

第三章 栈和队列

内容提要：

栈和队列是两种特殊的线性表，即它们都是线性数据结构，其特殊性是表现在操作定义上：插入和删除操作只能在两端进行（不允许在线性表的中间插入和删除）。

■ 栈

- 数据结构——线性关系
- 操作——一端固定，只允许一端插入删除
- 特性——“先进的后出，后进的先出”

■ 队列

- 数据结构——线性关系
- 操作——一端只允许插入，一端只允许删除
- 特性——“先进的先出，后进的后出”

栈:

■ 栈ADT的定义（特性）

- 数据结构
- 操作定义

■ 栈ADT的实现

- 基于顺序存储的实现
- 基于链式存储的实现

■ 栈ADT的应用——重点

队列:

■ 队列ADT的定义（特性）

- 数据结构
- 操作定义

■ 队列ADT的实现

- 基于顺序存储的实现
- 基于链式存储的实现

■ 队列ADT的应用——重点

3.1 栈

3.1.1 栈ADT的定义

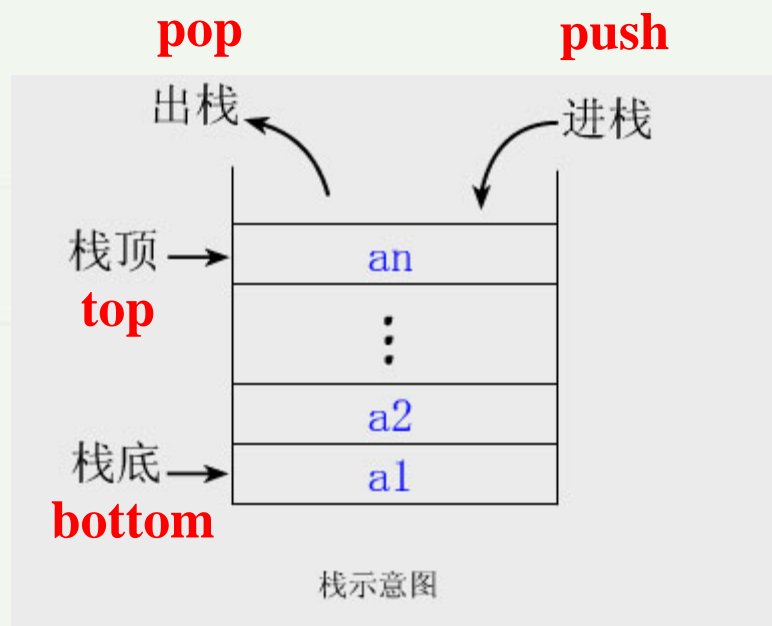
[栈 Stack] 一端固定，只允许在另一端插入和删除的线性表。

线性表的固定端——**栈底**

线性表的插入删除端——**栈顶**

线性表的插入操作——**入栈**
压入

线性表的删除操作——**出栈**
弹出



特性：线性表的元素符合“先进后出(FILO:First In Last Out)”或“后进先出(LIFO-Last In First Out)”

3.1 栈

3.1.1 栈ADT的定义

栈ADT定义:

ADT Stack

data structure:

$D = \{a_i \mid a_i \in D_0 \ i=1,2,\dots, n \geq 0\}$

$R = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D_0 \ i=2,3,4,\dots \}$

D_0 是某个数据对象

operations:

Initiate(S);

Push(S,x);

Pop(S);

GetTop(S,x);

IsEmpty(S);

IsFull(S)

Getsize(S);

END

3.1 栈

3.1.1 栈ADT的定义

栈ADT的
抽象类定义：

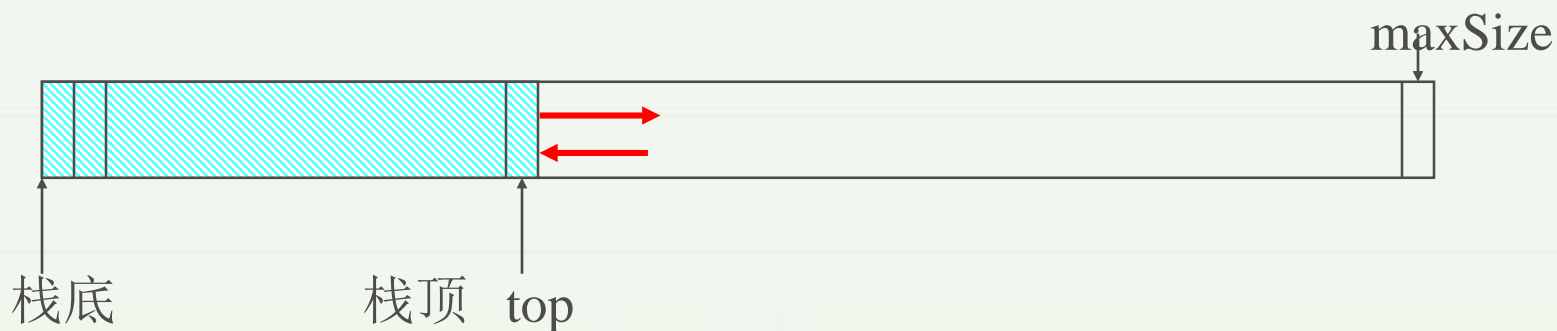
```
typedef 栈元素类型 ElemType;
class Stack
{ //栈的类定义
public:
    Stack(){ };                //构造函数
    ~Stack(){ };
    virtual void Push(ElemType x);    //进栈
    virtual bool Pop(ElemType & x);    //出栈
    virtual bool GetTop(ElemType & x);    //取栈顶
    virtual bool IsEmpty();            //判栈空
    virtual bool IsFull();            //判栈满
    virtual int GetSize( ) /栈大小
};
```

3.1 栈

3.1.2 栈ADT的实现

3.1.2.1 栈ADT的基于顺序存储的实现——顺序栈

1. 存储结构：同一般线性表的顺序存储结构基本一样



3.1 栈

3.1.2 栈ADT的实现

3.1.2.1 栈ADT的基于顺序存储的实现——顺序栈

1. 存储结构

特点：简单、方便，但易产生溢出。

■ 上溢(Overflow): 栈已经满，又要压入元素；

■ 下溢(Underflow): 栈已经空，还要弹出元素；

3.1 栈

3.1.2 栈ADT的实现

3.1.2.1 栈ADT的基于顺序存储的实现——顺序栈

2. C++类实现

顺序栈类的定义:

```
typedef 数据类型 ElemType;
class SeqStack
{ //顺序栈类定义
    private:
        ElemType *elements;           //数组存放栈元素
        int top;                       //栈顶指示器
        int maxSize;                   //栈最大容量
        void overflowProcess();         //栈的溢出处理
}
```


3.1 栈

3.1.2 栈ADT的实现

3.1.2.1 栈ADT的基于顺序存储的实现——顺序栈

public:

SeqStack(int sz); //构造函数

~SeqStack() { delete []elements; } //析构函数

void **Push**(ElemType &x); //进栈

bool **Pop**(ElemType& x); //出栈

bool GetTop(DataType& x); //取栈顶内容

bool IsEmpty() const { return top == -1; }

bool IsFull() const { return top == maxSize-1; }

int GetSize() const {return top+1;}

void MakeEmpty(){top=-1;}

friend ostream&operator<<(ostream&os,SeqStack &S)

};

3.1 栈

3.1.2 栈ADT的实现

3.1.2.1 栈ADT的基于顺序存储的实现——顺序栈

成员函数的实现:

(1) 构造函数

```
SeqStack::SeqStack (int sz)
{
    elements=new ElemType[sz]; //申请连续空间
    assert(elements!=NULL);
    top=-1;    //栈顶指示器指向栈底
    maxSize=sz; //栈的最大空间
}
```

时间复杂性: $O(1)$

3.1 栈

3.1.2 栈ADT的实现

3.1.2.1 栈ADT的基于顺序存储的实现——顺序栈

(2) 栈溢出处理

```
void SeqStack::overflowProcess()  
{//私有函数：当栈满则执行扩充栈存储空间处理  
    ElemType *newArray = new ElemType [2*maxSize];  
        //创建更大的存储数组  
    for (int i = 0; i <= top; i++) //复制元素  
        newArray[i] = elements[i];  
    maxSize += maxSize; //栈空间扩大  
    delete [ ]elements; //释放原来的连续空间  
    elements = newArray; //改变elements指针  
};
```

时间复杂性： $O(n)$

3.1 栈

3.1.2 栈ADT的实现

3.1.2.1 栈ADT的基于顺序存储的实现——顺序栈

(3) 入栈

```
void SeqStack::Push(ElemType x)
{ //若栈满,则溢出处理, 将元素x插入该栈栈顶
  if (IsFull() == true) overflowProcess( );    //栈满
  elements[++top] = x;    //栈顶指针先加1, 再元素进栈
};
```

时间复杂性: $O(1)$

3.1 栈

3.1.2 栈ADT的实现

3.1.2.1 栈ADT的基于顺序存储的实现——顺序栈

(4) 出栈

```
bool SeqStack::Pop(ElemType & x)
{ //若栈不空，函数退出栈顶元素并将栈顶元素的值赋给x，
  //返回true，否则返回false
    if (IsEmpty() == true) return false;
    x = elements[top--];      //先取元素，栈顶指针退1
    return true;              //退栈成功
};
```

时间复杂性: $O(1)$

其他顺序栈的操作请同学们自己写出!

