

第3章 栈和队列

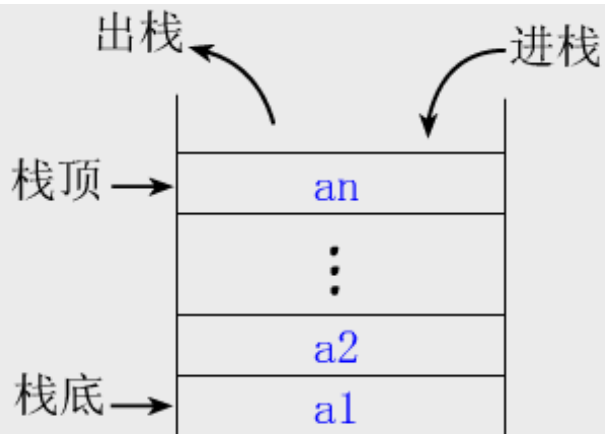
教学目标

1. 掌握栈和队列的**特点**，并能在相应的应用问题中正确选用
2. 熟练掌握栈的**两种存储结构**的基本操作实现算法，特别注意**栈满和栈空**的条件
3. 熟练掌握**循环队列和链队列**的基本操作实现算法，特别注意**队满和队空**的条件
4. 理解**栈与递归以及栈和队列的应用**

3.1 栈和队列的定义和特点

栈 (Stack)

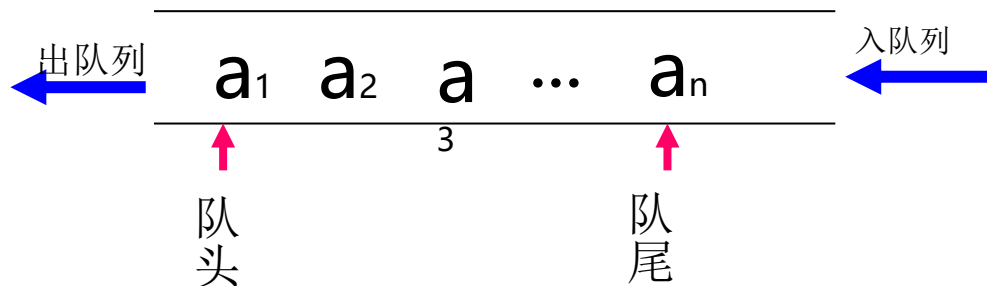
1. 定义
2. 逻辑结构
3. 存储结构
4. 运算规则
5. 实现方式



栈示意图

队列 (Queue)

1. 定义
2. 逻辑结构
3. 存储结构
4. 运算规则
5. 实现方式



3.1.1 栈的定义和特点

- 1. 定义 只能在表尾（栈顶）进行插入和删除运算的线性表
- 2. 逻辑结构 与线性表相同，仍为一对一关系
- 3. 存储结构 用顺序栈或链栈存储均可，但以顺序栈更常见

4.运算规则

只能在**栈顶**运算，且访问结点时依照**后进先出（LIFO）**或**先进后出（FILO）**的原则

5.实现方式

关键是编写**入栈**和**出栈**函数，具体实现依顺序栈或链栈的不同而不同

基本操作有**入栈**、**出栈**、**读栈顶元素值**、**建栈**、**判断栈满**、**栈空**等



“进”: **PUSH()**
“出”: **POP()**

栈与一般线性表的区别

栈是一种特殊的线性表，它只能在表的一端（**栈顶**）进行插入和删除运算

栈与一般线性表的区别：仅在于**运算规则**不同

一般线性表

逻辑结构：一对一

存储结构：顺序表、链表

运算规则：随机、顺序存取

插入和删除
位置随意

栈

逻辑结构：一对一

存储结构：顺序**栈**、链**栈**

运算规则：**后进先出**

插入和删除只允许在
表的一端（栈顶）

“进”：压入 **PUSH()**

“出”：弹出 **POP()**

3.1.2 队列的定义和特点

1. 定义 只允许在表的一端（队尾）进行插入，而在另一端（队头）删除元素的线性表。
2. 逻辑结构 与线性表相同，仍为一对一关系
3. 存储结构 用顺序或链式存储均可。

4.运算规则

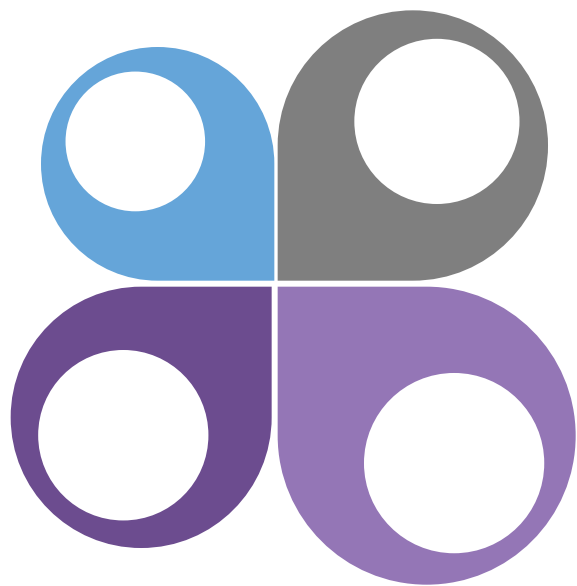
先进先出（**FIFO**）的原则

5.实现方式

关键是编写**进队**和**出队**函数，具体实现依顺序队列或链式队列的不同而不同

基本操作有**入队、出队、取对头元素、判断队满或空等**

3.2 案例引入



案例3.1：数制的转换



案例3.2：括号匹配的检验



案例3.3：表达式求值



案例3.4：舞伴问题

3.3 栈的表示和实现

3.3.1 栈的类型定义

ADT Stack {

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

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

约定 a_n 端为栈顶, a_1 端为栈底

基本操作:

InitStack(&S) // 构造一个空栈S

DestroyStack(&S) //销毁栈S

ClearStack (&S) //清空, S为空栈

StackEmpty (S) //判断栈S是否为空, 空为true

StackLength(S) //求栈S的长度, 即元素个数

Push(&S, e) // 元素e进栈

Pop(&S, &e) //元素出栈, 用e返回

GetTop(S,&e) //用e返回S的栈顶元素, 不出栈

StackTraverse(S) //遍历栈, 从栈底到栈顶

} ADT Stack

3.3.2 顺序栈的表示和实现

用一组地址连续的存储单元（动态申请，用**指针base**指向起始地址）依次存放自栈底到栈顶的数据元素，同时设立**指针top**指向栈顶元素在顺序栈中的位置。

顺序栈的动态分配定义：

```
#define MAXSIZE 100
```

```
typedef struct {  
    SElemType *base;
```

//栈底指针，数组名

```
    SElemType *top;
```

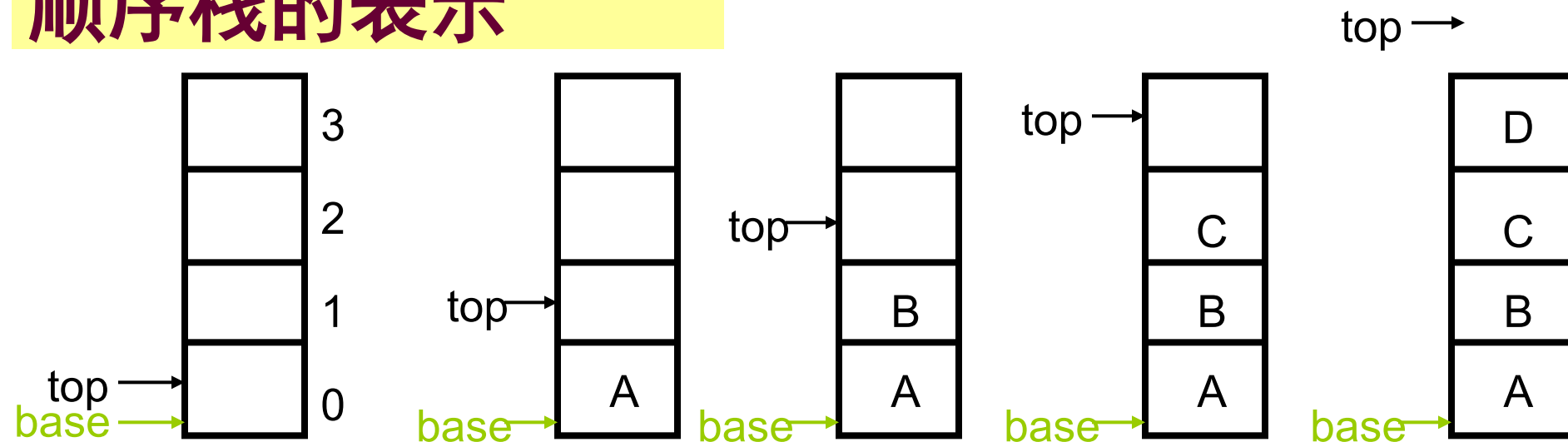
//栈顶指针，或 int top;

```
    int stacksize;
```

//栈的最大容量

```
}SqStack;
```

顺序栈的表示



空栈

base == top

是栈空标志

stacksize = 4

Base: 空间起始地址(固定不动)。

Top: 指示栈顶元素之上的位置,
即是下一个元素进栈的地址

栈满时的处理方法:

1.报错 2.扩大存储空间容量。

顺序栈初始化

```
Status InitStack( SqStack &S )
```

```
{
```

```
    S.base =(SElemType*)malloc  
            (MAXSIZE*sizeof(SElemType)); //申请空间
```

```
    if( !S.base )    exit(OVERFLOW);
```

