数据结构

北京邮电大学信息安全中心
武斌 杨榆

上次课内容

上次课（树和二叉树（下））内容：

- 熟练掌握二叉树的各种遍历算法，并能灵活运用遍历算法实现二叉树的其它操作
- 理解二叉树的线索化过程以及在中序线索化树上找给定结点的前驱和后继的方法
- 学习树和森林的关系及转换方法
- 了解最优树的特性，掌握建立最优树和赫夫曼编码的方法

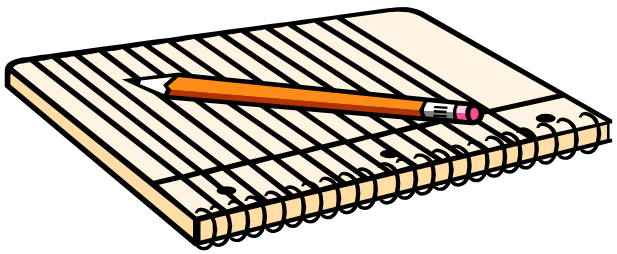
本次课程学习目标

学习完本次课程（图(上)），您应该能够：

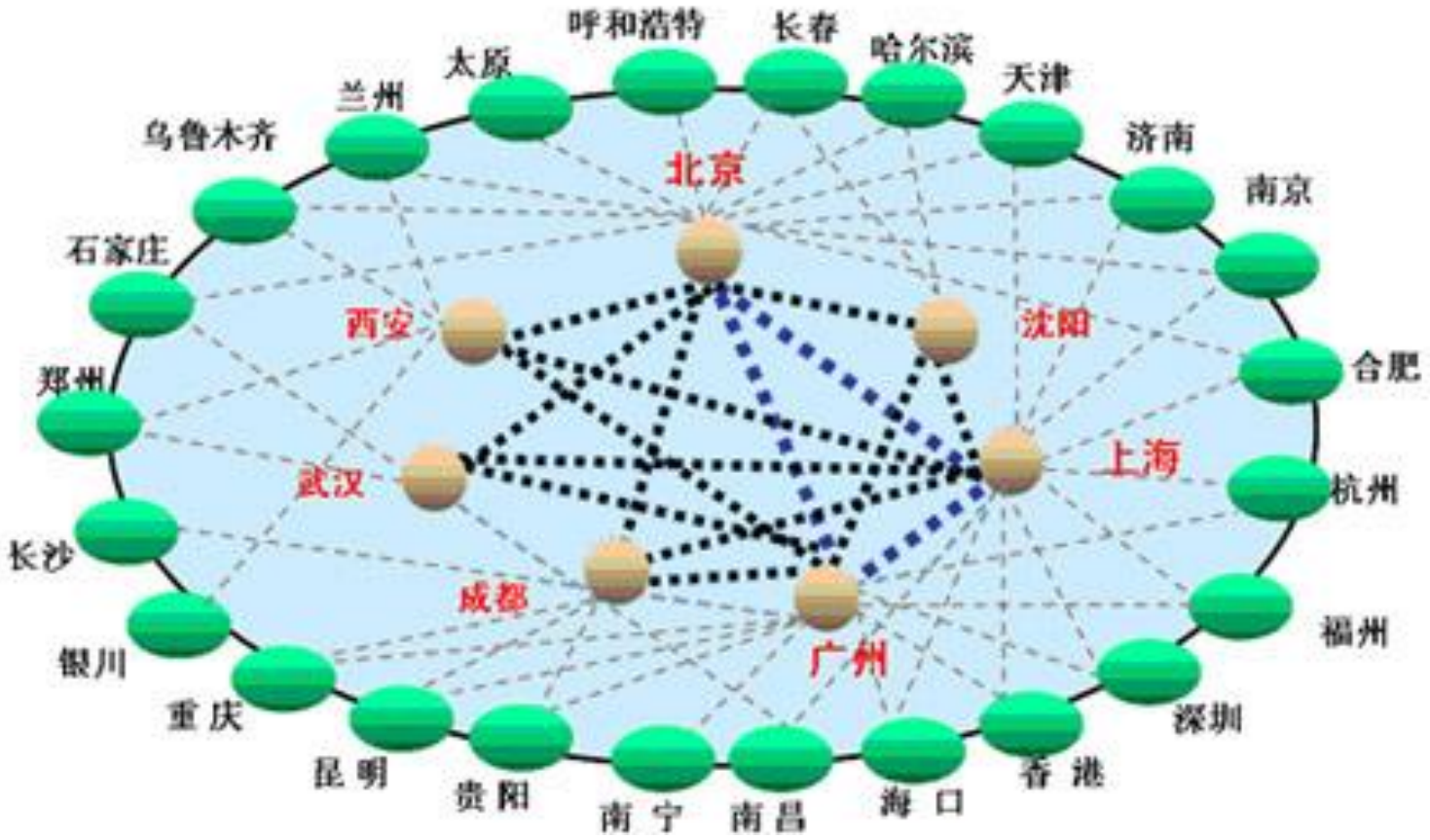
- 领会图的类型定义及术语。
- 熟悉图的各种存储结构及其构造算法，了解各种存储结构的特点及其选用原则。
- 熟练掌握图的两种遍历算法。

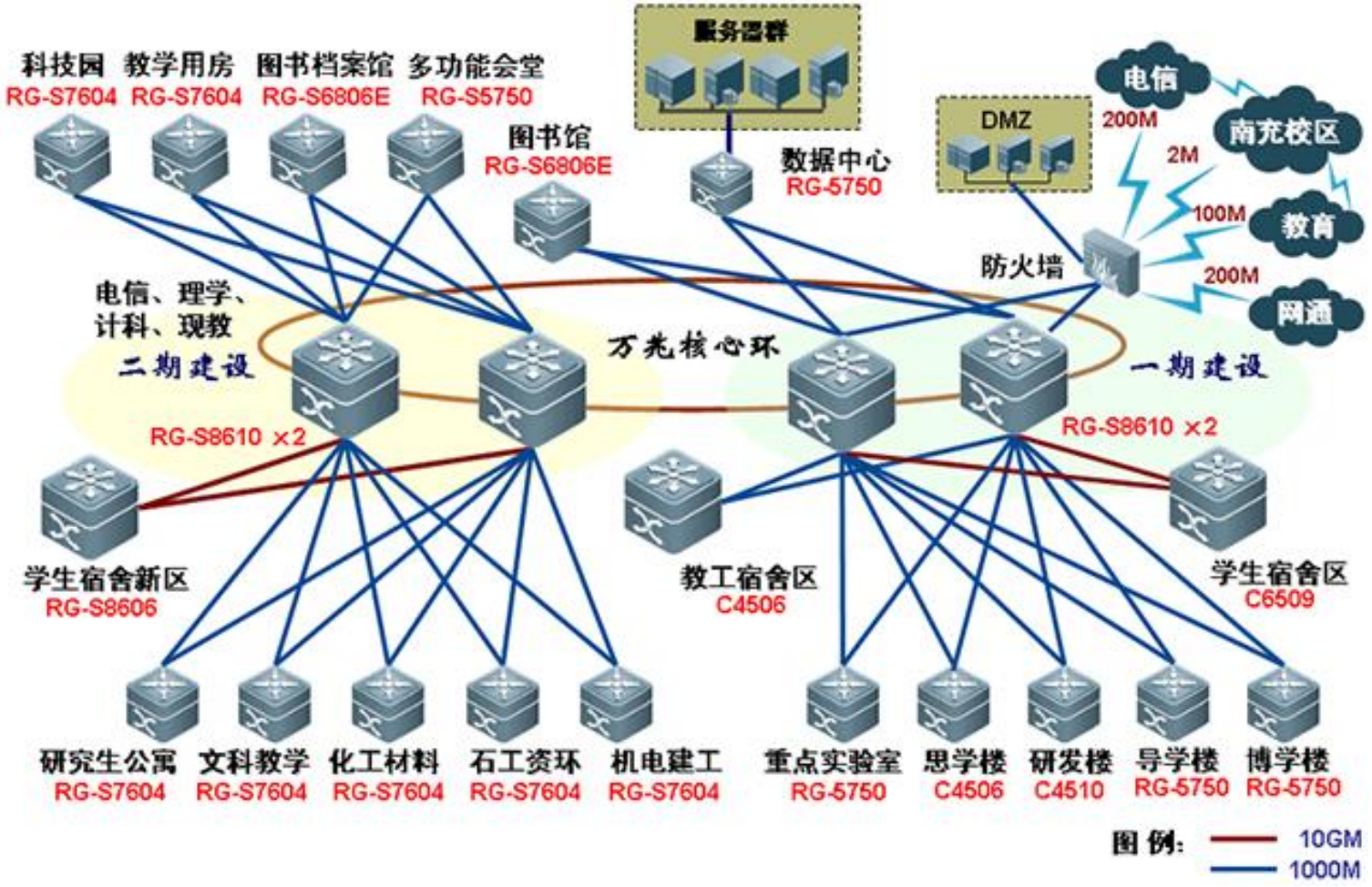
本章课程内容（第七章 图）

- 7.1 图的定义和术语
- 7.2 图的存储结构
- 7.3 图的遍历
- 7.4 图的连通性问题
- 7.5 有向无环图及其应用
- 7.6 最短路径



ChinaNet骨干网网络拓扑

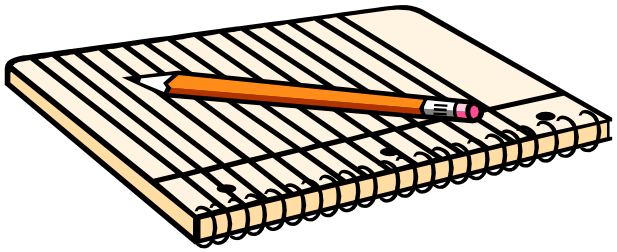



服务器群
科技园 教学用房 图书档案馆 多功能会堂
RG-S7604 RG-S7604 RG-S6806E RG-S5750
图书馆
RG-S6806E
数据中心
RG-5750
DMZ
电信
200M
南充校区
2M
100M
教育
防火墙
200M
网通
电信、理学、
计科、现教
二期建设
万兆核心环
一期建设
RG-S8610 ×2
RG-S8610 ×2
学生宿舍新区
RG-S8606
教工宿舍区
C4506
学生宿舍区
C6509
研究生公寓 文科教学 化工材料 石工资环 机电建工
RG-S7604 RG-S7604 RG-S7604 RG-S7604 RG-S7604
重点实验室 思学楼 研发楼 导学楼 博学楼
RG-5750 C4506 C4510 RG-5750 RG-5750
图例:
10GM
1000M

第七章 图

- 图（Graph）是一种较线性表和树更为复杂的数据结构。
  - 在线性表中，数据元素之间仅有线性关系，每个数据元素只有一个直接前驱和一个直接后继；
  - 在树形结构中，数据元素之间有着明显的层次关系，并且每一层上的数据元素可能和下一层中多个元素（即其孩子结点）相关，但只能和上一层中一个元素（即双亲结点）相关。
  - 而在图形结构中，结点之间的关系可以是任意的，图中任意两个数据元素之间都可能相关。
- 在此主要学习图的存储结构以及若干图的操作的实现。

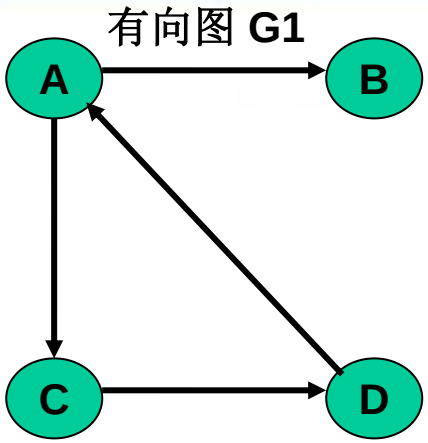
图的定义和术语



- 7.1 图的定义和术语
- 7.2 图的存储结构
- 7.3 图的遍历
- 7.4 图的连通性问题
- 7.5 有向无环图及其应用
- 7.6 最短路径

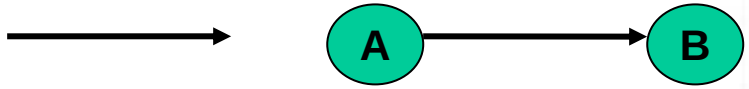
图的定义和术语

- **定义**：图由一个顶点集和弧集构成，通常写作：Graph=(V,VR)。由于空的图在实际应用中没有意义，因此一般不讨论空的图，即V是顶点的有穷非空集合，而VR是两个顶点之间的关系的集合。
  - 若< v, w >∈VR，则< v, w >表示从v到w的一条**弧(Arc)**，且称v为**弧尾**(Tail)或初始点(Initial Node)，称w为**弧头**(Head)或终端点(Terminal Node)，此时的图称为**有向图**(Digraph)。



- 结点或 顶点：

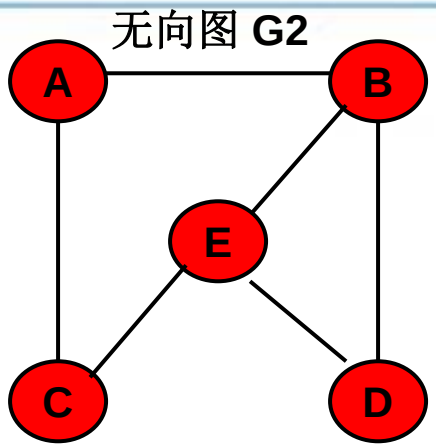
- 有向边（弧）、弧尾或初始结点、弧头或终止结点



- 有向图：G1 =（V1，{A1}）
  V1 = {A,B,C,D}
  A1 = {<A,B>, <A,C>,
  <C,D>, <D,A>}

图的定义和术语

- **定义**：图由一个顶点集和弧集构成，通常写作：Graph=(V,VR)。由于空的图在实际应用中没有意义，因此一般不讨论空的图，即V是顶点的有穷非空集合，而VR是两个顶点之间的关系的集合。
  - 若< v, w >∈VR，必有若< w, v >∈VR，即VR是对称的，则以无序对(v, w)代替这两个有序对，表示v和w之间的一条边(Edge)，此时的图称为无向图(Undigraph)。



- 结点或 顶点：

- 无向边或边

- 无向图：G2 =（V2，{A2}）
  V2 = {A,B,C,D,E}
  A2 = {(A,B), (A,C),(B,D),
  (B,E), (C,E),(D,E)}

图的定义和术语

●例如：下列定义的有向图如左下图所示。

G1=(V1, VR1)

其中：V1 = {A, B, C, D, E}

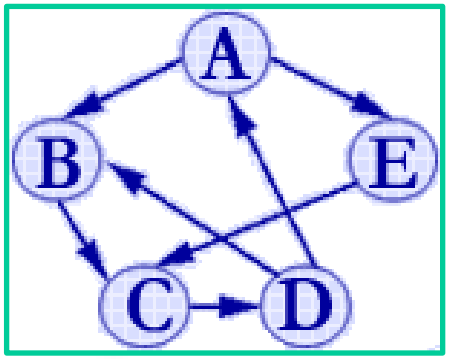
VR1 ={<A,B>,<A,E>,<B,C>,<C,D>,<D,B>,<D,A>,<E,C>}

●例如：下列定义的无向图如右下图所示。

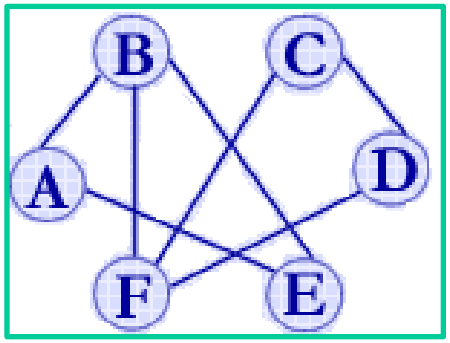
G2=(V2, VR2)

其中：V2={A, B, C, D, E, F}

VR2={(A,B),(A,E),(B,E) ,(B,F),(C,D) ,(C,F),(D,F) }



有向图G1



无向图G2

图的定义和术语

- 图的抽象数据类型定义如下：

ADT Graph {

数据对象V： V是具有相同特性的数据元素的集合，称为顶点集。

数据关系R： R = { VR }

VR＝{ <v,w>| v, w ∈V且P(v,w)，<v,w>表示从v到w的弧，
谓词P(v,w)定义了弧<v,w>的意义或信息 } 。

基本操作P：

{结构初始化}

CreateGraph(&G, V, VR);

初始条件：V 是图的顶点集，VR 是图中弧的集合。

操作结果：按 V 和 VR 的定义构造图 G。

图的定义和术语

{销毁结构}

DestroyGraph(&G);

初始条件：图 G 存在。

操作结果：销毁图 G。

{引用型操作}

LocateVex(G, u);

初始条件：图 G 存在，u 和 G 中顶点有相同特征。

操作结果：若 G 中存在和 u 相同的顶点，则返回该顶点在图中位置；否则返回其它信息。

GetVex(G, v);

初始条件：图 G 存在，v 是 G 中某个顶点。

操作结果：返回 v 的值。

FirstAdjVex(G, v);

初始条件：图 G 存在，v 是 G 中某个顶点。

操作结果：返回 v 的第一个邻接点。若该顶点在 G 中没有邻接点，则返回“空”。

图的定义和术语

{引用型操作}

NextAdjVex(G, v, w);

初始条件：图 G 存在，v 是 G 中某个顶点，w 是 v 的邻接顶点。操作结果：返回 v 的（相对于 w 的）下一个邻接点。若 w 是 v 的最后一个邻接点，则返回“空”。

{加工型操作}

PutVex(&G, v, value);

初始条件：图 G 存在，v 是 G 中某个顶点。

操作结果：对 v 赋值 value。

InsertVex(&G, v);

初始条件：图 G 存在，v 和图中顶点有相同特征。

操作结果：在图 G 中增添新顶点 v。

DeleteVex(&G, v);

