

第八章 图

张史梁

slzhang.jdl@pku.edu.cn

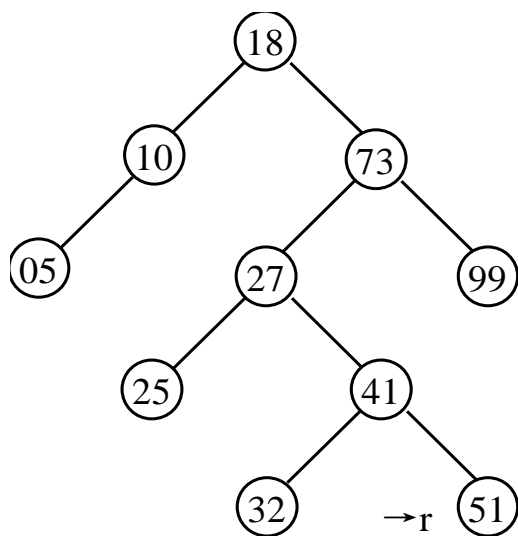
xuruihan@pku.edu.cn

字典与检索

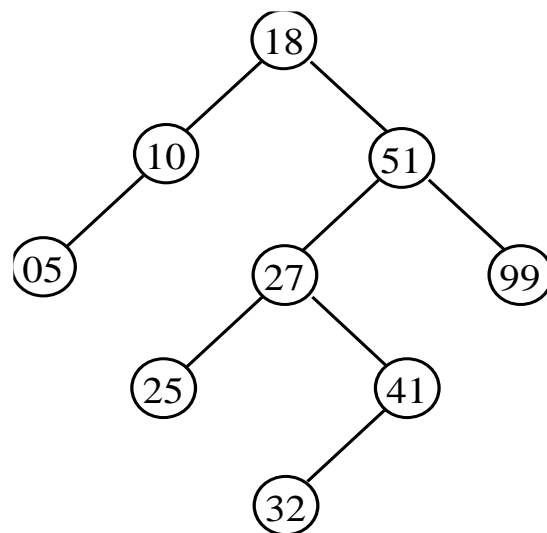
- 基本概念
- 字典的顺序表表示
- 字典的散列表示
 - 散列表
 - 散列函数
 - 存储表示与碰撞的处理
 - 开地址法（线性探查、双散列函数）
 - 拉链法
- 字典的索引和树型表示
 - 二叉排序树
 - AVL树

二叉排序树删除节点： 左子树中最大者【方法1】

- ① 若被删除结点 p 没有左子树，则用 p 的右子女代替 p 即可；
- ② 否则，在 p 的左子树中，中序遍历找出最后一个结点 r ，将 r 删除 (r 一定无右子女，用 r 的左子女代替 r 即可)
- ③ 用 r 节点代替被删除的节点 p



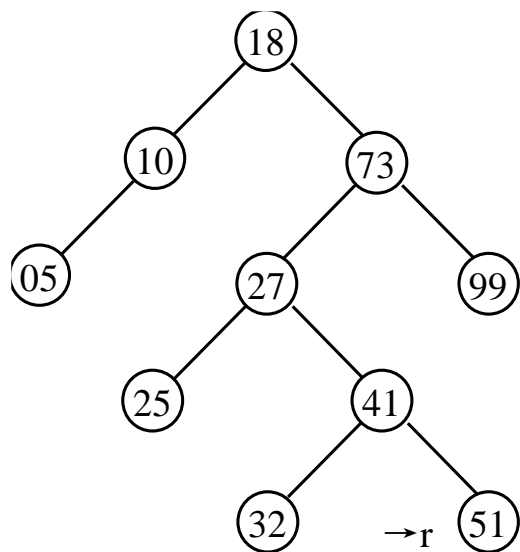
删除73



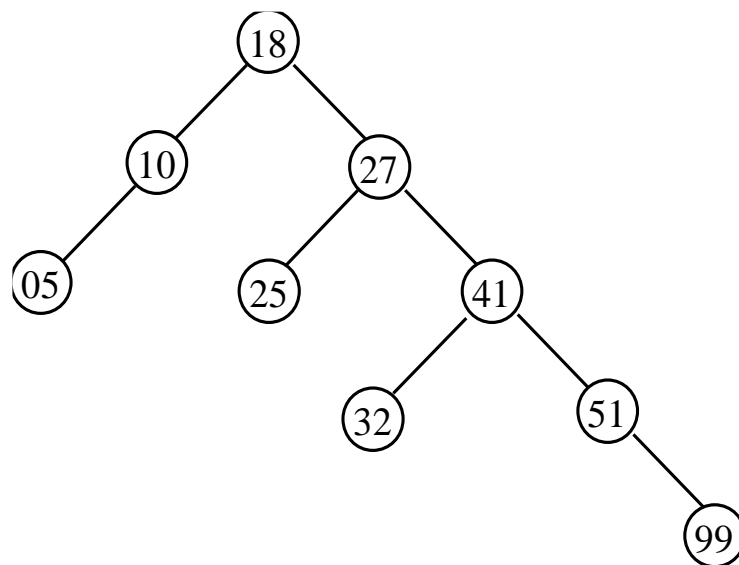
因为 r 在对称序列中紧排在 p 结点之前，所以用 r 代替 p 结点不会改变二叉排序树的性质

二叉排序树删除节点： 左替右换【方法2】

- ① 若被删除结点p没有左子树，用p的右子女代替p即可；
- ② 否则，在p的左子树中，中序周游找出最右一个结点r（r一定无右子女）将r的右指针指向p的右子女；
- ③ 用p的左子女代节点代替结点p。



删除73



AVL树的平衡调整

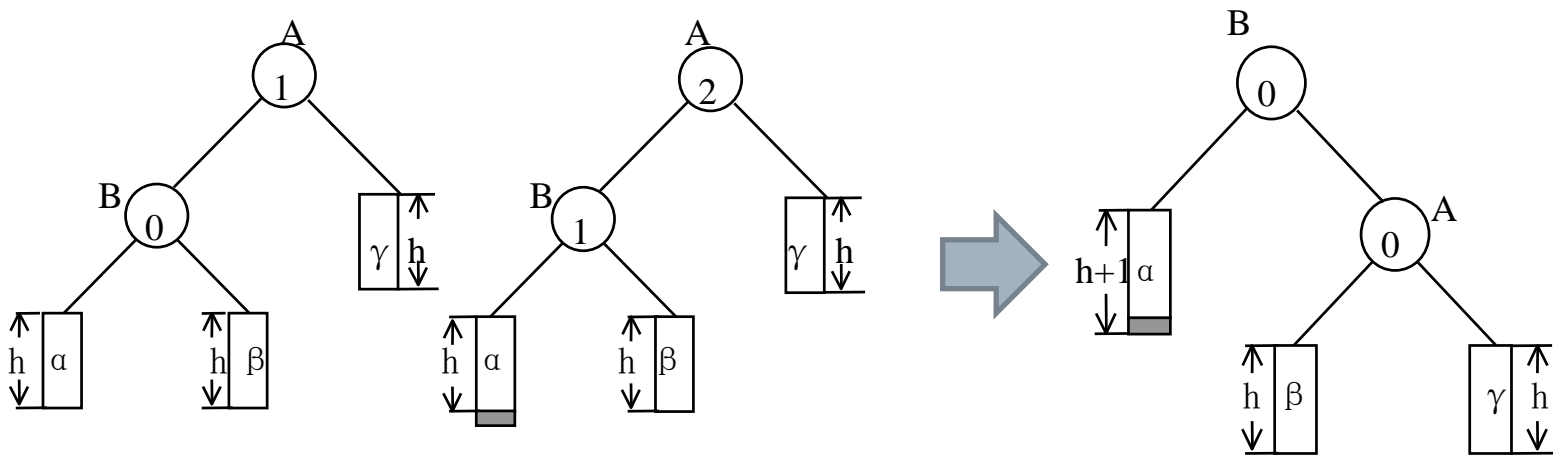
- 假设最小不平衡子树的根结点为A，调整子树的操作可归纳为以下四种情况
 - LL型调整
 - LR型调整
 - RL型调整
 - RR型调整

LL型调整

□ 调整规则

- 将A的左子女B提升为新二叉树的根
- 原来的根A连同其右子树 γ 向右下旋转成为B的右子树
- B的原右子树 β 作为A的左子树。

□ 结合律： $(\alpha B \beta) A (\gamma) = (\alpha) B (\beta A \gamma)$

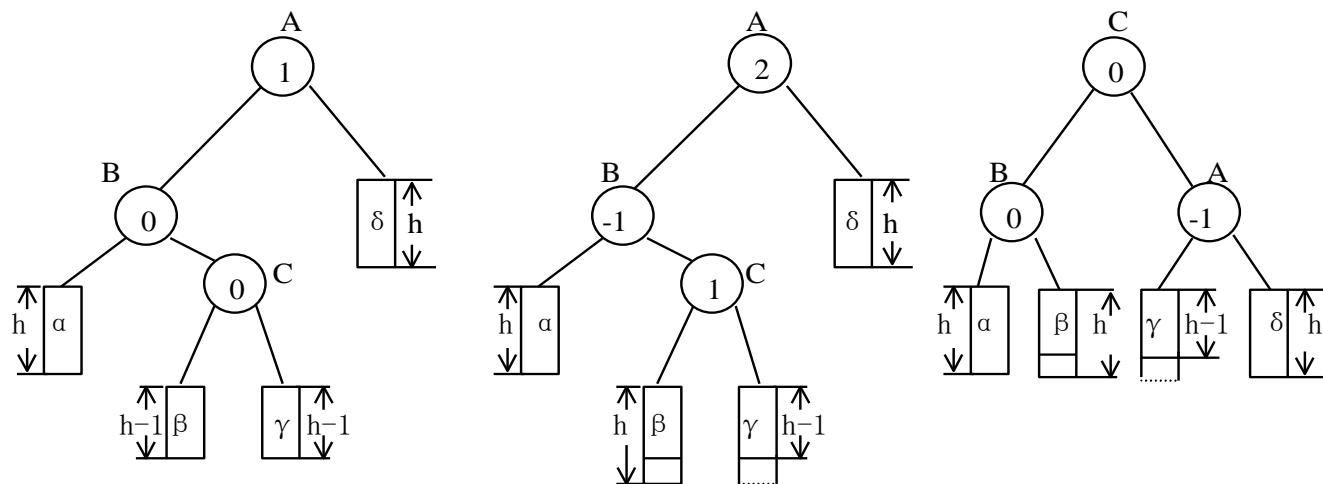


LR型调整

□ 调整规则：

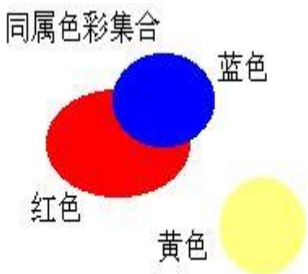
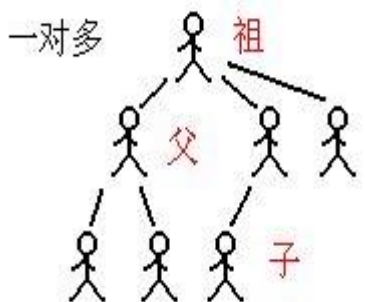
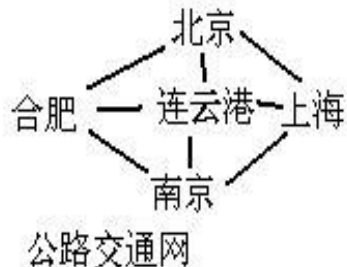
- 设C为A的左子女的右子女，将A的孙子结点C提升为新二叉树的根；
- 原C的父结点B连同其左子树 α 向左下旋转成为新根C的左子树，原C的左子树 β 成为B的右子树
- 原根A连同其右子树 δ 向右下旋转成为新根C的右子树，原C的右子树 γ 成为A的左子树

