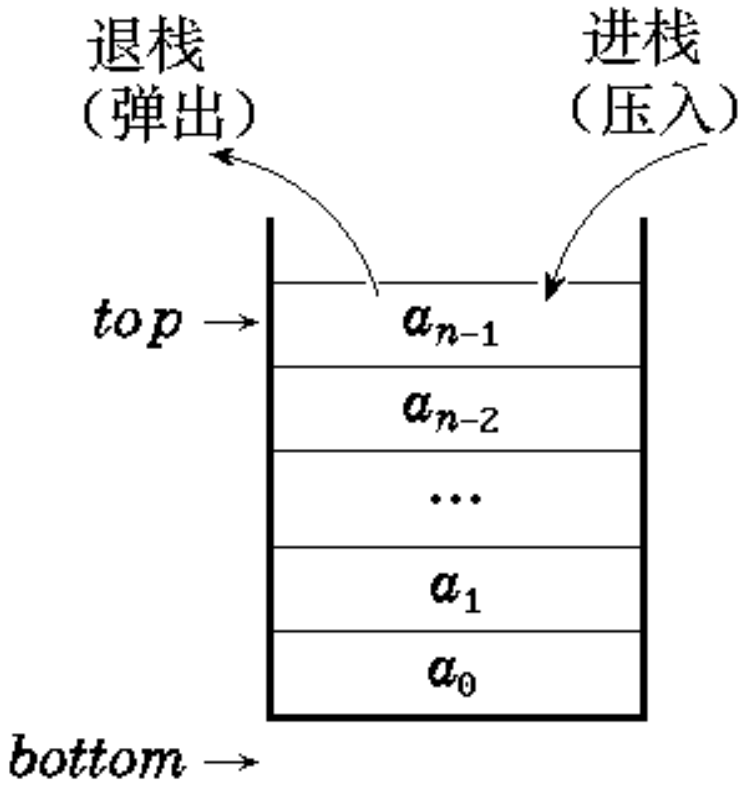
第四章 栈和队列

- 栈 (Stack)
- 队列 (Queue)
- 优先队列 (Priority Queue)
- 小结

栈（Stack）

- 只允许在一端插入和删除的顺序表
- 允许插入和删除的一端称为栈顶(*top*)，另一端称为栈底(*bottom*)
- 特点

  后进先出 (*LIFO*)



栈的抽象数据类型

```
template <class Type> class Stack {
public:
    Stack ( int=10 );                    //构造函数
    void Push ( const Type & item);      //进栈
    Type Pop ( );                        //出栈
    Type GetTop ( );                     //取栈顶元素
    void MakeEmpty ( );                  //置空栈
    int IsEmpty ( ) const;               //判栈空否
    int IsFull ( ) const;                //判栈满否
}
```

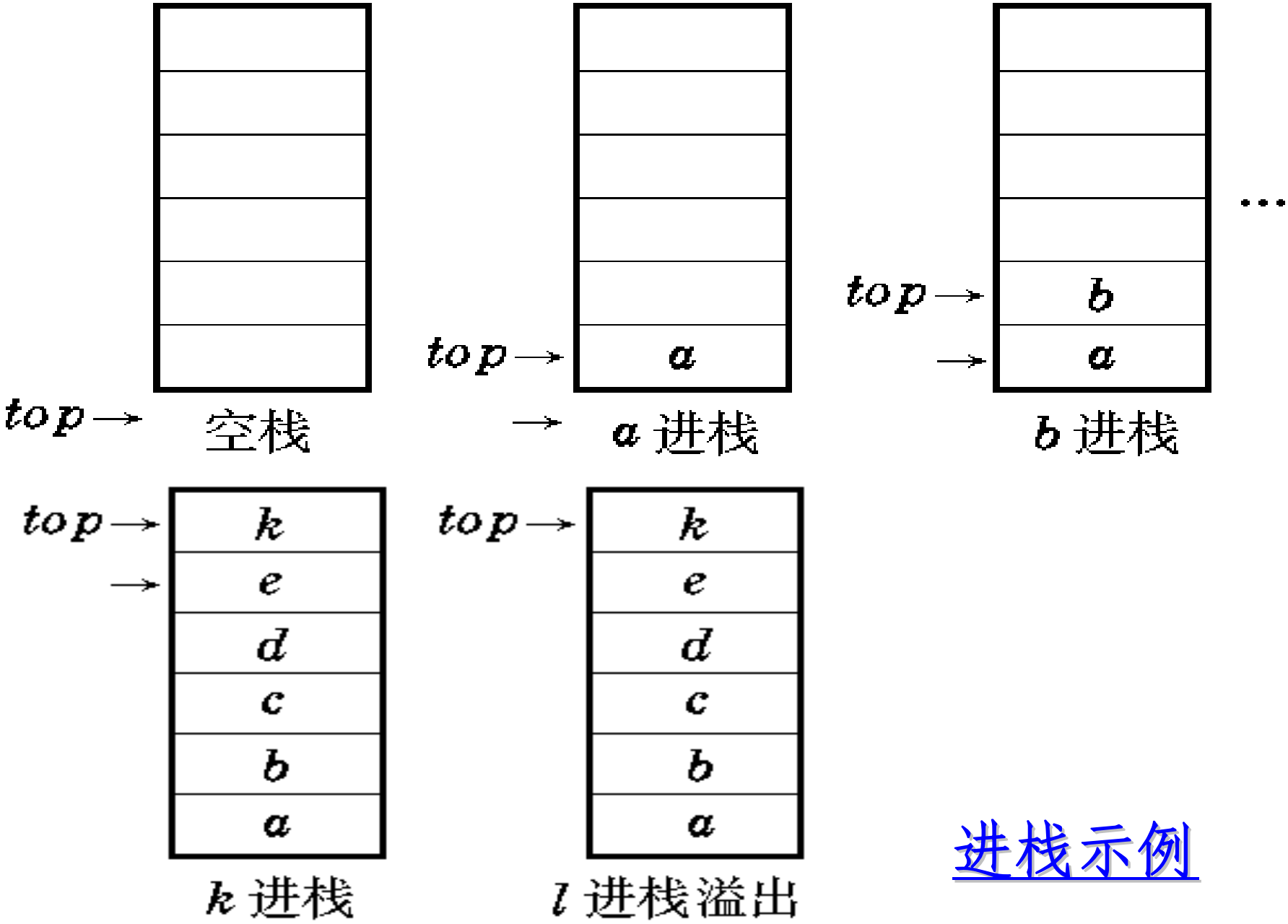
栈的数组表示 — 顺序栈

```
#include <assert.h>
template <class Type> class Stack {
public:
    Stack ( int=10 );                          //构造函数
    ~Stack ( ) { delete [ ] elements; }        //析构函数
    void Push ( const Type & item );           //进栈
    Type Pop ( );                              //出栈
    Type GetTop ( );                           //取栈顶
    void MakeEmpty ( ) { top = −1; }           //置空栈
    int IsEmpty ( ) const { return top == −1; }
```