初始条件：图 G 存在，v 是 G 中某个顶点。

操作结果：删除 G 中顶点 v 及其相关的弧。

图的定义和术语

{加工型操作}

InsertArc(&G, v, w);

初始条件：图 G 存在，v 和 w 是 G 中两个顶点。

操作结果：在G中增添弧<v,w>，若G是无向的，则还增添对称弧<w,v>。

DeleteArc(&G, v, w);

初始条件：图 G 存在，v 和 w 是 G 中两个顶点。

操作结果：在G中删除弧<v,w>，若G是无向的，则还删除对称弧<w,v>。

DFSTraverse(G, Visit());

初始条件：图 G 存在，Visit 是顶点的应用函数。

操作结果：对图G进行深度优先遍历。遍历过程中对每个顶点调用函数 Visit 一次且仅一次。一旦 visit() 失败，则操作失败。

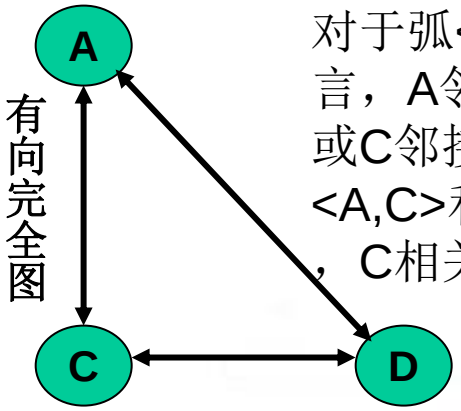
BFSTraverse(G, Visit());

初始条件：图 G 存在，Visit 是顶点的应用函数。

操作结果：对图G进行广度优先遍历。遍历过程中对每个顶点调用函数 Visit 一次且仅一次。一旦 visit() 失败，则操作失败。

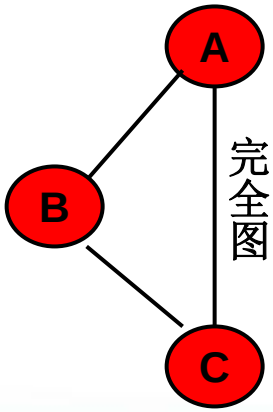
图的定义和术语

- **完全图**：有 n(n-1)/2 条边的无向图。其中 n 是结点个数。
- **有向完全图**：有 n(n-1) 条边的有向图。其中 n 是结点个数。
- 有很少的边或弧（e<nlogn）的图称为**稀疏图**，反之成为**稠密图**。
- **邻接点**：无向图G=(V,{E})，如果边(V,V')∈E，则称V和V'互为邻接点。边(V,V')和顶点V，V'相关联。



有向完全图

对于弧<A,C>而言，A邻接到C，或C邻接自A，弧<A,C>和顶点A，C相关联。

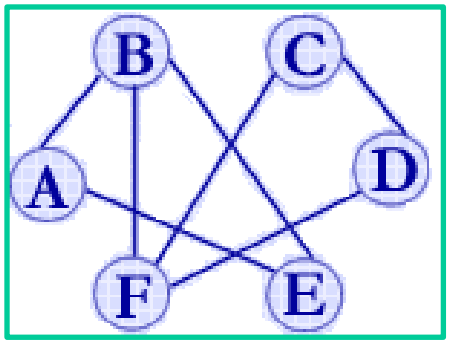


完全图

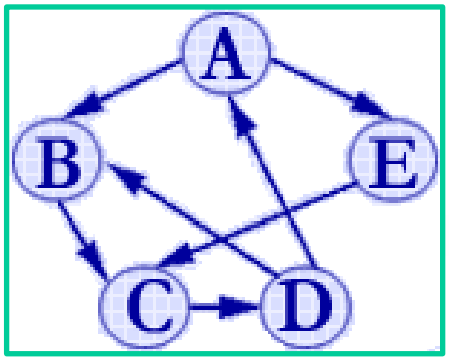
A和B互为邻接点，边(A,B)和顶点A，B相关联

顶点的度

- 顶点的度
  - 对无向图而言，与一个顶点相关联的边的数目，或者说与其相邻接的顶点数为该顶点的度（TD）。例如左下方无向图中顶点B的度为3，顶点C的度为2。
  - 对有向图而言，顶点的度为其出度和入度之和，其中出度（OD）定义为以该顶点为弧尾的弧的数目，入度（ID）定义为以该顶点为弧头的弧的数目。例如右下方有向图中顶点D的度为3，其中出度为2，入度为1，顶点B的度为3，其中出度为1，入度为2。



无向图G2

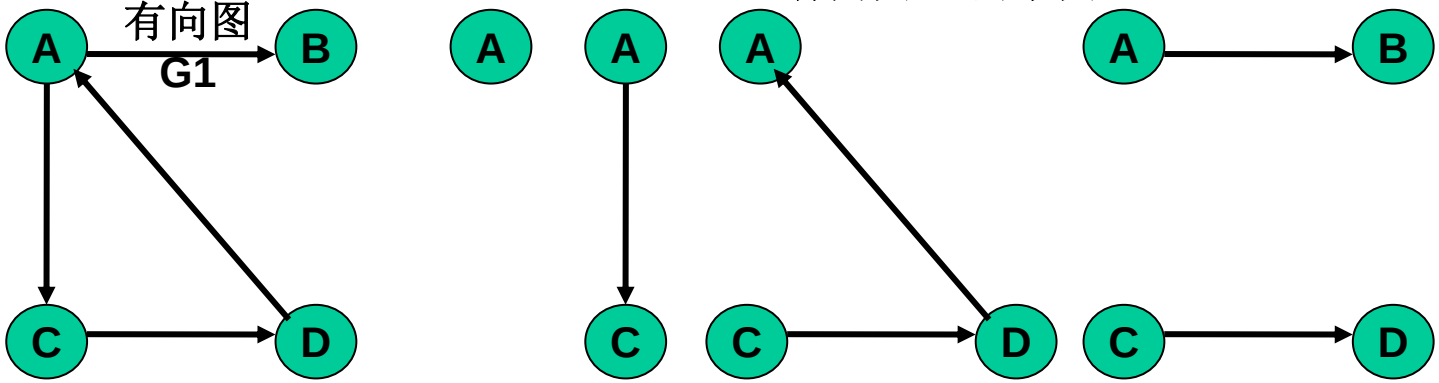


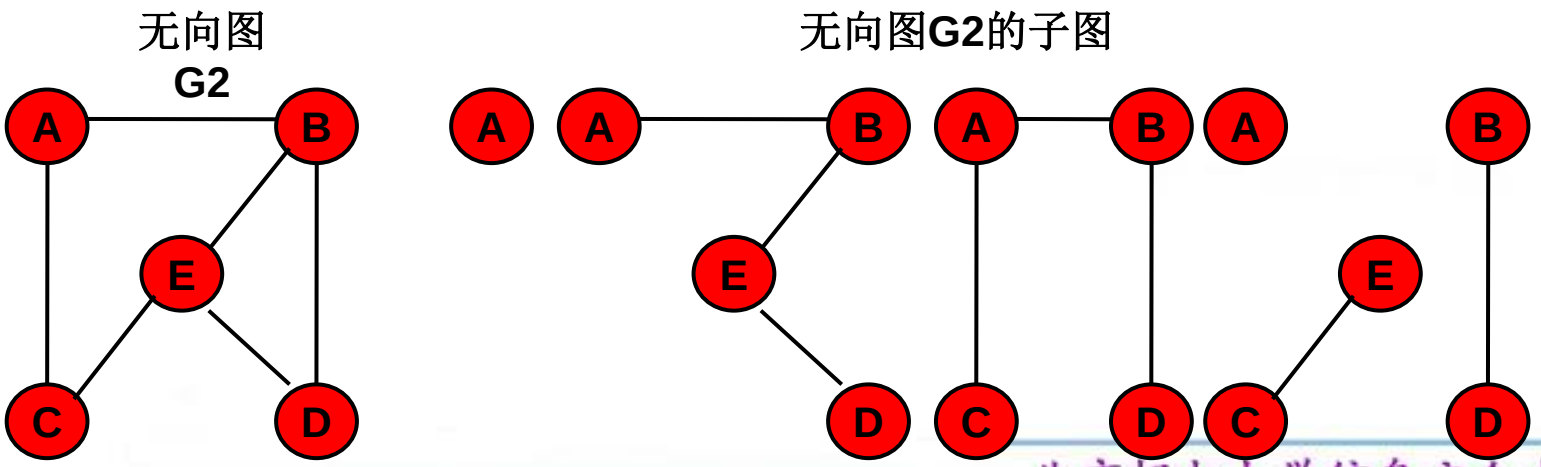
有向图G1

子图

- 子图

假设有两个图 G=(V,{E}) 和 G'=(V',{E'})，如果V'⊆V 且 E'⊆ E，则称G'为G的**子图(subgraph)**。

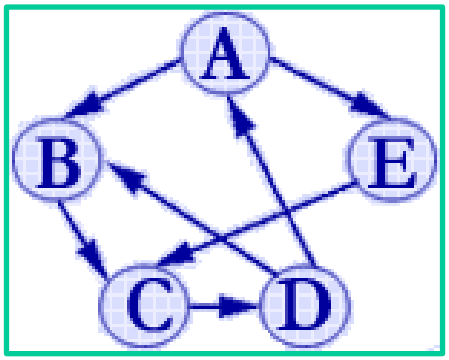




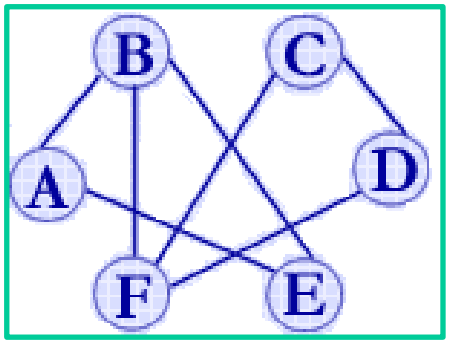
路径和回路

●路径和回路

- 若有向图G中k+1个顶点之间都有弧存在（即<v_0,v_1>,<v_1，v_2>，…<v_{k-1},v_k>是图 G 中的弧），则这个顶点的序列（ v_0, v_1,…, v_k ）为从顶点v_0到顶点v_k的一条有向路径。
- 例如：有向图中顶点序列（ A,B,C,D ）是一条从顶点A到顶点D的有向路径。
- 对无向图，顶点v_0到顶点v_k的路径是一个顶点序列（ v_0, v_1,…, v_k ），其中（ v_i, v_{i+1} ）i=0..k-1都是无向图中的边。称路径的边数为路径长度为 k。
- 例如：无向图中顶点序列 （ A,B,F,C,D ） 是一条长度为4的无向路径。



有向图G1

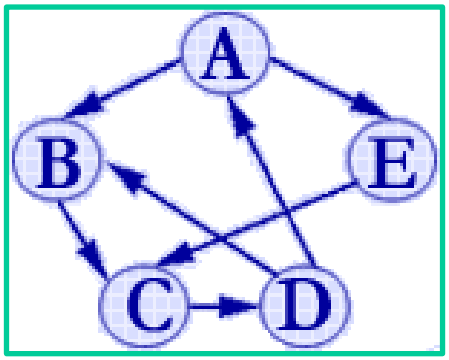


无向图G2

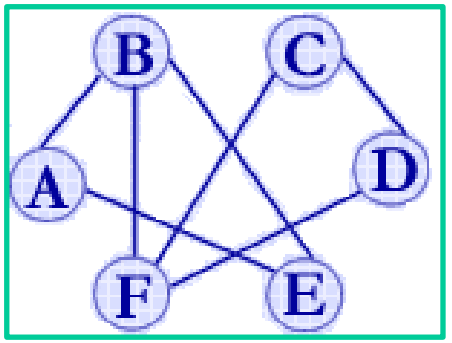
路径和回路

●路径和回路

- 路径中弧的数目定义为**路径长度**，若序列中的顶点都不相同，则为**简单路径**。
- 例如，有向图中顶点序列 （ A,E,C,D ）是一条路径长度为3的简单路径。无向图中顶点序列 （ A,B,F,C,D ） 是一条路径长度为4的简单路径。
- 若顶点序列 （ $v_0, v_1, \ldots, v_k$ ）第一个顶点和最后一个顶点相同，即顶点序列为一条由某个顶点出发又回到自身的路径，称这种路径为**回路**或**环**。
- 若除v_0和v_k外的顶点都不重复，称这种路径为**简单回路**。
- 例如，有向图中顶点序列（ A,B,C,D,A,E,C,D,A ）是回路 ，有向图中顶点序列 （ A,B,C,D,A ） 是一条长度为4的简单回路。



有向图G1



无向图G2

连通图和连通分量

- **连通图和连通分量、强连通图和强连通分量**

若无向图中任意两个顶点之间都存在一条无向路径，则称该无向图为连通图，否则称为非连通图。若有向图中任意两个顶点之间都存在一条有向路径，则称该有向图为强连通图。例如图1所示无向图和图2所示有向图分别为连通图和强连通图。

非连通图中各个极大连通子图称作该图的连通分量。如图3为由两个连通分量构成的非连通图。非强连通的有向图中的极大强连通子图称作有向图的强连通分量。例如图4中的有向图含有三个强连通分量。

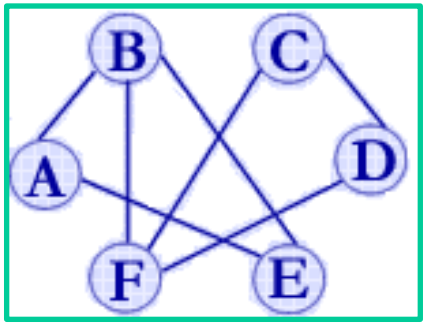


图1 无向图G2
连通图

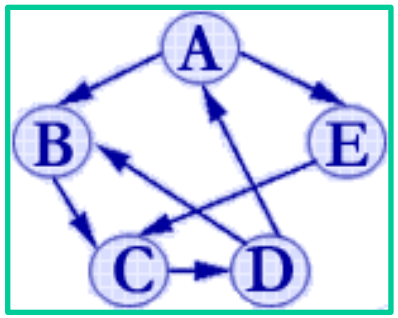


图2 有向图G1
强连通图

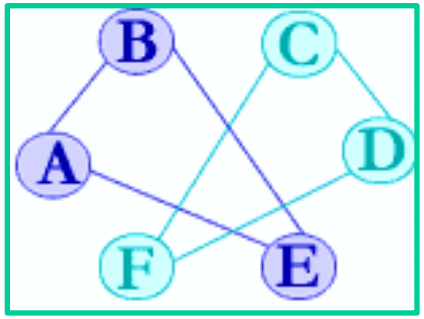


图3 非连通图（无向）
两个连通分量

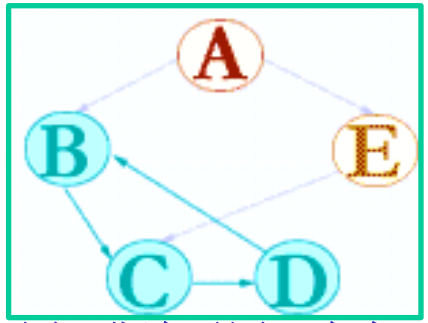
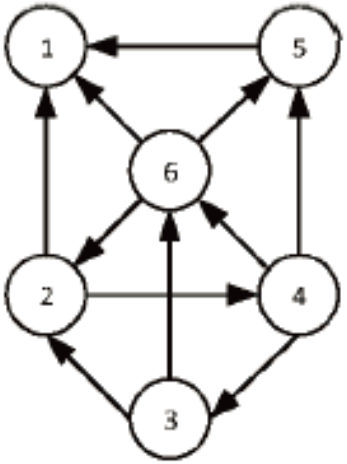


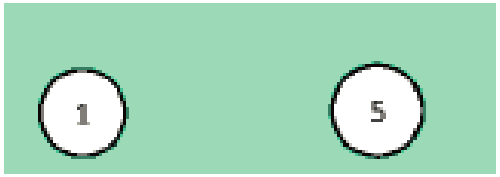
图4 非连通图（有向）
三个强连通分量

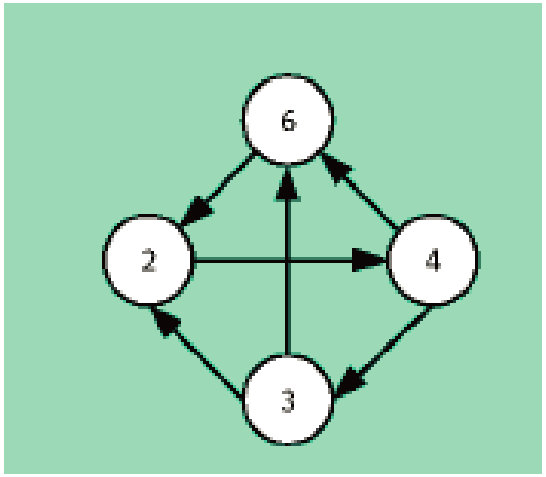
连通图和连通分量

- 连通图和连通分量、强连通图和强连通分量

练习：请给出下图的强连通分量。







生成树和生成森林

- 生成树和生成森林

一个含 n 个顶点的**连通图**的**生成树**是该图中的一个**极小连通子图**，它包含图中 n 个顶点和足以构成一棵树的 n-1 条边。如图2所示。

对于非连通图，对其每个连通分量可以构造一棵生成树，合成起来就是一个生成森林。如图6所示。

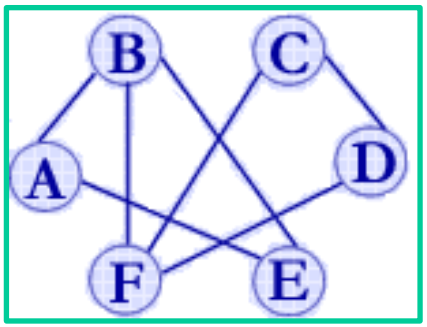


图1 无向图G1(连通图)