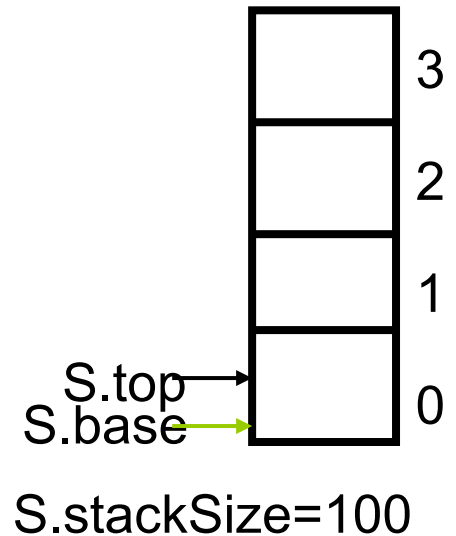
```
    S.top = S.base;    //S.top=0;
```

```
    S.stackSize = MAXSIZE;
```

```
    return OK;
```

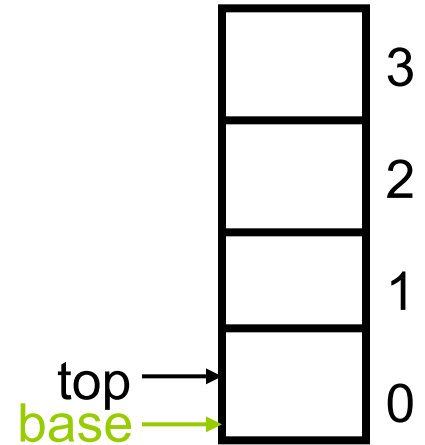
```
}
```

```
typedef struct{  
    SElemType *base;  
    SElemType *top; //或者 int top;  
    int stacksize;  
}SqStack;
```



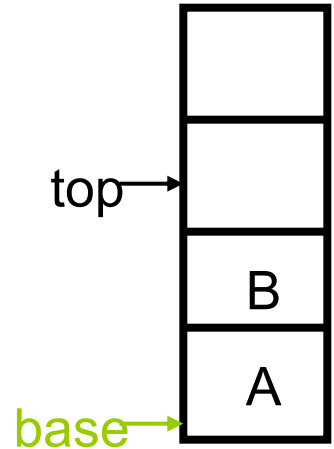
判断顺序栈是否为空

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



求顺序栈的长度

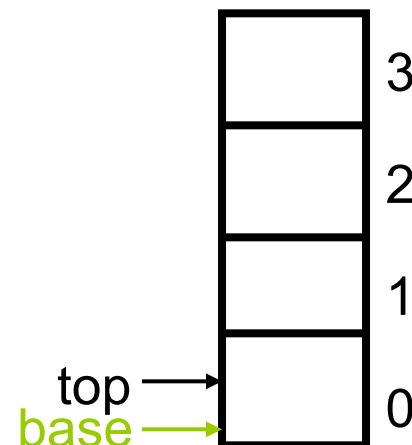
```
int StackLength( SqStack S )  
{  
    return S.top – S.base;  
}
```



if(S.top-S.base==S.stacksize)
则栈满

清空顺序栈

```
Status ClearStack( SqStack S )  
{ //将栈s置为空栈  
    S.top = S.base;  
    return OK;  
}
```

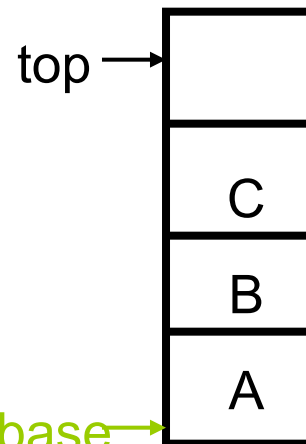


销毁顺序栈

```
Status DestroyStack( SqStack &S )
{
    if( S.base )
    {
        free(S.base) ;
        S.stacksize = 0;
        S.base = S.top = NULL;
    }
    return OK;
}
```

顺序栈进栈 *

- (1) 判断是否栈满，若满则报错
- (2) 元素 e 压入栈顶
- (3) 栈顶指针加1

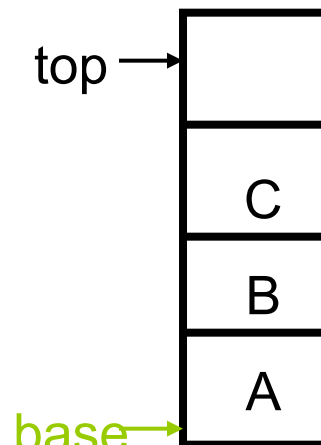


```
Status Push( SqStack &S, SElemType e)
{
    if( S.top - S.base == S.stacksize ) // 栈满
        return ERROR;
    *S.top++ = e;
    return OK;
}
```

***S.top = e;**
S.top++;

顺序栈出栈 *

- (1) 判断是否栈空，若空则出错
- (2) 栈顶指针减1
- (3) 获取栈顶元素e



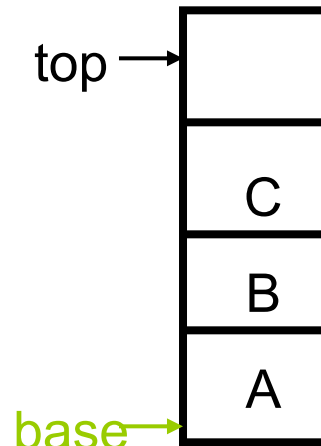
Status Pop(SqStack &S, SElemType &e)

```
{  
    if( S.top == S.base ) // 栈空  
        return ERROR;  
    e = *--S.top;  
    return OK;  
}
```

--S.top;
e=*S.top;

取顺序栈栈顶元素

- (1) 判断是否空栈，若空则返回错误
- (2) 否则通过栈顶指针获取栈顶元素

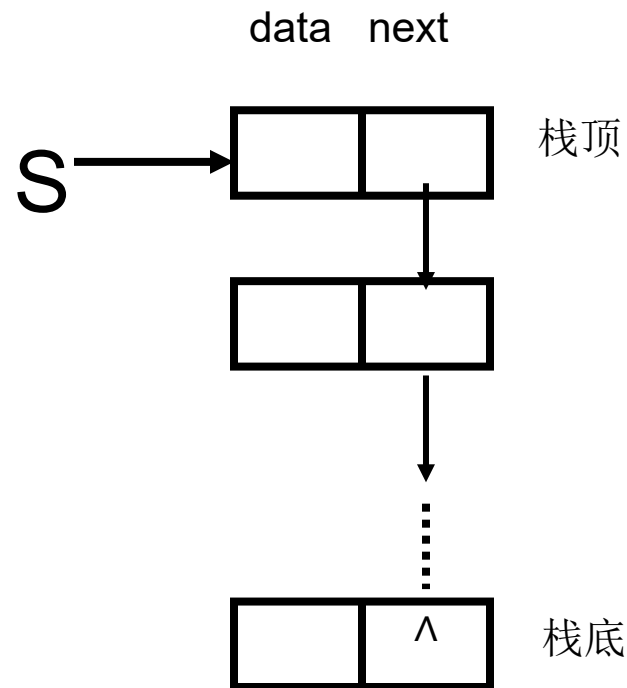


```
Status GetTop( SqStack S, SElemType &e)
{
    if( S.top == S.base ) return ERROR; // 栈空
    e = *( S.top - 1 ); //元素不出栈，故top不变
    return OK;
}
```

3.3.3 链栈的表示和实现

- ✓ 运算是受限的单链表，链表头部作为栈顶，没有必要附加头结点。栈顶指针就是链表的头指针

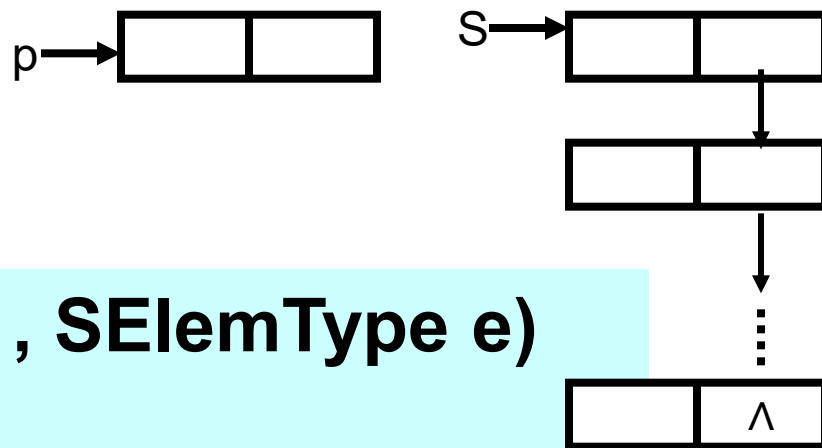
```
typedef struct StackNode {  
    SElemType data;  
    struct StackNode *next;  
} StackNode, *LinkStack;  
LinkStack S;
```



思考：

为什么把头指针所在的一端作为栈顶？

链栈进栈



```
Status Push(LinkStack &S , SElemType e)
```

```
{
```

```
    p=new StackNode;    //生成新结点p
```

```
    if (!p) return OVERFLOW;
```

```
    p->data=e;
```

```
    p->next=S;
```

```
    S=p;
```

```
    return OK;
```

```
}
```

✓ 实际就是在单链表表头插入一个元素。

链栈出栈

$e = 'A'$

```
Status Pop (LinkStack &S, SElemType &e)
```

```
{
```

```
    if (S==NULL) return ERROR;
```

```
     $e = S \rightarrow data;$  //栈顶元素值有e带回
```

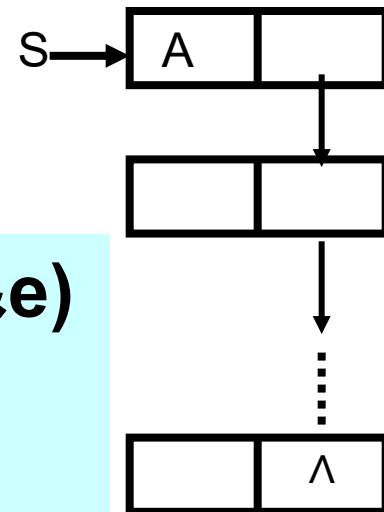
```
    p = S;
```

```
    S = S->next;
```

```
    free(p); //删除栈顶结点
```

```
    return OK;
```

```
}
```



