第3章 限定性线性表—栈和队列

3.1 栈

3.2 队列

引言

- 本章我们将学习两种重要而特殊的线性数据结构
  - 从逻辑结构角度看：它们是线性表
  - 从运算角度看：它们的基本运算是线性表运算的子集，是操作受限的线性表

本章内容：介绍栈和队列的基本概念、存储结构和基本运算的实现

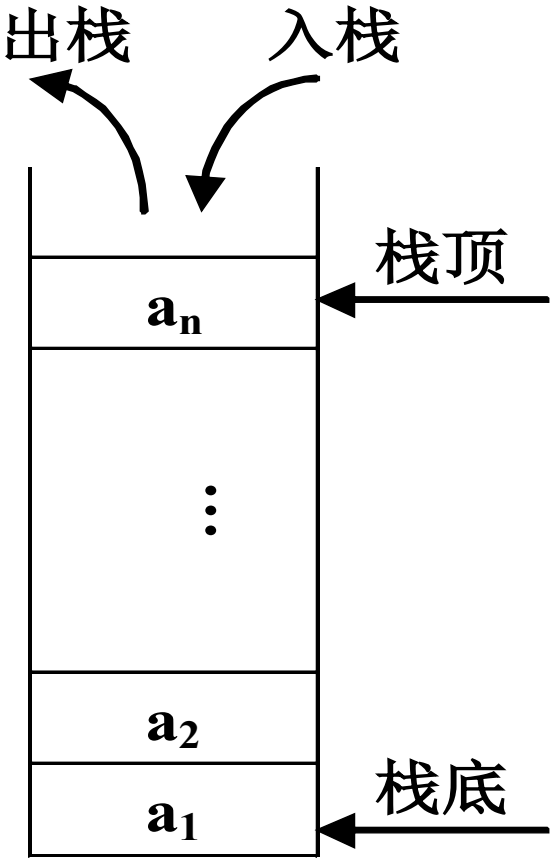
3.1 栈

1. 栈的定义

限定仅在表尾进行插入或删除操作的线性表。

$S = (a_1, a_2, ..., a_n)$

- 栈底(Bottom)：表头端称为栈底。
- 栈顶(Top)：表尾端称为栈顶。
- 进栈(入栈)：栈的插入操作。
- 出栈(退栈)：删除操作。
- 空栈：不含元素的空表。
- 栈底元素：a_1；栈顶元素：a_n



2. 栈的特点

“很窄的死胡同”

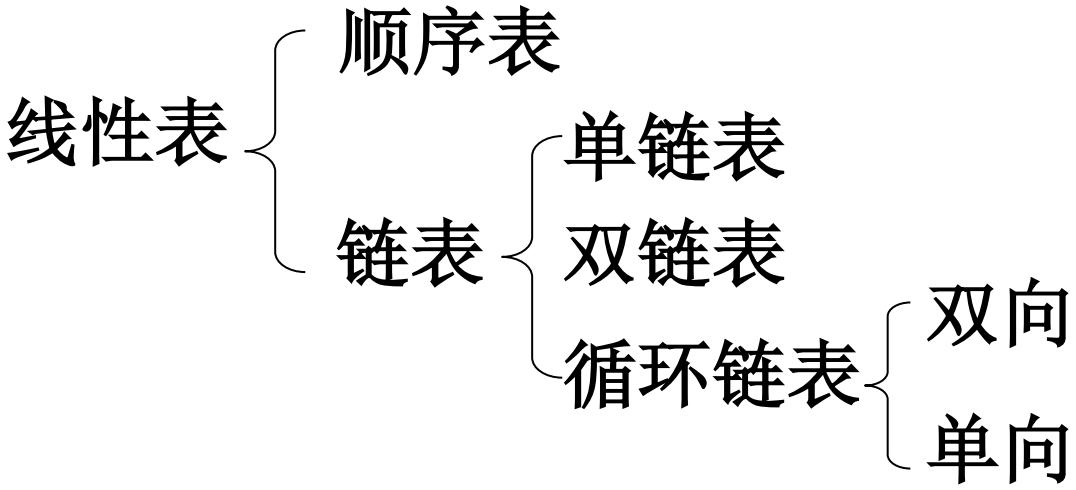
后进先出(Last In First Out)，简称LIFO结构。

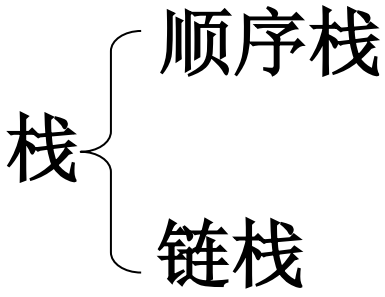
栈又称后进先出线性表。

3. 栈的基本运算

- 初始化 InitStack(&S)。构造一个空栈
- 入栈 Push(&S, e)。栈S已经存在，插入e为新的栈顶元素
- 出栈 Pop(&S, &e)。栈S存在且非空，删除栈顶元素，e返回
- 读栈顶元素 GetTop(S, &e)。栈S存在且非空，用e返回栈顶元素
- 判栈空 StackEmpty(S)。栈S存在，若为空栈，返回真，否则假

4.栈的表示和实现





●顺序栈

顺序栈的类型定义

```
typedef struct{
    SElemType  *base;  //栈底指针
    SElemType  *top;    //栈顶指针
    int stacksize;  //栈已分配的空间大小
}SqStack; //动态分配
```

```
typedef struct{
{
    SElemType  data[MAXSIZE];
    int              top;
} SqStack; //静态分配
```

SqStack S;

①栈结构不存在

S.base = NULL;

②空栈

S.base = S.top;

③栈满

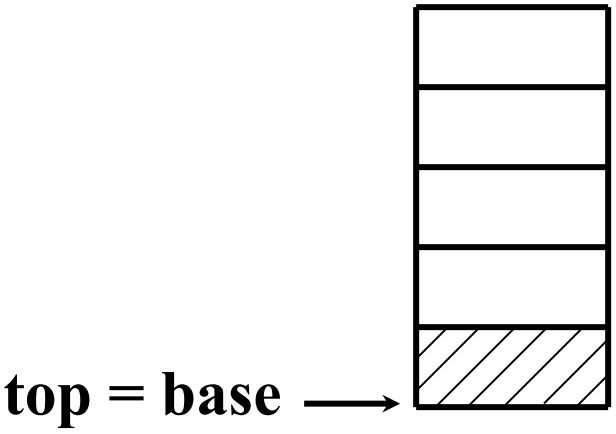
S.top – S.base >= S.stacksize;

①空栈

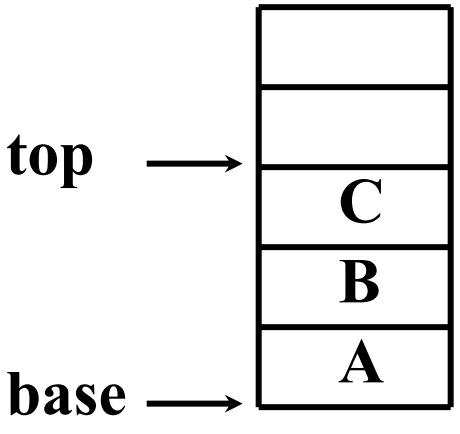
top = 0;

②栈满

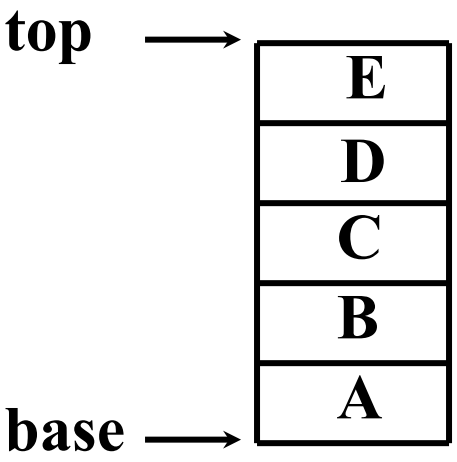
top = MAXSIZE;



