第3章 栈和队列

——刘亮亮

上海对外经贸大学
SHANGHAI UNIVERSITY OF INTERNATIONAL BUSINESS AND ECONOMICS

上章回顾

- 线性结构：一对一的关系，除第一个和最后一个，每个元素只有一个直接前驱和一个直接后继。
- 线性表：最简单最常用的线性结构
- 线性表根据存储结构：顺序表与链表
- 顺序表与链表的抽象类型定义以及操作
- 重点操作：插入元素、删除元素等

本章要点

- 另外两种非常重要的线性结构：栈和队列
- 栈和队列：也称为**操作受限**的线性表
- 栈的抽象数据类型定义以及操作：**进栈**和**出栈**
- 栈的应用举例
- 队列的抽象数据类型定义以及操作：**入队**和**出队**
- 队列的应用举例

目录

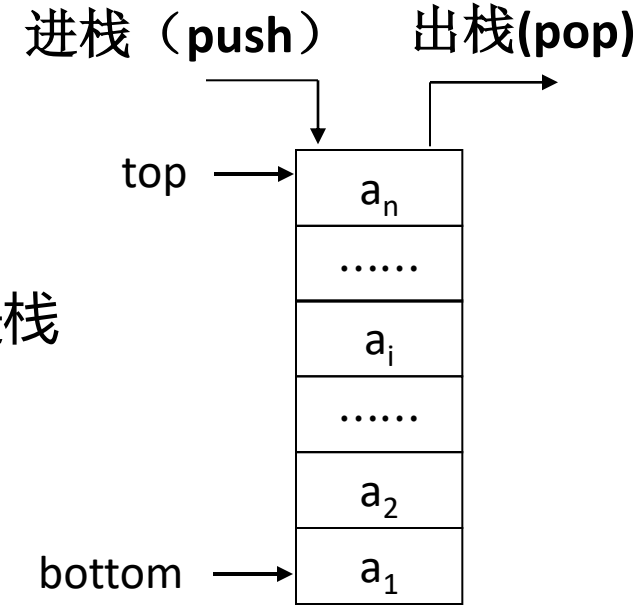
- 栈
- 栈的应用
- 队列
- 队列的应用

栈

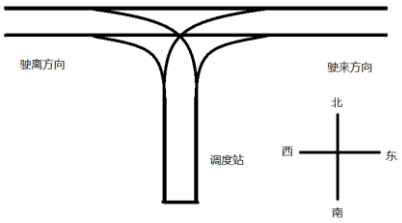
- **栈的定义**
  - 栈(Stack)：是限制在表的一端进行插入和删除操作的线性表。又称为**后进先出**LIFO (Last In First Out)或**先进后出**FILO (First In Last Out)线性表。
  - 栈顶(Top)：允许进行插入、删除操作的一端，又称为表尾。用栈顶指针(top)来指示栈顶元素。
  - 栈底(Bottom)：是固定端，又称为表头。
  - 空栈：当表中没有元素时称为空栈。

栈

- **栈的示意图**
  - 设栈S=(a_1，a_2，…a_n)，
    则a_1称为栈底元素，a_n为栈顶元素
  - 栈中元素按a_1，a_2，…a_n的次序进栈
  - 退栈的顺序：a_n，…，a_2，a_1
- **现实生活的栈：**
  - 铁路调度站
  - 弹夹



栈示意图



栈

- **栈的抽象数据类型定义**

ADT Stack{

数据对象：D ={ a_i | a_i ∈ ElemSet，i=1,2,…,n，n≥0 }

数据关系：R ={<a_{i-1}, a_i> | a_{i-1}，a_i ∈ D，i=2,3,…,n }

基本操作：初始化、进栈、出栈、取栈顶元素等

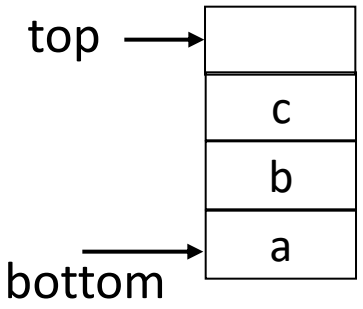
} ADT Stack

- **栈的两种重要的操作：进栈Push和出栈Pop**

栈

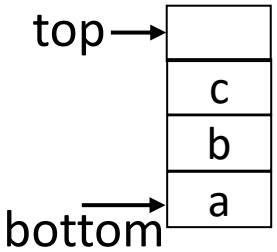
- **栈的表示与实现**
  - 顺序栈——表示
    - ✓利用一组地址连续的存储单元来依次存放自栈底到栈顶的数据元素，同时设定栈顶指针top指示栈顶元素在顺序栈中的位置。
    - ✓栈顶指针top始终指向栈顶元素的下一个位置。

```
//顺序栈的表示
#define  STACK_SIZE  100    /*  栈初始向量大小  */
#define STACKINCREMENT 10   /*  存储空间分配增量  */
#typedef  int  ElemType ;
typedef struct sqstack{
        ElemType  *bottom;     /*  栈不存在时值为NULL  */
        ElemType  *top;      /*  栈顶指针  */
        int   stacksize ;     /*  当前已分配空间，以元素为单位
        */
    }SqStack ;
```



