数据结构

北京邮电大学信息安全中心
武斌

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

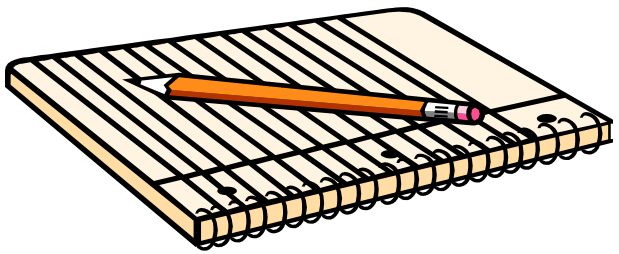
本次课程学习目标

学习完本次课程（图(上)），您应该能够：

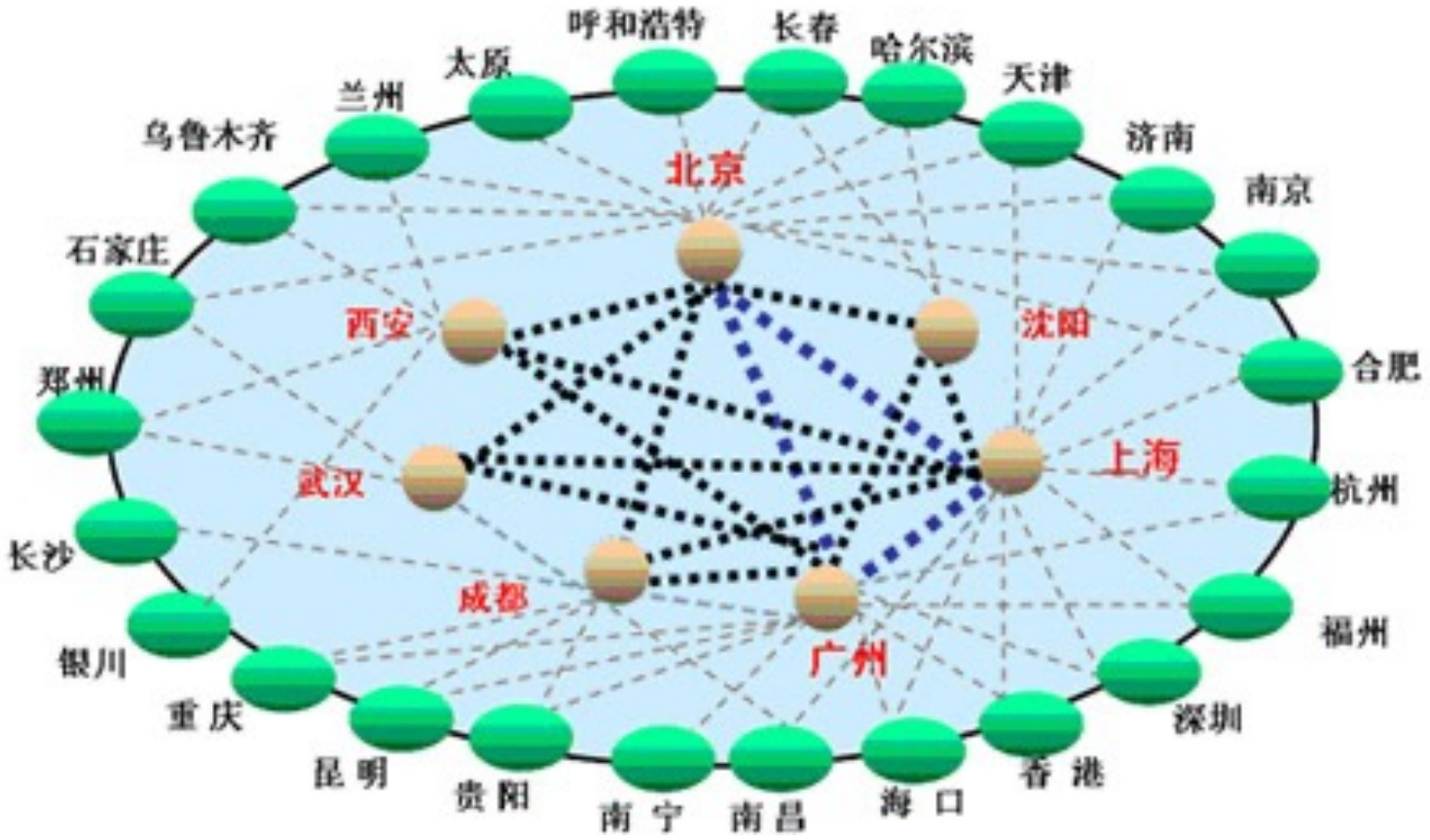
- 领会图的类型定义及术语。
- 熟悉图的各种存储结构及其构造算法，了解各种存储结构的特点及其选用原则。
- 熟练掌握图的两种遍历算法。

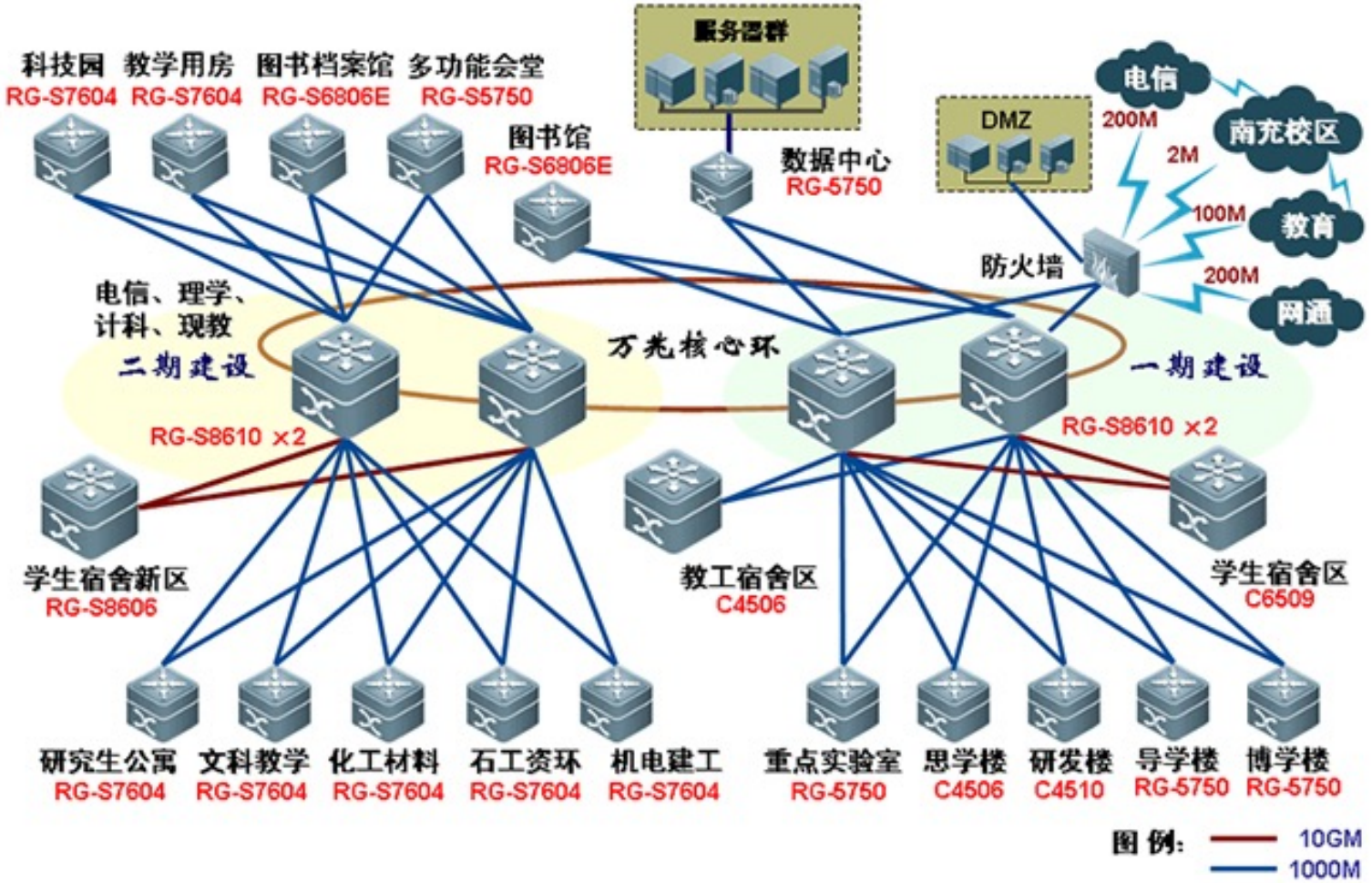
本章课程内容（第七章 图）

- 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

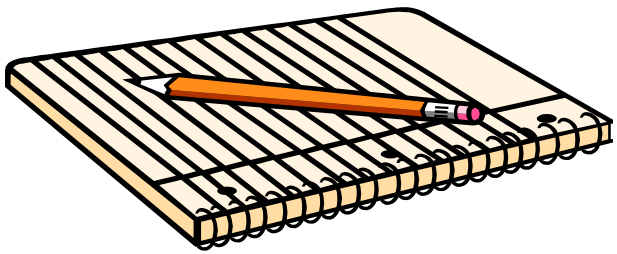



科技园 教学用房 图书档案馆 多功能会堂
RG-S7604 RG-S7604 RG-S6806E RG-S5750
图书馆
RG-S6806E
服务器群
数据中心
RG-5750
DMZ
电信
200M
南充校区
2M
100M
教育
防火墙
200M
网通
电信、理学、
计科、现教
二期建设
万兆核心环
一期建设
RG-S8610 ×2
RG-S8610 ×2
学生宿舍新区
RG-S8606
教工宿舍区
C4506
学生宿舍区
C6509
研究生公寓 文科教学 化工材料 石工资环 机电建工
RG-S7604 RG-S7604 RG-S7604 RG-S7604 RG-S7604
重点实验室 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

