

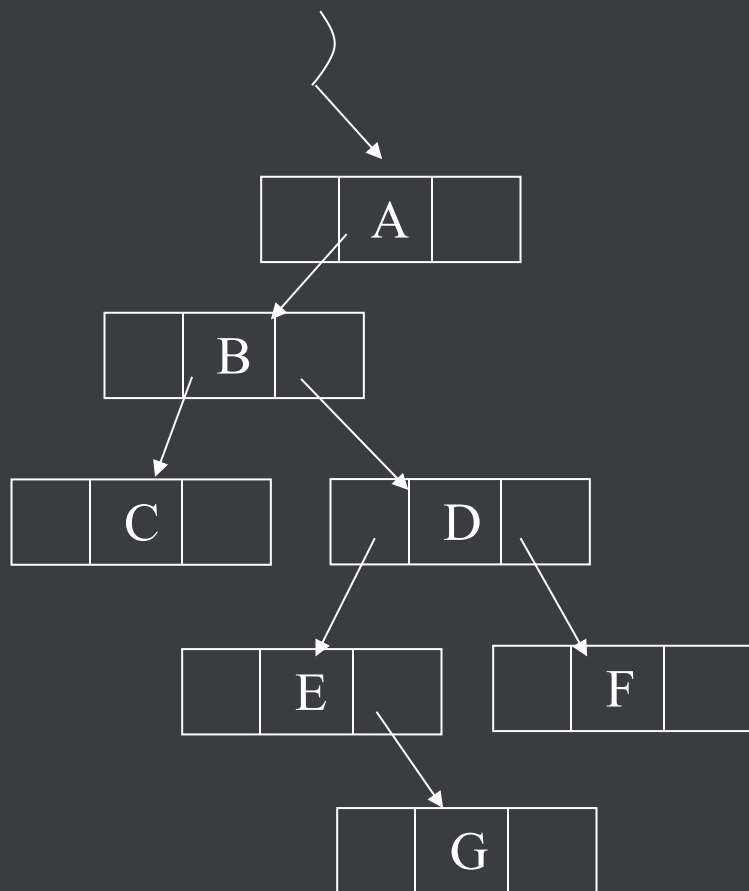
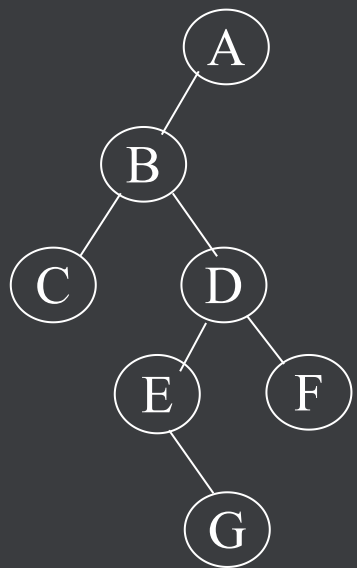
# 二叉树

链式存储结构：

```
typedef int data_t;           /*定义数据类型*/
typedef struct node_t;        /*定义二叉树节点的内部结构*/
{
    data_t data;              /*数据域*/
    struct node_t *lchild, *rchild; /*指向左孩子和右孩子的指针*/
} bitree_t;                   /*二叉树节点类型*/
bitree_t *root;               /*定义指向二叉树的指针*/
```

二叉树由根节点指针决定。

# 二叉树



4.3

## 二叉树 的遍历

# 二叉树的遍历

## 二叉树的遍历

遍历：沿某条搜索路径周游二叉树，对树中的每一个节点访问一次且仅访问一次。

“遍历”是任何类型均有的操作，对**线性结构**而言，只有一条搜索路径(因为每个结点均只有一个后继)，故不需要另加讨论。而二叉树是**非线性结构**，每个结点有两个后继，则存在如何遍历即按什么样的搜索路径进行遍历的问题。

# 二叉树的遍历

由于二叉树的递归性质，遍历算法也是递归的。三种基本的遍历算法如下：

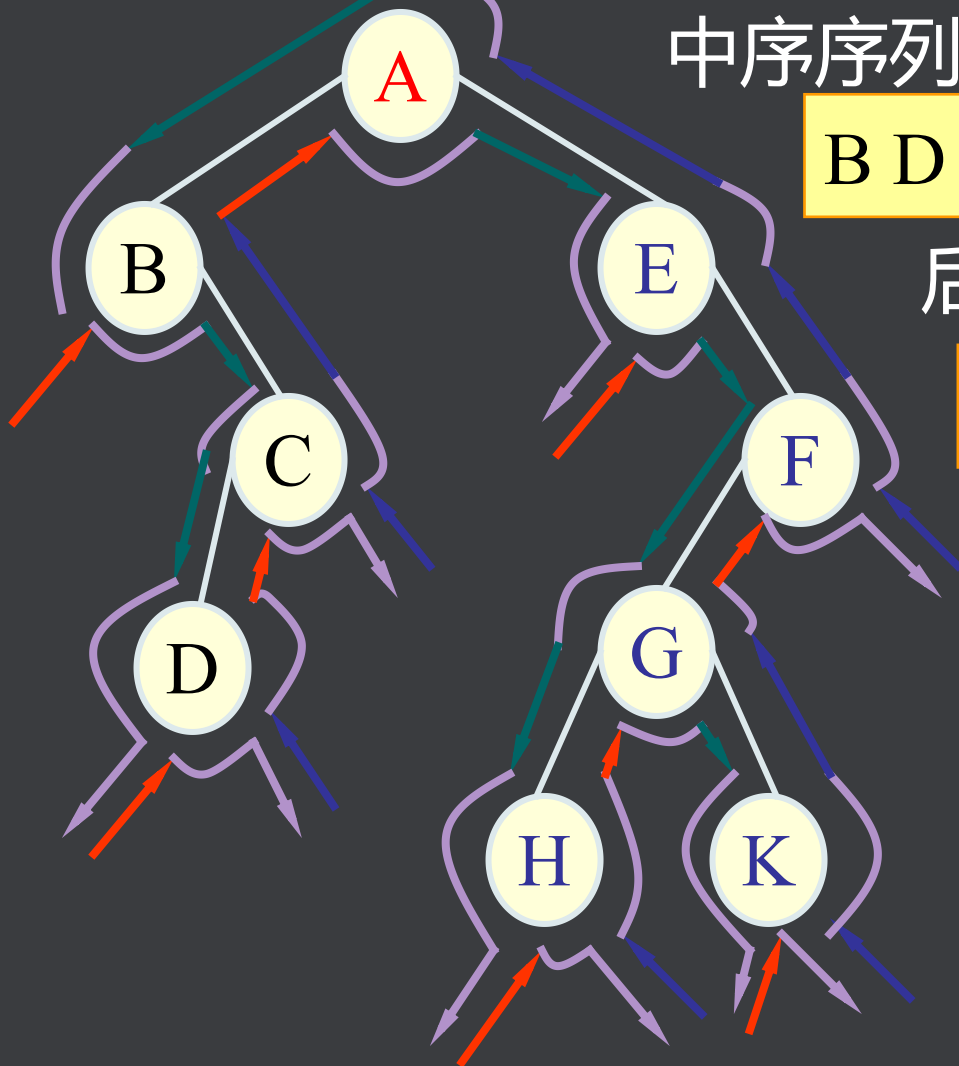
先访问树根，再访问左子树，最后访问右子树；

先访问左子树，再访问树根，最后访问右子树；

先访问左子树，再访问右子树，最后访问树根；

# 二叉树的遍历

例如：



先序序列：

A B C D E F G H K

中序序列：

B D C A E H G K F

后序序列：

D C B H K G F E A



# 二叉树的遍历

## 先序遍历算法

若二叉树为空树，则空操作；否则

访问根结点

先序遍历左子树

先序遍历右子树

# 二叉树的遍历

## 先序遍历算法

```
void PREORDER ( bitree *r)
{
    if ( r == NULL ) return ;    //空树返回
    printf ( " %c " ,r->data );    //先访问当前结点
    PREORDER( r->lchild );    //再访问该结点的左子树
    PREORDER( r->rchild );    //最后访问该结点右子树
}
```



# 二叉树的遍历

PREORDER ( bitree \*r)

{

if ( r == NULL ) return ;

printf ( " %c ",r->data );

PREORDER ( r->lchild );

PREORDER ( r->rchild );

}

左是空返回

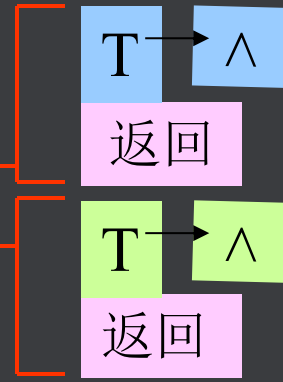
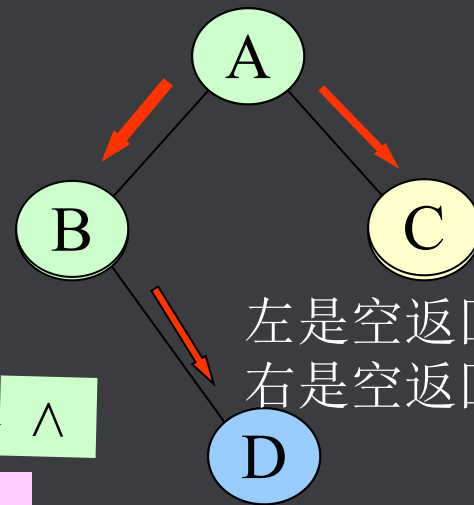
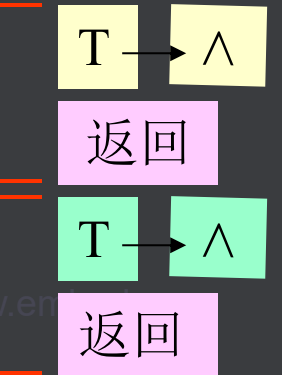
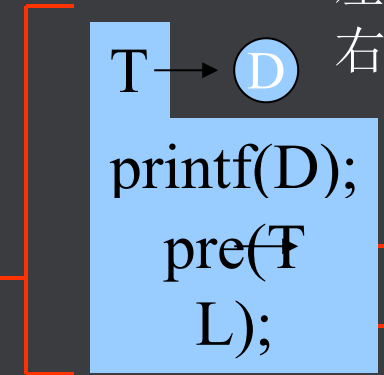
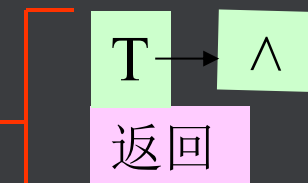
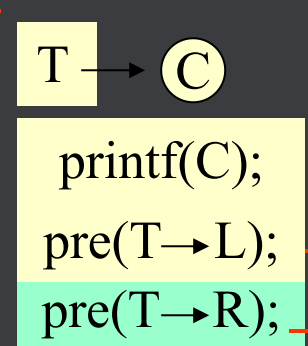
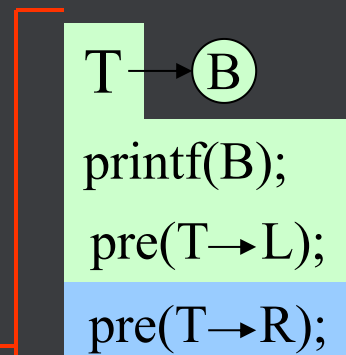
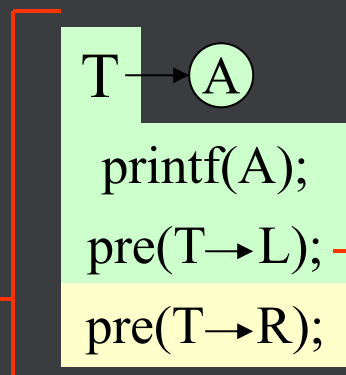
左是空返回  
右是空返回

左是空返回  
右是空返回

主程序

Pre( T )

先序序列: A B D C



# 二叉树的遍历

## 中序遍历算法

若二叉树为空树，则空操作；否则

中序遍历左子树

访问根结点

中序遍历右子树

# 二叉树的遍历

## 中序遍历算法

```
void INORDER ( bitree *r)
{
    if ( r == NULL ) return ; //空树返回
    INORDER( r->lchild ); //先访问该结点的左子树
    printf ( " %c " ,r->data ); //再访问当前结点
    INORDER( r->rchild ); //最后访问该结点右子树
}
```

# 二叉树的遍历

## 后序遍历算法

若二叉树为空树，则空操作；否则

后序遍历左子树

后序遍历右子树

访问根结点

# 二叉树的遍历

## 后序遍历算法

```
void POSTORDER ( bitree *r)
{
    if ( r == NULL ) return ;    //空树返回
    POSTORDER( r->lchild ); //先访问该结点的左子树
    POSTORDER( r->rchild ); //再访问该结点右子树
    printf ( " %c " ,r->data );    //最后访问当前结点
}
```

# 二叉树的遍历

遍历的路径相同，均为从根节点出发，逆时针沿二叉树的外缘移动，每个节点均经过三次。按不同的次序访问可得不同的访问系列，每个节点有它的逻辑前趋（父节点）和逻辑后继（子节点），也有它的遍历前趋和遍历后继（要指明遍历方式）。

# 二叉树的遍历

按编号遍历算法:

```
NOORDER ( bitree *r)                                /*按编号顺序遍历算法*/
{
    int front, rear;
    bitree *Q[N];
    if ( r == NULL ) return ;                          /*空树返回*/
    for (rear=1;rear<N; rear++) Q[rear] = NULL ;
    front = rear = 1; Q[rear] = r;
    while ( Q[front] != NULL ) {                        /*以下部分算法由学生完成设计*/

        /*访问当前出队节点*/

        /*若左孩子存在则左孩子入队*/

        /*若有孩子存在则右孩子入队*/

        /* front向后移动*/
    } }
```

# 最优二叉树

赫夫曼(Huffman)树，又称最优树，是带权路径长度最短的树，有着广泛的应用

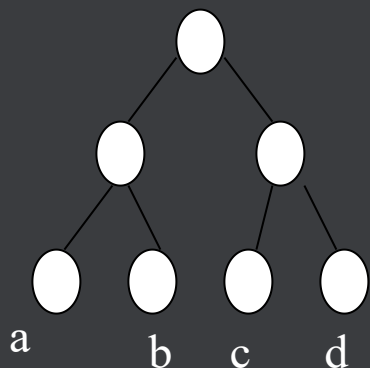
从树中一个结点到另外一个结点的分支构成一条路径，分支的数目称为路径的长度。树的路径长度是指从树根到每个结点的路径长度之和

进一步推广，考虑带权的结点。结点的带权路径长度指的是从树根到该结点的路径长度和结点上权的乘积。树的带权路径长度是指所有叶子结点的带权路径长度之和，记作  $WPL$ 。 $WPL$ 最小的二叉树就是最优二叉树，又称为赫夫曼树

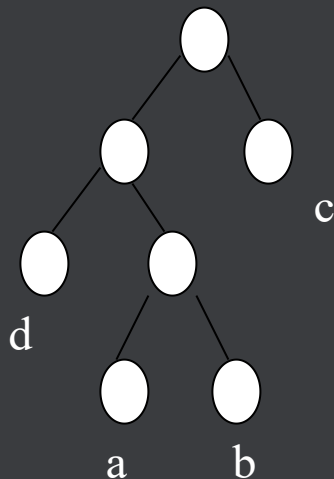


# 最优二叉树(赫夫曼树)

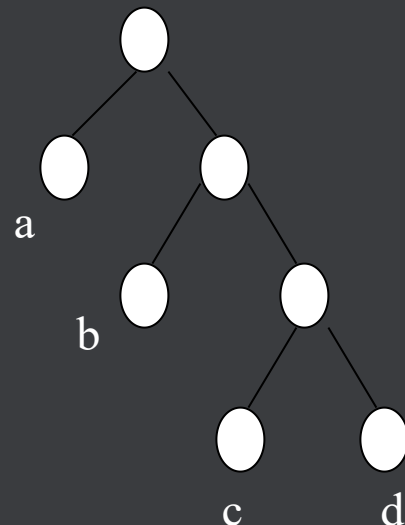
例如下面的三棵二叉树，都有四个叶子结点a, b, c, d，权值分别为7, 5, 2, 4。



$$WPL = 7*2 + 5*2 + 2*2 + 4*2$$

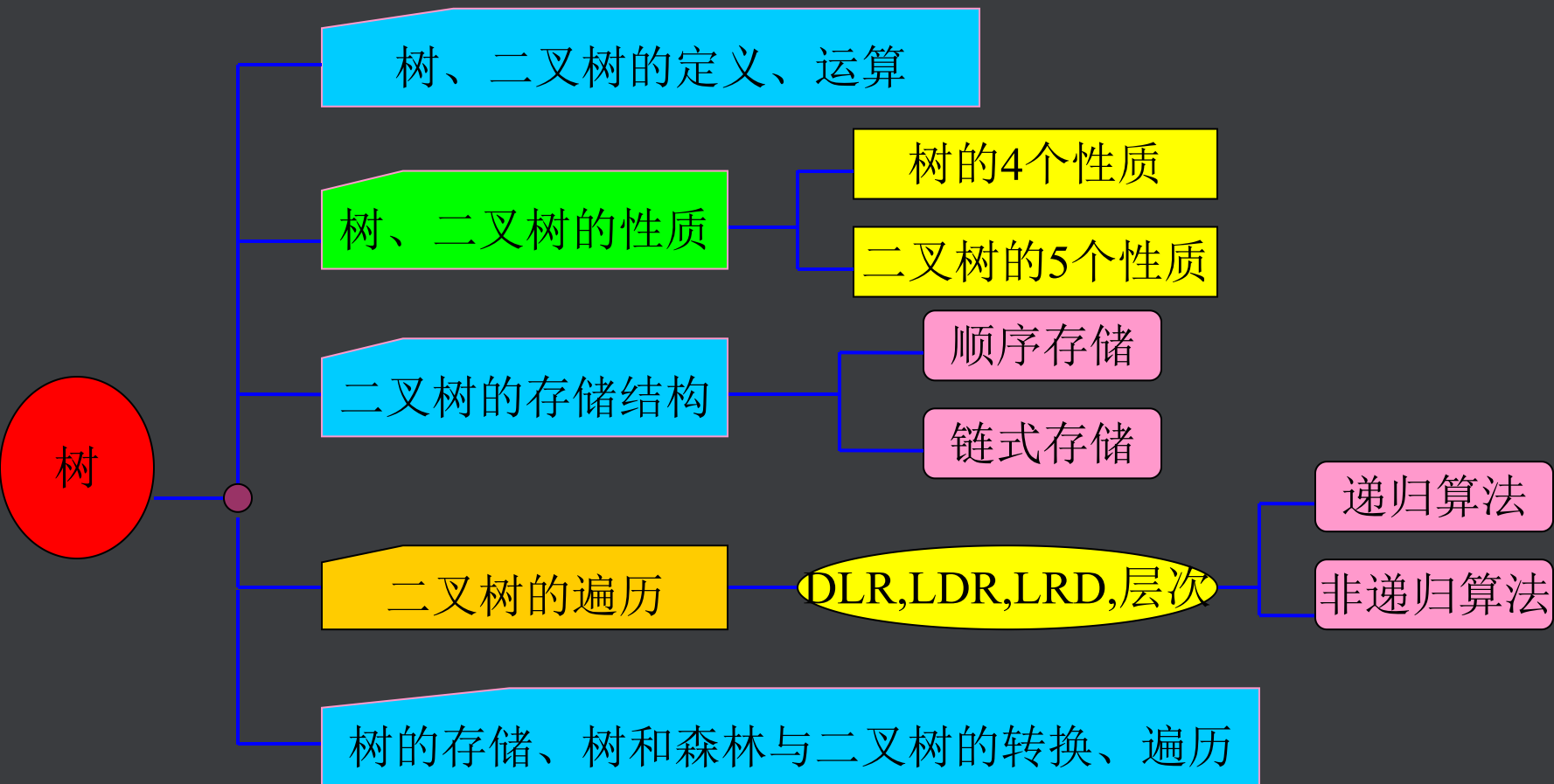


$$WPL = 7*3 + 5*3 + 2*1 + 4*2$$



$$WPL = 7*1 + 5*2 + 2*3 + 4*3$$

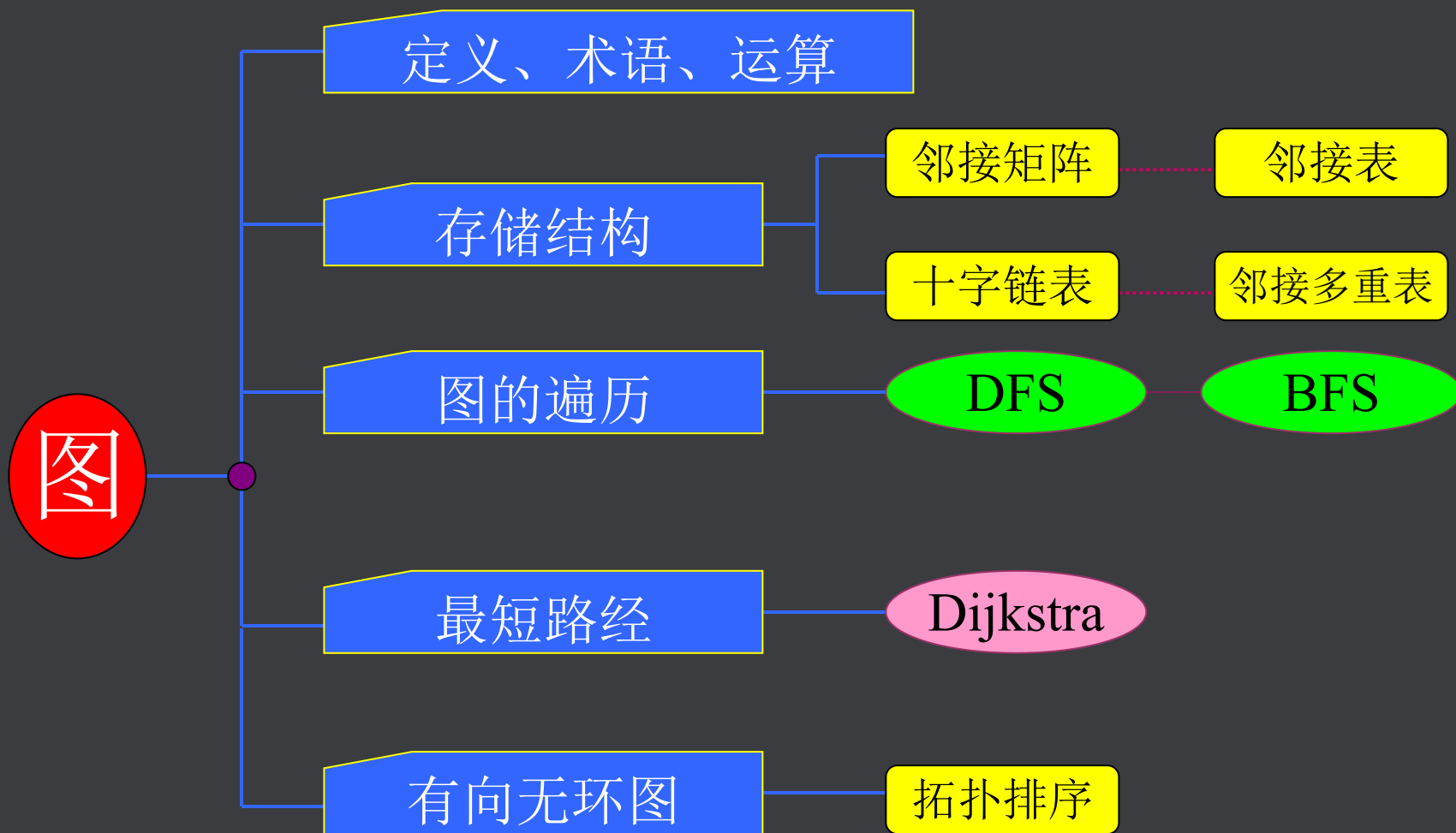
# 小结



5



# 知识点



# 5.1

## 图的定义 及运算

# 图的定义及运算

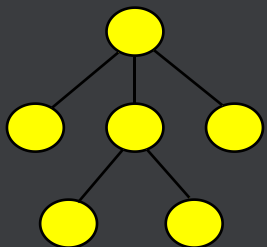
图是一种比线性表和树更为复杂的数据结构。

线性表:



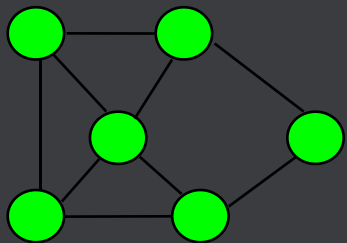
元素之间是线性关系，即表中元素最多一个直接前驱和一个直接后继；

树:



元素之间是层次关系。除根外，元素只有唯一直接前驱，但可以有若干个直接后继；

图:



任意的两个元素都可能相关，即图中任一元素可以有若干个直接前驱和直接后继，属于网状结构类型。

(实际上，树是图的特例——有向无环图。)

图的应用十分广泛，现已渗入到数学、物理、化学、语言学、逻辑学、电讯工程和计算机科学之中。特别是讨论计算机网络拓扑结构时，以图论作为理论基础。

# 图的定义及运算

定义:

图 (**Graph**) 是一种**非线性数据结构**, 形式化描述为:

其中,  $V = \{V_i \mid V_i \in \text{datatype}, i=0,1,\dots,n-1\}$  是图中元素  $V_i$  (称为**顶点Vertex**) 的集合,  $n=0$  时,  $V$  为空集, 记为  $\varphi$ 。

$R$  是顶点之间的关系集,  $P(V_i, V_j)$  为顶点  $V_i$  与  $V_j$  之间是否存在路径的判定条件, 即若  $V_i$  与  $V_j$  之间的路径存在, 则关系  $\langle V_i, V_j \rangle \in R$ 。

# 基本术语

基本术语:

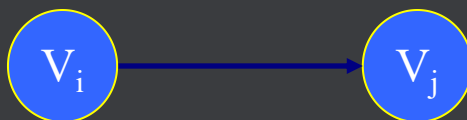
## 1) 有向图 (Digraph)

设  $V_i$ 、 $V_j$  为图中的两个顶点，若关系  $\langle V_i, V_j \rangle$  存在方向性，即：

$\langle V_i, V_j \rangle \neq \langle V_j, V_i \rangle$ ，则称相应的图为有向图。  $\langle V_i, V_j \rangle \in R$ ，

表示从顶点  $V_i$  到  $V_j$  的一条弧 (Arc)：

$\langle V_i, V_j \rangle$   $V_i$  为弧尾， $V_j$  为弧头。



## 2) 无向图 (Undigraph)

设  $V_i$ 、 $V_j$  为图中的两个顶点，若关系  $\langle V_i, V_j \rangle$  无方向性，即：

当  $\langle V_i, V_j \rangle \in R$  时，必有  $\langle V_j, V_i \rangle \in R$ ，则称此时的图为无向图。

关系用  $(V_i, V_j)$  或  $(V_j, V_i)$  表示，称为图中的一条边 (Edge)：

$(V_i, V_j)$





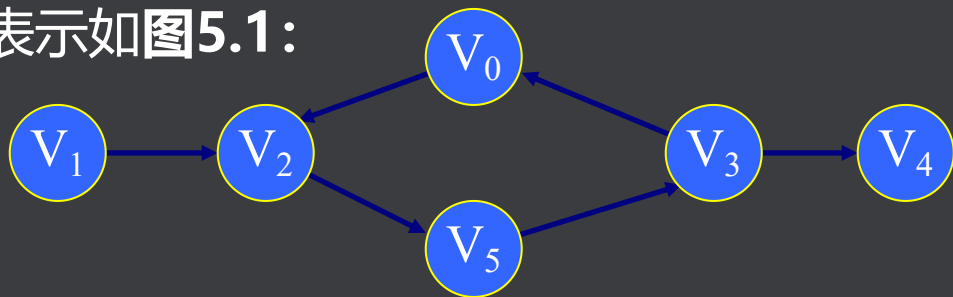
# 基本术语

例5-1 设有向图 $G_1 = (V, R)$

$$V = \{V_0, V_1, V_2, V_3, V_4, V_5\} (n = 6)$$

$$R = \{ \langle V_0, V_2 \rangle \langle V_1, V_2 \rangle \langle V_2, V_5 \rangle \langle V_3, V_0 \rangle \langle V_3, V_4 \rangle \langle V_5, V_3 \rangle \}$$

则 $G_1$ 的表示如图5.1:



值得注意的是，按照图的定义，图5.1中的顶点 $V_0$ 经过 $V_2$ 到达 $V_5$ ，存在一条路径 $(V_0, V_2, V_5)$ ，但 $G_1$ 的 $R$ 中并未出现 $\langle V_0, V_5 \rangle$ 这样的关系，这是因为在关系 $R$ 的等价闭包 $R'$ 中，应包括关系传递性，即若： $\langle V_i, V_j \rangle$ 、 $\langle V_j, V_k \rangle \in R$ ，则有 $\langle V_i, V_k \rangle \in R'$ 。

故图的关系集一般是取最小关系集，即只写出图中弧的条数即可。

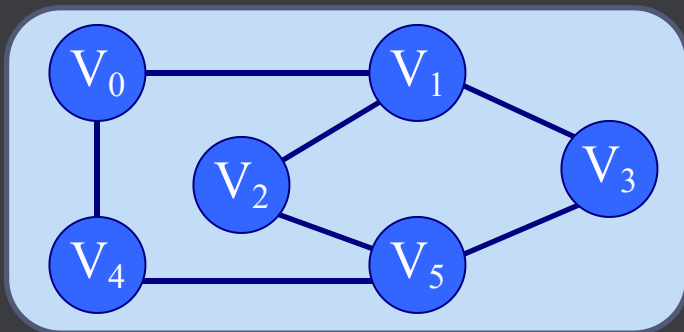
# 基本术语

例5-2 设无向图 $G_2 = (V, R)$

$$V = \{V_0, V_1, V_2, V_3, V_4, V_5\}$$

$$R = \{(V_0, V_1)(V_0, V_4)(V_1, V_2)(V_1, V_3)(V_2, V_5)(V_3, V_5)(V_4, V_5)\}$$

则 $G_2$ 的表示如图5.2：



设图中顶点集 $V = \{V_0, V_1, \dots, V_{n-1}\}$ ,  $e$ 为图中弧或边的条数, 若不考虑自反性: 即若 $\langle V_i, V_j \rangle \in R$ , 则 $V_i \neq V_j$ . 于是, 对于无向图,  $e$ 的范围是:

$$\left[ 0, \frac{1}{2}n(n-1) \right]$$

# 基本术语

其含义为：①图可以是一个个弧顶点的集合，边的条数可以为0；

②边数最多为：

$(V_0, V_1), (V_0, V_2), \dots, (V_0, V_{n-1})$  ——  $n-1$  条

$(V_1, V_2), \dots, (V_1, V_{n-1})$  ——  $n-2$  条

.....

$(V_{n-1}, V_{n-2})$  —— 1 条

$$\left[ 0, \frac{1}{2}n(n-1) \right]$$

有  $e = \frac{1}{2}n(n-1)$  条边的无向图称为**无向完全图**。

对于有向图，因为  $\langle V_i, V_j \rangle, \langle V_j, V_i \rangle \in R$  是图中两条不同的弧，故弧的条数  $e$  的范围是  $n(n-1)$ 。有  $[0, n(n-1)]$  条弧的有向图称为**有向完全图**。

(3) **稀疏图** (Sparse Graph) 和**稠密图** (Dense Graph)

弧或边的条数很少的图称为稀疏图，反之称为稠密图。

# 基本术语

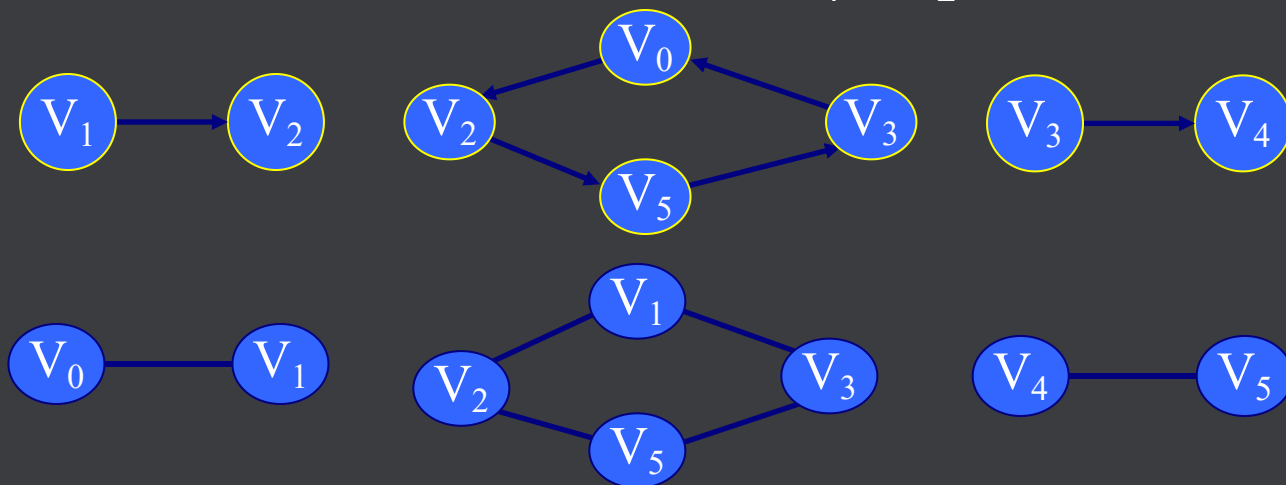
(4) **网** (Network) 若在图的关系  $\langle V_i, V_j \rangle$  或  $(V_i, V_j)$  上附加一个值  $w$ :



称  $w$  为弧或边上的权。带权的图称为网。权  $w$  的具体含义视图在不同领域的应用而定，如顶点表示城市，权  $w$  可以为两个城市间的距离等等。

(5) **子图** (Subgraph)

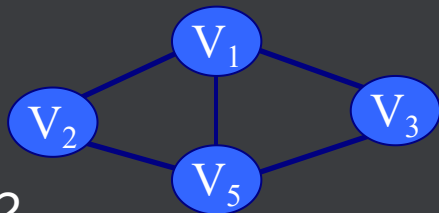
设图  $G = (V, R)$ 、 $G' = (V', R')$ ，若  $V' \subseteq V$  且  $R' \subseteq R$ ，则称  $G'$  为  $G$  的子图。例5-1、例5-2中  $G_1$  和  $G_2$  的一些子图如图5.3:



# 基本术语

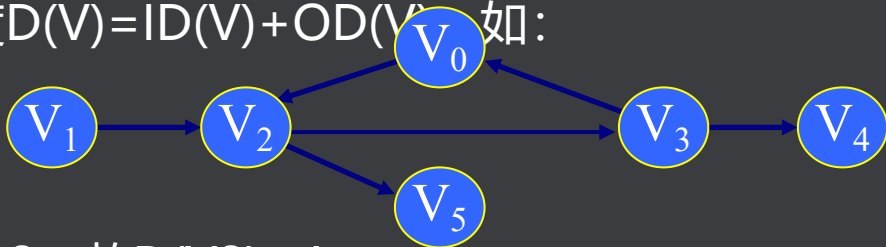
## 6) 顶点的度 (Degree)

设 $E$ 为无向图 $G$ 中边的集合,  $V$ 、 $V'$  为图中的顶点。若  $(V, V') \in E$ , 则称 $V$ 和 $V'$  互为邻接点, 或称 $V$ 与 $V'$  相邻接, 边  $(V, V')$  与 $V$ 、 $V'$  相关联。某顶点 $V$ 的度记为 $D(V)$ , 代表与 $V$ 相关联的边的条数。如:



$$D(V_1)=3, D(V_2)=2。$$

又设 $A$ 为有向图 $G$ 中弧的集合, 若  $\langle V, V' \rangle \in A$ , 则称 $V$ 邻接到 $V'$ ,  $V'$  邻接自 $V$ ,  $\langle V, V' \rangle$  与 $V$ 、 $V'$  相关联。顶点 $V$ 的入度记为 $ID(V)$ , 是图中以 $V$ 为弧头的弧的条数; 而顶点 $V$ 的出度记为 $OD(V)$ , 是图中以 $V$ 为弧尾的弧的条数。顶点 $V$ 的度 $D(V)=ID(V)+OD(V)$ 。如:



$$ID(V_2)=2, OD(V_2)=2, \text{ 故 } D(V_2)=4。$$

# 基本术语

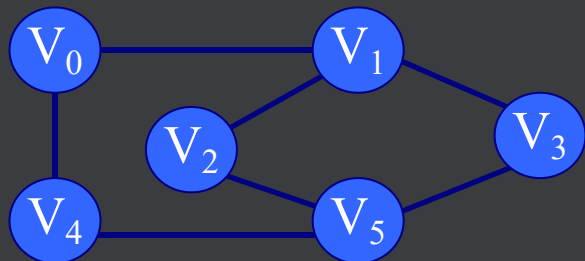
若图中顶点数为 $n$ ，边或弧的条数为 $e$ ，则：

$$e = \frac{1}{2} \sum_{i=0}^{n-1} D(V_i)$$

即图中所有顶点的度之和为 $e$ 的两倍。这是显然的，因为不管是无向图还是有向图，在计算某一顶点的度时，相关联的边或弧都被统计了两次。

如：边的条数：

$$(D(V_0) + D(V_1) + D(V_2) + D(V_3) + D(V_4) + D(V_5))/2$$



$$(2+3+2+2+2+3)/2=7。$$

# 基本术语

## (7) 路径 (Path)

若从顶点 $V$ 出发，经过某些顶点能到达另一顶点 $V'$ ，则称 $V$ 与 $V'$ 之间存在一条**路径**。由于图中两顶点间的关系是任意的，且可能存在回路，故两顶点间可能存在多条路径（树中根到叶结点的路径是唯一的）。无向图中，顶点 $V$ 到 $V'$ 的第 $i$ 条路径是一个顶点序列，记为 $(V=V_{i1}, V_{i2}, \dots, V_{im}=V')$ ，满足 $(V_{ij}, V_{ij+1}) \in E$ （ $E$ 是图中边的集合， $1 \leq j \leq m-1$ ）。路径上边的条数定义为该路径的长度。如例5-2的 $G_2$ 中，路径 $(V_0, V_1, V_2, V_5, V_3)$ 的长度为4。

有向图中，两顶点间的路径也是有向的。如例5-1的 $G_1$ 中，路径 $(V_0, V_2, V_5, V_3, V_4)$ 的长度为4。

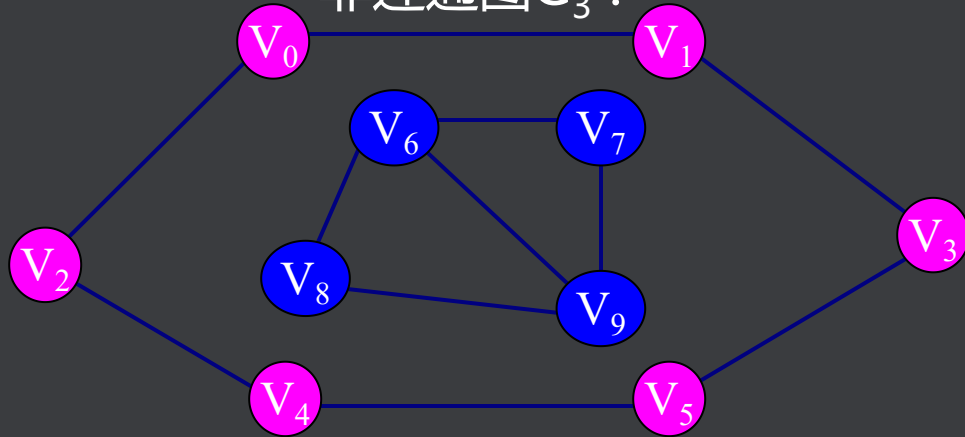
另外，若路径 $(V_{i1}, V_{i2}, \dots, V_{im})$ 中顶点不重复出现，则称其为**简单路径**；若路径中只有第一顶点 $V_{i1}$ 与最后一个顶点 $V_{im}$ 相同，则称其为**简单回路**或**简单环 (Cycle)**。如例5-2的 $G_2$ 中， $(V_0, V_1, V_2, V_5, V_4)$ 为简单路径，而 $(V_0, V_1, V_2, V_5, V_4, V_0)$ 为简单环；但 $(V_0, V_1, V_3, V_5, V_2, V_1, V_0)$ 为非简单环。

# 基本术语

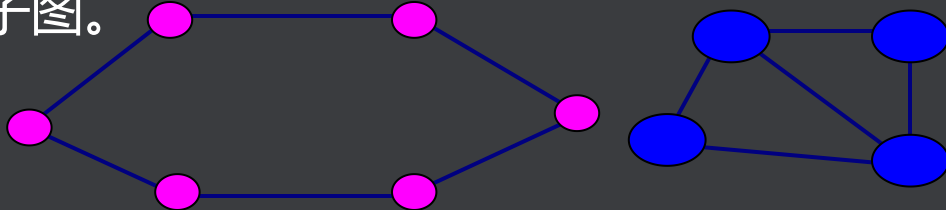
## (8) 连通性 (Connection)

在无向图中，两顶点 $V$ 与 $V'$ 间存在路径，称 $V$ 与 $V'$ 是连通的；若图中任意两顶点都连通，则称该图为无向连通图。下面举一个非连通图的例子，设某无向图 $G_3$ 如图5.4所示

非连通图 $G_3$ ：



但 $G_3$ 存在两个连通分量 (Connected Component)：图中极大的连通子图。

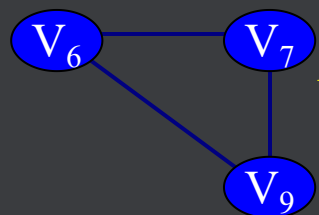




# 基本术语

“极大”在这里指的是：往一个连通分量中再加入顶点和边，就构不成原图中的一个连通子图，即连通分量是一个最大集的连通子图。

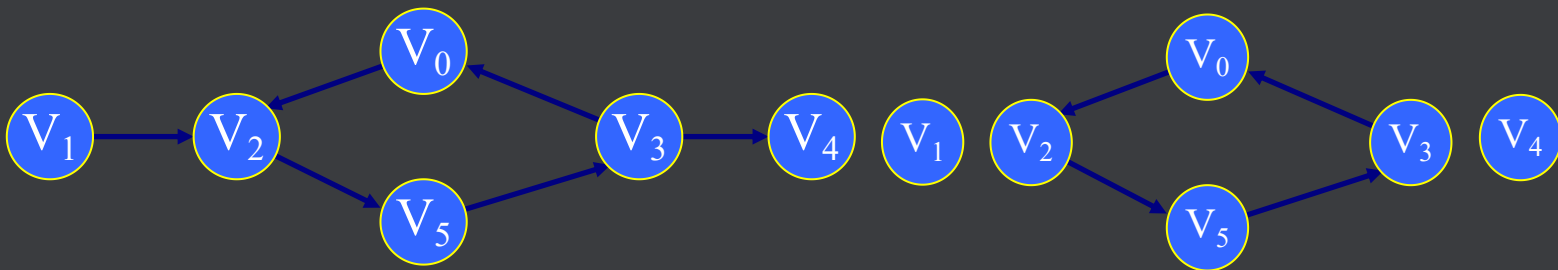
如图5.4 中子图：



非连通分量。因加入 $V_8$ 仍能构成原图中的一个子图。

在有向图中，若图中任意两顶点间都存在路径，则称其是强连通图。

**图中极大强连通子图称之为强连通分量**（其极大性同无向图）。如例5-2的 $G_1$ 显然是非强连通图，但它存在三个强连通分量，如图5.6所示。



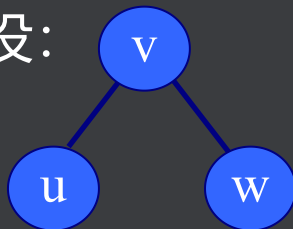
# 图的基本运算

顶点在图中的“位置”，不象表和树那样严格规定，可以根据顶点的输入序列随机确定。当然，顶点一旦输入到计算机中，它的位置（或地址、下标）就唯一确定。另外，某顶点的邻接点次序也是人为确定的。据此，图的基本运算如下：

- (1) 建立一个图：CreateGraph(G) 按照输入的顶点集和关系集，在计算机中建立图G的存储结构。
- (2) 定位：LocateVex(G, v) 若顶点 $v \in G$ ，则返回v在图中的位置（地址或序号），否则返回-1或NULL。
- (3) 取第i顶点：GetVex(G, i) 取图G中顶点 $V_i$ ，若G中不存在第i顶点，则返回NULL。
- (4) 取第一邻接点：FirstAdj(G, v) 求图G中顶点v的第一邻接点的地址（或序号），若v为孤结点或 $v \notin G$ ，则返回-1或NULL。

# 图的基本运算

(5) **取下一邻接点**:  $\text{NextAdj}(G, v, u)$  求图G中顶点v关于顶点u的下一邻接点地址 (或序号)。设:



则w的地址为所求。若w不存在, 则返回-1或NULL。

(6) **插入顶点**:  $\text{InsertVex}(G, v)$  将顶点u插入图G中。

(7) **插入弧或边**:  $\text{InsertArc}(G, v, u)$  在G中增加一条弧 $\langle v, u \rangle$ 或边 $(v, u)$

(8) **删除顶点**:  $\text{DeleteVex}(G, v)$  删除图G中的顶点v, 且与v相关联的弧或边也一并删除。若 , 则出错处理。

(9) **删除弧或边**:  $\text{DeleteArc}(G, v, u)$  删除图G中的一条弧 $\langle v, u \rangle$ 或边 $(v, u)$ 。若v与u不关联, 则出错处理。

(10) **遍历**:  $\text{TraverseGraph}(G)$  按某种规则 (或次序) 将图G中各顶点访问且仅访问一次的操作。

5.2

## 图的存储结构

# 图的存储结构

由于图中顶点间的关系（弧或边）无规律，故对图的存储较之表和树要复杂些，需要根据图的具体应用来构造图的存储结构。常用的存储表示有“数组表示法”、“邻接表”、“十字链表”和“邻接多重表”。

## 数组表示法

图的数组表示法又称“邻接矩阵”（Adjacency Matrix）表示法。设图：

$$G = (V, R)$$

用两个数组来存储图G：一个数组(一维)存储G中顶点集V；另一个数组(二维)映象图中顶点间的关系集R，该二维数组就是所谓的邻接矩阵。

邻接矩阵是一个n阶方阵：

其中：

$$A[n][n] = \begin{pmatrix} a_{00} & a_{01} & \dots & a_{0j} & \dots & a_{0n-1} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ a_{i0} & a_{i1} & \dots & a_{ij} & \dots & a_{in-1} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ a_{n-10} & a_{n-11} & \dots & a_{n-1j} & \dots & a_{n-1n-1} \end{pmatrix} \quad a_{ij} = \begin{cases} 1 & \langle V_i, V_j \rangle \in R \text{ 或 } (V_i, V_j) \in R \\ 0 & \text{否则} \end{cases}$$

