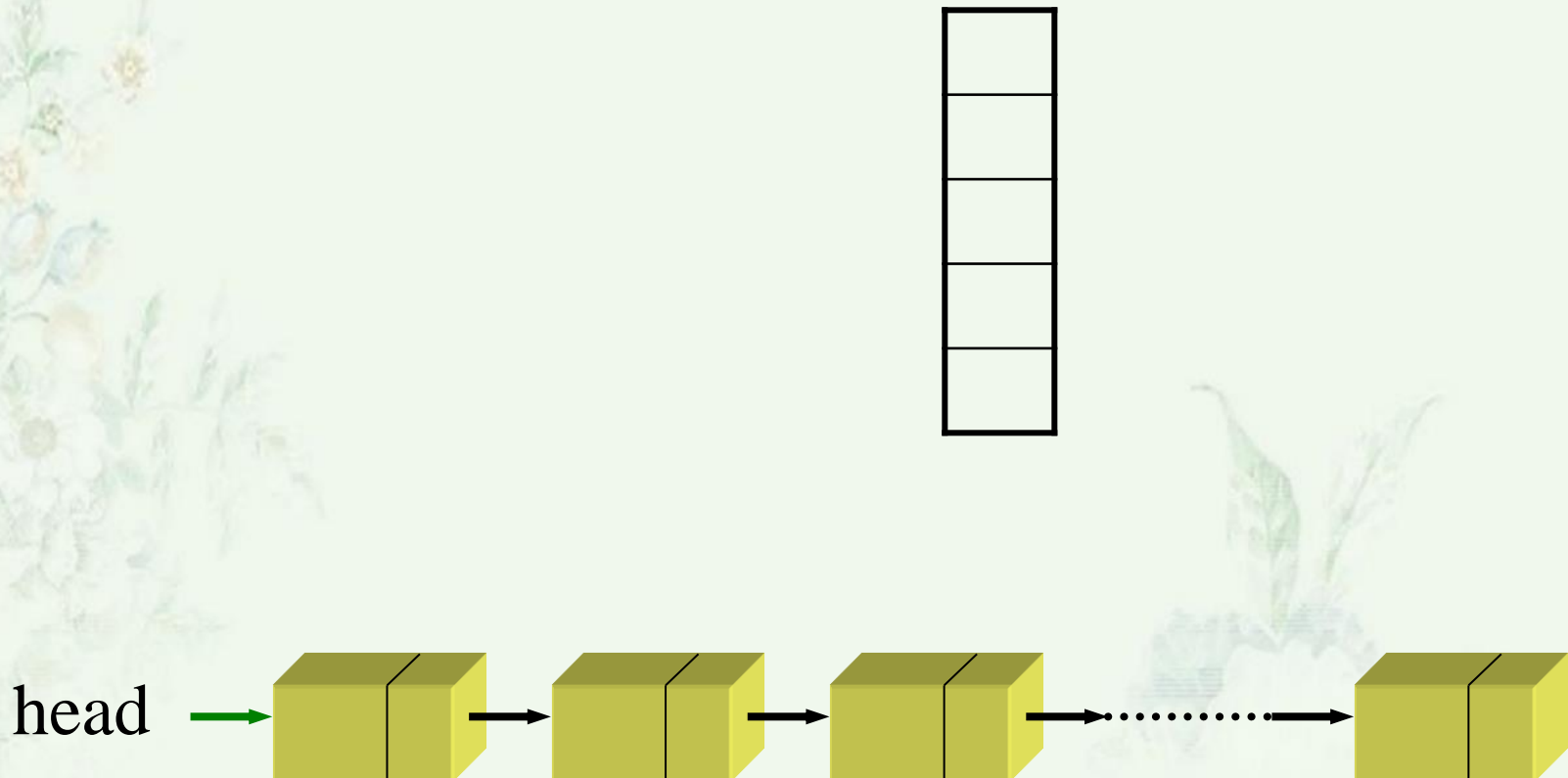


# 第三章 栈和队列

为什么要引入栈和队列？

# 栈在计算机中的实现



栈和队列是限定插入和删除只能在表的“端点”进行的线性表。

线性表

栈

队列

Insert(L, i, x)

Insert(S, n+1, x)

Insert(Q, n+1, x)

$1 \leq i \leq n+1$

Delete(L, i)

Delete(S, n)

Delete(Q, 1)

$1 \leq i \leq n$

可见,栈和队列的插入和删除算法的时间复杂度是 $O(1)$ 。

# 栈——逻辑结构

1. **定义**: 只允许在一端插入和删除的线性表

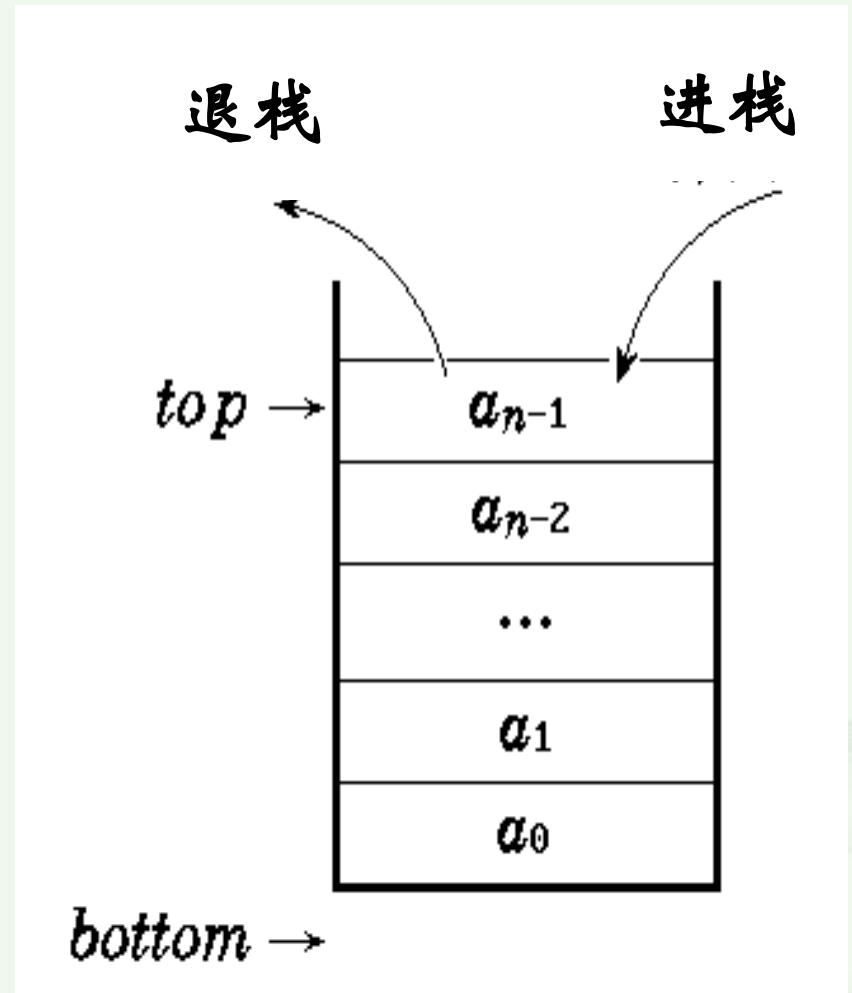
2. 栈顶(*top*):

允许插入和删除

3. 栈底(*bottom*):

4. 特点

后进先出 (*LIFO*)



# ADT stack

数据： 组成栈的一组数据元素

操作：

- (1) 构造函数 建立一个空栈；
- (2) 取栈顶元素 `gettop(x)` 若栈非空，则结果为栈顶元素，否则结果为空元素；
- (3) 进栈操作 `push(x)` 若栈未滿，则把元素 `x` 插入到栈的栈顶，否则产生出错；
- (4) 出栈操作 `pop(x)` 若栈不空，则结果为栈顶元素且删除栈元素，否则结果为空元素；
- (5) 判空栈 `isempty` 如果栈为空，则结果为“真”，否则为“假”。

end ADT stack

# 栈的抽象数据类型

```
template <class T>
```

```
class Stack {
```

**//栈的类定义**

```
public:
```

```
    Stack(){ };
```

**//构造函数**

```
    virtual void Push(T x) = 0;
```

**//进栈**

```
    virtual bool Pop(T& x) = 0;
```

**//出栈**

```
    virtual bool getTop(T& x) = 0;
```

**//取栈顶**

```
    virtual bool IsEmpty() = 0;
```

**//判栈空**

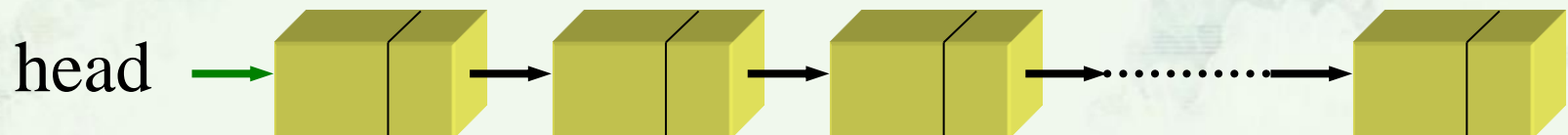
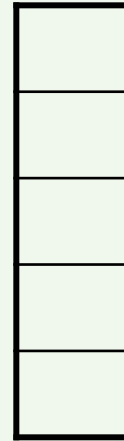
```
    virtual bool IsFull() = 0;
```

