数据结构与算法

1. 数据结构概述

定义：解决如何把现实中大量而复杂的问题以特定的数据类型和特定的存储结构保存到主存储器（内存）中，以及在此基础上为实现某个功能（比如查找某个元素、删除某个元素、插入、对所有元素进行排序）而执行的相应操作，这个相应的操作也叫做算法。

数据结构就是解决数据的存储问题，要解决该问题就要使用到算法。

数据结构=个体+个体的关系；

算法 = 对存储数据的操作；

算法的定义：算法就是解题的方法和步骤，在计算机中就是数据怎么存和怎么找；

衡量算法的标准：

1. 时间复杂度，某代码随问题规模的增加而得到的大概执行次、数的结果，而非执行的时间；
2. 空间复杂度，算法执行过程中所占用的最大内存；
3. 难易程度；
4. 健壮性；

数据结构的重要性：

数据结构是软件编程中最核心的课程

程序 = 数据的存储 + 数据的操作 + 可以被计算机执行的语言；

1. malloc()函数动态分配内存

int \* pArr = (int \*)malloc(sizeof(int) \* len);

free(pArr);//释放内存，C语言必须由程序员手动释放动态内存。

模块一：

1. 线性结构：把所有的节点用一根线穿起来：数组（连续储存），链表（随机存储）；线性结构的应用——栈、队列；
2. 连续存储与离散存储的优缺点：

连续存储（数组）：优点是占用空间较小，存取速度快，缺点是，插入删除元素速度较慢，空间受数组的长度限制；

离散存储（链表）：优点是，空间没有限制，插入删除元素的速度快，长度由随节点的个数变换；缺点是：存取的速度慢，占用内存较多（与数组相比）

1. 链表：离散存储

定义：多个节点离散分配，彼此通过指针相连，每个节点只有一个前驱节点与后续节点，首节点没有前驱节点，尾节点没有后续节点

专业术语：

头结点：只存放首节点的地址的，他的作用是方便对链表的操作，头结点的数据类型和链表一致；

首节点：第一个有效的节点（存放着有效数据的第一个节点）；

尾节点：最后一个有效节点；

头指针：指向头结点的指针变量；

尾指针：指向尾节点的指针变量；

确定一个链表所需参数：只需要头指针，就可以对该链表进行操作；

链表的分类：

单链表：每一个结点除首结点和尾结点外，都有一个直接前驱和一个直接后继，这样的构成一条线性关系的存储结构就是单链表；

单向循环链表：将单链表的尾结点指向它首结点而形成的一个闭合的链表；

双链表：每一个节点含有两个指针域；

循环链表：能通过任何一个节点找到其他的节点，即尾指针非空，一般存放首节点的地址；

非循环链表：

1. 算法：

遍历、查找、增加、删除、修改、求长度、排序、清空、销毁；

删除节点：需要程序员手动释放被删除的节点，free(r);

插入节点：

算法：狭义的算法与数据的存储密切相关，广义的算法与数据的存储无关（这就是泛型的思想）；

泛型：利用某种技术达到的效果就是：不同的存储方式，执行的操作是一样的；

模块二

一，栈

栈的定义：一种可以实现“先进后出”的数据存储结构；（限制在一端进行插入和删除操作的线性表）

栈的分类：

静态栈：

动态栈：

栈的算法：

压栈：

出栈：

栈的建立步骤：

1. 建立一个空结点（struct类型的），他的数据域和指针域为空；
2. 建立一个结构体，他由两部分组成 栈顶指针（类型是结构体 \*）和 栈底指针；并且初始化都指向空结点；目前为止就建立了一个空栈；
3. 入栈：再次建立一个新的结点，数据域存放有效数据，指针域存放空结点的地址，并且将新的结点的地址赋给栈顶指针；
4. 出栈：将当前栈顶所指的结点的数据输出，再将下一个节点的地址赋给栈顶指针（也就是将当前出栈结点的指针域的地址值赋给栈顶指针）；

栈的应用：

函数的应用 中断 表达式求值 内存分配 缓冲处理 迷宫

1. 队列

队列的定义：一种可以实现“先进先出”的存储结构；

队列的分类：

链式队列：用链表实现；

静态队列：用数组实现；静态队列通常都必须是循环队列

循环队列的分析

1. 静态队列必须是循环队列：因为静态数组无法满足对队列的操作（知道就行）
2. 循环队列确定所需参数：需要两个参数（font和rear）但场合不同有不同的含义，
3. 循环队列名参数的含义：