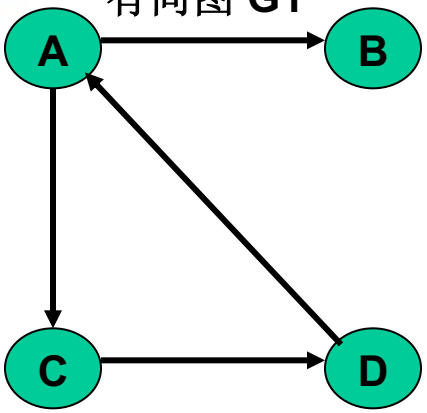


图的定义和术语

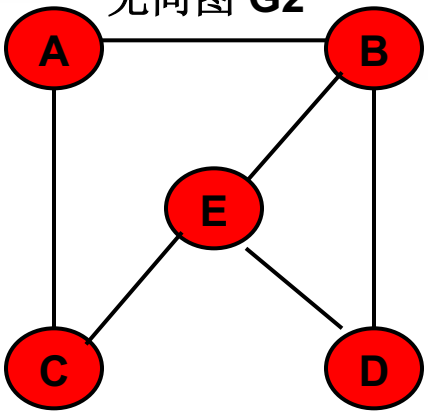
- **定义**：图由一个顶点集和弧集构成，通常写作：Graph=(V,VR)。由于空的图在实际应用中没有意义，因此一般不讨论空的图，即V是顶点的有穷非空集合，而VR是两个顶点之间的关系的集合。
  - 若< v, w >∈VR，则< v, w >表示从v到w的一条**弧(Arc)**，且称v为**弧尾**(Tail)或初始点(Initial Node)，称w为**弧头**(Head)或终端点(Terminal Node)，此时的图称为**有向图**(Digraph)。
  - 若< v, w >∈VR，必有若< w, v >∈VR，即VR是对称的，则以无序对(v, w)代替这两个有序对，表示v和w之间的一条**边**(Edge)，此时的图称为**无向图**(Undigraph)。

图的定义和术语

有向图 G1



无向图 G2



• 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>}

• 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}

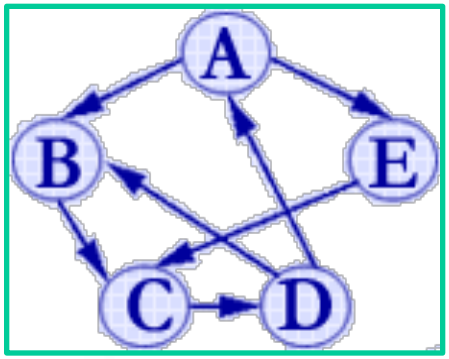
VR1 ={<A,B>,<A,E>,<B,C>,<C,D>,<D,B>,<D,A>,<E,C>}

●例如：下列定义的无向图如右下图所示。

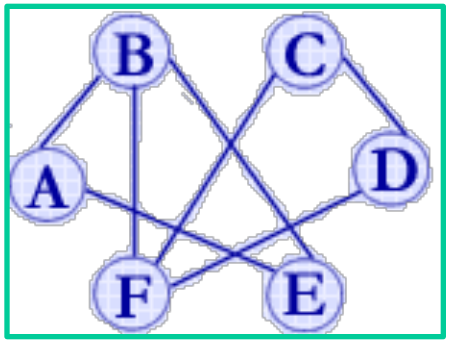
G2=(V2, VR2)

其中：V2={A, B, C, D, E, F}

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

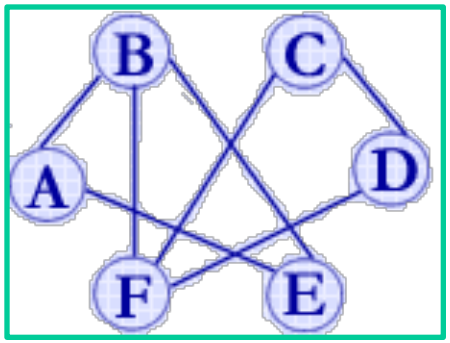
操作结果：对图G进行广度优先遍历。遍历过程中对每个顶点调用函数 Visit 一次且仅一次。一旦 visit() 失败，则操作失败。

图的定义和术语

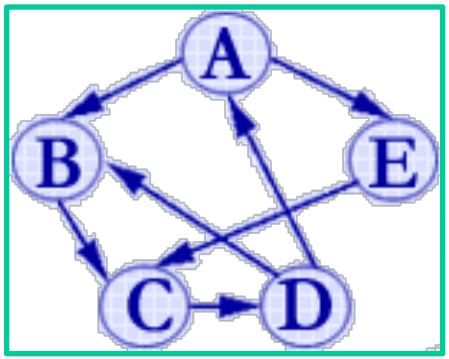
- **完全图**：有 n(n-1)/2 条边的无向图。其中 n 是结点个数。
- **有向完全图**：有 n(n-1) 条边的有向图。其中 n 是结点个数。
- 有很少的边或弧（e<nlogn）的图称为**稀疏图**，反之成为**稠密图**。
- 边和弧相关的数叫做边的**权**值，边有权的图称之为**网**。
- **邻接点**：无向图G=(V,{E})，如果边(V,V')∈E，则称V和V'互为邻接点。边(V,V')和顶点V，V'相关联。

顶点的度

- 顶点的度
  - 对无向图而言，邻接点的个数定义为顶点的度（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个数，入度（ID）定义为以该顶点为弧头的弧的个数。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

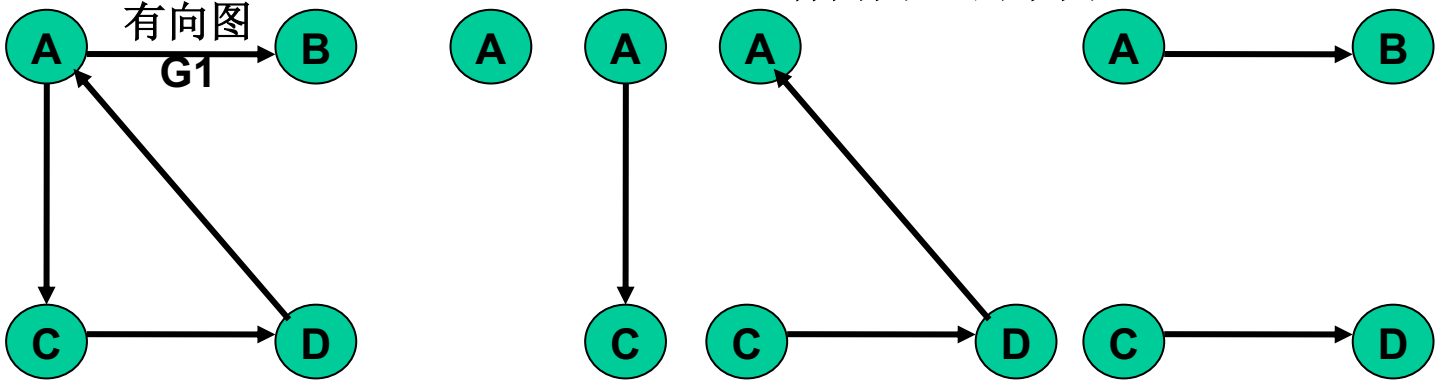


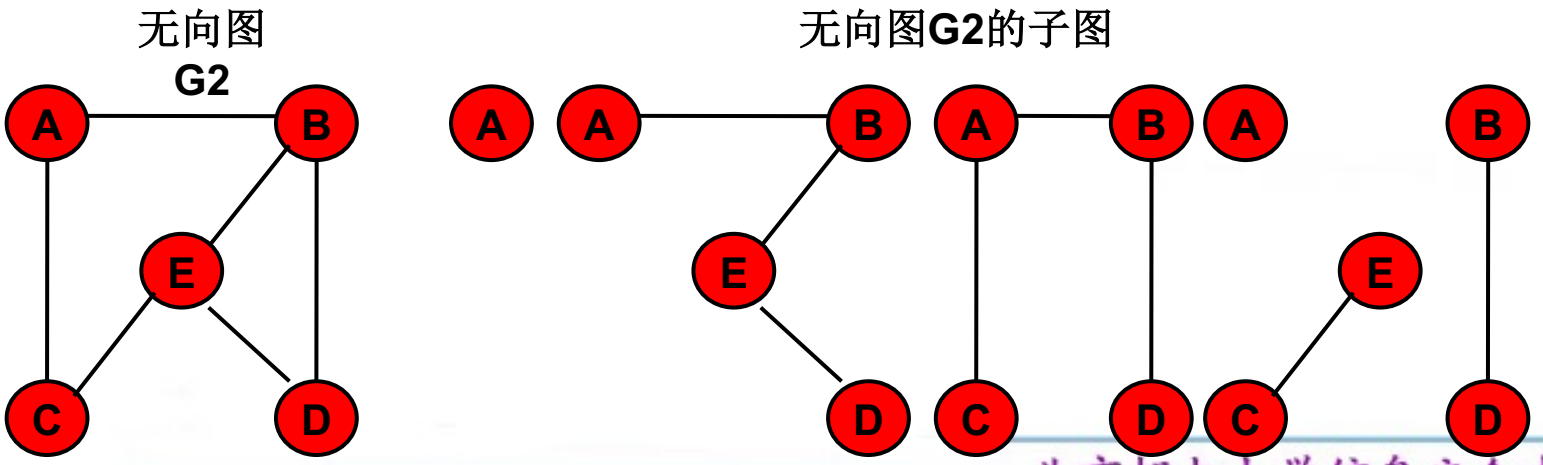
有向图G1

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$\{v_0, v_1,\ldots, v_k\}$为从顶点v_0到顶点v_k的一条有向路径。
- 对无向图，相邻顶点之间存在边的k+1个顶点序列构成一条长度为k的无向路径。
- 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。
- 如果v_0和v_k是同一个顶点，则是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。
- 例如：前页有向图中顶点序列{A,E,C,D}是一条路径长度为3的简单路径，顶点序列{A,B,C,D,A}是一条长度为4的简单回路。而前页无向图中顶点序列{A,B,F,C,D}是一条长度为4的无向路径。