□ 结合律： $((\alpha)B(\beta C\gamma))A(\delta) = (\alpha B\beta)C(\gamma A\delta)$



逻辑结构

- 线性结构：唯一前驱，唯一后继，线性关系
- 树形结构：唯一前驱，多个后继，层次关系
- 图形结构：多对多、任意，网状关系

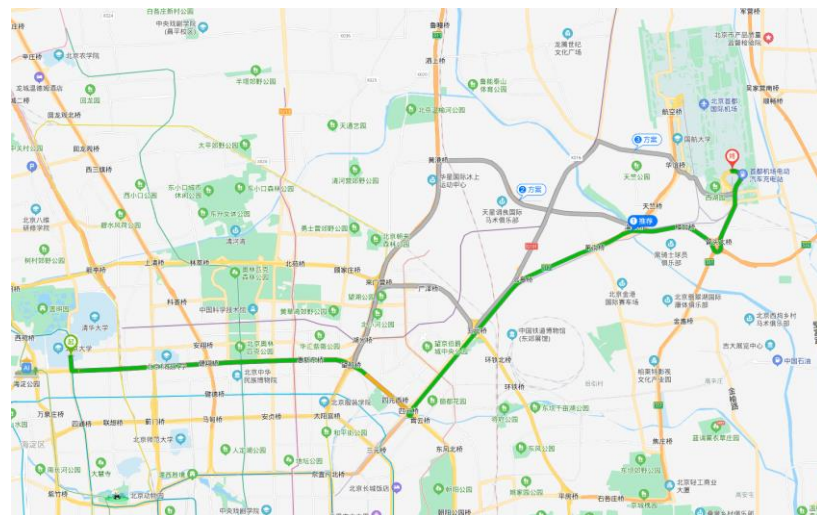
	集合	线性结构	树形结构	图状或网状结构
特征	元素间为松散的关系	元素间为严格的一对一关系	元素间为严格的一对多关系	元素间为多对多关系
示例	<p>同属色彩集合</p>  <p>蓝色 红色 黄色</p>	<p>如银行排队叫号时队列中的各元素</p>	<p>一对多</p> 	<p>多对多</p>  <p>公路交通网</p>

图

- 教学目的：介绍图的基本概念、两种常用的存储结构、两种遍历算法以及图的基本应用。
- 教学重点：掌握在图的两种存储结构上实现的遍历算法。
- 教学难点：图的应用算法，求最短路径，求关键路径以及拓扑排序等。

内容提要

- 图的基本概念
- 存储表示
- 图的基本运算与周游
- 最小生成树
- 拓扑排序
- 关键路径
- 最短路径



图的基本概念

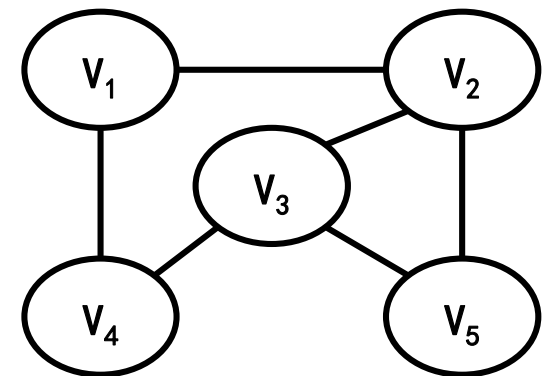
□ 图的定义：

- 由顶点的有穷非空集合 V 和顶点的偶对(边)集合 E 组成，记为 $G=(V, E)$,
- V 是结点(顶点)的有穷集合
- E 是边的集合，是结点的偶对

□ 图应用广泛：网络布设，航线管理，交通管理等。

无向图

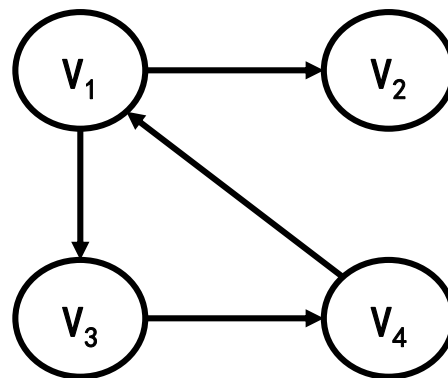
- 定义:若图中每条边都是无方向的, 则称为无向图。
- 表示:无向图中的边是由两个顶点组成的无序对, 无序对用圆括号表示
 - 无向图中 (v_i, v_j) 和 (v_j, v_i) 代表同一条边。
 - v_i 和 v_j 是相邻结点, (v_j, v_i) 是与顶点 v_j 和 v_i 相关联的边



无向图

有向图

- 定义：若图中每条边都是有方向的，则称为有向图。
- 表示：有向图中的边是由两个顶点组成的有序对，有序对用尖括号表示
 - 如 $\langle v_i, v_j \rangle$ 表示一条有向边， v_i 是边的始点， v_j 是边的终点。
 - $\langle v_i, v_j \rangle$ 和 $\langle v_j, v_i \rangle$ 代表两条不同的有向边。
 - 边 $\langle v_i, v_j \rangle$ 与顶点 v_i, v_j 相关联



有向图

假设条件

□ 下面的讨论中，

- 不考虑顶点到其自身的边，即若 (v_i, v_j) 或 $\langle v_i, v_j \rangle$ 是 $E(G)$ 中的边，则 $v_i \neq v_j$
- 图中不允许一条边重复出现

例子

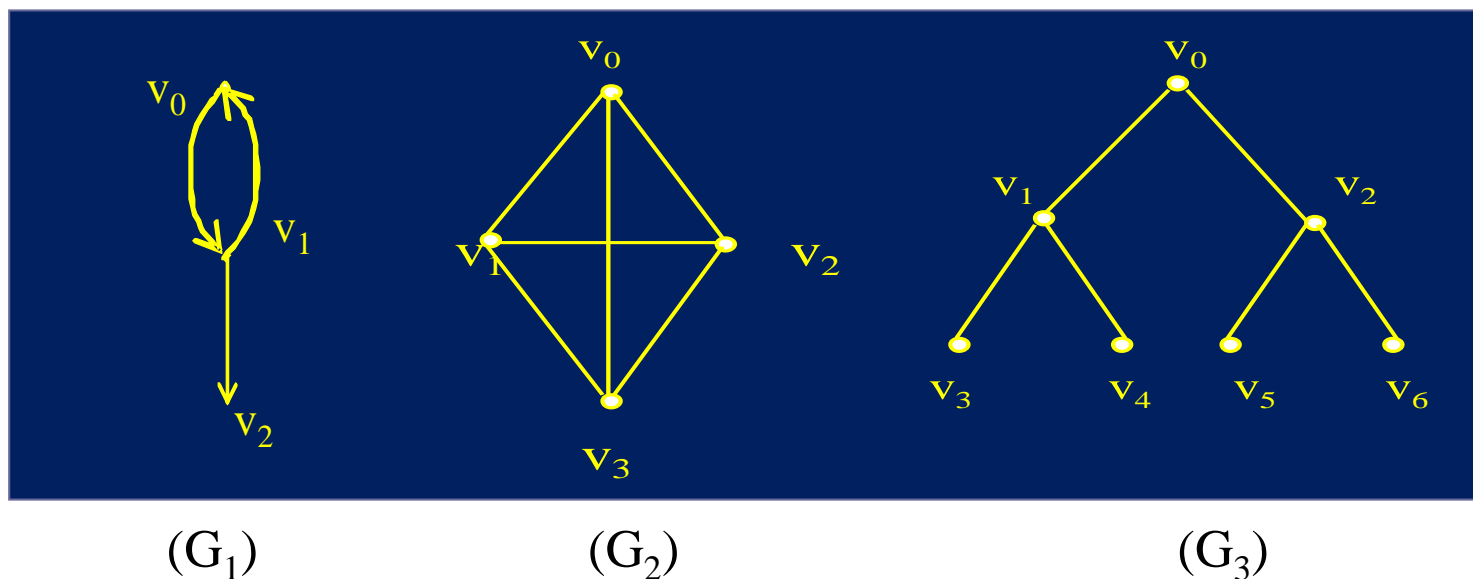
- G_1 为有向图，其顶点集合和边集合为：

$$V(G_1)=\{v_0, v_1, v_2\}, E(G_1)=\{<v_0, v_1>, <v_1, v_0>, <v_1, v_2>\}$$

- G_2 和 G_3 都是无向图

$$V(G_2)=\{v_0, v_1, v_2, v_3\}$$

$$E(G_2)=\{(v_0, v_1), (v_0, v_2), (v_0, v_3), (v_1, v_2), (v_1, v_3), (v_2, v_3)\}$$



概念1-完全图

□ 图G的顶点数 n 和边数 e 满足以下关系：

■ 若G是有向图，则 $0 \leq e \leq n(n-1)$

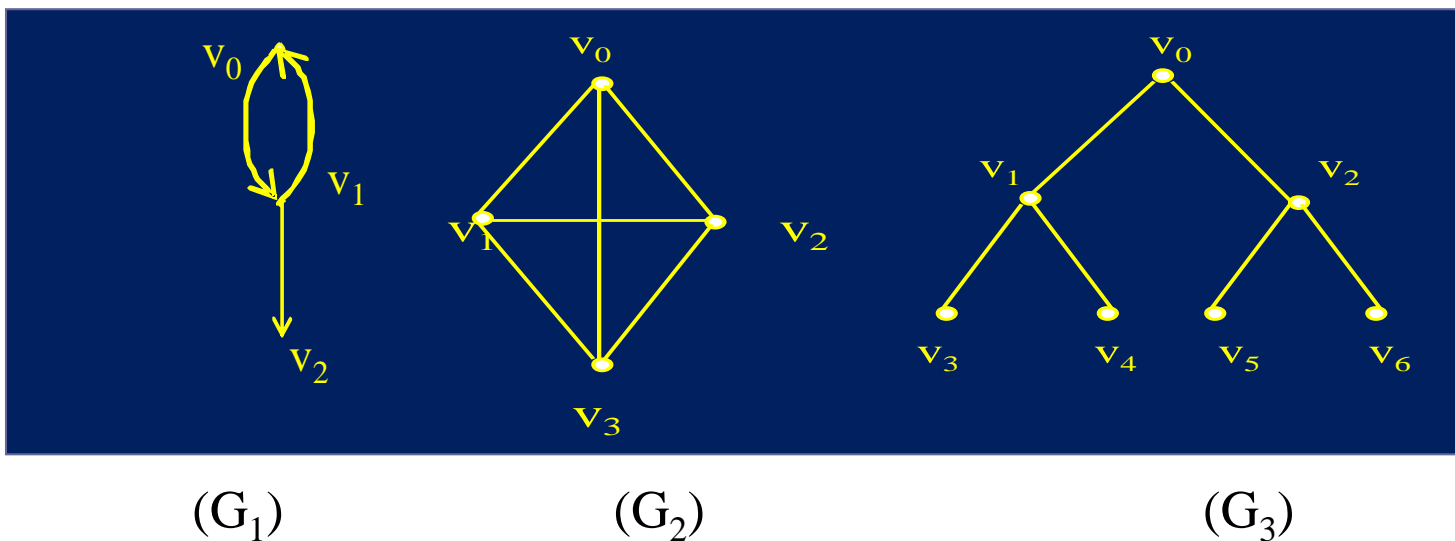
□ 有向完全图：有 $n(n-1)$ 条边的有向图

■ 若G是无向图，则 $0 \leq e \leq n(n-1)/2$

□ 无向完全图：有 $n(n-1)/2$ 条边的无向图

□ 在顶点个数相同的图中，完全图具有最多的边数

■ 如图G2就是一个具有4个顶点的无向完全图，边数为： $4 \times (4-1)/2 = 6$



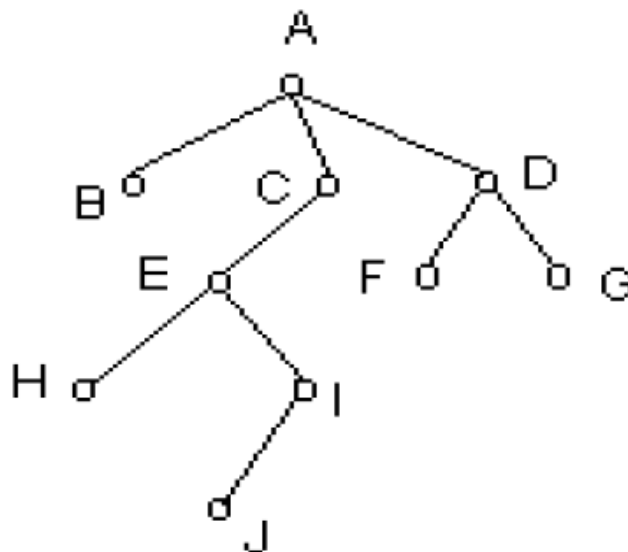
概念2-顶点的度

- **定义**：与顶点 v 相关联的边数称为顶点的度，记为 $D(v)$
- 有向图中，
 - **入度**：若 G 是一个有向图，则以顶点 v 为终点的边的数目称为 v 的入度，记为 $ID(v)$
 - **出度**：以 v 为始点的边的数目称为 v 的出度，记为 $OD(v)$
 - **顶点 v 的度**为其**入度和出度之和**，即 $D(v)=ID(v)+OD(v)$

