算法和计算机基础概念

1. **快照、备份、镜像**

### 快照

快照是一种基于时间点的数据拷贝技术，快照的目的在于能够记录出某一个时刻的数据信息并将其保存，如果之后发生某些故障需要数据恢复的时候，可以通过快照来将数据恢复到之前时间点的状态，而该时间点之后的数据都会丢失。备份系统是快照技术的主要应用领域之一，当备份软件需要备份某些不能停止运行的关键业务的时候，利用快照技术可以将某时间点的所有数据信息保存并备份，不会影响到业务的正常运行。

快照技术分为两类：物理拷贝和逻辑拷贝，物理拷贝就是对原始数据的完全拷贝；逻辑拷贝就是只针对发生过改变的数据进行拷贝。两种拷贝技术虽然都能够将数据恢复到某一个时间点，但是其也各有有缺点：

物理拷贝的优点是管理简单，不需要监控目标数据的状态，直接将所有数据拷贝到另外一个地方，而且可以作为数据备份直接保存起来。它的缺点是需要最大的存储空间，需要和目标数据一样大的空间才能将其完全拷贝下来。

逻辑拷贝的优点就是节省空间，一般来说，经常发生改变的数据只占所有数据的20％－30％，这样逻辑备份可以节省出70％左右的存储空间。但是逻辑备份也有它的缺点，因为它只是保存了发生改变的数据，所以如果目标数据发生损坏的话，快照也无能为力。当前文件系统和备份软件流行的写入时拷贝技术（copy on write）就是属于逻辑拷贝。

### 备份

备份是一种目的，快照是备份的一个技术手段，包括镜像也算是备份的一种技术手段。

### 镜像

镜像（Mirroring）是冗余的一种类型，一个磁盘上的数据在另一个磁盘上存在一个完全相同的副本即为镜像。

镜像的目的是为了保证数据冗余，在数据源发生故障的时候迅速恢复。如果用户将某个文件误删除，那么镜像文件下的该文件也会丢失，无法恢复。

反过来说，如果用户的目标数据源损坏，所有数据丢失，而镜像可以迅速恢复出所有的数据，保证业务的连续性。

1. **查找算法**

### 顺序查找

顺序查找也称为线形查找，属于无序查找算法。从数据结构线形表的一端开始，顺序扫描，依次将扫描到的结点关键字与给定值k相比较，若相等则表示查找成功；若扫描结束仍没有找到关键字等于k的结点，表示查找失败。

复杂度分析：查找成功时的平均查找长度为：（假设每个数据元素的概率相等） ASL = 1/n(1+2+3+…+n) = (n+1)/2 ;当查找不成功时，需要n+1次比较，时间复杂度为O(n);所以，顺序查找的时间复杂度为O(n)。

### 二分查找

说明：元素必须是有序的，如果是无序的则要先进行排序操作。

基本思想：也称为是折半查找，属于有序查找算法。用给定值k先与中间结点的关键字比较，中间结点把线形表分成两个子表，若相等则查找成功；若不相等，再根据k与该中间结点关键字的比较结果确定下一步查找哪个子表，这样递归进行，直到查找到或查找结束发现表中没有这样的结点。

复杂度分析：最坏情况下，关键词比较次数为+1，且期望时间复杂度为O()；

### 插值查找

说明：元素必须是有序的，如果是无序的则要先进行排序操作。这是对二分查找的改进

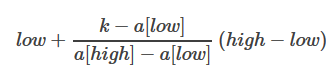
现在我们的新问题是，为什么一定要折半，而不是折四分之一或者折更多呢？

例如，在英文词典里查”apple”,你下意识里翻开词典是翻前面的书页还是后面的书页呢？如果再让你查”zoo”，你又怎么查？很显然，这里你绝对不会是从中间开始查起，而是有一定目的的往前或往后翻。

基于折半查找代码，我们略微变换等式后得到：



也就是mid等于最低下标low加上最高下标high与low的差的一半。算法科学家们考虑的就是将这个1/2进行改进，改进为下面的计算方案：



将1/2改成了有什么道理呢？假设a[11]={0,1,16,24,35,47,59,62,73,88,99}，low=1，high=11，则a[low]=1，a[high]=99，如果我们要找的是key=16时，按原来的折半查找的做法，我们需要四次才可以得到结果，但如果使用新办法

即mid≈1+0.153\*(10-1)=2.377取整得到mid=2，我们只需要两次查找到结果了，显著大大提高了查找的效率。

时间复杂度: O()

### 斐波那契查找

斐波那契查找的前提是待查找的查找表必须顺序存储并且有序。

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)。

名词解析：

斐波那契数列，f(n)=f(n-1)+f(n-2)

例如：1 1 2 3 5 8 13 21 34 55 89…

斐波那契查找的步骤：

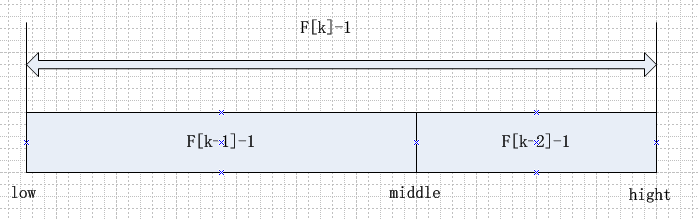
（1）首先我们要根据数组的长度大小构建一个斐波那契数列

比如我们的数组[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]

根据规则：数组长度为某个斐波那契数小1，及n=F（k）-1;n=20,F(k)=21。

因此我们构建的斐波那契数列长度至少为8（也可以更长，但是耗存储，8是最基本的要求），[1,1,2,3,5,8,13,21].

注：如果我们的数组长度不为n=F（k）-1，比如我们的数组长度是17，F(k)的取值还是21，这时用最大的数来填充残缺部分，也就是说我们还是要把数组长度设置为20，但是有数据的只有前17，后3个我们设置为a[16]，最大值是17，残缺部分的所有数值都是17。



（2）此时就是寻找拆分点，比如我们要查找的值是b=14，拆分点就是拿给14做比对的。

第一次拆分点计算：middle = low + fb[k - 1] - 1;k值是斐波那契数列长度8，middle = 0+13-1=12，也就是确认拿数组中的a[12]=12和b进行比较。

比较会有3种情况：

1）相等，mid位置的元素即为所求

2）b> a[12] ,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))= F(k)-1-F(k-1)= F(k)-F(k-1)-1=F(k-2)-1个，所以可以递归的应用斐波那契查找.此处也就是说k-2是因为数组的长度变化所决定的。

3）b< a[12] ,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.

然后不断重复上述操作就可以了。

代码实现：

**public** **class** FbonacciSearch {

**public** **static** **void** main(String[] args) {

// **TODO** Auto-generated method stub

**int**[] array = { 1, 5, 15, 22, 25, 31, 39, 42, 47, 49, 59, 68, 88, 88,

88, 88, 88 };

System.***out***.println("result: " + *fbSearch*(array, 59));

}

/\*\*

\* name: fbSearch

\* description: 斐波那契查找实现

\* return: int

\*/

**public** **static** **int** fbSearch(**int**[] array, **int** a) {

**if** (array == **null** || array.length == 0) {

**return** -1;

} **else** {

**int** length = array.length;

**int**[] fb = *makeFbArray*(8);// 制造一个长度为10的斐波数列

**int** k = 0;

**while** (length > fb[k] - 1) {// 找出数组的长度在斐波数列（减1）中的位置，将决定如何拆分

k++;

}

**int**[] temp = Arrays.*copyOf*(array, fb[k] - 1);// 构造一个长度为fb[k] - 1的新数列

**for** (**int** i = length; i < temp.length; i++) {

**if** (i >= length) {

temp[i] = array[length - 1];