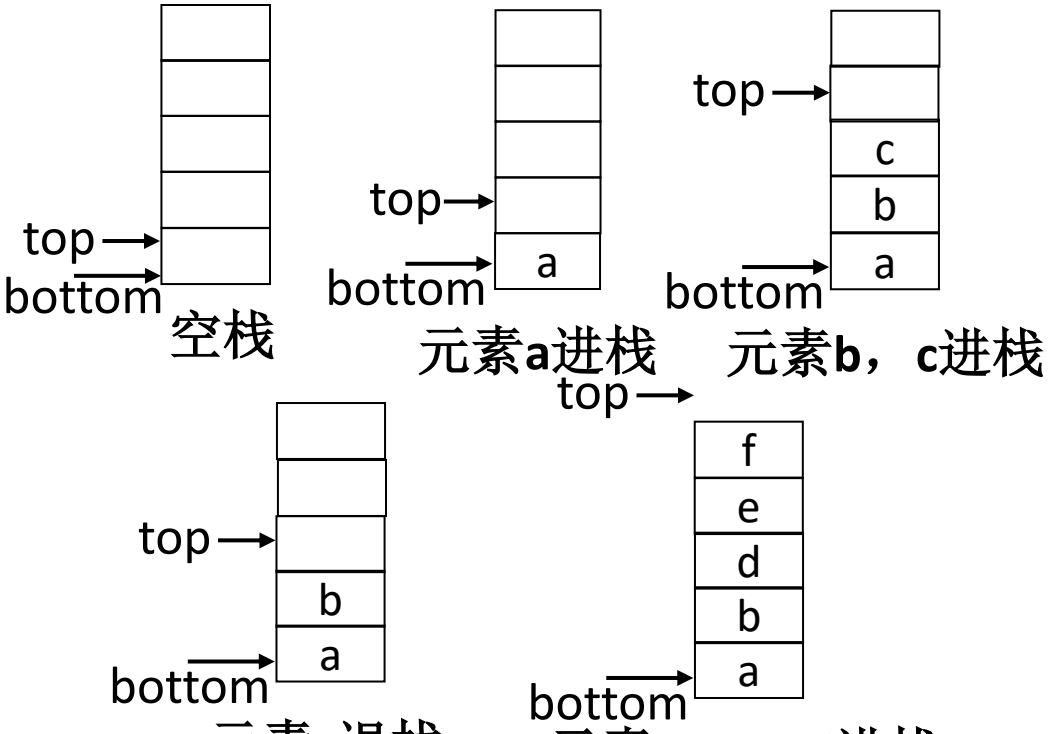
栈

- **栈的表示与实现**
  - 顺序栈——表示
    - ✓用bottom表示栈底指针，栈底固定不变的；
    - ✓用top（称为栈顶指针）指示当前栈顶位置，栈顶则随着进栈和退栈操作而变化。
    - ✓栈空：top=bottom作为栈空的标记
    - ✓结点进栈：首先将数据元素保存到栈顶(top所指的当前位置)，然后执行top加1，使top指向栈顶的下一个存储位置;
    - ✓结点出栈：首先执行top减1，使top指向栈顶元素的存储位置，然后将栈顶元素取出。



栈

- **栈的表示与实现**
  - 顺序栈——表示



栈变化示意图

栈

- **栈的表示与实现**
  - 顺序栈——实现
    - ✓操作1：栈的初始化

```
Status Init_Stack(void){
        SqStack  S ;
        S.bottom=(ElemType *)malloc(STACK_SIZE
        *sizeof(ElemType));
        if (! S.bottom) return  ERROR;
        S.top=S.bottom ;   /*  栈空时栈顶和栈底指针相同  */
        S. stacksize=STACK_SIZE;
        return OK ;
}
```

栈

- **栈的表示与实现**
  - 顺序栈——实现
    - ✓操作2：进栈push
      - ◘算法思想：
        - ◆ Step 1:判断栈是否满，如果满重新分配

          if (S.top-S.bottom>=S. stacksize-1)//栈满
        - ◆ Step 2:将元素赋值给栈顶指针top指向的位置

          ***S.top=e**
        - ◆ Step 3:栈顶指针top加1后移

          S.top++

栈

- **栈的表示与实现**
  - 顺序栈——实现
    - ✓操作2：进栈push
      - ◘算法描述：

```
Status push(SqStack S , ElemType  e){
      if  (S.top-S.bottom>=S. stacksize-1) {
          // / 栈满，追加存储空间
           S.bottom=(ElemType *)realloc((S. STACKINCREMENT+STACK_SIZE)
           *sizeof(ElemType));
           if (! S.bottom)  return  ERROR;
          S.top=S.bottom+S. stacksize ;
           S. stacksize+=STACKINCREMENT ;
      }
      *S.top=e;  //进栈，  e成为新的栈顶
       S.top++ ; //    栈顶指针加1，
      return OK;
   }
```

算法分析：
时间复杂度:O(1)

栈

- **栈的表示与实现**
  - 顺序栈——实现
    - ✓操作3：出栈pop
      - ◘算法思想：
        - ◆ Step 1:判断栈是否空，如果空返回失败标识

          if (s.top==s.bottom) //空栈
        - ◆ Step 2:栈顶指针减1

          ***s.top--；//栈顶指针减1**
        - ◆ Step 3:返回出栈的元素

          e=*s.top;

栈

- **栈的表示与实现**
  - 顺序栈——实现
    - ✓操作3：出栈push
      - ◘算法描述：

```
Status pop( SqStack   S, ElemType  *e )
//弹出栈顶元素
{
    //判断栈空
    if ( S.top== S.bottom )
        return ERROR ;     // 栈空，返回失败标志
    S.top-- ;
    e=*S. top ;
    return  OK ;
}
```

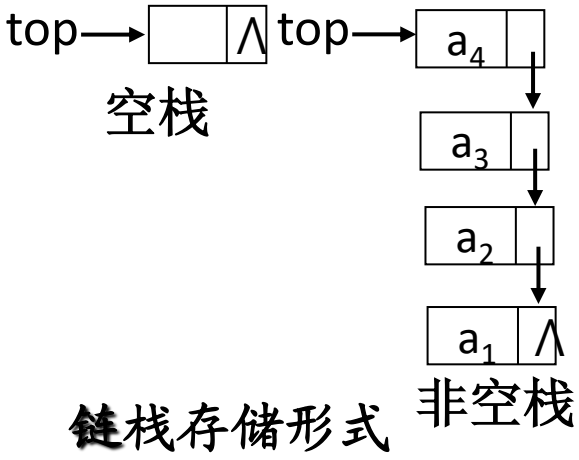
算法分析：

时间复杂度:O(1)

栈

- **栈的表示与实现**
  - 链栈——表示
    - ✓**栈的链式存储结构称为链栈，是运算受限的单链表。**
    - ✓**其插入和删除操作只能在表头位置上进行。**
    - ✓**栈顶指针top就是链表的头指针**

```
//链栈的结点类型说明如下：
typedef  struct  Stack_Node
{
      ElemType   data ;
       struct Stack_Node  *next ;
} Stack_Node ;
```



链栈存储形式

栈

- **栈的表示与实现**
  - 链栈——实现
    - ✓链栈的操作和单链表的操基本一样，唯一区别是只能在头指针处进行插入和删除。
    - ✓参考单链表的操作，自行实现
    - ✓例如：进栈

```
Status push(Stack_Node *top , ElemType  e)
 {
        Stack_Node  *p ;
        p=(Stack_Node  *)malloc(sizeof(Stack_Node)) ;
        if (!p)  return  ERROR;
              //申请新结点失败，返回错误标志
        p->data=e ;
        p->next=top->next  ;
        top->next=p ;
        return OK;
 }
```

思考题

- **问题1：若已知一个栈的入栈顺序是1、2、3、4.其出栈序列为P1、P2、P3、P4、，则P2、P4不可能是（）**
  - A. 2、4 B. 2、 1 C. 4 、 3 D. 3 、4
- **问题2：假设以I和O 分别表示入栈和出栈操作。栈的初态和终态均为空，入栈和出栈的操作序列可表示为仅由I和O组成的序列，可以操作的序列称为合法序列，否则称为非法序列。**

