数据结构

主讲：项若曦　助教：申智铭、黄毅
rxxiang@blcu.edu.cn
主楼南329

回顾

- **二叉排序树**
  - 概念、性质、存储
  - 操作：遍历、查找、插入、创建、删除
  - 性能分析：平均查找长度~

作业难点梳理

二叉排序树作业：
1. 难点梳理：
a) 回顾创建树的方法（增广的先序序列、双序列创建、表达式树、二叉排序树）
b) 正确认识插入创建树和查找之间的关系。都可以递归写，也可以迭代写。
c) 编程难点：递归、递归函数的debug~大家应该先完成非递归、再完成递归。提醒大家，写好的递归的代码，一定要**多debug**，观察系统调用栈的工作情况。同时，反过来写出**递归的形式化表达式**。
d) 遗留问题：文件引用关系、数据类型问题、输入形式

2. 平均查找长度：定义是“关键字”的比较次数。作业中体现的比较少，但也是需要大家掌握的重难点。

HanoiTower.cpp

test

```
1  #include <stdlib.h>
2  #include<stdio.h>
3  #include "string.h"
4  void Hanoi(int n, char A, char B, char C) {
5      //用A、B、C代表三个柱子，算法模拟汉诺塔问题
6      if (n == 1)
7          printf(" move %c  to  %c \n", A, C);
8      else { 已用时间<=215ms
9          Hanoi(n - 1, A, C, B);
10         printf(" move %c  to  %c \n", A, C);
11         Hanoi(n - 1, B, A, C);
12     }
13 }
14 void main() {
15     int n = 3;
16     char A = 'A', B = 'B', C = 'C';
17     Hanoi(n, A, B, C);
18 }
```

100 %

调用堆栈

名称
test.exe!Hanoi(int n=1, char A='A', char B='B', char C='C') 行 8
test.exe!Hanoi(int n=2, char A='A', char B='C', char C='B') 行 10
test.exe!Hanoi(int n=3, char A='A', char B='B', char C='C') 行 10
test.exe!main() 行 18
test.exe!invoke_main() 行 65
test.exe!__scrt_common_main_seh() 行 259
test.exe!__scrt_common_main() 行 302
test.exe!mainCRTStartup() 行 17
kernel32.dll!0000000076c859cd()
ntdll.dll!0000000076dba561()

调用堆栈 | 断点 | 异常设置 | 输出

“递归工作栈”

“工作记录”

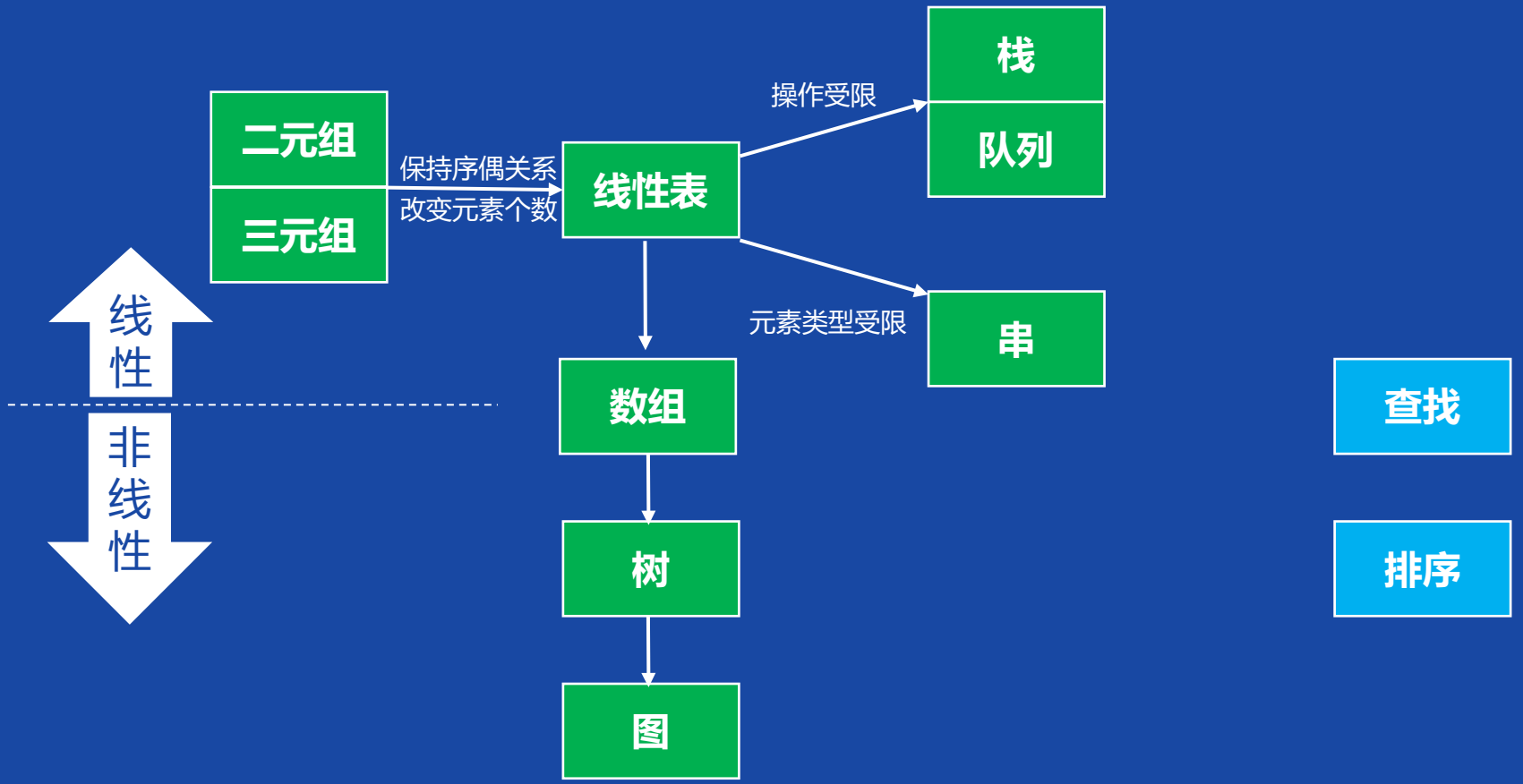
实在参数
局部变量
返回地址

“层次”

主函数	0层
第1次调用	1层
…	…
第 i 次调用	i 层

回顾

- **Huffman树**
  - 概念：带权路径长度、前缀编码、最优二叉树。
  - 创建
  - 编码
  - 解码



- 程序 = 数据结构 + 算法

本节内容

第六章 图

- 6.1 图的定义和术语
- 6.2 案例引入
- 6.3-6.4 图的类型定义和存储结构
- 6.5 图的遍历（DFS、BFS）
- 6.6 图的应用（MST、SP、拓扑排序、AOV/AOE）

重点：图在邻接矩阵和邻接表上的遍历算法(包括DFS, BFS)。

难点：基于图的遍历算法的应用、图的应用

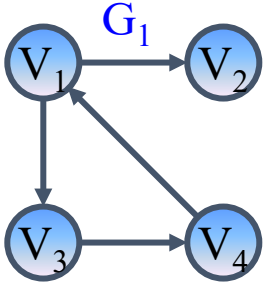
本节内容

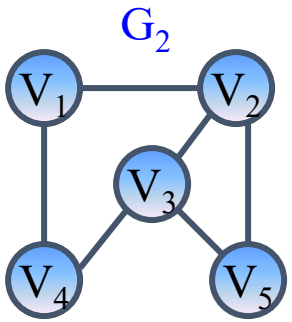
- 图的定义和术语

图的定义和术语

图的定义

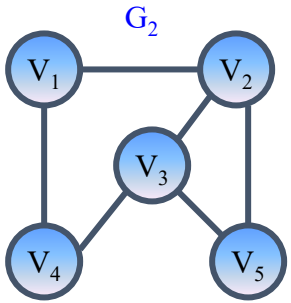
- 图是由顶点集合(vertex)及顶点之间的关系集合组成的一种数据结构。G = (V, {E}) V-顶点集，E-关系的集合
- 顶点之间的关系是任意的(m : n)
- 顶点之间的关系是否有方向性：有向图、无向图
- 无向图（UG）：边(v,w) ∈ E (v,w ∈V)，v与w互为邻接点，边(v, w)依附于顶点v和w，或者说边(v, w)和顶点v, w相关联。
  - 顶点v 的度是和v相关联的边的数目,记为TD(v) 。
- 有向图（DG）：弧<v,w> ∈ E (v,w ∈V)，w为弧头, v为弧尾; 顶点v 邻接到顶点w，顶点w 邻接自顶点v，弧< v, w >和顶点v、w相关联。
  - 顶点v 的入度是以v 为弧头的弧的数目,记为ID(v);
  - 顶点v 的出度是以v为弧尾的弧的数目,记为OD(v);
  - 顶点v 的度是TD(v) = ID(v) + OD(v)。