树的基本术语

□ 结点的度数、树的度数

- 结点的子女个数叫作结点的“度数”。树中度数最大的结点的度数叫作“树的度数”
- 例如t中A, C, E, J的度数分别为3, 1, 2, 0; t的度数为3



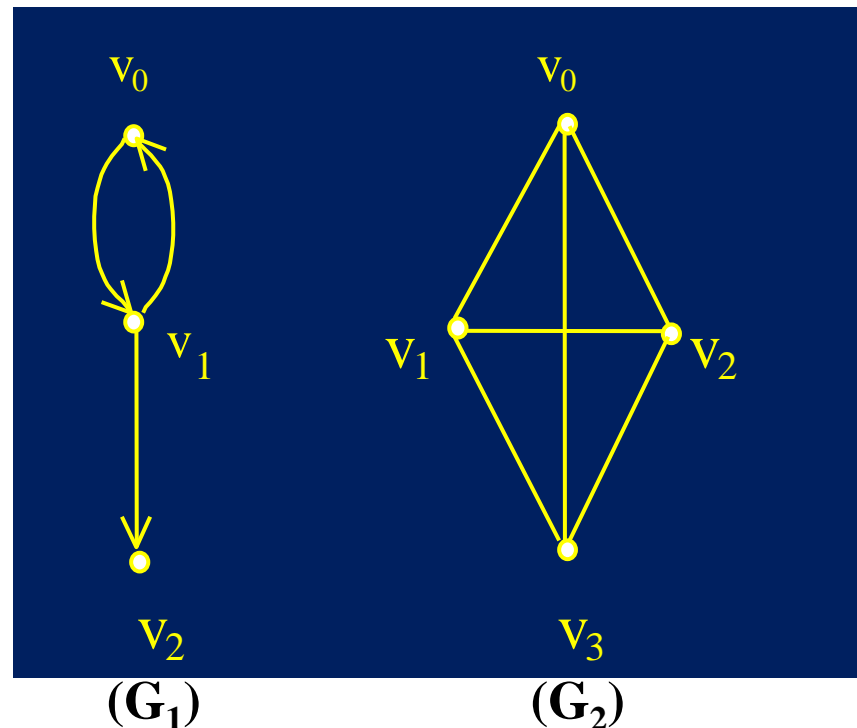
图的顶点、边数和度数之间的关系

- 如 G_2 中顶点 v_0 的度为3
- 如图 G_1 中顶点 v_1 的入度为1，出度为2，度为3
- 无论是有向图还是无向图，顶点数 n ，边数 e 和度数之间满足以下关系：

$$e = \sum_{i=1}^n D(v_i)/2$$

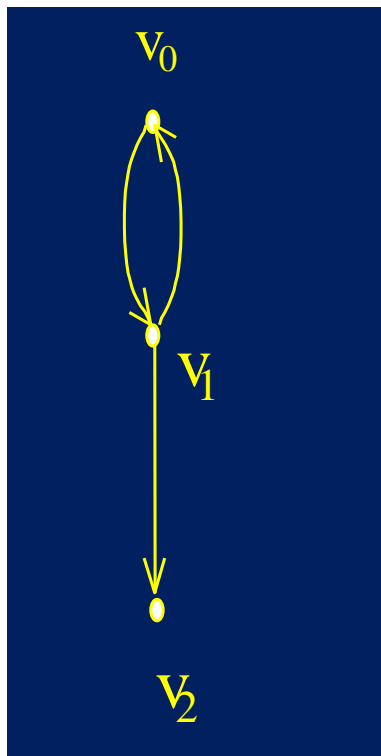
- 有向图 $\sum_{i=1}^n ID(v_i) = \sum_{i=1}^n OD(v_i)$

如： G_2 图中，边数为6，度数之和为 $(3+3+3+3)$

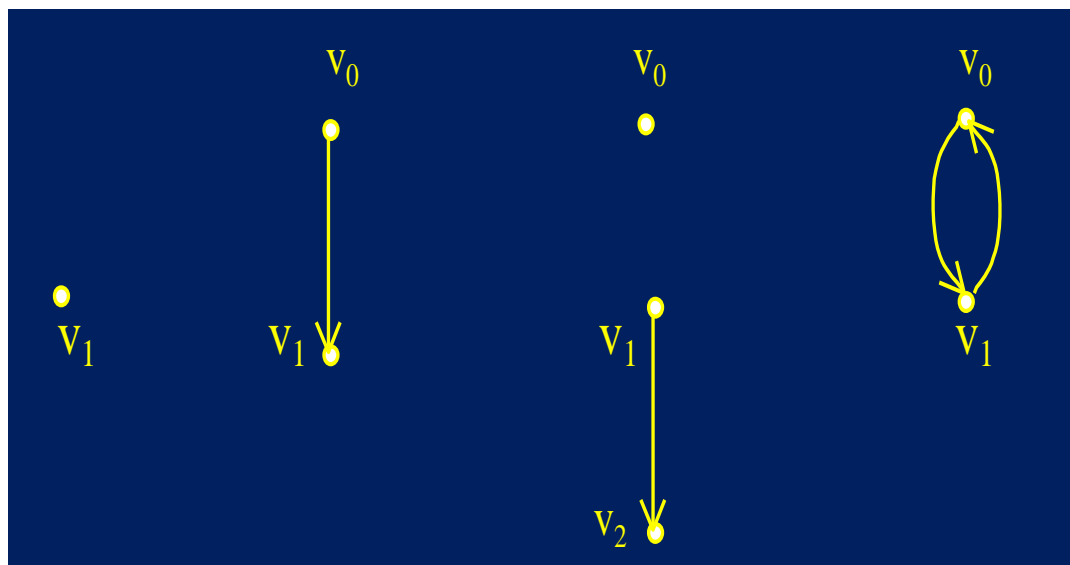


概念3-子图

- 定义：设有图 $G=(V, E)$ 和 $G'=(V', E')$ ，如果 V' 是 V 的子集， E' 是 E 的子集，则称 G' 是 G 的子图。
- 如下图给出了有向图 G_1 的若干子图。



(G_1)



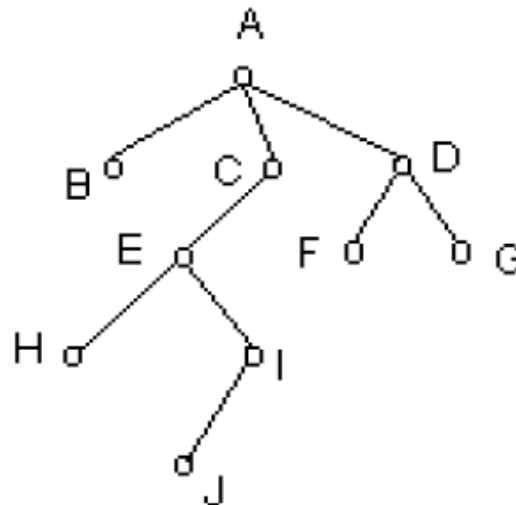
概念4-路径

- **定义**：图 $G=(V, E)$ 中，若存在顶点序列 $v_{i0}, v_{i1}, \dots, v_{in}$ ，使得 $(v_{i0}, v_{i1}), (v_{i1}, v_{i2}), \dots, (v_{in-1}, v_{in})$ 都在 E 中则称**从顶点 v_{i0} 到 v_{in} 存在一条路径**
 - 若 G 是有向图，则使得 $\langle v_{i0}, v_{i1} \rangle, \langle v_{i1}, v_{i2} \rangle, \dots, \langle v_{in-1}, v_{in} \rangle$ 都在 E 中
- **路径长度**：路径上的边数
- **简单路径**：若路径上的顶点除 v_{i0} 和 v_{in} **可以相同外**，其它顶点**都不相同**
- **回路或环**：起点和终点相同的**简单路径**

树的基本术语

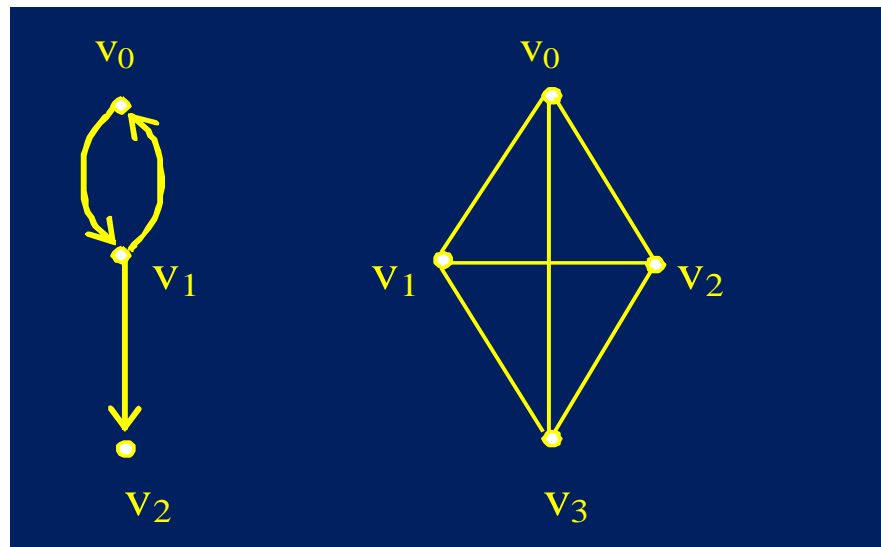
□ 路径、路径长度

- 如果 x 是 y 的一个祖先，又有 $x = x_0, x_1, \dots, x_n = y$ ，满足 x_i ($i = 0, 1, \dots, n-1$) 为 x_{i+1} 的父结点，则称 x_0, x_1, \dots, x_n 为从 x 到 y 的一条路径。 n 为这条路径的长度。
- 路径中相邻的两个结点可以表示成一条边。
- 例如树 t 中 A, C, E, I, J 是从 A 到 J 的一条路径，其长度为4



路径的例子

- 如图G1中顶点序列 v_0, v_1, v_0 是一长度为2的有向环
- G2中顶点序列 v_0, v_1, v_2, v_3 是一条从顶点 v_0 到 v_3 的长度为3的路径
- 顶点序列 v_0, v_1, v_3, v_0, v_2 是一条从顶点 v_0 到 v_2 的长度为4的路径，但不是简单路径
- 顶点序列 v_0, v_1, v_3, v_0 是一长度为3的环



概念5-图的根

□ **有向图中**，若存在一顶点 v ，从该顶点有路径可以到图中其它所有顶点，则称此有向图为**有根图**， v 称为**图的根**

显然，有根图中的根可能不唯一

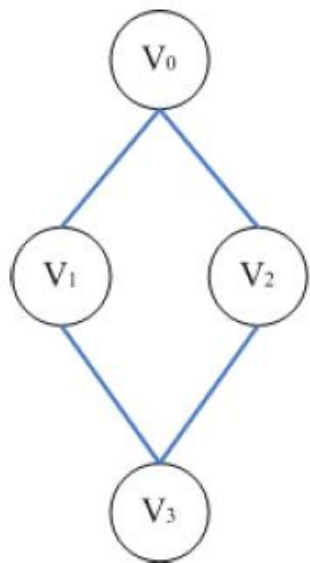


概念6-无向图的连通

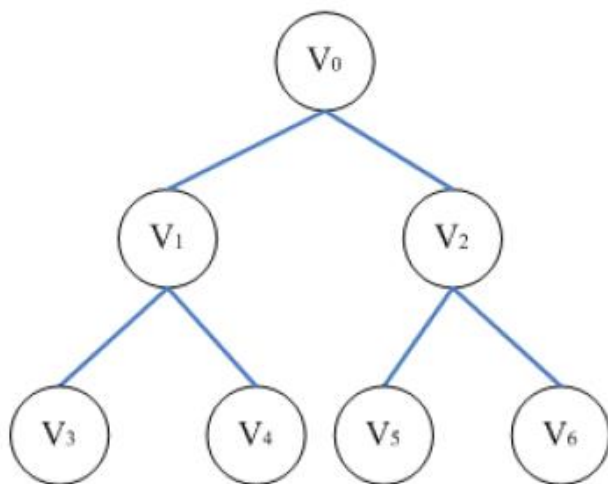
- **连通**：无向图 $G=(V, E)$ 中，若从 v_i 到 v_j 有一条路径(从 v_j 到 v_i 也一定有一条路径)，则称 v_i 和 v_j 是连通的
- **连通图**：若 $V(G)$ 中任意两个不同的顶点 v_i 和 v_j 都是连通的(即有路径)，则称 G 为连通图
- **连通分量**：无向图 G 中的最大连通子图（即任意增加 G 中结点或边以后所得到的 G 的子图都不再连通）称为 G 的连通分量
 - 连通图只有一个连通分量，就是其自身
 - 非连通的无向图有多个连通分量

无向图连通的例子

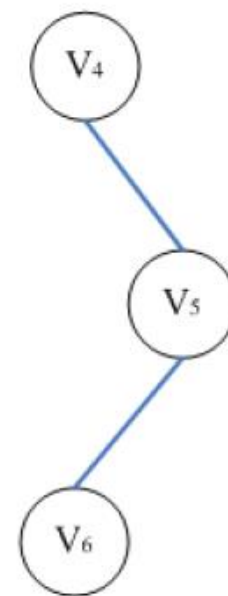
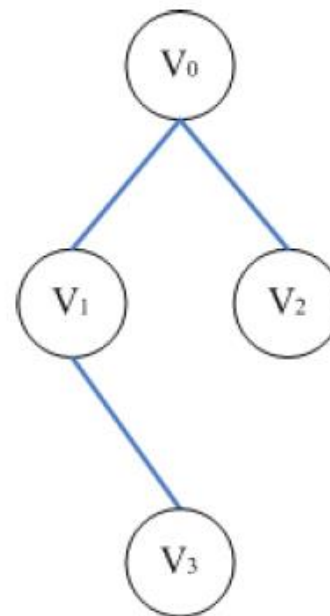
- 如图G2和G3都是连通图
- H1和H2是G4的两个连通分量



