

# 函数调用过程

## 调用前, 系统完成:

- (1) 将**实参, 返回地址**等传递给被调用函数
- (2) 为被调用函数的**局部变量**分配存储区
- (3) 将控制转移到被调用函数的**入口**

## 调用后, 系统完成:

- (1) 保存被调用函数的计算**结果**
- (2) 释放被调用函数的**数据区**
- (3) 依照被调用函数保存的**返回地址**将控制转移到调用函数

# 递归函数调用的实现

“层次”

主函数

0层

第1次调用

1层

第  $i$  次调用

$i$  层

“递归工作栈”

“工作记录”  实在参数,局部变量,返回地址

# 递归算法的效率分析

空间效率

与递归树的深度成正比

$O(n)$

时间效率

与递归树的结点数成正比

$O(2^n)$

# 递归算法的效率分析

1	2	3	4
$f(1)=1$	$f(1)+1+f(1)=3$	$f(2)+1+f(2)=7$	$f(3)+1+f(3)=15$

$$\begin{aligned}f(n) &= 2f(n-1)+1 \\f(n-1) &= 2f(n-2)+1 \\f(n-2) &= 2f(n-3)+1 \\&\dots\dots \\f(3) &= 2f(2)+1 \\f(2) &= 2f(1)+1\end{aligned}$$

$$\begin{aligned}2^0f(n) &= 2^1f(n-1)+2^0 \\2^1f(n-1) &= 2^2f(n-2)+2^1 \\2^2f(n-2) &= 2^3f(n-3)+2^2 \\&\dots\dots \\2^{n-3}f(3) &= 2^{n-2}f(2)+ 2^{n-3} \\2^{n-2}f(2) &= 2^{n-1}f(1)+ 2^{n-2}\end{aligned}$$

$$f(n) = 2^0+2^1+\dots+2^{n-2}+ 2^{n-1}f(1) = 2^n-1$$



**64片金片移动次数：**  $2^{64}-1=18446744073709551615$

假如每秒钟一次，共需多长时间呢？

一年大约有**31536926**秒，移完这些金片需要 **5 8 0 0** 多亿年

世界、梵塔、庙宇和众生都已经灰飞烟灭 .....

# 递归的优缺点

**优点：结构清晰，程序易读**

**缺点：每次调用要生成工作记录，保存状态信息，入栈；返回时要出栈，恢复状态信息。时间开销大。**

**递归→非递归**

# 递归→非递归

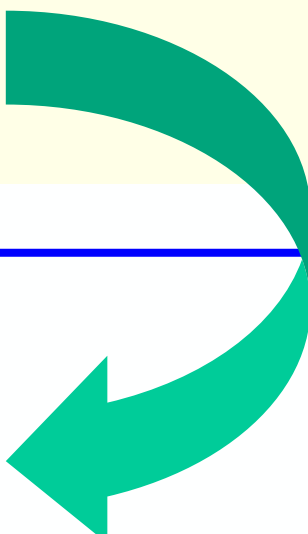
(1) 尾递归、单向递归→**循环结构**

(2) 自用**栈**模拟系统的运行时栈

## 尾递归→循环结构

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

```
long Fact ( long n ) {  
    t=1;  
    for(i=1; i<=n; i++)    t=t*i;  
    return t; }
```





## 单向递归→循环结构

虽然有一处以上的递归调用语句，但各次递归调用语句的参数**只和主调函数**有关，相互之间参数无关，并且这些**递归调用语句处于算法的最后**。

```
long Fib ( long n ) { // Fibonacci数列  
    if(n==1 || n==2) return 1;  
    else return Fib (n-1)+ Fib (n-2);}
```

$$Fib(n) = \begin{cases} 1 & \text{若 } n = 1 \text{ 或 } 2 \\ Fib(n-1) + Fib(n-2) & \text{其它} \end{cases}$$

## 尾递归、单向递归→循环结构

```
long Fib ( long n ) {  
    if(n==1 || n==2) return 1;  
    else return Fib (n-1)+ Fib (n-2);}
```

```
long Fib ( long n ) {  
    if(n==1 || n==2) return 1;  
    else{  
        t1=1; t2=1;  
        for(i=3; i<=n; i++){  
            t3=t1+t2;  
            t1=t2; t2=t3;    }  
        return t3;    }}
```



## 借助栈改写递归（了解）

- (1) 设置一个工作栈存放递归工作记录（包括实参、返回地址及局部变量等）。**
- (2) 进入非递归调用入口（即被调用程序开始处）将调用程序传来的实在参数和返回地址入栈（递归程序不可以作为主程序，因而可认为初始是被某个调用程序调用）。**
- (3) 进入递归调用入口：当不满足递归结束条件时，逐层递归，将实参、返回地址及局部变量入栈，这一过程可用循环语句来实现——模拟递归分解的过程。**
- (4) 递归结束条件满足，将到达递归出口的给定常数作为当前的函数值。**
- (5) 返回处理：在栈不空的情况下，反复退出栈顶记录，根据记录中的返回地址进行题意规定的操作，即逐层计算当前函数值，直至栈空为止——模拟递归求值过程。**