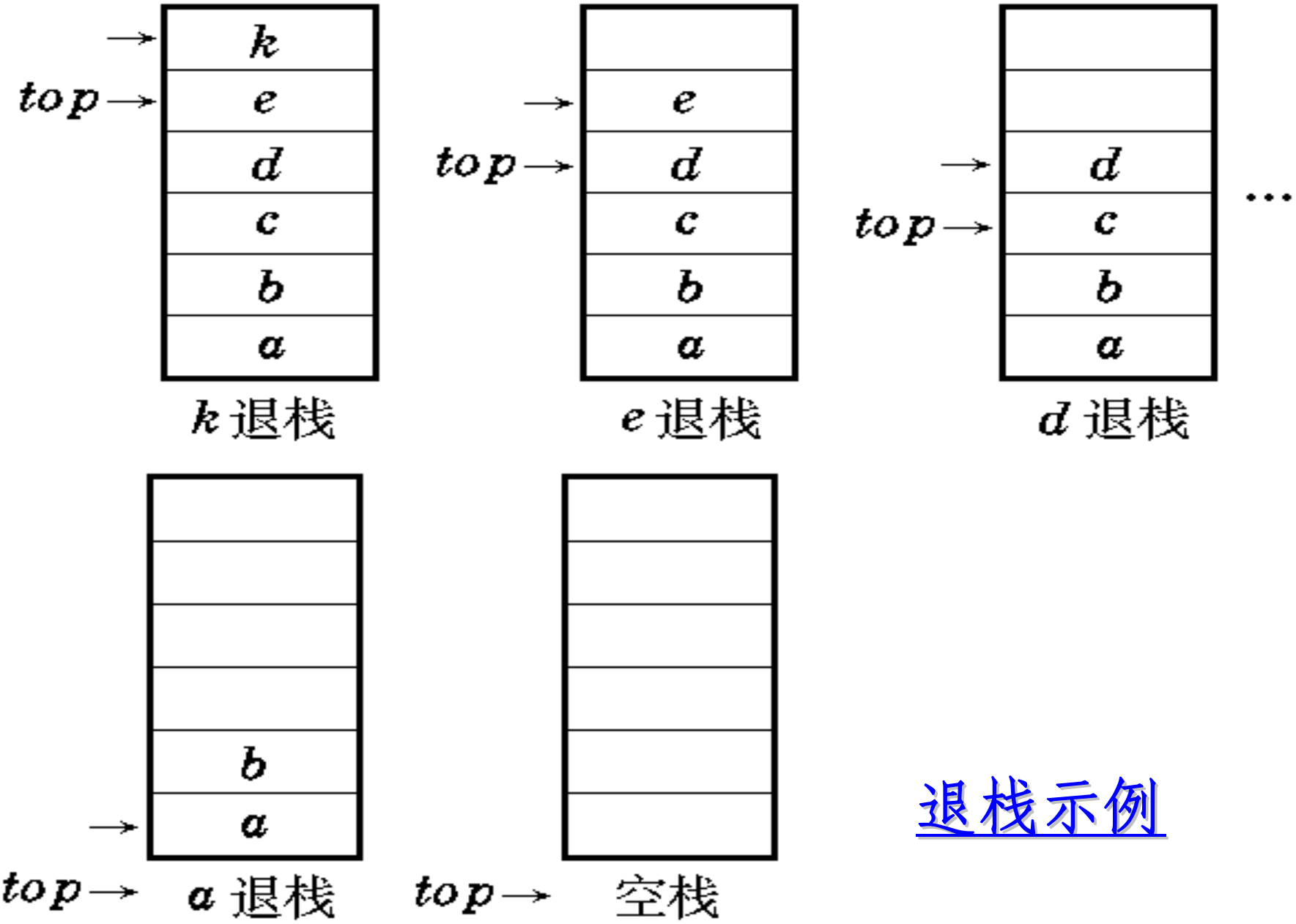
```
    int IsFull ( ) const
        { return top == maxSize−1; }
private:
    int top;                    //栈顶数组指针
    Type *elements;             //栈数组
    int maxSize;                //栈最大容量

}

template <class Type> Stack<Type>::
Stack ( int s ) : top (−1), maxSize (s) {
    elements = new Type[maxSize];
    assert ( elements != 0 );       //断言
}
```



