

# 数据结构

李春葆

清华大学

## 栈和队列

# 栈

## ■ 栈

### ■ 栈的定义

### ■ 栈的顺序存储结构及其基本运算实现

### ■ 栈的链式存储结构及其基本运算的实现

### ■ 栈的应用例子

### 3.1.1 栈的定义

---

- 栈是一种**只能在一端**进行插入或删除操作的线性表。表中允许进行插入、删除操作的一端称为**栈顶**。
  - 栈顶的当前位置是动态的,栈顶的当前位置由一个称为栈顶指针的位置指示器指示。表的另一端称为**栈底**。
  - 当栈中没有数据元素时,称为**空栈**。
  - 栈的插入操作通常称为进栈或**入栈**,栈的删除操作通常称为退栈或**出栈**。

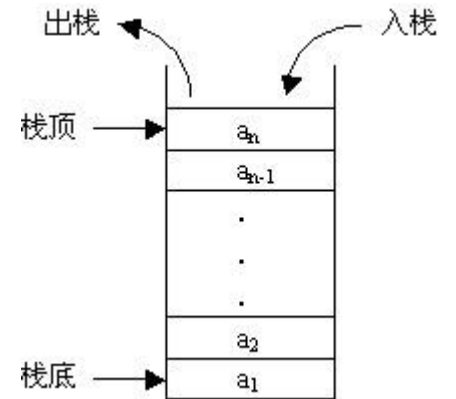


进栈

出栈

栈顶

栈底



栈示意图

# 【例】

- 例3.1 设有4个元素a、b、c、d进栈,给出它们所有可能的出栈次序。

- 答:所有可能的出栈次序如下:

**abcd abdc acbd acdb**

**adcb bacd badc bcad**

**bcda bdca cbad cbda**

**cdba dcba**



- 
- 例3.2 设一个栈的输入序列为A,B,C,D,则借助一个栈所得到的输出序列不可能是

**(A) A,B,C,D**

**(B) D,C,B,A**

**(C) A,C,D,B**

**(D) D,A,B,C**

- 答:可以简单地推算,得容易得出D,A,B,C是不可能的,因为D先出来,说明A,B,C,D均在栈中,按照入栈顺序,在栈中顺序应为D,C,B,A,出栈的顺序只能是D,C,B,A。所以本题答案为D。

- 
- 例3.3 已知一个栈的进栈序列是 $1, 2, 3, \dots, n$ , 其输出序列是 $p_1, p_2, \dots, p_n$ , 若 $p_1 = n$ , 则 $p_i$ 的值 。

(A)  $i$

(B)  $n-i$

(C)  $n-i+1$

(D) 不确定

- 答: 当 $p_1 = n$ 时, 输出序列必是 $n, n-1, \dots, 3, 2, 1$ , 则有:

$$p_2 = n-1,$$

$$p_3 = n-2,$$

$\dots,$

$$p_n = 1$$

推断出 $p_i = n-i+1$ , 所以本题答案为C。



- 例3.4 设 $n$ 个元素进栈序列是 $1, 2, 3, \dots, n$ , 其输出序列是 $p_1, p_2, \dots, p_n$ , 若 $p_1=3$ , 则 $p_2$ 的值 。

(A) 一定是2

(B) 一定是1

(C) 不可能是1

(D) 以上都不对

- 答: 当 $p_1=3$ 时, 说明 $1, 2, 3$ 先进栈, 立即出栈 $3$ , 然后可能出栈, 即为 $2$ , 也可能 $4$ 或后面的元素进栈, 再出栈。因此,  $p_2$ 可能是 $2$ , 也可能是 $4, \dots, n$ , 但一定不能是 $1$ 。所以本题答案为C。



# 栈的几种基本运算

---

- (1) 初始化栈 **InitStack(&s)**: 构造一个空栈s。
- (2) 销毁栈 **ClearStack(&s)**: 释放栈s占用的存储空间。
- (3) 求栈的长度 **StackLength(s)**: 返回栈s中的元素个数。
- (4) 判断栈是否为空 **StackEmpty(s)**: 若栈s为空, 则返回真; 否则返回假。

# 栈的几种基本运算

---

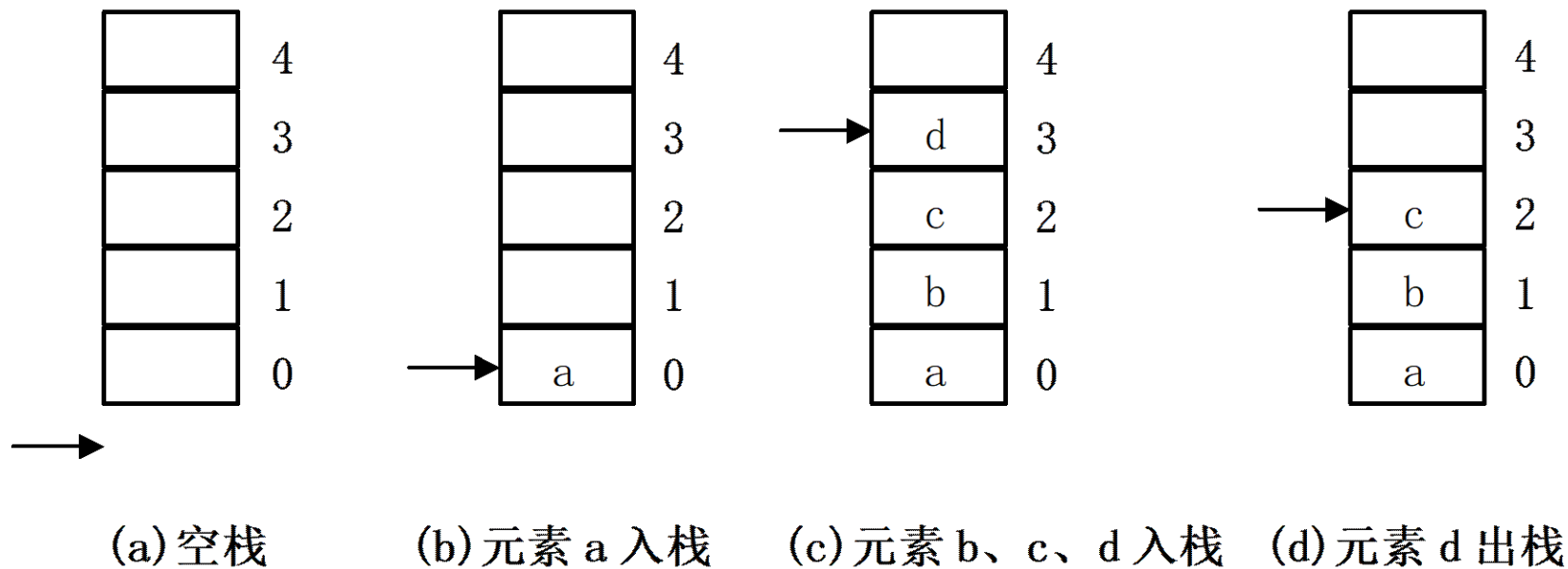
- (5) 进栈Push(&S,e):将元素e插入到栈s中作为栈顶元素。
- (6) 出栈Pop(&s,&e):从栈s中退出栈顶元素,并将其值赋给e。
- (7) 取栈顶元素GetTop(s,&e):返回当前的栈顶元素,并将其值赋给e。
- (8) 显示栈中元素DispStack(s):从栈顶到栈底顺序显示栈中所有元素。

## 3.1.2 栈的顺序存储结构及其基本运算实现

- 假设栈的元素个数最大不超过正整数MaxSize,所有的元素都具有同一数据类型ElemType,则可用下列方式来定义栈类型SqStack:

```
typedef struct
{
    ElemType data[MaxSize];
    int top;                /*栈指针*/
} SqStack;
```

MaxSize=5



## 顺序栈进栈和出栈示意图

# 在顺序栈中实现栈的基本运算算法:

---

- (1) 初始化栈initStack(&s)
  - 建立一个新的空栈s,实际上是将栈顶指针指向-1即可。  
对应算法如下:

```
void InitStack(SqStack *&s)  
{  
    s=(SqStack *)malloc(sizeof(SqStack));  
    s->top=-1;  
}
```

# 在顺序栈中实现栈的基本运算算法:

---

- (2) 销毁栈ClearStack(&s)
  - 释放栈s占用的存储空间。对应算法如下:

```
void ClearStack(SqStack *&s)  
{  
  
    free(s);  
  
}
```



# 在顺序栈中实现栈的基本运算算法:

---

- (3) 求栈的长度StackLength(s)
  - 返回栈s中的元素个数,即栈指针加1的结果。对应算法如下:

```
int StackLength(SqStack *s)
{
    return(s->top+1);
}
```



# 在顺序栈中实现栈的基本运算算法:

---

- (4) 判断栈是否为空StackEmpty(s)

- 栈S为空的条件是 $s \rightarrow \text{top} == -1$ 。对应算法如下:

```
int StackEmpty(SqStack *s)
{
    return(s->top==-1);
}
```

# 在顺序栈中实现栈的基本运算算法:

---

- (5) 进栈Push(&s,e)
  - 在栈不满的条件下,先将栈指针增1,然后在该位置上插入元素e。对应算法如下:

```
int Push(SqStack *&s,ElemType e)
{   if (s->top==MaxSize-1) return 0;
    /*栈满的情况,即栈上溢出*/
    s->top++;
    s->data[s->top]=e;
    return 1;
}
```

# 在顺序栈中实现栈的基本运算算法:

- (6) 出栈Pop(&s,&e)
  - 在栈不为空的条件下,先将栈顶元素赋给e,然后将栈指针减1。对应算法如下:

```
int Pop(SqStack *&s,ElemType &e)
{   if (s->top==-1) return 0;
        /*栈为空的情况,即栈下溢出*/
    e=s->data[s->top];
    s->top--;
    return 1;
}
```



# 在顺序栈中实现栈的基本运算算法:

- (7) 取栈顶元素GetTop(s)
  - 在栈不为空的条件下,将栈顶元素赋给e。对应算法如下:

```
int GetTop(SqStack *s,ElemType &e)
{
    if (s->top==-1) return 0;
    /*栈为空的情况,即栈下溢出*/
    e=s->data[s->top];
    return 1;
}
```



