宏定义与常量的区别

1. 宏定义是一种预处理指令，用于在编译之前替换代码中的文本。宏定义使用 define 关键字，后面跟着宏的名称和宏的替换文本，例如 #define Pi3.14，就表示一个宏定义，此时的 Pi 是一个符号常量。
2. - 1.存储方式:常量是在编译时存储在内存中的，而宏是在预处理阶段直接替换掉相应的代码，没有存储在内存中。
   - 2.类型:常量具有具体的类型(如 int,float 等)，而宏没有具体的类型。
3. 安全性:常量比宏更安全，因为编译器可以对常量的值进行检查，例如类型不匹配、数值溢出等。而宏只是简单的文本替换，编译器无法对宏进行这种检查。
4. 效率:由于宏是预处理器进行文本替换，不占用运行时内存，所以可能比使用常量更高效。然而，这也取决于具体的代码和编译器优化。

基本概念和术语

1. 数据：数据是客观事物的符号表示，是所有能输入到计算机中并被计算机程序处理的符号总称。(比如：c 语言程序中定义的整数实数，字符串，多媒体处理的图形 img,png 等)
2. 数据元素：数据元素是数据的基本单位，在计算机中通常作为一个整体进行考虑和处理。(比如：一名学生记录)
3. 数据项：数据项是组成数据元素的，有独立含义的，不可分割的最小单位。(比如：学生记录中的学号，姓名，性别)
4. 数据对象：数据对象是性质相同的数据元素的集合，是数据的一个子集。（比如：多个学生记录的集合）

从小到大排列
数据>数据对象>数据元素>数据项

数据结构

数据结构是相互之间存在一种或多种特定关系的数据元素的集合。通常情况下，精心选择的数据结构可以带来更高的运行或者存储效率。数据结构往往同高效的检索算法和索引技术有关。

1. 数据结构包括**逻辑结构**和**存储结构**
2. 数据的**逻辑结构**是从逻辑关系上描述数据，它与数据的存储无关，是独立于计算机的。
   - 数据逻辑结构的两个要素：1. 数据元素。2. 关系：指数据元素之间的逻辑关系。
   - 数据元素之间关系的不同，通常有四种基本结构：**1. 集合结构(单个) 2.线性结构(一对一) 3.树结构(一对多) 4.图结构或网状结构(多对多)**
   - 数据逻辑结构的分类： 1. 线性结构 2. 非线性结构(集合，树，图)

3. **数据对象在计算机中的存储表示称为数据的存储结构**，也叫**物理结构**
   分类：参考地址
   i. **顺序存储结构**：把数据放在地址连续的存储单元里面
   ii. **链式存储结构**：把数据元素存放在任意的存储单元，这组存储单元可连续，也可以不连续。
   iii. 索引存储结构
   iv. 散列存储结构

数据类型和抽象数据类型(可以理解为 C 语言的 struct 对象)

1. **数据类型**：一个值的集合以及定义在这个值集上的一组操作的总称。(比如：)
2. **数据抽象类型是指由用户定义的、表示应用问题的数学模型，以及定义在这个模型上的一组操作的总称，具体包括三部分:数据对象、数据对象上关系的集合，以及对数据对象的基本操作的集合。**
   - 数据对象
   - 数据对象上关系的集合
   - 对数据对象的基本操作

算法

1. **作用：算法是为了解决某类问题而规定的一个有限长的操作序列**

一个算法必须满足以下五个重要特性：

1. **有穷性**，一个算法必须总是在执行有穷步后结束，且每一步都必须在有穷时间内完成
2. **确定性**，每种情况下执行的操作，在算法中都有确切规定，不会产生二义性，使算法的执行者或阅读者都能明白其含义以及如何执行
3. **可行性**，算法中的所有操作都可以通过已经实现的基本操作运算执行有限次来实现
4. **输入**，一个算法有零个或多个输入，当用函数描述算法时，输入往往通过形参表示，在它们被调用时，从主调函数获得输入值。
5. **输出**，一个算法有一个或多个输出

算法优劣的基本标准

1. **正确性**：在合理的数据输入下，能够在有限的运行时间内得到正确的结果
2. **可读性**：好的算法应该易于人们理解和相互交流
3. **健壮性**：好的算法能适当做出正确反映或进行相应处理，而不会产生一些莫名其妙的输出效果
4. **高效性**：好的效果执行效率高

算法的两个主要方面

1. 算法分析的两个主要方面是分析算法的时间复杂度和空间复杂度，以考察算法的时间和空间效率。一般情况下，鉴于运算空间较为充足，故**将算法的时间复杂度作为分析的重点。**

算法的时间复杂度

1. 问题规模：指算法求解问题输入量的多少，用整数 n 表示。
2. 语句频度：一条语句重复执行的次数。
3. 定义：指执行算法所需要的计算工作量，定性描述该算法的运行时间。
   时间复杂度对比：O(1) < O(log2 ^n) < O(n) < O(nlog2 ^n) < O(n^2) < O(n^3); [log2 ^n(以 2 为底，n 的对数)]
4. 评估标准：一般在没有特殊说明的情况下，都是指最坏时间复杂度。

重要含义：算法的时间复杂度不仅与问题的规模有关，还取决于待处理数据的初始状态

算法的空间复杂度

定义：指一个算法在运行过程中临时占用存储空间大小的量度

线性表

定义：n(n>=0)个数据特性相同的元素构成的有限序列称为线性表,n 代表线性表的长度，n=0 时为空表,线性表存储的地址是连续的。

非空线性表的特点:

1. 存在唯一一个被称作"第一个"的数据元素
2. 存在唯一一个被称作"最后一个"的数据元素
3. 除第一个之外，结构中的每个数据元素均只有一个前驱
4. 除最后一个之外，数据中的每个元素均有一个后继

数组

1. 定义：由类型相同的数据元素构成的有序集合，每个元素称为数组元素

栈：从 0 开始

1. 定义：栈是限定仅在表尾进行插入或删除操作的线性表。对栈来说，表尾端称为**栈顶**，表头端称为**栈底**，不含元素的空表称为**空栈**。
2. 特点：后进先出(Last In Fisrt Out,LIFO);
3. 操作叫法：栈的插入叫进栈/压栈/入栈，栈的删除操作叫出栈，有的也叫弹栈。

队列：从 0 开始

1. 定义：**队列是一种先进先出(First In First Out, FIFO)的线性表。**只允许在表的一端进行插入，而在另一端删除。在队列中，允许插入的一端称为**队尾**，允许删除的的一段称为**队头**
2. 队列顺序存储的不足：1. **每次操作队列元素都得向前移动**，此时时间复杂度为 O(n)。 2. 由于地址连续，很容易因数组越界而导致程序的非法操作错误，称为**假溢出**现象。为解决这种问题，需要使用**循环队列**
3. **假溢出**：因数组越界导致程序的非法错误。

查找

1. **查找表**：由同一类型的数据元素构成的集合。比如：线性表，树表，散列表等
2. **关键字**：关键字是数据元素(或记录)中某个数据项的值，用它可以标识一个数据元素(或记录)。若关键字可以唯一地标识一个记录，则称此关键字为**主关键字**，比如某个字段的 id。反之，以识别若干记录的关键字为**次关键字**，比如某张表表的涨跌幅(可重复)。
3. **查找**：查找是指根据给定的某个值，在查找表中确定一个其关键字等于给定值的记录或数据元素。若表中存在这样的一个记录，则称**查找成功**;若表中不存在关键字等于给定值的记录，则称**查找不成功**，此时查找的结果可给出一个“空”记录或“空”指针。
4. **动态查找表和静态查找表**：若在查找的同时对表做修改操作(如插入和删除),则相应的表称之为**动态查找表**，否则称之为**静态查找表**。换句话说，动态查找表的表结构本身是在查找过程中动态生成的，即在创建表时，对于给定值，若表中存在其关键字等于给定值的记录，则查找成功返回;否则插人关键字等于给定值的记录。
5. **平均查找长度**：为确定记录在查找表中的位置，需和给定值进行比较的关键字个数的期望值，称为查找算法在查找成功时的平均查找长度

排序

1. **排序：排序是按关键字的非递减或非递增顺序对一组记录重新进行排列的操作。**或将无序序列排成一个有序序列的运算。
2. 内部排序和外部排序：
   - **内部排序**：指的是待排序记录全部存放在计算机内存中进行排序的过程;
   - **外部排序**：指的是待排序记录的数量很大，以致内存一次不能容纳全部记录，在排序过程中尚需对外存进行访问的排序过程。
3. 内部排序方法的分类
   - 插入法：插入排序。折半插入排序，希尔排序
   - 交换类：冒泡排序，快速排序
   - 选择类：简单选择排序，树形选择排序和堆排序。
   - 归并类：归并排序

- 分配类：基数排序

4. 排序算法效率的评价指标：**执行时间(时间复杂度)**，**辅助空间(空间复杂度)**
5. **排序算法的稳定性：即大小相同的两个值在排序之前和排序之后的先后顺序不变，这就是稳定的。**

