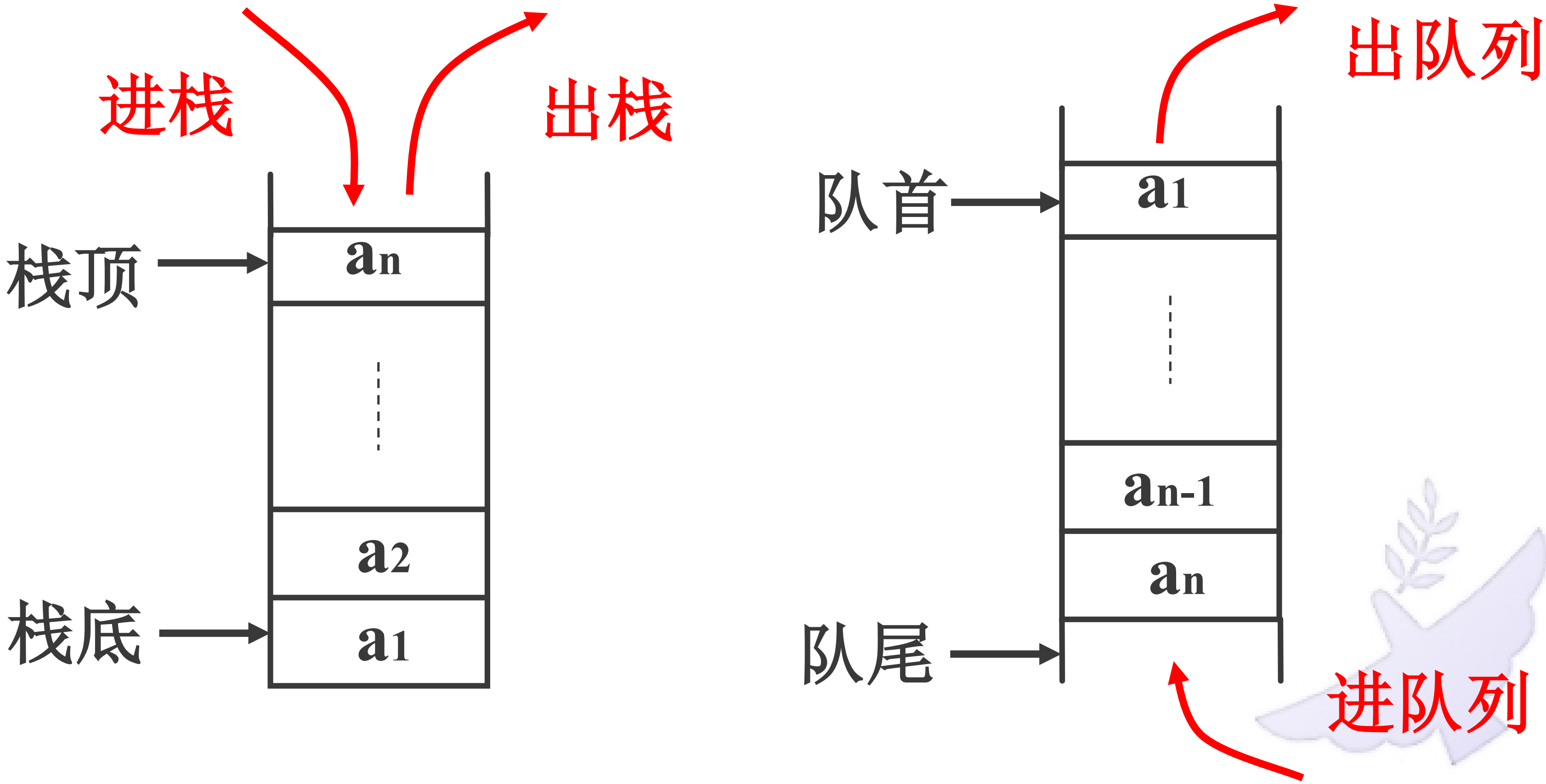
第三章 栈和队列

高春晓

北京理工大学

什么是栈？什么是队列？

- 栈和队列都是线性表
- 但是其操作是受限的

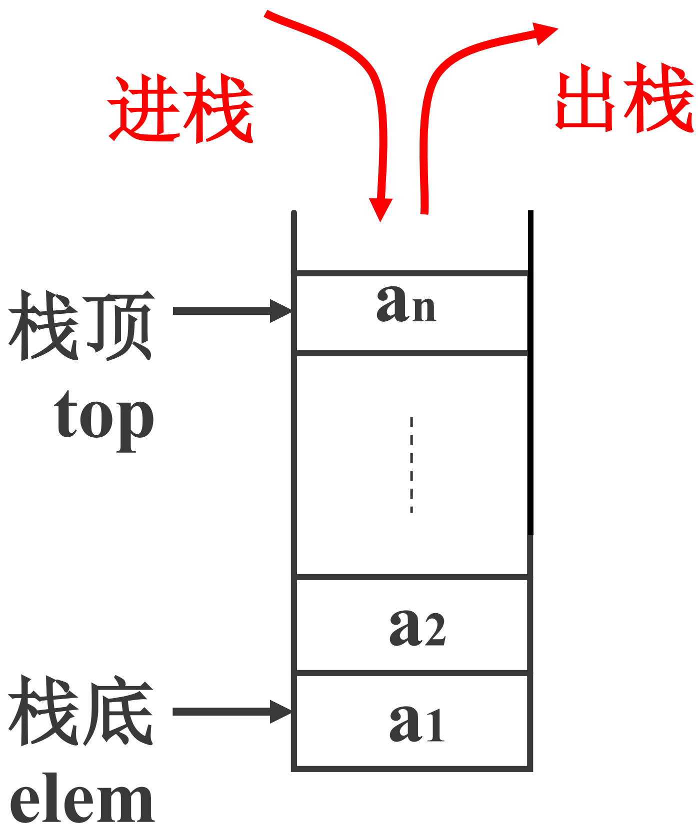


本章内容

- 3.1 栈
- 3.2 栈的实现
- 3.3 栈的应用
- 3.4 队列
- 3.5 离散事件模拟

3.1 栈

- **栈**：限定仅能在表尾一端进行插入、删除操作的线性表；
- **栈的特点**：后进先出（LIFO）



栈的应用-函数调用

- 当在一个函数的运行期间调用另一个函数时，在运行该被调用函数之前，需先完成三项任务：
  - 将所有的实参数、返回地址等信息传递给被调用函数保存；
  - 为被调用函数的局部变量分配存储区；
  - 将控制转移到被调用函数的入口。

栈的应用-函数调用

- 从被调用函数返回调用函数之前，应该完成下列三项任务：
  - 保存被调函数的计算结果；
  - 释放被调函数的数据区；
  - 依照被调函数保存的返回地址将控制转移到调用函数。

栈的应用-函数调用

- 多个函数嵌套调用的规则是：
  - 后调用先返回！
  - 此时的内存管理实行“**栈式管理**”

void main( ){	void a( ){	void b( ){
…	…	…
a( );	b( );	
…	…	
}//main	}// a	}// b

函数b的数据区

函数a的数据区

Main的数据区

3.1 栈的类型定义

ADT Stack {

数据对象：

$D = \{ a_i \mid a_i \in \text{ElemSet},\ i=1,2,\ldots,n,\ \ n \geq 0 \}$

数据关系：

$R1 = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D,\ i=2,\ldots,n \}$

约定a_n端为栈顶，a_1端为栈底

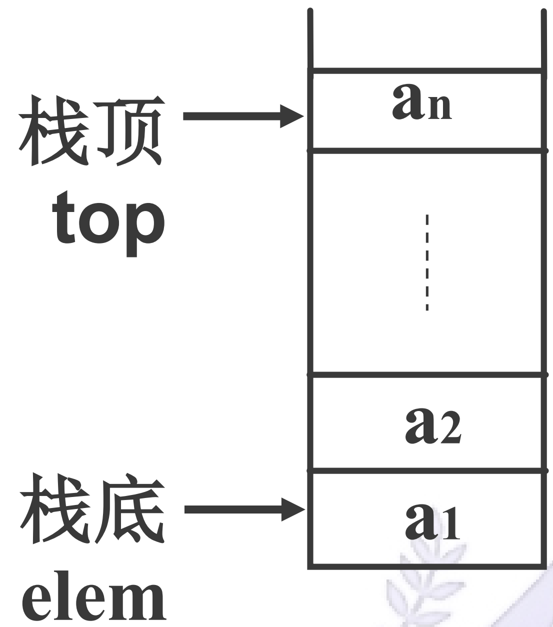
基本操作：……

} ADT Stack

栈的基本操作

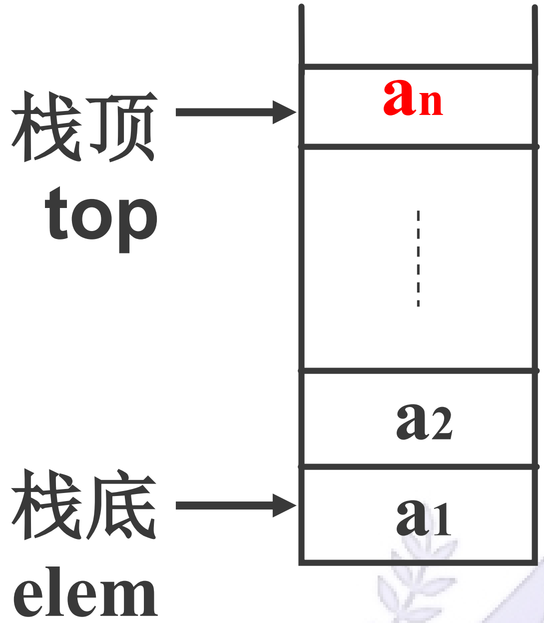
- 结构性操作

- **InitStack(&S)**
  - 操作结果：构造一个空栈S。
- **DestroyStack(&S)**
  - 初始条件：栈 S 已存在。
  - 操作结果：栈 S 被销毁。



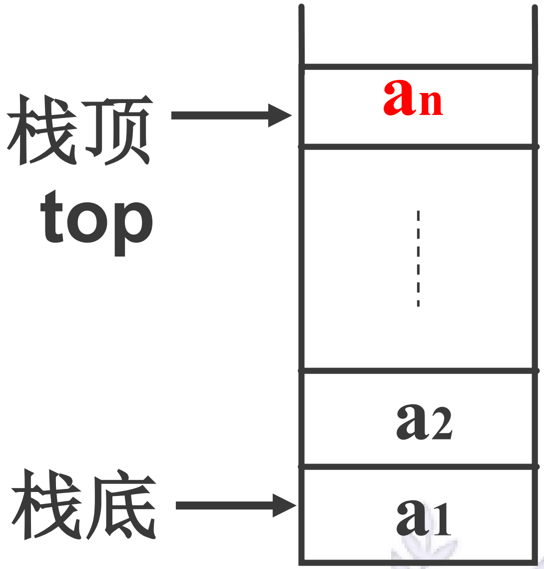
栈的基本操作

- **StackEmpty(S)**
  - 初始条件：栈 S 已存在。
  - 操作结果：若栈 S 为空栈，则返回 TRUE，否则 FALE。
- **StackLength(S)**
  - 初始条件：栈 S 已存在。
  - 操作结果：返回 S 的元素个数，即栈的长度。



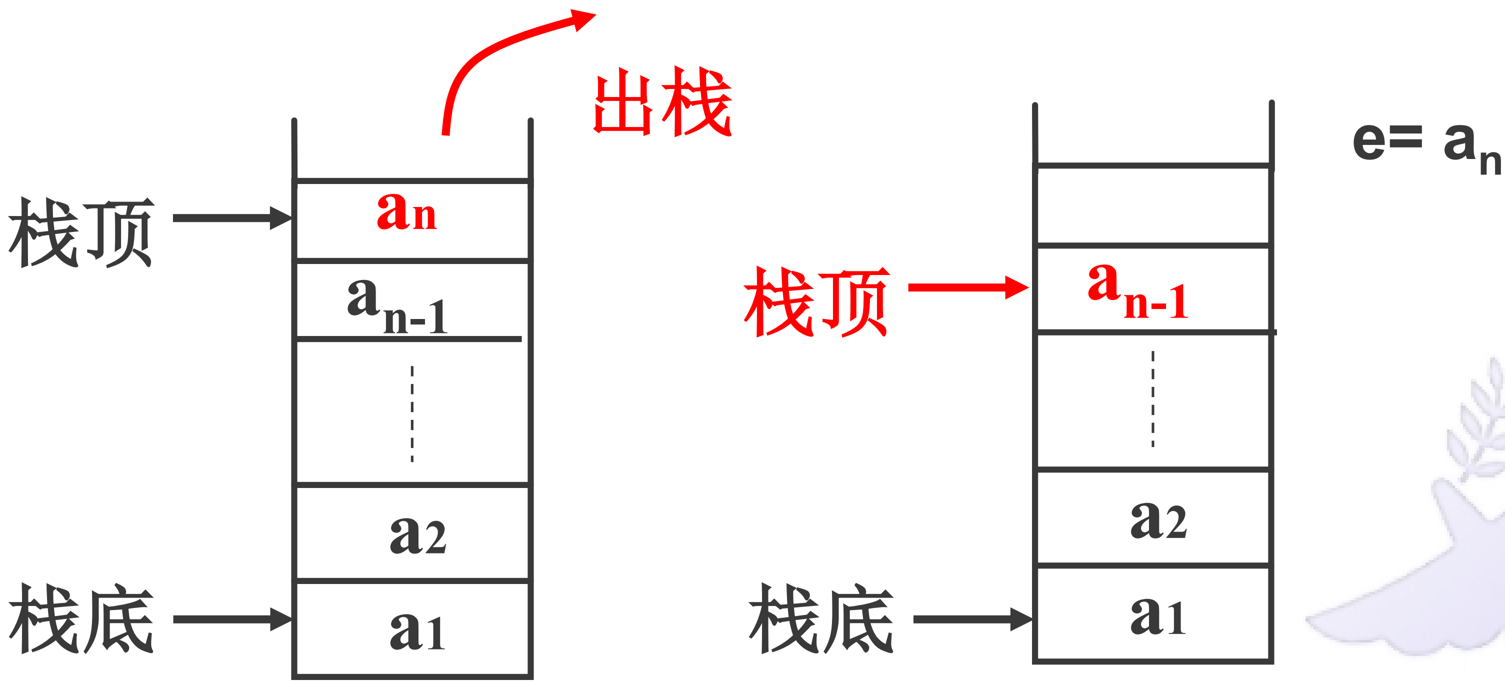
栈的基本操作

- ClearStack(&S)
  - 初始条件：栈 S 已存在。
  - 操作结果：将 S 清为空栈。
- GetTop(S, &e)
  - 初始条件：栈 S 已存在且非空。
  - 操作结果：用 e 返回 S 的栈顶元素。



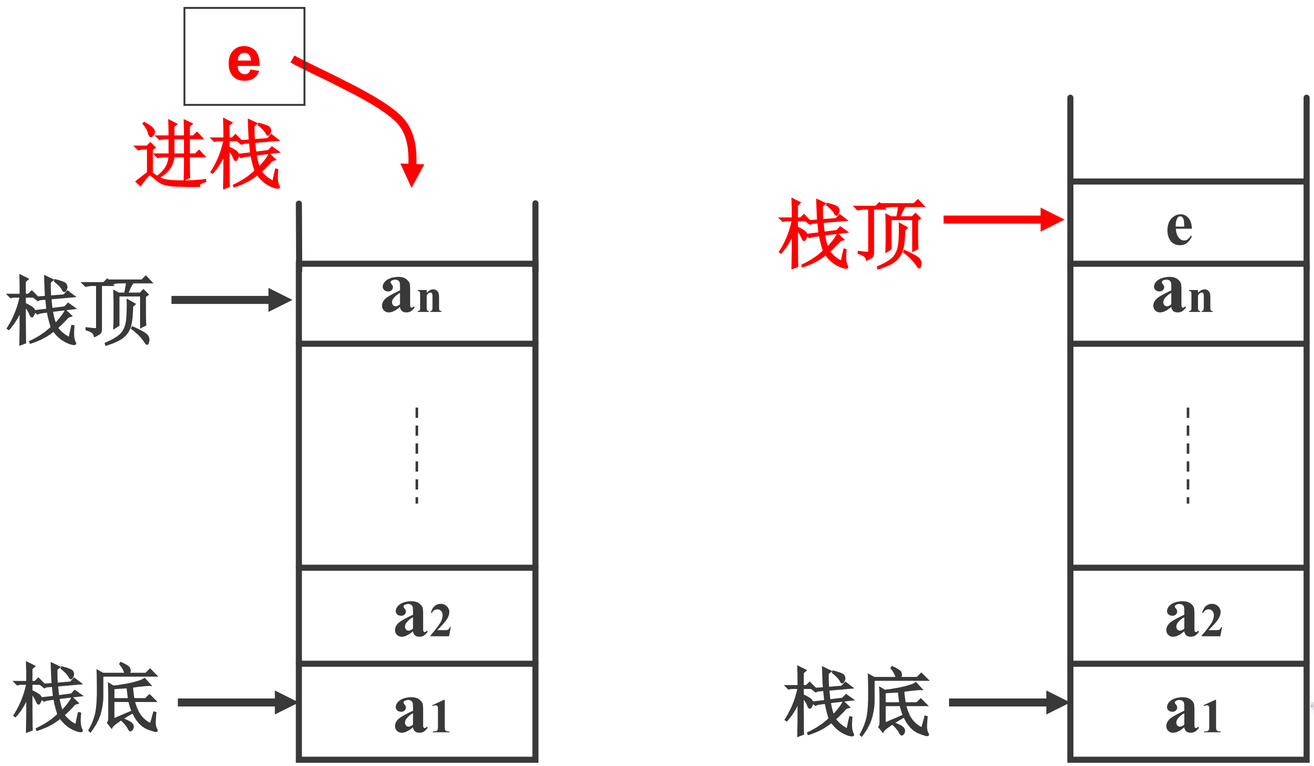
栈的基本操作

- **Pop(&S, &e)**
  - 初始条件：栈 S 已存在且非空。
  - 操作结果：删除 S 的栈顶元素，并用 e 返回其值。



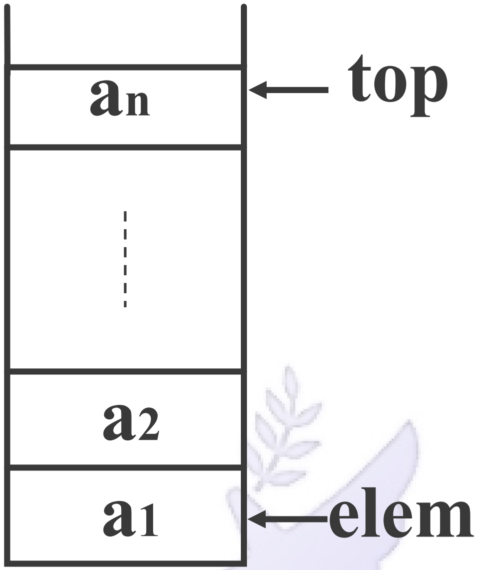
栈的基本操作

- **Push(&S, e)**
  - 初始条件：栈 S 已存在。
  - 操作结果：插入元素 e 为新的栈顶元素。



- 1）顺序栈一
  - 类似于线性表的顺序映象实现，指向表尾的指针可以作为栈顶指针。

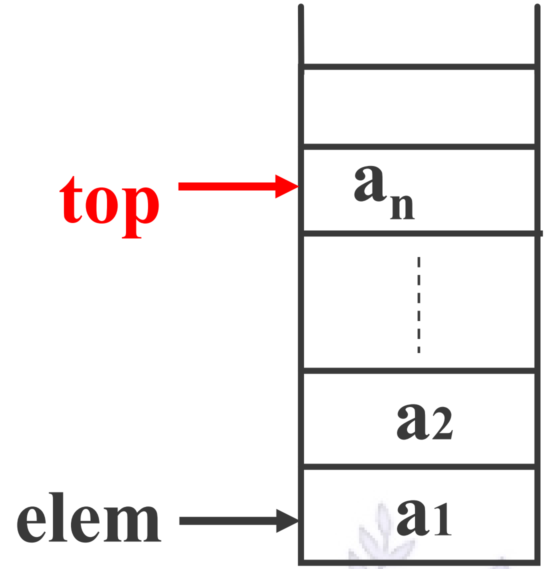
```
#define  STACK_INIT_SIZE  100
#define  STACKINCREMENT   10
typedef struct {
  SElemType  * elem;  //栈底
  int            top; //栈顶
  int  maxSize;    //栈的大小
} SeqStack;
```



