高级数据结构-图

信智学院　陈琰宏

主要内容

01 图概念与性质

02 图存储

03 邻接表的应用举例

04 图搜索

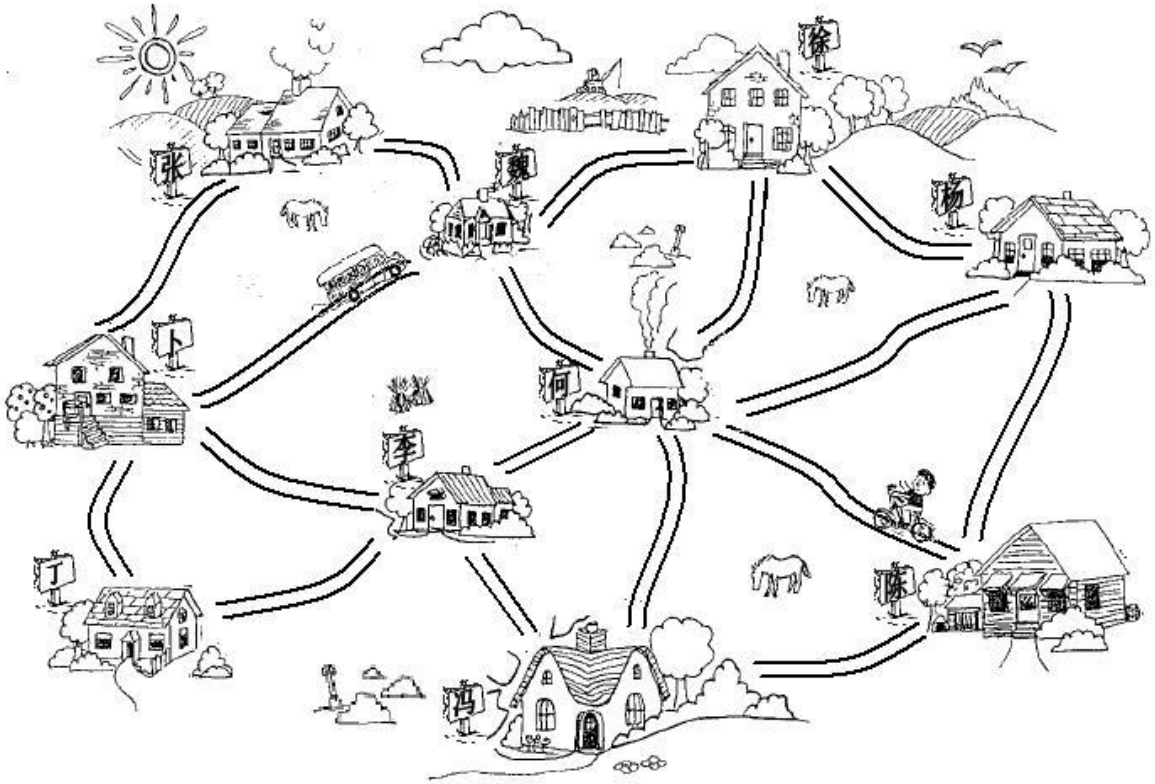
复习:线性结构、树形结构、图结构

数据的逻辑结构可以表示为**数据结点集合K**和**结点关系集合R**：**(K, R)**

- K：**系统中所有数据结点构成的集合**。
- R：**集合K中数据结点之间关系构成的集合**。 R是$K\times K$上的二元关系。

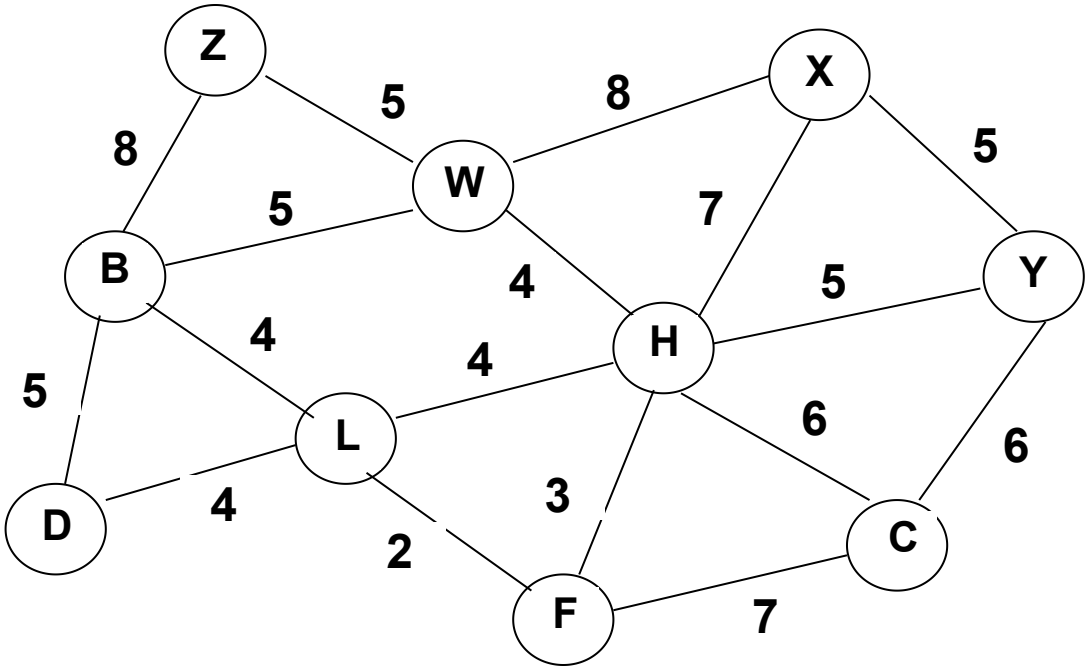
- **线性结构**：唯一前驱、唯一后继，反映一种**线性**关系。
- **树形结构**：唯一前驱、多个后继，反映一种**层次**关系。
- **图结构**：不限制前驱的个数，也不限制后继的个数，反映一种**网状**关系。

【例】图中村与村之间的道路是一个较长远的规划目标。



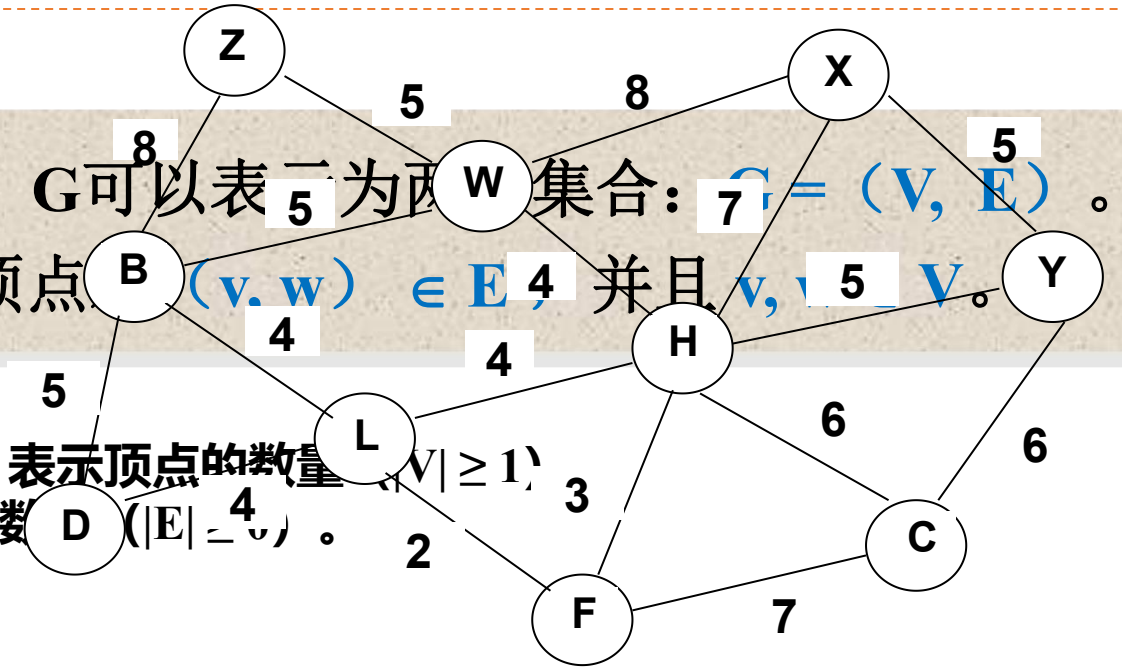
[问题1] 公路村村通项目要求用最小的投入实现每个村都能够有公路通达。那么应该选择建设哪些道路可以使这个投资最小呢？（假设每条道路的建设成本已知）

【例】下图为公路规划抽象及造价预算示例图。



[问题2] 在同样的抽象图中，假设把 “造价” 的含义修改成“距离”，那么我们就可以问：**要走遍每个村庄，并回到起点，该如何走才能够使得总的路程最短?**

6.1 图的定义与术语



“图”G可以表示为一个集合：G = (V, E)。每条边是一个顶点对(v, w) ∈ E，并且v, w ∈ V。

❖ 通常：用 |V| 表示顶点的数量（|V| ≥ 1），用 |E| 表示边的数量（|E| ≥ 0）。

[例] 上例图6.2给出了一个图的示例，在该图中：

集合V = { B，C，D，F，H，L，W，X，Y，Z }，

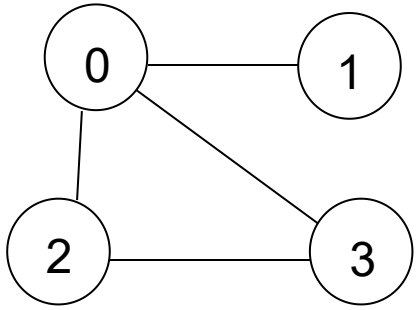
|V| = 10；

集合E = { (Z,B)，(Z,W)，(B,W)，(B,L)，(B,D)，(D,L)，(W,X)，(W,L)，(L,H)，(L,F)，(X,H)，(X,Y)，(H,Y)，(H,F)，(H,C)，(F,C)，(Y,C) }，

|E|= 17。

6.1 图的定义与术语

(1) **无向图**（Undirected Graphs）：边（v, w）等同于边（w, v）。用圆括号“（）”表示无向边。

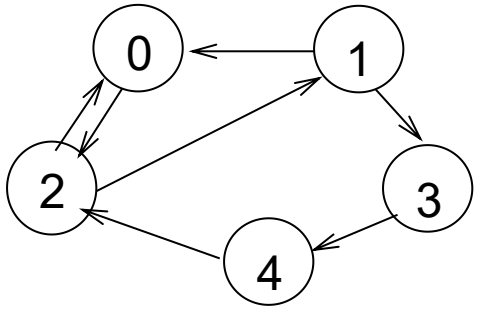


(a) 无向图G_1

$G_1 = (V_1, E_1)$,
$V_1 = \{0, 1, 2, 3\}$,
$E_1 = \{(0,1), (0,2), (0,3), (2,3)\}$。

6.1 图的定义与术语

(2) **有向图**（Directed Graphs）：边<v, w>不同于边<w, v>。用尖括号“< >”表示有向边；有向边也称“弧（Arc）”。

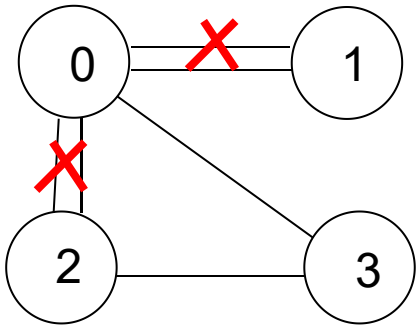


(b) 有向图G_2

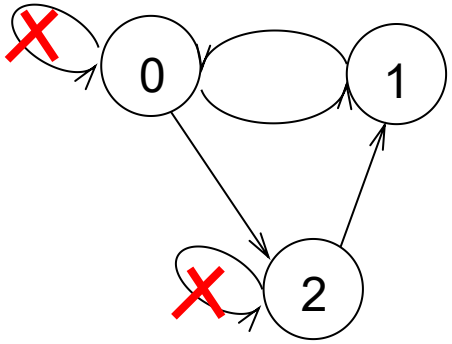
$G_2 = (V_2, E_2)$,
$V_2 = \{0, 1, 2, 3, 4\}$,
$E_2 = \{<1,0>, <2,0>, <0,2>, <2,1>, <4,2>, <1,3>, <3,4>\}$。

