序区(R1,R2,……Rn-1)和新的有序区(Rn),且满足R[1,2…n-1]<=R[n]；
* 由于交换后新的堆顶R[1]可能违反堆的性质，因此需要对当前无序区(R1,R2,……Rn-1)调整为新堆，然后再次将R[1]与无序区最后一个元素交换，得到新的无序区(R1,R2….Rn-2)和新的有序区(Rn-1,Rn)。不断重复此过程直到有序区的元素个数为n-1，则整个排序过程完成。

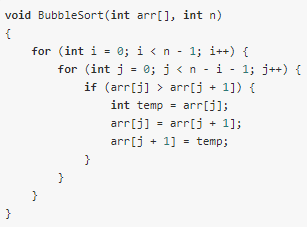
# 基数排序：



# 冒泡：

冒泡排序是一种简单的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果它们的顺序错误就把它们交换过来。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。这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端。

* 1.1 算法描述
* 比较相邻的元素。如果第一个比第二个大，就交换它们两个；
* 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，这样在最后的元素应该会是最大的数；
* 针对所有的元素重复以上的步骤，除了最后一个；
* 重复步骤1~3，直到排序完成。



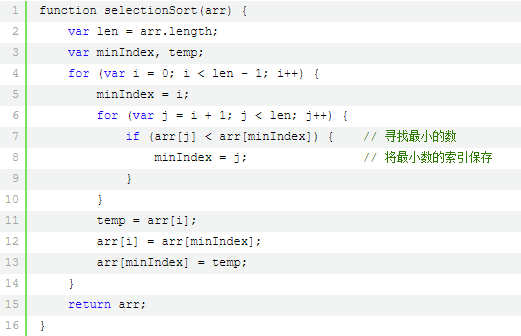
# 选择：

选择排序(Selection-sort)是一种简单直观的排序算法。它的工作原理：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。

2.1 算法描述

n个记录的直接选择排序可经过n-1趟直接选择排序得到有序结果。具体算法描述如下：

* 初始状态：无序区为R[1..n]，有序区为空；
* 第i趟排序(i=1,2,3…n-1)开始时，当前有序区和无序区分别为R[1..i-1]和R(i..n）。该趟排序从当前无序区中-选出关键字最小的记录 R[k]，将它与无序区的第1个记录R交换，使R[1..i]和R[i+1..n)分别变为记录个数增加1个的新有序区和记录个数减少1个的新无序区；
* n-1趟结束，数组有序化了。



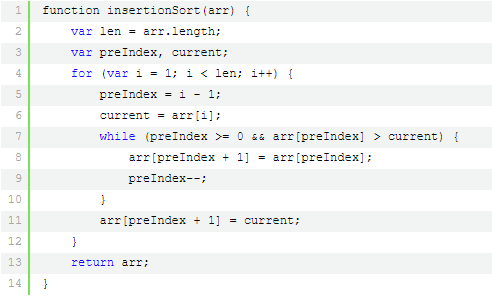
# 插入

插入排序（Insertion-Sort）的算法描述是一种简单直观的排序算法。它的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。

3.1 算法描述

一般来说，插入排序都采用in-place在数组上实现。具体算法描述如下：

* 从第一个元素开始，该元素可以认为已经被排序；
* 取出下一个元素，在已经排序的元素序列中从后向前扫描；
* 如果该元素（已排序）大于新元素，将该元素移到下一位置；
* 重复步骤3，直到找到已排序的元素小于或者等于新元素的位置；
* 将新元素插入到该位置后；
* 重复步骤2~5。



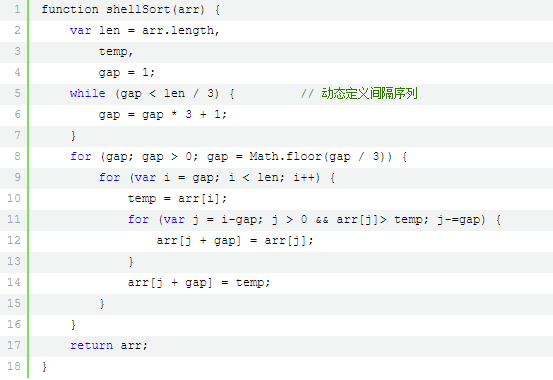
# 希尔排序

1959年Shell发明，第一个突破O(n2)的排序算法，是简单插入排序的改进版。它与插入排序的不同之处在于，它会优先比较距离较远的元素。希尔排序又叫**缩小增量排序**。

4.1 算法描述

先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，具体算法描述：

* 选择一个增量序列t1，t2，…，tk，其中ti>tj，tk=1；
* 按增量序列个数k，对序列进行k 趟排序；
* 每趟排序，根据对应的增量ti，将待排序列分割成若干长度为m 的子序列，分别对各子表进行直接插入排序。仅增量因子为1 时，整个序列作为一个表来处理，表长度即为整个序列的长度。



# 外部排序

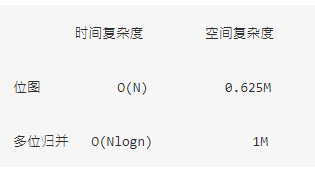
<http://www.cnblogs.com/LUO77/p/5838206.html>

外排序通常采用的是一种“排序-归并”的策略。在排序阶段，先读入能放在内存中的数据量，将其排序输出到一个临时文件，依此进行，将待排序数据组织为多个有序的临时文件；尔后在归并阶段将这些临时文件组合为一个大的有序文件，也即排序结果。

从2路到多路（k路），增大k可以减少外存信息读写时间，但k个归并段中选取最小的记录需要比较k-1次， 为得到u个记录的一个有序段共需要(u-1)(k-1)次，若归并趟数为s次，那么对n个记录的文件进行外排时，内部归并过程中进行的总的比较次数为 s(n-1)(k-1)，也即(向上取整)(logkm)(k-1)(n-1)=(向上取整)(log2m/log2k)(k-1)(n-1)，而(k- 1)/log2k随k增而增因此内部归并时间随k增长而增长了，抵消了外存读写减少的时间，这样做不行，由此引出了“败者树”tree of loser的使用。在内部归并过程中利用败者树将k个归并段中选取最小记录比较的次数降为(向上取整)(log2k)次使总比较次数为(向上取整) (log2m)(n-1)，与k无关。

    败者树是完全二叉树， 因此数据结构可以采用一维数组。其元素个数为k个叶子结点、k-1个比较结点、1个冠军结点共2k个。ls[0]为冠军结点，ls[1]--ls[k- 1]为比较结点，ls[k]--ls[2k-1]为叶子结点（同时用另外一个指针索引b[0]--b[k-1]指向）。另外bk为一个附加的辅助空间，不属于败者树，初始化时存着MINKEY的值。

**给10^7个数据量的磁盘文件排序**



# **解法一**：位图方案

采取这个位图的方案是因为我们面对的这个问题的特殊性：1输入数据限制在相对较小的范围内，2数据没有重复，3其中的每条记录都是单一的整数，没有任何其它与之关联的数据。

所以，此问题用位图的方案分为以下三步进行解决：

第一步，将所有的位都置为0，从而将集合初始化为空。第二步，通过读入文件中的每个整数来建立集合，将每个对应的位都置为1。第三步，检验每一位，如果该位为1，就输出对应的整数。

# **解法二**：多路归并

文件太大，无法一次性放入内存中计算处理，那这个时候咋办呢？分而治之，大而化小，也就是把整个大文件分为若干大小的几块，然后分别对每一块进行排序，最后完成整个过程的排序。k趟算法可以在kn的时间开销内和n/k的空间开销内完成对最多n个小于n的无重复正整数的排序。

多路归并排序算法的过程大致为：

   1）：首先将k个归并段中的首元素关键字依次存入b[0]--b[k-1]的叶子结点空间里，然后调用CreateLoserTree创建败者树，创建完毕之后最小的关键字下标（即所在归并段的序号）便被存入ls[0]中。然后不断循环：

   2）把ls[0]所存最小关键字来自于哪个归并段的序号得到为q，将该归并段的首元素输出到有序归并段里，然后把下一个元素关键字放入上一个元素本来所 在的叶子结点b[q]中，调用Adjust顺着b[q]这个叶子结点往上调整败者树直到新的最小的关键字被选出来，其下标同样存在ls[0]中。循环这个 操作过程直至所有元素被写到有序归并段里。

# ---------------------------------------

# 字典树 单词查找树 字符串排序O（n）

1、给出n个单词和m个询问，每次询问一个单词，回答这个单词是否在单词表中出现过。

答：简单！map，短小精悍。

2、给出n个单词和m个询问，每次询问一个前缀，回答询问是多少个单词的前缀。

答：map，把每个单词拆开。

judge：n<=200000，TLE！

这就需要一种高级数据结构——Trie树（字典树）

1、字典树用边表示字母

2、有相同前缀的单词公用前缀节点，那我们可以的得出每个节点最多有26个子节点（在单词只包含小写字母的情况下）

3、整棵树的根节点是空的。为什么呢？便于插入和查找，这将会在后面解释。

4、每个单词结束的时候用一个特殊字符表示，图中用的‘′，那么从根节点到任意一个‘′，那么从根节点到任意一个‘’所经过的边的所有字母表示一个单词。

**A、insert，插入一个单词**

1.思路

  从左到右扫这个单词，如果字母在相应根节点下没有出现过，就插入这个字母；否则沿着字典树往下走，看单词的下一个字母。

数组trie[i][j]=k，表示**编号为i**的节点的**第j个**孩子是**编号为k**的节点。

**B、search，查找**

Trie树的平均高度h为len，所以Trie树的查询复杂度为O（h）=O（len）

查找有很多种，可以查找某一个前缀，也可以查找整个单词。

再次我们以查找一个**前缀是否出现过**为例讲解

1、思路

  从左往右以此扫描每个字母，顺着字典树往下找，能找到这个字母，往下走，否则结束查找，即没有这个前缀；前缀扫完了，表示有这个前缀。

## 字典树的建立和查询

字典树的建立适合查询同步的。

1. #define MAX 26 *//字符集的大小*
2. *//结点定义*
3. typedef struct Node
4. {
5. struct Node \*next[MAX]; *//每个节点下面可能有MAX个字符*
6. int count; *//以从根节点到当前节点的字符串为公共前缀的字符串数目*
7. }TrieNode,\*TrieTree;
9. *//创建一棵仅有根节点的字典树*
10. TrieTree create\_TrieTree()
11. {
12. TrieTree pTree = (TrieTree)malloc(sizeof(TrieNode));
13. pTree->count = 0;
14. int i;
15. for(i=0;i<MAX;i++)
16. pTree->next[i] = NULL;
17. return pTree;
18. }
20. */\**
21. *插入字符串str到字典树pTree中*
22. *由于不可能改变根节点，因此这里不需要传入pTree的引用*
23. *\*/*
24. void insert\_TrieTree(TrieTree pTree,const char \*str)*//稍作修改可以变为查询*
25. {
26. int i;
27. TrieTree pCur = pTree; *//当前访问的节点，初始指向根节点*
28. int len = strlen(str);
29. for(i=0;i<len;i++)
30. {
31. int id = str[i] - 'a'; *//定位到字符应该在的位置*
32. if(!pCur->next[id])
33. { *//如果该字符应该在的位置为空，则将字符插入到该位置*
34. int j;
35. TrieNode \*pNew = (TrieNode \*)malloc(sizeof(TrieNode));
36. if(!pNew)
37. {
38. printf("pNew malloc fail\n");
39. exit(-1);
40. }
41. pNew->count = 1;
42. for(j=0;j<MAX;j++)
43. pNew->next[j] = NULL;
44. *//指针后移，比较下一个字符*
45. pCur->next[id] = pNew;
46. pCur = pCur->next[id];
47. }
48. else
49. { *//如果该字符应该在的位置不空，则继续比较下一个字符*
50. pCur = pCur->next[id];
51. pCur->count++; *//每插入一个字符，公共前缀数目就加1*
52. }
53. }
54. }
55. *//查询字典树中以str为公共前缀的字符串的个数*
56. int count\_TrieTree(TrieTree pTree,char \*str)
57. {
58. int i;
59. TrieTree pCur = pTree;
60. int len = strlen(str);
61. for(i=0;i<len;i++)
62. {
63. int id = str[i] - 'a';
64. if(!pCur->next[id])
65. return 0;
66. else
67. pCur = pCur->next[id];
68. }
69. return pCur->count;
70. }

## 字典树的应用

字典树的应用很广泛，下面是摘自百度百科的*3*个应用介绍：

*1*、在串快速检错中的应用

字典树的应用很广泛，下面是摘自百度百科的*3*个应用简介：

*1*、字典树在串的快速检索中的应用。

给出*N*个单词组成的熟词表，以及一篇全用小写英文书写的文章，请你按最早出现的顺序写出所有不在熟词表中的生词。在这道题中，我们可以用字典树，先把熟词建一棵树，然后读入文章进行比较，这种方法效率是比较高的。

*2*、字典树在“串”排序方面的应用

给定*N*个互不相同的仅由一个单词构成的英文名，让你将他们按字典序从小到大输出用字典树进行排序，采用数组的方式创建字典树，这棵树的每个结点的所有儿子很显然地按照其字母大小排序。对这棵树进行先序遍历即可。

*3*、字典树在最长公共前缀问题的应用

对所有串建立字典树，对于两个串的最长公共前缀的长度即他们所在的结点的公共祖先个数，于是，问题就转化为最近公共祖先问题。

4、在海量数据处理方面的应用

* 寻找热门查询：搜索引擎会通过日志文件把用户每次检索使用的所有检索串都记录下来，每个查询串的长度为1-255字节。假设目前有一千万个记录，这些查询串的重复读比较高，虽然总数是1千万，但是如果去除重复和，不超过3百万个。一个查询串的重复度越高，说明查询它的用户越多，也就越热门。请你统计最热门的10个查询串，要求使用的内存不能超过1G。  
  (1) 请描述你解决这个问题的思路；  
  (2) 请给出主要的处理流程，算法，以及算法的复杂度。

## 复杂度分析

**Trie树的平均高度h为len，所以Trie树的查询复杂度为O(h)=O(len)。**

**建立trie的复杂度为O(n\*len)。**

# 树表查找总结：

**二叉查找树平均查找性能不错，为O(logn)，但是最坏情况会退化为O(n)。在二叉查找树的基础上进行优化，我们可以使用平衡查找树。平衡查找树中的2-3查找树，这种数据结构在插入之后能够进行自平衡操作，从而保证了树的高度在一定的范围内进而能够保证最坏情况下的时间复杂度。但是2-3查找树实现起来比较困难，红黑树是2-3树的一种简单高效的实现，他巧妙地使用颜色标记来替代2-3树中比较难处理的3-node节点问题。红黑树是一种比较高效的平衡查找树，应用非常广泛，很多编程语言的内部实现都或多或少的采用了红黑树。**

**除此之外，2-3查找树的另一个扩展——B/B+平衡树，在文件系统和数据库系统中有着广泛的应用。**

## 二叉树和哈希对比

1. 原则上来说是hash的查找效率更高。
2. 针对具体的情况则不尽然。首先单纯的二叉树的查找效率是不高的，等于是无序数组的遍

历，需要转变成二叉排序树或者二叉平衡树才能提升查找效率，查找平均效率为O（logn）。其次hash的映射冲突的发生概率对hash的查找效率影响较大，在映射冲突较小的情况下平均查找效率为O（1），但如果映射函数对目标序列的适配性较差，则可能导致大量的数据发生映射冲突，从而大大降低查找效率。

## BST二叉搜索/排序/查找树

1.所有非叶子结点至多拥有两个儿子（Left和Right）；

2.所有结点存储一个关键字；

3.非叶子结点的左指针指向小于其关键字的子树，右指针指向大于其关键字的子树；

二叉查找树性质：对二叉查找树进行中序遍历，即可得到有序的数列。

复杂度分析：它和二分查找一样，插入和查找的时间复杂度均为O(logn)，但是在最坏的情况下仍然会有O(n)的时间复杂度。原因在于插入和删除元素的时候，树没有保持平衡

 二叉查找树的搜索，从根结点开始，如果查询的关键字与结点的关键字相等，那么就命中；否则，如果查询关键字比结点关键字小，就进入左儿子；如果比结点关键字大，就进入右儿子；如果左儿子或右儿子的指针为空，则报告找不到相应的关键字；如果二叉查找树的所有非叶子结点的左右子树的结点数目均保持差不多（平衡），那么二叉查找树的搜索性能逼近二分查找；但它比连续内存空间的二分查找的优点是，改变二叉查找树结构（插入与删除结点）不需要移动大段的内存数据，甚至通常是常数开销；

## AVL平衡二叉树

**平衡二叉树**（Balanced Binary Tree）又被称为AVL树（有别于AVL算法），且具有以下性质：它是一 棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。这个方案很好的解决了二叉查找树退化成链表的问题，把**插入，查找，删除的时间复杂度最好情况和最坏情况都维持在O(logN)**。但是频繁旋转会使插入和删除牺牲掉O(logN)左右的时间，不过相对二叉查找树来说，时间上稳定了很多。、

平衡二叉树大部分操作和二叉查找树类似，主要不同在于**插入删除**的时候平衡二叉树的平衡可能被改变，并且**只有从那些插入点 到 根结点 的路径上 的结点 的平衡性 可能被改变**，因为只有这些结点的子树可能变化。

**平衡二叉树不平衡的情形：**

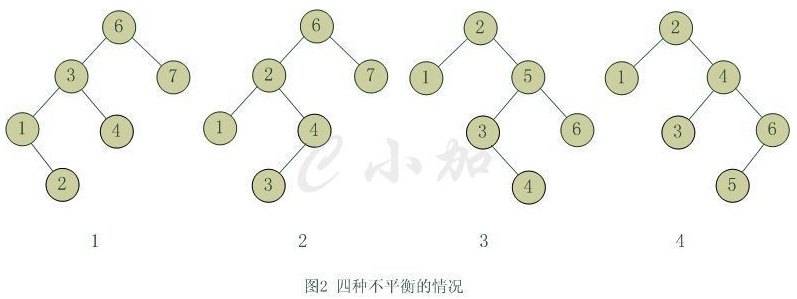
把需要重新平衡的结点叫做α，由于任意两个结点最多只有两个儿子，因此高度不平衡时，α结点的两颗子树的高度相差2.容易看出，这种不平衡可能出现在下面4中情况中：

1.对α的左儿子的左子树进行一次插入

2.对α的左儿子的右子树进行一次插入

3.对α的右儿子的左子树进行一次插入

4.对α的右儿子的右子树进行一次插入



情形1和情形4是关于α的镜像对称，二情形2和情形3也是关于α的镜像对称，因此理论上看只有两种情况，但编程的角度看还是四种情形。

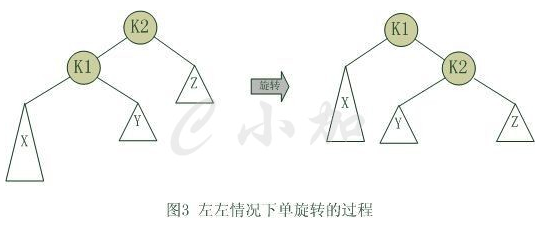
第一种情况是插入发生在“外边”的情形（左左或右右），该情况可以通过**一次单旋转**完成调整；第二种情况是插入发生在“内部”的情形（左右或右左），这种情况比较复杂，需要通过**双旋**转来调整。

**调整措施：**

**一、单旋转**

左左：根节点右旋

右右：根节点左旋



上图是左左的情况，k2结点不满足平衡性，它的左子树k1比右子树z深两层，k1子树中更深的是k1的左子树x，因此属于左左情况。

为了恢复平衡，我们把x上移一层，并把z下移一层，但此时实际已经超出了AVL树的性质要求。为此，重新安排结点以形成一颗等价的树。为使树恢复平衡，我们把k2变成这棵树的根节点，因为k2大于k1，把k2置于k1的右子树上，而原本在k1右子树的Y**大于k1，小于k2**，就把**Y置于k2的左子树上**，这样既满足了二叉查找树的性质，又满足了平衡二叉树的性质。