G2



G3



G4

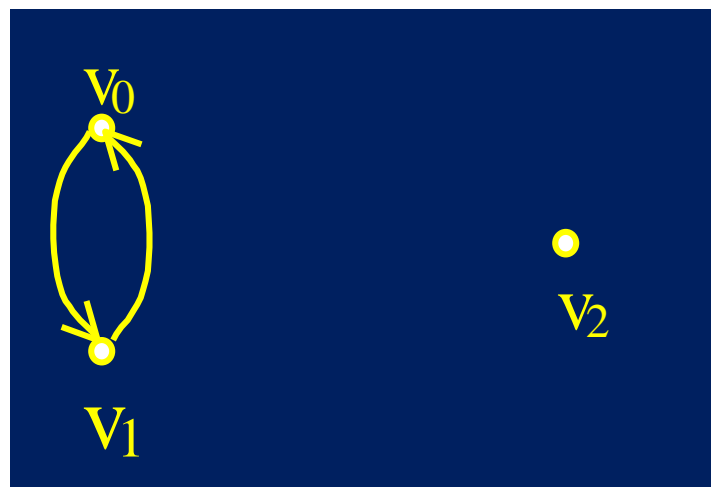
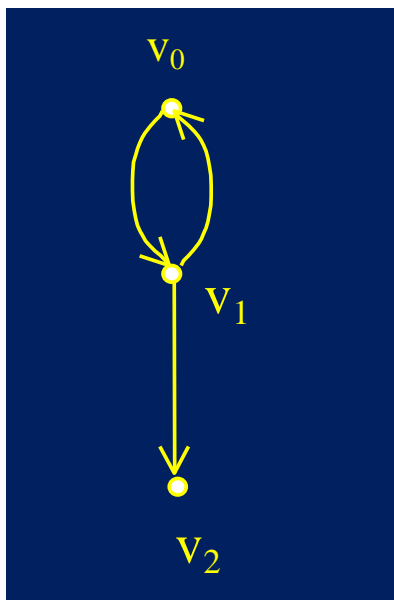
概念6-有向图的连通

- **强连通图**：有向图 $G=(V, E)$ 中，若 $V(G)$ 中任意两个不同的顶点 v_i 和 v_j 都存在从 v_i 到 v_j 以及从 v_j 到 v_i 的路径，则称图 G 是强连通图

- **强连通分量**：有向图的最大强连通子图称为图的强连通分量
 - 强连通图只有一个强连通分量，就是其自身。
 - 非强连通的有向图有多个强连通分量

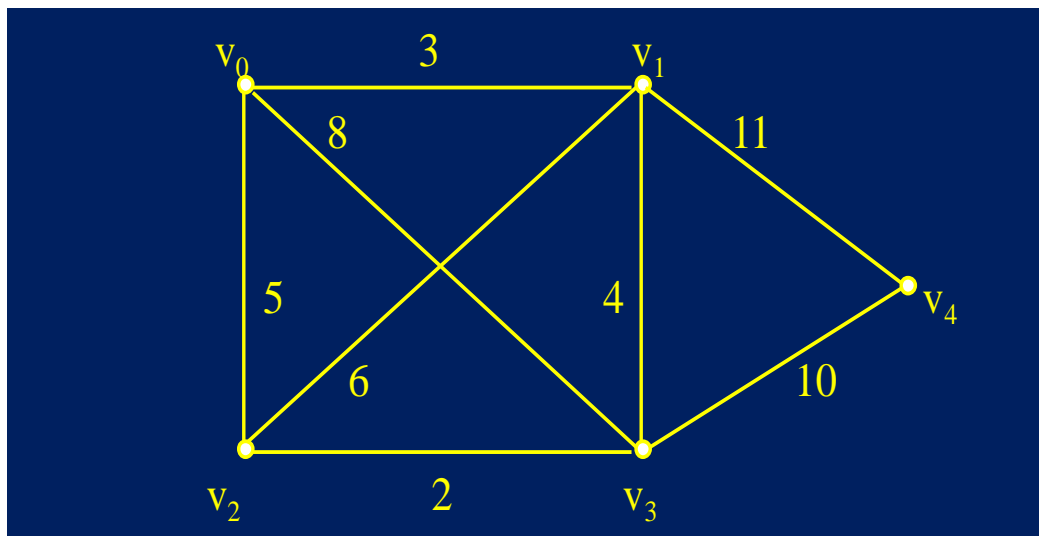
有向图连通的例子

- 如左图G1是非强连通图
- 它的两个强连通分量如右图所示



概念7-带权图、网络

- 若图的每条边都赋上一个权，则称为带权图
- 带权的连通图称为网络。
 - 通常权是具有某种意义的数。
 - 下图为一个网络。



内容提要

- 图的基本概念
- 存储表示
- 图的基本运算与周游
- 最小生成树
- 拓扑排序
- 关键路径
- 最短路径

图的存储

- 图的结构较复杂，任意两个顶点间都可能存在联系，因而图的存储方法也很多
- 应根据具体的应用和施加的操作选择图的存储表示法
 - 邻接矩阵表示法
 - 邻接表表示法
 - 邻接多重表表示法、图的十字链表等*

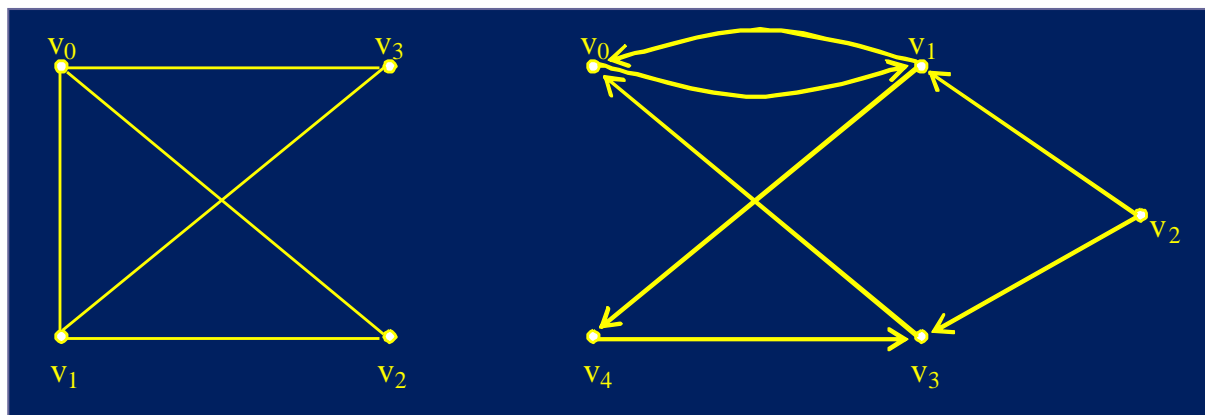
邻接矩阵表示法1

- **邻接矩阵**是表示顶点间相邻关系的矩阵：
 - 顶点信息
 - 关系信息：对称的n阶方阵（无向图）
- 设 $G=(V, E)$ 为具有n个顶点的图，其邻接矩阵A为具有以下性质的n阶方阵

$$A[i, j] = \begin{cases} 1, & \text{若 } (v_i, v_j) \text{ 或 } \langle v_i, v_j \rangle \text{ 是图 } G \text{ 的边} \\ 0, & \text{若 } (v_i, v_j) \text{ 或 } \langle v_i, v_j \rangle \text{ 不是图 } G \text{ 的边} \end{cases}$$

邻接矩阵表示法-例子

- 无向图G5和有向图G6的邻接矩阵分别为A1和A2



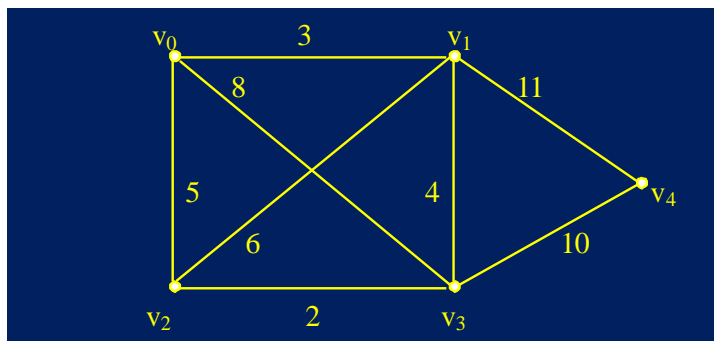
$$A_1 = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

$$A_2 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

邻接矩阵表示法- 带权图

□ 如果G是带权的图， w_{ij} 是边 (v_i, v_j) 或 $\langle v_i, v_j \rangle$ 的权，则其邻接矩阵A定义为：

$$A[i, j] = \begin{cases} w_{ij}, & \text{若 } (v_i, v_j) \text{ 或 } \langle v_i, v_j \rangle \text{ 是图 } G \text{ 的边} \\ 0 \text{ 或 } \infty, & \text{若 } (v_i, v_j) \text{ 或 } \langle v_i, v_j \rangle \text{ 不是图 } G \text{ 的边} \end{cases}$$



$$A_3 = \begin{bmatrix} 0 & 3 & 5 & 8 & 0 \\ 3 & 0 & 6 & 4 & 11 \\ 5 & 6 & 0 & 2 & 0 \\ 8 & 4 & 2 & 0 & 10 \\ 0 & 11 & 0 & 10 & 0 \end{bmatrix} \quad A_4 = \begin{bmatrix} \infty & 3 & 5 & 8 & \infty \\ 3 & \infty & 6 & 4 & 11 \\ 5 & 6 & \infty & 2 & \infty \\ 8 & 4 & 2 & \infty & 10 \\ \infty & 11 & \infty & 10 & \infty \end{bmatrix}$$

邻接矩阵表示法 - 存储结构

```
#define MAXVEX 100 // 常数1
```

```
typedef char VexType;
```

```
typedef float AdjType;
```

```
typedef struct {
```

```
    VexType vexs[MAXVEX];
```

```
/* 顶点信息 */
```

```
    AdjType arcs[MAXVEX] [MAXVEX];
```

```
/* 邻接矩阵 */
```

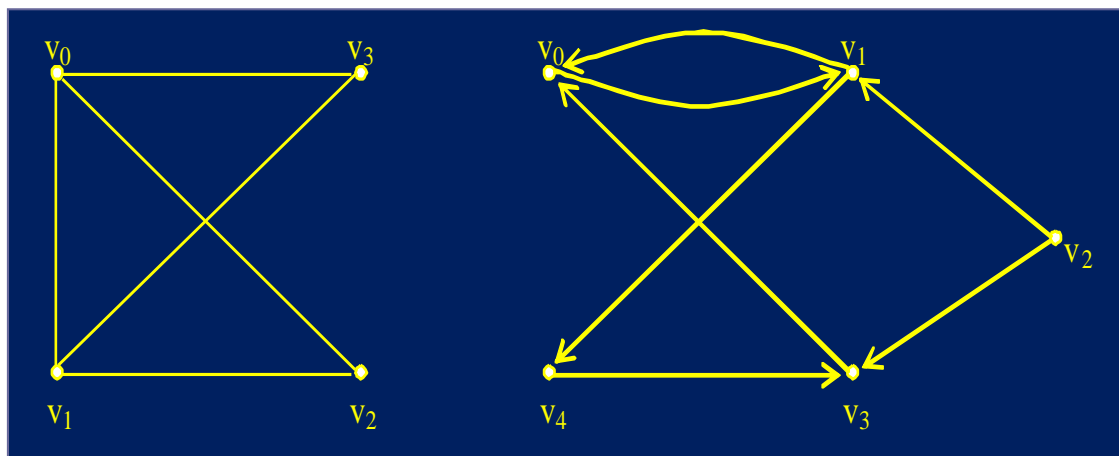
```
    int n;
```

```
/* 图的顶点个数 */
```

```
} Graph;
```

邻接矩阵表示法的特点

- 很容易确定图中任意两个顶点间是否有边连接
 - 无向图的邻接矩阵一定是一个**对称方阵**。
 - 无向图的邻接矩阵的第 i 行(或第 i 列)非零元素 (或非 ∞ 元素)个数为第 i 个顶点的度 $D(v_i)$ 。
 - 有向图的邻接矩阵的第 i 行非零元素(或非 ∞ 元素)个数为第 i 个顶点的出度 $OD(v_i)$
 - 有向图第 i 列非零元素(或非 ∞ 元素)个数就是第 i 个顶点的入度 $ID(v_i)$ 。



