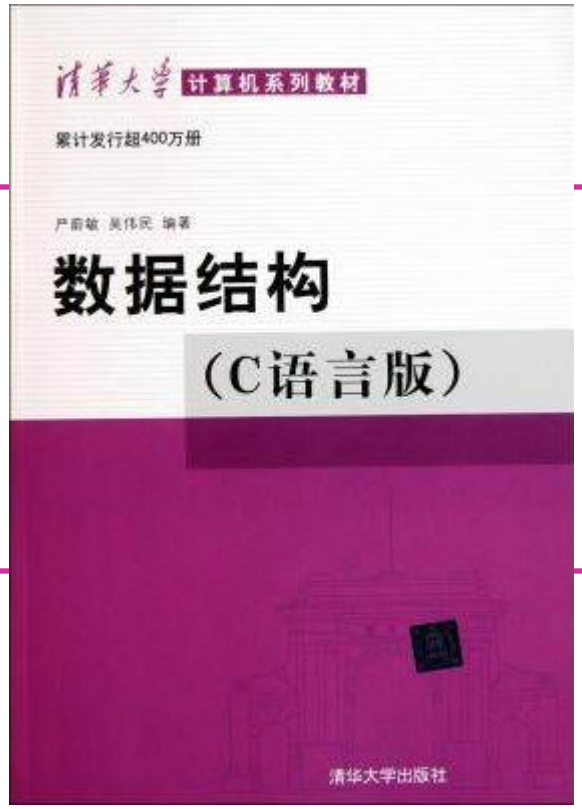


第3章　栈和队列

计算机科学与技术学院
主讲：孙玉霞

目　录

●基本要求：

1）理解栈和队列的概念，存储表示，进栈、退栈和进队、出队操作的算法；

2）对栈和队列的存储方式及基本操作有较深刻的理解；

3）初步了解栈的基本应用。如表达式的求值、递归的设计实现等。理解递归算法执行过程中栈的状态变化过程。

●学习重点：

1）栈和队列的基本操作；

2）栈在实现递归中的应用；

引　言

- 两种重要的受限线性结构：栈和队列

从数据结构角度看，栈和队列也是线性表，其特殊性在于栈和队列的基本操作是线性表操作的子集。它们是操作受限的线性表，即可称为限定性的数据结构。

但从数据类型角度看，它们又是和线性表大不相同的两类重要的抽象数据类型。

引　言

- **栈、队列**和线性表三者的操作规则比较

线性表： 可以在表的任何位置上插入、删除元素。

栈： 对栈仅限定在表的一端（表尾）进行插入或删除操作。可以在表的任何位置上插入、删除元素。

队列： 限定在表的一端进行插入，而在另一端删除元素。

因此，从ADT的角度看，栈、队列和线性表是不相同的。

3.1 栈

3.1.1 抽象数据类型"栈"的定义

● 定义：栈是限定仅在表的一端进行插入或删除操作的线性表。

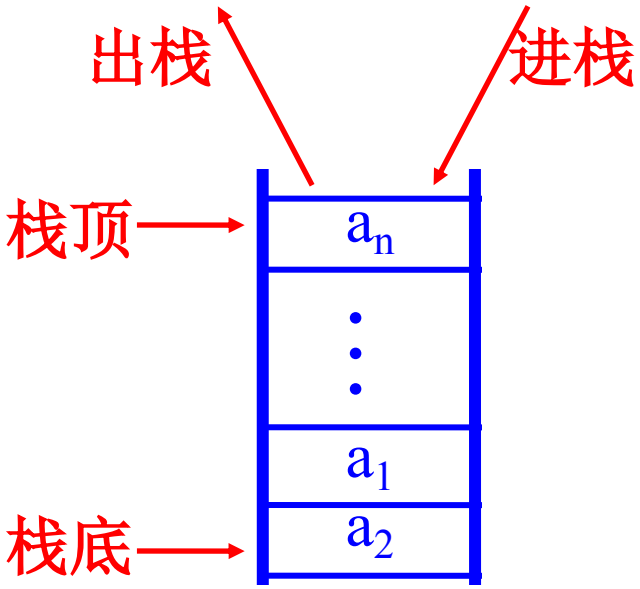


图3.1 栈

在栈中能进行插入和删除的一端称为栈顶（top），相应地另一固定端称为栈底（bottom）。将一个元素放入栈中的操作叫做进栈或压栈，从栈顶取出一个元素的操作叫做出栈或弹出。不含元素的空表称为空栈。

栈的存取操作符合后进先出（Last In First Out，LIFO）的原则，故栈又称为后进先出线性表，简称LIFO结构。

3.1 栈

3.1.1 抽象数据类型"栈"的定义

例3-1 ： A、B、C依次入栈，若不限出栈顺序，则出栈顺序有几种情况，请一一列出。

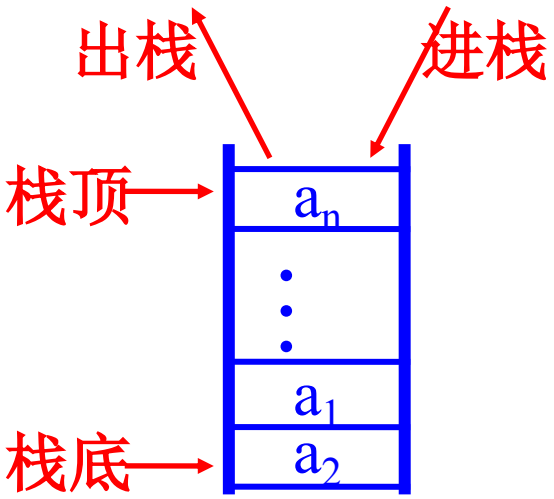


图3.1 栈

解： 出栈顺序有5种可能的情况，分别为：

ABC，ACB，BAC，BCA，CBA

CAB（加红色下划线的表示不可能）

3.1 栈

3.1.1 抽象数据类型"栈"的定义

例3-2 ： A、B、C、D依次入栈，若不限出栈顺序，则出栈顺序有几种情况，请一一列出。

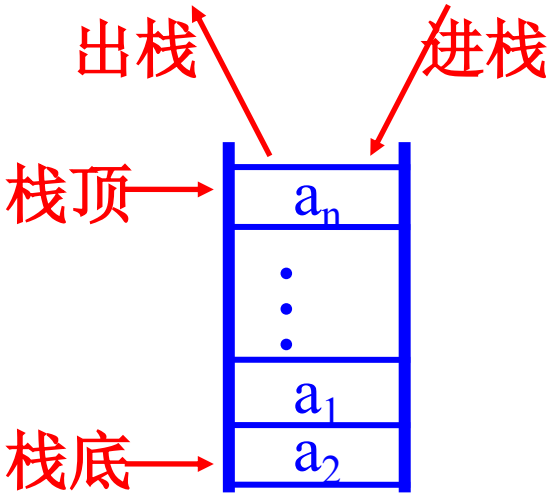


图3.1 栈

解： 出栈顺序有14种可能的情况，分别为：

ABCD，ABDC，ACBD，ACDB，ADBC，ADCB，
BACD，BADC，BCAD，BCDA，BDAC，BDCA，
CABD，CADB，CBAD，CBDA，CDAB，CDBA，
DABC，DACB，DBAC，DBCA，DCAB，DCBA。