6.1 图的定义与术语

(3) 简单图（Simple Graphs）:没有重边和自回路的图。



(a) 重边图



(b) 自回路图

6.1 图的定义与术语

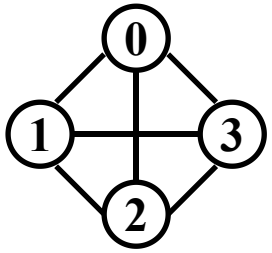
(4) 邻接点: 如果（v, w）或 < v, w >是图中任意一条边，那么称v和w互为“邻接点（Adjacent Vertices）”。

(5) 路径、简单路径、回路、无环图

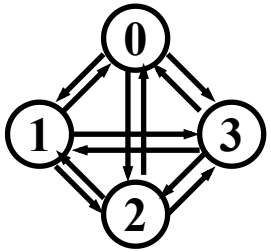
- 图G中从v_p *到* v_q 的路径 ::= { v_p, v_{i1}, v_{i2}, …, v_{in}, v_q } 使得(v_p, v_{i1}), (v_{i1}, v_{i2}), …, (v_{in}, v_q) 或 < v_p, v_{i1} >, …, < v_{in}, v_q > 都属于E(G)
- 路径长度 ::= 路径中边的数量
- 简单路径 ::= v_{i1}, v_{i2}, …, v_{in} 都是不同顶点
- 回路 ::= 起点和终点相同（$v_p = v_q$ ）的路径
- 无环图 ::= 不存在任何回路的图
- 有向无环图 ::= 不存在回路的有向图，也称DAG（Directed Acyclic Graph）

6.1 图的定义与术语

(6) **无向完全图**：在顶点数给定为n的情况下，边数达到最大的n(n-1)/2条边。（因为没有重边和自回路边）



$$|V| = n \Rightarrow$$
$$|E| = C_n^2 = \frac{n(n-1)}{2}$$



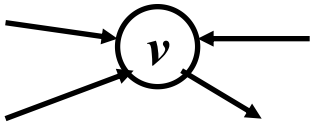
$$|V| = n \Rightarrow$$
$$|E| = P_n^2 = n(n-1)$$

(7)**有向完全图**：在顶点数给定为n的情况下，有向边数达到最大的n(n-1)条边。

6.1 图的定义与术语

(8)顶点的度(degree)、入度(in-degree)、出度(out-degree)：

- 度(v) ::= 与顶点v相关的边数



入度(v) = 3; 出度(v) = 1; 度(v) = 4

- 给定 n 个顶点和 e 条边的图G，则有：

$$e = \left(\sum_{i=0}^{n-1} d_i \right) \Big/ 2 \quad \text{其中 } d_i = \text{度}(v_i)$$

(9) 稠密图、稀疏图：是否满足 $|E| > |V|\log_2|V|$，作为稠密图和稀疏图的分界条件。

6.1 图的定义与术语

(10) 权（Cost）、网络（Network）

(11)图G的子图 G'： $V(G') \subseteq V(G)$ && $E(G') \subseteq E(G)$

(12) 无向图的顶点连通、连通图、连通分量：

- 如果无向图从一个顶点v_i到另一个顶点v_j $(i \neq j)$有路径，则称顶点v_i和v_j是“连通的（Connected）”
- 无向图中任意两顶点都是连通的，则称该图是“连通图（Connected Graph）”。
- 无向图的极大连通子图称为“连通分量（Connected Component）”。连通分量的概念包含以下4个要点：
  子图、连通、极大顶点数、极大边数

6.1 图的定义与术语

(13) 有向图的强连通图、连通分量：

- 有向图中任意一对顶点v_i和v_j $(i \neq j)$均既有从v_i到v_j的路径，也有从v_j到v_i的路径，则称该有向图是“强连通图（Strongly Connected Graph）”。

- 有向图的极大强连通子图称为“强连通分量（Strongly Connected Component）”。连通分量的概念也包含前面4个要点。

6.1 图的定义与术语

(14) 树、生成树：

- 树是图的特例：无环的无向图。
- 所谓连通图G的“生成树（Spanning Tree）”，是G的包含其全部n 个顶点的一个极小连通子图。它必定包含且仅包含G的n-1条边。
- 生成树有可能不唯一。
- 当且仅当G满足下面4个条件之一（完全等价）：
  - ① G有n-1条边，且没有环；
  - ② G有n-1条边，且是连通的；
  - ③ G中的每一对顶点有且只有一条路径相连；
  - ④ G是连通的，但删除任何一条边就会使它不连通。

类型名称：图（Graph）

数据对象集：一非空的顶点集合Vertex和一个边集合Edge，每条边用对应的一对顶点表示。

操作集：对于任意的图G ∈ Graph，顶点v、v_1和v_2 ∈ ertex，以及任一访问顶点的函数visit()，操作举例：

- Graph Create( )：构造并返回一个空图；
- void Destroy(Graph G)：释放图G占用的存储空间；
- Graph InsertVertex(Graph G, Vertex v)：返回一个在G中增加了新顶点v的图
- Graph InsertEdge(Graph G, Vertex v_1, Vertex v_2)：返回一个在G中增加了新边 (v_1, v_2) 的图；
- Graph DeleteVertex(Graph G, Vertex v)：删除G中顶点v及其相关边，将结果图返回；
- Graph DFS（ Graph G, Vertex v, visit()）：在图G中，从顶点v出发进行深度优先遍历；

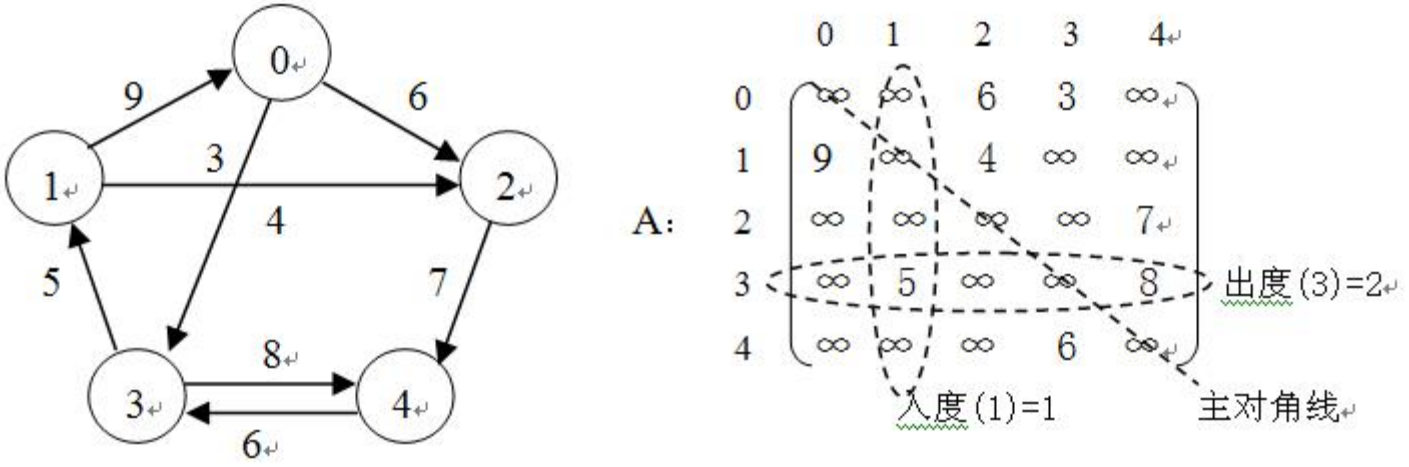
6.2 图的存储

✎**顶点信息：有n个顶点的图G(V, E) 用一维数组D [n] 表示；**

✎**边的信息：用邻接矩阵A [n] [n] 表示为：**

$$A[i][j]=\begin{cases}1 & \text{如果 } (v_i,v_j) \text{ 或 } <v_i,v_j> G\text{的边}\\ 0 & \text{其他}\end{cases}$$

❖**图的邻接矩阵表示示例**



6.2.1 邻接矩阵（Adjacency Matrix）

- 在邻接矩阵中，除了一个记录各个顶点信息的顶点数组外，还有一个表示各顶点之间关系的矩阵，称为**邻接矩阵**。

❖ **特点：**

- **无向图的邻接矩阵一定是一个对称矩阵。所需存储元素的个数是$|V|\times(|V|-1)/2$。**
- **对于无向图，邻接矩阵的第i行（**
- **对于有向图，邻接矩阵的第i行（或**
- **存储空间代价为$\Theta(|V|^2)$。要确定图中有多少条边，所花费的时间代价也是$\Theta(|V|^2)$。**

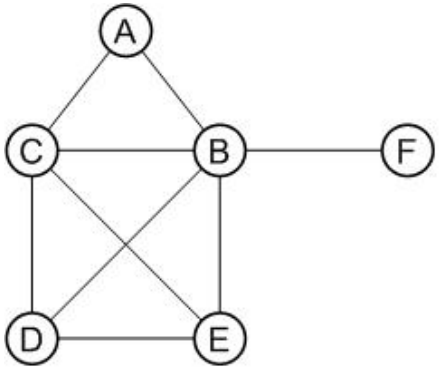
对稀疏图来说，这样的代价明显是不合理的！

6.2.1 邻接矩阵

1. 有向图(无向图)的邻接矩阵

设 $G(V,E)$ 是一个具有 n 个顶点的图，则图的邻接矩阵是一个二维数组 $Edge[n][n]$ ，它的定义为：

$$Edge[i][j]=\begin{cases}1 & \text{如果 } <i,j>\in \mathrm{E}，\text{或 } (i,j)\in \mathrm{E} \\ 0 & \text{否则}\end{cases}$$



