

# 数据结构复习概要

# 第一章绪论

程序设计=算法+数据结构

数据结构是讨论非数值类问题的对象描述、  
信息组织方法及其相应的操作

数据元素、数据对象、关系

数据结构：特性相同的数据元素构成的集合中，如果在数据元素之间存在一种或多种特定的关系，则称之为数据结构。

– **D t -Structure=(D,S)**

– 四类基本结构：1)纯集合 2)线性 3)树形 4)网状

# 抽象数据类型Abstract Data Type

- **ADT= (D, S, P)**
- **D**是数据对象,
- **S**是**D**上的关系的集合,
- **P**是对**D**的基本操作的集合

操作中的参数有两种

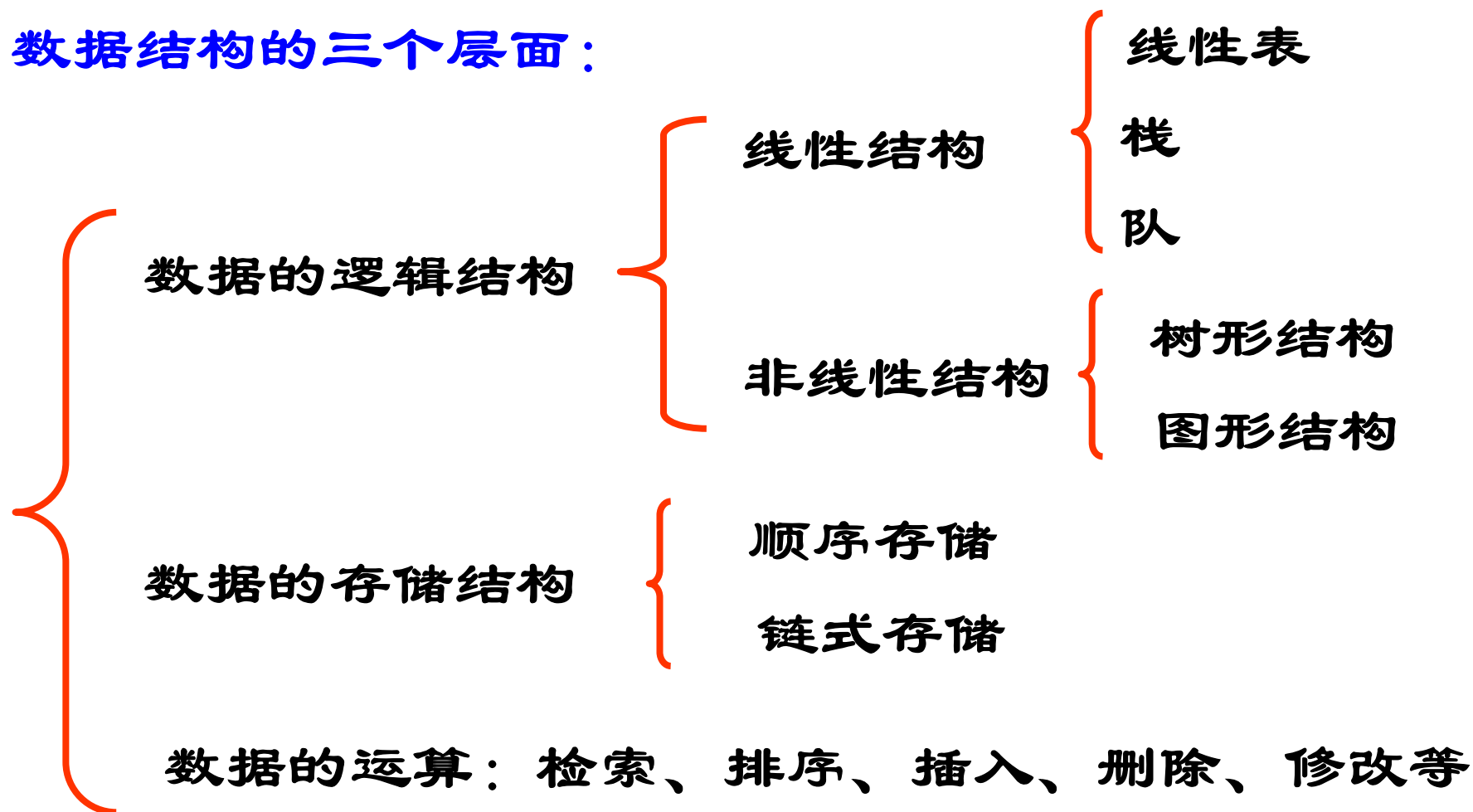
## 1) 赋值参数

```
void dd (int , int b, int * c){*c= +b;}  
dd(2,3,&c);
```

## 2) 引用参数

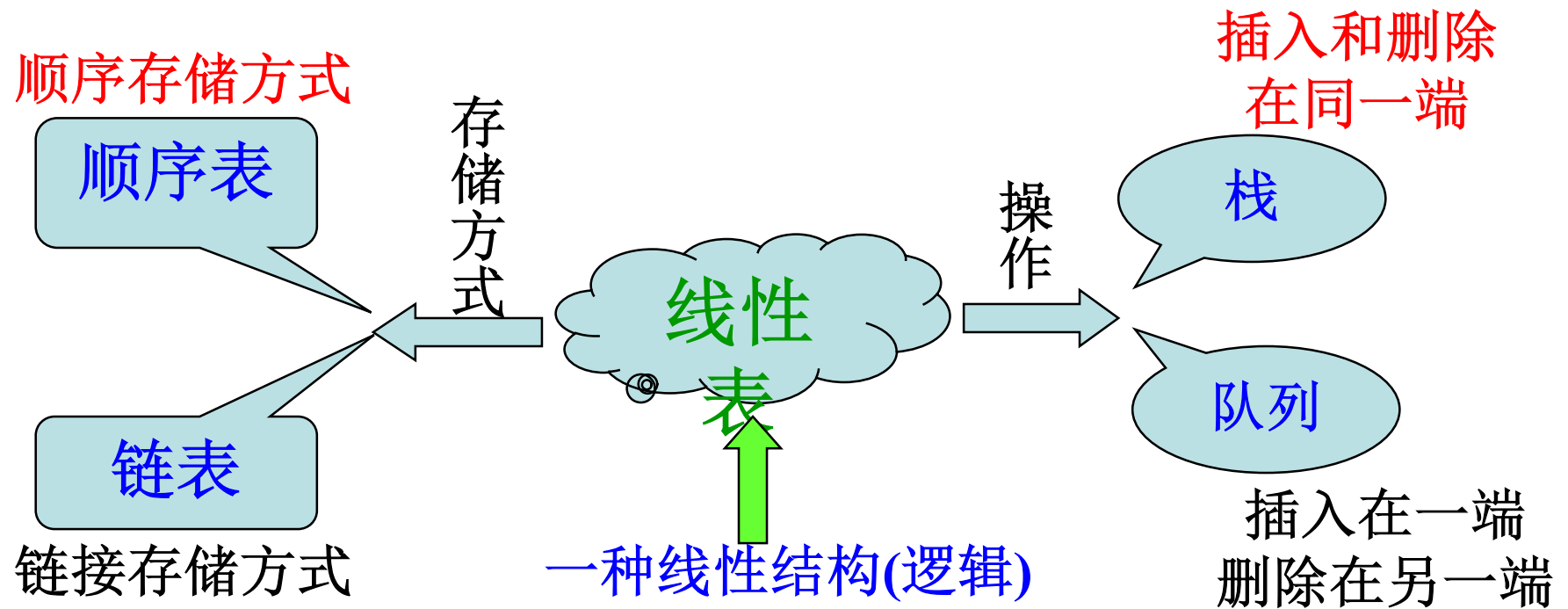
```
void dd (int , int b, int &c){c= +b}  
dd(2,3,c);
```

## 数据结构三个层面：



# 数据结构的命名

数据结构具有三个层次，数据结构命名的时候将数据的逻辑结构、存储结构甚至操作结合起来给数据结构进行命名。



# 算法和算法分析

1、算法:对问题求解步骤的描述，是一个确定的、有限长的操作序列。

1) 有穷性 2) 确定性 3) 可行性 4) 输入 5) 输出

2、算法设计要求

1) 正确性 2) 可读性 3) 健壮性 4) 效率与低存储需求

3、算法的描述:类C语言

4、算法效率衡量方法与准则：

时间复杂度： $T(n) = O(f(n))$

空间复杂度： $S(n) = O(g(n))$

# “O” 的形式定义

若 $f(n)$ 是正整数 $n$ 的一个函数，则 $x_n = O(f(n))$ 表示存在一个正的常数 $M, m$ ，使得当 $n \geq n_0$ 时都满足

$$m|f(n)| \leq |x_n| \leq M|f(n)|;$$

换句话说就是说这当整型自变量 $n$ 趋向于无穷大时，两者的比值是一个不等于0的常数。

$$\lim_{n \rightarrow \infty} \frac{|x_n|}{|f(n)|} = C$$

分析下列三个程序段：

①、  $x++; s=0;$   $O(1)$

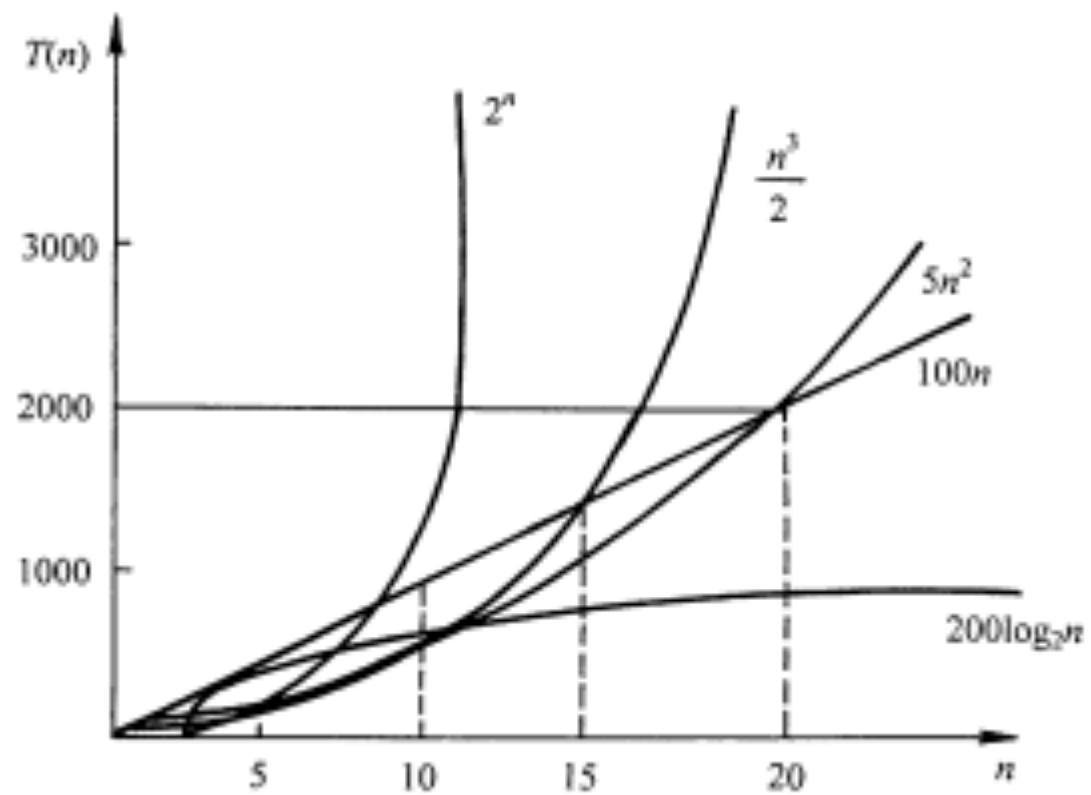
②、  $\text{for } (i=1; i \leq n; i++) \{x++; s+=x;\}$   $O(n)$

③、  $\text{for}( i=1; i \leq n; i++)$   
     $\text{for } (j=1; j \leq n; j++) \{x++; s+=x;\}$   $O(n^2)$

④、  $f(n)=2n^3+5n^2+7n+c$ ; ( $c$ 为某一常数) , 则

$T(n) = O(f(n))=O(n^3)$  。





常见函数的增长率

## 第二章 线性表

### 线性表的定义

线性表是 $n(n \geq 0)$ 个数据元素的有限序列，表中各个元素具有相同地属性，相邻元素间存在“序偶”关系。

记做： $(a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_{n-1}, a_n)$

$a_{i-1}$ 称为 $a_i$ 的直接前驱元素，

$a_{i+1}$ 是 $a_i$ 的直接后继元素

线性表的长度：表中的元素个数  $n$

位序： $i$ 称元素 $a_i$ 在线性表中的位序

ADT List{

数据对象:  $D=\{a_i | a_i \in \text{Elemset}, i=1, 2 \dots n, n \geq 0\}$

数据关系:  $R=\{ \langle a_{i-1}, a_i \rangle | a_{i-1}, a_i \in D, i=2, \dots n \}$

基本操作:

... ..

GetElem(L,i,&e)      $1 \leq i \leq \text{ListLength}(L)$

LocateItem(L,e,compare())

ListInsert(&L,i,e)      $1 \leq i \leq \text{ListLength}(L)+1$

ListDelete(&L,i,&e)      $1 \leq i \leq \text{ListLength}(L)$

... ..