# 图的存储结构

例5-4 设有向图 $G_5$ 如下:

| $V_0$ | $V_1$ | $V_2$ | $V_3$ | $V_4$ |
|-------|-------|-------|-------|-------|
| 0     | 1     | 2     | 3     | 4     |

$G_5$ 的数组表示法为:

$$A[5][5] = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

例5-5 设无向图 $G_6$ 如下:

$G_6$ 的数组表示法为:

$$A[5][5] = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

# 图的存储结构（数组表示法）

对无向图而言,  $(V_i, V_j) \in R$  时, 必有  $(V_j, V_i) \in R$ , 故其邻接矩阵是对称的。

邻接矩阵反映了图的逻辑结构, 即若  $a_{ij}=1$ , 则关系  $\langle V_i, V_j \rangle$  或  $(V_i, V_j)$  存在, 否则不存在。

另外, 利用邻接矩阵A可以求出图中顶点的度。

对无向图, 顶点  $V_i$  的度  $D(V_i)$  = 与  $V_i$  相关联的边的条数 = **A中第i行非0元素之和,**

} 如例5-5的  $G_6$  中,  $D(V_1)=3$  ;

对有向图, 顶点  $V_i$  的出度  $OD(V_i)$  为 **A中第i行非0元素之和**, 而顶点  $V_i$  的入度  $ID(V_i)$  为 **A中第i列的非0元素之和**。

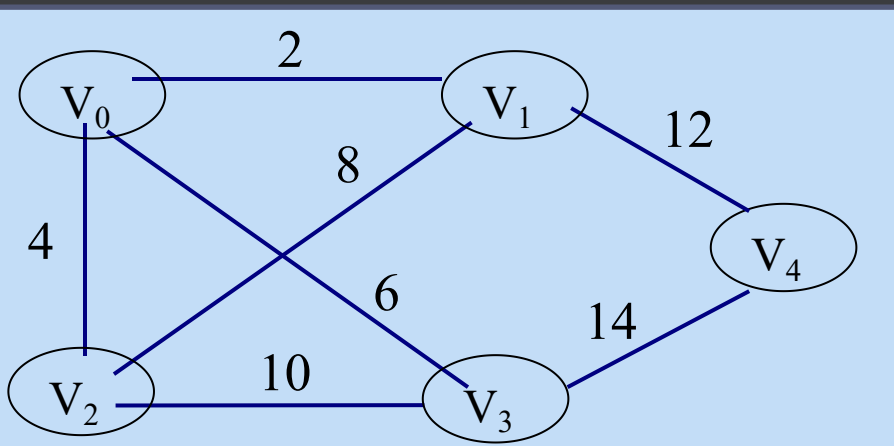
} 如例5-4的  $G_5$  中,  $OD(V_1)=1$ ,  $ID(V_1)=2$ 。

对于网的邻接矩阵A, 元素  $a_{ij}$  取值为:

$$\boxed{a_{ij}} = \begin{cases} w & \langle V_i, V_j \rangle \in R \text{ 或 } (V_i, V_j) \in R \text{ 且关系上的权} = w \\ \infty & \text{否则} \end{cases}$$

# 图的存储结构（数组表示法）

例5-6 设无向网 $G_7$ 如下:



$G_7$ 的数组表示法为:

| $V_0$ | $V_1$ | $V_2$ | $V_3$ | $V_4$ |
|-------|-------|-------|-------|-------|
|-------|-------|-------|-------|-------|

$$A = \begin{bmatrix} \infty & 2 & 4 & 6 & \infty \\ 2 & \infty & 8 & \infty & 12 \\ 4 & 8 & \infty & 10 & \infty \\ 6 & \infty & 10 & \infty & 14 \\ \infty & 12 & \infty & 14 & \infty \end{bmatrix}$$



# 数组表示法的C语言描述

```
#define MAXN 64           //最大顶点数//
typedef char vtype;       //设顶点为字符类型//
typedef int adjtype;      //设邻接矩阵A中元素 $a_{ij}$ 为整型//
typedef struct
{
    vtype V[MAXN];        //顶点存储空间//
    adjtype A[MAXN][MAXN]; //邻接矩阵//
} mgraph;
```

若说明：`mgraph G`；则G为存储图的一个结构体变量，`G.V[MAXN]`为顶点的存储空间，而`G.A[MAXN][MAXN]`为邻接矩阵。

数组表示法存储结构的建立算法比较简单：读入顶点和关系集（弧、边），建立顶点表和邻接矩阵即可。

```
void createmgraph(mgraph G) //建立无向网的数组表示法//
{
    int i, j, n;
    vtype ch, u, v;
    adjtype w;
    i = n = 0;
    ch=getchar(); //输入顶点//
```

# 数组表示法的C语言描述

```
while (ch!= '#' ) Ⅵ#为结束符Ⅵ
{  n++;    Ⅵ顶点计数Ⅵ
  if (n>MAXN-1) ERROR(n); Ⅵ溢出处理Ⅵ
  G.V[i++]=ch; ch=getchar( ); }Ⅵ存入顶点, 并读下一顶点Ⅵ
  for (i=0; i<n; i++)    Ⅵ初始化邻接矩阵Ⅵ
    for (j=0; j<n; j++)
      G.A[i][j]=max; Ⅵ设max为机器表示的 $\infty$ Ⅵ
  scanf( "%c %c %d" , &u, &v, &w); Ⅵ读入一条边<u, v>及权值 Ⅵ
  while (u != '#' )    Ⅵu= '#' 时结束Ⅵ
  {  i = locatevex(G,u); Ⅵ求u的序号Ⅵ
    j = locatevex(G,v); Ⅵ求v的序号Ⅵ
    G.A[i][j] = G.A[j][i] = w; Ⅵ邻接矩阵赋值 (对称) Ⅵ
    scanf ( "%c %c %d" ,&u,&v,&w); }Ⅵ读下一条边Ⅵ
  }
}
```

Ⅵ 设图中顶点数为n, 边的条数为e。第一个while循环执行次数为n;  
后两个for循环的执行次数约为 $n^2$ ; 最后一个while循环执行次数为e; 故算法  
的时间复杂度为 $T(n,e)=O(n^2+e)$ 。若 $n^2 \gg e$ , 则时间复杂度为 $O(n^2)$ 。

# 图的存储结构（邻接表表示法）

## 邻接表

所谓邻接表（Adjacency Lists），是将图中每一顶点 $V$ 和由 $V$ 发出的弧或边构成单链表，映象图的逻辑关系。邻接表是图的一种链式存储结构，类似树的孩子链表示法。

**链表结点：**表示某顶点 $V_i$ 到另一顶点 $V_j$ 的一条弧(或边)，如图5.12所示。

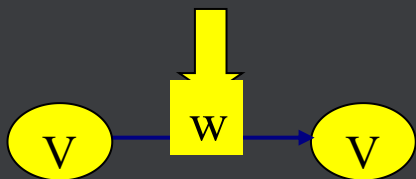


图 5.12

其中，adj为邻接点域，存放与 $V_i$ 相邻接的顶点 $V_j$ 的序号；w为弧或边上的权；next为指向与 $V_i$ 相邻接的下一条弧或边结点的指针。

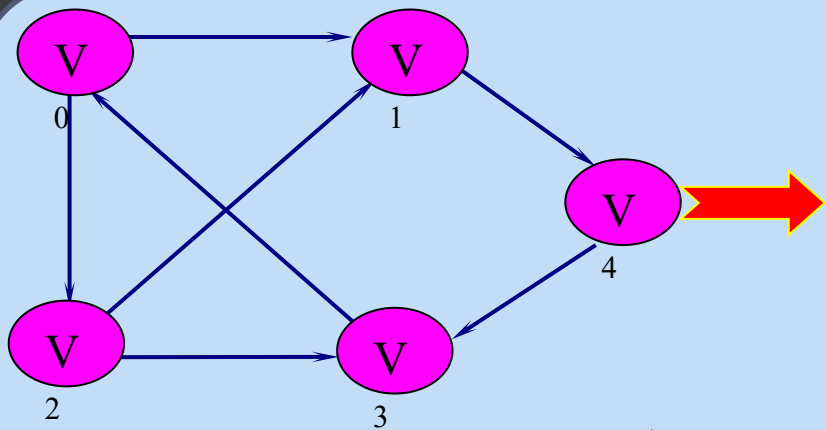
**顶点结点：**



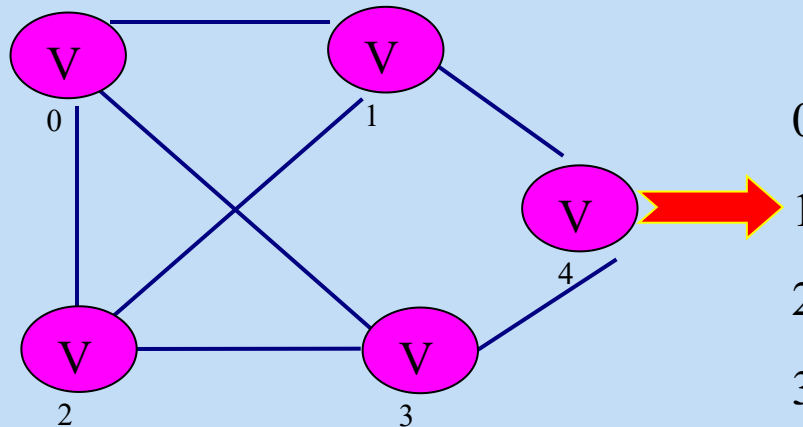
其中，data域存储顶点值；farc为指向该顶点发出的第一条弧或边结点的指针。另外，将所有结点组织成顶点表。

# 图的存储结构（邻接表表示法）

例5-4和例5-5中 $G_5$ 、 $G_6$ 的邻接表如图5.13 (a) 和 (b) 所示（权值省略）。



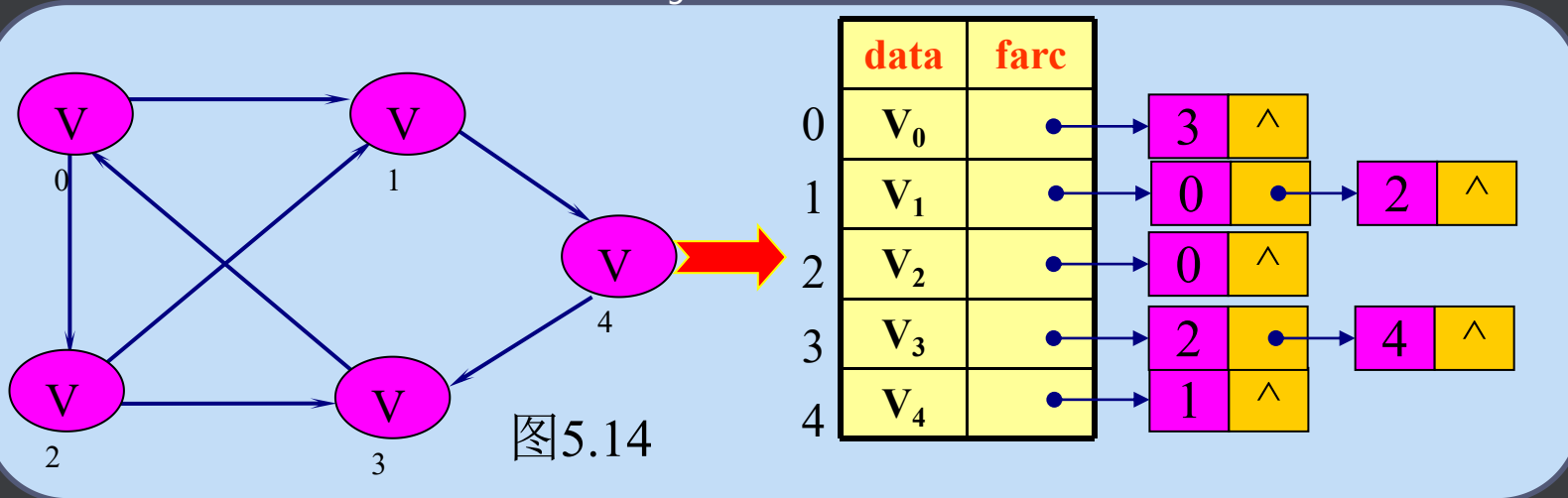
|   | data  | farc  |
|---|-------|---|
| 0 | $V_0$ | <div><div>1</div><div>2</div><div>^</div></div> |
| 1 | $V_1$ | <div><div>4</div><div>^</div></div>             |
| 2 | $V_2$ | <div><div>1</div><div>3</div><div>^</div></div> |
| 3 | $V_3$ | <div><div>0</div><div>^</div></div>             |
| 4 | $V_4$ | <div><div>3</div><div>^</div></div>             |



|   | data  | farc  |
|---|-------|---|
| 0 | $V_0$ | <div><div>1</div><div>2</div><div>3</div><div>^</div></div> |
| 1 | $V_1$ | <div><div>0</div><div>2</div><div>4</div><div>^</div></div> |
| 2 | $V_2$ | <div><div>0</div><div>1</div><div>3</div><div>^</div></div> |
| 3 | $V_3$ | <div><div>0</div><div>2</div><div>4</div><div>^</div></div> |
| 4 | $V_4$ | <div><div>1</div><div>3</div><div>^</div></div>             |

# 图的存储结构（邻接表表示法）

对于有向图的邻接表， $OD(V_i)=V_i$ 所在链表中结点的个数，如5.13(a)中， $OD(V_2)=2$ ；而 $ID(V_i)$ =所有链表中 $adj=i$ 的结点个数，如 $ID(V_2)=1$ 。为方便确定 $V_i$ 的入度，可以建立有向图的逆邻接表，如例5-4中 $G_5$ 的逆邻接表如图5.14所示。



于是， $ID(V_i)$ =逆邻接表中第 $i$ 个链表的链结点个数。但仅仅为了求得 $ID(V_i)$ 而另建一套存储结构未免浪费存储空间。可以将邻接表和逆邻接表合二为一，构成所谓的“十字链表”。

对于无向图的邻接表，某顶点 $V_i$ 的 $D(V_i)=V_i$ 所在单链表中结点的个数，如图5.13 (b)中， $D(V_2)=3$ 。因为无向图中每条边牵涉到两个链结点，所以当边数为 $e$ 时，链表结点数为 $2e$ 。可以对无向图的邻接表加以改进，构成所谓的“邻接多重表”，以减少链

5.3

图的遍历

# 图的存储结构

图的遍历是树的遍历的推广，是按照某种规则（或次序）访问图中各顶点一次且仅一次的操作，亦是将网状结构按某种规则线性化的过程。由于图存在回路，为区别一顶点是否被访问过和避免顶点被多次访问，在遍历过程中，应记下每个访问过的顶点，即每个顶点对应有一个标志位，初始为False，一旦该顶点被访问，就将其置为True，以后若又碰到该顶点时，视其标志的状态，而决定是否对其访问。

对图的遍历通常有“**深度优先搜索**”和“**广度优先搜索**”方法，二者是人工智能(AI)的一个基础。

# 深度优先搜索算法

深度优先搜索 (Depth First Search, 简称**DFS**)

## 1. 算法思路

类似树的先根遍历。设初始时，图中各顶点均未被访问，从图中某顶点（设为 $V_0$ ）出发，访问 $V_0$ ，然后搜索 $V_0$ 的一个邻接点 $V_i$ ，若 $V_i$ 未被访问，则访问之，再搜索 $V_i$ 的一个邻接点（深度优先）……。若某顶点的邻接点全部访问完毕，则回溯（Backtracking）到它的上一顶点，然后再从此顶点又按深度优先的方法搜索下去，……，直到能访问的顶点都访问完毕为止。



# 深度优先搜索算法

例5-9 设有一无向图 $G_{10}$ ，其DFS搜索过程如图5.20所示。

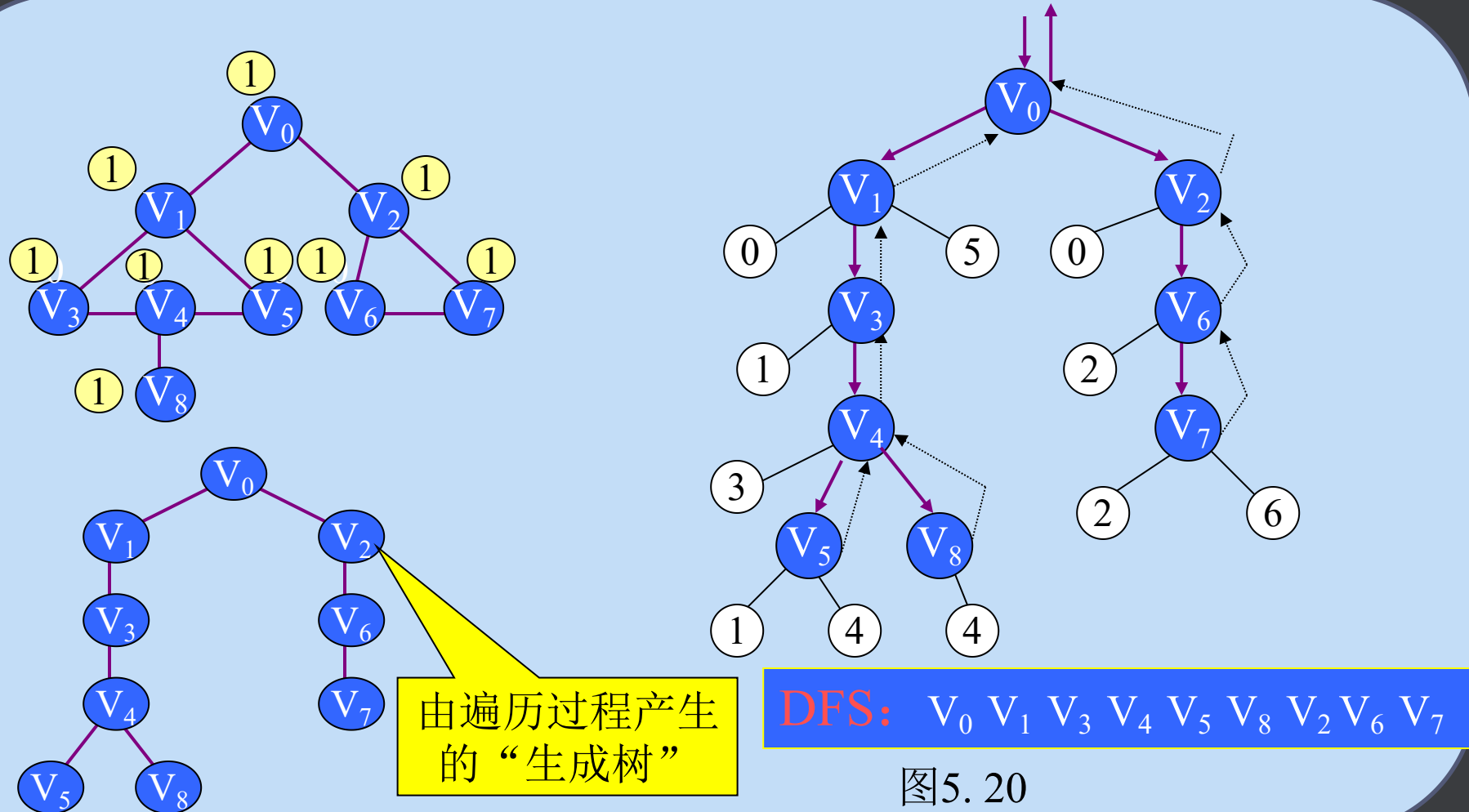


图5.20

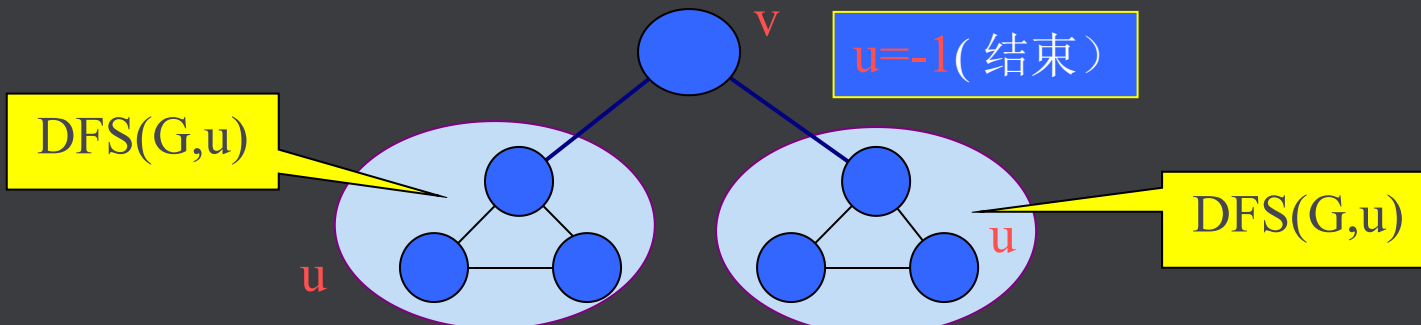
# 深度优先搜索算法

## 2.算法描述

设标志数组**Visited[n]** (n为当前图中的顶点数) 的初值为False (或0) ; 图采用邻接表表示法 (或数组表示法) 存储。

void DFS(int G[n][n], int v) // 对图G从序号为v的顶点出发, 按**DFS**方法搜索 //

```
{  int u;
    visit(G, v);           // 访问v号顶点 //
    visited[v] = True;     // 置标志位为True或1 //
    u = firstadj(G, v);    // 取v的第一邻接点序号u //
    while (u >= 0)         // 当u存在时 //
    { if(visited[u] == False) DFS(G, u);           // 若u未访问,调用函数
遍历从u 出发的子图 //
      u = nextadj(G, v, u);    // 取v关于当前u的下一邻接点序号 //
    }
}
```



# 广度优先搜索算法

广度优先搜索（Breadth First Search），简称**BFS**。

## 1. 算法思路

类似树的按层次遍历。初始时，图中各顶点均未被访问，从图中某顶点（设 $V_0$ ）出发，访问 $V_0$ ，并依次访问 $V_0$ 的各邻接点（广度优先）。然后，分别从这些被访问过的顶点出发，仍按照广度优先的策略搜索其它顶点，……，直到能访问的顶点都访问完毕为止。

为控制广度优先的正确搜索，要用到队列技术，即访问完一个顶点后，让该顶点的序号进队。然后取相应队头（出队），考察访问过的顶点的各邻接点，将未访问过的邻接点访问后再依次进队，……，直到队空为止。