3.4 栈的应用

例1：数制转换（十转N）----栈与递归

用栈暂存低位值

例2：括号匹配的检验

用栈暂存左括号

例3：表达式求值

用栈暂存运算符

1、数制转换

把一个十进制数转换成其他进制数（2~9进制）。

$$\begin{array}{r|l} 8 & 1348 \\ \hline & 4 \\ 8 & 168 \\ \hline & 0 \\ 8 & 21 \\ \hline & 5 \\ 8 & 2 \\ \hline & 2 \\ & 0 \end{array}$$

$$(1348)_{10} = (2504)_8$$

```

void conversion (int n ) {
    //递归算法
    if(n<8)
        printf("%d",n);
    else {
        conversion(n/8);
        printf("%d",n%8);
    }
} //

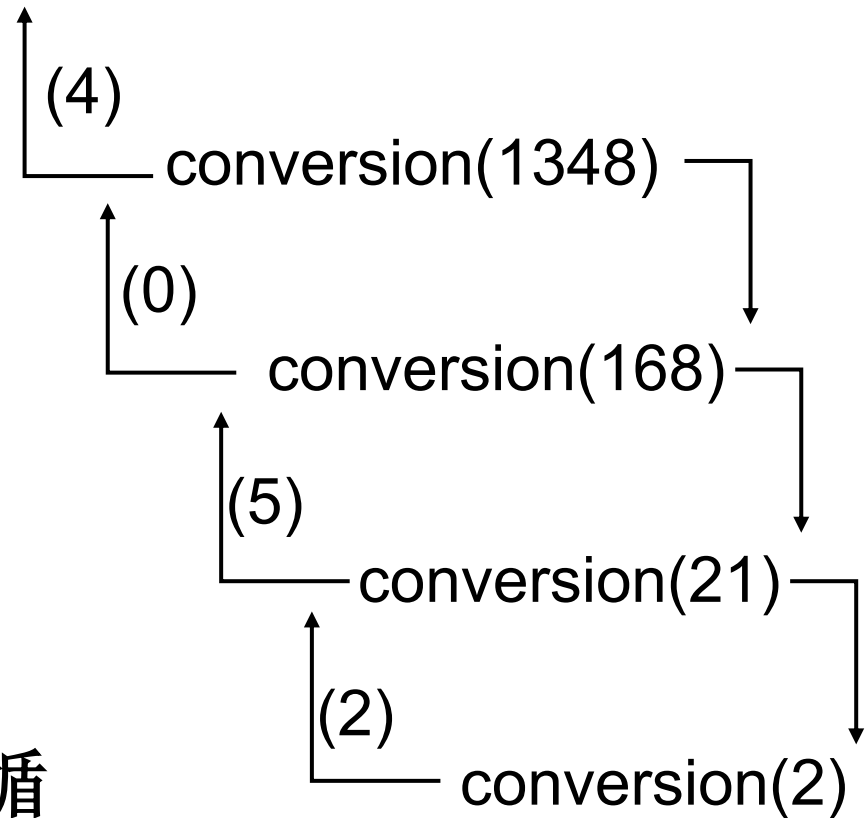
```

函数构成嵌套调用时, 遵循

后调用先返回



栈




```
void conversion ( ) {  
    InitStack (S);          //构造一个空栈S  
    scanf ( “%d” , N);  
    while (N) {  
        Push ( S , N % 8 ); // 取余 入栈  
        N = N / 8;          // 整除  
    }  
    While ( ! StackEmpty (S) ) {  
        Pop ( S , e );      // 出栈  
        printf ( “%d” , e );  
    }  
} //
```

递归→非递归

1. 单路递归→循环结构

```
long Fact ( long n ) {  
    if ( n == 0) return 1;  
    else return n * Fact (n-1); }
```

```
long Fact ( long n ) {  
    t=1;  
    for(i=1; i<=n; i++)    t=t*i;  
    return t; }
```

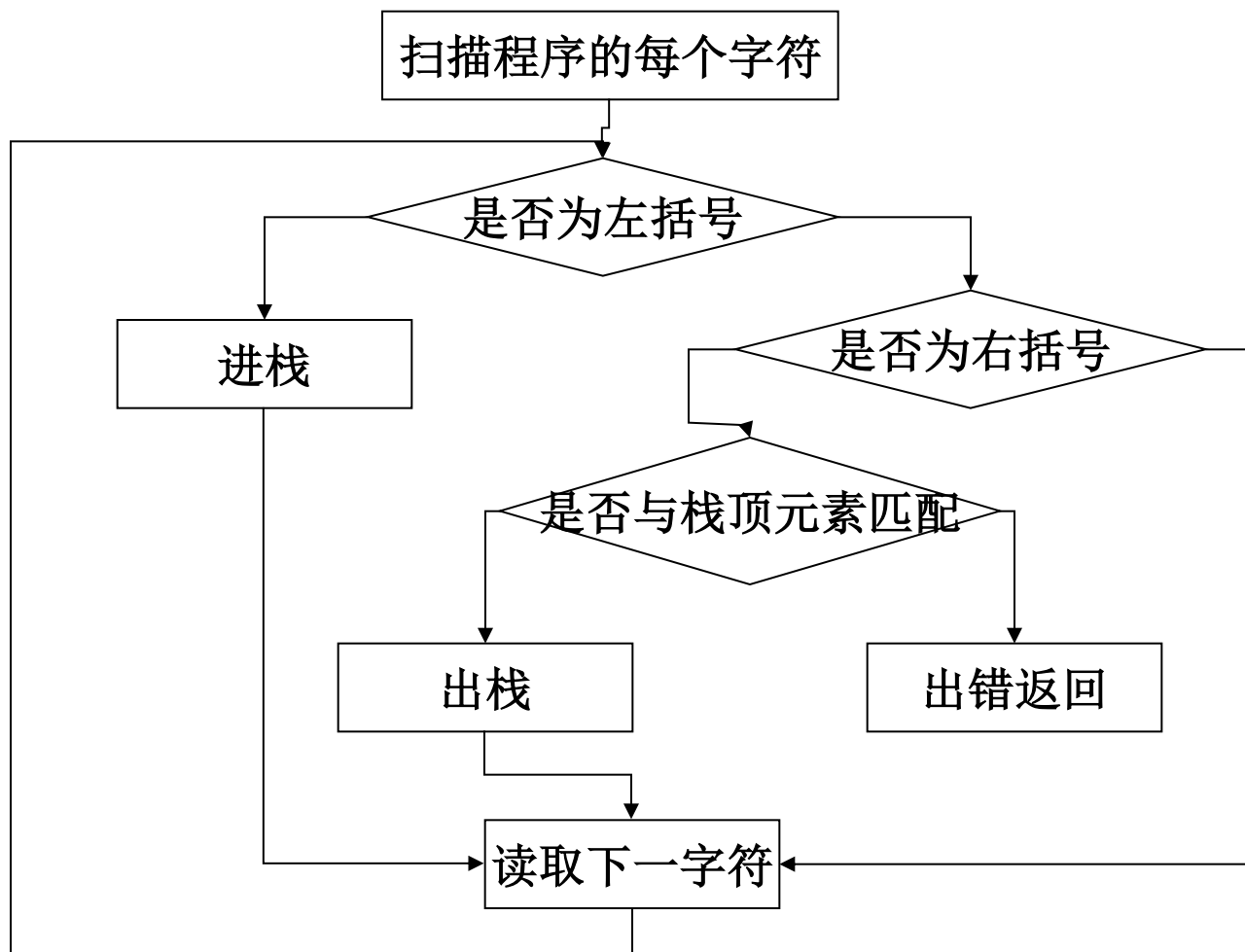


2. 自用栈模拟系统的运行时栈

如汉诺塔、二叉树遍历

2、括号匹配的检验

常用在计算机语言的编译过程中进行语法检查。其中一个就需检查程序中的大括号、方括号、圆括号是否配对。



Status Matching() {

//检验表达式中括号是否正确匹配，匹配返回true，否则false。表达式以“#”结束

int flag=1; //标记查找结果以控制循环及返回结果

InitStack(S); char c; cin>>c;

while(c!='#' && flag) {

switch (c) {

case '[': case '(': //若是左括号，则将其压入栈

Push(S,c); break;

case ')': //若是右括号“）”，则根据当前栈顶元素的值分情况考

if (!StackEmpty(S) && gettop(S)=='(') //若栈非空且栈顶元素是“（”

Pop(S,x);

else flag=0; //若栈空或栈顶元素不是“（”，则非法

break;

case ']': //若是右括号“】”，则根据当前栈顶元素的值分情况考虑

if (!StackEmpty(S) && GetTop(S)=='[')//若栈顶元素是“[”，则成功

Pop(S,x);

else flag=0; //若栈空或栈顶元素不是“[”，则非法

break;

//switch

cin>>c;

//继续读入下一个字符

//while

if (StackEmpty(S) &&flag) return true;

else return false;