}

}

**int** low = 0;

**int** hight = array.length - 1;

**while** (low <= hight) {

**int** middle = low + fb[k - 1] - 1;

**if** (temp[middle] > a) {

hight = middle - 1;

k = k - 1;

} **else** **if** (temp[middle] < a) {

low = middle + 1;

k = k - 2;

} **else** {

**if** (middle <= hight) {

**return** middle;// 若相等则说明mid即为查找到的位置

} **else** {

**return** hight;// middle的值已经大于hight,进入扩展数组的填充部分,即最后一个数就是要查找的数。

}

}

}

**return** -1;

// return recurse(array, fb, a, low, hight, k);

}

}

/\*\*

\* name: makeFbArray

\* description: 生成指定长度的斐波数列

\* return: int[]

\*/

**public** **static** **int**[] makeFbArray(**int** length) {

**int**[] array = **null**;

**if** (length > 2) {

array = **new** **int**[length];

array[0] = 1;

array[1] = 1;

**for** (**int** i = 2; i < length; i++) {

array[i] = array[i - 1] + array[i - 2];

}

}

**return** array;

}

/\*\*

\* name: recurse

\* description: 递归实现，可以来代替while (low <= hight)遍历

\* return: int

\*/

**public** **static** **int** recurse(**int**[] array, **int**[] fb, **int** a, **int** low, **int** hight,

**int** k) {

**if** (array == **null** || array.length == 0 || a < array[low]

|| a > array[hight] || low > hight) {

**return** -1;

}

**int** middle = low + fb[k - 1] - 1;

**if** (a < array[middle]) {

**return** *recurse*(array, fb, a, low, middle - 1, k - 1);

} **else** **if** (a > array[middle]) {

**return** *recurse*(array, fb, a, middle + 1, hight, k - 2);

} **else** {

**if** (middle <= hight) {

**return** middle;

} **else** {

**return** hight;

}

}

}

}

### logN和是一样的

1. **树**

**二叉树的性质：**

1. 在非空二叉树中，第i层的结点总数不超过2i-1, i>=1;
2. 具有n个结点的完全二叉树的深度为log2(n+1);

3) 深度为h的二叉树最多有2h-1个结点(h>=1)，最少有h个结点;

4) 对于任意一棵二叉树，如果其叶结点数为n0，而度数为2的结点总数为n2，则n0=n2+1;

5)有N个结点的完全二叉树各结点如果用顺序方式存储，则结点之间有如下关系：

若I为结点编号则 如果I>1，则其父结点的编号为I/2；

如果2I<=N，则其左儿子（即左子树的根结点）的编号为2I；若2I>N，则无左儿子；

如果2I+1<=N，则其右儿子的结点编号为2I+1；若2I+1>N，则无右儿子。

6)给定N个节点，能构成h(N)种不同的二叉树，其中h(N)为卡特兰数的第N项，h(n)=C(2\*n, n)/(n+1)。

7)设有i个枝点，I为所有枝点的道路长度总和，J为叶的道路长度总和J=I+2i。（？）

**二叉树：**

二叉树是数据结构中一种重要的数据结构，也是树表家族最为基础的结构。

**二叉树的定义：**二叉树的每个结点至多只有二棵子树(不存在度大于2的结点)，二叉树的子树有左右之分，次序不能颠倒。二叉树的第i层至多有2i-1个结点；深度为k的二叉树至多有2k-1个结点；对任何一棵二叉树T，如果其终端结点数为n0，度为2的结点数为n2，则n0=n2+1。如下：n0 = 6, n2 = 5 n2 (A B C D E) 其余是n0



**满二叉树：**

除最后一层无任何子节点外，每一层上的所有结点都有两个子结点。也可以这样理解，除叶子结点外的所有结点均有两个子结点。节点数达到最大值，所有叶子结点必须在同一层上。

满二叉树的性质：

1) 一颗树深度为h，最大层数为k，深度与最大层数相同，k=h;

2) 叶子数为2h;

3) 第k层的结点数是：2k-1;

4) 总结点数是：2k-1，且总节点数一定是奇数。

**完全二叉树：**

若设二叉树的深度为h，除第 h 层外，其它各层 (1～(h-1)层) 的结点数都达到最大个数，第h层所有的结点都连续集中在最左边，这就是完全二叉树。

完全二叉树是效率很高的数据结构，堆是一种完全二叉树或者近似完全二叉树，所以效率极高，像十分常用的排序算法、Dijkstra算法、Prim算法等都要用堆才能优化，二叉排序树的效率也要借助平衡性来提高，而平衡性基于完全二叉树。



**二叉查找树：**

**二叉查找树定义**：又称为是二叉排序树（Binary Sort Tree）或二叉搜索树。二叉排序树或者是一棵空树，或者是具有下列性质的二叉树：

1) 若左子树不空，则左子树上所有结点的值均小于它的根结点的值；

2) 若右子树不空，则右子树上所有结点的值均大于或等于它的根结点的值；

3) 左、右子树也分别为二叉排序树；

4) 没有键值相等的节点

二叉查找树的性质：对二叉查找树进行中序遍历，即可得到有序的数列。

**二叉查找树的时间复杂度**：它和二分查找一样，插入和查找的时间复杂度均为O()，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（比如，我们查找上图（b）中的“93”，我们需要进行n次查找操作）。我们追求的是在最坏的情况下仍然有较好的时间复杂度，这就是平衡查找树设计的初衷。

**二叉查找树的插入**过程如下：

1) 若当前的二叉查找树为空，则插入的元素为根节点;

2) 若插入的元素值小于根节点值，则将元素插入到左子树中;

3) 若插入的元素值不小于根节点值，则将元素插入到右子树中。

**二叉查找树的删除**，分三种情况进行处理：

1) z为叶子节点，直接删除该节点，再修改其父节点的指针（注意分是根节点和不是根节点），如图a;



2) z为单支节点（即只有左子树或右子树）。让z的子树与z的父亲节点相连，删除z即可（注意分是根节点和不是根节点），如图b;



3) z的左子树和右子树均不空。找到z的后继y，因为y一定没有左子树，所以可以删除y，并让y的父亲节点成为y的右子树的父亲节点，并用y的值代替z的值；或者方法二是找到z的前驱x，x一定没有右子树，所以可以删除x，并让x的父亲节点成为y的左子树的父亲节点。如图c。



**平衡二叉树：**

我们知道，对于一般的二叉搜索树（Binary Search Tree），其期望高度（即为一棵平衡树时）为log2n，其各操作的时间复杂度O(log2n)同时也由此而决定。但是，在某些极端的情况下（如在插入的序列是有序的时），二叉搜索树将退化成近似链或链，此时，其操作的时间复杂度将退化成线性的，即O(n)。我们可以通过随机化建立二叉搜索树来尽量的避免这种情况，但是在进行了多次的操作之后，由于在删除时，我们总是选择将待删除节点的后继代替它本身，这样就会造成总是右边的节点数目减少，以至于树向左偏沉。这同时也会造成树的平衡性受到破坏，提高它的操作的时间复杂度。于是就有了我们下边介绍的平衡二叉树。

**平衡二叉树定义**：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（平衡因子：BF，也就是高度差，BF>1则失衡），并且左右两个子树都是一棵平衡二叉树。平衡二叉树的常用算法有红黑树、AVL树等。在平衡二叉搜索树中，我们可以看到，其高度一般都良好地维持在O(log2n)，大大降低了操作的时间复杂度。

