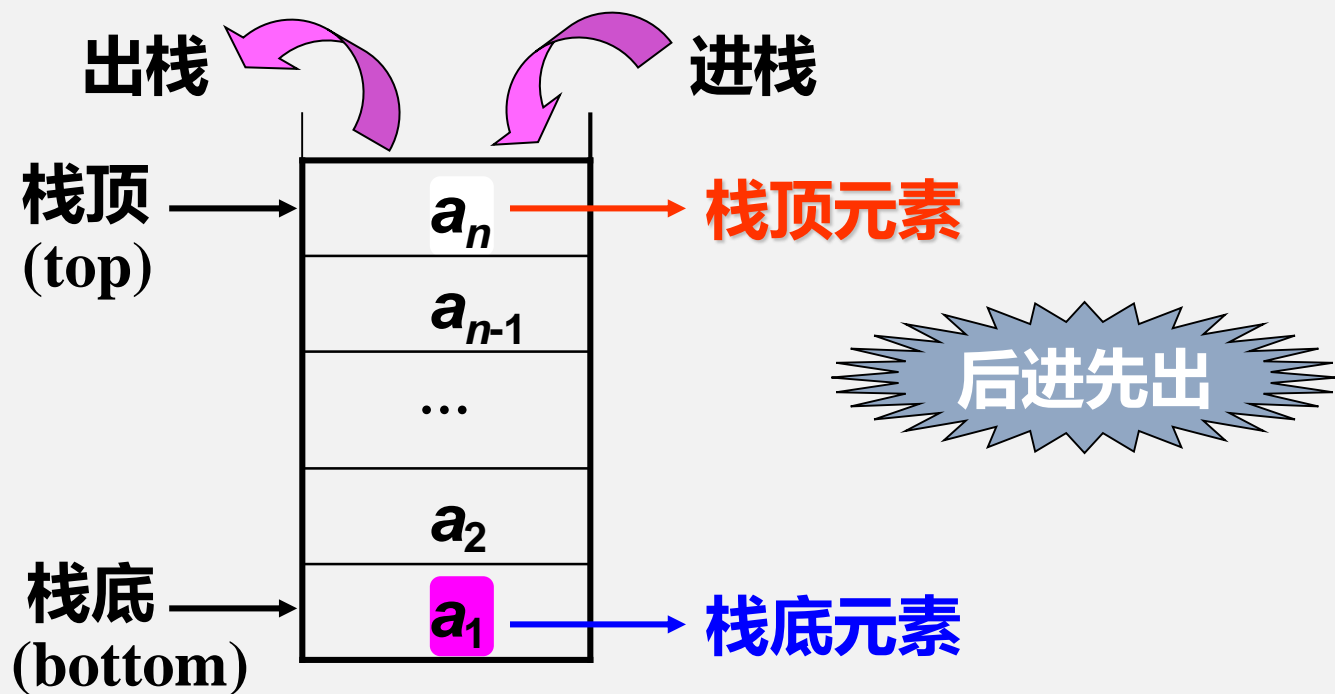




## 3.4.1 栈的定义

● **栈的定义：** 限定只能在表的同一端进行插入和删除运算的线性表（只能在栈顶操作）



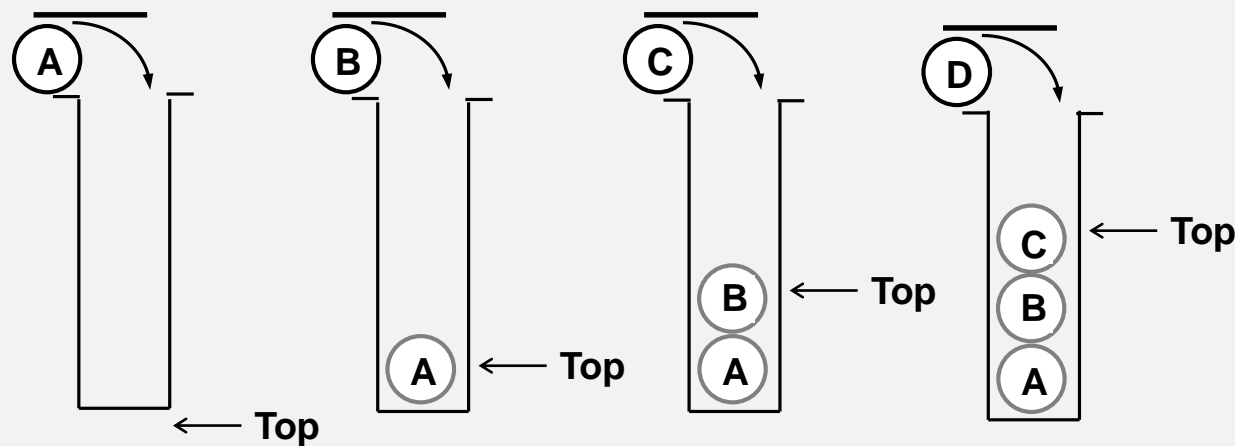
**类型名称:** 栈 (Stack)

**数据对象集:** 一个有0个或多个元素的有穷线性表。

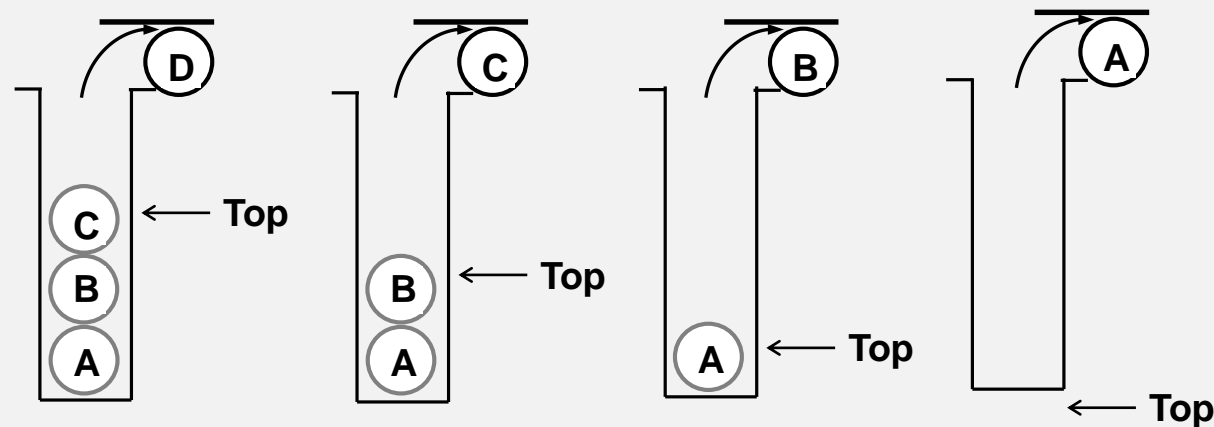
**操作集:** 对于一个具体的长度为正整数MaxSize的栈  $S \in \text{Stack}$ , 记栈中的任一元素  $\text{item} \in \text{ElementType}$ 。

- **Stack CreateStack( int MaxSize ):** **创建空栈**, 其最大长度为MaxSize;
- **int IsFull( Stack S, int MaxSize ):** **判断栈S是否已满**, 若S中元素个数等于MaxSize时返回1(TRUE); 否则返回0(FALSE);
- ✶ **void Push( Stack S, ElementType item ):** **入栈操作**, 将元素item压入栈。若栈已满, 返回堆栈为满信息; 否则将数据元素item插入到堆栈S栈顶处;
- **int IsEmpty ( Stack S ):** **判断栈S是否为空**, 若是返回1(TRUE); 否则返回0(FALSE);
- ✶ **ElementType Pop( Stack S ):** **出栈操作**, 删除并返回栈顶元素。若栈为空, 返回栈为空信息; 否则将栈顶数据元素从栈中删除并返回。
- **6、 ElementType Top(Stack S ):** .....

## 【例】 ABCD四个字符入栈/出栈的过程。



**CreatStack( ); Push(S,A); Push(S,B); Push(S,C);**



**x=Pop(S); x=Pop(S); x=Pop(S); IsEmpty (S)**

## 【分析】

- A入栈时:  $f(1) = 1$  //即 A
- AB入栈时:  $f(2) = 2$  //即 AB、BA
- ABC入栈时:  $f(3) = 5$  //即 ABC、ACB、BAC、CBA、BCA、**CAB不行**
- ABCD四个元素入栈时:
  - 如果元素A在1号位置, 那么只可能A入栈后马上出栈, 剩余元素B、C、D等待操作, 就是子问题 $f(3)$ ;
  - 如果元素A在2号位置, 那么一定有一个元素比A先出栈, 即BA 为 $f(1)$ , 剩余元素C、D, 即 $f(2)$ , 根据乘法原理顺序个数为 $f(1) * f(2)$ ;
  - 如果元素A在3号位置, 那么一定有两个元素比1先出栈, 即有 $f(2)$ 种可能顺序(只能是B、C), 还剩D, 即 $f(1)$ , 根据乘法原理顺序个数为 $f(2) * f(1)$ ;
  - 如果元素A在4号位置, 那么一定是A先进栈, 最后出栈, 那么元素B、C、D的出栈顺序即是此小问题的解, 即 $f(3)$ ;

1. 输出能否产生CABD、BDAC序列?

