

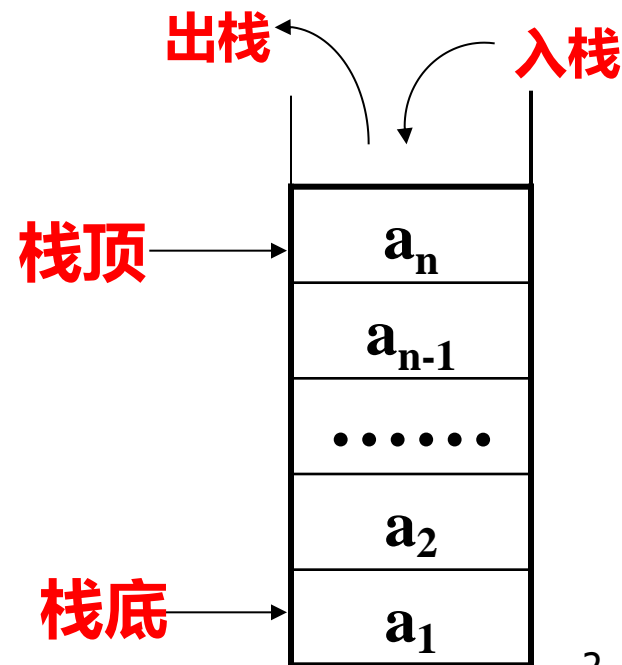


# 第三章 栈和队列

- 简介：
  - 线性结构，操作受限的线性表-限定性数据结构
  - 广泛用于程序设计、系统软件设计以及递归问题处理
  - 栈和队列定义，表示方法及在各种存储结构上基本操作的实现
  - 栈和队列的一些应用实例。
- 重点：
  - 栈和队列的定义、特性，能正确应用解决实际问题
  - 熟练掌握栈的顺序表示、链式表示及相应操作的实现
  - 熟练掌握队列顺序表示、链表表示及相应操作的实现
  - 循环队列满/空的判断条件
- 难点：
  - 能在具体问题中灵活运用栈和队列
  - 循环队列操作和应用
- 目的：加深对线性表(逻辑、存储结构) 的理解

## 3.1 栈的定义和示意图

- **栈(Stack)**: 限制在线性表一端插入和删除的线性表,
  - **栈顶(top)**: 插入、删除的一端, 对应表尾;
  - **栈底(bottom)**: 另一端, 对应表头。
  - **空栈**: 表中没有元素的栈
- 设栈 $S=(a_1, a_2, a_3, \dots a_n)$ ,
  - **栈底元素**:  $a_1$
  - **栈顶元素**:  $a_n$
  - **修改原则**: 后进先出 (**LIFO**)





# 栈的抽象数据类型的定义

ADT Stack {

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

数据关系:  $R1 = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i=2,\dots,n \}$

基本操作: **InitStack(&S), DestroyStack(&S)**

**ClearStack(&S), Push(&S, e), Pop(&S, &e)**

**StackLength(S), GetTop(S, &e)**

**StackTraverse(S, visit()), StackEmpty(S)**

LocateElem(S, e, compare()),

PriorElem(S, cur\_e, &pre\_e),

NextElem(S, cur\_e, &next\_e)

} ADT Stack

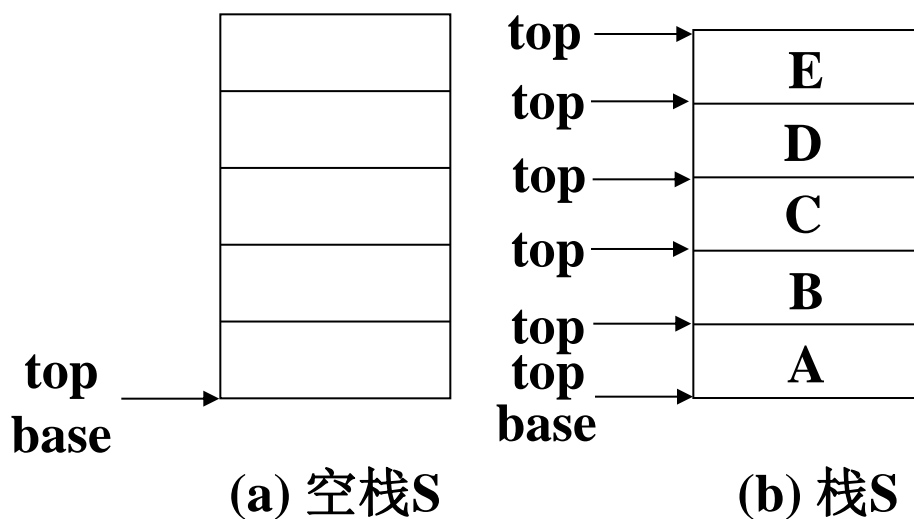


# 顺序栈的表示和实现

- **特点：**用一组地址**连续**的存储单元**依次**存放**自栈底到栈顶**的数据元素，附设指针top指示栈顶元素在顺序栈中的位置。
- **顺序栈的C语言描述：**

```
#define STACK_INIT_SIZE 100//为栈分配一个基本容量
#define STACKINCREMENT 10//存储空间分配增量
typedef struct{
    SElemType *base;//栈底指针，始终指向栈底；
    SElemType *top; //栈顶指针，始终指向栈顶元素的下一个位置
    int stacksize;   //栈的初始容量
}SqStack;
```

# 顺序栈栈顶、栈底指针和栈中元素的关系



**判空条件:  $S.top == S.base$**

**插入元素:  $S.base[S.top++] = 'A';$**

**判满条件:  $S.top - S.base == S.stacksize$**

**删除元素:  $e = S.base[--S.top];$**



## 顺序栈的基本操作(1)

```
Status InitStack(SqStack &S) //构造一个空栈S
{
    S.base=(SElemType*)malloc((STACK_INIT_SIZE)*sizeof(
        SElemType));
    if(!S.base) exit(OVERFLOW);
    S.top=S.base;//空栈
    S.stacksize=STACK_INIT_SIZE;//栈的初始容量分配
    return OK;
} //InitStack

Status DestroyStack(SqStack &S)
{
    if(S.base)
        free(S.base);
    return OK;
}
```



## 顺序栈的基本操作(2)

---

```
Status ClearStack(SqStack &S)
{
    while(S.top!=S.base) //在栈不空情况下
        Pop(S,e);
    return OK;
}

Status StackEmpty(SqStack S)
{
    if(S.top==S.base)
        return TRUE;
    else
        return FALSE;
}
```



## 顺序栈的基本操作(3)

---

```
SElemType StackLength(SqStack S)
```

```
{
```

```
    return S.top-S.base;
```

```
}
```

```
Status GetTop(SqStack S, SElemType &e)
```

```
{//用e返回栈顶元素
```

```
    if(S.top==S.base) return ERROR;//栈空
```

```
    e=*(S.top-1);    //取栈顶元素
```

```
    return OK;
```

```
}//GetTop
```





## 顺序栈的基本操作(4)

```
Status Push(SqStack &S,SElemType e)
{ //入栈，插入元素e为新的栈顶元素
    if(S.top-S.base>=S.stacksize)
    {   S.base=(SElemType*)realloc(S.base,(S.stacksize+STACKINCREMENT)*sizeof(SElemType));
        if(!S.base)        exit(OVERFLOW);
        S.top=S.base+S.stacksize; //栈顶指针复位
        S.stacksize+=STACKINCREMENT; //栈容量扩增
    }
    *S.top++=e;
    return OK;
} //Push
```



## 顺序栈的基本操作(5)

---

```
Status Pop(SqStack &S,SElemType &e)
```

```
{//出栈，删除栈顶元素用e返回
```

```
    if(S.top==S.base)//栈空
```

```
        return ERROR;
```

```
    e=*--S.top;//删除栈顶元素赋给e
```

```
    return OK;
```

```
}
```

```
Status StackTraverse(SqStackS, Status*(visit)(ElemType))
```

```
{ p=S.base;
```

```
    while(S.top>p)    visit(*p++);
```

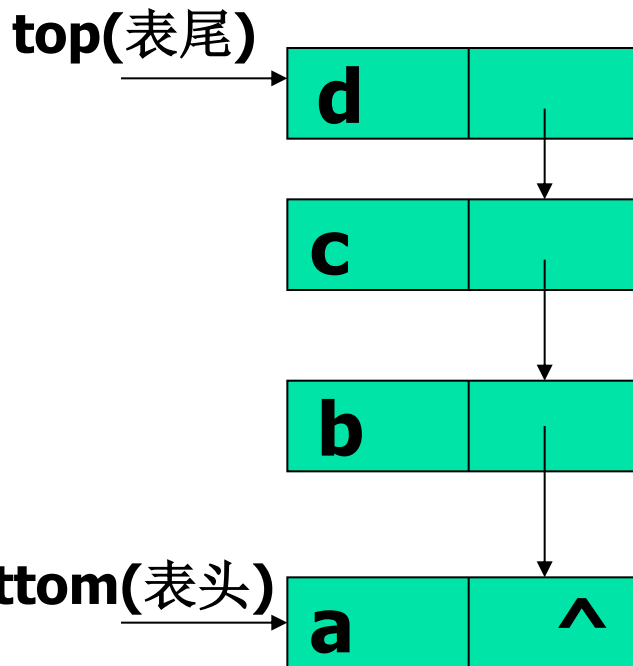
```
    return OK;
```

```
}
```

# 栈的链式存储结构

思考：

1. 结构如何定义？
2. 如何初始化？
3. 入、出栈操作？
4. 如何判断空栈？
5. 链式栈会溢出吗？



```
typedef struct StackNode {  
    ElemType data;           //存放栈中元素  
    struct StackNode *next;  //栈顶指针记录栈顶元素的位置  
} *LinkStack;               //链栈的类型定义  
LinkStack S=NULL;          //空栈
```



## 3.2 栈的应用—字符序列逆置

- 问题：将从键盘输入的字符序列逆置输出

```
void ReverseRead( )
{   InitStack(S);           //初始化栈
    while ((ch=getchar())!='\n') //输入字符，直到输入换行符
        Push(S, ch); //将输入的每个字符入栈
    while (!StackEmpty(S)) //依次退栈并输出退出的字符
    {   Pop(S,&ch);
        putchar(ch);
    }
    putchar('\n');
}
```



## 3.2 栈的应用—数制转换

- 问题:十进制数和其它进制数转换。
- 算法策略: $N=(N \text{ div } d)*d+N \text{ mod } d$
- 例:  $(1348)_{10}=(2504)_8$ , 其运算过程如下:

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

## 3.2 栈的应用—数制转换函数

```
void conversion( )
{   initstack(S);
    scanf ("%d",&n);//读入要转换的数
    while(n)
    {   push(S,n%8);//各个数位的数值入栈
        n=n/8;
    }
    while(! Stackempty(S))
    { pop(S,e);//按高位到低位顺序出栈
        printf("%d",e);
    }
}
```



十进制数转换成二  
进制、十六进制数？



## 3.2 栈的应用—括号匹配的检验

- 括号类型： ( ) [ ] { }
- 正确的匹配格式为  
[ ( ) [ ] ]、[ [ ( ) ] ]、{ ( ) { } [ ] }
- 可能出现的不匹配的情况：
  - ①到来的右括号不是所期待的： [ ]
  - ②只有右括号： } ] )
  - ③直到结束，也没有到来所期待的：([ ] #
- 检验括号匹配的关键：左括号期待右括号的“急迫”程度（**越是后来的越应优先得到匹配**）

## 3.2 栈的应用—括号匹配的检验

```
int BracketsMatch(SqStack &S)
{ c=getchar();
  while(c!='#')    //以 “#”作为表达式结束符
  { if(c=="("||"["||"{")    push(S,c);
    if(c==")"||"]"||"}")
    { if(StackEmpty(S)) {printf(" Missing Left Parenthesis ");return 0;}
      else pop(S,e),
      { if(e和c不匹配)
        {printf("not match");return 0;}
      }
    }
  }
  c=getchar();
}
if(StackEmpty(S)) {printf("match"); return 1;}
else               {printf(" Missing Right Parenthesis ");return 0;}
}
```



### 3.2.3 行编辑程序

- **功能：**接受用户从终端输入的程序或数据存入用户数据区。通常做法是每接受一个字符即存入用户数据区。
- **解决办法：**设立一个输入缓冲区，接受用户输入的一行字符，然后逐行存入用户数据区。允许用户输入出差错，并在发现有误时及时更正。
  - 用一个**退格符** “#”表示前一个字符无效；
  - 用一个**退行符** “@”，表示当前行中的字符均无效。
- **例：**设从终端接受了两行字符：

```
whli##ilr#e (s#*s)
```

```
outcha@putchar(*s=#++);
```

则实际有效的是下列两行：

```
while (*s)
```

```
    putchar(*s++);
```



## 3.2.3 行编辑程序

```
void LineEdit( ) { //利用字符栈S，从终端接收一行并传送至调用过程的数据区
    InitStack(S); //构造空栈
    ch=getchar(); //从终端接收第一个字符
    while(ch != EOF ) { //EOF 为全文结束符
        while(ch != EOF && ch!='\n') {
            switch(ch) {
                case '#': pop(S,c); break; //仅当栈非空时退栈
                case '@': ClearStack(S); break; //重置S为空栈
                default: push(S, ch); break; //有效字符进栈,未考虑栈满 }
            ch=getchar(); //从键盘上接收下一个字符 }
        //将栈底到栈顶的栈内字符传送至调用过程的数据区
        ClearStack(S); //重置S为空栈
        if (ch!=EOF) ch=getchar();
    }
    DestroyStack(S); }
```