}ADT List

# 顺序表——线性表的顺序存储表示

```
typedef Struct {  
    Elemtype          * elem;  
    int                length;  
    int                listsize;  
} SqList;
```

插入和删除操作的时间分析:

$$E_{in} = \sum p_i(n-i+1) = n/2$$

$$E_{del} = \sum p_i(n-i) = (n-1)/2$$

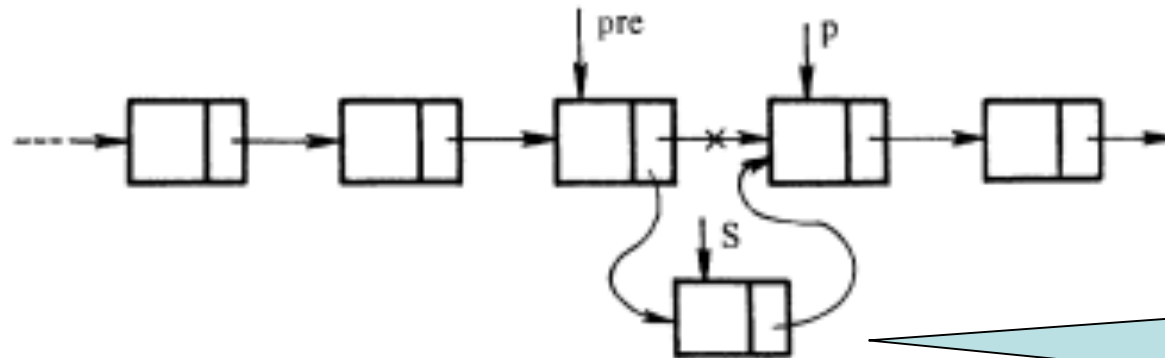
# 线性表的链式表示和实现

## 线性链表

- 数据域（**data**）和指针域（**next**）
- 存储表示

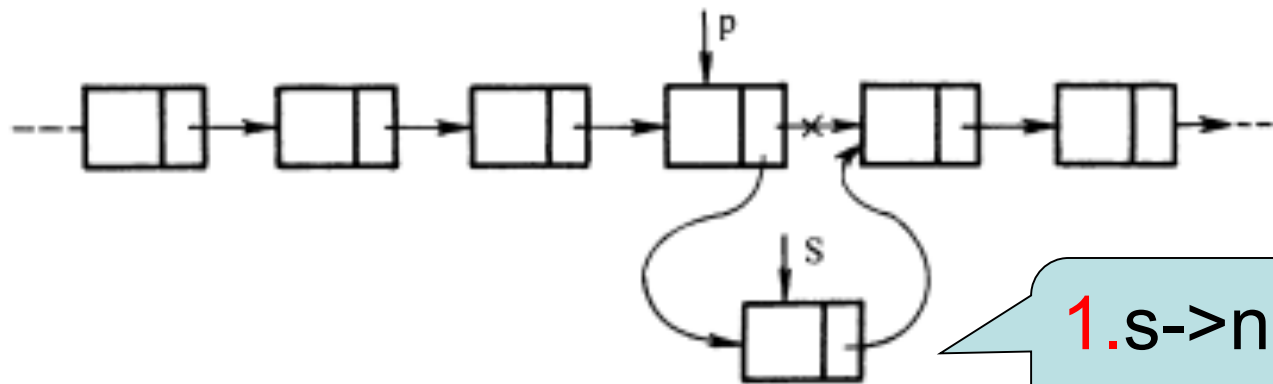
```
typedef struct LNode{  
    ElemType      data;  
    Struct LNode  *next;  
}LNode, *LinkList;
```

# 插入结点操作：前插、后插



(a) 插入到 P 之前(必须知道 p 的前驱 pre)

`pre->next=s;`  
`s->next=p;`



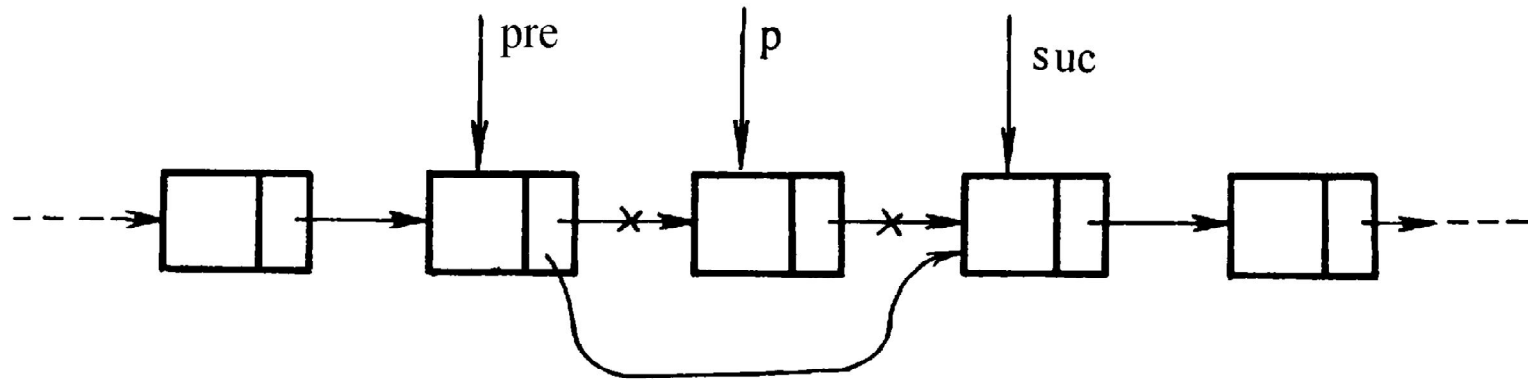
(b) 插入到 P 之后(不须知道 P 的前驱)

1. `s->next=p->next;`  
2. `p->next=s;`

时间复杂度：前插 $O(n)$ 、后插 $O(1)$

单链表插入：头部？尾部？

## 删除节点p



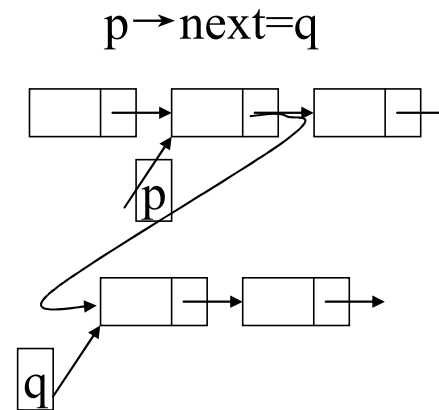
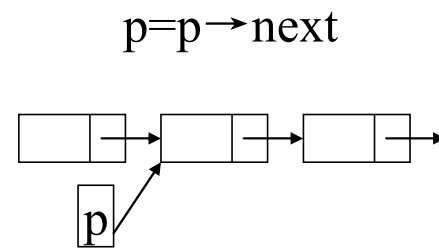
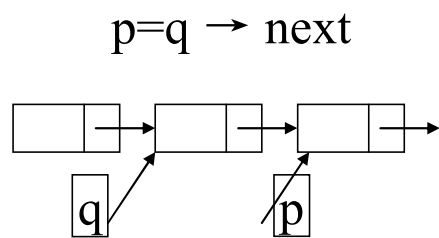
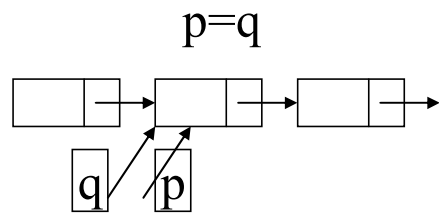
```
if(p==L)L=p->next;    //p是第一个结点
else {                //p不是第一个结点
    q=L;
    while(q&&q->next!=p)q=q->next;
    q->next=p->next;
}
delete p;
```

## 例：查找p前驱

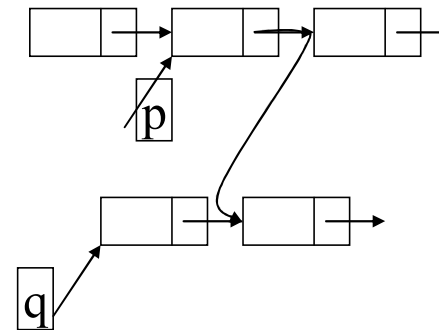
```
LNode *GetPre(LinkList L, LNode *p)
{
    if(p==L) return NULL;
    q=L;
    while(q->next!=p)q=q->next;
    rerurn q;
}
```



# 常见指针操作



$p \rightarrow \text{next}=q \rightarrow \text{next}$



# 循环链表

- 什么是循环链表

- 通常循环链表有头结点
- 最后一个结点的指针指向头结点
- 一般设置尾指针，而不设头指针，也可以都设
- 空的循环链表是头结点指针指向自己

两个带头结点的循环有序链表，表示集合A、B，完成 $C=A \cup B$ 的时间复杂度  $O(n+m)$ ，是归并排序的基础。

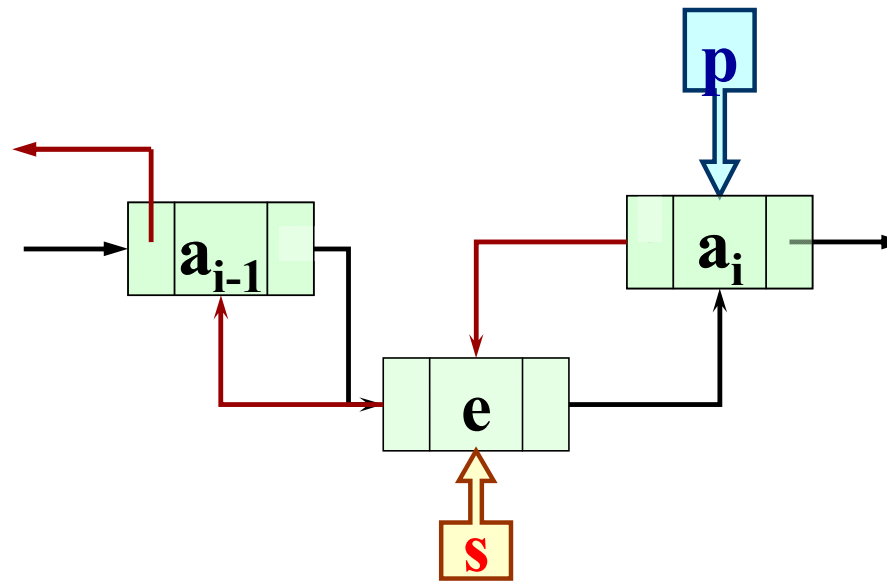
# 双向链表

概念：两个指针，分别指向前驱和后继，双向链表一般都是循环的。

```
typedef struct DuLNode{  
    ElemType    data;  
    struct DuLNode    *prior;  
    struct DuLNode    *next;  
}DuLNode, *DuLinkList;
```

## 插入结点操作 时间复杂度 $O(1)$

插入



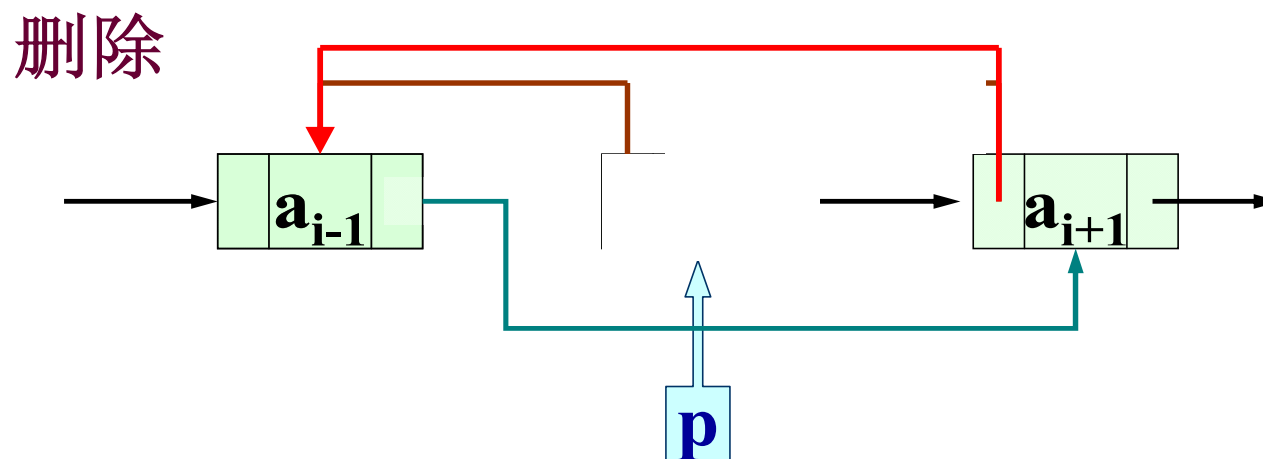
**$s \rightarrow \text{prior} = p \rightarrow \text{prior};$**

**$p \rightarrow \text{prior} \rightarrow \text{next} = s;$**

**$s \rightarrow \text{next} = p;$**

**$p \rightarrow \text{prior} = s;$**

## 删除结点操作 时间复杂度 $O(1)$



$p \rightarrow \text{prior} \rightarrow \text{next} = p \rightarrow \text{next};$

$p \rightarrow \text{next} \rightarrow \text{prior} = p \rightarrow \text{prior};$

$\text{free}(p);$

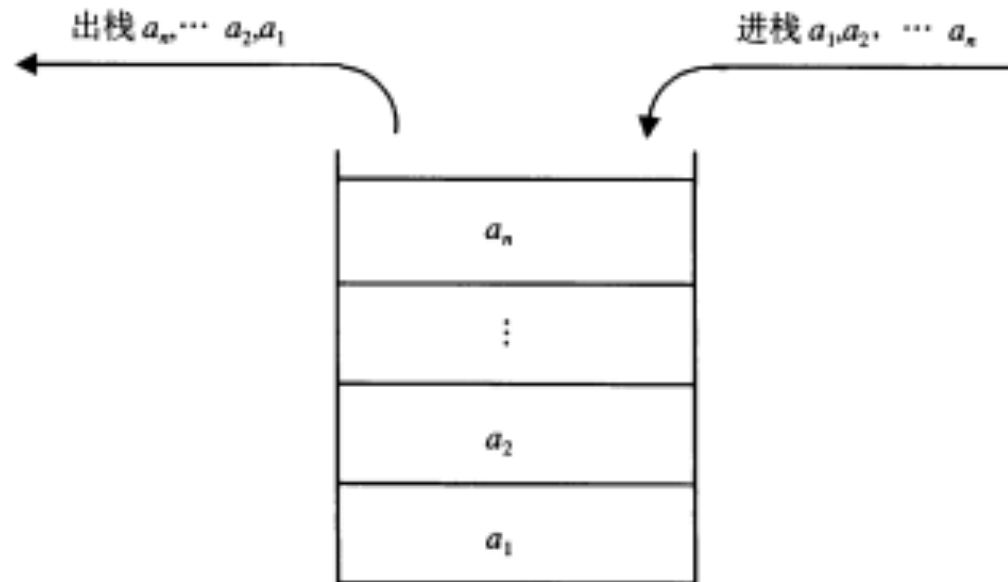
# 顺序表和链表的综合比较

- 线性表的长度能否预先确定？处理过程中变化范围如何？
  - 长度确定、变化小时用顺序表
  - 长度变化大、难以估计最大长度时宜采用链表
- 对线性表的操作形式如何？
  - 查询操作多、删除和插入操作少时使用顺序表
  - 频繁插入和删除操作宜采用链表

# 栈和队列

栈（**Stack**）是限定只能在表得一端进行插入和删除操作的线性表，又称限定性线性表结构

– 栈顶(Top)、栈底(Bottom),先入后出(LIFO)



ADT Stack{

数据对象: $D=\{a_i | a_i \in \text{ElemSet}, i=1,2,\dots,n, n \geq 0\}$

数据关系: $R1=\{ \langle a_{i-1}, a_i \rangle | a_{i-1}, a_i \in D, i=2,3,\dots,n \}$

基本操作:

.....

GetTop(S,&e)

Push(&S, e)

Pop(&S,&e)

.....