**【结果】**  $f(4) = f(3) + f(2) * f(1) + f(1) * f(2) + f(3) = 14$

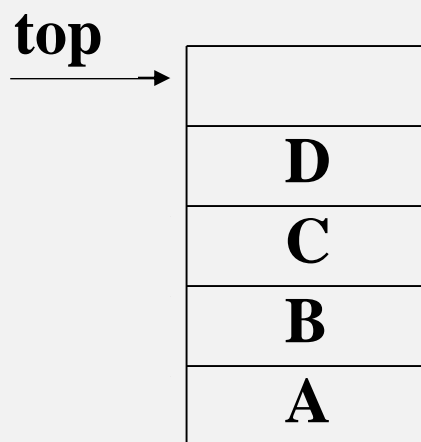


## 3.4.2 栈的实现

- 顺序栈 (Array-based Stack)
  - 由一个一维数组和一个记录栈顶元素位置的变量组成
- 链式栈 (Linked Stack)
  - 用单链表方式存储，其中指针的方向是从栈顶向下链接

## 1、栈的顺序存储实现

- 利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素，同时附设指针  $\text{top}$  指示栈顶元素在顺序栈中的位置。



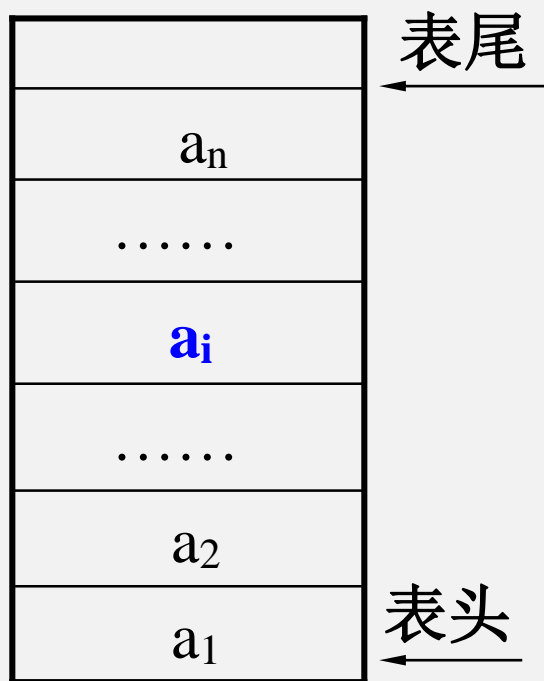
**栈满**

若再进行元素“出栈”操作，将产生“**下溢**”。

## 说明：表和栈的操作区别

对线性表  $s = (a_1, a_2, \dots, a_{n-1}, a_n)$

顺序表V[n]



写入:  $v[i] = a_i$   
读出:  $x = v[i]$

顺序栈S



压入:  $\text{PUSH}(a_{n+1})$   
弹出:  $\text{POP}(x)$

## 顺序栈类型Stack定义：

```
struct SNode{
    ElementType *Data; //存储元素的数组
    int Top;    //栈顶指针
    int MaxSize; //栈的最大容量
}
typedef struct SNode *PtrToSNode;
typedef PtrToSNode Stack;
```

### (1)顺序栈的创建过程

```
Stack CreateStack(int MaxSize)
{
    Stack S=(Stack)malloc(sizeof(Struct SNode));
    S->Data=(ElementType *)malloc(MaxSize*sizeof(Struct ElementType));
    S->Top=-1;
    S->MaxSize=MaxSize;
    return S;
}
```

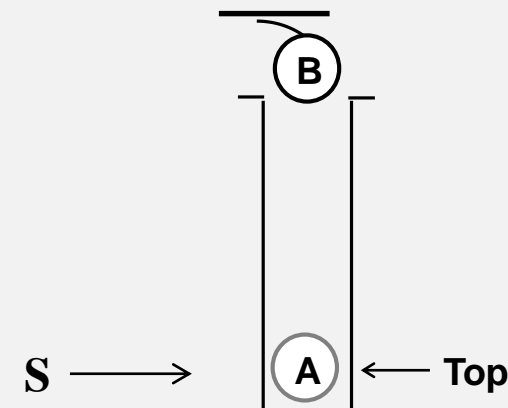


**(2)入栈：** 首先判断栈是否为满，若不满top加1，新元素存入data[top]

```
void Push( Stack S, ElementType X )
{
    if ( S->Top == MaxSize-1 )
    { printf("堆栈满");
      return false;
    }
    else {
        S->Data[++(S->Top)] = X;
        return true;
    }
}
```

**(3)判断栈是否为满**

```
bool IsFull( Stack S )
{
    return ( S->Top == MaxSize-1 )
}
```

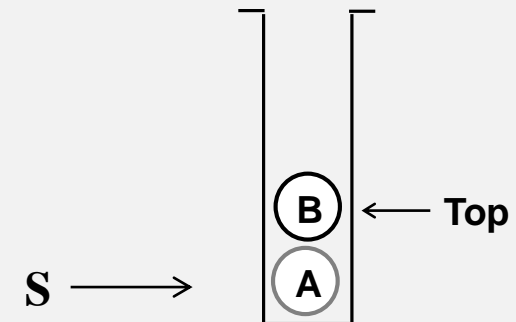


## (4)出栈:首先判断栈是否为空, 若不空返回data[top],top减1

```
ElementType Pop( Stack S )
{
    if ( S->Top == -1 ) {
        printf("堆栈空");
        return ERROR; /* ERROR是ElementType的错误标志 */
    } else
        return ( S->Data[ (S->Top) --] );
}
```

## (5)判断栈是否为空

```
bool IsEmpty( Stack S )
{
    return ( S->Top == -1 )
}
```



**【例】** 请用一个数组实现两个堆栈，要求最大地利用数组空间，使数组只要有空间入栈操作就可以成功。写出相应的入栈和出栈操作函数。

**【分析】** 定义一个数组，使两个栈分别从数组的**两头开始向中间生长**；当两个栈的**栈顶指针相遇**时，表示两个栈都已满。此时，最大化地利用了数组空间。

```
struct SNode {  
    ElementType *Data;  
    int Top1;    /* 堆栈 1 的栈顶指针 */  
    int Top2;    /* 堆栈 2 的栈顶指针 */  
    int MaxSize;  
};  
typedef struct *PtrToSNode;  
typedef PtrToSNode Stack;
```

初始情况:

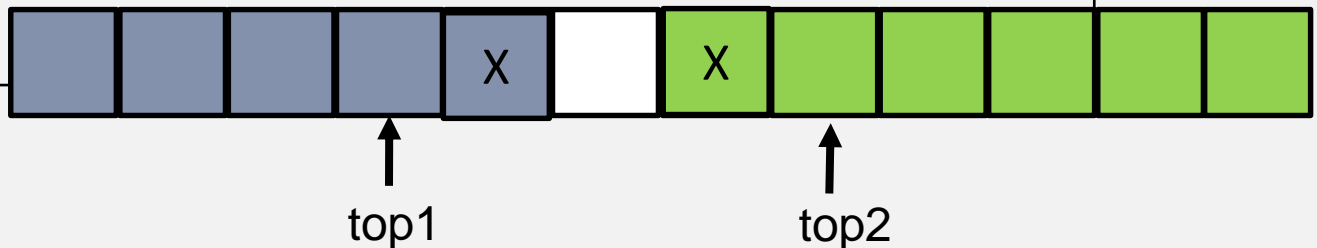
Top1=-1

Top2=MaxSize



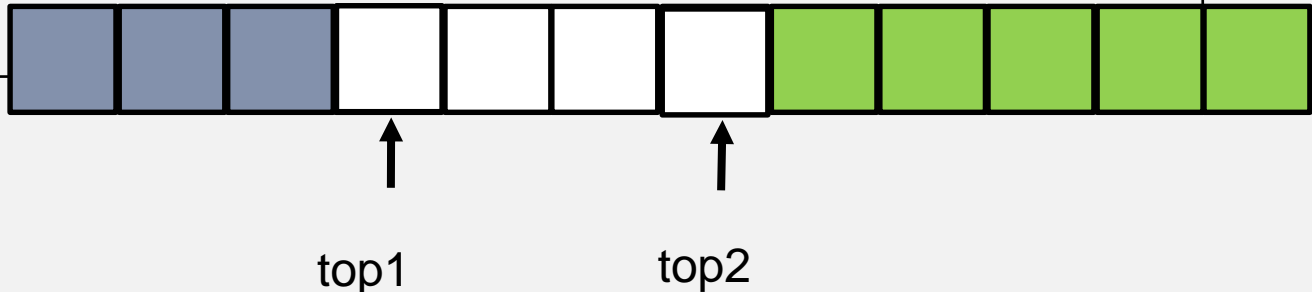
## (1)入栈

```
void Push(Stack S, ElementType X, int Tag )
{
    /* Tag作为区分两个堆栈的标志, 取值为1和2 */
    if ( S->Top2 - S->Top1 == 1) { /*堆栈满*/
        printf("堆栈满\n");
        return false;
    }
    else
    {
        if ( Tag == 1 )
            /* 对第一个堆栈操作 */
            S->Data[++(S->Top1)] = X;
        else
            /* 对第二个堆栈操作 */
            S->Data[--(S->Top2)] = X;
        return true;
    }
}
```