# 递归巩固练习1

输入一个整数，输出对应的2进制形式

```
void conversion(int n)
```

```
{
```

```
    if(n==0)    return ;
```

```
    else
```

```
{
```

```
    conversion(n/2);
```

```
    cout<<n%2;
```

```
}
```

```
if(n>0)
```

```
{conversion(n/2);
```

```
  cout<<n%2;}
```

```
}
```

```
void main()
```

```
{
```

```
    int n;    cin>>n;
```

```
    conversion(n); cout<<endl;}
```

```
void conversion(int n)  
{    if(n==1)    cout<<n%2;  
    else  
    {    conversion(n/2);  
        cout<<n%2;    }  
}
```

```
if(n>0)  
    {    conversion(n/2);  
        cout<<n%2;}
```



```
if(n==0) return ;  
else  
    {    n=n/2;    conversion(n);  
      cout<<n%2;  }
```



```
if(n==0)    return ;  
else {  
    int i=n%2; conversion(n/2);  
    cout<<i;    }
```



## 递归巩固练习2

### 组合问题

找出从自然数 1、  
2、.....、 $m$  中任取  $k$   
个数的所有组合。

例如  $m=5$ ,  $k=3$

5	4	3
5	4	2
5	4	1
5	3	2
5	3	1
5	2	1
4	3	2
4	3	1
4	2	1
3	2	1

## 递归思想:

- ✓ 设函数 `comb(int m,int k)` 为找出从自然数1、2、.....、 $m$  中任取 $k$ 个数的所有组合。
- ✓ 当组合的第一个数字选定时，其后的数字是从余下的 $m-1$ 个数中取 $k-1$ 数的组合。
- ✓ 这就将求 $m$ 个数中取 $k$ 个数的组合问题转化成求 $m-1$ 个数中取 $k-1$ 个数的组合问题。

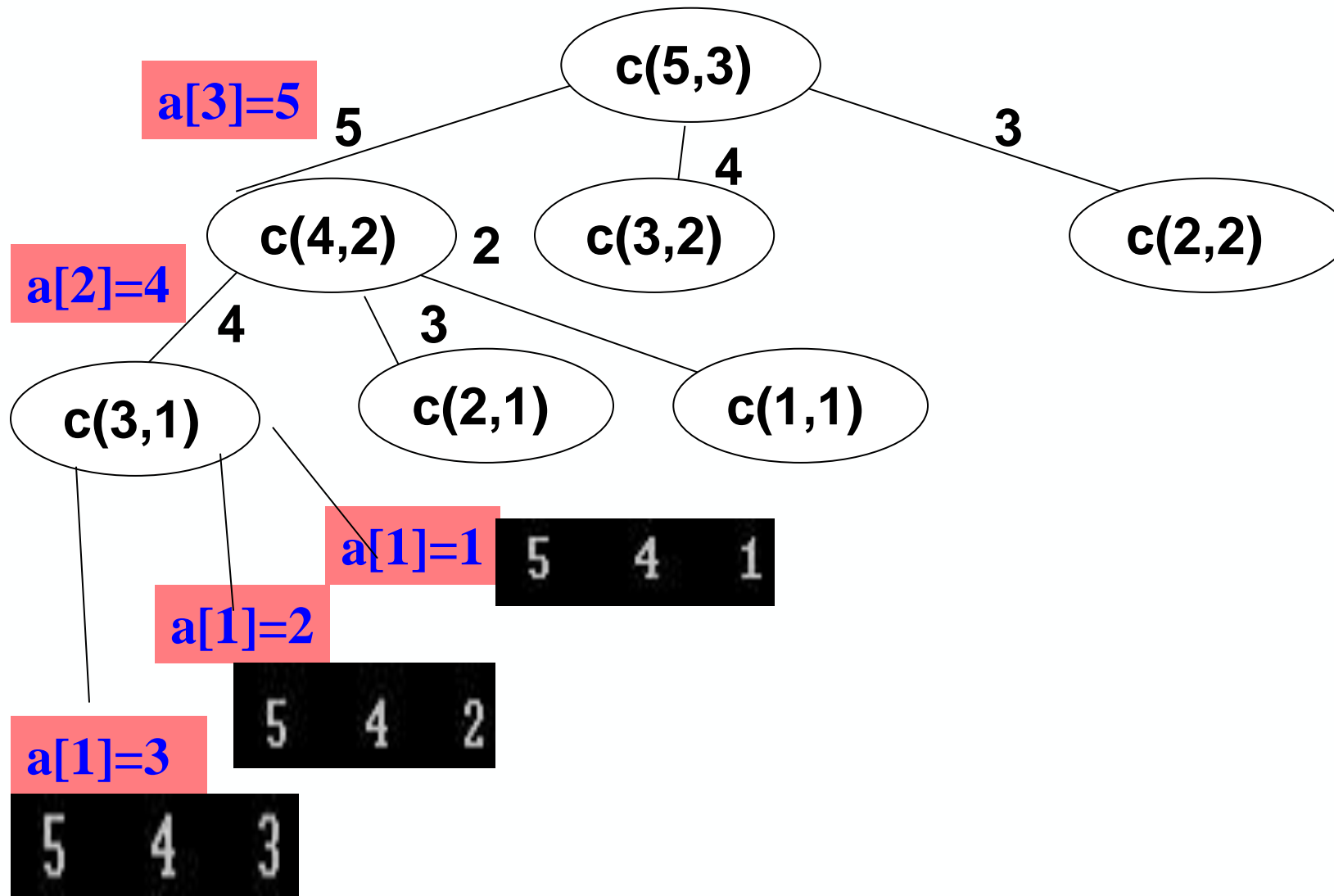


- ✓ 设数组a[ ]存放求出的组合的数字，将确定的k个数字组合的第一个数字放在a[k]中
- ✓ 当一个组合求出后，才将a[ ]中的一个组合输出
- ✓ 第一个数可以是m、m-1、.....、k
- ✓ 函数将确定组合的第一个数字放入数组后，有两种可能的选择
  - 还未确定组合的其余元素，继续递归
  - 已确定组合的全部元素，输出这个组合