空栈



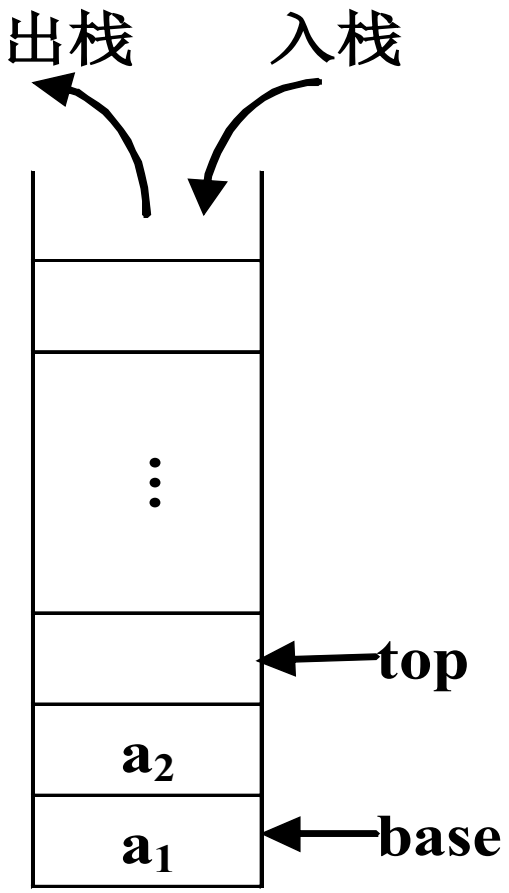
栈中有三个元素



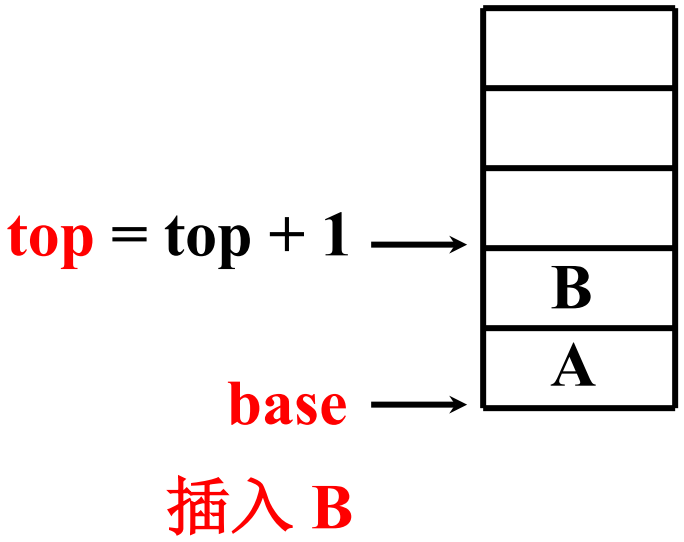
满栈

- 顺序栈中的进栈和出栈

栈仅在表的一端进行操作；
top指针始终指向栈顶元素的下一位置。

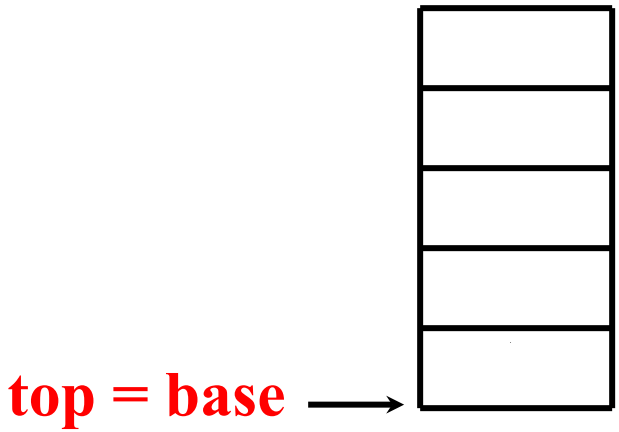


例，在栈中插入元素 A 和 B



插入 B

例，删除栈顶元素 B 和 A



删除 C

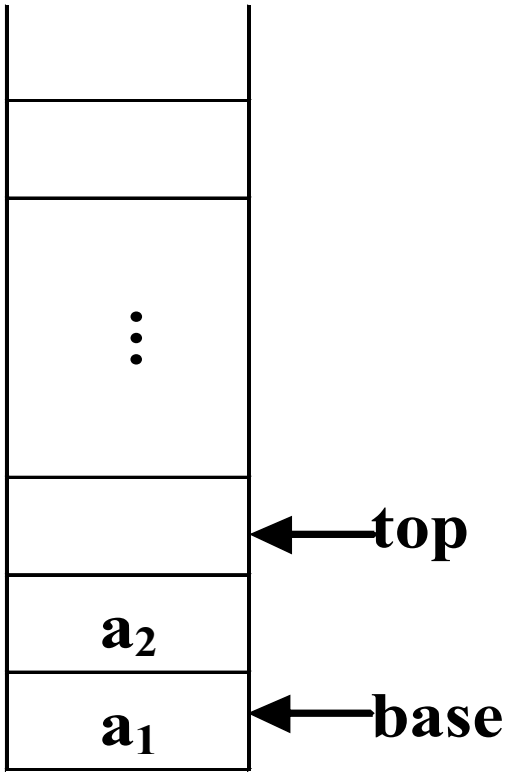
ERROR

- 初始化 InitStack(SqStack &S)

```
{
    S.base = S.top = (SElemType *)malloc(…);
    if(!S.base) exit(OVERFLOW);
    S.stacksize = STACK_INIT_SIZE;
   return OK;
}
```

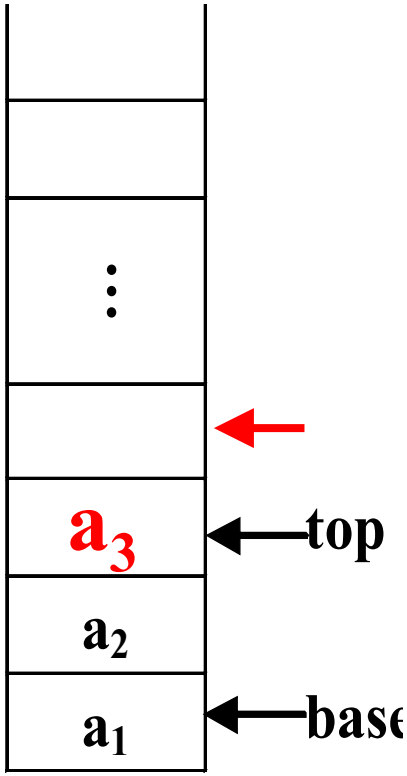
- 读栈顶元素 GetTop(SqStack S, SElemType&e)

```
{
    if(S.top == S.base)
        return ERROR;
    e = *(S.top – 1);
    return OK;
}
```



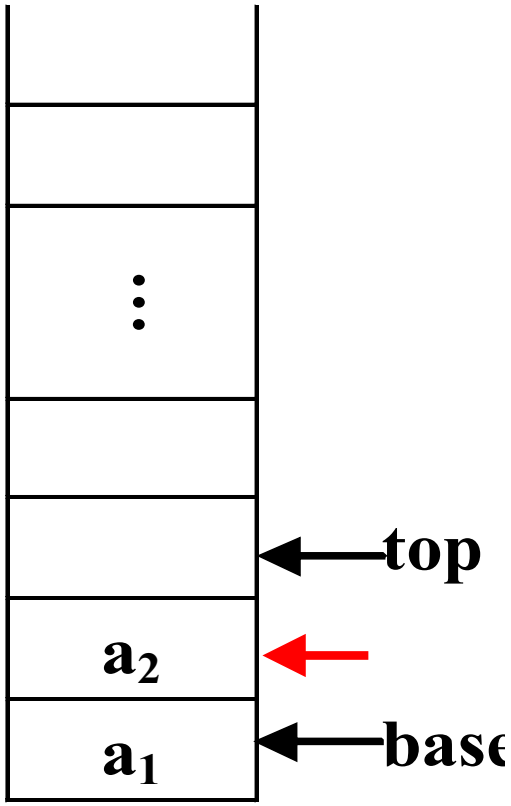
- 入栈 Push(SqStack &S, SElemType e)

```
{
   if(S.top–S.base >= S.stacksize){
      S.base = (SElemType *)realloc(…);
      if(!S.base)  exit(OVERFLOW);
      S.top = S.base + S.stacksize;
      S.stacksize += STACKINCREMENT;
   }
   *S.top++ = e;  //*S.top = e;  S.top++;
   return OK;
}
```



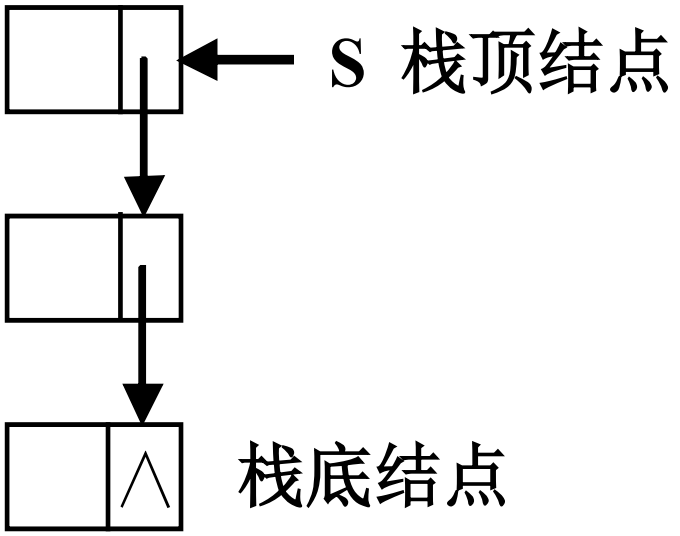
- 出栈 Pop(SqStack &S, SElemType &e)

```
{
   if(S.top == S.base)
        return ERROR;
  e = *--S.top;  //--S.top ;  e = *S.top;
  return OK;
}
```



- 链栈

```
typedef struct STNode{
    SElemType      data;
    struct STNode *next;
}STNode, *LinkStack;
```



- 栈顶结点
- 栈底结点
- 栈顶指针：链栈由栈顶指针S唯一确定
- 链栈本身无容量限制，在用户内存空间的范围内不会出现栈满情况

- 初始化 InitStack(LinkStack &S)

```
{
    S = NULL;
    return OK;
}
```

- 读栈顶元素 GetTop(LinkStack S, SElemType&e)

```
{
    if(S == NULL)
        return ERROR;
    e = S->data;
    return OK;
}
```

- 入栈 Push(LinkStack &S, SElemType e)

```
{
   p = (LinkStack) malloc( sizeof( STNode));
   if(!p)
     exit(OVERFLOW);
   p->data = e;
①p->next = S;
②S = p;
   return OK;
}
```

- 出栈 Pop(LinkStack &S, SElemType &e)

```
{
    if(S == NULL)
        return ERROR;
    e = S->data;
①p = S;
②S = S->next;
    free(p);
    return OK;
}
```

5.栈总结

- 栈是一种具有线性结构的数据结构，是操作受限的线性表；
- 栈的特点是后进先出，只能在栈顶进行插入和删除操作，分别称为入栈和出栈；
- 顺序栈中，栈空标志：S.top = S.base;