进栈示例



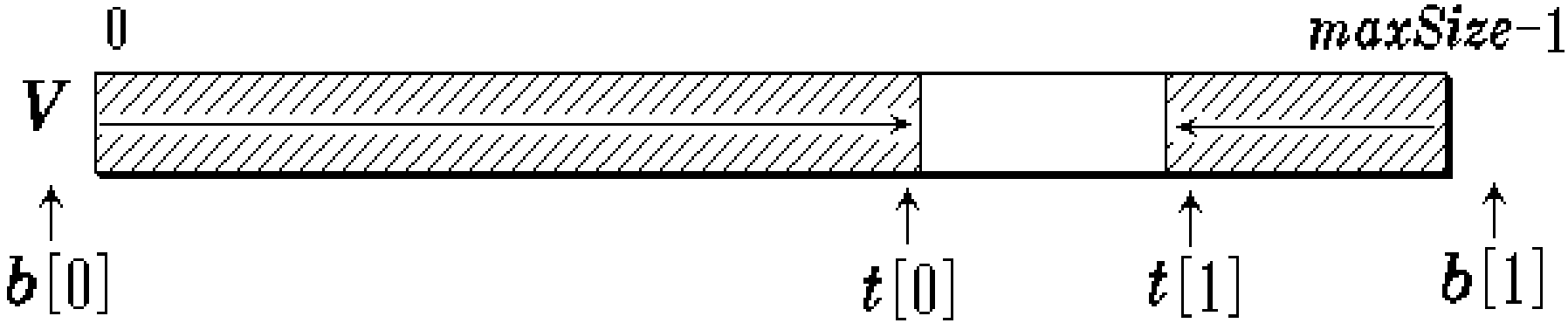
退栈示例

```
template <class Type> void Stack<Type>::
Push ( const Type & item ) {
    assert ( !IsFull ( ) );          //先决条件断言
    elements[++top] = item;          //加入新元素
}

template <class Type> Type Stack<Type>::
Pop ( ) {
    assert ( !IsEmpty ( ) );         //先决条件断言
    return elements[top--];          //退出栈顶元素
}
```

```
template <class Type> Type stack<Type>::
GetTop ( ) {
    assert ( !IsEmpty ( ) );        //先决条件断言
    return elements[top];           //取出栈顶元素
}
```

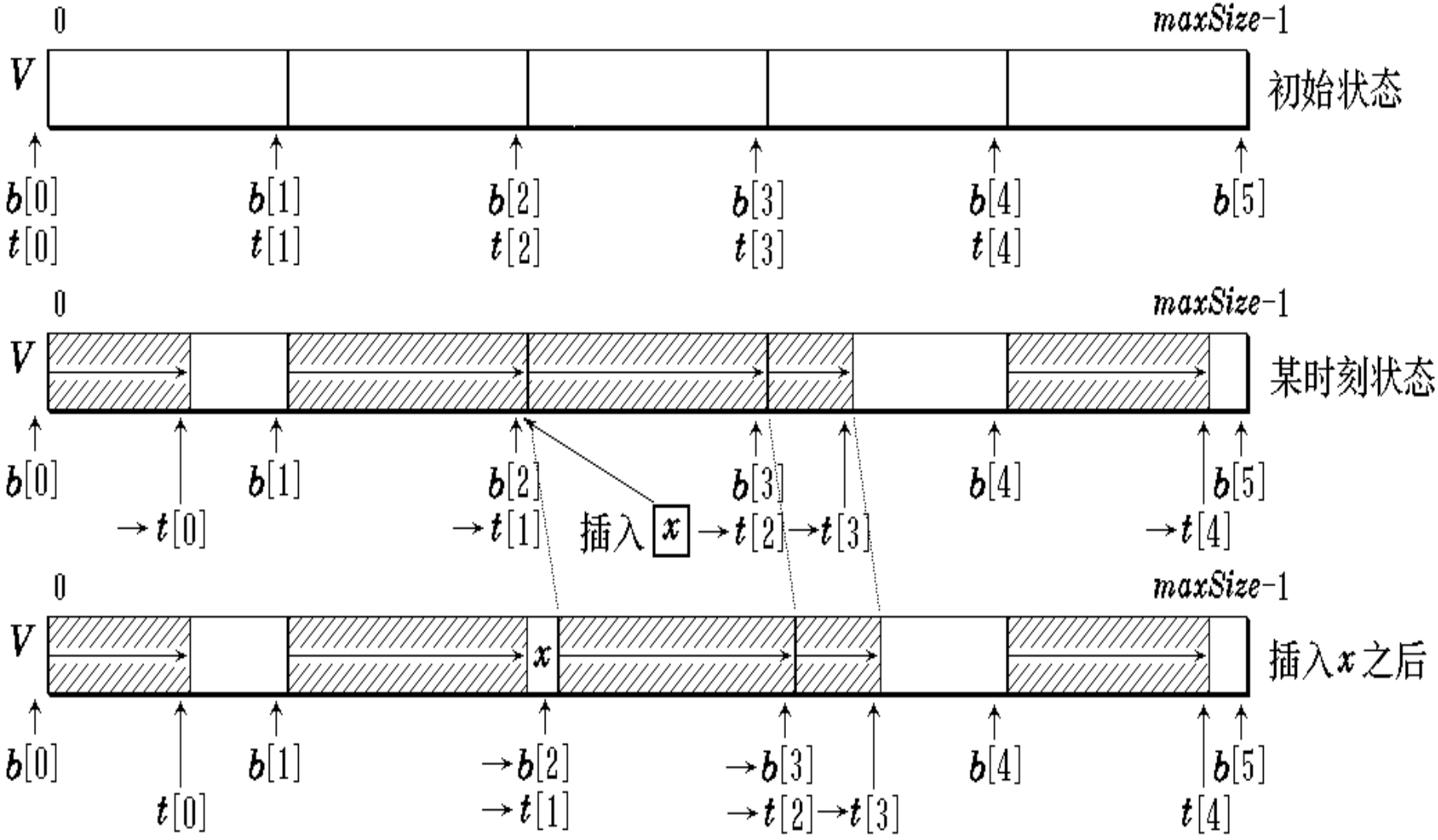
双栈共享一个栈空间



多栈处理 栈浮动技术

- n 个栈共享一个数组空间 $V[m]$
- 设立栈顶指针数组 $t[n+1]$ 和
  栈底指针数组 $b[n+1]$
- $t[i]$和$b[i]$分别指示第 i 个栈的栈顶与栈底
  $b[n]$作为控制量，指到数组最高下标
- 各栈初始分配空间 $s = \lfloor m / n \rfloor$
- 指针初始值 $t[0] = b[0] = -1 \quad b[n] = m-1$