  下面所示的序列中哪些是合法的?
  - A. IOIIOIOO B. IOOIOIIO C. IIIOIOIO D. IIIOOIOO

目录

- 栈
- 栈的应用
- 队列
- 队列的应用

栈的应用

- 栈是一种非常重要的数据结构
- 程序设计中常用的工具和数据结构
- 在软件系统中广泛使用
- 常见应用：数制转换、括号匹配检验、表达式求值等

栈的应用

- **应用1：数制转换**
  - 分析：
    - ✓ 十进制整数N向其它进制数d(二、八、十六)的转换是计算机实现计算的基本问题
    - ✓ 转换法则：n=(n div d)*d+n mod d
    - ✓ 示例： $(1348)_{10}= (2504)_8$
    - ✓ 取余顺序：4052,而最后的输出顺序为：2504，正好符合栈的“后进先出”的特点。

//运算过程：

n	n div 8	n mod 8
1348	168	4
168	21	0
21	2	5
2	0	2

栈的应用

- **应用1：数制转换**
  - 算法思想：
    - ✓构造一个空栈

      S=Init_Stack();
    - ✓循环模d(进制数)，取得余数，调用栈的执行进栈操作,直到除的结果为0

      ```
      while(n>0){
            k=n%d;
            push(S,k);
            n=n/d;
       }
      ```
    - ✓遍历栈，执行出栈操作，并且输出

      ```
      while(!StackEmpty(S))//S.top!=0{
          pop(S,e);
      }
      ```

栈的应用

- **应用1：数制转换**
  - 算法描述：

```
void conversion(int  n , int  d) {
        //将十进制整数N转换为d(2或8)进制数
        SqStack  S ;
        int k, *e ;
         S=Init_Stack();
         while  (n>0) {//求出所有的余数，进栈
            k=n%d ;
            push(S , k) ;
            n=n/d ;
          }
         while  (S.top!=0) {  //栈不空时出栈，输出
             pop(S, e) ;
              printf("%1d" , *e) ;
         }
   }
```

思考：

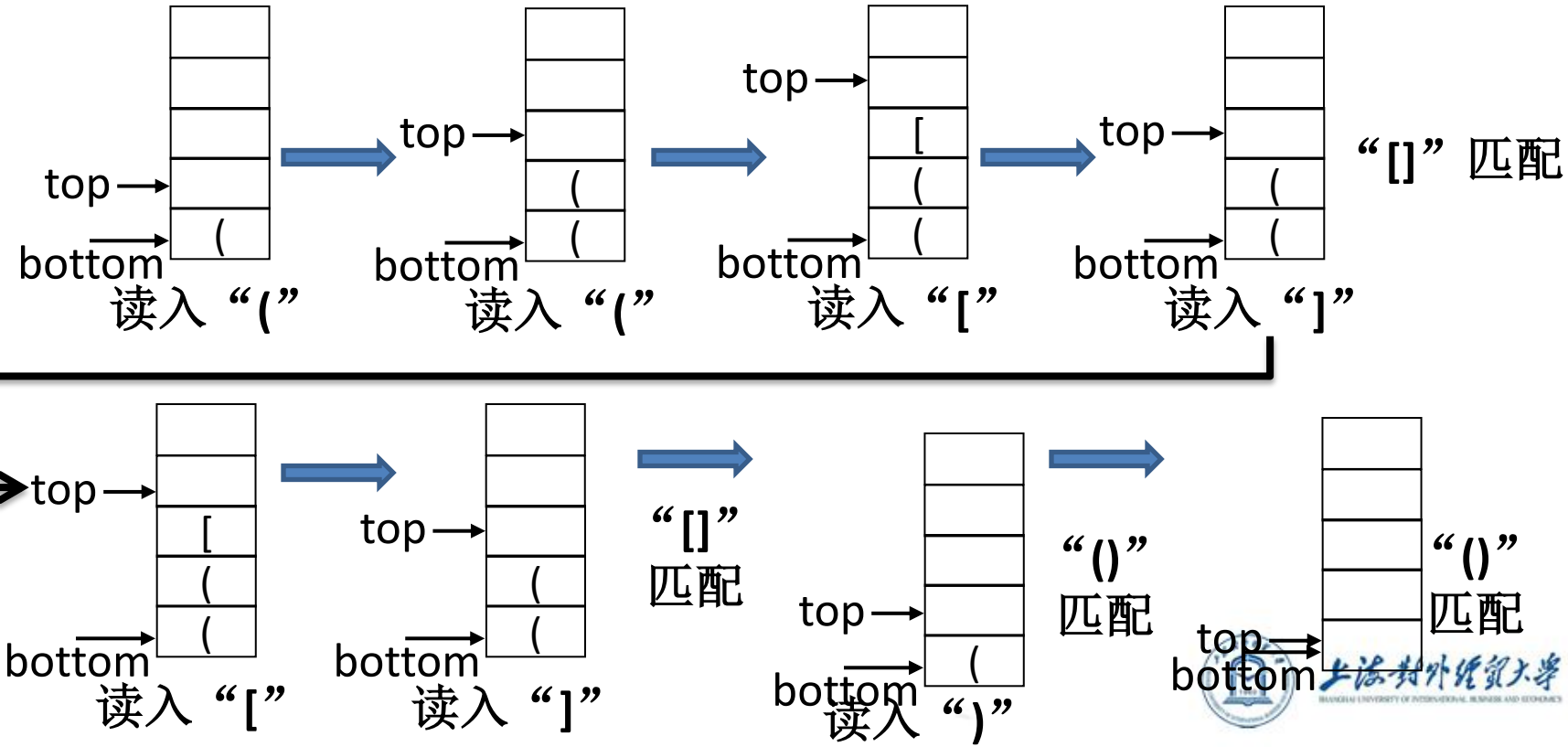
用数组等也可以实现，为何用栈来实现，有什么好处？？？

栈的应用

- **应用2：括号匹配检查**
  - 分析：从左至右扫描一个字符串(或表达式)，则每个右括号将与最近遇到的那个左括号相匹配。则可以在从左至右扫描过程中把所遇到的左括号存放到堆栈中。每当遇到一个右括号时，就将它与栈顶的左括号(如果存在)相匹配，同时从栈顶删除该左括号。
  - 算法思想：
    - ✓ (1)设置一个栈，当读到左括号时，左括号进栈。
    - ✓ (2)当读到右括号时，则从栈中弹出一个元素，与读到的左括号进行匹配，若匹配成功，继续读入；
    - ✓ (3)否则匹配失败，返回FLASE。
    - ✓ (4)如果最后栈为空，则匹配成功

