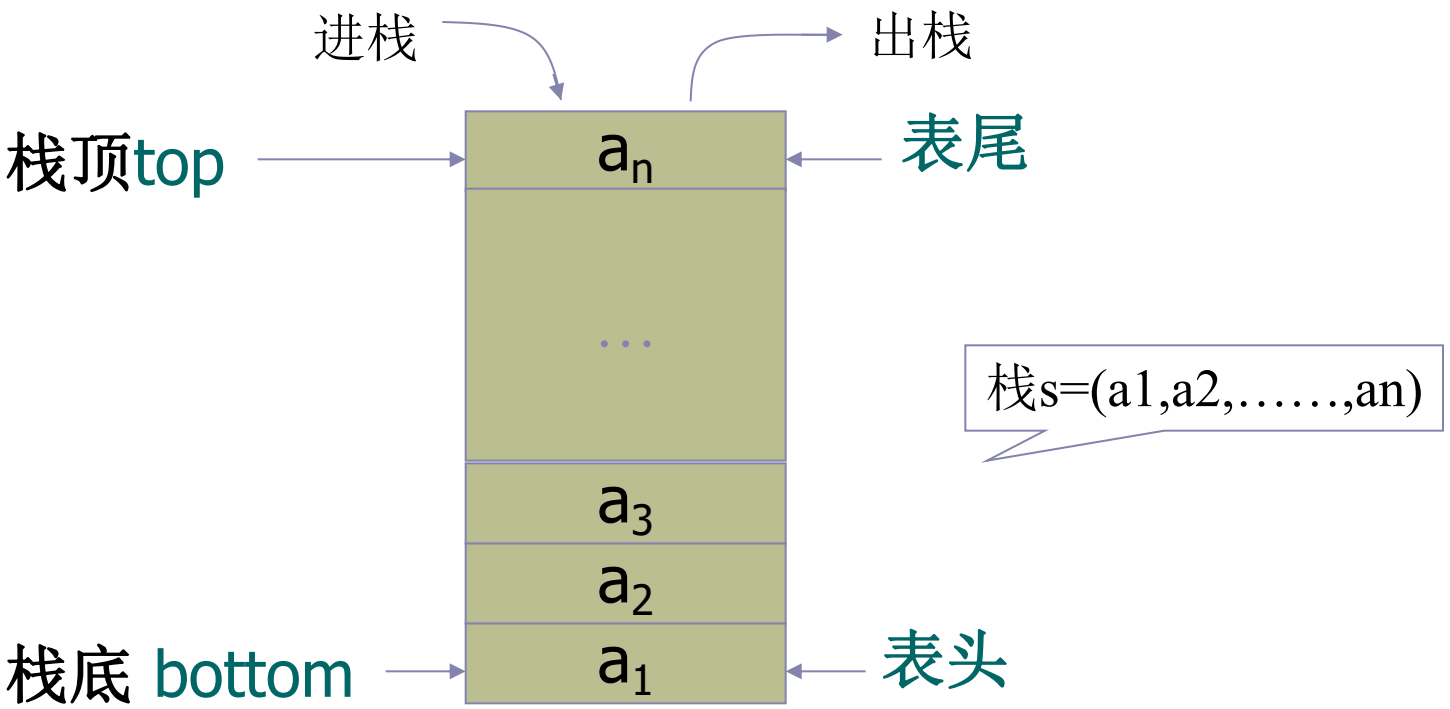
第3章 栈和队列

栈（Statck）类型的定义

- 栈 是操作受限制的线性表
- 定义：仅在表尾进行插入或删除操作的线性表；
- 概念：
  - 栈顶：在栈顶操作，是表尾，通常用top表示 ；
  - 栈底：bottom，是表头；
  - 空栈: 空表；
  - 通常栈底固定，栈顶移动。
- 特点：先进后出（FILO）或后进先出（LIFO）

栈示意图



操作原则：后进先出（**Last In First Out**），**LIFO**

举例：餐馆的盘子

栈的抽象数据类型

ADT Stack{

数据对象： $D=\{a_i \mid a_i \in ElemSet,\ i=1, 2, \ldots, n, n>=0\}$

数据关系： $R=\{<a_{i-1}, a_i> \mid a_{i-1}, a_i \in D, i=2,\ldots,n\}$

基本操作：

栈初始化： StackInit()

判栈空： StackEmpty(S)

入栈： Push(S, x)

出栈： Pop(S)

取栈顶元素： StackGetTop(S)

销毁栈： StackDestroy (S)

清空栈： StackClear (S)

求栈长： StackLength(S)