（1）队列初始化（front和rear为0）、（2）队列非空时，front代表的是队列的第一个元素，rear代表的是队列的最后一个有效元素的下一个元素、（3）队列空时front和rear的值相等但不一定是0；

1. 循环队列的入队：

由两步完成

1. 将值存入rear所指向的位置；
2. 再进行rear = (rear+1)%数组的长度，注意不是rear=rear+1;入队的时候front不动，对rear+1;
3. 循环队列的出队：

Front=(front+1)%队列的长度，也可以先将出队的元素保存再释放，出队rear不改变，对front+1；

1. 如何判断循环队列是否为空：

如果front = =(rear+1)%数组长度，则队列为空

1. 如何判断循环队列是否为满：
2. 增加一个参数用来存放元素个数，当个数与数组长度相同时，则表示队列满；队列满的时候不能对队列进行入队操作；
3. 少用一个元素，当(rear+1)%数组长度==front时，该队列已满；（我们通常使用第二种方式）
4. 队列的应用：所有和时间有关的操作都涉及到队列（先进先出）
5. 递归

定义：一个函数自己直接或间接的调用自己

函数的调用：

当在一个函数的运行期间调用另外一个函数时，在运行被调函数之前，系统需要完成三件事：

1. 将所有的实际参数，返回地址（就是主调函数的下一条语句的执行地址）等信息传递给被调函数保存；
2. 为被调函数的局部变量（也包括形参）分配内存；
3. 将控制权转移到被调函数的入口；

就可以执行该函数；

从被调函数返回给主调函数之前，系统也要完成三件事：

1. 保存被调函数的返回结果；
2. 释放被调函数所占的存储空间；
3. 依照被调函数保存的返回值地址将控制权转移给主调函数

当有多个函数相互调用时，按照“后调用先返回”的原则，而这些函数之间信息的传递和控制转移必须借助“栈”来实现，即系统将整个程序运行的所需要的数据空间安排在一个栈中，每当调用一个函数时，就在栈顶分配一个存储区，进行压栈操作，每当一个函数退出时，就释放它的存储区，就进行出栈操作，当前运行的函数永远都在栈顶位置。

例如：A函数调用A函数，和A函数调用B函数，在计算机看来是没有区别的，这和我们的日常思维方式有些异同而已，在计算机中他们执行方式是一致的。

递归的三个必须条件：

1. 递归必须得有一个明确的终止条件；
2. 该函数所处理的数据规模必须在递减（不一定是指值）；
3. 这个转化必须是可解的；

循环和递归：

递归：

易于理解，但执行速度慢，占用空间较大；

循环：

不易理解，速度较快，占用空间较小；

汉诺塔：

盘子的移动 （具体看11次练习代码）

汉诺塔不是线性递归，这是个非线性递归！

当n=1，时 移动 1 次；

N=2，时 移动3次；

N =n，时 移动2的n-1次幂

当n=64 这是一个天文数字，就算是世界上最快的计算机也解决不了，二我们的递归也是属于算法灵域的一种思路

递归的应用：

树和森林就是以递归的方式定义的

树和图的很多算法都是用递归来实现的

很多数学公式就是以递归的方式来定义的如：斐波拉契序列： 1,3,5,8，13~~;