} //Matching

3、表达式求值

算术四则运算规则

- (1) 先乘除,后加减
- (2) 从左算到右
- (3) 先括号内,后括号外

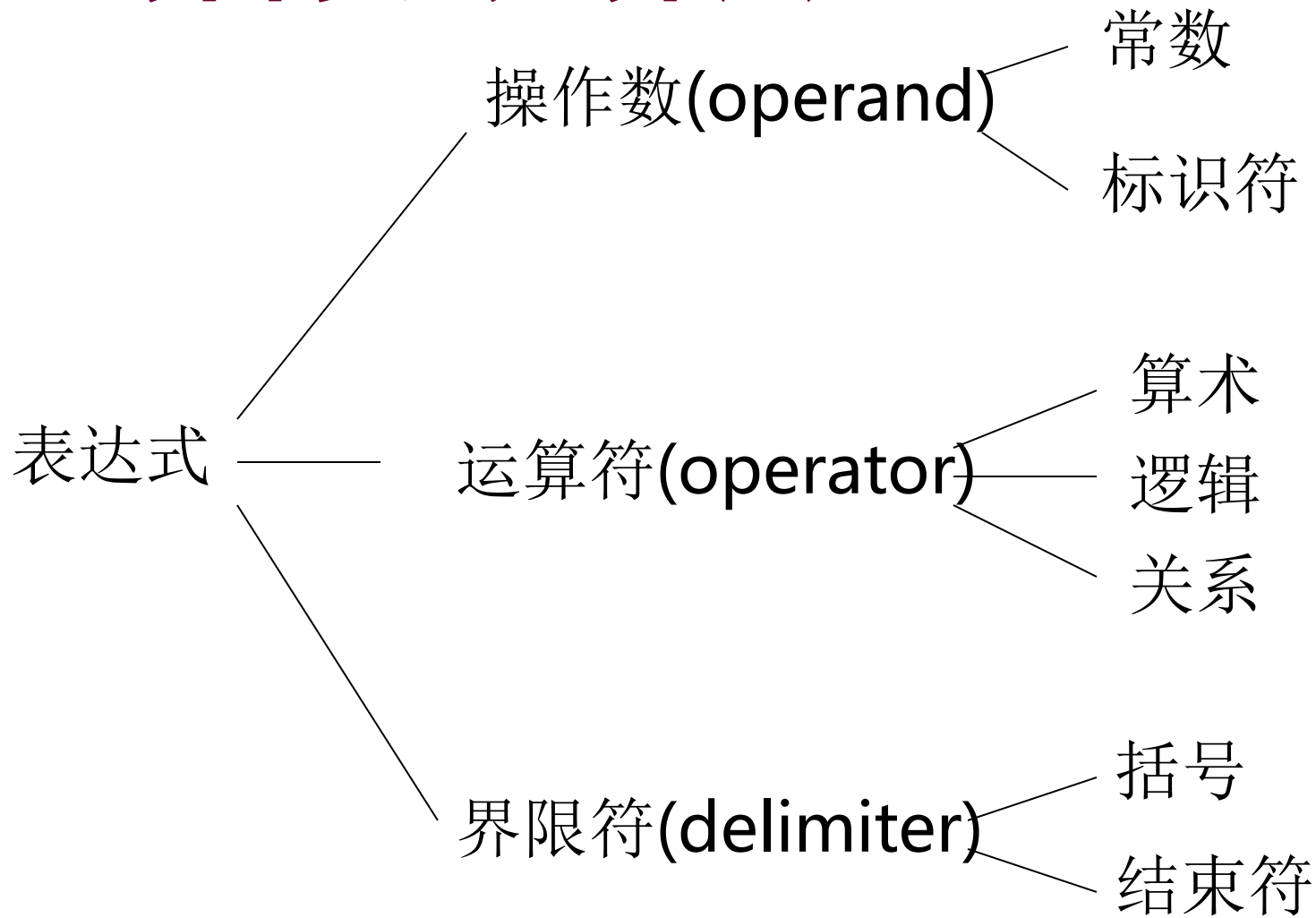
➤一般地，表达式都由操作数、运算符、界限符组成。

➤根据运算优先关系来实现对表达式的编译或解释。

称为算符优先算法

➤算符：运算符与界限符的总称。

➤算符优先算法



• 算符间的优先关系

算符的集合称为OP，对任意两个相继出现的算符 θ_1 、 θ_2 ，其优先关系必然存在以下关系：

$$\theta_1 < \theta_2 \text{ OR } \theta_1 = \theta_2 \text{ OR } \theta_1 > \theta_2$$

表3.1 算符间的优先关系

| $\theta_1 \backslash \theta_2$ | + | - | * | / | (|) | # |
|--------------------------------|---|---|---|---|---|---|---|
| + | > | > | < | < | < | > | > |
| - | > | > | < | < | < | > | > |
| * | > | > | > | > | < | > | > |
| / | > | > | > | > | < | > | > |
| (| < | < | < | < | < | = | |
|) | > | > | > | > | | > | > |
| # | < | < | < | < | < | | = |

【算法思想】

设定两栈：**OPND**-----操作数或运算结果 **OPTR**-----运算符

- (1) 初始化**OPTR**栈和**OPND**栈，将“**#**”压入**OPTR**
- (2) 依次读入字符**ch**，循环执行(3)至(5)
- (3) 取出**OPTR**的栈顶元素，当**OPTR**的栈顶元素和**ch**均为“**#**”时，表达式求值完毕，**OPND**栈顶元素为表达式的值
- (4) **if** (**ch**是操作数) 则**ch**进**OPND**，读入下一字符**ch**
- (5) **else** 比较**OPTR**栈顶元素和**ch**的优先级

case ‘<’: 运算符**ch** 进**OPTR**，读入下一字符**ch**

case ‘>’: 栈顶运算符退栈、计算，结果进**OPND**

case ‘=’: 脱括号（弹出左括号），读入下一字符**ch**

模拟： $2 + 3 * (1 + 4) \#$

```

OperandType EvaluateExpression() {
    InitStack (OPTR); Push(OPTR,'#');
    InitStack (OPND); ch = getchar( );
    while (ch!= '#' || GetTop(OPTR)!= '#') {
        if (! In(ch,OP)){ // ch不是运算符则进栈
            Push(OPND,ch); ch = getchar(); }
        else
            switch (Precede(GetTop(OPTR),ch)) { //比较优先权
                case '<': //当前字符ch压入OPTR栈，读入下一字符ch
                    Push(OPTR, ch); ch = getchar(); break;
                case '>': //弹出OPTR的运算符并运算，并将结果入栈
                    Pop(OPTR, theta);
                    Pop(OPND, b); Pop(OPND, a);
                    Push(OPND, Operate(a, theta, b)); break;
                case '=': //脱括号并接收下一字符
                    Pop(OPTR,x); ch = getchar();
                    break;
            } // switch
    } // while
    return GetTop(OPND);
} // EvaluateExpression

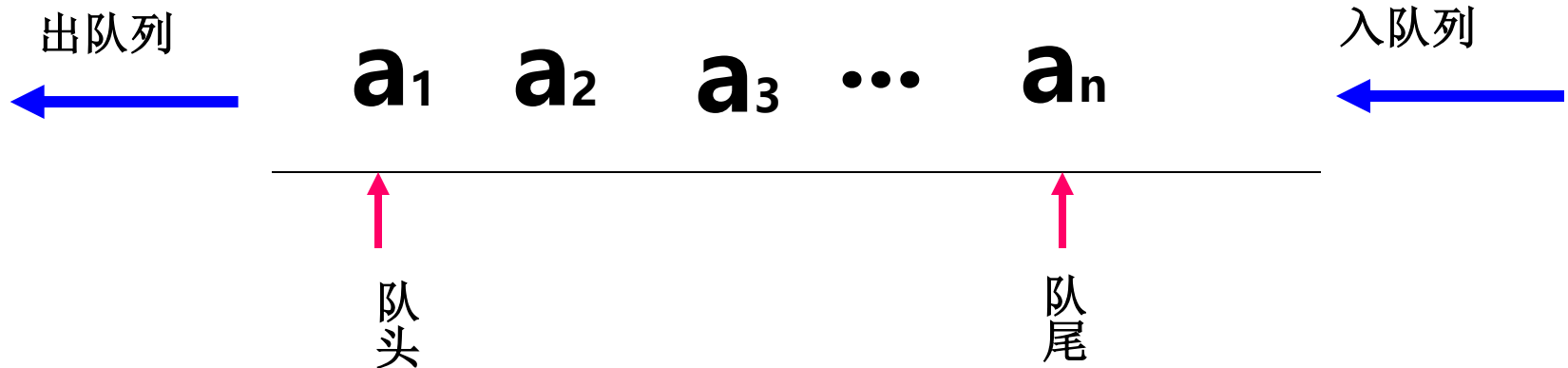
```

| OPTR | OPND | INPUT | OPERATE |
|---------|-------|----------|----------------|
| # | | 3*(7-2)# | Push(opnd,'3') |
| # | 3 | *(7-2)# | Push(optr,'*') |
| #,* | 3 | (7-2)# | Push(optr,'(') |
| #,*,(| 3 | 7-2)# | Push(opnd,'7') |
| #,*,(| 3,7 | -2)# | Push(optr,'-') |
| #,*,(,— | 3,7 | 2)# | Push(opnd,'2') |
| #,*,(,— | 3,7,2 |)# | Operate(7-2) |
| #,*,(| 3,5 |)# | Pop(optr) |
| #,* | 3,5 | # | Operate(3*5) |
| # | 15 | # | GetTop(opnd) |