最小二叉平衡树的节点的公式如下：

F(n)=F(n-1)+F(n-2)+1

这个类似于一个递归的数列，可以参考Fibonacci数列，1是根节点，F(n-1)是左子树的节点数量，F(n-2)是右子树的节点数量。

**平衡查找树之AVL树：**

**AVL树定义：**AVL树是最先发明的自平衡二叉查找树。AVL树得名于它的发明者 G.M. Adelson-Velsky 和 E.M. Landis，他们在 1962 年的论文 “An algorithm for the organization of information” 中发表了它。在AVL中任何节点的两个儿子子树的高度最大差别为1，所以它也被称为高度平衡树，n个结点的AVL树最大深度约1.44 log2n。查找、插入和删除在平均和最坏情况下都是O(log2n)。增加和删除可能需要通过一次或多次树旋转来重新平衡这个树。**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(**log2n**)。但是频繁旋转会使插入和删除牺牲掉O(**log2n**)左右的时间，不过相对二叉查找树来说，时间上稳定了很多。**

**AVL树的自平衡操作——旋转**：

AVL树最关键的也是最难的一步操作就是旋转。旋转主要是为了实现AVL树在实施了插入和删除操作以后，树重新回到平衡的方法。

**树的插入代码**：

插入数据以后，父节点的平衡因子必然会被改变！

首先判断父节点的平衡因子是否满足性质1（-1<= parent->\_bf <=1），如果满足，则要回溯向上检查插入该节点是否影响了其它节点的平衡因子值！

**1）当父节点的平衡因子等于0时，父节点所在的子树已经平衡，不会影响其他节点的平衡因子了。**

**2）当父节点的平衡因子等于1或者-1时，需要继续向上回溯一层，检验祖父节点的平衡因子是否满足条件（把父节点给当前节点）。**

**3）当父节点的平衡因子等于2或者-2时，不满足性质1，这时需要进行旋转 来降低高度 ：**

**旋转的目的是为了降低高度**

/\*

\* 将结点插入到AVL树中，并返回根节点

\*

\* 参数说明：

\* tree AVL树的根结点

\* key 插入的结点的键值

\* 返回值：

\* 根节点

\*/

**private** AVLTreeNode<T> insert(AVLTreeNode<T> tree, T key) {

**if** (tree == **null**) {

// 新建节点

tree = **new** AVLTreeNode<T>(key, **null**, **null**);

**if** (tree==**null**) {

System.***out***.println("ERROR: create avltree node failed!");

**return** **null**;

}

} **else** {

**int** cmp = key.compareTo(tree.key);

**if** (cmp < 0) { // 应该将key插入到"tree的左子树"的情况

tree.left = insert(tree.left, key);

// 插入节点后，若AVL树失去平衡，则进行相应的调节。

**if** (height(tree.left) - height(tree.right) == 2) {

**if** (key.compareTo(tree.left.key) < 0)

tree = leftLeftRotation(tree);

**else**

tree = leftRightRotation(tree);

}

} **else** **if** (cmp > 0) { // 应该将key插入到"tree的右子树"的情况

tree.right = insert(tree.right, key);

// 插入节点后，若AVL树失去平衡，则进行相应的调节。

**if** (height(tree.right) - height(tree.left) == 2) {

**if** (key.compareTo(tree.right.key) > 0)

tree = rightRightRotation(tree);

**else**

tree = rightLeftRotation(tree);

}

} **else** { // cmp==0

System.***out***.println("添加失败：不允许添加相同的节点！");

}

}

tree.height = max( height(tree.left), height(tree.right)) + 1;

**return** tree;

}

**红黑树：**

R-B Tree，全称是Red-Black Tree，又称为“红黑树”，它一种特殊的二叉查找树。红黑树的每个节点上都有存储位表示节点的颜色，可以是红(Red)或黑(Black)。

**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（NIL）是黑色。 [注意：这里叶子节点，是指为空(NIL或NULL)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。（比如：从根节点到它的每个叶子结点的路径所经过的黑结点数是一样的）**

**注意：**

**(01) 特性(3)中的叶子节点，是只为空(NIL或null)的节点。**

**(02) 特性(5)，确保没有一条路径会比其他路径长出俩倍。因而，红黑树是相对是接近平衡的二叉树。**



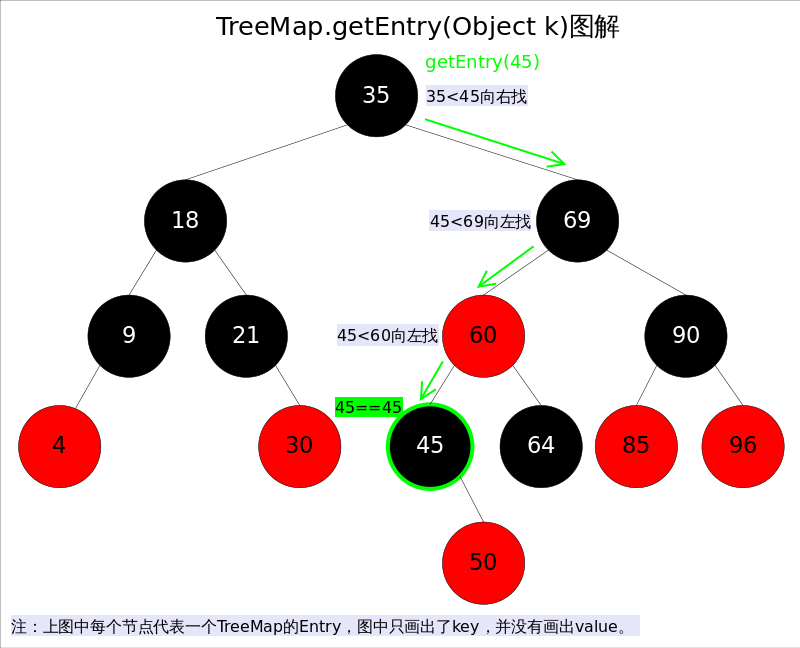
**红黑树的应用：**

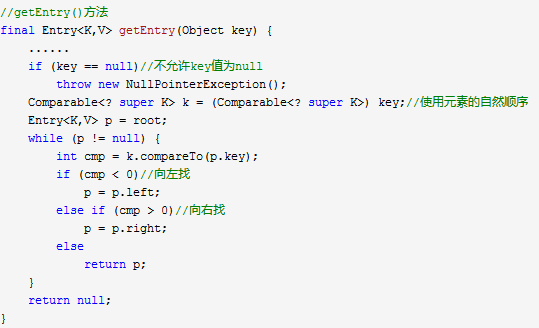
红黑树的应用比较广泛，主要是用它来存储有序的数据，它的时间复杂度是O(lgn)，效率非常之高。

例如，Java集合中的TreeSet和TreeMap，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。

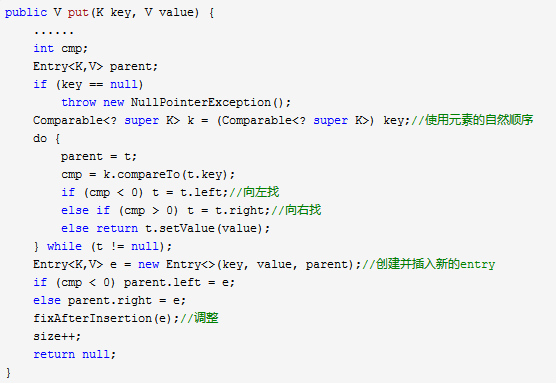
**效率：**时间复杂度是O(logn)

**get（）方法：**





**put()方法：**



//红黑树调整函数fixAfterInsertion()

**private** **void** fixAfterInsertion(Entry<K,V> x) {

x.color = RED;

**while** (x != **null** && x != root && x.parent.color == RED) {

**if** (parentOf(x) == leftOf(parentOf(parentOf(x)))) {

Entry<K,V> y = rightOf(parentOf(parentOf(x)));