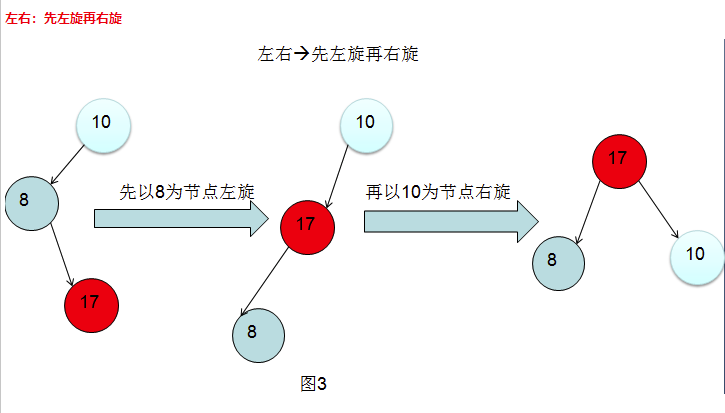
这种情况称为单旋转。

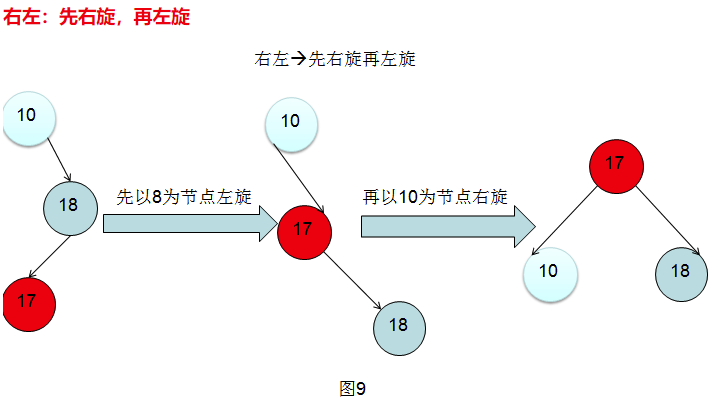
**二、双旋转**

对于左右和右左两种情况，单旋转不能解决问题，要经过两次旋转。

**“左右”情况：先以“左”为根节点进行左左旋转，然后对根节点进行右右旋转**

**“右左”情况：先以“右”为根节点进行右右旋转，然后对根节点进行左左旋转**





**AVL树的删除操作：**

同插入操作一样，删除结点时也有可能破坏平衡性，这就要求我们删除的时候要进行平衡性调整。

删除分为以下几种情况：

首先在整个二叉树中搜索要删除的结点，如果没搜索到直接返回不作处理，否则执行以下操作：

1.要删除的节点是当前根节点T。

如果左右子树都非空。在高度较大的子树中实施删除操作。

分两种情况：

(1)、左子树高度大于右子树高度，将左子树中最大的那个元素赋给当前根节点，然后删除左子树中元素值最大的那个节点。

(1)、左子树高度小于右子树高度，将右子树中最小的那个元素赋给当前根节点，然后删除右子树中元素值最小的那个节点。

如果左右子树中有一个为空，那么直接用那个非空子树或者是NULL替换当前根节点即可。

2、要删除的节点元素值小于当前根节点T值，在左子树中进行删除。

递归调用，在左子树中实施删除。

这个是需要判断当前根节点是否仍然满足平衡条件，

如果满足平衡条件，只需要更新当前根节点T的高度信息。

否则，需要进行旋转调整：

如果T的左子节点的左子树的高度大于T的左子节点的右子树的高度，进行相应的单旋转。否则进行双旋转。

3、要删除的节点元素值大于当前根节点T值，在右子树中进行删除。

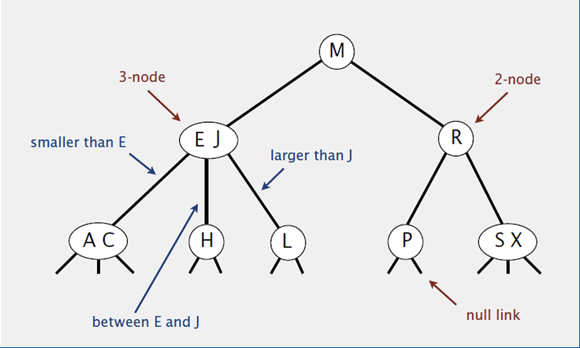
## 2-3Tree平衡查找树之2-3查找树

2-3查找树定义：和二叉树不一样，2-3树运行每个节点保存1个或者两个的值。对于普通的2节点(2-node)，他保存1个key和左右两个自己点。对应3节点(3-node)，保存两个Key，2-3查找树的定义如下：

　　1）要么为空，要么：

　　2）对于2节点，该节点保存一个key及对应value，以及两个指向左右节点的节点，左节点也是一个2-3节点，所有的值都比key要小，右节点也是一个2-3节点，所有的值比key要大。

　　3）对于3节点，该节点保存两个key及对应value，以及三个指向左中右的节点。左节点也是一个2-3节点，所有的值均比两个key中的最小的key还要小；中间节点也是一个2-3节点，中间节点的key值在两个跟节点key值之间；右节点也是一个2-3节点，节点的所有key值比两个key中的最大的key还要大。



2-3查找树的性质：

　　1）如果中序遍历2-3查找树，就可以得到排好序的序列；

　　2）在一个完全平衡的2-3查找树中，根节点到每一个为空节点的距离都相同。（这也是平衡树中“平衡”一词的概念，根节点到叶节点的最长距离对应于查找算法的最坏情况，而平衡树中根节点到叶节点的距离都一样，最坏情况也具有对数复杂度。）

复杂度分析：

　　2-3树的查找效率与树的高度是息息相关的。

* 在最坏的情况下，也就是所有的节点都是2-node节点，查找效率为lgN
* 在最好的情况下，所有的节点都是3-node节点，查找效率为log3N约等于0.631lgN

　　距离来说，对于1百万个节点的2-3树，树的高度为12-20之间，对于10亿个节点的2-3树，树的高度为18-30之间。

　　对于插入来说，只需要常数次操作即可完成，因为他只需要修改与该节点关联的节点即可，不需要检查其他节点，所以效率和查找类似。

## RBT平衡查找树之红黑树

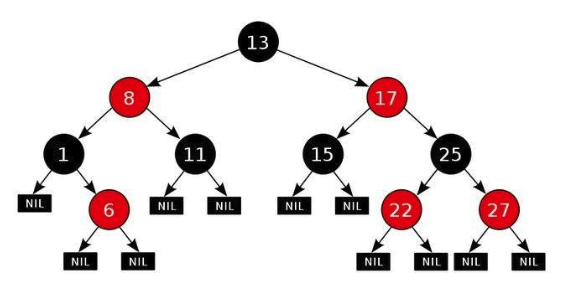
虽然本质上是一棵二叉查找树，但它在二叉查找树的基础上增加了着色和相关的性质使得红黑树相对平衡，从而保证了红黑树的查找、插入、删除的时间复杂度最坏为O(log n)。

红黑树的应用比较广泛，主要是用它来存储有序的数据，它的时间复杂度是O(lgn)，效率非常之高。  
例如，Java集合中的[TreeSet](http://www.cnblogs.com/skywang12345/p/3311268.html)和[TreeMap](http://www.cnblogs.com/skywang12345/p/3310928.html)，C++ STL中的set、map，以及Linux虚拟内存的管理，都是通过红黑树去实现的。Epoll底层。

平衡查找树中的2-3树以及其实现红黑树。2-3树种，一个节点最多有2个key，而红黑树则使用染色的方式来标识这两个key。2-3查找树能保证在插入元素之后能保持树的平衡状态，最坏情况下即所有的子节点都是2-node，树的高度为lgn，从而保证了最坏情况下的时间复杂度。但是2-3树实现起来比较复杂，于是就有了一种简单实现2-3树的数据结构，即红黑树（Red-Black Tree）。

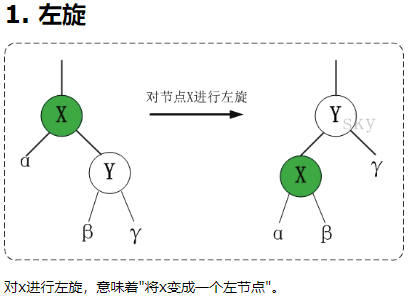
### 红黑树性质

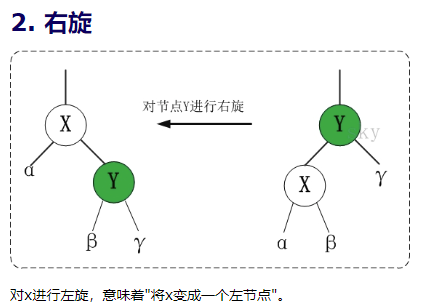
（1）每个结点或者是黑色，或者是红色；  
（2）根结点是黑色；  
（3）每个叶子结点（NIL）是黑色；（这里叶子结点，是指为空（NIL或NULL）的叶子结点）；  
（4）如果一个结点是红色的，则它的两个子结点必须是黑色的；  
（5）从一个结点到该结点的子孙结点的所有路径上包含相同数目的黑结点；

每个叶子结点都带有两个空的黑色结点（被称为黑哨兵），如果一个结点n的只有一个左孩子，那么n的右孩子是一个黑哨兵；如果结点n只有一个右孩子，那么n的左孩子是一个黑哨兵。  


### 树的旋转

旋转的原因是：添加或删除红黑树中的节点之后，红黑树就发生了变化，可能不满足红黑树的5条性质，也就不再是一颗红黑树了，而是一颗普通的树。而通过旋转，可以使这颗树重新成为红黑树。简单点说，旋转的目的是让树保持红黑树的特性





### 树的插入

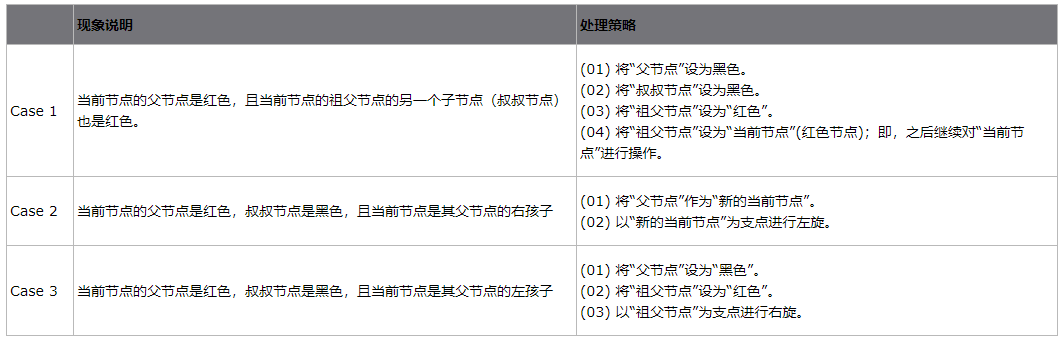
在讨论红黑树的插入操作之前必须要明白，任何一个即将插入的新结点的初始颜色都为红色。这一点很容易理解，因为插入黑点会增加某条路径上黑结点的数目，从而导致整棵树黑高度的不平衡。但如果新结点父结点为红色时（如下图所示），将会违返红黑树性质：一条路径上不能出现相邻的两个红色结点。这时就需要通过一系列操作来使红黑树保持平衡。

**第一步: 将红黑树当作一颗二叉查找树，将节点插入。**  
       红黑树本身就是一颗二叉查找树，将节点插入后，该树仍然是一颗二叉查找树。也就意味着，树的键值仍然是有序的。此外，无论是左旋还是右旋，若旋转之前这棵树是二叉查找树，旋转之后它一定还是二叉查找树。这也就意味着，任何的旋转和重新着色操作，都不会改变它仍然是一颗二叉查找树的事实。  
       好吧？那接下来，我们就来想方设法的旋转以及重新着色，使这颗树重新成为红黑树！

**第二步：将插入的节点着色为"红色"。**  
       为什么着色成红色，而不是黑色呢？为什么呢？在回答之前，我们需要重新温习一下红黑树的特性：  
(1) 每个节点或者是黑色，或者是红色。  
(2) 根节点是黑色。  
(3) 每个叶子节点是黑色。 [注意：这里叶子节点，是指为空的叶子节点！]  
(4) 如果一个节点是红色的，则它的子节点必须是黑色的。  
(5) 从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。  
       将插入的节点着色为红色，不会违背"特性(5)"！少违背一条特性，就意味着我们需要处理的情况越少。接下来，就要努力的让这棵树满足其它性质即可；满足了的话，它就又是一颗红黑树了。

**第三步: 通过一系列的旋转或着色等操作，使之重新成为一颗红黑树。**  
       第二步中，将插入节点着色为"红色"之后，不会违背"特性(5)"。那它到底会违背哪些特性呢？  
       对于"特性(1)"，显然不会违背了。因为我们已经将它涂成红色了。  
       对于"特性(2)"，显然也不会违背。在第一步中，我们是将红黑树当作二叉查找树，然后执行的插入操作。而根据二叉查找数的特点，插入操作不会改变根节点。所以，根节点仍然是黑色。  
       对于"特性(3)"，显然不会违背了。这里的叶子节点是指的空叶子节点，插入非空节点并不会对它们造成影响。  
       对于"特性(4)"，是有可能违背的！  
       那接下来，想办法使之"满足特性(4)"，就可以将树重新构造成红黑树了。

根据被插入节点的父节点的情况，可以将"当节点z被着色为红色节点，并插入二叉树"划分为三种情况来处理。  
① 情况说明：被插入的节点是根节点。  
    处理方法：直接把此节点涂为黑色。  
② 情况说明：被插入的节点的父节点是黑色。  
    处理方法：什么也不需要做。节点被插入后，仍然是红黑树。  
③ 情况说明：被插入的节点的父节点是红色。  
    处理方法：那么，该情况与红黑树的“特性(5)”相冲突。这种情况下，被插入节点是一定存在非空祖父节点的；进一步的讲，被插入节点也一定存在叔叔节点(即使叔叔节点为空，我们也视之为存在，空节点本身就是黑色节点)。理解这点之后，我们依据"叔叔节点的情况"，将这种情况进一步划分为3种情况(Case)。三种情况(Case)处理问题的核心思路都是：将红色的节点移到根节点；然后，将根节点设为黑色。

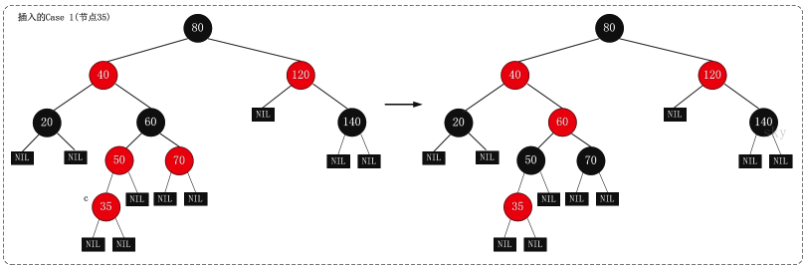


**1. (Case 1)叔叔是红色**

**1.1 现象说明**  
当前节点(即，被插入节点)的父节点是红色，且当前节点的祖父节点的另一个子节点（叔叔节点）也是红色。

**1.2 处理策略**  
(01) 将“父节点”设为黑色。  
(02) 将“叔叔节点”设为黑色。  
(03) 将“祖父节点”设为“红色”。  
(04) 将“祖父节点”设为“当前节点”(红色节点)；即，之后继续对“当前节点”进行操作。

**下面谈谈为什么要这样处理。**(建议理解的时候，通过下面的图进行对比)  
    “当前节点”和“父节点”都是红色，违背“特性(4)”。所以，将“父节点”设置“黑色”以解决这个问题。  
    但是，将“父节点”由“红色”变成“黑色”之后，违背了“特性(5)”：因为，包含“父节点”的分支的黑色节点的总数增加了1。  解决这个问题的办法是：将“祖父节点”由“黑色”变成红色，同时，将“叔叔节点”由“红色”变成“黑色”。关于这里，说明几点：第一，为什么“祖父节点”之前是黑色？这个应该很容易想明白，因为在变换操作之前，该树是红黑树，“父节点”是红色，那么“祖父节点”一定是黑色。 第二，为什么将“祖父节点”由“黑色”变成红色，同时，将“叔叔节点”由“红色”变成“黑色”；能解决“包含‘父节点’的分支的黑色节点的总数增加了1”的问题。这个道理也很简单。“包含‘父节点’的分支的黑色节点的总数增加了1” 同时也意味着 “包含‘祖父节点’的分支的黑色节点的总数增加了1”，既然这样，我们通过将“祖父节点”由“黑色”变成“红色”以解决“包含‘祖父节点’的分支的黑色节点的总数增加了1”的问题； 但是，这样处理之后又会引起另一个问题“包含‘叔叔’节点的分支的黑色节点的总数减少了1”，现在我们已知“叔叔节点”是“红色”，将“叔叔节点”设为“黑色”就能解决这个问题。 所以，将“祖父节点”由“黑色”变成红色，同时，将“叔叔节点”由“红色”变成“黑色”；就解决了该问题。  
    按照上面的步骤处理之后：当前节点、父节点、叔叔节点之间都不会违背红黑树特性，但祖父节点却不一定。若此时，祖父节点是根节点，直接将祖父节点设为“黑色”，那就完全解决这个问题了；若祖父节点不是根节点，那我们需要将“祖父节点”设为“新的当前节点”，接着对“新的当前节点”进行分析。

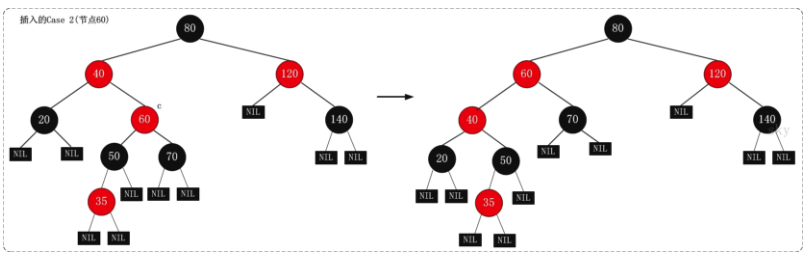


**2. (Case 2)叔叔是黑色，且当前节点是右孩子**

**2.1 现象说明**  
当前节点(即，被插入节点)的父节点是红色，叔叔节点是黑色，且当前节点是其父节点的右孩子

**2.2 处理策略**  
(01) 将“父节点”作为“新的当前节点”。  
(02) 以“新的当前节点”为支点进行左旋。

**下面谈谈为什么要这样处理。**(建议理解的时候，通过下面的图进行对比)  
      首先，将“父节点”作为“新的当前节点”；接着，以“新的当前节点”为支点进行左旋。 为了便于理解，我们先说明第(02)步，再说明第(01)步；为了便于说明，我们设置“父节点”的代号为F(Father)，“当前节点”的代号为S(Son)。  
为什么要“以F为支点进行左旋”呢？根据已知条件可知：S是F的右孩子。而之前我们说过，我们处理红黑树的核心思想：将红色的节点移到根节点；然后，将根节点设为黑色。既然是“将红色的节点移到根节点”，那就是说要不断的将破坏红黑树特性的红色节点上移(即向根方向移动)。 而S又是一个右孩子，因此，我们可以通过“左旋”来将S上移！   
      按照上面的步骤(以F为支点进行左旋)处理之后：若S变成了根节点，那么直接将其设为“黑色”，就完全解决问题了；若S不是根节点，那我们需要执行步骤(01)，即“将F设为‘新的当前节点’”。那为什么不继续以S为新的当前节点继续处理，而需要以F为新的当前节点来进行处理呢？这是因为“左旋”之后，F变成了S的“子节点”，即S变成了F的父节点；而我们处理问题的时候，需要从下至上(由叶到根)方向进行处理；也就是说，必须先解决“孩子”的问题，再解决“父亲”的问题；所以，我们执行步骤(01)：将“父节点”作为“新的当前节点”。

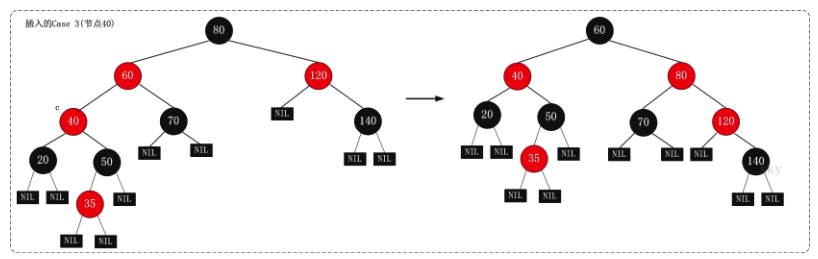


**3. (Case 3)叔叔是黑色，且当前节点是左孩子**

**3.1 现象说明**  
当前节点(即，被插入节点)的父节点是红色，叔叔节点是黑色，且当前节点是其父节点的左孩子

**3.2 处理策略**  
(01) 将“父节点”设为“黑色”。  
(02) 将“祖父节点”设为“红色”。  
(03) 以“祖父节点”为支点进行右旋。

**下面谈谈为什么要这样处理。**(建议理解的时候，通过下面的图进行对比)  
      为了便于说明，我们设置“当前节点”为S(Original Son)，“兄弟节点”为B(Brother)，“叔叔节点”为U(Uncle)，“父节点”为F(Father)，祖父节点为G(Grand-Father)。  
      S和F都是红色，违背了红黑树的“特性(4)”，我们可以将F由“红色”变为“黑色”，就解决了“违背‘特性(4)’”的问题；但却引起了其它问题：违背特性(5)，因为将F由红色改为黑色之后，所有经过F的分支的黑色节点的个数增加了1。那我们如何解决“所有经过F的分支的黑色节点的个数增加了1”的问题呢？ 我们可以通过“将G由黑色变成红色”，同时“以G为支点进行右旋”来解决。



### 树的删除

将红黑树内的某一个节点删除。需要执行的操作依次是：首先，将红黑树当作一颗二叉查找树，将该节点从二叉查找树中删除；然后，通过"旋转和重新着色"等一系列来修正该树，使之重新成为一棵红黑树。详细描述如下：

**第一步：将红黑树当作一颗二叉查找树，将节点删除。**  
       这和"删除常规二叉查找树中删除节点的方法是一样的"。分3种情况：  
       ① 被删除节点没有儿子，即为叶节点。那么，直接将该节点删除就OK了。  
       ② 被删除节点只有一个儿子。那么，直接删除该节点，并用该节点的唯一子节点顶替它的位置。  
       ③ 被删除节点有两个儿子。那么，先找出它的后继节点；然后把“它的后继节点的内容”复制给“该节点的内容”；之后，删除“它的后继节点”。在这里，后继节点相当于替身，在将后继节点的内容复制给"被删除节点"之后，再将后继节点删除。这样就巧妙的将问题转换为"删除后继节点"的情况了，下面就考虑后继节点。 在"被删除节点"有两个非空子节点的情况下，它的后继节点不可能是双子非空。既然"的后继节点"不可能双子都非空，就意味着"该节点的后继节点"要么没有儿子，要么只有一个儿子。若没有儿子，则按"情况① "进行处理；若只有一个儿子，则按"情况② "进行处理。

