1. 链表与数组。

2. 队列和栈，出栈与入栈。

3. 链表的删除、插入、反向。

（1）反向

链表的定义：

class Node{

public int val;

public Node next;

public Node(int val){this.val=val;}

}

**反向**分为4步：

两个临时变量：当前节点curr和前一个节点prev

**第一步**：保存当前节点的下一个节点到tmp中；

**第二步**；将当前节点的下一个指向前一个；

**第三步**；将前一个prev指向指向当前节点curr；

**第四步**：将当前节点curr指向tmp，也就是实际上的下一个节点

public static Node reversal(Node head){

Node prev=null;

Node curr=head;

while(curr!=null){

Node tmp=curr.next;

curr.next=prev;

prev=curr;

curr=temp;

}

return prev;

}

4. 字符串操作。

5. Hash表的hash函数，冲突解决方法有哪些。

答案：**Hash**

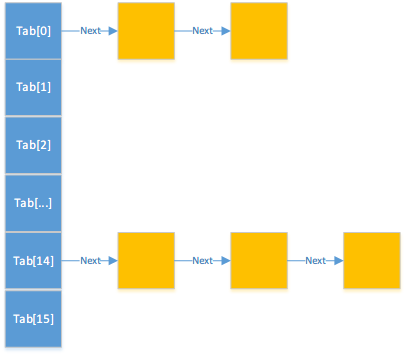
哈希表（Hash Table，也叫散列表），是根据关键码值 (Key-Value) 而直接进行访问的数据结构。也就是说，**它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度**。哈希表的实现主要需要解决两个问题，哈希函数和冲突解决。

**哈希函数**

哈希函数也叫散列函数，它对不同的输出值得到一个固定长度的消息摘要。理想的哈希函数对于不同的输入应该产生不同的结构，**同时散列结果应当具有同一性（输出值尽量均匀）和雪崩效应（微小的输入值变化使得输出值发生巨大的变化）**。

**冲突解决**

* 开放地址法：**以发生冲突的哈希地址为输入，通过某种哈希冲突函数得到一个新的空闲的哈希地址的方法**。有以下几种方式：
  + 线性探查法：从发生冲突的地址开始，依次探查下一个地址，直到找到一个空闲单元。
  + 平方探查法：设冲突地址为d0，则探查序列为：d0+1^2,d0-1^2,d0+2^2...
* 拉链法：把所有的同义词用单链表链接起来。在这种方法下，哈希表每个单元中存放的不再是元素本身，而是相应同义词单链表的头指针。HashMap就是使用这种方法解决冲突的。

[](https://github.com/FrankLee96/interview/blob/master/basic/algo/hashmap-structure.png)

6. 各种排序：冒泡、选择、插入、希尔、归并、快排、堆排、桶排、基数的原理、平均时间复杂度、最坏时间复杂度、空间复杂度、是否稳定。

**（1）常见排序算法**

**稳定排序：**

* 冒泡排序 — O(n²)
* 插入排序 — O(n²)
* 桶排序 — O(n); 需要 O(k) 额外空间
* 归并排序 — O(nlogn); 需要 O(n) 额外空间
* 二叉排序树排序 — O(n log n) 期望时间; O(n²)最坏时间; 需要 O(n) 额外空间
* 基数排序 — O(n·k); 需要 O(n) 额外空间

**不稳定排序**

* 选择排序 — O(n²)
* 希尔排序 — O(nlogn)
* 堆排序 — O(nlogn)
* 快速排序 — O(nlogn) 期望时间, O(n²) 最坏情况; 对于大的、乱数串行一般相信是最快的已知排序

7. 快排的partition函数与归并的Merge函数。

8. 对冒泡与快排的改进。

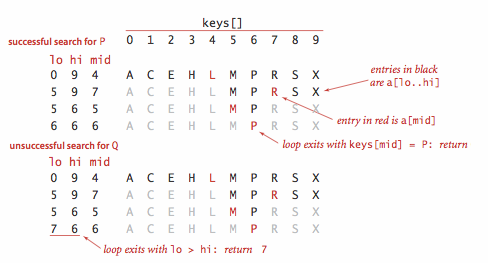
（1）对快排的改进

1. 当待排序序列的长度分割到一定大小后，使用插入排序。
2. 从左、中、右三个数中取中间值。

**9. 二分查找，与变种二分查找。**

**1 二分查找**

　　二分查找是一个基础的算法，也是面试中常考的一个知识点。二分查找就是将查找的键和子数组的中间键作比较，如果被查找的键小于中间键，就在左子数组继续查找；如果大于中间键，就在右子数组中查找，否则中间键就是要找的元素。



（图片来自《算法-第4版》）

[复制代码](javascript:void(0);)

/\*\*

\* 二分查找，找到该值在数组中的下标，否则为-1

\*/

static int binarySerach(int[] array, int key) {

int left = 0;

int right = array.length - 1;

// 这里必须是 <=

while (left <= right) {

int mid = (left + right) / 2;

if (array[mid] == key) {

return mid;

}

else if (array[mid] < key) {

left = mid + 1;

}

else {

right = mid - 1;

}

}

return -1;

}

[复制代码](javascript:void(0);)

