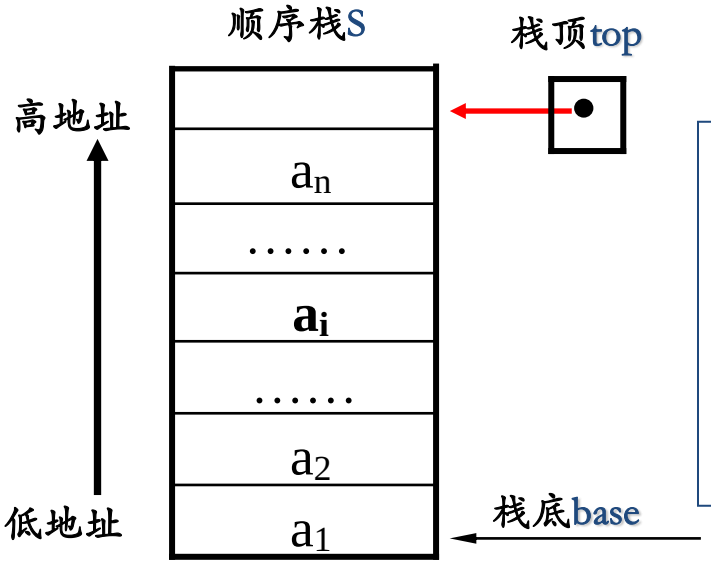
第三章 栈和队列

第三章 栈和队列

- 3.1 栈
- 3.2 栈的实现
- 3.3 栈的应用
- 3.4 队列
- 3.5 队列的实现
- 3.6 队列的应用

回顾

栈是一种后进先出(Last In First Out)的线性表，简称为LIFO表。



栈不存在的条件： base=NULL;

栈为空 的条件： base=top;

栈满的条件 ： top-base=stacksize;

一般栈顶不存放元素。

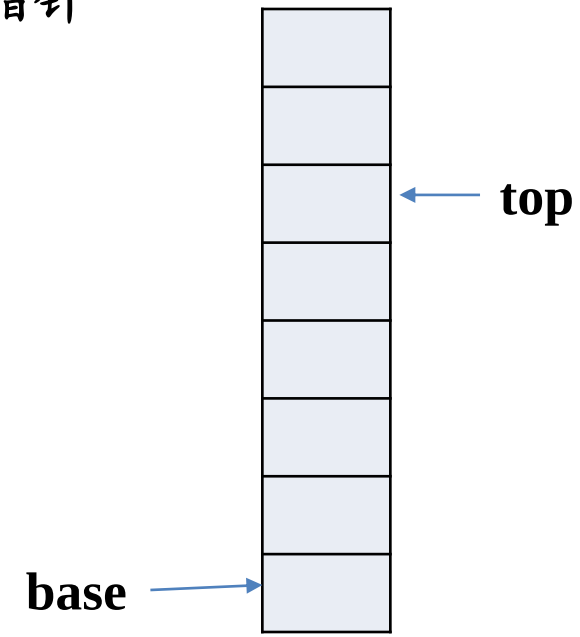
入栈Push：堆栈指针top “先压后加” ： $S[top++]=a_{n+1}$

出栈Pop：堆栈指针top “先减后弹” ： e=S[--top]

回顾

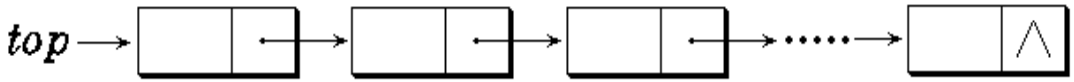
顺序栈

```
#define    STACK-INIT-SIZE     100  //存储空间初始分配量
#define    STACKINCREMENT    10   //存储空间分配增量
typedef    struct{
      SElemType     *base;    //栈的基址即栈底指针
      SElemType     * top;     //栈顶指针
      int              stacksize;  //当前分配的空间
}SqStack;
SqStack  s;
```



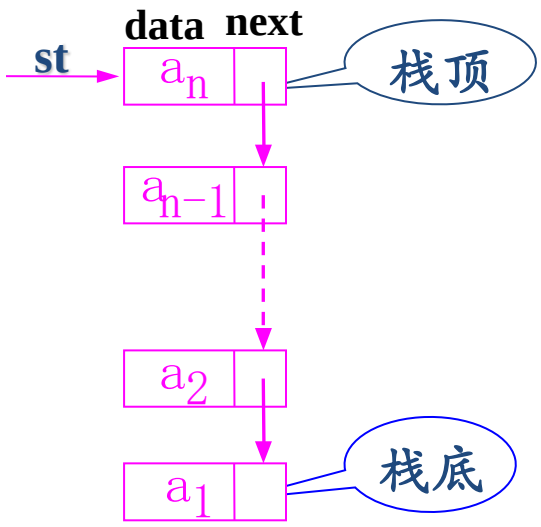
回顾

- 链栈示意图　　p47 图3.3



- 栈空条件：　s==NULL
- 栈满条件：　无　/ 无Free Memory可申请

```
typedef struct SNode{
  SElemType  data;
  struct SNode  *next;
}SNode, *LinkStack;
LinkStack s;
```



进栈　相当于链表在 top 结点前插入

出栈　相当于链表在将top 结点删除

回顾

入栈出栈次序判定规则

若输入序列是 …,P_i…P_j…P_k…(i<j<k,输入序号)，一定不存在这样的输出序列 …,P_k…P_i…P_j…

由元素1，2，3，4，5，6，7，8依次入栈、出栈，要求每次出栈之前至少有两次连续入栈操作，出栈时可以出栈一个元素，也可以出栈多个元素直至栈空，则数据的出栈序列可能是（ ）

A. 3，4，2，5，7，6，1，8

B. 2，4，3，1，8，7，6，5

C. 5，7，6，4，8，3，2，1

D. 4，3，5，2，1，6，8，7

第三章 栈和队列

（1）表达式求值

任何一个表达式都是由操作数（operand）、运算符（operator）和界限符组成的，我们称它们为单词。

把运算符和界限符统称为算符。

简单表达式的求值问题，这种表达式只含加、减、乘、除、乘方、括号等运算。

（1）表达式求值

对于两个相继出现的算符Q1和Q2，其优先关系：

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

（1）表达式求值

算法的基本思想

界限符 ‘#’优先级别最低

设置两个工作栈：运算符栈OPTR　　操作数栈OPND

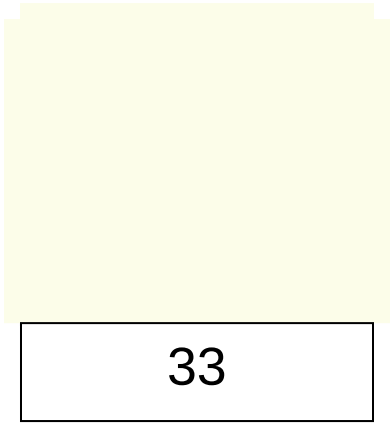
1）首先置操作数栈为空栈，表达式起始符‘#’为运算符栈的栈底元素。

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 栈的栈顶运算符**比较优先权后作相应操作，直至整个表达式求值完毕。

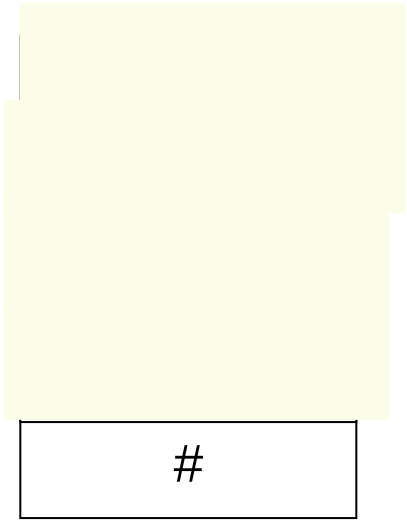
（1）表达式求值

输入：

#



操作数栈



运算符栈

第三章 栈和队列

（1）表达式求值

C是操作符吗?

```
Status   EvaluateExpression( OperandType  &result)  {
 InitStack(OPND); InitStack(OPTR);Push(OPTR ,'#');c=getchar();
 while((c!='#')&&(GetTop(OPTR)!='#'))
  {  if (!In(c,OP))  { Push(OPND,c);   c=getchar();}
      else   switch(precede(GetTop(OPTR) , c))
             { case '<' :  Push(OPTR , c); c=getchar();break;
               case '=':   Pop(OPTR);c=getchar();break;
               case '>' :  Pop(OPTR ,theta); Pop(OPND,b); Pop(OPND, a);
                           Push(OPND, Operate(a, theta,b)) ;break; ;
  } //switch  }//while
   result=GetTop(OPND);
}//EvaluateExpression
```

运算符与栈顶比较并查3.1表

Operate=a θ b

（2）栈与递归

递归的定义

如果一个对象部分的由自己组成，或者是按它自己定义的，则称为**递归**的。

一个递归定义必须一步比一步简单，最后是有终结的，决不能无限循环下去，终结条件就是递归定义的出口，简称为**递归出口**。

（2）栈与递归

递归函数的特点：在函数内部可以直接或间接地调用函数自身。如阶乘函数：

$$n!=\begin{cases}1 & ,\quad n=0\\ n*(n-1)! & ,\quad n>0\end{cases}$$

第三章 栈和队列

（2）栈与递归

```
int fact (int n)
{     if (n = = 0)
          return 1;
      else
          return(n*fact(n-1));
 };                          r2
main( )
 {  int n;
     scanf(“%d\n”,&n);
     printf(“%8d%8d\n”,n,fact(n));
 }                                  r1
```