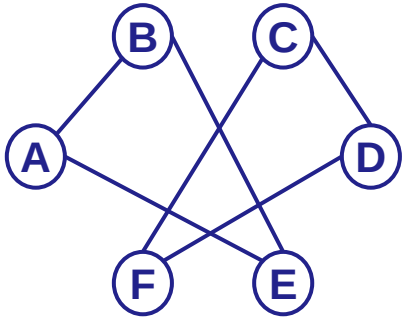


图3 无向图G2(非连通图)

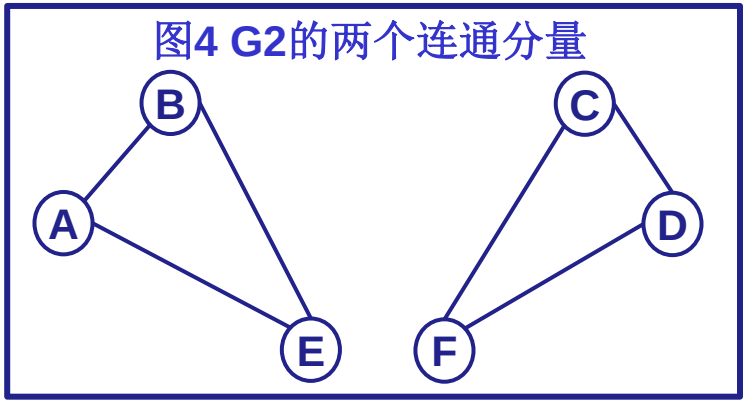


图4 G2的两个连通分量

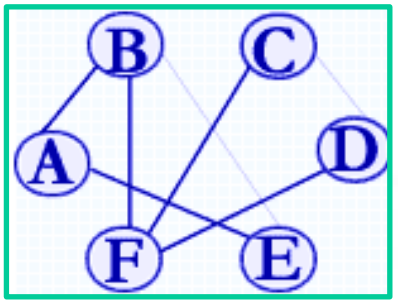


图2 G1的生成树

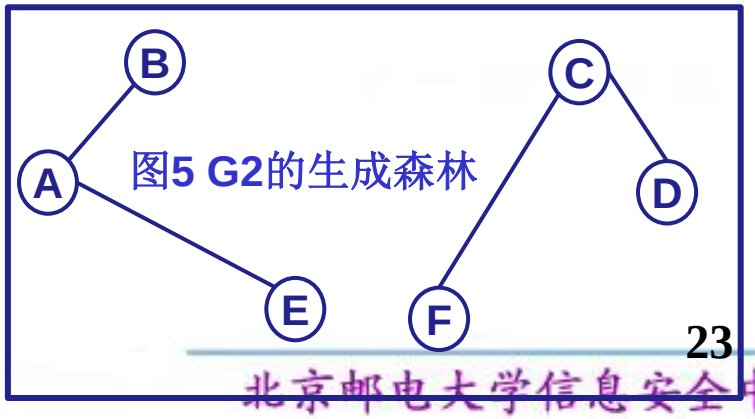


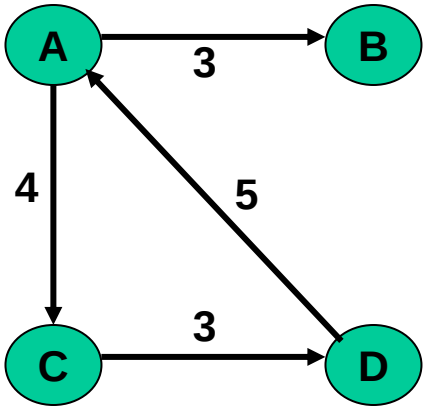
图5 G2的生成森林

生成树和生成森林

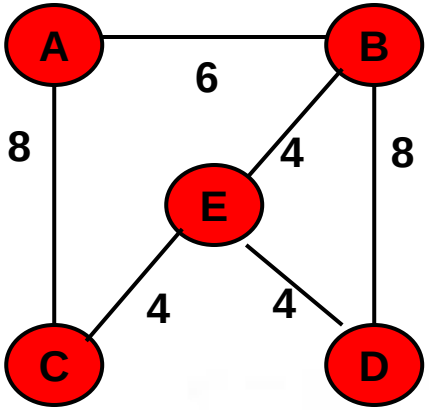
●无向网和有向网

在实际应用中，图的弧或边往往与具有一定意义的数相关，称这些数为“权”，带权的图通常成为“网”，分别称带权的有向图和无向图为有向网和无向网。

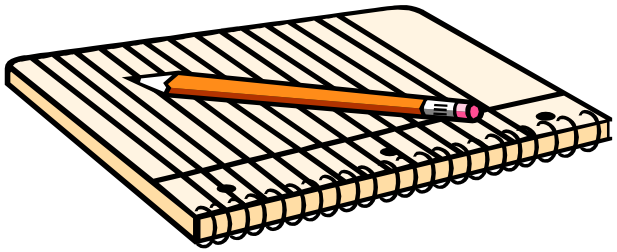
有向网



无向网2



图的存储结构



- 7.1 图的定义和术语
- 7.2 图的存储结构
- 7.3 图的遍历
- 7.4 图的连通性问题
- 7.5 有向无环图及其应用
- 7.6 最短路径

图的存储结构

- 由于图结构中任意两个顶点之间都可能存在“关系”，因此，无法以顺序存储映象表示这种关系，即**图没有顺序存储结构**。
- 图的四种常用的存储形式：
  - 邻接矩阵和加权邻接矩阵
  - 邻接表
  - 十字链表
  - 邻接多重表

邻接矩阵

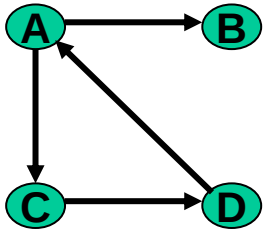
优点：判断任意两点之间是否有边方便，仅耗费 O(1) 时间。
缺点：即使 << n^2 条边，也需内存 n^2 单元; 仅读入数据耗费 O(n^2) 时间。

- 假设图中顶点数为n，则**邻接矩阵** $A=(a_{i,j})_{n\times n}$定义为

$$A[i][j]=\begin{cases}1 & \text{若}V_i\text{和}V_j\text{之间有弧或边存在}\\ 0 & \text{反之}\end{cases}$$

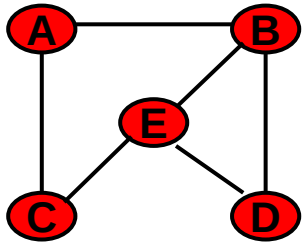
- **网的邻接矩阵**的定义为，当v_i到v_j有弧相邻接时，$a_{i,j}$的值应为该弧上的权值，否则为∞。

- 将图的顶点信息存储在一个一维数组中，并将它的邻接矩阵存储在一个二维数组中即构成图的数组表示。



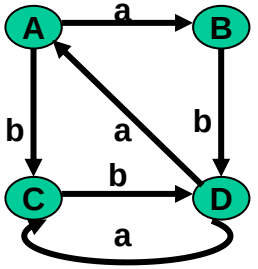
A[i,i] = 0
顶点i的出度为第 i行之和，其入度为第i列之和。

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



A[i,i] = 0
顶点i 的度为第 i 行或i 列之和。

0	1	1	0	0
1	0	0	1	1
1	0	0	0	1
0	1	0	0	1
0	1	1	1	0



∞	a	b	∞
∞	∞	∞	b
∞	∞	∞	b
a	∞	a	∞

图的存储结构

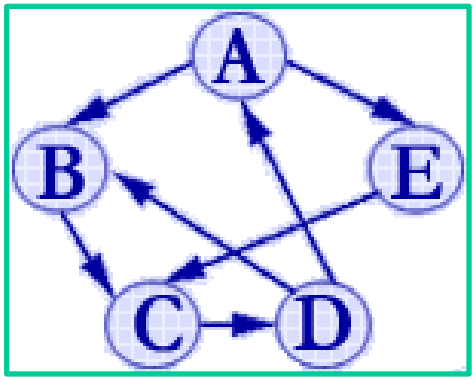
●一、图的数组(邻接矩阵)存储表示

图的“邻接矩阵”是以矩阵这种数学形式描述图中顶点之间的关系。

```
#define  INFINITY      INT_MAX            // 最大值∞
#define  MAX_VERTEX_NUM   20             // 最大顶点个数
typedef enum {DG, DN, UDG, UDN}  GraphKind;  // 类型标志{有向图,有向网,无向图,无向网}
typedef struct ArcCell {                // 弧的定义
      VRType adj;                       // VRType是顶点关系类型。
     // 对无权图，用1或0表示相邻否；对带权图，则为权值类型。
      InfoType      *info;              // 该弧相关信息的指针
} ArcCell, AdjMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];
typedef struct {                        // 图的定义
      VertexType   vexs[MAX_VERTEX_NUM]; // 顶点信息
      AdjMatrix      arcs;              // 表示顶点之间关系的二维数组
      int    vexnum, arcnum;            // 图的当前顶点数和弧(边)数
      GraphKind      kind;              // 图的种类标志
} MGraph;
```

图的存储结构

- 例如，有向图G1的数组表示存储结构为：
  - G1.vexs=[A,B,C,D,E]
  - $$G1.arcs = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$ （忽略相关信息指针）

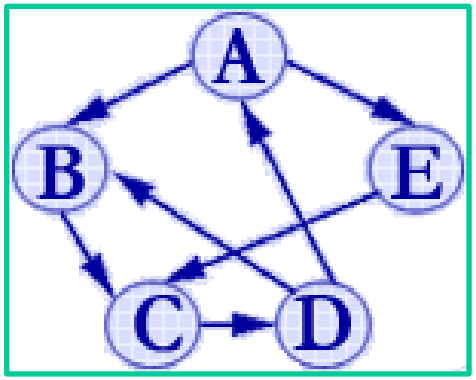


有向图G1

  - G1.vexnum=5
  - G1.arcsnum=7
  - G1.kind=DG
  - 有向图的邻接矩阵不一定对称。每一行中“1”的个数为该顶点的出度，反之，每一列中“1”的个数为该顶点的入度。

图的存储结构

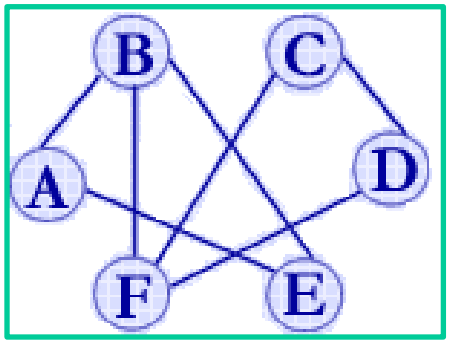
- 顶点的“第一个”邻接点就应该是该顶点所对应的行中**值为非零元素的最小列号**，其“下一个”邻接点就是同行中**离它最近的值为非零元素的列号**。
- **例如**，前页有向图中顶点A的第一个邻接点为"顶点B"(因为顶点A在顶点数组 G1.vexs 中的下标为0，又G1.arcs[0] 中非零元素的最小列下标为1，而G1.vexs[1]=B)，同样理由，顶点A相对于邻接点B的下一个邻接点是"顶点E"。



$$G1.arcs = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

图的存储结构

- 例如，无向图G2的数组表示存储结构为：
  - G2.vexs=[A,B,C,D,E,F]
  - $G2.arcs = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \end{bmatrix}$ (忽略相关信息指针)
  - G2.vexnum=6
  - G2.arcnum=7
  - G2.kind=UDG
  - 容易看出，无向图的邻接矩阵为对称矩阵。每一行中“1”的个数恰为该顶点的“度”。



无向图G2

图的存储结构

● 算法 7.1

```
Status CreateGraph ( MGraph &G)
{
    //采用数组（邻接矩阵）表示法，构造图G
    scanf(&G.kind);
    switch(G.kind){
       case DG: return CreateDG(G);  //构造有向图G
       case DN: return CreateDN(G);   //构造有向网G
       case UDG: return CreateUDG(G);  //构造无向图G
       case UDN: return CreateUDN(G);  //构造无向网G
       default : return ERROR;
     }
}// CreateGraph
```

算法7.1是在邻接矩阵存储结构MGrpah上对图的构造操作的实现框架，它根据图G的种类调用具体构造算法。如果G是无向网，则调用后页算法7.2。

图的存储结构

● 算法7.2

```
Status CreateUDN(MGraph &G)
{   //采用数组（邻接矩阵）表示法，构造无向图G
    scanf(&G.vexnum,&G.arcnum,&IncInfo); // IncInfo为0则各弧不含其他信息
    for(i=0;i<G.vexnum;++i)  scanf(&G.vexs[i]); // 构造顶点向量
    for(i=0;i<G.vexnum;++i)          // 初始化邻接矩阵
        for(j=0;j<G.vexnum;++j) G.arcs[i][j]={INFINITY, NULL};  // {adj, info}
    for(k=0;k<G.arcnum;++k){       // 构造邻接矩阵
        scanf(&v1,&v2,&w);              // 输入一条边依附的顶点及权值
        i =  LocateVex(G,v1);  j = LocateVex(G,v2);   // 确定v1和v2在G中的位置
        G.arcs[i][j].adi = w;           // 弧<v1,v2>的权值
        if(IncInfo)  Input(*G.arcs[i][j].info);   // 若弧含有相关信息，则输入
        G.arcs[j][i] = G.arcs[i][j];          // 置<v1,v2>的对称弧<v2,v1>
    }
    return OK;
}// CreateUDN
```

构造一个具有**n个顶点**和**e条边**的**无向网G**的时间复杂度是 $O(n^2+e\times n)$，其中对邻接矩阵G.arcs的初始化耗费了$O(n^2)$的时间。

图的存储结构

●二、图的邻接表存储表示

类似于树的孩子链表，将和**同一顶点“相邻接”的所有邻接点**链接在一个单链表中，**单链表的头指针则和顶点信息**一起存储在一个一维数组中。

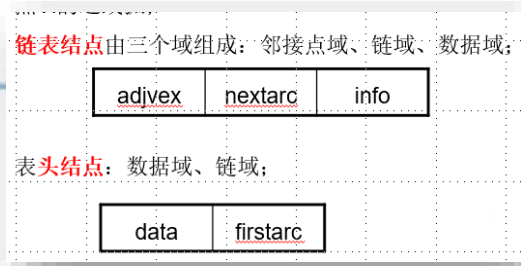
- 对图中每个顶点建立一个单链表，第i个单链表中的结点表示依附于顶点Vi的边或弧；
- **链表结点**由三个域组成：邻接点域、链域、数据域；

adjvex	nextarc	info

- 表**头结点**：数据域、链域；

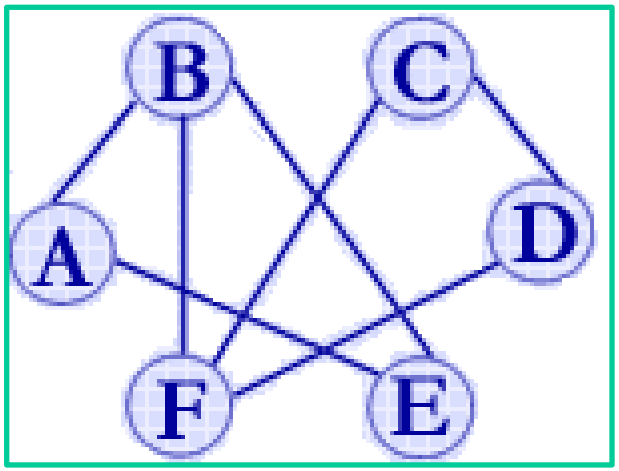
data	firstarc

图的邻接表存储表示

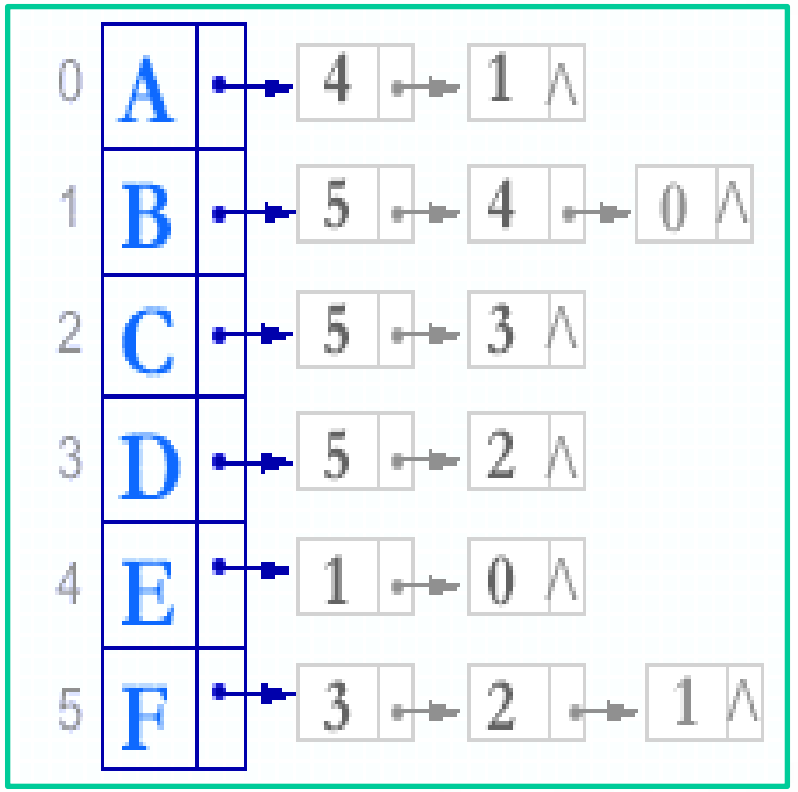



```
#define  MAX_VERTEX_NUM   20
    typedef struct ArcNode {          // 弧结点的结构
      int   adjvex;                   // 该弧所指向的顶点的位置
      struct ArcNode    *nextarc;     // 指向下一条弧的指针
      VRType      weight;             // 与弧相关的权值，无权则为0
           InfoType     *info;        // 指向该弧相关信息的指针
    }ArcNode ;
    typedef struct VNode {            // 顶点结点的结构
      VertexType    data;             // 顶点信息
      ArcNode   *firstarc;            // 指向第一条依附该顶点的弧
     }VNode, AdjList[MAX_VERTEX_NUM];
    typedef struct {                  // 图的邻接表结构定义
      AdjList          vertices;      // 顶点数组
      int    vexnum, arcnum;          // 图的当前顶点数和弧数
      GraphKind          kind;        // 图的种类标志
    } ALGraph;
```