}ADT Stack

栈的表示和实现

- 顺序存储结构：**顺序栈；**
- 链式存储结构：**链栈；**

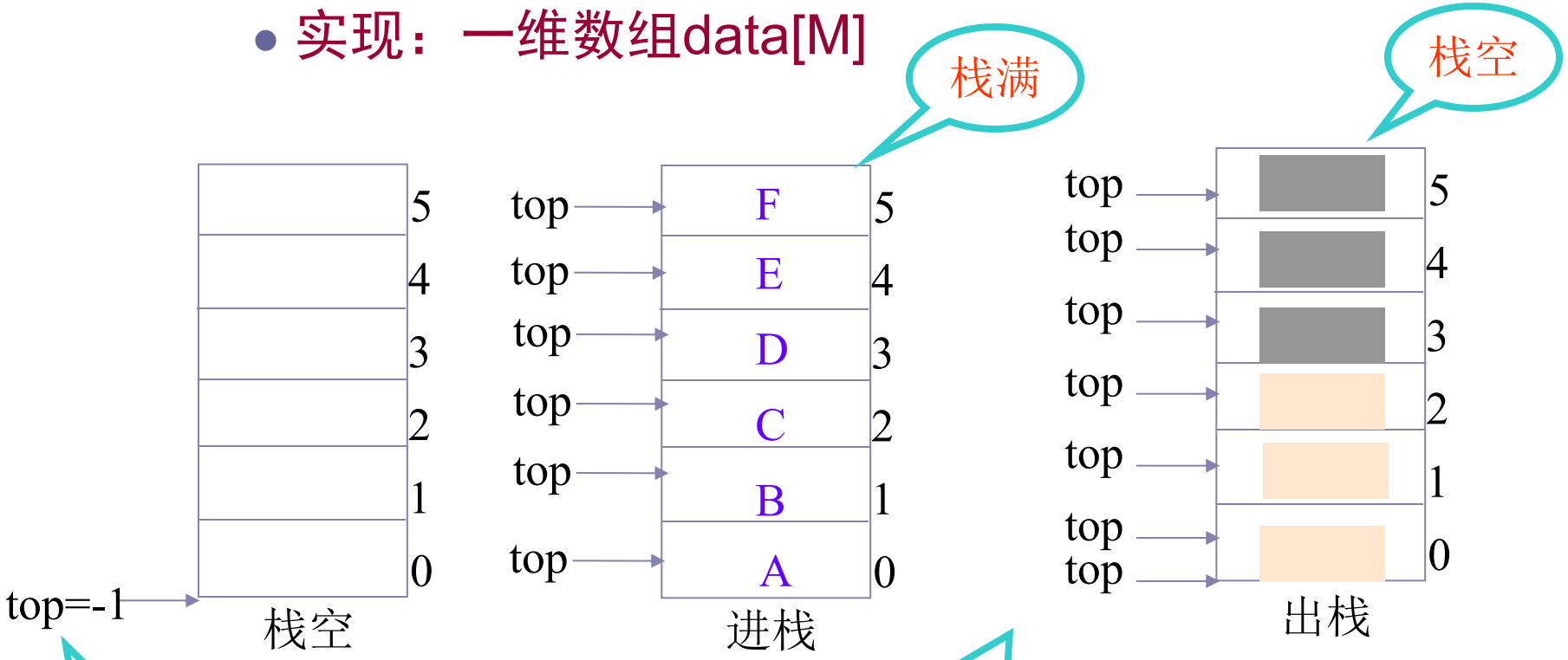
顺序栈

- 利用一组地址连续的存储单元依次自栈底到栈顶存放栈的数据元素.
- 在数组上实现时，栈底位置可以设置在数组的任一个端点，而栈顶是随着插入和删除而变化的，可以用一个整形变量top存放栈顶的指针，数据入栈或出栈时使整形变量 top分别加1或减1。

顺序栈的定义：

```
#define MAXSIZE  1024   /* 栈中能达到的最大容
   量*/
typedef  int  ElemType ;
typedef    struct          /* 顺序栈类型定义 */
{
   ElemType    data[MAXSIZE];
   int       top;              /*指示栈顶位置*/
} SeqStack;
```

- 1 顺序栈
  - 实现：一维数组data[M]



栈顶指针top，指向实际栈顶位置，初值为-1

设数组维数为M
top=-1，栈空，此时出栈，则下溢（underflow）
top=M-1，栈满，此时入栈，则上溢（overflow）

顺序栈基本算法(1)

初始化

```
Void SeqStackInit(SeqStack &S)
 {//构造一个空栈S

  S.top=-1;

 }
```

顺序栈基本算法(2)

判栈空

```
int SeqStackEmpty(SeqStack S)
{//判断栈S是否为空
  if (S.top==-1)
       return 1;
   else  return 0;
 }
```

顺序栈基本算法(3)

入栈

```
Int SeqStackPush(SeqStack &S, ElemType  x)
{‖ 插入元素x为新的栈顶元素
   if (S.top==MAXSIZE-1)
       { printf("栈满\n"); return 0;} ‖ 栈满，退出运行
   S.top++;
   S.data[S.top]=x;
   return 1;
 }
```

顺序栈基本算法(4)

- 出栈(并返回栈顶元素)

```
 int SeqStackPop(SeqStack &S,ElemType  &x)
{‖ 若栈S不空，删除S的栈顶元素，并返回其值
  if(S.top==-1)
     {printf("栈空\n"); return 0;} ‖ 栈已空，退出运行
  x=S.data[S.top];
  S.top--;
  return 1;
}
```

顺序栈基本算法(5)

- 取栈顶元素

```
ElemType SeqStackGetTop(SeqStack S)
{‖  若栈非空返回栈顶元素的值
    return (S.data[S.top]);
}
```

链栈

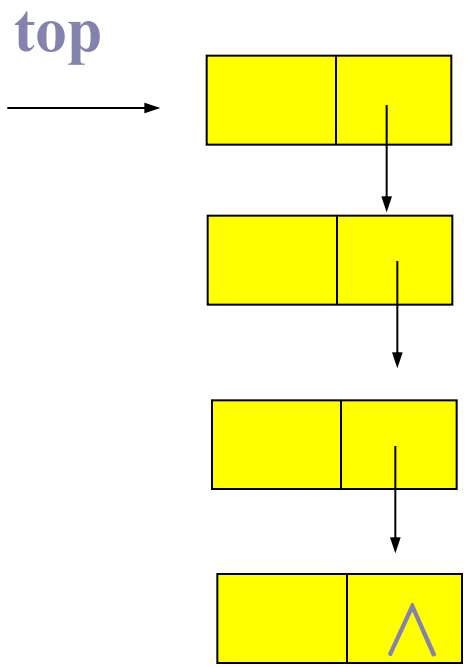
- 用链式存储结构实现的栈称为链栈。
- 通常链栈可用单链表表示，因此其结点结构与单链表的结构相同。

链栈结点的类型描述：

```
typedef int ElemType;
typedef  struct StackNode
{ElemType data;
  struct StackNode *next;
}StackNode，*LinkStack;
```

由于栈只在栈顶操作，因此，通常不设头结点。

链栈图示



链栈基本算法(1)

链栈的初始化

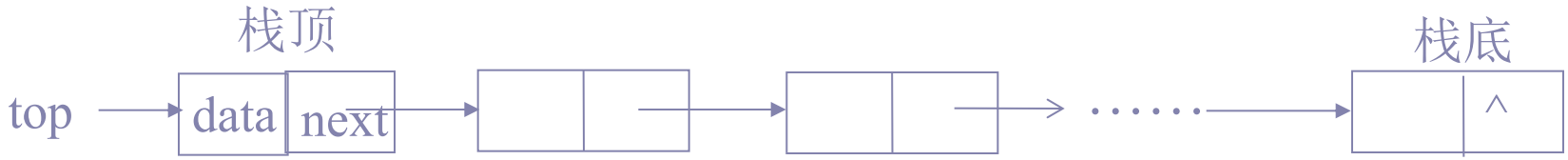
```
Void LinkedStackInit(LinkedStack &top)
{//构造一个空栈, 栈顶指针为top
   top=NULL;

}
```

链栈基本算法(2)

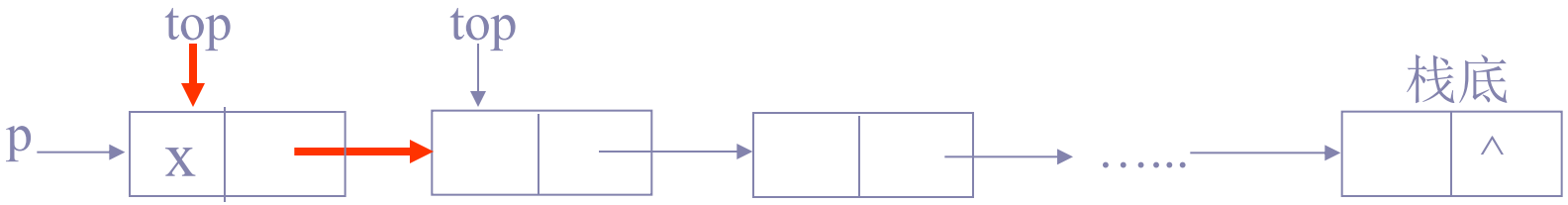
判栈空

```
int  LinkedStackEmpty(LinkedStack  top )
{‖  判定栈S是否是空栈
   if(top==NULL) return 1;
   else  return  0;
}
```