**//判栈满**

```
};
```

# 栈在计算机中的实现

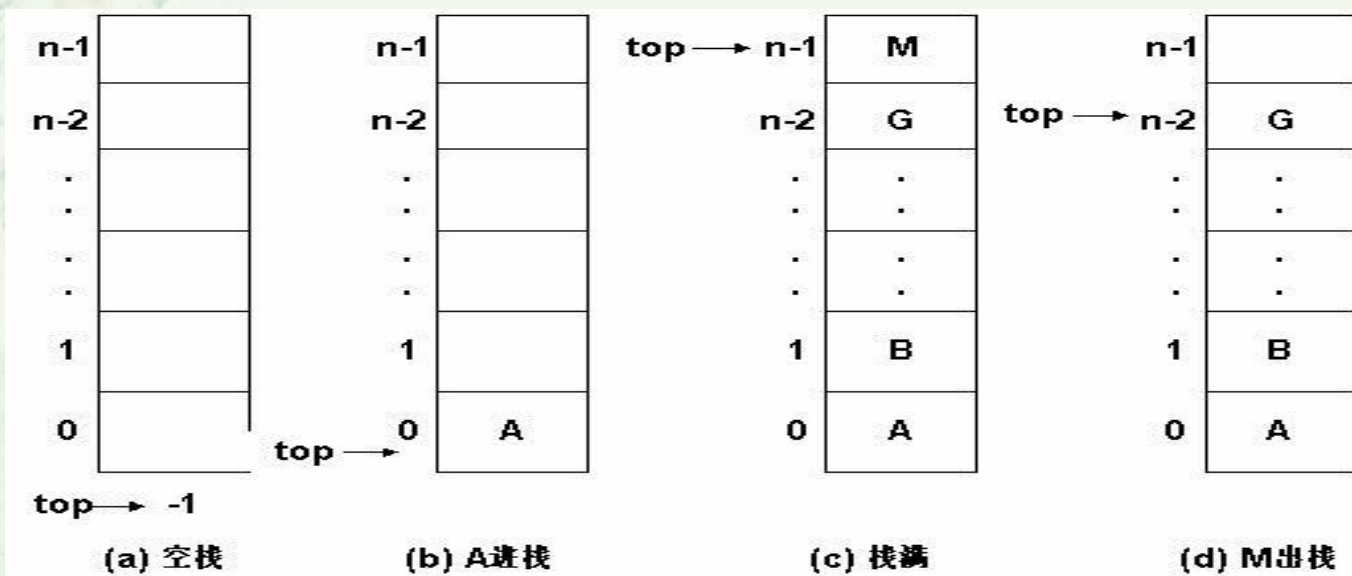
- 1. 存储结构
  - 顺序存储
  - 链式存储
- 2. 操作的实现



# 栈的顺序存储结构

顺序栈:指的就是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。

- 1.位置指示器 $top$ ,  $top=-1$ ,即为栈空;  $top=n-1$ ,即为栈满
- 2.“上溢”;
- 3.“下溢”。





```
class SeqStack : public Stack<T> { //顺序栈类定义  
private:
```

```
    T *elements;                //栈元素存放数组  
    int top;                     //栈顶指针  
    int maxSize;                //栈最大容量  
    void overflowProcess();      //栈的溢出处理
```

```
public:
```

```
    SeqStack(int sz =50);        //构造函数  
    ~SeqStack();                //析构函数  
    void Push(T x);              //进栈  
    bool Pop(T & x);             //出栈  
    bool getTop(T & x);          //取栈顶内容  
    bool IsEmpty() const { return top == -1; }  
    bool IsFull() const { return top == maxSize-1; }
```

```
};
```

**P89**

# 顺序栈类的构造函数与析构函数

```
template <class T>
SeqStack<T>::SeqStack(int sz):maxSize(sz)
{ top=-1;
  elements=new T[maxSize];
  assert(elements!=NULL); //判断是否分配成功
}
```

功能：按指定的长度分配顺序表空间，并设置maxSize及top变量初值

```
template <class T>
SeqStack<T>::~~SeqStack()
{ delete[] elements; }
```

功能：释放已分配的存储空间

# 顺序栈的操作

```
template <class T>
void SeqStack<T>::overflowProcess() {
//私有函数：当栈满则执行扩充栈存储空间处理
    E *newArray = new E[2*maxSize];
        //创建更大的存储数组
    for (int i = 0; i <= top; i++)
        newArray[i] = elements[i];
    maxSize += maxSize;
    delete [ ]elements;
    elements = newArray;    //改变elements指针
};
```

# 顺序栈的进栈操作

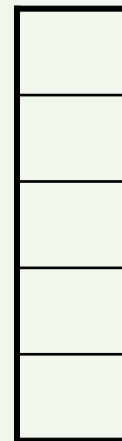
void Push (T x)

功能：在顺序栈中插入元素x，并使x成为栈顶。

处理过程：

(1)判是否栈满，若栈满则进行扩充空间，否则：

(2)栈顶指针加1，将x送入栈顶。



top=-1

```
template <class T>
```

```
void SeqStack<T>::Push(T x) {
```

```
//若栈不满,则将元素x插入该栈栈顶,否则溢出处理
```

```
    if (IsFull() == true) overflowProcess();    //栈满
```

```
    elements[++top] = x;        //栈顶指针先加1,再进栈
```

```
};
```

# 顺序栈的出栈操作

`bool Pop(T &x)`

功能：若指定的栈非空，则从栈中取出栈顶元素将其放入x中并返回真，否则返回假。

处理过程：

(1)判栈是否为空栈，若为空栈则返回假，否则：

(2)栈顶指针减1并返回原栈顶元素。

```
template <class T>
```

```
bool SeqStack<T>::Pop(T& x) {
```

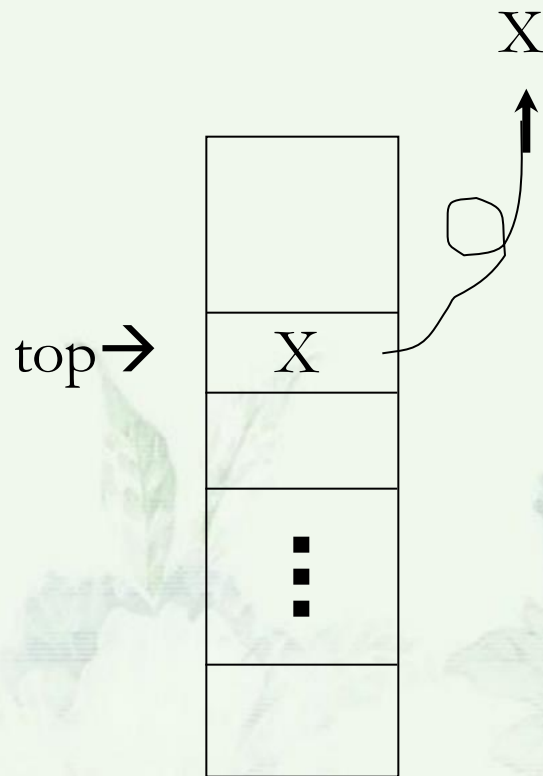
```
//函数退出栈顶元素并返回栈顶元素的值
```

```
    if (IsEmpty() == true) return false;
```

```
    x = elements[top--];           //栈顶指针退1
```

```
    return true;                  //退栈成功
```

```
};
```



## 顺序栈的取栈顶操作

`bool getTop(T &x)`

功能：若指定的栈非空，则将栈顶元素存入x中并返回真。

处理过程：判栈是否为空栈，若为空栈则返回假，否则复制原栈顶元素并返回真。

```
template <class T>
```

```
bool Seqstack<T>::getTop(T& x) {
```

```
//若栈不空则函数返回该栈栈顶元素的地址
```

```
    if (IsEmpty() == true) return false;
```

```
    x = elements[top];
```

```
    return true;
```

```
};
```

# 上述算法的时间复杂度

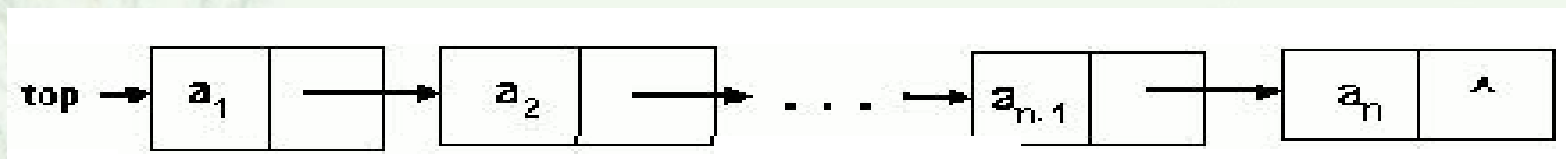
- 均为 $O(1)$



- 栈的链式存储结构

- 链栈:就是利用一个线性链表来存储栈中的数据元素。

- 下图就是链栈的一般形式。





# 链式栈 (LinkedStack) 类的定义

```
template <class T>
struct StackNode {                                //栈结点类定义
private:
    T data;                                       //栈结点数据
    StackNode<T> *link;                         //结点链指针
public:
    StackNode(T d = 0, StackNode<T> *next =
        NULL)
        : data(d), link(next) { }
};
```

```
template <class T>
class LinkedStack : public Stack<T> { //链式栈类定义
private:
    StackNode<T> *top;                //栈顶指针
    void output(ostream& os,
        StackNode <T> *ptr, int i);
                                    //递归输出栈的所有元素
public:
    LinkedStack() : top(NULL) {}      //构造函数
    ~LinkedStack() { makeEmpty(); } //析构函数
    void Push(T x);                  //进栈
    bool Pop(T & x);                 //退栈
};
```

```
bool getTop(T & x) const;           //取栈顶
bool IsEmpty() const { return top == NULL; }
void makeEmpty();                  //清空栈的内容
friend ostream& operator << (ostream& os,
    LinkedStack<T>& s) { output(os, s.top, 1); }
    //输出栈元素的重载操作 <<
};
```