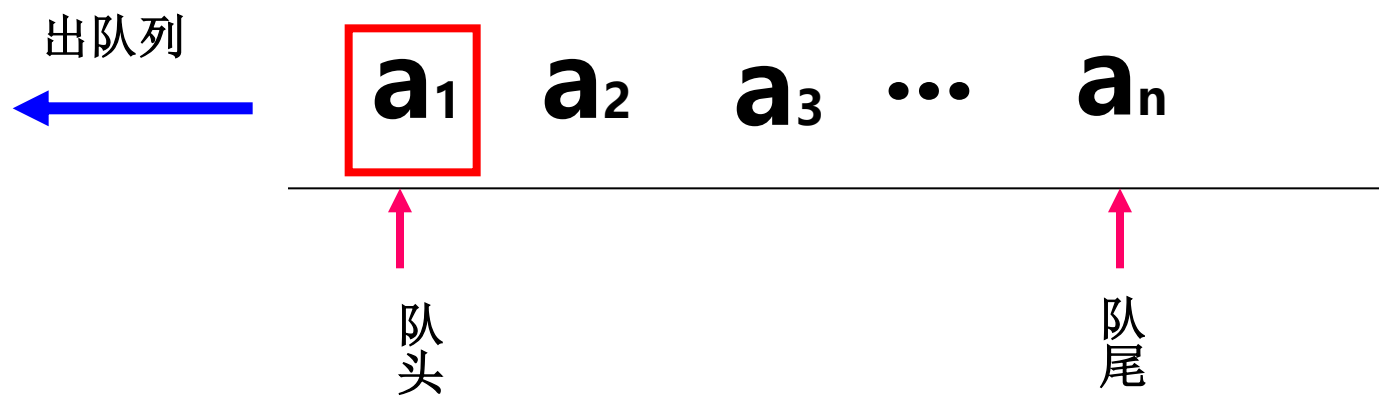
3.5 队列的表示和实现

队列是一种**先进先出(FIFO)**的线性表. 它只允许在表的一端进行插入,而在另一端删除元素

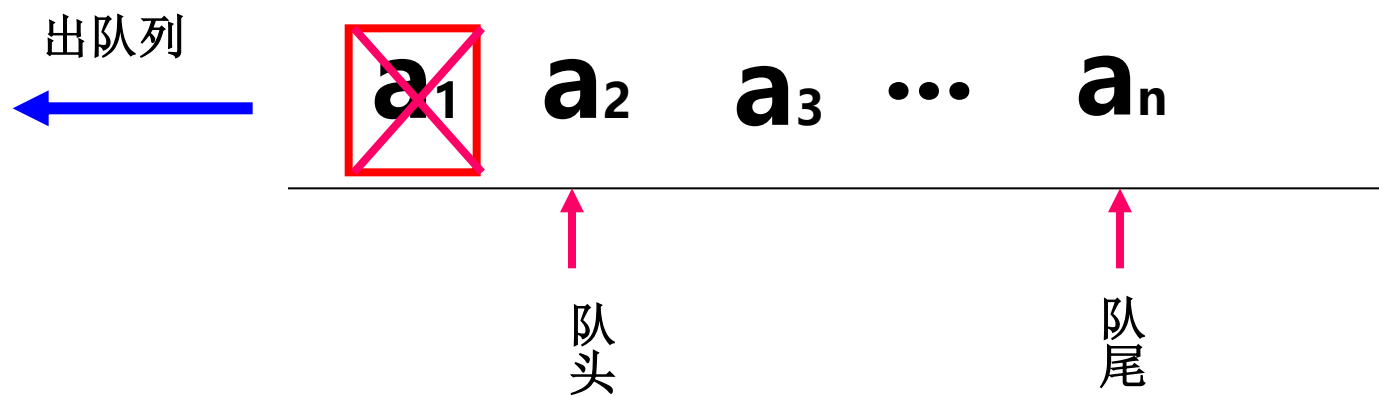
$$q = (a_1, a_2, \dots, a_n)$$



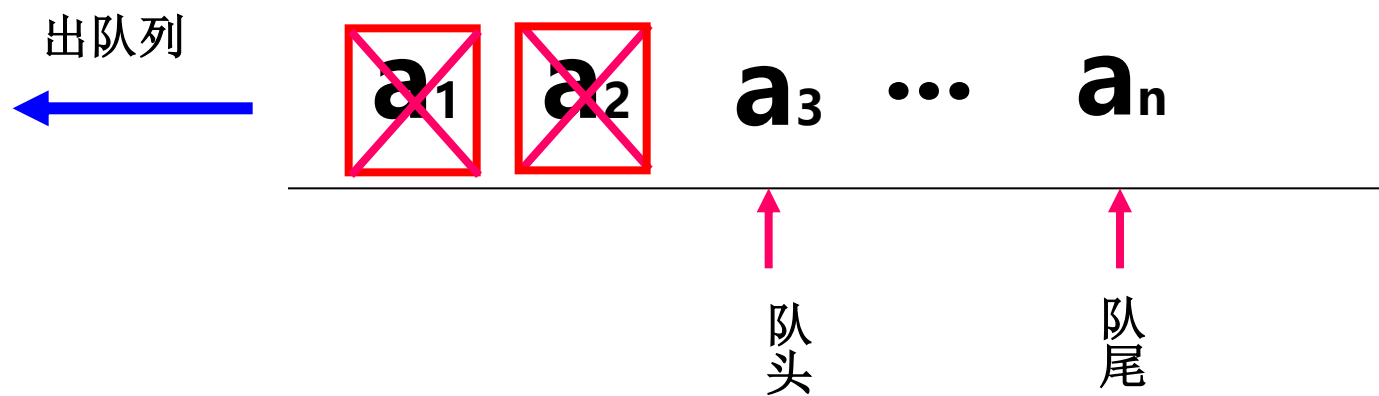
$$q = (a_1, a_2, \cdots a_n)$$



$$q = (a_1, a_2, \cdots a_n)$$



$$q = (a_1, a_2, \cdots a_n)$$



3.5.1 队列的抽象数据类型

ADT Queue {

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

数据关系: $R = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i=1,2,\dots,n \}$ //a1端为队头

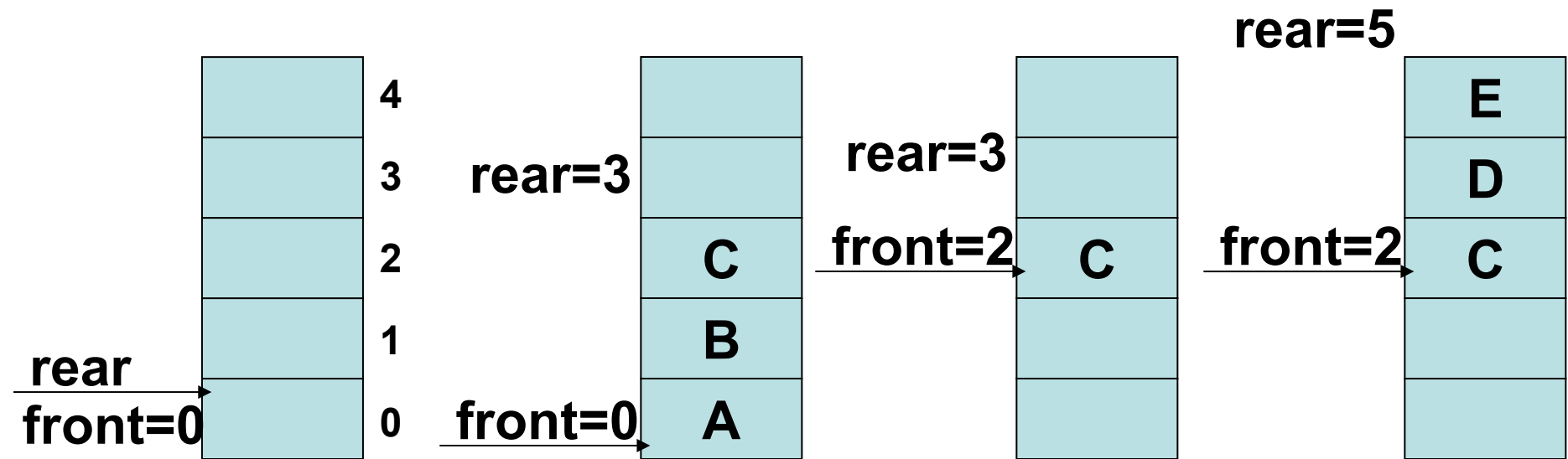
基本操作:

| | |
|--------------------------------|----------------|
| InitQueue (&Q) | //构造空队列 |
| DestroyQueue (&Q) | //销毁队列 |
| ClearQueue (&Q) | //清空队列 |
| QueueEmpty(Q) | //判空. 空--TRUE, |
| QueueLength(Q) | //取队列长度 |
| GetHead (Q,&e) | //取队头元素, |
| EnQueue (&Q,e) | //入队列 |
| DeQueue (&Q,&e) | //出队列 |
| QueueTraverse(Q) | //遍历 |

} ADT Queue

3.5.2 队列的顺序表示和实现——循环队列

利用一组地址连续的存储单元依次存放自队头到队尾的数据元素，同时设立front和rear两个指示器（int变量）分别指向队头元素的位置和队尾元素的下一个位置（数组下标）。