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。

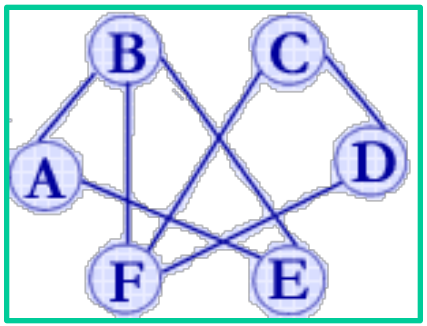


图1 无向图G2

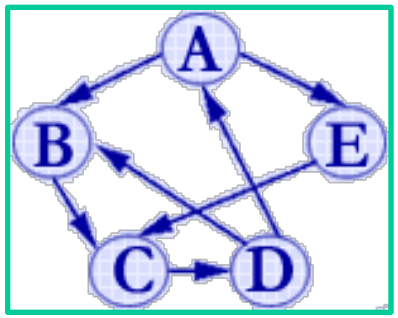


图2 有向图G1

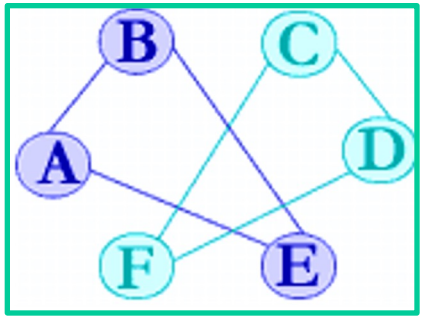


图3 非连通图

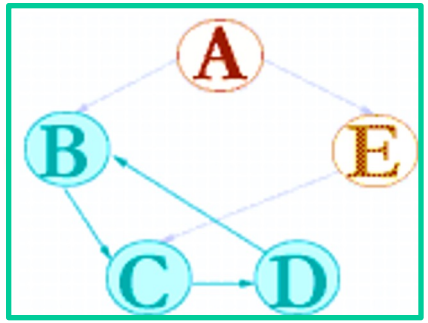


图4 强连通分量

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5所示。若在无向图 G2 中删除边 (B,F)，则成为一个非连通图，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。

● 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。

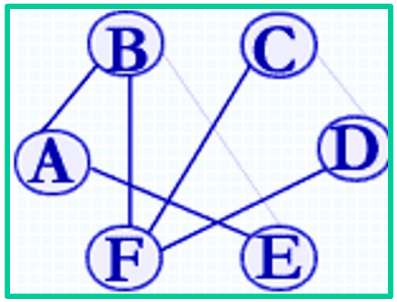


图5

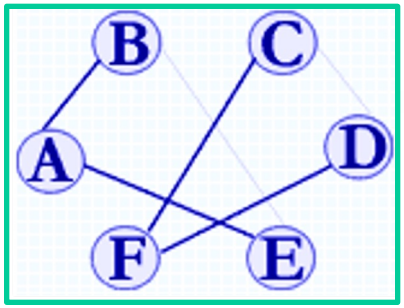
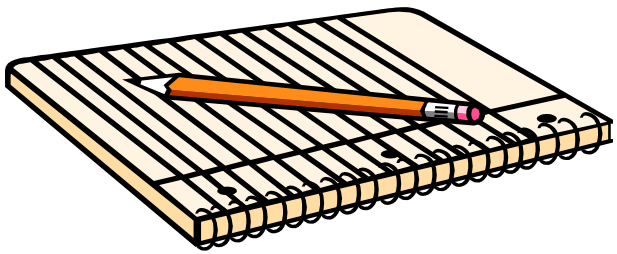


图6

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

- 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。

图的存储结构

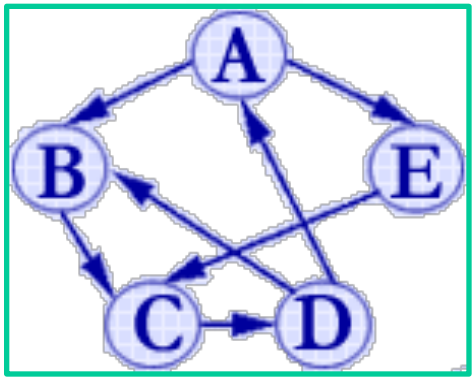
●一、图的数组(邻接矩阵)存储表示

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）
  - G1.vexnum=5
  - G1.arcsnum=7
  - G1.kind=DG



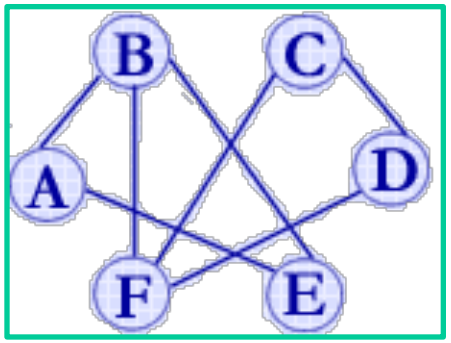
有向图G1

图的存储结构

- 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。
- 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"。

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
```

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
```

图的存储结构

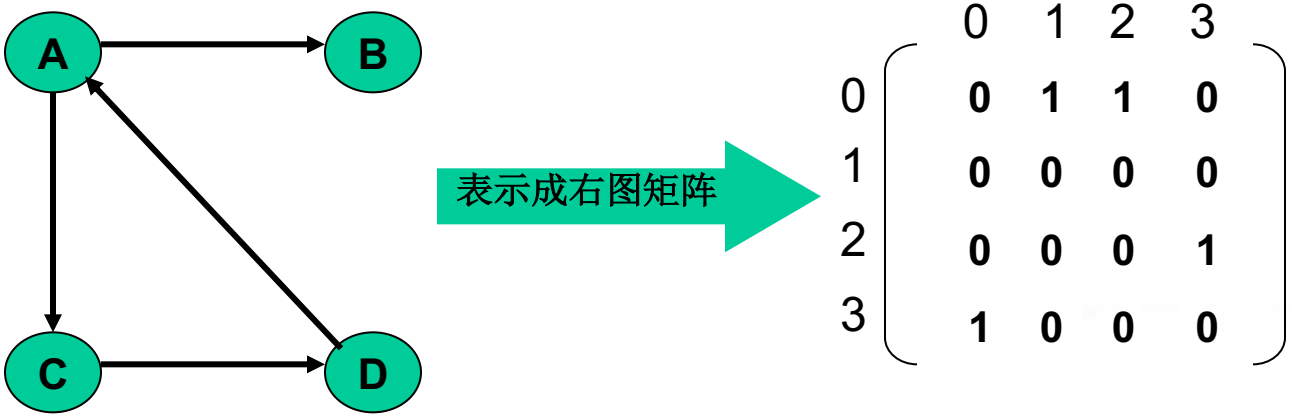
- 算法7.1是在邻接矩阵存储结构MGrpah上对图的构造操作的实现框架，它根据图G的种类调用具体构造算法。如果G是无向网，则调用后页算法7.2。
- 构造一个具有**n个顶点**和**e条边**的**无向网G**的时间复杂度是***O*(n²+e×n)**，其中对邻接矩阵G.arcs的初始化耗费了*O*(n²)的时间。

图的存储结构

●无权值的有向图的邻接矩阵

设有向图具有 n 个结点，则用 n 行 n 列的布尔矩阵 A 表示该有向图；

如果i 至 j 有一条有向边，并且 A[i,j] = 1 ,；如果 i 至 j 没有一条有向边，A[i,j] = 0 。 注意： **A[i,i] = 0**。**出度: i行之和。入度: j列之和**。



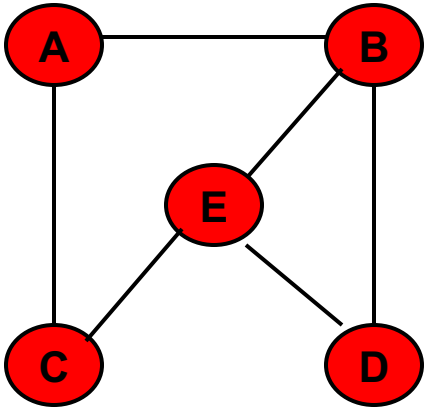
图的存储结构

●无权值的无向图的邻接矩阵

设无向图具有 n 个结点，则用 n 行 n 列的布尔矩阵 A 表示该无向图；

如果i 至 j 有一条无向边， A[i,j] = 1 ,；如果 i 至 j 没有一条无向边，A[i,j] = 0

注意： A[i,i] = 0。i 结点的度: i 行或i 列之和。



表示成右图矩阵

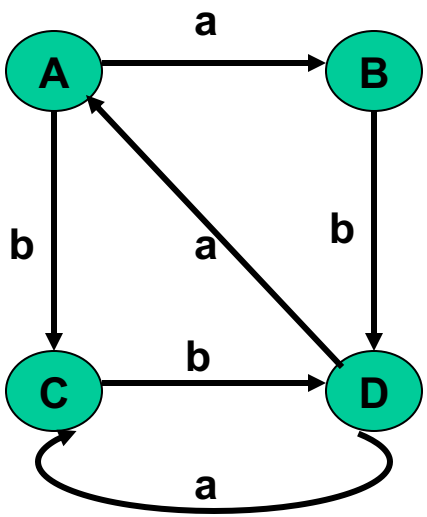
$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

图的存储结构

●有向图的加权邻接矩阵

设有向图具有 n 个结点，则用 n 行 n 列的矩阵 A 表示该有向图；

如果i 至 j 有一条有向边且它的权值为a，A[i,j] = a ；如果 i 至 j 没有一条有向边，A[i,j] = 空 或其它标志。



表示成右图矩阵

$$\begin{pmatrix} n & a & b & n \\ n & n & n & b \\ n & n & n & b \\ a & n & a & n \end{pmatrix}$$