# //一般的递归算法

```
#include <stdio.h>
# define    MAXN    100
int    a[MAXN];
void    comb(int m,int k)
{
    int i,j;
    for (i=m;i>=k;i--)
    {
        a[k]=i;
        if (k>1)    comb(i-1,k-1);
        else {
            for (j=a[0];j>0;j--)
                printf("%4d",a[j]);
            printf("\n");
        }
    }
}

void main( )
{
    a[0]=3;    // 用来表示k
    comb(5,a[0]); }
```



## //简单的枚举算法:

```
#include <stdio.h>
void main( )
{ int n,x,y,z,s=0;
  scanf("%d",&n);
  for (x=1;x<=n; x++)
    for (y=1; y<=n; y++)
      for (z=1; z<=n; z++)
        if ( x<y && y<z )
        { s++;
          printf("%5d%5d%5d\n",x,y,z); }
  printf("s=%d\n",s);
}
```

## //加速的枚举算法:

```
#include <stdio.h>
```

```
void main( )
```

```
{   int n,x,y,z,s=0;
```

```
    scanf("%d",&n);
```

```
    for (x=1;x<=n-2; x++)
```

```
        for (y=x+1; y<=n-1; y++)
```

```
            for (z=y+1; z<=n; z++)
```

```
                {   s++;
```

```
                    printf("%5d%5d%5d\n",x,y,z); }
```

```
                printf("s=%d\n",s);
```

```
}
```

## 递归巩固练习3

输出一个正整数 $n$ 的所有整数和形式。如 $n=4$

```
4
3 1
2 2
2 1 1
1 1 1 1
1 1 1 1 1
1 1 1 1 1 1
s=8
```

//源程序1:

```
#include <stdio.h>
```

```
int s=0, a[10]={0};
```

```
void f(int n ,int k)
```

```
{    int i;
```

```
    if (n>0) for(i=n; i>=1; i--)
```

```
        { a[k]=i; f(n-i,k+1) ;}
```

```
    else { for (i=0; i<k; i++) printf("%5d",a[i]);  
          printf("\n");      s++;  }
```

```
}
```

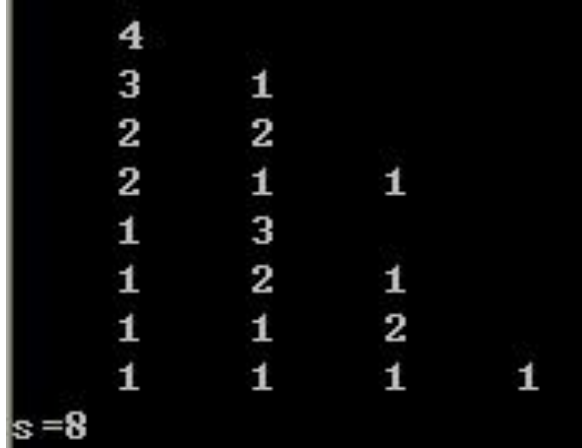
```
void main( )
```

```
{    int n;
```

```
    scanf("%d",&n);    f(n, 0);
```

```
    printf("s=%d\n",s); }
```

```
4      1      1      1  
3      2      1      1  
2      1      2      1  
1      3      1      2  
s=8
```





//源程序2:

```
#include <stdio.h>
```

```
int s=0, a[10]={0};
```

```
void f(int n ,int k)
```

```
{    for(int i=1; i<=n; i++)
```

```
    {    a[k]=i;
```

```
        if (n-i>0)    f(n-i,k+1);
```

```
        else if (n-i==0)
```

```
            {    for (int j=0; j<=k; j++)
```

```
                printf("%5d",a[j]);
```

```
                printf("\n");    s++;    }
```

```
    }
```

```
}
```

```
void main( )
```

```
{    int n;
```

```
    scanf("%d",&n);    f(n, 0);
```

```
    printf("s=%d\n",s);    }
```

```
4
    1      1      1      1
    1      1      2
    1      2      1
    1      3
    2      1      1
    2      2
    3      1
    4
s=8
```

## 递归巩固练习4

输出一个正整数 $n$ 的分解形式。例如,当 $n=4$ 时:

$$4=4$$

$$4=3+1$$

$$4=2+2$$

$$4=2+1+1$$

$$4=1+1+1+1$$

共计 5 种形式 。

当 $n=7$ 时, 共有15种形式。

当 $n=10$ 时, 共有42种形式。

**注意: 与练习3的区别**

```

#include <stdio.h>
int s=0, a[10]={0};
void f(int n ,int k)
{ for(int i=1; i<=n; i++)
  { a[k]=i;
    if ( n-i>0 && a[k-1]<=i ) f(n-i,k+1);
    else if ( n-i==0 && a[k-1]<=a[k] )
      { for (int j=1; j<=k; j++) printf("%5d",a[j]);
        printf("\n");      s++;  }
  }
}

void main( )
{   int n;
    scanf("%d",&n);  f(n, 1);
    printf("s=%d\n",s);  }