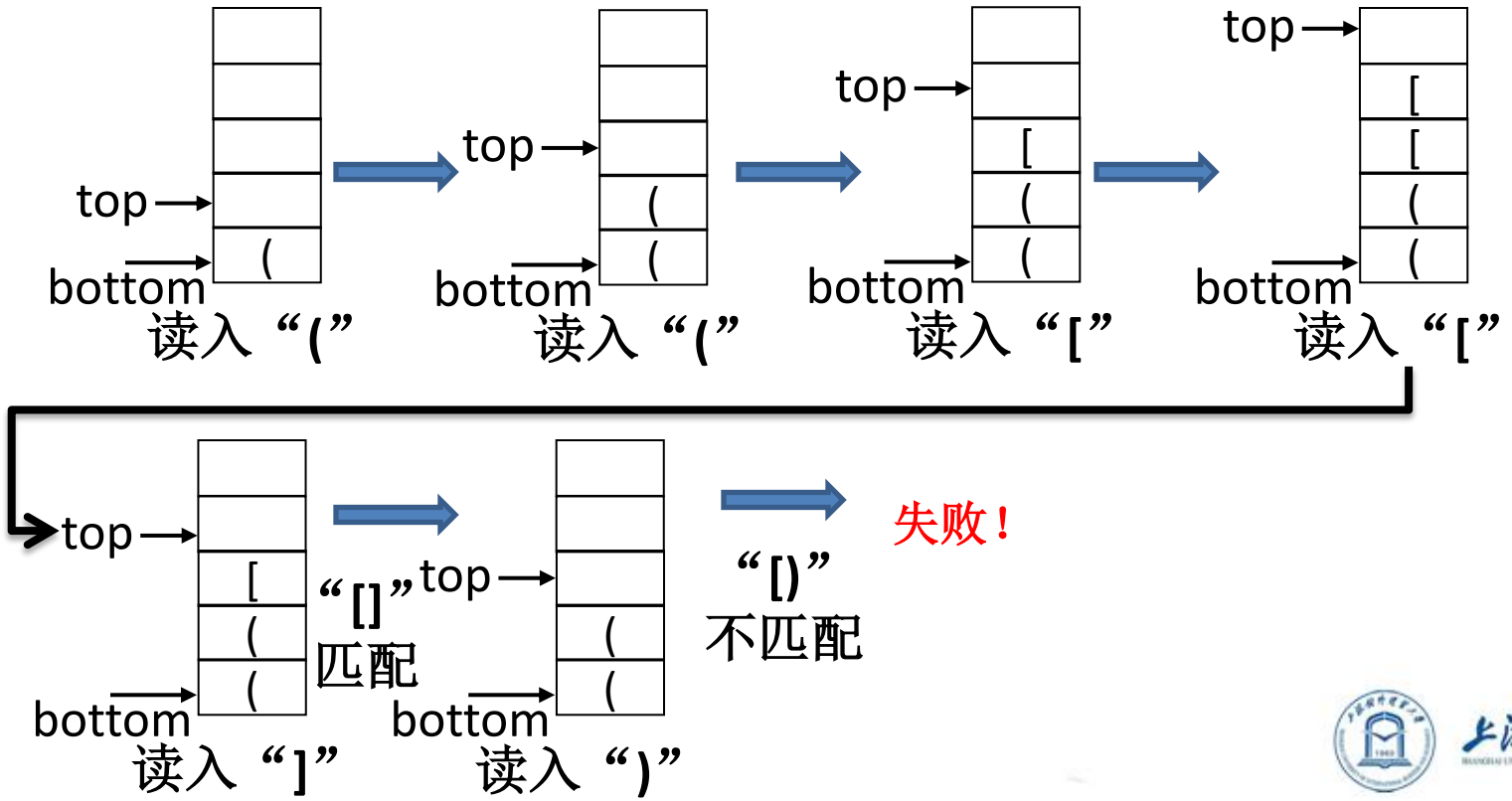
栈的应用

- **应用2：括号匹配检查**
  - 示例：　((()))



栈的应用

- **应用2：括号匹配检查**
  - 示例1： ((([[]))) ——不合法



栈的应用

- **应用2：括号匹配检查**

```
#define TRUE  0
#define  FALSE -1
SqStack S;
S=Init_Stack();//堆栈初始化
int Match_Brackets( ){
    char ch , x ;
    scanf("%c" , &ch) ;
     while (asc(ch)!=13){
         if  ((ch=='(')||(ch=='[')){
             push(S,ch);
         } else if(ch==']'){
             x=pop(S) ;
            if (x!='['){
                printf("'['括号不匹配" ) ;
                return FLASE  ;
            }
         } else if  (ch==')')  {
            x=pop(S) ;
           if (x!='(') {
                printf("'('括号不匹配" ) ;
             return FLASE  ;
          }
         }
    }
```

```
    if  (S.top!=0) {
       printf("括号数量不匹配！" ) ;
     return FLASE  ;
     }
    return TRUE  ;
}
```

Note:

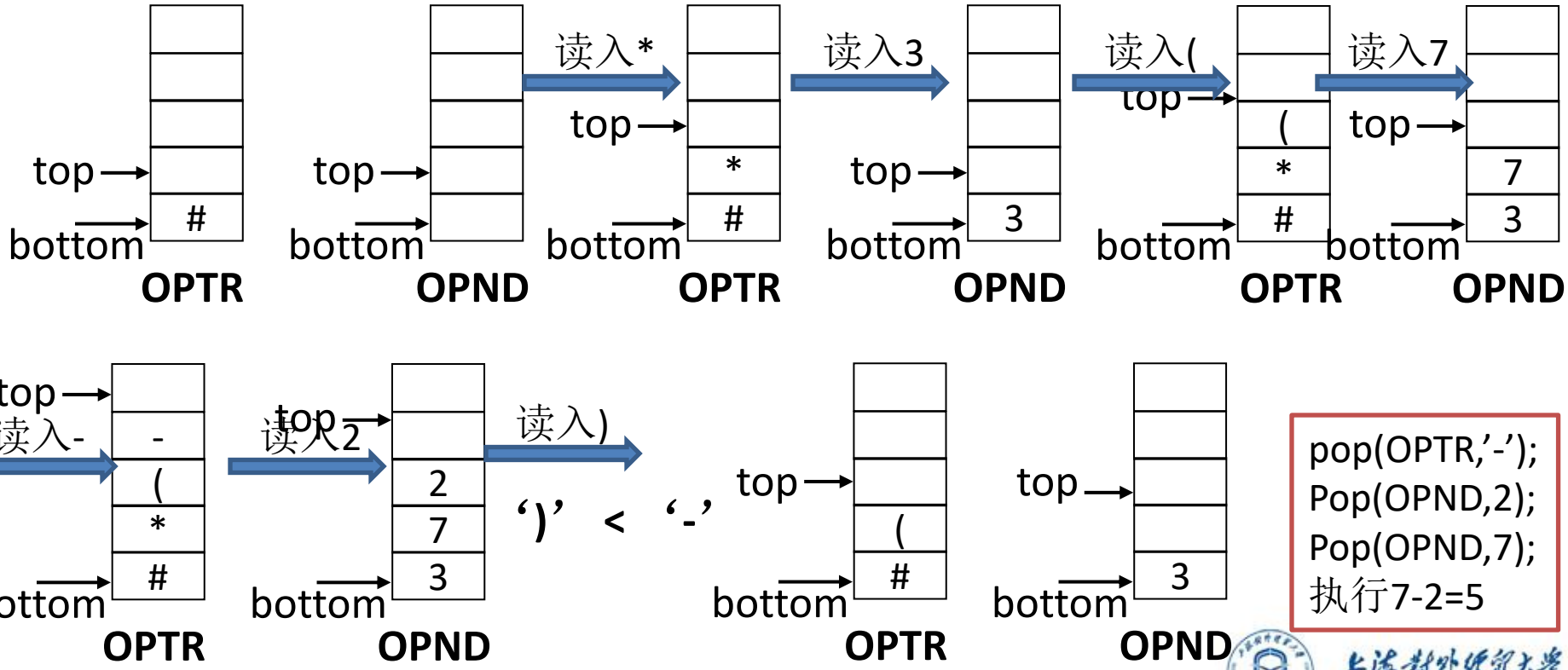
编程没有什么诀窍，唯有不断的实践！

栈的应用

- **应用3：表达式求值**
  - 分析：
    - ✓表达式求值是程序设计语言编译中一个最基本的问题；
    - ✓表达式：操作符、运算符、界限符组成；
    - ✓采用算符优先法
  - 算法思想：
    - ✓(1)定义两个工作栈，一个运算符栈OPTR、一个操作数栈OPND-寄存操作数和运算结果
    - ✓(2)首先置操作数栈OPND为空栈，表达式起始符“#”为运算符栈的栈底元素；
    - ✓(3)依次读入每个字符：
      - 若是操作数则进入OPND；
      - 若是运算符则和OPTR栈的栈顶元素比较优先级后进行运算，直到求值完毕（即：OPTR栈顶元素和当前字符为“#”）

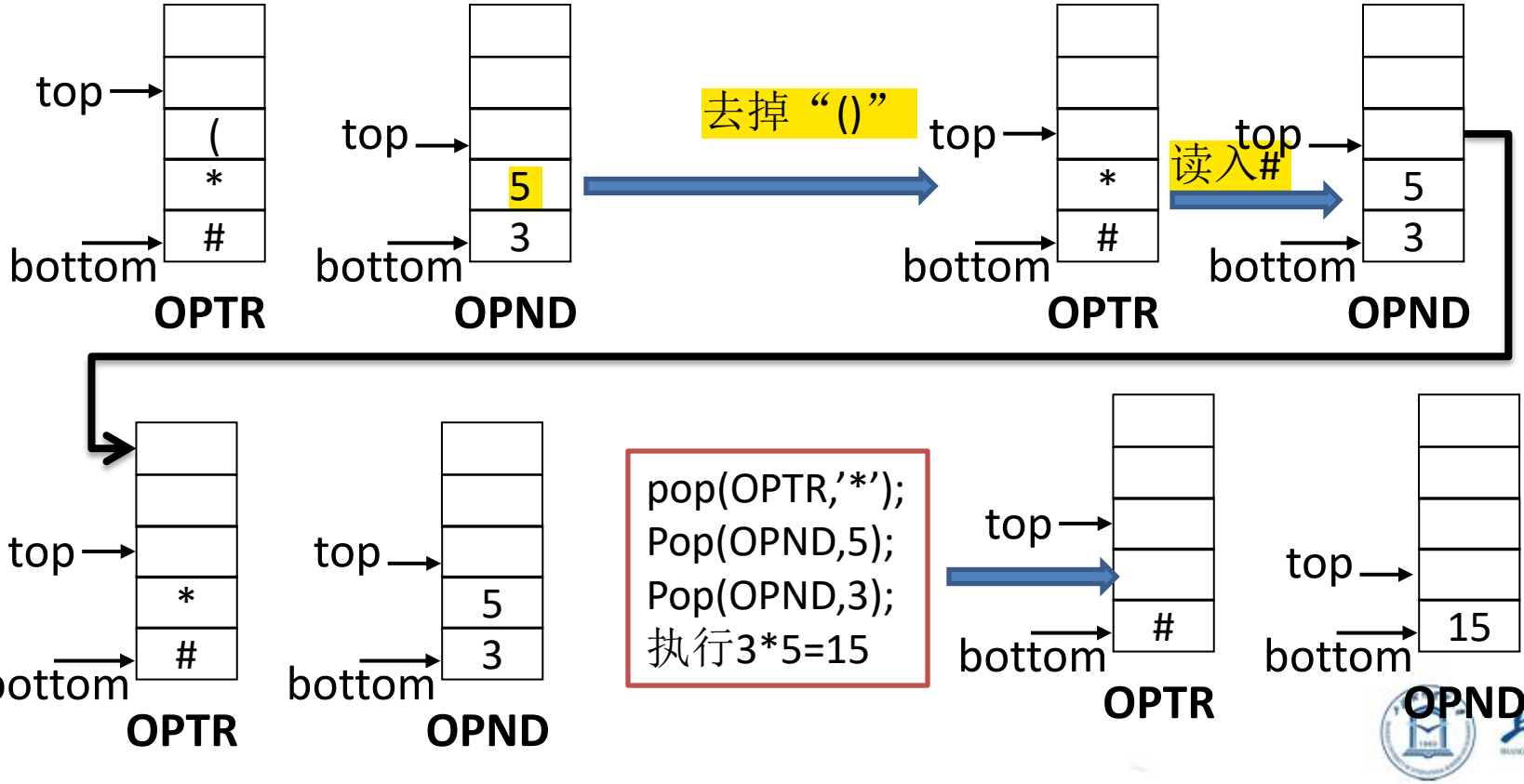
栈的应用

- **应用3：表达式求值**
  - 图解：3*(7-2)



栈的应用

- **应用3：表达式求值**
  - 图解：3*(7-2)