（2）栈与递归

递归函数到非递归函数的转换

调用前：

（1）为被调用函数的局部变量分配存储区；
（活动记录， 数据区）

（2）将所有的实参、返回地址传递给被调用函数；
实参和形参的结合，传值。

（3）将控制转移到被调用函数入口。

调用后返回：

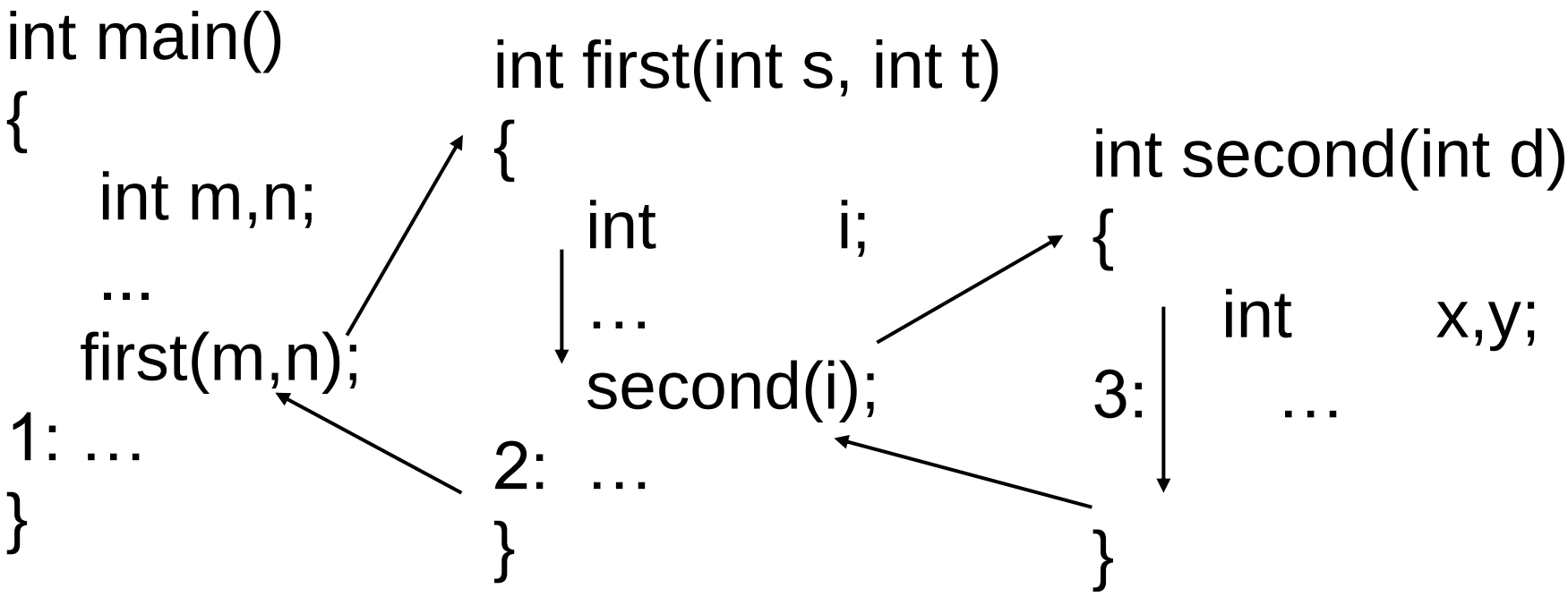
（1）保存被调用函数的计算结果；

（2）释放被调用函数的存储区；

（3）依照被调用函数保存的返回地址将控制转移到调用函数。

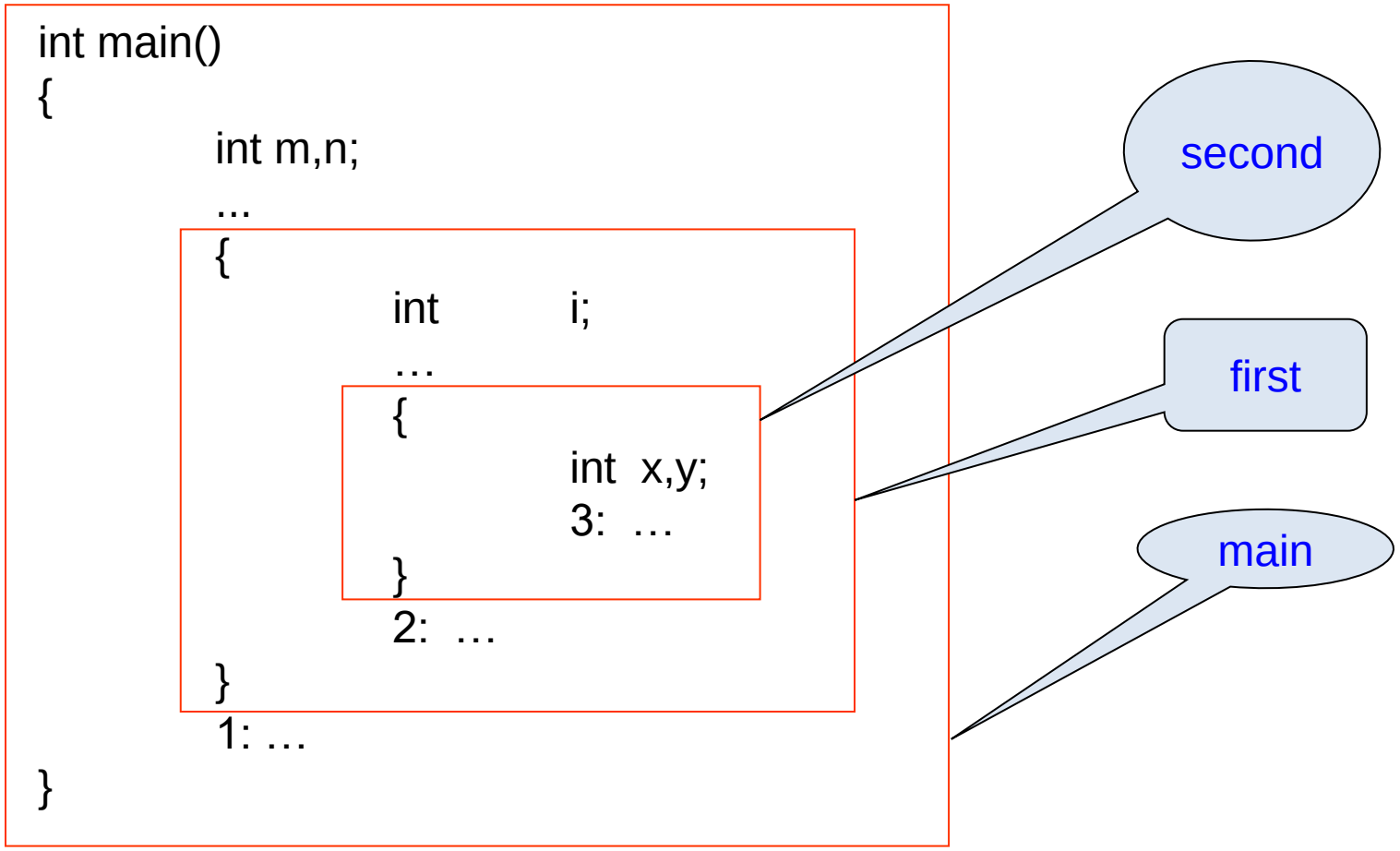
（2）栈与递归

函数嵌套调用时，按照“后调用先返回”的原则进行。



（2）栈与递归

函数嵌套结构



（2）栈与递归

```
int  nfact(int n);
{    int res;
     SqStack st;
     InitStack(st);
     while (n>0)
     {  Push(st,n);
        n=n-1;       }
     res=1;
}
```

```
while (!StackEmpty(st)
{   res=res*GetTop(st);
    Pop(st);       }
    free(st);  return(res);
```

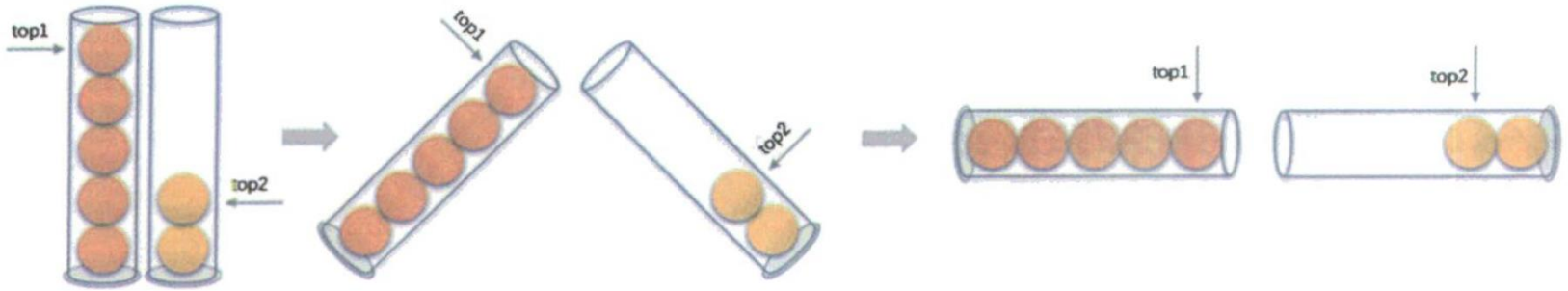
阶乘递归算法：

```
int fact (int n)
{    if (n = = 0)
          return 1;
          else
          return(n*fact(n-1));
};
```