# 广度优先搜索算法

对例5-9中 $G_{10}$ ，从 $V_0$ 出发，按**BFS**方法搜索的过程及生成树如图5.21所示。

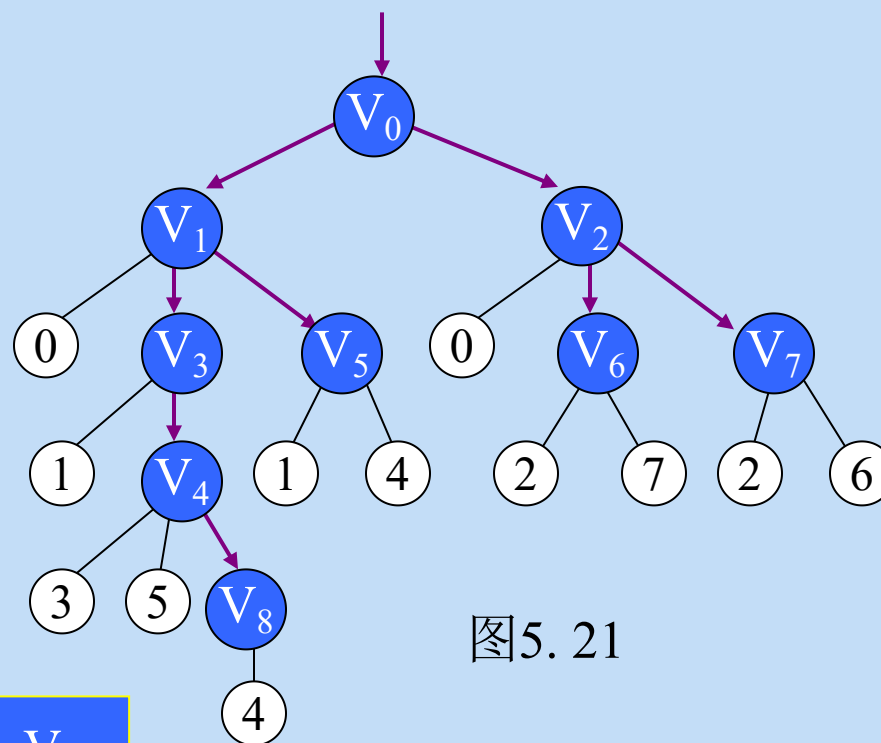
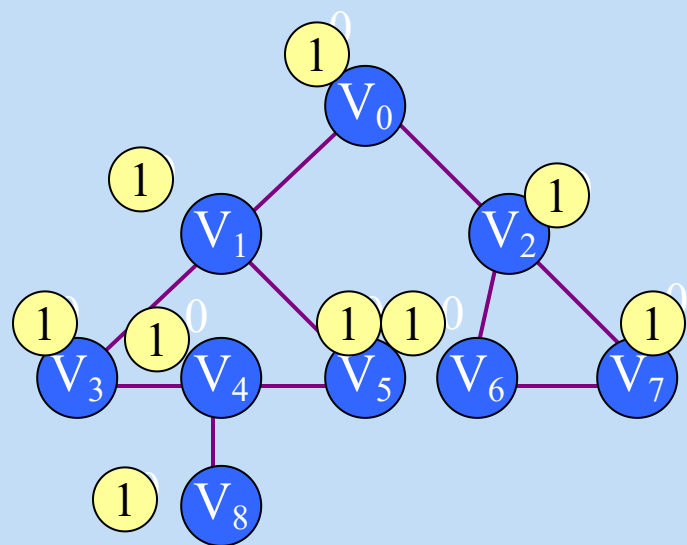


图5.21

**BFS:**  $V_0 V_1 V_2 V_3 V_5 V_6 V_7 V_4 V_8$

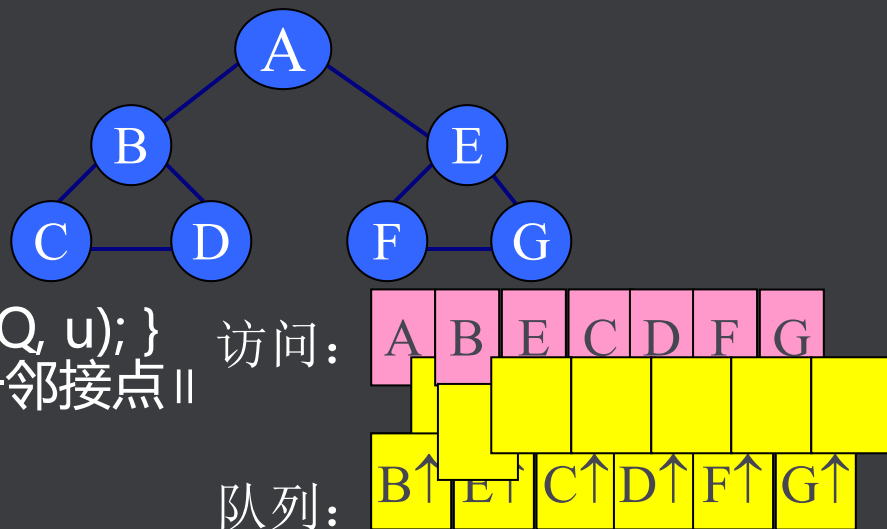
# 广度优先搜索算法

## 2. 算法描述

void BFS(int G[n][n], int v) // 对图G从序号为v的顶点出发，按BFS遍历 //

```
{ int u; qtype Q;
  Clearqueue(Q); // 置队Q为空 //
  visit(G, v); visited[v] = True; // 访问顶点、置标志为“真” //
  Enqueue(Q, v); // v进队 //
  while( !Emptyqueue(Q) ) // 队非空时 //
  { v = Delqueue(Q); // 出队，队头送v //
    u = firstadj(G, v); // 取v的第一邻接点序号 //
    while (u >= 0)
    { if (visited[u] == False)
      // 若u未访问，则访问后进队 //
      { visit(G, u); visited[u] = True; Enqueue(Q, u); }
      u = nextadj(G, v, u); // 取v关于u的下一邻接点 //
    }
  }
}
```

例：



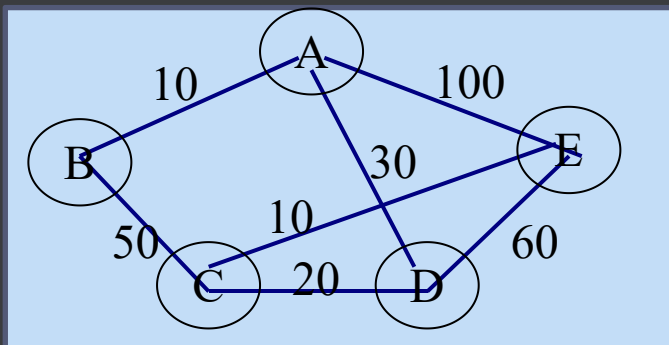
5.4

## 最短路径问题

# 最短路径问题

在开发一个交通咨询系统时，它对应的数据结构是一个网络，如图5.32的

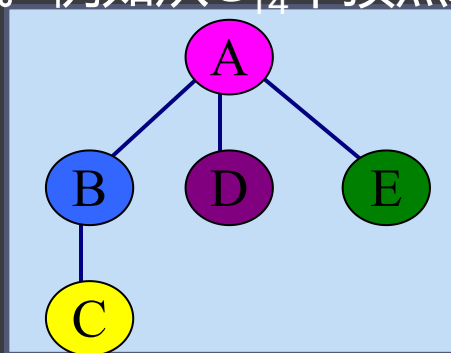
$G_{14}$ ：



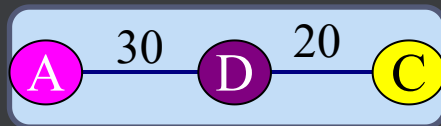
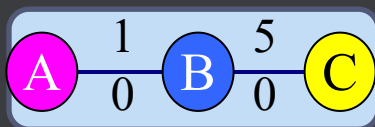
如图5.32

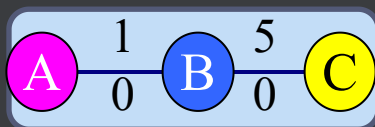
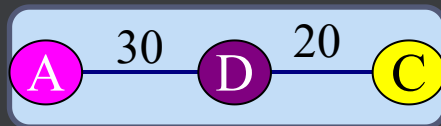
其中，顶点A~E表示城市，边上权可为两城市间的里程、乘车速度、耗费等等。咨询系统要解决的基本问题是：从某城市到另一城市，如何选择一条周转次数最少的路线。即如何寻找从一顶点到另一顶点所含边数最少的路径。可以从指定的顶点出发，对图作广度优先搜索，一旦遇到目标顶点就终止，由此得出两点间边数最少路径。例如从 $G_{14}$ 中顶点A出发，寻找到达顶点C的边数最少路径，其BFS为：

故从A到C边数最少路径为 (A,B,C)，  
周转次数最少为1。



# 最短路径问题



显然，从  改为从  为好，即除了考虑周转次数外，还应考虑里程、费用、速度等问题，它们是赋在边上的权。此时，两顶点间路径长度 $L$ 定义为**路径上边的权值之和**。两顶点间可能存在多条路径，**路径长度最短的那条路径为最短路径** (Shortest Path)。

如路径  $(A,B,C)$  的 $L=60$ ，而  $(A,D,C)$  的 $L=50$ ，显然A到C的最短路径为  $(A,D,C)$ ，称A为源点，C为终点。

交通图一般是有向的，因为就一条路线而言，上下坡、水路的逆顺水、时速等情况是不同的。

解决带权有向图中两顶点间最短路径问题有两个经典算法，分别为**Dijkstra (迪杰斯特拉) 算法**和**Floyd (弗洛伊德) 算法**。



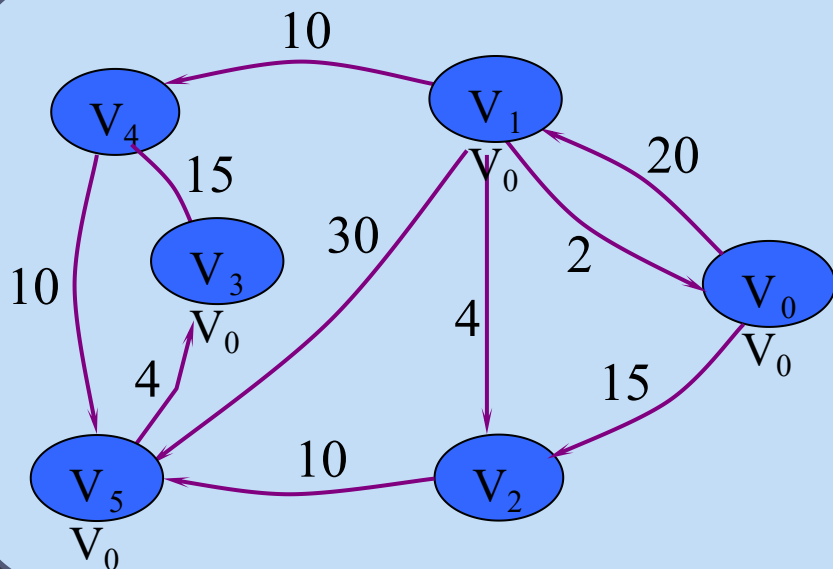
# Dijkstra算法

Dijkstra算法是解决从网络中任一顶点（源点）出发，求它到其它各顶点（终点）的最短路径问题（或单源点最短路径问题）。

## 1. 算法思路

按路径长度递增次序产生从某源点V到图中其余各顶点的最短路径。

例5-13 设有向网 $G_{15}$ 如下：



从 $V_0$ 出发，到其余各顶点的最短路径及长度L分别为：

$(v_0, v_2)$        $L=15$

$(v_0, v_1)$        $L=20$

$(v_0, v_2, v_5)$        $L=25$

$(v_0, v_2, v_5, v_3)$        $L=29$

$(v_0, v_1, v_4)$        $L=30$

（递增）

# Dijkstra算法

设网**G**用邻接矩阵**G.A[n][n]**表示，**n**为当前**G**中顶点数。为方便，各顶点用其序号0, 1, 2.....**n** - 1代替（即 $V_i=i$ ），并对邻接矩阵中元素作适当调整：

$$G.A[i][j] = \begin{cases} w_{ij} & \text{若 } \langle v_i, v_j \rangle \in R \text{ 且 } \langle v_i, v_j \rangle \text{ 上的权} = w_{ij}; \\ \infty & \text{若 } \langle v_i, v_j \rangle \notin R; \\ 0 & \text{若 } i = j. \end{cases}$$

即认为顶点到其自身的路径长度**L**为0。

# Dijkstra算法

引进几个辅助向量:

①**向量S[n]**: 其中

$$S[i] = \begin{cases} 1 & \text{当源点 } v \text{ 到 } v_i \text{ 的最短路径求出时;} \\ 0 & \text{否则} \end{cases}$$

初始令 $S[v]=1$  (即路径 $(v,v)$ 已求出,  $L=0$ ),  $S[i]=0$  ( $0 \leq i \leq n-1, i \neq v$ ), 表示 $v$ 到其它顶点的最短邻接未求出。

②**向量dist[n]**: 其中 $dist[i]$ 存放从 $v$ 到 $v_i$ 的最短路径长度( $0 \leq i \leq n-1$ )。

初始为:

$$dist[i] = \begin{cases} \langle v, v_i \rangle \text{ 上的权 } w & \text{若 } \langle v, v_i \rangle \in R \\ \infty & \text{若 } \langle v, v_i \rangle \notin R \end{cases}$$

若 $\langle v, v_i \rangle \in R$ , 则 $v$ 到 $v_i$ 的路径一定存在, 但不一定是最短的。

# Dijkstra算法

③**向量path[n]**：其中path[i]存放从源点v到 $v_i$ 的最短路径 ( $v, \dots, v_i$ )。初始path[i]={v} (表示从v出发)。

显然，从源点v到其它各顶点的第一条最短路径长度dist[u] (u为最短路径的终点)，可根据dist[n]的初始值决定，即：

$$\text{dist}[u] = \min\{\text{dist}[w] \mid w=0, 1, \dots, n-1 \text{ 且 } s[w]=0\}$$

表示在所有未求出的当前最短路径中找出一条最短的，其长度作为当前求出的最短路径长度。

当某条最短路径的终点u求出后，考察所有未求出的最短路径，即对所有 $s[w]=0$ 的w，若 $\text{dist}[w] > \text{dist}[u] + \langle u, w \rangle$ 上的权，则令：

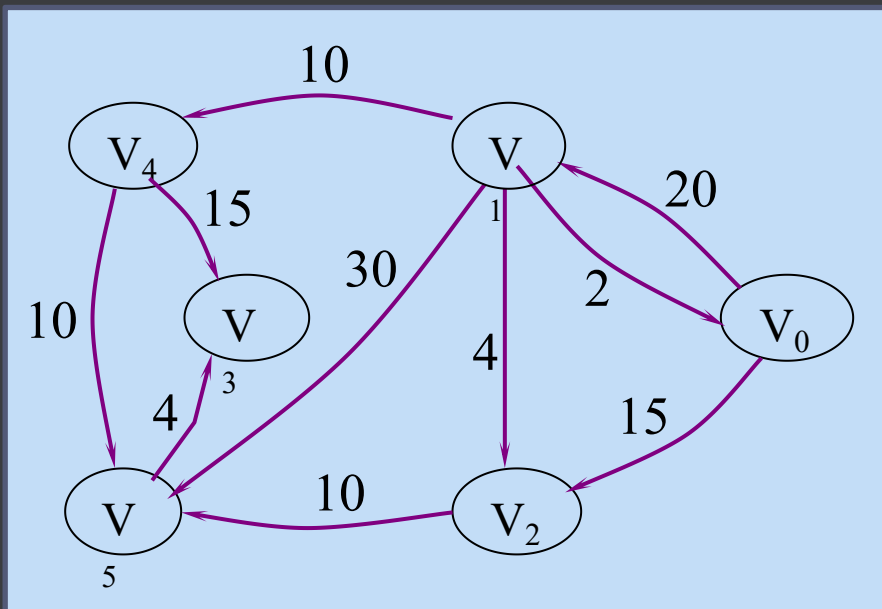
$$\text{dist}[w] = \text{dist}[u] + \langle u, w \rangle \text{ 上的权}$$

(表示从v经过u再到达w的路径长度小于原来从v到w的路径长度时，以小者代之)。修改完dist向量后，再从中选取下一条最短路径长度，……，直到从v到其它各顶点的最短路径均被求出为止 (最多为n - 1条)。

# Dijkstra算法

根据以上思路，求例5-13的 $G_{15}$ 中源点 $v_0$ 到其它各顶点最短路径的过程如下：

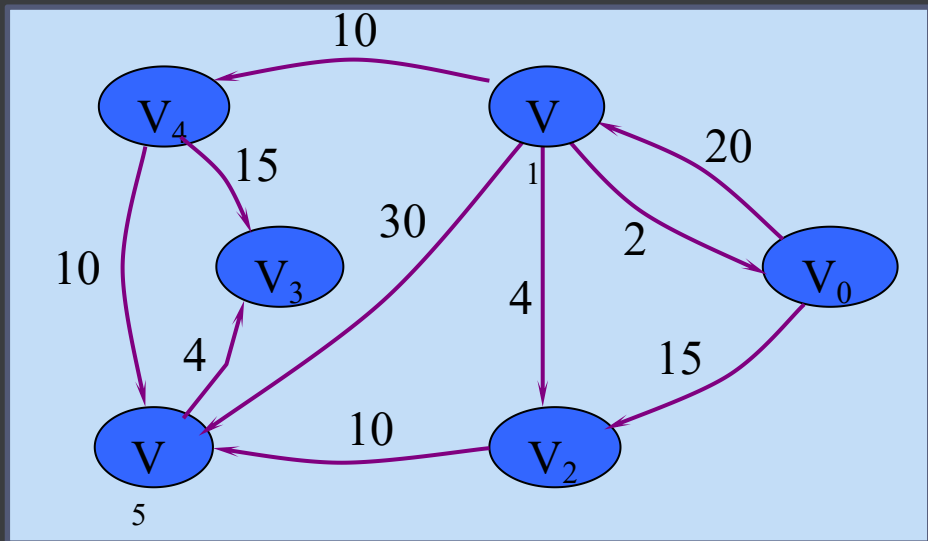
$G_{15}$ 的邻接矩阵：



$$G.A[n][n] = \begin{bmatrix} 0 & 20 & 15 & \infty & \infty & \infty \\ 2 & 0 & 4 & \infty & 10 & 30 \\ \infty & \infty & 0 & \infty & \infty & 10 \\ \infty & \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & 15 & 0 & 10 \\ \infty & \infty & \infty & 4 & \infty & 0 \end{bmatrix}$$

# Dijkstra算法

各向量的状态:



|   | S | dist | path    |
|---|---|------|---------|
| 0 | 1 | 0    | 0       |
| 1 | 1 | 20   | 0,1     |
| 2 | 1 | 15   | 0,2     |
| 3 | 1 | 29   | 0,2,5,3 |
| 4 | 1 | 30   | 0,1,4   |
| 5 | 1 | 25   | 0,2,5   |

从dist中看出, 第一条最短路径长度为 $\text{dist}[2]=15$ , 最短路径为 $(v_0, v_2)$ 。 $V_2$ 求出后, 修改各向量状态: 原来 $v_0$ 到 $v_5$ 的路径长度 $\text{dist}[5]=\infty$ ,  $v_2$ 求出后, 现 $v_0$ 经过 $v_2$ 到 $v_5$ 的路径长度为25, 所以令:

$\text{dist}[5]=\text{dist}[2]+<v_2, v_5>$ 上的权=25 (而 $v_0$ 到 $v_1$ 、 $v_3$ 、 $v_4$ 的路径长度不会改变), 并将路径  $(0,2)$  赋给 $\text{path}[5]$  (即从 $v_0$ 到 $v_5$ 可能要经过 $v_2$ )。

再求第二条最短路径长度 (找满足 $S[w]=0$ 且 $\text{dist}[w]$ 最小的), 并改变各向量状态。最后, 从 $v_0$ 到各顶点的最短路径存于向量path中, 最短路径长度存于dist中。

# Dijkstra算法

## 2. 算法描述

typedef struct

{ int pi[n];    // 存放v到 $v_i$ 的一条最短路径, n为图中顶点数 //

    int end;

} pathtype;

pathtype path[n]; // v到各顶点最短路径向量 //

int dist[n];    // v到各顶点最短路径长度向量 //

void Dijkstra(int G[n][n], pathtype path[], int dist[], int v)

// 求G(用邻接矩阵表示)中源点v到其他各顶点最短路径, n为G中顶点数 //

{   int i, count, s[n], m, u, w;

    for (i=0; i<n; i++)    // 初始化 //

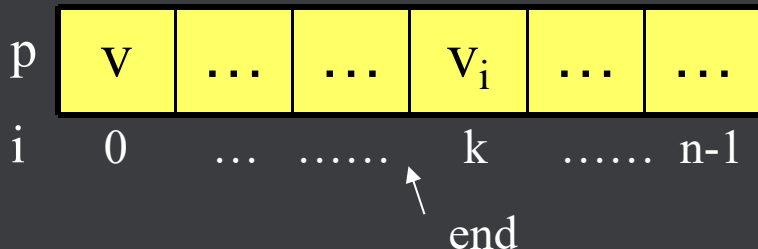
    { s[i] = 0;

        dist[i] = G[v][i];   // v到其他顶点的权为当前最短路径, 送dist[i] //

        path[i].pi[0] = v;

        path[i].end = 0;

    }



# Dijkstra算法

```
s[v] = 1; count = 1; // count为计数器 //  
while (count <= n-1) // 求n - 1条最短路径 //  
{ m = max; // max为当前机器表示的最大值 //  
  for (w=0; w<=n-1; w++) // 找当前最短路径长度 //  
  { if (s[w] == 0 && dist[w] < m)  
    { u = w; m = dist[w]; }  
  }  
  if (m == max) break; // 最短路径求完 (不足n - 1条) , 跳出while循环 //  
  s[u] = 1; // 表示v到vu最短路径求出 //  
  path[u].end++; // 置当前最短路径 //    path[u].pi[end] = u;  
  for (w=0; w<=n-1; w++) // u求出后, 修改dist和path向量 //  
  { if (s[w] == 0 && dist[w] > dist[u]+G[u][w])  
    { dist[w] = dist[u] + G[u][w];  
      path[w] = path[u]; }  
  }  
  count++; // 最短路径条数计数 //  
}
```



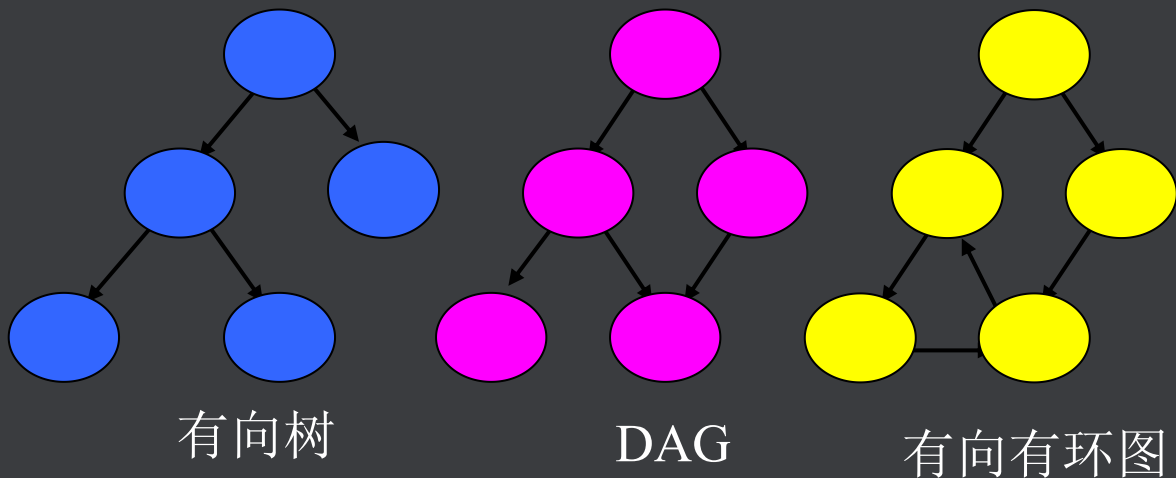
# 5.5

## 有向无环 图的应用

# 有向无环图的应用

有向无环图 (Directed Acycline Graph) 简称**DAG**。

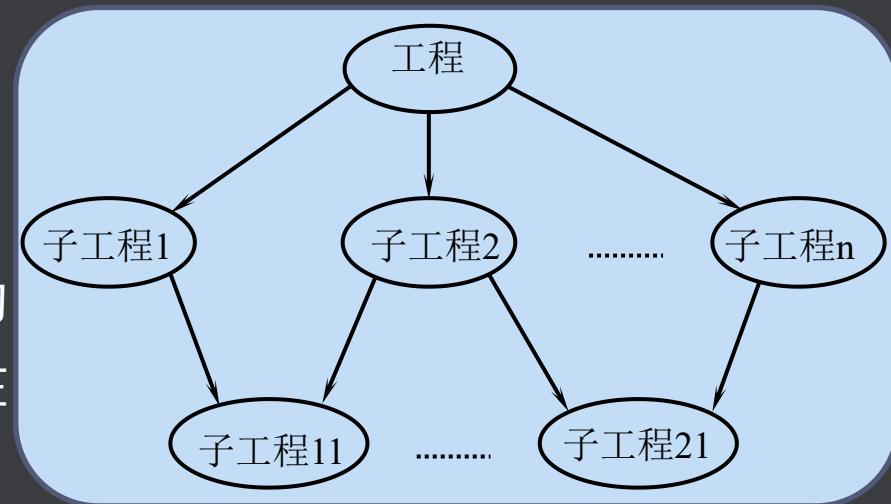
有向树、DAG和有向有环图如图5.41:



当然，有向树一定是DAG，但DAG不一定是树。这与“凡狗有四足，四足者皆为狗乎？”是一个道理。

# 有向无环图的应用

DAG是描述工程（或系统）进行过程的工具。通常一个大的工程由若干个子工程来构成，子工程又可分为若干个更小的工程，且各子工程之间有一定的约束关系，如某些子工程开始必须等到另一些子工程的结束（如房屋内装修是在房屋落成之后）。用图表示为：



对于一项工程而言，人们关心的一般有两点：

- ①工程能否顺序进行；
- ②工程完成所需的时间。

对这两个问题，实际上可以转换为如何对相应图进行拓扑排序（Topological Sort）和求图中一条关键路径（Critical Path）的问题。

如果工程能够顺序进行，即相应描述工程的图不存在环（即属于DAG）。那么如何知道图中是否存在环呢？这就要考察图的一个拓扑序列，若图中全部顶点（子工程）都在相应的拓扑序列中，则相应图不存在环（即是一个DAG），否则工程无法顺利进行。

# 拓扑排序

例5-15 计算机专业课程设置，如表5.1所示。

| 课程编号            | 课程名称     | 先行课程                             |
|-----------------|----------|----------------------------------|
| C <sub>0</sub>  | 程序设计导论   | 无                                |
| C <sub>1</sub>  | 数字电路     | 无                                |
| C <sub>2</sub>  | 计算机组成原理  | C <sub>1</sub>                   |
| C <sub>3</sub>  | 离散数学     | C <sub>0</sub>                   |
| C <sub>4</sub>  | 汇编语言     | C <sub>0</sub> , C <sub>2</sub>  |
| C <sub>5</sub>  | 算法语言程序设计 | C <sub>0</sub>                   |
| C <sub>6</sub>  | 数据结构     | C <sub>3</sub> , C <sub>5</sub>  |
| C <sub>7</sub>  | 软件工程     | C <sub>3</sub> , C <sub>6</sub>  |
| C <sub>8</sub>  | 编译原理     | C <sub>5</sub> , C <sub>6</sub>  |
| C <sub>9</sub>  | 操作系统     | C <sub>2</sub> , C <sub>6</sub>  |
| C <sub>10</sub> | 数据库系统原理  | C <sub>3</sub> , C <sub>6</sub>  |
| C <sub>11</sub> | 管理信息系统   | C <sub>7</sub> , C <sub>10</sub> |
| C <sub>12</sub> | 人工智能原理   | C <sub>6</sub> , C <sub>10</sub> |
| C <sub>13</sub> | 专家系统     | C <sub>12</sub>                  |

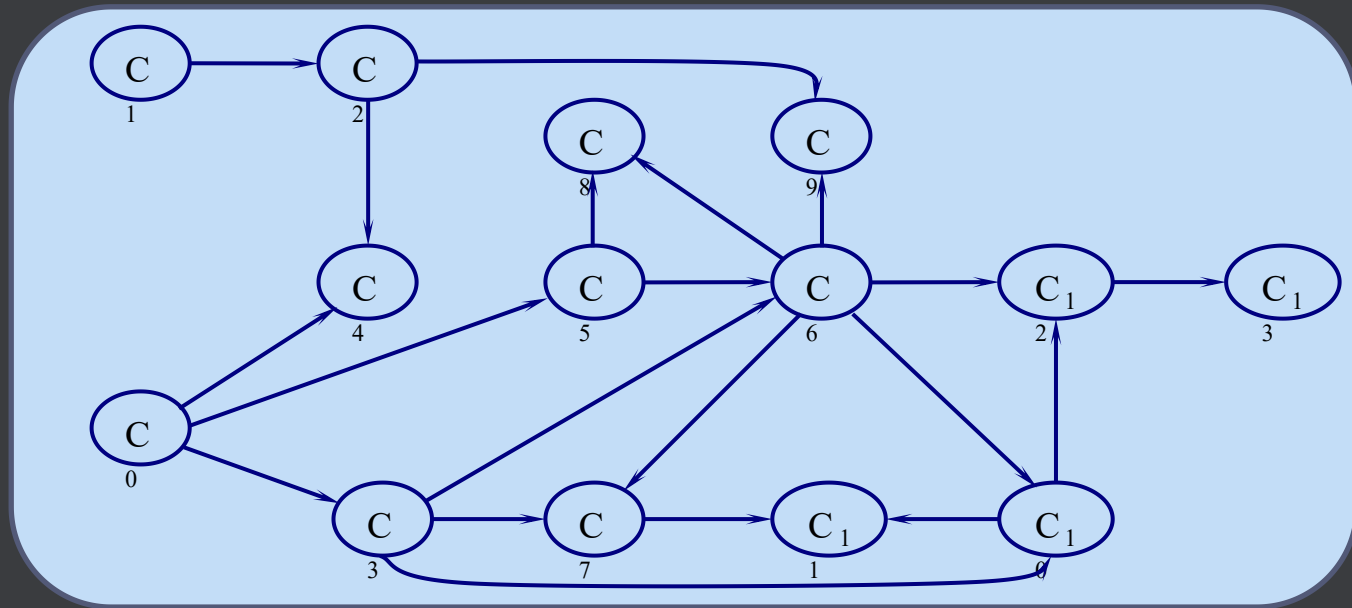
若某门课程C<sub>i</sub>是C<sub>j</sub>的先行课程（或先开课程），则关系<C<sub>i</sub>, C<sub>j</sub>>存在，表示为：



称C<sub>i</sub>为C<sub>j</sub>的直接前驱; C<sub>j</sub>是C<sub>i</sub>的直接后继。

# 拓扑排序

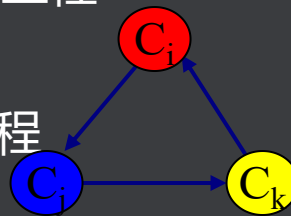
由表5.1中各课程之间的优先关系,构造图 $G_{17}$ 如下:



这种由顶点表示活动, 用弧表示活动间优先关系的有向图又称为**AOV**

(Activity On Vertex Network) 网。若AOV网中出现环, 则对应的工程无法进行 (互相等待)。

如上图中的网, 若出现如右侧的环, 会有几门课程互为先行课程, 课程是很难开设下去的。



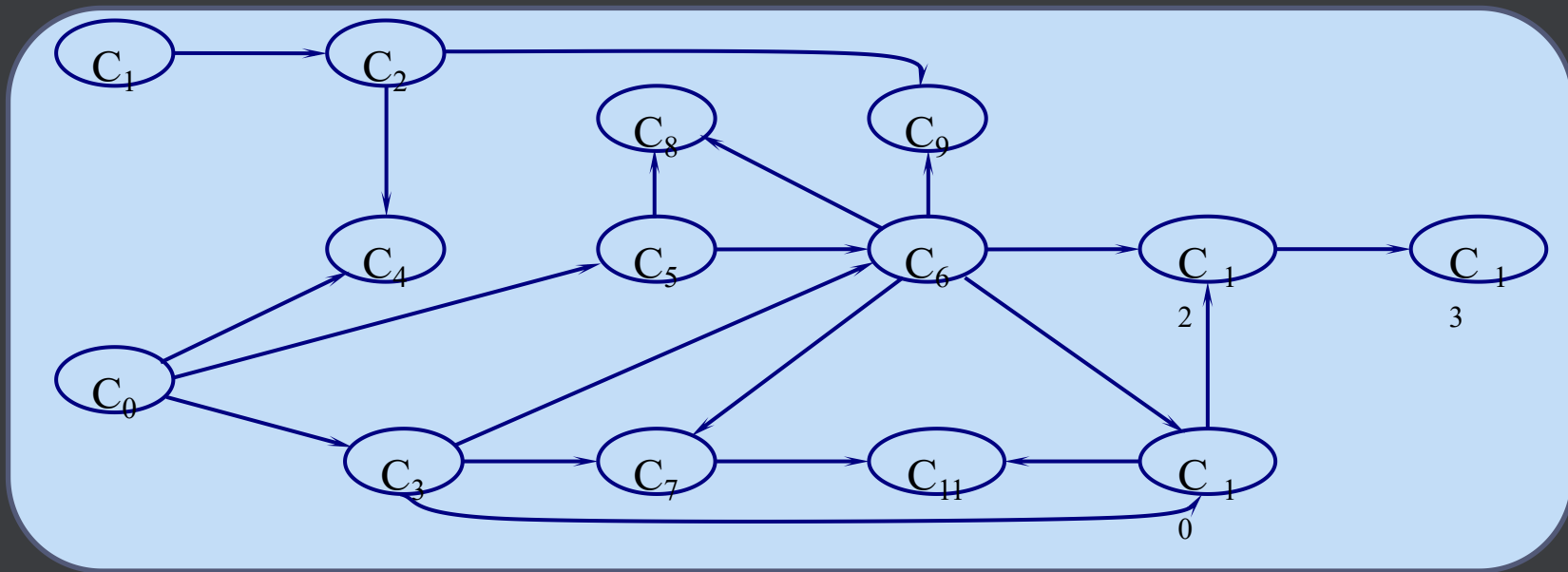
因此, 要确定已绘制出的**AOV**网是否是一个**DAG**, 即对**AOV**网考察其拓扑排序序列。

# 拓扑排序

- (1) 在AOV网中任选一个无前驱的顶点输出;
- (2) 删去输出过的顶点和由它发出的各条弧;
- (3) 重复 (1)、(2) 直到所有可输出顶点全部输出完为止。

若AOV网中全部顶点都已输出，则AOV网是一个DAG，得到的顶点序列为一个拓扑序列，否则该图一定存在环（因为剩下的顶点都有前驱）。

$G_{17}$ 的一个拓扑序列为:  $(C_1 C_2 C_0 C_5 C_4 C_3 C_6 C_{10} C_{12} C_{13} C_9 C_8 C_7 C_{11})$ 。



输出的顶点数与相应AOV网中顶点数相等，则此图不存在环，是一个DAG，亦即课程的设置是合理的。

# 拓扑排序

## 拓扑序列的特点

对图中任意两顶点 $v_i$ 和 $v_j$ ，若图无环且 $v_i$ 是 $v_j$ 的前驱，则 $v_i$ 在序列中一定位于 $v_j$ 之前。由于在某一时刻可能有多个无前驱的顶点，这时可选择其中一个输出，故拓扑序列不唯一。

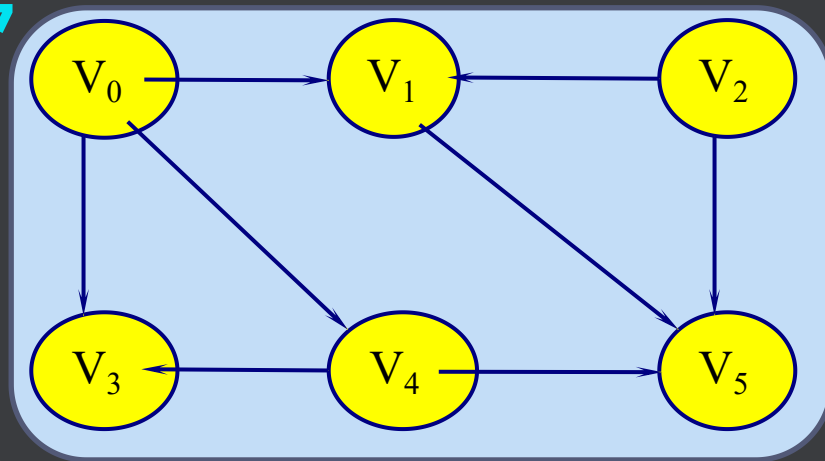
## 拓扑排序算法

设AOV网用十字链表表示，以方便求每顶点的入度。算法步骤为：

- ①查十字链表中入度（ID）为0的顶点（即无前驱的顶点）并进栈；
- ②重复以下过程，直到栈为空：输出栈顶 $v_j$ ，并退栈；查顶点 $v_j$ 的直接后继 $v_k$ （可能多个），将 $v_k$ 的入度减1（表示抹去 $v_j$ 到 $v_k$ 的弧），并将新的入度为0的顶点进栈；
- ③若输出顶点数为AOV网中的顶点数 $n$ ，则网中不存在环，输出序列为拓扑序列，否则网中存在环。

# 拓扑排序

例5-16 设AOV网 $G_{18}$ 如下:



$G_{18}$  的十字链表如图5.46所示。

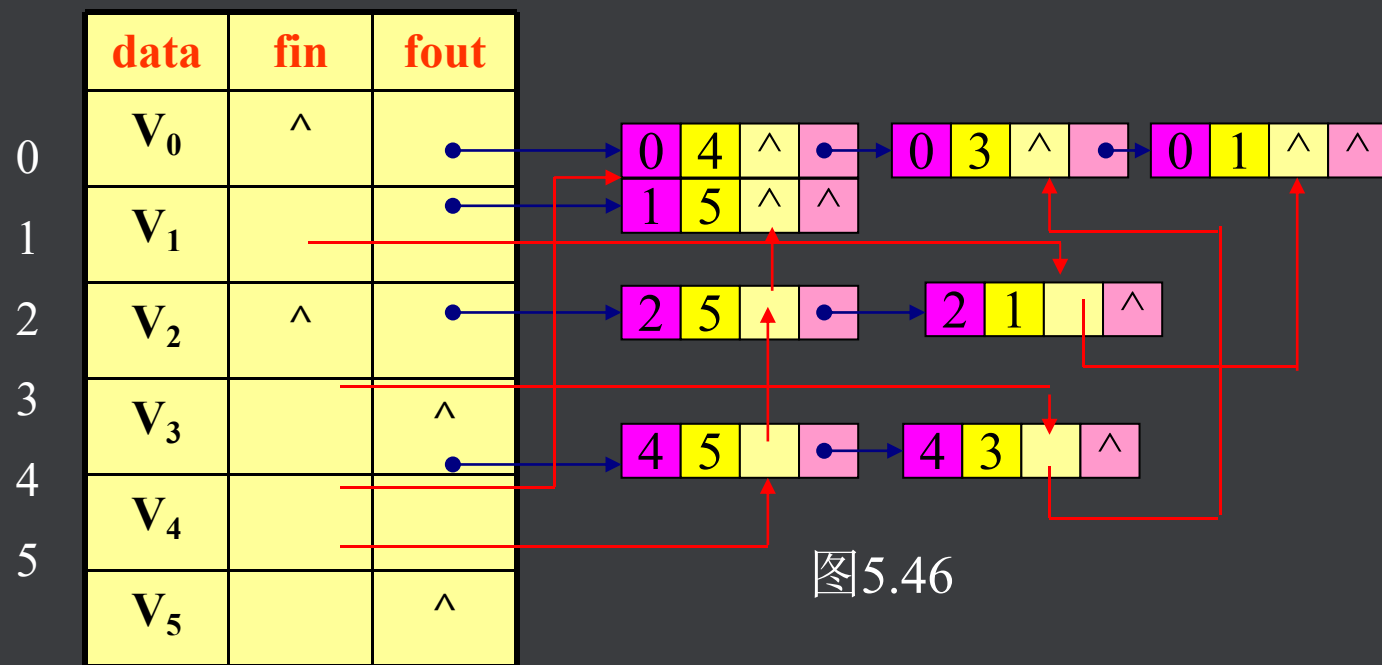


图5.46



# 拓扑排序

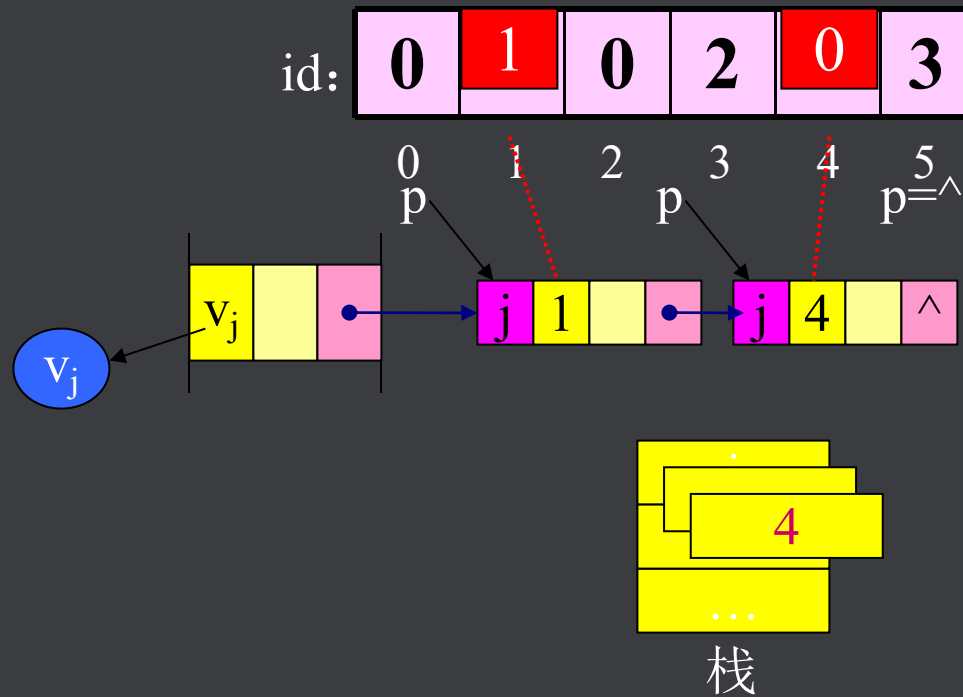
## 2.算法描述

```
void Creatid (Vexnode G[ ], int n,id[ ]) //建立顶点的入度表id, 顶点数=n //
{ int count,i; Arcnode *p;
  for (i=0;i<n;i++) //求n个顶点的入度//
  { id[i]= 0; count=0; //入度值计数//
    p=G[i].fin; //取以 $v_i$ 为弧头的第一弧结点//
    while (p) { count++; p=p->hlink; } //取以 $v_i$ 为弧头的下一弧结点//
    id[i]=count; } //入度赋值//
void Topsort (Vexnode G[ ], int n) //对网G拓扑排序的算法//
{ int i,j,k,count,id[ ]; Arcnode *p;
  Creatid (G,n,id); //建立G的入度表id//
  Clearstack (s); //置栈空//
  for (i=0;i<n;i++) if (id[i]==0) Push (s,i); //入度为0的顶点序号进栈//
  count=0; //输出顶点计数//
```

# 拓扑排序

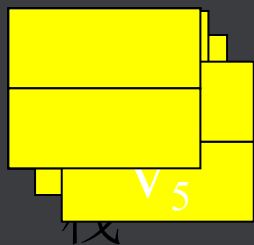
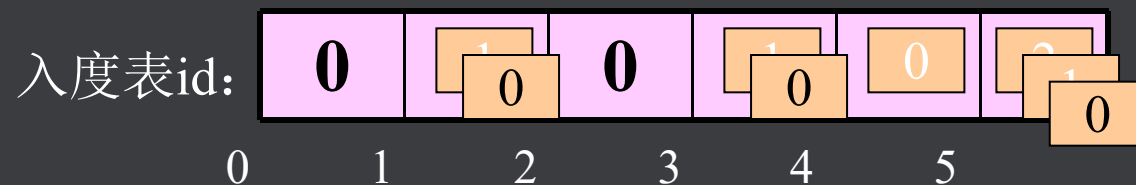
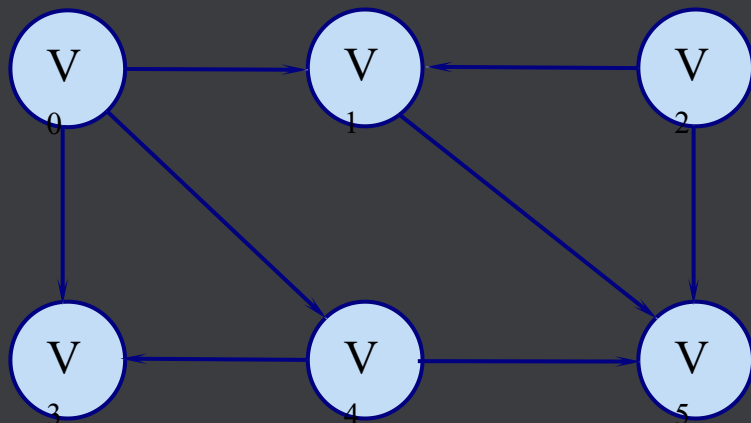
```
while (!Emptystack (s)) // 栈非空时
{
    j=Pop (s); // 退栈, 栈顶赋给j
    output (j,G[j].data); // 输出vj
    count++; p=G[j].fout; // 取vj发出的第一条弧
    while (p)
    {
        k=p->head; // 取vj之后继vk
        id[k]--; // vk的入度减1
        if (id[k]==0)
            Push (s,k); // 入度为0的顶点序号进栈
        p=p->tlink; // 取vj的下一后继
    }
}

if (count==n)
    printf ( "This graph has not cycle." )
else
    printf ( "This graph has cycle." );
}
```



# 拓扑排序

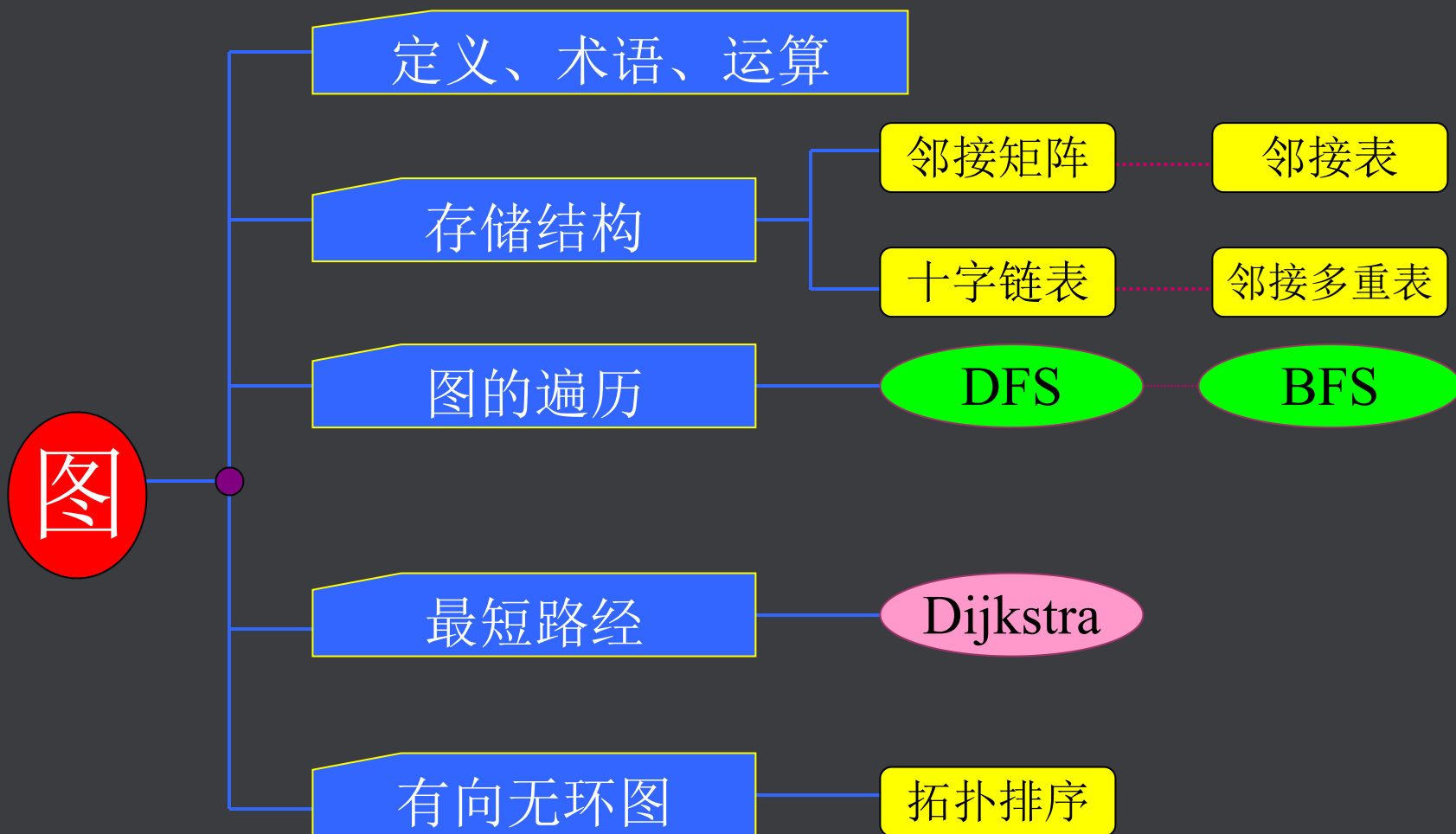
例5-16:



输出顶点:  $V_2$   $V_0$   $V_1$   $V_4$   $V_3$   $V_5$

count==6 This graph has not cycle.

