# 绪论和一些概念

## 第一章

**数据元素**是数据的**基本单位**，**数据项**是数据不可分割的**最小单位**。

数据元素之间关系在计算机中表示为：顺序映像和非顺序映像

四种基本结构：集合，线性结构，树形结构，网状结构。

//非线性结构指的，集合，树形结构，网状结构

**逻辑结构**：**逻辑关系**称为数据的逻辑结构是对关系的描述，比如线性结构和非线性结构，四种基本结构，和栈队列等，虽然范围的大小不同，但都可以称为逻辑结构

**存储结构**：是在计算机中的表示，又可以分为**顺序存储**(顺序映像)和**链式存储**(非顺序映像)，**索引存储**，**散列存储。**

区分二者主要看是否是独立于编程语言和计算机实体而存在的概念。这里有一些哲学的逻辑，比如队列和栈都是逻辑结构，他们独立于计算机和编程语言之外存在，没有计算机也有这个概念，而循环队列作为队列在编程中具体表达方式，它是存储结构，对应顺序存储。

* 注意提法”有序表”和”顺序表”,前者是元素有序，是逻辑结构；后者指存储地址有序，是存储结构

**抽象数据类型ADT** = 数据对象，数据关系，基本操作

戴帽子：**数据元素是基本单位，数据项是最小单位**

ADT它可以细分为原子类型，固定聚合类型，可变聚合类型

算法的五个特性：**有穷性，确定性，可行性，输入，输出**

1.又穷性，算法是执行时候运行的有穷性，程序只是一段实现算法的代码  
2.确定性，算法对于特定的输入有特定的输出，程序提供了确定算法结果的平台  
3.可行性，算法需要考虑设计的可能，程序则具体是实现算法上的设计  
4.输入，算法有输入，算法的输入依靠程序的平台提供  
5.输出，算法的输出也靠代码的支持

算法的四个设计要求：**正确性，可读性，健壮性，高效率和低存储量**

在考研中1>2>3>4 算法竞赛中1>4>2>3

* 时间复杂度取决于两方面：问题规模和问题初始状态。它与执行时间成正比而非问题规模。
* O(m+n)的时间复杂度，也可以表示成O (max(n,m))，因为m+n就可以近似看成2max(n,m),
* 提法”存取”指的就是查找，而不是插入删除。

# 线性表

## 概念和性质

大题步骤略，只写概念。

**顺序表**：相当于c++的vector或数组，有空和满的概念，元素在逻辑上物理上都相邻，它的插入删除是o(n)的，但修改和访问o(1)，所以支持随机访问。插入平均移动次数，在前插入，后，删除平均移动次数。

注意如果没说在哪插入，也就是有n+1个位置可以插入，平均移动次数是

顺序表插入的机制：和vector是一样的，有一个预留的额外缓冲空间，插入时，如果缓冲空间没位置，则从系统另申请一段空间把原本内容拷贝过去。比如原本长度是n,要插入m个元素但没有预留位置了，则重新申请n+m的连续空间，把原本内容拷贝(复杂度O(n)),再进行插入。

* 线性表的头尾没有直接前驱后继，而其他节点有

**链表**：元素在逻辑相邻而物理上不相邻，没什么好说的，考研注意看，看这个链表是不是有前驱节点，是单向还是双向，是不是循环链表

* 链表有没有前驱后继要具体分析，看他是否保存了相关信息
* 在代码填空题里，如果传进引用作为返回值，应该给引用赋值。
* 注意区分提法：”头结点”,”头指针”, ”尾结点”,” 尾指针”，凡是说节点：指的是单申请一个节点不存数据，而指针就是做一个标识，本身还是有数据的。

带头节点的链表有一个头L总是不变，故L->next是第一个元素，判断是空的条件是L->next==NULL

* 链表增加头结点目的是方便运算，而非找到首个元素位置，找到首个元素位置用”头指针”就够
* 对于有头节点的循环单链表，其头结点虽然不存元素，但也是和其他节点”成环”的，而不是孤立出来的节点。
* 严蔚敏的代码数组线性表里，下标范围[0,length) 但提”第i个”，i还是从1开始

## 多项式

对于一元多项式，直接用数组按照升幂的顺序储存系数

如果项数不多但指数很大，可以用二元组数组或者链表存储：

如：

可表示为：((5,0),(2,1),(-8,199),(1,999),(-10,1000007))

* 多项式采用升幂排列

# 栈队列

栈是后进先出LIFO，队列是先进先出FIFO。

实现略，注意循环队列：

* 栈和队列有着相同逻辑结构：线性结构，只不过它们基本操作不同

## 栈

数组写的栈有多重标准，严蔚敏的往往是[0,top)开区间

链栈往往用单项链表表示，头结点是栈顶top，每次插入删除要更新top

*\*共享栈是采取数组两端为栈底同时向中间插入的栈，优点是不会发生数组越界(上溢)*

## 循环队列

front代表队头，是数组下标较小的一端，rear代表队尾，是数组下标较大的一端。但是内容是[front,rear)，也就是队尾元素所在下标不是rear而是(rear+1)%MaxSize。注意循环队列不一定front<rear因为rear到达了MaxSize要回来从0开始。

上述规则为了让循环队列可以区分满和空，当(rear+1)%MaxSize ==front时为满

当rear==front时为空，相当于少用一个元素来区分满和ø空

循环队列也可以用其它方法区分满和空，比如单门做一个标记，在入队出队时改变，或者维护元素个数size。

循环队列应记住，头小尾大，左闭右开也就是[front, rear)

但链表队列是： [rear ,front),逻辑上是：front->rear

## 一些应用

### 前缀中序后续的表达式的转换

后缀表达式：操作运算符靠后的表达式 //什么缀指得是运算符在哪

前缀表达式：操作运算符靠前的表达式

转换的总规则是：后：出大于等于，前：出严格大于还得倒着

**中序转后缀表达式的转换规则**：

1. 从中序表达式字符串里，按正着的顺序取出字符
2. 遇到操作数：直接输出（添加到后缀表达式中）
3. 遇到左括号”(”，直接入栈。
4. 遇到右括号”)”：执行出栈操作，并将出栈的元素输出，直到弹出栈的是左括号。 注意左右括号都不输出到后缀表达式里。
5. 遇到其他运算符：加减乘除：弹出所有优先级**大于等于**该运算符的栈顶元素，停止条件是：栈为空 || 遇到左括号 || 遇到了优先级**小于**的运算符。

然后将该运算符入栈

1. 最终将栈中的元素依次出栈，输出。

样例：1×3/(2-1)+3×(4-1) 答案：1 3 \* 2 1 - / 3 4 1 - \* +

**中序转前序表达式的转换规则**：

1. 从中序表达式字符串里，按倒着的顺序取出字符
2. 遇到操作数：直接输出（添加到后缀表达式中）
3. 遇到右括号”)”，直接入栈。
4. 遇到左括号”(”：执行出栈操作，并将出栈的元素输出，直到弹出栈的是左括号。 注意左右括号都不输出到后缀表达式里。
5. 遇到其他运算符：加减乘除：弹出所有优先级**大于**该运算符的栈顶元素，停止条件是：栈为空 || 遇到左括号 || 遇到了优先级**小于等于**的运算符。

然后将该运算符入栈

1. 最终将栈中的元素依次出栈，输出，将答案反转一次即可。

样例：1×3/(2-1)+3×(4-1) 答案：+ / \* 1 3 – 2 1 \* 3 – 4 1

# 字符串//考前记一记

## 概念

严蔚敏书中字符串所以的”pos”都从1开始，不仅如此，字符串操作所有的数组下标都从1开始.

操作函数：

Assign(s1,s2) 表示把s2赋值给s1,即s1=s2,属于地址复制

Creat(s1,s2) 表示把s2的拷贝赋值给s1 即c++的strcpy

Concat (s1,s2)从左到右链接字符串，

Substr(s,**pos**,len)截取pos开始的长度为len的字符串·

index(s,t,**pos**)：在s中查找t,从pos开始找的第一个位置，s中没有t返回-1

replace(s,t,v)：用v替换s中所有与t相等的，还不重叠的字串t.

strinsert(s,**pos**,t)：在s中第pos位之前插入t

strdelete(s,**pos**,len)：在s中第pos位开始删除长度为len的字符串

## KMP记忆方法

KMP的next数组不尽相同，考研主要是对next和nextval记忆

统一记忆next[0]=-1的形式，之后再全+1转化，只要记住了next数组是

1. nex[0]= k=-1;
2. **for**(i=1;i<len2;i++){  //这是普通 next的
3. k=nex[i-1];  //改进的没有这句
4. **while**(k>-1&&s2[k]!=s2[i-1]){
5. k=nex[k];
6. }
7. k++;
8. nex[i]=k; //改进的：nex[i]=(s2[k]==s2[i]?nex[k]:k);
9. }

**Next流程**：

1. 找到上一个的next值，作为初始值k。

2. 如果k是-1，则该点next是0

如果k满足s2[k]==s2[i-1]，则该点next是k+1

如果k是s2[k]!=s2[i-1]，则k=next[k],重复此步骤2

**NextVAL流程**：

1. 手动记录一个k,写在纸的空白处，时刻核对和更改，初始值k=-1。

2. 如果k是-1，k++,

如果k满足s2[k]==s2[i-1], k++,

如果k满足s2[k]!=s2[i-1]，则k=next[k],重复步骤2

3. 要注意，给next赋值时，还有判断当前位置和next[k]的关系

如果相等，next[i]赋值成next[k]，不等就按照普通kmp那样next[i]赋值成k

而nextval和next代码不同点

1. 循环内部k不需要每次重新初始值而是沿用上次的
2. 对nextval [i]赋值时判断s2[k]==s2[i],相等说明：有可能匹配，就赋值成nextval[k]；不等说明不可能在这个位置匹配，就和next的一样赋值成k

* Next数组用这个方法甚至用定义很好找，而nextval就比较麻烦，容易错

# 矩阵压缩和广义表

## 广义表和存储

### 定义

**表头**：指广义表非空时，第一个元素称为表头

**表尾**：指广义表非空时，第一个元素以外的元素组成的广义表称为表尾

表头是元素，而表尾是广义表，且只有非空广义表才能有表头和表尾。

**原子：**不可再分割的元素，如(a,b,c,(a,b,c))的a,b,c都是原子，考试时使用小写字母表示原子。

**列表，子表：**可再分割元素，如(a,b,c,(a,b,c))的(a,b,c)是子表，考试时使用大写字母表示子表,上面例子表示成(a,b,c,A)。A就代表(a,b,c)

**长度：**最外层广义表元素个数，数”逗号”数量加一

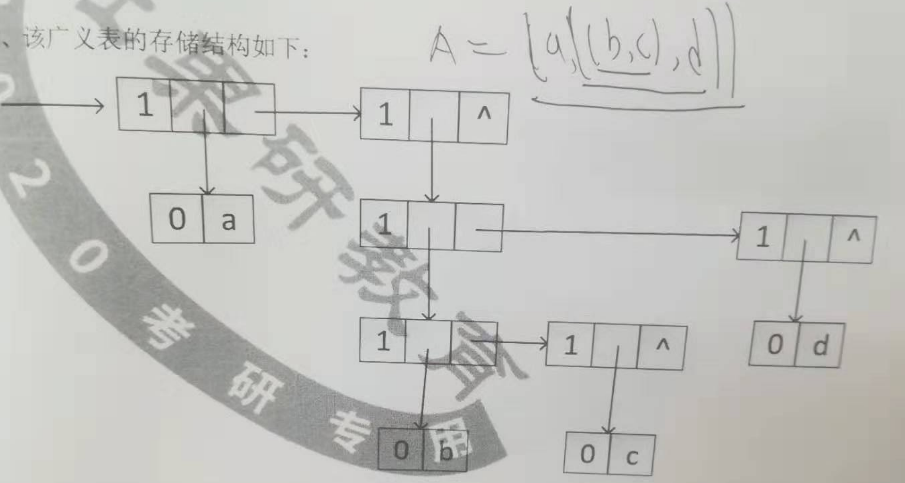
**深度：**有多少层，数括号层数(最多的)，比如(a,c,f,(a,(a,c),c),(((a)),b)) 深度是4

### 存储方式

多重链表

1. **enum** AtomType{**ATOM**,LIST};
2. **template**<**class** Data>**struct** Node{
3. AtomType type;//枚举类型，也可以直接用ool表示
4. **union**{//联合体用于节省内存，内部二者只能有一个同时存在
5. Data atom;
6. Node \*hp;//下一层表
7. };
8. Node \*tp;//下一个元素
9. };

在画图时，广义表应该区分原子和非原子节点，用两种框框画具体如下：



**广义表存储结构示意图和表推导：**

图推表：

如果某个节点有2个及以上分支，应该写一层括号

如果某个节点有1个分支，应该继续找，什么都不做，往下写

表推图：

一层一层画，每层节点数是这层的表长度

框框含义：

原子节点用2个方框，左边写”0”右边字母。

非原子节点用3个方框，左边”1”，中间写向下指针，右边写向右指针，没有指针写“^”。

从上到下只看非原子节点，其层数代表：广义表的深度

每层从左到右节点数代表：广义表对应层的长度，最顶层节点个数就是广义表长度

## 矩阵的压缩存储

无规则矩阵在计算机中仅仅二维数组即可，但有规则的矩阵就可以减少空间。

以下约定：矩阵的元素a[i][j]下标从1开始，而数组A[k]下标k从0开始

先要熟悉如下概念：

**行主序存储**：按照横着编号，到尽头转向下一行开始位置

**列主序存储**：按照列着编号，到尽头转向下一列开始位置

* **对称矩阵**：对称矩阵的有一半相同,用数组存储一半可以减少空间

为了计算简便和对称性，一般**行存储下一半，列存储上一半**

其中，上述公式分布代表存矩阵 下一半 和 上一半

同时，已知k，还可以知道位置(i,j),反向索引无o(1)公式。

对于上三角部分，找不大于k的最大三角数s(x).则(i,j)=(k%s(x)+1,x+1)