第三章 栈和队列

口 共享栈

两个栈共享同一片存储空间，这片存储空间不单独属于任何一个栈，某个栈需要的多一点，它就可能得到更多的存储空间；
两个栈的栈底在这片存储空间的两端，当元素入栈时，两个栈的栈顶指针相向而行。



第三章 栈和队列

- 3.1 栈
- 3.2 栈的实现
- 3.3 栈的应用
- 3.4 队列
- 3.5 队列的实现
- 3.6 队列的应用

3.4 队列

3.4 队列

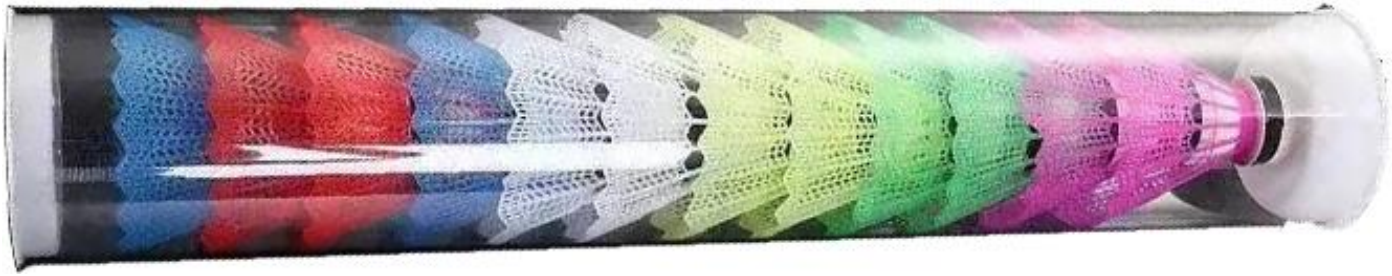
队列定义　只能在表的一端进行插入运算，在表的另一端进行删除运算的线性表。

尾部插入

首部删除

逻辑结构　与线性表相同，仍为一对一关系。

存储结构　顺序队或链队，循环顺序队更常见。

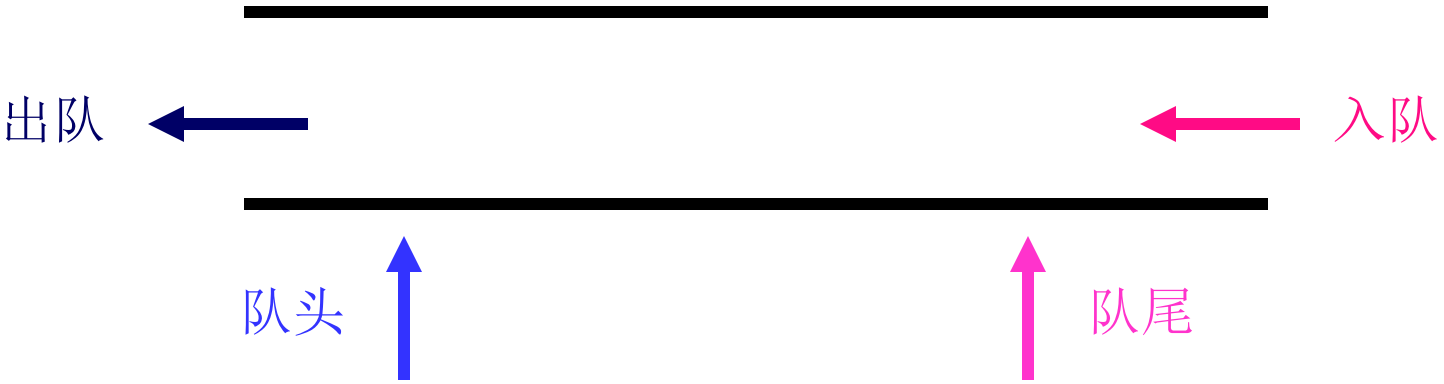


3.4 队列

运算规则　只能在队首和队尾运算，且访问结点时依照先进先出（FIFO）的原则。

实现方式　关键是掌握入队和出队操作，具体实现依顺序队或链队的不同而不同。

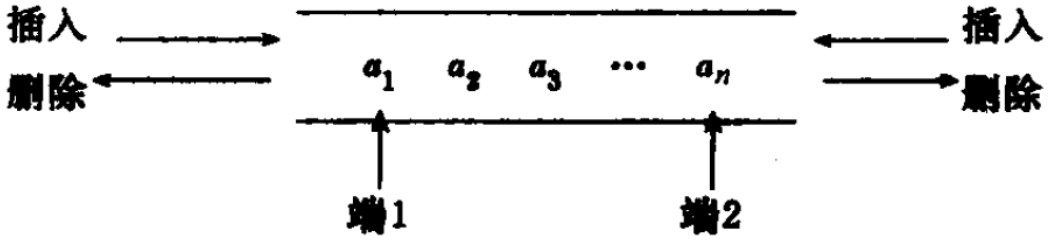
基本操作：　入队或出队，建空队列，判队空 或队满等操作。



实质是带头结点的线性链表

3.4 队列

双端队列　　限定插入和删除操作在表的两端进行的线性表



3.4 队列

队列的实现方式是本节重点，关键是掌握入队和出队操作。

具体实现依存储结构（链队或顺序队）的不同而不同。

队的抽象数据类型定义:

ADT Queue{

数据对象：D=……

数据关系：R=……

基本操作：

……

} ADT Queue

建队、入队或出队、判队空或队满等，教材P59-60罗列了9种基本操作。

第三章 栈和队列

- 3.1 栈
- 3.2 栈的实现
- 3.3 栈的应用
- 3.4 队列
- 3.5 队列的实现
- 3.6 队列的应用

3.5 队列的实现

（1）链队列

队列的链式存储结构简称为链队列，它是限制仅在表头删除和表尾插入的单链表。

在单链表的头指针不便于在表尾做插入操作，为此再增加一个尾指针，指向链表上的最后一个结点。于是，**一个链队列由一个头指针和一个尾指针唯一地确定。**

3.5 队列的实现

(1) 链队列

- 两个指针：
  - 队头指针Q.front指向头结点
  - 队尾指针Q.rear指向尾结点
- 初始态：队空条件
  - 头指针和尾指针均指向头结点

    Q.front = Q.rear

3.5 队列的实现

(1) 链队列

```
typedef  struct QNode { //元素结点
   QElemType    data;
   struct QNode *next;
}QNode, *QueuePtr;

typedef struct{                  //特殊结点
   QueuePtr  front;      //队头指针
   QueuePtr  rear;                 //队尾指针
}LinkQueue;
LinkQueue  Q;
```