**第二步：通过"旋转和重新着色"等一系列来修正该树，使之重新成为一棵红黑树。**  
       因为"第一步"中删除节点之后，可能会违背红黑树的特性。所以需要通过"旋转和重新着色"来修正该树，使之重新成为一棵红黑树。

前面我们将"删除红黑树中的节点"大致分为两步，在第一步中"将红黑树当作一颗二叉查找树，将节点删除"后，可能违反"特性(2)、(4)、(5)"三个特性。第二步需要解决上面的三个问题，进而保持红黑树的全部特性。  
      为了便于分析，我们假设"x包含一个额外的黑色"(x原本的颜色还存在)，这样就不会违反"特性(5)"。为什么呢？  
      通过RB-DELETE算法，我们知道：删除节点y之后，x占据了原来节点y的位置。 既然删除y(y是黑色)，意味着减少一个黑色节点；那么，再在该位置上增加一个黑色即可。这样，当我们假设"x包含一个额外的黑色"，就正好弥补了"删除y所丢失的黑色节点"，也就不会违反"特性(5)"。 因此，假设"x包含一个额外的黑色"(x原本的颜色还存在)，这样就不会违反"特性(5)"。  
      现在，x不仅包含它原本的颜色属性，x还包含一个额外的黑色。即x的颜色属性是"红+黑"或"黑+黑"，它违反了"特性(1)"。

      现在，我们面临的问题，由解决"违反了特性(2)、(4)、(5)三个特性"转换成了"解决违反特性(1)、(2)、(4)三个特性"。RB-DELETE-FIXUP需要做的就是通过算法恢复红黑树的特性(1)、(2)、(4)。RB-DELETE-FIXUP的思想是：将x所包含的额外的黑色不断沿树上移(向根方向移动)，直到出现下面的姿态：  
a) x指向一个"红+黑"节点。此时，将x设为一个"黑"节点即可。  
b) x指向根。此时，将x设为一个"黑"节点即可。  
c) 非前面两种姿态。

将上面的姿态，可以概括为3种情况。  
① 情况说明：x是“红+黑”节点。  
    处理方法：直接把x设为黑色，结束。此时红黑树性质全部恢复。  
② 情况说明：x是“黑+黑”节点，且x是根。  
    处理方法：什么都不做，结束。此时红黑树性质全部恢复。  
③ 情况说明：x是“黑+黑”节点，且x不是根。  
    处理方法：这种情况又可以划分为4种子情况。这4种子情况如下表所示：

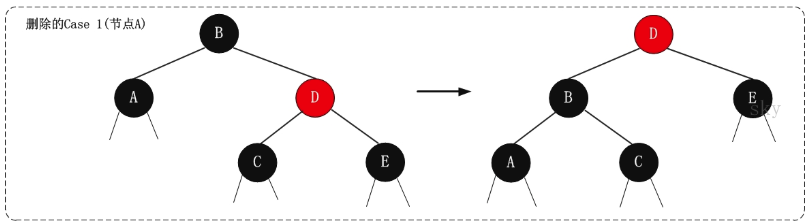


**1. (Case 1)x是"黑+黑"节点，x的兄弟节点是红色**

**1.1 现象说明**  
x是"黑+黑"节点，x的兄弟节点是红色。(此时x的父节点和x的兄弟节点的子节点都是黑节点)。

**1.2 处理策略**  
(01) 将x的兄弟节点设为“黑色”。  
(02) 将x的父节点设为“红色”。  
(03) 对x的父节点进行左旋。  
(04) 左旋后，重新设置x的兄弟节点。

**下面谈谈为什么要这样处理。**(建议理解的时候，通过下面的图进行对比)  
      这样做的目的是将“Case 1”转换为“Case 2”、“Case 3”或“Case 4”，从而进行进一步的处理。对x的父节点进行左旋；左旋后，为了保持红黑树特性，就需要在左旋前“将x的兄弟节点设为黑色”，同时“将x的父节点设为红色”；左旋后，由于x的兄弟节点发生了变化，需要更新x的兄弟节点，从而进行后续处理。

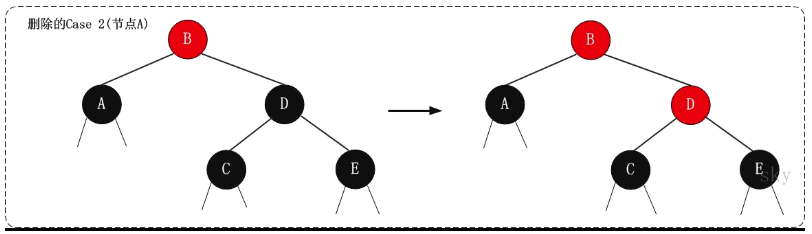


**2. (Case 2) x是"黑+黑"节点，x的兄弟节点是黑色，x的兄弟节点的两个孩子都是黑色**

**2.1 现象说明**  
x是“黑+黑”节点，x的兄弟节点是黑色，x的兄弟节点的两个孩子都是黑色。

**2.2 处理策略**  
(01) 将x的兄弟节点设为“红色”。  
(02) 设置“x的父节点”为“新的x节点”。

**下面谈谈为什么要这样处理。**(建议理解的时候，通过下面的图进行对比)  
      这个情况的处理思想：是将“x中多余的一个黑色属性上移(往根方向移动)”。 x是“黑+黑”节点，我们将x由“黑+黑”节点 变成 “黑”节点，多余的一个“黑”属性移到x的父节点中，即x的父节点多出了一个黑属性(若x的父节点原先是“黑”，则此时变成了“黑+黑”；若x的父节点原先时“红”，则此时变成了“红+黑”)。 此时，需要注意的是：所有经过x的分支中黑节点个数没变化；但是，所有经过x的兄弟节点的分支中黑色节点的个数增加了1(因为x的父节点多了一个黑色属性)！为了解决这个问题，我们需要将“所有经过x的兄弟节点的分支中黑色节点的个数减1”即可，那么就可以通过“将x的兄弟节点由黑色变成红色”来实现。  
      经过上面的步骤(将x的兄弟节点设为红色)，多余的一个颜色属性(黑色)已经跑到x的父节点中。我们需要将x的父节点设为“新的x节点”进行处理。若“新的x节点”是“黑+红”，直接将“新的x节点”设为黑色，即可完全解决该问题；若“新的x节点”是“黑+黑”，则需要对“新的x节点”进行进一步处理。

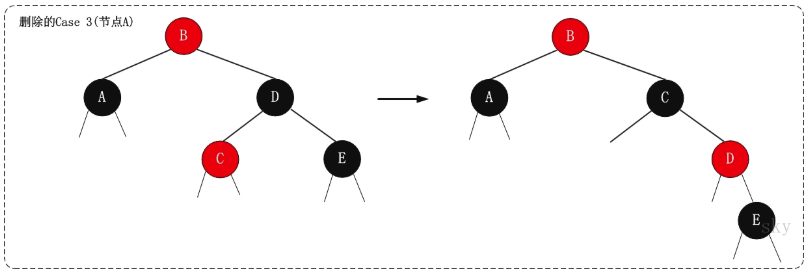


**3. (Case 3)x是“黑+黑”节点，x的兄弟节点是黑色；x的兄弟节点的左孩子是红色，右孩子是黑色的**

**3.1 现象说明**  
x是“黑+黑”节点，x的兄弟节点是黑色；x的兄弟节点的左孩子是红色，右孩子是黑色的。

**3.2 处理策略**  
(01) 将x兄弟节点的左孩子设为“黑色”。  
(02) 将x兄弟节点设为“红色”。  
(03) 对x的兄弟节点进行右旋。  
(04) 右旋后，重新设置x的兄弟节点。

**下面谈谈为什么要这样处理。**(建议理解的时候，通过下面的图进行对比)  
       我们处理“Case 3”的目的是为了将“Case 3”进行转换，转换成“Case 4”,从而进行进一步的处理。转换的方式是对x的兄弟节点进行右旋；为了保证右旋后，它仍然是红黑树，就需要在右旋前“将x的兄弟节点的左孩子设为黑色”，同时“将x的兄弟节点设为红色”；右旋后，由于x的兄弟节点发生了变化，需要更新x的兄弟节点，从而进行后续处理。



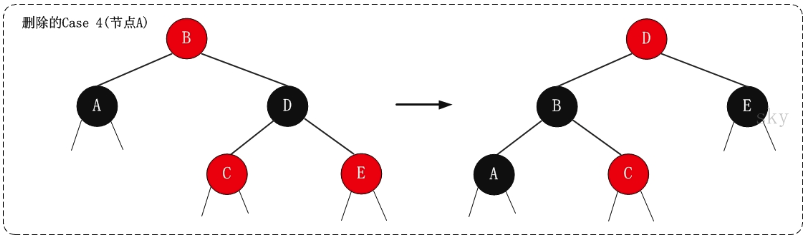
**4. (Case 4)x是“黑+黑”节点，x的兄弟节点是黑色；x的兄弟节点的右孩子是红色的，x的兄弟节点的左孩子任意颜色**

**4.1 现象说明**  
x是“黑+黑”节点，x的兄弟节点是黑色；x的兄弟节点的右孩子是红色的，x的兄弟节点的左孩子任意颜色。

**4.2 处理策略**  
(01) 将x父节点颜色 赋值给 x的兄弟节点。  
(02) 将x父节点设为“黑色”。  
(03) 将x兄弟节点的右子节设为“黑色”。  
(04) 对x的父节点进行左旋。  
(05) 设置“x”为“根节点”。

**下面谈谈为什么要这样处理。**(建议理解的时候，通过下面的图进行对比)  
      我们处理“Case 4”的目的是：去掉x中额外的黑色，将x变成单独的黑色。处理的方式是“：进行颜色修改，然后对x的父节点进行左旋。下面，我们来分析是如何实现的。  
      为了便于说明，我们设置“当前节点”为S(Original Son)，“兄弟节点”为B(Brother)，“兄弟节点的左孩子”为BLS(Brother's Left Son)，“兄弟节点的右孩子”为BRS(Brother's Right Son)，“父节点”为F(Father)。  
      我们要对F进行左旋。但在左旋前，我们需要调换F和B的颜色，并设置BRS为黑色。为什么需要这里处理呢？因为左旋后，F和BLS是父子关系，而我们已知BL是红色，如果F是红色，则违背了“特性(4)”；为了解决这一问题，我们将“F设置为黑色”。 但是，F设置为黑色之后，为了保证满足“特性(5)”，即为了保证左旋之后：  
      第一，“同时经过根节点和S的分支的黑色节点个数不变”。  
             若满足“第一”，只需要S丢弃它多余的颜色即可。因为S的颜色是“黑+黑”，而左旋后“同时经过根节点和S的分支的黑色节点个数”增加了1；现在，只需将S由“黑+黑”变成单独的“黑”节点，即可满足“第一”。  
      第二，“同时经过根节点和BLS的分支的黑色节点数不变”。  
             若满足“第二”，只需要将“F的原始颜色”赋值给B即可。之前，我们已经将“F设置为黑色”(即，将B的颜色"黑色"，赋值给了F)。至此，我们算是调换了F和B的颜色。  
      第三，“同时经过根节点和BRS的分支的黑色节点数不变”。  
             在“第二”已经满足的情况下，若要满足“第三”，只需要将BRS设置为“黑色”即可。  
经过，上面的处理之后。红黑树的特性全部得到的满足！接着，我们将x设为根节点，就可以跳出while循环(参考伪代码)；即完成了全部处理。

至此，我们就完成了Case 4的处理。理解Case 4的核心，是了解如何“去掉当前节点额外的黑色”。



### 查找插入删除操作代价分析：

（1）查找代价：由于红黑树的性质（最长路径长度不超过最短路径长度的2倍），可以说明红黑树虽然不像AVL一样是严格平衡的，但平衡性能还是要比BST要好；其查找代价基本维持在O（logN）左右，但在最差情况下（最长路径是最短路径的2倍少1），比AVL要略逊色一点；  
（2）插入代价：RBT插入结点时，需要旋转操作和变色操作；但由于只需要保证RBT基本平衡就可以了，因此插入结点最多只需要2次旋转，这一点和AVL的插入操作一样；虽然变色操作需要O（logN），但是变色操作十分简单，代价很小；  
（3）删除代价：RBT的删除操作代价要比AVL要好的多，删除一个结点最多只需要3次旋转操作；  
（4）RBT效率总结：查找效率最好情况下时间复杂度为O（logN），但在最坏情况下比AVL要差一些，但也远远好于BST；插入和删除操作改变树的平衡性的概率要远远小于AVL（RBT不是高度平衡的）；因此需要旋转操作的可能性要小，而且一旦需要旋转，插入一个结点最多只需要旋转2次，删除最多只需要旋转3次（小于AVL的删除操作所需要的旋转次数）；虽然变色操作的时间复杂度在O（logN），但是实际上，这种操作由于简单所需要的代价很小；

## B树/B+树 O(log n)

B和B+树的区别在于，B+树的非叶子结点只包含导航信息，不包含实际的值，所有的叶子结点和相连的节点使用链表相连，便于区间查找和遍历。

　B树（B-tree）是一种树状数据结构，它能够存储数据、对其进行排序并允许以O(log n)的时间复杂度运行进行查找、顺序读取、插入和删除的数据结构。B树，概括来说是一个节点可以拥有多于2个子节点的二叉查找树。与自平衡二叉查找树不同，B树为系统最优化 大块数据 的读和写操作。B-tree算法减少 定位记录时 所经历的 中间过程，从而**加快存取速度**。**普遍运用在数据库和文件系统。**

　　B树定义：

　　B树可以看作是对2-3查找树的一种扩展，即他允许每个节点有M-1个子节点。

* 根节点至少有两个子节点
* 每个节点有M-1个key，并且以升序排列
* 位于M-1和M key的子节点的值位于M-1 和M key对应的Value之间
* 其它节点至少有M/2个子节点

**B+树：**

　B+树定义：

　　B+树是对B树的一种变形树，它与B树的差异在于：

* 有k个子结点的结点必然有k个关键码；
* 非叶结点仅具有索引作用，跟记录有关的信息均存放在叶结点中。
* 树的所有叶结点构成一个有序链表，可以按照关键码排序的次序遍历全部记录。

B+ 树的优点在于：

* 由于B+树在内部节点上不好含数据信息，因此在内存页中能够存放更多的key。 数据存放的更加紧密，具有更好的空间局部性。因此访问叶子几点上关联的数据也具有更好的缓存命中率。
* B+树的叶子结点都是相链的，因此对整棵树的便利只需要一次线性遍历叶子结点即可。而且由于数据顺序排列并且相连，所以便于区间查找和搜索。而B树则需要进行每一层的递归遍历。相邻的元素可能在内存中不相邻，所以缓存命中性没有B+树好。

　　但是B树也有优点，其优点在于，由于B树的每一个节点都包含key和value，因此经常访问的元素可能离根节点更近，因此访问也更迅速。

# 二分查找

1. //非递归实现
2. **int** BinarySearch(**int** array[], **int** len, **int** value)
3. {
4. **if** (array == NULL || len <= 0)
5. **return** -1;
7. **int** low = 0;
8. **int** high = len - 1;
9. **while** (low <= high)
10. {
11. **int** mid = low + (high - low) / 2;
12. **if** (array[mid] == value)
13. **return** mid;
14. **else** **if** (array[mid] > value)
15. high = mid - 1;
16. **else**
17. low = mid + 1;
18. }
20. **return** -1;
21. }
22. //递归实现
23. **int** BinarySearch\_Recursive(**int** array[], **int** low, **int** high, **int** value)
24. {
25. **if** (low > high)
26. **return** -1;
27. **int** mid = low + (high - low) / 2;
28. **if** (array[mid] == value)
29. **return** mid;
30. **else** **if** (array[mid] > value)
31. **return** BinarySearch\_Recursive(array, low, mid - 1, value);
32. **else**
33. **return** BinarySearch\_Recursive(array, mid + 1, high, value);
35. }

# 二分查找-序列先增后减求峰值

(1)如果数组先增大再减小，一定有答案。

(2)如果数组单调递增，最后一个元素就是答案。

(3)如果数组单调递简，第一个元素就是答案。

(4)如果数组全部都一样，任何一个元素都是答案，更极端地，如果数组只有一个元素，那么这唯一的元素就是答案。

(5)如果数组出现了几个峰值，那么任何一个峰值都是答案。

二分一次找到中点，如果中点Array[mid]比Array[mid-1]和Array[mid+1]都大，那么就返回mid。否则，如果Array[mid]>Array[mid-1], 则最大值在mid右边，如果Array[mid]>Array[mid+1], 则最大值在mid左边。不断进行下去，直到找到为止。

1. **int** findPeakUtil(**int** arr[], **int** low, **int** high, **int** n)
2. {
3. **int** mid = low + (high - low)/2;  /\* (low + high)/2 \*/
5. **if** ((mid == 0 || arr[mid-1] <= arr[mid]) &&
6. (mid == n-1 || arr[mid+1] <= arr[mid]))
7. **return** mid;
9. **else** **if** (mid > 0 && arr[mid-1] > arr[mid])
10. **return** findPeakUtil(arr, low, (mid -1), n);
12. **else** **return** findPeakUtil(arr, (mid + 1), high, n);
13. }

# 二分查找-**旋转排序数组log(n)**

中间位置的两边是有序的

每次根据L和R求出m后，m左边[L, m-1]和右边[m+1, R]这两部分中至少一个是有序的。

　　　　　　　（1）arr[m]=X，返回m

　　　　　　　（2）arr[m]<arr[R]，位于m'位置右侧是有序的；当arr[m']<X<arr[R]时，则L=m'+1，否则R=m'-1。

　　　　　　　（3）arr[m]>=arr[R]，位于m位置左侧是有序的；当arr[L]<X<arr[m]时，则R=m-1，否则L=m+1。

int BSearch(int \*arr, int len, int X){

int L = 0, R = len - 1;

int m;

while (L <= R){ //循环条件

m = (L + R) / 2;

if (arr[m] == X) //找到了，终止

return m;

if (arr[m] < arr[R]){ //右侧有序

if (arr[m] < X && arr[R] >= X){

L = m + 1;

}

else

R = m - 1;

}

else{ //左侧有序

if (arr[m] > X && arr[L] <= X){

R = m - 1;

}

else

L = m + 1;

}

}

return -1; //循环结束，没找到。

}

上面要求的是没有重复元素，现在稍微扩展一下，可以有重复的元素，其它的要求不变。

思路：大致思路和原来相同，只不过相等的情况单独处理。找到的元素位置，并不一定是第一次出现的位置。

arr[L]<arr[m]；左边有序

arr[L]>arr[m]；右边有序

arr[L]=arr[m]；相等，特殊处理，++L（L=m+1也可以）。

和arr[R]比较也可以。

int BSearch(int \*arr, int len, int X){

int L = 0, R = len - 1;

int m;

while (L <= R){ //循环条件

m = (L + R) / 2;

if (arr[m] == X) //找到了，终止

return m;

if (arr[m] < arr[L]){ //右侧有序

if (arr[m] < X && arr[R] >= X){

L = m + 1;

}

else

R = m - 1;

}

else if (arr[m] > arr[L]){ //左侧有序

if (arr[m] > X && arr[L] <= X){

R = m - 1;

}

else

L = m + 1;

}

else //相等情况

L = m + 1;

}

return -1; //循环结束，没找到。

}

# 斐波那契查找

二分查找的一种提升算法

根据斐波那契序列的特点对有序表进行分割的。他要求开始表中记录的个数为某个斐波那契数小1，及n=F(k)-1;

 开始将k值与第F(k-1)位置的记录进行比较(及mid=low+F(k-1)-1),比较结果也分为三种

　　1）相等，mid位置的元素即为所求

　　2）>，low=mid+1,k-=2;

　　说明：low=mid+1说明待查找的元素在[mid+1,high]范围内，k-=2 说明范围[mid+1,high]内的元素个数为n-(F(k-1))= Fk-1-F(k-1)=Fk-F(k-1)-1=F(k-2)-1个，所以可以递归的应用斐波那契查找。

　　3）<，high=mid-1,k-=1。

　　说明：low=mid+1说明待查找的元素在[low,mid-1]范围内，k-=1 说明范围[low,mid-1]内的元素个数为F(k-1)-1个，所以可以递归 的应用斐波那契查找。

**复杂度分析：最坏情况下，时间复杂度为O(log2n)，且其期望复杂度也为O(log2n)。**

# 哈希查找

什么是哈希表（Hash）？

　　我们使用一个下标范围比较大的数组来存储元素。可以设计一个函数（哈希函数， 也叫做散列函数），使得每个元素的关键字都与一个函数值（即数组下标）相对应，于是用这个数组单元来存储这个元素；也可以简单的理解为，按照关键字为每一个元素"分类"，然后将这个元素存储在相应"类"所对应的地方。但是，不能够保证每个元素的关键字与函数值是一一对应的，因此极有可能出现对于不同的元素，却计算出了相同的函数值，这样就产生了"冲突"，换句话说，就是把不同的元素分在了相同的"类"之中。后面我们将看到一种解决"冲突"的简便做法。

　　总的来说，"直接定址"与"解决冲突"是哈希表的两大特点。

　　什么是哈希函数？

　　哈希函数的规则是：通过某种转换关系，使关键字适度的分散到指定大小的的顺序结构中，越分散，则以后查找的时间复杂度越小，空间复杂度越高。

　　算法思想：哈希的思路很简单，如果所有的键都是整数，那么就可以使用一个简单的无序数组来实现：将键作为索引，值即为其对应的值，这样就可以快速访问任意键的值。这是对于简单的键的情况，我们将其扩展到可以处理更加复杂的类型的键。