对于下三角部分，找不大于k的最大三角数s(x).则(i,j)=(x+1,k%s(x)+1)

三角数是满足0 1 3 6 10 15………..它的定义是

* (上/下)三角矩阵：(上/下)半部分和对称矩阵一样的方式存储, (下/上)半部分元素相同，都是c, 当k==+n 时A[+n]=c
* 对角矩阵：把对角线元素放在一维数组即可。
* 三对角矩阵：从按照a11,a12,a21,a22,a23,a32,a33,a34,a43…k=2(I-1)+J-1

//数组的行下标指的是矩阵第几行，也就是平面矩阵的”高”

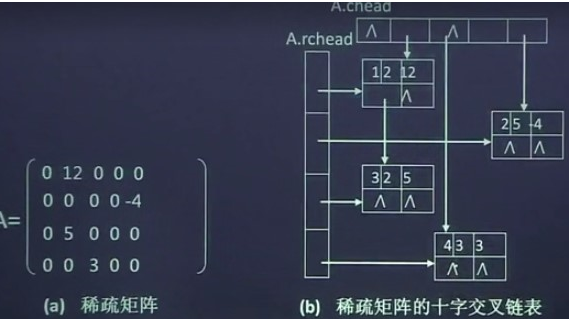
### 稀疏矩阵压缩

设矩阵A是一个nxm的矩阵，里面有s个非零元素。

设 δ = s /(nm)，称矩阵δ为稀疏因子，如果某一矩阵的稀疏因子δ满足

δ<=0.05时称为稀疏矩阵。

**三元组**.即：(i,j,a[i][j])的方式，类似图的存储里边集数组。

**十字链表**：

**带辅助向量的三元组(带行三元组表)**

在普通三元组基础上，对三元组按照先行后列的规则排序。并且增加维护pos[i]代表矩阵第i行的非0元素在三元组表的第一个的下标。

也可用num[i]代表矩阵第i行非0元素个数，但这是非必要属性，用pos就可以得到个数和遍历每行。所谓带行三元组表，的”**行表**”，就指的pos数组。

# 树

## 概念和性质

**双亲节点，父亲节点**：顾名思义，就是父亲，now->p表示

**路径：**从树中一个结点到另一个结点之间的分支构成两个结点的路径。

**树的路径长度：**从树根到每一个叶子结点的路径长度

**树的带权路径长度WPL**：从树根到每一个叶子结点的路径长度 乘以 叶子节点权值 的之和。

**m叉树：**出度最多是m的树。

**正则m叉树：**每个节点出度都是m,即：每个节点都有m个叉。

**森林**：多棵树的集合，叉数也不一定是二叉

**节点度数**：有几个分叉

**树的分支**：总共有多少个叉，数量上等于节点数减一

单向树：如果一个节点不是它的父亲任意儿子，则

**整棵树的度数**：度数最大的节点的度数

有结论：任何二叉树里表示度数是i的节点个数，= +1

**平衡二叉树**：对于任何节点，它的左右两个子树的高度差的绝对值不超过1，空树也算平衡二叉树。

**满二叉树**：每一个层的结点数都达到最大值的二叉树,节点数恰好是的二叉树，h是高度

完全二叉树：除了最后一层，其余层都达到最大个数。最后一层至少一个节点。

**完全二叉树(严蔚敏)**：除了最后一层，其余层都达到最大个数。最后一层至少一个节点，且从最左侧依次无间隔排列。

**平衡二叉树**：定义每个节点**平衡因子**是右子树高度减去左子树高度，则平衡二叉树定义为每个节点**平衡因子绝对值不大于1**的二叉树

**层序遍历**：相当于广度优先搜索

**先序遍历**：先访问根，然后按顺序递归儿子， 多叉树叫**先根便利**

**中序遍历**：把访问根方中间，只有二叉树有中序遍历

**后序遍历**：最后访问根，先按顺序递归儿子， 多叉树叫**后根便利**

**性质** (约定根节点高度是1，是第一层)

* 第k层有至多个节点

前k层有至多个节点

* 不论什么二叉树，叶子节点个数= +1

不论什么m叉树，叶子节点个数=

可见任何树里，度数是1的节点不影响叶子节点数

* n个节点的完全二叉树深度是floor()+1, 或ceil()

n个节点的完全m叉树深度是ceil(),

* n个叶子节点的完全二叉树深度是floor()+1,
* 完全二叉树节点关系：

n1= 1或0， n=2n0 + n1-1, n0=floor((n+1)/2)

* 深度为h的完全二叉树叶子节点个数范围[]

深度为h的完全二叉树所有节点个数范围[]

* 深度为h的平衡二叉树叶子节点个数范围[Fib[h],]

深度为h的平衡二叉树所有节点个数范围[F[h],]

其中Fib代表**斐波纳妾数列**Fib[1]=1, Fib [2]=1,Fib[n]=Fib[n-1]+Fib[n-2]

其中F代表**类斐波纳妾数列**F[1]=1, F [2]=2,F[n]=F[n-1]+F[n-2]+1

* 只有度数为m和0的m叉树，总结点数必定为

## 编码细节：

### 严蔚敏版树的存储

双亲表示法：只存储一个父亲

孩子表示法：也就是多叉链表

孩子兄弟表示法：二叉链表存储2个指针：firstChild和nextSibling表示该节点的第一个儿子，以及下一个兄弟

### 索引方式

**指针索引：**就是定义结构体时，储存左右位置的数据用指针表示

1. **struct** Node{
2. **int** sum=0;
3. Node \*son[2];
4. };  //用的时候要一个个实例化节点

**乘法索引：**就是定义结构体时，不储存左右位置的，而是通过乘法获取，对于m叉树,tree[now]代表当前节点的话，tree[now\*m+i]代表左起第i个子树,i∈[0,m-1]

1. **struct** Node{
2. **int** sum=0;
3. };
4. Node tree[MAX];
5. //可以证明tree[now\*2]代表左，tree[now\*2+1]代表右，能严格紧凑的把节点排列在数组里

任何静态的完全二叉树都可以用这种方式，因为完全二叉树按照自上而下自左到右编码顺序正好能满足tree[now\*2]代表左，tree[now\*2+1]代表右，且能严格紧凑的把节点排列在数组里

**拟指针索引：**就是定义结构体时，储存左右位置的数据,且用数组模拟指针，对于m叉树,tree[now]代表当前节点的话，tree[tree[now].son[i]] 代表左起第i个子树，i∈[0,m-1]

1. **struct** Node{
2. **int** sum,son[2];
3. };
4. Node tree[MAX];

以上三种索引方式，实际用法里效率依次提高，最高的是拟指针索引，因为它一次性申请了一大片数据。相比指针索引(用的时候要一个个开辟空间)，一次性申请MAX个节点的内存比分着申请MAX个快。相比乘法索引，不需要通过乘法和加法运算来得到位置。

### 非递归遍历

## 前序中序后序，知其二推树：

必须已知中序便利才能唯一确定一颗二叉树。

在进行分析前读者需要知道不同遍历结果的特点，设树有n个节点。

**前序遍历，序列是DLR**：

* 第一元素是整个二叉树的根节点。
* 剩余元素可以分成连续的两部分：前c1个和后c2个，c1+c2=n-1.其中c1是根的左子树，c2是根的右子树。

**中序遍历，序列是LDR**：

* 根的位置不确定，但可知道：根的左边全部是左子树的节点，右边是右子树的节点。

**后序遍历，序列是LRD**：

* 最后元素是整个二叉树的根节点。
* 剩余元素可以分成连续的两部分：前c1个和后c2个，c1+c2=n-1.其中c1是根的左子树，c2是根的右子树。

**已知先中，求后**

1. 首先确立根的位置r=DLR[0]
2. 找到DLR[0]在中序遍历LDR里所在位置设为p查看它左边元素个数c1，右边个数是c2
3. 在先序遍历DLR里顺着r往后看c1个，作递归，右边同理。继续返回步骤1。此时DLR[1]就表示r的左儿子(如果存在的话), 注意递归的过程中在序列DLR和LDR上的位置不是对齐的。所以要维护l1,r1,l2,r2

**已知中后，求先**

1. 首先确立根的位置r=LRD [n-1]
2. 找到LRD [n-1]在中序遍历LDR里所在位置设为p查看它左边元素个数c1,右边个数是c2
3. 在后序遍历LRD里从0往后看c1个，作递归，继续返回步骤1。

LRD [0]就表示r的左儿子(如果存在的话), 注意递归的过程中在序列LRD和LDR上的位置不是对齐的。所以要维护l1,r1,l2,r2

**已知先后，求中，结果不唯一**

不唯一，对于单链的子树可以有很多种。

## 二叉树线索化

### 概念和性质

**二叉树的访问顺序和线索化**

二叉树的线索化就是按照访问顺序把二叉树的节点链接起来(这只是逻辑上，实际上无我们不这样干) ,二叉树的深度优先遍历是深度优先搜索(dfs)的过程，访问节点的顺序统称：”**dfs序**”。

由于二叉树可分为先序，中序，后序遍历，那么线索二叉树也分为，分为先序线索二叉树，中序线索二叉树，后序线索二叉树。

**线索二叉树**

对于n个节点的二叉树(无论形态如何)，有2n个指针，其中一定有n-1个非空，n+1个是空，就用这n+1个作为线索指针，包括了2个指向空的指针(第一个节点的前驱指针和最后一个节点的后继指针)

对于n个节点的二叉树，建立线索二叉树，并不是说要单门维护一个属性来存线索，

而是只需要用其中n+1个空指针即可(把度数小于2的节点的空指针当线索指针)：对于左指针储存**线索前驱**，右指针储存**线索后继**。

相当于把空闲的指针”废物利用”，加上原本有用的n-1个，正好完整的用完了2n个指针，所以一个线索二叉树有n+1个线索指针。

线索化就是要找每个点在搜索过程中的访问顺序的前驱后继结点。

* 注意，所谓“访问顺序”是指代码里输出的环节的顺序，而不是递归函数的执行顺序。
* 而且线索二叉树逻辑上和实际实现的区别,即，逻辑上只对度数小于2的节点做索引。

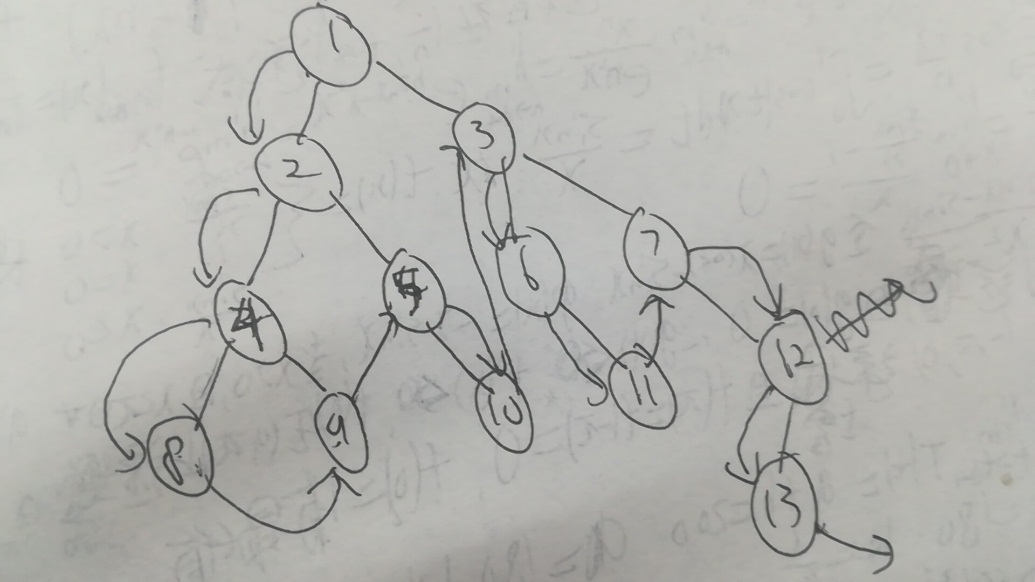
本文档的图片为了方便理解，均采用后继式的逻辑上的表达方式。会出现一个节点有三个指针的情况。

### 先序线索二叉树

规律：

* 第一个节点是：根节点。
* 最后一个节点是：从根开始先右后左的一直走的最后一个节点。
* 左叶子节点的后继是右兄弟，没有的话找父亲的兄弟…..
* 右叶子节点的后继是:向上找第一个fp()==0的节点的右兄弟节点
* 非叶子节点的后继是：先左儿子后右儿子

