



数据结构与算法

Data Structure and Algorithm

第3章 栈和队列



目 录

3.1 栈

3.2 队列

退出



3.1 栈

3.1.1 抽象数据类型栈的定义

3.1.2 栈的表示和实现

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

3.2.4 迷宫求解

3.2.5 递归实现



3.1 栈

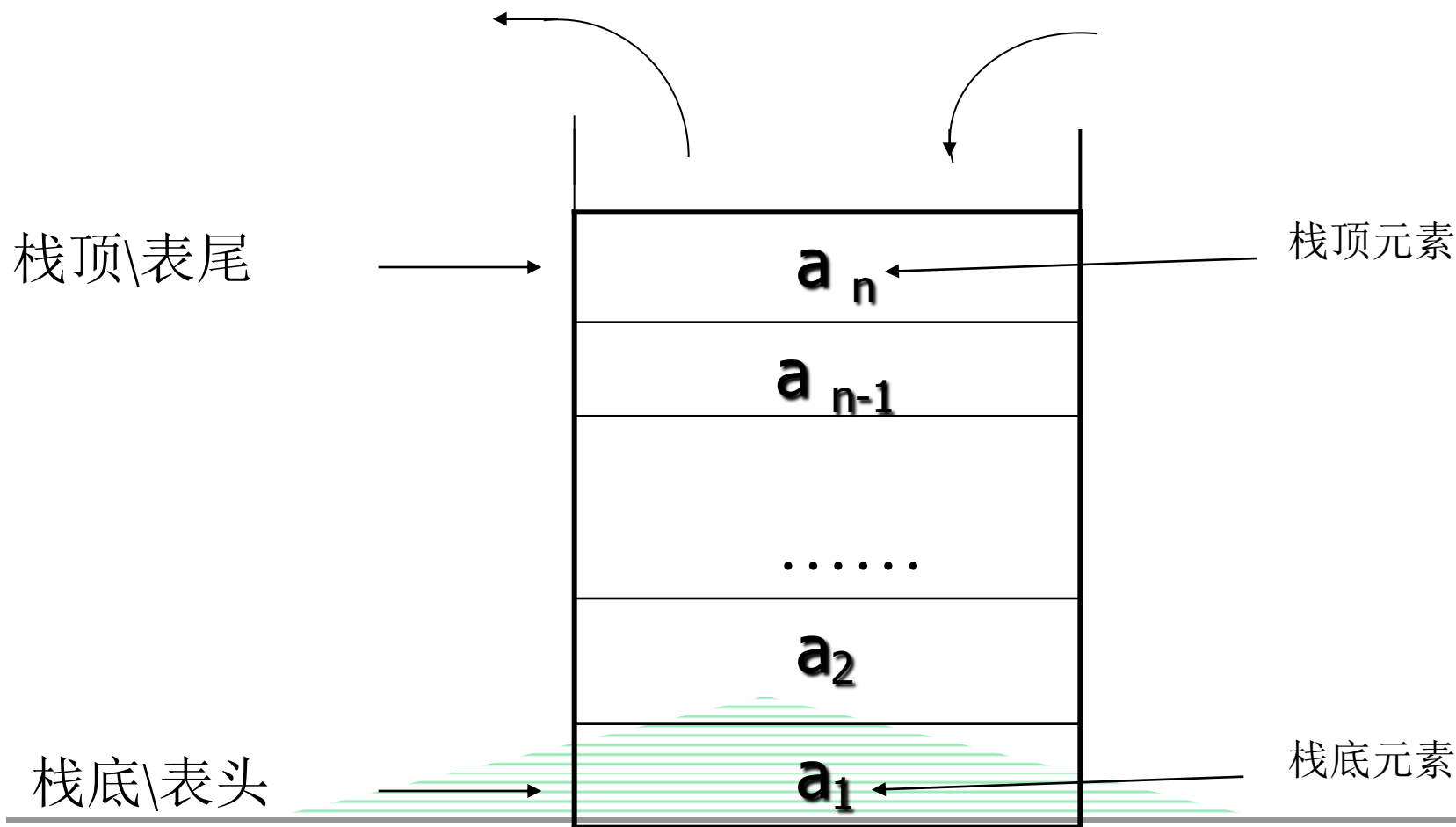
3.1.1 抽象数据类型栈的定义

栈(**Stack**)是限制在表的一端进行插入和删除运算的线性表，通常称插入、删除的这一端为栈顶(**Top**)，另一端为栈底(**Bottom**)。当表中没有元素时称为空栈。

假设栈 $S=(a_1, a_2, a_3, \dots, a_n)$ ，则 a_1 称为栈底元素， a_n 为栈顶元素。栈中元素按 $a_1, a_2, a_3, \dots, a_n$ 的次序进栈，退栈的第一个元素应为栈顶元素。换句话说，栈的修改是按后进先出的原则进行的。因此，栈称为后进先出表（**LIFO**）