图的存储结构



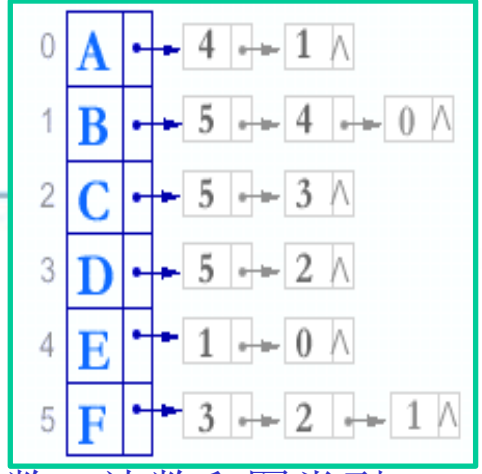
无向图G2



G2.vexnum=6
G2.arcnum=7
G2.kind=UDG
邻接表存储表示

问：N个顶点e条边的邻接表，顶点结点和边结点各有几个？
解：N个顶点结点，2e个边结点。

图的存储结构

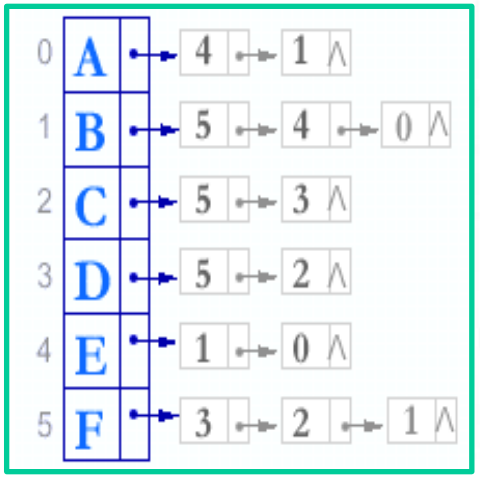


- 算法7.3

```
void CreateGraph(ALGraph &G)
{// 生成图G的存储结构-邻接表
    scanf(&G.vexnum, &G.arcnum, &G.kind);// 输入顶点数、边数和图类型
    for (i=0; i<G.vexnum; ++i) {            // 构造顶点数组
        scanf (G.vertices[i].data);             // 输入顶点
        G.vertices[i].firstarc = NULL;      // 初始化链表头指针为"空“
    }// for
    for (k=0; k<G.arcnum; ++k) {            // 输入各边并构造邻接表
        scanf (&v1, &v2);                       // 输入一条弧的始点和终点
        i = LocateVex(G, v1); j = LocateVex(G, v2);
                    // 确定v1和v2在G中位置，即顶点在G.vertices中的序号
        pi = (ArcNode *) malloc(sizeof(ArcNode));
        if (!pi) exit(1);                   // 存储分配失败
        pi -> adjvex = j;                       // 对弧结点赋邻接点"位置“
```

图的存储结构

```
    if (G.kind==DG || G.kind==DN)
        scanf ( &w, &p);                          // 输入权值和其它信息存储地址
     else { w=0; p=NULL; }
     pi->weight = w; pi->info = p;
     pi -> nextarc = G.vertices[i].firstarc;
     G.vertices[i].firstarc = pi;          // 插入链表G.vertices[i]
     if (G.kind==UDG || G.kind==UDN)
      {                                    // 对无向图或无向网尚需建立v2的邻接点
        pj = (ArcNode *) malloc(sizeof(ArcNode));
        if (!pj) exit(1);                          // 存储分配失败
        pj -> adjvex = i;                          // 对弧结点赋邻接点"位置"
        pj -> weight = w; pj->info = p;
        pj -> nextarc = G.vertices[j].firstarc;
        G.vertices[j].firstarc = pj;       // 插入链表G.vertices[j]
     } // if
   } // for
} // CreateGraph
```



图的存储结构

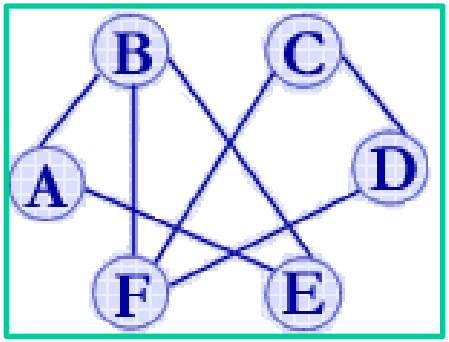
- 例如，按算法7.3生成的无向图G2的邻接表如右下所示：
  - 依次输入的数据为：

    6 7 UDG

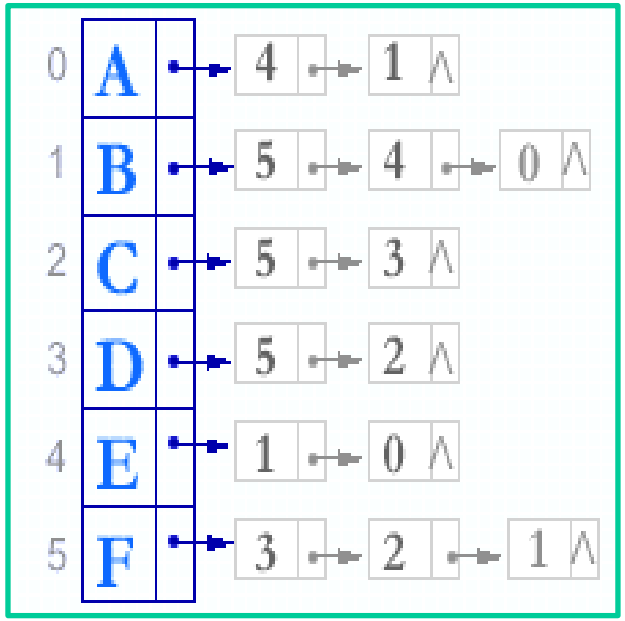
    A B C D E F

    AB AE BE BF CD CF DF

    G2.vexnum=6
    G2.arcnum=7
    G2.kind=UDG



无向图G2

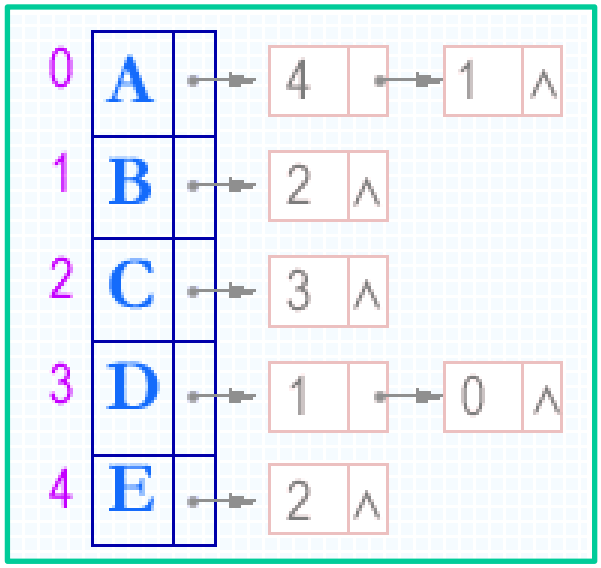


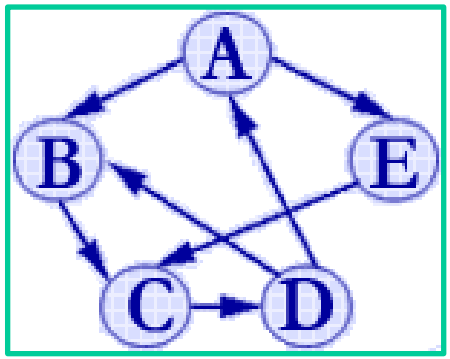
  - 从算法可见，每生成一个弧的结点之后是按“倒插”的方法插入到相应顶点的邻接表中的。并且对于无向图，每输入一条边需要生成两个结点，分别插入在这条边的两个顶点的链表中。即无向图的邻接表中弧结点的个数为图中边的数目的两倍。

图的存储结构

- 例如，按算法7.3生成的有向图G1的邻接表如下所示：

G1.vexnum=5
G1.arcnum=7
G1.kind=DG
邻接表存储表示



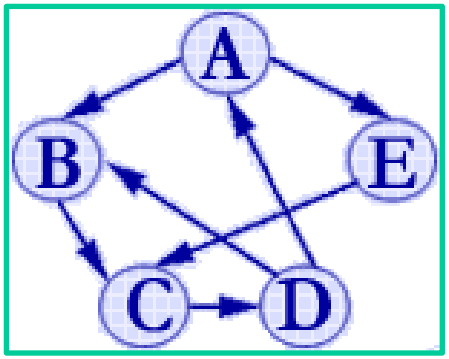


有向图G1

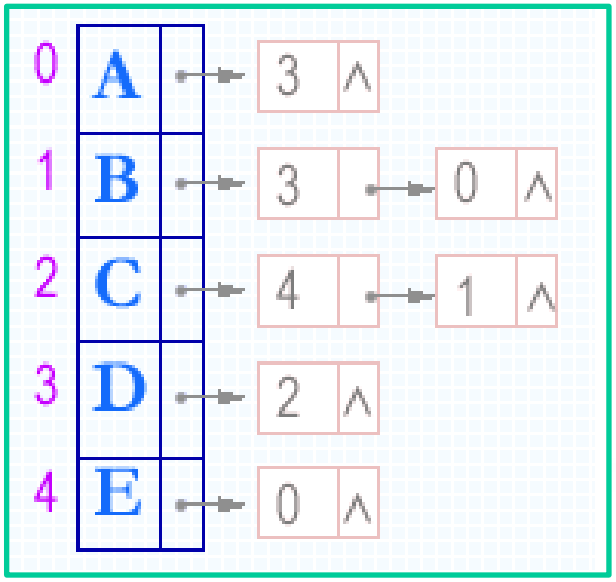
- 有向图的邻接表中，**顶点i的链表中**，若存在链表结点j，则图中弧<i,j>，即：都是以该**顶点(i)为弧尾的弧**。
- 每个单链表中的弧结点的个数恰为弧尾顶点的出度，每一条弧在邻接表中只出现一次。虽然在邻接表中也能找到所有以某个顶点为弧头的弧，但必须查询整个邻接表。

图的存储结构

- 若在应用问题中主要是对以某个顶点为弧头的弧进行操作，则可以为该有向图建立一个“逆邻接表”。例如，有向图G1的逆邻接表如下所示：



有向图G1



G1.vexnum=5
G1.arcnum=7
G1.kind=DG
逆邻接表存储表示

有向图(网)的十字链表存储

●三、有向图(网)的十字链表存储表示

十字链表是**有向图**的另一种**链式存储结构**，目的是将在**有向图的邻接表和逆邻接表中两次出现的同一条弧用一个结点表示**；

●结点结构

➔ 弧结点

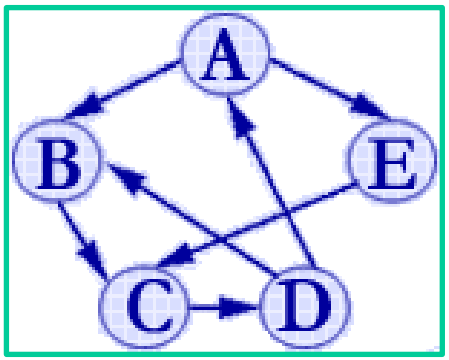
弧尾	弧头	弧头相同的下一条弧	弧尾相同的下一条弧	弧相关信息
tailvex	headvex	hlink	tlink	Info

➔ 顶点结点

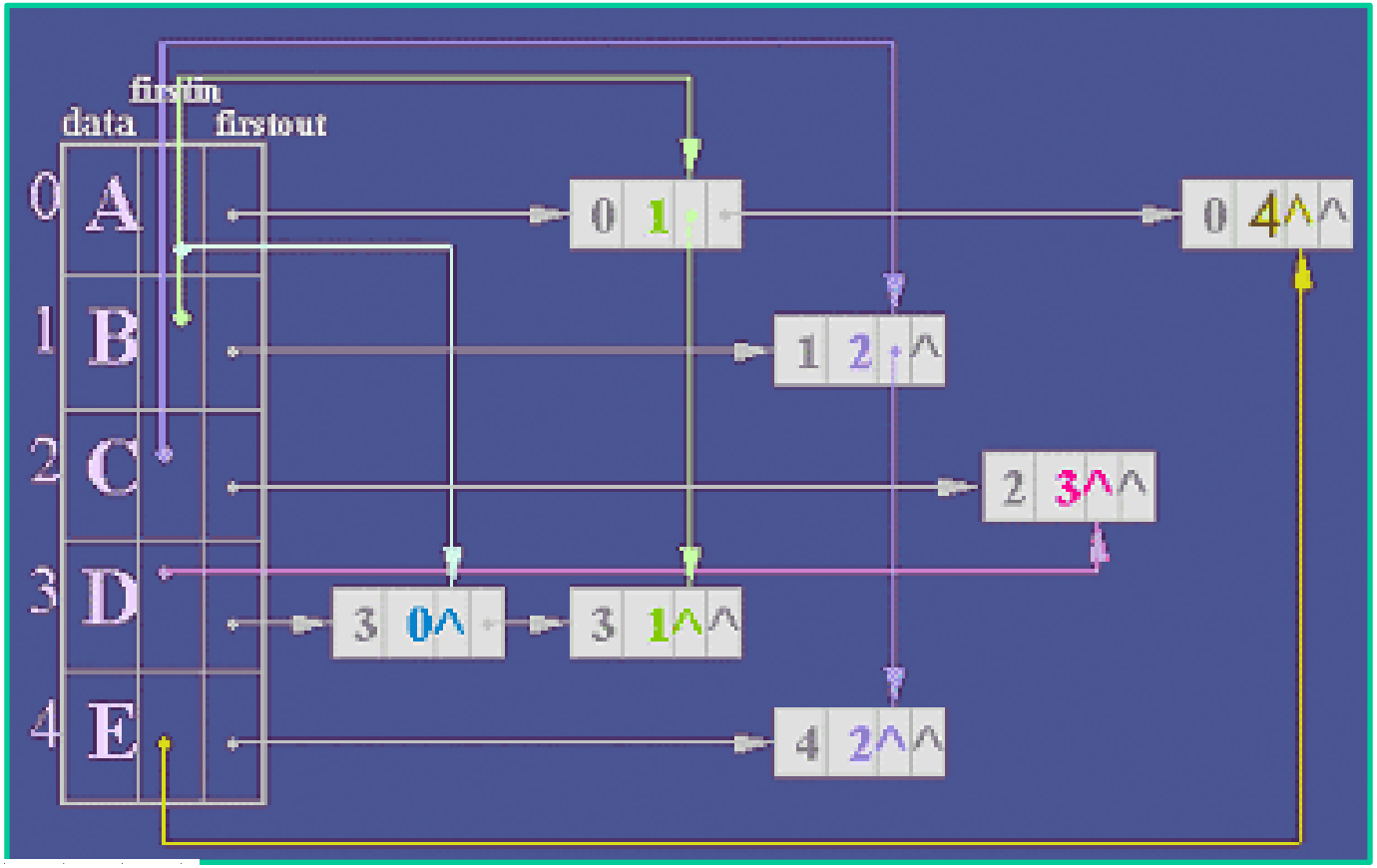
顶点相关信息	以该顶点为弧头的第一个弧结点	以该顶点为弧尾的第一个弧结点
data	firstin	firstout

图的存储结构

- 例如有向图G1的十字链表如下所示（忽略与弧相关的信息指针）。演示 7-2-1.swf



有向图G1



弧结点

弧尾	弧头	弧头相同的下一条弧	弧尾相同的下一条弧	弧相关信息
tailvex	headvex	hlink	tlink	Info

顶点结点

顶点相关信息	以该顶点为弧头的第一个弧结点	以该顶点为弧尾的第一个弧结点
data	firstin	firstout

图的存储结构

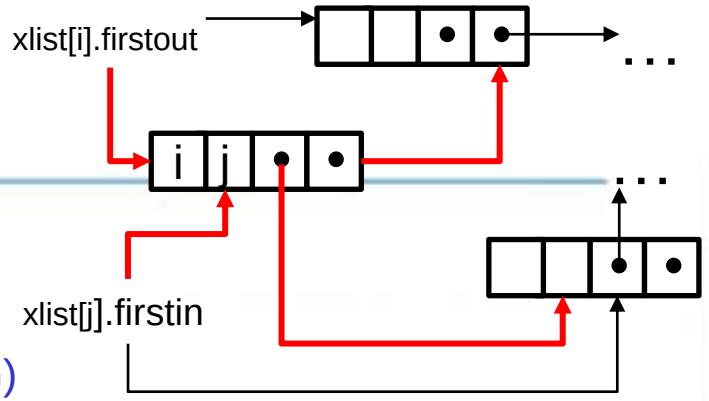
- 从例图可见，有向图的十字链表类似于第5章中讨论的稀疏矩阵的十字链表。图中每个弧结点恰好对应有向图G1的邻接矩阵中的非零元素，可将邻接矩阵看成是一个稀疏矩阵，同一行的非零元构成一个链表，同一列的非零元构成一个链表，行链表和列链表的头指针都合在顶点的结点中。
- 由此在十字链表中不仅容易找到从任意一个顶点出发的弧，也容易找到指向任意一个顶点的弧。