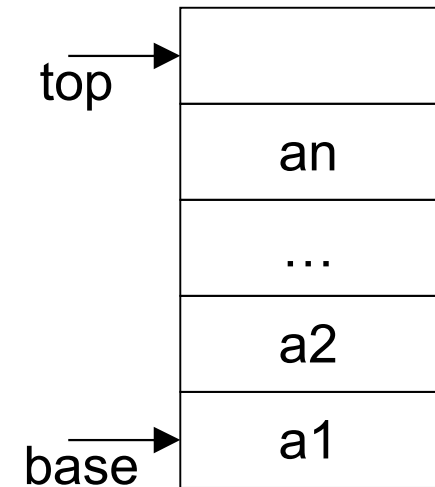
}ADT Stack



# 顺序栈

```
typedef struct{  
    SElemtype    *base;  
    SElemtype    *top;  
    int          stacksize;  
}SqStack;
```

- **top**指向栈顶元素的下一个单元!
- 空栈:  $top == base$
- 满栈:  $top - base == stacksize$



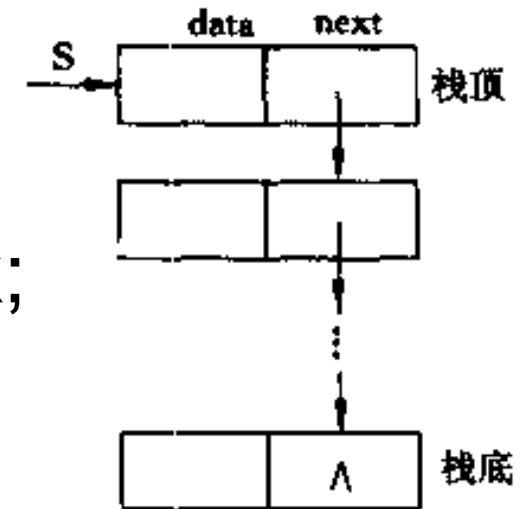
# 顺序栈实现

```
Status GetTop_Sq(SqStack S, SelemType &e){  
    ...e=*(S.top-1);...;  
}  
Status Push_Sq(SqStack &S ,SelemType e) {  
    ... *S.top++=e;    ...  
}  
Status Pop_Sq(SqStack S ,SelemType &e){  
    ... e=*--S.top;    ...  
}
```

# 链栈

```
typedef LinkList LinkStack;
```

- 空栈:  $S = \text{NULL}$ ;
- 满栈: 没有限制, 取决于内存;



```
Status GetTop_L(Linkstack S, SElemType &e){  
    if(!S) return ERROR;  
    e=S->data;  
    return OK;  
}
```

# 栈问题常见类型

一个栈s的入栈元素序列是1,2,3,4,5,6，出栈序列为3,5,4,6,2,1，栈s至少需要容纳几个元素？

答：4个

若让a、b、c依次入栈，那么下列那种出栈s次序不会出现：

A.cba    **B.cab**    C.bac    D. acb

# 递归问题

递归问题策略：

- 把 $n$ 阶规模问题，描述为通过 $n-1$ 阶规模问题来解决
- 解决简单规模问题，如1阶规模问题，或某个特殊条件(递归退出条件)

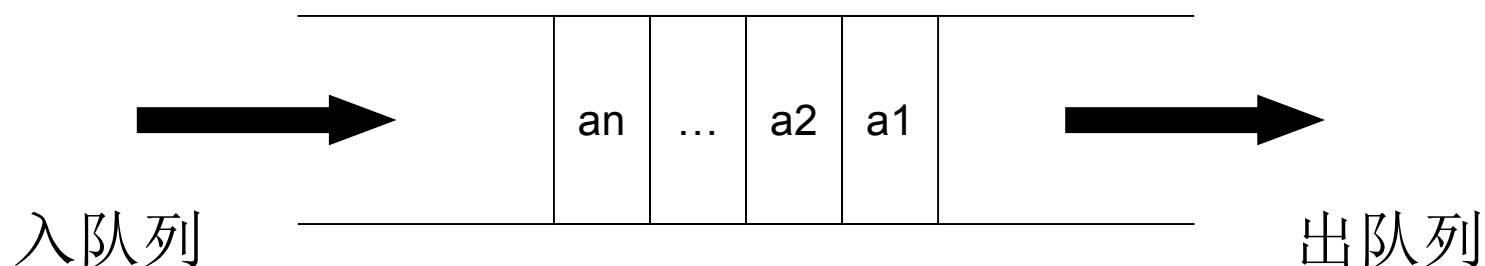
例：求 $n$ 阶乘。① $f(n)=n*f(n-1)$  ② $f(1)=0$

```
int f(int n){  
    if(n==1)return 1;  
    return n*f(n-1);  
}
```

# 队列

队列(**Queue**)是限定只能在表得一端进行插入在表的另一端进行删除操作的线性表。

- 队首(**front**)、队尾(**rear**)，先入先出(**FIFO**)



## ADT Queue{

数据对象: $D=\{a_i | a_i \in \text{ElemSet}, i=1,2,\dots,n, n \geq 0\}$

数据关系: $R=\{ \langle a_{i-1}, a_i \rangle | a_{i-1}, a_i \in D, i=2,\dots,n \}$

基本操作:

....

GetHead(Q,&e);

EnQueue(&Q,e);

DeQueue(&Q,&e);

....

}

# 链队列

```
typedef Linklist Queueptr;
```

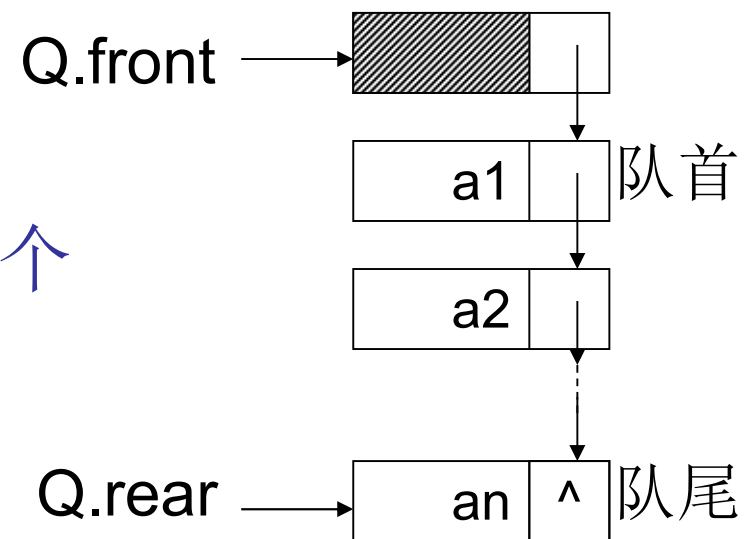
```
typedef struct {
```

```
    Queueptr  front; //头结点指针,不是队首指针
```

```
    Queueptr  rear;  //队尾指针
```

```
}LinkQueue;
```

—注意：删除的元素为最后一个元素时应改写**rear**指针

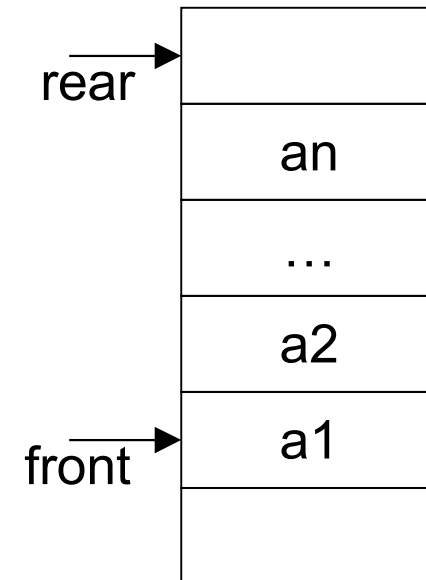




# 循环队列

```
typedef struct {  
    QElemtype    *base;  
    int          front;  
    int          rear;  
}SqQueue;
```

- rear指向队尾元素下一单元
- front指向队首元素（如有）
- 插入 rear++ ;删除front++
- 空队列: front==rear;
- 满队列: (rear+1)%Qsize==front



# 串和数组

字符串:由零个或多个字符组成的有限序列;  
是数据元素为字符的线性表。

记作:  $S = \text{“}a_0a_1\text{.....}a_{n-1}\text{”}$

- 串的长度:  $n$
- 空串:  $n=0$ , Null String
- 子串与主串, 子串的位置
- 串的比较: 最大相等前缀子序列

## ADT String{

数据对象: $D=\{a_i | a_i \in \text{CharacterSet}, i=1,2,\dots,n, n \geq 0\}$

数据关系: $R=\{ \langle a_{i-1}, a_i \rangle | a_{i-1}, a_i \in D, i=2,\dots,n \}$

基本操作:

StrAssign(&T,chars)	strcpy
StrCopy(&T,S)	strcpy
StrEmpty(S)	strlen(S)==0
StrCompare(S,T)	strcmp
StrLength(S)	strlen
ClearString(&S)	strcpy
Concat(&T,S1,S2)	strcat
Substring(&Sub,S,Pos,len)	strcpy 0<=pos<=Strlength(S)-1
Index(S, T, pos)	strstr 0<=pos<=Strlength(S)-1
Replace(&S,T,V)	strstr strcpy
StrInsert(&S,pos,T)	strcpy 0<=pos<=Strlength(S)
StrDelete(&S,pos,len)	strcpy 0<=pos<=StrLength(S)-len
DestroyString(&S)	

} ADT String

# 串 的表示和实现

## 顺序存储两种方法

- 下标为0的数组单元存放长度 (pascal)

```
typedef char SString[MAXSTLEN+1];
```

```
SString s;
```

- 在串值后面加 ‘\0’结束 (C语言,null-terminated)

```
char str[10]="abc";    //s[3]='\0';
```

```
s[0]=3;s[1]='a';s[2]='b';s[3]='c';
```

则s和str存储内容一样,都占有4字节空间

# 字符串编程实例

使用null-terminated串存储，编程实现strlen(char \*)

```
int strlen(char *s){  
    int i;  
    for(i=0;s[i]!=0;i++);  
    return i;  
}
```

# 数组

二维数组:其数据元素是一维数组的线形表

N维数组:其数据元素是N-1维数组的线形表

数组元素两种存储映射方式

- 行主序存储(C语言方式)
- 列主序存储(Fortran方式)

数组中元素在内存映象中的关系 (行主序)

- 二维数组A[m][n]

$$\text{Loc}(i,j)=\text{Loc}(0,0)+(i*n+j)*L \quad //L\text{为每个元素占有空间}$$

- 三维数组B[p][m][n]

$$\text{LOC}(i,j,k)=\text{LOC}(0,0,0)+(i*m*n+j*n+k)*L$$

# 数组的压缩（矩阵）

特殊形状矩阵的存储表示

对称矩阵：  $A[n][n]$  存储到  $B[n(n+1)/2]$

$$A[i,j] \rightarrow B[k] \quad \begin{cases} k=(i+1)i/2+j & i \geq j \\ k=(j+1)j/2+i & i < j \end{cases}$$

三角矩阵：  $A[n][n]$  存储到  $B[n(n+1)/2]$

$$A[i,j] \rightarrow B[k] \quad \begin{cases} k=(i+1)i/2+j & i \geq j \\ 0 & i < j \end{cases}$$

带状矩阵  $A[n][n]$  存储到  $B[3n-2]$

$$A[i,j] \rightarrow B[k] \quad \begin{cases} k=3i-1+(j-i+2)-1=2i+j & |i-j| \leq 1 \\ 0 & \text{else} \end{cases}$$

## 随机稀疏矩阵(非零元)

- 非零元比零元少的多且分布无规律的矩阵。

## 三元组顺序表

```
const MAXSIZE=1000
```

```
typedef struct{
```

```
    int            i,j;
```

```
    ElementType    e;
```

```
}Triple;
```

```
typedef struct{
```

```
    Triple    data[MAXSIZE+1];
```

```
    int      mu,nu,tu;
```

```
}TSMatrix;
```



# 二叉树

$n$ 个元素的有限集，或为空集，或含有唯一的根元素，其余元素分成两个互不相交的子集，每个子集本身也是一颗二叉树。分别称为根的左子树、右子树，集合为空的二叉树称空树。

- 结点的度：非空子树的个数
- 层次：根的层次为1，层次为 $k$ 的结点其孩子层次为 $k+1$
- 二叉树的深度：二叉树中叶子结点的最大层次数
- 满二叉树：所有结点度为2，叶子结点在同一层次
- 完全二叉树：深度 $h$ ， $h-1$ 为满二叉树， $h$ 层的结点都集中在左侧

# 二叉树的性质

性质1 在二叉树的第 $i$ 层的结点个数最多  $2^{(i-1)}$

性质2 深度为 $k$  的二叉树的最大结点数为 $2^k-1$

性质3 任一二叉树 $T$ ，如果其叶子结点数为 $n_0$ ，度为2的结点数为 $n_2$ ，则 $n_0=n_2+1$

$$n_0+n_1+n_2=n=2n_2+n_1+1$$

性质4 具有 $n$ 个结点的完全二叉树深度为  
 $\lfloor \log n \rfloor + 1$  或  $\lceil \log(n+1) \rceil$

性质5 如果对 $n$ 个结点的完全二叉树 $T$ 结点编号，则有：

- 双亲结点编号为 $\lfloor i/2 \rfloor$
- 左孩子编号为 $2i$
- 右孩子编号为 $2i+1$

例：

若一棵度为3的树，有8个度为3的结点，有10个度为2的结点，5个度为1的结点，则度为0的结点数是多少？

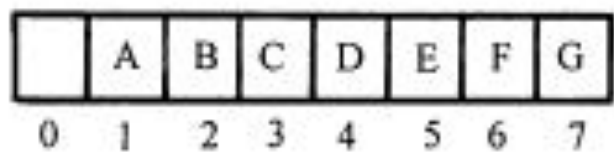
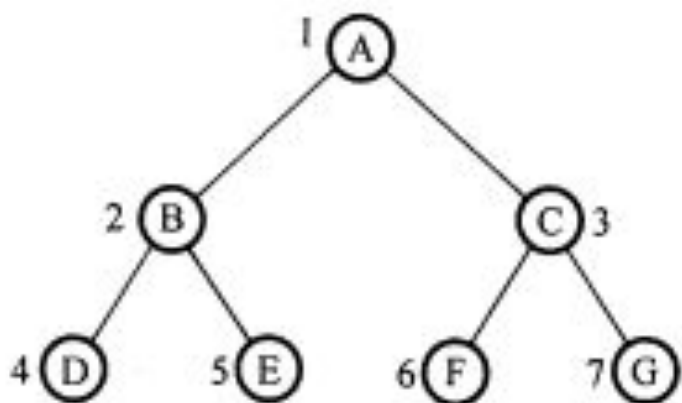
$$8*3+10*2+5*1+1-(8+10+5)=27$$

# 二叉树的存储

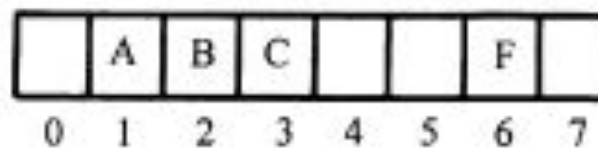
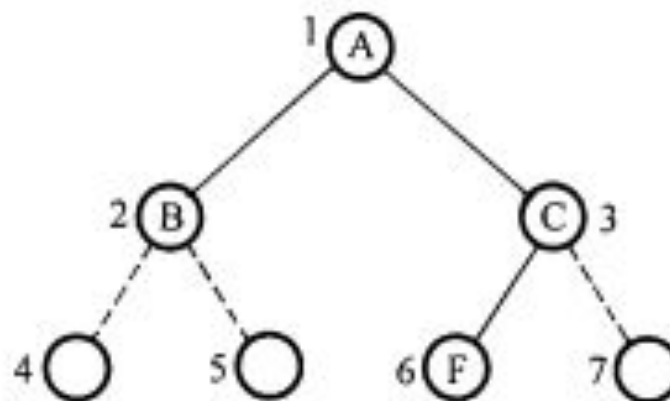
## 顺序存储结构

```
typedef struct{
    ElemType
    int
}SqBiTree
```