1）顺序栈的实现

```
#define  STACK_INIT_SIZE  100
#define  STACKINCREMENT   10
typedef struct {
  SElemType  * elem;  //栈底
  int        top; //栈顶
  int  maxSize;    //栈的大小
} SeqStack;
```

- **InitStack (SeqStack &S)**
  - 申请空间
  - 设置参数：S.top = -1;...
- **Push (SeqStack &S, SElemType e)**
  - S.top++;S.elem[S.top] = e;
  - 但要首先判断栈是否已满
  - if (S.top >= S.maxSize-1)
- **Pop (SeqStack &S, SElemType &e)**
  - e = S.elem[S.top]; S.top--;
  - 但要先判断栈是否空:
  - if (S.top == -1)

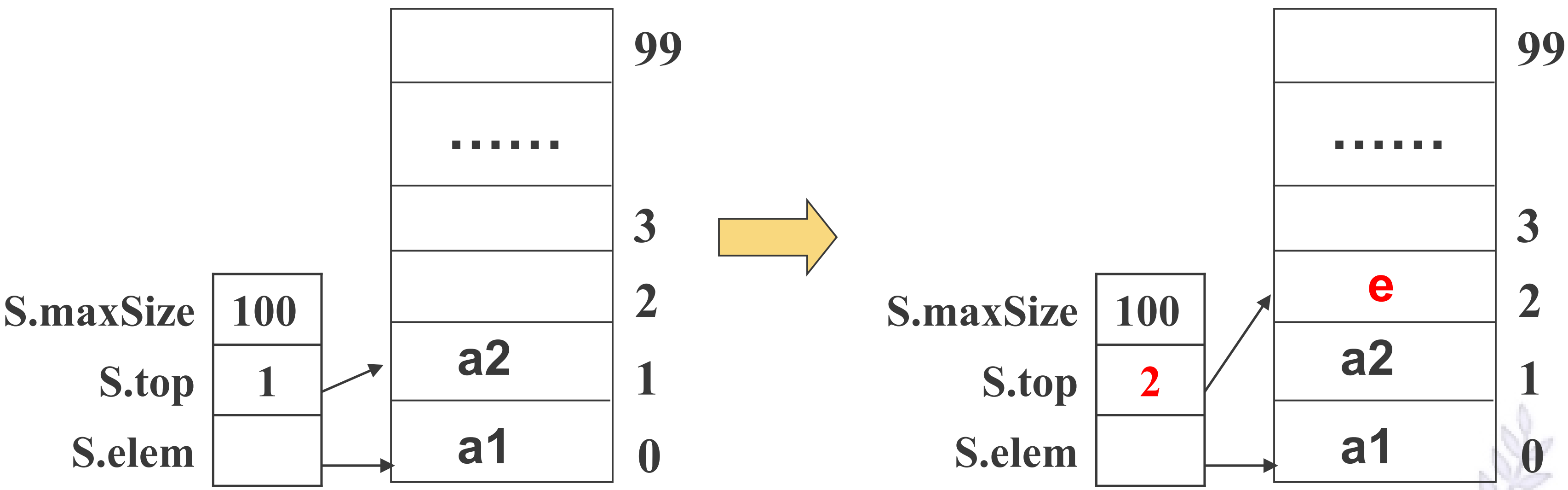


InitStack (SeqStack &S)

```
Status InitStack (SeqStack &S)
{// 构造一个空栈S

    S.elem=(SElemType*)malloc(
            STACK_INIT_SIZE*sizeof(ElemType));
     if (S.elem==NULL) {
         printf (“存储分配失败!\n”);  exit(1);}
     S.top = -1;
     S.maxSize = STACK_INIT_SIZE;
     return OK;

}// InitStack
```

Push (SeqStack &S, SElemType e)

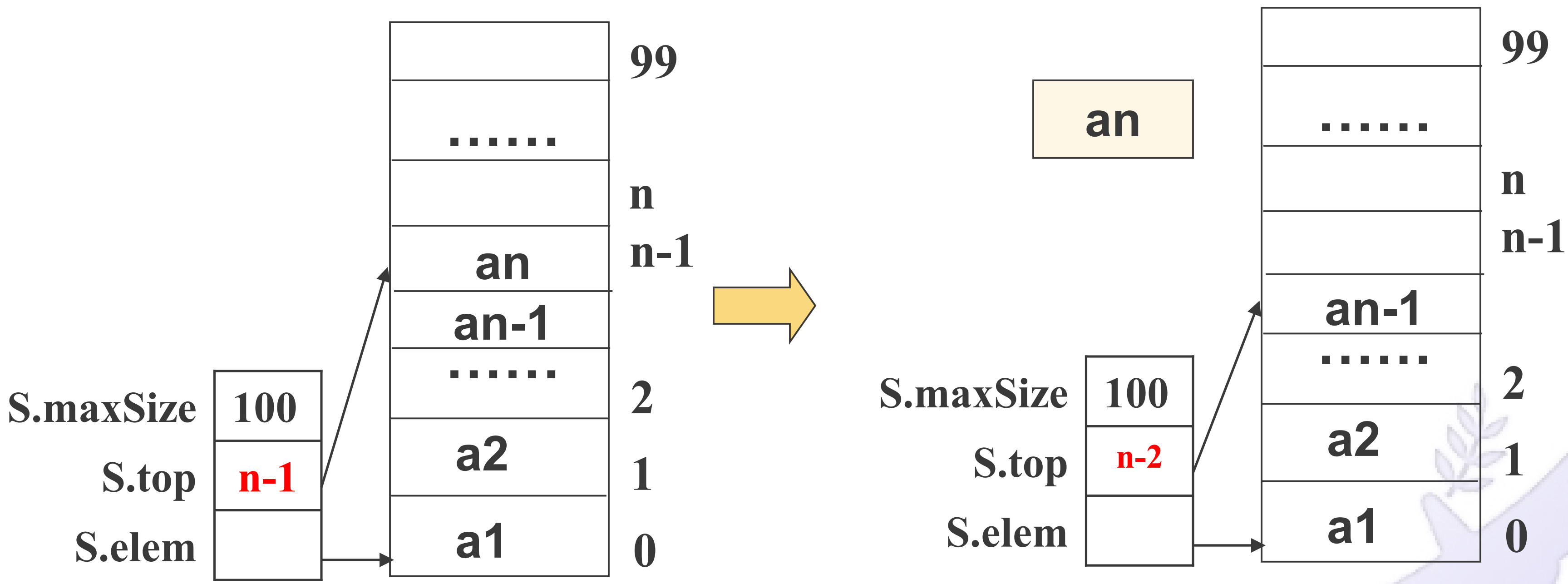
- **功能：** 元素 e 进栈。



```
Status Push (SeqStack &S, SElemType e) {
  if (S.top >= S.maxSize-1) {//栈满，追加存储空间
      S.elem = (SElemType *) realloc ( S.elem,
                  (S.maxSize + STACKINCREMENT) *
                                      sizeof (SElemType));
      if (!S.elem) exit (OVERFLOW); //存储分配失败
      S.maxSize += STACKINCREMENT;
  }
    S.top++;S.elem[S.top] = e;
    return OK;
}//Push
```

Pop (SeqStack &S, SElemType &e)

- 出栈操作
- 功能：栈顶元素退栈，并用 e 返回。

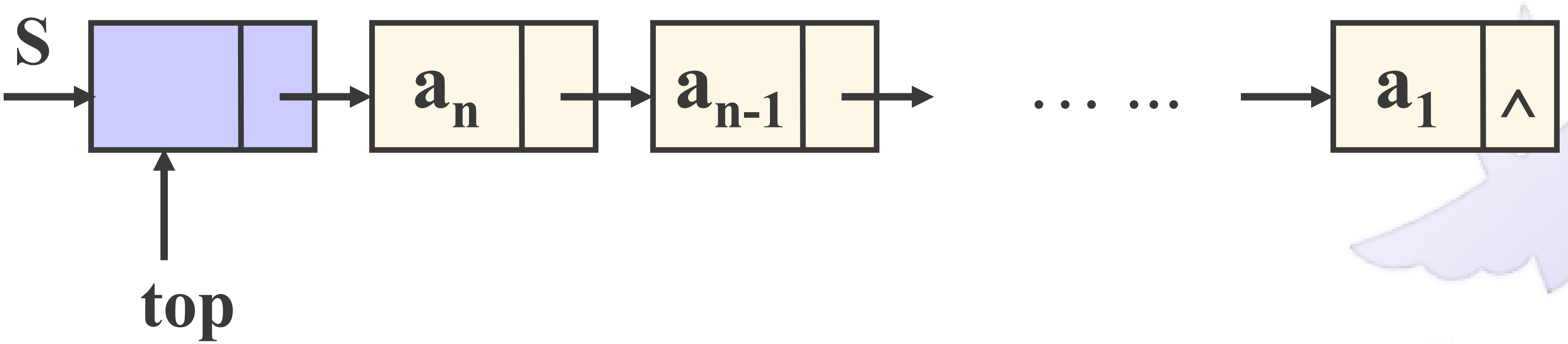


出栈操作

```
Status Pop (SeqStack &S, SElemType &e) {
    // 若栈不空，则删除S的栈顶元素，
    // 用e返回其值，并返回OK；
    // 否则返回ERROR
    if (S.top == -1) return ERROR;
    e = S.elem[S.top];  S.top--;
    return OK;
}//Pop
```

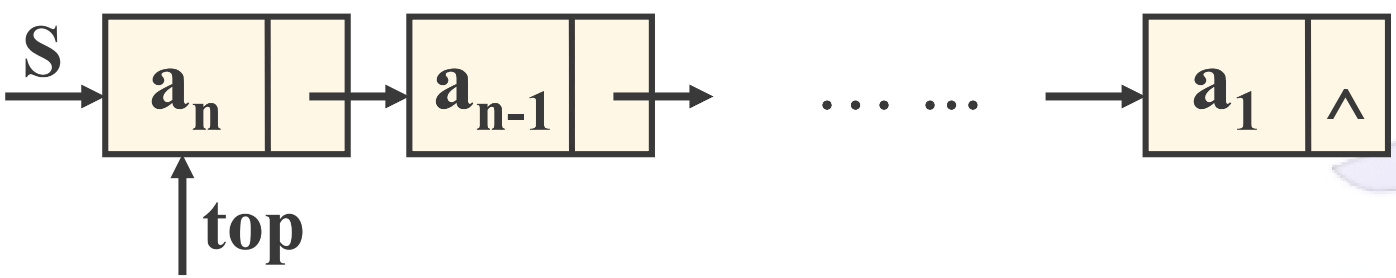
2）链栈的实现

- 实现方式一：带头结点的单链表
- InitStack (LinkStack &S)
  - 创建头结点
- Push (LinkStack &S, SElemType e)
  - 创建新节点，并插入到头节点之后
- Pop (LinkStack &S, SElemType &e)
  - 先判断栈是否为空，若不为空则删除第一个结点



2）链栈的实现

- 实现方式二：不带头结点的单链表
- InitStack (LinkStack &S)
  - S = NULL;
- Push (LinkStack &S, SElemType e)
  - 创建新节点，插入第一个结点之前，并更新栈指针
- Pop (LinkStack &S, SElemType &e)
  - 先判断栈是否为空，若不为空则删除第一个结点，并更新栈指针



小 结

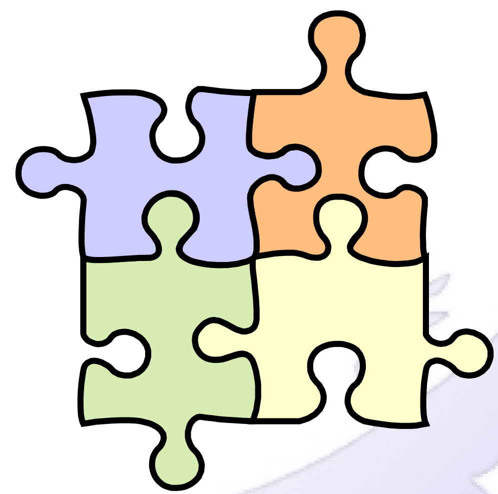
1、栈是限定仅能在表尾一端进行插入、删除操作的线性表

2、栈的元素具有后进先出的特点

3、栈顶元素的位置由一个称为栈顶指针的变量指示，进栈和出栈操作都要修改栈顶指针

3.3 栈的应用

- 例一、 数制转换
- 例二、 括号匹配的检验
- 例三、 行编辑程序问题
- 例四、 迷宫求解
- 例五、 表达式求值
- 例六、 实现递归



例一、 数制转换

- 算法基于原理：
  - $N = (N \text{ div } d) \times d + N \text{ mod } d$
- 例如： $(1348)_{10} = (?)_8$

计算顺序

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

输出顺序

例一、数制转换

- 算法思路：
  - 用栈实现先计算，后输出
- 操作步骤

1. 初始化栈
   - InitStack(S);
2. 依次计算当前的d进制数，并将其压栈
   - Push(S, N % 8); N = N/8;
3. 依次从栈中取出计算的结果，并输出
   - Pop(S, e);
4. 删除栈 DestroyStack(S);

```
void conversion () {
InitStack(S); //步骤1：初始化栈
    scanf ("%d",N);
    while (N !=0 ) {  //步骤2：余数压栈
        Push(S, N % 8);
        N = N/8;
    }
    while (!StackEmpty(S)) { //步骤3:出栈并输出
        Pop(S,e);
        printf ( "%d", e );
    }
    DestroyStack(S); // 步骤4：删除栈
} // conversion
```

例二、 括号匹配的检验

- 正确的格式：
  - （[]（））
  - [（[][]）]
- 错误的格式：
  - [（]）
  - （[（））或（（）]）
- 判断是否正确方法
  - 检验括号是否匹配
  - 匹配：按照**最近匹配原则**

例二、 括号匹配的检验

- 算法的设计思想：

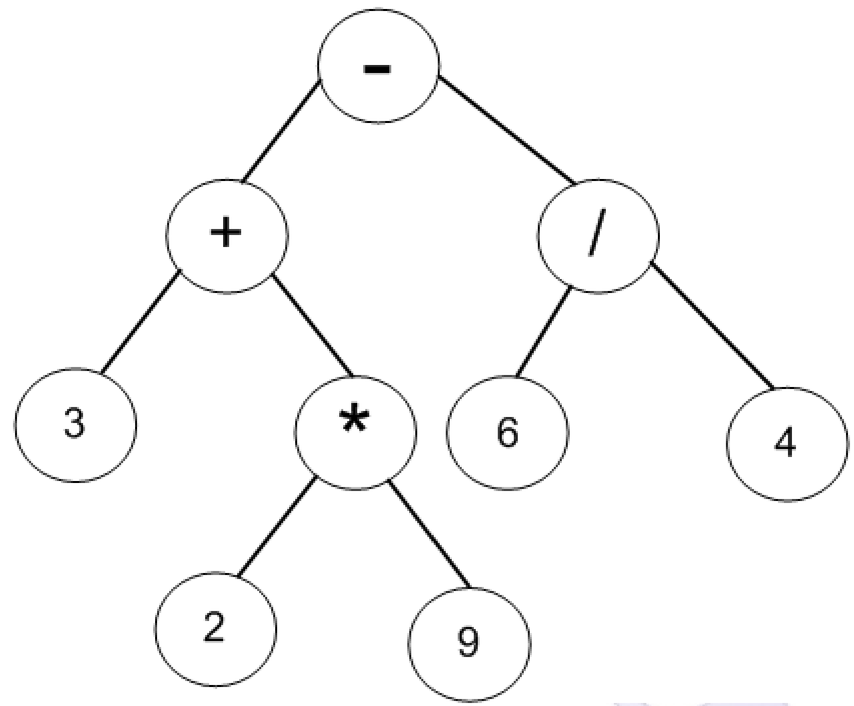
1. 凡出现左括弧，则进栈；
2. 凡出现右括弧，首先检查栈是否空
   - 若栈空，则表明该“右括弧”多余，
   - 否则和栈顶元素比较，
   - 若相匹配，则“左括弧出栈”，
   - 否则表明不匹配。
3. 表达式检验结束时：
   1. 若栈空，则表明表达式中匹配正确；
   2. 否则表明“左括弧”有余。