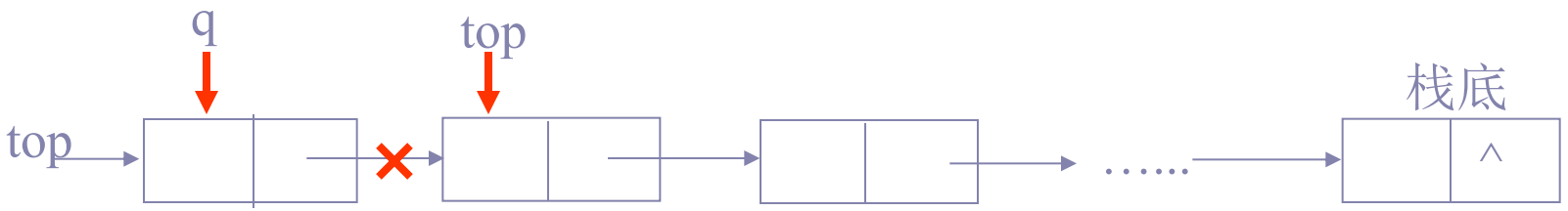
- 链栈



- 入栈算法



- 出栈算法



链栈基本算法(3)

入栈

```
void  LinkedStackPush(LinkedStack  &top , ElemType x)
{//在链栈top中插入元素x，x成为新的栈顶元素
  StackNode *s;
 s=(StackNode*)malloc(sizeof(StackNode));
   ‖ 创建一个新结点
  s->data=x;                    ‖ 设置新结点的数据域
  s->next=top;                  ‖ 设置新结点的指针域
  top=s;                        ‖ 设置新栈顶

}
```

链栈基本算法(4)

退栈

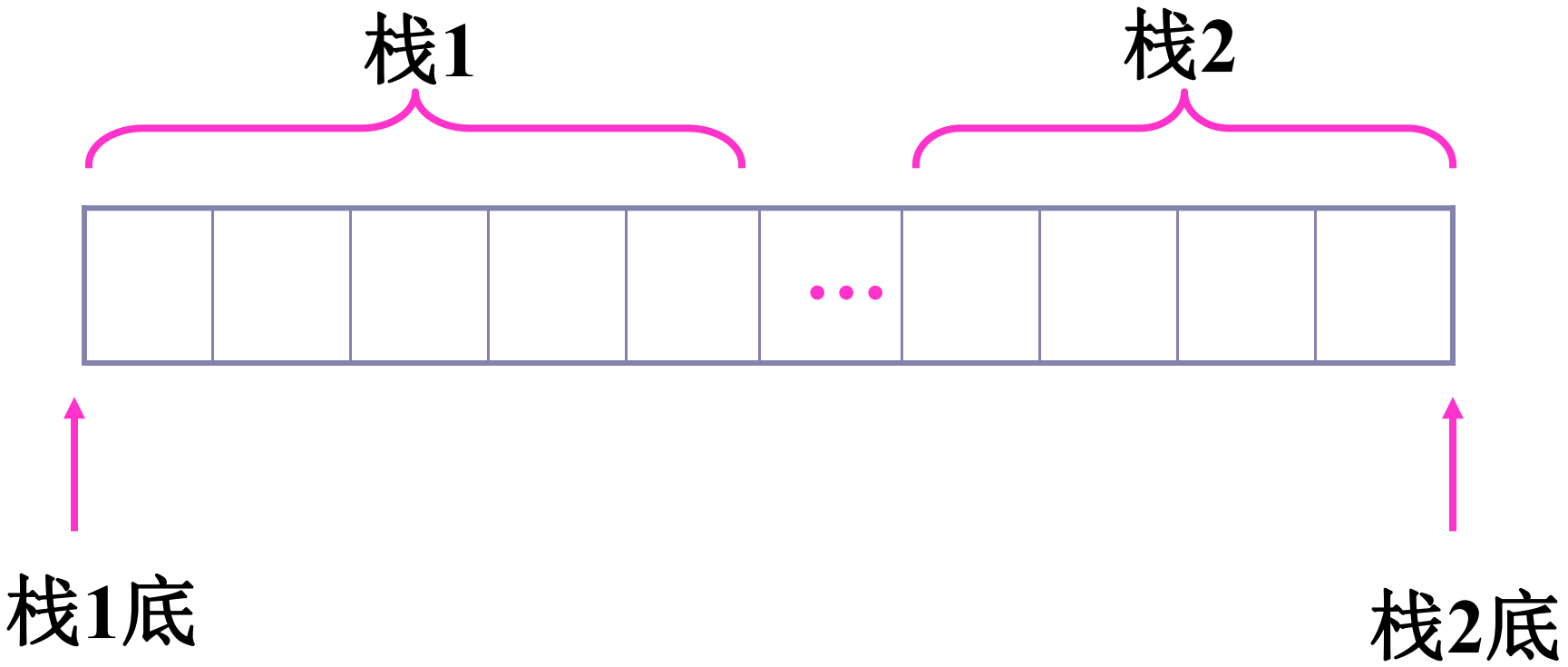
```
ElemType  LinkedStackPop (LinkedStack  &top)
{//若栈S不空，则删除S的栈顶元素，并返回其值
  if（top!=NULL）
     {x=top->data;
      p=top;
      top=top->next;
      free (p);
      return  x;
     }
}
```

顺序栈和链栈的比较

- 顺序栈需预先定义栈的长度，浪费空间；链栈长度可变但需增加结构性开销；
- 程序中用多个顺序栈时，为减少栈的溢出，提高空间利用率，通常用一个数组空间存放多个顺序栈。

双向栈

两个栈共享一个向量空间，栈底分别设在两端，写出进栈和退栈操作



双向栈的类型定义

```
#define  M  100    ‖ 栈容量（向量大小）
typedef int ElemType;
typedef  struct DStack
   {ElemType v[M];
    int top1,top2;
   }DStack；
```

初始化时，top1=-1和top2=M表示栈空，栈顶相遇时表示栈满，入栈时栈顶指针相向移动。

进栈

```
int  push(Dstack s,int i, ElemType x)
/* 两栈共享向量空间,i是1或2，表示两个栈，x是进栈元素
/* 本算法是入栈操作 */
{ if (s.top2 – s.top1==1) return(0);/* 栈满 */
  else {switch (i)
        {case 1: s.v[++(s.top1)]=x; break;
         case 2: s.v[--(s.top2)]=x; break;
         default: printf("栈编号输入错误" );return(0);
         }
        return(1);      /* 入栈成功 */
       }
}
```

退栈

```
ElemType pop(DStack s,int i)
  /* 两栈共享向量空间,i是1或2，表示两个栈，本算法是退栈
     操作 */
{ ElemType x;
 switch (i)
        {case 1: if (s.top1==-1) return(0);/* 栈1空 */
                    else x=s.v[(s.top1)--];break;
         case 2: if (s.top2==M) return(0);/* 栈2空 */
                   else x=s.v[(s.top2)++];break;
         default: printf("栈编号输入错误" );return(0);
         }
        return(x);      /* 退栈成功 */
}
```

