**线性表查找**

**顺序查找**

非常简单

示例1∶在成绩中查询分数是100的第一个分数

示例2∶在学生中查询分数是100的第一个学生

存储结构可以是顺序表，也可以是链表

逐个比较查询，如果找到，返回数据或者索引，如果到最后也没有找到，返回null

可以是在顺序表中，也可以是在链表中

在各个节点查找概率相同情况下，默认查询长度为一半长度，

所以时间复杂度是T（n）=O（n）

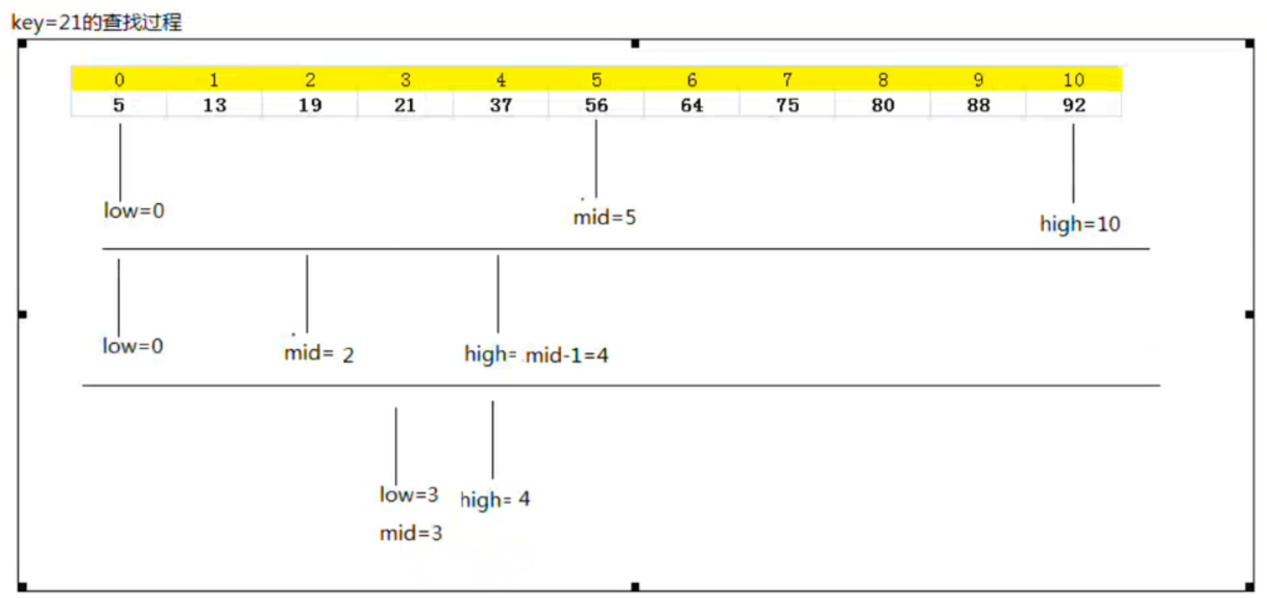
**折半查找**

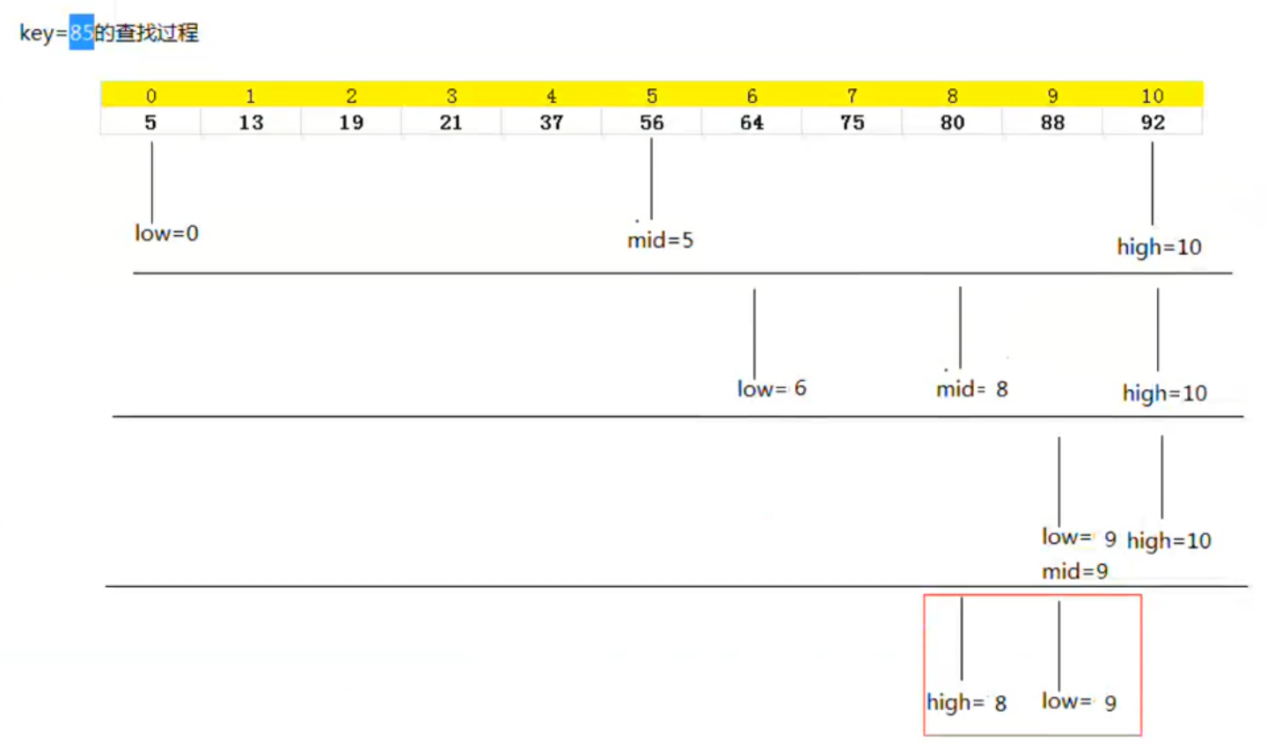
折半查找又称为二分查找，这种查找方法需要待查的查找表满足两个条件∶

首先，查找表必须使用顺序存储结构;