## 3.2 算术表达式求值(1)

---

- 计算机中表达式的组成
  - **操作数**(operand): 可以是常数、变量或常量标识符
  - **运算符**(operator): 算术运算符、关系运算符和逻辑运算符
  - **界限符**(delimiter): 左右括号和标识表达式结束的结束符 “#”
- 设置两个栈:
  - 操作数栈 (OPND): 存放表达式中的操作数, 置为空。
  - 运算符栈 (OPTR): 存放表达式中的运算符。栈底虚设 “#”。

## 3.2 算术表达式求值(2)

### ■ 简单算术表达式的求值问题

- (1) 先乘除后加减;
- (2) 同级运算时先左后右;
- (3) 先括号内后括号外。

由规则 (2) , 当 $\theta_1 = \theta_2$ 时, 令 $\theta_1 > \theta_2$ , “(” 除外

由规则 (3) , +、-、\*、/为 $\theta_1$ 时, 优先级均低于 ‘ ( ’ 但高于 ‘ ) ’

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

# 计算机系统处理表达式的步骤

- 从左到右依次读入表达式符号**c**，根据运算规则处理：

- 若**c**是**操作数**，**Push(OPND,c); c=getchar();**

- 若**c**是**运算符**

```
switch(Precede(GetTop(OPTR),c))
```

```
{case '<':{ Push(OPTR,C);  
          c=getchar();  
          break; }
```

```
  case '=':{ Pop(OPTR,c); c=getchar(); break;}
```

```
  case '>':{ a=Pop(OPND); b=Pop(OPND);  
          theta=POP(OPTR);  
          Push(OPND,operate(a,theta,b));  
          break; }
```

```
}
```



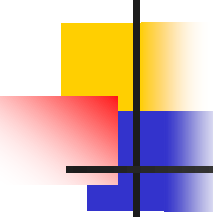
# 算术表达式的算符优先算法

```
OperandType EvaluateExpression() //OP为运算符集合，运算符栈、运算数栈
{ InitStack(OPTR); Push(OPTR,'#');
  InitStack(OPND); c=getchar();
  while(c!='#' || GetTop(OPTR)!='#')
  { if(!In(c,op)){Push(OPND,c); c=getchar();} //非运算符进栈
    else
      switch(Precede(GetTop(OPTR), c))
      { case '<': Push(OPTR, c); c=getchar(); break; //栈顶元素优先权低
        case '=': pop(OPTR,x); c=getchar(); break; //脱括号并接收下一字符
        case '>': pop(OPTR,theta); pop(OPND,b); pop(OPND, a);
                  push(OPND,Operate(a, theta, b)); break; //运算并保存结果}
      }
  }
  return GetTop(OPND);
}
```

# 用栈计算表达式 $5+(6-4/2)*3\#$ 的过程

	)
	#
2	-
<del>3</del>	*
<del>14</del>	+
<del>15</del>	#
OPND	OPTR

$\theta_1 \backslash \theta_2$	+	-	*	/	(	)	#
+	✓	✓	∧	∧	∧	✓	✓
-	✓	✓	∧	∧	∧	✓	✓
*	✓	✓	✓	✓	∧	✓	✓
/	✓	✓	✓	✓	∧	✓	✓
(	∧	∧	∧	∧	∧		
)	✓	✓	✓	✓		✓	✓
#	∧	∧	∧	∧	∧		



## 3.2 栈的应用—函数的嵌套调用

在高级语言编制的程序中，调用函数与被调用函数之间的链接和信息交换通过**栈实现**。当一个函数在运行期间调用另一个函数，在运行被调用函数前，需先完成三件事：

- 1) 将所有的实参、返回地址等信息传递给被调用函数**保存**；
- 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。

从被调用函数返回调用函数之前，应该完成：

- 1)**保存**被调函数的**计算结果**；
  - 2)**释放**被调函数的**数据区**；
  - 3)依照被调函数保存的返回地址将控制**转移**到调用函数。
- 多个函数嵌套调用的规则：**后调用先返回**
  - 内存管理实行“**栈式管理**”。

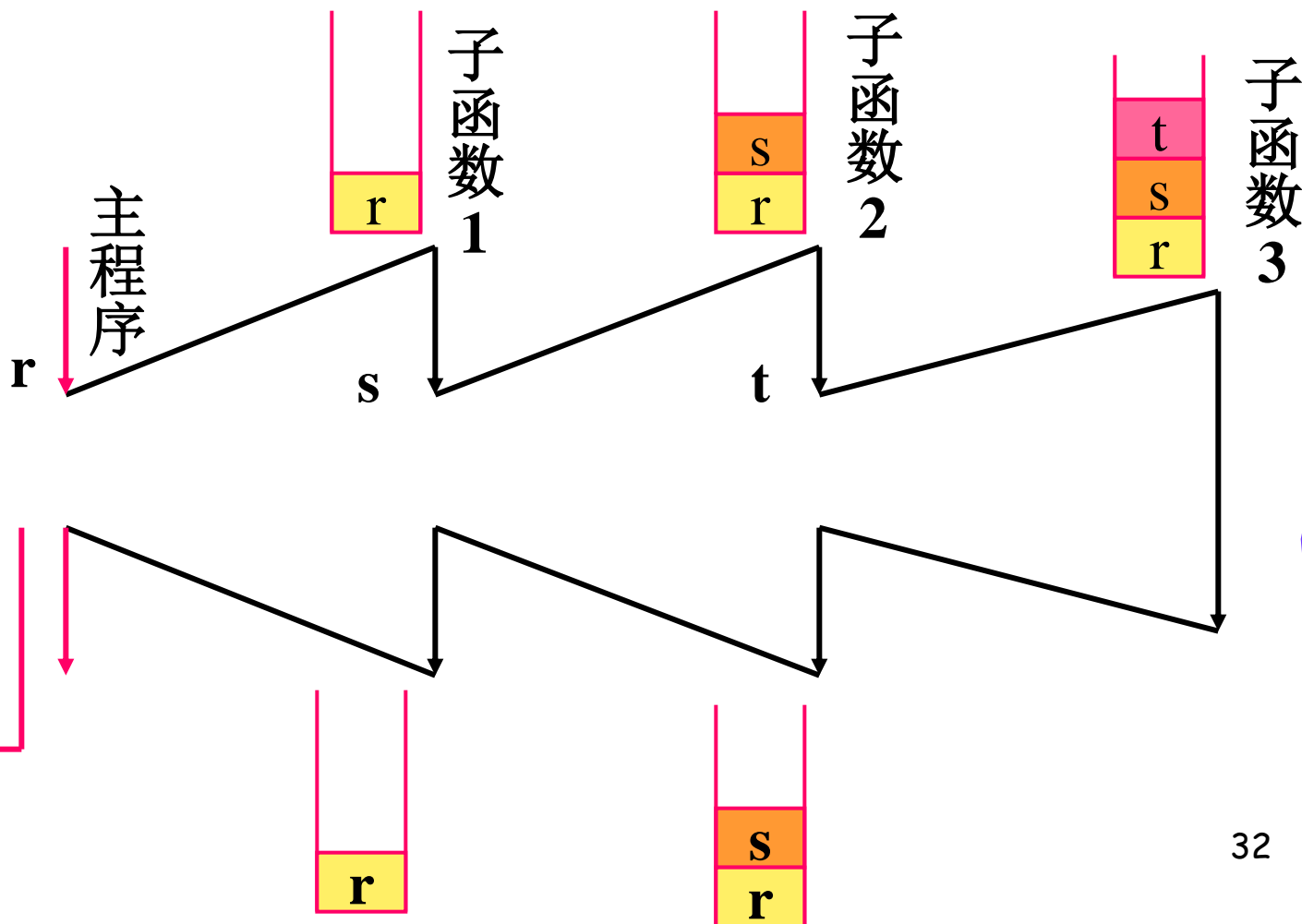


# 栈的应用-函数的嵌套调用

子函数1=NextElem(L,cur\_e,next\_e)

子函数2=LocateElem(L,cur\_e,compare())

子函数3=compare(cur\_e,p->data)



## 3.3 栈与递归的实现



- **递归函数**：一个直接调用自己或通过一系列调用语句间接调用自己的函数。
- **递归的应用**：
  - 数学函数：阶乘函数、2阶Fibonacci数列、Ackman函数
  - 数据结构：二叉树、广义表
  - 八皇后、汉诺塔(Tower of Hanoi)问题
- **递归算法设计**：
  - 将规模较大的原问题分解为一个或多个规模更小的与原问题类似的子问题-**递归步骤**
  - 确定一个或多个无须分解可直接求解的子问题-**终止条件**

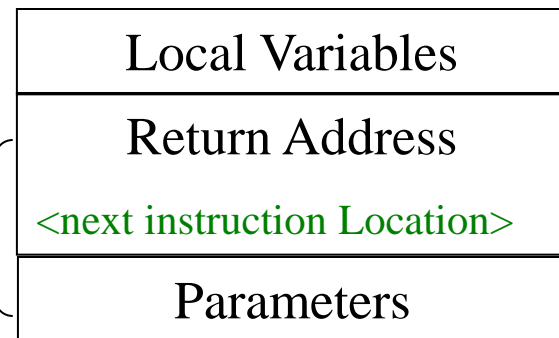
## 3.3 栈与递归的实现

- 递归的实现：建立递归工作栈  
(后调用者先返回)

- 调用时

- 为调用者生成一个活动记录
- 压入运行时刻栈
- 程序控制权转交给被调函数

Activation Record



- 返回时

- 被调函数执行完毕，运行时刻栈顶的活动结构退栈
- 根据退栈的活动结构中保存的返回地址将程序控制权交给调用者继续执行

### 3.3 栈与递归的实现-阶乘函数

```
int fact(int n)
{  if (n==0)
    {  s=1; return 1;}
  else
    {  s=n*fact(n-1); ←—ReLoc2
      return s;
    }
}

void main()
{  int N= fact(5);} ←—ReLoc1
```

$$n! = \begin{cases} 1 & (n = 0) \\ n * (n-1)! & (n \geq 1) \end{cases}$$

调用者	参数表	返回地址
fact(1)	0	ReLoc2
fact(2)	1	ReLoc2
fact(3)	2	ReLoc2
fact(4)	3	ReLoc2
fact(5)	4	ReLoc2
main()	5	ReLoc1

活动记录进栈示意图

## 3.3 栈与递归的实现-阶乘函数

调用者

fact(1)

fact(2)

fact(3)

fact(4)

fact(5)

main()

参数表

0

1

2

3

4

5

返回地址

ReLoc2

ReLoc2

ReLoc2

ReLoc2

ReLoc2

ReLoc1

0 ReLoc2

s=1\*fact(0)

return s; //fact(1)返回1

1 ReLoc2

s=2\*fact(1)

return s; //fact(2)返回2

2 ReLoc2

s=3\*fact(2)

return s; //fact(3)返回6

3 ReLoc2

s=4\*fact(3)

return s; //fact(4)返回24

4 ReLoc2

s=5\*fact(4)

