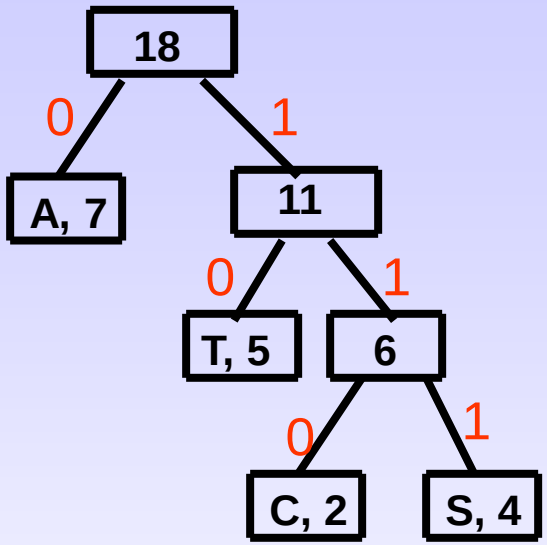
6.6 赫夫曼树及其应用

```
typedef struct
{     unsigned int weight ;
      unsigned int  parent, lchild, rchild ;
 } HTNode, * HuffmanTree;
 typedef char ** HufmanCode ;
```



赫夫曼编码的实现：

1、建立具有2n-1个单元的数组，其中n个单元用于保存初始结点，n-1个结点用于表示内部结点。

2、构造一棵赫夫曼树，即执行n-1次循环，每次产生一个内部结点。权值最小的两个结点为其左右子结点。

3、根据赫夫曼树计算每个字符的赫夫曼编码。

6.6 赫夫曼树及其应用

```
void  hufmanCode(HufumanTree &HT, HumanCode &HC, int *w, int n)
{  if  ( n<=1 ) return;
    m = 2*n-1;
    HT=(HuffmanTree)malloc((m+1)*sizeof(HTNode)); // 不用0号单元
    for ( p=HT+1, i=1; i<=n; ++i, ++p, ++w )   *p = { *w, 0, 0, 0 } ;
    for ( ; i <= m; ++i, ++p )  *p = { 0, 0, 0, 0 } ;
    for ( i = n+1; i <= m; ++i )  // 构造赫夫曼树
    {  Select ( HT, i-1, s1, s2 ) ; //小于等于i-1、根结点权值最小、无双亲
        HT[s1].parent = i; HT[s2].parent = i;
        HT[i].lchild = s1; HT[i].rchild = s2  ;
        HT[i].weight =  HT[s1].weight + HT[s2].weight;
    }
```

6.6 赫夫曼树及其应用

```
    HC = ( HuffmanCode ) malloc (( n + 1) * sizeof ( char * ));
    cd = ( char * ) malloc ( n * sizeof ( char ) );
    cd[n-1] ='/0' ;
    for ( i = 1; i <= n; ++i )    // 求每个字符的赫夫曼编码
    {  start = n-1;
       for ( c=i, f=HT[i].parent;  f != 0;  c=f, f=HT[f].parent )
          if (HT[f].lchild == c )   cd[--start ] = '0' ;
          else  cd [--start ] = '1' ;
       HC[i] = ( char * ) malloc (( n - start ) * sizeof ( char ));
       strcpy(HC[i], &cd[start]);
    }
    free(cd);
}
```

6.6 赫夫曼树及其应用

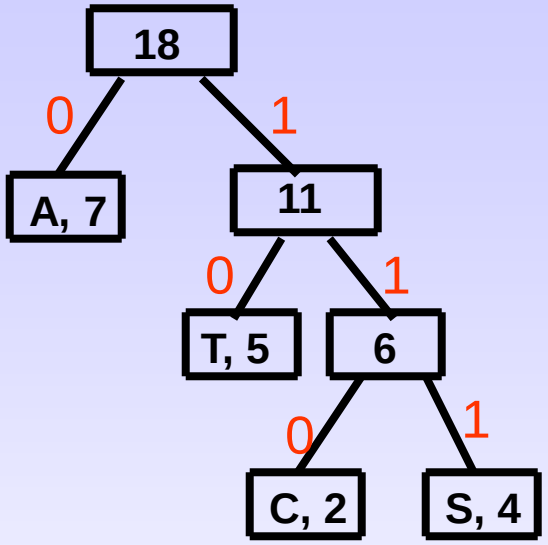
HC

字符	编码			
A，7	0	‘/0’		
T，5	1	0	‘/0’	
C，2	1	1	0	‘/0’
S，4	1	1	1	‘/0’

cd（在求T的编码时的状态）

	1	0	‘/0’

start=1



	wt	pa	lc	rc
1	7	7	0	0
2	5	6	0	0
3	2	5	0	0
4	4	5	0	0
5	6	6	3	4
6	11	7	2	5
7	18	0	1	6

第七章 图

7.1 图的定义和术语

- 图定义　图是由顶点集合(vertex)及顶点间的关系集合组成的一种数据结构：Graph＝(*V*, *E*)

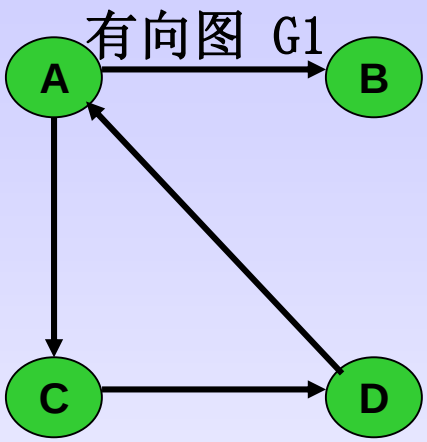
  其中　$V = \{ x \mid x \in$ 某个数据对象$\}$ 是顶点的有穷非空集合；

  $E = \{(x, y) \mid x, y \in V\}$　或

  $E = \{<x, y> \mid x, y \in V \ \&\& \ Path\ (x, y)\}$ 是顶点之间关系的有穷集合，也叫做边(edge)集合。*Path* (*x*, *y*) 表示从 *x* 到 *y* 的一条单向通路, 它是有方向的。

- 有向图与无向图　在有向图中，顶点对<x, y>是有序的。在无向图中，顶点对(x, y)是无序的。

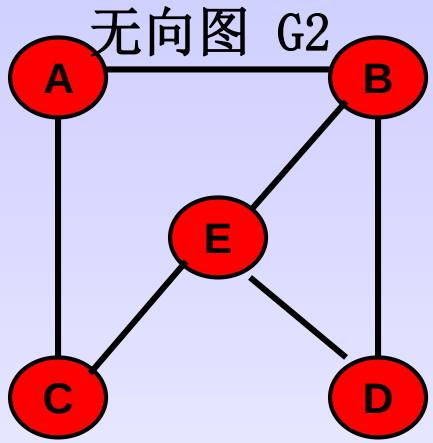
7.1 图的定义和术语



- 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>}



- 顶点：

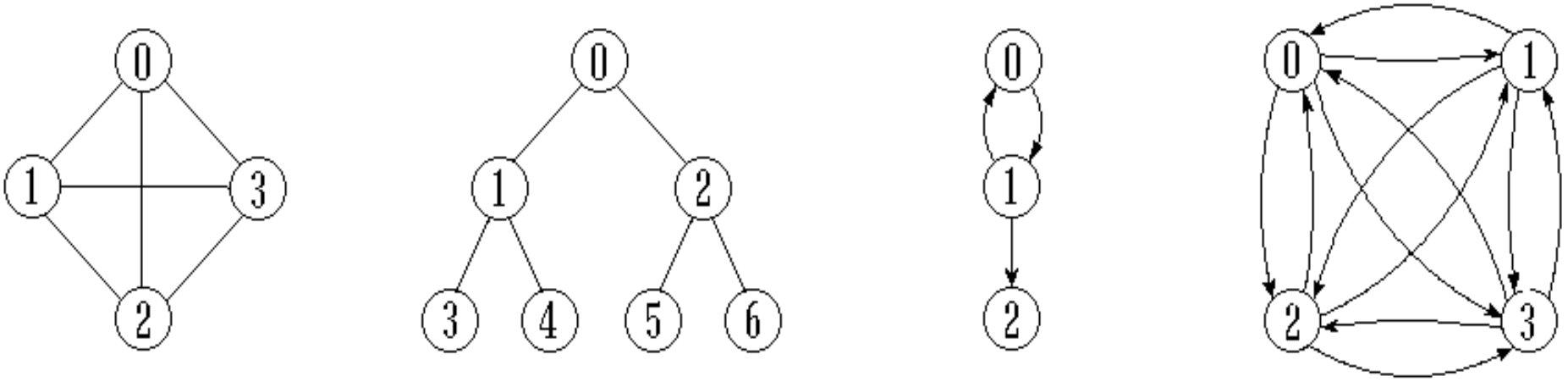
- 无向边或边

- 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)}

7.1 图的定义和术语

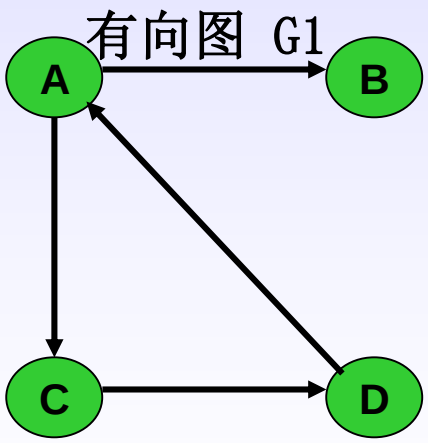
- 完全图　若有 *n* 个顶点的无向图有 *n*(*n*-1)/2 条边, 则此图为完全无向图。有 *n* 个顶点的有向图有*n*(*n*-1) 条边, 则此图为完全有向图。
- 权　某些图的边具有与它相关的数, 称之为权。带权图叫做网。



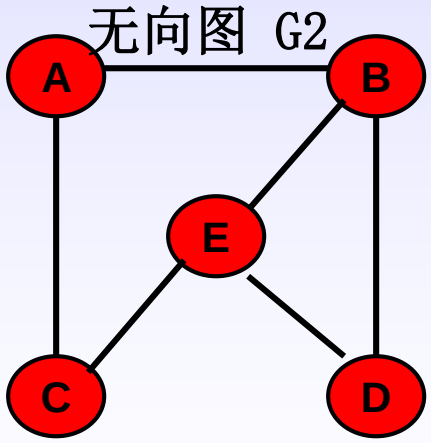
(a) G1　(b) G2　(c) G3　(d) G4

7.1 图的定义和术语

- 邻接顶点　如果 (u, v) 是 $E(G)$ 中的一条边，则称 u , v 互为邻接顶点。
- 顶点的度　顶点v的度是与它相关联的边的条数。记作$TD(v)$。在有向图中,顶点v的入度是以v为终点的有向边的条数, 记作 $ID(v)$; 顶点v的出度是以 v 为始点的有向边的条数, 记作$OD(v)$。



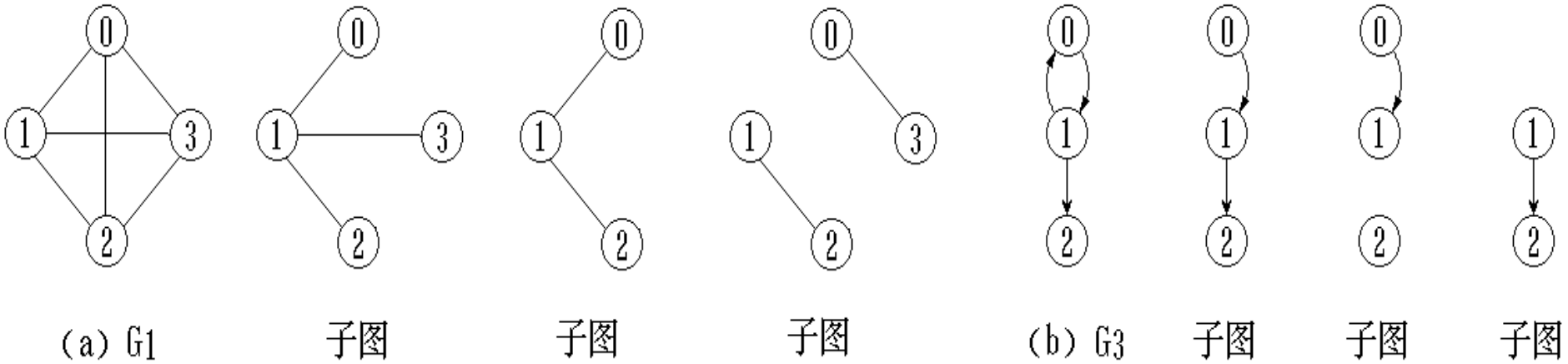
ID(A)=1; OD(A)=2;
ID(B)=1; OD(B)=0;
……



TD(A)=2; TD(B)=3;
TD(C)=2; TD(D)=2;
TD(E)=3

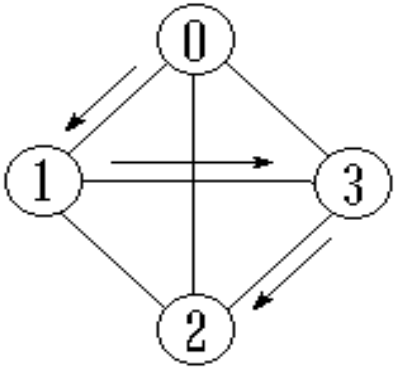
7.1 图的定义和术语

- 子图 设有两个图 G＝(*V*, *E*) 和 G′＝(*V*′, *E*′)。若 *V*′⊆ *V* 且 *E*′⊆*E*, 则称 图G′ 是图G 的子图。