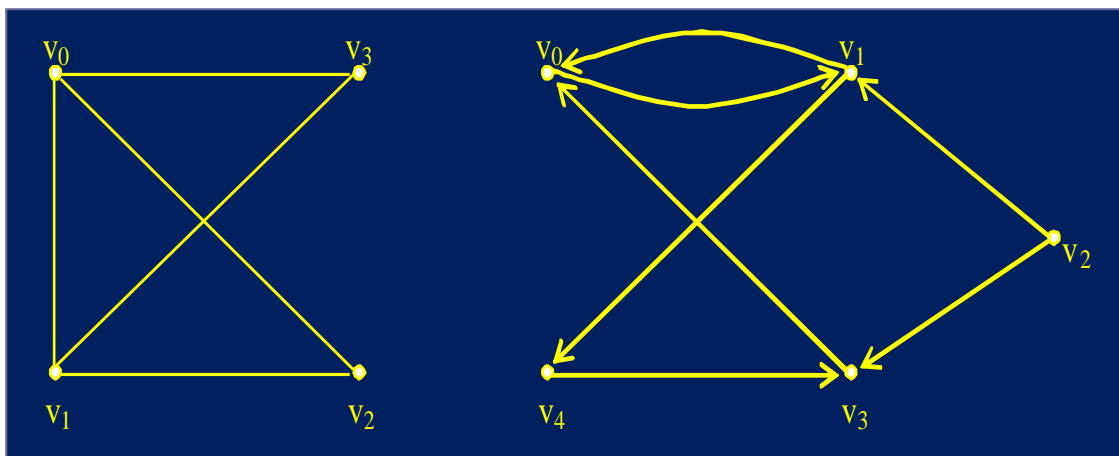
$$A_1 = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

$$A_2 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

邻接矩阵的特点

□ 邻接矩阵的存储代价

- 需要存储一个包括n个结点的顺序表来保存结点的数据或指向结点的数据指针
- 需存储一个n*n的邻接矩阵来指示结点间的关系
- 有向图：需n*n个单元来存储邻接矩阵 $O(n^2)$
- 无向图：由于邻接矩阵对称，只需存储相邻矩阵的下三角部分



$$A_1 = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

$$A_2 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

邻接矩阵的特点

□ 求解度数

- 有向图：矩阵第 i 行元素值的和是第 i 个结点的出度，第 i 列元素值的和是第 i 个结点的入度
- 无向图：邻接矩阵第 i 行（或第 i 列）元素值的和就是第 i 个结点的度数

□ 邻接矩阵表示法的优缺点

- 优点：各种基本操作都易于实现。
- 缺点：空间浪费严重。某些算法时间效率低。

图的存储

- 图的结构较复杂，任意两个顶点间都可能存在联系，因而图的存储方法也很多
- 应根据具体的应用和施加的操作选择图的存储表示法
 - 邻接矩阵表示法
 - 邻接表表示法

邻接表表示法

□ 由顺序存储的顶点表 + n 个链式存储的边表

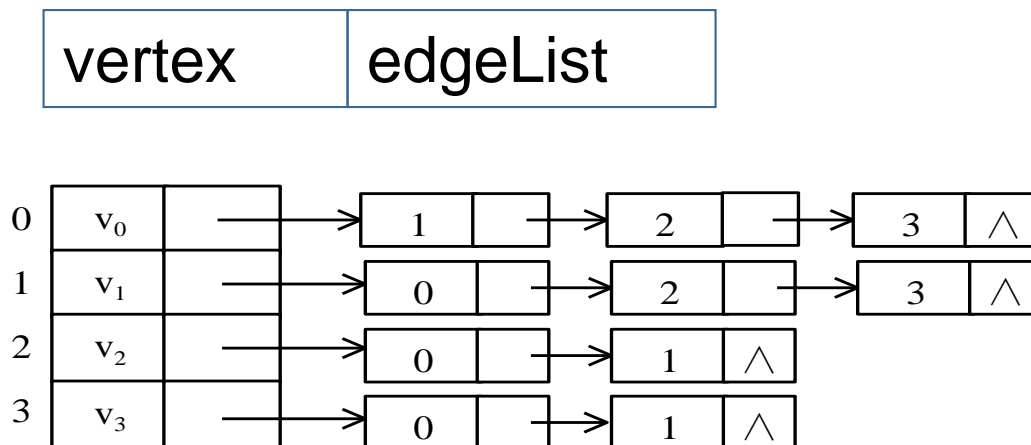
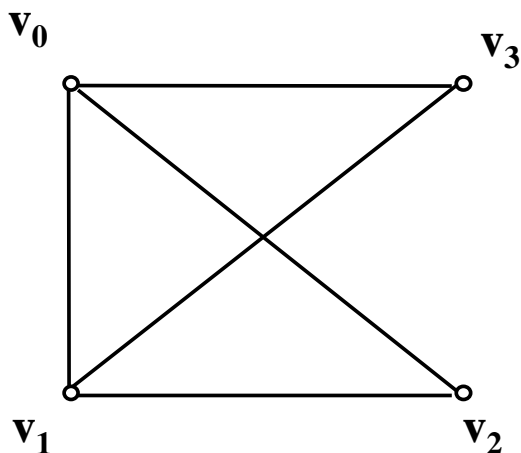
① 顺序存储的顶点表

■ 顶点表存放顶点本身的数据信息

■ 表中每个表目对应于图的一个顶点，包括两个字段

□ 顶点字段(vertex)存放顶点 v_i 的信息

□ 指针字段(edgeList)存放与 v_i 相关联的边表中的第一个边结点的地址



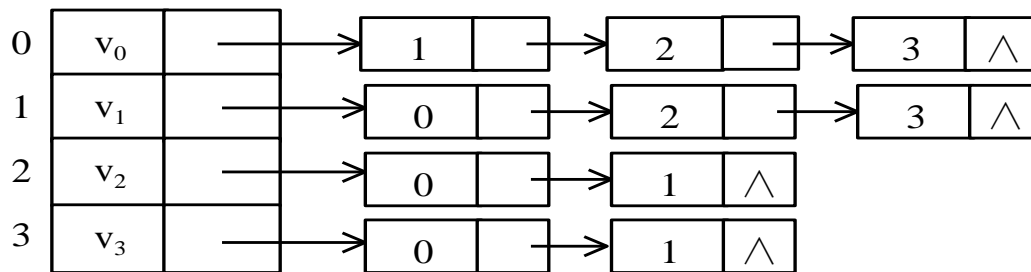
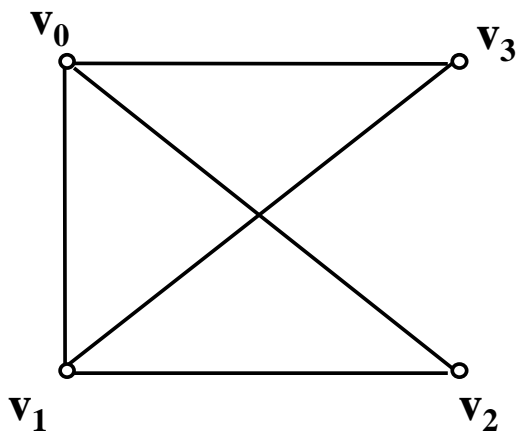
邻接表表示法（续）

② n个链式存储的边表

■ 边表中每个边结点包括三个字段：

- 相邻顶点字段(endvex)存放当前边的另一个顶点 v_j 在顶点表中的位置j
- 权字段(weight)存放边结点所代表的边的权值
- 链字段(nextedge)指向边表的下一个边结点

endvex	nextedge	weight
--------	----------	--------

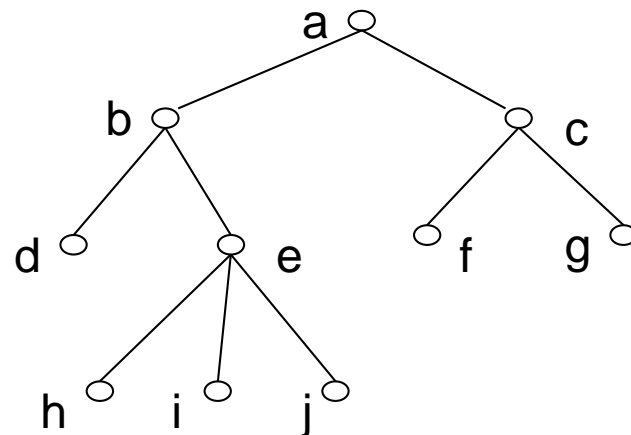


省略权值

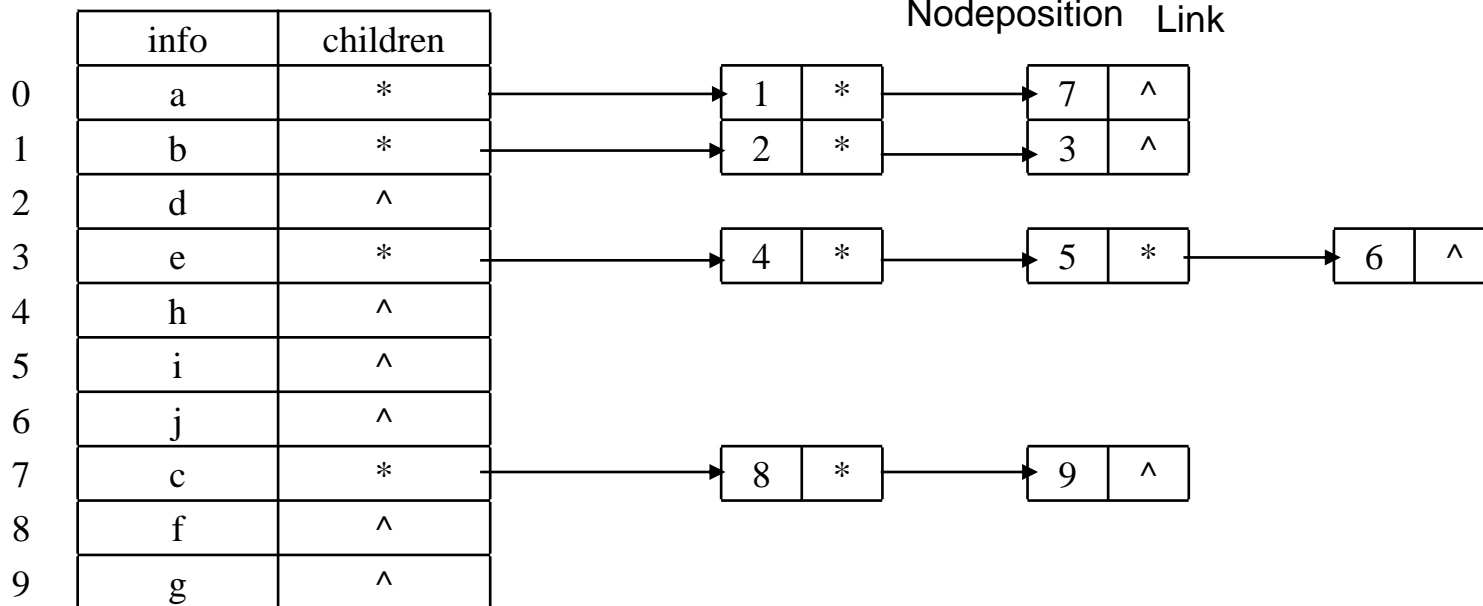
树和树林的存储表示2

□ 子表表示法

- 整棵树组织成一个结点顺序表，其中每一结点包含一个子表，存放该结点的所有子结点，子表用链接表示。

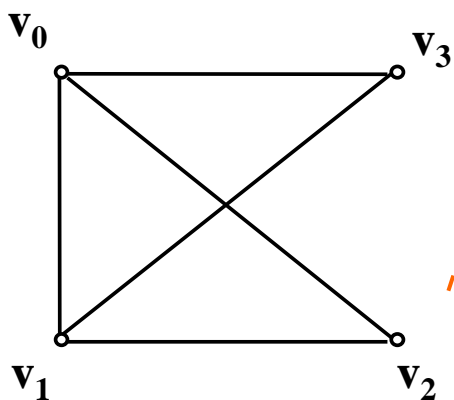
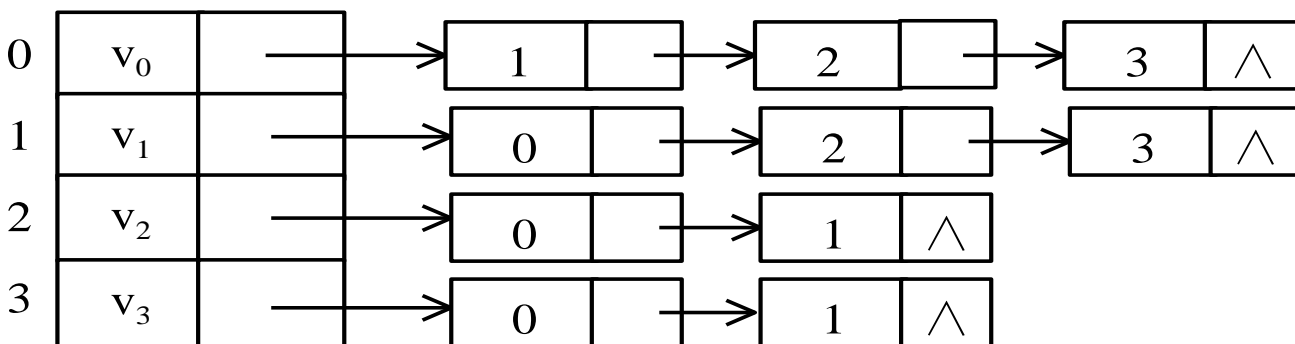