**if** (colorOf(y) == RED) {//如果y为null，则视为BLACK

setColor(parentOf(x), BLACK); // 情况1

setColor(y, BLACK); // 情况1

setColor(parentOf(parentOf(x)), RED); // 情况1

x = parentOf(parentOf(x)); // 情况1

} **else** {

**if** (x == rightOf(parentOf(x))) {

x = parentOf(x); // 情况2

rotateLeft(x); // 情况2

}

setColor(parentOf(x), BLACK); // 情况3

setColor(parentOf(parentOf(x)), RED); // 情况3

rotateRight(parentOf(parentOf(x))); // 情况3

}

} **else** {

Entry<K,V> y = leftOf(parentOf(parentOf(x)));

**if** (colorOf(y) == RED) {

setColor(parentOf(x), BLACK); // 情况4

setColor(y, BLACK); // 情况4

setColor(parentOf(parentOf(x)), RED); // 情况4

x = parentOf(parentOf(x)); // 情况4

} **else** {

**if** (x == leftOf(parentOf(x))) {

x = parentOf(x); // 情况5

rotateRight(x); // 情况5

}

setColor(parentOf(x), BLACK); // 情况6

setColor(parentOf(parentOf(x)), RED); // 情况6

rotateLeft(parentOf(parentOf(x))); // 情况6

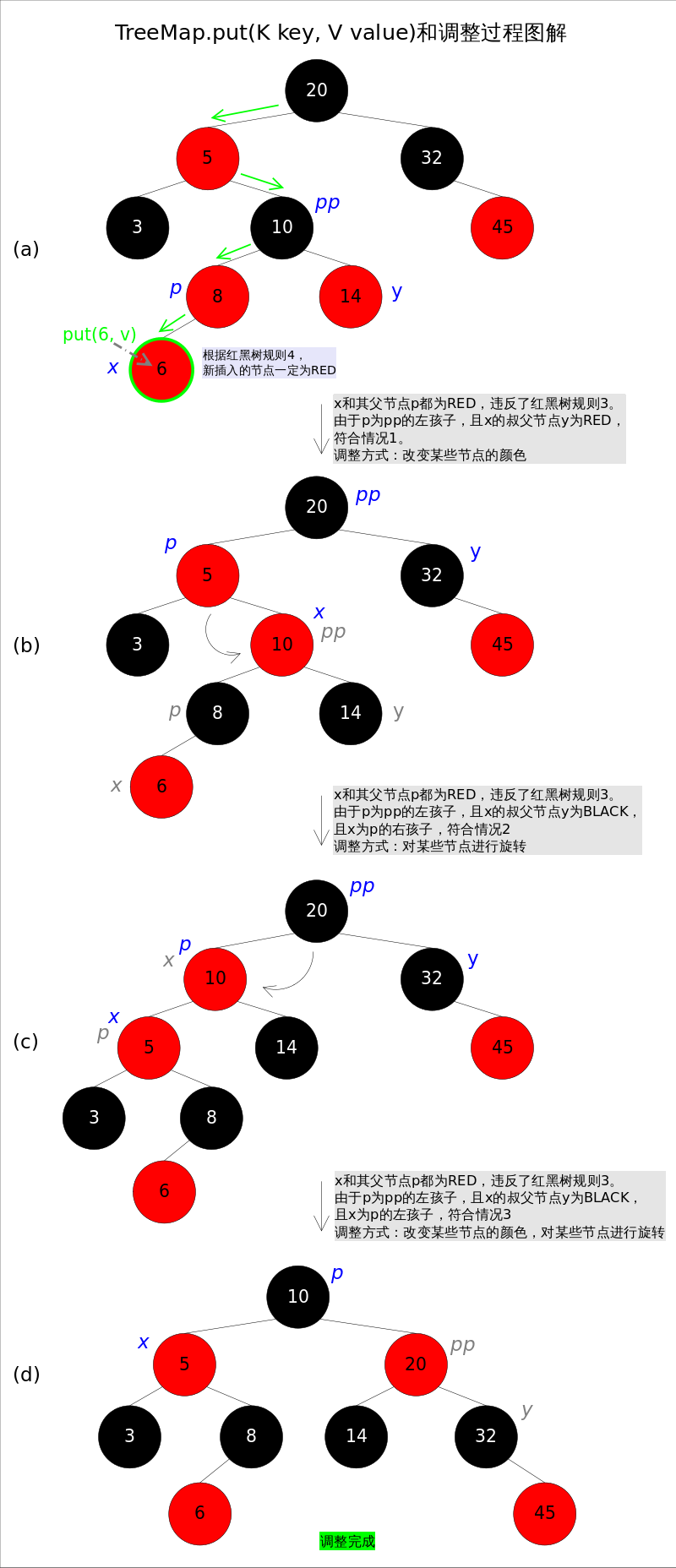
}

}

}

root.color = BLACK;

}



**树的平衡因子：**

每个节点的平衡因子bf 等于右子树的高度减去左子树的高度

**树的旋转：单旋和双旋**

**旋转有两个属性：**轴 和 旋转方向；（旋转轴即是原最小树经过旋转修正后的符合AVL的最小树的根节点）

**旋转轴的确定 ： （干货——单双旋转的旋转轴确定问题）**

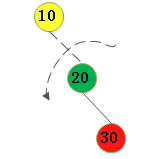
**单旋转：**旋转轴为 不满足AVL条件的最小树的树根的相应孩子节点；

**多旋转：**旋转轴为 不满足AVL条件的最小树的树根的相应孙子节点；

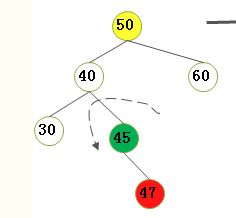
**顺时针为右旋，逆时针为左旋**

**如何判断进行单旋转还是双旋转 （什么时候需要单旋转，而什么时候需要多旋转？）**

**单旋转**： 插入点不介于 满足AVL条件的树根 和 树根对应孩子节点之间；如下图，插入点30，不介于根节点10和其孩子结点20之间

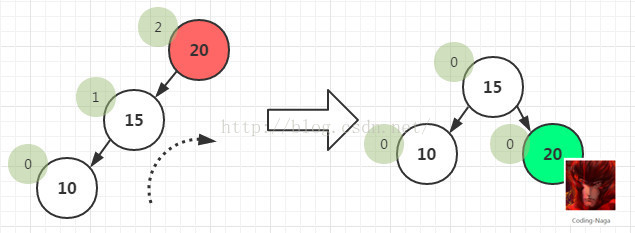


**双旋转**：插入点介于 满足AVL条件的树根 和 树根对应孩子节点之间；如下图，插入点47，介于根节点50和左孩子结点40之间

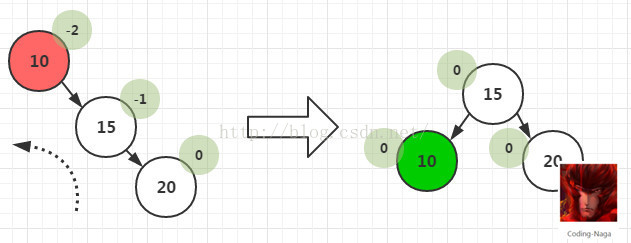


1.0）高度不平衡需要α点的两棵子树高度差为2，故可得高度不平衡可能出现在下面四种情况中：

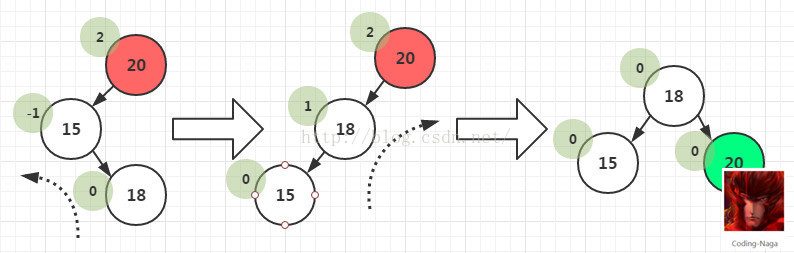
1.在一个节点的左子树的左子树上插入一个新节点10。即LL左左。在这种情况下，我们可以通过将节点右旋使其平衡。如图-2所示；