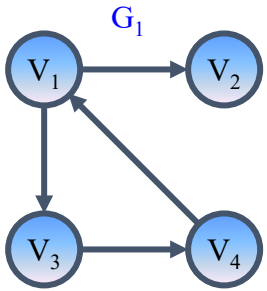


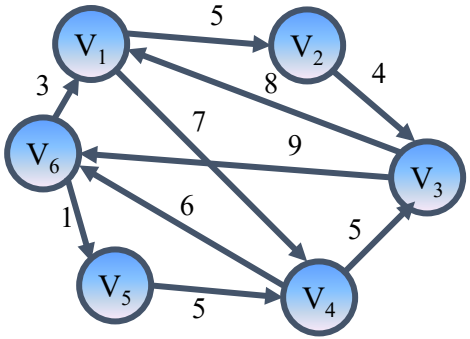


图的分类

- **根据顶点之间的关系是否有方向性、有权值，可分四类：**
  - 无向图(Undirected Graph, UG)
  - 有向图(Directed Graph, DG)：每条边是有方向的，此时边也称为弧（Arc）
  - (无向、有向)网：边（弧）上带权值(Weighted)

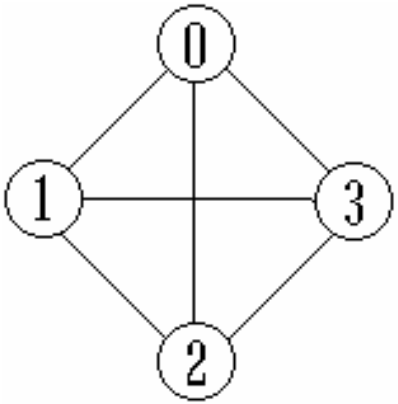






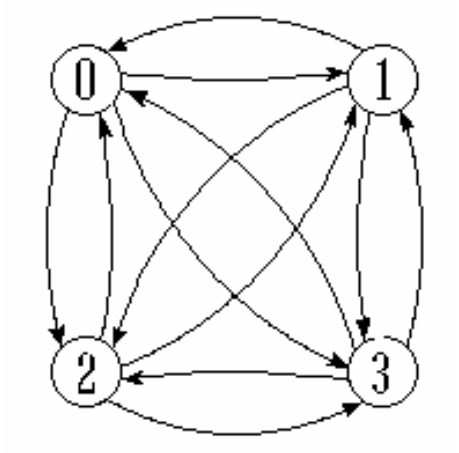
特殊的图：完全图

- **完全图**
  - 任意两个点都有一条边相连



无向完全图

$n(n-1)/2$ 条边



有向完全图

$n(n-1)$ 条边（弧）

图的分类

- **根据边的稠稀程度：**
  - 稀疏图(Sparse)：有很少边或弧的图。
  - 稠密图(Dense)：有较多边或弧的图。

图的定义和术语(续)

边/弧数目

- 用n 表示图中顶点数目，用e 表示图中边或弧的数目
- 无向图：$0 \le e \le \frac{1}{2} n(n-1)$　　完全图 $e = \frac{1}{2} n(n-1)$
- 有向图：$0 \le e \le n(n-1)$　　有向完全图 $e = n(n-1)$
- 稀疏图：　$e < n\log n$　　稠密图
- 权，网　有向图、有向网、无向图、无向网
- 思考：若无向图中有21条边，则：
  1) 保持该图不连通至少应具有多少个顶点?
  2) 保持该图连通至少有多少个顶点，至多有多少个顶点?

1) m个顶点形成完全图，另一个顶点与这m个顶点不连通
   $\frac{1}{2} m(m-1)=21$ ➔ $m=7$ ➔ $n=m+1=8$
2) 至少有 7 个顶点(完全图)，至多有22个顶点(生成树)

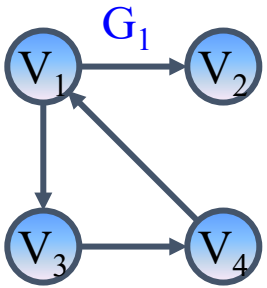
图的相关概念

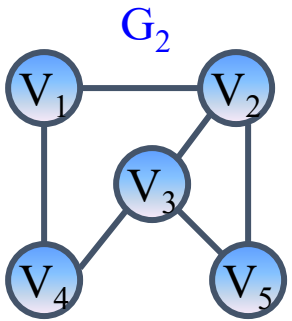
- **邻接（Adjacent）：**
  - 有边/弧相连的两个顶点之间的关系。
    - 存在(v_i, v_j)，则称v_i和v_j互为邻接点；
    - 存在$<v_i, v_j>$，则称v_i邻接到v_j， v_j邻接于v_i
- **关联（依附，Incident）：**
  - 边/弧与顶点之间的关系。
    - 存在(v_i, v_j)/ $<v_i, v_j>$， 则称该边/弧关联于v_i和v_j

顶点的度（Degree）

顶点的度

- 无向图（UG）：
  - 顶点v 的度是和v相关联的边的数目,记为TD(v) 。
- 有向图（DG）
  - 顶点v 的入度是以v 为弧头的弧的数目,记为ID(v);
  - 顶点v 的出度是以v为弧尾的弧的数目,记为OD(v);
  - 顶点v 的度是TD(v) = ID(v) + OD(v)。

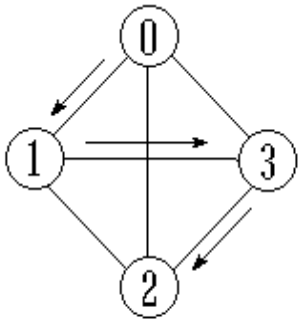




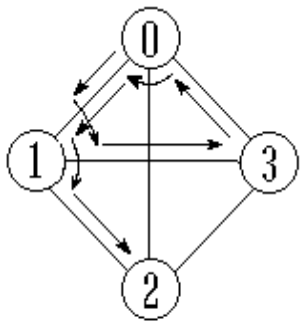
路径

路径、简单路径、回路（环）等

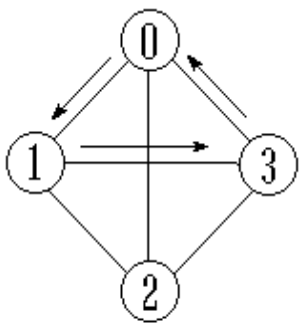
- 路径：接续的边构成的顶点序列。
- 路径长度：路径上边或弧的数目/权值之和。
- 回路(环)：第一个顶点和最后一个顶点相同的路径。
- 简单路径：除路径起点和终点可以相同外，其余顶点均不相同的路径。
- 简单回路(简单环)：除路径起点和终点相同外，其余顶点均不相同的路径。