Nodelist



无向图的表示

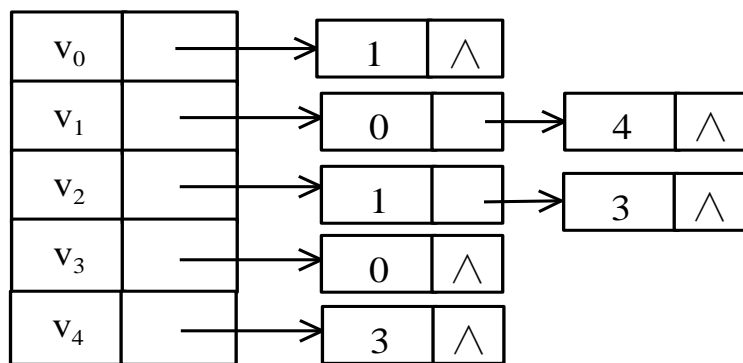
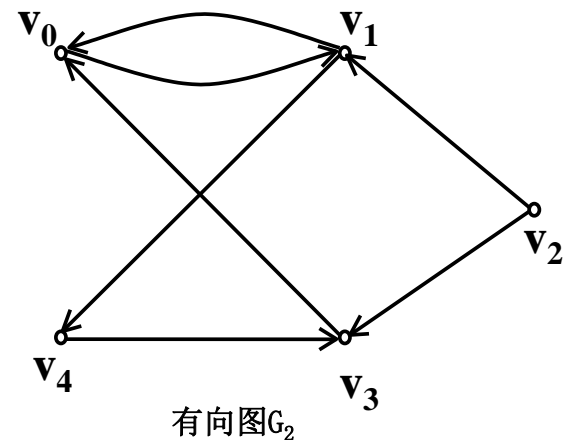
- 每条边 (v_i, v_j) 在两个顶点 v_i, v_j 的边表中都占一个结点，因此，每条边在边表中**存储两次**。
- 顶点 v_i 的边表中结点个数为**顶点 v_i 的度**



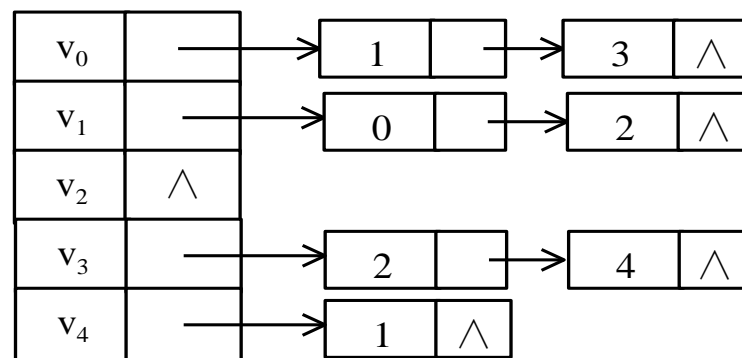
无向图 G_1 的邻接表

有向图的表示

- 顶点 v_i 的边表中每个结点对应的是以 v_i 为始点的一条边，因此，将有向图邻接表的边表称为出边表。
- 顶点 v_i 的边表中结点个数为顶点 v_i 的出度
- 也可将表表示为入边表



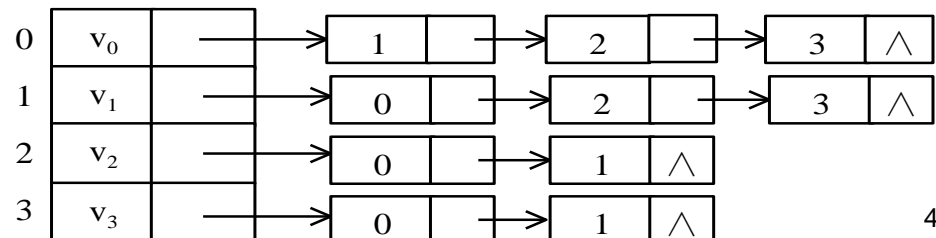
有向图 G_2 的邻接表(出边表)



有向图 G_2 的逆邻接表(入边表)

邻接表表示法的实现

```
struct EdgeNode;
typedef struct EdgeNode * PEdgeNode;
typedef struct EdgeNode * EdgeList;
struct EdgeNode
{
    int endvex;           /* 相邻顶点位置 */
    AdjType weight;      /* 边的权 */
    PEdgeNode nextedge;  /* 链字段 */
};                       /* 边表中的结点 */
typedef struct
{
    VexType vertex;      /* 顶点信息 */
    EdgeList edgelist;    /* 边表头指针 */
} VexNode;              /* 顶点表中的结点 */
typedef struct
{
    VexNode vexs[MAXVEX]; /* 顶点表 */
    int n;                /* 图的顶点个数 */
} GraphList;
```

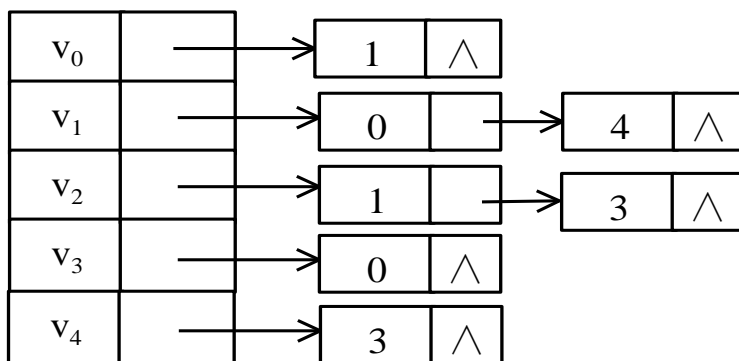
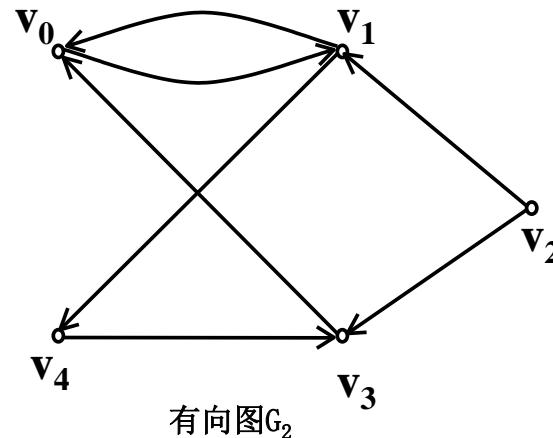


邻接表的存储代价

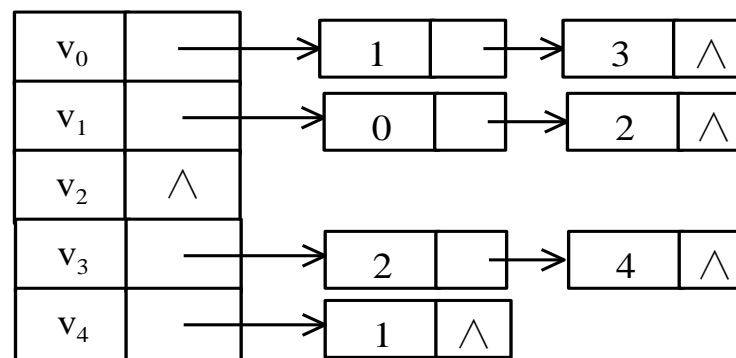
- 若图G是无向图，则图的邻接表表示的空间代价为 $O(n+2e)$
- 若图G是有向图，则图的邻接表表示的空间代价为 $O(n+e)$

一些基本操作的实现

- 求无向图中某个顶点的度：
为该结点所指向的链表中的结点总数
- 求有向图的出度：
该结点所指向的链表的结点总数
- 求有向图的入度：
必须搜索整个邻接表才能得到
改进：增加逆邻接表（入边表）



有向图 G_2 的邻接表（出边表）



有向图 G_2 的逆邻接表（入边表）

邻接表的优缺点

□ 优点

- 容易找任一结点的第一个邻接点和下一个邻接点；
- 存储量小

□ 缺点：

- 判定任意两个结点之间是否有边不方便（需要扫描顶点边表）。

如何选择合适的存储方法？

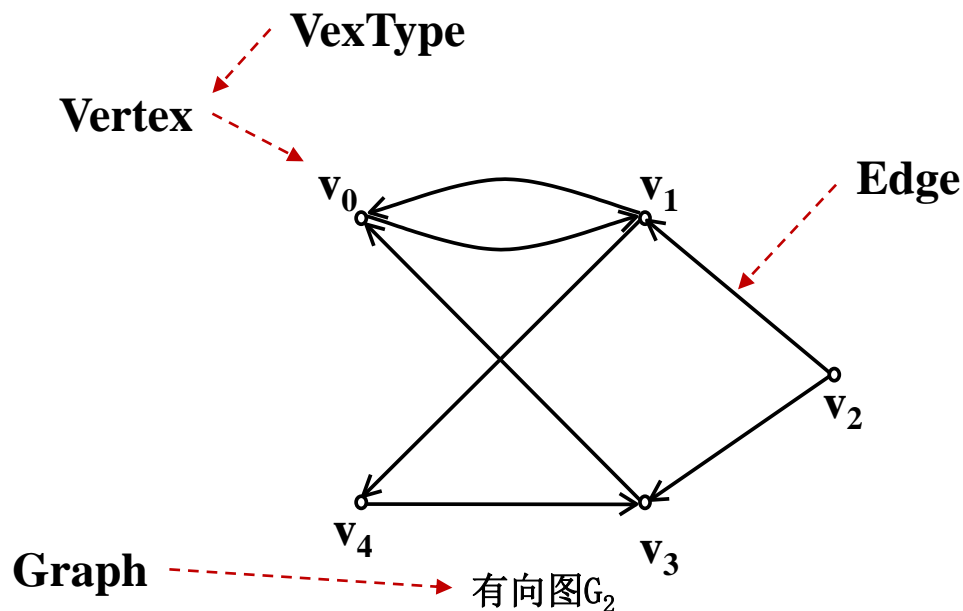
- 邻接矩阵表示的空间代价**只与图的顶点数有关**
- 若图中边的数目小于顶点的数目，则用邻接表表示比较节省空间
- 如果边数达到 n^2 数量级时，由于邻接表中增加了辅助的链域，采用邻接矩阵表示图更节省空间。特别对于无权图而言，邻接矩阵的每个元素只要一个位就可以表示

内容提要

- 图的基本概念
- 存储表示
- 图的基本运算与周游
- 最小生成树
- 拓扑排序
- 关键路径
- 最短路径

图的基本运算

- 定义图的基本类型为Graph，图的顶点类型为Vertex，顶点信息的类型为VexType，图中边的类型为Edge



图的基本运算

□ 图中的基本运算可以定义如下：

- ① 创建一个空图 `Graph creatGraph()`
- ② 判断图g是否是空图 `int isNullGraph(Graph g)`
- ③ 把图g置为空图 `void makeNullGraph(Graph g)`
- ④ 销毁一个图g，即释放图g占用的空间
`void destroyGraph (Graph g)`
- ⑤ 查找图中值为value的顶点
`Vertex searchVertex(Graph g, VexType value)`

图的基本运算

- ⑥ 在图g中增加一个值为value的顶点

`Graph addVertex(Graph g, VexType value)`

- ⑦ 在图g中删除一个顶点和与该顶点相关联的所有边

`Graph deleteVertex(Graph g, Vertex v)`

- ⑧ 在图g中删除/增加一条边 $\langle v_i, v_j \rangle$

`Graph deleteEdge/addEdge(Graph g, Vertex v_i , Vertex v_j)`

- ⑧ 判断图g中是否存在一条指定边 $\langle v_i, v_j \rangle$

`int findEdge(Graph g, Vertex v_i , Vertex v_j)`

.....