(a) G_1

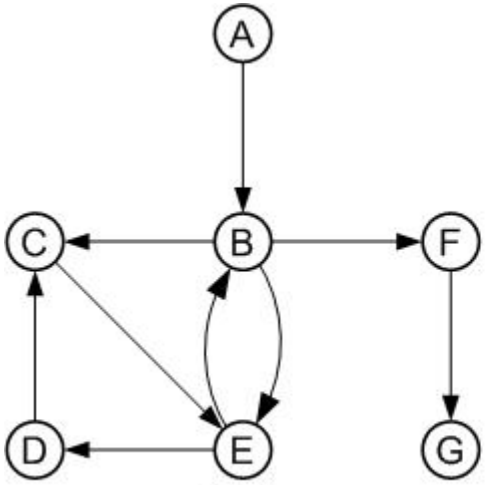
0	A
1	B
2	C
3	D
4	E
5	F

(b) 顶点数组

$$Edge=\begin{bmatrix}0&1&1&0&0&0\\1&0&1&1&1&1\\1&1&0&1&1&0\\0&1&1&0&1&0\\0&1&1&1&0&0\\0&1&0&0&0&0\end{bmatrix}$$

(c) 邻接矩阵

6.2.1 邻接矩阵



(a) G_2

0	A
1	B
2	C
3	D
4	E
5	F
6	G

(b) 顶点数组

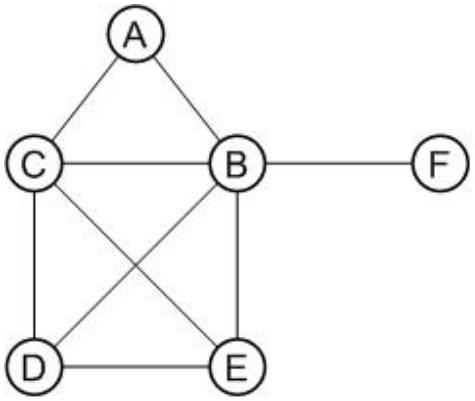
$$Edge = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(c) 邻接矩阵

有向图的邻接矩阵表示

注意：如果图中存在环（连接某个顶点自身的边）和重边（多条边的起点一样，终点也一样）的情形，则无法用邻接矩阵存储。

- **问题：从图的邻接矩阵可以获得什么信息？**



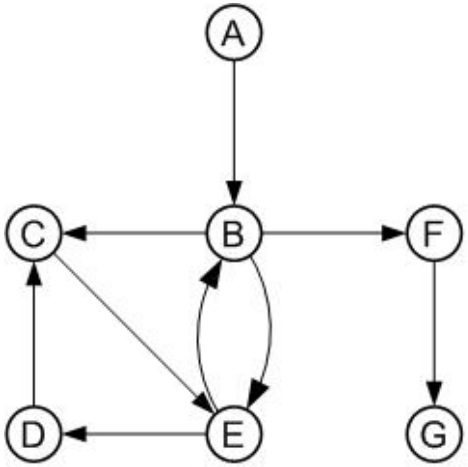
(a) G_1

0	A
1	B
2	C
3	D
4	E
5	F

(b) 顶点数组

$$Edge = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(c) 邻接矩阵



(a) G_2

0	A
1	B
2	C
3	D
4	E
5	F
6	G

(b) 顶点数组

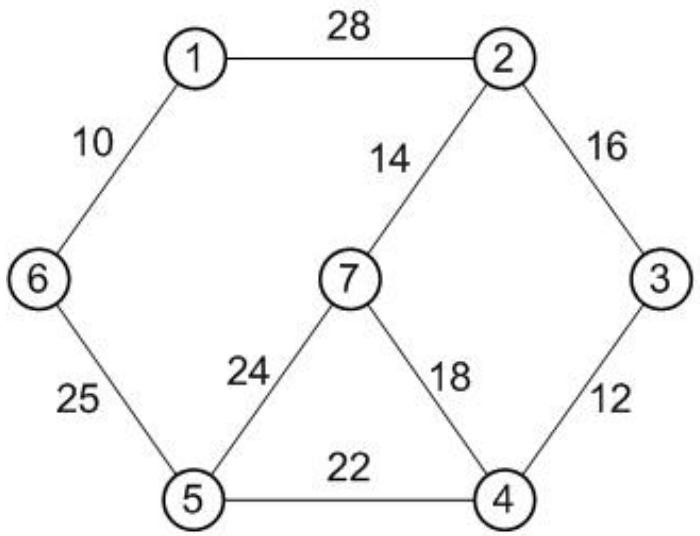
$$Edge = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(c) 邻接矩阵

2. 有向网(无向网)的邻接矩阵

对于网络(即带权值的图)，邻接矩阵定义为：

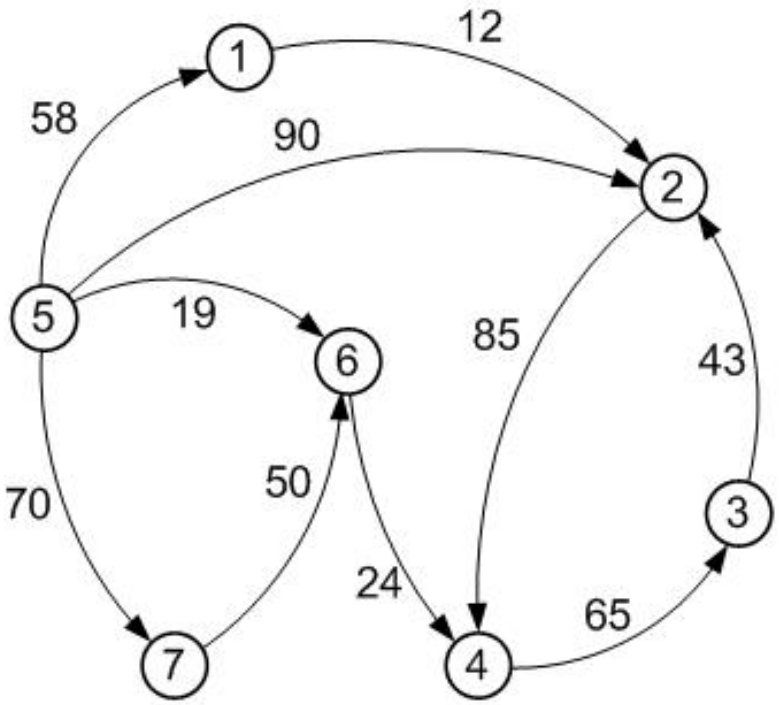
$$Edge[i][j]=\begin{cases} W(i,j) & \text{如果} i!=j，\text{且} <i,j>\in E（或(i,j)\in E） \\ \infty & \text{如果} i!=j，\text{但} <i,j>\notin E（或(i,j)\notin E） \\ 0 & \text{对角线上的位置，即} i=j \end{cases}$$



(a) G_1

$$Edge=\begin{bmatrix} 0 & 28 & \infty & \infty & \infty & 10 & \infty \\ 28 & 0 & 16 & \infty & \infty & \infty & 14 \\ \infty & 16 & 0 & 12 & \infty & \infty & \infty \\ \infty & \infty & 12 & 0 & 22 & \infty & 18 \\ \infty & \infty & \infty & 22 & 0 & 25 & 24 \\ 10 & \infty & \infty & \infty & 25 & 0 & \infty \\ \infty & 14 & \infty & 18 & 24 & \infty & 0 \end{bmatrix}$$

(b) 邻接矩阵



(a) G_2

$$Edge = \begin{bmatrix} 0 & 12 & \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & 85 & \infty & \infty & \infty \\ \infty & 43 & 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & 65 & 0 & \infty & \infty & \infty \\ 58 & 90 & \infty & \infty & 0 & 19 & 70 \\ \infty & \infty & \infty & 24 & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & 50 & 0 \end{bmatrix}$$

(b) 邻接矩阵

有向网的邻接矩阵表示

6.2.2 [3310]邻接矩阵实现

例 用邻接矩阵存储有向图，并输出各顶点的出度和入度。

输入描述：

输入文件中包含多个测试数据，每个测试数据描述了一个有向图。每个测试数据的第一行为一个正整数n (1≤n≤100)，表示该有向图的顶点数目，顶点的序号从1开始计起。接下来包含若干行，每行为两个正整数，用空格隔开，分别表示一条边的起点和终点。每条边出现一次且仅一次，图中不存在环和重边。0 0代表该测试数据的结束。输入数据最后一行为0，表示输入数据结束。

输出描述：

对输入文件中的每个有向图，输出两行：第1行为n个正整数，表示每个顶点的出度；第2行也为n个正整数，表示每个顶点的入度。每两个正整数之间用一个空格隔开，每行的最后一个正整数之后没有空格。

6.2.2 [3310]邻接矩阵实现

样例输入：

```
7
1 2
2 3
2 5
2 6
3 5
4 3
5 2
5 3
6 5
0 0
0
```

起点和终点

代表该测试数据的结束

输入数据结束

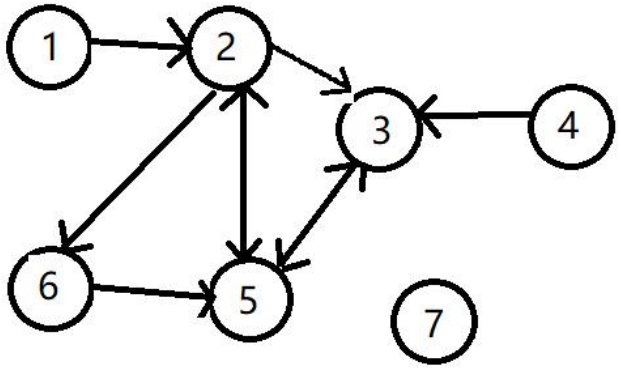
样例输出：

```
1 3 1 1 2 1 0
0 2 3 0 3 1 0
```

表示每个顶点的出度

表示每个顶点的入度

7 有向图的顶点数目，顶点的序号从1开始计起



6.2.2 [3310]邻接矩阵实现

分析：

在程序中可以使用一个二维数组存储表示邻接矩阵。注意，输入文件中顶点的序号是从1开始计起的，而二维数组中的下标是从0开始计起，所以在将边的信息存入到邻接矩阵时，需要将边的起点和终点的序号减1。在将有向边<u, v>存储表示到邻接矩阵Edge中时，只需要将元素Edge[u-1][v-1]的值置为1。