例：一个杯子或一叠盘子。

栈的抽象数据类型的定义如下： P₄₄





3.1.2 栈的表示和实现

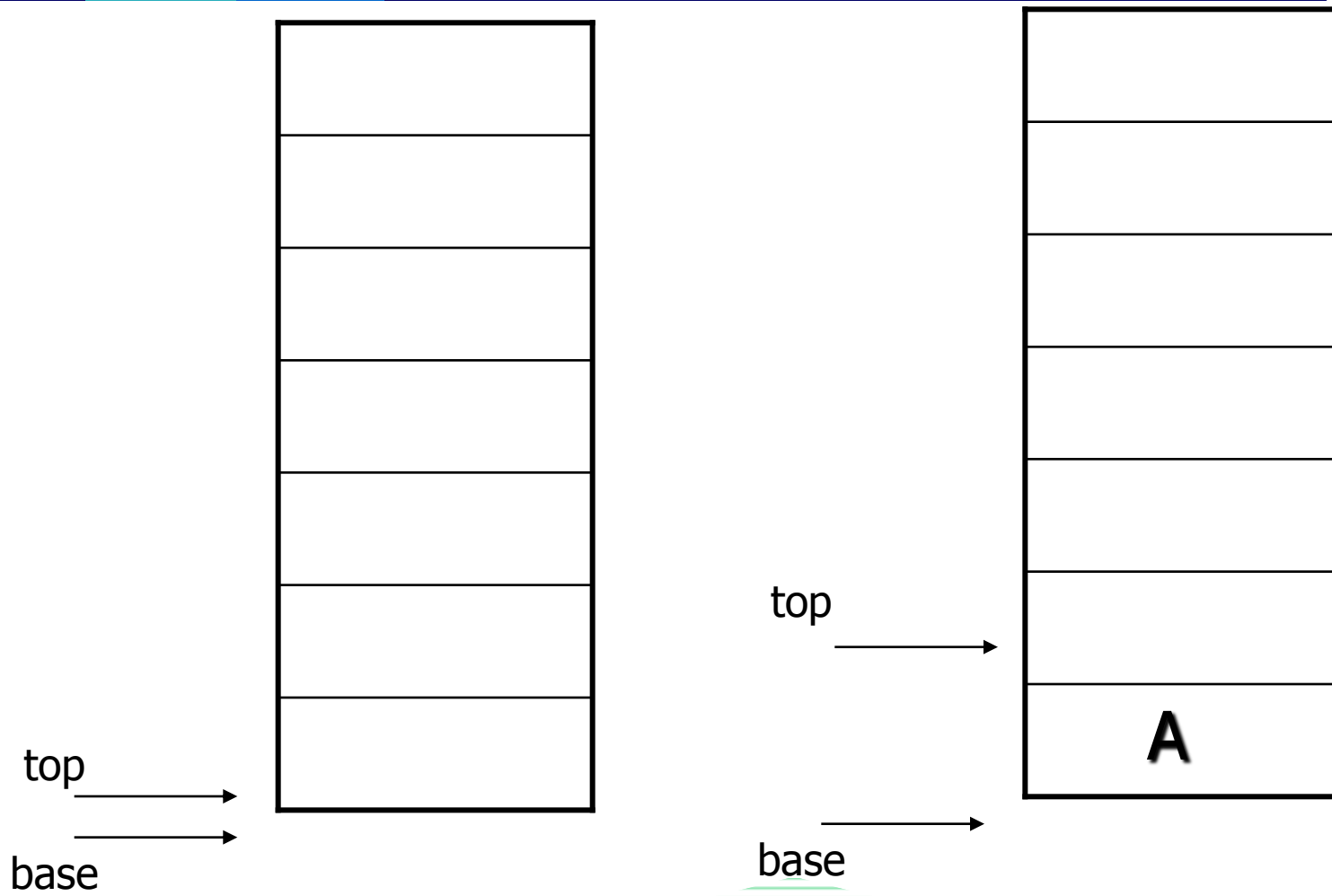
1. 顺序栈

由于栈是运算受限的线性表，因此线性表的存储结构对栈也适应。

栈的顺序存储结构简称为顺序栈，它是运算受限的线性表。因此，可用数组来实现顺序栈。因为栈底位置是固定不变的，所以可以将栈底位置设置在数组的两端的任何一个端点；栈顶位置是随着进栈和退栈操作而变化的，故需用一个整型变量`top`

顺序栈的类型定义如下：

```
#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; // 在栈构造之前和销毁之后,
                        // base==NULL
    SElemType * top; // 栈顶指针, 初值指向栈底, 既 top==base
                    // 非空栈中的栈顶指针始终指向栈顶元素
                    // 的下一个位置
    int stacksize; // 当前已分配的存储空间, 以元素为单位
}sqstack;
```



空栈

非空栈



栈的运算

1) 初始化栈

```
Status InitStack(SqStack &S) {  
    //构造一个空栈  
    S.base = (SElemType * )malloc(STACK_INIT_SIZE *  
        sizeof(ElemType));  
    if(!S.base) exit(OVERFLOW);  
    S.top=S.base;  
    S.stacksize=STACK_INIT_SIZE;  
    return ok;  
} //InitStack
```



2) 取栈顶元素

```
Status GetTop(SqStack S, SElemType &e){  
    //若栈不空，则用e返回S的栈顶元素，并返回OK；否则  
    // 返回ERROR  
    if( S.top==S.base ) return ERROR;  
    e = *(S.top -1);  
    return OK;  
} //GetTop
```



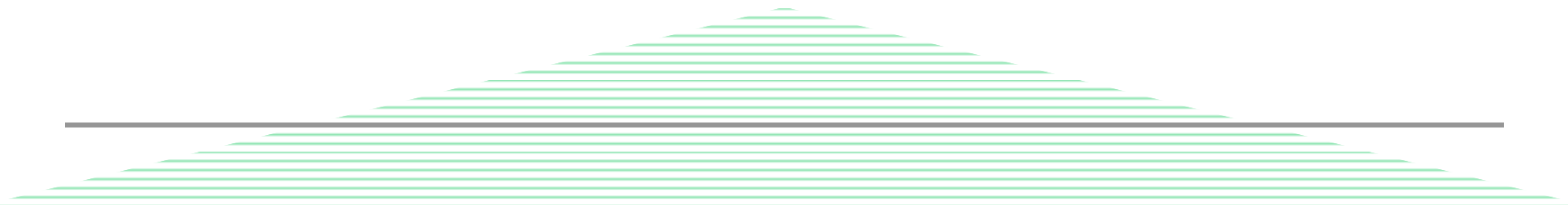
3) 插入元素

```
Status Push(SqStack &S, SElemType e) {  
    //插入元素e为新的栈顶元素  
    if(S.top-S.base>=S.stacksize){//栈满，追加存储空间  
        S.base=(ElemType * )realloc  
        (S.base,(S.stacksize+STACKINCREMENT)*sizeof(ElemType));  
        if(!S.base) exit(OVERFLOW);//存储分配失败  
        S.top=S.base+S.stacksize;  
        S.stacksize+=STACKINCREMENT;  
    }  
    *S.top++=e; //相当于 * S.top=e; S.top++;  
    return OK;  
} //Push
```



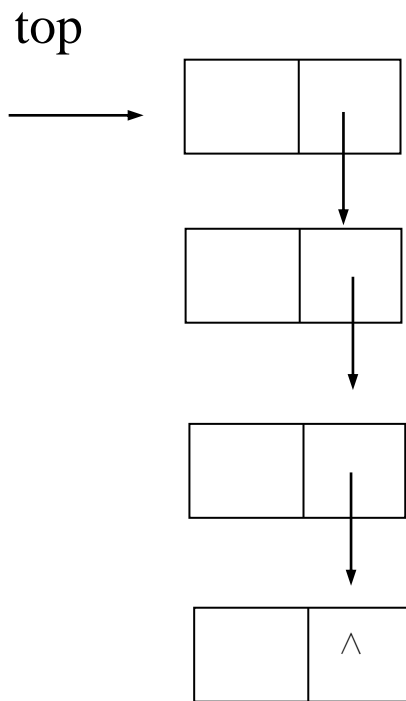
4) 删除元素

```
Status Pop(SqStack &S, SElemType &e){  
    //若栈不空，则删除S的栈顶元素，用e返回其值，并返  
    //回OK； 否则返回ERROR;  
    if( S.top==S.base ) return ERROR;  
    e = * --S.top; //相当于S.top--;  e = * S.top;  
    return OK;  
} //Pop
```



2.链栈

栈的链式存储结构,也称为链栈,它是一种限制运算的链表,即规定链表中的插入和删除运算只能在链表开头进行。



链栈示意图

删除栈顶元素

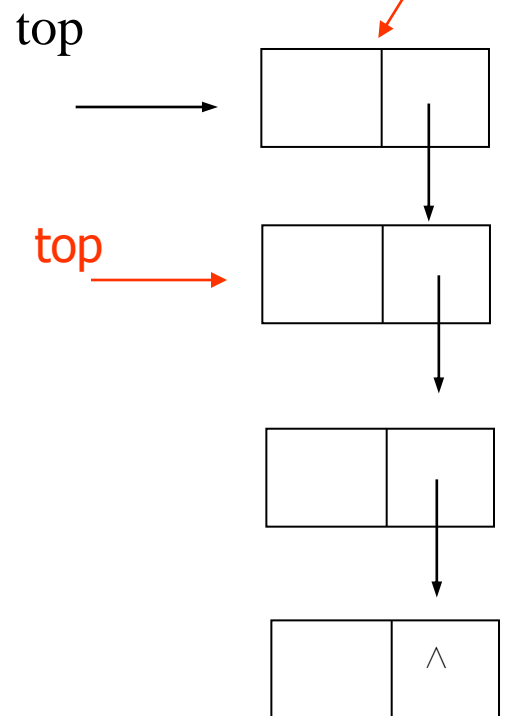
Pop:

`p=top;`

`top=top->next;`

`e=p->data;`

`free(p);`





3.2 栈的应用举例


由于栈结构具有的后进先出的固有特性，致使栈成为程序设计中常用的工具。以下是几个栈应用的例子。

3.2.1 数制转换

十进制**N**和其它进制数的转换是计算机实现计算的基本问题,其解决方法很多,其中一个简单算法基于下列原理:

$$N=(n \text{ div } d)*d+n \text{ mod } d$$

(其中:**div**为整除运算,**mod**为求余运算)



例如 $(1348)_{10} = (2504)_8$ ，其运算过程如下：

$n \div 8$	$n \bmod 8$	余数
1348	168	4
168	21	0
21	2	5
2	0	2

2
5
0
4



```
void conversion( ) {  
    //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  
    initstack(S);  
    scanf ("%d",N);  
    while(N){  
        push(S,n%8);  
        N=N/8;  
    }  
    while(! Stackempty(S)){  
        pop(S,e);  
        printf("%d",e);  
    }  
} //conversion
```



3.2.2 括号匹配的检验


假设表达式中充许括号嵌套, 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。例:

(() () (()))



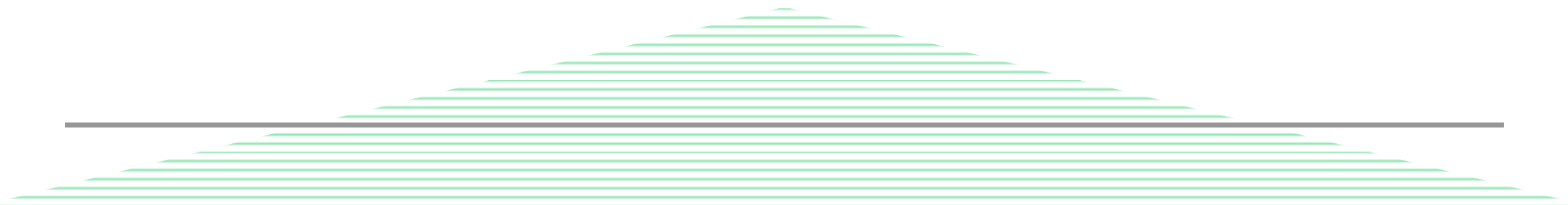
3.2.3 行编辑程序

在编辑程序中，设立一个输入缓冲区，用于接受用户输入的一行字符，然后逐行存入用户数据区。允许用户输入错误，并在发现有误时可以及时更正。



行编辑程序算法如下：

```
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);  
    ch=getchar( ); //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,c);      break;  
                case '@' : clearstack(s); break;  
                default : push(s,ch);      break;  
            }  
        }  
    }  
}
```





```
ch=getchar( ); //接收下一个字符
```

```
}
```

将从栈底到栈顶的栈内字符送至调用过程的数据区

```
clearstack(s);
```

```
if(ch!=EOF) ch=getchar( );
```

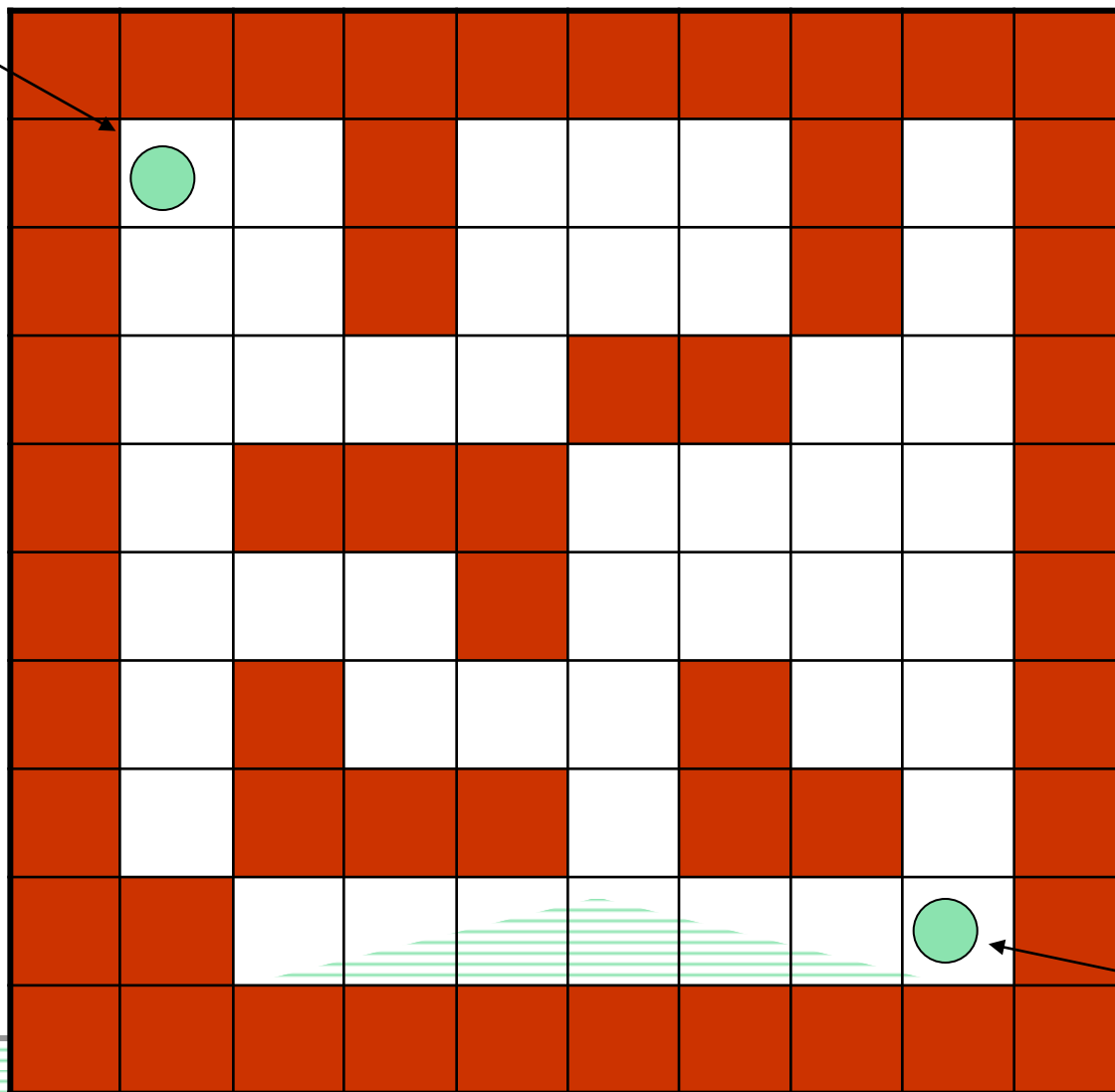
```
}
```

```
destroystack(s);
```

```
}//LineEdit
```