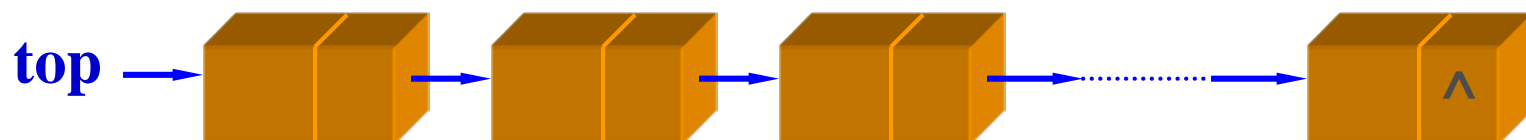
3.1 栈

3.1.2 栈ADT的实现

3.1.2.2 栈ADT的基于链式存储的实现——单链栈

1. 存储结构：同一般线性表的单链式存储基本一样

那么链表的哪端应该作为栈顶？



3.1 栈

3.1.2 栈ADT的实现

3.1.2.2 栈ADT的基于链式存储的实现——单链栈

1. 存储结构:

特点: 栈空间易扩充, 减少了溢出的可能。只有系统没有空间了, 才会溢出;

存储结构的高级语言实现:

请同学们自己写出!

3.1 栈

3.1.2 栈ADT的实现

3.1.2.2 栈ADT的基于链式存储的实现——单链栈

2. C++类实现

链式栈类的定义:

```
typedef 数据元素类型 ElemType;
struct StackNode      //栈结点类定义
{ public:
    ElemType data;      //栈结点数据
    StackNode *link;    //结点链指针
    StackNode(ElemType d = 0, StackNode *next =
NULL) : data(d), link(next) { }
    ~StackNode();
};
```


3.1 栈

3.1.2 栈ADT的实现

3.1.2.2 栈ADT的基于链式存储的实现——单链栈

```
class LinkedStack    //链式栈类定义
{   private:
    StackNode *top;    //栈顶指针
    public:
        LinkedStack() : top(NULL) {} //构造函数
        ~LinkedStack() { makeEmpty(); } //析构函数
        void Push(DataType &x);    //进栈
        bool Pop(DataType & x);    //出栈
        bool getTop(DataType & x) const;    //取栈顶 元素
        bool IsEmpty() const { return (top == NULL)? true:false; }
        int getSize()const;
        void makeEmpty(); //清空栈的内容
        friend ostream& operator << (ostream& os, LinkedStack& s) ;
            //输出栈元素的重载操作 <<
};
```

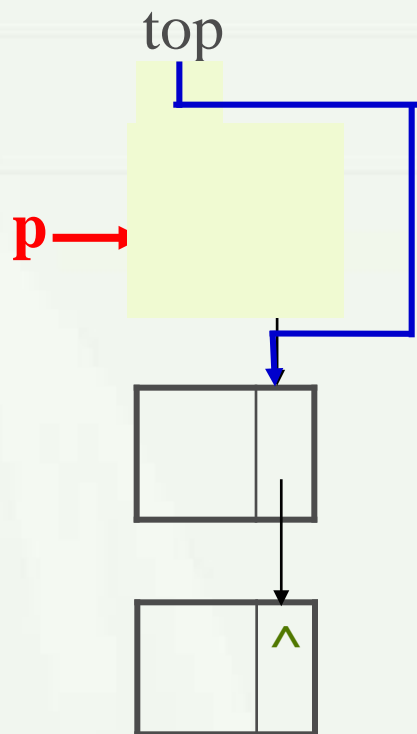
3.1 栈

3.1.2 栈ADT的实现

3.1.2.2 栈ADT的基于链式存储的实现——单链栈

成员函数的实现：

(1) 栈置空：同单链表置空



```
void LinkedStack::makeEmpty()
{ //逐次删去链式栈中的元素直至栈顶指针为空。
  StackNode *p;
  while (top != NULL)
  { //逐个结点释放
    p = top;
    top = top->link;
    delete p;  }
};
```

时间复杂性： $O(n)$

3.1 栈

3.1.2 栈ADT的实现

3.1.2.2 栈ADT的基于链式存储的实现——单链栈

(2) 入栈

```
void LinkedStack::Push(ElemType &x)
{ //将元素值x插入到链式栈的栈顶,即链头。
  top = new StackNode(x, top);    //创建新结点
  assert (top != NULL);    //创建失败退出
};
```

时间复杂性: **O(1)**

3.1 栈

3.1.2 栈ADT的实现

3.1.2.2 栈ADT的基于链式存储的实现——单链栈

(3) 出栈

```
bool LinkedStack::Pop(ElemType & x)
{ //删除栈顶结点, 返回被删栈顶元素的值。
    if (IsEmpty() == true) return false;    //栈空返回
    StackNode *p = top;    //记录退栈结点
    top = top->link;    //退栈
    x = p->data;    //返回退栈元素
    delete p;    //释放结点
    return true;
};
```

时间复杂性: $O(1)$

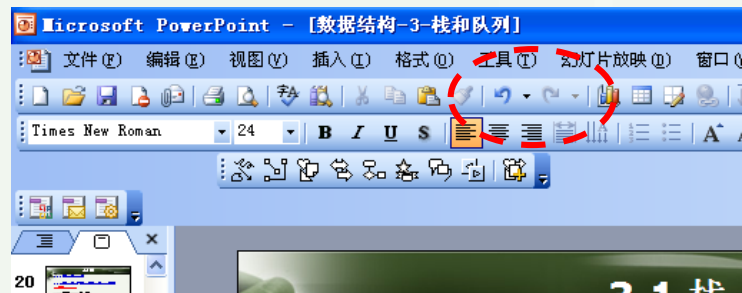
其他链栈的操作请同学们自己写出!

3.1 栈

3.1.3 栈ADT的应用

栈数据结构由于其具有“先进后出”的特性，是一种应用非常广泛的数据结构，在人或计算机求解现实问题过程中，你可能会没意识到，但是确实使用了它！

- 你的信件管理
- 你做家务洗盘子
- 迷失了回家的路？
- 你经常用浏览器和各种编辑软件，里面有栈应用，在哪里？



内容小结

1. 栈是特殊的线性表，那么：
和一般线性表相比特殊在哪里？
2. 栈ADT的定义
3. 栈ADT的实现的方式有几种？
 - 栈顶和栈底是怎么设置的？
 - 各有什么优缺点？
 - 栈的基本操作的时间复杂性如何？
4. 有3个元素，其入栈顺序为a,b,c。那么可能得到abc的
出栈序列吗？ 能得到acb、 cab吗？

3.1 栈

3.1.3 栈ADT的应用

由于栈的“后进先出”特性，在很多实际问题中都利用栈做一个辅助的数据结构来进行求解。

栈应用举例之一： **进制转换**

问题描述： 将一个非负的十进制整数 N 转换为另一个等价的基为 B 的 B 进制数。

算法基本思想： “除留余法”

关键： 最先得到的余数是转化结果的最低位，最后得到的余数是转化结果的最高位。

如何解决： 利用栈（当然也有其他方法）

3.1 栈

3.1.3 栈ADT的应用

栈应用举例之一：进制转换

确定数据元素类型 ElemType
用栈ADT说明一个栈 S

重复：

$x = N \% B$; // 得到一位

S.Push(x); //入栈

$N = N / B$; //得到商

直到 N为0

重复：

S.Pop(x); //出栈

输出x;

直到**S.IsEmpty()**为真

3.1 栈

3.1.3 栈ADT的应用

栈应用举例之二：平衡符号

问题描述：在语言中往往需要判断一些符号是否是成对出现的，比如<>, {}, [], ()等等，通常在C++中也只有这几种符号的对称问题。那么，如何让计算机自动判断符号是否对称？

算法基本思想：读符号，判断是对称符号，如果是“左”出现则记录下来，如果是“右”出现则匹配，看看最终是否都匹配上。

关键：“后出现的先匹配”——栈

3.1 栈

3.1.3 栈ADT的应用

栈应用举例之二：平衡符号

确定数据元素类型 `char`
用栈ADT说明一个栈 `S`

重复：

读字符`ch`，如果是对称符号，则：

“左”出现，**`S.Push(ch)`**；//入栈

“右”出现，**`S.IsEmpty()`**为真，则失衡；

否则，则**`S.Pop(x)`**；`x`与`ch`是否匹配

直到 所有字符处理完毕；

`S.IsEmpty()`为真，则符号平衡；

否则，则符号失衡

3.1 栈

3.1.3 栈ADT的应用

栈应用举例之三： 表达式求值

问题描述：计算机对数据的最基本处理就是表达式求值，即运算量按照运算符去执行运算操作，得到结果值。请设计算法求表达式的值。

表达式的一般形式：

运算量₁

运算符

运算量₂

- 运算量：可以常量、变量，也可以表达式
- 常量、变量可以看作是最简单的表达式
- 运算量1可以没有（单目运算）

3.1 栈

3.1.3 栈ADT的应用

栈应用举例之三： 表达式求值

表达式的种类(以C语言为例):

- 算术表达式: 算术运算符 $+$ $-$ $*$ $/$ $\%$ $++$ $--$
- 关系表达式: 关系运算符 $>$ $>=$ $<$ $<=$ $!=$
- 逻辑表达式: 逻辑运算符 $\&\&$ $\|\$ $!$
- 赋值表达式: 赋值运算符 $=$ $+=$ $-=$ $*=$ $/=$ $\%=$
- 逗号表达式: 逗号运算符 $,$
- 条件表达式: 条件运算符 $?$ $:$
- 成员运算表达式: 成员运算符 $.$ $->$
- 指针运算表达式: 指针运算符 $*$ $\&$

3.1 栈

3.1.3 栈ADT的应用

栈应用举例之三： 表达式求值

表达式的书写形式： 中缀式、前缀式、后缀式

例如，表达式： $a \times b + (c - d / e) \times f$

前缀式： $+ \times a b \times - c / d e f$

中缀式： $a \times b + (c - d / e) \times f$

后缀式： $a b \times c d e / - f \times +$

- 1) 操作数之间的相对次序不变;
- 2) 运算符的相对次序不同;
- 3) 中缀式有操作符的优先级问题，还有可加括号改变运算顺序的问题，所以编译程序一般不使用中缀表示处理表达式。

3.1 栈

3.1.3 栈ADT的应用

栈应用举例之三： 表达式求值

4)前缀式的运算规则为:

连续出现的两个操作数和它们在之前且紧靠它们的运算符构成一个最小表达式;

5)后缀式的运算规则为:

运算符在式中出现的顺序恰为表达式的运算顺序;
每个运算符和在它之前出现且紧靠它的两个操作数构成一个最小表达式。

3.1 栈

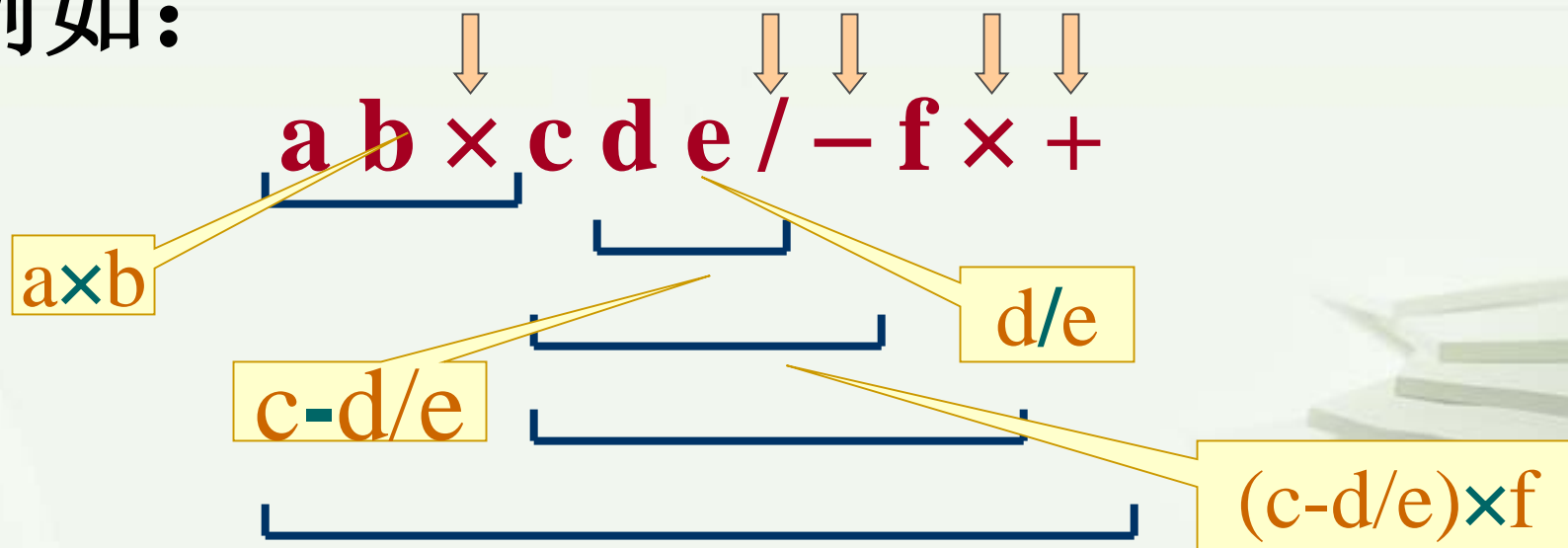
3.1.3 栈ADT的应用

栈应用举例之三：表达式求值

■ 后缀式求值

求值规则：运算符前面相邻的运算量参加该运算符的运算，结果是一个运算量。

例如：



3.1 栈

3.1.3 栈ADT的应用

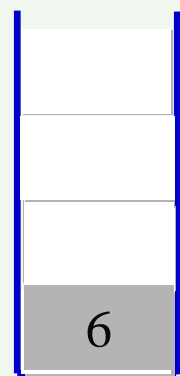
栈应用举例之三： 表达式求值

算法基本思想：从左边读表达式，遇到运算符时，取前面刚刚读过去的两个运算量参加运算，结果是一个新的运算量参与后面的运算。

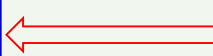
关键：用一个栈来辅助，存放运算量。

例如，计算机后缀表达式： $1\ 2\ +\ 8\ 2\ -\ 7\ 4\ -\ /\ *$ 的值

求值过程：



运算量栈



结果值为6



3.1 栈

3.1.3 栈ADT的应用

栈应用举例之三： 表达式求值

算法步骤：

- (1) 设置一栈，辅助求值。开始置空 **SeqStack S(50)**
- (2) 从左向右，读入表达式的每一个项 **t**
- (3) 如果 **t** 是运算量，则压入堆栈 **S.Push(t)**
- (4) 如果 **t** 是运算符，则从堆栈中弹出此运算符需要用到的运算量，**S.Pop(b), S.Pop(a)**，进行相应运算得到结果值 **c**，压入堆栈 **S.Push(c)**
- (5) 如果表达式处理完毕，栈中的数据就是表达式的值 **S.getTop(x)**

见教材 P98 程序3.10

3.1 栈

3.1.3 栈ADT的应用

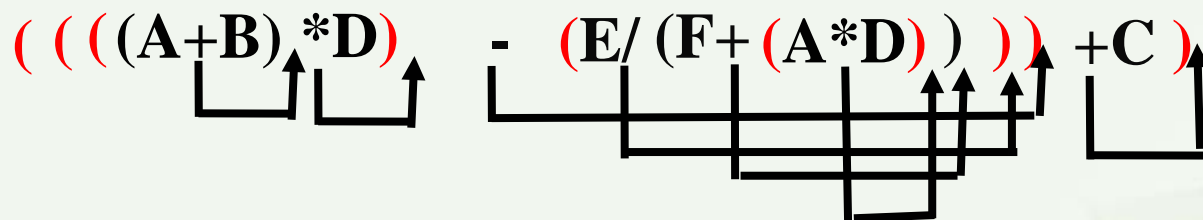
栈应用举例之三： 表达式求值

■ 中缀式求值

方法1：把中缀表达式转换为后缀表达式，求值。

中缀变后缀的人工方法：先对中缀表达式按“运算优先次序”加上括号，再把操作符后移到所在层右括号的后面，最后将所有括号消去。

例： $(A+B)*D-E/(F+A*D)+C$



后缀表示： $A B + D * E F A D * + / - C +$

3.1 栈

3.1.3 栈ADT的应用

栈应用举例之三： 表达式求值

■ 中缀式求值

中缀变后缀的计算机方法：—**算符优先法：优先级高的先计算。**

基本思想：确定算符的优先级，只有确定一个算符的优先级比左右的优先级高，它才可以运算—输出算符。

对于两个相继出现的算符 θ_1 、 θ_2 ，它们优先关系为：

$\theta_1 < \theta_2$ ： θ_1 的优先级低于 θ_2 ，或 θ_2 的优先级高于 θ_1 ；

$\theta_1 = \theta_2$ ： θ_1 的优先级等于 θ_2 （特殊情况）

$\theta_1 > \theta_2$ ： θ_1 的优先级大于 θ_2

3.1 栈

3.1.3 栈ADT的应用

栈应用举例之三： 表达式求值

■ 中缀式求值

单个算符的优先级

由高到低

优先级 运算符

- | | |
|---|--------------------------|
| 1 | 括号() |
| 2 | 负号- |
| 3 | 乘方** |
| 4 | 乘*, 除/, 求余% |
| 5 | 加+, 减- |
| 6 | 小于<, 小于等于<=, 大于>, 大于等于>= |
| 7 | 等于==, 不等于!= |
| 8 | 逻辑与&& |
| 9 | 逻辑或 |

3.1 栈

3.1.3 栈ADT的应用

栈应用举例之三：表达式求值

■ 中缀式求值

两个算符的优先级比较

θ_1 、 θ_2 是相继出现的两个算符。

θ_1 —栈内
 θ_2 —栈外

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

注：为了处理方便，用#作为表达式的界符。

3.1 栈

3.1.3 栈ADT的应用

栈应用举例之三： 表达式求值

关键：用栈辅助存放算符，确定算符优先级。

算法：

① 操作符栈初始化，将界符 ‘#’ 压进栈。然后读入中缀表达式字符流的首字符ch。

② 重复执行a，b步骤，直到ch = ‘#’，同时栈顶的操作符也是 ‘#’，停止循环，执行③。

a. 若ch是运算量，则直接输出，读入下一个字符ch。

b. 若ch是运算符，判断ch和栈顶操作符op的优先级

- 若 $ch > op$ ，则ch进栈，读入下一个字符ch。
- 若 $ch < op$ ，op优先级高，op退栈，输出。
- 若 $ch = op$ ，op是 “(”，退栈，读入下一个字符ch。

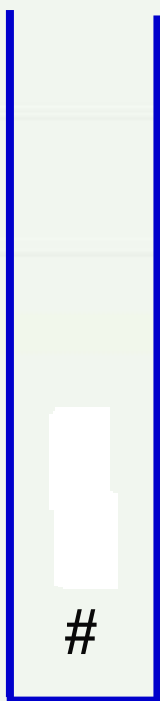
③ 结束。输出序列即为后缀表达式。

3.1 栈

3.1.3 栈ADT的应用

栈应用举例之三：表达式求值

例如：写出中缀表达式 $\#(a+b)/5-x*y/c\#$ 的后缀形式



运算符栈

$\# (a + b) / 5 - x * y / c \#$

a b + 5 / x y * c / -

3.1 栈

3.1.3 栈ADT的应用

栈应用举例之三： 表达式求值

■ 中缀式求值

方法2：直接对中缀表达式求值。

基本思想：思想和中缀转换为后缀基本一样。
设置两个栈：一个存放运算符的栈 **OPTR**，
一个存放运算量的栈 **OPND**；

在方法1中，把运算量的输出改为“入**OPND**栈”，把运算符的“输出”改为“计算”：根据运算符，出栈需要的运算量，计算值，结果作为运算量再入栈。其他不需要变化。

3.1 栈

3.1.3 栈ADT的应用

栈应用举例之三： 表达式求值

算法
如右

```
OperandType EvaluateExpression()
{ //OP为运算符集合
  InitStack(OPTR); Push(OPTR,'#');
  InitStack(OPND); c=getchar();
  while(c!='#' || GetTop(OPTR)!='#')
  { if(!In(c,OP)) { Push(OPND,c); c=getchar(); }
    else switch (Precede(GetTop(OPTR),c))
      { case '<': Push(OPTR,c); c=getchar(); break;
        case '=': Pop(OPTR,x); c=getchar(); break;
        case '>': Pop(OPTR,theta); Pop(OPND,b); Pop(OPND,a);
                  Push(OPND,Operate(a,theta,b)); break;
        }
  }
  return(Gettop(OPND));
}
```