```

```

4
1 1 1 1 1
1 1 2 1
1 3 2
4
s=5

```

## 3.5 队列的表示和操作的实现



### 练习

设栈S和队列Q的初始状态为空，元素e1、e2、e3、e4、e5和e6依次通过S，一个元素出栈后即进入Q，若6个元素出队的序列是e2、e4、e3、e6、e5和e1，则栈S的容量至少应该是**(B)**。

(A) 2

(B) 3

(C) 4

(D) 6

# 队列的抽象数据类型

ADT Queue {

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

**数据关系:**  $R_1 = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i = 1, 2, \dots, n \}$

**基本操作:** 约定 $a_1$ 端为队列头, $a_n$ 端为队列尾

- (1) InitQueue (&Q)      //构造空队列
- (2) DestroyQueue (&Q)    //销毁队列
- (3) ClearQueue (&S)      //清空队列
- (4) QueueEmpty(S)        //判空. 空--TRUE,

# 队列的抽象数据类型

(5) **QueueLength(Q)**      //取队列长度

(6) **GetHead (Q,&e)**      //取队头元素,

(7) **EnQueue (&Q,e)**      //入队列

(8) **DeQueue (&Q,&e)**      //出队列

(9) **QueueTraverse(Q,visit())**      //遍历

**}ADT Queue**

## 队列的顺序表示 - - 用一维数组base[M]

```
#define M 100 //最大队列长度
```

```
Typedef struct {
```

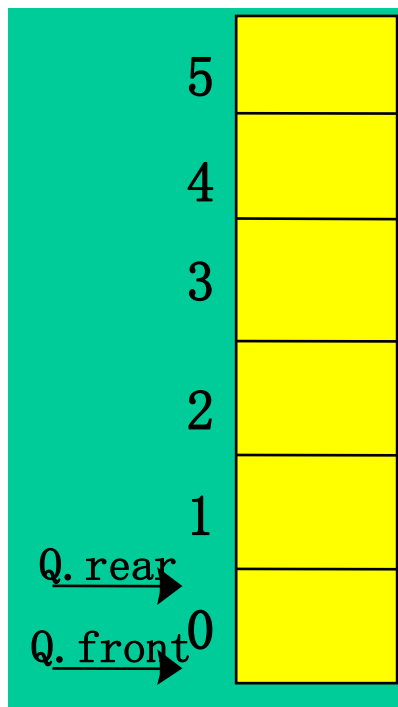
```
    QElemType *base; //初始化的动态分配存储空间
```

```
    int front;        //头指针
```

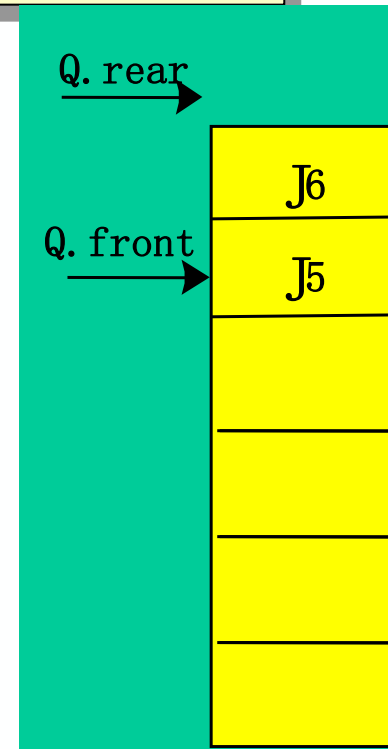
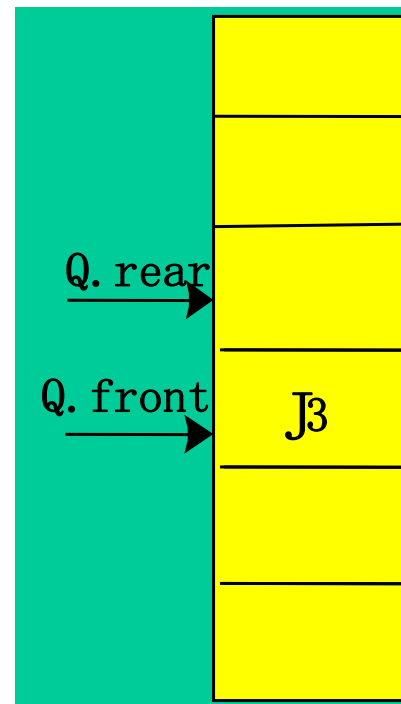
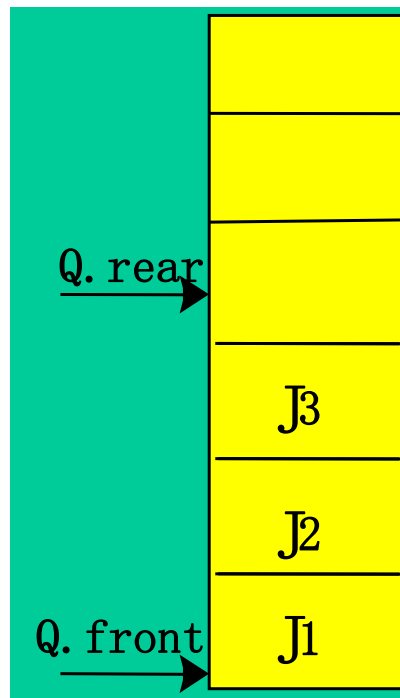
```
    int rear;         //尾指针
```

```
}SqQueue;
```