本题中的有向图都是无权图，邻接矩阵中每个元素要么为1，要么为0。

第i个顶点的出度等于邻接矩阵中第i行所有元素中元素值为1的个数。把第i行所有元素值累加起来，得到的结果也是该顶点的出度。

同理，在计算第i个顶点的入度时，也只需将第i列所有元素值累加起来累加起来即可。

题目要求输出n个顶点的出度(入度)时，每两个正整数之间用一个空格隔开，最后一个正整数之后没有空格。可以采取的策略是：输出第0个顶点的出度时前面没有空格，输出后面n-1个顶点的出度时输出一个空格。

6.2.2 [3310]邻接矩阵实现-写法1

```
    while( 1 )
    {
        scanf( "%d", &n );  //读入顶点个数n
        if( n==0 )  break;
        memset( Edge, 0, sizeof(Edge) );
        while( 1 )
        {
            scanf( "%d%d", &u, &v );//读入边的起点和终点
            if( u==0 && v==0 )  break;
            Edge[u-1][v-1] = 1; //构造邻接矩阵
        }
        for( i=0; i<n; i++ ) //求各顶点的出度
        {
            od = 0;
            for( j=0; j<n; j++ )  od += Edge[i][j];
            if(i==0)  printf( "%d", od );
            else  printf( " %d", od );
        }
        printf( "\n" );
        for( i=0; i<n; i++ ) //求各顶点的入度
        {
            id = 0;
            for( j=0; j<n; j++ )  id += Edge[j][i];
            if(i==0)  printf( "%d", id );
            else  printf( " %d", id );
        }
        printf( "\n" );
    }
```

```
#include <stdio.h>
#include <string.h>
#define MAXN 100
int Edge[MAXN][MAXN];   //邻接矩阵
int main( )
{
    int i, j;   //循环变量
    int n;  //顶点个数
    int u, v;       //边的起点和终点
    int od, id; //顶点的出度和入度
    while( 1 )
    {
    return 0;
}
```

6.2.2 [3310]邻接矩阵实现-写法2

```
1   #include<iostream>
2   #include<cstring>
3   using namespace std;
4   const int MAXN=100;
5   int edge[MAXN][MAXN];
6   int main(){
7       int n,u,v;
8       while(cin>>n&&n!=0){
9           memset(edge,0,sizeof(edge));
10          while(cin>>u>>v){
11              if(u==0&&v==0)break;
12              edge[u][v]=1;//构造邻接矩阵
13          }
```

6.2.2 [3310]邻接矩阵实现-写法2

```cpp
#include<iostream>
#include<cstring>
using namespace std;
const int MAXN=100;
int edge[MAXN][MAXN];
int main(){
    int n,u,v;
    while(cin>>n&&n!=0){
        memset(edge,0,sizeof(edge));
        while(cin>>u>>v){
            if(u==0&&v==0)break;
            edge[u][v]=1;//构造邻接矩阵
        }
        int sum1=0;
        for(int i=1;i<=n;i++){
            sum1=0;
            for(int j=1;j<=n;j++)sum1+=edge[i][j];
            cout<<sum1<<" ";//求出度
        }
        cout<<endl;
        int sum2=0;
        for(int i=1;i<=n;i++){
            sum2=0;
            for(int j=1;j<=n;j++)sum2+=edge[j][i];
            cout<<sum2<<" ";//求入度
        }
        cout<<endl;
    }
    return 0;
}
```

邻接矩阵的局限性

- 尽管绝大多数图论的题目可以采用邻接矩阵存储图，但由于邻接矩阵**无法表达环和重边的情形**，所以有时不得不采用邻接表去存储图。
- 另外，当**图的边数(相对顶点个数)较少**时，使用邻接矩阵存储会浪费较多的存储空间，而用邻接表存储可以节省存储空间。

6.2.3 邻接表(Adjacency List)

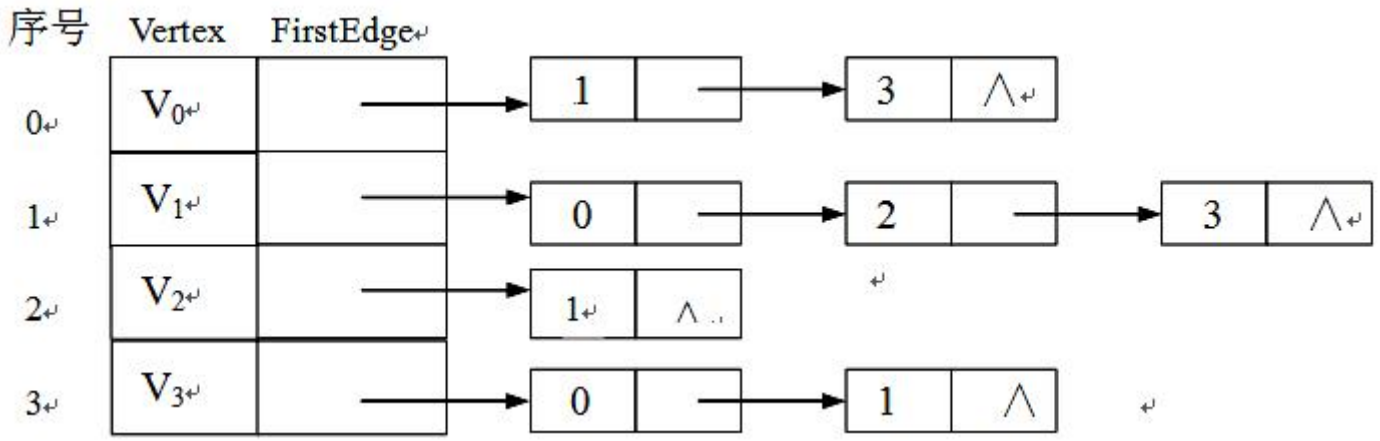
❖ 对于图G中的每个顶点v_i，将所有邻接于v_i的顶点v_j链成一个单链表，这个单链表就称为顶点v_i的邻接表，再将所有点的邻接表表头放到一个数组中，就构成了图的**邻接表**(这种邻接表也称为**出边表**)

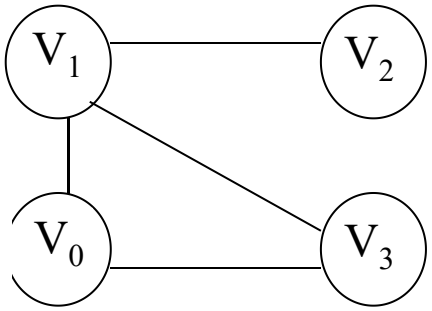
Vertex	FirstEdge

顶点域　边表头指针

AdjV	Next

邻接点域　指针域





6.2.3 邻接表(Adjacency List)

❖ 对于图G中的每个顶点v_i，将所有邻接于v_i的顶点v_j链成一个单链表，这个单链表就称为顶点v_i的邻接表，再将所有点的邻接表表头放到一个数组中，就构成了图的**邻接表**(这种邻接表也称为**出边表**)