3.1 栈

3.1.3 栈ADT的应用

栈应用举例之三： 表达式求值

例如： $\# -3*((7-2)+5)/2\#$



内容小结

1. 栈是一种非常简单、但应用非常广泛、非常重要的数据结构。那么主要用它做什么呢？
2. 表达式的书写形式有几种？ 有什么不同？
3. 后缀表达式如何求值？
4. 中缀表达式如何转化为后缀表达式？ 写出下面表达式的后缀表达式。
(人工做、计算机做) $a+b*c+(d*e+f)*g$
5. 中缀表达式如何直接求值？

$$5*(2+6)/4-((8-4)/2+7)*2$$

3.1 栈

3.1.3 栈ADT的应用

栈应用举例之四：函数调用—递归处理

■ 递归的基本概念

1. 递归的定义：

如果一个事物或对象 **部分** 地由它自身构成或定义。

2. 递归过程（函数）：

过程（函数）直接或间接调用自己。

3. 递归举例

♥ $n = n(n-1)!$

♥ Hanoi塔问题

♥ 求n个数的和(一个数加上其他 **n-1个数的和**)

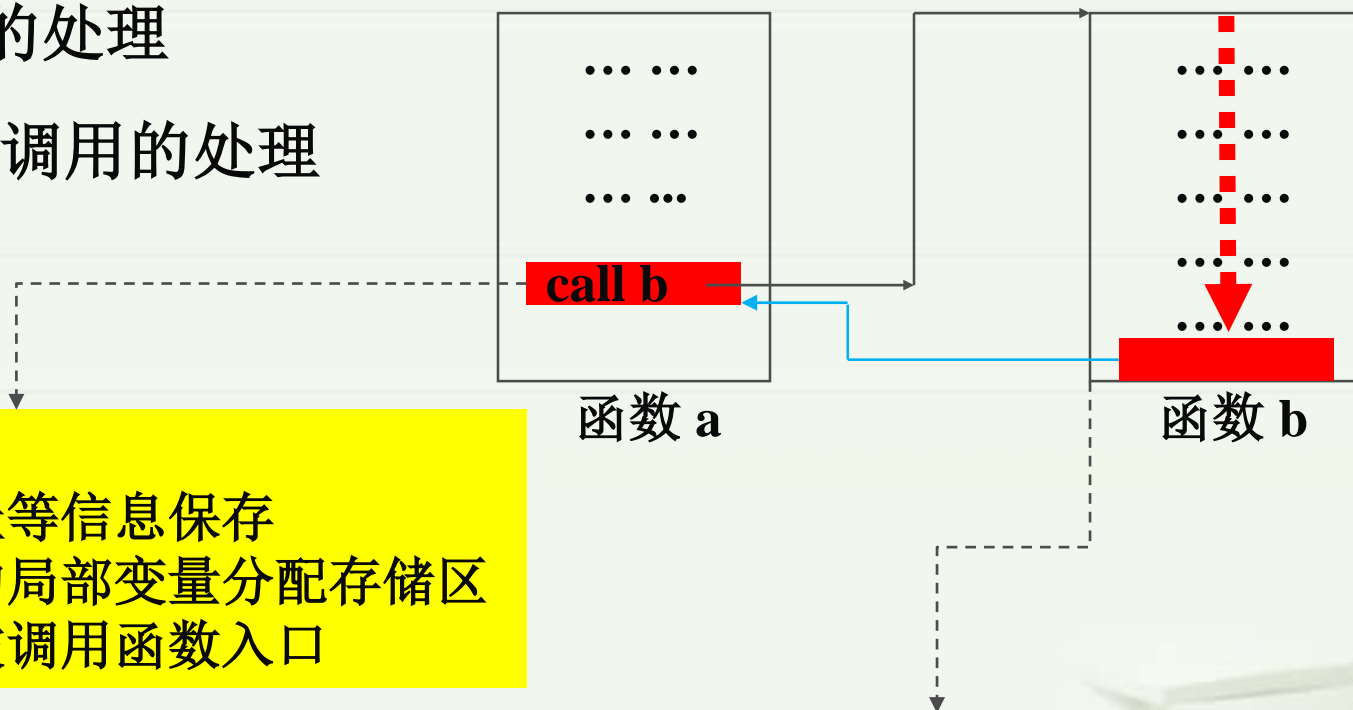
3.1 栈

3.1.3 栈ADT的应用

栈应用举例之四：函数调用—递归处理

■ 递归函数的处理

1. 一般函数调用的处理



调用前:

- 参数、返回地址等信息保存
- 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入口

调用后 (返回):

- 保存返回参数值 (包括函数值)
- 释放被调用函数的存储区
- 将控制转移到调用函数的返回地址处

3.1 栈

3.1.3 栈ADT的应用

栈应用举例之四：函数调用—递归处理

■ 递归函数的处理

因此：计算机在处理函数调用时，必须开辟内存空间保存“现场”。而且为了确保不会导致调用混乱，应该遵循“后调用的先返回”的原则，所以保存现场的存储空间的组织应该遵循“栈”结构，即“先保存的现场后恢复”。

栈工作记录：一次调用保存的信息（现场），称为一个工作记录（栈元素）。包括：返回地址、参数、局部变量。

3.1 栈

3.1.3 栈ADT的应用

栈应用举例之四：函数调用—递归处理

■ 递归函数的处理

2. 递归函数调用的处理

递归函数调用与一般函数调用类似，而且每次都是调用自己，只是现场非常相似而已，所以管理更要准确！



3.1 栈

3.1.3 栈ADT的应用

栈应用举例之四：函数调用—递归处理

■ 递归过程的处理

2. 递归函数调用的处理

递归调用：——**压入一工作记录**

- 参数、返回地址等信息保存
- 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入口

递归返回：——**弹出一工作记录**

- 保存返回参数值（包括函数值）
- 释放被调用函数的存储区
- 将控制转移到调用函数的返回地址处

3.1 栈

3.1.3 栈ADT的应用

栈应用举例之四：函数调用—递归处理

■ 递归过程的处理

4. 递归的特点

- ◆ 结构清晰、可读性强；
- ◆ 算法正确性易证明（数学归纳法）；
- ◆ 运行效率低（时间—栈操作、空间—栈空间）

3.1 栈

3.1.3 栈ADT的应用

栈应用举例之四： 函数调用—递归处理

注：递归部分要求：

- (1) 对递归概念的理解；
- (2) 能设计递归算法；
- (2) 了解递归算法的执行过程（栈的作用）

3.2 队列

3.2.1 队列ADT的定义

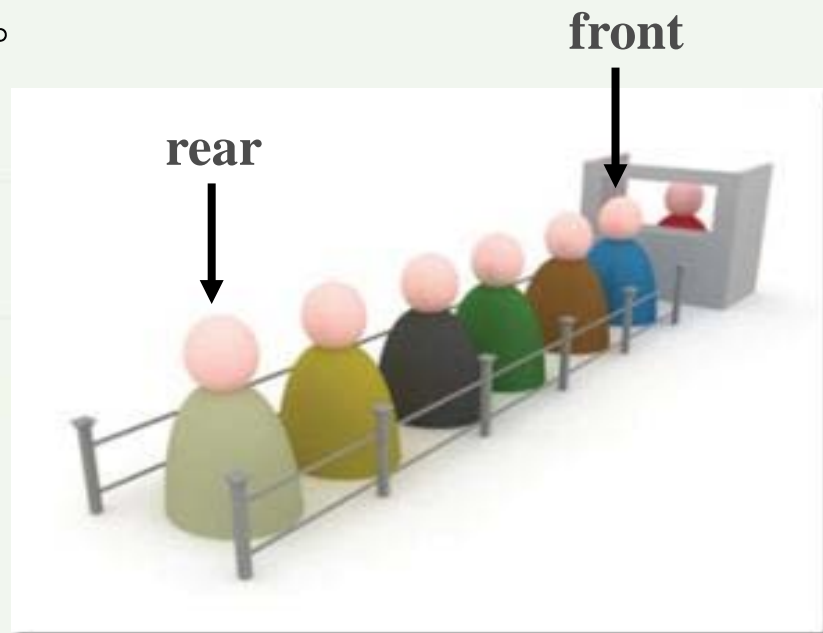
[队列 **Queue**] 是一种特殊的线性表（数据元素之间的关系是线性关系），其插入、删除操作分别在表的两端进行，一端只能插入、另一端只能删除。

队首(front): 进行删除的一端；

队尾(rear): 进行插入的一端；

入队: 在队尾插入一个元素；

出队: 在队首删除一个元素；



特性:元素的操作顺序符合“先进先出(FIFO)”
或“后进后出(LILO)”。

3.2 队列

3.2.1 队列ADT的定义

ADT Queue

data structure:

$D = \{a_i \mid a_i \in D_0 \ i=1,2,\dots, n \geq 0\}$

$R = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D_0 \ i=2,3,4,\dots \}$

D_0 是某个数据对象

operations:

Inqueue(Q);

Enqueue(Q,x);

Dequeue(Q,x);

GetFront(Q,x);

IsEmpty(Q);

Current_size(Q);

Clear(Q);

END

队列ADT定义:

3.2 队列

3.2.1 队列ADT的定义

队列ADT的抽象类定义：

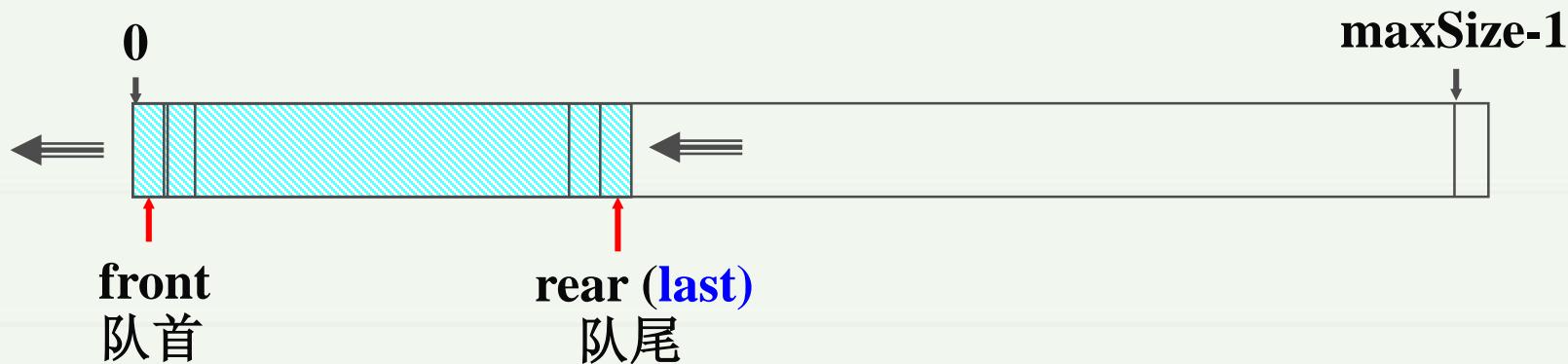
```
typedef 数据元素类型 ElemType;
class Queue
{ public:
    Queue() { };           //构造函数
    ~Queue() { };         //析构函数
    virtual bool EnQueue(ElemType x) = 0;    //进队列
    virtual bool DeQueue(ElemType& x) = 0;    //出队列
    virtual bool GetFront(ElemType& x) = 0;   //取队头
    virtual bool IsEmpty() const = 0;         //判队列空
    virtual bool IsFull() const = 0;          //判队列满
};
```

3.2 队列

3.2.2 队列ADT的实现

3.2.2.1 队列ADT的基于顺序存储的实现——顺序队列

1. 存储方式：同一般线性表的顺序存储结构完全相同。



存储的具体实现：

与栈不同，两端都有操作，为了方便，需设置两个指示器；

```
#define maxSize 队列最大长度
typedef 数据元素类型 ElemType;
typedef struct
{ ElemType elem[maxSize]; //连续空间
  int front; //队首指示器
  int rear; //队尾指示器
} SeqQueue;
```

3.2 队列

3.2.2 队列ADT的实现

3.2.2.1 队列ADT的基于顺序存储的实现——顺序队列

1. 存储方式：同一般线性表的顺序存储结构完全相同。

例如： `SeqQueue q, que;`

那么，队列的头、尾如何设置？

假设队首在地址低端，队尾在地址高端，队列的操作如何进行？

队列置空： `q.front=0; q.rear=0;`

入队： `q.elem[q.rear]=x; q.rear++;`

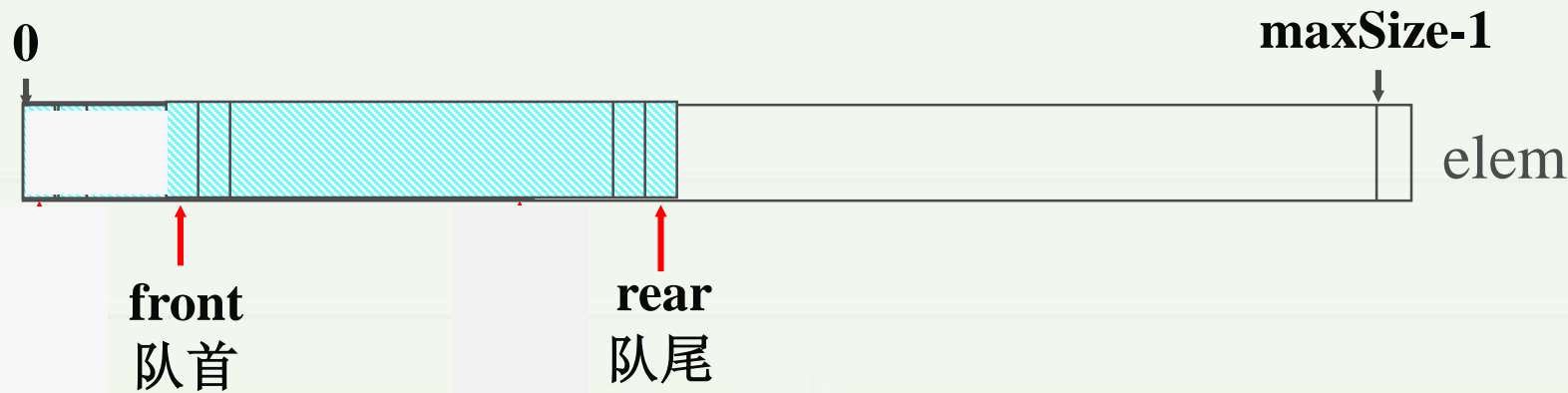
出队： `x=q.elem[q.front]; q.front++;`

3.2 队列

3.2.2 队列ADT的实现

3.2.2.1 队列ADT的基于顺序存储的实现——顺序队列

1. 存储方式：同一般线性表的顺序存储结构完全相同。



取队首:

`q.elem[q.front]`?

求队列长度:

`q.rear-q.front+1`?

判断是否空:

`q.rear==q.front+1`?

判断是否满:

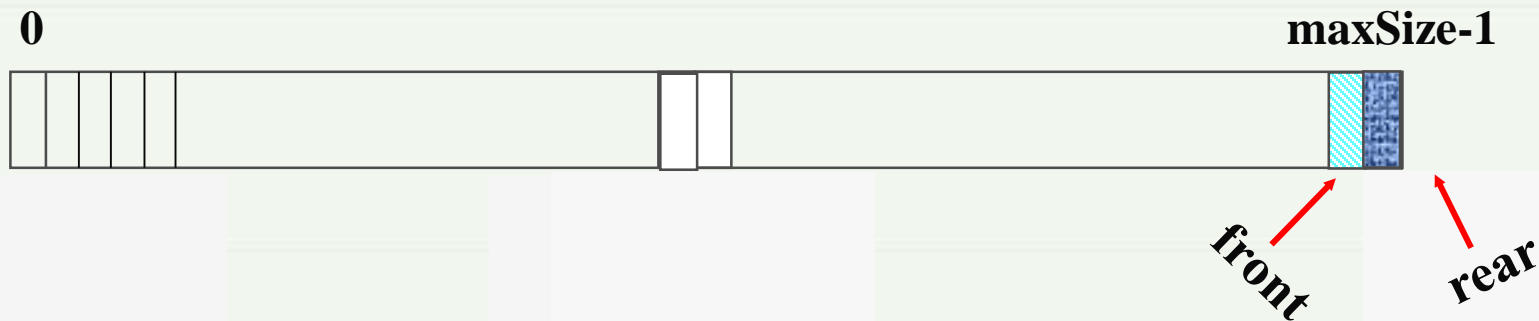
`q.rear==maxSize-1`?

队首在地址高端、
队尾在地址低端，
情况如何？

3.2 队列

3.2.2 队列ADT的实现

3.2.2.1 队列ADT的基于顺序存储的实现——顺序队列



队列开始空(置空): $\text{q.front}=\text{q.rear}=0$

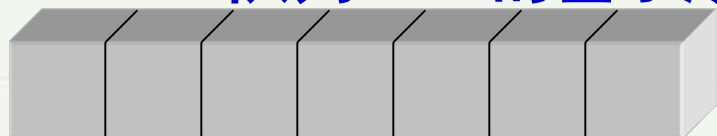
判断队列是否空: $\text{q.front}==\text{q.rear} ?$