  $t[i] = b[i] = b[i-1] + s, \quad i = 1, 2, \ldots, n-1$

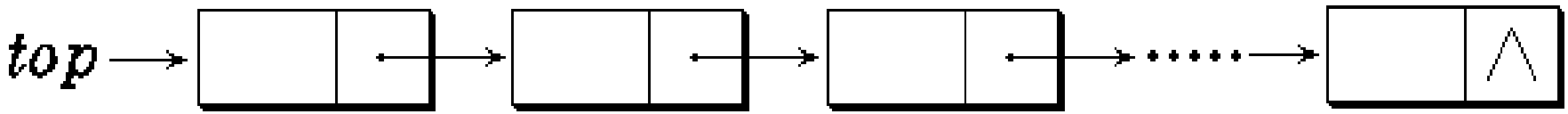


插入新元素时的栈满处理 *StackFull ( )*

```
template <class Type>
void Push ( const int i, const Type & item ) {
    if ( t [i] == b[i+1] ) StackFull (i);
    else V[++t[i]] = item;        //第i 个栈进栈
}

template <class Type> Type *Pop ( const i) {
    if ( t[i] == b[i] )
        { StackEmpty(i);  return 0; }
    else return & V[t[i]--];     //第i 个栈出栈
}
```

栈的链接表示 — 链式栈



- 链式栈无栈满问题，空间可扩充
- 插入与删除仅在栈顶处执行
- 链式栈的栈顶在链头
- 适合于多栈操作

链式栈 (*LinkedStack*)类的定义

```
template <class Type> class Stack;

template <class Type> class StackNode {
friend class Stack<Type>;
private:
    Type data;                          //结点数据
    StackNode<Type> *link;              //结点链指针
    StackNode ( Type d = 0, StackNode<Type>
        *l = NULL ) : data ( d ), link ( l ) { }
};
```

```
template <class Type> class Stack {
public:
    Stack ( ) : top ( NULL ) { }
    ~Stack ( );
    void Push ( const Type & item);
    Type Pop ( );
    Type GetTop ( );
    void MakeEmpty ( );          //实现与~Stack( )同
    int IsEmpty ( ) const
        { return top == NULL; }
private:
    StackNode<Type> *top;        //栈顶指针
}
```

```
template <class Type> Stack<Type>::
~Stack ( ) {
    StackNode<Type> *p;
    while ( top != NULL )      //逐结点回收
        { p = top;  top = top→link;  delete p; }
}

template <class Type> void Stack<Type>::
Push ( const Type &item ) {
    top = new StackNode<Type> ( item, top );
    //新结点链入top之前,并成为新栈顶
}
```

```
template <class Type> Type Stack<Type>::
Pop ( ) {
    assert ( !IsEmpty ( ) );
    StackNode<Type> *p = top;
    Type retvalue = p→data;      //暂存栈顶数据
    top = top→link;              //修改栈顶指针
    delete p;   return retvalue;  //释放,返回数据
}

template <class Type> Type Stack<Type>::
GetTop ( ) {
    assert ( !IsEmpty ( ) );
    return top→data;
}
```

队列 (Queue)

- 定义
  - 队列是只允许在一端删除，在另一端插入的顺序表
  - 允许删除的一端叫做队头(*front*)，允许插入的一端叫做队尾(*rear*)。
- 特性
  - 先进先出(*FIFO, First In First Out*)

队列的抽象数据类型

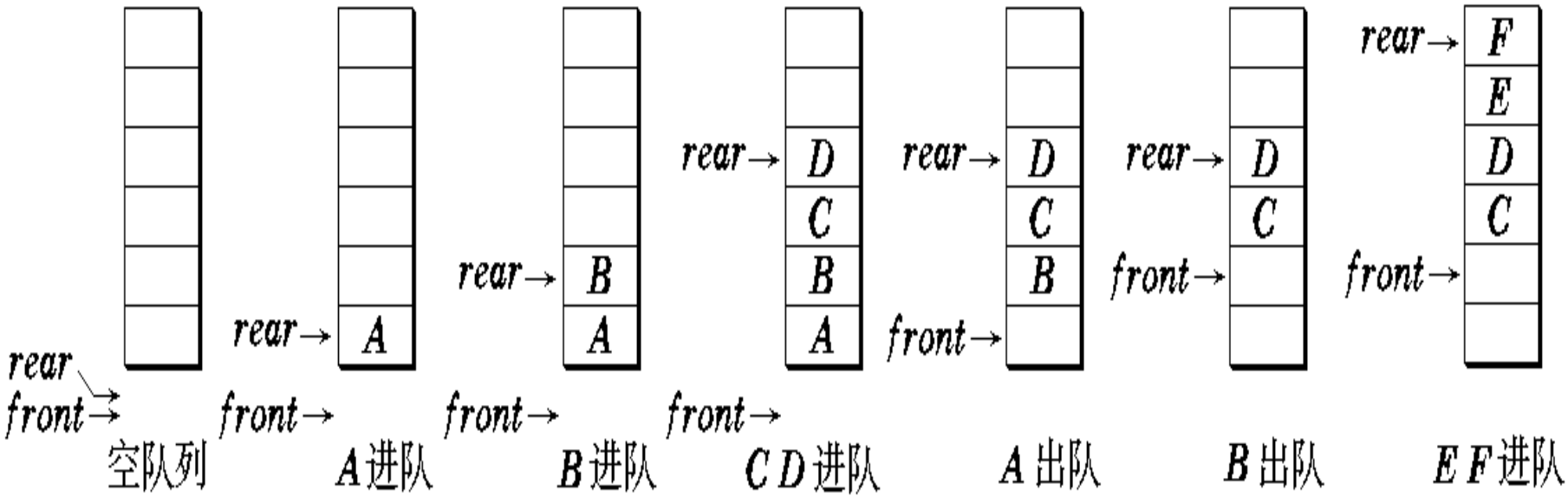
```
template <class Type> class Queue {
public:
    Queue ( int=10 );                  //构造函数
    void EnQueue ( const Type & item); //进队
    Type DeQueue ( );                  //出队列
    Type GetFront ( );                 //取队头元素值
    void MakeEmpty ( );                //置空队列
    int IsEmpty ( ) const ;            //判队列空否
    int IsFull ( ) const;              //判队列满否
}
```