例五、 表达式求值

- 限于二元运算符的表达式定义:
  - 表达式 ::= (操作数) + (运算符) + (操作数)
  - 操作数(operand) ::= 简单变量 | 表达式
  - 简单变量 :: = 标识符 | 无符号整数
  - 运算符(operator)
- 表达式的三种标识方法：**Exp = S1 + OP + S2**
  - **OP + S1 + S2** 为前缀表示法 –波兰表示法
  - **S1 + OP + S2** 为中缀表示法
  - **S1 + S2 + OP** 为后缀表示法-逆波兰表示法
  - （**RPN**：Reverse Polish notation，波兰逻辑学家J.Lukasiewicz于1929年提出。）

表达式的三种标识方法

- 中缀表达式：**3 + 2*9 – 6 / 4**
- 后缀表达式：**329*+64/-**
- 前缀表达式：**-+3*29/64**



表达式树

应用后缀表示计算表达式的值

- 从左向右顺序地扫描表达式，并用一个栈暂存扫描到的操作数或计算结果。
- 在后缀表达式的计算顺序中已隐含了加括号的优先次序，括号在后缀表达式中不出现。
- 中缀表达式：3 + 2*9 – 6 / 4
- 后缀表达式：3 2 9 * + 6 4 / -

rst_1 rst_3

rst_2

rst_4

应用后缀表示计算表达式的值

- 基本思想：
- 顺序扫描表达式的每一项，根据它的类型做如下相应操作：
  a) 若该项是**操作数**，则将其**压栈**；
  b) 若该项是**操作符<op>**，则**连续从栈中退出两个操作数Y和X**，**形成运算指令X<op>Y**，**并将计算结果重新压栈**。
- 当表达式的所有项都扫描并处理完后，栈顶存放的就是最后的计算结果。

后缀表达式：3 2 9 * + 6 4 / -

利用栈将中缀表示转换为后缀表示

- 使用栈可将表达式的中缀表示转换成它的前缀表示和后缀表示。
- 为了实现这种转换，需要考虑各操作符的优先级。

C/C++中操作符的优先级

优先级	1	2	3	4	5	6	7
操作符	单目 -, !	*, /, %	+, -	<, <=, >, >=	==, !=	&&	||

利用栈将中缀表示转换为后缀表示

- 举例：中缀表达式：#a + b * (c - d) – e / f #
- 后缀表达式：a b c d - * + e f / -

$$\underbrace{\underbrace{a + \underbrace{b * \underbrace{(c - d)}_{rst1}}_{rst2}}_{rst3} - \underbrace{e / f}_{rst4}}_{rst5}$$

操作符ch	#	(	*,/,%	+,-	)
优先级	0	6	4	2	1

各个算术操作符的优先级

#a + b * (c - d) – e / f#

操作符ch	#	(	*,/,%	+,-	)
isp(栈内)	0	1	5	3	6
icp(栈外)	0	6	4	2	1

- isp 叫做栈内 (in stack priority) 优先数
- icp 叫做栈外 (in coming priority) 优先数。
- 操作符优先数相等的情况只出现在括号配对或栈底的 "#" 号与输入流最后的 "#"号配对时。

中缀表达式转换为后缀 # a + b * (c - d) – e / f

- 算法步骤：
- 1. 操作符栈初始化，将结束符'#' 进栈。然后令当前字符ch为中缀表达式字符流的首字符 。
- 2. 重复执行以下步骤，直到 ch = '#'，同时栈顶的操作符也是'#'，停止循环。
  - 2.1 若ch是操作数，直接输出并读入下个字符 ch.
  - 2.2 若ch是操作符，判断 ch 的优先级 icp 和位于栈顶的操作符op的优先级 isp:
    - 2.2.1 icp (ch) > isp (op)
    - 2.2.2 icp (ch) < isp (op)
    - 2.2.3 icp (ch) == isp (op)

中缀表达式转换为后缀 # a + b * (c - d) – e / f

- 算法步骤（续）：
- 2.2 若ch是操作符，判断 ch 的优先级 icp 和位于栈顶的操作符op的优先级 isp:
  - 2.2.1 icp (ch) > isp (op): 令ch进栈，读入下一个字符ch。（看后面是否有更高的）
  - 2.2.2 icp (ch) < isp (op)，退栈并输出。(执行先前保存在栈内的优先级高的操作符)
  - 2.2.3 icp (ch) == isp (op)，退栈但不输出，若退出的是 "("号读入下一个字符ch。（销括号）
- 算法结束, 输出序列即为所需的后缀表达式。

示例

操作符ch	#	(	*,/,%	+,-	)
isp(栈内)	0	1	5	3	6
icp(栈外)	0	6	4	2	1

- isp 叫做栈内 (in stack priority) 优先数
- icp 叫做栈外 (in coming priority) 优先数。
- 操作符优先数相等的情况只出现在括号配对或栈底的 “#”号与输入流最后的 “#”号配对时。

#a + b * (c - d) – e / f #

后缀表达式：a b c d - * + e f / -

应用中缀表示计算表达式的值

$$\underbrace{\underbrace{a + \underbrace{b * \underbrace{(c - d)}_{rst1}}_{rst2}}_{rst3} - \underbrace{e / f}_{rst4}}_{rst5}$$

- 使用两个栈，**操作符栈OPTR (operator)**，**操作数栈OPND(operand)**

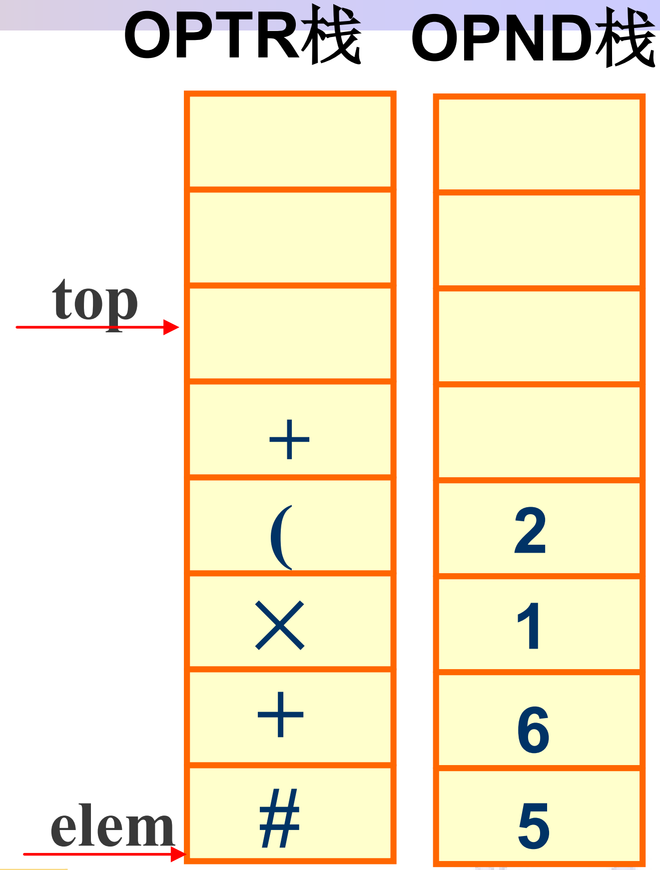
算符优先级的另一种表示：优先级关系表

$\theta_1 < \theta_2$ $\theta_1 = \theta_2$ $\theta_1 > \theta_2$

$\theta_1 \backslash \theta_2$	+	-	×	/	(	)	#
+	>	>	<	<	<	>	>
-	>	>	<	<	<	>	>
×	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

表达式求值—基本操作

- 从左向右扫描表达式：
- # 5+6 ×(1+2)－4 #
- 遇操作数：保存在操作数栈；
- 遇运算符号 θ_2：与前面运算符θ_1比较
  - 若 $\theta_1 < \theta_2$则保存θ_2到运算符栈；
  - 若 $\theta_1 > \theta_2$ 则θ_1可进行运算；
  - 若 $\theta_1 = \theta_2$ 需要消去括号；

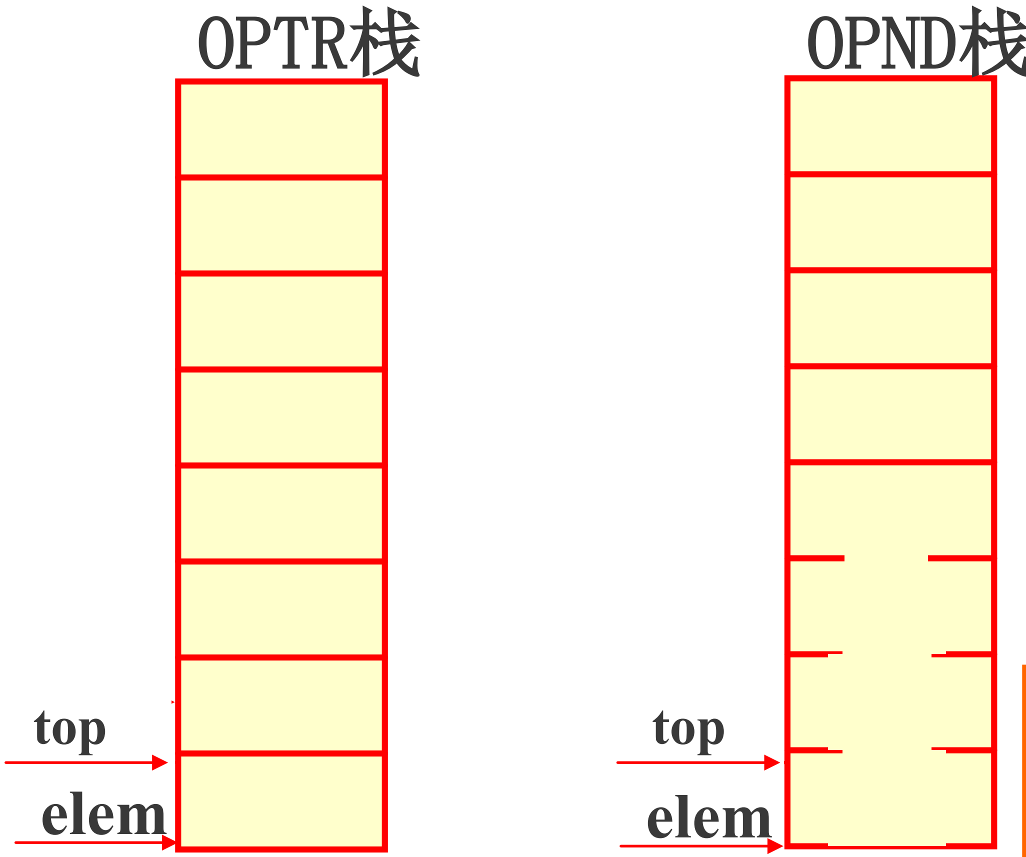


$\theta_1 \backslash \theta_2$	+	-	×	/	（	）	#
+	>	>	<	<	<	>	>
×	>	>	>	>	<	>	>

OPTR栈

表达式求值示意图（算符优先算法）

读入表达式：# 5 +6 ×(1+ 2)−4 # =19



1+2=3

6×3=18

5+18=23

23-4=19

程序结束标志？

OPTR栈顶元素为#
当前字符也为#

栈的应用：递归

- 递归函数：直接或者间接调用自身的函数
- 递归调用有两种方式:
  - 直接调用其本身
  - 通过其他函数间接地调用。

```
int  f(int x)
{    int y,z;
       ......
     z=f(y);
     .......
     return(2*z);
}
```

```
int  f1(int x)
{    int y,z;
       ......
     z=f2(y);
     .......
     return(2*z);
}
```

```
int  f2(int t)
{    int a,c;
       ......
     c=f1(a);
     .......
     return(3+c);
}
```

递归程序特点

- 通常以递归形式定义算法与非递归算法比较，算法结构会更紧凑、清晰;
- 但是，递归函数在递归调用过程中，会占用更多的内存空间和需更多的运行时间;
- 它必须满足以下两个条件：
  - 1) 在每一次调用自己时，必须是（在某种意义上）更接近于解;
  - 2) 必须有一个终止条件。
  - 防止振荡式的相互递归调用，否则会产生无限递归调用现象;

- 一个问题是否可以转换为递归来处理必须满足以下条件：
  - （1）必须包含一种或多种非递归的基本形式；
  - （2）一般形式必须能最终转换到基本形式；
  - （3）由基本形式来结束递归。

递归求解的问题

- 在以下三种情况下，常常用到递归方法
  - 1）定义是递归的
  - 2）数据结构是递归的
  - 3）问题的解法是递归的

1）定义是递归的

定义递归问题：

(1) 写出递归公式；

(2) 将递归公式函数化；

例1 求n！

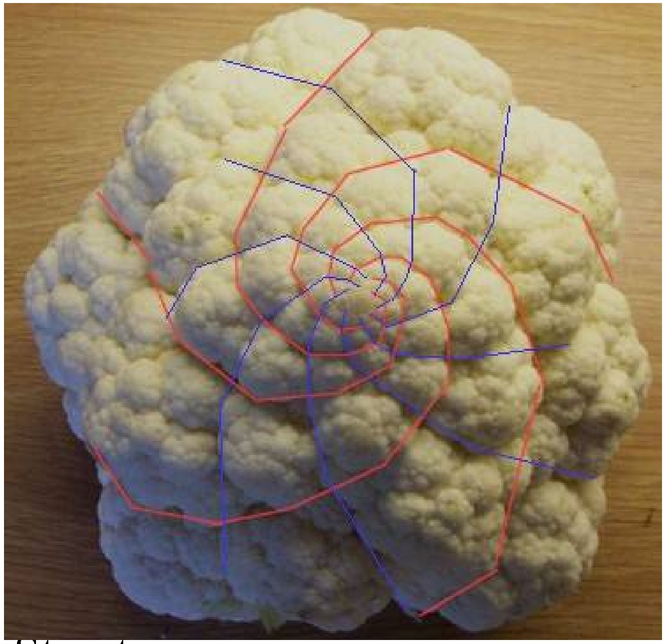
$$n! = \begin{cases} 1 & (n = 0,1) \\ n \cdot (n-1)! & (n > 1) \end{cases}$$

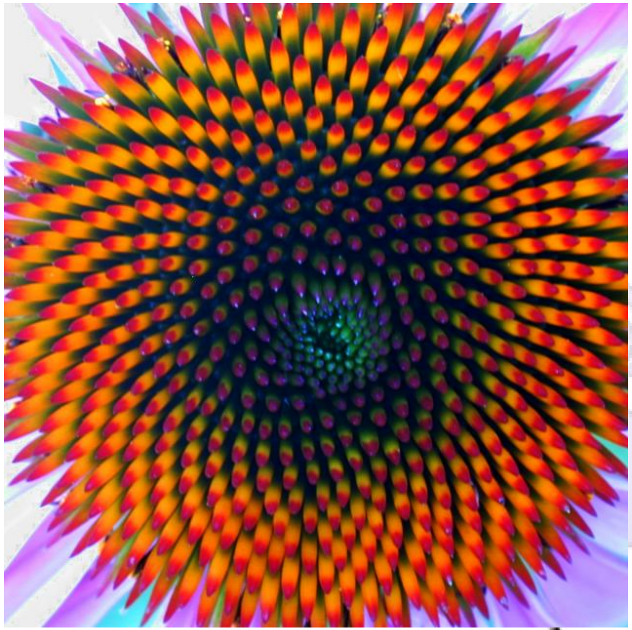
```
int fac(int n)
{   int f;
    if(n<0)  printf("n<0,data error!");
    else if(n==0||n==1)  f=1;
    else f=fac(n-1)*n;
    return(f);
}
```

1）定义是递归的

- 例2：斐波那契数列
- 1，1，2，3，5，8，13，21……
  - F(1)=F(2)=1,
  - F(n)=F(n-1)+F(n-2) (n≥3)

```
long Fib（long n）//递归程序
{
      If (n==1 || n==2) return 1;//终止条件及处理
      Else return Fib(n-1) + Fib(n-2);//递归处理
}
```