总结：

逻辑结构：（与计算机没什么关系）

线性：数组，链表；（栈和队列是一种特殊的线性结构）

非线性：树，图

物理结构：（实际的计算机存储）

1. 树

树定义：1有且只有一个称为根的节点，2有若干个互不相交的子树，3这颗子树本身也是一颗树；

通俗的说：1树是由节点和边（即指针域）组成，2每一个节点只有一个父节点，但可以有多个子节点；3，但根节点没有父节点；

专业术语：

节点，父节点，子节点，子孙，堂兄弟；

叶子节点：没有子节点的节点叫叶子节点；

非终端节点：实际就是非叶子节点；

度：一个节点的子节点的个数称为度；

节点的度：是指该结点的直接子节点的个数；度为0的结点称为树叶或终端结点；度数不为0的结点称为分支结点，除根外分支结点也叫内部结点；

树的度：是指在所有节点中，度最大的结点的度数；

深度（高度）：从根节点到最底层节点的层数称之为深度，根节点是第一层；

树分类：

一般树：任意一个节点的子节点个数都不受限制；

二叉树：任意一个节点的子节点的个数最多有两个，且两个子节点的位置不可更改；

二叉树分类：

一般二叉树：

满二叉树：在不增加树的层次的前提下，无法再添加节点的二叉树就是满二叉树；（也就是说，每一层节点都是满的）

满二叉树的性质：

1）、最后一层只能是叶子节点；

2）、非叶子节点的度一定是2；

3）、在相同深度的二叉树中，满二叉树的节点数一定是最多的，同时叶子也是最多的；

完全二叉树：如果只是删除满二叉树最底层最右边的连续若干个节点，这样的二叉树就是完全二叉树；满二叉树是完全二叉树的一个特例；

完全二叉树的性质：

1）、叶子节点只能出现在最下两层；

2）、最下层的叶子一定是从左至右连续排列的；

3）、倒数第二层若有叶子节点，一定都在右部连续位置；

4）、如果有节点的度为1，则该结点的孩子一定是左孩子；

5）、相同节点的二叉树中，完全完全二叉树的深度最小；

满二叉树一定是完全二叉树，但完全二叉树不一定是满二叉树；

二叉树与顺序结构：顺序结构可以看做一颗左斜树

二叉树的性质：

1. 二叉树的第i层上最多有2^(i-1)个结点；
2. 深度为k的二叉树，最多有2^k – 1个结点；

森林：N个互不相交的树的集合（也就是有多个树的集合）；

二叉树的性质：

1、在二叉树的第i层上最多有2^(i-1)个节点；（i>1）

2、深度为k的二叉树至多有2^k – 1个节点（k>=1）

3、如果对一颗有N个节点的完全二叉树的节点，按层次序号编程，对任一节点i（1<i<N）有以下性质：

1）、如果i=1，则节点i是二叉树的根，没有双亲；如果i>1，则其双亲是节点i/2的值取下线；(i表示节点的序号)

2）、如果2\*i>N，则节点i没有左孩子，（节点i为叶子节点），否则（2\*i<=N）其左孩子是节点2\*i；（即任意非叶子节点的左孩子（前提是由有孩子）是该结点的2\*i号）

3）、如果2\*i+1>N，则该结点无右孩子，否则其右孩子是节点2\*i+1；

树的存储：

二叉树的存储：连续存储（数组），该方式优点：查找某个父节点和子节点速度快（也包括判断二者是否存在），缺点是该方式存储耗费内存空间过大。

链式存储（随机）：

一般树的存储：

双亲表示法：每个节点都保存有父节点的下标，父节点的父下标为-1，求父节点方便；

孩子表示法：每个节点都保存子节点的下标，没有孩子的节点的孩子下标为空，求子节点方便

双亲孩子表示法示法：每个节点都存放着父节点的下标，并且每个节点都存放着子节点的指针；求父节点和子节点方便；

二叉树表示法：把一个普通树转化为二叉树进行存储（孩子兄弟表）：一般树转换为二叉树；

每一个节点的左指针域指向他的第一个孩子，右指针域指向他的兄弟；所以一个普通的树转换为二叉树，该二叉树一定没有右子树

森林的存储：先把森林转换成二叉树（即将所有的树统称为兄弟），再存储二叉树；

树的操作：（通常指二叉树）

遍历：（遍历而是相对于根而言的）

先序遍历：先访问根节点，再先序访问左子树，最后先序访问右子树；（先访问根节点）

中序遍历：中序遍历左子树，再访问根节点，再中序遍历右子树；（中间访问根节点）

后序遍历：中序遍历左子树，中序遍历右子树，再访问根节点；（最后访问根节点）

二叉树的还原：二叉树的还原只有两种方式：

已知先序和中序：先从先序中最先出现的确定为根，再从中序中确定根的位置分为左右，接着结合中序与先序（先出现者为根的方法）分别求得根左右的树。

已知中序和后序：先从后序中最后出现的确定为根，再从中序中确定根的位置分为左右，接着结合中序与后续（后出现者为根的方法）分别求得根左右的树。

利用以上两种方法求得原始二叉树后便能得到第三种遍历。

除了以上两种方式外，已知先序和后序或者只知道三种遍历的其中一种是无法将该二叉树还原的（也就是说必须要有中序）。

树应用：

树是数据库中数据组织的一种重要形式；

操作系统子父进程的关系就是一颗树；

面向对象语言中类的继承关系也是一颗树；

赫夫曼树也是利用树实现。

线索二叉树：

定义：在n个结点的二叉树中，有n+1个空指针域。用空指针域来指向结点在某种遍历次序下的前驱和后继结点的指针（这种附加的指针称为线索）；

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| lchild | ltag | data | rtag | rchild |