下图是先的后继线索二叉树，其线索顺序是先序遍历的dfs序

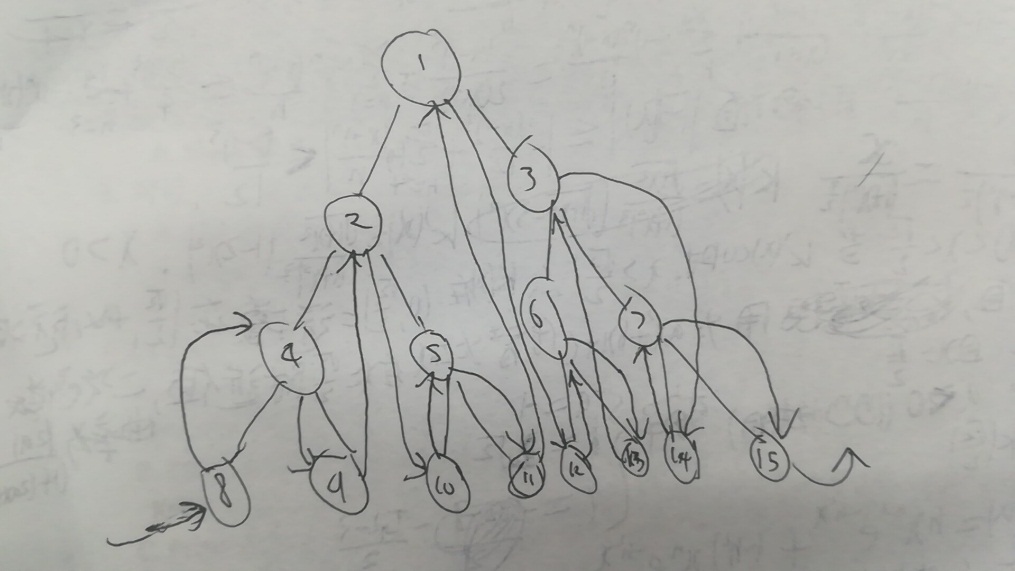


### 中序线索二叉树

规律：

* 第一个节点是：最左侧节点 (左进右出)
* 最后一个节点是：最右侧节点 (左进右出)
* 左叶子节点的后继是父亲，
* 右叶子节点的后继是:向上找第一个fp()==0的节点的父亲节点
* 非叶子节点的后继是：右子树的最左侧节点

下图是中序的后继线索二叉树，其线索顺序是中序遍历的dfs序

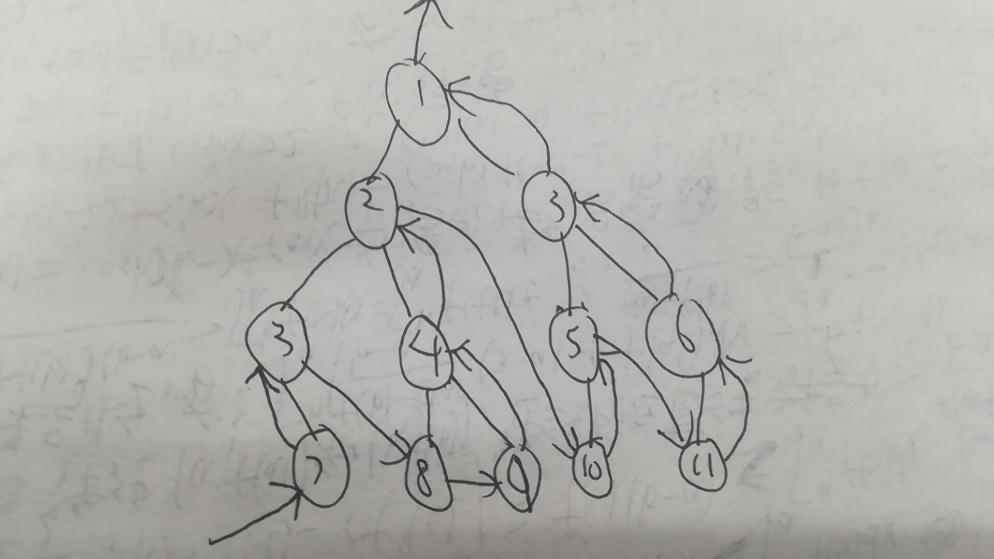


### 后序线索二叉树

规律：

* 第一个节点是：从根开始先左后右的一直走的最后一个节点。
* 最后一个节点是：根节点。
* fp()==0节点的后继是：从右兄弟开始先左后右的一直走的最后一个节点，没有右兄弟就指向父亲。
* fp()==1节点的后继是：父亲。

下图是后序的后继线索二叉树，其线索顺序是后序遍历的dfs序

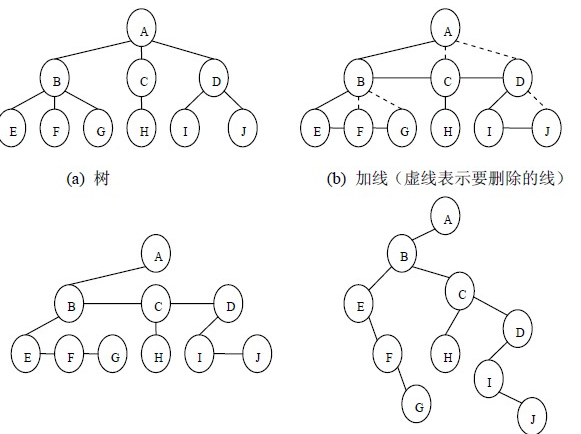


### 细节

* 在考研提法中：只有后续遍历可以得到路径，因为不需要额外存储结构 //实际上任何遍历维护p[]数组即可
* 用遍历进行节点编号，要求每个节点的编号大于其左右儿子，且右儿子编号也大于左儿子，应该采用后序遍历，而非层序遍历
* 后续线索二叉树的遍历，仍然不能通过其线索指针实现，依然需要递归也就是栈。
* 给定先序遍历序列A[n]，求二叉树形态数量，可以归结为求A[n]每个元素按顺序进栈再出栈，正好是卡特兰数经典模型。
* 单独说“二叉树”是逻辑结构，而”线索二叉树”是存储结构或物理结构

## 森林与树与二叉树的转换

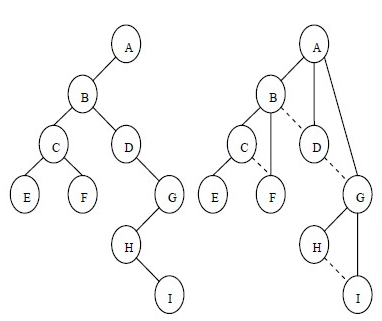
### 树变成二叉树



采用的方法如图，对于每个有2个及以上儿子的节点的儿子们操作：

把兄弟节点之间连线，且线是由左指向右的有向线段，之后擦去除了第一个节点之外的其他兄弟的与父亲的连线。即可得到二叉树，然后整理一下即可。

### 二叉树变成树



定义getfp()为每个节点是父亲的左还是右，getfp()==0代表左，getfp()==1代表右

对于二叉树的每个非根节点，如果getfp()==1，就进行操作。

先执行如下程序

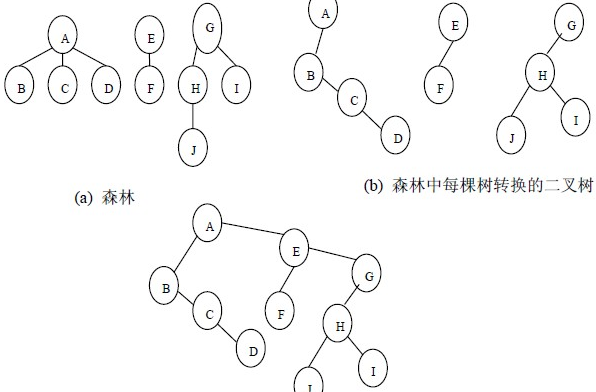
1. now=node;
2. **while**(now->parent!=NULL&&now->getfp()==1){
3. now= now->parent;
4. }
5. PP= now->parent;//不用担心是空，因为能变成树的二叉树的root没有右儿子

找到最早的祖先成为PP，对刚才遍历的路径再遍历一次，父亲赋值成PP

1. now=node;
2. **while**(now->parent!=NULL&&now->getfp()==1){
3. now= now->parent;
4. next=now->parent;
5. now->parent=PP;
6. now=next;
7. }

### 森林变成二叉树

将森林转换为二叉树的步骤是：  
（1）先把每棵树转换为二叉树；  
（2）第一棵二叉树不动，从第二棵二叉树开始，依次把后一棵二叉树的根结点作为前一棵二叉树的根结点的右孩子结点，用线连接起来。当所有的二叉树连接起来后得到的二叉树就是由森林转换得到的二叉树。



### 二叉树变成森林

二叉树转换为森林比较简单，其步骤如下：  
（1）先把每个结点与右孩子结点的连线删除，得到分离的二叉树；  
（2）把分离后的每棵二叉树转换为树；  
（3）整理第（2）步得到的树，使之规范，这样得到森林。

### 一些性质和提法

* 树转换成的二叉树一定没有右儿子

能转换成树的二叉树，也一定没有右儿子

* 森林转换成的二叉树一定有右儿子

能转换成森林的二叉树，也一定有右儿子

* 一个树一定能转换二叉树，对应唯一的二叉树；

一个森林一定能转换二叉树，对应唯一的二叉树；

一棵二叉树能转换森林或者树，要看一直往右找能找到几个节点。

* 普通多叉树或森林为F，B是F对应的二叉树，F非叶子节点个数为n

则二叉树B中右指针为空的节点个数为n+1,或B中度数为0和1的节点总和

则二叉树B中左指针为空的节点个数为n

* 树，二叉树，森林三者的形态之间关系

|  |  |  |
| --- | --- | --- |
| 普通多叉树 | 森林(普通树组成的) | 二叉树 |
| 先根遍历 | 先序遍历->按顺序先根遍历每棵树 | 先序遍历 |
| 后根遍历 | 中序遍历->按顺序后根遍历每棵树 | 中序遍历 |
| NAN(无中序遍历) | NAN(无后序遍历) | 后序遍历 |

森林永远没有后序遍历。普通多叉树远没有中序遍历。

森林的(先/中)序遍历相当于它内部的每棵树进行(先/后)根遍历。

只要记住，不管是森林对应的多树还是其内部每棵子树，都是中序对应普通多叉树的后跟遍历，反之亦成立。

**树和森林用二叉树存储表示叫做：孩子兄弟表示法**

* 森林转换成二叉树的左子树节点数n和原本森林第一棵树节点树m关系是：n=m-1

因为根节点也属于第一棵树。

## 哈夫曼树

### 概念和性质

对于一个m阶哈夫曼树(m>=2)又称最优树，任何叶子节点度数为0,非叶子度数为m，但任何m阶哈夫曼树不存在度数是1的节点。

对于二阶哈夫曼树，又叫**最优二叉树**

**路径：**从树中一个结点到另一个结点之间的分支构成两个结点的路径。

**路径长度：**某条路径上边的条数。

**树的路径长度WPL**：从树根到每一个叶子结点的路径长度 乘以 叶子节点权值 的之和。

WPL= //它是叶子节点的带权路径长度之和

代表某个叶子节点的权值，一共n个叶子节点，代表从根节点到某个权值为的叶子节点的路径长度

**哈夫曼树整体的权值**：就是除了根节点外其余节点权值总和,因为根节点权值是，所以**哈夫曼树整体的权值**也是所有节点权值和 减去

**性质**

* n个数可以构建的哈夫曼树有个叶子,非叶子节点,总节点数叶子是2-1

对于m阶哈弗曼树，

* 哈夫曼树的形态不唯一，根源是可能中间某个过程出现两个权值相同的节点，就可能产生2个分支。只有当：包括原本的数在内的任何过程中，都没有相同的数时，才能保证哈夫曼树形态唯一。
* 最优二叉树也是一类**叶子节点**带权路径长度最短的树，意思是说：用n个数作为叶子，构建二叉树里，最优二叉树的**叶子节点**带权路径长度最小
* 提法：”哈夫曼殊非叶子节点的权值是儿子的和”在考研里是错误的，因为非叶子节点没有权值，只是构建的时候模拟出来的。

### 构建

给你n个节点:{}构建哈夫曼树

方法是：选取权值最小的两个节点，求它的的权值之和s=建新的节点权值是s，把权值是和的节点当它的儿子。把s当做新的节点，和其他节点继续做上述操作。每次这么做会把2个节点合并成一棵树。到最后执行n-1次后，所有节都会被合并成一棵树，算上新建的节点，最终树的节点总数是2n-1

哈夫曼树的构建需要用到二叉堆。

由此可见，最优二叉树也是一类**叶子节点**带权路径长度最短的树，意思是说：用n个数作为叶子，构建二叉树里，最优二叉树的**叶子节点**带权路径长度最小。

### 哈夫曼编码

定义一个树的某个节点的左边方向为0，右边方向为1，则一个权值的哈夫曼编码定义为：对哈夫曼树的到这个权值所对应的的叶子，走的路径的方向组成的01字符串。

n个数可以构建的哈夫曼树有n个叶子,也就有n个01字符串。由于哈夫曼树不一定唯一，所以这些01字符串也不一定唯一，但哈夫曼树的**带权路径长度**唯一，且相比其他构建方法(每次不取最小的两个)来说是最小的。

**码字:**一个编码后的01字符串。因为这个01字符串代表一个字符，且之间相互不同。

**前缀编码：**每个编码后的字符的01字符串，都不是其他01字符串的前缀，前缀编码是对一组01字符串来说的。

### 应用：

**编码字符串：**

给定字符串str,假设它有26\*2个大写和小写字母组成，每个字母对应权值就是出现频率，哈夫曼树的叶子节点的点权是字母出现频率，每个点左边代表0，右边代表1

从根节点走到叶子的路径形成的01字符串代表这个字符的编码。

如此得到的**哈夫曼编码**有如下性质：

* 一定是**前缀编码。**
* 每个字符的编码长度总和最小(相比不用字母出现频率作为权值)
* 如果给定每个字符的概率，则把概率当作权值构建哈夫曼树。即：每次选取概率最小的加和构建哈夫曼树
* 如果让哈曼编码极可能种类多，就让位数多，因为哈弗曼编码要求不存在一个是另一个的前缀。