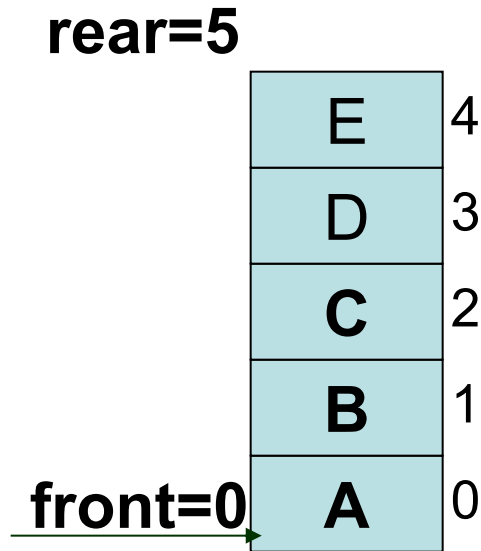
$front=rear=0$

空队标志: $front==rear$
入队: $base[rear++]=x;$
出队: $x=base[front++];$

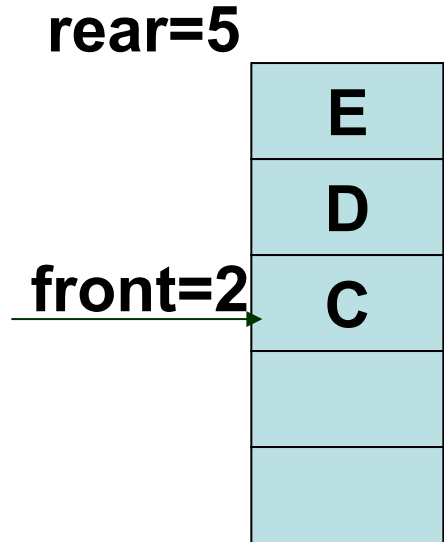
存在的问题

设数组大小为 $M=5$

入队: $\text{base}[\text{rear}++] = x;$

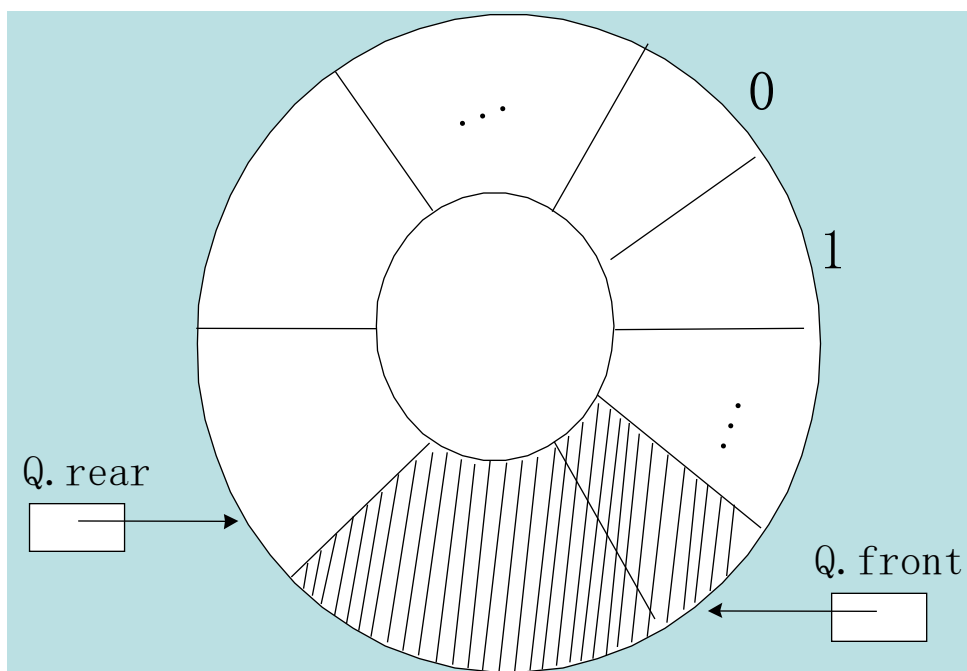


front==0
rear==M 时
再入队——真溢出



front != 0
rear==M 时
再入队——假溢出

解决的方法——循环队列



base[0]接在base[M-1]之后
若 $\text{rear}+1==M$
则令 $\text{rear}=0$;

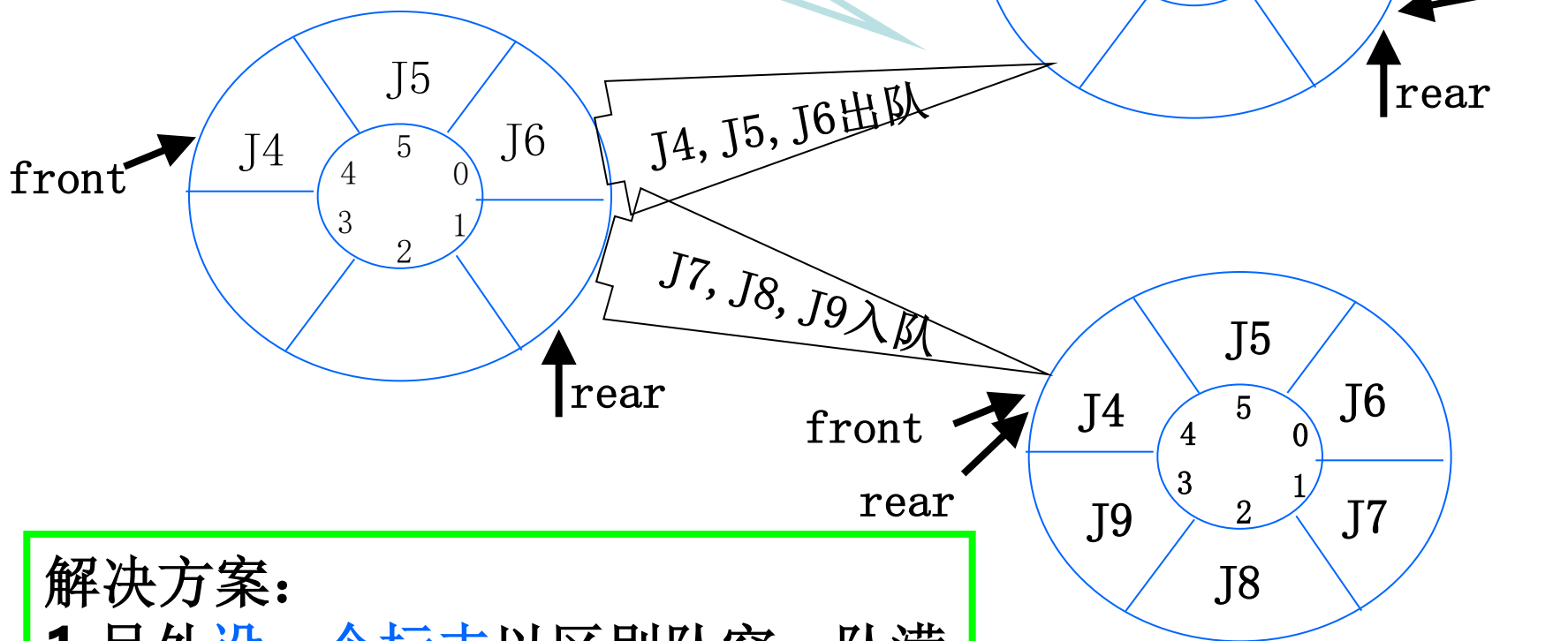
实现：利用“模”运算
入队：

$\text{base}[\text{rear}]=x;$
 $\text{rear}=(\text{rear}+1)\%M;$

出队：

$x=\text{base}[\text{front}];$
 $\text{front}=(\text{front}+1)\%M;$

队空: $front == rear$
队满: $front == rear$



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:

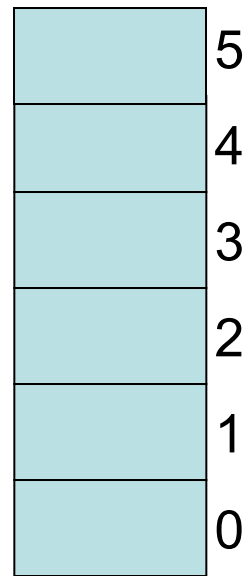
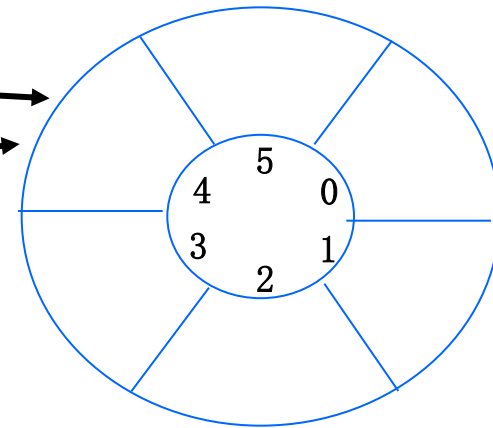
队空: $front == rear$

队满: $(rear + 1) \% M == front$

front==4

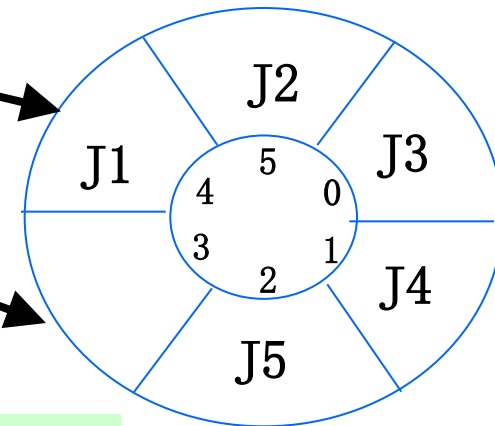
rear==4

队空: $\text{front} == \text{rear}$



front==4

rear==3



队满: $(\text{rear} + 1) \% M == \text{front}$
此时, rear所指单元不能使用

循环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;        //头指示器
```

```
    int rear;         //尾指示器
```

```
}SqQueue;
```

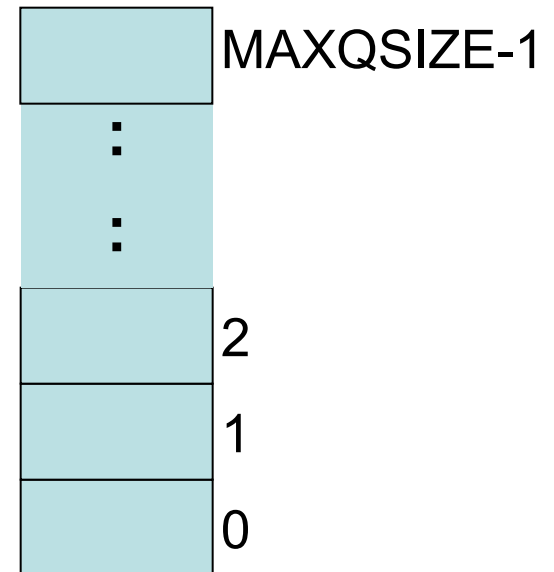

循环队列初始化

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

Q.front=0

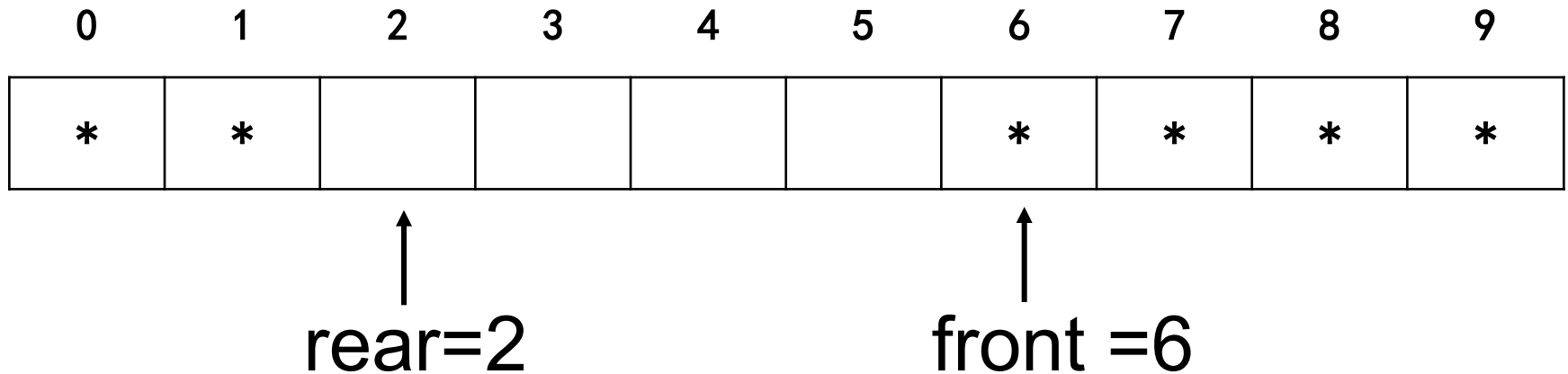
Q.rear=0

Q.base →



求队列的长度

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



元素入队