取栈顶元素

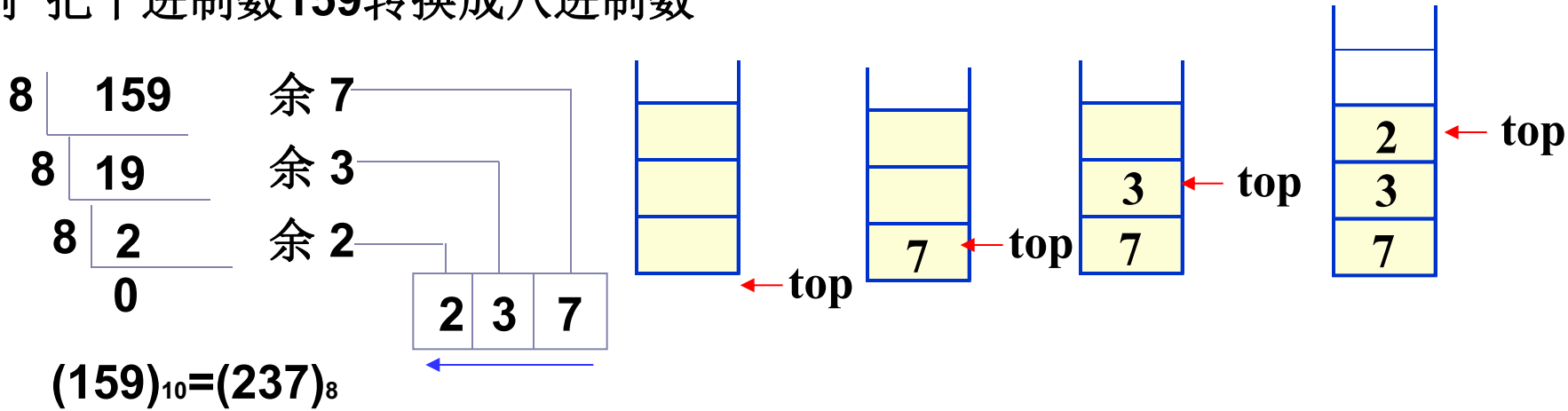
```
ElemType top (DStack s,int i)
  /* 两栈共享向量空间,i是1或2，表示两个栈，本算法是取栈
     顶元素操作 */
{ ElemType x;
  switch (i)
   {case 1: if (s.top1==-1) return(0);/* 栈空 */
                x=s.v[s.top1]; break;
    case 2: if (s.top2==m) return(0);/* 栈空 */
                x=s.v[s.top2];break;
    default: printf("栈编号输入错误" );return(0);
   }
   return(x);      /* 取栈顶元素成功 */
 }
```

栈的应用

- 数制转换
- 括号匹配
- 栈与递归过程

多进制输出：

例 把十进制数159转换成八进制数



$(159)_{10}=(237)_8$

N	N/8	N%8
159	19	7
19	2	3
2	0	2

数制转换

```
void convert(int N, int d)
{‖ 将非负十进制整数N转为d进制整数
 StackInit(S);       ‖ 栈初始化
  while(N!=0)
      {Push(S，N%d); ‖ 将得到的八进制数位入栈
       N=N/d;        ‖ 数N除以8作新的被除数
       }‖ while
   while(!StackEmpty(S))
      {e=Pop(S);
       printf(“%d”，e);‖ 输出d进制数      }‖ while
  }
```

表达式中括号匹配的检查

假设在一个算术表达式中，可以包含三种括号：圆括号“（”和“）”，方括号“[”和“]”和花括号“{”和“}”，并且这三种括号可以按任意的次序嵌套使用。比如，...[...{...}...[...]...]...[...]...(...)...。现在需要设计一个算法，用来检验在输入的算术表达式中所使用括号的合法性。

算术表达式中各种括号的使用规则为：出现左括号，必有相应的右括号与之匹配，并且每对括号之间可以嵌套，但不能出现交叉情况。我们可以利用一个栈结构保存每个出现的左括号，当遇到右括号时，从栈中弹出左括号，检验匹配情况。

在检验过程中，若遇到以下几种情况之一，就可以得出括号不匹配的结论:

- （1）当遇到某一个右括号时，栈已空，说明到目前为止，右括号多于左括号；
- （2）从栈中弹出的左括号与当前检验的右括号类型不同，说明出现了括号交叉情况；
- （3）算术表达式输入完毕，但栈中还有没有匹配的左括号，说明左括号多于右括号。

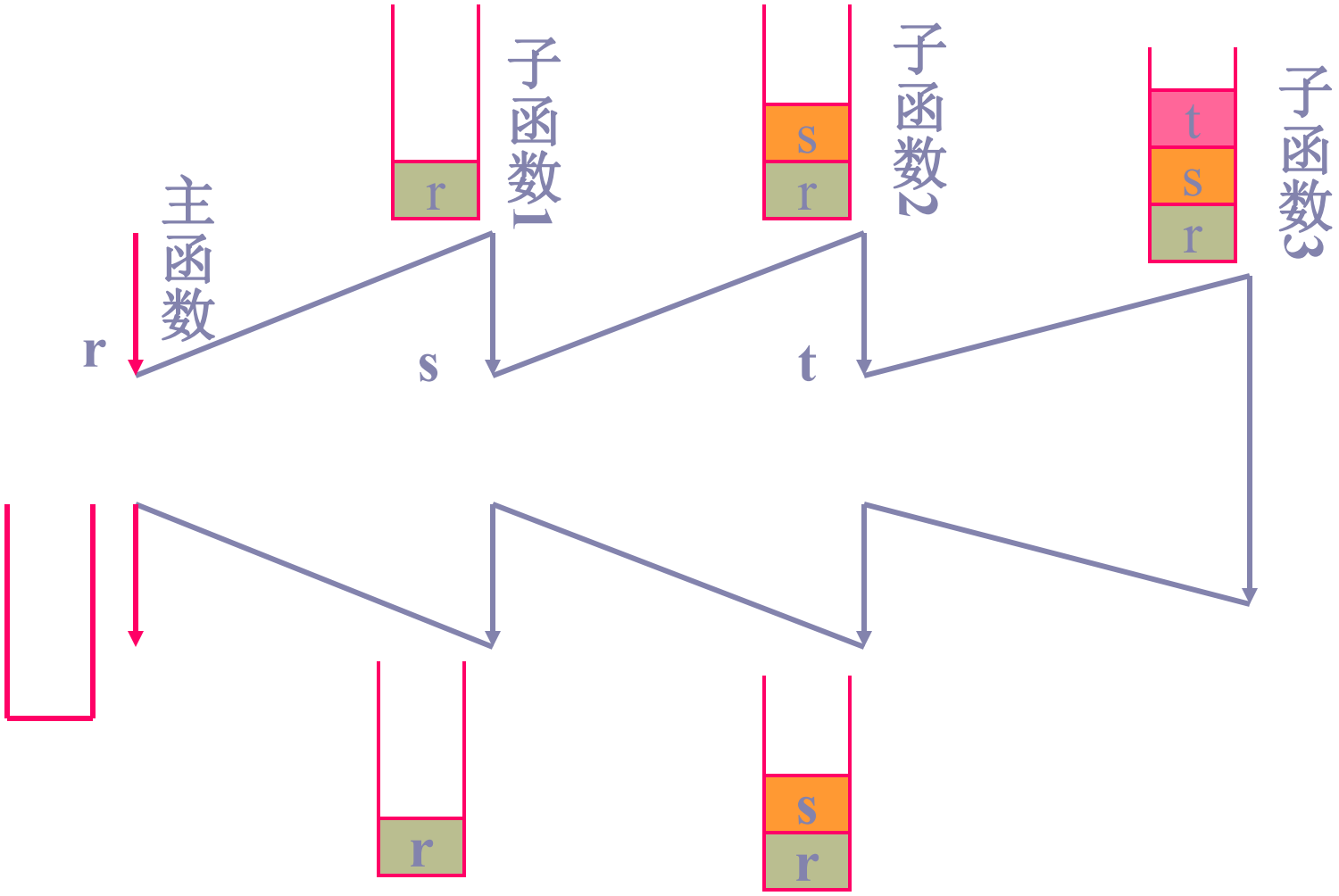
```
typedef char DataType;
int Check( )
{StackInit(s); char ch;
  while ((ch=getchar())!='\n')
 {//以字符序列的形式输入表达式
    switch (ch) {
    //遇左括号入栈
    case (ch=='('||ch== '['||ch== '{'):
          push(s,ch);break;
     //在遇到右括号时，分别检测匹配情况
    case (ch== ')'):
          if (StackEmpty(s))   retrun FALSE;
          else {pop(s,c);
                if (c!= '(') return FALSE; }
         break;
```

```
case (ch== ']'):
            if (StackEmpty(s))
                    retrun FALSE;
            else {pop(s,c);
                  if (c!= '[') return FALSE; }
           break;
  case (ch== '}'):
            if (StackEmpty(s))
                    retrun FALSE;
            else {pop(s,c);
                  if (c!= '{') return FALSE; }
           break;
  default:break;}
 }
 if (StackEmpty(s)) return TRUE;
  else return FALSE;
}
```