规律： 按"大，小，中"顺序出栈是不可能的。

3.1 栈

3.1.1 抽象数据类型"栈"的定义

- 栈的抽象数据类型的定义如下：

ADT Stack{

数据对象：D={a_i|a_i∈ElemSet, i=1,2,...,n, n>=1}

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

基本操作：

InitStack(&S)；初始化操作生成一个空栈S

DestroyStack(&S)；释放栈

ClearStack(&S) ；清空栈

Push(&S, x)；入栈操作

Pop(&S, &e)；出栈操作

GetTop(S, &e) ；取栈顶元素函数

EmptyStack(&S)；置栈空操作

int StackLength(S)；求当前栈中元素个数

}

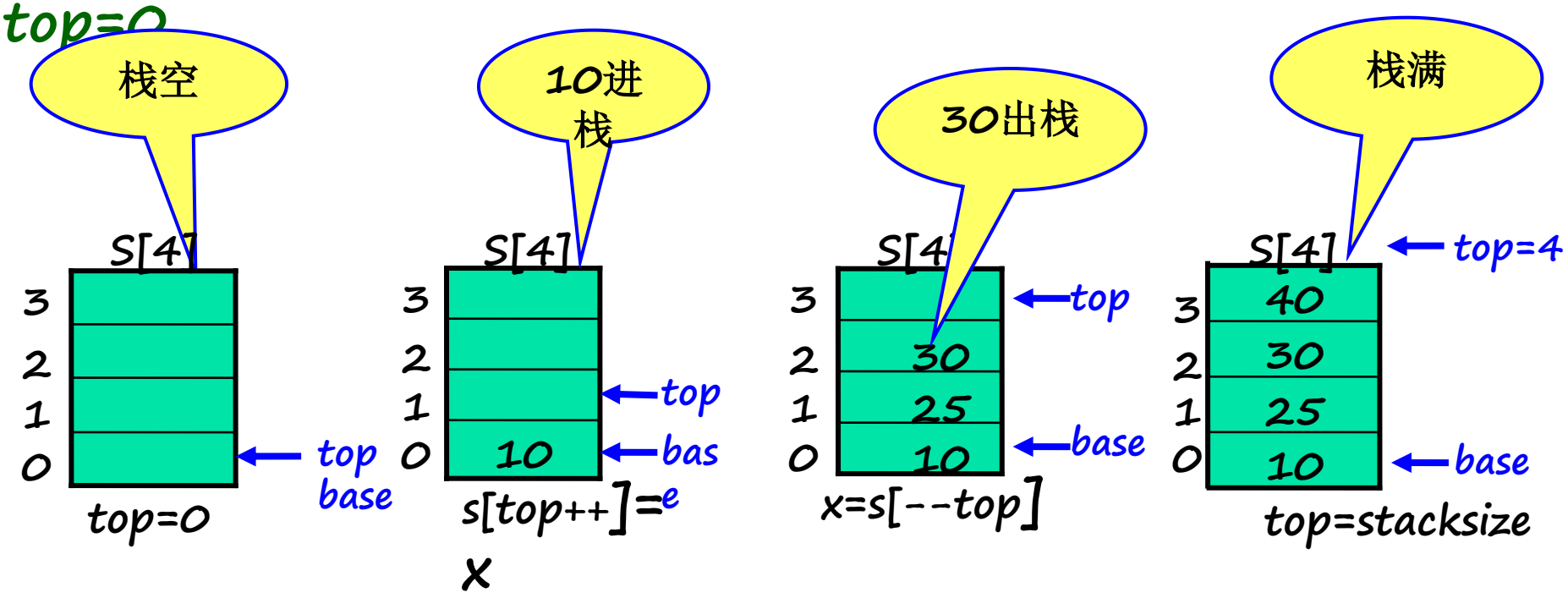
3.1 栈

3.1.2 "栈"的表示和实现

- **顺序栈——栈的顺序存储结构**

(1) 静态数组表示与实现

设数组S是一个顺序栈，栈的最大容量stacksize=4，初始状态top=0



3.1 栈

3.1.2 "栈"的表示和实现

- 顺序栈的表示与实现（静态数组）

```
const int MAX_STACK = 1000;
typedef struct {
        ElemType Element[MAX_STACK];
        int Top;
} SqStack;

//基本操作的算法描述（部分实现）
InitStack( SqStack &s)
{ s.top = 0
}//init_stack
Status isStackEmpty(SqStack s)
{  return ( s.top == 0 ) ? True : False;
}  //empty_stack
```

3.1 栈

3.1.2 "栈"的表示和实现

●顺序栈的表示与实现（静态数组）

```
Status Push(SqStack &s, ElemTp x )
{ if( s.top == MAX_STACK )
    return false;
  s.top++;
  s.elem[s.top] = x;
  return True;
} //push_stack

ElemTp Pop(SqStack &s )
{  if( s.top == 0 ) return;
   s.top--;
   return s.elem[s.top+1]
}   //pop_stack
```

3.1 栈

3.1.2 "栈"的表示和实现

●顺序栈的表示与实现（静态数组）

```
ElemTp GetTop(SqStack s )
{
    if( s.top == 0 ) return;
    return s.elem[s.top]
}  //gettop_stack

ClearStack(SqStack &s )
{
     s.top = 0
}     //clear_stack
```

3.1 栈

3.1.2 "栈"的表示和实现

●顺序栈的表示与实现（动态数组）

```
#define STACK_INI_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;
      SElemType *top;
      int stacksize;
}SqStack;
```

3.1 栈

3.1.2 ”栈”的表示和实现

●顺序栈的表示与实现（动态数组）

InitStack(&S) 操作结果：构造一个空栈 S。

```
Status InitStack(SqStack &S)
{
        S.base=(SElemType *)malloc(
                STACK_INI_SIZE*sizeof(SElemType));
        if(!S.base) exit(OVERFLOW);
        S.top=S.base;
        S.stacksize=STACK_INI_SIZE;
        return OK;
}
```

StackEmpty(S)

初始条件：栈 S 已存在。

操作结果：若栈 S 为空栈，则返回 TRUE，否则 FALE。

```
Status StackEmpty(SqStack S)
{
        if(S.top==S.base)
                return 1;
        else
                return 0;
}
```

```
Status Push(SqStack &S,SElemType e)
{  //栈 S 已存在。操作结果：插入元素 e 为新的栈顶元素
   if(S.top-S.base>=S.stacksize)
   {    //如果栈满，追加存储空间
        S.base=(SElemType *)realloc(S.base,
(S.stacksize+STACKINCREMENT)*sizeof(SElemType));
        if(!S.base) exit(OVERFLOW);
        S.top=S.base+S.stacksize;
        S.stacksize+=STACKINCREMENT;
   }
   *S.top=e;
   S.top++;
   return OK;
} //Push
```

```
Status Pop(SqStack &S,SElemType &e)
{ //栈 S 已存在且非空。操作结果：删除 S 的栈顶元素，并用 e 返回其值。
   if(S.top==S.base) return ERROR;
   e=*--S.top;
   return OK;
}
```

3.1 栈

3.1.2 "栈"的表示和实现

- **链式栈——栈的链式存储结构**

设不带头结点，用单链表表示栈。

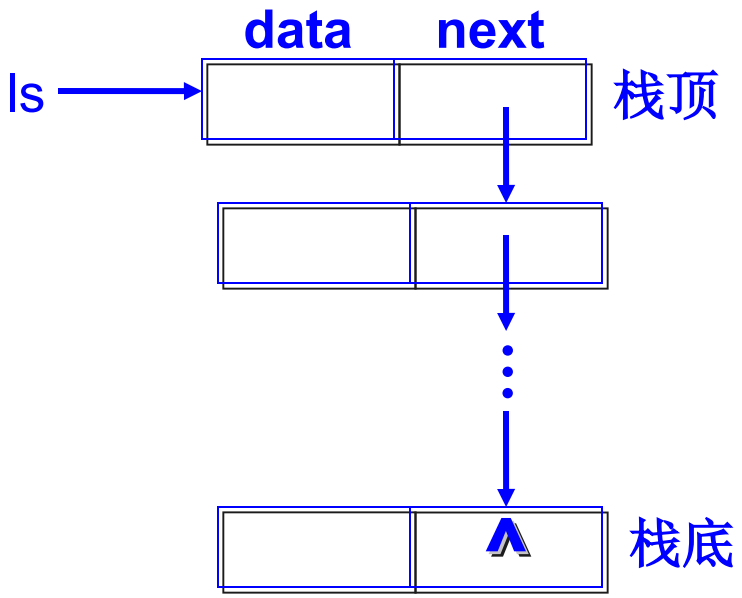


图3.3 链栈示意图

栈空：ls == NULL;
栈满：当内存分配无法实现时（堆区满）
进栈：在表头插入;
出栈：从表头删除;

数据结构的描述

```
elemtype … …  //任何一种数据类型

typedef struct tlkstktp{
  ElemType elem;
  tlkstktp *Next
} *lkstktp;
```

3.1 栈

3.1.2 "栈"的表示和实现

- 链式栈的表示与实现

```
PushStack(lkstktp &ls; elemtp x)
{ //算法中p为结点类型指针变量
        p = new(ElemType); p->elem = x;
        p->next = ls;
        ls = p
} //push_stack

elemtp PopStack(lkstktp &ls)
{ //算法中p为结点类型指针变量
        if( ls == NULL )    return NULL  // pop为elemtp类型变量
        p = ls; ls = p->next;
        pop = p->elem;
        free(p);
        return pop;
} //pop_stack
```

3.2 栈的应用举例

3.2.1 数制转换

3.2.2 括号匹配的检验

3.2.3 行编辑程序问题

3.2.4 迷宫求解

3.2.5 表达式求值

3.2 栈的应用举例

3.2.1 数制转换

算法基于原理： $N = (N \text{ div } d) \times d + N \bmod d$

```
void conversion ()
{   InitStack(S);
    scanf ("%d",N);
    while (N) {
    Push(S, N % 8);
    N = N/8;
 }
   while (!StackEmpty(S))
  {  Pop(S,e);
     printf ( "%d", e );
   }
} // conversion
```