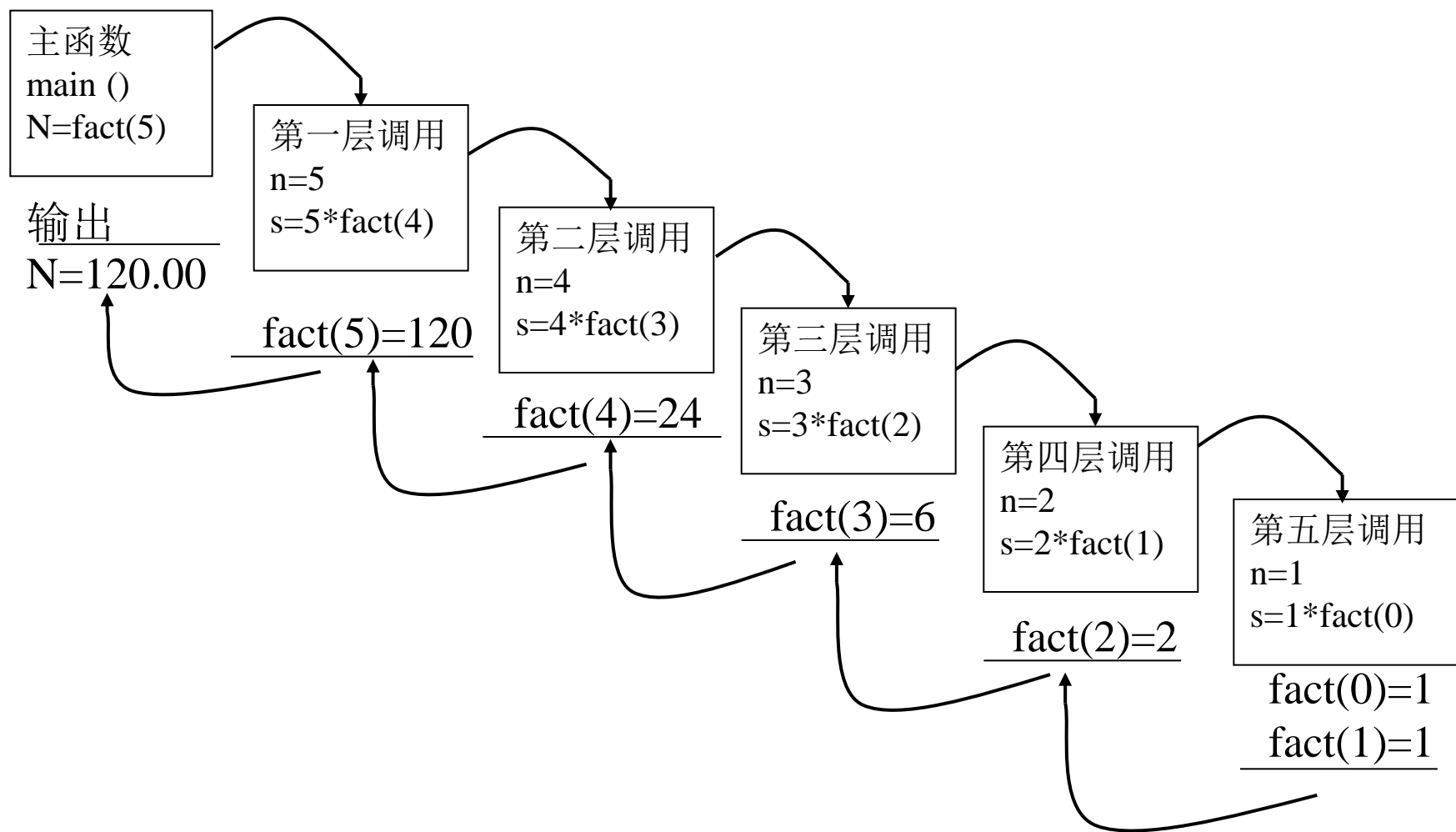
return s; //fact(5)返回120

5 ReLoc1

s=120; //来自fact(5)

活动记录进栈示意图

活动记录退栈过程示意图<sup>36</sup>



递归调用过程示意图

从图中可看到fact函数共被调用5次，即fact(5)、fact(4)、fact(3)、fact(2)、fact(1)。其中，fact(5)被主函数调用，其它在fact函数内被调用。每一次递归调用并未立即得到结果，而是进一步向深度递归调用，直到n=1时，函数fact结果为1，然后再一一返回计算，得到最终结果。

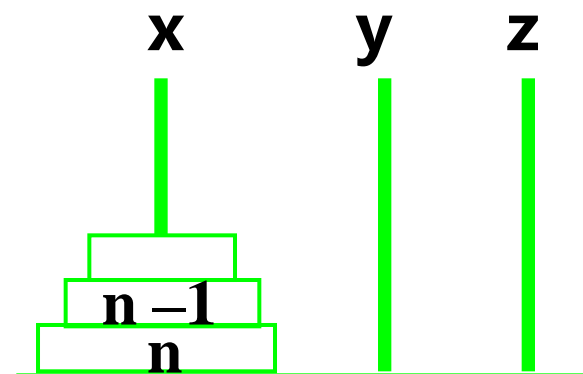
### 3.3 栈与递归的实现例3—汉诺塔问题

传说创世纪时有个叫Brahma的寺庙，有三个柱子，一柱上从小到大依次叠放64个盘子，僧侣的工作是将这64个盘子从一根柱子移到另一个柱子上。

**移动规则：**

- 每次只能移动一个盘子；
- 只能小盘子在大盘子上面；
- 可以使用任一柱子。

当工作做完之后，世界将永远和平。



**分析：**

设三根柱子分别为  $x$ ,  $y$ ,  $z$ ，盘子在  $x$  柱上，要移到  $z$  柱上。

1、当  $n=1$  时，盘子直接从  $x$  柱移到  $z$  柱上；

2、当  $n>1$  时，

- ① 把前  $n-1$  个盘子借助  $z$  柱从  $x$  柱移到  $y$  柱上；
- ② 把第  $n$  号盘子从  $x$  柱移到  $z$  柱上；
- ② 把  $n-1$  个盘子借助  $x$  柱从  $y$  柱移到  $z$  柱上。



## 3.3 栈与递归的实现例3 – 汉诺塔问题

---

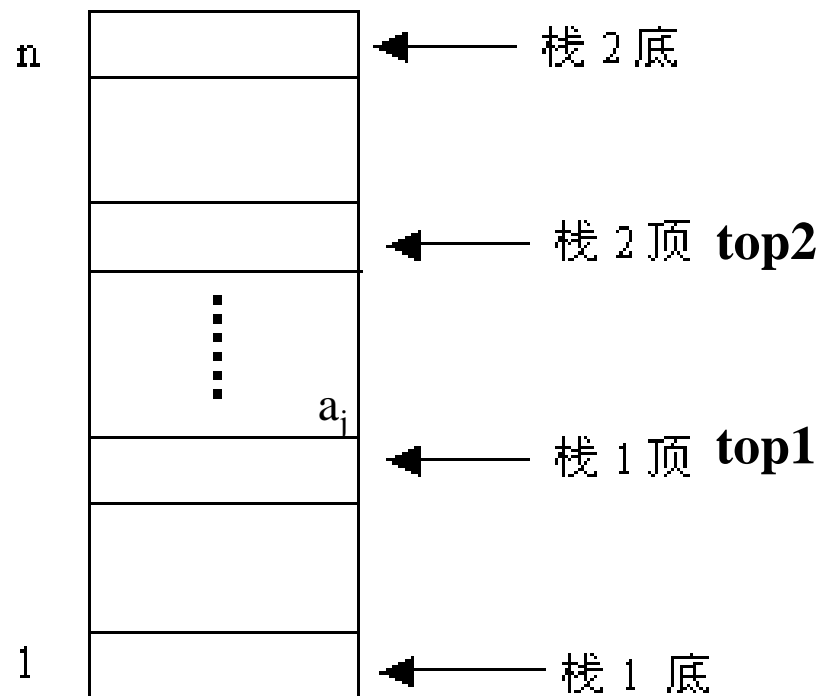
```
void Hanoi( int n, char x, char y, char z )
{ //将 n 个 编号从上到下为 1...n 的盘子从 x 柱，借助 y 柱移到 z 柱
  if (n == 1)
    move(x, 1, z) ; //将编号为 1 的盘子从 x 柱移到 z 柱
  else
  { //将 n-1个 编号从上到下为1...n-1的盘子从 x 柱，借助 y 柱移到 z 柱
    Hanoi(n-1, x, z, y) ;
    move(x, n, z) ; //将编号为 n 的盘子从 x 柱移到 z 柱
    //将 n-1个 编号从上到下为 1...n-1的盘子从 y 柱，借助 x 柱移到 z 柱
    Hanoi(n-1, y, x, z);
  }
} //Hanoi
```



# 两个栈共享数组空间(1)

当程序中同时使用两个栈时，可以将两个栈的栈底设在数组的两端，让两个栈各自向中间延伸。只有当整个空间被两个栈占满（即两个栈顶相遇）时，才会发生上溢。

两个栈共享一个长度为 $n$ 的向量空间和两个栈分别占用两个长度为 $\lfloor n/2 \rfloor$ 和 $\lceil n/2 \rceil$ 的空间比较，前者**发生上溢的概率比后者要小得多**



共享空间的双向栈示意图



## 两个栈共享数组空间(2)

---

- 双向栈存储结构

```
#define MAXSTACKSIZE 100
```

```
typedef struct
```

```
{ ElemType elem[MAXSTACKSIZE];
```

```
    int top1;
```

```
    int top2;
```

```
    int flag;
```

```
}DulSqStack;
```

- 判空判满条件

- **S.top1==0&&S.top2==MAXSTACKSIZE** //两个栈同时空

- **(S.top1+1)==S.top2或S.top1>S.top2** //两个栈同时满



## 双向栈的入栈操作

```
Status DuSqStackPush(DuSqStack &S, ElemType x)
{
    if((S.top1+1)==S.top2) //栈满
        return ERROR;
    if((S.flag!=1)&&(S.flag!=2)) //入的栈不是栈1和栈2
        return ERROR;
    else
    {
        switch(S.flag)
        {
            case 1: S.elem[S.top1]=x; //入栈1
                    S.top1++;
                    break;
            case 2: S.elem[S.top2]=x;
                    S.top2--;
                    break;
        }
    }
    return OK;
}
```



## 双向栈的出栈操作

---

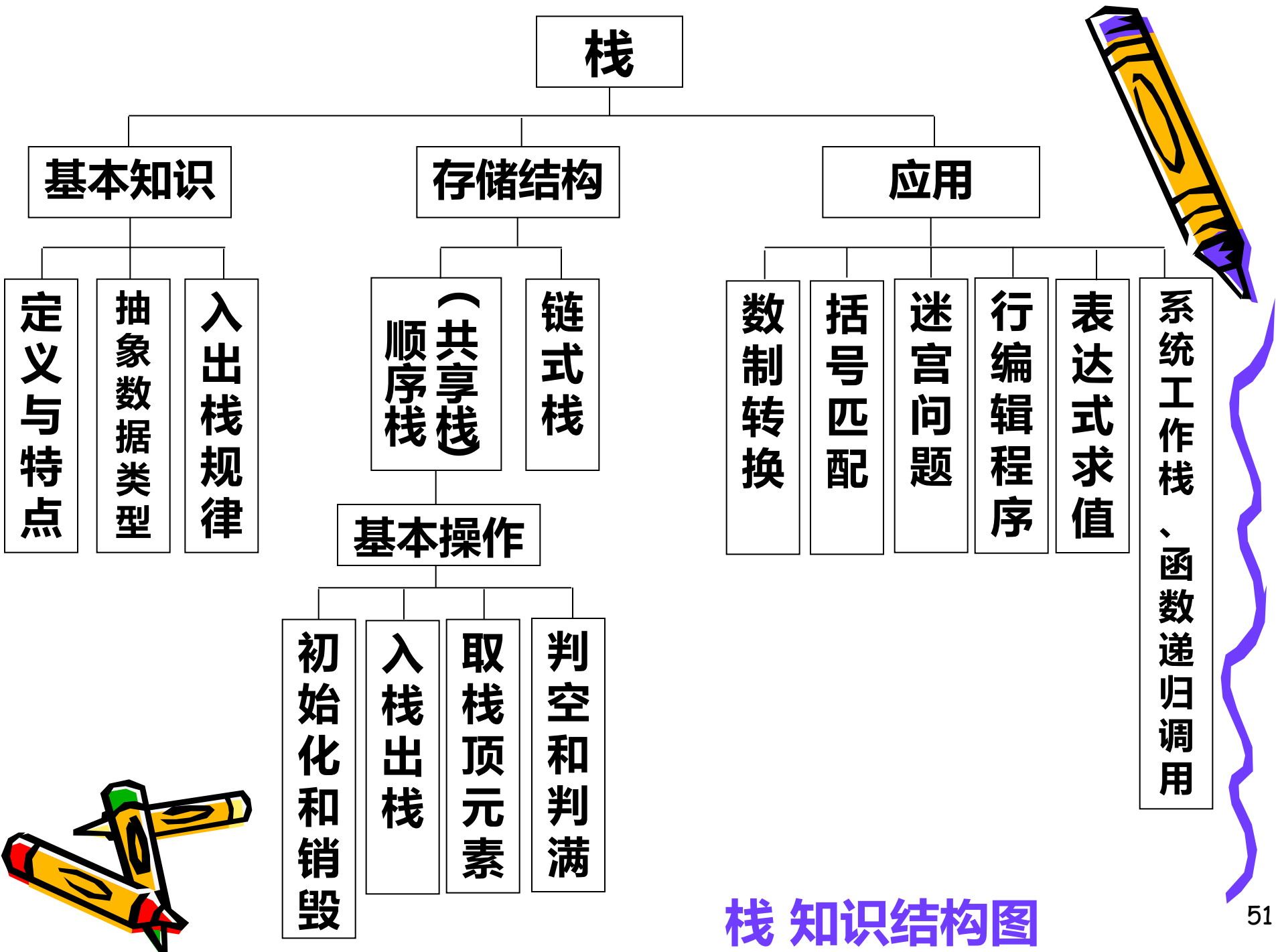
```
Status DuSqStackPop(DuSqStack &S, ElemType &x)
{ if((S.flag!=1)&&(S.flag!=2)) return ERROR; // 出栈不是栈1和栈2
  else
  { switch(S.flag)
    { case 1: if(S.top1>0)
              { S.top1--; x=S.elem[S.top1]; }
              else return ERROR;
              break;
      case 2: if(S.top2<MAXSTACKSIZE-1)
              { S.top2++; x=S.elem[S.top2]; }
              else return ERROR;
              break;
    } // switch
  } // else
  return OK;
}
```



# 双向栈元素输出操作

---

```
Status Print(DuSqStack &S)
{
    int p;
    switch(S.flag)
    {
        case 1: p=0; while(p<S.top1)
            {
                printf("%d",S.elem[p]);
                p++;
            }
            break;
        case 2: p=MAXSTACKSIZE;
            while(p>S.top2)
            {printf("%d",S.elem[p]);p--;}
            break;
    }
    return OK;
}
```