栈与递归过程

- 递归：若在一个函数、过程或者数据结构定义的内部，直接（或间接）出现定义本身的应用，则称它们是递归的，或者是递归定义的。
- 递归过程的应用：
  - 问题的定义是递归的：f(n)=n*f(n-1)
  - 数据结构是递归的：链表、二叉树
  - 问题的解法是递归的：Hanoi 塔问题
- “递归工作栈”——栈顶为“工作记录”，包括参数、局部变量以及上一层的返回地址

- 函数的嵌套调用演示

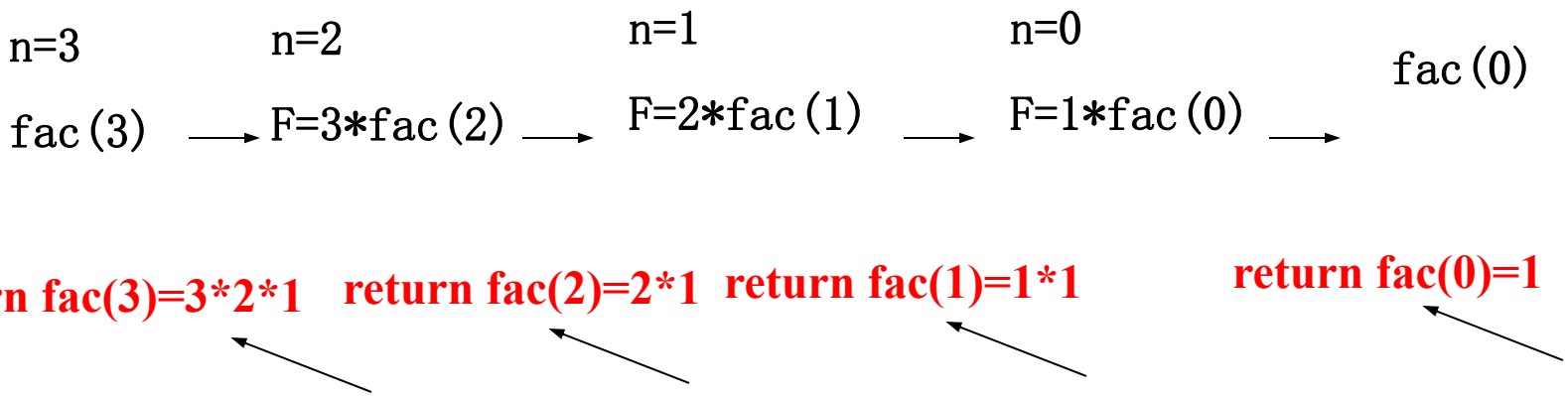


递归过程的应用（1）

（1）问题的定义是递归的：求n的阶乘n!

```
long  Fact(int n)
{
  if(n==0) return(1);
 else return(n*Fact(n-1));
}
```

求阶乘(n!)过程的模拟


n=3
fac(3)
n=2
F=3*fac(2)
n=1
F=2*fac(1)
n=0
F=1*fac(0)
fac(0)
return fac(3)=3*2*1
return fac(2)=2*1
return fac(1)=1*1
return fac(0)=1

递归过程的应用（2）

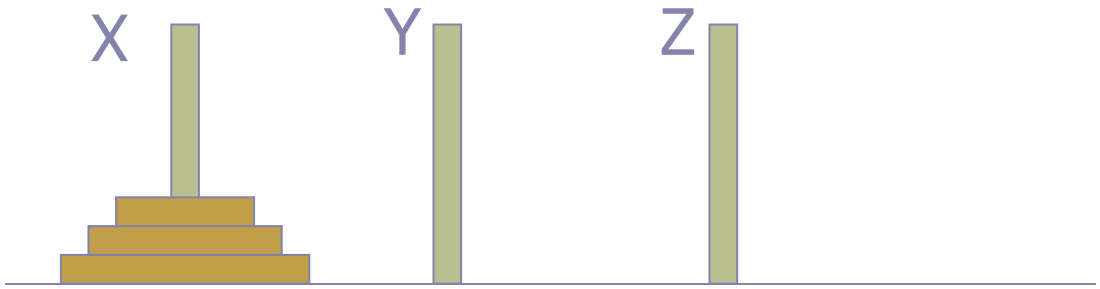
（2）数据结构是递归的：

逆序打印链表中各结点的值。

```
void print(LinkedList head)
    {if (head!=NULL)
        {print(head->next);
          printf("%d",head->data); //设元素为整型
        }
    }
```

递归过程的应用（3）
-----问题的解法是递归的

- n阶Hanoi塔问题：假设有三个分别命名为X,Y,Z的塔座，在X塔座上插有n个直径大小各不相同，依小到大编号为1，2，...,n的圆盘，要求：把X上的n个圆盘移到Z上，排列顺序相同，移动规则为：
  - 每次只能移动一个圆盘；
  - 圆盘可以在任一塔上做多次移动；
  - 在任何时刻，大盘不能压在小盘的上面。