图的存储结构

- 有向图的十字链表存储表示

```
#define  MAX_VERTEX_NUM    20
typedef struct ArcBox {          // 弧结点结构定义
        int       tailvex, headvex;          // 该弧的尾和头顶点的位置
        struct  ArcBox  *hlink, *tlink;      // 分别为弧头相同和弧尾相同的弧的链域
        VRType weight;                       // 与弧相关的权值，无权则为0
        InfoType *info;                    // 该弧相关信息的指针
} ArcBox;
typedef struct VexNode {         // 顶点结点结构定义
        VertexType  data;
        ArcBox   *firstin, *firstout;    // 分别指向该顶点第一条入弧和出弧
    } VexNode;
typedef struct {                 // 十字链表结构定义
        VexNode  xlist[MAX_VERTEX_NUM]; // 表头向量
        int   vexnum, arcnum;            // 有向图的当前顶点数和弧数
        GraphKind    kind;               // 图的种类标志
} OLGraph;
```

图的存储结构



- 算法7.4（课堂）

```
Status CreateDG(OLGraph &G)
{   //采用十字链表存储表示，构造有向图G(G.kind = DG)
    scanf(&G.vexnum, &G.arcnum, &IncInfo);    // IncInfo为0则各弧不含其他信息
    for(i=0; i<G.vexnum; ++i){            // 构造表头向量
        scanf(&G.xlist[i].data);                  // 输入顶点值
        G.xlist[i].firstin = NULL; G.xlist[i].firstout = NULL;  // 初始化指针
    }
    for(k=0;  k<G.arcnum; ++k){                              // 输入各弧并构造十字链表
        scanf(&v1,&v2);                                      // 输入一条弧的始点和终点
        i = LocateVex(G,v1);  j = LocateVex(G,v2);    // 确定v1和v2在G中的位置
        p = (ArcBox * )malloc(sizeof(ArcBox));      // 假定有足够空间
        *p = { i, j, G.xlist[j].firstin, G.xlist[i].firstout, NULL};  // 对弧结点赋值
                                                     // {tailvex,headvex,hlink,tlink,info}
        G.xlist[ j ].firstin = G.xlist[ i ].firstout  = p;   // 完成在入弧和出弧链头的插入
        if(IncInfo)  Input (*p->info);    // 若弧含有相关信息，则输入
    } // for
} // CreateDG
```

图的存储结构

- 只要输入n个顶点的信息和e条弧的信息，便可建立有向图的十字链表。在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(或需要，可在建立十字链表的同时求出)。同时，由算法7.4可知，**建立十字链表的时间复杂度和建立邻接表是相同的**。
- **十字链表 = 邻接表 + 逆邻接表**

无向图(网)的邻接多重链表存储

●四、无向图(网)的邻接多重链表存储表示

邻接表存储表示中，边(Vi,Vj)相关的顶点Vi和Vj分表在第i条链和第j条链中。有时，需要对边进行某种操作，例如标记已搜索过的边。此时，需要分别操作第i和j两条链中结点，操作不便。

类似于有向图的十字链表，若将无向图中表示同一条边的两个结点合在一起，将得到无向图的另一种表示方法--邻接多重表。

●结点结构

- 边结点（mark：标志，是否被搜索过…，ivex、jvex：边(Vi,Vj)所依附的顶点在图中的位置i,j，ilink：下一条依附于顶点ivex的边，jlink：下一条依附于顶点jvex的边，info：与边信息相关的指针）

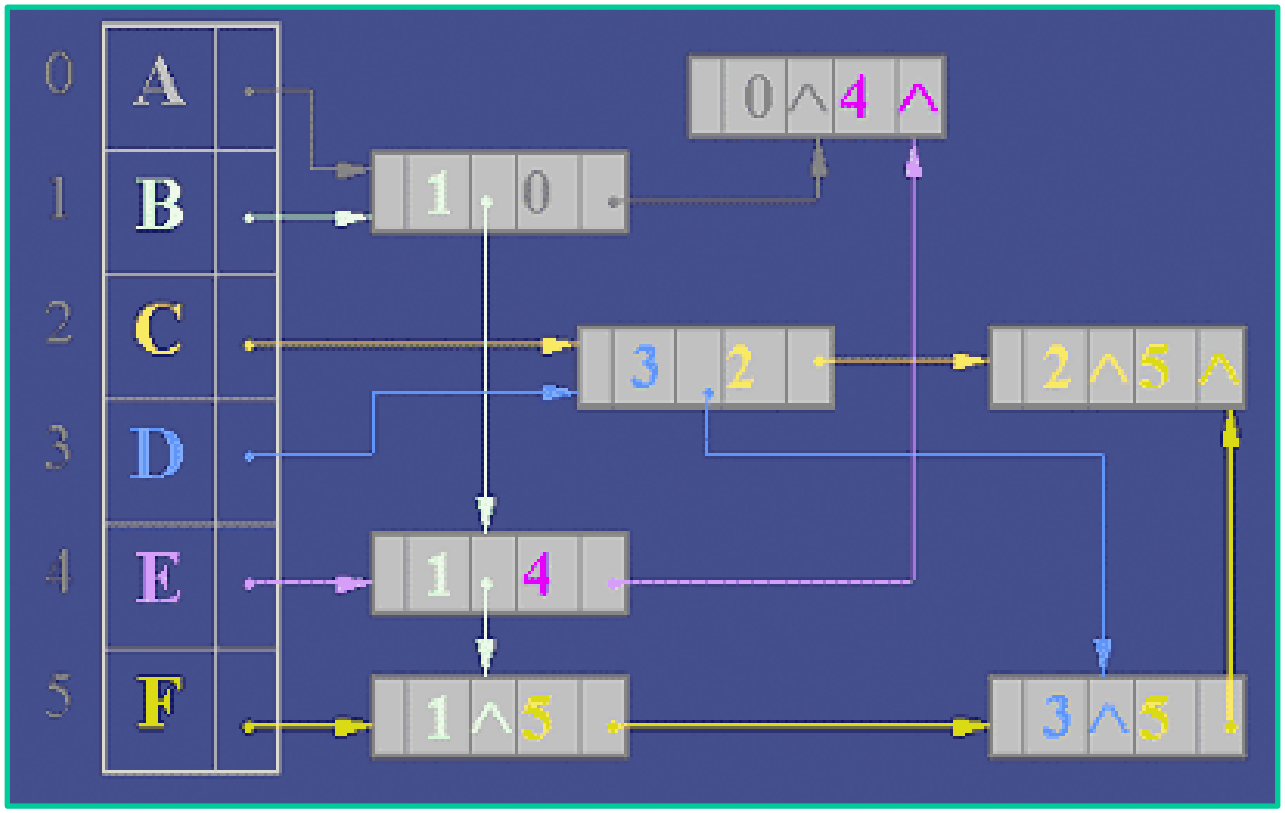
mark	ivex	ilink	jvex	jlink	Info

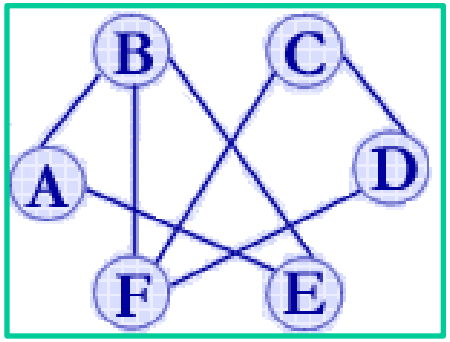
- 顶点结点

data	firstedge

图的存储结构

- 例如，无向图G2的邻接多重表如下所示(忽略相关信息指针)：演示7-2-2.swf





无向图G2

- 边结点（mark：标志，是否被搜索过…，ivex、jvex：边(Vi,Vj)所依附的顶点在图中的位置i,j，ilink：下一条依附于顶点ivex的边，jlink：下一条依附于顶点jvex的边，info：与边信息相关的指针）

mark	ivex	ilink	jvex	jlink	Info

- 顶点结点

data	firstedge

图的存储结构

- 无向图的邻接多重表存储表示

```
#define  MAX_VERTEX_NUM    20
typedef enum {unvisited, visited}   VisitIf;
typedef struct Ebox {                    // 边结点结构定义
        VisitIf      mark;                          // 访问标记
        int          ivex, jvex;                    // 该边依附的两个顶点的位置
        struct  EBox    *ilink, *jlink;        // 分别指向依附这两个顶点的下一条边
        VRType    weight;                       // 与弧相关的权值，无权则为0
        InfoType   *info;                      // 与该边相关信息的指针
} EBox;
typedef struct  VexBox {                       // 顶点结点结构定义
        VertexType     data;
        EBox           *firstedge;                    // 指向第一条依附该顶点的边
} VexBox;
typedef struct {                          // 多重链表结构定义
        VexBox  adjmulist[MAX_VERTEX_NUM];
        int     vexnum, edgenum;                  // 无向图的当前顶点数和边数
      GraphKind        kind;                       // 图的种类标志
} AMLGraph;
```

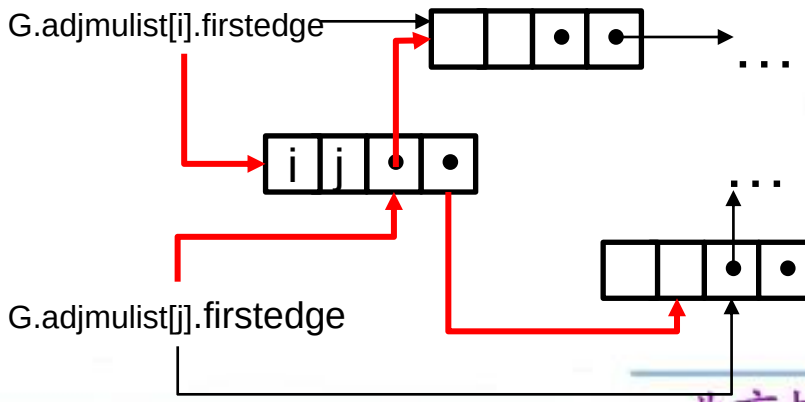
图的存储结构

- 算法7.5

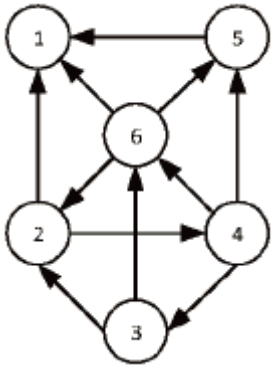
```
void CreateGraph(AMLGraph &G)
{   // 生成无向图G的存储结构-邻接多重表
    scanf( &G.vexnum , &G.edgenum , &G.kind);   // 输入顶点数、边数和类型
    for (i=0; i<G.vexnum; ++i) {                // 构造顶点数组
      scanf( &G.adjmulist[i].data);             // 输入顶点
      G.adjmulist[i].firstedge = NULL;          // 初始化链表头指针为“空”
    }//for
    for (k=0; k<G.edgenum; ++k) {               // 输入各边并构造邻接多重表
      scanf( &v_i , &v_j);                      // 输入一条边的两个顶点
      i = LocateVex(G, v_i ); j = LocateVex(G, v_j);
                                // 确定vi和vj在G中位置，即顶点在G.adjmulist中的序号
      pi = (EBox *)malloc(sizeof(EBox));
      if (!pi) exit(1);                         // 存储分配失败
```

图的存储结构

```
        pi->mark = unvisited;
        pi -> ivex = i; pi->jvex = j;        // 对弧结点赋顶点“位置”
        if (G.kind==UDN)
              scanf( &w, &p);                // 输入权值和其它信息存储地址
        else { w=0; p=NULL; }
        pi->weight = w; pi->info = p;
        pi->ilink = G.adjmulist[i].firstedge;
        G.adjmulist[i].firstedge = pi;       // 插入顶点vi的链表
        pi->jlink = G.adjmulist[j].fistedge;
        G.adjmulist[j].firstedge = pi;       // 插入顶点vj的链表
    } //for
} // CreateGraph
```



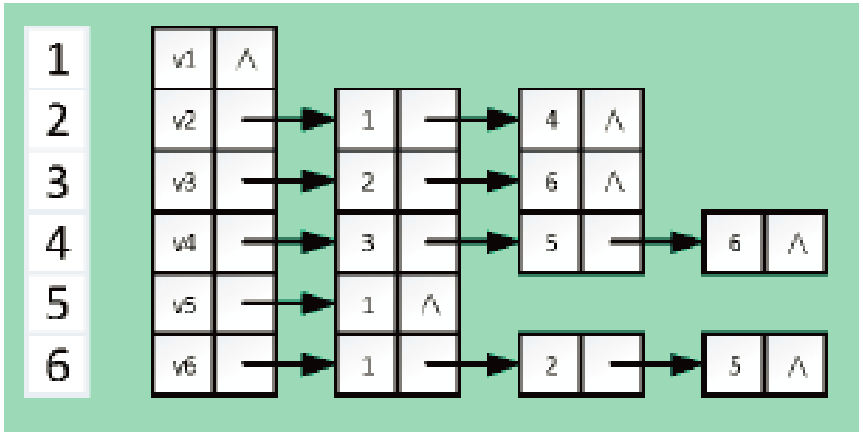
图的存储结构

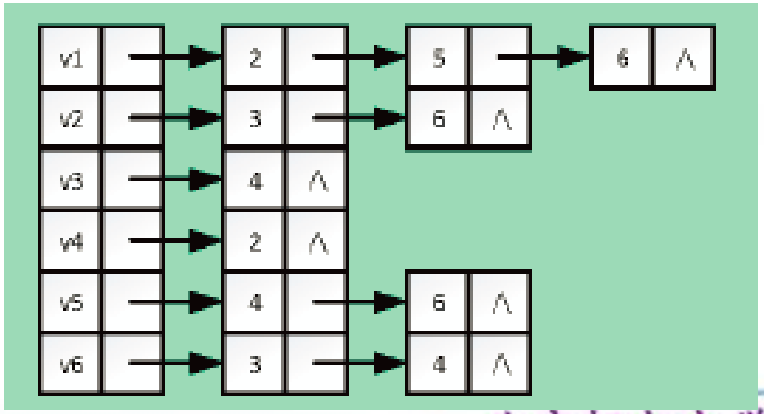


●练习

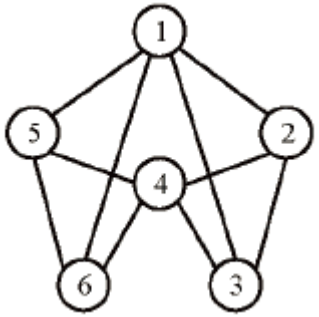
已知如图所示的有向图，请给出该图的邻接矩阵、邻接表（序号0空置，顶点序号按升序排列）、逆邻接表（序号0空置，顶点序号按升序排列）。

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



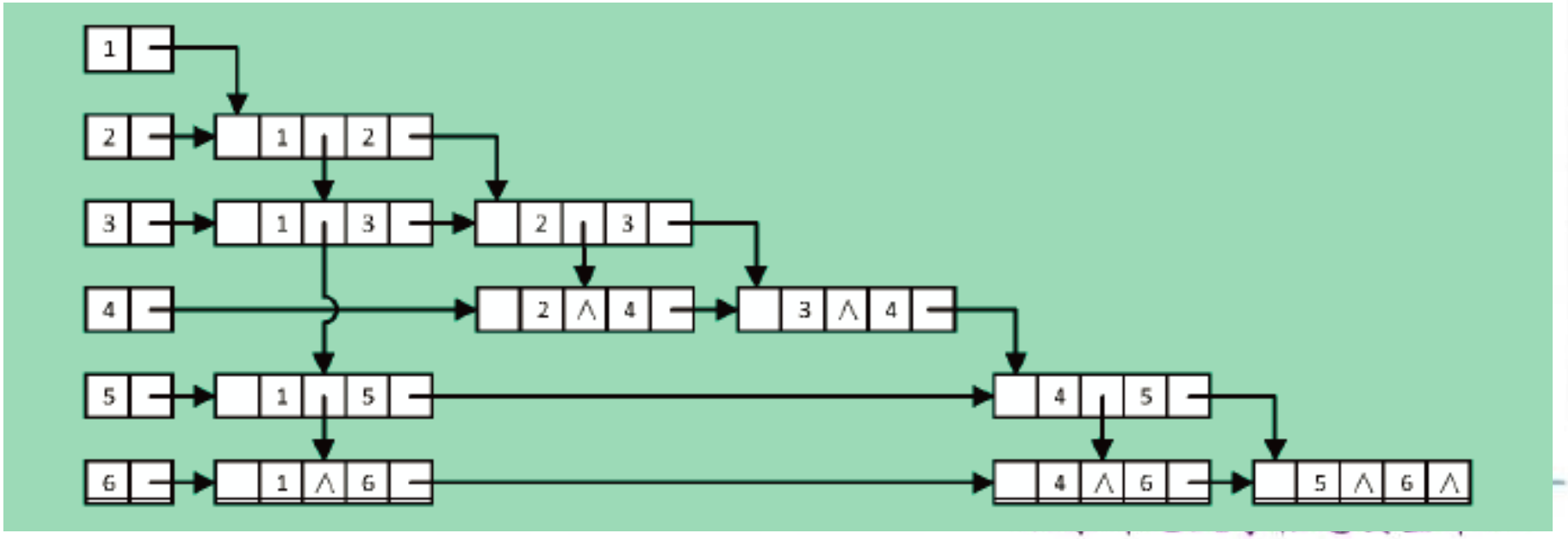


图的存储结构

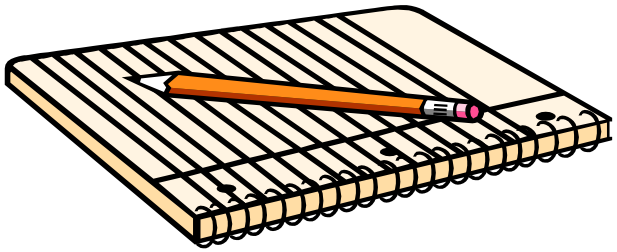


●练习

画出图所示的无向图的邻接多重表，使得其中每个无向边结点中第一个顶点号小于第二个顶点号，且每个顶点的各邻接边的链接顺序，为它所邻接到的顶点序号由小到大的顺序。



图的遍历



- 7.1 图的定义和术语
- 7.2 图的存储结构
- 7.3 图的遍历
- 7.4 图的连通性问题
- 7.5 有向无环图及其应用
- 7.6 最短路径

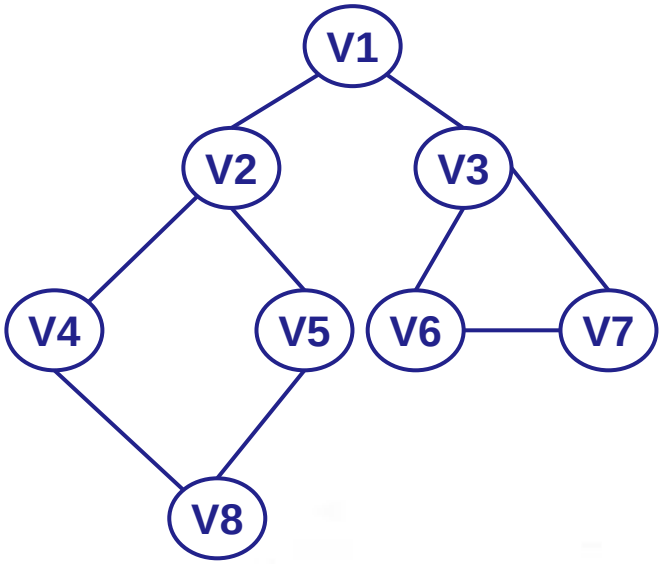
图的遍历

- 与二叉树和树的遍历相同，**图的“遍历”是对图中的每个顶点都进行一次访问且仅进行一次访问**。但由于图结构较树和二叉树更为复杂，图中任意两个顶点之间都可能存在一条弧或边，反之也可能存在某个顶点和其它顶点之间都不存在弧或边。
- 因此对图的遍历而言，除了要确定一条搜索路径之外，还要解决两个问题：
  - (1) **如何确保每个顶点都被访问到**；
  - (2) **如何确保每个顶点只被访问一次**。