队列的数组表示
——循环队列的类定义

```
#include <assert.h>
template <class Type> class Queue {
public:
    Queue ( int=10 );
    ~Queue ( ) { delete [ ] elements; }
    void EnQueue ( const Type & item);
    Type DeQueue ( );
    Type GetFront ( );
    void MakeEmpty ( ) { front = rear = 0; }
```

```
    int IsEmpty ( ) const
        { return front == rear; }
    int IsFull ( ) const
        { return (rear+1) % maxSize == front; }
    int Length ( ) const
        { return (rear−front+maxSize) % maxSize;}
private:
    int rear, front;
    Type *elements;
    int maxSize;
}
```

队列的进队和出队

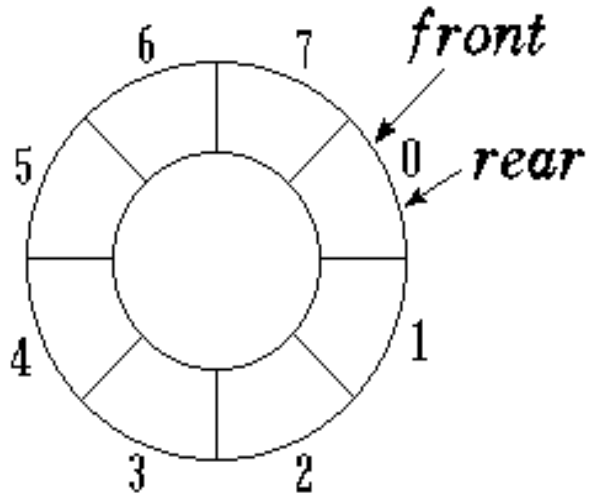


- 进队时队尾指针先进一 ***rear = rear + 1***，再将新元素按 ***rear*** 指示位置加入。
- 出队时队头指针先进一 ***front = front + 1***，再将下标为 ***front*** 的元素取出。
- 队满时再进队将溢出出错；队空时再出队将队空处理。

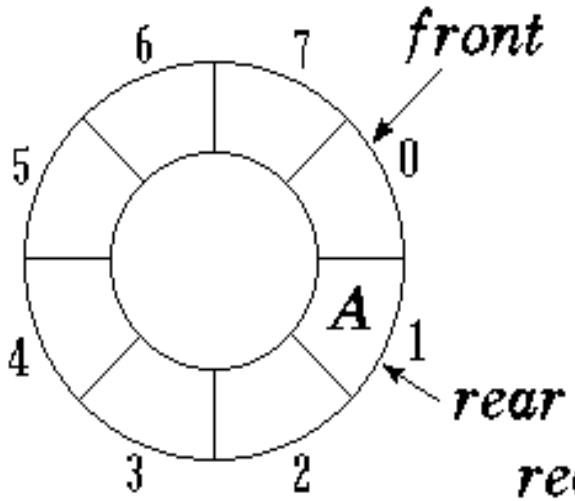
循环队列 (Circular Queue)

- 存储队列的数组被当作首尾相接的表处理。
- 队头、队尾指针加1时从*maxSize* -1直接进到0，可用语言的取模(余数)运算实现。
- 队头指针进1: *front* = (*front* + 1) % *maxSize*;
  队尾指针进1: *rear* = (*rear* + 1) % *maxSize*;
- 队列初始化: *front* = *rear* = 0;
- 队空条件: *front* == *rear*;
- 队满条件: (*rear* + 1) % *maxSize* == *front*