如给定1和01，让你求长度不超过4的编码最多还能有多少：如果取3位的001,0001,

0000这样只能有3个。但如果全取四位：0000,0001,0010,0011则有4个

## 二叉堆

* 考研中给序列建立二叉堆，要把整个堆先写出来，然后自下而上调整，而不是一个个节点的插入

## 败者树与胜者树

## BST树和AVL树。

* 对于BST树把一个节点删除再插入：如果删了叶子节点，前后两颗树一定相同；如果删了非叶子节点，前后两颗树一定不同
* 对于AVL树把一个节点删除再插入：如果删了叶子节点，前后两颗树不一定相同，如果删了非叶子节点，前后两颗树也不一定相同

# 图论

## 概念

图由顶点和边构成，用V表示点数，E表示边数

图的**路径**：由顶点和相邻序偶构成的边所形成的序列。

**回路**：从一点出发，走不同的边又回到原点，回路不一定是简单路径。

**有向图**：顶点集+弧集，**弧**是有向图的概念，边u->v,点u为**弧尾**，v为**弧头**

//弧尾是起点，弧头是末点

**无向图**：顶点集+边集

**稀疏图，稠密图**：E<VlogV为稀疏图，不同资料可能会有所调整。

**完全图**: 无向图需要满足E=V(V-1)/2，有向图需要满足E==V(V-1)

**网**：就是有边权的图

**度**：有向图的度是临接点个数，u的度TD(u)

无向图入度记作ID(u),出度记作OD(u),则TD(u)= ID(u)+OD(u)

任何图E= //表述为所有顶点度数之和除以2等于边数

**简单路径**：路径中每个点只出现一次。

**简单回路，简单环**：环的路径上，除了第一个和最后一个点，每个点只出现一次。

**割点**：无向连通图或有向连通图中，如果删除某点后，图变成不连通，则称该点为割点。

**割边**：也叫桥，无向连通图或有向连通图中，如果删除某边后，图变成不连通，则称该边为桥或割边。

**极大连通子图：**

”图的块”。对于图的某个子图，它内部的点之间相互可达，但从这个子图之外再引入一个点，构成的图变得不连通，那这个子图就是整个图的极大连通子图，在有向图里叫**极大强连通字图**。

**连通图**：

对无向图来说的，任意一对点u,v都存在至少一条路径相互可达的图，叫做连通图，无向图的**极大连通子图**，称为**连通分量。**

**双连通分图**：

无向图来说的，双连通图分为**点双连通分图**和**边双连通分图**，去掉一个**(点/边)**后，剩余的**(点/边)**构成的图仍然是连通图的图，即不存在**割(点/边)**，叫做**(点/边)**双连通图。无向图的**极大(点/边)双连通子图**，称为**(点/边)双连通分量。**

**强连通图**：

对有向图来说的，任意一对点u,v都存在至少一条路径相互可达的图，叫做强连通图，有向图的**极大强连通子图**，称为**强连通分量**

**简单图**：不含有重边和自环的图

**仙人掌图**：任意一条边之多属于一个**简单回路**的**简单图**，我们就称这张图为仙人图（cactus）。所谓简单回路就是指在图上不重复经过任何一个顶点的回路。

**有向无环图(DAG)**：一定是有向图，且是没有任何环的**简单图**。

**二分图**：又称作二部图， 设G=(V,E)是一个无向图，如果顶点V可分割为两个互不相交的子集(A,B)，并且图中的每条边（i，j）所关联的两个顶点i和j分别属于这两个不同的顶点集(i in A,j in B)，则称图G为一个二分图。

**度矩阵**：对角线元素是图的对应点的度数，所构成的对角矩阵

* 提法“若图G点集为V边集为E，V1∈V，E1∈E，则V1和E1构成的图G1是G的子图”是错误的。因为有可能E1的边的端点不包含在V1里，G1都构不成图，前提就不成立，G1有可能都不是图。
* =2cnt(E),即：无向图顶点度数的总和等于边数的两倍.
* 区分生成树和联通分量，对于无向图的联通分量定义为极大连通子图，也就是，这个非连通图的某一块，要选取尽可能多的边，而生成树则是选取尽可能少的边。二者是相反关系。

## 图的遍历

**时间复杂度**：不管dfs还是bfs，时间复杂度相同，邻接矩阵V\*V，邻接表V+E。

邻接表找邻接点时间为E，但容易忽略遍历每点本身的复杂度V

* 图的遍历注意，输出边还是点，而且先序后续要搞清楚，尤其是关注第一个点输出什么
* 在考研数据结构里，临接表由点表和边表构成，其中点表也是真实存在的，所以遍历临接表时间是O(n+e)，这和平时写代码忽略点表不一样。
* 判断是否是深度优先序列，容易漏掉：退栈后应该从上一个节点继续遍历 这个条件

## 图的表示以及优劣

设图G的点数是V，边数是E

### 邻接矩阵

用一个宽度是V的矩阵表示图,如果图的点编号从1开始，为了方便，也可以宽度是V+1，int map[V+1][V+1]

map[u][v]=val;表示点u到点v存在一条有向边，权值是val,

初始化map数组是-1，这个人为定义。map[u][v]=-1表示不存在

**一般邻接表行号表示u,列号表示v,边是u->v**

**优点**

* 优点是已知两个点编号u和v直接可以知道权值，访问任何边是o(1)

**缺点**

* 空间复杂度高，是o(v^2)，且储存不是特别稠密的图会有很多浪费，只适用于V较小的题目。
* 遍历某个点到其他边时候，会有浪费，因为遍历数组某一行或者列，势必会访问到那些无意义的不存在的边
* 如果想得到每个点都能被哪个点到达，可以建立逆邻解表

### 邻接表

struct Edge{

Int b,e,val;//b是边的起点，e是终点，val是权值

}

list< Edge>adList[V+1]

adList[i]表示以i为起点的边所形成的边集，list表示链表，动态添加删除边，可以根据删除插入的频繁程度来选取用c++STL的list和vector。

访问以u为起点，v为终点的边，需沿着起点遍历adList[u][i],i代表边集第几个；

因此要迭代到adList[u][i].e=v

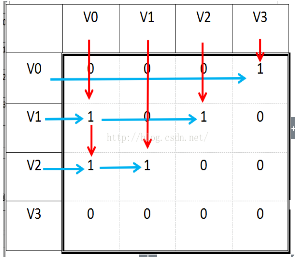
**优点**

* 不浪费内存，有多少边存多少边，就算V是100000也可以用
* 遍历以u为起点的所有边，这个操作十分方便

**缺点**

* 实现寻找反向边的功能不方便，需要在输入时候预处理，且Edge增加一个属性k,表示以b为终点的边是adList[e][k]，
* 寻找一u为起点的某条边的平均时间复杂度是o(E/V),，比邻接矩阵O(1)慢

### 十字链表



十字链表就是邻接表和逆邻接表的合体，也是邻接矩阵的压缩存储。

它优缺点同邻接表。

构建方法是：插入一条边(u-v)，先new出一个edge,对u和v所在链表各向前增加一个指向edge的指针。

### 邻接多重表

专门用来存储无向图的，它还是邻接表的改版，由于无向图用邻接表存储的情况下，每条边需要存2次，虽然内存只多一倍，但对于删除边和修改边的操作还有找反向，十分麻烦。所以邻接表的相同边有一个节点存储，就叫邻接多重表。

它的构建与十字链表一样，只不过意义不同 。

### 边集数组

struct Edge{

Int b,e,val;//b是边的起点，e是终点，val是权值

}

edgeSet[E+1];

edgeSet[i]表示编号为i的边，

优点只是代码少一点，且排序方便，适合对边集排序的情况。寻找以u为起点的某条边需要顺序查找，时间复杂度是o(E),效率极低。

### 前向星

struct Edge{

Int b,e,val;//b是边的起点，e是终点，val是权值

}

edgeSet[E+1];head[V+1]

edgeSet[i]表示编号为i的边，

对edgeSet边集按照 先b后e 的顺序排序

head[u]表示点u为起点的第一条边的下标，这样能实现类似邻接表的效果

，但它需要个E\*logE的排序，支持索引，不支持动态添加和删除，是链式前向星的前身。

### 链式前向星

struct Edge{

Int e,i ,val;// e是终点，下一条边所在edgeSet 数组中的位置，val是权值

}

edgeSet[E+1];head[V+1]

edgeSet[i]表示编号为i的边，

head[u]表示点u为起点的最后一条边的下标，

* 链式前向星实际就是顺序表实现的临接表，有邻接表的全部功能，实际效率高于动态数组vector实现邻接表，和list实现的临接表相当。
* 对于不需要删除边的图，实现查找反向边的功能很容易，用亦或1的操作，而用邻接表实现要储存额外信息。所以链式前向星在网络流问题的实现上有独特的优势。

## 最小生成树问题

最小生成树算法都需要考虑重复边，不用考虑自环，且须是联通图。

### Prim算法：

时间复杂度和边数没关系，因此稠密图最好

这里记顶点数v，边数e

|  |  |  |
| --- | --- | --- |
|  | 邻接矩阵 | 邻接表 |
| 无优化 | O(v2) | O(v2) |
| 普通堆优化 | O(elog2v) | O(elog2v) |

基本思路

步骤：

图节点数目为N,正在构造的生成树为T,

•维护Dist数组,Dist[i]表示Vi到T的“距离”

•开始所有Dist[i] = 无穷大, T 为空集

若|T| = N，最小生成树完成。否则取Dist[i]最小的不在T中的点Vi, 将其加入T

更新所有与Vi有边相连且不在T中的点Vj的Dist值： Dist[j] = min(Dist[j],W(Vi,Vj)) ，回到步骤1

**堆优化**：用堆维护边，初始条件下，只把初始点的边全部入堆，之后逐个取出，对于每条边，查看它的终止点是否在集合T中，如不在，则把这个终止点的所有其它不在T中边放入堆。

### Kruskal算法

1.用边集数组存边，把所有边按照权值从小到大排序，

2.调用并查集初始化所有点各自是一个集合，遍历边集数组，如果某边的点不在同一集合就合并它们，同时记录在最小生成树里。

时间复杂度E\*loge

## 经典最短路问题

以下E代表边数，V代表点数,dist[i]代表原点到编号为i的点的最短路，定义**松弛操作**:

给两个点u和v,dist[v]=min(dist[u],dis[u,v]); dis[u,v]代表点u到v的边的长度。

### Dijkstra算法：

**算法思路：**

给的一个图，把点分为两部分，已经确定了最短路的点(用集合X表示)，和没有确定最短路的点(用集合Y表示)，初始dist[s]=0。X只有原点s

在Y集合选取所有与X集合有边相连的点中，dist值最小的点u，对u周围的点全都进行**松弛操作,**之后江这个点u加入X集合，这里保证能确定dist[u]一定是s到u的最短路，因为我们是选取Y集合dist值最小的点，Y集其他点dist一定大于u，意味着你怎么松弛，也不能使dist[u]更小。

反复进行上述步骤V次，就能找到所有点最短路。若果暴力的找Y集合dist值最小的点，时间复杂度是o(V^2+E) ,堆优化是o(ElgV) ，斐波那契堆是o(VlogV+E)

**代码思路：**

使用优先队列(堆)优化时间复杂度ElgV，单原点算法，不能有负数价值边。

数组dist表示源点到各个顶点的最短距离，堆que表示需要以此为中转点更新其他店的点的集合，堆每个节点有2个值，点编号i,和到原点距离dis

用邻接表或者链式前向星存图的数据。设原点是s，初始:dist除了 （dis[s] = 0）。其余全是INF，类似广度优先，把原点入堆，每次弹出堆顶元素,把它叫now，遍历当前点now的所有可到达点next，若dist[now]+w（now,next）小于dist[next]本来值，就更新并且入队等待下次遍历。

迪杰斯特拉算法有个大优化就是：对于每次的当前点now， 若now.dis> dist[now],那么说明now.dis代表的路径肯定不是最短的路，直接跳过这层搜索即可。由于now.dis不可能小于dist[now]，有些地方写成if(now.dis!=dist[now.i]) continue;

* 存在重边怎么办？

在邻接表建立的时候(一般是输入时)，就过滤掉，取最小的

* 有无法到达的点怎么办？

不需要考虑，有的话dist[i]会等于INF

* 对于有向图也适用
* 对于含有负边的图不适用此算法
* 自环需要考虑，输入时候排除掉
* 迪杰斯特拉也可以像bf或者spfa那样记录入队次数，判断图的环路，但是图不能有负边，会导致错误。

### 弗洛伊德算法：

设dis[i][j]代表点i到j的最短路，dis[i][j]==INF表示i到j不存在路

逐对的枚举两个点，称之为a和b,之后再枚举点,称之为c。

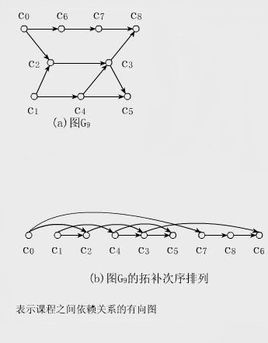
每次进行松弛操作：dis[a][b]=min(dis[a][b],dist[a][c]+ dist[c][b])这里若dist[a][c]或dist[c][b]是INF，说明不存在路，当然就不进行松弛操作

弗洛伊德算法是三重循环嵌套，时间复杂度是V^3。它时间复杂度高一般只是适用V<=100的情况，但是优势是能求出任意两点最短。且时间常数小，比进行V次Dijkstra强

## 拓扑排序与关键路径：

### 概念

给你一个DAG(有向无环图) 进行拓扑排序，是将G中所有顶点排成一个线性序列，使得图中任意一对顶点u和v，若边(u,v)∈E(G)，则u在线性序列中出现在v之前。通常，这样的线性序列称为满足拓扑次序(Topological Order)的序列，简称**拓扑序列**。简单的说，由某个集合上的一个[偏序](https://baike.baidu.com/item/%E5%81%8F%E5%BA%8F/2439087)得到该集合上的一个[全序](https://baike.baidu.com/item/%E5%85%A8%E5%BA%8F/10577699)，这个操作称之为拓扑排序。 拓扑排序是对点来说的，输出是一组拓扑序的点。



**AOV网：**是顶点表示活动(点权代表活动时间)，用弧表示活动之间的关系的DAG。注意AOV网边没有权，它仅仅用方向来代表关系。

**AOE网：**是顶点表示事件(活动之间的关系)，用弧表示活动(边权代表活动时间)的DAG。注意AOE网是带边权的，但没有点权//”点表事件，边表活动，活动有权“

在AOE网里，如果只有：1个入度为0的点(源点S)和1个出度为0的点(汇点T)

定义S到T的最长路径为**关键路径**，关键路径上的活动(边)叫**关键活动。**

**机动时间：**等待的时间，

**关键路径的实际意义**：整个工程完成需要的时间是最长路径长度，而关键路径上的活动时间(边权)会影响工程进度(增大或减小总时间)

定义如下概念：

|  |  |  |
| --- | --- | --- |
| 事件(顶点) | 最早发生时间ve(u) | 最晚发生时间vl(u) |
| 活动(边) w(u,v) | 最早开始时间e(u,v) | 最晚开始时间l(u,v) |

要想求得边的l(u,v)与e(u,v)，需要先求点的ve(u)与vl(u).