3.2.4 迷宫求解

入口



出口



入口(1,1)

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

出口 (6,8)



- 迷宫的定义如下：
- `#define m 6 /* 迷宫的实际行 */`
- `#define n 8 /* 迷宫的实际列 */`
- `int maze [m+2][n+2] ;`

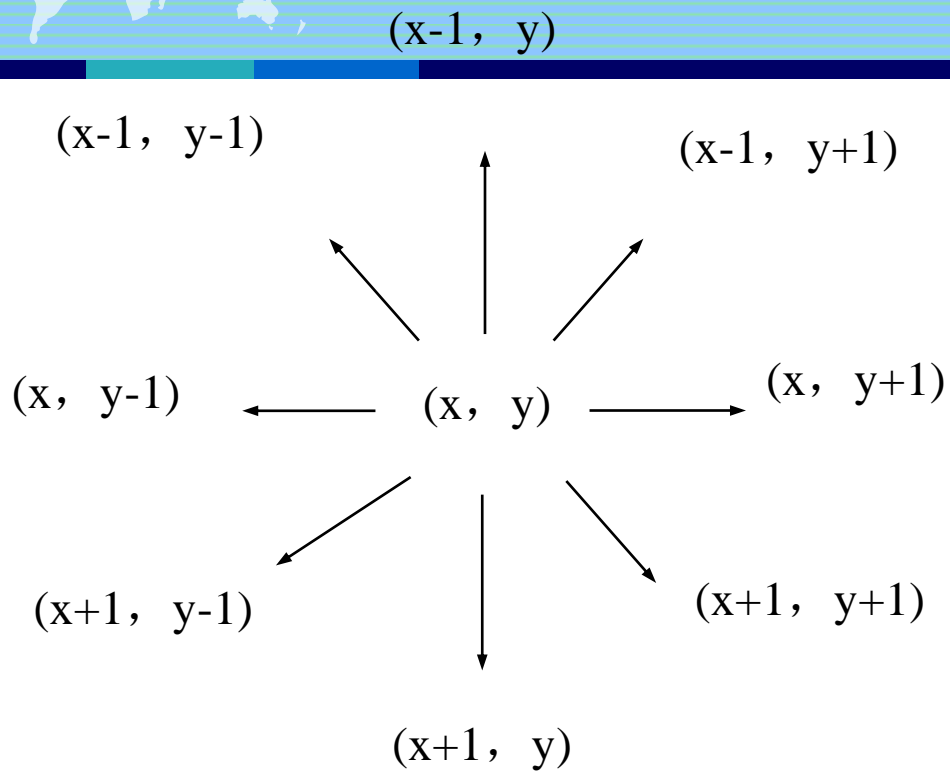


图3.5 与点 (x, y) 相邻的8个点及坐标

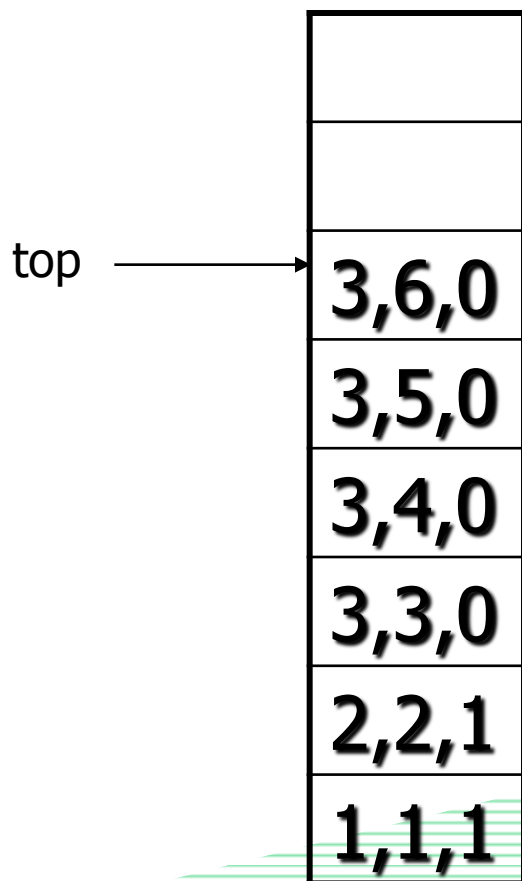
	x	y
0	0	1
1	1	1
2	1	0
3	1	-1
4	0	-1
5	-1	-1
6	-1	0
7	-1	1

图3.6 增量数组move



栈的设计:

