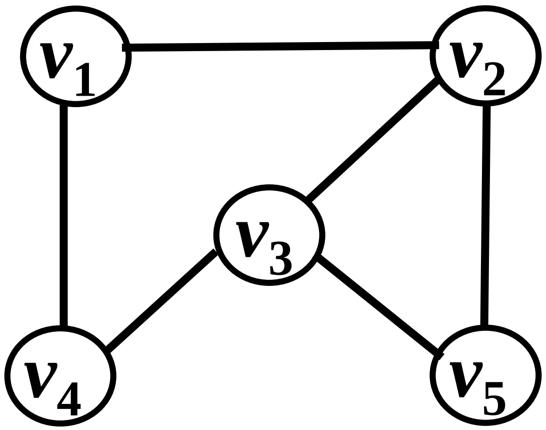
7.2 图的存储结构

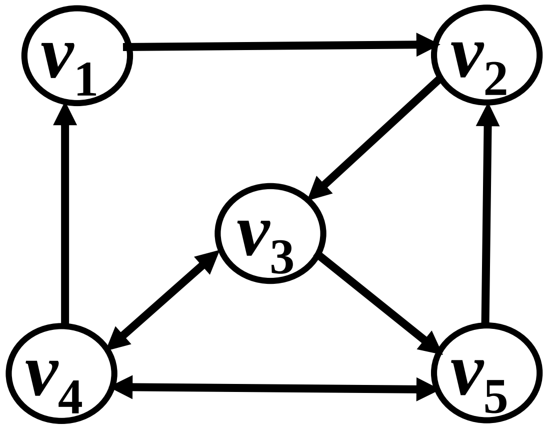
7.2.1 邻接矩阵

7.2.2 邻接表

7.2.3 有向图的十字链表存储表示 （了解）

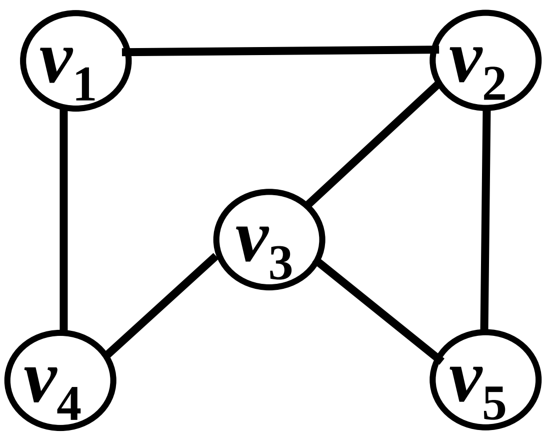
7.2.4 无向图的邻接多重表存储表示（了解）

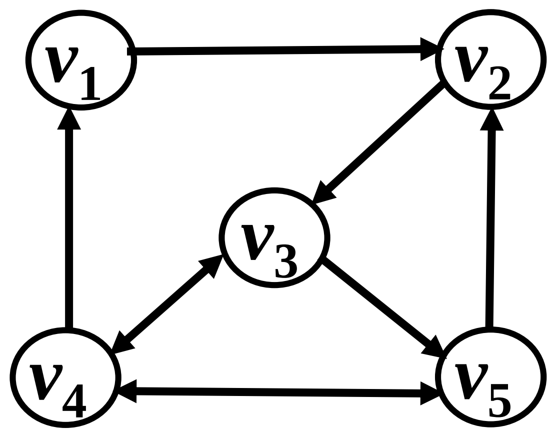




7.2 图的存储结构

- **是否可以采用顺序存储结构存储图(一维数组）？**
  - 图的特点：顶点之间的关系是*m*:*n*，即任何两个顶点之间都可能存在关系（边），无法通过存储位置表示这种任意的逻辑关系，所以，图无法采用顺序存储结构。
- **如何存储图？**
  - 考虑图的定义，图是由顶点和边组成的；
  - 如何存储顶点、如何存储边----顶点之间的关系。



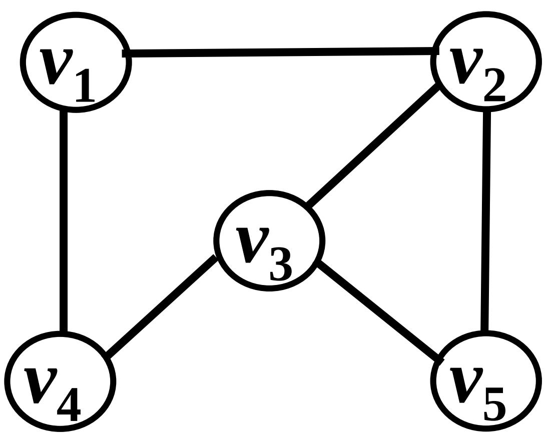


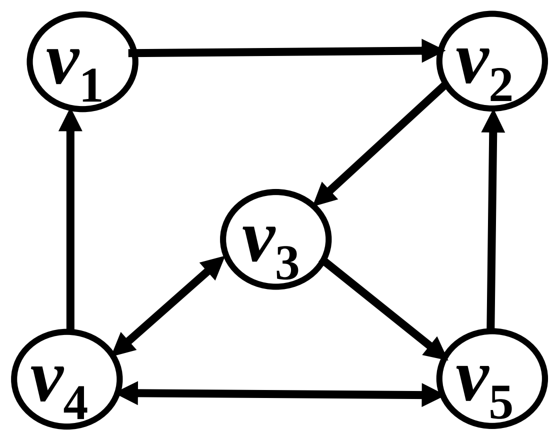
7.2 图的存储结构

邻接矩阵 （*Adjacency Matrix*）表示（数组表示法）

- **基本思想：**
  - 用一个一维数组存储图中顶点的信息，用一个二维数组（称为邻接矩阵）存储图中各顶点之间的邻接关系。
  - 假设图*G*＝(*V*，*E*)有*n*个顶点，则邻接矩阵是一个*n*×*n*的方阵，定义为：

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

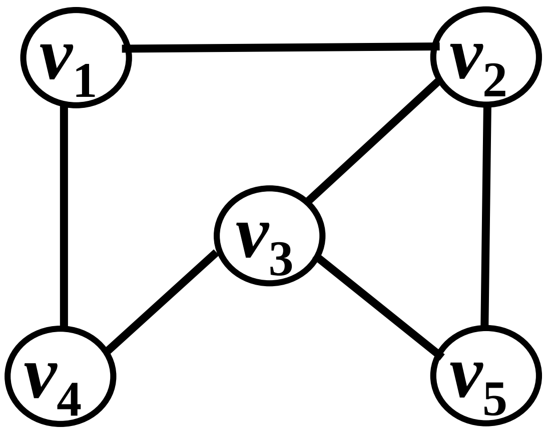




7.2 图的存储结构

邻接矩阵 （***Adjacency Matrix***）表示（数组表示法）

- 无向图的邻接矩阵：



vertex=

v_1	v_2	v_3	v_4	v_5

$$
\textbf{edge} = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array}
\begin{array}{c}
\begin{array}{ccccc} v_1 & v_2 & v_3 & v_4 & v_5 \end{array} \\
\begin{pmatrix}
0 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 0
\end{pmatrix}
\end{array}
$$

- 存储结构特点：
  - 主对角线为 **0** 且一定是对称矩阵；

问题：**1.** 如何求顶点v_i的度？

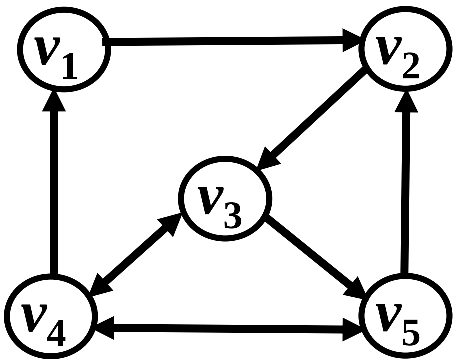
2.如何判断顶点 v_i和 v_j 之间是否存在边？

3.如何求顶点 v_i的所有邻接点？

7.2 图的存储结构

邻接矩阵 （*Adjacency Matrix*）表示（数组表示法）

- 有向图的邻接矩阵：



vertex=

v_1	v_2	v_3	v_4	v_5

$$
\text{edge} = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array}
\begin{array}{c} \begin{array}{ccccc} v_1 & v_2 & v_3 & v_4 & v_5 \end{array} \\
\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix} \end{array}
$$

- 存储结构特点：
  - 有向图的邻接矩阵不一定对称

问题：1.如何求顶点v_i的出度？

2. 如何判断顶点 v_i和 v_j 之间是否存在有向边？

3.如何求邻接自（于）顶点 v_i的所有顶点？

7.如何求邻接到顶点 v_i的所有顶点？

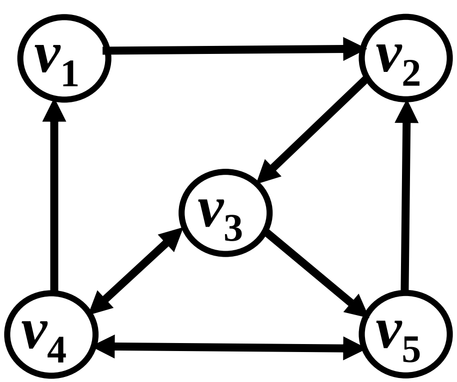
7.2 图的存储结构

邻接矩阵 （*Adjacency Matrix*）表示（数组表示法）

- 存储结构定义：

> 假设图*G*有*n*个顶点e条边，则该图的存储需求为 $O(n+n^2) = O(n^2)$，与边的条数e无关。

```
typedef struct {
    VertexData verlist [NumVertices];   //顶点表
    EdgeData edge[NumVertices][NumVertices];
        //邻接矩阵—边表, 可视为边之间的关系
    int n, e;  //图的顶点数与边数
} MTGraph;
```



vertex

v_1	v_2	v_3	v_4	v_5

$$\text{edge}= \begin{array}{c} \\ \\ \\ \\ \\ \end{array} \begin{array}{ccccc} v_1 & v_2 & v_3 & v_4 & v_5 \end{array}$$

$$\text{edge}=\left[\begin{array}{ccccc} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{array}\right] \begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array}$$

