数据结构

袁平波 2021.2

课程主页：http://staff.ustc.edu.cn/~ypb/

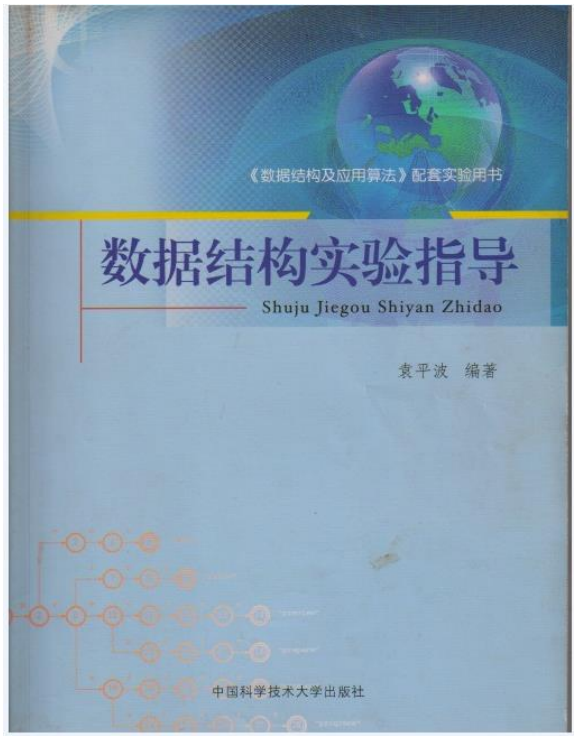
课程情况

- 60理论课时：数据结构40 数据库20
  - 作业一章提交一次
- 30实验课时
  - 10次 /每次3课时 或 8次/每次4课时
- 课程考核
  - 考试 60-70%，平时（作业，出勤）10%，实验30-20%
  - 6个实验，提交1-2份实验报告，其他实验随堂检查
- 课堂要求
  - 请假在上课前完成

参考教材


清华大学 计算机系列教材
累计发行超400万册
数据结构
（C语言版）
清华大学出版社

实验指导教材


《数据结构及应用算法》配套实验用书
数据结构实验指导
Shuju Jiegou Shiyan Zhidao
袁平波 编著
中国科学技术大学出版社

投票

最多可选1项

设置

你是否学习过C++语言？

A 是

B 否

提交

投票 最多可选3项

设置

你主要使用下面哪种C语言环境

A MS VC6

B DEV C++

C Linux gcc

D 其他

提交

投票 最多可选1项

设置

你使用过debug调试程序吗？

A 是

B 否

提交

第一章 绪论

1.1《数据结构》讨论范畴

1.2 基本概念和术语

1.3 抽象数据类型的表示与实现

1.4 算法和算法分析

1.1《数据结构》研究什么

☞算法+数据结构=程序设计

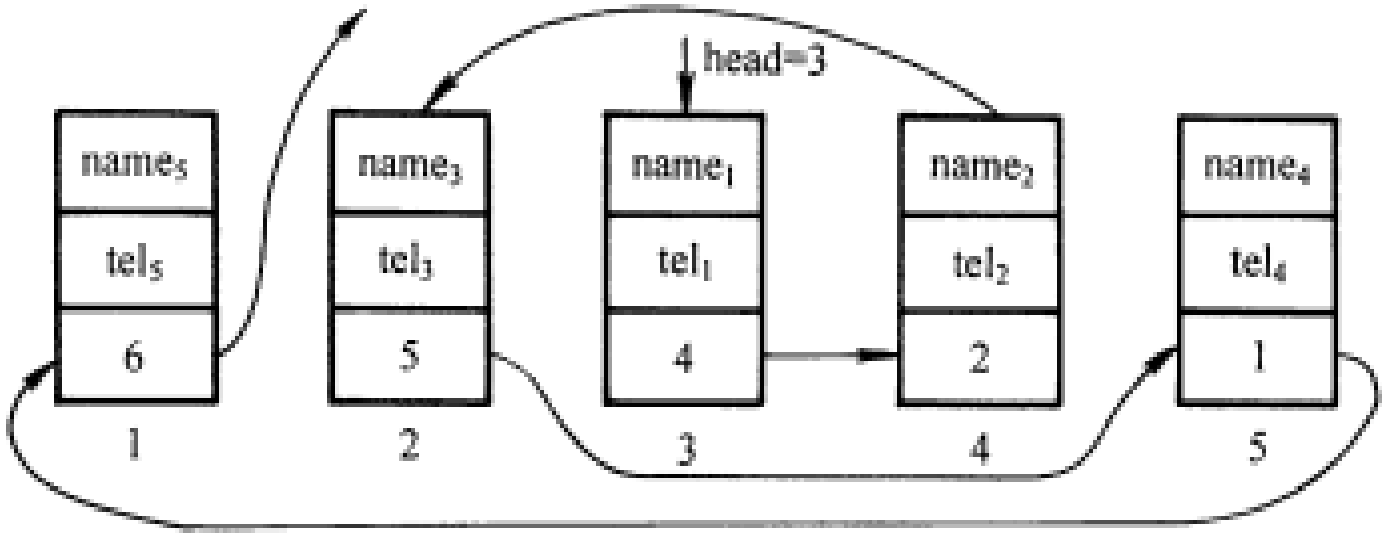
- 1968 美国唐·欧·克努特 设立《数据结构》课程
- 1976 瑞士 Niklaus Wirth 提出算法+数据结构=程序设计

☞ 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

[例]、设有一个电话号码薄，有N个人的姓名和电话号 码。要求设计一个程序，按人名查找号码，若不存在则给出不存在的信息。

姓　　名	$name_1$	$name_2$	$name_3$	……	$name_n$
电话号码	tel_1	tel_2	tel_3	……	tel_n

(a) 顺序存储



(b) 链式存储

- [例] 下棋 井子棋 非线性数据结构-树

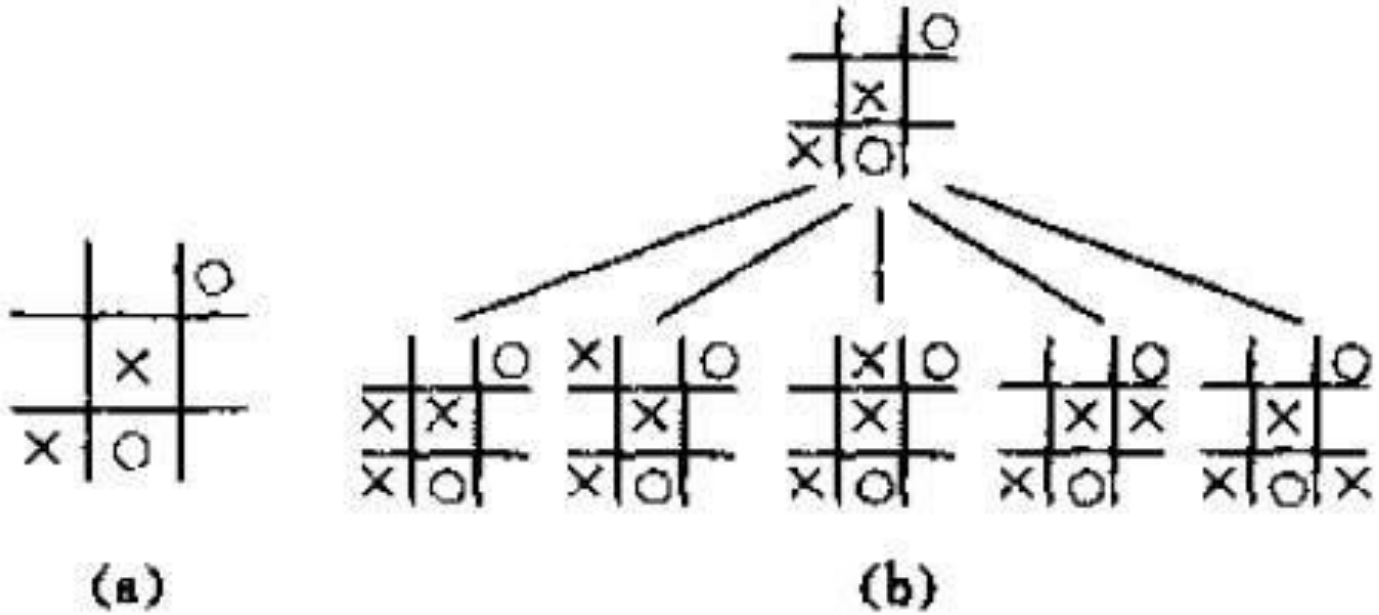


图 1.2 井字棋对弈"树"

(a) 棋盘格局示例；(b) 对弈树的局部。

[例] 交叉口交通灯设置（顶点着色问题）

非线性——图

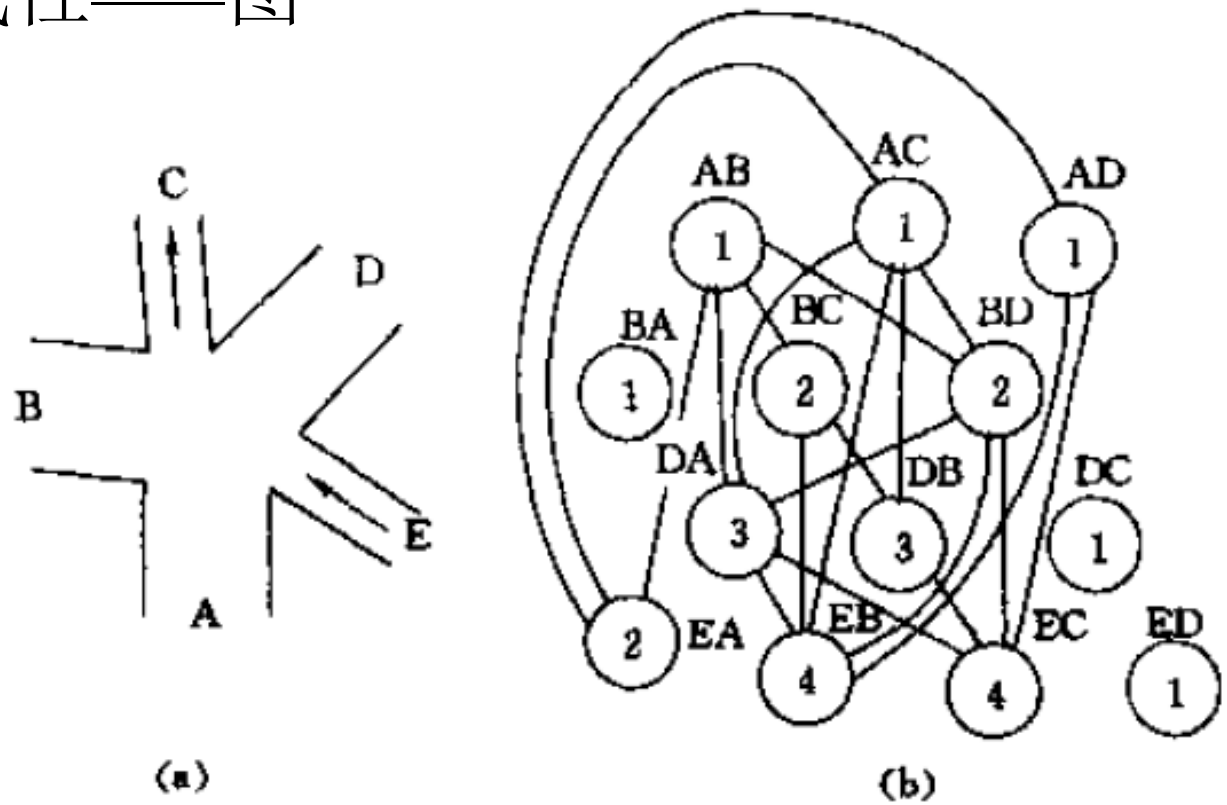


图 1.3　五叉路口交通管理示意图

(a) 五叉路口；(b) 表示通路的图

1.2 基本概念和术语

- **数据：**数据是客观事物的符号表示。
- **数据元素：**数据的基本单位，通常看作一个整体。
- **数据对象：**性质相同的数据元素的集合。
- **关系：**集合中元素之间的相关性。
- **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 。

Data-Structure=(D,S)

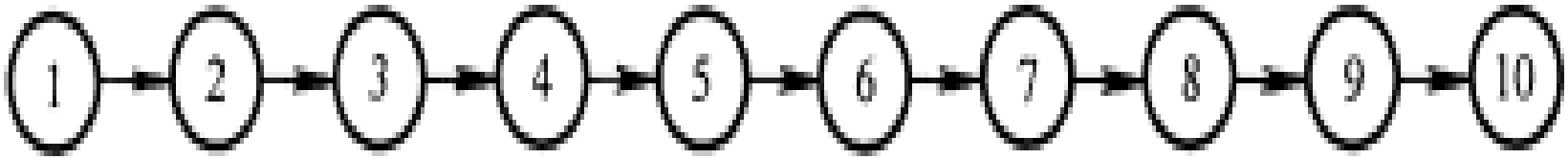
D是元素有限集，S是关系有限集。

四类基本结构：1)集合 2)线性 3)树形 4)网状(图)

[例] linear=(D,R)

D={1,2,3,4,5,6,7,8,9,10}

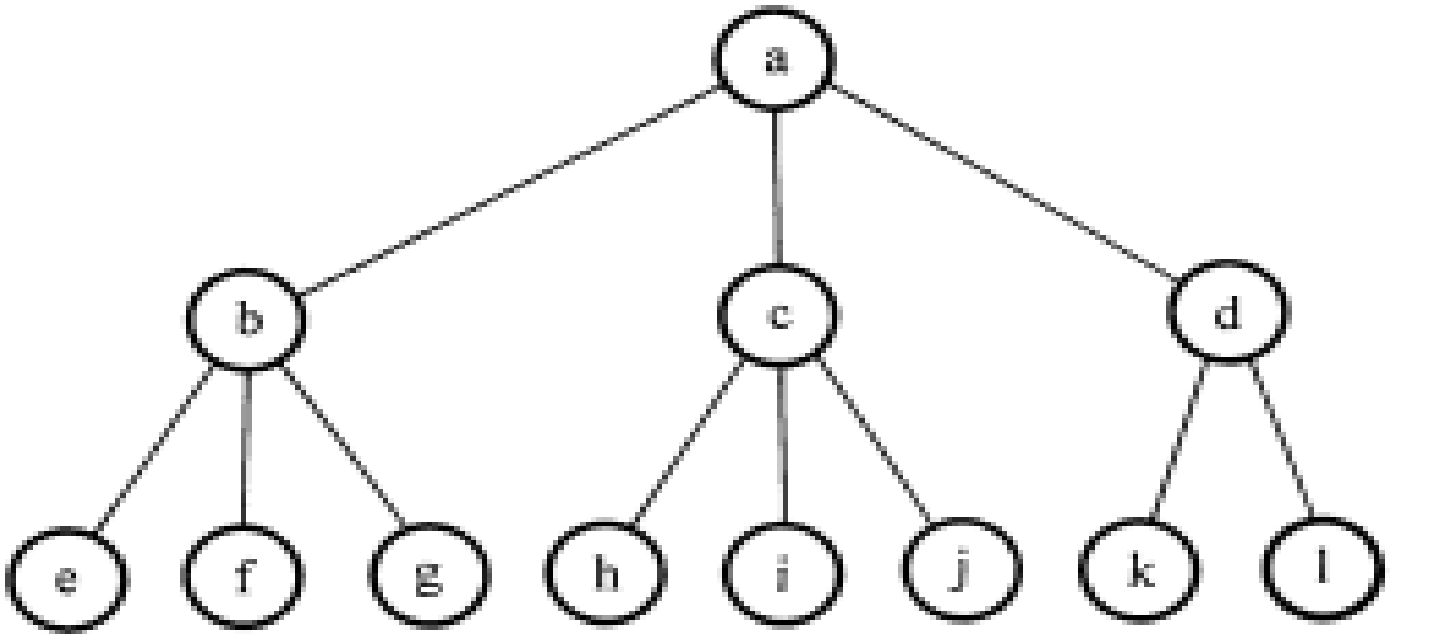
R={<1,2>,<2,3>,<3,4>,<4,5>,<5,6>,
<6,7>,<7,8>,<8,9>,<9,10>}



[例]tree=（D,R）

D={a, b, c, d, e, f, g, h, i, j, k, l}

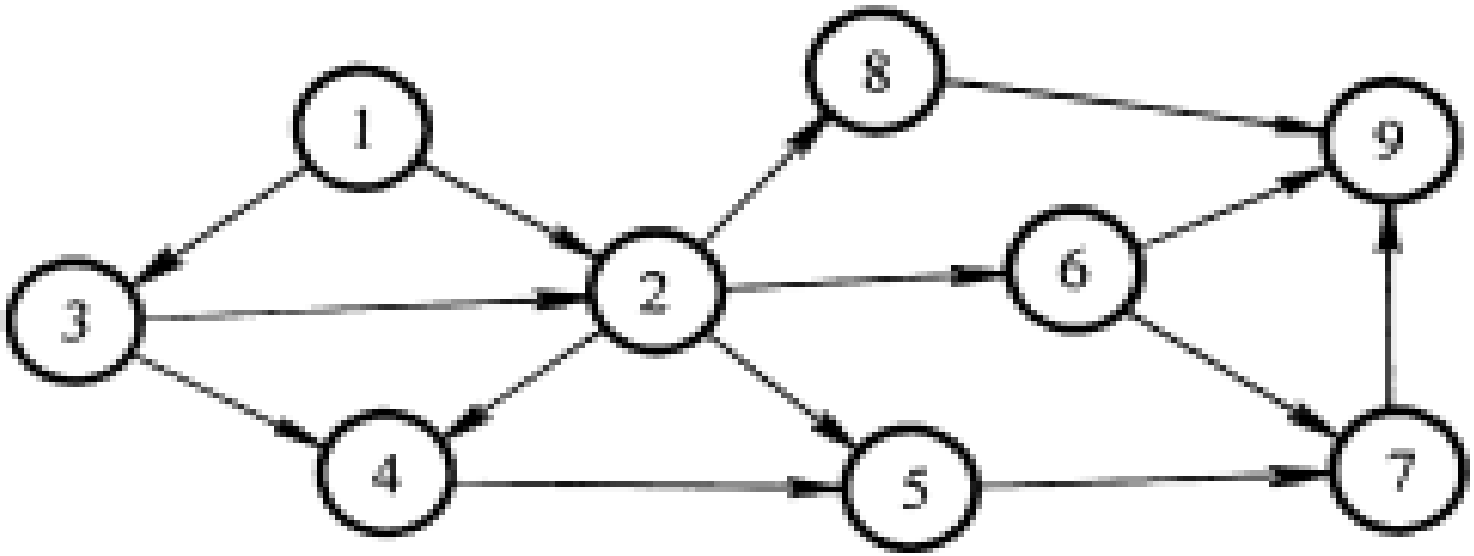
R={<a,b>,<a,c>,<a,d>,<b,e>,<b,f>,<b,g>,<c,h>,<c,i>,
<c,i>, <d,k>,<d,l>}



[例] graph=(D,R)

D={1, 2, 3, 4, 5, 6, 7, 8, 9}

R={<1,2>,<1,3>,<2,4>,<2,5>,<2,6>,<2,8>,<3,2>,<3,4>, <4,5>,<5,7>,<6,7>,<6,9>,<7,9>,<8,9>}



其他概念与术语

- 逻辑结构/物理结构(存储结构)
- 位(bit)/字节(Byte)
- 元素(Element)/结点(Node)/数据域(DataField)
- 顺序存储/链式存储
- **数据类型**：值的集合和定义在这个值集上的一组操作的总称。分**原子**和**结构**两种类型。
- **抽象数据类型**(Abstract Data Type)：
  一个数学模型以及定义在该模型上的一组操作。

1.3 抽象数据类型的表示与实现

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。

格式：

ADT 抽象数据类型名{

数据对象：<数据对象的定义>

数据关系：<数据关系的定义>

基本操作：<基本操作的定义>

} ADT 抽象数据类型名

基本操作的 定义格式：

基本操作名（参数表）

初始条件：〈初始条件描述〉

操作结果：〈操作结果描述〉

[例]抽象数据类型三元组的定义

ADT Triplet{

数据对象:D={e1,e2,e3|e1,e2,e3∈Elemset}

数据关系:R={<e1,e2>,<e2,e3>}

基本操作:

InitTriplet(&T,v1,v2,v3)

操作结果:构造三元组T，元素e1,e2,e3分别赋予v1,v2,v3

Get(T,i,&e)

初始条件:三元组T存在，1≤i≤3。

操作结果:用e返回T的第i元的值。

Max(T,&e)

初始条件:三元组T存在。

操作结果:用e返回T中三个元素的最大值。

… …

}ADT Triplet

操作中的参数有两种

1）赋值参数

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

2）引用参数

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

类C语言——介于伪码和C语言之间

1）预定义常量和类型

```
#define TRUE 1
typedef int Status;
```

2）数据元素类型约定为ElemType，由用户使用时自行定义。

3）基本操作的算法用函数描述：

```
函数类型 函数名（函数参数表）{
        //算法说明
        语句序列
}//函数名
```

4）赋值语句

简单赋值：变量名=表达式；

串联赋值：变量名1=变量名2=…=表达式；

成组赋值：变量名1[起始下标..终止下标]=变量名2[起始下标..终止下标]；

交换赋值：变量名1←→变量名2；

条件赋值：变量名=条件表达式?表达式T：表达式F

5)选择语句

条件语句1：if(表达式)语句;

条件语句2：if(表达式)语句; else 语句;

开关语句1：
```
switch(表达式){
    case 值1:语句序列1;break;
    … …
    case 值n:语句序列n;break;
    default:语句序列n+1;
}
```

开关语句2：
```
switch{
    case 条件1:语句序列1;break;
    … …
    case 条件n:语句序列n;break;
    default:语句序列n+1;
}
```

6)循环语句

for语句:　　for(赋初值;条件;修改表达式)语句;

while语句:　　while(条件)语句;

do-while语句:　do{语句序列}while(条件);

7)结束语句

函数结束语句　return; return 表达式;

异常结束语句　exit;

8)输入输出语句

输入语句　scanf

输出语句　printf

9)注释　　单行注释用“//”

10)基本函数: max,min,abs,floor,ceil,eof,eoln等。

11)逻辑运算

与运算&&　　或运算||

1.4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≥n0时都满足

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

换句话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。

$$\lim_{n\to\infty}\frac{|X_n|}{|f(n)|}=C$$

分析下列三个程序段：

①、x++;s=0; O（1）

②、for (i=1;i<=n;i++) {x++;s+=x;} O（n）

③、for(i=1;i<=n;i++)

for (j=1;j<=n; j++) {x++; s+=x;} O（n^2）

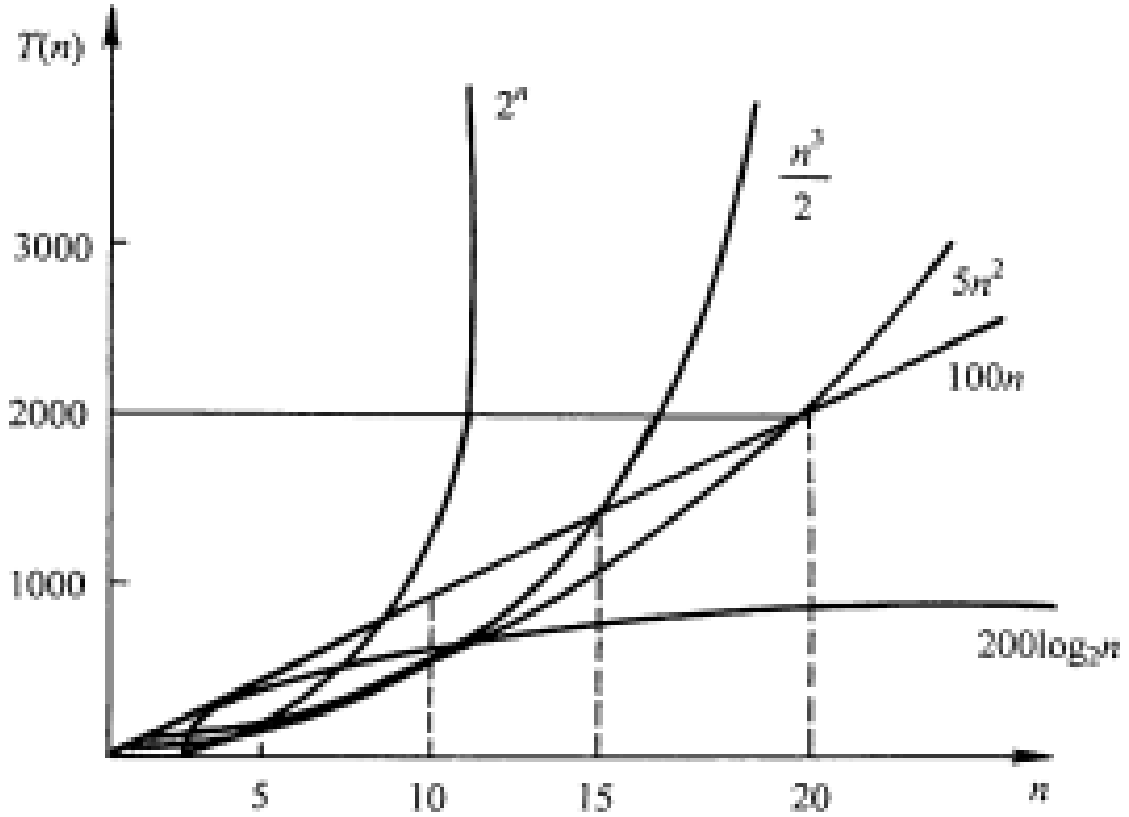
④、$f(n)=2n^3+5n^2+7n+c$;（c为某一常数），则

T（n）= O (f(n))=O（n^3）。

[例] 起泡排序　　算法 1.3

```
void bubble_sort(int a[], int n){
    for (i=n-1, change=TRUE;  i>=1 && change;  --i) {
      change = FALSE;
      for (j=0;  j<i;  ++j)
        if (a[j] > a[j+1])
          {a[j]←→a[j+1]; change = TRUE }
    }
} // bubble_sort
```

时间复杂度$O(n^2)$



常见函数的增长率

第二章　线

2.1 线性表的类型定义

2.2 线性表的顺序存储

2.3 线性表的链式存储

2.4 一元多项式的表示和相加

2.1 线性表的类型定

线性表的定义

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

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

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 ai 在线性表中的位序

线性表的 ADT 定义

ADT List{

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

数据关系：$R=\{<a_{i-1},a_i>|a_{i-1},a_i \in D, i=2,\ldots n\}$

基本操作：

InitList(&L)

DestroyList(&L)

ClearList(&L)

ListEmpty(L)

ListLength(L)

GetElem(L,i,&e)　　1<=i<= ListLength (L)

LocateItem(L,e,compare())

PriorElem(L,Cur_e,&pre_e)

NextElem(L,cur_e,&next_e)

ListInsert(&L,i,e)　　1<=i<= ListLength (L)+1

ListDelete(&L,i,&e)　　1<=i<= ListLength (L)

ListTraverse(L,visit())

}ADT List

【例】对两个线性表 La，Lb 表示的集合 A 和 B，求一个新集合 A = AUB // 算法 L

a) 从 Lb 中取一个元素 (重复直至遍历全部 Lb 元素)

b) 在 La 中查询

c) 若 La 中不存在，则将其插入 La

【例】对一个非纯集合 *B*，构造一个纯集合 *A*，使 *A* 中只包涵 *B* 中不同值的成员。同上，只是创建 *La*。

【例】判断两个集合是否相等。构造 *C* = *A*

【例】已知线性表 La，Lb 中数据元素按值非递减排列，现要求将 La 和 Lb 归并为一新的线性表 Lc, 且 Lc 也非递减排列。 // 算法 2.?

2.2 线性表的顺序表示和实现

顺序表—线性表的顺序存储表示

```
#define LIST_INIT_SIZE 100          (C 规范 )
#define LISTINCREMENT 10
Typedef  Struct {
        Elemtype                * elem;
        int                     length;
        int                     listsize;
}SqList;
```

顺序表中基本操作的实现

初始化操作　InitList_Sq

查找元素操作　LocateItem_Sq O(n)

插入元素操作　ListInsert_Sq　O(n)

删除元素操作　ListDelete_Sq　O(n)

归并操作　MergeList_Sq　O(n+m)　// 算法 2.7

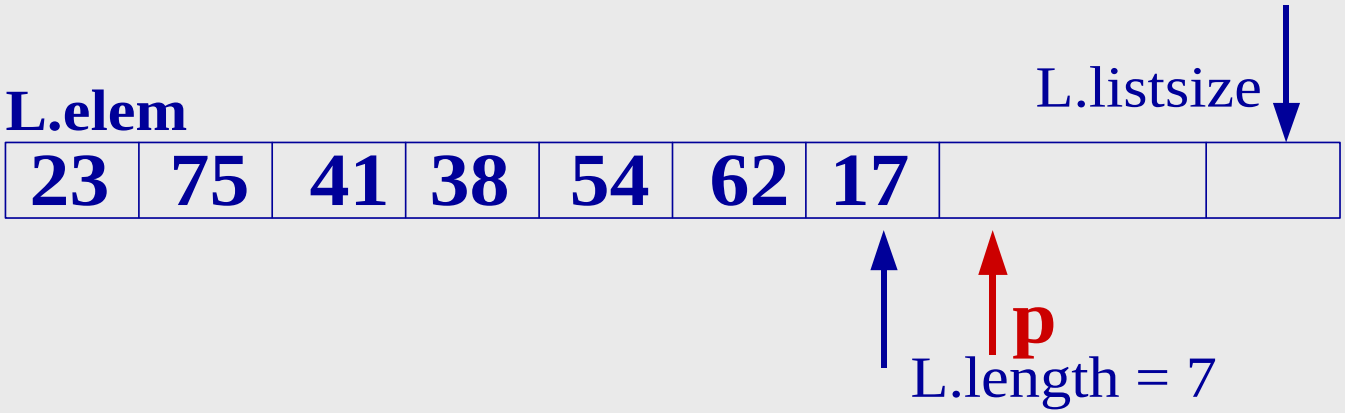
对比算法 2.2　2.7 2.12

插入和删除操作的时间分析：

$$Ein=\Sigma p_i(n-i+1)=n/2$$

$$Edl=\Sigma p_i(n-i)=(n-1)/2$$

例如：顺序表



i 8

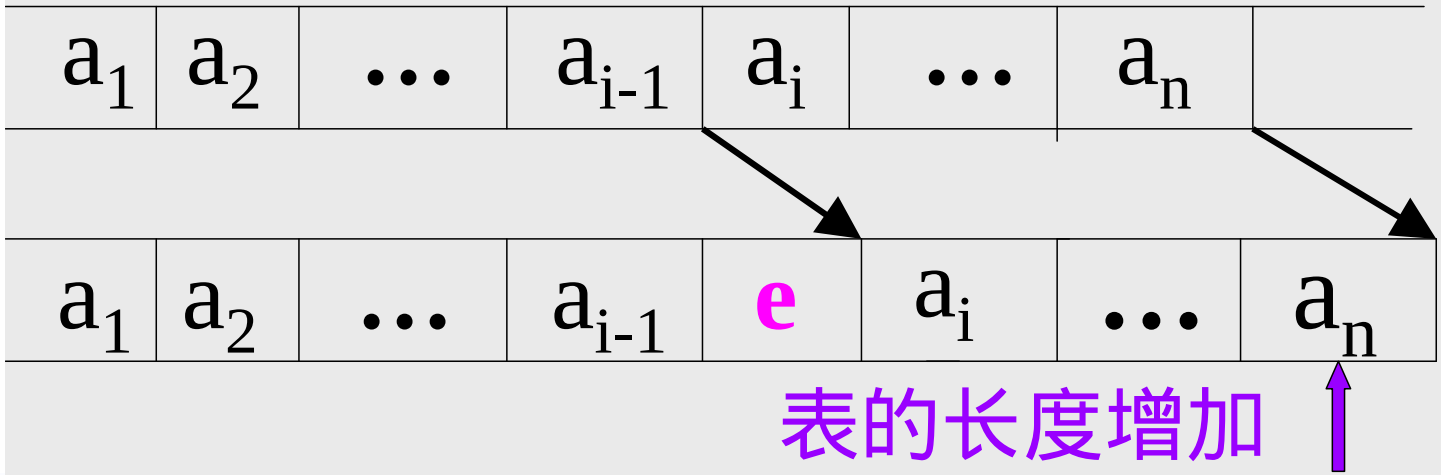
e = 38　　返回值 = 4

e = 50　　返回值 = 0

$(a_1, \ldots, \mathbf{a_{i-1}, a_i}, \ldots, a_n)$ 改变为

$(a_1, \ldots, \mathbf{a_{i-1}, e, a_i}, \ldots, a_n)$

$\mathbf{<a_{i-1}, a_i>}$ ⟶ $\mathbf{<a_{i-1}, e>, <e, a_i>}$

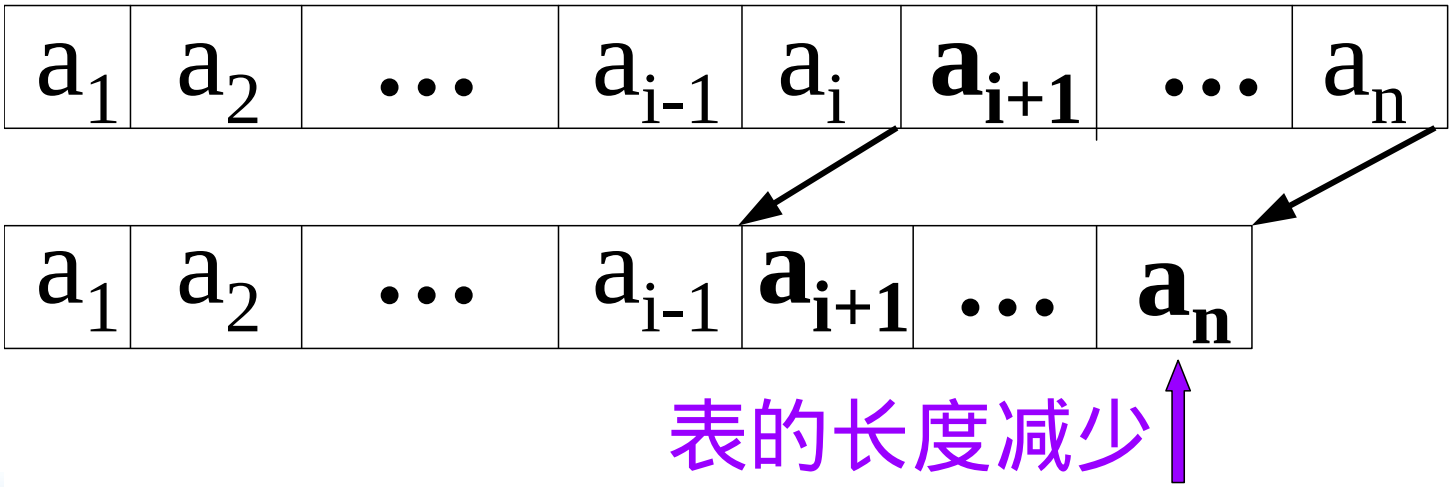


删除元素操作　时间复杂度 O(n)

$(a_1, \ldots, \mathbf{a_{i-1}, a_i, a_{i+1}}, \ldots, a_n)$ 改变为

$(a_1, \ldots, \mathbf{a_{i-1}, a_{i+1}}, \ldots, a_n)$

$\mathbf{<a_{i-1}, a_i>, <a_i, a_{i+1}>} \Rightarrow \mathbf{<a_{i-1}, a_{i+1}>}$



顺序表其它算法举例 *

- 比较两个顺序表的大小 （逐个比较元素字符）

  时间复杂度 O(Min(A.length,B.length))

- 用尽量少得辅助空间将前 m 个元素和后 n 个元素互换
  - *exchange1* 时间复杂度： $O(m \times n)$
  - *invert* 时间复杂度： $O(t-s+1)$
  - *exchange2* 时间复杂度： $O(m+n)$

注意：代码简洁和效率是两回事

2.3 线性表的链式表示和实

2.3.1 线性链表（单链表和指针）

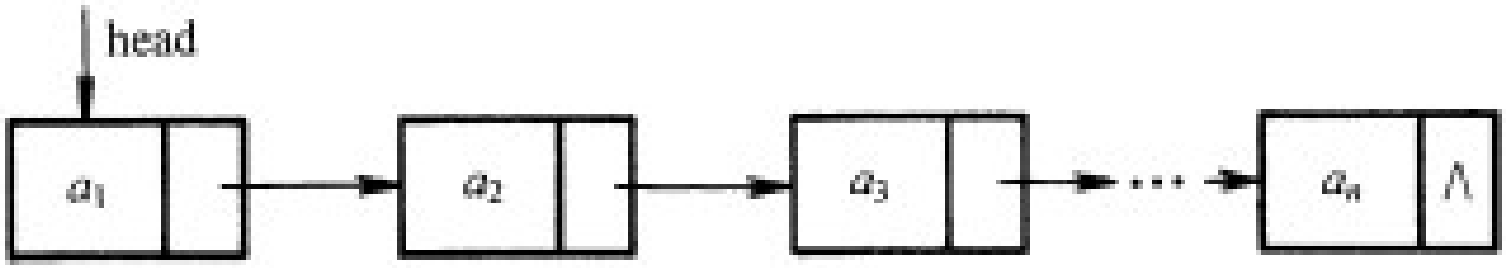
- 数据域（ data ）和指针域（ next ）
- 存储表示

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

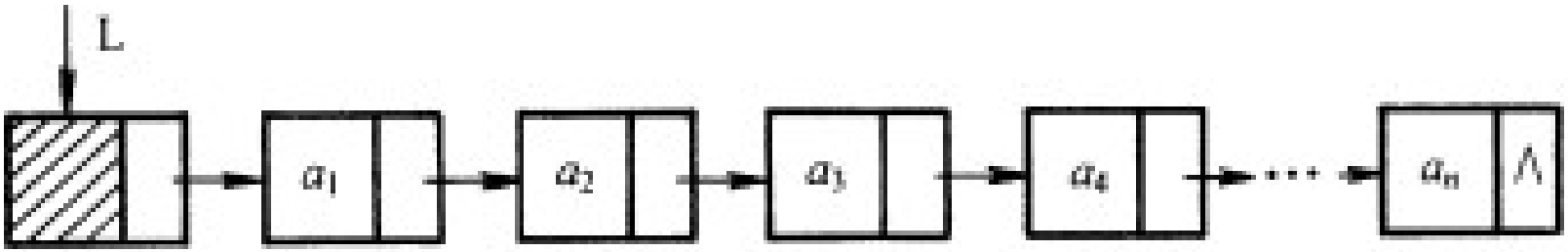
单链表种类

不带头结点单链表

带头结点单链表



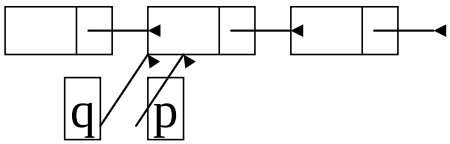
(a) 不带头结点的单链表



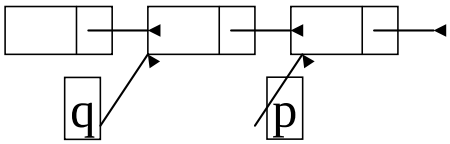
(b) 带头结点的单链表

常见指针操作 *

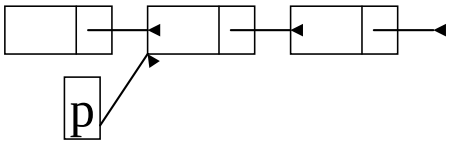
p=q



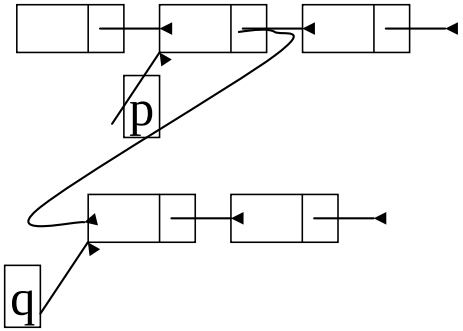
p=q → next



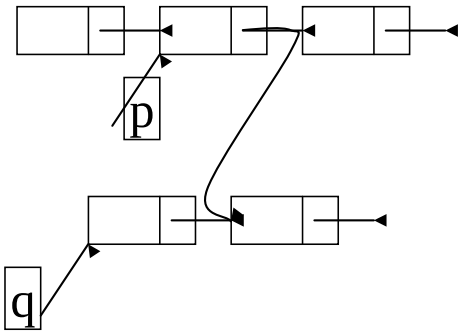
p=p→ next



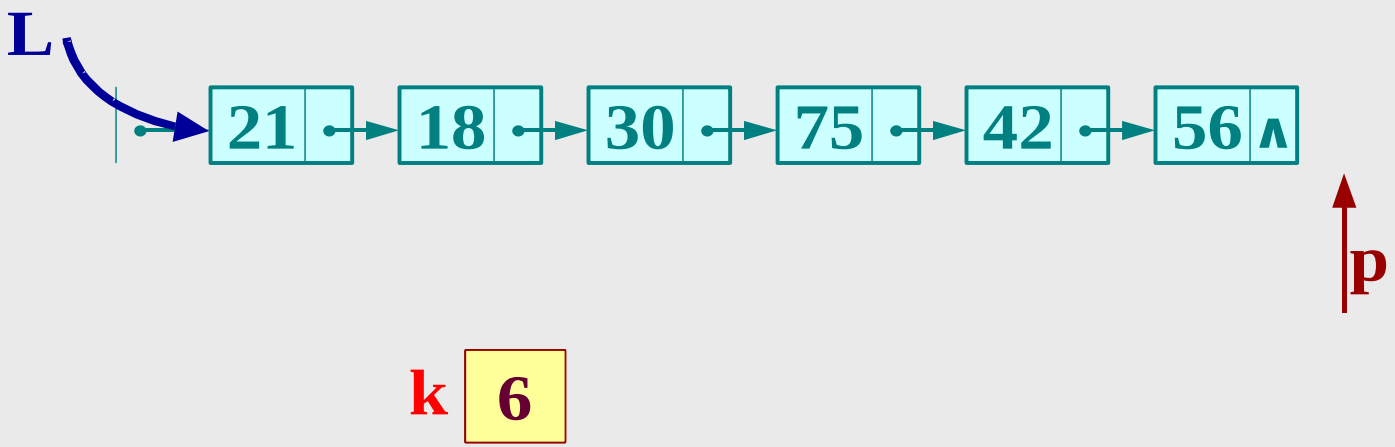
p→ next=q



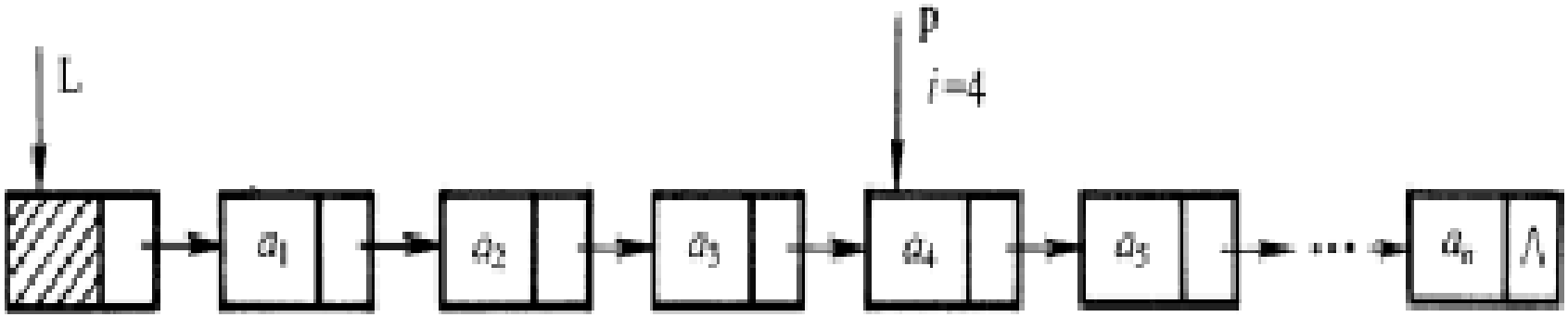
p→ next=q→ next



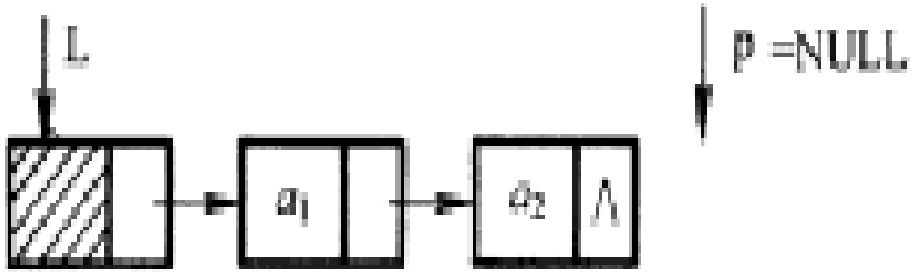
求线性表的长度　时间复杂度： O(n)



查找元素操作　　时间复杂度：O(n)



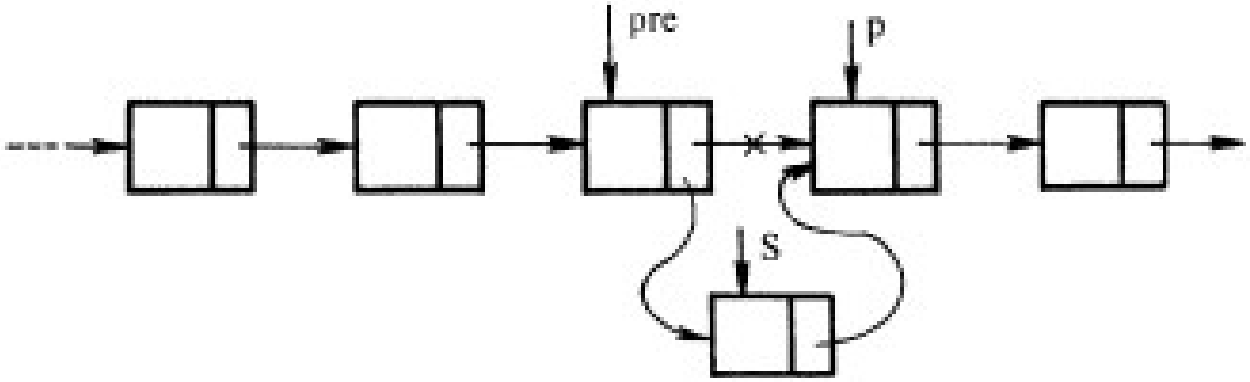
(a) 查找成功(从链头开始后移指针 $i-1$ 次)



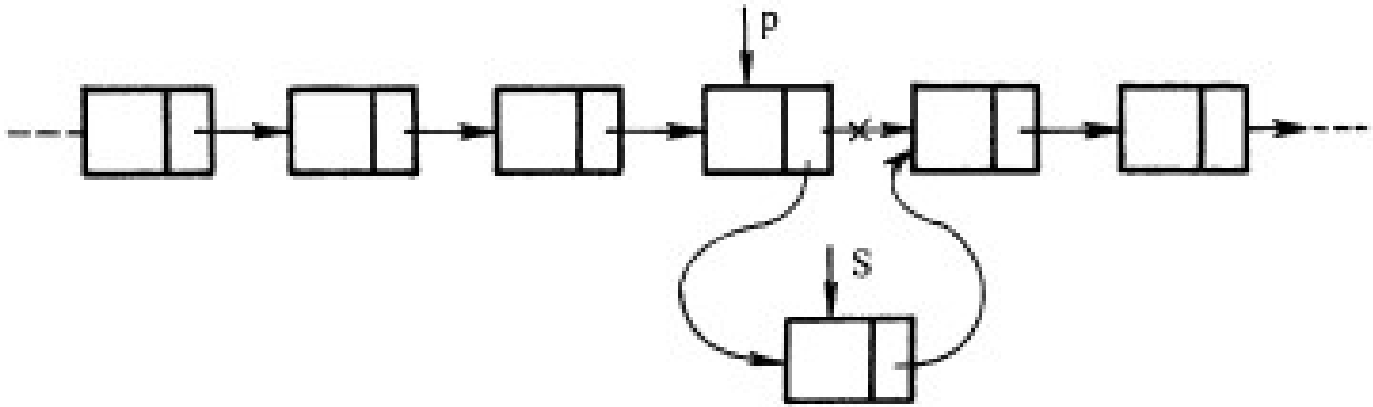
(b) 查找失败(从链头开始后移指针未到 $i-1$ 次,指针变空)

插入结点操作 ：前插、后插

时间复杂度：前插 O(n)、后插 O(1)

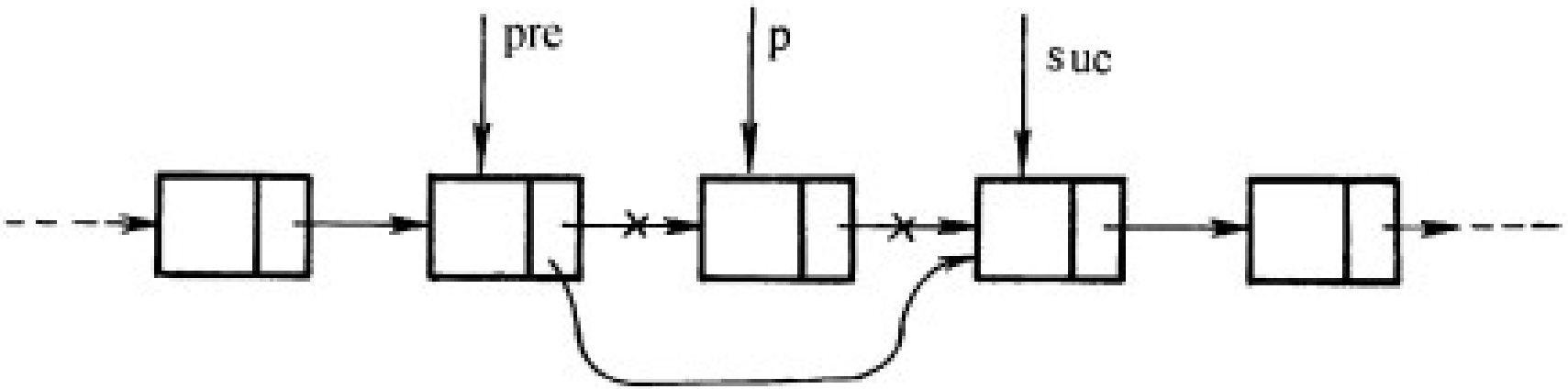


(a) 插入到 p 之前(必须知道 p 的前驱 pre)



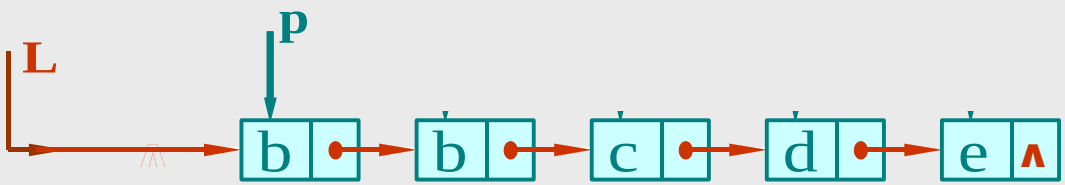
(b) 插入到 p 之后(不须知道 p 的前驱)

删除结点操作　　时间复杂度 O(n)

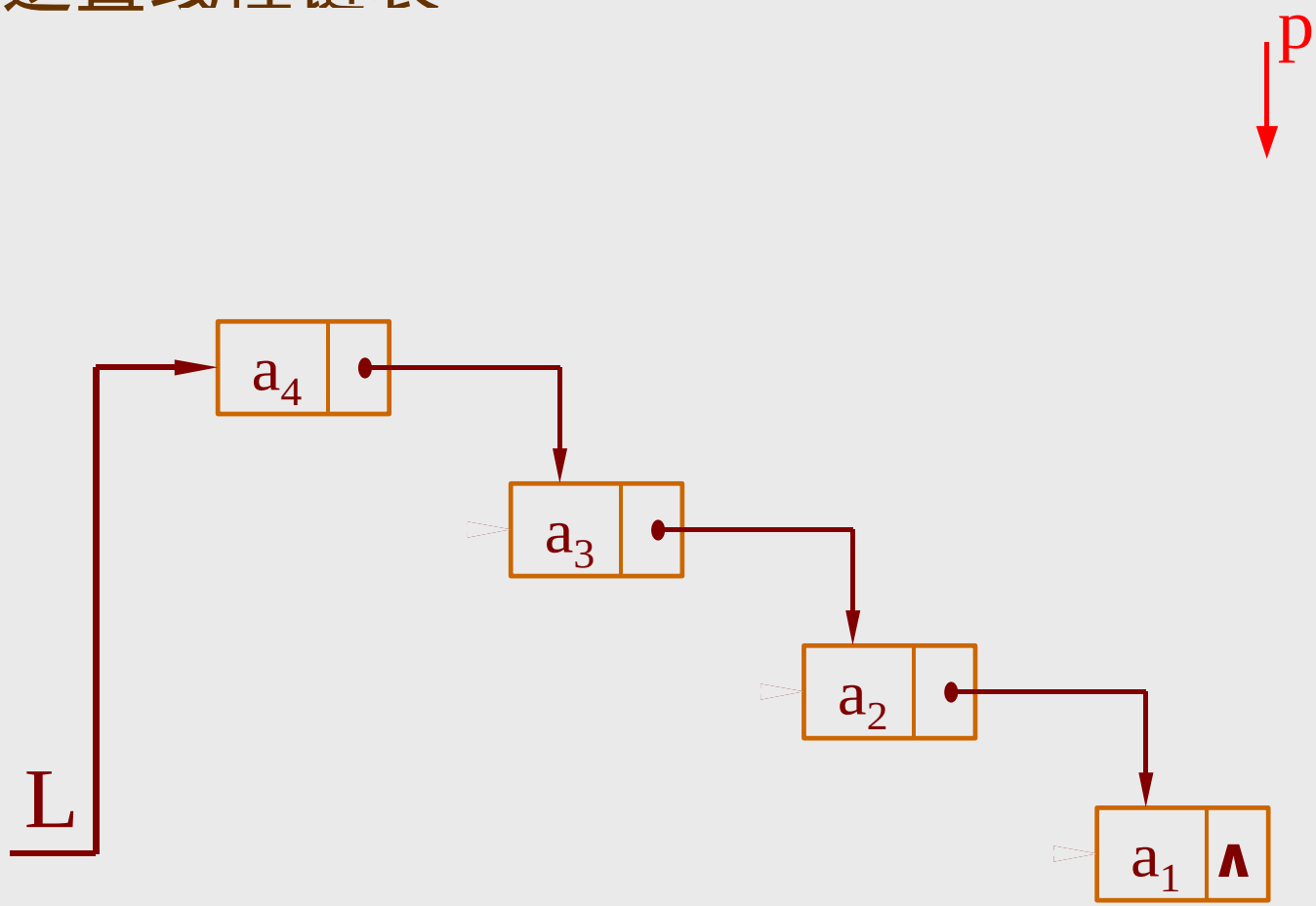


单链表的其它操作举例

【例】逆序创建链表 时间复杂度 O(n)



逆置线性链表



【例】以单链表表示线性表实现集合的并 *

```
void union_L( LinkList &La, LinkList &Lb ) {
    if (!La) {La = Lb;return;}
    while ( Lb ) {
        s = Lb;  Lb = Lb->next;
        p = La;
        while ( p && p->data != s ->data ) { // 查找
            pre  =  p;   p  =  p->next;
        }//while
        if ( p ) delete s;  // 找到
        else { pre->next = s;  s->next = NULL;} // 插入
    }// while
}// union_L
```

对比算法 2.1

时间复杂度 O(m×n)

【算法改进】Lb 中元素只需要和原 La 元素比较 *

```
void union_L( LinkList &La, LinkList &Lb ) {
    if (!La) La = Lb;return;
    q=La; while(q->next) q=q->next; // 先找到链表 La 尾部 q
    pre=q;   //pre 为并操作以后的尾部
    while ( Lb ) {
    s = Lb;  Lb = Lb->next;
    p = La;
    while ( p !=q->next&& p->data != s ->data )p =p->next;
    if ( p!=q->next ) delete s;
    else { pre->next = s; pre=s}
    }// while(Lb)
    s->next=NULL;
}// union_L
```

数据有序性对链表的影响——有序表 *

【例】 两个带头结点的有序单链表 La ， Lb 分别表示集合 A ， B ，求 C=AUB? // 算法 2.12

时间复杂度 O(n)

对比：无序表完成同样功能的时间复杂度为 O(n*n)

静态链表

```
#define MAXSIZE 1000 // 链表最大长度
Typedef struct {
    ElemType  data;
    int                cur;
}component,SLinkList[MAXSIZE];
```

int LocateElem_SL(SLinkList S ,ElemType e);// 查找元素

int InitSpace_SL(SLinkList &space); // 生成备用链表

int Malloc_SL(SLinkList &space); // 返回可用下标

int Free_SL(SLinkList &space, int k);// 回收 k 下标节点

求集合 (A-B)U(B-A)

静态链表实现 (A-B)U(B-

A

Space(0 ： 11)

0		8
1		2
2	c	3
3	b	4
4	e	5
5	g	6
6	f	7
7	d	0
8		9
9		10
10		11
11		0

Space(0 ： 11)

0		6
1		2
2	c	4
3	n	8
4	e	5
5	g	7
6	f	9
7	d	3
8	a	0
9		10
10		11
11		0

r=7 a 和 n 依次
插在 r 后面

(A-B)∪(B-A)

- A={c,b,e,g,f,d} B={a,b,n,f}

2.3.2 循环链表

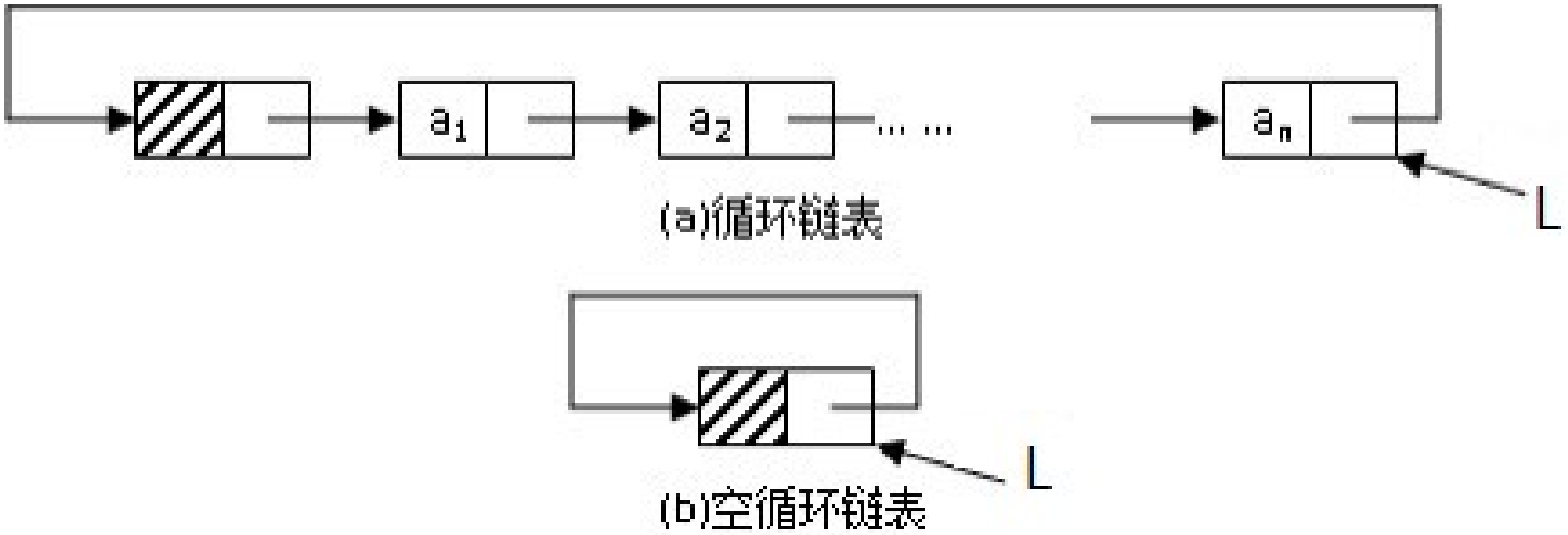
什么是循环链表

- 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
- 空的循环链表是头结点指针指向自己

判断表尾的循环条件：

- 不是 p==NULL ，而是 p 是否等于头指针（尾指针）。

单循环链表



有设头指针和尾指针两种，设尾指针更方便

【例】 两个带头结点的循环有序链表，表示集合 A ， B ，完成 C=A U B

时间复杂度 O(n+m)

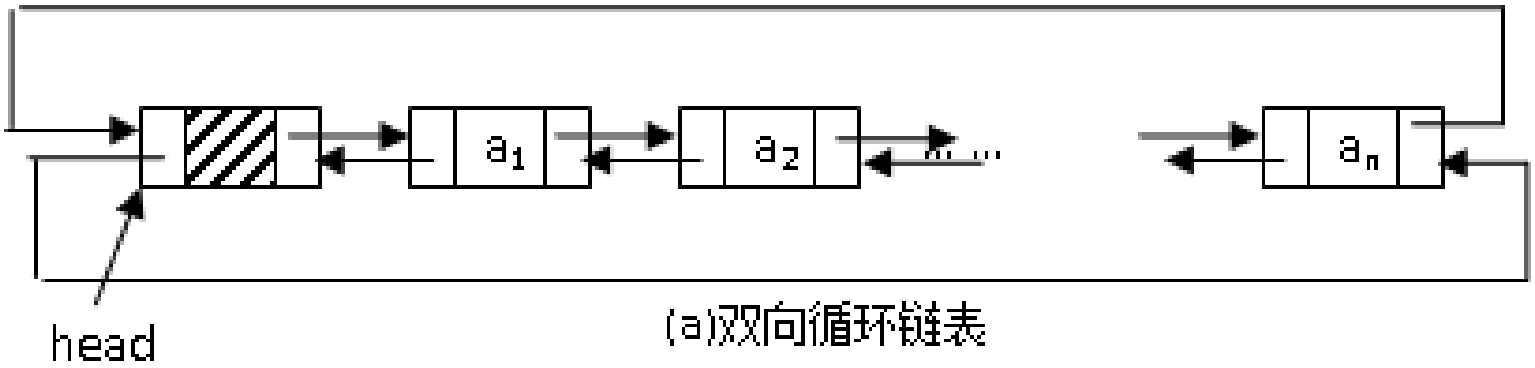
在头元素中放最大元素 MAX 简化操作 ，时间复杂度 O(n+m), 时间略长，算法表达略简单