（a）简单路径



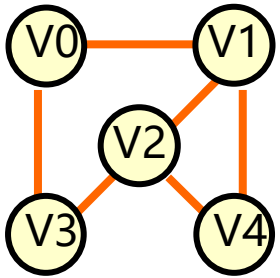
（b）非简单路径



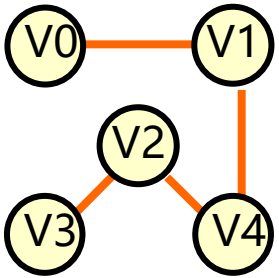
（c） 回路

子图

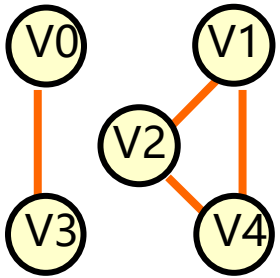
- 子图—对于G=(V, {E})，若图G'满足：1) V'⊆ V; 2) E'⊆ E; 3) G'=(V', {E'})，则称G'是G的子图。
- 例:(b)、(c) 是 (a) 的子图



(a)



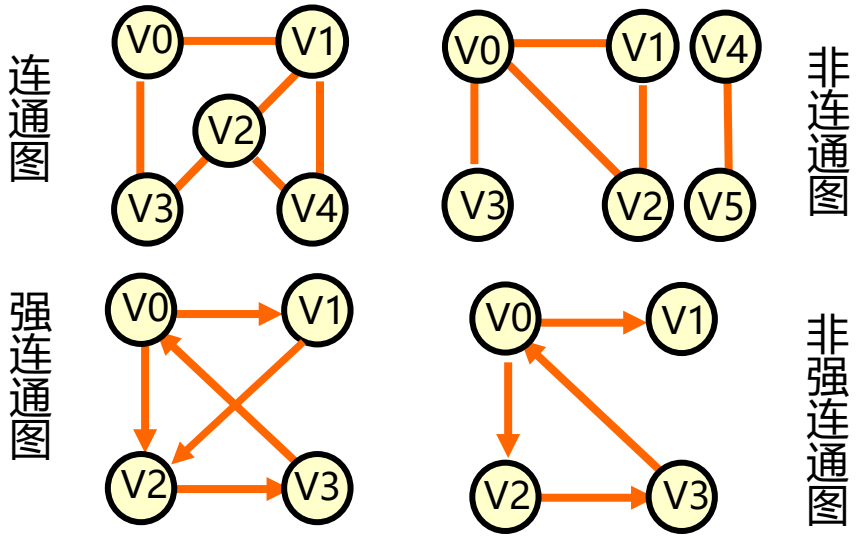
(b)



(c)

连通、（强）连通图

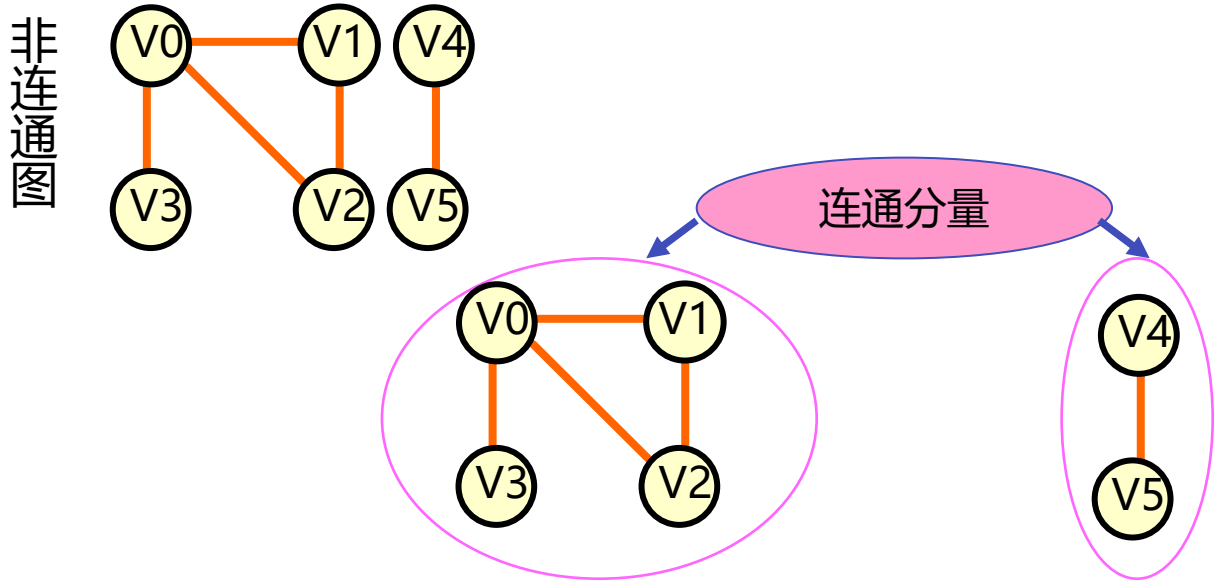
- 在无（有）向图G=(V, {E})中，若对任何两个顶点 v、u 都存在从v 到 u 的路径，则称G是连通图（强连通图）。



（强）连通分量

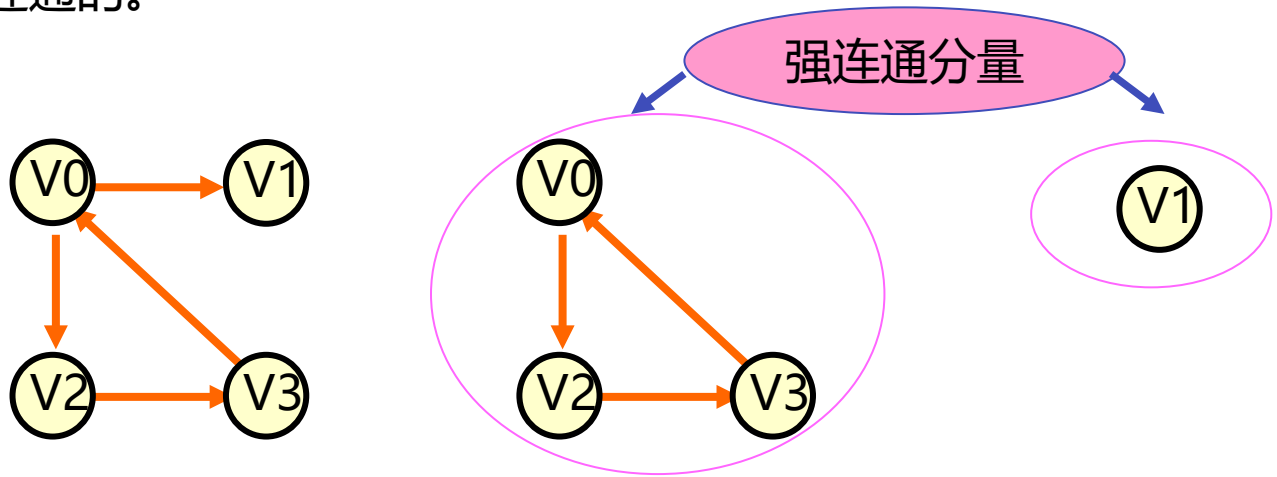
无向图连通分量

- 无向图G 的极大连通子图称为G的连通分量。极大连通子图意思是：该子图是 G 连通子图，将G 的任何不在该子图中的顶点加入，子图不再连通。



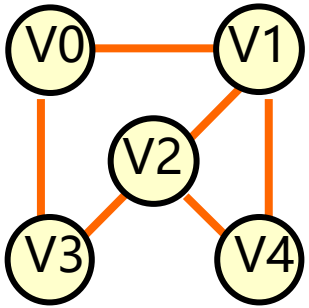
有向图强连通分量

- 有向图G 的极大强连通子图称为G的强连通分量。极大强连通子图意思是：该子图是G的强连通子图，将D的任何不在该子图中的顶点加入，子图不再是强连通的。

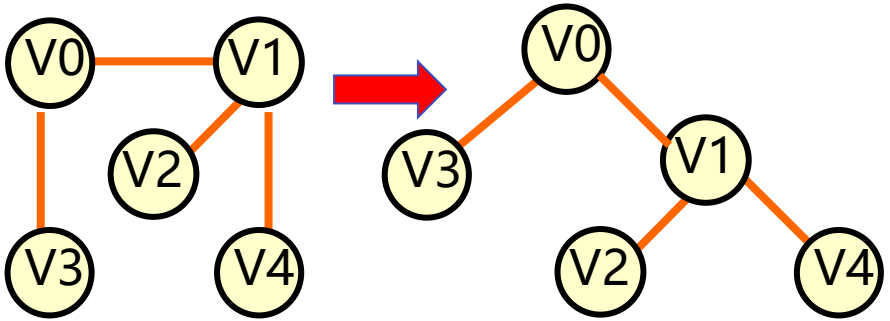


极小连通子图、生成树（Spanning Tree）、生成森林

- 极小连通子图：该子图是G 的连通子图，在该子图中删除任何一条边，子图不再连通。
- 生成树：包含无向图G 所有顶点的极小连通子图。
- 生成森林：对非连通图，由各个连通分量的生成树的集合。