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。

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。

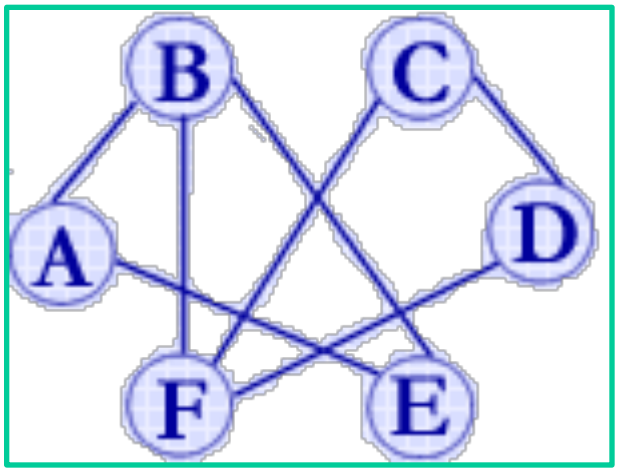
- 对图中每个顶点建立一个单链表，第i个单链表中的结点表示依附于顶点Vi的边或弧；
- 表结点由三个域组成：邻接点域、链域、数据域；

adjvex	nextarc	info

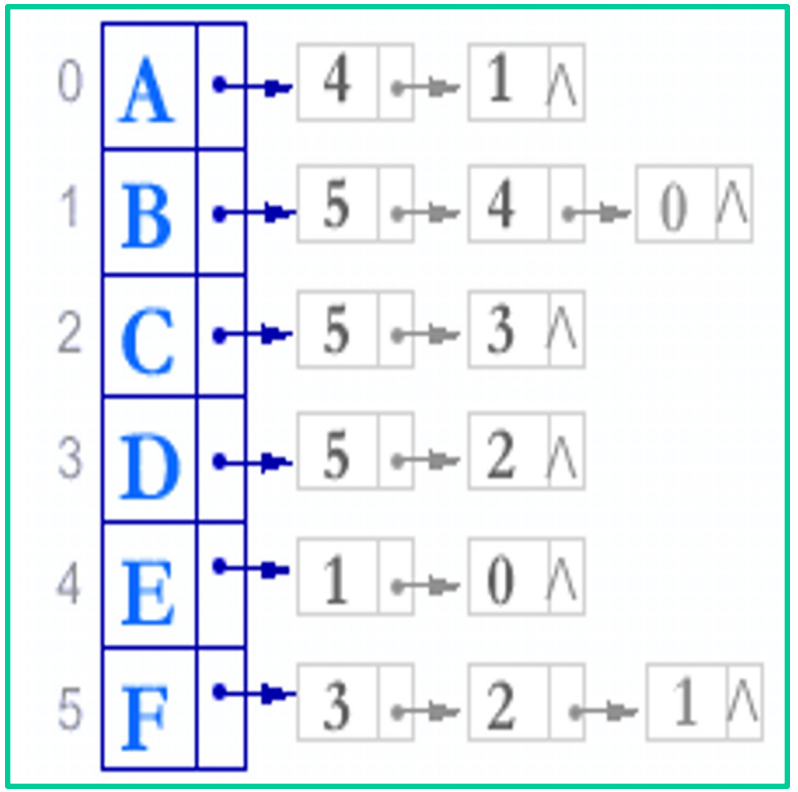
- 表头结点：数据域、链域；

data	firstarc

图的存储结构



无向图G2



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

- 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

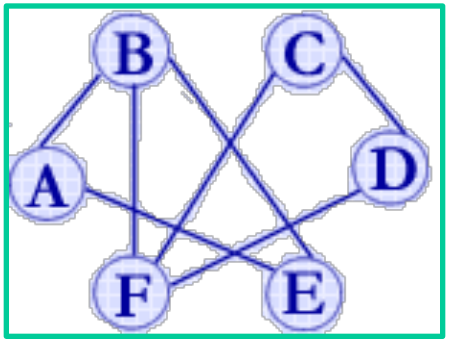
- 例如，按算法7.3生成的无向图G2的邻接表如右下所示：
  - 依次输入的数据为：

    6 7 UDG

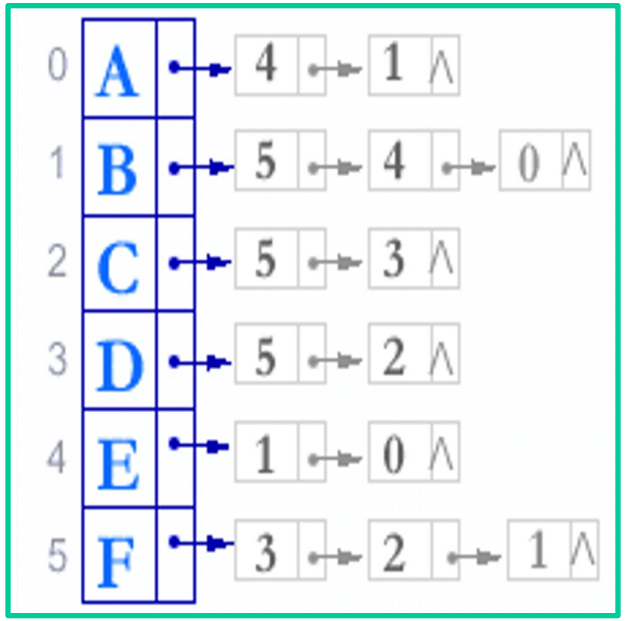
    A B C D E F

    AB AE BE BF CD CF DF

  - 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。

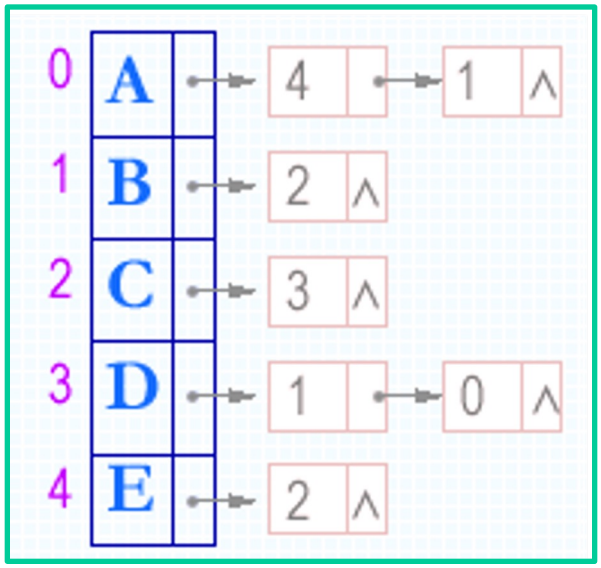


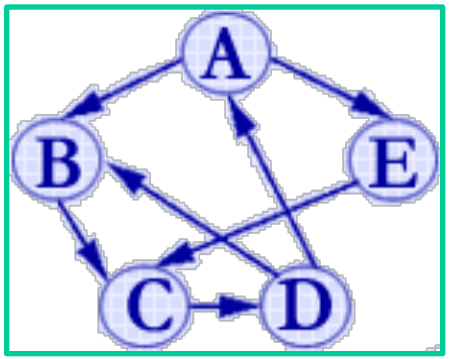
无向图G2



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：



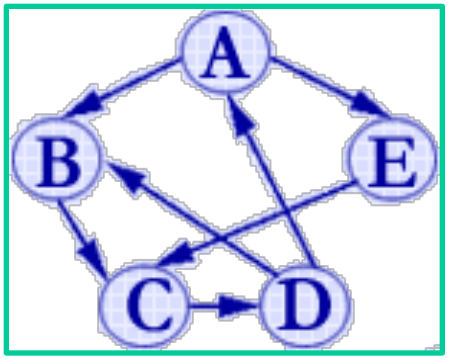


有向图G1

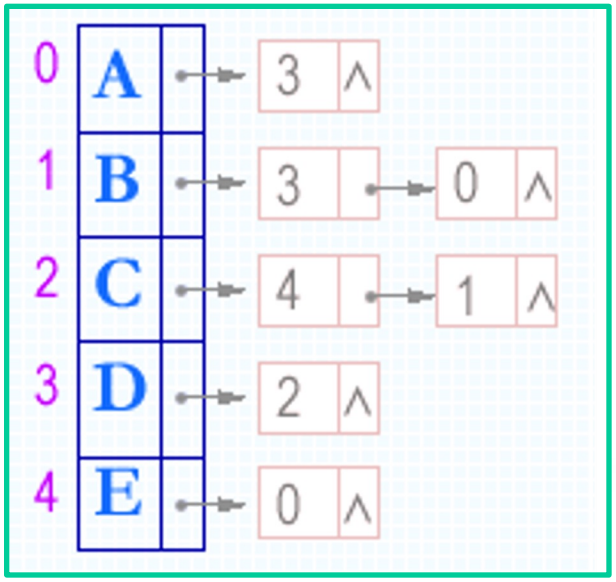
- 有向图的邻接表中链接在每个顶点结点中的都是以该顶点为弧尾的弧，每个单链表中的弧结点的个数恰为弧尾顶点的出度，每一条弧在邻接表中只出现一次。虽然在邻接表中也能找到所有以某个顶点为弧头的弧，但必须查询整个邻接表。