  栈满标志：S.top – S.base >= S.stacksize;
- 链栈中，不设头结点，头指针就是栈顶指针，栈空S = NULL。

练习：

1．设一个栈的输入序列为A,B,C,D,则借助一个栈所得到的输出序列不可能是___。

(A) A,B,C,D　　(B) D,C,B,A

(C) A,C,D,B　　(D) D,A,B,C

答:可以简单地推算,得容易得出D,A,B,C是不可能的,因为D先出来,说明A,B,C,D均在栈中,按照入栈顺序,在栈中顺序应为D,C,B,A,出栈的顺序只能是D,C,B,A。所以本题答案为D。

练习：

2.已知栈的输入序列为1,2,3,…,n。输出序列为 $a_1, a_2, \ldots, a_n$，若$a_1=n$，问$a_i=?$

$a_i=n-i+1$

3. I表示入栈，O表示出栈，若元素入栈顺序为1234，为了得到1342的出栈顺序，相应的I和O的操作串是什么？

$I_1O_1I_2I_3O_3I_4O_4O_2$

4. 设n个元素进栈序列是1,2,3,…,n,其输出序列是$p_1,p_2,\cdots,p_n$,若$p_1=3$,则p_2的值__。

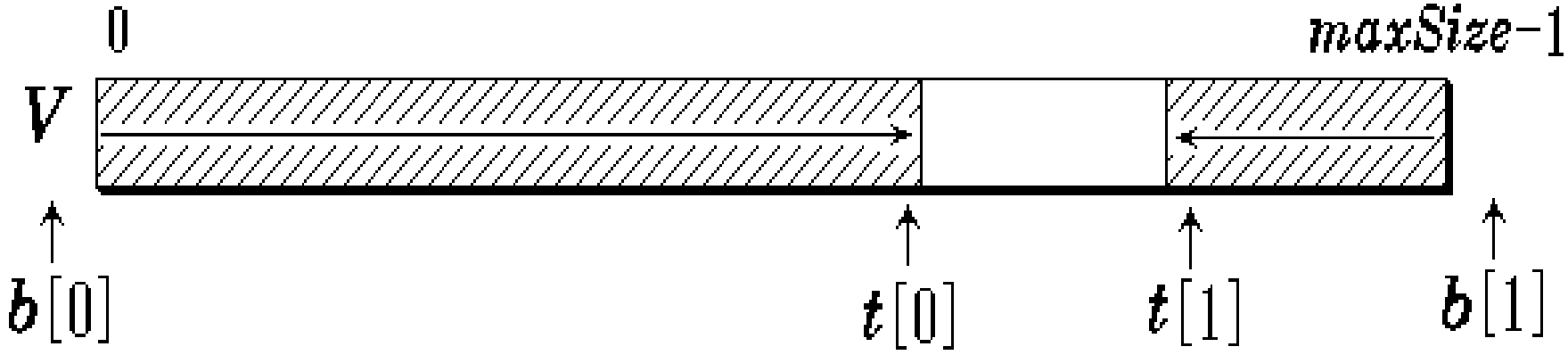
(A) 一定是2　　(B) 一定是1

(C) 不可能是1　　(D) 以上都不对

答:当$p_1=3$时,说明1,2,3先进栈,立即出栈3,然后可能出栈,即为2,也可能4或后面的元素进栈,再出栈。因此,p_2可能是2,也可能是4,…,n,但一定不能是1。所以本题答案为C。

- 双栈共享一个栈空间

```
typedef struct{
        SElemtype *base[2];
        SElemtype *top[2];
        int              stacksize;
}DSqStack; //双向栈类型
```



- 作业：

  3.3

  3.4

3.2 栈的应用举例

1.数制转换

N=(N div d)*d+N mod d

$(1348)_{10} = (2504)_8$

8	1348	……4
8	168	……0
8	21	……5
8	2	……2
	0	

```
void conversion()
{
    InitStack(S);
    scanf("%d", &N);
    while(N){
        Push(S, N%8);
        N = N/8;}
    while(!StackEmpty(S)){
        Pop(S, e);
        printf("%d", e);}
}
```

2.括号匹配的检验

假设在表达式中

（ [] （）） 或 [（ [ ] [ ] ）]

为正确的格式，

[（] ） 或 （ [（ ） ） 或 (( )])

均为不正确的格式。

分析可能出现的不匹配的情况:

- 到来的右括弧非是所“期待”的;
- 到来的是“不速之客”;
- 直到结束，也没有到来所“期待”的括弧;

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**，

若**栈空**，则表明表达式中**匹配正确**

否则表明“**左括弧**”**有余**

2. 括号匹配的检验

```
Status match(char *str)
{   InitStack(S);
    for(p=str; *p; p++){
        switch(*p){
            case '(':
            case '[':
            case '{':  Push(S, *p); break;
            case ')':  if (StackEmpty(S)) return ERROR;
                       Pop(S, e);  if(e != '(')  return ERROR; break;
            case ']':  if (StackEmpty(S)) return ERROR;
                       Pop(S, e);  if(e != '[')  return ERROR; break;
            case '}':  if (StackEmpty(S)) return ERROR;
                       Pop(S, e);  if(e != '{')  return ERROR;
        }//switch
    } //for
    if(StackEmpty(S))  return OK;
    else return ERROR;
}
```

3. 行编辑程序

在用户输入一行的过程中，允许 用户输入出差错，并在发现有误时可以及时更正。

设立一个输入缓冲区，用以接受用户输入的一行字符，然后逐行存入用户数据区; 并假设“#”为退格符，“@”为退行符。

假设从终端接受两行字符：

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

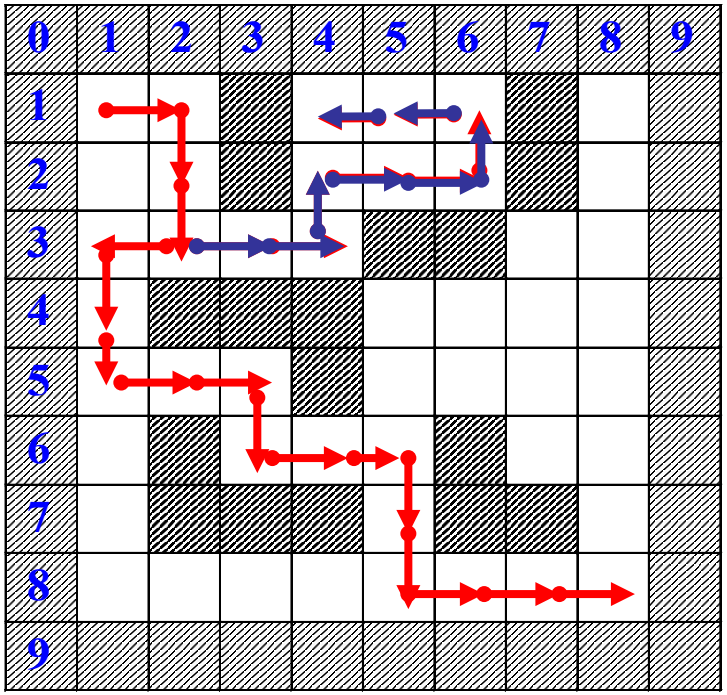
实际有效行为：

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

```
Void LineEdit()
{
        InitStack(s);
        ch=getchar();
        while (ch != EOF) { //EOF为全文结束符
                while (ch != EOF && ch != '\n') {
                    switch (ch) {
                            case '#' : Pop(S, c); break;
                            case '@': ClearStack(S); break;
                                        // 重置S为空栈
                             default : Push(S, ch);  break;
                             }
                     ch = getchar();  // 从终端接收下一个字符
                 }
                将从栈底到栈顶的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 栈的应用举例

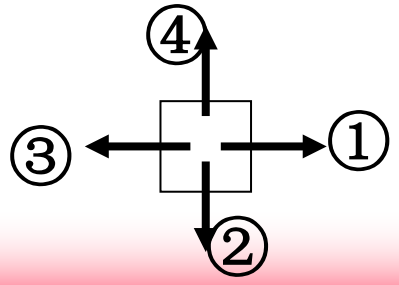
4. 迷宫求解 （穷举法）



●当前位置：某一时刻所在图中某个方块位置。

●当前位置可通：未曾走到过的通道块，既要求该方块位置不在当前路径上，也不是曾经纳入过路径的通道块。

①回路不能出现，简单路径；
②防止在死胡同内转圈。



●下一位置：当前位置四周四个方向上相邻的方块。

为了表示迷宫,设置一个数组mg,其中每个元素表示一个方块的状态,为0时表示对应方块是通道,为1时表示对应方块为墙,如图3.3所示的迷宫,对应的迷宫数组mg如下:

```
int mg[M+1][N+1]={  /*M=10,N=10*/
    {1,1,1,1,1,1,1,1,1,1},
    {1,0,0,1,0,0,0,1,0,1},
    {1,0,0,1,0,0,0,1,0,1},
    {1,0,0,0,0,1,1,0,0,1},
    {1,0,1,1,1,0,0,0,0,1},
    {1,0,0,0,1,0,0,0,0,1},
    {1,0,1,0,0,0,1,0,0,1},
    {1,0,1,1,1,0,1,1,0,1},
    {1,1,0,0,0,0,0,0,0,1},
    {1,1,1,1,1,1,1,1,1,1}};
```

- 下一位置：int movei[4] ={0, 1, 0, -1};
  int movej[4] ={1, 0, -1, 0};

  设当前位置为e(i, j)，则e(i, j)的下一位置：

  i = e.i + movei[e.dir-1];

  j = e.j + movej[e.dir-1];

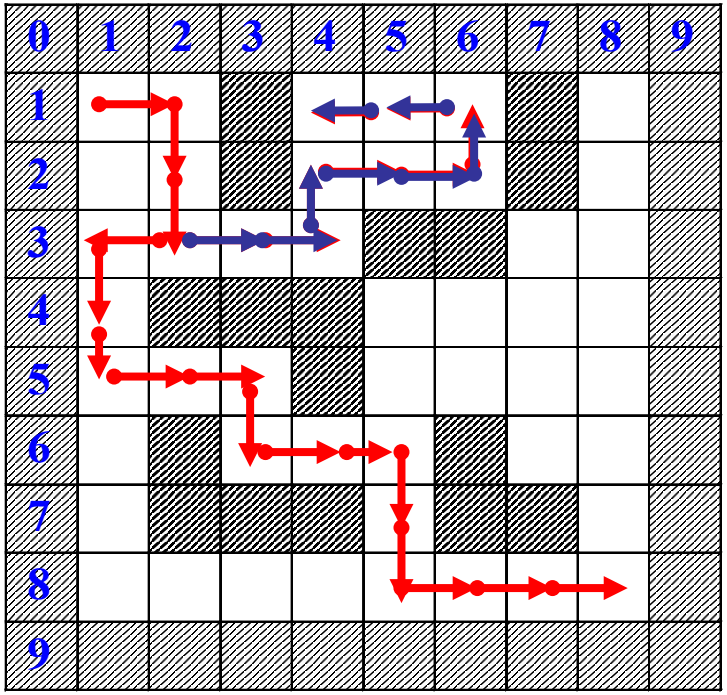
- 当前通道块是否可通：

  if(maze[i][j] ==0)　可通；

- 从入口到出口的路径算法

设当前位置的初值为入口位置；

```
do{
    若当前位置可通，则{
        将当前位置插入栈顶；
        若该位置为出口位置，则结束；
        将该位置的东邻块置为当前位置；
    }
    否则{
        若栈不空且栈顶位置尚有其它方向未经探索，则
            设定新的当前位置为栈顶位置的下一相邻块；
        若栈不空但栈顶位置的四周均不通，则
            删去栈顶位置；
            若栈不空，则
                重新测试新的栈顶位置，直至找到一个可通的
                相邻块或至栈空；
    }
}while(栈不空)
```




```
typedef struct{
    int ord;  //在路径上的序号
    int i, j;   //坐标
    int dir; //当前位置下一方向
}SElemType;
```

ord	i	j	dir
1	1	1	1
2	1	2	2
3	2	2	2
4	3	2	3
5	3	3	1
6	3	4	4
7	2	4	1
8	2	5	1
9	2	6	4
10	1	6	3
11	1	5	3
12	1	4	X
13	3	1	2
14	4	1	2

●表达式求值（算符优先算法）

这里限定的表达式求值问题是:用户输入一个包含“+”、“-”、“*”、“/”、正整数和圆括号的合法数学表达式,计算该表达式的运算结果。

操作数（operand）、运算符（operator）、界限符(delimiter 左右括号和表达式结束符），运算符和界限符统称算符。

#（7+15）*（23-28/4）#

(1) 从左算到右;

(2) 先乘除， 后加减;

(3) 先括号内， 后括号外。

算术表达式的求值

算术表达式中运算符(+,-,*,/等)的优先规则

- 设置两个工作栈：**OPND栈**和**OPTR栈**

运算符栈OPTR和操作数栈OPND 。OPND也放表达式的运算结果。

算法思想：

1 首先置操作数栈OPND为空栈，置运算符栈OPTR的栈底为表达式的起始符#(优先级最低)。

2 依次读入表达式中的每个字符ch，直至表达式结束：

若ch是操作数,则进OPND栈；

若ch是运算符，若其优先级不高于栈顶运算符的优先级时，则取出栈OPND的栈顶和次栈顶的两个元素以及栈OPTR的栈顶运算符，进行相应的运算，并将结果放入栈OPND中；如此下去，直至ch的优先级高于栈顶运算符的优先级，将ch入OPTR栈。

θ_1 \ θ_2	+	—	*	/	(	)	#
+	>	>	<	<	<	>	>
—	>	>	<	<	<	>	>
*	>	>	>	>	<	>	>
/	>	>	>	>	<	>	>
(	<	<	<	<	<	=	
)	>	>	>	>		>	>
#	<	<	<	<	<		=

- θ1<θ2，表明不能进行θ1 的运算，θ2 入栈，读下一字符。
- θ1>θ2，表明可以进行θ1 的运算，θ1退栈，运算，运算结果入栈。
- θ1=θ2，脱括号，读下一字符或表达式结束。

- #3*（7-2）#

步骤	OPTR栈	OPND栈	输入字符	主要操作
1	#		3*(7-2)#	PUSH(OPND,’3’)
2	#	3	*(7-2)#	PUSH(OPTR,’*’)
3	#*	3	(7-2)#	PUSH(OPTR,’(‘)
4	#*(	3	7-2)#	PUSH(OPND,’7’)
5	#*(	37	-2)#	PUSH(OPTR,’-’)
6	#*(-	37	2)#	PUSH(OPND,’2’)
7	#*(-	372	)#	operate(‘7’,’-’,’2’)
8	#*(	35	)#	POP(OPTR)
9	#*	35	#	operater(‘3’,’*’,’5’)
10	#	15	#	RETURN

表达式求值算法

```
ElemType EvalExpression()
{char ch,opnd[maxch],optr[maxch];//opnd是运算数栈,optr是运算符栈
 init(optr); push(optr, '#');
 init(opnd); ch=getchar();
 while(ch!='#' || gettop(optr)!='#')
    if(!in(ch,op)) {push(opnd,ch); ch=getchar();}  //不是运算符就进栈
    else switch(precede(gettop(optr),ch))
          {case '<':   //栈顶元素优先级低
                       push(optr,ch); ch=getchar(); break;
           case '=':   //脱括号并接收下一字符
                       pop(optr,x); ch=getchar(); break;
           case '>':   //退栈并将运算结果入栈
                       pop(optr,theta);
                       pop(opnd,b);  pop(opnd,a);
                       push(opnd,operate(a,theta,b)); break;
           }//switch
   return  gettop(opnd)
}
```

3.3 栈与递归的实现

1. 递归的概念：

一个过程（或子程序，函数）在完成之前又直接或间接地调用自己，则称之为递归过程（或子程序，函数）。

若直接调用自己称为直接递归，若通过其它函数并由后者反过来调用前者，称为间接递归。

2. 递归不是一种数据结构，而是一种算法设计方法。

3. 以下三种情况，常用到递归方法

- 数学函数是递归定义的

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

求阶乘的递归算法

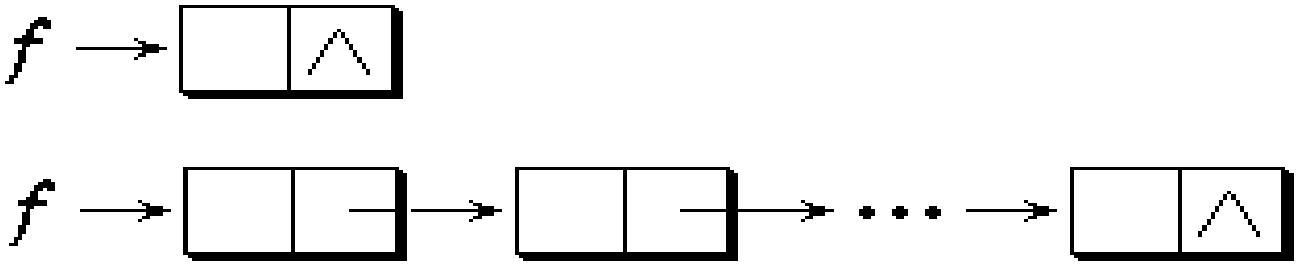
```
long Factorial ( long n ) {
    if ( n == 0 )
        return 1;
    else
        return n*Factorial (n-1);
}
```

- 求解阶乘 *n*! 的过程


参数 计算 返回
0 0! = 1 1
参数 计算 返回
1 1*Factorial(0) 1
参数 计算 返回
2 2*Factorial(1) 2
参数 计算 返回
3 3*Factorial(2) 6
参数 计算 返回
4 4*Factorial(3) 24
主程序 main
0
1
2
3
4
参数传递
1
1
2
6
24
返回值

- 数据结构是递归的

  例：单链表结构

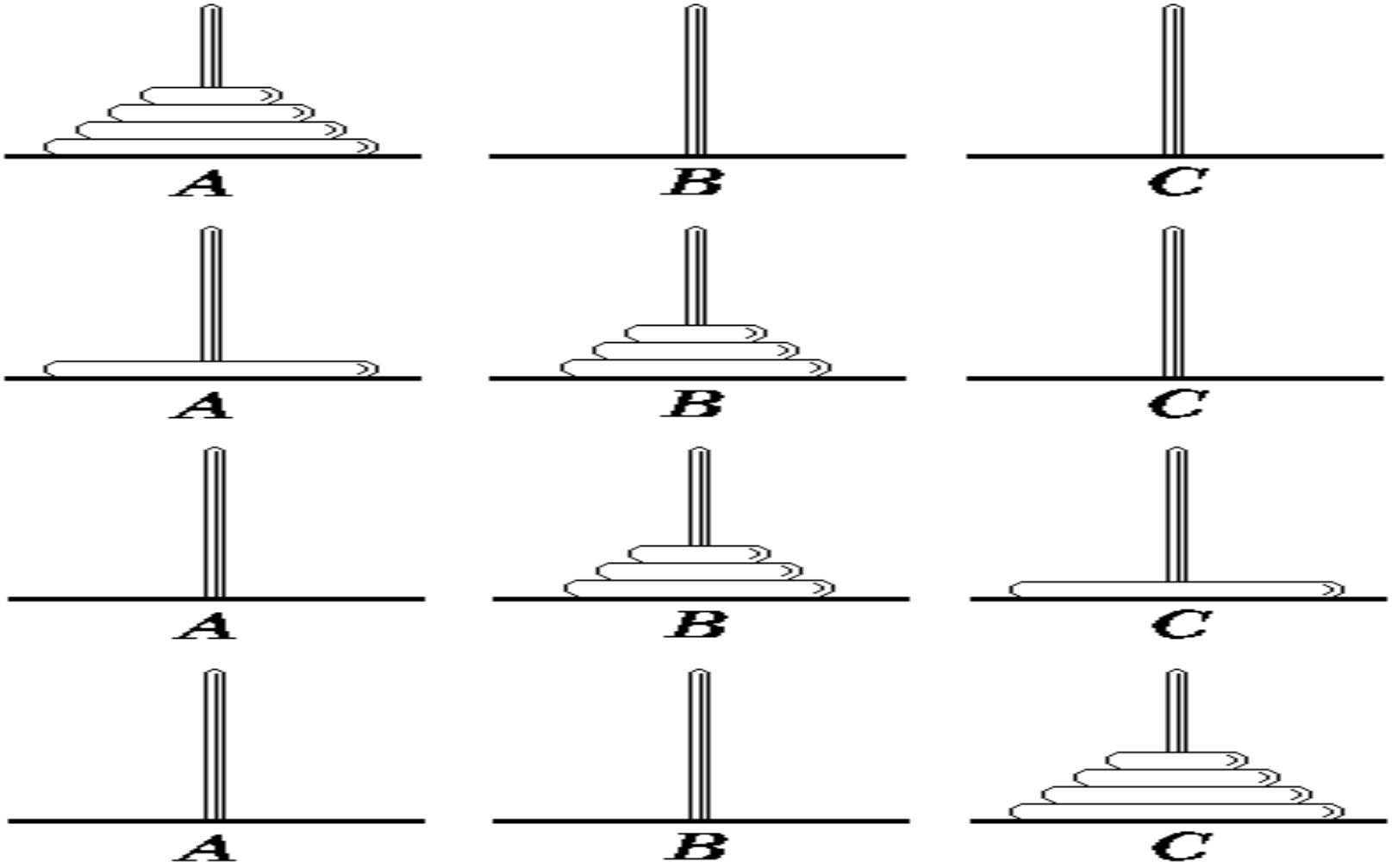


搜索链表最后一个结点并打印其数值