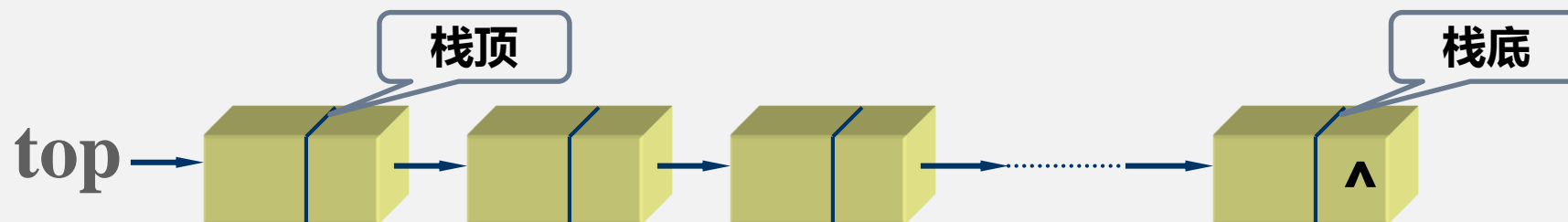


## (2)出栈

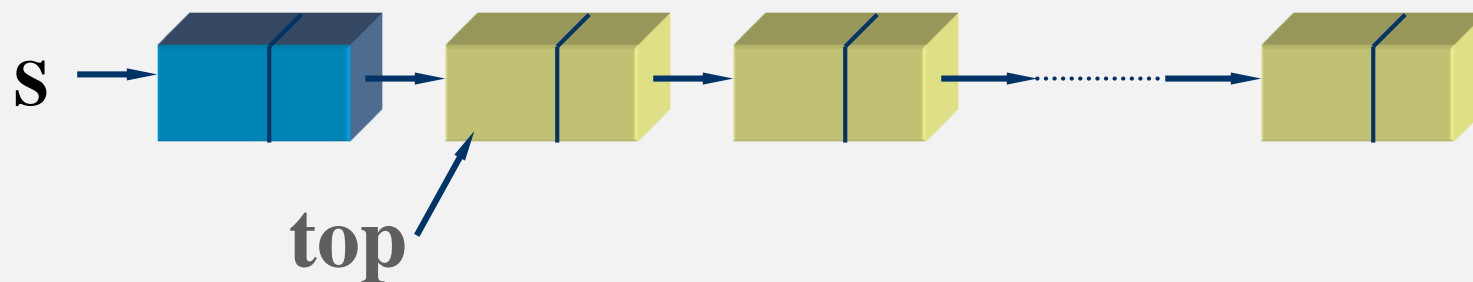
```
ElementType Pop( Stack *S, int Tag )
{
    /* Tag作为区分两个堆栈的标志, 取值为1和2 */
    if ( Tag == 1 ) {
        /* 对第一个堆栈操作 */
        if ( S->Top1 == -1 ) { /*堆栈1空 */
            printf("堆栈1空");
            return NULL;
        } else return S->Data[ (S->Top1) -- ] ;
    } else {
        /* 对第二个堆栈操作 */
        if ( S->Top2 == MaxSize ) { /*堆栈2空 */
            printf("堆栈2空");
            return NULL;
        } else return S->Data[ (S->Top2) ++ ] ;
    }
}
```



## 2、栈的链式存储实现



- 链式栈的栈顶在链头，插入与删除仅在**栈顶处**执行。
- 栈顶指针top就是链表的头指针，栈底结点的Next为NULL；
- 为了方便操作，链栈也可以带一个空的头结点，表头结点后面的第一结点就是链栈的栈顶结点。





## 链式栈的结构定义

```
typedef int ElementType; //每个栈元素的数据类型
struct SNode {           //栈元素结点定义
    ElementType data;      //结点数据
    struct SNode *next;    //结点链接指针
} *Stack; //链式栈
```

或者：

```
typedef struct SNode *PtrToSNode;
struct SNode {           //栈元素结点定义
    ElementType data;      //结点数据
    PtrToSNode next;       //结点链接指针
}
typedef PtrToSNode Stack;
```

## (1) 堆栈初始化 (建立空栈)

```
Stack CreateStack ()
{   /* 构建一个堆栈的头结点, 返回指针 */
    Stack S;
    S = malloc( sizeof(struct SNode ) );
    S->Next = NULL;
    return S;
}
```

## (2) 判断堆栈S是否为空

```
int IsEmpty( Stack S )
{   /*判断堆栈s是否为空, 若为空函数返回整数1, 否则返回0 */
    return ( S->Next == NULL );
}
```

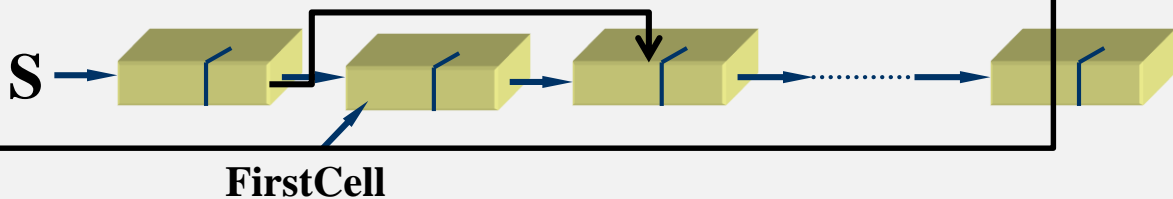


### (3) 入栈

```
void Push( ElementType X, Stack S )
{   /* 将元素x压入堆栈s */
    struct Node *TmpCell;
    TmpCell = malloc( sizeof( struct SNode ) );
    TmpCell->Data = X;
    TmpCell->Next = S->Next;    /* 新结点插入链表头部*/
    S->Next = TmpCell;
}
```

### (4) 出栈

```
ElementType Pop( LinkStack *S )
{   /* 删除并返回堆栈s的栈顶元素 */
    struct Node *FirstCell;
    ElementType TopElem;
    if( IsEmpty( S ) ) {
        printf("堆栈空"); return NULL; }
    else {
        FirstCell = S->Next;    /* 链表中删除首元结点*/
        S->Next = FirstCell->Next;
        TopElem = FirstCell->Data;
        free(FirstCell);
        return TopElem;
    }
}
```



### 3.4.3 栈的应用举例

#### 1. 数制转换

十进制数  $N$  和其他  $d$  进制数  $M$  的转换是计算机实现计算的基本问题，其解决方法很多，其中一个简单算法是逐次除以基数  $d$  取余法，它基于下列原理：

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

- 具体作法为：

- 用  $N$  除以  $d$ , 得到的余数是  $d$  进制数  $M$  的最低位  $M_0$ ;
- 以前一步得到的商作为被除数, 再除以  $d$ , 得到的余数是  $d$  进制数  $M$  的次最低位  $M_1$ ;
- 依次循环, 直到商为 0 时得到的余数是  $M$  的最高位  $M_s$  (假定  $M$  共有  $s + 1$  位) 。

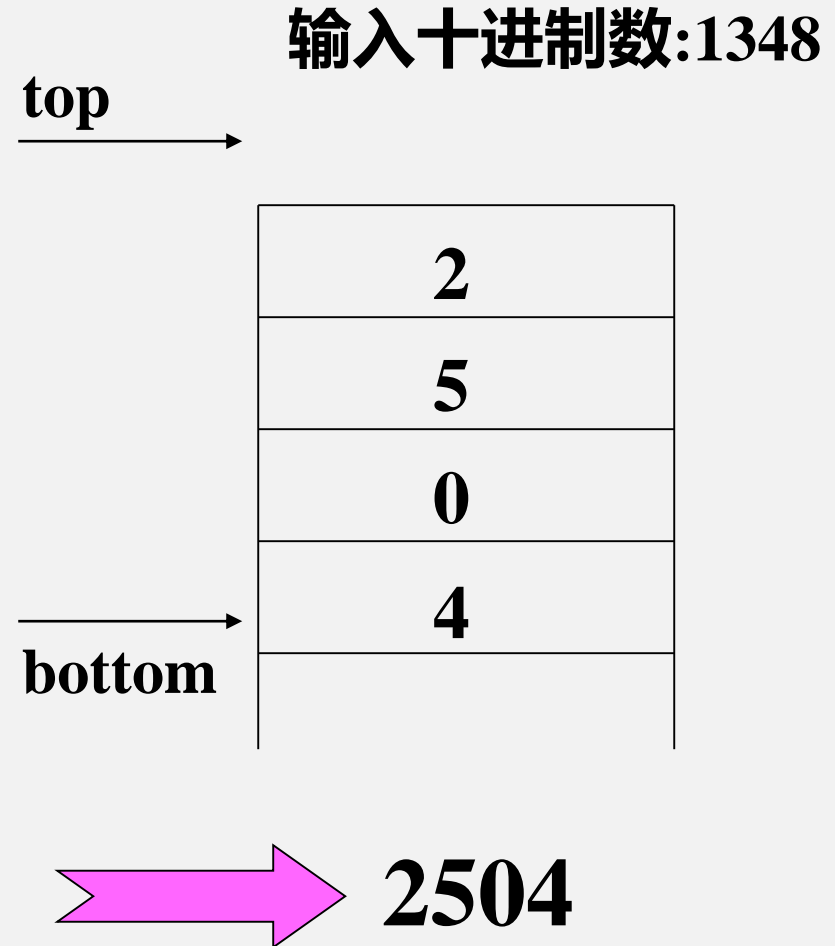
【例】  $(1348)_{10} = (2504)_8$ , 其运算过程如下:

$N$	$N \div 8$	$N \bmod 8$	$8 \overline{)1348}$	余数	对应的 8 进制数位
1348	168	4	$8 \overline{)168} \dots$	4	$M_0$
168	21	0	$8 \overline{)21} \dots$	0	$M_1$
21	2	5	$8 \overline{)2} \dots$	5	$M_2$
2	0	2	$8 \overline{)0} \dots$	2	$M_3$

计算顺序 ↓

输出顺序 ↑

```
void main()  
{  
    int stack[4];  
    int top=0, N;  
    scanf("%d", &N);  
    do  
    {  
        stack[top]=N%8;  
        N=N/8;  
        top++;  
    }while (N!=0);  
    for(top=top-1; top>=0; top--)  
        printf("%d",stack[top]);  
}
```