2.3.3 双向链表

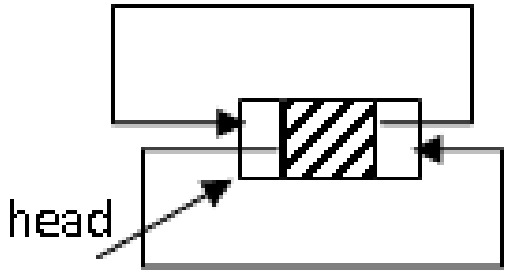
概念：两个指针，分别指向前驱和后继，双向链表一般都是循环带头结点的。

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

双向循环链表



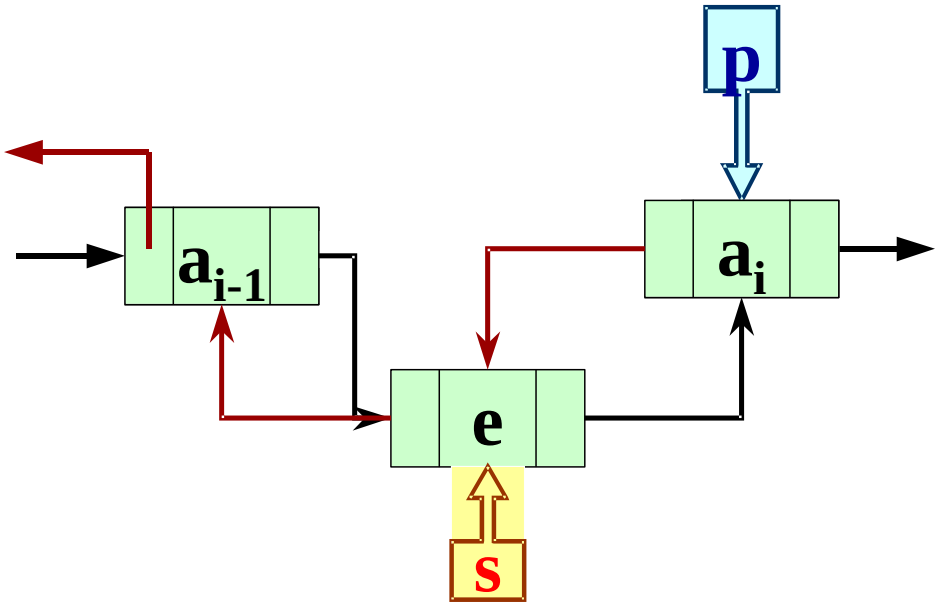
(a)双向循环链表



(b)空双向循环链表

插入结点操作 时间复杂度 O(1)

插入



s->prior = p->prior;

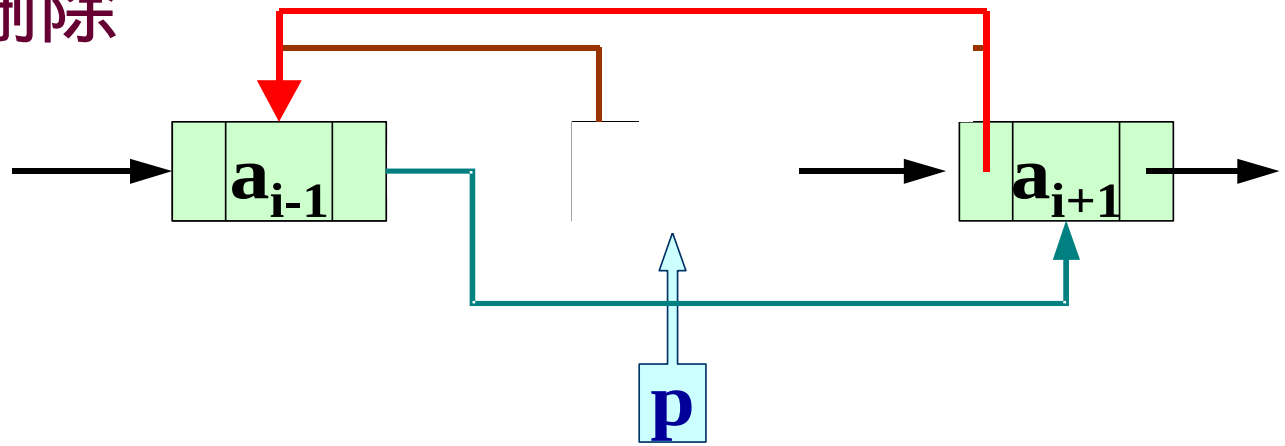
p-> prior -> next = s;

s->next = p;

p->prior = s;

删除结点操作 时间复杂度 O(1)

删除




```
p->prior->next = p->next;
p->next->prior = p->prior;
delete p;
```

单链表的实际应用改进

```
typedef struct LNode{
      ElemType            data;
      Struct LNode       *next;
         }*Link, *Position;
typedef struct {
     Link       head,tail;
     int        len;
     }LinkList;
Status MakeNode(Link &p,ElemType e);// 分配由 p 指向值为 e 的结点
void FreeNode(Link &p); // 释放由 p 指向的结点
Status InitList(LinkList &L) // 注意这里的 L 是一个结构体
Status InsFirst(Link h,link s);//h 头结点，插在第一位置
   …
Status ListInsert(LinkList &L,int i,ElemType e);
Status MergeList_L2(LinkList &La, LinkList &Lb, LinkList &Lc);
```

2.4 一元多项式的表示和相加

一元多项式 $P_n(x)=p0+p_1x^1+p_2x^2+\ldots+p_nx^n$

用 $((p_0,e_0), (p_1,e_1), \ldots,(p_n,e_n),)$ 表示 ，是线性表

```
typedef  struct{
    float    coef;   // 系数
    int              expn;  // 指数
}term,ElemType;
```

ADT Polynomial{

数据对象：$D=\{a_i|a_i\in TermSet,i=1,\ldots m\}$

数据关系：$R=\{<a_{i-1},a_i>|a_{i-1}\in D,a_i\in D,e_{i-1}<e_i\}$

基本操作：

CreatePolyn(&p,m);

AddPolyn(&pa,&pb);

初始条件：一元多项式 pa ， pb 已经存在

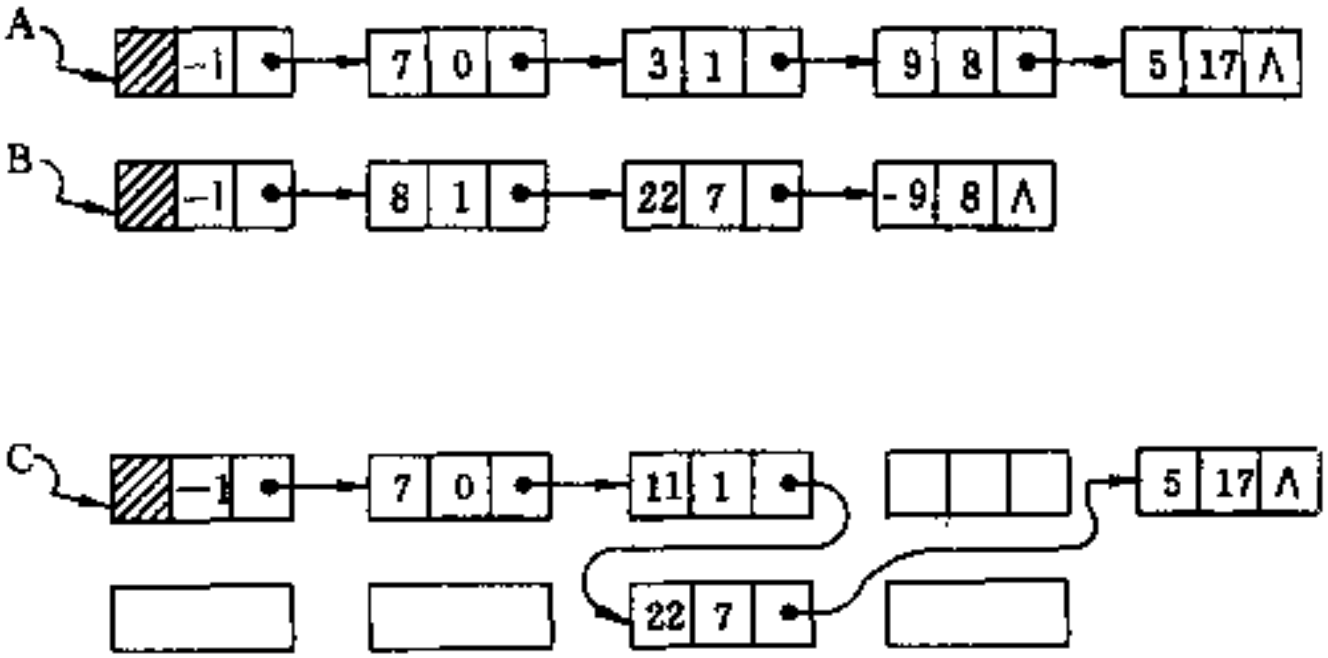
操作结果：完成 pa=pa+pb

SubtractPolyn(&pa,&pb)

… …

} ADT Polynomial;

一元多项式相加链表示意图



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

谢谢！

```
void Union(List &La, List Lb) {  // 算法 2.1
  // 将所有在线性表 Lb 中但不在 La 中的数据元素插入到 La 中
  int La_len,Lb_len,i;
  ElemType e;
  La_len = ListLength(La);          // 求线性表的长度
  Lb_len = ListLength(Lb);
  for (i=1; i<=Lb_len; i++) {
    GetElem(Lb, i, e);              // 取 Lb 中第 i 个数据元素赋给 e
    if (!LocateElem(La, e, equal))  // La 中不存在和 e 相同的数据元素
      ListInsert(La, ++La_len, e);  // 插入
  }
} // union
```

```
void MergeList(List La, List Lb, List &Lc) {  // 算法 2.2
  // 已知线性表 La 和 Lb 中的元素按值非递减排列。归并 La 和 Lb
     得到新的线性表 Lc ， Lc 的元素也按值非递减排列。
  int i= j=1, k=0;
  InitList(Lc);   La_len = ListLength(La);      Lb_len = ListLength(Lb);
  while ((i <= La_len) && (j <= Lb_len)) {  // La 和 Lb 均非空
    GetElem(La, i, ai);     GetElem(Lb, j, bj);
    if (ai <= bj) {      ListInsert(Lc, ++k, ai);      ++i;    }
    else { ListInsert(Lc, ++k, bj); ++j; }
  }
  while (i <= La_len) { GetElem(La, i++, ai);  ListInsert(Lc, ++k, ai); }
  while (j <= Lb_len) { GetElem(Lb, j++, bj);  ListInsert(Lc, ++k, bj);}
} // MergeList
```

```
Status InitList_Sq(SqList &L) {  // 算法 2.3
  // 构造一个空的线性表 L 。
  L.elem=(ElemType*)malloc(LIST_INIT_SIZE*sizeof(ElemType));
  if (!L.elem) return ERROR;        // 存储分配失败
  L.length = 0;                     // 空表长度为 0
  L.listsize = LIST_INIT_SIZE;   // 初始存储容量
  return OK;
} // InitList_Sq
```

```
Status ListInsert_Sq(SqList &L, int i, ElemType e) {  // 算法 2.4
    //  在顺序线性表 L 的第 i 个元素之前插入新的元素 e，1≤i≤ListLength_Sq(L)+1
  ElemType *p;
  if (i < 1 || i > L.length+1) return ERROR;  // i 值不合法
  if (L.length >= L.listsize) {   // 当前存储空间已满，增加容量
    ElemType *newbase = (ElemType *)realloc(L.elem,
                (L.listsize+LISTINCREMENT)*sizeof (ElemType));
    if (!newbase) return ERROR;   // 存储分配失败
    L.elem = newbase;             // 新基址
    L.listsize += LISTINCREMENT;  // 增加存储容量
  }
  ElemType *q = &(L.elem[i-1]);   // q 为插入位置
  for (p = &(L.elem[L.length-1]); p>=q; --p) *(p+1) = *p;    // 元素右移
  *q = e;       // 插入 e
  ++L.length;   // 表长增 1
  return OK;
} // ListInsert_Sq
```

可以使用 malloc 吗

- 使用 **malloc**

```
ElemType *newbase = (ElemType *)malloc(
                (L.listsize+LISTINCREMENT)*sizeof (ElemType));
  if (!newbase) return ERROR;   //  存储分配失败
   for(i=0;i<L.listsize;i++)
            newbase[i]=L.elem[i];
   free(L.elem);
  L.elem = newbase;
```

```
Status ListDelete_Sq(SqList &L, int i, ElemType &e) {  // 算法 2.5
    //  在 L 中删除第 i 个元素，并用 e 返回其值。 1≤i≤ListLength_Sq(L) 。
  ElemType *p, *q;
  if (i<1 || i>L.length) return ERROR;  // i 值不合法
  p = &(L.elem[i-1]);                   // p 为被删除元素的位置
  e = *p;                               //  被删除元素的值赋给 e
  q = L.elem+L.length-1;                //  表尾元素的位置
  for (++p; p<=q; ++p) *(p-1) = *p;     //  被删除元素之后的元素左移
  --L.length;                           //  表长减 1
  return OK;
} // ListDelete_Sq
```

```
int LocateItem_Sq(SqList L, ElemType e,
        Status (*compare)(ElemType, ElemType)) {  //  算法 2.6
  //  在顺序线性表 L 中查找第 1 个值与 e 满足 compare() 的元素的
     位序。
  //  若找到，则返回其在 L 中的位序，否则返回 0 。
  int i;   ElemType *p;
  i = 1;        // i 的初值为第 1 个元素的位序
  p = L.elem;   // p 的初值为第 1 个元素的存储位置
  while (i <= L.length && !(*compare)(*p++, e))   ++i;
  if (i <= L.length) return i;
  else return 0;
} // LocateItem_Sq
```

```
void MergeList_Sq(SqList La, SqList Lb, SqList &Lc) {  //  算法 2.7
  //  已知顺序线性表 La 和 Lb 的元素按值非递减排列。归并得到新的顺序线性表 Lc 。
  ElemType *pa,*pb,*pc,*pa_last,*pb_last;
    pa = La.elem;    pb = Lb.elem;    Lc.listsize = Lc.length = La.length+Lb.length;
  pc = Lc.elem = (ElemType *)malloc(Lc.listsize*sizeof(ElemType));
  if (!Lc.elem)  exit(OVERFLOW);  //  存储分配失败
  pa_last = La.elem+La.length-1;
  pb_last = Lb.elem+Lb.length-1;
  while (pa <= pa_last && pb <= pb_last) {  //  归并
    if (*pa <= *pb) *pc++ = *pa++;
    else *pc++ = *pb++;
  }
  while (pa <= pa_last) *pc++ = *pa++;     //  插入 La 的剩余元素
  while (pb <= pb_last) *pc++ = *pb++;     //  插入 Lb 的剩余元素
} // MergeList
```

```
void exchang1( SqList &A, int m, int n ) {
 // 本算法实现顺序表中前 m 个元素和后 n 个元素的互换
   for (k = 1; k<=n; k++ ) {
     w = A.elem[m+k-1];
     for ( j=m+k-1; j>=k; j-- ) A.elem[j] = A.elem[j-1];
     A.elem[k-1] = w;
   } // for
} // exchang1   O(m*n)
```

假设已经完成 k-1 个元素移动，先考虑第 k 个元素

```
void invert( ElemType &R[] ， int s ， int t ){
 // 本算法将数组 R 中下标自 s 到 t 的元素逆置，即将 ( Rs, Rs+1, …, Rt-1, Rt ) 改变为 ( Rt, Rt-1, …, Rs+1, Rs )
   for ( k=s; k<=(s+t)/2; k++ ) {
      w = R[k]; R[k] = R[t-k+s];  R[t-k+s] = w;
    }//for
}// invert
void exchange2( SqList &A; int m; int n ) {
 // 本算法实现顺序表中前 m 个元素和后 n 个元素的互换
   invert( A.elem, 0, m+n-1 );
   invert( A.elem, 0, n-1 );
   invert( A.elem, n, m+n-1 );
} // exchange2
```

```
int ListLength_L( LinkList L )
{ // L 为带头结点的单链表的头指针
  // 本函数返回  L  所指链表的长度
    p=L->next;  k=0;
    while ( p ) {
      k++;  p=p->next;   // k  计非空结点数
    }//while
    return k;
} // ListLength_L
```

频度 n

```
Status GetElem_L(LinkList L,int i, ElemType &e) {  // 算法 2.8
  // L 为带头结点的单链表的头指针。
  // 当第 i 个元素存在时，其值赋给 e 并返回 OK ，否则返回 ERROR
  LinkList p;
  p = L->next;
  int j = 1;          // 初始化， p 指向第一个结点， j 为计数器
  while (p && j<i) { // 顺指针向后查找直到 p 指向第 i 个元素或 p 为空
    p = p->next;  ++j;
  }
  if ( !p || j>i ) return ERROR;  // 第 i 个元素不存在
  e = p->data;   // 取第 i 个元素
  return OK;
} // GetElem_L
```

频度 i-1

何时成立？

```
Status ListInsert_L(LinkList &L, int i, ElemType e) {  // 算法 2.9
  // 在带头结点的单链线性表 L 的第 i 个元素之前插入元素 e
  LinkList p,s;
  int j = 0;
  p = L;
while (p && j < i-1) {  // 寻找第 i-1 个结点
    p = p->next;    ++j;
  }
  if (!p || j > i-1) return ERROR;      // i 小于 1 或者大于表长
  s = (LinkList)malloc(sizeof(LNode));  // 生成新结点
  s->data = e;  s->next = p->next;      // 插入 L 中
  p->next = s;    return OK;
} // LinstInsert_L
```

对比 GetElem_L 为什么不从：j=1,p=L->next 开始？

i 为 1

频度 i-1

```
Status ListDelete_L(LinkList &L, int i, ElemType &e) {  // 算法 2.10
  // 在带头结点的单链线性表 L 中，删除第 i 个元素，并由 e 返回其值
  LinkList p,q;
  p = L;  int j = 0;
  while (p->next && j < i-1) {  // 寻找第 i 个结点，并令 p 指向其前趋
    p = p->next;     ++j;
  }
  if (!(p->next) || j > i-1) return ERROR;  // 删除位置不合理
  q = p->next;
  p->next = q->next;           // 删除并释放结点
  e = q->data;  free(q);   return OK;
} // ListDelete_L
```

对比 List_Insert_L 为何不是 p ？

删除最后一个结点

频度 i-1

```
void CreateList_L(LinkList &L, int n) {  // 算法 2.11
  // 逆位序输入 n 个元素的值，建立带表头结点的单链线性表 L
  LinkList p;   int i;
  L = (LinkList)malloc(sizeof(LNode));
  L->next = NULL;              // 先建立一个带头结点的单链表
  for (i=n; i>0; --i) {
    p = (LinkList)malloc(sizeof(LNode));  // 生成新结点
    scanf(&p->data);     // 改为一个随机生成的数字 (200 以内 )
    p->next = L->next;    L->next = p;    // 插入到表头
  }
} // CreateList_L
```

```
void InvertLinkedList( LinkList &L )
{   //  逆置头指针 L 所指链表
     p = L;  L = NULL;    //  设逆置后的链表的初态为空表
     while ( p ) {          //  p  为待逆置链表的头指针
       s = p;  p = p->next;
                          //  从   p  所指链表中删除第一个结点 (s
    结点 )
        s->next = L;  L = s;  //  将   s  结点插入到逆置表的表
    头
      }
} // InvertLinkedList
```

```
void MergeList_L(LinkList &La, LinkList &Lb, LinkList &Lc) {
  // 已知单链线性表 La 和 Lb 的元素按值非递减排列。 // 算法 2.12
  // 归并 La 和 Lb 得到新的单链线性表 Lc ， Lc 的元素也按值非递减
     排列。
  LinkList pa, pb, pc;
  pa=La->next; pb = Lb->next;Lc = pc = La;// Lc 的头结点 =La 头结点
  while (pa && pb) {
    if (pa->data <= pb->data) { pc->next = pa;   pc = pa;   pa = pa->next; }
    else { pc->next = pb;   pc = pb;   pb = pb->next; }
  }
  pc->next = pa ? pa : pb;  // 插入剩余段
  free(Lb);                 // 释放 Lb 的头结点
} // MergeList_L
```

```
int LocateElem_SL(SLinkList S, ElemType e) {//  算法 2.13
  //  在静态单链线性表 L 中查找第 1 个值为 e 的元素。
  //  若找到，则返回它在 L 中的位序，否则返回 0 。
  int i;  i = S[0].cur;    // i 指示表中第一个结点
  while (i && S[i].data != e) i = S[i].cur;  //  在表中顺链查找
  return i;
} // LocateElem_SL
void InitSpace_SL(SLinkList space) {  //  算法 2.14
 //  将一维数组 space 中各分量链成一个备用链表
 // space[0].cur 为头指针 , "0" 表示空指针
    for (int i=0; i<MAXSIZE-1; ++i)
    space[i].cur = i+1;  // 前 space[0..MAXSIZE-2].cur 指向后一个
    space[MAXSIZE-1].cur = 0;
} // InitSpace_SL
```

增加一个头位置 int s

i=S[s].cu

```
int Malloc_SL(SLinkList &space) {
// 若备用空间链表非空，则返回分配的结点下标，否则返回 0
  int i = space[0].cur;
  if (space[0].cur) space[0].cur = space[space[0].cur].cur;
  return i;
} // Malloc_SL
void Free_SL(SLinkList &space, int k) {
  // 将下标为 k 的空闲结点回收到备用链表
  space[k].cur = space[0].cur;    space[0].cur = k;
} // Free_SL
```

```
void difference(SLinkList &space, int &S) {
  // 依次输入集合 A 和 B 的元素，在一维数组 space 中建立表示集合 (A-B)∪(B-A)
     的静态链表，S 为头'指针'。假设备用空间足够大， space[0].cur 为头指针。
  InitSpace_SL(space); S = Malloc_SL(space); r = S; scanf(m,n); //r 指向 S 最后结点
   for (j=1; j<=m; ++j) { i =Malloc_SL(space);scanf(space[i].data);space[r].cur = i;  r =
      i; }
  space[r].cur = 0;              // 尾结点的指针为空
  for (j=1; j<=n; ++j) {    // 依次输入 B 的元素，若不在当前表中，则插入，否则删除
    scanf(b); p = S;   k = space[S].cur;  // k 指向集合 A 中第一个结点
    while (k!=space[r].cur && space[k].data!=b) {  p = k;    k = space[k].cur; }//p 跟随 k
    if (k == space[r].cur) { //  当前表中不存在该元素，插入在 r 所指结点之后， r 不动
      i = Malloc_SL(space);   space[i].data = b;
      space[i].cur = space[r].cur;    space[r].cur = i; // 插在 r 后第一个元素前
    } else {//  该元素已在表中，删除之
      space[p].cur = space[k].cur;      Free_SL(space, k); if (r == k)  r = p;
    //  若删除的是尾元素，则需修改尾指针
    }//else
  }//for
} // difference
```

P 作用？

两个带头节点链表：

可用空间 头指针 space

数据链表 头指针 S ，尾指针 r

```
void union_OL( LinkList &La ， LinkList &Lb ) {
    // La 和 Lb 分别为表示集合 A 和 B 的循环链表的尾指针，求 C=A∪B ，操作
    // 完成之后， La 为表示集合 C 的循环链表的尾指针，集合 A 和 B 的链表不再存在
    pa = La->next->next; pb = Lb->next->next; rc = La->next; // pa 指向 A 中当前考察的
    结点 ,rc 指向 C 当前的表尾
    while ( pa!=La->next && pb!=Lb->next ) {
      if ( pa->data < pb->data ) {rc->next = pa;  rc = pa;  pa = pa->next; }
      else if ( pa->data > pb->data ){ rc->next = pb;  rc = pb;  pb = pb->next; }
      else {   // 链接 A 的元素，释放 B 的结点， pa 、 pb 分别指向各自下一元素
        rc->next = pa;  rc = pa;  pa = pa->next;
        qb = pb;  pb = pb->next;  delete qb;
      }//else
    }//while
    if ( pb == Lb->next )  rc->next = pa;   // 链接 A 的剩余段
    else {   // 链接 B 的剩余段
      rc->next = pb;  pb = Lb->next;   // pb 指向 B 的头结点
      Lb->next = La->next;  La = Lb;   // 构成 C 的循环链
    }//else
    free(pb);   // 释放 B 表的表头
} // union_OL
```

```
Status ListInsert_DuL(DuLinkList &L, int i, ElemType e) {
  // 在带头结点的双链循环线性表 L 的第 i 个元素之前插入元素 e ，
  // i 的合法值为 1≤i≤ 表长 +1 。
if (!(p = GetElemP_DuL(L, i)))  // 在 L 中确定第 i 个元素的位置指
     针 p
    return ERROR;                    // p=NULL，即第 i 个元素不存在
  if (!(s = (DuLinkList)malloc(sizeof(DuLNode))))
    return ERROR;
  s->data = e;
  s->prior = p->prior;
  p->prior->next = s;
  s->next = p;
  p->prior = s;
  return OK;
} // ListInsert_DuL
```

```
Status ListDelete_DuL(DuLinkList &L, int i, ElemType &e) {
  //  删除带头结点的双链循环线性表 L 的第 i 个元素 ， i 的合法值
     为 1≤i≤ 表长
  DuLinkList p;
  if (!(p = GetElemP_DuL(L, i)))  //  在 L 中确定第 i 个元素位置指针
     p
    return ERROR;                  // p=NULL，即第 i 个元素不存在
  e = p->data;
  p->prior->next = p->next;
  p->next->prior = p->prior;
  free(p);
  return OK;
} // ListDelete_DuL
```

```
Status MergeList_L2(NLinkList &La, NLinkList &Lb, NLinkList &Lc,
                    int (*compare)(ElemType, ElemType)) {  // 算法 2.21
  // 已知单链线性表 La 和 Lb 的元素按值非递减排列。
  // 归并 La 和 Lb 得到新的单链线性表 Lc ，Lc 的元素也按值非递减排列。
if (!InitList(Lc)) return ERROR;  // 存储空间分配失败
ha = GetHead(La); hb = GetHead(Lb); // ha 和 hb 分别指向 La 和 Lb 的头结点
pa = NextPos(La, ha); pb = NextPos(Lb, hb); // pa 和 pb 分别指向 La 和 Lb 中当前结点
while (pa && pb) {     // La 和 Lb 均非空
    a = GetCurElem(pa);  b = GetCurElem(pb);
    if ((*compare)(a, b)<=0) {  // a≤b
      DelFirst(ha, q);  Append(Lc, q);  pa = NextPos(La, pa);
    } else {DelFirst(hb, q);  Append(Lc, q);  pb = NextPos(Lb, pb);   }
  } // while
  if (pa) Append(Lc, pa);        //  链接 La 中剩余结点
  else Append(Lc, pb);           //  链接 Lb 中剩余结点
  FreeNode(ha);   FreeNode(hb);  //  释放 La 和 Lb 的头结点
  return OK;
} // MergeList_L2
```

```
void CreatPolyn(PLinkList &P, int m) {  //  算法 2.22
  //  输入 m 项的系数和指数，建立表示一元多项式的有序链表 P
  InitList(P);   h = GetHead(P);
  e.coef = 0.0;  e.expn = -1;  SetCurElem(h, e);       //  设置头结点
  for (i=1; i<=m; ++i) {  //  依次输入 m 个非零项
    scanf ("%f,%d\n",&e.coef, &e.expn);
    if (!LocateElem(P, e,q, cmp)) { //  当前链表中不存在该指数项
      if (MakeNode(s,e)) InsFirst(q, s);  //  生成结点并插入链表
    } else i--;  //   如果没有产生插入，则将 i 值减 1
  }
} // CreatPolyn
```

q 指向小于 e 的最大值结点

```
void AddPolyn(PLinkList &Pa, PLinkList &Pb) {
// 多项式加法：Pa = Pa＋Pb，利用两个多项式的结点构成"和多项式"。
ha = GetHead(Pa); hb = GetHead(Pb); qa = NextPos(Pa,ha); qb = NextPos(Pb,hb);
 while (qa && qb) {    // Pa和Pb均非空
  a = GetCurElem (qa); b = GetCurElem (qb);// a和b为两表中当前比较元素
switch (*cmp(a,b)) {
   case -1: // 多项式PA中当前结点的指数值小
     ha = qa;        qa = NextPos (Pa, qa);        break;
   case 0:  // 两者的指数值相等
     sum = a.coef + b.coef ;
     if (sum != 0.0) {  // 修改多项式PA中当前结点的系数值
       temp.coef=sum;  temp.expn=a.expn;  SetCurElem(qa, temp) ; ha = qa;}
     else {DelFirst(ha, qa); FreeNode(qa); } // 删除多项式PA中当前结点
     DelFirst(hb, qb); FreeNode(qb); qb = NextPos(Pb, hb); qa = NextPos(Pa, ha); break;
   case 1:  // 多项式PB中当前结点的指数值小
     DelFirst(hb, qb); InsFirst(ha, qb);  qb = NextPos(Pb, hb);
     ha = NextPos(Pa, ha);   break;
  } // switch
 } // while
 if (!Empty(Pb)) Append(Pa, qb);   FreeNode(hb);
} // AddPolyn
```

ha 表示已处理的最后一个结点？

cmp 如何定义？Cmp 未传入，只能当函数

ha 不断移动，结果集最后一个节点， hb 不动，其后的元素不断被删除，插入 Pa 中。 qa qb 是当前处理元素

第三章 栈和队列

3.1 栈

3.2 栈的应用

*3.3 栈与递归的实现

3.4 队列

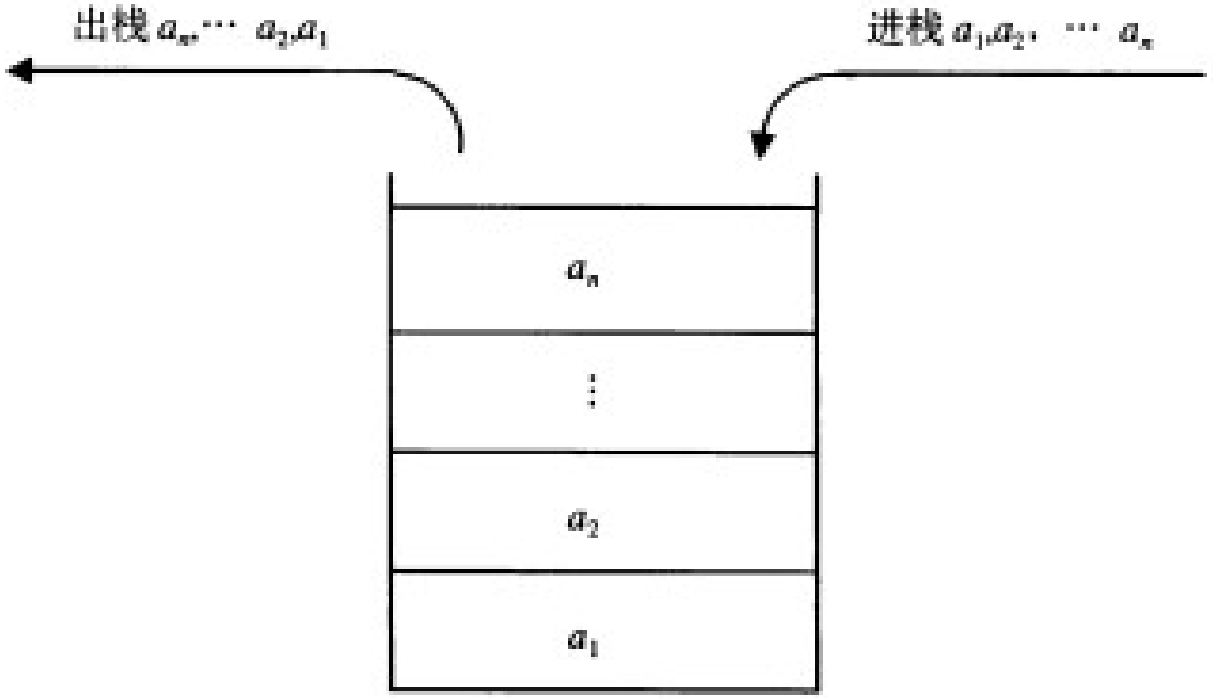
*3.5 离散事件模拟

3.1 栈

3.1.1 抽象数据类型栈的定义

- 定义：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)



栈的 ADT 描述

ADT Stack{

数据对象 :D={a_i|ai∈ElemSet,i=1,2,…n,n≥0}

数据关系 :R1={<a_{i-1},a_i>| a_{i-1},a_i ∈D,i=2,3,…n}

基本操作 :

InitStack （ &S ）

DestroyStack(&S)

ClearStack(&S)

StackEmpty （ S ）

GetTop(S,&e)

Push(&S, e)

Pop(&S,&e)

StackTraverse(S)

}ADT Stack

StackLength （ S ）

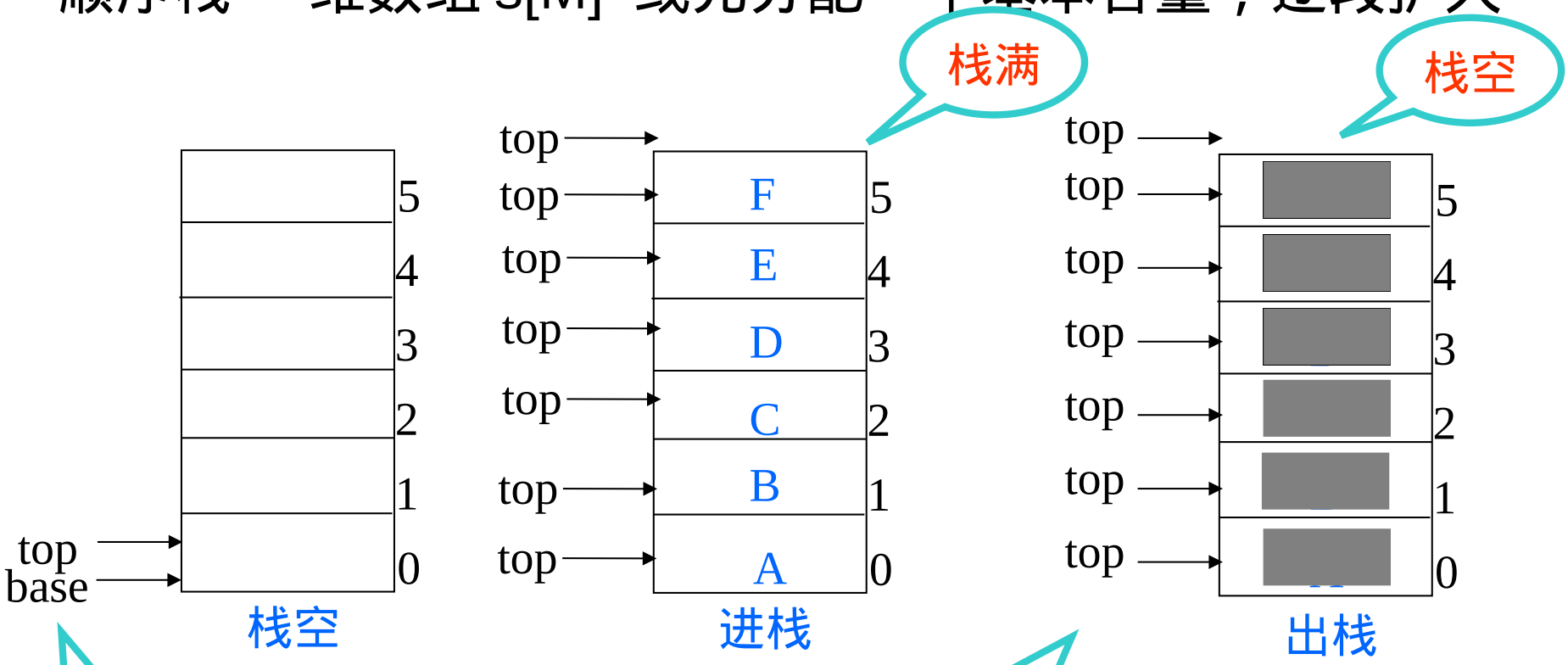
3.1.2 栈的表示和实现

- 顺序栈

```
#define STACK_INIT_SIZE 100
#define STACKINCREMENT 10
Typedef struct{
    SElemtype          *base;
    SElemtype          *top;
    int                stacksize;
}SqStack;
```

– 栈的表示和实现

顺序栈 一维数组 s[M] 或先分配一个基本容量，逐段扩大



栈顶指针 top, 指向实际栈顶后的空位置，初值为 base. base 保持不变

设数组维数为 M

top==base 栈空，此时出栈，则下溢（ underflow)

top-base==M, 栈满，此时入栈，则上溢（ overflow)

相关的操作实现

```
Status InitStack_Sq(SqStack &S) {
    S.base=(SElemType *)malloc(STACK_INIT_SIZE*sizeof(SElemType));
    if(!S.base)exit(OVERFLOW);
    S.top=S.base;
    S.stacksize=STACK_INIT_SIZE;        return OK;
} //InitStack_Sq
Status GetTop_Sq(SqStack S,SelemType &e){
    if(S.top==S.base)return ERROR;
    e=*(S.top-1);return OK;
}
Status Push_Sq(SqStack &S ,SelemType e) {
…    *S.top++=e;    …
}
Status Pop_Sq(SqStack &S ,SelemType &e){
…    e=*--S.top;    …
}
```

栈满、栈空？

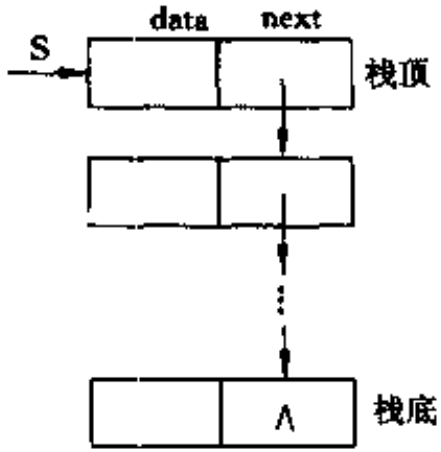
- 链栈

typedef LinkList LinkStack;

InitStack_L(LinkStack &S)

Push_L(LinkStack &S ,SelemType e)

Pop_L(LinkStack &S ,SelemType &e)




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

3.2 栈的应用

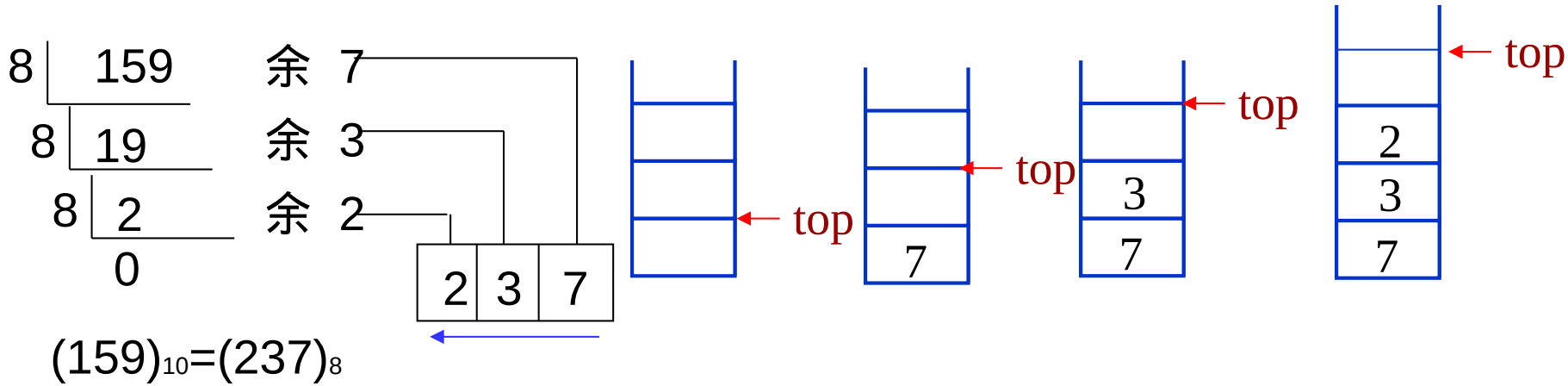
1. 数制转换

$$N=a_n.d^n+a_{n-1}.d^{n-1}+\ldots+a_1.d+a_0$$

计算过程 - 入栈

打印过程 - 出栈

例 把十进制数 159 转换成八进制数



$(159)_{10}=(237)_8$

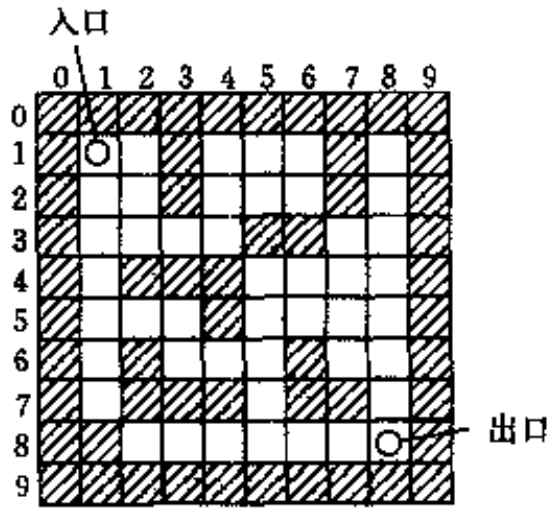
3.2.2 括号匹配检验

{[()]} 等括号的匹配 *

3.2.3 行编辑程序

– 算法 void LineEdit()

3.2.4 迷宫求解



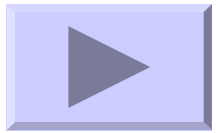
步序，坐标，方向

```
typedef struct{int  r,c;} PosType;
typedef struct{int ord;Postype seat;int di} SElemType;
Typedef int MazeType[MAXR][MAXC];
Status Pass(MazeType maze, PosType CurPos);
void FootPrint(MazeType &maze, PosType CurPos);
void MarkPrint(MazeType &maze, PosType CurPos);
PosType NextPos(PosType CurPos, int Dir);
```

```
Status MazePath(MazeType &maze, PosType start,
    PosType end) ;
```

3.2.5 表达式求值

- 前缀表达式、中缀表达式、后缀表达式
- 波兰式、逆波兰式
- 算法 3.4 void evaluateExpression()



使用两个栈 :OPTR, 运算符栈； OPND 操作数栈

步骤：

1 ）首先置 OPND 空， OPTR 栈底为’ #’;

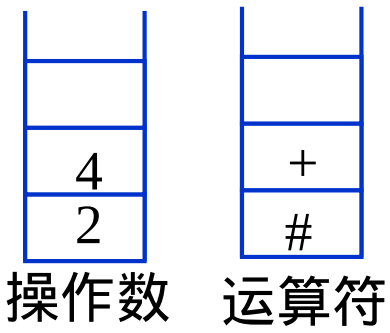
2 ）依次读入表达式中每个字符，若是操作数则进 OPND 栈，若是运算符，则和 OPTR 的栈顶运算符比较优先权后做相应操作，直至整个表达式求值完毕（ OPTR 的栈顶和当前读入元素均为’ #’ ）。

- θ1 表示栈内， θ2 表示当前运算符
- 运算符优先级相等时取 “ >”
- “(“=“)” “#”=“#“ ，特殊处理，后者比较不会出现

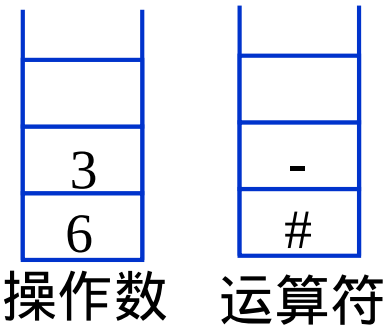
表 3.1 算符间的优先关系

θ_1 \ θ_2	+	−	*	/	(	)	#
+	>	>	<	<	<	>	>
−	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

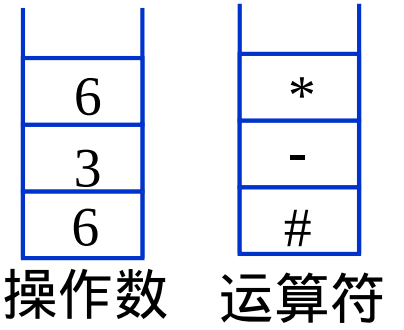
例　计算 2+4-3*6#



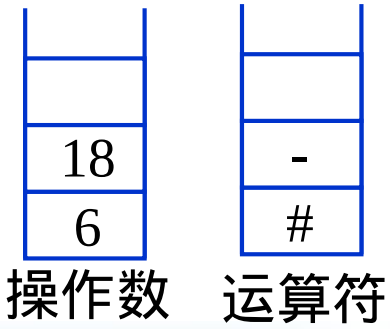
“+” > “ _”



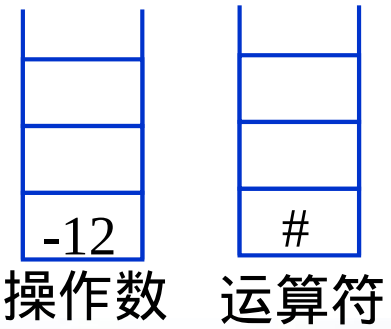
“_” < “*”



“*” > “#”



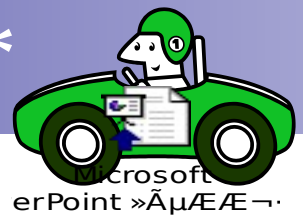
“_” > “#”



栈顶和 c 均为“ #”

循环结束

原表达式转化为后缀表达式规则 *

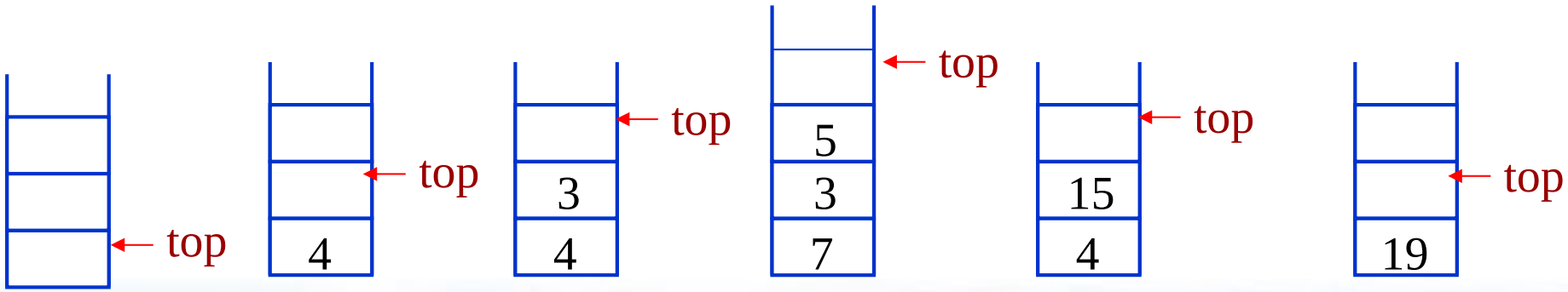


1) 设立运算符栈
2) 设表达式的结束符为“ #” 预设栈底为“ #”
3) 若当前字符是操作数，直接发送后缀表达式
4) 若当前字符是运算符，且优先级大于栈顶运算符，则入栈，否则退出栈顶运算符发送给后缀表达式。
5) 若当前字符是结束符“ #”，则自栈顶至栈底依次将栈中所有运算符发送给后缀表达式。
6) 若当前运算符是“ (”，进栈，对其之前底运算符起隔离作用。
7) 若当前运算符是“)”，可视为自相应的“ (” 开始的表达式的结束，从栈顶起依次退出栈顶运算符发送给后缀表达式，直至栈顶相应运算符为“ (”，再将“ (” 也出栈

后缀表达式求值步骤

1、读入表达式一个字符
2、若是操作数，压入栈，转 4
3、若是运算符，从栈中弹出 2 个数，将运算结果再压入栈
4、若表达式输入完毕，栈顶即表达式值；
　　若表达式未输入完，转 1

例　计算 4+3*5　后缀表达式：435*+

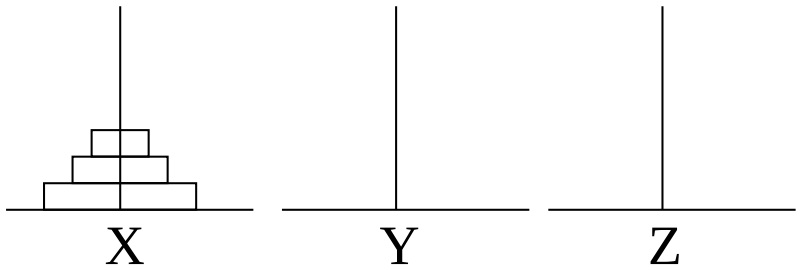


3.3* 栈与递归的实现

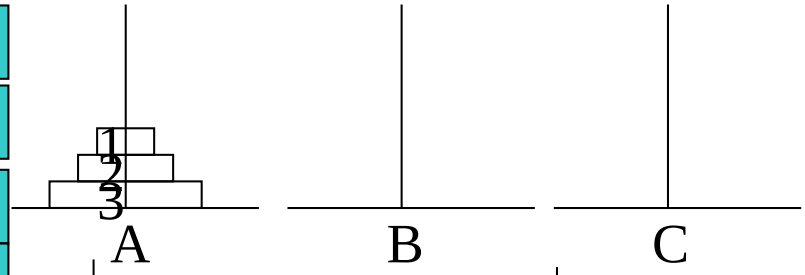
汉诺塔问题：

– 有三个塔座 X，Y，Z，X 座上插有 n 个直径大小各不相同编号为 1...n 的圆盘。现在要求将 X 上的圆盘移动到 Z 塔座上。要求遵循以下规则：

- 每次只能移动一个圆盘
- 圆盘可以插在 X，Y，Z 中任一个上
- 任何时刻都不能将一个大盘压在小盘上



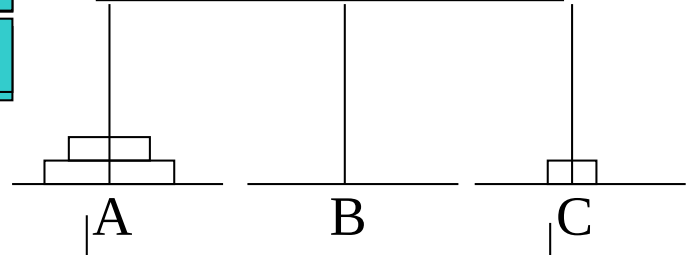
```
main()
  {   int m;
      printf("Input number of disks”);
      scanf("%d",&m);
      printf(”Steps : %3d disks”,m);
      hanoi(m,'A','B','C');
(0) }
void hanoi(int n,char x,char y,char z)
(1)  {
(2)     if(n==1)
(3)         move(x,1,z);
(4)     else{
(5)             hanoi(n-1,x,z,y);
(6)             move(x,n,z);
(7)             hanoi(n-1,y,x,z);
(8)         }
(9)  }
```



3	A	B	C	0

2	A	C	B	6
3	A	B	C	0

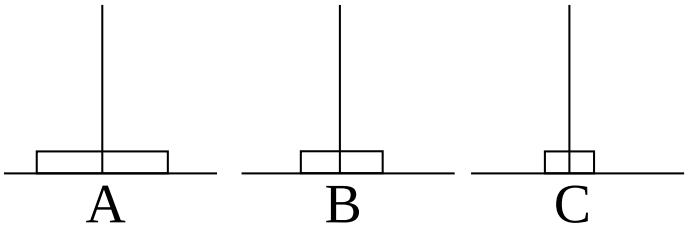
1	A	B	C	6
2	A	C	B	6
3	A	B	C	0



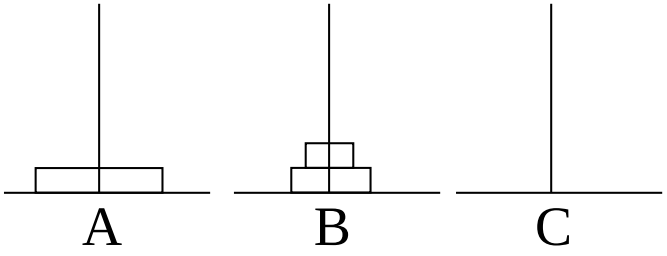
2	A	C	B	6
3	A	B	C	0

```
main()
  {   int m;
      printf("Input the number of disks”);
      scanf("%d",&m);
      printf(" Steps : %3d disks”,m);
      hanoi(m,'A','B','C');
(0) }
void hanoi(int n,char x,char y,char z)
(1)  {
(2)     if(n==1)
(3)         move(x,1,z);
(4)     else{
(5)             hanoi(n-1,x,z,y);
(6)             move(x,n,z);
(7)             hanoi(n-1,y,x,z);
(8)         }
(9)  }
```

2	A	C	B	6
3	A	B	C	0



1	C	A	B	8
2	A	C	B	6
3	A	B	C	0

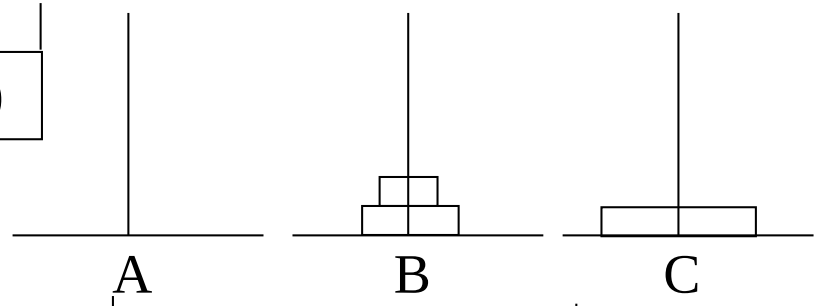


2	A	C	B	6
3	A	B	C	0

3	A	B	C	0

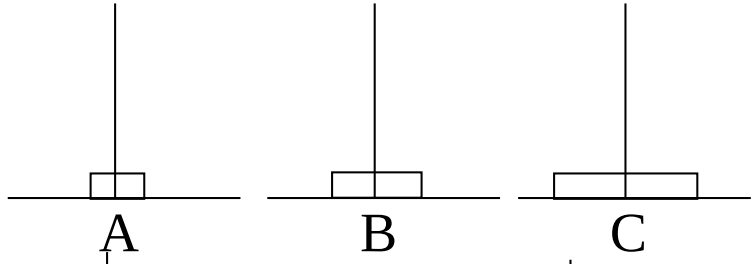
```
main()
  {   int m;
      printf("Input the number of disks");
      scanf("%d",&m);
      printf(" Steps : %3d disks",m);
      hanoi(m,'A','B','C');
(0) }
void hanoi(int n,char x,char y,char z)
(1)  {
(2)     if(n==1)
(3)          move(x,1,z);
(4)     else{
(5)               hanoi(n-1,x,z,y);
(6)               move(x,n,z);
(7)               hanoi(n-1,y,x,z);
(8)          }
(9)  }
```

3	A	B	C	0



2	B	A	C	8
3	A	B	C	0

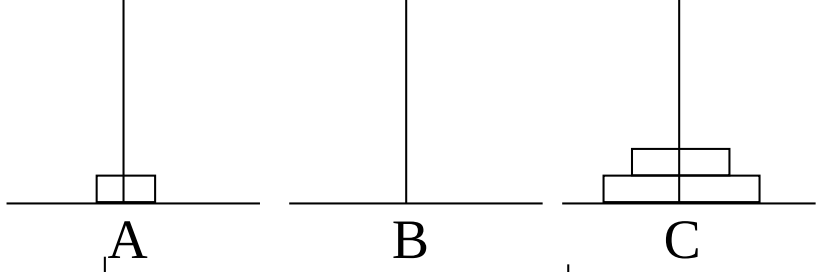
1	B	C	A	6
2	B	A	C	8
3	A	B	C	0



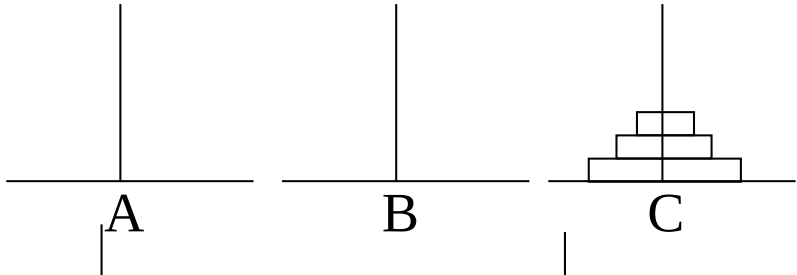
2	B	A	C	8
3	A	B	C	0

```
main()
  {   int m;
      printf("Input the number of disks”);
      scanf("%d",&m);
      printf(" Steps : %3d disks”,m);
      hanoi(m,'A','B','C');
(0) }
void hanoi(int n,char x,char y,char z)
(1)  {
(2)     if(n==1)
(3)          move(x,1,z);
(4)     else{
(5)               hanoi(n-1,x,z,y);
(6)               move(x,n,z);
(7)               hanoi(n-1,y,x,z);
(8)          }
(9)  }
```

2	B	A	C	8
3	A	B	C	0



1	A	B	C	8
2	B	A	C	8
3	A	B	C	0



2	B	A	C	8
3	A	B	C	0

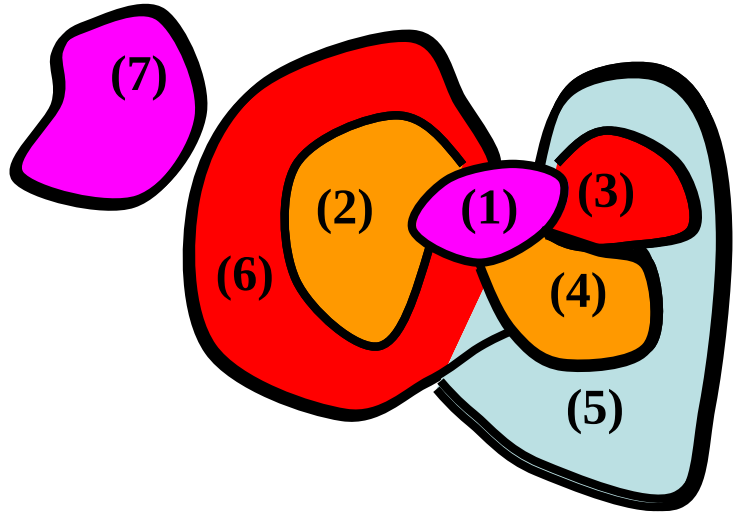
3	A	B	C	0

栈空

- 地图四染色问题 *

R [7][7]

	1	2	3	4	5	6	7
1	0	1	1	1	1	1	0
2	1	0	0	0	0	1	0
3	1	0	0	1	1	0	0
4	1	0	1	0	1	1	0
5	1	0	1	1	0	1	0
6	1	1	0	1	1	0	0
7	0	0	0	0	0	0	0



1# 紫色

2# 黄色

3# 红色

4# 蓝色

1	2	3	4	5	6	7
1	2	3	2	4	3	1

3.4 队列

- 定义：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
- 3.4.1 抽象数据类型队列的定义

队头 (front) 、队尾 (rear) ，先入先出 (FIFO)

ADT Queue{

数据对象 :D={a_i|a_i∈ElemSet,i=1,2,…n,n≥0}

数据关系 :R={<a_{i-1},a_i>| a_{i-1} , a_i∈D,i=2,…n}

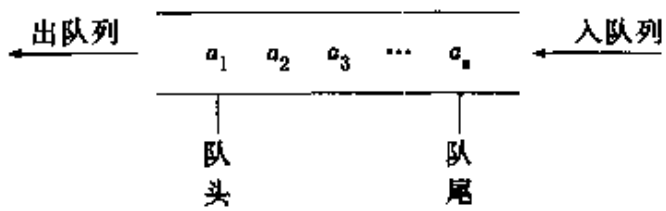
基本操作 :

InitQueue(&Q) ;

操作结果 : 构造一个空队列；

DestroyQueuek(&Q) ；

初始条件 : 队列 Q 已存在　　操作结果 : 销毁队列；



ClearQueue(&Q) ；

初始条件 : 队列 Q 已存在　　操作结果 : 清空队列；

QueueEmpty(Q) ；

初始条件 : 队列 Q 已存在　　操作结果 : 判断队列是否为空；

QueueLength(Q) ；

初始条件 : 队列 Q 已存在　　操作结果 : 返回队列长度；

GetHead(Q,&e) ;　　操作结果 : 返回队头元素给 e ；

EnQueue(&Q,e) ;　　操作结果 : 插入 e 为队尾元素；

DeQueue(&Q,&e) ;　　操作结果 : 删除队头元素给 e ；

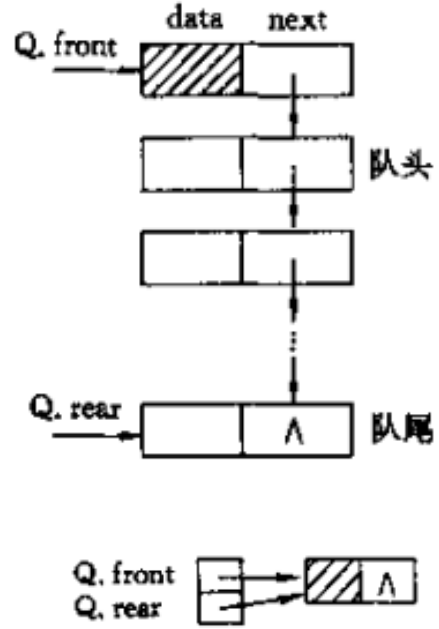
QueueTraverse(Q) ;　　操作结果 : 遍历队列元素

} ADT Queue;

3.4.2 链队列 -- 队列的链式表示和实现

- 表示结构：带头结点的单链表

```
typedef Linklist Queueptr;
typedef struct {
        Queueptr front; // 指向头结点
        Queueptr rear;  // 指向队尾
}LinkQueue;
```



- 操作实现

```
Status  InitQueue_L(LinkQueue &Q);
Status DestroyQueue_L(LinkQueue &Q);
Status EnQueue_L(LinkQueue &Q,QElemtype e);
Status DeQueue_L(LinkQueue &Q,QElemtype &e);
```