```
void Find ( LNode  *L ) {
    if ( L ->next == NULL )
        cout << L->data << endl;
    else
        Find ( L ->next );
}
```

- 问题的解法是递归的

例如，汉诺塔(Tower of Hanoi)问题



解决汉诺塔问题的算法

```
void Hanoi (int  n, char  A, char  B, char  C )
1{
2    if ( n == 1 )
3         cout << " Move disk"<< n<<"from"<<A
                << "to " << C << endl;
4    else {
5         Hanoi ( n-1, A, C, B );
6         cout << "Move disk"<< n<<"from"<<A
                <<"to"<< C<< endl;
7         Hanoi ( n-1, B, A, C );
8    }
9}
```

4. 递归设计

- 对原问题*f(s)*进行分解，给出合理的较小问题*f(s')*；
- 假设*f(s')*是可解的，在此基础上确定*f(s)*的解，给出*f(s)*与*f(s')*之间的关系；
- 确定一个特定情况的解，由此作为递归出口。

含一个递归调用的递归过程的一般形式

```
void  p(参数)
{
    if(数据为递归出口)    操作;
    else{   操作;  p(参数);  操作; }
}
```

5.递归调用的内部实现原理

（1）一般函数调用

运行被调用函数之前

- 实在参数，返回地址传给被调用函数保存；
- 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入口，转被调用函数执行。

被调用函数返回之前

- 保存被调用函数的计算结果；
- 释放被调用函数的数据区；
- 按返回地址将控制权转移到调用函数。

（2）递归调用　调用自身代码的复制件

函数之间的信息传递和控制转移通过“栈”来实现。

递归工作栈

- 每一次递归调用时，需要为过程中使用的参数、局部变量等另外分配存储空间。
- 每层递归调用需分配的空间形成递归工作记录，按后进先出的栈组织。

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**”

例如：

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

Main的数据区

递归函数执行的过程可视为**同一函数**进行嵌套调用

递归工作栈：递归过程执行过程中占用的数据区。

递归工作记录：每一层的递归参数合成一个记录。

当前活动记录：栈顶记录指示当前层的执行情况。

当前环境指针：递归工作栈的栈顶指针。

- 递归算法具有两个特性：

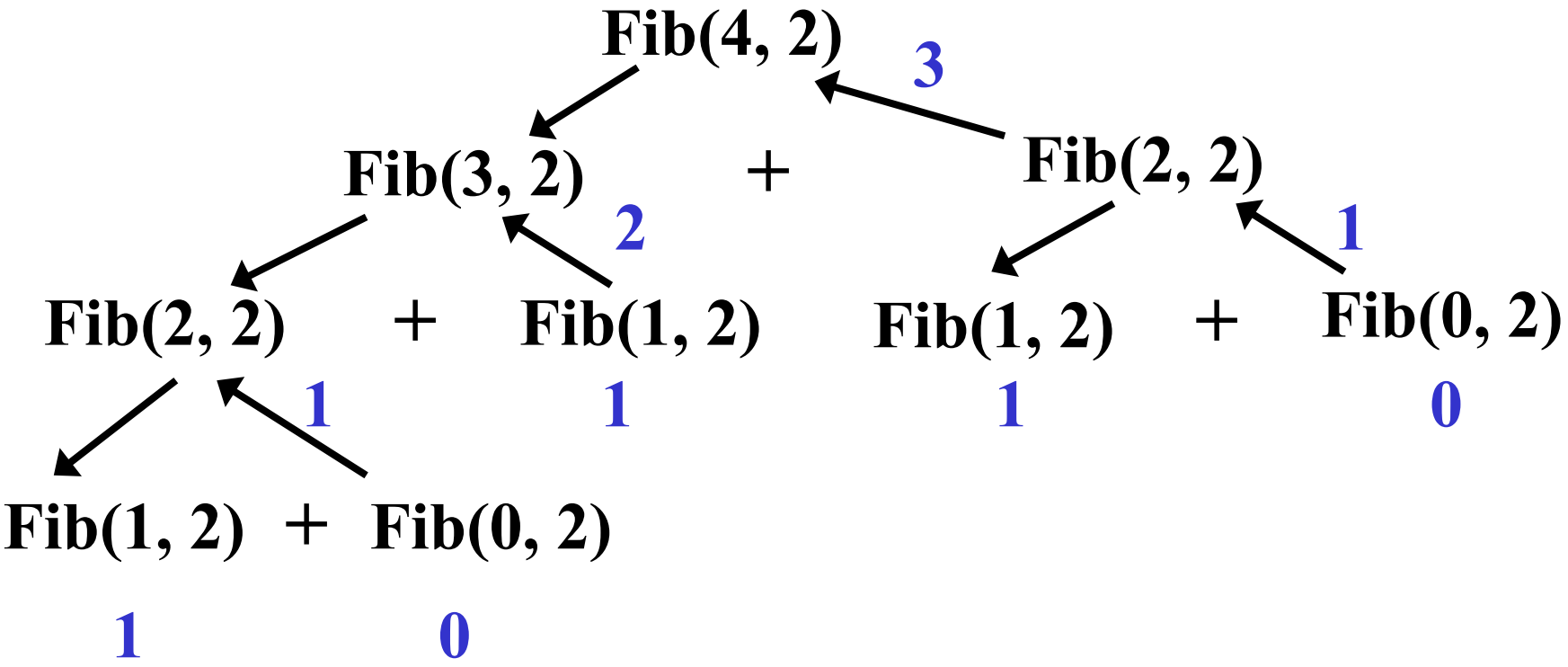
  (1) 递归算法是一种分而治之、把复杂问题分解为简单问题的求解问题方法，对求解某些复杂问题，递归算法的分析方法是有效的。

  （2） 递归算法的时间效率低。递归执行时需要系统提供隐式栈实现递归，效率低且费时。

- 练习：递归算法计算K阶斐波那契(Fibonacci)数列的第m项值。