# 链式栈类操作的实现

## 清空操作

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

# 进栈操作

```
void push(T el)
```

功能：在当前链栈中插入元素x，使x成为栈顶元素。处理过程：

(1)生成一个新的结点，并令其元素值为x。

(2)将该结点从栈顶插入到链栈中。

```
template <class T>
```

```
void LinkedStack<T>::Push(T x) {
```

```
//将元素值x插入到链式栈的栈顶,即链头。
```

```
    top = new StackNode<T> (x, top);           //创建新结点
```

```
    assert (top != NULL);                       //创建失败退出
```

```
};
```



# 出栈操作

bool pop(T &x)

功能：从当前链栈中弹出栈顶结点，将其元素值放入x中并返回真  
处理过程：

- (1)判链栈是否为空，若为空栈则返回假。
- (2)保存栈顶结点的元素值于x，并修改栈顶指针使之指向其后继。
- (3)置返回函数值为真。

template <class T>

bool **LinkedStack<T>::Pop(T & 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<T> \*p = top; //暂存栈顶元素**

**top = top->link; //退栈顶指针**

**x = p->data; delete p; //释放结点**

**return true;**

**};**



# 取栈顶元素操作

**bool getTop(T& x)**

功能：从当前链栈的栈顶结点元素值放入x中并返回真

处理过程：

- (1)判链栈是否为空，若为空栈则返回假。
- (2)保存栈顶结点的元素值于x。
- (3)置返回函数值为真。

**template <class T>**

**bool LinkedStack<T>::getTop(T& x) const {**

**if (IsEmpty() == true) return false; //栈空返回**

**x = top->data; //返回栈顶元素的值**

**return true;**

**};**



# 进栈和出栈算法的时间复杂度

均为 $O(1)$



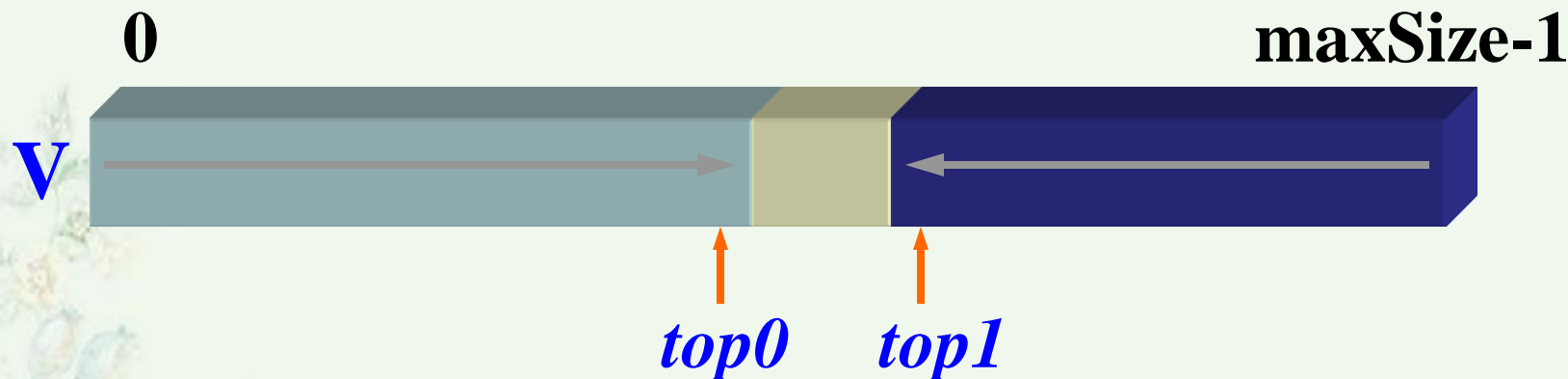
# 顺序栈和链栈的比较

- 1.顺序存储结构与链式结构的比较.
- 2.采用链式存储方式则可以利用操作系统来完成实现多个栈空间的共享。
- 3.使用一个数组来存储两个栈

# 问题

一个问题求解过程中同时用到两个数据类型一样的栈？

# 双栈共享一个栈空间

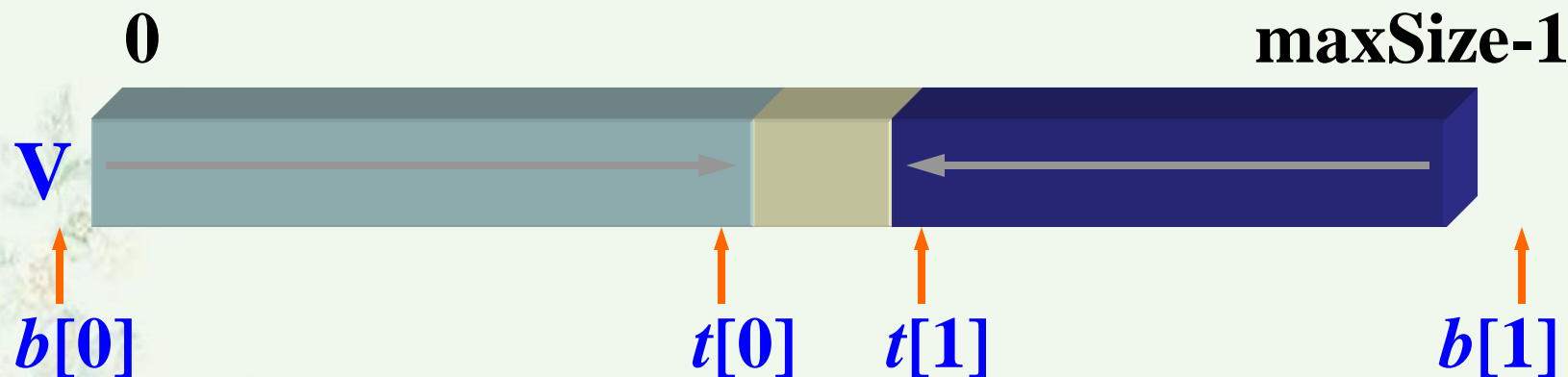


- 两个栈共享一个数组空间  $V[\text{maxSize}]$
- 设立栈顶指针数组  $\text{top0}$  和  $\text{top1}$
- 初始  $\text{top0} = -1$ ,  $\text{top1} = \text{maxSize}$
- 栈满条件:  $\text{top0}+1 == \text{top1}$  //栈顶指针相遇
- 栈空条件:  $\text{top0} = -1$ 或 $\text{top1} = \text{maxSize}$  //退到栈底

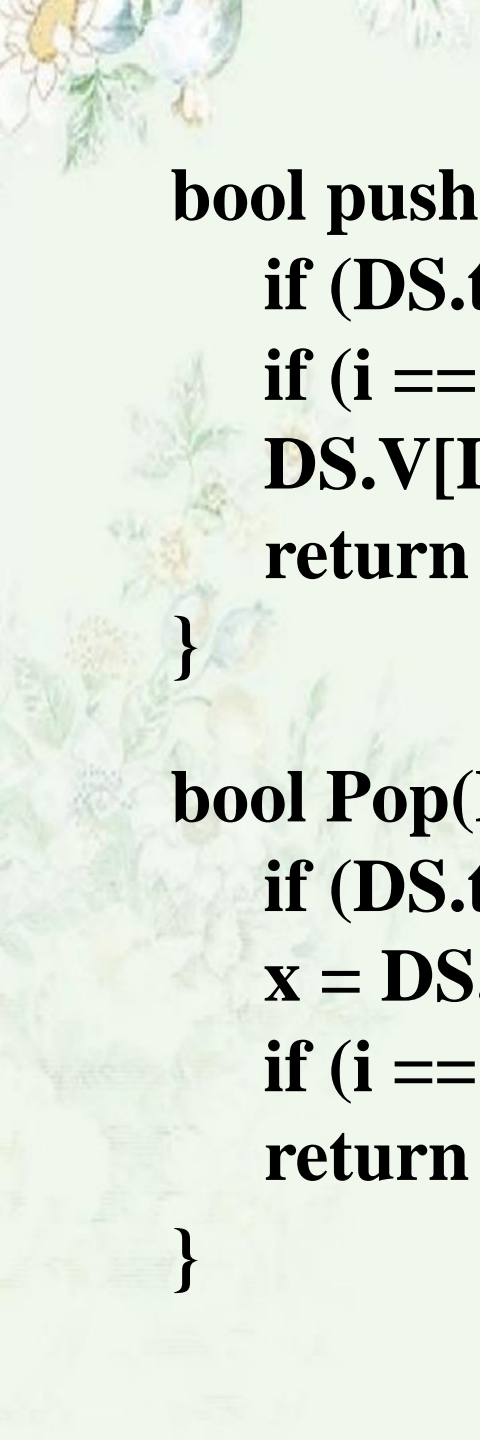
```
bool push(DualStack& DS, Type x, int i)  
{ if (DS.top0+1 == DS.top1) return false;  
  if (i == 0) { DS.top0++; DS.V[DS.top0] = x; }  
  else { DS.top1--; DS.V[DS.top1] = x; }  
  return true;  
}
```

```
bool Pop(DualStack& DS, Type& x, int i)  
{ if(i==0) {  
  if (DS.top0 == -1) return false;  
  x = DS.V[DS.top0]; DS.top0--; }  
  else {  
    if (DS.top1 == maxsize) return false;  
    x = DS.V[DS.top1]; DS.top1++; }  
  return true;  
}
```

