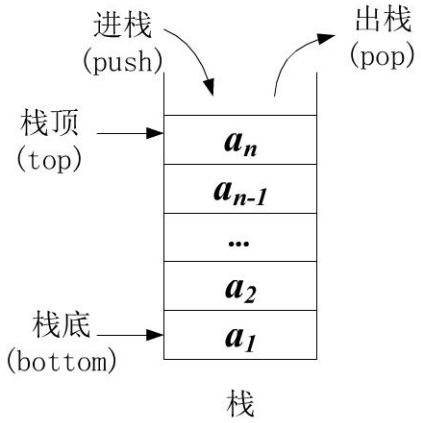
第三章 栈和队列

- 栈
- 栈的应用
- 队列
- 队列的应用

栈和队列　--栈的定义

- 栈和队列称为**运算受限的线性表。**
- **栈（stack）**是指只允许在表的末端进行插入和删除的线性表。栈又叫做**后进先出（LIFO）**的线性表。



栈

栈和队列 --栈的抽象数据类型定义

ADT Stack {

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

数据关系：$R1=\{ <a_{i-1}, a_i> | a_{i-1}, a_i \in D, i=2,...,n \}$

约定a_n端为栈顶， a_1端为栈底。

基本操作：

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。

ClearStack(&S)

初始条件：栈 S 已存在。

操作结果：将 S 清为空栈。

栈和队列 --栈的抽象数据类型定义

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，否则返回FALSE。

StackLength(S)
初始条件：栈 S 已存在。
操作结果：返回栈 S 中元素个数，即栈的长度。

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。

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。

栈和队列 --栈的抽象数据类型定义

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。

StackTraverse(S, visit( ))

初始条件：栈 S 已存在且非空，visit( )为元素的访问函数。

操作结果：从栈底到栈顶依次对S的每个元素调用函数visit( )，一旦visit( )失败，则操作失败。

} ADT Stack

栈和队列 --顺序栈

- 基于数组的存储表示实现的栈称为**顺序栈**。
- 顺序栈的存储表示

```
#define STACK_INIT_SIZE 100;
#define STACKINCREMENT 10;
typedef struct {
   SElemType *base;
   SElemType *top; // 栈顶指针
   int stacksize; // 当前已分配的存储空间,以元素为单位
}SqStack;
```

p46

栈和队列 --顺序栈的函数实现

```
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); /* 存储分配失败 */
  S.top=S.base;
  S.stacksize=STACK_INIT_SIZE;
  return OK;
 }
```


栈和队列 --顺序栈的函数实现

```
Status DestroyStack(SqStack &S) /* 销毁栈S*/
{
   free(S.base);
  S.base=NULL;
  S.top=NULL;
  S.stacksize=0;
   return OK;
 }
```


栈和队列 --顺序栈的函数实现

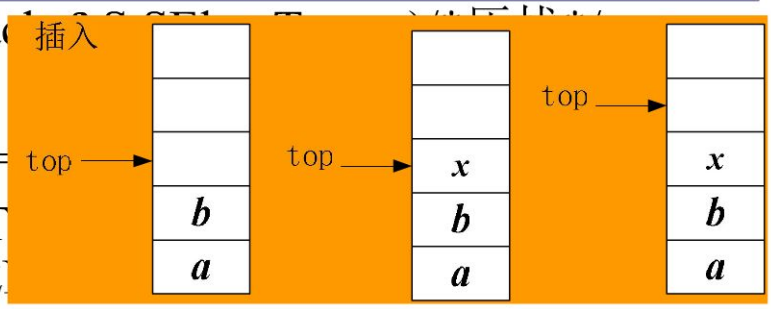
```
Status ClearStack(SqStack &S) /* 把S置为空栈 */
{
   S.top=S.base; return TRUE;
}
Status StackEmpty(SqStack S)
{
  if(S.top==S.base)
        return TRUE;
  else
        return FALSE;
}
```