该线索二叉树的存在是为了解决，二叉链表在找左、右孩子困难的问题，但出现了无法直接找到该结点在某种遍历序列中的前驱和后继的问题。

结构

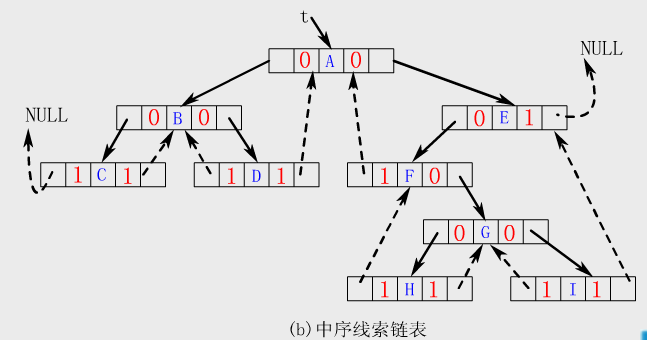
|  |  |  |
| --- | --- | --- |
| ltag | 0 | lchild是指向结点的左孩子的指针 |
| ltag | 1 | lchild是指向结点的前驱的左线索 |
| rtag | 0 | rchild是指向结点的右孩子的指针 |
| rtag | 1 | rchild是指向结点的后继的右线索 |

分类：

前序线索二叉树：

中序线索二叉树：

后续线索二叉树：



虚线表示前驱（左）和后继（右）；

NULL表示没有前驱或后继；

在线索二叉树中：判断结点是否是叶子节点的充要条件是：左右标志均为1；

赫夫曼树（最优二叉树）：

权：树结点与结点（前驱结点）之间连线相关的值叫做权（weight）；

结点的路劲长度：从根结点到该结点的路劲的连接数；

树的路劲长度：树中所有叶子结点的路劲长度之和；

结点带权路劲长度：结点路劲长度与该结点权值的乘积；

树的带权路劲长度：（WPL）是指树中所有叶子结点的带权路劲长度之和。

赫夫曼树：也叫最优二叉树，是一种带全路劲长度最短的二叉树；

赫夫曼树的构造：

（1）、将结点依照权值大小依次排序；

（2）、取出（相当于剪切）两个权值最小的结点，按照小的在左、大的在右，组成一颗新的树其根的权值为两孩子权值之和；

（3）、将新树以根的形式以权值大小从新插入排序到剩下的结点；

（4）、重复步骤2；直到结点全部取完；

赫夫曼编码：

定长编码：像ASCII编码统一长度编码

变长编码：单个编码的长度不一致，可根据整体出现频率来调节所需编码的长度；

前缀码：就是没有任何码字是其他码字的前缀（贪心算法）；

1. 图：

定义：图是由顶点的有穷非空集合和顶点之间的边的集合组成，通常表示为G(V, E)，其中，G表示一个图，V表示G中顶点（即图中的元素）的集合，E是G中边（顶点之间的逻辑关系）的集合