**通用方法：**输入一个非负十进制整数，输出等值的k进制数。

```
void main() {
    int N,k,d;
    cout<<"请输入一个非负十进制整数:";
    cin>>N;
    cout<<"要转换多少进制数?";
    cin>>k;
    LinkStack S;
    initStack(S);
    while(N>0){push(S,N%k);N=N/k;}
    while(!StackEmpty(S))
        {Pop(S,d);
         cout<<d;
        }
    cout<<endl;
}
```

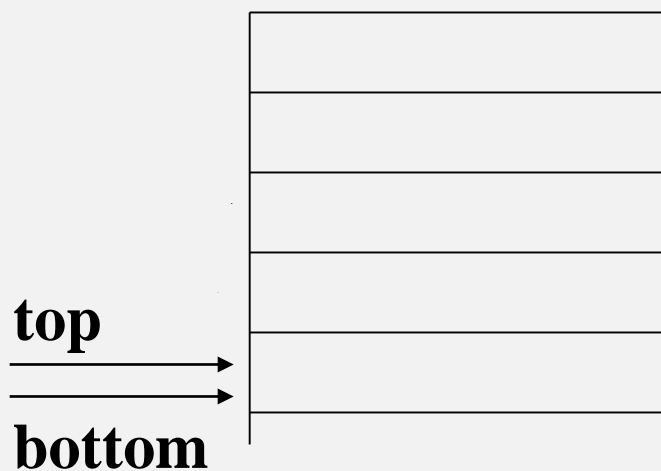
## 2. 括号匹配的检验

假设表达式中允许包含三种括号：圆括号、方括号和花括号，允许括号嵌套，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。

当计算机接收了一个左括号，它期待与其匹配的右括号，并且期待程度随着新的左括号的出现而降低

## 【例】

[ ( [ ] [ ] ) ]  
1 2 3 4 5 6 7 8



左括号入栈，右括号出栈

可能出现的不匹配的情况：

- 盼来的右括号不是所“期待”的；
- 盼来的是“不速之客”；
- 到结束也未盼来所“期待”的括号。

不正确的匹配：[ ( ] ) 或者 ( ( [ ] ) )  
或者 ( [ ( ) )



## 算法的设计思想:

- 1) 凡出现**左括号**，则**进栈**；
- 2) 凡出现**右括号**，首先检查栈是否空。
  - 若**栈空**，则表明该 **“右括号” 多余**；
  - 否则和栈顶元素比较，
    - 若相**匹配**，则 **“左括号出栈”**， 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 **“左括号” 有多余的**。

```

void matching()
{
    ElemType ch;
    int state=1;
    Stack *s;
    s=initstack();
    push (s, '#');
    printf("请输入括号,以Q结束: \n");
    do
    {
        switch(ch)
        {
            case '(': push (s,ch); break;
            case ')': if(s->length>1&&s->top->data=='(')
                      { pop(s); }
                      else
                      { state=0; }
                      break;
            case '[': push (s,ch); break;
            case ']': if(s->length>1&&s->top->data=='[')
                      { pop(s); }
                      else
                      { state=0; }
                      break;
        }
    }
}

```

```
        case '{': push(s, ch);  
                break;  
        case '}': if(s->length>1&& s->top->data=='{')  
                { pop(s); }  
                else  
                { state=0; }  
                break;  
    }  
    ch=getchar();  
}while(ch!='\n' && state);
```

```
if(s->top->data=='#' && state)  
    printf("括号匹配成功!\n");  
else  
    printf("括号匹配失败!\n");
```

```
}
```

### 3.应用后缀表示计算表达式的值

- 表达式的表示

- 表达式主要由两类对象构成的，即**运算数**（如2、3、4等）和**运算符**（如+、-、\*、/等）。

- 表达式的基本形式： $a_1 op_1 a_2 op_2 a_3$

- 中缀表达式：运算符位于两个运算数之间

- 后缀表达式：运算符位于两个运算数之后，也叫逆波兰表达式。

【例】      $a + b * (c - d) - e / f$               $a b c d - * + e f / -$

说明:

- 在后缀表达式的计算顺序中已**隐含了加括号的优先次序**,  
括号在后缀表达式中不出现。
- 例:  $a+b*(c-d)-e/f$

$$\begin{array}{c} a + b * (c - d) - e / f \\ \underbrace{\qquad\qquad\qquad}_{R_1} \\ \underbrace{\qquad\qquad\qquad}_{R_2} \\ \underbrace{\qquad\qquad\qquad}_{R_3} \\ \underbrace{\qquad\qquad\qquad}_{R_5} \end{array}$$

$$\begin{array}{c} a \ b \ c \ d \ - \ * \ + \ e \ f \ / \ - \\ \underbrace{\qquad\qquad\qquad}_{R_1} \\ \underbrace{\qquad\qquad\qquad}_{R_2} \\ \underbrace{\qquad\qquad\qquad}_{R_3} \\ \underbrace{\qquad\qquad\qquad}_{R_5} \end{array}$$

➤ 实现后缀表达式求值的基本过程：

- 从左到右读入后缀表达式的各项（运算符或运算数）；
- 根据读入的对象（运算符或运算数）判断执行操作；
- 操作分下列3种情况：
  - 1.当读入的是一个运算数时，把它被压入栈中；
  - 2.当读入的是一个运算符时，就从堆栈中弹出适当数量的运算数，对该运算进行计算，计算结果再压回到栈中；
  - 3.处理完整个后缀表达式之后，堆栈顶上的元素就是表达式的结果值。

• 举例  $a\ b\ c\ d\ -\ *\ +\ e\ f\ ^\wedge\ g\ /\ -$

步	输入	类 型	动 作	栈内容
1			置空栈	空
2	a	操作数	进栈	a
3	b	操作数	进栈	a b
4	c	操作数	进栈	a b c
5	d	操作数	进栈	a b c d
6	-	操作符	d、c 退栈, 计算 c-d, 结果 r1 进栈	a b r1
7	*	操作符	r1、b 退栈, 计算 b*r1, 结果 r2 进栈	a r2
8	+	操作符	r2、a 退栈, 计算 a+r2, 结果 r3 进栈	r3

步	输入	类 型	动 作	栈内容
9	e	操作数	进栈	r3 e
10	f	操作数	进栈	r3 e f
11	^	操作符	f、e 退栈, 计算 e^f, 结果 r4 进栈	r3 r4
12	g	操作数	进栈	r3 r4 g
13	/	操作符	g、r4 退栈, 计算 r4/g, 结果 r5 进栈	r3 r5
14	-	操作符	r5、r3 退栈, 计算 r3-r5, 结果 r6 进栈	r6



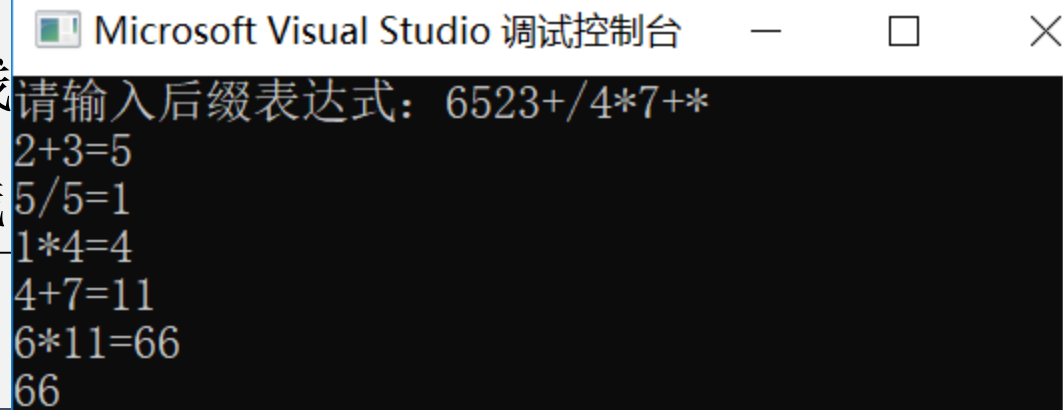
```

int calculate(string input)
{
    stack<int> num1; //操作数堆栈
    int temp;
    for (int i = 0; i < input.length(); i++) {
        if (isdigit(input[i])) //数字入栈
            { num1.push(input[i] - '0'); }
        else //遇到操作符就弹出两个操作数进行运算，结果入栈
            {
                int a = num1.top();
                num1.pop();
                int b = num1.top();
                num1.pop();
                switch (input[i])
                {
                    case '+': temp = b + a; break;
                    case '-': temp = b - a; break;
                    case '*': temp = b * a; break;
                    case '/': temp = b / a; break;
                    default: break;
                }
                cout << b << input[i] << a << "=" << temp << endl;
                num1.push(temp);
            }
    }
    return num1.top(); //最后的结果为栈顶元素
}