# 队列的顺序表示 - - 用一维数组base[M]



**front=rear=0**

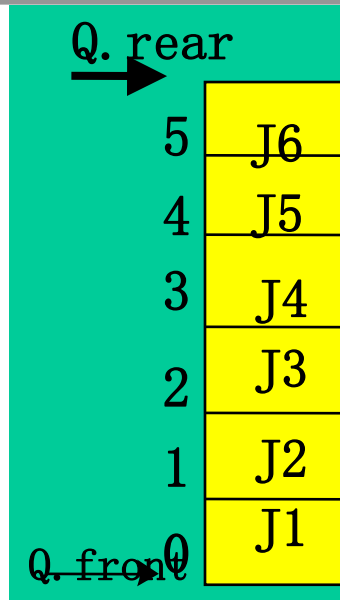


**空队标志:  $front == rear$**   
**入队:  $base[rear++] = x;$**   
**出队:  $x = base[front++];$**

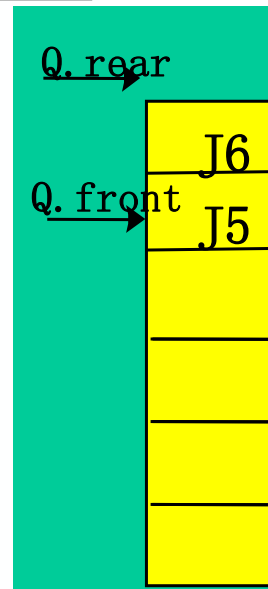


# 存在的问题

设大小为M

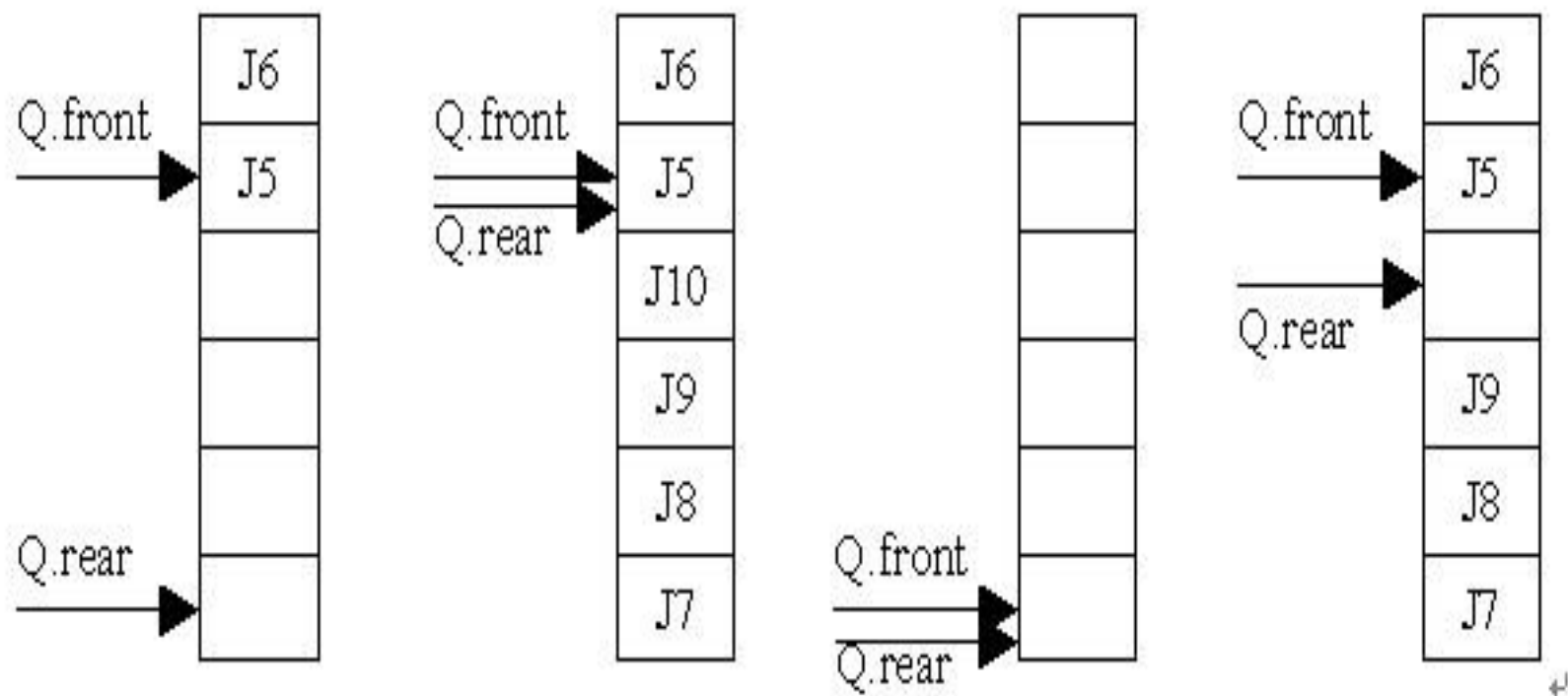


front=0  
rear=M时  
再入队—真溢出



front≠0  
rear=M时  
再入队—假溢出



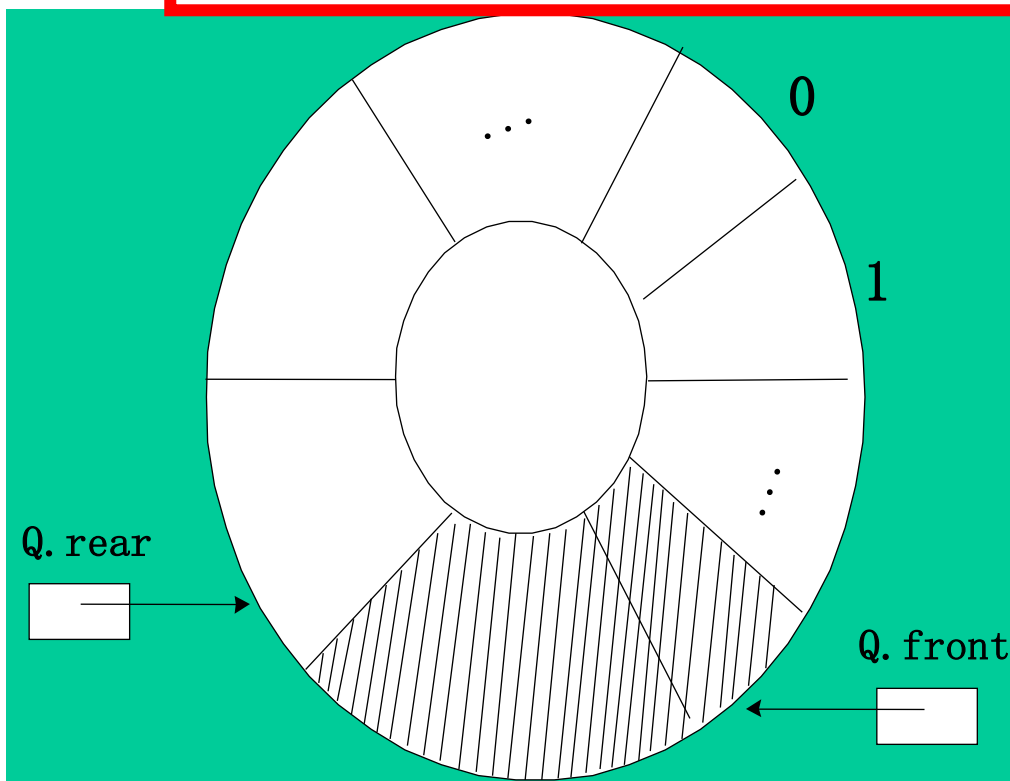


(a) 一般情况      (b) 队列空间被占满      (c) 空队      (d) 呈“满”状态的循环队列

图 3.12 循环队列中头、尾指针和元素之间的关系

# 解决的方法 - - 循环队列

base[0]接在base[M-1]之后  
若 $\text{rear}+1==M$   
则令 $\text{rear}=0$ ;



