**ArrayList和LinkedList的区别：**

ArrayList是一种顺序存储的线性表，底层采用数组来保存元素，向其中添加、删除元素时，需要对数组进行“整体搬家”，因此性能非常差。

LinkedList是一种链式存储的线性表，其本质上是一个双向链表，它不仅实现了List接口，还实现了Deque接口。对于删除和添加元素，性能非常好。

但对于读取指定索引处的元素，ArrayList性能更好。所以，大部分情况下ArrayList性能要比LinkedList好，如果要经常使用删除、添加操作，应考虑使用LinkedList。

**TreeSet和TreeMap的区别与联系：**

TreeSet里面的绝大部分方法都是直接调用TreeMap方法来实现的。

相同点：

1. TreeMap和TreeSet都是有序的集合，也就是说它们存储的值都是排好序的；
2. TreeMap和TreeSet都是非同步集合，因此它们不能在多线程之间共享；
3. 运行速度都比Hash集合慢，它们内部对元素操作的时间复杂度为O(logN)，而HashMap和HashSet则为O(1)。

不同点：

1. 最主要的区别就是TreeSet实现了Set接口，而TreeMap实现了Map接口；
2. TreeSet只存储一个对象，而TreeMap存储两个对象Key和Value（仅仅Key对象有序）；
3. TreeSet中不能有重复的对象，而TreeMap中可以存在。

**HashSet、HashMap和HashTable的区别与联系：**

区别：

1. HashSet继承自AbstractSet类，实现了Set接口，HashMap继承自AbstractMap类，实现了Map接口，HashTable继承自Dictionary类，实现了Map接口；
2. HashTable中的方法是Synchronize（线程安全）的，而HashMap中的方法是非Synchronize（线程不安全）的，因为HashSet中的方法都是HashMap实现的，所以也是线程不安全的；
3. HashMap是Key-Value的形式存储对象，Key不能重复，但可以为null，但这样的键只能有一个；而HashSet只存储一个对象并且不能有重复的对象，可以有一个null值；HashTable不允许null值（键与值均不能为空）；
4. HashMap和HashSet初始大小为16，HashTable初始大小为11；
5. 内存扩容方式：HashTable采用的是2\*old+1，而HashMap和HashSet都是2\*old；
6. 哈希值的计算方法不同，Hashtable直接使用的是对象的hashCode,而HashMap则是在对象的hashCode的基础上还进行了一些变化（先计算hashcode,然后再求其在哈希表的相应位置）。

联系：

1. HashSet底层方法是用HashMap来实现的，所以很多性质都相同：都是线程不安全的，Key都不能重复，初始大小都为16，扩容方式都是2\*old，hashCode的计算方式也相同；
2. HashTable和HashMap都是以Key-Value的形式存储对象；
3. HashTable和HashMap都实现了Map接口。

**ArrayList、LinkedList、Vector的区别与联系：**

区别：

1. ArrayList和LinkedList是非Synchronize（线程不安全）的而Vector是线程安全的，所以如果多线程情况下可以使用Vector；
2. ArrayList和Vector采用数组方式存储数据，所以数据的插入较慢，查询较快；LinkedList采用双向链表存储数据，查询时需要按序号索引向前或向后遍历，速度较慢，而插入数据只需记录本项前后项即可，插入数据较快；
3. ArrayList的默认初始容量为10（用户可自己定义初始容量），增长为原来的1.5倍（java1.7之前的为oldCapacity\*1.5+1，java1.7之后为1.5倍）；