先对AOE网拓扑排序，再进行操作。

**求ve:** ve(S)=0, ve(u)=max{ve(v)+prew(u,v)}

表述：原点的最早发生时间是0，其他点的ve,是和自己前驱点的边权，取最大。

**求vl:** vl(T)=ve(T), vl (u)=min{ vl (v)+nextw(u,v)}

表述：汇点最晚发生时间和ve相同，其他点的vl,是和自己后继点的边权，取最小。

**求e(u,v):** e(u,v)=ve(u) //一条边的最早开始时间，是有向边起点的ve值

**求l(u,v) :** l(u,v)=vl(v)-w(u,v) //一条边的最晚开始时间，是有向边末点的ve值 减去 边权。

记忆方法

“ve找前驱加边权最大,vl找后继减边权最小,

e找入点ve,l找出点vl减边权,先ve再vl再e再l”

* 对于一个活动(边)，完成它的**剩余时间**是：l(u,v)-e(u,v)， 特别的，对于所有关键活动l(u,v)-e(u,v)==0，即：e(u,v)==l(u,v)。
* **工程总时间**就是ve(T)
* **关键路径**是最长路径，用拓扑排序求ve,vl,e,l时，可以利用e(u,v)==l(u,v)确定点是否属于关键路径的点e(u,v)=l(u,v)。
* 如果有多条关键路径，必须减小某些边权(可能1个也可能多个)，使得每个路径长度都减少，才能提高总体效率，否则不行。

如果有多条关键路径，只要增加某一个关键路径上边的活动时间(边权)，总体时间就会变长。

* “关键路径不可以有机动时间”，这个提法正确，因为关键路径本来就不能等待，要不就不叫关键了。
* 如果一有向图具有有序的拓扑序列，邻接矩阵一定是三角矩阵。

### Kahn算法:

**普通拓扑排序**

显然拓扑排序的顺序是每次取入度是0的点，然后删除与其相邻的边。

Kahn的算法的思路其实就是，手动模拟的拓扑排序，我们先使用一个栈保存入度为0 的顶点，然后输出栈顶元素并且将和栈顶元素有关的边删除(代码中不必真的删除边，而是改变其他点入度)，减少和栈顶元素有关的顶点的入度数量并且把入度减少到0的顶点也入栈。

**关键路径**

找关键路径和求l(u,v)和vl(u)都是要从后往前找的，所以要进行正向反向拓扑排序各一次。正向拓扑排序如前面介绍，反向拓扑排序就是记录每个点出度，和正向类似，只是从出度为0的点开始入栈，每次”删去”前驱边即可。

至于如何求关键路径，如果学过迪杰斯特拉算法就很容易想到了.就是维护数组dist[]和next[]，dist[u]表示：点u到达汇点T的最长路径的长度。next[u]==0表示这个节点不是关键路径的节点，next[u]>0表示下一个节点的编号。

### Dfs搜索法:

由于图是有向无环的，显然可以以每个入度是0的点开始dfs,搜到一个点如果他入度是0，和 Kahn很像就删除边(代码中不必真的删除边，而是改变其他点入度) ，dfs如果判断环路要比Kahn麻烦。但是dfs寻找关键路径和相关属性(e,l,ve,vl)的代码比kahn简单，此时考虑dfs法。

* 拓扑排序一定要考虑重边自环。因为这会影响度数。

考研图拓扑排序大题

# 查找

## 概念和顺序表查找

**查找表**：同一类型数据构成的集合

**静态查找(表/数据结构)**：不带有插入，删除，修改等改变原本内容的操作

**动态态查找(表/数据结构)**：带有插入，删除，修改等改变原本内容的操作

**查找不成功：**表中不存在给定关键字，查找到了空地址。

**平均查找长度：**ASL=

**查找树：**把每次查找的每种可能的情况画成一棵树。

**顺序查找：**

显然，查找树是一个链子

查找成功时平均查找长度ASL= , 查找失败时平均查找长度ASL= ,

当每次查找成功与查找不成功概率相同(一半概率时，ASL=

**二分查找：**

必须有序表才能进行，显然，查找树不是**完全二叉树(严蔚敏)**，也不一定是满的。

但是

小数据查找成功时，平均查找长度ASL=

大数据查找成功时，平均查找长度ASL=

* 二分查找失败的平均查找长度通过查找树的空叶子节点来手算。二分查找失败平均查找长度的计算，不需要计入空节点本身的高度
* 考研填空选择中的二分查找是以[1,n]作为数组下标的。
* 折半查找的二叉判定树一定是平衡二叉树
* 折半查找的二叉判定树计算下标时，要么向下取整要么向上取整只能选一种，但不能通过查找树中度数为1的节点是否都是左/右孩子来判定，应该带入数据去试验。

具体题目是王道查找7.1习题15

**分块查找(索引顺序表)：**

把长度是n的表分成b块，每块之间有序，而块内无序，所以结合顺序和二分查找进行，最佳分块方法是令b=sqrt(n),每块元素个数为s

顺序查找每块，平均查找长度ASL=

二分查找每块，平均查找长度ASL=

* 注意考研中提法：对线性表要求高效率又适应动态变化，可采取分块查找
* 分块查找的平均查找长度计算时，访问每块本身的指针也算一次查找

**静态树表查找**

## 普通BST树

二叉查找树（Binary Search Tree），（又：[二叉搜索树](https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%90%9C%E7%B4%A2%E6%A0%91)，二叉排序树）它或者是一棵空树，或者是具有下列性质的[二叉树](https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%A0%91)： 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结点的值； 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值； 它的左、右子树也分别为[二叉排序树](https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%8E%92%E5%BA%8F%E6%A0%91)。

* 每个点的值d都不同，因此若插入时候某个点的值在树中存在，则插入失败
* 左边的d一定小于当前点，右边点一定大于当前点
* 中序遍历可得到有序序列

### 查找算法：

查找算法基础，插入删除也会用到，查找值是d0的点如何查？

设当前点now是根节点head,如果：

当前节点小于d0,向右找，

当前节点大于d0,向左找，

当前节点等于d0,说明找到了，

当前节点是空，穷途末路，说明不存在

### 插入结点的算法：

设当前点now是根节点head, 如果：

当前节点小于d0,向右找，

当前节点大于d0,向左找，

当前节点等于d0,根据BST树性质，不能有同样节点，插入错误

当前节点是空，可以插入，那就插入吧

### 删除结点的算法：

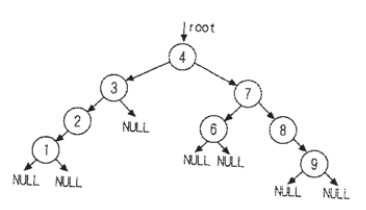
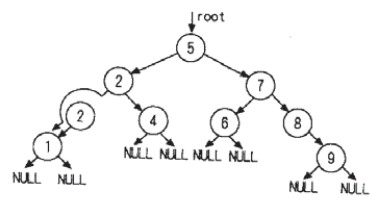
**情况1：**

对于左边或者右边有一个空的节点(或者左右都空)，只要直接删除再把下面连上即可

**情况2：**

如果两边都有节点：

寻找要删除的节点node的所有子节点中比它大一点(左子树最大值)，或者比它小一点的(右子树最小值)，和node交换，再删除那个节点(那个最值点一定是左右有一个是空的节点，也就是一定能变成情况1)。如下，删除根节点5



比如找比它小的点，只需要遍历node->left的每个最右的节点走，直到走不了。

## AVL平衡树

是平衡的BST树，插入删除节点时，一旦发现不平衡了马上旋转操作让他平衡。要有变量记录平衡因子,AVL树插入删除原则是，插删后再旋转。

### 查找算法：

同BST树

### 插入结点的算法：

先同普通bst树那样插入，在插入完成后,改变平衡因子，如果发现不平衡，要经过多次旋转使树平衡，这个旋转时间是log(n)级别的，总复杂度是logn(logn的旋转操作)，AVL树的插入删除后调整规则是一样的，后面说

### 删除结点的算法：

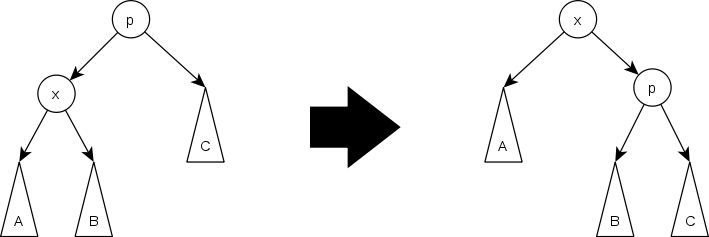
总体和BST一样；但是由于删除使树不平衡，又要转，在被删除的节点开始递归回溯的旋转树，类似插入操作。

总复杂度是logn (logn的旋转操作)

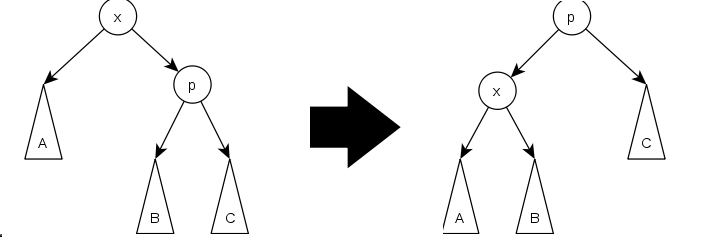
AVL树的插入删除后调整规则是一样的，后面说

### 调整算法：

定义一种基础操作，rot(now,fp)代表以now为支点向fp方向旋转，fp有2种取值，fp==1代表向右旋转，fp==0代表向左旋转：如下所示：



上图是rot(P,1)



上图是rot(X,0)

对于每个节点记录高度h,h代表从子树的最深叶子节点到这个节点的最短路径上的节点数。

不管插入还是删除算法，是一样的调整思路，都是从(插入/删除)的节点起，这是个子下而上的递归的过程，一直检查到根，对于这条路径上的每个节点，发现某个点左右不平衡，就要旋转调整平衡了。所谓” 左右不平衡”,指的是平衡度绝对值大于1

设平衡度difh=左子节点的h - 左子节点的h

平衡度不需要单独记录，通过h计算即可

对于某个点now，检查它的abs(difh)是否大于1，如果abs(difh)>1：看now的较高的儿子的节点,设它是s1

看s1的较高的儿子的节点,设它是s2

设s1是now的fp方向的儿子

分为如下两种情况：

**s1和s2相对于它们的父亲同方向**

则以s1的父亲(就是节点now)为支点向fp^1方向旋转,旋转后不要忘了刷新now和s1的h属性

**s1和s2相对于它们的父亲异方向**

则以s2的父亲(就是节点s1)为支点向fp方向旋转,旋转后不要忘了刷新s1和s2的h属性，之后一定变成了情况1，继续执行。

## 哈希表实现和哈希冲突

### 概念

所谓哈希，就是给定一个数据key，对它编码成一个数字，然后再通过这个数字，做到可以索引给定的另一个数据value.这就是哈希的<key,value>模型，key一定是可哈希对象(字符串，浮点数，整数)，value不限。

储存这些数据的数据结构叫**哈希表**或者**散列表**，应该用支持随机索引的数据结构来存储，那就是**顺序表**。实际应用当中，会在此基础上优化。

对key编码的过程需要哈希函数，设它为int hash(key)，哈希函数在后面说，它有多种实现方法，哈希函数是函数，是一对一的关系，一个key只能对应一个value,且每个hash(key)的数值一定不同，否则就会出现哈希冲突，哈希函数设计的好坏的标准就是产生冲突的几率大小，处理哈希冲突在下面会详细介绍。