```
int  Fib ( int m, int k)
{    if( k <  2 || m< 0)  return -1;
     if( m < k-1)  return 0;
     else if( m == k-1 ) return 1;
     else{
          sum = 0;
          for( i = 1; i <= k; i++)
                sum += Fib( m - i, k);
          return sum;
      }
}
```

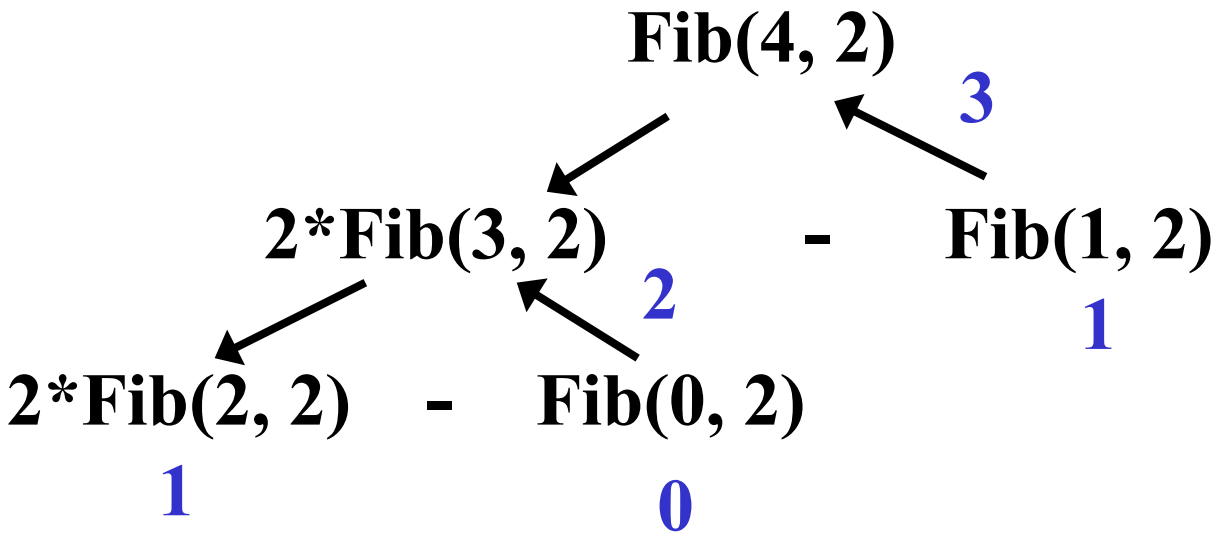
m=4, *k*=2


Fib(4, 2)
3
Fib(3, 2)
+
Fib(2, 2)
2
1
Fib(2, 2)
+
Fib(1, 2)
Fib(1, 2)
+
Fib(0, 2)
1
1
1
0
Fib(1, 2)
+
Fib(0, 2)
1
0

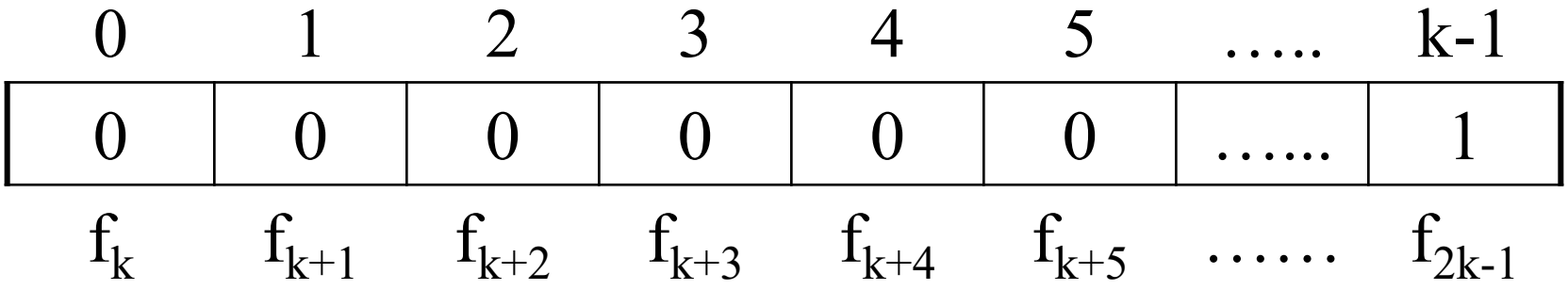
$$f_m = 2 * f_{m-1} - f_{m-(k+1)}$$

```
int  Fib ( int m, int k)
{    if( k <  2 || m< 0)  return -1;
     if( m < k-1)  return 0;
     else if( m == k-1 || m == k ) return 1;
     else
          return  2*Fib( m-1, k) – Fib( m-k-1, k);
}
```

m=4, *k*=2



- 非递归算法计算K阶斐波那契(Fibonacci)数列的第m项值。



```
int Fib(int k, int m)
{
    if(k<2 || m<0)   return -1;
    if(m<k-1) return 0;
    else if(m == k-1) return 1;
    else{
        temp = new int[k];
        for(i=0; i<=k-2; i++)  temp[i] = 0;
        temp[k-1] = 1;
        for(i=k; i<=m; i++){
            sum = 0;
            for(j=0; j<k; j++)   sum += temp[j];
            temp[i%k] = sum;
        }
        r=temp[m%k];
        delete [] temp;
      return r;
    }
}
```

```
int Fib(int k, int m)
{
    if(k<2 || m<0)   return -1;
    if(m<k-1) return 0;
    else if((m == k-1) || (m == k))  return 1;
    else{
        temp = new int[k];
        for(i=1; i<=k-2; i++)  temp[i] = 0;
        temp[0] = temp[k-1] = 1;  s = 0;
        for(i=k+1; i<=m; i++){
            sum = 2*temp[(i-1)%k]-s;
            s = temp[i%k];
            temp[i%k] = sum;
        }
         r=temp[m%k];
        delete [] temp;
        return r;
    }
}
```

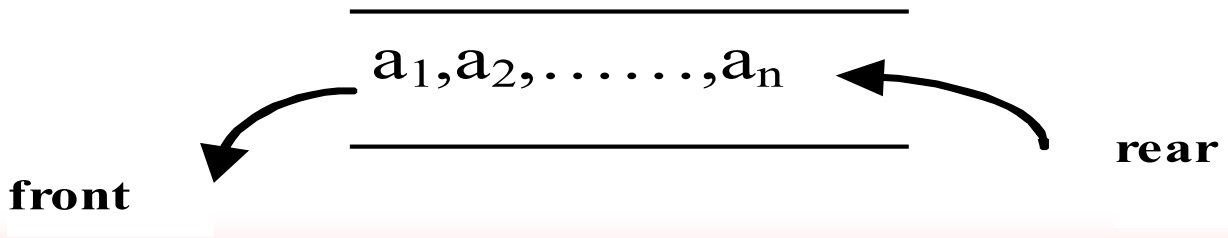
$$f_m = 2 * f_{m-1} - f_{m-(k+1)}$$

3.4 队列

1. 队列的定义：一种先进先出的线性表。

$q = (a_1, a_2, ..., a_n)$

- 队尾(rear)：允许插入的一端。
- 队头(front)：允许删除的一端。
- 空队列：不含元素的空表。
- 队头元素：a_1；队尾元素： a_n



2. 队列的特点

先进先出(First In First Out)，简称FIFO结构。

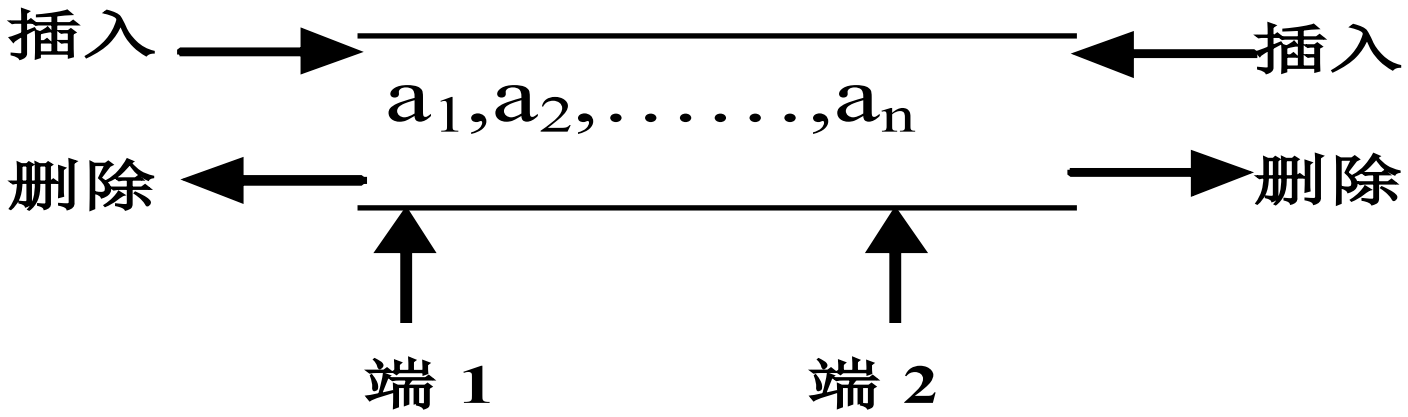
一种限定性数据结构。

3. 队列的基本运算

- 队列初始化 InitQueue(&Q)。设置一个空队列
- 入队列 EnQueue(&Q, e)。队列Q存在，插入e为新的队尾元素
- 出队列 DeQueue(&Q, &e)。Q存在且非空，删除队头元素
- 读队头元素 GetHead(Q, &e)。Q存在且非空，用e返回队头元素
- 判队列空 QueueEmpty(S)。Q存在, 若为空队列, 返回真，（假）

4. 双端队列（Deque）

限定插入和删除操作在表的两端进行的线性表。



- 输出受限的双端队列：一端允许插入和删除，另一端只允许插入；
- 输入受限的双端队列：一端允许插入和删除，另一端只允许删除。

5.队列的表示和实现

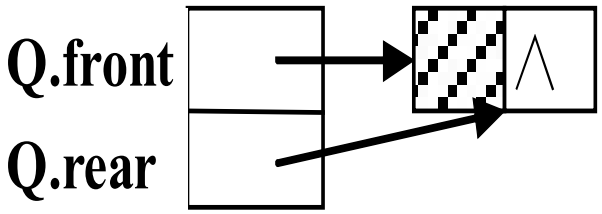
队列
- 链队列：队列的链式表示和实现
- 循环队列：队列的顺序表示和实现

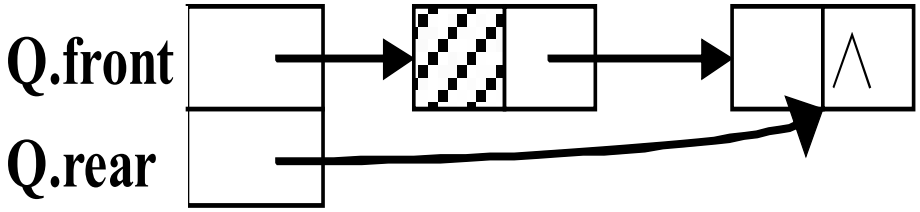
●链队列

LinkQueue Q;

链队列的类型定义

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





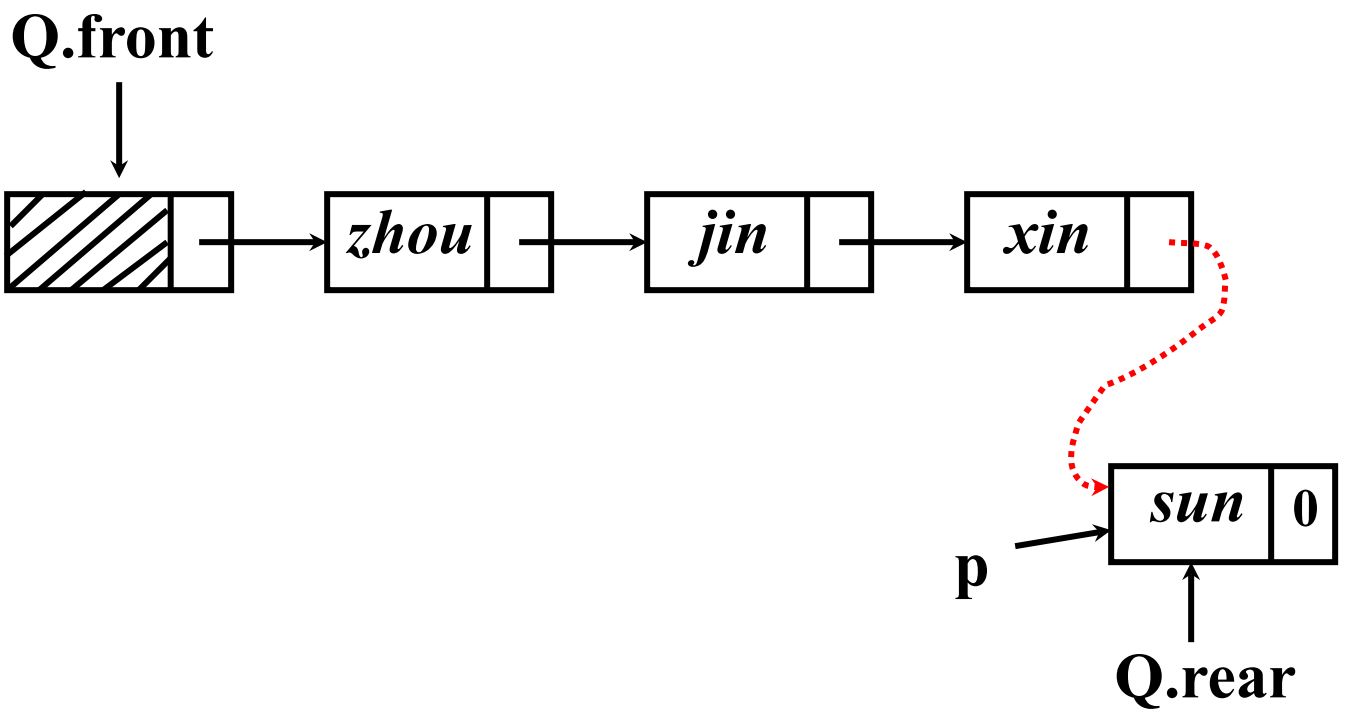