# 在顺序栈中实现栈的基本运算算法:

---

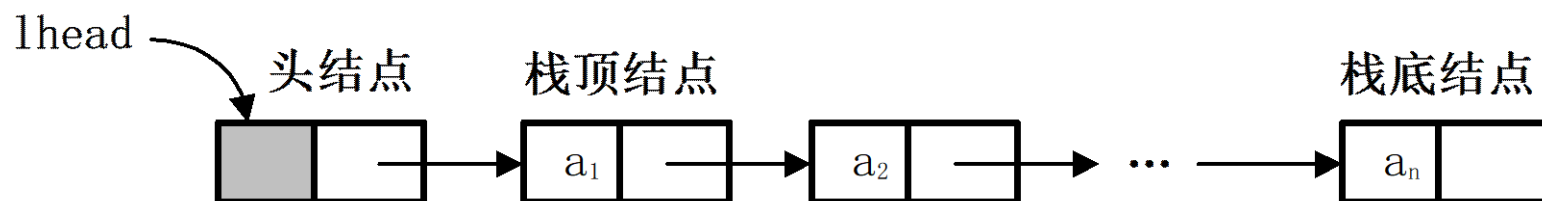
- (8) 显示栈中元素DispStack(s)
  - 从栈顶到栈底顺序显示栈中所有元素。对应算法如下:

```
void DispStack(SqStack *s)
{
    int i;
    for (i=s->top;i>=0;i--)
        printf("%c ",s->data[i]);
    printf("\n");
}
```



### 3.1.3 栈的链式存储结构及其基本运算的实现

- 采用链式存储的栈称为链栈,这里采用单链表实现。
    - 链栈的优点是不存在栈满上溢的情况。
- 我们规定栈的所有操作都是在单链表的表头进行的,下图是头结点为\**lhead*的链栈,第一个数据结点是栈顶结点,最后一个结点是栈底结点。栈中元素自栈顶到栈底依次是 $a_1$ 、 $a_2$ 、...、 $a_n$ 。



## 链栈示意图

- 
- 链栈中数据结点的类型LiStack定义如下:

```
typedef struct linknode
```

```
{   ElemType data;           /*数据域*/
```

```
    struct linknode *next;    /*指针域*/
```

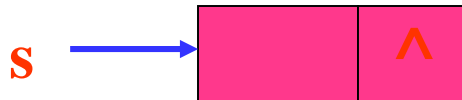
```
} LiStack;
```



# 链栈的基本运算

- (1) 初始化栈initStack(&s)
  - 建立一个空栈s。实际上是创建链栈的头结点,并将其next域置为NULL。对应算法如下:

```
void InitStack(LiStack *&s)
{
    s=(LiStack *)malloc(sizeof(LiStack));
    s->next=NULL;
}
```



# 链栈的基本运算



## ■ (2) 销毁栈ClearStack(&s)

- 释放栈s占用的全部存储空间。对应算法如下：

```
void ClearStack(LiStack *&s)
```

```
{   LiStack *p=s->next;
```

```
    while (p!=NULL)
```

```
    {   free(s);
```

```
        s=p;
```

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

```
    }
```

```
}
```

# 链栈的基本运算



- (3) 求栈的长度StackLength(s)
  - 从第一个数据结点开始扫描单链表,用i记录访问的数据结点个数,最后返回i值。对应算法如下:

```
int StackLength(LiStack *s)
{
    int i=0;
    LiStack *p;

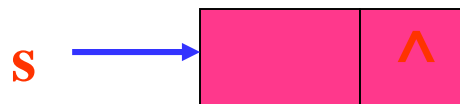
    p=s->next;
    while (p!=NULL)
    {
        i++;p=p->next;
    }
    return(i);
}
```

# 链栈的基本运算



- (4) 判断栈是否为空 StackEmpty(s)
  - 栈S为空的条件是  $s \rightarrow \text{next} == \text{NULL}$ , 即单链表中没有数据结点。对应算法如下:

```
int StackEmpty(LiStack *s)
{
    return(s->next==NULL);
}
```



# 链栈的基本运算



- (5) 进栈Push(&s,e)

- 将新数据结点插入到头结点之后。对应算法如下：

```
void Push(LiStack *&s,ElemType e)
```

```
{
```

```
    LiStack *p;
```

```
    p=(LiStack *)malloc(sizeof(LiStack));
```

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

```
    p->next=s->next; /*插入*p结点作为第一个数据结点*/
```

```
    s->next=p;
```

```
}
```

# 链栈的基本运算



- (6) 出栈Pop(&s,&e)
  - 在栈不为空的条件下,将头结点后继数据结点的数据域赋给e,然后将其删除。对应算法如下:

```
int Pop(LiStack *&s,ElemType &e)
{   LiStack *p;
    if (s->next==NULL) return 0; /*栈空的情况*/
    p=s->next;   /*p指向第一个数据结点*/
    e=p->data;
    s->next=p->next;
    free(p);
    return 1;
}
```

# 链栈的基本运算

---

- (7) 取栈顶元素GetTop(s)
  - 在栈不为空的条件下,将头结点后继数据结点的数据域赋给e。对应算法如下:

```
int GetTop(LiStack *s,ElemType &e)
{
    if (s->next==NULL) return 0;  /*栈空的情况*/
    e=s->next->data;
    return 1;
}
```

# 链栈的基本运算



- (8) 显示栈中元素DispStack(s)
  - 从第一个数据结点开始扫描单链表,并输出当前访问结点的数据域值。对应算法如下:

```
void DispStack(LiStack *s)
{
    LiStack *p=s->next;
    while (p!=NULL)
    {
        printf("%c ",p->data);
        p=p->next;
    }
    printf("\n");
}
```



# 【例】括号配对

---

- 假设表达式中允许包含三种括号:圆括号、方括号和大括号。编写一个算法判断表达式中的括号是否正确配对。
  - 解: 设置一个括号栈,扫描表达式: 遇到左括号(包括(、[和{)时进栈,遇到右括号时,若栈是相匹配的左括号,则出栈,否则,返回0。
  - 若表达式扫描结束,栈为空,返回1表示括号正确匹配,否则返回0。

# 【例】 括号配对

---

```
int correct(char exp[],int n)
{   char st[MaxSize];
    int top=-1,i=0,tag=1;
    while (i<n && tag)
    {   if (exp[i]=='(' || exp[i]=='[' || exp[i]=='{')
        /*遇到'(', '['或'{',则将其入栈*/
        {
            top++;
            st[top]=exp[i];
        }
        if (exp[i]==')')
            /*遇到),若栈顶是(,则继续处理,否则以不配对返回*/
```

# 【例】括号配对

---

```
if (st[top]=='(') top--;  
    else tag=0;  
    if (exp[i]=='') /*遇到], 若栈顶是[, 则继续,否则以不配对返回*/  
        if (st[top]=='[') top--;  
        else tag=0;  
    if (exp[i]=='}') /*遇到 '{',若栈顶是 '{',则继续处  
        理,否则以不配对返回*/  
        if (st[top]=='{') top--;  
        else tag=0;  
    i++;  
} /*表达式扫描完毕*/  
if (top>-1)  
    tag=0; /*若栈不空,则不配对*/  
return(tag);  
}
```

# 【例】进制转换 ---- 十进制到二进制

---

## ■ 1 数制转换


- 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:

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

( 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:

# 【例】进制转换 ---- 十进制到二进制

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

算法思想如下：当 $N > 0$ 时重复1，2

1. 若  $N \neq 0$ ，则将  $N \% r$  压入栈s中，执行2；  
若  $N = 0$ ，将栈s的内容依次出栈，算法结束。
2. 用  $N / r$  代替  $N$

# 【例】进制转换 ---- 十进制到二进制

## 算法一

```
typedef int datatype;
void conversion(int N, int r)
{
    SeqStack s;
    datatype x;
    Init_SeqStack(&s);
    while ( N )
    {
        Push_SeqStack ( &s , N % r );
        N=N / r ;
    }
    while (! Empty_SeqStack(&s) )
    {
        Pop_SeqStack (&s , &x );
        printf ( “ %d ”, x );
    }
}
```

# 【例】进制转换 ---- 十进制到二进制

---

算  
法  
二

```
#define L 10

void conversion(int N, int r)
{ int s[L],top;
  int x;
  top=-1;
  while ( N )
  { s[++top]=N%r;
    N=N / r ;
  }
  while (top!=-1)
  { x=s[top--];
    printf("%d",x);
  } }
```

# [例]利用栈，将二进制转换为十进制数

- 分析：每个二进制数转换成相应的十进制数方法如下：

$$(X_n X_{n-1} \dots X_3 X_2 X_1)_2 = X_1 * 2^0 + X_2 * 2^1 + \dots + X_n * 2^{(n-1)}$$

$$\begin{aligned} 110011_{(2)} &= 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \\ &= 1 \times 32 + 1 \times 16 + 1 \times 2 + 1 \\ &= 51 \end{aligned}$$

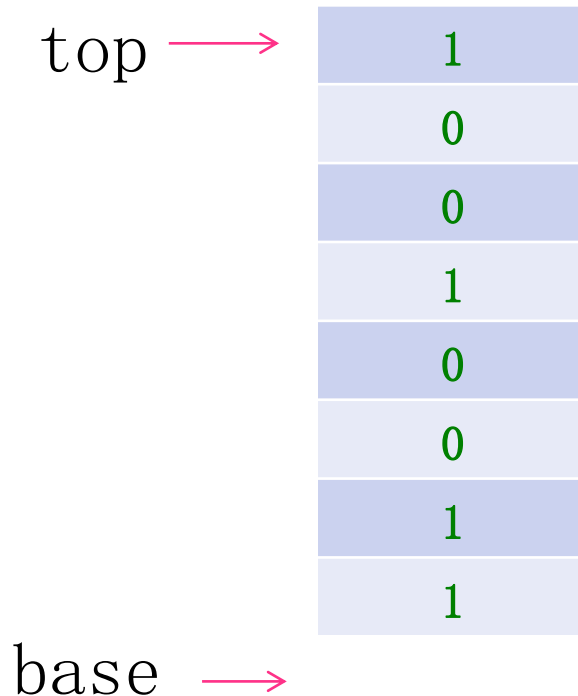
2	97	1
2	48	0
2	24	0
2	12	0
2	6	0
2	3	1
2	1	1
	0	