3.2 栈的应用举例

3.2.2 括号匹配的检验

假设在表达式中

（［］（））或［（［］［］）］等为正确的格式，

［（］）或（［（））或（（）］)均为不正确的格式。

则 检验括号是否匹配的方法可用“期待的急迫程度”这个概念来描述

分析可能出现的不匹配的情况：

- 到来的右括弧并非是所“期待”的；
- 到来的是“不速之客”；
- 直到结束，也没有到来所“期待”的括弧。

3.2 栈的应用举例

3.2.2 括号匹配的检验

算法的设计思想：

1）凡出现**左括弧**，则**进栈**；

2）凡出现**右括弧**，首先检查栈是否空

若**栈空**，则表明该**“右括弧”多余**，否则**和栈顶元素**比较，若**相匹配**，则**“左括弧出栈”**，否则表明**不匹配**。

3）**表达式**检验**结束时**，

若**栈空**，则表明表达式中**匹配正确**，否则表明**“左括弧”有余**。

```
Status matching(string& exp) {
  int state = 1;
  while (i<=Length(exp) && state) {
    switch of exp[i] {
      case 左括弧:{Push(S,exp[i]); i++; break;}
      case”)”: {
        if(NOT StackEmpty(S)&&GetTop(S)=“(“
          {Pop(S,e);  i++;}
        else {state = 0;}
        break;  }   … …
  }
  if (StackEmpty(S)&&state) return OK; ……
```

3.2 栈的应用举例

3.2.3 行编辑程序

如何实现？

“每接受一个字符即存入存储器”？ 并不恰当！

在用户输入一行的过程中，允许用户输入出差错，并在发现有误时 可以及时更正。

合理的作法是：

设立一个输入缓冲区，用以接受用户输入的一行字符，然后逐行存入用户数据区，并假设“#”为退格符，“@”为退行符。

3.2 栈的应用举例

3.2.3 行编辑程序

输入字符串。用“#”表示退格符，即它使前一个字符无效；用“@”表示退行符，以表示此前整行字符均无效。编写行输入处理过程，内设一个栈来逐行处理从终端输入的字符串。每次从终端接收一个字符后先作如下判别：如果它既不是退格符也不是退行符，则将该字符插入栈顶；如果是一个退格符，则从栈顶删去一个字符；如果它是一个退行符，就把字符域清为空栈。比如，若键入

```
BGE##EGIM#N
RAD(A@      READ(A);
```

则实际有效的是下面两行

```
BEGIN
    READ(A);
```

3.2 栈的应用举例

3.2.3 行编辑程序

```
//设s为栈。本算法从终端接收一个正文文件并逐行传送至调用过程的数据
Void Line_Edit(){
    InitStack( S );                    //构造空栈S
    ch = getchar();                    //从终端接收一个字符
    while( ch != EOF )                 //EOF为全文结束符
    {    while( ch != EOF && ch != '\n' )
         { switch(ch){
              case '#': Pop(s,c); break;          //若栈非空则退栈
              case '@': ClearStack(s); break;     //将栈重新设置为空域
              default:  Push(s,ch); break;        //有效字符入栈
         };
```

3.2 栈的应用举例

3.2.3 行编辑程序

```
            ch = getchar()          //继续接收下一字符
        }
        将自栈底至栈顶的栈内字符作成一行,并传送至调用过程的数据区.
        ClearStack(s);              //重置S为空栈
        if( ch != EOF )
            ch = getchar()//继续接收下一行（先接收下一行第一个字符）
    }
} //LineEdit
```

3.2 栈的应用举例

3.2.4 迷宫求解

通常用的是“穷举求解”的方法

#	#	#	#	#	#	#	#	#	#
#	→	↓	#	$	$	$	#		#
#		↓	#	$	$	$	#		#
#	↓	←	$	$	#	#			#
#	↓	#	#	#				#	#
#	→	→	↓	#				#	#
#		#	→	→	↓	#			#
#	#	#	#	#	↓	#	#		#
#					→	→	→	Θ	#
#	#	#	#	#	#	#	#	#	#

3.2 栈的应用举例

3.2.4 迷宫求解

求迷宫路径算法的**基本思想**是：

- 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。

求迷宫中一条从入口到出口的路径的算法：

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**； //纳入路径

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）；

3.2 栈的应用举例

3.2.5 表达式求值

- 讨论简单算术表达式的求值问题。

$$\left.\begin{array}{l}\text{运算符}:+,-,*,/\\ \text{界限符}:(,),\#\end{array}\right\}\text{算符}op$$

使用**栈**实现简单算术表达式的求值运算。

依据算术四则运算的规则，在运算的每一步中。任何两个相继出现的算符θ_1和θ_2之间的优先关系，可能是

$\theta_1<\theta_2$	θ_1的优先数低于θ_2
$\theta_1=\theta_2$	θ_1的优先数等于θ_2
$\theta_1>\theta_2$	θ_1的优先数高于θ_2

3.2 栈的应用举例

3.2.5 表达式求值

表3.1 算符间的优先关系

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

3.2 栈的应用举例

3.2.5 表达式求值

说明：

1）+、－、*和/为θ1时的优先性均低于“(”，但高于“)”；

2）当$\theta_1=\theta_2$时，令$\theta_1>\theta_2$；

3）“(”＝“)”表示当左、右括号相遇时，括号内的运算已完成；

4）“#”是表达式的结束符。设表达式的最左边也虚设一个#符，以构成整个表达式的一对“括号”，

5）“(”与“#”，“)”与“(”以及“#”与“)”之间无优先关系，若出现此类情况，应是语法错误。下面的讨论，假定不会出现语法错误；

6）“#”＝“#”表示整个表达式求值完毕。

3.2 栈的应用举例

3.2.5 表达式求值

```
operandtype EvaluateExpression(){
/*算术表达式求值的算符优先算法。假定从终端输入的表达式无语法错误，以
'#'作结束符。设OPTR和OPND分别为运算符栈和操作数栈，OP为运算符的集合*/
    InitStack(OPTR); //初始化运算符栈
    Push(OPTR, '#'); //在运算符栈的栈底压入表达式左端的虚设字符#
    InitStack(OPND); //初始化操作数栈
    c = getchar();   //从终端接收一个字符
    while(( c != '#') || (GetTop(OPTR) != '#'))
    {   if( !In(c, OP)) {Push(OPND,c); c=getchar()}
        else
```

3.2 栈的应用举例

3.2.5 表达式求值

```
switch(precede(GetTop(OPTR),c)){ //比较优先级
    case '<':  //栈顶元素优先权低，运算符入栈
        Push(OPTR,c);c=getchar(); break;
    case '=':  //左括号遇右括号，脱括号并接收下一字符
        x =pop(OPTR);c=getchar(); break;
    case '>' : /*运算符及两个操作数分别退栈，结果入栈*/
        theta = Pop(OPTR);b=Pop(OPND);a=pop(OPND);
        Push(OPND,operate(a,theta,b));break;
}}     //CASE    WHILE
return(GetTop(OPND)) {返回最终结果值}
} //EvaluateExpression
```

3.2 栈的应用举例

3.2.5 表达式求值

例3-3：利用算法EvaluateExpression对算术表达式3*(7-2)求值。其操作过程如表3.2所示：

表3.2 3*(7-2)求值的操作过程

步骤	OPTR栈	OPND栈	输入字符	主要操作
1	#		3*(7-2)#	PUSH (OPND, '3')
2	#	3	*(7-2)#	PUSH (OPTR, '*')
3	# *	3	(7-2)#	PUSH (OPTR, '(')
4	# * (	3	7-2)#	PUSH (OPND, '7')
5	# * (	3 7	2)#	PUSH (OPTR, '-')
6	# * (-	3 7	2)#	PUSH (OPND, '2')
7	# * (-	3 7 2	)#	operate ('7','-','2')
8	# * (	3 5	)#	POP (OPTR) {消去一对括号}
9	# *	3 5	#	operate ('3','*','5')
10	#	15	#	RETURN (GETTOP (OPND))

3.3 栈与递归的实现

递归的概念

栈还有一个重要应用是在程序设计中实现递归。一个直接调用自己或通过一系列的调用语句间接地调用自己的函数，称做递归函数。

当在一个函数的运行期间调用另一个函数时，在运行该被调用函数之前，需先完成三项任务：

- 将所有的实参、返回地址等信息传递给被调用函数保存；
- 为被调用函数的局部变量分配存储区；
- 将控制转移到被调用函数的入口。

3.3 栈与递归的实现

从被调用函数返回调用函数之前，应该完成下列三项任务：

- 保存被调函数的计算结果；
- 释放被调函数的数据区；
- 依照被调函数保存的返回地址将控制转移到调用函数。

3.3 栈与递归的实现

递归函数的3个示例

例 阶乘函数

非递归形式

$$Fact(n)=1\cdot 2\cdot 3\cdot\cdots\cdot(n-1)\cdot n \quad (n\geq 0,\ \text{定义}Fact(0)=1)$$