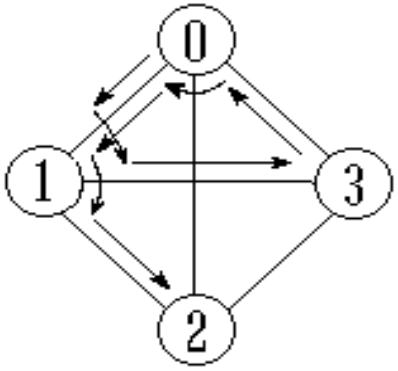


7.1 图的定义和术语

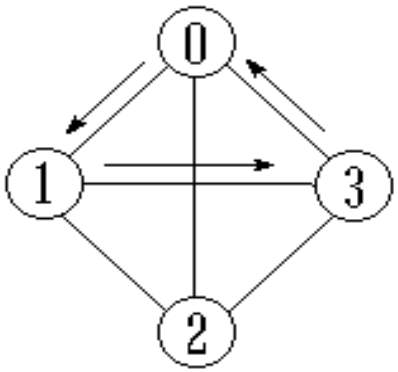
- 路径　在图 G＝(*V*, *E*) 中, 若从顶点 v_i 出发, 沿一些边经过一些顶点 v_{p1}, v_{p2}, …, v_{pm}，到达顶点v_j。则称顶点序列 (v_i v_{p1} v_{p2} … v_{pm} v_j) 为从顶点v_i 到顶点 v_j 的路径。它经过的边(v_i, v_{p1})、(v_{p1}, v_{p2})、…、(v_{pm}, v_j)应是属于*E*的边。
- 简单路径　若路径上各顶点 v_1,v_2,...,v_m 均不互相重复, 则称这样的路径为简单路径。



（a）简单路径



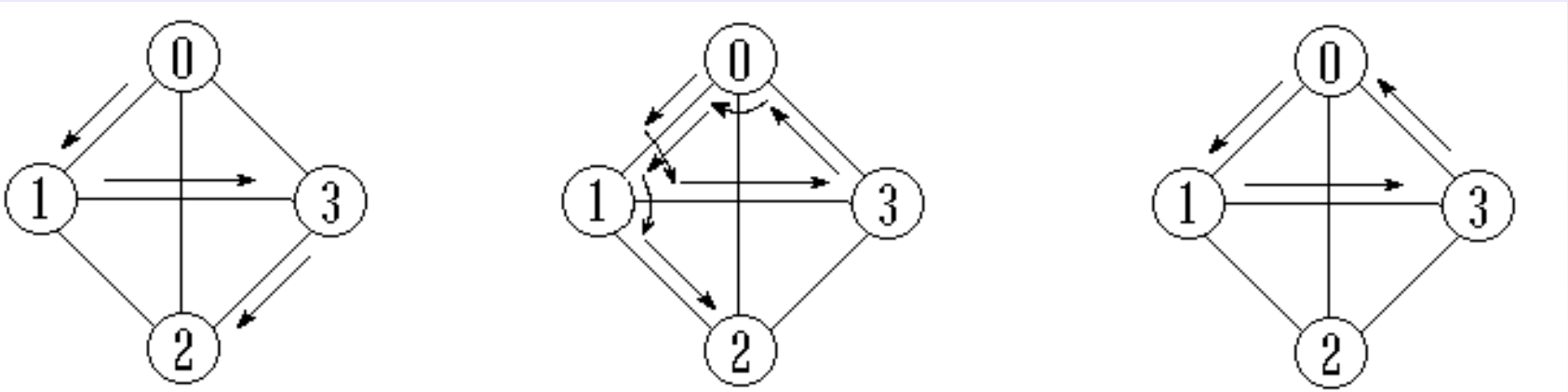
（b）非简单路径



（c） 回路

7.1 图的定义和术语

- 回路　若路径上第一个顶点 v_1 与最后一个顶点 v_m 重合，则称这样的路径为回路或环。
- 路径长度
  - 非带权图的路径长度是指此路径上边的条数。
  - 带权图的路径长度是指路径上各边的权之和。

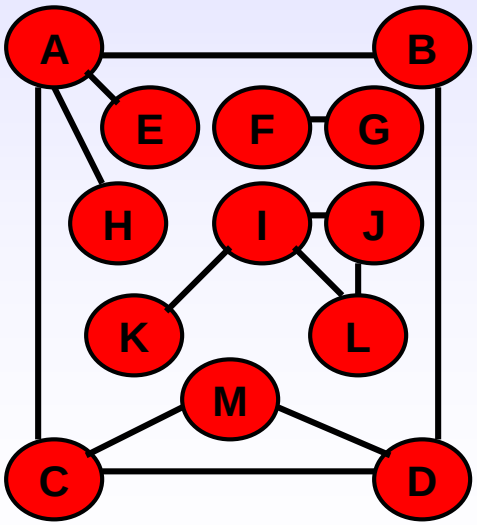


（a）简单路径　（b）非简单路径　（c） 回路

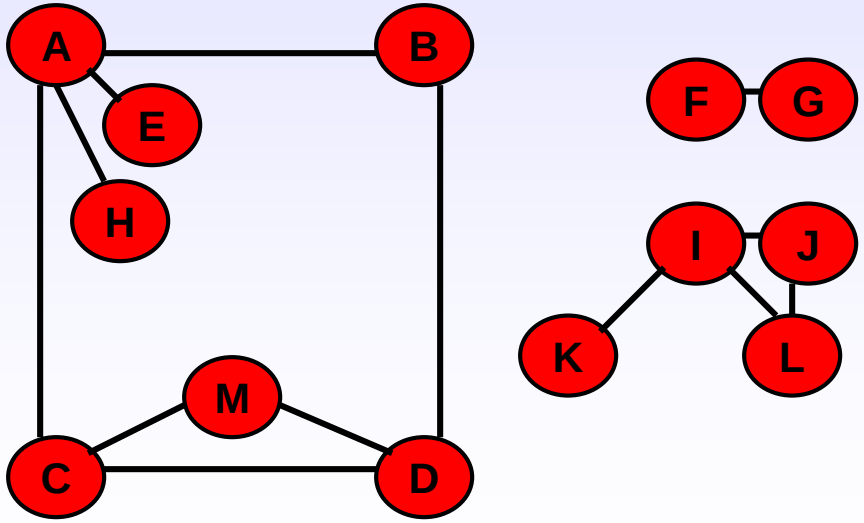
7.1 图的定义和术语

- **连通图与连通分量** 在无向图中, 若从顶点v_1到顶点v_2有路径, 则称顶点v_1与v_2是连通的。如果图中任意一对顶点都是连通的, 则称此图是连通图。非连通图的极大连通子图叫做连通分量。

无向图G



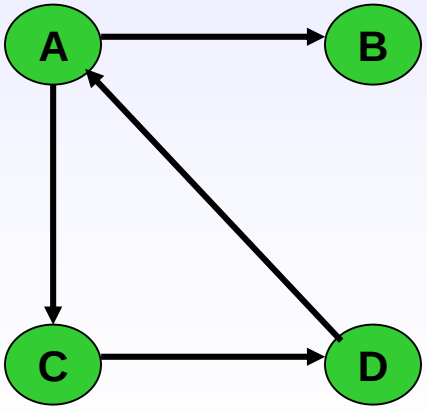
无向图G的三个连通分量



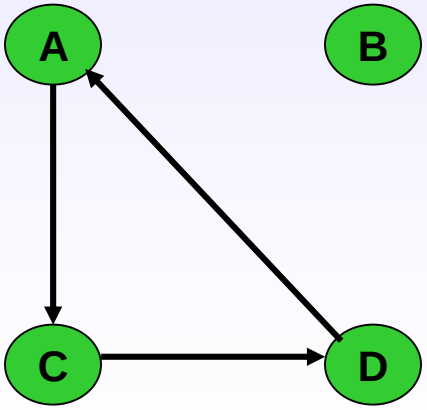
7.1 图的定义和术语

- 强连通图与强连通分量　在有向图中, 若对于每一对顶点v_i和v_j, 都存在一条从v_i到v_j和从v_j到v_i的路径, 则称此图是强连通图。非强连通图的极大强连通子图叫做强连通分量。

有向图G



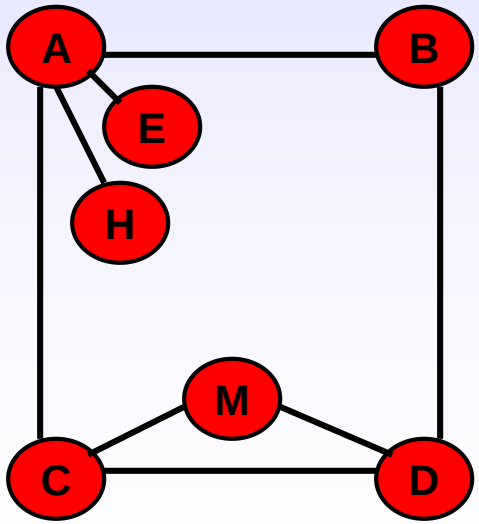
有向图G的两个强连通分量



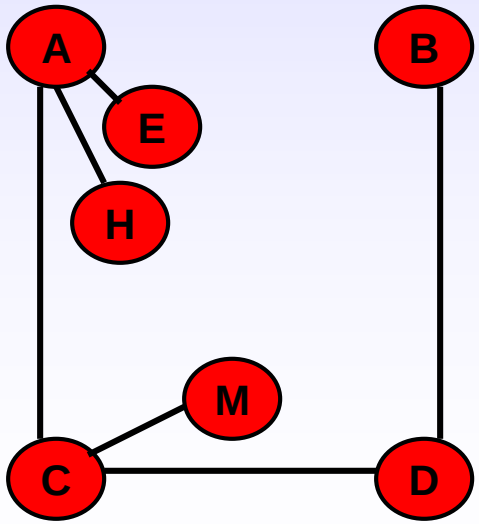
7.1 图的定义和术语

- 生成树　一个连通图的生成树是它的极小连通子图，包含连通图的全部*n*个顶点，但只有构成一棵树的*n*-1条边。

无向图G



无向图G的生成树



7.1 图的定义和术语

图的基本操作

- 结构的建立和销毁

```
CreatGraph(&G, V, VR);    // 按定义(V, VR) 构造图

DestroyGraph(&G);  // 销毁图
```

- 对顶点的访问操作

```
LocateVex(G, u);  // 若G中存在顶点u，则返回该顶点在
                  //  图中“位置”；否则返回其它信息。

GetVex(G, v);       // 返回 v 的值。

PutVex(&G, v, value);   // 对 v 赋值value。
```

7.1 图的定义和术语

- 对邻接点的操作

FirstAdjVex(G, v); // 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。

NextAdjVex(G, v, w); // 返回 v 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”。