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



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

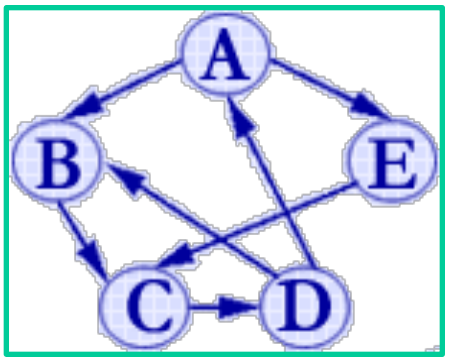
tailvex	headvex	hlink	tlink	Info

➔ 顶点结点

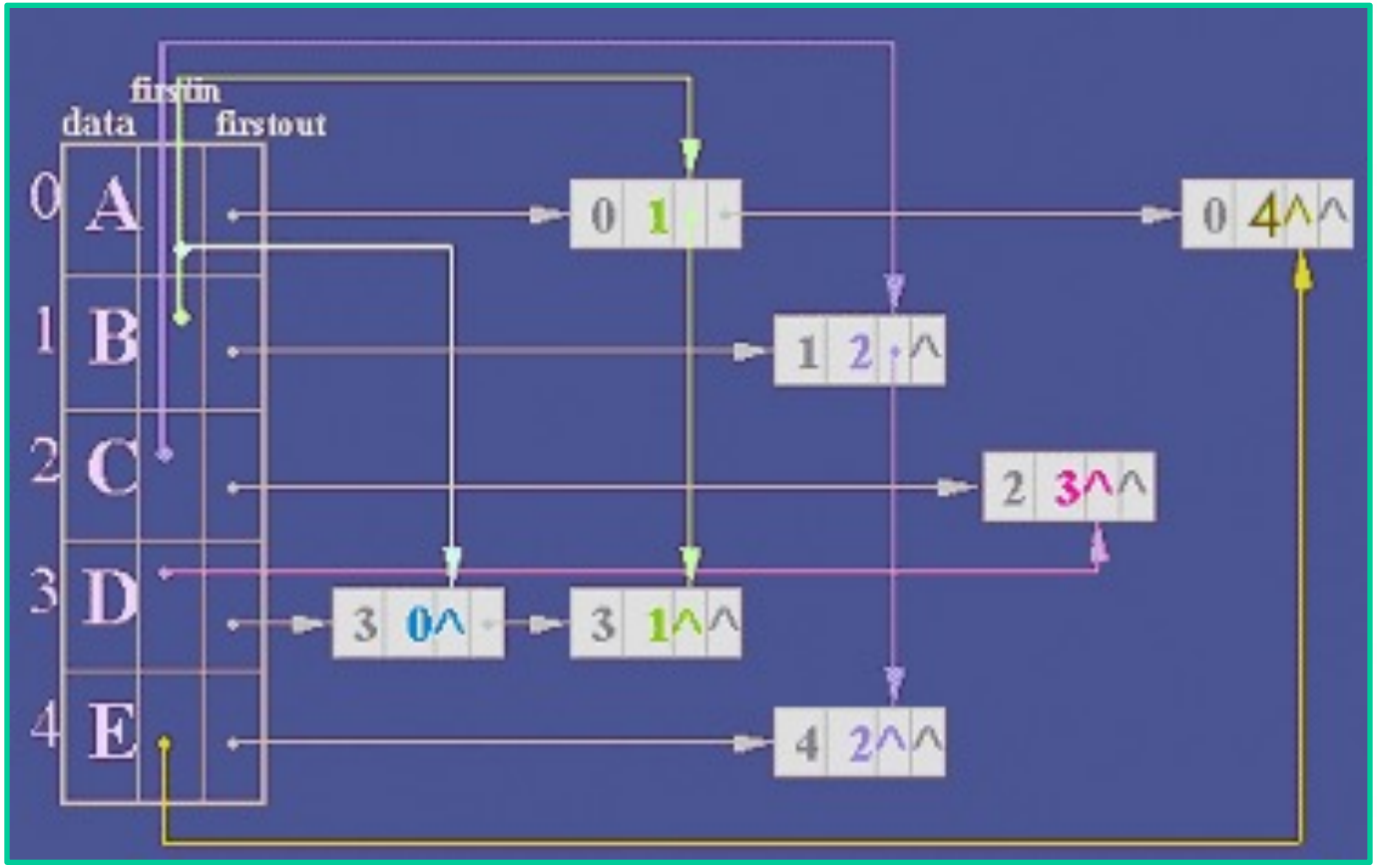
data	firstin	firstout

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



有向图G1



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

```
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**

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

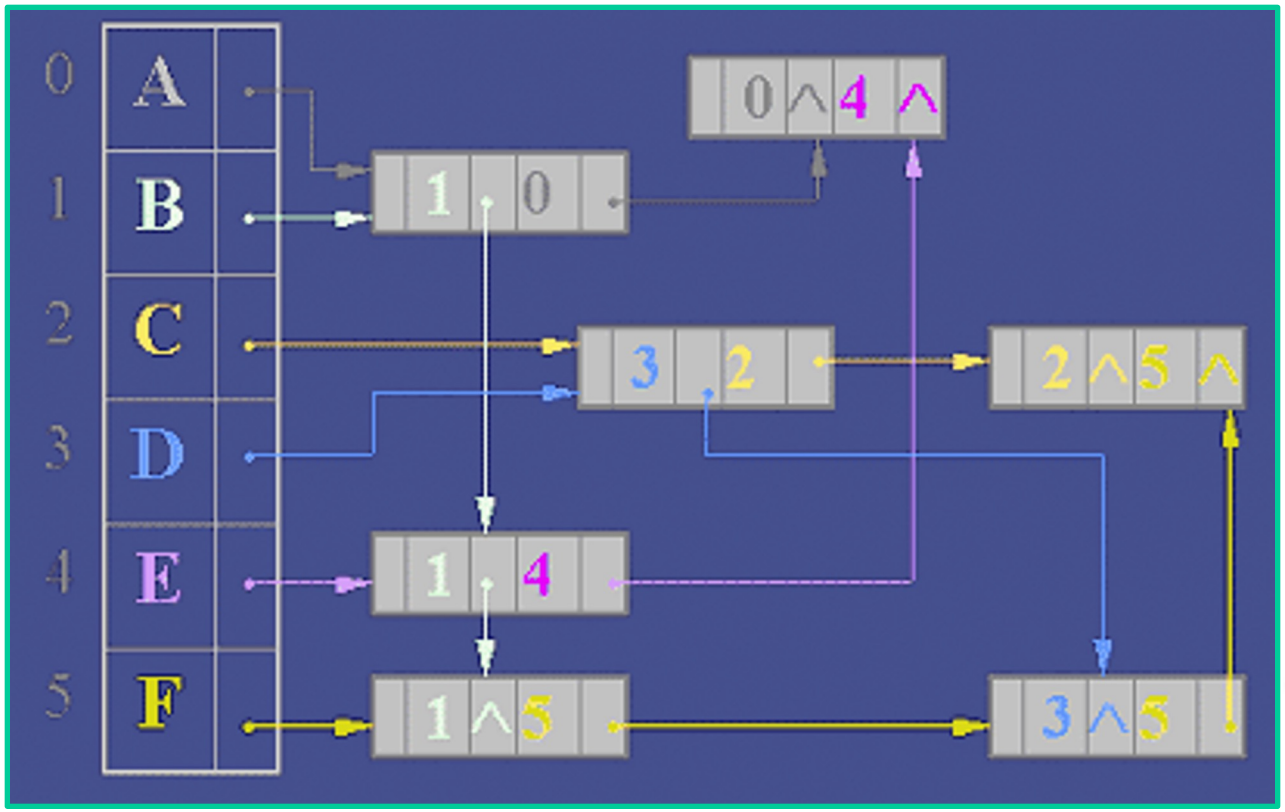
mark	ivex	ilink	jvex	jlink	info

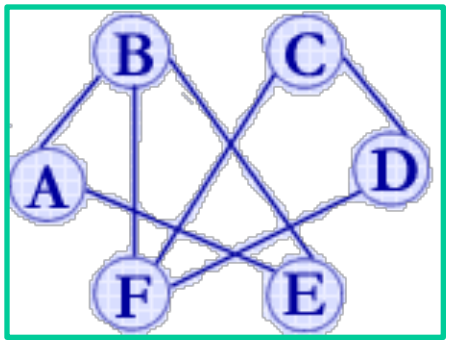
  - 顶点结点

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





无向图G2

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 emnu {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

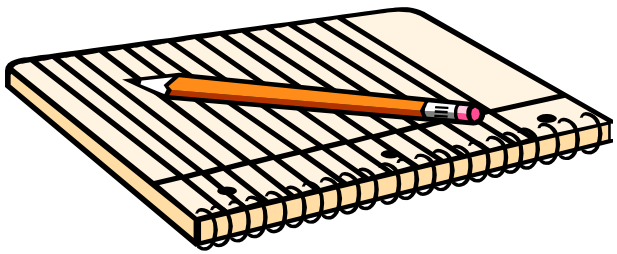
- 算法7.5

```
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( &vi , &vj);                        // 输入一条边的两个顶点
      i = LocateVex(G, vi ); j = LocateVex(G, vj);
                                // 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==AN)
                  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

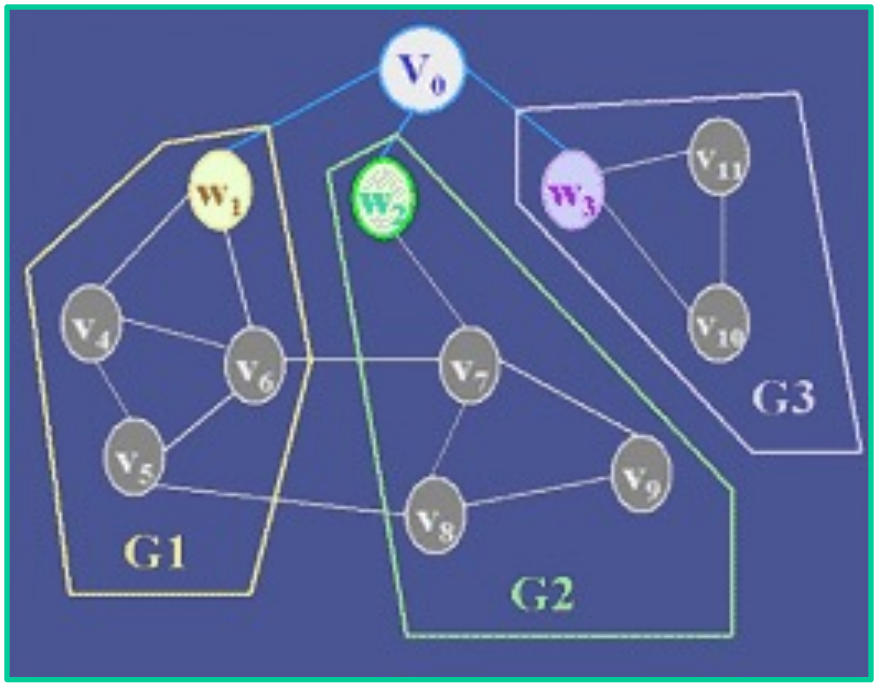
- 与二叉树和树的遍历相同，**图的“遍历”是对图中的每个顶点都进行一次访问且仅进行一次访问**。但由于图结构较树和二叉树更为复杂，图中任意两个顶点之间都可能存在一条弧或边，反之也可能存在某个顶点和其它顶点之间都不存在弧或边。
- 因此对图的遍历而言，除了要确定一条搜索路径之外，还要解决两个问题：
  - (1) **如何确保每个顶点都被访问到**；
  - (2) **如何确保每个顶点只被访问一次**。