递归形式

$$Fact(n)=\begin{cases}1 & ,n=0\\ n.Fact(n-1) & ,n>0\end{cases}$$

用类C写求阶乘函数

```
int fact( int n )
{  if( n<0 ) ERROR('n应大于0');
   else if( n == 0 ) return 1;          //递归出口
        else return( n * fact(n-1));
}    //fact
```

注：递归函数必须有递归终结分支（递归出口）！

3.3 栈与递归的实现

递归函数的3个示例

例 斐波那契(Fibonacci)数列

非递归形式

$$Fib(n)=\frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n-\left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

递归形式

$$Fib(n)=\begin{cases}0 & ,n=0\\ 1 & ,n=1\\ Fib(n-1)+Fib(n-2), & n\geq 2\end{cases}$$

写出Fibonacci数列递归函数：

```
int fib( int n )
{  switch( n ){
   case n<0: ERROR('n应大于0'); break;
   case n=0: return(0); break;   //递归出口
   case n=1: return(1); break;   //递归出口
   default:
      return(fib(n-1)+fib(n-2));
} }     //switch //fib
```

3.3 栈与递归的实现

递归函数的3个示例

例： n阶Hanoi塔问题(p55)

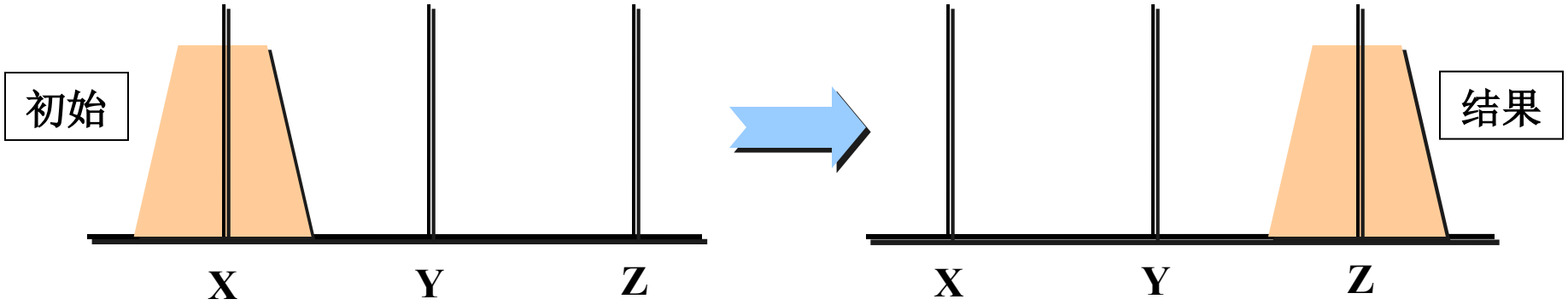


图3.4 3阶Hanoi塔问题的初始与结果状态

圆盘移动规则

1）每次只能移动一个圆盘；

2）圆盘可以插在X，Y和Z中的任一塔座上；

3）任何时刻都不能将一个较大的圆盘压在较小的圆盘上。

3.3 栈与递归的实现

递归函数的3个示例

[分析]此问题可归之于三个子问题

(1) 从X移动1~n-1号圆盘至Y，Z作辅助；（递归）

(2) 从X移动n号圆盘至Z；（一次搬运）

(3) 从Y移动1~n-1号圆盘至Z，X作辅助。（递归）

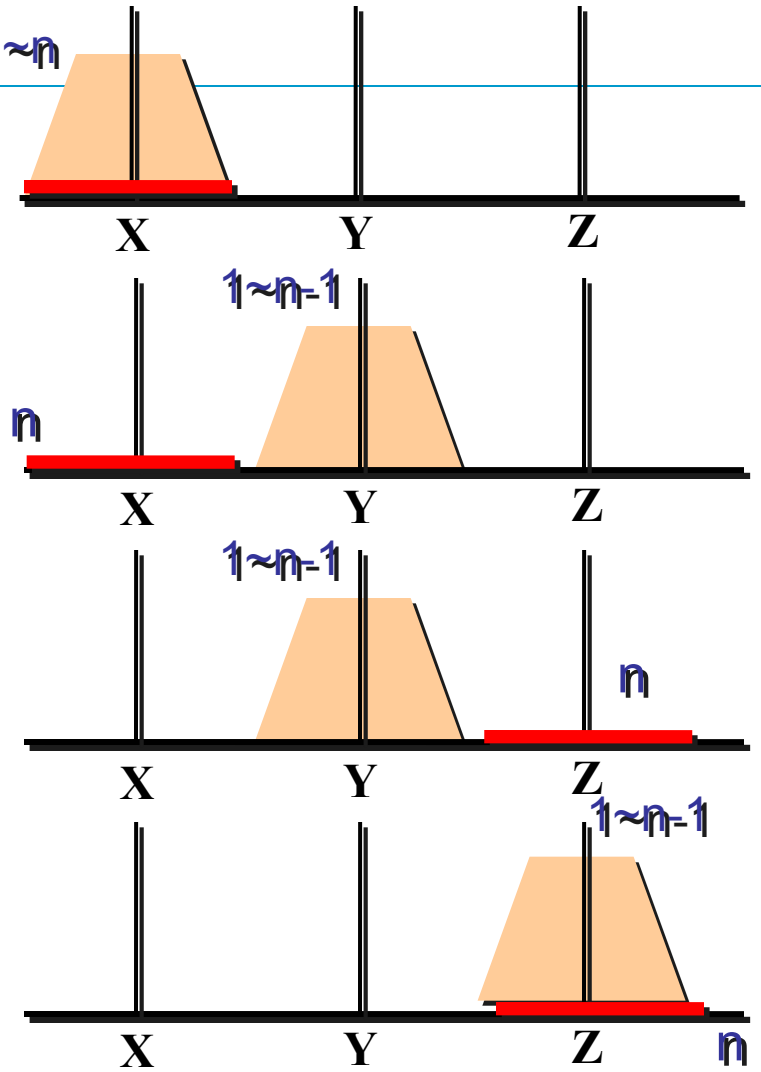
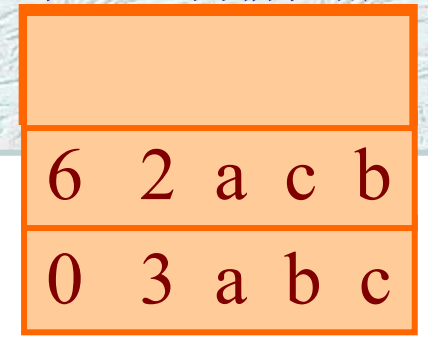


图3.5

3.3 栈与递归的实现

[解答]求解n阶Hanoi塔问题的递归算法。

```
Void hanoi( int n, char x, char y, char z)//将塔座x上编号从1至n、直径依小至大的n个圆盘移到塔座z上，y可用作辅助塔座
1 {
2     if( n==1 )
3         move(x,1,z);   //将编号为1的圆盘从x移到z
4     else if( n>1 )
      {
5         hanoi(n-1,x,z,y); //将x上编号从1至n-1的圆盘移到y上，z为辅助塔座
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上编号从1至n-1的圆盘移到z上，x为辅助塔座
8     }
9 }     //hanoi
```



返址 n x y z

3.4 队 列

3.4.1 抽象数据类型"队列"的定义

● 队列的定义

队列是限定仅能在表的一端进行插入，在表的另一端进行删除的线性表。在队列中可以插入的一端称为队尾(rear)，可以删除的一端称为队头或队首(front)。

将一个元素插到队列中的操作叫做入队列或进队，从队列中删除一个元素的操作叫做出队列或出队。队列是一种先进先出(First In First Out,FIFO)的线性表，或简称队列是一种FIFO结构。