各排序方法简介表

表 8.2　　各种内部排序方法的比较

排序方法	时间复杂度			空间复杂度	稳定性
	最好情况	最坏情况	平均情况		
直接插入排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
折半插入排序	$O(n\log_2 n)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
希尔排序			$O(n^{1.3})$	$O(1)$	不稳定
冒泡排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	$O(\log_2 n)$	不稳定
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定
基数排序	$O(d(n+rd))$	$O(d(n+rd))$	$O(d(n+rd))$	$O(n+rd)$	稳定

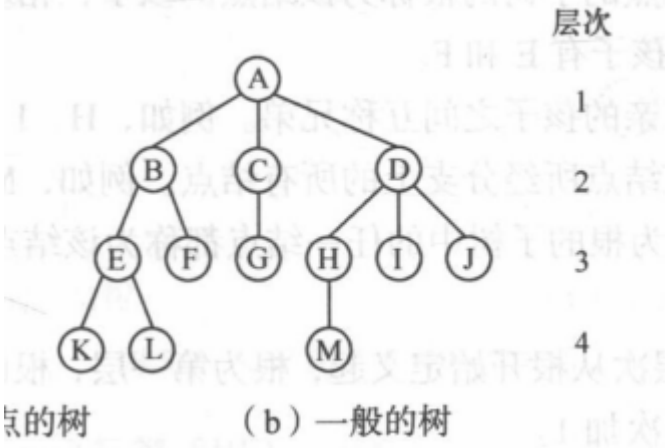
树

1. 定义：树的 n(n>=0)个结点的有限集，它或为空树或为非空树

   非空树的特点

   - 有且仅有一个称之为根的结点
   - 除根结点以外的其余结点可分为 m(m>0)个不相交的有限集 T1，T2，...Tm，其中每个集合本身又是一棵树，并且称为根的子树(SubTree)。
   - n>0 时根结点是唯一的
   - m>0 时，子树的个数没有限制，但他们**一定互不相交**

树的基本术语



点的树　　（b）一般的树

1. **结点**:树中的一个独立单元。包含一个数据元素及若干指向其子树的分支，如图 5.1(b)中的 A、B、C、D 等。
2. **结点的度**:结点拥有的子树数称为结点的度。例如，A 的度为 3,C 的度为 1,F 的度为 0。
3. **树的度**:树的度是树内各结点度的最大值。图 5.1 (b) 所示的树的度为 3。
4. **叶子**:度为 0 的结点称为叶子或终端结点。结点 K、L、F、G、M、I、J 都是树的叶子。
5. **非终端结点**:度不为 0 的结点称为非终端结点或分支结点。除根结点之外，非终端结点也称为内部结点。
6. **双亲和孩子**:结点的子树的根称为该结点的孩子，相应地，该结点称为孩子的双亲。例如，B 的双亲为 A，B 的孩子有 E 和 F。
7. **兄弟**:同一个双亲的孩子之间互称兄弟。例如，H、I 和 J 互为兄弟。
8. **祖先**:从根到该结点所经分支上的所有结点。例如，M 的祖先为 A、D 和 H。
9. **子孙**:以某结点为根的子树中的任一结点都称为该结点的子孙。如 B 的子孙为 E、K、L 和 F。
10. **层次**:结点的层次从根开始定义起，根为第一层，根的孩子为第二层。树中任一结点的层次等于其双亲结点的层次加 1。
11. **堂兄弟**:双亲在同一层的结点互为堂兄弟。例如，结点 G 与 E、F、H、I、J 互为堂兄弟。
12. **树的深度**:树中结点的最大层次称为树的深度或高度。图 5.1 (b)所示的树的深度为 4。
13. **有序树和无序树**:如果将树中结点的各子树看成从左至右是有次序的（即不能互换),则称该树为有序树，否则称为无序树。在有序树中最左边的子树的根称为第一个孩子，最右边的称为最后一个孩子。
14. **森林**:是 m (m≥0)棵互不相交的树的集合。对树中每个结点而言，其子树的集合即为森林。由此,也可以用森林和树相互递归的定义来描述树。
15. **树的高度、深度、层的区别(严蔚敏版)**：深度，高度，层次都从 1 开始，都一样
16. 线索二叉树：线索二叉树是指添加了直接指向节点的前驱和后继指针的二叉树。Ltag = 0 时，LChild 域指向结点的左孩子。 Ltag=1 时， LChild 域指向结点的遍历前驱。

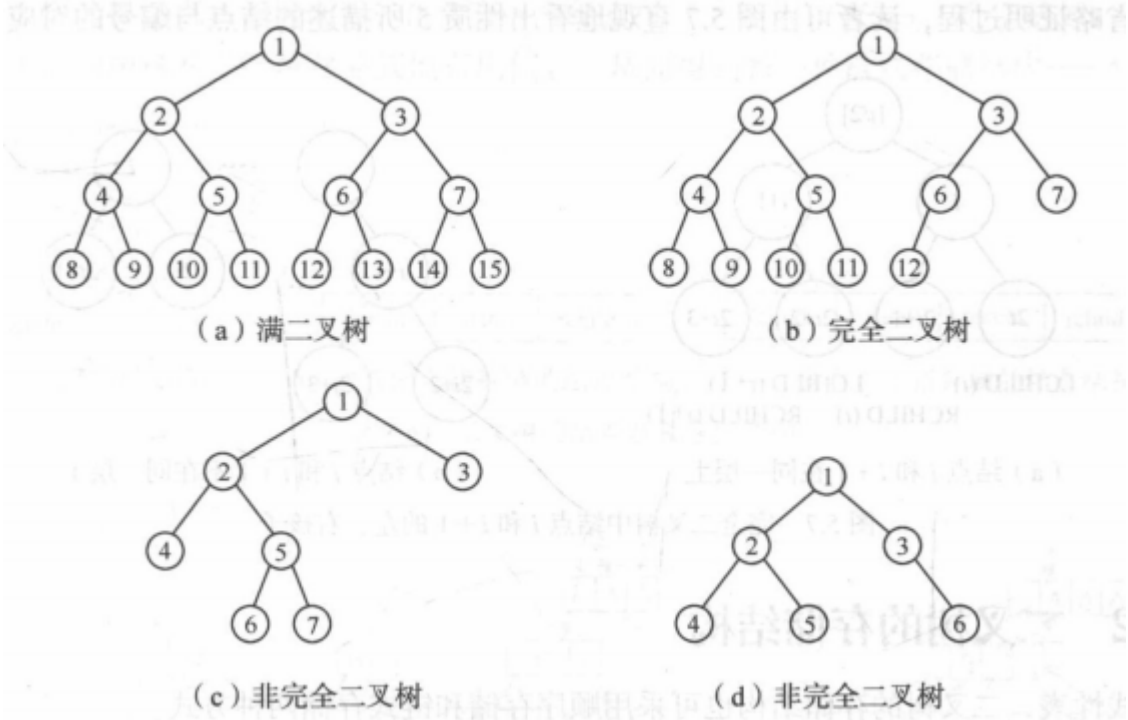
哈夫曼树的概念:

1. **路径**：从树中的一个结点到另外一个结点之间的分支构成的这两个结点之间的路径。
2. **路径长度**：路径上的分支数目称作路径长度。
3. **树的路径长度**：从树根到每一个结点的路径长度之和。
4. **权**：赋予某一个实体的一个量，是对实体的某个或某些属性的数值化描述。(其实就是属性值)
5. **结点的带权路径长度**：从该结点到树根之间的路径长度与结点上权的乘积。(权*结点路径)
6. **树的带权路径长度 WPL**：树中所有叶子结点的带权路径长度之和。
7. 哈夫曼树：假设有 m 个权值(W1,W2,W3...W(m)),可以构造一颗还有 n 个叶子结点的二叉树，每个叶子结点的权为 W(i)，**带权路径长度 WPL 最小的二叉树称做最优二叉树或哈夫曼树，哈夫曼树中，权值越大的结点离根结点越近**

二叉树

1. 定义：树的 n(n>=0)个结点所构成的集合，它或为空树(n=0)或为非空树

   非空二叉树的特点
   - 有且仅有一个称之为根的结点
   - 除根结点以外的其余点分为两个不交的子集 T1 和 T2，分别称为 T 的左子树和右子树，且 T1 和 T2 本身又都是二叉树。
   - 二叉树每个结点至多只有两棵子树(即二树中不存在度大于 2 的结点)
   - 二叉树的子树有左右之分，其次序不能任意颠倒
2. 由第一点可知：二叉树有 5 种形态：1. 空二叉树 2. 仅有根节点的二叉树 3. 右子树为空的二叉树 4. 左子树为空的二叉树 5. 左右子树均非空的二叉树
3. **二叉树的性质**
   i. 在二叉树的第 i 层上至多有 2^(i-1)个结点(i>=1)
   ii. 深度为 K 的二叉树至多有(2^k)-1 个结点
   iii. 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

- 

iv. 斜树：所有的结点都只有左子树的二叉树叫左斜树。所有结点都只有右子树的二叉树叫右斜树。这两者统称为斜树。

v. **满二叉树**：深度为 K 且含有(2^k)-1 个结点的二叉树

vi. **完全二叉树(从上到下次顺序相同，不存在空位)**：深度为 k 的有个结点的二叉树，当且仅当其每一个结点都与深度为 k 的满二叉树中编号从 1 至 n 的结点一一对应时，称之为完全二叉树。