判断队列是否满: $\text{q.rear}==\text{maxSize} ?$

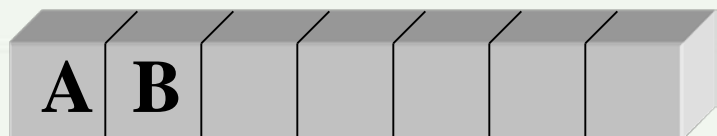
3.2 队列

3.2.2 队列ADT的实现

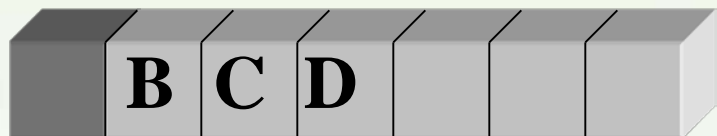
3.2.2.1 队列ADT的基于顺序存储的实现——顺序队列



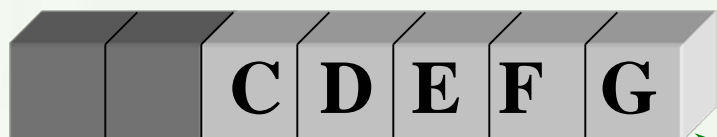
front rear 空队列



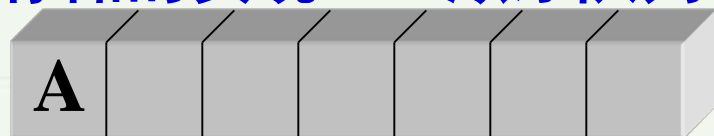
front rear B进队



front rear A退队



front rear E,F,G进队



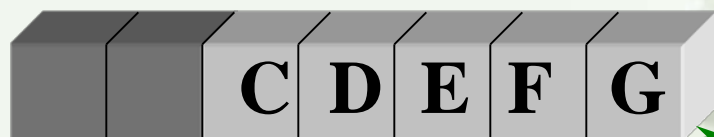
front rear A进队



front rear C,D进队



front rear B退队 假溢出



front rear H进队,溢出

3.2 队列

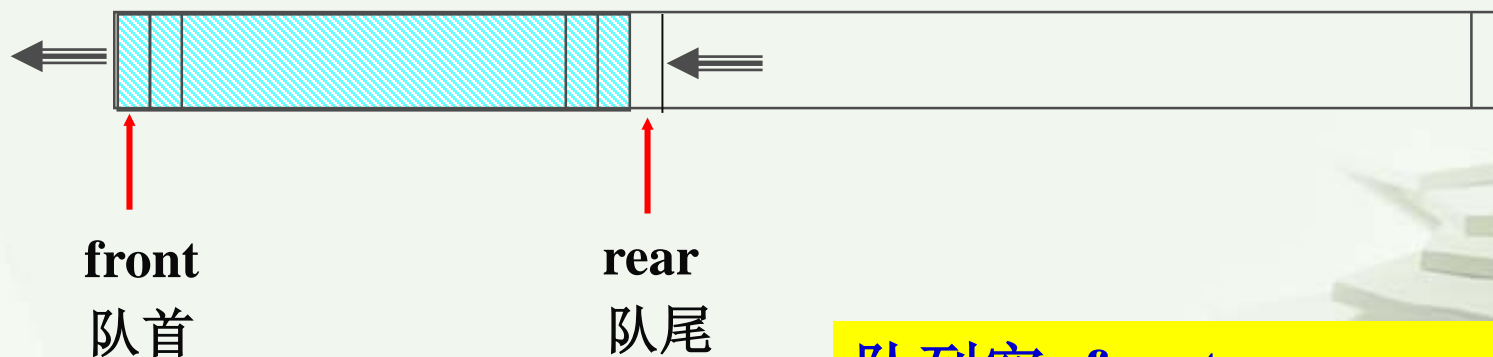
3.2.2 队列ADT的实现

3.2.2.1 队列ADT的基于顺序存储的实现——顺序队列

1. 存储方式：同一般线性表的顺序存储结构完全相同。

(1) 假溢出问题：后面空间占满了，而由于有出队，前面还有空间。——假溢出

解决办法之一：固定队首位置，每出队一个元素，队列中的其他元素前移动



队列空: $\text{front} = \text{rear}$

队列真满: $\text{rear} = \text{maxsize}$

3.2 队列

3.2.2 队列ADT的实现

3.2.2.1 队列ADT的基于顺序存储的实现——顺序队列

1. 存储方式：同一般线性表的顺序存储结构完全相同。

解决办法之二：把地址空间“人为”形成环状，即后面满了，但如果前面因出队还有空间的话，就把地址“接”上。
——环队列

模运算%可实现空间“形成环”：

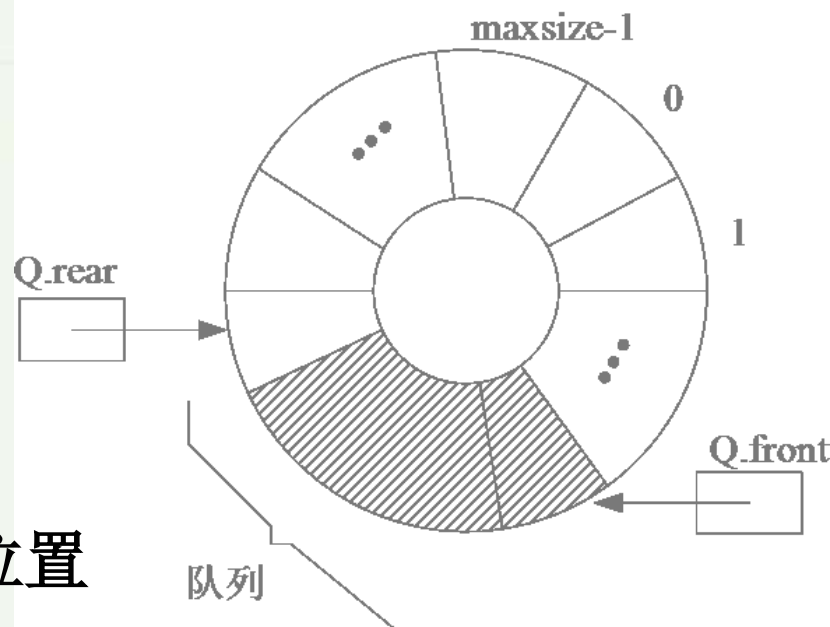
$i \% \text{maxSize}$

队首指示器**front**：

——指向队列队首元素

队尾指示器**rear**：

——指向最后一个元素的后一个位置



3.2 队列

3.2.2 队列ADT的实现

3.2.2.1 队列ADT的基于顺序存储的实现——顺序队列

队列初始化:

$q.front = q.rear = 0;$

队空条件:

$q.front == q.rear;$

出队:

$y = q.elem[q.front]$

$q.front = (q.front + 1) \% maxSize;$

入队:

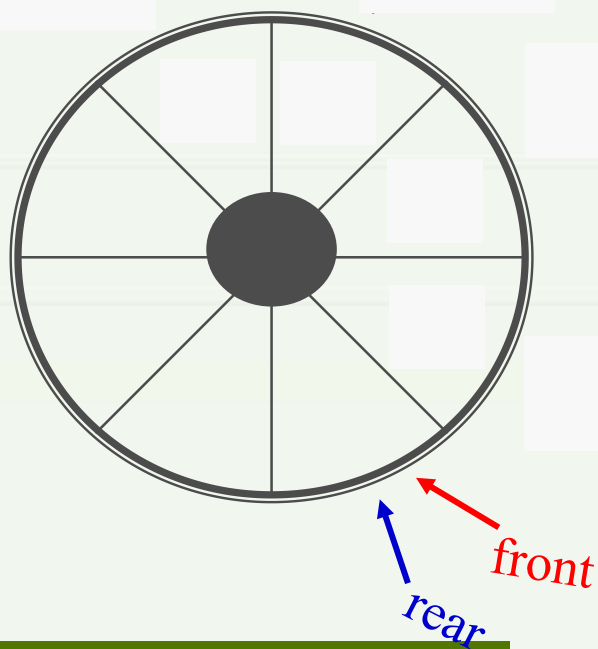
$q.elem[q.rear] = x$

$q.rear = (q.rear + 1) \% maxSize;$

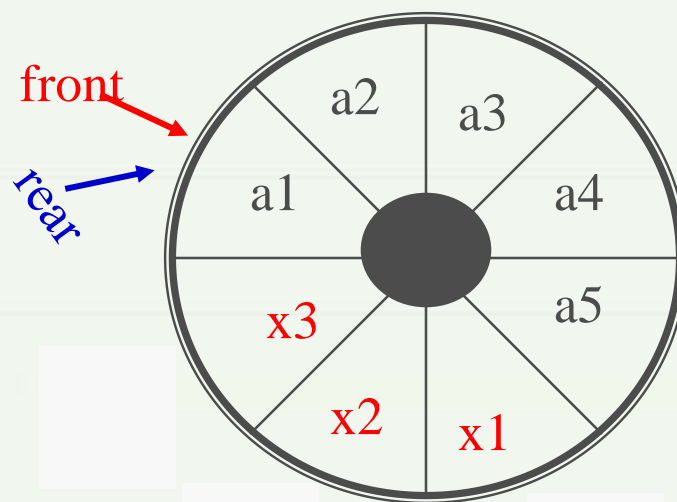
3.2 队列

3.2.2 队列ADT的实现

3.2.2.1 队列ADT的基于顺序存储的实现——顺序队列



队列为空：
 $q.front = q.rear$



队列为满：
 $q.front = q.rear$

3.2 队列

3.2.2 队列ADT的实现

3.2.2.1 队列ADT的基于顺序存储的实现——顺序队列

显然：无法区分队列是空还是满！！

处理方法有两种：

其一：设一个布尔变量来注明队列是空还是满；

其二：约定队列元素达到 $\text{maxsize}-1$ 个时，队列为满
即 $\text{rear}+1=\text{front}$

判断队列空：

$\text{rear}==\text{front}$

判断队列满：

$(\text{rear}+1)\% \text{maxSize}==\text{front}$

3.2 队列

3.2.2 队列ADT的实现

3.2.2.1 队列ADT的基于顺序存储的实现——顺序队列

2. 具体实现——循环队列类定义

```
typedef 数据元素类型 ElemType;
class SeqQueue      //队列类定义
{   protected:
    int rear, front;      //队尾与队头指示器
    ElemType *elements;   //队列存放数组
    int maxSize;          //队列最大容量
public:
    SeqQueue(int sz = 10);      //构造函数
    ~SeqQueue() { delete[ ] elements; } //析构函数
```


3.2 队列

3.2.2 队列ADT的实现

3.2.2.1 队列ADT的基于顺序存储的实现——顺序队列

2. 具体实现——循环队列类定义

```
bool EnQueue(ElemType x);    //元素入列
bool DeQueue(ElemType& x);    //队头元素出队
bool getFront(ElemType& x);    //取队头元素值
void makeEmpty() { front = rear = 0; }
bool IsEmpty() const { return front == rear; }
bool IsFull() const
    { return ((rear+1)% maxSize == front); }
int getSize() const
    { return (rear-front+maxSize) % maxSize; }
};
```

3.2 队列

3.2.2 队列ADT的实现

3.2.2.1 队列ADT的基于顺序存储的实现——顺序队列

(1) 构造函数

```
SeqQueue::SeqQueue(int sz) //构造函数
{
    front=0; rear=0; maxSize=sz;
    elements = new DataType[maxSize];
    assert ( elements != NULL );
};
```

(2) 入队

```
bool SeqQueue::EnQueue(ElemType x)
{
    if (IsFull() == true) return false;
    elements[rear] = x;
    rear = (rear+1) % maxSize;
    return true;
};
```

3.2 队列

3.2.2 队列ADT的实现

3.2.2.1 队列ADT的基于顺序存储的实现——顺序队列

(3) 出队

```
bool SeqQueue::DeQueue(ElemType& x)
{   if (IsEmpty() == true) return false;
    x = elements[front];
    front = (front+1) % maxSize;
    return true;
};
```