# 双栈共享一个栈空间



- 两个栈共享一个数组空间  $V[\text{maxSize}]$
- 设立栈顶指针数组  $t[2]$  和栈底指针数组  $b[2]$   
 $t[i]$  和  $b[i]$  分别指示第  $i$  个栈的栈顶与栈底
- 初始  $t[0] = b[0] = -1$ ,  $t[1] = b[1] = \text{maxSize}$
- 栈满条件:  $t[0] + 1 == t[1]$  // 栈顶指针相遇
- 栈空条件:  $t[0] = b[0]$  或  $t[1] = b[1]$  // 退到栈底



```
bool push(DualStack& DS, Type x, int i) {  
    if (DS.t[0]+1 == DS.t[1]) return false;  
    if (i == 0) DS.t[0]++; else DS.t[1]--;  
    DS.V[DS.t[i]] = x;  
    return true;  
}
```

**P92**

```
bool Pop(DualStack& DS, Type& x, int i) {  
    if (DS.t[i] == DS.b[i]) return false;  
    x = DS.V[DS.t[i]];  
    if (i == 0) DS.t[0]--; else DS.t[1]++;  
    return true;  
}
```

## • 栈的应用

- 1.只要实际问题符合先进后出特点.
- 2.栈在计算机科学领域具有广泛的应用。
  - 例如，在编译和程序执行过程中，就需要利用栈进行语法检查、计算表达式的值、实现函数或过程的嵌套调用或递归调用。
- 3.在一般问题的算法设计中栈的应用也十分广泛。
  - 特别是在穷举（又称搜索或遍历）满足问题部分要求求解的过程中往往需要记录某一阶段穷举的特例，以便无法找到所要结果时能退回去（通常称为回溯）重新继续穷举。

# 1 数制转换

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

$$N = (n / d) * d + n \% d$$

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

|   | n    | n / 8 | n % 8 |
|---|------|-------|-------|
| ↓ | 1348 | 168   | 4     |
|   | 168  | 21    | 0     |
|   | 21   | 2     | 5     |
|   | 2    | 0     | 2     |
|   |      |       | ↑     |



```
void conversion( )
{
    seqstack s;
    cin>>n;
    while(n)
    { s.push(n%8);
      n=n/8;
    }
    while(! S.IsEmpty())
    { s.pop( e );
      cout<<e;
    }
}
```

## 2 括号匹配的检验

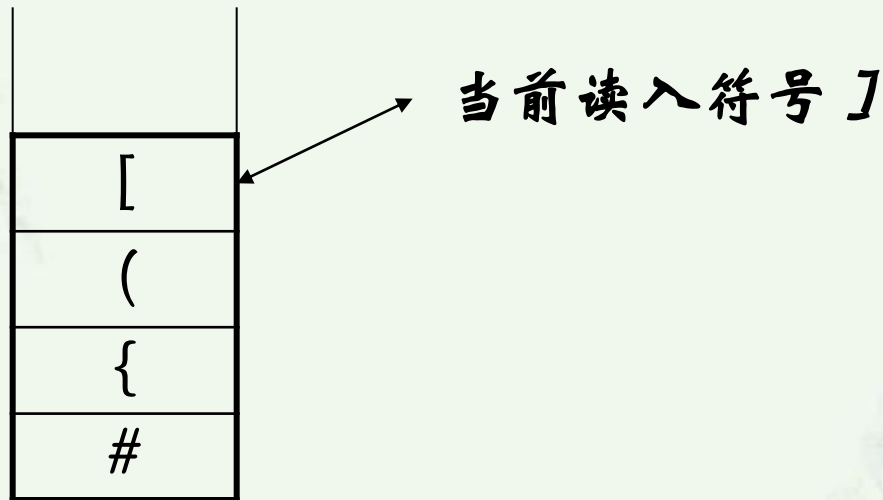
假设表达式中允许括号嵌套,则检验括号是否匹配的方法可用“期待的急迫程度”这个概念来描述。例:

$$\{ ([ \dots ] \dots ) \dots \{ ( \dots ) \dots \} \}$$

# 括号配对检查程序

- 执行过程中栈的变化

# { ( [ ..... ] ... ) ... { ( ..... ) ..... } } #



### 3. 表达式的计算

P95-P101

## 4.函数的嵌套调用和递归调用

在程序设计中函数或过程嵌套调用和逐层返回的处理也是栈的应用的一个典型例子。

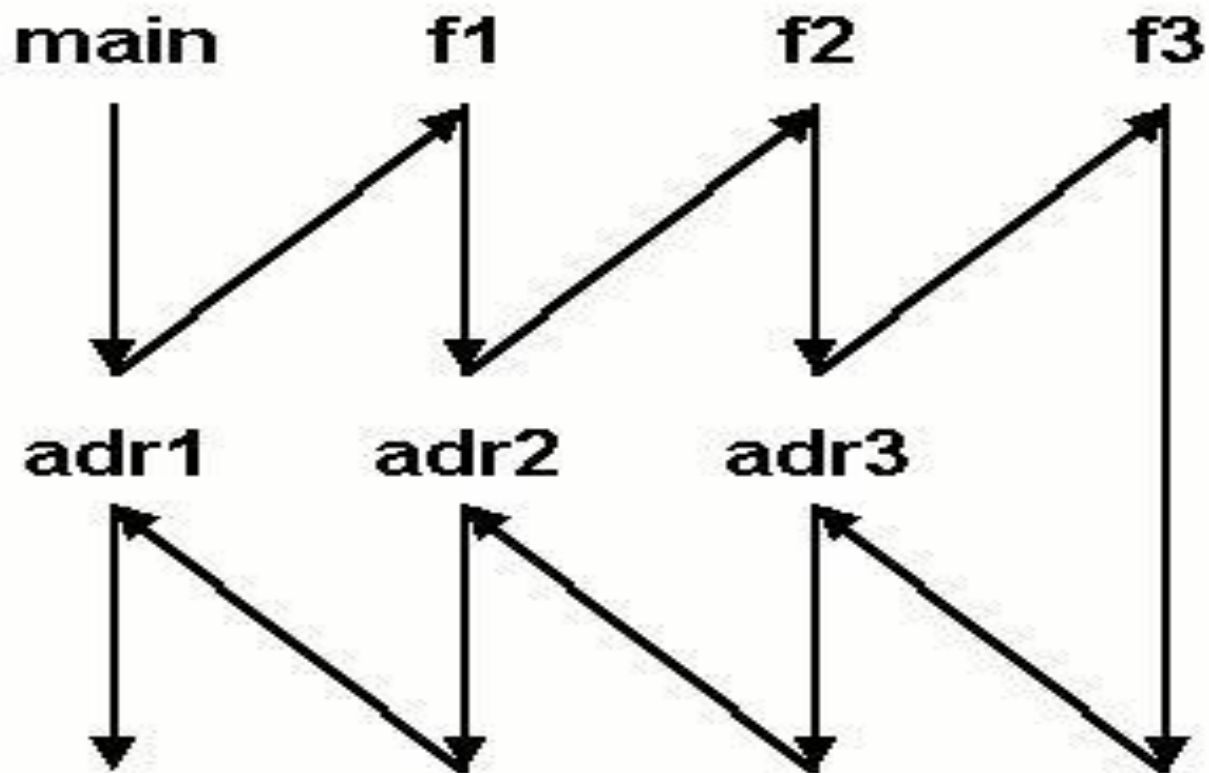
所谓函数的嵌套调用是指在调用一个函数的过程中又调用另一个函数。

例如，在一个C/C++程序中有如下嵌套调用的主函数main()和三个函数f1(),f2(),f3():

| main( ) | f1( )  | f2( )  | f3( )  |
|---------|--------|--------|--------|
| {       | {      | {      | {      |
| ...     | ...    | ...    | ...    |
| f1( );  | f2( ); | f3( ); | ...    |
| adr1→   | adr2→  | adr3→  | ...    |
| ...     | ...    | ...    | ...    |
| ...     | return | return | return |
| }       | }      | }      | }      |

其中: adr1, adr2, adr3 分别表示返回地址

- 程序被执行情况如下图所示



## 递归函数的执行

```
int f( int n)
{
    if (n==1) return 1;
    else return n*f(n-1);
}
```



# 递归函数的编制

- 在对问题进行分解、求解的过程中得到的是和原问题性质相同的子问题时，都可以设计出求解该问题的递归算法。
- 算法逻辑性强、结构清晰、且算法的正确性易于证明，因此递归是程序设计中一个非常重要的方法。
- 递归算法一般由基本项和归纳项两部分组成。
  - 基本项描述了递归过程的终止状态，即不需要继续递归而可直接求解的状态；
  - 归纳项则描述了如何将一个复杂问题，分解成具有相同性质的若干子问题，它们解决了，原问题也就解决了。