- 插入或删除顶点

InsertVex(&G, v); //在图G中增添新顶点v。

DeleteVex(&G, v); // 删除G中顶点v及其相关的弧。

7.1 图的定义和术语

- 插入和删除弧

InsertArc(&G, v, w); // 在G中增添弧<v,w>

DeleteArc(&G, v, w); //在G中删除弧<v,w>

- 遍历

DFSTraverse(G, v, Visit());
// 从顶点v起**深度优先**遍历图G，并对每个顶点调用函数
// Visit一次且仅一次。

BFSTraverse(G, v, Visit());
// 从顶点v起广度优先遍历图G，并对每个顶点调用函数
// Visit一次且仅一次。

7.2 图的存储结构

7.2.1 邻接矩阵（Adjacency Matrix）

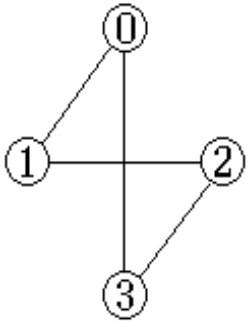
- 在图的邻接矩阵表示中，有一个记录各个顶点信息的顶点表，还有一个表示各个顶点之间关系的邻接矩阵。
- 设图 A = (*V*, *E*)是一个有 *n* 个顶点的图，则图的邻接矩阵是一个二维数组 A.*edge*[*n*][*n*]，定义：

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

i，*j*表示存放在顶点表中第*i*个和第*j*个顶点。

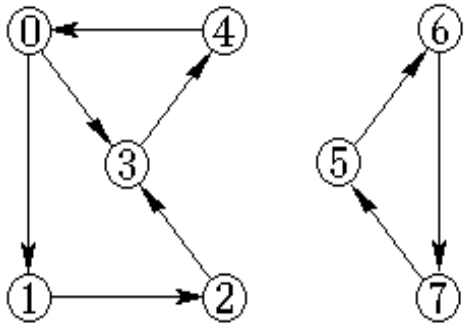
- 无向图的邻接矩阵是对称的，有向图的邻接矩阵可能是不对称的。

7.2 图的存储结构



G8

G3



G7

$$A.Edge = \begin{array}{c} \begin{matrix} ⓪ & ① & ② & ③ \end{matrix} \\ \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \begin{matrix} ⓪ \\ ① \\ ② \\ ③ \end{matrix} \end{array}$$

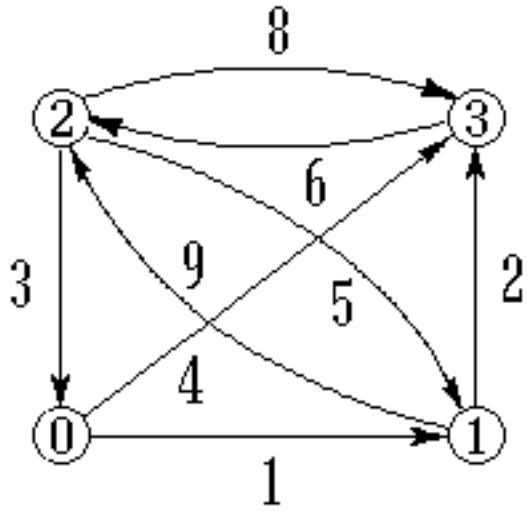
$$A.Edge = \begin{array}{c} \begin{matrix} ⓪ & ① & ② \end{matrix} \\ \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \begin{matrix} ⓪ \\ ① \\ ② \end{matrix} \end{array}$$

$$A.Edge = \begin{array}{c} \begin{matrix} ⓪ & ① & ② & ③ & ④ & ⑤ & ⑥ & ⑦ \end{matrix} \\ \left(\begin{array}{ccccc:ccc} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hdashline 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{array}\right) \begin{matrix} ⓪ \\ ① \\ ② \\ ③ \\ ④ \\ ⑤ \\ ⑥ \\ ⑦ \end{matrix} \end{array}$$

7.2 图的存储结构

- 网的邻接矩阵

$$A.Edge[i][j]=\begin{cases} W(i,j), & \text{如果 } i!=j \text{ 且 } <i,j>\in E \text{ 或 } (i,j)\in E \\ \infty, & \text{否则, 但是 } i!=j \\ 0\text{或}\infty, & \text{对角线 } i==j \end{cases}$$



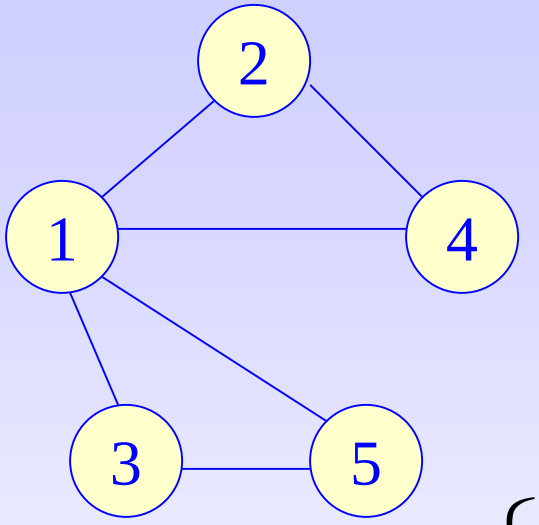
$$A.Edge=\begin{array}{c} \begin{matrix} ⓪ & ① & ② & ③ \end{matrix} \\ \begin{pmatrix} 0 & 1 & \infty & 4 \\ \infty & 0 & 9 & 2 \\ 3 & 5 & 0 & 8 \\ \infty & \infty & 6 & 0 \end{pmatrix} \end{array} \begin{matrix} ⓪ \\ ① \\ ② \\ ③ \end{matrix}$$

7.2 图的存储结构

- 用邻接矩阵表示图时，除了存储用于表示顶点间相邻关系的邻接矩阵外，还需要一个数组存储顶点。表示形式为：

```
#define  MAX  20
typedef enum{ DG, DN, AG, AN} GraphKind;
typedef struct{
        VertexType    vexs[MAX];        //顶点数组
        int           Edge[MAX][MAX];   //邻接矩阵
        int           vexnum;           //顶点数
        int           arcnum;           //边数
        GraphKind     kind;             //图的类型
} Mgraph;
```

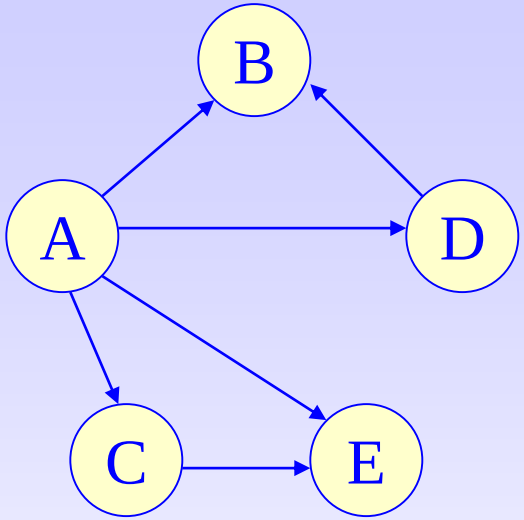
7.2 图的存储结构



$$vexs = \begin{Bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{Bmatrix}$$

$$\text{Edge} = \begin{Bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{Bmatrix}$$

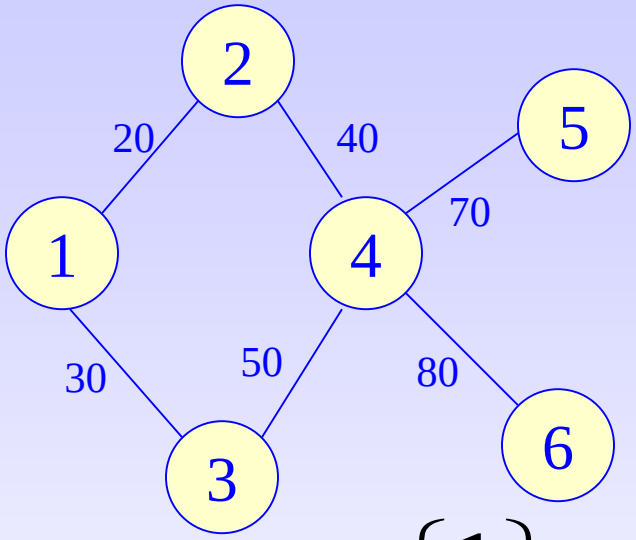
7.2 图的存储结构



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

$$vexs = \begin{Bmatrix} A \\ B \\ C \\ D \\ E \end{Bmatrix}$$

7.2 图的存储结构



$$vexs = \begin{Bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{Bmatrix}$$

$$\text{Edge} = \begin{Bmatrix} \infty & 20 & 30 & \infty & \infty & \infty \\ 20 & \infty & \infty & 40 & \infty & \infty \\ 30 & \infty & \infty & 50 & \infty & \infty \\ \infty & 40 & 50 & \infty & 70 & 80 \\ \infty & \infty & \infty & 70 & \infty & \infty \\ \infty & \infty & \infty & 80 & \infty & \infty \end{Bmatrix}$$

7.2 图的存储结构

7.2.2 邻接表 (Adjacency List)

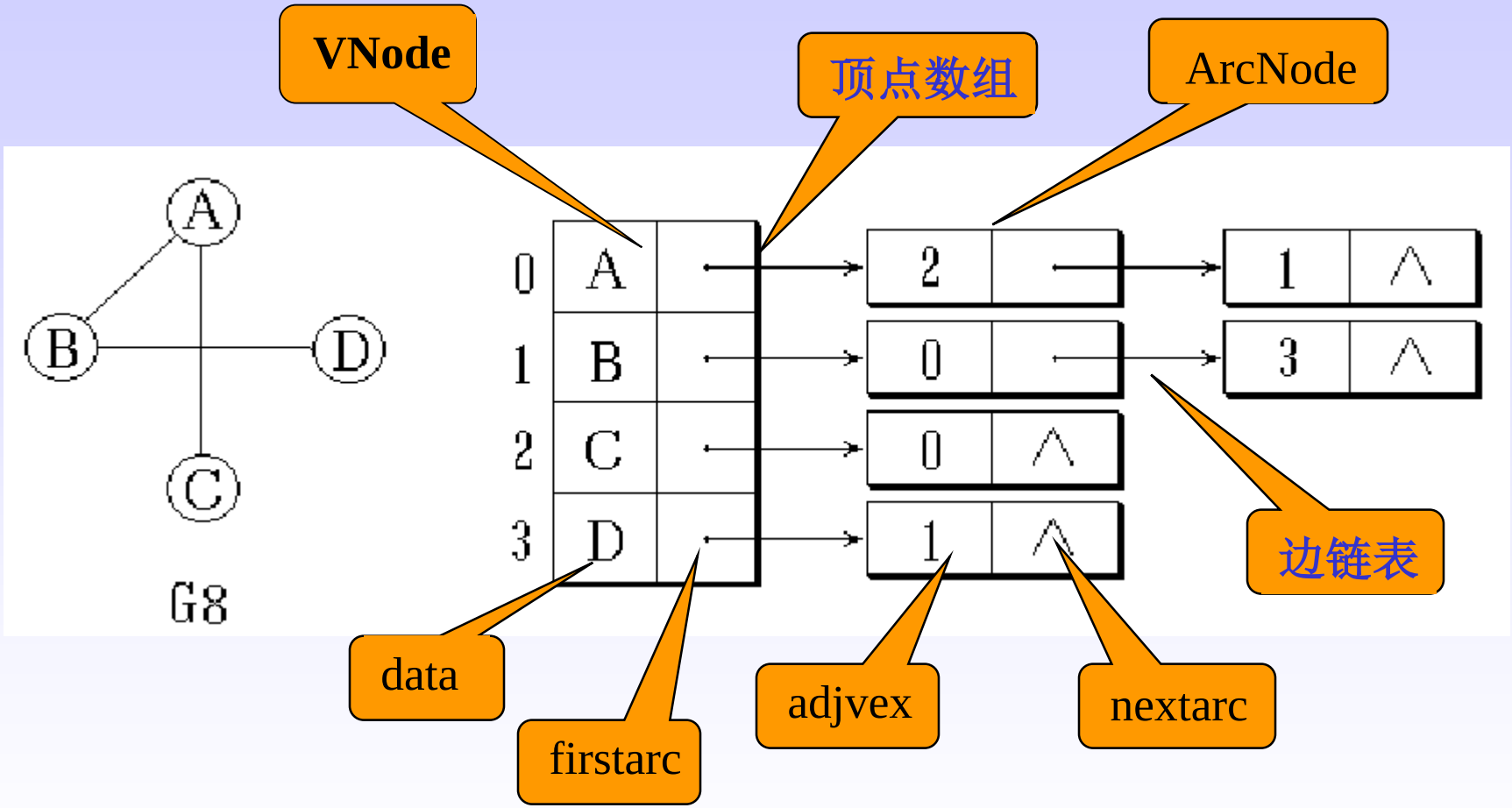
- 无向图的邻接表：把同一个顶点发出的边链接在同一个链表中，链表的每一个结点代表一条边，叫做边结点，边结点中保存有与该边相关联的另一顶点的顶点下标 adjvex 和指向同一链表中下一个边结点的指针 nextarc。
- 顶点表：用数组的形式存放所有的顶点及对应的边链表的头指针。
- 边链表：每条边用一个结点进行表示。同一个结点的所有的边形成它的边结点单链表。
- 在邻接表的边链表中，各个边结点的链接顺序任意，视边结点输入次序而定。