栈的应用

- **应用3：表达式求值**

```
Operand Type EvaluateExpression(){
    InitStack(OPTR);
    Push(OPTR,'#');
    InitStack(OPND);
    c=getchar();
   while(c!='#'||GetTop(OPTR)!='#'){
      if(!In(c,OP)){//不是运算符，进OPND栈
        Push(OPND,c);
        c=getchar();
       }else{
        switch(Precede(GetTop(OPTR),c)){ //判断优先级
          case '<': //栈顶元素优先权低
            Push(OPTR,c);
            c=getchar();
            break;
          case '=': //去掉括号
            Pop(OPTR,c);
            c=getchar();
            break;
          case '>': //退栈，并且计算，结果进栈
            Pop(OPTR,theta);
            Pop(OPND,b);
            Pop(OPND,a);
            Push(OPND,Operate(a,theta,b));
            break;
        }//end swich
      }//end else
    }//end while
    return GetTop(OPND);
}
```

栈的应用

- **应用4：栈与递归（选学）**
  - 递归调用：一个函数（或过程）直接或间接地调用自己本身，简称递归(Recursive)。
  - 程序设计语言中实现递归是栈的一个非常重要的应用。
  - **有效的递归调用函数（或过程）应包括两部分：递推规则（方法），终止条件。**
  - **例如：求n!**

$$Fact(n)=\begin{cases}1 & \text{当n=0时} \quad \text{终止条件}\\ n*fact(n-1) & \text{当n>0时} \quad \text{递推规则}\end{cases}$$

栈的应用

- **应用4：栈与递归（选学）**
  - 为保证递归调用正确执行，系统设立一个“递归工作栈”，作为整个递归调用过程期间使用的数据存储区。
  - 每一层递归包含的信息如：参数、局部变量、上一层的返回地址构成一个“工作记录”。每进入一层递归，就产生一个新的工作记录压入栈顶；每退出一层递归，就从栈顶弹出一个工作记录。

栈的应用

- **应用4：栈与递归（选学）**
  - 从被调函数返回调用函数的一般步骤：
    - ✓**(1) 若栈为空，则执行正常返回。**
    - ✓**(2) 从栈顶弹出一个工作记录。**
    - ✓**(3) 将“工作记录”中的参数值、局部变量值赋给相应的变量；读取返回地址。**
    - ✓**(4) 将函数值赋给相应的变量。**
    - ✓**(5) 转移到返回地址。**

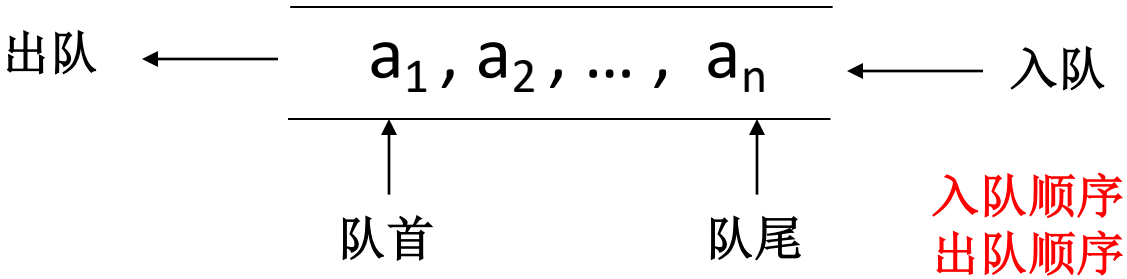
- 更多内容请参考教材3.3节

目录

- 栈
- 栈的应用
- 队列
- 队列的应用

队列

- **队列的定义**
  - 队列：也是运算受限的线性表。是一种先进先出(First In First Out，简称FIFO)的线性表。只允许在表的一端进行插入，而在另一端进行删除。
  - 队首(front)：允许进行删除的一端称为队首。
  - 队尾(rear)：允许进行插入的一端称为队尾。



入队顺序：$a_1, a_2, \ldots, a_n$
出队顺序：$a_1, a_2, \ldots, a_n$

- **现实生活中的队列**
  - 排队购物、买票、操作系统的作业队列

队列

- **队列的抽象数据类型定义**

ADT Queue{

数据对象：D ={ a_i | a_i ∈ ElemSet，i=1, 2, …, n, n >= 0 }

数据关系：R = {<ai-1, ai> | ai-1, ai ∈ D，i=2,3,…,n }

约定a1端为队首，an端为队尾。

基本操作：

Create()：创建一个空队列；

EmptyQue()：若队列为空，则返回true ，否则返回flase ；

……

InsertQue(x) ：向队尾插入元素x；

DeleteQue(x) ：删除队首元素x；

} ADT Queue

- **队列的顺序存储结构：顺序队列**
- **队列的链式存储结构：链队列**

特殊队列：双端队列——一端运行插入删除，另一端只运行插入。

队列

- **顺序队列的表示与实现**
  - 顺序队列：利用一组连续的存储单元(一维数组) 依次存放从队首到队尾的各个元素。
  - 和顺序栈类似
  - 类型定义：

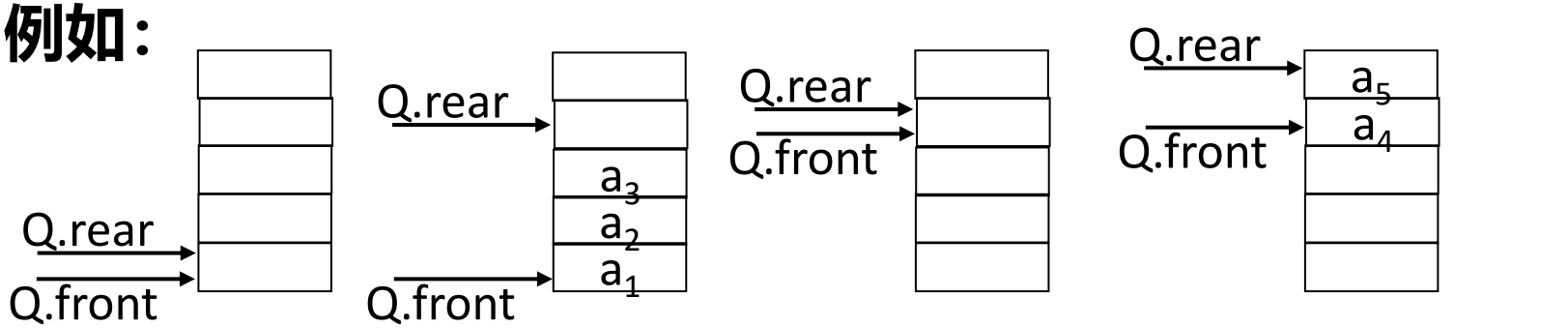
```
#define  MAX_QUEUE_SIZE   100
typedef  struct  queue
{
        ElemType   Queue_array[MAX_QUEUE_SIZE] ;
         int   front ;  //队首指针
         int  rear ;   //队尾指针
}SqQueue;
```