\*data;  
nodenum;



(a) 完全二叉树



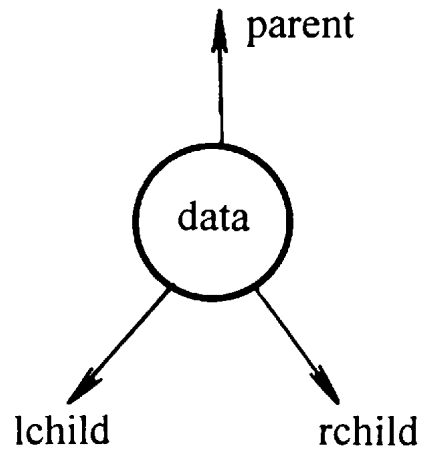
(b) 一般二叉树

## 链式存储结构

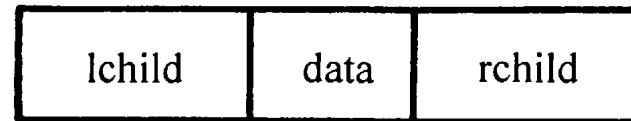
- 二叉链表 包涵data, lchild, rchild三个域, 找双亲必须从根开始
- 三叉链表 包涵data, lchild, rchild, parent四个域, 找双亲容易

```
typedef BiTNode{  
    ElemType          data;  
    struct BiTNode    *lchild,*rchild[,*parent];  
}BiTNode,*BiTree;
```

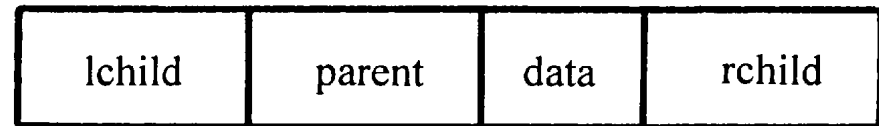
# 存储结点示意图



(a) 结点的逻辑结构



(b) 含两个指针域的结点结构



(c) 含三个指针域的结点结构

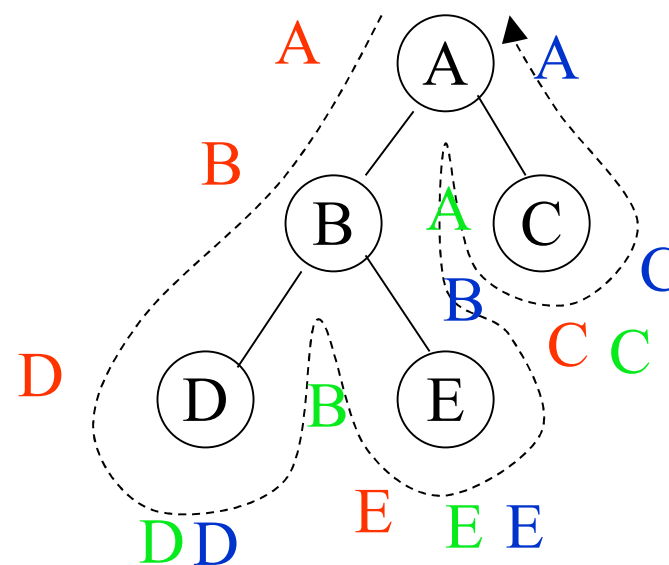
# 二叉树遍历

对所有结点进行访问，且仅被访问一次

LDR、LRD、DLR、DRL、RLD、RDL六种次序

先序（根）DLR、中序（根）LDR、后序（根）LRD

- 右图得三种遍历序列
  - 先序遍历: **ABDEC**
  - 中序遍历: **DBEAC**
  - 后序遍历: **DEBCA**



图中哪两个节点在三种遍历中先后次序不变? **BC、DE、EC、DC**

例:

- 已知一二叉树的先序扩展序列为 **AB##CD##E##**，画出该二叉树。
- 已知一二叉树的后续扩展序列为 **##D#B##CA**，画出该二叉树。
- **已**知一二叉树的先序遍历序列为**ABDECF**，中序遍历序列为**DBEAFC**，画出该二叉树。



# 二叉树遍历模板代码

```
void preorder(Bitree T)
{
    if(!T)return;  //空树，退出递归
    visit(T->data); //访问当前结点数据域
    preorder(T->lchild);
    preorder(T->rchild);
}
```

— visit()函数放置位置决定先、中、后序。

# 二叉树遍历应用

建立二叉树的存储结构(先序扩展二叉树序列)

```
void CreateBiTree(BiTree &T) //递归 先序
```

```
{
```

```
    cin>>ch;
```

```
    if (ch=='#'){T=NULL;return;} //退出递归
```

```
    T=new BiTNode;
```

```
    T->data=ch;
```

```
    CreateBiTree(T->lchild);
```

```
    CreateBiTree(T->rchild);
```

```
}
```

}

**visit ( )**

## 求二叉树的结点数和叶子结点数（先序）

```
void Count(BiTree T,int &C1, int &C2)
{
    if(!T)return;
    C1++;
    if(T->lchild==NULL && T->rchild==NULL)C2++;
    count(T->lchild,C1,C2);
    count(T->rchild,C1,C2);
}
```

## 求二叉树的深度 递归算法

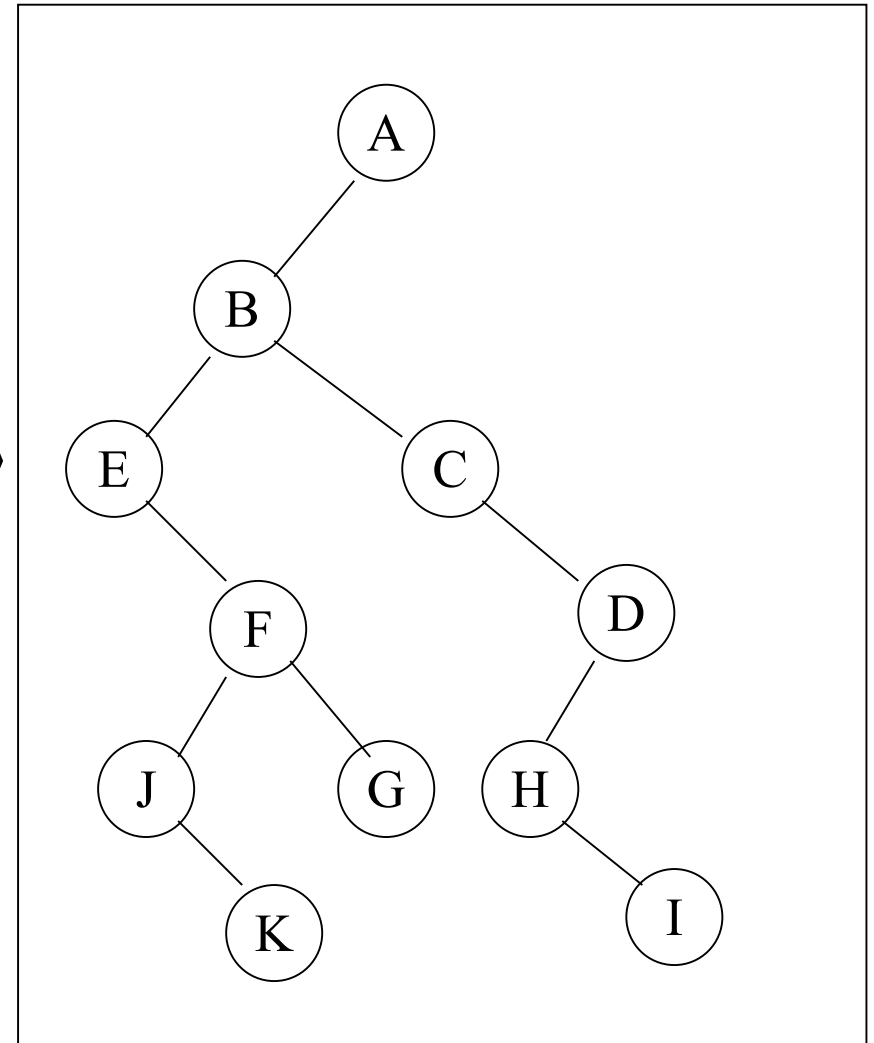
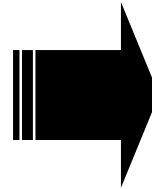
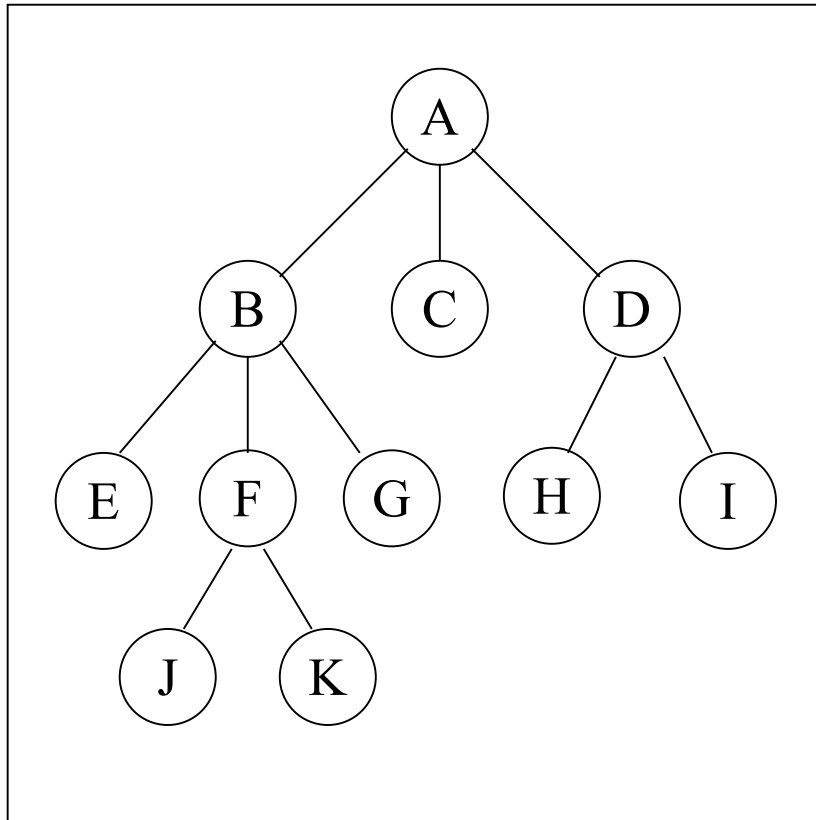
```
int Depth1(BiTree T) {    //后序
    if(!T) return 0;
    hl=Depth1(T->lchild);
    hr=Depth1(T->rchild);
    return hl>hr?hl+1:hr+1;
}
```

```
void Depth2(BiTree T,int h,int &depth){    //先序
    if(!T) return;
    if(h>depth)depth=h;
    Depth2(T->lchild,h+1,depth);
    Depth2(T->rchild,h+1,depth);
}
```

# 树的二叉链表表示法

## 孩子—兄弟表示法

```
typedef struct CSNode{  
    Elem data;  
    struct CSNode    *firstchild, *nextsibling;  
}CSNode,*CSTree;
```



# 树与二叉树的变换

以树的结点为二叉树结点

树根与最左子树的父—子关系改为父—左子关系，去除根与其他子树的父—子关系

将其他各子树根（除最右子树）到其右兄弟之间的隐含关系以父—右子关系表示。

对于无右子树的二叉树可以实施逆变换

# 森林与二叉树的变换

森林到二叉树的变换： $F(T_1, T_2, \dots, T_n) \rightarrow B$

- 若森林**F**空，则二叉树**B**空。
- 由森林中的第一颗树的根结点**ROOT** ( $T_1$ ) 作为二叉树的根，其子树转换为二叉树的左子树。
- 由森林中其余树构成的森林 $\{T_2, T_3, \dots, T_n\}$ 转化得到的二叉树**B**的右子树

二叉树到森林的变换： $B \rightarrow F(T_1, T_2, \dots, T_n)$

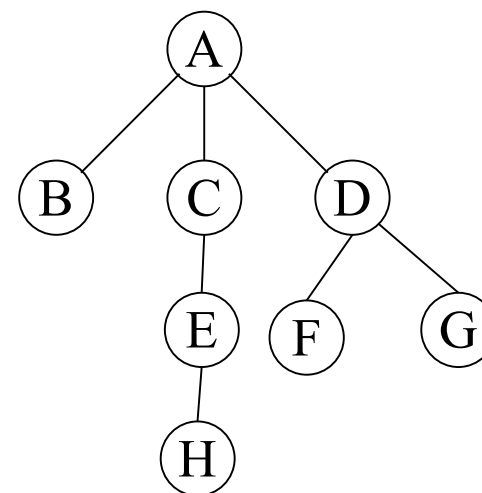
- 若二叉树**B**空，则森林**F**空。
- 二叉树的根为森林中的第一颗树的根结点**ROOT** ( $T_1$ )，二叉树的左子树转化为**T1**的子树
- 二叉树的右子树转化为森林 $\{T_2, T_3, \dots, T_n\}$



# 树和森林的遍历

## 树遍历的三种搜索路径（右图）

- 先根次序遍历 **ABCEHDFG**
- 后根次序遍历 **BHECFGDA**
- 按层次序遍历 **ABCDEFGH**



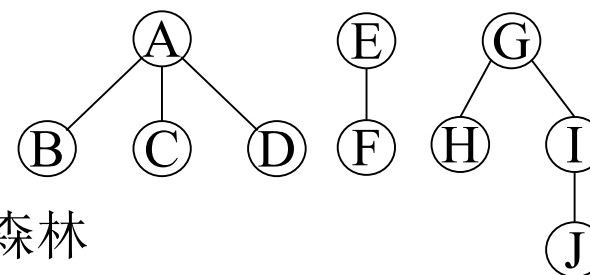
## 对森林遍历的两种方法（右图）

- 先序遍历: **(ABCDEFGHJIJ)**

访问森林中的第一颗树根结点

先序遍历第一颗树中根结点的子树森林

先序遍历除去第一颗树之后剩余树构成的森林



- 中序遍历: **(BCDAFEHJIG)**

中序遍历第一颗树中根结点的子树森林

访问森林中的第一颗树根结点

中序遍历除去第一颗树之后剩余树构成的森林

# 结论

树的先根遍历和后根遍历可利用二叉树的先序和中序遍历算法实现

森林的先序遍历和中序遍历即是对应二叉树的先序和中序遍历

## 例求森林的深度

```
int TreeDepth(CsTree T)  后序
{
    if(!T)return 0;
    h1=TreeDepth(T->firstchild);
    h2=TreeDepth(T->nextsibling);
    return (h1+1>h2)?h1+1:h2;
}
```