7.2 图的存储结构

- **存储结构的建立----算法实现的步骤：**

1.确定图的顶点个数n和边数e；

2.输入顶点信息存储在一维数组vertex中；

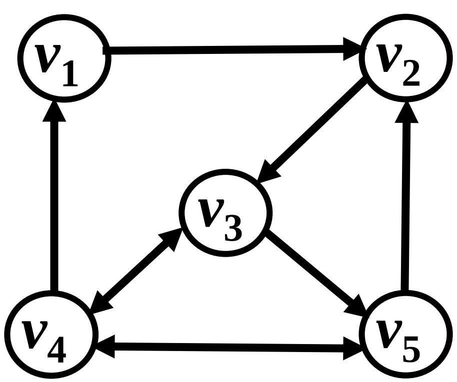
3.初始化邻接矩阵；

7.依次输入每条边存储在邻接矩阵edge中；

 7.1 输入边依附的两个顶点的序号i, j；

 7.2 将邻接矩阵的第i行第j列的元素值置为1；

 7.3 将邻接矩阵的第j行第i列的元素值置为1。



vertex

v_1	v_2	v_3	v_4	v_5

$$\text{edge}=\begin{array}{c} \begin{matrix} v_1 & v_2 & v_3 & v_4 & v_5 \end{matrix} \\ \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix} \end{array} \begin{matrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{matrix}$$

7.2 图的存储结构

- 存储结构的建立算法的实现：

```
void  CreateMGragh (MTGragh *G) //建立图的邻接矩阵
{     int  i, j, k, w;
      cin>>G→n>>G→e;            //1.输入顶点数和边数

      for (i=0; i<G→n; i++)        //2.读入顶点信息，建立顶点表
            G→vertlist[i]=getchar( );

      for (i=0; i<G→n; i++)
            for (j=0;j<G→n;j++)
                  G→edge[i][j]=0;        //3.邻接矩阵初始化

      for (k=0; k<G→e; k++) {      //4.读入e条边建立邻接矩阵
            cin>>i>>j>>w;            // 输入边（i,j）上的权值w
            G→ edge[i][j]=w; G→ edge[j][i]=w;
      }
} //时间复杂度：
```

$T = O(n + n^2 + e)$ 。$T = O(n^2)$

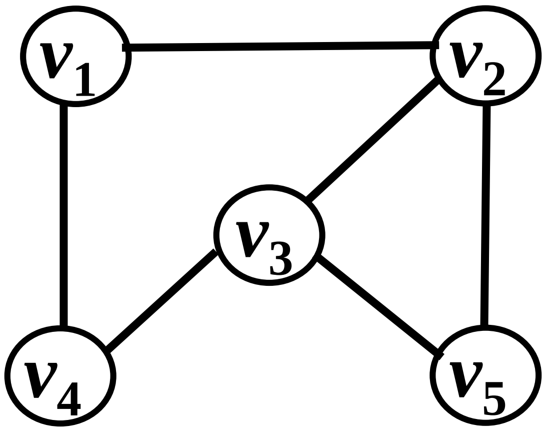
7.2 图的存储结构

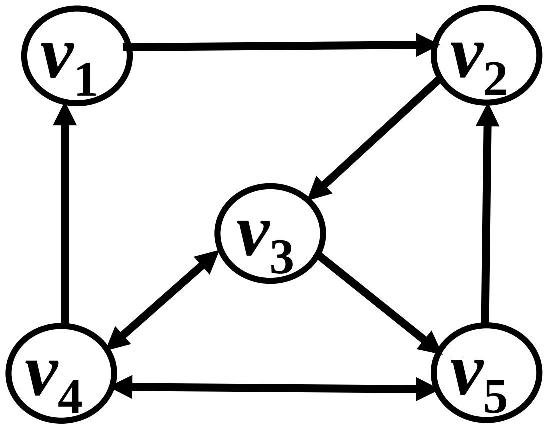
7.2.1 邻接矩阵

7.2.2 邻接表

7.2.3 有向图的十字链表存储表示 （了解）

7.2.4 无向图的邻接多重表存储表示（了解）

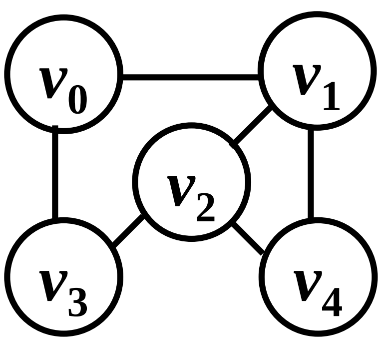


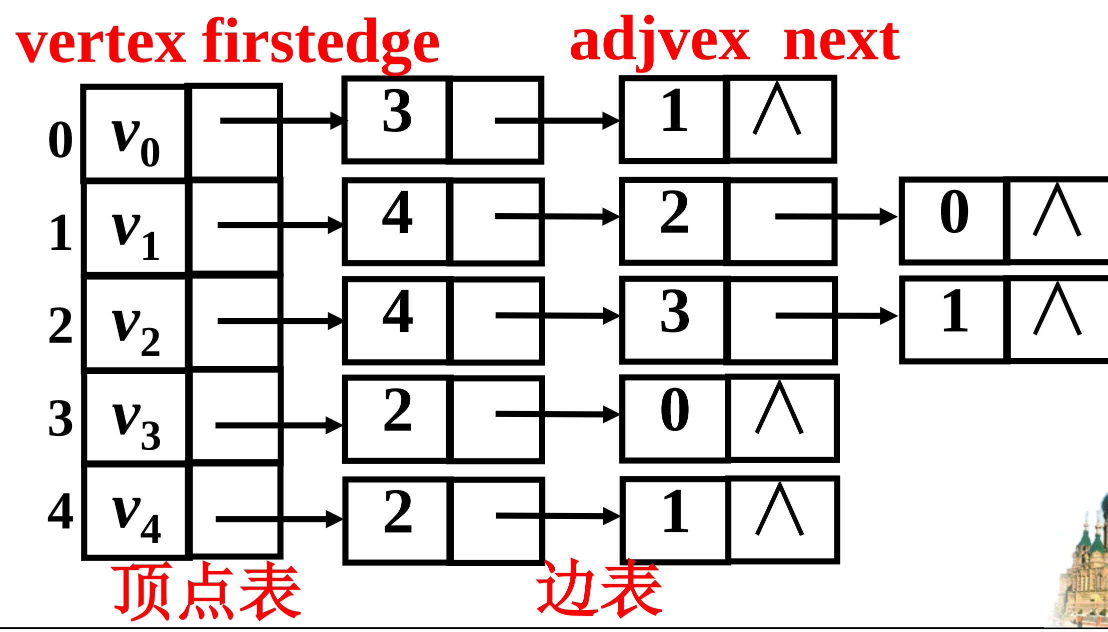


7.2 图的存储结构

邻接表（Adjacency List）表示

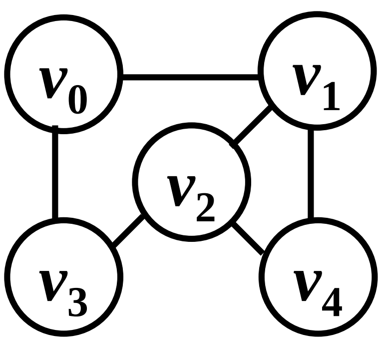
- 无向图的邻接表：
  - 对于无向图的每个顶点v_i，将所有与v_i相邻的顶点链成一个单链表，称为顶点v_i的边表（顶点v_i的邻接表）；
  - 再把所有边表的指针和存储顶点信息的一维数组构成顶点表。

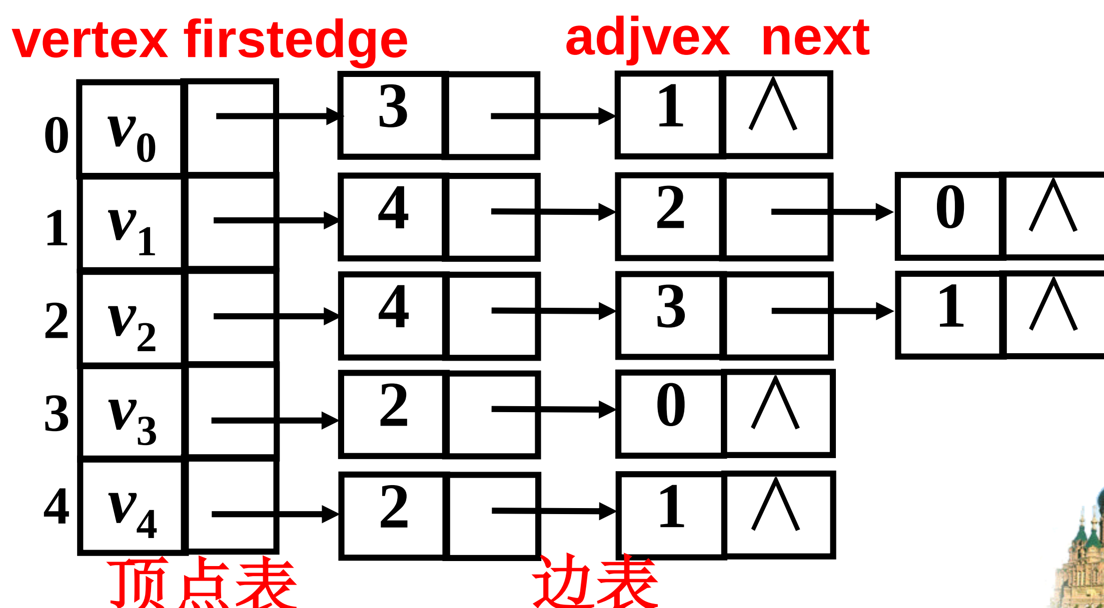




7.2 图的存储结构

- 无向图的邻接表存储的特点：
  - 边表中的结点表示什么?
  - 如何求顶点 v_i的度?
  - 如何判断顶点v_i和顶点v_j之间是否存在边?
  - 如何求顶点 v_i的所有邻接点?
  - 空间需求O(n+2e)