- 完全二叉树的叶子结点只可能在层次最大的两层上出现
- 对任一结点，若其右分支下的子孙的最大层次为 ，则其左分支下的子孙的最大层次必为 1 或 1+1。图 5.6 中(c)和(d)不是完全二叉树。
- 具有 n 个结点的完全二叉树的深度为(log2^n)+1,(底数 2，指数 n，log 以 2 为底 n 的对数),
- 如果对一颗有 n 个结点的完全二叉树的结点按层序编号，则对任一结点 i 有
  a. 如果 i=1，则结点 i 是二叉树的根，无双亲，如果 i>1，其双亲是结点 i/2
  b. 如果 2i>n, 则结点无左孩子；否则其左孩子是结点 2i
  c. 如果 2i+1>n,则结点 i 无右孩子；否则其右孩子是结点 2i+1

二叉树的存储结构

1. 顺序存储：顺序存储仅适用于完全二叉树。因为，按树的结构可以会存在空子树，空子树的每个结点都会占用存储空间，会造成存储空间的浪费
2. 链式存储：对于非完全二叉树比较适合

```
typedef struct BitNode
{
  int data;
  struct BitNode *lchild, *rchild;
} BitNode;
```

3. **二叉树的链式存储和遍历方式**

三种遍历方式：可以记序为根，即先根遍历，中根遍历，后根遍历

1. 先序遍历(根左右)
2. 中序遍历(左根右)
3. 后序遍历(左右根)
4. 层序遍历(从上到下，从左到右)
5. **只有知道树的先序遍历和中序遍历，或者树的中序遍历和后序遍历，才能确定一颗二叉树**

参考

线索二叉树

1. 定义：将传统的二叉链表的空指针指向其前驱或者后继的指针，这样就可以像遍历单链表那样方便地遍历二叉树，以这种结点构成的二叉列表称为线索链表。加上线索的二叉树称为线索二叉树。参考链接
2. 概念：
   - 对于一个有 n 个结点的二叉链表，每个结点有指向左右孩子的两个指针域，所以共有**2n 个指针域**
   - **n 个结点的二叉树有 n-1 条分支，存在 2n-(n-1)= n+1 个空指针域**
3. 线索化：对二叉树以某种次序遍历，使其变为线索二叉树的过程称做是线索化
4. **出现的问题？**：无法知道某一节点的 lchild 是指向做孩子还是前驱，rchild 是指向右孩子还是后继，对此需要设置一个区分标志，即设置两个标志域**ltag**和**rtag**，只存放 0 或 1 数字的布尔型变量
   - ltag 为 0 时指向该结点的左孩子，为 1 指向前驱
   - rtag 为 0 时指向该结点的右孩子，为 1 指向后驱
5. **如何作用**：
   i. 若结点的左孩子不为空，LChild 指针域仍指向其左孩子，否则，LChild 指针域指向遍历过程序列的前驱结点。
   ii. 若结点的右孩子不为空，RChild 指针域仍指向其右孩子，否则，RChild 指针域指向遍历过程序列的后继结点。
   iii. 对于 Ltag 和 Rtag 的标志域的定义：
   Ltag = 0 时，LChild 域指向结点的左孩子。

Ltag=1 时， LChild 域指向结点的遍历前驱。
Rtag=0 时，RChild 域指向结点的右孩子。
Rtag=1 时，RChild 域指向结点的遍历后继。

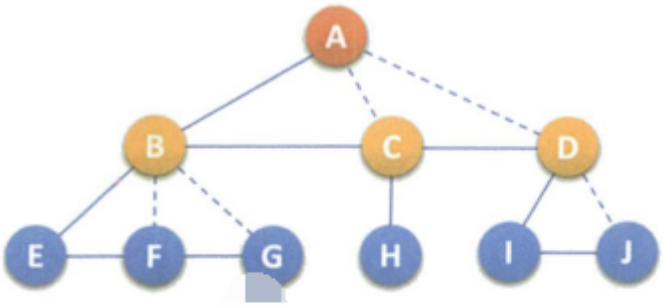
```
// 线索二叉树结构
typedef struct Node
{
   TypeData data;
   int Ltag,Rtag;
   struct Node*LChild;
   Struct Node*RChild;
} BiTree, *BiThrTree;
```

好处：加快查找结点的前驱或后继的速度

森林，树与二叉树的转换步骤

https://blog.csdn.net/weixin_44162361/article/details/119044059

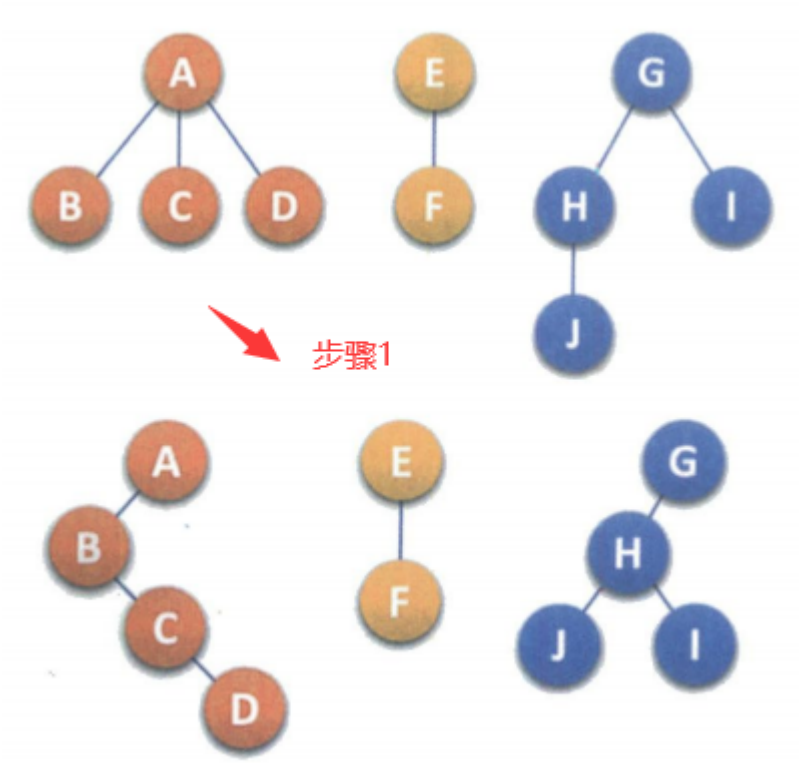
树转换为二叉树

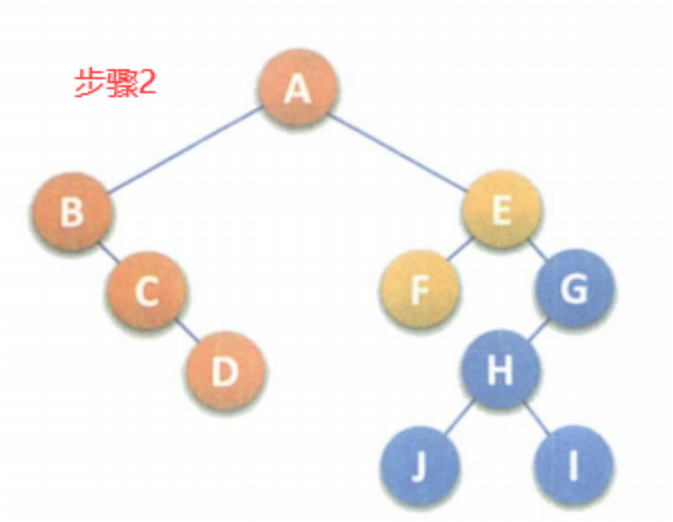


1. 加线，在所有兄弟节点之间加一条连线

2. 去线，对树中每个结点，只保留它与第一个孩子结点的连线，删除他与其他孩子结点间的连线
3. 层次调整，以树的根结点为轴心，将整棵树顺时针旋转一定的角度，使之结构层次分明。注意第一个孩子是二叉树结点的左孩子，兄弟转换过来的孩子是结点的右孩子。如右图所示。

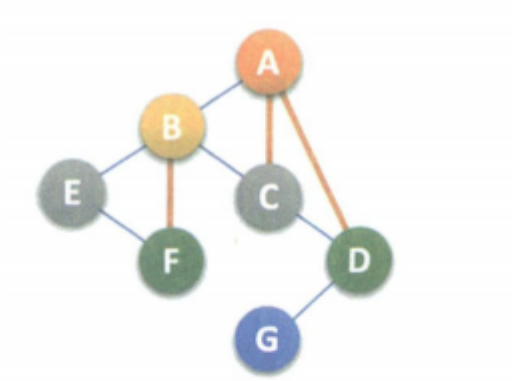
森林转换为二叉树

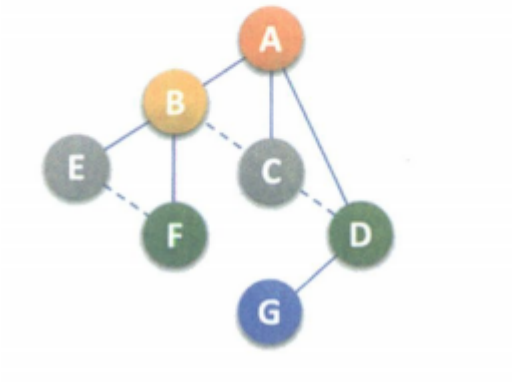




1. 把森林中的每颗树转换为二叉树
2. 第一棵二叉树不动，从第二棵二叉树开始，依次把后一棵二叉树的根结点作为前一棵二叉树的根结点的右孩子，用线连接起来。当所有的二叉树连接起来后就得到了由森林转换来的二叉树。如右图所示。