结果为：1100001



# [例]利用栈，将二进制转换为十进制数

- 由于栈具有后进先出的特性，输入二进制数11001001

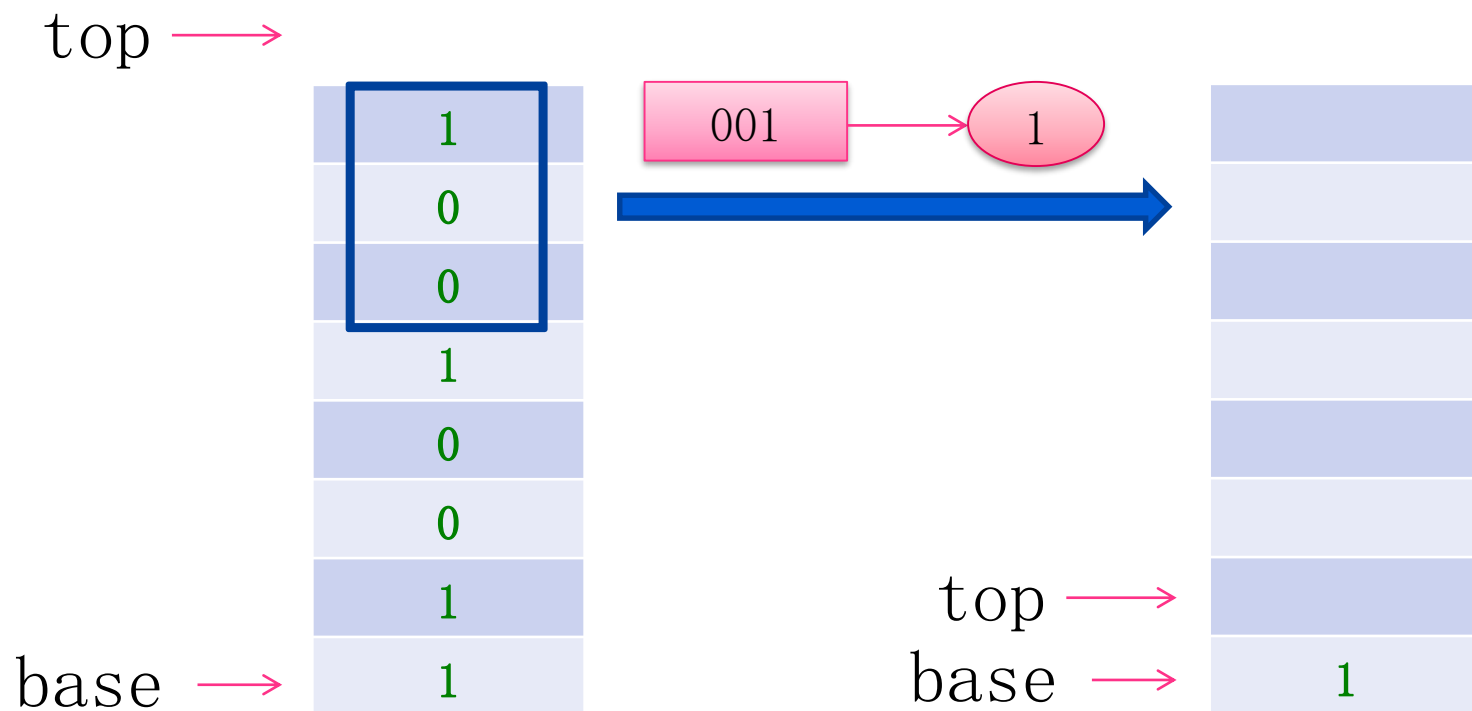


# 二进制到八进制

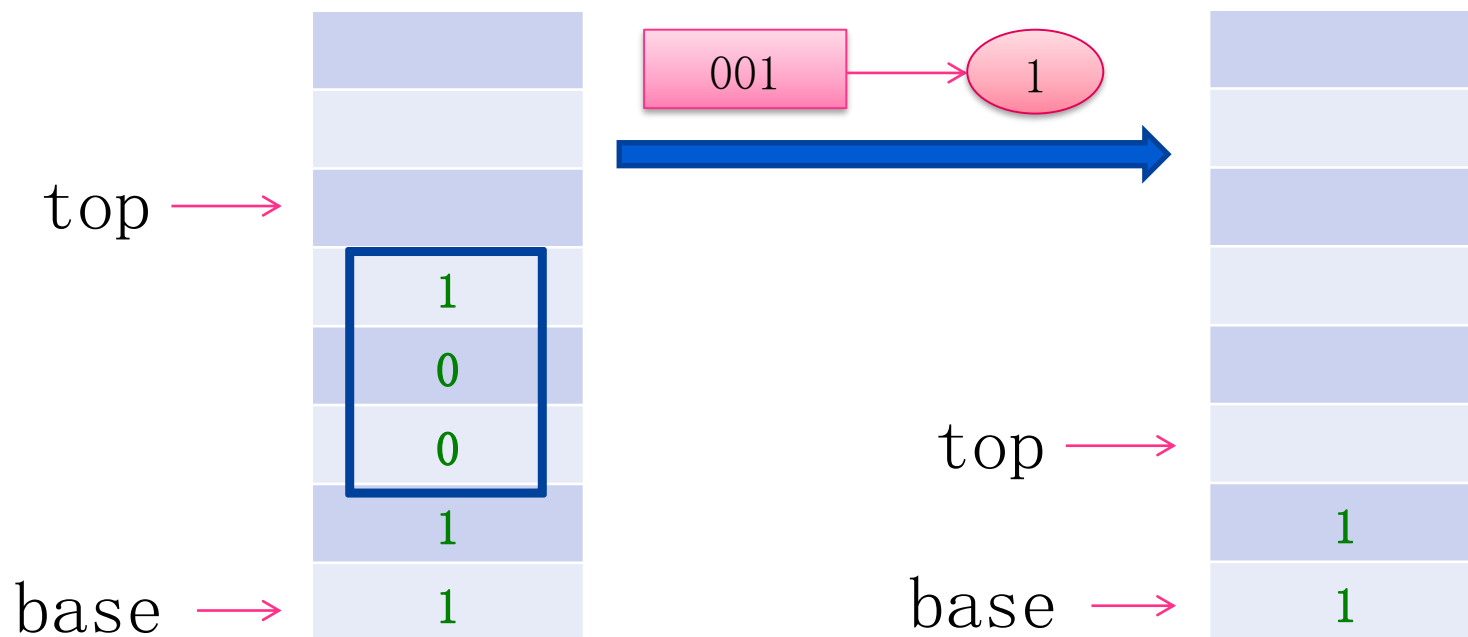
- 仔细观察二进制跟十六进制的对应关系
  - 可见一个字节用两个十六进制数可以表示完整，也大大的节省了显示空间。
  - 那八进制呢？因为早期的计算机系统都是三的倍数，所以用八进制比较方便。
  - 在进行二进制到八进制的转换时，要将二进制数的每三位抓换成一个八进制数来表示，然后按顺序输出即可。

0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

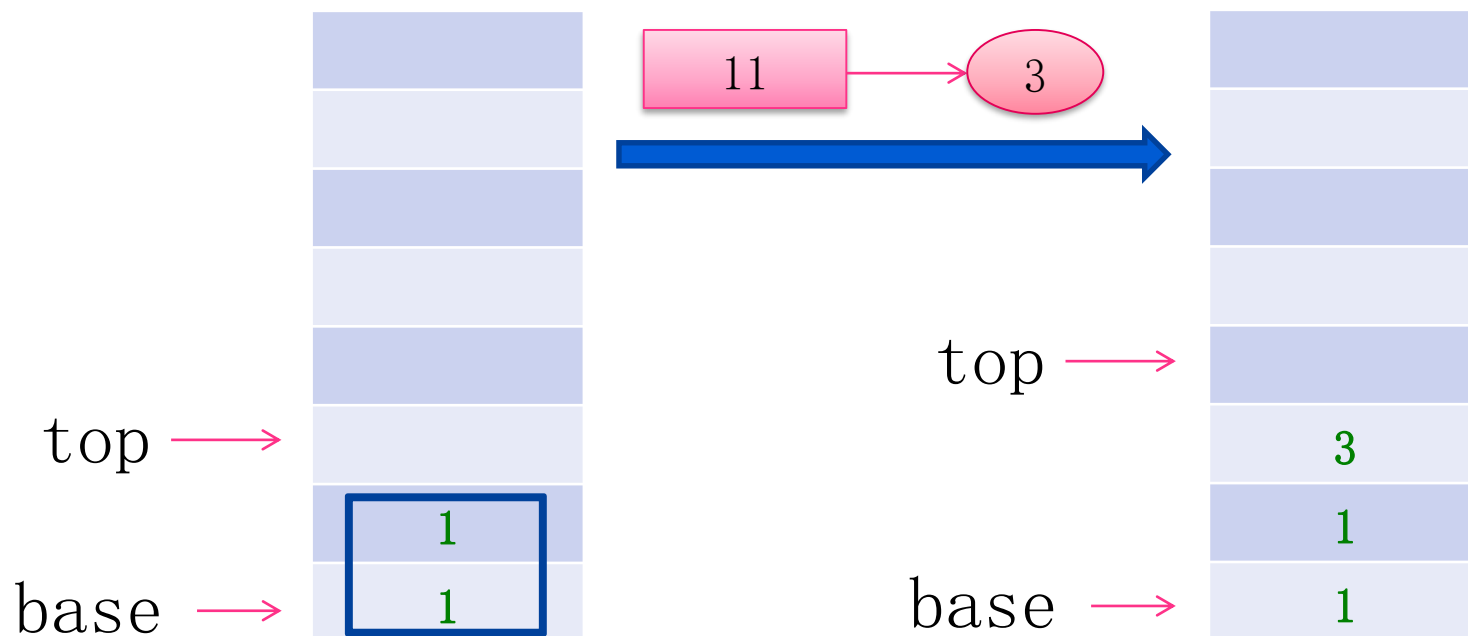
# 从二进制到八进制



# 从二进制到八进制



# 从二进制到八进制



## 3.1.4 栈的应用例子--- 表达式求值

---

### ■ 1. 表达式求值

- 这里限定的表达式求值问题是:用户输入一个包含“+”、“-”、“\*”、“/”、正整数和圆括号的合法数学表达式,计算该表达式的运算结果。

# 【例】中缀表达式 后缀表达式

---

- **中缀表达式**:在程序语言中,运算符位于两个操作数中间的表达式.例如:

$$1+2*3$$

- 遵循“先乘除,后加减,从左到右计算,先括号内,后括号外”的规则。
  - 因此,中缀表达式不仅要依赖运算符优先级,而且还要处理括号。

# 【例】中缀表达式 后缀表达式

---

- **后缀表达式**:运算符在操作数的后面
  - 如 $1+2*3$ 的后缀表达式为 $123*+$ 。
  - 在后缀表达式中已考虑了运算符的优先级,没有括号,只有操作数和运算符。
- 通常把它称为逆波兰表达式(RPN)



# 【例】中缀表达式求解

---

- 1. 中缀表达式求值：
  - 设运算规则为：
    - 运算符的优先级为：  $() \longrightarrow ^ \longrightarrow *, /, \% \longrightarrow +, -$  ；
    - 有括号出现时先算括号内的，后算括号外的，多层括号，由内向外进行；
    - 乘方连续出现时先算最右面的；

# 【例】中缀表达式求解

---

- 处理过程：
  - 需要两个栈：对象栈s1和算符栈s2。
  - 当自左至右扫描表达式的每一个字符时，若当前字符是运算对象，入对象栈，是运算符时，若这个运算符比栈顶运算符高则入栈，继续向后处理，若这个运算符比栈顶运算符低则从对象栈出栈两个运算量，从算符栈出栈一个运算符进行运算，并将其运算结果入对象栈，继续处理当前字符，直到遇到结束符。

