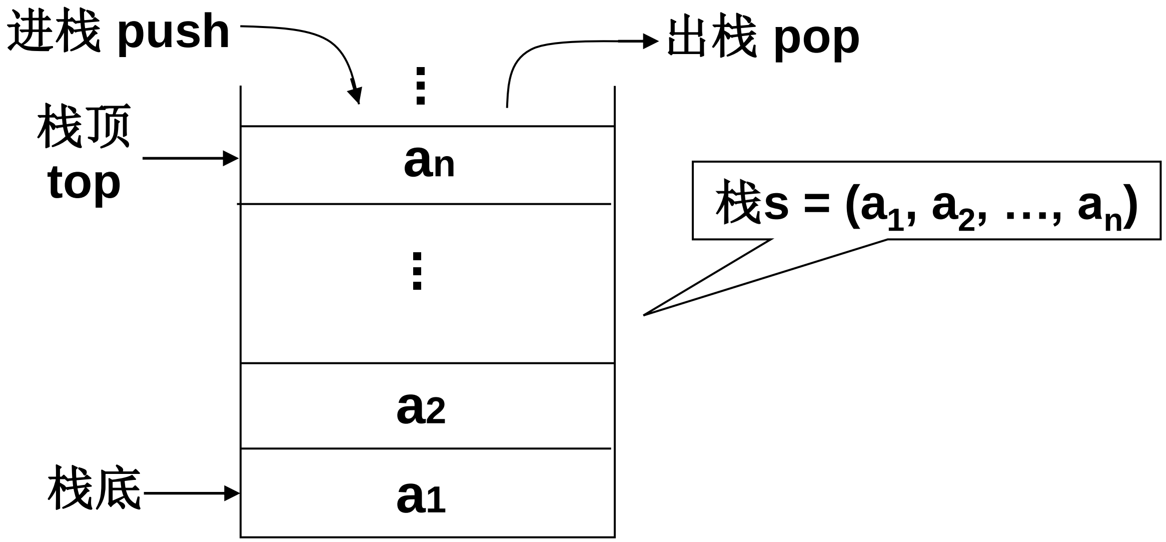
# Ch3 栈和队列

- 栈(stack)的定义和特点

定义　限定仅在表尾进行插入或删除运算的线性表, 表尾 -- 栈顶 (top), 表头 -- 栈底(bottom), 不含数据元素的空表称空栈

特点　后进先出(Last-In-First-Out / LIFO)　或 新进后出(FILO)




**Applications of stack:** (生活中, 厨房里叠在一起的盘子)

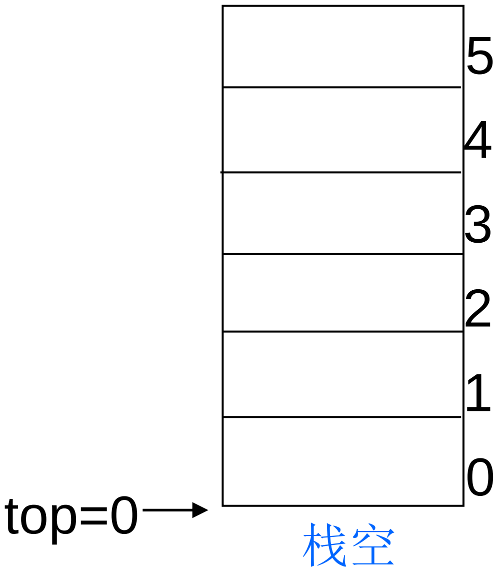
- **Balancing of symbols**
- **Redo-undo features at many places like editors, photoshop**
- **Forward and backward feature in web browsers**
- **Used in many algorithms like Tower of Hanoi, tree traversals**

- 栈的存储结构 -- 顺序结构和链式结构

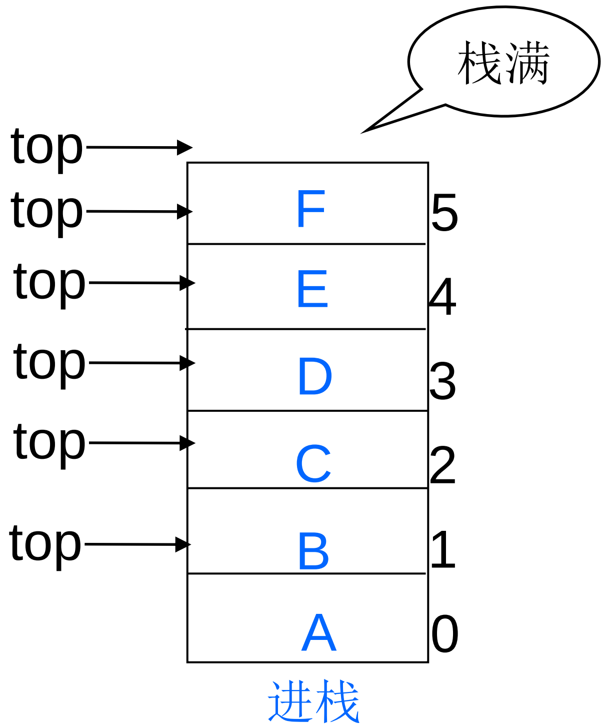
顺序结构的栈

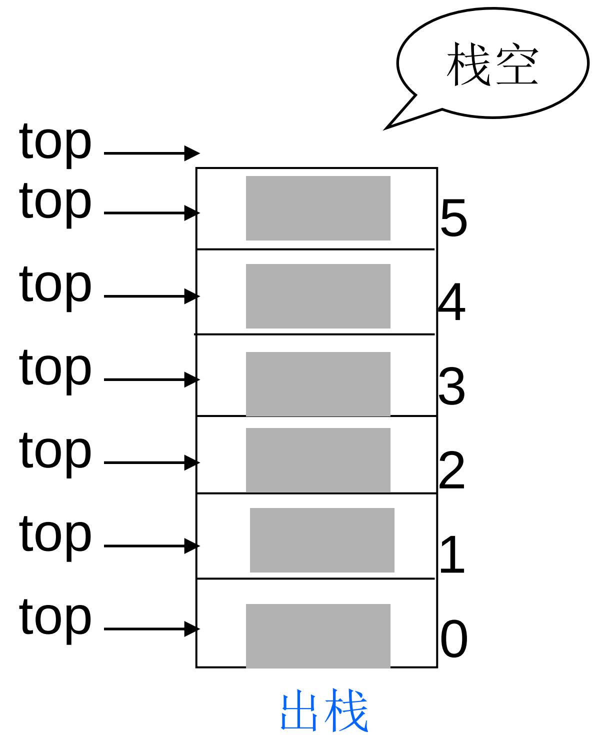
数据结构的表示: 用一维数组实现 ➔

```
#define MaxSize 6
typedef char StackElem;
typedef struct seqstack {
    int top; //指示栈顶位置
    StackElem elem[MaxSize];
} SeqStack;
```




**约定: 栈顶“指针”(“索引”/“下标”)top, 指示实际栈顶 后 / 下一个 空位置,初值为top=0**




**top == MaxSize: 栈满, 若此时入栈, 则上溢(overflow)**
**top == 0: 栈空, 若此时出栈, 则下溢(underflow)**

- 栈的存储结构 -- 顺序结构和链式结构

顺序结构的栈

几个运算及其算法描述 P46-47

```
//create an empty stack
void InitialStack(SeqStack *s)
{ s->top = 0; }
```

```
//StackEmpty: returns non-zero
//if the stack is empty
Boolean StackEmpty(SeqStack *s)
{ return s->top <= 0; }
```

```
Boolean StackFull(SeqStack *s)
{ return s->top >= MaxSize; }
```

```
//push an item onto the stack
void Push(StackElem item, SeqStack *s)
{ if (StackFull(s))
      Error("Stack is full");
  else s->elem[s->top++] = item; }
```

```
//return and remove the item that
//was inserted most recently
StackElem Pop(SeqStack *s)
{ if (StackEmpty(s)) Error("Stack is empty");
  else return s->elem[--s->top]; }
```

若约定栈顶“指针”top 指示实际栈顶的位置, 则其初值为top=–1. 上述基本运算需作适当修改: ➜

初始化: s->top = –1;
判栈空: return s->top <= –1;
判栈满: return s->top >= MaxSize –1;
入栈: s->elem[++ s->top] = item;
出栈:s->elem[s->top--]; 取栈顶元素:s->elem[s->top];

- 栈的存储结构 -- 顺序结构和链式结构

链式结构的栈

数据结构的表示 用单链表实现 ➔

```
typedef struct node {
    StackElem elem;
    struct node *next;
} Stack, *LinkedStack;
```

在存储空间足够大的情况下, 链栈一般不考虑溢出现象.