栈和队列 --顺序栈的函数实现

```
int StackLength(SqStack S) /* 返回栈的长度 */
{return S.top-S.base;}
Status GetTop(SqStack S,SElemType &e)
{if(S.top>S.base)
   {
    e=*(S.top-1);
    return OK;
   }
   else
    return ERROR;
}
```

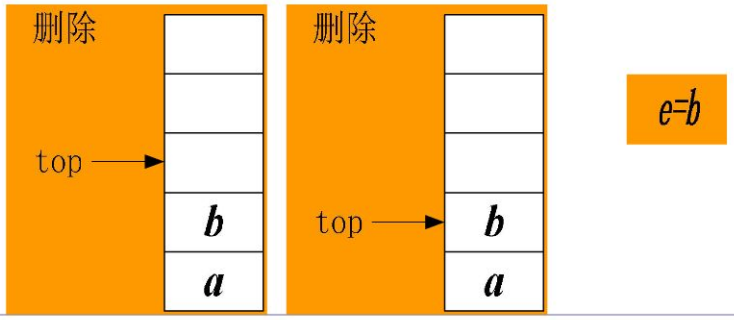

栈和队列 --顺序栈的函数实现

```
Status Push(SqSta
{
  if(S.top-S.base>=
  S.base=(SElemT
  ACKINCREME
  if(!S.base) exit(OVERFLOW);
  S.top=S.base+S.stacksize;
  S.stacksize+=STACKINCREMENT;
 }
 *S.top++=e; return OK;
}
```



栈和队列 --顺序栈的函数实现

```
Status Pop(SqStack &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==S.base)  return ERROR;
  e=*--S.top;
  return OK;
}
```

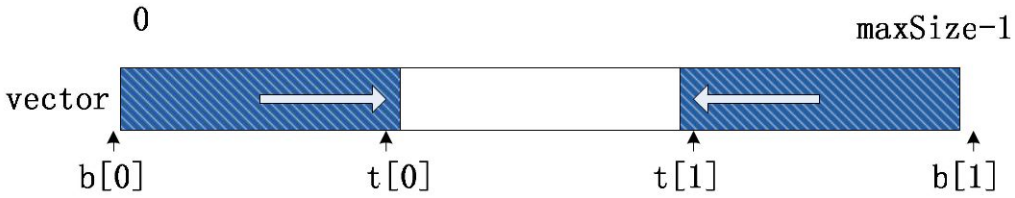


栈和队列 --顺序栈的函数实现

- 总结：
  - 栈不存在的条件： base=NULL;
  - 栈为空 的条件 ： base=top;
  - 栈满的条件 ： top-base=stacksize;

栈和队列 --双栈共享一个栈空间

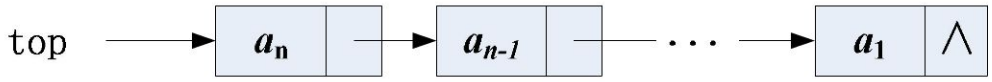
- 为了既能减少由于栈满而引起的**溢出**，又能有效的**利用存储空间**，提出一种双栈共享一个栈空间的策略。



双栈的情形

栈和队列 --链式栈

- **链式栈**是栈的链接存储表示。
- 链式栈的栈顶在链表的表头。因此，新结点的插入和栈顶结点的删除都在链表的表头，即栈顶进行。



链式栈

- 链栈不必设头结点，因为栈顶（表头）操作频繁；
- 采用链栈存储方式，可使多个栈共享空间；当栈中元素个数变化较大，且存在多个栈的情况下，链栈是栈的首选存储方式。

栈和队列

- 例1：一个栈的入栈序列是a、b、c、d、e，则栈的不可能的输出序列是（C）。

  A．edcba　B．decba　C．dceab　D．abcde

- 例2：对于一个栈，给出输入项a、b、c。如果输入序列由a、b、c组成，试给出全部可能的输出序列。

  abc　acb　bac　bca　cba

栈和队列 --栈的应用举例

- 数制转换

  十进制N和其它进制数的转换是计算机实现计算的基本问题,其解决方法很多,其中一个简单算法基于下列原理:

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

如：

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

栈和队列 --栈的应用举例

对于输入的任意一个非负十进制整数，打印输出与其等值的八进制数

```
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);
        }
}
```


栈和队列 --栈的应用举例

- 括号匹配的检验

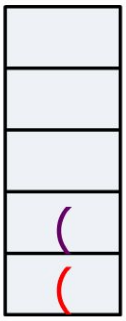
  算法思想：

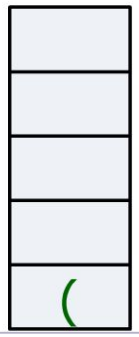
  - 如果输入为左括号，则入栈；
  - 当输入右括号时，立即与栈顶元素配对，若配对成功，则弹出栈顶元素。否则证明输入的括号序列不匹配；
  - 当输入结束符时，栈非空（栈中仍有左括号）则输入的括号序列不匹配。否则匹配成功。

栈和队列 --栈的应用之括号匹配

- 观察：每个右括号将与最近遇到的那个未匹配的左括号相匹配。

$(a\times(b+c)-d)$ $(a+b))($





没有与右括号匹配的左括号

没有与左括号匹配的右括号

栈和队列 --栈的应用举例

```
Status AllBrackets_Test(char *str)//判别表达式中三种括号是否匹配
   { InitStack(s);
     for(p=str;*p;p++)
     {
      if(*p=='('||*p=='['||*p=='{') push(s,*p);
      else if(*p==')'||*p==']'||*p=='}')
      {
        if(StackEmpty(s)) return ERROR;
        pop(s,c);
        if(*p==')'&&c!='(') return ERROR;
        if(*p==']'&&c!='[') return ERROR;
        if(*p=='}'&&c!='{') return ERROR;      }
     }//for
     if(!StackEmpty(s)) return ERROR;
     return OK;
   }//AllBrackets_Test
```


栈和队列 --栈的应用举例

- 行编辑程序

  可设输入缓冲区为一个栈结构，每当从终端接受一个字符之后先作如下判断：如果它既不是退格符也不是退行符，则将该字符压入栈顶；如果是一个退格符，则从栈顶删去一个字符；如果他是一个退行符，则将字符栈清空。 P50

如：

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

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

While(*s)

putchar(*s++);

栈和队列 --栈的应用举例

- 表达式求值

（1）要正确求值，首先了解算术四则运算的规则：

a. 从左算到右

b. 先乘除，后加减

c. 先括号内，后括号外

（2）根据上述三条运算规则，在运算的每一步中，对任意相继出现的算符θ1和θ2，都要比较优先权关系。算符优先法所依据的算符间的优先关系见教材P53表3.1。

栈和队列 --栈的应用举例

（3）算法思想：

设定两栈：操作符栈 OPTR ，操作数栈 OPND

栈初始化：设操作数栈 OPND 为空；操作符栈OPTR 的栈底元素为表达式起始符 ‘#’；

依次读入字符：是操作数则入OPND栈，是操作符则要判断：

当操作符 ＜栈顶元素，则退栈、计算，结果压入OPND栈；

当操作符＝栈顶元素且不为‘#’，脱括号（弹出左括号）；

当操作符＞栈顶元素，压入OPTR栈。 算法见p53

栈和队列 --栈的应用之表达式求值2

- 一个表达式由操作数（亦称运算对象）、操作符（亦称运算符）和分界符组成。
- 算术表达式有三种表示：
  - 中缀(infix)表示

    <操作数> <操作符> <操作数>，如 A+B；
  - 前缀(prefix)表示

    <操作符> <操作数> <操作数>，如 +AB；
  - 后缀(postfix)表示

    <操作数> <操作数> <操作符>，如 AB+；

栈和队列 --栈的应用之表达式求值2

- 表达式示例
  - 中缀表达式　A+B*(C-D)-E/F
  - 后缀表达式　ABCD-*+EF/-
  - 前缀表达式　-+A*B-CD/EF
- 表达式中相邻两个操作符的计算次序为：
  - 优先级高的先计算；
  - 优先级相同的自左向右计算；
  - 当使用括号时从最内层括号开始计算。