连通图 G1



G1的生成树

图的定义和术语(续)

边/弧数目

- 用n 表示图中顶点数目，用e 表示图中边或弧的数目
- 无向图：$0 \le e \le \frac{1}{2} n(n-1)$　　完全图 $e = \frac{1}{2} n(n-1)$
- 有向图：$0 \le e \le n(n-1)$　　有向完全图 $e = n(n-1)$
- 稀疏图：　$e < n\log n$　　稠密图
- 权，网　有向图、有向网、无向图、无向网
- 思考：若无向图中有21条边，则：
  1) 保持该图不连通至少应具有多少个顶点?
  2) 保持该图连通至少有多少个顶点，至多有多少个顶点?

1) m个顶点形成完全图，另一个顶点与这m个顶点不连通
   $\frac{1}{2} m(m-1)=21$ ➔ $m=7$ ➔ $n=m+1=8$
2) 至少有 7 个顶点(完全图)，至多有22个顶点(生成树)

本节内容

- 图的类型定义和存储结构

图的定义-ADT Graph

ADT Graph

- 查找：LocateVex(G, u)
  GetVex(G, v)
  FirstAdjVex(G, v) // 顶点v的第一个邻点
  NextAdjVex(G, v, w) // 顶点v的在w之后的下一个邻点
- 插入：InsertVex(&G, v)
  InsertArc(&G, v, w) //插入边（v，w）
- 删除：DeleteVex(&G, v)
  DeleteArc(&G, v, w)
- 遍历：DFSTraverse(G, v, Visit()) // 深(广)度优先遍历
  BFSTraverse(G, v, Visit()) 图G上所有的顶点

图的存储表示

已学过的数据结构存储表示分析

- 线性表：1对1的关系，通过结点之间的存储位置反映结点之间的逻辑关系
  - 逻辑关系：线性结构
  - 存储结构：顺序表示、链式表示
- 二叉树：1对2的关系，通过非线性结构的线性化来反映结点之间的逻辑关系
  - 逻辑关系：树型结构
  - 存储结构：顺序表示。（结点i的左孩子为2i，右孩子为2i+1）
    二叉链表。（从根到每一个结点的一条路径都是一个线性表）
- 树：1对n的关系，通过“孩子-兄弟链表”转换成二叉树。

图：多对多的关系，如何表示？

图的存储结构

图的存储表示分析

- 特点：顶点之间的关系是多对多(m : n)
  m 和 n 都是不定的，无法进行非线性结构的线性化。
- 顶点之间的逻辑关系无法通过顶点之间的存储位置来 反映；必须引入额外的存储空间专门描述顶点之间的邻接关系。
- 图的存储信息
  - 整体信息：顶点数、边/弧数、图的种类(DG/DN/UG/UN)
  - 顶点信息：描述n个顶点的基本情况，可用一个顺序表来描述。
  - 边（弧）信息：描述e条边（弧）的基本信息。（邻接矩阵、邻接表、多重邻接表、十字链表)

- 邻接矩阵

图的存储结构-邻接矩阵

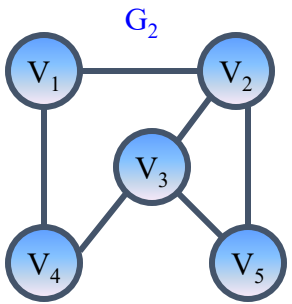
数组表示法——邻接矩阵

```
#define MAX_VERTEX_NUM 10
typedef int Status;
typedef int InfoType;
typedef int VertexType;
```

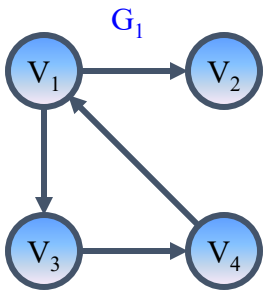
```
#define  INFINITY         INT_MAX
typedef  enum{DG, DN, AG, AN} GraphKind;    // 图的种类


typedef struct ArcCell{
   int        adj;    // 顶点间关系，无权图：0-不相邻，1-相邻
                      //                 有权图：权值-相邻，INFINITY-不相邻
   infoType        * info;   // 该弧相关信息的指针，可以略掉
}ArcCell, AdjMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];


typedef struct {
   VertexType  vexs[MAX_VERTEX_NUM];          // 顶点向量
   AdjMatrix        arcs;                     // 关系集
   int              vexnum, arcnum;           // 顶点数、边/弧数
   GraphKind   kind;                          // 图的种类，选定做哪种图后，可以略掉
}MGraph;
```

图的存储结构-邻接矩阵

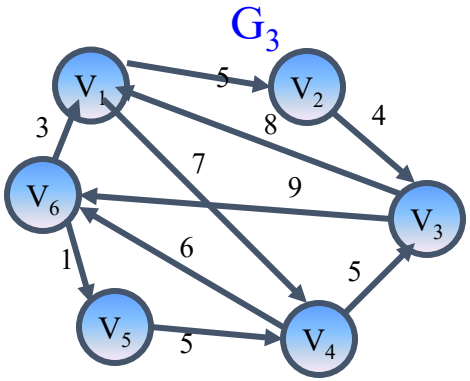
- 无向图/网—对称矩阵 有向图/网—未必是对称矩阵



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



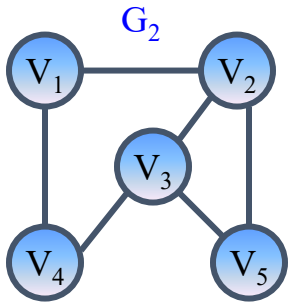
$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

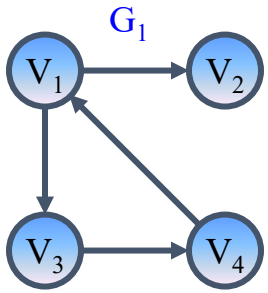


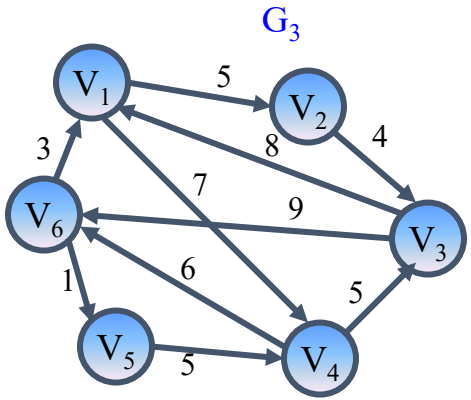
$$\begin{pmatrix} \infty & 5 & \infty & 7 & \infty & \infty \\ \infty & \infty & 4 & \infty & \infty & \infty \\ 8 & \infty & \infty & \infty & \infty & 9 \\ \infty & \infty & 5 & \infty & \infty & 6 \\ \infty & \infty & \infty & 5 & \infty & \infty \\ 3 & \infty & \infty & \infty & 1 & \infty \end{pmatrix}$$