注意：删除元素时为最后一个元素时应改写 rear 指针。

3.4.2 循环队列 -- 队列的顺序表示和实现

```
#define QUEUE_INIT_SIZE 100
typedef struct {
QElemtype      *base;
int        front,rear;
int        qsize;
}SqQueue;
```

–约定：

空队列 front==rear; 插入 rear++ ; 删除 front++

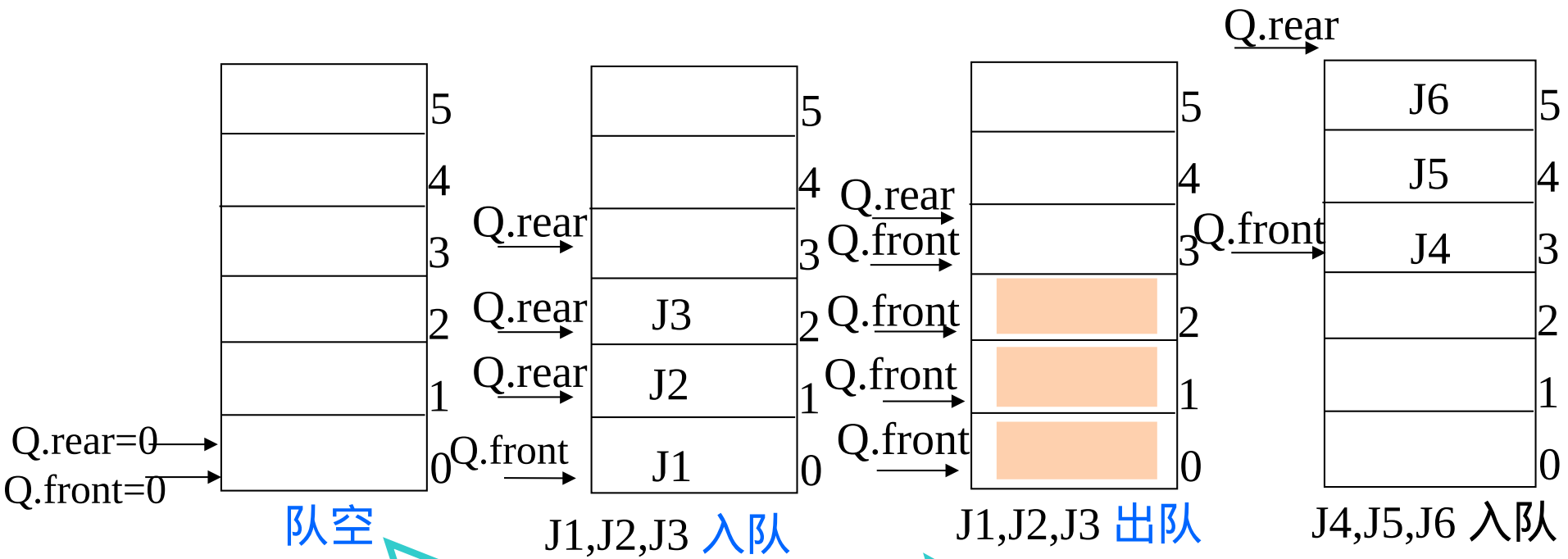
–操作：EnQueue,DeQueue,GetHead,QueueLength

–空队列判断

不可用用首尾指针相等来判断队列的空

解决办法： 1 ：增加标志位 2 ：少用一个元素

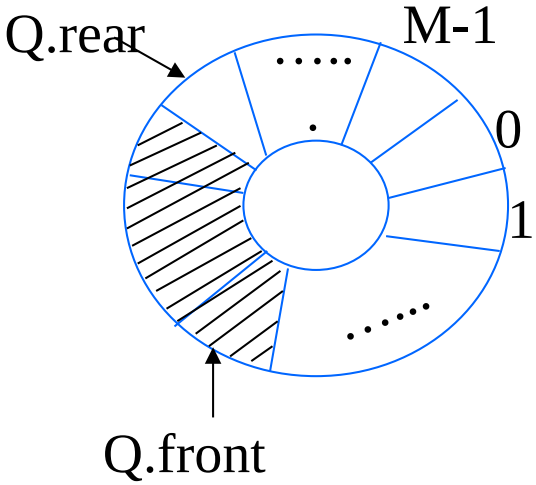
用一维数组实现 sq[M]



设两个指针 Q.front,Q.rear, 约定：
Q.rear 指示队尾元素下一位置；
Q.front 指示队头元素
初值 Q.front=Q.rear=0

空队列条件： Q.front = = Q.rear
入队列： Q.base[Q.rear++]=x;
出队列： x=Q.base[Q.front++];

- 存在问题
  - 设数组大小为 **M** ，则：
  - 当 **Q.front=0** ， **Q.rear=M** 时，再有元素入队发生溢出——真溢出
  - 当 **Q.front≠0** ， **Q.rear=M** 时，再有元素入队发生溢出——假溢出
- 解决方案
  - 队首固定，每次出队剩余元素向前移动——浪费时间
  - 循环队列
    - 基本思想：把队列设想成环形，让 **sq[0]** 接在 **sq[M-1]** 之后，若 **Q.rear+1==M** ，则令 **Q.rear=0;**

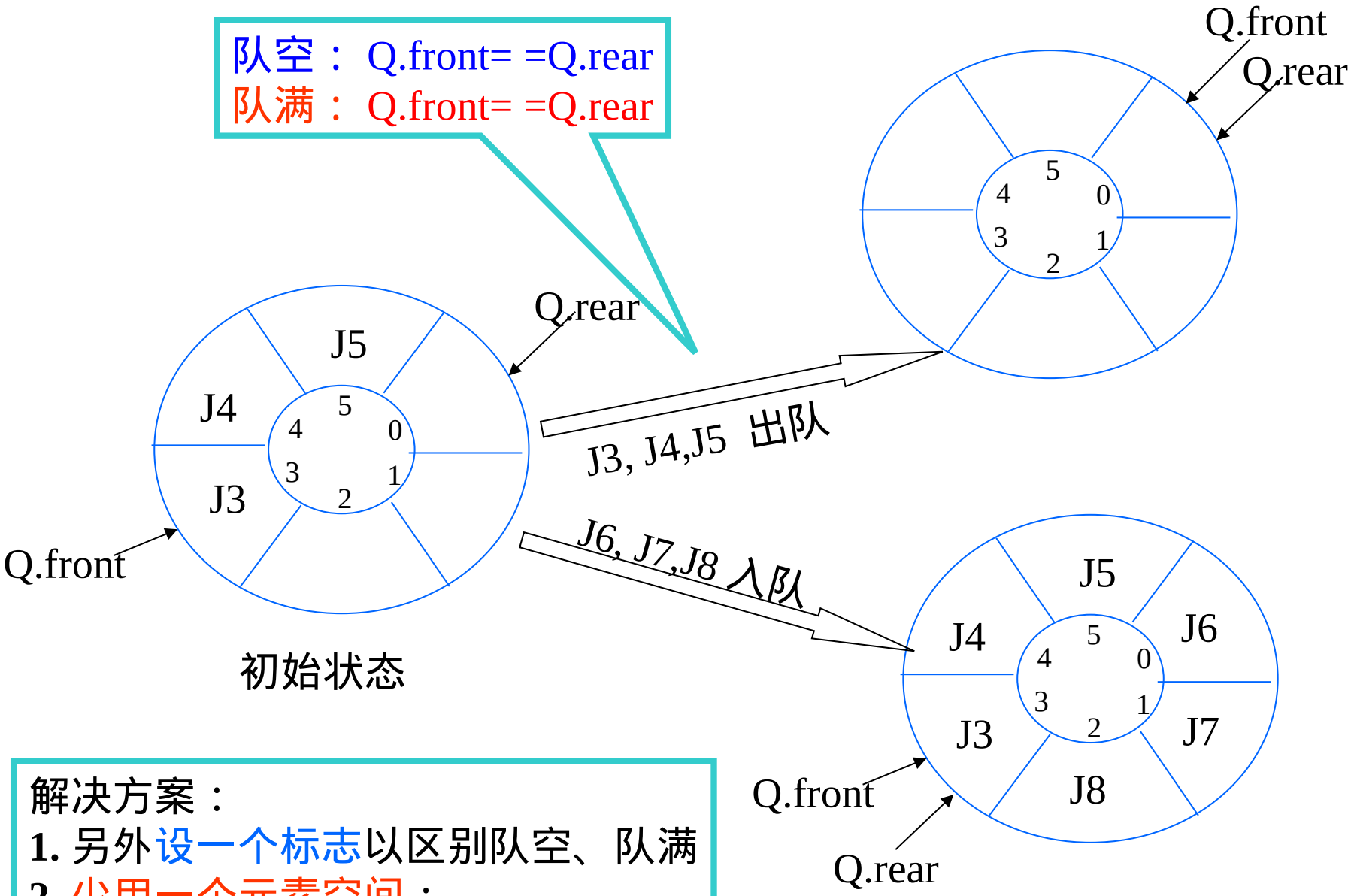


实现：利用“模”运算

入队： Q.base[Q.rear]=x; Q.rear=(Q.rear+1)%M;

出队： x=Q.base[Q.front]; Q.front=(Q.front+1)%M;

队满、队空判定条件



解决方案：

1. 另外设一个标志以区别队空、队满
2. 少用一个元素空间：

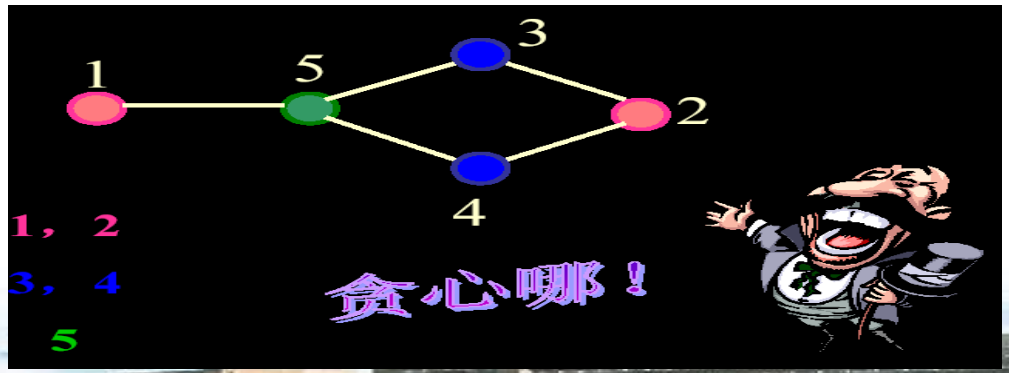
队空： **Q.front= =Q.rear**

队满： **(Q.rear+1)%M= =Q.front**

* 队列应用

- 例打印二项式系数 *
  - void yanghui(int n)
  - 注意：最后一行单独处理

- 例　为比赛划分无冲突子集 *
  - void Division(int R[][],int n,int result)
  - 贪婪法，不是全局最优解
  - 类似地图着色问题

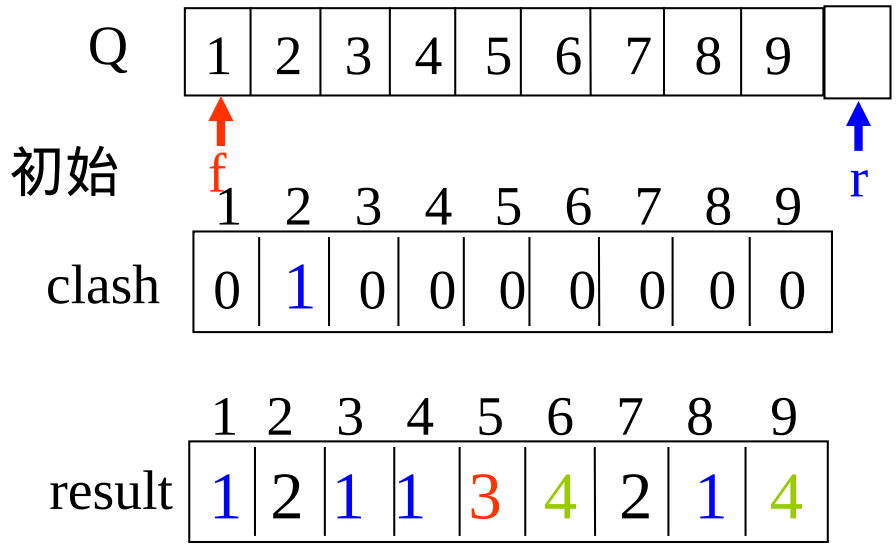


划分无冲突子集算法描述

R={ (2,8), (9,4), (2,9), (2,1), (2,5), (6,2), (5,9),
(5,6), (5,4), (7,5), (7,6), (3,7), (6,3) }

R=

	1	2	3	4	5	6	7	8	9
1	0	1	0	0	0	0	0	0	0
2	1	0	0	0	1	1	0	1	1
3	0	0	0	0	0	1	1	0	0
4	0	0	0	0	1	0	0	0	1
5	0	1	0	1	0	1	1	0	1
6	0	1	1	0	1	0	1	0	0
7	0	0	1	0	1	1	0	0	0
8	0	1	0	0	0	0	0	0	0
9	0	1	0	1	1	0	0	0	0



可行的子集划分为：

A1={ 1,3,4,8 }

A2={ 2,7 }

A3={ 5 }

A4={ 6,9 }

谢　　谢

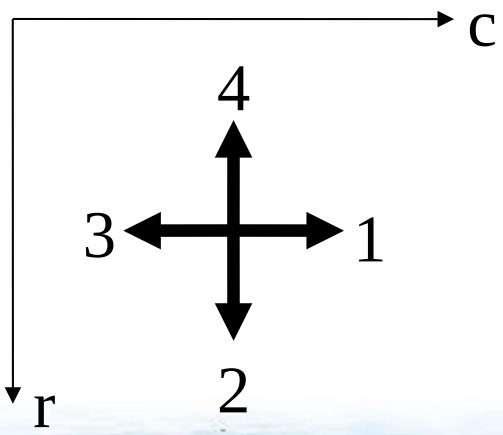
```
void conversion (int Num) {
  // 对于输入的任意一个非负十进制整数，打印输出与其等值的八进制数
    ElemType e;
    SqStack S;
    InitStack(S);      // 构造空栈
    while (Num) {
        Push(S, Num % 8);
        Num = Num/8;
    }
    while (!StackEmpty(S)) {
        Pop(S,e);
        printf ("%d", e);
    }
    printf("\n");
} // conversion
```

括号匹配 *

```
bool matching(char exp[] ) {     // 检验表达式中所含括弧是否正确嵌套，若是，则返回 TRUE ，否则返回 FALSE.  '#' 为表达式的结束符
    int state = 1;    ch = *exp++;
    while (ch != '#' && state) {
      switch of ch {
        case '('||'[': { Push(S, ch);  break; } //  凡左括弧一律入栈
        case ')':
    if ( !StackEmpty(S) && GetTop(S)== '(' )Pop(S,e);else  state = 0 ;
    break;
        case ']':
                  if ( !StackEmpty(S) && GetTop(S) =='[‘ )Pop(S,e);else  state = 0 ;
    break;
       } //switch
                 ch = *exp++;
     } // while
                  if (state && StackEmpty(S) ) return TRUE;  else  return FALSE;
}//matching
```

```
void LineEdit() {// 利用字符栈 S ，从终端接收一行并传送至调用过程的数据区。
InitStack(S);       // 构造空栈 S
  ch = getchar();     // 从终端接收第一个字符
  while (ch != EOF) { //EOF 为全文结束符
    while (ch != EOF && ch != '\n') {
      switch (ch) {
        case '#': Pop(S, ch);    break;  //  仅当栈非空时退栈
        case '@': ClearStack(S); break;  //  重置 S 为空栈
        default : Push(S, ch);   break;  //  有效字符进栈 , 未考虑栈满情形
      }
      ch = getchar();  //  从终端接收下一个字符
    }//while
     StackTraverse(S); // 将从栈底到栈顶的栈内字符传送至调用过程的数据区
     ClearStack(S);  // 重置 S 为空栈
     if (ch != EOF) ch = getchar();
  }//while
  DestroyStack(S);
}
```

```
void FootPrint(MazeType &maze,PosType &p)
      {maze[p.r,p.c]='*';} // 留下足迹
void MarkPrint(MazeType &maze,PosType &p)
      {maze[p.r,p.c]='X';} // 不能通行
bool Pass(MazeType &maze,PosType &p)){
     return(maze[p.r,p.c]=='0')}  // 标记' 0' 为通
PosType NextPos(PosType c,int d){
    n.c=c.c+((d==1)?1:(d==3)?-1:0);
    n.r=c.r+((d==2)?1:(d==4)?-1:0);
    return n;
}
```



迷宫问题

```
Status MazePath(MazeType &maze, PosType start, PosType end) {
curpos = start; curstep = 1;  // 设定 " 当前位置 " 为 " 入口位置“    探索第一步
  do {
    if (Pass(maze,curpos)){// 当前位置可通过，未曾走到过，非回溯
      FootPrint(maze,curpos); e=(curstep,curpos,1);    Push(S,e);
     if (curpos.r == end.r && curpos.c==end.c) return (TRUE); //  到达终点
      curpos = NextPos(curpos, 1); curstep++;                 //  探索下一步
    } else {  //  当前位置不能通过
      if (!StackEmpty(S)) {   Pop(S,e);
            while (e.di==4 && !StackEmpty(S))
     Pop(S,e);}
        if (e.di<4) {  e.di++; Push(S, e); curpos = NextPos(e.seat, e.di);
     curstep=e.curstep+1; }
       } // if
    } // else
  } while (!StackEmpty(S) ); //do
  return FALSE;
} // MazePath
```

int(maze,e.seat);

回溯

```
float EvaluateExpression(char* MyExpression) {
  InitStack (OPTR);    Push (OPTR, '#');
  InitStack (OPND);      c = MyExpression;
  while (*c!= '#' || GetTop(OPTR)!= '#') {
    if (!In(*c, OPSET)) {Push(OPND, *c); c++;}
    else //  不是运算符则进栈
        switch (precede(GetTop(OPTR), *c)) {
          case '<':   //  栈顶元素优先权低
              Push(OPTR, *c);  c++;   break;
          case '=':   //  脱括号并接收下一字符
              Pop(OPTR, x);    c++;     break;
          case '>':   //  退栈并将运算结果入栈
              Pop(OPTR, theta);  Pop(OPND, b);  Pop(OPND, a);
              Push(OPND, Operate(a, theta, b));    break;
        } // switch
  } // while
  return GetTop(OPND);
} // EvaluateExpression
```

这里 c 没有 +

+

```
void hanoi (int n, char x, char y, char z)
1{  //  算法 3.5
  //  将塔座 x 上按直径由小到大且至上而下编号为 1 至 n 的 n 个圆盘按规则搬到
  //  塔座 z 上， y 可用作辅助塔座。
  //  搬动操作   move (x, n, z)  可定义为：
  //   (c 是初值为 0 的全局变量，对搬动计数 )
  //   printf("%i. Move disk %i from  %c  to  %c\n", ++c, n, x, z);
2  if (n==1)
3    move(x, 1, z);        // 将编号为 1 的圆盘从 x 移到 z
4  else {
5    hanoi(n-1,x,z,y);
6    move(x, n, z);        // 将编号为 n 的圆盘从 x 移到 z
7    hanoi(n-1, y, x, z);  // 将 y 上编号为 1 至 n-1 的圆盘移到 z
8  }
9}
```

链队列实现

```
Status DestroyQueue(LinkQueue &Q){
    while(Q.front){Q.rear=Q.front->next; free(Q.front);
                        Q.front=Q.rear;}
}
Status EnQueue(LinkQueue &Q,QElemType e){
    …p->data=e;Q.rear->next=p; Q->rear=p;…
}
Status DeQueue(LinkQueue &Q,QElemType &e){
    …p=Q.front->next;e=p->data;
       Q.front->next=p->next;
       if(Q->rear==p)Q->rear=Q->front; free(p);  …
}
```

循环队列实现

```
int QueueLength(SqQueue Q){
    return (Q.rear-Q.front+Q.size)%Q.size;
}
Status EnQueue(SqQueue &Q,QElemType e){
    if((Q.rear+1)%Q.size==Q.front)QueueIncrement(Q);
    Q.base[Q.rear]=e;Q.rear=(Q.rear+1)%Q.size;…
}
Status DeQueue(SqQueue &Q,QElemType &e){
    … e=Q.base[Q.front];
    Q.front=(Q.front+1)%Q.size; …
}
```

InitQueue_Sq/QueueIncrement

```
Status InitQueue_Sq(SqQueue &Q) {
    Q.base=(QElemType *)
           malloc(QUEUE_INIT_SIZE*sizeof(QElemType));
    if(!Q.base)exit(OVERFLOW);
    Q.front=Q.rear=0;
    Q. size=QUEUE_INIT_SIZE; return OK;}

Status QueueIncrement(SqQueue &Q){
    QElemType *newbase = (QElemType *)malloc(
             (Q. size+QUEUEINCREMENT)*sizeof (QElemType));
   if (!newbase) return ERROR;   //  存储分配失败
    for(i=0;i<Q.size-1;i++)
    newbase[i]=Q.base[(i+Q.front)%Q.size];
    free(Q.base);
   Q.base = newbase;   }
```

杨辉三角 *

```
void Yanghui ( int n ) { //  打印输出杨辉三角的前   n ( n > 0 ) 行
   for( i=1;  i<=n;  i++) cout<<' ' ;    cout<<'1'<<endl  //  在中心位置输出 "1" ;
   InitQueue _Sq (Q); EnQueue_Sq (Q, 0 );
   EnQueue_Sq ( Q, 1);  EnQueue_Sq ( Q, 1 );      k = 1;
   while ( k < n ) {                    //  通过循环队列输出前   n-1  行的值
      for( i=1;  i<=n-k;  i++) cout<<' ' ;  EnQueue_Sq ( Q, 0 ); //  行界值 "0" 入
      do {   //  输出第   k  行，计算第   k+1  行
         Dequeue_Sq ( Q, s );  GetHead_Sq ( Q, e );
         if (e) cout<<e<<' ' ;    else cout<<endl; //  若 e 为非行界值 0 打印输出   e
         EnQueue(Q, s+e);
      } while (e!=0);
      k++;
    }//while
    DeQueue_Sq ( Q, e );
    while (!QueueEmpty (Q) ) {DeQueue_Sq ( Q, e ); cout<<e<<' ';} // 处理   n  行
} // yanghui
```

分割无冲突子集 *

```
void division ( int R[ ][ ],  int n,  int  result [ ] ) {
  // 结果  记入 result  数组，result[k] 的值是编号为   k  的元素所属组号
  pre = n;  group = 0;  InitQueue_Sq ( Q, n+1 );   // 设置容量为   n  的空队列
  for ( e = 0;  e < n;  e++ )  EnQueue_Sq ( Q, e, );
  while ( !QueueEmpty ( Q ) ) {
      DeQueue_Sq ( Q, i );
      if ( i < pre ) {group ++;      // 增加一个新的组
      for ( j = 0;  j < n;  j ++ )   clash[j] = 0;}
      if ( clash[i] == 0 ) {
         result[i] = group;                              // 编号为 i  的元素入 group  组
         for ( j = 0;  j < n;  j ++ )  clash[j] += R[i][j];   // 添加和   i 冲突的信息
      }
      else  EnQueue ( Q, i );       // 编号为 i  的元素不能入当前组
      pre = i;
  } // while
}// division
```

第五章 数组

5.1 数组的定义

5.2 数组的表示和实现 *

5.3 数组的压缩

5.1 数组的定义

ADT Array{

数据对象： $D=\{a_{j1}a_{j2}\ldots a_{jn} \mid a_{j1}a_{j2}\ldots a_{jn} \in Elemset,$

$j_i=0,1,\ldots b_i-1$，bi 是数组第 i 维的长度，n 是位数 }

数据关系： $R=\{R_1,R_2\ldots R_n\}$

$R_i=\{< a_{j1}\ldots a_{ji}\ldots a_{jn} , a_{j1}\ldots a_{ji+1}\ldots a_{jn} > \mid$

$a_{j1}\ldots a_{ji}\ldots a_{jn} , a_{j1}\ldots a_{ji+1}\ldots a_{jn} \in D, i=2,3\ldots n$

$0<=j_k<=b_k-1, 1<=k<=n, k!=I$

$0<=j_i<=b_i-1$ }

基本操作 ：

initArray(&A,n,bound1,bound2...boundn)

DestroyArray(&A)

Value(A,&e,index1,index2.....indexn)

Assign(&A,e,index1,index2.....indexn)

}ADT Array

- 二维数组定义
  - 其数据元素是一维数组的线形表
- N 维数组定义
  - 其数据元素是 N - 1 维数组的线形表

5.2 数组的顺序表示和实现

- 数组元素的两种存储方式
  - 行主序存储
  - 列主序存储
- 数组中元素在内存映象中的关系：
  - 二维数组 A[m][n]

    $LOC(i,j)=LOC(0,0)+(i*n+j)*L$
  - 三维数组 B[p][m][n]

    $LOC(i,j,k)=LOC(0,0,0)+(i*m*n+j*n+k)*L$
  - 多维：

$$LOC(j_1,j_2\ldots j_n)=LOC(0,0\ldots0)+(b_2*\ldots*b_n*j_1+b_3*\ldots*b_n*j_2$$
$$+\ldots+bn*j_{n-1}+j_n)*L=LOC(0,0\ldots0)+\sum c_ij_i$$
$$c_n=L,c_{i-1}=b_i*c_i,1<i<=n$$

数组应用

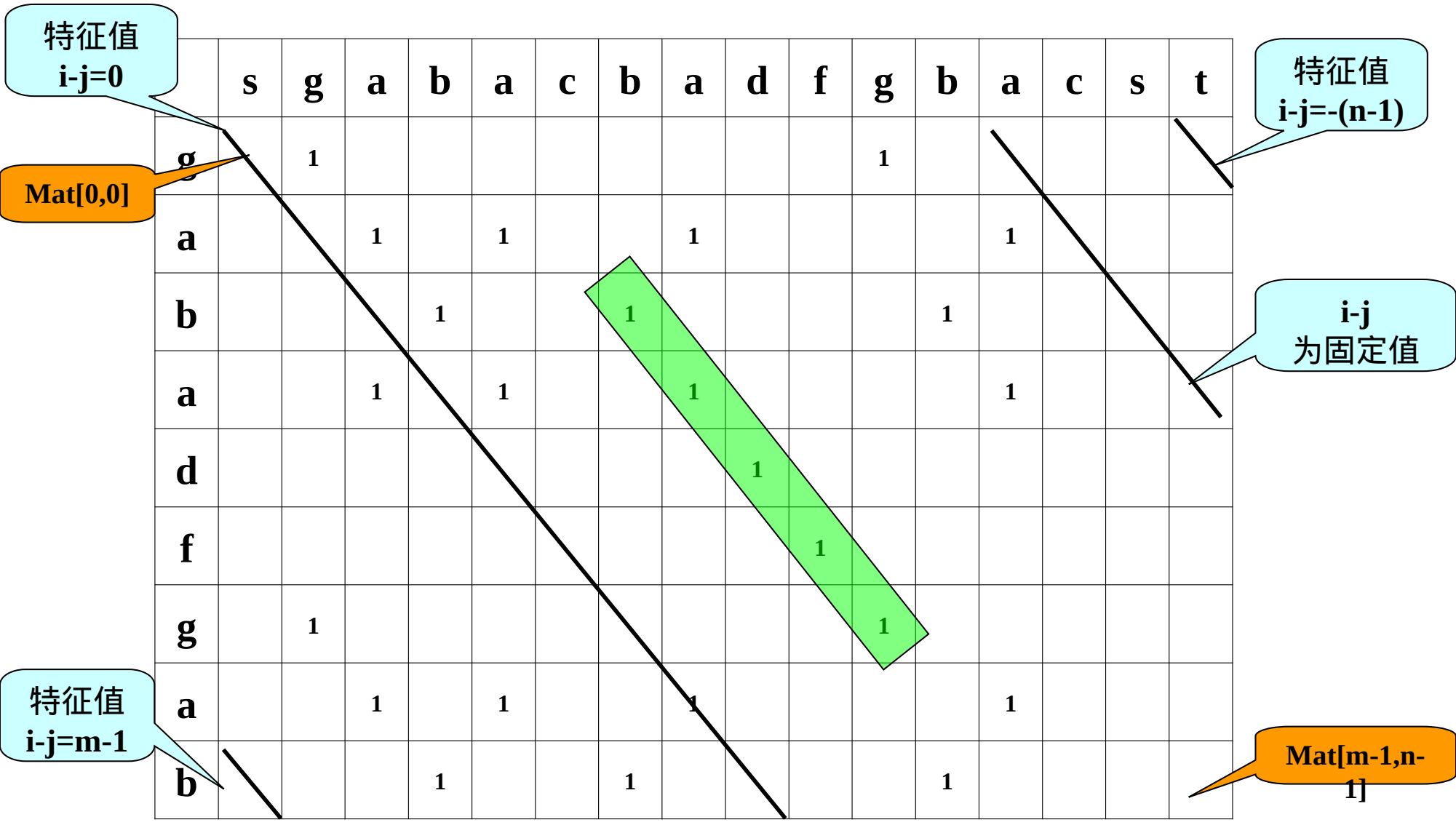
- 例寻找两个串最长的公共子串
  - 通常的匹配算法复杂度 $O(m*n^2)$
  - 利用二维数组构造串之间的关系来求解

    S1=“sgabacbadfgbacst”

    S2=“gabadfgab”

    最大公共子串“ badfg” 。

    算法时间复杂度 $O(m*n)$


特征值
i-j=0
Mat[0,0]
特征值
i-j=-(n-1)
i-j
为固定值
特征值
i-j=m-1
Mat[m-1,n-1]

5.3 数组的压缩

- 特殊形状矩阵的存储表示 (下标从 0 开始)

对称矩阵： A[n][n] 存储到 B[n(n+1)/2]

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

三角矩阵： A[n][n] 存储到 B[n(n+1)/2]

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

带状矩阵 A[n][n] 存储到 B[3n-2]

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

随机稀疏矩阵

– 非零元比零元少的多且分布无规律的矩阵。

随机稀疏矩阵的存储表示

- 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;
```

$$\begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{pmatrix}$$

三元组顺序表示例(下标从 1 开始)

(a) M.data

	i	j	e
1	1	2	12
2	1	3	9
3	3	1	-3
4	3	6	14
5	4	3	24
6	5	2	18
7	6	1	15
8	6	4	-7

(b) T.data

	i	j	e
1	1	3	-3
2	1	6	15
3	2	1	12
4	2	5	18
5	3	1	9
6	3	4	24
7	4	6	-7
8	6	3	14

$$\begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{pmatrix}$$

cpot[1]=1

cpot[col]=cpot[col-1]+num[col-1] 2<=col<=nu

col	1	2	3	4	5	6	7
num[col]	2	2	2	1	0	1	0
cpot[col]	1	3	5	7	8	8	9

- 矩阵转置

  void transpose(ElemType M[][],T[][])

  时间复杂度 O(n×m)

- “按需点菜”的思想

  复杂度 O(M.nu×M.tu)

- “按位就座”的思想
  - 建立辅助数组 num[n] cpot[n]

    void createpos(TSMatrix M)
  - 快速转置

    status fastTransposeSMatrix(TSMatrix M, TSMatrix T)

    时间复杂度 O(M.nu+M.tu)

- 逻辑链接的顺序表
  - 为了能随机存取三元组表示数组的任意一行非零元素，在建立三元组顺序表存储结构同时建立 rpos[n] 辅助数组，这种带行链接信息的三元组表称为“逻辑连接的顺序表”

```
typedef struct{
        Triple  data[MAXSIZE+1];
        int     rpos[MAXMN+1];
        int     mu,nu,tu;
}RLSMatrix;
```

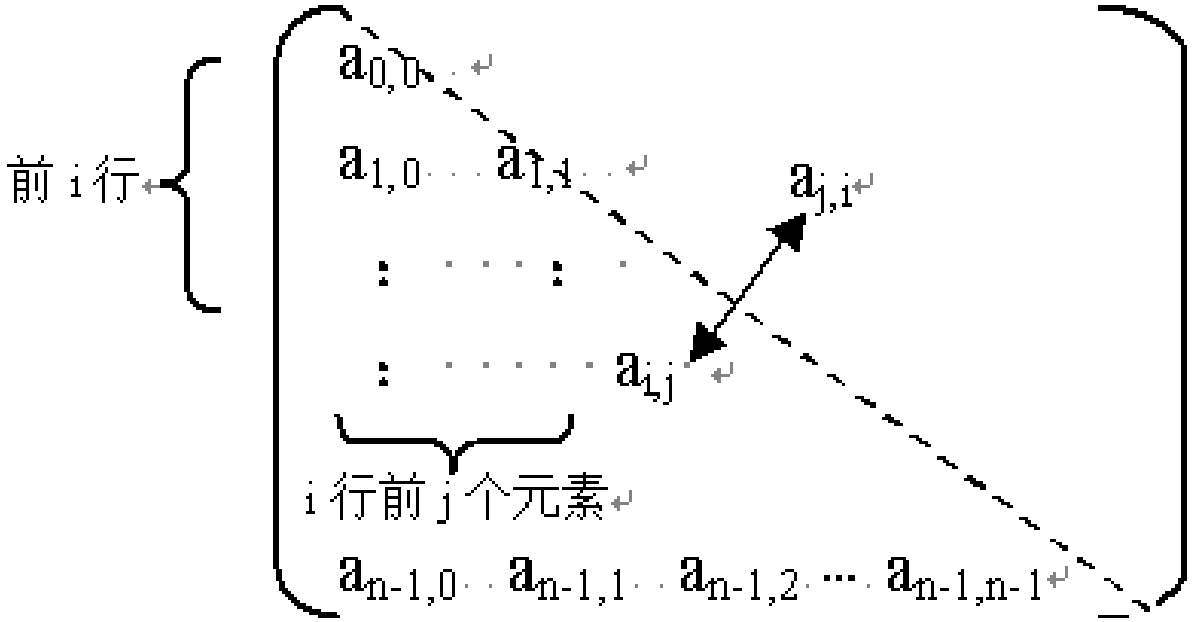
十字链表

– 矩阵运算增加或减少非零元时，使用链表结构来表示三元组序列。

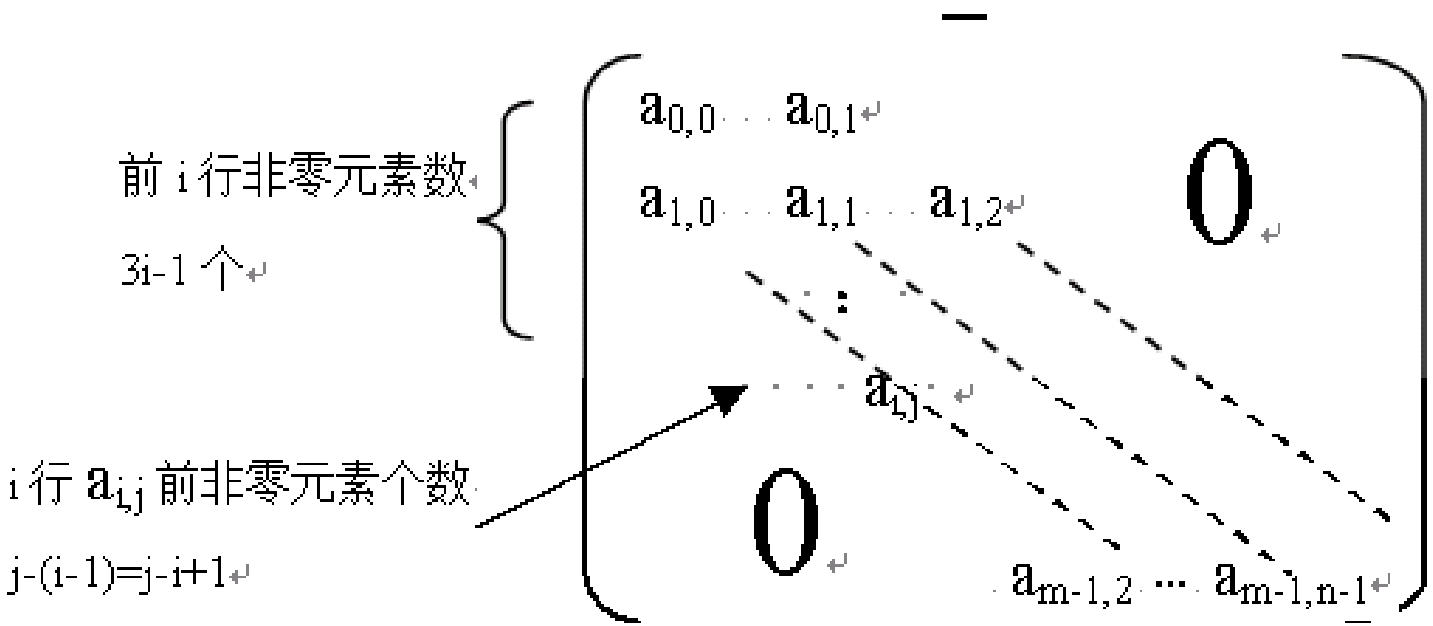
```
typedef struct OLNode{
        int                 i,j;
        ElementType         e;
        struct OLNode       *rnext,*cnext;
}OLNode,*Olink;
typedef struct {
        Olink   *rhead,*chead;
        int     mu,nu,tu;
}CrossList;
```

谢　谢

对称阵



条带阵


前 i 行非零元素数
3i-1 个
i 行 $a_{i,j}$ 前非零元素个数
j-(i-1)=j-i+1
$a_{0,0}$ $a_{0,1}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$
0
$a_{i,j}$
0
$a_{m-1,2}$ … $a_{m-1,n-1}$

三元组存储数组的快速转置

```
Status  FastTransposeSMatrix(TSMatrix M, TSMatrix &T) {  // 算法 5.2
  // 采用三元组顺序表存储表示，求稀疏矩阵 M 的转置矩阵 T
  T.mu = M.nu;  T.nu = M.mu;  T.tu = M.tu;
  if (! T.tu) return FAIL;
    for (col=1; col<=M.nu; ++col) num[col] = 0;
    for (t=1; t<=M.tu; ++t) //  求   M  中每一列所含非零元的个数
       ++num[M.data[t].j];
    cpot[1] = 1;
    //  求   M  中每一列的第一个非零元在   b.data  中的序号
    for (col=2; col<=M.nu; ++col) cpot[col] = cpot[col-1]+num[col-1];
    for (p=1; p<=M.tu; ++p) {
                  col = M.data[p].j;   q = cpot[col];
                  T.data[q].i =M.data[p].j;  T.data[q].j =M.data[p].i;
                  T.data[q].e =M.data[p].e;  ++cpot[col];
    } // for
} // FastTransposeSMatrix
```

第六章 二叉树和树

6.1 树的基本概念

6.2 二叉树

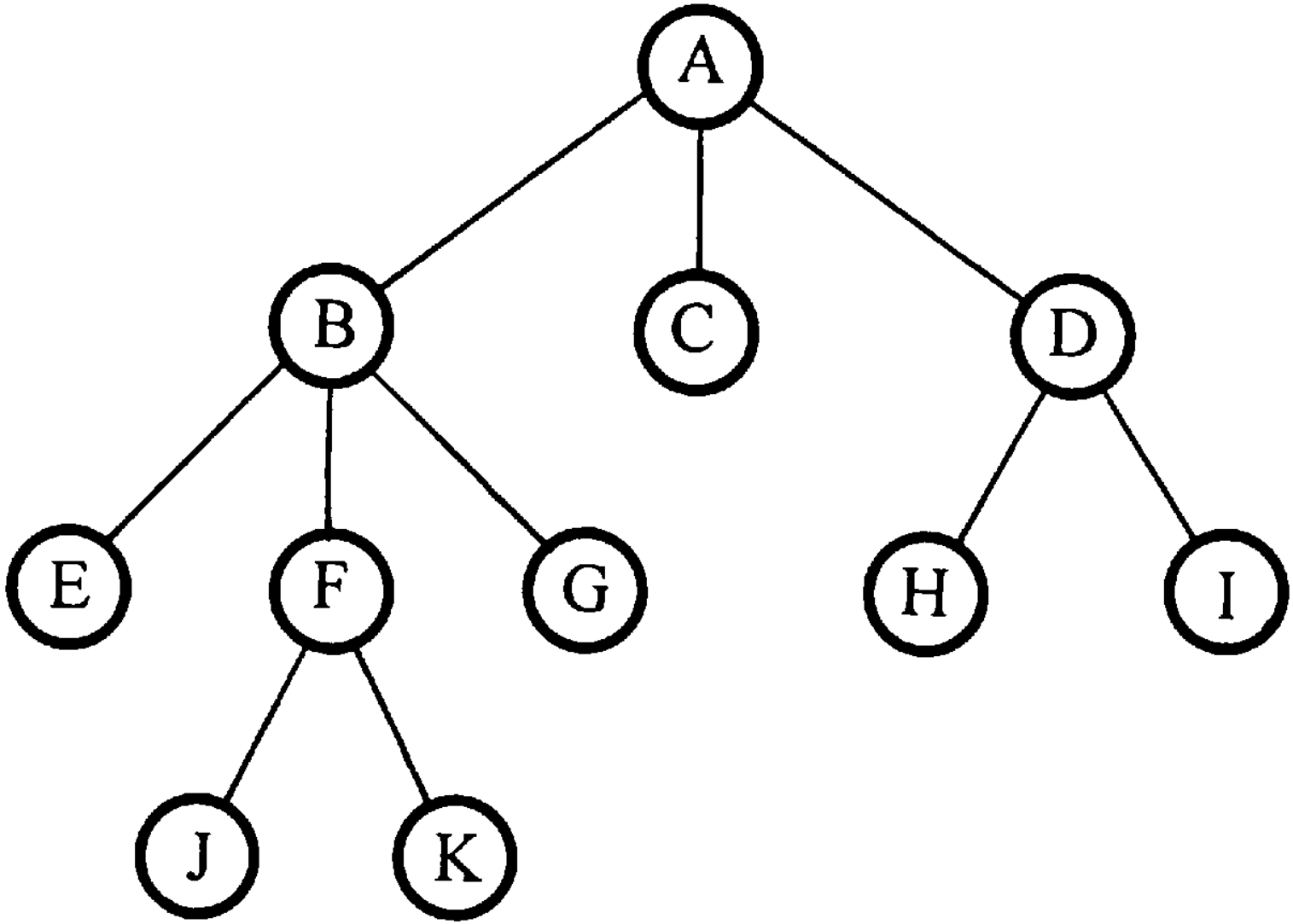
6.3 二叉树遍历和线索二叉树

6.4 树和森林

6.5 * 树的等价问题 / 树的应用

6.6 霍夫曼树及其应用

6.1 树的基本概念



树的深度：树中叶子结点的最大层次数

ADT Tree{

数据对象 D: D 是具有相同特性的数据元素的集合。

数据关系：若 D 为空或一个元素时，R 为空集。否则 R={H} ，H 是如下二元关系：

1 ）在 D 中存在唯一元素称根 root ，它在关系 H 下无前驱 ;

2 ）若 D-{root}≠ϕ, 则存在 D-{root} 的一个划分 $D_1,D_2,\ldots D_m(m>0)$, 对任意 $j \neq k$ （ $1\leq j,k \leq m$ ）有 $D_j \cap D_k = \phi$, 且对任意 $i(1 \leq i \leq m)$, 存在唯一元素 $x_i \in D_i$, 有 $<root,x_i> \in H$;

3) 对应于 D-{root} 的一个划分，$H-\{<root,x_1>,\ldots,<root,x_m>\}$ 有唯一的一个划分 $H_1,H_2,\ldots H_m(m>0)$, 对任意 $j \neq k$ （ $1\leq j,k \leq m$ ）有 $D_j \cap D_k = \phi$, 且对任意 $i(1 \leq i \leq m)$,H_i 是 Di 上的二元关系，（ Di,{Hi} ）是一个符合本定义的树，称根 root 的子树。

基本操作：

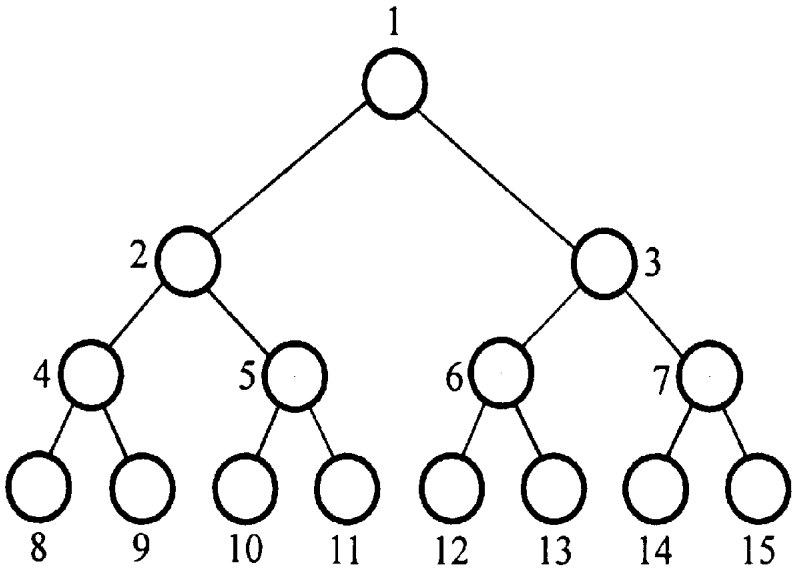
```
    InitTree(&T);
    DestroyTree(&T);
    CreateTree(&T,definition);
    ClearTree(&T);
    TreeEmpty(T);
    TreeDepth(T);
    Root(T);
    Value(T,cur_e);
    Assign(&T,cur_e,val);
    Parent(T,cur_e);
    FirstChild(T,cur_e);
    RightSibling(T,cur_e);
    InsertChild(&T,p,i,c);
    DeleteChild(&T,p,i);
    TraverseTree(T);
}//ADT Tree
```

6.2 二叉树

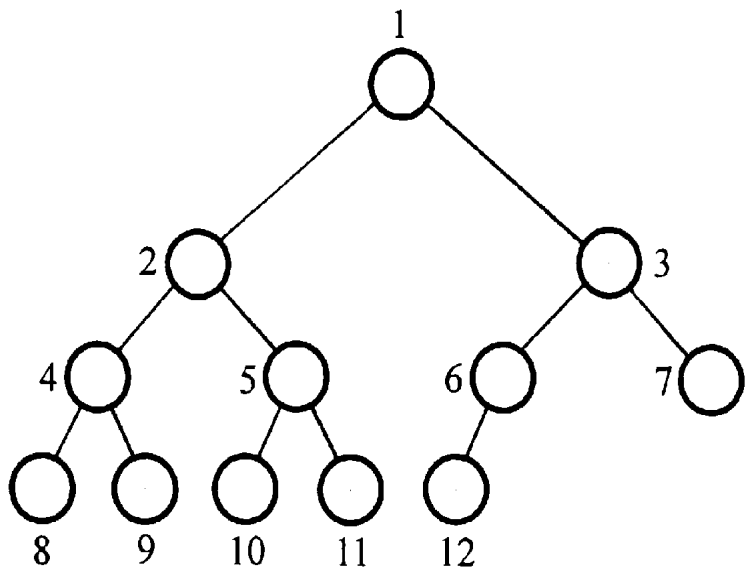
6.2.1 二叉树定义（ binary Tree ）

- 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
- 二叉树的元素又称结点
- 左孩子，右孩子，父亲，兄弟，堂兄弟，祖先，子孙
- 结点的度：非空子树的个数
- 叶子结点：左右子树均空的结点；度为零
- 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（ full binary tree ）：所有非叶子结点度为 2 ，叶子结点在同一层次
- 完全二叉树（ complete binary tree ）：深度 h ， h-1 为满二叉树， h 层的结点都集中在左侧
- 二叉树的基本操作



(a) 满二叉树



(b) 完全二叉树

6.2.2 二叉树的几个基本性质

- 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 $n_0=n_2+1$

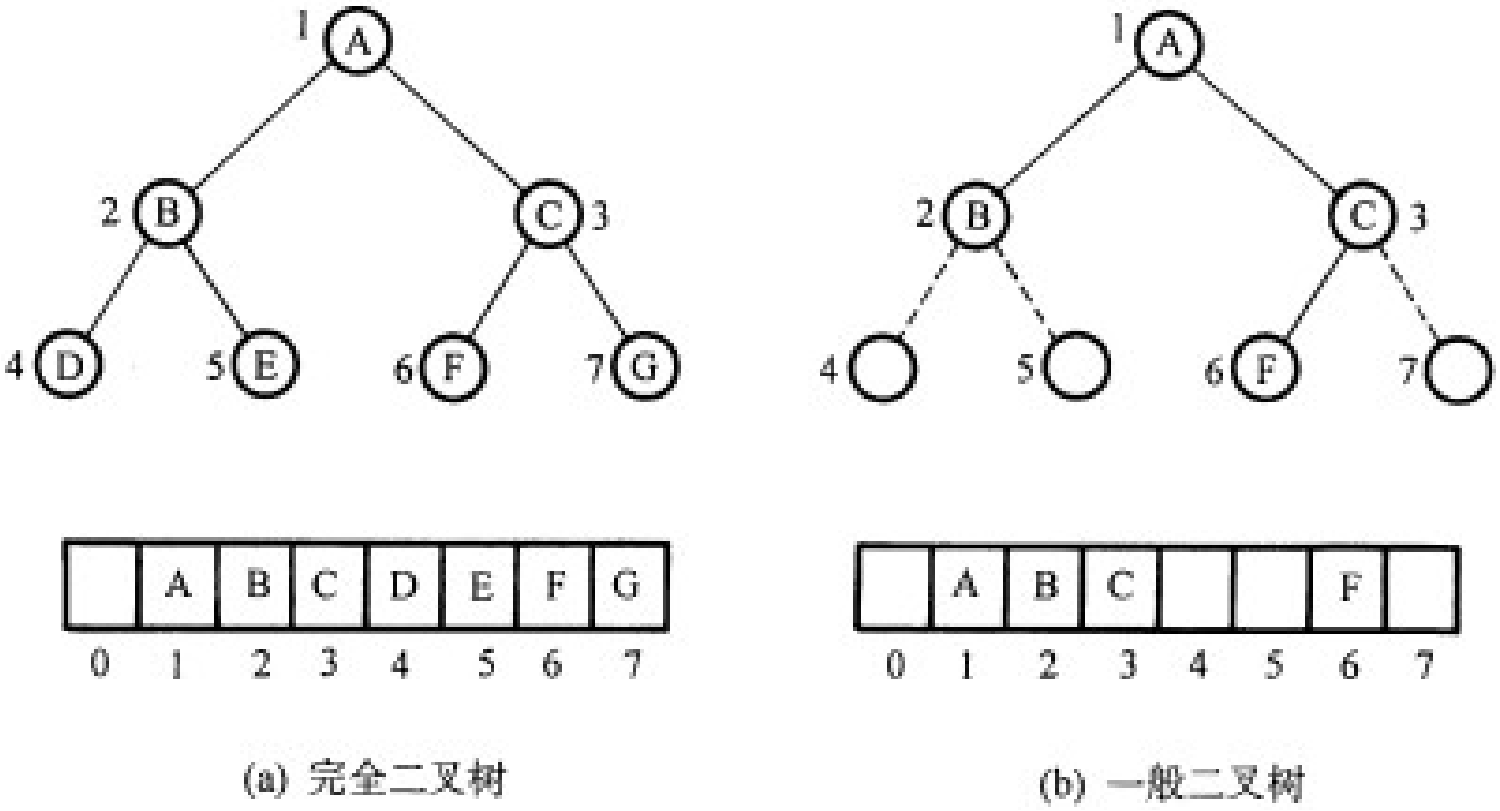
  $n_0+n_1+n_2=n=2n_2+n_1+1$
- 性质 **4**　具有 n 个结点的完全二叉树深度为 $\lfloor \log n \rfloor+1$ 或 $\lceil \log(n+1) \rceil$

- 性质 5　如果对一个有 n 个结点的完全二叉树 T 的结点按层序（从第一层到第 [logn]+1 层，层内从左到右）从 1 开始编号，则对任意一个编号为 i(1<=i<=n) 的结点有：
  - 如果 i = 1 ，则该结点是二叉树的根，无双亲；如果 i>1 ，则其双亲结点 Parent(i) 的编号为 [i/2]
  - 如果 2i>n ，则编号为 i 的结点没有左孩子，为叶子结点；否则其左孩子 LChild(i) 的编号为 2i
  - 如果 2i+1>n ，则编号为 i 的结点没有右孩子；否则其右孩子 RChild(i) 的编号为 2i+1

6.2.3 二叉树的存储结构

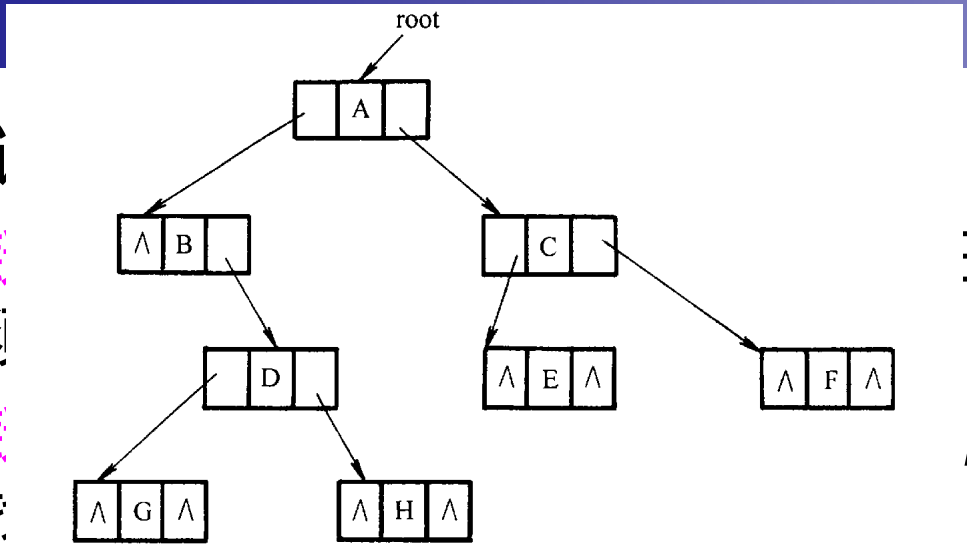
- 顺序存储结构

  Const MAXSIZE = 100

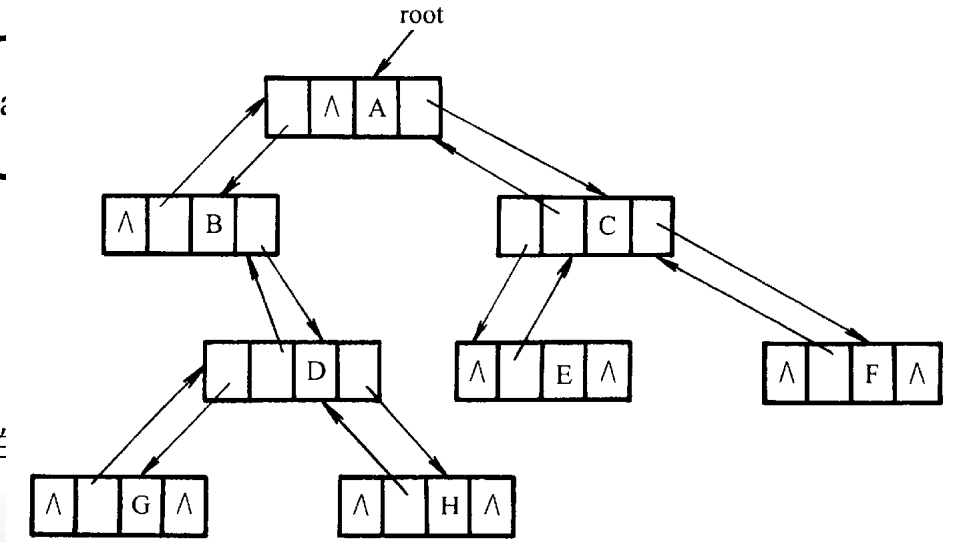


(a) 完全二叉树　　(b) 一般二叉树

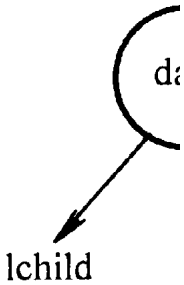
- 链式存储
  - 二叉链表：三个域，找父亲必须
  - 三叉链表：，parent 四个域，找

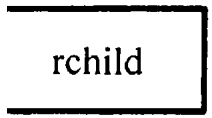


(a) 二叉链表

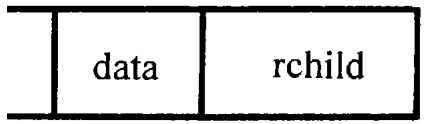


(b) 三叉链表





结点结构



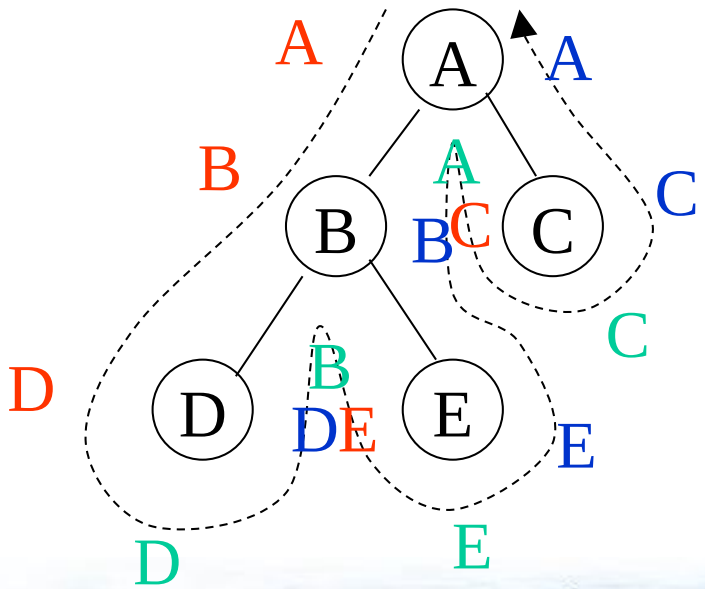
(a) 结点

结点结构

6.3 二叉树遍历和线索二叉树

6.3.1 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
- 例：右图得三种遍历序列
  - 先序遍历：ABDEC
  - 中序遍历：DBEAC
  - 后序遍历：DEBCA



遍历的递归实现

先序遍历二叉树的定义：

若二叉树为空，则空操作；否则

(1) 访问根结点；

(2) 先序遍历左子树；

(3) 先序遍历右子树。

```
void PreOrder(BiTree T){
    if(!T) return;
    visite(T->data);            // 访问根结点
    PreOrder(T->lchild);        // 先序遍历左子树
    PreOrder(T->rchild);        // 先序遍历右子树
}//PreOrder
```

中序遍历二叉树的定义：

若二叉树为空，则空操作；否则

(1) 中序遍历左子树；

(2) 访问根结点；

(3) 中序遍历右子树。

```
void InOrder (BiTree T){
    if(!T) return;
    InOrder (T->lchild);     // 中序遍历左子树
    visite(T->data);         // 访问根结点
    InOrder (T->rchild);     // 中序遍历右子树
}//InOrder
```

后序遍历二叉树的定义：

若二叉树为空，则空操作；否则

(1) 后序遍历左子树；

(2) 后序遍历右子树；

(3) 访问根结点。

```
void PostOrder (BiTree T){
    if(!T) return;
    PostOrder (T->lchild);  // 后序遍历左子树
    PostOrder (T->rchild);  // 后序遍历右子树
    visite(T->data);        // 访问根结点
}//InOrder
```

扩展二叉树

先序扩展序列：

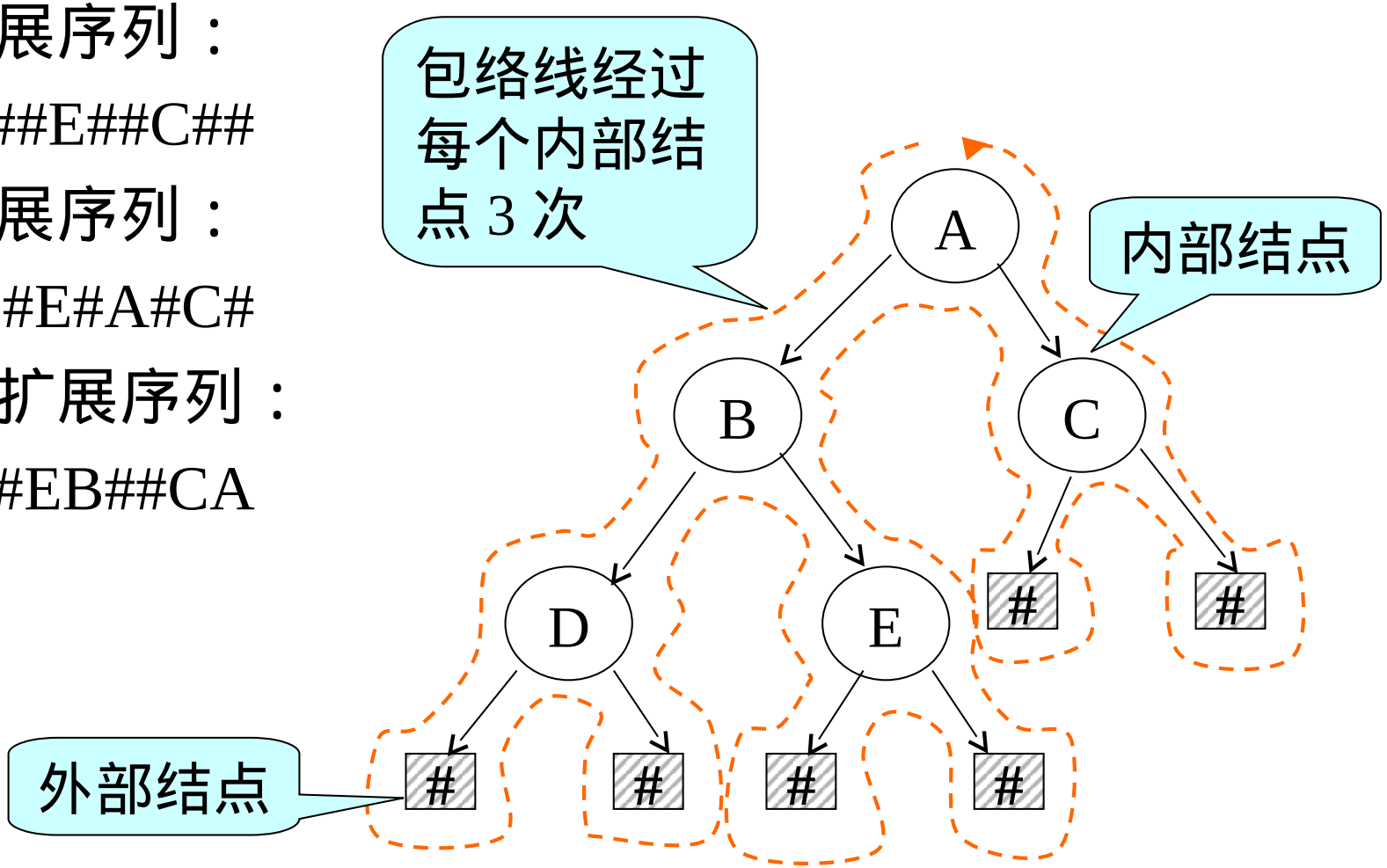
ABD##E##C##

中序扩展序列：

#D#B#E#A#C#

后续序扩展序列：

##D##EB##CA



序列确定二叉树

- 一个先 / 中 / 后序序列能否确定一个二叉树？
- 一个先序扩展序列能否确定一个二叉树？
- 一个后序扩展序列能否确定一个二叉树？
- 一个中序扩展序列能否确定一个二叉树？
- 一个先序和中序序列能否唯一确定一个二叉树？
- 一个中序和后序序列能否确定一个二叉树？
- 一个先序和后序能序列能否确定一个二叉树？
- 例：
  - 已知先序扩展序列： AB#DF###C#E##
  - 已知后序扩展序列： ##D##G#EB###H#FCA
  - 已知先序 :ABCDEFGHIJ 中序 :BCDAFEHJIG
  - 已知中序 :BFDAEGC 后序 :FDBGECA

遍历算法的描述

- 算法　递归实现

  void preorder(BiTree T,void (* visit)(BiTree))

- 算法　非递归实现