栈 知识结构图



## 2022年408数据结构真题

2. 给定有限符号集  $S$ ,  $in$  和  $out$  均为  $S$  中所有元素的任意排列。对于初始为空的栈  $ST$ , 下列叙述中, 正确的是
- A. 若  $in$  是  $ST$  的入栈序列, 则不能判断  $out$  是否为其可能的出栈序列
  - B. 若  $out$  是  $ST$  的出栈序列, 则不能判断  $in$  是否为其可能的入栈序列
  - C. 若  $in$  是  $ST$  的入栈序列,  $out$  是对应  $in$  的出栈序列, 则  $in$  与  $out$  一定不同
  - D. 若  $in$  是  $ST$  的入栈序列,  $out$  是对应  $in$  的出栈序列, 则  $in$  与  $out$  可能互为倒序



# 2020年408考试真题

02. 对空栈  $S$  进行 Push 和 Pop 操作，入栈序列为  $a, b, c, d, e$ ，经过 Push, Push, Pop, Push, Pop, Push, Push, Pop 操作后得到的出栈序列是 ( )。
- A.  $b, a, c$       B.  $b, a, e$       C.  $b, c, a$       D.  $b, c, e$

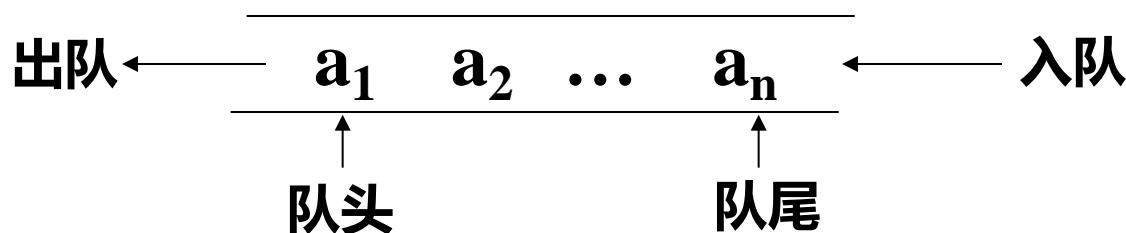
## 补充类型题

- 1 对单词bookkeeper，按字母顺序进栈，有多少种出栈顺序使其依然是原单词？
- A 8      B 20      C 12      D 16
- 2 若采用带头、尾指针的线性单链表表示一个栈，栈顶指针top设置在哪里比较合适？
- A 头指针处      B 尾指针处      C 头、尾均可      D 不确定



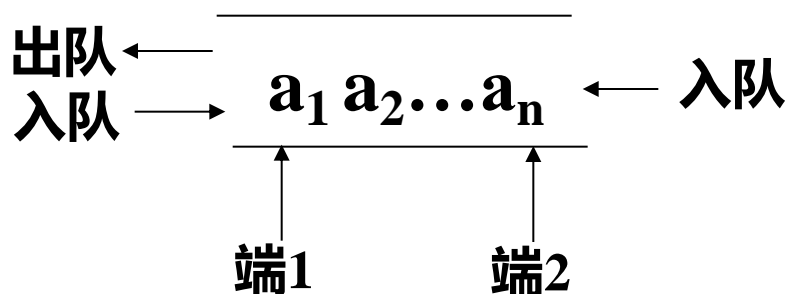
## 3.4 队列

- **队列(Queue)**: 只允许在表的一端插入, 在另一端删除的线性表。假设 $Q=\{a_1, a_2, \dots, a_n\}$ 
  - **队头(front)**: 允许删除的一端。
  - **队尾(rear)**: 允许插入的一端。
  - **空队列**: 队列中没有元素。
- 队列的修改的原则: **先进先出**(First In First Out, FIFO)
- 队列示意图

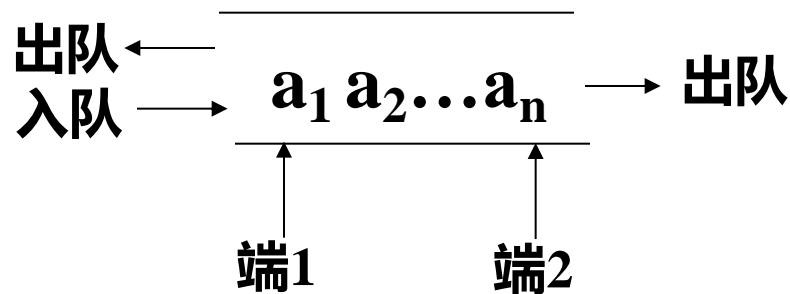


## 3.4 队列-双端队列

- 双端队列：限定输入和删除操作在表两端进行的线性表。
  - 输出受限队列：一端允许插入和删除，另一端只允许插入。
  - 输入受限队列：一端允许插入和删除，另一端只允许删除。



输出受限的双端队列



输入受限的双端队列



## 3.4.1 队列-抽象数据类型定义

ADT Queue {

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

**数据关系:**  $R1 = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i=1,2,\dots,n \}$

**基本操作:** InitQueue(&Q) , DestroyQueue(&Q)

ClearQueue(&Q) , QueueEmpty(Q)

**EnQueue(&Q, e), DeQueue(&Q, &e),**

QueueLength(Q), GetHead(Q, &e)

QueueTraverse(Q, visit())

}ADT Queue



## 3.4.2 队列的链式表示——链队列

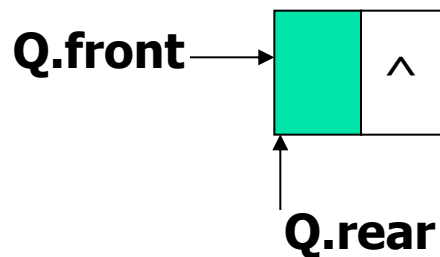
---

- 链队列：用链表表示的队列。
- 链队列的存储结构

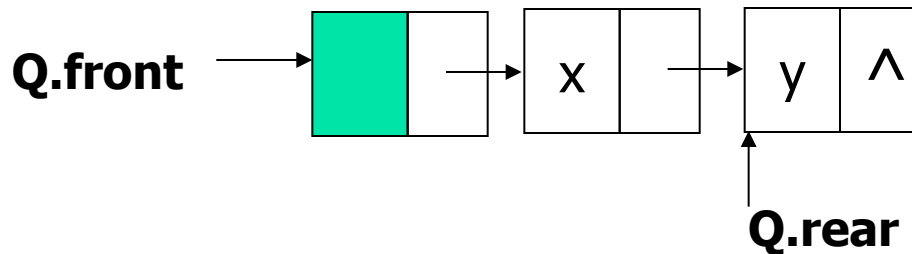
```
typedef struct QNode
{
    QelemType data;
    struct QNode *next;
}QNode,*QueuePtr;
typedef struct
{
    QueuePtr front; //队头指针
    QueuePtr rear;  //队尾指针
}LinkQueue;
```

## 3.4.2 队列的链式表示——链队列

### ■ 链队列示意图



(a) 空链队列



(b) 非空链队列

- 链队列的判空条件:  $Q.front == Q.rear$



## 3.4.2 队列的链式表示-链队列基本操作

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

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

```
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;  
}
```

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

# 练习题

## ● 综合应用题

1. 设以带头结点的循环链表表示队列, 并且只设一个指针指向队尾元素结点(注意不设头指针), 循环链表的类型定义如下:

【广东工业大学2017年】

```
typedef struct LNode {
```

```
    ElemType data
```

```
    struct LNode *next;
```

```
} LNode, *Linklist;
```

算法f4实现相应的出队操作。请在空缺处填入合适的内容, 使其成为完整的算法。

```
status f4(LinkList& rear, ElemType &e){
```

```
    if ( (1) )
```

```
        return ERROR;
```

```
        p=rear->next->next;
```

```
        (2)=p->next;
```

```
        if ( (3) )
```

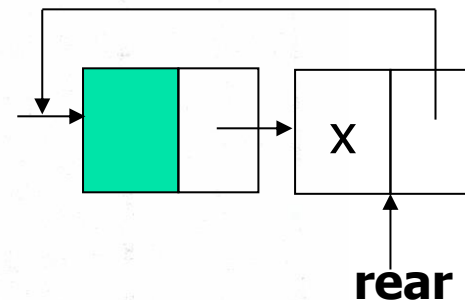
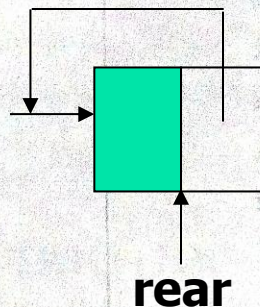
```
            rear=(4)
```

```
            e=p->data;
```

```
            free(p);
```

```
            return OK;
```

```
        }
```





### 3.4.3 循环队列—队列的顺序表示和实现

- 队列的顺序表示：用一组地址**连续**的存储单元**依次**存储从队头到队尾的数据元素。
- 队列顺序存储结构的实现

```
#define MAXQSIZE 100
```

```
typedef struct {
```

```
    QElemType *base; //初始空间基地址
```

```
    int front;        //队头指针，指向队头元素
```

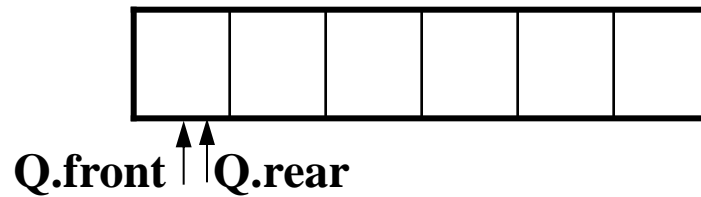
```
    int rear;         //队尾指针指向队尾元素下一个位置
```

```
}SqQueue;
```





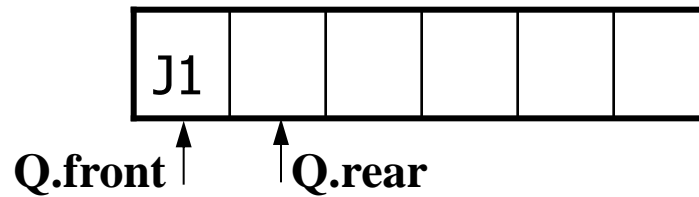
# 头、尾指针和队列中元素的关系



**初始化空队列:  $Q.front == Q.rear$**



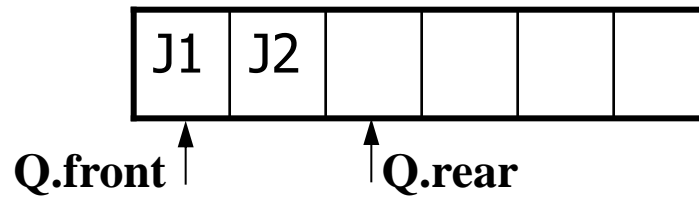
# 头、尾指针和队列中元素的关系



入队列:  $Q.rear++$  (尾指针增 1)



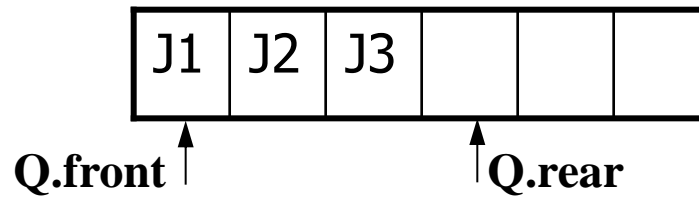
# 头、尾指针和队列中元素的关系



入队列:  $Q.rear++$  (尾指针增 1)



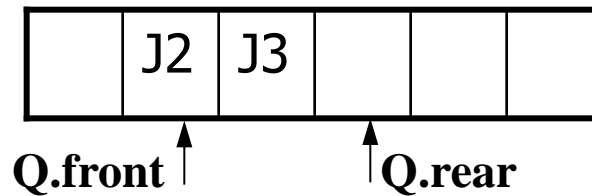
# 头、尾指针和队列中元素的关系



**入队列:  $Q.rear++$  (尾指针增 1)**



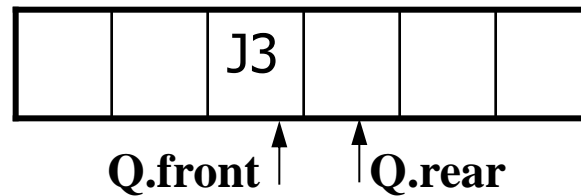
# 头、尾指针和队列中元素的关系



**出队列:  $Q.front++$  (头指针增 1)**



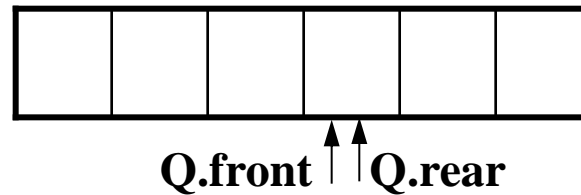
# 头、尾指针和队列中元素的关系



**出队列:  $Q.front++$  (头指针增 1)**



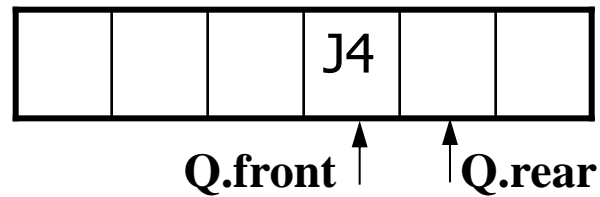
# 头、尾指针和队列中元素的关系



**队列空:  $Q.front == Q.rear$**



# 头、尾指针和队列中元素的关系

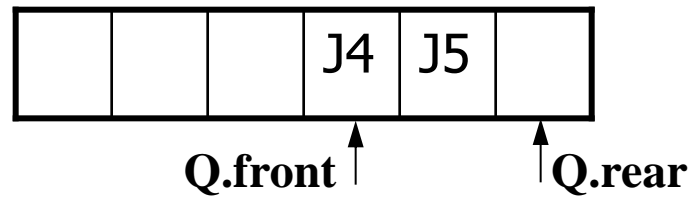


入队列:  $Q.rear++$  (尾指针增 1)