其次，查找表必须按关键字大小有序排列

使用非递归实现折半查找：



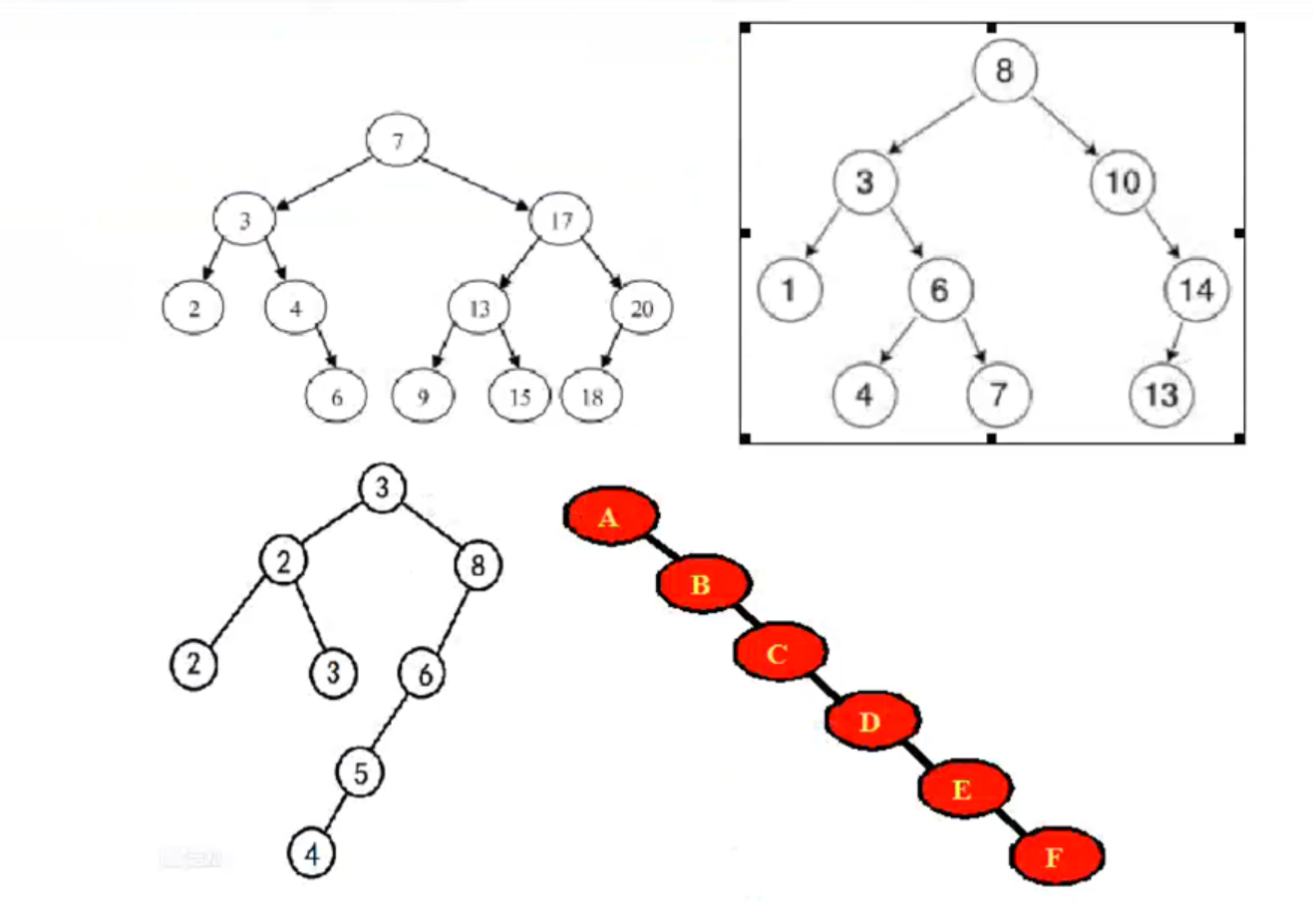


**二叉查找/搜索/排序树BST（binary search/sort tree）**

或者是一棵空树;