实现：利用“模”运算

入队：

$\text{base}[\text{rear}]=x;$

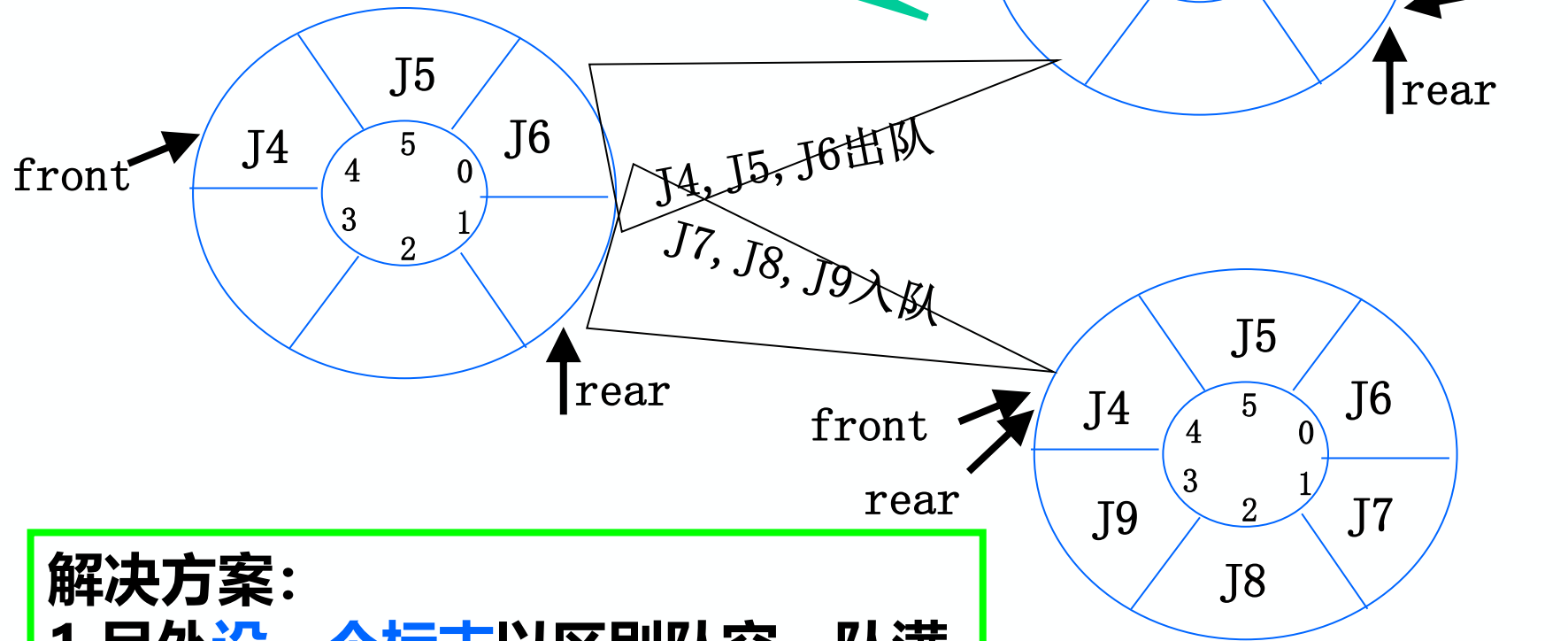
$\text{rear}=(\text{rear}+1)\%M;$

出队：

$x=\text{base}[\text{front}];$

$\text{front}=(\text{front}+1)\%M;$

队空:  $\text{front} == \text{rear}$   
队满:  $\text{front} == \text{rear}$



解决方案:

1. 另外设一个标志以区别队空、队满

2. 少用一个元素空间:

队空:  $\text{front} == \text{rear}$

队满:  $(\text{rear} + 1) \% M == \text{front}$

# 循环队列

```
#define MAXQSIZE 100 //最大长度
```

```
Typedef struct {
```

```
    QElemType *base; //初始化的动态分配存储空间
```

```
    int front;        //头指针
```

```