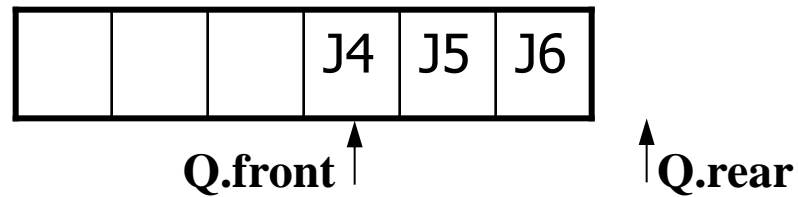
# 头、尾指针和队列中元素的关系



入队列:  $Q.rear++$  (尾指针增 1)



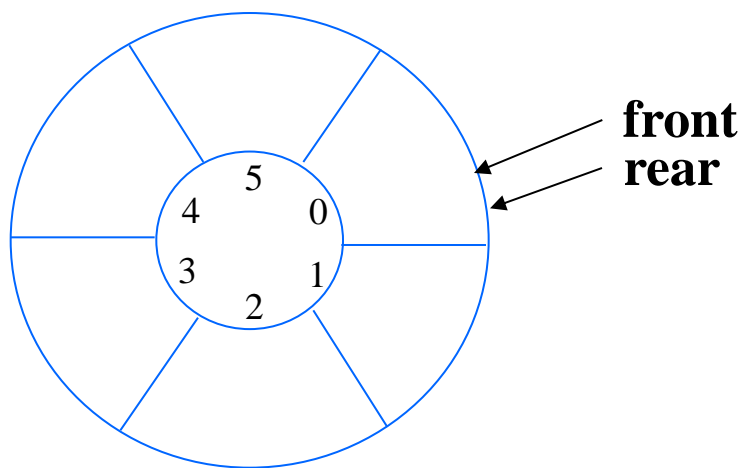
# 头、尾指针和队列中元素的关系



入队列:  $Q.rear++$  (尾指针增 1)

■ 假上溢现象

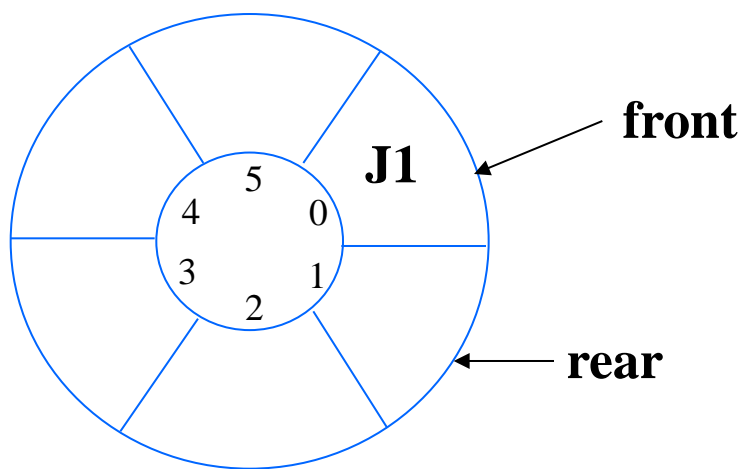
# 循环队列入队列和出队列示意图



初始状态

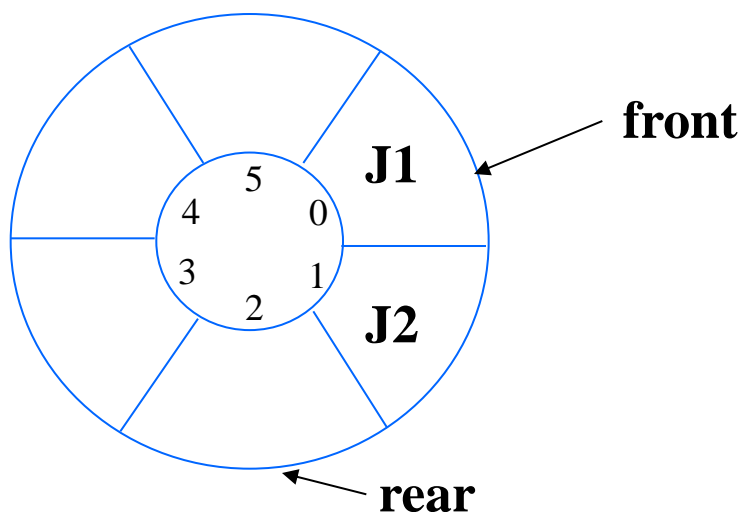
队列空:  $Q.front == Q.rear$

# 循环队列入队列和出队列示意图



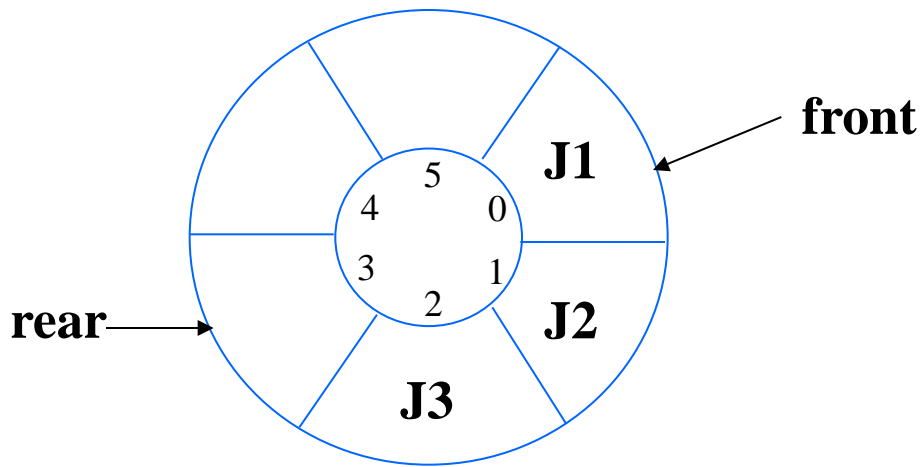
**入队列:  $Q.rear = (Q.rear + 1) \% MAXQSIZE$**

# 循环队列入队列和出队列示意图



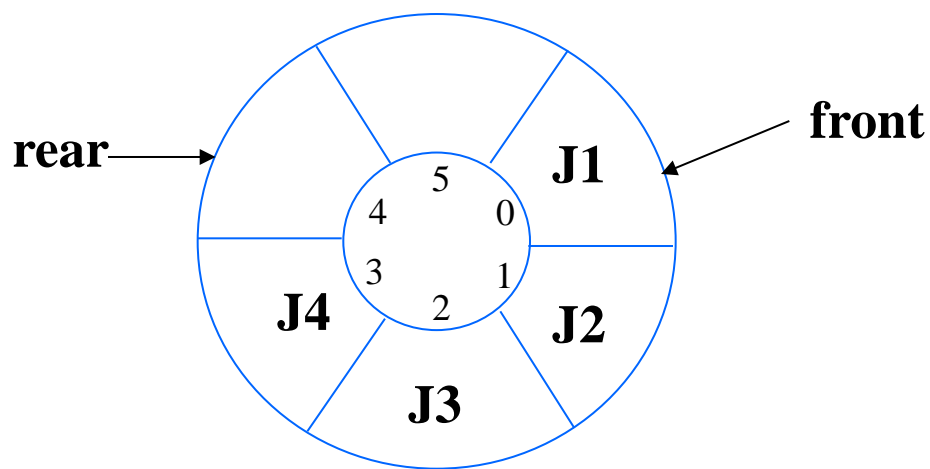
**入队列:  $Q.rear = (Q.rear + 1) \% MAXQSIZE$**

# 循环队列入队列和出队列示意图



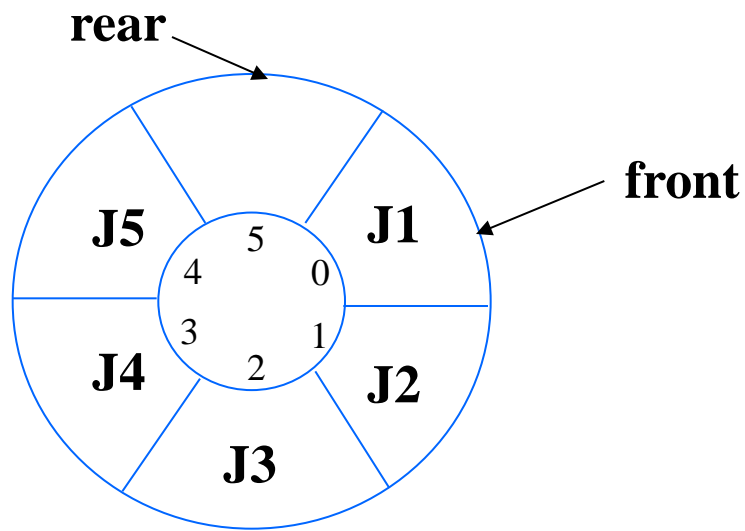
**入队列:  $Q.rear = (Q.rear + 1) \% MAXQSIZE$**