或者是具有下列性质的二叉树∶

1. 若它的左子树不空，则左子树上所有结点的值均小于它的根节点的值
2. 若它的右子树上所有结点的值均大于它的根节点的值
3. 它的左、右子树也分别为二叉排序树



注意∶对二叉查找树进行中序遍历，得到有序集合

**平衡二叉树（Self-balancing binary search tree）**自平衡二叉查找树 又被称为AVL树（有别于AVL算法）

它是一棵空树

或它的左右两个子树的高度差（平衡因子）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

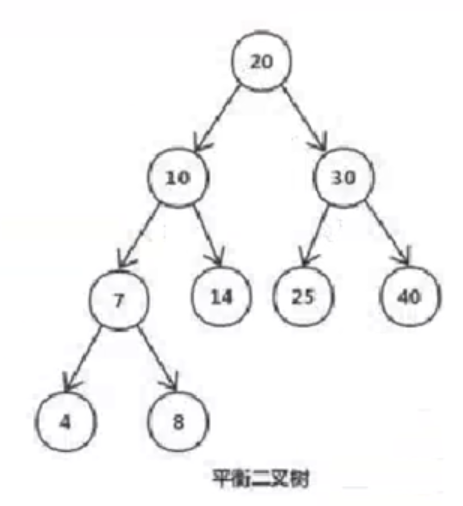
同时，平衡二叉树必定是二叉搜索树，**反之则不一定**

平衡因子（平衡度）∶结点的平衡因子是结点的左子树的高度减去右子树的高度（或反之定义）

平衡二叉树∶每个结点的平衡因子都为1、-1、0的二叉排序树，或者说每个结点的左右子树的高度最多差1的二叉排序树

平衡二叉树的目的是为了减少二叉查找树层次，提高查找速度

平衡二叉树的常用实现方法有AVL、红黑树、替罪羊树、Treap、伸展树等



**红黑树**

1. BTree，全称是Red-Black Tree，又称为"红黑树"，它是一种平衡二叉树。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）

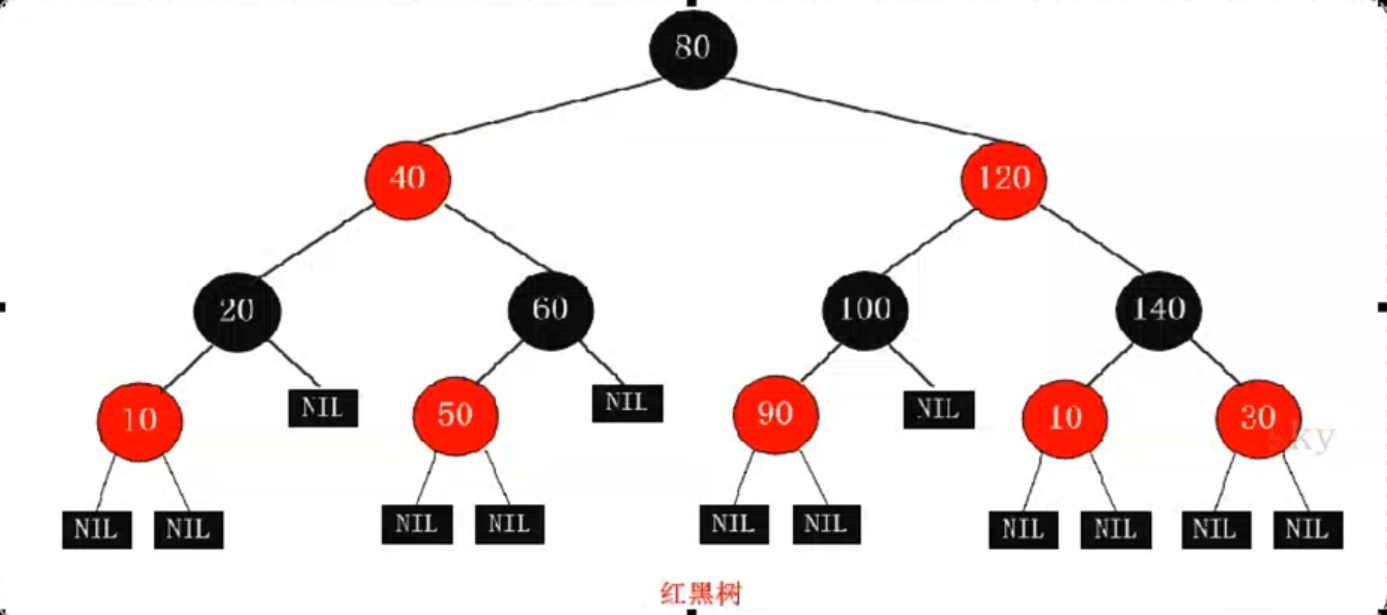
红黑树的特性∶

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

注意∶

（01）特性（3）中的叶子节点，是值为空（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两倍。因而，红黑树是相对是接近平衡的二叉树



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（log2N），效率非常之高

它虽然是复杂的，但它的最坏情况运行时间也是非常良好的，并且在实践中是高效的∶它可以在O（log2 n）时间内做查找，插入和删除，这里的n 是树中元素的数目。

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

**B树（balanced tree）**

你觉得平衡树，和二叉平衡树有什么差别呢?