# 小结



6

查找

# 查找概念

查找(或检索)是在给定信息集上寻找特定信息元素的过程。

待查找的数据单位(或数据元素)称为记录。记录由若干数据项(或属性)组成，如学生记录：

|    |    |    |    |       |
|----|----|----|----|-------|
| 学号 | 姓名 | 性别 | 年龄 | ..... |
|----|----|----|----|-------|

其中，“学号”、“姓名”、“性别”、“年龄”等都是记录的数据项。

若某个数据项的值能标识(或识别)一个或一组记录，称其为关键字(key)。

若一个key能唯一标识一个记录，称此key为主key。如“学号”的值给定就唯一对应一个学生，不可能多个学生的学号相同，故“学号”在学生记录里可作为主key。

若一个key能标识一组记录，称此key为次key。

如“年龄”值为20时，可能有若干同学的年龄为20岁，故“年龄”可作次key。下面主要讨论对主key的查找。

# 查找概念

## 查找定义

设记录表 $L=(R_1 R_2 \dots R_n)$ ，其中 $R_i(1 \leq i \leq n)$ 为记录，对给定的某个值 $k$ ，在表 $L$ 中确定 $\text{key}=k$ 的记录的过程，称为查找。若表 $L$ 中存在一个记录 $R_i$ 的 $\text{key}=k$ ，记为 $R_i.\text{key}=k$ ，则查找成功，返回该记录在表 $L$ 中的序号 $i$ (或 $R_i$ 的地址)，否则(查找失败)返回0(或空地址Null)。

## 查找方法

查找方法有顺序查找、折半查找、分块查找、Hash表查找等等。查找算法的优劣将影响到计算机的使用效率，应根据应用场合选择相应的查找算法。

# 查找概念

## 平均查找长度

评价一个算法的好坏，一是时间复杂度 **$T(n)$** ， $n$ 为问题的体积，此时为表长；二是空间复杂度 **$D(n)$** ；三是算法的结构等其他特性。

对查找算法，主要分析其 **$T(n)$** 。查找过程是**key**的比较过程，时间主要耗费在各记录的**key**与给定**k**值的比较上。比较次数越多，算法效率越差（即 **$T(n)$** 量级越高），故用“比较次数”刻画算法的 **$T(n)$** 。另外，不能以查找某个记录的时间来作为 **$T(n)$** ，一般以“平均查找长度”来衡量 **$T(n)$** 。

平均查找长度**ASL** (Average Search Length)：对给定**k**，查找表**L**中记录比较次数的期望值(或平均值)，即：

$$ASL = \sum_{i=1}^n P_i C_i$$

$P_i$ 为查找 **$R_i$** 的概率。等概率情况下 **$P_i = 1/n$** ； **$C_i$** 为查找 **$R_i$** 时key的比较次数(或查找次数)。



# 顺序表的查找

所谓顺序表(Sequential Table)，是将表中记录( $R_1 R_2 \dots R_n$ )按其序号存储于一维数组空间，如图所示。其特点是相邻记录的物理位置也是相邻的。

记录 $R_i$ 的类型描述如下：

```
typedef struct
{   keytype key; //记录key//
    .....      //记录其他项//
} Rtype;
```

其中，类型keytype是泛指，即keytype可以是int、float、char或其他的结构类型等等。为讨论问题方便，一般取key为整型。

顺序表类型描述如下：

```
#define maxn 1024    //表最大长度//
typedef struct {   Rtype data[maxn]; //顺序表空间//
                  int len; //当前表长，表空时len=0//
                  } slist;
```

若说明：slist r，则 ( $r.data[1], \dots, r.data[r.len]$ ) 为记录表( $R_1 \dots R_n$ )， $R_i.key$ 为 $r.data[i].key$ ， $r.data[0]$ 称为监视哨，为算法设计方便所设。

# 顺序查找(Sequential Search)算法及分析

**算法思路** 设给定值为 $k$ ，在表 $(R_1 R_2 \dots R_n)$ 中，从 $R_n$ 开始，查找 $\text{key}=k$ 的记录。若存在一个记录 $R_i$  ( $1 \leq i \leq n$ ) 的 $\text{key}$ 为 $k$ ，则查找成功，返回记录序号 $i$ ；否则，查找失败，返回0。

## 算法描述

```
int sqsearch(sqlist r, keytype k) //对表r顺序查找的算法//
{
    int i;
    r.data[0].key = k; //k存入监视哨//
    i = r.len; //取表长//
    while(r.data[i].key != k) i--; //顺序往前查找//
    return (i);
}
```

算法用了一点技巧：先将 $k$ 存入监视哨，若对某个 $i$  ( $\neq 0$ ) 有 $r.data[i].key=k$ ，则查找成功，返回 $i$ ；若 $i$ 从 $n$ 递减到1都无记录的 $\text{key}$ 为 $k$ ， $i$ 再减1为0时，必有 $r.data[0].key=k$ ，说明查找失败，返回 $i=0$ 。

# 顺序查找(Sequential Search)算法及分析

## 算法分析

设 $C_i (1 \leq i \leq n)$ 为查找第 $i$ 记录的key比较次数(或查找次数):

若 $r.data[n].key = k$ ,  $C_n = 1$ ;

若 $r.data[n-1].key = k$ ,  $C_{n-1} = 2$ ;

.....

若 $r.data[i].key = k$ ,  $C_i = n - i + 1$ ;

.....

若 $r.data[1].key = k$ ,  $C_1 = n$

故 $ASL = O(n)$ 。而查找失败时, 查找次数等于 $n+1$ , 同样为 $O(n)$ 。

对查找算法, 若 $ASL = O(n)$ , 则效率是很低的, 意味着查找某记录几乎要扫描整个表, 当表长 $n$ 很大时, 会令人无法忍受。下面关于查找的一些讨论, 大多都是围绕降低算法的ASL量级而展开的。

# 折半查找算法及分析

当记录的key按关系 $\leq$ 或 $\geq$ 有序时，即：

$R_1.key \leq R_2.key \leq \dots \leq R_n.key$  (升序)

或  $R_1.key \geq R_2.key \geq \dots \geq R_n.key$  (降序)

## 算法思路

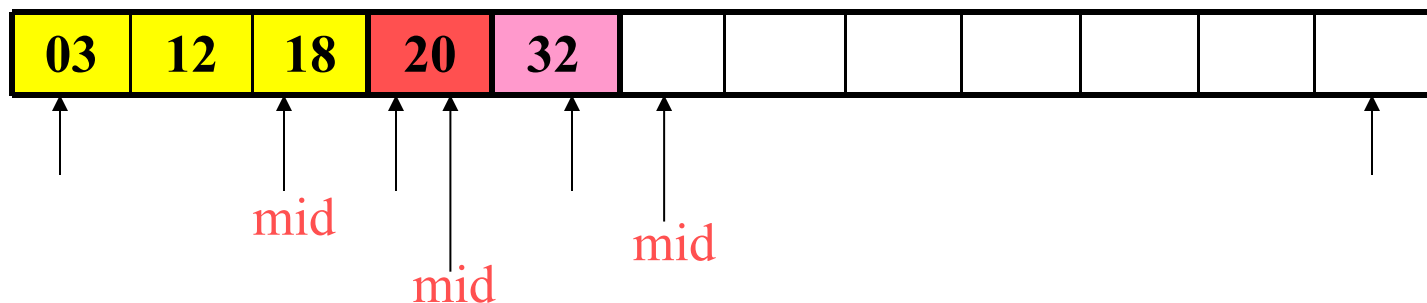
对给定值k，逐步确定待查记录所在区间，每次将搜索空间减少一半(折半)，直到查找成功或失败为止。

设两个指针(或游标)low、high，分别指向当前待查找表的上界(表头)和下界(表尾)。对于表( $R_1 R_2 \dots R_n$ )，初始时low=l、high=n，令：

mid=

指向当前待查找表中间的那个记录。下面举例说明折半查找的过程。

# 折半查找算法及分析



$$\lfloor (1+12)/2 \rfloor$$

&

$$\lfloor (1+5)/2 \rfloor$$

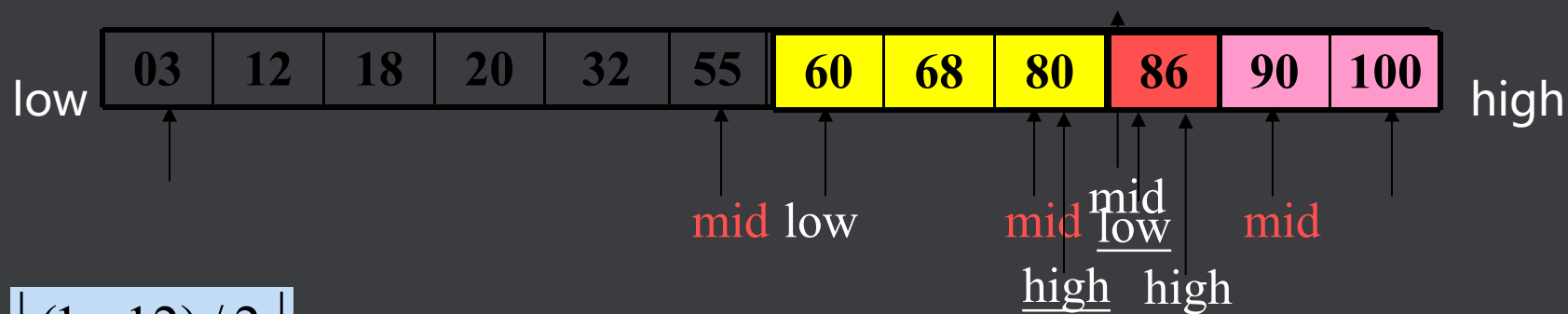
半区间。令：low=mid+1。

&●mid= $\lfloor (4+5)/2 \rfloor=4$ 。因k=r.data[4].key=20，查找成功，返回mid=4。

# 折半查找算法及分析

再看查找失败的情况，设要查找 $k=85$ 的记录。

序号： 1    2    3    4    5    6    7    8    9    10    11    12 (n=12)



$mid = \lfloor (1 + 12) / 2 \rfloor = 6$ 。因 $k > r.data[6].key = 55$ ，若85存在，一定落在“55”的右半区间。令： $low = mid + 1$ 。

$mid = \lfloor (7 + 12) / 2 \rfloor = 9$ 。因 $k > r.data[9].key = 80$ ，若85存在，一定落在“80”的右半区间。令： $low = mid + 1$ 。

$mid = \lfloor (10 + 12) / 2 \rfloor = 11$ 。因 $k < r.data[11].key = 90$ ，若85存在，一定落在“90”的左半区间。令： $high = mid - 1$ 。

$mid = \lfloor (10 + 10) / 2 \rfloor = 10$ 。因 $k < r.data[10].key = 86$ ，若85存在，一定落在“86”的左半区间。令： $high = mid - 1$ 。此时，下界 $low = 10$ ，而上界 $high = 9$ ，表明搜索空间不存在，故查找失败，返回0。

# 折半查找算法及分析

## 算法描述

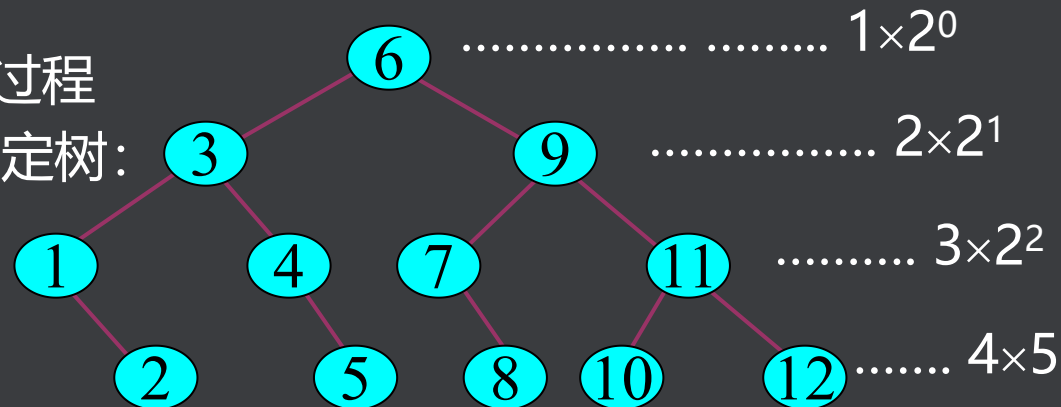
```
int Binsearch(sqlist r, keytype k)  //对有序表r折半查找的算法//
{ int low, high, mid; low = 1;high = r.len; //上下界初值//
  while (low <= high)  //表空间存在时//
  { mid = (low+high) / 2; //求当前mid//
    if (k == r.data[mid].key) return (mid); //查找成功, 返回mid//
    if (k < r.data[mid].key) high = mid-1; //调整上界, 向左部查找//
    else low = mid+1; } //调整下界, 向右部查找//
  return(0); } //low>high, 查找失败//
```

查找次数

## 算法分析

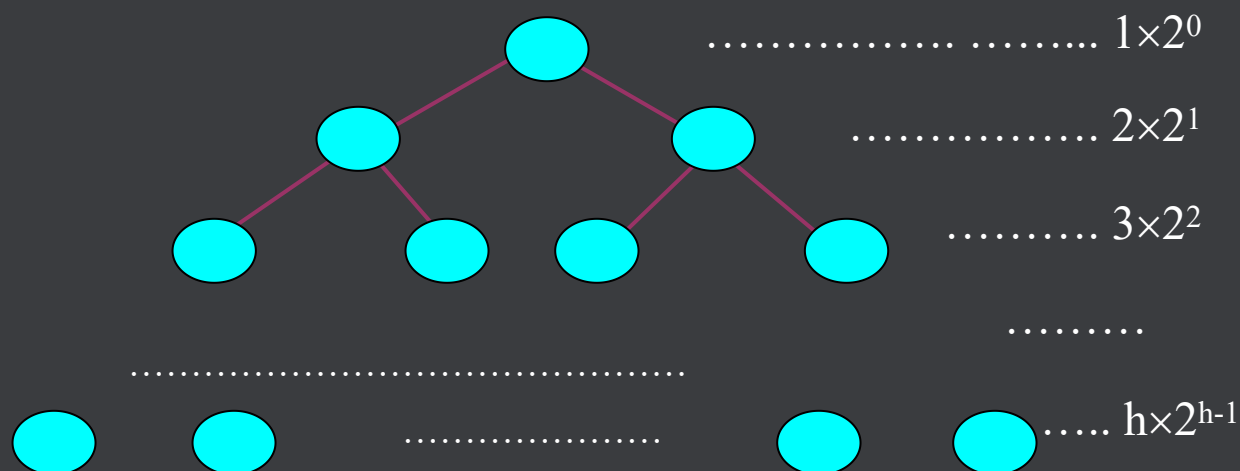
对例1中记录表的查找过程

可得到如图所示的一棵判定树:



# 折半查找算法及分析

不失一般性，设表长 $n=2^h-1$ ， $h=\log_2(n+1)$ 。记录数 $n$ 恰为一棵 $h$ 层的满二叉树的结点数。对照例1，得出表的判定树及各记录的查找次数如图所示。



$$ASL = \sum_{i=1}^n P_i C_i = \frac{1}{n} \sum_{i=1}^h i \cdot 2^{i-1} \quad \text{令 } S = \sum_{i=1}^h i \cdot 2^{i-1} = 1 \cdot 2^0 + 2 \cdot 2^1 + 3 \cdot 2^2 + \dots + (h-1) 2^{h-2} + h \cdot 2^{h-1}$$

$$2S = 1 \cdot 2^1 + 2 \cdot 2^2 + 3 \cdot 2^3 + \dots + (h-1) 2^{h-1} + h \cdot 2^h$$

$$S = 2S - S = h \cdot 2^h - (2^0 + 2^1 + 2^2 + \dots + 2^{h-1}) = h \cdot 2^h - (2^h - 1) = (n+1) \log_2(n+1) - n$$

$$\text{故 } ASL = \frac{1}{n} ((n+1) \log_2(n+1) - n) = \frac{n+1}{n} \log_2(n+1) - 1$$

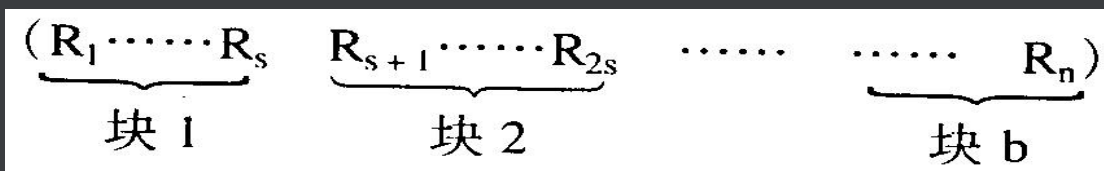
$n \rightarrow \infty$  时,  $ASL = O(\log_2(n+1))$ , 大大优于  $O(n)$ 。



# 分块查找(索引顺序查找) 算法及分析

## 分块

设记录表长为 $n$ ，将表的 $n$ 个记录分成 $b = \lceil n/s \rceil$  个块，每块 $s$ 个记录（最后一块记录数可以少于 $s$ 个），即：



且表分块有序，即第 $i$  ( $1 \leq i \leq b-1$ ) 块所有记录的key小于第 $i+1$ 块中记录的key，但块内记录可以无序。

## 建立索引

每块对应一索引项：

|            |      |
|------------|------|
| $k_{\max}$ | link |
|------------|------|

其中 $k_{\max}$ 为该块内记录的最大key；link为该块第一记录的序号（或指针）。

# 分块查找(索引顺序查找)

**例2** 设表长 $n=19$ , 取 $s=5$ ,  $b=\lceil 19/5 \rceil = 4$ , 分块索引结构, 如图所示。

算法思路 分块索引查找分两步进行:

(1)由索引表确定待查找记录所在的块;

(2)在块内顺序查找。

如查找 $k=19$ 的记录, 因 $19 > 18$ , 不会落在第1块; 又 $19 < 50$ , 若19存在, 必在第2块内。取第2块起址(6), 查找到key为19的记录号为9。

查找失败情况, 一是给定 $k$ 值超出索引表范围; 二是若 $k$ 落在某块内, 但该块中无 $key=k$ 的记录。

索引表是按照 $k_{\max}$ 有序的, 可对其折半查找。而块内按顺序方法查找。

| $K_{\max}$ | 序号 |
|------------|----|
| 18         | 1  |
| 50         | 6  |
| 84         | 11 |
| 108        | 16 |

(索引表)

序号 R.key

|    |     |
|----|-----|
| 1  | 18  |
| 2  | 10  |
| 3  | 9   |
| 4  | 8   |
| 5  | 16  |
| 6  | 20  |
| 7  | 38  |
| 8  | 42  |
| 9  | 19  |
| 10 | 50  |
| 11 | 84  |
| 12 | 72  |
| 13 | 56  |
| 14 | 55  |
| 15 | 76  |
| 16 | 100 |
| 17 | 90  |
| 18 | 88  |
| 19 | 108 |

# Hash表的查找

## Hash表的含义

Hash表，又称散列表。在前面讨论的顺序、折半、分块查找和树表的查找中，其ASL的量级在 $O(n) \sim O(\log_2 n)$ 之间。不论ASL在哪个量级，都与记录长度 $n$ 有关。随着 $n$ 的扩大，算法的效率会越来越低。ASL与 $n$ 有关是因为记录在存储器中的存放是随机的，或者说记录的key与记录的存放地址无关，因而查找只能建立在key的“比较”基础上。

理想的查找方法是：对给定的 $k$ ，不经任何比较便能获取所需的记录，其查找的时间复杂度为常数级 $O(C)$ 。这就要求在建立记录表的时候，确定记录的key与其存储地址之间的关系 $f$ ，即使key与记录的存放地址 $H$ 相对应：

$$\text{key} \xrightarrow[f]{f} H: \text{记 录}$$

或者说，记录按key存放。

# Hash表的查找

之后，当要查找 $\text{key} = k$ 的记录时，通过关系 $f$ 就可得到相应记录的地址而获取记录，从而免去了 $\text{key}$ 的比较过程。这个关系 $f$ 就是所谓的Hash函数（或称散列函数、杂凑函数），记为 $H(\text{key})$ 。它实际上是一个地址映象函数，其自变量为记录的 $\text{key}$ ，函数值为记录的存储地址（或称Hash地址）。

另外，不同的 $\text{key}$ 可能得到同一个Hash地址，即当 $\text{key}_1 \neq \text{key}_2$ 时，可能有 $H(\text{key}_1) = H(\text{key}_2)$ ，此时称 $\text{key}_1$ 和 $\text{key}_2$ 为**同义词**。这种现象称为“**冲突**”或“**碰撞**”，因为一个数据单位只可存放一条记录。

一般，选取Hash函数只能做到使冲突尽可能少，却不能完全避免。这就要求在出现冲突之后，寻求适当的方法来解决冲突记录的存放问题。

# Hash表的查找

例3 设记录的key集合为C语言的一些保留字，即 $k=\{\text{case、char、float、for、int、while、struct、typedef、union、goto、viod、return、switch、if、break、continue、else}\}$ ，构造关于k的Hash表时，可按不同的方法选取 $H(\text{key})$ 。