# 循环队列入队列和出队列示意图



**入队列:  $Q.rear = (Q.rear + 1) \% MAXQSIZE$**

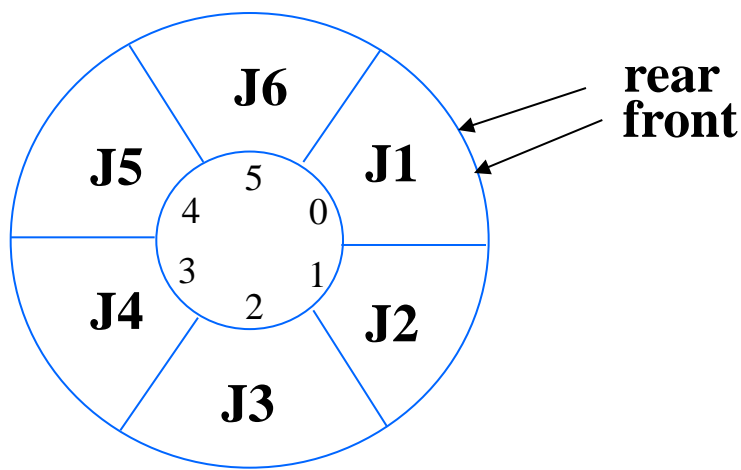
# 循环队列入队列和出队列示意图



**入队列:  $Q.rear = (Q.rear + 1) \% MAXQSIZE$**



# 循环队列入队列和出队列示意图



**队列满:  $Q.front == Q.rear$**

队列判空和判满条件相同，解决方法：

1) 另设标志位区别队列空、满

$Q.rear = Q.front$        $Q.rear = Q.front$

$\left\{ \begin{array}{l} S=0 \end{array} \right.$

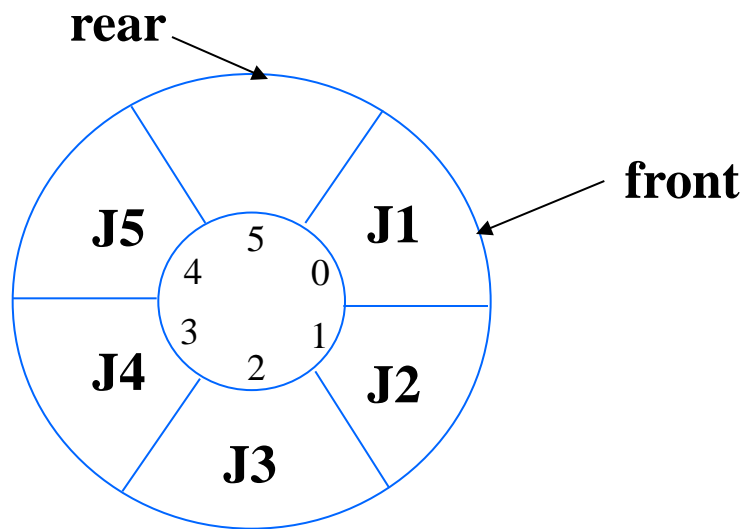
**队列空**

$\left\{ \begin{array}{l} S=1 \end{array} \right.$

**队列满**

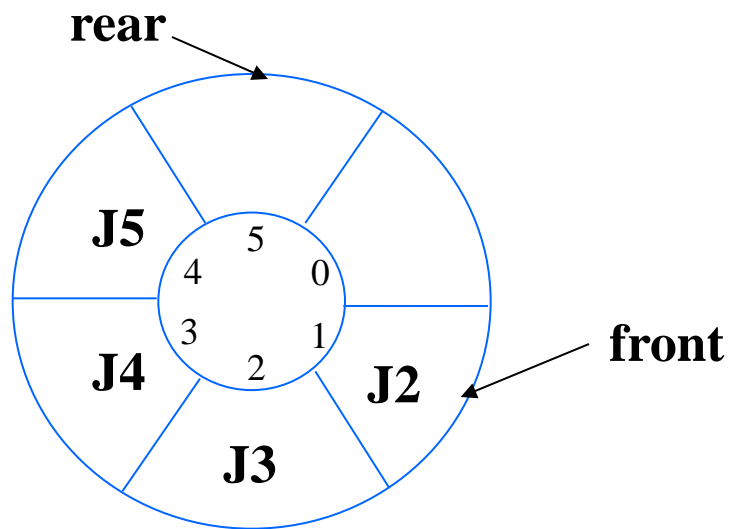
2) 少用一个元素空间，约定队头指针在队尾指针下一个位置为队列满。

# 循环队列入队列和出队列示意图



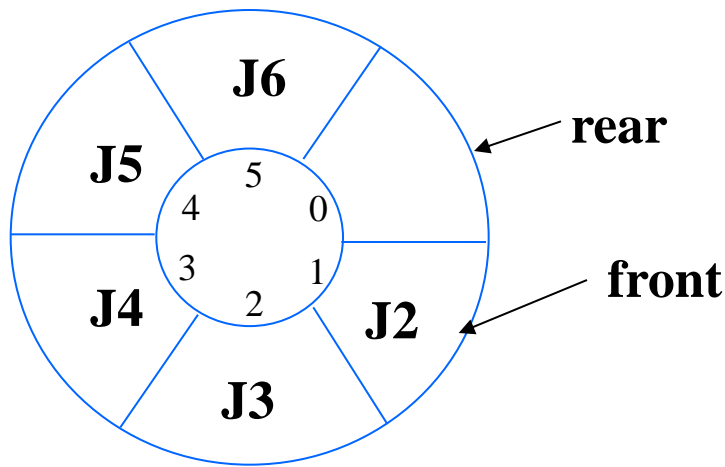
**队列满:  $(Q.rear+1)\%MAXQSIZE==Q.front$**

# 循环队列入队列和出队列示意图



**出队列:  $Q.front = (Q.front + 1) \% MAXQSIZE$**

# 循环队列入队列和出队列示意图



- 在C语言中不能用动态分配的一维数组实现循环队列。
- 使用循环队列，必须为它设定最大长度。
- 若无法估计队列长度，宜采用链队列。

**入队列:  $Q.rear=(Q.rear+1)\%MAXQSIZE$**



## 循环队列基本操作的实现(1)

---

```
Status InitQueue(SqQueue &Q) { // 初始化队列Q
    Q.base=(QElemType)malloc(MAXQSIZE*sizeof(QElemType));
    if(!Q.base)
        exit(OVERFLOW);
    Q.front = Q.rear=0;
    return OK;
}

int QueueLength(SqQueue Q)
{ // 返回Q的元素个数，即队列的长度
    return (Q.rear- Q.front+MAXSIZE)%MAXSIZE;
}
```



## 循环队列基本操作的实现(2)

```
Status EnQueue(SqQueue &Q, QElemType e)  
{ //队列空间未满，入队列  
    if((Q.rear+1)%MAXSIZE==Q.front)    return ERROR;  
    Q.base[Q.rear]=e;  
    Q.rear=(Q.rear+1)%MAXSIZE;  
    return OK;  
}
```

```
Status DeQueue(SqQueue &Q, QElemType &e)  
{ //队列不空删去队头，用e返回值  
    if (Q.front==Q.rear)    return ERROR;  
    e=Q.base[Q.front];  
    Q.front=(Q.front+1)%MAXSIZE;  
    return OK;  
}
```



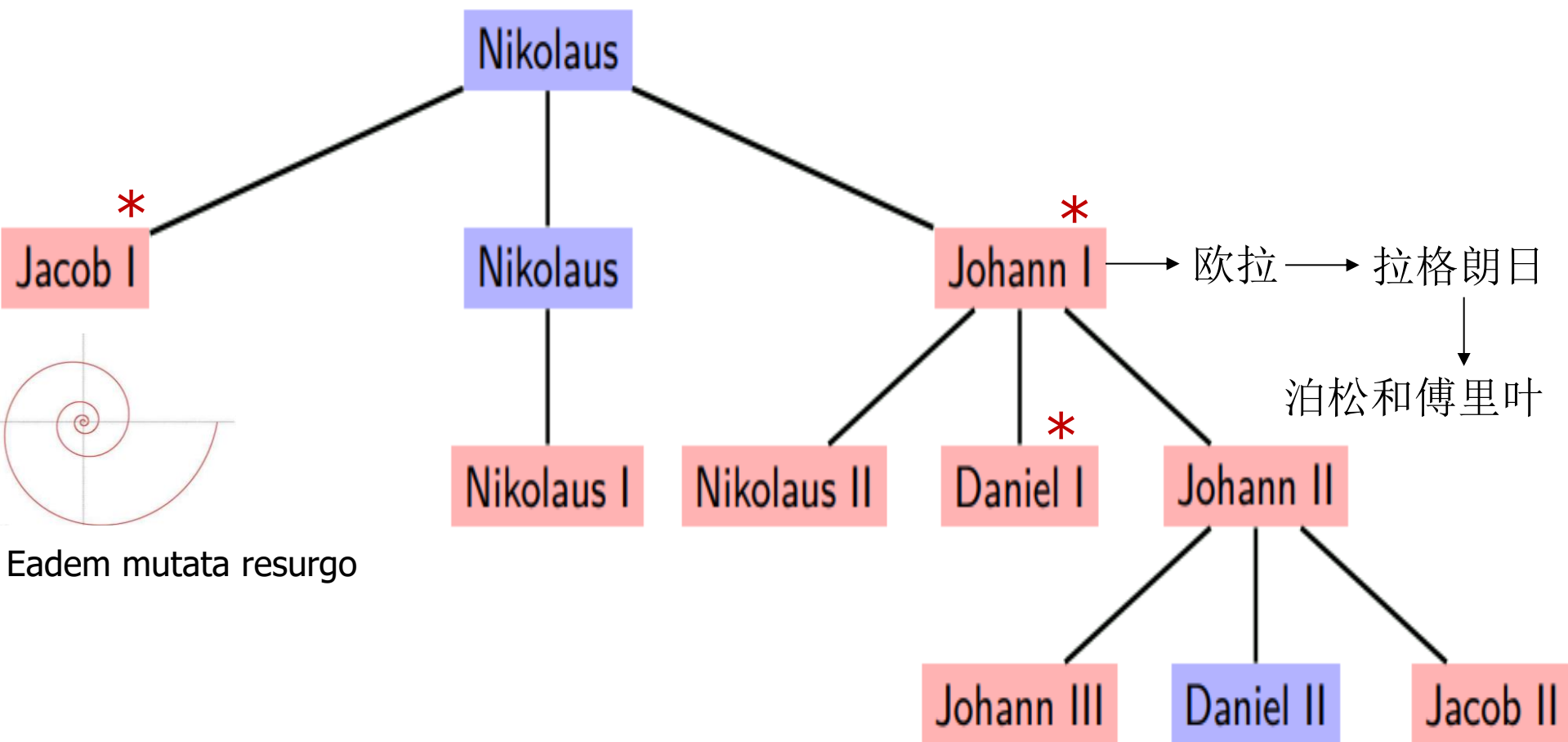
# 队列的应用

---

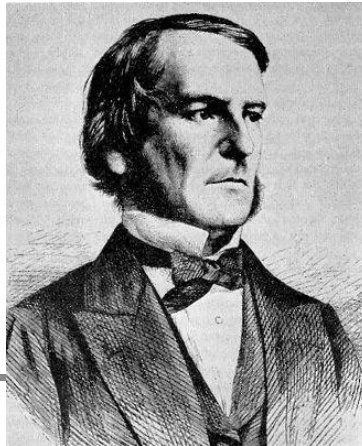
1. 舞伴问题
2. 解决主机与打印机之间不匹配问题
3. 飞机起降问题、医院挂号问题、银行排队问题
4. 多用户系统
5. 优先级队列
6. 树的层序遍历
7. 图的广度优先遍历
8. 基数排序
9. 一些数学问题

# 队列应用

问题：按年龄大小输出伯努利家族所有成员。







George Boole

与Mary Everest结婚

Mary Ellen 与数学家Charles Howard Hinton结婚

George Hinton

Howard Everest Hinton 著名昆虫学家，1961年当选英国皇家学会会士

Geoffrey Everest Hinton



1970年，英国剑桥大学实验心理学学士学位

1978年，爱丁堡大学人工智能博士学位

1986年，作为主要贡献者提出BP算法

1998年，英国皇家学会会士

2006年，提出深度信念网络

2018年，图灵奖

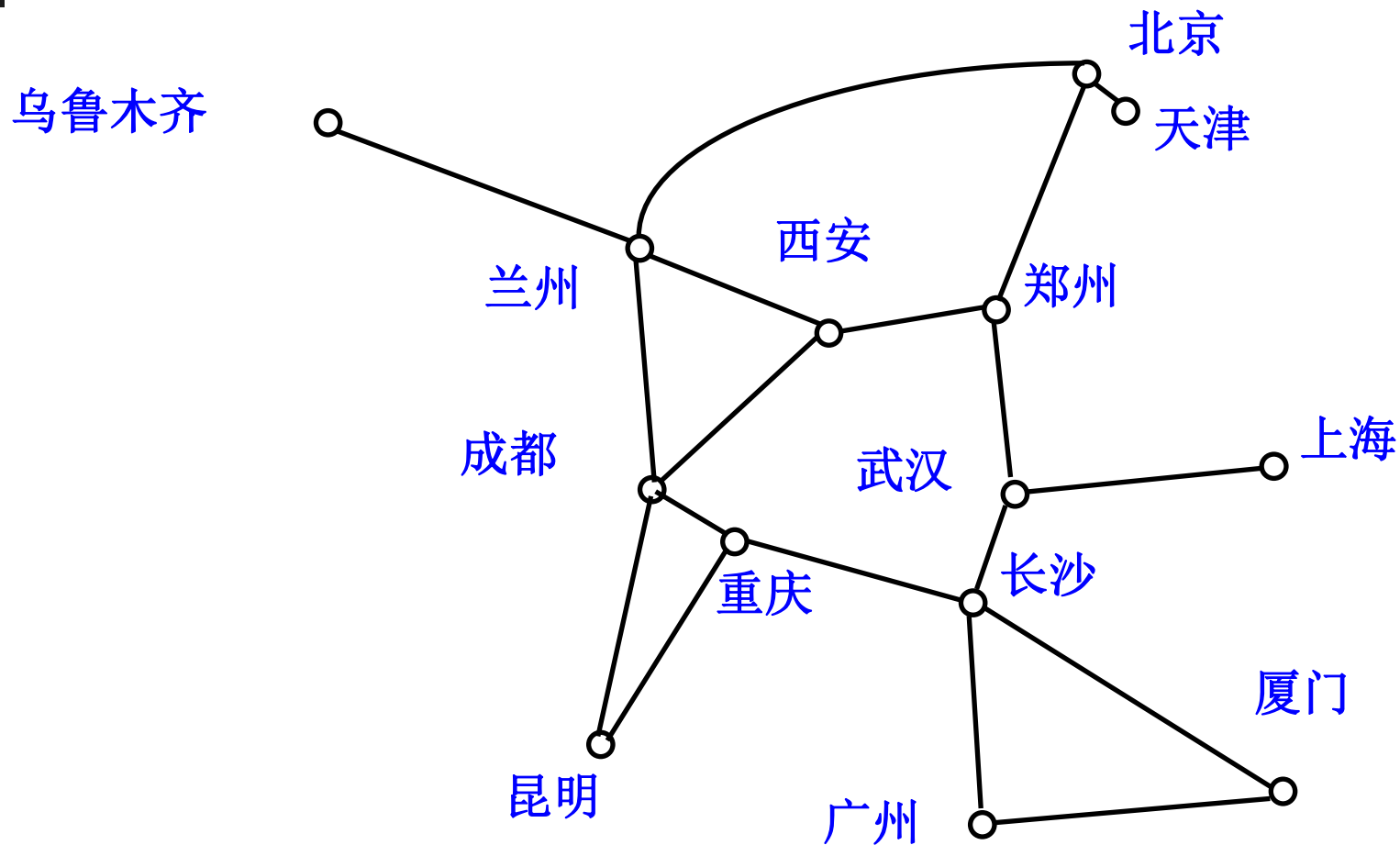
2024年，诺贝尔物理学奖



# 清华大学招博士研究生的条件



# 队列应用-图的广度优先遍历



# 2014年408真题

3. 循环队列放在一维数组  $A[0 \dots M-1]$  中,  $end1$  指向队头元素,  $end2$  指向队尾元素的后一个位置。假设队列两端均可进行入队和出队操作, 队列中最多能容纳  $M-1$  个元素。初始时空。下列判断队空和队满的条件中, 正确的是\_\_\_\_\_。

- A. 队空:  $end1 == end2$ ;                      队满:  $end1 == (end2+1) \bmod M$
- B. 队空:  $end1 == end2$ ;                      队满:  $end2 == (end1+1) \bmod (M-1)$
- C. 队空:  $end2 == (end1+1) \bmod M$ ;    队满:  $end1 == (end2+1) \bmod M$
- D. 队空:  $end1 == (end2+1) \bmod M$ ;    队满:  $end2 == (end1+1) \bmod (M-1)$