图的遍历

●一、深度优先搜索遍历图

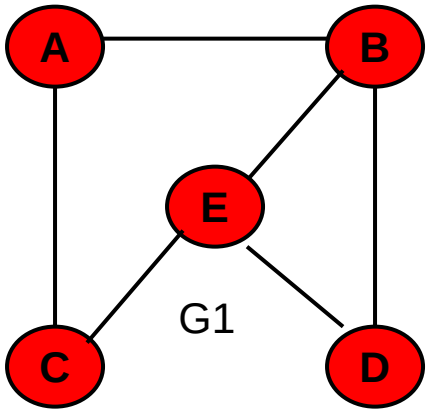
- 扩展树的先根遍历思想运用于图的遍历，得到深度优先遍历方法。
- （1）从图中某个顶点v出发，访问此顶点；
- （2）从v未被访问的邻接点出发，深度优先遍历图，直至所有与v路径相通的顶点都被访问；
- （3）若仍有顶点尚未被访问，则另选图中一个未被访问的顶点出发，重复上述过程，直至图中所有顶点都被访问为止。



例：左图从顶点v1出发深度优先遍历过程为：
V1→V2→V4→V8→V5→V3→V6→V7
（顶点的邻接点有多个时，邻接顶点的访问顺序与邻接点的实际存储顺序有关，为便于讨论，这里按邻接点“升序”访问，例如，V1的邻接点有V2和V3，先访问V2，再访问V3，依次类推）

图的遍历

- 一、深度优先搜索遍历图



例：

G1从顶点A出发深度优先遍历过程为：

A→B→D→E→C

G2从顶点A出发深度优先遍历过程为：

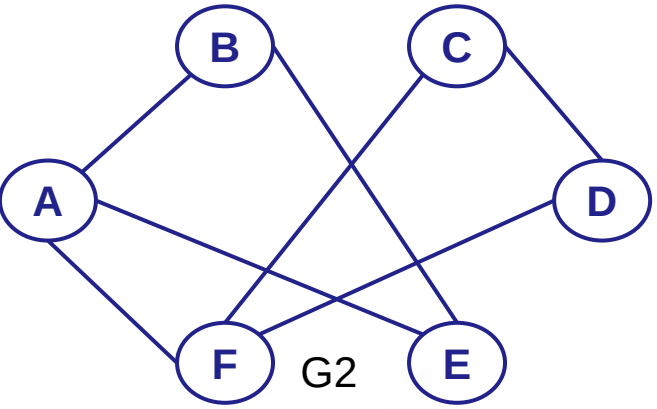
A→B→E→F→C→D

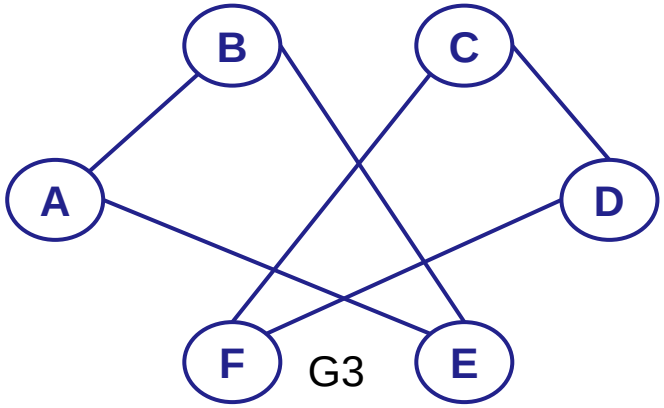
G3先从顶点A，而后从顶点C出发，深度优先遍历过程为：

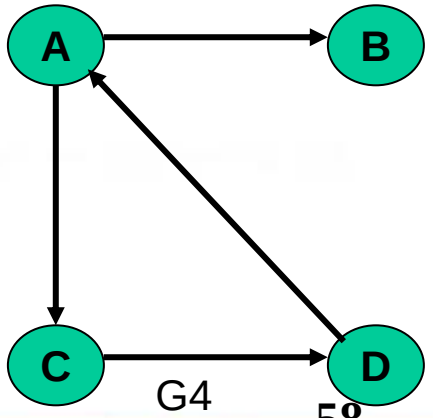
A→B→E→C→D→F

G4从顶点A出发深度优先遍历过程为：

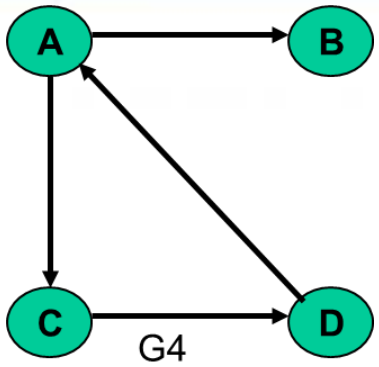
A→B→C→D







图的遍历



●// -------算法7.6和7.7使用的全局变量--------

```
Boolean visited[MAX];      // 访问标志数组
Status (* VisitFunc) (int v);   // 函数变量
```

●算法7.6

```
void  DFSTraverse (Graph G, Status (* visit)(int v)){
                              // 对G作深度优先遍历
     VisitFunc = Visit;   // 使用全局变量VisitFunc，使DFS不必设函数指针参数
     for (v=0; v<G.vexnum; ++v)      visited[v] = FALSE; // 访问标识数组初始化
     for (v=0; v<G.vexnum; ++v)
        if (!visited[v])   DFS(G, v);      // 对尚未访问的顶点调用DFS
}
```

●算法7.7

```
void DFS(Graph G, int v)
{   // 从顶点v出发递归地深度优先遍历图G
      visited[v] = TRUE; VisitFunc(v);                  // 访问顶点 v
      for ( w=FirstAdjVex(G, v); w>=0; w=NextAdjVex(G, v, w) )
         if (!visited[w]) DFS(G, w);      // 对v的尚未访问过的邻接顶点w递归调用DFS
} // DFS
```

广度优先搜索

●二、广度优先搜索遍历图

广度优先搜索(Breadth_First Search)的基本思想是：

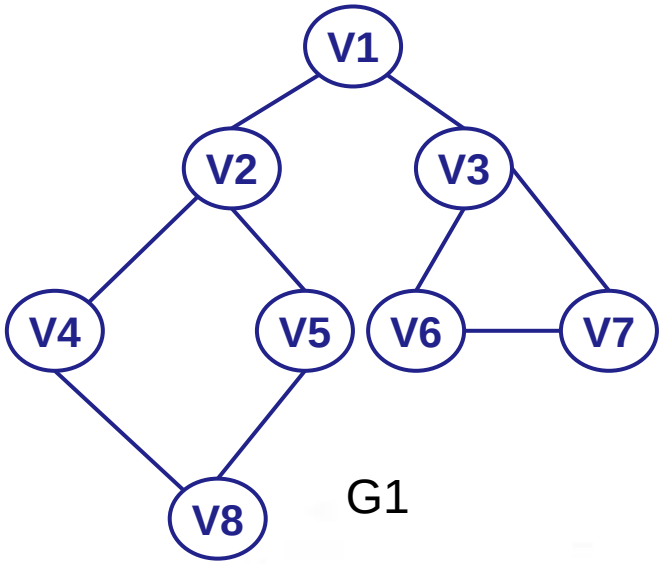
- 从图中某个顶点 v 出发
- 在访问了 v 之后依次访问 v 的各个未曾访问过的邻接点
- 然后分别从这些邻接点出发依次访问它们的邻接点
- 并使得“先被访问的顶点的邻接点”先于“后被访问的顶点的邻接点”进行访问，直至图中所有已被访问的顶点的邻接点都被访问到
- 如若此时图中尚有顶点未被访问，则需另选一个未曾被访问过的顶点作为新的起始点，重复上述过程，直至图中所有顶点都被访问到为止。

●换句话说，广度优先搜索遍历图的过程是以v为起始点，由近至远，依次访问和v有路径相通且最短路径长度为 1,2,… 的顶点。

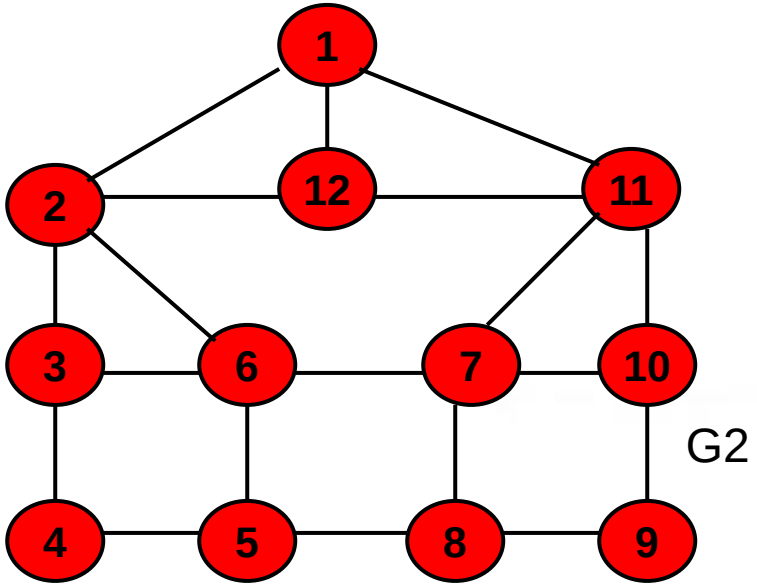
图的遍历

二、广度优先搜索遍历图

- 例：G1从顶点v1出发广度优先遍历过程为：
- V1→V2→V3→V4→V5→V6→V7→V8
- 例：G2从顶点1出发广度优先遍历过程为：（邻接结点的访问次序以序号为准。序号小的先访问。如： 结点 1 的邻接结点有三个 2、12、11，则先访问结点 2、11，再访问结点 12。）
- 1→2→11→12→3→6→7→10→4→5→8→9
- 适用的数据结构：队列