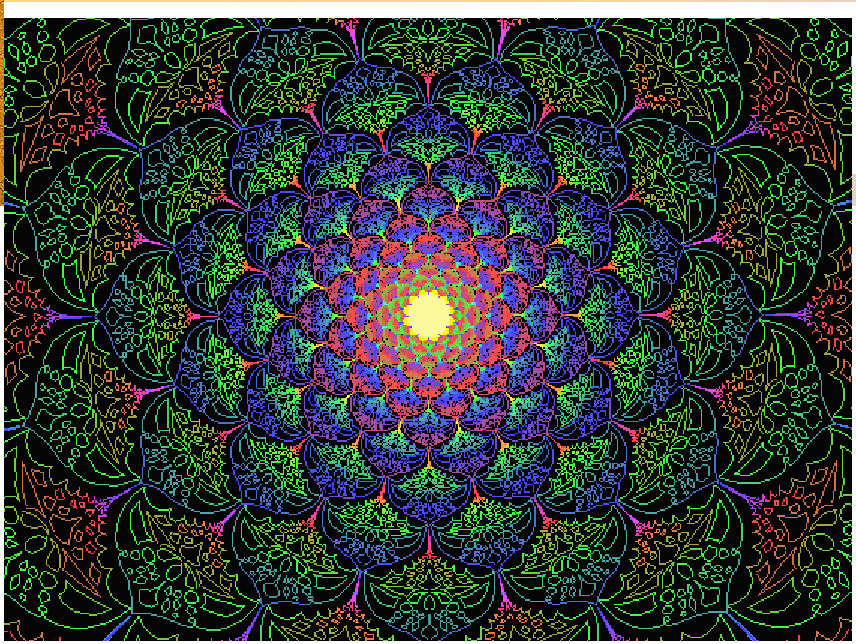


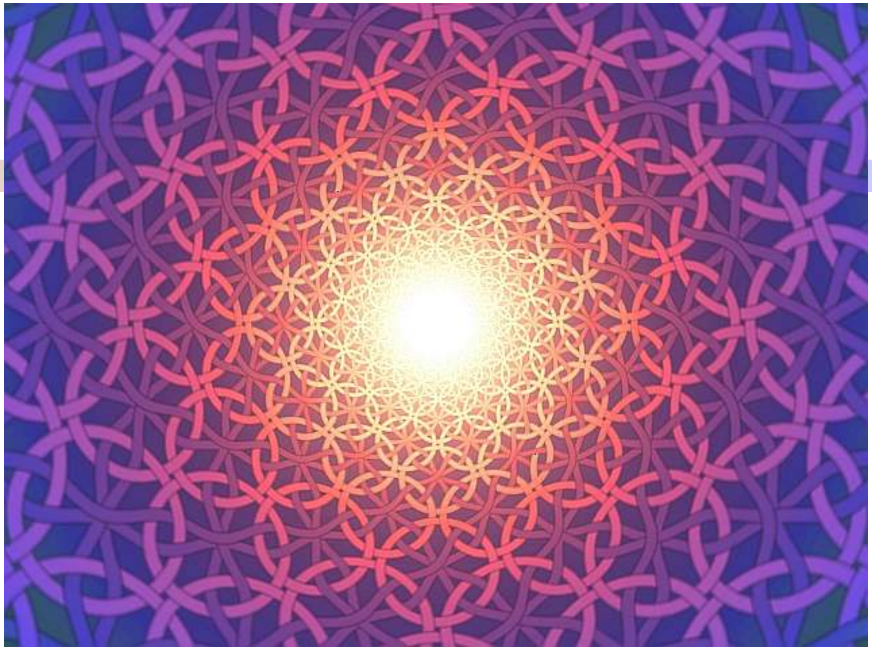
例2：斐波那契数列

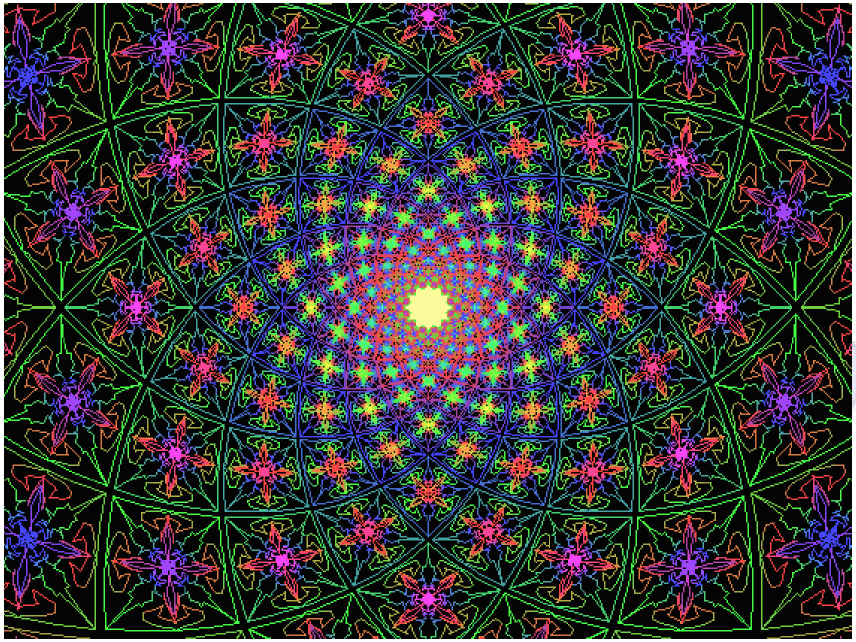
```
long CalFib (long n) //非递归程序方法1
{
  if(n < = 2)  return 1;
  else
  {
      long f1 = 1, f2 = 1, f = 0;
      for ( int i = 3; i <= n; i++)
            { f = f1+f2; f2 = f1; f1 = f; }
       return f;
   }
}
```

- 方法2：求出通项公式

$$F(n)=\frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n}-\left(\frac{1-\sqrt{5}}{2}\right)^{n}\right)(n\geq 1)$$







应用

- 如果一对大兔子每月能生一对小兔，而每对小兔在出生后的第三个月又能生一对小兔。问由一对小兔开始，*n*个月后将有多少对兔子？

1	2	3	4	5	6	7	8	9	10	11	12
S1	M1	L1 S1	L1 M1 S1	L2 M1 S2	L3 M2 S3	L5 M3 S5	L8 M5 S8	L13 M8 S13	L21 M13 S21	L34 M21 S34	L55 M34 S55
1	1	2	3	5	8	13	21	34	55	89	144

F(1)=F(2)=1,
F(n)=F(n-1)+F(n-2) (n≥3)

例3：整数划分

- 指把一个正整数n写成多个大于等于1且小于等于其本身的整数的和，则其中各加数所构成的集合为n的一个划分。
- n=6

{6}
{5,1}
{4,2}, {4,1,1}
{3,3},{3,2,1},{3,1,1,1}
{2,2,2},{2,2,1,1},{2,1,1,1,1}
{1,1,1,1,1,1}

例3：整数划分

- 令q(n,m)表示：n的划分个数，划分中最大数不超过m
- 根据n和m的关系，考虑以下几种情况：
  - n=1: 只有一种划分{1}。
  - m=1: 只有一种划分{1,1,……,1}
  - n
  - n

$$q(n,m)=\begin{cases}1 & ,n=1|m=1\\ q(n,n) & ,n<m\\ 1+q(n,m-1) & ,n=m\\ q(n,m-1)+q(n-m,m) & ,n>m\end{cases}$$

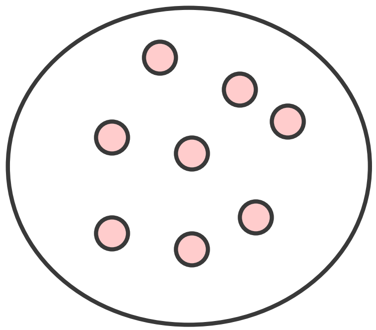
    - 划分中不包含n: 即m=n-1: q(n,m-1)
  - n>m: 两种划分：
    - 划分中包含m: {m, x_1, x_2, …x_i}，q(n-m,m)
    - 划分中不包含m: 即m=n-1: q(n,m-1)

例3：整数划分

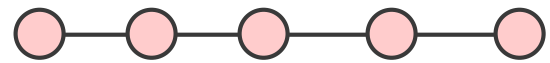
```
int q(int n,int m){
    if(n<1||m<1) return 0; //若正整数或最大加数小于1，则返回0
        //若正整数或最大加数等于1，则划分个数为1（n个1相加）
    if(n==1||m==1) return 1;
        //若大于则划分个数等于最大加数为n的划分个数
    if(n<m) return q(n,n);
        //若正整数等于最大加数，则划分个数等于
    if (n==m) return 1+q(n,n-1);
        //若正整数大于最大加数 返回如下划分结果
    return q(n,m-1)+q(n-m,m);
}
void main(){// 测试
     q(4,4) //返回结果为 5
}
```

2）数据结构是递归的

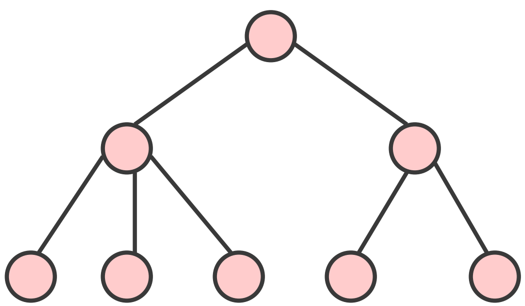
- 递归结构：



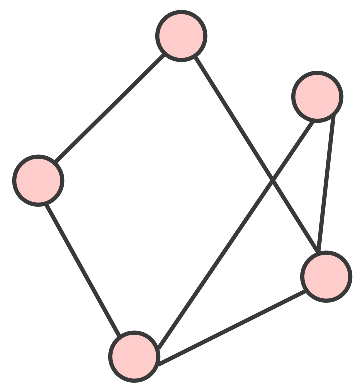
a. 集合关系



b. 线性关系



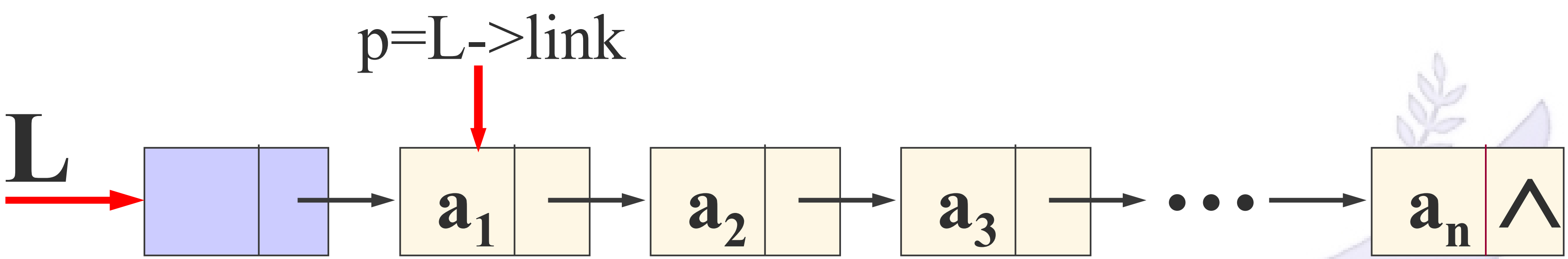
c. 树型关系



d. 图型关系

2）数据结构是递归的

- 打印单链表中所有数据元素
- 基本思想：
  - 1) 单链表是一种顺序结构，必须从第一个结点起，逐个检查每个结点的数据元素；
  - 2) 从另一角度看，链表又是一个递归结构
  - 若 L 是线性链表 (a_1, a_2, …, a_n) 的头指针
  - 则 L->link是线性链表 (a_2, …, a_n)的头指针。



◆ 打印单链表中所有数据元素

```
void printValue ( LinkNode *L ) {
    if ( L!= NULL) {   //递归结束条件
        printf ( L ->data );     //打印当前结点的值
        PrintValue ( L->link );   //递归打印后续链表
    }
}
```

3）问题的解法是递归的

例1、求两个数的最大公约数的数学模型

Greatest Common Divisor 辗转相除法

$$\mathrm{gcd(a,b)}=\begin{cases} b & (\mathrm{a\%b==0}) \\ \mathrm{gcd}(b, a\%b) & (\mathrm{a\%b!=0}) \end{cases}$$

例：求511和292的GCD

511÷292＝1余219

292÷219＝1余73

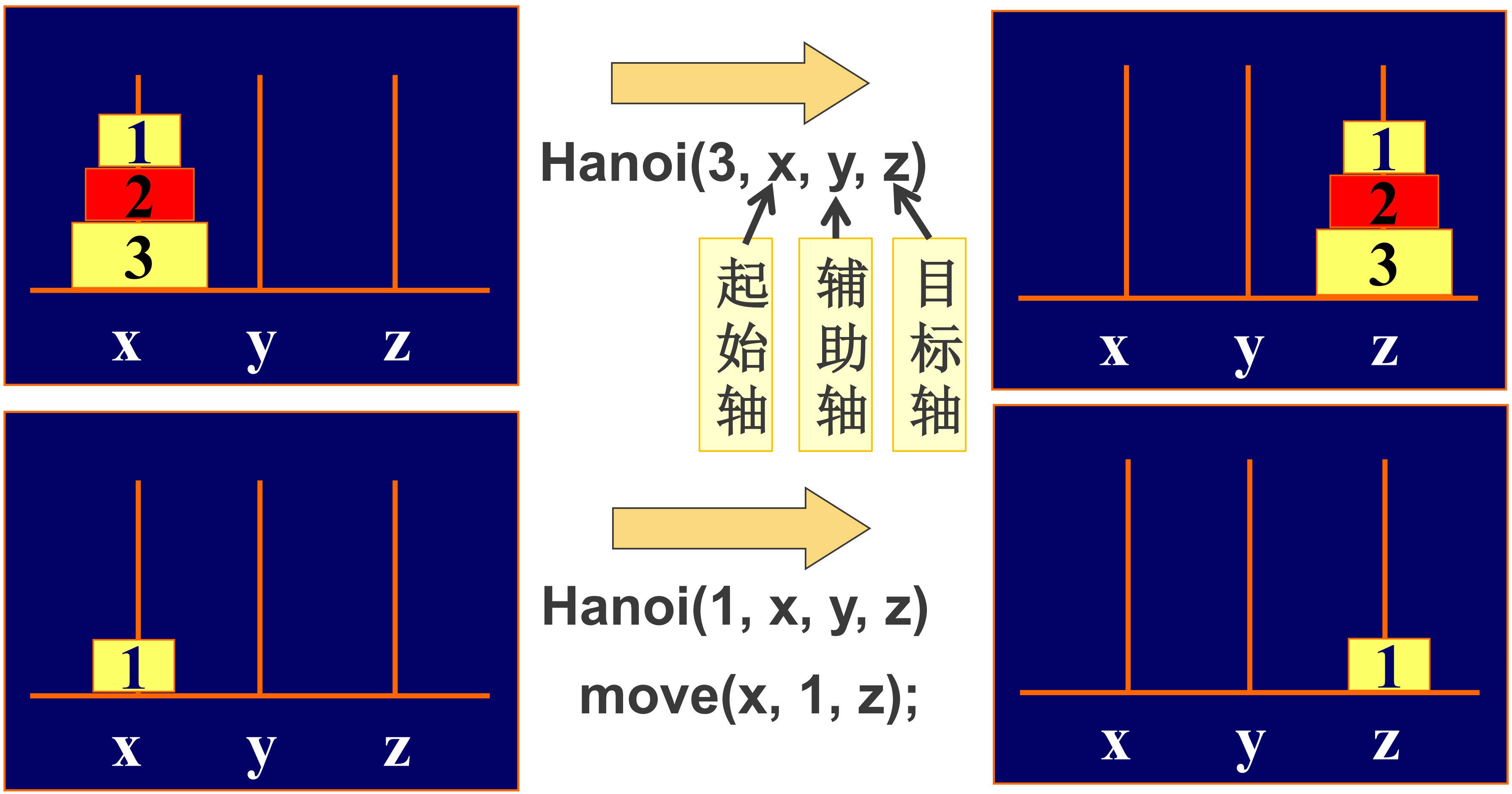
219÷73＝3

所以：GCD ＝ 73

```
int gcd(int a,int b)
{
    if (a%b==0)
        return(b); //终止条件
    else
        return(gcd(b, a%b);
}
```

例2、Hanoi Problem

◆ 递归函数执行的过程可视为同一函数进行嵌套调用

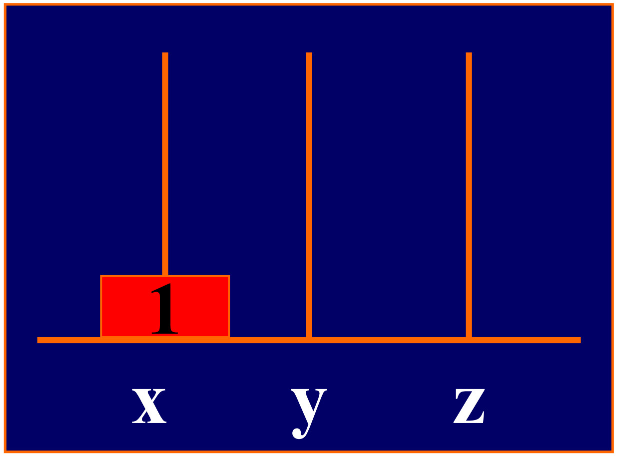


Hanoi Problem

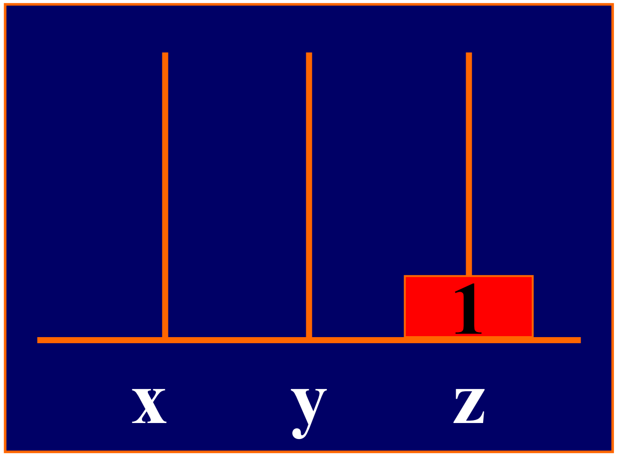
```
void hanoi (int n, char x, char y, char z) {

    if (n==1)
          move(x, 1, z);
     else {
        hanoi(n-1, x, z, y);
        move(x, n, z);
        hanoi(n-1, y, x, z);
     }
 }
```

←结束条件

Hanoi Problem

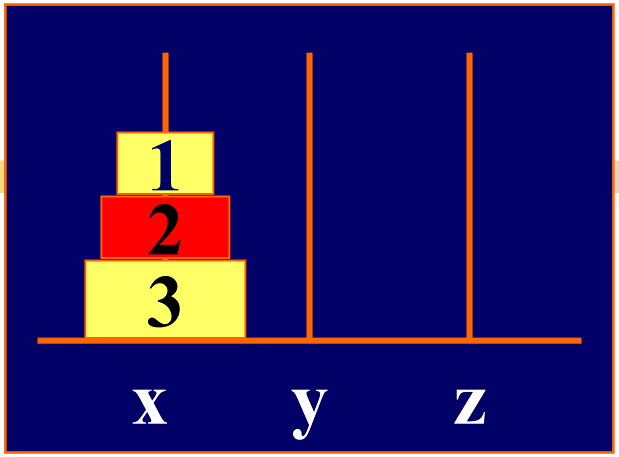


Hanoi（1, x, y, z)

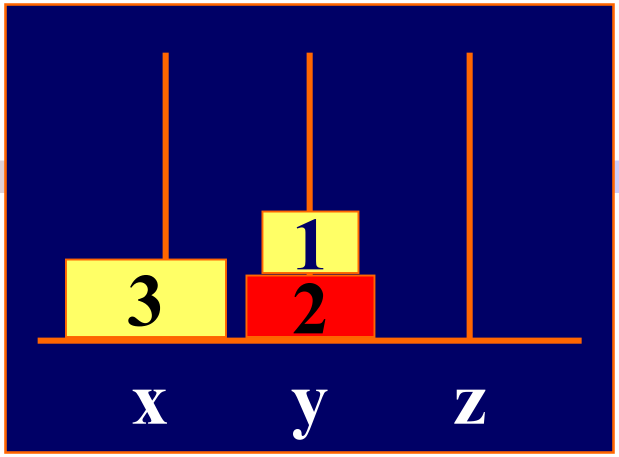


move(x, 1, z);

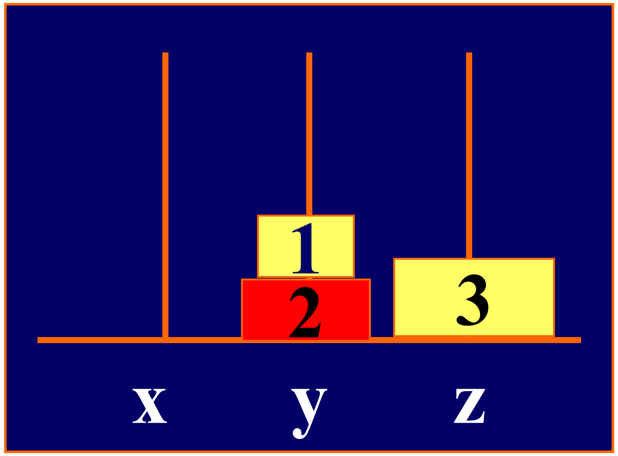
```
if (n==1)
        move(x, 1, z);
```