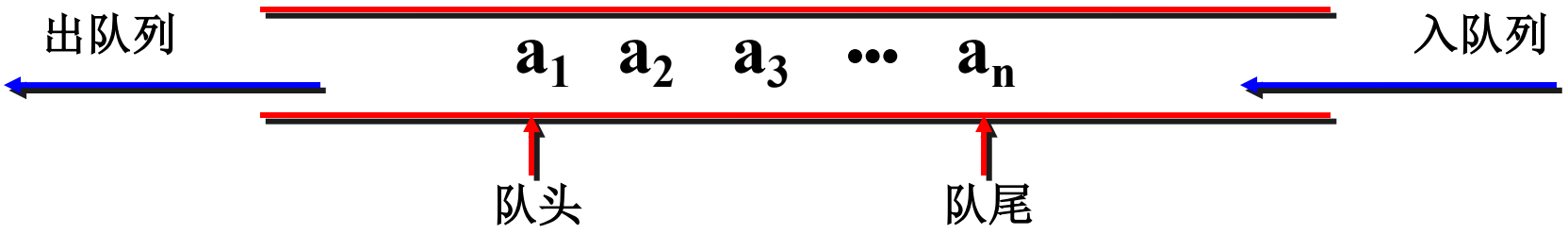


图3.6 队列示意图

3.4 队 列

3.4.1 抽象数据类型"队列"的定义

- 队列的ADT定义

```
ADT Queue{
  数据对象：D={a_i|a_i∈ElemSet, i=1,2,...,n, n>=1}
  数据关系：R={<a_i,a_i+1>|ai,a_i+1 ∈D, i=1,2,...,n}
  基本操作：
    InitQueue( &Q )；初始化构造一个空队列Q
    DestroyQueue( &Q )；释放队列Q
    ClearQueue( &Q ) ；将Q清为空队列
    EnQueue( &Q, x )；入队列操作
    DeQueue( &Q, &e )；出队列操作
    GetTop( S, &e ) ；取栈顶元素函数
    QueueEmpty ( &S )；判断栈是否为空
    QueueLength( Q )；求当前队列Q中元素个数
}
```

3.4 队 列

3.4.2 链队列——队列的链式表示和实现

- **链队列**：用链表表示的队列。

链队列应包含两个指针：队头指针、队尾指针。
为了操作方便，给链队列附加一个头节点。

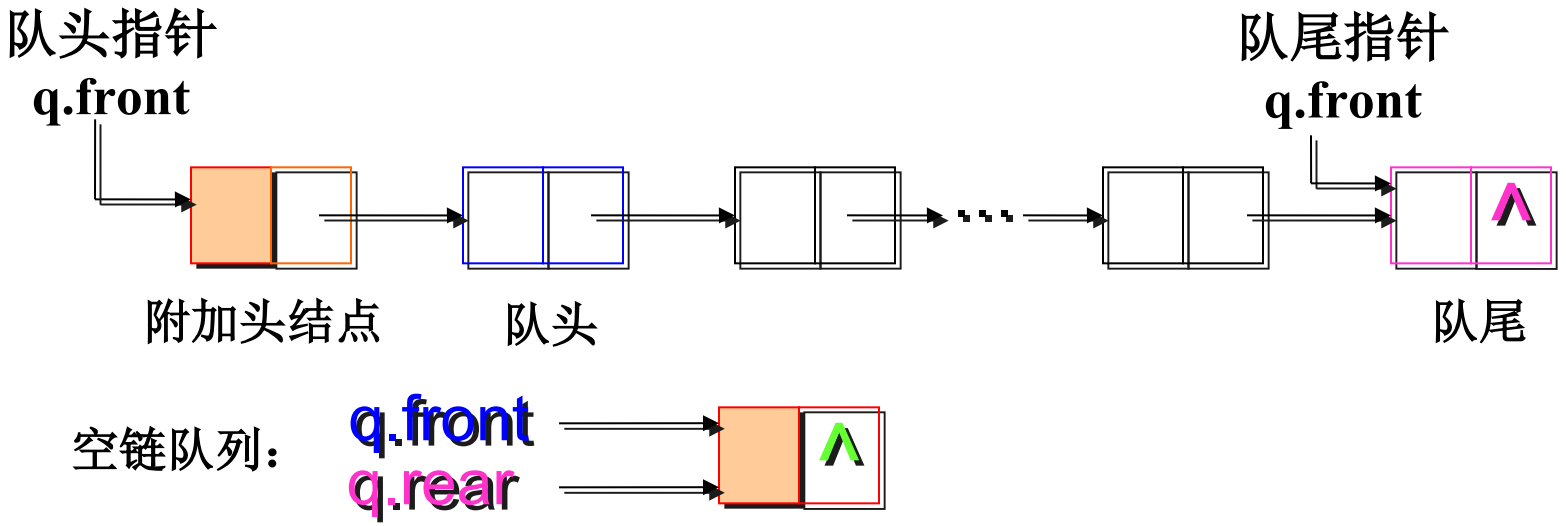


图3.7 链队列示意图

3.4 队 列

3.4.2 链队列——队列的链式表示和实现

- 链队列的表示

```
typedef struct QNode
{
   QElemType data;
   struct QNode *next;
}QNode,*QueuePtr;
```

```
typedef struct LQueue
{
    QNode *front;
    QNode *rear;
}LinkQueue;
```

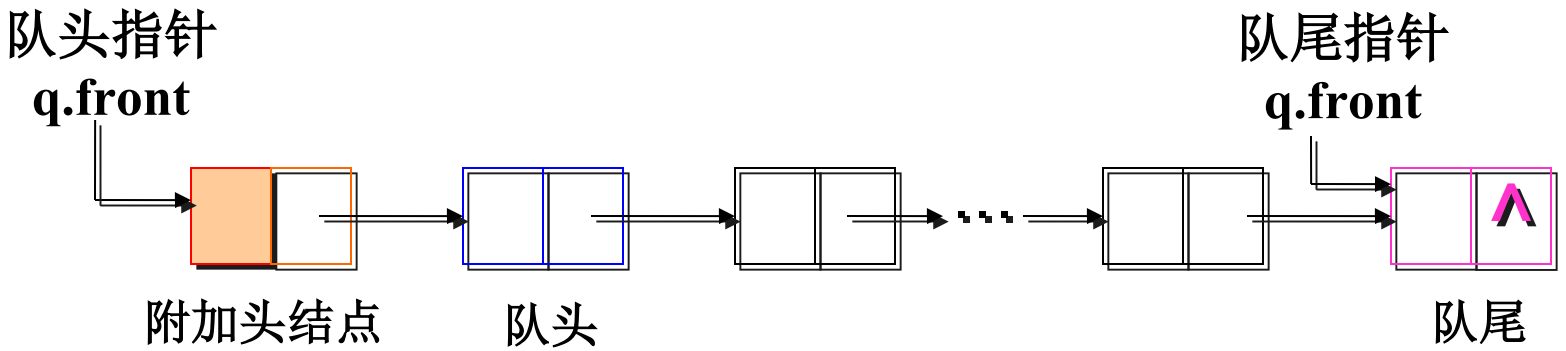
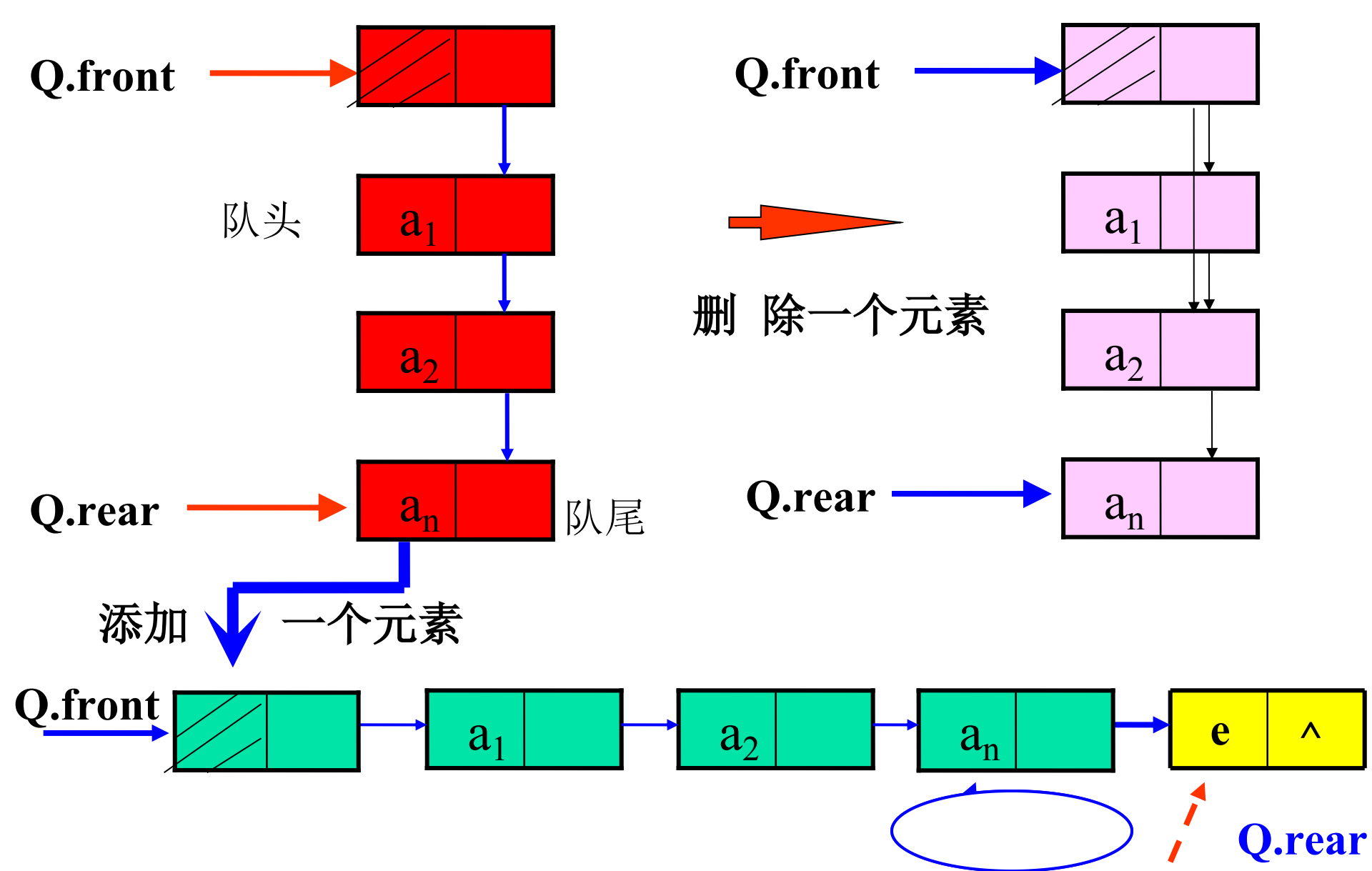