# 【例】中缀表达式求解

每个运算符栈内、栈外的级别如下：

算符	栈内级别	栈外级别
$\wedge$	3	4
$*$ 、 $/$ 、 $\%$	2	2
$+$ 、 $-$	1	1
$($	0	4
$)$	-1	-1

## ■ 进栈的原则：

- 要保证栈顶的运算符的优先级最高，当遇到的运算符的优先级别若高于栈顶运算符的优先级别时，则进栈作为栈顶元素，否则，将栈顶元素退栈输出，然后再把优先级低的运算符进栈。

# 【例】 $A * (B - C) + D$ 的中缀表达式求值过程

	对象栈S1	算符栈S2
A	A	
*	A	*
(	A	*(
B	A B	*(
-	A B	*(-
C	A B C	*(-
)	A B-C	*(
	$A * (B - C)$	

# 【练习】中缀表达式求解

---

- 用中缀表达式法求表达式
- $3*2^{\wedge}(4+2*2-1*3)-5$  的值

$$3*2^{\left(4+2*2-1*3\right)}-5$$

3	3	(	3入栈s1
*	3	(*	*入栈s2
2	3, 2	(*	2入栈s1
^	3, 2	(*^	^入栈s2
(	3, 2	(*^(	(入栈s2
4	3, 2, 4	(*^(	4入栈s1
+	3, 2, 4	(*^(+	+入栈s2
2	3, 2, 4, 2	(*^(+	2入栈s1
*	3, 2, 4, 2	(*^(+*	*入栈s2
2	3, 2, 4, 2, 2	(*^(+*	2入栈s1
-	3, 2, 4, 4	(*^(+	做2+2=4, 结果入栈s1
	3, 2, 8	(*^(	做4+4=8, 结果入栈s2
	3, 2, 8	(*^(-	-入栈s2

1	3, 2, 8, 1	(*^(-	1入栈s1
*	3, 2, 8, 1	(*^(-*	*入栈s2
3	3, 2, 8, 1, 3	(*^(-*	3入栈s1
)	3, 2, 8, 3	(*^(-	做1*3, 结果3入栈s1
	3, 2, 5	(*^(	做8-3, 结果5入栈s2
	3, 2, 5	(*^	( 出栈
-	3, 32	(*	做2^5, 结果32入栈s1
	96	(	做3*32, 结果96入栈s1
	96	(-	-入栈s2
5	96, 5	(-	5入栈s1
结束符	91	(	做96-5, 结果91入栈s1

# 【例】逆波兰表达式

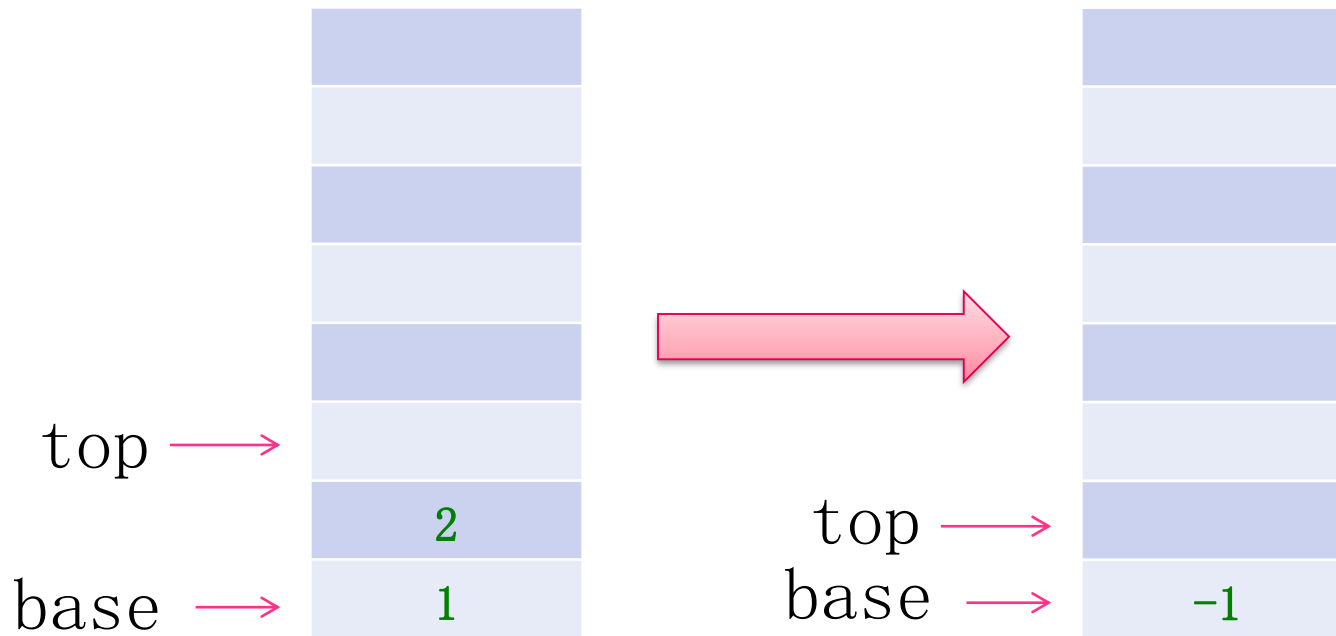
---

- $(1-2)*(4+5)$ , 用逆波兰表示法:  $1\ 2\ -\ 4\ 5\ +\ *$



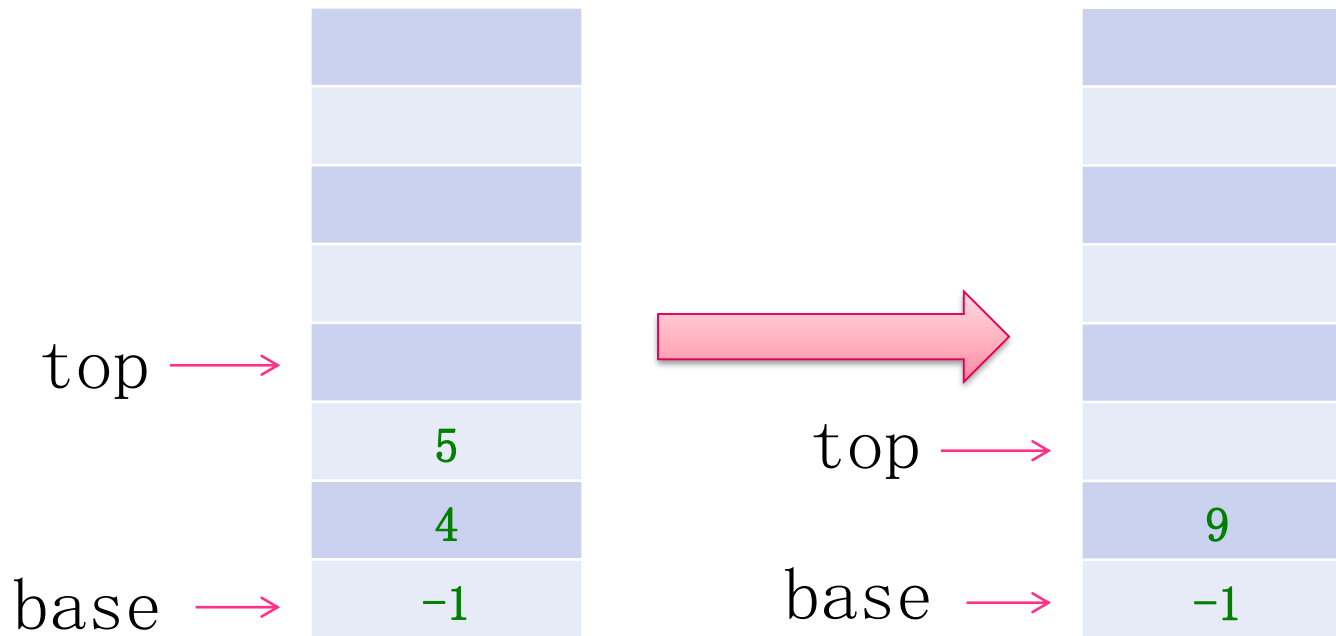
# 【例】逆波兰表达式

- $(1-2)*(4+5)$ ，用逆波兰表示法： $1\ 2\ -\ 4\ 5\ +\ *$
- 数字1和2进栈，遇到减号运算符则弹出两个元素进行运算并把结果入栈。



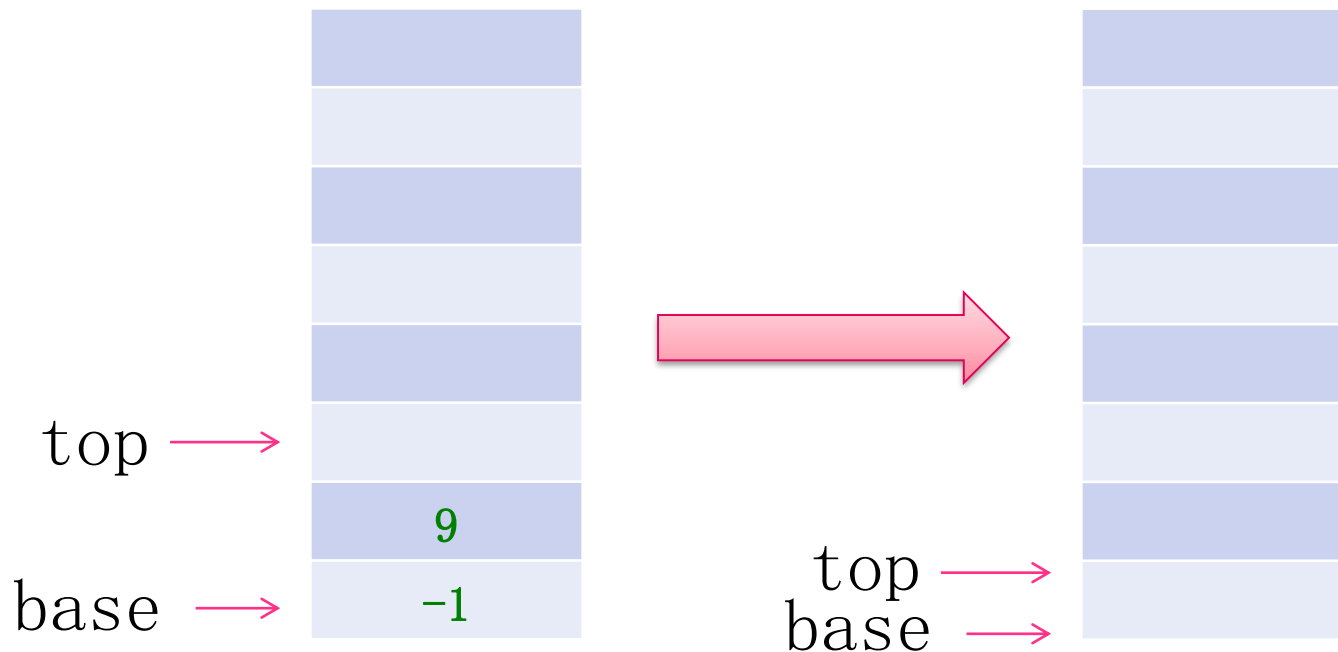
# 【例】逆波兰表达式

- $(1-2)*(4+5)$ ，用逆波兰表示法： $1\ 2\ -\ 4\ 5\ +\ *$
- 4和5入栈，遇到加号运算符，4和5弹出栈，相加后将结果9入栈。