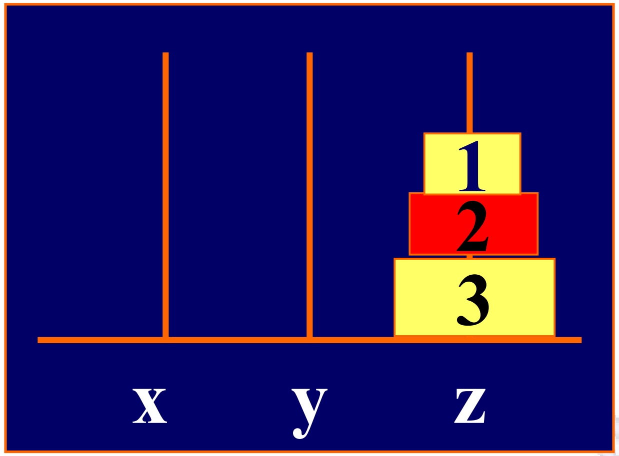
Hanoi (3, x, y, z)



Hanoi (2, x, z, y)

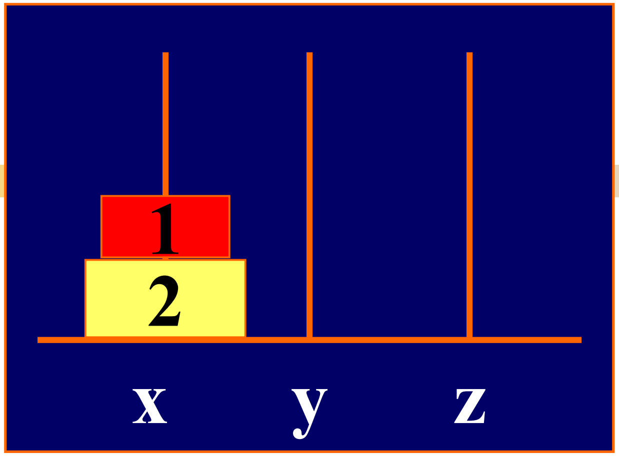


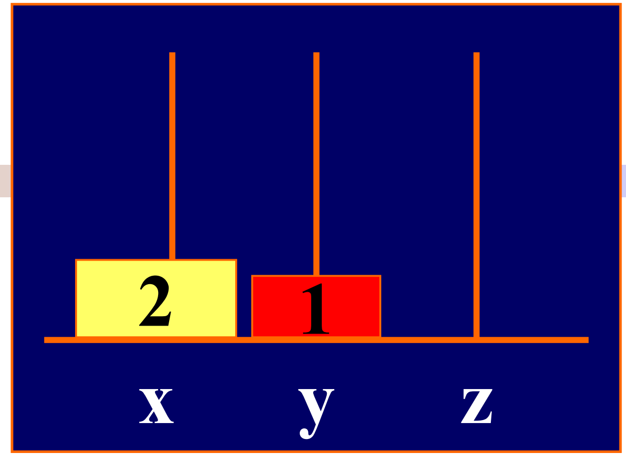
move(x, 3, z);



Hanoi (2, y, x, z)

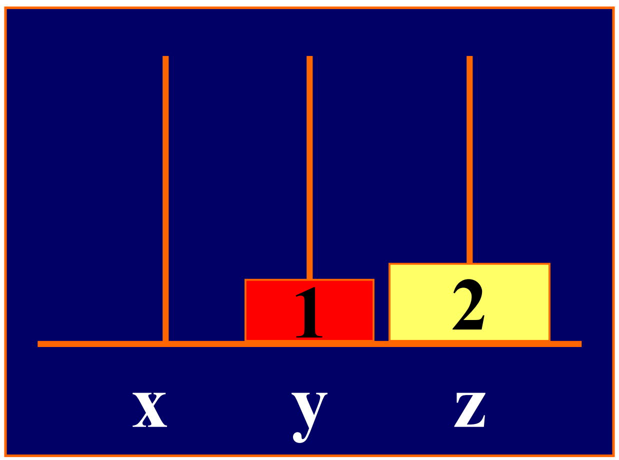
hanoi(n-1, x, z, y); move(x, n, z); hanoi(n-1, y, x, z);

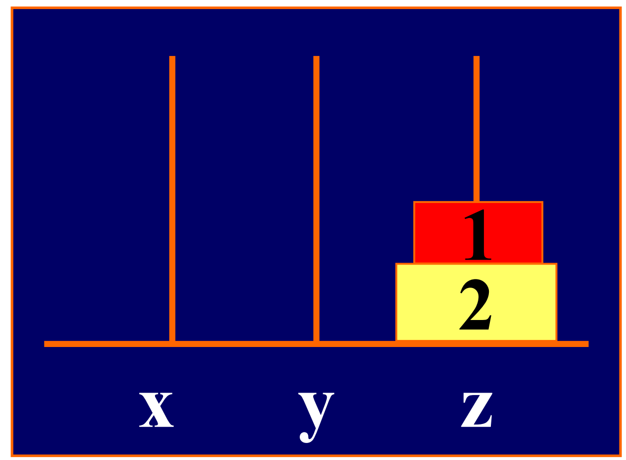




Hanoi (2, x, y, z)

Hanoi (1, x, z, y)





move(x, 2, z);

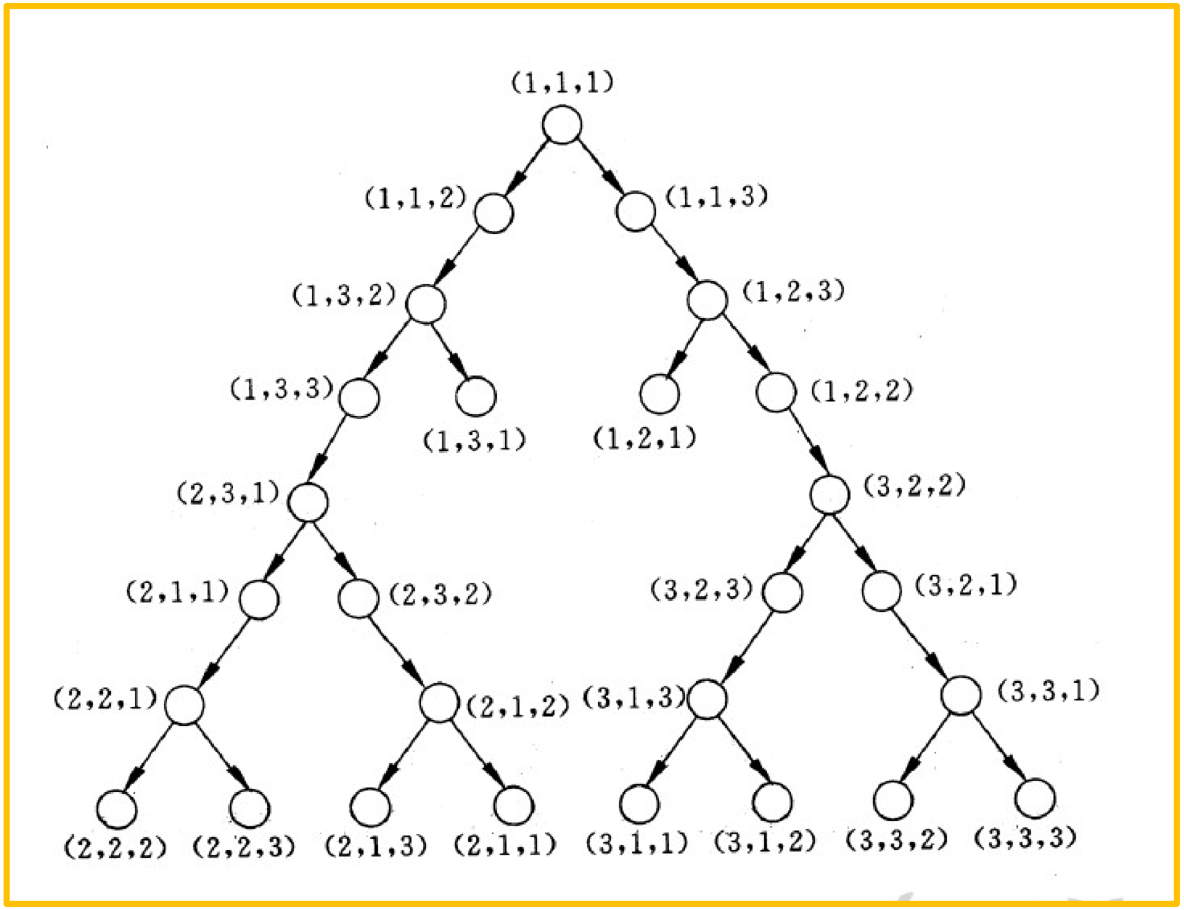
Hanoi (1, y, x, z)

hanoi(n-1, x, z, y); move(x, n, z); hanoi(n-1, y, x, z);

Hanoi Problem

```
void hanoi (int n, char x, char y, char z) {

    if (n==1)
        move(x, 1, z);   // 将编号为 1 的圆盘从x移到z
   else {
      hanoi(n-1, x, z, y);  // 将x上编号为 1 至n-1的
                            //圆盘移到y, z作辅助轴
      move(x, n, z);     // 将编号为n的圆盘从x移到z
      hanoi(n-1, y, x, z);  // 将y上编号为 1 至n-1的
                             //圆盘移到z, x作辅助轴
    }
}//hanoi
```

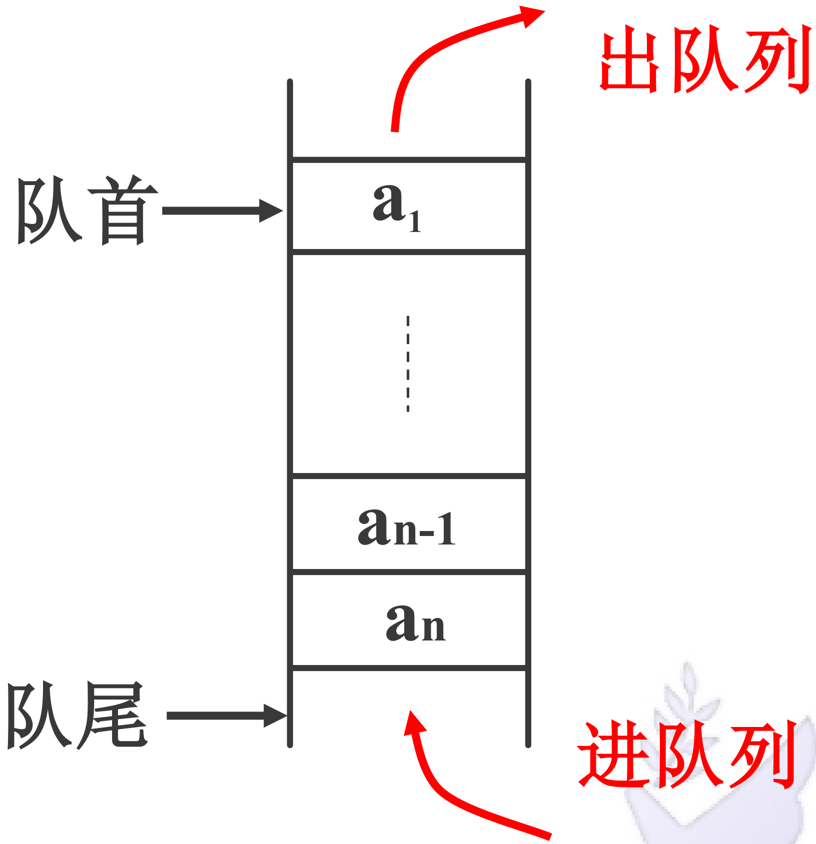
三阶梵塔问题的状态空间图



3.4 队列

- 队列：
- 一端插入，另一端删除
- 从尾进，从头出

- 队列的特点：
- 先进先出的线性表
- **FIFO—fist in fist out**



队列的类型定义

ADT Queue {

数据对象：

$D = \{a_i \mid a_i \in ElemSet, i=1,2,...,n, n \geq 0\}$

数据关系：

$R1 = \{ <a_{i-1}, a_i> \mid a_{i-1}, a_i \in D, i=2,...,n\}$

约定其中a_1 端为队列头，a_n 端为队列尾

基本操作：……

} ADT Queue

队列的基本操作

- InitQueue(&Q)
  - 操作结果：构造一个空队列Q
- DestroyQueue(&Q)
  - 初始条件：队列Q已存在。
  - 操作结果：队列Q被销毁，不再存在。
- ClearQueue(&Q)
  - 初始条件：队列Q已存在。
  - 操作结果：将Q清为空队列。