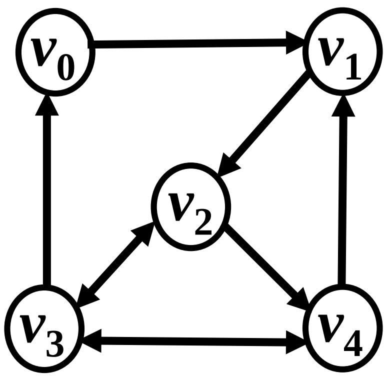


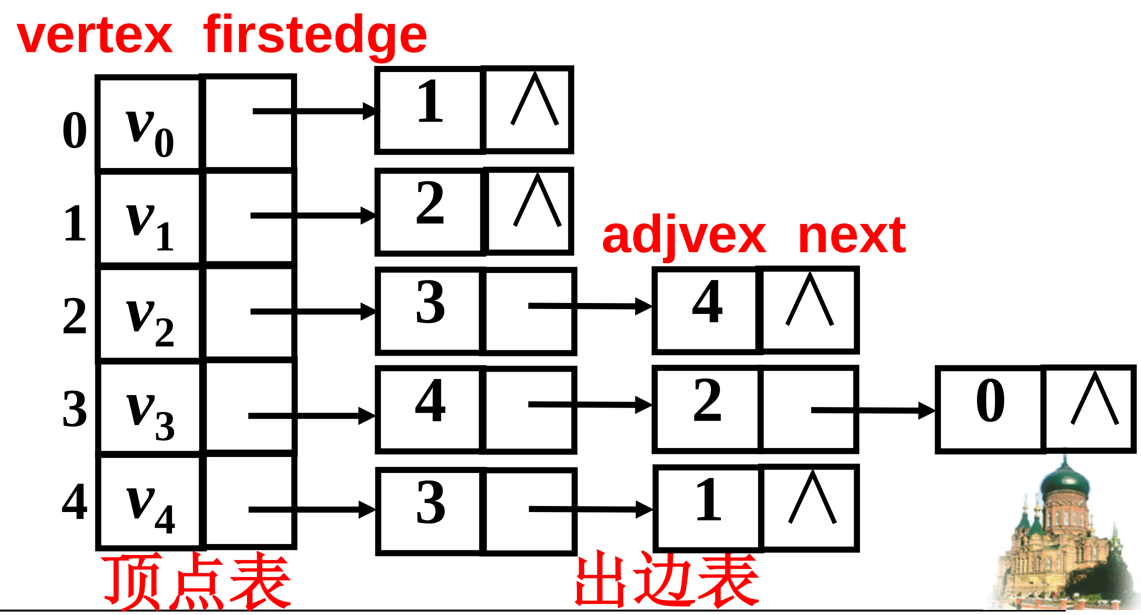


7.2 图的存储结构

邻接表（Adjacency List）表示

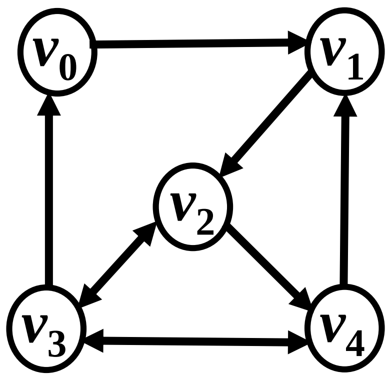
- 有向图的邻接表---正邻接表
  - 对于有向图的每个顶点v_i，将邻接于v_i的所有顶点链成一个单链表，称为顶点v_i的出边表；
  - 再把所有出边表的指针和存储顶点信息的一维数组构成顶点表。

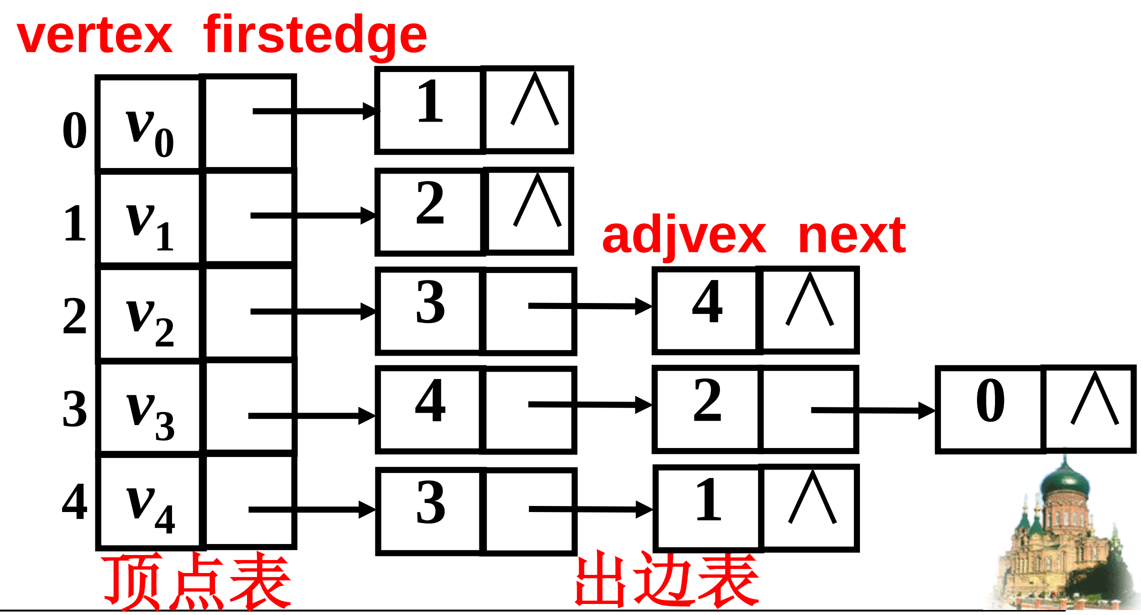




7.2 图的存储结构

- 有向图的正邻接表的存储特点
  - 出边表中的结点表示什么？
  - 如何求顶点 v_i的出度？如何求顶点 v_i的入度？
  - 如何判断顶点 v_i和顶点v_j之间是否存在有向边?
  - 如何求邻接自顶点 v_i的所有顶点?
  - 如何求邻接到顶点 v_i的所有顶点?
  - 空间需求:O(n+e)

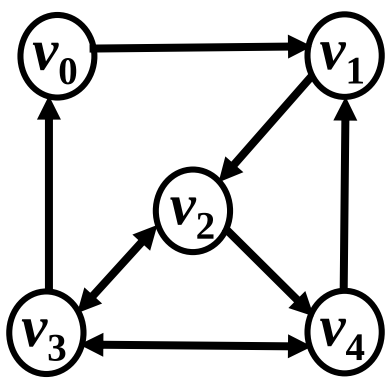


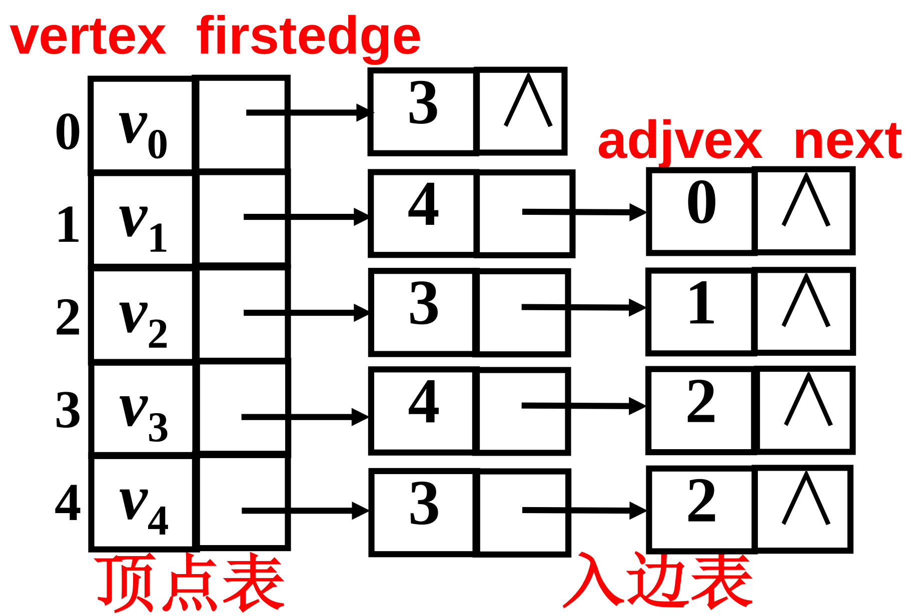


7.2 图的存储结构

邻接表（Adjacency List）表示

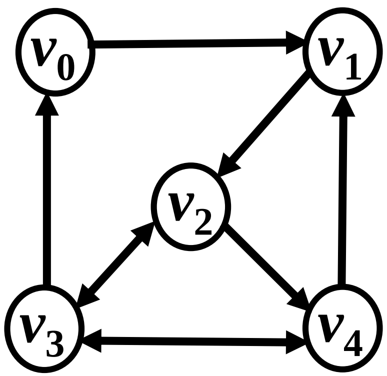
- 有向图的邻接表----逆邻接表
  - 对于有向图的每个顶点v_i，将邻接到v_i的所有顶点链成一个单链表，称为顶点v_i的入边表；
  - 再把所有入边表的指针和存储顶点信息的一维数组构成顶点表。

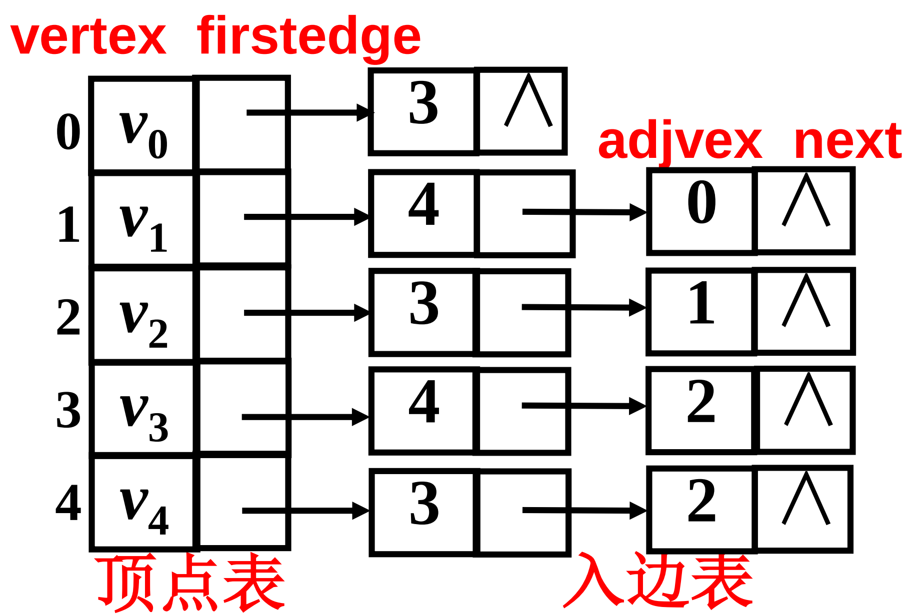




7.2 图的存储结构

- 有向图的逆邻接表的存储特点
  - 入边表中的结点表示什么？
  - 如何求顶点 v_i的入度？如何求顶点 v_i的出度？
  - 如何判断顶点 v_i和顶点v_j之间是否存在有向边?
  - 如何求邻接到顶点 v_i的所有顶点?
  - 如何求邻接自顶点 v_i的所有顶点?
  - 空间需求:O(n+e)