7.2 图的存储结构

邻接表的表示形式：

```
typedef struct ArcNode{
        int                 adjvex;
        struct ArcNode *nextarc;
        InfoType          *info; }  ArcNode;  // 边结点类型
typedef struct VNode{
        VertexType        data;
        ArcNode           *fistarc;
} Vnode;          // 顶点结点类型
typedef struct{
        Vnode             vertices[MAX];    //顶点数组
        int               vexnum,arcnum; //顶点数和边数
        GraphKind         kind;              //图的类型
} ALGraph;
```

7.2 图的存储结构

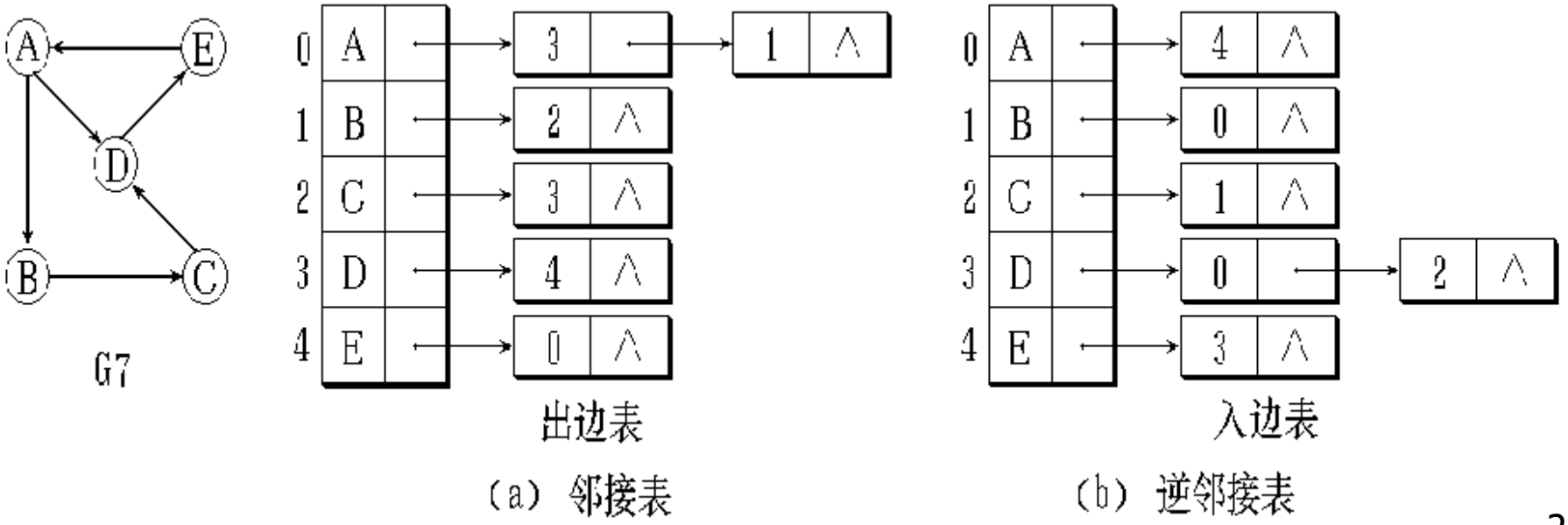

VNode
顶点数组
ArcNode
A
B
C
D
G8
0
1
2
3
A
B
C
D
2
0
0
1
1
3
边链表
data
firstarc
adjvex
nextarc

7.2 图的存储结构

- 有向图可以建立邻接表和逆邻接表：

  1）在有向图的邻接表中，第 *i* 个边链表链接的边都是顶点 *i* 发出的边。也叫做出边表。

  2）在有向图的逆邻接表中，第 *i* 个边链表链接的边都是进入顶点 *i* 的边。也叫做入边表。



7.2 图的存储结构

```
void CreateALGraph(ALGraph &G)   // 创建无向图的邻接表
{  scanf(&G.vexnum, &G.arcnum, &G.kind);
   for(k=1; k<=G.vexnum; k++)     // 输入顶点，建立顶点数组
   {  G.vextices[k].firstarc=NULL; scanf(&G.vextices[k].data); }
   for(k=1; k<=G.arcnum; k++)     // 输入边，建立边链表
   {   scanf(&x1, &x2);
       i=Locatevex(G,x1); j=Locatevex(G, x2);
       p=(ArcNode *)malloc(sizeof(ArcNode));
       p->adjvex=j; p->nextarc=G.vextices[i].firstarc;
       G.vextices[i].firstarc=p;          // 插入到顶点i的边链表
       p=(ArcNode *)malloc(sizeof(ArcNode));
       p->adjvex=i; p->nextarc=G.vextices[j].firstarc;
       G.vextices[j].firstarc=p;          // 插入到顶点j的边链表
   }
}
```

7.2 图的存储结构

```
void CreateALGraph(ALGraph &G)   // 创建无向图的邻接表
{  scanf(&G.vexnum, &G.arcnum, &G.kind);
   for(k=1; k<=G.vexnum; k++)     // 输入顶点，建立顶点数组
   {  G.vextices[k].firstarc=NULL; scanf(&G.vextices[k].data); }
   for(k=1; k<=G.arcnum; k++)     // 输入边，建立边链表
   {   scanf(&x1, &x2);
       i=Locatevex(G,x1); j=Locatevex(G, x2);
       p=(ArcNode *)malloc(sizeof(ArcNode));
       p->adjvex=j; p->nextarc=G.vextices[i].firstarc;
       G.vextices[i].firstarc=p;          // 插入到顶点i的边链表
       p=(ArcNode *)malloc(sizeof(ArcNode));
       p->adjvex=i; p->nextarc=G.vextices[j].firstarc;
       G.vextices[j].firstarc=p;          // 插入到顶点j的边链表
   }
}
```

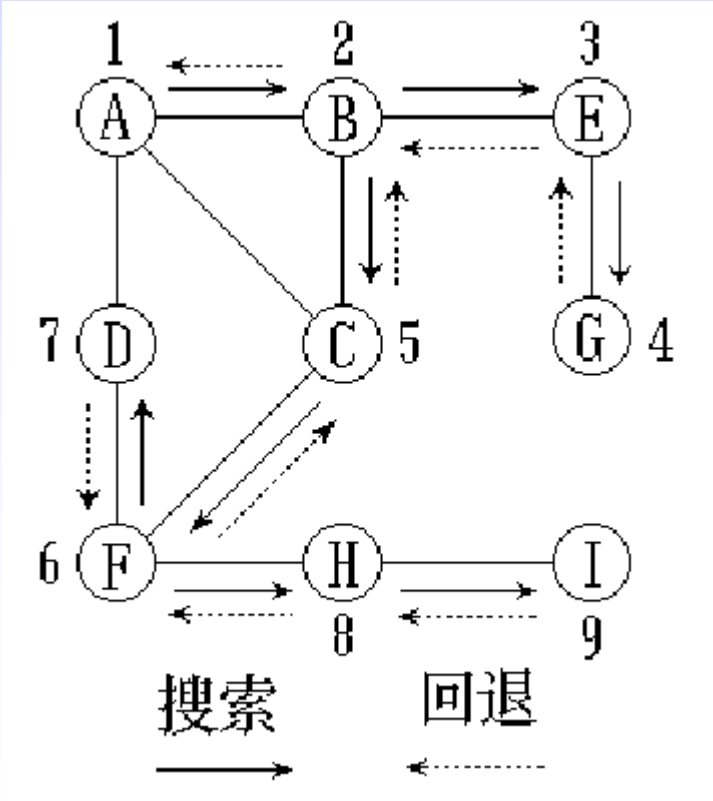
7.3 图的遍历

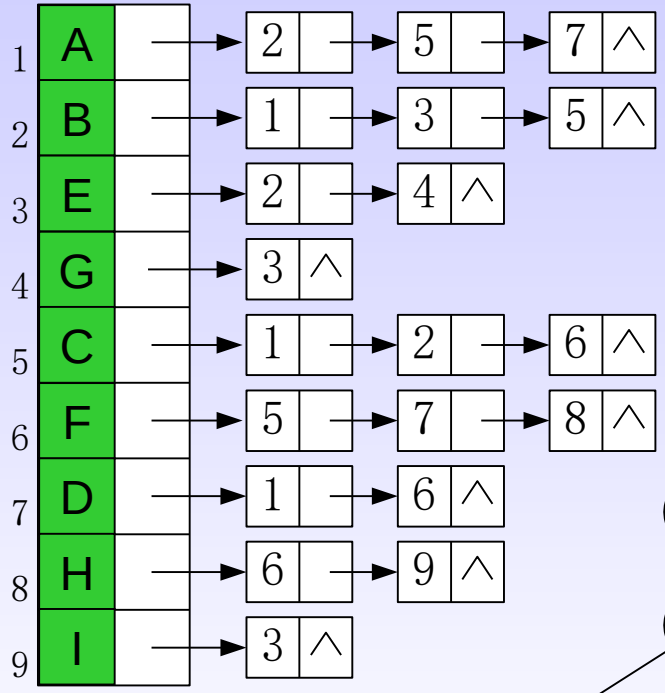
- 从图中某一顶点出发，沿着一些边访问图中所有的顶点，且使每个顶点仅被访问一次，就叫做图的遍历 (Graph Traversal)。
- 图中可能存在回路，且图的任一顶点都可能与其它顶点相通，在访问完某个顶点之后可能会沿着某些边又回到了曾经访问过的顶点。
- 为了避免重复访问，可设置一个标志顶点是否被访问过的辅助数组 *visited* [1..n]，它的初始状态为 FALSE，在图的遍历过程中，一旦某一个顶点 *i* 被访问，就立即让 *visited* [*i*] 为 TRUE，防止它被多次访问。
- 常用的遍历方法有深度优先搜索（DFS）和广度优先搜索（BFS）。