# 二叉排序树

## 二叉排序树定义：

- 或空树，或满足如下特征：
- 若左子树不空，则左子树上的所有结点值均小于根结点值
- 若右子树不空，则右子树上的所有结点值均大于或等于根结点值
- 左右子树分别也是二叉排序树

判断一棵树是否二叉排序树 ,遍历算法？

# 二叉排序树插入结点

```
void Insert_BST(BiTree &T, TElemType e) {  
    s=new BiTNode;  
    s->data=e;s->lchild=s->rchild=NULL;  
    if(!T)T=s;  //第一个结点  
    else{  
        p=T;  
        while(p)  
            if(e.key<p->data.key){f=p;p=p->lchild;}  
            else {f=p;p=p->rchild;}  
        if(e.key<f->data.key)f->lchild=s;  
        else f->rchild=s;  
    }//else  
}
```

# 例子：二叉排序树构建

将下列元素依次插入一个空二叉排序树。

{33,10,25, 19,06,49,37,76,60}

- 1.画出最终的二叉排序树；
- 2.画出删除结点10以后的二叉树；(在查找一章讲)
- 3.若重新调整输入次序，使得形成的二叉树是完全二叉树，请画出。

# 霍夫曼(huffman)树及其应用

- 最优二叉树（霍夫曼树）：
  - $n$ 个叶子构成的所有二叉树中，其中WPL最小的二叉树称霍夫曼树。
  - 树的带权路径长度： $WPL = \sum \omega_k l_k$  (叶子结点)

## 霍夫曼树算法

- 1)根据给定的 $n$ 个权值 $\{w_1, w_2, \dots, w_n\}$ ，构成 $n$ 颗二叉树的集合 $F = \{T_1, T_2, \dots, T_n\}$ ，每个二叉树只有一个带权 $W_i$ 的根结点。
- 2)在 $F$ 中选取两颗根结点的权值最小的树作为左右子树，构造一个新二叉树，且置其根结点的权值为两个子树根结点权值之和。
- 3)在 $F$ 中删除这两棵树，同时在 $F$ 中加入新树。
- 4)重复2)、3) 直至 $F$ 只含一颗树为止



## 前缀编码:

- 任一字符的编码都不是另一字符编码的前缀

## 霍夫曼编码:

- 以n种字符出现的概率（频率）为权，设计一个赫夫曼树，由此得到的二进制前缀编码称为霍夫曼编码

## 例:

- 有8个字符a、b、c、d、e、f、g、h，出现频率为5%、29%、7%、8%、14%、23%、3%、11%，设计霍夫曼编码。

# 图的基本概念

图：一个顶点（**vertex**）的有穷集 $V(G)$ 和一个弧（**arc**）的集合 $E(G)$ 组成。记做： $G=(V, E)$ 。 $V$ 是数据元素集， $E$ 是关系集。

弧和边： $\langle u,v \rangle$  和  $(u,v)$

有向图(**digraph**)、无向图(**undigraph**)

有向网、无向网

完全图、有向完全图

稀疏图、稠密图

子图、连通图、连通分量

生成树、生成森林

度、出度、入度

## ADT Graph{

数据对象：**V**是具有相同特性数据元素的集合。

数据关系： $R=\{<v,w>|v,w\in V\text{且}P(v,w)\}$ ，其中 $<v,w>$ 表示从 $v$ 到 $w$ 的弧，谓词 $P(v,w)$ 表示弧 $<v,w>$ 的信息}

基本操作：

... ..

FirstAdjVex( $G, v$ )

NextAdjVex( $G, v, w$ )

DFSTraverse( $G, v, \text{visit}()$ )

BFSTraverse( $G, v, \text{visit}()$ )

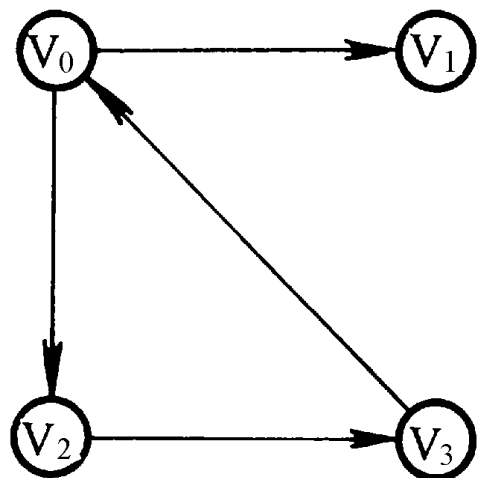
... ..

} ADT Graph

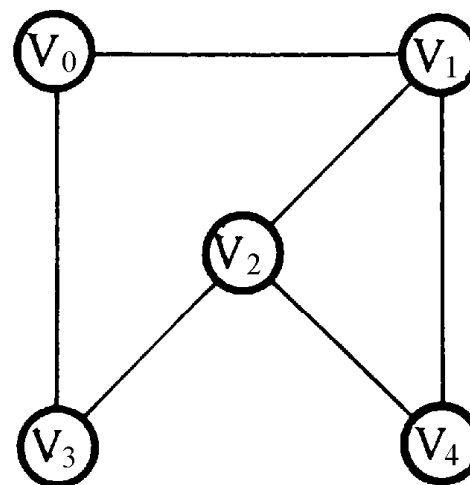
# 图的存储

图的数组（邻接矩阵）存储表示

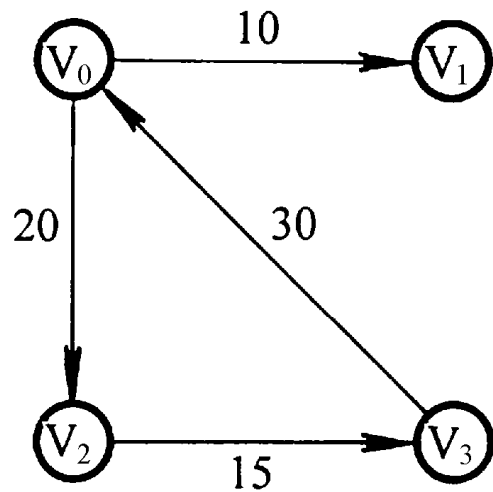
```
typedef enum{DG,DN,AG,AN} GraphKind;
typedef struct ArcCell{
    VRType      adj;
    InfoType     *info;
}ArcCell,AdjMatrix[MAX_V_NUM][ MAX_V_NUM];
typedef struct{
    VertexType    vexs[MAX_V_NUM];
    AdjMatrix     arcs;
    int           vexnum,arcnum;
    GraphKind     kind;
}MGraph;
```



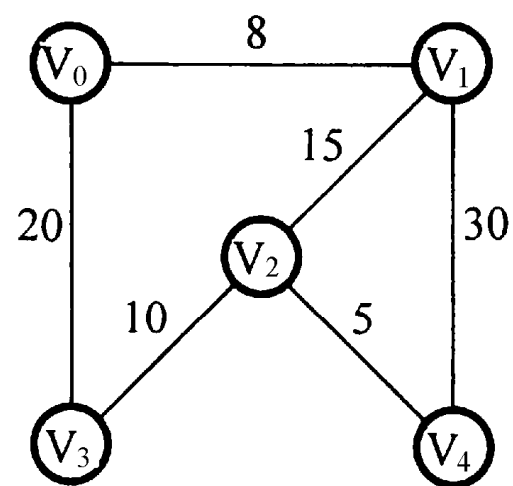
(a) 有向图  $G_1$



(b) 无向图  $G_2$



(a) 有向网  $G_3$



(b) 无向网  $G_4$

邻接矩阵行列  
1数量的含义

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

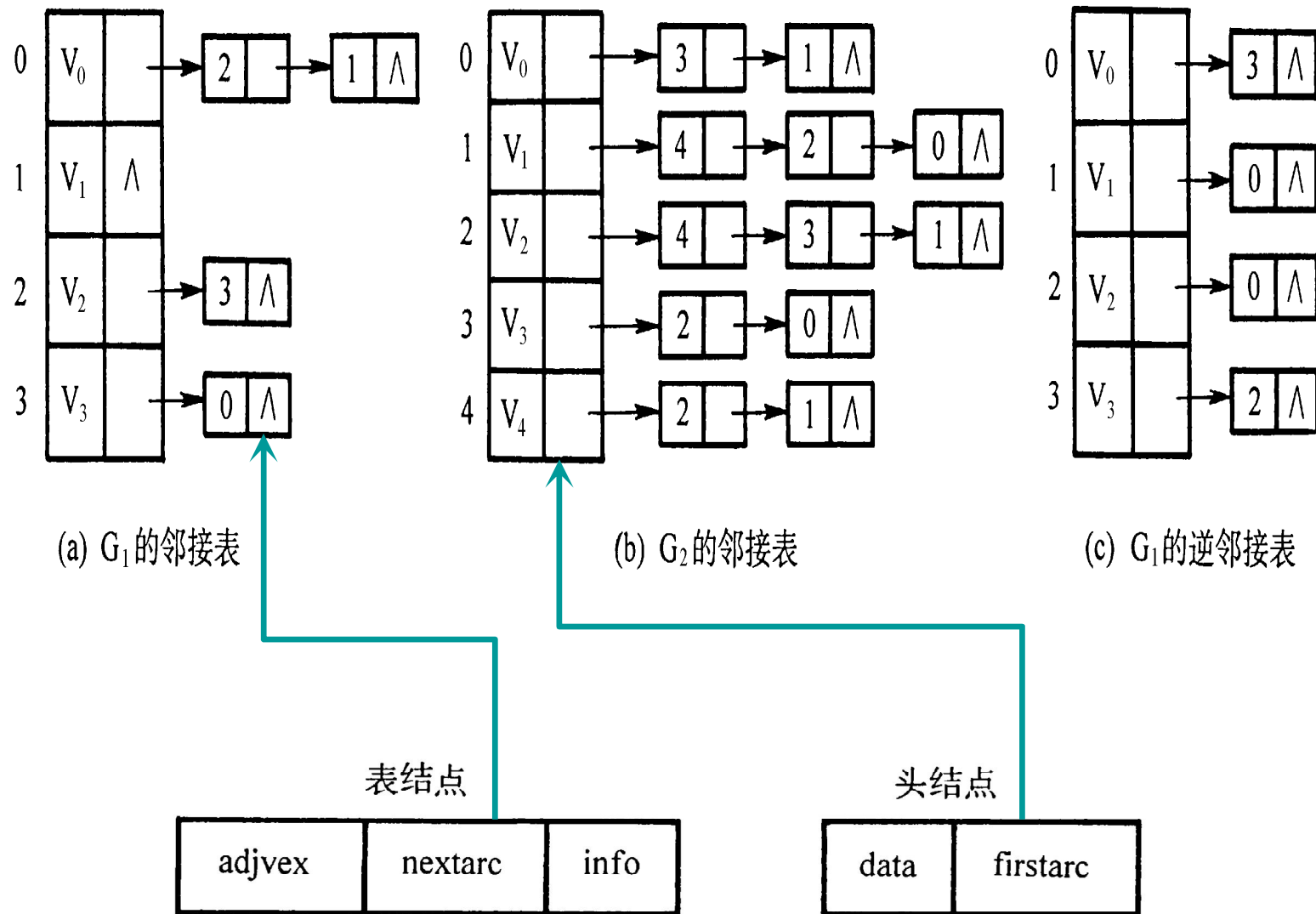
(a)  $G_1$  的邻接矩阵

$$A_2 = \begin{pmatrix} \infty & 8 & \infty & 20 & \infty \\ 8 & \infty & 15 & \infty & 30 \\ \infty & 15 & \infty & 10 & 5 \\ 20 & \infty & 10 & \infty & \infty \\ \infty & 30 & 5 & \infty & \infty \end{pmatrix}$$

(b)  $G_4$  的邻接矩阵

# 图的邻接表存储表示

```
typedef struct ArcNode{
    int                adjvex;
    struct ArcNode*nextarc;
    InfoType          *info;
} ArcNode;
typedef struct VNode{
    VertetType      data;
    ArcNode          *firsrarc;
}VNode,AdjList[MAX_VERTEX_NUM];
typedef struct{
    AdList          vertices;
    int             vexnum,arcnum;
    int             kind;
}ALGraph;
```

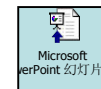


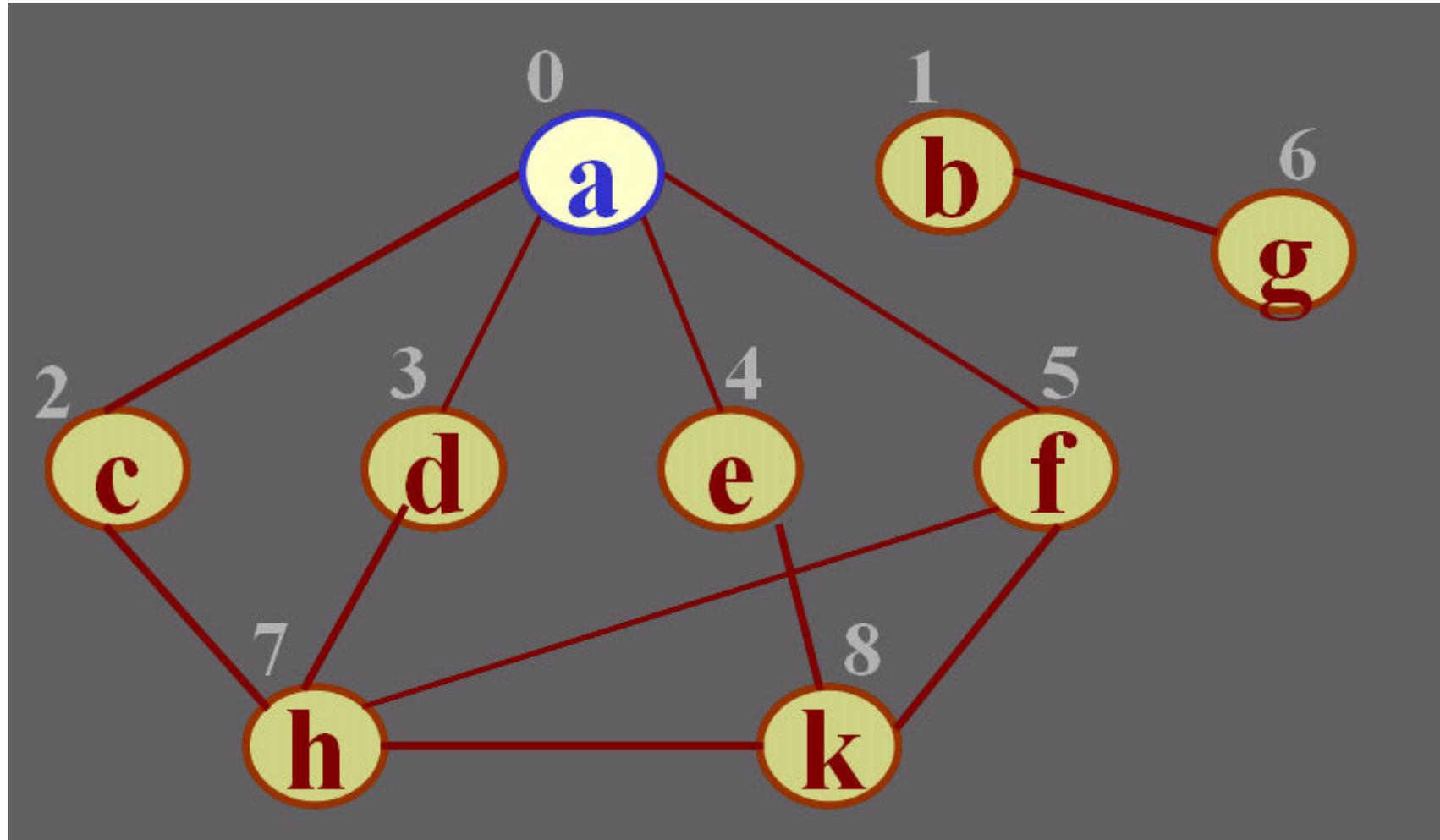


# 图的遍历

## 深度优先搜索(Depth First Search):

- 从某个顶点 $v$ 出发，首先访问该顶点，然后依次从它的各个未被访问的邻接点出发，深度优先遍历图。直至图中所有和 $v$ 有路径相通的点都被访问到；
- 若仍有其他顶点未被访问，则另选一个未被访问的顶点作为起始点，重复上述过程，直至图中所有顶点都被访问到。
- 深度优先搜索生成树