# 【例】逆波兰表达式

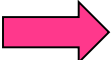
- $(1-2)*(4+5)$ ，用逆波兰表示法： $1\ 2\ -\ 4\ 5\ +\ *$
- 然后又遇到乘法运算符，将9和-1弹出栈进行乘法计算，此时栈空并无数据压栈，-9为最终运算结果！




# 【例】中缀表达式转换为后缀表达式

---

## ■ 中缀表达式 ---> 逆波兰表达式

$a+b$    $a b +$

$a+(b-c)$    $a b c - +$

$a+(b-c)*d$    $a b c - d * +$

$a+d*(b-c)$    $a d b c - * +$

# 【例】中缀表达式转换为后缀表达式

---

- 在后缀表达式中已考虑了运算符的优先级,没有括号,只有操作数和运算符。
  - **具体做法:** 只使用一个对象栈,当从左向右扫描表达式时,每遇到一个操作数就送入栈中保存,每遇到一个运算符就从栈中取出两个操作数进行当前的计算,然后把结果再入栈,直到整个表达式结束,这时送入栈顶的值就是结果。

# 【例】中缀表达式转换为后缀表达式

$A*(B-C)+D$

	对象栈S1	输出
A	^	A
*	*	A
(	*(	A
B	*(	AB
-	*(-	AB
C	*(-	ABC
)	*	ABC-
+	+	ABC-*
D	+	ABC-*D
完成	^	ABC-*D+

# 【练习】中缀表达式转换为后缀表达式

---

- 求  $3 * 2^{\wedge} (4 + 2 * 2 - 1 * 3) - 5$  的后缀表达式法

$$3*2^{\wedge}(4+2*2-1*3)-5$$

3	3	3 入栈
2	3, 2	2 入栈
4	3, 2, 4	4 入栈
2	3, 2, 4, 2	2 入栈
2	3, 2, 4, 2, 2	2 入栈
*	3, 2, 4, 4	计算 $2 * 2$ ，将结果 4 入栈
+	3, 2, 8	计算 $4 + 4$ ，将结果 8 入栈
1	3, 2, 8, 1	1 入栈
3	3, 2, 8, 1, 3	3 入栈
*	3, 2, 8, 3	计算 $1 * 3$ ，将结果 4 入栈
-	3, 2, 5	计算 $8 - 5$ ，将结果 5 入栈
^	3, 32	计算 $2^5$ ，将结果 32 入栈
*	96	计算 $3 * 32$ ，将结果 96 入栈
5	96, 5	5 入栈
-	96	计算 $96 - 5$ ，结果入栈
结束符	空	结果出栈



# [例]求解迷宫问题

- 求迷宫问题就是求出从入口到出口的路径。在求解时,通常用的是“**穷举求解**”的方法,即从入口出发,顺某一方向向前试探,若能走通,则继续往前走;否则沿原路退回,换一个方向再继续试探,直至所有可能的通路都试探完为止。为了保证在任何位置上都能沿原路退回(称为**回溯**),需要用一个后进先出的栈来保存从入口到当前位置的路径。
  - 首先用如图所示的方块图表示迷宫。对于图中的每个方块,用空白表示通道,用阴影表示墙。所求路径必须是简单路径,即在求得的路径上不能重复出现同一通道块。

# [例]求解迷宫问题

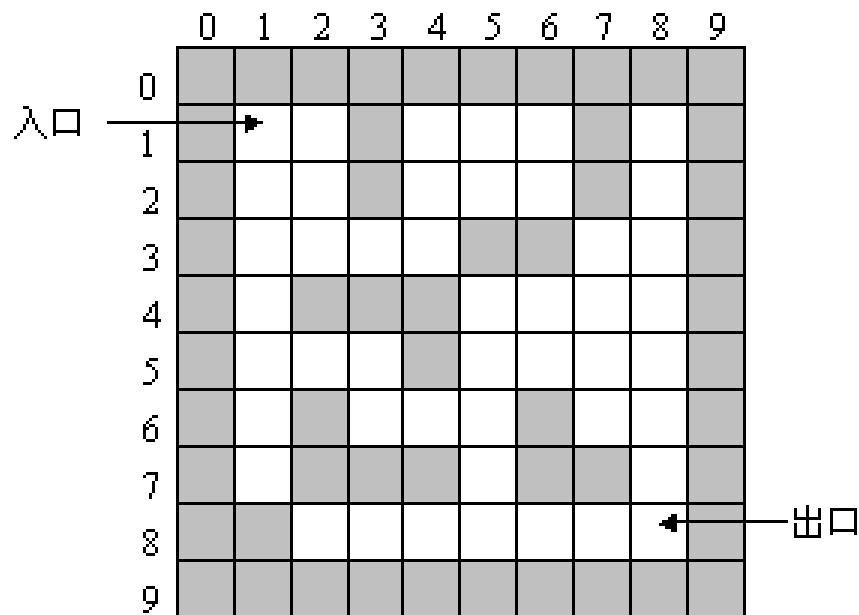


图 3.3 迷宫示意图

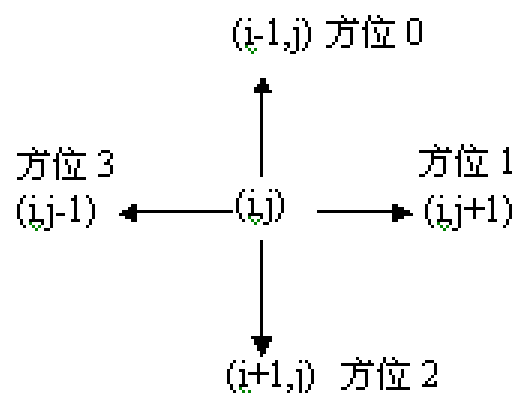
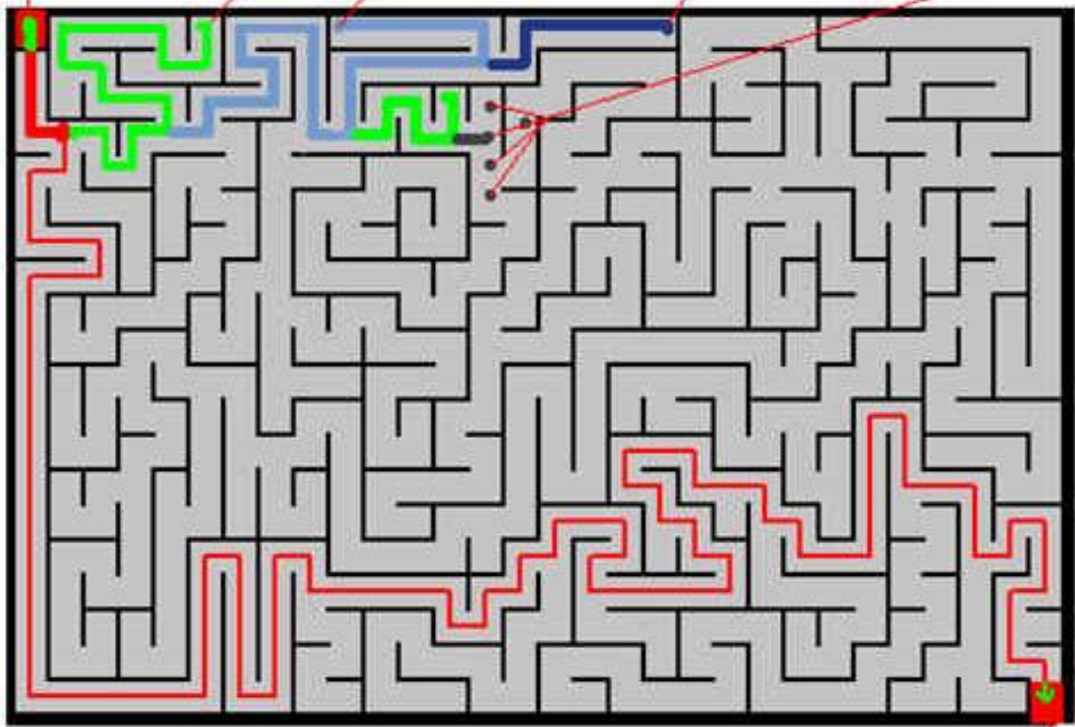