　　每次移动left和right指针的时候，需要在mid的基础上+1或者-1， 防止出现死循环， 程序也就能够正确的运行。

　　注意：代码中的判断条件必须是while (left <= right)，否则的话判断条件不完整，比如：array[3] = {1, 3, 5};待查找的键为5，此时在(low < high)条件下就会找不到，因为low和high相等时，指向元素5，但是此时条件不成立，没有进入while()中。

**2 二分查找的变种**

　　关于二分查找，如果条件稍微变换一下，比如：数组之中的数据可能可以重复，要求返回匹配的数据的最小（或最大）的下标；更近一步， 需要找出数组中第一个大于key的元素（也就是最小的大于key的元素的）下标，等等。 这些，虽然只有一点点的变化，实现的时候确实要更加的细心。

　　二分查找的变种和二分查找原理一样，主要就是变换判断条件（也就是边界条件），如果想直接看如何记忆这些变种的窍门，请直接翻到本文最后。下面来看几种二分查找变种的代码：

**2.1 查找第一个与key相等的元素**

　　查找第一个相等的元素，也就是说等于查找key值的元素有好多个，返回这些元素最左边的元素下标。

[复制代码](javascript:void(0);)

// 查找第一个相等的元素

static int findFirstEqual(int[] array, int key) {

int left = 0;

int right = array.length - 1;

// 这里必须是 <=

while (left <= right) {

int mid = (left + right) / 2;

if (array[mid] >= key) {

right = mid - 1;

}

else {

left = mid + 1;

}

}

if (left < array.length && array[left] == key) {

return left;

}

return -1;

}

[复制代码](javascript:void(0);)

**2.2 查找最后一个与key相等的元素**

　　查找最后一个相等的元素，也就是说等于查找key值的元素有好多个，返回这些元素最右边的元素下标。

[复制代码](javascript:void(0);)

// 查找最后一个相等的元素

static int findLastEqual(int[] array, int key) {

int left = 0;

int right = array.length - 1;

// 这里必须是 <=

while (left <= right) {

int mid = (left + right) / 2;

if (array[mid] <= key) {

left = mid + 1;

}

else {

right = mid - 1;

}

}

if (right >= 0 && array[right] == key) {

return right;

}

return -1;

}

[复制代码](javascript:void(0);)

**2.3 查找最后一个等于或者小于key的元素**

　　查找最后一个等于或者小于key的元素，也就是说等于查找key值的元素有好多个，返回这些元素最右边的元素下标；如果没有等于key值的元素，则返回小于key的最右边元素下标。

[复制代码](javascript:void(0);)

// 查找最后一个等于或者小于key的元素

static int findLastEqualSmaller(int[] array, int key) {

int left = 0;

int right = array.length - 1;

// 这里必须是 <=

while (left <= right) {

int mid = (left + right) / 2;

if (array[mid] > key) {

right = mid - 1;

}

else {

left = mid + 1;

}

}

return right;

}

[复制代码](javascript:void(0);)

**2.4 查找最后一个小于key的元素**

　　查找最后一个小于key的元素，也就是说返回小于key的最右边元素下标。

[复制代码](javascript:void(0);)

// 查找最后一个小于key的元素

static int findLastSmaller(int[] array, int key) {

int left = 0;

int right = array.length - 1;

// 这里必须是 <=

while (left <= right) {

int mid = (left + right) / 2;

if (array[mid] >= key) {

right = mid - 1;

}

else {

left = mid + 1;

}

}

return right;

}

[复制代码](javascript:void(0);)

**2.5 查找第一个等于或者大于key的元素**

　　查找第一个等于或者大于key的元素，也就是说等于查找key值的元素有好多个，返回这些元素最左边的元素下标；如果没有等于key值的元素，则返回大于key的最左边元素下标。

[复制代码](javascript:void(0);)

// 查找第一个等于或者大于key的元素

static int findFirstEqualLarger(int[] array, int key) {

int left = 0;

int right = array.length - 1;

// 这里必须是 <=

while (left <= right) {

int mid = (left + right) / 2;

if (array[mid] >= key) {

right = mid - 1;

}

else {

left = mid + 1;

}

}

return left;

}

[复制代码](javascript:void(0);)

**2.6 查找第一个大于key的元素**

　　查找第一个等于key的元素，也就是说返回大于key的最左边元素下标。

[复制代码](javascript:void(0);)

// 查找第一个大于key的元素

static int findFirstLarger(int[] array, int key) {

int left = 0;

int right = array.length - 1;

// 这里必须是 <=

while (left <= right) {

int mid = (left + right) / 2;

if (array[mid] > key) {

right = mid - 1;

}

else {

left = mid + 1;

}

}

return left;

}

[复制代码](javascript:void(0);)

**3 二分查找变种总结**

[复制代码](javascript:void(0);)

// 这里必须是 <=

while (left <= right) {

int mid = (left + right) / 2;

if (array[mid] ? key) {

//... right = mid - 1;

}

else {

// ... left = mid + 1;

}

}

return xxx;