图的存储结构-邻接矩阵

- 在邻接矩阵上求度、入度、出度







V1的度：第1行或列中值为1的元素个数

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

V1的出度

V1的入度

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

V1的出度

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

图的存储结构-邻接矩阵优缺点P155

- 优点：直观，易于实现
  - 便于判断两个顶点之间是否有边；
  - 便于统计顶点的度；

- 缺点：O（n^2）空间复杂度，对稀疏图而言太浪费空间
  - 不便于增加和删除结点；
  - 不便于统计边的数目；
  - 空间复杂度高

- **邻接矩阵**
  - 存储

图的存储结构-邻接矩阵

数组表示法——邻接矩阵

```
#define MaxInt 32767                    //表示极大值，即∞
#define MVNum 100                       //最大顶点数
typedef char VerTexType;                //假设顶点的数据类型为字符型
typedef int ArcType;                    //假设边的权值类型为整型
typedef struct{
  VerTexType vexs[MVNum];                       //顶点表
  ArcType arcs[MVNum][MVNum];                   //邻接矩阵
  int vexnum,arcnum;                //图的当前点数和边数
}MGraph;
```

使用邻接矩阵表示法创建无向网

算法思想

（1）输入总顶点数和总边数。
（2）依次输入点的信息存入顶点表中。
（3）初始化邻接矩阵，使每个权值初始化为极大值。
（4）构造邻接矩阵。

```
typedef struct{
  VerTexType vexs[MVNum];            //顶点表
  ArcType arcs[MVNum][MVNum];        //邻接矩阵
  int vexnum,arcnum;                 //图的当前点数和边数
}MGraph;
```

```
4    5
A B C D
A B 500
A C 200
A D 150
B C 400
C D 600
```

- **邻接矩阵的操作**
  - 创建及其他操作

```
4  5
A B C D
A B 500
A C 200
A D 150
B C 400
C D 600
```

```
1.  Status CreateUDN(MGraph &G){
2.    //采用邻接矩阵表示法，创建无向网G
3.    cin>>G.vexnum>>G.arcnum;                    //输入总顶点数，总边数
4.    for(i = 0; i<G.vexnum; ++i)
5.      cin>>G.vexs[i];                          //依次输入点的信息
6.    for(i = 0; i<G.vexnum;++i)                 //初始化邻接矩阵，边的权值均置为极大值
7.      for(j = 0; j<G.vexnum;++j)
8.        G.arcs[i][j] = MaxInt;
9.    for(k = 0; k<G.arcnum;++k){                //构造邻接矩阵
10.     cin>>v1>>v2>>w;                          //输入一条边依附的顶点及权值
11.     i = LocateVex(G, v1);  j = LocateVex(G, v2);  //确定v1和v2在G中的位置
12.     G.arcs[i][j] = w; //边<v1, v2>的权值置为w
13.     G.arcs[j][i] = G.arcs[i][j];             //置<v1, v2>的对称边<v2, v1>的权值为w
14.   }//for
15.   return OK;
16. }//CreateUDN
```

图的存储结构不同、图的类型不同，都会影响创建算法的实现细节；但是，图的总体创建流程是一致的!

```
1.  int LocateVex(MGraph G,VertexType u) {
2.  //存在则返回u在顶点表中的下标;否则返回-1
3.     int i;
4.     for(i=0;i<G.vexnum;++i)
5.       if(u==G.vexs[i])
6.          return i;
7.     return -1;
8.   }
```

```
int GraphEmpty(const MGraph& G) {
    return G.vexnum == 0;
}
//判图G空否,空则返回1,否则返回0。
```

```
int GetWeight (const MGraph& G, int u, int v) {
//给出以顶点 u 和 v 为两端点的边上的权值
    if (u != -1 && v != -1) return G.arcs[u][v];
    else return 0;
}
```

```
VertexType GetValue (const MGraph& G, int i ){
//给出第 i 个顶点的数据值
      return i >= 0 && i < G.vexnum ? G.vexs[i] :   '\0' ;
}    // VertexType 为char
```

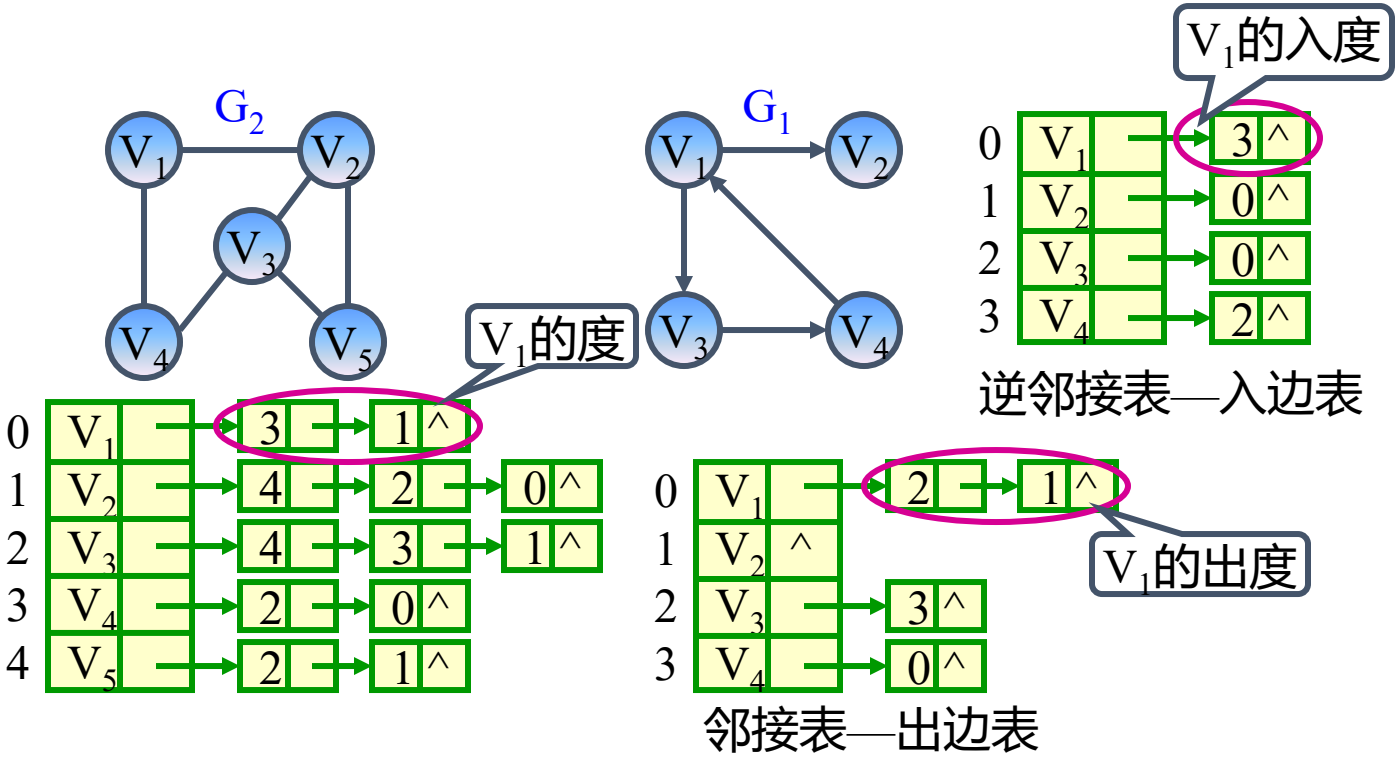
```
1.  int FirstAdjVex (MTGraph& G, int v) {
2.  //给出顶点位置为 v 的第一个邻接顶点的位置
3.      if ( v != -1 ) {
4.          for ( int col = 0; col < G.vexnum;  col++ )
5.              if ( G.arcs[v][col] > 0 && G.arcs[v][col] < INFINITY )
6.                  return col;
7.              //顺序检测第 v 行寻找第一个邻接顶点
8.      }
9.      return -1;
10. }
```

```
1.  int NextAdjVex (MTGraph& G, int v,  int w) {
2.  //给出顶点位置为 v 的第一个邻接顶点的位置
3.      if ( v != -1 ) {
4.          for ( int col = w+1; col < G.vexnum;  col++ )
5.              if ( G.arcs[v][col] > 0 && G.arcs[v][col] < INFINITY )
6.                  return col;
7.              //在第 v 行顺序寻找下一个邻接顶点
8.      }
9.      return -1;
10. }
```