例1, 求n的阶乘 $n!$  ( $n > 0$ )

终止状态: 当 $n=1$ 时,  $n!=1$

归纳项: 当 $n > 1$ 时,  $n!=n*(n-1)!$


因此, 其递归函数可描述如下:

```
int fact (int n)
{
    if (n==1) return 1; // 终止状态
    else return n*fact(n-1); // 归纳项
} // fact
```



```
int fact ( int n )  
{  
    if ( n==1 ) return 1; // 终止状态  
    else return n*fact(n-1); // 归纳项  
} // fact
```

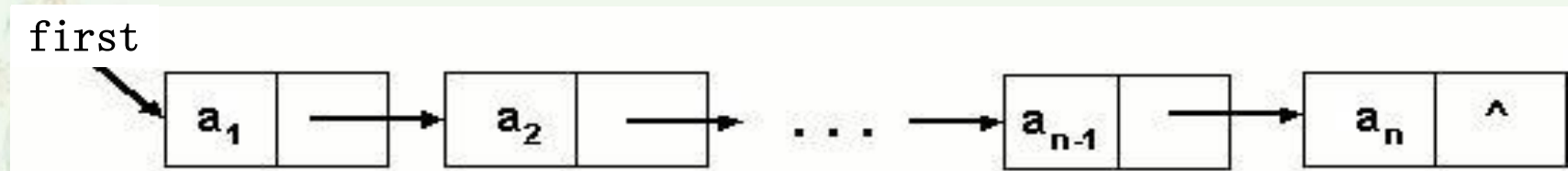
```
void main()  
{ cout<<fact(5); }
```



# 练习

- 输出单链表中所有结点的数据值
- 求单链表的表长.
- 单链表的建立

# 1.输出单链表各结点的元素值.



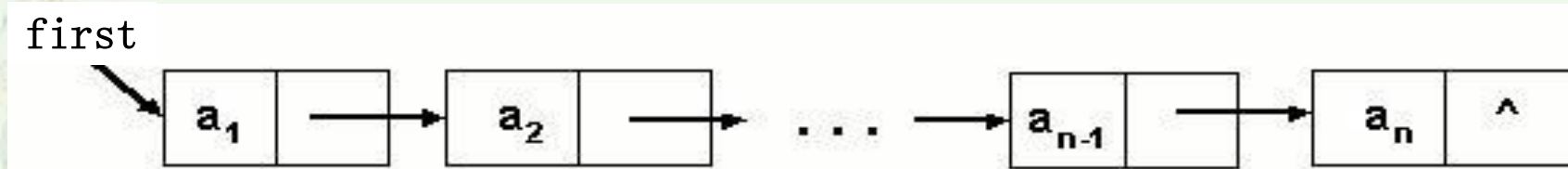
# 方法一

```
void print2(linknode *head) //正向输出
{
    if (head==0) return ;
    else
    {
        cout<<head->data<<" ";
        print2(head->next);
    }
}
```

## 方法二

```
void print1(linknode *head) // 反向输出
{
    if (head==0) return ;
    else
    {
        print1(head->next);
        cout<<head->data<<" ";
    }
}
```

## 2. 求单链表的表长——递归算法





# 方法一

```
int counter=0;
```

```
void Length1(linknode *h)
{
    if (h==0)    //终止条件
        return ;
    else
    {
        counter++;
        Length1(h->next); //递归项
    }
}
```

## 方法二

```
int counter=0;
```

```
void Length1(linknode *h)
{
    if (h==0)    //终止条件
        return ;
    else
    {
        Length1(h->next); //递归项
        counter++;
    }
}
```

## 方法三

```
int Length(linknode *h)
{
    if (h==0)    //终止条件
        return 0;
    else
        return 1+Length(h->next); //递归项
}
```

## 方法四

```
• void Length2(linknode *h,int &x)
{
    if (h==0)    //终止条件
        return ;
    else
    {
        x++;
        Length2(h->next,x); //递归项
    }
}
```

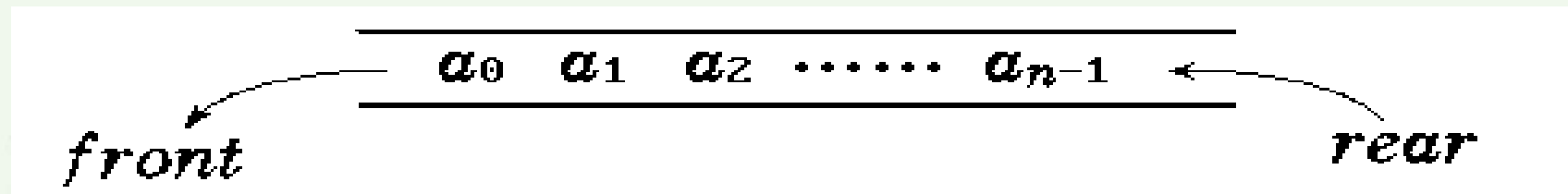
# 练习

- 单链表的建立

# 练习

- 在单链表中查找一个元素 $x$
- 求一个数组全部元素的和值。
- 以递减的顺序输出一个递增有序单链表中的元素值。
- 将单链表进行逆置处理。

# 队列 (Queue)——逻辑结构



- 定义

- 队列是只允许在一端删除，在另一端插入的线性表
- 队头(*front*):允许删除的一端
- 队尾(*rear*):允许插入的一端

- 特性

- 先进先出(*FIFO, First In First Out*)

## ADT queue

数组：组成队列的一组数据元素

操作：

- (1) 构造函数 建立一个空的队列；
- (2) 取队头元素 `getFront(x)` 若队列不空，则结果为队头元素，否则结果为空元素；
- (3) 入列操作 `EnQueue(x)` 若队列未满，则把元素 $x$ 插入到队尾，否则产生出错；
- (4) 出列操作 `DeQueue(x)` 若队列不空，则结果为队头元素且删除队头元素，否则结果为空元素；
- (5) 判空列 `IsEmpty` 如果队列为空，则结果为“真”，否则为“假”。

end ADT queue



# 队列的抽象数据类型

```
template <class T>
```

```
class Queue {
```

```
public:
```

```
    Queue() { };           //构造函数
```

```
    ~Queue() { };         //析构函数
```

```
    virtual bool EnQueue(T x) = 0;           //进队列
```

```
    virtual bool DeQueue(T& x) = 0;          //出队列
```

```
    virtual bool getFront(T& x) = 0;         //取队头
```

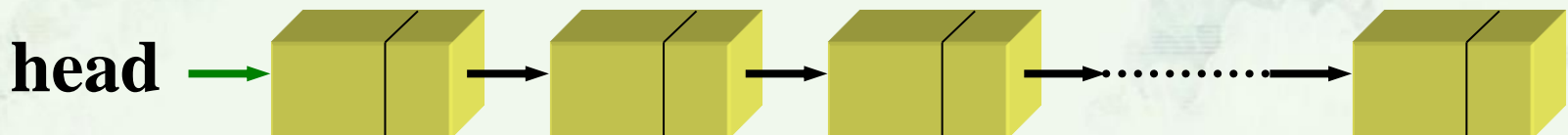
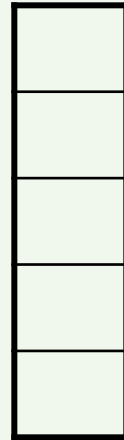
```
    virtual bool IsEmpty() const = 0;        //判队列空
```

```
    virtual bool IsFull() const = 0;        //判队列满
```

```
};
```

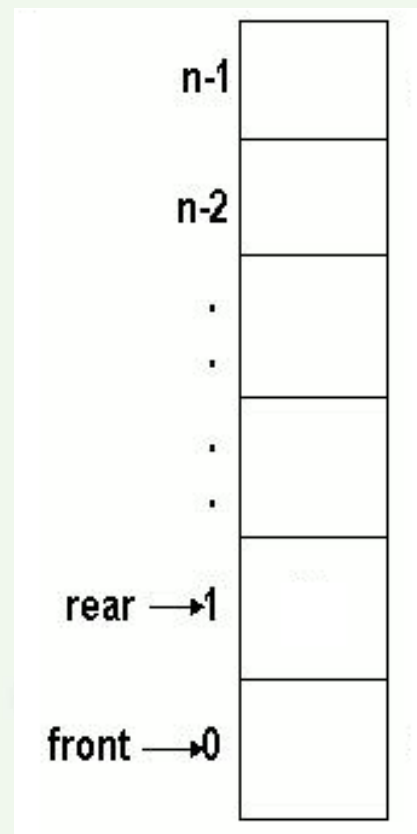
# 队列在计算机中的实现

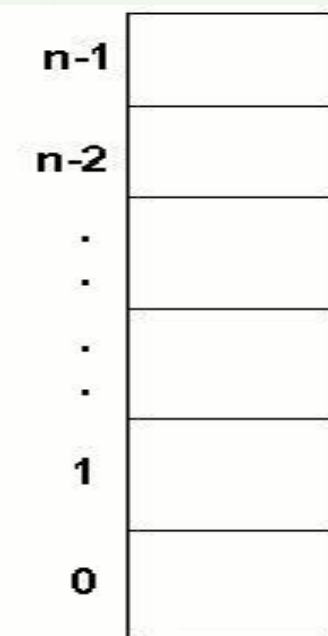
- 1. 存储结构
  - 顺序存储
  - 链式存储
- 2. 操作的实现



# 队列的顺序存储

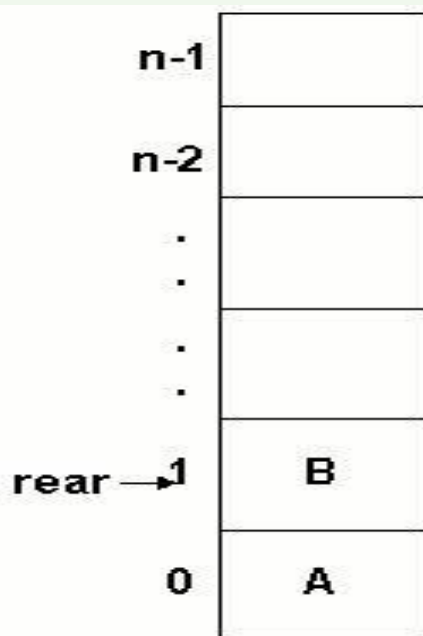
- 顺序队列: 顺序队列使用一个一维数组来存储数据元素。
- 队头位置指示器 (front):
  - 指示当前队头元素在队列
  - 中实际位置的前一个位置
- 队尾位置指示器 (rear):
  - 指示当前队尾元素在队列
  - 中实际位置。