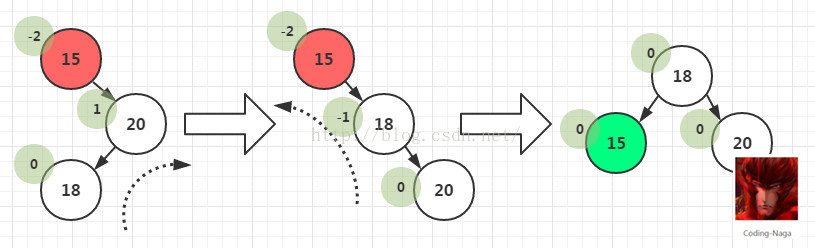
2. 在一个节点的右子树的右子树上插入一个新节点20。即RR右右。在这种情况下，我们可以通过将节点左旋使其平衡。如图-3所示；



3. 在一个节点的左子树的右子树上插入一个新节点18。即LR左右。在这种情况下，我们不能直接通过将节点左旋或右来使其平衡了。这里需要两步来完成，先让树中高度较低的进行一次左旋，这个时候就变成了LL了。再进行一次单右旋操作即可。如图-4所示；



4. 在一个节点的右子树的左子树上插入一个新节点18。即RL右左。在这种情况下，我们不能直接通过将节点左旋或右来使其平衡了。这里需要两步来完成，先让树中高度较低的进行一次右旋，这个时候就变成了RR了。再进行一次单左旋操作即可。如图-5所示；

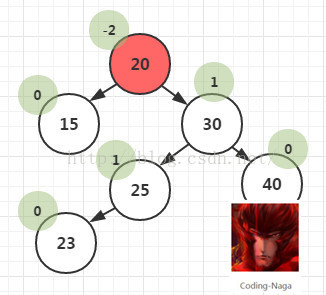


**我们在旋转的过程中，比如左旋，到底是怎么旋转法，以哪个节点为中心点旋转？**

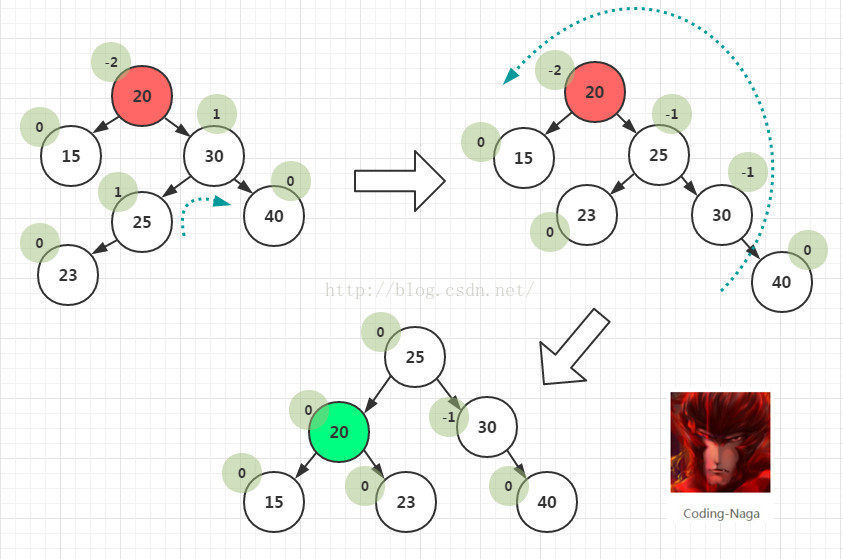
失衡节点的平衡因子的绝对值是大于1的。AVL树失衡类型的判断都基于这个失衡节点的。也就是说LL型需要调整的是失衡节点的左子树的左子树，LR型需要调整的是失衡节点左子树的右子树。是不是有一点绕，看看上面的旋转操作图就应该知道了。那么，对于单旋转操作也就很简单了，就是以中间节点为中心旋转。而双旋转中，因为不是一次旋转，不能在一开始就确定旋转中心点，我们在第一次旋转的过程中，只是旋转后面的两个节点，并保证节点值的大小关系，如图-4所示。后面的问题就转化成了单旋转操作了。

双旋转的轴：相信你也看到了， 双旋转的轴显然是插入点的直接父节点；（除了两个特例） （干货——双旋转的轴显然是插入点的直接父节点（除了两个特例， 而两个特例的轴是插入点本身））

{20, 15, 30, 40, 25, 23}插入后的树



这时我们就不能直接进行一次左旋或是右旋就可以搞定的了。虽然我们不能一步直接达到我们的要求，但是分两步就可以了呀。操作图示参见图-7.