考虑到计算的方便性, 链栈中一般无需设置头结点

链栈的初始化

```
void InitialStack(LinkedStack top)
{ top = NULL; }
```

判断链栈是否空栈

```
boolean StackEmpty(LinkedStack top)
{ return top == NULL; }
```

- 栈的存储结构 -- 顺序结构和链式结构

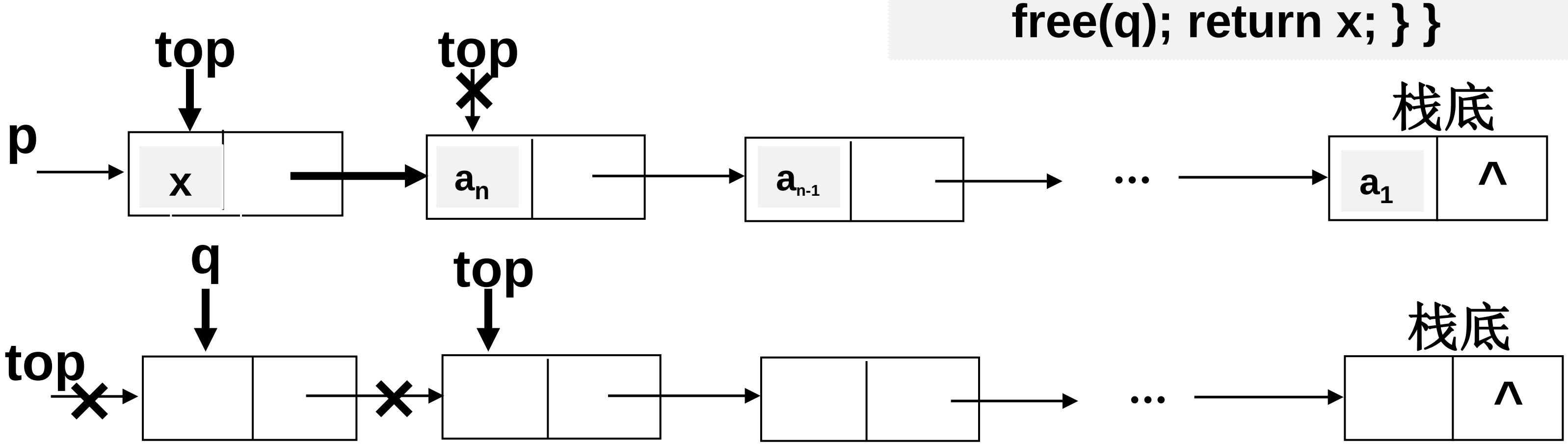
  链式结构的栈 用单链表实现

## 链栈入栈运算

```
void Push(LinkedStack top, StackElem x) {
    p = (Stack *)malloc(sizeof(Stack));
    p->elem = x;
    p->next = top; top = p; }
```

## 链栈出栈运算

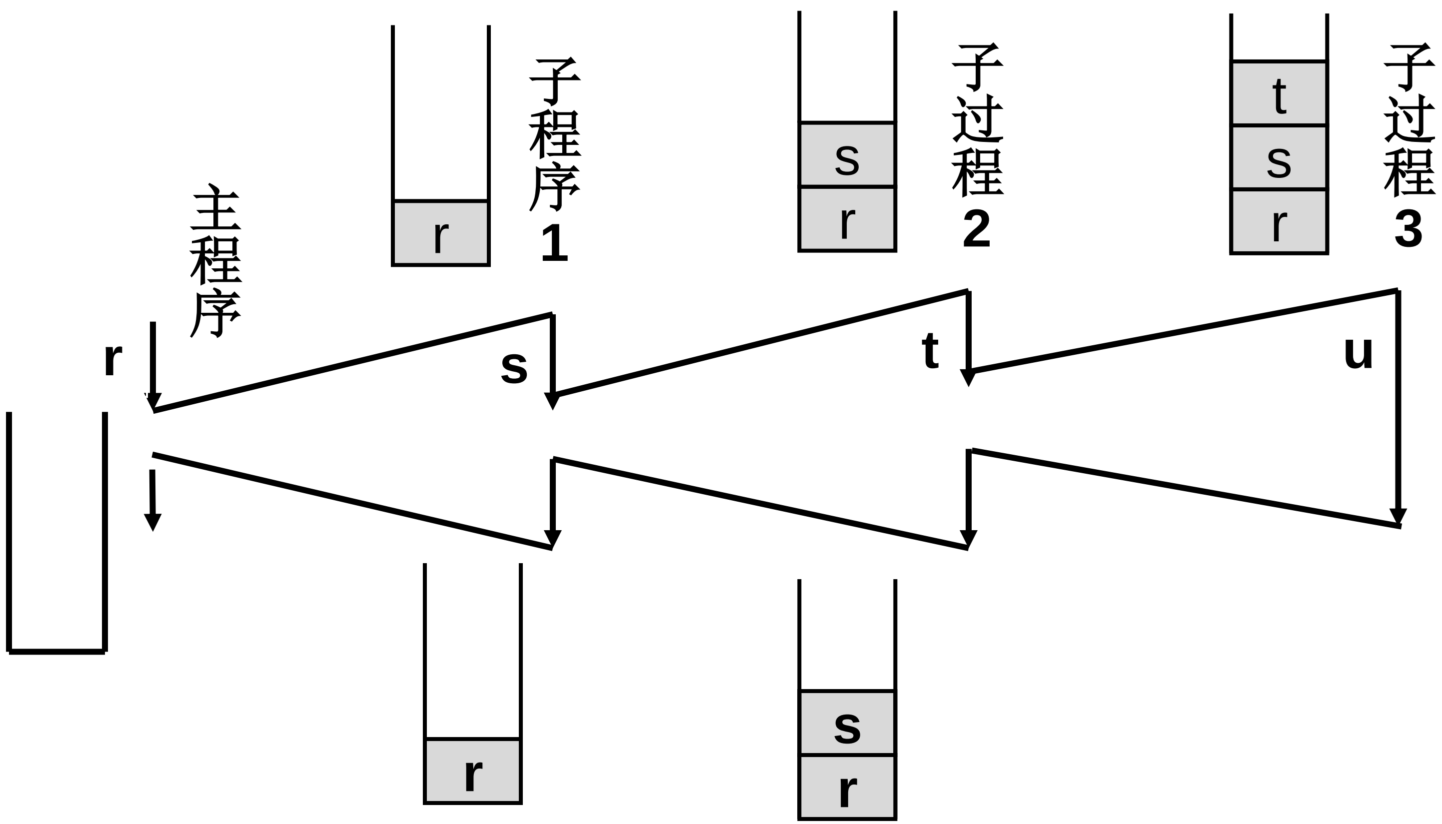
```
StackElem Pop(LinkedStack top) {
    if ( StackEmpty(top) )
      Error(“Stack is empty”);
    else {
      x = top -> elem;
      q = top; top = top->next;
      free(q); return x; } }
```



- 栈的应用举例

## 1. 子程序(过程/函数/方法)的嵌套调用-- Subprogram calls

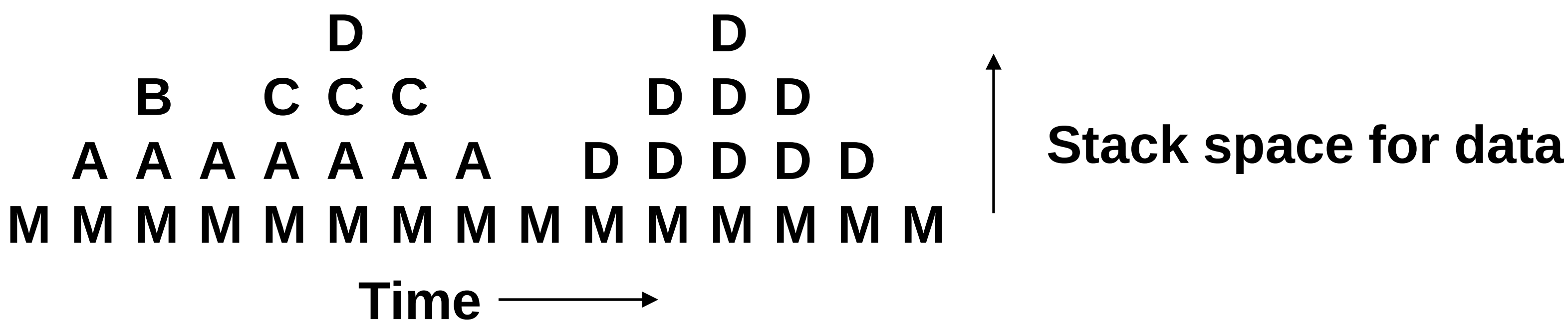
| 主程序r | 子程序1-s | 子程序2-t | 子程序3-u |
|---|---|---|---|
| { | { | { | { |
| …; | …; | …; | …; |
| call s; | call t; | call u; | …; |
| …; | …; | …; | …; |
| } | } | } | } |