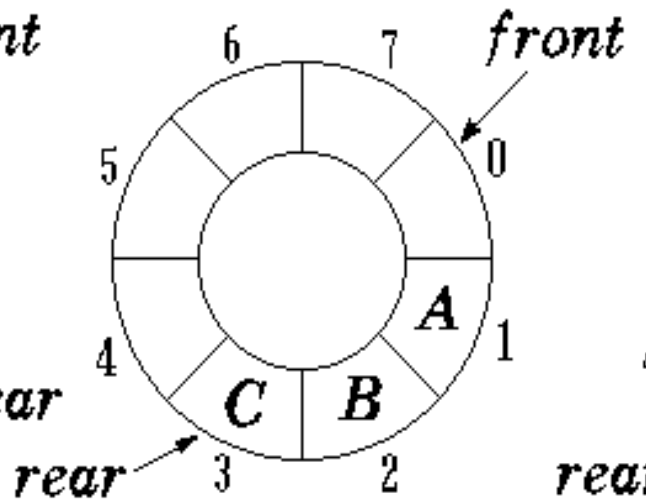
循环队列的进队和出队



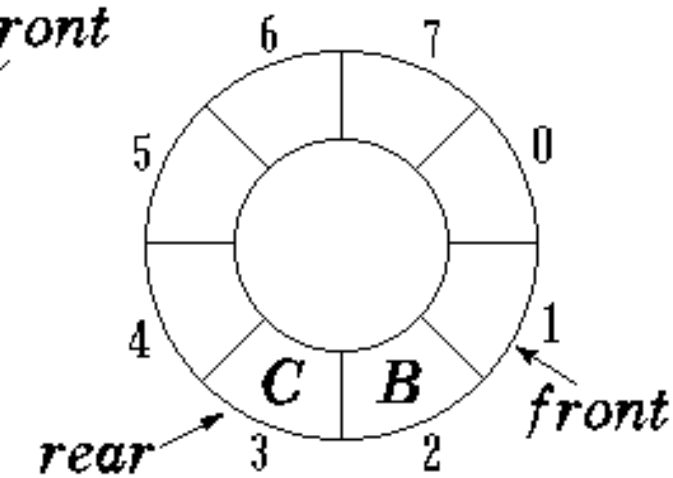
空队列



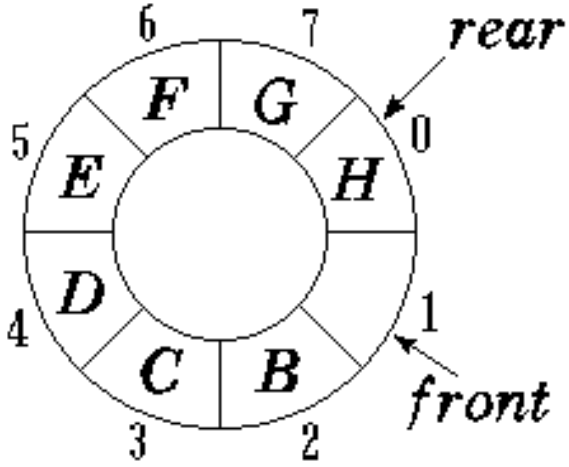
A 进队



B C 进队



A 出队



DEFGH 进队

循环队列部分成员函数的实现

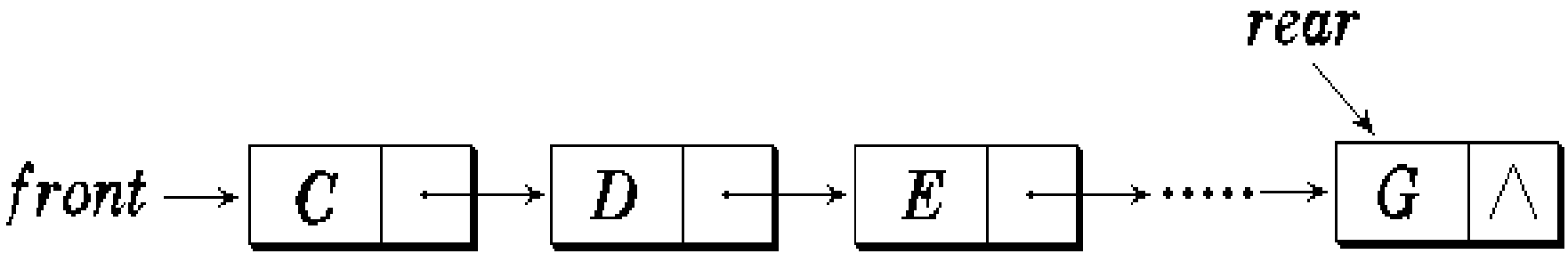
```
template <class Type> Queue<Type>::
Queue ( int sz ) : front (0), rear (0), maxSize (sz) {
    elements = new Type[maxSize];
    assert ( elements != 0 );
}

template <class Type> void Queue<Type>::
EnQueue ( const Type & item ) {
     assert ( !IsFull ( ) );
     rear = (rear+1) % MaxSize;
     elements[rear] = item;
}
```

```
template <class Type>
Type Queue<Type>::DeQueue ( ) {
    assert ( !IsEmpty ( ) );
    front = ( front+1) % MaxSize;
    return elements[front];
}

template <class Type>
Type Queue<Type>::GetFront ( ) {
    assert ( !IsEmpty ( ) );
    return elements[front];
}
```

队列的链接表示 — 链式队列



- 队头在链头，队尾在链尾。
- 链式队列在进队时无队满问题，但有队空问题。
- 队空条件为 ***front == NULL***

链式队列类的定义

```
template <class Type> class Queue;

template <class Type> class QueueNode {
friend class Queue<Type>;
private:
    Type data;                          //队列结点数据
    QueueNode<Type> *link;    //结点链指针
    QueueNode ( Type d=0, QueueNode
      *l=NULL ) : data (d), link (l) { }
};
```

```
template <class Type> class Queue {
public:
    Queue ( ) : rear ( NULL ), front ( NULL ) { }
    ~Queue ( );
    void EnQueue ( const Type & item );
    Type DeQueue ( );
    Type GetFront ( );
    void MakeEmpty ( );        //实现与~Queue( )同
    int IsEmpty ( ) const
                    { return front == NULL; }
private:
    QueueNode<Type> *front, *rear; //队列指针
};
```

链式队列成员函数的实现

```
template <class Type>
Queue<Type>::~Queue ( ) {
//队列的析构函数
    QueueNode<Type> *p;
    while ( front != NULL ) {     //逐个结点释放
        p = front;  front = front→link;  delete p;
    }
}
```

```
template <class Type> void Queue<Type>::
EnQueue ( const Type & item ) {
//将新元素item插入到队列的队尾
    if ( front == NULL )    //空, 创建第一个结点
        front = rear = new QueueNode
                        <Type> ( item, NULL );
    else                            //队列不空, 插入
        rear = rear→link = new QueueNode
                        <Type> ( item, NULL );
}
```

```
template <class Type> Type
Queue<Type>::DeQueue ( ) {
//删去队头结点，并返回队头元素的值
    assert ( !IsEmpty ( ) );          //判队空的断言
    QueueNode<Type> *p = front;
    Type retvalue = p→data;           //保存队头的值
    front = front→link;               //新队头
    delete p;
    return retvalue;
}
```

```
template <class Type>
Type Queue<Type>::GetFront ( ) {
//若队不空，则函数返回队头元素的值; 若
//队空，则函数返回0。
    assert ( !IsEmpty ( ) );
    return front→data;
}
```