a->c->h->d->f->k->e->b->g

## 广度优先搜索(Breadth First Search):

- 从某个顶点 $v$ 出发，首先访问该顶点，然后依次访问 $v$ 的各个未曾访问的邻接点，然后分别从这些邻接点出发依次访问它们的邻接点。并使得“先被访问的顶点的邻接点”先于“后被访问的顶点的邻接点”被访问，直至图中所有已被访问的顶点的邻接点都被访问到；
- 若仍有其他顶点未被访问，则另选一个未被访问的顶点作为起始点，重复上述过程，直至图中所有结点都被访问到
- 广度优先搜索生成树

如何获得关键路径：设源点V1，汇点Vn

– **ve** (j) :事件Vj可能发生的最早时刻。即从V1到Vj最长带权路径。

$$\text{ve}(j) = 0 \quad (j=1)$$

$$= \max\{\text{ve}(i) + w(V_i \rightarrow V_j)\} \quad (j=2, 3, \dots, n)$$

– **vl** (i) :在不延误整个工期的前提下，事件Vi发生所允许的最晚时刻。等于**ve**(n)减去从Vi到Vn的最长带权路径长度。

$$\text{vl}(i) = \text{ve}(n) \quad (i=n)$$

$$= \min\{\text{vl}(j) - w(V_i \rightarrow V_j)\} \quad (k=1, 2, 3, \dots, m)$$

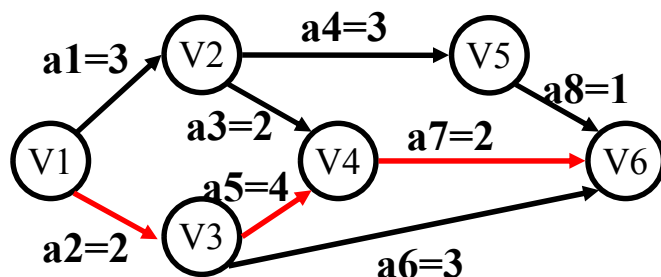
– **ee** (k) :活动ak (Vi->Vj) 可能开始的最早时刻。ee(k)=ve(i)

– **el** (k) :在不延误整个工期的前提下，活动ak (Vi->Vj) 开始所允许的最晚时刻。

$$\text{el}(k) = \text{vl}(j) - w(V_i \rightarrow V_j) \quad (k=1, 2, 3, \dots, m)$$

若某弧 $a_k$ 的**el** (k) 和**ee** (k) 相等，则 $a_k$ 为关键活动；否则**el** (k) - **ee** (k) 为活动的余量。

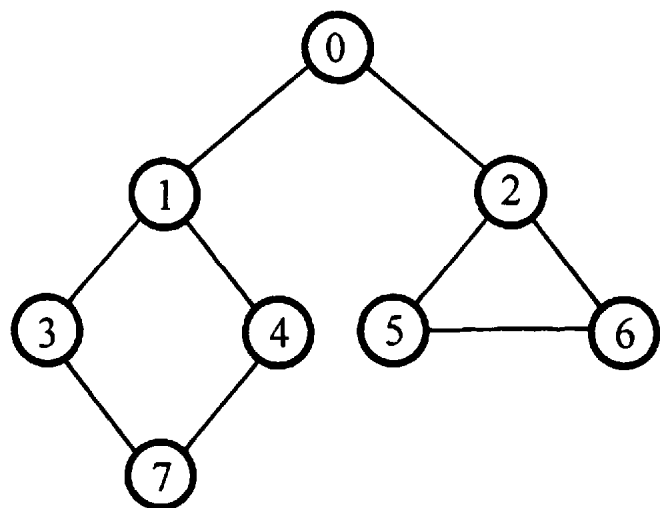
# AOE网络关键路径



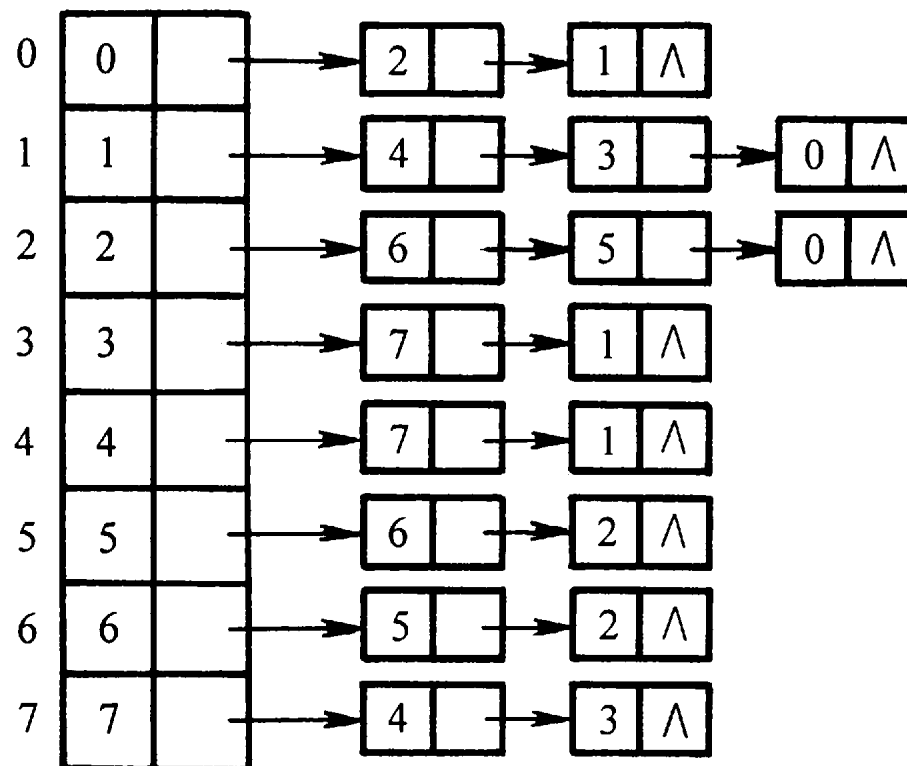
顶点	Ve	VI
V1	0	0
V2	3	4
V3	2	2
V4	6	6
V5	6	7
V6	8	8

活动	ee	el	el-ee
a1	0	1	1
a2	0	0	0
a3	3	4	1
a4	3	4	1
a5	2	2	0
a6	2	5	3
a7	6	6	0
a8	6	7	1

# 遍历序列和物理存储次序有关



(a) 无向图  $G_5$



(b)  $G_5$  的邻接表

**DFS序列: 0, 2, 6, 5, 1, 4, 7, 3**

**BFS序列: 0, 2, 1, 6, 5, 4, 3, 7**

# 图的连通性问题

极小连通子图：

- $n$ 个结点的连通图中，包涵 $n$ 个结点和 $n-1$ 个边构成的连通子图

连通图的生成树：即极小连通子图

连通网最小生成树：权值和最小的生成树

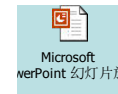
求最小生成树的算法

- 克鲁斯卡尔（Kruskal）算法
- 普里姆（Prim）算法
- 算法比较：Prim算法针对结点；Kruskal算法针对边；稠密图用前者，稀疏图用后者。

# 克鲁斯卡尔算法(Kruskal)

## 算法思想

- 1)构造只含 $n$ 个结点的森林
- 2)按权值从小到大选择边加入到森林中，  
并使森林不产生回路
- 3)重复2直到森林变成一颗树

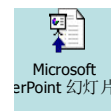




# 普里姆算法 (Prim)

## 算法思想

1. 将所有结点分为两类集合：一类是已经落在生成树上的结点集合，另一类是尚未落在生成树上的结点集合；
2. 在图中任选一个结点 $v$ 构成生成树的树根，形成生成树结点集合；
3. 在连接两类结点的边中选出权值最小的边，将该边所连接的尚未落在生成树上的结点加入到生成树上。同时保留该边作为生成树的树枝；
4. 重复3直至所有结点都加入生成树.



# 拓扑排序

活动顶点网络（AOV, activity on vertex）

- 以顶点表示活动，以弧表示活动之间的优先制约关系的有向图

死锁：

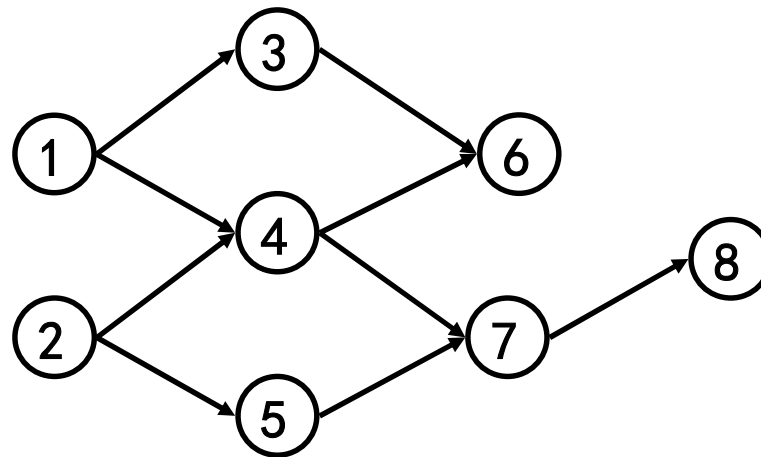
- AOV中不允许出现回路，回路意味某活动以自己的结束作为开始的先决条件。称为死锁。

拓扑排序/拓扑有序序列

- 若在有向图中从u到v有一条弧，则在序列中u排在v之前，称有向图的这个操作为拓扑排序。所得序列为拓扑有序序列。若有死锁，则无法获得拓扑有序序列。

算法思想：

- 1) 选取一个没有前驱的顶点，输出它，并从AOV中网中删除此顶点以及所有以它为尾的弧。
- 2) 重复1) 直至输出所有结点



```

void topologic ISort(AlGr ph G)
{
    int indgr[MAX_VERTEX_NUM]={0};
    InitSt ck(S);
    for(i=0;i<G.vernum;i++)
        for(p=G.vertices[i].first rc;p;p=p->next rc)
            indgr[p-> djvex]++;
    for(i=0;i<G.vernum;i++)if(!indgr[i])Push(S,i);
    while(!St ckEmpty(S)){
        Pop(S,j);cout<<G.vertices[j].d t ;
        for(p=G.vertices[j].first rc;p;p=p->next rc)
            if(--indgr[p-> djvex]==0)Push(S,p-> djvex);
    }
}

```

前页图的输出： 2 5 1 4 7 8 3 6

# 关键路径

事件：

- 关于活动开始或完成的断言或陈述。

活动边网络（**AOE, activity on edge**）：

- 以弧表示活动，以顶点表示事件的有向图。弧有权值，表示活动所需的时间。

源点、汇点：

- 整个工程的起始点为源点，整个工程的结束点为汇点。  
一个工程的**AOE**是一个单源、单汇点的无环图。

带权路径长度：

- 一条路径上所有弧权值之和。

关键路径、关键活动：

- 一个工程的最短完成时间是从源点到汇点的最长带权路径，称该路径为关键路径；该路径上的活动为关键活动

如何获得关键路径：设源点V1，汇点Vn

– **ve** (j) :事件Vj可能发生的最早时刻。即从V1到Vj最长带权路径。

$$\text{ve}(j) = 0 \quad (j=1)$$

$$= \max\{\text{ve}(i) + w(V_i \rightarrow V_j)\} \quad (j=2, 3, \dots, n)$$

– **vl** (i) :在不延误整个工期的前提下，事件Vi发生所允许的最晚时刻。等于**ve**(n)减去从Vi到Vn的最长带权路径长度。

$$\text{vl}(i) = \text{ve}(n) \quad (i=n)$$

$$= \min\{\text{vl}(j) - w(V_i \rightarrow V_j)\} \quad (k=1, 2, 3, \dots, m)$$

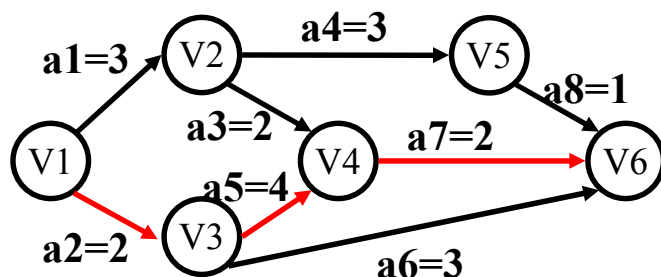
– **ee** (k) :活动ak (Vi->Vj) 可能开始的最早时刻。ee(k)=ve(i)

– **el** (k) :在不延误整个工期的前提下，活动ak (Vi->Vj) 开始所允许的最晚时刻。

$$\text{el}(k) = \text{vl}(j) - w(V_i \rightarrow V_j) \quad (k=1, 2, 3, \dots, m)$$

若某弧 $a_k$ 的**el** (k) 和**ee** (k) 相等，则 $a_k$ 为关键活动；否则**el** (k) - **ee** (k) 为活动的余量。

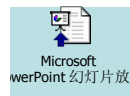
# AOE网络关键路径



顶点	Ve	VI
V1	0	0
V2	3	4
V3	2	2
V4	6	6
V5	6	7
V6	8	8

活动	ee	el	el-ee
a1	0	1	1
a2	0	0	0
a3	3	4	1
a4	3	4	1
a5	2	2	0
a6	2	5	3
a7	6	6	0
a8	6	7	1