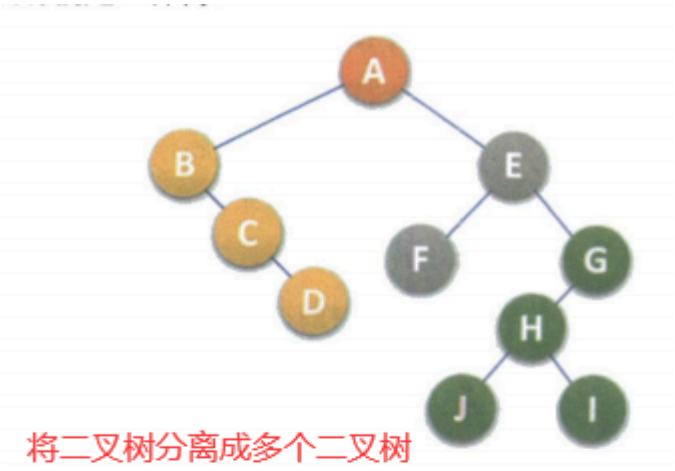
二叉树转换为树



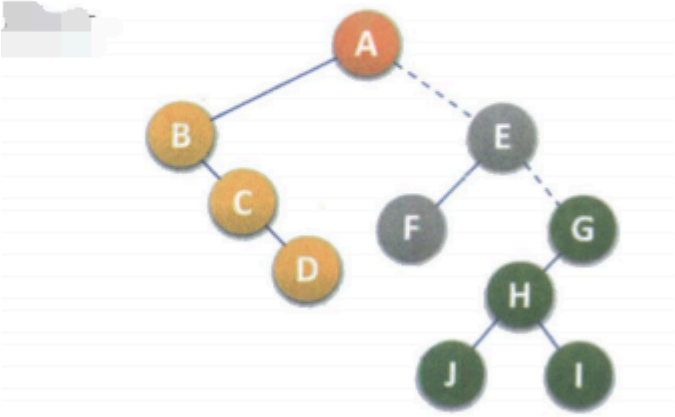


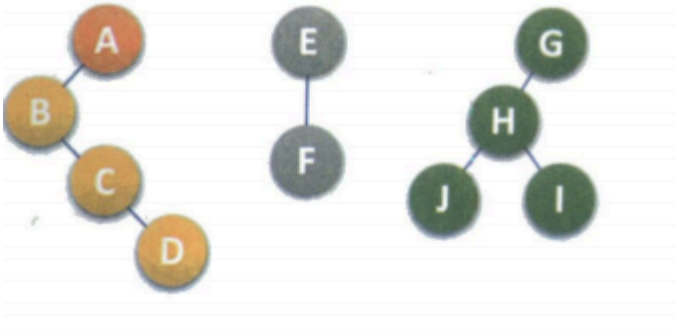
1. 加线：若某节点的左孩子的右孩子结点存在，则将这个左孩子的 n 个右孩子结点都作为此结点的孩子。将这些结点与这些有孩子结点用线连接起来
2. 去线：删除原二叉树中所有结点与其右孩子结点的连线
3. 层次调整

二叉树转换为森林



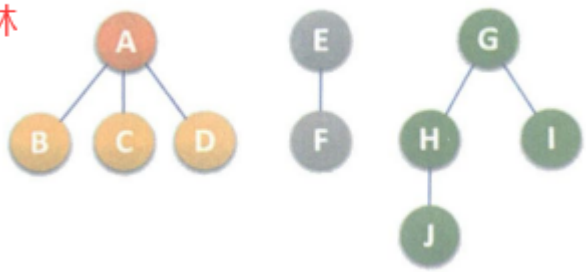
将二叉树分离成多个二叉树





1. 从根结点开始，若右孩子存在，则把与右孩子结点的连线删除(注：是根节点的右孩子 n 个右孩子)
2. 再查看分离后的二叉树，若右孩子存在，则连线删除……，直到所有右孩子连线都删除为止，得到分离的二叉树。如右图所示。

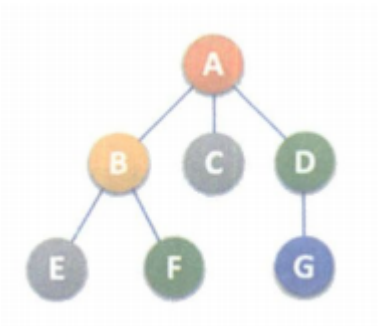
将分离后的二叉树分别转换成树，多个树形成森林



3. 将每颗分离后的二叉树转换为树即可。

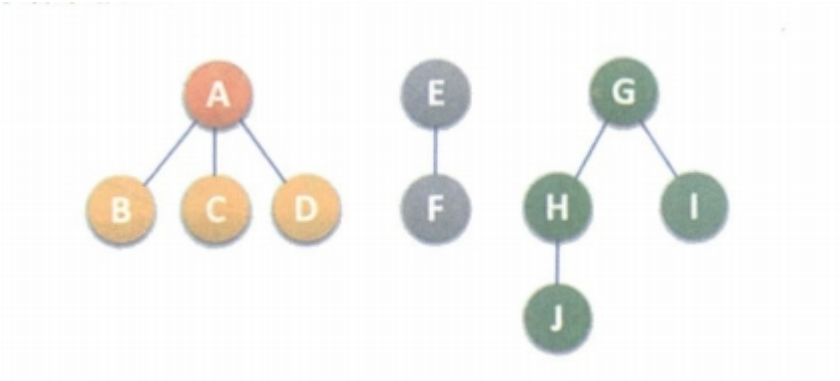
树与森林的遍历

树的遍历



1. 先根遍历：先访问树的根结点，然后依次先序遍历树的每颗子树。ABEFCDG
2. 后根遍历：先依次后序遍历每棵子树，然后再访问根结点。EFBCGDA
    - **树的先根遍历和二叉树的先序遍历结果相同**
    - **树的后根遍历和二叉树的中序遍历结果相同**
3. 口诀：树先先，树后中

森林的遍历



1. 先序遍历：先访问森林中第一棵树的根结点，然后再依次先根遍历根的每棵子树，再依次用同样方式遍历除去第一棵树的剩余树构成的森林。ABCDEFGHJI。

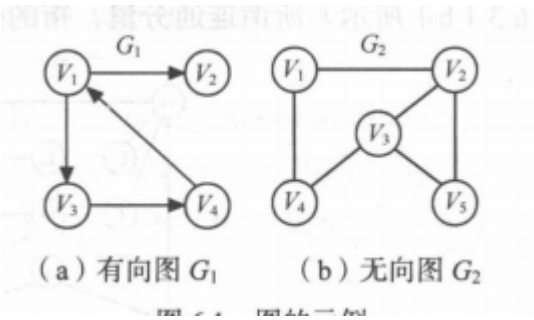
2. 中序遍历：先访问森林中第一棵树，中遍历的方式遍历每棵子树，然后再访问根结点，再依次用同样方式遍历除去第一棵树的剩余树构成的森林。BCDAFEJHIG。
    - **森林的前序遍历和二叉树的前序遍历结果相同**
    - **森林的中序遍历和二叉树的中序遍历结果相同**
3. 口诀：森先先，森中中

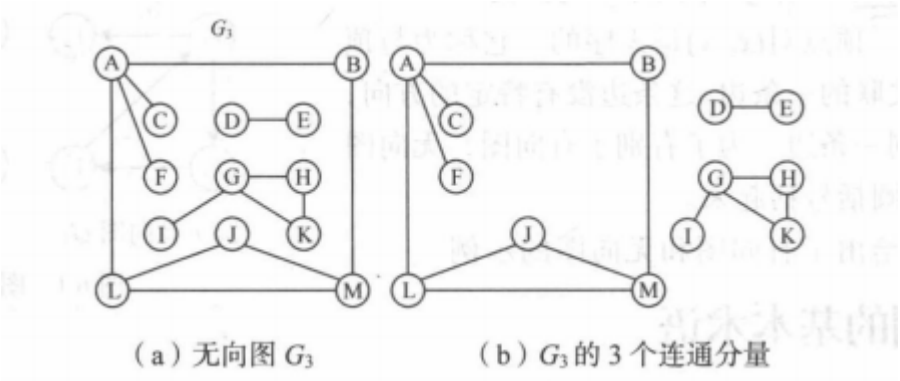
图(Graph)