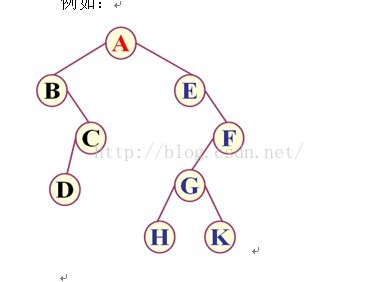


**二叉树的遍历：**

先序遍历：遍历顺序规则为【根左右】

中序遍历：遍历顺序规则为【左根右】

后序遍历：遍历顺序规则为【左右根】



先序遍历：ABCDEFGHK

中序遍历：BDCAEHGKF

后序遍历：DCBHKGFEA

**以中序遍历为例：**

中序遍历的规则是【左根右】，我们从root节点A看起；

此时A是根节点，遍历A的左子树；

A的左子树存在，找到B，此时B看做根节点，遍历B的左子树；

B的左子树不存在，返回B，根据【左根右】的遍历规则，记录B，遍历B的右子树；

B的右子树存在，找到C，此时C看做根节点，遍历C的左子树；

C的左子树存在，找到D，由于D是叶子节点，无左子树，记录D，无右子树，返回C，根据【左根右】的遍历规则，记录C，遍历C的右子树；

C的右子树不存在，返回B，B的右子树遍历完，返回A；

至此，A的左子树遍历完毕，根据【左根右】的遍历规则，记录A，遍历A的右子树；

A的右子树存在，找到E，此时E看做根节点，遍历E的左子树；

E的左子树不存在，返回E，根据【左根右】的遍历规则，记录E，遍历E的右子树；

E的右子树存在，找到F，此时F看做根节点，遍历F的左子树；

F的左子树存在，找到G，此时G看做根节点，遍历G的左子树；

G的左子树存在，找到H，由于H是叶子节点，无左子树，记录H，无右子树，返回G，根据【左根右】的遍历规则，记录G，遍历G的右子树；

G的右子树存在，找到K，由于K是叶子节点，无左子树，记录K，无右子树，返回G，根据【左根右】的遍历规则，记录F，遍历F的右子树；

F的右子树不存在，返回F，E的右子树遍历完毕，返回A；

至此，A的右子树也遍历完毕；

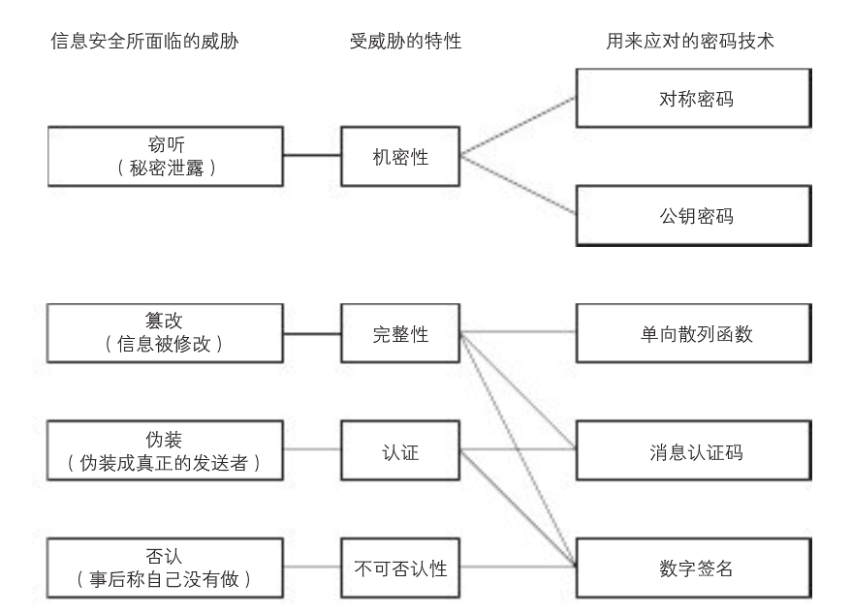
**前驱和后继：**

按照某种顺序遍历二叉树后在这个遍历中位于某一节点前面和后面的结点。

以上面的中序遍历来说，中序遍历：BDCAEHGKF，结点如果是D，那么D的前驱就是结点B，后继就是结点C

1. **加密算法**

**数据存储和传输存在的风险：**



**base64：**

**单向散列函数(MD5 加密)：**

MD5，全名Message Digest Algorithm 5 ，中文名为消息摘要算法，为计算机安全领域广泛使用的一种散列函数，用以提供消息的完整性保护。它属Hash算法一类。MD5算法对输入任意长度的消息进行运行，产生一个128位的消息摘要。它是压缩+加密+hash算法的结合体，是绝对不可逆的。

**特点**：

（1）长度固定:不管多长的字符串,加密后长度都是一样长。

作用:方便平时信息的统计和管理

（2）易计算:字符串和文件加密的过程是容易的。

作用: 开发者很容易理解和做出加密工具

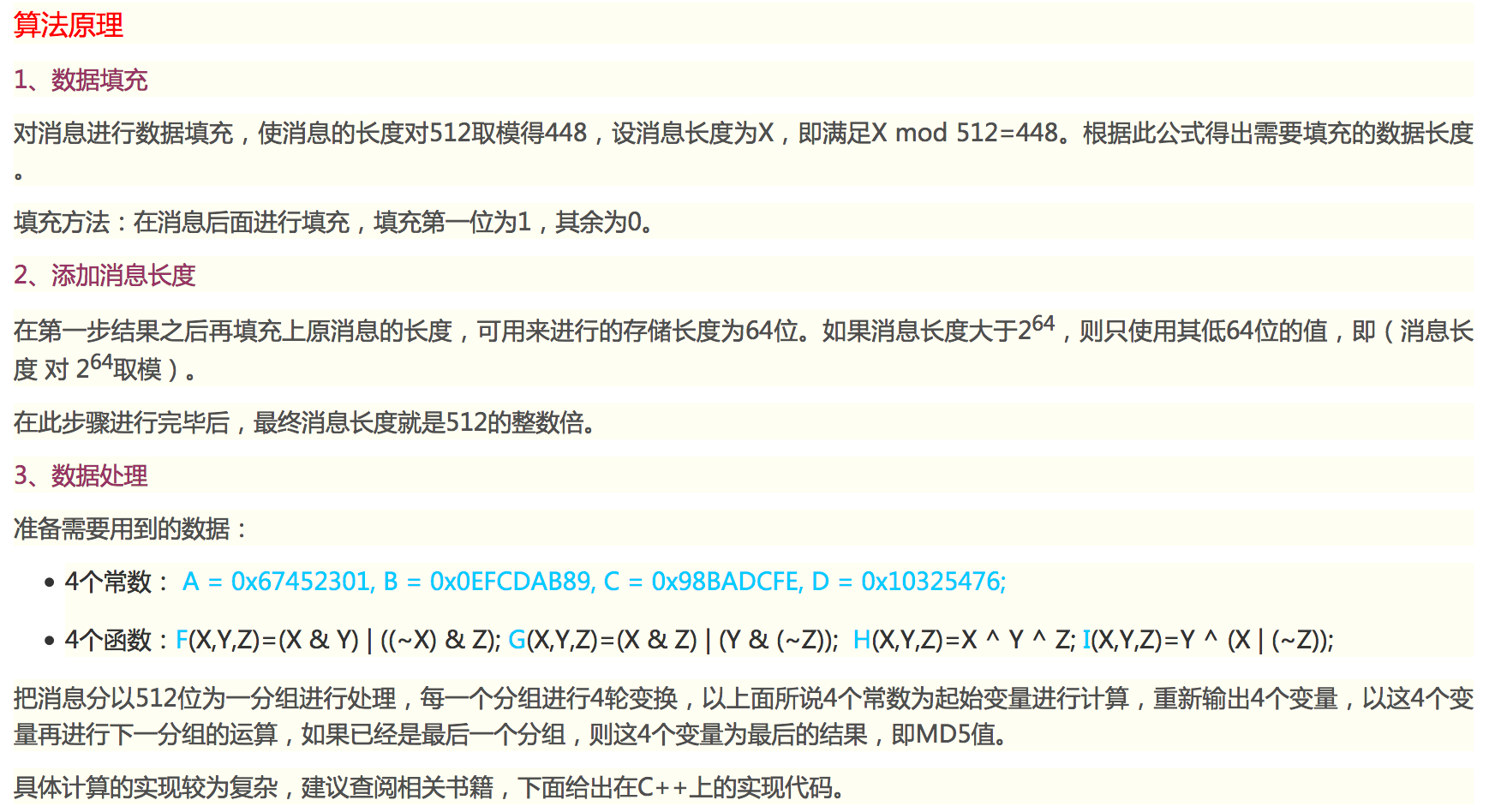
（3）细微性：一个文件,不管多大,小到几k,大到几G,你只要改变里面某个字符,那么都会导致MD5值改变。

作用:很多软件和应用在网站提供下载资源,其中包含了对文件的MD5码,用户下载后只需要用工具测一下下载好的文件,通过对比就知道该文件是否有过更改变动。

（4）不可逆性：你明明知道密文和加密方式,你却无法反向计算出原密码。

作用:基于这个特点,很多安全的加密方式都是用到.大大提高了数据的安全性。

**原理：**



**撞库破解:**

这是概率极低的破解方法,原理就是:

（1）建立一个大型的数据库,把日常的各个语句,通过MD5加密成为密文,不断的积累大量的句子,放在一个庞大的数据库里。

（2）比如一个人拿到了别人的密文,想去查询真实的密码,就需要那这个密文去到提供这个数据库的公司网站去查询.