设n代表已经有n个位置被占据，len代表哈希表长度,L(x)代表查找x所需处理冲突的次数加上1

**装填因子：** α=n/len。

平均查找长度分为查找成功的平均查找长度和查找失败的平均查找长度

**成功的平均查找长度：**

**失败的平均查找长度：**

**//**表示哈希表数组下标为i的位置开始，查找元素直到查找失败所需的次数，其中查找失败指按照给定处理冲突的方法找，找到空位置都没找到。

常见哈希处理冲突方法的平均查找长度表如下。

|  |  |  |
| --- | --- | --- |
| 处理冲突方法 | 成功时 | 失败时 |
| 线性探测 | ≈ | ≈ |
| 随机，二次探测，多哈希 | ≈ | ≈ |
| 拉链法 | ≈ | ≈ |

* 实际中，当装填因子到达设置好的值(视情况)，为了提高查找效率就要对哈希表**扩容**：开辟新的更大的一段空间，把原来数据拷贝过去，新表比原本的表大多少也要视情况而定。扩容复杂度o(len),所以扩容不能太频繁。
* 拉链法的查找不成功的查找长度依然是除以表长(一般是mod的数值)，从每个位置下查找失败的长度等于其元素个数,不能加1
* 平均查找长度与n和表长都无关(尽管在具体的公式里有)，而是和哈希函数与处理冲突的方法与装填因子有关，且影响作用递减：

哈希函数：是影响平均查找长度的首要因素，但一般题目假设哈希函数都是均匀，且足够好的

处理冲突的方法：其次，同样的哈希函数，平均查找长度取决于处理冲突方法

装填因子：再其次，同样的哈希函数和处理冲突的方法，平均查找长度取决于装填因子

装填因子越大，说明哈希表快满了，越容易冲突，平均查找长度也就越大。

我们当然希望平均查找长度越小越好，这样效率高，用政治的话说，提高平均查找长度的**根本途径**是用优秀的哈希函数。

* 散列表堆积现象是同义词和非同义词发生冲突的结果
* “堆积”现象指的是在哈希表中堆积，而拉链法单独放在一起是不算做堆积的。
* 问法:”堆积现象主要原因”是处理冲突的方法。而法:”产生冲突的根本原因”是哈希函数

### 开放地址法

所谓开放地址法，就是从发生冲突的那个单元起，按照一定的次序，从哈希表中找到一个空闲的单元，然后把发生冲突的元素存入到该单元的一种方法。这个数组大小也决定了这个哈希表最多能装多少元素，假设数组是d[]长度是len。

对于某个哈希函数hash()，如果发生冲突,也就是hash(data1)和之前的某个数据的hash相同。就是要处理冲突，开放地址法分以下几种：

有些地方把一下方法的名称后面加”再散列”，如“线性探测再散列”，因为再散列法是：线性探测补偿探测随机探测二次探测等的总成。

设⊕为某认为定义的合适的运算符号

**线性探测**

循环去看下一个位置或者上一个位置是否被占用，直到发现没有被占用的位置为止。特别的：如果正着走到了d[len-1]要从d[0]探测，表现为代码是i=(i+1)%len；逆向走到d[0]要从d[len-1]开始探测，表现为代码是i=(i-1+len)%len。

广义的线性探测表示为：hash(key)⊕k ,k=1,2,3,4…

显然这样效率不高，出现数据”堆积”现象时，退化成o(len)的时间处理冲突。

**补偿线性探测法**

将线性探测的步长从 1 改为 q ，表现为代码是i＝(i＋1) % len 改为：i＝(i＋q) % len，而且要求 len 与 q是互质的，以便能探测到哈希表中的所以位置

广义的补偿线性探测表示为：hash(key)⊕qk ,k=1,2,3,4…

**随机探测法**

将线性探测的步长从 1 改为 随机数randq ，表现为代码是i＝(i＋1) % len 改为：i＝(i＋randq) % len， randq可以每次随机生成，也可以事先生成随机序列，每次冲突就从这里面取。

广义的随机探测表示为：hash(key)⊕randq

**平方探测法(二次探测)**

将线性探测的步长从 1 改为 是：1，-1，4，-4，9，-9…..

表现为代码是i＝(i＋1) % len 改为：i＝(i＋q2) % len,

q2代表步长,上限是tableSize /2，一般len==tableSize,所以上限取到即可

广义的二次探测表示为：hash(key)⊕k ,k=1,-1,4,-4,9,-9….

这种方法不一定能探测到哈希表所有位置，但是可以探测到大部分位置，这就足够了。但根据二次剩余理论，len是4k+3的素数时，可以探测全部位置。

### 多哈希法

多哈希法是建立很多哈希函数，一个哈希函数冲突了用另一个计算，这种方法计算耗时，但可以保证几乎不会冲突。

### 拉链法(链地址法)

这种方法里，不再用普通数组当做哈希表，而是用链表数组当做哈希表，假设数组是d[]长度是len。这种方法解决哈希冲突比较容易，对于某个哈希函数hash()，如果发生冲突,也就是hash(data1)和之前的某个数据的hash相同。直接在对应hash(data1) 位置的链表里添加一个位置，放进data1即可。

这样解决哈希冲突复杂度是o(1)，但缺点是在查找这个元素时，要用顺序查找逐个遍历，假如哈希函数垃圾冲突太多，还是慢，但如果哈希函数好的话，这个方法很快，是java哈希表内部处理冲突的方法。

* 拉链法的装填因子有可能大于1，因为在哈希函数极烂的情况下，使得某位置的链表变得很长，以至于比链表数组总长度len还大。
* 拉链法的查找不成功的查找长度依然是除以表长(一般是mod的数值)，从每个位置下查找失败的长度等于其元素个数,不能加1

故拉链法失败的平均查找长度是：数组链表总节点数除以表长len

### 建立公共溢出区

将哈希表分为公共表和溢出表，当溢出发生时，将所有溢出数据统一放到溢出区

## B树

B树也叫B-树或是平衡多路查找树，这几个名字说的是同一个东西

B树是一颗平衡树，它不会像普通BST树那样变成一条链子

B树不是二叉树而是多叉树，一个节点可以有很多个儿子，所以它相当于是多叉平衡查找树

### 性质

对于一个m(m>=3)阶B树有如下性质

1. 每个节点最多m-1个关键字，最多m个儿子，可以说几阶B树就最多几个儿子
2. 每个非根非叶子节点至少ceil(m/2)个儿子和ceil(m/2)-1个关键字，根节点至少2个儿子。”叶子”节点关键字数也要至少ceil(m/2)-1个,实际上定义中B树的真正叶子是空节点不存任何信息
3. 所以的NULL节点都在同一层
4. 有j个儿子的节点有j-1个关键字，关键码按照递增排序
5. 满足搜索树性质，关键字keylist[i]和keylist[i+1]之间的节点以及它的所有子节点的数都介于区间(keylist[i],keylist[i+1])

keylist[0]之前的节点以及它的所有子节点的数都介于区间(-oo,keylist[0])

keylist[sz-1]之后的节点以及它的所有子节点的数都介于区间(keylist[sz-1],+oo)

* 在考研里：B树的”叶子节点”指的是空节点，也就是空指针。

B树也可以像红黑树等其他平衡树那样，在每个节点储存<key,value>对key建树，用来索引对应value,在这片文章中，为了方便看，B树每个节点省略了value

还有是，不论B树还是红黑树，有些地方把空节点看成叶子节点(因为红黑树里空节点算黑色节点)，但我习惯于把空节点就叫空节点，叶子结点指最底下的节点 。

**其他性质**

* B树的非根非叶子节点至少有ceil(m/2)个子树可得到：第一次1个节点，第二层最少2个节点,第h层：h>=3时，范围如下

至少2×个节点，叶子节点范围：[2×,]

* B树节点总数通过等差数列求和，设r=易知，总数∈[,]
* 非空B树每个节点关键字个数范围：

根节点,叶子节点:[1,m-1], 其他节点:[ceil(m/2)-1,m-1]

* N个节点的m阶B树一定有n+1个”叶子”节点。因为此”叶子”指的是空节点，也就是空指针

### 节点数据

keylist[] //

vallist[] //

son[] //

p //父节点

为了方便，sz表示节点keylist的元素个数，由B树性质，sz+1∈[m/2,m]

### 查找算法

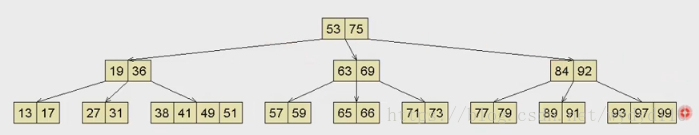
方法是从从根节点开始，每遇到一个节点，就遍历它的keylist, 若找到就返回true

但若带查找的k出现：

* k<keylist[0],说明要找的数据可能存在于son[0]里，往son[0]走继续找
* keylist[i]<k<keylist[i+1],说明要找的数据可能存在于son[i+1]里，往son[i+1]走继续找
* k>keylist[sz-1],说明要找的数据可能存在于son[sz]里，往son[sz]走继续找

可见B数的keylist元素永远是升序的

按照上述步骤，找到叶子也没找到，说明这个元素不存在，再插入算法里，这个位置就是要插入的位置。



如果查找75

先对根节点进行一次顺序查找，正确命中75。

如果查找69

先对根节点进行一次顺序查找，这次查找以失败终止于介乎53和75之间的这个引用。对63和69所在的这个节点进行一次顺序查找，成功找到69。

如果查找45

 先对根节点进行一次顺序查找，这次查找以失败终止53左侧的引用。将19和36这个节点通过IO载入内存，并做一次顺序查找，这次查找以失败终止于36右侧的引用，......最终查找失败于41与49之间的引用，底下是节点了，说明查找

### 插入算法

B树插入算法，首先初始化一个root,初始里面没有数据，对于每个数据，需要确认插入的位置，通过查找算法找到要插入的叶子结点的相应位置，之后只需记住核心思想是”分裂”：

1. 直接插入在相应位置，如果该节点<m，则什么都不做，插入结束。否则进入下一步
2. 此时节点关键字个数超过了m-1，要”分裂”：把keylist最中间的元素keymid拿出来，按照：keymid左边的元素，keymid本身，keymid右边的元素。分成3部分，把keymid本身插入到父亲节点里，放到合适的位置，把当前节点按第1和3部分分开成2个节点，放在父节点的keymid两侧。

如果当前节点是root怎么办，它没有父亲，那就创造个父亲，令它是root.

插入k完成后，如果有value应该把value也插入相应位置

1. 步骤二进行之后，可能父节点也不满足B树性质，即关键字个数超过了m-1,并且叉数也超过了m，应该把它在分裂。所以B树插入是个自下而上的递归过程,直到分裂完root为止：

for(now=叶子;now!=NULL&&now-> keylist.size()<m;now=now->parent){}

<https://www.01hai.com/note/av124483>

### 查找前驱后继

前驱：比某个数小的最大的数；

查找前驱：和BST树类似，对于查找node中的某个数keylist[i]的前驱，先向

keylist[i-1]到keylist[i]之间的分叉走(如果是keylist[0]直接走son[0])，也就是沿着son[i]走一个节点，然后一直沿着son[sz]走，直到走到叶子节点，这个节点的keylist[sz-1]就是前驱。

后继:比某个数大的最小的数

查找前驱：和BST树类似，对于查找node中的某个数keylist[i]的后继，先向

keylist[i]到keylist[i+1]之间的分叉走(如果是keylist[sz-1]直接走son[sz])，也就是沿着son[i+1]走一个节点，然后一直沿着son[0]走，直到走到叶子节点，这个节点的keylist[0]就是后继。

B树的NULL节点都在同一层，也就是叶子节点下面，且如果son[sz]存在，则son[0]，son[1]…..son[sz]全都存在，如果son[sz]不存在，则son[0]，son[1]…..son[sz]全都不存在，不会出现像bst树那种，前驱后继不是叶子节点的情况。

### 删除算法

删除操作类似BST树，要先看是不是在叶子，如果不在叶子，找到这个元素的后继或者前驱，替换它，而后继或者前驱一定在叶子节点。

那么问题变成了如何删除叶子的某个值，定义里说B树节点的值个数不能小于m/2，所以删除节点可能会破坏这个性质。步骤是

1. 直接删除，如果keylist.size()>=m/2什么都不做，否则继续下面步骤
2. 此时keylist.size()<m/2，首先试图从兄弟那里索取一个数，(左兄弟的keylist[sz-1]或者右兄弟的keylist[0])。

兄弟节点的keylist.size()必须满足keylist.size()>=m/2+1,因为得要完了也满足B树性质才可以借。如果可以索取，也不是直接拿过来，因为直接拿不满足搜索树的性质(B树这两个节点应该，左边的所有数都要比右边的小)

假设被拿的数是a，这两个节点的父亲的夹值是keymid，先把keymid放到当前缺少元素的节点，再把a放到原本keymid的位置

此后B树满足性质，删除的调整操作结束。

1. 若不能从兄弟那里索取一个数，此时说明兄弟节点的keylist.size()==m/2,而且自己本身的keylist.size()<m/2 ，应合并自己和兄弟节点,同时把这两个节点的父亲的夹值是keymid放到合并后的节点里。但此时由于从父亲拿了一个数，父亲可能出现keylist.size()<m/2的情况，要继续检查父亲节点，并且调整，