```
typedef  enum {Travel=1,Visit=0} TaskType;
typedef  struct {
            BiTree          ptr;
            TaskType        task;
}SElemType;
void InOrder_iter (BiTree BT,void (* visit)(BiTree))
```

用栈实现遍历

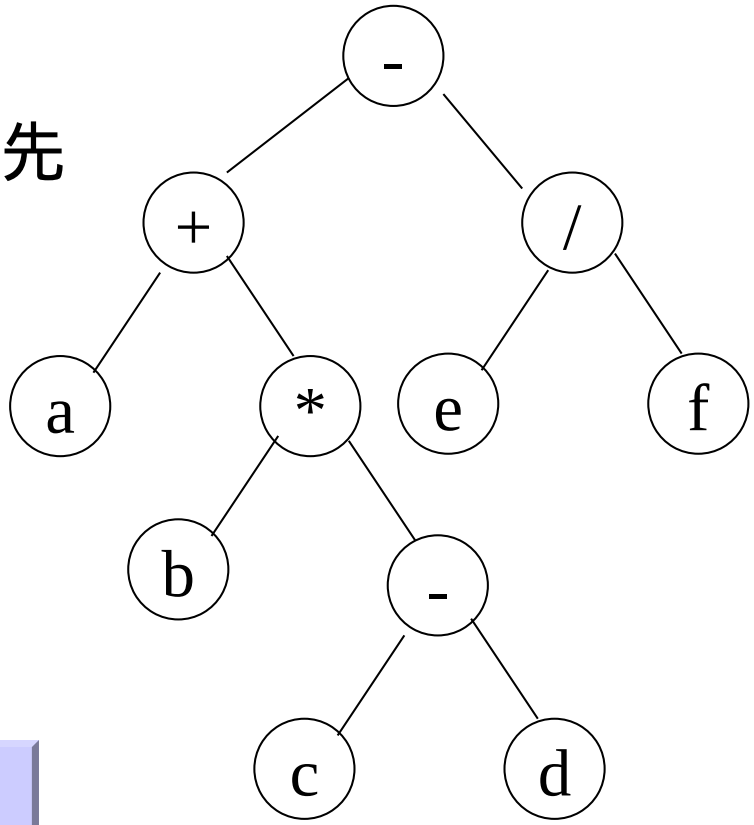
非递归循环实现

层次遍历 *

```
void LayerTraversal （ BiTree T ）{// 按层序遍历二叉树 T
    InitQueue(Q);   // 初始化一个空队列
    if(T) EnQueue(Q,T);      // 非空根指针入队
    while(!QueueEmpty(Q)){
      DeQueue(Q,p);          // 队头出队
      visite(p->data);       // 访问出队的结点 *p
      if(p->lchild) EnQueue(Q, p->lchild); //*p 左孩子入队
      if(p->rchild) EnQueue(Q, p->rchild); //*p 右孩子入队
    }
    }LayerTraversal
```

二叉树和表达式

- 表达式 a+b*(c-d)-e/f （二叉树表示）
- 中序遍历
  - a+b*c-d-e/f （中缀，丢失计算优先级）
- 先序遍历
  - -+a*b-cd/ef （前缀 / 波兰式）
- 后序遍历
  - abcd-*+ef/- （后缀 / 逆波兰式）



二叉树遍历应用举例

- 例 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;
    else{
                T=new BiTNode;
                T->data=ch;
                CreateBiTree(T->lchild);
                CreateBiTree(T->rchild);
    }
}
```

- 例　复制二叉树　递归算法

```
Bitree CopyTree(BiTree T){
    if(!T)return NTLL;
    newlp=CopyTree(T->lchild);
    newrp=CopyTree(T->rchild);
    newnode=new BiTNode;
    newnode->data=T->data;
    newnode->lchild=newlp;
    newnode->rchild=newrp;
    return newnode;
}
```

- 例 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);
}
```

- 例 输出二叉树每个结点的层次数

```
void level(BiTree T,int lev){    //lev=1 调用
    if(!T) return;
    cout<<T->data<<“:”<<lev<<endl;
    lev++;
    level(T->lchild,lev);
    level(T->rchild,lev);
}
```

```
void numb(BiTree T, int &lev){        // 对比：输出的还是层次吗？
    if(!T) return;
    cout<<T->data<<“:”<<lev<<endl;
    lev++;
    numb (T->lchild,lev);
    numb (T->rchild,lev);
}
```

- 例 求二叉树的深度 递归算法

```
int Depth(BiTree T) {  // 后序
    if(!T) return 0;
    hl=Depth(T->lchild);hr=Depth(T->lchild);
    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);
}
```

6.3.2 线索二叉树

- 在二叉链表的结点中增加 Tag 域，分别表示指针类型，0 表示孩子，1 表示后继或前驱。指向前驱或后继的指针称线索。加上线索的二叉树成为线索二叉树

```
typedef enum PointerTag {Link,Thread};
typedef BiTHrNode{
    ElemType          data;
    struct BiTHrNode          *lchild,*rchild;
     PointerTag                    ltag,rtag;
}BiTHrNode,*BiThrTree;
Void InOrderThreading(BiThrTree &H, BiThrTree T);
Void InThreading(BiThrTree p, BiThrTree &pre);
InOrderTraverse_Thr(BiThrTree  T,  Status(*Visi
   ())
```

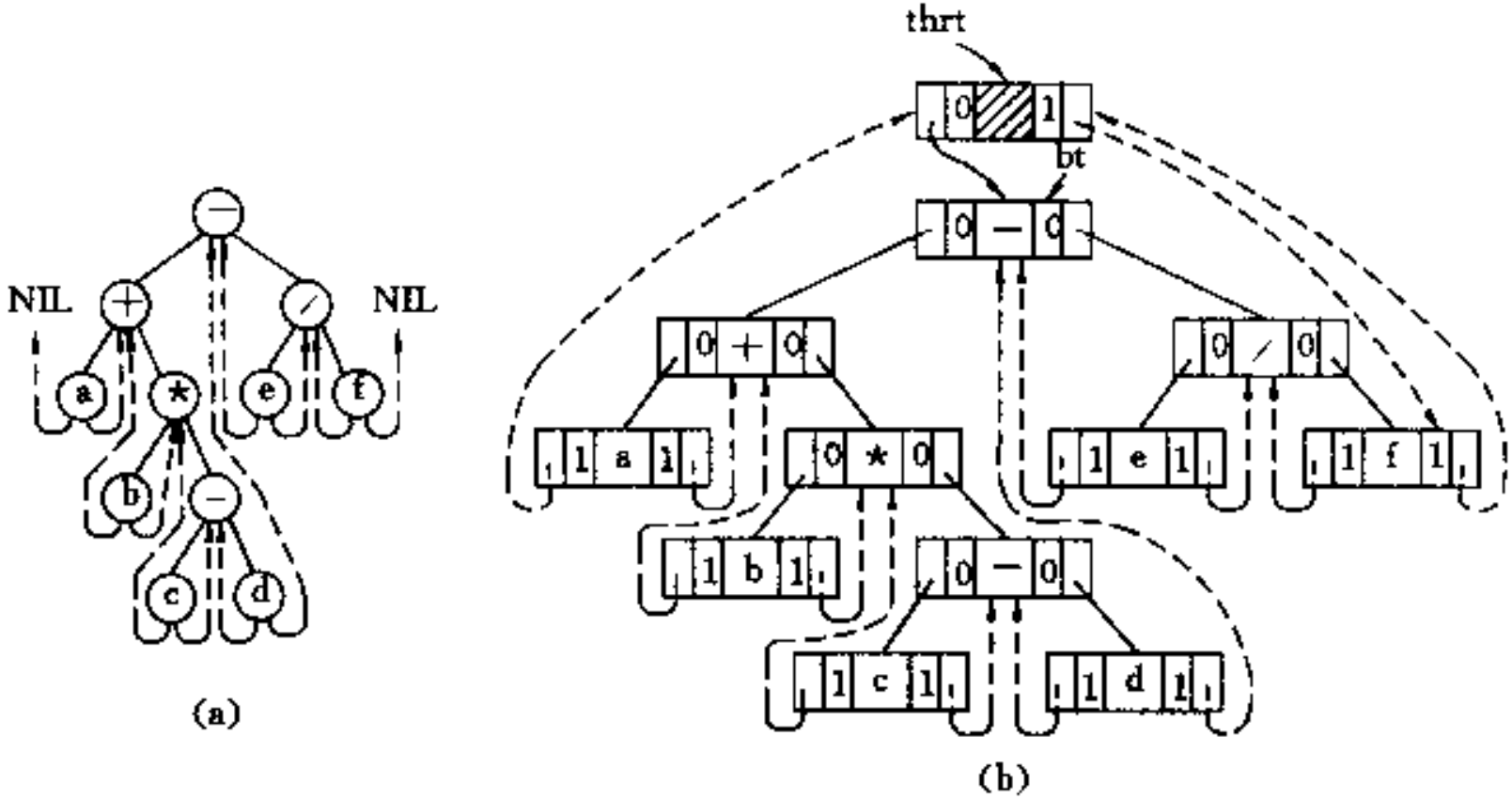


图 6.11　线索二叉树及其存储结构

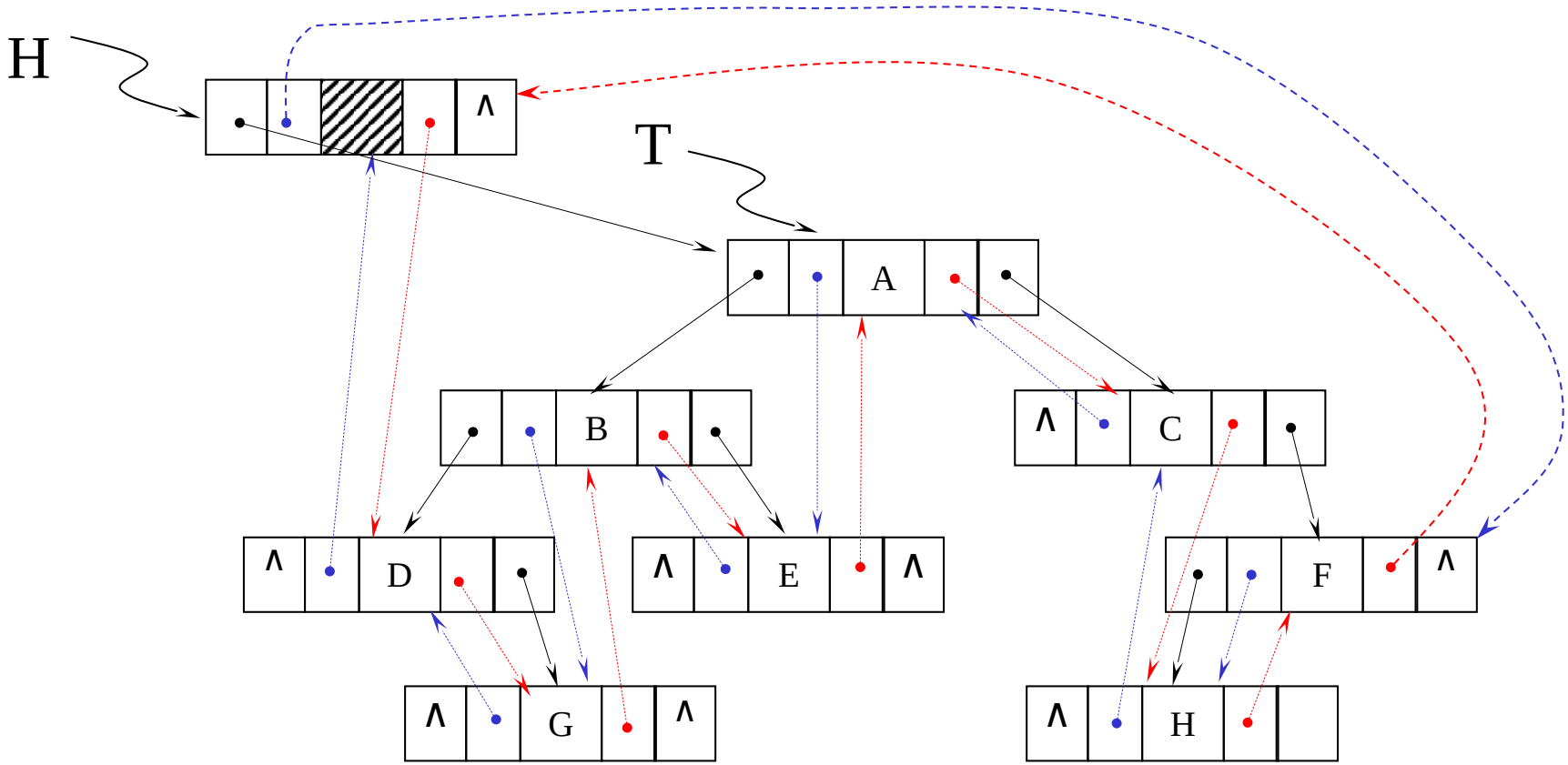
(a) 中序线索二叉树；(b) 中序线索链表

另一种线索二叉树 *

- 在二叉链表的结点中增加两个指针域，分别指向“前驱”和“后继”
- typedef BiTHrNode{

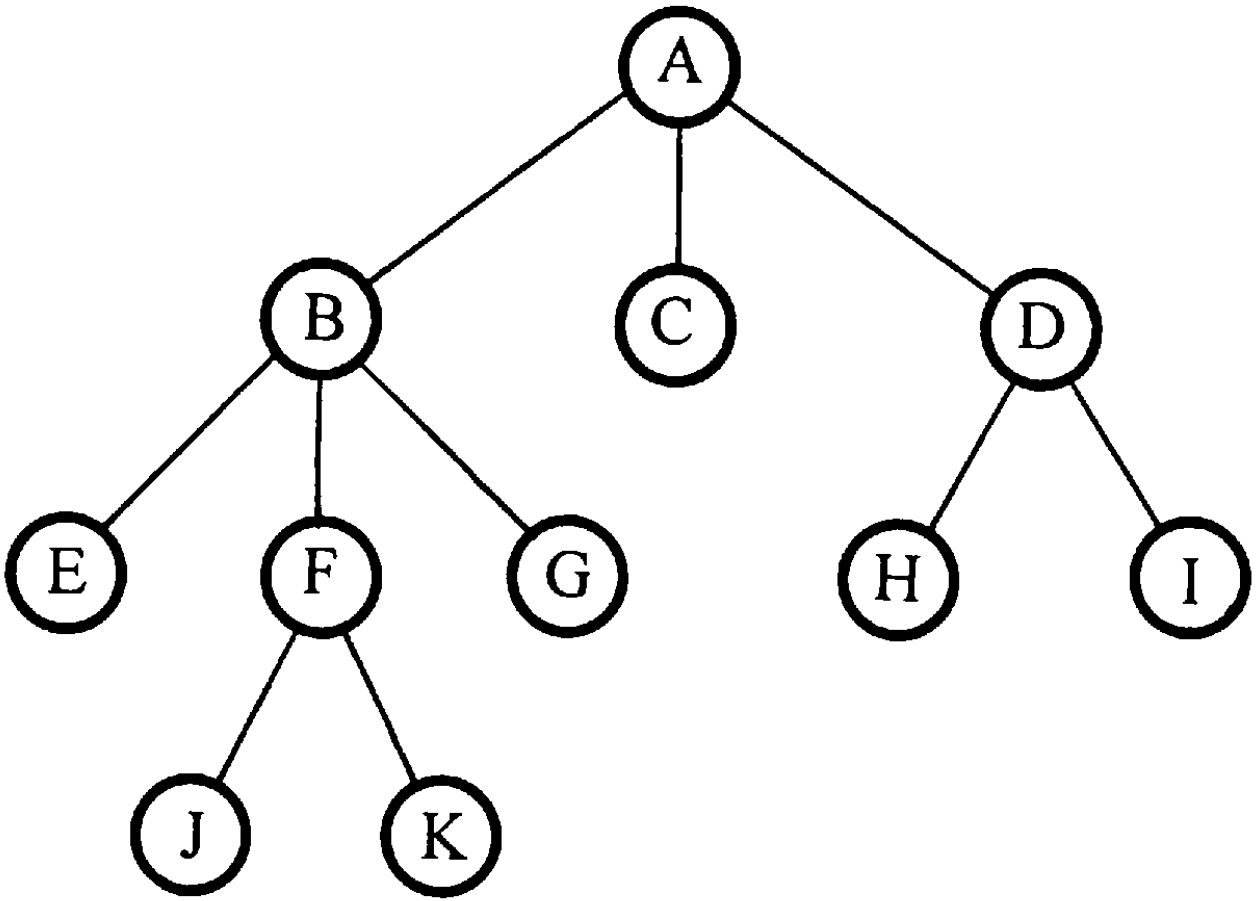
```
    ElemType              data;
    struct BiTHrNode      *lchild,*rchild;
    struct BiTHrNode      *pred,*succ;
}BiTHrNode,*BiThrTree;
```

- 二叉树中序线索化



6.4 树和森林

- 森林的定义
  - 是 m(m>=0) 颗互不相交的树的集合



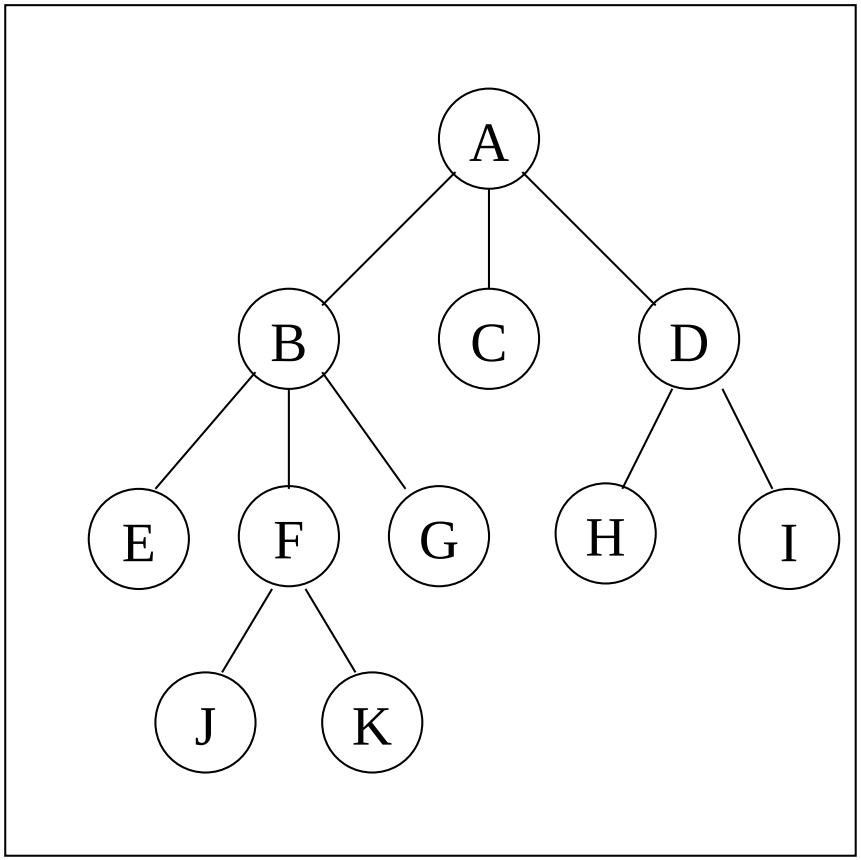
- 树的基本操作：
  - InitTree（ &T ）
  - DestroyTree（ &T ）
  - CreateTree(&T,definition)
  - TreeEmpty(T)
  - TreeDepth(T)
  - Parent(T, e)
  - LeftChild（ T ， e ）
  - Rightsibling(T,e)
  - InsertChild（ &T,&p,i,C ）
  - DeleteChild（ &T,&p,i ）
  - Traverse（ T ）

6.4.1 树和森林的存储结构

1、双亲表示法

```
Const MAX_TREE_SIZE=100
typedef struct PTNode{
        Elem        data;
        int         parent;
}PTNode;
typedef struct {
        PTNode      nodes[MAX_TREE_SIZE];
        int         r,n;
}PTree;
```

例：前图的双亲存储示意图　r=0 n=11



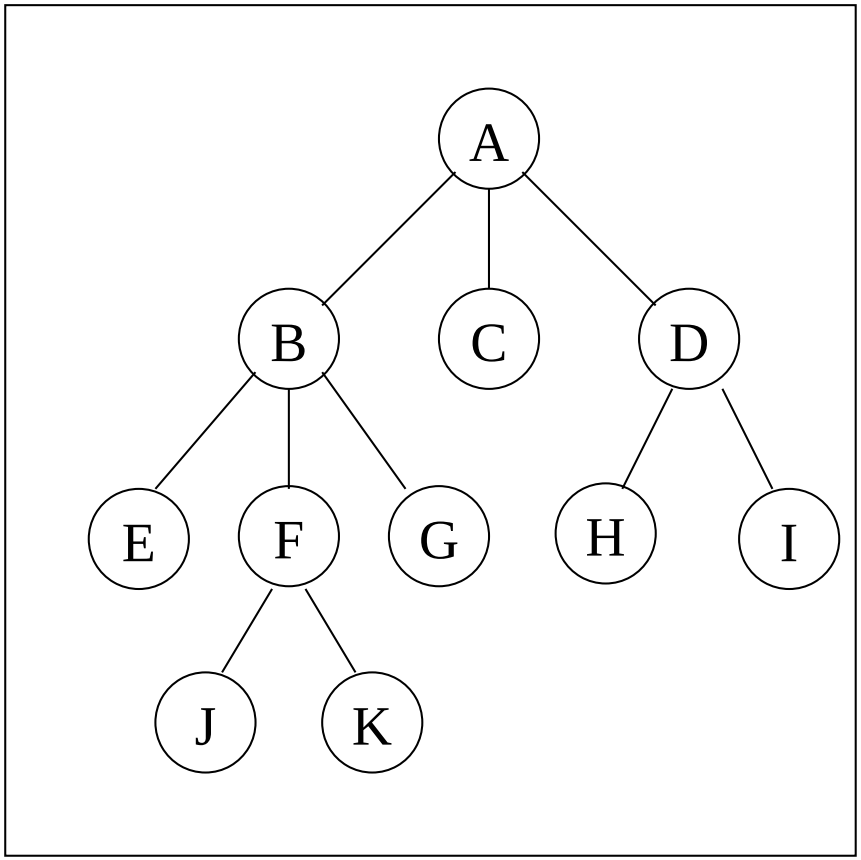
	data	parent
0	A	-1
1	B	0
2	C	0
3	D	0
4	E	1
5	F	1
6	G	1
7	H	3
8	I	3
9	J	5
10	K	5

2、孩子表示法

1) 孩子链表示法

```
typedef struct CTNode{
        int                         child;
        struct     CTNode *next;
}CTNode,*Childptr;
typedef struct {
        Elem              data;
        Childptr firstchild;
}CTBox;
typedef struct {
        CTBox       nodes[MAX_TREE_SIZE];
        int             n,r;
}CTree;
```

例：前图孩子链表示， r=0,n=11



	data	fchild
0	A	→ 1 → 2 → 3 ∧
1	B	→ 4 → 5 → 6 ∧
2	C	∧
3	D	→ 7 → 8 ∧
4	E	∧
5	F	→ 9 → 10 ∧
6	G	∧
7	H	∧
8	I	∧
9	J	∧
10	K	∧

2 ）树的多重链表表示法

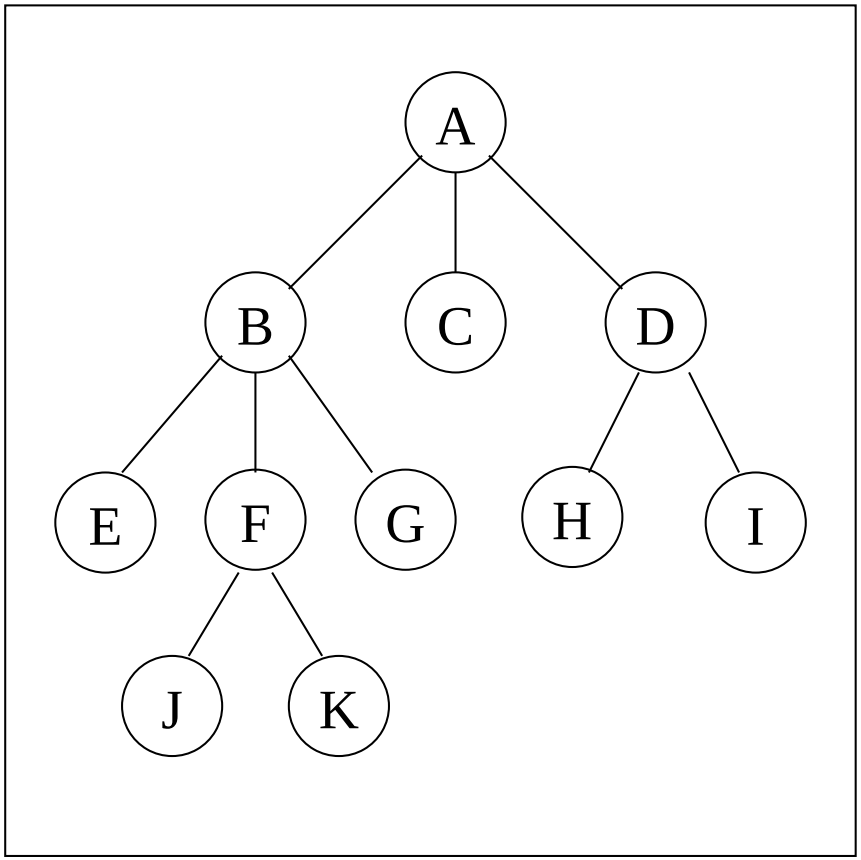
```
typedef struct TNode{
        Elem    data;
        struct   TNode  *child[MAX_NODE];
}TNode,*Tree;
```

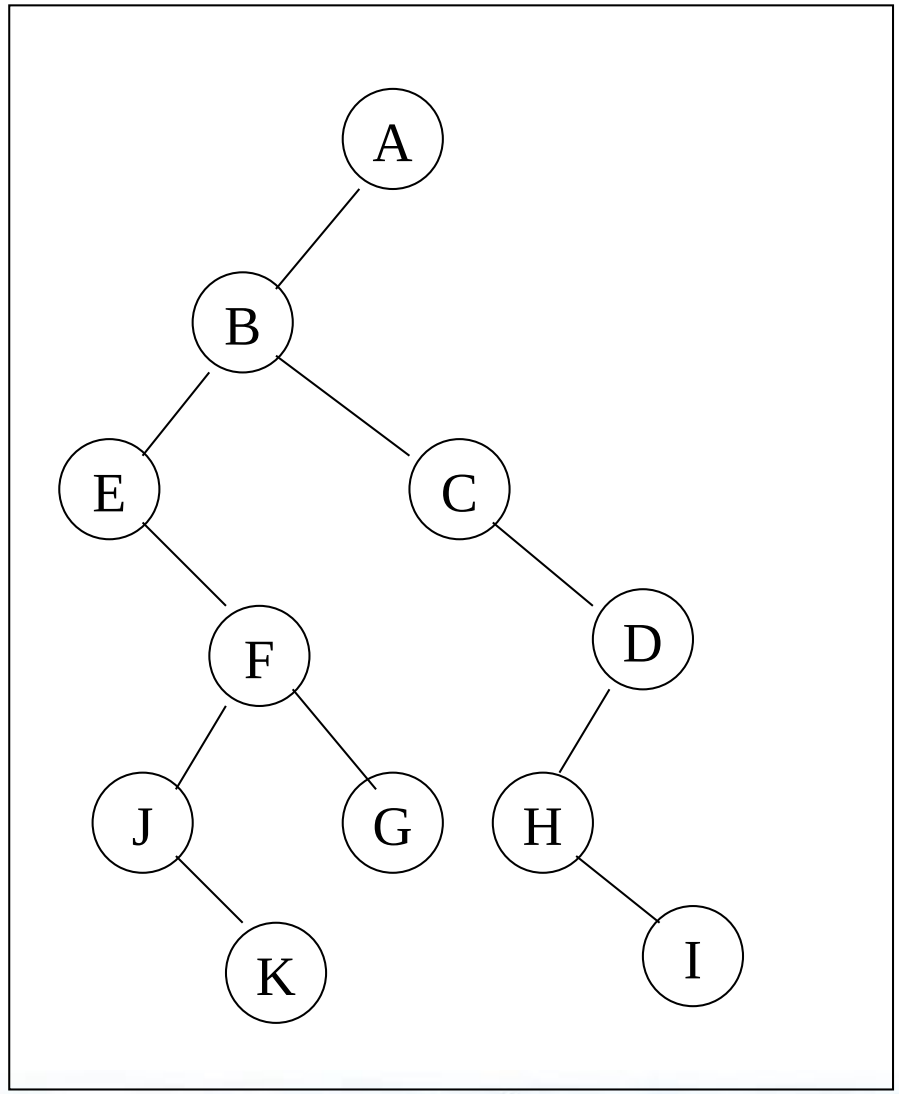
3 、孩子双亲表示法

结合了双亲表示法和孩子链表示法

4 、树的二叉链表（孩子 - 兄弟）表示法

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


A
B
C
D
E
F
G
H
I
J
K


A
B
E
C
F
D
J
G
H
K
I

6.4.2 森林和二叉树的转换

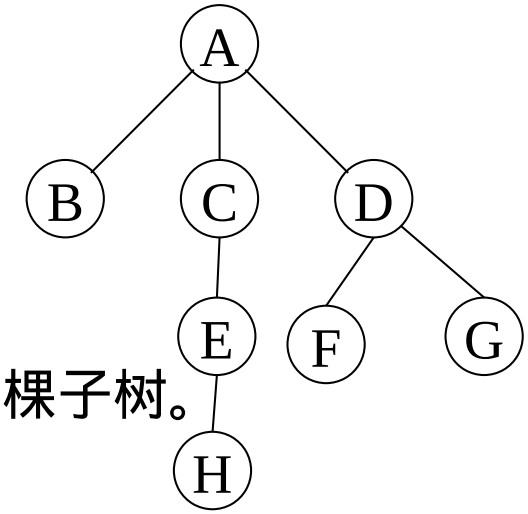
- 树与二叉树的变换
  - 以树的结点为二叉树结点
  - 树根与最左子树的父 - 子关系改为父 - 左子关系，去除根与其他子树的父 - 子关系
  - 将其他各子树根（除最右子树）到其右兄弟之间的隐含关系以父 - 右子关系表示。
  - 对于无右子树的二叉树可以实施逆变换

- 森林到二叉树的变换： F(T1,T2.....Tn)-> 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 （ T1 ）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 {T2 ， T3.....Tn} 转化得到的二叉树 B 的右子树

- 二叉树到森林的变换： B -> F （ T1 ， T2.....Tn ）
  - 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 （ T1 ），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 {T2 ， T3.....Tn}

6.4.3 树和森林的遍历

- 树遍历的三种搜索路径（右图）



  - 先根遍历　　ABCEHDFG
    - 若树为空，则空操作；否则
    - 1）访问根结点；
    - 2）按照从左至右次序先根遍历根结点的每一棵子树。
  - 后根遍历　　BHECFGDA
    - 若树为空，则空操作；否则
    - 1）按照从左至右次序后根遍历根结点的每一棵子树；
    - 2）访问根结点。
  - 层次遍历　　ABCDEFGH
    - 从树的第 1 层开始，自上而下逐层遍历，在同一层中，按从左至右的次序逐个访问结点。

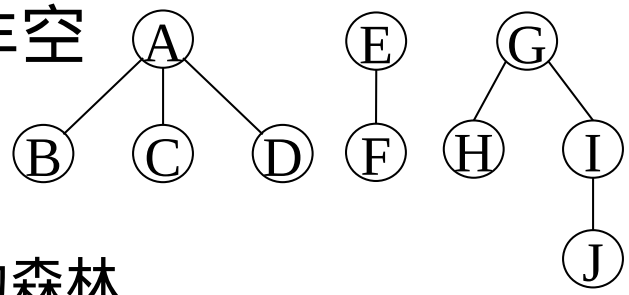
先根遍历树*（孩子链表存储）

```
void PreOrderTree(CTNode T[], int root){
//T[] 是表头结点数组， root 是树根在 T 中的位置下标
if(root==-1) return;   // 空树，空操作
visite(T[root].data);  // 访问根
for(p=T[root].firstchild; p; p=p->next)
    PreOrderTree(T, p->child);   // 依次先根遍历根的各个子树
}//PreOrderTree
```

后根遍历树 *（孩子链表存储）

```
void void PostOrderTree(CTNode T[], int root){
//T[] 是表头结点数组， root 是树根在 T 中的位置下标
    if(root==-1) return;          // 空树，空操作
    for(p=T[root].firstchild; p; p=p->next)
            PostOrderTree(T, p->child);
    visited(T[root].data);  // 访问根
} //PostOrderTree
```

- 对森林遍历的两种方法：（右图）
  - 先序遍历：(ABCDEFGHIJ) 若森林非空
    - 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) 若森林非空
    - 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



- 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

树和森林遍历的应用 *

- 求树的深度

```
int TreeDepth1(CSTree T){
    //T 用孩子兄弟链表存储
    int maxh=0;   //maxh 用来记录 T 所有子树深度的最大值
    if(!T) return 0;    // 空树的深度为 0
    for(p=T->firstchild; p; p=p->nextsibling){
        h=TreeDepth(p);    // 计算一棵子树的深度
        if(h>maxh) maxh=h;  // 求子树深度的最大值
    }
     return maxh+1  ;
} //TreeDepth1
```

```
int TreeDepth2(CTNode T[], int root){
    // 计算树的深度，树用孩子链表存储
    //T 是表头结点数组 , root 是根结点在数组 T 中的位置下标
    int maxh=0;   //maxh 用来记录各个子树深度的最大值
    if(root==-1) return 0;  // 空树的深度为 0
    for(p=T[root].firstchild; p!=NULL; p=p->next){
        h=TreeDepth2(T, p->child);  // 计算子树的深度
        if(h>maxh) maxh=h;  // 求子树深度的最大值
    }
    return 1+maxh;
}// TreeDepth2
```

- 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;
}
```

求树的度

```
void TreeDegree(CSTree T, int &degree)
{ // 孩子兄弟链表存储 ,degree 实参的初值应为 0
    if(!T)return;
    for(k=0,p=T->firstchild; p; p=p->nextsibling){
            k++;
            TreeDegree(p,degree);
    }
    if(k>degree) degree=k;
} //TreeDegree
```

- 例 输出树中从根结点到所有叶子结点的路径 *

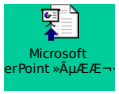
```
void OutPath(CSTree T,Stack &S) {
    while(T){
            Push(S,T->data);
            if(!T->firstchild)StackTraverse(S);
            else OutPath(T->firstchild,S);
            Pop(S,e);
            T=T->nextsibling;
    }
}
```

  - 递归用来遍历左分支，即真正的孩子
  - while 循环用来遍历右分支几兄弟
  - 栈底到栈顶为一条路径

树的层次遍历 *

```
void  TreeLayerOrder (CSTree T){
   // 层序遍历树 T ， T 用孩子兄弟链表存储
    if(!T)return;
    InitQueue(Q);   // 初始化一个空队列 Q
    EnQueue(Q,T);  // 树根入队
    while(!EmptyQueue(Q)){
              DeQueue(Q, s);        // 队头元素 s 出队
              visite （ s->data ）；   // 访问 s
              for(p=s->firstchild; p!=NULL; p=p->nextsibling)
                   EnQueue(Q,p);   //s 的所有孩子依次入队
    }
}// TreeLayerOrder
```

- 例 建立树的存储结构—孩子 - 兄弟链表



```
void CreateTree(CSTree T) {
    T=NULL;
    for(cin>>fa>>ch;ch!='#';cin>>fa>>ch){
        p=new CSNode;
        p->data=ch;p->firstchild=p->nextsibling=NULL;
        EnQueue(Q,p);
        if(fa=='#')T=p;
        else{
                while(GetHead(Q,s) && s->data!=fa){DeQueue(Q,s);}
                if(!s->firstchild){s->firstchild=p;r=p;};
                else{r->nextsibling=p;r=p;}
         }//else
    }//for
} //CreateTree
```

6.5 树的应用 *

6.5.1 堆排序的实现

- 排序的实现：
  - 调整为堆 ，从结点 [n/2] 到 1 依次调整
  - 交换数据，再调整堆顶元素，恢复成堆

typedef SqTable HeapType;

void HeapAdjust(HeapType &H,int s,int m)

- 调整 H.r[s..m] 为大顶堆 , 其中仅 H.r[s] 不满足大顶堆条件

void HeapSort(HeapType &H)

- 从结点 [n/2] 到 1 依次调整，使 H 为大顶堆
- 交换堆顶与堆底记录，再利用 HeapAdjust 重新调整为堆
- 重复进行第二步，直到所有元素有序

```
void HeapAdjust(HeapType &H,int s,int m){
// 调整 H.r[s..m] 为大顶堆 , 其中仅 H.r[s] 不满足条件
    rc=H.r[s];
    for(j=2*s;j<=m;j*=2){
    if(j<m && H.r[j].key<H.r[j+1].key)++j;
    if(rc>=H.r[j].key)break;
    H.r[s]=H.r[j];s=j;
    }//for
    H.r[s]=rc;
}
```

堆排序

```
void HeapSort(HeapType &H){
    for(i=H.length/2;i>0;--i)
            HeapAdjust(H,i,H.length);
    w=H.r[1];H.r[1]=H.r[H.length];H.r[H.length]=w;
    for(i=H.length-1;i>1;--i){ //i 为 2 交换后无需调整
            HeapAdjust(H,1,i);
            w=H.r[1];H.r[1]=H.r[i];H.r[i]=w;
    }//for
}//HeapSort
```

6.5.2 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

```
typedef    TelemType      RcdType;
typedef     KeyType        int;
void Insert_BST(BiTree &T, TElemType e)
```

在二叉排序树中插入一个结点

```
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
}
```

利用二叉排序树排序

```
void BSTSort(SqTable &L) {
    BiTree=NULL;
    for(i=1;i<L.length;i++);
            Insert_BST(T,L.r[i]);
    i=1;
    InOrder(T,Output(T,L,i));
} //BSTSort
Void Output(BiTree T,SqTable L,int &i){
    L.r[i++]=T->data;
}
```

- 补充：求排序二叉树的最大元素

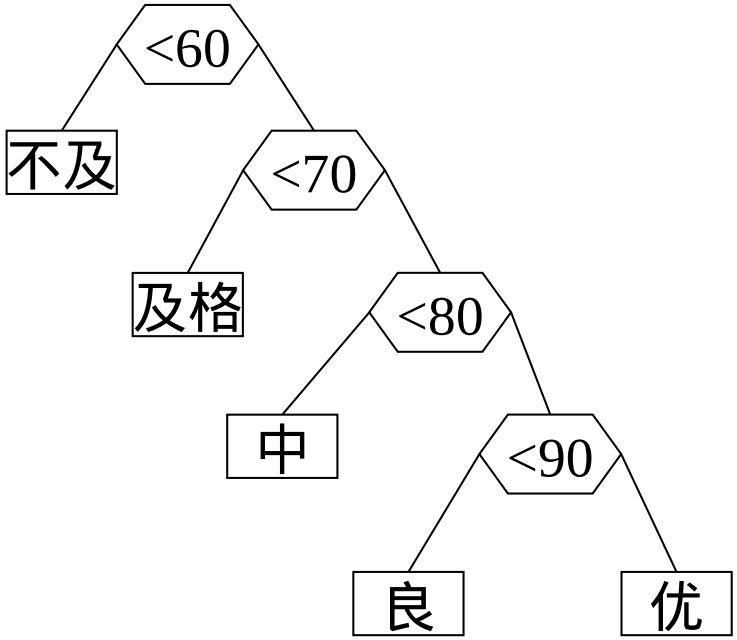
```
keytype getmax(BiTree &T)
{
        if(!T)
        {
                errormessage("NULL Tree,Can't get  value!");
                return -1;
        }
        p=T;
        while(p->rchild)p=p->rchild;
        return p->data.key;
}
```

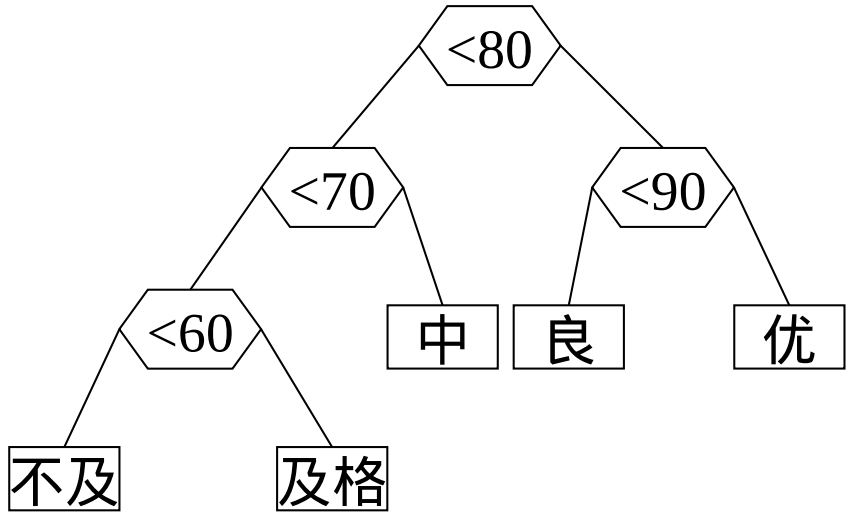
6.6 霍夫曼 (huffman) 树及其应用

- 路径：树中一个结点到另一结点之间的分支
- 路径长度：路径上分支的数目
- 树的路径长度：树根到每个结点的路径长度之和
- 结点带权路径长度：从树根到该结点之间的路径长度与该结点上所带权值的乘积
- 树的带权路径长度： $WPL = \Sigma\omega_k l_k$ （叶子结点）
- 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。

例 将百分制转换为五分制

分数	0~59	60~69	70~79	80~89	90~100
比例	0.05	0.15	0.40	0.30	0.10





- **10000 个输入数据，左图需要判断 31500 次，右图为 22000 次**

- 霍夫曼树算法

  1) 根据给定的 n 个权值 {w1,w2.....wn} ，构成 n 颗二叉树的集合 F = {T1,T2.....Tn} ，每个二叉树只有一个带权 Wi 的根结点。

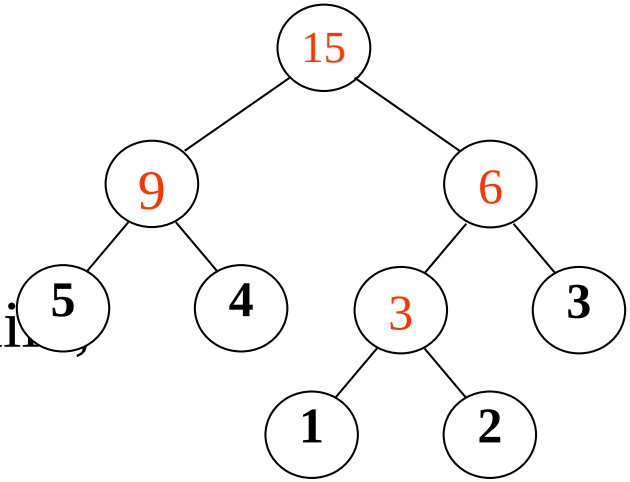
  2) 在 F 中选取两颗根结点的权值最小的树作为左右子树，构造一个新二叉树，且置其根结点的权值为两个子树根结点权值之和。

  3) 在 F 中删除这两颗树，同时在 F 中加入新树。

  4) 重复 2 ）、 3 ）直至 F 只含一颗树为止

- 霍夫曼树存储结构

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



- 创建霍夫曼树

```
void CreateHuffman(HuffmanTree &HT, int *w,int n)
```

	weight	parent	lchild	rchild
0				
1	2	0	0	0
2	4	0	0	0
3	3	0	0	0
4	1	0	0	0
5	5	0	0	0
6	0	0	0	0
7	0	0	0	0
8	0	0	0	0
9	0	0	0	0

Huffman 树初态

	weight	parent	lchild	rchild
0				
1	2	6	0	0
2	4	8	0	0
3	3	7	0	0
4	1	6	0	0
5	5	8	0	0
6	3	7	4	1
7	6	9	3	6
8	9	9	2	5
9	15	0	7	8

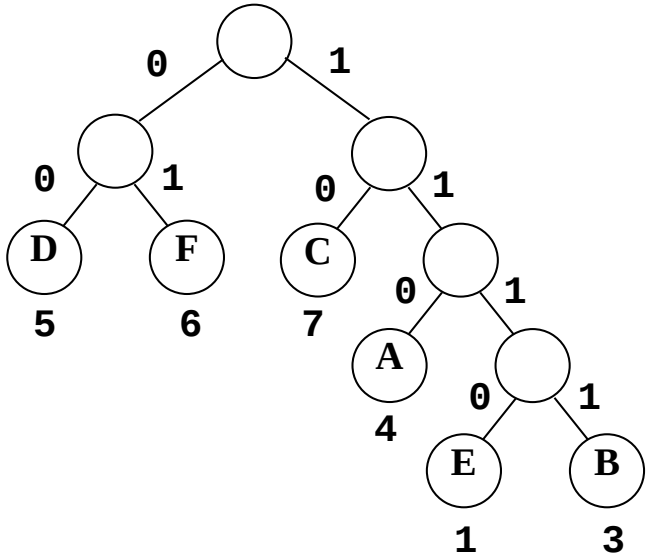
Huffman 树终态

- 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
- 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
- 先序遍历

void HuffmanCoding(HuffmanTree HT,HuffmanCode &HC, int n)　　逆向编码

递归编码

HT	weight	parent	lchild	rchild
0				
1	4	8	0	0
2	3	7	0	0
3	7	10	0	0
4	5	9	0	0
5	1	7	0	0
6	6	9	0	0
7	4	8	5	2
8	8	10	1	7
9	11	11	4	6
10	15	11	3	8
11	26	0	9	10



HC	
0	
1	110\0
2	1111\0
3	10\0
4	00\0
5	1110\0
6	01\0

谢　　谢

递归实现二叉树遍历

```
Void PreOrderTraverse( BiTree T,
   Status(*Visit)(ElemType) )
{
    if (!T) return;
    Visit(T->data);
    PreOrderTraverse(T->lchild, Visit);
    PreOrderTraverse(T->rchild, Visit);
} // PreOrderTraverse
```

非递归实现 *

```
Void InOrder(BiTree BT,void (*visit)(BiTree T)){
    InitStack(S);
    e.ptr=BT;e.task=travel;
    if(BT)push(S,e);
    while(!StackEmpty(S)){
    Pop(S,e);
    if(e.task=Visit)  visit(e.ptr);
    else if(e.ptr){
          p=e.ptr;
          e.ptr=p->rchild;e.task=Travel;Push(s,e);
          e.ptr=p;e.task=Visit;Push(s,e);
          e.ptr=p->lchild;e.task=Travel;Push(s,e);
    }//if
    }//while
}//InOrder
```

```
Status InOrderTraverse(BiTree T, Status (*Visit)(ElemType)) {
// 采用二叉链表存储结构，中序遍历二叉树 T 的非递归算法
  stack S,  BiTree p;
  InitStack(S);  p = T;
  while (p || !StackEmpty(S)) {
    if (p) { Push(S, p);  p = p->lchild; }  // 非空指针进栈，继续左进
    else {       // 上层指针退栈，访问其所指结点，再向右进
      Pop(S, p);
      if (!Visit(p->data)) return ERROR;
      p = p->rchild;
    }
  }
  return OK;
} // InOrderTraverse
```

先序

后序？

表达式求值

```
typedef struct BiTNode{
    double  val;                    // 存放操作数
    char     op;                    // 存放运算符
    unsigned char tag;              // 标志   0: 操作数   1 ：运算符
    struct BiTNode *lchild,*rchild;
}BiTNode, *BiTree;

double Calculate(BiTree T){
    if(T->tag)==0)return T->val;
    a=Calculate(T->lchild);
    b=Calculate(T->rchild);
    return operate(a, T->op, b);
}
```

线索二叉树的中序建立

```
Void InOrderThreading(BiThrTree &H, BiThrTree T){
    H=new BiThrNode;
    H->Ltag=Link; H->Rtag=Thread;
    if(!T){H->lchild=H;}  // 若二叉树空，则左指针回指
    else{
    H->lchild = T;    pre =H;
    InThreading(T,pre);
    pre->rchild = H;  pre->RTag = Thread; //  最后一个结点
                H->rchild = pre;
    }
}//InOrderThreading
```

```
Void InThreading(BiThrTree p, BiThrTree &pre)
{   //p 是当前指针， pre 是跟随指针，比 p 慢一个结点
    if(!p) return;
    Inthreading(p->lchild,pre);
    if(!p->lchild){p->Ltag=Thread;p->lchild=pre;
    if(!pre->rchild){pre->Rtag=Thread;pre->rchild=p;
    pre=p;
    Inthreading(p->rchild,pre);
} //InThreading
```

线索二叉树的非递归遍历

```
Status InOrderTraverse_Thr(BiThrTree T, Status (*Visit)(ElemType)) {  // 算法 6.5
  // T 指向头结点，头结点的左链 lchild 指向根结点，头结点的右链 lchild 指向
  // 中序遍历的最后一个结点。中序遍历二叉线索链表表示的二叉树 T ，
  // 对每个数据元素调用函数 Visit 。
  BiThrTree p;
  p = T->lchild;                    // p 指向根结点
  while (p != T) {                  // 空树或遍历结束时， p==T
    while (p->LTag==Link) p = p->lchild;   // 找最左侧结点
    if (!Visit(p->data)) return ERROR;     // 访问其左子树为空的结点
    while (p->RTag==Thread && p->rchild!=T) {
      p = p->rchild;  Visit(p->data);      // 访问后继结点
    }
    p = p->rchild;  // p 进至其右子树根
  }
  return OK;
} // InOrderTraverse_Thr
```

```
void CreateHuffman(HuffmanTree &HT, int *w,int n){
  m = 2 * n - 1;
  HT = (HuffmanTree)malloc((m+1) * sizeof(HTNode));  // 0 号单元未用
  for (i=1; i<=n; i++) {HT[i].weight=w[i-1];
        HT[i].parent= HT[i].lchild= HT[i].rchild=0;  }
  for (i=n+1; i<=m; i++) { HT[i].weight=0;
         HT[i].parent= HT[i].lchild= HT[i].rchild=0; }
  for (i=n+1; i<=m; i++) {  //  建哈夫曼树
    //  在 HT[1..i-1] 中选择 parent 为 0 且 weight 最小的两个结点，
      // 其序号分别为 s1 和 s2 。
    Select(HT, i-1, s1, s2);
    HT[s1].parent = i;  HT[s2].parent = i;
    HT[i].lchild = s1;  HT[i].rchild = s2;
    HT[i].weight = HT[s1].weight + HT[s2].weight;
   }
}
```

逆向求霍夫曼编码

```
void HuffmanCoding(HuffmanTree HT,HuffmanCode &HC, int n){
  cd = (char *)malloc(n*sizeof(char));    // 分配求编码的工作空间
  cd[n-1] = '\0';                         // 编码结束符。
  for (i=1; i<=n; ++i) {                  // 逐个字符求哈夫曼编码
    start = n-1;                          // 编码结束符位置
    for (c=i, f=HT[i].parent; f!=0; c=f, f=HT[f].parent)
                  if (HT[f].lchild==c) cd[--start] = '0';
                  else cd[--start] = '1';
    HC[i] = (char *)malloc((n-start)*sizeof(char));
    strcpy(HC[i], &cd[start]);    // 从 cd 复制编码 ( 串 ) 到 HC
  }
  free(cd);   // 释放工作空间
}
```

递归求霍夫曼编码 *

```
void HuffmanCoding(HuffmanTree HT,HuffmanCode &HC, int n){
  HC = new (char *)[n]; InitStack(S);
  Coding(HT ， 2*n-1, S);
}
void Coding( HuffmanTree T ， i ， Stack &S) {
    if (!i)return;  // 当前为空结点
    if ( (T.Tree[i].lchild==0) && (T.Tree[i].rchild==0) ) {
      HC[i] = new char[StackLength(S)];
      StackTraverse(S);       //  从栈底到栈顶将栈中字符复制到 HC[i] 中
    }//if
    Push(S, '0');
    Coding( T, T.Tree[i].lchild, S );
    Pop(S, e);
    Push(S, '1');
    Coding( T, T.Tree[i].rchild, S );
    Pop(S, e);
}//Coding
```

无栈非递归求霍夫曼编码 *

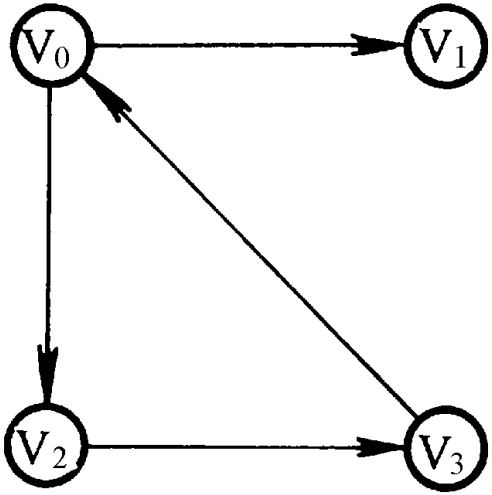
```
void HuffmanCoding( HuffmanTree HT ， int n) {
HC = new (char *)[n];
 cd = (char *)malloc(n*sizeof(char)); //  分配求编码的工作空间
  p = 2*n-1;  cdlen = 0;
  for (i=1; i<=m; ++i) HT[i].weight = 0;    //  遍历哈夫曼树时用作结点状态标志
while (p) {
    if (HT[p].weight==0) {              //  向左  第一次经过
      HT[p].weight = 1;
      if (HT[p].lchild != 0) { p = HT[p].lchild;  cd[cdlen++] ='0'; }
      else if (HT[p].rchild == 0) {     //  登记叶子结点的字符的编码
        HC[p] = (char *)malloc((cdlen+1) * sizeof(char));
        cd[cdlen] ='\0';  strcpy(HC[p], cd);    //  复制编码 ( 串 )
      }
    } else if (HT[p].weight==1) {       //  向右第二次经过
      HT[p].weight = 2;
      if (HT[p].rchild != 0) { p = HT[p].rchild;  cd[cdlen++] ='1'; }
    } else {   // HT[p].weight==2 ，第三次经过，退回退到父结点，编码长度减 1  第三次
       经过
      HT[p].weight = 0; p = HT[p].parent; --cdlen;
    }
  }
} // HuffmanCoding
```

退出循环条件？

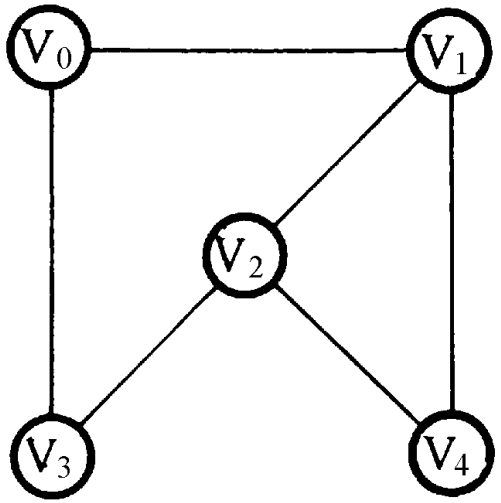
第七章 图

7.1 图的基本概念

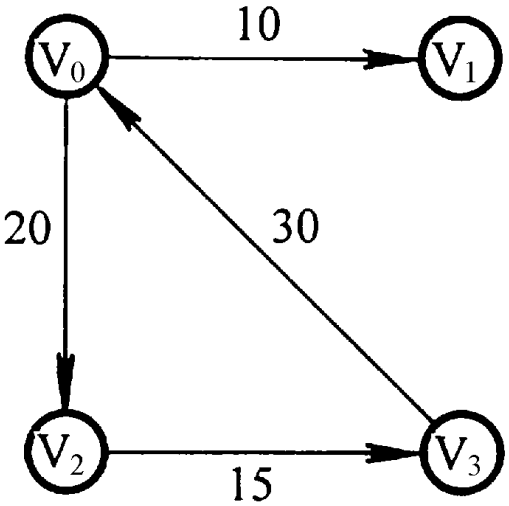
- 图 **(graph)** ：
  - 一个顶点（ vertex ）的有穷集 V(G) 和一个弧 (arc) 的集合 E(G) 组成。记做： G = (V ， E)。 V 是数据结构中的数据元素， E 是集合上的关系
- 弧 **(arc)** 、弧头（终点）、弧尾（起点）：
  - <v,w> 表从 v 到 w 的弧
- 有向图 **(digraph)** 、无向图 **(undigraph)** 、边：
  - (v,w) 代表 <v,w> 和 <w,v>
- 有向网、无向网：
  - 带权的有向图和无向图
- 完全图（ complete graph ）：边 e 为 n(n-1)/2
- 有向完全图：弧 e 为 n(n-1)



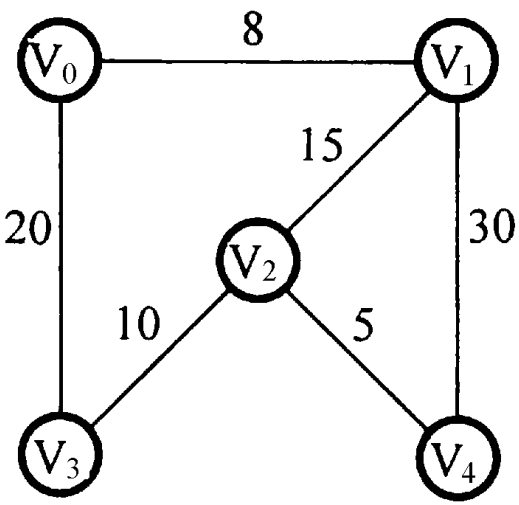
(a) 有向图 G_1



(b) 无向图 G_2



(a) 有向网 G_3



(b) 无向网 G_4

- 稀疏图（ **sparse graph** ）：有向图 e<nlogn
- 稠密图（ **dense graph** ）：有向图 e>nlogn
- 子图（ **subgraph** ）：
  - G=(V,E),G'=(V',E'), 如 V'≦V 且　E≦E', 则称 G' 是 G 的子图
- 度 **(degree)** 、出度 **(OutDegree)** 、入度 **(Indegree):**
  - <u,v> 称 u 邻接到 v ，或 v 邻接自 u 。邻接到某顶点的弧的数目称该顶点的入度 ID(v) ；邻接自某顶点的弧的数目称该顶点的出度 OD(u) ；某顶点的入度、出度之和为该顶点的度 TD(v)
- 路径和回路：
  - 有向路径 / 无向路径，路径长度、回路或环
- 连通图和连通分量：
  - 连通图 (无向)，强连通图（有向），连通分量
- 生成树、生成森林：
  - 连通图的生成树是极小连通子图。
  - 有向图的生成森林由若干有向树组成，含有图中全部顶点和部分足以构成若干颗不相交有向树的弧。

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∈V 且 P(v,w)，其中 <v,w> 表示从 v 到 w 的狐，谓词 P(v,w) 表示狐 <v,w> 的信息 }

基本操作：

1) CreateGraph(&G, V, E)

2) DestroyGraph(&G)

3) LocateVex(G ， u)

4) GetVex(G ， v)

5) PutVex(&G ， v ， value)

6) FirstAdjVex(G ， v)

7) NextAdjVex(G ， v ， w)

8) InsertVex(&G ， v)

9) DeleteVex(&G ， v)

10) InsertArc(&G ， v ， w)

11) DeleteArc(&G ， v ， w)

12) DFSTraverse(G ， v ， visit())

13) BFSTraverse(G ， v ， visit())