图相关术语：

（1）、无向边：顶点到顶点之间没有方向的边（E），用无序偶(vi, vj)表示，(vj,vi)没有先后；

（2）、有向边：顶点到顶点之间有方向的边，叫有向边，也称之为弧（Arc），用有序偶<vi, vj>来表示，vi称为弧尾，vj称为弧头，也就是从vi--->vj；

（3）、无向图：由无向边组成的图；

（4）、有向图：由有向边组成的图；

（5）、简单图：在G中，不存在顶点到其自身（自己指向自己）的边，且同一条边不重复出现，这样的G叫简单图；

（6）、无向完全图：在无向图中，任意两个顶点之间都存在边，该G就称之为无向完全图；含有n个顶点的无向完全图的边有 n \* (n-1) / 2 条；

（7）、有向完全图：在有向图中，任意两个顶点之间都存在方向相反的两条弧，则该图叫有向完全图；该图的边有 n \* (n-1) 条；

（8）、稀疏图和稠密图：稀疏和稠密是相对而言的，通常认为，边或弧 < n \* long n（n是顶点个数）的是稀疏图，反之称之为稠密图。

权：图的边或弧赋予的值叫权（weight），带权的图通常称之为网；

（9）、子图：就是截取图中某一部分的图叫子图，他可以是一个顶点，但在有向图中必须对应弧的方向，列：

两个图G1=(V1, E1)和G2=(V2, E2):如果V2V1，E2E1；则乘G2为G1的子图；

图的顶点与边之间的关系：

对于无向图G = (V, E)，如果边(v1, v2)  E，则称顶点v1，v2互为邻接点，即v1，v2相邻，边(v1, v2)依附于顶点v1，v2，也说边(v1, v2)与顶点v1或v2相关联。

顶点的度：顶点的度是指和该顶点相关联的边的数目，记为TD(V)；

对于有向图G=(V, E)，若有弧<v1, v2>E，则称顶点v1邻接到v2，或顶点v2邻接自v1；

以顶点V为弧头的弧的数目称为V的入度（indegree），记为ID(V)，以V为弧尾的弧的数目称为V的出度，记为(OD)；因此顶点V的度为TD(V)=ID(V)+OD(V)。

路径：

无向图G(V, E)中，从顶点V1--->V2所经过的边的总和称之为V1到V2的路径；

有向图G(V, E)中，从V1--->V2的路劲是由弧组成的；

路劲的长度：指的是进过的边或弧的数目；

环：从第一个顶点到最后一个顶点的所有边或弧能形成闭合图形叫环：

回路：从某一个顶点出发，能回到该顶点的闭合路径，称之为回路；

简单路径：从某一顶点到达最后一个顶点，除第一个顶点和最后一个顶点外，在任意一条路劲中都没有出现过相同顶点的路劲叫简单路径，而这样的路劲构成的回路或环叫简单环或简单回路；（了解就行）

连通图：在无向图中，如果从顶点V1到V2有路径，则称V1和V2是连通的，如果无向图中任意两个顶点之间都是连通的（有路径），则该图G是连通图；

连通分量：在无向图中，其某个连通子图具有极大顶点数，称为极大连通子图也叫连通分量，而极大顶点数的连通子图包含该图依附于的顶点的所有边。

强连通图：在有向图中：如果任意两个顶点之间都存在直接路劲，则该有向图是强连通图。

有向图中，极大强连通子图称为强连通分量。

连通图的生成树：

所谓的一个连通图的生成树是一个极小的连通图，它含有极小连通图的全部顶点（n），但只有足以构成一颗树的n-1条边。

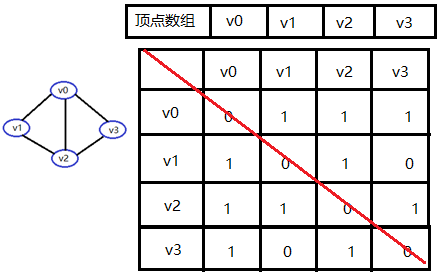
有向树：如果一个有向图只有一个顶点的入度为0，其余顶点的入度均为1，则该图是一颗有向树；

图的存储结构：

1）、邻接矩阵

图的邻接矩阵存储方式是用两个数组来表示的，一个一维数组存储图中的顶点信息，一个二维数组（称作邻接矩阵）存储图中的边或弧的信息。

无向图：



无向图的矩阵也叫对称矩阵：1表示有边，0表示不存在边；

在图用二维数组表示时，元素称作元，如：a[v0][v1] = 1；表示边v0到v1有边；a[v1][v3] = 0;表示v1到v3没有边；

邻接矩阵顶点的度：每一行或列的元之和就是表示该顶点的度，行或列均可表示（也可以表示该顶点的邻接点个数）；

有向图：有向图的邻接矩阵与无向图的构造基本类似；只不过有向图不一定是对称矩阵；

有向图的度是行和列元的总和：

入度：是该顶点列的和；

出度：是该顶点行的和；

网的邻接矩阵：



网的邻接矩阵的元，表示该顶点到某一个顶点的弧的权值，自己到自己用0表示，其中正无穷表示两个顶点之间没有弧；

邻接表：

无向图中：是将每个顶点存储在一维数组中，每一个顶点都有指向下一个邻接点的指针，依次往下，构成一个单链表形式；

有向图中：也是将每个顶点存储在一维数组中，每个弧尾都有指向下一个邻接点的指针，依次往下，构成一个单链表形式；

逆邻接表：是相对于有向图而言的，原理同上，只是将弧头建立指针，指向弧尾，形成单链表；

十字链表：

也是建立一个一维数组，存储所有的顶点，另外分别创建两个指针，一个指向它的下一个出度邻接点，另外一个指向它的另外一个入度邻接点，（有可能两个节点是同一个）；这样依次形成两条单链表（由一个顶点出发的两条单链表）；