    int rear;         //尾指针
```

```
}SqQueue;
```



# 循环队列初始化

```
Status InitQueue (SqQueue &Q){  
    Q.base =new QElemType[MAXQSIZE]  
    if(!Q.base) exit(OVERFLOW);  
    Q.front=Q.rear=0;  
    return OK;  
}
```

# 求循环队列的长度

```
int QueueLength (SqQueue Q){  
    return (Q.rear-Q.front+MAXQSIZE)%MAXQSIZE;  
}
```

# 循环队列入队

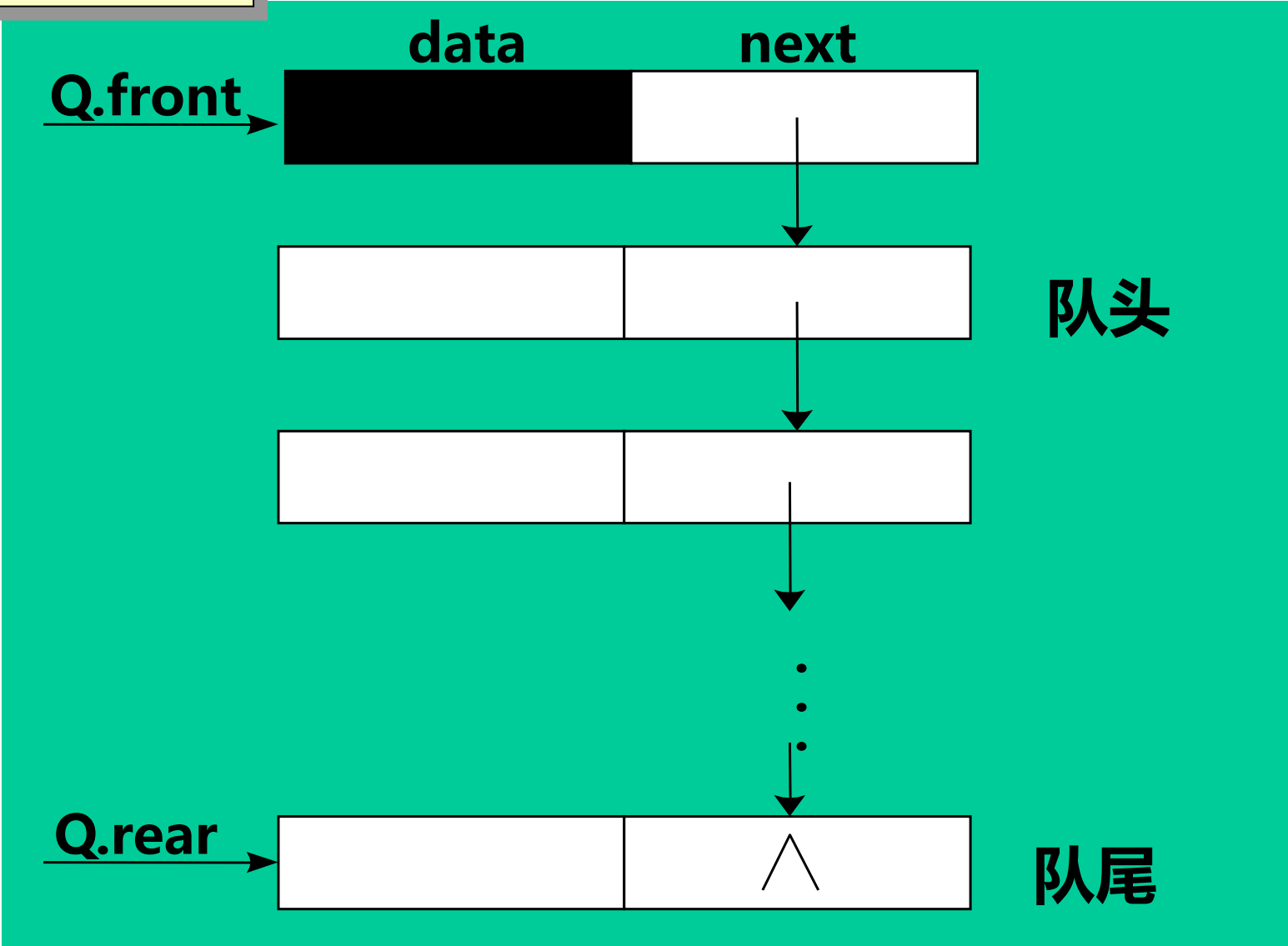
```
Status EnQueue(SqQueue &Q, QElemType e){  
    if((Q.rear+1)%MAXQSIZE==Q.front) return ERROR;  
    Q.base[Q.rear]=e;  
    Q.rear=(Q.rear+1)%MAXQSIZE;  
    return OK;  
}
```