栈与递归的实现:Hanoi

- 数学归纳法
  - n = 1, OK;
  - 设n = k 时，
    - 若可以以Y为辅助塔，把k个盘从X移动到Z;
  - 当n = k + 1时，方法：
    - 把X中k个盘，以Z为辅助塔，移动到Y;
    - 把X中第k+1个盘，移动到Z;
    - 把Y中k个盘，以X为辅助塔，移动到Z;

栈与递归的实现:Hanoi

```
void Hanoi( int n, char x, char y, char z )
{   if( n == 1 )
        move( x, 1, z );        // 把1号盘，从x移到z
    else
        {Hanoi( n – 1, x, z, y );// 把n-1个盘从x移到y，z为辅助塔
        move( x, n, z );        // 把n号盘，从x移到z
        Hanoi( n – 1, y, x, z );//把n-1个盘从y移到z，x为辅助塔
        }
}
```

队列(Queue)定义和概念

- 队列：队列是一种只允许在表的一端插入，在另一端删除的存取受限的线性表。
- 概念：
  - 队尾rear： 插入端，线性表的表尾。
  - 队头front：删除端，线性表的表头。
  - 当队列中没有任何元素时，称为空队列。

队列(Queue)图示

- FIFO（First In First Out）（先进先出表）

队列的抽象数据类型

ADT Queue{

数据对象：$D=\{a_i \mid a_i \in ElemSet，\ i=1，2，...，n，n>=0\}$

数据关系：$R=\{<a_{i-1}，a_i>\mid a_{i-1}，a_i \in D，\ i=2,...,n\}$

基本操作：

队列初始化： QueueInit ()

入队： QueueIn (Q , x),

出队： QueueOut (Q)

读队头元素： QueueGetHead(Q)

判队空： QueueEmpty (Q)

}ADT Queue

队列的表示和实现

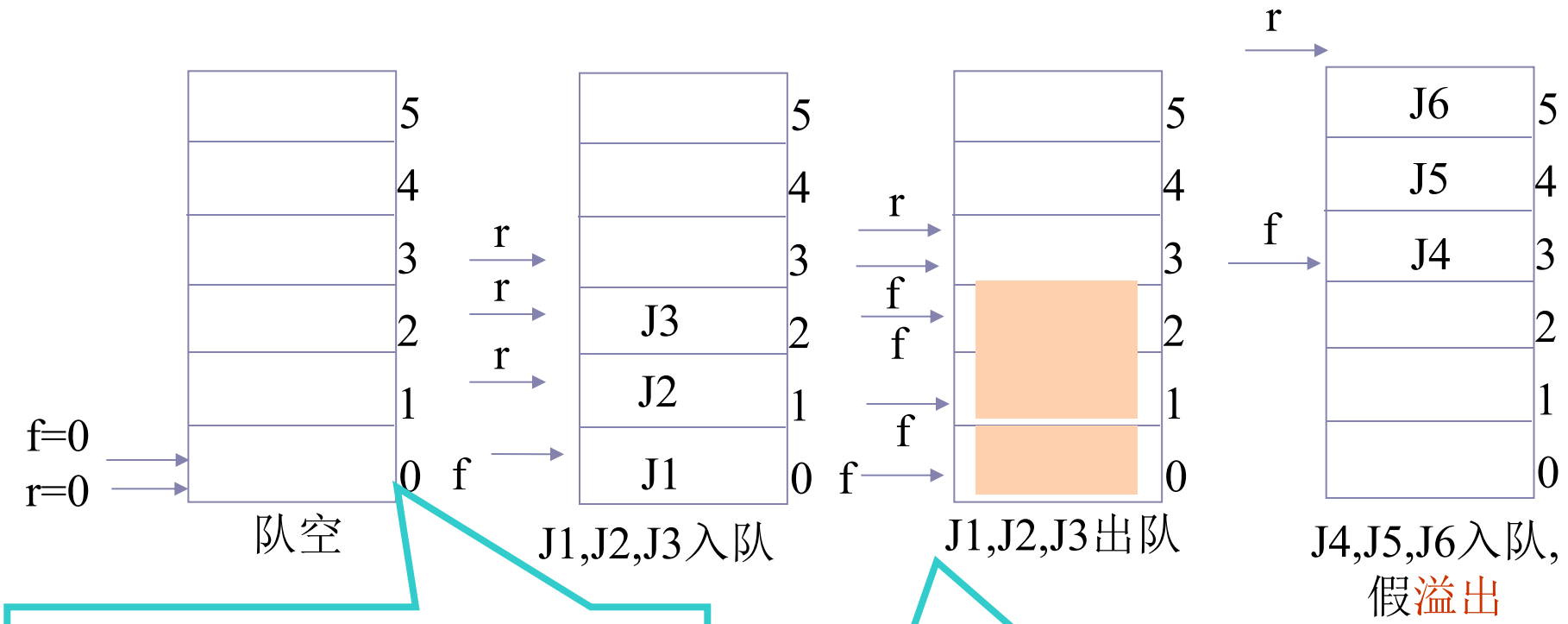
- 顺序存储结构
- 链式存储结构

- 队列的顺序存储结构

实现：用一维数组实现:

```
# define MAXSIZE 100 /*队列中最大元素个数*/
typedef int ElemType;
typedef struct     /*顺序队列类型定义*/
{
      ElemType data[MAXSIZE];
      int front,rear;
} SeQueue;
```

普通队列



设两个指针:

Q.f, Q.r,约定:

Q .r指示队尾元素后一位置;

Q.f指示队头元素;

初值Q.f = Q .r=0

空队列条件： Q .f == Q.r

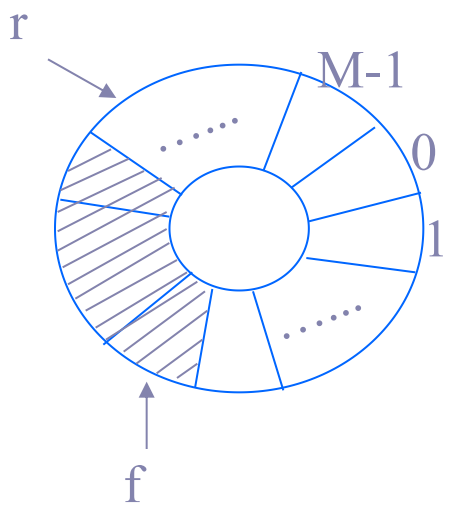
入队列： Q .data[Q .r]=x; Q .r++;

出队列： x=Q .data[Q.f]; Q.f++;