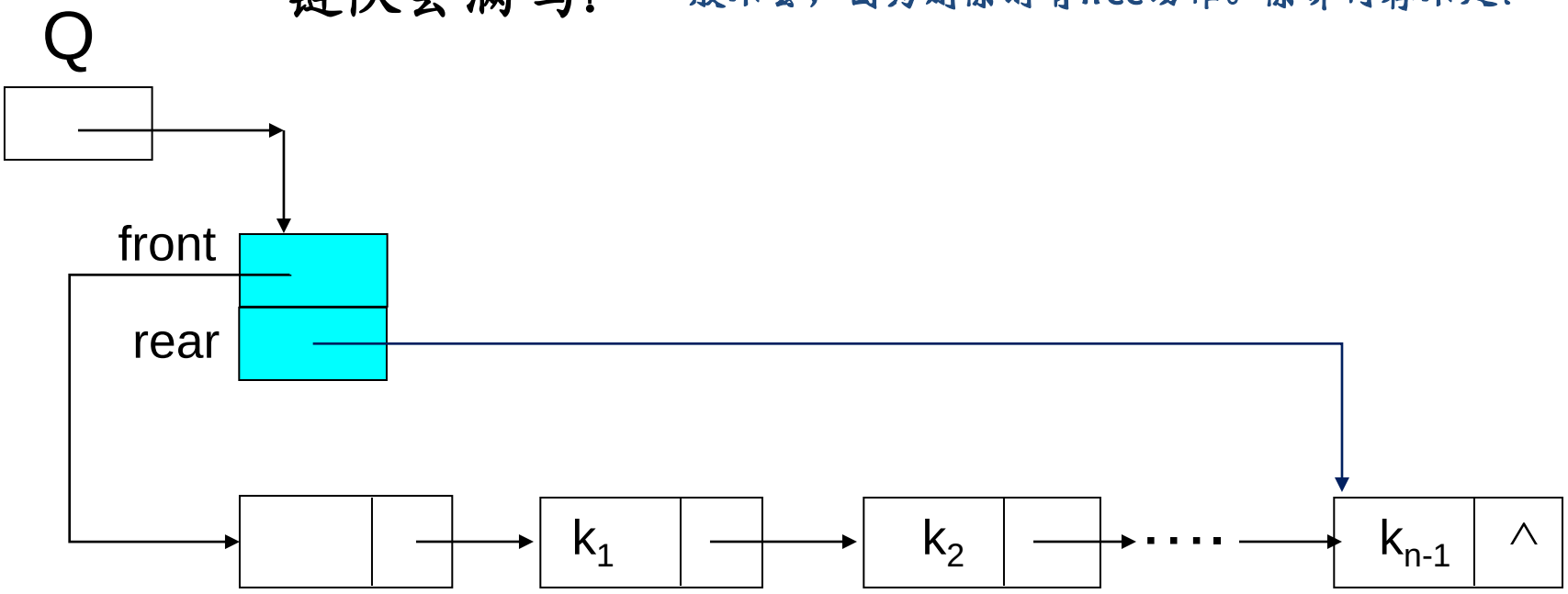
Q.front——指向链头结点

Q.rear ——指向链尾结点

3.5 队列的实现

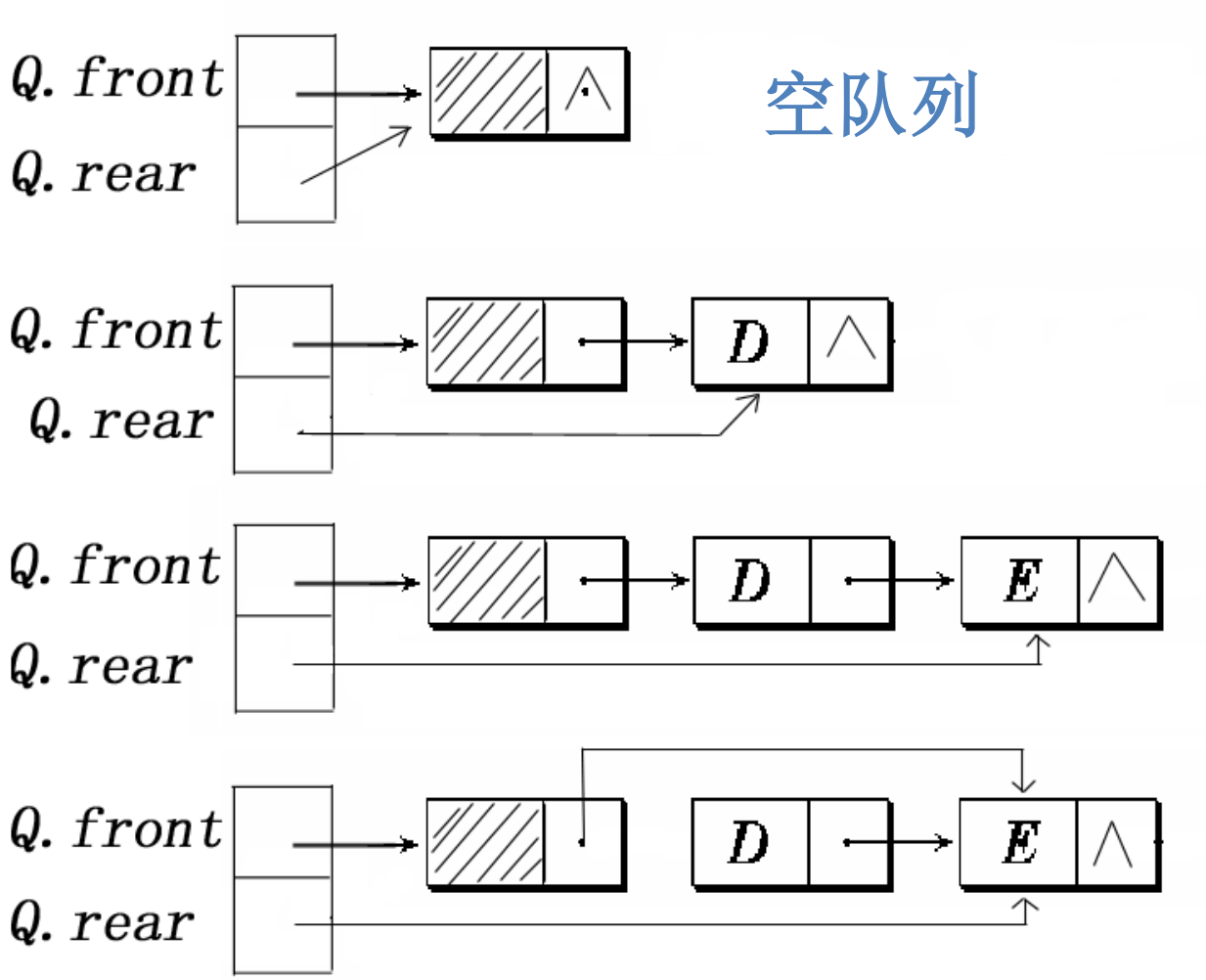
(1) 链队列

链队会满吗？ 一般不会，因为删除时有free动作。除非内存不足！



3.5 队列的实现

（1）链队列



3.5 队列的实现

(1) 链队列

- 初始化　p62
  Status InitQueue (LinkQueue &Q)
- 销毁队列　p62
  Status DestroyQueue (LinkQueue &Q)
- 入队　p62
  Status EnQueue(LinkQueue &Q, QElemType e)
- 出队　p62
  Status DeQueue(LinkQueue &Q, QElemType &e)
- 判队空
  Status QueueEmpty(LinkQueue Q)
- 取队头元素
  Status GetHead(LinkQueue Q,QElemType &e)

3.5 队列的实现

(1) 链队列

1) 链队列初始化

```
Status InitQueue (LinkQueue &Q)
{
   Q.front=Q.rear=(QueuePtr)malloc(sizeof(QNode));
   if (!Q.front) exit(OVERFLOW);
   Q.front->next=NULL;
   return OK;
}
```

3.5 队列的实现

(1) 链队列

2) 链队列的销毁

```
Status DestroyQueue (LinkQueue &Q)
{
   while (Q.front)
   {
        Q.rear=Q.front->next;
        free(Q.front);
        Q.front=Q.rear;
   }
   return OK;
}
```

3.5 队列的实现

(1) 链队列

3) 链队列的插入（入队）

```
Status EnQueue (LinkQueue &Q, QElemType e)
{
     p=(QueuePtr)malloc(sizeof(QNode));
     if (!p) exit(OVERFLOW);
     p->data = e;     p->next = NULL;
     Q.rear->next = p;
     Q.rear = p;
     return OK;
}
```

3.5 队列的实现

(1) 链队列

4) 链队列的删除（出队）

```
Status DeQueue (LinkQueue &Q, ElemType &e)
{
   if (Q.front==Q.rear) return ERROR;
   p=Q.front->next;
   e=p->data;
   Q.front->next=p->next;
   if (Q.rear == p) Q.rear=Q.front;
   free(p);
   return OK;
}
```

3.5 队列的实现

（1）链队列

5）判断链队列是否为空

```
Status QueueEmpty(LinkQueue Q)
{
   if (Q.front==Q.rear) return TRUE;
   return FALSE;
}
```

3.5 队列的实现

（1）链队列

6）取链队列的第一个元素结点

```
Status GetHead(LinkQueue Q,QElemType &e)
{
   if (QueueEmpty(Q)) return ERROR;
   e=Q.front->next->data;
   return OK;
}
```

3.5 队列的实现

(2) 顺序队列

队列的顺序存储结构简称为顺序队列。顺序队列实际上是运算受限的顺序表。

用一组地址连续的存储单元依次存放从队列头到队列尾的元素

设两个指针：

——Q.front 指向队列头元素；

——Q.rear 指向队列尾元素的下一个位置

3.5 队列的实现

(2) 顺序队列

```
#define MAXSIZE 100
typedef struct {
    QElemType  *base;
    int    front;
    int    rear;
}SqQueue;
SqQueue  Q;
```

3.5 队列的实现

(2) 顺序队列

讨论：

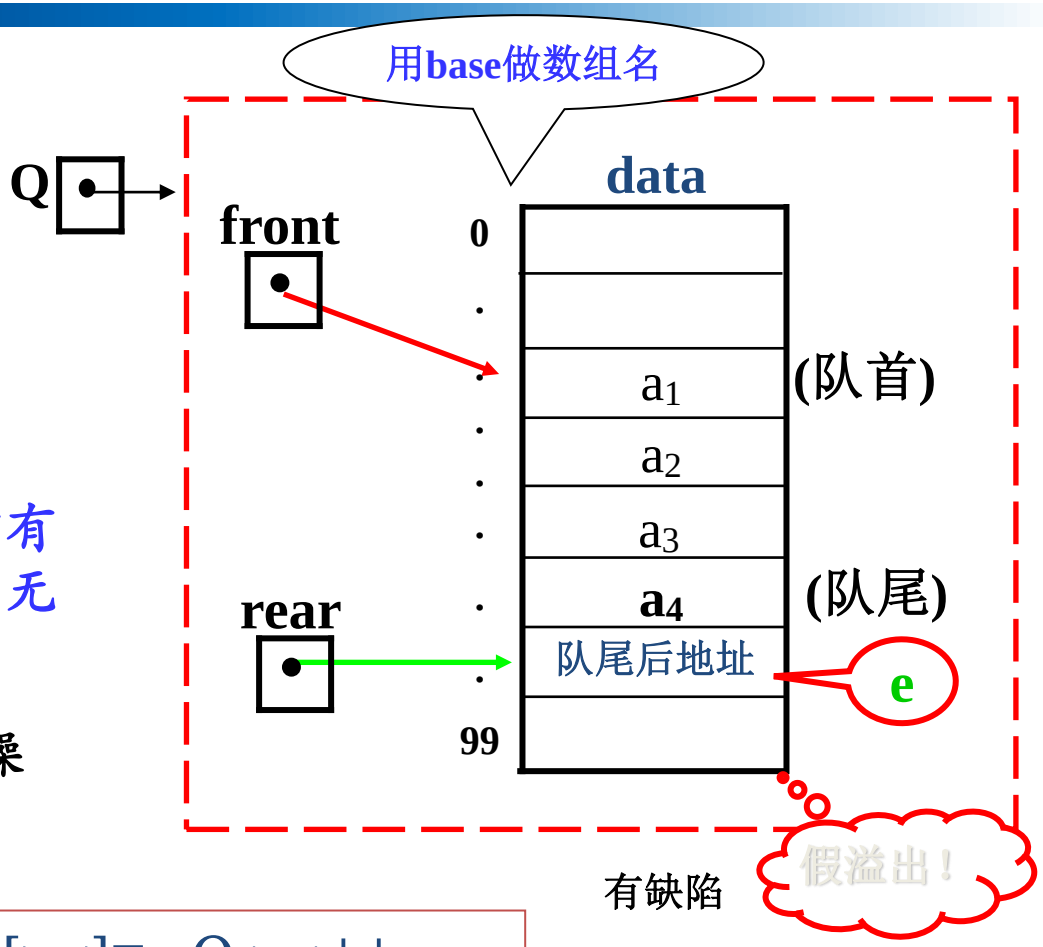
① 空队列的特征?