7.3 图的遍历

7.3.1 深度优先搜索

- 深度优先搜索（连通图）：

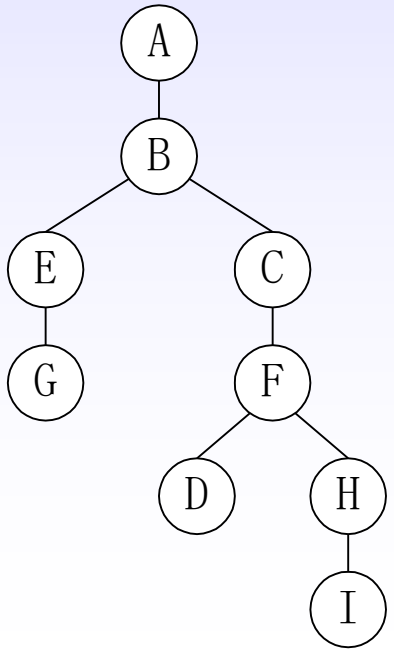




遍历序列：

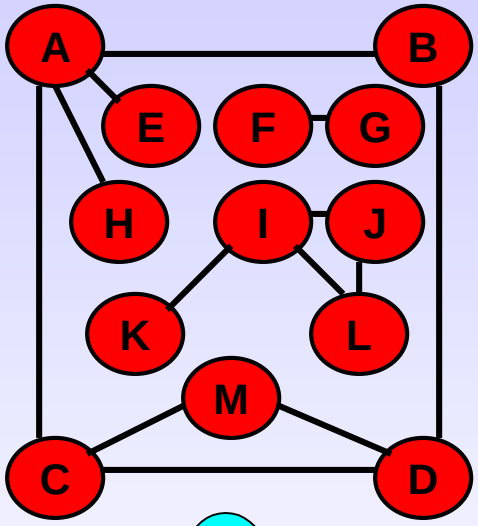
A，B，E，G，C，

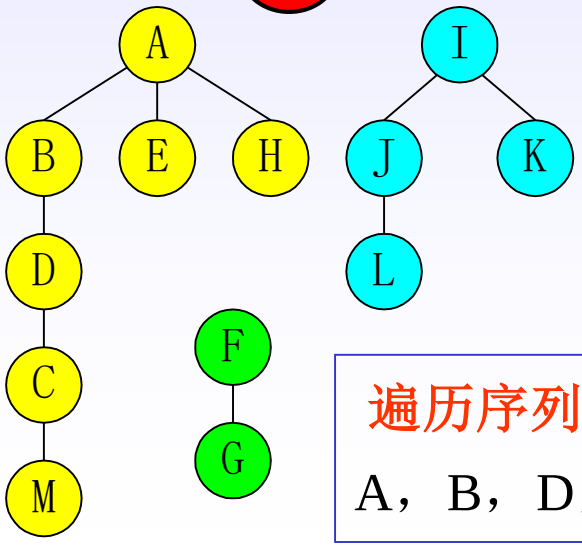
F，D，H，I

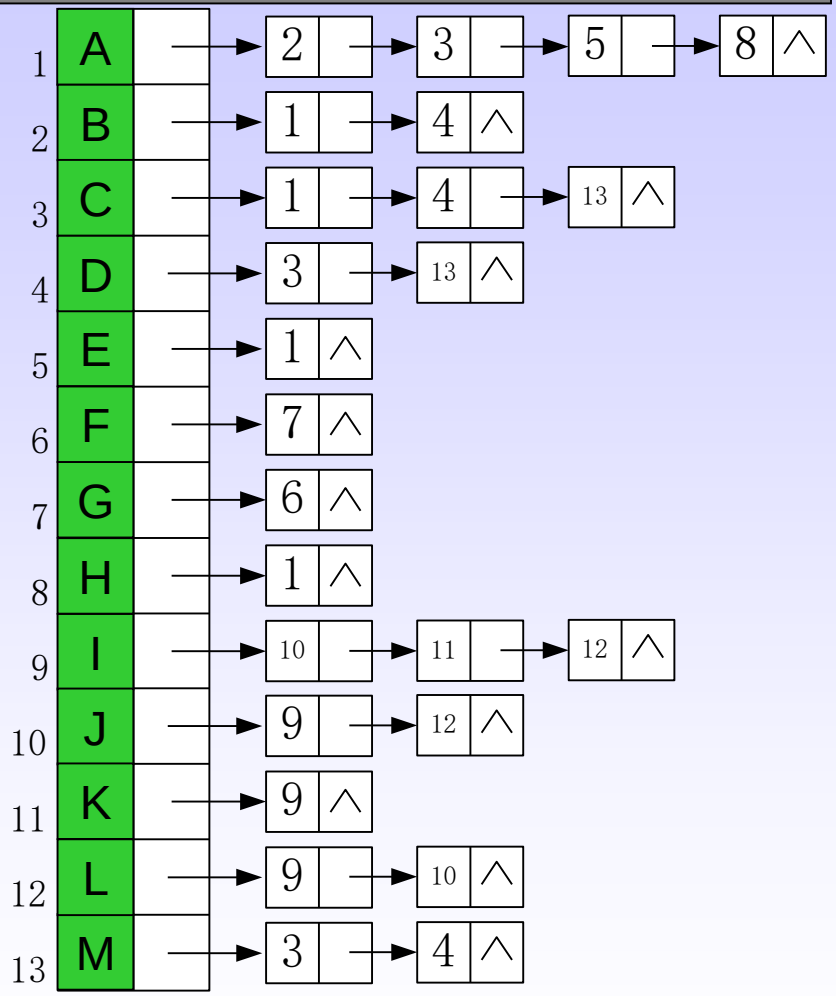


7.3 图的遍历

- 深度优先搜索（非连通图）：







遍历序列：

A，B，D，C，M，E，H，　F，G，　I，J，L，K

7.3 图的遍历

[总结]:

DFS 在访问图中某一起始顶点 *v* 后，由 *v* 出发，访问它的任一邻接顶点 w_1；再从 w_1 出发，访问与 w_1 邻 接但还没有访问过的顶点 w_2；然后再从 w_2 出发，进行类似的访问，… 如此进行下去，直至到达所有的邻接顶点都被访问过的顶点 *u* 为止。接着，退回一步，退到前一次刚访问过的顶点，看是否还有其它没有被访问的邻接顶点。如果有，则访问此顶点，之后再从此顶点出发，进行与前述类似的访问；如果没有，就再退回一步进行搜索。重复上述过程，直到连通图中所有顶点都被访问过为止。

7.3 图的遍历

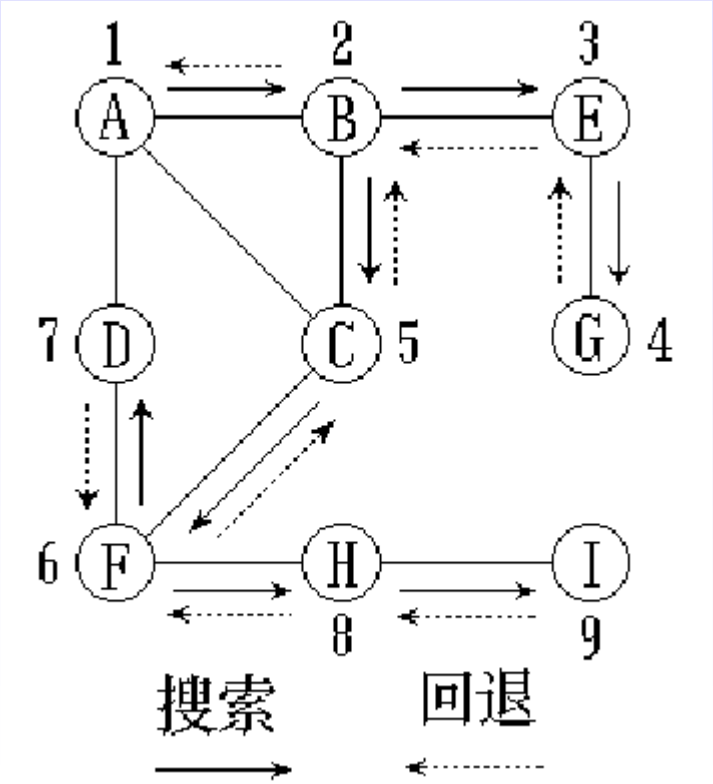
- 深度优先搜索的实现

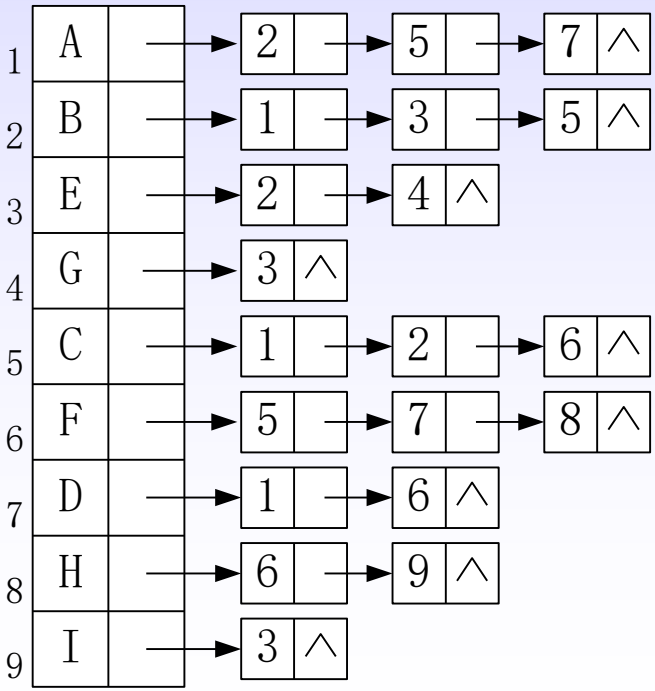
变量说明：**Boolean visited[MAX]；**//用于标识结点是否已被访问过

```
void DFSTraverse( Graph G )
{  for ( k=1; k<=G.vexnum; ++k )  visited[k] = FALSE;
    for ( k=1; k<=G.vexnum; ++k )
          if ( !visited[ k ] )     DFS(G, k);
}
void DFS( Graph G,  int v);
{  visited[v] = TRUE;
    VISIT( v );  // 访问图G中第v个顶点
    for (w=FirstAdjVex(G, v); w; w=NextAdjVex(G, v, w))
         if ( !visited[ w ] )     DFS(G, w);
}
```

7.3 图的遍历

```
int FirstAdjVex(ALGraph G, int v)
{  // 返回G中第v个顶点的第1个邻接点的序号。如果v无邻接点，返回0。
   if(!G.vertices[v].firstarc)          return(0);
   else return(G.vertices[v].firstarc->adjvex);
}
```





7.3 图的遍历

int NextAdjVex(ALGraph G, int v, int w) // 返回G中第v个顶点的相对于顶点w的下一个邻接点的序号。如果v无相对于顶点w的下一个邻接点，返回0。

```
{  ArcNode *p;
   p=G.vetrices[v].firstarc;
   while( p && p->adjvex!=w)     p=p->nextarc;
   if(p->adjvex==w && p->nextarc)     return(p->nextarc->adjvex);
   else return(0);
}
```

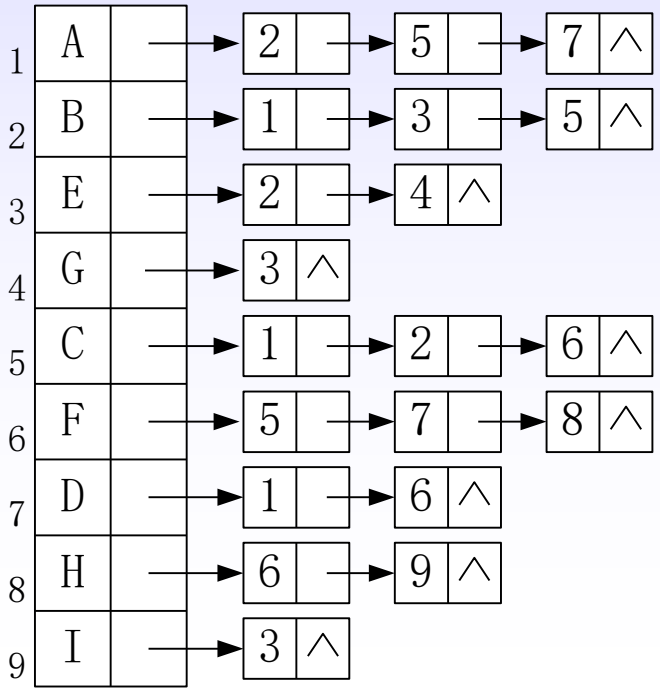
例：右边图中

NextAdjVex(G, 1, 2)=5;

NextAdjVex(G, 6, 7)=8;

NextAdjVex(G, 1, 4)=0;