```
typedef struct{
    QueuePtr  front;  //队头指针
    QueuePtr  rear;   //队尾指针
}LinkQueue;
```

队空的条件：Q.front = Q.rear;(有头结点）

Q.front = NULL;(无头结点)

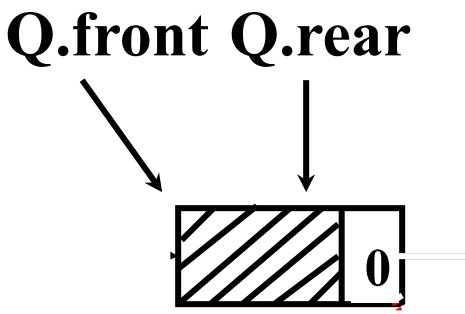
- 初始化 InitQueue(LinkQueue &Q)

```
{
    Q.front = Q.rear =new QNode;
    if(!Q.front) exit(OVERFLOW);
    Q.front->next = NULL;
    return OK;
}
```


Q.front
zhou
jin
xin
sun
0
p
Q.rear

- 入队列 EnQueue(LinkQueue &Q, QElemType e)

```
{
    p = new QNode;
    if(!p) exit(OVERFLOW);
    p->data = e;
    p->next = NULL;
 ①Q.rear->next = p;
 ②Q.rear = p;
    return OK;
}
```



e = p −> data = *xin*

- 出队列DeQueue(LinkQueue &Q, QElemType &e)

```
{
    if(Q.front == Q.rear)  return ERROR;
    p = Q.front->next;
    e = p->data;
 ①Q.front->next = p->next;(Q.front->next->next)
 ②if(Q.rear == p)  Q.rear = Q.front;
    delete p;
    return OK;
}
```

循环队列—顺序存储结构

- typedef struct{