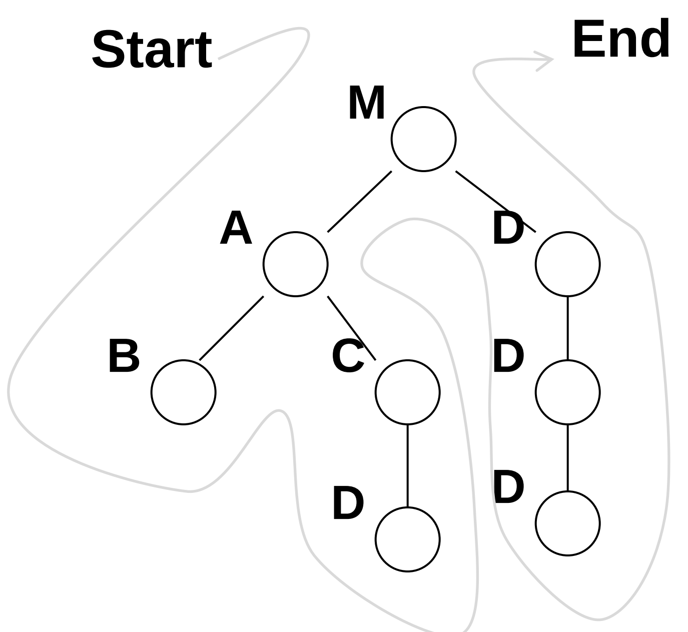


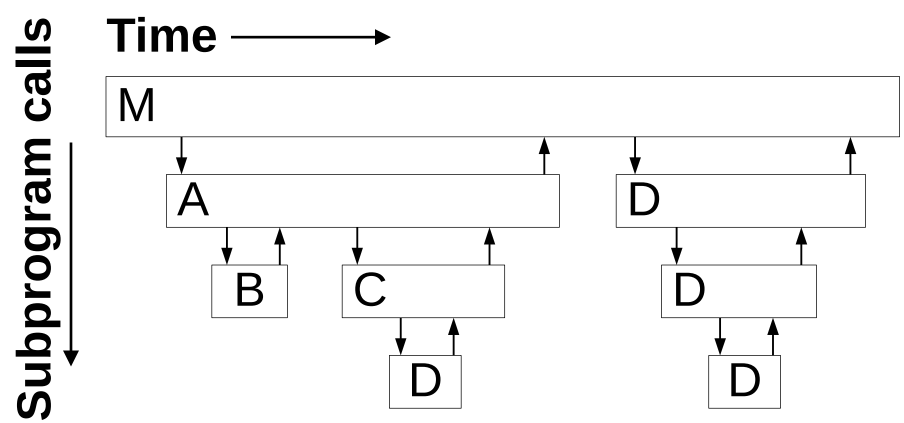
- 栈的应用举例

**1. 子程序(过程/函数/方法)的嵌套调用-- Subprogram calls**

又如: **Stack frames for subprogram calls**




**Tree of subprogram calls**

- 栈的应用举例

## 2. 递归函数及其实现

↱ 即递归过程R包含另一过程D, 而D又调用R

递归函数: 函数<u>直接</u> 或 <u>间接</u>调用自身的过程 / function that calls itself.

一个例子 -- 递归函数的执行情况分析

```
void  print(int  w)
{   int  i;
     if ( w != 0 )
     {  print(w-1);
         for( i=1; i<=w; ++i )  printf(“%3d,”,w);
         printf(“/n”);
      }
}
```

设w=3,则
运行结果:

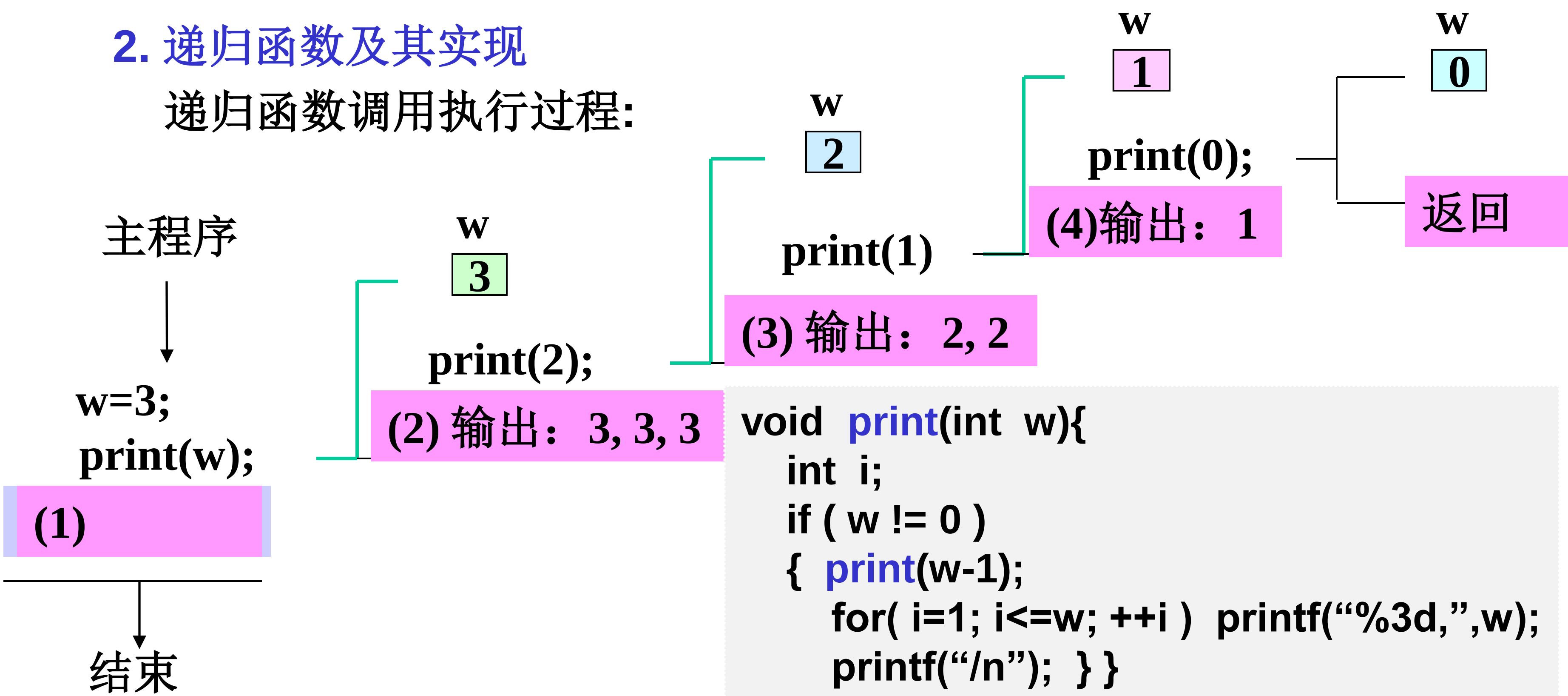
1,
2, 2,
3, 3, 3,

执行过程

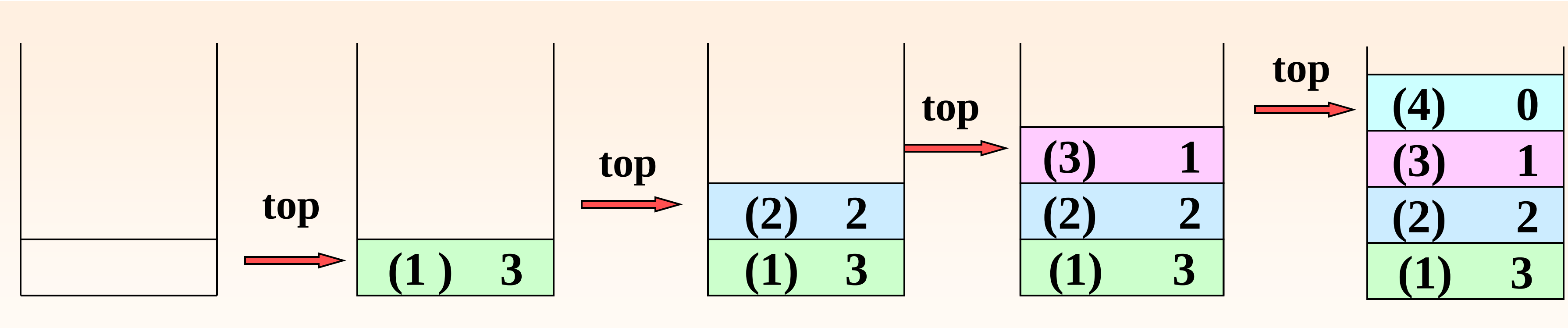
- 栈的应用举例

**2. 递归函数及其实现**

递归函数调用执行过程:




```
void  print(int  w){
    int  i;
    if ( w != 0 )
    {  print(w-1);
       for( i=1; i<=w; ++i )  printf("%3d,",w);
       printf("/n");  } }
```



- 栈的应用举例

## 2. 递归函数及其实现

又如 The factorial function: $n! = n\times(n-1)\times \cdots \times 1$

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n\times(n-1)! & \text{if } n > 0 \end{cases}$$

$$\begin{aligned} 4! &= 4\times 3! = 4\times(3\times 2!) \\ &= 4\times(3\times (2\times 1!)) \\ &= 4\times(3\times (2\times (1\times 0!))) \\ &= 4\times(3\times (2\times (1\times 1))) \\ &= 4\times(3\times (2\times 1)) \\ &= 4\times(3\times 2) = 4\times 6 = 24 \end{aligned}$$

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

*COMMENTS* Every recursive process consists of two parts:

1. A base case that is processed without recursion; //边界条件
2. A general method that reduces a particular case to one or more of the smaller cases, thereby making progress toward eventually reducing the problem all the way to the base case. //收敛: 使问题向边界条件转化的规则

That is find a stopping rule and the reduction step (i.e. define the recursion).