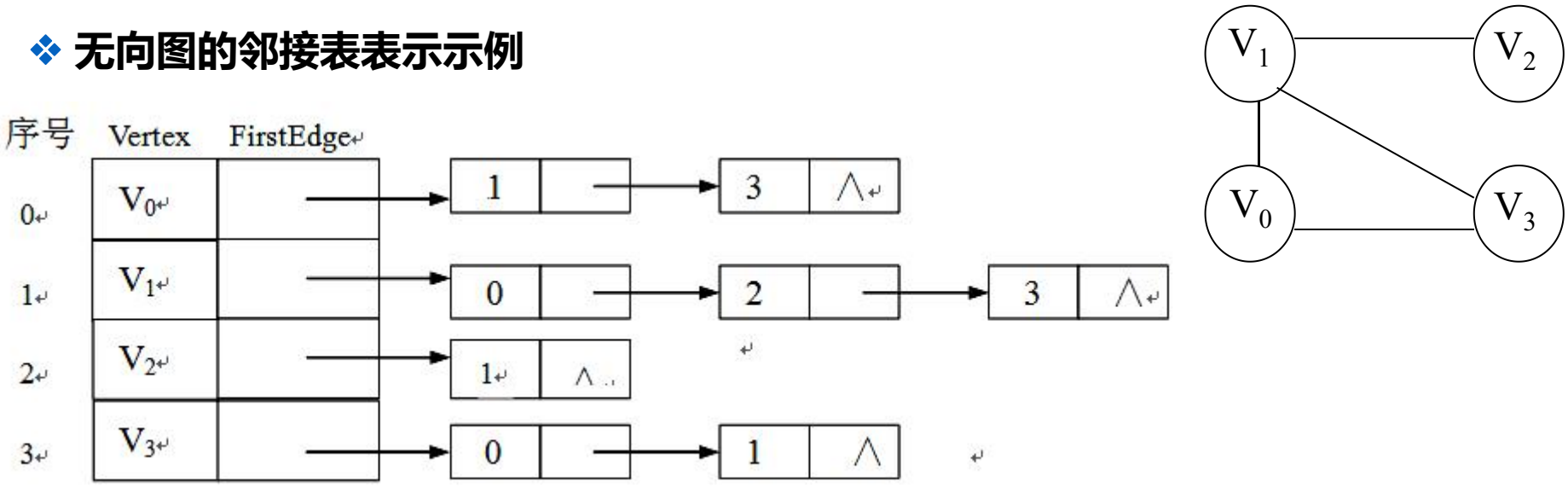
Vertex	FirstEdge

顶点域　边表头指针

AdjV	Next

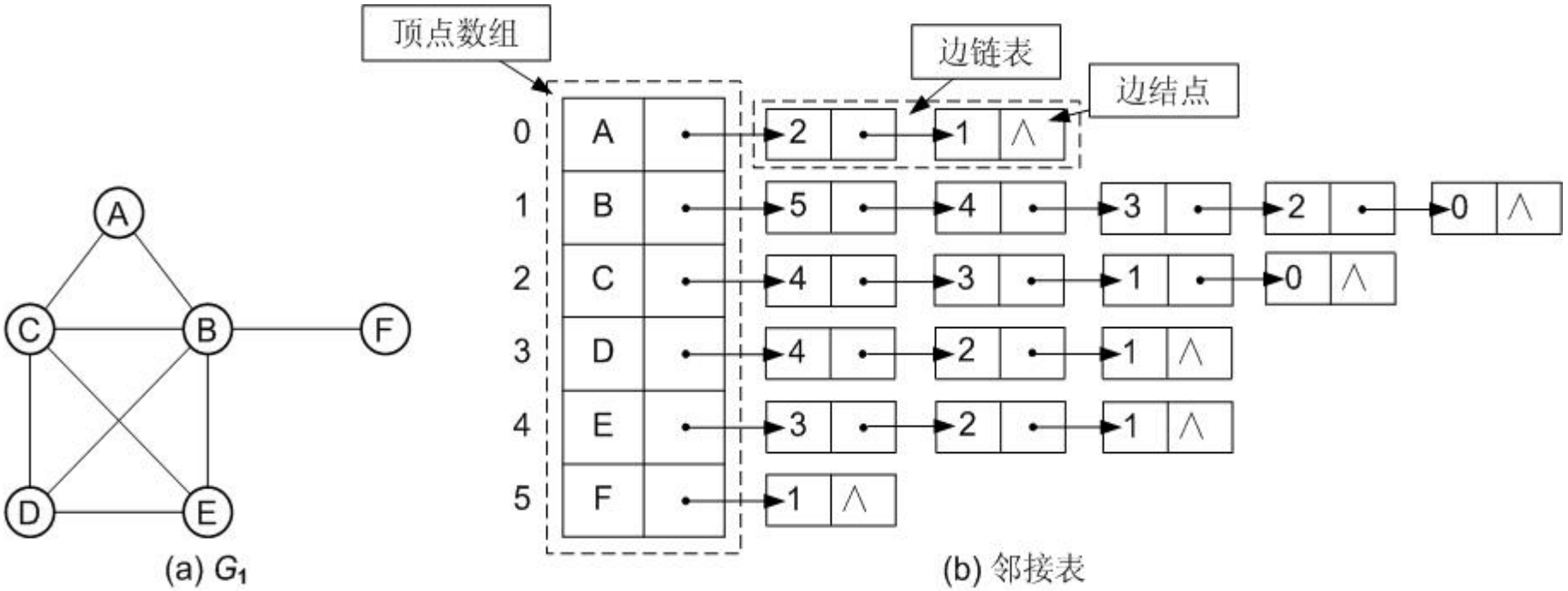
邻接点域　指针域

❖ 无向图的邻接表表示示例



6.2.3.1 无向图的邻接表

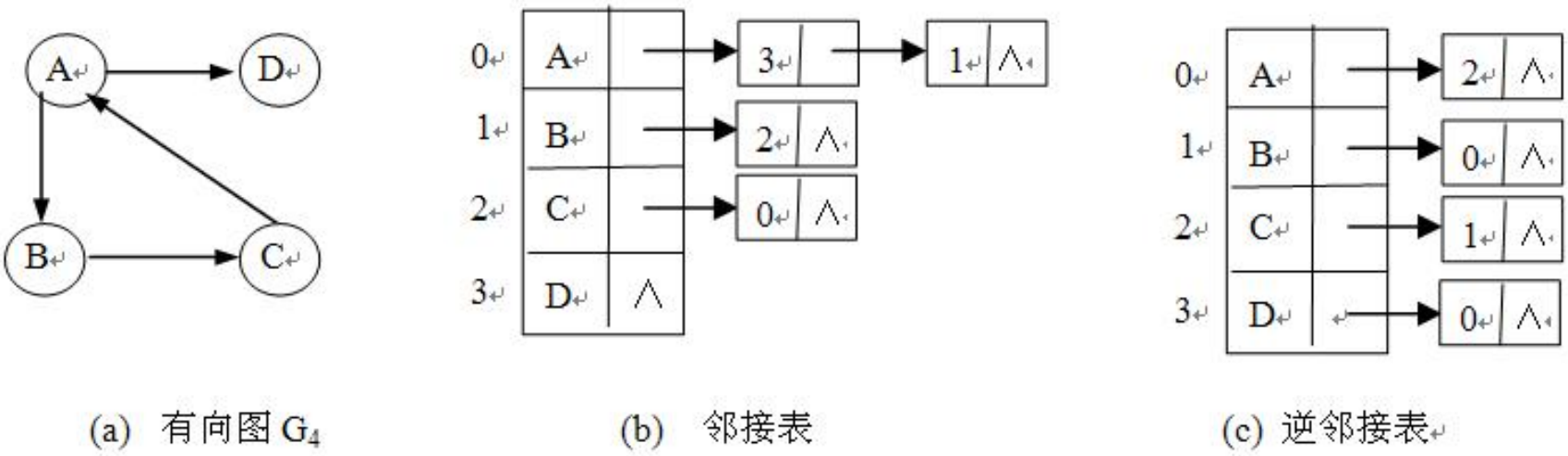
❖**无向图**中有n 个顶点和e条边，则它的邻接表需n个头结点和2e个表边结点。显然，在边**稀疏** (e << n(n-1)/2) 的情况下，用**邻接表表示图比邻接矩阵节省存储空间**；



无向图的邻接表：**每条边在邻接表里出现2次**

6.2.3.2 有向图的邻接表

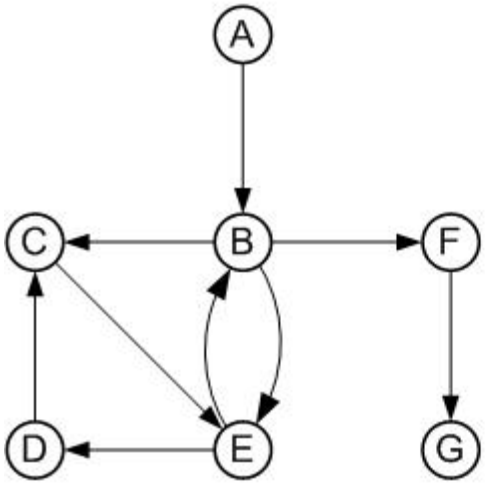
❖无向图的邻接表，顶点v_i的度恰为第i个链表中的结点数；而在有向图中，第i个链表中的结点个数只是顶点v_i的出度，为便于确定顶点v_i的入度，可以建立一个有向图的逆邻接表（入边表），即对每个顶点v_i建立一个链接以v_i为头的弧的链表。 例如：