图的遍历

● 一、深度优先搜索遍历图

➔ 1、连通图的遍历

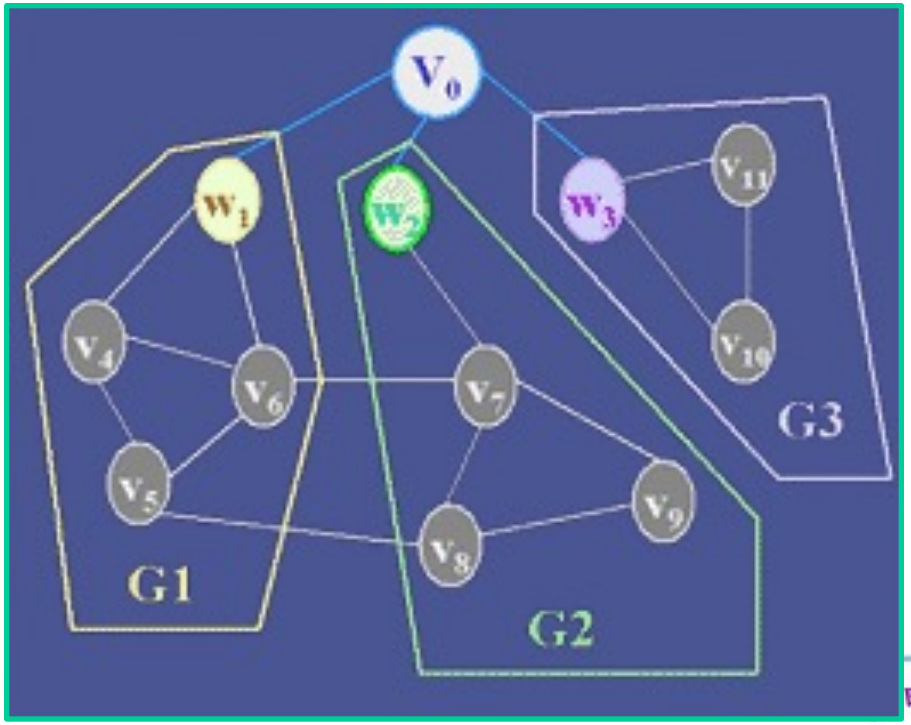
从某个顶点V_0出发**深度优先搜索遍历连通图**的定义为：首先访问该顶点，然后依次从V_0的各个**未被访问过的邻接点**出发进行深度优先搜索遍历。



注意上述定义中对邻接点的**“未被访问过的”**修饰词语。如左图，假设从图中顶点 V0 出发进行深度优先搜索遍历，顶点V0 有三个邻接点w1、w2 和w3 ，它们分别为 G 的三个连通子图 G1、G2 和 G3 中的一个顶点，由于G1和G2连通，从w1出发的遍历过程中已经访问了顶点w2，由此不能再从w2出发进行遍历了。

图的遍历

- 为确保每个顶点在遍历过程中只被访问一次，建立一个“访问标志visited[i]”，遍历前，设为 “FALSE”，访问后，则令 visited[i] 为"TRUE"。
- 容易看出图的深度优先搜索遍历的过程类似于树的先根遍历。可将V_0看作是树的根结点，w_1、w_2 和w_3可看成是它的三个“子树根结点”，如果 G1、G2 和 G3 分别为三个连通子图且相互之间不连通，则在访问V_0之后可依次从w_1、w_2和w_3出发对图进行深度优先搜索遍历。



图的遍历

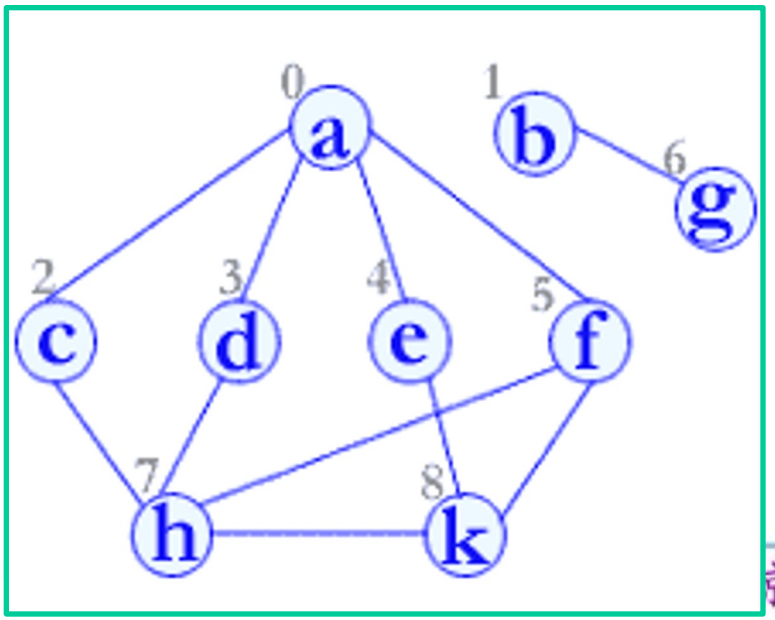
➔ 2、非连通图的遍历

对于非连通图，如何确保每个顶点都能被访问到？

只能对图中所有顶点巡查一遍“挨个检查”，即从第一个顶点起，如果该顶点未被访问，则从该顶点出发进行深度优先遍历，否则接着检查下一顶点，直至所有顶点都被访问到为止。

例如，对下列非连通图G3进行深度优先搜索遍历，得到顶点的访问序列为：

a→c→h→d→k→f→e→b→g　　演示



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
```

图的遍历

- 分析上述算法，在遍历图时，对图中每个顶点至多调用DFS一次，因为一旦某个顶点被标识成已被访问，就不再从它出发进行搜索，而DFS过程中耗费的时间主要在于找邻接点的过程，其耗费的时间则取决于所采用的数据结构。
- 当用二维数组表示邻接矩阵作图的存储结构时，查找每个顶点的邻接点所需的时间为$O(n^2)$，其中n为图中顶点数。
- 而当以邻接表作图的存储结构时，找邻接点所需时间为$O(e)$，其中e为无向图中边的数或有向图中弧的数。由此，当以邻接表作存储结构时，深度优先搜索遍历图的时间复杂度为$O(n+e)$。

广度优先搜索

●二、广度优先搜索遍历图

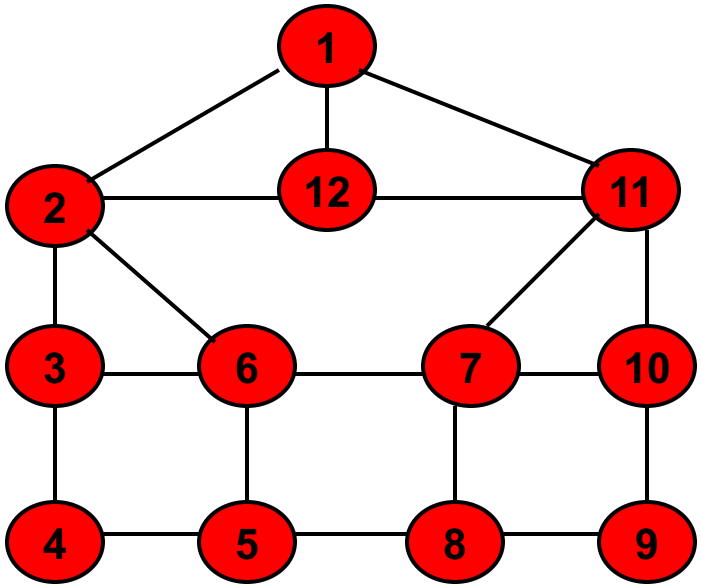
广度优先搜索(Breadth_First Search)**的基本思想**是：

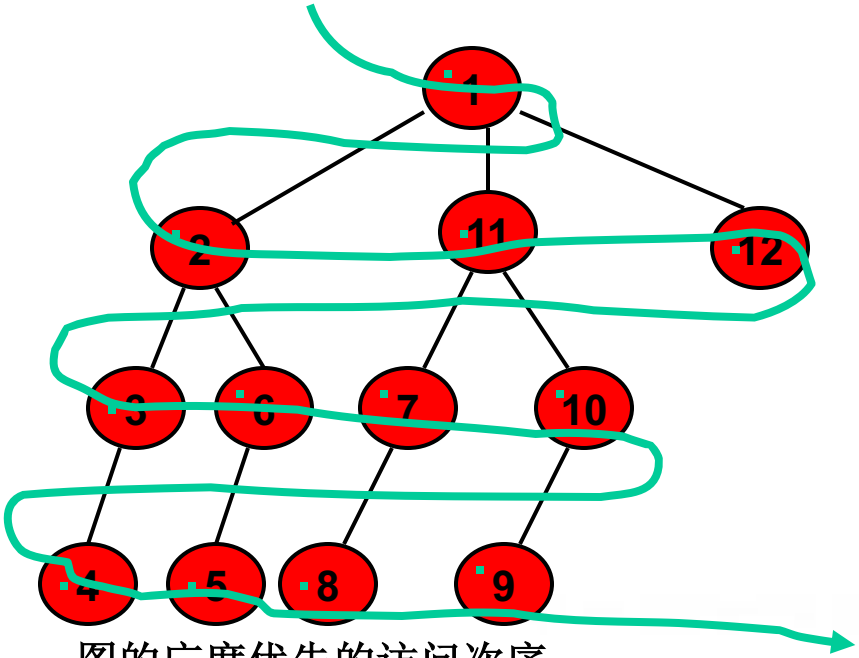
- 从图中某个**顶点 v 出发**
- 在访问了 v 之后**依次访问 v 的**各个未曾访问过的**邻接点**
- 然后分别**从这些邻接点出发依次访问**它们的**邻接点**
- 并使得**“先被访问的顶点的邻接点”**先于**“后被访问的顶点的邻接点”**进行访问，直至图中所有已被访问的顶点的邻接点都被访问到
- 如若此时图中**尚有顶点未被访问**，则需另选一个未曾被访问过的顶点作为新的起始点，重复上述过程，直至图中所有顶点都被访问到为止。

●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。

广度优先搜索

● 无向图的实例：为了说明问题，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。





图的广度优先的访问次序：

1、2、11、12、3、6、7、10、4、5、8、9

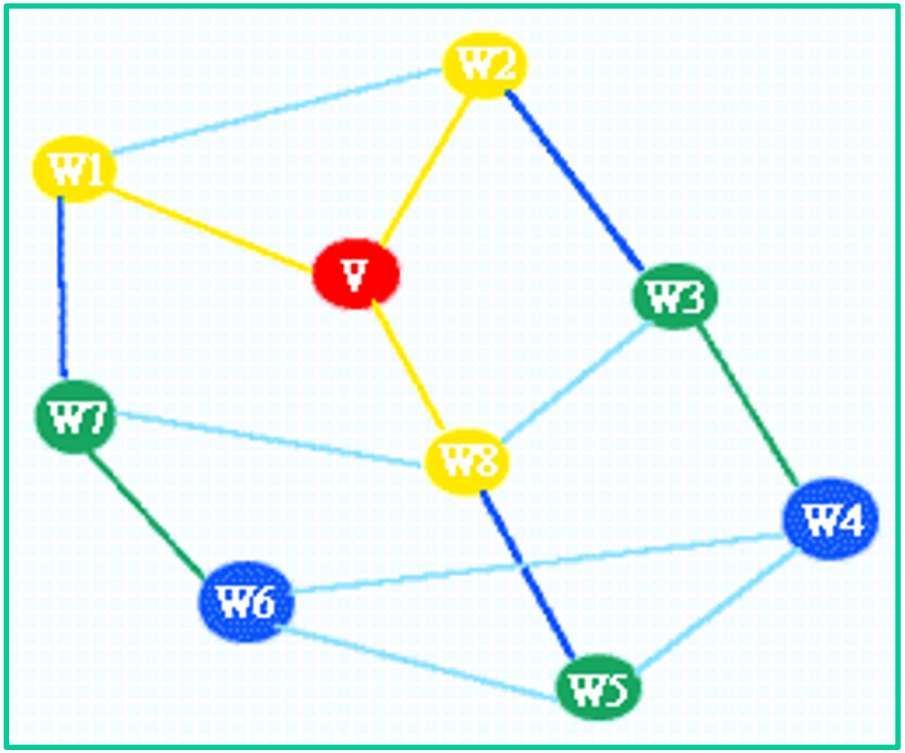
适用的数据结构：队列

时间复杂性：
邻接矩阵：O（n^2）
邻接表： O（n + e）
n：结点数 e：边数

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：

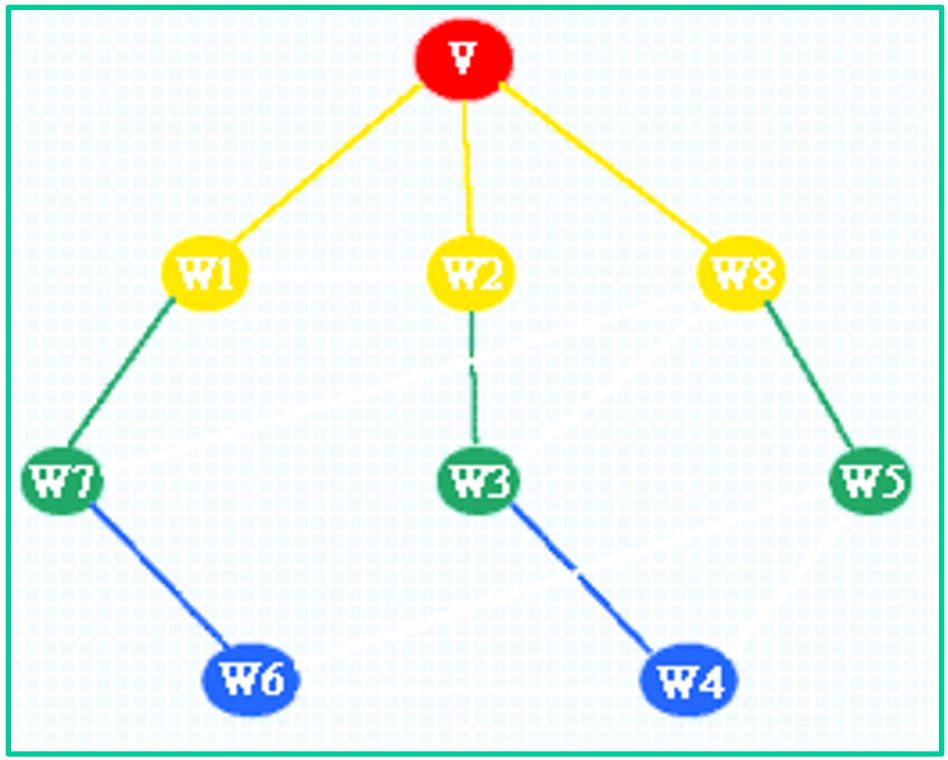
v→ w_1 → w_2 → w_8 → w_7 → w_3 → w_5 → w_6 → w_4 演示



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

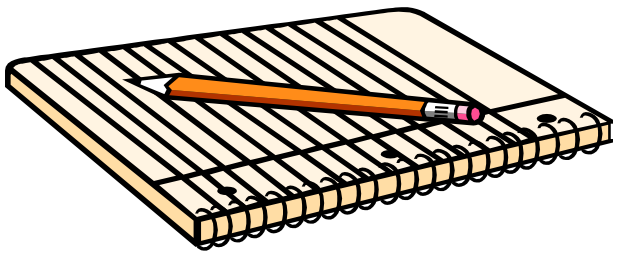
- 算法7.8