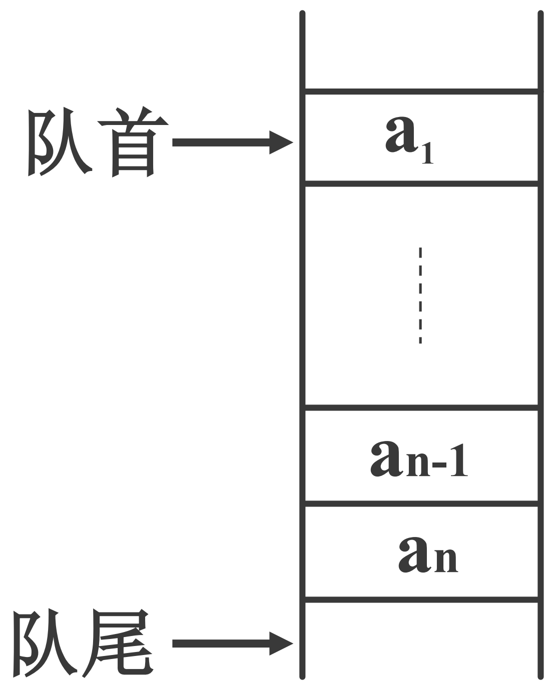
队列的基本操作

- **QueueEmpty(Q)**
  - 初始条件：队列Q已存在。
  - 操作结果：若Q为空队列，则返回TRUE，否则返回FALSE。

- **QueueLength(Q)**
  - 初始条件：队列Q已存在。
  - 操作结果：返回Q的元素个数，即队列的长度。

队列的基本操作

- **GetHead(Q, &e)**
  - 初始条件：Q为非空队列。
  - 操作结果：用e返回Q的队首元素。

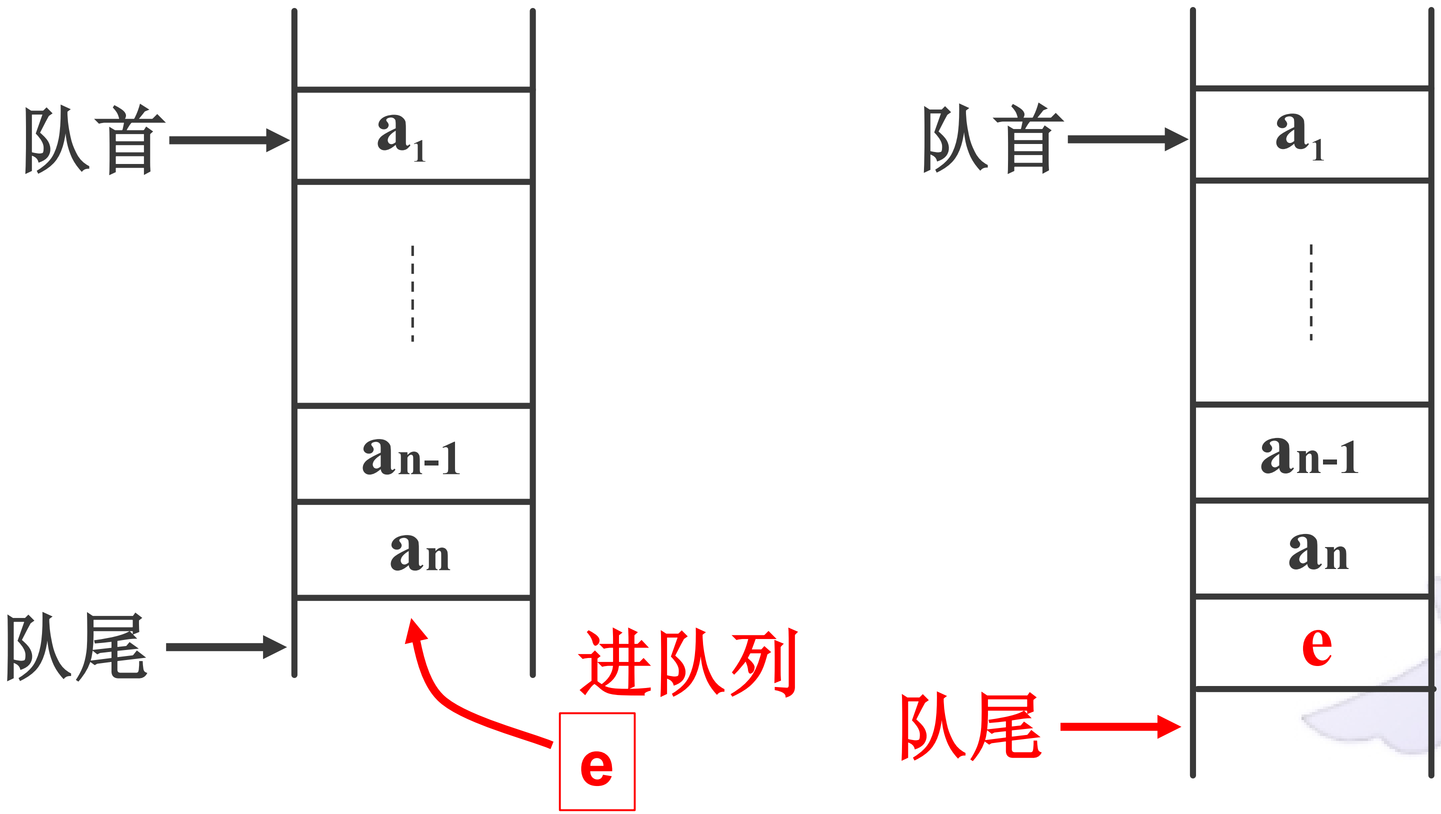


读取a_1

队首不变

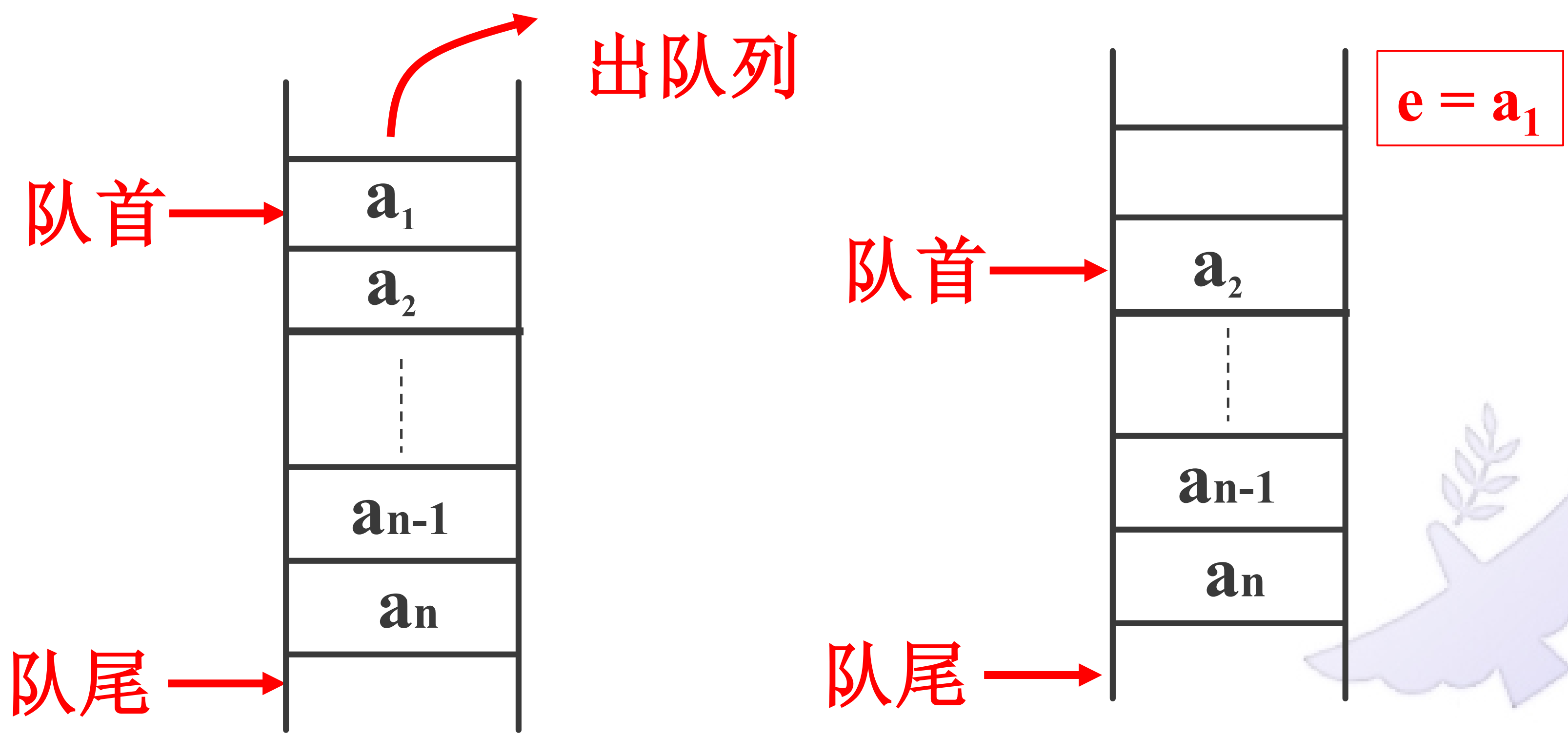
队列的基本操作

- EnQueue(&Q, e)
  - 初始条件：队列Q已存在。
  - 操作结果：插入元素e为Q的新的队尾元素。
-



队列的基本操作

- DeQueue(&Q, &e)
  - 初始条件：Q为非空队列。
  - 操作结果：删除Q的队头元素，并用e返回其值。



3.5 队列类型的实现

- 1）链队列——链式映象
- 2）循环队列——顺序映象

1）链队列——链式映象

结点类型

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

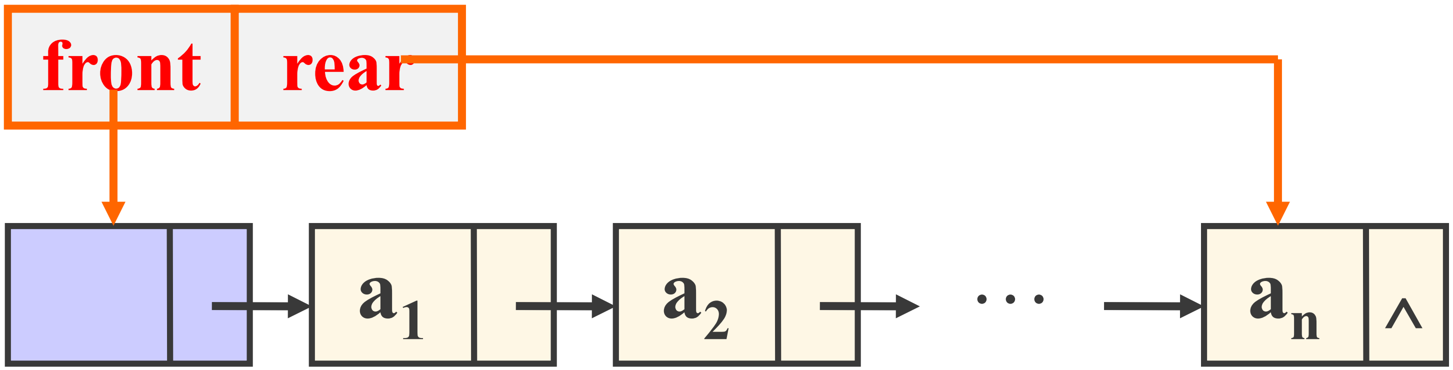
链队列类型

```
typedef struct {
        QueuePtr  front;  // 队头指针
        QueuePtr  rear;   // 队尾指针
} LinkQueue;
```

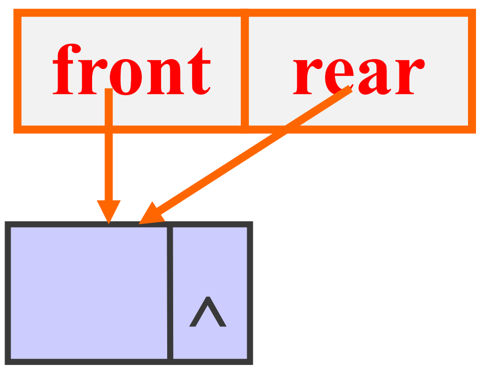
1）链队列——链式映象

实现方式一：带头结点的单链表，队首指针指向头结点

LinkQueue Q;



空队列



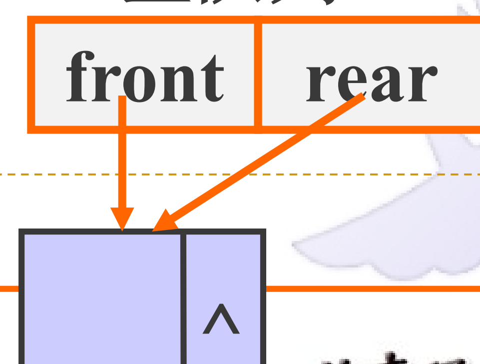
判空的条件：

Q.front->link== NULL

带头结点的链式队列

```
Status InitQueue (LinkQueue &Q) {
  // 构造一个有头结点的空队列Q
    Q.front = (QueuePtr)malloc(sizeof(QNode));
     if (!Q.front) exit (OVERFLOW);
                                    //存储分配失败
     Q.rear = Q.front;
     Q.front->link= NULL;
     return OK;
}// InitQueue
```

空队列



带头结点的链式队列

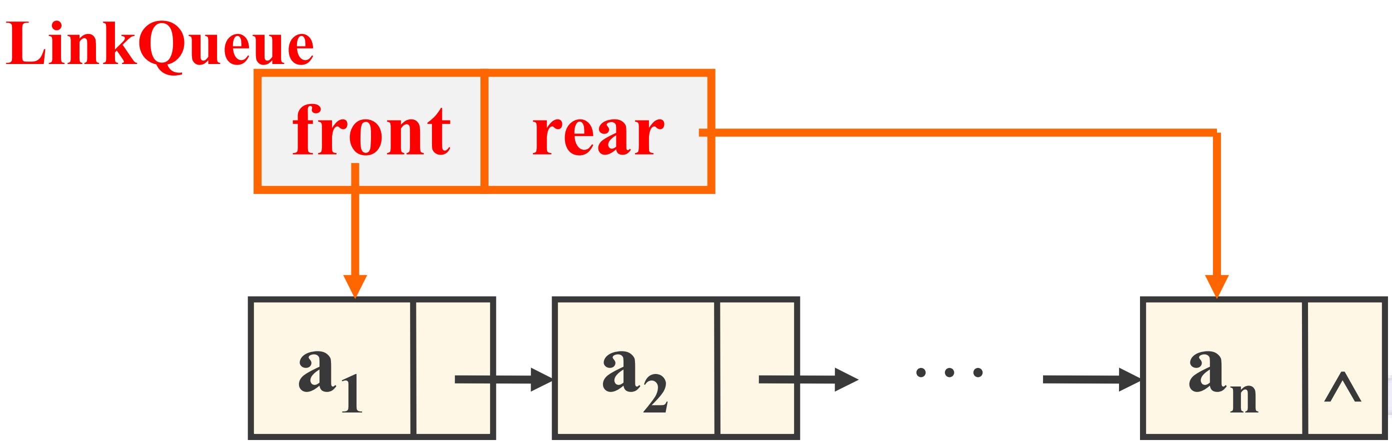
```
Status EnQueue (LinkQueue &Q, QElemType e) {
  // 插入元素e为Q的新的队尾元素
    p = (QueuePtr) malloc (sizeof (QNode));
    if (!p)  exit (OVERFLOW);   //存储分配失败
    p->data = e;    p->link = NULL;
    Q.rear->link = p;     Q.rear = p;
    return OK;
}// EnQueue
```

带头结点的链式队列

```
Status DeQueue (LinkQueue &Q, QElemType &e) {
 // 若队列不空，则删除Q的队头元素，
 //用 e 返回其值，并返回OK；否则返回ERROR
    if (Q.front == Q.rear)    return ERROR;
    p = Q.front->link;   e = p->data;
    Q.front->link = p->link;
     if (Q.rear == p)  Q.rear = Q.front;   //修改rear
     free (p);
    return OK;
}// DeQueue
```

链式队列

- 实现方式二：不带头结点的单链表
  - 队首指针指向第一个数据结点，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 **front == NULL**。
- 链式队列特别适合多个队列同时操作的情形。在并行处理、排序等方面有用。



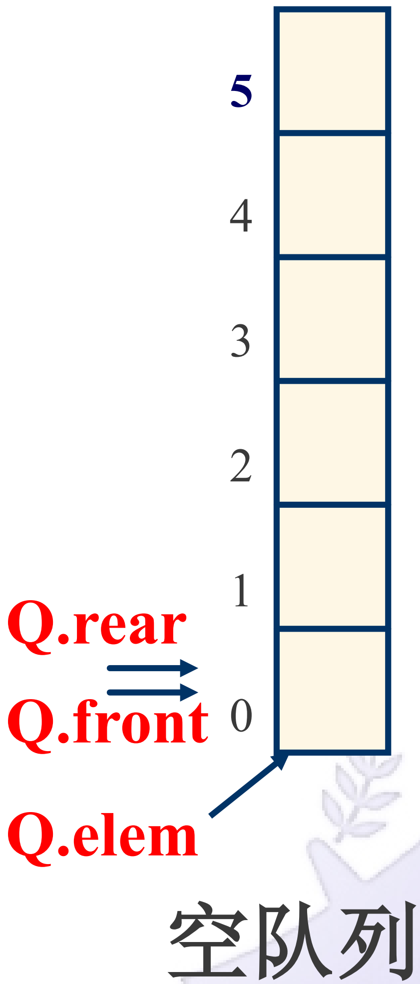
2）循环队列——顺序映象

```
#define MAXQSIZE  100  //最大队列长度
  typedef struct {
    QElemType  elem[MAXQSIZE];  //静态空间
    int  front;     // 头指针，若队列不空，
                    //  指向队列头元素
    int  rear;      // 尾指针，若队列不空，指向
                    // 队列尾元素 的下一个位置
  } SeqQueue, CircQueue;
```

2）循环队列——顺序映象

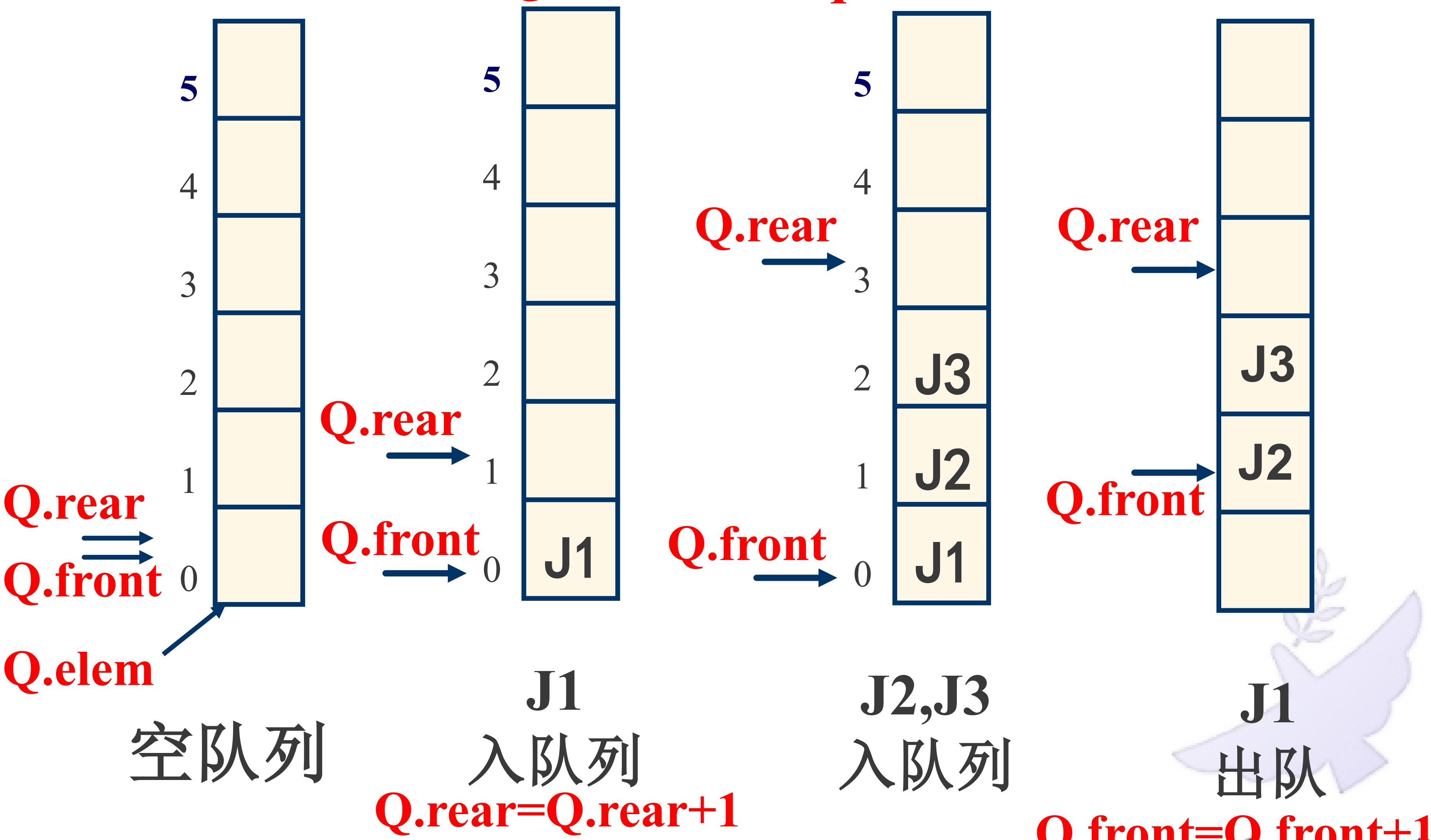
InitQueue

```
Status InitQueue (SeqQueue &Q) {
  // 构造一个空队列Q
    Q.front = 0;
    Q.rear = 0;
    return OK;
}// InitQueue
```