由于不同的应用要求，实现的操作有很大差别

图的周游

- **图的周游：**从图中某一顶点出发，按照某种方式系统地访问图中所有顶点，且使每一个结点**被访问且仅被访问一次**。也称为**图的遍历**。

- **连通图或强连通图：**
 - 从图中任意一顶点出发都可以访问图中所有顶点

- 由于图中每个顶点都可能与图中其它多个顶点邻接并存在回路，为了避免重复访问已访问过的顶点，在图的周游中，**通常对已访问过的顶点作标记**。

图的周游方法

□ 深度优先周游

■ (Depth First Search/Traversal, DFS)

□ 广度优先周游

■ (Breadth First Search/Traversal, BFS)

深度优先周游

□ 具体的思想：

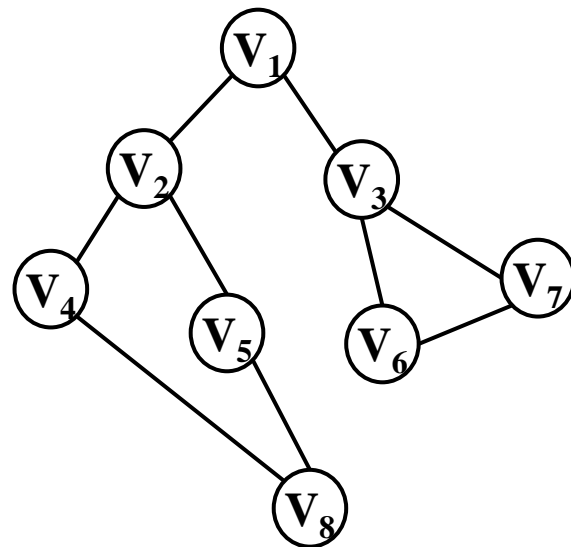
- ① 从图的指定顶点 v 出发，先访问顶点 v ，并将其**标记为已访问过**
- ② 然后依次从 v 的**未被访问过的邻接顶点** w 出发进行深度优先搜索，直到图中与 v 相连的所有顶点都被访问过
- ③ 如果图中还有未被访问的顶点，则从**另一未被访问过的顶点出发重复上述过程**，直到图中所有顶点都被访问过为止

□ 举例：V1出发深度优先搜索序列：

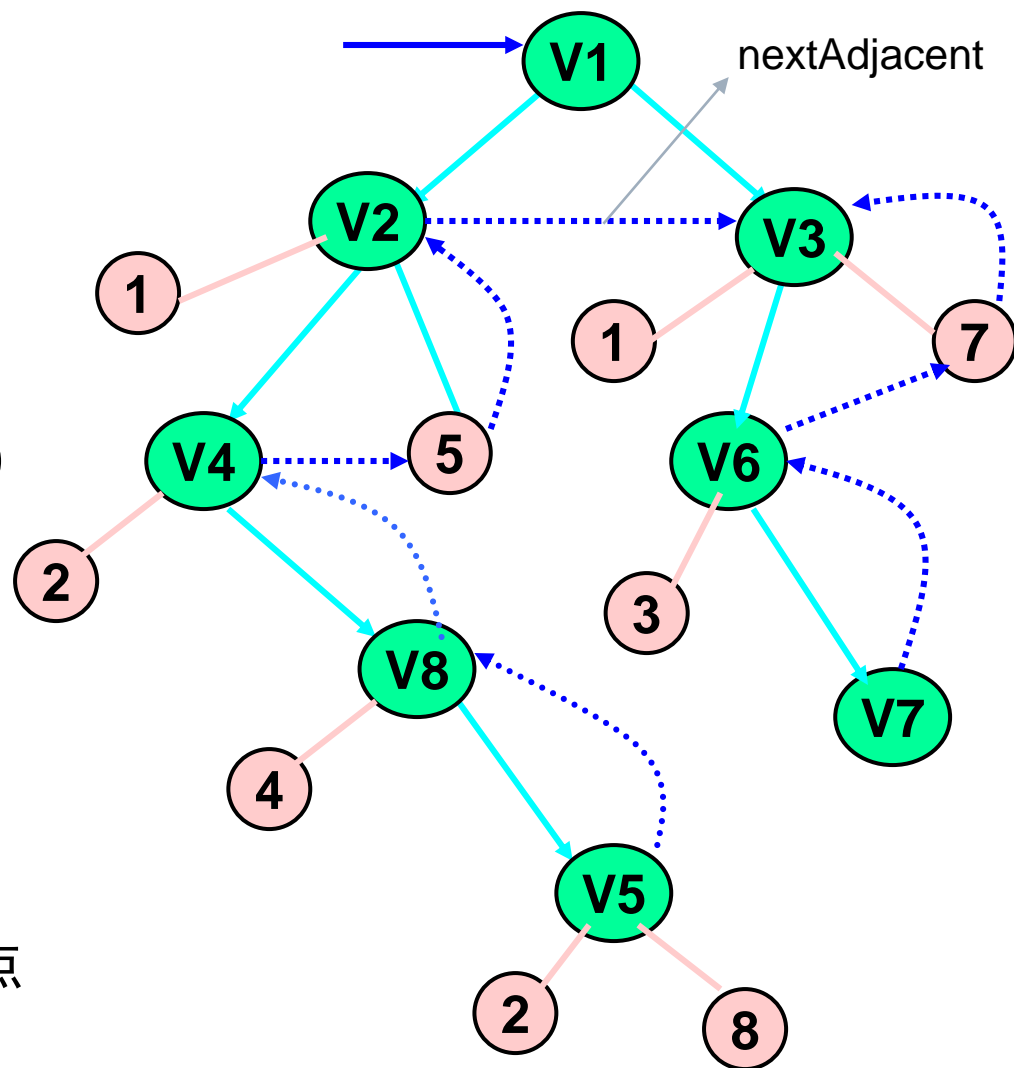
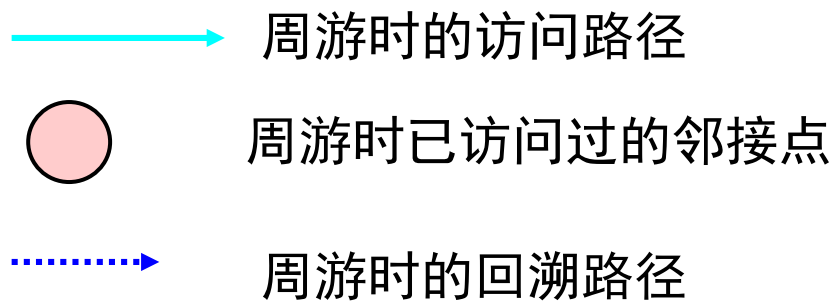
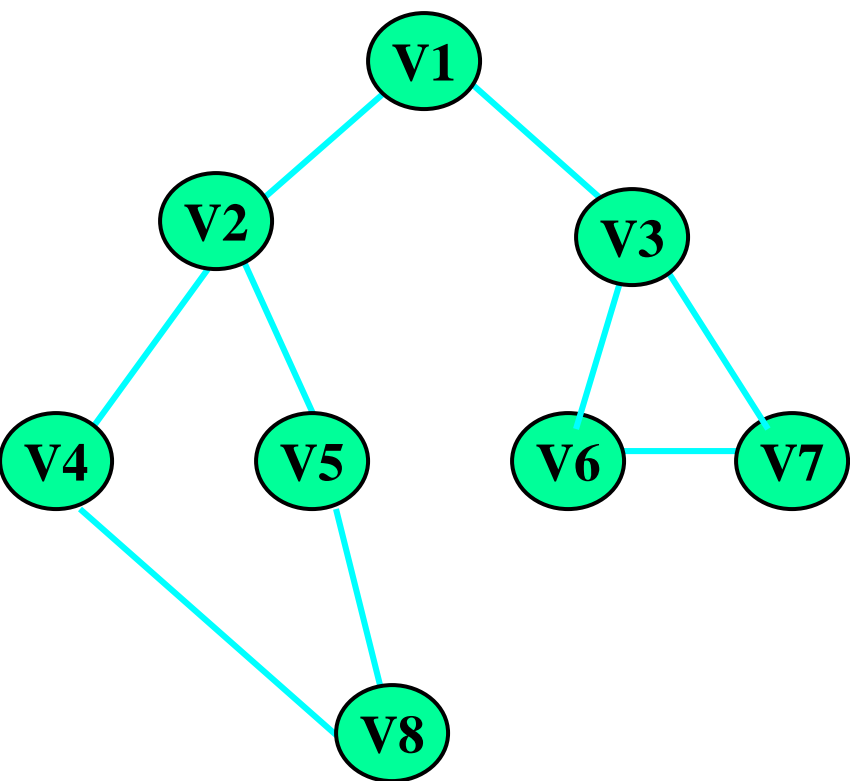
V1->V2->V4->V8->V5->V3->V6->V7

■ 访问过的结点需要特殊标志，避免回路。

□ 对图进行深度优先周游时，按访问顶点的先后次序所得到的顶点序列，称为该图的**深度优先搜索序列**，简称**DFS序列**

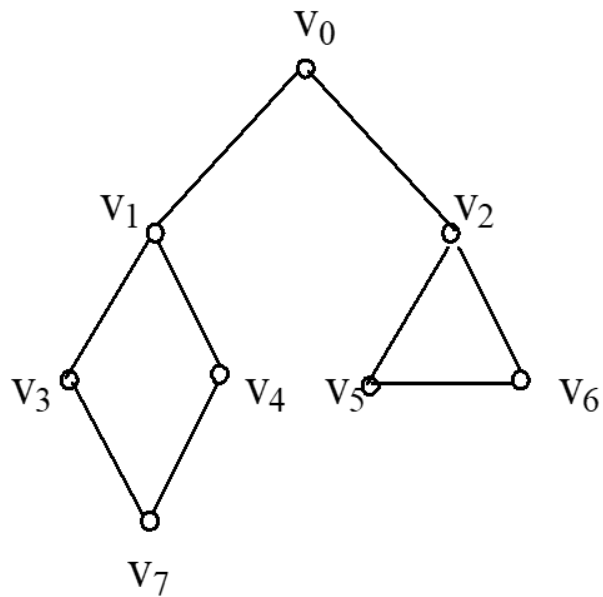


例子_1:



v1→v2→ v4 → v8→v5→v3→v6→v7

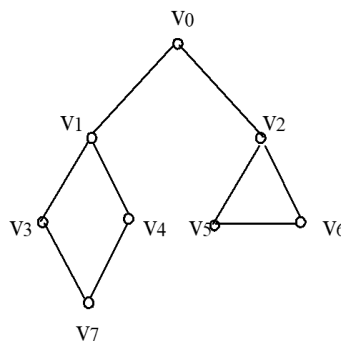
已知无向图及其邻接表，求DFS序列



V ₀		→	1		→	2	∧
V ₁		→	0		→	3	→ 4 ∧
V ₂		→	0		→	5	→ 6 ∧
V ₃		→	1		→	7	∧
V ₄		→	1		→	7	∧
V ₅		→	2		→	6	∧
V ₆		→	2		→	5	∧
V ₇		→	3		→	4	∧

/* 从一个顶点出发进行深度优先搜索，图采用邻接表存储 */

```
void dFSInList(GraphList * pgraphlist, int visited[], int i)
{
    int j;
    PEdgeNode p;
    printf("node: %c\n", pgraphlist->vexs[i].vertex);
    visited[i]=TRUE;           /*标记为已经访问过*/
    p=pgraphlist->vexs[i].edgelist; /* 取边表中的第一个边结点 */
    while(p!=NULL)
    {
        if (visited[p->endvex]==FALSE) /* 未被访问 */
            dFSInList(pgraphlist,visited,p->endvex); /*从邻接节点开始深度优先搜索*/
        p=p->nextedge; /* 取边表中的下一个边结点 */
    }
}
```



V0	→	1	→	2	∧
V1	→	0	→	3	→ 4 ∧
V2	→	0	→	5	→ 6 ∧
V3	→	1	→	7	∧
V4	→	1	→	7	∧
V5	→	2	→	6	∧
V6	→	2	→	5	∧
V7	→	3	→	4	∧

图的深度优先周游

```
traverDFS(GraphList * pgraphlist);
{
    int n= pgraphlist->n;
    int visited[n];
    for(i=0; i<n; i++)
        visited[i] = FALSE;    /* 初始化数组visited */
    for(i=0; i<n; i++)
        if(visited[i] == FALSE)
            dFSInList(pgraphlist, visited, i);
}
```

说明：

- 设图有n个顶点，e条边，若图是连通图，则traverDFS()只需调用dFSInList一次，就可完成图的深度优先周游；
- 否则调用几次则表明图中有几个连通分量

从一个顶点出发进行深度优先搜索，图采用邻接矩阵表示法

/*初始化，首先设置一个数组visited(大小n=8，为图中顶点个数)，各元素的初值为FALSE，表示所有顶点均未访问过。*/

```
void dFSInMatrix(Graph * pGraph, int visited[ ], int i)
```

```
{  int j;
```

```
    printf(“node: %c\n”, pGraph->vexs[i]);  /* 访问出发点i */
```