图 3.4 方位图

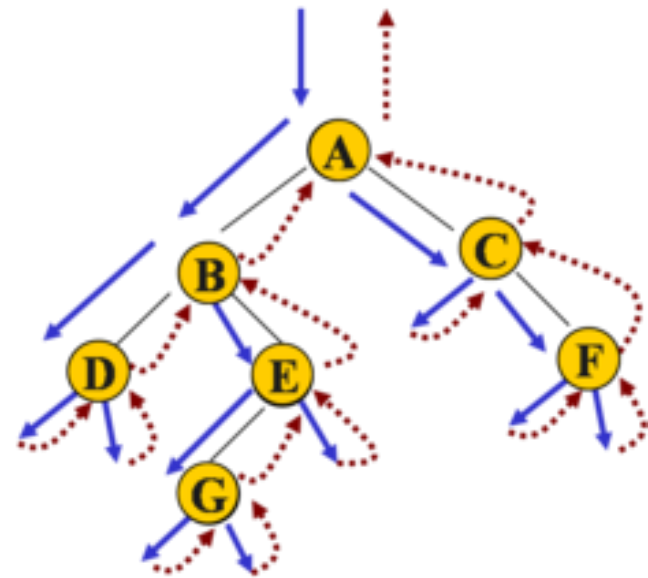
## [例]求解迷宫问题

### ■ 例：迷宫游戏

开始	第一次回朔	第二次回朔	第三次回朔	其他回朔



## 结束



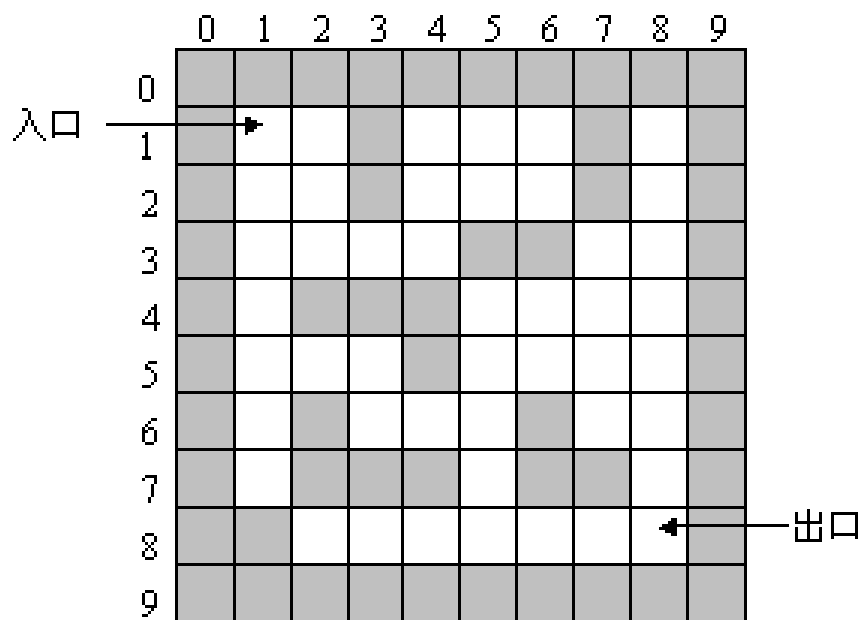
# 求解迷宫问题

---

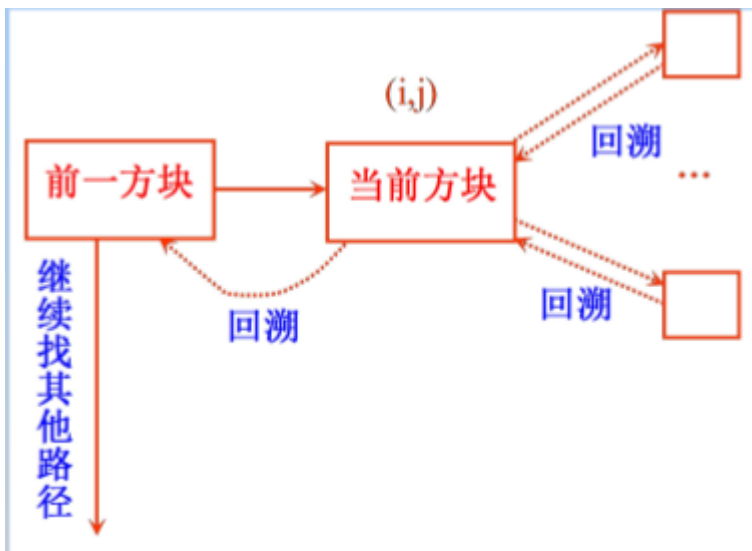
- 关键：
  - 保证在任何位置上都能沿原路退回（回溯）
  - 需要用一個后进先出的栈来保存从入口到当前位置的路径

# [例]求解迷宫问题

- 为了表示迷宫,设置一个数组mg,其中每个元素表示一个方块的状态,为0时表示对应方块是通道,为1时表示对应方块为墙,
- ```
int mg[M+1][N+1]={ /*M=10,N=10*/  
    {1,1,1,1,1,1,1,1,1,1},  
    {1,0,0,1,0,0,0,1,0,1},  
    {1,0,0,1,0,0,0,1,0,1},  
    {1,0,0,0,0,1,1,0,0,1},  
    {1,0,1,1,1,0,0,0,0,1},  
    {1,0,0,0,1,0,0,0,0,1},  
    {1,0,1,0,0,0,1,0,0,1},  
    {1,0,1,1,1,0,1,1,0,1},  
    {1,1,0,0,0,0,0,0,0,1},  
    {1,1,1,1,1,1,1,1,1,1}};
```



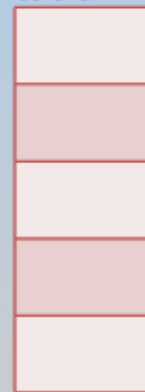
# [例]求解迷宫问题



```
typedef struct
{
    int i; //当前方块的行号
    int j; //当前方块的列号
    int di; //di是下一可走相邻方位的方位号
} Box; //定义方块类型

typedef struct
{
    Box data[MaxSize];
    int top; //栈顶指针
} StType; //顺序栈类型
```

试探轨迹栈



# 队列

■ **队列的定义**

■ **队列的顺序存储结构及其基本运算的实现**

■ **队列的链式存储结构及其基本运算的实现**

■ **队列的应用例子**



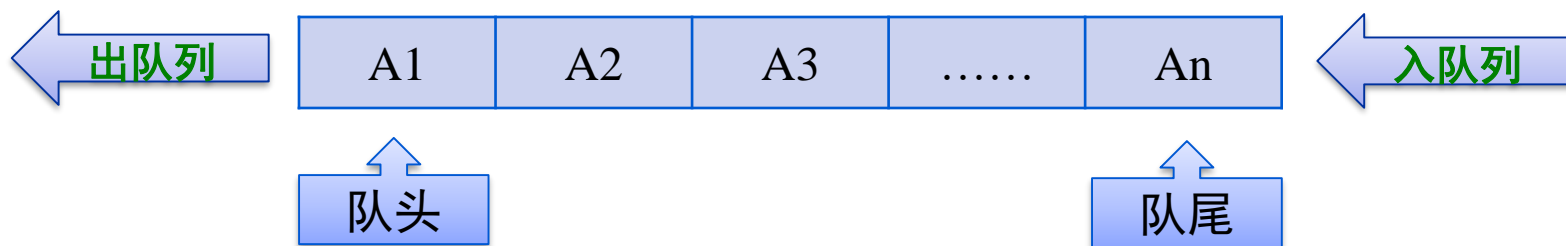
## 3.2.1 队列的定义

### ■ 队列？

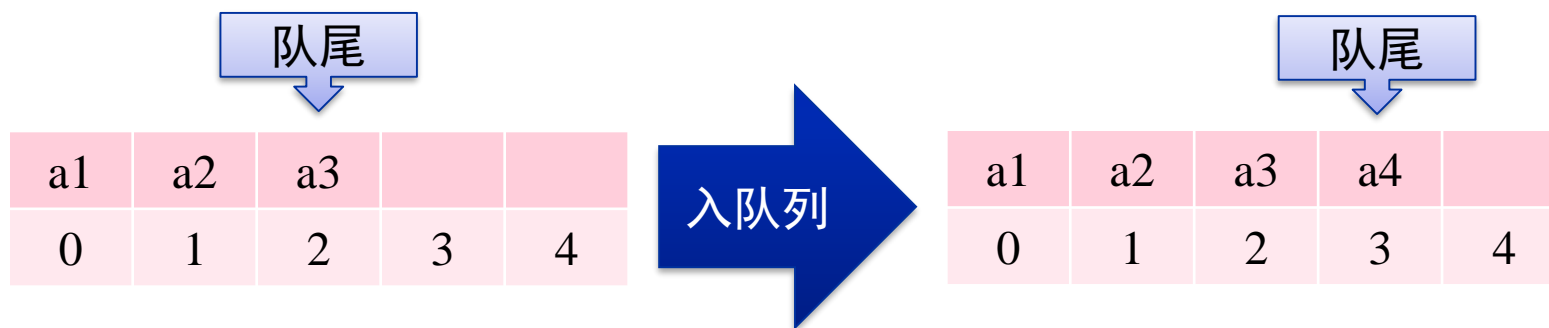


## 3.2.1 队列的定义

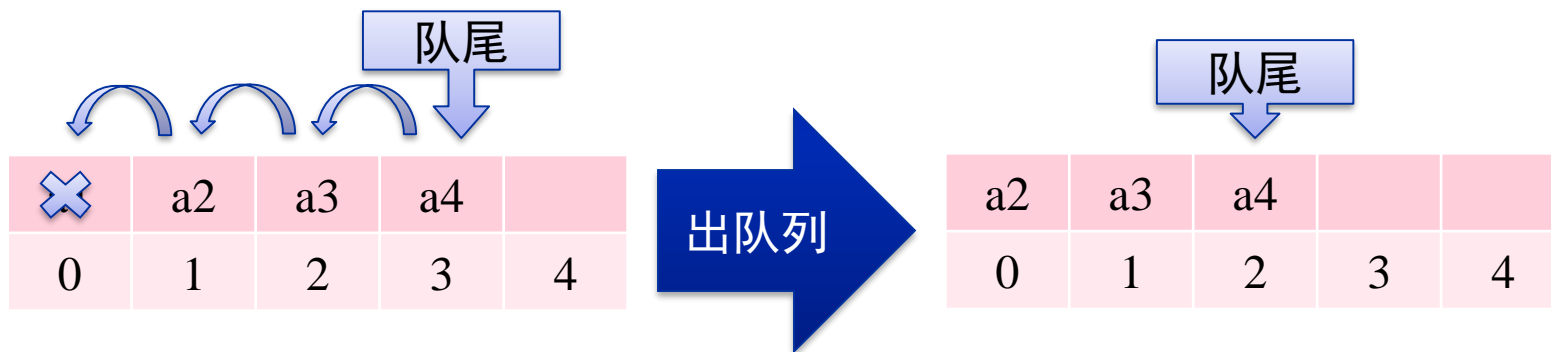
- 队列简称队,也是一种**运算受限**的线性表,其限制仅允许在表的一端进行**插入**,而在表的另一端进行**删除**。
- 我们把进行插入的一端称做**队尾(rear)**,进行删除的一端称做**队首(front)**。
- 向队列中插入新元素称为**进队**或**入队**,新元素进队后就成为新的队尾元素;从队列中删除元素称为**出队**或**离队**,元素出队后,其后继元素就成为队首元素。