```
Status EnQueue(SqQueue &Q, QElemType e){  
    if((Q.rear+1)%MAXQSIZE==Q.front) //队列满  
        return ERROR;  
  
    Q.base[Q.rear]=e;           //新元素插入队尾  
    Q.rear=(Q.rear+1)%MAXQSIZE; //修改队尾指针  
    return OK;  
}
```

元素出队

```
Status DeQueue (SqQueue &Q, QElemType &e){  
    if( Q.front == Q.rear )    //队列为空  
        return ERROR;  
    e=Q.base[Q.front];        //取出队首元素  
    Q.front=(Q.front+1)%MAXQSIZE; //修改队头指针  
    return OK;  
}
```

取队头元素，不出队

```
Status GetHead (SqQueue Q, QElemType &e){  
    if( Q.front == Q.rear )    //队列为空  
        return ERROR;  
    e=Q.base[Q.front];        //取出队首元素  
    return OK;  
}
```

顺序循环队列特别注意：

1. **front**所指的是队头元素所在的位置，**rear**所指的是队尾元素的下一个位置，即下次元素进队的位置。

2. 空队列条件：

$$\mathbf{Q.ront == Q.rear}$$

3. 满队列条件：

$$\mathbf{(Q.rear + 1) \% MAXQSIZE == Q.front}$$

4. 入队的变化：

$$\mathbf{Q.rear = (Q.rear + 1) \% MAXQSIZE}$$

5. 出队的变化

$$\mathbf{Q.front = (Q.front + 1) \% MAXQSIZE}$$

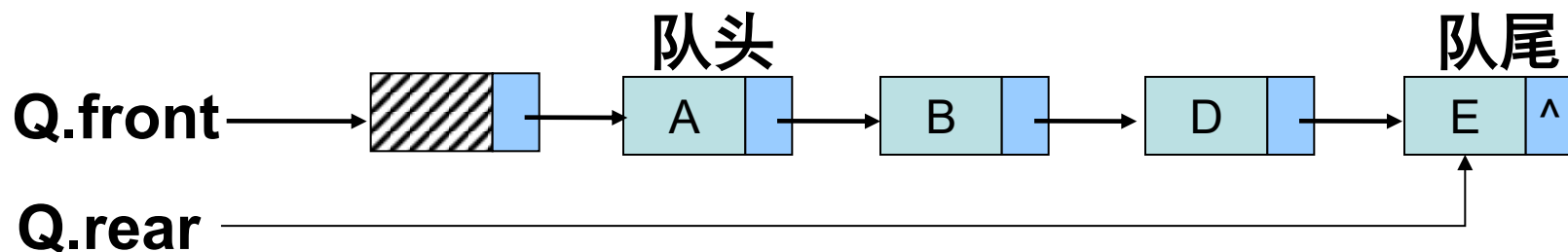
6. 当前队列元素的个数（长度）

$$\mathbf{(Q.rear - Q.front + MAXQSIZE) \% MAXQSIZE}$$

3.5.3 队列的链式表示和实现——链队列

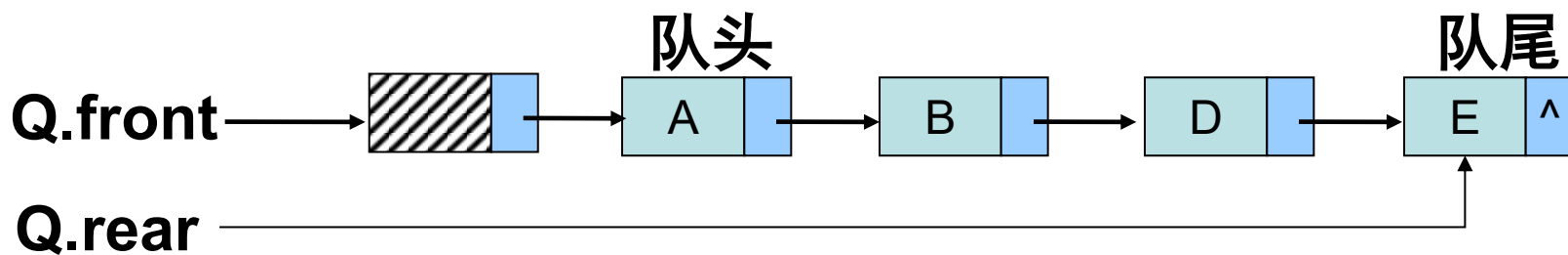
与单链表结构一样，用一带头结点的单链表表示队列，成为链队列。显然，需要两个分别指向队头和队尾的指针，并约定**头指针端为队头**。

由此，链队列的结构与单链表一样，唯一区别在于操作不同。入队和出队操作实际是单链表插入和删除的特殊情况。



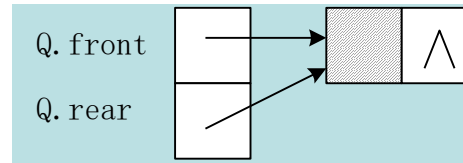
链队列存储结构

```
typedef struct QNode{  
    QElemType data;  
    struct QNode *next;  
}Qnode, *QueuePtr;  
  
typedef struct {  
    QueuePtr front;           //队头指针  
    QueuePtr rear;           //队尾指针  
}LinkQueue;
```



(a) 空队列

$Q.front == Q.rear$

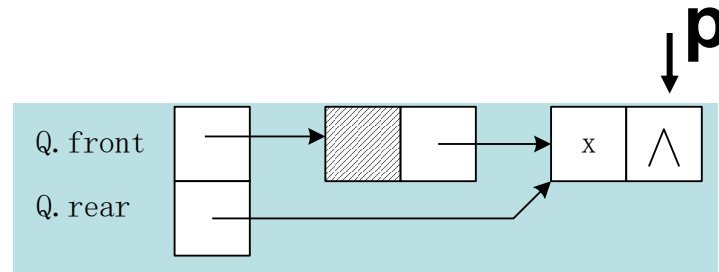


(b) 元素x入队列

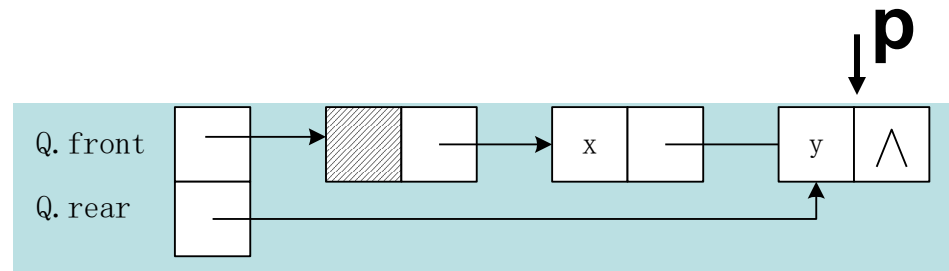
$p \rightarrow next = NULL;$

$Q.rear \rightarrow next = p;$

$Q.rear = p$



(c) 元素y入队列



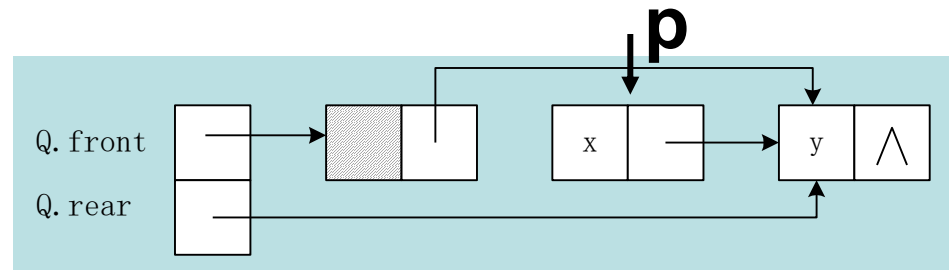
(d) 元素x出队列

$p = Q.front \rightarrow next;$

$e = p \rightarrow data;$

$Q.front \rightarrow next = p \rightarrow next;$

$delete\ p;$



链队列初始化

```
Status InitQueue (LinkQueue &Q){
```

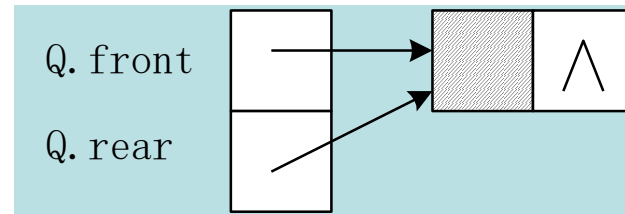
```
    Q.front=Q.rear=new QNode; //生成头结点
```

```
    if(!Q.front) exit(OVERFLOW);
```

```
    Q.front->next=NULL; //头结点指针域置空
```

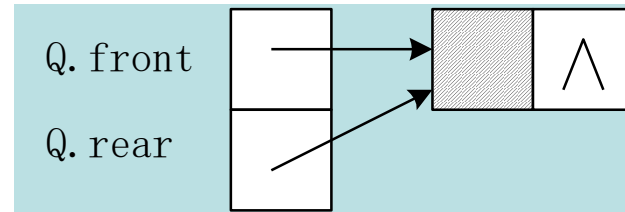
```
    return OK;
```