约定：front=rear

② 队列会满吗?

极易装满！因为数组通常有长度限制，而其前端空间无法释放。

③ 怎样实现入队和出队操作？核心语句如下：

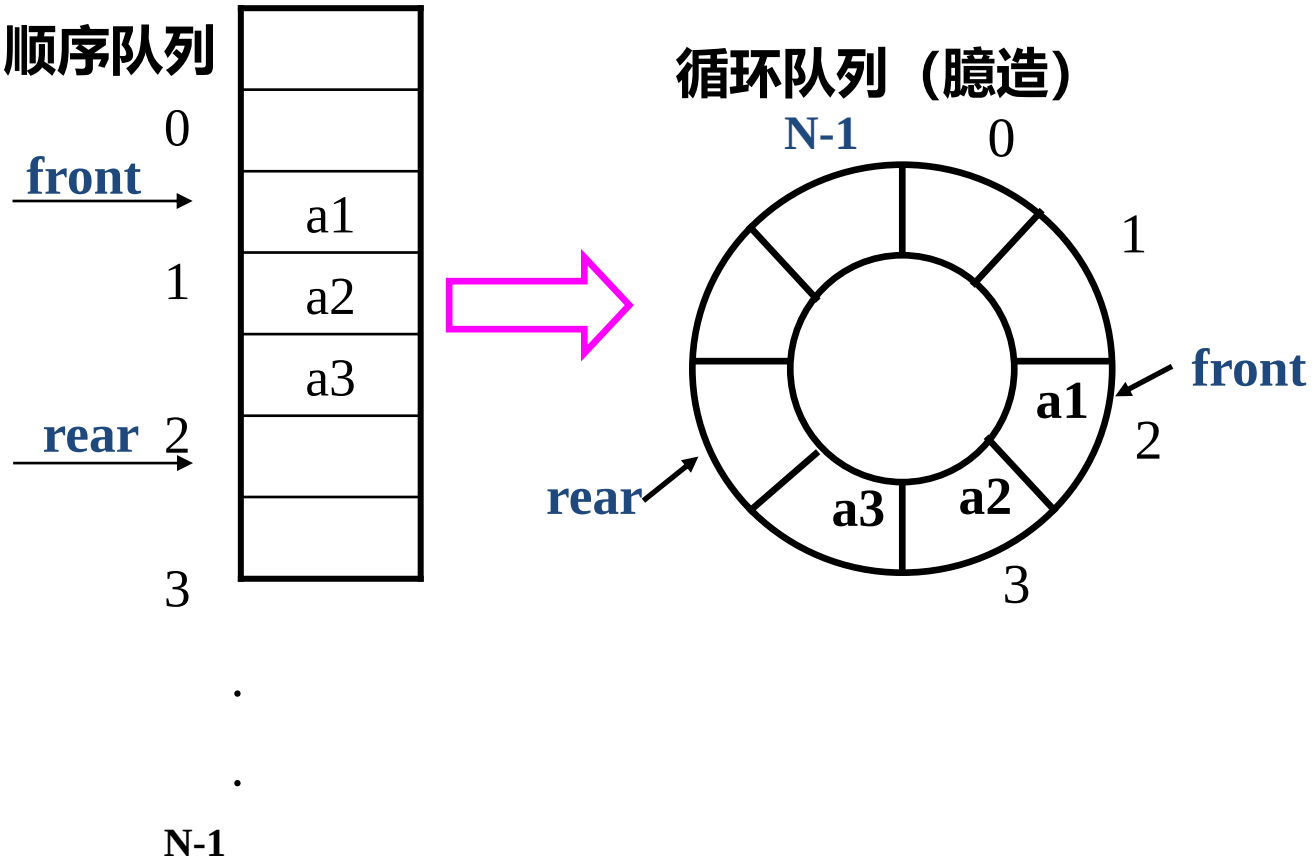


入队(尾部插入)： Q.base[rear]=e; Q.rear++ ;

出队(头部删除)： e=Q.base[front]; Q.front++;

3.5 队列的实现

(2) 顺序队列——循环队列



3.5 队列的实现

(2) 顺序队列——循环队列

新问题：在循环队列中，空队特征是front=rear；队满时也会有front=rear；判决条件将出现二义性！

解决方案有三：

①使用一个计数器记录队列中元素个数（即队列长度）；

②加设标志位，删除时置1,插入时置0,则可识别当前front=rear属于何种情况

③ 人为浪费一个单元，则队满特征可改为front=(rear+1)%N；

3.5 队列的实现

(2) 顺序队列——循环队列

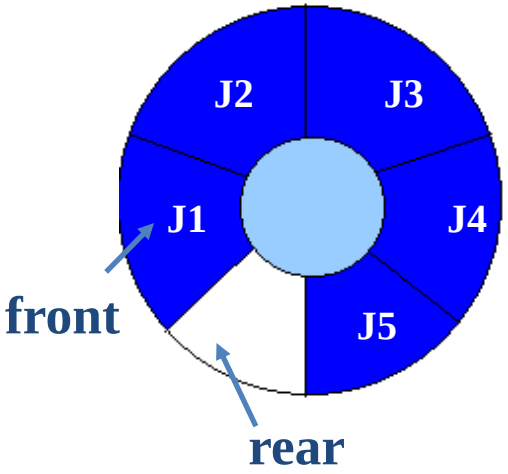
实际中常选用方案3（人为浪费一个单元）：

即front和rear二者之一指向实元素，另一个指向空闲元素。

队空条件：front = rear　　(初始化时：front = rear)

队满条件：front = (rear+1) % N　　(N=maxsize)

队列长度（即数据元素个数）：L= (N+rear−front) % N



问1：左图中队列maxsize N=？ 6

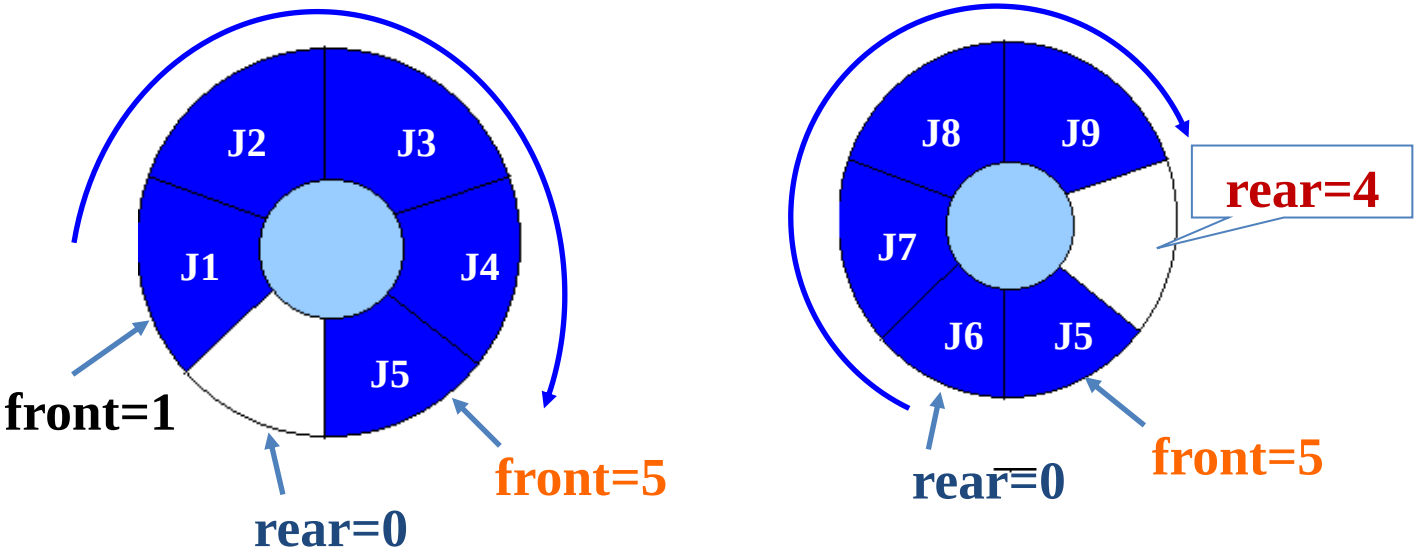
问2：左图中队列长度L=？ 5

问3：在具有n个单元的循环队列中，队满时共有多少个元素？ N-1个

3.5 队列的实现

(2) 顺序队列——循环队列

例：一循环队列如下图所示，若先删除4个元素，接着再插入4个元素，请问队头和队尾指针分别指向哪个位置?