当到达了某点而无路可走时需返回前一点，再从前一点开始向下一个方向继续试探。因此，压入栈中的不仅是顺序到达的各点的坐标，而且还要有从前一点到达本点的方向。



3.2.5. 递归调用

栈的一个重要应用是在程序设计语言中实现递归过程。现实中，有许多实际问题是递归定义的，这时用递归方法可以使许多问题的结果大大简化。

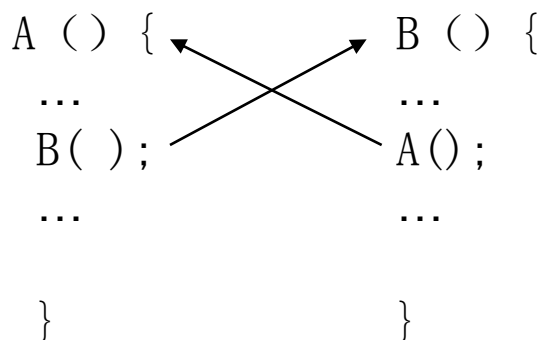
递归即自己调用自己，分两种，一种为直接调用，一种为间接调用。

直接调用：

```
A ( ) {  
...  
A ( );  
...  
}
```

间接调用：

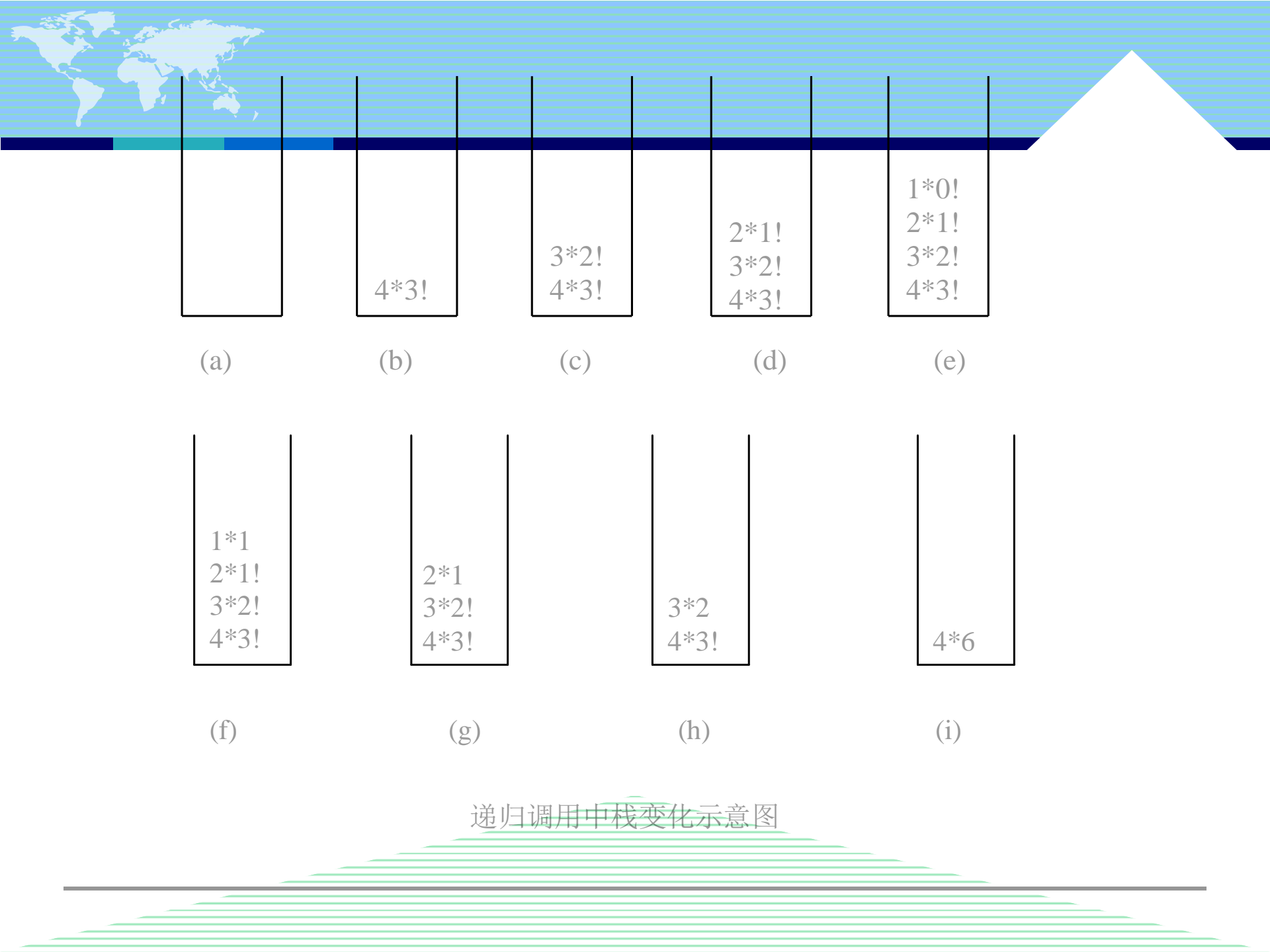
```
A ( ) {  
...  
B ( ) ;  
...  
}  
B ( ) {  
...  
A ( ) ;  
...  
}
```





例 求 $n!$ 可用递归函数描述如下:

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



递归调用中栈变化示意图



作业(栈):

3.1

3.4

3.15





3.4 队列

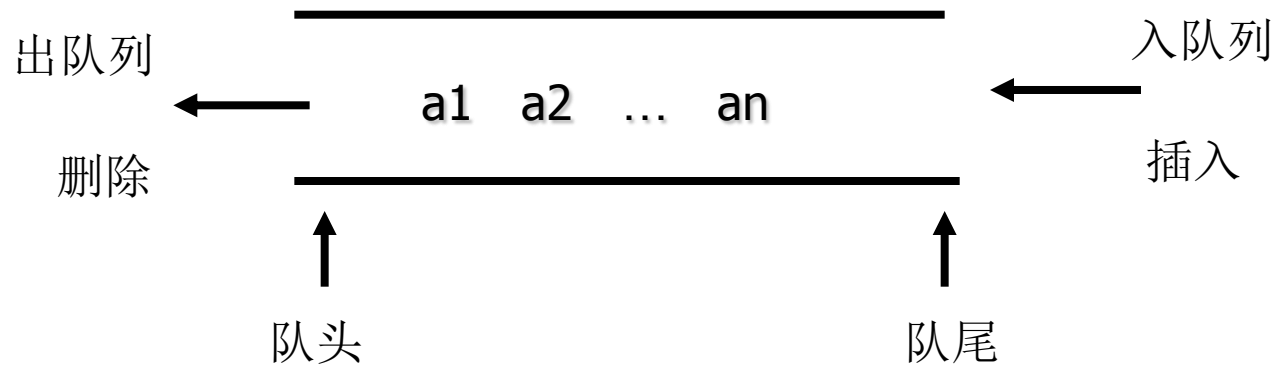
3.4.1 抽象数据类型队列的定义

队列 (Queue) 也是一种运算受限的线性表。它只允许在表的一端进行插入，而在另一端进行删除。允许删除的一端称为队头 (front)，允许插入的一端称为队尾 (rear)。

例如：排队购物。操作系统中的作业排队。先进入队列的成员总是先离开队列。因此队列亦称作先进先出 (First In First Out) 的线性表，简称FIFO表。

当队列中没有元素时称为空队列。在空队列中依次加入元素 a_1, a_2, \dots, a_n 之后， a_1 是队头元素， a_n 是队尾元素。显然退出队列的次序也只能是 a_1, a_2, \dots, a_n ，也就是说队列的修改是依先进先出的原则进行的。

下图是队列的示意图：



队列的抽象数据定义 P 59



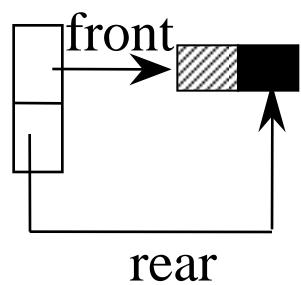
3.4.2 链队列

1. 链队列的数据类型描述

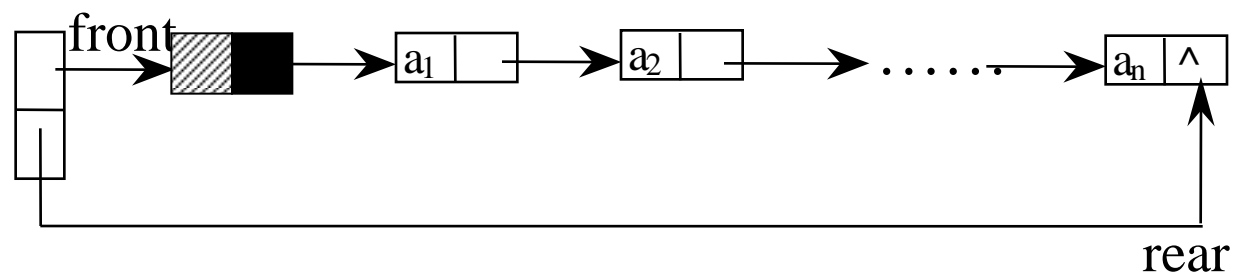
队列的链式存储，称为链队列，与前面介绍的单链表类似，但为了使头指针，尾指针统一起来，另外定义一种数据类型如下：

```
typedef struct QNode {           //定义单链表数据类型
    QElemType data;
    struct QNode *next;
}QNode,*QueuePtr;

typedef struct LinkQueue {       //定义链队列数据类型
    QueuePtr front,rear;        //定义头指针和尾指针
}LinkQueue;
```