　　算法流程：

　　1）用给定的哈希函数构造哈希表；

　　2）根据选择的冲突处理方法解决地址冲突；

　　　　常见的解决冲突的方法：拉链法和线性探测法。

　　3）在哈希表的基础上执行哈希查找。

　　哈希表是一个在时间和空间上做出权衡的经典例子。如果没有内存限制，那么可以直接将键作为数组的索引。那么所有的查找时间复杂度为O(1)；如果没有时间限制，那么我们可以使用无序数组并进行顺序查找，这样只需要很少的内存。哈希表使用了适度的时间和空间来在这两个极端之间找到了平衡。只需要调整哈希函数算法即可在时间和空间上做出取舍。

　　复杂度分析：

单纯论查找复杂度：对于无冲突的Hash表而言，查找复杂度为O(1)（注意，在查找之前我们需要构建相应的Hash表）。

哈希表的几种方法

1、直接定址法：取关键字key的某个线性函数为散列地址，如Hash(key) = key  或 Hash(key) = A\*key+B；A,B为常数

2、除留取余法：关键值除以比散列表长度小的素数所得的余数作为散列地址。Hash(key) = key % p;

3、平均取中法：先计算构成关键码的标识符的内码的平方，然后按照散列表的大小取中间的若干位作为散列地址。

4、折叠法：把关键码自左到右分为位数相等的几部分，每一部分的位数应与散列表地址位数相同，只有最后一部分的位数可以短一些。把这些部分的数据叠加起来，就可以得到具有关键码的记录的散列地址。分为移位法和分界法。

5、随机数法：选择一个随机函数，取关键字的随机函数作为它的哈希地址。

6、数学分析法：设有N个d位数，每一位可能有r种不同的符号。这r种不同的符号在各位上出现的频率不一定相同，可能在某些位上分布均匀些，每种符号出现的机会均等；在某些位上分布不均匀，只有某几种符号经常出现。可根据散列表的大小，选取其中各种符号分布均匀的若干位作为散列地址。

## 哈希查找冲突解决：

1）开放定址法：闭散列

**线性探测再散列   di = 1 , 2 , 3 , … , m-1**

**平方探测再散列   di = 1 2 , -12 , 22 , -22 , 32 , -32 , … , k2 ,  -k2**

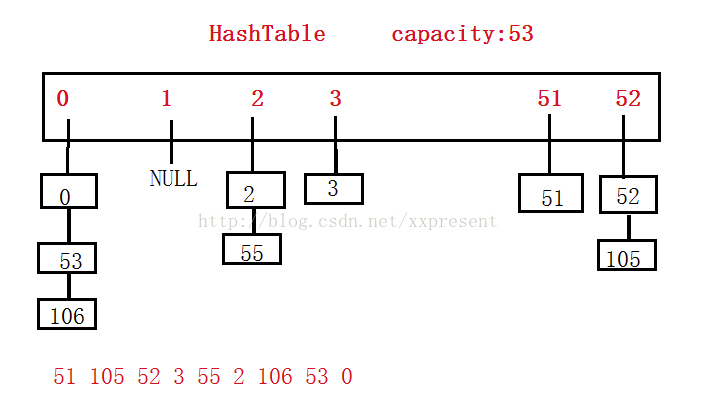
随机探测再散列   di 是一组伪随机数列

2）链地址法

Java hashmap就是这么做的

这种方法的基本思想是将所有哈希地址为i的元素构成一个称为同义词链的单链表，并将单链表的头指针存在哈希表的第i个单元中，因而查找、插入和删除主要在同义词链中进行。链地址法适用于经常进行插入和删除的情况。

当用线性探测和二次探测时，总是在一个有限的哈希表中存储数据，当数据特别多时，效率就比较低。因此采用拉链法的方式来降低哈希冲突。还有一种情况是，当一个链上链的数据过多时，我们可以采用红黑树的方式来降低高度，保持平衡且不至于过载。



3）再哈希

这种方法是同时构造多个不同的哈希函数：

Hi=RH1（key）  i=1，2，…，k

当哈希地址Hi=RH1（key）发生冲突时，再计算Hi=RH2（key）……，直到冲突不再产生。这种方法不易产生聚集，但增加了计算时间。

4）建立公共溢出区

将哈希表分为基本表和溢出表两部分，凡是和基本表发生冲突的元素，一律填入溢出表。

**优缺点**

**开放散列（open hashing）/ 拉链法（针对桶链结构）**

1）优点： ①对于记录总数频繁可变的情况，处理的比较好（也就是避免了动态调整的开销） ②由于记录存储在结点中，而结点是动态分配，不会造成内存的浪费，所以尤其适合那种记录本身尺寸（size）很大的情况，因为此时指针的开销可以忽略不计了 ③删除记录时，比较方便，直接通过指针操作即可

2）缺点： ①存储的记录是随机分布在内存中的，这样在查询记录时，相比结构紧凑的数据类型（比如数组），哈希表的跳转访问会带来额外的时间开销 ②如果所有的 key-value 对是可以提前预知，并之后不会发生变化时（即不允许插入和删除），可以人为创建一个不会产生冲突的完美哈希函数（perfect hash function），此时封闭散列的性能将远高于开放散列 ③由于使用指针，记录不容易进行序列化（serialize）操作

**封闭散列（closed hashing）/ 开放定址法**

1）优点： ①记录更容易进行序列化（serialize）操作 ②如果记录总数可以预知，可以创建完美哈希函数，此时处理数据的效率是非常高的

2）缺点： ①存储记录的数目不能超过桶数组的长度，如果超过就需要扩容，而扩容会导致某次操作的时间成本飙升，这在实时或者交互式应用中可能会是一个严重的缺陷 ②使用探测序列，有可能其计算的**时间成本过高**，导致哈希表的处理性能降低 ③由于记录是存放在桶数组中的，而桶数组必然存在空槽，所以当记录本身尺寸（size）很大并且记录总数规模很大时，空槽占用的空间会导致明显的**内存浪费** ④**删除记录时，比较麻烦**。比如需要删除记录a，记录b是在a之后插入桶数组的，但是和记录a有冲突，是通过探测序列再次跳转找到的地址，所以如果直接删除a，a的位置变为空槽，而空槽是查询记录失败的终止条件，这样会导致记录b在a的位置重新插入数据前不可见，所以不能直接删除a，而是设置删除标记。这就需要额外的空间和操作。

# B+树索引 插入删除 O(log(N))

查找一个特定值二叉查找树挺好用，但是当你需要查找两个值之间的多个元素时，就会有大麻烦了。你的成本将是 O(N)，因为你必须查找树的每一个节点，以判断它是否处于那 2 个值之间（例如，对树使用中序遍历）。而且这个操作不是磁盘I/O有利的，因为你必须读取整个树。我们需要找到高效的范围查询方法。为了解决这个问题，现代数据库使用了一种修订版的树，叫做B+树。在一个B+树里：

* 只有最底层的节点（叶子节点）才保存信息（相关表的行位置）
* 其它节点只是在搜索中用来指引到正确节点的。

你可以看到，节点更多了（多了两倍）。确实，你有了额外的节点，它们就是帮助你找到正确节点的『决策节点』（正确节点保存着相关表中行的位置）。但是搜索复杂度还是在 O(log(N))（只多了一层）。一个重要的不同点是，最底层的节点是跟后续节点相连接的。

用这个 B+树，假设你要找40到100间的值：

* 你只需要找 40（若40不存在则找40之后最贴近的值），就像你在上一个树中所做的那样。
* 然后用那些连接来收集40的后续节点，直到找到100。

比方说你找到了 M 个后续节点，树总共有 N 个节点。对指定节点的搜索成本是 log(N)，跟上一个树相同。但是当你找到这个节点，你得通过后续节点的连接得到 M 个后续节点，这需要 M 次运算。那么这次搜索只消耗了 M+log(N) 次运算，区别于上一个树所用的 N 次运算。此外，你不需要读取整个树（仅需要读 M+log(N) 个节点）,这意味着更少的磁盘访问。如果 M 很小（比如 200 行）并且 N 很大（1,000,000），那结果就是天壤之别了。

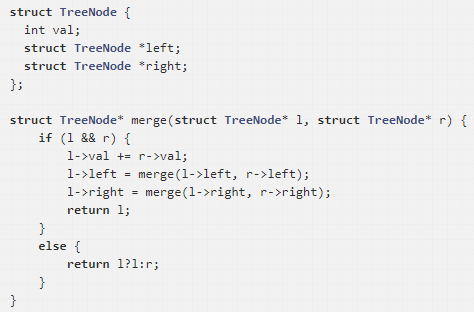
然而还有新的问题（又来了！）。如果你在数据库中增加或删除一行（从而在相关的 B+树索引里）：

* 你必须在B+树中的节点之间保持顺序，否则节点会变得一团糟，你无法从中找到想要的节点。
* 你必须尽可能降低B+树的层数，否则 O(log(N)) 复杂度会变成 O(N)。

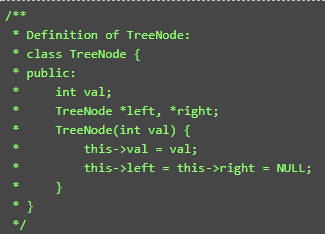
换句话说，B+树需要自我整理和自我平衡。谢天谢地，我们有智能删除和插入。但是这样也带来了成本：**在B+树中，插入和删除操作是 O(log(N)) 复杂度**。所以有些人听到过使用太多索引不是个好主意这类说法。没错，你减慢了快速插入/更新/删除表中的一个行的操作，因为数据库需要以代价高昂的每索引 O(log(N)) 运算来更新表的索引。再者，增加索引意味着给事务管理器带来更多的工作负荷。

# **---------------------------------------**

# 合并两个二叉树对应位置求和



# [二叉树的建立/查找/删除](https://blog.csdn.net/w397090770/article/details/7530826)

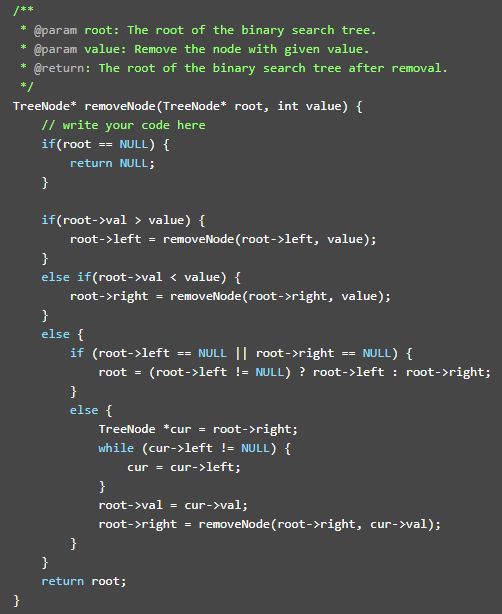


若要删除一个BST的一个结点，需要考虑如下三种情况：

1. 需要删除的节点下并没有其他子节点
2. 需要删除的节点下有一个子节点（左或右）
3. 需要删除的节点下有两个子节点（既左右节点都存在）

对这三种情况分别采取的措施是：

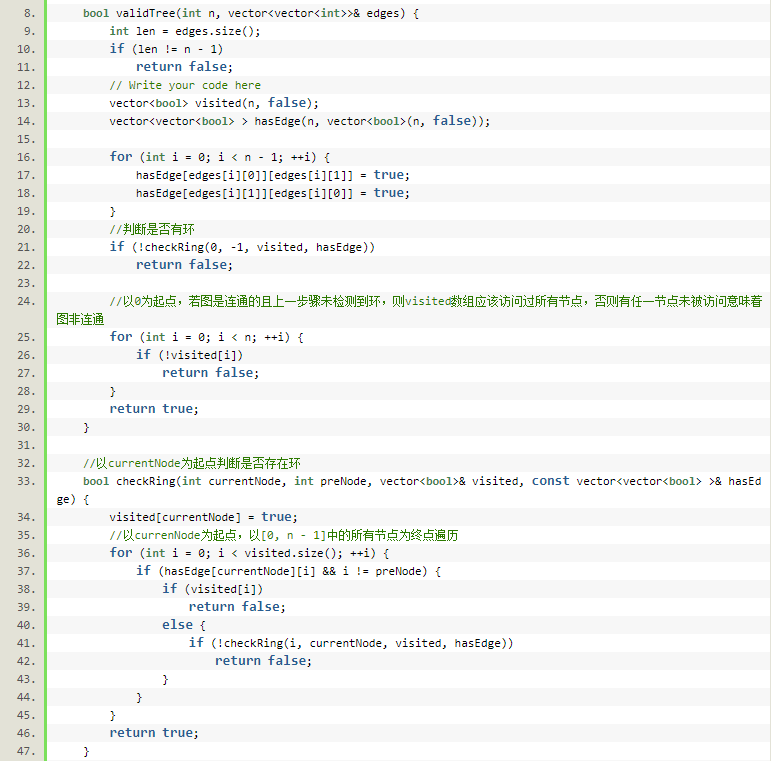
1. 直接删除此结点
2. 删除此结点，将此结点父节点连接到此结点左（右）子树
3. 找出此结点右子树中的最小结点，用以代替要删除的结点，然后删除此最小结点



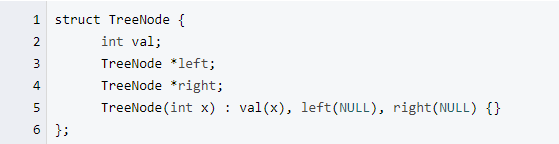
# [判断一个图是否是一个树/无向图](http://www.cppblog.com/fm369o802340/archive/2012/12/10/196151.html)

1. 是连通图：任意两点有且只有一条路径可达，这条路径的每个边只走一次
2. 无环

解决的方法：  
想象一开始这个图只有顶点，没有边，我们来一条一条的添加边。  
每遇到一条边，判断这边的两个端点是否在同一个集合里？集合的property:表示一个连通分量，里面的任意两点有且只有一条路径可达。在的话，表示有环：因为两个点在一个集合里就表示这两个点已经有一条路径了，现在再加一条路径，必然构成环。不在的话，表示不构成环，我们应该合并这两个集合：因为加上这条边，两个集合就被连起来了，合并成了一个集合

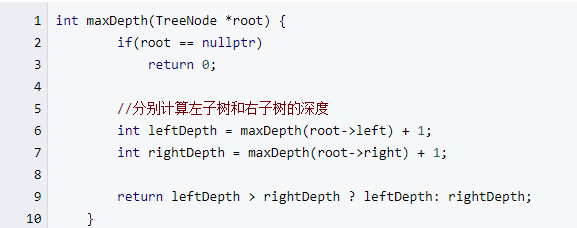


# **二叉树深度**



最大深度：

如果二叉树为空，则深度为0   
如果不为空，分别求左子树的深度和右子树的深度，取最大的再加1。



最小深度：

两种实现方法：

一种就是计算左子树和右子树深度的时候，判断是否等于0，如果等于0，说明该子树不存在，深度赋值为最大值。



第二种就是判断左子树或右子树是否为空，若左子树为空，则返回右子树的深度，反之返回左子树的深度，如果都不为空，则返回左子树和右子树深度的最小值。



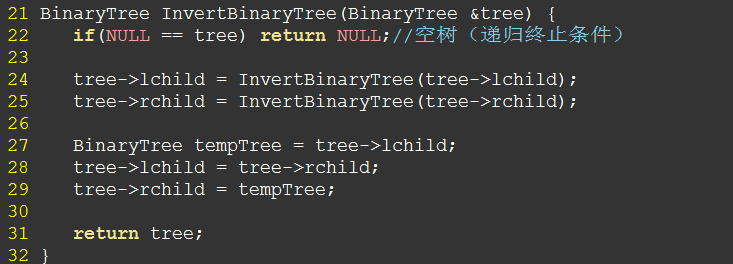
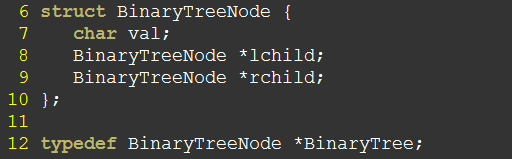
# **二叉树反转直接递归DFS**

递归算法步骤：

  1、对二叉树的左子树进行翻转

  2、对二叉树的右子树进行翻转

3、将左右子树的根节点进行交换（不用考虑原二叉树的根节点，因为原二叉树的根节点在翻转前后没有改变）



非递归（栈）

void NonRecursive\_Exchange(TreeNode \*T)

{

    Stack s;

    TreeNode \*p;

    if (NULL == T)

        return;

    InitStack(&s);

    Push(&s, T);

    while (!isEmpty(&s))

    {

        T = Pop(&s);

        p = T->left;

        T->left = T->right;

        T->right = p;

        if (T->right )

            Push(&s, T->right );

        if (T->left )

            Push(&s, T->left );

    }

    DestroyStack(&s);

}

# **二叉树的层序遍历dfs**

构造树：

struct Node

{

Node \*Left;

int Value;

Node \*Right;

Node(int value=0, Node \*left=NULL, Node \*right=NULL):Value(value) , Left(left), Right(right) {};

};

Node\* Root = node[10];

node[1]=new Node(1);

node[5]=new Node(5, node[1], node[2]);

## 深度优先搜索：

Node \*node[11];

void DFS(Node\* Root)

{

cout<<Root->Value<<" ";

if (Root->Left!=NULL)

DFS(Root->Left);

if (Root->Right!=NULL)

DFS(Root->Right);

return;

}

## 广度优先搜索：

void BFS(Node \*Root)

{

queue<Node\*> Q;//队列的声明

Node \* node ;

Q.push(Root);//先压入整棵树的根节点Root

while(!Q.empty())//如果队列不空则一直进行下去

{

node = Q.front();//访问队列的第一个元素

cout<<node->Value<<" ";//输出当前节点的值

if (node->Left!=NULL)

{

Q.push(node->Left); //如果左节点不为空则压入左节点

}

if (node->Right!=NULL)

{

Q.push(node->Right); //如果右节点不为空则压入右节点

}

Q.pop(); //弹出当前节点

}

cout<<endl;

}

# **二叉树前序遍历**

void PreOrder(BiTree T)//先序递归遍历

2 {

3 if(T!=NULL)

4 {

5 cout<<T->data<<" ";

6 PreOrder(T->lchild);

7 PreOrder(T->rchild);

8 }

9 }

10 void SqlPreOrder(BiTree T)//先序非递归遍历

11 {

12 stack<BiTree> s;

13 BiTree p=T;

14 while(p || !s.empty())

15 {

16 while(p)

17 {

18 **cout<<p->data<<" ";**

19 s.push(p);

20 p=p->lchild;

21 }

22 if(!s.empty())

23 {

24 p=s.top();

25 s.pop();

p=p->rchild;

27 }

28 }

29 }

# **二叉树中序遍历**

void InOrder(BiTree T)//中序递归遍历

34 {

35 if(T!=NULL)

36 {

37 InOrder(T->lchild);

38 cout<<T->data<<" ";

39 InOrder(T->rchild);

40 }

41 }

42 void SqInOrder(BiTree T)//中序非递归遍历

43 {

44 stack<BiTree> s;

45 BiTree p=T;

46 while(p || !s.empty())

47 while(p)

48 {

49 s.push(p);

50 p=p->lchild;

51 }

52 if(!s.empty())

53 {

54 p=s.top();

55 **cout<<p->data<<" ";**

56 s.pop();

57 p=p->rchild;

58 }

59 }

60

# **二叉树后序遍历**

void PostOrder(BiTree T)//后序递归遍历

64 {

65 if(T!=NULL)

66 {

67 PostOrder(T->lchild);

68 PostOrder(T->rchild);

69 cout<<T->data<<" ";

70 }

71 }

72

73 //后序非递归遍历1思路：因为后序非递归遍历二叉树的顺序是先访问左子树，再访问后子树，最后

74 //访问根结点。当用堆栈来存储结点，必须分清返回根结点时，是从左子树返回的，还是从右子树

75 //返回的。所以，**使用辅助指针r，其指向最近访问过的结点。**

76 void SqlPostOrder1(BiTree T)//后序非递归遍历1

77 {

78 stack<BiTree> s;

79 BiTree p=T,r;

80 while(p || !s.empty())

81 {

82 while(p) //走到最左边

83 {

84 s.push(p);

85 p=p->lchild;

86 }

87 if(!s.empty()) //向右

88 {

89 p=s.top();//取栈顶结点

90 if(p->rchild && p->rchild!=r)//如果右子树存在，且未被访问过

91 {

92 p=p->rchild;

93

95 }

96 else //否则，访问栈顶结点并弹出

97 {

98 cout<<p->data<<" ";

99 r=p; //记录该结点

100 s.pop();

101 p=NULL; }

103 }

104 }

105 }

106 //思路2：在结点中增加标志域，记录是否已被访问。

107 void SqlPostOrder2(BiTree T)//后序非递归遍历2

108 {

109 stack<BiTree> s;

110 BiTree p=T;

111 while(p || !s.empty())

112 {

113 if(p && p->lvisited==0) //左走，且左子树未被访问

114 {

115 p->lvisited=1;

116 s.push(p);

117 p=p->lchild;

118 }

119 else

120 {

121 p=s.top();

122 if(p->rchild!=NULL && p->rvisited==0)//右子树未被访问，右走一步

123 {

124 p->rvisited=1;

125 p=p->rchild;