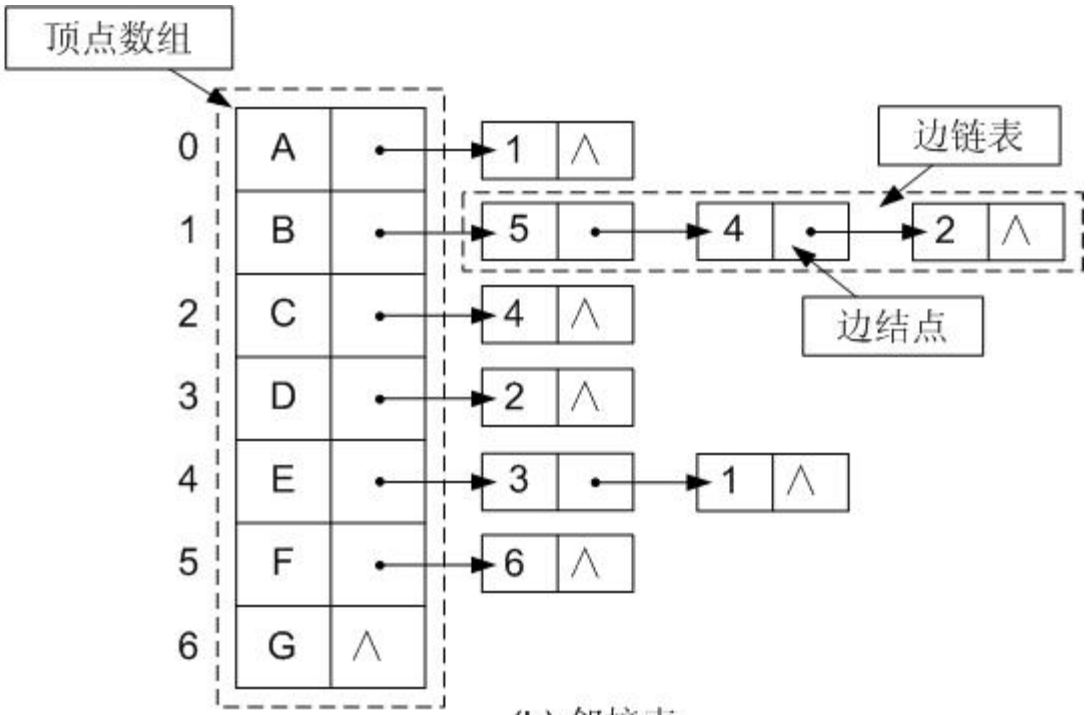
(a) 有向图 G_4　　(b) 邻接表　　(c) 逆邻接表

图 6.16　有向图 G4 及其邻接表和逆邻接表

有向图的邻接表（出边表）

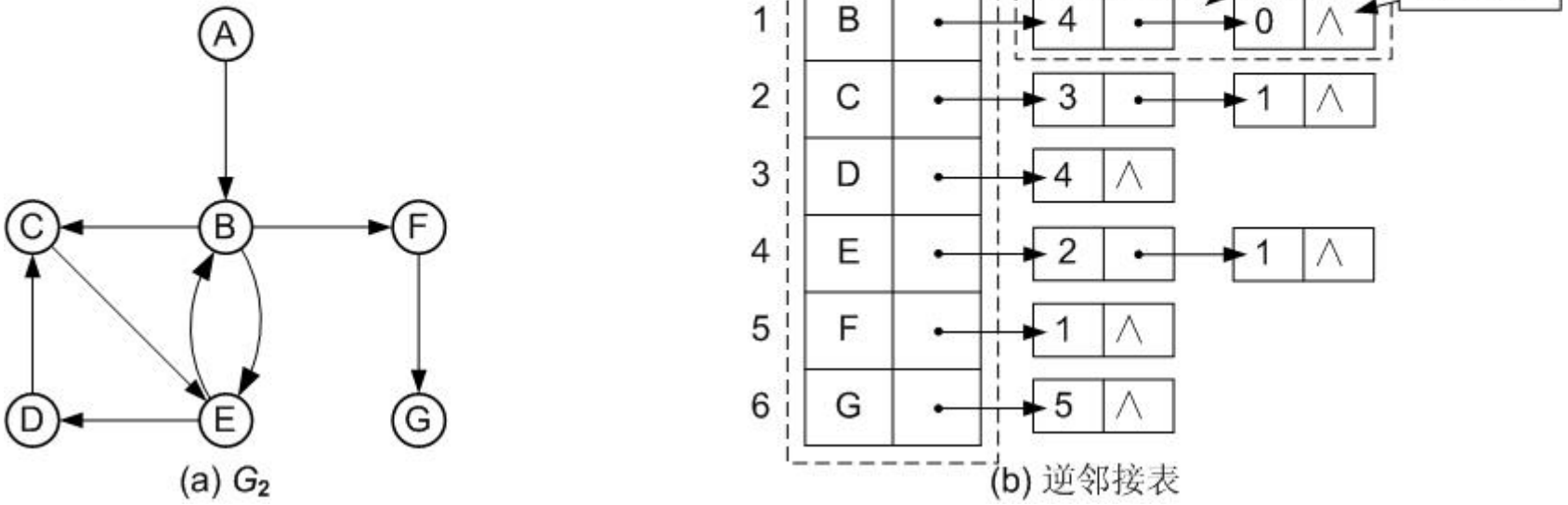


(a) G_2



(b) 邻接表

有向图的逆邻接表（入边表）



6.2.3.3 [3311]邻接表的实现-方法1

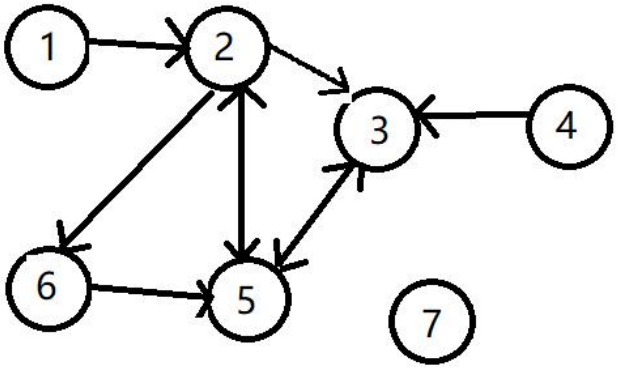
用邻接矩阵存储有向图，并输出各顶点的出度和入度。

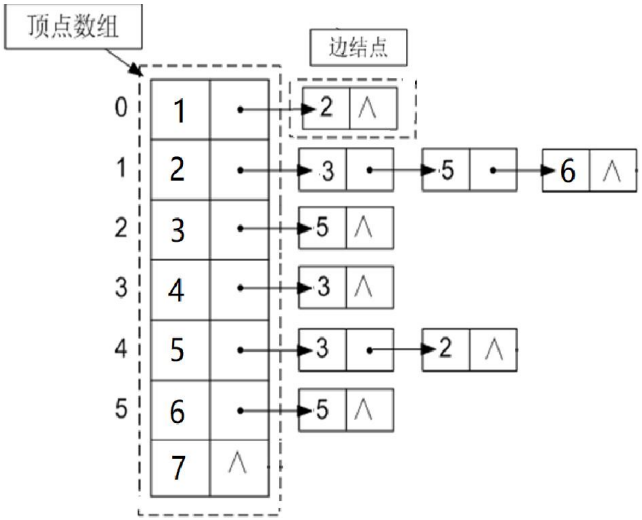
样例输入

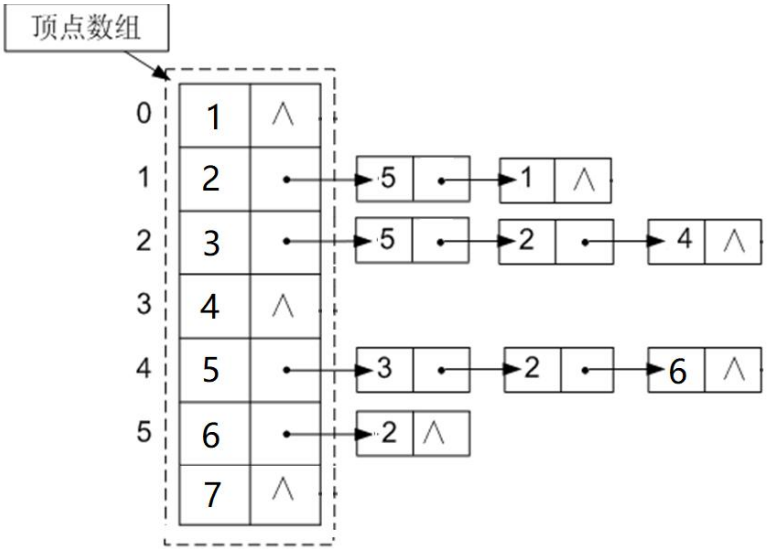
7
1 2
2 3
2 5
2 6
3 5
4 3
5 2
5 3
6 5
0 0
0

样例输出

1 3 1 1 2 1 0
0 2 3 0 3 1 0

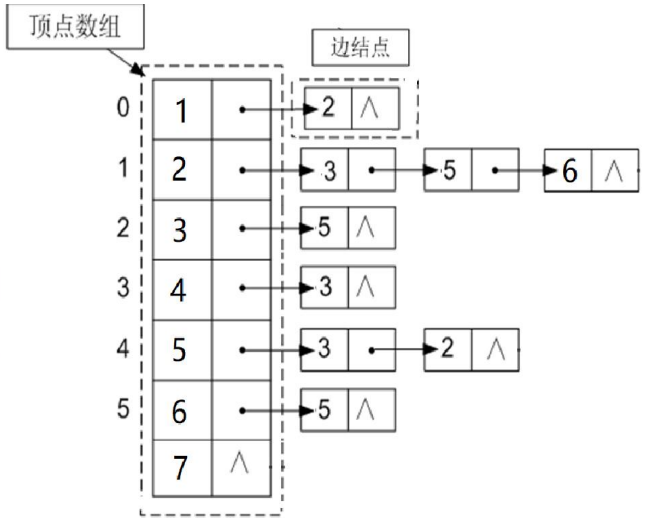


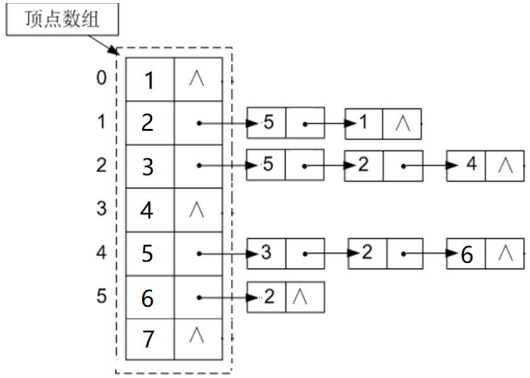




a. 建立头节点、边节点

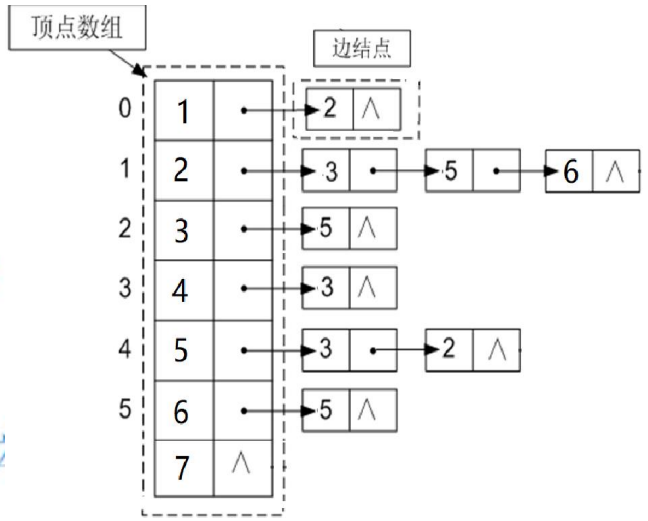
```
1   # define MAXN 101
2   int Edge[MAXN][MAXN];
3   struct vNode{//顶点
4       int data;//顶点信息
5       arcNode *head1;//出边表的表头指针
6       arcNode *head2;//入边表的表头指针
7
8   };
9   struct arcNode{//边节点
10      int adjvex;//边的另一个邻接点的序号
11      arcNode *nextarc;//指向下一个边节点的指针
12  };
13  struct graph{//图的邻接表存储结构
14      vNode vertex[MAXN]; //顶点数组
15      int vernum,arcnum;//顶点数和边(弧)数
16  }lg;
```

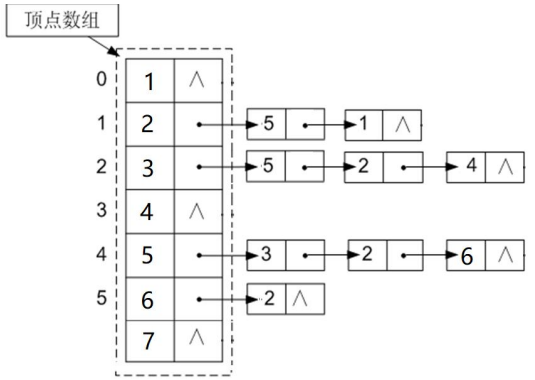




b. 建立入边、出边表

```
22  void createLG(){
23      int i=0;
24      int v1,v2;
25      arcNode *tmp;
26      for(int i=1;i<=MAXN;i++)//初始化链表节点
27          lg.vertex[i].head1=lg.vertex[i].head2=NULL;
28      while(1){
29          cin>>v1>>v2;
30          if(v1==0&&v2==0)break;
31          tmp=new arcNode;//建立准备先插入的节点   建立
32
33          tmp->adjvex=v2;//对新节点的数据域赋值
34          //采用头插法，把该节点插入到自己的链中
35          tmp->nextarc=lg.vertex[v1].head1;
36          lg.vertex[v1].head1=tmp;
37
38          tmp=new arcNode;// 建立入度表
39          tmp->adjvex=v1;
40          tmp->nextarc=lg.vertex[v2].head2;
41          lg.vertex[v2].head2=tmp;
42      }
43  }
```





c. 计算入度、出度

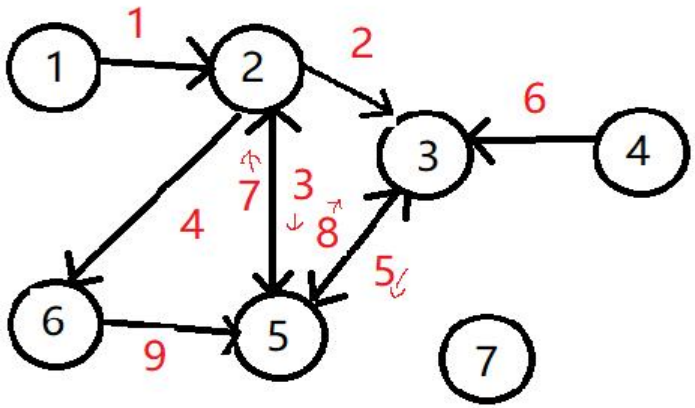
```
int main(){
    int i,start,end;
    arcNode *tmp;
    while(1){
        cin>>lg.vernum;//顶点数量
        if(lg.vernum==0)break;
        createLG();
        for(i=1;i<=lg.vernum;i++){
            start=0;
            tmp=lg.vertex[i].head1;
            while(tmp!=NULL){
                start++;//根据出边表计算出度
                tmp=tmp->nextarc;
            }
            cout<<start<<" ";
        }
        cout<<endl;
        for(i=1;i<=lg.vernum;i++){
            end=0;
            tmp=lg.vertex[i].head2;
            while(tmp!=NULL){
                end++;//根据入边表计算入度
                tmp=tmp->nextarc;
            }
            cout<<end<<" ";
        }
        cout<<endl;
    }
    return 0;
}
```

6.2.3.3 邻接表的实现-方法2（数组模拟）

静态建立的邻接表，也称为链式前向星建表法

对如图的9条边进行编号，第一条边是1以此类推编号[1~9]，
然后我们要知道两个变量的含义：

Next，表示与这个边起点相同的上一条边的编号。
head[i]数组，表示以 i 为起点的最后一条边的编号。
head数组一般初始化为-1或0



根据输入的数据就可以写出下面的过程：

1 对于1 2 这条边：　edge[1].to = 2;　edge[1].next = -1;　head[1] = 1;