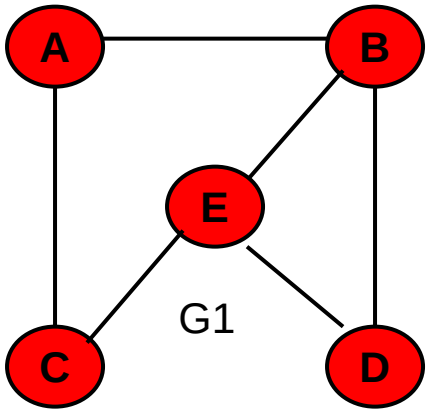
G1



G2

图的遍历

● 二、广度优先搜索遍历图



例：

G1从顶点A出发广度优先遍历过程为：

A→B→C→D→E

G2从顶点A出发广度优先遍历过程为：

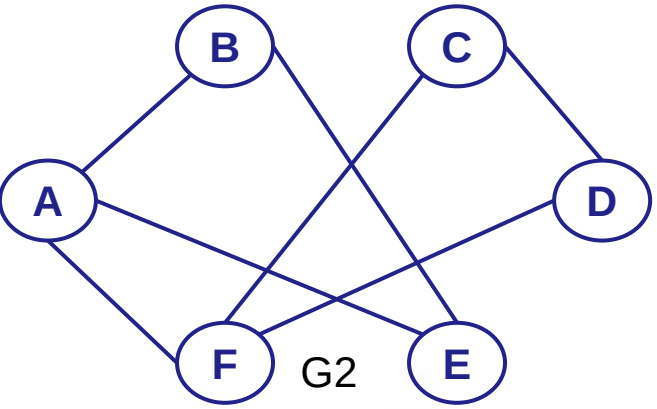
A→B→E→F→C→D

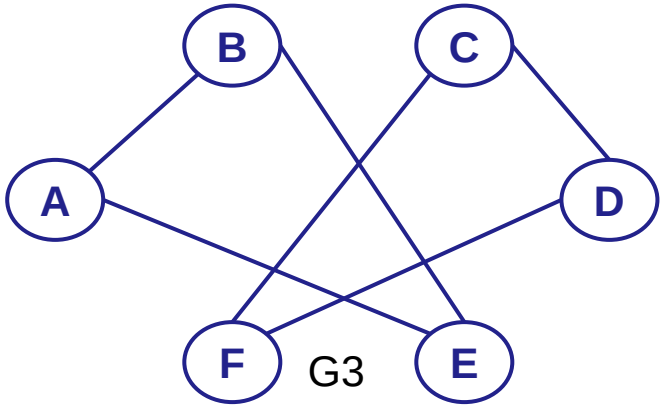
G3先从顶点A，而后从顶点C出发，广度优先遍历过程为：

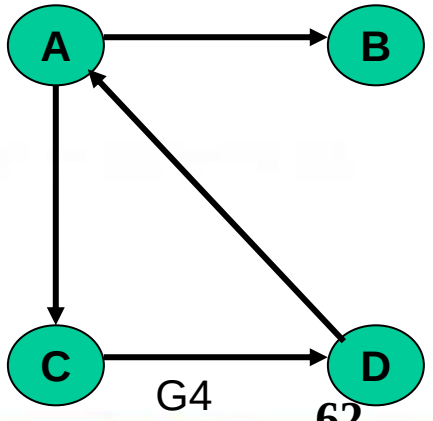
A→B→E→C→D→F

G4从顶点A出发广度优先遍历过程为：

A→B→C→D

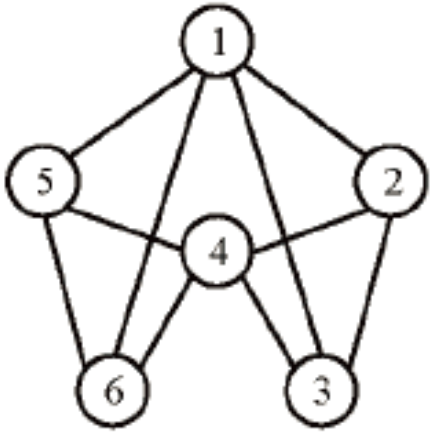


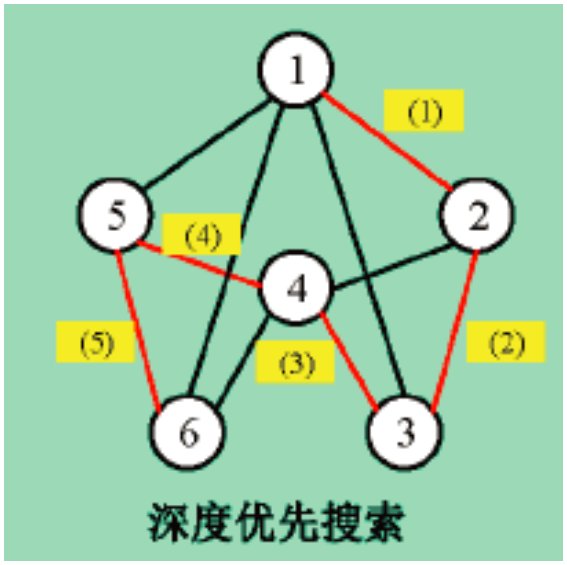


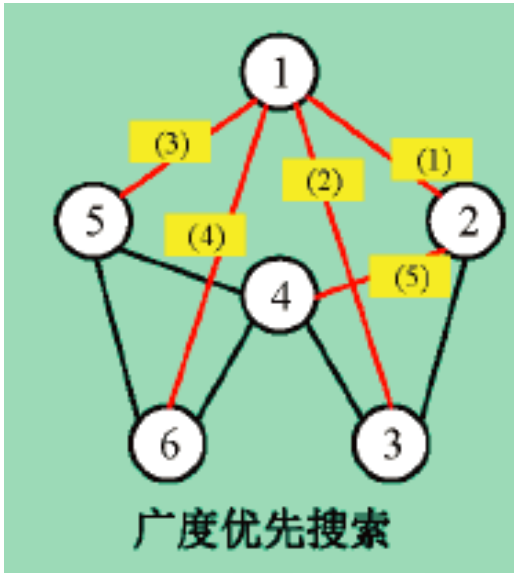


●练习

对于所示无向图，列出深度优先和广度优先搜索遍历该图所得顶点序列和边的序列。



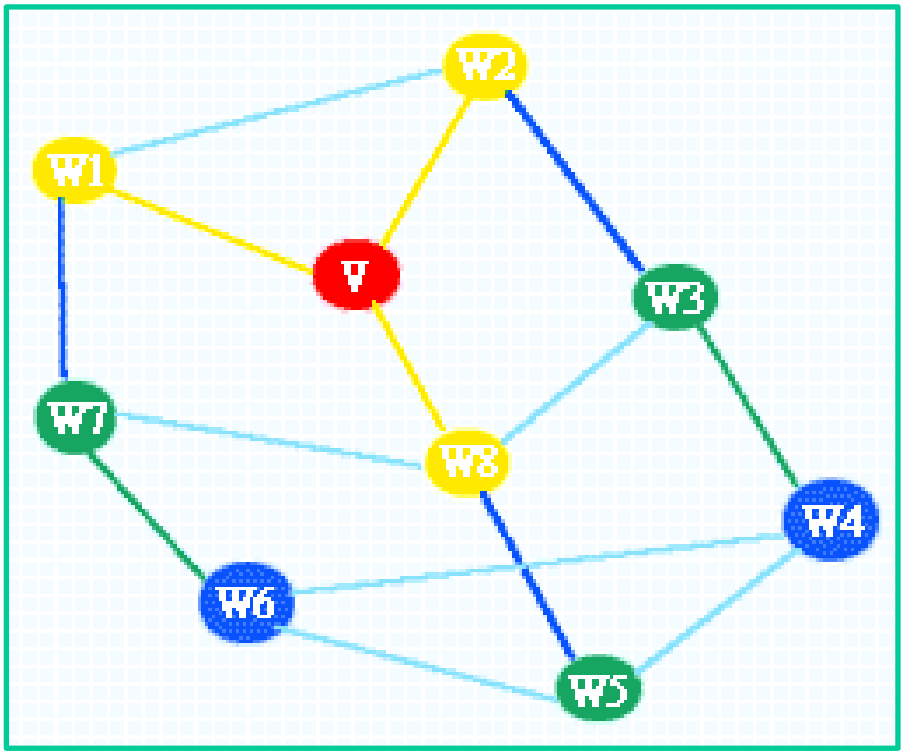




广度优先搜索

- 例如，对下列连通图G4进行广度优先搜索遍历，假设从顶点v开始，则图中其它顶点和v之间都有路径相通，其中从v到w_1、w_2 和w_8 的最短路径为1，从v到w_7、w_3和w_5的最短路径长度为2，从v到w_6和w_4的最短路径长度为3。由此对图G4进行广度优先搜索遍历时顶点的访问次序为：

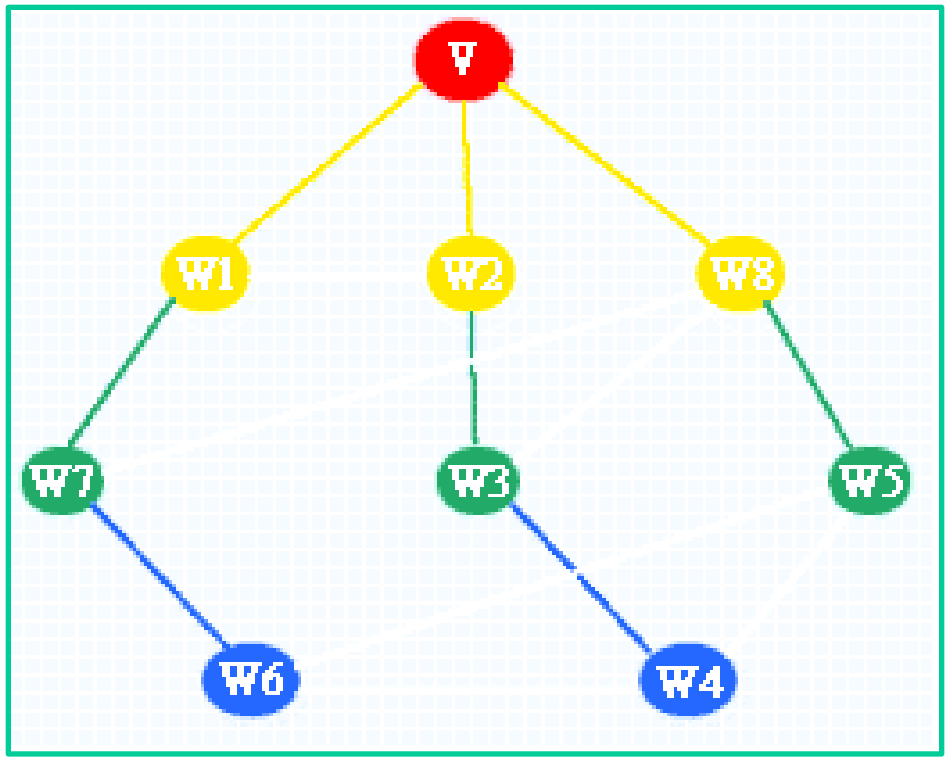
v→ w_1 → w_2 → w_8 → w_7 → w_3 → w_5 → w_6 → w_4 演示



其中，w_7先于w_3和w_5进行访问是因为w_1先于w_2和w_8进行访问。

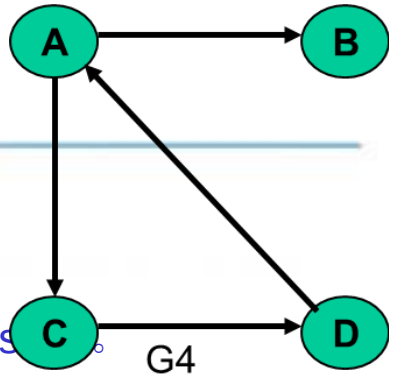
广度优先搜索

- 可以改变布局重新画图 G4，将顶点 v 放在上方中央，其余顶点按从 v 到该顶点的最短路径长度分别放在第2、3和4层上，则图的广度优先搜索遍历的过程类似于树的按层次遍历的过程。演示



➔由于广度优先搜索遍历要求先被访问的顶点的邻接点也先于后被访问的顶点的邻接点进行访问，因此在遍历过程中需要一个队列保存被访问顶点的次序，以便按照已被访问过的顶点的访问次序先后访问它们的未曾被访问过的邻接点。

广度优先搜索



G4

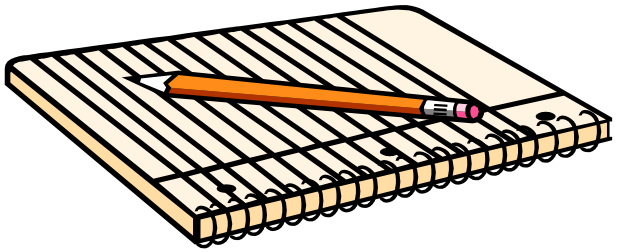
- 算法7.8

```
void BFSTraverse(Graph G, Status (* Visit) (int v))
{   // 按广度优先非递归遍历图G。使用辅助列队Q和访问标志数组vis。
      for (v=0; v<G.vexnum; ++v)   visited[v] = FALSE;
      InitQueue (Q);                                  // 置空的辅助队列 Q
      for (v=0; v<G.vexnum; ++v)
          if (!visited[v]) {                          // v尚未访问
             visited[v] = TRUE;  Visit(v);
             EnQueue(Q,v);                            // v入队列
       while(!QueueEmpty(Q)){
                DeQueue(Q,u);                         // 队头元素出队并置为u
            for ( w = FirstAdjVex(G,u); w >= 0; w = NextAdjVex(G,u,w) )
              if (! visited[w] ) {                    // w 为u尚未访问的邻接顶点
                 visited[w] = TRUE; Visit(w);         // 访问图中第 w 个顶点
                 EnQueue(Q, w);                       // 当前访问的顶点 w 入队列
               } // if
             } // while
          } // if
} // BFSTraverse
```

图的遍历

- 遍历图的过程**实质上是通过边或弧找邻接点的过程**；
- 其消耗时间取决于所采用的存储结构；
- 因此**若采用同样的存储结构，广度优先遍历的时间复杂度和深度优先遍历相同**；
- 两者不同之处仅仅在于对顶点访问的顺序不同。

图的连通性问题



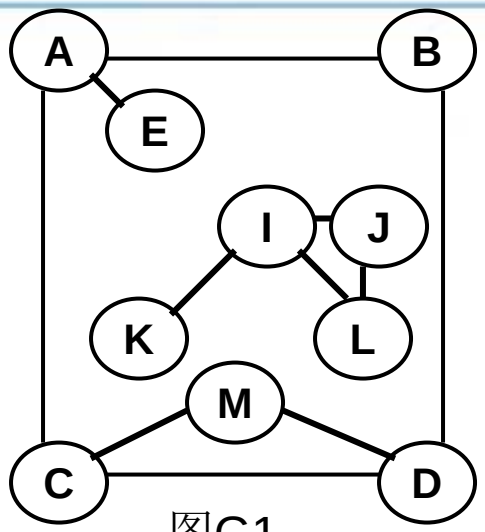
- 7.1 图的定义和术语
- 7.2 图的存储结构
- 7.3 图的遍历
- 7.4 图的连通性问题（一）
- 7.5 有向无环图及其应用
- 7.6 最短路径

图的连通性问题

- 图遍历是图的基本操作，也是一些图的应用问题求解算法的基础，以此为框架可以派生出许多应用算法。
- 例如：
- 在**遍历过程中可以求得非连通图的连通分量或强连通分量**；
- 可利用**深度优先搜索遍历求得图中两个顶点之间一条简单路径**；
- 可利用**广度优先搜索遍历求得两个顶点之间的最短路径**等等。
- 在此仅以求**生成树或生成森林**为例说明图遍历的应用。

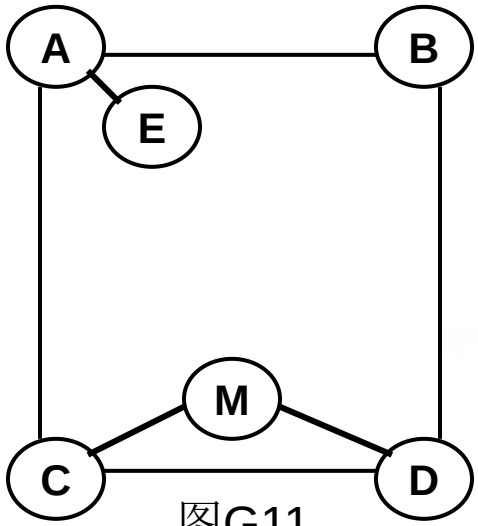
无向图的连通分量和生成树

- 分析遍历算法：
- 对于深度优先遍历算法，从任一顶点v出发，依次深度优先遍历其邻接未访问顶点。
- 显而易见，相邻两次访问的顶点彼此相邻。因此，从任一顶点v出发能访问到的顶点之间必然存在路径。
- 也就是说，第一：
- 对于**连通图**，一次遍历，便可访问**图中所有顶点**；
- 对于**非连通图**，需要从多个顶点出发，**一次遍历所能访问的顶点集是对应连通分量顶点集**。



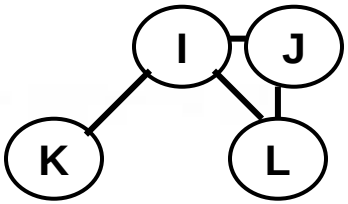
图G1

从A点出发深度优先遍历得到顶点序列：
A->B->D->C->M->E
一次访问顶点集合即为连通分量G11顶点集。



图G11
图G1连通分量1

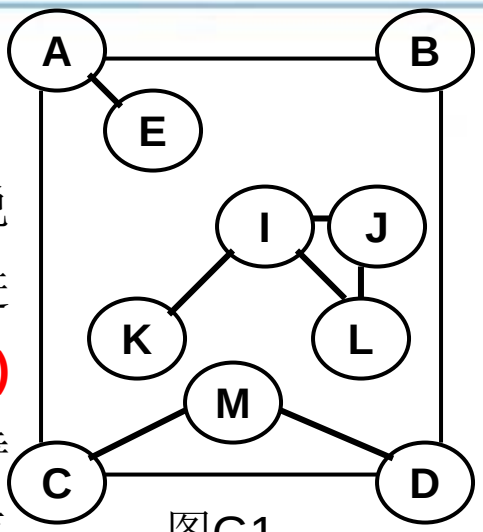
从I点出发深度优先遍历得到顶点序列：
I->J->L->K
一次访问顶点集合即为连通分量G12顶点集。



图G12
图G1连通分量2

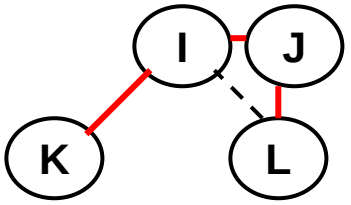
无向图的连通分量和生成树

- 分析遍历算法：
- 第二，任何顶点仅被访问一次，也就是说到任意顶点的路径仅有一条。即，对图进行遍历的过程中，将边分成两个**集合 T(E) 和 B(E)**。其中 T(E) 中的边具有这样的特性：通过它找到 “未被访问” 的邻接点。
- 显而易见，遍历访问所经路径T(E)与一次遍历所能访问顶点集构成极小连通子图。
- 对于**连通图**，该子图就是一棵**生成树**；
- 对于**非连通图**，该子图是一个连通分量的生成树，多次遍历得到**生成森林**。

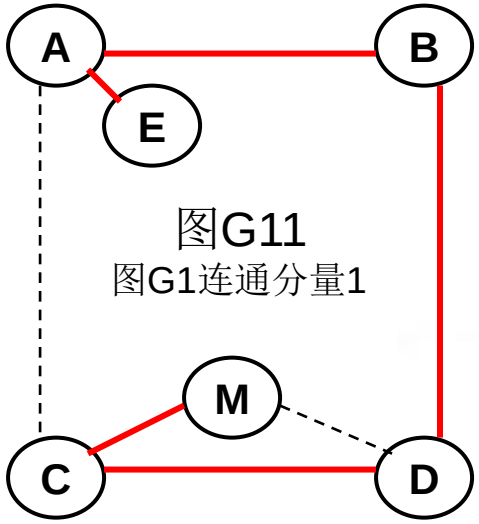


图G1

从I点出发深度优先遍历所经历的边为：
(I,J) (J,L) (I,K)
用红实线和黑虚线标识深度优先遍历经过和没有经过的边，可见一次访问顶点集合与遍历所经历的边组成连通分量G12的生成树。



图G12
图G1连通分量2



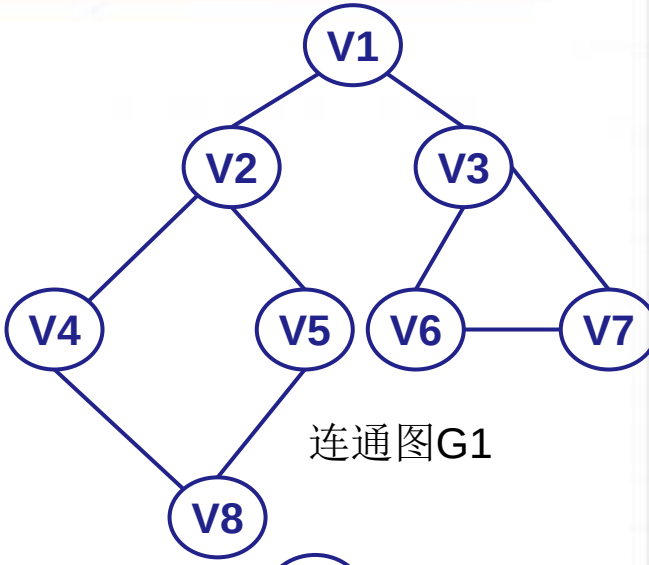
图G11
图G1连通分量1

从A点出发深度优先遍历所经历的边为：
(A,B) (B,D) (D,C) (C,M) (A,E)
用红实线和黑虚线标识深度优先遍历经过和没有经过的边，可见一次访问顶点集合与遍历所经历的边组成连通分量G11的生成树。

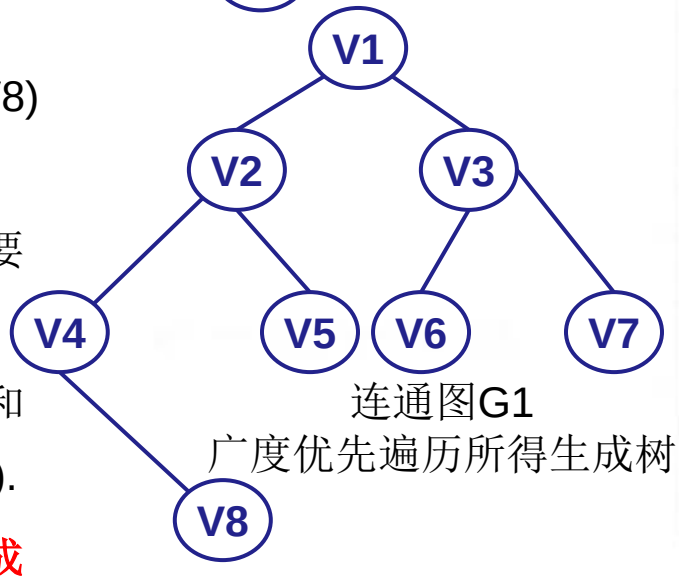
无向图的连通分量和生成树

- 以广度优先遍历时，上述结论也成立。遍历右图过程为：

1. 从顶点V1出发，遍历V1所有邻接点。通过边(V1,V2)和边(V1,V3)找到V1的邻接顶点V2和V3;
2. 遍历V2所有邻接点。通过边 (V2,V4)和边(V2,V5)找到V2的邻接点V4和V5;
3. 遍历V3所有邻接点。通过边 (V3,V6)和边(V3,V7)找到V3的邻接点V6和V7;
4. 遍历V4所有邻接点。因为已访问过顶点V2，仅通过边 (V4,V8)找到V4的邻接点V8;
5. 遍历V5所有邻接点。因为已访问过顶点V8，所以遍历不需要使用边 (V5,V8)，即遍历不经历边(V5,V8);
6. 类似地，遍历V6和V7所有邻接点时，因为已访问过顶点V6和V7,所以遍历不需要使用边 (V6,V7)，即遍历不经历边(V6,V7).
7. 可见，遍历所经历的边与其访问的顶点构成了广度优先生成树。