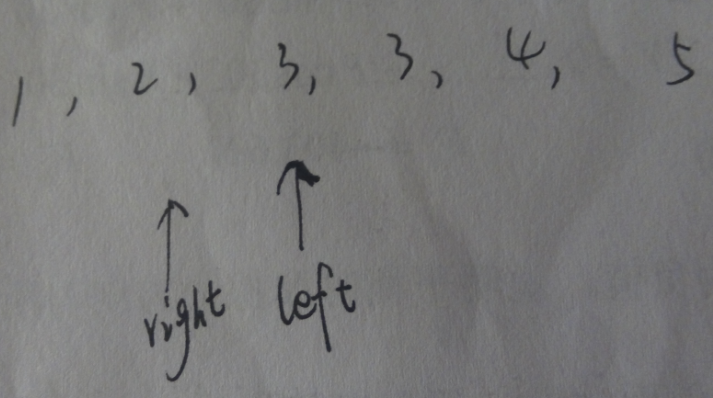
[复制代码](javascript:void(0);)

　　二分查找变种较多，不过它们的“套路”是一样的，以上代码就是其套路，如何快速写出二分查找的代码，只需按照以下步骤即可：

**1 首先判断出是返回left，还是返回right**

　　因为我们知道最后跳出while (left <= right)循环条件是right < left，且right = left - 1。最后right和left一定是卡在"边界值"的左右两边，如果是比较值为key，查找小于等于（或者是小于）key的元素，则边界值就是等于key的所有元素的最左边那个，其实应该返回left。

　　以数组{1, 2, 3, 3, 4, 5}为例，如果需要查找第一个等于或者小于3的元素下标，我们比较的key值是3，则最后left和right需要满足以下条件：



　　我们比较的key值是3，所以此时我们需要返回left。

**2 判断出比较符号**

[复制代码](javascript:void(0);)

int mid = (left + right) / 2;

if (array[mid] ? key) {

//... right = xxx;

}

else {

// ... left = xxx;

}

[复制代码](javascript:void(0);)

　　也就是这里的 if (array[mid] ? key) 中的判断符号，结合步骤1和给出的条件，如果是查找小于等于key的元素，则知道应该使用判断符号>=，因为是要返回left，所以如果array[mid]等于或者大于key，就应该使用>=，以下是完整代码

[复制代码](javascript:void(0);)

// 查找小于等于key的元素

int mid = (left + right) / 2;

if (array[mid] >= key) {

right = mid - 1;

}

else {

left = mid + 1;

}

[复制代码](javascript:void(0);)

10. 二叉树、B+树、AVL树、红黑树、哈夫曼树。

**（1）红黑树**

红黑树和AVL树的区别在于它使用颜色来标识结点的高度，它所追求的是局部平衡而不是AVL树中的非常严格的平衡。之前我们在讲解AVL树时，已经领教过AVL树的复杂，但AVL树的复杂比起红黑树来说简直是小巫见大巫。红黑树是真正的变态级数据结构。

**一、红黑树的平衡**

红黑树首先是一棵二叉查找树，它每个结点都被标上了颜色（红色或黑色），红黑树满足以下5个性质：

**1、 每个结点的颜色只能是红色或黑色。**

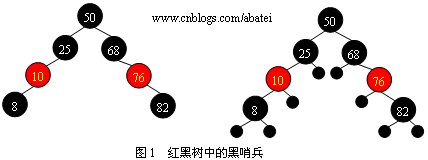
**2、 根结点是黑色的。**

**3、 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。**

**4、 如果一个结点是红的，则它的两个儿子都是黑的。也就是说在一条路径上不能出现相邻的两个红色结点。**

**5、 对于每个结点来说，从该结点到其子孙叶结点的所有路径上包含相同数目的黑结点。**

红黑树的这5个性质中，第3点是比较难理解的，但它却非常有必要。我们看图1中的左边这张图，如果不使用黑哨兵，它完全满足红黑树性质，结点50到两个叶结点8和叶结点82路径上的黑色结点数都为2个。但如果加入黑哨兵后（如图1右图中的小黑圆点），叶结点的个数变为8个黑哨兵，根结点50到这8个叶结点路径上的黑高度就不一样了，所以它并不是一棵红黑树。



要看真正的红黑树请在以上动画中添加几个结点，看看是否满足以上性质。

**二、红黑树的旋转操作**

红黑树的旋转操作和AVL树一样，分为LL、RR、LR、RL四种旋转类型，差别在于旋转完成后改变的是结点的颜色，而不是平衡因子。旋转动画演示请参考AVL这篇文章中的Flash动画：

<http://www.cnblogs.com/abatei/archive/2008/11/17/1335031.html>

**三、红黑树上结点的插入**

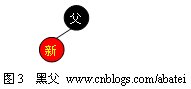
在讨论红黑树的插入操作之前必须要明白，任何一个即将插入的新结点的初始颜色都为红色。这一点很容易理解，因为插入黑点会增加某条路径上黑结点的数目，从而导致整棵树黑高度的不平衡。但如果新结点父结点为红色时（如图2所示），将会违返红黑树性质：一条路径上不能出现相邻的两个红色结点。这时就需要通过一系列操作来使红黑树保持平衡。



为了清楚地表示插入操作以下在结点中使用“新”字表示一个新插入的结点；使用“父”字表示新插入点的父结点；使用“叔”字表示“父”结点的兄弟结点；使用“祖”字表示“父”结点的父结点。插入操作分为以下几种情况：

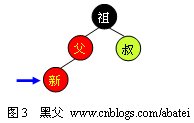
**1、黑父**

如图3所示，如果新点的父结点为黑色结点，那么插入一个红点将不会影响红黑树的平衡，此时插入操作完成。红黑树比AVL树优秀的地方之一在于黑父的情况比较常见，从而使红黑树需要旋转的几率相对AVL树来说会少一些。