解：由图可知，队头和队尾指针的初态分别为front=1和rear=0。

再插入4个元素后，r=(0+4)%6=4

删除4个元素后　　f=5；

3.5 队列的实现

(2) 顺序队列——循环队列

- 初始化　p64

Status InitQueue (SqQueue &Q)

- 求队列的长度　p64

int QueueLength(SqQueue Q)

- 入队　p65

Status EnQueue (SqQueue &Q, QElemType e)

- 出队　p65

Status DeQueue(SqQueue &Q, QElemType &e)

- 判队空

Status QueueEmpty(SqQueue Q)

- 取队头元素

Status GetHead(SqQueue Q,QElemType &e)

3.5 队列的实现

(2) 顺序队列——循环队列

1) 循环队列初始化

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

3.5 队列的实现

(2) 顺序队列——循环队列

2) 循环队列长度

```
int QueueLength(SqQueue Q)
{
    return (Q.rear-Q.front+MAXQSIZE)%MAXQSIZE;
}
```

3.5 队列的实现

(2) 顺序队列——循环队列

3) 循环队列入队

```
Status EnQueue (SqQueue &Q, QElemType e)
{
   if ((Q.rear+1) % MAXQSIZE ==Q.front)
      return ERROR;
   Q.base[Q.rear]=e;
   Q.rear = (Q.rear+1)%MAXQSIZE;
   return OK;
}
```

3.5 队列的实现

(2) 顺序队列——循环队列

4) 循环队列出队

```
Status DeQueue(SqQueue &Q, QElemType &e)
{
    if (Q.rear==Q.front) return ERROR;
    e=Q.base[Q.front];
    Q.front=(Q.front+1)%MAXQSIZE;
    return OK;
}
```

3.5 队列的实现

(2) 顺序队列——循环队列

5) 循环队列是否为空

```
Status QueueEmpty(SqQueue Q)
{
    if (Q.front==Q.rear) return TRUE;
    return FALSE;
}
```

3.5 队列的实现

(2) 顺序队列——循环队列

6) 循环队列取对头

```
Status GetHead(SqQueue Q,QElemType &e)
{
    if QueueEmpty(Q) return ERROR;
    e=Q.base[Q.front];
    return OK;
}
```

3.5 队列的实现

(2) 顺序队列

非循环队列

- 类型定义：与循环队列完全一样
- 关键：修改队尾/队头指针
  Q.rear = Q.rear + 1;　　Q.front = Q.front+1;
- 在判断时，有%MAXQSIZE为循环队列，否则为非循环队列
- 队空条件：　Q.front = Q.rear
- 队满条件：　Q.rear>= MAXQSIZE
- 注意“假上溢”的处理
- 长度：　　Q.rear - Q.front

第三章 栈和队列

- 3.1 栈
- 3.2 栈的实现
- 3.3 栈的应用
- 3.4 队列
- 3.5 队列的实现
- 3.6 队列的应用

3.6 队列的应用

（1）打印二项展开式 $(a+b)^i$ 的系数

$$(a+b)^n=\sum_{r=0}^{n}C_n^r a^{n-r}b^r$$

$$(a+b)^n=C_n^0a^n+C_n^1a^{n-1}b+\cdots+C_n^ka^{n-k}b^k+\cdots+C_n^nb^n$$

$$C_n^0,C_n^1,C_n^2,\cdots C_n^r,\cdots C_n^n$$

- 项数：总共二项式展开有n+1项
- 通项：第 $i+1$ 项的系数为 $C_n^i=\frac{i!}{i!(n-i)!}$
- 如果二项式的幂指数是偶数，中间的一项二次项系数最大。如果是奇数，则最中间2项最大并且相等

3.6 队列的应用

（1）打印二项展开式 $(a+b)^i$ 的系数

$(a+b)^1$ …… C_1^0 C_1^1

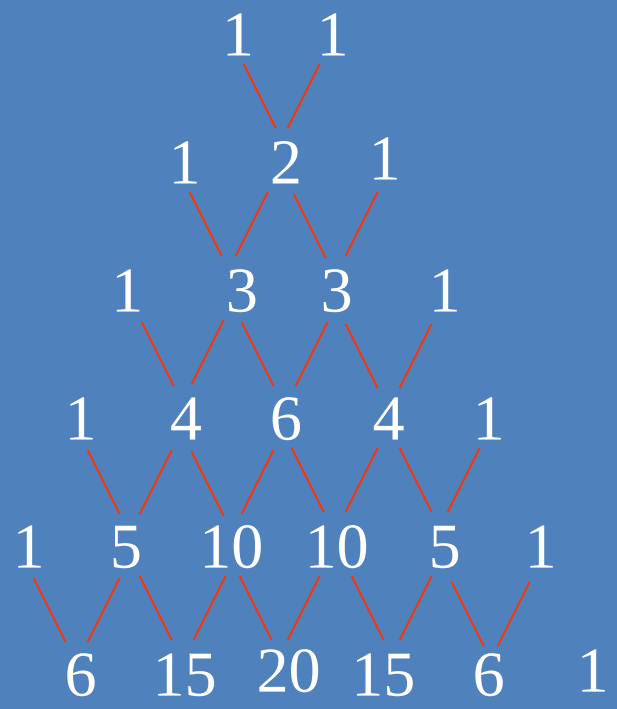
$(a+b)^2$ …… C_2^0 C_2^1 C_2^2

$(a+b)^3$ …… C_3^0 C_3^1 C_3^2 C_3^3

$(a+b)^4$ …… C_4^0 C_4^1 C_4^2 C_4^3 C_4^4

$(a+b)^5$ …… C_5^0 C_5^1 C_5^2 C_5^3 C_5^4 C_5^5

$(a+b)^6$ …… C_6^0 C_6^1 C_6^2 C_6^3 C_6^4 C_6^5 C_6^6

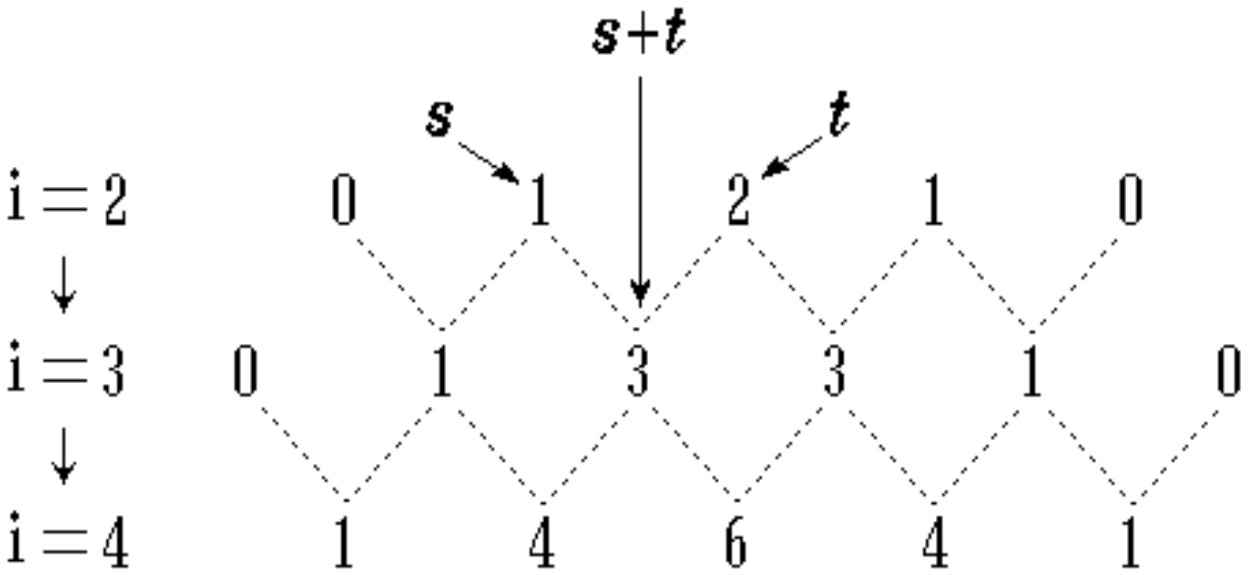


杨辉三角形（Pascal’s triangle）

3.6 队列的应用

（1）打印二项展开式 $(a+b)^i$ 的系数

杨辉三角形（Pascal’s triangle)