7.2 图的存储结构

- 邻接表存储结构的定义

```
typedef struct node {//边表结点
    int adjvex;          //邻接点域（下标）
    EdgeData cost;           //边上的权值
    struct node *next; //下一边链接指针
} EdgeNode;
typedef struct {            //顶点表结点
    VertexData  vertex; //顶点数据域
    EdgeNode * firstedge;//边链表头指针
} VertexNode;
typedef struct {         //图的邻接表
   VertexNode vexlist [NumVertices];
    int n, e;             //顶点个数与边数
} AdjGraph;
```

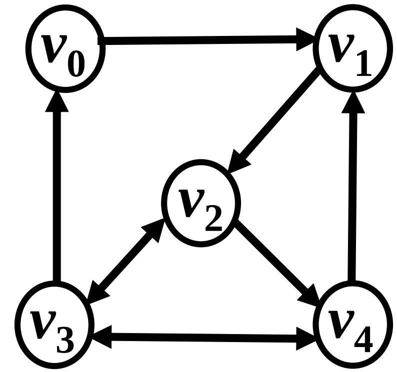
边表结点

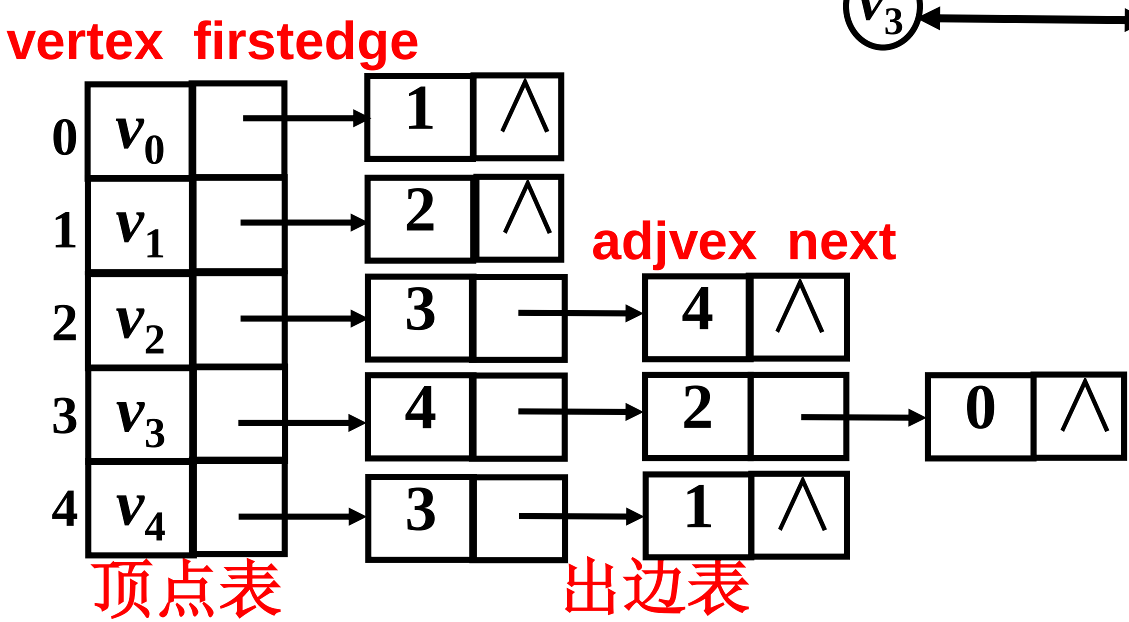
adjvex	cost	next

顶点表结点

vertex	firstedge

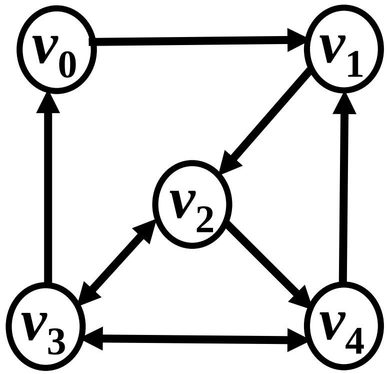
7.2 图的存储结构


v0
v1
v2
v3
v4


vertex firstedge
0 v0
1 v1
2 v2
3 v3
4 v4
1 Λ
2 Λ
3
4
3
adjvex next
4 Λ
2
1 Λ
0 Λ
顶点表
出边表

7.2 图的存储结构

- 邻接表存储结构建立算法实现的步骤：



1. 确定图的顶点个数和边的个数；

2. 建立顶点表：

   2.1 输入顶点信息；

   2.2 初始化该顶点的边表；

3. 依次输入边的信息并存储在边表中；

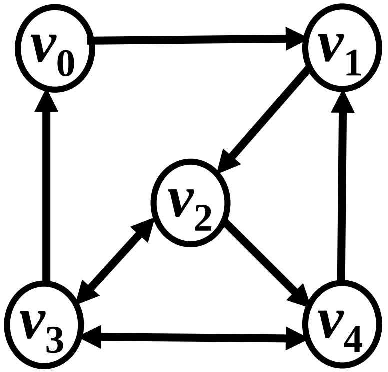
   3.1 输入边所依附的两个顶点的序号tail和head和权值w；

   3.2 生成邻接点序号为head的边表结点p；

   3.3 设置边表结点p；

   3.4 将结点p插入到第tail个边表的头部；

7.2 图的存储结构



- 邻接表存储结构建立算法的实现：

```
void CreateGraph (AdjGraph G)
{  cin >> G.n >> G.e;                    //1.输入顶点个数和边数
   for ( int i = 0; i < G.n; i++) {      //2.建立顶点表
      cin >> G.vexlist[i].vertex;        //2.1输入顶点信息
      G.vexlist[i].firstedge = NULL; }   //2.2边表置为空表
   for ( i = 0; i < G.e; i++) {          //3.逐条边输入,建立边表
      cin >> tail >> head >> weight;           //3.1输入
      EdgeNode * p = new EdgeNode;             //3.2建立边结点
      p→adjvex = head;  p→cost = weight; //3.3设置边结点
      p→next = G.vexlist[tail].firstedge;      //3.4链入第 tail 号链表的前端
      G.vexlist[tail].firstedge = p;
      p = new EdgeNode;
      p→adjvex = tail;  p→cost = weight;
      p→next = G.vexlist[head].firstedge; //链入第 head 号链表的前端
     G.vexlist[head].firstedge = p; }
} //时间复杂度：O(2e+n)
```

7.2 图的存储结构

图的存储结构的比较——邻接矩阵和邻接表

	邻接表	邻接矩阵
空间复杂度	无向图 $O(\lvert V\rvert + 2\lvert E\rvert)$； 有向图$O(\lvert V\rvert + \lvert E\rvert)$	$O(\lvert V\rvert^2)$
适合用于	存储稀疏图	存储稠密图
表示方式	不唯一	唯一
计算度/出度/入度	计算有向图的度、入（出）度不方便，其余很方便	必须遍历对应行或列
找相邻的边	找有向图的入（出）边不方便，其余很方便	必须遍历对应行或列

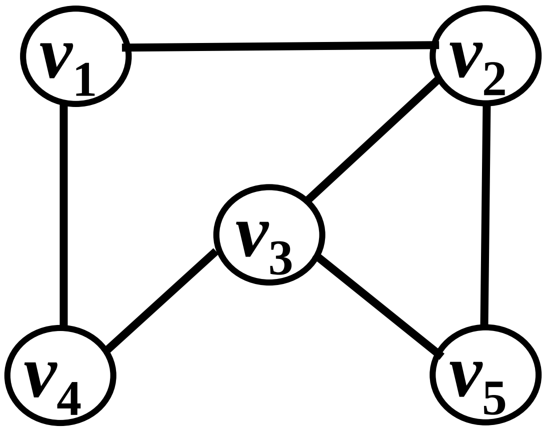
7.2 图的存储结构

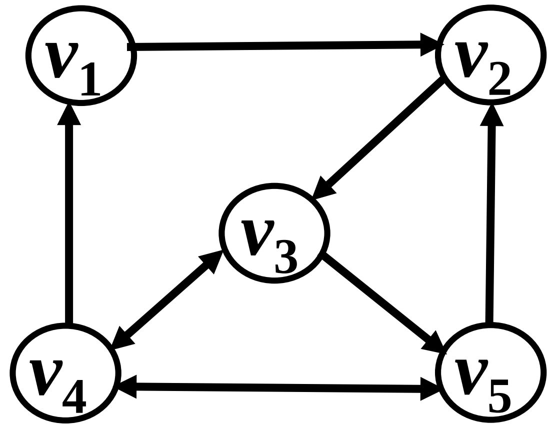
7.2.1 邻接矩阵

7.2.2 邻接表

7.2.3 有向图的十字链表存储表示 （了解）

7.2.4 无向图的邻接多重表存储表示（了解）





◆ 十字链表(有向图)

- 十字链表是有向图的另一种链式存储结构。
- 将有向图的邻接表和逆邻接表结合起来的结构。
- 在十字链表中有两种结点：
  - 弧结点：存储每一条弧的信息，用链表链接在一起。