- 栈的应用举例

## 2. 递归函数及其实现

又如 **The factorial function: $n! = n \times (n-1) \times \cdots \times 1$**

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{if } n > 0 \end{cases}$$

$$\begin{aligned} 4! &= 4 \times 3! = 4 \times (3 \times 2!) \\ &= 4 \times (3 \times (2 \times 1!)) \\ &= 4 \times (3 \times (2 \times (1 \times 0!))) \\ &= 4 \times (3 \times (2 \times (1 \times 1))) \\ &= 4 \times (3 \times (2 \times 1)) \\ &= 4 \times (3 \times 2) = 4 \times 6 = 24 \end{aligned}$$

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

如何评估递归描述形式的算法的时间复杂性:

- **T(n) is time used for input n:**

$$T(n) = \begin{cases} 1 & \text{if } n \text{ is } 0 \\ 1 + T(n-1) & \text{otherwise} \end{cases}$$

- **Repeated substitution expands recurrence:**

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

- **Generalize pattern:**

$$= i*1 + T(n - i)$$

- **Solve for i = n:**

$$\begin{aligned} &= n + T(0) \\ &= n + 1 \quad \rightarrow \quad O(n) \end{aligned}$$

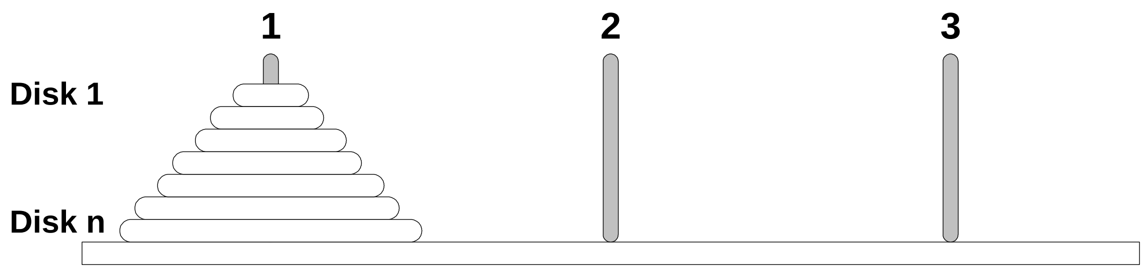
- 栈的应用举例

**2. 递归函数及其实现**

再如 梵·塔问题(Towers of Hanoi Problem)

**Move of 64 disks from tower 1 to tower 3 using tower 2 as temporary storage.**

记作: **Move(64, 1, 3, 2).**

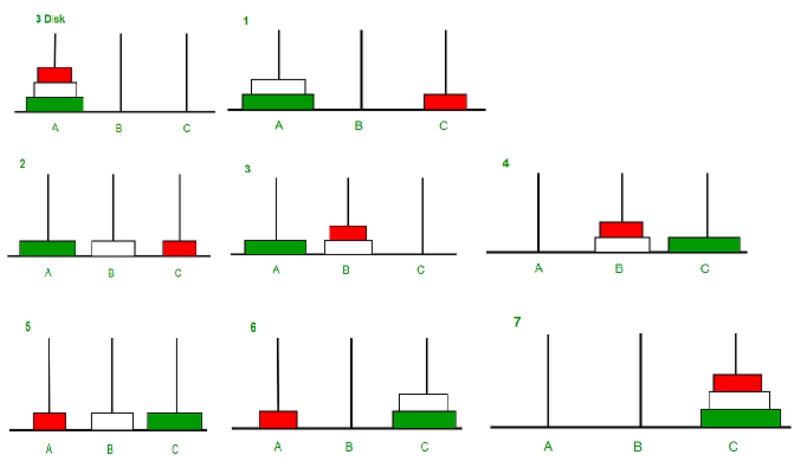



**Moving rules:**
- Move only one disk at a time
- Never place a larger disk on a smaller one

**Trace for Disks = 2:**

Move disk 1 from 1 to 2
Move disk 2 from 1 to 3
Move disk 1 from 2 to 3

**Trace for Disks = 3:**




**The pattern is:**

Move 'n-1' disks from 1 to 2
Move the last ($n^{th}$) disk from 1 to 3 ⇦
Move 'n-1' disks from 2 to 3

- 栈的应用举例

## 2. 递归函数及其实现

再如 梵·塔问题(Towers of Hanoi Problem)

```
#define Disks 64 //Number of disks on the first towers. P55/A.3.5
void main( ) { Move(Disks, 1, 3, 2); }
void Move(int count, int start, int finish, int temp)
{   if (count == 1) {
        printf("Move disk 1 from %d to %d.\n", start, finish);
        return; }
    Move(count – 1, start, temp, finish);
    printf("Move disk %d from %d to %d.\n", count, start, finish);
    Move(count – 1, temp, finish, start);  }
```

- T(n) is time used for input n: $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 1 + 2T(n-1) & \text{otherwise} \end{cases}$
- Repeated substitution expands recurrence:
  $T(n) = 1 + 2T(n-1) = 1 + (2 + 4T(n-2)) = 1 + (2 + (4 + 8T(n-3))) = \ldots$
- Generalize pattern: $= 1 + 2 + 4 + 8 + 16 + \ldots + 2^{i-1} + 2^iT(n – i) = 2^i – 1 + 2^iT(n – i)$
- Solve for n – i = 0 ( i = n): $= 2^n – 1 + 2^nT(0) = 2^n – 1$ -> $O(2^n)$

n=64个disks共移动次数:

$\Sigma = 1 + 2 + 4 + \cdots + 2^{63} = 2^0 + 2^1 + 2^2 + \cdots + 2^{63} = 2^{64} – 1$

$= 2^n – 1$

- 栈的应用举例

## 2. 递归函数及其实现

**Comparisons between Recursion(递归) and Iteration(迭代/递推)**

**Fibonacci Numbers:**

$F_0 = 0,$

$F_1 = 1,$

$F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$

```
// Fibonacci: recursive version.
int Fibonacci(int n)
{   if(n <= 0) return 0;
    else if (n == 1) return 1;
            else return Fibonacci(n-1) + Fibonacci(n-2); }
```

```
// Fibonacci: iterative version.
int Fibonacci(int n)
{  int i, twoback, oneback, current;
    if(n <= 0) return 0;
    else if (n == 1) return 1;
            else { twoback = 0; oneback =1;
                    for (i = 2; i <= n; i++) {
                          current = twoback + oneback;
                          twoback = oneback; oneback = current; }
                    return current; } }
```

- 栈的应用举例

## 2. 递归函数及其实现

| Recursion | Iteration |
|---|---|
| an implicit stack<br>sometimes duplicate tasks<br>转子(递归调用)语句的时间开销 | sometimes an explicit stack<br>much more complicated and<br>harder to understand |

**Conclusions (be at issue)**

**It’s possible to take any recursive program into nonrecursive form. But the only reason for removal recursion is if you are forced to program in a language that does not support recursion.**

- 栈的应用举例

## 2. 递归函数及其实现

### 递归小结 -- Thinking recursively

- **Recursive decomposition is the hard part**
  - **Find recursive sub-structure – solve problems using result from smaller subproblem(s)**
  - **Identify base case – simple possible case, directly solvable, recursion advances to it**
- **Common patterns**
  - **Handle first and/or last, recur on remaining**
  - **divide in half, recur on one/both halves**
  - **make a choice among options, recur on updated state**
- **Placement of recursive call(s)**
  - **Recur-then-process vs. process-then-recur**

递归算法适用的一般场合:

- 数据的定义形式按递归定义, 如 计算裴波那契数列 等;
- 数据之间的关系按递归定义, 如 树的遍历, 图的搜索 等;
- 问题解法按递归算法实现, 如 分治法, 回溯法 等.