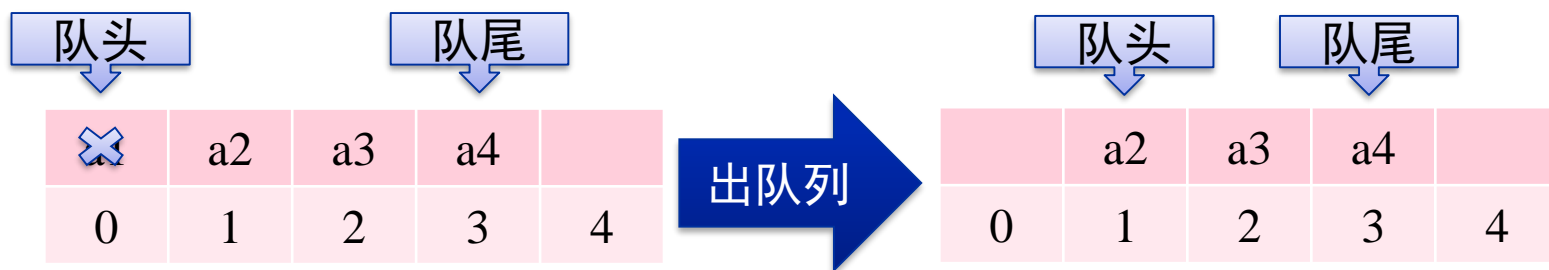
# 队列的顺序存储方案（一）



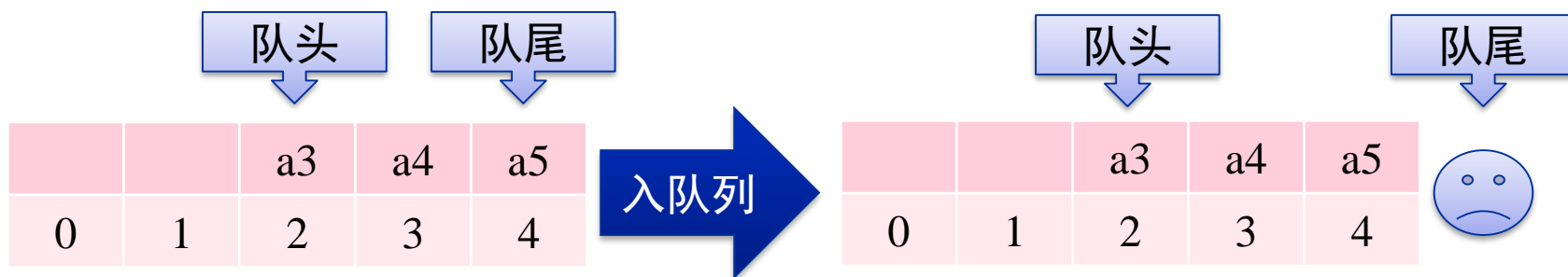
入队列操作其实就是在队尾追加一个元素，不需要任何移动，时间复杂度为 $O(1)$ 。出队列则不同，因为现在假设的是下标为0的位置是队列的队头，因此每次出队列操作所有元素都要向前移动。



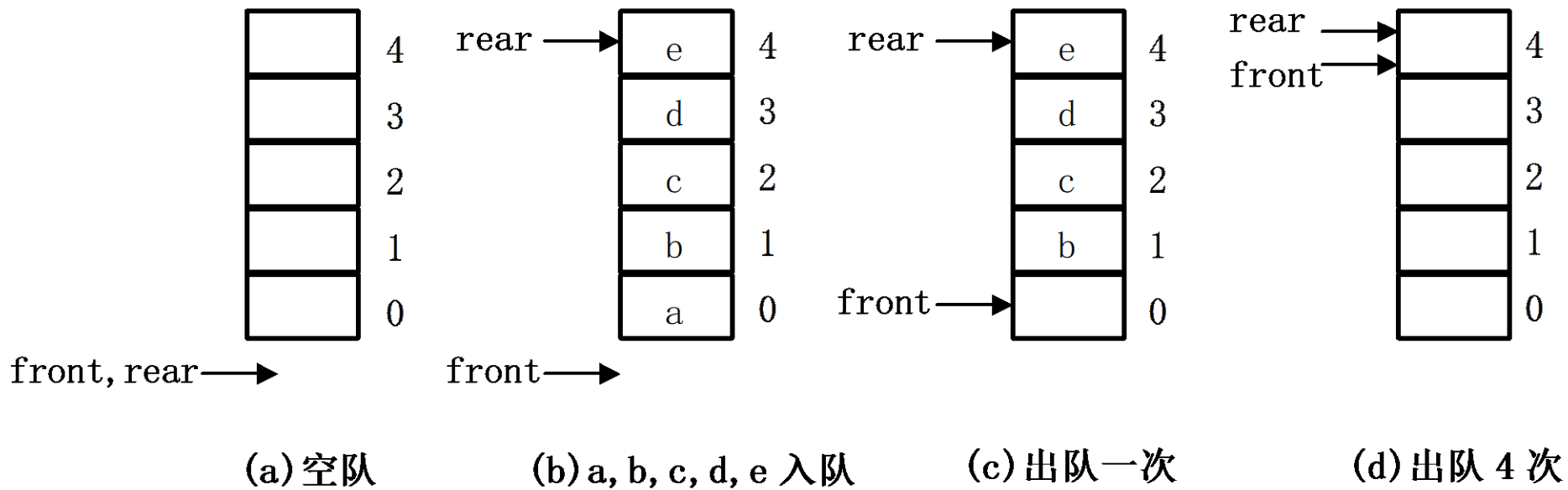
# 队列的顺序存储方案（二）



- 但是这样也会出现一些问题，例如按下边的情形继续入队列，就会出现数组越界的错误。



- 可事实上我们有0和1两个下标还空着，这叫假溢出。



## 队列的入队和出队操作示意图

## 3.2.2 队列的顺序存储及其基本运算的实现

- 假设队列的元素个数最大不超过整数MaxSize,所有的元素都具有同一数据类型ElemType,则顺序队列类型SqQueue定义如下:

```
typedef struct
```

```
{    ElemType data[MaxSize];
```

```
    int front,rear;                /*队首和队尾指针*/
```

```
} SqQueue
```

- 图(a)为队列的初始状态,有 $\text{front} == \text{rear}$ 成立,该条件可以作为队列空的条件。
- 那么能不能用 $\text{rear} == \text{MaxSize} - 1$ 作为队满的条件呢?
  - 显然不能,在图(d)中,队列为空,但仍满足该条件。这时入队时出现“上溢出”,这种溢出并不是真正的溢出,在`elem`数组中存在可以存放元素的空位置,所以这是一种假溢出。
- 为了能够充分地使用数组中的存储空间,把数组的**前端和后端连接起来**,形成一个环形的顺序表,即把存储队列元素的表从逻辑上看成一个环,称为**循环队列**。

# 队列的顺序存储方案（三）---循环队

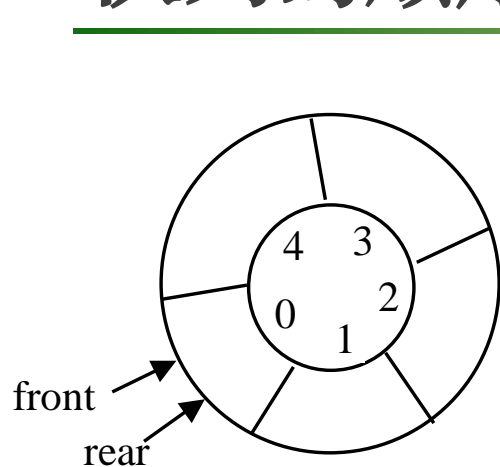
- 循环队列首尾相连,当队首front指针满足 $\text{front}=\text{MaxSize}-1$ 后,再前进一个位置就自动到0,这可以利用除法取余的运算( $\%$ )来实现:
  - 队首指针进1: $\text{front}=(\text{front}+1)\% \text{MaxSize}$
  - 队尾指针进1: $\text{rear}=(\text{rear}+1)\% \text{MaxSize}$
  - 循环队列的除头指针和队尾指针初始化时都置0: $\text{front}=\text{rear}=0$ 。在入队元素和出队元素时,指针都按逆时针方向进1。



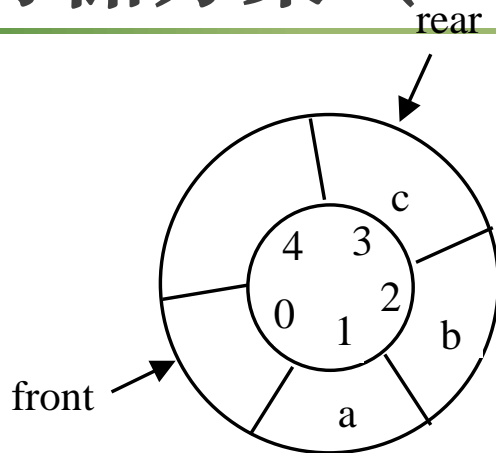
# 队列的顺序存储方案（三）---循环队

- 怎样区分这两者之间的差别呢?在入队时少用一个数据元素空间,以队尾指针加1等于队首指针判断队满,即队满条件为:
  - $(q \rightarrow rear + 1) \% \text{MaxSize} == q \rightarrow front$
- 队空条件仍为:
  - $q \rightarrow rear == q \rightarrow front$

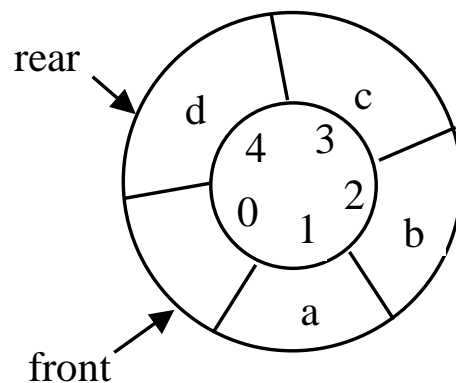
# 队列的顺序存储方案（三）---循环队



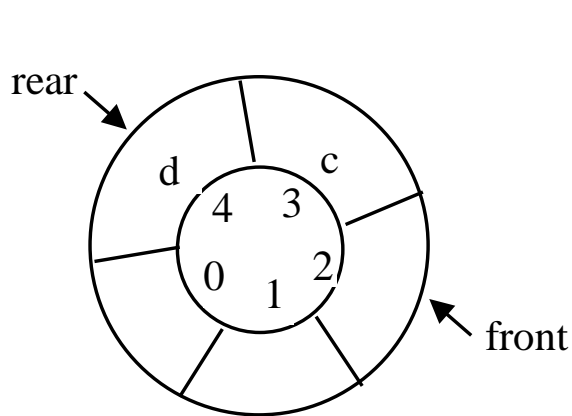
(a)空队



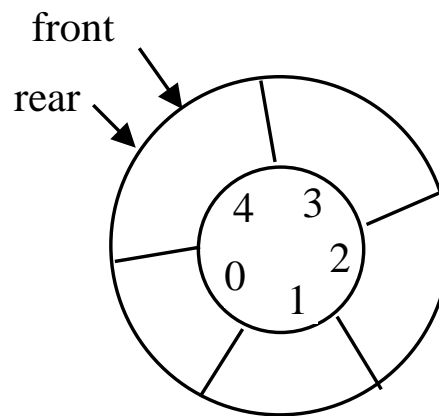
(b)a,b,c 入队



(c)d 入队,队满



(d)出队 2 次



(e)出队 2 次,队空

循环队的入队和出队操作示意图

# 循环队的基本运算

---

- (1) 初始化队列InitQueue(&q)
  - 构造一个空队列q。将front和rear指针均设置成初始状态即0值。对应算法如下：

```
void InitQueue(SqQueue *&q)
```

```
{
```

```
    q=(SqQueue *)malloc (sizeof(SqQueue));
```

```
    q->front=q->rear=0;
```

```
}
```