弧结点结构:

tailvex	headvex	hlink	tlink	info

tailvex：弧尾　**headvex**：弧头

hlink：指向弧头相同的下一条弧

tlink：指向弧尾相同的下一条弧

Info：弧的其他信息

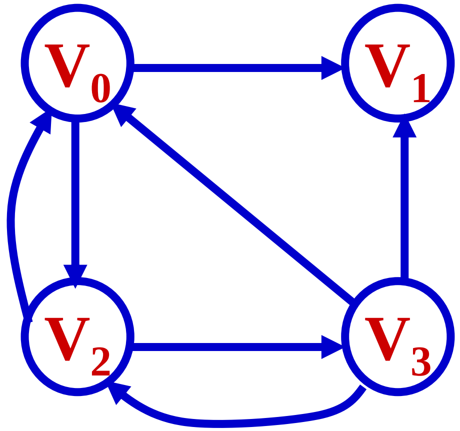
- 顶点结点：存储每一个顶点的信息，使用一维数组来存储。

顶点结点结构:

data	firstin	firstout

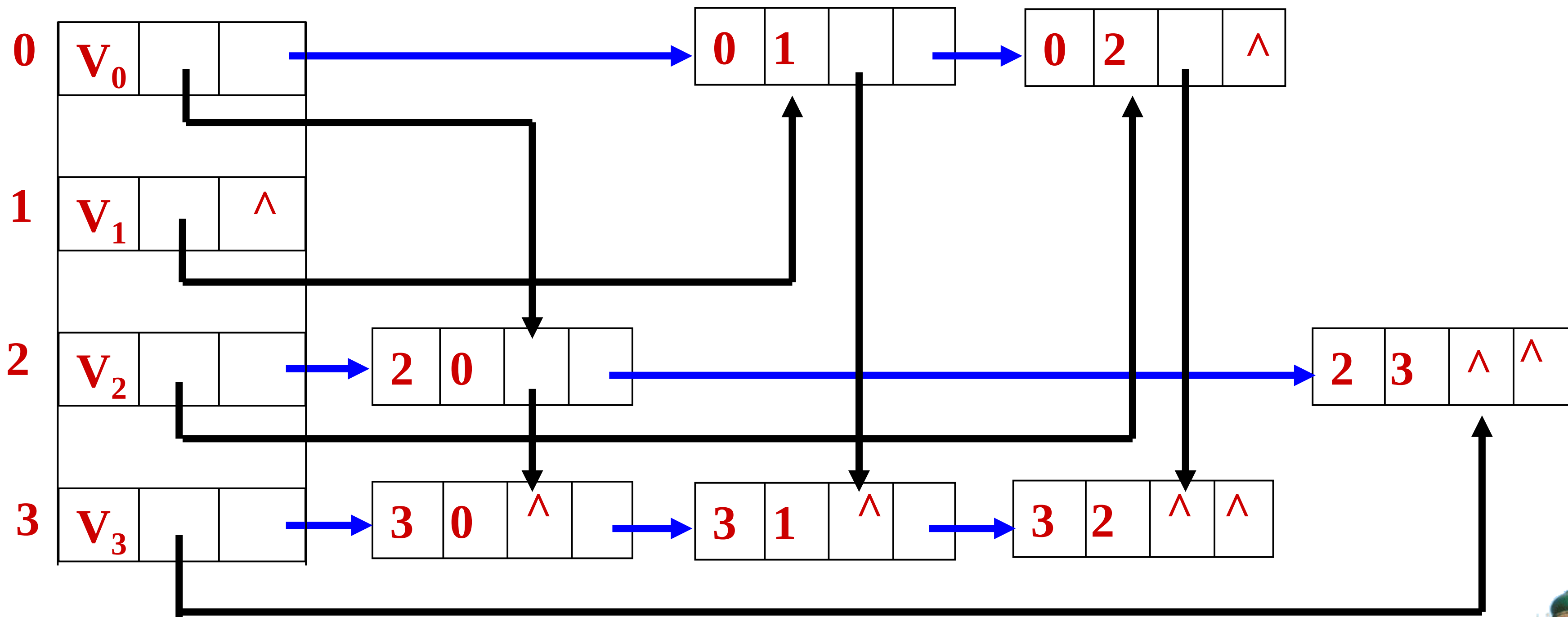
data	firstin	firstout

tailvex	headvex	hlink	tlink	info

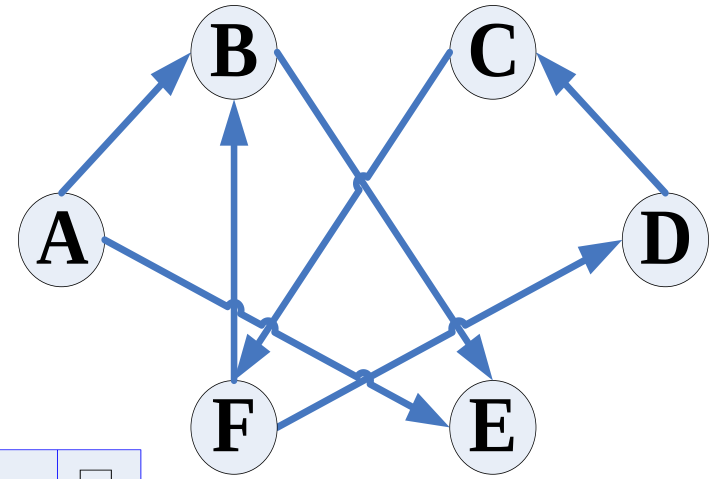


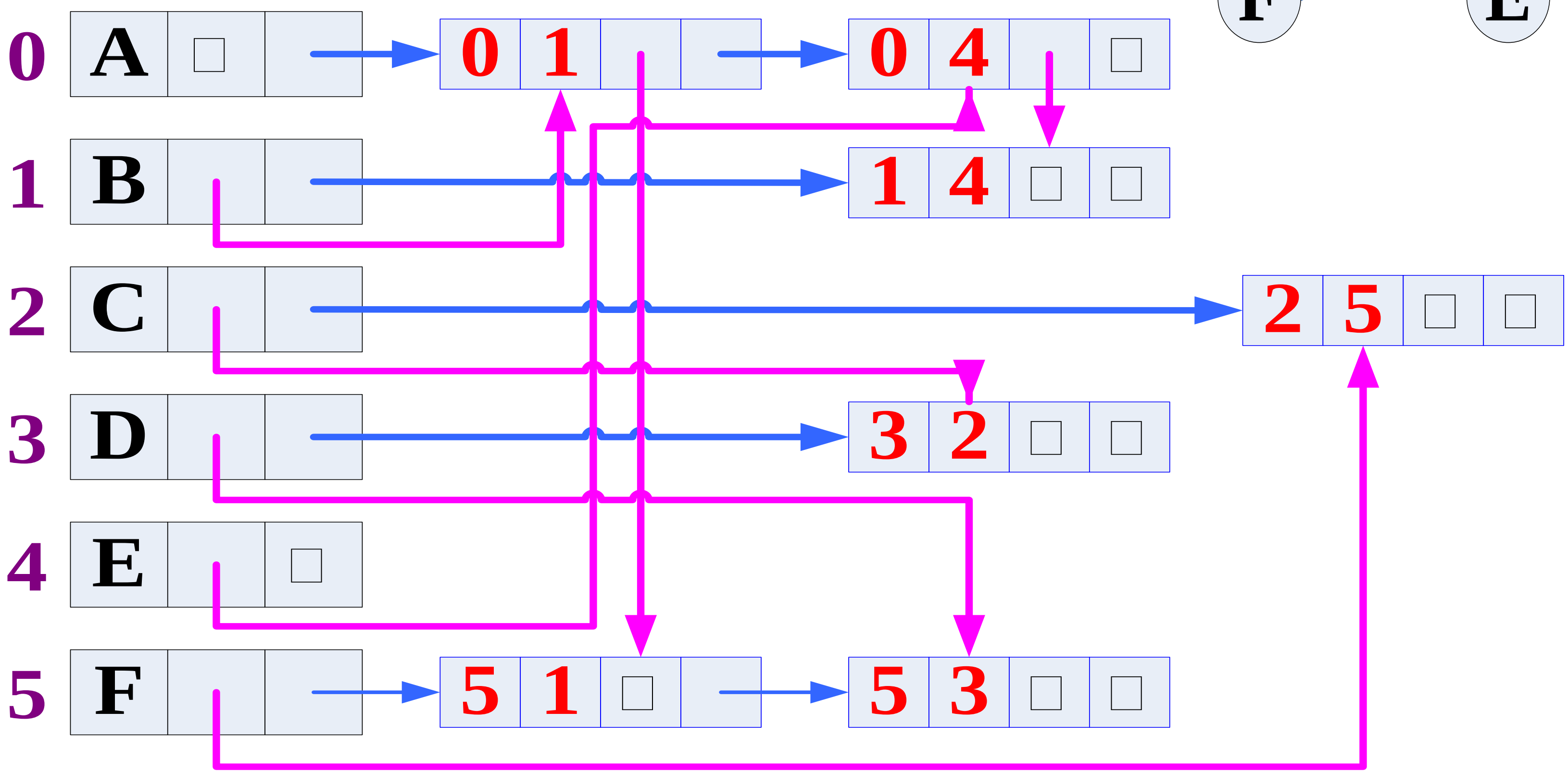
顶点结点

弧结点



➢十字链表中既容易找到以v_i为尾的弧，也容易找到以v_i为头的弧，因而容易求得顶点的出度和入度。





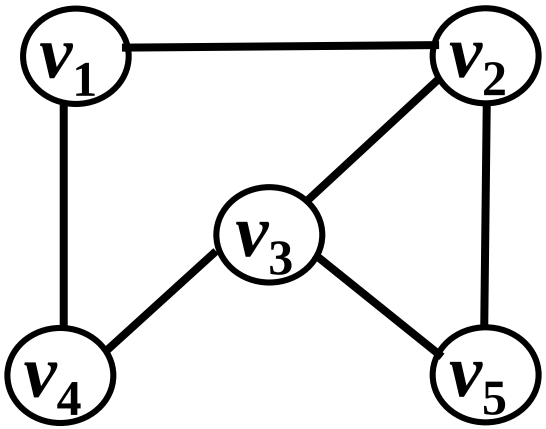
7.2 图的存储结构

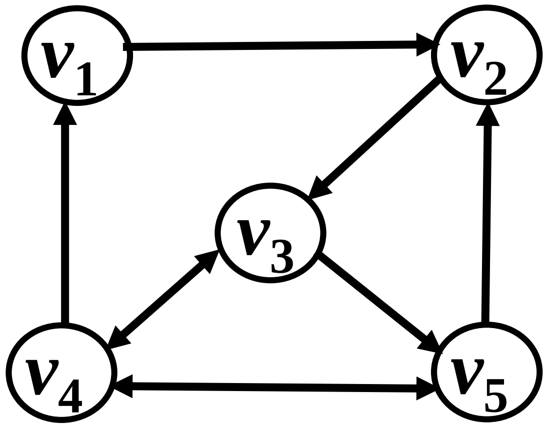
7.2.1 邻接矩阵

7.2.2 邻接表

7.2.3 有向图的十字链表存储表示 （了解）

7.2.4 无向图的邻接多重表存储表示（了解）





◆ 邻接多重表(无向图)

- 邻接多重表是无向图的另一种链式表示结构。
- 和十字链表类似。邻接多重表中，每一条边用一个结点表示。
- 在邻接多重表中有两种结点：
  - 边结点：存储每一条边的信息，用链表链接在一起。