# 最短路径



## 迪杰斯特拉(Dijkstra)算法

- 1) 设  $\text{arcs}[n,n]$  为有向网的邻接矩阵， $S$  为已找到最短路径的终点的集合，其初值只有一个顶点，即源点。 $D[n]$  的每个分量表示当前所找到的从源点出发（经过集合  $S$  中的顶点）到各个终点的最短路径长度。其初值为  
$$D[k] = \text{arcs}[i,k] \quad (i \text{ 为源点}, k=0, 1, \dots, n)$$
- 2) 选择  $v_j$ ，使得  $D[j] = \min\{D[i] | v_i \notin S, v_i \in V\}$ ， $v_j$  为目前找到的从源点出发的最短路径的终点。将  $j$  加入  $S$  集合。
- 3) 修改  $D$  数组中不在  $S$  中的终点对应的分量值。如果  $\text{arcs}[j,k]$  为有限值，即  $j, k$  有弧存在，且  
$$D[j] + \text{arcs}[j,k] < D[k] \quad \text{则令} \quad D[k] = D[j] + \text{arcs}[j,k]$$
- 4) 重复 2)、3) 直至求得源点到所有终点的最短路径



# 查找表

## 查找表：

- 由同一类元素或记录构成的集合。对数据元素间的关系未作限定。

## 对查找表的操作有

- 查找某个“特定”的元素是否在表中。
- 查找某个“特点”的元素的各种属性。
- 在查找表中插入一个元素。
- 在查找表中删除一个元素

## 静态查找表、动态查找表

## 关键字

- 数据元素中的某个数据项值。可以表示一个数据元素，如可以唯一表示，则为主关键字（primary key）。

## 查找

- 根据给定的某个值，在查找表中确定一个关键字等于给定值的数据元素。若找到表示查找成功，返回该元素详细信息或在查找表中的位置；负责返回**NULL**

# 静态查找表

## 顺序查找

- 查找成功和失败
- **平均查找长度**: 查找过程中先后和给定值进行比较的关键字的个数的期望值.

$$ASL = \sum P_i C_i \quad \sum P_i = 1 \quad i = 1, 2, \dots, n$$

$$C_i = n - i + 1 \quad P_i = 1/n$$

$$ASL_{ss} = 1/n \sum (n - i + 1) = (n+1)/2$$

折半查找 (binary Search):二分查找

$$- ASL_{bs} = (n+1)/n \log_2(n+1) - 1$$

索引顺序查找:分块查找

$$- ASL_{idx} = ASL(b) + ASL(n/b) \approx \log_2(b+1) - 1 + (n/b + 1)/2$$

# 动态查找表

## 二叉查找树(二叉排序树)

- 通过合根结点的关键字进行比较可以将继续查找的范围缩小到某一颗子树中，具有该特性的树称查找树。二叉查找树即为二叉排序树。

## 二叉查找树的平均查找长度

$$ASL_{BT} = 2(n+1)/n * \log n + C$$

## 二叉查找树删除结点的处理方法

- 若是叶子结点，直接删除
- 只有一个孩子，则将其孩子直接挂到其双亲上。
- 有两个孩子，找左孩子中最大的一个元素，代替被删除结点，最大元素肯定只有一个孩子，按2) 处理删除最大元素

举例：将下列元素依次插入一个空二叉排序树{33,10,25, 19,06,49,37,76,60}

- 1.画出最终的二叉排序树；
- 2.画出删除结点10以后的二叉树

# 哈希表

## 哈希表

- 在记录的关键字和其存储位置（数组下标）之间设定一个确定的对应关系 $f$ ，关键字为 $kval$ 的记录存储在 $f(kval)$ 位置处

## 哈希表的装填系数 $\alpha = n/m$

- 其中 $m$ 为哈希表分配空间
- $n$ 为填入的记录数。 实际应用：0.65~0.85

## 哈希函数

- 可以对关键字做简单的算术或逻辑运算。

$f1(key)=key$

$f2(key)=(key \text{ 第一个字母ASCII码}) - ('A' \text{ 的 ASCII码})$

$f3(key)=(f2(key \text{ 第一个字母}) + f2(key \text{ 最后一个字母}))/2$

## 冲突

- 若出现 $f(key1) = f(key2)$ 的现象称冲突。没有冲突的哈希函数很少存在，只能尽量均匀。称“再散列”。在建哈希表时，不仅要设定哈希函数，还要设定处理冲突的方法

## 哈希表的严格定义：

- 根据设定的哈希函数和处理冲突的方法为一组记录建立的一种存储结构，哈希函数又称散列函数。构建哈希表又称散列技术。

# 处理冲突的方法

## 开放定址法

- 从哈希地址 $\text{Hash}(\text{key})$ 求得一个地址序列 $H_1, H_2, \dots, H_k, 0 \leq k \leq m-1$ , 即 $H_1, H_2, \dots, H_{k-1}$ 都不空, 直到 $H_k$ 为空为止。若哈希表未满, 必能找到 $k < m$ , 使 $H_k$ 空。
$$H_i = (\text{Hash}(\text{key}) + d_i) \bmod m \quad i=1, 2, \dots, k \quad k \leq m-1$$
$$d_i \text{ 为递增序列, 有三种取法:}$$
  - a)  $d_i = 1, 2, \dots, m-1$
  - b)  $d_i = 1^2, -1^2, 2^2, -2^2, \dots, k^2, -k^2 \quad (k \leq m/2)$
  - c)  $d_i = \text{伪随机序列}$
- 例 设一组关键字(07,15,20,31,48,53,64,76, 82,99)试构建哈希表取 $m=11$ ,  $p=11$ ,  $\text{Hash}(\text{key}) = \text{key} \bmod 11$



0	1	2	3	4	5	6	7	8	9	10
53	64	76	99	15	48	82	07		20	31
64	76	99		48	82				31	53
76	99								53	64
99									64	76
3	4	4	4	1	2	2	1		1	2

$$ASL = (3 + 4 + 4 + 4 + 1 + 2 + 2 + 1 + 1 + 2) / 10 = 2.4$$

## 链地址法

- 将所有关键字为同义词（即具有相同的哈希函数值）的记录存贮在同一线性链表中，而哈希表中下标为*i*的分量存储哈希函数值为*i*的链表的头指针

0	1	2	3	4	5	6	7	8	9	10
↓	Λ	Λ	Λ	↓	↓	Λ	↓	Λ	↓	Λ
99				15	82		07		20	76
				↓					↓	
				48					31	
									↓	
									53	
									↓	
									64	

$$ASL = (1 \times 6 + 2 \times 2 + 3 + 4) / 10 = 1.7$$

# 排序

## ☞ 排序

设N个记录  $\{r_1, r_2 \dots r_n\}$ , 相应关键字  $\{k_1, k_2 \dots k_n\}$   
求一种排列  $p_1, p_2 \dots p_n$ , 使得  $k_{p_1} \leq k_{p_2} \leq \dots \leq k_{p_n}$   
即  $\{r_{p_1}, r_{p_2} \dots r_{p_n}\}$  是按关键字有序序列

## ☞ 稳定与不稳定排序

若  $k_i = k_j$ , 并且在待排序列中  $r_i$  领先于  $r_j$  (即  $i < j$ ), 若排序后的序列中  $r_i$  仍然领先  $r_j$ , 则称该排序方法稳定。

## ☞ 内部排序与外部排序

内部排序: 仅在内部存储器中完成的排序

外部排序: 在排序中需要访问外部存储器的排序

☞ 本章排序操作对象:

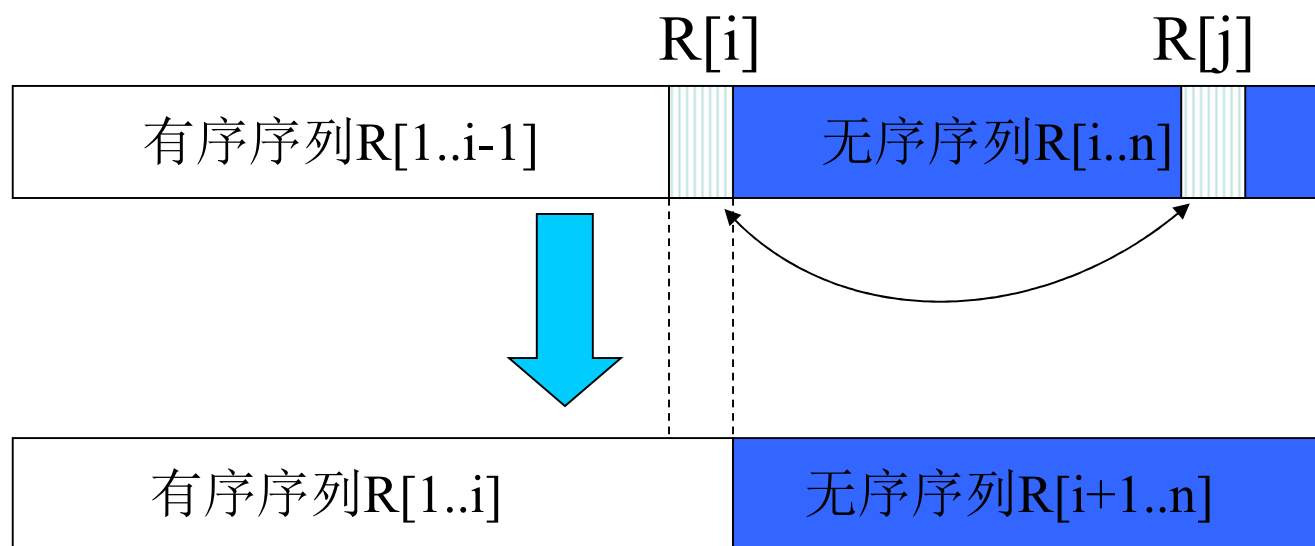
```
typedef struct{
    RcdType r[MAXSIZE+1];
    Int    length;
}Sqlist;
```

# 选择排序

算法复杂度：时间： $O(n^2)$  空间： $O(1)$

是否稳定排序：是

数据有序性影响：无

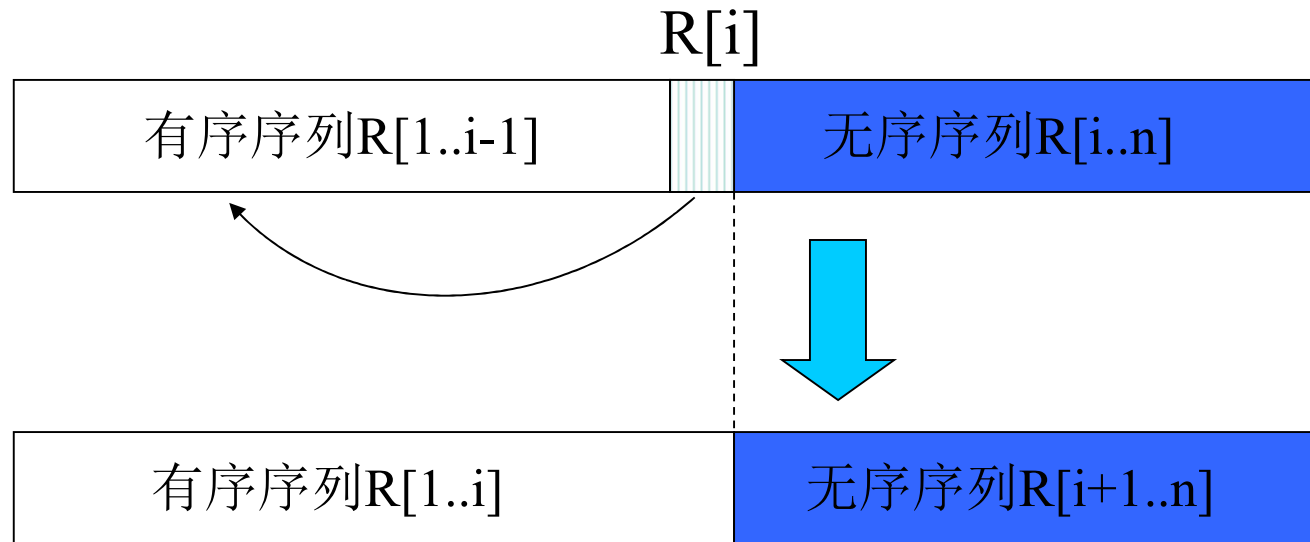


# 插入排序

算法复杂度：时间： $O(n^2)$  空间： $O(1)$

是否稳定排序：是

数据有序性影响：有好的影响可以达 $O(n)$

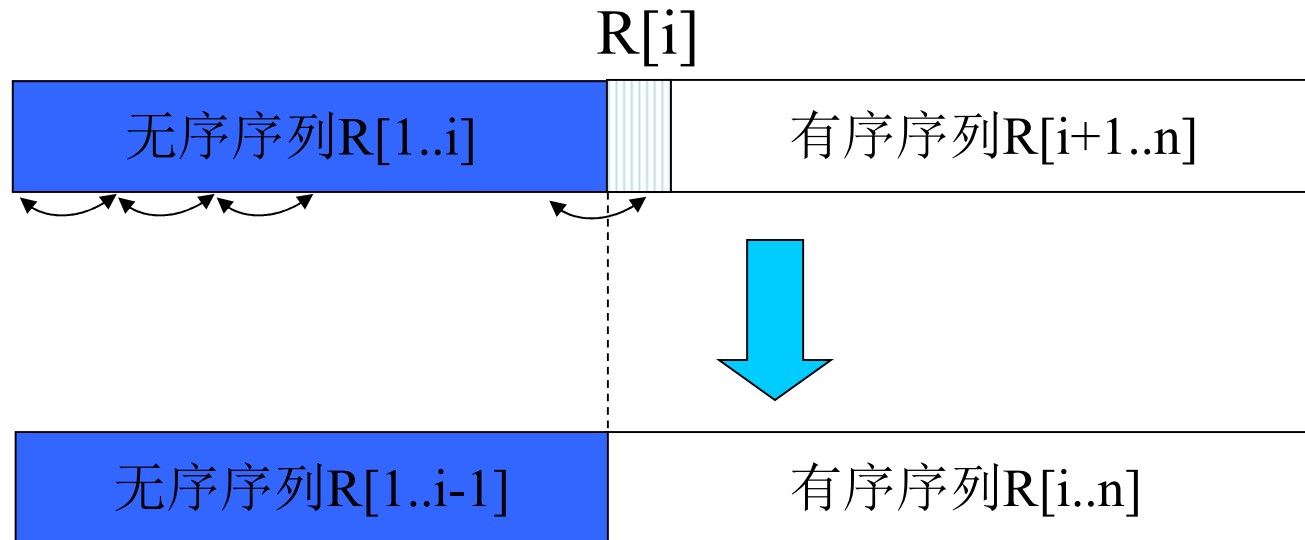


# 起泡排序

算法复杂度：时间： $O(n^2)$  空间： $O(1)$

是否稳定排序：是

数据有序性影响：有好的影响可以达 $O(n)$



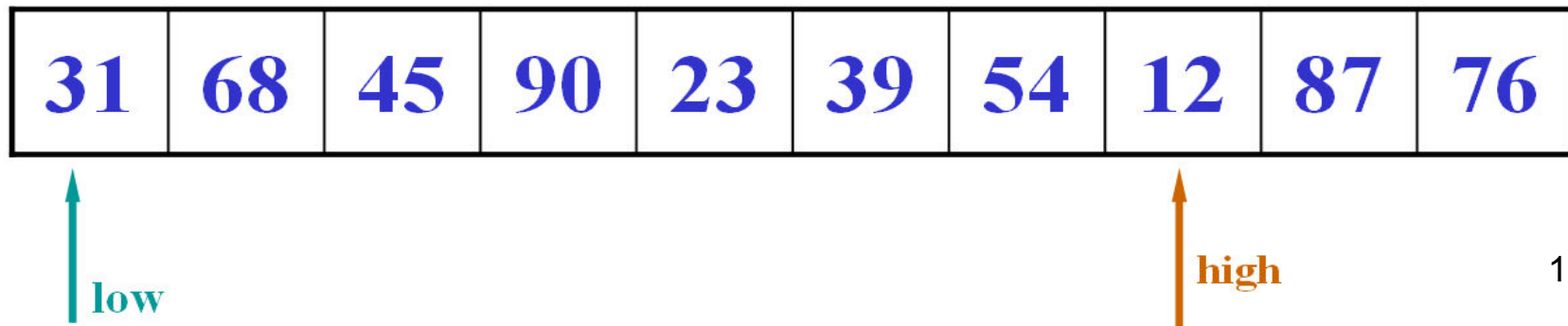
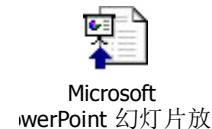
# 快速排序