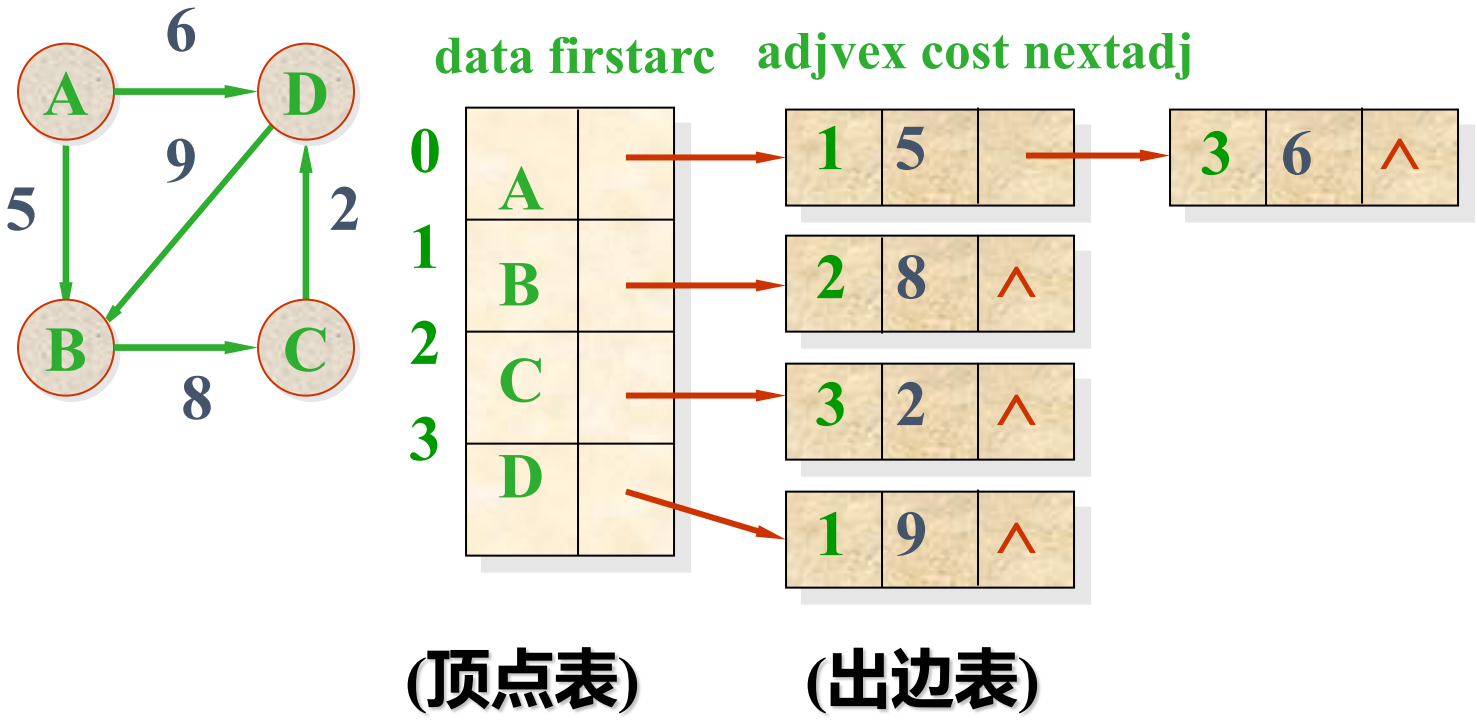
- 邻接表

图的存储结构-邻接表

- 无向图/网—边表 有向图/网—出边表



带权值，改造结点


6
A
D
9
5
2
B
C
8
data firstarc
adjvex cost nextadj
0
1
2
3
A
B
C
D
1 5
3 6 ^
2 8 ^
3 2 ^
1 9 ^
(顶点表)
(出边表)

图的存储结构-邻接表优缺点P158

- 优点：
  - 便于增加和删除结点；
  - 便于统计边的数目；
  - 空间使用效率高；

- 缺点：
  - 不便于判断顶点之间是否有边；
  - 不便于计算有向图各个顶点的度（*）；

- **邻接表**
  - 存储

7.2.2 图的存储结构-邻接表

- **邻接表：通过把顶点的所有邻接点组织成一个单链表来描述边**

```
#define MVNum 100                    //最大顶点数
typedef struct ArcNode{              //边结点
    int adjvex;                      //该边所指向的顶点的位置
    struct ArcNode * nextarc;        //指向下一条边的指针
    OtherInfo info;                  //和边相关的信息
}ArcNode;
typedef struct VNode{
    VerTexType data;                 //顶点信息
    ArcNode * firstarc;              //指向第一条依附该顶点的边的指针
}VNode, AdjList[MVNum];              //AdjList表示邻接表类型
typedef struct{
    AdjList vertices;                //邻接表
    int vexnum, arcnum;              //图的当前顶点数和边数
}ALGraph;
```

7.2.2 图的存储结构-邻接表

- **邻接表：通过把顶点的所有邻接点组织成一个单链表来描述边**

```
#define MVNum 100                    //最大顶点数
typedef struct ArcNode{              //边结点
    int adjvex;                      //该边所指向的顶点的位置
    struct ArcNode * nextarc;        //指向下一条边的指针
    EdgeData cost;                   //改造结点，typedef int EdgeData;
}ArcNode;
typedef struct VNode{
    VerTexType data;                 //顶点信息
    ArcNode * firstarc;              //指向第一条依附该顶点的边的指针
}VNode, AdjList[MVNum];              //AdjList表示邻接表类型
typedef struct{
    AdjList vertices;                //邻接表
    int vexnum, arcnum;              //图的当前顶点数和边数
}ALGraph;
```

- **邻接表**
  - 存储及其他操作

使用邻接表表示法创建无向网

算法思想

（1）输入总顶点数和总边数。
（2）依次输入点的信息存入顶点表中，使每个表头结点的指针域初始化为NULL。
（3）(头插法)创建邻接表。

使用邻接表表示法创建无向网


```
1.  Status CreateUDG(ALGraph &G){
2.        //采用邻接表表示法，创建无向图G
3.     cin>>G.vexnum>>G.arcnum;                  //输入总顶点数，总边数
4.     for(i = 0; i<G.vexnum; ++i){              //输入各点，构造表头结点表
5.        cin>> G.vertices[i].data;              //输入顶点值
6.        G.vertices[i].firstarc=NULL;           //初始化表头结点的指针域为NULL
7.     }//for
```