边结点结构:

mark	ivex	ilink	jvex	jlink	info

Mark：标志域，用以标志该条边是否被搜索过

Ivex和jvex：该边依附的两个顶点在图中的位置

Ilink和jlink：链域，指向下一条依附于顶点的边

Info：弧的其他信息

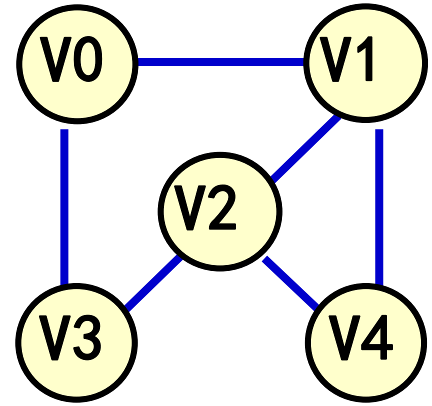
  - 顶点结点：存储每一个顶点的信息，使用一维数组来存储。

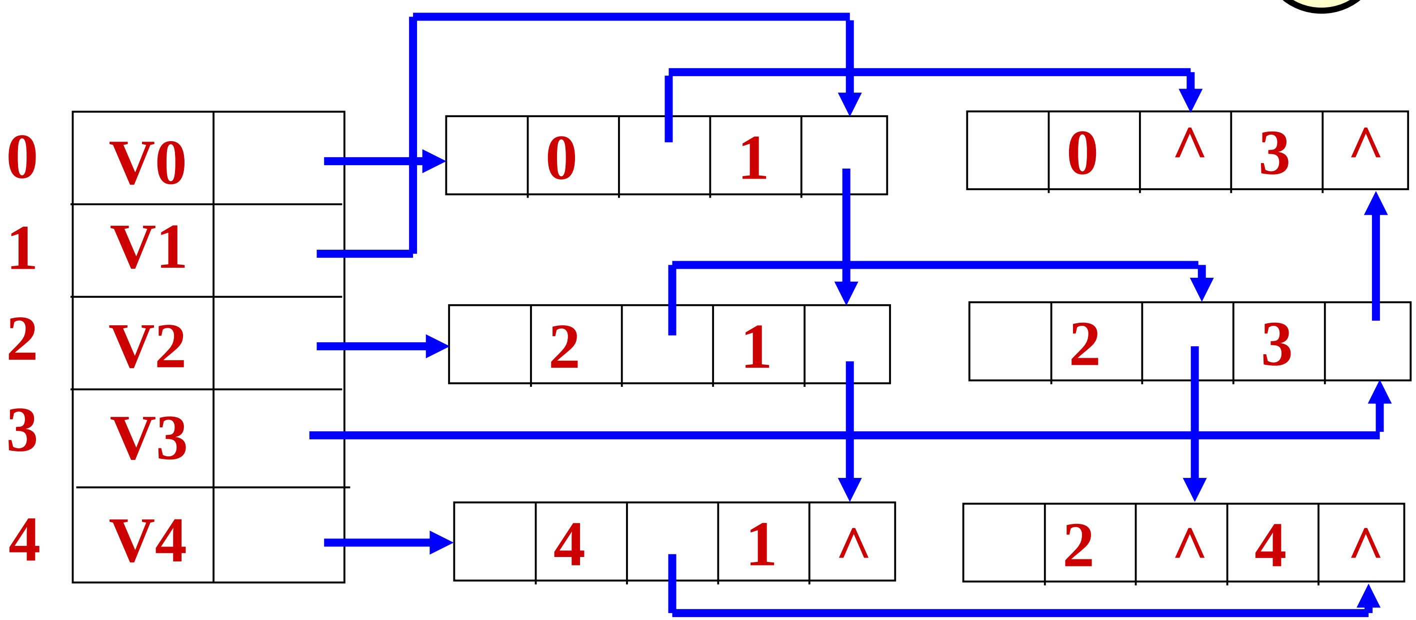
顶点结点结构:

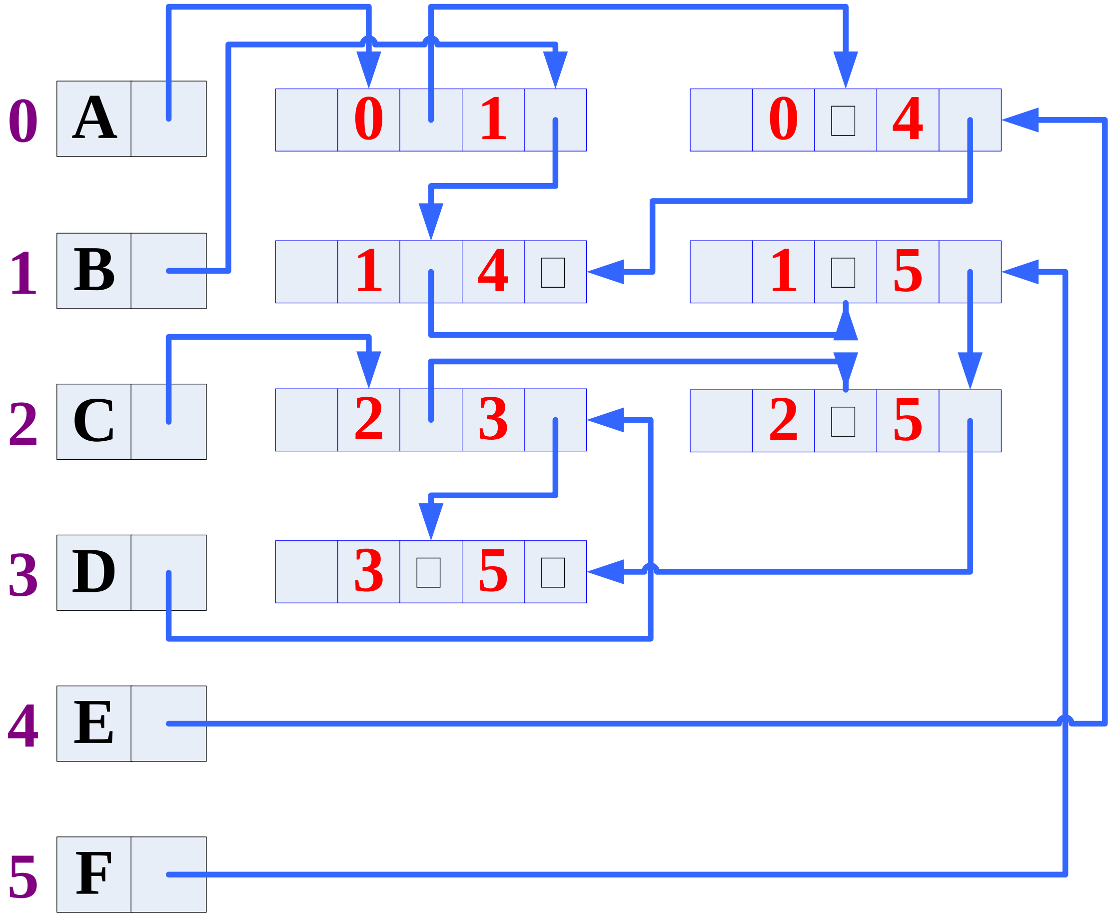
data	firstedge

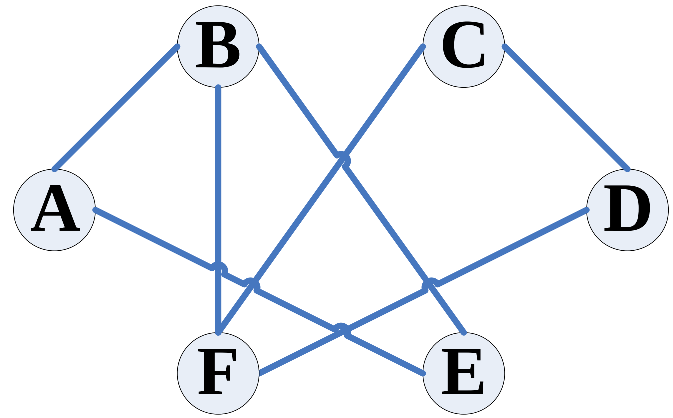
data	firstedge

mark	ivex	ilink	jvex	jlink






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


B
C
A
D
F
E

7.3 图的搜索（遍历）

- 图的遍历（图的搜索）
  - 从图中某一顶点出发，对图中所有顶点访问一次且仅访问一次。
  - 访问：抽象操作，可以是对结点进行的各种处理
- 图结构的复杂性
  - 在线性表中，数据元素在表中的编号就是元素在序列中的位置，因而其编号是唯一的；
  - 在树结构中，将结点按层序编号，由于树具有层次性，因而其层序编号也是唯一的；
  - 在图结构中，任何两个顶点之间都可能存在边，顶点是没有确定的先后次序的，所以，顶点的编号不唯一。

7.3 图的搜索（遍历）

- 图的遍历要解决的关键问题
  - 在图中，如何选取遍历的起始顶点？
    - 解决办法：从编号小的顶点(任取一顶点，适合编程）开始 。
  - 从某个起点始可能到达不了所有其它顶点，怎么办？
    - 解决办法：多次调用遍历图（起点选没有用过的）的算法。
  - 图中可能存在回路，且图的任一顶点都可能与其它顶点“相通”，在访问完某个顶点之后可能会沿着某些边又回到了曾经访问过的顶点。如何避免某些顶点可能会被重复访问？
    - 解决办法：设访问标志数组visited[n] 。
  - 在图中，一个顶点可以和其它多个顶点相连，当这样的顶点访问过后，如何选取下一个要访问的顶点？
    - 解决办法：深度优先搜索（**Depth First Search**）和广度优先搜索（**Breadth First Search**）。

7.3.1 深度优先搜索DFS (Depth First Search)

- 深度优先遍历----类似于树结构的先序遍历

(1)首先访问图中某一个顶点V_i，以该顶点为出发点；

(2)任选一个与顶点V_i邻接的未被访问的顶点V_j；访问V_j；

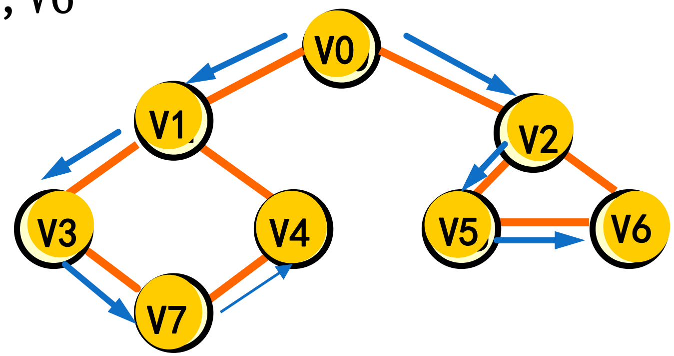
(3)以V_j为新的出发点继续进行深度优先搜索，直至图中所有和V_i有路径的顶点均被访问到。

7.3.1 深度优先搜索DFS (Depth First Search)