- 栈的应用举例

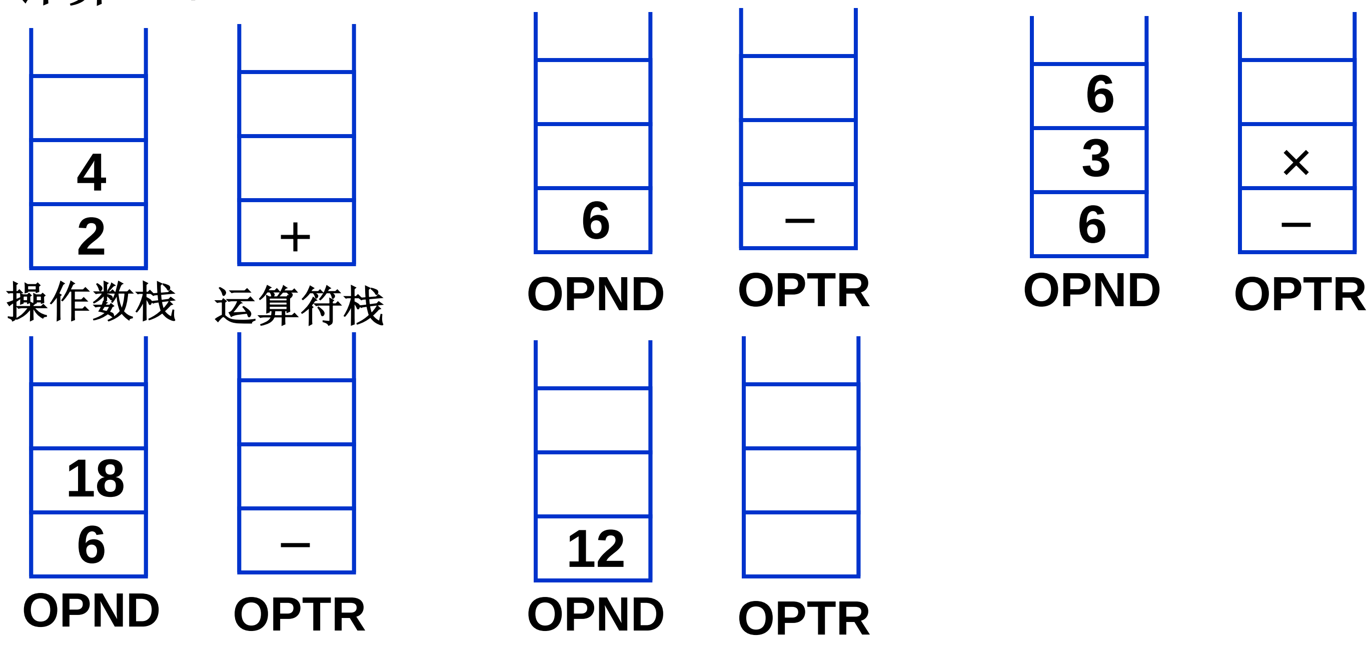
## 3. (中缀)表达式的计算

**P52-53**
- 操作数(operand)和运算符(operator)
- 运算规则(关键是运算符的优先级的考虑: **P53/表3.1**)
- 算法的基本思想: 设有两个栈 – **OPND & OPTR**

依次读表达式中的每个字符, 若是操作数则进**OPND**栈; 若是操作符, 则和**OPTR**栈的栈顶运算符比较优先级后作相应操作, 直至表达式空为止.

算法描述: **P53-54/A.3.4**

例 计算 $2 + 4 - 3 \times 6$



- 栈的应用举例

## 4. 其它应用

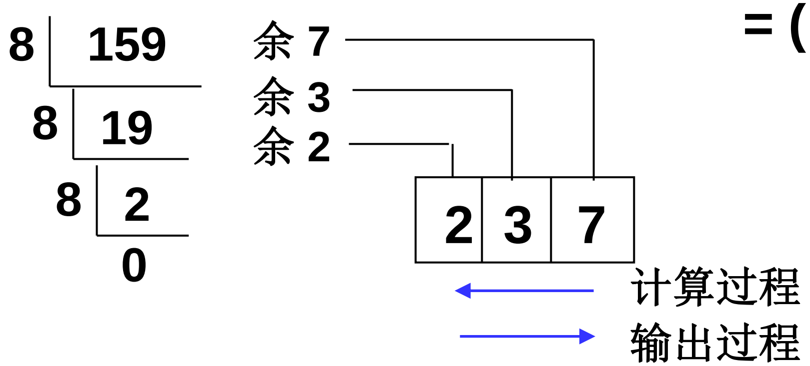
### 数制转换、地图着色等

例 把十进制的数(N=159)转换成八进制(B=8)表示的数:

$$N = 159_{10} = 2\cdot8^2 + 3\cdot8^1 + 7\cdot8^0 = 237_8 = \sum_{i=0}^{\lfloor \log_B N \rfloor} b_i\cdot B^i \quad 其中: 0 \le b_i \le B-1$$

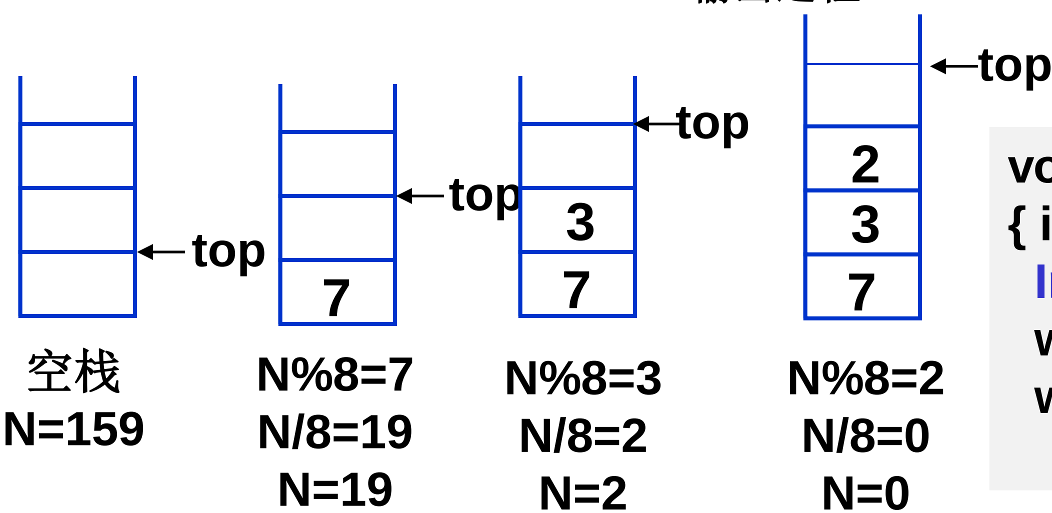
令 $j = \lfloor \log_B N \rfloor$, 则 $N = b_j\cdot B^j + b_{j-1}\cdot B^{j-1} + \cdots + b_1\cdot B + b_0$

$$= ( b_j\cdot B^{j-1} + b_{j-1}\cdot B^{j-2} + \cdots + b_2\cdot B + b_1 ) \cdot B + b_0$$




上式中的$b_0$为N整除B所得的余数,
括号中的和式恰为N整除B所得的商.
由此看出其本质是递归运算:

$$N = (N \text{ div } B) \times B + N \text{ mod } B$$




```
void MultiBaseOutput(int N, int B)
{ int i;  Stack *s;
  InitStack(s);
  while ( N ) { Push(s, N%B); N = N/B; }
  while ( !StackEmpty(s) )
      printf( Pop(s) ); }  //O(log_B N)
```

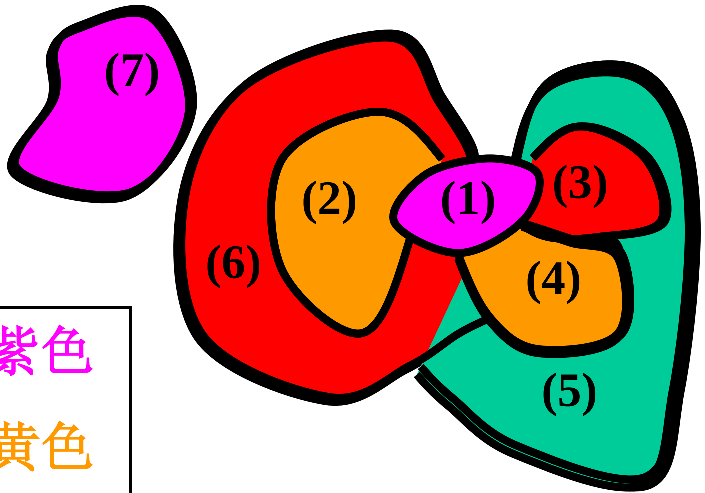
▪ 栈的应用举例

## 4. 其它应用

### 数制转换、地图着色等

例 地图四染色问题: 用四种不同的颜色给地图的各个图块填色,
要求相邻的图块所填的颜色不同.