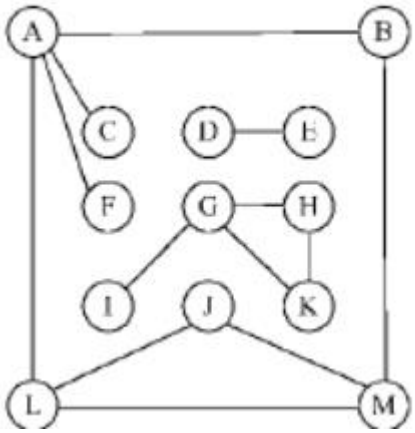
连通图G1

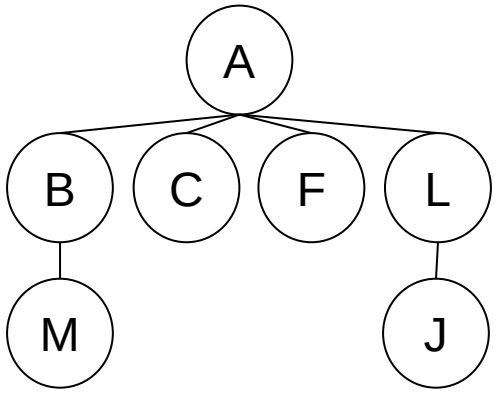


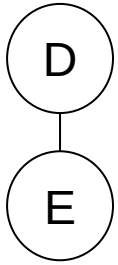
连通图G1
广度优先遍历所得生成树

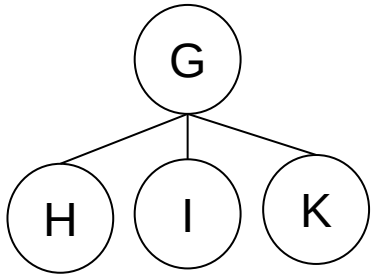
无向图的连通分量和生成树

- 请画出下图的广度优先生成森林





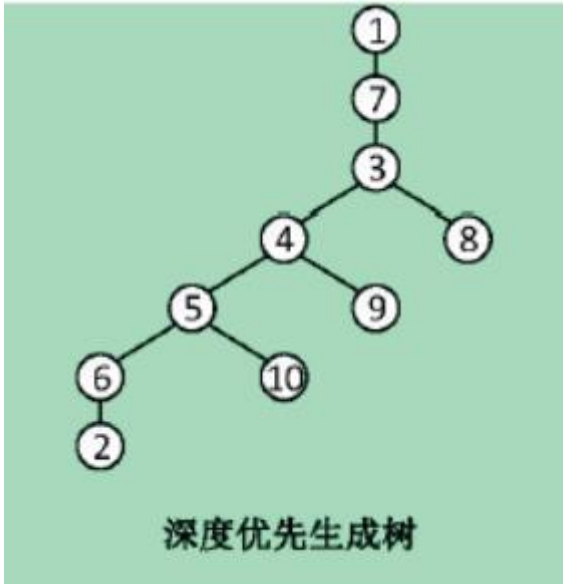




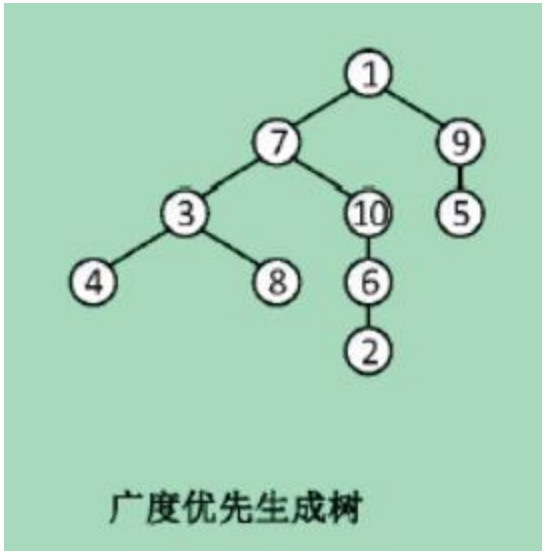
无向图的连通分量和生成树

- 已知以二维数组表示的图的邻接矩阵如下图所示。试分别画出自顶点1出发进行遍历所得的深度优先生成树和广度优先生成树

	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	1	0	1	0
2	0	0	1	0	0	0	1	0	0	0
3	0	0	0	1	0	0	0	1	0	0
4	0	0	0	0	1	0	0	0	1	0
5	0	0	0	0	0	1	0	0	0	1
6	1	1	0	0	0	0	0	0	0	0
7	0	0	1	0	0	0	0	0	0	1
8	1	0	0	1	0	0	0	0	1	0
9	0	0	0	0	1	0	1	0	0	1
10	1	0	0	0	0	1	0	0	0	0



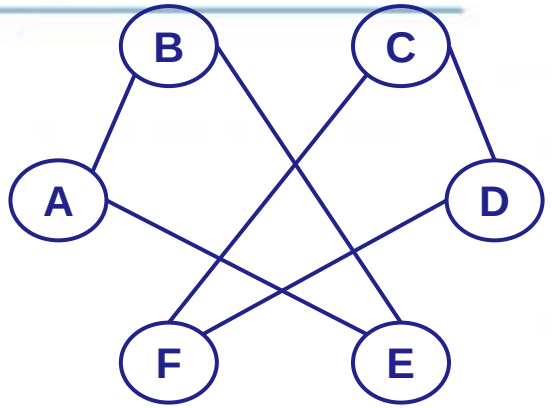
深度优先生成树



广度优先生成树

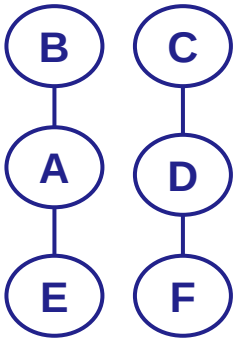
无向图的连通分量和生成树

- 示例2，深度优先遍历右图过程为：

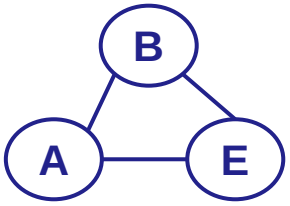


非连通图G1

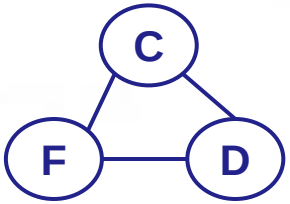
1. 从顶点B出发，通过边(B,A) 找到B的邻接顶点A;
2. 从顶点A，通过边(A,E) 找到A的邻接顶点E;
3. 对于顶点E，因为已访问过顶点B，所以遍历不需要使用边(E,B)，即遍历不经历边(E,B);
4. 相关联的顶点都已处理，遍历逐层从E回溯到A再回溯到B。这样，遍历所经历的边与其能访问的顶点构成了连通分量G11的深度优先生成树。
5. 再次遍历。从顶点C出发，通过边(C,D) 找到C的邻接顶点D;
6. 从顶点D，通过边(D,F) 找到D的邻接顶点F;
7. 对于顶点F，因为已访问过顶点C，所以遍历不需要使用边(F,C)，即遍历不经历边(F,C);
8. 相关联的顶点都已处理，遍历逐层从F回溯到D再回溯到C。这样，遍历所经历的边与其能访问的顶点构成了连通分量G12的深度优先生成树。
9. 可见，通过多次遍历可得到深度优先生成森林。



图G1深度优先生成森林

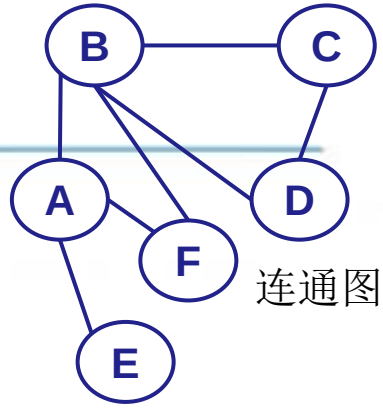


图G1连通分量G11



图G1连通分量G12

深度优先森林的算法

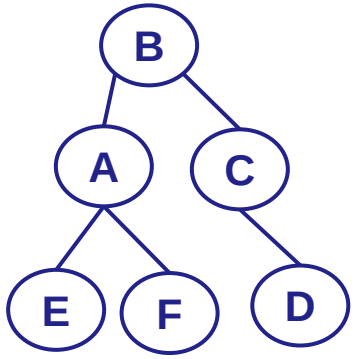


连通图G

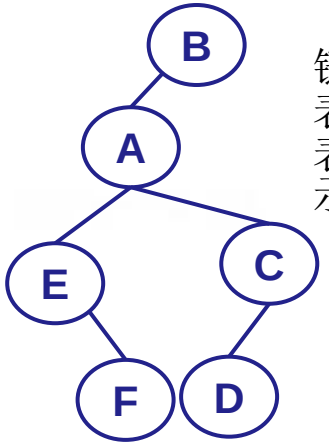
- 根据以上分析可知，**可通过遍历找到图的生成树或生成森林**。
- **树或森林可用孩子兄弟链表链表存储**。若顶点v的邻接点为$u_1,u_2,\ldots u_m$，则u_1是v的左孩子，u_{i+1}是u_i的右孩子（i=2,…,m）。

详细过程如下：

1. 从顶点B出发，B是二叉树的根；深度优先遍历B的邻接点，首先访问顶点A。
2. 当前访问顶点A是上层顶点B的孩子，所以二叉树中，结点A是结点B的左孩子。深度优先遍历A的邻接点，首先访问顶点E。
3. 当前访问顶点E是上层顶点A的孩子，所以二叉树中，结点E是结点A的左孩子。深度优先遍历E的邻接点。E的没有更多未被访问邻接点，深度优先遍历结束，回溯到上一层顶点A，访问A的下一邻接点F。
4. 当前访问顶点F是同层上一访问顶点E的兄弟，所以二叉树中，结点F是结点E的右孩子。深度优先遍历F的邻接点，类似地，回溯到上一层顶点A。A也没有更多未被访问的邻接点了，于是再次回溯到上一层顶点B。访问B的下一邻接点C。
5. 当前访问顶点C是同层上一访问顶点A的兄弟，所以二叉树中，结点C是结点A的右孩子。深度优先遍历C的邻接点，首先访问顶点D。
6. 当前访问顶点D是上层顶点C的孩子，所以二叉树中，结点D是结点C的左孩子。深度优先遍历D的邻接点，类似地，回溯到上一层顶点C。C也没有更多未被访问的邻接点了，于是再次回溯到上一层顶点B。遍历结束。



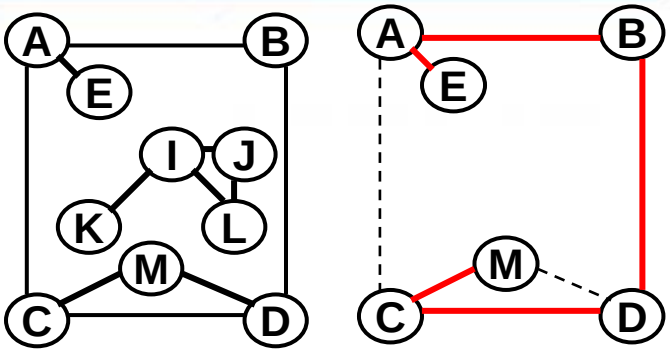
图G的生成树



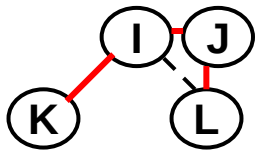
生成树的孩子兄弟链表表示

深度优先森林的算法

图G11
图G1连通分量1



图G1



图G12
图G1连通分量2

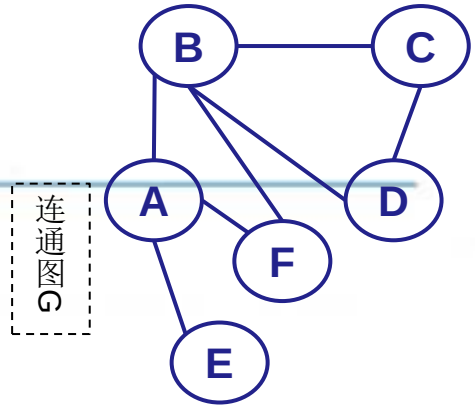
●算法7.9

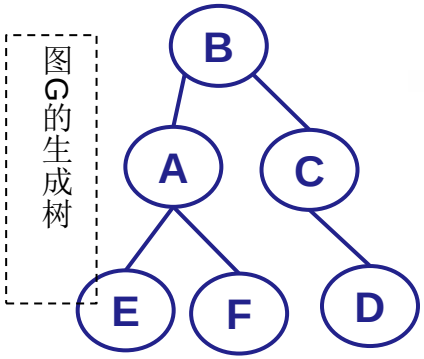
```
void DFSForest(Graph G, CSTree &T)
{   // 建立无向图 G 的深度优先生成森林的(最左)孩子
    // (右)兄弟链表，T 为孩子-兄弟链表的根指针
    T = NULL;
    for (v=0; v<G.vexnum; ++v)   visited[v] = FALSE;
    for (v=0; v<G.vexnum; ++v)
      if (!visited[v]) {                    // 第v顶点为新的生成树的根结点
          p = (CSTree) malloc(sizeof(CSNode));          // 分配根结点
          *p = {GetVex(G,v),NULL,NULL};          // 给该结点赋值
          if (!T) T = p;                          // 是第一棵生成树的根(T的根)
          else q->nextsibling = p;      // 是其它生成树的根(前一棵的根的“兄弟”)
          q = p;                                  // q 指示当前生成树的根
          DFSTree(G,v,p);                         // 建立以 p 为根的生成树
      } // if
} // DFSForest
```

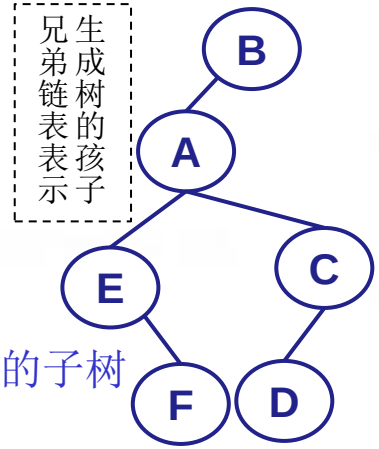
深度优先森林的算法

● 算法7.10

```
void DFSTree(Graph G, int v, CSTree &T)
{   // 从第 v 个顶点出发深度优先遍历图 G，建立以 T 为根的生成树
    visited[v] = TRUE; first =TRUE;
    for (w=FisrtAdjVex(G,v); w>=0; w=NextAdjVex(G,v,w))
      if (!visited[w]) {
        p = (CSTree)malloc( sizeof(CSNode));          // 分配孩子结点
        *p = {GetVex(G,w),NULL,NULL};
        if (first) {                          // w 是 v 的第一个未被访问的邻接顶点
          T->lchild = p; first = FALSE;                // 是根的左孩子结点
        } // if
        else {                                // w 是 v 的其它未被访问的邻接顶点
          q->nextsibling = p;          // 是上一邻接顶点的右兄弟结点
        } // else
        q = p;
        DFSTree(G,w,q);  // 从第 w 个顶点出发深度优先遍历图G,建立以 q 为根的子树
      } // if
} // DFSTree
```







本章小结

- 图是一种比线性表和树更复杂的数据结构。
- 在图形结构中，结点之间的关系可以是任意的，图中任意两个元素之间都可能相邻。
- 和树类似，图的遍历是图的一种主要操作，可以通过遍历判别图中任意两个顶点之间是否存在路径、判别给定的图是否是连通图并可求得非连通图的各个连通分量。

本章知识点与重点

- **知识点**

图的类型定义、图的存储表示、图的深度优先搜索遍历和图的广度优先搜索遍历

- **重点和难点**

图的应用极为广泛，而且图的各种应用问题的算法都比较经典，因此本章重点在于理解各种图的算法及其应用场合。

练习

1. 遍历图的基本方法有_____优先搜索和广度优先搜索。
2. _____优先搜索适于用使用递归方法实现，_____优先搜索适于使用队列实现。
3. 已知图有n个顶点和e条边，若图为有向图，采用邻接表存储，那么边结点有_____个；若图为无向图，采用逆邻接表存储，边结点有_____个，用邻接多重链表存储，边结点有_____个。
4. n个顶点的连通图至少有_____条边，而n个顶点的强连通图至少有_____条弧。
5. 从图中任意顶点出发，一次广度或深度优先遍历就能访问图中所有顶点（）（判断：说法正确写T，错误写F）

练习

6. 已知邻接矩阵定义如下，则这是一个____。请绘制出该图（网）

∞ 3 2 1 ∞

3 ∞ 6 ∞ 5

2 6 ∞ 2 2

1 ∞ 2 ∞ ∞

∞ 5 2 ∞ ∞

6. _____优先搜索适于用使用递归方法实现，_____优先搜索适于使用队列实现。