} ADT Graph

7.2 图的存储

- 图的数组（邻接矩阵）存储表示

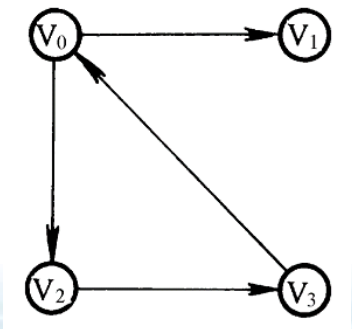
```
typedef  enum{DG ， DN ， AG ， AN} GraphKind ；
typedef  struct ArcCell{
    VRType          adj;
    InfoType        *info;
}ArcCell,AdjMatrix[MAX_V_NUM][ MAX_V_NUM] ；
typedef struct{
    VertexType              vexs[MAX_V_NUM];
    AdjMatrix               arcs;
    int                     vexnum,arcnum;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

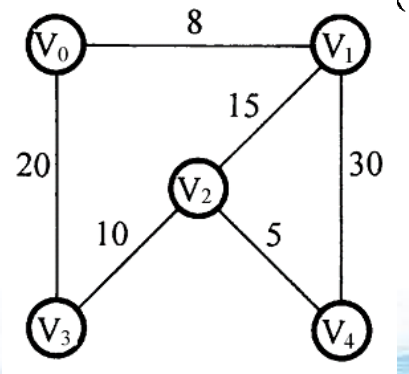
$$A_1=\begin{pmatrix}0&1&1&0\\0&0&0&0\\0&0&0&1\\1&0&0&0\end{pmatrix}$$

(a) G_1 的邻接矩阵

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

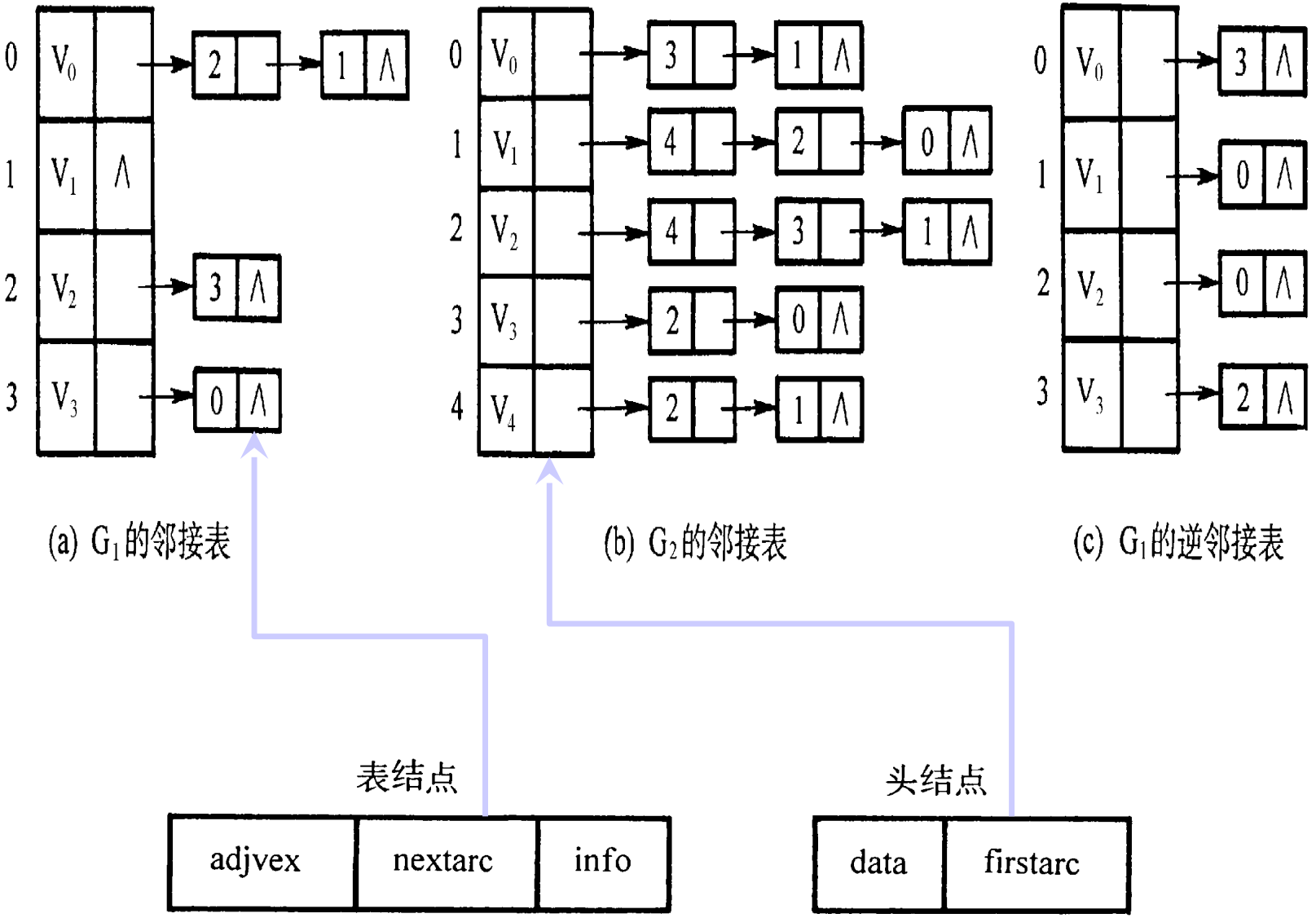
(b) G_4 的邻接矩阵





7.2.2 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;
```


0 V0 2 1 ∧
1 V1 ∧
2 V2 3 ∧
3 V3 0 ∧
(a) G1的邻接表
0 V0 3 1 ∧
1 V1 4 2 0 ∧
2 V2 4 3 1 ∧
3 V3 2 0 ∧
4 V4 2 1 ∧
(b) G2的邻接表
0 V0 3 ∧
1 V1 0 ∧
2 V2 0 ∧
3 V3 2 ∧
(c) G1的逆邻接表
表结点
adjvex
nextarc
info
头结点
data
firstarc

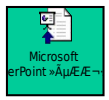
创建邻接矩阵存的无向网

```
Status CreateUDN(MGraph &G) {
// 采用数组（邻接矩阵）表示法，构造无向网 G。
  scanf("%d,%d,%d",&G.vexnum, &G.arcnum, &IncInfo);
  for (i=0; i<G.vexnum; i++ ) scanf("%c",&G.vexs[i]);
  for (i=0; i<G.vexnum; ++i )  //  初始化邻接矩阵
    for (j=0; j<G.vexnum; ++j ) {
      G.arcs[i][j].adj = INFINITY; //{adj,info}
      G.arcs[i][j].info= NULL;
    }
    for (k=0; k<G.arcnum; ++k ) {  //  构造邻接矩阵
    scanf(&v1, &v2, &w); //  输入一条边依附的顶点及权值
    i = LocateVex(G, v1);  j = LocateVex(G, v2);    //  确定 v1 和 v2 在 G 中位置
    G.arcs[i][j].adj = w;                 //  弧 <v1,v2> 的权值
    if (IncInfo) scanf(G.arcs[i][j].info); //  输入弧含有相关信息
    G.arcs[j][i].adj = G.arcs[i][j].adj; //  置 <v1,v2> 的对称弧 <v2,v1>
  }
   return OK;
} // CreateUDN
```

7.3 图的遍历

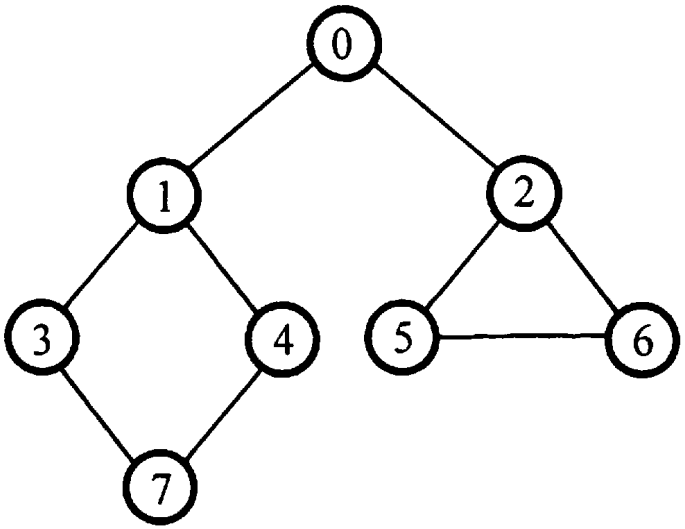
7.3.1 深度优先遍历

- 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。
  - 算法　void DFS（ Graph G ， int 
  - DFS 生成树

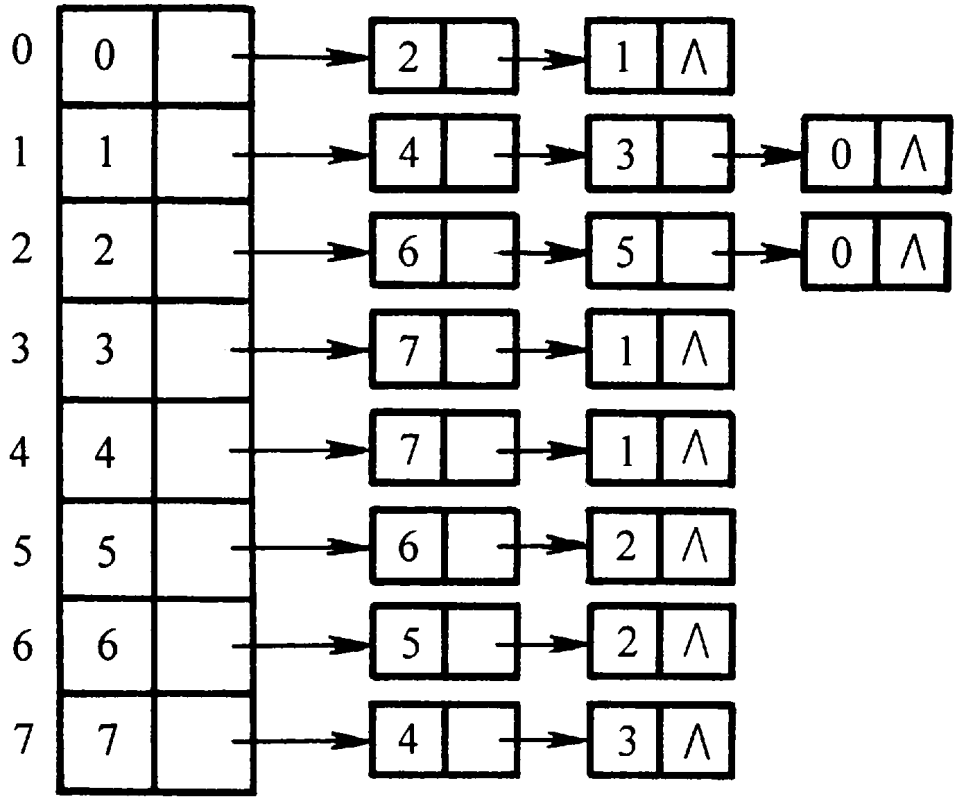


7.3.2 广度优先遍历

- 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
  - 算法 void BFSTraverse （ Graph G ,int v ）
  - BFS 生成树



(a) 无向图 G_5



(b) G_5的邻接表

深度优先遍历的序列是：**0，2，6，5，1，4，7，3**

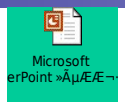
广度优先遍历的序列是：**0，2，1，6，5，4，3，7**

7.4 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
- 非连通图的生成森林
- 连通网的最小生成树：权值和最小的生成树
- 求连通网最小生成树的算法
  - 克鲁斯卡尔（ Kruskal ）算法 复杂度： O(eloge)
  - 普里姆（ Prim ）算法 复杂度： O(n*n)
  - 算法比较：当 e （边）与 n*n 差不多时，采用 Prim 算法快；当 e 远小于 n*n 时，采用 Kruskal 算法快

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

– 算法描述：

1). 设 G（V，E），把 V = {1，2，.....n} 看成孤立的 n 个连通子图。边按照权的非递减次序排列。

2). 顺序查看边。对于第 k 条边（v，w），如果 v，w 分别属于两个连通字图 T1、T2，则用边（v、w）将 T1、T2 连成一个连通字图。

3). 重复 2，直至 n 个结点同属于一个连通图

普里姆算法（ Prim ）

- 算法思想　复杂度 O(n*n)

  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

- 算法描述

  1. 设 G = (V , E) ，权 A[m,n] ，令 U = {1}。

  2. if (U≦V) 取 min(A[i,j]) ，使 i∈U ， j∈V - U。

  3. 将 j 加入 U

  4. 重复 2 、 3 ，直至 U = V

7.5 有向无环图及其应用

7.5.1 拓扑排序

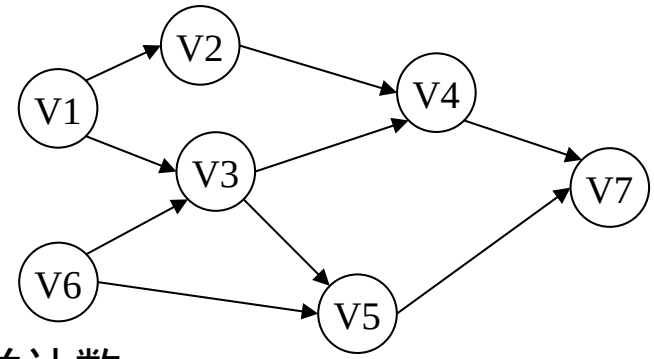
- 活动顶点网络（ AOV ， activity on vertex ）
  - 以顶点表示活动，以弧表示活动之间的优先制约关系的有向图。
- 死锁：
  - AOV 中不允许出现回路，回路意味某活动以自己的结束作为开始的先决条件。称为死锁。
- 拓扑排序、拓扑有序序列
  - 若在有向图中从 u 到 v 有一条弧，则在序列中 u 排在 v 之前，称有向图的这个操作为拓扑排序。所得序列为拓扑有序序列。若有死锁，则无法获得拓扑有序序列

- 操作方法：
  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
- 统计有向图邻接表各顶点的入度

```
void FindIndegree(ALGraph G){
    for(i=0;i<G.vexnum;i++)indegree[i]=0;
    for(i=0;i<G.vexnum;i++){
    p=G.vertices[i].firstarc;
    while(p){
    indegree[p->adjvex]++;
    p=p->nextarc;
    }//while
    }//for
} //FindIndegree
```

拓扑排序 (复杂度 O(n+e))

```
Status TopologicalSort(ALGraph G) {  // 算法 7.12
  char indegree[MAX_VERTEX_NUM];
  FindInDegree(G, indegree);   // 对各顶点求入度 indegree[0..vernum-1]
  InitStack(S);
  for (i=0; i<G.vexnum; ++i)       // 建零入度顶点栈 S
    if (!indegree[i]) Push(S, i);  // 入度为 0 者进栈
  count = 0;                       // 对输出顶点计数
  while (!StackEmpty(S)) {
    Pop(S, i);
    printf(i, G.vertices[i].data);  ++count;  // 输出 i 号顶点并计数
    for (p=G.vertices[i].firstarc;  p;  p=p->nextarc) {
      k = p->adjvex;               // 对 i 号顶点的每个邻接点的入度减 1
      if (!(--indegree[k])) Push(S, k);  // 若入度减为 0 ，则入栈
    }
  }
  if (count<G.vexnum) return ERROR;      // 该有向图有回路
  else return OK;
} // TopologicalSort
```



抽象方法
如何改

写出右图的拓扑排序序列：

7.5.2 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

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

  ve（ j ）= 0 （ j = 1 ）

  = max{ve(i)+w(Vi->Vj)} (j=2,3.....n)

  2) 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。

  vl（ i ）= ve（ n ） （ i = n ）

  = min{vl(j)-w(Vi->Vj)} (k=1,2,3... ...m)

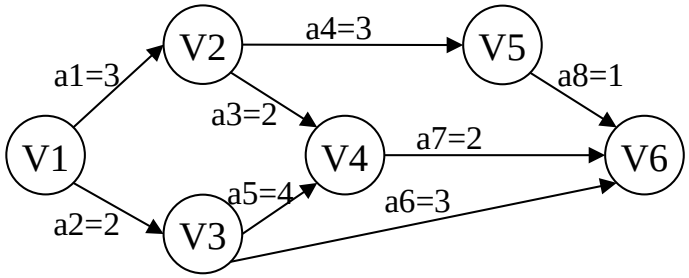
  3) ee（ k ）: 活动 ak（ Vi->Vj ）可能开始的最早时刻。 ee(k)=ve(i)

  4) 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。

  el（ k ）= vl（ j ） - w(Vi->Vj) (k=1,2,3... ...m)

- 若某弧 ak 的 el（ k ）和 ee（ k ）相等，则 ak 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	Vl
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

7.6 最短路径

- 迪杰斯特拉 (Dijkstra) 算法

1）设 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

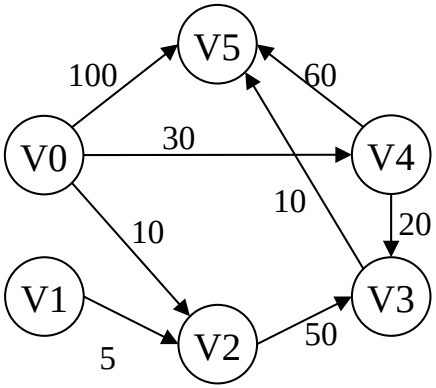
D[k] = arcs [i,k] (i 为源点，k = 0，1，… n)

2) 选择 vj，使得 D[j] = min{D[i]|vi!∈S，vi∈V（G）}，vj 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 arcs[j,k] 为有限值，即 j，k 有弧存在，且

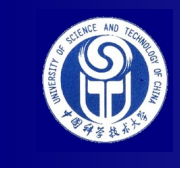
D [j]+arcs[j,k]<D[k] 则令 D [k]= Dist[j]+arcs[j,k] 。

4) 重复 2）、3）直至求得源点到所有终点的最短路径



	V0	V1	V2	V3	V4	V5
V0	∞	∞	10	∞	30	100
V1	∞	∞	5	∞	∞	∞
V2	∞	∞	∞	50	∞	∞
V3	∞	∞	∞	∞	∞	10
V4	∞	∞	∞	20	∞	60
V5	∞	∞	∞	∞	∞	∞

终点	i=1		i=2		i=3		i=4		i=5	
V1	∞		∞		∞		∞		∞	
V2	10	{v0,v2}								
V3	∞		60	{v0,v2,v3}	50	{v0,v4,v3}				
V4	30	{v0,v4}	30	{v0,v4}						
V5	100	{v0,v5}	100	{v0,v5}	90	{v0,v4,v5}	60	{v0,v4,v3,v5}		
Vj	V2		V4		V5		V5			
S	{v0,v2}		{v0,v2,v4}		{v0,v2,v4,V3}		{v0,v2,v4,V3,v5}			



谢　　谢

DFS

```
void DFSTraverse(Graph G, Status (*Visit)(int v)) {
    VisitFunc = Visit;   // VisitFunc 全局变量
    for (v=0; v<G.vexnum; ++v) visited[v] = false;
    for (v=0; v<G.vexnum; ++v)
    if (!visited[v]) DFS(G, v);       //  对尚未访问的顶点调用 DFS
}

void DFS(Graph G, int v) {
  visited[v] = true;   VisitFunc(v);  //  访问第 v 个顶点
  for (w=FirstAdjVex(G, v);  w!=-1;  w=NextAdjVex(G, v, w))
    if (!visited[w])   //  对 v 的尚未访问的邻接顶点 w 递归调用 DFS
      DFS(G, w);
}
```

BFS

```
void BFSTraverse(Graph G, Status (*Visit)(int v )) {
  for (v=0; v<G.vexnum; ++v) visited[v] = FALSE; // 初始化标志数组
  InitQueue(Q);                    //  置空的辅助队列 Q
  for (v=0;  v<G.vexnum;  ++v)
    if (!visited[v]) {             // v 尚未访问
      visited[v] = TRUE;  Visit(v); //  访问 v
      EnQueue(Q, v);               // v 入队列
      while (!QueueEmpty(Q)) {
        DeQueue(Q, u);             //  队头元素出队并置为 u
        for (w=FirstAdjVex(G, u);  w>=0;  w=NextAdjVex(G, u, w))
          if (!visited[w]) {       // u 的尚未访问的邻接顶点 w 入队列 Q
            visited[w] = TRUE;  Visit(w);
            EnQueue(Q, w);
          }//if
      }//while
    }//if
} // BFSTraverse
```

PRIM 算法

```
void MiniSpanTree_PRIM(MGraph G, VertexType u) {  // 算法 7.9
  //struct {VertexType adjvex;VRType lowcost;}closedge[MAX_V_NUM];
  k = LocateVex ( G, u );
  for ( j=0; j<G.vexnum; ++j ) // 辅助数组初始化
     if (j!=k) closedge[j] ={u, G.arcs[k][j].adj };
  closedge[k].lowcost = 0;      // 初始，U = {u}
  for (i=1; i<G.vexnum; ++i) {  // 选择其余 G.vexnum-1 个顶点
    k = minimum(closedge);      // 求出 T 的下一个结点：第 k 顶点
    printf(closedge[k].adjvex, G.vexs[k]);  // 输出生成树的边
    closedge[k].lowcost = 0;    // 第 k 顶点并入 U 集
    for (j=0; j<G.vexnum; ++j)
      if (G.arcs[k][j].adj < closedge[j].lowcost)
                  closedge[j] = { G.vexs[k], G.arcs[k][j].adj };
  }//for
} // MiniSpanTree
```

Dijkstra 算法

```
void ShortestPath_DIJ(MGraph G,int v0,int P[][MAX_V_NUM],int D[MAX_V_NUM]){
 for (v=0; v<G.vexnum; ++v) {
   S[v] = 0; D[v] = G.arcs[v0][v].adj;
   if (D[v] != INFINITY) { P[v][0] =v0; P[v][1] = v;P[v][2]=-1; }//-1 表示结束
  }
  D[v0] = 0;  S[v0] =1;       //  初始化，v0 顶点属于 S 集
  for (i=1; i<G.vexnum; ++i) {        //  其余 G.vexnum-1 个顶点
   min = INFINITY;                  //  当前所知离 v0 顶点的最近距离
   for (w=0; w<G.vexnum; ++w)
    if (!S[w]&&D[w]<min) { v = w;  min = D[w]; }  // w 顶点离 v0 顶点更近
   S[v] = 1;                                      //  离 v0 顶点最近的 v 加入 S 集
   for (w=0; w<G.vexnum; ++w)           //  更新当前最短路径及距离
    if (!S[w] && (min+G.arcs[v][w].adj<D[w])) {  //  修改 D[w] 和 P[w], w∈V-S
      D[w] = min + G.arcs[v][w].adj;
      for(j=0; P[v][j]!=-1;j++) P[w][j] = P[v][j]; // 第 v 行赋值于第 w 行
      P[w][j++] = w; P[w][j]=-1;
    }//if
  }//for
} // ShortestPath_DIJ
```

第十章 排序

10.1 排序的基本概念

10.2 简单排序方法（复杂度 O(n2)）

10.3 先进排序方法（复杂度 O(nLogn) /$O(n^{3/2})$）

10.4 基数排序（复杂度 O(d×n)）

10.5 各种排序的综合比较

10.1 排序的基本概念

- 排序单元结构定义

```
typedef int       KeyType;   // 简单起见，只要可比就可以
typedef   struct{
KeyType           key;
InfoType          otherinfo;
}RcdType;
```

- 例：

```
Typedef struct {
char name[10];
char  sex[2];
char birthday[8];
char department[4];
}InfoType
```

- 排序

  设 N 个记录 $\{r_1,r_2\ldots r_n\}$ ， 相应关键字 $\{k_1,k_2\ldots k_n\}$

  求一种排列 $p_1,p_2\ldots p_n$ ，使得 $k_{p1}\le k_{p2}\le\ldots\le k_{pn}$

  即 $\{r_{p1},r_{p2}\ldots r_{pn}\}$ 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：

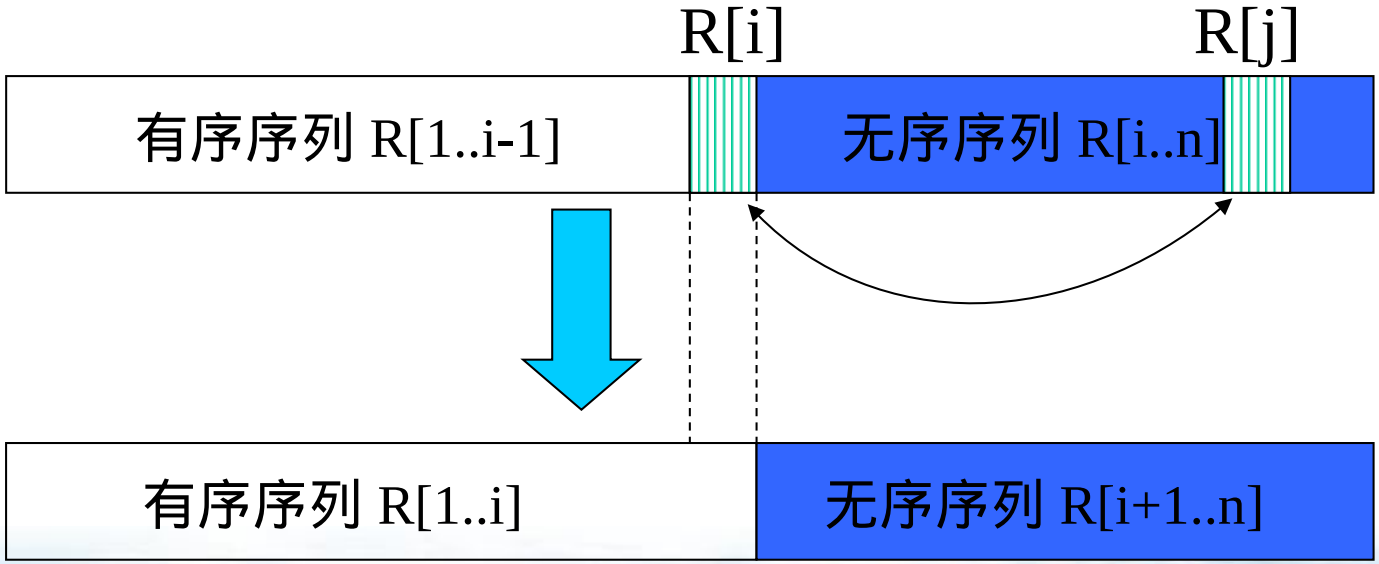
```
Typedef struct{
        RcdType r[MAXSIZE+1];
        Int     length;
}Sqlist;
```

10.2.1 选择排序

void SelectPass （ Sqlist &L,int i ）复杂度 O(n)

void SelectSort （ Sqlist &L ）复杂度 $O(n^2)$

- n(n + 1)/2 次比较和 3 （ n-1 ）次交换。
- 本身是稳定算法，本例由交换引起不稳定，可采用移动策略获稳定算法



选择排序 (一趟)

```
void SelectPass( SqList &L, int i ) {
    RcdType  W ;
    j = i ;          //j 为最小值位置
    for ( k=i+1; k<=L.length; k++ )
    if ( L.r[k].key < L.r[j].key )  j = k   ;
    if ( i != j )
                { W=L.r[j]  ;  L.r[j] =L.r[i]  ;  L.r[i] =
    W ; }
} // SelectPass
```

选择排序

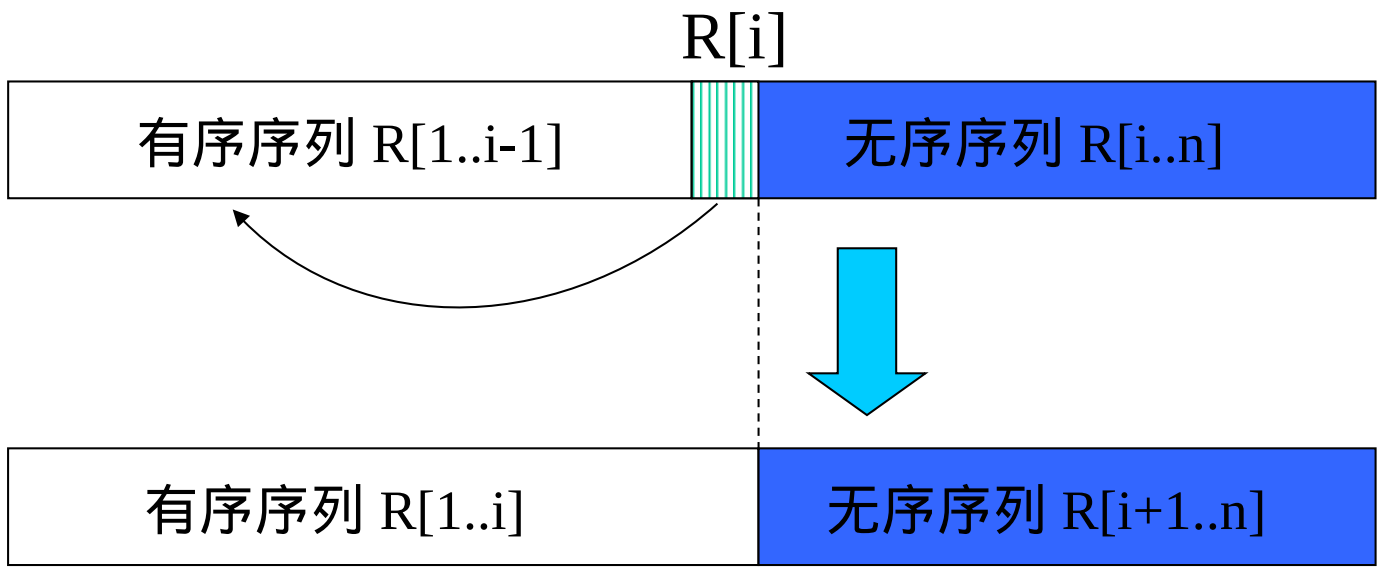
```
void SelectSort (SqList &L) {
        RcdType  W  ;
    for (i=1; i<L.length; ++i) {
      j = i  ;
      for ( k=i+1; k<=L.length; k++ )
          if ( L.r[k].key < L.r[j].key )  j =k   ;
      if ( i!=j ) { W=L.r[j];L.r[j] =L.r[i];L.r[i] = W;}
    }
} // SelectSort
```

10.2.2 插入排序

void InsertPass(Sqlist &L,int i) 复杂度　O(n)

void InsertSort(Sqlist &L) 复杂度　$O(n^2)$

- 数据的有序性影响算法 最差比较移动 (n+4)(n-1)/2
- 是稳定排序



插入排序 (一趟)

```
void InsertPass( SqList &L, int i ) {
    L.r[0] = L.r[i];          //  复制为哨兵
    for ( j=i-1;  L.r[0].key < L.r[j].key;  --j )
      L.r[j+1] = L.r[j];        //  记录后移
    L.r[j+1] = L.r[0];          //  插入到正确位置
} // InsertPass
```

插入排序

```
void InsertSort ( SqList &L) {
    //  对顺序表 L 作插入排序
    for ( i=2; i<=L.length; ++i )
      if ( L.r[i].key < L.r[i-1].key )
        L.r[0] = L.r[i];                    //  复制为哨兵
        for ( j=i-1;  L.r[0].key < L.r[j].key;  --j )
          L.r[j+1] = L.r[j];                //  记录后移
        L.r[j+1] = L.r[0];                  //  插入到正确位置
      } // if
} // InsertSort
```

其他插入排序

折半插入：

- 插入时因为有序区可以使用折半查找，从而提高插入速度

2 路插入：

- 为了减少移动，分两个半区插入，可以和折半一起使用

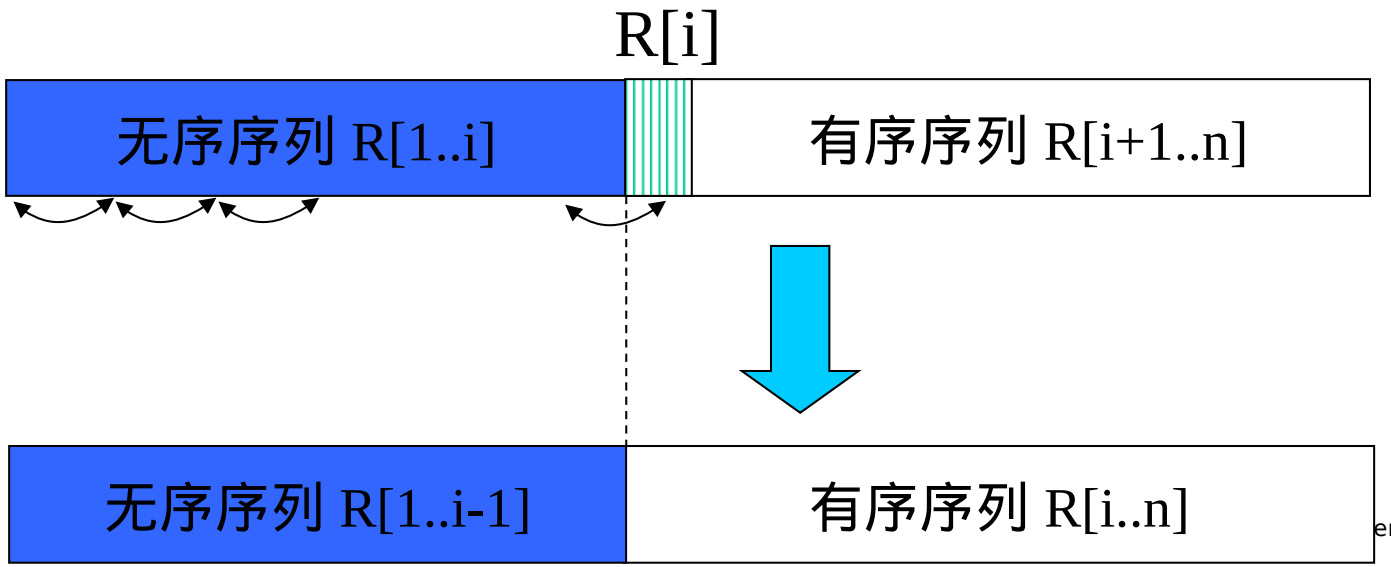
表插入：

- 通过静态链表形式存储数据元素，通过修改指针的方式避免元素的移动，不可以使用折半插入。

10.2.3 起泡排序

void BubbleSort （ Sqlist &L ）复杂度 $O(n^2)$

- 有序序列的比较次数是 n-1 次
- 逆序序列的比较次数是 n(n-1)/2 ；移动次数 3n(n-1)/2
- 是稳定排序



起泡排序

```
void BubbleSort( SqList &L ){
    i = L.length;
    while (i >1) {
      lastExchangeIndex = 1;
      for (j = 1; j < i; j++){
        if (L.r[j+1].key < L.r[j].key) {
          W=L.r[j] ; L.r[j] =L.r[j+1] ; L.r[j+1] = W ;
          lastExchangeIndex = j;
        } //if
      } //for
      i = lastExchangeIndex;
    } // while
} // BubbleSort
```

10.3 先进排序方法

10.3.1 快速排序

void Partition　一趟快速排序（分割）

void Qsort（ RcdType R[],int s,int t ）递归函数

void QuickSort（ SqList &L ）快速排序

- 时间复杂度　O(nLogn)　最坏情况有序 $O(n^2)$
- 空间复杂度　O(logn)
- 改进：取中原则： R[s],R[t],R[(s+t)/2]

Microsoft
erPoint »ÃµÆÆ¬·

快速排序 (分割)

```
int Partition ( RcdType R[], int low, int high) {
    R[0] = R[low];            //  将枢轴记录移至数组的闲置分量
    pivotkey = R[low].key;    //  枢轴记录关键字
    while (low<high) {        //  从表的两端交替地向中间扫描
      while(low<high&& R[high].key>=pivotkey) --high;
      R[low++] = R[high];     //  将比枢轴记录小的记录移到低端
      while (low<high && R[low].key<=pivotkey)  ++low;
      R[high--] = R[low];     //  将比枢轴记录大的记录移到高端
    } //while
    R[low] = R[0];            //  枢轴记录移到正确位置
    return low;               //  返回枢轴位置
} // Partition
```

快速排序

```
void QSort (RedType R[],  int s,  int t ) {
      //  对记录序列 R[s..t] 进行快速排序
      if (s < t) {                  //  长度大于 1
        pivotloc = Partition(R, s, t);//  对   R[s..t] 一分为二
        QSort(R, s, pivotloc-1);      //  对低子序列递归排序
        QSort(R, pivotloc+1, t);      //  对高子序列递归排序
      } // if
} // Qsort

void QuickSort( SqList & L) {
      QSort(L.r, 1, L.length);
} // QuickSort
```

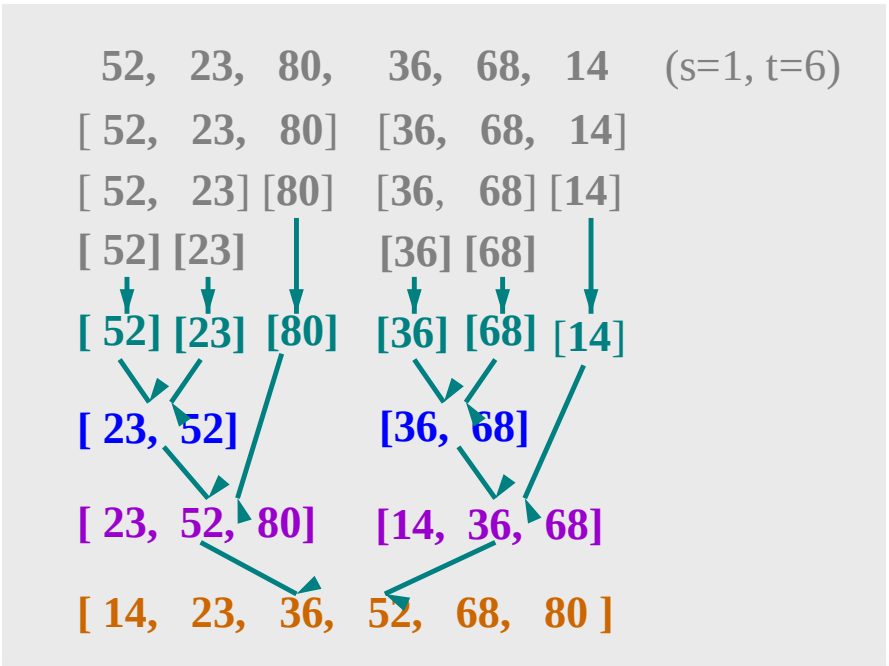
10.3.2 归并排序

void Merge (RcdType SR[], RcdType TR[], int i, int m, int n)

void Msort (RcdType SR[], RcdType TR1[], int s, int t) 递归函数

void MergeSort （ SqList &L ）快速排序

- 时间复杂度 O(nLogn)
- 空间复杂度 O(n)



归并排序 (一趟)

```
void Merge (RcdType SR[], RcdType TR[], int l, int
    m, int n) {
     //  将有序的 SR[l..m] 和 SR[m+1..n] 归并为有序
    的 TR[l..n]
     for (i=l,j=m+1, k=l;  i<=m && j<=n;  ++k) {
       if (SR[i].key<=SR[j].key)  TR[k] = SR[i++];
       else TR[k] = SR[j++];
     }
     while (i<=m) TR[k++] = SR[i++];
     while (j<=n) TR[k++] = SR[j++];
} // Merge
```

归并排序

```
void Msort ( RcdType SR[], RcdType TR1[], int s, int t ) {
    //  对 SR[s..t] 进行归并排序，排序后的记录存入 TR1[s..t] 。
    if (s==t)  TR1[s] = SR[s];
    else {
      m = (s+t)/2;
      Msort (SR, TR2, s, m);
      Msort (SR, TR2, m+1, t);
      Merge (TR2, TR1, s, m, t);
    } // else
} // MSort
void MergeSort (SqList &L) {
     MSort(L.r, L.r, 1, L.length);
} // MergeSort
```

2 路归并排序程序

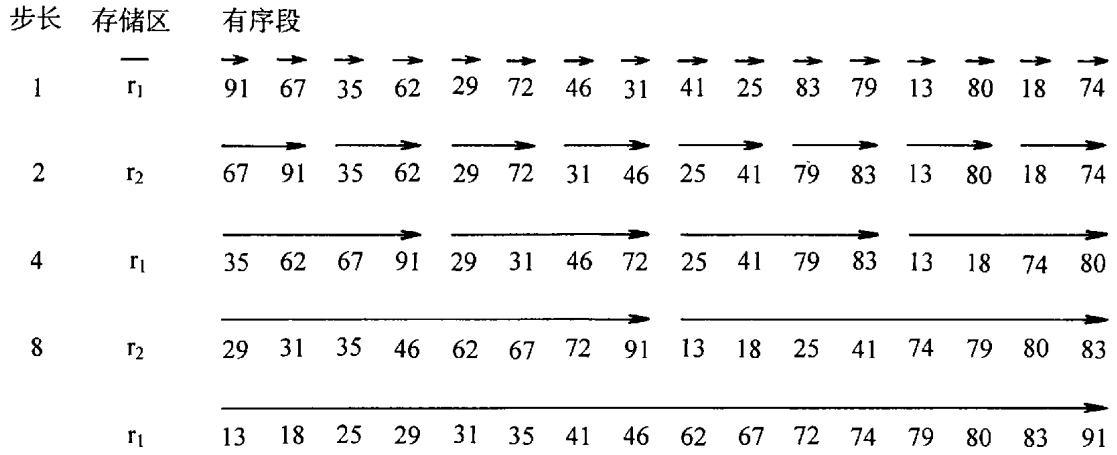
1 ）一趟排序

```
void mergepass(RcdType SR[],RcdType TR[],int s,int n){
    i=1;
    while(i+2s-1<=n) {
            merge(SR,TR,i,i+s-1,i+2*s-1);
            i +=2*s;
}
if(i+s-1<n)          // 最后还有两个集合，但不等长
    merge(SR,TR,i,i+s-1,n);
else                 // 最后还有一个集合
    while(i< = n){TR[i]=SR[i];i++};
}
```

2 路归并排序程序（ 2 ）

2 ） 2 路归并

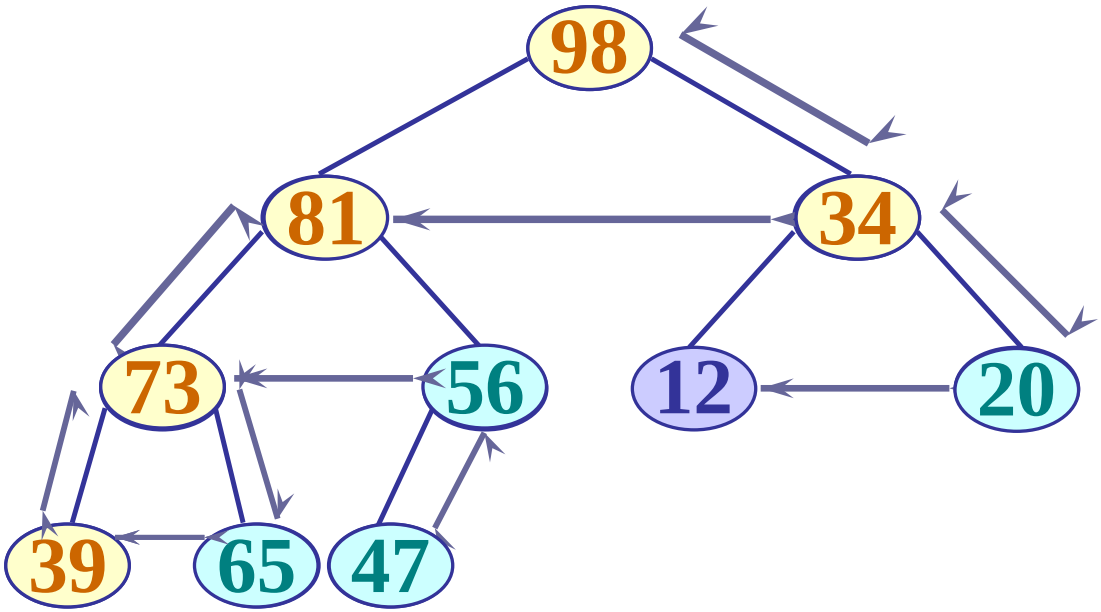
```
void mergesort((Sqlist &L){
    RcdType TR[L.length+1];
    s=1;
    while （ s<n ）{
            mergepass(L.r,TR,s,n);      s*=2;
            mergepass(TR.L.r.s.n):      s*=2:
    }
}
```



10.3.3 堆排序

小顶堆 : $r_i<r_{2i}$； $r_i<r_{2i+1}$　　大顶堆　:$ri>r_{2i}$， $r_i>r_{2i+1}$

- 时间复杂度　O(nLogn)　　空间复杂度　O(1)

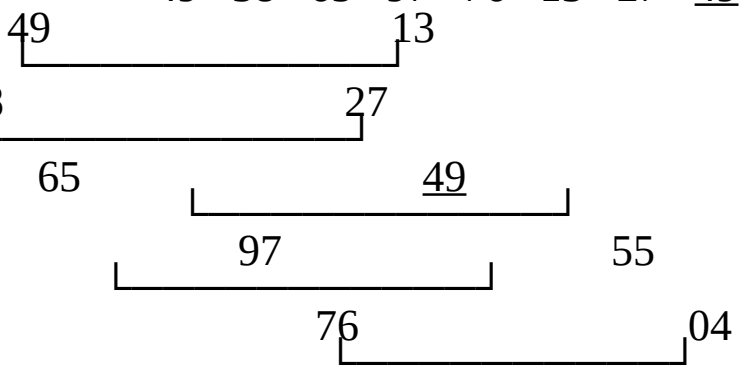


```
void HeapAdjust(HeapType &H, int s, int m) {  // 算法 10.10
  // 已知 H.r[s..m] 中记录的关键字除 H.r[s] 之外均满足堆的定义，
  rc = H.r[s];
  for (j=2*s; j<=m; j*=2) {   // 沿 key 较大的孩子结点向下筛选
    if (j<m && H.r[j].key<H.r[j+1].key) ++j; // j 为 key 较大的记录的下标
    if (rc.key >= H.r[j].key) break;      // rc 应插入在位置 s 上
    H.r[s] = H.r[j];  s = j;
  }//for
    H.r[s] = rc;  // 插入
} // HeapAdjust
void HeapSort(HeapType &H) {
     for (i=H.length/2; i>0; --i) HeapAdjust ( H, i, H.length );
     for (i=H.length; i>1; --i) {H.r[i] ↔ H.r[1]     HeapAdjust(H, 1, i-1); }
} // HeapSort
```

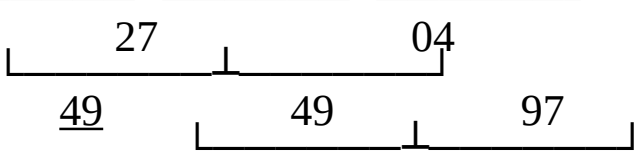
10.3.4 希尔排序

- 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

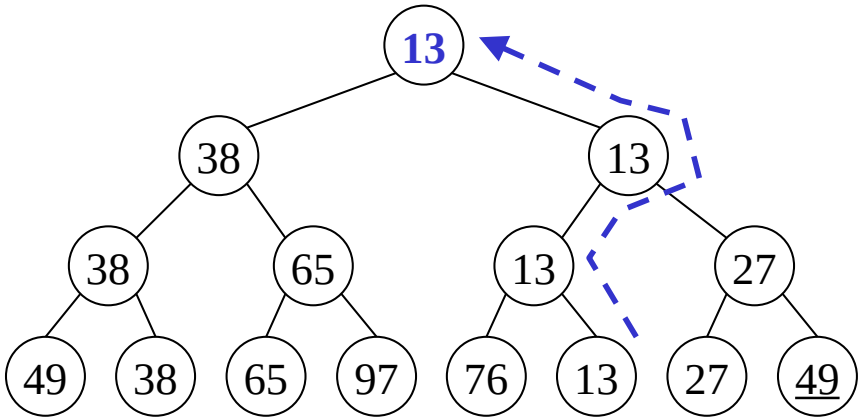
三趟排序： 04 13 27 38 49 49 55 65 76 97

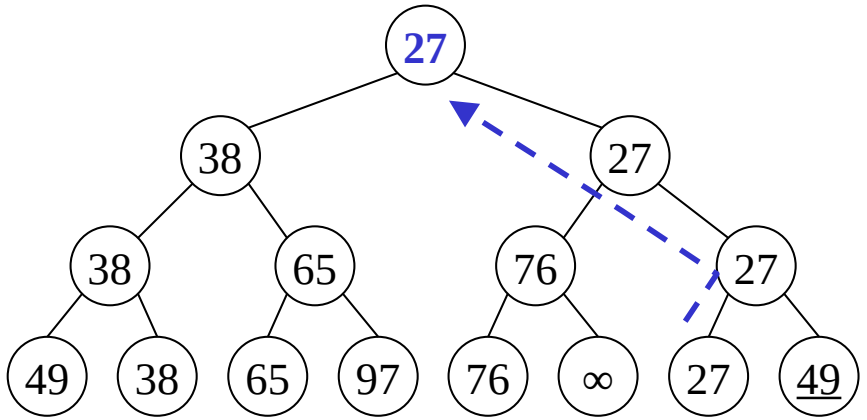
希尔排序

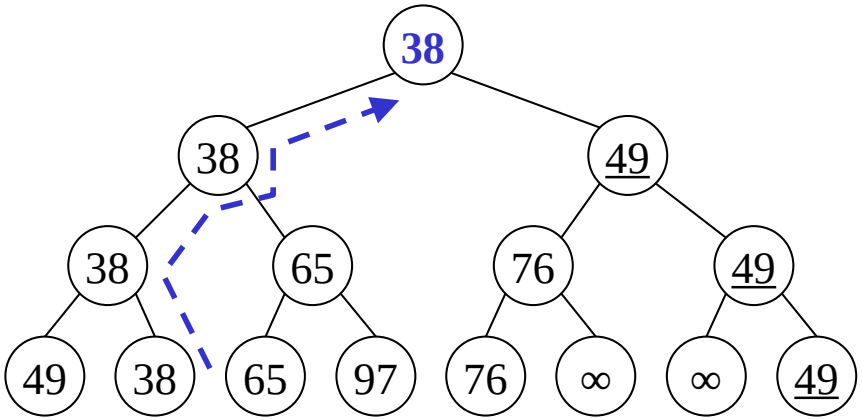
```
void ShellInsert(SqList &L, int dk) {
  for (i=dk+1; i<=L.length; i++) // 逐个间隔插入，不分子集
    if (LT(L.r[i].key, L.r[i-dk].key)) {
      for (j=i-dk; j>0 && LT(L.r[j+dk].key, L.r[j].key); j-=dk)
        L.r[j+dk] = L.r[j];            //  记录后移，查找插入位置
     L.r[j+dk] = L.r[0];               //  插入
    }
} // ShellInsert
void ShellSort(SqList &L, int dlta[], int t) {
   for (int k=0; k<t; ++k)
      ShellInsert(L, dlta[k]);  //  一趟增量为 dlta[k] 的插入排序
} // ShellSort
```

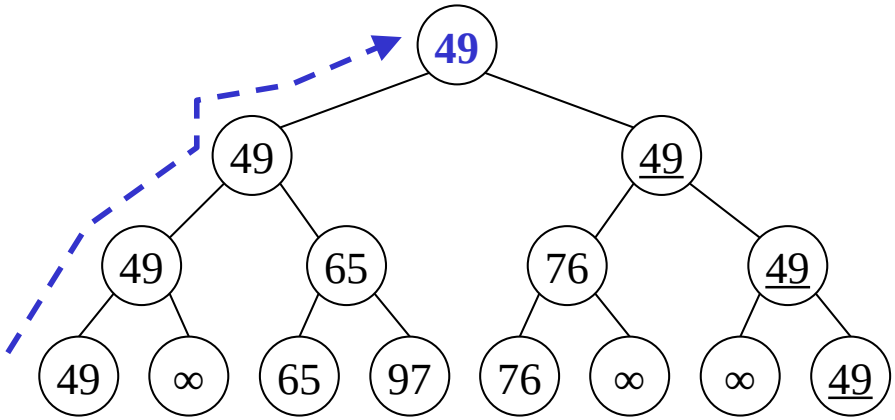
能用哨卡吗？

10.3.5 树形选择排序


13
38
13
38
65
13
27
49
38
65
97
76
13
27
49


27
38
27
38
65
76
27
49
38
65
97
76
∞
27
49


38
38
49
38
65
76
49
49
38
65
97
76
∞
∞
49


49
49
49
49
65
76
49
49
∞
65
97
76
∞
∞
49

10.4 基数排序（复杂度 O(d×n)）

- 逻辑关键字、关键字，关键字分级
- 对于有 d 个关键字的记录排序，要进行 d 趟分配和收集
- 时间复杂度 O(n×d); 空间复杂度 O(n)

待排关键字序列

A

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

B

关键字		30	63	73	83	95	65	78	89	69

count	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

10.5 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（ Logn ）
  - 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。

    例：（ 4 ，3 ，4 ，2 ）进行快速排序不稳定
- 经过一趟排序能将一个数据放在最终位置
  - 选择、起泡、堆

排序的另一种分类

选择排序：

- 简单选择、树形选择、堆排序

交换排序：

- 起泡排序、快速排序

插入排序：

- 简单插入、折半插入、表插入、希尔排序

归并排序

基数排序

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(nLogn)	$O(n^2)$	O(nLogn)	O(Logn)	N	坏
归并	O(nLogn)	O(nLogn)	O(nLogn)	O(n)	Y	无
堆	O(nLogn)	O(nLogn)	O(nLogn)	O(1)	N	无
基数	O(n*d)	O(n*d)	O(n*d)	O(n)	Y	无

数据库基础

袁平波　　2020.9

课程主页： http://staff.ustc.edu.cn/~ypb

第一章 绪 论

1.1 引言

1.2 数据模型

1.3 数据库系统的结构及功能

1.4 数据库管理系统

1.5 数据库工程与应用

1.1 引言

- 数据库技术产生于 20 世纪 60 年代中期
- 是数据管理的最新技术
- 计算机科学的重要分支
  - 计算机应用的三个方面：科学计算、数据管理、过程控制

1.1.1 数据库系统等相关概念

- 数据（ Data ）
  - 数据是信息的符号记录。数据是数据库处理和研究的对象。
- 数据库（ Database ）
  - 长期存储在计算机内，有组织的、可共享的相关数据的集合
- 数据库管理系统（ DBMS ）
  - 位于用户和操作系统之间的一层数据管理软件。
- 数据库系统（ DBS ）
  - 计算机硬件为基础的记录保持系统。包括数据库、数据库管理系统、应用系统、管理员和用户，有时还包括计算机硬件。

1.1.2 数据管理技术的发展

- 数据管理是指对数据进行分类、组织、编码、存储、检索和维护
- 分为三个阶段 ：
  - 人工管理阶段
  - 文件系统阶段
  - 数据库系统阶段
- 三个阶段的对比

	阶段	人工	文件系统	数据库系统
背景	时间	20 世纪 50 年代末	60 年代中期	60 年代末
	应用背景	/	科学计算、管理	大规模管理
	硬件背景	无直接存储设备	磁盘、磁鼓	大容量磁盘
	软件背景	无 OS	有文件系统	有 DBMS
	处理方式	批处理	联机和批处理	联机和批处理
特点	数据管理者	人	文件系统	DBMS
	数据面向对象	/	某一应用程序	现实世界
	数据共享程度	无，冗余大	共享性差，冗余较大	共享性高冗余小
	数据独立性	无	独立性差（无逻辑）	有高度独立性
	数据结构化	无	记录有结构，整体无	数据模型描述
	数据控制能力	/	应用程序控制	DBMS 保护： 安全性、完整性、并发控制、数据恢复

1.1.3 数据库技术的研究领域

- 数据库管理系统软件的研制（面向对象、多媒体数据库等）
- 数据库设计（设计方法学和设计工具、数据模型与建模、设计规范与标准）
- 数据库理论（规范化理论）

1.1.4* 数据库系统的特点

- 数据的结构化
- 数据的共享性好、冗余度低
- 数据的独立性高：物理、逻辑
- 数据由 DBMS 统一管理
  - 数据的安全性（ Security ）
  - 数据的完整性（ Integrity ）
  - 并发控制（ Concurrency ）
  - 数据库恢复（ Recovery ）
- 良好的用户接口

1.1.5* 数据库在信息科学中的应用

- 三个世界

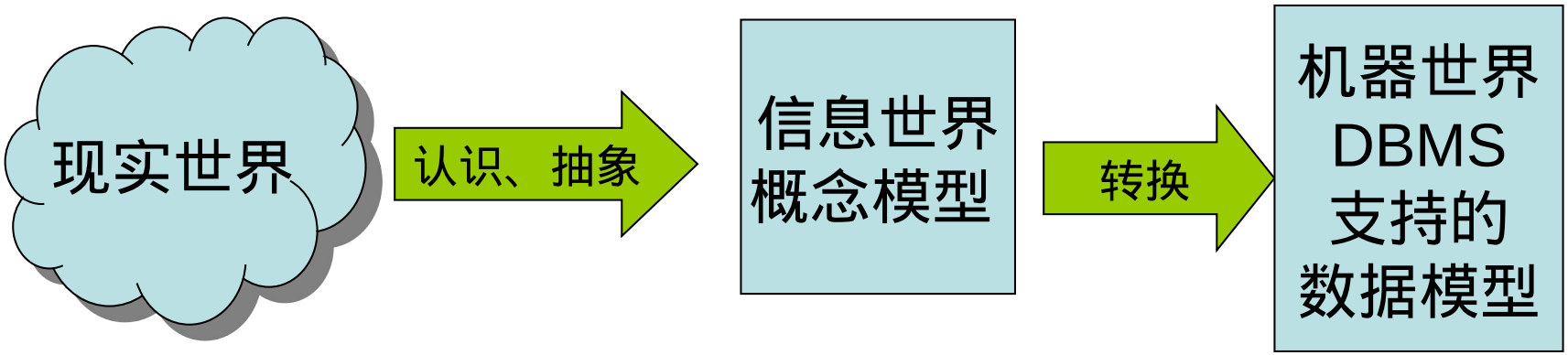
现实世界	信息世界	计算机世界
实体 (Entity) 实体集	实体记录 (Record) 记录集	数据 (Data) 数据集
特征 特征值 特征取值范围	属性 (Attribute) 属性值 属性值域	数据项（属性） 数据项值 数据项值域

1.2 数据模型

- 数据的组织是数据库技术的核心问题
- 数据库的数据组织是通过数据模型来实现的
- 数据模型是创建数据库维护数据库的方式，是数据库系统定义数据内容和数据间联系的方法
- 数据模型的定义：表示实体类型和实体间联系的模型称数据模型

数据模型的三个方面要求：

- 比较真实模拟真实世界
- 容易为人所理解
- 便于计算机实现



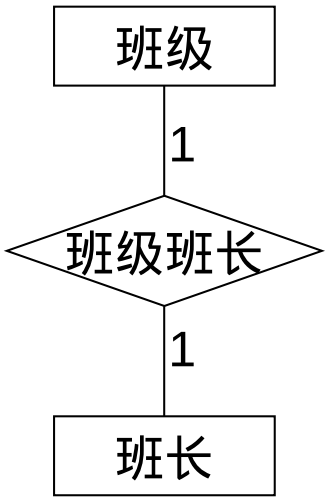
数据模型的两个层次：

- 概念（数据）模型：
  - 也称信息模型。用来描述信息结构，又称实体联系模型（ **ER** ）
  - 按照用户观点对信息建模
- （结构）数据模型：
  - 面向数据库的逻辑结构，直接涉及到计算机系统和 DBMS ，又称为（基本）数据模型
  - 按照计算机系统的观点对数据建模

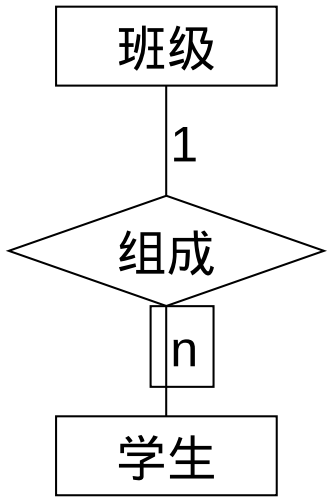
1.2.1 概念模型

- 实体联系模型（ Entity Relationship Model ，简记 ER 模型）
  - 直接从现实世界中抽象出实体和实体间联系，然后用实体联系图（ ER 图）表示信息模型
  - ER 模型实际是信息世界的模型。
- ER 图的四个组成部分
  - 矩形框：实体型
  - 菱形框：联系
  - 椭圆框：实体型和联系的属性
  - 直线：连接实体类型和联系类型，表示联系的种类

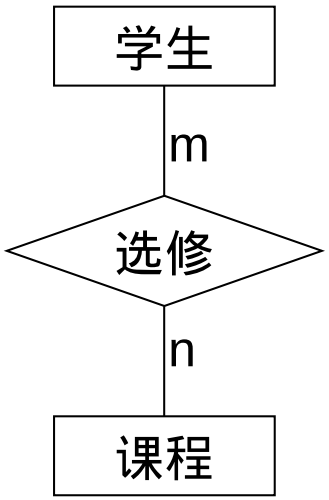
ER 图举例



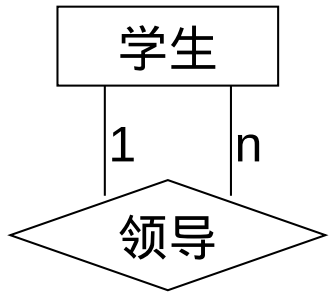
两个实体 1:1



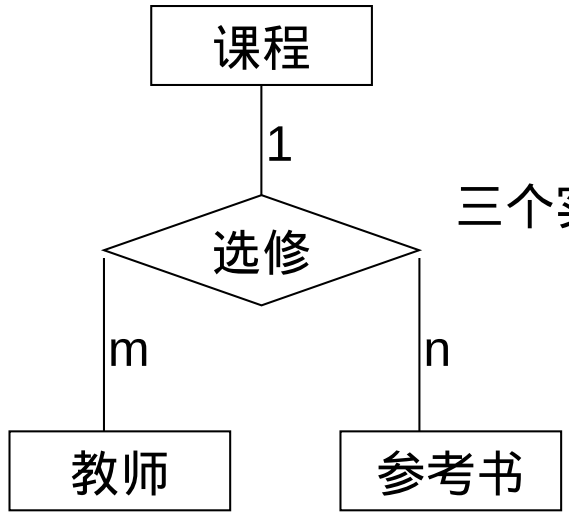
两个实体 1:n



两个实体 m:n



同一实体 1:n



三个实体 1:n

1.2.2 结构模型的要素

- 数据结构：
  - 用于描述系统的静态特征。
  - 是对实体类型和实体间联系的表达和实现，命名依据。
- 数据操作：
  - 是用于描述系统的动态特征。
  - 是对数据库检索和更新（插入、修改、删除）两类操作
- 数据的约束条件：
  - 一组完整性规则的集合。
  - 给出数据及其联系所具有的制约和依赖原则。

1.2.3（结构）数据模型

- 结构数据模型主要有三种
  - 层次模型、网状模型、关系模型
  - 未来的发展有：对象模型、语义模型
- 非关系模型中的数据结构的单位是基本层次联系。所谓基本层次联系是指的两个记录以及它们之间的一对多（包括一对一）的联系