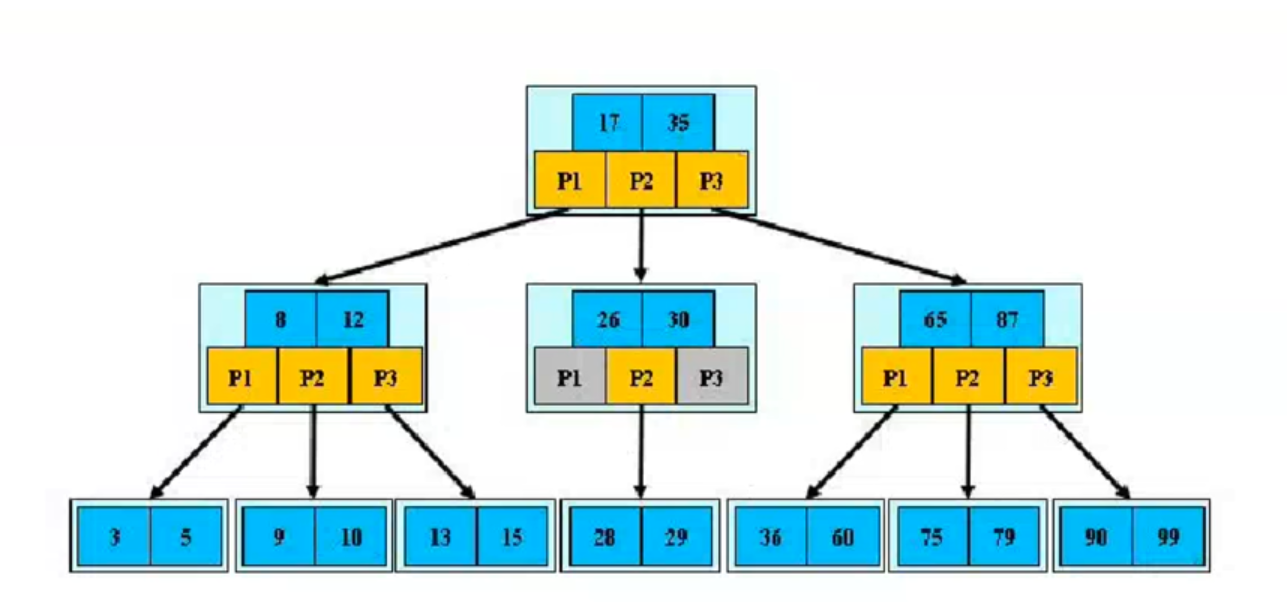
B树与二叉平衡树相比，是多叉的 二叉平衡树肯定是B树，反之不一定

B树可以降低树的深度，提高查找效率

B树应文件系统的要求而发展起来的，大量数据存放在外存中，通常存放在硬盘中

由于是海量数据，不可能一次调入内存。因此，要多次访问外存。但硬盘的驱动受机械运动的制约，速度慢。

所以，主要矛盾变为减少访外存次数在1972年由R.Bayer和E Macreight 提出用B树作为索引组织文件。提高访问速度、减少时间

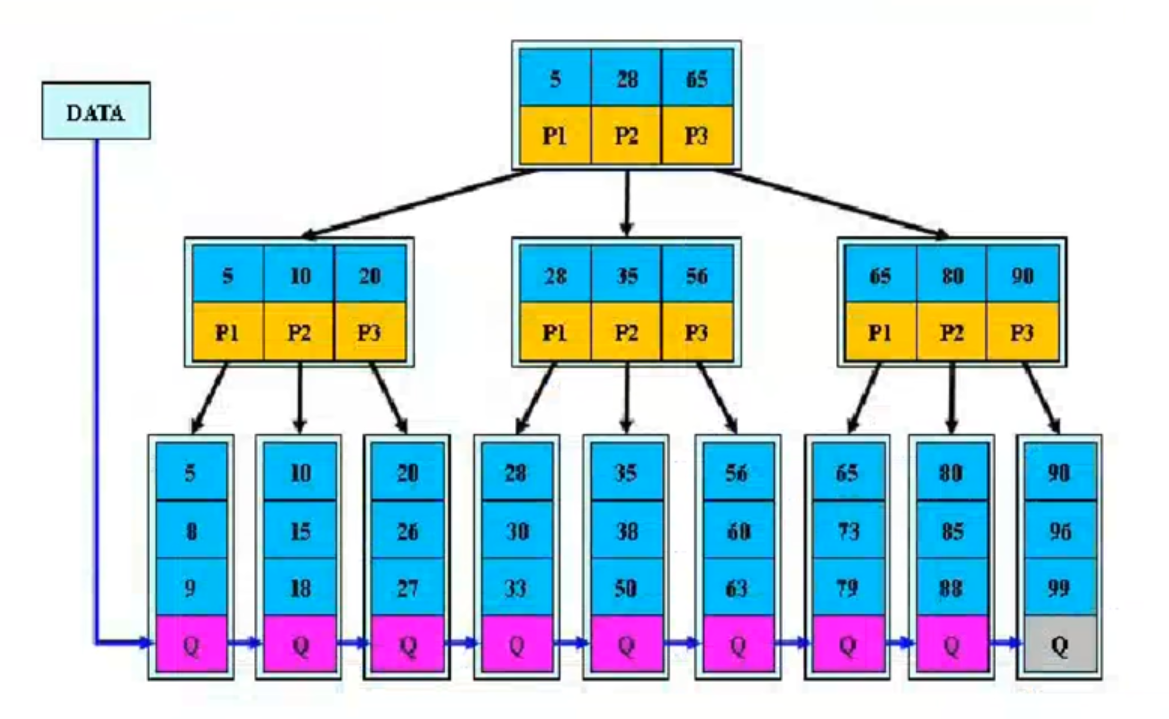


**B+树**

在B-树基础上，为叶子结点增加链表指针，所有关键字都在叶子结点中出现，非叶子结点作为叶子结点的索引;