1. 定义：**是由顶点的有穷非空集合和顶点之间边的集合组成**，通常表示为：G(V, E)，其中，G 表示一个图，V(Vertex)是图 G 中顶点的集合，E(Edge) 是图 G 中边的集合。其中**V(G)为图 G 顶点集合，V(G)不能为空，即在图结构中不能没有顶点**，**E(G)为图边的集合，E(G)可以为空集**
2. 分类：分为有向图和无向图。
    - **有向图**：边集 E(G)为有向边的集合
    - **无向图**：边集 E(G)为无向边的集合
3. **弧**：**弧**在有向图中，比如从顶点 x 到顶点 y 的一条有向边，因此<x,y>和<y,x>是不同的两条边，用尖括号括起来，x 是有向边的始点，y 是有向边的终点。<x,y>也称作一条弧，x 为弧尾，y 为弧头。
4. **边**：**边**在无向图中，顶点对(x,y)是无序的，称为顶点 x 到顶点 y 的一条边，这条边无特定方向，所以(x,y)和(y,x)是同一条边
5. **弧和边的关系**：边包括弧，弧不包括边，**有向边也称为弧。**如<A,D> A 是弧尾，D 是弧头，注意不能写成<D,A>，**为区别弧和边，无向图的顶点用一堆圆括号括起来**

图的基本术语

注：下面用 n 表示图中顶点数目，用 e 表示边的数目：





（a）无向图 G_3　　（b）G_3 的 3 个连通分量

4. **子图**：假如有两个图 G=(V,E)和 G1=(V1,E1),如果 V1 ∈ V,E1 ∈ E,则称 G1 为 G 的子图。
5. **无向完全图和有向完全图**：对于无向图，如果任意两个顶点之间都存在边，即存在 **n(n-1)/2** 条边，则称为无向完全图。对于有向图，如果任意两个顶点之间都存在方向互为相反的两条边，即有 **n(n-1)**条弧，则称为有向完全图。
6. **稀疏图和稠密图**：有很少条边或弧(如：e < nlog2^n，e 表示图中边（或弧）的数量，n 表示图中顶点的数量)的图称为稀疏图，反之称为稠密图
7. **权和网**：每条边/弧可以用一个实数来表示特定含义，这个实数就是权，而带权的图通常称为网
8. **邻接点**： 对于无向图 G，如果图的边(v，v')∈E，则称顶点 v 和 v 互为邻接点，即 v 和 v'相邻接。边(v, v)依附于顶点 v 和 v,或者说边(v, v')与顶点 v 和 v 相关联。其实就是**一条边上的两个顶点，这两个顶点就叫邻接点**
9. **度**: 顶点 v 的度是指和 v 相关联的边的数目，记为 TD(v)，**类似于树的结点的度。**
   - 度的计算公式：e(边数) = (图中各顶点的度相加)/2
   - **入度**：以顶点 v 为头的弧的数目，记为 ID(v);
   - **出度**：以顶点 v 为尾的弧的数目，记为 OD(v);
   - **出入度范例**：上图 G1 中的顶点 v1 的入度 ID(v1)=1，出度 OD(v1)=2，度 TD(v1)=ID(v1)+OD(v1)=3。
10. **路径和路径长度**：路径就是一个顶点到另外一个顶点的顶点序列，类似于一个数组，数组里面是路径的每一个点。**路径长度是一条路径上经过的边或弧的数目**
11. **回路或环**:第一个顶点和最后一个顶点相同的路径称为回路或环。
12. **简单路径、简单回路或简单环**:序列中顶点不重复出现的路径称为简单路径。除了第一个顶点和最后一个顶点之外，其余顶点不重复出现的回路，称为简单回路或简单环。
13. **连通、连通图和连通分量**：在无向图 G 中，如果从顶点 v 到顶点 v'有路径(**也就是不管中间隔了几个顶点，能通过顶点或边走到另外一个顶点**)，则称 v 和 v'是**连通**的。如果对于图中任意两个顶点 v、v'，都是连通的，则称 G 是连通图。上图无向图 G2 就是连通图。
    - **连通分量**：指无向图中极大连通子图。上图 G3 是非连通图，但 G3 有 3 个连通分量
14. **强连通图**：在**有向图**中，若任意两个顶点 Vi 和 Vj，满足从 Vi 到 Vj 以及从 Vj 到 Vi 都连通，也就是都含有至少一条通路，则称此有向图为强连通图。
    - **强连通分量**：有向图中的极大强连通子图称作有向图的强连通分量。
15. **连通图的生成树**: 一个极小连通子图,它含有图中全部顶点，但只有足以构成一棵树的 n-1 条边,这样的连通子图称为连通图的生成树。图 6.5 所示为 G3 中最大连通分量的一棵生成树。

16. **有向树和生成森林**:有一个顶点的入度为 0，其余顶点的入度均为 1 的有向图称为有向树。一个有向图的生成森林是由若干棵有向树组成，含有图中全部顶点，但只有足以构成若干棵不相交的有向树的弧。图 6.6 所示为其一例。

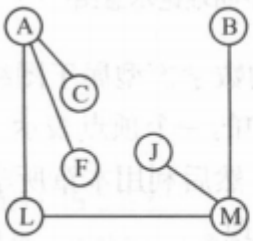


图 6.5　G_3 的最大连通分量的一棵生成树

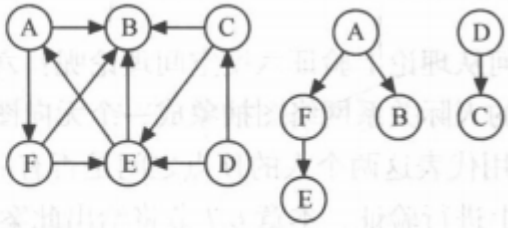


图 6.6　一个有向图及其生成森林

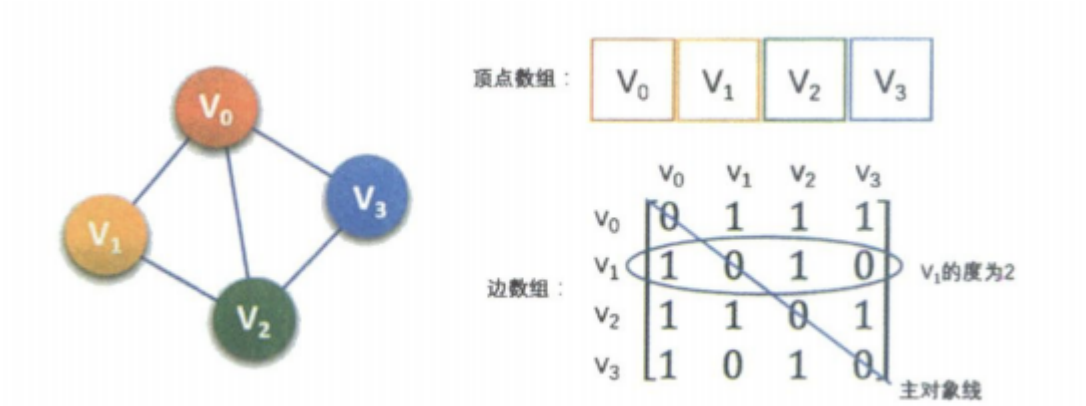
17. 更多参考相关概念参考

图的基本总结

1. 图按照有无方向分为有向图和无向图。有向图由顶点和弧构成，无向图由顶点和边构成。弧有弧尾和弧头之分。
2. 图按照边或弧的多少分为稀疏图和稠密图。如果任意两个顶点之间都存在边叫完全图，有向的叫有向完全图。若无重复的边或顶点到自身的边则叫简单图。
3. 图中顶点之间有邻接点、依附的概念。无向图顶点的边数叫做度，有向图顶点分为入度和出度。
4. 图上的边或弧上带权则称为网。
5. 图中顶点间存在路径，两顶点存在路径则说明是连通的，如果路径最终回到起始点则称为环，当中不重复叫简单路径。若任意两顶点都是连通的，则图就是连通图，有向则称强连通图。图中有子图，若子图极大连通则就是连通分量，有向的则称强连通分量。
6. 无向图中连通且 n 个顶点 n-1 条边叫生成树。有向图中一顶点入度为 0 其余顶点入度为 1 的叫有向树。一个有向图由若干棵有向树构成生成森林。

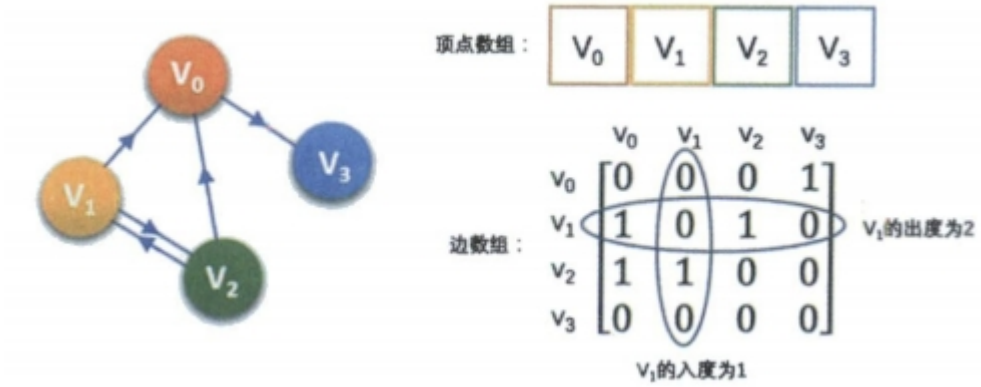
图的存储结构

1. 邻接矩阵(顺序存储)=> **无向图**：图的邻接矩阵（Adjacency Matrix)存储方式是用两个数组来表示图。一个一维数组存储图中顶点信息，一个二维数组（称为邻接矩阵）存储图中的边或弧的信息。适用于**稠密图**
   - 判断点 Vi 到 Vj 是否存在边或弧，只需查找矩阵中 arc[i][j]是否为 1 即可。
   - 判断顶点的度：这个顶点 Vi 在第 i 行的元素之和，
   - 求顶点 Vi 的所有邻接点就是将矩阵隔中的第 i 行元素扫描一遍