算法复杂度：时间： $O(n\log n)$  空间： $O(\log n)$

是否稳定排序：否

数据有序性影响：有坏的影响,可达 $O(n^2)$

暂存枢轴记录 R[0] **31**

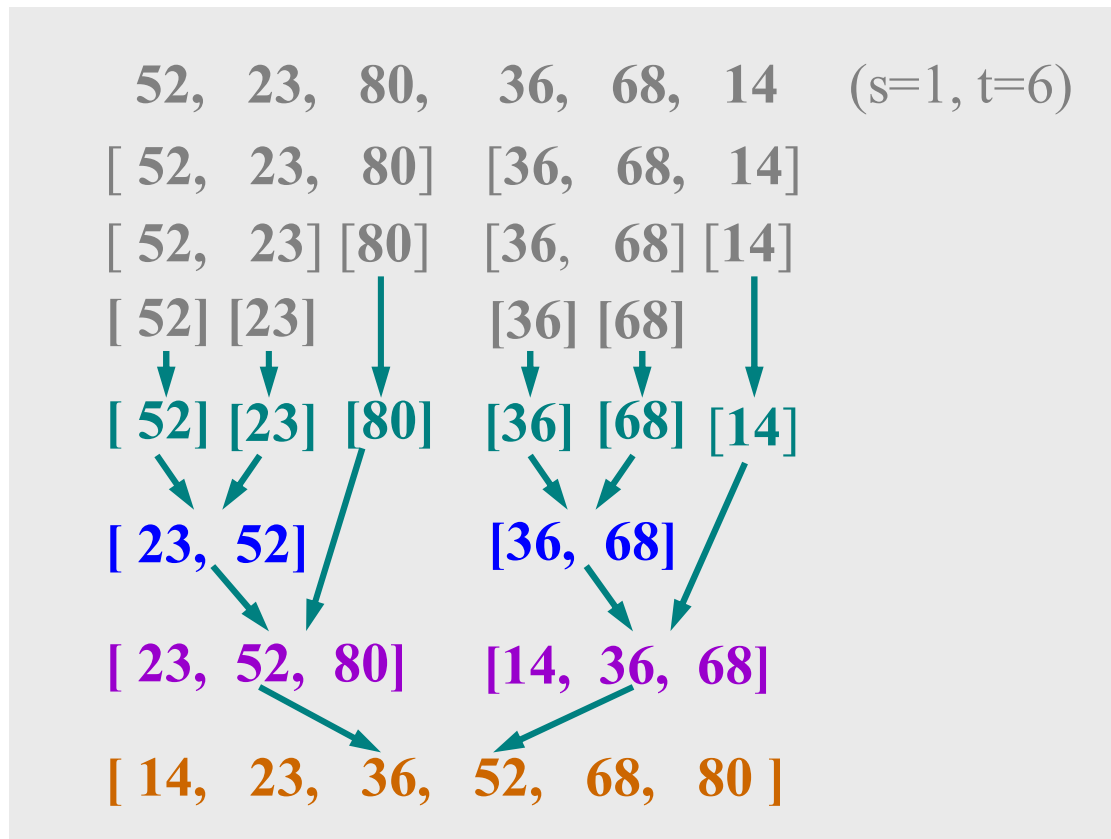


# 归并排序

算法复杂度：时间: $O(n\log n)$  空间: $O(n)$






是否稳定排序：是

数据有序性影响：无





# 两路归并

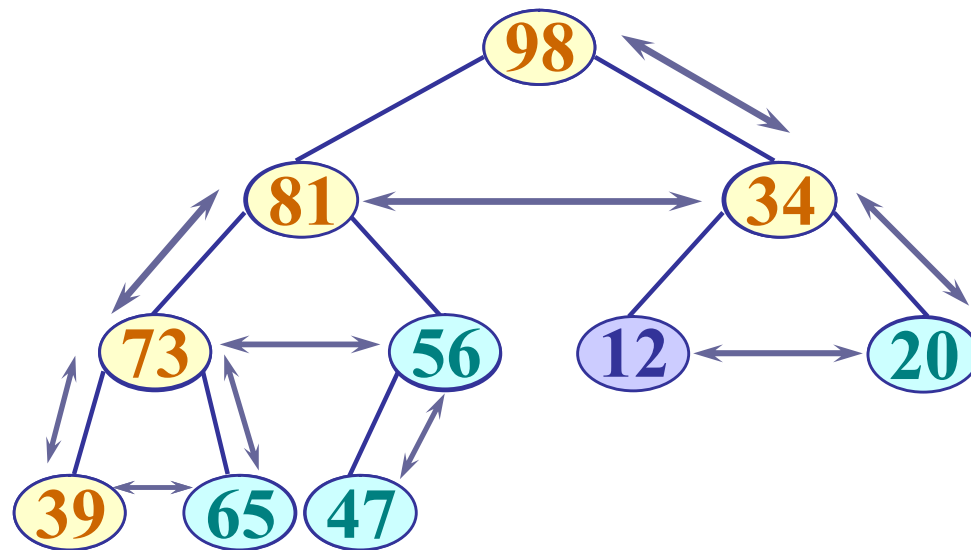
步长	存储区	有序段
1	$r_1$	 91 67 35 62 29 72 46 31 41 25 83 79 13 80 18 74
2	$r_2$	 67 91 35 62 29 72 31 46 25 41 79 83 13 80 18 74
4	$r_1$	 35 62 67 91 29 31 46 72 25 41 79 83 13 18 74 80
8	$r_2$	 29 31 35 46 62 67 72 91 13 18 25 41 74 79 80 83
	$r_1$	 13 18 25 29 31 35 41 46 62 67 72 74 79 80 83 91

# 堆排序

算法复杂度：时间： $O(n\log n)$  空间： $O(1)$

是否稳定排序：否

数据有序性影响：无



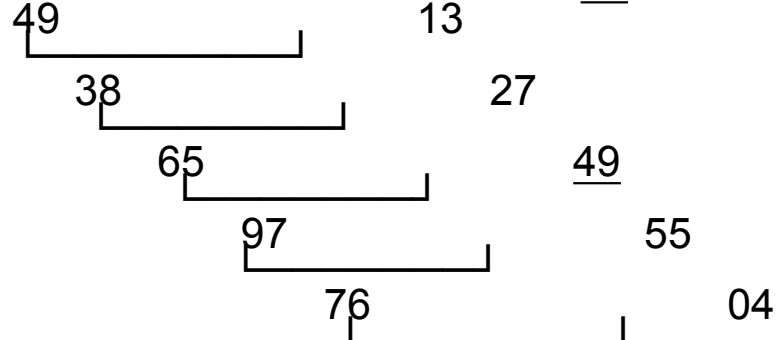
大顶堆：

$$\begin{cases} r_i \geq r_{2i} \\ r_i \geq r_{2i+1} \end{cases}$$

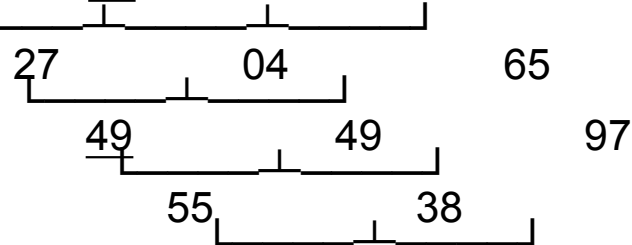
# 希尔排序

又称“缩小增量排序”

初始: 49 38 65 97 76 13 27 49 55 04



一趟排序: 13 27 49 55 04 49 38 65 97 76



二趟排序: 13 04 49 38 27 49 55 65 97 76

三趟排序: 04 13 27 38 49 49 55 65 76 97

时间 $O(n^{3/2})$

空间 $O(1)$

不稳定排序

# 基数排序

关键字分级

有d级关键字，要进行d趟分配和收集

时间复杂度  $O(n \times d)$ ; 空间复杂度  $O(n)$

稳定排序

待排关键字序列

<b>A</b>		0	1	2	3	4	5	6	7	8
关键字		30	63	65	69	73	78	83	89	95

<b>B</b>										
关键字		30	63	73	83	95	65	78	89	69

<b>count</b>	0	0	0	0	1	1	1	4	6	8
	0	1	2	3	4	5	6	7	8	9

6) 从后到前将B中的记录依照“count”指示的地址复制到A

# 各种排序的综合比较

选择排序方法考虑因素

- 待排序记录数 $n$
- 记录本身其他信息量大小
- 关键字分布情况
- 稳定性要求

时间性能

- 按时间性能分三类 :简单、先进、基数
- 有序性对算法的影响
  - 插入和起泡有好的影响 $O(n)$
  - 快速有坏的影响 $O(n^2)$
  - 选择、归并、堆、基数没有影响
- 记录其他信息复杂度对算法的影响
  - 起泡、快速、堆等移动较多的算法影响较大

## 空间性能

- 所有简单排序、堆是 $O(1)$
- 快速是 $O(\log n)$
- 归并、基数 $O(n)$

## 稳定性

- 快速和堆大范围交换的算法不稳定
- 其他算法均稳定。

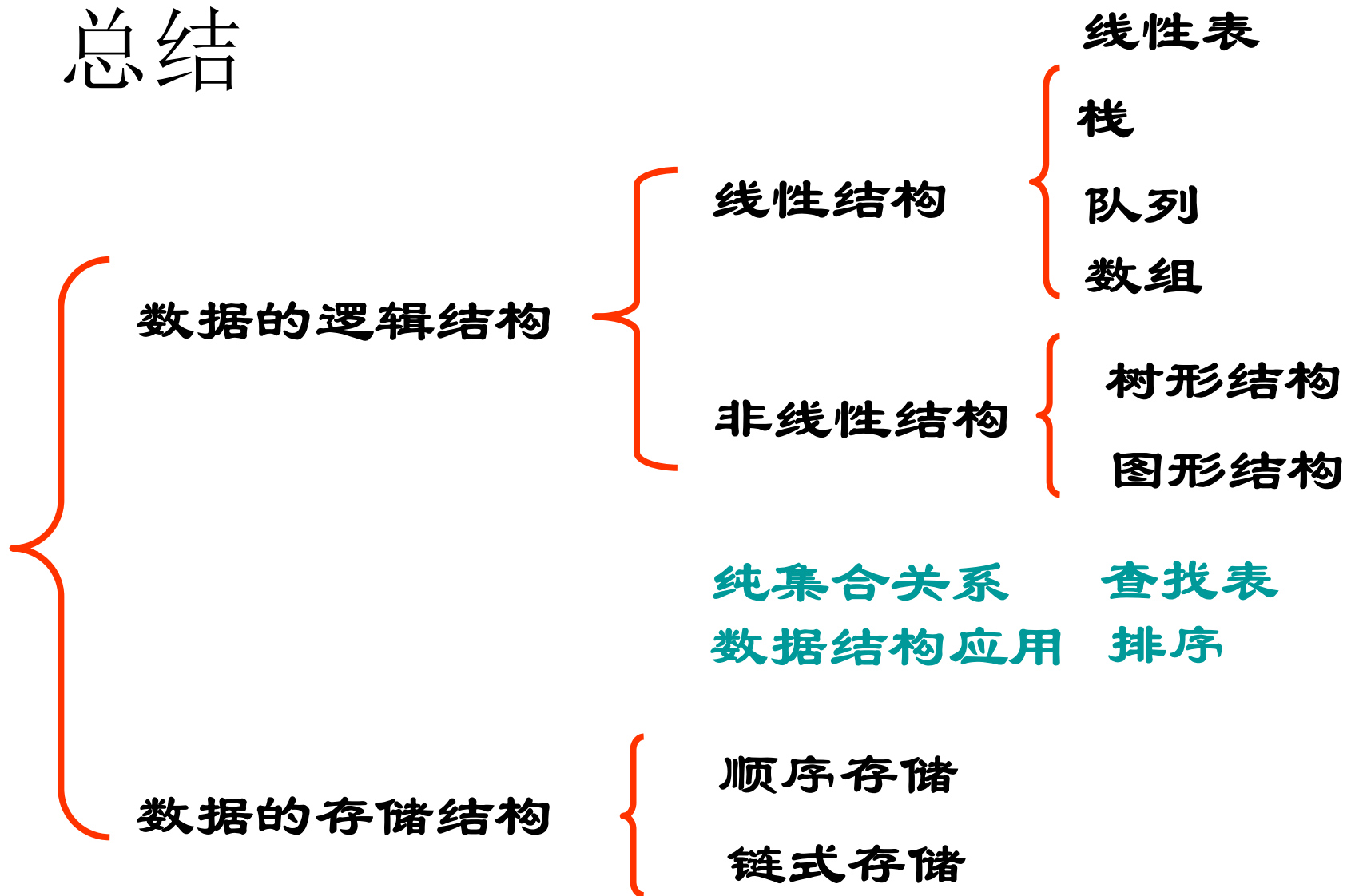
经过一趟排序能将一个数据放在最终位置

- 选择、起泡、堆

# 各种排序的综合比较表

算法	平均时间	最坏情况	最好情况	辅助空间	稳定性	有序性影响
选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Y	无
插入	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	Y	好
起泡	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	Y	好
快速	$O(n\log n)$	$O(n^2)$	$O(n\log n)$	$O(\log n)$	N	坏
归并	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$	Y	无
堆	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$	N	无
基数	$O(n*d)$	$O(n*d)$	$O(n*d)$	$O(n)$	Y	无

# 总结





谢 谢！