使用邻接表表示法创建无向网

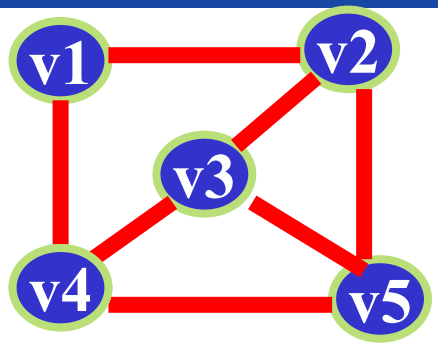
```
8.      for(k = 0; k<G.arcnum;++k){                          //输入各边，构造邻接表
9.          cin>>v1>>v2;                                     //输入一条边依附的两个顶点
10.         i = LocateVex(G, v1);  j = LocateVex(G, v2);
11.         p1=new ArcNode;                                  //生成一个新的边结点*p1
12.         p1->adjvex=j;                                    //邻接点序号为j
13.         p1->nextarc= G.vertices[i].firstarc;  G.vertices[i].firstarc=p1;
14.       //将新结点*p1插入顶点vi的边表头部
15.         p2=new ArcNode; //生成另一个对称的新的边结点*p2
16.         p2->adjvex=i;                                    //邻接点序号为i
17.         p2->nextarc= G.vertices[j].firstarc;  G.vertices[j].firstarc=p2;
18.       //将新结点*p2插入顶点vj的边表头部
19.     }//for
20.     return OK;
21. }//CreateUDG
```

```
1.  EdgeData GetWeight (const ALGraph& G,int u,int v){
2.  //给出以顶点 u 和 v 为两端点的边上的权值
3.      if (u != -1 && v != -1) {
4.          ArcNode * p = G.vertices[u].firstarc;
5.          while ( p != NULL ) {
6.              if ( p->adjvex == v ) return p->cost;
7.               else p = p->nextarc;
8.          }
9.      }
10.     return 0;
11. }
```

```
1.  VertexData GetValue (const ALGraph& G, int i ){
2.      return i >= 0 && i < G.n ? G.vertices[i].data :  “\0” ;
3.  }

1.  int FirstAdjVex (const ALGraph& G, int v) {
2.  //查找顶点 v 第一个邻接顶点在邻接表中位置
3.      if ( v != -1 ) {                          //若顶点存在
4.          ArcNode * p = G.vertices[v].firstarc;
5.          if ( p != NULL ) return p->adjvex;
6.      }
7.      return -1;                                //若顶点不存在
8.  }
```

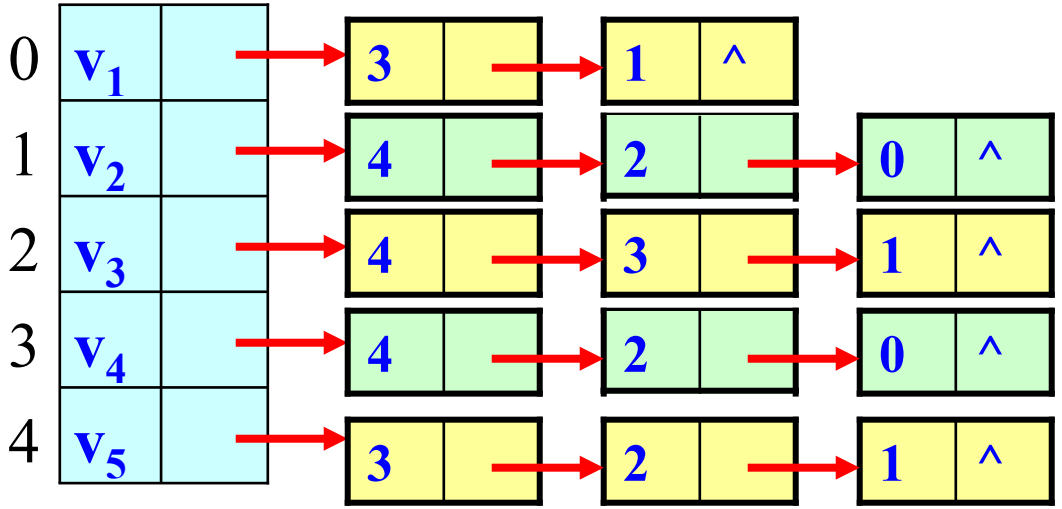
```
1.  int NextAdjVex (const ALGraph& G, int v, int w ) {
2.  //查找顶点 v 在邻接顶点 w 后下一个邻接顶点
3.      if ( v != -1 ) {
4.          ArcNode * p = G.vertices[v].firstarc;
5.          while ( p != NULL ) {
6.              if ( p->adjvex == w && p->next != NULL )
7.                  return p->nextarc->adjvex;
8.                      //返回下一个邻接顶点在邻接表中位置
9.              else p = p->nextarc;
10.         }
11.     }
12.     return -1;       //没有查到下一个邻接顶点
13. }
```

邻接矩阵和邻接表的对比



（ v1 v2 v3 v4 v5 ）

0	1	0	1	0	v1
1	0	1	0	1	v2
0	1	0	1	1	v3
1	0	1	0	1	v4
0	1	1	1	0	v5



1. 联系：邻接表中每个链表对应于邻接矩阵中的一行，链表中结点个数等于一行中非零元素的个数。

邻接矩阵和邻接表的对比

区别：

① 对于任一确定的无向图，邻接矩阵是唯一的（行列号与顶点编号一致），但邻接表不唯一（链接次序与顶点编号无关）。

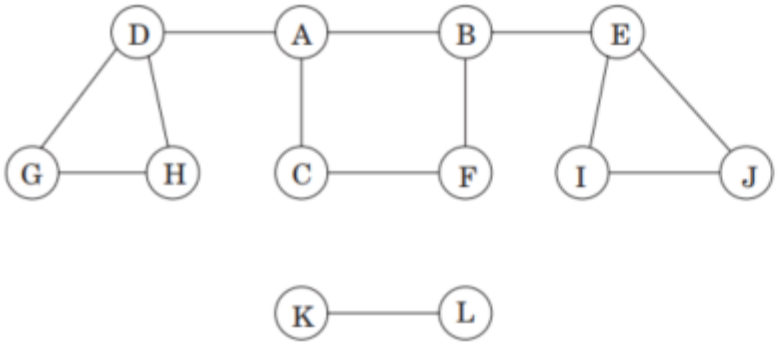
② 邻接矩阵的空间复杂度为$O(n^2)$,而邻接表的空间复杂度为$O(n+e)$。

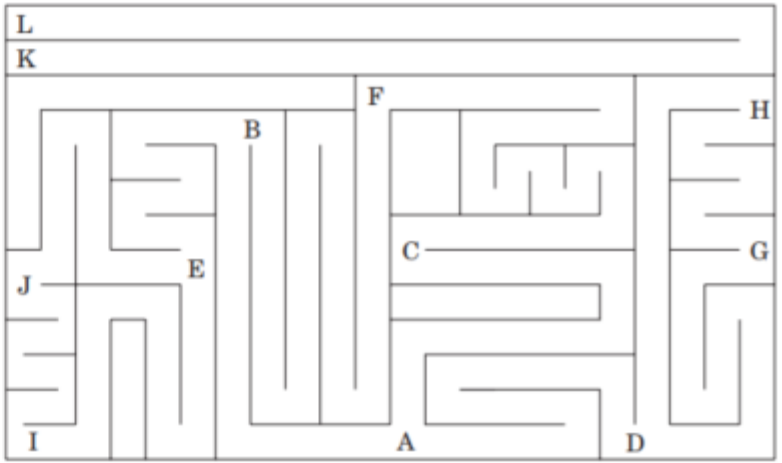
用途：邻接矩阵多用于稠密图；而邻接表多用于稀疏图

本节内容

- **图的遍历**
  - 深度优先遍历（DFS）
  - 广度优先遍历（BFS）

Figure 3.2 Exploring a graph is rather like navigating a maze.