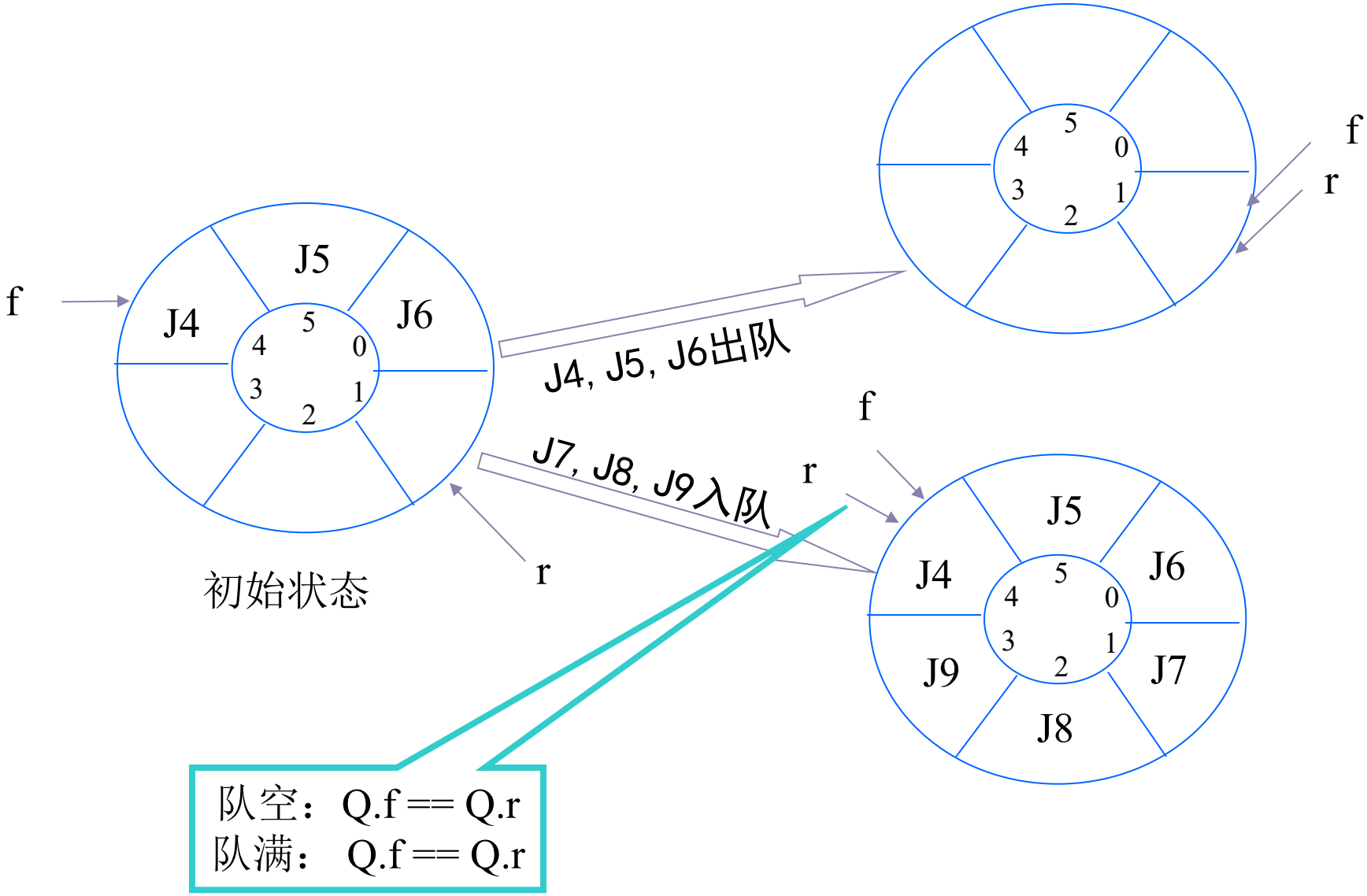
- 存在问题

  设数组维数为M，则：

  - 当f=0,r=M时，再有元素入队发生溢出——真溢出
  - 当f≠0,rear=M时，再有元素入队发生溢出——假溢出
- 解决方案
  - 队首固定，每次出队剩余元素向下移动——浪费时间
  - 循环队列
    - 基本思想：把队列设想成环形，让q[0]接在q[M-1]之后，若r==M,则令r=0;



- 实现：利用“模”运算
- 入队： q[r]=x; r=(r+1)%M;
- 出队： x=q[f]; f=(f+1)%M;
- 队满、队空判定条件


f
J5
J4
J6
5
4
0
3
1
2
初始状态
r
J4, J5, J6出队
5
4
0
3
1
2
f
r
J7, J8, J9入队
f
r
J5
J4
J6
5
4
0
3
1
2
J9
J7
J8
队空：Q.f == Q.r
队满： Q.f == Q.r

循环队列

- 空队列条件：
  - Q.rear == Q.front;
- 满队列条件：
  - Q.rear == Q.front;

问题：如何区别队空和队满

有三种方法
1. 少用一个存储空间
2. 引入一个标志变量区别空和满
3. 使用计数器

循环队列类型定义

循环队列的类型定义如下：

```
#define MAXSIZE 100    // 队列可能达到的最大长度
typedef int ElemType;
typedef struct
   {ElemType data[MAXSIZE];
    int front,rear;
    }SeQueue;
```

循环队列的初始化

```
Void  SeQueueInit(SeQueue Q)
{// 初始化队列Q
Q.front=0;
Q.rear=0;
}
```

循环队列的入队

```
int  SeQueueIn(SeQueue &Q , ElemType  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 0;            ‖ 队列满，返回
    }
    Q.data[Q.rear]=x;    ‖ 入队
    Q.rear=(Q.rear+1) % MAXSIZE;
     return 1;
 }
```

循环队列的取队头和出队

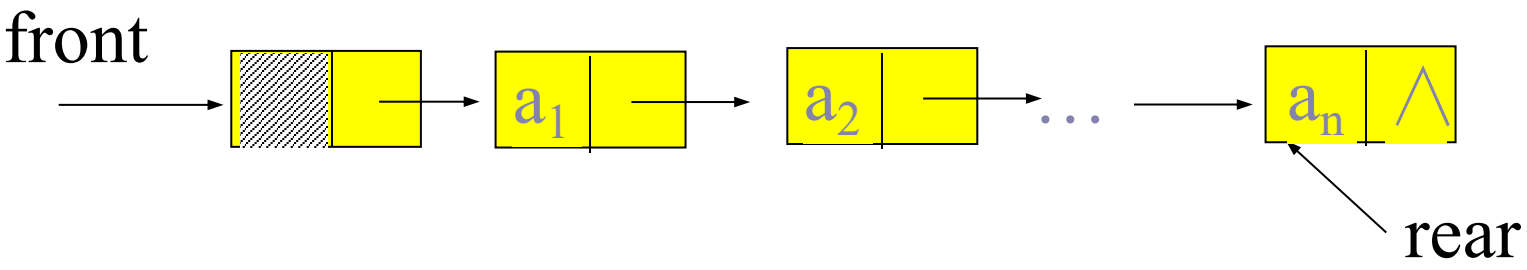
```
ElemType  GetHead(SeQueue Q) //读出队头元素
{ return Q.data[Q.front];}

int SeQueueOut (SeQueue &Q,ElemType &x)
{if (Q.front==Q.rear)
    {printf("Empty\n");  return 0;   //队已空,退出运行
    }
x=Q.data[Q.front];
  Q.front=( Q.front+1) % MAXSIZE;
  return 1;
}
```