3.2 队列

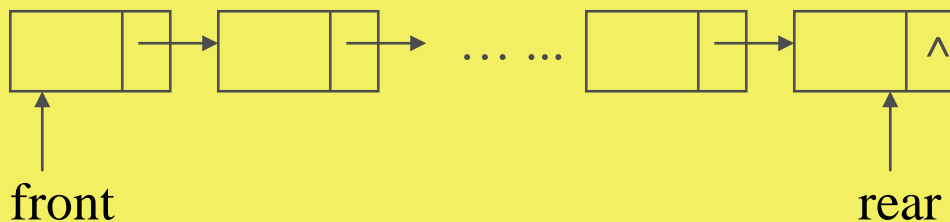
3.2.2 队列ADT的实现

3.2.2.2 队列ADT的基于单链存储的实现——链队列

1. 存储方式：同一般线性表的不带头结点的单链存储一样

但是由于插入、删除在两端进行，需要两个指针front、rear分别指向队列的两端。

队首队尾设置有两种形式：哪个好？



3.2 队列

3.2.2 队列ADT的实现

3.2.2.2 队列ADT的基于单链存储的实现——链队列

2. C++类实现

链队列的类定义：

```
#include <iostream.h>
typedef 数据元素类型 ElemType;
struct QueueNode //队列结点类定义
{ private:
    ElemType data; //队列数据元素
    QueueNode *link; //结点链指针
public:
    QueueNode(ElemType d , QueueNode
    *next = NULL) : data(d), link(next) { }
};
```

3.2 队列

3.2.2 队列ADT的实现

3.2.2.2 队列ADT的基于单链存储的实现——链队列

```
class LinkedQueue //队列类定义
{ private:
    QueueNode *front, *rear; //队头、队尾指针
public:
    LinkedQueue() : rear(NULL), front(NULL) { }
    ~LinkedQueue(MakeEmpty());
    bool EnQueue(ElemType x);
    bool DeQueue(ElemType & x);
    bool GetFront(ElemType & x);
    void MakeEmpty();
    bool IsEmpty() const { return (front == NULL)?true:false ;}
    int GetSize() const;
    friend ostream& operator<<(ostream&os, LinkedQueue &Q)
};
```

3.2 队列

3.2.2 队列ADT的实现

3.2.2.2 队列ADT的基于单链存储的实现——链队列

成员函数的实现：

① 置空队列操作：同单链表释放

```
LinkedQueue::MakeEmpty()  
{ //释放链表中所有结点  
    QueueNode *p;  
    while (front != NULL) //逐个释放结点  
    { p = front;  
      front = front->link;  
      delete p;    }  
};
```

3.2 队列

3.2.2 队列ADT的实现

3.2.2.2 队列ADT的基于单链存储的实现——链队列

② 入队

```
bool LinkedListQueue::EnQueue(ElemType x)
{ if (front == NULL) //创建第一个结点
  { front = rear = new QueueNode (x);
    if (front == NULL) return false; } //分配失败
else //队列不空, 插入
  { rear->link = new QueueNode(x);
    if (rear->link == NULL) return false; //分配失败
    rear = rear->link; }
return true;
};
```


3.2 队列

3.2.2 队列ADT的实现

3.2.2.2 队列ADT的基于单链存储的实现——链队列

③ 出队

```
bool LinkedQueue::DeQueue(ElemType & x)
{
    if (IsEmpty() == true) return false;    //判队空
    QueueNode *p = front;
    x = front->data;
    front = front->link;
    delete p;
    return true;
};
```

3.2 队列

3.2.2 队列ADT的实现

3.2.2.2 队列ADT的基于单链存储的实现——链队列

④ 取队首元素

```
bool LinkedQueue::GetFront(ElemType & x)
{
    if (IsEmpty() == true) return false;
    x = front->data;
    return true;
};
```

3.2 队列

3.2.2 队列ADT的实现

3.2.2.2 队列ADT的基于单链存储的实现——链队列

⑤ 求队长度：同求不带头结点的单链表长度

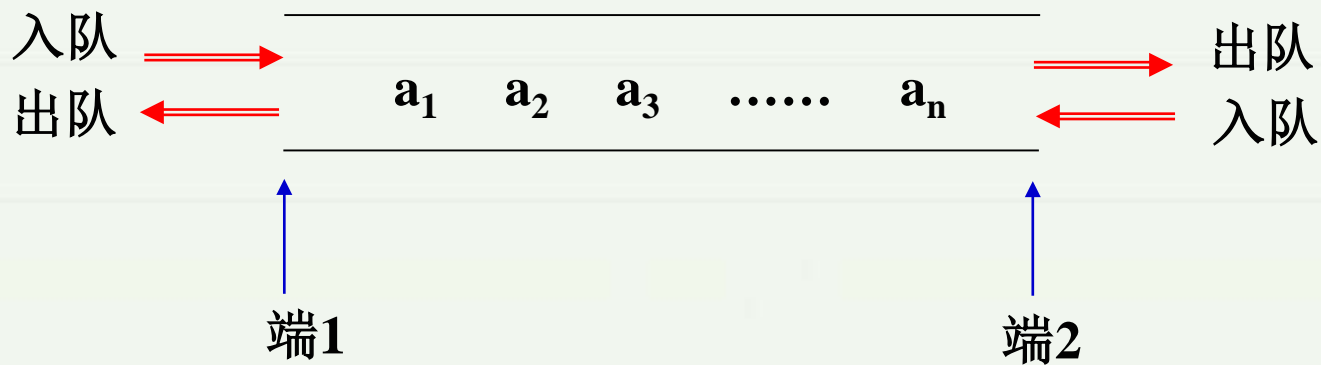
```
int LinkedQueue::GetSize( )const
{
    QueueNode *p = front;
    int k=0;
    while(p!=NULL)
    { k++;
      p= p->link; }
    return k;
};
```

3.2 队列

3.2.3 队列的变形

3.2.3.1 双端队列

[双端队列 Double-ended Queue] 可以在两端进行插入、删除的队列。



端1操作:

- 取队首元素
- 入队
- 出队

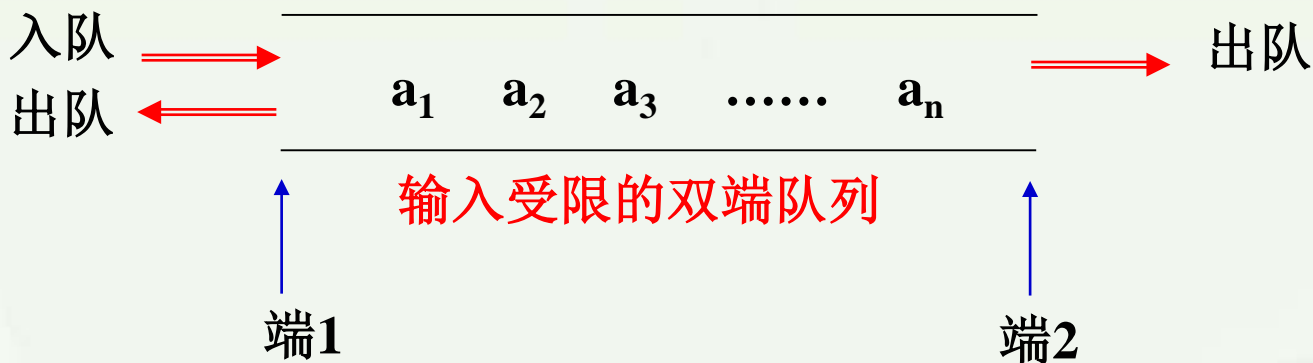
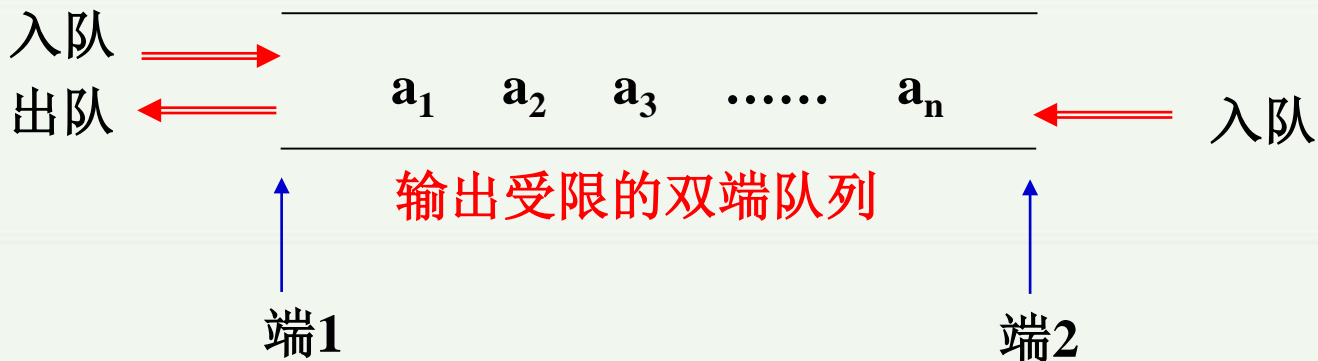
端2操作:

- 取队尾元素
- 入队
- 出队

3.2 队列

3.2.3 队列的变形

3.2.3.1 双端队列



双端队列的实现：顺序存储，链式存储。 **p126-313, 不要求!**

3.2 队列

3.2.3 队列的变形

3.2.3.2 优先队列

[优先队列 Priority Queue] 同一般队列类似，在一端进行插入、一端进行删除。但是每个元素有优先级（优先数），优先级高的先出队。

操作：

- 取队首元素：优先级最高的元素
- **入队**：把元素插入在队尾，同时根据优先数调整顺序
- **出队**：删除队首元素（优先数最高的）

关键：

- 始终保持优先数高的元素在队首。

3.2 队列

3.2.3 队列的变形

3.2.3.2 优先队列

方法:

(1) 当有元素入队后, 按优先数排序。

排序方法很多, 常用插入排序方法:

“在一个有序序列中插入一个元素, 使其仍然有序”

$O(n^2)$

(2) 当有元素入队后, 调整成“堆”, 根据堆的性质:
堆顶的元素就是优先数最高的元素。 $O(\log_2 n)$

p124-126, 不要求!