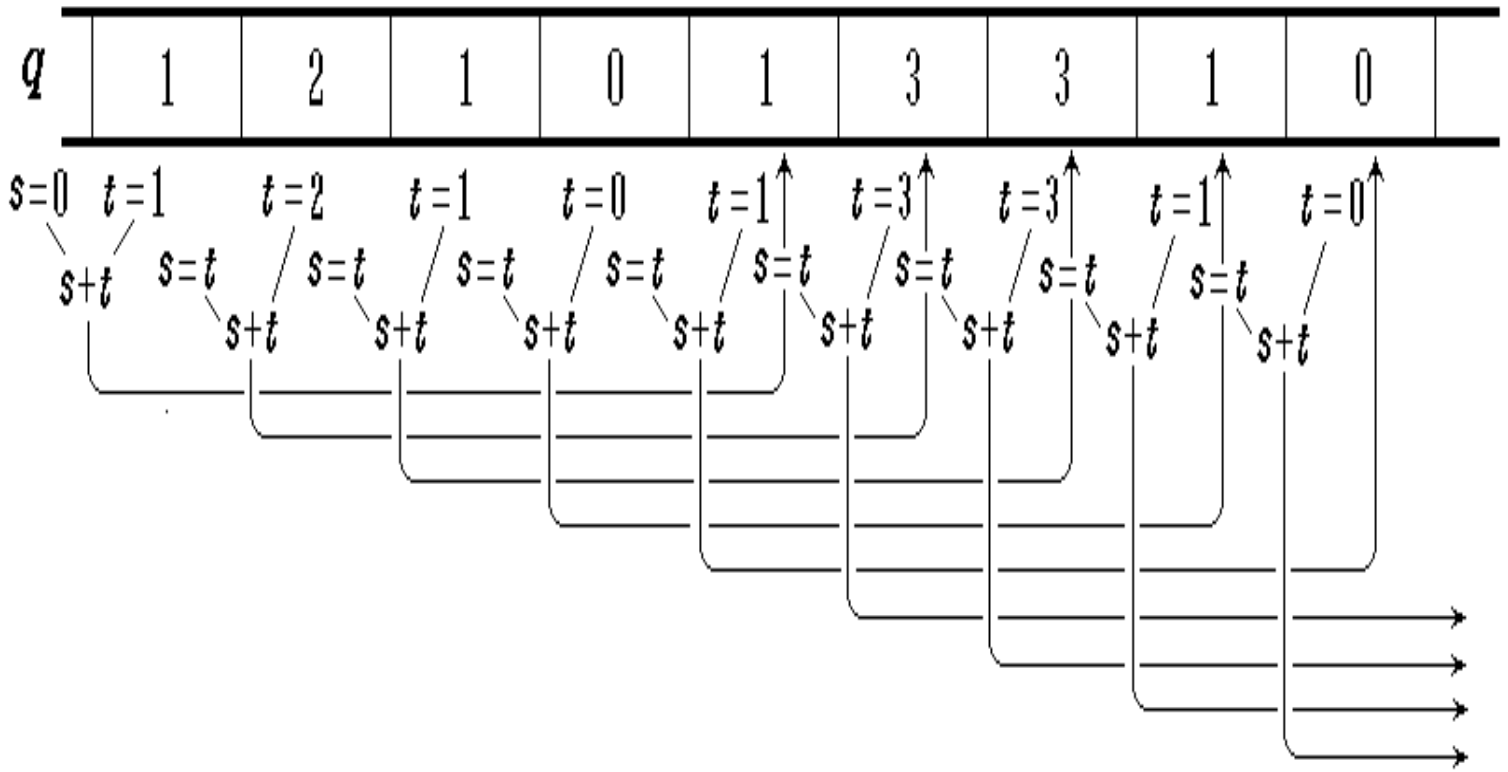


表中除“1”以外的每一个数都等于它肩上的两个数之和

3.6 队列的应用

（1）打印二项展开式 $(a+b)^i$ 的系数

从第 i 行数据计算并存放第 $i+1$ 行数据



3.6 队列的应用

（1）打印二项展开式 $(a+b)^i$ 的系数

```
void YANGVI ( int n ) {
    SqQueue q;
    q.InitQueue(&q);
    q.EnQueue (1);  q.EnQueue (1);//预放入第一行的两个系数
    int s = 0;
    for ( int i=1; i<=n; i++ ) {   //逐行处理
        cout << endl;
        q.EnQueue (0);
        for ( int j=1; j<=i+2; j++ ) {  //处理第i行的i+2个系数
            int t = q.DeQueue ( );//读取系数
            q.EnQueue (s+t);//计算下一行系数，并进队列
            s = t;
            if ( j != i+2 )
                cout << s << ‘ ’;//打印一个系数，第 i+2个为0
        }
    }
}
```

3.6 队列的应用

(2) 医院门诊部病人管理系统

- 工作过程：当一病人进入门诊室时，负责挂号的义务人员就根据观察和简短询问发给他一个从0（病危）到4（一般）变化的优先数，让他到相应优先数队列中去排队等待 。当一医生空闲时，就根据优先数和等待时间，通知某候诊病人就诊。

- 原则：优先级高的先考虑，同一优先级中，则先来先考虑。

3.6 队列的应用

(2) 医院门诊部病人管理系统

- **组织数据的方式 - - 数据结构**

✓ 组织方式一：若病人以其到达门诊部的先后次序组织到一个队列，则具体到达时间可不考虑。

这样的单队列对按就诊原则来确定下一就诊病人是很不方便的，还将破坏队列的操作原则。

到达次序	优先数	姓名
1	2	林文
2	3	郑江
3	0	鲁明
4	3	陈真
5	0	李军
6	2	王红
7	1	张兵

3.6 队列的应用

(2) 医院门诊部病人管理系统

- 组织方式一：若病人以其到达门诊部的先后次序组织到一个队列，则具体到达时间可不考虑。

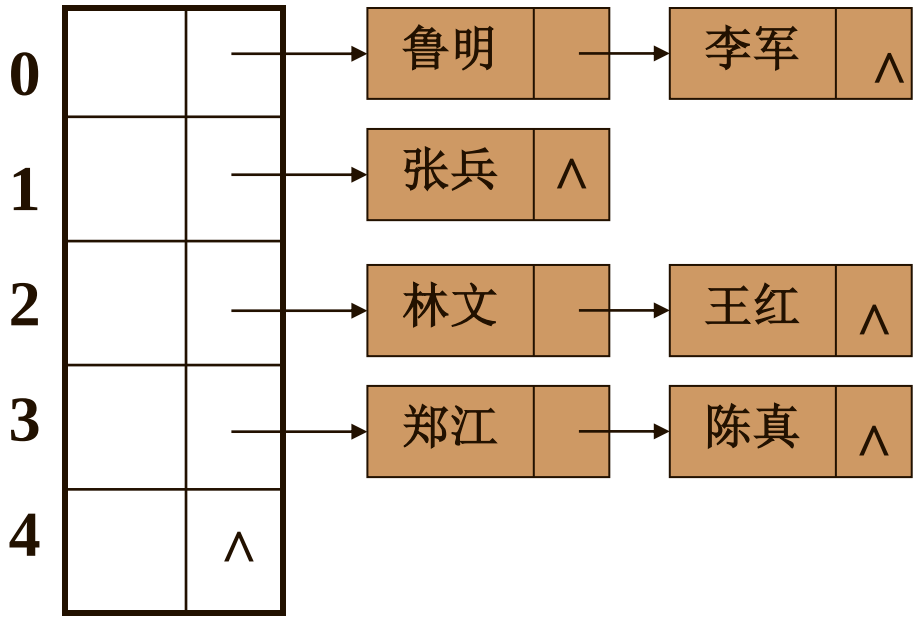
到达次序	优先数	姓名
1	2	林文
2	3	郑江
3	0	鲁明
4	3	陈真
5	0	李军
6	2	王红
7	1	张兵

这样的单队列对按就诊原则来确定下一就诊病人是很不方便的，还将破坏队列的操作原则。

3.6 队列的应用

(2) 医院门诊部病人管理系统

- 组织方式二：多个队列（队列数组）：队列数组的第i个元素是优先级为i的队列。优先数隐含在队列的编号中。



3.6 队列的应用

(2) 医院门诊部病人管理系统

- 就病人管理系统来说，功能菜单中至少有以下两个功能

（1）病人登记**AddPatient( )**

①提示并读入病人优先数i；

②提示并读入病人名

③调用链队列的入队算法EnQueue()

（2）确定下一个就诊病人 **GetPatient( )**

按就诊原则确定和显示下一个就诊病人名

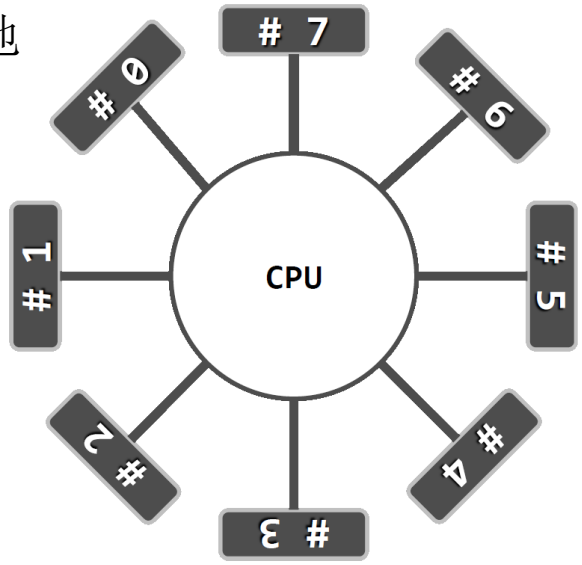
即：若优先数0的队列非空，则下一就诊病人为队首元素，否则，考虑优先数1的队列；依次类推。