```
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visited。
      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



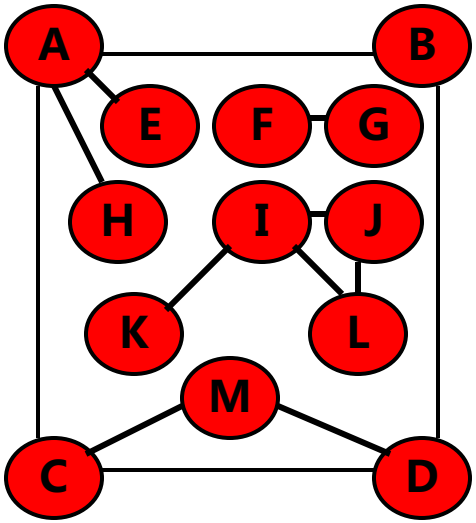
- 7.1 图的定义和术语
- 7.2 图的存储结构
- 7.3 图的遍历
- 7.4 图的连通性问题（一）
- 7.5 有向无环图及其应用
- 7.6 最短路径

图的连通性问题

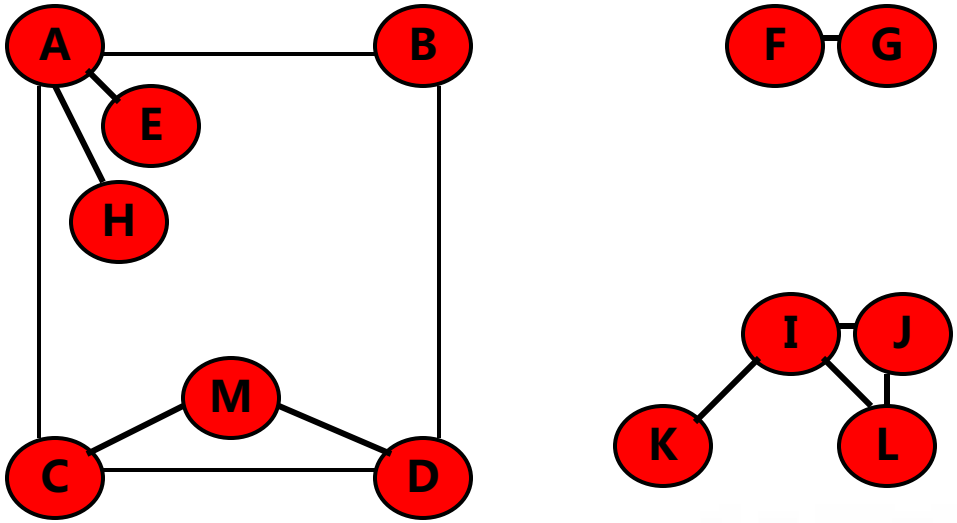
- 图遍历是图的基本操作，也是一些图的应用问题求解算法的基础，以此为框架可以派生出许多应用算法。例如在**遍历过程中可以求得非连通图的连通分量或强连通分量**，可以利用**深度优先搜索遍历求得图中两个顶点之间一条简单路径**，或利用**广度优先搜索遍历求得两个顶点之间的最短路径**等等。在此仅以求**生成树或生成森林**为例说明图遍历的应用。

无向图的连通分量和生成树

- **连通**：顶点v至v′ 之间有路径存在
- **连通图**：无向图图 G 的任意两点之间都是连通的，则称G是连通图。
- **连通分量**：极大连通子图



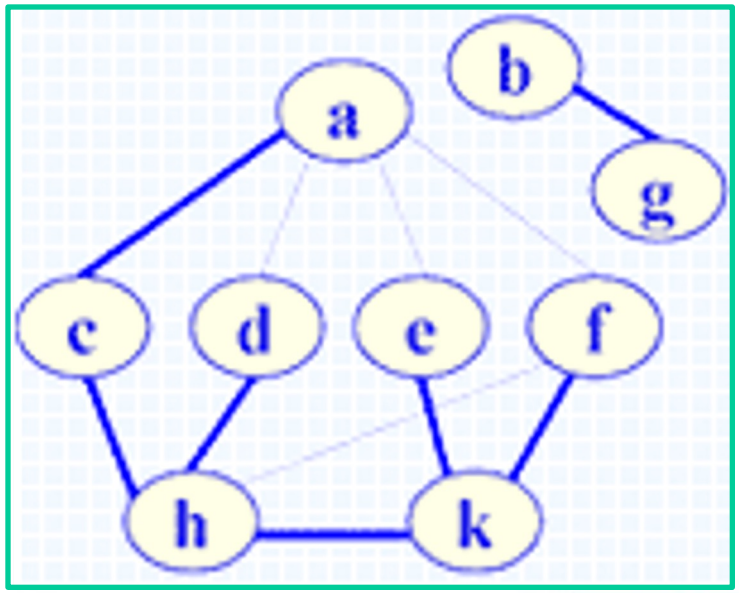
无向图G



无向图G的三个连通分量

无向图的连通分量和生成树

- 假设图 G=(V,E)，其中V是顶点的集合，E是边(或弧)的集合。则在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，如上图中由粗线表示的边，可称这些边为**“树边”**；其余的边均为集合 B(E) 中的边，可称这些边为**“回边”**。