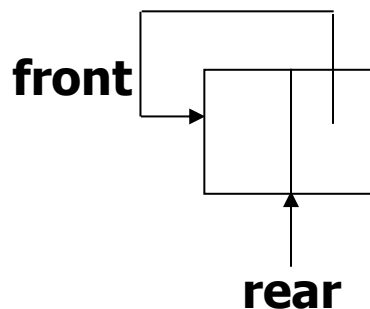
# 2021年408考试真题

02. 已知初始为空的队列  $Q$  的一端仅能进行入队操作, 另外一端既能进行入队操作又能进行出队操作。若  $Q$  的入队序列是 1, 2, 3, 4, 5, 则不能得到的出队序列是 ( )。
- A. 5, 4, 3, 1, 2      B. 5, 3, 1, 2, 4      C. 4, 2, 1, 3, 5      D. 4, 1, 3, 2, 5

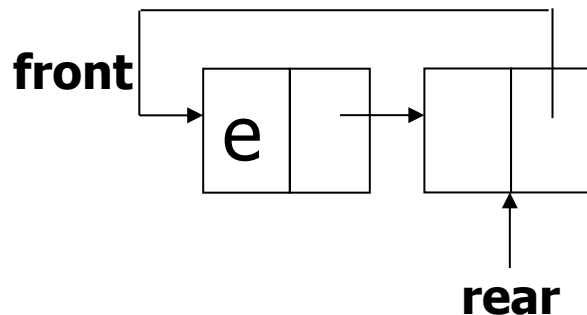
# 2019年408真题

42. (10分) 请设计一个队列，要求满足：①初始时队列为空；②入队时，允许增加队列占用空间；③出队后，出队元素所占用的空间可重复使用，即整个队列所占用的空间只增不减；④入队操作和出队操作的时间复杂度始终保持为 $O(1)$ 。请回答下列问题：

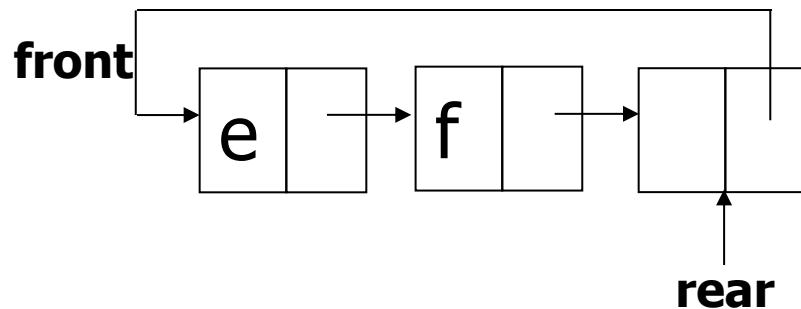
- (1) 该队列应该选择链式存储结构，还是顺序存储结构？
- (2) 画出队列的初始状态，并给出判断队空和队满的条件
- (3) 画出第一个元素入队后的队列状态。
- (4) 给出入队操作和出队操作的基本过程。



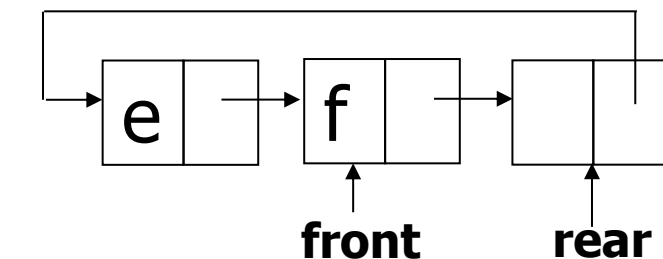
(a) 空队列



(b) e入队列



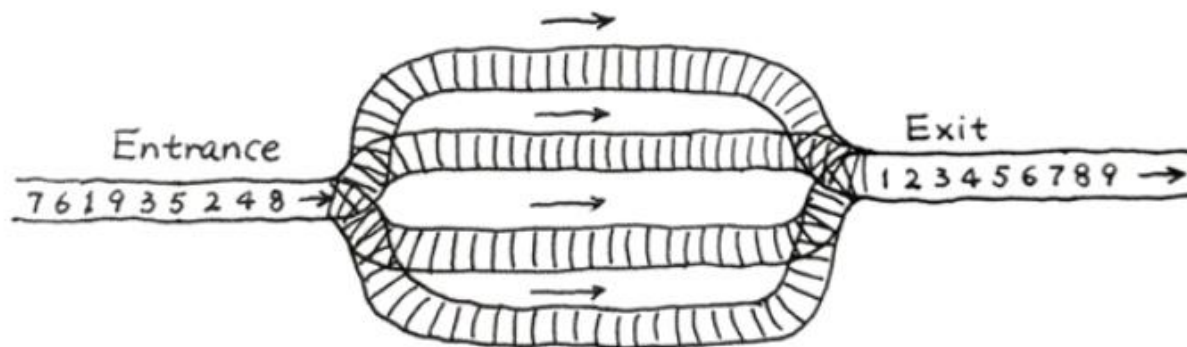
(c) f入队列



(d) e出队列



火车站的列车调度铁轨的结构如下图所示。



两端分别是一条入口 (Entrance) 轨道和一条出口 (Exit) 轨道，它们之间有  $N$  条平行的轨道。每趟列车从入口可以选择任意一条轨道进入，最后从出口离开。在图中有9趟列车，在入口处按照{8, 4, 2, 5, 3, 9, 1, 6, 7}的顺序排队等待进入。如果要求它们必须按序号递减的顺序从出口离开，则至少需要多少条平行铁轨用于调度？

**输入格式：**

输入第一行给出一个整数  $N$  ( $2 \leq N \leq 10^5$ )，下一行给出从1到  $N$  的整数序号的一个重排列。数字间以空格分隔。

**输出格式：**

在一行中输出可以将输入的列车按序号递减的顺序调离所需要的最少的铁轨条数。

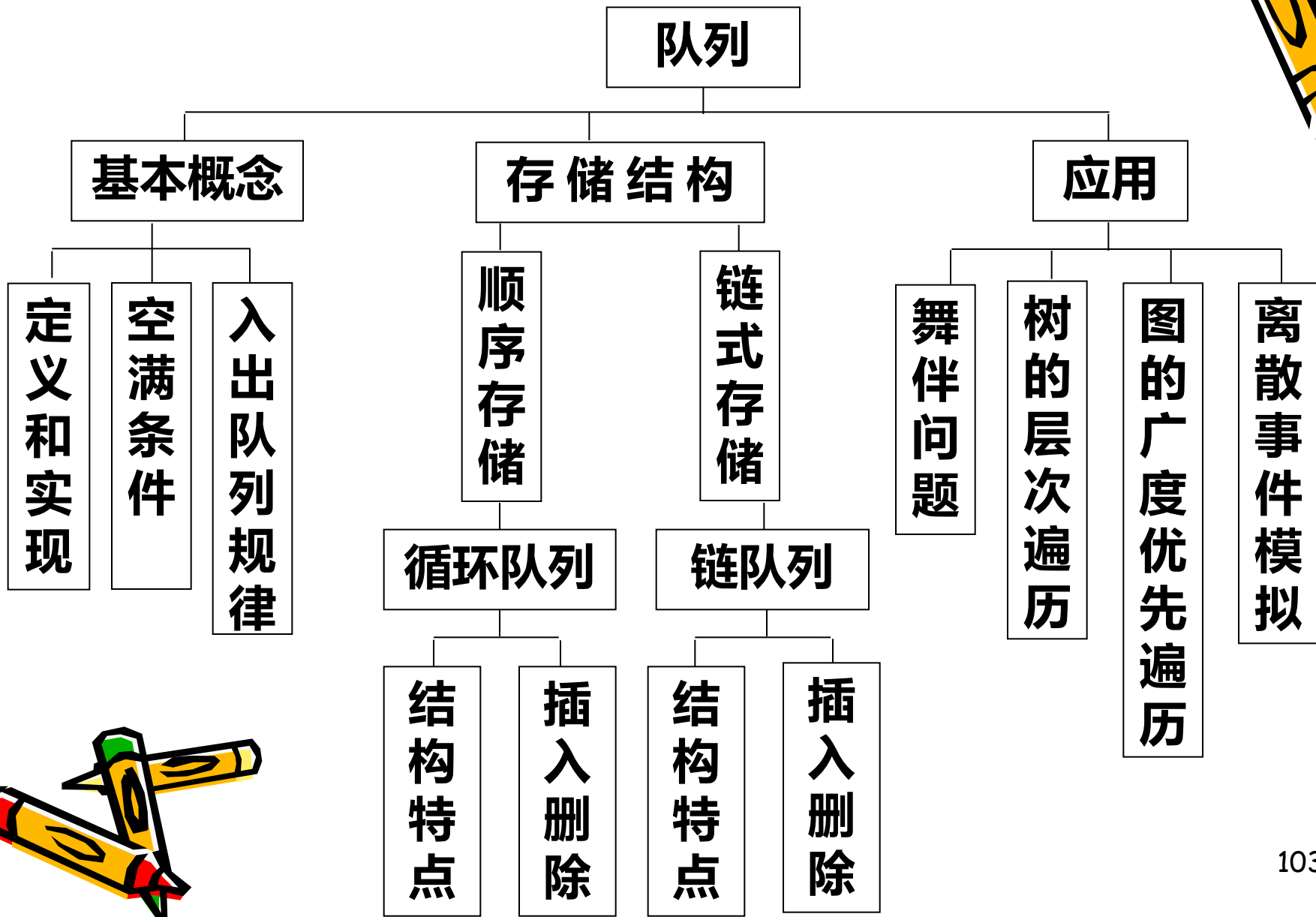
**输入样例：**

```
9
8 4 2 5 3 9 1 6 7
```

**输出样例：**

```
4
```

# 队列的知识结构图







## 本章重点

---

- **掌握栈和队列的术语、特性、抽象数据类型定义**
- **掌握栈和队列的两种存储结构及在各存储结构上基本操作的实现**
- **熟悉栈和队列的应用**
- **在具体问题中灵活运用栈和队列，加深对线性结构的理解**



# 栈的特性 (1)

- 1 一个栈的入栈序列是1, 2, ....., n, 其输出序列为  $p_1, p_2, \dots, p_n$ , 若  $p_1 = n$ , 则  $p_i = \underline{\hspace{1cm}}$ 。
- 2 在一个n个单元的顺序栈中, 设以地址高端为栈底, 用top做栈底指针, 向栈中压入一个元素, 其指针top的变化是\_\_\_\_\_。
- 3 设栈的输入序列是1234, 则\_\_不可能是其出栈序列。  
A 1243    B 2134    C 1432    D 4312    E 3214
- 4 设入栈序列为1234.....n, 输出序列为  $a_1 a_2 \dots a_n$ , 若存在  $1 \leq k \leq n$  使得  $a_k = n$ , 当  $k \leq i \leq n$  时,  $a_i$  为\_\_。  
A  $n-i+1$     B  $n-(i-k)$     C 不确定
- 5 按顺序输入元素f、o、r, 当元素经过栈后到达输出序列, 有哪些输出序列可作为高级语言的变量名。  
A ofr    B for    C rof    D rfo    E fro    F orf