思路: 采用解决“回溯”问题的算法, 即一个问题由若干步组成,
在每一步的处理中都有若干种选择可能, 整个问题最终
由每一步的恰当选择而集成. 通过运用栈的元素“后进先出”
的特点来具体实现回溯.




| | | |
|---|---|---|
| 1# | 紫色 | |
| 2# | 黄色 | |
| 3# | 红色 | |
| 4# | 绿色 | |

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 2 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 3 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| 4 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| 5 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 6 | 1 | 1 | 0 | 1 | 1 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 2 | 4 | 3 | 1 |

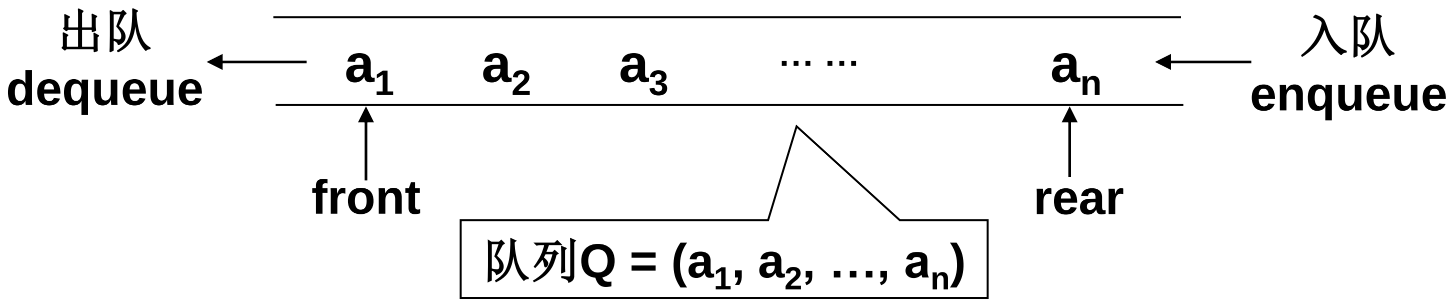
**Exercises (After class):**

1. 字符a, b, c, d依次通过一个栈, 按出栈的先后次序组成字符串, 至多可以组成多少个不同的字符串? 并分别写出它们.
2. 若栈的数据元素存放在一维数组S[1..Max]中, 栈顶指示top的初值为0, 若top<Max, 写出将数据元素x入栈的算法关键语句.
3. 画出计算表达式 (3+3)/2×5-6 时操作数栈和操作符栈的变化情况.
4. 利用栈的基本运算写出求解下列问题的算法(回文游戏): read one line of input and write it backward.
5. Euclid’s algorithm: The greatest common divisor(GCD) of two positive integers is the largest integer that divides both of  them. E.g.GCD(216,192)=24.
   1) Write a recursive function GCD(x,y:integer): if y=0,then the GCD of x and y is x; otherwise the GCD of x and y is the same as the GCD of y and x%y.
   2) Rewrite the function in iterative form.

- 队列(queue)的定义及特点

  定义　队列是限定只能在表的一端进行插入运算, 在表的另一端进行删除的线性表. 其中, 允许插入的一端称之为队尾(rear), 允许删除的一端称之为队头(front)

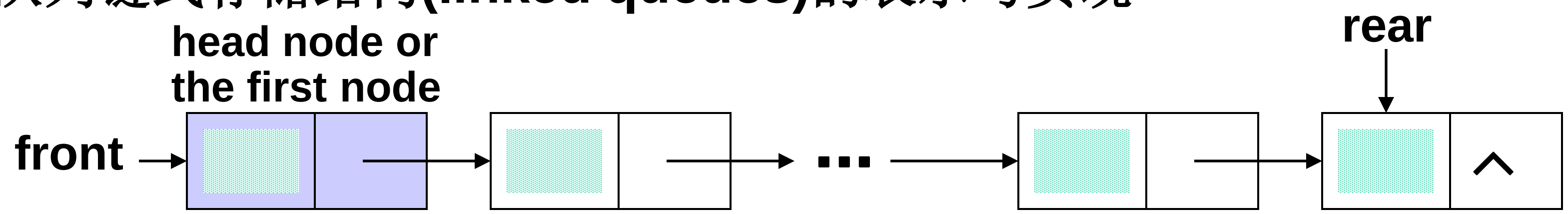
  特点　先进先出(FIFO)




**The property of Queue makes it useful in following kind of scenarios:**

- **When a resource is shared among multiple consumers. Examples include Call Center, CPU scheduling, etc. [*Priority Queues*]**
- **When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, etc.**

- 队列链式存储结构(linked queues)的表示与实现

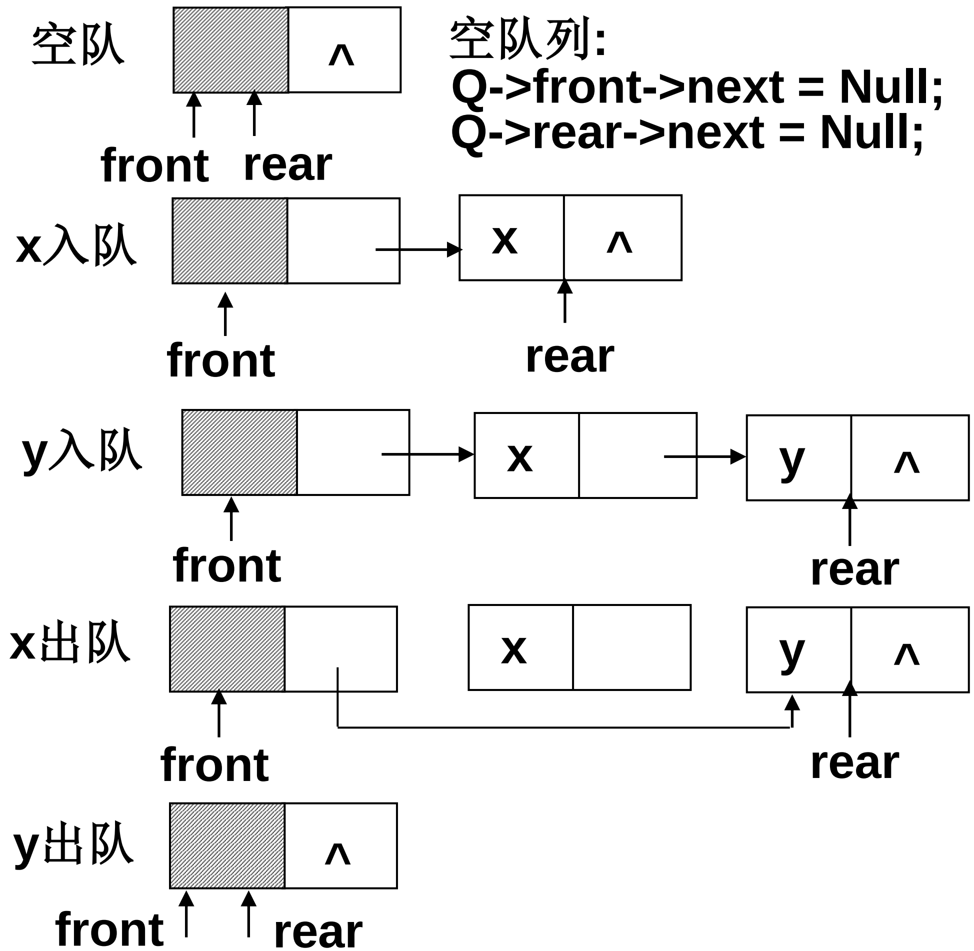


- 队列链式存储结构(linked queues)的表示与实现

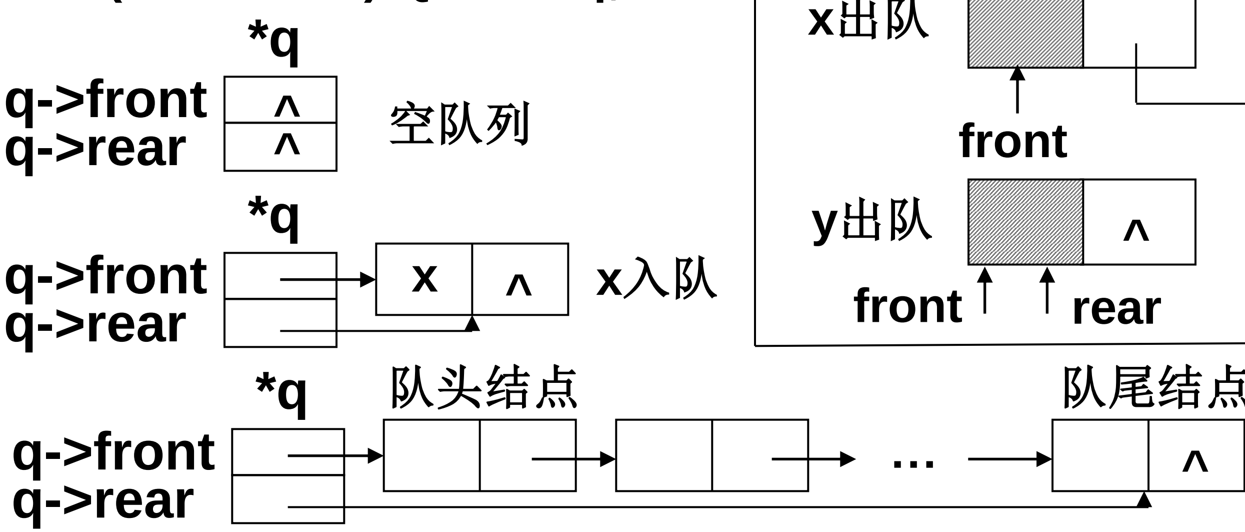
链式结构的队列 用单链表表示↓

```
typedef  char QueueElem;
typedef struct queuenode {
    QueueElem elem;
    struct queuenode *next;
 } QueueNode;
typedef struct queue {
    QueueNode *front;
    QueueNode *rear;
 } Queue;
```