2 对于2 3 这条边：　edge[2].to = 3;　edge[2].next = -1;　head[2] = 2;

3 对于2 5这条边：　edge[3].to = 5;　edge[3].next = 2;　head[3] = 3;

4 对于2 6这条边：　edge[4].to = 6;　edge[4].next = 3;　head[1] = 4;

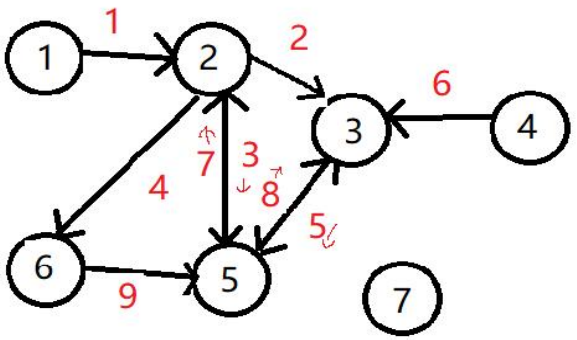
5 对于3 5 这条边：　edge[5].to = 5;　edge[5].next = -1;　head[4] = 5;

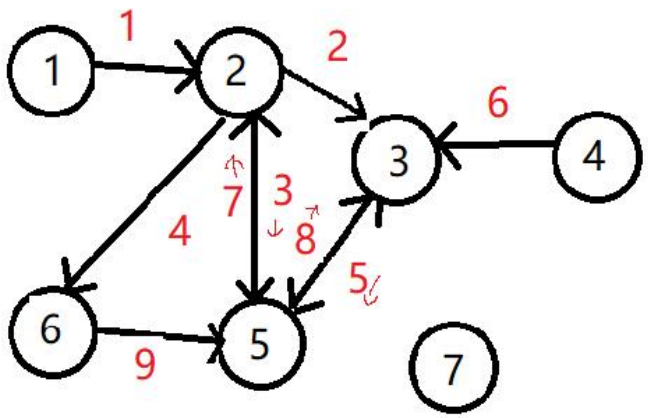
6 对于4 3 这条边：　edge[6].to = 3;　edge[6].next = -1;　head[1] = 6;

7 对于5 2 这条边：　edge[7].to = 2;　edge[7].next = -1;　head[4] = 7;

8 对于5 3 这条边：　edge[8].to = 3;　edge[8].next = 7;　head[4] = 8;

9 对于6 5 这条边：　edge[9].to = 5;　edge[9].next = -1;　head[4] = 9;

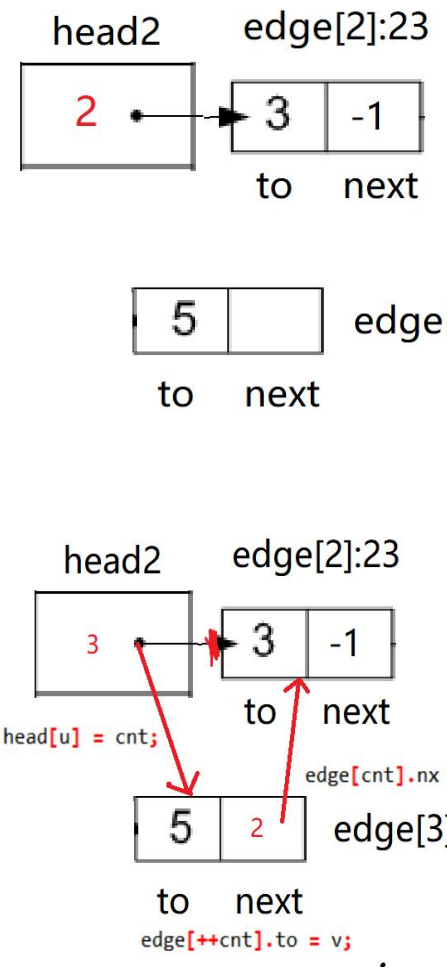




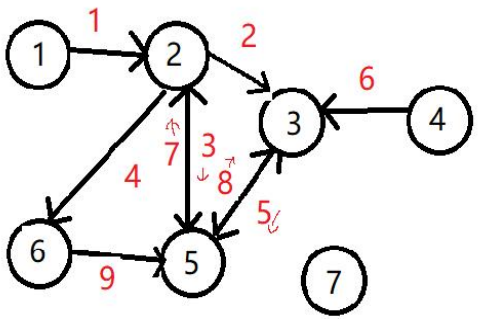

```
4   int head[maxn];//数组，表示以
5   int cnt;//边的编号,表示第几条
6   int rudu[maxn], chudu[maxn];
7   struct node {
8       int to;//边的另外一个顶点
9       int  nx;//Next表示与这个
10  }edge[maxn];//定义边节点
11  void add(int u, int v) {//建
12      edge[++cnt].to = v;
13      edge[cnt].nx = head[u];
14      head[u] = cnt;
15  }
```

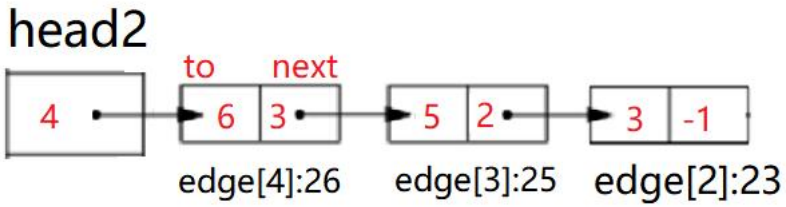
```
1   7
2   1 2
3   edge[1].to=2  edge[1].next=  -1  head[1]=1
4   2 3
5   edge[2].to=3  edge[2].next=  -1  head[2]=2
6   2 5
7   edge[3].to=5  edge[3].next=  2  head[2]=3
8   2 6
9   edge[4].to=6  edge[4].next=  3  head[2]=4
10  3 5
11  edge[5].to=5  edge[5].next=  -1  head[3]=5
12  4 3
13  edge[6].to=3  edge[6].next=  -1  head[4]=6
14  5 2
15  edge[7].to=2  edge[7].next=  -1  head[5]=7
16  5 3
17  edge[8].to=3  edge[8].next=  7  head[5]=8
18  6 5
19  edge[9].to=5  edge[9].next=  -1  head[6]=9
```

```cpp
#include<bits/stdc++.h>
using namespace std;
const int maxn = 1e2 + 10;
int head[maxn];//数组，表示以 i 为起点的最后一条边的编号
int cnt;//边的编号,表示第几条边
int rudu[maxn], chudu[maxn];
struct node {
    int to;//边的另个一个顶点
    int  nx;//Next表示与这个边起点相同的上一条边的编号。
}edge[maxn];//定义边节点

void add(int u, int v) {//建立邻接表
    edge[++cnt].to = v;
    edge[cnt].nx = head[u];//头插法，指向头结点指向的边
    head[u] = cnt;//头节点指向当前插入的边
}

void init() {
    memset(head, -1, sizeof(head));
    memset(rudu, 0, sizeof(rudu));
    memset(chudu, 0, sizeof(chudu));
    cnt = 0;
}
```



```
25  int main()
26  {
27      int n;
28      while (~scanf("%d", &n) && n) {
29          int u, v;
30          init();//初始化
31          while (~scanf("%d %d", &u, &v) && (u + v)) {
32              add(u, v);//建立邻接表
33          }
34          for (int i = 1; i <= n; i++) {//仔细理解for循环
35              for (int j = head[i]; j != -1; j = edge[j].nx) {
36                  rudu[i]++;//从头结点开始，最后一条边的next为-
37                  chudu[edge[j].to]++;
38              }
39          }
40          for (int i = 1; i <= n; i++) {
41              if (i != 1) printf(" ");
42              printf("%d", rudu[i]);
43          }
44          printf("\n");
45          for (int i = 1; i <= n; i++) {
46              if (i != 1) printf(" ");
47              printf("%d", chudu[i]);
48          }
49          printf("\n");
50      }
51      return 0;
```





6.2.3.3 邻接表的实现-方法3（vector）

在使用数组的时候，必须指定数组的长度，一旦配置了就不能改变了，通常我们的做法是尽量配置一个大的空间，以免不够用，这样做的缺点是比较浪费空间，预估空间不当会引起很多不便。

STL实现了一个Vector容器，该容器就是来改善数组的缺点。vector是一个动态空间，随着元素的加入，它的内部机制会自行扩充以容纳新元素。因此，vector的运用对于内存的合理利用与运用的灵活性有很大的帮助，再也不必因为害怕空间不足而一开始就配置一个大容量数组了，vector是用多少就分配多少。

Vector介绍

使用vector容器之前必须加上<vector>头文件：

```
#include<vector>;
```

vector成员函数

c.push_back(elem)在尾部插入一个elem数据。

```
vector<int> v;
v.push_back(1);
```

c.pop_back()删除末尾的数据。

```
vector<int> v;
v.pop_back();
```

c.assign(beg,end)将[beg,end)一个左闭右开区间的数据赋值给c。

Vector介绍

clear() 清空当前的vector
max_size() 得到vector最大可以是多大
empty() 判断vector是否为空
size() 当前使用数据的大小

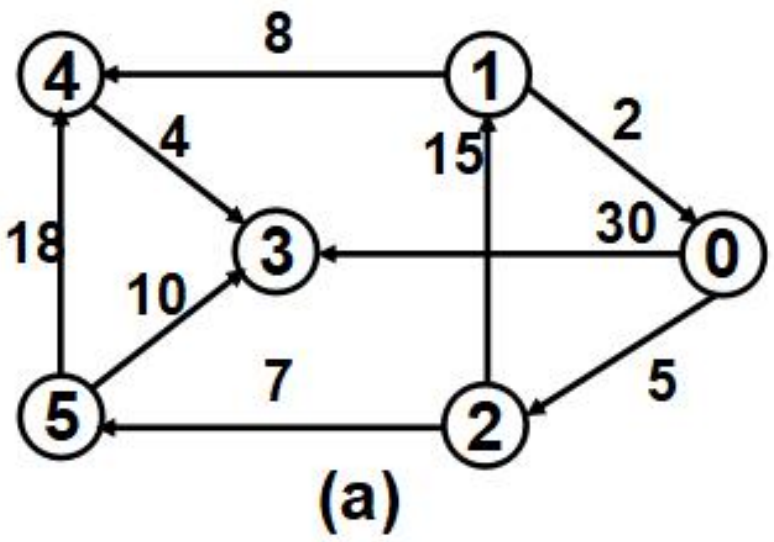
at 得到编号位置的数据
begin 得到数组头的指针
end 得到数组的最后一个单元+1的指针
front 得到数组头的引用
back 得到数组的最后一个单元的引用

capacity 当前vector分配的大小
resize 改变当前使用数据的大小，如果它比当前使用的大，者填充默认值
reserve 改变当前vecotr所分配空间的大小
erase 删除指针指向的数据项