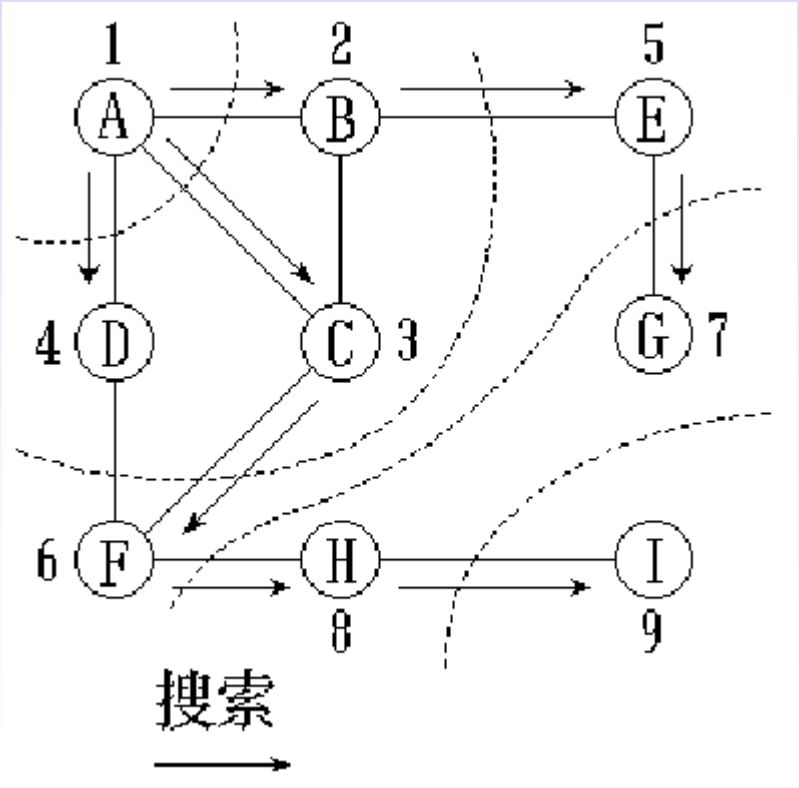
NextAdjVex(G, 1, 7)=0;

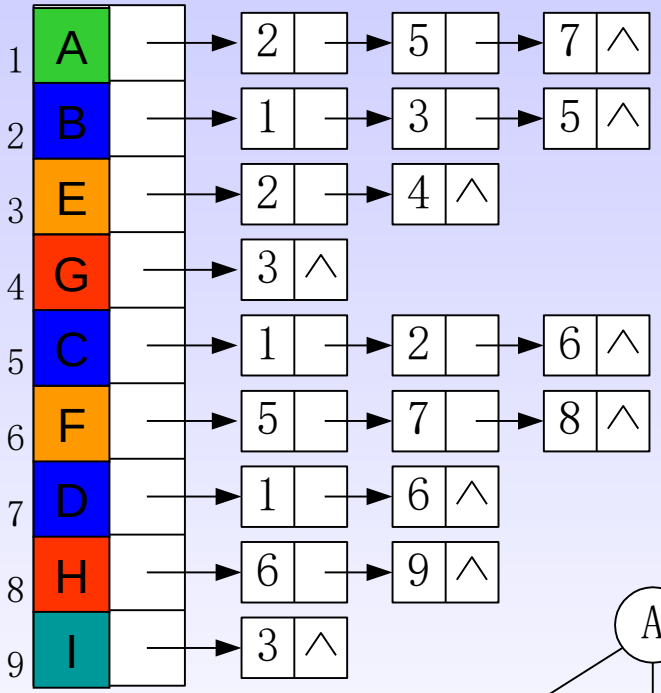


7.3 图的遍历

7.3.2 广度优先搜索

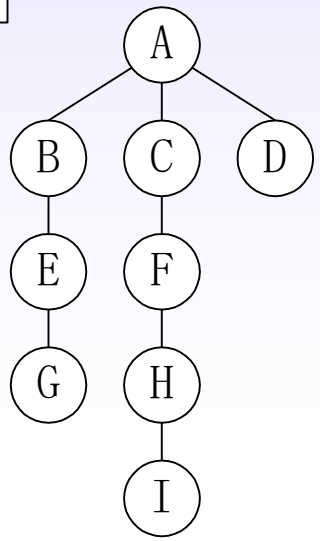
- 广度优先搜索（连通图）：





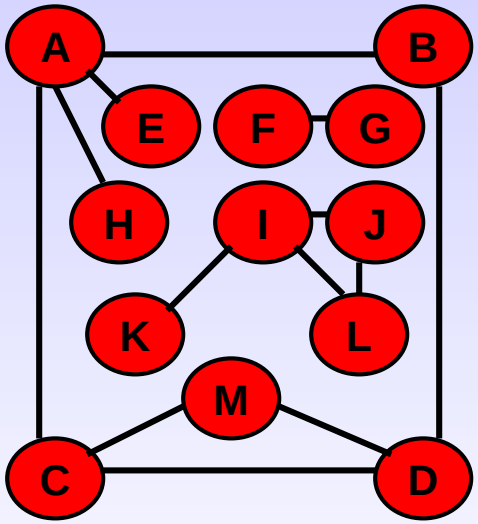
遍历序列：

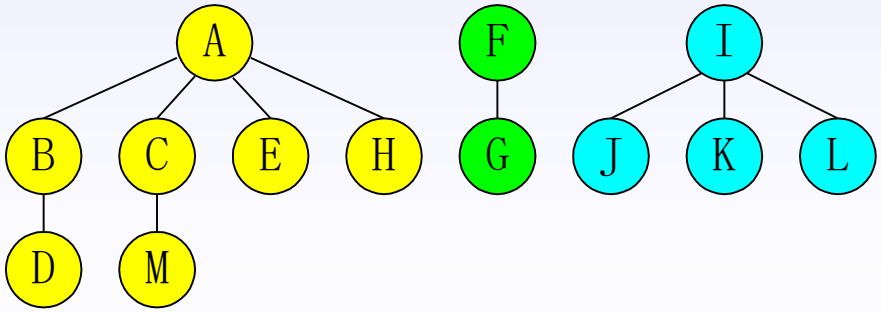
A，B，C，D，E，F，G，H，I

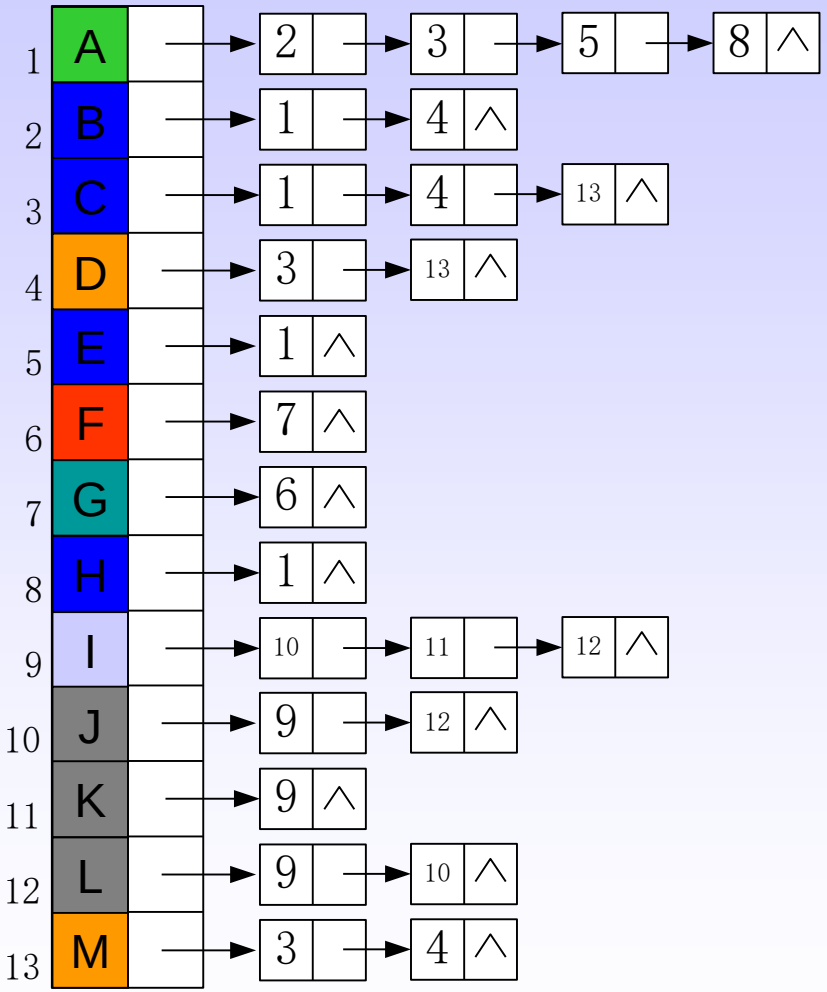


7.3 图的遍历

- 广度优先搜索（非连通图）：







遍历序列：A，B，C，E，H，D，M，　　F，G，　　I，J，K，L

7.3 图的遍历

[总结]：

- 广度优先搜索在访问了起始顶点 v 之后，由 v 出发，依次访问 v 的各个未被访问过的邻接顶点 $w_1, w_2, \ldots, w_t$，然后再顺序访问 $w_1, w_2, \ldots, w_t$ 的所有还未被访问过的邻接顶点。再从这些访问过的顶点出发，访问它们的所有未被访问过的邻接顶点，… 如此做下去，直到图中所有顶点都被访问到为止。
- 广度优先搜索是一种分层的搜索过程，每向前走一步可能访问一批顶点，不像深度优先搜索有往回退的情况。因此，广度优先搜索不是一个递归的过程，其算法也不是递归的。
- 为了实现逐层访问，算法中使用了一个队列，以记忆正在访问的这一层和上一层的顶点，以便于向下一层访问。

广度优先搜索的实现

变量说明：**Boolean visited[MAX]**；//用于标识结点是否已被访问过

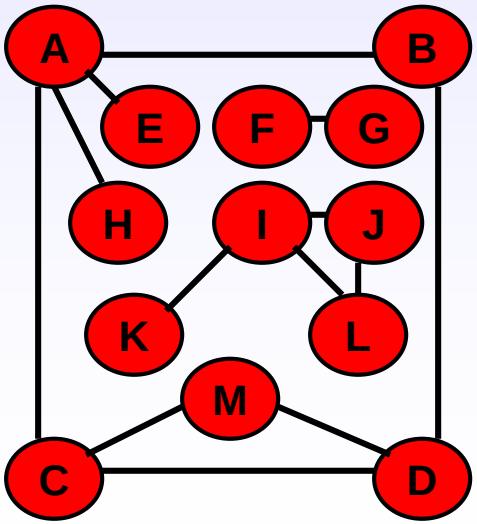
```
void BFSTraverse( Graph G )
{ for ( v=1; v<=G.vexnum; ++v )  visited[v] = FALSE;
   InitQueue(Q);
   for( v=1; v<=G.vexnum; ++v )
      if ( !visited[v] )
      {  visited[v]=TRUE;     VISIT(v);        EnQueue(Q, v);
         while (!EmptyQueue(Q))
         {  DeQueue(Q,u);
            for(w=FirstAdjVex(G, u); w; w=NextAdjVex(G, u, w))
               if( !visited[w] )
               {  visited[w]=TRUE; VISIT(w); EnQueue(Q, w); }
         } // end while
      } // end if
}
```