栈和队列 --栈的应用之表达式求值2

- 应用后缀表示计算表达式的值

说明：这里只考虑双目运算的情形。

- 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。

计算例 A B C D － × + E F /－

栈和队列 --栈的应用之表达式求值2

- 通过后缀表示计算表达式值的过程：
  - 顺序扫描表达式的每一项，如果该项是操作数，则将其入栈；
  - 如果该项是操作符<op>，则连续从栈中退出两个操作数X和Y，形成运算指令X<op>Y，并将计算结果重新压入栈中。
  - 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。

栈和队列 --栈的应用之表达式求值2

A B C D －×+ E F /－

步	扫描项	项类型	动作	栈中内容
1			置空栈	空
2	A	操作数	进栈	A
3	B	操作数	进栈	AB
4	C	操作数	进栈	ABC
5	D	操作数	进栈	ABCD
6	－	操作符	D、C退栈，计算C－D，结果R_1进栈	ABR_1
7	×	操作符	R_1、B退栈，计算B×R_1，结果R_2进栈	A R_2

栈和队列 --栈的应用之表达式求值2

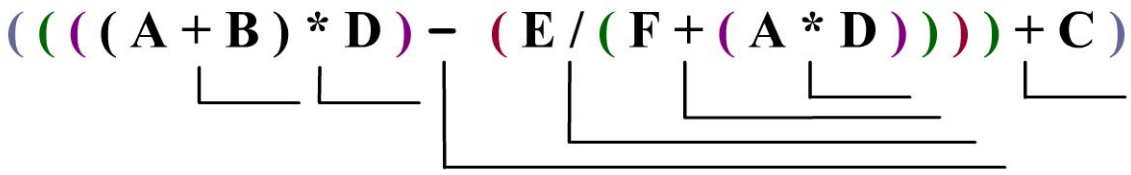
A B C D －×+ E F /－

步	扫描项	项类型	动作	栈中内容
8	+	操作符	R_2、A退栈，计算A +R_2，结果R_3进栈	R_3
9	E	操作数	进栈	R_3 EF
10	F	操作数	进栈	R_3 EF
11	/	操作符	E、F退栈，计算E/F，结果R_4进栈	R_3 R_4
12	－	操作符	R_3、R_4退栈，计算R_3 －R_4，结果R_5进栈	R_5

栈和队列 --栈的应用之表达式求值2

- 中缀表示→转后缀表示
  - 先对中缀表达式按运算优先次序加上括号；
  - 再把操作符后移到右括号的后面并以就近移动为原则；
  - 最后将所有括号消去。

例：中缀表示(A+B)*D-E/(F+A*D)+C，转换为后缀表示



后缀表示 A B + D * E F A D * + / - C +

栈和队列 --栈的应用之栈与递归

- 递归的定义

  若一个对象部分地包含它自己，或用它自己给自己定义，则称这个对象是递归的；若一个过程直接地或间接地调用自己，则称这个过程是递归的过程。

- 以下三种情况常常用到递归方法。
  - 定义是递归的
  - 数据结构是递归的
  - 问题的解法是递归的

栈和队列 --栈的应用之栈与递归

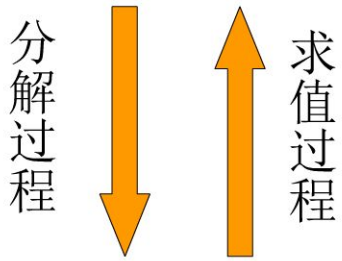
- 定义是递归的

例如：

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

- 求解阶乘函数的递归算法

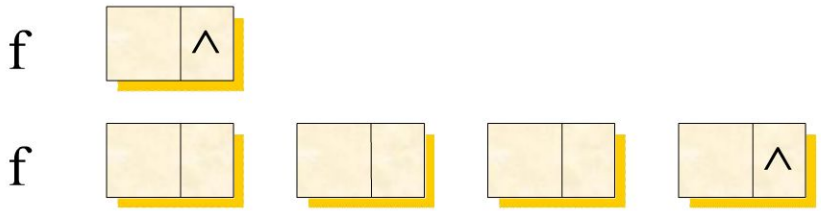
```
① long Factorial(long n) {
②     if (n == 0) return 1;
③     else return n*Factorial(n-1);
④ }
```