Vector存图

如图，输入：顶点数n 边数m，接着输入m条边的顶点和权值
输出：顶点0到每个顶点的最短路径
样例输入

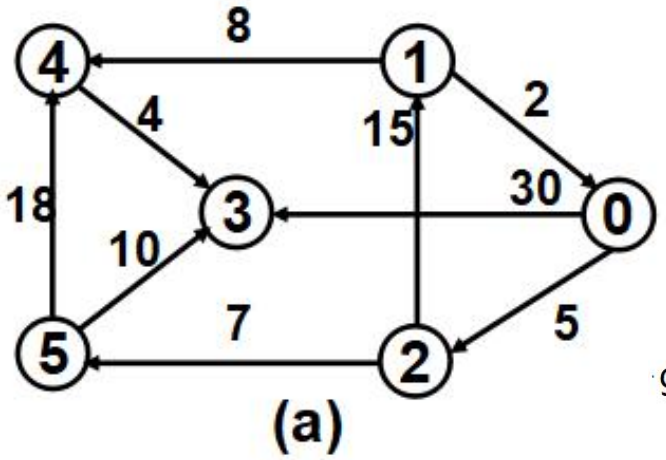
6 9
0 2 5
0 3 30
1 0 2
1 4 8
2 1 15
2 5 7
4 3 4
5 3 10
5 4 18
0 4



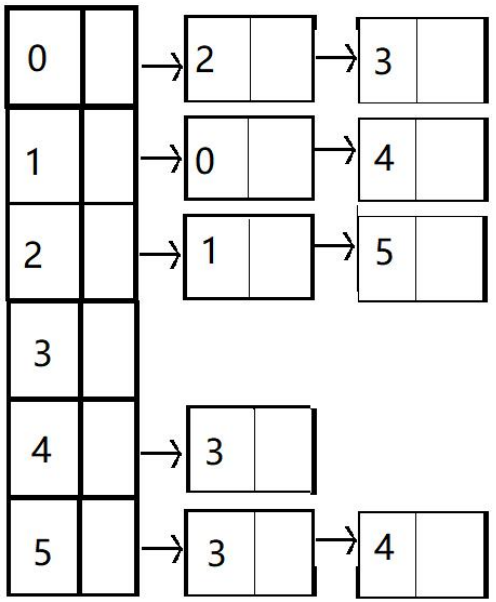
样例输出

28

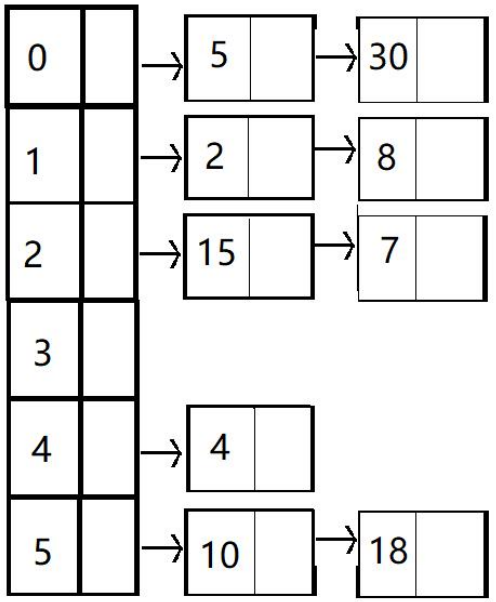
Vector存图



g[MAXN] g[u].push_back(v)

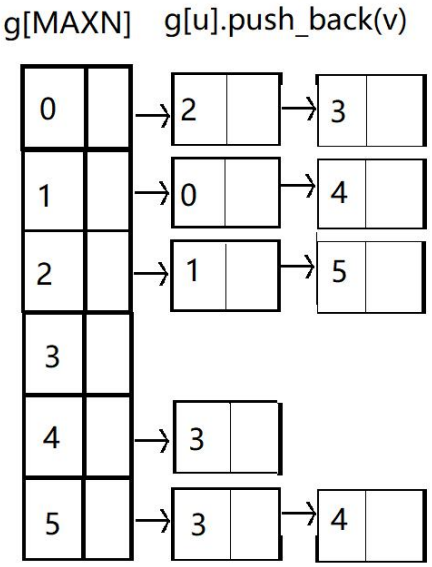


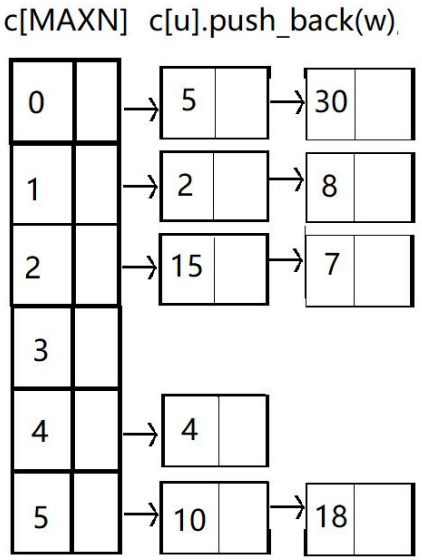
c[MAXN] c[u].push_back(w)



Vector存图

```
1   const int MAXN=100;
2   vector<int>g[MAXN], c[MAXN];
3   cin>>n>>m;
4   //队尾添加元素:  vec.push_back();
5   //队尾删除元素:  vec.pop_back();
6   for(int i=0;i<m;i++){
7       cin>>u>>v>>w;
8       g[u].push_back(v);//u是邻接表出来的端点，uv是一条边
9       c[u].push_back(w);//存权值
10  }
```





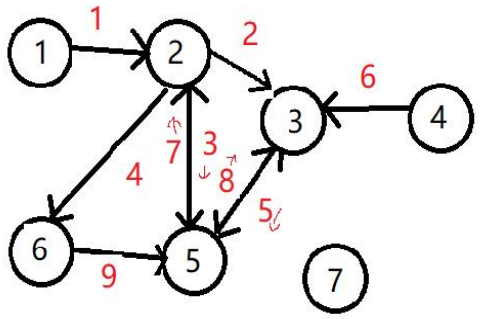
6.2.3.3 邻接表的实现-方法3（vector）

```
1  #include<iostream>
2  #include <cstring>
3  #include <vector>
4  using namespace std;
5  const int MAXN=105;
6  vector<int> G[MAXN];//定义向量
7  int indeg[MAXN];
8  int main(){
9      int n;
10     int u,v,od,id;
11     while(cin>>n){
12         if(n==0)break;
13         for (int i = 1; i <= n; ++ i)
14             G[i].clear(), indeg[i] = 0;//初始化
15         int a, b;
16         while (~scanf("%d %d", &a, &b))
17         {
21         for (int i = 1; i <= n; ++ i)
22         {
31         for (int i = 1; i <= n; ++ i)
32             printf("%d%c", indeg[i], i == n ? '\n' : ' ');
33     }
34 }
```

6.2.3.3 邻接表的实现-方法3（vector）

```
15      int a, b;
16      while (~scanf("%d %d", &a, &b))
17      {
18          if (a == 0 and b == 0) break;
19          G[a].push_back(b);//插入
20      }
21      for (int i = 1; i <= n; ++ i)
22      {
23          int outdeg = G[i].size();
24          for (int j = 0; j < outdeg; ++ j)
25          {
26              int t = G[i][j];
27              indeg[t] ++;
28          }
29          printf("%d%c", outdeg, i == n ? '\n' : ' ');
30      }
31      for (int i = 1; i <= n; ++ i)
32          printf("%d%c", indeg[i], i == n ? '\n' : ' ');
```

6.2.3.3 邻接表的实现-写法4（常用）




```
2 6
head[2]=4, next[4]=3
3 5
head[3]=5, next[5]=0
4 3
head[4]=6, next[6]=0
5 2
head[5]=7, next[7]=0
5 3
head[5]=8, next[8]=7
6 5
head[6]=9, next[9]=0
```

head值

1	1
2	4
3	5
4	6
5	8
6	9
7	0

next 数组

1	2	3	4	5	6	7	8	9
0	0	2	3	0	0	0	7	0

```
for(int j=head1[i];j!=0;j=next1[j])
    start++;
```

```
void add(int f, int t){
    next1[cnt]=head1[f];//next指向下一条边
                        //(原头结点指向的边)
    head1[f]=cnt;//把当前边插入都头结点后面，头插法
    cnt++;
}
```

```cpp
#include<bits/stdc++.h>
using namespace std;
# define MAXN 101
int head1[101],head2[101];
int next1[101],next2[101];
int cnt;
void add(int f, int t){
    next1[cnt]=head1[f];//next指向下一条边
                        //(原头结点指向的边)
    head1[f]=cnt;//把当前边插入都头结点后面，头插法
    next2[cnt]=head2[t];
    head2[t]=cnt;
    cnt++;
}
int main(){
    int n,i,u,v;
    while(cin>>n){
        cnt=1;
        if(n==0)break;
        memset(head1,0,sizeof(head1));
        memset(head2,0,sizeof(head2));
        memset(next1,0,sizeof(next1));
        memset(next2,0,sizeof(next2));
        while(cin>>u>>v){
            if(u==0&&v==0)break;
            add(u,v);
        }
        int start,end;
        for(i=1;i<=n;i++){
            start=0;
            for(int j=head1[i];j!=0;j=next1[j])
                start++;
            cout<<start<<" ";
        }
        cout<<endl;
        for(i=1;i<=n;i++){
            end=0;
            for(int j=head2[i];j!=0;j=next2[j])
                end++;
            cout<<end<<" ";
        }
        cout<<endl;
    }
```

关于邻接矩阵和邻接表的进一步讨论

1. 存储方式对算法复杂度的影响

时间复杂度： 邻接表里直接存储了边的信息，浏览完所有的边，对有向图来说，时间复杂度复杂度是O(m)，对无向图是O(2×m)。而邻接矩阵是间接存储边，浏览完所有的边，复杂度是$O(n^2)$。

空间复杂度： 邻接表里除了存储m条边所对应的边结点外，还需要一个顶点数组，存储各顶点的顶点信息及各边链表的表头指针；而用邻接矩阵存储图，需要n×n规模的存储单元。当边的数目相对于n×n比较小时，邻接矩阵里存储了较多的无用信息，而用邻接表可以节省较多的存储空间。

2. 在求解问题时可以灵活地存储表示图

在求解实际问题时，有时并不需要严格采用邻接矩阵或邻接表来存储图。

例如，当图中顶点个数确定以后(这里假设顶点序号是连续的)，图的结构就唯一地取决于边的信息，因此可以**把每条边的信息(起点、终点、权值等)存储到一个数组里**，在针对该图进行某种处理时只需要**访问该数组中每个元素即可**。对于一些可以直接针对边进行处理的算法，

今天的课程结束啦……

下课了…
同学们再见！