7.4 图的连通性问题

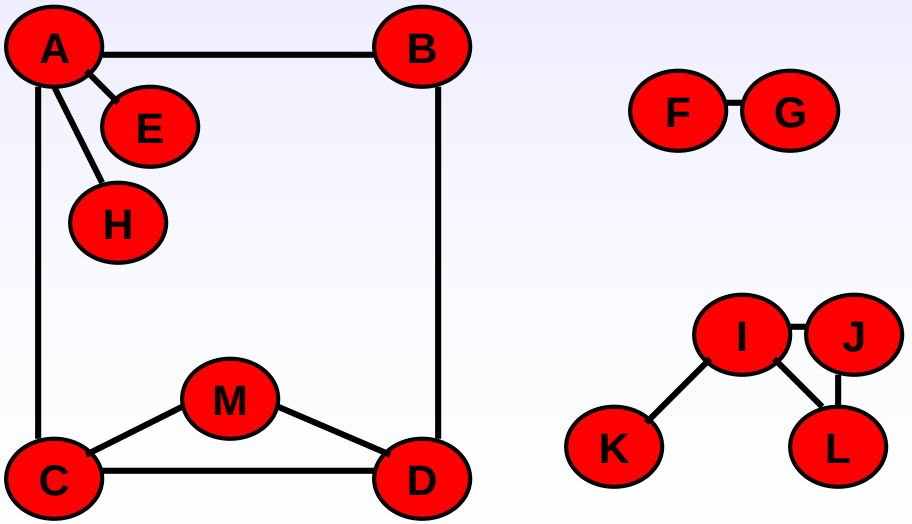
7.4.1 无向图的连通分量和生成树

- 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

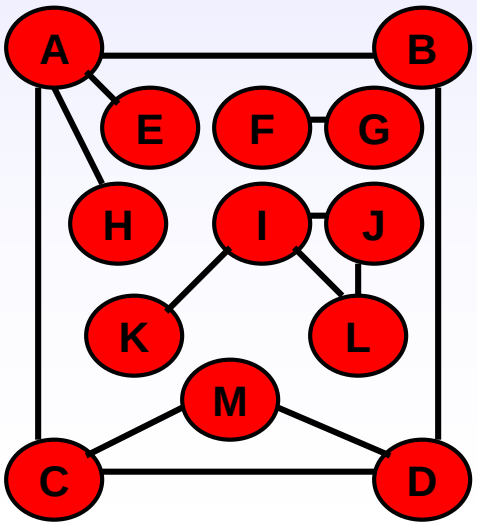


7.4 图的连通性问题

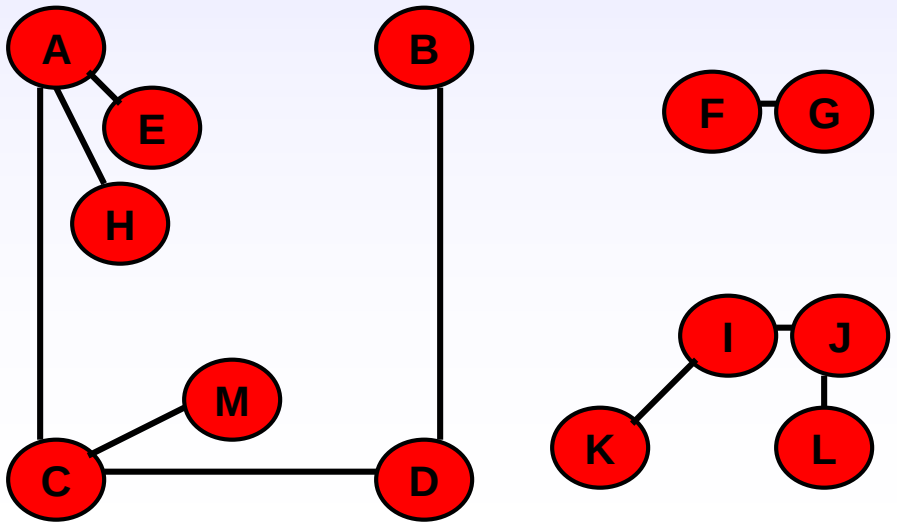
•生成树：极小连通子图。包含图的所有 n 个结点，但只含图的 n-1 条边。在生成树中添加一条边之后，必定会形成回路或环。

•生成森林：非连通图的每个连通分量都可以构造一棵生成树，这些生成树组成了非连通图的生成森林。

无向图G



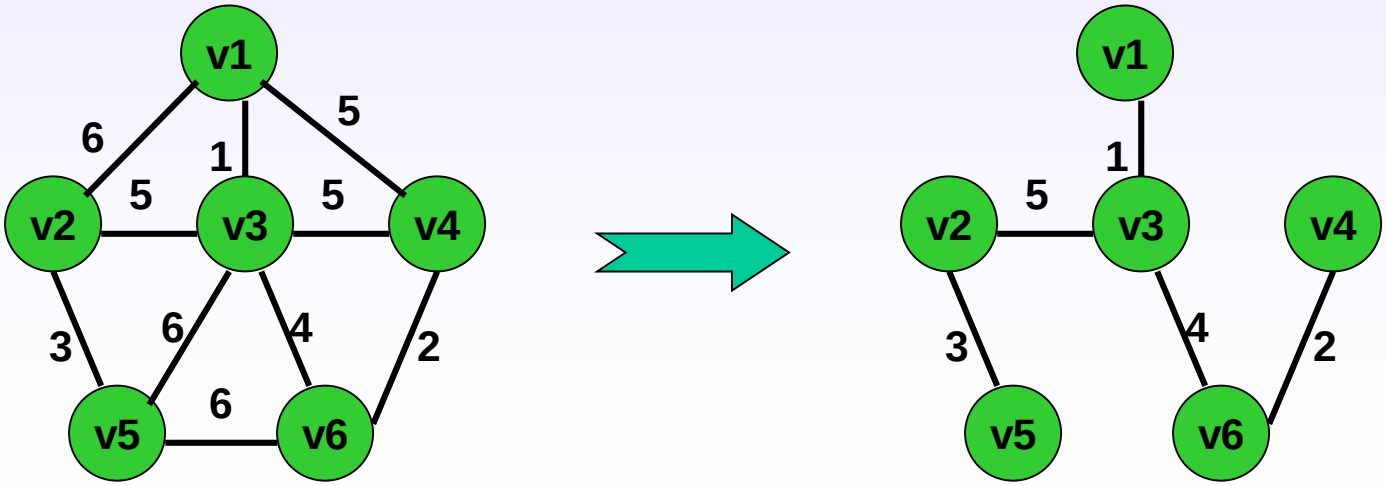
无向图G的生成森林



7.4 图的连通性问题

7.4.2 最小代价生成树

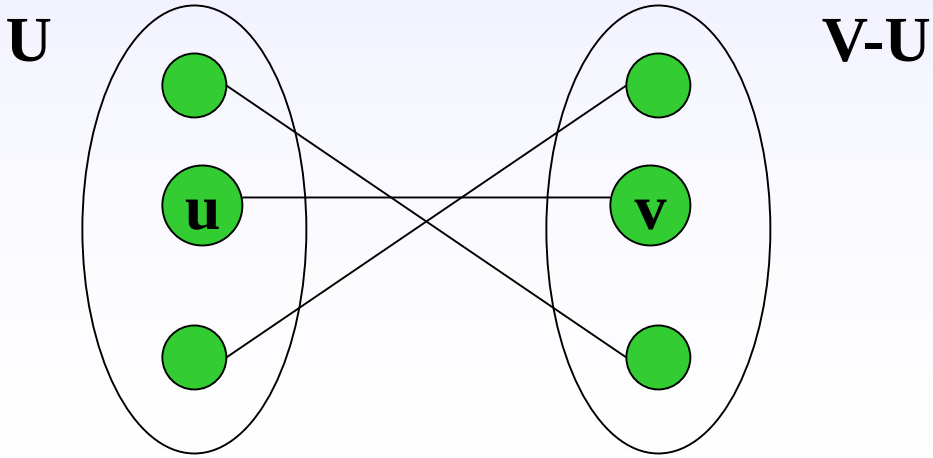
- 最小代价生成树：对于网络，其生成树中的边也带权，将生成树各边的权值总和称为生成树的权，并把权值最小的生成树称为最小生成树(Minimum Spanning Tree)。
- 实例：



7.4 图的连通性问题

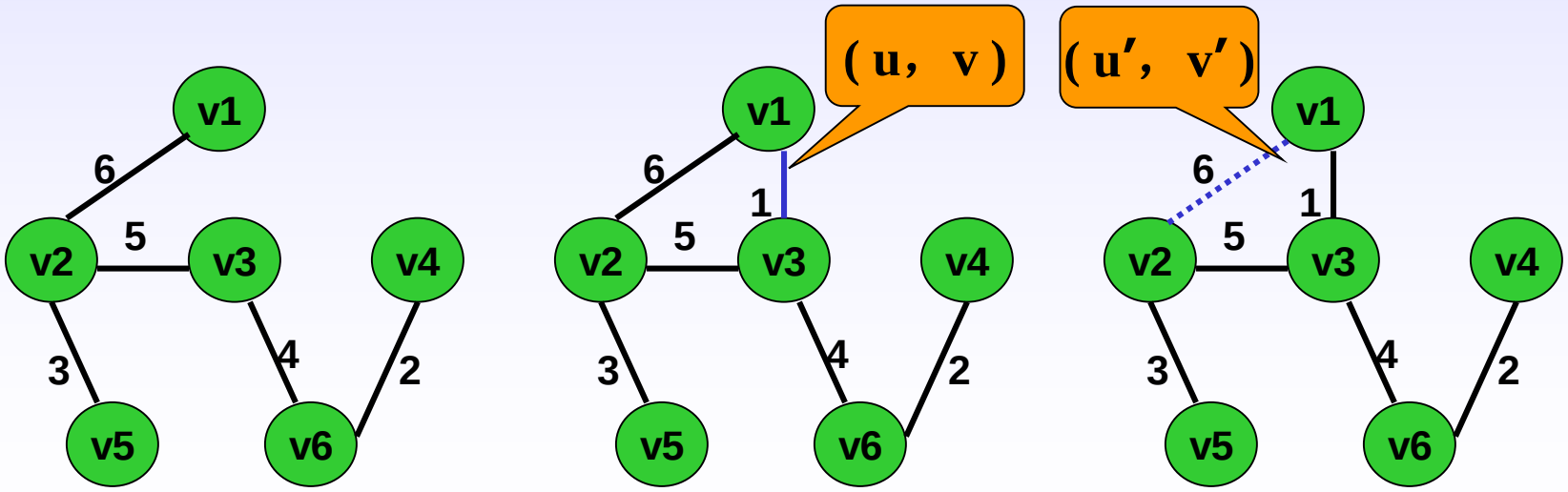
- MST 性质：

假设 G = {V, { E } } 是一个连通网，U 是顶点集合 V 的一个非空子集。若 (u, v) 是U与V-U之间一条权值最小的边，即 u 属于 U , v 属于 V - U，则必存在一棵包括边 (u, v) 在内的最小代价生成树。



7.4 图的连通性问题

证明：假定存在一棵不包括边 (u, v) 在内的最小代价生成树，设其为 T 。将边(u, v) 添加到树 T ，则形成一条包含 (u, v) 的回路。因此，必定存在另一 条边 (u′，v′) ，且 u′ 属于 U , v′属于 V - U。删去边 (u′，v′) ，得到另一棵生成 树 T′; 因为边 (u, v) 的代价小于边(u′，v′) 的代价，所以新的生成树T′将是代价最小的树。和原假设矛盾。



(a)最小生成树T　　(b)加入边(v1,v3)　　(c)最小生成树T′

7.4 图的连通性问题

1 Prim算法

- 基本思想：

从连通网 N = { V, E }中的某一顶点 u_0 出发，选择与它关联的具有最小权值的边(u_0, v)加入集合T，将其顶点加入到生成树的顶点集合U中。

以后每一步从一个顶点在U中，而另一个顶点不在U中的各条边中选择权值最小的边(u, v)加入集合T,把它的顶点加入到集合U中。如此继续下去，直到网中的所有顶点都加入到生成树顶点集合U中为止。

此时，最小生成树={U，T}。

7.4 图的连通性问题

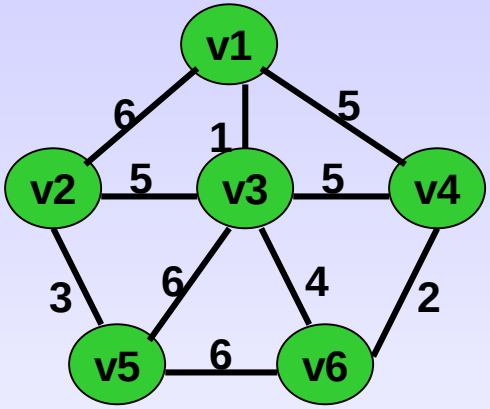
Prim 算法的基本描述

设G＝(V，E)是个带权无向连通图，U是最小生成树的顶点集合，T是最小生成树的边集合：

```
{ 从G中任选u∈V;
  U＝{u}; T = φ;
  while ( U != V )
  { 在u∈U, v∈V-U中找一条权值最小的边(u,v);
    U = U + {v};
    T = T + (u,v);
  }
}
```

7.4 图的连通性问题

- **Prim 算法的实例**



辅助数组closedge的定义：

```
struct{
        VertexType  adjvex;
        VRType      lowcost;
} closedge[ MAX ];
```

对于每个$v_i \in V-U$：

- closedge [i].lowcost = Min{ cost(u,vi) | u∈U }，表示v_i与U中顶点之间存在的一条权值最小的边的权值；
- closedge[i].adjvex表示U中与v_i之间存在权值最小的边的顶点的序号。