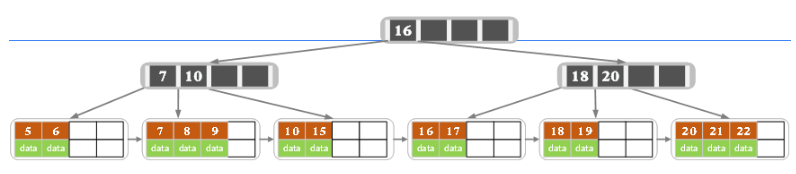
所以B树删除是个自下而上的递归过程,直到检查合格或者到root为止。

for(now=叶子;now!=root&&now-> keylist.size()>=m/2;now=now->parent){}

## B+树

B+树是B树的变体，B+树的空间消耗量比B树更大，它是为了做<key,value>这种索引而设计的，在索引的扩展性和效率上比B树好。

B+树和B树区别在于，B+树节点分为两种，叶子节点和内部节点，内部节点只储存key,不出村value,而叶子结点储存key和value，叶子结点用链表连起来



### 性质

对于一个m(m>=3)阶B+树有如下性质

1. 每个节点最多m-1个关键字，最多m个儿子，可以说几阶B+树就最多几个儿子
2. 每个非根非叶子节点至少m/2个儿子，根节点至少2个儿子。
3. 所以的NULL节点都在同一层
4. 有j个儿子的节点有j-1个关键字，关键码按照递增排序
5. 满足搜索树性质，关键字keylist[i]和keylist[i+1]之间的节点以及它的所有子节点的数都介于区间[keylist[i],keylist[i+1])

keylist[0]之前的节点以及它的所有子节点的数都介于区间(-oo,keylist[0])

keylist[sz-1]之后的节点以及它的所有子节点的数都介于区间[keylist[sz-1],+oo)

1. 叶子结点通过左到右的指针连接，除了叶子结点，其他节点只存key,不存value

* B+树由于有底层从左到右的链子，可以支持顺序查找，而B树不可以

B树和B+树都支持随机查找

### 节点数据

内部节点：

keylist[] //

son[] //

p //父节点

叶子节点：

keylist[] //

vallist[] //

next //右节点指针

p //父节点

为了方便，sz表示节点keylist的元素个数，由B+树性质，sz+1∈[m/2,m]

### 查找算法

由于B+树叶子节点储存value,所以查找数据一定要找到叶子节点。

方法是从从根节点开始，每遇到一个节点，就遍历它的keylist,

但若带查找的k出现：

* k<keylist[0],说明要找的数据可能存在于son[0]里，往son[0]走继续找
* keylist[i]<=k<keylist[i+1],说明要找的数据可能存在于son[i+1]里，往son[i+1]走继续找
* k>=keylist[sz-1],说明要找的数据可能存在于son[sz]里，往son[sz]走继续找

按照上述步骤，一直走到叶子节点，若叶子节点的keylist没找到，说明这个元素不存在，再插入算法里，这个位置就是要插入的位置。如果找到说明存在，返回true.

### 查找前驱后继

### 插入算法

B+树插入算法，首先初始化一个root，且作为叶子结点,初始里面没有数据，对于每个数据，需要确认插入的位置，通过查找算法找到要插入的叶子结点的相应位置。B+树的插入的核心思想是”分裂”：

1. 直接插入在相应位置，如果该节点<m，则什么都不做，插入结束。否则进入下一步
2. 此时节点关键字个数超过了m-1，要”分裂”：把keylist最中间的元素keymid拿出来，按照：keymid左边的元素，keymid本身，keymid右边的元素。分成3部分，把当前节点分开成2个节点，keymid左边的元素在左边的节点，keymid本身和keymid右边的元素在右边的节点，拆开后它们还是叶子节点，通过next指针相连。同时把keymid的数值也加入到父亲节点的合适位置。

如果当前节点是root怎么办，它没有父亲，那就创造个父亲，令它是root，且这里创造的节点都是内部节点。

插入k完成后，如果有value应该把value也插入相应位置

1. 步骤二进行之后，可能父节点也不满足B+树性质，即关键字个数超过了m-1，应该把它在分裂。所以B+树插入是个自下而上的递归过程,直到分裂完root为止：for(now=叶子;now!=NULL&&now-> keylist.size()<m;now=now->parent){}

<https://www.01hai.com/note/av124483>

### 删除算法

B+树所有数据都存在叶子，所以待删除节点一定是叶子

那么问题变成了如何删除叶子的某个值，定义里说B+树节点的值个数不能小于m/2，所以删除节点可能会破坏这个性质。步骤是

1. 直接删除，如果keylist.size()>=m/2什么都不做，否则继续下面步骤
2. 此时keylist.size()<m/2，首先试图从兄弟那里索取一个数

(左兄弟的keylist[sz-1]或者右兄弟的keylist[0])。

兄弟节点的keylist.size()必须满足keylist.size()>=m/2+1,因为得要完了也满足B+树性质才可以借。如果可以索取，直接把它拿来：如果拿的是左兄弟的keylist[sz-1]，就把它放在当前节点最左边；如果拿的是右兄弟的keylist[0]，就把它放在当前节点最右边；

假设被拿的数是a，这两个节点的父亲的夹值是keymid，用a替换父亲节点原本的keymid

此后B+树满足性质，删除的调整操作结束。

1. 若不能从兄弟那里索取一个数，此时说明兄弟节点的keylist.size()==m/2,而且自己本身的keylist.size()<m/2 ，应合并自己和兄弟节点,

设靠左的节点是left,靠右的是right,它们的keylist顺序拼合即可。

同时对父亲节点删除数据，left和right的夹值。由于从父亲删除了一个数，父亲可能出现keylist.size()<m/2的情况，要继续检查父亲节点，并且调整，

所以B+树删除是个自下而上的递归过程,直到检查合格或者到root为止。

for(now=叶子;now!=root&&now-> keylist.size()>=m/2;now=now->parent){}

# 内部排序

稳定：如果a原本在b前面，而a=b，排序之后a仍然在b的前面。(基于交换的排序算法默认相同关键字不交换)

* 基于比较的排序最少比较次数是

## 直接选择排序：

很简单的排序，就是两个指针i和j，迭代i从0扫描到n-1, j就用来找区间[i+1,n)的最小值，如果这个最小值比A[i]小就和A[i]交换。

时间复杂度是稳定的o(n\*n),需要o(1)的额外空间 ,由于5 8 5 2 9这种数据，存在相同元素，但第一次交换后把它换到了后面，就使得两个5相对顺序改变了，所以它不稳定

## 冒泡排序：

很简单的排序，第一次迭代i范围是[0,n-2]，每次如果A[i]>A[i+1]就交换。

第二次迭代i范围是[0,n-3]，每次如果A[i]>A[i+1]就交换。

…

一直进行上面操作，最后一次是：迭代i范围是[0,0]，有一个优化是如果一次冒泡之后没有任何数被交换了，说明这就是有序数组了，不需要继续执行程序

冒泡排序就像一系列得泡泡从小到大飘起来，但每次高度有所降低。

时间复杂度是最坏都是o(n\*n), 但由于有优化，如果数组本身就是有序的，时间复杂度最好是o(n)，空间复杂度O(1),冒泡排序在对比相同数字时不交换，所以永远不会改变相同关键字的相对顺序，是稳定的

* 对于升序排序，冒泡排序每次可以确定最大值位置，降序反之

## 直接插入排序：

很简单的排序，建立一个空的新数组B，迭代A数组每个元素，插入新数组B适当位置，但由于如果每次都插入在B数组最前面，就要把B后面的数向后移动，移位操作是o(n)复杂度。这里的插入实际可以在原数组进行，不需要B数组。

时间复杂度不稳定，最好情况o(n),平均和最坏是o(n\*n), 空间复杂度O(1)

插入排序每次循环的条件就是A[j]<A[i],所以永远不会改变相同关键字的相对顺序，是稳定的

插入排序在小规模数据下(指n是个位数),效率最好

## 堆排序：

堆排序是选择排序的优化，就是把A数组所有数放在一个堆里(堆参考另一片文章)，再一个个取出堆顶元素组成有序数组。堆取出元素要进行调整

此算法时间复杂度是稳定的o(n\*log(n)),如果单开一个数据结构存堆，空间复杂度是o(n),但若在原本给定数组上建立堆，空间复杂度就是O(1)

堆排序建堆再拿出来的过程可能会破坏稳定性

**特别说明**：

严蔚敏版和我自己写的不同，她的思路是：

HeapAdjust(int \*A,int l,int r) //表示

## 希尔排序：

是插入排序的优化，设置一个增量s=n/2;把数组分层s组，每组有n/s个数，意思如图所示。

时间复杂度是不稳定的，最好是o(n)，最坏是o(n\*sqrt(n)),平均还是o(n\*sqrt(n))，不需要额外空间

由于多次插入排序，我们知道一次插入排序是稳定的，不会改变相同元素的相对顺序，但在不同的插入排序过程中，相同的元素可能在各自的插入排序中移动，最后其稳定性就会被打乱，所以shell排序是不稳定的。

* 希尔排序的所谓”一趟”，是对s组每组都进行依次插入排序。

## 快速排序：

取数组任意一个数(严蔚敏书是取第一个)，把所有比它小的数放左边一部分, 把所有比它大的数放在右边一部分，最后要得到左右的那个分界线，之后递归的对左右部分进行上述操作。递归到每层函数只有1个元素，作为结束。这是大致思想。

对于较快得实现方式，不需要把轴心那个数移动到中间位置，而严蔚敏书中是最老套的方法，不仅代码麻烦，常数还大。

严蔚敏书思路：初始选取A[i]=A[0]，i=0，j=n-1,找左边第一个严格比A[begin]大的数a，右边第一个严格比A[begin]小的数b。然后先交换这两个数a和b，再b和A[begin]交换，之后指针i和j各偏移一次。这还不够,要一直进行多次做这个操作，直到指针i>=j为止才算结束一趟排序。

每次交换即：…A[begin]……a……b…… => …b……A[begin]……a……

以上循环是先后再前的。

此算法最好时间复杂度o(n\*logn),最坏时间复杂度o(n\*n),平均时间复杂度o(n\*logn)，需要递归的内存O(logn) 的空间

对于算法在某些极端数据下，可能退化成o(n\*n)。

可以有改进算法，每次随机取数，也可以对递归采取归并排序躲开。

* 快速排序的最差情况何时出现：对于选择轴元素策略固定来说，如果它选择了数值上居中的元素，则效率较好，选择了最大最小值，则效率低下，因为此时比较次数多。因此，总体数组若是正序或者倒序，则效率最低，且越接近正序或倒序，效率越。

## 归并排序：

很好的利用分治算法，把数组一半一半拆开，如gif所示[归并排序\归并排序.gif](归并排序/归并排序.gif)

时间复杂度是n\*log(n),最好是log(n)，需要额外o(n)的空间去储存每次归并时用的数组

* 归并排序虽然时间复杂度稳定，但比较次数仍然和初始状态有关，因为问题在合并时候。

## 计数排序：

很简单的非比较类型的排序，它只能对数字进行排序，开一个足够大小的数组叫做ind，初始化全部是0，能装下min(A)和max(A),遍历A的所有元素，把 ind[A[i]]++;

之后按下标顺序遍历ind数组，每次访问到一个不是0的ind[]元素就输出。这么做输出的序列是有序的。

这是一个空间换时间的算法，时间复杂度和空间复杂度都是o(max(A)- min(A));对于数组元素存在很大或者很小的值的情况，此算法不适用。

计数排序虽然是计数出个数，但也相当于按照顺序把数放进对应的索引里，每次取出数，是按照顺序取的(队列的先进先出)，永远不会改变相对位置，是稳定的排序

## 基数排序：

基数排序有：最高位优先MSD和最低位优先LSD 之分,MSD是从数的最左侧开始(最高数位)，LSD是从数的最右侧开始(个位)

非比较类型的排序，它只能对有基数的数据进行排序。设所有数当中，位数最长的数位为d位,进制为HEX,在数据结构书里叫rd。

以普通10进制数为例，初始化10个链表(其实就是图论里的邻接表)，首先遍历所有数，把个位是k的数放在编号为k的链表里。

然后第0个链表开始按顺序取出里面的数，放回原来数组。下一步，遍历所有数，把十位是k的数放在编号为k的链表里。 然后第0个链表开始按顺序取出里面的数，放回原来数组。….以此类推，直到把mlen位是k的数放在编号为k的链表里。

时间复杂度是o(d(n+rd))也可以说是o(nd)，一般情况下d不会很大，所以此算法很快。

基数排序可以处理有负数的情况，提前分类找出正数负数分别排序，最后再合并即可

空间上，不需要真的去存储邻接表，可以在原地址进行，但依然要存储rd个头指针，复杂度是o(n+rd)。

基数排序每次取出数，是按照顺序取的(队列的先进先出)，永远不会改变相对位置，是稳定的排序

## 排序算法性能分析

注意排序算法的稳定性和时间复杂度的稳定性不是一回事。

排序的稳定性：相同值的节点相对位置是否会发生改变。   
稳定：如果a原本在b前面，且数值上a==b，排序之后必定a仍然在b的前面。

时间复杂度 Time Complexity 用TC简写，空间复杂度Space Complexity简写SC

计数排序的N是指max(A)- min(A)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 选择 | 冒泡 | 插入 | 堆 | 希尔 |
| av(TC) | O(n^2） | O(n^2） | O(n^2） | O(nlogn) | O(n\*sqrt(n)) |
| min(TC) | O(n^2） | O(n） | O(n） | O(nlogn) | *O(n)* |
| max(TC) | O(n^2） | O(n^2） | O(n^2） | O(nlogn) | O(n\*sqrt(n)) |
| SC | O(1） | O(1） | O(1） | O(1) | O(1) |
| 稳定性 | false | Ture | Ture | false | false |
| 一趟能否确定一个最终位 | Ture | Ture | false | Ture | false |
| 比较次数与初始状态 | 无关 | 有关 | 有关 | 有关 | 有关 |
| 排序趟数与初始状态 | 无关 | 有关 | 无关 | 无关 | 无关 |
| 何为一趟 | 从头到尾找一遍最小的数 | 冒一轮泡 | 从第i个位置往前换，i=2时是第一趟 | 每次把堆顶拿到尾部并且调整堆 | 对于某个增量，排完整个数组 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 快排 | 归并 | 计数 | 基数 |  |
| av(TC) | O(n^2） | O(nlogn) | O(N) | O(d(n+rd)) |  |
| min(TC) | O(nlogn） | O(nlogn) | O(N) | O(d(n+rd)) |  |
| max(TC) | O(nlogn) | O(nlogn) | O(N) | O(d(n+rd)) |  |
| SC | O(logn) | O(n) | O(N) | O(rd) |  |
| 稳定性 | False | ture | ture | true |  |
| 一趟能否确定一个最终位 | Ture | false | false | false |  |
| 比较次数与初始状态 | 有关 | 有关 | NAN | NAN |  |
| 排序趟数与初始状态 | 有关 | 无关 | NAN | 无关 |  |
| 何为一趟 | 每两段都排序一次，算一趟 | 按顺序2个一组交换整个数组完成后算一趟。 |  | 一次分配和收集 |  |