B+树总是到叶子结点才命中

**数据库的索引的默认数据结构就是采用B+树**



**引入哈希表**

前面查找方法共同特点∶通过将关键字值与给定值比较，来确定位置。效率取决比较次数

理想的方法是∶不需要比较，根据给定值能直接定位记录的存储位置

这样，需要在记录的存储位置与该记录的关键字之间建立一种确定的对应关系，使每个记录的关键字与一个存储位置相对应

**1.哈希表的结构和特点**

**hashtable 也叫散列表**

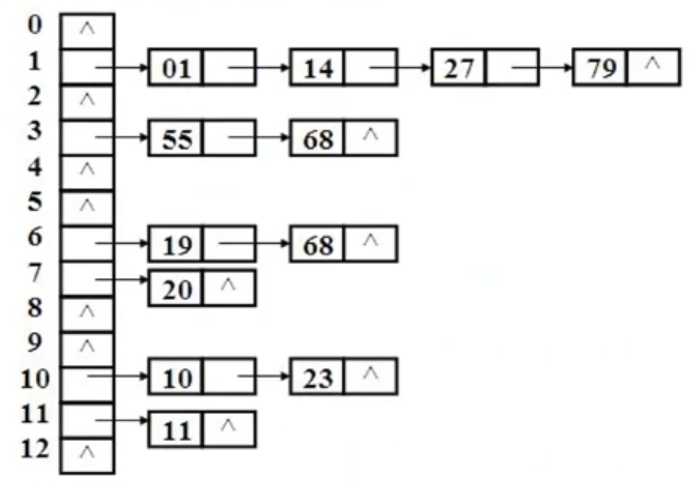
特点∶快 很快 神奇的快

结构∶结构有多种

最流行、最容易理解∶顺序表+链表

主结构∶顺序表

每个顺序表的节点再单独引出一个链表



**2.哈希表是如何添加数据的**

1.计算哈希码（调用hashCode()，结果是一个int值，整数的哈希码取自身即可）

2.计算在哈希表中的存储位置 **y= k（x）=x%11**

x:哈希码 k（x）函数 y∶在哈希表中的存储位置 11：顺序表的长度

3.存入哈希表

情况1∶一次添加成功

情况2∶多次添加成功（出现了冲突，调用equals()和对应链表的元素进行比较，比较到最后，结果都是false，创建新节点，存储数据，并加入链表末尾）

情况3∶不添加（出现了冲突，调用equals()和对应链表的元素进行比较，

经过一次或者多次比较后，结果是true，表明重复，不添加）

结论1∶哈希表添加数据快（3步即可，不考虑冲突）

结论2∶唯一

结论3∶无序

**3.哈希表是如何查询数据的**

和添加数据的过程是相同的

情况1∶一次找到 23 86 76

情况2∶多次找到67 56 78

情况3∶找不到 100 200

结论1∶哈希表查询数据快

结论2∶哈希表删除数据快

结论3∶哈希表更新数据快（如果更新后影响到哈希码值，就比较繁琐了，比如要删除再添加了）

**4.hashCode()和equals()到底有什么神奇的作用**

hashCode()∶计算哈希码，是一个整数，根据哈希码可以计算出数据在哈希表中的存储位置

Equals()∶添加时出现了冲突，需要通过equals进行比较，判断是否相同，查询时也需要使用equals进行比较，判断是否相同

**5.各种类型数据的哈希码应该如何获取**

1.int 取自身看Integer的源码

2.double 3.14 3.15 3.145 6.567 9.87取整不可以看Double的源码

3.String java oracle j+a+v+a 将各个字符的编码值相加不可以

Abc cba bac a:97 b:98 c:99

abc 1\*97+2\*98+3\*99

cba 1\*99+2\*98+3\*97

4.Student属性的哈希码，各个进行某些相加相乘的运算

int id

String name

Int age

double score;

**6.如何减少冲突**

**此次采用的链地址哈希表结构很大程度上减少冲突(equals方法)，但是会造成运行损耗**

1）哈希表的长度和表中的记录数的比例--装填因子∶

如果Hash表的空间远远大于最后实际存储的记录个数，则造成了很大的空间浪费，

如果选取小了的话，则容易造成冲突

在实际情况中，一般需要根据最终记录存储个数和关键字的分布特点来确定Hash表的大小

还有一种情况时可能事先不知道最终需要存储的记录个数，则需要动态维护Hash表的容量，此时可能需要重新计算Hash地址。

装填因子=表中的记录数/哈希表的长度，

如果**装填因子越小，表明表中还有很多的空单元，则添加发生冲突的可能性越小**;

而**装填因子越大，则发生冲突的可能性就越大，在查找时所耗费的时间就越多**。

有相关文献证明当装填因子在0.5左右的时候，Hash的性能能够达到最优。

因此，一般情况下，装填因子取经验值0.5。

(2)哈希函数选择：直接定址法，除留取余法（y=x%11），平方取中法，折叠法