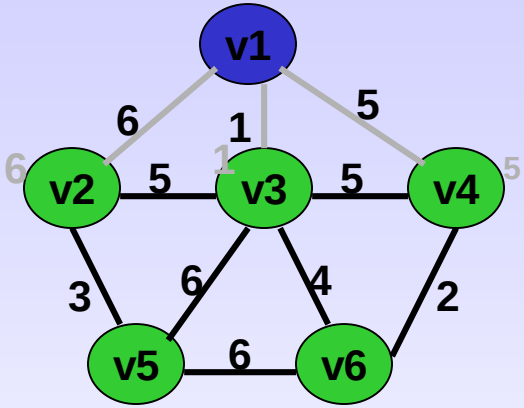
7.4 图的连通性问题

- Prim 算法的实例

（1）选择v1加入U，设置closedge数组；

U: {v1}

T: {φ}



	1	2	3	4	5	6
closedge[ ]						

adjvex

lowcost

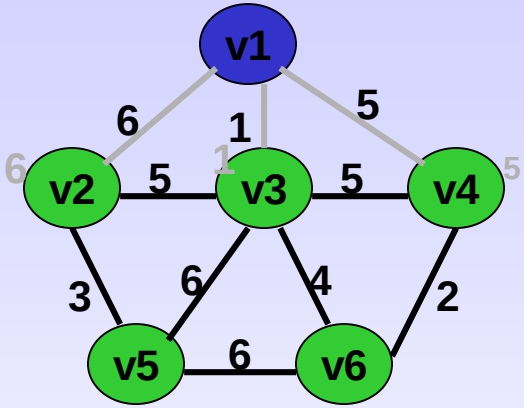
7.4 图的连通性问题

- Prim 算法的实例

（1）选择v1加入U，设置closedge数组；

U: {v1}

T: {φ}



	1	2	3	4	5	6	
closedge[ ]		v1	v1	v1			adjvex
	0	6	1	5	∞	∞	lowcost

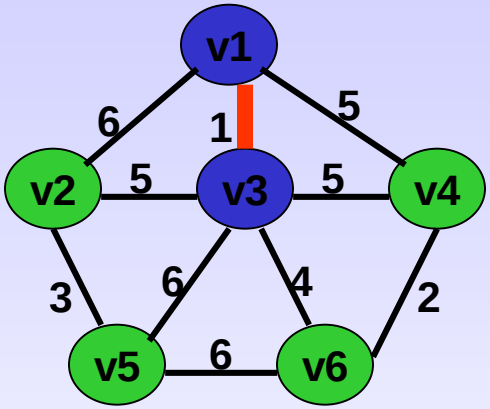
7.4 图的连通性问题

（2）选择v3加入U，(v1,v3)

加入T；

U: {v1, v3}

T: {(v1,v3)}



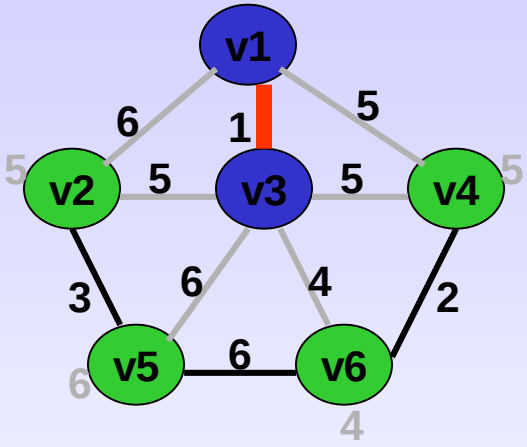
	1	2	3	4	5	6	
closedge[ ]		v1		v1			adjvex
	0	6	0	5	∞	∞	lowcost

7.4 图的连通性问题

（3）调整closedge数组；

U: {v1, v3}

T: {(v1,v3)}



	1	2	3	4	5	6	
closedge[ ]		v3		v1	v3	v3	adjvex
	0	5	0	5	6	4	lowcost

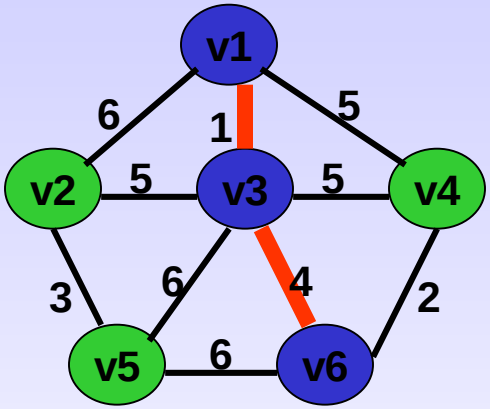
7.4 图的连通性问题

（4）选择v6加入U，(v3,v6)

加入T；

U: {v1, v3, v6}

T: {(v1,v3),(v3,v6)}



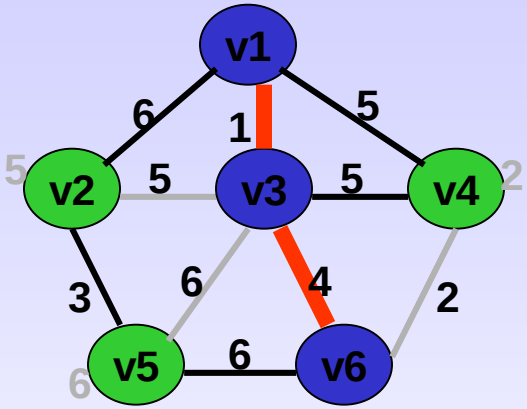
	1	2	3	4	5	6	
closedge[ ]		v3		v1	v3		adjvex
	0	5	0	5	6	0	lowcost

7.4 图的连通性问题

（5）调整closedge数组；

U: {v1, v3, v6}

T: {(v1,v3),(v3,v6)}



	1	2	3	4	5	6	
closedge[ ]		v3		v6	v3		adjvex
	0	5	0	2	6	0	lowcost

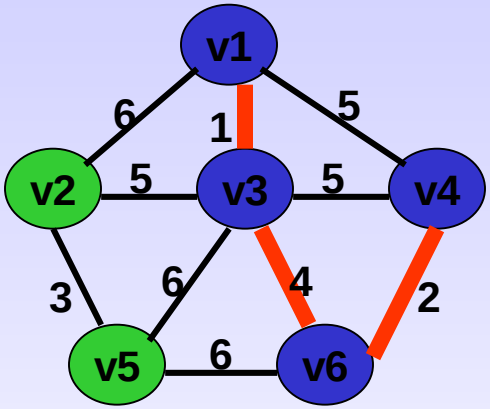
7.4 图的连通性问题

（6）选择v4加入U，(v6,v4)

加入T；

U: {v1, v3, v6, v4}

T: {(v1,v3),(v3,v6),(v6,v4)}



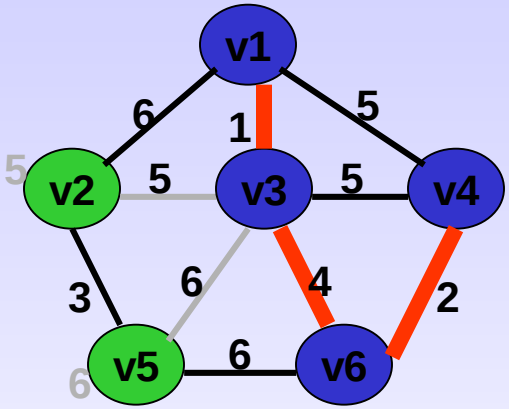
	1	2	3	4	5	6	
closedge[ ]		v3			v3		adjvex
	0	5	0	0	6	0	lowcost

7.4 图的连通性问题

（7）调整closedge数组；

U: {v1, v3, v6, v4}

T: {(v1,v3),(v3,v6),(v6,v4)}



	1	2	3	4	5	6	
closedge[ ]		v3			v3		adjvex
	0	5	0	0	6	0	lowcost

7.4 图的连通性问题

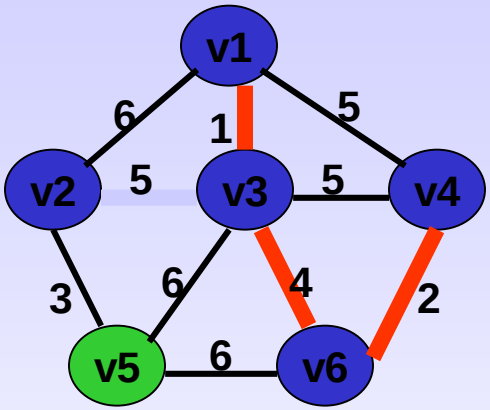
（8）选择v2加入U，(v3,v2)

加入T;

U: {v1, v3, v6, v4, v2}

T: {(v1,v3),(v3,v6),(v6,v4),

(v3,v2)}



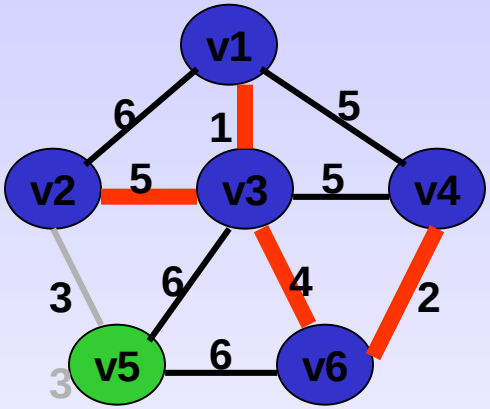
	1	2	3	4	5	6	
closedge[ ]					v3		adjvex
	0	0	0	0	6	0	lowcost

7.4 图的连通性问题

（9）调整closedge数组；

U: {v1, v3, v6, v4, v2}

T: {(v1,v3),(v3,v6),(v6,v4),
(v3,v2)}



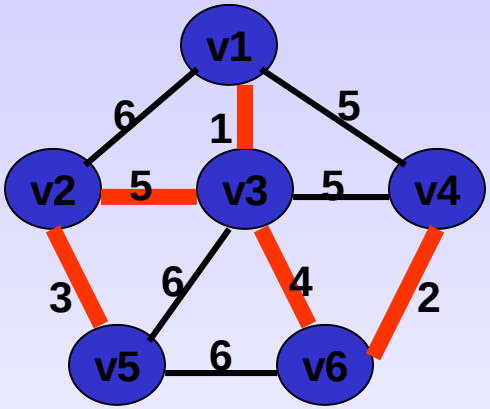
	1	2	3	4	5	6	
closedge[ ]					v2		adjvex
	0	0	0	0	3	0	lowcost

7.4 图的连通性问题

（10）v5加入U，(v2,v5)加入T，结束；

U: {v1, v3, v6, v4, v2, v5}

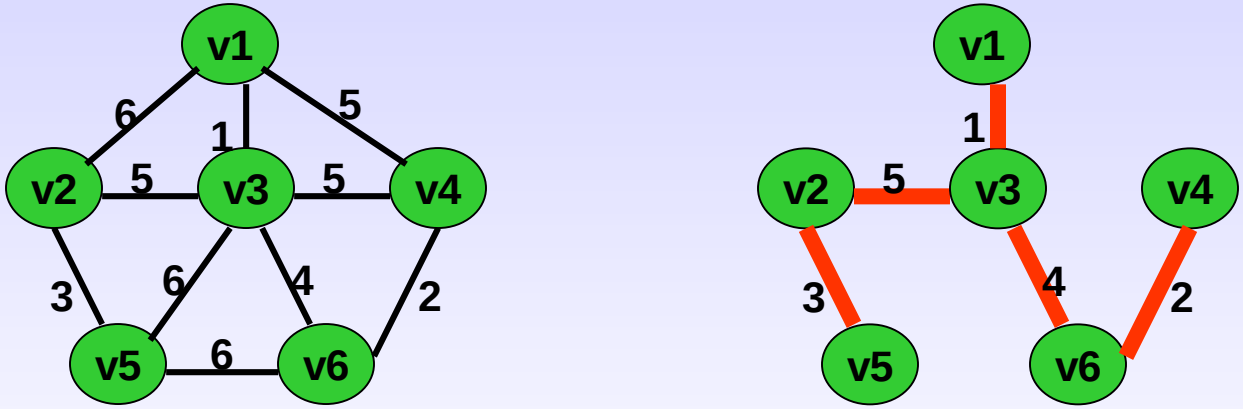
T: {(v1,v3),(v3,v6),(v6,v4),(v3,v2),(v2,v5)}



	1	2	3	4	5	6	
closedge[ ]							adjvex
	0	0	0	0	0	0	lowcost

7.4 图的连通性问题

- Prim 算法的实例



要点：每当新的顶点并入 U 之后，则调整仍在V-U集合中的顶点至 U 中顶点的最小距离。