LinkedList默认初始容量为1，每次增长的容量为1；

Vector默认初始容量为10，每次增长为原来的2倍；

联系：

1. 都是实现了List接口；
2. ArrayList和LinkedList是线程不安全的；
3. ArrayList和Vector都是采用数组方式存储数据；

**Java对对象的引用**

强引用，软引用，弱引用，虚引用

**强引用**：程序创建一个对象，并把这个对象赋给一个引用变量，这个引用变量就是强引用。被强引用所引用的java对象不会被垃圾回收机制回收，即使系统内存非常紧张，或者这些java对象以后也不会用到，JVM也不会回收被强引用所引用的对象。

**软引用**：通过SoftReference类实现，系统内存足够时，软引用的对象不会被系统回收，程序也可以使用该对象，当系统内存不足时，系统就会回收它。

**弱引用**：通过WeakReference类实现，对于只有弱引用的对象，当系统垃圾回收机制运行时，不管系统内存是否足够，都会回收该对象所占的内存。

**虚引用**：不能单独使用，主要作用是跟踪对象被垃圾回收的状态，程序通过检查与虚引用关联的引用队列中是否已经包含指定的虚引用，从而了解虚引用所引用的对象是否即将被回收。

**Java中Scanner类和BufferReader类之间的区别**

BufferedReader是支持同步的，而Scanner不支持。如果我们处理多线程程序，BufferedReader应当使用。

BufferedReader相对于Scanner有足够大的缓冲区内存。

Scanner有很少的缓冲区(1KB字符缓冲)相对于BufferedReader(8KB字节缓冲)，但是这是绰绰有余的。

BufferedReader相对于Scanner来说要快一点，因为Scanner对输入数据进行类解析，而BufferedReader只是简单地读取字符序列。

**Java内存泄漏的原因**

1. 静态集合类引起内存泄露
2. 当集合里面的对象属性被修改后，再调用remove（）方法时不起作用。
3. 释放对象的时候没有去删除监听器，从而增加了内存泄漏的机会
4. 各种连接，比如数据库连接（dataSourse.getConnection()），网络连接(socket)和io连接，除非其显式的调用了其close（）方法将其连接关闭，否则是不会自动被GC 回收的。
5. 内部类和外部模块等的引用

内部类的引用是比较容易遗忘的一种，而且一旦没释放可能导致一系列的后继类对象没有释放。此外程序员还要小心外部模块不经意的引用，例如程序员A 负责A 模块，调用了B 模块的一个方法如：

public void registerMsg(Object b);

这种调用就要非常小心了，传入了一个对象，很可能模块B就保持了对该对象的引用，这时候就需要注意模块B 是否提供相应的操作去除引用。

1. 不正确使用单例模式。不正确使用单例模式是引起内存泄露的一个常见问题，单例对象在被初始化后将在JVM的整个生命周期中存在（以静态变量的方式），如果单例对象持有外部对象的引用，那么这个外部对象将不能被jvm正常回收，导致内存泄露

**一些常见的垃圾回收器**

**串行回收器：**对于young代采用串行复制算法，对于old代采用串行标记压缩算法

对于young代，系统将Eden和Survivor1区中的活动对象直接复制到初始为空的Survivor2区中，如果有些对象占用空间很大，就把它复制到old代中。复制完后Eden和Survivor1区中就剩下不可达的对象了，然后系统直接把Eden区和Survivor1的所有内存直接回收，此时Survivor2区保存活动对象，下次回收的时候Survivor2区就变成了Survivor1区。

对于old代，回收阶段分为三个阶段：标记可达对象，清除，压缩。在标记可达对象阶段，回收器先识别哪些对象是可达的，然后在清除阶段会回收不可达（未标记）的对象所占的内存，在压缩阶段，把标记的活动对象往old代的前端移动，而在尾部保留一块连续的空间，以便下次为新对象分配内存。

**并行回收器：**回收算法与串行回收器相似，只是增加了多CPU并行的能力。

**并行压缩回收器：**

并行压缩回收器对于young代采用与并行回收器相同的算法。

对于old代回收，分为三个阶段：mark，summary，compact。系统先将Old代分为几个固定大小的区域，在mark阶段，多个垃圾回收线程会标记Old代中的可达对象，然后在summary阶段从old代的左边区域开始检查每个区域的密度，当检测到某个区域的可达对象密度较小时，垃圾回收器会判定该区域以及该区域右边的区域都应该进行回收，该区域左边的区域被标记为密集区域。该区域和其右边的区域都会被进行压缩并回收空间（summary阶段目前还是串行操作）。最后是compact阶段，回收器利用summary阶段生成的数据识别出有哪些区域是需要装填的，多个垃圾回收线程可以并行的把数据复制到这些区域中。经过这个过程后，Old代的一端密集的存在大量活动的对象，另一端则存在大块的空闲块。

**内存管理小技巧**

1. 尽量使用直接量

当需要使用字符串，还有Byte、Short、Integer、Long、Float、Double、Boolean、Character包装类的实例时，程序不应该使用new 的方式来创建对象，而应该直接采用直接量来创建它们。比如：应该采用String str = “hello”;这样的方式来创建字符串对象。如果采用