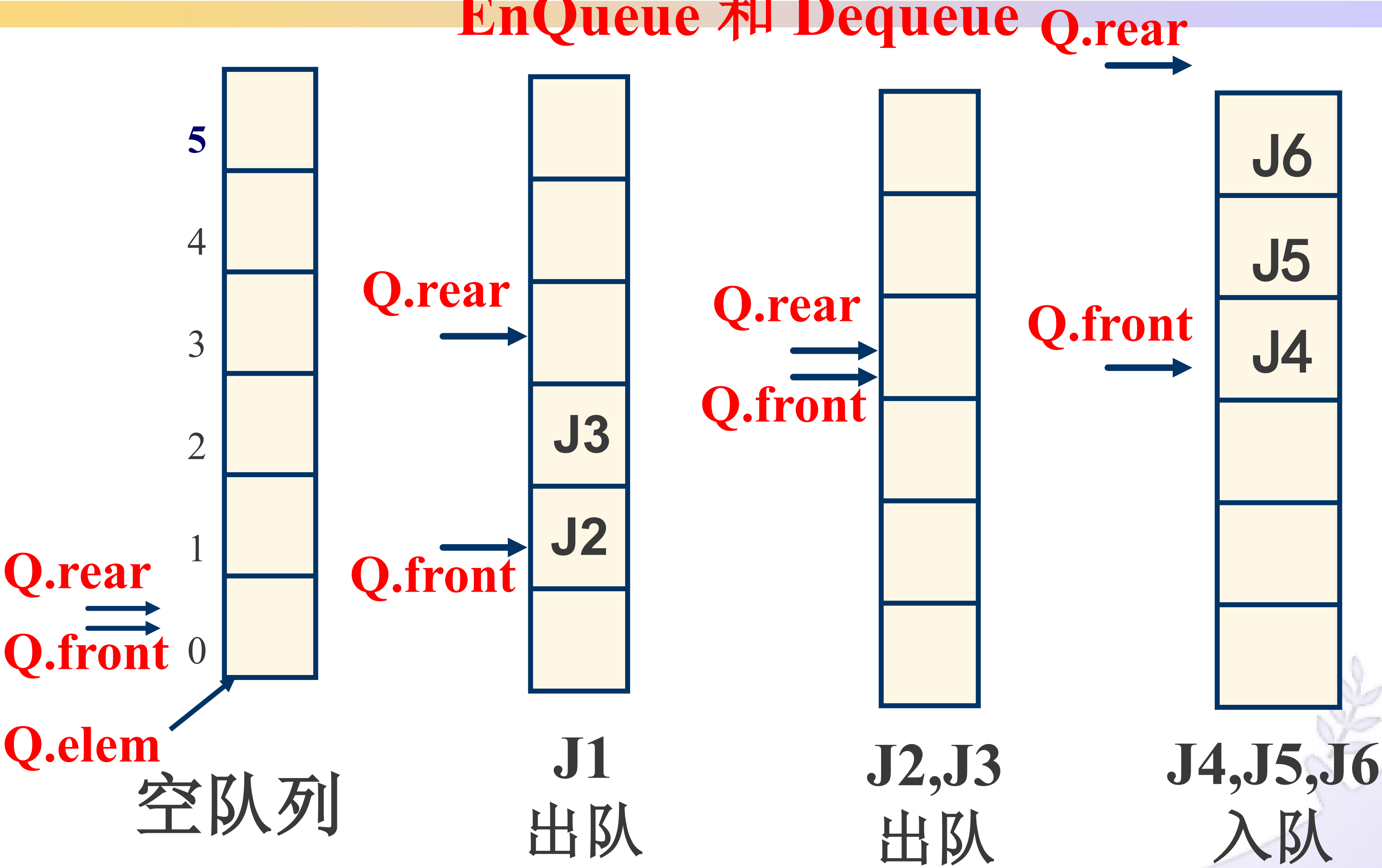
2）循环队列——顺序映象

EnQueue 和 Dequeue



2）循环队列——顺序映象

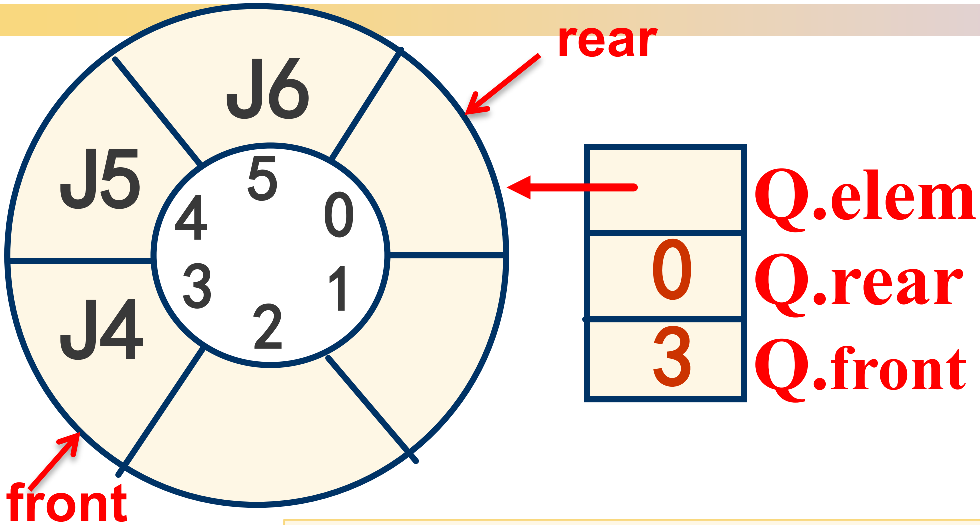
EnQueue 和 Dequeue

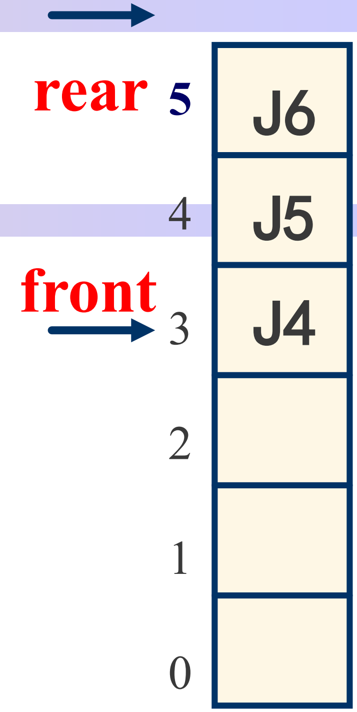


Q.rear=Q.rear+1　Q.front=Q.front+1

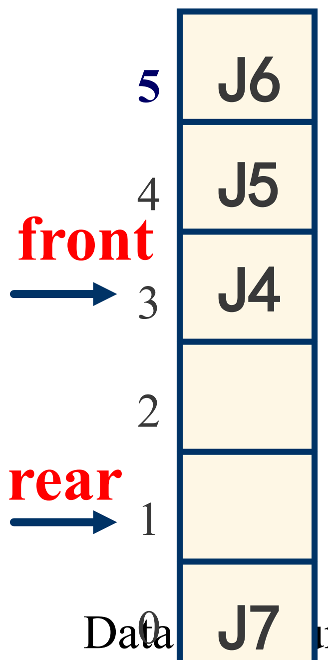
循环使用队列空间

J7入队操作前

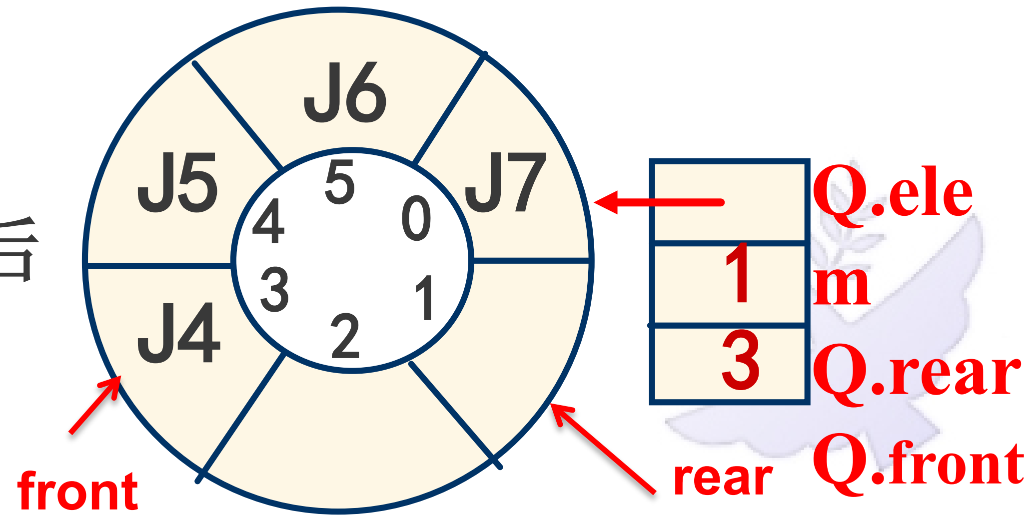




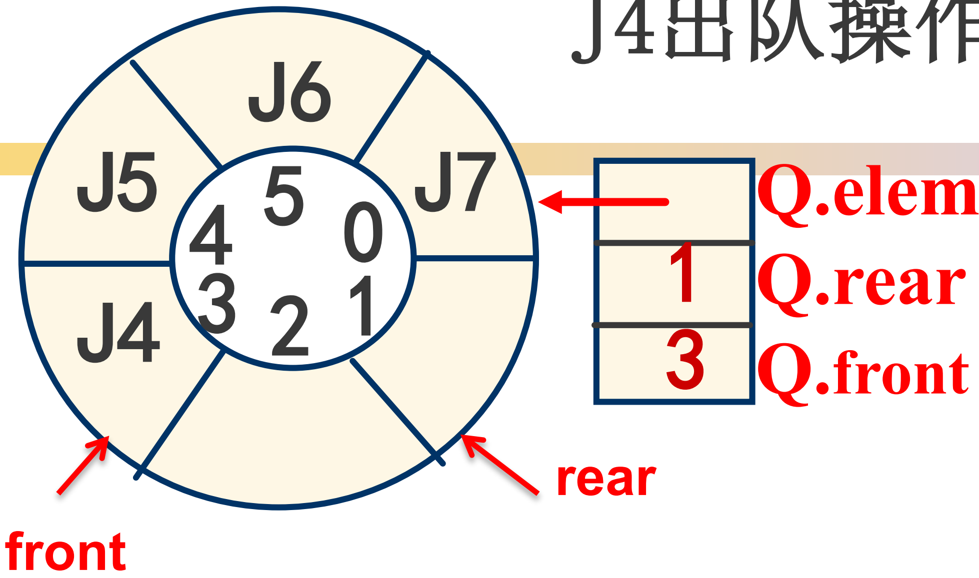
Q.rear=(Q.rear+1)%MAXQSIZE

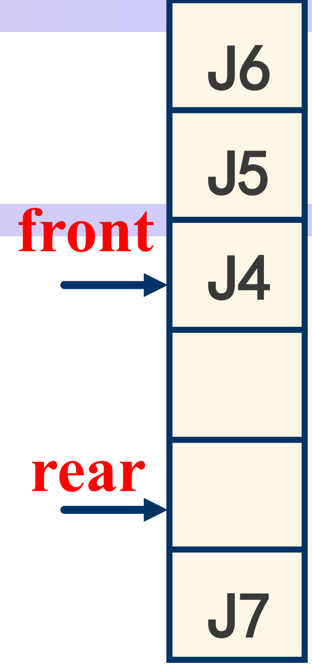


J7入队后

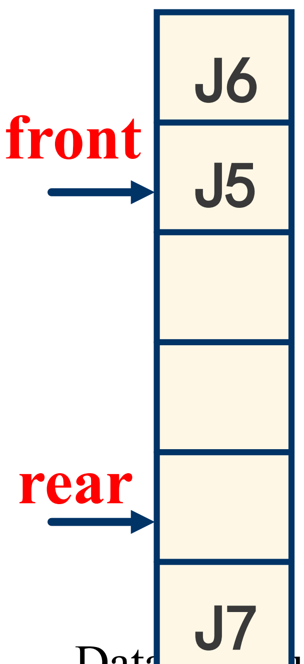


J4出队操作前

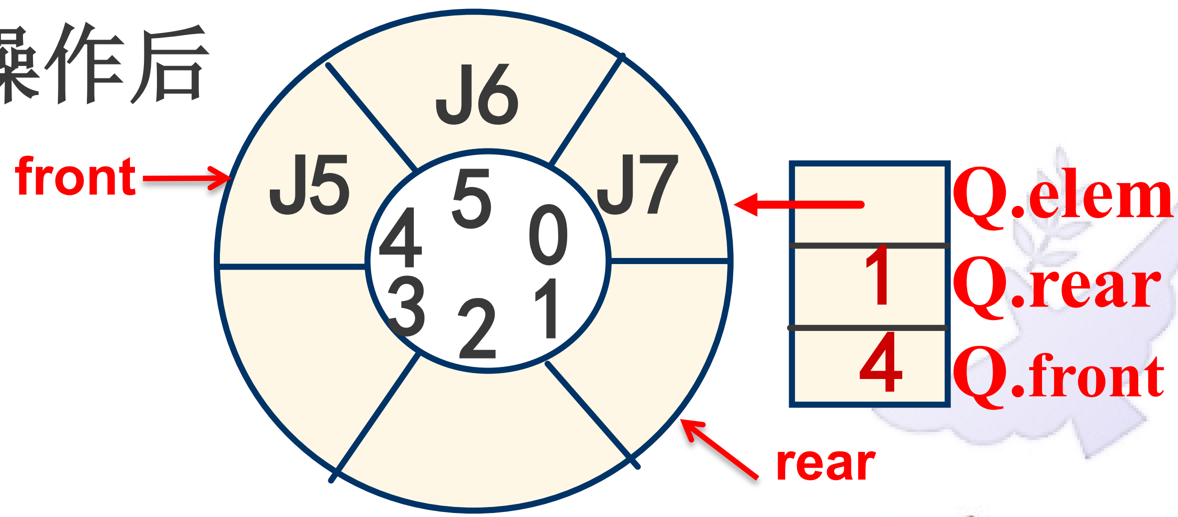




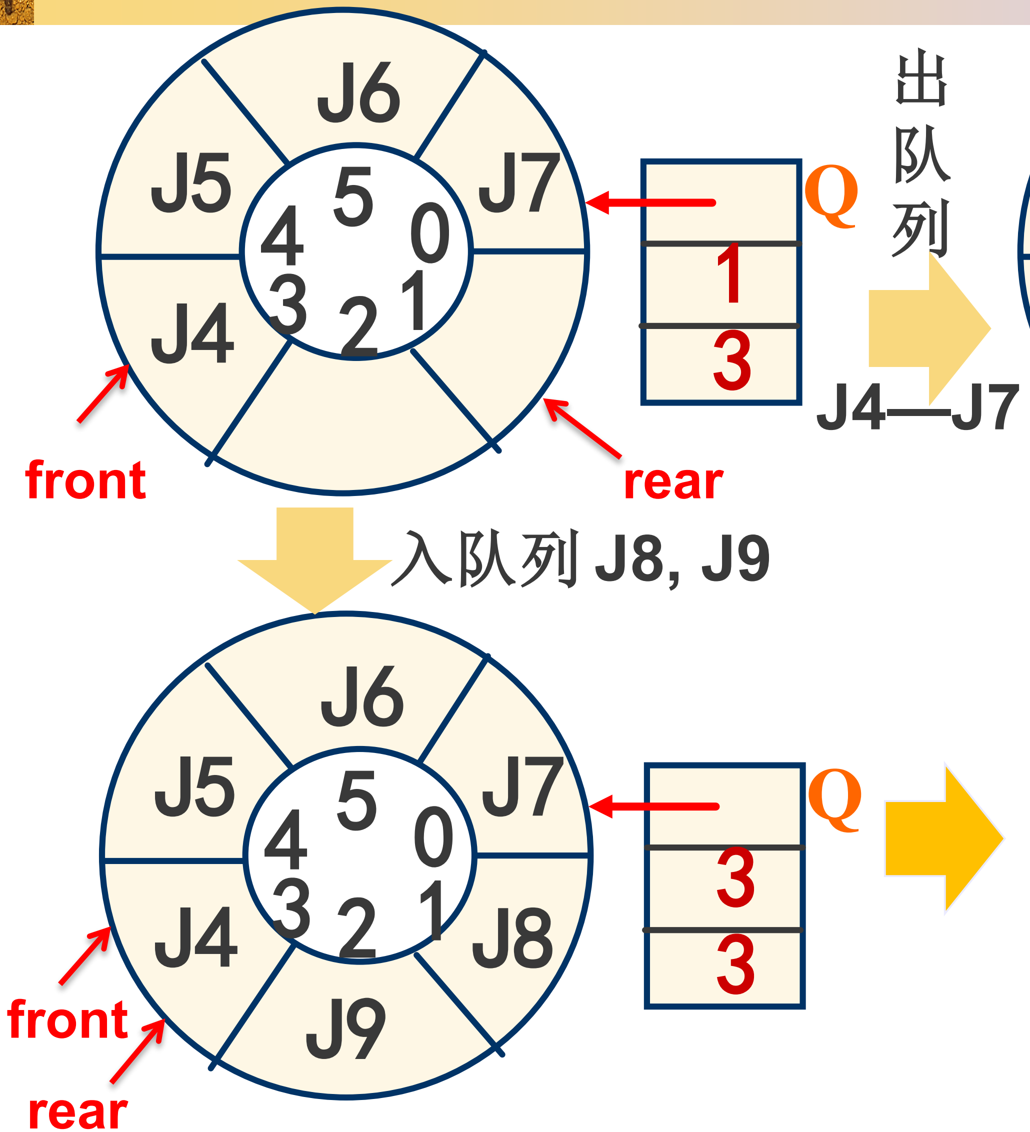
$$Q.front=(Q.front+1)\%\ MAXQSIZE$$



J4出队操作后



循环队列判空、判满的方法及条件



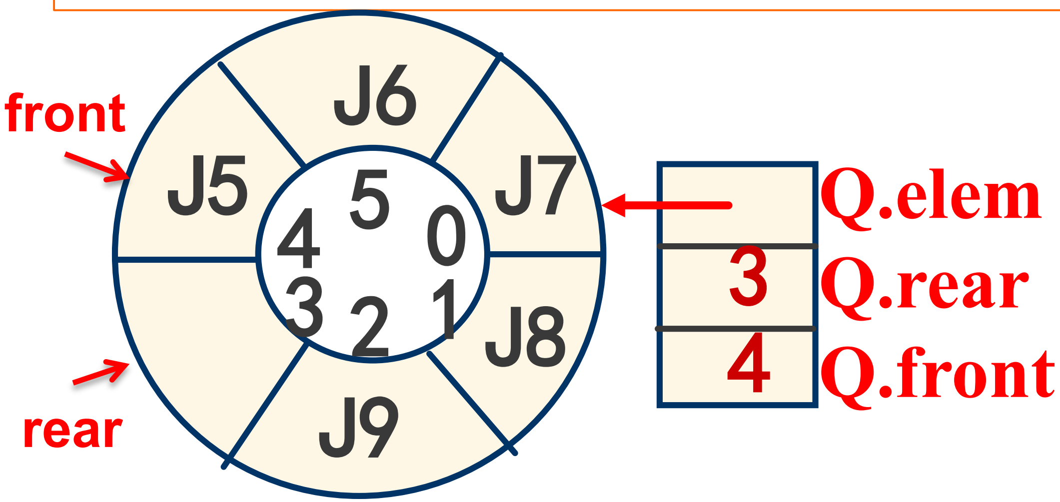
Q.rear = =Q.front

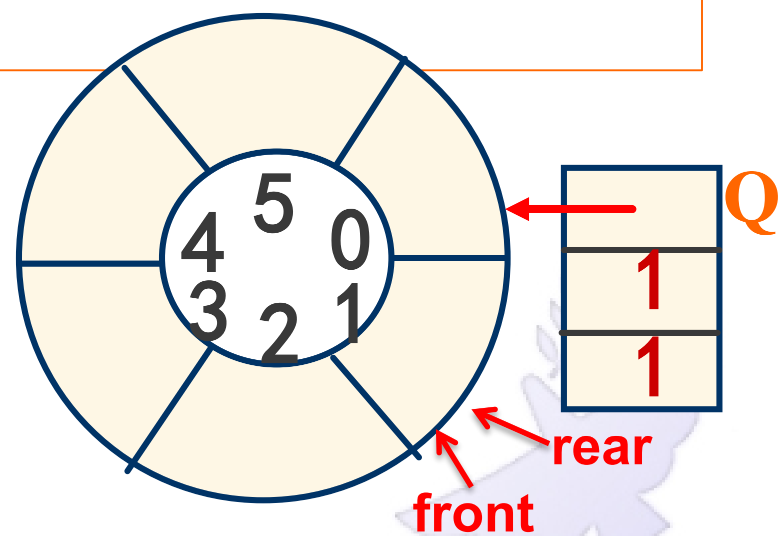
不能作为判断依据

- 解决的方法：
  - 方法1：保存队长
  - 方法2：少用一个存储单元
  - 方法3：设标志tag，元素入队时设为1，队列满时恢复0.