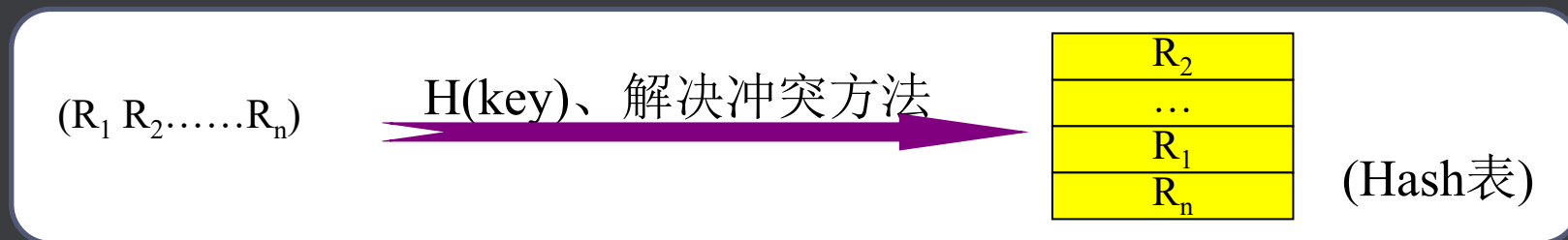
(1) 令 $H_1(\text{key})=\text{key}[0]-\text{'a'}$ ，其中 $\text{key}[0]$ 为key的第一个字符。显然这样选取的Hash函数冲突现象频繁。如： $H_1(\text{float})=H_1(\text{for})=\text{'f'}-\text{'a'}=5$ 。  
解决冲突的方法之一是为“for”寻求另一个有效位置。

(2) 令 $H_2(\text{key})=(\text{key}[0]+\text{key}[\text{i}-1]-2*\text{'a'})/2$ 。其中 $\text{key}[\text{i}-1]$ 为key的最后一个字符。如： $H_2(\text{float})=(\text{'f'}+\text{'t'}-2*\text{'a'})/2=12$ ， $H_2(\text{for})=(\text{'f'}+\text{'r'}-2*\text{'a'})/2=11$ ，从而消除了一些冲突的发生，但仍无法完全避免，如： $H_2(\text{case})=H_2(\text{continue})$ 。

# Hash表的查找

综上所述，对Hash表的含义描述如下：

根据选取的Hash函数 $H(\text{key})$ 和处理冲突的方法，将一组记录( $R_1, R_2, \dots, R_n$ )映象到记录的存储空间，所得到的记录表称为Hash表，如图：



关于Hash表的讨论关键是两个问题，一是选取Hash函数的方法；二是确定解决冲突的方法。选取（或构造）Hash函数的方法很多，原则是尽可能将记录均匀分布，以减少冲突现象的发生。以下介绍几种常用的构造方法。

直接地址法

平方取中法

叠加法

保留除数法

随机函数法

# 直接地址法

此方法是取key的某个线性函数为Hash函数，即令：

$$H(\text{key}) = a \cdot \text{key} + b$$

其中a、b为常数，此时称H(key)为直接Hash函数或自身函数。

**例4** 设某地区1 ~ 100岁的人口统计表如表：

| 记录 | $R_1$ | $R_2$ | ..... | $R_{25}$ | ..... | $R_{100}$ |
|----|-------|-------|-------|----------|-------|-----------|
| 岁数 | 1     | 2     | ..... | 25       | ..... | 100       |
| 人数 | 3000  | 2500  | ..... | 20000    | ..... | 10        |

给定的存储空间为 **$b+1 \sim b+100$** 号单元（b为起始地址），每个单元可以存放一条记录 **$R_i (1 \leq i \leq 100)$** 。取“岁数”为key，令：

$$H(\text{key}) = \text{key} + b$$

则按此函数构造的Hash表如下图所示：

# 保留除数法

又称质数除余法，设Hash表空间长度为m，选取一个不大于m的最大质数

p, 令:  $H(\text{key}) = \text{key} \% p$

例如: m=8 16 32 64 128 256 512 1024 .....

p=7 13 31 61 127 251 503 1019 .....

为何选取p为不大于m的最大质数呢？举例说明。

**例8** 设记录的key集合 $k=\{28, 35, 63, 77, 105, \dots\}$ ，若选取 $p=21=3*7$ （包括质数因子7），有：

key: 28 35 63 77 105 .....

$H(\text{key}) = \text{key} \% 21$ : 7 14 0 14 0 .....

使得包含质数因子7的key都可能被映象到相同的单元，冲突现象严重。

若取 $p=19$ （质数），同样对上面给定的key集合k，有：

key: 28 35 63 77 105

$H(\text{key}) = \text{key} \% 19$ : 9 16 6 1 10

$H(\text{key})$ 的随机度就好多了。



# 处理冲突的方法

选取随机度好的Hash函数可使冲突减少，一般来讲不能完全避免冲突。因此，如何处理冲突是Hash表不可缺少的另一方面。

设Hash表地址空间为 $0 \sim m-1$ （表长为 $m$ ）：



H(key):    0        1                    j-1    j        j+1                    m-1

冲突是指：表中某地址 $j \in [0, m-1]$ 中已存放有记录，而另一个记录的H(key)值也为j。

处理冲突的方法一般为：在地址j的前面或后面找一个空闲单元存放冲突的记录，或将相冲突的诸记录拉成链表等等。

在处理冲突的过程中，可能发生一连串的冲突现象，即可能得到一个地址序列 $H_1, H_2, \dots, H_n$ ， $H_i \in [0, m-1]$ 。 $H_1$ 是冲突时选取的下一地址，而 $H_1$ 中可能已有记录，又设法得到下一地址 $H_2, \dots$ 直到某个 $H_n$ 不发生冲突为止。这种现象称为“聚积”，它严重影响了Hash表的查找效率。

冲突现象的发生有时并不完全是由于Hash函数的随机性不好引起的，聚积的发生也会加重冲突。还有一个因素是表的装填因子 $\alpha$ ， $\alpha = n/m$ ，其中 $m$ 为表长， $n$ 为表中记录个数。一般 $\alpha$ 在 $0.7 \sim 0.8$ 之间，使表保持一定的空闲余量，以减少冲突和聚积现象。

# 开放地址法

当发生冲突时，在 $H(\text{key})$ 的前后找一个空闲单元来存放冲突的记录，  
即在 $H(\text{key})$ 的基础上获取下一地址：

$$H_i = (H(\text{key}) + d_i) \% m$$

其中 $m$ 为表长， $\%$ 运算是保证 $H_i$ 落在 $[0, m-1]$ 区间； $d_i$ 为地址增量。

$d_i$ 的取法有多种：

- (1)  $d_i = 1, 2, 3, \dots, (m-1)$ ——称为线性探查法；
- (2)  $d_i = 1^2, -1^2, 2^2, -2^2, \dots$ ——称为二次探查法。

式 (1)、(2)表示：第1次发生冲突时，地址增量 $d_1$ 取1或 $1^2$ ；再冲突时， $d_2$ 取2或 $-1^2$ ，……，依此类推。

# 开放地址法

例9 设记录的key集合 $k=\{23, 34, 14, 38, 46, 16, 68, 15, 07, 31, 26\}$ , 记录数 $n=11$ 。令装填因子 $\alpha=0.75$ , 取表长 $m=\lceil n/\alpha \rceil=15$ 。用“保留余数法”选取Hash函数 ( $p=13$ ) :

$$H(\text{key})=\text{key}\%13$$

采用“线性探查法”解决冲突。依据以上条件, 依次取 $k$ 中各值构造的Hash表HT, 如下图所示(表HT初始为空)。

$k=\{23, 34, 14, 38, 46, 16, 68, 15, 07, 31, 26\}$   
 $H(\text{key})=\text{key}\%13; H_i=(H(\text{key})+d_i)\%15; d_i=1, 2, 3, \dots(m-1)$

HT:
 

|    |    |    |    |    |    |   |    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|---|----|----|----|----|----|----|----|----|
| 26 | 14 | 15 | 16 | 68 | 31 | ^ | 46 | 34 | 07 | 23 | ^  | 38 | ^  | ^  |
| 0  | 1  | 2  | 3  | 4  | 5  | 6 | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 |

$H(\text{key})$

$H(68)=68\%13=3$ (冲突), 取 $H_1=(3+1)\%15=4$ (空), 故68存入4单元。

$H(07)=7\%13=7$ (冲突), 取 $H_1=(7+1)\%15=8$ (冲突), 取 $H_2=(7+2)\%15=9$ (空), 故07存入9单元。

若采用二次探测法:  $d_i=1^2, -1^2, 2^2, -2^2, \dots$ , 表为:

|        |    |    |    |    |    |    |    |    |    |   |    |    |    |    |    |
|--------|----|----|----|----|----|----|----|----|----|---|----|----|----|----|----|
| HT:    | 26 | 14 | 15 | 16 | 68 | 31 | 07 | 46 | 34 | ^ | 23 | ^  | 38 | ^  | ^  |
| H(key) | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9 | 10 | 11 | 12 | 13 | 14 |

其中,  $H(07)=7\%13=7$ (冲突), 取 $H_1=(7+1^2)\%15=8$ (冲突), 取 $H_2=(7-1^2)\%15=6$ (空), 故07存入6单元。

# 链地址法

发生冲突时，将各冲突记录链在一起，即同义词的记录存于同一链表。

设 $H(\text{key})$ 取值范围（值域）为 $[0, m-1]$ ，建立头指针向量 $HP[m]$ ， $HP[i]$  ( $0 \leq i \leq m-1$ ) 初值为空。

凡 $H(\text{key})=i$ 的记录都链入头指针为 $HP[i]$ 的链表。

**例10** 设 $H(\text{key})=\text{key}\%13$ ，

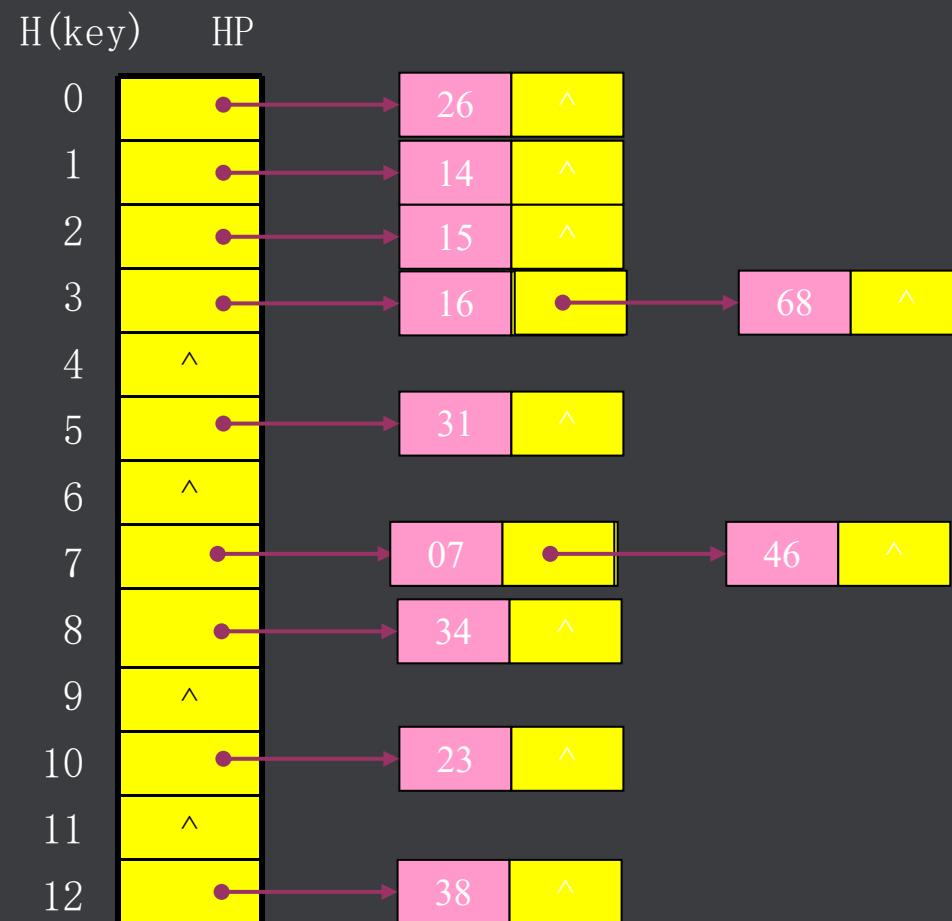
其值域为 $[0, 12]$ ，建立指针向量

$HP[12]$ 。 对例9中：

$k=\{23, 34, 14, 38, 46, 16, 68, 15, 07, 31, 26\}$

依次取其中各值，用链地址法

解决冲突时的Hash表如图：



链地址法解决冲突的优点：无聚集现象；删除表中记录容易实现。而开放地址法的Hash表作删除时，不能将记录所在单元置空，只能作删除标记。

# Hash表的查找及分析

Hash表的查找特点是：怎么构造的表就怎么查找，即造表与查找过程统一。

**算法思路：**对给定 $k$ ，根据造表时选取的 $H(\text{key})$ 求 $H(k)$ 。  
若 $H(k)$ 单元 $= \wedge$ ，则查找失败，否则 $k$ 与该单元存放的 $\text{key}$ 比较，若相等，则查找成功；若不等，则根据设定的处理冲突方法，找下一地址 $H_i$ ，直到查找到或等于空为止。

# 线性探查法解决冲突时Hash表的查找及插入

```
#define m 64 //设定表长m//
typedef struct
{ keytype key; //记录关键字//
    .....
}Hrtype;
Hrtype HT[m]; //Hash表存储空间//
int Lhashsearch(Hrtype HT[m],keytype k) //线性探查法解决冲突时的查找//
{
    int j,d,i=0;
    j=d=H(k); //求Hash地址并赋给j和d//
    while((i<m)&&(HT[j].key!=NULL)&&(HT[j].key!=k))
    { i++; j=(d+i)%m; } //冲突时形成下一地址//
    if(i==m) return(-1); //表溢出时返回-1//
    else return(j);
} //HT[j].key==k,查找成功;HT[j].key=NULL,查找失败//
void LHinsert(Hrtype HT[m], Hrtype R) //记录R插入Hash表的算法//
{
    int j=LHashsearch(HT,R.key); //查找R, 确定其位置//
    if((j== -1)|| (HT[j].key==R.key)) ERROR(); //表溢出或记录已存//
    else HT[j]=R; //插入HT[j]单元//
}
```

# 链地址法解决冲突时Hash表的查找及插入

```
typedef struct node //记录对应结点//
```

```
{    keytype key;
```

```
.....
```

```
    struct node *next;
```

```
}Renode;
```

```
Renode *LinkHsearch(Renode *HT[m],keytype k) //链地址法解决冲突时的查找//
```

```
{
```

```
    Renode *p;
```

```
    int d=H(k); //求Hash地址d//
```

```
    p=HT[d]; //取链表头结点指针//
```

```
    while(p&&(p->key!=k))
```

```
        p=p->next; //冲突时取下一同义词结点//
```

```
    return(p);
```

```
} //查找成功时p->key==k, 否则p=^//
```

# 链地址法解决冲突时Hash表的查找及插入

void LinkHinsert(Renode \*HT[m], Renode \*S)//将指针S所指记录插入表HT的算法 (如图所示)

{

int d;

Renode \*p;

p=LinkHsearch(HT,S->key); //查找S结点//

if(p) ERROR(); //记录已存在//

else

{

d=H(S->key);

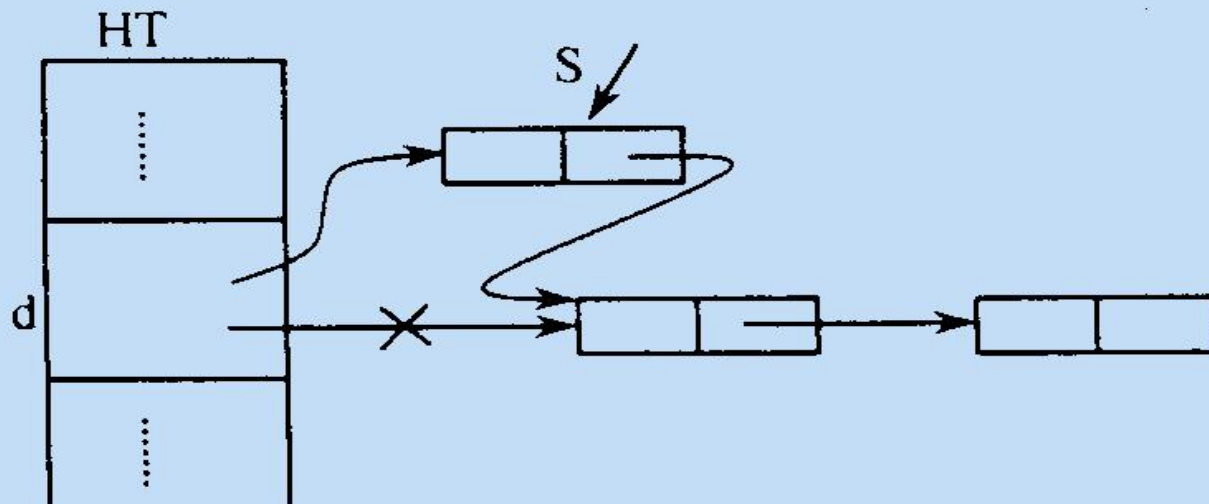
S->next=HT[d];

HT[d]=S;

}

}

插入图示:





7

排序

# 概述

排序(Sort)是将无序的记录序列（或称文件）调整成有序的序列。

对文件（File）进行排序有重要的意义。如果文件按key有序，可对其折半查找，使查找效率提高；在数据库（Data Base）和知识库（Knowledge Base）等系统中，一般要建立若干索引文件，就牵涉到排序问题；在一些计算机的应用系统中，要按不同的数据段作出若干统计，也涉及到排序。排序效率的高低，直接影响到计算机的工作效率。

# 排序定义

设含有 $n$ 个记录的文件 $f=(R_1 R_2 \dots R_n)$ ，相应记录关键字 (key) 集合 $k=\{k_1 k_2 \dots k_n\}$ 。若对 $1, 2, \dots, n$ 的一种排列：

$$P_{(1)} P_{(2)} \dots P_{(n)} \quad (1 \leq P_{(i)} \leq n, i \neq j \text{ 时}, P_{(i)} \neq P_{(j)})$$

有：  $k_{P_{(1)}} \leq k_{P_{(2)}} \leq \dots \leq k_{P_{(n)}}$  ——递增关系

或  $k_{P_{(1)}} \geq k_{P_{(2)}} \geq \dots \geq k_{P_{(n)}}$  ——递减关系

则使 $f$ 按key线性有序： $(R_{P_{(1)}} R_{P_{(2)}} \dots R_{P_{(n)}})$ ，称这种运算为排序(或分类)。

关系符“ $\leq$ ”或“ $\geq$ ”并不一定是数学意义上的“小于等于”或“大于等于”，而是一种次序关系。但为讨论问题方便，取整型数作为key，故“ $\leq$ ”或“ $\geq$ ”可看作通常意义上的符号。

# 排序定义

## 稳定排序和非稳定排序

设文件 $f = (R_1 \dots R_i \dots R_j \dots R_n)$ 中记录 $R_i$ 、 $R_j$  ( $i \neq j, i, j = 1 \dots n$ ) 的key相等, 即 $K_i = K_j$ 。若在排序前 $R_i$ 领先于 $R_j$ , 排序后 $R_i$ 仍领先于 $R_j$ , 则称这种排序是稳定的, 其含义是它没有破坏原本已有序的次序。反之, 若排序后 $R_i$ 与 $R_j$ 的次序有可能颠倒, 则这种排序是非稳定的, 即它有可能破坏了原本已有序记录的次序。

## 内排序和外排序

若待排文件 $f$ 在计算机的内存储器中, 且排序过程也在内存中进行, 称这种排序为内排序。内排序速度快, 但由于内存容量一般很小, 文件的长度

(记录个数)  $n$ 受到一定限制。若排序中的文件存入外存储器, 排序过程借助于内外存数据交换 (或归并) 来完成, 则称这种排序为外排序。我们重点讨论内排序的一些方法、算法以及时间复杂度的分析。

# 内排序方法

截止目前，各种内排序方法可归纳为以下五类：

- (1) 插入排序
- (2) 交换排序
- (3) 选择排序
- (4) 归并排序
- (5) 基数排序。

# 插入排序 (Insert Sort)

直接插入排序

折半插入排序

链表插入排序

Shell (希尔) 排序

.....