126 }

127 else //访问栈顶元素并弹栈

128 {

129 cout<<p->data<<" ";

130 s.pop();

131 if(!s.empty())

132 p=s.top();

133 else //当最后一个元素弹栈出去后，结束

134 return ;

135 }

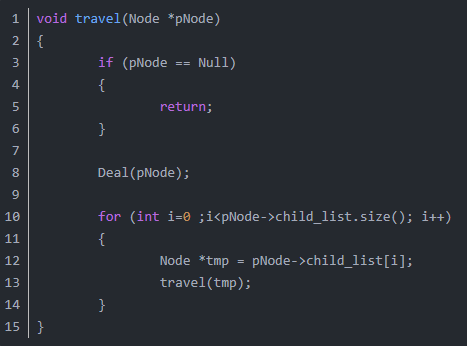
136 }

137 }

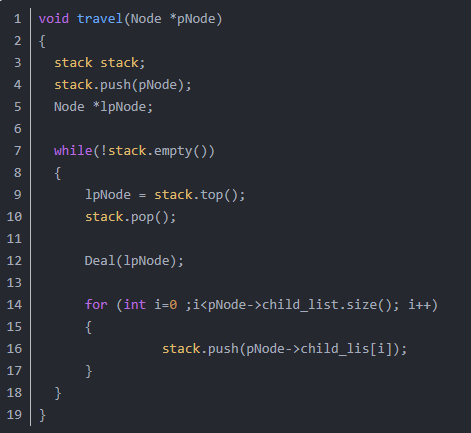
138 }

# **多叉树遍历**

1、递归方法DFS



2、非递归方法



# **哈弗曼树（最优二叉树）**

**从叶节点往根扫描，若为左子树则标记为0，为右子树则标记为1**

**https://blog.csdn.net/cqnuztq/article/details/8919204**

https://blog.csdn.net/u012532559/article/details/44589997

**哈夫曼树的存储结构：**

[cpp] [view plain](https://blog.csdn.net/u012532559/article/details/44589997) [copy](https://blog.csdn.net/u012532559/article/details/44589997)

1. <span style="font-size:18px;">/\*
2. 哈夫曼树的存储结构，它也是一种二叉树结构，
3. 这种存储结构既适合表示树，也适合表示森林。
4. \*/
5. typedef struct Node
6. {
7. int weight;                //权值
8. int parent;                //父节点的序号，为-1的是根节点
9. int lchild,rchild;         //左右孩子节点的序号，为-1的是叶子节点
10. }HTNode,\*HuffmanTree;          //用来存储哈夫曼树中的所有节点
11. typedef char \*\*HuffmanCode;    //用来存储每个叶子节点的哈夫曼编码</span>

**根据哈夫曼树的构建步骤，写出构建哈夫曼树的代码如下：**

[cpp] [view plain](https://blog.csdn.net/u012532559/article/details/44589997) [copy](https://blog.csdn.net/u012532559/article/details/44589997)

1. <span style="font-size:18px;">/\*
2. 根据给定的n个权值构造一棵赫夫曼树,wet中存放n个权值
3. \*/
4. HuffmanTree create\_HuffmanTree(int \*wet,int n)
5. {
6. //一棵有n个叶子节点的赫夫曼树共有2n-1个节点
7. int total = 2\*n-1;
8. HuffmanTree HT = (HuffmanTree)malloc(total\*sizeof(HTNode));
9. if(!HT)
10. {
11. printf("HuffmanTree malloc faild!");
12. exit(-1);
13. }
14. int i;
16. //以下初始化序号全部用-1表示，
17. //这样在编码函数中进行循环判断parent或lchild或rchild的序号时，
18. //不会与HT数组中的任何一个下标混淆
20. //HT[0],HT[1]...HT[n-1]中存放需要编码的n个叶子节点
21. for(i=0;i<n;i++)
22. {
23. HT[i].parent = -1;
24. HT[i].lchild = -1;
25. HT[i].rchild = -1;
26. HT[i].weight = \*wet;
27. wet++;
28. }
30. //HT[n],HT[n+1]...HT[2n-2]中存放的是中间构造出的每棵二叉树的根节点
31. for(;i<total;i++)
32. {
33. HT[i].parent = -1;
34. HT[i].lchild = -1;
35. HT[i].rchild = -1;
36. HT[i].weight = 0;
37. }
39. int min1,min2; //用来保存每一轮选出的两个weight最小且parent为0的节点
40. //每一轮比较后选择出min1和min2构成一课二叉树,最后构成一棵赫夫曼树
41. for(i=n;i<total;i++)
42. {
43. select\_minium(HT,i,min1,min2);
44. HT[min1].parent = i;
45. HT[min2].parent = i;
46. //这里左孩子和右孩子可以反过来，构成的也是一棵赫夫曼树，只是所得的编码不同
47. HT[i].lchild = min1;
48. HT[i].rchild = min2;
49. HT[i].weight =HT[min1].weight + HT[min2].weight;
50. }
51. return HT;
52. }</span>

**上述代码中调用到了select\_minium（）函数，它表示从集合中选出两个最小的二叉树，下面列出两种求两个最小值的方法。**

1. /\*
2. 从HT数组的前k个元素中选出weight最小且parent为-1的两个，分别将其序号保存在min1和min2中
3. \*/
4. void select\_minium(HuffmanTree HT,int k,int &min1,int &min2)
5. {
6. int i = 0;
7. Int tempMin;
8. while(HT[i].parent != -1)
9. i++;
10. min1 = HT[i].weight;
11. while(HT[i].parent != -1)
12. i++;
13. min2 = HT[i].weight;
14. if(min1>min2)
15. {
16. tempMin = min1;
17. min1=min2;
18. Min2=tempMin;
19. }
20. //选出weight最小且parent为-1的元素，并将其序号赋给min
21. for(;i<k;i++)
22. {
23. if(HT[i].weight<min1 && HT[i].parent==-1)//min1始终保存最小的，min2保存第二小的数据
24. {
25. min2 = min1;
26. min1 = HT[i].weight;
27. }else if(HT[i].weight<min2 && HT[i].parent==-1)
28. {
29. Min2 = HT[i].weight;
30. }
31. }
32. }

# **红黑树**

<https://blog.csdn.net/sun_tttt/article/details/65445754>

红黑树是在普通二叉树上，对没个节点添加一个颜色属性形成的，同时整个红黑二叉树需要同时满足一下五条**性质** ：**性质一：节点是红色或者是黑色；性质二：根节点是黑色；** **性质三：每个叶节点（NIL或空节点）是黑色；** **性质四：每个红色节点的两个子节点都是黑色的（也就是说不存在两个连续的红色节点）；性质五：从任一节点到其没个叶节点的所有路径都包含相同数目的黑**色节点；

旋转的目的是将节点多的一支出让节点给另一个节点少的一支，旋转操作在插入和删除操作中经常会用到。

下面是可能遇到的插入的几种状况：   
1、当插入的节点是根节点时，直接涂黑即可；   
2、当要插入的节点的父节点是黑色的时候。 这个时候插入一个红色的节点并没有对这五个性质产生破坏。所以直接插入不用在进行调整操作。

3、如果要插入的节点的父节点是红色且父节点是祖父节点的左支的时候。

这个要分两种情况，一种是叔叔节点为黑的情况，一种是叔叔节点为红的情况。   
当叔叔为黑时，也分为两种情况，一种是要插入的节点是父节点的左支，另一种是要插入的节点是父亲的右支。 

# **----------------------------------------**

# 两个队列实现栈



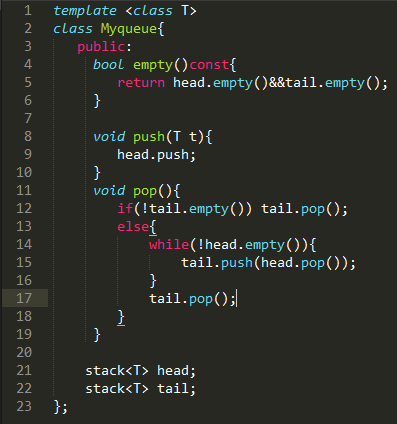
# 两个栈实现队列

入队时，将元素压入s1。

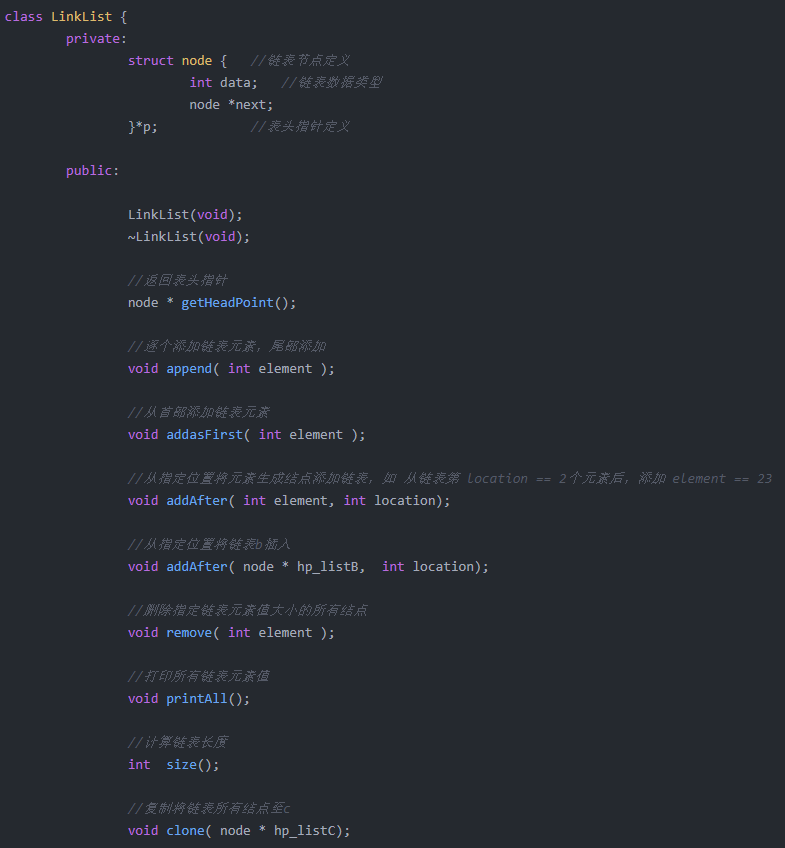
出队时，判断s2是否为空，如不为空，则直接弹出顶元素；如为空，则将s1的元素逐个“倒入”s2，把最后一个元素弹出并出队。

这个思路，避免了反复“倒”栈，仅在需要时才“倒”一次。

2个栈都为空的时候，出队操作一定会引起异常

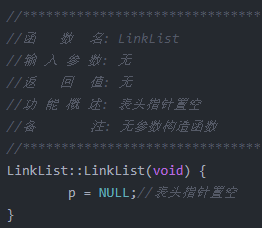


# 定义链表

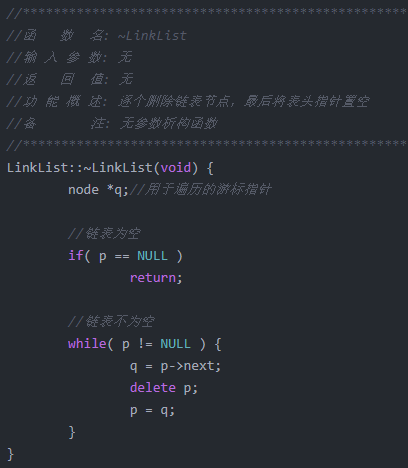




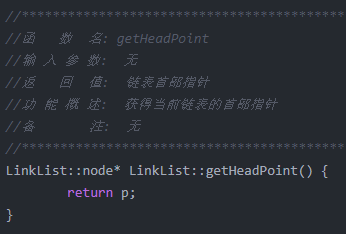
## 表头指针置空



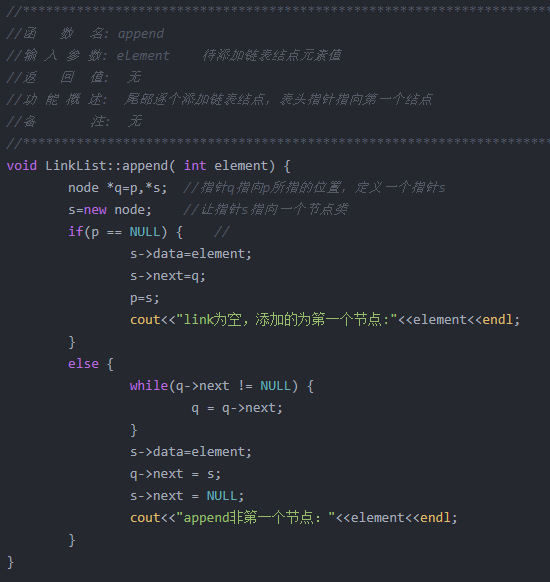
## 逐个删除链表节点，然后将表头置空



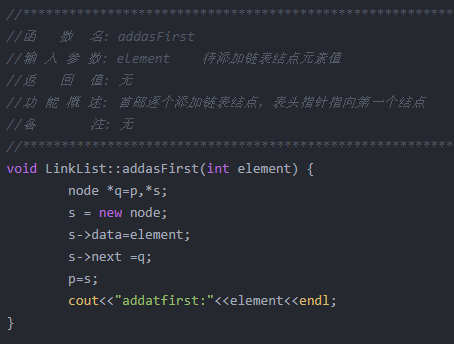
## 获得表头



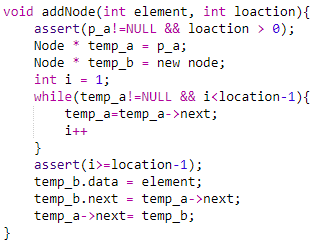
## 尾部添加节点

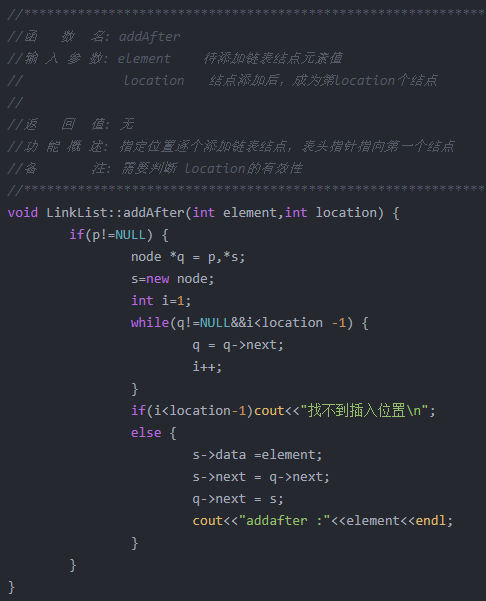


## 头部添加节点

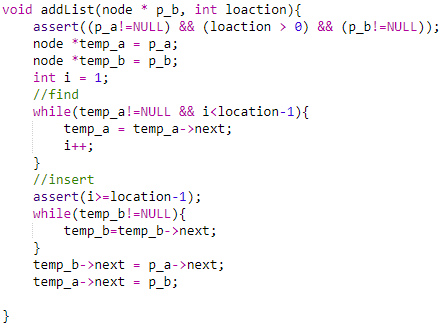


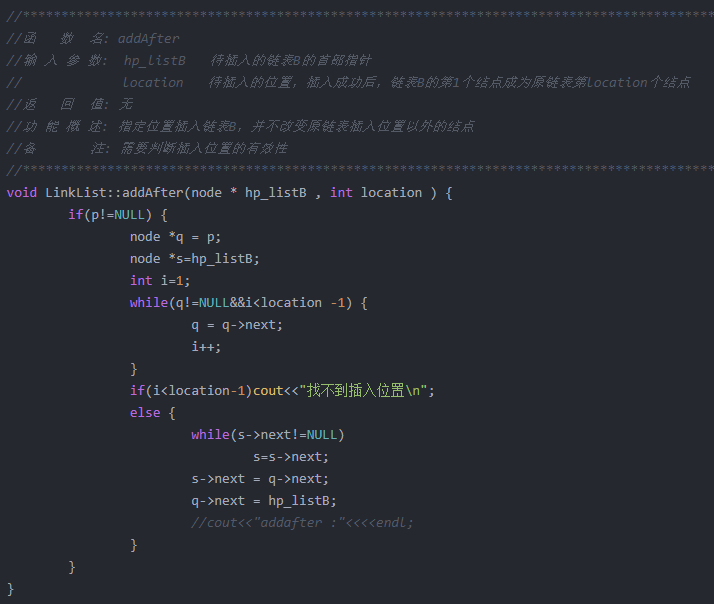
## 指定位置添加节点



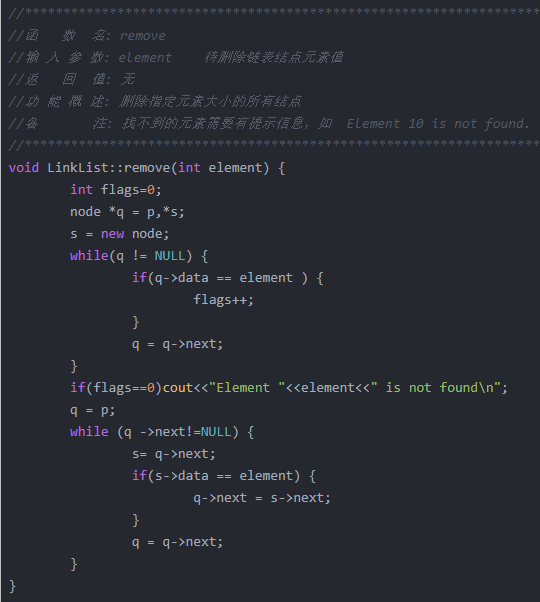


## 指定位置插入链表

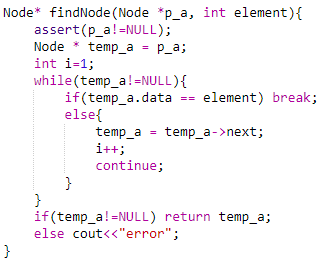




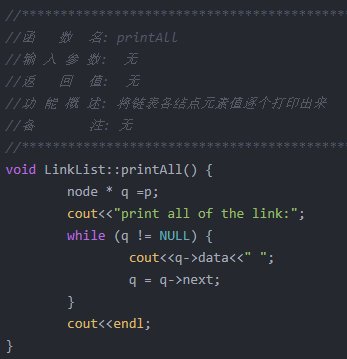
## 删除指定值的节点



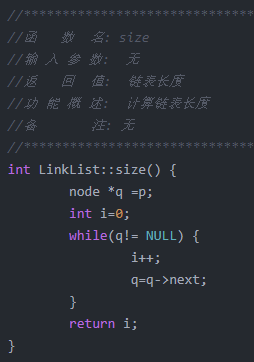
## 查找节点



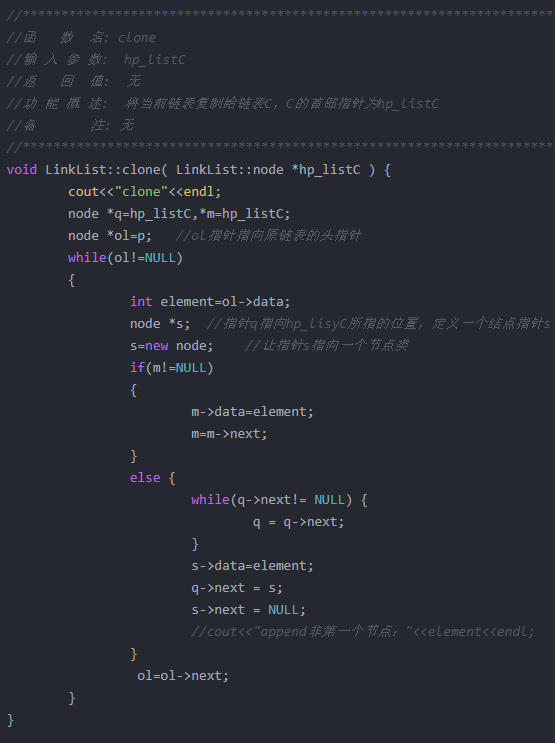
## 打印链表



## 计算长度



## 复制链表

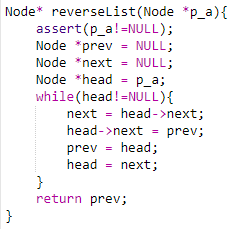


# 判断两个单链表是否相交

如果相交，给出相交的第一个点（两个链表都不存在环）。  
  
比较好的方法有两个：  
  
一、将其中一个链表首尾相连，检测另外一个链表是否存在环，如果存在，则两个链表相交，而检测出来的依赖环入口即为相交的第一个点。  
  
二、如果两个链表相交，那个两个链表从相交点到链表结束都是相同的节点，我们可以先遍历一个链表，直到尾部，再遍历另外一个链表，如果也可以走到同样的结尾点，则两个链表相交。  
  
这时我们记下两个链表length，再遍历一次，长链表节点先出发前进(lengthMax-lengthMin)步，之后两个链表同时前进，每次一步，相遇的第一点即为两个链表相交的第一个点。

# **单**链表**逆序**

初始状态，prev是NULL，head指向当前的头节点A，next指向A节点的下一个节点B。首先从A节点开始逆序，将A节点的next指针指向prev，因为prev的当前值是NULL，所以A节点就从链表中脱离出来了，然后移动head和next指针，使它们分别指向B节点和B的下一个节点C（因为当前的next已经指向B节点了，因此修改A节点的next指针不会导致链表丢失）。