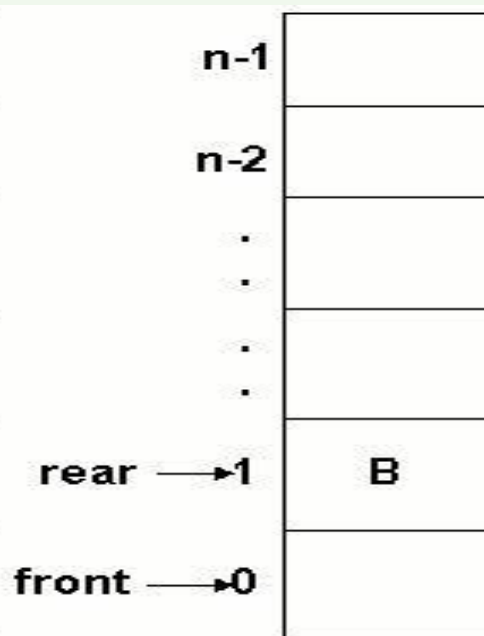
front  $\rightarrow$  -1  
 rear  $\rightarrow$  -1

(a) 初始队



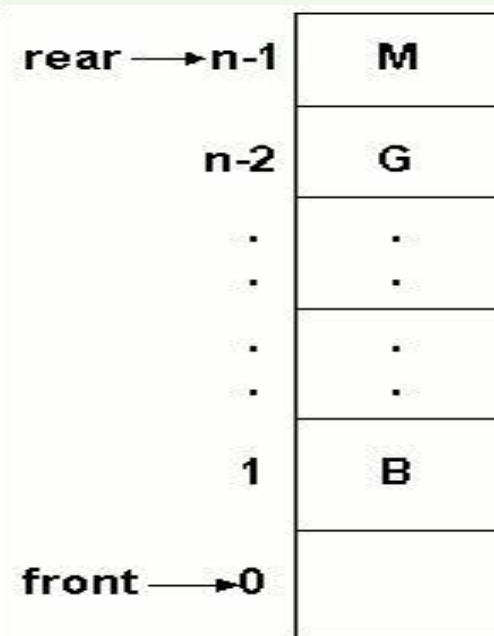
front  $\rightarrow$  -1

(b) A、B入队



front  $\rightarrow$  0

(c) A出队



(d) 队满

# 队列的数组存储表示——顺序队列

```
template <class T>
class SeqQueue : public Queue<T> { //队列类定义
protected:
    int rear, front;                //队尾与队头指针
    T *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
```

**bool EnQueue(const T &x);**      //新元素进队列

**bool DeQueue(T& x);**      //退出队头元素

**bool getFront(T& x);**      //取队头元素值

**void makeEmpty() { front = rear = -1 ; }**

**bool IsEmpty() const { return front == rear; }**

**bool IsFull() const**

**{ return ((rear == maxSize-1; }**

**int getSize() const**

**{ return rear-front; }**

**};**

## 1.入列操作

步骤:

- 1) 先检查队列是否已满, 若队列满则进行“溢出”错误处理;
- 2) 将队尾位置指示器后移一个位置;
- 3) 将新元素填入队尾位置;

• 算法的实现如下:

```
bool SeqQueue<T> ::EnQueue(const T &x ) //新元素进队列
{
    if ( rear== maxSize-1 ) // 队列已满的处理
    { cout<<"overflow"; return false ; }
    else {
        rear++; // 队尾位置指示器后移一个位置
        elements[rear]=x; // 填入新元素
        return true;
    }
} // EnQueue
```



## 2.出列操作

算法步骤为：

- (1) 检查队列是否为空列，如果队列为空，则作出错处理并返回假；
- (2) 将队头位置指示器后移一个位置；
- (3) 将队头元素暂存到一个中间变量中；
- (4) 如果队列中只剩下一个元素，则将队头、队尾指示器同时设置为-1；
- (5) 返回成功信息。

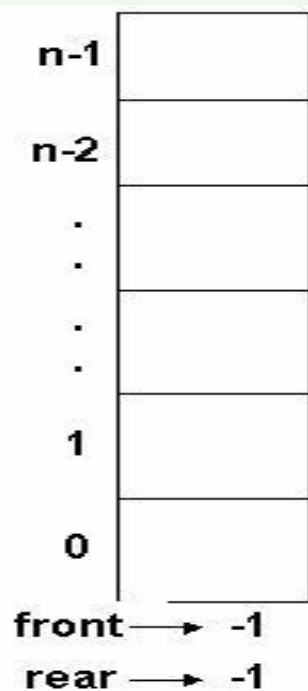
算法实现如下：

```
bool SeqQueue<T>:: DeQueue(T & x);    //退出队头元素
{
    if (front==rear) //队列为空的处理
    {    cout<< "underflow"; return false; //不成功信息}
    else {
        front++; //队头位置指示器后移一个位置
        x=elements[front]; //暂存队头元素
        return true; //返回出列成功信息
    }
} // DelQueue
```

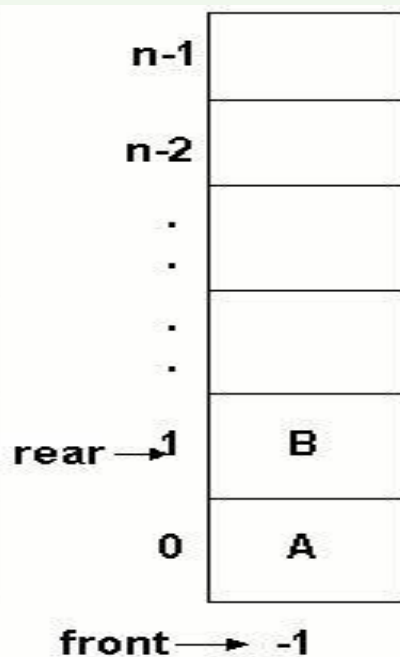


- 上述算法的时间复杂度均为 $O(1)$

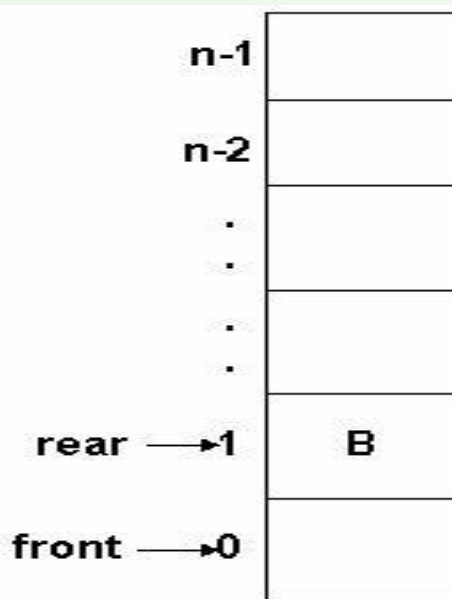
# 顺序队列的问题分析



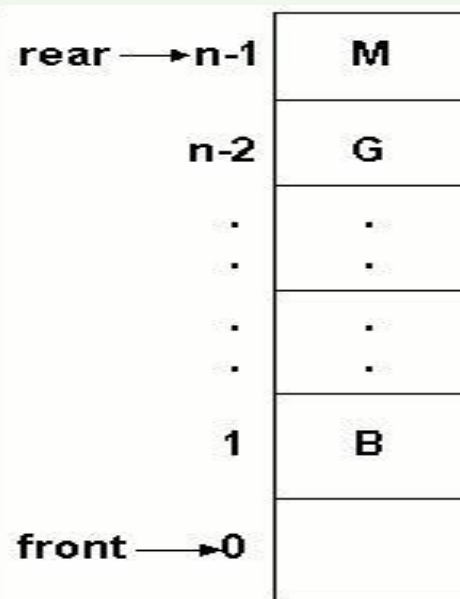
(a) 初始队



(b) A、B入队



(c) A出队



(d) 队满

# 顺序队列问题分析

- “假溢出”的最坏情况:
- $\text{rear} = \text{maxSize} - 1, \text{front} = \text{maxSize} - 1$  (空列)
- 顺序队列的空间利用率非常低, 浪费存储空间。