(a) 空链队列



(b) 非空链队

链队列示意图

2. 链队列上的基本运算

链队列上的四种运算如下：

(1) 链队列上的初始化

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

```
    //构造一个空队列Q
```

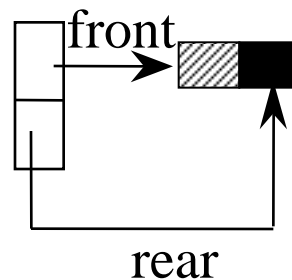
```
    Q.front = Q.rear=(QueuePtr) malloc(sizeof(QNode));
```

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

```
    Q.front->next=NULL;
```

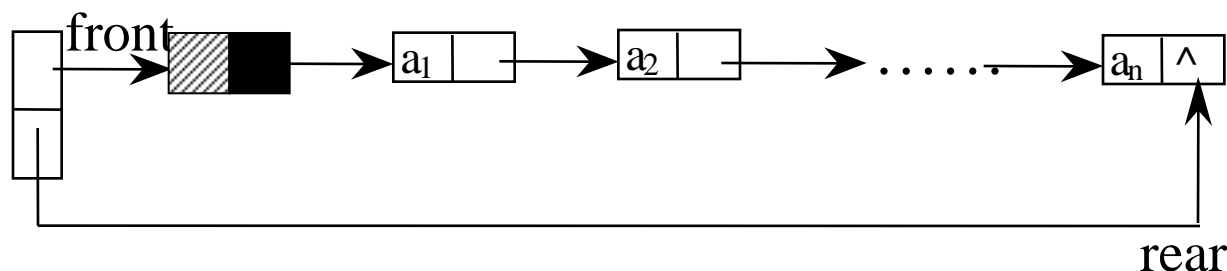
```
    return OK;
```

```
}//InitQueue
```



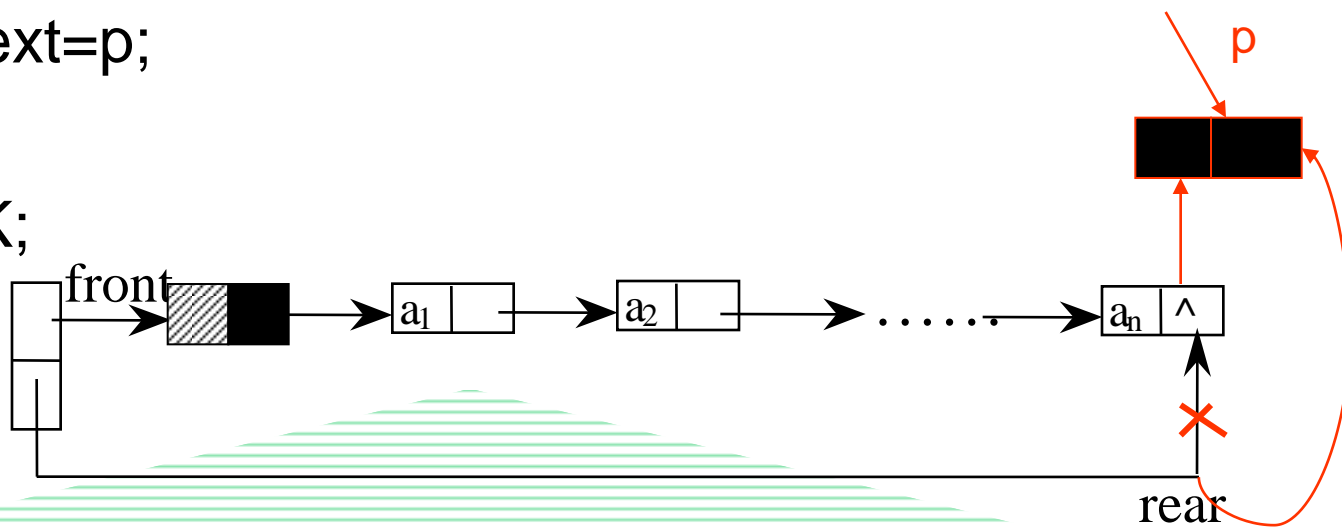
(2) 销毁链队列

```
Status DestoryQueue(LinkQueue &Q){  
    //销毁队列Q,既头结点也删除, 区别于倾空队列  
    while(Q.front){  
        Q.rear=Q.front->next;  
        free(Q.front);  
        Q.front=Q.rear;  
    }  
    return OK;  
} //DestoryQueue
```



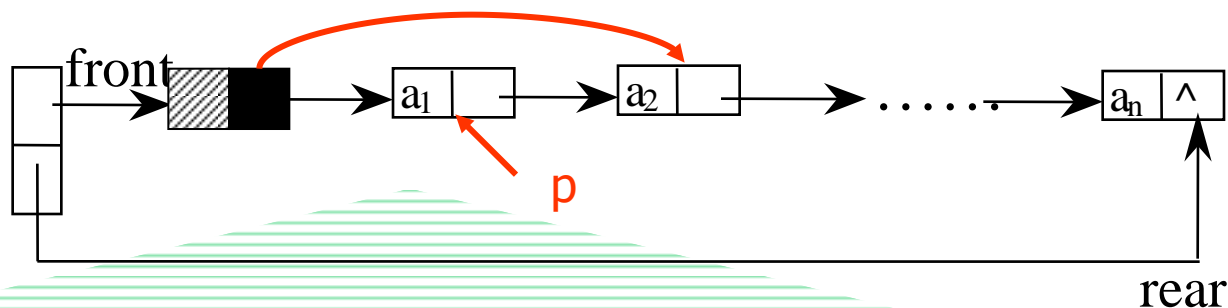
(3) 入队列

```
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);  
    p->data=e;  
    p->next=null;  
    Q.rear->next=p;  
    Q.rear=p;  
    return OK;  
} //EnQueue
```



(4) 出队列

```
Status DeQueue(LinkQueue &Q, QElemType &e){  
    //若队列不为空，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;  
    free(p);  
    return OK;  
} //DeQueue
```