这就是撞库的概念。

**MD5加盐:**

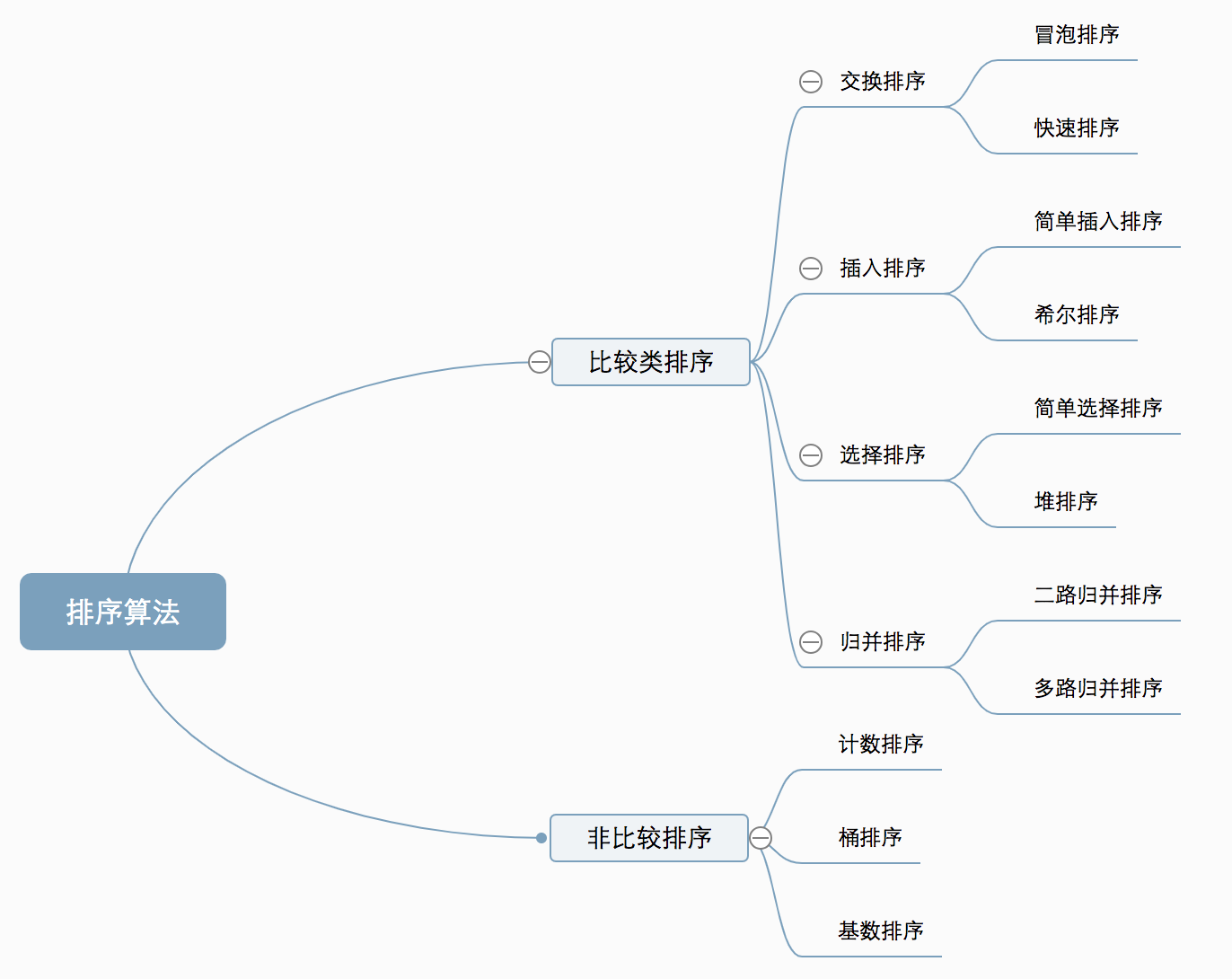
比如我的银行密码是”12345”

1. 得到的MD5是:827ccb0eea8a706c4c34a16891f84e7b
2. 一个人截取到这个密文,那么通过撞库肯定容易撞出12345.
3. 我们要做的就是加盐,银行密码还是”12345”,然后我把银行密码加上我特定的字符串才计算MD5

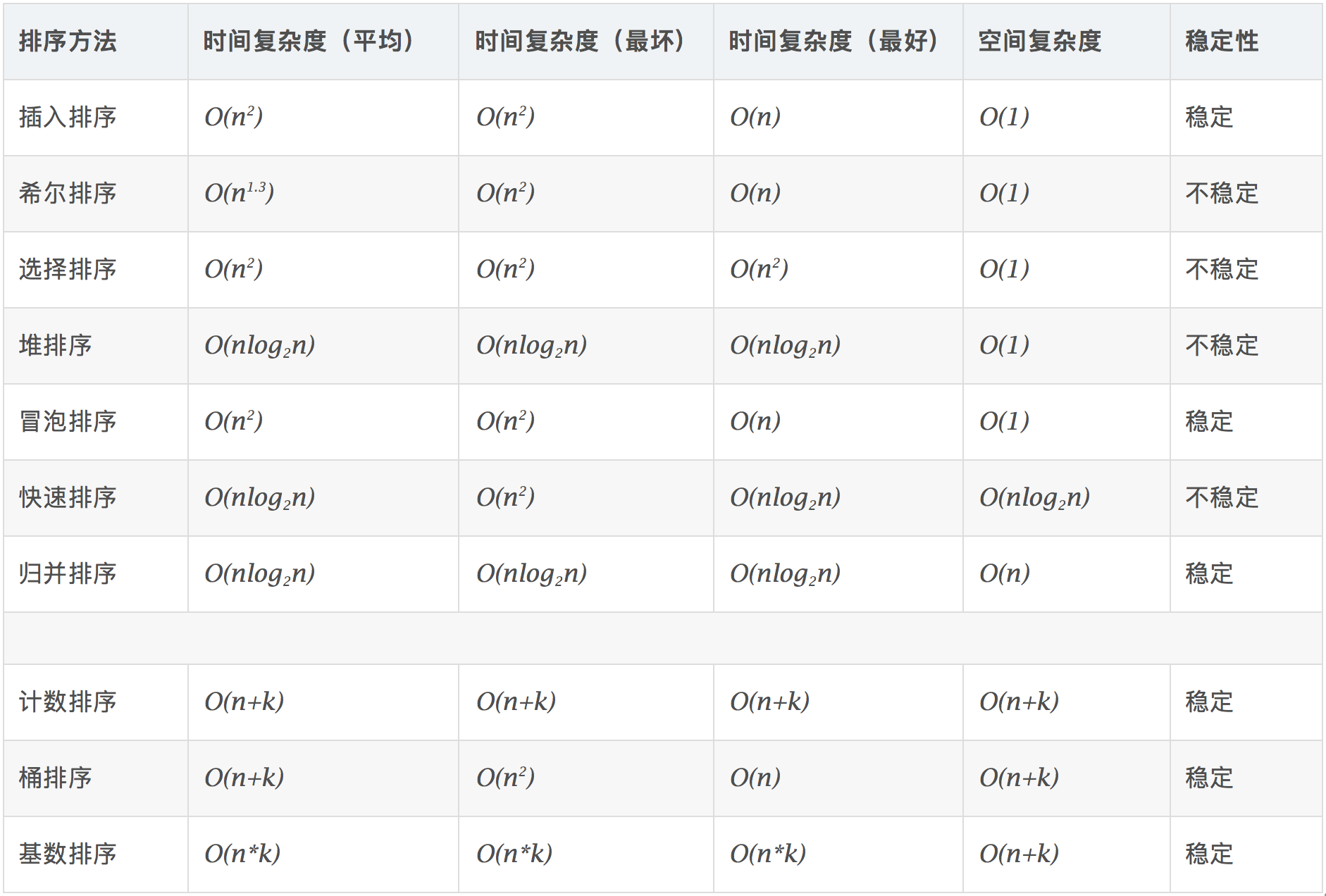
所以密码还是那个密码,但是变成求”12345密码加盐987”的MD5值,然后再得到MD5,那么这个MD5起码可以确认那个数据库不会有。