例 求图G以V0起点的的深度优先序列：

V0, V1, V3, V7, V4, V2, V5, V6,

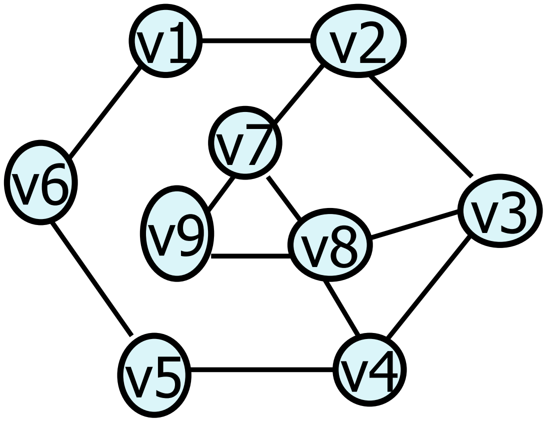
V0, V1, V4, V7, V3, V2, V5, V6

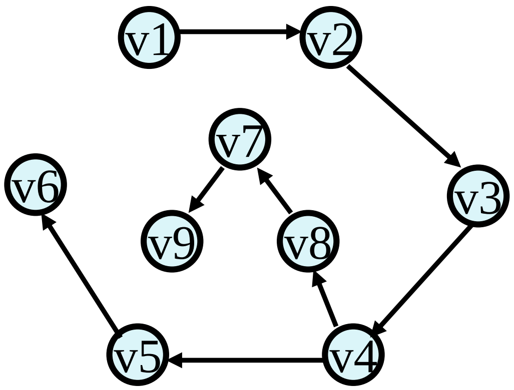


由于没有规定
访问邻接点的顺序，
深度优先序列不是唯一的

7.3.1 深度优先搜索DFS (Depth First Search)

- 深度优先搜索的示例演示



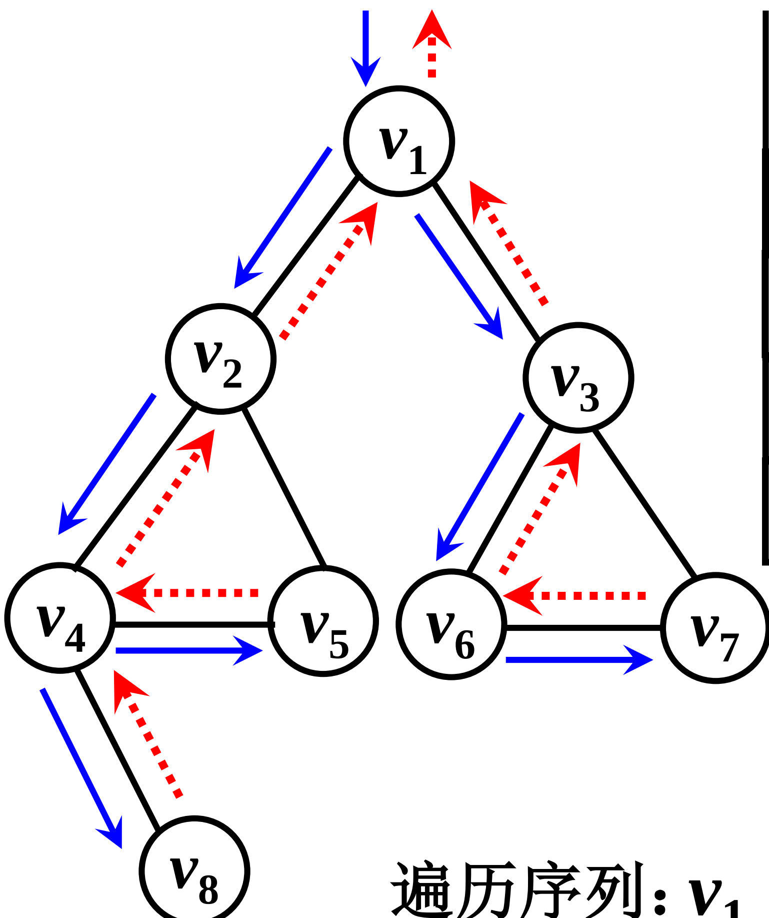


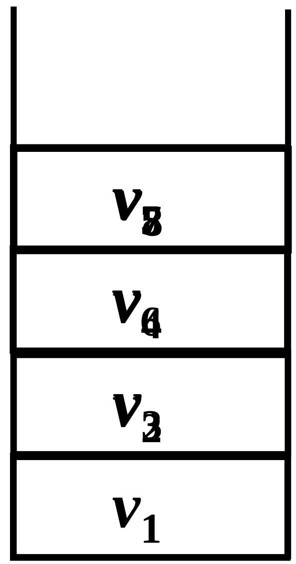
Visited[9]

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

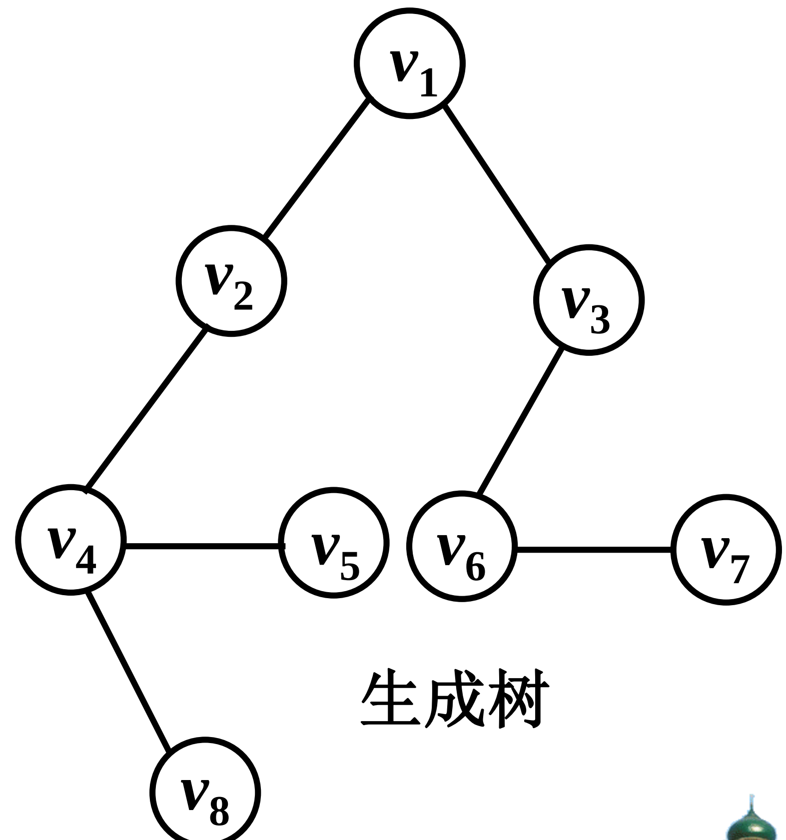
7.3 图的搜索（遍历）

- 深度优先遍历示例
  - 深度优先遍历序列?入栈序列?出栈序列?





栈



生成树

遍历序列：v_1 v_2 v_4 v_5 v_8 v_3 v_6 v_7

7.3 图的搜索（遍历）

- 从一个顶点出发的一次深度优先遍历算法：
  - 实现步骤：

1. 访问顶点v; visited[v]=1;

2. w=顶点v的第一个邻接点；

3. while (w存在)

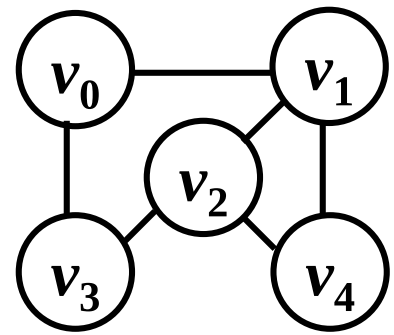
3.1 if (w未被访问) 从顶点w出发递归执行该算法;

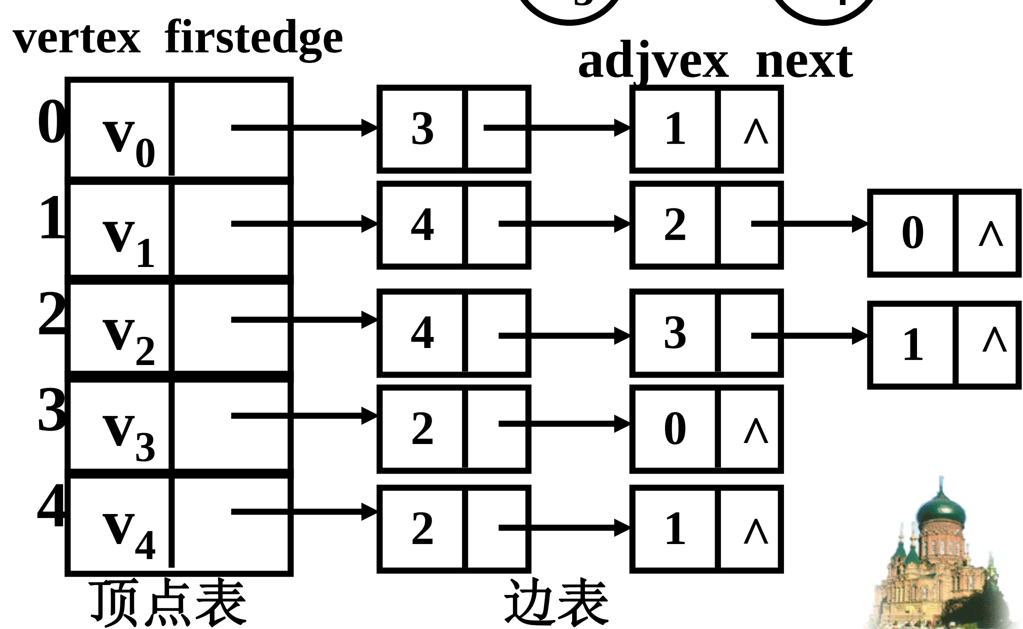
3.2 w=顶点v的下一个邻接点;

7.3 图的搜索（遍历）

- 从一个顶点出发的一次深度优先遍历算法：

```
void DFS1 (AdjGraph* G, int i)
//以vi为出发点时对邻接表表示的图G进行先深搜索
{     EdgeNode *p;
      cout<<G→vexlist[i].vertex;
      visited[i]=True;
     p=G→vexlist[i].firstedge;
      while( p ) {
          if( !visited[ p→adjvex ] )
              DFS1(G, p→adjvex);
          p=p→next;
      }
} //DFS1
```