1.2.3.1 层次模型

- 典型代表： IBM1968 年的 IMS
- 定义
  - 用树型（层次）结构表示实体类型·及实体间联系的数据模型
- 数据结构特点
  - 有且仅有一个结点无双亲，该结点是根结点
  - 其他结点有且仅有一个双亲
  - 层次模型中每个结点（片断）表示一个记录类型
  - 每个记录类型包含若干个字段，字段有字段名、数据类型和长度等

- 数据操纵与完整性约束
  - 插入时，没有双亲结点不能插入子女结点
  - 删除时，删除双亲结点要同时删除子女结点
  - 更新时，要更新所有相应的记录
- 存储结构
  - 邻接法
  - 链接法

- 优点
  - 模型本身简单
  - 实体间联系是固定的，预先设计好的应用系统，性能优于关系型，不低于网状模型
  - 层次模型提供了良好的完整性支持。
- 缺点
  - 实现多对多关系时，需要采用冗余或虚结点方式，易形成不一致性
  - 对插入和删除操作限制较多。
  - 查询子结点需通过双亲结点。
  - 结构严谨，层次命令趋于程序化。

1.2.3.2 网状模型

- 典型代表：
  - DBTG（ DataBase Task Group ）开发系统
  - HP 公司的 IMAGE/3000
  - Honeywll 公司的 IDS/II
  - Univac 公司的 DMS1100 等
- 定义
  - 用有向图（网络）结构表示实体类型及实体间联系的数据模型。（ network model ）
- 数据结构特征
  - 允许一个以上的结点无双亲
  - 至少一个结点不止一个双亲

- 数据操纵与完整性约束
  - 插入时，允许插入未确定双亲结点的子女结点值
  - 删除时，允许只删除双亲结点值
  - 更新时，只需更新指定记录
- 存储结构
  - 常用链接法
  - 其它如引元阵列法、二进制阵列法、索引法等

- 优点
  - 更加直接描述现实世界
  - 存取效率高，性能好
- 缺点
  - 结构复杂，不利于用户掌握， DDL ， DML 语言复杂
  - 数据独立性差

1.2.3.3 关系模型

- 定义
  - 用表格表示实体集和实体间联系，用外键来实现关系间的联系（ Relation model ）
- 数据结构特征
  - 在用户看来，一个关系模型的逻辑结构就是一张二维表；
  - 必须使用规范化的关系，如分量是原子的；
  - 不仅实体用关系表示，实体间的联系也用关系表示

- 术语
  - 关系（ relation ）：一个关系对应通常一张表 记做 R
  - 元组（ Tuple ）：表中的一行 （ V1 ， V2…… Vn ）
  - 字段（ field ）：表中的一列 A1 ， A2……An
  - 域（ Domain ）：属性的取值范围
  - 分量：元组中的一个属性的取值

- 关系模式：是记录型。对关系的描述，关系模式的实例是一个关系。关系模式的描述包括：关系名、属性名、属性类型、属性长度、主键包含的属性等，记做：
  R（U，D，DOM，F）
- 关系数据库模式：一组关系模式的总称即关系数据库模式，简称模式。
- 候选键（Candidate Key）：在给定的关系中，有这样的属性或最小属性组，它在不同的元组中的值是不同的，利用这个（些）值可以唯一地标示关系中地元组，则称该属性（组）为候选键。

- 主码（键）（ Primary key ）：候选键可能不止一个，被指定正在使用地候选键称主键。
- 超键（ Super Key ）：能唯一标识元组的属性集。超键包括所有候选键。
- 组合键与全键：当主键不是单一的属性时称组合键，组合键包括所有属性时称全键。
- 外部键（ Foreign Key ）：关系模式 R 中的属性集是其它关系模式的主键，那么该属性集对关系模式 R 而言是外键。
- 主属性和非主属性：包含在任一候选键中的属性叫主属性，否则称非主属性

- 操纵与完整性约束
  - 插入、删除和更新操作必须满足关系的完整性约束；
  - 关系的完整性约束包括：实体完整性、参照完整性和自定义完整性
- 存储结构
  - 实体和实体间的联系都用表来表示
  - 表通常以文件形式存储

- 优点：
  - 建立在严格的数学概念基础上
  - 概念单一，实体、联系均用关系来表示
  - 存取路径对用户透明，数据独立性更高，保密性更好。简化程序员工作和数据库开发建立工作。
- 缺点：
  - 存取路径对用户透明，查询效率不高
  - 因存取路径对用户透明，必须对用户查询进行优化，增加了开发 DBMS 的难度

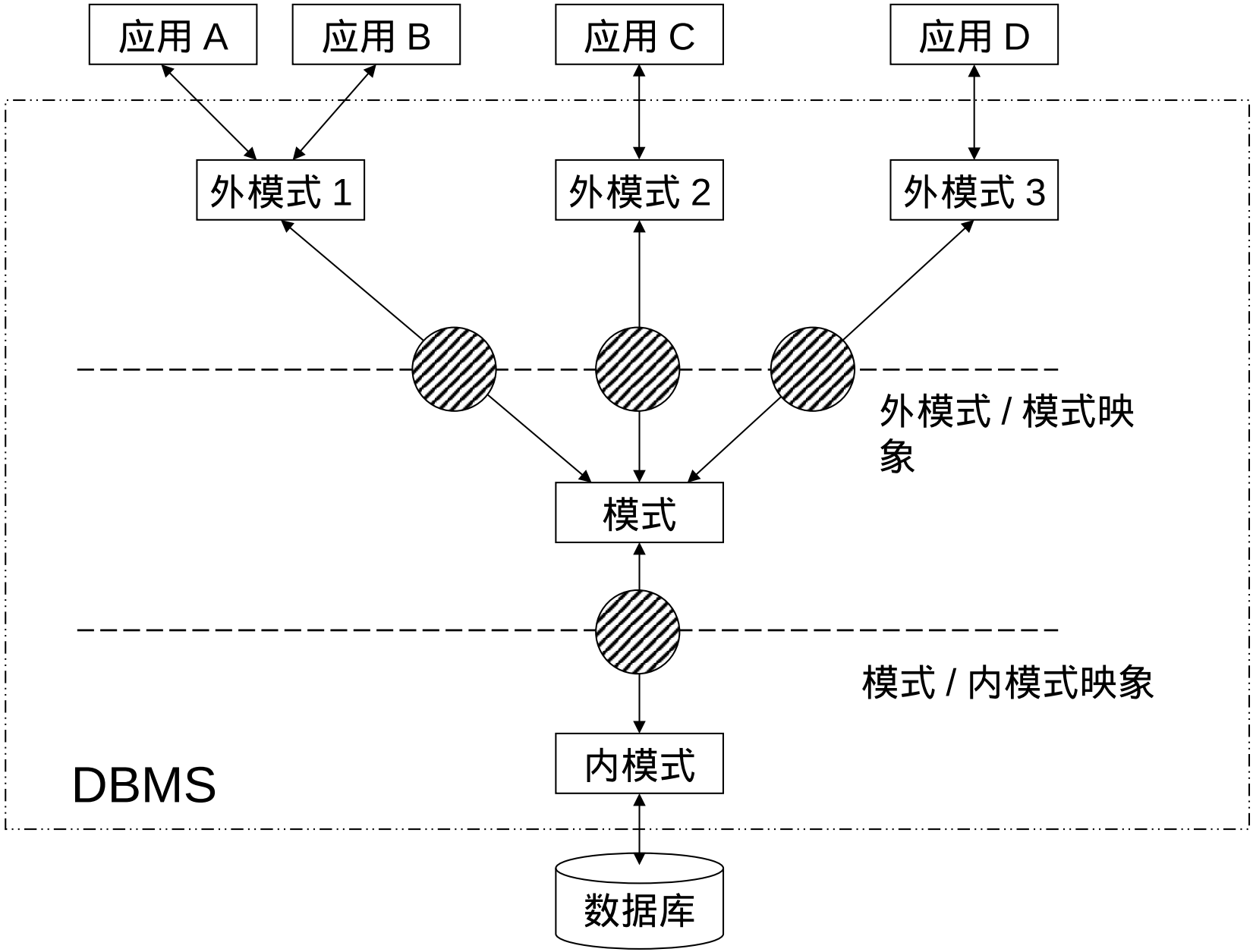
1.3 数据库系统的结构及功能

- 1.3.1 数据库系统的模式结构 (从管理系统角度看)
  - **SPARC** 接口分三级结构

	schema	**view**	**level**
External	外模式 子模式	用户视图 外视图	外部级 局部逻辑级
conceptual	模式 概念模式	概念视图 DBA 视图	概念级 全局逻辑级
internal	内模式 存储模式	内部视图	内部级 物理级 存储级

– 数据库的二级映象功能与数据独立性

- 概念模式 / 内模式的映象
- 概念模式 / 外模式的映象
- 物理独立性：当存储结构改变时，可以通过修改概念模式 / 内模式的映象使概念级保持不便，这样外部级和应用程序也不会改变。
- 逻辑独立性：当概念级发生改变时，可以通过修改概念模式 / 外模式的映象使外部级尽量保持不变，应用也就不需改变。


应用 A
应用 B
应用 C
应用 D
外模式 1
外模式 2
外模式 3
外模式 / 模式映象
模式
模式 / 内模式映象
内模式
DBMS
数据库

数据库系统模式结构

1.3.2 数据库系统的体系结构（从最终用户角度来看）

- 单用户 DBS
- 主从式结构 DBS（终端方式）
- 客户 / 服务器结构 DBS
- 分布式结构 DBS

1.3.3* 数据库系统的组成

- 计算机硬件　CPU 、 内存、 硬盘
- 计算机软件　OS ， DBMS 、 应用软件包、 应用程序
- 数据（ 库 ）
- 管理员、 用户

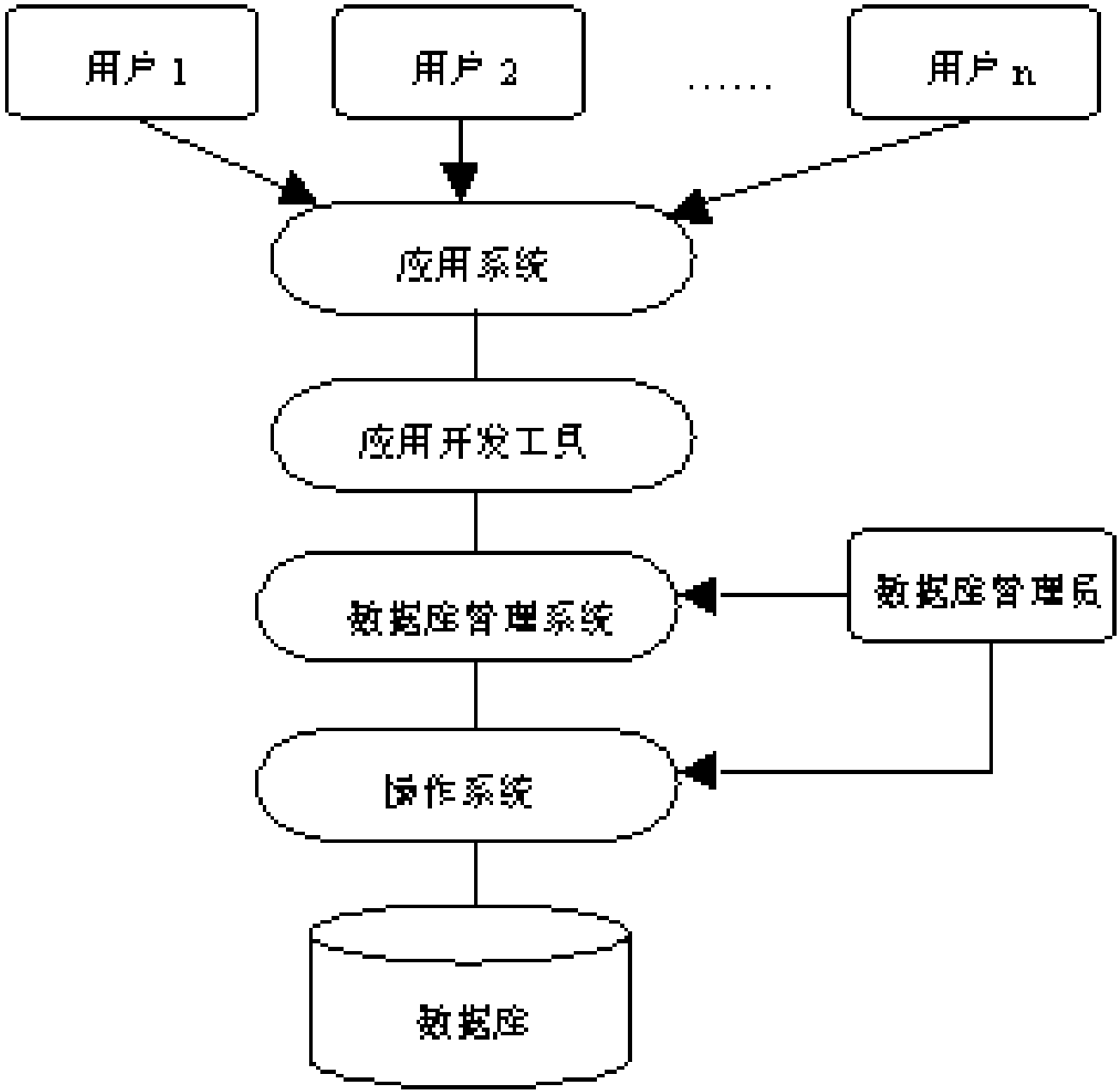


图 1-1　数据库系统

1.4 数据库管理系统

1.4.1 数据库管理系统的功能与组成

- DBMS 功能
  - 数据定义
  - 数据操纵数
  - 据库运行管理
    - 安全性检查 / 完整性检查 / 并发控制 / 索引、数据字典等内部维护
  - 数据组织、存储、管理
  - 数据库建立与维护功能
  - 数据通信接口功能

- DBMS 组成
  - 数据定义语言

    DDL Data Definition/Description Language
  - 数据操纵语言

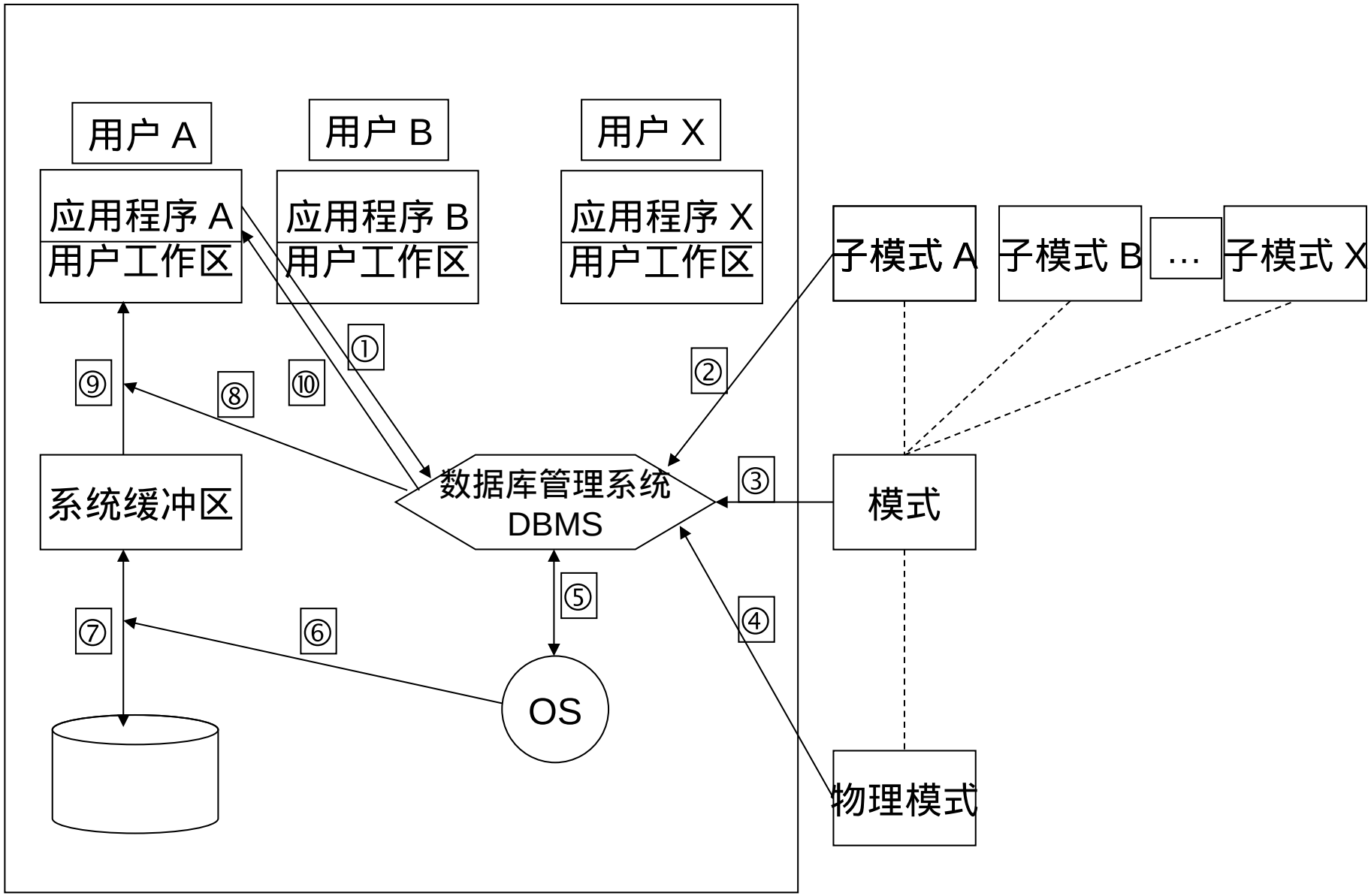
    DML Data Manipulation Language
  - 数据库运行控制语言

    DCL Data Control Language
  - 实用程序

1.4.2 数据库管理系统的工作过程

- 用户存取数据库数据的过程

① 用户在程序中嵌入 DML 的一个读记录语句。控制转向 DBMS
② DBMS 检查合法性，查找子模式表，确定对应的存取权限
③ DBMS 依据子模式 / 模式映象的定义，确定应读入的模式记录
④ DBMS 依据模式 / 内模式映象的定义，确定应读入的物理记录
⑤ DBMS 向 OS 发送读取所需物理记录的命令
⑥ OS 启动 IO 程序，执行读操作
⑦ OS 将数据从数据库的存储区送到系统缓冲区
⑧ DBMS 依据子模式 / 模式映象定义，导出用户所要读取的记录格式
⑨ DBMS 将数据记录从系统缓冲区传送到程序的用户工作区
⑩ DBMS 向应用程序返回命令执行情况和状态信息



从数据库中读取记录的过程

1.4.3 数据库管理系统的实现方法

- N 方案
  - DBMS 模块加入到用户进程， DBMS 代码出现多副本，性能大幅下降
- 2N 方案
  - 每个用户有一个 DBMSshdow 进程和一个后台负责读写和日值的进程
- M ＋ N 方案
  - 2N 方案的改进方案， N 用户， M 个 DBMS 进程（ M<N ）
- N ＋ 1 方案
  - 1 个 DBMS 进程 (可设计成多线程的)， N 个用户进程，消息开销大

1.5 数据库工程与应用

- 1.5.1 数据库设计的目标和特点
  - 数据库设计的任务是：在 DBMS 的支持下，按照应用的要求，为某一部门或组织设计一个结构合理、使用方便、效率较高的数据库及其应用系统
  - 数据库设计包含两方面内容
    - 结构（数据）设计
    - 行为（处理）设计

1.5.2 数据库设计方法

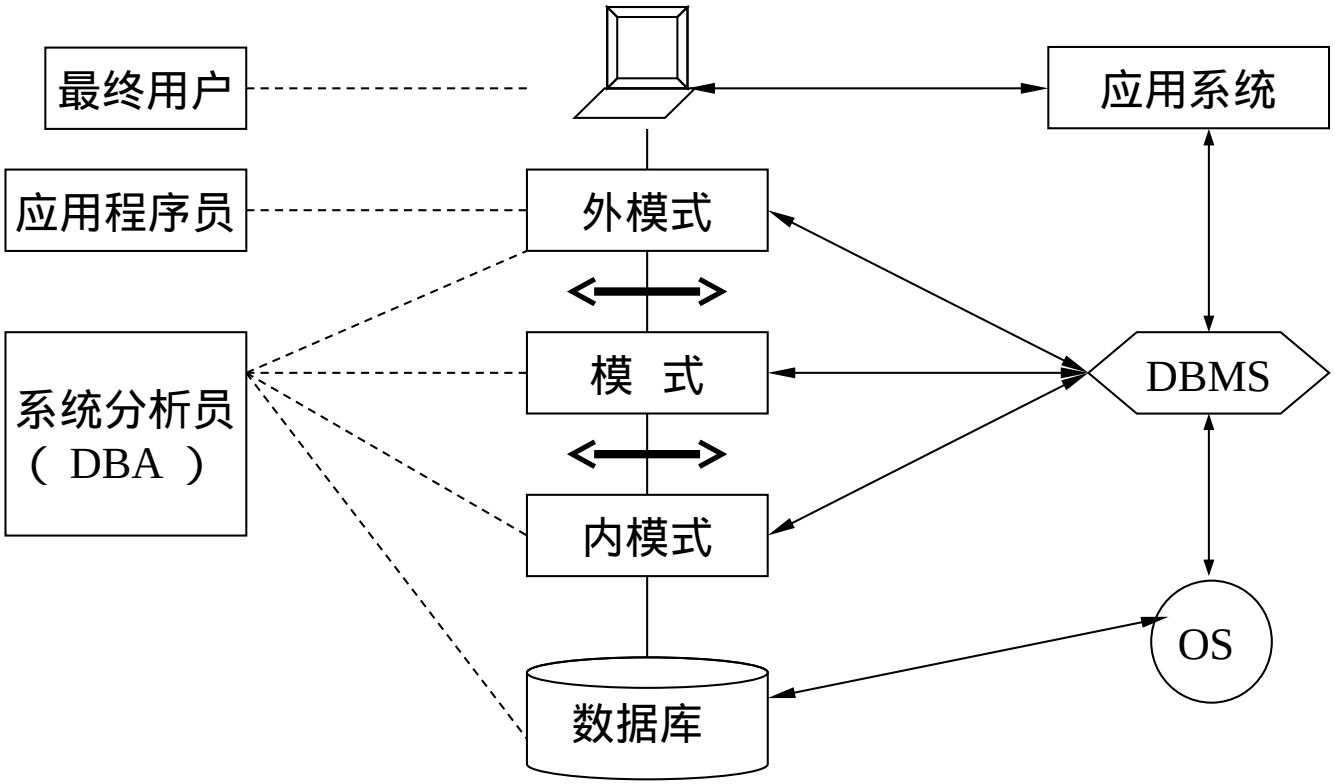
- 核心是：逻辑设计和物理设计
- 著名设计方法
  - 新奥尔良设计：需求分析、概念设计、逻辑设计、物理设计
  - S.B.Yao 设计：需求分析、模式构成、模式汇总、模式重构、模式分析、物理设计

1.5.3 数据库设计步骤

- 步骤
  - 需求分析、概念结构、逻辑结构、物理结构、实施、运行维护
- 设计注意的问题
  - 让用户参与，调动用户积极性
  - 充分考虑系统的可扩充性（可扩充性有限度）
  - 设计新系统要考虑旧系统的数据平稳迁移到新系统

1.5.4 数据库应用

- 各种用户的数据视图



DBA 的职责

- 设计与定义数据库
- 帮助最终用户使用数据库
- 监督和控制数据库运行
- 改进和重组数据库
- 转储和恢复数据库
- 重构数据库

第二章关系数据库

2.1 关系数据库概述
2.2 关系数据结构
2.3 关系的完整性
2.4 关系代数
2.5 关系演算 **
2.6 关系数据库管理系统

2.1 关系数据库概述

- 关系数据库系统是支持关系模型的数据库系统
- 关系理论是建立在集合代数理论基础上的，关系的定义和各种操作运算可以用集合代数给出
- 关系模型的三要素
  - 关系数据结构：二维表
  - 关系操作：选择、投影、连接、除、并，交、差等查询以及增、删、改
  - 完整性约束 ：实体、参照、自定义

关系数据语言

- 关系代数语言　　ISBL
- 关系演算语言
  - 元组关系演算语言　　ALPHA ， QUEL
  - 域关系演算语言　　QBE
- 具有关系代数和关系演算双重特点的语言SQL

2.2 关系数据结构

- 2.2.1 关系
  - 域：域是一组具有相同数据类型的值的集合。值的个数称为域的基数
  - 笛卡儿乘积 ：给定一组域： D1 ， D2 ，…… Dn ，域可以相同，定义 D1D2……Dn 的笛卡儿乘积为： D1×D2×……×Dn = {(d1 ， d2 ，…… dn) |di∈Di ， i = 1 ， 2 ，…… n} ；（ d1 ， d2 ，…… dn ）称为一个元组
  - 关系 (Relation) ：笛卡儿乘积 D1×D2×……×Dn 的任一子集 D’，称作 D1 ， D2 ，…… Dn 上的关系。

    用 R （ D1 ， D2……Dn ）来表示

    D’ 中的每个元素（ d1 ， d2 ，…… dn ）是关系的一个元组

    实际应用中关系往往是笛卡儿乘积中有意义的子集构成

    n = 1 是单元关系 / 一元关系； n = 2 是二元关系

举例

- 域
  - 性别集 = { 男、女 } 。基数 = 2
  - 月份集 = {1 ， 2 ， 3 ， 4 ， 5 ， 6 ， 7 ， 8 ， 9 ， 10 ， 11 ， 12} ，基数 = 12
- 笛卡儿乘积
  - D1 = 姓名集合 = { 赵一平，钱峰，孙英 }
  - D2 = 性别集合 = { 男，女 }
  - D3 = 年龄集合 = {16 ， 17 ， 18}
- 关系

姓名	性别	年龄
赵一平	男	16
钱峰	男	17
孙英	女	17

2.2.2 关系模式

- 关系的描述称为关系模式（ **Relation schema** ），一般表示为 **R(U,D,DOM,F)** 其中，R 是关系名，U 是组成该关系的属性集合，D 为属性组 U 中属性所来自的域，DOM 是属性向域的映象集合，F 是属性间数据的依赖关系集合。

2.2.3 关系数据库

- 在一个给定的现实世界领域里，所有实体及实体间的联系的关系所构成的集合是一个关系数据库
- 关系数据库有型和值之分：关系数据库的型也称关系数据库模式，是对关系数据库的描述它包括若干域的定义以及在这些域上定义的若干关系模式；关系数据库的值也称为关系数据库，是这些关系模式在某一时刻对应的关系的集合
- 关系数据库的值与关系数据库模式通称为关系数据库

2.3 关系的完整性

实体完整性

- 若属性 A 是基本关系 R 的主属性，则 A 不能取空值

参照完整性

- 若属性（或属性组）F 是基本关系 R 的外码，它与基本关系 S 的主码 Ks 相对应（关系 R ， S 不一定是不同的关系），则对于 R 中的每一个元组在 F 上的取值必须：
  - 取空值（ F 的每个属性值均取空值）
  - 等于 S 中某个元组的主码值]

- 自定义完整性

2.4 关系代数

- 关系代数由一组关系运算组成，是对于关系的操作集。关系运算以一个或多个关系作为操作的对象，运算结果是一个新的关系。用关系运算实现查询
- 关系代数运算符
  - 集合运算符：∪（并）-（差）∩（交）×（笛卡儿积）
  - 专门运算符：σ选择 Π 投影 ▷◁ 连接 ÷ 除
  - 比较运算符： ＞≥＜≤＝≠
  - 逻辑运算符：¬ 非 ∧与 ∨或
- 常用的关系运算
  - 交、并、差、笛卡儿积、投影、选择、连接、除
- 基本关系运算有
  - 并、差、笛卡儿积、投影、选择
- 同类关系：具有相同的度，且两个关系每个属性属同一个域

2.4.1 传统的集合运算

假设：

R

Name	Sex	Age
Zhang	F	22
Wang	M	25
Lu	M	37
Chen	F	27

S

Name	Sex	Age
Zhang	F	22
Wang	M	25
Lu	F	30
Sun	M	28

并（ Union ）：

- 同类关系 R 和 S 的并记为 R∪S ，或 R union S
- 定义： R∪S={t|t∈R ∨ t∈S} 注意去除重复元组

R∪S

Name	Sex	Age
Zhang	F	22
Wang	M	25
Lu	M	37
Chen	F	27
Lu	F	30
Sun	M	28

交（ Intersection ）

- 同类关系 R 和 S 的交记为 R∩S ，或 R intersect S
- 定义： $R \cap S = \{ t \mid t \in R \wedge t \in S \} \equiv R - (R - S)$

R∩S

Name	Sex	Age
Zhang	F	22
Wang	M	25

差（ Minus/Difference ）

- 同类关系 R 和 S 的差记为 R - S 或 R minus S
- 定义： $R - S = \{ t | t \in R \wedge t \notin S \}$

R - S

Name	Sex	Age
Lu	M	37
Chen	F	27

笛卡儿积（ Cartesian Product ）

- 关系 R 和 S 的笛卡儿积记为 R×S
- 定义： R×S = { t□s|t∈R ， s∈S }

R

CNo	CN
C-11	OS
C-21	DB

S

SNo	SN	Age
S-01	Huang	21
S-21	Lin	20
S-30	Shao	22

R×S

CNo	CN	SNo	SN	Age
C-11	OS	S-01	Huang	21
C-11	OS	S-21	Lin	20
C-11	OS	S-30	Shao	22
C-21	DB	S-01	Huang	21
C-21	DB	S-21	Lin	20
C-21	DB	S-30	Shao	22

2.4.2 专门的关系运算

- 引入以下记号：
  - 设关系模式 R（A_1，A_2，…… A_n），它的一个关系为 R_t，$t \in R_t$ 表示 t 是 R_t 的一个元组。$t[A_i]$ 则表示元组 t 中相应于 A_i 的一个分量
  - 若 $A = \{A_{i1}, A_{i2}, \ldots\ldots A_{ik}\}$ 是 A_1，A_2，…… A_n 的一部分，k<=n，则 A 称为属性列（组）或域列。$t[A]=(t[A_{i1}], t[A_{i2}], \ldots\ldots t[A_{ik}])$ 表示元组在属性列 A 上诸分量的集合
  - R 为 n 元关系，S 为 m 元关系。$tr \in R$，$ts \in S$，$tr \frown ts$ 称为元组的连接（Concatenation）。它是一个 m + n 列的元组，前 n 个分量为 R 中的一个 n 元组，后 m 个分量为 S 中的一个 m 元组
  - 给定一个关系 R（X，Z），X 和 Z 为属性组，定义：当 t[X]=x 时，x 在 R 中的象集（image set）为 $Zx = \{t[Z] | t \in R, t[X]=x\}$，表示 R 中属性组 X 上值为 x 的诸元组在 Z 属性组上的分量的集合

投影（ Projection ）

- 关系 R 上的投影是从 R 中选择出若干属性，并且去掉重复元组组成一个新关系，属于单目运算
- 记作： $\prod_A(R) = \{t[A] | t \in R\}$ A 为 R 中的属性列

假设 Student

SNo	SName	Sex	Age
S01	Wang	F	17
S02	Zhang	M	20
S03	Lin	M	18
S04	Sun	F	19

$S_{na}=\prod_{Sname\ ,\ Age}(Student)$

SName	Age
Wang	17
Zhang	20
Lin	18
Sun	19

选择（ Selection ）

- 又称限制（ Restriction ），在给定的关系 R 中，抽出满足条件的元组，组成一个新关系，新关系与原关系同类，是原关系一个子集
- 记做： $\sigma_F(R)=\{t| t\in R \wedge F（t）=$ ’真’ $\}$ F 表示条件

$S_{a18} = \sigma_{age>=18}(Student)$

SNo	SName	Sex	Age
S02	Zhang	M	20
S03	Lin	M	18
S04	Sun	F	19

连接（ Join ）

- 从两个关系的笛卡儿乘积中选取属性满足一定条件的元组，组成新的关系
- 记做： $R \underset{A\theta B}{\triangleright\triangleleft} S = \{tr^\frown ts \mid tr \in R \wedge ts \in S \wedge tr[A]\theta ts[B]\} \equiv$

  $\sigma_{A\theta B}(R \times S)$
- AθB 表示 R 上的属性 A 和 S 上的属性 B 满足θ条件，θ是比较运算符，A、B 的度数相等且可比。这里假设 AB 分别在 R、S 关系的第 i、j 列，R 度为 r

- 等值连接（ equi - join ）：θ为“ = ”时称为等值连接

  记： $R \underset{A=B}{\triangleright\triangleleft} S = \{tr^\frown ts \mid tr \in R \wedge ts \in S \wedge tr[A] = ts[B]\}$

- 自然连接（ Natioal Join ）：两个关系中具有相同的属性，并且在相同的属性上做等值连接。自然连接需要取消重复列，而等值连接不需要 。

  记： $R \triangleright\triangleleft S = \{tr^\frown ts \mid tr \in R \wedge ts \in S \wedge tr[A] = ts[A]\}$

假设 R

A	C	D
30	C1	D3
40	C2	D3
50	C3	D1
10	C4	D1

S

B	E	F
20	E1	F1
50	E2	F3
40	E3	F1

$R \triangleright\triangleleft_{A>B} S$

A	B	C	D	E	F
30	20	C1	D3	E1	F1
40	20	C2	D3	E1	F1
50	20	C3	D1	E1	F1
50	40	C3	D1	E3	F1

除法（ Division ）

- 给定关系 R （ X ， Y ）和 S （ Y ， Z ），其中 X ， Y ， Z 为属性组， R 中的 Y 与 S 中的 Y 可以不同属性名，但必须有相同的域。记 R÷S 。令 P （ X ） = R÷S ，则 P 是 R 中满足以下条件的元组在 X 属性列上的投影：元组在 X 上的分量值 x 的象集 Yx 包含 S 在 Y 上投影的集合
- 记做： $R \div S = \{tr[X] | tr \in R \wedge Yx \supseteq \prod_Y(S)\}$
- $R \div S \equiv \prod_X(R) - \prod_X((\prod_X(R) \times \prod_Y(S)) - R)$

假设 R

A	B	C
a1	b1	c2
a2	b3	c7
a3	b4	c6
a1	b2	c3
a4	b6	c6
a2	b2	c3
a1	b2	c1

S

B	C	D
b1	c2	d1
b2	c1	d1
b2	c3	d2

R÷S

A
a1

象集：

$Y_{x = a1}$

b1	c2
b2	c1
b2	c3

$Y_{x = a2}$

b2	c3
b3	c7

$Y_{x = a3}$

b4	c6

$Y_{x = a4}$

b6	c6

- 外连接（ Outer Join ）
  - 如果 R 和 S 在做自然连接时，把该舍弃的元组也保存在新关系中，在新增加的属性上填空值（ null ），这种操作称为“外连接”。如果把 R 中该舍弃的元组保留在新关系中称左连接；把 S 中该舍弃的元组保留在新关系中称右连接
- 外部并（ Outer Union ）
  - 若关系 R 和 S 不同类，则新关系的属性由 R 和 S 的属性组成，公共属性只取一次，新关系的元组由属于 R 或 S 的元组构成，新增的属性上均填空（ null ）
- 半连接（ Semijoin ）
  - 关系 R 和 S 的半连接定义为 R 和 S 的自然连接在关系 R 的属性集上的投影

假设 R

A	B	C
a	b	c
b	b	f
c	a	d

S

B	C	D
b	c	d
b	c	e
a	d	b
e	f	g

R Outer Join S

A	B	C	D
a	b	c	d
a	b	c	e
c	a	d	b
b	b	f	null
null	e	f	g

R left Outer Join S

A	B	C	D
a	b	c	d
a	b	c	e
c	a	d	b
b	b	f	null

R right Outer Join S

A	B	C	D
a	b	c	d
a	b	c	e
c	a	d	b
null	e	f	g

R Outer Union S

A	B	C	D
a	b	c	null
b	b	f	null
c	a	d	null
null	b	c	d
null	b	c	e
null	a	d	b
null	e	f	g

R Semijoin S $\equiv \prod_R(R \triangleright\!\triangleleft S)$

A	B	C
a	b	c
a	b	c
c	a	d

S Semijoin R $\equiv \prod_S (R \triangleright\!\triangleleft S)$

B	C	D
b	c	d
b	c	e
a	d	b

2.4.3* 关系代数运算应用举例

假设：

S(S # ,SN,SSEX,SAGE)

C(C # ,CN,TEACHER)

SC(S # ,C # ,GRADE)

- 检索学习课程号为 C2 的学生学号与成绩

  $\Pi_{S\#,GRADE}(\sigma_{C\#='C2'}(SC))$ 或 $\Pi_{1,3}(\delta_{2='C2'}(SC))$

- 检索学习课程号为 C2 的学生学号与姓名

  $\Pi_{S\#,SN}(\sigma_{C\#='C2'}(S \triangleright\triangleleft SC))$

- 检索选修课程名为 Maths 的学生学号与姓名

  $\Pi_{S\#,SN}(\sigma_{CN='Maths'}(S \triangleright\triangleleft SC \triangleright\triangleleft C))$

- 检索选修课程为 C2 或 C4 的学生学号

  $\Pi_{S\#}(\sigma_{C\#='C2' \vee C\#='C4'}(SC))$
- 检索至少选修课程为 C2 和 C4 的学生学号

  $\Pi_{S\#}(\sigma_{1=4 \wedge 2='C2' \wedge 5='C4'}(SC \times SC))$
- 检索不选修 C2 课程的学生姓名与年龄

  $\Pi_{SN,SAGE}(S) - \Pi_{SN,SAGE}(\sigma_{C\#='C2'}(SC \triangleright\triangleleft S))$
- 检索选修全部课程的学生姓名

  $\Pi_{SN}(S \triangleright\triangleleft (\Pi_{S\#,C\#}(SC) \div \Pi_{C\#}(C)))$
- 检索所学课程包含学生 S3 所学课程的学生学号

  $\Pi_{S\#,C\#}(SC) \div \Pi_{C\#}(\sigma_{S\#='S3'}(SC))$

2.4.4 关系代数式的等价规则

1. 连接、笛卡尔积交换律

$E1 \times E2 \equiv E2 \times E1$

$E1 \triangleright\triangleleft E2 \equiv E2 \triangleright\triangleleft E1$

$E1 \triangleright\triangleleft_F E2 \equiv E2 \triangleright\triangleleft_F E1$

2. 连接、笛卡尔积结合律

$(E1 \times E2) \times E3 \equiv E1 \times (E2 \times E3)$

$(E1 \triangleright\triangleleft E2) \triangleright\triangleleft E3 \equiv E1 \triangleright\triangleleft (E2 \triangleright\triangleleft E3)$

$(E1 \triangleright\triangleleft_{F1} E2) \triangleright\triangleleft_{F2} E3 \equiv E1 \triangleright\triangleleft_{F1} (E2 \triangleright\triangleleft_{F2} E3)$

3. 投影的串接定律

$\Pi_{A1,A2,\ldots An}(\Pi_{Ak1,Ak2,\ldots Akm}(R)) \equiv \Pi_{A1,A2,\ldots An}(R)$

–A1,A2,…An 是 Ak1,Ak2,…Akm 的子集

4. 选择的串接定律

$\sigma_{F1}(\sigma_{F2}(R)) \equiv \sigma_{F1\wedge F2}(R)$

5. 选择与投影的交换律

$\Pi_{A1,A2,\ldots An}(\sigma_F(R)) \equiv \sigma_F(\Pi_{A1,A2,\ldots An}(R))$

$\Pi_{A1,A2,\ldots An}(\sigma_F(R)) \equiv \Pi_{A1,A2,\ldots An}(\sigma_F(\Pi_{A1,A2,\ldots An,B1,B2,..Bn}(R)))$

例： $\Pi_A(\sigma_{R.A=S.B}(R\times S))$

$\equiv \Pi_A(\sigma_{R.A=S.B}(\Pi_{A,B}(R\times S)))$

$\equiv \Pi_A(\sigma_{R.A=S.B}(\Pi_A(R)\times\Pi_B(S)))$

6. 选择对笛卡尔积的分配率

$\sigma_F(E1\times E2) \equiv \sigma_{F1}(E1)\times \sigma_{F2}(E2)$

F=F1^F2,F1 只涉及 E1 ， F2 只涉及 E2

$\sigma_F(E1\times E2) \equiv \sigma_F(E1)\times E2$

F 只涉及 E1

$\sigma_F(E1\times E2) \equiv \sigma_{F2}(\sigma_{F1}(E1)\times E2)$

F1 只涉及 E1 ， F2 涉及 E1 ， E2

7. 选择对并的分配率

$$\sigma_F(E1 \cup E2) \equiv \sigma_{F1}(E1) \cup \sigma_{F2}(E2)$$

8. 选择对差的分配率

$$\sigma_F(E1 - E2) \equiv \sigma_{F1}(E1) - \sigma_{F2}(E2)$$

9. 投影对笛卡尔积的分配率

$$\prod_{A1,A2,\ldots An,B1,B2,..Bn}(E1 \times E2) \equiv \prod_{A1,A2,\ldots An}(E1) \times \prod_{B1,B2,..Bn}(E2)$$

–A1,A2,…An 是 E1 属性， B1,B2,..Bn 是 E2 属性

10. 投影对并的分配率

$$\prod_{A1,A2,\ldots An}(E1 \cup E2) \equiv \prod_{A1,A2,\ldots An}(E1) \cup \prod_{A1,A2,\ldots An}(E2)$$

11. 选择对自然连接的分配率

$\sigma_F(E1 \triangleright\triangleleft E2) \equiv \sigma_{F1}(E1) \triangleright\triangleleft \sigma_{F2}(E2)$

F=F1^F2,F1 只涉及 E1 ， F2 只涉及 E2

12. 选择与连接操作的结合率

$\sigma_F(E1 \times E2) \equiv E1 \infty_F E2$

F 形如 E1.A θ E2.B

$\sigma_{F1}(E1 \triangleright\triangleleft_{F2} E2) \equiv E1 \triangleright\triangleleft_{F1^F2} E2$

F1,F2 形如 E1.A θ E2.B

利用规则优化查询

例：设学生选课系统中，学生关系 S 有 1000 条记录，每个学生平均选课 10 门，则 SC 关系有 10000 条记录，课程关系 C 有 1000 条记录。若需要查询学生“王芳”所选修课程的成绩在 85 分以上的课程名。

设　F1 表示 S.sno=SC.sno　F2 表示 SC.cno=C.cno

F3 表示 S.sn=‘ 王芳’　F4 表示 SC.grade>=85

①$\prod_{cn}(\sigma_{F1\wedge F2\wedge F3\wedge F4}(S\times SC\times C))$　10^{10} 条连接记录 $O(10^{10})$

②$\prod_{cn}(\sigma_{F3\wedge F4}(S\triangleright\triangleleft_{F1}SC\triangleright\triangleleft_{F2}C))$　10^{4} 条记录，运算 $O(10^{7})$

③$\prod_{cn}(\sigma_{F3}(S)\triangleright\triangleleft\sigma_{F4}(SC)\triangleright\triangleleft C))$　<=10 条记录，运算 $O(10^{4})$

优化过程

$$\Pi_{cn}(\sigma_{F1\wedge F2\wedge F3\wedge F4}(S\times SC\times C)) \quad //① 式$$

$$= \Pi_{cn}(\sigma_{F3\wedge F4}\sigma_{F2}(\sigma_{F1}((S\times SC)\times C))) \text{ // 规则 4 ， 2}$$

$$= \Pi_{cn}(\sigma_{F3\wedge F4}\sigma_{F2}(\sigma_{F1}(S\times SC)\times C)) \text{ // 规则 6}$$

$$= \Pi_{cn}(\sigma_{F3\wedge F4}\ \sigma_{F2}\ ((S \triangleright\triangleleft SC)\times C)) \quad \text{// 规则 12}$$

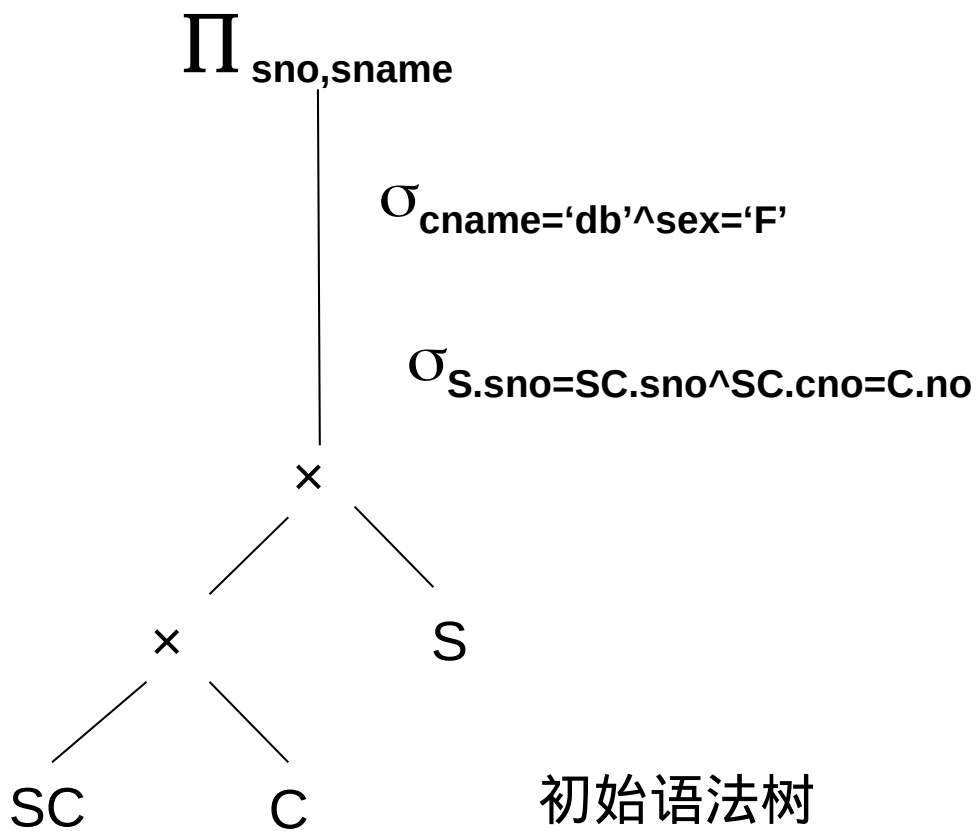
$$= \Pi_{cn}(\sigma_{F3\wedge F4}\ ((S \triangleright\triangleleft SC) \triangleright\triangleleft C)) \quad \text{// 规则 12 ② 式}$$

$$= \Pi_{cn}(\sigma_{F3}(S) \triangleright\triangleleft \sigma_{F4}(SC) \triangleright\triangleleft C) \quad \text{// 规则 11 ③ 式}$$

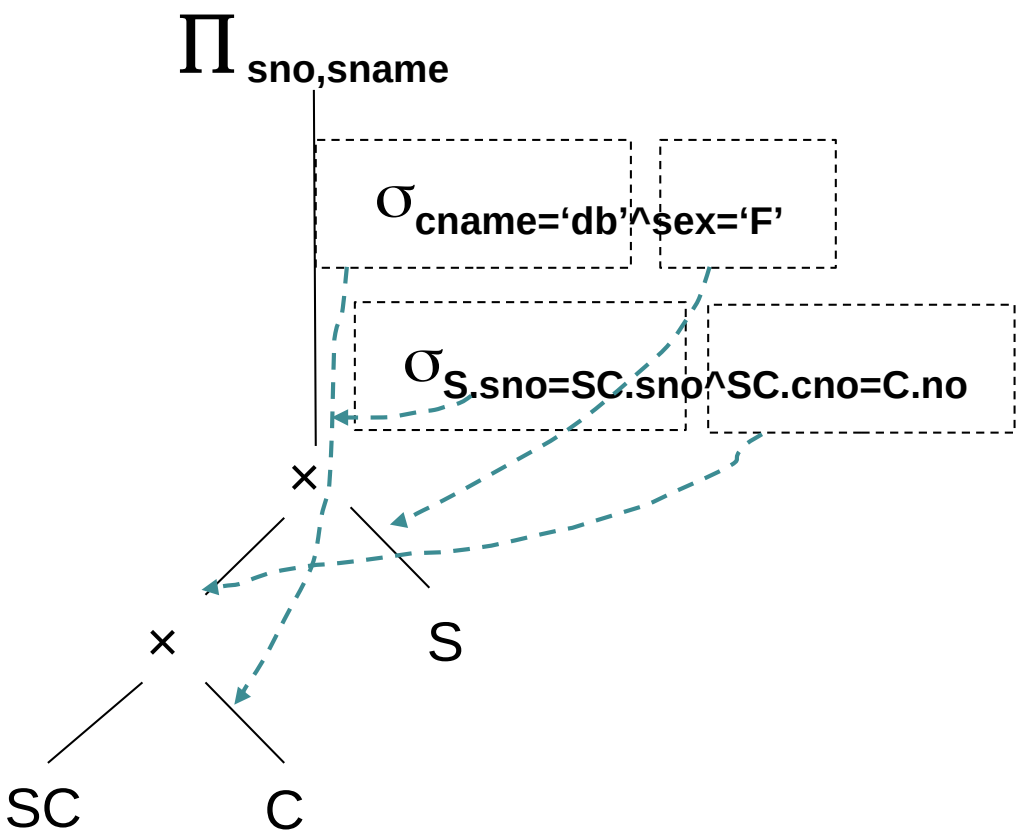
基于语法树优化

- 检索选修 DB 课程的女生学号及姓名。

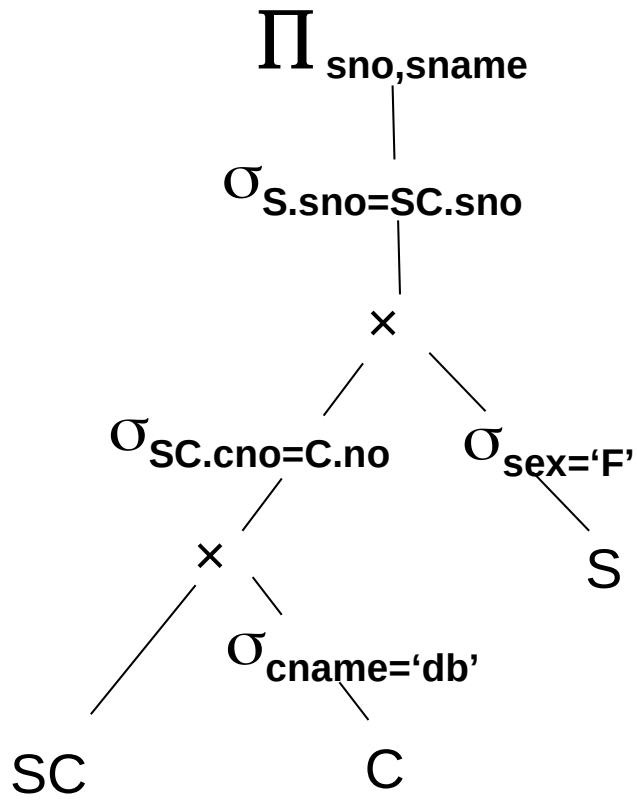
$$\Pi_{sno,sname}(\sigma_{cname='db'\wedge sex='F'}(\sigma_{S.sno=SC.sno\wedge SC.cno=C.no}(S\times SC\times C)))$$



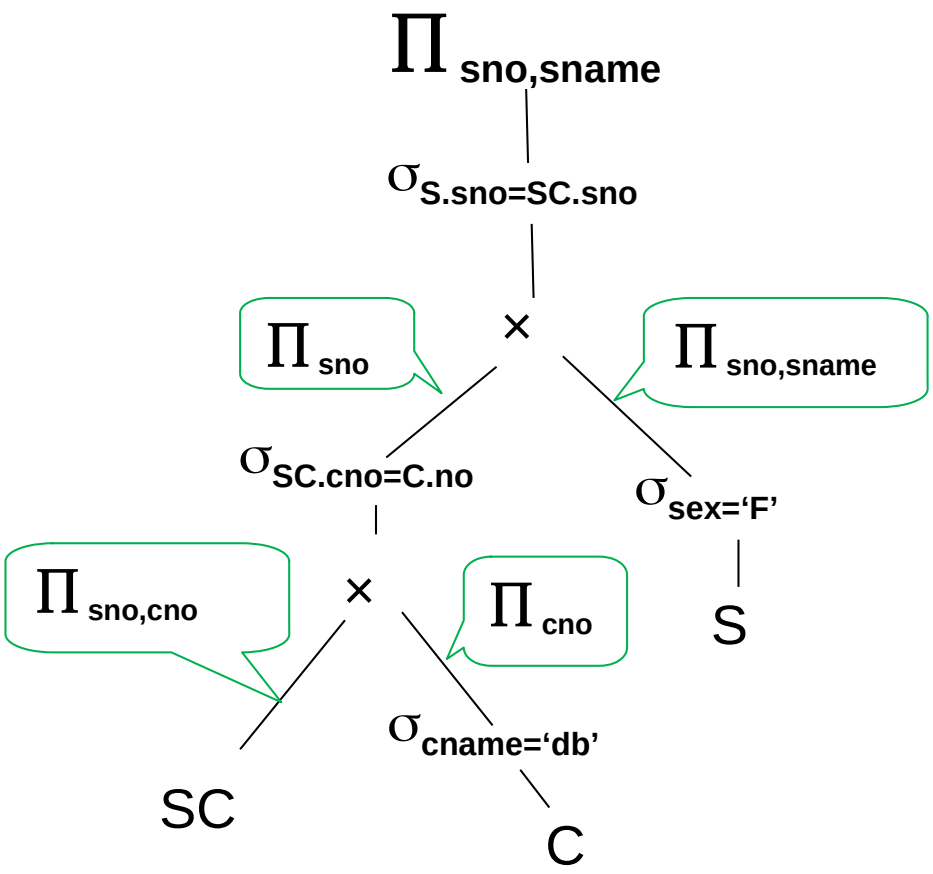
初始语法树


Π sno,sname
σ cname='db'^sex='F'
σ S.sno=SC.sno^SC.cno=C.no
×
×
S
SC
C

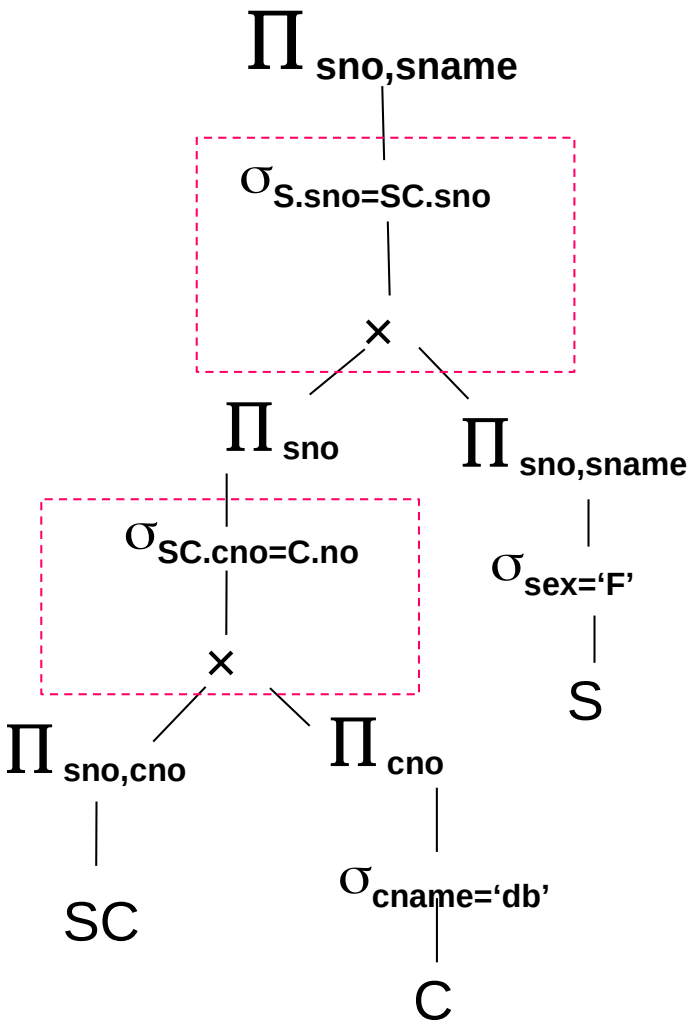
条件分解


Π sno,sname
σ S.sno=SC.sno
×
σ SC.cno=C.no
σ sex='F'
×
S
σ cname='db'
SC
C

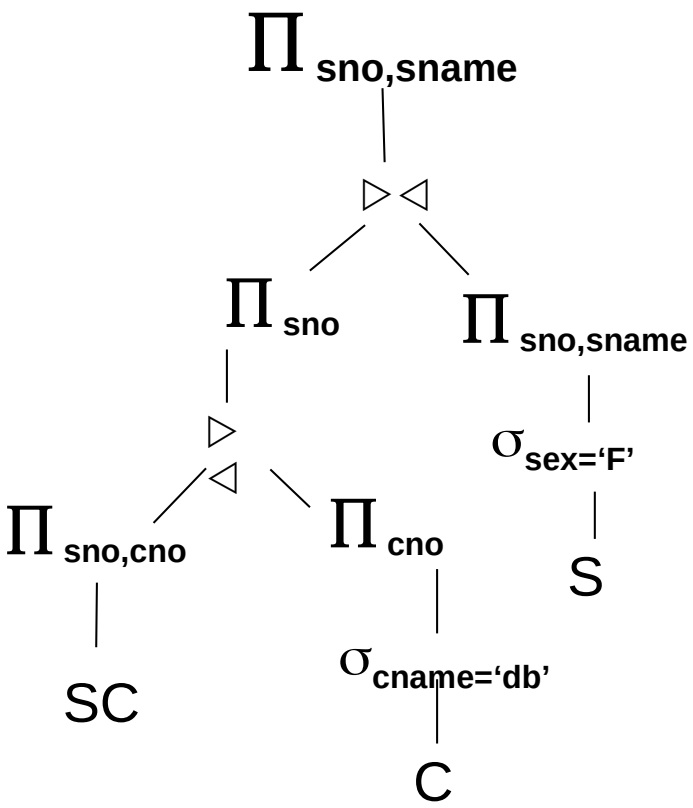
条件下移



投影前移



笛卡尔积和选择合成连接


Π sno,sname
Π sno
Π sno,sname
σ sex='F'
Π sno,cno
Π cno
S
SC
σ cname='db'
C

最终语法树

2.5 关系演算 **

- 2.5.1 元组关系演算语言 ALPHA
- 2.5.2 域关系演算语言 QBE

2.6 关系数据库管理系统

- 关系数据库管理系统简称关系系统
- 一个数据库管理系统可成为关系系统的最小条件
  - 关系数据库（即关系数据结构）
  - 支持选择、投影和连接运算
- E.F.Codd 思想对关系系统的分类（ P63 图 2-5 ）
  - 表式系统：仅只是关系数据结构，不支持集合级操作
  - 最小关系系统：支持关系结构和选择、投影、连接集合操作
  - 关系完备系统：支持关系结构和所有关系代数操作
  - 全关系系统：支持关系模型的所有特征。

第三章 关系数据库标准查询语言 SQL

- 3.1 SQL 概述
- 3.2 数据定义语言（ DDL ）
- 3.3 SQL 的数据查询（ DML ）
- 3.4 SQL 的数据更新（ DML ）
- 3.5 视图
- 3.6 数据控制语言（ DCL ）
- 3.7 嵌入式 SQL 语言
- 3.8 存储过程 *（ T-SQL ）

3.1 SQL 概述

- SQL(Standard/Structured Query Language) 是关系数据库标准
- 1986 年 10 月，美国国家标准局（ American National Standard Institute ANSI ）公布第一个标准 ANSI X3.135-1986 ，国际标准化组织（ International Organization for Standardization ISO ）也通过这一标准称 SQL - 86
- 1989 年 ANSI 再次公布标准 ANSI X3.135-1989 ， ISO 相应 SQL - 89
- 1999 年， ISO 公布 SQL-1999（ SQL99 ， SQL 3 ）
- 2003 年， ISO 公布 SQL-2003

3.1.1 SQL 语言的组成

- 数据定义（ DDL Data Definition/Description Language ）
  - 定义数据库的逻辑结构，包括基本表、视图、索引等
- 数据操纵（ DML Data Manipulation Language ）
  - 包括查询和更新，更新又包含插入、删除和修改
- 数据控制（ DCL Data Control Language ）
  - 授权、完整性规则描述、事务控制等
- 嵌入式 SQL （ ESQL ）
  - 在宿主语言中使用 SQL 的规则

3.1.2 SQL 语言的特点

- 综合统一：集 DDL、 DML、 DCL 于一体，语言风格统一
- 面向集合的操作方式：操作对象、查询结果都可以是元组的集合
- 高度非过程化：存取路进透明。
- 以统一的语法结构提供两种使用方式：自含式、嵌入式
- 语言简洁，易学易用，主要共使用 11 个关键词
  - DDL ： create drop alter
  - DML ： select insert delete update
  - DCL ： grant revoke commit rollback

3.2 数据定义语言（ DDL ）

3.2.1 定义、删除与修改基本表

- 定义基本表语法

  CREATE TABLE < 表名 > (< 列名 >< 数据类型 >[列级约束条件][,< 列名 >< 数据类型 >[列级约束条件]... ...] [,< 表级完整性约束条件 >])

例： CREATE TABLE S （

S # CHAR （ 5 ） ，

SN CHAR （ 20 ） NOT NULL UNIQUE ，

SA INT ，

SD CHAR （ 3 ），

PRIMARY KEY (S#)

）；

- 修改表语法

  ALTER TABLE < 表名 >

  [ADD < 新列名 >< 数据类型 >[列级约束条件]]

  [DROP < 完整性约束条件 >]

  [MODIFY < 列名 >< 数据类型 >];

例：

  ALTER TABLE S ADD SCome DATE ；

  ALTER TABLE S MODIFY SA SMALLINT ；

  ALTER TABLE S DROP UNIQUE(S#) ；

- 删除表语法

  DROP TABLE < 表名 >

例：

  DROP TABLE S

3.2.2 建立和删除索引

- 索引的建立语法

  CREATE [UNIQUE][CLUSTER] INDEX < 索引名 >

  ON < 表名 > （ < 列名 1>[< 次序 >][，< 列名 2>< 次序 >... ...] ）

  < 次序 > 可以是 ASC 和 DESC

例：

CREATE UNIQUE INDEX S_S# ON S(S#)

CREATE UNIQUE INDEX C_C# ON C(C#)

CREATE UNIQUE INDEX SC_S#_C# ON SC(S# ASC,C# DESC)

- 索引的删除语法

  DROP INDEX [< 表名 >.]< 索引名 >

  DROP INDEX [S.]S_S#

3.3 SQL 的数据查询（ DML ）

- 关系代数表达式

  $\prod_{A1,A2,.....An}(\sigma_F (R_1 \times R_2 \times ... ... \times R_n))$

- SQL 语句

  SELECT A_1 ， A_2 ， .....A_n

  FROM R_1 ， R_2 ， ... ...R_m

  WHERE F

详细语法

SELECT [ALL|DISTINCT] {*|< 目标表达式 1> [,< 目标表达式 2> ... ...]}

FROM < 表名或视图名 1> [， < 表名或视图名 2>]... ...