队列

- **顺序队列的表示与实现**
  - 设立一个队首指针front，一个队尾指针rear，分别指向队首和队尾元素。
  - 初始化：front=rear=0。
  - 入队：将新元素插入rear所指的位置，然后rear加1。
  - 出队：删去front所指的元素，然后加1并返回被删元素。
  - 队列为空：front=rear。
  - 队满：rear=MAX_QUEUE_SIZE-1或front=rear。

队列

- **顺序队列的表示与实现**
  - “假溢出”问题：**因为在入队和出队操作中，头、尾指针只增加不减小，致使被删除元素的空间永远无法重新利用。因此，尽管队列中实际元素个数可能远远小于数组大小，但可能由于尾指针已超出向量空间的上界而不能做入队操作。**
  - **例如：**



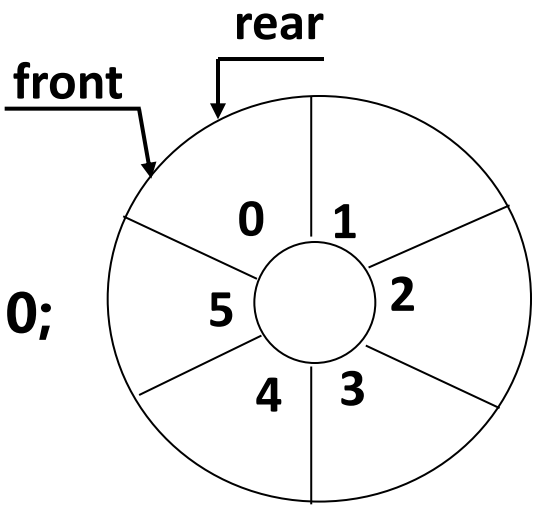
(a) 空队列　**(b)**入队**3**个元 素　**(c)** 出队**3**个元素　**(d)** 入队**2**个元素

  - 解决办法：**循环队列**

队列

- **循环队列**
  - 为充分利用向量空间，克服上述“假溢出”现象的方法是：将为队列分配的向量空间看成为一个首尾相接的圆环，并称这种队列为循环队列(Circular Queue)。
  - 在循环队列中进行出队、入队操作时，队首、队尾指针仍要加1，朝前移动。
  - 当队首、队尾指针指向向量上界(MAX_QUEUE_SIZE-1)时，其加1操作的结果是指向向量的下界0。例如：

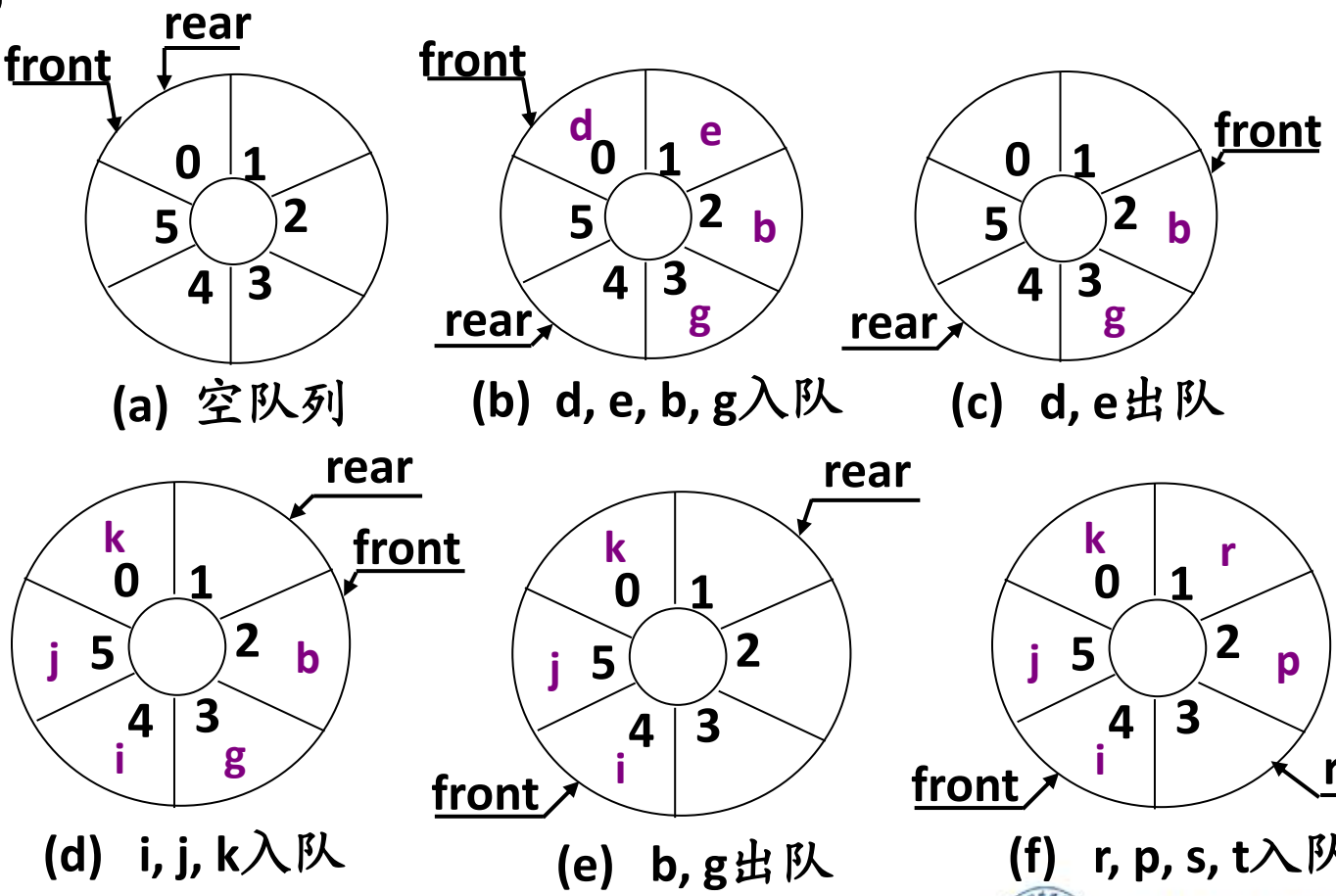
```
if (rear+1==MAX_QUEUE_SIZE) rear=0;
else rear++;
```