运算/操作 (带头结点): Queue *Q;




运算/操作 (不带头结点):Queue *q;



- 队列链式存储结构(linked queues)的表示与实现

  算法描述 **P61-62**

```
//create an empty queue
void InitialQueue(Queue *q)
{ q->front = q->rear = Null; }  //P62 Q->front->next = Q->rear->next = Null
                                //because Q points to the head node
```

```
//is the queue empty?
boolean QueueEmpty(Queue *q)
{ return q->front == Null || q->rear == Null; }
//Q->front == Q->rear || Q->front->next == Null
```

```
//get the front item of the queue
QueueElem QueueFront (Queue *q)
{ if (QueueEmpty(q))
     Error(“The queue is empty.”);
  else
     return q->front->elem;  //Q->front->next->elem
}
```

- 队列链式存储结构(linked queues)的表示与实现

  算法描述 **P61-62**

```
//insert an item onto queue
void EnQueue(QueueNode *p, Queue *q)
{ if (!p) Error("Attempt to append a nonexistent node to the queue.");
  else if (QueueEmpty(q))
            q->front = q->rear = p;  //set both front and rear to p
                                     //若带头结点: Q->front->next = Q->rear = p;
        else {  //place p after previous rear of the queue
             q->rear->next = p;
             q->rear = p; } }
```

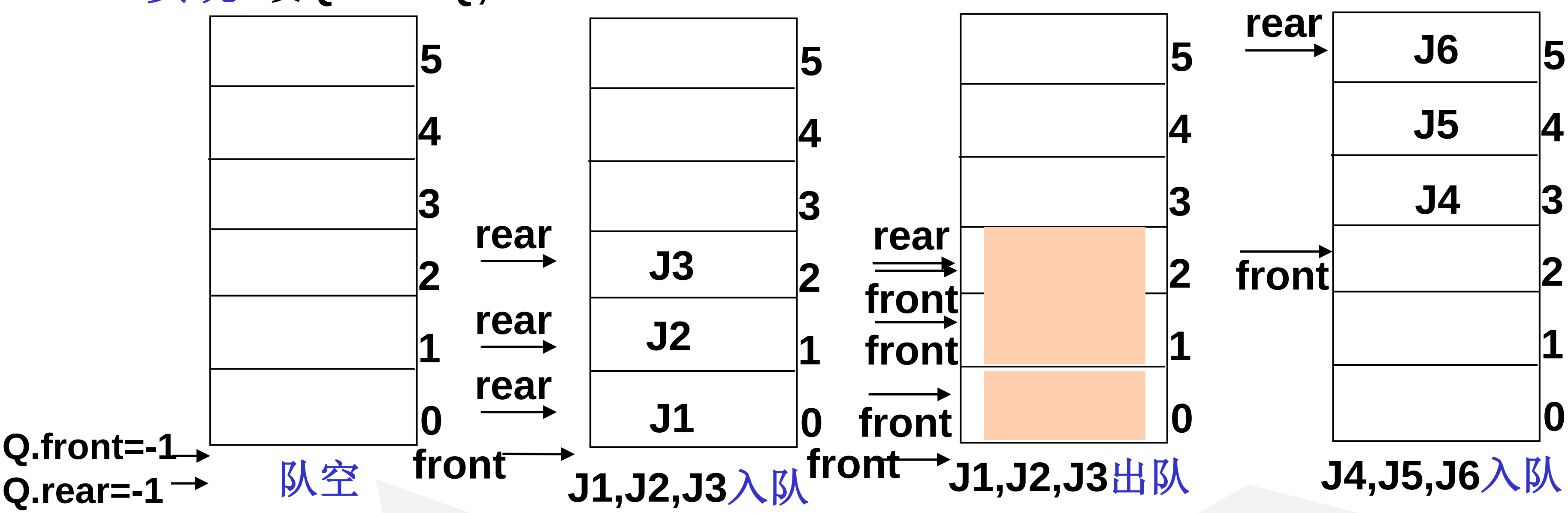
```
//return and remove the item that was inserted least recently
QueueElem DeQueue(Queue *q)
{ if (QueueEmpty(q))
      Error("Attempt to delete a node from an empty queue.");
  else {  p = q->front;   //P62 p = Q->front->next
          x = p->elem;
          q->front = p->next;  //Q->front->next = p->next;
          if (q->rear == p)      //当队列中只有一个结点时, 出队后队列为空. Q->rear == p.
              q->rear = Null;    //所以队尾指针置为空. Q->rear = Q->front(指向头结点)
          free(p); return x; } }
```

- 队列顺序存储结构(contiguous queues)的表示与实现

表示　一维数组➜

```
#define MaxQueue 6
typedef  char QueueElem;
typedef struct queue {
     int front;
     int rear;
     QueueElem elem[MaxQueue];
 } Queue;
```

实现　设Queue Q;




约定:
- Q.rear 指示队尾元素
- Q.front 指示队头元素前一位置

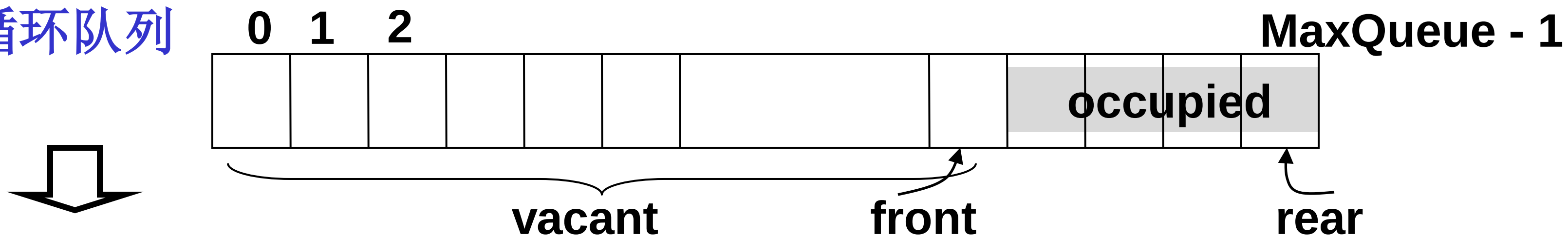
则初值 Q.front = Q.rear = -1

设Queue Q; 则
队列满条件: Q.rear == MaxQueue-1
入队列: Q.elem[++rear] = x
空队列条件: Q.front == Q.rear
出队列: x = Q.elem[++front]

假溢出

- 队列顺序存储结构(contiguous queues)的表示与实现

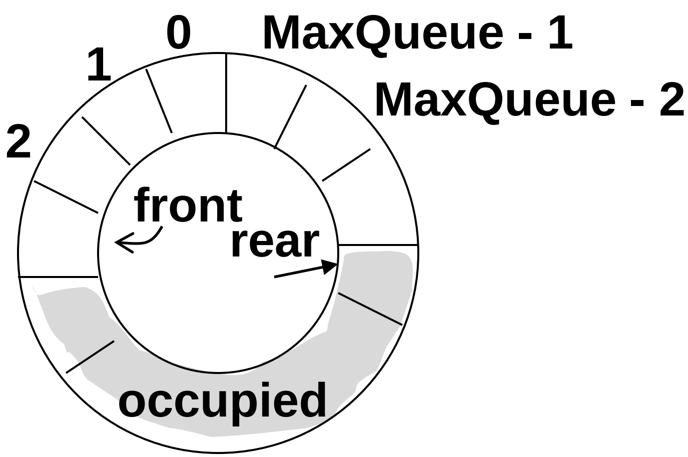
循环队列




**Implementation of circular arrays, 即当rear指示数组上界MaxQueue-1时,则:**

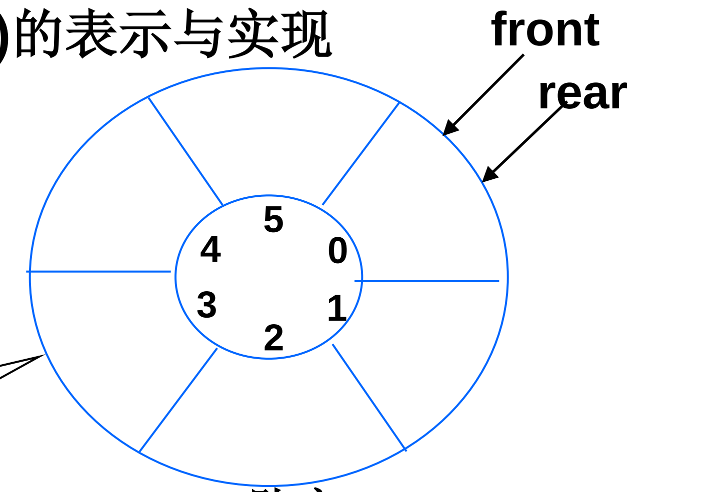
**If (Q.rear + 1 == MaxQueue) Q.rear = 0; else Q.rear++;**