[WHERE < 条件表达式 >]

[GROUP BY < 列名表达式 1>[， < 列名表达式 2>]]
[HAVING < 条件表达式 >]

[ORDER BY < 列名表达式 1> [ASC|DESC]], < 列名表达式 2> [ASC|DESC]]

执行过程

1) 先按 WHERE 子句条件从 FROM 子句指定的表 / 视图中　找出满足条件的元组（选择）；

2) 如有 GROUP 子句，则将结果按 < 列名表达式 > 的值分　组，该 < 列名表达式 > 值相等的元组为一个组，通　　常会在每组中使用聚合函数。

3) 如果 GROUP 子句带 HAVING 子句，则对组过虑，将　　满足条件的组输出

4) 再按 SELECT 子句中的目标表达式选择出元组中的　　属性，形成结果表（投影）；

5) 如果 ORDER 子句，则将结果按 < 列名表达式 1> 的值　　升序或降序排列

3.3.1 单表查询

假设：　　S（S#，SN，SS，SA SD）

C（C#，CN，CP，CR）

SC（S#，C#，GR）

- 选取表中的某些列，即投影运算
  - 查指定列

    SELECT S#,SN FROM S
  - 查全部列

    SELECT * FROM STUDENT
  - 查经过计算的列

    SELECT SN，2015-SA FROM S

- 选择表中的若干元组，即选择运算 表
  - 消除取值重复行

    SELECT DISTINCT SD FROM S
  - 查询满足条件的元组
    - 比较大小：< 、 <= 、 > 、 >= 、 = 、 <>

      SELECT SN,SA FROM S WHERE SD=’CS’

      SELECT * FROM S WHERE SA<20
    - 确定范围： BETWEEN... AND

      SELECT * FROM S WHERE SA BETWEEN 20 AND 21
    - 确定集合： IN

      SELECT * FROM S WHERE SD IN (‘CS’,’IS’,’MA’)
    - 字符匹配： LIKE ，转义字符’ \’

      SELECT * FROM S WHERE S# LIKE ‘TB%’

      SELECT * FROM S WHERE SN LIKE ‘ 刘 _’
    - 涉及空值的查询： IS NULL

      SELECT * FROM SC WHERE GR IS NULL
    - 多重条件查询：

      SELECT * FROM S WHERE SD=’CS’ AND SA<20

- 查询结果排序

  ORDER BY < 字段表达式 > ASC|DESC

  SELECT * FROM SC WHERE C#=’3’ ORDER BY GR DESC

- 使用集 (聚合) 函数

  COUNT ， SUM， AVG， MAX， MIN

  SELECT COUNT(*) FROM S

  SELECT COUNT(DISTINCT S#) FROM SC

  SELECT AVG(GR) FROM SC WHERE S#=’95001’

  SELECT MAX(GR) FROM SC WHERE C#=’1’

- 查询分组： GROUP BY

  SELECT C#,COUNT(*) FROM SC GROUP BY C#

  SELECT S# FROM SC GROUP BY S# HAVING COUNT(*) >3

  检索选修 >3 门的课学生学号

3.3.2 连接查询

- 等值与非等值连接查询 自然连接

  SELECT S.*,SC.* FROM S,SC WHERE S.S# = SC.S#

- 自身连接

  检索每门课的间接预修课

  SELECT f.C#, s.CP FROM C f,C s WHERE f.CP=s.C#

C #	CN	CP	CR
1	DB	5	4
2	MA		2
3	IS	1	4
4	OS	6	3
5	DataStruct	7	4
6	DataProcess		2
7	PASCAL	6	4

C #	CN	P	CR
1	DB	5	4
2	MA		2
3	IS	1	4
4	OS	6	3
5	DataStruct	7	4
6	DataProcess		2
7	PASCAL	6	4

表

➢ 外连接

列出所有学生的修课情况，如果没有选课也列出其基本信息 (左外连接)

SELECT S#,SN,SS,SA,SD,C#,GR FROM S, SC

WHERE S.S# *=SC.S# (T-SQL 语法 SYBASE)

SELECT S#,SN,SS,SA,SD,C#,GR FROM S, SC

WHERE S.S# =SC.S#(+) (PL/SQL 语法 ORACLE)

SELECT S#,SN,SS,SA,SD,C#,GR

FROM S LEFT OUTER JION SC ON S.S#=SC.S#

(MYSQL MSSQL)

表

复合条件连接

检索选修课程号‘ 2’ 且成绩在 90 分以上的所有学生

```
SELECT S.S# ,SN FROM S,SC
WHERE S.S# = SC.S# AND SC.C#=’2’ AND SC.GR>=90
```

检索每个学生选修的课程名及其成绩

```
SELECT S.S#,SN,C.CN,SC.GR from S,SC,C
WHERE S.S# = SC.S# AND SC.C# = C.C#
```

表

3.3.3 嵌套查询

- 带 IN 谓词的子查询

检索与“刘晨”同在一系的学生信息

```
SELECT S#,SN,SD FROM S WHERE SD IN
(SELECT SD FROM S WHERE SN = ‘刘晨’ )
```

本例可以通过自连接来实现

```
SELECT s1.S#, s1.SN, s1.SD FROM S s1, S s2
WHERE s1.SD = s2.SD AND s2.SN=’ 刘晨’
```

表

检索选修了课程名的为‘ MA’ 的学生学号和姓名

```
SELECT S#, SN FROM S WHERE S# IN
(SELECT S# FROM SC WHERE C#  IN
(SELECT C# FROM C WHERE CN=’MA’) )
```

本例同样可以用连接来实现

```
SELECT S#,SN FROM S ,SC,C
WHERE S.S# = SC.S# AND SC.C# = C.C#
AND C.CN=’MA’
```

- 带比较运算的子查询

  当确定子查询的返回值是唯一时，可以使用比较运算符 (注意子查询在比较符后)

  SELECT S#,SN FROM S WHERE SD=

  (SELECT SD FROM S WHERE CN=’刘晨’)

- 带 ANY 和 ALL 的子查询（子查询返回多值时用）

检索其他系中比 IS 系任一学生年龄小的学生名单

```
SELECT S # ,SN FROM S WHERE SA < ANY
(SELECT SA FROM S WHERE SD = ' IS')
AND SD<>'IS'
ORDER BY SA DESC
```

等价于

```
SELECT S # , SN FROM S WHERE SA <
(SELECT MAX ( SA ) FROM S WHERE SD = ' IS')
AND SD <> 'IS'
ORDER BY SA DESC
```

检索其他系中比 IS 系所有学生年龄都小的学生名单

SELECT S ＃ ， SN FROM S WHERE SA < ALL
(SELECT SA FROM S WHERE SD ＝‘ IS’)
AND SD<>‘IS’

等价于

SELECT S ＃ ， SN FROM S WHERE SA <
(SELECT MIN （ SA ） FROM S WHERE SD ＝‘ IS’)
AND SD <> ‘IS’
ORDER BY SA DESC

- 带 EXISTS 的子查询（不返回任何数据，只返回 Ture 和 False ）

  检索所有选修了课程号为‘ 1’ 的学生姓名

  SELECT SN FROM S WHERE EXISTS

  (SELECT * FROM SC WHERE S# = S.S# AND C# = ‘1’)

  - 注意：此例中子查询的查询条件依赖于外层父查询，称此类查询为相关子查询 (corelated subquery)。

  等价连接实现：

  SELECT SN FROM S,SC WHERE S.S# = SC.S# AND C# = ‘1’

- SQL 中没有 $(\forall x)p$ ，故须转换为 $\neg(\exists x(\neg p))$

如检索选修了全部课程的学生，即没有一门课没有选的学生

```
SELECT SN FROM S WHERE NOT EXISTS
(SELECT * FROM C WHERE NOT EXISTS
(SELECT * FROM SC WHERE C#  =  C.C# AND S# = S.S#))
```

- p->q 应被等价为 $\neg p \vee q$

如检索至少选修了学生 S001 选修的全部课程的学生

令 p=‘ 学生 S001 选修了 y’ q = ‘学生 x 选修了 y’

$(\forall y)$（ p->q ）$=\neg\exists y(\neg(p\text{->}q))= \neg\exists y(\neg(\neg p \vee q))= \neg\exists y(p \wedge \neg q))$

即不存在这么一门课程，学生 S001 选修了而 x 没有选修

```
SELECT SN FROM S WHERE NOT EXISTS(
SELECT * FROM SC SC2 WHERE S# = ‘S001’ AND
NOT EXISTS (SELECT * FROM SC WHERE C# =
   SC2.C# AND S# = S.S#))
```

3.3.4 集合查询

- 使用交、并、差的集合运算概念，INTERSECT，UNION ，MINUS
  - 检索计算机科学系及年龄不大于 19 岁的学生

SELECT * FROM S WHER SD=’CS’

UNION SELECT * FROM S WHERE SA< = 19

等价于：

SELECT * FROM S WHERE SD =‘ CS’ OR SA <=19

  - 检索选修了课程号为 C01 或 C02 的学生学号

SELECT S# FROM SC WHERE C # =‘ C01’

UNION SELECT S# FROM SC WHERE C # =‘ C02’

等价于：

SELECT S# FROM SC WHERE C# IN (‘C01’ ,‘ C02’)

❑检索同时选修了课程号为 C01 和 C02 的学生学号

SELECT S# FROM SC WHERE C ＃ = ‘ C01’

INTERSECT

SELECT S# FROM SC WHERE C ＃ = ‘ C02’(仅 ORACLE)

等价于：

SELECT S# FROM SC WHERE C# = ‘C01’ AND S# IN

(SELECT S# FROM SC WHERE C# = ‘C02’)

❑检索选修了课程号为 C01 而未选修 C02 的学生学号。

SELECT S# FROM SC WHERE C# = ‘ C01’

MINUS

SELECT S# FROM SC WHERE C# = ‘ C02’(仅 ORACLE)

等价于：

SELECT S# FROM SC WHERE C# = ‘ C01’AND

S# NOT IN (SELECT S# FROM SC WHERE C＃ = ‘ C02’)

3.4 SQL 的数据更新（ DML ）

3.4.1 数据插入

- 插入单个元组

  语法：

  INSERT INTO < 表名 >[(< 列名 1> [, < 列名 2>].....)]

  VALUES(< 常量 1>[,< 常量 2>].....)

  INSERT INTO S VALUES(’S001’,‘ 张三’ ,‘ 男’ ,18,‘IS’)

- 插入子查询结果

  语法：

  INSERT INTO < 表名 > [(< 列名 1> [, < 列名 2>].....)]< 子查询 >

  例：为所有学生插入一条选修 C01 课程的记录

  INSERT INTO SC SELECT S#,‘C01’,null FROM S

3.4.2 数据修改

- 语法 :

  UPDATE < 表名 > SET < 列名 > = < 表达式 >[, < 列名 > =

  < 表达式 >]..... [WHERE < 条件 >];

- 修改某一个元组的值

  将学生 S001 的年龄该为 22 岁

  UPDATE S SET SA=22 WHERE S# =’S001’

- 修改多个元组的值

  将所有的学生年龄增加 1 岁

  UPDATE S SET SA = SA + 1

- 带子查询的修改语句

  将计算机科学系所有的学生成绩置零

  UPDATE SC SET GR=0

  WHERE ‘CS’ = (SELECT SD FROM S WHERE S# = SC.S#) (相关子查询)

  不同的 DBMS 可以使用 join 实现同样功能，如 SYBASE

  UPDATE SC set GR=0 from S where S.S#=SC.S#

  and SD=’CS’

  或 UPDATE SC set GR=0 where S# in

  (SELECT S# from S where SD=’CS’)

- 修改操作与数据库的一致性

  如同时修改 S 和 SC 两表中的 S# 的值

  为了保证数据库的一致性，引入事务概念

3.4.3 数据删除

- 语法：

  DELETE FROM < 表名 > [WHERE < 条件 >];

- 删除某一个元组的值

  删除学号为 S001 的学生

  DELETE FROM S WHERE S#＝‘S001’

- 删除多个元组的值

  删除所有学生的选课记录

  DELETE FROM SC

- 带子查询的删除语句

  删除计算机科学系所有学生的选课记录

  DELETE FROM SC WHERE ‘CS’=(

  SELECT SD FROM S WHERE S#＝SC.S#) (相关子查询)

  DELETE from SC where S# in

  (SELECT S# from S where SD=’CS’) (非相关子查询)

3.5 视图

- 视图只是一个窗口，其数据依赖于基本表

3.5.1 定义视图

- 建立视图
- 语法 ：

  CREATE VIEW < 视图名 > [(< 列名 1>[， < 列名 2 >.....])]

  AS < 子查询 > [WITH CHECK OPTION]

  - 列名在以下情况必须列出
    - 子查询的目标列是集函数等，不是单纯的列
    - 多表连接时出现同名的列作为视图字段
    - 需要在视图中启用新的名字

- WITH CHECK OPTION 表示对视图更新时自动验证子查询条件
- 行列子集视图：若一个视图是从单个基本表导出的，并且只是去掉了基本表的某些行和某些列，但保留了码，称行列子集视图
- 例：建立学生视图

```
CREATE VIEW IS_S AS
SELECT S#, SN , SA FROM S
WHERE SD=’IS’
```

- 建立信息系选修了 C1 号课程的学生视图

```
CREATE VIEW IS_S1 (S#, SN ,GR)          AS
SELECT S.S# ,SN ,GR FROM S, SC
WHERE S.S# = SC.S# AND S.SD=’IS’
AND SC.C# = ‘C1’
```

- 视图之上可以建立视图
  - 建立选修了 C1 课程且成绩在 90 以上的学生视图

    CREATE VIEW IS_S2 AS

    SELECT S#, SN, GR FROM IS_S1 WHERE GR>=90

- 其它视图建立例子
  - 建立一个反映学生出生年月的视图

    CREATE VIEW BT_S（S#，SN，SB）AS

    SELECT S#, SN,2003 - SA FROM S
  - 建立一个学生学号和平均成绩的视图

    CREATE VIEW S_G（S#，AVG_GR）AS

    SELECT S#, AVG(GR) FROM SC GROUP BY S#

- 建立一个女学生的视图

  CREATE VIEW S_F (S#，SN，SS，SA，SD) AS

  SELECT * FROM S WHERE SS=’女’

  本视图在 S 表结构改变时会出错，解决办法是去掉列说明或改 * 为列表

➢ 删除视图

● 语法

- DROP VIEW < 视图名 >

  例子：删除视图 IS_S

  DROP VIEW IS_S

3.5.2 查询视图

- 把对视图的查询转化为对基本表的查询称为视图的消解 (View Resolution)
  - SELECT S ＃ ， SA FROM IS_S WHERE SA <20

  消解为 :

  SELECT S# ,SA FROM S WHERE SD=’IS’ AND SA <20

  - SELECT * FROM S_G WHERE AVG_GR>90

  消解为 :

  SELECT S#, AVG(GR) FROM SC WHERE AVG(GR)>90
  GROUP BY S#（错误）

  SELECT S#, AVG(GR) FROM SC GROUP BY S#
  HAVING AVG(GR)>90 （正确）

3.5.3 更新视图

- 视图的修改
  - 将信息系学生视图中学号为 S001 的学生姓名改为‘刘辰’

  UPDATE IS_S SET SN=‘ 刘辰’ WHERE S#=‘S001’

  视图消解为：

  UPDATE S SET SN=‘ 刘辰’ WHERE S#=‘S001’AND SD=‘IS’
- 视图的插入
  - 在信息系学生视图中插入记录

  INSERT INTO IS_S VALUES (‘S001’,‘ 刘辰’ ,20)

  视图消解：

  INSERT INTO S VALUES (‘S001’,‘ 刘辰’ ,NULL,20,‘IS’)

- 视图的删除：
  - 在信息系学生视图中插入记录

  DELETE FROM IS_S WHERE S#=‘S001’

  视图消解 ：

  DELETE FROM S WHERE S#=‘S001’AND SD=‘IS’

- 某些带聚合 / 集函数的视图是不可修改的

  例如 :

  UPDATE S_G SET AVG_GR=80 WHERE S#=‘S001’

  (错误)

- 不运行更新的视图规则
  - 由两个以上基本表导出的视图
  - 视图的字段来自常数或表达式，只运行 DELETE
  - 视图的字段来自集函数
  - 视图中含有 GROUP BY 子句
  - 视图中含有 DISTINCT 语句
  - 视图定义有嵌套查询，且内层查询涉及到导出本视图的基本表
  - 不允许更新的视图上定义的视图

3.5.4 视图的用途

- 视图能简化用户的操作
- 视图可以使用户多角度看待同一数据
- 视图对重构数据库提供了一定的逻辑独立性
  - 例如： S(S#, SN, SS, SA, SD) 需要拆分为

    SX(S# ， SN, SS, SA)， SY(S#, SD)

    则可以通过视图来保证应用不需改变

    CREATE VIEW S AS

    SELECT SX.S#, SN, SS, SA, SD

    FROM SX, SY WHERE SX.S# = SY.S#
- 视图能对数据提供安全保护

3.6 数据控制语言（ DCL ）

3.6.1 授权

- 语法

  GRANT {ALL PRIVILEGES|< 权限 >[,< 权限 >... ...]}
  [ON < 对象类型 > < 对象名 >]
  TO {PUBLIC|< 用户 >[,< 用户 >]... ...}
  [WITH GRANT OPTION];

- 示例：
  - GRANT SELECT ON TABLE S TO USER1;
  - GRANT ALL Privileges ON TABLE S, C TO U2,U3;
  - GRANT SELECT ON TABLE SC TO PUBLIC;
  - GRANT UPDATE(SD),SELECT ON TABLE S TO U4;
  - GRANT INSERT ON TABLE SC TO U5 WITH GRANT OPTION;
  - GRANT CREATETAB ON DATABASE S_C TO U8;

3.6.2 收回权限

- 语法

  REVOKE {ALL PRIVILEGES|< 权限 >[,< 权限 >... ...]}

  [ON < 对象类型 > < 对象名 >]

  FROM {PUBLIC|< 用户 >[,< 用户 >]... ...};

- 示例

  REVOKE SELECT ON TABLE SC FROM PUBLIC;

  REVOKE UPDATE(SD),SELECT ON TABLE S FROM U4;

  REVOKE INSERT ON TABLE SC FROM U5;

3.7 嵌入式 SQL 语言

- SQL 语言是非过程的，而应用大多是过程化的，故通过高级语言来弥补 SQL 过程控制的不足 .

- 将 SQL 嵌入 (Embedded SQL) 高级语言（宿主语言）来执行，称嵌入式 SQL 语言 (简称 ESQL)。

3.7.1 嵌入式 SQL 的一般形式

- 对于 ESQL 的处理，DBMS 一般有两种处理方式：
  - 预编译
  - 修改和扩充宿主语言以处理 SQL
- ESQL 一般形式：EXEC SQL <SQL 语句 >
- ESQL 根据其作用不同分为两类 ：
  - 可执行语句
  - 说明性语句

  可执行语句包括 DDL ， DML ， DCL 。两类 SQL 语句应和宿主语言两类语句出现在同一地方

3.7.2 嵌入式 SQL 语句与主语言之间的通信

- 数据库工作单元和主语言工作单元之间的通信有
  - 向主语言传递 SQL 语句的执行状态
  - 主语言向 SQL 语句提供参数
  - 将 SQL 语句查询数据库结果交主语言进一步处理
- 相应地通过 SQLCA 、主变量、游标来实现

① SQL 通信区

- SQLCA （ SQL Communication Area ）是一个数据结构，定义语句

  EXEC SQL INCLUDE SQLCA ;

- SQLCODE 反映每次执行 SQL 语句的结果

② 主变量

- 主要功能： ESQL 可以使用主语言的变量来输入和输出数据
- 分类：输入、输出主变量、指示变量
- 使用方法
  - 所有主变量在定义区定义（ BEGIN DECLARE SECTION， END DECLARE SECTION ）oracle 中可以在之外定义。
  - 可以在 SQL 中任意表达式的对方出现
  - 在 SQL 语句中，主变量前要加‘：’, 而在主语言中不必加。
  - 指示变量用于为输入变量赋空值或指示输出变量是否空值

③ 游标

- 使用原因： SQL 语句是面向集合的，而主语言是面向记录的
- 主语言和 SQL 语言的分工
  - SQL 语言负责直接与数据库打交道
  - 主语言用来控制程序流程以及对 SQL 的执行结构进一步处理
  - SQL 语言用主变量从主语言接受执行参数操作数据库 - >SQL 语言的执行状态由 DBMS 送至 SQLCA-> 主语言从 SQLCA 取出状态信息，据此决定下一步操作。
  - SQL 的执行结果通过主变量或游标传给主语言处理

3.7.3 不使用游标的 SQL 语句

- 说明性语句
- 数据定义语句
- 数据控制语句
- 查询结果为单记录的 SELECT 语句
- 非 CURRENT 形式的 UPDATE 语句
- 非 CURRENT 形式的 DELETE 语句
- INSERT 语句

1 ）说明性语句

EXEC SQL INCLUDE SQLCA ；

EXEC SQL BEGIN DECALRE SECTION ；

EXEC SQL END DECALRE SECTION

2 ）数据定义语句

EXEC SQL CREATE TABLE S(

S# char(10), SN char(10), SS char(2),

SA int, SD char(5));

EXEC SQL DROP TABLE ；

– 数据定义语句中不允许使用主变量

EXEC SQL DROP TABLE :tablename; （错误）

3）数据控制语句

授权： EXEC SQL GRANT SELECT ON TABLE S TO U1

出错处理： EXEC SQL WHENEVER SQLERROR do sql_error()

连接数据库：

EXEC SQL CONNECT :user IDENTIFIED BY :pass USING :tnsname;

4）查询结果为单条记录 SELECT 语句

- 语法

EXEC SQL SELECT [ALL|DISTINCT] {*|< 目标表达式 1>
[,< 目标表达式 2> ... ...]} INTO < 主变量 1> [< 指示变量 1>]
[,< 主变量 1> [< 指示变量 1>].....]
FROM < 表名或视图名 1> [， < 表名或视图名 2>]... ...
[WHERE < 条件表达式 >]
[GROUP BY < 列名表达式 1>[， < 列名表达式 2>]]
[HAVING < 条件表达式 >]
[ORDER BY < 列名表达式 1> [ASC|DESC][,
< 列名表达式 2> [ASC|DESC]]

- 例：

```
EXEC SQL SELECT S#, SN INTO :sno, :sn
                    FROM S WHERE S# =:GivenSno
```

- 注意：

– into ， where 和 having 子句中均可以使用主变量，需要事先申明。

– 返回值某列为 NULL 时，系统会将指示变量赋值为 -1 ，主变量不变。

– 如查询结果没有满足条件的记录，则 DBMS 置 sqlcode 值为 100 ，正常有结果为 0 。

– 如结果不止单条，程序出错， SQLCA 中包含返回信息

5 ）非 CURRENT 形式的 UPDATE 语句

– 将课程 C01 全部提分

```
EXEC SQL UPDATE SC SET GR=GR+:Raise
WHERE C# =’C01’
```

– 重新设置某个学生成绩

```
EXEC SQL UPDATE SC SET GR=:newgr
WHERE S# =’S001’
```

– 将计算机系所有同学成绩置空

```
Grid=-1
EXEC SQL UPDATE SC SET GR=:newgr  :grid WHERE
S#  IN  （ SELECT S# FROM S WHERE SD=‘CS’ ）
```

等价：

```
EXEC SQL UPDATE SC SET GR=NULL WHERE
S#  IN  （ SELECT S# FROM S WHERE SD=‘CS’ ）
```

6 ）非 CURRENT 形式的 DELETE 语句

- 例
  - 删除某学生的选课情况

```
EXEC SQL  DELETE FROM SC WHERE S# IN
(SELECT S# FROM S WHERE SN=:sname)
```

或者 :

```
EXEC SQL  DELETE FROM SC WHERE :sname=
(SELECT SN FROM S WHERE S.S# = SC.S#)
```

7 ）INSERT 语句

- 例：
  - 某个学生选修了一门课程

```
grid=-1;
EXEC SQL INSERT INTO SC
VALUES (:sno, :cno, :gr  :grid);
```

或者：

```
EXEC SQL INSERT INTO SC (S#, C#)
VALUES (:sno, :cno);
```

3.7.4 使用游标的 SQL 语句

- 使用游标的语句有
  - 查询结果为多条记录的 SELECT 语句
  - CURRENT 形式的 UPDATE 语句
  - CURRENT 形式的 DELETE 语句

1 ）查询结果为多条记录的 SELECT 语句

- 游标在 SELECT 语句的集合和主语言的一次只能处理一条记录之间架起桥梁

➢ 游标步骤

- 说明游标：仅仅是定义，并不执行查询

  EXEC SQL DECLARE < 游标名 > CURSOR FOR <SELECT 语句 >

- 打开游标：执行相应的查询，把结果放进缓冲区，并把指针指向第一条记录

  EXEC SQL OPEN < 游标名 >

- 读取当前记录并推进游标指针

  EXEC SQL FETCH < 游标名 >

  INTO < 主变量 >[< 指示变量 >][,< 主变量 >[< 指示变量 >].....]

- 关闭游标：释放缓冲区等资源

  EXEC SQL CLOSE < 游标名 > ；

- 例：查询某个指定系的所有学生情况

```
EXEC SQL INCLEDE SQLCA ；    // 说明性语句
EXEC SQL BEGIN DECLARE SECTION ；
   VARCHAR    depname[5] ；   /*oracle 中当作 struct 处理 */
   VARCHAR    HSno[10] ；      /* 用 char  更简单 */
   VARCHAR    HSname[10] ；
   VARCHAR    HSex[2] ；
   int        HSage ；
EXEC SQL END DECLARE SECTION ；
…
```

```
... ...
gets(depname) ;
... ...
EXEC SQL DECLARE SX CURSOR FOR      // 说明游标
            SELECT S# , SN ,SS , SA FROM S
            WHERE SD = :depname  ;
EXEC SQL OPEN SX  ;                                  // 打开游标
while (1){
   EXEC SQL FETCH SX INTO :Hsno,:Hsname,:Hsex,:HSage  ;
                                  // 读取当前记录并推进游标指针
   if  (  sqlca.sqlcode!=SUCCESS  )  break  ;
   printf("%s,%s,%s,%d\n", Hsno, Hsname, Hsex, HSage);
   ... ...
}
EXEC SQL CLOSE SX  ;     // 关闭游标
```

- :depname 值改变后可以重新打开游标，获得不同的集合。

2） CURRENT 形式的 UPDATE 和 DELETE 语句

- 操作步骤
  - 说明游标

    EXEC SQL DECLARE < 游标名 > CURSOR FOR <SELECT 查询 > **FOR UPDATE [OF < 列名 >]** ；
  - OPEN 游标
  - FETCH 游标
  - 检查是否要修改或删除，若是执行 DELETE 或 UPDATE ，并且使用 WHERE CURRENT OF < 游标名 >
  - 处理完毕 CLOSE 游标

- 例：检索某系的学生，根据要求处理数据

```
… …
EXEC SQL INCLEDE SQLCA ；
EXEC SQL BEGIN DECLARE SECTION ；
   VARCHAR    depname[5] ；
   VARCHAR    HSno[10] ；
   VARCHAR    HSname[10] ；
   VARCHAR    HSex[2] ；
   int              HSage ；
EXEC SQL END DECLARE SECTION ；
…. …
```

```
gets(depname) ;
... ...
EXEC SQL DECLARE SX CURSOR FOR
   SELECT S# , SN ,SS , SA FROM S
   WHERE SD =:depname
   [FOR UPDATE OF SA] ;
EXEC SQL OPEN SX ;
while (1){
   EXEC SQL FETCH SX INTO :Hsno, :Hsname,
   :Hsex, :Hsage;
   if ( sqlca.sqlcode!=SUCCESS ) break ;
```

```
    printf("%s,%s,%s,%d\n", Hsno, Hsname, Hsex, HSage);
    printf("UPDATE(U) or DELETE(D) or NO(N)?\n");
    scanf("%c",&op);
    if(op=='U'){
            printf("Input new age:");
            scanf("%d",&newage);
            EXEC SQL UPDATE S SET SA=:newage WHERE
            CURRENT OF SX;
    }
    else if(OP=='D')
    EXEC SQL DELETE FROM S WHERE CURRENT OF SX;
    else continue;
    ... ...
}
EXEC SQL CLOSE SX  ;
```

3.7.5 动态 SQL 语句（以 SYBASE 的 ESQL 为例）

- 在预编译时无法获得如下信息的必须使用动态 SQL 技术 ，未知信息可能包括：
  - SQL 语句正文
  - 主变量个数
  - 主变量数据类型
  - SQL 语句引用的数据对象

- 动态语句的四种实现方式（仅介绍方法一）
  - 特点：执行语句不能返回任何结果
  - 语法

    EXEC SQL EXECUTE IMMEDIATE {: host_variable | string}
  - 例：

```
EXEC SQL BEGIN DECLARE SECTION;
CS_CHAR sqlstring[200];
EXEC SQL END DECLARE SECTION;
char cond[150];
strcpy(sqlstring,"update titles set price=price*1.10 where ");
printf("Enter search condition:");
scanf("%s", cond);
strcat(sqlstring, cond);
EXEC SQL EXECUTE IMMEDIATE :sqlstring;
```

3.8 存储过程*（ T-SQL ）

- 语法

  create procedure [owner.]procedure_name
  [(@parameter_name datatype [= default][output]
  [, @parameter_name datatype [= default][output]]...)]
  [with recompile]
  as <*SQL_statements*>

- 语言要素：
  - 语句块

    begin
    <*statement block*>
    end

- 变量

  以 @ 开始的为用户变量，以 @@ 开始的为全局变量

  定义变量： DECLARE

  @@rowcount 操作影响的行数

  @@sqlstatus 游标 Fetch 的状态

- 条件控制

  if *logical_expression*

  statements

  [else

  [if *logical_expression*]

  statements]

– 循环控制

while *boolean_expression*

statement

break

statement

continue

– 顺序控制

label:

goto label

– 返回值

return [*integer_expression*]

- 打印信息

  print {format_string | @local_variable | @@global_variable} [, arg_list]

  select @local_variable | @@global_variable

- 执行

  [execute] [@return_status =] [[[server.]database.]owner.]procedure_name

  [[@parameter_name =] value | [@parameter_name =] @variable [output]

  [,[@parameter_name =] value|[@parameter_name =] @variable [output]...]]

  [with recompile]

- 例：给定学号，获得该学生成绩，若是 C01 课程，成绩加 1 ，否则加 2

```
CREATE PROCEDURE get_gr @sno varchar(10), @GR int OUTPUT
AS
DECLARE @cno varchar(5)
BEGIN
  SELECT top 1 @cno=C#,@GR=GR FROM SC WHERE S# = @sno
  IF (@cno =’C01’)
    select @GR=@GR+1
  ELSE
    select @GR=@GR+2
END
```

执行：

```
declare @gr int
execute get_gr ‘s001’,@gr  output
select @gr  或 print @gr
```

Oracle PL/SQL— 自含式环境 (sqlplus 等)

```
set serveroutput on
declare
   v_sid S.sid%type;    v_sname S.sn%type;
   v_sa      S.sa%type;  v_sa S.sd%type;   v_prom varchar(10);
   cursor c1 is select sid,sname,sa from S where sd=&v_sd;
begin
   open c1;
   loop
       fetch c1 into v_sid,v_sname,v_sa;
       exit when c1%notfound
      case
          when v_sa <18  then v_prom:=‘ 未成年’;
          else v_prom:=‘ 成年’;
      end case;
      dbms_output.put_line(v_sname||”:”||v_prom);
  end loop; close c1;
end;
```

Oracle PL/SQL--procedure

- CREATE OR REPLACE PROCEDURE myproc(
  parm1 IN number:=0,parm2 OUT number) as
  r1 TAB1%rowtype ;
  type t_rec2 is record(c1 TAB2.c1%type, c2 char(2));
  r2 t_rec2;

```
begin
  for r1 in (select * from TAB1) loop
     dbms_output.put_line(r1.field1);
     r2.c1=r2.c1+r1.filed1;
  end loop;
  select  max(field2) into r2.c2 from TAB1
end;
```

S

S#	SN	SS	SA	SD
95001	李勇	男	20	CS
95002	刘晨	女	19	IS
95003	王名	女	18	MA
95004	张立	男	18	IS

C

C#	CN	CP	CR
1	DB	5	4
2	MA		2
3	IS	1	4
4	OS	6	3
5	DataStruct	7	4
6	DataProcess		2
7	PASCAL	6	4

SC

S#	C#	GR
95001	1	92
95001	2	85
95001	3	88
95002	2	90
95002	3	80

SELECT S#,SN FROM S

S#	SN
95001	李勇
95002	刘晨
95003	王名
95004	张立

SELECT SN ， 2005-SA FROM S

SN	2015-SA
李勇	1995
刘晨	1996
王名	1997
张立	1997

SELECT DISTINCT SD FROM S

SD
CS
IS
MA
IS

SELECT * FROM SC WHERE C#=’3’ ORDER BY GR DESC

S#	C#	GR
95001	3	88
95002	3	80

SELECT C#,COUNT(C#) FROM SC GROUP BY C#

C#	COUNT(C#)
1	1
2	2
3	2

SELECT S# FROM SC GROUP BY S# HAVING COUNT(*) >3

S#
95001

SELECT S.*,SC.* FROM S,SC WHERE S.S# = SC.S#

S.S#	SN	SS	SA	SD	SC.S#	C#	GR
95001	李勇	男	20	CS	95001	2	85
95001	李勇	男	20	CS	95001	1	92
95001	李勇	男	20	CS	95001	3	88
95002	刘晨	女	19	IS	95002	2	90
95002	刘晨	女	19	IS	95002	3	80

SELECT f.C#, s.CP FROM C f,C s WHERE f.CP=s.C#

f.C #	s.CP
1	7
3	5
5	6

SELECT S#,SN,SS,SA,SD,C#,GR
FROM S LEFT OUTER JION SC ON S.S#=SC.S#

S#	SN	SS	SA	SD	C#	GR
95001	李勇	男	20	CS	2	85
95001	李勇	男	20	CS	1	92
95001	李勇	男	20	CS	3	88
95002	刘晨	女	19	IS	2	90
95002	刘晨	女	19	IS	3	80
95003	王名	女	18	MA	null	null
95004	张立	男	18	IS	null	null

SELECT S.S# ,SN FROM S,SC
WHERE S.S# = SC.S# AND SC.C#=’2’ AND SC.GR>=90

S#	SN	SC.C#	SC.GR
95002	刘晨	2	90

SELECT S.S#,SN,C.CN,SC.GR from S,SC,C
WHERE S.S# = SC.S# AND SC.C# = C.C#

S.S#	SN	C.C#	C.CN	SC.GR
95001	李勇	1	DB	92
95001	李勇	2	MA	85
95001	李勇	3	IS	88
95002	刘晨	2	MA	90
95002	刘晨	3	IS	80

第四章关系数据库设计理论

4.1 数据依赖

4.2 范式

4.3 关系模式的规范化

4.1 数据依赖

4.1.1 关系模式中的数据依赖

- 完整的关系模式的描述： R(U ， D ， DOM ， F)
  - R 关系名
  - U 属性组
  - D 是 U 的取值范围，是域的集合
  - DOM 是属性向域映象的集合
  - F 是属性间数据的依赖关系集合
- 关系模式是静态的、稳定的；关系是动态的，不同时刻关系模式中的关系可能不同，但关系都必须满足关系模式中数据依赖关系集合 F 指定的完整性约束
- 影响数据库模式设计的主要是 U 和 F ，所以一般关系模式简化为： R （ U ， F ）

4.1.2 数据依赖对关系模式的影响

- 数据依赖有：
  - 函数依赖、多值依赖和连接依赖
- 一个关系模式示例

  U = {Sno,Sdept,Mname,Cname,Grade}

  F = {Sno->Sdept, Sdept->Mname, (Sno,Cname)->Grade}
- 该关系模式存在如下问题：
  - 数据冗余太大：系主任名字重复出现，和所有学生的所有课程成绩次数一样
  - 更新异常：更换系主任必须修改每一个学生信息
  - 插入异常：刚成立的系如果还没有招生就无法存储系主任信息
  - 删除异常：某个系的学生全部毕业删除时会丢失系主任信息

4.1.3 相关概念

- 函数依赖

  R（U）是一个关系模式，U是R的属性集合，X和Y是U的子集，对于R（U）的任意一个可能的关系r，如果r中不存在两个元组w，v，使得w[X]=v[X]而w[Y]<>v[Y]，称X函数决定Y，或Y函数依赖于X，记X->Y

- 平凡的和非平凡的函数依赖

  关系模式R（U），X和Y是U的子集，如果X->Y，且Y⊄X，则称X->Y是非平凡的函数依赖，否则称平凡的函数依赖，我们讨论的都是非平凡的函数依赖

- 完全函数依赖和部分函数依赖

  关系模式 R（U），如果 $X \rightarrow Y$，且对于任意的 X 的真子集 X’ 都有 X’ \ Y，则称 Y 完全函数依赖于 X，记 $X \xrightarrow{f} Y$。反之则 Y 不完全依赖于 X, 称 Y 部分依赖于 X，记 $X \xrightarrow{P} Y$

- 传递函数依赖

  关系模式 R（U），如果 $X \rightarrow Y$，$Y \not\rightarrow Z$，且 Y \ X，则称 Z 传递函数依赖于 X，记 $X \xrightarrow{t} Z$

- 码的重新定义

  关系模式 R（U,F），K 为属性组合，若 $K \xrightarrow{f} U$，则 K 是一个候选码。

4.2 范式

- 范式定义

  数据依赖满足某种条件级别的关系模式的集合

- 目前共 6 种范式：

  1NF⊃2NF⊃3NF⊃BCNF⊃4NF⊃5NF

4.2.1 第一范式（ 1NF ）

- 1NF 定义
  - 如果一个关系模式 R 的所有属性都是原子的，即不可再分的基本数据项，则 R∈1NF

例： SCL(S # , SN, SA, CLS, MON, C # , CN, CRD, GR)

属于 1NF 它有以下问题：

- 数据冗余大，如 MON ， CRD 等
- 插入异常，当无课程时学生信息无法插入
- 删除异常，当某个学生的选课信息全部删除时无法保留学生基本信息

SCL 存在的函数依赖关系

$(S\#,C\#) \xrightarrow{f} GR$

$(S\#,C\#) \xrightarrow{P} SN \quad S\# \xrightarrow{f} SN$

$(S\#,C\#) \xrightarrow{P} SA \quad S\# \xrightarrow{f} SA$

$(S\#,C\#) \xrightarrow{P} CLS \quad S\# \xrightarrow{f} CLS$

$(S\#,C\#) \xrightarrow{P} CN \quad C\# \xrightarrow{f} CN$

$(S\#,C\#) \xrightarrow{P} CRD \quad C\# \xrightarrow{f} CRD$

$CLS \rightarrow MON \quad S\# \xrightarrow{t} MON$

4.2.2 第二范式（ 2NF ）

- 2NF 定义
  - 如果一个关系模式 R∈1NF ，并且每一非主属性都完全依赖于 R 的码，则 R∈2NF 。
  - 显然码只包含一个属性的 R 如果是 1NF ，则必是 2NF

例： S_L(S#,SN,SA,CLS,MON)　C(C#,CN,CRD)

S_C(S#,C#,GR) 都属于 2NF

存在问题：

- 数据冗余大，如 MON
- 插入异常，无学生信息无法插入班长信息
- 删除异常，当学生的信息删除无法保存班长

函数依赖关系：

$S\# \xrightarrow{f} SA$

$S\# \xrightarrow{f} CLS$

$S\# \xrightarrow{f} SN$

$CLS \longrightarrow MON \quad S\# \xrightarrow{t} MON$

$C\# \xrightarrow{f} CN$

$C\# \xrightarrow{f} CRD$

$(S\#,C\#) \xrightarrow{f} GR$

4.2.3 第三范式（ 3NF ）

- 3NF 定义
  - 如果一个关系模式 R 中不存在非主属性对码的传递依赖，则 R∈3NF

例：S（S #，SN，SA，CLS）

L（CLS，MON）

C（C #，CN，CRD）

S_C（S #，C #，GR）都属于 3NF

4.2.4 BC 范式（ BCNF ）

- BCNF 定义
  - 如果一个关系模式 R （ U,F ）∈ 1NF ，对 R 中的任意一个非平凡的函数依赖 X - >Y ， X 都含有候选码，则 R∈BCNF

例： STC （ S ，T ，C ） S 学生 T 教师 C 课程

（ S ，C ）- >T

（ S ，T ）- >C

T - >C

所以 STC 不是 BCNF

分解为： ST （ S ，T ），TC （ T ，C ）

则都属于 BCNF

4.2.5 第四范式（ 4NF ）

- 多值依赖
  - 关系模式 R （ U ）属性集 U ， X ， Y 和 Z 是 U 不相交的子集，且 Z = U - X - Y ，若关系模式 R 的任一关系 r 对于 X 的一个给定值，存在 Y 的一组值与之对应，且 Y 的这一组值与 Z 无关，称 Y 多值依赖于 X ，记 X - > - >Y 。当 Z 非空时称非平凡的多值依赖
- 4NF 定义
  - 如果一个关系模式 R （ U,F ）∈ 1NF ，对 R 中的任意一个非平凡的多值依赖 X - > - >Y ， X 都含有候选码，则 R∈4NF
  - 第四范式一般尽量将一个 3 元关系分解为两个 2 元关系，若不能在不丢失信息的前提下分解则称已达 4NF 。

例：

CTX（C，T，X） C 课程，T 教师，X 参考书

候选码：（C，T，X）

C ->->T，C ->->X，C 不是候选码，故 CTX 不属于 4NF

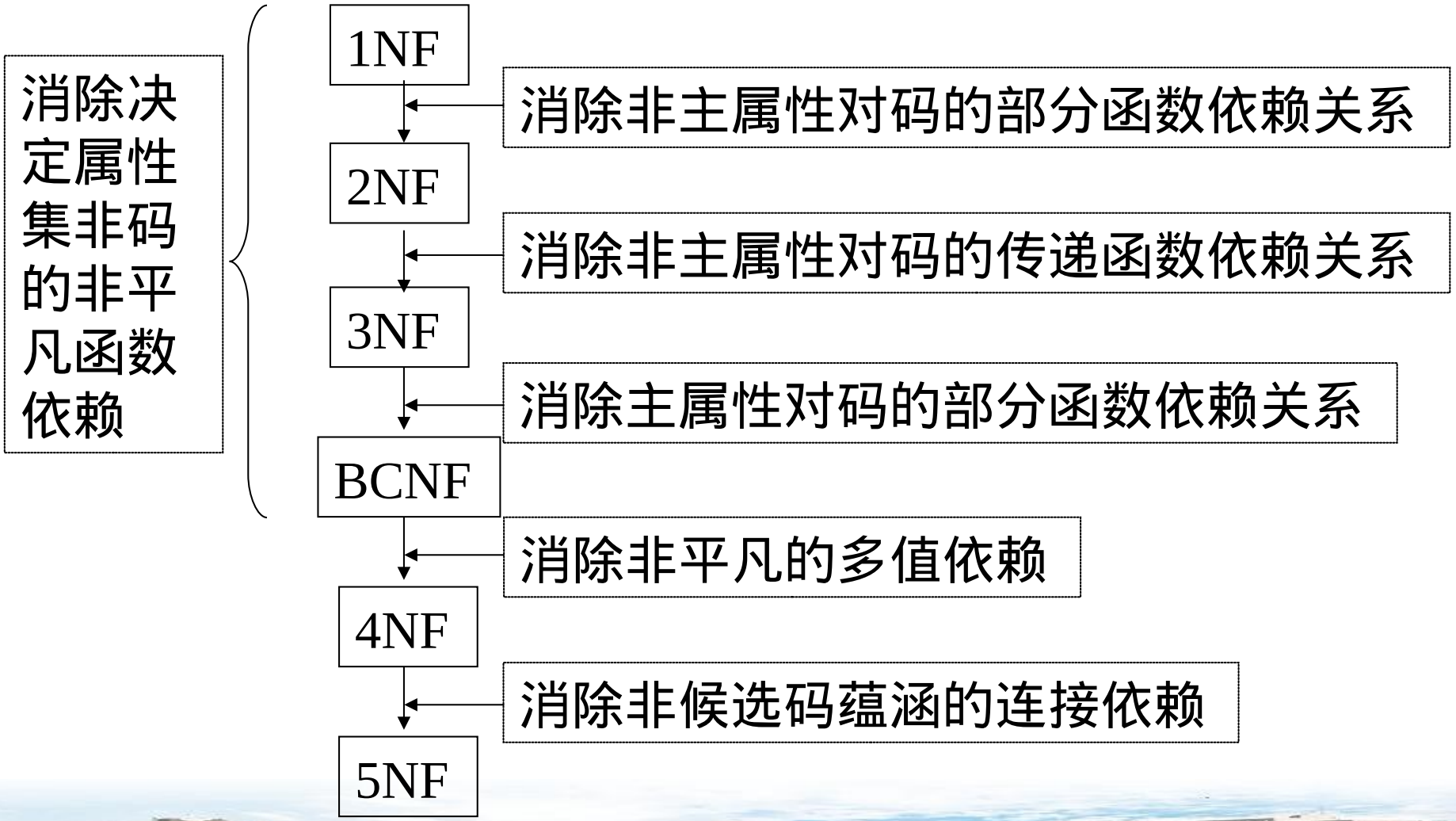
分解为 CT（C，T），CX（C，X），则都满足 4NF

4.2.6 第五范式（ 5NF ）

- 定义
  - 关系模式 R ，其属性集 U ， X1 ， X2.....Xn 分别为 U 的子集，∪ Xi = U ，如果对于 R 的每一个关系 r 都有 r = ∞ Xi ，则称连接依赖（ JD ）在关系模式 R 上成立，记为 *（ X1 ， X2 ， .....Xn ），若某个 Xi 就是 R ，称平凡的连接依赖。
- 定义
  - 如果一个关系模式 R （ U,F ）∈ 1NF ，对 R 中的任意一个连接依赖都都由候选码蕴涵，则 R∈5NF
  - 第五范式建议，最好把现有的三元关系分解为 3 个二元关系，但某些情况下不可以再分解，则称已处于 5NF ，若某个关系所有的链接依赖要么是平凡的，要么是每个分解都蕴含候选键，则称满足 5NF 。

4.3 关系模式的规范化

4.3.1 关系模式的规范化步骤



4.3.2 关系模式的分解

- 关系模式的规范化
  - 通过对关系模式的分解来实现的
  - 把低级别的关系模式分解为高级别的关系模式
  - 分解不唯一，只有保证分解后的关系模式与原关系模式等价，才有意义

- 例：

关系模式 SL （ Sno ， Sdept ， Sloc ）

Sno - >Sdept ， Sdept - >Sloc, Sno - >Sloc

设 SL 有如下关系

Sno	Sdept	Sloc
95001	CS	A
95002	IS	B
95003	MA	C
95004	IS	B
95005	PH	B

- 分解方法一：

  SN(Sno)， SD(Sdept)， SO(Sloc)

  SN， SD， SO 都是很高的范式，属于 5NF ，但分解后丢失很多信息

SN	SD	SO
Sno	Sdept	Sloc
95001	CS	A
95002	IS	B
95003	MA	C
95004		
95005	PH	

- 分解方法二：

NL(Sno ， Sloc)， DL(Sdept ， Sloc)

分解后的关系

NL		DL	
Sno	Sloc	Sdept	Sloc
95001	A	CS	A
95002	B	IS	B
95003	C	MA	C
95004	B	PH	B
95005	B		

NL 和 DL 的自然连接结果

Sno	Sdept	Sloc
95001	CS	A
95002	IS	B
95002	PH	B
95003	MA	C
95004	IS	B
95004	PH	B
95005	IS	B
95005	PH	B

- NL 和 DL 的自然连接结果
  - 多出三个元组，而实际上无法确定哪个是多余的，因此丢失了信息

- 定义
  - 如果关系模式 R（U，F）在分解为若干个关系模式 Ri（Ui，Fi）后（其中 U = ∪ Ui），若 Ri 的自然连接和原关系相等，则称该分解具有无损连接性

- 分解方法三

  ND(Sno ， Sdept)，NL(Sno ， Sloc)

  分解后关系

ND		NL	
Sno	Sdept	Sno	Sloc
95001	CS	95001	A
95002	IS	95002	B
95003	MA	95003	C
95004	IS	95004	B
95005	PH	95005	B

- NL 和 DL 的自然连接结果
  - 和原关系相同，但当某个学生转系后需要修改两个表，如 95005 转为 CS 系后需 PH - >CS ，B - >A 。原因是分解时丢失了 Sdept - >Sloc 的函数依赖关系。

- 定义
  - 如果在分解过程中原函数依赖 F 被每个分解后的某个关系函数依赖 Fi 所逻辑蕴涵，则称该分解是保持函数依赖的

- 分解方法四

ND(Sno ， Sdept)，DL(Sdept ， Sloc)

分解后关系

ND		DL	
Sno	Sdept	Sdept	Sloc
95001	CS	CS	A
95002	IS	IS	B
95003	MA	MA	C
95004	IS	PH	B
95005	PH		

NL 和 DL 的自然连接结果和原关系相同，该分解是保持函数依赖的

- 衡量关系分解有两个标准
  - 是否具有无损连接性
  - 是否保持了函数依赖

- 关系分解理论定理
  - 若要求关系模式分解具有无损连接性，则分解一定可达到 4NF
  - 若要求关系模式分解保持函数依赖，则分解一定可达到 3NF ，但不一定达到 BCNF
  - 若要求关系模式分解既具有无损连接性，又保持函数依赖，则分解一定可达到 3NF ，但不一定达到 BCNF

谢　谢

- 例：关系 AFP

AFP

A	F	P
a1	f1	p1
a1	f1	p2
a1	f2	p1
a2	f1	p2
a2	f3	p2

AF

A	F
a1	f1
a1	f2
a2	f1
a2	f3

FP

F	P
f1	p1
f1	p2
f2	p1
f3	p2

AP

A	P
a1	p1
a1	p2
a2	p2

AF∞FP

A	F	P
a1	f1	p1
a1	f1	p2
a1	f2	p1
a2	f1	p1
a2	f1	p2
a2	f3	p2

AF∞FP∞AP=AFP

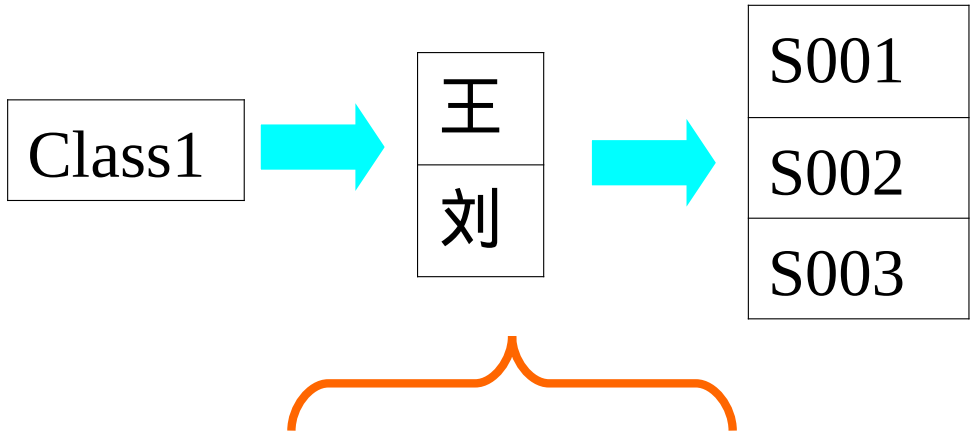
A	F	P
a1	f1	p1
a1	f1	p2
a1	f2	p1
a2	f3	p2
a2	f1	p1

只有 AF ， FP ， AP 三个关系连接才可以得到 AFP ，因此称 AFP 具有连接依赖 JD*(A,F,P)

4NF

班级	教师	学生
Class1	王	S001
Class1	王	S002
Class1	王	S003
Class1	刘	S001
Class1	刘	S002
Class1	刘	S003

非 4NF



班级	教师
Class1	王
Class1	刘

班级	学生
Class1	S001
Class1	S002
Class1	S003

第四范式总是尽量将一个 3 元关系分解为两个 2 元关系

5NF

教师	学生	课程
王	S001	语文
王	S002	数学
刘	S002	语文

非 5NF

教师	学生
王	S001
王	S002
刘	S002

教师	课程
王	语文
王	数学
刘	语文

王	S002	语文

第五范式总是尽力把三元关系分解为 3 个二元关系

教师	学生
王	S001
王	S002
刘	S002

教师	课程
王	语文
王	数学
刘	语文

学生	课程
S001	语文
S002	数学
S002	语文

第六章　数据库设计

6.1 数据库设计的步骤

6.2 需求分析

6.3 概念结构设计

6.4 逻辑结构设计

6.5 数据库物理设计

6.6 数据库实施

6.7 数据库运行维护

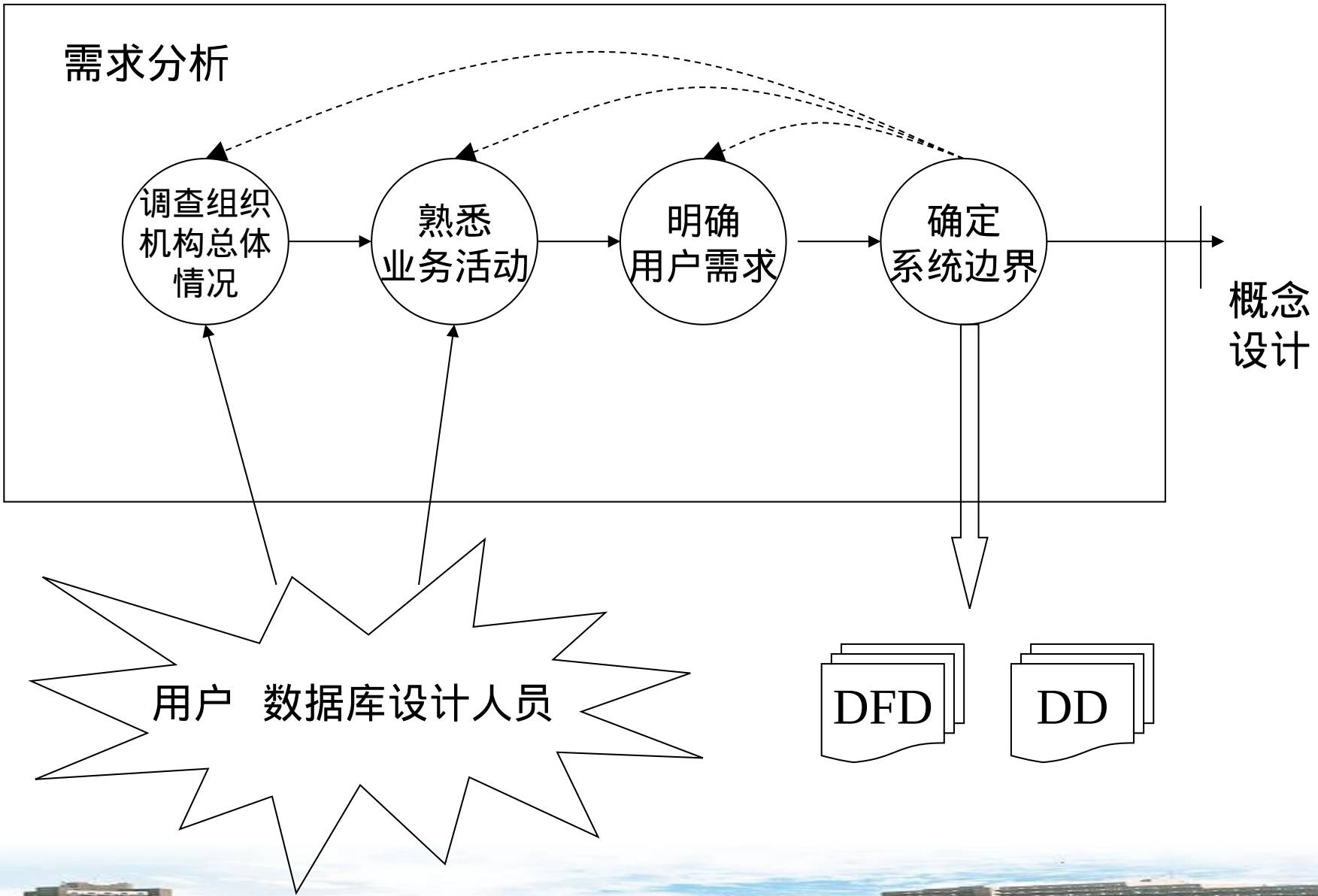
6.1 数据库设计的步骤

- 需求分析
- 概念结构设计
  - 设计局部视图
  - 集成视图
- 逻辑结构设计
  - 设计逻辑结构
  - 优化逻辑结构
- 数据库物理设计
  - 设计物理结构
  - 评价物理结构
- 数据库实施
  - 数据库系统的物理实现
  - 试验性运行
- 数据库运行维护

6.2 需求分析

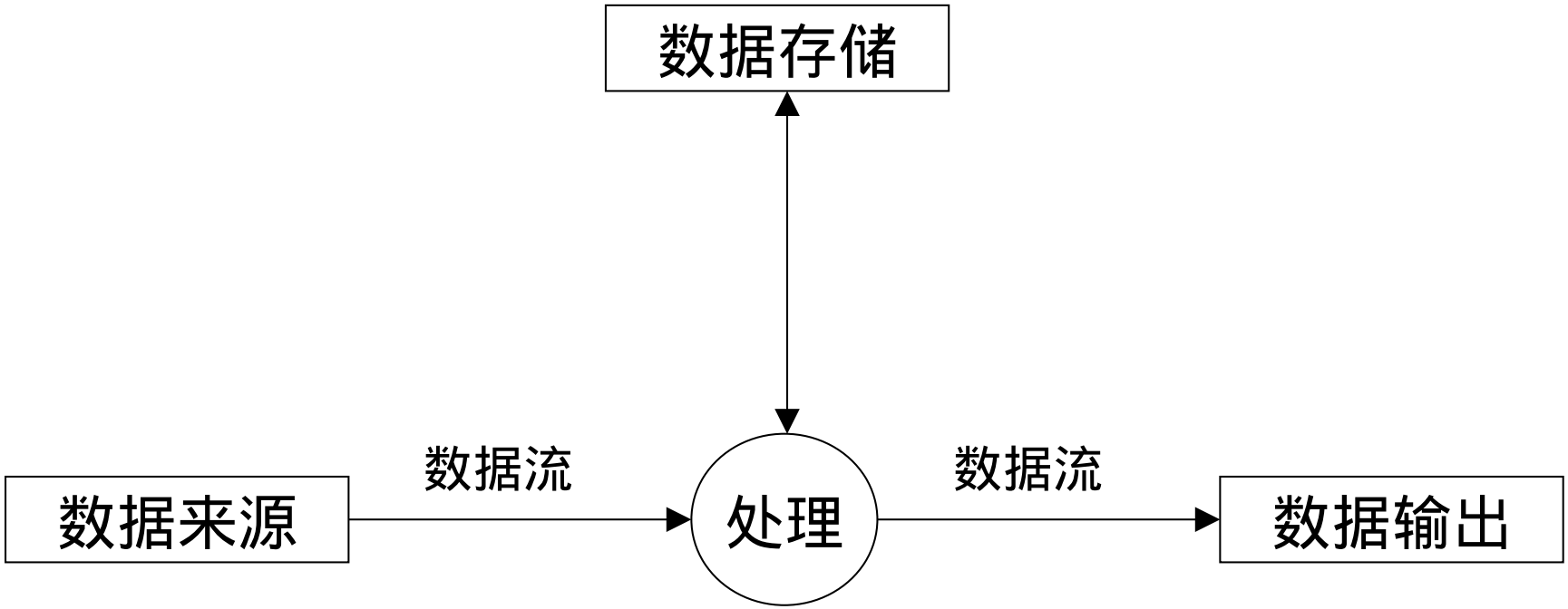
6.2.1 需求分析的任务

- 需求分析的任务
  - 通过详细调查现实世界和要处理的对象（组织、部门、企业等），充分了解原系统的工作概况，明确用户的各种需求，然后在此基础上确定新的系统功能。新系统应该考虑可扩展性。
- 需求分析的重点
  - 调查、收集与分析用户在数据库管理中的信息要求、处理要求、安全要求和完整性要求。
- 需求分析的结果
  - DD（数据字典）
  - DFD（数据流图）


需求分析
调查组织机构总体情况
熟悉业务活动
明确用户需求
确定系统边界
概念设计
用户 数据库设计人员
DFD
DD

6.2.2 需求分析的方法

- 调查与初步分析的步骤
  - 调查组织机构情况
  - 调查各部门业务活动情况
  - 在熟悉业务基础上，协调用户明确对新系统得要求
  - 对上述结果初步分析，确定新系统得边界，及人与计算机得工作边界。
- 常用的调查方法
  - 跟班作业
  - 开调查会
  - 请专业人事介绍
  - 询问
  - 设计调查表请用户填写
  - 查阅记录
- 分析用户需求的方法
  - 自顶而下，结构化分析方法 (Structured Analysis ，简称 SA)
  - 自底向上



数据流图的表示

6.2.3 数据字典

- 数据字典是详细数据收集和数据分析的结果。包涵以下内容 ：
  - 数据项：不可再分的数据单位。

    对数据项的描述包括：数据项名、含义说明、别名、数据类型、长度、取值范围、取值含义、与其他数据项的逻辑关系
  - 数据结构：反映了数据之间的组合关系。

    数据结构的描述包括：数据结构名 , 含义说明 , 组成 (数据项、数据结构)