# 链表的中间节点

思路：   
（1）首先求解单链表的长度length，然后遍历 length / 2 的距离即可查到单链表的中间节点，但一般此种方法需要遍历两次链表，第一次遍历求解单链表的长度，第二次遍历根据索引获取中间节点。   
（2）如果是双向链表，可以首尾并行，利用两个指针一个从头到尾，一个从尾到头，当两个指针相遇的时候，就找到中间元素。

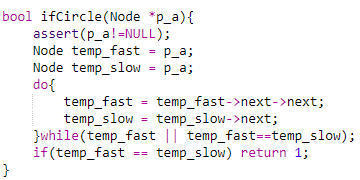
以此思想为基础，进一步思考，单链表也可以采用双指针的方式来实现中间节点的快速查找。但双指针的使用和双向链表不同，具体如下：   
<1> 有2个指针同时从头开始遍历；   
<2> 一个快指针一次走2步，一个慢指针一次走一步。   
<3> 快指针先到链表尾部，而慢指针则恰好到达链表中部；   
快指针到达链表尾部，当链表长度为奇数时，慢指针指向的就是链表中间节点。当链表长度为偶数时，慢指针指向的节点和慢指针指向节点的下一个节点都是链表的中间节点。

1. **while** (fast&&fast->next )
2. {
3. fast = fast->next->next;
4. slow = slow->next;
5. }

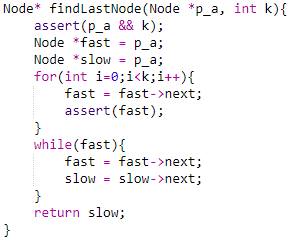
# **链表成环**

设置两个指针p1, p2，每次p1向前走一步，p2向前走两步，直到p2碰到NULL或两个指针相等时结束循环。

如果两个指针相等，则说明该链表存在环。



# 链表倒数第k个节点



# 合并两个有序链表

递归：

1.对空链表存在的情况进行处理，假如pHead1为空则返回pHead2，pHead2为空则返回pHead1。（两个都为空此情况在pHead1为空已经被拦截）

2.比较两个链表第一个结点的大小 确定 头结点的位置

3.头结点确定后 继续在剩下的结点中 选出下一个结点去链接到第二步选出的结点后面，然后在继续重复2 3步 直到有链表为空

PNode MergeTwoOrderedLists(PNode pHead1, PNode pHead2)

{

    PNode newHead = NULL;

    if (NULL == pHead1)

    {

        return pHead2;

    }

    else if(NULL ==pHead2)

    {

        return pHead2;

    }

    else

    {

        if (pHead1->data < pHead2->data)

        {

            newHead = pHead1;

            newHead->next = MergeTwoOrderedLists(pHead1->next, pHead2);

        }

        else

        {

            newHead = pHead2;

            newHead->next = MergeTwoOrderedLists(pHead1, pHead2->next);

        }

        return newHead;

    }

}

非递归：

1.第一步与递归一样，对空链表存在的情况进行处理，假如pHead1为空则返回pHead2，pHead2为空则返回pHead1。（两个都为空此情况在pHead1为空已经被拦截）

2.在两个链表无空链表的情况下确定第一个结点，比较链表1和链表2的第一个结点的值，将值小的结点保存下来为合并后的第一个结点。并且把第一个结点为最小的链表向后移动一个元素。

3.继续在剩下的元素中选择小的值，连接到第一个结点后面，并不断next将值小的结点连接到第一个结点后面，直到某一个链表为空。

4.当两个链表长度不一致时，也就是比较完成后其中一个链表为空，此时需要把另外一个链表剩下的元素都连接到第一个结点的后面。

PNode MergeTwoOrderedLists(PNode pHead1, PNode pHead2)

{

    PNode pTail = NULL;//指向新链表的最后一个结点 pTail->next去连接

    PNode newHead = NULL;//指向合并后链表第一个结点

    if (NULL == pHead1)

    {

        return pHead2;

    }

    else if(NULL == pHead2)

    {

        return pHead1;

    }

    else

    {

        //确定头指针

        if ( pHead1->data < pHead2->data)

        {

            newHead = pHead1;

            pHead1 = pHead1->next;  //指向链表的第二个结点

        }

        else

        {

            newHead = pHead2;

            pHead2 = pHead2->next;

        }

        pTail = newHead;    //指向第一个结点

        while ( pHead1 && pHead2)

        {

            if ( pHead1->data <= pHead2->data )

            {

                pTail->next = pHead1;

                pHead1 = pHead1->next;

            }

            else

            {

                pTail->next = pHead2;

                pHead2 = pHead2->next;

            }

            pTail = pTail->next;

        }

        if(NULL == pHead1)

        {

            pTail->next = pHead2;

        }

        else if(NULL == pHead2)

        {

            pTail->next = pHead1;

        }

        return newHead;

}

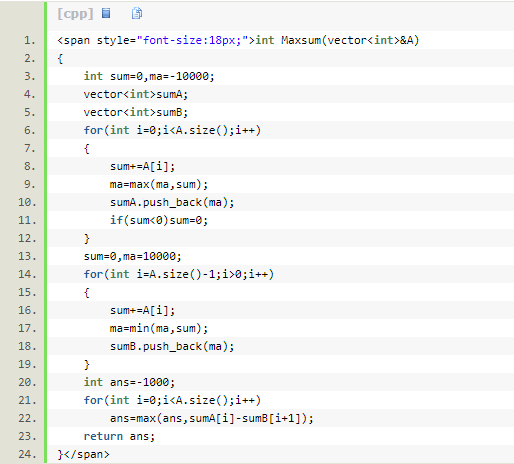
# ----------------------------------------

# 寻找二维数组的鞍点

该行上最大,在该列上最小

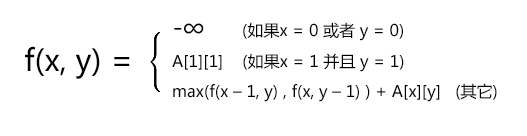
1. **int** main()
2. {
3. **int** arr[n][m];
4. **int** i,j,jmax,k;
5. **int** flag=1,count=0;
6. //输入数组元素
7. **for**(**int** i=0;i<n;i++){
8. **for**(**int** j=0;j<m;j++){
9. scanf("%d",&arr[i][j]);
10. }
11. }
12. //判断
13. **for**(i=0;i<n;i++){
14. **int** max=arr[i][0];
15. **for**(j=0;j<m;j++){
16. **if**(arr[i][j]>=max){
17. max=arr[i][j];
18. jmax=j;             //求出第i行最大值max位于第几列,并记录
19. }
20. }
21. flag=1;                     //默认flag=1,该点为鞍点
22. **for**(k=0;k<n;k++){
23. **if**(arr[k][jmax]<max){
24. flag=0;
25. **break**;              //若第j列中存在比max小的数,则该点不是鞍点,flag=0
26. }
27. }
28. **if**(flag){
29. printf("%d %d\n",i,jmax);
30. count++;                //count计算鞍点个数
31. }
32. }
33. **if**(count==0) printf("None");        //若count=0,无鞍点时输出none
34. **return** 0;
35. }

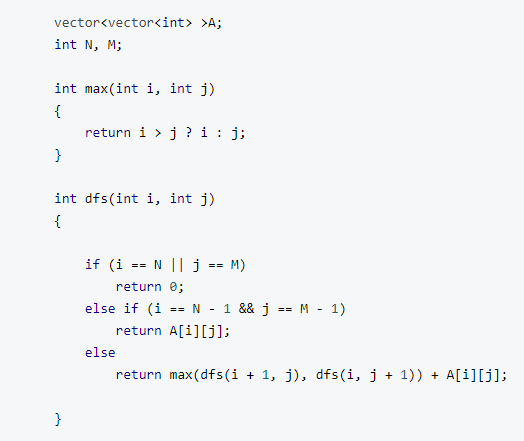
# Max(Sum(A)-Sum(B))

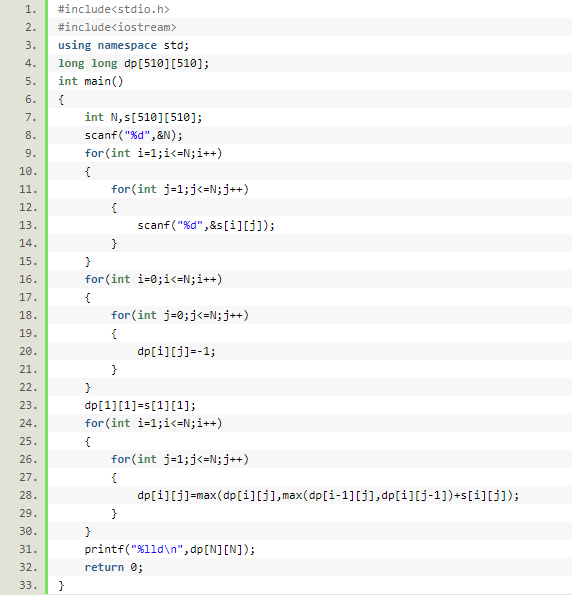


# 矩阵走一遍能够取到的最大分值

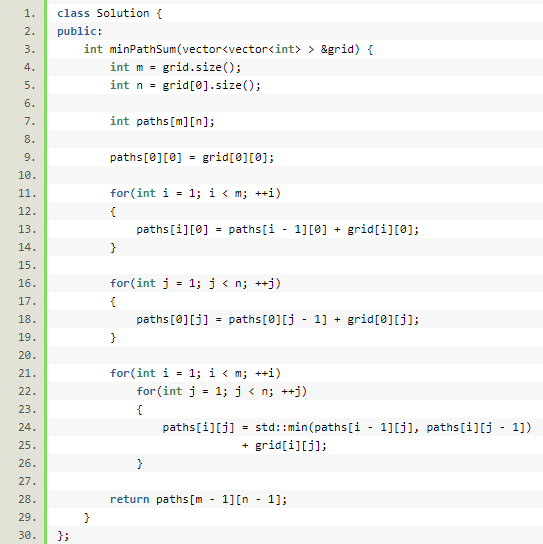
一个矩阵上每一点都有分值，从左上角走到右下角，只能往下或者往右，问走一遍能够取到的最大分值





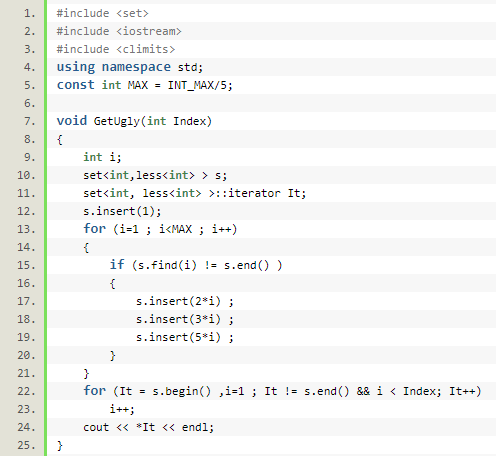


最小值：

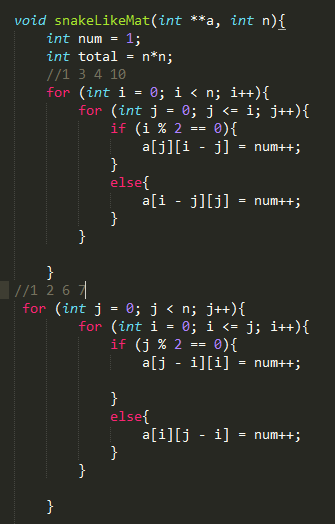


问走两遍能够取到的最大分值

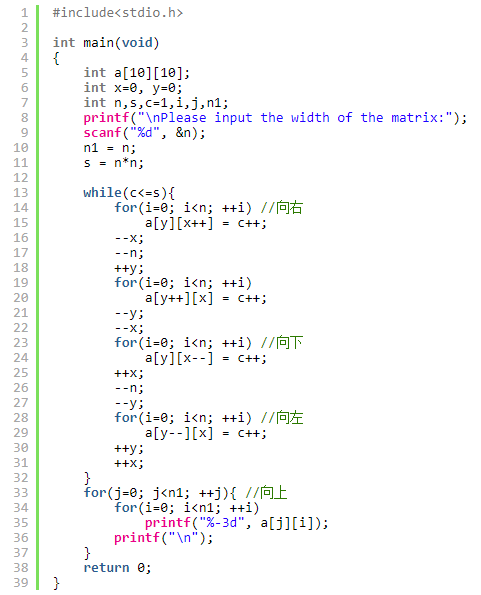
# 丑数



# 蛇形矩阵

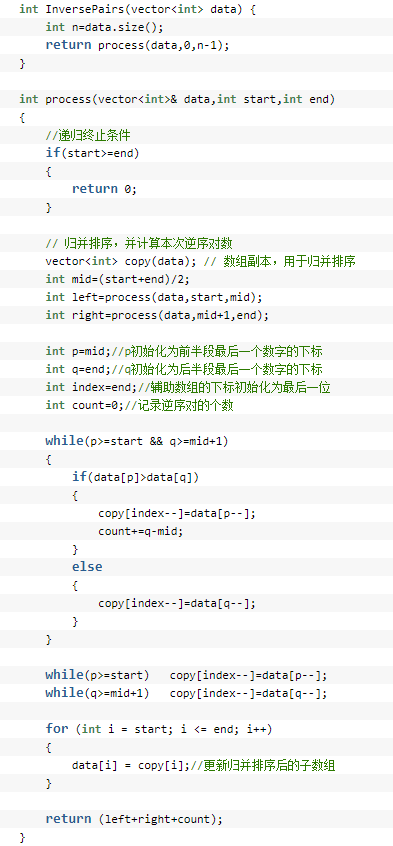


# 螺旋矩阵



# 数组中的逆序对

归并，只不过归并是从前往后比较大小；逆序对是从后往前，如果第一部分的最后一个大于第二部分的最后一个，则count加第二部分全部个数；然后比较第一部分倒数第二个….



# 求两个数组中和为定值的组合

先排序，然后头尾指针遍历

# 求数组中和为m的所有子序列

# 寻找和为定值的多个数

注意到取n，和不取n个区别即可，考虑是否取第n个数的策略，可以转化为一个只和前n-1个数相关的问题。

首先定义一个标志位数组flag[]，flag[i]如果为true，则表示a[i]在当前解中，如果flag[i]为false则表示不在。这个数组元素个数与数组a的元素个数相同。

bool flag[100] = { false };

//输出一种组合，该组合有n个元素  
void Output(int\* a, int n)  
{  
    for(int i = 0; i < n; ++i)  
    {  
        if(flag[i])  
            cout << a[i] << ", " ;  
    }  
    cout << endl ;  
}

// a: 待搜索的数组  
// n: 数组元素个数  
// t: 已经存储的元素个数  
// sum: 给定的和  
void FixedSum(int\* a, int n, int t, int sum)  
{  
   if(sum == 0)  
        Output(a, t) ;  
    else  
    {  
        if(t == n)  
            return ;  
        else  
        {  
            flag[t] = true ;  
            if(sum - a[t] >= 0)  
                FixedSum(a, n, t + 1, sum - a[t]) ;  
            flag[t] = false ;  
            if(sum >= 0)  
                FixedSum(a, n, t + 1, sum) ;  
        }  
    }  
}

解法一：

如果取第n个数，那么问题就转化为“取前n-1个数使得它们的和为sum-n”，对应的代码语句就是sumOfkNumber(sum - n, n - 1)；如果不取第n个数，那么问题就转化为“取前n-1个数使得他们的和为sum”，对应的代码语句为sumOfkNumber(sum, n - 1)。

list<int>list1;

void SumOfkNumber(int sum, int n)

{

// 递归出口

if (n <= 0 || sum <= 0)

return;

// 输出找到的结果

if (sum == n)

{

// 反转list

list1.reverse();

for (list<int>::iterator iter = list1.begin(); iter != list1.end(); iter++)

cout << \*iter << " + ";

cout << n << endl;

list1.reverse()//此处还需反转回来

}

list1.push\_front(n); //典型的01背包问题

SumOfkNumber(sum - n, n - 1); //“放”n，前n-1个数“填满”sum-n

list1.pop\_front();

SumOfkNumber(sum, n - 1); //不“放”n，n-1个数“填满”sum

}

解法二：

本程序采用回溯法+剪枝，其中X数组是解向量，t=∑(1,..,k-1)Wi\*Xi, r=∑(k,..,n)Wi，且

* 若t+Wk+W(k+1)<=M,则Xk=true，递归左儿子(X1,X2,..,X(k-1),1)；否则剪枝；
* 若t+r-Wk>=M && t+W(k+1)<=M,则置Xk=0，递归右儿子(X1,X2,..,X(k-1),0)；否则剪枝；

本题中W数组就是(1,2,..,n),所以直接用k代替WK值。

void SumOfkNumber(int t, int k, int r, int& M, bool& flag, bool\* X)

{

X[k] = true; // 选第k个数

if (t + k == M) // 若找到一个和为M，则设置解向量的标志位，输出解

{

flag = true;

for (int i = 1; i <= k; ++i)

{

if (X[i] == 1)

{

printf("%d ", i);

}

}

printf("\n");

}

else

{ // 若第k+1个数满足条件，则递归左子树

if (t + k + (k + 1) <= M)

{

SumOfkNumber(t + k, k + 1, r - k, M, flag, X);

}

// 若不选第k个数，选第k+1个数满足条件，则递归右子树

if ((t + r - k >= M) && (t + (k + 1) <= M))

{

X[k] = false;

SumOfkNumber(t, k + 1, r - k, M, flag, X);

}

}

}

void search(int& N, int& M)

{

// 初始化解空间

bool\* X = (bool\*)malloc(sizeof(bool)\* (N + 1));

memset(X, false, sizeof(bool)\* (N + 1));

int sum = (N + 1) \* N \* 0.5f;

if (1 > M || sum < M) // 预先排除无解情况

{

printf("not found\n");

return;

}

bool f = false;

SumOfkNumber(0, 1, sum, M, f, X);

if (!f)

{

printf("not found\n");

}

free(X);

}

# ----------------------------------------

# KMP字符串模式匹配O(m+n)

对于模式串来说，我们会提前计算出每个匹配失败的位置应该移动的距离，花费的时间就成了常数时间。

Next[]怎么求：

（1）next[0] = -1   意义：任何串的第一个字符的模式值规定为 -1。

（2）next[j] = -1    意义：模式串T中下标为j的字符，如果与首字符

         相同，且j的前面的1—k个字符与开头的1—k

         个字符不等（或者相等但T[k] == T[j]）（1 ≤ k < j）。

         如：T = ”abCabCad” 则 next[6] = -1，因T[3] = T[6]

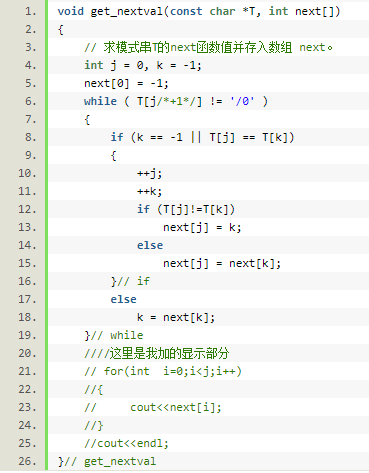
（3）next[j] = k     意义： 模式串T中下标为j的字符，如果j的前面k个

         字符与开头的k个字符相等，且T[j] != T[k] （1 ≤ k < j）。

         即T[0]T[1]T[2]。。。T[k-1] == T[j-k]T[j-k+1]T[j-k+2]…T[j-1] 且T[j] != T[k].（1 ≤ k < j）;

（4）next[j] = 0     意义： 除（1）（2）（3）的其他情况。





# 输出相邻字符不相同的字符串

给你一个字符串：“aabb cccg h iii”, 如何遍历一遍输出相邻字符不相同的字符串。

如果上述的结果是“ab cg h i”

STL set容器：相同的值不存，存进去自动排序好了。

1. **int** main()
2. {
3. set<**int**> s;
4. s.insert(5); //第一次插入5，可以插入
5. s.insert(1);
6. s.insert(6);
7. s.insert(3);
8. s.insert(5); //第二次插入5，重复元素，不会插入
9. set<**int**>::iterator it; //定义前向迭代器
10. //中序遍历集合中的所有元素
11. **for**(it = s.begin(); it != s.end(); it++)
12. {
13. cout << \*it << " ";
14. }
15. cout << endl;
16. **return** 0;
17. }
18. //运行结果：1 3 5 6

# 字符串翻转

reverse(s.begin(), s.end())之后，s =“! dlrow olleh”

"hello world"翻转成"world hello"

void reverse(string &s, int start, int end){//hello world -> dlrow olleh

int len=end+start;

int center=len/2;

for(int i=start;i<center;i++){

swap(s[i],s[len-1-i]);

}

}

void reverseWords(string &s) {

if(s=="")

return;

reverse(s,0,s.length());//全部反转

int start=0;

int end=start;

int len=s.length();

while(end<len){

while(start<len&&s[start]==' '){

start++;

}

end=start;

while(end<len&&s[end]!=' '){

end++;

}

reverse(s,start,end);//翻转每个单词，使之字母序恢复原状

start=end+1;