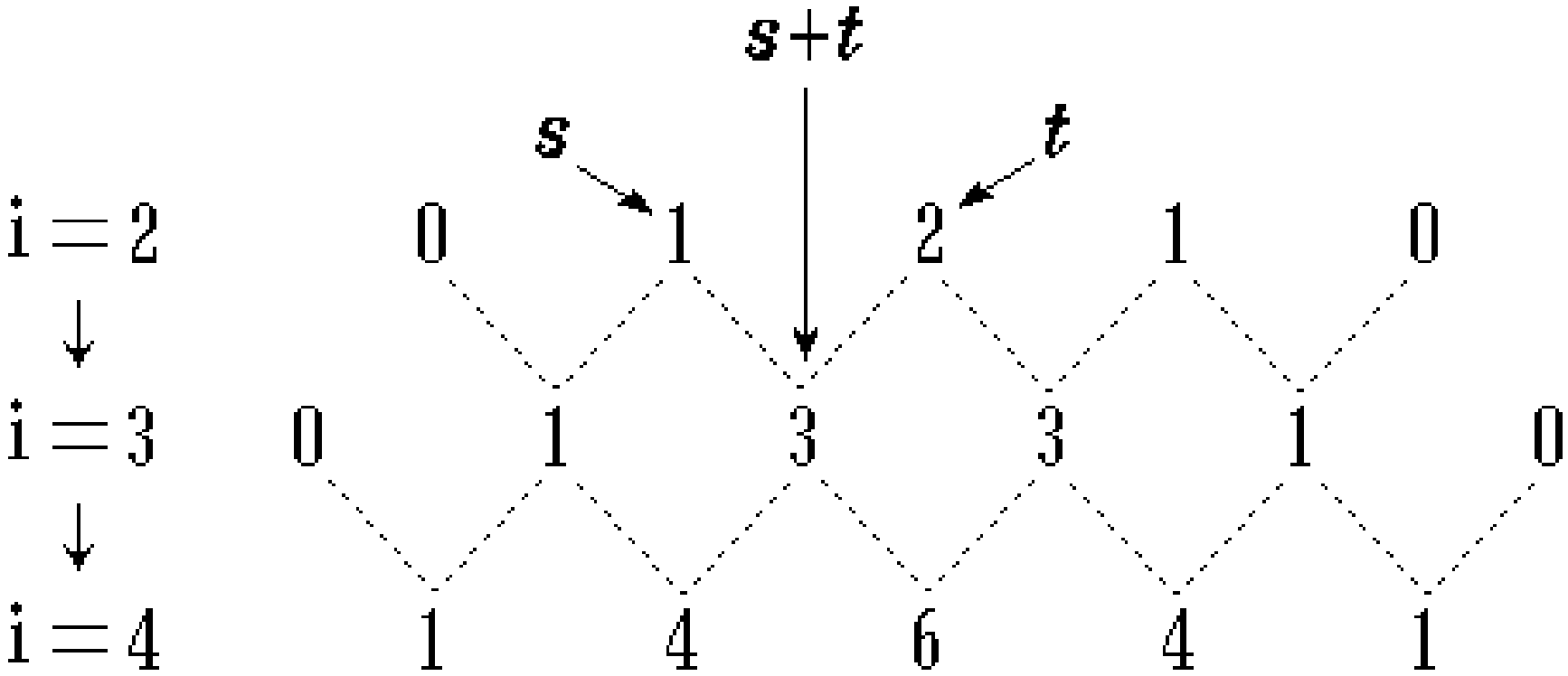
队列的应用举例

—逐行打印二项展开式 $(a+b)^i$ 的系数

杨辉三角形 (Pascal's triangle)