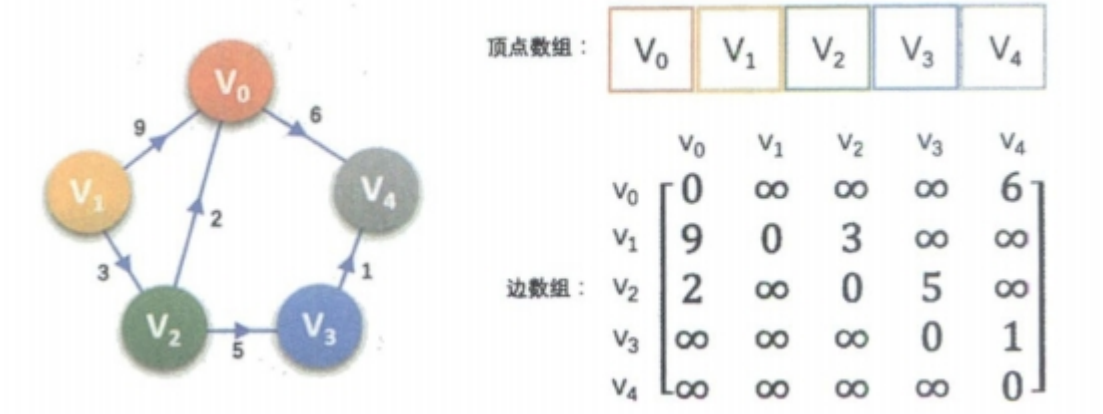
- **上图详解：**：矩阵里面的每个值，以行算，对应列。都是代表是否存在两个顶点间的边，比如 a[0][2]=1，代表有 横 V0 到 纵向 V2 的边，a[0][0]=0，代表是不存在 V0 到 V0 的边

2. 邻接矩阵(顺序存储)=> **有向图**



- Vi 的入度为第 Vi 列各数之和，出度为第 Vi 行各数之和：口诀**行出列入**

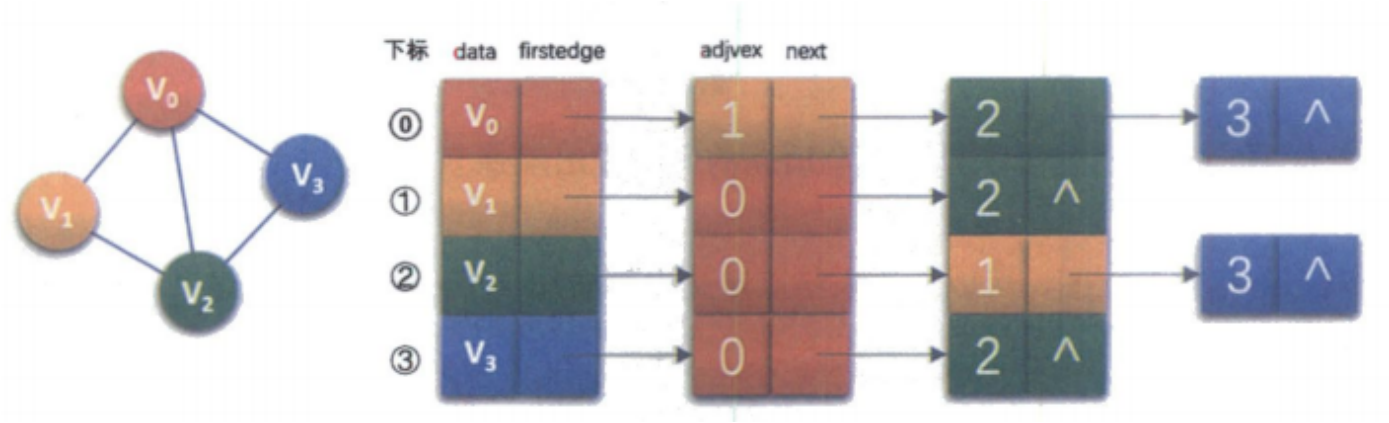
3. 邻接矩阵(顺序存储)=> **带权有向图**



- 带权图，两顶点之间如果不直接相连，则用 ∞ 表示

4. - **邻接矩阵优点**：
   i. 便于判断两顶点之间是否有边
   ii. 便于计算各个顶点的度。**对于有向图，第 i 行元素之和就是顶点 i 的出度，第 i 列元素之和就是顶点 i 的入度**
5. - **邻接矩阵缺点**：

   i. 不便于增加和删除顶点，
   ii. 不便于统计边的数目，时间复杂度为 O(n^2)
   iii. 空间复杂度高
6. 邻接表(链式存储)：由**表头节点表和边表组成**，数组与链表相结合，适用于**稀疏图**，类似于**二叉树的兄弟表示法**，**有向图的邻接表按是按图的出度作为链表项**



   i. 优点：
      - 便于增加和删除顶点
      - 便于统计边的数目，时间复杂度为 O(n+e)
      - 空间效率高
   ii. 缺点：
      - 不便于判断顶点之间是否有边。
      - 不便于计算有向图各个顶点的度
   iii. 参考：大话数据结构 P224
7. 十字链表(链式存储)：**即邻接表和逆邻接表结合**，
   - 优点：容易求得顶点的出度和入度
8. 邻接多重表(链式存储)：
9.

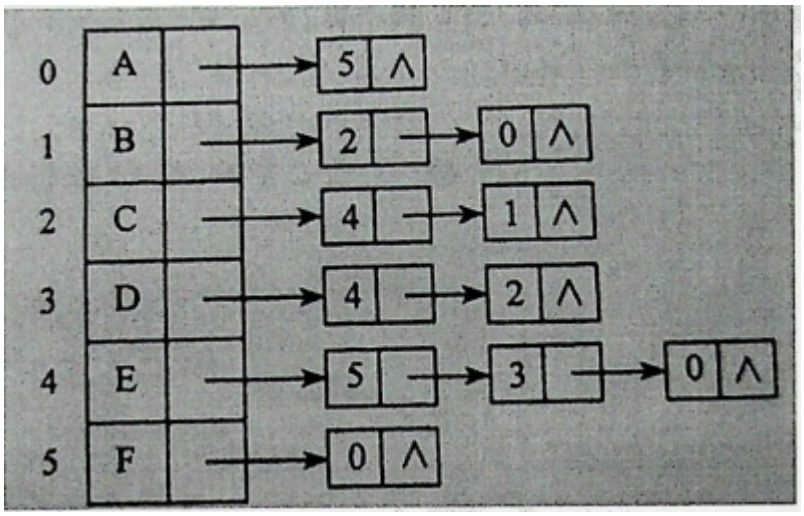
图的遍历

1. 深度优先遍历(Deep First Search)：也称深度优先搜索，沿着某一个深度方向遍历，**借助栈结构来实现(递归)，类似于前序遍历**
   - 当用邻接矩阵表示图时，查找每个顶点的邻接点的时间复杂度为 O(n^2)，其中 n 为图中顶点数。
   - 当以邻接表做图的存储结构时，查找邻接点的时间复杂度为 O(e)，其中 e 为图中边数。
   - 当以邻接表做存储结构时，深度优先搜索遍历图的时间复杂度为 O(n+e)。
2. 广度优先遍历(Breadth First Search): 也称广度优先搜索, 由第一个顶点从上到下顺序遍历，**借助队列结构来实现**
   - 当用邻接矩阵存储时，时间复杂度为 O(n^2);
   - 用邻接表存储时，时间复杂度为 O(n+e)。

3. 共同点：**两者时间复杂度相同，且都不唯一**

图的遍历例子 1
图的遍历例子 2

图的遍历例题



如上图所示，已知图 G 的邻接表如图所示若以顶点 B 为出发点，请分别写出深度优先搜索和广度优先搜索的项点序列。

/*
以上图的邻接表为例，链表中为当前顶点的下标，根据邻接表构造分析，

=> 深度优先以栈结构为主，查找当前结点的下一个结点，直到没有下一个结点再从头回溯

1．B的链接点为C(2)，C的链接点为E(4)，E的链接点为F(5)，F的链接点为A(0)，至此A没有连接点，往B的链接点回溯

所以答案为BCEFAD

2．根据邻接表画图，看下图，有4种可能，深度遍历有向图时需要根据方向来，没方向按序号或者字母序号来

1．BAFCED，这个比较特殊，因为走到F就停止了，需要向前回溯，而ED没有被访问也没有被BAF连接，所以只能回溯回C
2．BCEAFD
3．BCEFAD
4．BCEDAF