```

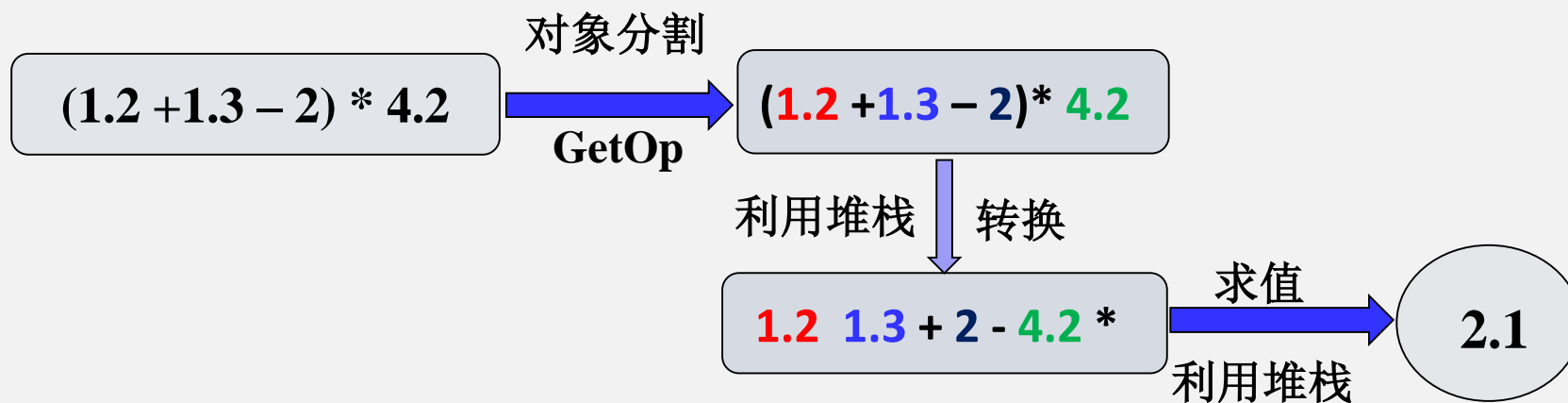
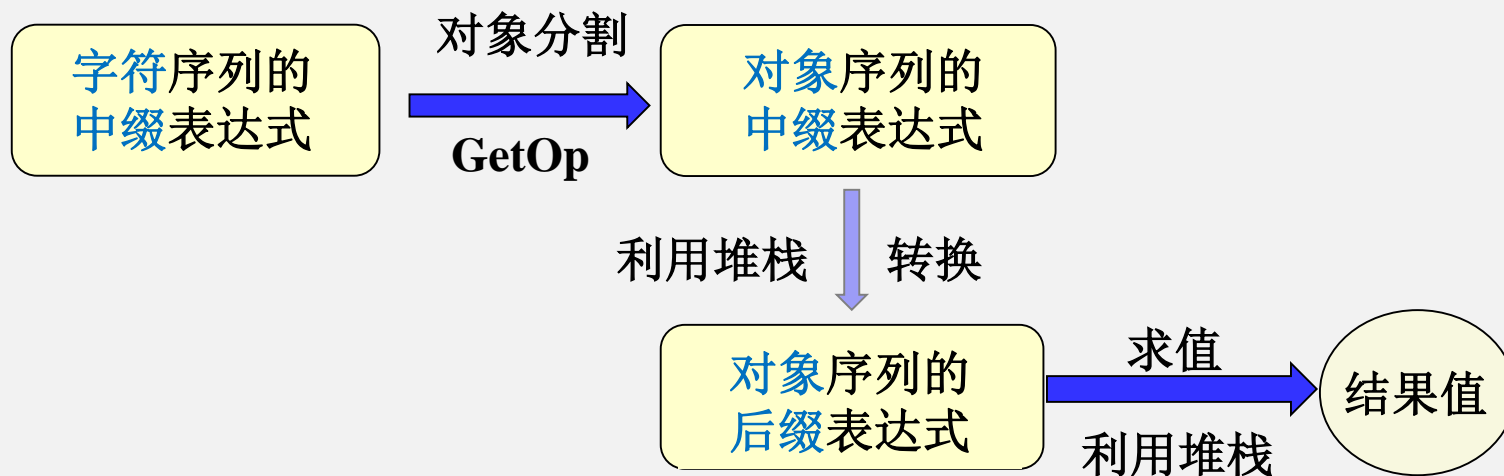
## 【例】 后缀表达式 $6\ 5\ 2\ 3\ +\ /\ 4\ *\ 7\ +\ *$

- 将前面的数字入栈， 栈为：6 5 2 3
- 遇到 ‘+’ 取栈顶元素  $2+3=5$ ，5入栈， 栈为：6 5 5
- 遇到 ‘/’ 取栈顶元素  $5/5=1$ ，1入栈， 栈为：6 1
- 遇到 ‘4’ 入栈， 栈为：6 1 4
- 遇到 ‘\*’ 取栈顶元素  $1*4=4$ ，入栈， 栈为：6 4
- 遇到 ‘7’ 入栈， 栈为：6 4 7
- 遇到 ‘+’ 取栈顶元素  $4+7=11$ 入栈， 栈为：6 11
- 遇到 ‘\*’ 取栈顶元素  $6*11=66$ 入栈， 栈为：66



```
Microsoft Visual Studio 调试控制台
请输入后缀表达式: 6523+/4*7+*
2+3=5
5/5=1
1*4=4
4+7=11
6*11=66
66
```

## 中缀表达式的计算?



## 4.利用栈将中缀表示转换为后缀表示

➤ 从头到尾读取**中缀表达式的每个对象**，对不同对象按不同的情况处理。对象分下列6种情况：

- ①如遇到**空格**则认为是分隔符，不需处理；
- ②若遇到**运算数**，则直接输出；
- ③若遇到**左括号**，则将其**压入堆栈**中；
- ④若遇到**右括号**，表明括号内的中缀表达式已经扫描完毕，将**栈顶的运算符弹出并输出**，直到遇到**左括号**（左括号也出栈，但不输出）；

⑤若遇到运算符，

- 若该运算符的优先级**大于**栈顶运算符的优先级时，则把它**压栈**；
- 若该运算符的优先级**小于等于**栈顶运算符时，将**栈顶运算符弹出并输出**，再比较新的栈顶运算符，按同样处理方法，直到该运算符**大于**栈顶运算符优先级为止，然后将该**运算符压栈**；

⑥若中缀表达式中的各对象**处理完毕**，则把堆栈中存留的**运算符一并输出**。

c <sub>1</sub> \ c <sub>2</sub>	+	-	*	/	(	)	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
*	>	>	>	>	<	<	<
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

例:将中缀表达式  $a + b * (c - d) - e / f \#$   
转换为后缀表达式  $a b c d - * + e f / -$

步	输入	栈内容	语义	输出	动作
1		#			栈初始化
2	a	#		a	操作数 a 输出, 读字符
3	+	#	+>#		操作符+进栈, 读字符
4	b	#+		b	操作数 b 输出, 读字符
5	*	#+	*>+		操作符*进栈, 读字符
6	(	#+*	(>*		操作符(进栈, 读字符
7	c	#+* (		c	操作数 c 输出, 读字符
8	-	#+* (	->(		操作符-进栈, 读字符
9	d	#+* (-		d	操作数 d 输出, 读字符
10	)	#+* (-	)<-	-	操作符-退栈输出
11		#+* (	)=(		(退栈, 消括号, 读字符

步	输入	栈内容	语义	输出	动作
12	-	#+*	-<*	*	操作符*退栈输出
13		#+	-<+	+	操作符+退栈输出
14		#	->#		操作符-栈, 读字符
15	e	#-		e	操作数 e 输出, 读字符
16	/	#-	/>-		操作符/进栈, 读字符
17	f	#-/		f	操作数 f 输出, 读字符
18	#	#-/	#</	/	操作符/退栈输出
19		#-	#<-	-	操作符-退栈输出
20		#	#=#		#配对, 转换结束

```

#include "pch.h"
#include <iostream>
#include <stack>
#include <map>
#include <string>
#include <cstdio>
#define MAX 100
using namespace std;
map<char, int> p;    //关联容器

```

```

struct Node{
double num; //操作数
    char op;    //操作符
    bool flag;
        //true操作数, false操作符
};

```

```

typedef struct Node expnode;
stack<expnode> s;    //操作符栈

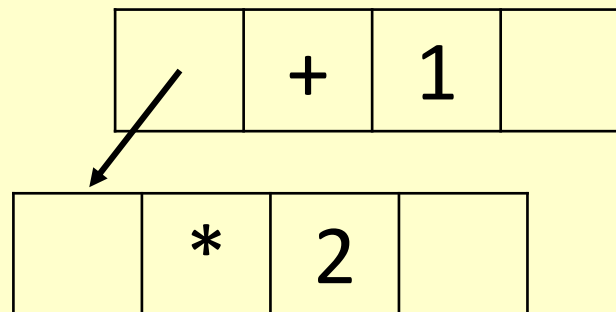
```

```

int main() {
expnode cur;
string str;
p['+'] = p['-'] = 1; //通过hashmap赋值
p['*'] = p['/'] = 2;
p['#'] = 0;
cout << "请输入中缀表达式: ";
cin >> str;
exchange(str);
}

```

➤ Map是一个容器，map以模板(泛型)方式实现，可以存储任意类型的数据，包括使用者自定义的数据类型。





```

void exchange(string str){
    expnode temp;
    temp.op = '#';
    temp.flag = false;
    s.push(temp);
    for (int i = 0; i < str.length(); )
    { if (str[i] >= '0' && str[i] <= '9') {
        //2.如果是数字
        temp.flag = true;
        temp.num = str[i] - '0';
        i++;
        while (i < str.length() && str[i] >=
'0' && str[i] <= '9')
        {
            temp.num = temp.num * 10 + (str[i] - '0');
            i++;
        }
        cout << temp.num;
    }
    else if (str[i] == '(') { //3.左括号入栈
        temp.flag = false;
        temp.op = str[i];
        s.push(temp);
        i++;
    }
}