图3.8 链队列示意图


Q.front
队头
a_1
a_2
a_n
Q.rear
队尾
添加 一个元素
Q.front
a_1
a_2
a_n
e
^
Q.rear
Q.front
a_1
a_2
a_n
Q.rear
删 除一个元素

3.4 队 列

3.4.2 链队列——队列的链式表示和实现

链队列的实现

```
Status InitQueue (LinkQueue &Q)
{
  // 构造一个空队列Q
  Q.front = Q.rear =(QueuePtr)malloc(sizeof(QNode));
  if (!Q.front) exit (OVERFLOW);      //存储分配失败
  Q.front->next = NULL;
  return OK;
}
```

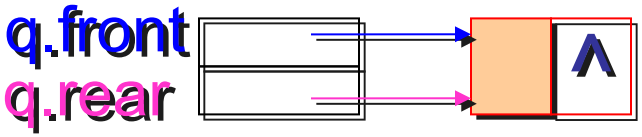


图3.9 链队列初始化

3.4 队 列

3.4.2 链队列——队列的链式表示和实现

链队列的实现

```
Status EnQueue (LinkQueue &Q, QElemType e)
{
   // 插入元素e为Q的新的队尾元素
   p = (QueuePtr) malloc (sizeof (QNode));
   if (!p)  exit (OVERFLOW);   //存储分配失败
   p->data = e;   p->next = NULL;
   Q.rear->next = p;    Q.rear = p;
   return OK;
}
```

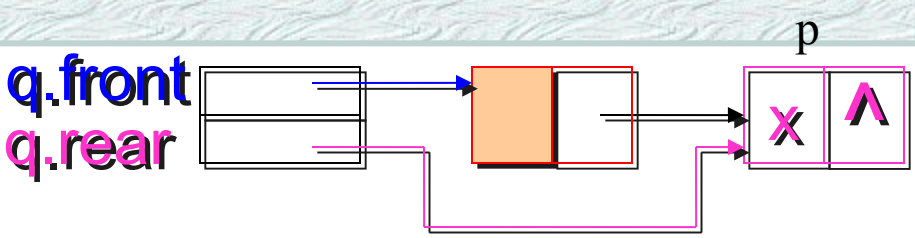


图3.10　建立一个新结点（值为x），并插入链队列队尾

3.4 队 列

3.4.2 链队列——队列的链式表示和实现

链队列的实现

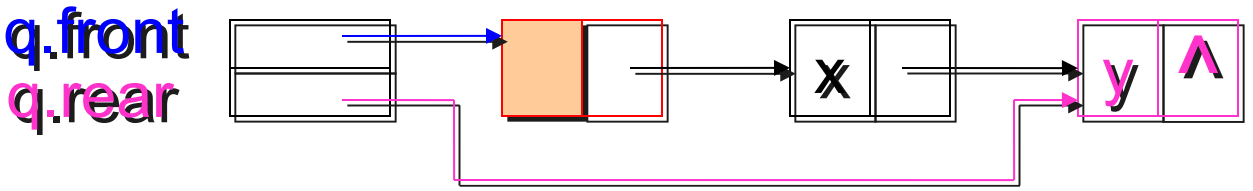


图3.11 建立一个新结点（值为y），并插入链队列队尾

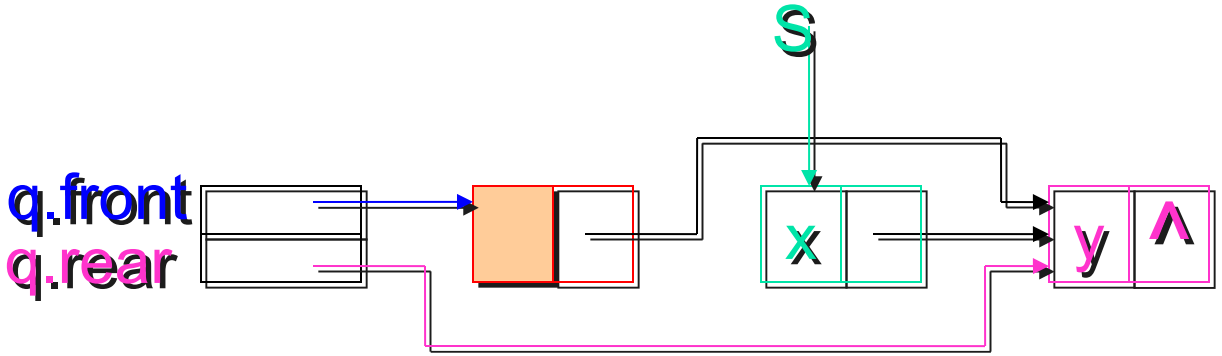


图3.12 将队头结点（值为x）删除（出队列）

3.4 队 列

3.4.2 链队列——队列的链式表示和实现

- 链队列的实现

```
Status DeQueue (LinkQueue &Q, QElemType &e)
{
  // 若队列不空，则删除Q的队头元素，
  //用 e 返回其值，并返回OK；否则返回ERROR
   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;
                              //删的是最后一个结点
   free (p);        return OK;
}
```

3.4 队 列

3.4.3 顺序队列——队列的顺序表示和实现

用数组实现队列：定义一个数组和分别指向队头元素和队尾元素的游标。

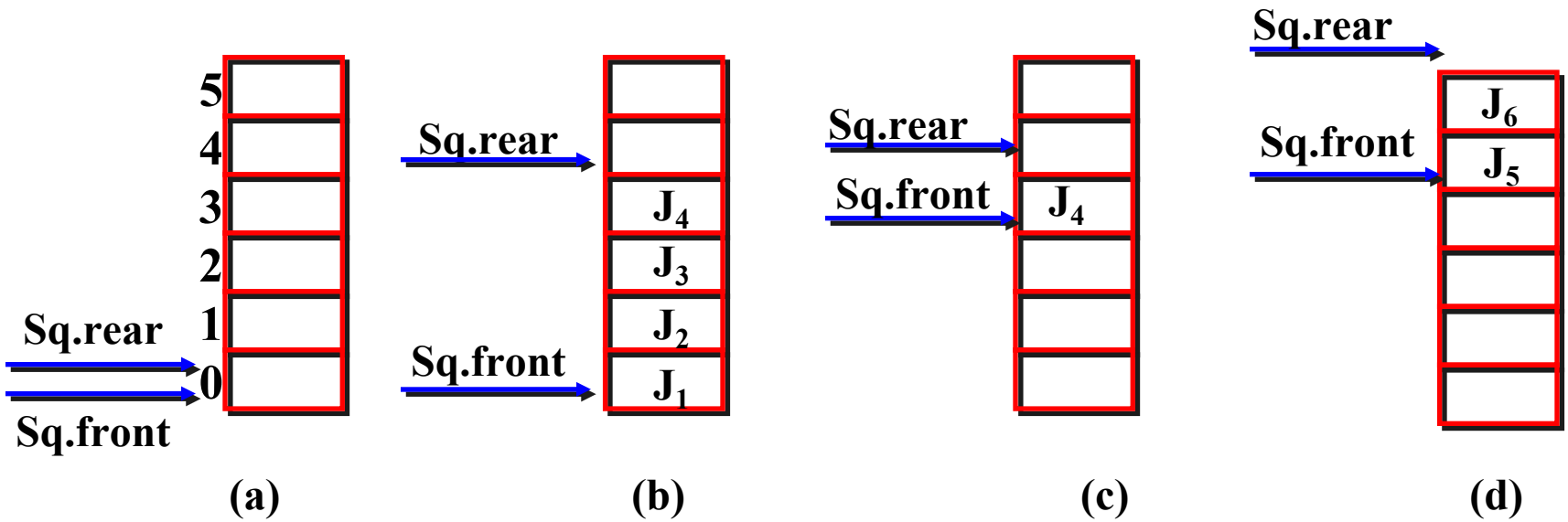
约定：尾指针指示队尾元素在队列中的下一个位置，头指针指示队列中队头元素的当前位置。