String str = new String(“hello”); 来创建，程序会创建一个缓存在字符串缓存池中的“hello”字符串，除此之外，str所引用的String对下给你底层还包含一个char[]数组，这个数组里依次存放了h、e、l、l、o等字符串。

1. 使用StringBuilder和StringBuffer进行字符串连接

如果程序使用多个String对象进行字符串连接运算，在运行时将生成大量临时字符串，这些字符串会保存在内存中从而导致程序性能下降。

1. 尽早释放无用对象的引用
2. 尽量少使用静态变量

因为静态变量在类加载的时候分配内存，在类被卸载（进程结束）的时候销毁，只要静态变量没有被销毁也没有置null，其对象一直被保持引用，也即引用计数不可能是0，因此不会被垃圾回收

1. 避免在经常调用的方法、循环中创建java 对象

因为其中的对象生存时间并不长，接下来系统又要回收它们的内存，这种不断分配回收的操作中，程序的性能受到巨大的影响。

1. 缓存经常使用的对象

常见的缓存是数据连接池，还有HashMap、直接使用某些开源的缓存项目。

1. 尽量不要使用finalize 方法

导致垃圾回收器的负担更大，导致程序运行效率更差。

1. 考虑使用SoftReference

当内存足够时，它的功能等同于普通引用，当内存不足时，它会牺牲自己，释放软引用所引用的对象。

**二叉树遍历**

非递归方式实现**先序遍历**：

1. 首先申请一个新的栈，记为stack；
2. 然后将头节点head压入stack中；
3. 每次从stack中弹出栈顶节点，记为cur，然后打印cur节点的值。如果cur右孩子不为空的话，将cur的右孩子先压入stack中，最后如果cur的左孩子不为空的话，将cur的左孩子压入stack中。
4. 不断重复步骤3，知道stack为空，全部过程结束。

代码实现：

**public** **static** **void** preOrderStack\_1(Node root){

**if**(root==**null**)**return**;

Stack<Node> s=**new** Stack<Node>();

s.push(root);

**while**(!s.isEmpty()){

Node cur=s.pop();

System.***out***.println(cur.value);

**if**(cur.right!=**null**) s.push(cur.right);

**if**(cur.left!=**null**) s.push(cur.left);

}

}

非递归方式实现**中序遍历**：

1. 申请一个新的栈，记为stack，申请一个变量cur，初始时令cur等于头节点；
2. 先把cur节点压入栈中，对以cur节点为头的整棵子树来说，依次把整棵树的左边界压入栈中，即不断令cur=cur.left，然后重复步骤2；
3. 不断重复步骤2，直到发现cur为空，此时从stack中弹出一个节点，记为node。打印node的值，并让cur=node.right，然后继续重复步骤2；
4. 当stack为空并且cur为空时，整个步骤结束。

代码实现：

非递归方式实现**后序遍历**：

**public** **static** **void** inOrderStack(Node root){

**if**(root==**null**)**return**;

Stack<Node> s=**new** Stack<Node>();

Node cur = root;

**while**(!s.isEmpty() && cur!=**null**){

**while**(cur!=**null**){

s.push(cur);

cur = cur.left;

}

Node node = s.pop();

System.***out***.println(node.value);

cur = node.right;

}

}

方法一：使用两个栈实现

1. 申请一个栈，记为s1，然后将头节点压入s1；
2. 从s1中弹出的节点记为cur，然后先把cur的左孩子压入s1中，然后把cur1的右孩子压入s1中；
3. 在整个过程中，每一个从s1中弹出的节点都放进第二个栈s2中；
4. 不断重复步骤2和步骤3，直到s1为空，过程停止；
5. 从s2中依次弹出节点并打印，打印的顺序就是后序遍历的顺序了。

代码实现

**public** **static** **void** postOderStack(Node root) {

**if**(root == **null**){

**return**;

}

Stack<Node> s1 = **new** Stack<Node>();

Stack<Node> s2 = **new** Stack<Node>();

s1.push(root);

**while** (!s1.isEmpty()) {

Node cur = s1.pop();

s1.push(cur.left);

s1.push(cur.right);

s2.push(cur);

Node node = s2.pop();

System.***out***.println(node.value);