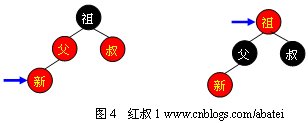
**2．红父**

如果新点的父结点为红色，这时就需要进行一系列操作以保证整棵树红黑性质。如图3所示，由于父结点为红色，此时可以判定，祖父结点必定为黑色。这时需要根据叔父结点的颜色来决定做什么样的操作。青色结点表示颜色未知。由于有可能需要根结点到新点的路径上进行多次旋转操作，而每次进行不平衡判断的起始点（我们可将其视为新点）都不一样。所以我们在此使用一个蓝色箭头指向这个起始点，并称之为判定点。



**2.1 红叔**

当叔父结点为红色时，如图4所示，无需进行旋转操作，只要将父和叔结点变为黑色，将祖父结点变为红色即可。但由于祖父结点的父结点有可能为红色，从而违反红黑树性质。此时必须将祖父结点作为新的判定点继续向上进行平衡操作。



需要注意，无论“父”在“叔”的左边还是右边，无论“新”是“父”的左孩子还是右孩子，它们的操作都完全一样。

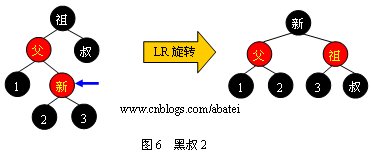
**2.2 黑叔**

当叔父结点为黑色时，需要进行旋转，以下图示了所有的旋转可能

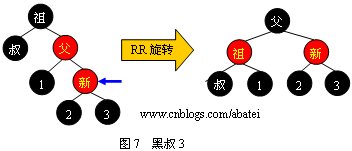
**情形1：**



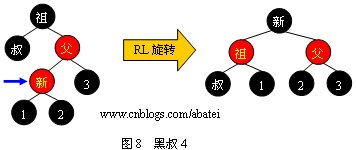
**情形2：**



**情形3：**



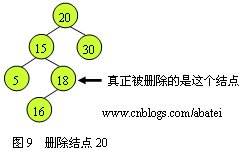
**情形4：**



可以观察到，当旋转完成后，新的旋转根全部为黑色，此时不需要再向上回溯进行平衡操作，插入操作完成。需要注意，上面四张图的“叔”、“1”、“2”、“3”结点有可能为黑哨兵结点。其实红黑树的插入操作不是很难，甚至比AVL树的插入操作还更简单些。但删除操作就远远比AVL树复杂得多，下面就介绍红黑树的删除操作。

**四、红黑树上结点的删除**

红黑树本身是一棵二叉查找树，它的删除和二叉查找树的删除类似。首先要找到真正的删除点，当被删除结点n存在左右孩子时，真正的删除点应该是n的中序遍在前驱，关于这一点请复习二叉查找树的删除。如图9所示，当删除结点20时，实际被删除的结点应该为18，结点20的数据变为18。

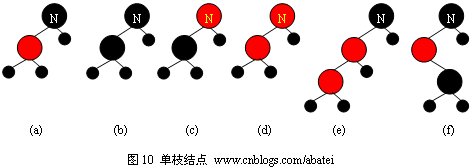


 可以可以推断出，在进行删除操作时，真正的删除点必定是只有一个孩子或没有孩子的结点。而根据红黑树的性质可以得出以下两个结论：

1、 **删除操作中真正被删除的必定是只有一个红色孩子或没有孩子的结点**。

2、 **如果真正的删除点是一个红色结点，那么它必定是一个叶子结点**。

理解这两点非常重要，如图10所示，除了情况(a)外，其他任一种况结点N都无法满足红黑树性质。



在以下讨论中，我们使用蓝色箭头表示真正的删除点，它也是旋转操作过程中的第一个判定点；真正的删除点使用“旧”标注，旧点所在位置将被它的的孩子结点所取代（最多只有一个孩子），我们使用“新”表示旧点的孩子结点。删除操作可分为以下几种情形：

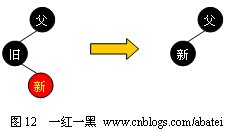
**1、旧点为红色结点**

若旧点为红色结点，则它必定是叶子结点，直接删除即可。如图11所示



**2、一红一黑**

当旧点为黑色结点，新点为红色结点时，将新点取代旧点位置后，将新点染成黑色即可（如图12所示）。这里需要注意：旧点为红色，新点为黑色的情况不可能存在。

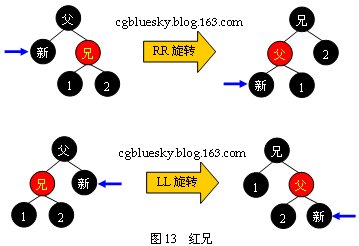


**3、双黑**

当旧点和新点都为黑色时（新点为空结点时，亦属于这种情况），情况比较复杂，需要根据旧点兄弟结点的颜色来决定进行什么样的操作。我们使用“兄”来表示旧点的兄弟结点。这里可分为红兄和黑兄两种情况：