# 直接插入排序

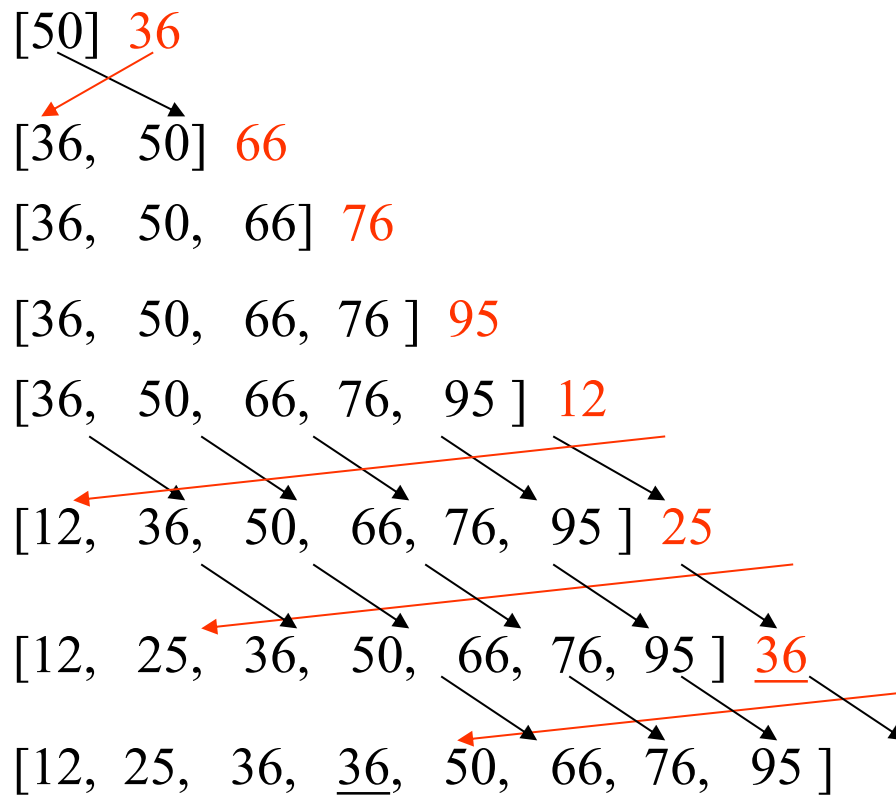
设待排文件 $f = (R_1 R_2 \dots R_n)$  相应的key集合为 $k = \{k_1 k_2 \dots k_n\}$ , 文件 $f$ 对应的结构可以是前面所述的三种文件结构之一。

## 排序方法

先将文件中的 $(R_1)$  看成只含一个记录的有序子文件, 然后从 $R_2$ 起, 逐个将 $R_2$ 至 $R_n$ 按key插入到当前有序子文件中, 最后得到一个有序的文件。插入的过程上是一个key的比较过程, 即每插入一个记录时, 将其key与当前有序子表中的key进行比较, 找到待插入记录的位置后, 将其插入即可。另外, 假定排序后的文件按递增次序排列 (以下同)。

# 直接插入排序

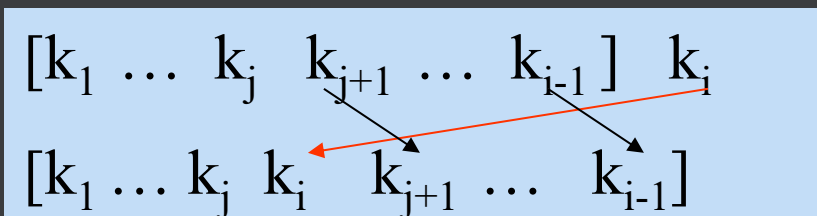
**例1** 设文件记录的key集合 $k=\{50, 36, 66, 76, 95, 12, 25, \underline{36}\}$ （考虑到对记录次key排序的情况，允许多个key相同。如此例中有2个key为36，后一个表示成36，以示区别），按直接插入排序方法对k的排序过程如下： $k=\{50, 36, 66, 76, 95, 12, 25, \underline{36}\}$





# 直接插入排序

一般, 插入 $k_i$  ( $2 \leq i \leq n$ ), 当 $k_j \leq k_i < k_{j+1} \dots k_{i-1}$ 时, 先将子表 $[k_{j+1} \dots k_{i-1}]$ 从 $k_{i-1}$ 起顺序向后移动一个记录位置, 然后 $k_i$ 插入到第 $j+1$ 位置, 即:



文件结构说明:

```
#define maxsize 1024  //文件最大长度//
```

```
typedef int keytype;  //设key为整型//
```

```
typedef struct  //记录类型//
```

```
{ keytype key;  //记录关键字//
```

```
.....
```

```
.....  //记录其它域//
```

```
}Rtype;
```

```
typedef struct  //文件或表的类型//
```

```
{ Rtype R[maxsize+1];  //文件存储空间//
```

```
int len;  //当前记录数//
```

```
} sqfile;
```

若说明sqfile F; 则:  $(F.R[1], F.R[2] \dots F.R[F.len])$  为待排文件, 它是一种顺序结构; 文件长度 $n = F.len$ ;  $F.R[0]$ 为工作单元, 起到“监视哨”作用; 记录的关键字 $k_i$ 写作 $F.R[i].key$ 。

# 直接插入排序算法描述

```
void Insort (sqfile F)  //对顺序文件F直接插入排序的算法//
{ int i, j;
  for (i=2; i<=F.len; i++)  //插入n - 1个记录//
  { F.R[0] = F.R[i]; //待插入记录先存于监视哨//
    j = i-1;
    while (F.R[0].key < F.R[j].key) //key比较//
    { F.R[j+1] = F.R[j]; //记录顺序后移//
      j--;
    }
    F.R[j+1] = F.R[0]; //原R[i]插入j+1位置//
  }
}
```

排序的时间复杂度取耗时最高的量级，故直接插入排序的 $T(n)=O(n^2)$ 。

# 折半插入排序

排序算法的 $T(n)=O(n^2)$ ,是内排序时耗最高的时间复杂度。随着文件记录数 $n$ 的增大,效率降低较快。下面的一些插入排序的方法,都是围绕着改善算法的时间复杂度而展开的。另外,直接插入排序是稳定排序。

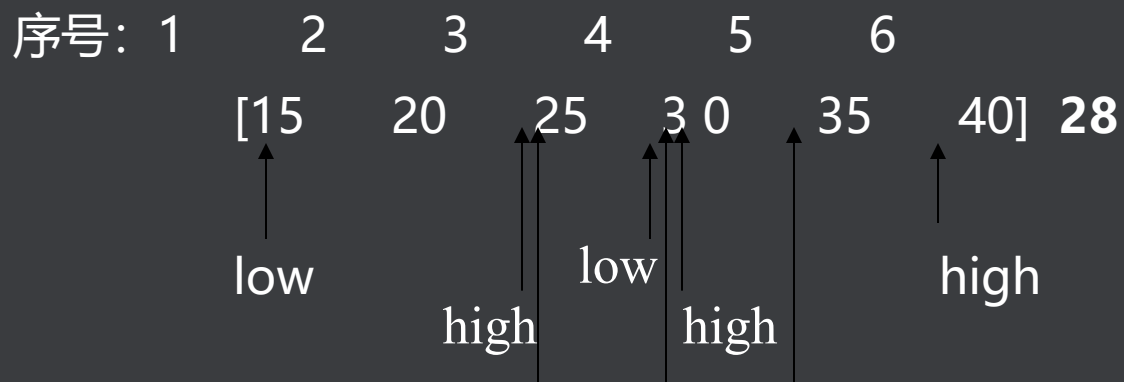
为减少插入排序过程中key的比较次数,可采用折半插入排序。

## 排序方法

同直接插入排序,先将 $(R[1])$ 看成一个子文件,然后依次插入 $R[2].....R[n]$ 。但在插入 $R[i]$ 时,子表 $[R[1].....R[i-1]]$ 已是有序的,查找 $R[i]$ 在子表中的位置可按折半查找方法进行,从而降低key的比较次数。

# 折半插入排序

例2 设当前子表key序列及插入的 $k_i=28$ 如下：



令：

$$mid = \lfloor (low + high) / 2 \rfloor$$

$28 > R[3].key = 25$ ,  $low = 3 + 1 = 4$ ;

$28 < R[5].key = 35$ ,  $high = 5 - 1 = 4$ ;

$28 < R[4].key = 30$ ,  $high = 4 - 1 = 3$ 。  
此时， $low > high$ ，所以“28”应  
插在 $low=4$ 所指示的位置。

# 折半插入排序算法描述

void Binsort (sqfile F) // 对文件F折半插入排序的算法 //

```
{  int i, j, low, high, mid;
  for (i=2; i<=F.len; i++)  // 插入n - 1个记录 //
  {  F.R[0] = F.R[i];  // 待插入记录存入监视哨 //
    low = 1; high = i-1;
    while (low <= high)  // 查找R[i]的位置 //
    {  mid = (low+high) / 2;
      if (F.R[0].key >= F.R[mid].key) low = mid+1;  // 调整下界 //
      else high = mid-1;  // 调整上界 //
    }
    for (j=i-1; j>=low; j--)
      F.R[j+1] = F.R[j];  // 记录顺移 //
    F.R[low] = F.R[0];  // 原R[i]插入low位置 //
  }
}
```

插入R[i] ( $2 \leq i \leq n$ ) 时, 子表记录数为i - 1。同折半查找算法的讨论, 查找R[i]的key比较次数为  $\log_2 i$ , 故总的key比较次数C为:

$$C = \sum_{i=2}^n \log_2 i < (n-1) \log_2 n = O(n \log_2 n)$$

显然优于 $O(n^2)$ 。但折半插入排序的记录移动次数并未减少, 仍为 $O(n^2)$ 。故算法的时间复杂度T(n)仍为 **$O(n^2)$** 。

# 链表插入排序

设待排序文件 $f = (R_1 R_2 \dots R_n)$ ，对应的存储结构为单链表结构，如图：

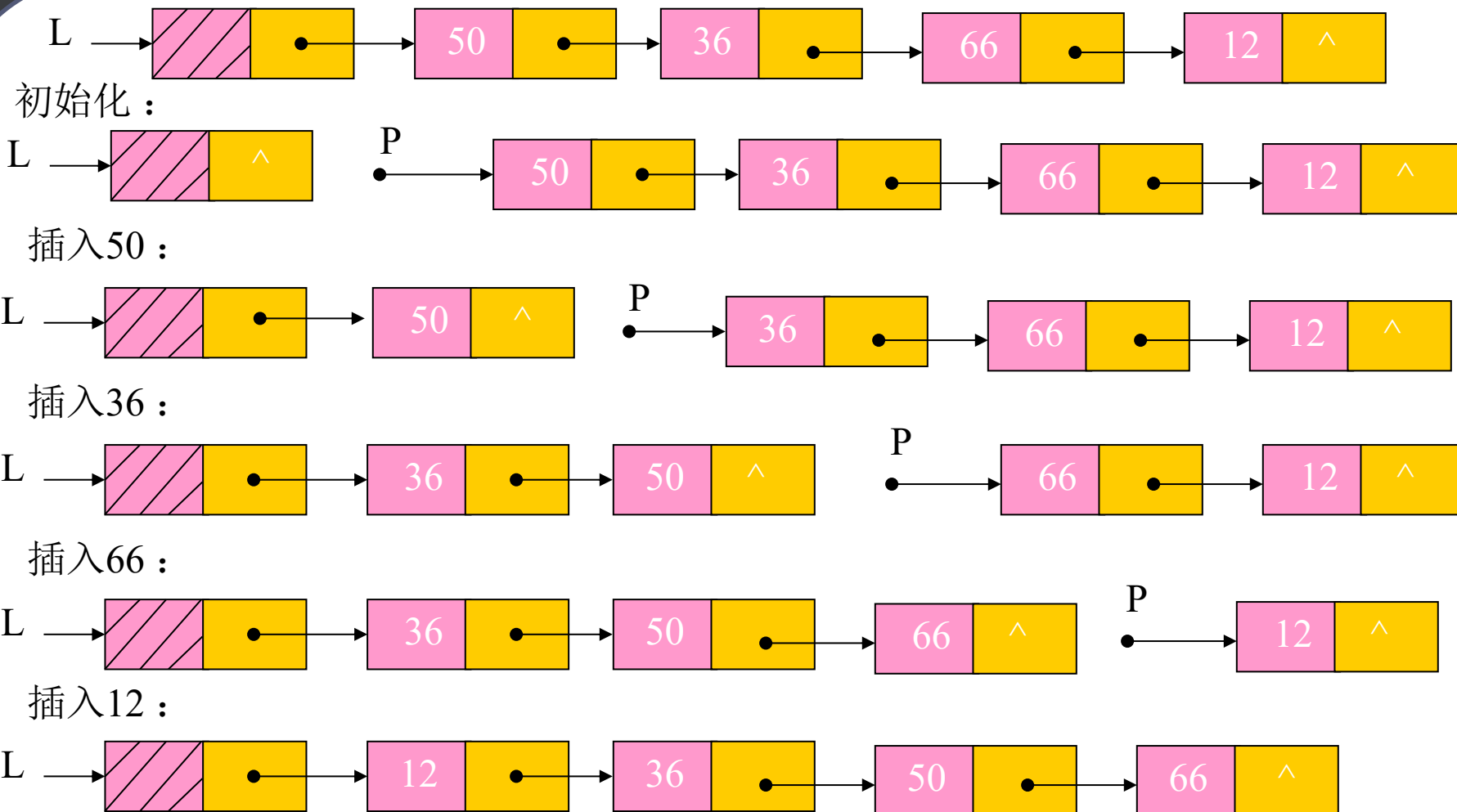


## 排序方法

链表插入排序实际上是一个对链表遍历的过程。先将表置为空表，然后依次扫描链表中每个结点，设其指针为 $p$ ，搜索到 $p$ 结点在当前子表的适当位置，将其插入。

# 链表插入排序

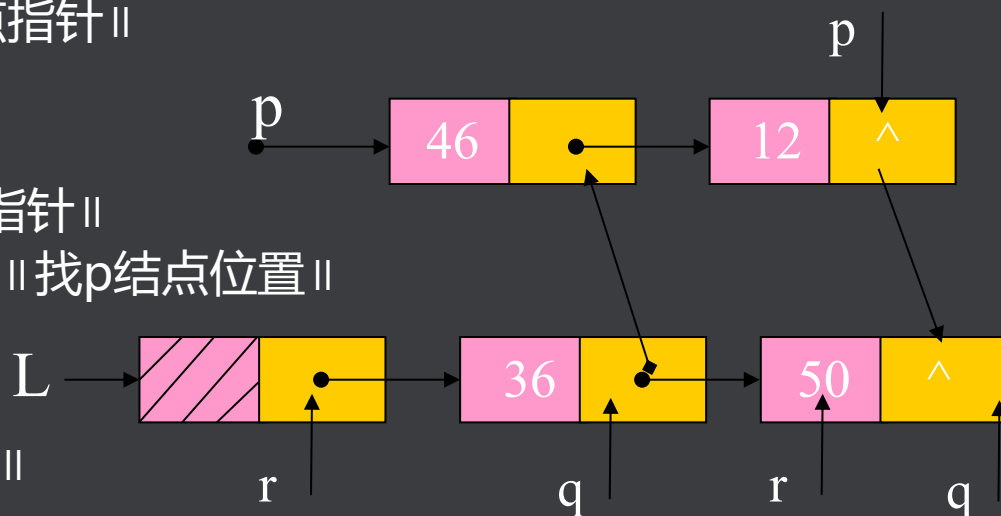
例3 设含4个记录的链表如图：



# 链表插入排序算法描述

```
typedef struct node_t  //存放记录的结点//
{ keytype key;  //记录关键字//
  .....  //记录其它域//
  struct node_t *next;  //链指针//
} linkode_t,*linklist_t;

void Linsertsort (linklist_t L)  //链表插入排序算法//
{ linklist_t p, q, r, u;
  p = L->next;  //p为待插入结点指针//
  L->next=NULL;  //置子表为空//
  while ( p )  //若待排序结点存在//
  { r = L; q = L->next;  //r为q的前驱指针//
    while (q && q->key<=p->key)  //找p结点位置//
    { r = q; q = q->next; }
    u = p->next;
    p->next = q; r->next = p;  //插入//
    p=u;
  }
}
```





# 交换排序

“起泡” 排序 (Bubble Sort)

“快速” 排序 (Quick Sort)

# 起泡排序

设待排文件 $f = (R_1 R_2 \dots R_n)$ ，相应key集合 $k = \{k_1 k_2 \dots k_n\}$ 。

## 排序方法

从 $k_1$ 起，两两key比较，逆序时交换，即：

$k_1 \sim k_2$ ，若 $k_1 > k_2$ ，则 $R_1 \leftrightarrow R_2$ ；

$k_2 \sim k_3$ ，若 $k_2 > k_3$ ，则 $R_2 \leftrightarrow R_3$ ；

.....

$k_{n-1} \sim k_n$ ，若 $k_{n-1} > k_n$ ，则 $R_{n-1} \leftrightarrow R_n$ 。

经过一趟比较之后，最大key的记录沉底，类似水泡。接着对前 $n - 1$ 个记录重复上述过程，直到排序完毕。

起泡排序的时间复杂度 $T(n) = O(n^2)$ 。

注意：在某趟排序的比较中，若发现两两比较无一记录交换，则说明文件已经有序，不必进行到最后两个记录的比较。

# 起泡排序

例5 设记录key集合 $k=\{50, 36, 66, 76, 95, 12, 25, 36\}$ , 排序过程如下:

| K         | 第1趟       | 第2趟       | 第3趟       | 第4趟       | 第5趟      | 第6趟 |      |
|-----------|-----------|-----------|-----------|-----------|----------|-----|------|
| 50        | 36        | 36        | 36        | 36        | 12       | 12  | 排序完毕 |
| 36        | 50        | 50        | 50        | 12        | 25       | 25  |      |
| 66        | 66        | 66        | 12        | 25        | 36       | 3   |      |
| 76        | 76        | 12        | 25        | <u>36</u> | <u>3</u> | 6   |      |
| 95        | 12        | 25        | <u>36</u> | 5         | <u>6</u> |     |      |
| 12        | 25        | <u>36</u> | 6         | 0         |          |     |      |
| 25        | <u>36</u> | 7         | 6         |           |          |     |      |
| <u>36</u> | 9         | 6         |           |           |          |     |      |
|           | 5         |           |           |           |          |     |      |

从此例可以看出, 起泡排序属于稳定排序。

# 起泡排序算法描述

```
void Bubblesort (sqfile F)  //对顺序文件起泡排序的算法 //
{  int i, flag;  //flag为记录交换的标志 //
    Retype temp;
    for (i=F.len; i>=2; i--)  //最多n - 1趟排序 //
    {  flag = 0;
        for (j=1; j<=i-1; j++)  //一趟的起泡排序 //
        if (F.R[j].key > F.R[j+1])  //两两比较 //
        {  temp = F.R[j];  //R[j]  $\leftrightarrow$  R[j+1] //
            F.R[j] = F.R[j+1];
            F.R[j+1] = temp;
            flag=1;
        }
        if (flag == 0) break;  //无记录交换时排序完毕 //
    }
}
```

# 快速排序

快速排序是对起泡排序的一种改进，目的是提高排序的效率。

## 排序方法

经过key的一趟比较后，确定某个记录在排序后的最终位置。

设待排文件的key集合 $k=\{k_1 \ k_2 \dots k_i \dots k_j \dots k_{n-1} \ k_n\}$ ，对 $k$ 中的 $k_1$ ，称作枢轴（Pivot）或基准。

(1) **逆序比较**： $k_1 \sim k_n$ ，若 $k_1 \leq k_n$ ，则 $k_1$ 不可能在 $k_n$ 位置， $k_1 \sim k_{n-1}$ ，.....直到有个 $k_j$ ，使得 $k_1 > k_j$ ，则 $k_1$ 有可能落在 $k_j$ 位置，将 $k_j \Rightarrow k_1$ 位置，即key比基准（ $k_1$ ）小的记录向左移。

(2) **正序比较**： $k_1 \sim k_2$ ，若 $k_1 > k_2$ ，则 $k_1$ 不可能在 $k_2$ 位置， $k_1 \sim k_3$ ，.....直到有个 $k_i$ ，使得 $k_1 < k_i$ ，则 $k_1$ 有可能落在 $k_i$ 位置，将 $k_i \Rightarrow k_1$ 位置（原 $k_j$ 已送走），即key比基准大的记录向右移。

反复逆序、正序比较，当 $i=j$ 时， $i$ 位置就是基准 $k_1$ 的最终落脚点（因为比基准小的key统统在其“左部”，比基准大的统统在其“右部”，作为基准的key自然落在排序后的最终位置上），

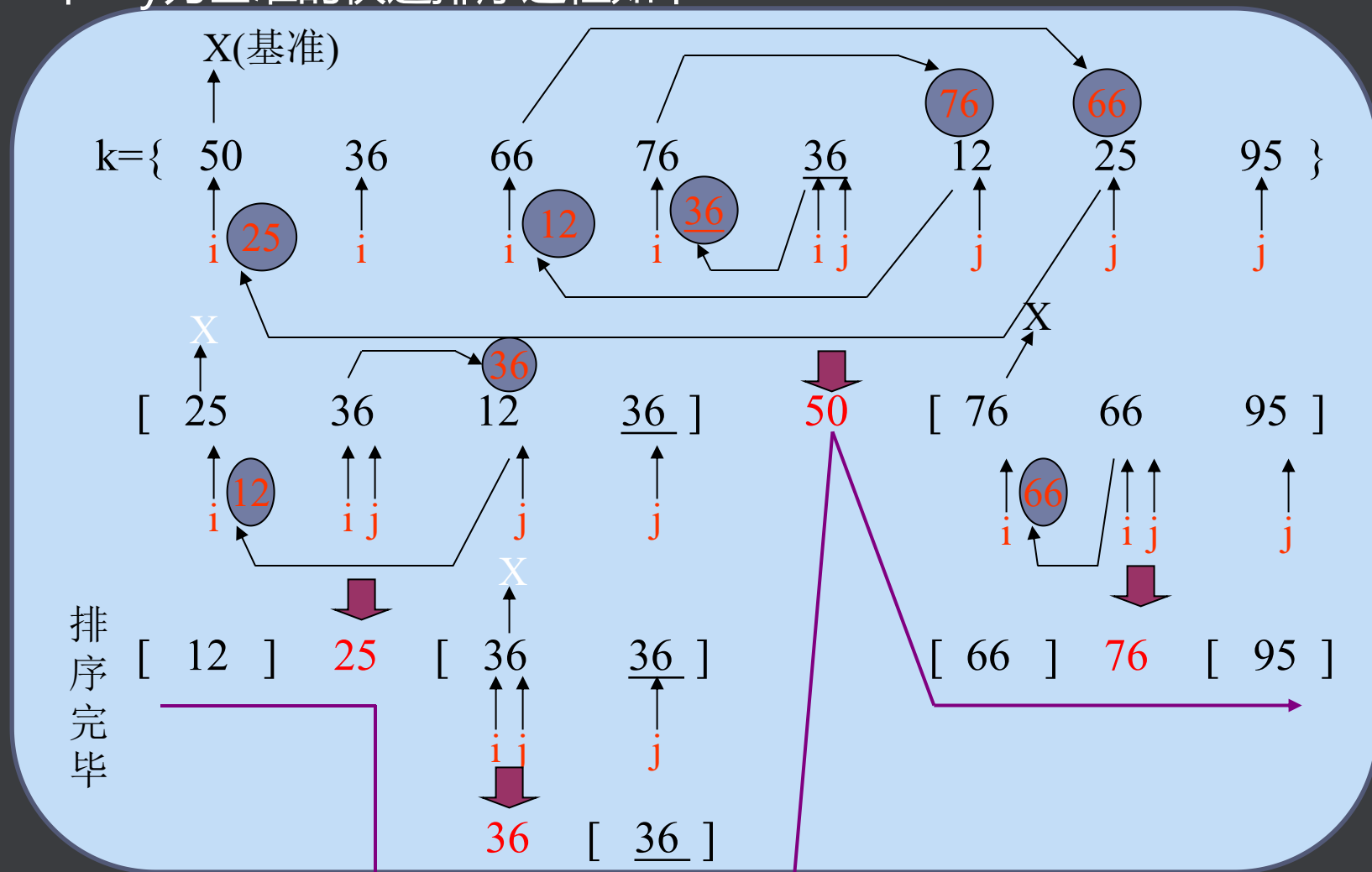
并且 $k_1$ 将原文件分成了两部分：

|      |       |       |
|------|-------|-------|
| $k'$ | $k_1$ | $k''$ |
| 左部   |       | 右部    |

对 $k'$  和 $k''$ ，套用上述排序过程(可递归)，直到每个子表只含有一项时，排序完毕。

# 快速排序

例6 设记录的key集合 $k=\{50, 36, 66, 76, 36, 12, 25, 95\}$ , 每次以集合中第一个key为基准的快速排序过程如下:



# 快速排序算法描述

```
typedef struct  //栈元素类型//
{ int low, high;  //当前子表的上下界//
} stacktype;

int qkpass(sqfile F, int low, int high) //对子表(R[low].....R[high])一趟快排算法//
{ int i = low, j = high;
  keytype x = F.R[low].key; //存入基准key//
  F.R[0] = F.R[low];          //存入基准记录//
  while (i < j)
  { while (i < j && x <= F.R[j].key) j--; //逆序比较//
    if (i < j) F.R[i] = F.R[j]; //比x小的key左移//
    while (i < j && x >= F.R[i].key) i++; //正序比较//
    if (i < j) F.R[j] = F.R[i]; //比x大的key右移//
  }
  F.R[i] = F.R[0]; //基准记录存入第i位置//
  return (i); //返回基准位置//
}
```

# 快速排序算法描述

```
void qksort (sqfile F)  //对文件F快速排序的算法（非递归）  //
{  int i, low, high;
    stacktype u;    //栈元素//
    S = CreateStack(); //置栈空//
    u.low = 1; u.high = F.len; PushStack(S, u); //上下界初值进栈//
    while ( ! EmptyStack (S) )
    { u = PopStack(S);    //退栈//
        low = u.low; high = u.high; //当前表的上下界//
        while (low < high)
        { i = qkpass(F, low, high); //对当前子表的一趟快排//
            if (i+1 < high)
            { u.low = i+1; u.high = high; PushStack (S, u); } //i位置的右部上下界进栈//
            high = i - 1; //排当前左部//
        }
    }
}
```