邻接多重表：

边集数组：

图的遍历：

深度优先遍历：

根据某种原则，如右手原则，在没有碰到重复的顶点的情况下，遇到分支结点，靠右遍历，并且每经过的顶点都对其进行标记，当遍历到无法再进行该原则时，继续对没有遍历过的节点进行遍历；直到无法进行，依次退回寻找未曾遍历过的节点，对其进行遍历，直到退回到起初的节点，到此为止图中所有的节点都进行了遍历；

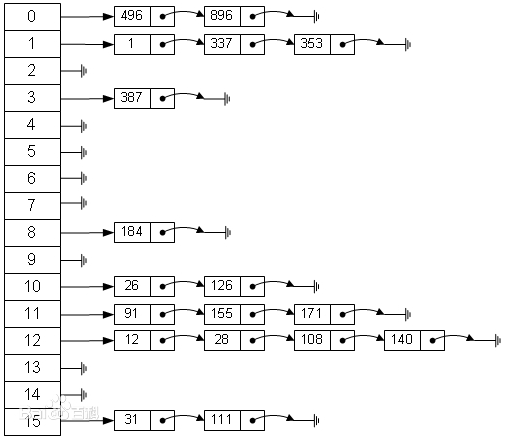
广度优先遍历：

图的广度优先遍历，是将图看作一颗树，对该树进行层次遍历，按照队列方式进行遍历；

HASH表：

Hash表也称散列表，是将静态存储和离散查询方式结合的一种存储查询操作，也就是通过key值计算出元素(value)所在hash表中的位置（相当于数组下标）来存储或查找元素，其获取元素所在位置的算法称为hash算法；

冲突：是指表项索引的冲突，由于hash索引的值是由hash算法计算出来的，根据算法的特性，多个值可能会计算出相同的索引（如 9%2 == 11%2），而在哈希表中，冲突是不可避免的，处理冲突的方式很多，通常的hash表都是数组和链表的结合，冲突的元素都放在同一个索引所指向的链表中，如下图所示：



Hash算法：hash算法就是根据要存储的元素的需求，所设计的key值所在下标的计算方法，常见的有取模运算、平方取中，随机种子等，所有的算法的最终目的就是通过key值计算出对应的hash索引；

排序和查找：

（1）排序和查找的关系：排序是查找的前提；

（2）排序：

排序的标准：时间，空间，稳定；

冒泡：将每个元素依次与他后面的元素两两比较，满足比较条件的都要与其互换，再进入下一个元素使用相同的方法，直到最后一个元素为止；

选择排序：依次选择每一个元素，将它与该数后面所有的元素比较，若发现满足排序条件，就将其与之兑换，直到选择到最后一个元素为止。

插入：

折半插入排序 ：

希尔排序：

归并排序：

快速排序：

方法1：

1）、将第一个元素（称作基准元素）保存起来(val)；

2）、定义两个指针l、h，分别指向低位置和高位置；

3）、先比较h所指的值与val的大小；

4）、如果\*h<=val 那么将\*h赋给\*l(\*l = \*h)；那么h不动；如果\*h>=val,那么h向低位置移动(h--)，直到找到一个(\*h<val)为止，并且将该值赋给l所指位置(\*l = \*h)，那么h就不变；

5）、现在\*l已经改变；将\*l与val比较，如果\*l<=val；那么l向后移动；直到发现\*l>val时则进行赋值（\*h = \*l）；

6）、重复上面3步骤，直到l和h相等，即同时指向一个位置，那么这个位置值就是val元素在排序中的位置；至此找到了第一个元素的位置，但其他元素依然是紊乱的，但是该元素左边的一定比该数小，右边的一定比该数大；

7）、用上面排好的一个元素作为分割元素，将剩下的元素分为左右两半，左右分别重复（1~7）步骤（即左快排与右快排），利用递归的思想，则将所有的数排列起来；