栈和队列 --栈的应用之栈与递归

- 数据结构是递归的

例如，单链表结构



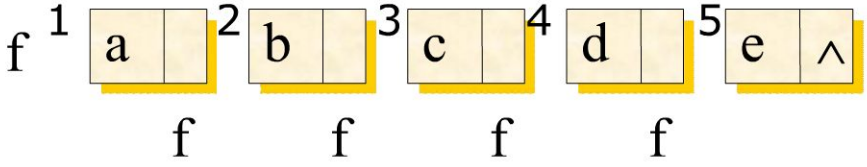
一个结点，它的指针域为NULL，是一个单链表;
一个结点，它的指针域指向单链表，仍是一个单链表。

栈和队列 --栈的应用之栈与递归

搜索链表最后一个结点并打印其数值

```
① void Print(LinkList f) {
②     if (f ->next == NULL)
③         cout << f ->data << endl;
④     else Print(f ->next);
⑤ }
```

递归找链尾



栈和队列 --栈的应用之栈与递归

- 问题的解法是递归的
  - 例如汉诺塔(Tower of Hanoi)问题

问题描述：

有A,B,C三个塔座，A上套有n个直径不同的圆盘，按直径从小到大叠放，形如宝塔,编号1,2,3……n。要求将n个圆盘从A移到C，叠放顺序不变，移动过程中遵循下列原则：

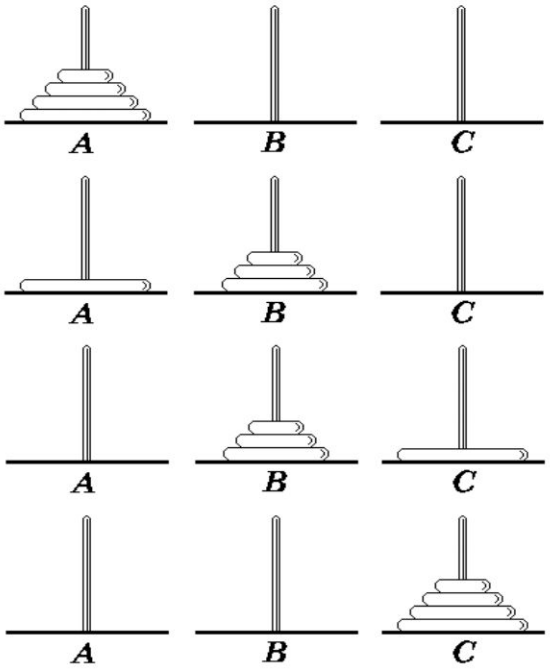
- 每次只能移一个圆盘
- 圆盘可在三个塔座上任意移动
- 任何时刻，每个塔座上不能将大盘压到小盘上

栈和队列 --栈的应用之栈与递归

解决方法：

如果 n = 1 柱移到 C 柱上。否

- ☑ 用 C 个盘子移到 B 柱上
- ☑ 将 A C 柱上；
- ☑ 用 A 个盘子移到 C 柱上



栈和队列 --栈的应用之栈与递归

☑ 算法：

```
void hanoi (int n, char x, char y, char z) {  // 算法3.5
①  if (n==1)
②        move(x, 1, z);  //将编号为 1 的圆盘从x移到z
③  else {
④     hanoi(n-1,x,z,y);
⑤     move(x, n, z);        //将编号为n的圆盘从x移到z
⑥     hanoi(n-1, y, x, z);
        //将y上编号为 1 至n-1的圆盘移到z,x作辅助塔
  }}
void move(char x, int n, char z) {
  printf("  %2i. Move disk %i from %c to %c\n",++Count,n,x,z);}
```