```

```

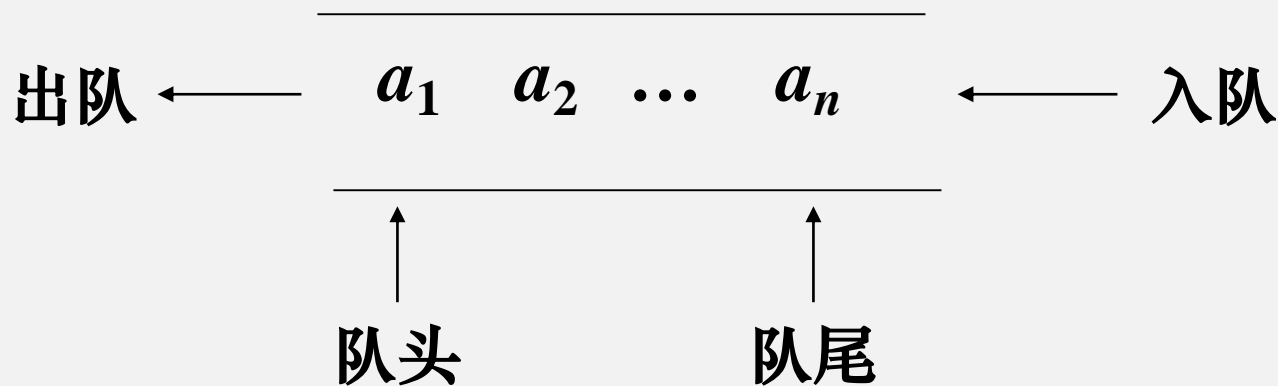
    else if (str[i] == ')') {
        //4.右括号出栈操作
        while (!s.empty() && s.top().op != '(')
        { cout << s.top().op;
          s.pop();
        }
        s.pop(); //弹出左括号
        i++;
    }
    else { //5.如果是操作符
        temp.flag = false;
        while (!s.empty() &&
            p[s.top().op] >= p[str[i]]) {
            if (str[i] != '#') cout << s.top().op;
            s.pop();
        }
        temp.op = str[i];
        s.push(temp);
        i++;
    }
    while (!s.empty()) { //6.将栈中剩余内容依次输出
        if (s.top().op != '#')
        { cout << s.top().op;
          s.pop();
        }
    }
}

```

## 3.5.1 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)。

队列的示意图：



新来者排在队伍的末尾，排在队伍前端的人先得到服务，不允许插队

- 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。
- 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。

## ❖ 队列的基本操作

类型名称：队列(Queue)

数据对象集：一个有**0个或多个元素**的**有穷线性表**。

操作集：对于一个长度为正整数MaxSize的队列 $Q \in \text{Queue}$ ，记队列中的任一元素 $\text{item} \in \text{ElementType}$ 。

- 1、**Queue CreatQueue( int MaxSize )**：生成长度为MaxSize的空队列。
- 2、**int IsFullQ( Queue Q, int MaxSize )**：判断队列Q是否已满，若是返回1(TRUE)；否则返回0(FALSE)。
- ❖ **void AddQ( Queue Q, ElementType item )**：若队列已满，返回已满信息；否则将数据元素item插入到队列Q中去。
- ❖ **int IsEmptyQ( Queue Q )**：判断队列Q是否为空，若是返回1(TRUE)；否则返回0(FALSE)。
- 5、**ElementType DeleteQ( Queue Q )**：若队列为空信息，返回队列空信息；否则将队头数据元素从队列中删除并返回。



## 3.5.2 队列的实现

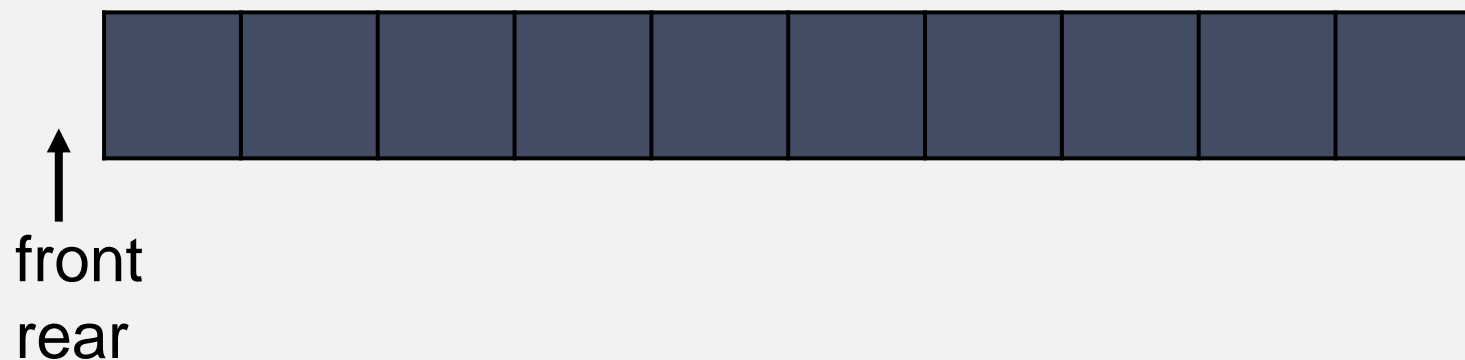
队列类型的实现：

- ◆ 顺序存储——循环队列
- ◆ 链式存储——链队列

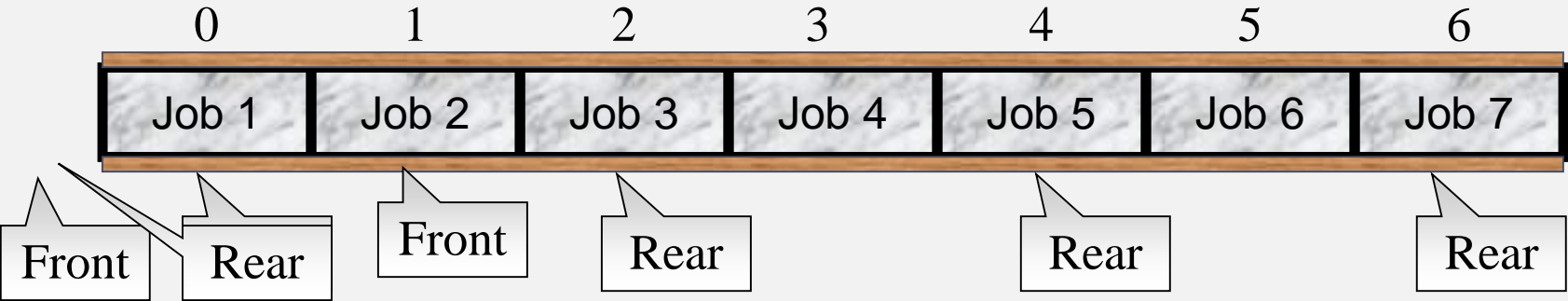
# 1. 队列的顺序存储实现

队列的顺序存储结构通常由一个**一维数组**和一个记录队列头元素位置的变量**front**以及一个记录队列尾元素位置的变量**rear**组成。

```
#define MaxSize <储存数据元素的最大个数>
typedef struct {
    ElementType Data[ MaxSize ];
    int rear;
    int front;
} Queue;
```



【例】 一个工作队列的入队和出队操作。



AddQ Job 1	AddQ Job 2	AddQ Job 3	DeleteQ Job 1
AddQ Job 4	AddQ Job 5	AddQ Job 6	DeleteQ Job 2
AddQ Job 7	AddQ Job 8		

- **假溢出**

- 当  $\text{rear} = \text{MAXSize} - 1$  时，再作插入运算就会产生溢出，如果这时队列的前端还有许多空的（可用的）位置，这种现象称为假溢出

- **上溢（满员）**

- 当队列满时，再做进队操作，所出现的现象

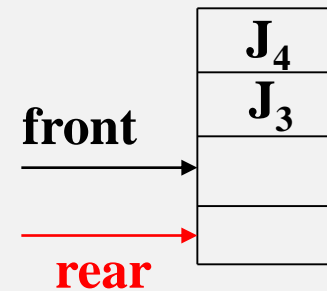
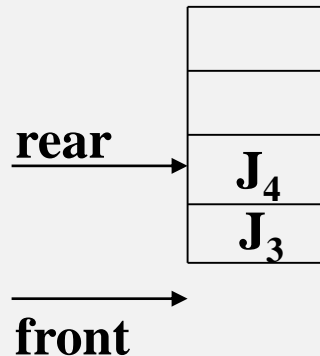
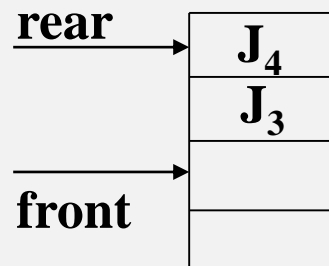
- **下溢**

- 当队列空时，再做删除操作，所出现的现象

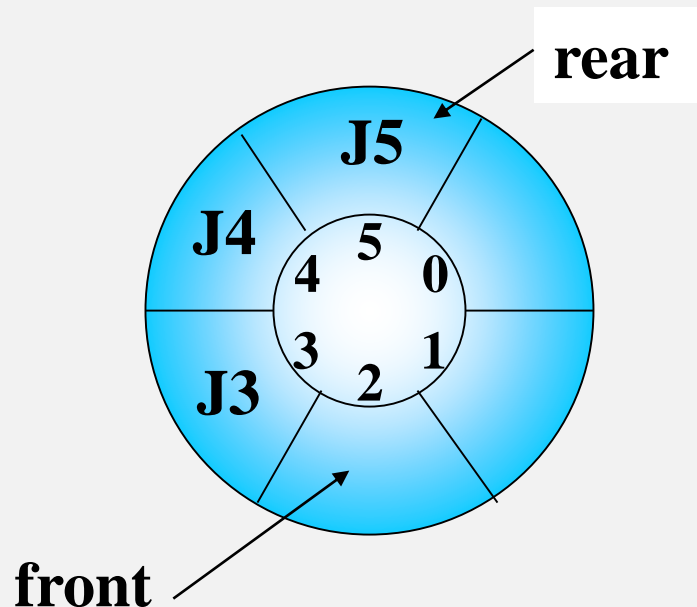


## 解决“假溢出”的问题有两种可行的方法：

- (1) 平移元素：把所有元素平移到队列的首部。**效率低。**
- (2) 将顺序队列的存储区假想为一个环状的空间，当发生假溢出时，将新元素插入到第一个位置上，这样做，虽然物理上队尾在队首之前，但逻辑上队首仍然在前。入列和出列仍按“先进先出”的原则进行，这就是**循环队列**。**操作效率、空间利用率高。**