**3.1 红兄**

由于兄弟结点为红色，所以父结点必定为黑色，而旧点被删除后，新点取代了它的位置。下图演示了两种可能的情况：



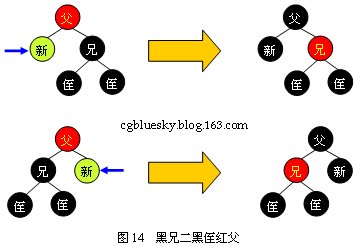
红兄的情况需要进行RR或LL型旋转，然后将父结点染成红色，兄结点染成黑色。然后重新以新点为判定点进行平衡操作。我们可以观察到，旋转操作完成后，判定点没有向上回溯，而是降低了一层，此时变成了黑兄的情况。

**3.2 黑兄**

黑兄的情况最为复杂，需要根据黑兄孩子结点（这里用“侄”表示）和父亲结点的颜色来决定做什么样的操作。

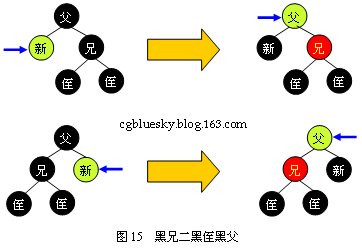
**3.2.1 黑兄二黑侄红父**

如图14所示，这种情况比较简单，只需将父结点变为黑色，兄结点变为黑色，新结点变为黑色即可，删除操作到此结束。



**3.2.2 黑兄二黑侄黑父**

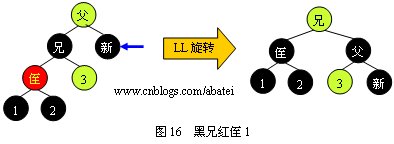
如图15所示，此时将父结点染成新结点的颜色，新结点染成黑色，兄结点染成红色即可。当新结点为红色时，父结点被染成红色，此时需要以父结点为判定点继续向上进行平衡操作。



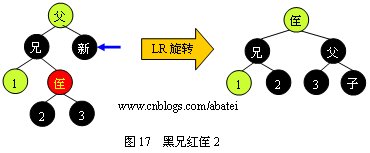
**3.2.3 黑兄红侄**

黑兄红侄有以下四种情形，下面分别进行图示：

情形1：



情形2：



情形3：



情形4：



由以上图例所示，看完以上四张图的兄弟有可能会有一个疑问，如果情形1和情形2中的两个侄子结点都为红色时，是该进行LL旋转还是进行LR旋转呢？答案是进行LL旋转。情形3和情形4则是优先进行RR旋转的判定。

11. 二叉树的前中后续遍历：递归与非递归写法，层序遍历算法。

（1）前中后续遍历的**递归**写法

见书

（2）前中后续遍历的**非递归**写法

见博客：<http://www.jianshu.com/p/49c8cfd07410>

（3）层序遍历算法

**public** **static** **void** levelOrder(TreeNode Node, **int** level) {

**if** (Node == **null** || level < 1) {

**return**;

}

**if** (level == 1) {

System.***out***.print(Node.val + " ");

**return**;

}

// 左子树

*levelOrder*(Node.left, level - 1);

// 右子树

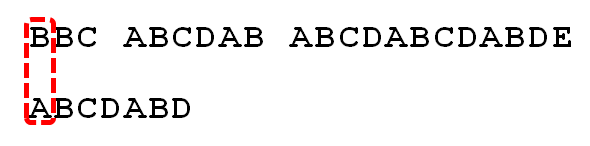
*levelOrder*(Node.right, level - 1);

}

12. 图的BFS与DFS算法，最小生成树prim算法与最短路径Dijkstra算法。

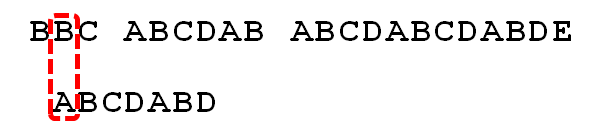
**13. KMP算法——字符串匹配算法**

1.



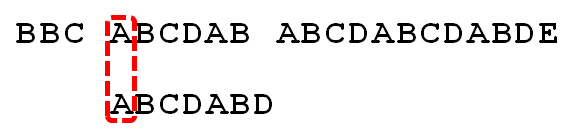
　　首先，字符串"BBC ABCDAB ABCDABCDABDE"的第一个字符与搜索词"ABCDABD"的第一个字符，进行比较。因为B与A不匹配，所以搜索词后移一位。

　　2.



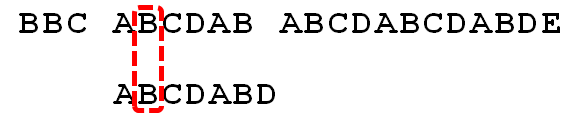
　　因为B与A不匹配，搜索词再往后移。

　　3.



　　就这样，直到字符串有一个字符，与搜索词的第一个字符相同为止。

　　4.



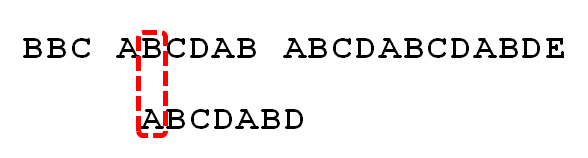
　　接着比较字符串和搜索词的下一个字符，还是相同。

　　5.



　　直到字符串有一个字符，与搜索词对应的字符不相同为止。

　　6.



　　这时，最自然的反应是，将搜索词整个后移一位，再从头逐个比较。这样做虽然可行，但是效率很差，因为你要把"搜索位置"移到已经比较过的位置，重比一遍。

　　7.