栈和队列　　--栈的应用之栈与递归

栈和队列 --栈的应用之栈与递归

- 递归过程改为非递归过程
  - 递归过程简洁、易编、易懂
  - 递归过程效率低，重复计算多
  - 改为非递归过程的目的是提高效率
  - 单向递归和尾递归可直接用迭代实现其非递归过程
  - 其他情形必须借助栈实现非递归过程

栈和队列 --栈的应用之栈与递归

Ackerman函数定义如下：

$$akm(m,n)=\begin{cases} n+1 & m=0 \\ akm(m-1,1) & m\neq 0, n=0 \\ akm(m-1,akm(m,n-1)) & m\neq 0, n\neq 0 \end{cases}$$

栈和队列　　--栈的应用之栈与递归

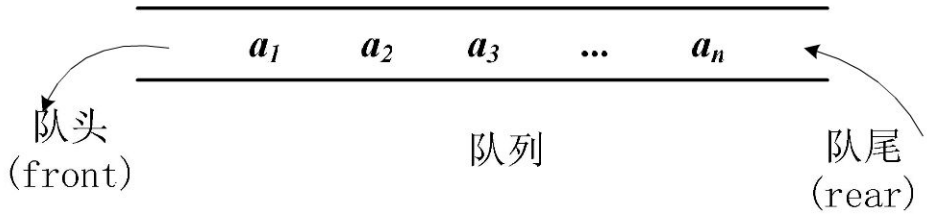
递归算法：

```
unsigned akm(unsigned m, unsigned n)
{
   if(m==0) return n+1;     //m==0
   else if(n==0) return akm(m-1,1);  //m>0,n==0
        else return akm(m-1,akm(m,n-1));//m>0,n>0
}
```

非递归算法：　见习题册p189

栈和队列　　--队列的定义

- 队列（Queue）是只允许在表的一端插入，在另一端删除的线性表。队列又叫先进先出（**FIFO**）的线性表。



栈和队列 --队列的抽象数据类型定义

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$\mathbf{a}_1$端为队列头， $\mathbf{a}_n$端为队列尾

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。

栈和队列 --队列的抽象数据类型定义

ClearQueue(&Q)

初始条件：队列 Q 已存在。

操作结果：将 Q 清为空队列。

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。

GetHead(Q,&e)

初始条件：Q 为非空队列。

操作结果：用 e 返回Q的队头元素。

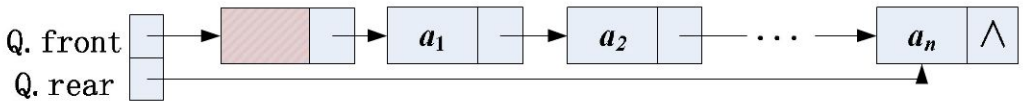
栈和队列 --队列的抽象数据类型定义

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。
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。
QueueTraverse(Q,visit( ))
初始条件：队列 Q 已存在且非空，visit( )为元素的访问函数。
操作结果：依次对 Q 的每个元素调用函数 visit( )，一旦 visit( ) 失败则操作失败。
} **ADT** Queue

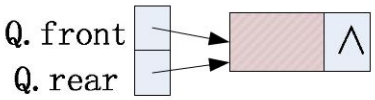
栈和队列　　--链式队列

- **链式队列**是基于单链表的一种存储表示。
- 队列的队头指针指向单链表的头结点，队尾指针指向单链表的最后一个结点。

链式队列



空队列



栈和队列　　--链式队列

```
typedef struct QNode {// 结点类型
   QElemType     data;
   struct QNode  *next;
 } QNode, *QueuePtr;
typedef struct { // 链队列类型
   QueuePtr  front;  // 队头指针
   QueuePtr  rear;   // 队尾指针
} LinkQueue;
```