## 2.循环队列



循环意义下的加 1 操作可以描述为：

```
if (rear + 1 >= maxSize)
```

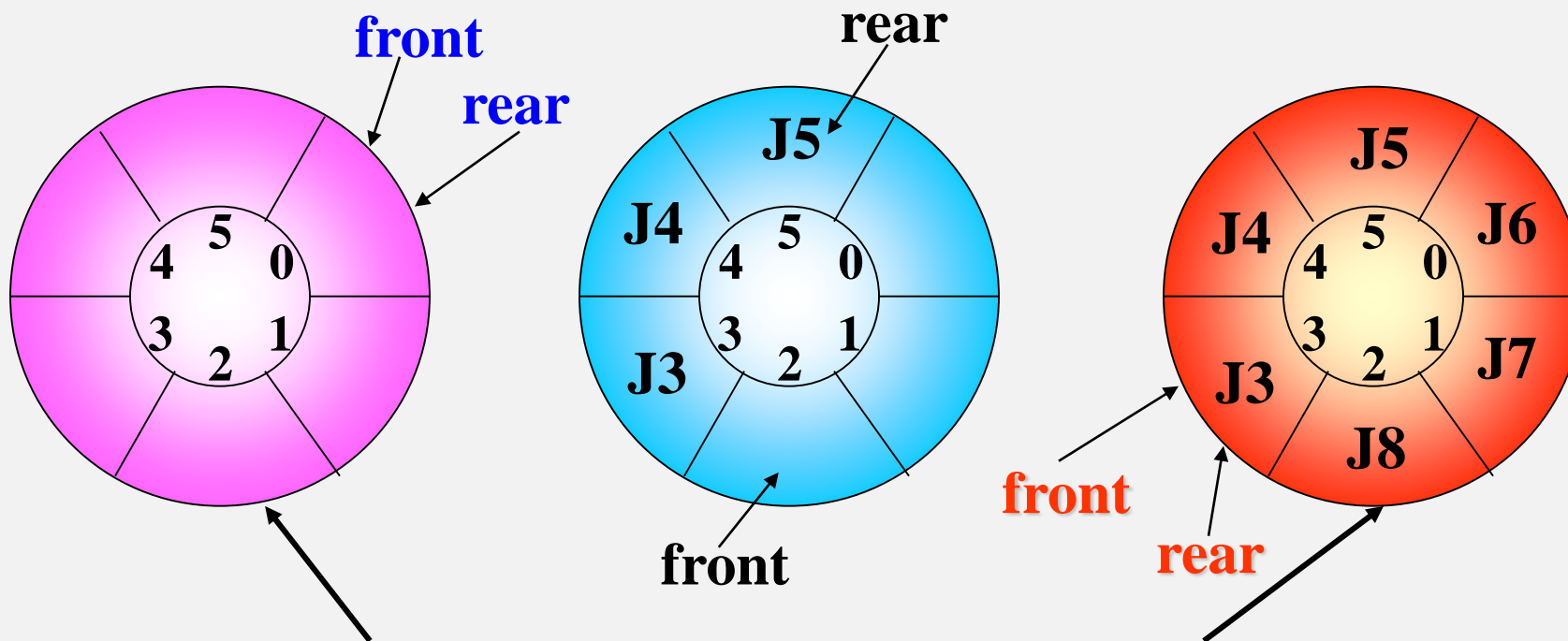
```
    rear = 0;
```

```
else
```

```
    rear ++;
```

利用模运算可简化为：  $\text{rear} = (\text{rear} + 1) \% \text{maxSize}$

# 循环队列的三种状态



**注：** 仅凭  $\text{front} = \text{rear}$  不能判定队列是空还是满。

如何根据Front和Rear值判断队列是否为空或满？

- **解决办法:**

- (1)另设一个变量，例如记录当前队列元素个数的变量Size或者一个Flag变量记录最后一次操作是入队还是出队；
- (2)少用一个元素的空间，约定入队前，测试尾指针在循环意义下加 1 后是否等于头指针，若相等则认为队满。

队列为空:  $Q \rightarrow rear == Q \rightarrow front$

队列为满:  $(Q \rightarrow rear + 1) \% MAXSIZE == Q \rightarrow front$

## 队列的顺序存储结构:

```
typedef struct QNode{  
    ElemType *data;  
    int front;        // 头指针, 指向队列头元素的前一个位置  
    int rear;         // 尾指针, 指向队列尾元素  
    int MaxSize;  
}  
typedef struct QNode *PtrToQNode;  
typedef PtrToQNode Queue;
```

# 队列的基本操作在循环队列中的实现：

## (1) 队列的初始化：

```
Queue CreateQueue (int MaxSize) {  
    // 构造一个空队列Q  
    Queue Q =(Queue) malloc(sizeof(struct QNode));  
    Q->Data=(ElementType *)malloc(MaxSize*sizeof(ElementType));  
    Q->front =0;  
    Q->rear = 0;  
    Q->MaxSize=MaxSize;  
    return Q;  
}
```

## (2) 插入操作在循环队列中的实现

```
bool AddQueue (Queue Q, ElementType X) {
```

```
    // 插入元素 x 为 Q 的新的队尾元素
```

```
    if ((Q->rear + 1) % MAXSIZE == Q->front) {
```

```
        printf("队列满\n");
```

```
        return false; } // 队列满
```

```
    else{
```

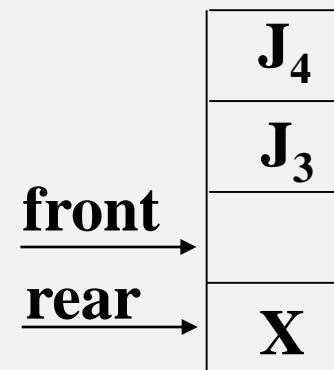
```
        Q->rear = (Q->rear + 1) % MAXSIZE;
```

```
        Q->data[Q->rear] = x;
```

```
        return true; }
```

```
}
```

Bool IsFull (Queue Q)



### (3) 删除操作在循环队列中的实现

```
ElementType DeleteQu (Queue Q) {
```

```
    // 若队列不空, 则删除 Q 的队头元素,
```

```
    if (Q->front == Q->rear)
    {
        printf("队列空");
        return Error;
    }
```

```
    else {
```

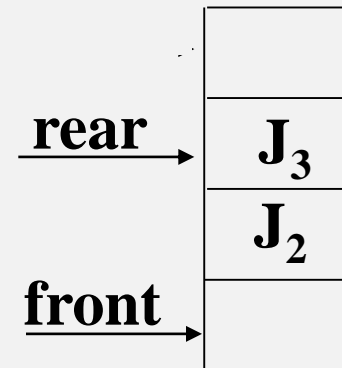
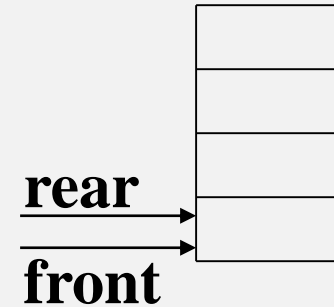
```
        Q->front = (Q->front + 1) % MAXSIZE;
```

```
        return Q->Data[Q->Front];
```

```
    }
```

```
}
```

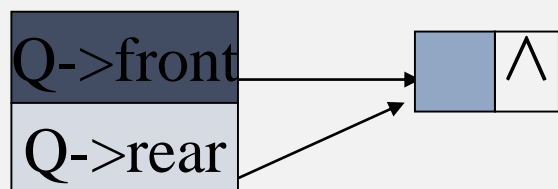
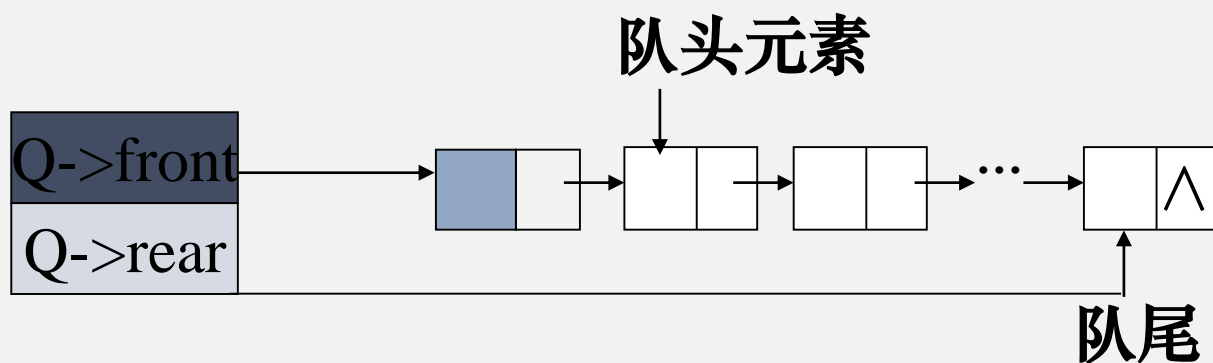
Bool IsEmpty (Queue Q)





### 3 链队列——队列的链式表示和实现

**链队列：**用链表表示的队列（队列的链式存储结构）。是限制仅在表头删除和表尾插入的单链表。



## 定义链队列结构如下：

```
typedef struct Node{
    ElementType  data;
    struct Node  *next;
}QNode;
typedef struct Node *QueuePtr;
```

```
typedef struct {           /* 链队列结构  */
    QueuePtr rear;         /* 指向队尾结点 */
    QueuePtr front;        /* 指向队头结点 */
}LinkQueue;
```