3.6 队列的应用

（3）资源循环分配

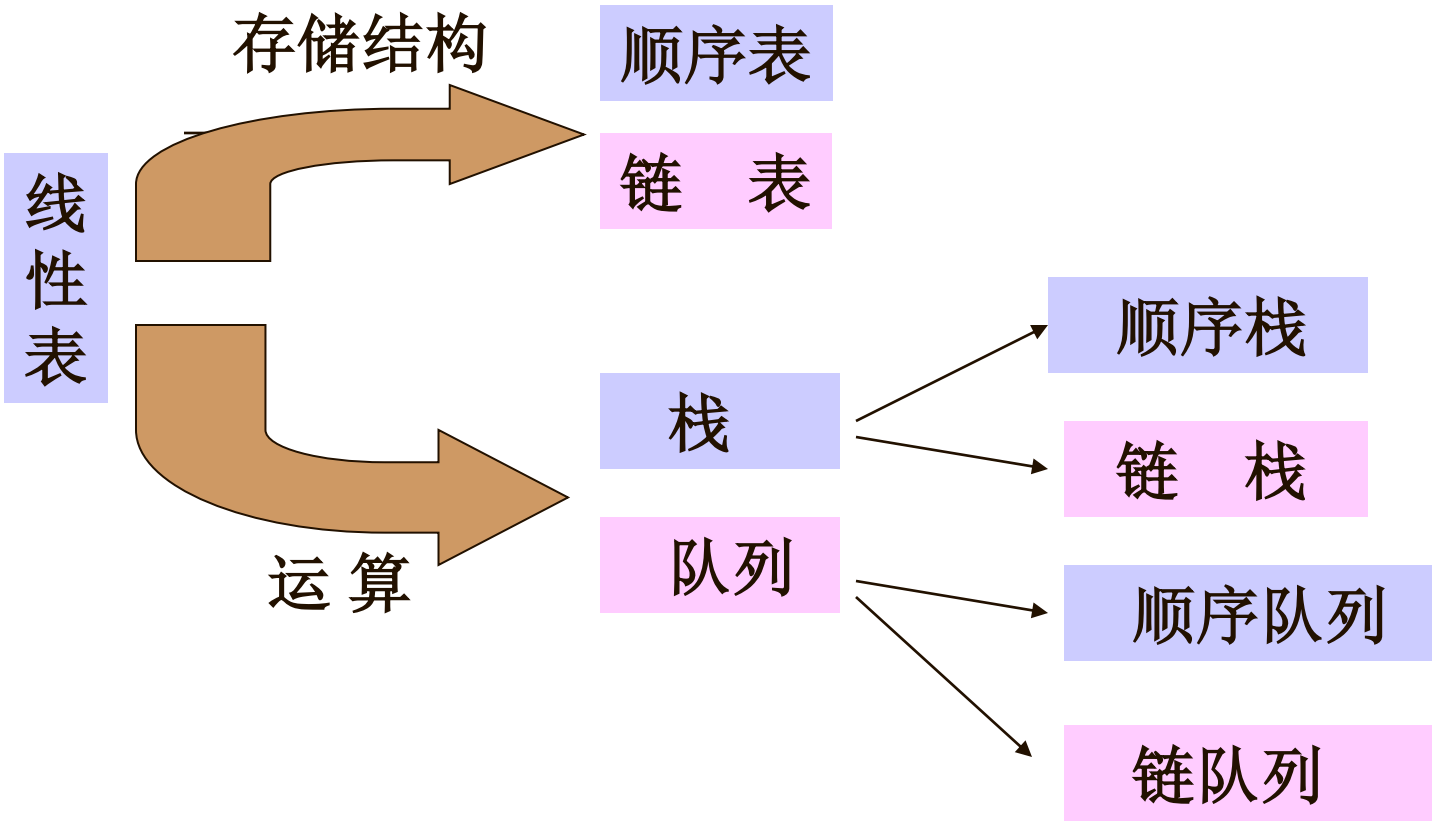
一群客户(client)共享同一资源时，如何兼顾公平与效率。比如，多个应用程序共享CPU，实验室成员共享打印机，……

```
RoundRobin { //循环分配器
    SqQueue Q(clients);//参与资源分配的所有客户组成队列
    while(!serviceclosed()){ //在服务关闭之前，反复地
        e= Q.DeQueue();//令队首的客户出队，并
         serve(e);//接受服务，然后
         Q.EeQueue(e);//重新入队
    }
}
```



- 利用队列改进迷宫算法，找出最短的通路

小结



小结

线性表、栈、队的异同点：

不同点：① 运算规则不同：

- ✓ 线性表为随机存取；
- ✓ 而栈是只允许在一端进行插入和删除运算，因而是后进先出表LIFO；
- ✓ 队列是只允许在一端进行插入、另一端进行删除运算，因而是先进先出表FIFO。

② 用途不同，线性表比较通用；堆栈用于函数调用、递归和简化设计等；队列用于离散事件模拟、OS作业调度和简化设计等。

小结

例：数组Q[n]用来表示一个循环队列，f为当前队列头元素的前一位置，r为队尾元素的位置。假定队列中元素的个数小于n，计算队列中元素的公式为:

(A) r - f　　(B) (n+f - r) % n
(C) n+r - f　　(D)√ (n+r - f) % n

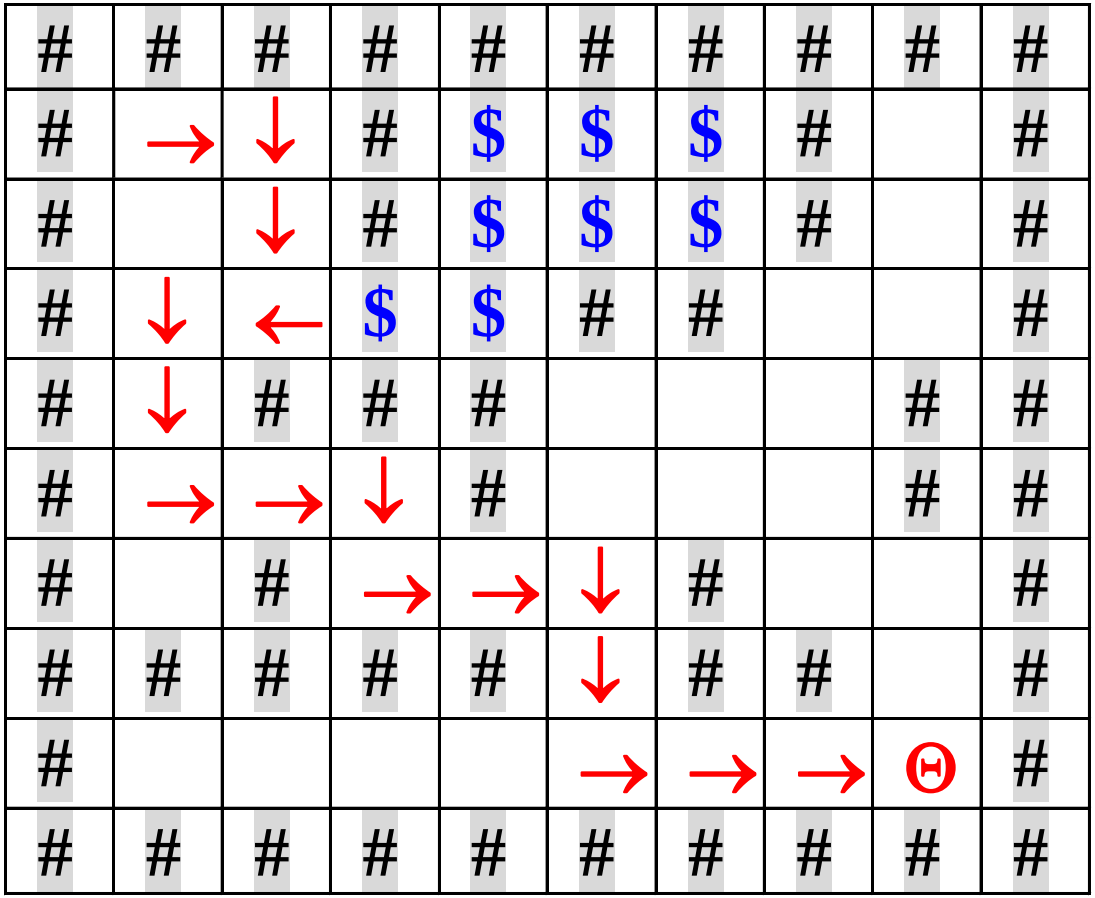
答：　要分析4种公式哪种最合理?

当r ≥f时（A）合理;
当r＜f时（C）合理;

} 综合2种情况，以（D）的表达最为合理

例题

- 迷宫求解



通常用的是“穷举求解”的方法

例题

- 迷宫求解

基本思想：

- 若当前位置“可通”，则纳入路径，继续前进;
- 若当前位置“不可通”，则后退，换方向继续探索;
- 若四周“均无通路”，则将当前位置从路径中删除出去。

例题

迷宫求解

```
设定当前位置的初值为入口位置;
 do {
  若当前位置可通,
  则 {将当前位置插入栈顶;
      若该位置是出口位置, 则算法结束;
      否则切换当前位置的东邻方块为新的当前位置;
  }
  否则 {
      若栈不空且栈顶位置尚有其他方向未被探索,
          则设定新的当前位置为: 沿顺时针方向旋转找到的栈顶位置的下一相邻块;
      若栈不空但栈顶位置的四周均不可通,
          则 { 删去栈顶位置; // 从路径中删去该通道块
               若栈不空, 则重新测试新的栈顶位置,
               直至找到一个可通的相邻块或出栈至栈空;
          }
   }
} while (栈不空) ;
```

例题

```
typedef struct {
  int ord；//通道块在路径上的 " 序号”
  PosType seat; //通道块 “坐标位置”
  int di; //从此通道块走向下一通道块的”方向”
} SElemType;
```

```
bool MazePath(PosType start, PosType end) {
    PosType curpos = start; Stack S; SELemType e;
    int curstep = 1; InitStack(S);
    do {
        if (Pass(curpos)) {//当前位置不是墙和已经走过的路径
            FootPrint(curpos); //留下足迹
            e.ord = curstep; e.seat = curpos; e.di = 1;
            Push(S, e); //加入路径
            if (IsEqual(curpos,end)) return true;
            curpos = NextPos(curpos, 1); curstep++; //探索下一步
        }
        else {//当前位置是墙或者是已经走过的路径
            if (!StackEmpty(S)) {
                Pop(S, e);
                while (e.di == 4 && (!StackEmpty(S))) {
                    MarkPrint(e.seat); Pop(S, e);//标记这个路线
                }
                if (e.di < 4) {
                    e.di++; Push(S, e); //换下一个方向探索
                    curpos = NextPos(e.seat, e.di);//设定当前位置是该新方向上的相邻块
                }
            }
        }
    } while (!StackEmpty(S));
    return false;
}
```

正在答疑