# 解决方案

- “假溢出”解决方法：
- (1) 整体性移动
- (2) 循环队列

# 循环队列的实现

- 如何实现首尾结合?
- 方法1:用条件判断来实现首尾结合

```
if( r == maxSize-1)  
    r=0;  
else  
    r++;
```

方法2: 只要把队头、队尾指示器对maxSize取模后加1即可。

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

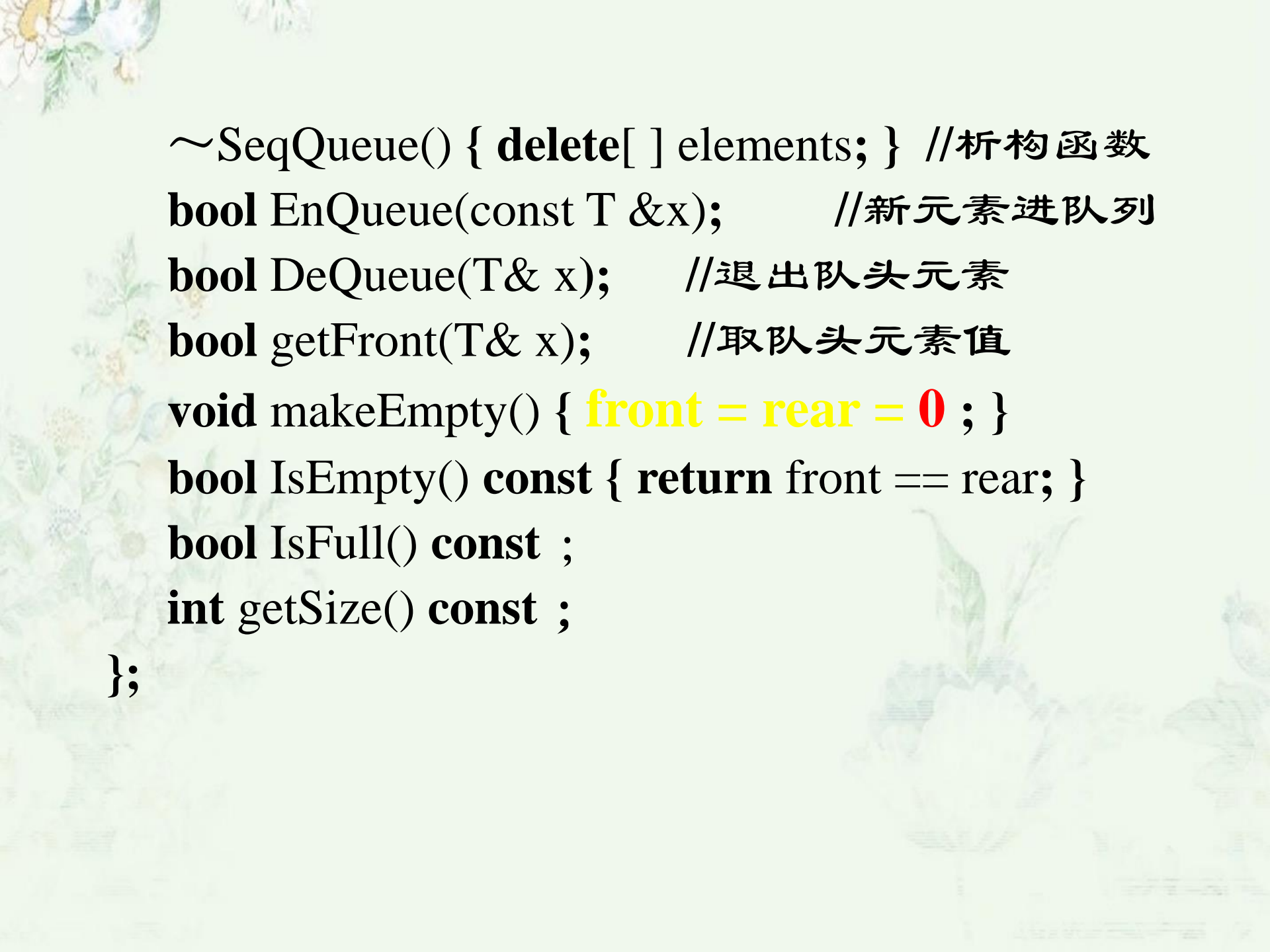
**注意事项:** 为了算法的方便, 我们可以约定:  
初始时, 队头、队尾指示器均指向最后一个单元位置, 即  $\text{front} = \text{rear} = \text{maxSize} - 1$ 。

当然: 初始化时,  $\text{front} = \text{rear} = 0$  也可以。

# 队列的数组存储表示——顺序队列

```
template <class T>
class SeqQueue : public Queue<T> {    //队列类定义
protected:
    int rear, front;                  //队尾与队头指针
    T *elements;                      //队列存放数组
    int maxSize;                      //队列最大容量
public:
    SeqQueue(int sz = 10);           //构造函数
```

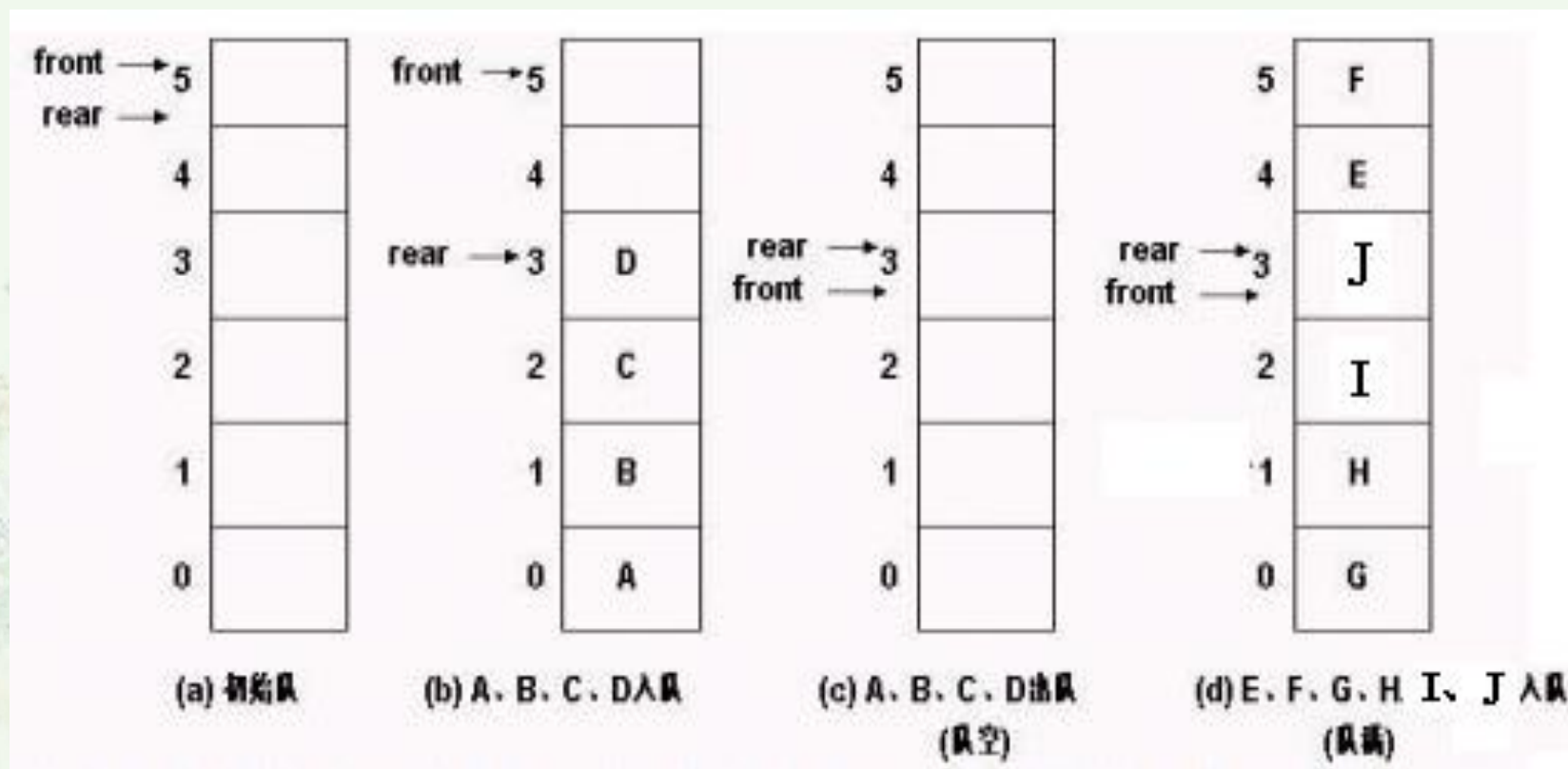
P116



```
~SeqQueue() { delete[ ] elements; } //析构函数
bool EnQueue(const T &x);      //新元素进队列
bool DeQueue(T& x);      //退出队头元素
bool getFront(T& x);      //取队头元素值
void makeEmpty() { front = rear = 0 ; }
bool IsEmpty() const { return front == rear; }
bool IsFull() const ;
int getSize() const ;
};
```



## 循环队列存在的问题



队列“空”、“满”分不开 ( $\text{front} = \text{rear}$ )