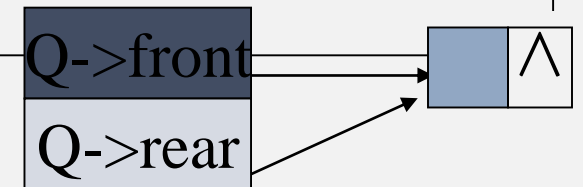
# 队列的基本操作在链队列中的实现:

## (1) 队列的初始化(创建空队列):

```
CreateQueue (LinkQueue *Q)
{
    // 构造一个空队列 Q

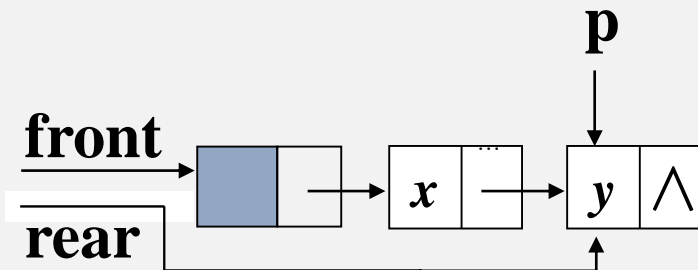
    Q->front=Q->rear=(QueuePtr)malloc(sizeof(Node));
    if(!Q.front){printf("存储分配失败! ");exit(1);}

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



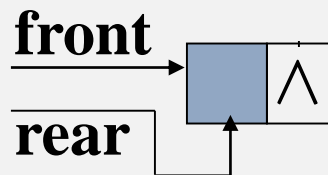
## (2) 插入操作在链队列中的实现

```
bool AddQueue (LinkQueue *Q, ElementType x)
{    // 插入元素 x 为 Q 的新的队尾元素
    QueuePtr p;
    p =(QueuePtr)malloc(sizeof(Node));
    if (p==NULL) {
        printf("存储分配失败! "); exit (1);
    }
    p -> data = x;
    p ->next = NULL;
    Q->rear -> next = p;
    Q->rear = p;
    return true;
}
```



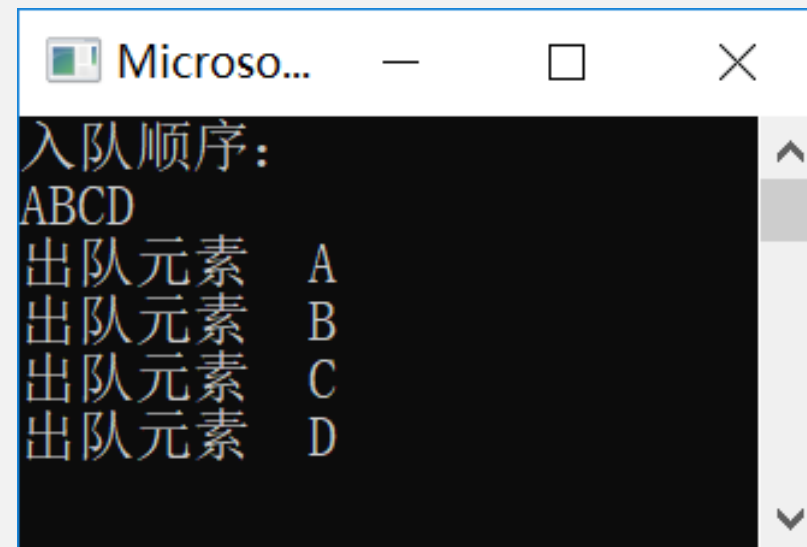
### (3) 删除操作在链队列中的实现

```
ElementType DeleteQ (LinkQueue *Q )
{
    QNode *p;
    ElementType FrontData;
    if ( Q->front->next == NULL) {
        printf("队列空");      return ERROR;
    }
    p = Q->front->next;
    FrontData = p->Data;
    Q->front -> next = p -> next;
    if (Q->rear == p) Q->rear = Q->front;
    free( p );                /* 释放被删除结点空间 */
    return FrontData;
}
```



## 【例】ABCD四个元素的入队和出队操作。

```
int main() {
    LinkQueue testQ;
    char x,y;
    CreateQueue(&testQ);
    printf("入队顺序: \n");
    for (int i = 0; i <= 3; i++) {
        scanf_s("%c", &x);
        AddQueue(&testQ, x);
    }
    printf("出队元素   %c \n", DeleteQ(&testQ));
    printf("出队元素   %c \n", DeleteQ(&testQ));
    printf("出队元素   %c \n", DeleteQ(&testQ));
    printf("出队元素   %c \n", DeleteQ(&testQ));
}
```



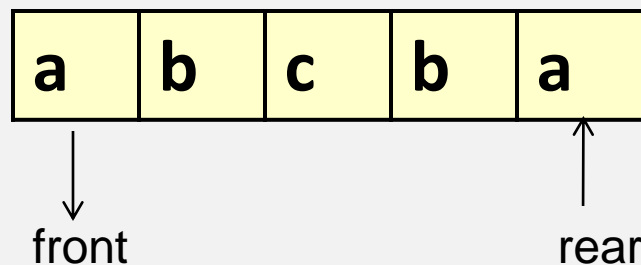
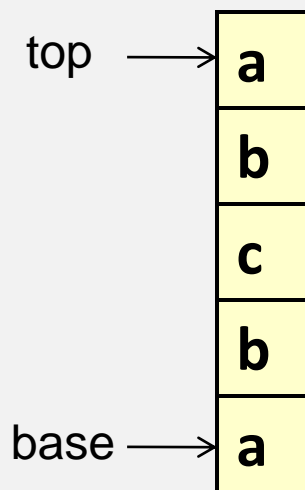
```
Microso...
入队顺序:
ABCD
出队元素   A
出队元素   B
出队元素   C
出队元素   D
```



## 4 队列的应用

【例】判断一个字符序列是否是回文.

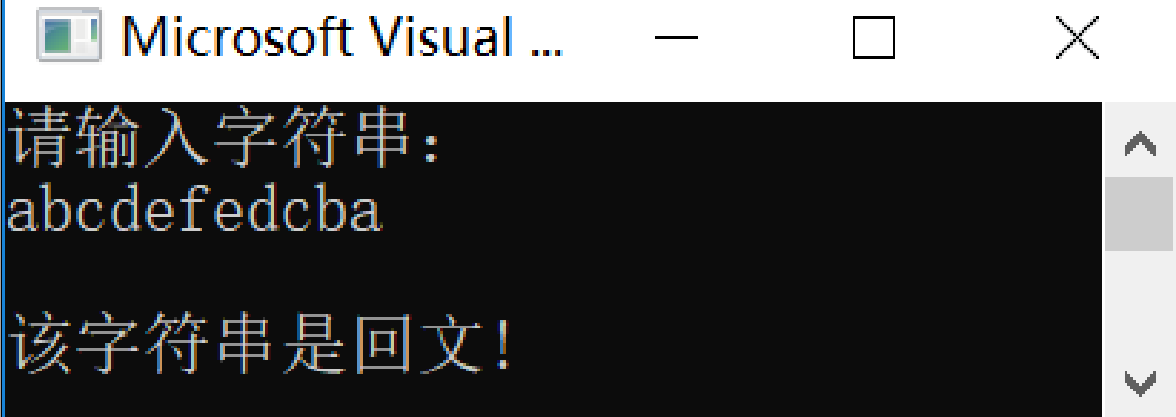
字符序列以中间字符  
为基准两侧字符完全  
对称相等



从队列和栈中退出一个字符,比较退出队列的字符和退栈的字符是否相等. 若不相等,则不是回文.

```
#include<stack>
#include<queue>
bool HuiWen(char *str) {
    stack<char> s;
    queue<char> q;
    char ch1, ch2;
    for (int i = 0; i < strlen(str); i++)
    {    q.push(str[i]);
        s.push(str[i]);}
    while (!s.empty() && !q.empty()) {
        ch1 = s.top();s.pop();
        ch2=q.front();q.pop();
        if (ch1 != ch2)    return false;
    }
    return true;
}
```

```
int main() {
    char str[100];
    cout << "请输入字符串: \n" << endl;
    cin >> str;
    if (HuiWen(str))
        {cout<<"该字符串是回文!\n";}
    else
        {cout<<"该字符串不是回文!\n";}
}
```





## • 打印杨辉三角形和逐行处理

二项式  $(a+b)^n$  展开式的二项式系数，当  $n$  依次取  $1, 2, 3, \dots$  时，列出的一张表，叫做二项式系数表，因它形如三角形，南宋的杨辉对其有过深入研究，所以我们又称它为杨辉三角。（表 1）

								1									
								1		1							
							1		2		1						
					1		3		3		1						
			1		4		6		4		1						
		1		5		10		10		5		1					
	1		6		15		20		15		6		1				
1		7		21		35		35		21		7		1			

```

void YANGVI(int n) {
    queue<int> q;
    int i, j, l;
    int s = 0;
    int t;
    cout << "          1";
    q.push(1);
    q.push(1);
    for (i = 1; i <= n; i++)
    {
        cout << endl;
        q.push(0);
        for (l = 0; l < n - i; l++) cout << " ";
        for (j = 1; j <= i + 2; j++) {
            t=q.front();
            q.pop();
            q.push(s + t);
            s = t;
            if (j != i + 2) cout << s<<" ";
        }
    }
}

int main()
{
    YANGVI(10);
}

```

1	1								初始状态
				1	3	3	1		
					1	4	6	4	1

Microsoft Visual Studio 调试控制台

```

          1
        1 1
      1 2 1
    1 3 3 1
  1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
1 10 45 120 210 252 210 120 45 10 1

```