　　一个基本事实是，当空格与D不匹配时，你其实知道前面六个字符是"ABCDAB"。KMP算法的想法是，设法利用这个已知信息，不要把"搜索位置"移回已经比较过的位置，继续把它向后移，这样就提高了效率。

　　8.



　　怎么做到这一点呢？可以针对搜索词，算出一张《部分匹配表》（Partial Match Table）。这张表是如何产生的，后面再介绍，这里只要会用就可以了。

　　9.

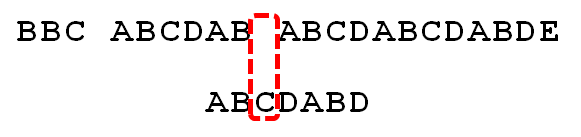


　　已知空格与D不匹配时，前面六个字符"ABCDAB"是匹配的。查表可知，最后一个匹配字符B对应的"部分匹配值"为2，因此按照下面的公式算出向后移动的位数：

　　移动位数 = 已匹配的字符数 - 对应的部分匹配值

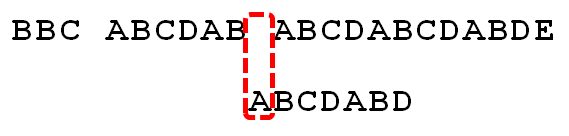
　　因为 6 - 2 等于4，所以将搜索词向后移动4位。

　　10.



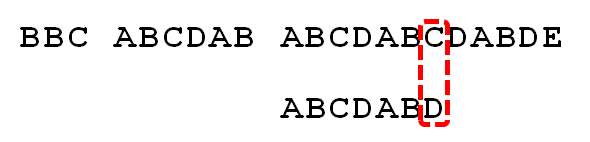
　　因为空格与Ｃ不匹配，搜索词还要继续往后移。这时，已匹配的字符数为2（"AB"），对应的"部分匹配值"为0。所以，移动位数 = 2 - 0，结果为 2，于是将搜索词向后移2位。

　　11.



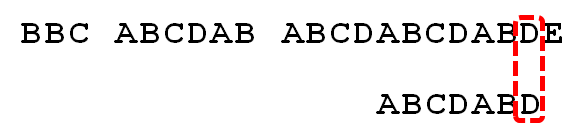
　　因为空格与A不匹配，继续后移一位。

　　12.



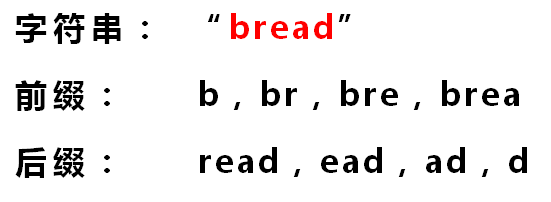
　　逐位比较，直到发现C与D不匹配。于是，移动位数 = 6 - 2，继续将搜索词向后移动4位。

　　13.



　　逐位比较，直到搜索词的最后一位，发现完全匹配，于是搜索完成。如果还要继续搜索（即找出全部匹配），移动位数 = 7 - 0，再将搜索词向后移动7位，这里就不再重复了。

　　14.



　　下面介绍《部分匹配表》是如何产生的。

　　首先，要了解两个概念："前缀"和"后缀"。 "前缀"指除了最后一个字符以外，一个字符串的全部头部组合；"后缀"指除了第一个字符以外，一个字符串的全部尾部组合。

　　15.



　　"部分匹配值"就是"前缀"和"后缀"的最长的共有元素的长度。以"ABCDABD"为例，

　　－　"A"的前缀和后缀都为空集，共有元素的长度为0；

　　－　"AB"的前缀为[A]，后缀为[B]，共有元素的长度为0；

　　－　"ABC"的前缀为[A, AB]，后缀为[BC, C]，共有元素的长度0；

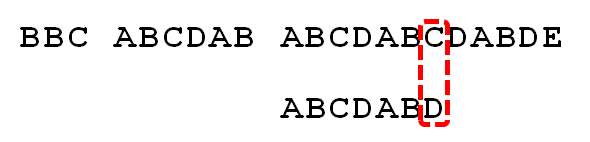
　　－　"ABCD"的前缀为[A, AB, ABC]，后缀为[BCD, CD, D]，共有元素的长度为0；

　　－　"ABCDA"的前缀为[A, AB, ABC, ABCD]，后缀为[BCDA, CDA, DA, A]，共有元素为"A"，长度为1；

　　－　"ABCDAB"的前缀为[A, AB, ABC, ABCD, ABCDA]，后缀为[BCDAB, CDAB, DAB, AB, B]，共有元素为"AB"，长度为2；

　　－　"ABCDABD"的前缀为[A, AB, ABC, ABCD, ABCDA, ABCDAB]，后缀为[BCDABD, CDABD, DABD, ABD, BD, D]，共有元素的长度为0。

　　16.