3.2 队列

3.2.4 队列ADT的应用

队列由于具有“**先进先出**”的特性，也是一种应用非常广泛的线性数据结构。

例如，计算机主机与外部设备之间速度不匹配的问题的处理。

策略： 设置一个“先进先出”打印数据缓冲区，主机把要打印输出的数据送入缓冲区（入队），队满后就暂停输出，继而去做其它的事情，打印机则从缓冲区中取出数据并打印（出队）。打印完后再向主机发出请求，主机接到请求后再向缓冲区写入打印数据，这样利用队列既保证了打印数据的正确，又使主机提高了效率。

3.2 队列

3.2.4 队列ADT的应用

队列应用举例之一：打印杨辉三角

问题描述：将二项式 $(a+b)^i$ 展开，其系数构成杨辉三角。现要求把前 n 行打印出来。

				1		1				$i = 1$		
			1		2		1			2		
		1		3		3		1		3		
		1		4		6		4		4		
	1		5		10		10		5		5	
1		6		15		20		15		6		6

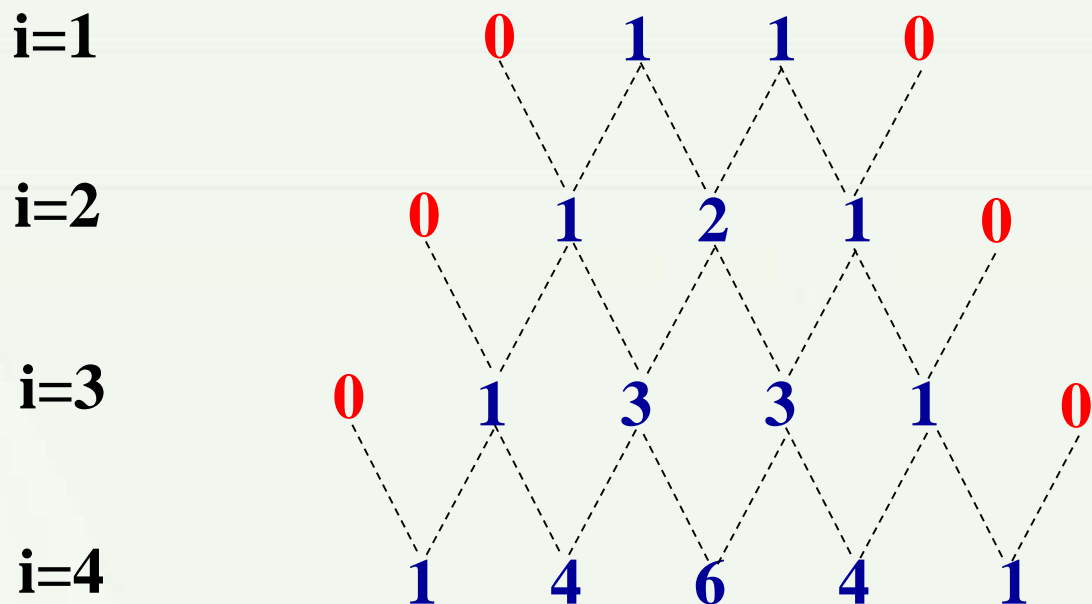
3.2 队列

3.2.4 队列ADT的应用

队列应用举例之一：打印杨辉三角

分析：由前一行系数可以生成下一行的系数。

第 i 行元素与第 $i+1$ 行元素的关系如下：

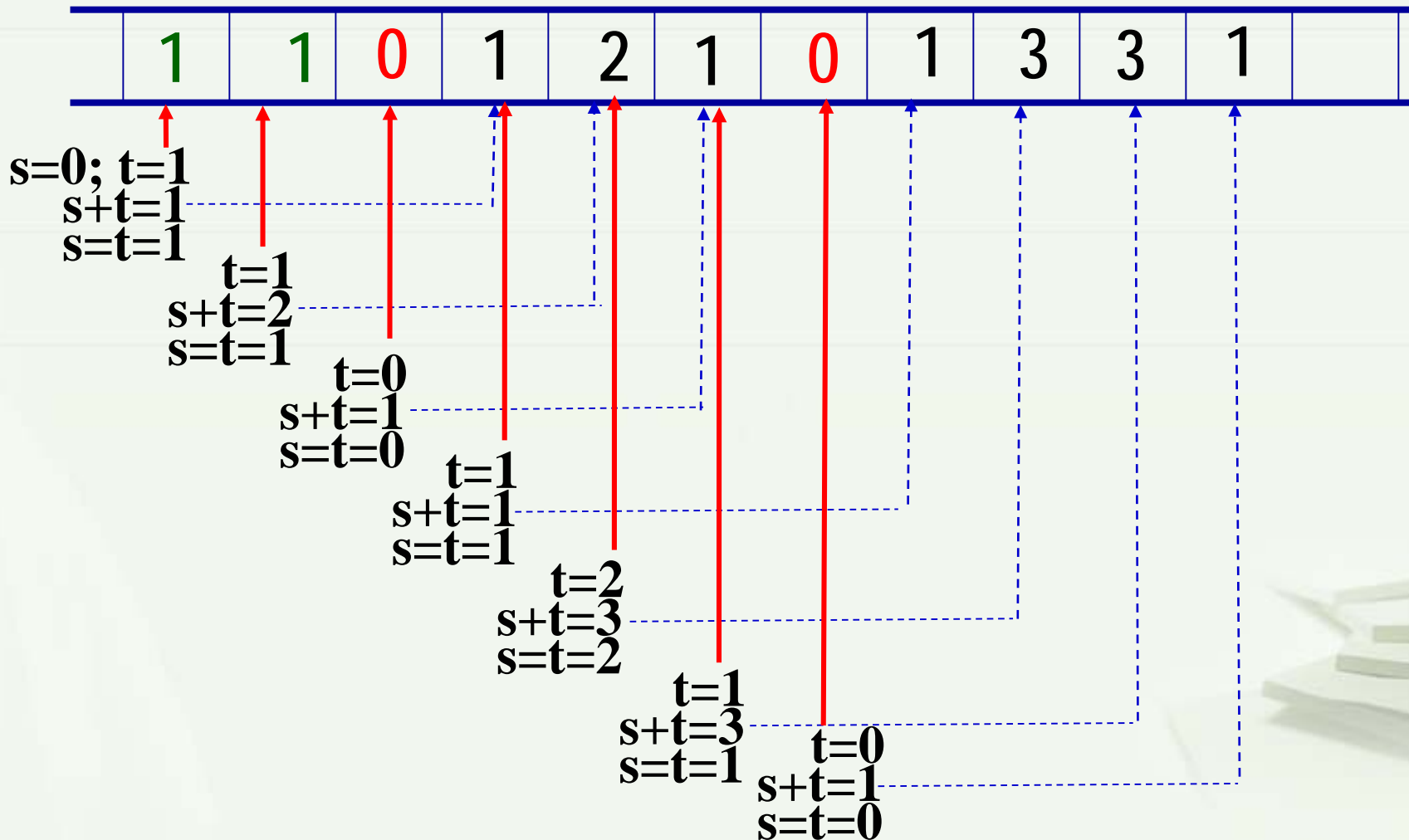


对于这种“逐层”求解、下层与上层有关系的问题，队列是一个重要的辅助工具。

3.2 队列

3.2.4 队列ADT的应用

队列应用举例之一：打印杨辉三角



3.2 队列

3.2.4 队列ADT的应用

队列应用举例之一： 打印杨辉三角

程序(见P121)：

```
#include <stdio.h>
#include <iostream.h>
#include "queue.h"
void YANGHUI (int n)
{   SeqQueue q(n+2);           //建立队列对象并初始化
    q.Enqueue(1);              //入队第一行
    q.Enqueue(1);
    int s = 0, t;               //s是记当前行的已经出对的系数
```

3.2 队列

3.2.4 队列ADT的应用

队列应用举例之一： 打印杨辉三角

```
for (int i = 1; i <= n; i++) //逐行计算并输出
{
    cout << endl;
    q.Enqueue(0); //每行开始加入一个0
    for (int j = 1; j <= i+2; j++) //得到下一行， 逐个入队
    {
        q.DeQueue(t); //当前行系数出队
        q.Enqueue(s + t); //计算出下行的系数， 入队
        s = t;
        if (j != i+2) //输出当前行的一个系数
            cout << s << ' ';
    }
}
```

3.2 队列

3.2.4 队列ADT的应用

队列应用举例之一： 打印杨辉三角

思考：

1. 如何按照规定的格式输出？
2. 用两个队列如何解决该问题？

3.2 队列

3.2.4 队列ADT的应用

队列应用举例之二： 服务大厅模拟

每个服务大厅是一种场景，现在要求利用计算机来模拟这种场景：

1. 上班时间到，开始营业，窗口队列初始化
下班时间到，停止营业，处理完队列里的业务，关门
上班、下班时间可以自己设定。
2. 服务窗口的个数可以自己定义
3. 顾客达到的时间随机产生，办理业务的时间随机产生
4. 可以定义VIP客户及其等级

3.2 队列

3.2.4 队列ADT的应用

队列应用举例之三： 广度遍历

在学习树结构、图结构时再介绍。

本章小结

重点和难点：

1. 栈是特殊线性数据结构，特殊在哪里？
 - 栈的特性
2. 栈ADT 的实现：注意栈顶、栈底，栈空、栈满
 - 顺序栈：溢出问题
 - 链式栈
3. 栈ADT 应用（重点）：利用栈的特点辅助我们解决问题
 - 表达式求值
 - 递归处理

本章小结

重点和难点：

4. 队列是特殊线性数据结构，特殊在哪里？

- 队列的特性
- 优先队列
- 双端队列

5. 队列ADT 的实现：注意队首、队尾，队列空、队列满

- 顺序队列：假溢出，循环存储的队列
- 链式队列

6. 队列的应用（重点）：利用队列的特点辅助我们解决问题

- 广度遍历



END