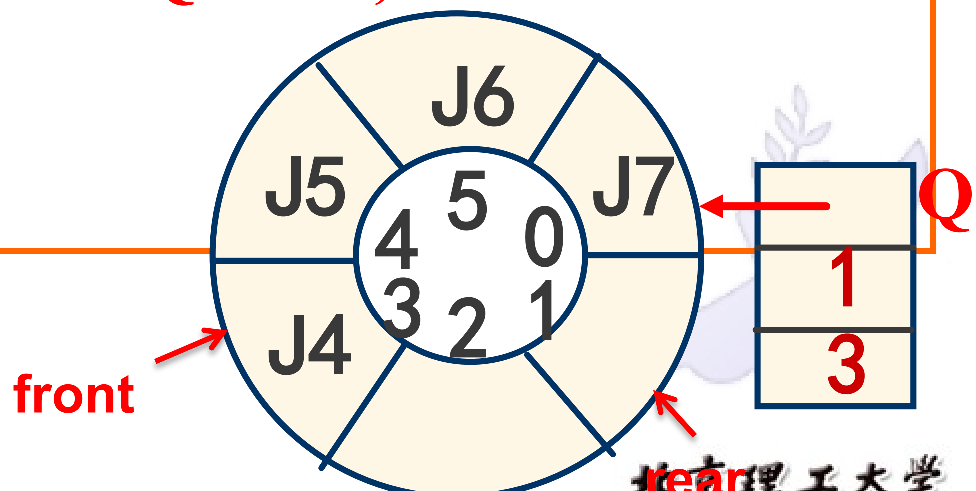
判满条件(Q.rear+1)%MAXQSIZE = =Q.front

判空条件 Q.rear = =Q.front

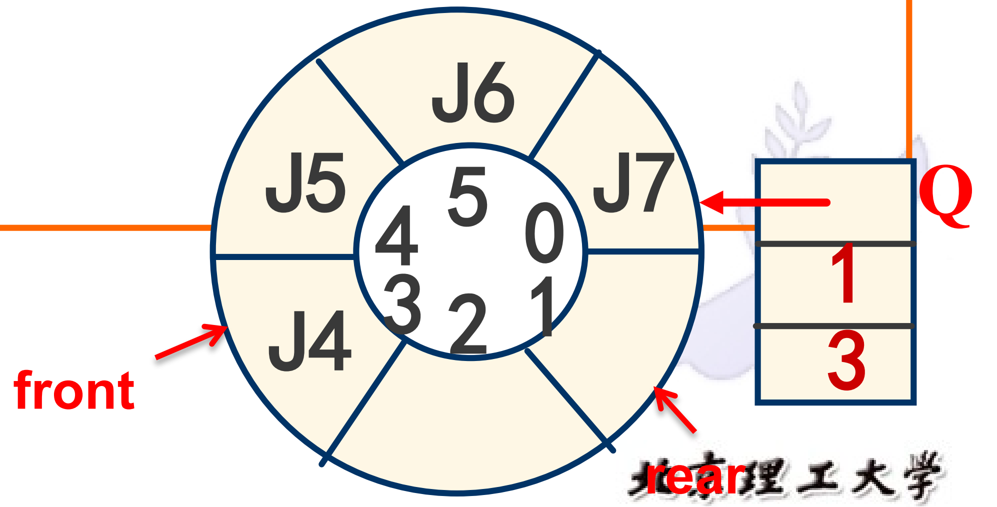




```
Status EnQueue (SeqQueue &Q, ElemType e) {
        // 插入元素e为Q的新的队尾元素
  if ((Q.rear+1) % MAXQSIZE == Q.front)
        return ERROR; //队列满
  Q.elem[Q.rear] = e;
  Q.rear = (Q.rear+1) % MAXQSIZE;
  return OK;
}// EnQueue
```



```
Status DeQueue (SeqQueue &Q, ElemType &e) {
  // 若队列不空，则删除Q的队头元素，
  // 用e返回其值，并返回OK；否则返回ERROR
   if (Q.front == Q.rear)  return ERROR;
   e = Q.elem[Q.front];
   Q.front = (Q.front+1) % MAXQSIZE;
   return OK;
}// DeQueue
```



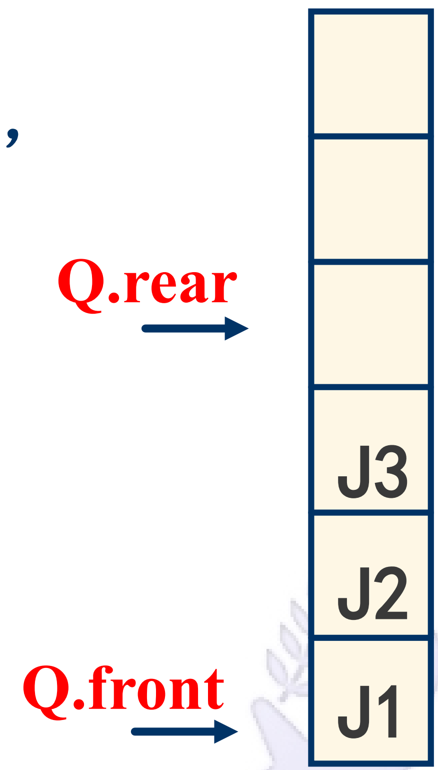
注意

- 在循环队列中为何不能用动态数组？
- 循环队列必须要设置最大长度！

注：队列的进队和出队的原则

- 有两种进/出队列的方案：

1. 先加元素再动指针(本课程中为该方案)
   - <u>队尾指针指示实际队尾的后一位置。</u>
   - 进队时先将新元素按 **rear** 指示位置加入，再让队尾指针进一 **rear = rear + 1**。
   - <u>队头指针指示实际队头的位置。</u>
   - 出队时先将下标为 **front** 的元素取出，再将队头指针进一 **front = front + 1**。
   - 清华、北大教材均为此方案。



注：队列的进队和出队的原则

2. 先动指针再加元素

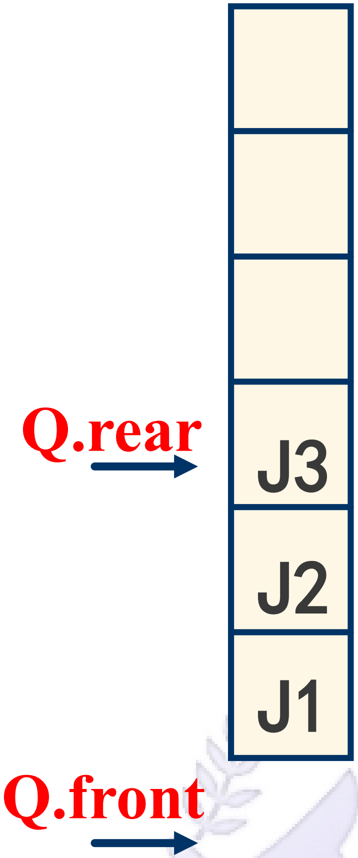
¶ <u>队尾指针指示实际队尾的位置。</u>

¶ 进队时先让队尾指针进一 **rear = rear + 1**，再将新元素按 **rear** 指示位置加入。

¶ <u>队头指针指示实际队头的前一位置。</u>

¶ 出队时先将队头指针进一 **front = front + 1**，再将下标为 **front** 的元素取出。

¶ 微软Visual C++ STL 按此处理。



3.5队列的应用

- 日常生活中的排队
  - 搭乘公共汽车；
  - 顾客到商店购买物品；
  - 病员到医院看病；
  - 旅客到售票处购买车票；
  - 学生去食堂就餐
  - “无形”排队：多个顾客打电话叫出租车，如果出租汽车站无足够车辆、则部分顾客只得在各自的要车处等待，他们分散在不同地方，却形成了一个无形队列在等待派车

- 物体队列：
  - 通讯卫星与地面若干待传递的信息；
  - 生产线上的原料、半成品等待加工；
  - 因故障停止运转的机器等待工人修理；
  - 码头的船只等待装卸货物；
  - 要降落的飞机因跑道不空而在空中盘旋等等

3.5 队列的应用

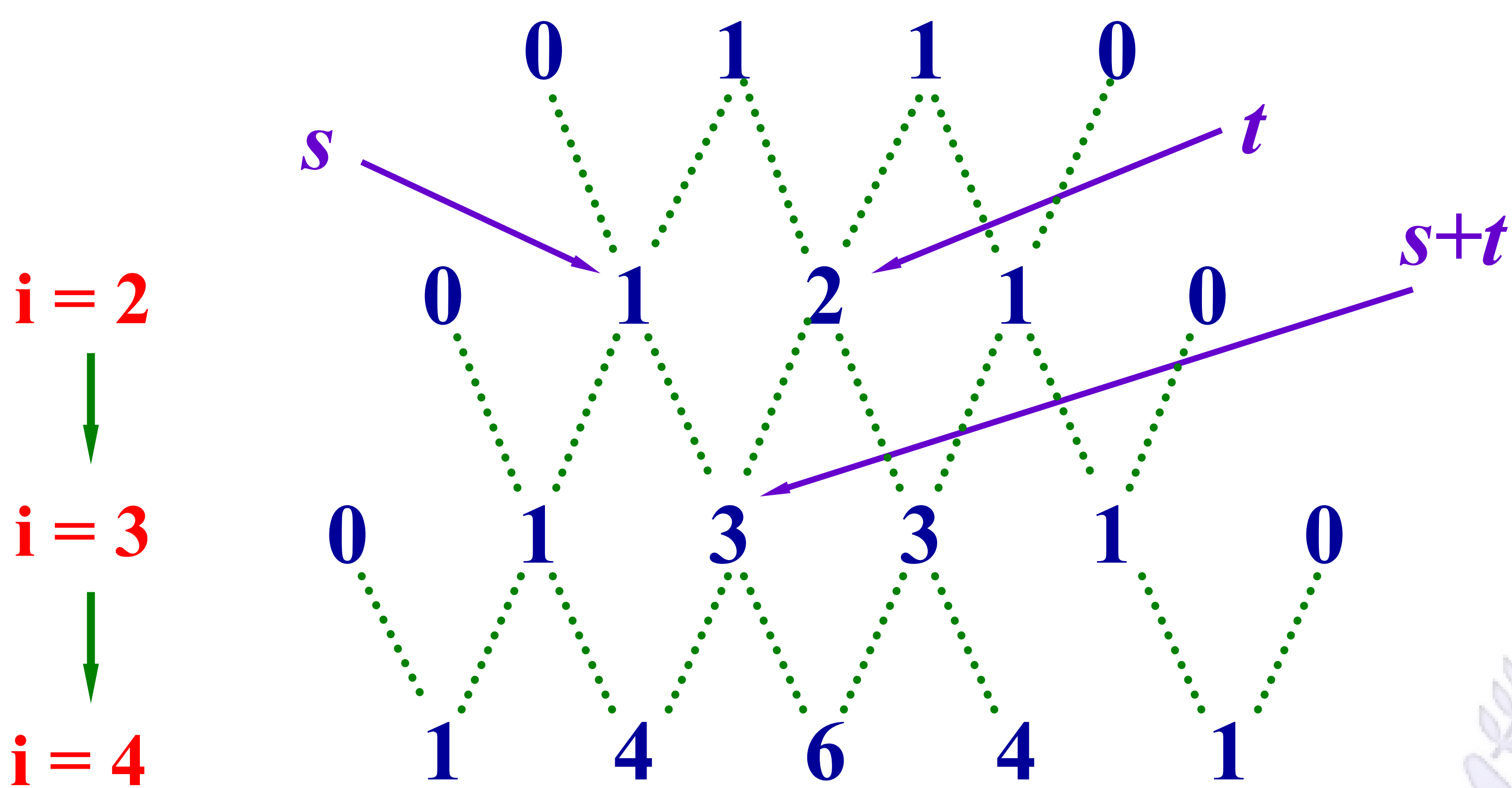
- 排队论(Queuing Theory)，又称随机服务系统理论(Random Service System Theory),是一门研究拥挤现象(排队、等待)的科学。
- 具体地说，它是在研究各种排队系统概率规律性的基础上，解决相应排队系统的最优设计和最优控制问题。例如：
  1. 停车场：如何设计停车场的车位和停车规则，提高停车场的使用效率。
  2. 信号交叉口：根据不同的车流量，设计区域内所有信号灯的变化模式。减少由于设计不当造成车流堆积现象。
  3. 公共交通系统运营优化：减少乘客等待时间，并尽量降低运营成本。

队列的应用：打印杨辉三角形

- 算法逐行打印二项展开式 $(a+b)^i$ 的系数：杨辉三角形 **(Pascal's triangle)**

1 1	$i = 1$
1 2 1	2
1 3 3 1	3
1 4 6 4 1	4
1 5 10 10 5 1	5
1 6 15 20 15 6 1	6

分析第 i 行元素与第 i+1行元素的关系



从前一行的数据可以计算下一行的数据

1	1	0	1	2	1	0	1	3	3	1	0	1	4	6

```
EnQueue(Q, 1);  EnQueue(Q, 1); //第 1 行系数进队
int s = 0, t;
for (int i = 1; i <= n; i++ ) {             //逐行输出
    printf("\n")      EnQueue (Q, 0); //各行间插入一个0
    for ( int j = 1; j <= i+2; j++ ) {    //第i行的i+2个系数
                DeQueue(Q, t);   //退出一个系数放入t
        EnQueue(Q, s+t); //计算下一行系数，并进队列
        s = t;
        if ( j != i+2 ) printf("%d ", s);        //输出一个系数
    }//for i
}//for j
```

本章学习要点

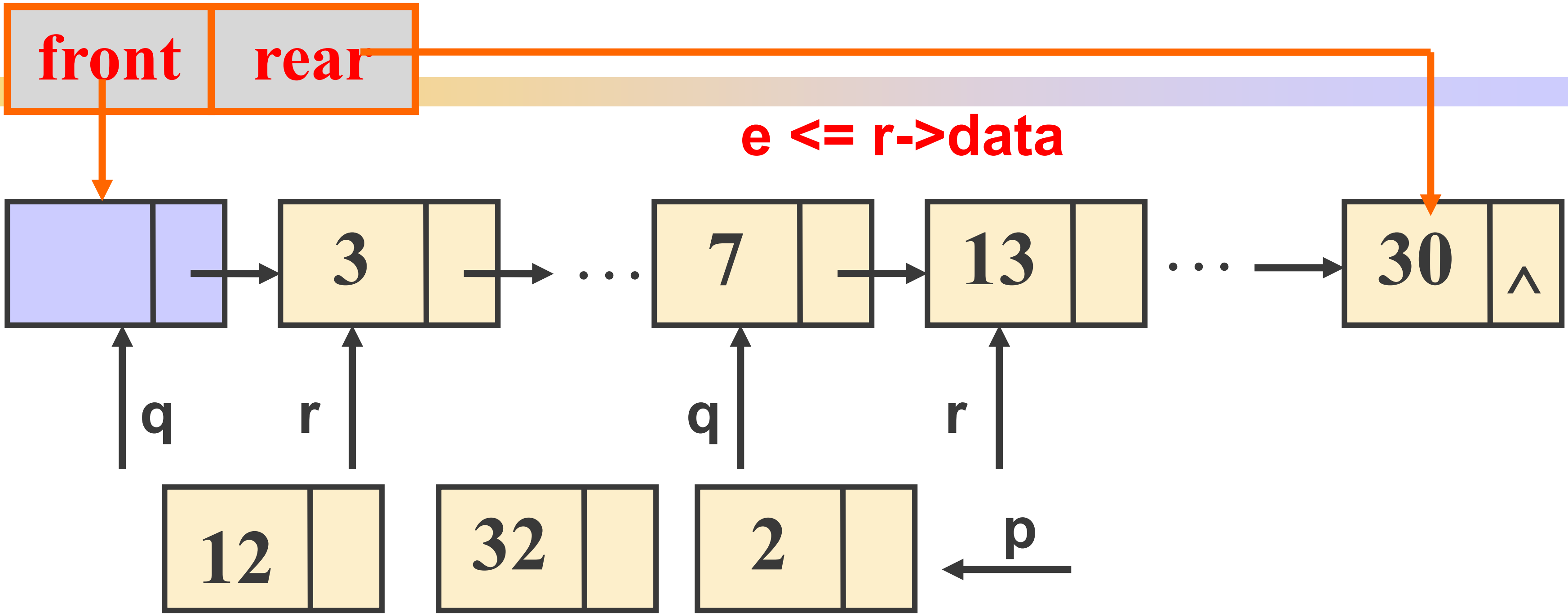
1. 掌握栈和队列类型的特点，并能在相应的应用问题中正确选用它们。
2. 熟练掌握栈类型的两种实现方法，特别应注意栈满和栈空的条件以及它们的描述方法
3. 熟练掌握循环队列和链队列的基本操作实现算法，特别注意队满和队空的描述方法
4. 理解递归算法执行过程中栈的状态变化过程

思考题：优先级队列

- 问题：如何实现有序队列的插入操作？

```
//普通队列的入队操作
Status EnQueue (LinkQueue &Q, QElemType e) {
    // 插入元素e为Q的新的队尾元素
    p = (QueuePtr) malloc (sizeof (QNode));
    if (!p)  exit (OVERFLOW);   //存储分配失败
    p->data = e;   p->link = NULL;
    Q.rear->link = p;    Q.rear = p; //插在队尾
    return OK;
}
```

•找到第一个大于等于e的结点，插在它之前




```
while(r != NULL) {
        if(e <= r->data ) break;
        q=r;  r = r->link;}
 p->link = r;    q->link = p;
if(r == NULL)  Q.rear = p;
```

END OF CHPATER III