循环队列的判空

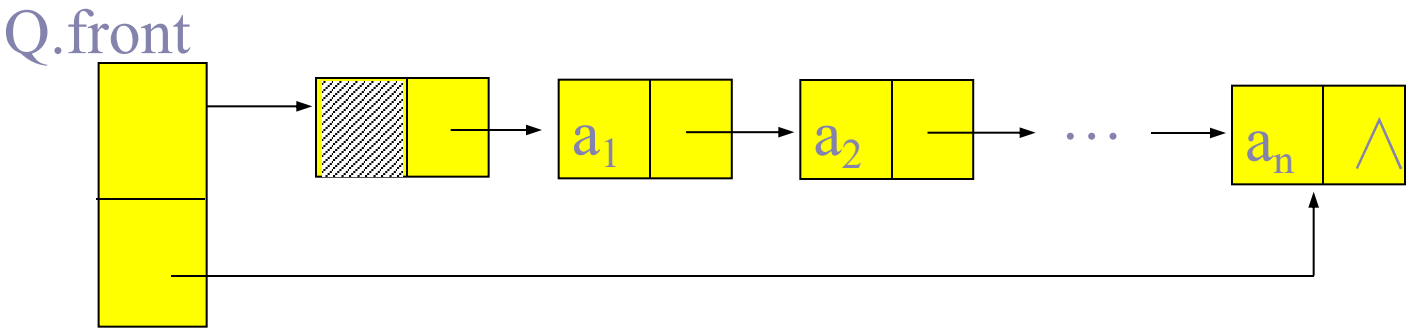
```
int SeQueueEmpty (SeQueue  Q)
{
  if (Q.front==Q.rear) return 1;
 else  return 0;
}
```

链队列

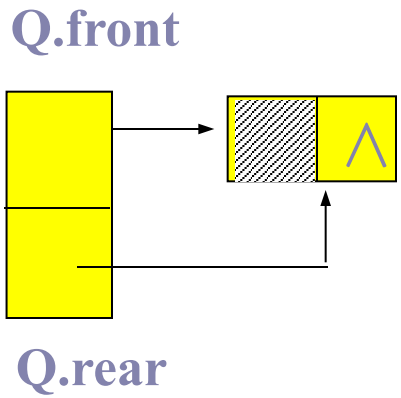
- 用链表表示的队列简称为链队列。
- 一个链队列显然需要分别指示队头和队尾的两个指针。



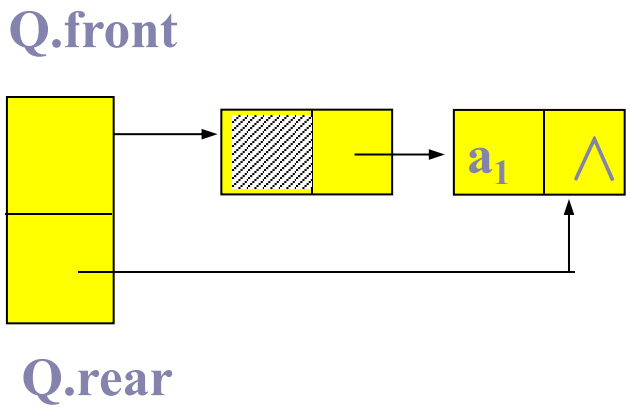
链队示意图



(a)非空队



(b) 空队



(c) 链队中只有一个元素结点

链队列的类型描述

```
typedef struct LQNode
 { ElemType  data;
   struct  LQNode *next;
 }LQNode,*LinkQNode; //链队结点的类型及指针

typedef struct
{ LinkQNode front,rear;
} LinkQueue;  //将头尾指针封装在一起的链队
```

链队列的初始化

创建一个带头结点的空队

```
int  LinkedQueueInit(LinkQueue &Q)
{
Q.front=Q.rear=(LinkedQNode)malloc(sizeof(LQNode));
 if(!Q.front)  {printf("error"); return 0;}
 Q.front->next=NULL;
 return 1;
}
```

链队列的入队

```
void LinkedQueueIn( LinkQueue &Q, ElemType  x)
{LinkQNode p;
  p=(LinkQNode)malloc(sizeof(LQNode));
  p->data=x;
  p->next=NULL;
  Q.rear->next=p;
  Q.rear=p;
}
```

链队列的判空

判队空

```
int LinkedQueueEmpty(LinkedQueue Q)
{if (Q.front==Q.rear)
    return 1;
 else  return 0;
 }
```

链队列的出队

出队

```
ElemType LinkedQueueOut(LinkQueue  Q)
{if(Q.front!=Q.rear)
  {p=Q.front->next;  //队列带头结点
   x=p->data;        //队头元素放x中
   Q.front->next=p->next;
   if (Q.rear==p) Q.rear=Q.front;
      //只有一个元素时，出队后队空，此时还要修改队尾指针
   free(p);
   return x;
   }
}
```

假设以带头结点的循环链表表示队列，只设一个指向队尾结点的指针，试设计相应的入队列和出队列的算法

```
typedef struct Lqueue
{ElemType data;
 struct Lqueue *next;
}Lqueue,*Queueptr;
```

```
void QueueIn(Queueptr rear, ElemType x)
{//入队列
 s=(Queueptr)malloc(sizeof(LQueue));
 s->data=x;
 s->next=rear->next;      //元素插入队尾
 rear->next=s;rear=s;      //求得新的队尾
}
```

（续） 出队列的算法

```
ElemType QueueOut(Queueptr rear)
{//出队列
if(rear->next==rear)
  printf("队列为空");
else {p=rear->next; q=p->next;
      p->next=q->next;x=q->data;
          //删除队头元素
      if(q==rear) rear=p;
      free(q);
      return(x);
     }
}//QueueOut
```

如果希望循环队列中的元素都能得到利用，则可以设置一个标志域tag，并以tag的值是0或 1 来区分尾指针和头指针相同时的队列状态是“空”还是“不空”。编写相应的入队列和出队列的算法。

类型定义：

```
typedef struct
 {ElemType Q[m] ;
  int  rear,front;  //队尾和队头指针
  int tag;    //标记, 0为空,1为非空
 } CycQueue;
```

只设标志的循环队列的入队

```
CycQueue  QueueIn (CycQueue cq, ElemType x)
{ if(cq.tag==1 && cq.front==cq.rear)
    {printf("队满    "); exit(0);    }
  else {cq.rear=(cq.rear+1) % m;
        cq.Q[cq.rear]=x;
        if (cq.tag==0) cq.tag=1;
            //由空变不空标记
       }
  return cq;
 }
```

只设标志的循环队列的出队

```
CycQueue  QueueOut(CycQueue cq)
  {if (cq.tag==0)
       {printf("队空" ); exit(0); }
    else {cq.front=(cq.front+1) % m;
          if (cq.front==cq.rear)
          cq.tag=0;    //队列由不空变空
         }
  return cq;
  }
```