# 循环队列出队

```
Status DeQueue (LinkQueue &Q, QElemType &e){  
    if(Q.front==Q.rear) return ERROR;  
    e=Q.base[Q.front];  
    Q.front=(Q.front+1)%MAXQSIZE;  
    return OK;  
}
```

# 链队列



# 链队列

```
typedef struct QNode{  
    QElemType  data;  
    struct Qnode *next;  
}Qnode, *QueuePtr;
```

```
typedef struct {
```

```
    QueuePtr front;
```

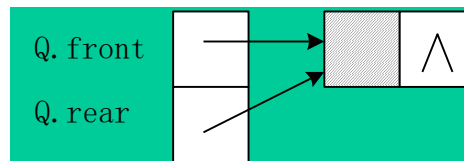
**//队头指针**

```
    QueuePtr rear;
```

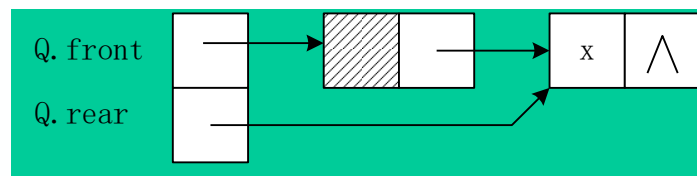
**//队尾指针**

```
}LinkQueue;
```

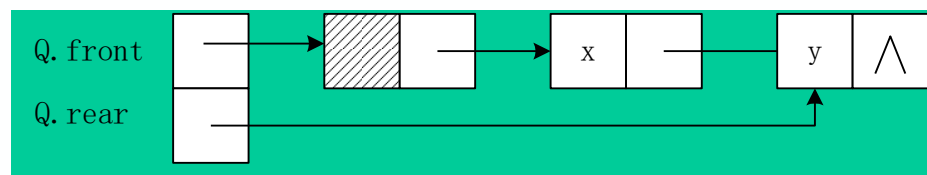
# 链队列



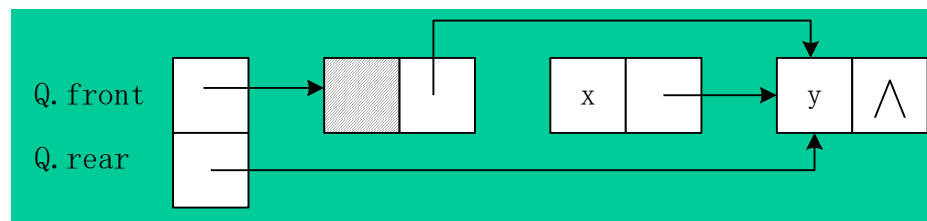
(a) 空队列



(b) 元素`x`入队列



(c) 元素`y`入队列



(d) 元素`x`出队列

# 链队列初始化

```
Status InitQueue (LinkQueue &Q){  
    Q.front=Q.rear=(QueuePtr) malloc(sizeof(QNode));  
    if(!Q.front) exit(OVERFLOW);  
    Q.front->next=NULL;  
    return OK;  
}
```

# 销毁链队列

```
Status DestroyQueue (LinkQueue &Q){  
    while(Q.front){  
        Q.rear=Q.front->next;  
        free(Q.front);  
        Q.front=Q.rear;    }  
    return OK;  
}
```

## 判断链队列是否为空

```
Status QueueEmpty (LinkQueue Q){  
    return (Q.front==Q.rear);  
}
```

# 求链队列的队头元素

```
Status GetHead (LinkQueue Q, QElemType &e){  
    if(Q.front==Q.rear) return ERROR;  
    e=Q.front->next->data;  
    return OK;  
}
```



# 链队列入队

```
Status EnQueue(LinkQueue &Q, QElemType e){  
    p=(QueuePtr)malloc(sizeof(QNode));  
    if(!p) exit(OVERFLOW);  
    p->data=e; p->next=NULL;  
    Q.rear->next=p;  
    Q.rear=p;  
    return OK;  
}
```