　　"部分匹配"的实质是，有时候，字符串头部和尾部会有重复。比如，"ABCDAB"之中有两个"AB"，那么它的"部分匹配值"就是2（"AB"的长度）。搜索词移动的时候，第一个"AB"向后移动4位（字符串长度-部分匹配值），就可以来到第二个"AB"的位置。

14. 排列组合问题。

15. 动态规划、贪心算法、分治算法。（一般不会问到）

**16. 大数据处理：类似10亿条数据找出最大的1000个数.........等等**

答案：

大数据处理问题汇总：

**1、海量日志数据，提取出某日访问百度次数最多的那个IP。**

      首先是这一天，并且是访问百度的日志中的IP取出来，逐个写入到一个大文件中。注意到IP是32位的，最多有个2^32个IP。同样可以采用映射的方法，比如模1000，把整个大文件映射为1000个小文件，再找出每个小文中出现频率最大的IP（可以采用hash\_map进行频率统计，然后再找出频率最大的几个）及相应的频率。然后再在这1000个最大的IP中，找出那个频率最大的IP，即为所求。

或者如下阐述（雪域之鹰）：  
**算法思想：分而治之+Hash**

1.IP地址最多有2^32=4G种取值情况，所以不能完全加载到内存中处理；   
2.可以考虑采用“分而治之”的思想，按照IP地址的Hash(IP)%1024值，把海量IP日志分别存储到1024个小文件中。这样，每个小文件最多包含4MB个IP地址；   
3.对于每一个小文件，可以构建一个IP为key，出现次数为value的Hash map，同时记录当前出现次数最多的那个IP地址；  
4.可以得到1024个小文件中的出现次数最多的IP，再依据常规的排序算法得到总体上出现次数最多的IP；

**2、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。

    典型的Top K算法，还是在这篇文章里头有所阐述，详情请参见：[十一、从头到尾彻底解析Hash表](http://blog.csdn.net/v_JULY_v/archive/2011/03/17/6256463.aspx)算法。  
      
    文中，给出的最终算法是：  
    第一步、先对这批海量数据预处理，在O（N）的时间内用Hash表完成**统计**（之前写成了排序，特此订正。July、2011.04.27）；  
    第二步、借助堆这个数据结构，找出Top K，时间复杂度为N‘logK。  
        即，借助堆结构，我们可以在log量级的时间内查找和调整/移动。因此，维护一个K(该题目中是10)大小的小根堆，然后遍历300万的Query，分别和根元素进行对比所以，我们最终的时间复杂度是：O（N） + N'\*O（logK），（N为1000万，N’为300万）。ok，更多，详情，请参考原文。

    或者：采用trie树，关键字域存该查询串出现的次数，没有出现为0。最后用10个元素的最小推来对出现频率进行排序。

**3、有一个1G大小的一个文件，里面每一行是一个词，词的大小不超过16字节，内存限制大小是1M。返回频数最高的100个词。**

    方案：顺序读文件中，对于每个词x，取hash(x)%5000，然后按照该值存到5000个小文件（记为x0,x1,...x4999）中。这样每个文件大概是200k左右。

    如果其中的有的文件超过了1M大小，还可以按照类似的方法继续往下分，直到分解得到的小文件的大小都不超过1M。  
    对每个小文件，统计每个文件中出现的词以及相应的频率（可以采用trie树/hash\_map等），并取出出现频率最大的100个词（可以用含100个结点的最小堆），并把100个词及相应的频率存入文件，这样又得到了5000个文件。下一步就是把这5000个文件进行归并（类似与归并排序）的过程了。

**4、有10个文件，每个文件1G，每个文件的每一行存放的都是用户的query，每个文件的query都可能重复。要求你按照query的频度排序。**

    还是典型的TOP K算法，解决方案如下：  
    方案1：  
    顺序读取10个文件，按照hash(query)%10的结果将query写入到另外10个文件（记为）中。这样新生成的文件每个的大小大约也1G（假设hash函数是随机的）。  
      
    找一台内存在2G左右的机器，依次对用hash\_map(query, query\_count)来统计每个query出现的次数。利用快速/堆/归并排序按照出现次数进行排序。将排序好的query和对应的query\_cout输出到文件中。这样得到了10个排好序的文件（记为）。

    对这10个文件进行归并排序（内排序与外排序相结合）。

    方案2：  
     一般query的总量是有限的，只是重复的次数比较多而已，可能对于所有的query，一次性就可以加入到内存了。这样，我们就可以采用trie树/hash\_map等直接来统计每个query出现的次数，然后按出现次数做快速/堆/归并排序就可以了。

    方案3：  
    与方案1类似，但在做完hash，分成多个文件后，可以交给多个文件来处理，采用分布式的架构来处理（比如MapReduce），最后再进行合并。

**5、 给定a、b两个文件，各存放50亿个url，每个url各占64字节，内存限制是4G，让你找出a、b文件共同的url？**

    方案1：可以估计每个文件安的大小为5G×64=320G，远远大于内存限制的4G。所以不可能将其完全加载到内存中处理。考虑采取分而治之的方法。

    遍历文件a，对每个url求取hash(url)%1000，然后根据所取得的值将url分别存储到1000个小文件（记为a0,a1,...,a999）中。这样每个小文件的大约为300M。

    遍历文件b，采取和a相同的方式将url分别存储到1000小文件（记为b0,b1,...,b999）。这样处理后，所有可能相同的url都在对应的小文件（a0vsb0,a1vsb1,...,a999vsb999）中，不对应的小文件不可能有相同的url。然后我们只要求出1000对小文件中相同的url即可。

    求每对小文件中相同的url时，可以把其中一个小文件的url存储到hash\_set中。然后遍历另一个小文件的每个url，看其是否在刚才构建的hash\_set中，如果是，那么就是共同的url，存到文件里面就可以了。