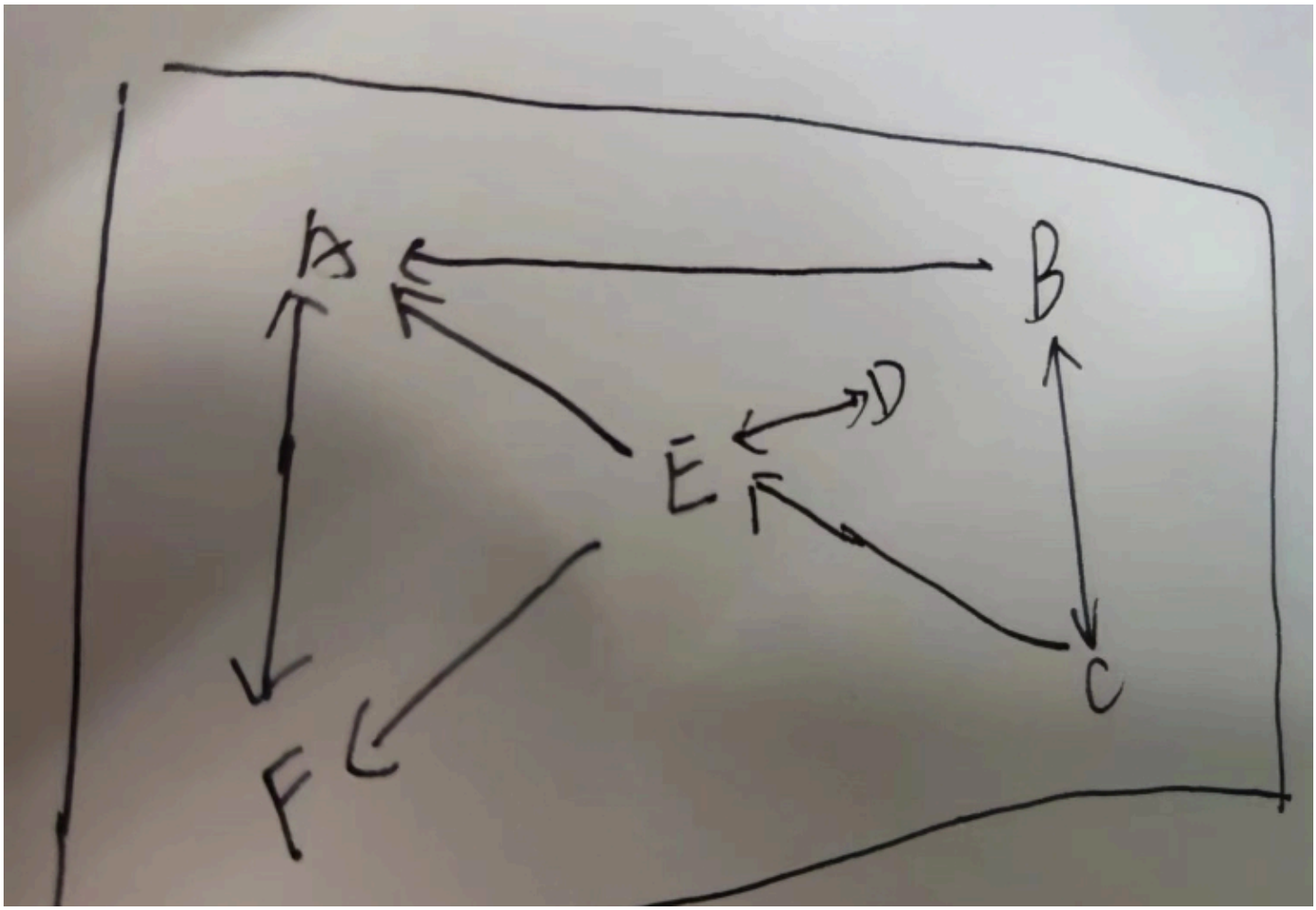
=> 广度优先以队列结构为主，可对图进行分层，

以B为结点A作为第2层的第一个结点有
B
AC
FE
D

以C为结点A作为第2层的第一个结点有
B
CA
EF
D

所以广度遍历优先为BACFED或BCAEFD

*/

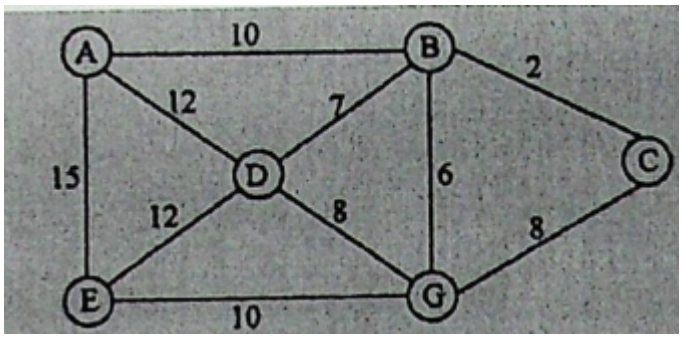


最小生成树

1. 定义：**构造连通网的最小代价生成树(也就是路径最小的权值)**，（最小生成树的应用：如城市间造路，要求造价最低）,**最小生成树不唯一，但最小生成树的边的权值之和总是唯一的**
2. 构造最小生成树的算法：
   - 普里姆算法(Prim)：**核心是归并点，时间复杂度 O(n^2)，适用于稠密网**，过程可以看大话数据结构 239 页，严蔚敏数据结构 166 页
   - 普里姆算法(Prim)过程：选择一个起点，选择一条权值最小的边，**如果遇到相同的边，则选择最新添加的顶点的边**，且所选的顶点不能形成环
   - 克鲁斯卡尔算法(Kruskal)：**核心是归并边，时间复杂度 O(elog2e)，适用于稀疏网**，过程可以看大话数据结构 244 页

最小生成树例题

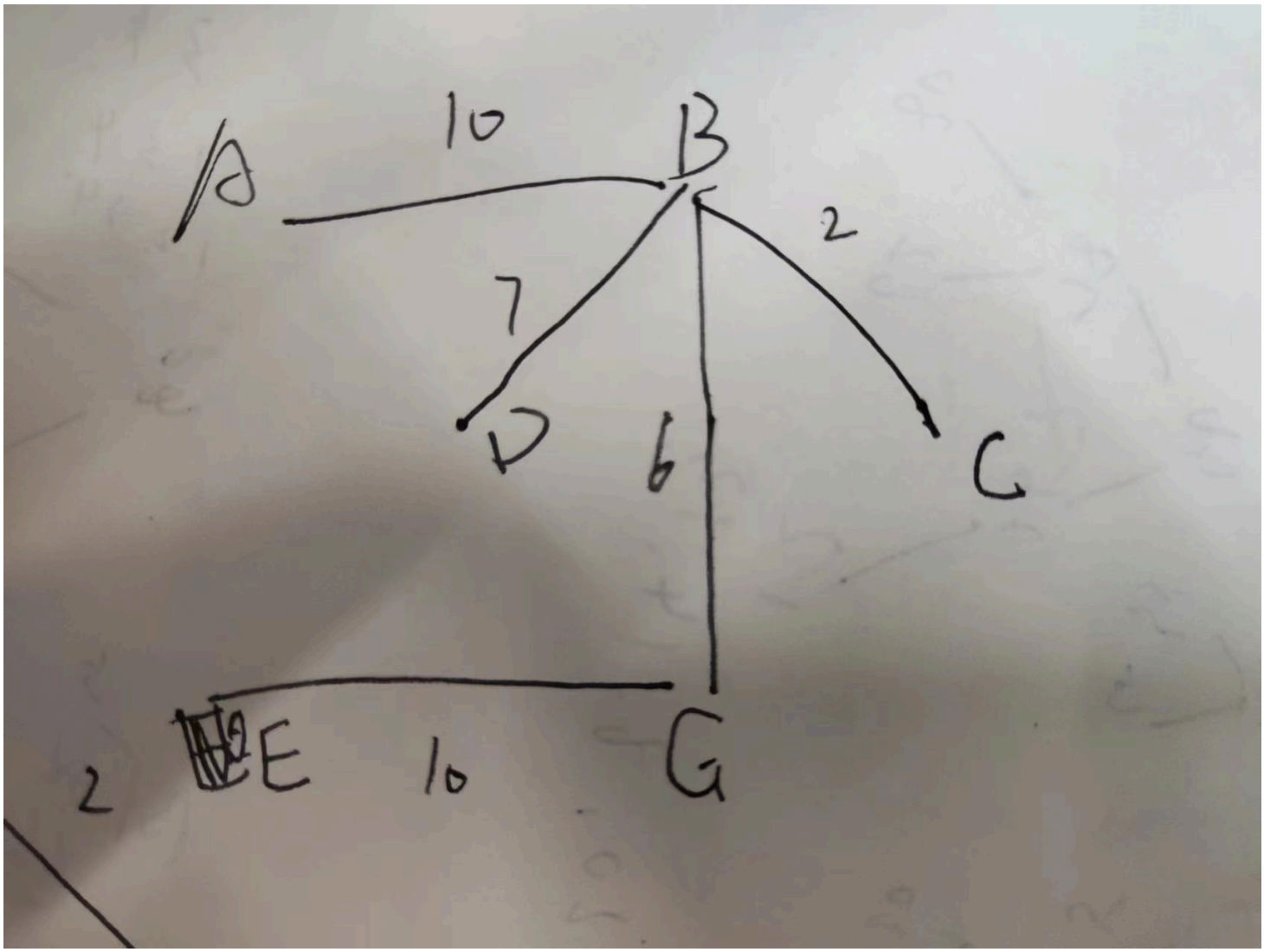
1. 寻找某一个顶点，选择这个顶点权值最小的边，以该点和其连接的点为基础，不断寻找最小值的点
2. 重复上面步骤(边不能围合合并)