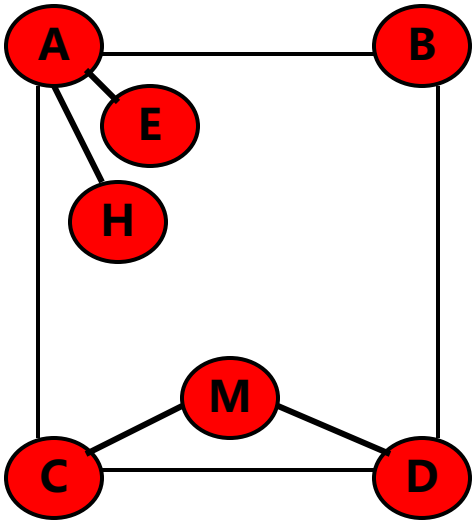
无向图的连通分量和生成树:续

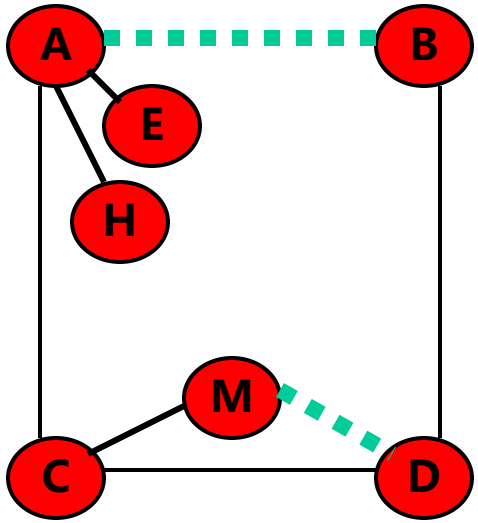
●**生成树**：极小连通子图。

◆包含图的所有 n 个结点，但只含图的 n-1 条边。

◆在生成树中添加一条边之后，必定会形成回路或环。因为在生成树的任意两点之间，本来就是连通的，添加一条边之后，形成了这两点之间的第二条通路。

无向图G



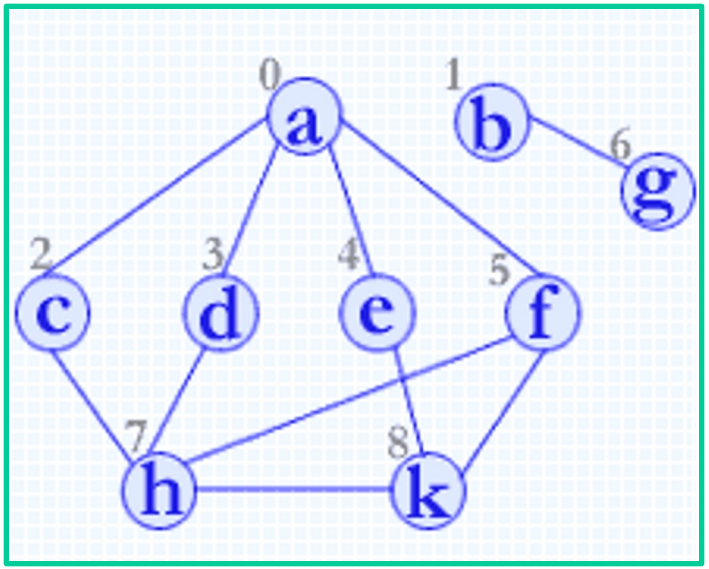


无向图G的生成树

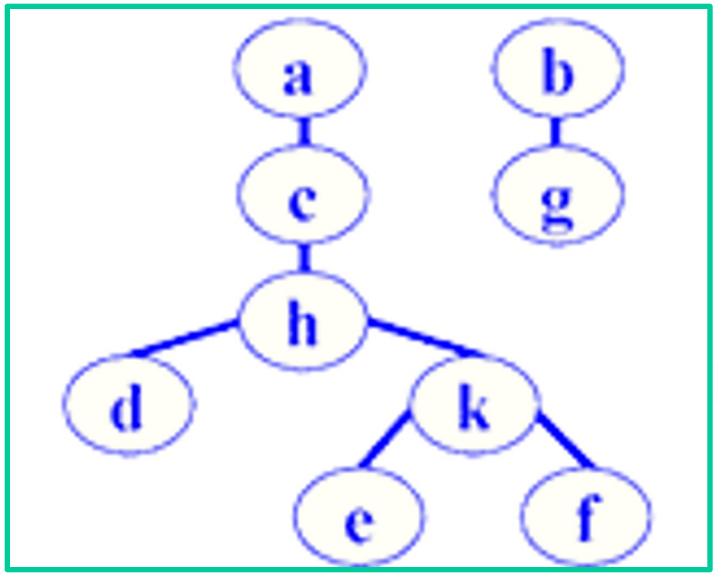
●**生成方法**：进行深度为主的遍历或广度为主的遍历，得到**深度优先生成树**或**广度优先生成树**。

图的连通性问题

- 例如，在对图 G3 进行深度优先搜索遍历过程中，从顶点 h 出发通过边找到邻接点 d，由于顶点 d 未被访问，则从 d 出发进行深度优先搜索，在对 d 进行访问之后，通过边找到邻接点 a，由于 a 已被访问，不再需要进行从 a 出发的遍历。由此 (h,d) 为树边，而 (d,a) 为回边。由深度优先搜索遍历过程中所有的树边 (T(E)) 和 n 个顶点即构成深度优先森林。例如，图 G3 的深度优先森林如右下所示。



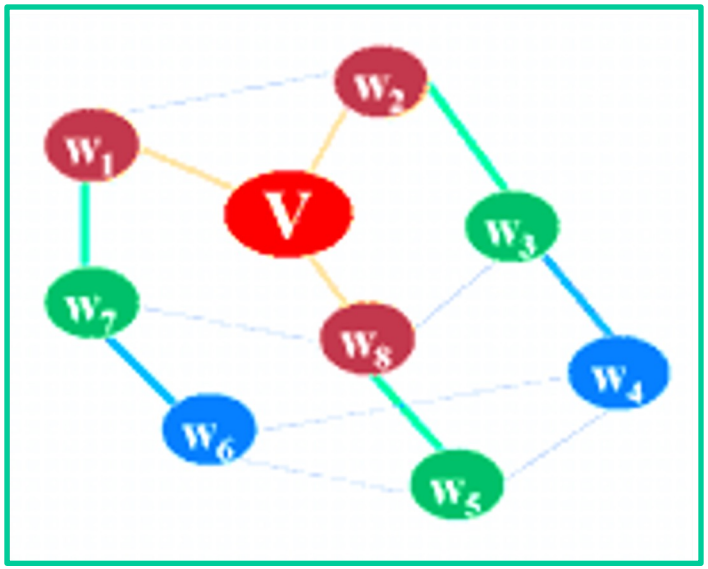
图G3



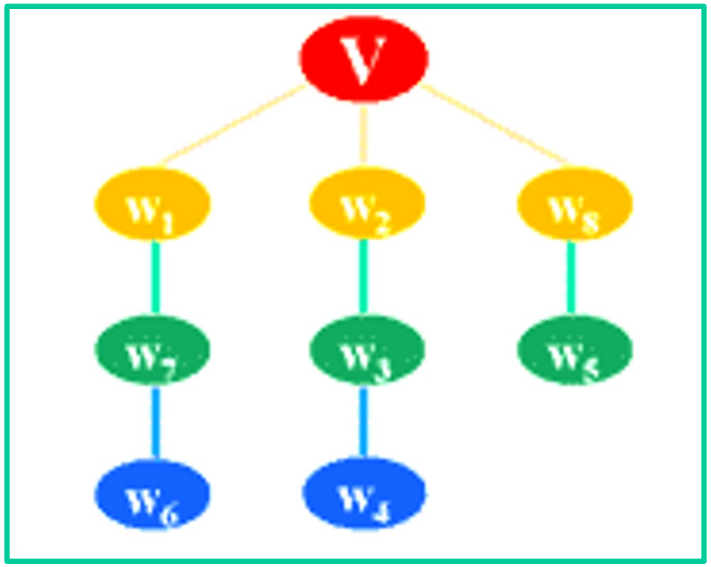
图G3的深度优先森林

图的连通性问题

- 同样，在广度优先搜索遍历的过程中，也将边集E分成树边 T(E) 和回边 B(E) 两个子集。例如，左下图所示为在对图 G4 进行广度优先搜索遍历过程中形成的树边(以粗线表示)和回边(以细线表示)。由广度优先搜索遍历过程中所有的树边 (T(E)) 和 n 个顶点即构成广度优先森林。例如，图G4的广度优先森林如右下所示。



图G4

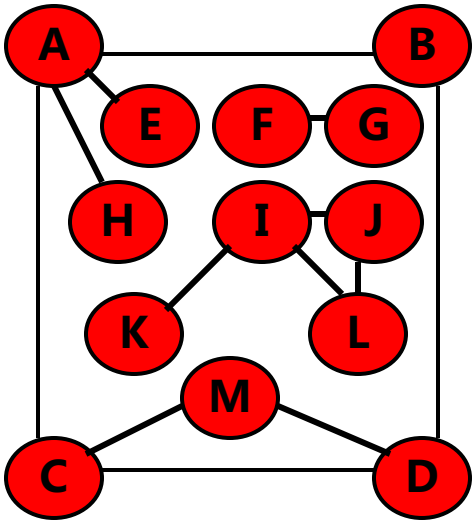


图G4的广度优先森林

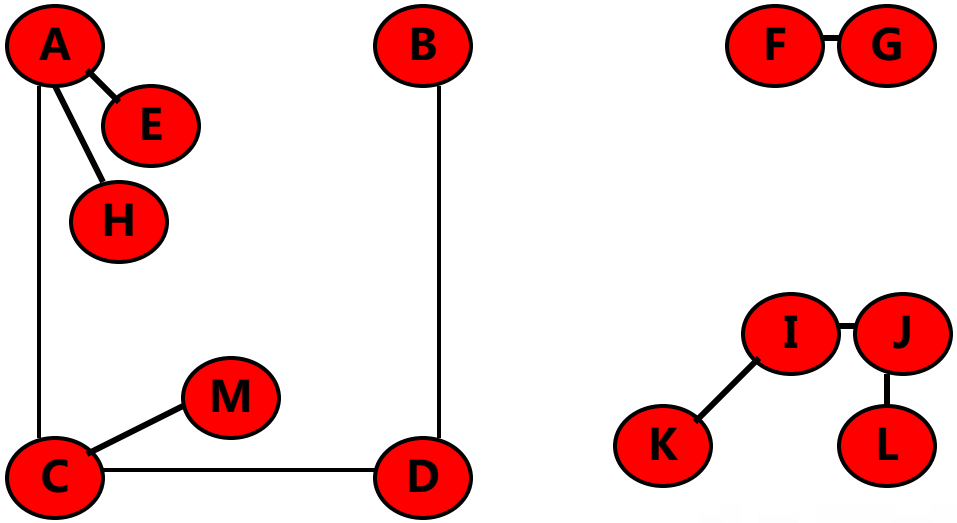
无向图的连通分量和生成树:续

●**生成森林**：在进行深度为主的遍历或广度为主的遍历时，对于非连通图，将得到多棵深度优先生成树或广度优先生成树。称之为生成森林。

无向图G



无向图G的生成森林



深度优先森林的算法

●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。