}

}

方法二：使用一个栈实现

1. 申请一个栈，记为stack，将头节点压入stack，同时设置两个变量h和c。在整个流程中，h代表最近一次弹出并打印的节点，c代表当前stack的栈顶节点，初始时令h为头节点，c为null；
2. 每次令c等于当前stack的栈顶节点，但是不从stack中弹出节点，此时分为一下三种情况：
   1. 如果c的左孩子不为空，并且h不等于c的左孩子，也不等于c的右孩子，则把c的左孩子压入stack中；
   2. 如果情况1不成立，并且c的右孩子不为空，并且h不等于c 的右孩子，则把c的右孩子压入stack中；
   3. 如果情况1和情况2都不成立，那么从stack中弹出c并打印，然后令h等于c
3. 一直重复步骤2，直到stack为空，过程停止。

代码实现：

**public** **static** **void** postOrderStack2(Node root){

**if**(root == **null**){

**return**;

}

Stack<Node> s = **new** Stack<Node>();

Node h = root;

Node c = **null**;

**while**(!s.isEmpty()){

c = s.peek();

**if**(c.left != **null** && h != c.left && h != c.right){

s.push(c.left);

}

**else** **if**(c.right != **null** && h != c.right){

s.push(c.right);

}

**else**{

c = s.pop();

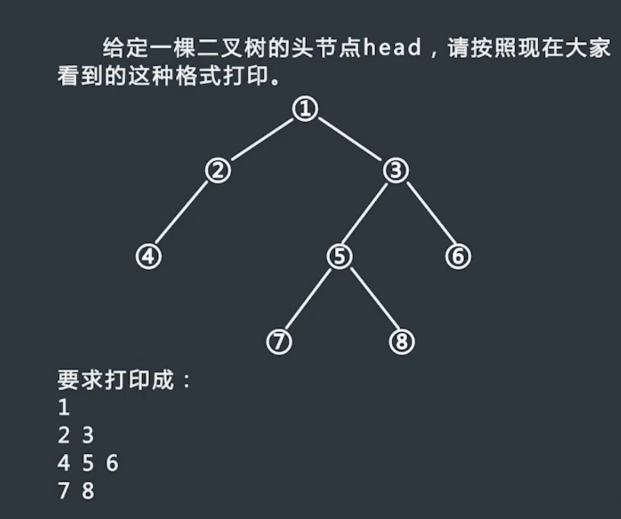
System.***out***.println(c.value);

}

}

}

**二叉树打印**

针对二叉树的宽度优先遍历，尝试用队列结构，面试中，常对换行有所要求，需要打印出行号。

last：表示正在打印的当前行的最右节点

nlast：表示下一行的最右节点

1. 让last等于节点1，然后把节点1放入queue，从queue中弹出节点1并打印；
2. 弹出队列中的节点，如果当前结点有左节点，将左节点压入队列中，如果当前节点有右节点，将右节点压入队列中，当前打印节点为当前行最右节点时换行。
3. 重复2步骤直到队列为空。

代码实现：

**public** **static** **void** print(BinaryTree root) {

Queue<BinaryTree> queue = **new** LinkedList<BinaryTree>();

BinaryTree last;

BinaryTree nlast = **null**;

last = root;

queue.add(root);

**while** (queue.size() > 0) {

BinaryTree nowTree = queue.poll();

**if** (nowTree.left != **null**) {

queue.add(nowTree.left);

nlast = nowTree.left;

}

**if** (nowTree.right != **null**) {

queue.add(nowTree.right);

nlast = nowTree.right;

}

System.***out***.print(" " + nowTree.value);

**if** (last.equals(nowTree)) {

System.***out***.println();

last = nlast;

}

}

}

**二叉树序列化和反序列化**

二叉树🡪字符串（序列化）

字符串🡪二叉树（反序列化）

序列化的方式：

1. 根据先序遍历序列化
2. 根据中序遍历序列化
3. 根据后序遍历序列化
4. 按层序列化

先序遍历对二叉树进行序列化

1. 假设序列化结果为str，初始时str为空字符串；
2. 先序遍历二叉树时如果遇到空节点，在str末尾加上“#！”；
3. 如果遇到不为空的节点，假设节点值为3，就在str的末尾加上“3！”

用一个特殊字符表示一个二叉树节点值的结束的意义；

序列化和反序列化的代码实现：

**static** **int** *index* = -1;

**static** String Serialize(TreeNode root) {

StringBuffer str = **new** StringBuffer();

**if**(root==**null**){

str.append("#!");

**return** str.toString();

}

str.append(root.val + "!");

str.append(*Serialize*(root.left));

str.append(*Serialize*(root.right));

**return** str.toString();

}

**static** TreeNode Deserialize(String str) {

*index*++;