}

}

# 最长公共子字符串（字符连续）

1. **int** \*\*c = **new** **int**\*[length1+1];
2. **for**(**int** i = 0; i < length1+1; i++){
3. c[i] = **new** **int**[length2+1];
4. c[i][0]=0;        //第0列初始化为0
5. }
6. **for**(**int** j = 0; j<length2+1; j++)
7. c[0][j]=0;        //第0行初始化为0
8. **int** maxI=0;//当前最长公共子字符串的位置
9. **int** length=0;//最长公共子字符串的长度
10. **for**(**int** i = 1; i < length1+1; i++)
11. {
12. **for**(**int** j = 1; j < length2+1; j++)
13. {
14. **if**(str1[i-1]==str2[j-1])//由于c[][]的0行0列没有使用，c[][]的第i行元素对应str1的第i-1个元素
15. c[i][j]=c[i-1][j-1]+1;  //加上其斜对角上的累计值，表明需要连续
16. **else**
17. c[i][j]=0;
19. **if**(c[i][j]>length){
20. length=c[i][j];
21. maxI=i;
22. }
23. }
24. }
25. cout<<"最长公共子字符串为---";
26. **for**(**int** i=0;length&&i<length;i++)
27. cout<<str1[maxI-length+i];

# 字符串回文

 for(int i=0;i<strlen(s)/2;i++)

        if(s[i]!=s[strlen(s)-i-1])

        {

            b++;

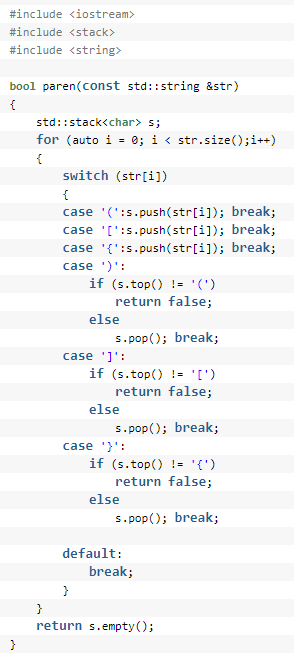
            break;

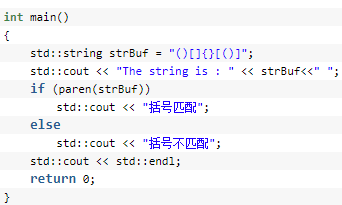
        }

# 判断字符串是否含有回文

# 括号匹配问题

遇到一个左括号就压栈，遇到一个右括号就弹栈，看是否匹配就好了。最后检查下栈是否为空就行了





# 字符串最长公共前缀

字典树

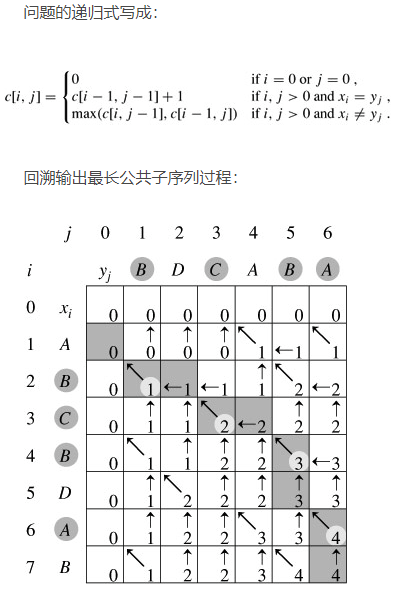
# 返回字符串里不重复第一次出现的子串

功能测试（存在，不存在，所有子串都只出现一次）

特殊输入测试（字符串为nullptr指针）

# 求两个字符串的最大交集，输出这个字符串和其长度

# 最长公共子序列（字符不连续）



 由于每次调用至少向上或向左（或向上向左同时）移动一步，故最多调用(m + n)次就会遇到i = 0或j = 0的情况，此时开始返回。返回时与递归调用时方向相反，步数相同，故算法时间复杂度为Θ(m + n)。

子序列：

1. **void** PrintLCS(**int** \*\*b, **char** \*str1, **int** i, **int** j)
2. {
3. **if**(i==0 || j==0)
4. **return** ;
5. **if**(b[i][j]==0)
6. {
7. PrintLCS(b, str1, i-1, j-1);   //从后面开始递归，所以要先递归到子串的前面，然后从前往后开始输出子串
8. printf("%c",str1[i-1]);        //c[][]的第i行元素对应str1的第i-1个元素
9. }
10. **else** **if**(b[i][j]==1)
11. PrintLCS(b, str1, i-1, j);
12. **else**
13. PrintLCS(b, str1, i, j-1);
14. }

长度c[length1][length2]：

1. //双指针的方法申请动态二维数组
2. **int** \*\*b = **new** **int**\*[length1+1];
3. **for**(i= 0; i < length1+1; i++)
4. b[i] = **new** **int**[length2+1];
5. len=LCSLength(str1,str2,b);
6. printf("最长公共子序列的长度为：%d\n",len);
7. //双指针的方法申请动态二维数组
8. **int** \*\*c = **new** **int**\*[length1+1];      //共有length1+1行
9. **for**(i = 0; i < length1+1; i++)
10. c[i] = **new** **int**[length2+1];      //共有length2+1列
12. **for**(i = 0; i < length1+1; i++)
13. c[i][0]=0;        //第0列都初始化为0
14. **for**(j = 0; j < length2+1; j++)
15. c[0][j]=0;        //第0行都初始化为0
17. **for**(i = 1; i < length1+1; i++)
18. {
19. **for**(j = 1; j < length2+1; j++)
20. {
21. **if**(str1[i-1]==str2[j-1])   //由于c[][]的0行0列没有使用，c[][]的第i行元素对应str1的第i-1个元素
22. {
23. c[i][j]=c[i-1][j-1]+1;
24. b[i][j]=0;          //输出公共子串时的搜索方向
25. }
26. **else** **if**(**c[i-1][j]>c[i][j-1]**)  //行列指针都往前移一个
27. {
28. c[i][j]=**c[i-1][j]**;  //选择更长的那个长度累计
29. b[i][j]=1;
30. }
31. **else**
32. {
33. c[i][j]=**c[i][j-1]**;
34. b[i][j]=-1;
35. }
36. }
37. }
38. len=c[length1][length2];

# 字符串中求最长数字子序列的长度

1. unsigned **int** continumax(**char** \* inputstr,**char**\* outputstr)
2. {
3. **int** len=strlen(inputstr);
4. **int** flag=1,cnt=0,max=0,index;
5. **for**(**int** i=0;i<len;i++)
6. {
7. **if**(isdigit(inputstr[i]))
8. {
9. flag=1;
10. cnt++;
11. **if**(cnt>max)
12. {
13. max=cnt;
14. index=i;
15. }
16. }
17. **else** **if**(!isdigit(inputstr[i]))
18. {
19. flag=0;
20. cnt=0;
21. }
22. }
23. **for**(**int** i=0;i<max;i++)
24. outputstr[max-1-i]=inputstr[index-i];
25. **return** max;
26. }

# 最长递增子序列

比如8个整数的序列 186 186 **150** 200 **160** 130 **197 200**，最长递增子序列是 150 160 197 200, 长度为4。

int findLoogestIncreaseSeq(vector<int> &vect)

{

int len = 0;

int \*count = new int[vect.size()];

for (int i = 0; i < vect.size(); i++)

count[i] = 1;

for (int i = 0; i < vect.size(); i++)

{

for (int j = i - 1; j >= 0; j--)

{

if (vect[j] < vect[i] && count[j] >= count[i])

{

count[i] = count[j] + 1;

}

}

if (count[i] > len)

len = count[i];

}

delete [] count;

return len;

}

// 动态规划算法O(nlogn)   
int main()   
{   
    int n, i, j, k;   
    cin >> n;   
    for (i=1; i<=n; i++)   
        cin >> A[i];   
    MaxV[1] = A[1];   
    int nmaxv = 1;  // 目前找到的最长递增子序列的长度   
    // 有n个阶段，每个阶段有1个状态   
    for (i=2; i<=n; i++)   
    {   
        // 每个状态有nmaxv种决策，以得出以元素i结尾的最长递归子序列的长度   
        int u = 1, v = nmaxv;   
        while (u<=v)   
        {   
            int mid = (u+v)>>1;   
            if (MaxV[mid] < A[i])   
                u = mid+1;   
            else   
                v = mid-1;   
        }   
   
        // 每个元素都会对数组MaxV中的某个元素产生影响   
        // 使用二分搜索确定其在第v+1个位置   
        nmaxv = max(nmaxv, v+1);   
        MaxV[v+1] = A[i];   
    }   
    cout << nmaxv;   
}

# 连续和最大的区间

对这个问题，有一个相对复杂的O(NlogN)的解法，就是使用递归。如果要是求出序列的位置的话，这将是最好的算法了（因为我们后面还会有个O(N)的算法，但是不能求出最大子序列的位置）。该方法我们采用“分治策略”（pide-and-conquer）。

在我们例子中，最大子序列可能在三个地方出现，或者在左半部，或者在右半部，或者跨越输入数据的中部而占据左右两部分。前两种情况递归求解，第三种情况的最大和可以通过求出前半部分最大和（包含前半部分最后一个元素）以及后半部分最大和（包含后半部分的第一个元素）相加而得到。

//递归法，复杂度是O(nlogn)

long maxSumRec(const vector<int>&a, int left, int right)  
{  
       if (left== right)  
       {  
              if (a[left]> 0)  
                     return a[left];  
              else  
                     return 0;  
       }  
       int center= (left + right) / 2;  
       long maxLeftSum= maxSumRec(a, left, center);  
       long maxRightSum= maxSumRec(a, center+1, right);  
   
       //求出以左边最后一个数字结尾的序列最大值  
       long maxLeftBorderSum= 0, leftBorderSum = 0;  
       for (int i= center; i >= left; i--)  
       {  
              leftBorderSum+= a[i];  
              if (leftBorderSum> maxLeftBorderSum)  
                     maxLeftBorderSum= leftBorderSum;  
       }  
   
       //求出以右边最后一个数字结尾的序列最大值  
       long maxRightBorderSum= 0, rightBorderSum = 0;  
       for (int j= center+1; j <= right; j++)  
       {  
              rightBorderSum+= a[j];  
              if (rightBorderSum> maxRightBorderSum)  
                     maxRightBorderSum= rightBorderSum;  
       }  
   
       return max3(maxLeftSum,maxRightSum,   
              maxLeftBorderSum+ maxRightBorderSum);  
}  
   
long maxSubSum3(const vector<int>&a)  
{  
       return maxSumRec(a,0, a.size()-1);  
}  
另外max3(long,long,long)表示求三个long中的最大值：  
//求出三个long中的最大值  
long max3(long a, long b, long c)  
{  
       if (a< b)  
       {  
              a= b;  
       }  
       if (a> c)  
              return a;  
       else  
              return c;  
}

# 连续子序列的最大和

 重点的一个思想是：如果a[i]是负数那么它不可能代表最有序列的起点，因为任何包含a[i]的作为起点的子序列都可以通过用a[i+1]作为起点来改进。类似的有，任何的负的子序列不可能是最优子序列的前缀。例如说，循环中我们检测到从a[i]到a[j]的子序列是负数，那么我们就可以推进i。

关键的结论是我们不仅可以把i推进到i+1，而且我们实际可以把它一直推进到j+1。O(N)

long maxSubSum4(const vector<int>&a)  
{  
       long maxSum= 0, thisSum = 0;  
       for (int j= 0; j < a.size(); j++)  
       {  
              thisSum+= a[j];  
              if (thisSum> maxSum)  
                     maxSum= thisSum;  
              else if (thisSum< 0)  
                     thisSum= 0;  
       }  
       return maxSum;  
}

# 某个数出现的次数大于n/2次，求该数

我们从前往后遍历，设key为第一个数，key出现的次数为ntime，初始化为1，代表key出现了一次，从前往后，如果某个数不等于key，则他俩抵消，key的出现次数减一，如果等于key，则key的出现次数加1，如果key的出现次数变成了0，则说明key已经用完了，所以需要重新初始化key为另一个数，再重复以上步骤，因为一定有一个数大于n/2，所以遍历到最后剩下的那个数，就是要求的数。

1. /\*在大小为n的数组中寻找次数超过n/2的数\*/
2. **int** find\_data(vector<**int**> &arry)
3. {
4. **int** ntime = 0; //表示其中某一个数出现的次数
5. **int** result;
6. **for**(unsigned **int** i = 0; i < arry.size(); i++) {
7. **if**(ntime == 0) {   //在i前面的数全部删除完，或者起始的时候，将arry[i]放入结果
8. result = arry[i];
9. ntime = 1;  //arry[i]出现的次数为1；
10. } **else** { //如果前面有数，就说明result还没抵消完
11. **if**(result == arry[i]) //如果相等result出现的次数+1
12. ntime++;
13. **else**
14. ntime--;  /\*如果此时的arry[i]不等于result，则它们两个抵消，result的次数减一，由与那个数大于n/2所以它抵消不完，ntime最小为1
15. 也就是说这个数出现的次数是大于等于n/2+1\*/
16. }
17. }
18. **return** result;
19. }

# ----------------------------------------

# 动态规划和分治法

**分治法适用的情况**

    分治法所能解决的问题一般具有以下几个特征：

    1) 该问题的规模缩小到一定的程度就可以容易地解决

    2) 该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质。

    3) 利用该问题分解出的子问题的解可以合并为该问题的解；

    4) 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子子问题。

第一条特征是绝大多数问题都可以满足的，因为问题的计算复杂性一般是随着问题规模的增加而增加；

第二条特征是应用分治法的前提它也是大多数问题可以满足的，此特征反映了递归思想的应用；、

第三条特征是关键，能否利用分治法完全取决于问题是否具有第三条特征，如果具备了第一条和第二条特征，而不具备第三条特征，则可以考虑用贪心法或动态规划法。

第四条特征涉及到分治法的效率，如果各子问题是不独立的则分治法要做许多不必要的工作，重复地解公共的子问题，此时虽然可用分治法，但一般用动态规划法较好。

能采用**动态规划**求解的问题的一般要具有3个性质：

    (1) 最优化原理：如果问题的最优解所包含的子问题的解也是最优的，就称该问题具有最优子结构，即满足最优化原理。

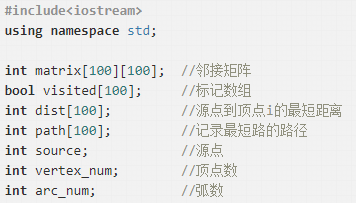
    (2) 无后效性：即某阶段状态一旦确定，就不受这个状态以后决策的影响。也就是说，某状态以后的过程不会影响以前的状态，只与当前状态有关。

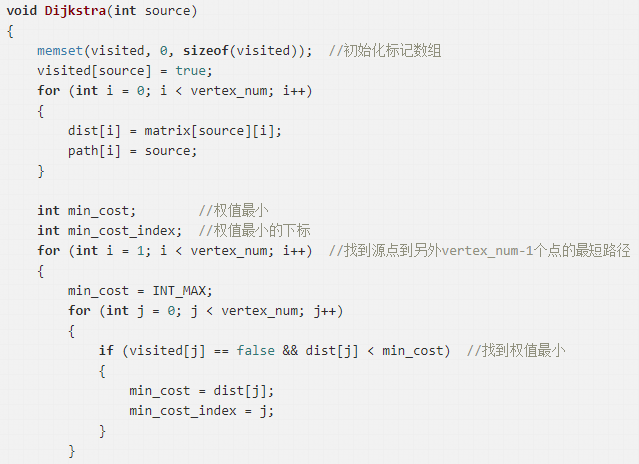
   （3）有重叠子问题：即子问题之间是不独立的，一个子问题在下一阶段决策中可能被多次使用到。（该性质并不是动态规划适用的必要条件，但是如果没有这条性质，动态规划算法同其他算法相比就不具备优势）

与分治法最大的差别是：适合于用动态规划法求解的问题，经分解后得到的子问题往往不是互相独立的（即下一个子阶段的求解是建立在上一个子阶段的解的基础上，进行进一步的求解）。

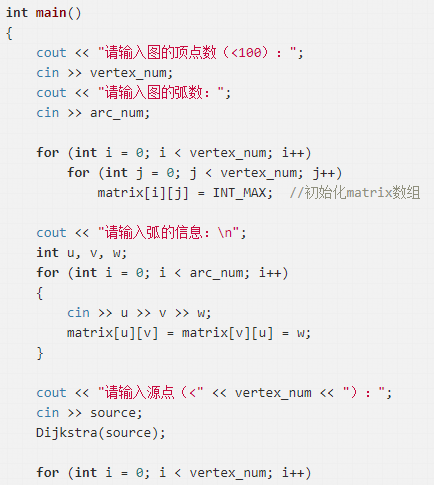
# 单源最短路径

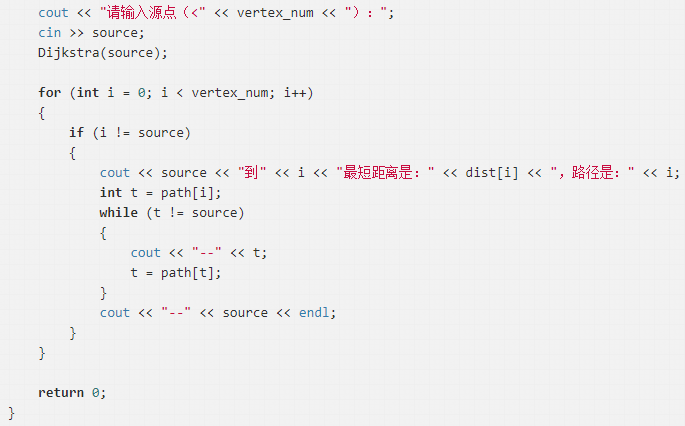
<https://segmentfault.com/a/1190000009475858>

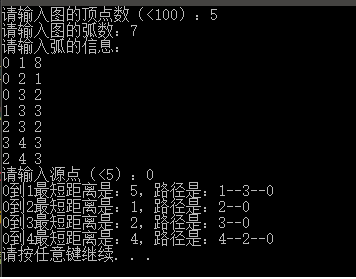












# 找零钱问题：

假设只有 1 分、 2 分、五分、 1 角、二角、 五角、 1元的硬币。在超市结账时，

如果需要找零钱， 收银员希望将最少的硬币数找给顾客。那么，给定需要找的零钱数目，

如何求得最少的硬币数呢  
static int[] arr = {1, 2, 5, 10, 20, 50, 100};

int f(int N) {

int m = N;

int c = 0;

int a = 0;

int count = 0;

for (int i = arr.length - 1; i >= 0; i--) {

a = m / arr[i];

c = m % (arr[i]);

m = c;

count = count + a;

}

return count;

}

int main() {

Cout >> f(188);

Return 0;

}

# 停车问题

小明中午路过一家公共停车场，出于程序员的职业习惯，他很想知道这个停车场上午的最大化利用率有多少。

经与门卫大叔沟通，他获得了该停车场上午车辆入场时间与出场时间的记录表（数据格式参考样例输入），

你能通过拿到的数据写一个函数快速的帮小明算出这家停车场，上午最多的时候同时停放了多少辆车吗？

要求时间复杂度不高于：O(n)\*lgN

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | 注意事项：  a、为方便起见，简化计算，驶入时间和开出时间以整点记录，如9点，10点。  b、如停车记录中入场时间晚于出场时间，该停车记录视为无效，如7,3  c、假定如果有多辆车同时出入场，出场车辆优先。  样例输入：    8,9;4,6;3,7;6,8 （车辆以分号分隔，车辆入场和出场时间以逗号分隔）  样例输出：    2 |

维护一个24大小的数组，对于每个车，两个时间之间的值+1。最后统计数组的最大值。复杂度n

要是出去的同时又有进来的，还是算一辆车？前闭后开

# ----------------------------------------

# 如何计算出一个整数的二进制表示中，有多少个bit的值等于1

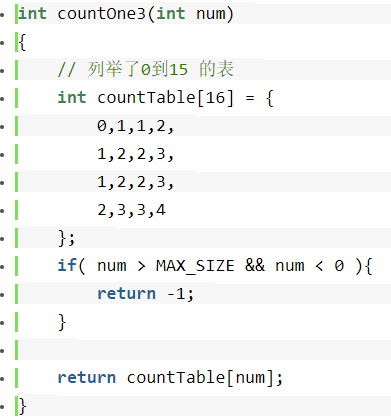
**以字节为单位算：**

一个数 再大， 也是要存在内存中的，能存在 内存中 ，那么就可以分割成一个字节一个字节的来算  
统计一个 字节有多少个0，多少个1，然后这个数占用多少个个字节，你挨个加起来就可以了

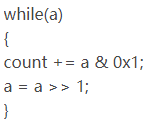
**查表法：**

将有限个数的对应的1的个数存在一个数组中，查表的时间复杂度为O(1),但是空间复杂度较大.