```
        QElemType  *base;
         int              front;
         int              rear;
}SqQueue;
```

约定：空队列Q.front = Q.rear;

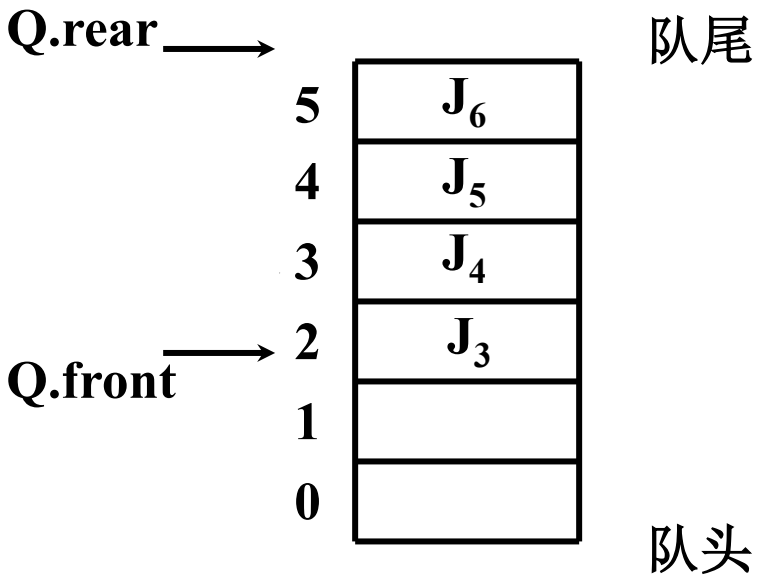
入队列（插入元素）Q.rear++;

出队列（删除元素）Q.front++;

非空队列，front指针始终指向队头元素，

rear指针始终指向队尾元素的下一位置。

插入、删除操作



插入元素 J_1;

插入元素 J_2、J_3;

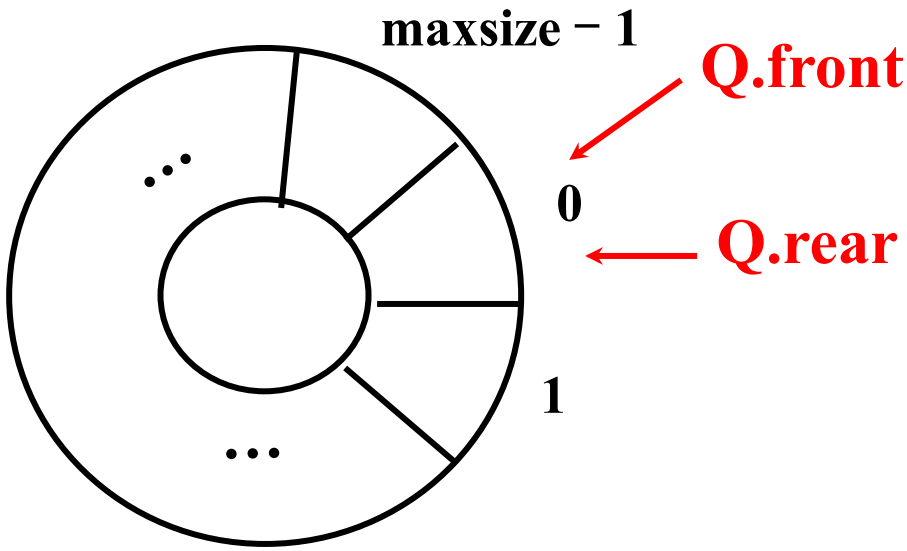
删除元素 J_1、J_2;

插入元素 J_4、J_5、J_6;

此时，队满，无法再插入新的元素，但实际队列中的可用空间并未真的被占满。

3. 循环队列—顺序存储结构

将顺序队列改造为一个环状的空间。



指针 front 、rear 分别指示队头和队尾下一个元素。

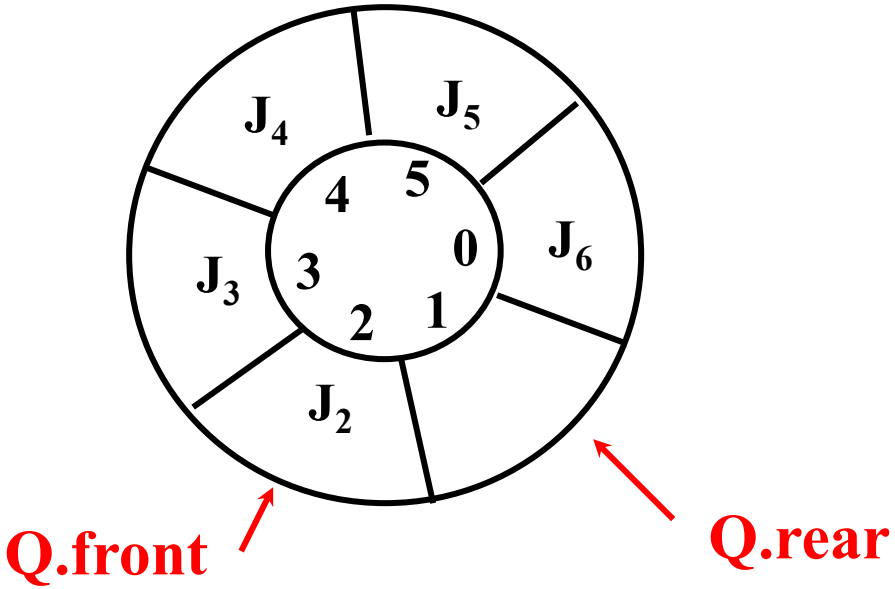
令 front = rear = 0 表示空队列。

每插入一新元素，rear = (rear + 1) % maxsize，

每删除一元素，front = (front + 1) % maxsize 。　// %：求余

插入、删除操作

maxsize = 6



初始，Q.front = Q.rear = 0，空队列。

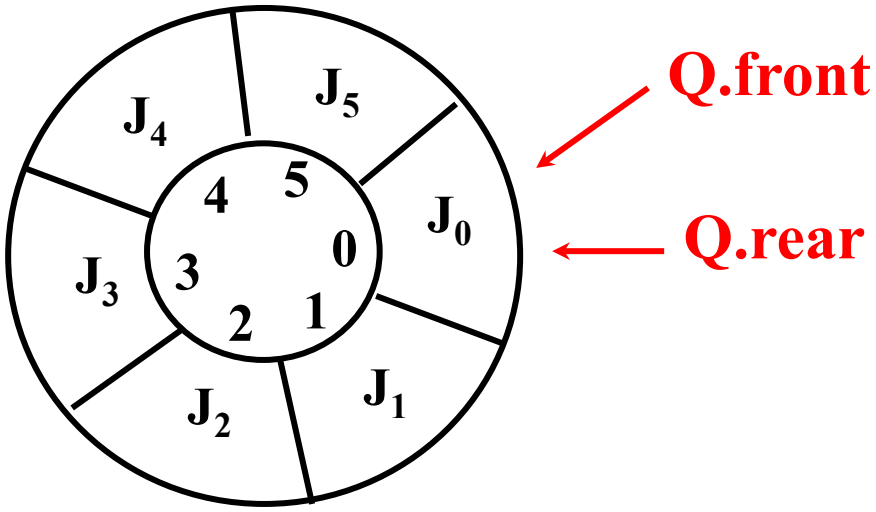
插入元素 J_0、J_1、J_2、J_3、J_4；

删除元素 J_0、J_1；

插入元素 J_5、J_6；

问题:

maxsize = 6



初始，Q.front = Q.rear = 0 ，空队列。

插入元素 J_0、J_1、J_2、J_3、J_4、J_5；

Q.front = Q.rear = 0 ，满队列。

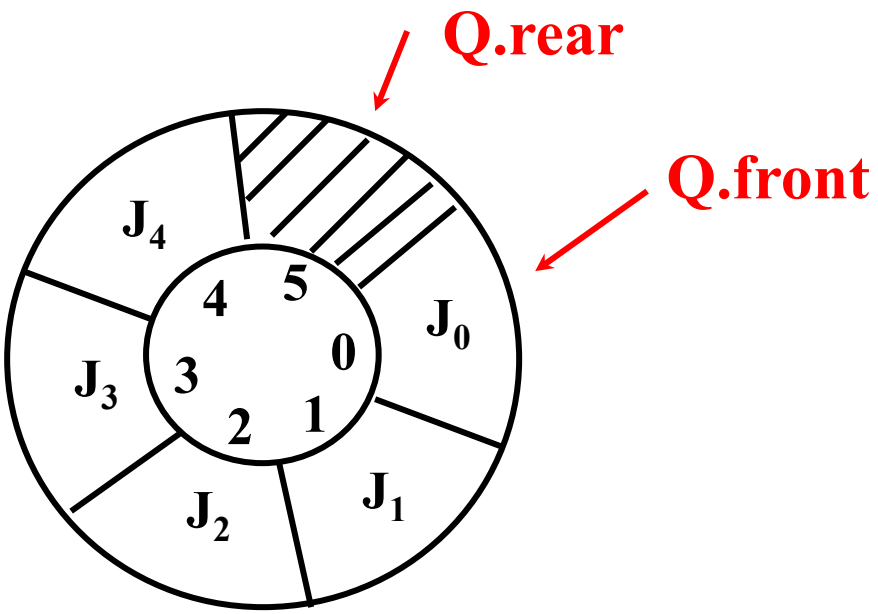
故无法通过 front = rear = 0 来分辨队空或队满。

解决方案:

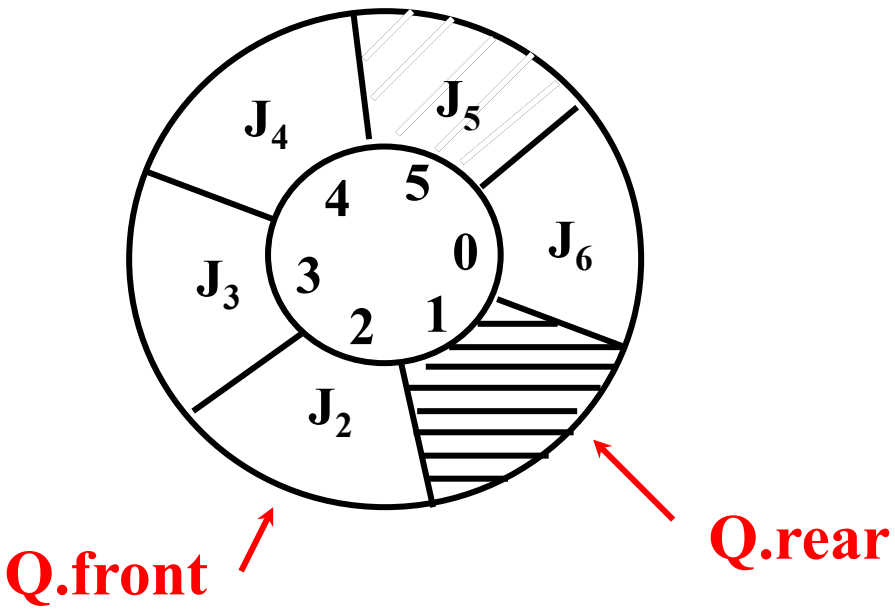
特殊空间，规定 front 与rear 之间总空出一个空间。

队空: Q.front == Q.rear

队满: Q.front == (Q.rear + 1) % maxsize



maxsize = 6



初始，Q.front = Q.rear = 0，空队列。

插入元素 J_0、J_1、J_2、J_3、J_4；

Q.front == (Q.rear + 1) % maxsize 队满

删除元素 J_0、J_1；

插入元素 J_5、J_6；

Q.front == (Q.rear + 1) % maxsize 队满

⑤循环队列中元素的个数

$$
\text{元素个数} = \begin{cases} 0; & \text{若Q.front = Q.rear 队列空} \\ \text{Q.rear – Q.front;} & \text{若Q.rear > Q.front} \\ \text{Q.rear – Q.front + MAXQSIZE;} & \text{若 Q.rear < Q.front} \\ \text{MAXQSIZE -1.} & \text{若队满} \end{cases}
$$

元素个数 = (Q.rear – Q.front + MAXQSIZE)% MAXQSIZE

- 初始化 InitQueue(SqQueue &Q)

```
{
    Q.base = (QElemType *) malloc(……);
    if(!Q.base)  exit(OVERFLOW);
    Q.front = Q.rear = 0;
    return OK;
}
```

- 队列长度 QueueLength(SqQueue Q)

```
{
    return (Q.rear – Q.front + MAXQSIZE)% MAXQSIZE;
}
```

- 入队列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;
}
```

- 出队列 DeQueue(SqQueue &Q, QElemType &e)

```
{
    if(Q.front == Q.rear)
        return ERROR;
    e = Q.base[Q.front];
    Q.front = (Q.front+1)%MAXQSIZE;
    return OK;
}
```

6.队列总结

- 队列是一种具有线性结构的数据结构，是操作受限的线性表；
- 队列的特点是先进先出，只能在队尾插入元素和队头删除元素，称为入队列和出队列；
- 循环队列中，队空标志： Q.front == Q.rear

  队满标志： (Q.rear+1)%MAXQSIZE == Q.front;
- 链队列中，Q.front = Q.rear,

  在用户空间范围内不会出现队满的情况。

- 作业：

  3.13

练习：

设A是一个栈，栈中共有n个元素$a_1a_2\ldots a_n$，a_n为栈顶元素，B是一个队列，队列中有n个元素$b_1b_2\ldots b_n$，b_1为队头元素，b_n为队尾元素，A，B采用顺序存储结构且空间足够大，现将栈中元素全部移到队列中，使得队列中元素与栈中元素交替排列，即B中元素为$b_1a_1b_2a_2\ldots b_na_n$，问至少多少次基本操作才能完成上述工作。（除A，B外使用的其它附加存储量为1）。

①先将栈中元素依次入队列，栈空；

②将$b_1b_2\ldots b_n$依次出队列，入队列；

③将$a_na_{n-1}\ldots a_1$出队列，入栈；

④顺次b_i出队列，入队列，a_i出栈，入队列。

$2n+2n+2n+4n = 10n$ 个基本操作

2.19 删除元素递增排列的链表L中值>mink且<maxk的所有元素

```
void Delete_Between(Linklist &L,int mink,int maxk)
{
      p=L;
      while(p->next && p->next->data<=mink)
            p=p->next;  //p是最后一个不大于mink的元素
       q=p->next;
       while(q && q->data<maxk){
                  s = q; q=q->next;  free(s);
       }
       p->next=q;
}
```