数据结构描述：

```
typedef struct {
    ElemType  *base;  // 动态分配存储空间
    int  front;    //头指针,若队不空,指向队头元素
    int  rear;     //尾指针,若队不空,指向队尾元素的下一位置
  } SqQueue;
```

3.4 队 列

3.4.3 队列的顺序表示和实现



由图可以看出：

队空：sq.front==sq.rear

队满：sq.rear==maxsize　　//可能是假溢出

入队：sq.elem[sq.rear]=x；　//x入队列

　　sq.rear++；　　//修改尾指针

出队：sq.front++；　　//修改头指针

3.4 队 列

如何解决顺序队列的假溢出问题？

可采取四种方法：

① 采用循环队列；

② 按最大可能的进队操作次数设置顺序队列的最大元素个数；

③ 修改出队算法，使每次出队列后都把队列中剩余数据元素向队头方向移动一个位置；

④ 修改入队算法，增加判断条件，当假溢出时，把队列中的数据元素向对头移动，然后方完成入队操作。

3.4 队 列

3.4.3 循环顺序队列

由图3.13(d)的**假满**状态可以看出时，**不可**再继续向队列**插入新的队尾元素**，但队列的**实际可用空间并未占满**。一个较巧妙的办法是将顺序队列构造为一个**环状的空间**。如图3.14所示，称之为**循环队列。**

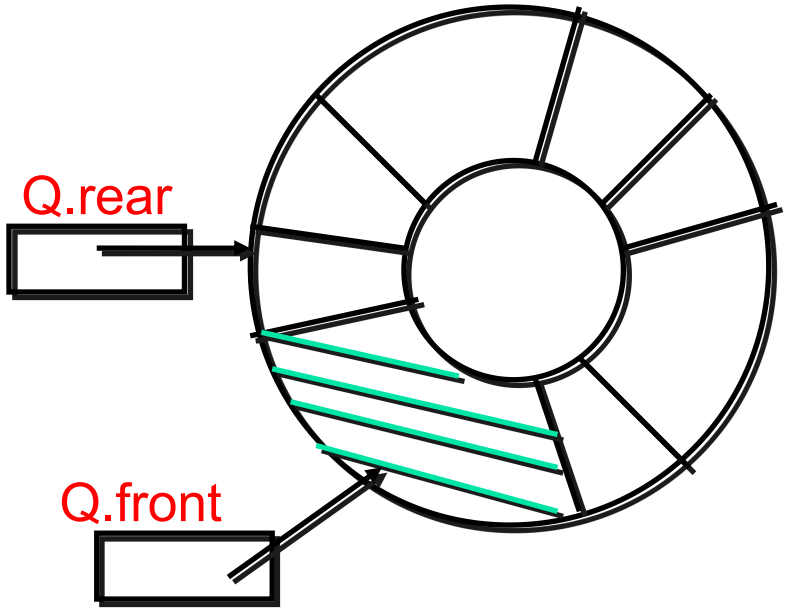
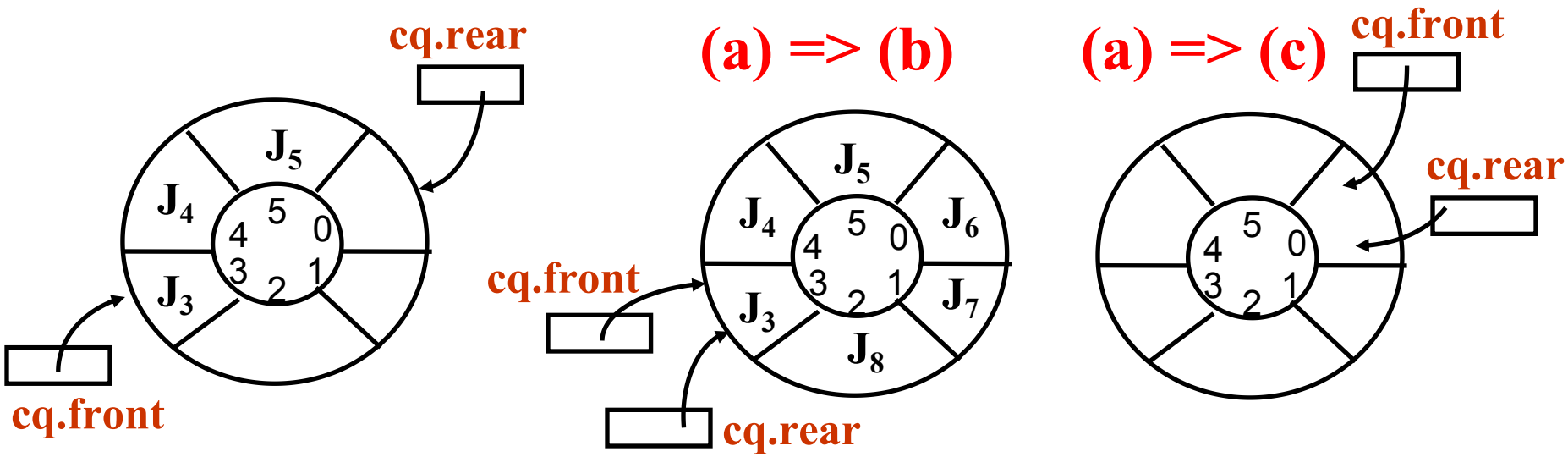


图3.14 循环队列示意图

3.4 队 列

3.4.3 循环顺序队列



（a）一般情况　　（b）队列满时　　（c）空队列

图3.15　循环队列的头尾指针

入队：cq.elem[cq.rear]=x；cq.rear=(cq.rear+1) % maxsize；
出队：cq.front=(cq.front+1) % maxsize；

3.4 队 列

3.4.3 循环顺序队列

由图3.15可以看出：

图3.15（b）队满：sq.front==sq.rear

图3.15（c）队空：sq.front==sq.rear

由此可见，只凭等式sq.front==sq.rear无法判别队列空间是队满？还是队空？

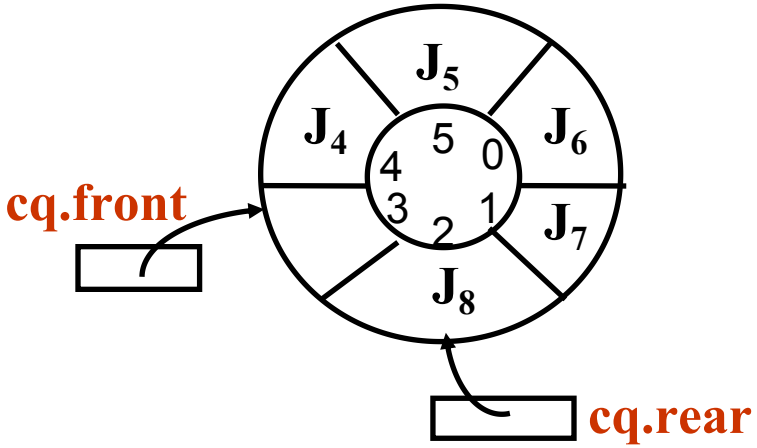
区分队满与队空的解决方法：

1）另设一标志位记录队列的状态；

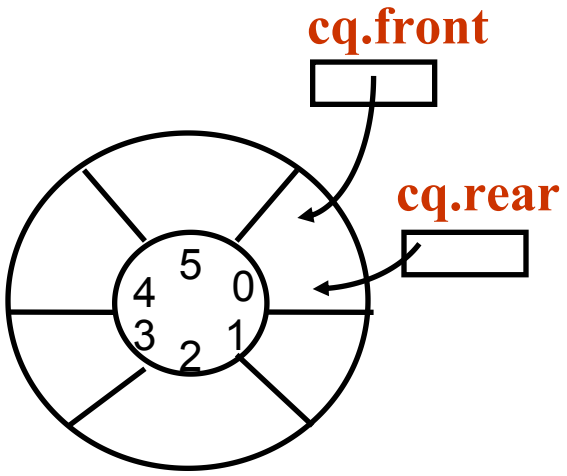
2）少用一个元素的空间（如图3.16所示）。

3.4 队 列

3.4.3 循环队列——队列的顺序表示和实现



（b）队列满时



（c）空队列

图3.16 区分队列“满”与“空”的头尾指针

队满：cq.front = = (cq.rear+1) % maxsize ；
队空：cq.front = = cq.rear ；

3.4 队 列

3.4.3 循环队列

- 循环队列的表示

```
//－－－－－－循环队列——队列的顺序存储结构－－－－－－

#define  MAXQSIZE  100      //最大队列长度
typedef  struct
{
        QElemType  *base; //初始化的动态分配存储空间
        int  front;   //头指针，若队列不空，指向队列头元素
        int  rear;   //尾指针，若队列不空，指向队尾元素的下一个位置
}SqQueue;
```