将0x00～0xFF中对应二进制1的个数构成一张表  
unsigned char bit1count[256]={0,1,...,8};  
对需要求二进制1的个数的字节序列中的每个字节查该表并累加。



**常规解法：**



确定一个整数的2进制中有几个bit"1"写法

注意不断右移判断最低位是1有多少个的算法是错误的，

因为根据GNU C/C++的实现来看，负数是带符号填充进行右移的，所以如果是负数，该算法会死循环。

C++可以强制转换成无符号整数再处理

下面的程序是正解。其中num &= (num - 1) 会消除num的最小位置的"1"



# Top K问题

## STL multiset 求1000个数里面最大的k个数

## 代码 求出现次数最多的k个数

题目

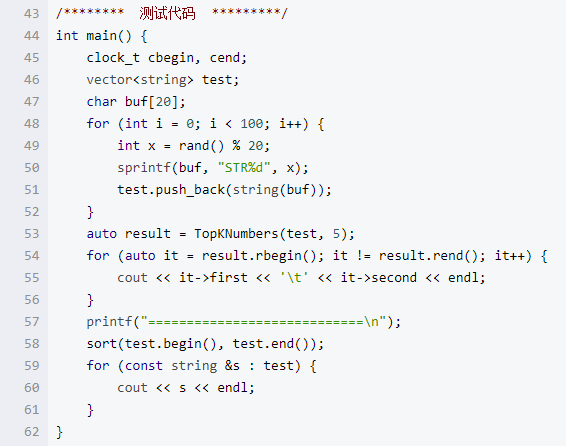
搜索引擎会通过日志文件把用户每次检索使用的所有检索串都记录下来，每个查询串的长度为1-255字节。   
假设目前有一千万个记录（这些查询串的重复度比较高，虽然总数是1千万，但如果除去重复后，不超过3百万个。一个查询串的重复度越高，说明查询它的用户越多，也就是越热门。），请你统计最热门的10个查询串，要求使用的内存不能超过1G。

思路

第一步：用Hashmap（STL中叫unordered\_map）统计词频

第二步：用容量为K的最小堆取出出现次数最大的K个词   
（参考 <http://blog.csdn.net/fuyufjh/article/details/48369801>）





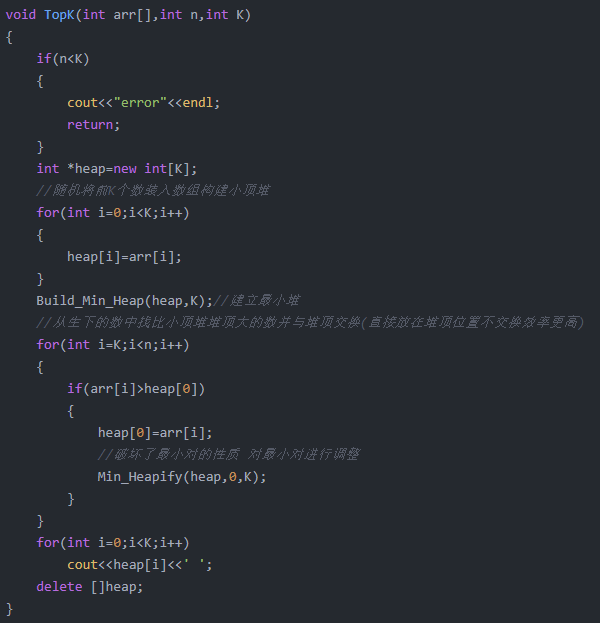
## 求一个数组前K小的数

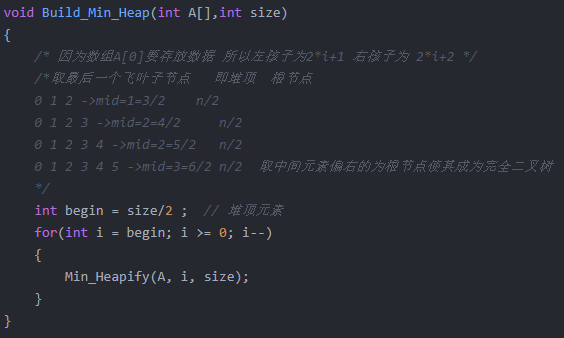
用小根堆得办法寻找最大的K个数

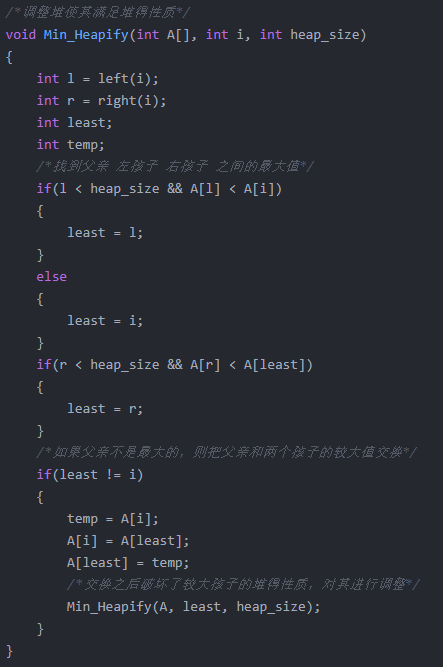
用容量为K的最小堆来存储最大的K个数。最小堆的堆顶元素就是最大K个数中的最小的一个。

每次扫描一个数据X，如果X比堆顶元素Y小，则不需要改变原来的堆。如果X比堆顶元素大，那么用X替换堆顶元素Y，在替换之后，X可能破坏了最小堆的结构，需要调整堆来维持堆的性质。

调整过程时间复杂度为O(logK)。 全部的时间复杂度为O(N\*logK)。这种方法当数据量比较大的时候，比较方便。因为对所有的数据只会遍历一次。







## 搜索词汇的topK

找每个文件的topK个URL，之后再进行排序，这样就比找出全部的URL在排序方法优秀。还有一个topK个URL到最后还是需要排序，所以我们在找每个文件的topK时，是否只需要找到topK个，其中顺序不用管，那么我们就可以用大小为K的小根堆遍历哈希表。这样又可以降低查找的时间。

这里我来讲一下为什么用小根堆。   
小根堆是一棵完全二叉树存在如下特性   
(1)若树根结点存在左孩子，则根结点的值(或某个域的值)小于等于左孩子结点的值(或某个域的值)；   
(2)若树根结点存在右孩子，则根结点的值(或某个域的值)小于等于右孩子结点的值(或某个域的值)；   
(3)以左、右孩子为根的子树又各是一个堆。   
建最小堆的过程,从最后一个叶节点的父节点开始,往前逐个检查各个节点,看其是不是符合父节点小于它的子节点,如果不小于,则将它的 子节点中最小的那个节点与父节点对换;否则,不交换,

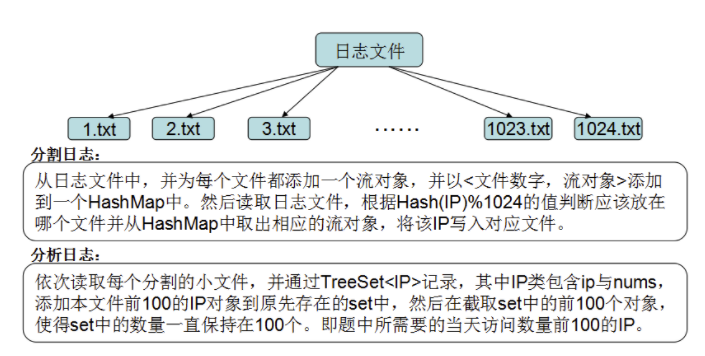
## 统计文章中出现次数最高的10个单词

算法的思路是：

1. 从头到尾遍历文件，从文件中读取遍历到的每一个单词。
2. 把遍历到的单词放到hash\_map中，并统计这个单词出现的次数。
3. 遍历hash\_map，将遍历到的单词的出现次数放到优先级队列中。
4. 当优先级队列的元素个数超过k个时就把元素级别最低的那个元素从队列中取出，这样始终保持队列的元素是k个。
5. 遍历完hash\_map，则队列中就剩下了出现次数最多的那k个元素。
6. //出现次数最多的是个单词
7. **void** top\_k\_words()
8. {
9. timer t;
10. ifstream fin;
11. fin.open("modern c.txt");
12. **if** (!fin)
13. {
14. cout<<"can nont open file"<<endl;
15. }
16. string s;
17. hash\_map<string,**int**> countwords;
18. **while** (**true**)
19. {
20. fin>>s;
21. **if** (fin.eof())
22. {
23. **break**;
24. }
25. countwords[s]++;
26. }
27. cout<<"单词总数 （重复的不计数）:"<<countwords.size()<<endl;
28. priority\_queue<pair<**int**,string>,vector<pair<**int**,string>>,greater<pair<**int**,string>>> countmax;
29. **for**(hash\_map<string,**int**>::const\_iterator i=countwords.begin();
30. i!=countwords.end();i++)
31. {
32. countmax.push(make\_pair(i->second,i->first));
33. **if** (countmax.size()>10)
34. {
35. countmax.pop();
36. }
37. }
38. **while**(!countmax.empty())
39. {
40. cout<<countmax.top().second<<" "<<countmax.top().first<<endl;
41. countmax.pop();
42. }
43. cout<<"time elapsed "<<t.elapsed()<<endl;
44. }

## [海量日志中提取访问次数前100的IP](http://c610367182.iteye.com/blog/1965382)

由于IP是32位的，最多有个2^32个IP，也就是4G,所以不能完全加载到内存中处理；可以采用映射的方法，按照IP地址的Hash(IP)%1024值,把整个大文件映射为1024个小文件，再依次分析每个小文中出现频率最大的前100个IP及相应的频率，因为涉及到排序，所以使用到了TreeSet。



# 大数据量排序

如果有一个100G的大文件（假定文件里头都是一行行的字符串记录、每个记录约100个字符），要求对这个文件进行排序，请给出你的排序算法。

 我们可以先遍历所有行，根据每行的前3个字符作为键放到一个hash结构中，这样问题已经可以用O(N^2\*lgn)解决了。被否决，有可能前三个字符都是一样的，需要对整个字符串进行排序

 或者我们可以先搞一个Trie树，然后遍历一遍全部存好，每一最后字符再多存一个数字表示这一字符串出现次数，然后对这个Trie树进行层次遍历，依次append，返回。回复说文件有100G大小，假设你手头电脑只有8g内存。嗯嗯，我可能会采用先把所有的文件分开很多份分别处理，关键在于最后怎么合并起来还是有序的

 其实就是外部排序，先将这个大文件分成很多份，把这些子文件依次读入内存，并利用自定义的有效的内部排序方法（也就是我的方法吧）对它们进行排序，再将排序后得到的有序子文件重新写入外存；可以使用多路归并排序。

# 一亿个随机数的中位数怎么找

《编程之美》里给出一个方案：设k<K,且k个数可以完全读进内存，那么先构建k个数的堆，先找出第0到k大的数，再扫描一遍数组找出第k+1到2k的数，再扫描直到找出第K个数。虽然每次时间大约是nlog(k)，但需要扫描ceil(K/k) 次，这里要扫描5次

解法：首先假设是32位无符号整数。

1. 读一遍10G个整数，把整数映射到256M个区段中，用一个64位无符号整数给每个相应区段记数。

说明：整数范围是0 - 2^32 - 1，一共有4G种取值，映射到256M个区段，则每个区段有16（4G/256M = 16）种值，每16个值算一段， 0～15是第1段，16～31是第2段，……2^32-16 ～2^32-1是第256M段。一个64位无符号整数最大值是0～8G-1，这里先不考虑溢出的情况。总共占用内存256M×8B=2GB。

2. 从前到后对每一段的计数累加，当累加的和超过5G时停止，找出这个区段（即累加停止时达到的区段，也是中位数所在的区段）的数值范围，设为[a，a+15]，同时记录累加到前一个区段的总数，设为m。然后，释放除这个区段占用的内存。

3. 再读一遍10G个整数，把在[a，a+15]内的每个值计数，即有16个计数。

4. 对新的计数依次累加，每次的和设为n，当m+n的值超过5G时停止，此时的这个计数所对应的数就是中位数。

总结：

1.以上方法只要读两遍整数，对每个整数也只是常数时间的操作，总体来说是线性时间。

2. 考虑其他情况。

若是有符号的整数，只需改变映射即可。若是64为整数，则增加每个区段的范围，那么在第二次读数时，要考虑更多的计数。若过某个计数溢出，那么可认定所在的区段或代表整数为所求，这里只需做好相应的处理。噢，忘了还要找第5G+1大的数了，相信有了以上的成果，找到这个数也不难了吧。

3. 时空权衡。

花费256个区段也许只是恰好配合2GB的内存（其实也不是，呵呵）。可以增大区段范围，减少区段数目，节省一些内存，虽然增加第二部分的对单个数值的计数，但第一部分对每个区段的计数加快了（总体改变？？待测）。

4. 映射时尽量用位操作，由于每个区段的起点都是2的整数幂，映射起来也很方便。

# 输出按登录次数排序的IP

一个文件里写了一亿条记录，记录的形式是“IP地址+登录时间”，要求，内存限制一百兆。  
  
我说可以按照取模或者hash把IP投射到若干个小文件里，  
然后在小文件中直接排序，再把小文件归并排序。。。

# 找出A,B文件共同的URL

**给你A,B两个文件，各存放50亿条URL，每条URL占用64字节，内存限制是4G，让你**  
  
50亿\*64B约320g，内存限制4g，所以，用一个hash函数把A文件中的url分到1000个小文件中去，把url通过hash后的值当做文件名，然后B文件也用同样的hash函数去分到1000个小文件中，这样寻找的时候，按照同名的文件从AB两个文件中去找重复（相同的url肯定分在相同文件名的小文件找那个），这样的话内存肯定能满足。

# 将单词相同的划分为一类，1000W个单词

# 找到100亿个URL中的重复URL

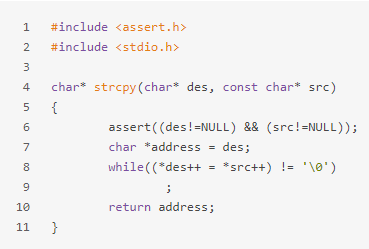
将文件通过哈希函数成多个小的文件，由于哈希函数所有重复的URL只可能在同一个文件中，在每个文件中利用一个哈希表做次数统计。就能找到重复的URL。这时候要注意的就是给了多少内存，我们要根据文件大小结合内存大小决定要分割多少文件

# 已知某个文件内包含一些电话号码，每个号码为8位数字，统计不同号码的个数。

8位最多99 999 999，大概需要99m个bit，大概10几m字节的内存即可。

# Strcpy的实现

函数strcpy的原型是char\* strcpy(char\* des , const char\* src)，des 和 src 所指内存区域不可以重叠且 des 必须有足够的空间来容纳 src 的字符串。



要知道 strcpy 会拷贝’\0’，还有要注意：

1、源指针所指的字符串内容是不能修改的，因此应该声明为 const 类型。

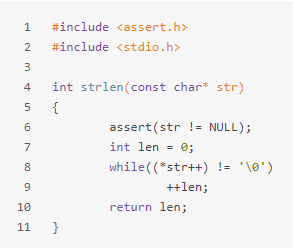
2、要判断源指针和目的指针为空的情况，思维要严谨，这里使用assert。

3、要用一个临时变量保存目的串的首地址，最后返回这个首地址。

4、函数返回 char\* 的目的是为了支持链式表达式，即strcpy可以作为其他函数的实参。

# strlen的实现

函数strlen的原型是size\_t strlen(const char \*s)，其中 size\_t 就是 unsigned int。



strlen 与 sizeof 的区别：

sizeof是运算符，strlen是库函数。

sizeof可以用类型、变量做参数，而strlen只能用 char\* 变量做参数，且必须以\0结尾。

sizeof是在编译的时候计算类型或变量所占内存的大小，而strlen的结果要在运行的时候才能计算出来，用来计算字符串的长度。

数组做sizeof的参数不退化，传递给strlen就退化为指针了。

# 字符串连接strcat的实现

函数strcat的原型是char\* strcat(char\* des, char\* src)，des 和 src 所指内存区域不可以重叠且 des 必须有足够的空间来容纳 src 的字符串。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 | #include <assert.h> #include <stdio.h>  char\* strcat(char\* des, const char\* src) // const表明为输入参数  {   assert((des!=NULL) && (src!=NULL));  char\* address = des;  while(\*des != '\0') // 移动到字符串末尾  ++des;  while(\*des++ = \*src++)  ;  return address; } |

# 字符串比较strcmp的实现

函数strcmp的原型是int strcmp(const char \*s1,const char \*s2)。

若s1==s2，返回零；

若s1>s2，返回正数；

若s1<s2，返回负数。

即：两个字符串自左向右逐个字符相比（按ASCII值大小相比较），直到出现不同的字符或遇\0为止。

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | #include <assert.h> #include <stdio.h>  int strcmp(const char \*s1,const char \*s2) {  assert((s1!=NULL) && (s2!=NULL));  while(\*s1 == \*s2)  {  if(\*s1 == '\0')  return 0;    ++s1;  ++s2;  }  return \*s1 - \*s2; } |

# ----------------------------------------

# 7个等重量的球还有一个稍重的球共8个怎么用天平2次找出最重的一个？

去6个均分成两组称一次，如果相等则再称剩下的两个。如果不相等取重的一组，取两个称。

# 每张牌经过洗牌后不在现在的位置上

1.可以用交换位置的方式，每次取完之后就把取牌位置的牌换成最后一张牌，然后总牌数减1，下一次就在剩下的牌中取，因为每次取票的位置随机，所以即使交换了位置取到的牌还是随机的。所以总结一下就是每次在1到(n-i)中取一张牌，然后与(n-i)交换位置，i=0~(n-2).

**即：牌固定，位置随机：1到(n-i)中取一张牌，然后与(n-i)交换位置**

2.局部整理法。假如有n张牌已经随机排列，此时新增一张牌，要如何使得新的牌随机排列？只需要将新增的牌随机放入一个位置即可，我们可以生成一个随机数，将随机数所在位置的数与最后一个位置（新牌）交换。利用这种思想，我们可以先整理局部的牌，然后慢慢地加牌进来。而这个“局部”，递归到最后就是一张牌，一张牌本身就是“随机”的排列，然后我们加入另一张牌，与其随机交换...最终的实现方式就是：

在1~i之间产生一个随机位置，与i位置的牌交换，i=2~n.

**即：位置固定，牌随机放。1~i i+1~n，假设i+1~n已经排好了，然后从1~i随机产生一个位置t，arr[t]和arr[i]交换；递归。最小的i是2。**

# [随机抽5张牌判断是不是顺子，即这5张牌是不是连续的。](https://blog.csdn.net/Jarvan_Song/article/details/52416039)

https://blog.csdn.net/Jarvan\_Song/article/details/52416039

法一：把数组排序，统计数组中0的个数，统计排序之后的数组相邻数字之间的空缺总数。如果空缺的总数小于或者等于0的个数，那么这个数组就是连续的；反之则不连续。

法二：

1）确认5张牌中除了0，其余数字没有重复的（可以用表统计的方法）;  
2) 满足这样的逻辑：（max，min分别代表5张牌中的除0以外的最大值最小值）  
       如果没有0，则max-min=4，则为顺子，否则不是  
       如果有一个0，则max-min=4或者3，则为顺子，否则不是  
       如果有两个0，则max-min=4或者3或者2，则为顺子，否则不是  
最大值和最小值在1）中就可以获得，这样就不用排序了。

法三：建立一个数组rank[13],并根据5张牌的点数花色为其赋值，使rank[i]为j代表点数为i的牌有j张，例如rank[3] = 2代表点数为3的牌有2张。  
示例代码：  
    int rank[13] = {0};  
    for (i = 0; i < 5; i ++){  
        rank[hand[i][0]] ++;    //hand[5][2]存储的是5张牌的点数和花色  
然后根据rank[13]来判断是否顺子。  
示例代码：  
    int i;  
    int num\_consec = 0;  
    bool straight = false;  
    for (i = 0; rank[i] == 0; i ++);  
        while (i < 13 && rank[i] > 0){  
            num\_consec ++;  
            i ++;  
        }  
    if (num\_consec == 5){  
        straight = true;  
        return;  
    }

# 统计二进制字符里面1的个数

Bitset bit.Count()

# 递归/回溯/DFS

# 找出一个数组n中和为m的所有可能组合

# 链表

# MapReduce

先把要处理的数据分成n多个小块，然后交由n多个处理器处理，最后再通过一定的手段进行数据汇总，得出答案。

假设我们有一组数据：1,2,3,…,100。求这一组数据的平方和。现在我们用MapReduce这个模型解决这一个问题。首先我们把这组数据分成100份，交由100台处理器去处理。每一台处理器只做一件事，就是把自己要处理的数据平方一下。这样一来，最初的那组数[1,2,3,…,100]就被映射成了[1,4,9,16,…,10000]了。这就是所谓的Map操作。而Reduce操作呢？Reduce操作就是把映射后得到的这100个新的数据累

MapReduce编程模型的原理是：利用一个输入key/value pair集合来产生一个输出的key/value pair集合。MapReduce库的用户用两个函数表达这个计算：Map和Reduce。

    用户自定义的Map函数接受一个输入的key/value pair值，然后产生一个中间key/value pair值的集合。MapReduce库把所有具有相同中间key值I的中间value值集合在一起后传递给reduce函数。

    用户自定义的Reduce函数接受一个中间key的值I和相关的一个value值的集合。Reduce函数合并这些value值，形成一个较小的value值的集合。一般的，每次Reduce函数调用只产生0或1个输出value值。通常我们通过一个迭代器把中间value值提供给Reduce函数，这样我们就可以处理无法全部放入内存中的大量的value值的集合。

MapReduce架构的程序能够在大量的普通配置的计算机上实现并行化处理。这个系统在运行时只关心：如何分割输入数据，在大量计算机组成的集群上的调度，集群中计算机的错误处理，管理集群中计算机之间必要的通信。采用MapReduce架构可以使那些没有并行计算和分布式处理系统开发经验的程序员有效利用分布式系统的丰富资源。

例如，计算一个大的文档集合中每个单词出现的次数，下面是伪代码段：  
map(String key, String value):  
    // key: document name  
    // value: document contents  
    for each word w in value:  
        EmitIntermediate(w, “1″);  
reduce(String key, Iterator values):  
    // key: a word  
    // values: a list of counts  
    int result = 0;  
    for each v in values:  
        result += ParseInt(v);  
    Emit(AsString(result));

    Map函数输出文档中的每个词、以及这个词的出现次数(在这个简单的例子里就是1)。Reduce函数把Map函数产生的每一个特定的词的计数累加起来。