7.3 图的搜索（遍历）

从一个顶点出发的一次深度优先遍历算法：

void DFS2(MTGraph *G, int i)

//以v_i为出发点对邻接矩阵表示的图G进行深度优先搜索

{ int j;

cout<<G→vexlist[i]; //访问定点v_i

visit[i]=True; //标记v_i已访问

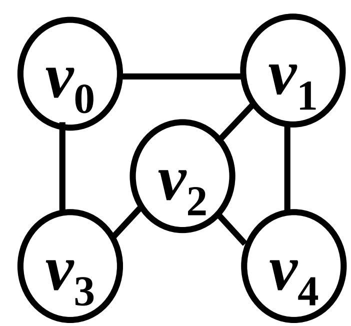
for(j=0; j<G→n; j++) //依次搜索v_i的邻接点

if ((G→edge[i][j] == 1)&&！ visited[j]) //若v_j尚未访问

DFS2(G, j);

}//DFS2

$$
\begin{array}{c|c}
 & \begin{matrix} v_0 & v_1 & v_2 & v_3 & v_4 \end{matrix} \\
\begin{matrix} v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{matrix} & \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix}
\end{array}
$$



7.3.1 深度优先搜索DFS (Depth First Search)

非连通图的深度优先搜索遍历

- 首先将图中每个顶点的访问标志设为 **FALSE**，之后搜索图中每个顶点
- 若已被访问过，则该顶点一定是落在图中已求得的连通分量上；
- 若还未被访问，则从该顶点出发遍历图，可求得图的另一个连通分量。

如果一个无向图是非连通图，如何遍历？

图示

7.3 图的搜索（遍历）

- 深度优先遍历主算法：

```
bool visited[NumVertices];  //访问标记数组是全局变量
void DFSTraverse (AdjGraph G)     //主算法
// 先深搜索一邻接表表示的图G；而以邻接矩阵表示G时，算法完全相同

{   int i;
    for ( int i = 0; i < G.n; i++ )
        visited [i] =False;   //标志数组初始化
    for ( int i = 0; i < G.n; i++ )
        if ( ! visited[i] )
            DFS ( G, i ); //从顶点 i 出发的一次深度优先搜索
}
```

7.3 图的搜索（遍历）

- 深度优先遍历特点：
  - 是递归的定义，是尽可能对纵深方向上进行搜索，故称先深或深度优先搜索。
- 先深或深度优先编号。
  - 搜索过程中，根据访问顺序给顶点进行的编号，称为先深或深度优先编号。
- 先深序列或**DFS**序列：
  - 先深搜索过程中，根据访问顺序得到的顶点序列，称为先深序列或**DFS**序列。
- 生成树（森林）：
  - 有原图的所有顶点和搜索过程中所经过的边构成的子图。
- 先深搜索结果不唯一
  - 即图的**DFS**序列、先深编号和生成森林不唯一。

7.3 图的搜索（遍历）

- 广度优先遍历----类似于树结构的层序遍历

  设图G的初态是所有顶点都“未访问过（False）”，在G中任选一个顶点 v 为源点，则广度优先搜索可定义为:

  ①首先访问出发点 v，并将其标记为“访问过（True）”；

  ②接着依次访问所有与 v 相邻的顶点w_1，w_2…w_t;

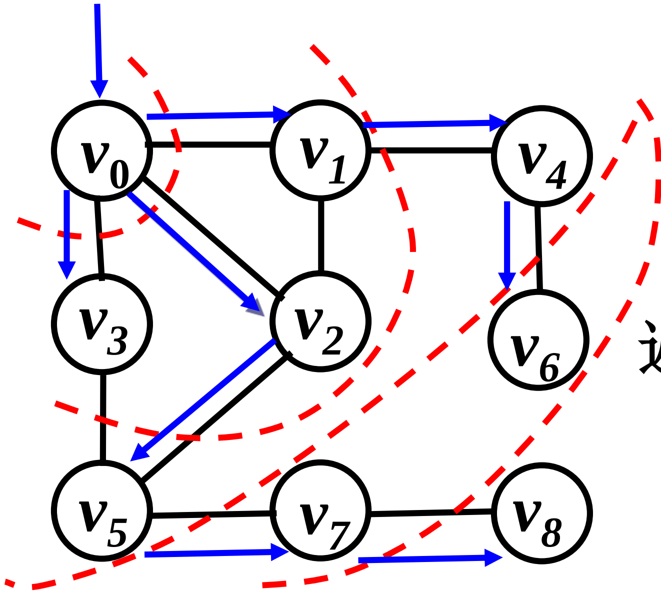
  ③然后依次访问与w_1，w_2… w_t相邻的所有未访问的顶点;

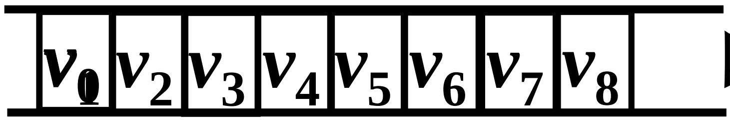
  ④依次类推，直至图中所有与源点v有路相通的顶点都已访问过为止；

  ⑤此时，从 v 开始的搜索结束，若G是连通的，则遍历完成；否则在G中另选一个尚未访问的顶点作为新源点，继续上述搜索过程，直到G中的所有顶点均已访问为止。

7.3 图的搜索（遍历）

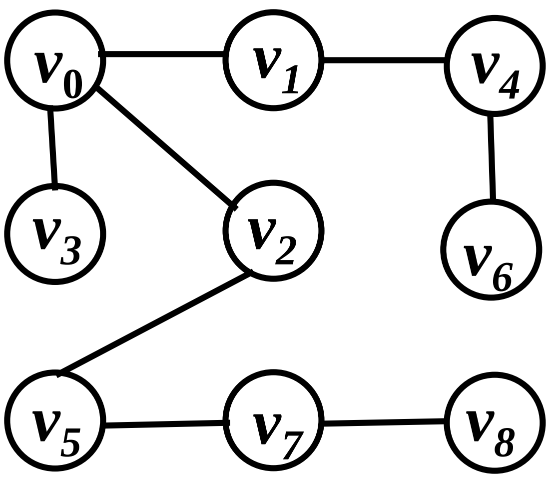
- 广度优先遍历示例
- 广度优先遍历序列?入队序列?出队序列?





遍历序列: v_0 v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8

生成树



7.3 图的搜索（遍历）

- 广度优先遍历特点：
  - 尽可能横向上进行搜索，并使“先被访问的顶点的邻接点”先于“后被访问的顶点的邻接点”被访问，故称先广搜索或广度优先搜索。
- 先广或广度优先编号：
  - 搜索过程中，根据访问顺序给顶点进行的编号，称为先广或广度优先编号
- 先广序列或BFS序列：
  - 先广搜索过程中，根据访问顺序得到的顶点序列，称为先广序列或BFS序列。
- 生成树（森林）：
  - 有原图的所有顶点和搜索过程中所经过的边构成的子图。
- 先广搜索结果不唯一：
  - 即图的BFS序列、先广编号和生成森林不唯一。

7.3 图的搜索（遍历）

- 广度优先遍历主算法：

```
bool visited[NumVertices];  //访问标记数组是全局变量
int dfn[NumVertices];  //顶点的先广编号
void BFSTraverse (AdjGraph G)    //主算法
//* 先广搜索一邻接表表示的图G；而以邻接矩阵表示G时，算法完
    全相同
{   int i , count = 1;
    for ( int i = 0; i < G.n; i++ )
        visited [i] =False;   //标志数组初始化
    for ( int i = 0; i < G.n; i++ )
        if ( ! visited[i] )
            BFS ( G, i ); //从顶点 i 出发的一次广度优先搜索
}
```

7.3 图的搜索（遍历）

- 从一个顶点出发的一次广度优先遍历算法：
  - 实现步骤：

1 . 初始化队列Q;

2. 访问顶点v; visited [v]=1; 顶点v入队Q;

3. while (队列Q非空)

  3.1 v=队列Q的队头元素出队;

  3.2 w=顶点v的第一个邻接点;

  3.3 while (w存在)

    3.3.1 如果w 未被访问，则

      访问顶点w; visited[w]=1; 顶点w入队列Q;

    3.3.2 w=顶点v的下一个邻接点;

7.3 图的搜索（遍历）

```
void BFS1 (AdjGraph *G, int k)//以vk为出发点时对用邻接表表示的图G进行先广搜索
{    int i; EdgeNode *p; Queue Q;   MakeNull(Q);
     cout << G→vexlist[ k ] .vertex;   visited[ k ] = True;
     EnQueue (k, Q);                          //进队列
     while ( ! Empty (Q) ) {                  //队空搜索结束
          i=DeQueue(Q);                       //vi出队
          p =G→vexlist[ i ].firstedge;        //取vi的边表头指针
          dfn[i] = count++;                   //先广编号
          while ( p ) {                       //若vi的邻接点 vj (j= p→adjvex)存在,依次搜索
               if ( !visited[ p→adjvex ]) {   //若vj未访问过
                    cout << G→vexlist[ p→adjvex ].vertex;   //访问vj
                    visited[ p→adjvex ]=True;               //给vj作访问过标记
                    EnQueue ( p→adjvex , Q );               //访问过的vj入队
               }
               p = p→next;                    //找vi的下一个邻接点
          } /          / 重复检测 vi的所有邻接顶点
     }                       //外层循环，判队列空否
}
```

7.3 图的搜索（遍历）

```
void BFS2 (MTGraph *G, int k) //以vk为出发点时对用邻接矩阵表示的图G进行先广搜索
{    int i , j; Queue Q;   MakeNull(Q);
     cout << G→vexlist[ k ];   //访问vk
     visited[ k ] = True; //给vk作访问过标记
     EnQueue (k, Q);   // vk进队列
     while ( ! Empty (Q) ) {  //队空时搜索结束
          i=DeQueue(Q); //vi出队
          for(j=0; j<G→n; j++) {   //依次搜索vi的邻接点 vj
               if ( G→edge[ i ][ j ] ==1 && !visited[ j ]) { //若vj未访问过
                    cout << G→vexlist[ j ];//访问vj
                    visited[ j ]=True;  //给vj作访问过标记
                    EnQueue ( j , Q ); //访问过的vj入队
               }
          } //重复检测 vi的所有邻接顶点
     } //外层循环，判队列空否
} //这里没有进行先广编号
```

无向图（的搜索）及其应用

- 无向图连通性判定
  - 不连通：生成森林
    求连通分量个数；
    求出每个连通分量；
  - 连通：一棵生成树
    判断是否有环路；
    求带权连通图的最小生成树；