**if**(str == **null**){

**return** **null**;

}

String[] de = str.split("!");

TreeNode node = **null**;

**if**(!de[*index*].equals("#")){

node = **new** TreeNode(Integer.*valueOf*(de[*index*]));

node.left = *Deserialize*(str);

node.right = *Deserialize*(str);

}

**return** node;

}

**排序**

内部排序分类：

选择排序（直接选择排序，堆排序）、交换排序（冒泡排序，快速排序）、插入排序（直接插入排序，折半插入排序，Shell排序）、归并排序、桶排序、基数排序



1. **直接选择排序**

算法思想：

第1趟比较：程序将记录定位在第一个数据上，拿第1个数据依次和它后面的每个数据进行比较，如果第1个数据大于后面的某个数据，交换他们……以此类推。经过第1趟比较，这组数据中最小的数据被选出，它被排在第1位。

第2趟比较：程序将记录定位在第2个数据上，拿第2个数据依次和它后面的每个数据进行比较，如果第2个数据大于后面的某个数据，交换他们……以此类推。经过第2趟比较，这组数据中第2小的数据被选出，它被排在第2位。……

按次规则一共进行n-1趟比较，这组数据中第n-1小（第2大）的数据被选出，被排在第n-1位（倒数第1位）；剩下的就是最大的数据，排在最后。

代码实现：

**public** **static** **void** selectSort(**int**[] data) {

System.***out***.println("开始排序");

**int** arrayLength = data.length;

**for**(**int** i = 0; i < arrayLength - 1; i++){

**for**(**int** j = i+1; j < arrayLength; j++){

**if**(data[i] > data[j]){

**int** tmp = data[i];

data[i] = data[j];

data[j] = tmp;

}

}

System.***out***.println(Arrays.*toString*(data));}}

1. **堆排序**

小根堆：假设有n个数据元素的序列k0，k1，···，kn-1，当且仅当满足如下关系时，可以将这组数据成为小顶堆（小根堆）。

ki<=k2i-1 且ki<=k2i+2 （其中i=0，2，···，(n-1)/2）

或者满足如下关系时，可以将这组数据称为大根堆（大顶堆）。

ki>=k2i-1 且ki>=k2i+2 （其中i=0，2，···，(n-1)/2）

算法思想：堆排序的关键在于建堆，从最后一个非叶子结点开始，比较该节点和它两个子节点的值，如果某个子节点的值大于父节点的值，把父节点和交大的子节点交换，向前逐步调整直到根节点，保证每个父节点的值都大于等于其左右子节点的值，建堆完成。

1. **冒泡排序**

算法思想：

第一趟：依次比较0和1、1和2、2和3……n-2和n-1索引的元素，如果发现第一个数据大于后一个数据，交换它们。经过第1趟，最大的元素排到了最后。

第二趟：一次比较0和1、1和2、2和3……n-3和n-2索引的元素如果发现第一个数据大于后一个数据，交换它们。经过第2趟，第2大的元素排到了最后。

……

第n-1趟：依次比较0和1元素，如果发现第一个数据大于后一个数据，交换它们。经过第n-1趟，第2小（第n-1大）的元素排到了第2位。

**public** **static** **void** bubbleSort(**int**[] data){

System.***out***.println("开始排序");

**int** arrayLength = data.length;

**for**(**int** i = 0; i < arrayLength-1; i++){

**boolean** flag = **false**;

**for**(**int** j = 0; j < arrayLength-1-i; j++){

**if**(data[j] > data[j+1]){

**int** tmp = data[j+1];

data[j+1] = data[j];

data[j] = tmp;

flag = **true**;

}

}

System.***out***.println(Arrays.*toString*(data));

**if**(!flag){

**break**;

}

}

}

1. **快速排序**

算法思想：

从待排序的数据序列中任取一个数据（如第一个数据）作为分界值，所有比它小的数据元素一律放到左边，所有比它大的数据元素一律放到右边。这样一趟下来，该序列形成左右两个子序列，左边序列中数据元素的值都比分界值小，右边序列中数据元素都比分界值大。接下来对左右两个子序列进行递归，对两个子序列重新选择中心元素并依此规则调整，直到每个子表的元素只剩下一个，排序完成。