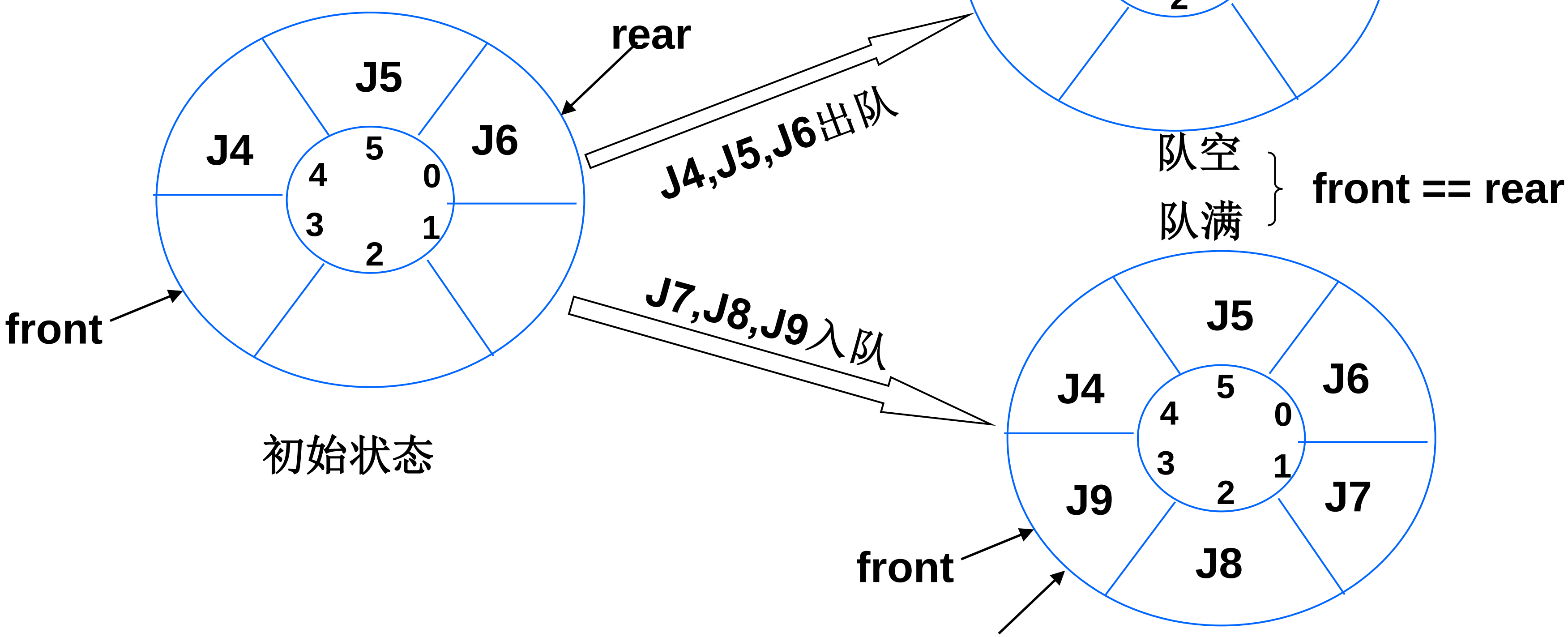
**Q.rear = (Q.rear + 1) % MaxQueue;**



- 队列顺序存储结构(contiguous queues)的表示与实现

循环队列队空、队列满解决方案




设 Queue Q;

1. 使用一个计数器记录队列中实际元素个数

队空: Q.count==0
队满: Q.connt==MaxQueue

2. 少用一个元素空间：

队空：Q.front == Q.rear

队满：(Q.rear+1)%MaxQueue == Q.front

"必须用有MaxQueue+1个元素的数组才能表示一个长度为MaxQueue的循环队列".

- 队列顺序存储结构(contiguous queues)的表示与实现

队列算法的描述---基于循环队列 P64~65

```
void InitialQueue(Queue *q) { q->front = q->rear = -1; }
```

```
boolean QueueEmpty(Queue *q) { return q->front == q->rear; }
```

```
boolean QueueFull(Queue *q) { return (q->rear+1)%MaxQueue == q->front; }
```

```
void EnQueue(QueueElem x, Queue *q)
{ if (QueueFull(q))  Error("Attempt to add an element to a full queue.");
  else { q->rear = (q->rear + 1)%MaxQueue; q->elem[q->rear] = x; } }
```

```
QueueElem DeQueue(Queue *q)
{ if (QueueEmpty(q))  Error("Attempt to delete an element from a empty queue.");
  else { q->front = (q->front + 1)%MaxQueue; return q->elem[q->front]; } }
```

```
QueueElem QueueFront (Queue *q)  //get the front element of the queue
{ if (QueueEmpty(q)) Error("The queue is empty.");
  else return q->elem[q->front + 1]; }
```

若约定队列的rear指示队尾元素的下一个空位置, front 指示队头元素, 则这些算法应如何调整?

初始化: q->front = 0; q->rear = 0; 判队空、队满: 不变化;
入队: q->elem[q->rear] = x; q->rear = (q->rear+ 1)%MaxQueue;
出队: x = q->elem[q->front]; q->front=(q->front+1)%MaxQueue;
return x;
取队首元素: q->elem[q->front];

- 队列的应用

**The Problem of Dancing Partner: 假设在周末舞会上, 男士们和女士们进入舞厅时各自排成一队. 跳舞开始时, 依次从男队和女队的队头上各出一人配成舞伴. 若两队初始人数不同, 则较长的那一队中未配对者需等待下一轮舞曲. 现要求写一算法模拟上述舞伴配对问题.**

**问题分析及算法思想: 该问题具有典型的先进先出的特征, 可用队列作为算法的数据结构. 在算法中假设男士和女士的记录存放在一个数组中作为输入, ①扫描该数组中各元素并根据其性别分别进入男队和女队; ②依次将两队列当前的队头元素出队配成舞伴, 直至某队列空为止; ③若某队仍有等待配对者, 算法将输出该队列的队头等待者的名字, 他/她将是下一轮舞曲开始时第一个可获得舞伴的人.**

```
typedef struc person {
    char *name;
    char sex;    // F --- female, M --- male
} Person;

typedef Person QueueElem;    // 将队列中元素的数据类型改为Person
```

- 队列的应用

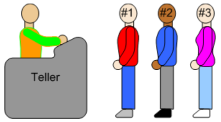
```
void DancingPartners(Person dancer[ ], int num) {
    Person p;
    Queue *Fdancers, *Mdancers; //定义两个循环队列

    InitQueue(*Fdancers); InitQueue(*Mdancers);  //队列初始化
    for ( i = 0; i < num; i++ )  //入队列
        if ( dancer[i].sex == 'F' )  EnQueue(dancer[i], Fdancers);
        else EnQueue(dancer[i], Mdancers);

    while ( !QueueEmpty(*Fdancers) && !QueueEmpty(*Mdancers) ) {
        p = DeQueue(*Fdancers);  printf("%s  ", p.name);
        p = DeQueue(*Mdancers);  printf("%s\n", p.name); }

    if ( !QueueEmpty(*Fdancers) )  { //输出女队剩下的队首女士名字
        p = QueueFront(*Fdancers);
        printf("%s  will be the first to get a partner.\n", p.name); }
    else if ( !QueueEmpty(*Mdancers) )  { //输出男队剩下的队首男士名字
            p = QueueFront(*Mdancers);
            printf("%s  will be the first to get a partner.\n", p.name); }
 }
```

队列的其它应用: 另见实习题



**Exercises (After class):**

1. **Write the C function QueueSize( ) for a linked queue that returns the number of items in it.**
2. 数组Q[1..n]表示一个循环队列, 设f的值为队列中第一个元素的位置, r的值为队列中实际队尾的位置加1, 并假定队列中最多只有n-1个元素, 则计算队列中元素个数的公式是什么?
3. 设栈 S = (1,2,3,4,5,6,7),其中7为栈顶元素.写出调用algo(&S)后栈的状态.

```
void algo(Stack *S)
{ int i=0;
  Queue Q; Stack T;
  InitialQueue(&Q); InitialStack(&T);
  while (!StackEmpty(S))
      if ( ( i = !i ) != 0 ) Push(&T, Pop(&S)); else EnQueue(&Q, Pop(&S));
  while (!QueueEmpty(Q)) Push(&S, DeQueue(&Q));
  while (!StackEmpty(T)) Push(&S, Pop(&T)); }
```

4. 假设有一循环队列中只设rear和queuelen来分别指示队尾元素的位置和队列中实际元素的个数. 试给出判断此循环队列的队空、队满条件, 并写出相应的入队和出队算法.