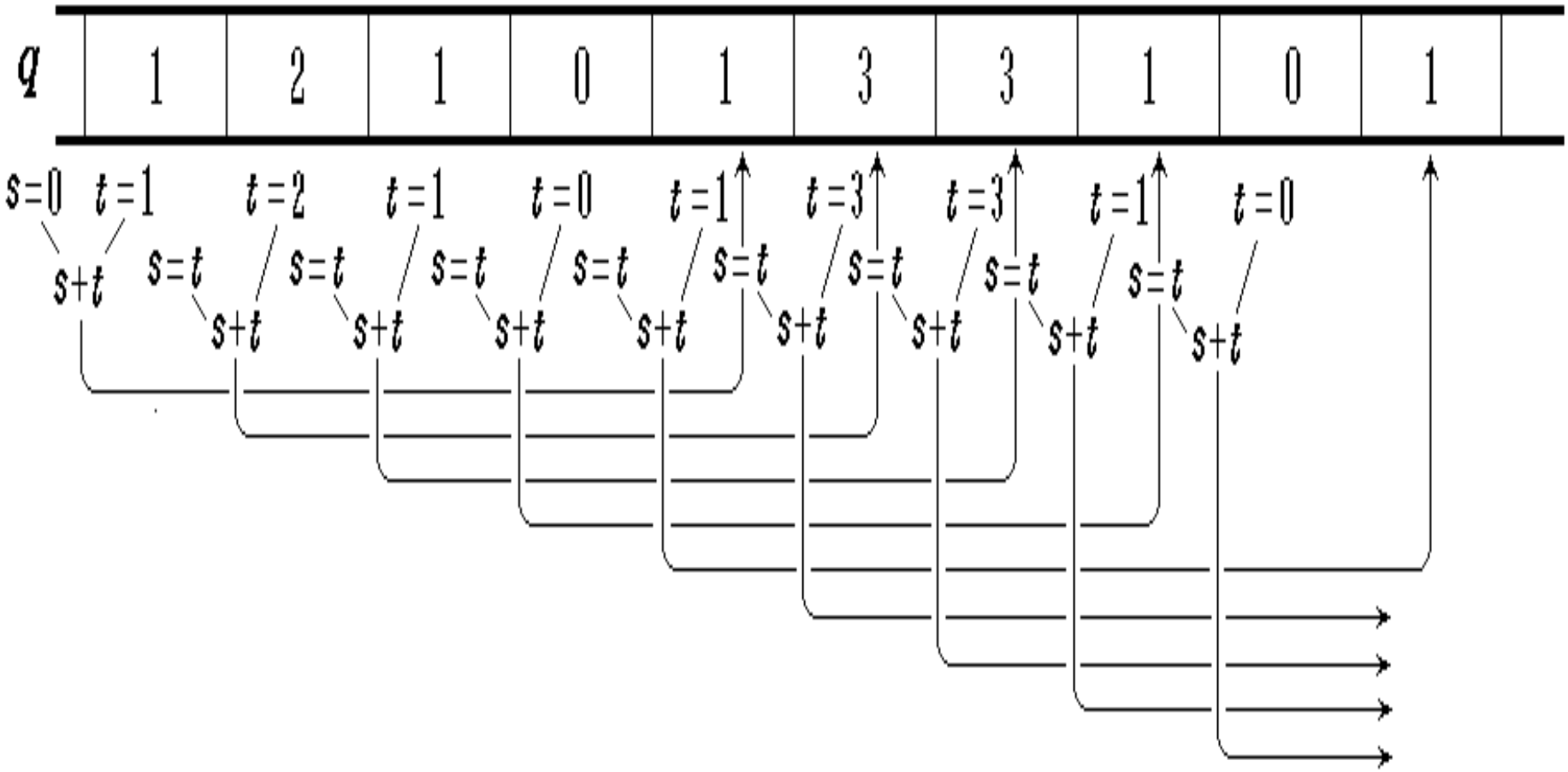
				1		1					$i=$	1
			1		2		1					2
		1		5		5		1				3
	1		4		6		4		1			4
1		5		10		10		5		1		5
	6		15		20		15		6		1	6

分析第 *i* 行元素与第 *i*+1行元素的关系



目的是从前一行的数据可以计算下一行的数据

从第 *i* 行数据计算并存放第 *i*+1 行数据



利用队列打印二项展开式系数的程序

```
#include <stdio.h>
#include <iostream.h>
#include "queue.h"
void YANGVI ( int n ) {
    Queue q;                                    //队列初始化
    q.MakeEmpty ( );
    q.EnQueue (1);  q.EnQueue (1);
    int s = 0;
```

```
    for ( int i=1; i<=n; i++ ) {          //逐行计算
        cout << endl;
        q.EnQueue (0);
        for ( int j=1; j<=i+2; j++ ) {  //下一行
            int t = q.DeQueue ( );

            q.EnQueue ( s+t );
            s = t;
            if ( j != i+2 ) cout << s << ' ';
        }
    }
}
```

优先级队列 (Priority Queue)

- 优先级队列 是不同于先进先出队列的另一种队列。每次从队列中取出的是具有最高优先权的元素
- 例如下表：任务的优先权及执行顺序的关系

任务编号	1	2	3	4	5
优先权	20	0	40	30	10
执行顺序	3	1	5	4	2

数字越小，优先权越高

优先队列的类定义

```
#include <assert.h>
#include <iostream.h>
$include <stdlib.h>
const int maxPQSize = 50;    //缺省元素个数

template <class Type> class PQueue {
public:
    PQueue ( );
    ~PQueue ( ) { delete [ ] pqelements; }
    void PQInsert ( const Type & item );
    Type PQRemove ( );
```

```
    void makeEmpty ( ) { count = 0; }
    int IsEmpty ( ) const
          { return count == 0; }
    int IsFull ( ) const
          { return count == maxPQSize; }
    int Length ( ) const { return count; }
private:
    Type *pqelements;          //存放数组
    int count;                 //队列元素计数
}
```

优先队列部分成员函数的实现

```
template <class Type>
PQueue<Type>::PQueue ( ) : count (0) {
    pqelements = new Type[maxPQSize];
    assert ( pqelements != 0 );      //分配断言
}

template <class Type> void PQueue<Type> ::
PQInsert ( const Type & item ) {
    assert ( !IsFull ( ) );              //判队满断言
    pqelements[count] = item;    count ++;
}
```

```
template <class Type>
Type PQueue<Type>::PQRemove ( ) {
   assert ( !IsEmpty ( ) );          //判队空断言
   Type min = pqelements[0];
   int minindex = 0;
   for ( int i=1; i<count; i++ )     //寻找最小元素
     if ( pqelements[i] < min )      //存于min
          { min = pqelements[i];   minindex = i; }
   pqelements[minindex] = pqelements[count−1];
   count−−;                          //删除
   return min;
}
```

小结 需要复习的知识点

- 栈
  - 栈的抽象数据类型
  - 栈的数组存储表示
  - 栈的链接存储表示
  - 栈的应用：用后缀表达式求值；中缀表达式向后缀表达式转换
- 队列
  - 队列的抽象数据类型

- 队列的顺序存储表示
- 队列的链接存储表示
- 队列的应用：打印杨辉三角形
- 用循环队列实现双端队列的插入与删除算法

- 优先级队列
  - 优先级队列的定义
  - 优先级队列的链接存储表示
  - 优先级队列的应用举例