基本操作的函数原形说明　p61

栈和队列 --链式队列的函数实现

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


栈和队列 --链式队列的函数实现

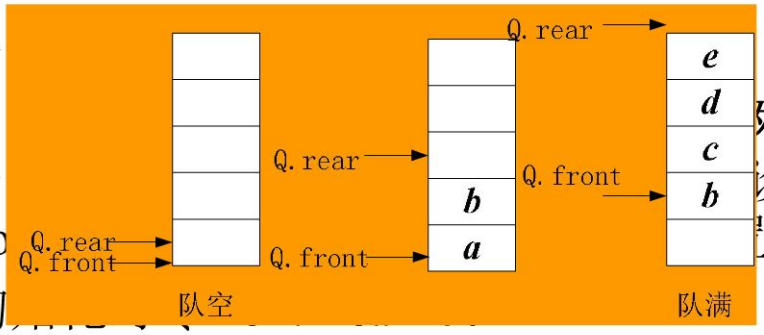
```
Status EnQueue (LinkQueue &Q, QElemType e) {
   // 插入元素e为Q的新的队尾元素
   p = (QueuePtr)malloc(sizeof(QNode));
   if (!p)  exit (OVERFLOW);   //存储分配失败
   p->data = e;   p->next = NULL;
   Q.rear->next = p;    Q.rear = p;
   return OK;
}
```


栈和队列　　--链式队列的函数实现

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


栈和队列　　--循环队列

- 顺序



队列　　列。利用一个一　　设置两个指针fro　　。

  - 初
  - 队尾指针rear指示了实际队尾位置的后一位置，队头指针front则指示真正队头元素所在位置。
  - 当front==rear，队列为空；
  - 当rear==MAXQSIZE，队列满。

栈和队列 --循环队列

- 循环队列的概念

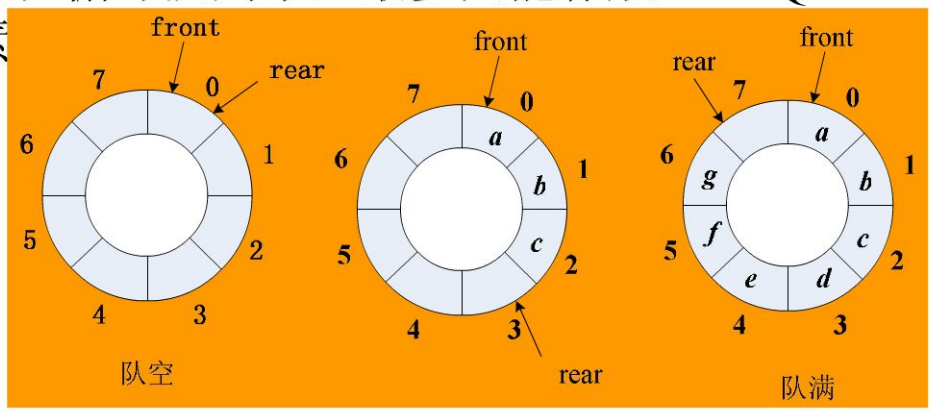
  解决假溢出的办法之一：将队列元素存放数组首尾相接，形成循环（环形）队列。

  - 队列存放数组被当作首尾相接的表处理。
  - 队头、队尾指针加1时从MAXQSIZE -1直接进到0，可用语言的取模(余数)运算实现。
  - 队头指针进1：front = (front+1) % MAXQSIZE;
  - 队尾指针进1：rear = (rear+1) % MAXQSIZE;

栈和队列 --循环队列

- 队列初始化：front = rear = 0;
- 队空条件：front == rear;
- 队满条件：(rear+1) % MAXQSIZE == front;

注意，在循环队列中，最多只能存放MAXQSIZE -1个元素



栈和队列　　--循环队列

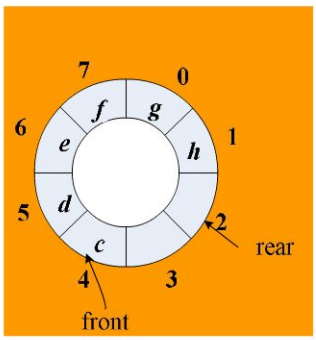
```
#define MAXQSIZE  100  //最大队列长度
typedef struct {
   QElemType  *base;  // 动态分配存储空间
   int  front;     // 头指针，若队列不空，指向队列头
                   //元素
   int  rear;      // 尾指针，若队列不空，指向队列尾
                   //元素 的下一个位置
} SqQueue;
```