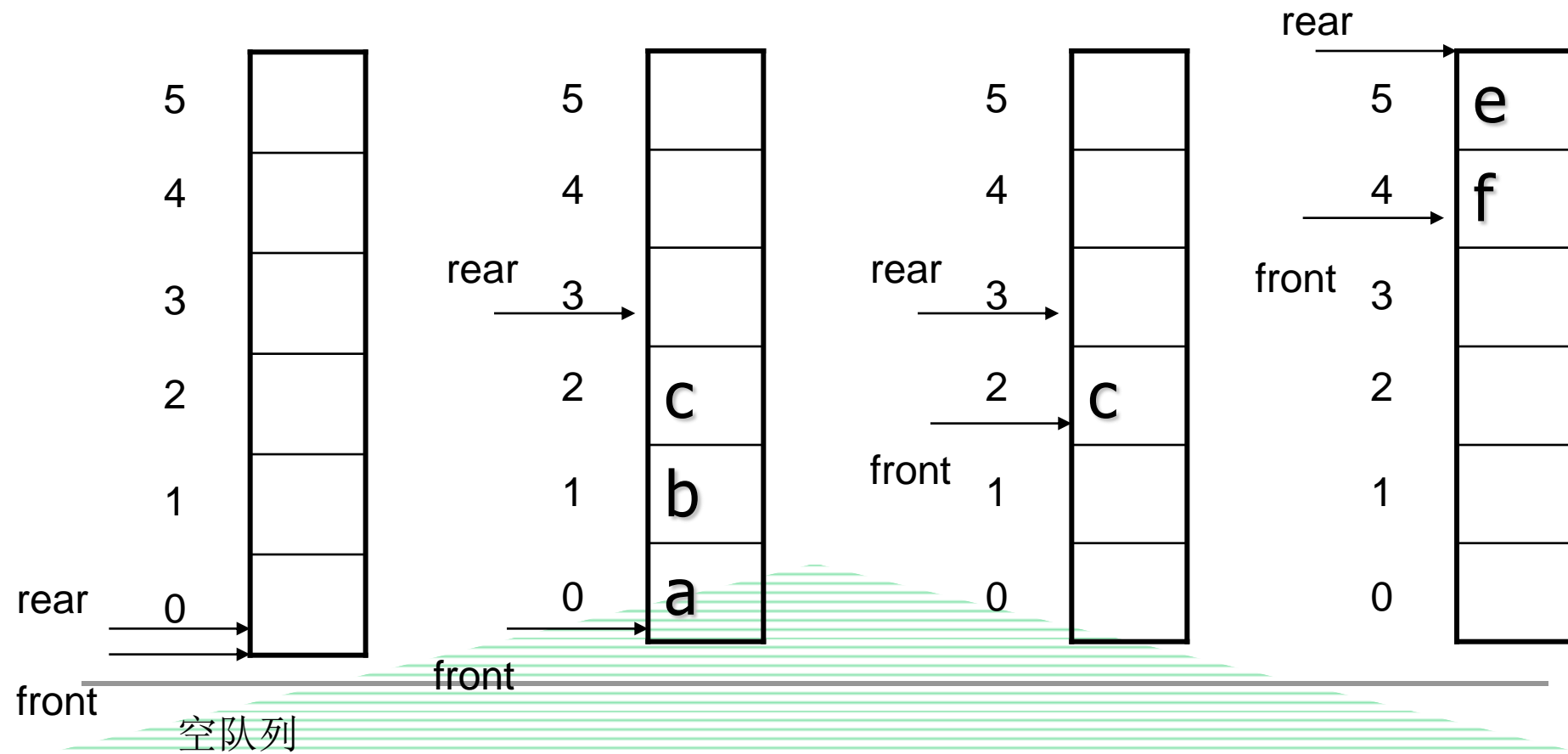




3.4.3 循环队列

1. 顺序队列

将队列中元素全部存入一个一维数组中,数组的低下标一端为队头,高下标一端为队尾,将这样的队列看成是顺序队列。





顺序队列指针和队列中元素之间年的关系:

- 空的顺序队列: $\text{front} == \text{rear}$
- 非空顺序队列: front 指向队头元素, rear 指向队尾元素的下一个位置
- 插入新的队尾元素: $\text{rear}++$
- 删除队头元素: $\text{front}++$

2. 循环队列

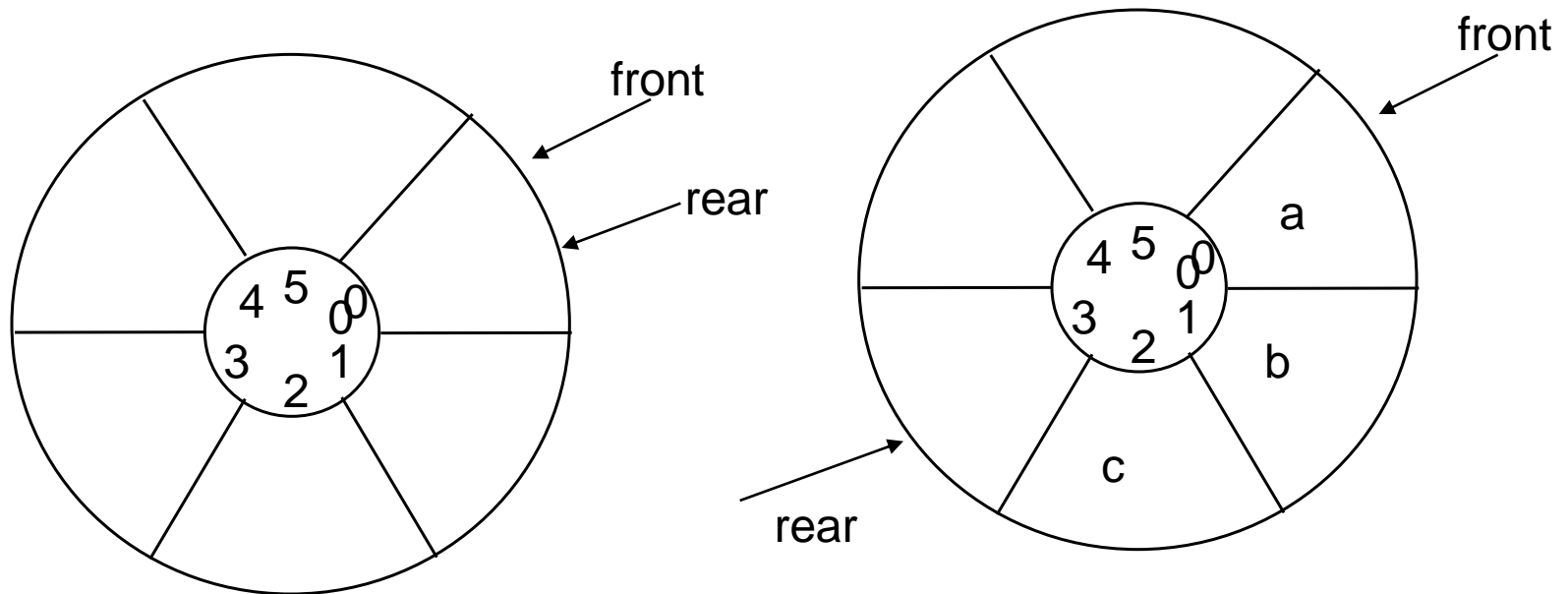
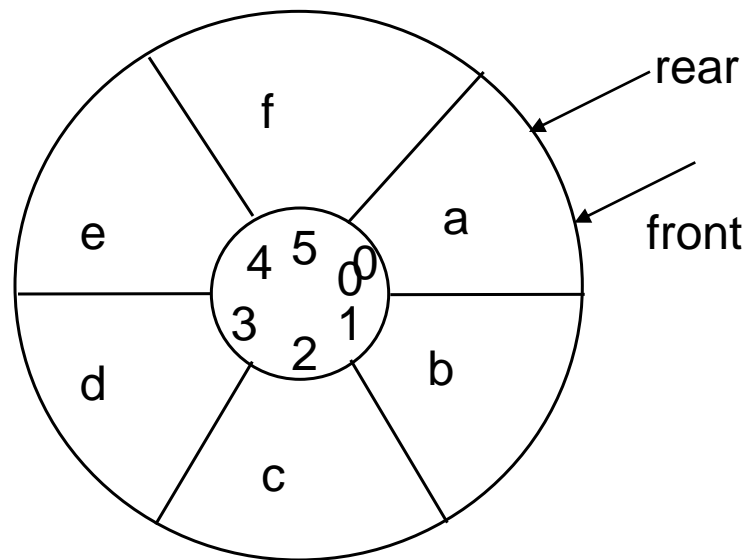
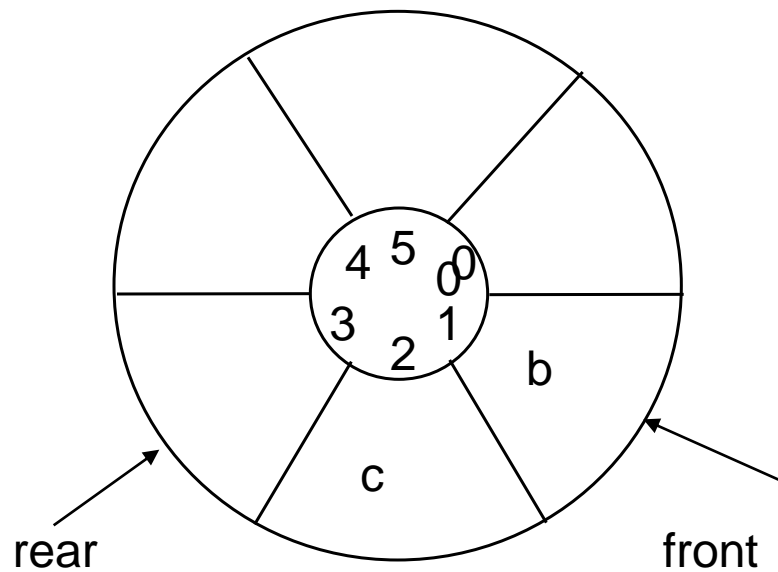