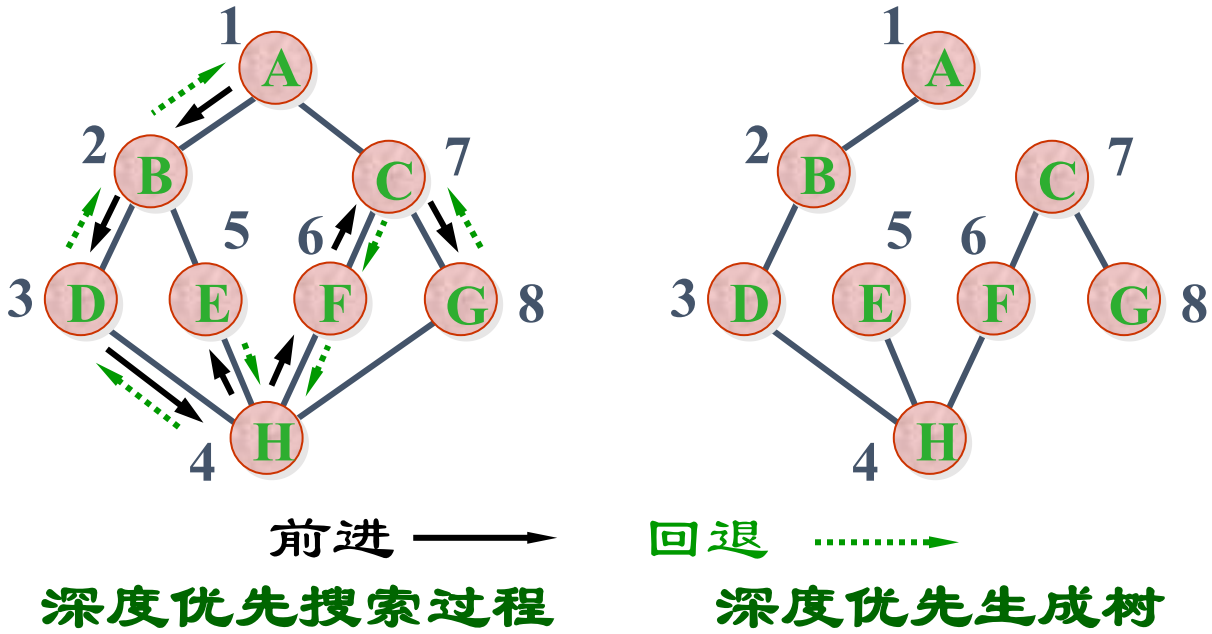




图的深度优先遍历 DFS(Depth First Search)

- **深度优先遍历：**
  - 从某顶点v 出发，访问该顶点；
  - 然后依次从v 的未被访问的邻接点出发深度优先遍历图；
    - 递归结束后，图中所有和v有路径相通的顶点都将被访问到。
  - 若图中还存在尚未访问过的顶点，则另选图中一个未曾被访问的顶点作起始点，继续重复上述过程
  - 分析
    - 类似于树的先序遍历
    - 引入访问标志数组visit[0:n-1]，区分顶点是否已被访问
    - 非连通图，需引入多个深度优先搜索的起始顶点
    - 递归算法或基于栈的非递归算法皆可实现

深度优先搜索的示例



图的深度优先遍历算法 DFS

```
1.  Bool visited[MAX_VERTEX_NUM];
2.  void  DFS(G,v ){
3.          visited[v] = true;          Visit(G, v);
4.          for ( w = FirstAdjVex( G, v);  w!=-1 ; w = NextAdjVex(G, v, w) )
5.                      if ( !visited[w] )        DFS(G, w);
6.  }
7.  void DFSTraverse(Graph G){
8.        for ( v = 0; v < G.vexnum; ++ v)        visited[v] = false;
9.        //memset(visited, 0, G.vexnum*sizeof(bool);
10.       for ( v = 0; v < G.vexnum; ++ v)
11.                     if ( !visited[v] )        DFS(G, v);
12.}
```

每调用一次 DFS() 就遍历了图的一个连通分量。

图的深度优先遍历算法


```
DFS(Graph G,  v1){
      visited[v1] = TRUE;        Visit(G1, 'v1');
      for ( w =(v2,v3))
            if ( !visited[w] )              DFS(G1, w);
}
```

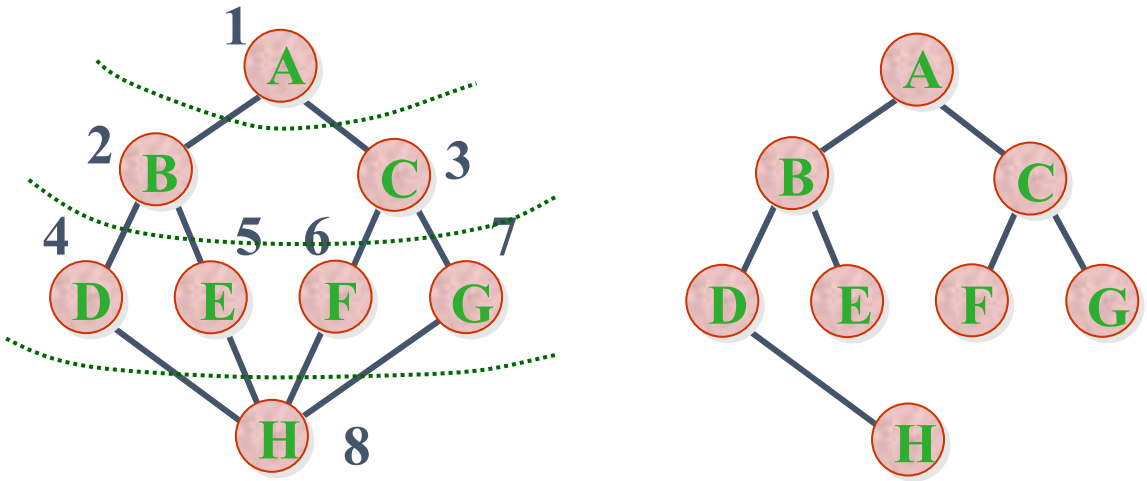
- DFS(Graph G, v2){
  - visited[v2] = TRUE; Visit(G1, 'v2');
  - for (w =(v1,v4,v5))
    - if (!visited[w]) DFS(G1, w);
  - }
    - DFS(Graph G, v4){
      - visited[v4] = TRUE; Visit(G1, 'v4');
      - for (w =(v2,v8))
        - if (!visited[w]) DFS(G1, w);
      - }
        - DFS(Graph G, v8){
          - visited[v8] = TRUE; Visit(G1, 'v8');
          - for (w =(v4,v5))
            - if (!visited[w]) DFS(G1, w);
          - }
            - DFS(Graph G, v5){
              - visited[v5] = TRUE; Visit(G1, 'v5');
              - for (w =(v2,v8))
                - if (!visited[w]) DFS(G1, w);
              - }
  - DFS(Graph G, v3){

- 请调试观察递归的变换！
- 另外，如果不用递归，应该怎么写代码？

图的广度优先遍历

- **广度优先遍历：**
  - 从某顶点v 出发，访问该顶点;
  - 然后依次访问v 的所有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“先被访问的顶点的邻接点”先于“后被访问的顶点的邻接点” 被访问，直至图中所有已被访问的顶点的邻接点都被访问到;
  - 若图中还存在尚未访问过的顶点，则另选图中一个未曾被访问的顶点作起始点，继续重复上述过程
  - 分析
    - 类似于树的层次遍历
    - 引入visited访问标志数组
    - 非连通图，需引入多个广度优先搜索的起始顶点
    - 引入队列保存“顶点已访问，但其邻接点未全访问”的顶点编号

广度优先搜索的示例



广度优先搜索过程 ·····► 广度优先生成树

图的遍历—广度优先遍历算法（BFS）


```
1.  void BFSTraverse(Graph G){
2.    for ( v = 0; v < G.vexnum; ++ v)     visited[v] = FALSE;
3.    InitQueue(Q);
4.    for ( v = 0; v < G.vexnum; ++ v)
5.        if ( !visited[v] ){
6.            visited[v] = TRUE;  Visit(G, v);
7.            EnQueue(Q, v);
8.            while(!QueueEmpty(Q)){
9.                DeQueue(Q,u);
10.               for ( w = FirstAdjVex( G, u) ; w ; w = NextAdjVex(G, u, w))
11.                   if ( !visited[w] ){
12.                       visited[w] = TRUE; Visit(G, w);
13.                       EnQueue(Q, w);
14.                   } //~if
15.           } //~while
16.       } //~if
17. }
```

作业预告

- 图的存储、创建及遍历