代码实现：

**private** **static** **void** subSort(**int**[] data, **int** start, **int** end){

**int** temp;

**if**(start < end){

**int** base = data[start];

**int** i = start;

**int** j = end + 1;

**while**(**true**){

**while**(i < end && data[++i] <= base);

**while**(j > start && data[--j] >= base);

**if**(i < j){

temp = data[i];

data[i] = data[j];

data[j] = temp;

System.***out***.println(Arrays.*toString*(data));

}

**else**{

**break**;

}

}

temp = data[start];

data[start] = data[j];

data[j] = temp;

*subSort*(data, start, j - 1);

*subSort*(data, j+1, end);

}

}

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

**int**[] data = {9,-16,21,23,-30,-49,21,30,13};

System.***out***.println("排序之前：\n"+ Arrays.*toString*(data));

System.***out***.println("开始排序：");

*subSort*(data, 0, data.length - 1);

System.***out***.println("排序之后：\n" + Arrays.*toString*(data));

}

1. **插入排序**

算法思想：

第1趟：将第2个元素与第1个元素进行比较，如果小于第一个元素，就交换；

第n趟：将第n+1个元素与前n个元素分别进行比较，当遇到比它小的元素时，则把它插入到这个元素后面。

代码实现：

**public** **static** **void** insertSort(**int**[] data){

System.***out***.println("开始排序：");

**for**(**int** i = 1; i < data.length; i++){

**int** tmp = data[i];

**if**(data[i] < data[i-1]){

**int** j = i-1;

**for**( ; j >= 0 && data[j] > tmp; j--){

data[j+1] = data[j];

}

data[j+1] = tmp;

}

System.***out***.println(Arrays.*toString*(data));

}

}

1. **折半插入排序**

算法思想：

折半插入排序利用前面i个元素有序的特点，先计算0 ～ i-1个元素的中间点，也就是用i索引处的元素和(0+i-1)/2索引处的元素进行比较，如果i索引处的元素大，就直接在(0+i-1)/1～i-1半个范围内搜索；反之，就在0～(0+i-1)/2范围内搜索；在半个范围内搜索时，再进行折半搜索，直到确定第i个元素的插入位置；然后将该位置以后的元素整体后移一位，然后将第i个元素放入该位置。

代码实现：

**public** **static** **void** binaryInsertSort(**int**[] data){

System.***out***.println("开始排序：");

**int** arrayLength = data.length;

**for**(**int** i = 1; i < arrayLength; i++){

**int** tmp = data[i]; **int** low = 0; **int** high = i-1;

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

**int** mid = (low + high)/2;

**if**(tmp <= data[mid]){

high = mid - 1;

}**else**{ low = mid + 1; }

}

**for**(**int** j = i; j > low; j--){

data[j] = data[j-1];

}

data[low] = tmp;

System.***out***.println(Arrays.*toString*(data));

}

}

1. **Shell排序（希尔排序）**

算法思想：

初期选用大跨步（增量较大）间隔比较，使记录跳跃式接近它的排序位置；然后增量缩小；最后增量为1，这样记录移动次数大大减少，提高了排序效率。①先取一个正整数 h(h<n)，把全部记录分成h个组，所有距离为h的倍数的记录看成一组，然后在各组内进行插入排序；②然后取h2(h2 < h )。③重复上述分组和排序操作；直到取 hi=1(i>=1) ，即所有记录成为一个组为止。一般选h约为 n/2，h2为d 1 /2，h3为 d 2 /2，…，hi =1 。

代码实现：

**public** **static** **void** shellSort(**int**[] data){

System.***out***.println("开始排序：");

**int** arrayLength = data.length;

**int** h = 1;

**while**(h <= arrayLength/3){

h = h \* 3+1;

}

**while**(h > 0){

System.***out***.println("====h的值：" + h + "====");

**for**(**int** i = h; i < arrayLength; i++){

**int** tmp = data[i];

**if**(data[i] < data[i-h]){

**int** j = i-h;

**for**( ; j >= 0 && data[j] > tmp; j-=h){

data[j+h] = data[j];

}

data[j + h] = tmp;

}

System.***out***.println(Arrays.*toString*(data));

}

h = (h - 1)/3;

}

}

## 数据库复习

**内连接，外连接，左连接，右连接**

INNER JOIN 两边表同时有对应的数据，即任何一边缺失数据就不显示。

LEFT JOIN 会读取左边数据表的全部数据，即便右边表无对应数据。

RIGHT JOIN 会读取右边数据表的全部数据，即便左边表无对应数据。