3.4 队 列

3.4.3 循环顺序队列

- 循环队列的实现

```
//－－－－－－循环队列的基本操作的算法描述－－－－－－
Status  InitQueue(SqQueue &Q)
{    Q.base=(QElemType*)
malloc(MAXQSIZE*sizeof(QElemType));
      if(!Q.base)    exit(OVERFLOW);     //存储分配失败
      Q.front = Q.rear = 0;
      return OK;
}
int  QueueLength(SqQueue Q)
{         //返回Q的元素个数，即队列的长度
      return (Q.rear-Q.front+MAXQSIZE)%MAXQSIZE;
}
```

3.4 队 列

3.4.3 循环顺序队列

- 循环队列的实现

```
//插入元素e为Q的新的队尾元素
Status  EnQueue(SqQueue &Q, QElemType e)
{
      if ( (Q.rear+1) % MAXQSIZE == Q.front)
            return  ERROR;//队列满
      Q.base[Q.rear] = e;
      Q.rear = (Q.rear+1) % MAXQSIZE ;
      return  OK;
}
```

3.4 队 列

3.4.3 循环顺序队列

- 循环队列的实现

```
//若队列不空，则删除Q的队头元素，用e返回其值，
并返回OK;//否则返回ERROR
Status  DeQueue(SqQueue &Q, QElemType &e)
{
      if (Q.rear == Q.front)
            return  ERROR;   //若队列为空，返回ERROR
      e = Q.base[Q.front];
      Q.front = (Q.front+1) % MAXQSIZE ;
      return  OK;
}
```

补充:队列的应用

例1：舞伴问题：舞会上，男士们和女士们进入舞厅时，各自排成一队。跳舞开始时，依次从男队和女队的队头上各出一人配成舞伴。若两队初始人数不相同，则较长的那一队中未配对者等待下一轮舞曲。 舞伴问题的类型描述：

```
typedef struct{
        char name[20];
         char sex;  //性别，'F'表示女性，'M'表示男性
    }Person;
typedef Person Elemtype;
//队列中元素的数据类型为Person
```

舞伴问题的算法：

```
void DancePartner(Person dancer[],int num)
 {//结构数组dancer中存放跳舞的男女，num是跳舞的人数。
      int i;
      Person p;
      SqQueue *Mdancers,*Fdancers;
      InitQueue(Mdancers);   //男士队列初始化
      InitQueue(Fdancers);   //女士队列初始化
      for( i=0; i<num; i++ )
     {//依次将跳舞者依其性别入队
        p=dancer[i];
        if(p.sex=='F')
              EnQueue(Fdancers.p);   //排入女队
        else
              EnQueue(Mdancers.p);   //排入男队
      }
```

```
printf("The dancing partners are: \n \n");
 while(!QueueEmpty(Fdancers)&&!QueueEmpty(Mdancers))
{       //依次输出男女舞伴名
        p=DeQueue(Fdancers);     //女士出队
        printf("%s ",p.name);          //打印出队女士名
        p=DeQueue(Mdancers);     //男士出队
        printf("%s\n",p.name);         //打印出队男士名
 }
  if( !QueueEmpty(Fdancers) )
{ //输出女士剩余人数及队头女士的名字
    printf("\n %d waiting in  next
round.\n",Fdancers.count);
    p=GetHead (Fdancers);      //取队头
    printf("%s will be the first to get a partner. \n",p.name);
}
 else
```

```
if(!QueueEmpty(Mdancers))
    {//输出男队剩余人数及队头者名字
          printf("\n There are%d men waiting
    for the next     round.\n",Mdacers.count);
          p=GetHead(Mdancers);
          printf("%s will be the first to get a
    partner.\n",p.name);
     }
  }
```

例2：采用队列求解迷宫问题

使用一个队列Qu记录走过的方块,该队列的结构如下：

```
typedef struct
{    int i,j;    /*方块的位置*/
     int pre;  /*本路径中上一方块在Qu中的下标*/
}ElemType;
```

这里使用的队列不是循环队列(因为要利用出队的元素找路径),因此在出队时,不会将出队元素真正从队列中删除,因为要利用它输出路径。

```
char mg[10][10]={       //表示迷宫的矩阵
   //' '表示可通过  '!'表示不通
   //0 1  2  3  4  5 6  7  8  9
   {'!','!','!','!','!','!','!','!','!','!'},  //0
   {'!',' ',' ',' ','!',' ',' ',' ',' ','!'},  //1
   {'!',' ',' ',' ','!',' ',' ',' ',' ','!'},  //2
   {'!',' ',' ',' ',' ',' ','!','!',' ','!'},  //3
   {'!',' ','!','!','!',' ',' ',' ',' ','!'},  //4
   {'!',' ',' ',' ','!',' ',' ',' ',' ','!'},  //5
   {'!',' ','!',' ',' ',' ','!',' ',' ','!'},  //6
   {'!',' ','!','!','!',' ','!','!',' ','!'},  //7
   {'!','!',' ',' ',' ',' ',' ',' ',' ','!'},  //8
   {'!','!','!','!','!','!','!','!','!','!'} };//9
```

搜索从(1, 1)到(8, 8)路径

(1) 首先将(1,1)入队;

(2) 在队列不为空时循环:出队一次(由于不是循环队列,该出队元素仍在队列中),称该出队的方块为当前方块,front为该方块在队列中的下标。

①如果当前方块是出口,则输出路径并结束。

②否则,按顺时针方向找出当前方块的四个方位中可走的相邻方块(对应的mg数组值为' '),将这些可走的相邻方块均插入到队列中,其pre设置为本搜索路径中上一方块在队列中的下标值,也就是当前方块的front值,并将相邻方块对应的mg数组值置为'*',以避免回过来重复搜索。

(3) 若队列为空仍未找到出口,即不存在路径。

实际上,本算法的思想是从(1,1)开始,利用队列的特点,一层一层向外扩展可走的点,直到找到出口为止,这个方法就是将在第7章介绍的图的广度优先搜索方法。

```
void mgpath()  /*搜索路径为:(1,1)->(8,8)*/
{       int i,j,find=0,di,k;
        SqQueue Q;
        InitQueue(Q,200);
        ElemType e,d;
        e.i=1;   e.j=1;   e.pre=-1;
        EnQueue(Q,e);              //(1,1)入队
        mg[1][1]='*';              //将其赋值'*',以避免回过来重复搜索
        while (!QueueEmpty(Q) && !find)
        {       k=Q.front;
                DeQueue(Q,e); //出队, 不是循环队列,该元素仍在队中
                i=e.i;  j=e.j;
                if (i==8 && j==8)       /*找到了出口,输出路径*/
                {
                        find=1;
                        cout<<"找到一条路径:"<<endl;
                        print(Q,e);                 //调用print函数输出路径
                        return;
                }
```

```
for (di=0;di<=3;di++)          //把每个可走的方块插入队列中
{       switch(di)
        {   case 0:     i=e.i;     j=e.j+1;   break;     //向东
            case 1:     i=e.i+1;   j=e.j;     break;     //向南
            case 2:     i=e.i;     j=e.j-1;   break;     //向西
            case 3:     i=e.i-1;   j=e.j;     break;     //向北
        }
        if (mg[i][j]==' ')
            {   /*将该相邻方块插入到队列中*/
                d.i=i;          d.j=j;          d.pre=k;
                EnQueue(Q,d);
                mg[i][j]='*';   //将其赋值-1,以避免重复搜索
            }
 }    }
   if (!find) printf("不存在路径!\n");
 }
```

本章小结

线性表、栈、队的异同点：

相同点： 逻辑结构相同，都是线性的；都可以用顺序存储或链表存储；栈和队列是两种特殊的线性表，即受限的线性表（只是对插入、删除运算加以限制）。

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、操作系统作业调度和简化设计等。

第3章结束