# 循环队的基本运算

---

- (2) 销毁队列ClearQueue(&q)

- 释放队列q占用的存储空间。对应算法如下：

```
void ClearQueue(SqQueue *&q)  
{  
    free(q);  
}
```

# 循环队的基本运算

---

- (3) 判断队列是否为空QueueEmpty(q)
  - 若队列q满足 $q \rightarrow \text{front} == q \rightarrow \text{rear}$ 条件,则返回1; 否则返回0。对应算法如下:

```
int QueueEmpty(SqQueue *q)
{
    return(q->front==q->rear);
}
```

# 循环队的基本运算

- (4) 入队列enQueue(q,e)
  - 在队列不满的条件下,先将队尾指针rear循环增1,然后将元素添加到该位置。对应算法如下:

```
int enQueue(SqQueue *&q,ElemType e)
{
    if ((q->rear+1)%MaxSize==q->front) /*队满*/
        return 0;
    q->rear=(q->rear+1)%MaxSize;
    q->data[q->rear]=e;
    return 1;
}
```

# 循环队的基本运算

- (5) 出队列deQueue(q,e)
  - 在队列q不为空的条件下,将队首指针front循环增1,并将该位置的元素值赋给e。对应算法如下:

```
int deQueue(SqQueue *&q,ElemType &e)
{
    if (q->front==q->rear) /*队空*/
        return 0;
    q->front=(q->front+1)%MaxSize;
    e=q->data[q->front];
    return 1;
}
```

# 【思考】上溢现象和假溢出现象

- 什么是队列的上溢现象和假溢出现象？解决它们有哪些方法？
  - 答：在队列的顺序存储结构中,设头指针为front,队尾指针rear,队的容量(存储空间的大小)为MaxSize。当有元素加入到队列时,若  $\text{rear}=\text{MaxSize}$ (初始时 $\text{rear}=0$ )则发生队列的**上溢现象**,该元素不能加入队列。
  - 特别要注意的是队列的**假溢出现象**:队列中还有剩余空间但元素却不能进入队列,造成这种现象的原因是由于队列的操作方法所致。



# 解决队列上溢的方法

---

- 解决队列上溢的方法有以下几种:
  - (1) 建立一个足够大的存储空间,但这样做会造成空间的使用效率降低。
  - (2) 当出现假溢出时可采用以下几种方法:
    - ①采用平移元素的方法:每当队列中加入一个元素时,队列中已有的元素向队头移动一个位置(当然要有空闲的空间可供移动);

# 解决队列上溢的方法

---

- ②每当删除一个队头元素时,则依次移动队中的元素,始终使front指针指向队列中的第一个位置;
- ③采用循环队列方式:把队列看成一个首尾相接的循环队列,在循环队列上进行插入或删除运算时仍然遵循“先进先出”的原则。

# 【例】

---

- 对于顺序队列来说,如果知道队首元素的位置和队列中元素个数,则队尾元素所在位置显然是可以计算的。也就是说,可以用队列中元素个数代替队尾指针。编写出这种循环顺序队列的初始化、入队、出队和判空算法。

- 解: 当已知队首元素的位置`front`和队列中元素个数`count`后:

队空的条件为:`count==0`

队满的条件为:`count==MaxSize`

计算队尾位置`rear`:

`rear=(front+count)%MaxSize`

- 
- 对应的算法如下:

```
typedef struct
```

```
{   ElemType data[MaxSize];
```

```
    int front;           /*队首指针*/
```

```
    int count;          /*队列中元素个数*/
```

```
} QuType;              /*队列类型*/
```

---

```
void InitQu(QuType *&q)    /*队列q初始化*/
```

```
{
```

```
    q=(QuType *)malloc(sizeof(QuType));
```

```
    q->front=0;
```

```
    q->count=0;
```

```
}
```

---

```
int EnQu(QuType *&q,ElemType x)    /*进队*/
{   int rear;
    if (q->count==MaxSize)    return 0; /*队满上溢出*/
    else
    {   rear=(q->front+q->count+MaxSize)%MaxSize;
        /*求队尾位置*/
        rear=(rear+1)%MaxSize;    /*队尾位置进1*/
        q->data[rear]=x;
        q->count++;
        return 1;
    }
}
```

---

```
int DeQu(QuType *&q,ElemType &x)    /*出队*/
{
    if (q->count==0)    /*队空下溢出*/
        return 0;
    else
    {
        q->front=(q->front+1)%MaxSize;
        x=q->data[q->front];
        q->count--;
        return 1;
    }
}
```

---

```
int QuEmpty(QuType *q)  /*判空*/  
{  
    return(q->count==0);  
}
```



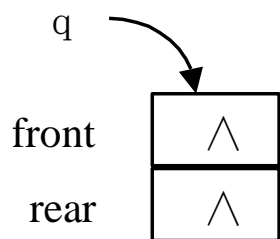
## 3.2.3 队列的链式存储及其基本运算的实现

---

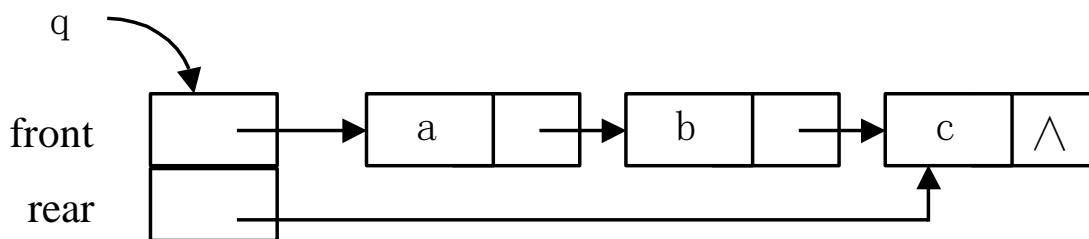
- 链队组成:

- (1) 存储队列元素的单链表
- (2) 指向队头和队尾指针的链队头结点

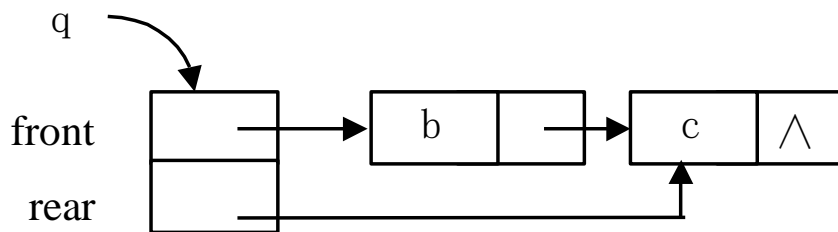




(a) 链队初态



(b) 入队 3 个元素



(c) 出队 1 个元素

## 链列的入队和出队操作示意图

- 
- 单链表中数据结点类型QNode定义如下:

```
typedef struct qnode
```

```
{    ElemType data;    /*数据元素*/
```

```
    struct qnode *next;
```

```
} QNode;
```

- 链队中头结点类型LiQueue定义如下:

```
typedef struct
```

```
{    QNode *front;    /*指向单链表队头结点*/
```

```
    QNode *rear;    /*指向单链表队尾结点*/
```

```
} LiQueue;
```

# 链队的基本运算:

---

- (1) 初始化队列InitQueue(q)
  - 构造一个空队列,即只创建一个链队头结点,其front和rear域均置为NULL,不创建数据元素结点。对应算法如下:

```
void InitQueue(LiQueue *&q)
{
    q=(LiQueue *)malloc(sizeof(LiQueue));
    q->front=q->rear=NULL;
}
```

# 链队的基本运算:

- (2) 销毁队列ClearQueue(q)
  - 释放队列占用的存储空间,包括链队头结点和所有数据结点的存储空间。对应算法如下:

```
void ClearQueue(LiQueue *&q)
{
    QNode *p=q->front,*r;
    if (p!=NULL)                                /*释放数据结点占用空间*/
    {
        r=p->next;
        while (r!=NULL)
        {
            free(p);
            p=r;r=p->next;                        /*p和r指针同步后移*/
        }
    }
    free(q);                                    /*释放链队结点占用空间*/
}
```

# 链队的基本运算:

---

- (3) 判断队列是否为空QueueEmpty(q)
  - 若链队结点的rear域值为NULL,表示队列为空,返回1; 否则返回0。对应算法如下:

```
int QueueEmpty(LiQueue *q)
{
    if (q->rear==NULL)
        return 1;
    else
        return 0;
}
```

# 链队的基本运算:

---

- (4) 入队列enQueue(q,e)
  - 创建data域为e的数据结点\*s。若原队列为空,则将链队结点的两个域均指向\*s结点,否则,将\*s链到单链表的末尾,并让链队结点的rear域指向它。对应算法如下:

# 链队的基本运算:

---

```
void enQueue(LiQueue *&q,ElemType e)
{
    QNode *s;
    s=(QNode *)malloc(sizeof(QNode));
    s->data=e;
    s->next=NULL;
    if (q->rear==NULL)
        /*若原链队为空,新结点是队首结点又是队尾结点*/
        q->front=q->rear=s;
    else
    {
        q->rear->next=s;
        /*将*s结点链到队尾,rear指向它*/
        q->rear=s;
    }
}
```



# 链队的基本运算:

---

- (5) 出队列deQueue(q,e)
  - 若原队列不为空,则将第一个数据结点的data域值赋给e,并删除之。若出队之前队列中只有一个结点,则需将链队结点的两个域均置为NULL,表示队列已为空。对应的算法如下:

# 链队的基本运算:

---

```
int deQueue(LiQueue *&q, ElemType &e)
{
    QNode *t;
    if (q->rear==NULL) return 0;          /*队列为空*/
    t=q->front;                             /*t指向第一个数据结点*/
    if (q->front==q->rear)
        /*原链队中只有一个结点时*/
        q->front=q->rear=NULL;
    else
        /*原链队中有多个结点时*/
        q->front=q->front->next;
    e=t->data;
    free(t);
    return 1;
}
```

- 
- 队列是以后学习广度优先搜索以及队列优化Bellman-Ford最短路算法的核心数据结构

---

## 本章小结

本章基本学习要点如下：

(1) 理解栈和队列的特性以及它们之间的差异,知道在何时使用哪种数据结构。

(2) 重点掌握在顺序栈上和链栈上实现栈的基本运算算法,注意栈满和栈空的条件。

---

(3) 重点掌握在顺序队上和链队上实现队列的基本运算算法,注意循环队上队满和队空的条件。

(4) 灵活运用栈和队列这两种数据结构解决一些综合应用问题。