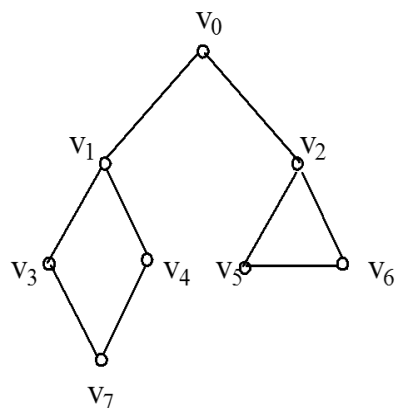
```
    visited[i] = TRUE;
```

```
    for(j=0; j<pGraph->n; j++)
```

```
        if( pGraph->arcs[i][j] == 1 && visited[j]==FALSE )
```

```
            dFSInMatrix(pGraph, visited, j);
```

```
}
```



$$A_5 = \begin{matrix} & v0, & v1, & v2, & v3, & v4, & v5, & v6, & v7 \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \end{bmatrix} \end{matrix}$$

图的深度优先周游

```
void traverDFS(Graph * pGraph)
```

```
{  int visited[MAXVEX];          /* 初始化数组visited */
```

```
    for(i=0;i<pGraph->n;i++) visited[i]=FALSE;
```

```
    for(i=0; i< pGraph->n; i++)
```

```
        if(visited[i]==FALSE)
```

```
            dFSInMatrix(pGraph,visited,i);
```

```
/*dFSInList(pGraph,visited,i); 调用多少次，表示图中有多少个连通分量*/
```

```
}
```

DFS算法分析

□ 时间复杂度：

- 采用邻接矩阵表示— $O(n^2)$ ，得到一个顶点的所有邻接点需要检查矩阵相应行中的 n 个元素
- 采用邻接表表示— $O(n+e)$ ，检查一个顶点的所有邻接点是对其边表中边结点扫描一遍

□ 空间复杂度：

算法所用的辅助空间都是标志数组及实现递归所用的栈，算法的辅助空间为 $O(n)$

图的周游方法

□ 深度优先周游

■ (Depth_First Search/Traversal, DFS)

□ 广度优先周游

■ (Breadth_First Search/Traversal, BFS)

广度优先周游

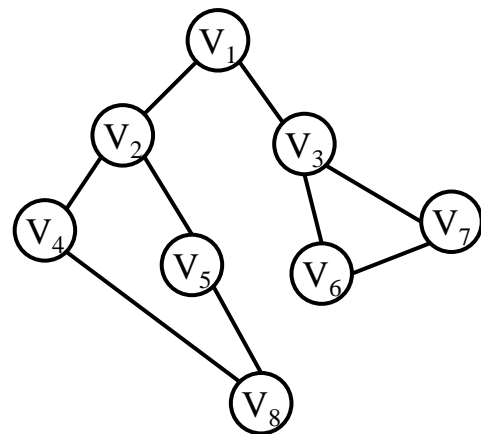
□ 具体的思想：

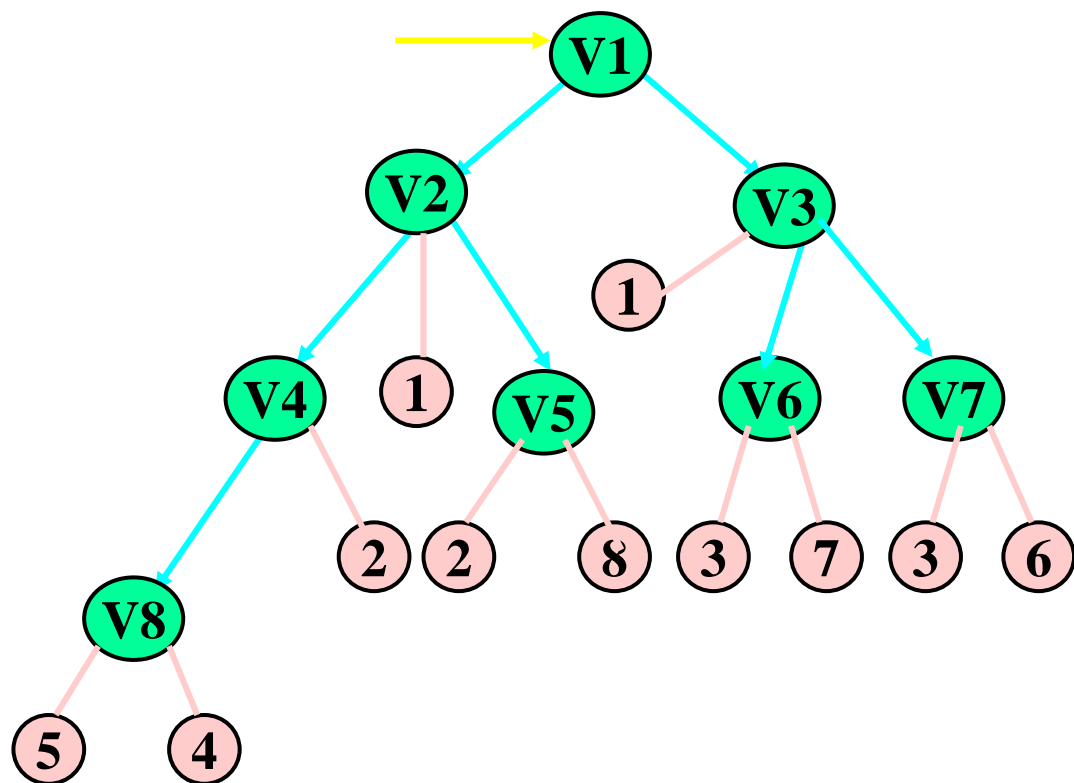
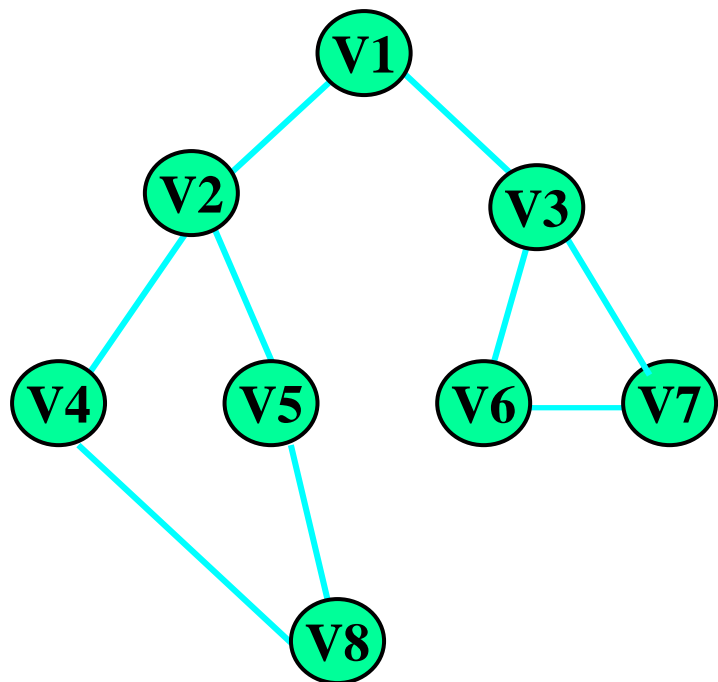
- ① 从图的指定顶点 v 出发，先访问顶点 v ，并将其标记为已访问过；
- ② 接着依次访问 v 的所有邻接点 w_1, w_2, \dots, w_x
- ③ 然后，再依次访问与 w_1, w_2, \dots, w_x 邻接的所有未被访问过的顶点；
- ④ 以此类推，直到所有已访问顶点的邻接点都被访问过；
- ⑤ 如果图中还有未被访问过的顶点，则从另一未被访问过的顶点出发进行广度优先搜索，直到所有顶点都被访问过为止

□ 举例

- V_1 出发广度优先： $V_1 \rightarrow V_2 \rightarrow V_3 \rightarrow V_4 \rightarrow V_5 \rightarrow V_6 \rightarrow V_7 \rightarrow V_8$
- 访问过的结点需要特殊标志，避免回路

□ 广度优先周游得到的顶点序列称为广度优先搜索序列，简称BFS序列





→ 周游时的访问路径

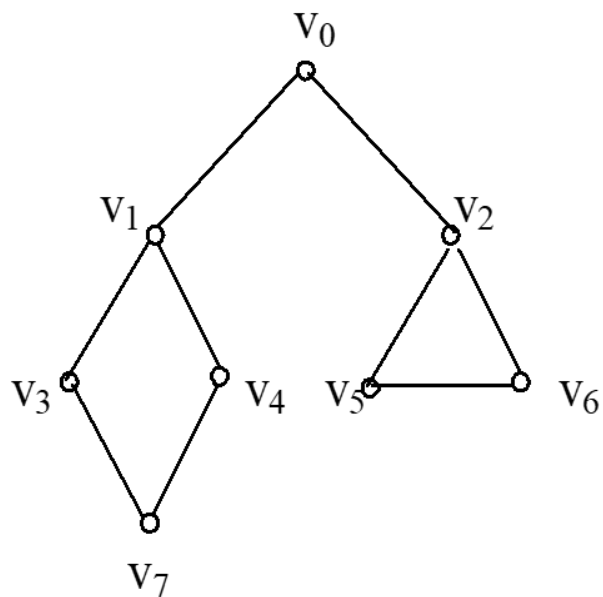
○ 周游时已访问过的邻接点

$v1 \rightarrow v2 \rightarrow v3 \rightarrow v4 \rightarrow v5 \rightarrow v6 \rightarrow v7 \rightarrow v8$

广度优先周游算法

- 对于广度优先周游，关键在于怎么保证“先被访问的顶点的邻接点”要先于“后被访问的顶点的邻接点”被访问。=> 队列控制
 - 设立一个队列，将访问过的顶点依次进队列，
 - 按顶点进队列先后顺序访问它们的邻接点。

已知无向图及其邻接表，求BFS序列



V_0	→	1	→	2	\wedge	
V_1	→	0	→	3	→	4 \wedge
V_2	→	0	→	5	→	6 \wedge
V_3	→	1	→	7	\wedge	
V_4	→	1	→	7	\wedge	
V_5	→	2	→	6	\wedge	
V_6	→	2	→	5	\wedge	
V_7	→	3	→	4	\wedge	

v0→v1→v2→ v3 → v4→v5→v6→v7

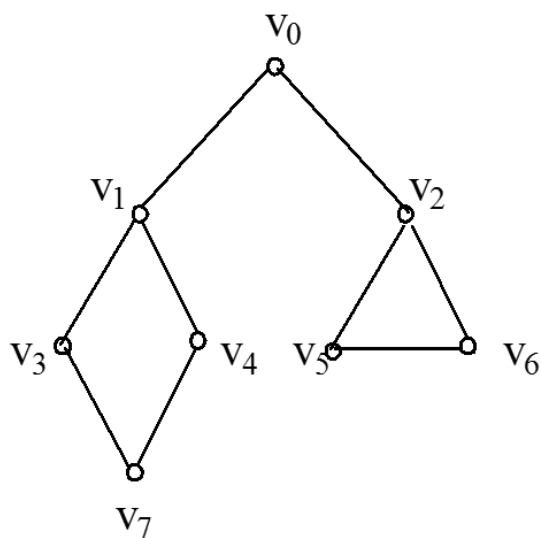
```

void bFSInList(GraphList *pgraphlist, int visited[], int i)
{
    PLinkQueue pq; PEdgeNode p; int j;
    pq=createEmptyQueue_link();           /* 置队列为空 */
    printf("node:%c\n",pgraphlist->vexs[i].vertex);
    visited[i]=TRUE;                      /*设置已访问标志*/
    enqueue_link(pq,i);                   /* 将顶点序号进队 */
    while (!isEmptyQueue_link(pq) ) {     /* 队列非空时执行 */
        j=deQueue_link(pq);               /* 队头顶点出队 */
        p=pgraphlist->vexs[j].edgelist;   /*取队头元素的邻接边表*/
        while( p!=NULL) {
            if (!visited[p->endvex]) {     /*访问相邻接的未被访问过的顶点 */
                printf("node:%c\n",pgraphlist->vexs[p->endvex].vertex);
                visited[p->endvex]=TRUE;   /*设置已访问标志 */
                enqueue_link(pq,p->endvex); /*访问的顶点入队 */
            }
            p=p->nextedge;
        }
    }
}

traverBFS(GraphList *pgraphlist)
{
    int visited[MAXVEX];
    int i,n;
    n=pgraphlist->n;
    for(i=0;i<n;i++)
        visited[i]=FALSE;
    for(i=0; i<n; i++)
        if(visited[i]==FALSE)
            bFSInList(pgraphlist,visited,i);
}

```

已知无向图的邻接矩阵，求BFS序列



$$A_5 = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

BFS算法分析

□ 时间复杂度:

- 算法bFSInMatrix的时间复杂度为 $O(n^2)$
- 算法bFSInList的时间复杂度为 $O(n+e)$

□ 空间复杂度:

- 两个算法的辅助空间主要是标志数组及一个队列，大小不超过 $O(n)$