如题 图所示为一个地区的交通网，顶点表示城市,边表示连接城市间的公路，边上的权值表示修建公路需花费的造价。现在需要选择能够连通每个城市且总造价最省的 5 条公路请画出修建公路的方案,并给出该工程的总造价。

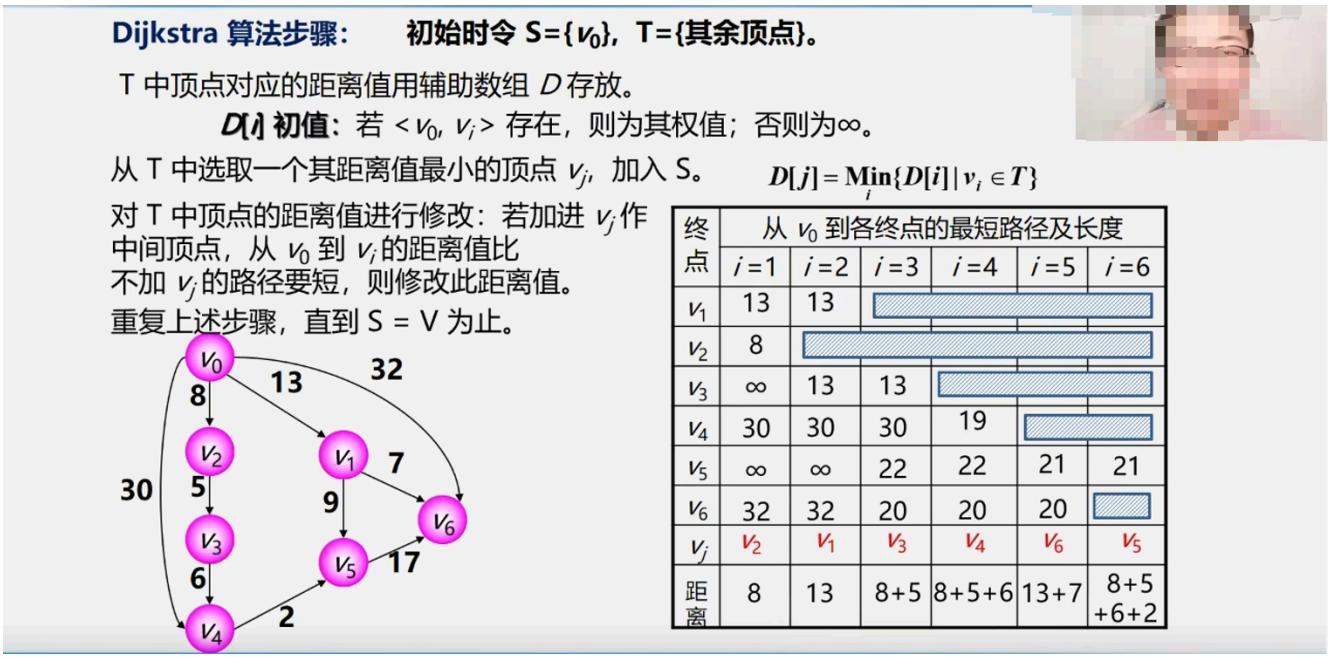
```
/*
1. 以B为基准点，最小的边为2，连接B和C
2. C和B的所有边种最小的为6，链接B和G
3. BCG最小的边为7，连接B和D
4. BDGC所有便中最小的边为8，但是不能选8，选择8的话会围合，
5. 排除8之后剩下两个都是10，都不围合，所以连接起来，最小生成树成功

总造价为2+6+7+10+10 = 35
*/
```



最短路径

1. 定义：是指两顶点之间经过的边上权值之和最少的路径，并且我们称**路径上的第一个顶点是源点，最后一个顶点是终点。拓扑排序不唯一**
2. 构造最短路径的算法：
    - 迪杰斯特拉算法(Dijkstra)：



- 弗洛伊德算法(Floyd)：主要目标是求解每一对顶点之间的最短路径

最小生成树和最短路径的区别

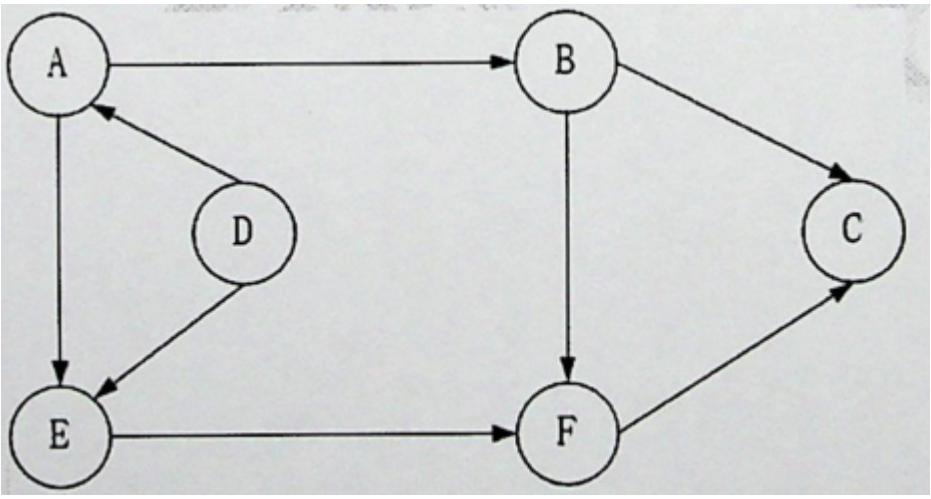
1. 定义：
   - 最小生成树能够保证整个拓扑图的所有路径之和最小，但不能保证任意两点之间是最短路径。
   - 最短路径是从一点出发，到达目的地的路径最小。
2. 总结：
   - 遇到求所有路径之和最小的问题用最小生成树&并查集解决；
   - 遇到求两点间最短路径问题的用最短路，即从一个城市到另一个城市最短的路径问题。
3. 区别：最小生成树构成后所有的点都被连通，而最短路只要到达目的地走的是最短的路径即可，与所有的点连不连通没有关系。

拓扑排序

1. 目的：**解决一个工程是否能顺序进行的问题**，检测是否存在环
2. 有向无环图：一个无环的有向图，简称 DAG 图
3. AOV-网：这种用顶点表示活动,用弧表示活动间的优先关系的有向图称为顶点表示活动的网
4. 拓扑排序：将 AOV-网中所有顶点排成一个线性序列，该序列满足:若在 AOV-网中由顶点 Vi 到顶点 Vj 有一条路径，则在该线性序列中的顶点 Vi 必定在顶点 Vj 之前(**前面走过的，不能再走一次，即不能存在回路**)。

拓扑排序例题：求拓扑排序序列

1. 寻找入度为 0 的顶点，删除该顶点和以它为弧尾的弧
2. 不断重复上面的步骤




```
/*
1．以上图为例：入度为0的顶点为D，删除D及其弧，序列为[D]
2．去掉D之后，入度为0的顶点为A，删除A及其弧，序列为[DA]
3．去掉A之后，入度为0的顶点为E,B，删除这两个之一都可以

4．以E为例子，删除E，序列为[DAE]，
5．去掉E后，入度为0的顶点为B，删除B及其弧，序列为[DAEB]
6．去掉B之后，入度为0的顶点为F，删除F及其弧，序列为[DAEBF]，剩下C，最终为[DAEBFC]

4．以B为例子，删除B，序列为[DAB]，
5．去掉B后，入度为0的顶点为E，删除E及其弧，序列为[DABE]
6．去掉E之后，入度为0的顶点为F，删除F及其弧，序列为[DAEBF]，剩下C，最终为[DABEFC]

所以上面图的拓扑排序序列为[DAEBFC]和[DABEFC]
*/
```

关键路径 AOE-网

1. 目的：**找一条从源点到汇点的带权路径长度最长的路径,称为关键路径**
2. AOE-网：即以边表示活动的网。AOE-网是一个带权的有向无环图，其中，顶点表示事件，弧表示活动，权表示活动持续的时间。通常，AOE-网可用来估算工程的完成时间.
3. **源点**：由于整个工程只有一开始点和完成，所以在正常情况下，**网中只有一个入度为 0 的点，称为源点**
4. **汇点**：出度为零的点
5. **关键活动**：关键路径上的活动。
6. **事件**：图的顶点
7. **活动**：图的边