- 数据流：数据流是数据结构在系统内传输的路径。

  数据流的描述包括：数据流名，说明，数据流来源、数据流去向、组成（数据结构）、平均流量、高峰期流量等
- 数据存储：数据存储是数据结构停留或保存的地方，也就是数据流的来源和去向之一。

  数据存储的描述：数据存储名、说明、编号、流入数据流、流出数据流、组成（数据结构）、数据量、存取方式
- 处理过程：处理过程的处理逻辑一般用判定树和判定表来描述。数据字典一般只是描述说明性信息。描述包括：处理过程名、说明、输入（数据流）、输出（输出流）、简要说明

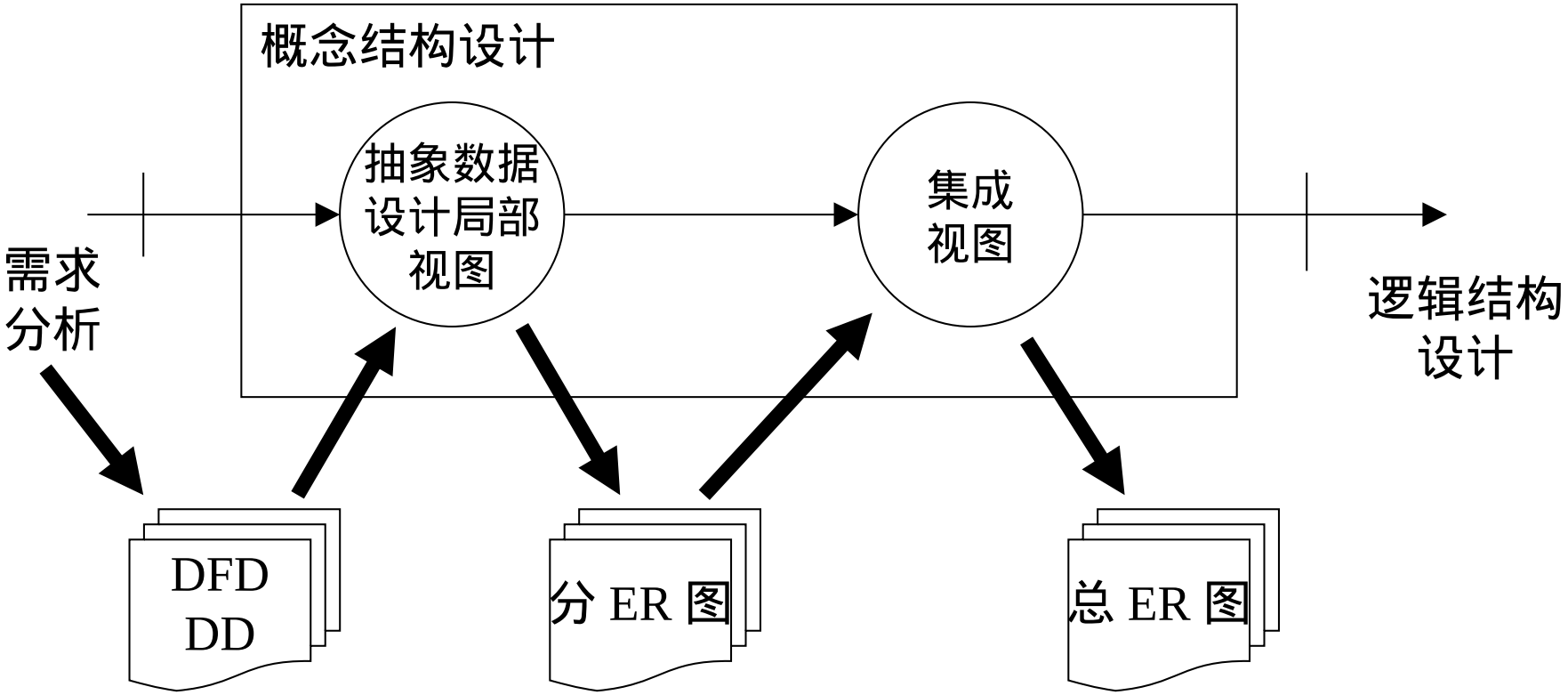
6.3 概念结构设计

6.3.1 概念结构的设计方法与步骤

- 自顶向下
  - 先定义全局概念结构，再细化
- 自底向上
  - 先定义局部应用的概念结构，再集成起来，得到全局概念结构
- 逐步扩张
  - 先定义核心概念结构，再逐步向外扩充，直至全局概念结构。
- 混合策略
  - 即使用自顶向下、自底向上相集合

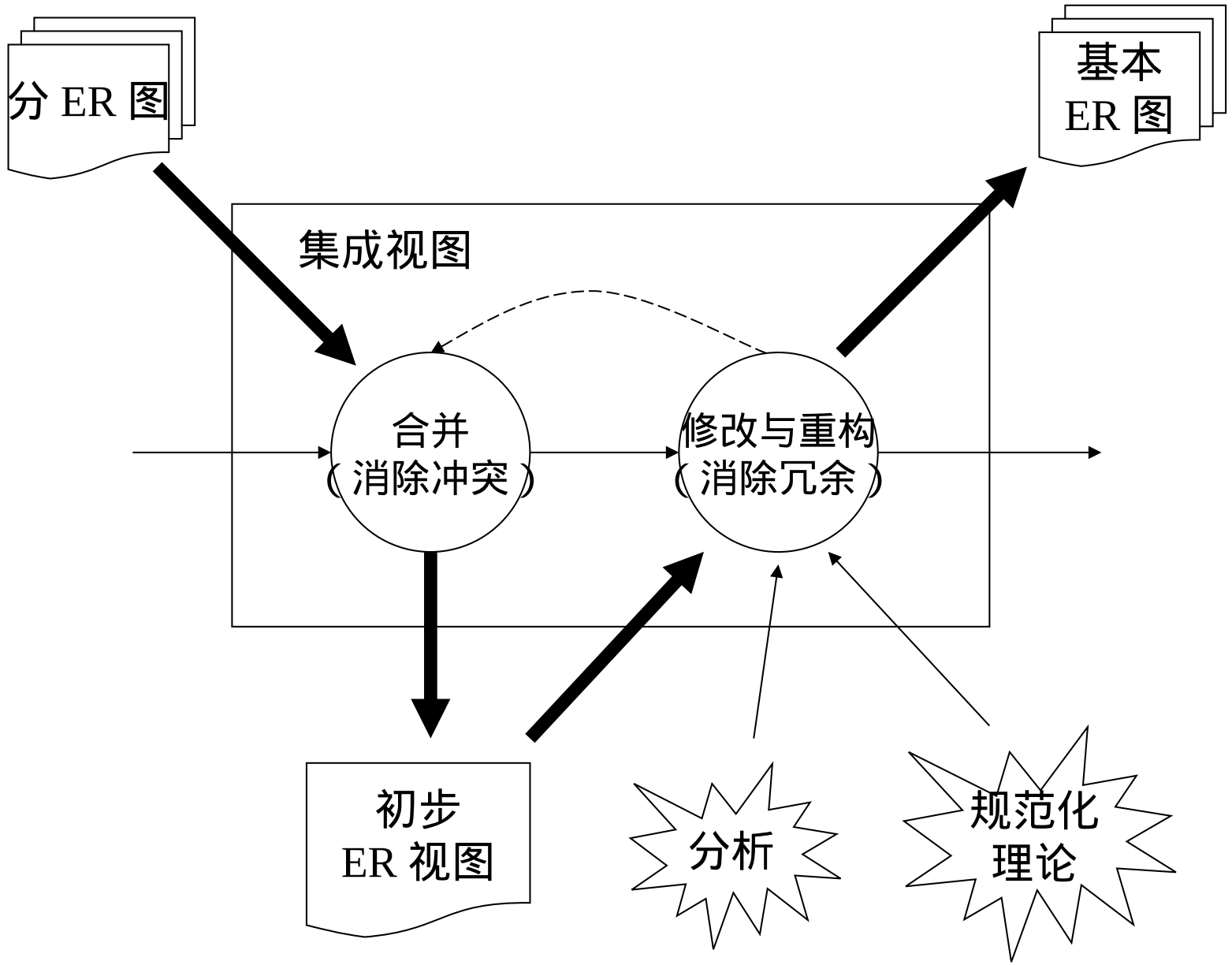
6.3.2 数据抽象与局部视图设计

- 选择局部应用
- 逐一设计分 E - R 图
  - 属性与实体很难有截然划分的界线
    - 属性不能再具有需要描述的性质
    - 属性不能与其他实体具有联系


概念结构设计
抽象数据
设计局部
视图
集成
视图
需求
分析
逻辑结构
设计
DFD
DD
分 ER 图
总 ER 图

6.3.3 视图的集成

- 合并分 E - R 图，生产初步 E - R 图，合并分 E - R 图过程中存在的冲突有 ：
  - 属性冲突：属性域冲突、属性单位冲突
  - 命名冲突：同名异义，异名同义
  - 结构冲突：同一对象抽象不同，同一实体属性不同，联系类型不同
- 修改与重构，生成基本 E - R 图。初步 E - R 图消除不必要冗余后得到基本 E - R 图。视图集成后形成整体概念结构，必须满足
  - 结构内部必须具有一致性，不能有互相矛盾的表达
  - 整体结构必须能反映原来的每一个视图结构，包括实体，属性和联系
  - 结构能满足需求分析阶段的所有需求

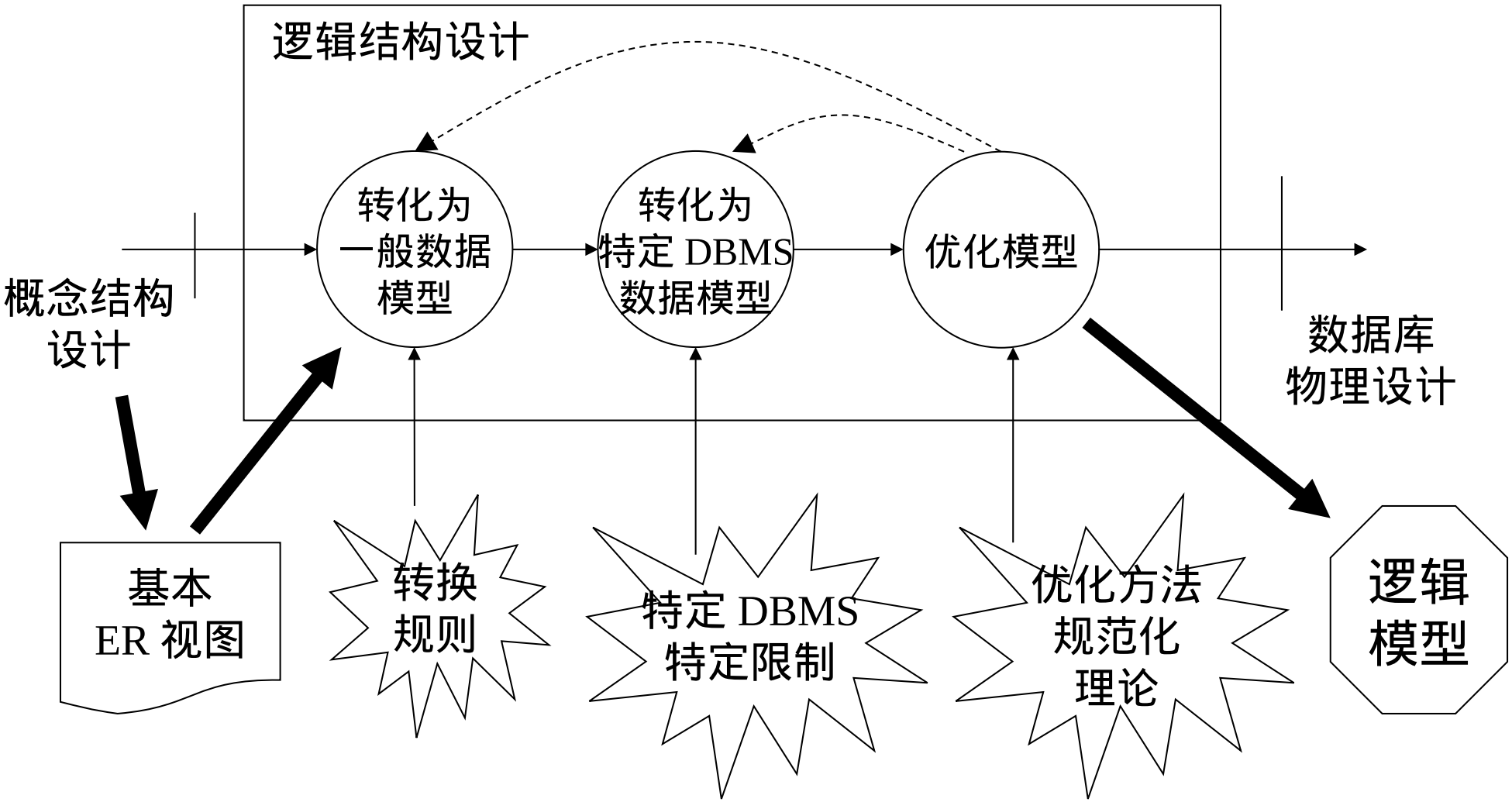

分 ER 图
基本
ER 图
集成视图
合并
（消除冲突）
修改与重构
（消除冗余）
初步
ER 视图
分析
规范化
理论

6.4 逻辑结构设计

- 逻辑结构设计的任务
  - 将概念结构转化为某一数据模型
- 逻辑结构设计的步骤
  - 将概念模型转化为一般的关系、层次、网状模型。
  - 将转化来的关系、层次和网状模型向特定的 DBMS 支持下的数据模型转换。
  - 对数据模型进行优化

6.4.1 E - R 图向数据模型转换

- 转化的原则
  - 一个实体型转化为一个关系模式。
  - 一个 m ： n 的联系转化为一个关系模式，码为各实体码组合。
  - 一个 1 ： n 的联系　　转化为一个独立的关系模式，码为 n 端实体码；也可以与 n 端关系模式合并。
  - 一个 1 ： 1 的联系转化为一个独立的关系模式，每个实体的码均是候选码；也可以与任一端关系模式合并。
  - 三个及三个以上实体间的一个多元联系转化为一个关系模式。
  - 同一实体集的实体间的联系即自联系，也可按上面的联系方式处理。
  - 具有相同码的各模式可以合并。


逻辑结构设计
概念结构
设计
转化为
一般数据
模型
转化为
特定 DBMS
数据模型
优化模型
数据库
物理设计
基本
ER 视图
转换
规则
特定 DBMS
特定限制
优化方法
规范化
理论
逻辑
模型

6.4.2 数据模型的优化

- 确定数据依赖
- 对各个关系模式间的数据依赖进行极小化处理，消除冗余的联系。
- 按照数据依赖理论对关系模式逐一进行分析考察是否存在部分函数依赖、传递函数依赖、多值依赖等，确定各关系模式分别属于第几范式。
- 按照需求分析阶段得到的各种应用对数据处理的要求，分析这样的应用环境这些关系模式是否合适，确定是否要对它们进行合并和分解。
- 对关系模式进行必要的分解和合并 。

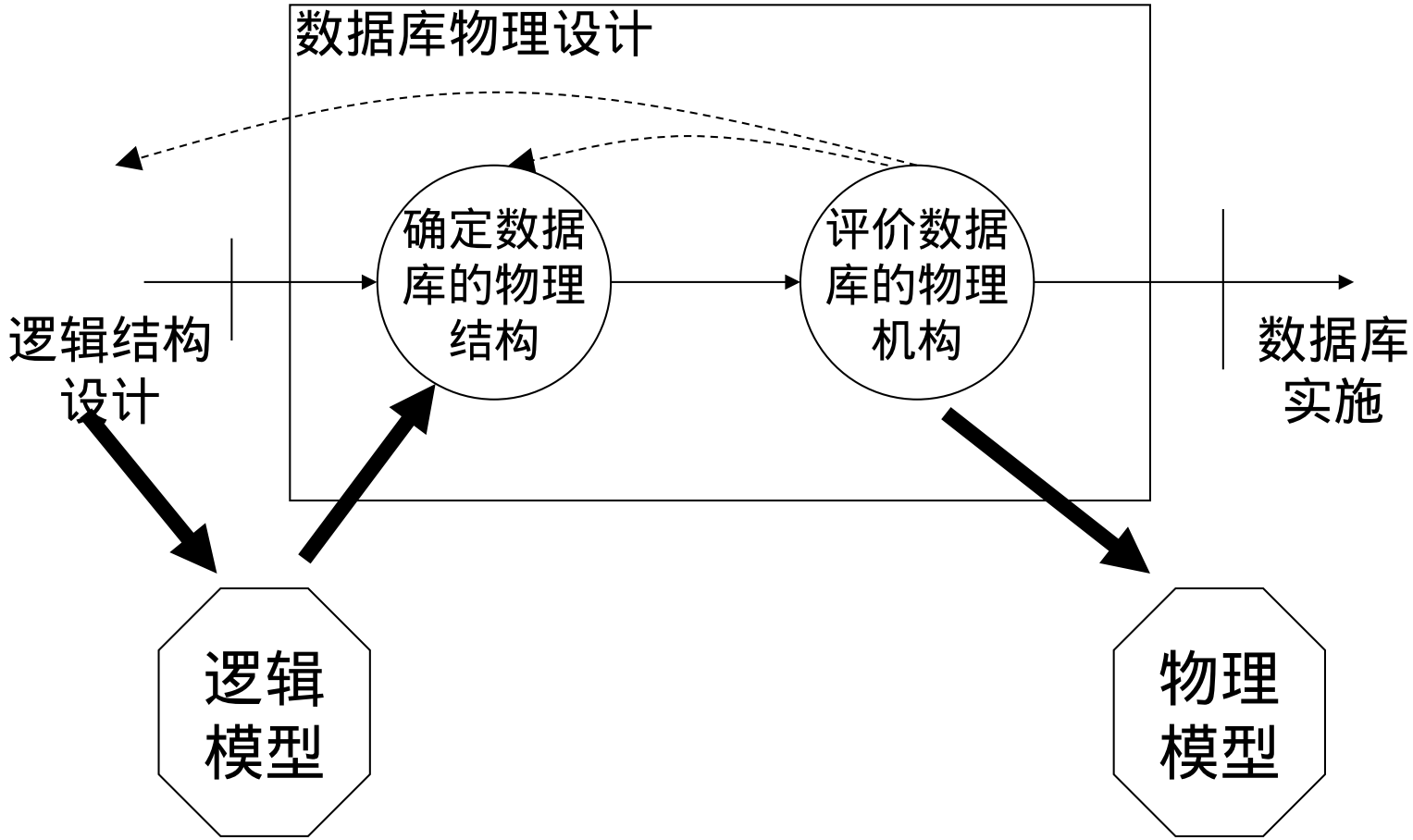
6.4.3 设计用户子模式

- 使用更符合用户习惯的别名；
- 针对不同级别的用户，定义不同的外模式，以满足系统对安全性的要求；
- 简化用户对系统的使用 ；

6.5 数据库物理设计

6.5.1 确定数据库的物理结构

- 确定数据的存储结构。综合考虑存取时间、存储空间利用率和维护代价。聚簇的使用条件：
  - 通过聚簇码进行访问是该关系的主要应用。
  - 对应与每个聚簇码的平均元组数既不太少，也不太多。
  - 聚簇码值相对稳定，以减少修改码值引起的维护开销
- 设计数据存取路径。主要是如何建立索引。
- 确定数据存放位置。主要是日志 / 数据、索引 / 数据的存放尽量分开。
- 确定系统配置。打开对象数、缓冲区大小、时间片大小、锁数目等

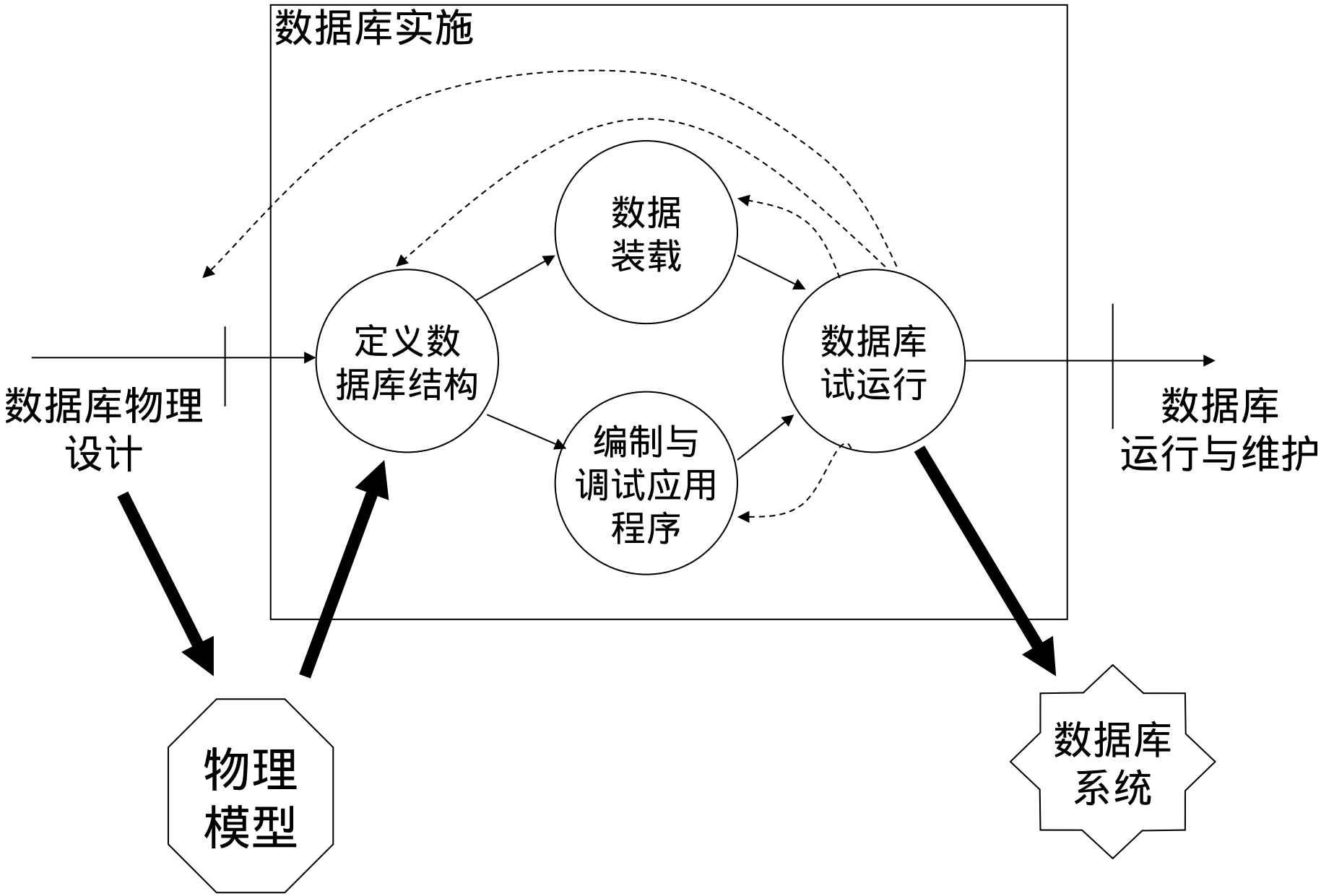

数据库物理设计
确定数据库的物理结构
评价数据库的物理机构
逻辑结构设计
数据库实施
逻辑模型
物理模型

6.5.2 评价物理结构

- 需要权衡的因素：
  - 时间效率
  - 空间效率
  - 维护代价
  - 用户需求
- 评价数据库的方法完全依赖于选用的 DMBS

6.6 数据库实施

- 数据库实施的主要工作包括
  - 用 DDL 定义数据库结构
  - 组织数据入库
  - 编制调试应用程序
  - 数据库试运行 ：包括功能测试和性能测试


数据库实施
数据
装载
定义数
据库结构
数据库
试运行
编制与
调试应用
程序
数据库物理
设计
数据库
运行与维护
物理
模型
数据库
系统

6.7 数据库运行维护

- 本阶段主要是 DBA 的工作
  1、数据库的转储和恢复
  2、数据库的安全性完整性控制
  3、数据库性能的监督、分析和改进
  4、数据库的重组织和重构造
    - 数据库重组：不会改变数据逻辑和物理结构，只是重新安排存储，回收垃圾等
    - 数据库重构：应用需求改变，要求改变逻辑设计，如表结构等，就是重构
    - 数据库系统重新设计：数据库重构的程度是十分有限的，当重构的代价太大时，就表示现有的数据库系统的生命周期已经结束，应该重新设计新的数据库系统了

第七章　关系数据库管理系统实例

7.1 关系数据库管理系统产品概述

7.2 ORACLE 数据库

7.3 Sybase 数据库

7.4 INFORMIX 数据库

7.5 DB2 数据库

7.6 INGRES 数据库

7.7* 两层及多层应用系统体系架构

7.1 关系数据库管理系统产品概述

- 对关系数据库支持的三阶段
  - 70 年代，支持数据结构和基本数据操作（选择、投影、连接等），DBASE 等。
  - 80 年代，支持国际标准 SQL ，甚至超出（ TSQL ， PL/SQL ）,Oracle 等。
  - 90 年代，加强安全性和完整性

- 运行环境发展的三阶段
  - 一般多为多用户系统的大中小型机器上运行的单机 RDMBS ，微机上的均为单用户的，因为微机 DOS 是单用户操作系统
  - 两个方向发展：一是提高移植性，使之在多种硬件和操作系统上；二是数据库联网，向分布式系统发展，支持多网络协议
  - 在网络环境下，分布式数据库和客户 / 服务器结构数据库系统的推出。追求数据库的开放性（可移植性 portability 、可连接性 connectivity 、可伸缩性 scalability ）。

- RDBMS 的系统构成 变化
  - 早期的 RDBMS 主要实现 DDL ，DML ，DCL 等基本操作以及数据存储组织、并发控制、安全性完整性检查、系统恢复、数据库的重组和重构
  - 后期的 RDBMS 以数据管理的基本功能为核心，开发外围软件系统，包括 FORMS ，REPORT ，GRAPHICS 等

- RDBMS 对应用的支持
  - 第一阶段主要是用于信息管理，对联机速度要求不高
  - 第二阶段主要针对联机事务处理，一提高事务吞吐量；二缩短联机响应时间
  - RDBMS 的改善技术主要有：
    - 性能
    - 可靠性

7.2 ORACLE 数据库

7.2.1 Oracle 公司简介

- 成立于 1977 年
- 1979 年，Oracle 第一版是世界上首批商用 RDBMS 之一。
- 1992 Oracle7，1997 年 Oracle8 、随后 Oracle 9i ，Oracle 10g

7.2.2 Oracle 产品特性

- 兼容性（ compatibility ）：兼容其他厂商的数据库兼容
- 可移植性（ portability ）：可以安装 70 多种机器，多种操作系统
- 可联结性（ connectability ）：支持多种网络协议，如 TCP/IP ､DECnet 等
- 高生产率（ high productivity ）：提供 PRO*C ､Forms 等接口与工具。
- 开放性：前述特性保障了其开放性

7.2.3 Oracle 数据库服务器产品

- 标准服务器
  - 多进程、多线程体系结构
  - 为提高性能改进核心技术：无限制行级锁、无竞争查询、多线程顺序号产生
  - 高可用性
  - SQL 的实现：符合 ISO 的 SQL 标准
- 并行服务器选件（ Oracle Parallel Server ）和并行查询（ Parallel query Option ）
- 分布式选件（ distributed Option ）
- 过程化选件（ Procedural option ）：提供用户自定义数据库对象

7.2.4 Oracle 工具

- Developer/2000
  - ORACLE Forms ：屏幕工具
  - ORACLE Reports ：报表工具
  - ORACLE Graphics ：图形工具，如直方图等。
  - ORACLE Book ：用于生产联机文档
- Designer/2000
  - BPR ：用于过程建模
  - Modellers ：用于系统设计与建模
  - Generators ：应用生产器。
- Discoverer/2000 ： OLAP 工具，应用于数据仓库
- Oracle Office ：办公自动化。
- SQL DBA ：用户动态性能监控。
- ORACLE 预编译器 Proc*C
- ORACLE 调用接口 OCI

7.2.5 Oracle 连接产品

- SQL *Net ：负责 Client 和 Server 的通讯
- Oracle 多协议转换器
- Oracle 开放式网关：利用透明网关和过程化网关，可以实现对其他数据库的直接

7.2.6 Oracle 数据仓库解决方案

- OracleOLAP
  - Oracle Express Server
  - Oracle Express Objects
  - Oracle Express Analyzer

7.2.7 Oracle 的 Internet 解决方案

- Oracle WebServer 2.0

7.3 Sybase 数据库

7.3.1 Sybase 公司简介

- 1984 年成立， INGRES 大学版本的主要设计人员之一 Dr.Robert Epstein 是 Sybase 的创始人之一
- 致力于 C/S 数据库体系结构以满足 OLTP 应用要求， 1987 年推出 SYBASE SQL Server
- Sybase 11.0 、11.5 、11.9 、12.0 、12.5 都是很优秀的版本

7.3.2 Sybase 关系数据库产品

- 数据库服务器：
  - Sybase SQL Server – Sybase MPP
  - Sybase IQ – Sybase Anywhere
- 中间件：
  - Server – Open Client
  - Open Server – OmniCONNECT
  - ObjectCONNECT for C++ Directconnect
- 工具：
  - PowerBuilder – S - Designor
  - Optima + + – NetImpact Studio
  - Internet Developer Toolkit for PowerBuilder

7.3.3 Sybase 数据库服务器 (Adaptive Server)

- SQL Server ： RDBMS ，负责高速计算、数据管理、事务管理。
  - 单进程多线程体系结构
  - 高性能
  - 数据完整性检查和控制
  - 加强安全保密功能，基于角色管理，提供审计支持分布式查询和更新

- 备份服务器（ backup server ）：附属于 SQL Server ，完成对数据的备份工作
  - 支持联机备份：备份时不影响 SQL Server 处理
  - 支持转储分解：允许使用多台外设进行转储。
  - 支持异地转储： DBA 可以管理多个远程服务器的备份和装载。
  - 支持限值转储：日志转储可以在限值事件触发下自动完成。
- SYBASE MPP ：多处理器并行服务器产品
- SYBASE IQ ：高性能决策支持和交互式数据集成产品。
- SYBASE SQL Anywhere ：基于 PC 的具有 SQL 功能的分布式数据库管理系统

7.3.4 Sybase 开发工具

- **PowerBuilder** ：基于图形界面的 C/S 前端应用开发工具。
- **PowerDesigner** ：一组紧密集成的计算机辅助软件工程（ CASE ）工具，用于数据库的分析、设计维护、建立文档和创建数据库等功能。
- **PowerJ** ：开发基于 java 应用的快速开发工具。
- **Power++**（ **Optima++** ）： **RAD C++ C/S** 和 **Internet** 面向对象的开发工具。
- **SQL Manager** ：可视化的系统和数据库管理工具。

7.3.5 Sybase 中间件

- Open Client/Open Server ：构成 Sybase 开放式 C/S 互连基础。 Open Client 用于建立有效的前端应用； Open Server 是一个服务构造工具，用于集成企业的资源与服务。
- Jaguar CTS ：是 jaguar 组件事务服务器（ component transaction server ）简称，专门为 NetOLTP 应用设计的事务服务器
- Replication Server ：复制服务器。
- OmniCONNECT ：不同数据库管理系统（都是 sybase ）之间完全透明的数据集成。
- DirectConnect ：用于同非 Sybase 数据源建立联系的访问服务器。

7.3.6 Sybase 数据仓库解决方案

- Sybase Web. Works

7.4 INFORMIX 数据库

- 1988 年 Infomix 推出第一代数据库服务器 INFORMIX - TURBO 后来被 IBM 收购。

7.5 DB2 数据库

- DB2 是 IBM 的数据库管理系统，起源于 System R 和 System R* 。

7.6 INGRES/PostgreSQL 数据库

- INGRES 公司成立于 1980 。其技术源于加州伯克利大学。 PostgreSQL 的前身。
- PostgreSQL 可以说是最富特色的自由数据库管理系统，过于学院味，因为首先它的目的是数据库研究，因此不论在稳定性，性能还是使用方便方面都比商用数据库欠缺。

7.7 MS-SQL Server 数据库

- MS-SQL Server 是 window 平台最流行的中型关系数据库 DBMS 。

7.8 MySQL 数据库

- Mysql 数据库是非常流行的开放源码 DBMS ，以性能卓越而著称，可以运行与各种 OS 平台，为了性能而较大多数 DBMS 精简。

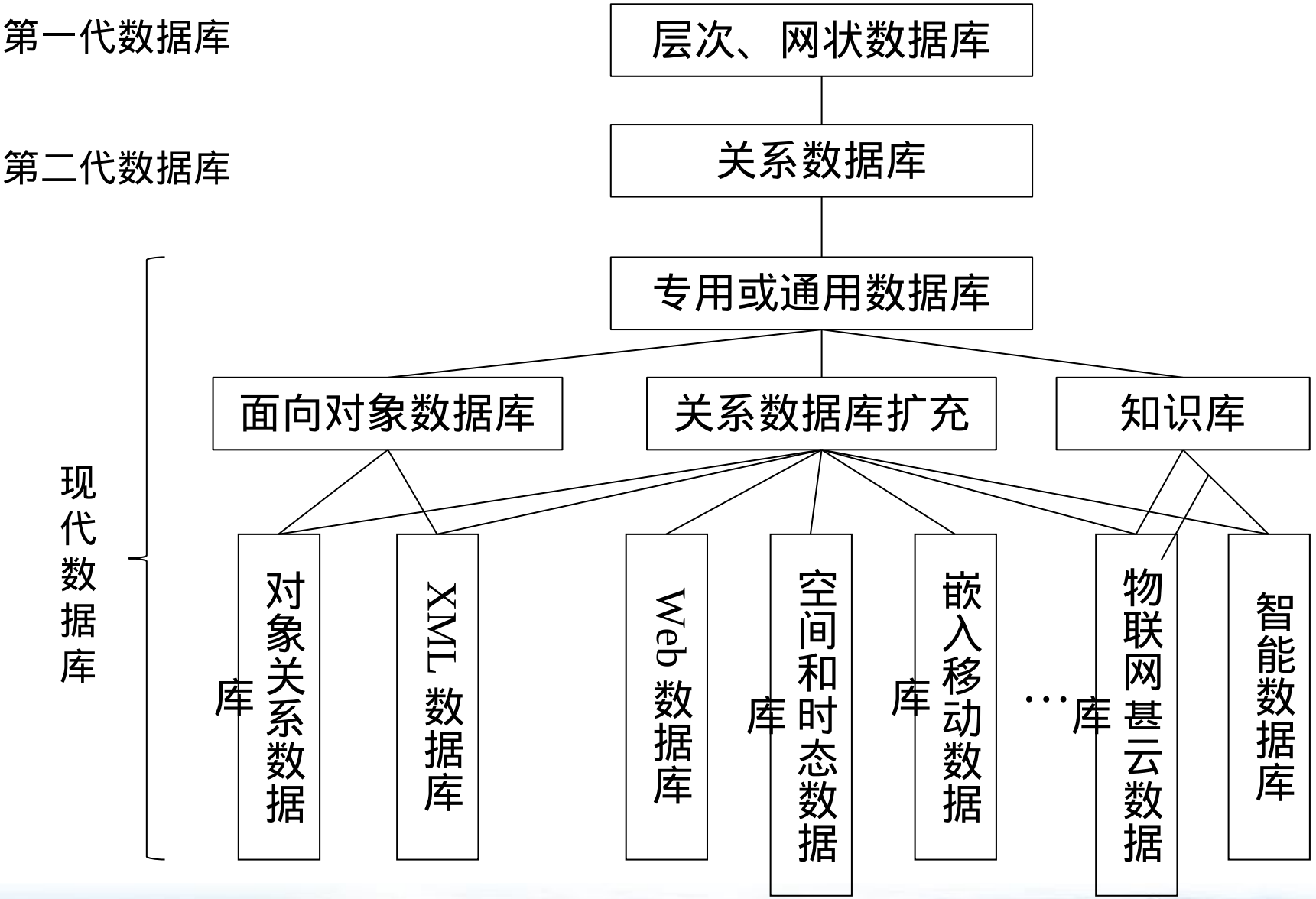
第八章 现代数据库技术及进展

8.1 数据库发展概述

8.2 数据库技术与其他技术的结合

8.3 现代数据库技术研究

8.1 数据库发展概述


第一代数据库
层次、网状数据库
第二代数据库
关系数据库
现代数据库
专用或通用数据库
面向对象数据库
关系数据库扩充
知识库
对象关系数据库
XML 数据库
Web 数据库
空间和时态数据库
嵌入移动数据库
…库
物联网甚云数据库
智能数据库

8.2 数据库技术与其他技术的结合

8.2.1 分布式数据库

- 分布式数据库特点：
  - 数据的物理分布性
  - 数据的逻辑整体性
  - 数据的分布独立性（透明性）：除了物理独立性、逻辑独立性，用户不必关心数据的分布细节。
  - 场地自治与协调：各结点能执行局部应用请求，也能通过网络处理全局请求。
  - 数据的冗余及冗余透明性：适当冗余提高系统效率和可靠性，冗余对用户透明

- 分布式数据库优点：
  - 分布式控制：减少通信开销
  - 数据共享：全局和局部
  - 可靠性和可用性加强：可以根据冗余数据恢复某一场地数据
  - 性能改善：数据分片，减少资源竞争
  - 可扩充性好：因分布独立性，不会影响用户程序。
- 分布式数据库缺点：
  - 复杂：主要是自治与协调。
  - 开销大：硬件、通信、冗余开销、安全性、完整性和并发控制开销

- 分布式数据库体系结构：（若干局部数据模式 + 一个全局数据模式）
  - 全局数据模式
    - 全局外模式
    - 全局概念模式
    - 分片模式：水平、垂直、混合分片；完全性、可重构性、不相交性。
    - 分布模式
  - 局部数据模式
    - 局部外模式□　局部概念模式□ 局部内模式
  - 四级映象
    - 全局外模式与全局概念模式之间对应关系。
    - 全局概念模式（关系）与全局分片模式（片断）之间对应关系
    - 全局分片模式（片断）与全局分布模式（网络结点）之间的对应关系。
    - 全局分布模式与局部概念模式之间的映射

8.2.2 并行数据库

- 并行计算机体系类型：
  - 紧耦合全对称多处理器（ SMP ）所有 CPU 共享内存与磁盘（ share memory ）。
  - 松耦合群集机系统，所有 CPU 共享磁盘（ share disk ）
  - 大规模并行处理（ MPP ），所有 CPU 有自己内存和磁盘无共享资源（ share nothing ）

- 相应并行数据库的体系结构：
  - 共享内存（ share memory ）（ SMP ）
  - 共享磁盘（ share disk ）
  - 无共享资源（ share nothing ）
- 并行数据库的粒度：
  - 不同事务间并行
  - 同一事务不同查询间并行
  - 同一查询不同操作间并行
  - 同一操作内并行

8.2.3 多媒体数据库

- 主要特征：
  - 能够表示多种媒体数据。
  - 能够协调处理各种媒体数据。
  - 应提供比传统数据管理系统更强的适合非格式化数据查询的搜索功能。
  - 应提供特种事务处理与版本管理能力。

8.2.4 主动数据库

- 目标是提供紧急情况及时反应能力，同时提高数据库模块化程度。
- 通常采用方法是在传统数据库中嵌入 ECA（事件 - 条件 - 动作）规则。
- 要解决的问题：
  - 数据模型和知识模型
  - 执行模型
  - 条件检测
  - 事务调度
  - 体系结构
  - 系统效率

8.2.5 对象 - 关系数据库

- 对象 - 关系数据库兼有关系数据库和面向对象数据库的两方面特征。即在关系数据库基础上提供一下功能：
  - 允许扩充基本数据模型
  - SQL 中支持复杂对象
  - 支持子类对超类的特性继承
  - 提供通用规则系统

8.3 现代数据库技术研究

1 ） web 数据库

在互联网中以 web 查询接口方式访问的数据库资源。

2 ） XML 数据库

关系数据库和 XML 结合的一个重要方向

3 ）数据仓库

从传统的事务处理到决策、统计型发展的数据库技术。

数据库处理分两类：操作型（ OLTP ）和分析型（ OLAP ）

4 ）主动数据库

结合人工智能和面向对象技术而产生的新技术。

5 ）嵌入式移动数据库

微型化方向发展，应用于嵌入式系统和移动通信领域。

6 ）空间和时态数据库

地理信息系统在计算机区里存储介质上存储的应用相关的地理空间数据的总和。体现了随时间的变化关系。

7 ）内存数据库

内容价格大幅度降低，对时间要求的苛刻，使得数据库常驻内存。

8 ）物联网数据库

互联网基础上的延伸和扩展的网络，物品与物品的信息交换和通信要求“数据海”—物联网数据库

9) 云数据库

广义云计算的一种高级应用

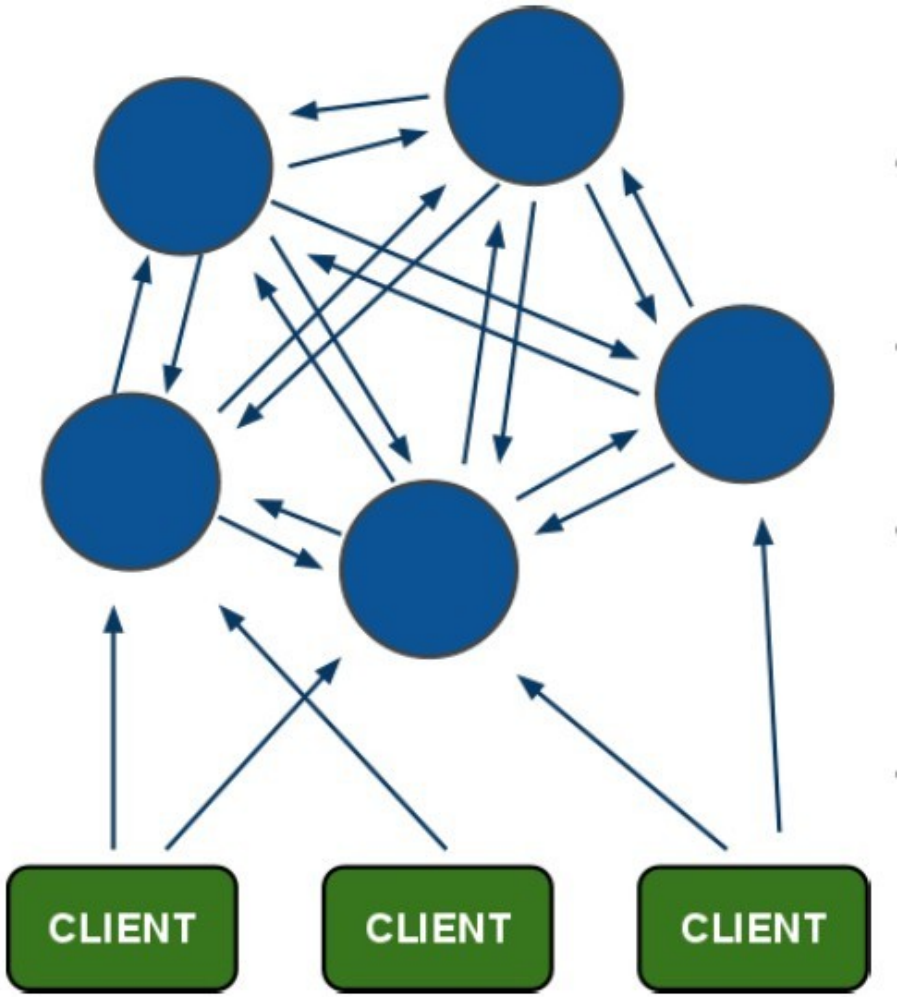
10 ）知识与智能数据库

知识工程中结构化、易操作全面有组织的知识集群。

Redis

- 一个开源的 KV 存储解决方案，用于构建高性能，可扩展的 Web 应用程序。
  - Redis 将其数据完全保存在内存中，仅使用磁盘进行持久化。
  - 与其它键值数据存储相比， Redis 有一组相对丰富的数据类型
  - Redis 可以将数据复制到任意数量的从机中

Redis cluster 结构



Redis 的优点

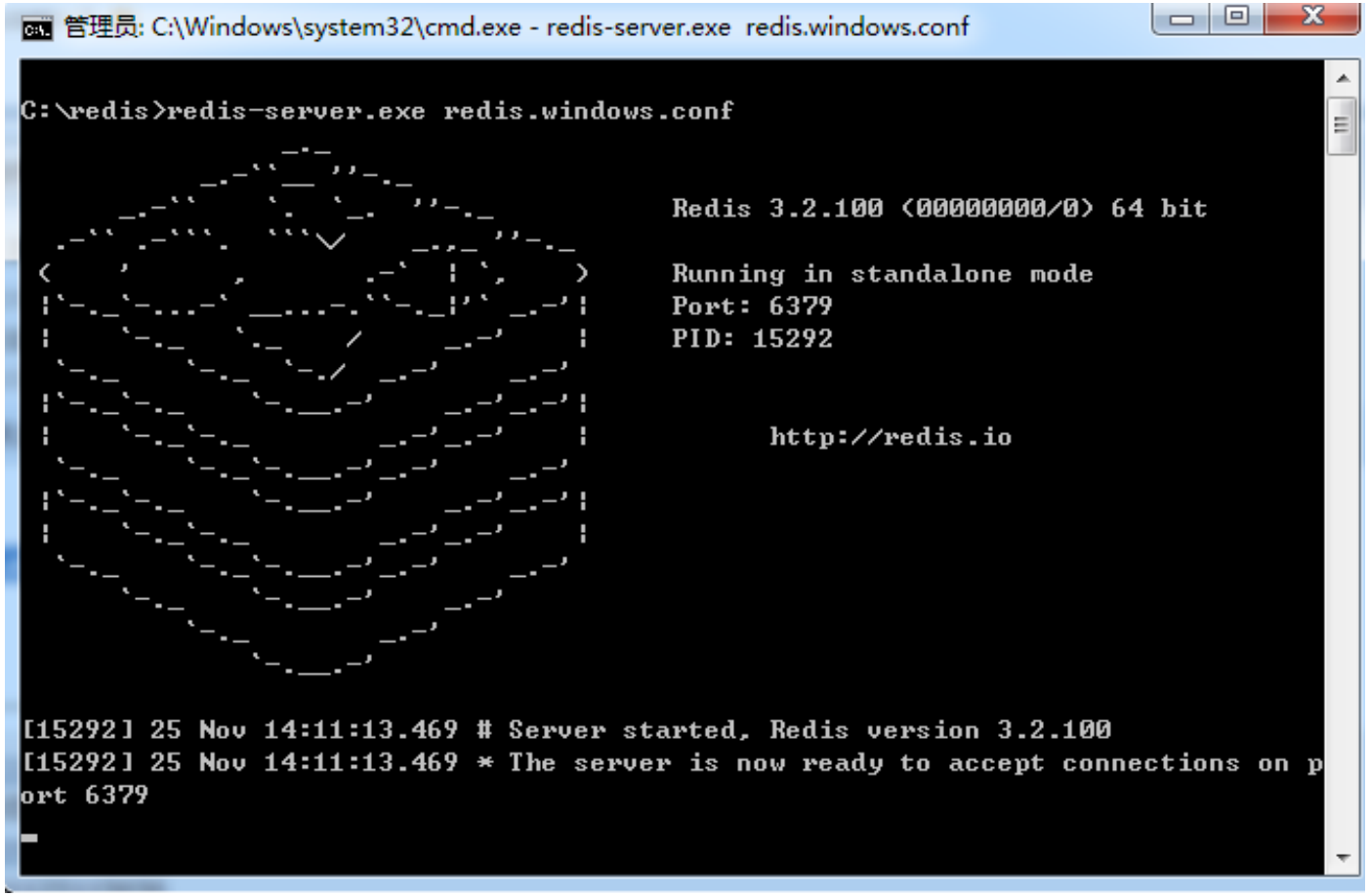
- 异常快
  - 每秒约 110000 次 SET 操作， 81000 次 GET 操作
- 丰富的数据类型
  - string ， hash ， list ， set 及 zset
- 操作具有原子性
  - 确保如果两个客户端并发访问， Redis 服务器能接收更新的值。
- 多实用工具
  - Redis 是一个多实用工具，可用于多种用例，如：缓存，消息队列，应用程序中的任何短期数据（ web 应用程序中的会话，网页命中计数等）

Redis 安装

- **Window 下安装**
  - 下载地址：https://github.com/MSOpenTech/redis/releases 。
  - 打开一个 **cmd** 窗口 使用 cd 命令切换目录到 **C:\redis** 运行

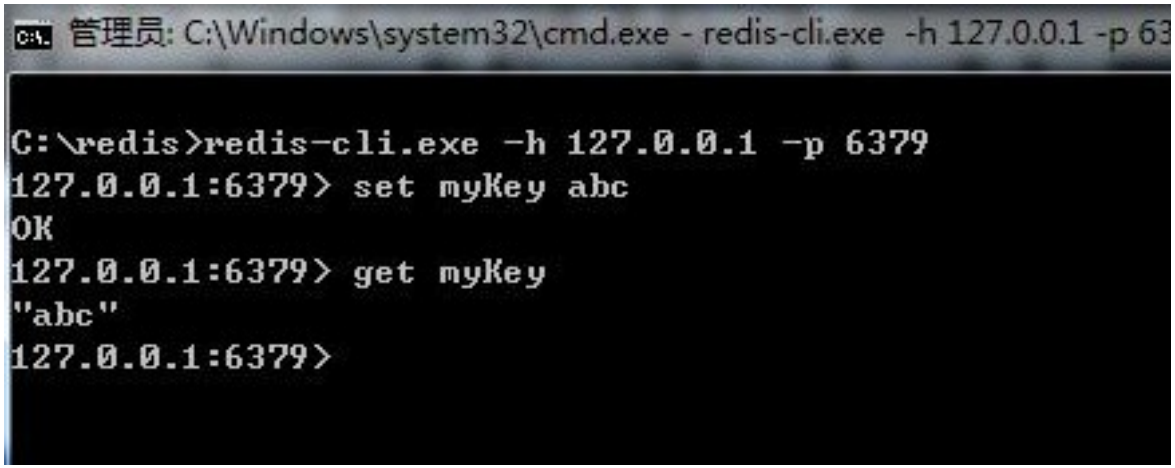
    redis-server.exe redis.windows.conf 。

Redis 服务运行界面



利用 redis-cli 命令测试

- 这时候另启一个 cmd 窗口。
- 切换到 redis 目录下运行
  - **redis-cli.exe -h 127.0.0.1 -p 6379** 。
  - 设置键值对 **set myKey abc**
  - 取出键值对 **get myKey**



Linux 安装

- 下载地址： http://redis.io/download

```
$ wget http://download.redis.io/releases/redis-2.8.17.tar.gz
$ tar xzf redis-2.8.17.tar.gz
$ cd redis-2.8.17
$ make
```

```
$ cd src
$ ./redis-server redis.conf
```

```
$ cd src
$ ./redis-cli
redis> set foo bar
OK
redis> get foo
"bar"
```

Redis PHP 字符串实例

- PHP redis
  驱动下载地址为：**https://github.com/phpredis/phpredis/releases**

```php
<?php
   //连接本地的 Redis 服务
   $redis = new Redis();
   $redis->connect('127.0.0.1', 6379);
   echo "Connection to server sucessfully";
   //设置 redis 字符串数据
   $redis->set("tutorial-name", "Redis tutorial");
   // 获取存储的数据并输出
   echo "Stored string in redis:: " . $redis->get("tutorial-name");
?>
```

Java 使用 Redis

- 需要下载驱动包 下载 **jedis.jar**

```
import redis.clients.jedis.Jedis;

public class RedisStringJava {
    public static void main(String[] args) {
        //连接本地的 Redis 服务
        Jedis jedis = new Jedis("localhost");
        System.out.println("连接成功");
        //设置 redis 字符串数据
        jedis.set("runoobkey", "www.runoob.com");
        // 获取存储的数据并输出
        System.out.println("redis 存储的字符串为: "+ jedis.get("runoobkey"));
    }
}
```

OceanBase 分布式数据库系统

- 是一个支持海量数据的高性能分布式数据库系统，实现了数千亿条记录、数百 TB 数据上的跨行跨表事务，由 alibaba 核心系统研发部等部门共同完成
- 设计和实现的时候暂时摒弃了不紧急的 DBMS 的功能，例如临时表，视图
- 主要解决数据更新一致性、高性能的跨表读事务、范围查询、 join 、数据全量及增量 dump 、批量数据导入

现有数据库的弊端

- 数据和负载增加后添加机器的操作比较复杂，往往需要人工介入；
- 有些范围查询需要访问几乎所有的分区，例如，按照 user_id 分区，查询收藏了一个商品的所有用户需要访问所有的分区；
- 目前广泛使用的关系数据库存储引擎都是针对机械硬盘的特点设计的，不能够完全发挥新硬件（ SSD ）的能力。
- Google Bigtable 系统虽然解决了可扩展性问题，但往往无法支持事务

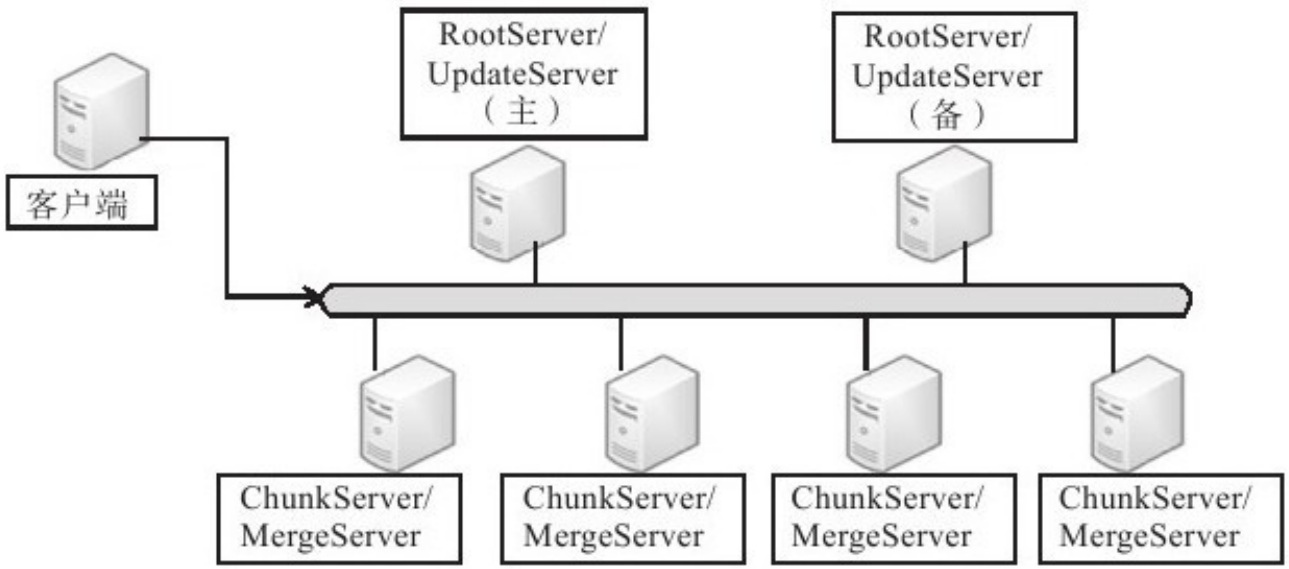
设计思路

- 在线业务的数据量十分庞大，例如几十亿条、上百亿条甚至更多记录，但最近一段时间（例如一天）的修改量往往并不多，通常不超过几千万条到几亿条，因此， OceanBase 采用单台更新服务器来记录最近一段时间的修改增量，而以前的数据保持不变，以前的数据称为基线数据。

- 基线数据以类似分布式文件系统的方式存储于多台基线数据服务器中，每次查询都需要把基线数据和增量数据融合后返回给客户端。这样，写事务都集中在单台更新服务器上，避免了复杂的分布式事务，高效地实现了跨行跨表事务

- 更新服务器上的修改增量能够定期分发到多台基线数据服务器中，避免成为瓶颈，实现了良好的扩展性。
- 更新服务器的硬件配置相对较好，如内存较大，网卡及 CPU 较好
- 最近一段时间的更新操作往往总是能够存放在内存中，在软件层面也针对这种场景做了大量的优化。

体系架构


RootServer/
UpdateServer
（主）
RootServer/
UpdateServer
（备）
客户端
ChunkServer/
MergeServer
ChunkServer/
MergeServer
ChunkServer/
MergeServer
ChunkServer/
MergeServer

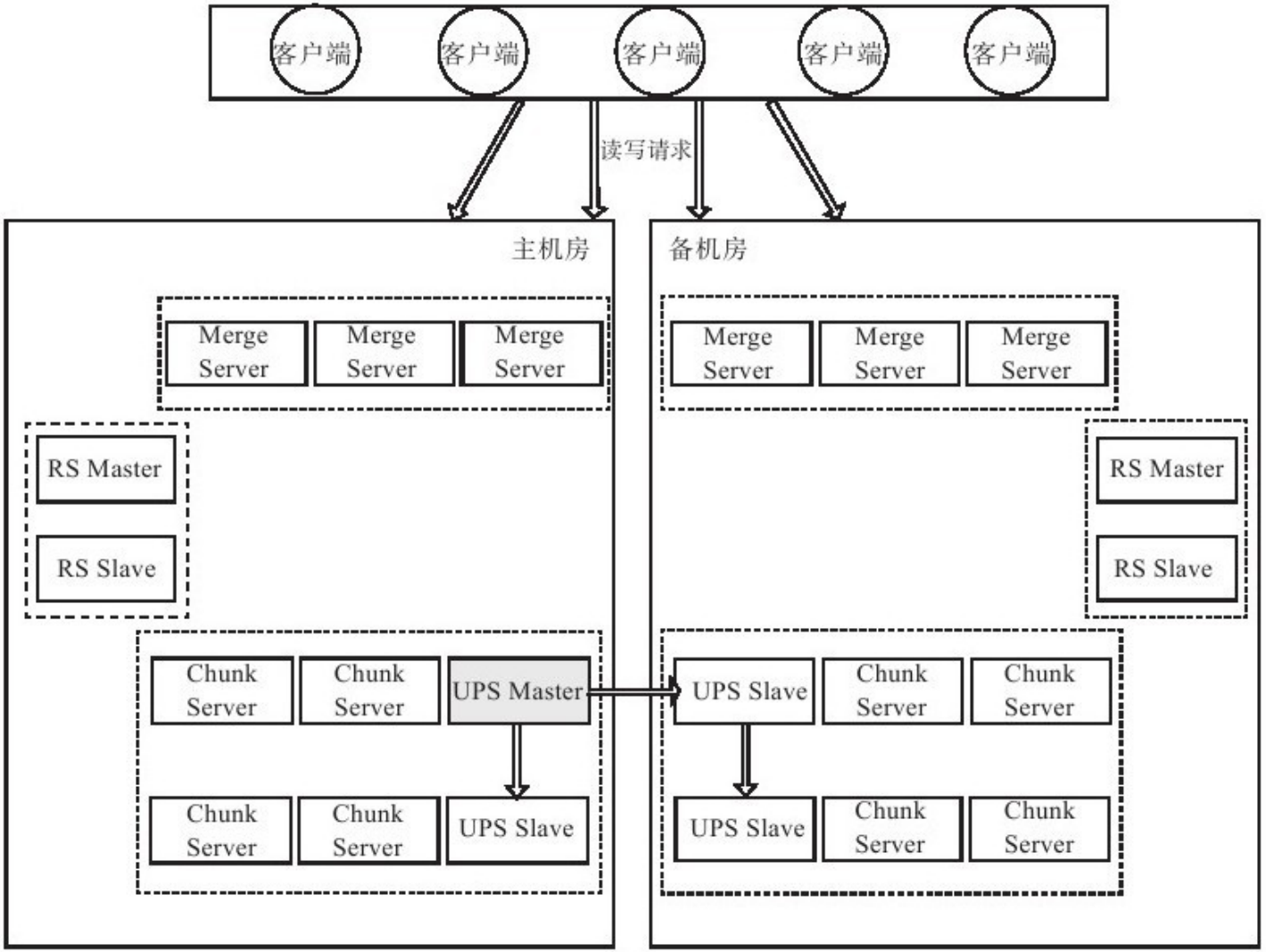
Oceanbase 的组成

- 客户端
  - 用户使用 OceanBase 的方式和 MySQL 数据库完全相同，支持 JDBC 、 C 客户端访问，等等。基于 MySQL 数据库开发的应用程序、工具能够直接迁移到 OceanBase
- RootServer
  - 管理集群中的所有服务器，子表（ tablet ）数据分布以及副本管理。 RootServer 一般为一主一备，主备之间数据强同步

- UpdateServer
  - 存储 OceanBase 系统的增量更新数据。 UpdateServer 一般为一主一备，主备之间可以配置不同的同步模式。部署时， UpdateServer 进程和 RootServer 进程往往共用物理服务器
- ChunkServer
  - 存储 OceanBase 系统的基线数据。基线数据一般存储两份或者三份，可配置

- MergeServer
  - 接收并解析用户的 SQL 请求，经过词法分析、语法分析、查询优化等一系列操作后转发给相应的 ChunkServer 或者 UpdateServer 。如果请求的数据分布在多台 ChunkServer 上， MergeServer 还需要对多台 ChunkServer 返回的结果进行合并。客户端和 MergeServer 之间采用原生的 MySQL 通信协议，MySQL 客户端可以直接访问 MergeServer

多机房体系机构


客户端
客户端
客户端
客户端
客户端
读写请求
主机房
备机房
Merge Server
Merge Server
Merge Server
Merge Server
Merge Server
Merge Server
RS Master
RS Slave
RS Master
RS Slave
Chunk Server
Chunk Server
UPS Master
UPS Slave
Chunk Server
Chunk Server
Chunk Server
Chunk Server
UPS Slave
Chunk Server
Chunk Server
UPS Slave
Chunk Server
Chunk Server

相关链接

- http://code.taobao.org/svn/OceanBase/
- https://github.com/alibaba/oceanbase/
- https://www.cnblogs.com/LiJianBlog/p/4779934.html

谢　谢！