1. **排序算法**

算法分类



算法复杂度

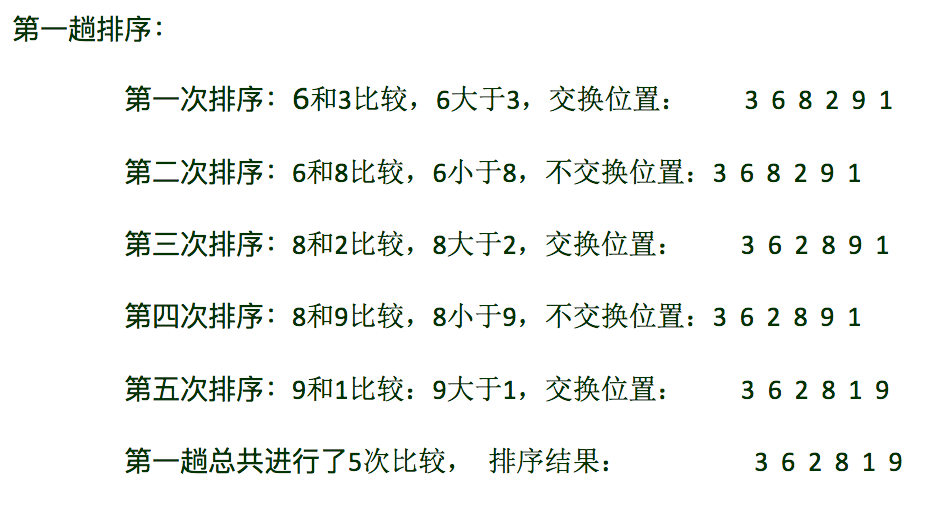


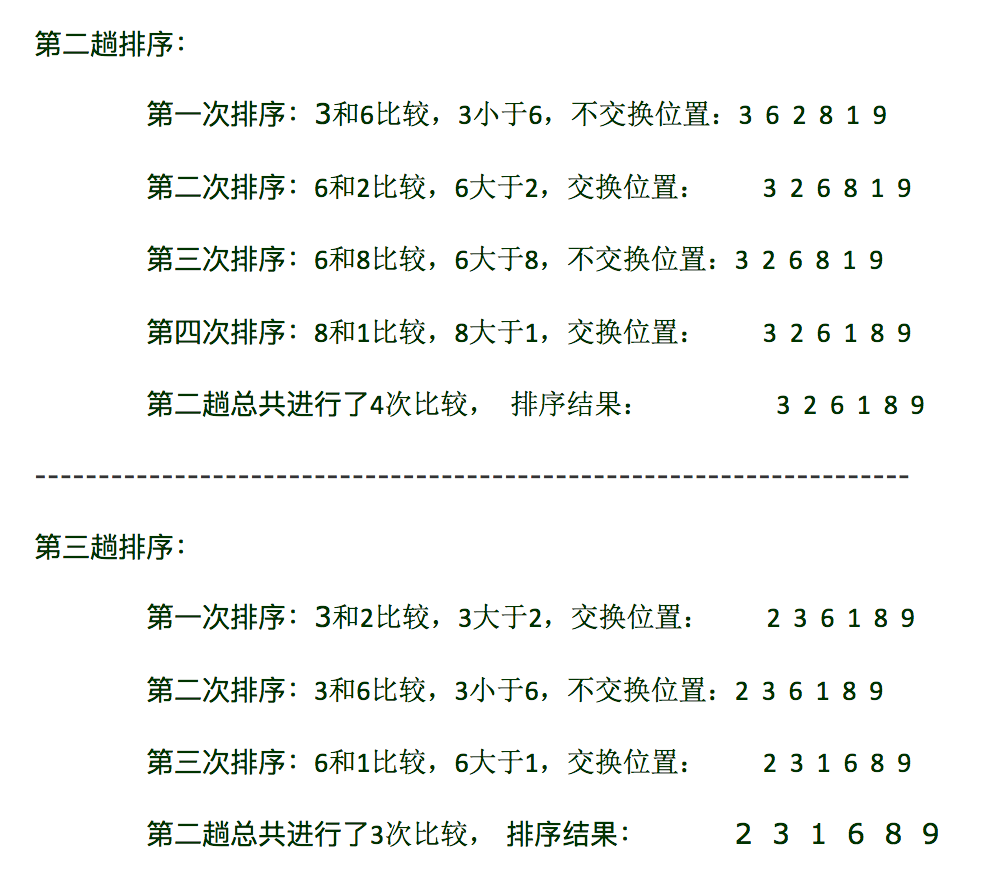
**冒泡排序：**

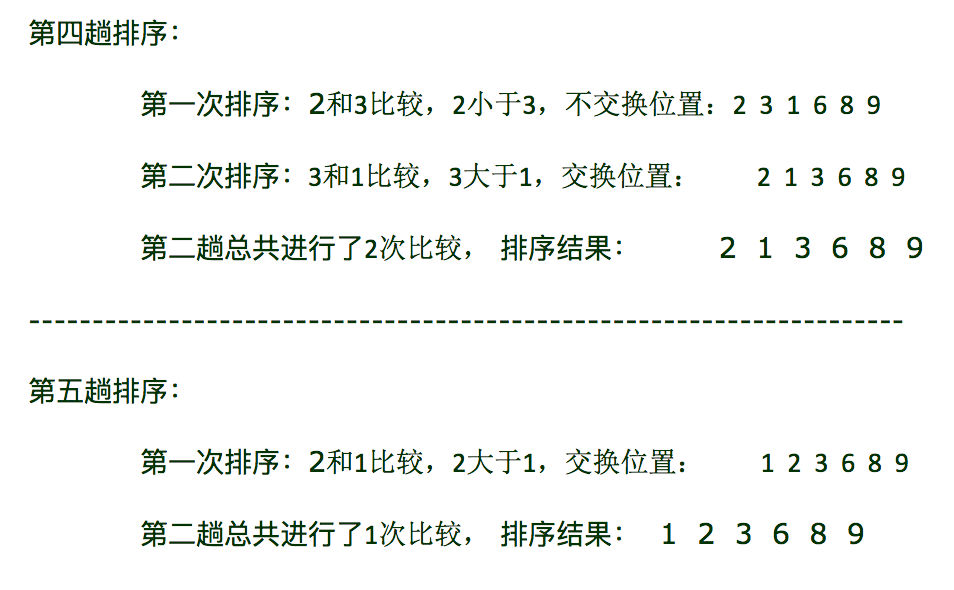
原理：比较两个相邻的元素，将值大的元素交换至右端。

思路：依次比较相邻的两个数，将小数放在前面，大数放在后面。即在第一趟：首先比较第1个和第2个数，将小数放前，大数放后。然后比较第2个数和第3个数，将小数放前，大数放后，如此继续，直至比较最后两个数，将小数放前，大数放后。重复第一趟步骤，直至全部排序完成。

举例说明：要排序数组：int[] arr={6,3,8,2,9,1};

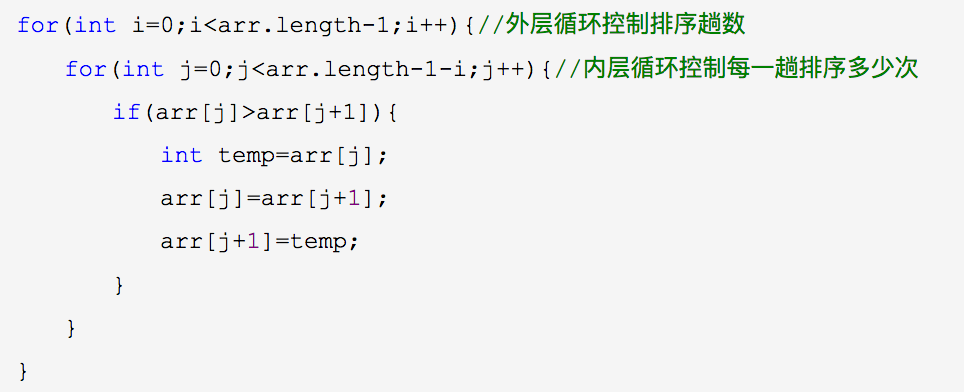






最终结果：1 2 3 6 8 9

由此可见：N个数字要排序完成，总共进行N-1趟排序，每i趟的排序次数为(N-i)次，所以可以用双重循环语句，外层控制循环多少趟，内层控制每一趟的循环次数，即



1. **aa**