队列

- **循环队列**
  - 示例：

循环队列操作及指针变化情况



(a) 空队列　(b) d, e, b, g入队　(c) d, e出队

(d) i, j, k入队　(e) b, g出队　(f) r, p, s, t入队

队列

- **循环队列**
  - 问题：如何判断“队满” 还是“队空”？
    - ✓front==rear?
    - ✓无法区分队满还是队空
    - ✓如何解决?
      - 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。即：
        - **rear所指的单元始终为空。**
        - **循环队列为空：front=rear 。**
        - **循环队列满：(rear+1)%MAX_QUEUE_SIZE =front。**

队列

- **循环队列**
  - 操作1：*循环队列的初始化*

```
SqQueue Init_CirQueue(void)
{
        SqQueue  Q ;
        Q.front=Q.rear=0;
        return(Q) ;
 }
```

队列

- **循环队列**
  - 操作2：入队操作

```
Status Insert_CirQueue(SqQueue  Q , ElemType  e)
 //   将数据元素e插入到循环队列Q的队尾
{
       if  ((Q.rear+1)%MAX_QUEUE_SIZE== Q.front)
          return  ERROR;     // 队满，返回错误标志
       Q.Queue_array[Q.rear]=e ;  //  元素e入队
       Q.rear=(Q.rear+1)% MAX_QUEUE_SIZE ; //队尾指针向前移动
      return OK;
}
```

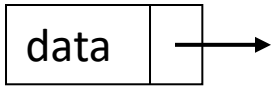
队列

- **循环队列**
  - 操作2：出队操作

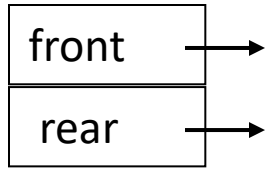
```
// 将循环队列Q的队首元素出队
Status Delete_CirQueue(SqQueue  Q, ElemType  *x )
{
    if  (Q.front== Q.rear)
        return ERROR ;     // 队空，返回错误标志
    *x=Q.Queue_array[Q.front] ;  // 取队首元素
     Q.front=(Q.front+1)% MAX_QUEUE_SIZE ; // 队首指针向前移动
    return OK ;
}
```

队列

- **链队列**
  - 队列的链式存储结构简称为链队列，它是限制仅在表头进行删除操作和表尾进行插入操作的单链表。
  - 两类不同的结点：数据元素结点，队列的队首指针和队尾指针的结点



数据元素结点



指针结点

- **数据元素结点类型定义：**

```
typedef struct Qnode
{  ElemType    data ;
     struct Qnode  *next ;
}QNode ;
```

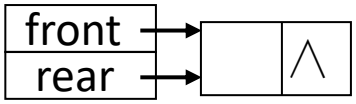
- **指针结点类型定义：**

```
typedef struct link_queue
{
        QNode  *front , *rear ;
}Link_Queue ;
```

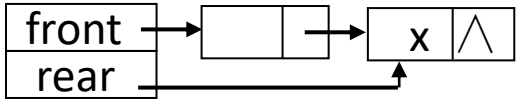
上海对外经贸大学

队列

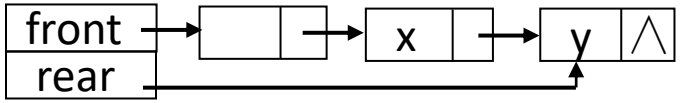
- **链队列**
  - 操作示意图



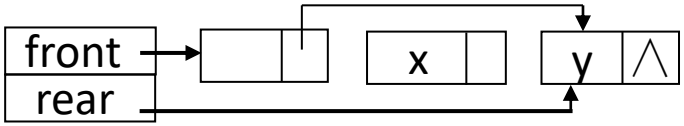
(a) 空队列



(b) x入队



(c) y再入队



(d) x出队

队列

- **链队列**
  - 操作1：链队列的初始化

```
LinkQueue *Init_LinkQueue(void)
{
    LinkQueue  *Q ;  QNode  *p ;
     p=(QNode *)malloc(sizeof(QNode)) ; /* 开辟头结点 */
     p->next=NULL ;
      Q=(LinkQueue  *)malloc(sizeof(LinkQueue)) ;
      /*  开辟链队的指针结点  */
    Q.front=Q.rear=p ;
      return(Q) ;
}
```

队列

- **链队列**
  - 操作2：入队操作

```
//将数据元素e插入到链队列Q的队尾
Status  Insert_CirQueue(LinkQueue  *Q , ElemType  e)
{
      p=(QNode *)malloc(sizeof(QNode)) ;
      if (!p) {//  申请新结点失败，返回错误标志
              return  ERROR;
      }
      p->data=e ;
      p->next=NULL;
     Q.rear->next=p ;
    Q.rear=p ;  //新结点插入到队尾
      return OK;
}
```

队列

- **链队列**
  - 操作3：出队操作

```
Status  Delete_LinkQueue(LinkQueue  *Q, ElemType *x){
    QNode *p ;
    if  (Q.front==Q.rear) {
        return ERROR ;   //  队空
     }
    p=Q.front->next ;  //  取队首结点
    *x=p->data ;
     Q.front->next=p->next ;     //  修改队首指针
    if  (p==Q.rear) {//当队列只有一个结点时应防止丢失队尾指针
         Q.rear=Q.front ;
    }
   free(p) ;
  return OK ;
}
```

目录

- 栈
- 栈的应用
- 队列
- 队列的应用

队列的应用

- 队列的应用——银行业务处理的模拟程序
- 参考教材3.5节

小结

- 两种操作受限的线性表：栈和队列
- 栈：后进先出的线性表
- 栈的存储表示：顺序栈、链栈
- 栈的重点操作：出栈和进栈（*）
- 队列：先进先出的线性表
- 队列的重点操作：入队、出队（*）
- 队列存储结构：顺序队列、循环队列、链队列

小结

数据结构 / 项	栈	队列
特点	后进先出(LIFO)	先进先出(FIFO)
插入	栈尾插入（进栈）	队尾插入（入队）
删除	栈尾删除（出栈）	队首删除（出队）
头指针	栈顶指针	队尾指针、队首指针

习题

- 1. 循环队列的优点是什么?如何判断它的空和满?
- 2. 利用栈的基本操作，写一个返回栈S中结点个数的算法 int StackSize(SeqStack S)
- 3.一个双向栈S是在同一向量空间内实现的两个栈，它们的栈底分别设在向量空间的两端。试为此双向栈设计初始化InitStack(S)，入栈Push(S,i,x)，出栈Pop(S,i,x)算法，其中i为0或1，用以表示栈号

习题

- 4、假设Q[0,5]是一个循环队列，初始状态为front=rear=0，请画出做完下列操作后队列的头尾指针的状态变化情况，若不能入队，请指出其元素，并说明理由。

  d, e, b, g, h入队

  d, e出队

  i , j , k , l , m入队

  b出队

  n, o, p, q, r入队