```
}
```



判断链队列是否为空

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



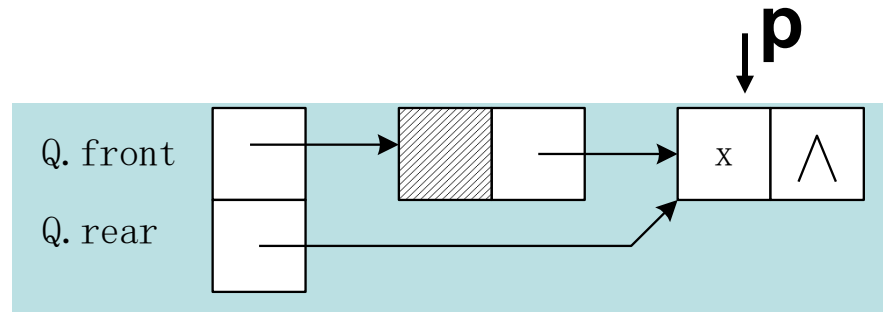
求链队列的队头元素

```
Status GetHead (LinkQueue Q, QElemType &e){  
    //由引用参数e带回  
    if(Q.front==Q.rear) return ERROR;  
    e=Q.front->next->data;  
    return OK;  
}
```

```
QElemType GetHead (LinkQueue Q){  
    //直接返回，另一方式  
    if(Q.front==Q.rear) exit(0);  
    return Q.front->next->data;  
}
```

链队列入队

```
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;  
}
```



链队列出队

Status DeQueue (LinkQueue &Q, QElemType &e){

//删除队头元素，用e返回其值

if(Q.front==Q.rear) return ERROR;

p=Q.front->next;

e=p->data;

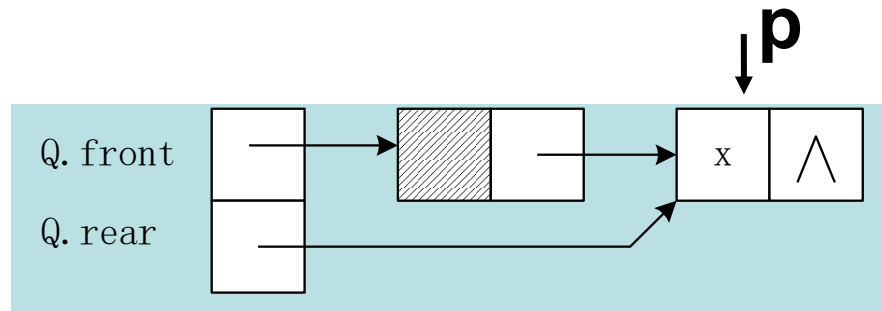
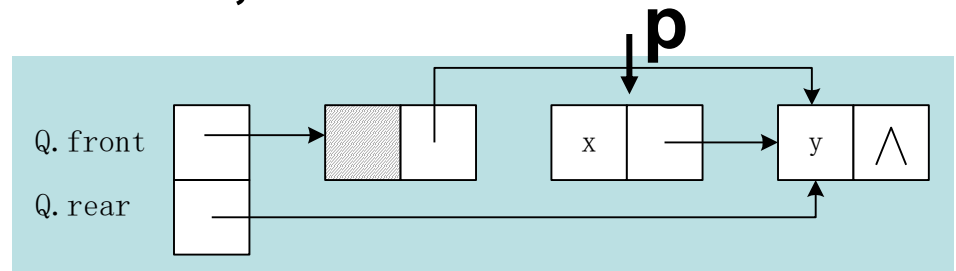
Q.front->next=p->next;

if(Q.rear==p) Q.rear=Q.front;

free(p);

return OK;

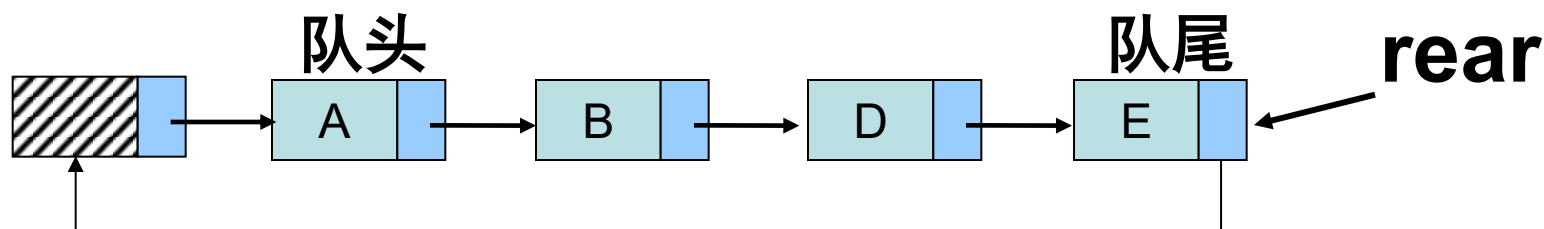
}



当队列只有一个元素时的删除

链队列的变化-----循环队列

用一个带头结点的循环链表来表示循环队列，
且该队列只设尾指针。



队列的应用



【例1】汽车加油站

结构：入口和出口为单行道，
加油车道若干条 n ，每辆车加油都要经过三段路程，三个队列：

- 入口处排队等候进入加油车道
- 在加油车道排队等候加油
- 出口处排队等候离开

若用算法模拟，需要设置 $n+2$ 个队列。



【例2】模拟打印机缓冲区

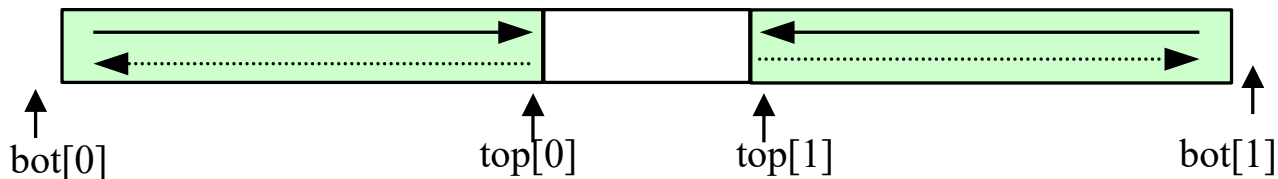
在主机将数据输出到打印机时，会出现主机速度与打印机的打印速度不匹配的问题。这时主机就要停下来等待打印机。显然，这样会降低主机的使用效率。为此人们设想了一种办法：为打印机设置一个打印数据缓冲区，当主机需要打印数据时，先将数据依次写入这个缓冲区，写满后主机转去做其他的事情，而打印机就从缓冲区中按照先进先出的原则依次读取数据并打印，这样做即保证了打印数据的正确性，又提高了主机的使用效率。由此可见，打印机缓冲区实际上就是一个队列结构。

【例3】约瑟夫环

算法设计题----共享栈

将编号为0和1的两个栈存放于一个数组空间 $V[m]$ 中，栈底分别处于数组的两端。当第0号栈的栈顶指针 $top[0]$ 等于-1时该栈为空；当第1号栈的栈顶指针 $top[1]$ 等于 m 时，该栈为空。两个栈均从两端向中间增长。试编写双栈初始化，判断栈空、栈满、进栈和出栈等算法的函数。双栈数据结构的定义如下：

```
typedef struct{  
    int top[2], bot[2]; //栈顶和栈底指针  
    SElemType *V;      //栈数组  
    int m;              //栈最大可容纳元素个数  
}DbtStack;
```



实现

//初始化一个大小为m的双向栈s

Status Init_Stack(DbIStack &s,int m)

{

s.V=(SElemType*)malloc(m*sizeof(SElemType));

s.bot[0]=-1;

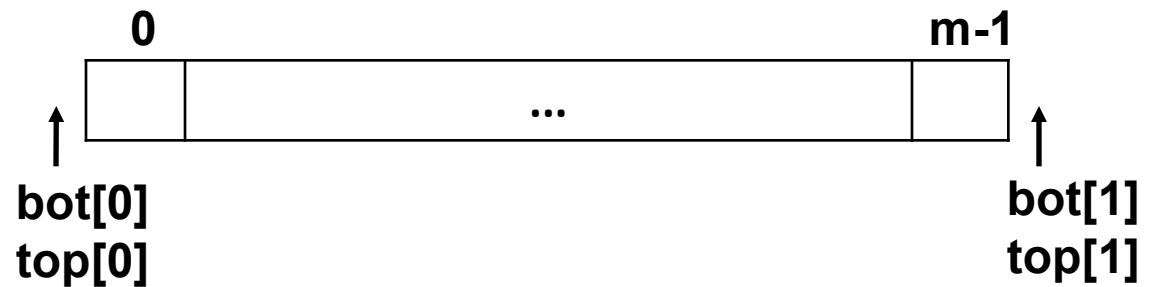
s.bot[1]=m;

s.top[0]=-1;

s.top[1]=m;

return OK;

}



//判栈i空否, 空返回1, 否则返回0

```
int IsEmpty(DblStack s,int i)
{    return s.top[i] == s.bot[i];    }
```

//判栈满否, 满返回1, 否则返回0

```
int IsFull(DblStack s)
{
    if(s.top[0]+1==s.top[1])
        return 1;
    else
        return 0;
}
```

```
void Dbllpush(DbllStack &s,SElemType x,int i)  
{  
    if( IsFull (s ) ) exit(1);  
    // 栈满则停止执行  
    if ( i == 0 ) s.V[ ++s.top[0] ] = x;  
    //栈0情形： 栈顶指针先加1, 然后按此地址进栈  
    else s.V[--s.top[1]]=x;  
    //栈1情形： 栈顶指针先减1, 然后按此地址进栈  
}
```

```
int Dbllpop(DblStack &s,int i,SElemType &x)  
{if ( IsEmpty ( s,i ) ) return 0;  
    //判栈空否, 若栈空则函数返回0  
    if ( i == 0 ) s.top[0]--; //栈0情形: 栈顶指针减1  
    else s.top[1]++; //栈1情形: 栈顶指针加1  
    return 1;  
}
```