以上算法的定性均为严蔚敏书上最原始的方法情况下的结论。

* 提法”希尔排序是插入排序”在王道书是正确的，严蔚敏书未知
* 看一个序列是否是增量为k的希尔排序得到的，要从第一个元素开始做希尔排序的比较，看是否为升序。而不能只看前两个或第一趟
* 快速排序中，先找后面，如果后面找到了，前面没找到就只交换枢轴和后面较小的
* 快速排序的在序列本身是正序或逆序时效率最慢，在一趟排序后枢轴越靠近中间时效率最快。
* 冒泡排序分为从前往后和从后往前，要看清，默认为从前往后
* 排序即有可能从小到大又有可能从大到小，不说明两种情况都要考虑
* 易错点：堆排序一次交换比较次数为2，如果比较到叶子只有一个，那就是比较一个
* 堆排序建堆时间是O(n)
* 在考研中求静态序列前k大用堆排序。
* n个元素的m路归并的趟数为
* 在考研语境下，最好时间复杂度达到O(n)的基于比较的排序只有冒泡插入，不考虑希尔。

# 外部排序

外部排序指的是大文件的排序，即待排序的记录存储在外存储器上，待排序的文件无法一次装入内存，需要在内存和外部存储器之间进行多次数据交换，以达到排序整个文件的目的。外部排序最常用的算法是**多路归并排序**，即将原文件分解成多个能够一次性装入内存的部分，分别把每一部分调入内存完成排序。然后，对已经排序的子文件进行多路归并排序。

# 严蔚敏的代码(可能考到)

# 自己的代码

## 选择冒泡插入

1. **for**(i=1;i<n;i++){//插入
2. **for**(j=i;j>0&&A[j-1]>A[j];j--)
3. swap(A[j-1],A[j]);
4. }
5. **for**(i=0;i<n;i++){//冒泡
6. **for**(j=0;j<n-i-1;j++){
7. **if**(A[j]>A[j+1])
8. swap(A[j+1],A[j]);
9. }
10. }
11. **for**(i=0;i<n-1;i++){//选择
12. mina=A[i];
13. **for**(j=i+1;j<n;j++){
14. **if**(mina>A[j]){
15. mina=A[j];
16. k=j;
17. }
18. }
19. **if**(mina<A[i])
20. swap(A[k],A[i]);
21. }

## 希尔

1. **void** \_\_shellSort(**int** \*A,**int** n,**int** b,**int** m){//直接插入排序
2. **for**(**int** i=b+m;i<n;i+=m){
3. **for**(**int** j=i;j-m>=0&&A[j-m]>A[j];j-=m){
4. swap(A[j-m],A[j]);
5. }
6. }
7. }
8. **void** shellSort(**int** \*A,**int** n){
9. **int** i,s=n/2;
10. **while**(s){
11. **for**(i=0;i<s;i++){
12. \_\_shellSort(A,n,i,s);
13. }
14. s/=2;
15. }
16. }

## 希尔排序严蔚敏

1. **void** ShellInsert(**int** A[], **int** n, **int** dk){//数组从1开始
2. **int** i,j;
3. **for**(i=dk+1;i<=n;i++){//每次
4. **if** (A[i]<A[i-dk]){//逆序就换
5. A[0] = A[i];
6. **for**(j=i-dk;j>0&&A[0]<A[j];j-=dk)////往前做增量是dk的插入排序，并且不是换而是后面赋值成前面，因为前面大
7. A[j+dk]=A[j];
8. A[j+dk]=A[0];
9. }
10. }
11. }//严蔚敏的思路与我不同，它是对于每个节点用插入排序向前插入到适当位置，
12. //而我是分别从[0,dk)开始每次加dk做插入排序，执行k次次插入排序。这是两种思路
13. **void** ShellSort(**int** A[], **int** n,vector<**int**>&delta){//delta是增量数组
14. **for**(auto dk:delta){
15. ShellInsert(A,n,dk);
16. }
17. }

## 快速排序自己//此代码不是严蔚敏代码，思路与其有区别

1. **void** quickSort(**int** \*as, **int** left, **int** right){//[b,e]之间
2. **int** i=left,j=right;
3. **int** m=as[(i+j)/2];
4. **while**(i<=j){  //有=
5. **while**(as[i]<m)  i++;
6. **while**(as[j]>m)  j--;
7. **if**(i<=j){   //有=
8. swap(as[i],as[j]);
9. i++;j--;
10. }
11. }
12. **if**(i<right)
13. quickSort(as,i,right);
14. **if**(left<j)
15. quickSort(as,left,j);
16. }

## 堆排序自己

1. **void** heapSort(**int** \*A,**int** n){//升序,数组从1开始
2. **int** i,j;
3. **for**(i=2;i<=n;i++)
4. **for**(j=i;j>1&&A[j]>A[j/2];j/=2)
5. swap(A[j],A[j/2]);
6. **for**(i=n;i>1;i--){
7. swap(A[1],A[i]);
8. **for**(j=2;j<i-1;j\*=2){
9. **if**(A[j/2]<A[j]&&A[j]>=A[j+1]){//左边大
10. swap(A[j/2],A[j]);
11. }**else** **if**(A[j/2]<A[j+1]&&A[j]<A[j+1]){//右边大
12. swap(A[j/2],A[j+1]);
13. j++;
14. }**else**{
15. **break**;
16. }
17. }
18. **if**(j==i-1&&A[j/2]<A[j]){
19. swap(A[j/2],A[j]);
20. }
21. }
22. }

## 堆排序严蔚敏

1. **void** HeapAdjust(**int** \*A,**int** s,**int** e){//调整函数，自上而下
2. **int** i,rc=A[s];//记录原本根的数字，避免被后面覆盖
3. **for**(i=2\*s;i<=e;i<<=1){
4. **if**(i<e && A[i]<A[i+1])//如果右边大就看右边
5. i++;
6. **if**(rc>=A[i])//如果根节点最大就退出循环，无需调整
7. **break**;
8. A[s]=A[i];//把根变成最大的那个
9. s=i;//因为和某个儿子交换了，故向下走一步，把那个儿子看成新的根
10. }
11. A[s]=rc;//此时的s不一定是叶子
12. }
13. **void** HeapSort(**int** \*A,**int** n){//A[0]位置空闲，从1开始
14. **for**(**int** i=n/2;i>0;i--)
15. HeapAdjust(A,i,n);//建堆，倒着调整，从n/2,如此相当于从后往前一层一层倒着调整回来
16. **for**(**int** i=n;i>1;i--){
17. swap(A[i],A[1]);
18. HeapAdjust(A,1,i-1);
19. }
20. }//严蔚敏的代码建堆和我不同，他是自上而下调整
21. //堆排序每趟大家都一样都是从顶端调整，只是写法不一样

## 归并

1. **void** mergeSort(**int** \*as, **int** left, **int** right){//[b,e]之间,t是临时数组
2. **if**(left==right)
3. **return**;
4. **int** mid=(left+right)/2;
5. mergeSort(as,left,mid);
6. mergeSort(as,mid+1,right);
7. **int** i=left,j=mid+1,k=left;
8. **while**(i<=mid&&j<=right)
9. t[k++]=as[i]<as[j]?as[i++]:as[j++];
10. **while**(i<=mid)
11. t[k++]=as[i++];
12. **while**(j<=right)
13. t[k++]=as[j++];
14. **for**(i=left;i<=right;i++)
15. as[i]=t[i];
16. }

## 迪杰斯特拉

1. **void** Dijkstra(**int** start){//类似广搜,此算法针对无负数边的图
2. **int** i,j,k;
3. Node now,next;
4. now.sett(start,0);//等于初始值
5. memset(dist,127,**sizeof** dist);
6. dist[start]=0;
7. que.push(now);
8. **while**(!que.empty()){
9. now=que.top();
10. que.pop();
11. **if**(now.dis!=dist[now.i])//大优化，勿省略
12. **continue**;
13. **for**(i=g.head[now.i];i!=-1;i=g.edge[i].k){
14. next.i=g.edge[i].e;
15. next.dis=now.dis+g.edge[i].val;
16. **if**(dist[next.i]>next.dis&&now.i!=next.i){
17. dist[next.i]=next.dis;
18. que.push(next);
19. }
20. }
21. }
22. }

## 佛洛伊德

1. **int** dis[MAX][MAX],path[MAX][MAX];//path代表是从i到j需要经过中介点是哪个
2. **void** FloydWarshall(**int** N){//dis初始化规则,i==j是0，否则是INF
3. **int** i,j,k;
4. **for**(i=1;i<=N;i++){
5. **for**(j=1;j<=N;j++){
6. dis[i][j]=adMatrix[i][j];//事实上dis数组初始就是初始的邻接矩阵
7. path[i][j]=j;
8. }
9. }
10. **for**(k=1;k<=N;k++){
11. **for**(i=1;i<=N;i++){
12. **for**(j=1;j<=N;j++){
13. **if**(dis[i][k]==INF||dis[k][j]==INF){//i到i为无穷大INF
14. }**else** **if**(dis[i][j]>dis[i][k]+dis[k][j]){
15. dis[i][j]=dis[i][k]+dis[k][j];//变成最短路径
16. path[i][j]=path[i][k];//从i到j的最短路径的中介点变成从i到k的中介点
17. }
18. }
19. }
20. }
21. }

## 普利姆

1. **bool** isTree[MAX];//当前点是否在树里，true表示在
2. **int** MST\_prim(){//前提是连通图，如果longlong就要改变最大值INF
3. **int** i,t,ans=0;
4. memset(isTree,0,**sizeof** isTree);
5. isTree[1]=1;
6. priority\_queue<Edge>que;
7. **for**(i=g.head[1];i!=-1;i=g.edge[i].k){
8. **if**(isTree[g.edge[i].e]==0)
9. que.push(g.edge[i]);
10. }
11. t=g.V-1;
12. **while**(t&&!que.empty()){
13. Edge now=que.top();
14. que.pop();
15. **if**(isTree[now.e])
16. **continue**;
17. isTree[now.e]=1;
18. ans+=now.val;
19. **for**(i=g.head[now.e];i!=-1;i=g.edge[i].k){
20. **if**(isTree[g.edge[i].e]==0)
21. que.push(g.edge[i]);
22. }
23. t--;
24. }
25. **return** ans;
26. }

## 拓扑排序的kahn

1. **int** du[MAX];//代表点的入度
2. **bool** Kahn(Graphic &g){//有向图存在环返回0
3. rg **int** i,j,k,now;
4. memset(du,0,**sizeof**(du));
5. stack<**int**>st;
6. **for**(i=1;i<=g.V;i++){
7. **for**(j=g.head[i];j!=-1;j=g.edge[j].k){
8. du[g.edge[j].e]++;
9. }
10. }
11. **for**(i=1;i<=g.V;i++){
12. **if**(du[i]==0)
13. st.push(i);
14. }
15. **while**(!st.empty()){
16. now=st.top();
17. st.pop();
18. printf("%d ",now);
19. **for**(i=g.head[now];i!=-1;i=g.edge[i].k){
20. **if**((--du[g.edge[i].e])==0){
21. st.push(g.edge[i].e);
22. }
23. }
24. }
25. **for**(i=1;i<=g.V;i++){//检测是不是无环图
26. **if**(du[i]>0){
27. **return** 0;
28. }
29. }
30. **return** 1;
31. }

## 二叉树中序遍历

1. **int** dfs(Node \*T){//遍历普通二叉树
2. stack<Node\*>st;//课本是Initstack(S);
3. Node \*p=T;
4. **while**(p||!st.empty()){//p存在，栈不是空
5. **if**(p){
6. st.push(p);//Push(S,p);
7. p=p->l;
8. }**else**{
9. p=st.top();st.pop();//课本是POP(S,p);
10. printf("%d ",p->data);
11. //if(!visit(p->data))return erro;//课本是检查数据访问是否异常，这里不检查
12. p=p->r;
13. }
14. }
15. **return** 1;
16. }
17. **int** dfsTra(Node \*T){//遍历线索的二叉树，不再需要栈
18. Node \*p=T->l;
19. **while**(p!=T){//没有返回根，T本身是NULL或遍历结束
20. **if**(p->tag==0){//树上是p->tag==Link
21. p=p->r;
22. }//if(!visit(p->data))return erro;//课本是检查数据访问是否异常，这里不检查
23. **while**(p->tag==1&&p->r!=T){
24. p=p->r;
25. //visit(p->data);课本访问数据，这里不访问
26. }
27. p=p->r;
28. }
29. **return** 1;
30. }