图 3-11 循环队列示意图



[解决方法](#)

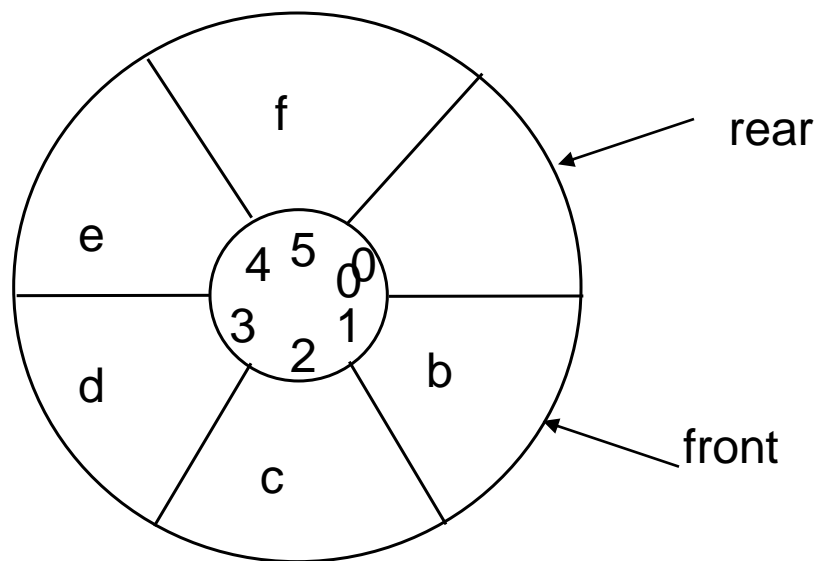


注意：

- 空间大小一次开辟： MAXQSOZE
- 增加一个元素： $\text{rear} = (\text{rear} + 1) \% \text{MAXQSOZE}$
- 删除一个元素： $\text{front} = (\text{front} + 1) \% \text{MAXQSOZE}$

解决循环队列“空”和“满”时都为 $front == rear$:

- 另设一个标志位以区别队列是“空”还是“满”
练习册3.29
- 少用一个元素空间, 约定 $front$ 在 $rear$ 的下一个位置上时, 作为队列“满”状态的标志, 即 $(rear+1) \% MAXQSIZE == front$



满是存储空间大小为 $MAXQSIZE-1$



循环队列的运算：

(1) 队列初始化

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

```
    //构造一个空队列
```

```
    Q.base=(ElemType *)malloc(MAXQSIZE*sizeof(ElemType));
```

```
    if(Q.base) exit(OVERFLOW);
```

```
    Q.front=Q.rear=0;
```

```
    return OK;
```

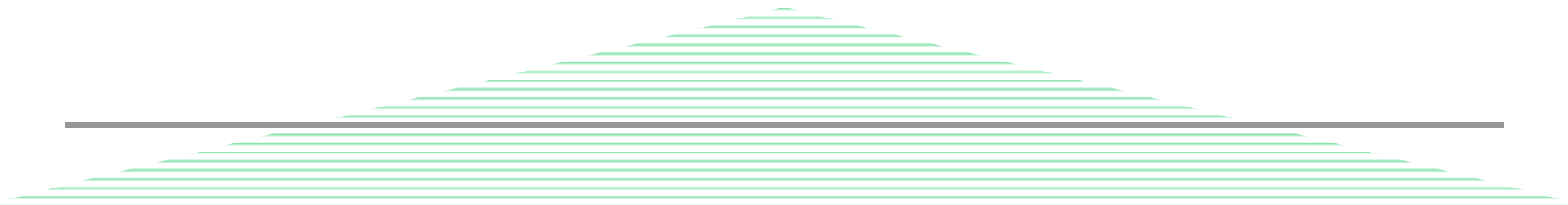
```
}//InitQueue
```





(2)求队列的长度

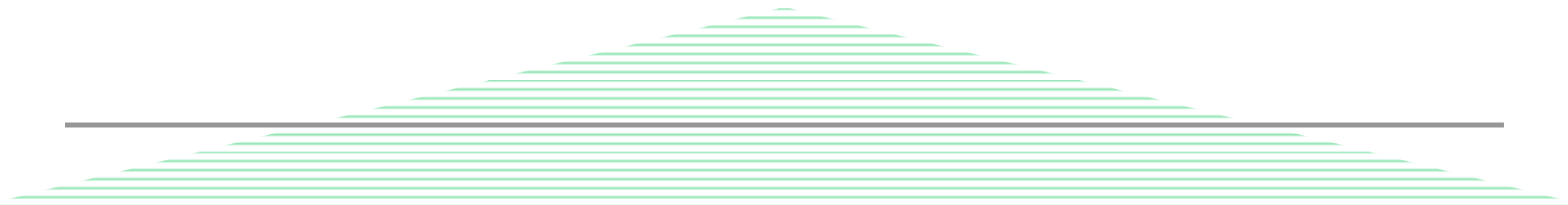
```
Int QueueLength(SqQueue Q){  
    //返回Q的元素个数，即队列的长度  
    return (Q.rear-Q.front+MAXQSIZE)%MAXQSIZE;  
}
```





(3) 入队列

```
Status EnQueue ( SqQueue &Q, QElemType e){  
    //插入元素e为Q的新的队尾元素  
    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;  
}  
//EnQueue
```





(4) 出队列

```
Status DeQueue(SqQueue &Q,QElemType &e){
```

```
    //若队列为空，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; //队列空
```

```
    e=Q.base [ Q.front ];
```

```
    Q.front = (Q.front+1)%MAXQSIZE;
```

```
    return OK;
```

```
}//DeQueue
```





作业（队列）：

3.12

3.28