## 解决方法

- 方法一:使用一个计数器记录队列中元素的总数（实际上是队列长度）。
- 方法二:另设一个布尔变量以区别队列的空和满；
- 方法三: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（注意：front所指的单元始终为空）；

# 循环队列 (Circular Queue)

- 队列存放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:  $\text{front} = (\text{front} + 1) \% \text{maxSize};$
- 队尾指针进1:  $\text{rear} = (\text{rear} + 1) \% \text{maxSize};$
- 队列初始化:  $\text{front} = \text{rear} = 0;$
- 队空条件:  $\text{front} == \text{rear};$
- 队满条件:  $(\text{rear} + 1) \% \text{maxSize} == \text{front}$

# 解决方法三的实现

## 入队操作:

bool EnQueue (const T & x)

功能：在循环队列中插入元素el。若循环队列未滿，则插入el为新的队尾元素并返回函数值true，否则队列的状态不变且返回函数值false。

(1) 判是否队列滿，若队列滿则返回false，否则：

(2) 将el从队尾插入队列，尾指针加1并返回true。

```
template <class T>
```

```
bool SeqQueue <T>::EnQueue (const T & x)
```

```
{ if ( (rear+1)% maxSize == front ) return false;
```

```
  else {
```

```
      rear= (rear+1)% maxSize;
```

```
      elements[rear]=x;
```

```
      return true ;
```

```
  }
```

```
}
```

## 出队操作:

T QeQueue(T & x)

功能：若循环队列非空，则从队列中取出队头元素并返回真，否则返回假。

(1)判队列是否为空队列，若为空队列则返回假，否则：

(2)保存队头元素，队列头指针加1并返回真。

```
template <class T>
```

```
T SeqQueue <T>::DeQueue(T &x)
```

```
{ if (rear == front) return false;
```

```
    else {
```

```
        front= (front+1)% maxSize;
```

```
        x=elements[front];
```

```
        return true;
```

```
    }
```

```
}
```

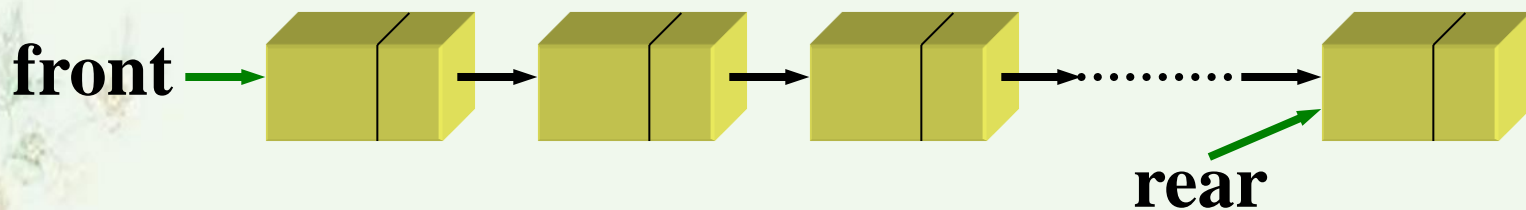
## 判满操作

```
bool SeqQueue::IsFull() const  
{ return ((rear+1)% maxSize == front); }
```

计算元素个数:

```
int SeqQueue::getSize() const  
{ return (rear-front+maxSize) % maxSize; }
```

## 队列的链接存储表示——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  $\text{front} == \text{NULL}$



## 链式队列类的定义

```
template <class T>
struct QueueNode {                                //队列结点类定义
private:
    T data;                                       //队列结点数据
    QueueNode<T> *link;                         //结点链指针
public:
    QueueNode(E d = 0, QueueNode<T>
        *next = NULL) : data(d), link(next) { }
};
```





```
template <class T>  
class LinkedQueue {  
private:
```

```
    QueueNode<T> *front, *rear; //队头、队尾指针
```

```
public:
```

```
    LinkedQueue() : rear(NULL), front(NULL) { }
```

```
    ~LinkedQueue();
```

```
    bool EnQueue(const T &x);
```

```
    bool DeQueue(T& x);
```

```
    bool GetFront(T& x);
```

```
    void MakeEmpty();    //实现与~Queue()同
```

```
    bool IsEmpty() const { return front == NULL; }
```

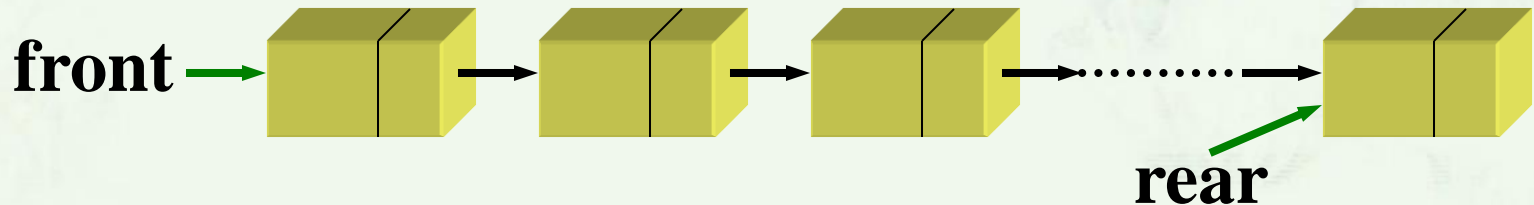
```
};
```

**P118**

## 入列操作

```
template <class T>
bool LinkedListQueue<T>::EnQueue(const T &x)
{ //将新元素x插入到队列的队尾

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



## 出列操作

```
template <class T>
```

```
//如果队列不空，将队头结点从链式队列中删去
```

```
bool LinkedQueue<T>::DeQueue(T& x)
```

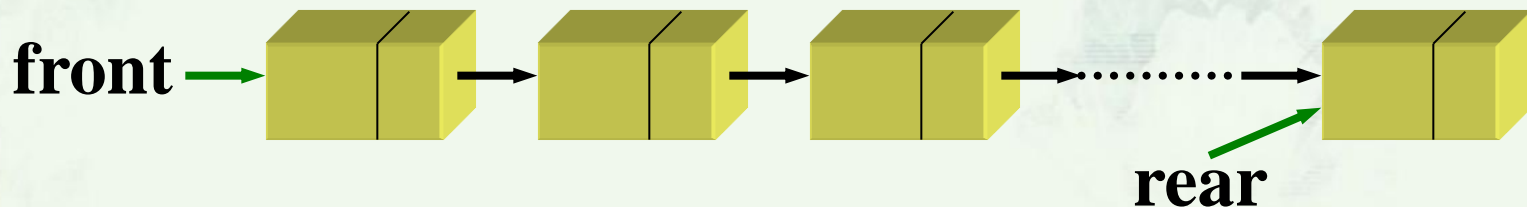
```
{ if (IsEmpty() == true) return false;    //判队空
```

```
    QueueNode<T> *p = front;
```

```
    x = front->data; front = front->link;
```

```
    delete p; return true;
```

```
};
```



## 取列头元素操作

```
template <class E>
bool LinkedQueue<E>::GetFront(E& x) {
//若队列不空，则函数返回队头元素的值
    if (IsEmpty() == true) return false;
    x = front->data; return true;
};
```

## 析构函数

```
template <class T>
LinkedList<T>::~LinkedList() { //析构函数
    QueueNode<T> *p;

    while (front != NULL)
    {
        //逐个结点释放
        p = front; front = front->link; delete p;
    }
};
```

- 从上面的算法可知，链队的出列、入列算法的时间复杂度均为 $O(1)$ 。

- 用**循环线性链表**作为队列的存储结构，就不必设置两个指针分别指向队头、队尾结点，而只要设置**一个队尾结点**就可以，请自行实现相应的算法。

# 队列的应用

- (1) 程序设计中，凡是要按照“先来先处理”的原则操作的问题都可利用队列来加以解决。
- (2) 打印缓冲。
- (3) 分时系统的管理

# 优先级队列 (Priority Queue)

- 优先级队列 每次从队列中取出的是具有最高优先权的元素
- 如下表：任务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



# 例1：杨辉三角形

1  
1 1  
1 2 1  
1 3 3 1  
1 4 6 4 1  
.....

1 1 1 1 2 1 1 3 3 1 1 4 6 4 1.....

# 杨辉三角形的形成过程

|   |   |   |   |  |  |  |  |  |
|---|---|---|---|--|--|--|--|--|
| 1 | 2 | 1 | 1 |  |  |  |  |  |
|---|---|---|---|--|--|--|--|--|



|   |   |   |   |   |  |  |  |  |
|---|---|---|---|---|--|--|--|--|
| 1 | 2 | 1 | 1 | 3 |  |  |  |  |
|---|---|---|---|---|--|--|--|--|



↑ front

|   |   |   |   |   |   |  |  |  |
|---|---|---|---|---|---|--|--|--|
| 1 | 2 | 1 | 1 | 3 | 3 |  |  |  |
|---|---|---|---|---|---|--|--|--|



↑ rear

|   |   |   |   |   |   |   |   |  |
|---|---|---|---|---|---|---|---|--|
| 1 | 2 | 1 | 1 | 3 | 3 | 1 | 1 |  |
|---|---|---|---|---|---|---|---|--|



- 1.设有编号为1,2,3,4的四辆列车，顺序进入一个栈式结构的站台，试写出这四辆列车开出车站的所有可能的顺序。
- 2.写出一个算法，借助栈将一个单链表逆置。
- 3.对于循环队列，试写出求队列中元素个数的算法。