    方案2：如果允许有一定的错误率，可以使用Bloom filter，4G内存大概可以表示340亿bit。将其中一个文件中的url使用Bloom filter映射为这340亿bit，然后挨个读取另外一个文件的url，检查是否与Bloom filter，如果是，那么该url应该是共同的url（注意会有一定的错误率）。

    Bloom filter日后会在本BLOG内详细阐述。

**6、在2.5亿个整数中找出不重复的整数，注，内存不足以容纳这2.5亿个整数。**

    方案1：采用2-Bitmap（每个数分配2bit，00表示不存在，01表示出现一次，10表示多次，11无意义）进行，共需内存2^32 \* 2 bit=1 GB内存，还可以接受。然后扫描这2.5亿个整数，查看Bitmap中相对应位，如果是00变01，01变10，10保持不变。所描完事后，查看bitmap，把对应位是01的整数输出即可。

    方案2：也可采用与第1题类似的方法，进行划分小文件的方法。然后在小文件中找出不重复的整数，并排序。然后再进行归并，注意去除重复的元素。

**7、腾讯面试题：给40亿个不重复的unsigned int的整数，没排过序的，然后再给一个数，如何快速判断这个数是否在那40亿个数当中？**

    与上第6题类似，我的第一反应时快速排序+二分查找。以下是其它更好的方法：  
    **方案1：**oo，申请512M的内存，一个bit位代表一个unsigned int值。读入40亿个数，设置相应的bit位，读入要查询的数，查看相应bit位是否为1，为1表示存在，为0表示不存在。

    dizengrong：  
    **方案2：**这个问题在《编程珠玑》里有很好的描述，大家可以参考下面的思路，探讨一下：  
又因为2^32为40亿多，所以给定一个数可能在，也可能不在其中；  
这里我们把40亿个数中的每一个用32位的二进制来表示  
假设这40亿个数开始放在一个文件中。

    然后将这40亿个数分成两类:  
      1.最高位为0  
      2.最高位为1  
    并将这两类分别写入到两个文件中，其中一个文件中数的个数<=20亿，而另一个>=20亿（这相当于折半了）；  
与要查找的数的最高位比较并接着进入相应的文件再查找

    再然后把这个文件为又分成两类:  
      1.次最高位为0  
      2.次最高位为1

    并将这两类分别写入到两个文件中，其中一个文件中数的个数<=10亿，而另一个>=10亿（这相当于折半了）；  
    与要查找的数的次最高位比较并接着进入相应的文件再查找。  
    .......  
    以此类推，就可以找到了,而且时间复杂度为O(logn)，方案2完。

**附：**这里，再简单介绍下，位图方法：  
    使用位图法判断整形数组是否存在重复   
    判断集合中存在重复是常见编程任务之一，当集合中数据量比较大时我们通常希望少进行几次扫描，这时双重循环法就不可取了。

    位图法比较适合于这种情况，它的做法是按照集合中最大元素max创建一个长度为max+1的新数组，然后再次扫描原数组，遇到几就给新数组的第几位置上1，如遇到5就给新数组的第六个元素置1，这样下次再遇到5想置位时发现新数组的第六个元素已经是1了，这说明这次的数据肯定和以前的数据存在着重复。这种给新数组初始化时置零其后置一的做法类似于位图的处理方法故称位图法。它的运算次数最坏的情况为2N。如果已知数组的最大值即能事先给新数组定长的话效率还能提高一倍。

    欢迎，有更好的思路，或方法，共同交流。

**8、怎么在海量数据中找出重复次数最多的一个？**     
    方案1：先做hash，然后求模映射为小文件，求出每个小文件中重复次数最多的一个，并记录重复次数。然后找出上一步求出的数据中重复次数最多的一个就是所求（具体参考前面的题）。

**9、上千万或上亿数据（有重复），统计其中出现次数最多的钱N个数据。**

    方案1：上千万或上亿的数据，现在的机器的内存应该能存下。所以考虑采用hash\_map/搜索二叉树/红黑树等来进行统计次数。然后就是取出前N个出现次数最多的数据了，可以用第2题提到的堆机制完成。

**10、一个文本文件，大约有一万行，每行一个词，要求统计出其中最频繁出现的前10个词，请给出思想，给出时间复杂度分析。**

    方案1：这题是考虑时间效率。用trie树统计每个词出现的次数，时间复杂度是O(n\*le)（le表示单词的平准长度）。然后是找出出现最频繁的前10个词，可以用堆来实现，前面的题中已经讲到了，时间复杂度是O(n\*lg10)。所以总的时间复杂度，是O(n\*le)与O(n\*lg10)中较大的哪一个。

**附、100w个数中找出最大的100个数。**

    方案1：在前面的题中，我们已经提到了，用一个含100个元素的最小堆完成。复杂度为O(100w\*lg100)。

    方案2：采用快速排序的思想，每次分割之后只考虑比轴大的一部分，知道比轴大的一部分在比100多的时候，采用传统排序算法排序，取前100个。复杂度为O(100w\*100)。

    方案3：采用局部淘汰法。选取前100个元素，并排序，记为序列L。然后一次扫描剩余的元素x，与排好序的100个元素中最小的元素比，如果比这个最小的要大，那么把这个最小的元素删除，并把x利用插入排序的思想，插入到序列L中。依次循环，知道扫描了所有的元素。复杂度为O(100w\*100)。