## 栈的特性 (2)

---

6 设输入元素为1、2、3、p、a，输入次序为123pa，当元素经过栈后到达输出序列，有哪些输出序列可作为高级语言的变量名。

A ap321   B pa321   C p3a21   D p32a1   E p321a

7 在高级语言编制的程序中，调用函数和被调用函数之间的链接和信息交换需通过\_\_来进行。

A.栈      B.队列      C.堆      D.广义表



## 栈与递归的实现

**例2：递归的执行情况分析，输出程序运行结果。**

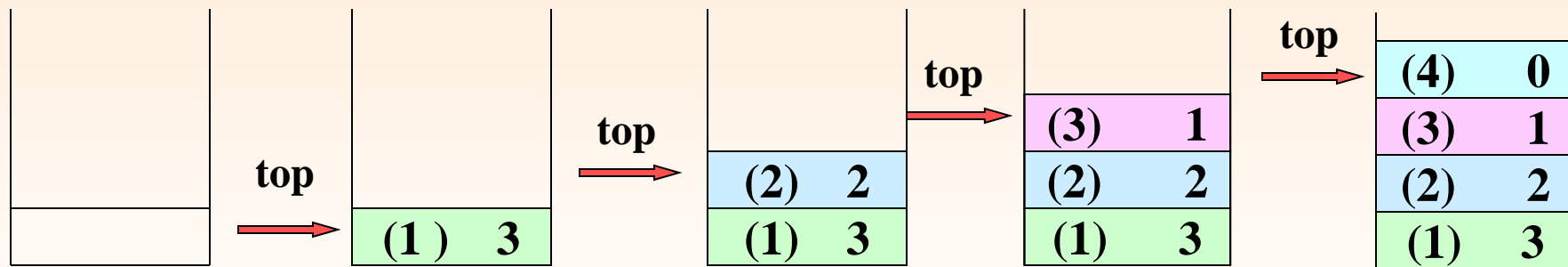
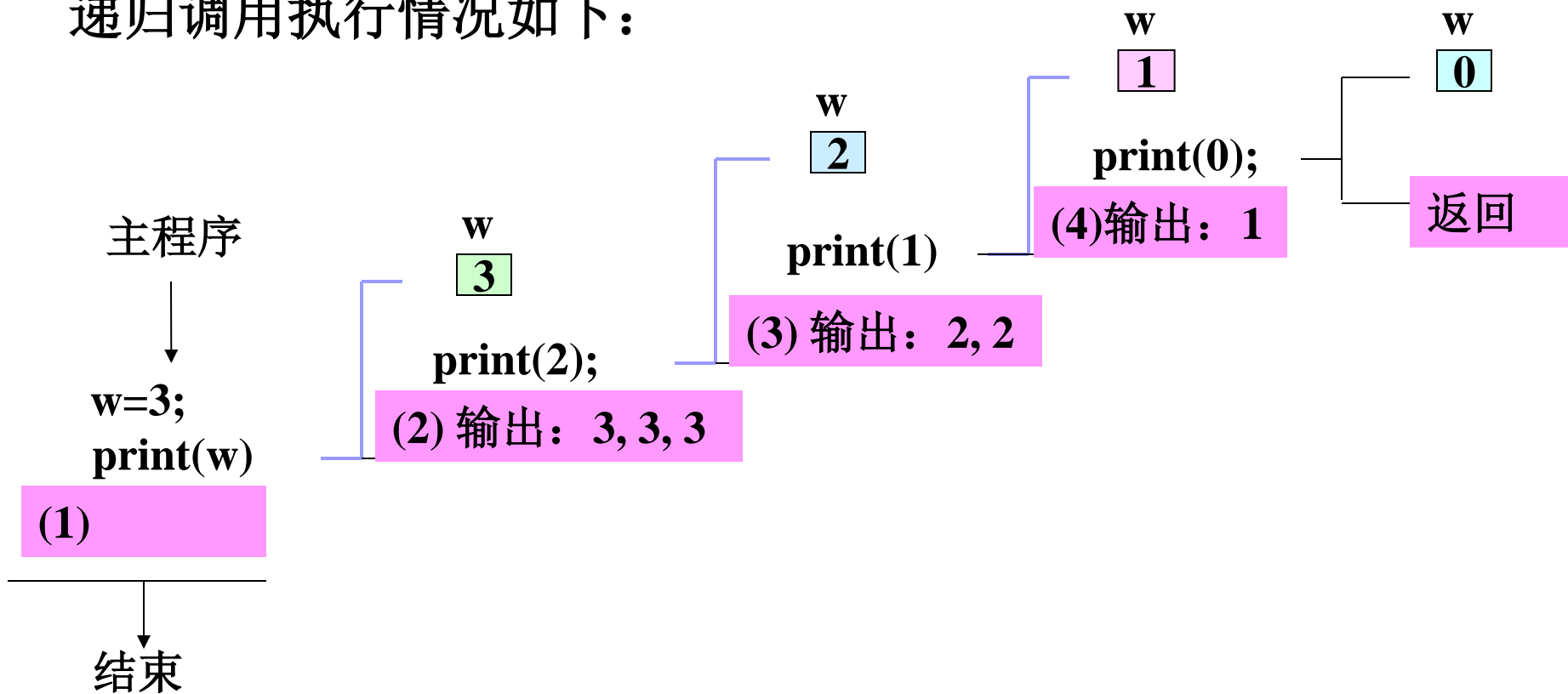
```
void print(int w)
{   int i;
    if (w!=0)
    {   print(w-1);
        for(i=1;i<=w;++i)
            printf("%3d,",w);
        printf("/n");
    }
}

void main()
{   print(3);
}
```

运行结果：

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

递归调用执行情况如下：





# 队列的特性 (1)

- 1 一个大小为**6**的数组实现循环队列，当前**rear=0**，**front=3**，从队列中删除一个元素，再入队两个元素时，**rear=\_\_\_**，**front=\_\_\_**。
- 2 设栈**S**和队列**Q**的初始状态为空，元素**a、b、c、d、e、f**依次通过**S**，一个元素出栈后即进入队列**Q**，若这**6**个元素入队列顺序是**b、d、c、f、e、a**，则**S**的容量至少\_\_\_。
- 3 用循环链表表示的队列长度为**n**，若只设头指针，出队和入队的时间复杂度分别是\_\_和\_\_\_，若只设尾指针，出队和入队的时间复杂度分别是\_\_\_和\_\_\_。
- 4 为解决计算机主机与打印机速度不匹配问题，常设置打印数据缓冲区，主机将要输出的数据依次写入该缓冲区，打印机从该缓冲区取出数据打印，该缓冲区是一个\_\_\_。  
**A 堆 B 队列 C 数组 D 线性表**



## 队列的特性 (2)

- 5 循环队列用数组 $A[0..m-1]$ 存放其元素值, 已知其头尾指针分别是front、rear, 则当前队列中元素个数为\_\_\_\_\_。
- 6 用向量表示的循环队列容量为MAXSIZE, 队首和队尾位置分别为1和max\_size, 队列为空的条件\_\_\_\_, 队列为满的条件\_\_\_\_\_。
- 7 循环队列的下标范围是 $0 \sim n-1$ , 其头, 尾指针分别为f和r, 则元素个数是\_\_\_\_\_。



## 队列的特性 (3)

**8** **Q**是非空队列，**S**是一空栈，设计算法，仅用队列和栈的基本操作和少量工作变量逆置队列**Q**的元素。

```
void Reverse_Queue(SqQueue Q,SqStack s)
{  while(!QueueEmpty(Q))
    {  data=DeQueue(Q);
        push(S,data);
    }//while
    while(!StackEmpty(S))
    {  data=pop(S);
        EnQueue(Q,data);
    }//while
}//Reverse_Queue
```



# 郑州大学研究生入学试题选(1)

- 对循环队列，仅凭 $Q.front=Q.rear$ 无法判断循环队列是空还是满，为了能判别，可使用多种处理方法，试写出其中的两种方法。（设一布尔变量来区分队列满和空、队列中少用一个单元、设置一个计数器）
- 由于栈在使用过程中所需最大空间的大小很难估计，合理的做法是为栈分配尽可能大的容量，因为一旦栈满，就没有办法使用栈了。（对 / 错）
- 将下列递归函数改写为非递归函数：

```
void test(int &sum)
{ int x; scanf(x);
  if(x==0) sum=0;
  else {
    test(sum);
    sum+=x;
  }
  printf(sum);
}
```

```
void test(int &sum)
{ int x; stack s; initstack(&s); scanf(x);
  while(x) { push(s,x); scanf(x);}
  sum=0; printf(sum);
  while (pop(s,x)) {
    sum=sum+x; printf(sum);}
}
```

# 郑州大学研究生入学试题选(2)

- 双端队列是限定插入和删除在表的两端进行的线性表，如果限定双端队列从某个端点插入元素只能从该端点删除，则该双端队列就会变成两个栈底相邻接的栈。

- 将下列函数改写为递归函数：

```
void test ( int n )
```

```
{   int i ;i=n ;
```

```
    while ( i > 1 ) printf( i - - );
```

```
}
```

```
void test(int n){  
    if(n>1) {  
        printf("%d",n);  
        test(n-1);  
    }  
}
```

- 向循环队列 $Q[0..M_0-1]$ 插入一个结点的函数Enqueue和从该队列中取出一个结点的Dequeue函数( $M_0 = 100$ )。
- 假定用一个循环链表表示队列（循环队列），该队列只设一个队尾指针rear，不设队头指针，试编写向循环链表中插入一个元素x的结点算法和从循环链表中删除一个结点的算法。

# 郑州大学研究生入学试题选(3)

- 设数组S[1..m]供一个栈和一个队列使用，栈与队列实际占用的空间事先未知，要求在任何时刻它们存放的数据量都不超过m
- (1) 如何安排栈和队列才能充分利用空间，并写出栈底bottom、栈顶top，队列头front、队列尾rear的初始值。
- (2) 编写插入算法，满足当参数i=1时，将x插入到栈中；当参数i=2时，将x插入到队列中。

解：(1) 初始值：**bottom=top=0; rear=front=m+1;** 满条件：

$$\text{top} + (\text{front} - \text{rear}) = m$$

```
void overflow-false()
{ for (i=front-1; i<=rear; i--)
    s[m+i-front+1]=s[i];
  rear=m-(front-rear)+1; front=m+1;}
void insert(arraytype s, int i, elementype x)
{ if (rear=top+1) overflow-false();
  if (i==1) { top=top+1; s[top]=x; }
  else if (i==2){ rear=rear-1;s[rear]=x; }
}
```



# 郑州大学研究生入学试题选(4)

- 内存中有一连续的存储空间（设1到m单元），现将该空间分配给两个栈S1和S2，为了提高空间的利用率，减少S1和S2栈的上溢的可能性，

- (1) 应如何分配两个栈的空间？
- (2) 用高级语言描述出栈的定义。
- (3) 写出栈满的条件。

解(1)两个栈共享1至m，且两栈底分别设在空间的两端，两栈顶都向中间延伸。

(2) `typedef struct`  
    { `ElemType elem[m];`  
        `int top;`  
    }`s1,s2;`

(3) `s2.top=s1.top+1`

# 郑州大学研究生入学试题选(5)

- 利用两个栈s1、s2模拟一个队列，写出入队和出队算法（可以使用栈的基本操作，并假设栈s1、s2的容量足够大）。

//入队列在**S1**中，在**S2**中出队列

```
EnQueue(stack &s1,stack &s2 ,elemtype k)
{ elemtype e;
  while(!stackempty(s2)) {pop(s2,e); push(s1,e);}
  push(s1,k);
}
DeQueue(stack &s1,stack &s2 )
{ elemtype e;
  while(!stackempty(s1)) {pop(s1,e); push(s2,e);}
  pop(s2,e);
  return(e);
}
```

# 郑州大学研究生入学试题选(6)

- 假设以带头结点的循环单链表表示队列，并只设一个指针rear指向队尾结点，但不设头指针，请写出相应的入队算法和出队算法，要求算法中含有单链表结点结构的描述。

```
typedef struct slink {
    elemtype data;
    slink *next;
}slink;
enqueue(slink &rear, elemtype x)
{ slink *s;
  s=(slink*)malloc(sizeof(slink));
  if (!s)
  { printf("overflow"); exit(0);}
  s->data=x;
  s->next=rear->next ;
  rear->next=s; rear=s;
}
```

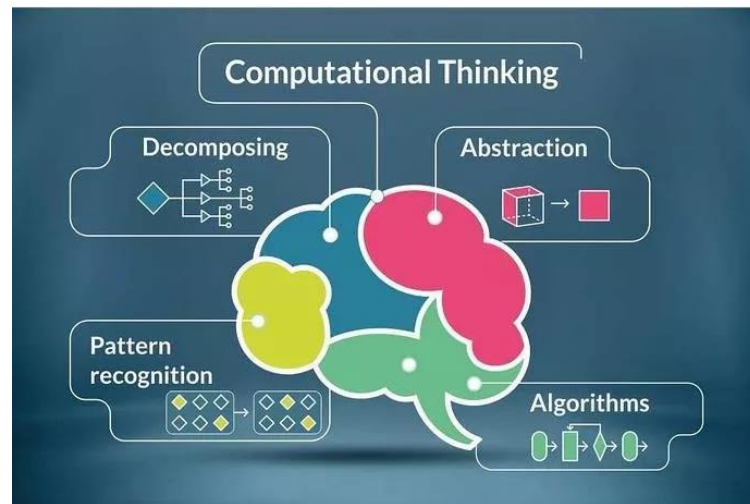
```
dequeue(slink &rear, elemtype &x)
{ slink *p;
  if(rear->next==rear)return 0;
  //*****
  p=rear->next->next;
  if(p==rear)
  { rear=rear->next;
    rear->next=rear;} // ***
  else
    rear->next->next=p->next;
  x=p->data;
  free(p);
  return 1;
}
```

# 郑州大学研究生入学试题选(7)

- (2006)假定用一个循环链表表示队列（称为循环队列），该队列只设一个队尾指针rear，不设队头指针，试编写向循环链表中插入一个元素x的结点算法和从循环链表中删除一个结点的算法。

# 能力培养

- 逻辑思维能力：运用科学的逻辑方法，对事物进行观察、比较、分析、综合、抽象、概括、判断、推理的能力，还包括准确、有条理地表达自己思维过程的能力。



- 计算思维能力：周以真教授于**2006**年首次提出，是运用计算机科学的基本概念进行问题求解、系统设计及人类行为理解等涵盖计算机科学之广度的一系列思维活动，是与形式化问题及其解决方案相关的思维过程。



在c语言中内存可大致分为以下五种：

	内容	管理和权限	容量
栈区	局部变量，函数参数	编译器分配，空间不够报栈溢出“ <b>stack overflow</b> ”	有限
堆区	动态申请的变量	程序员手动申请和释放，释放不及时会导致内存泄露	几乎无空间限制
静态变量区	全局变量和（ <b>static</b> ）（全局、局部）静态变量	可读可写	
常量区	字符串常量、 <b>const</b> 修饰的全局变量	只读	
代码区	程序二进制代码	操作系统管理，只读	