栈和队列 --循环队列的函数实现

```
Status InitQueue (SqQueue &Q) {
  // 构造一个空队列 Q
  Q.base =(QElemType*)malloc(MAXQSIZE*sizeof
   (QElemType));
   if (!Q.base) exit (OVERFLOW);
   Q.front = Q.rear = 0;
    return OK;
}
```

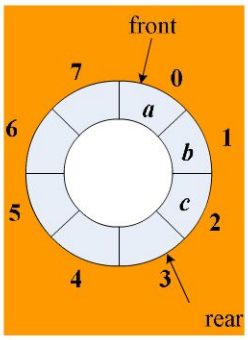

栈和队列　　--循环队列的函数实现

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



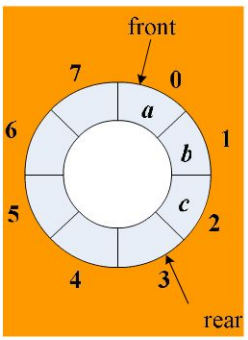
栈和队列　　--循环队列的函数实现

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



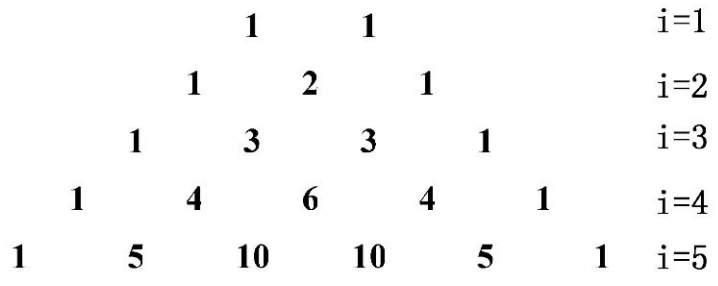
栈和队列 --循环队列的函数实现

```
Status DeQueue (SqQueue &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;
   e = Q.base[Q.front];
   Q.front = (Q.front+1) % MAXQSIZE;
   return OK;
}
```



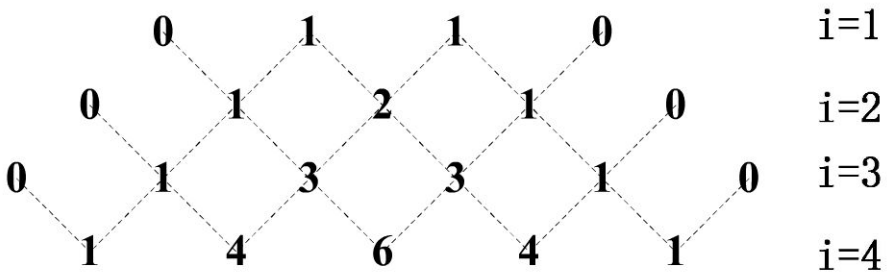
栈和队列
--队列的应用之打印杨辉三角形

- 将二项式 $(a+b)^i$ 展开，其系数构成杨辉三角形。



杨辉三角形

栈和队列
--队列的应用之打印杨辉三角形



第i行元素与第i+1行元素的关系

栈和队列
--队列的应用之打印杨辉三角形

```
void YANGVI(int n) {
        SqQueue  Q;
        InitQueue(Q);  //队列初始化
        int i=1,j,s=k=0,t,u;
        EnQueue(Q,i);  EnQueue(Q,i);
```

1	1	0	1	2	1	0				

某时刻队列情形

栈和队列
--队列的应用之打印杨辉三角形

```
for (int i = 1; i <= n; i++) { //逐行计算
        cout << endl;
        EnQueue(Q,k);
        for (int j = 1; j <= i+2; j++) {  //下一行
           DeQueue(Q,t);
           u=s+t; EnQueue(Q,u);
           s = t;
           if (j!= i+2) cout << s << ' ';  //第j+2个是0
       }}}
```


栈和队列 --优先级队列的概念

- 每次从队列中取出的是具有最高优先权的元素，这种队列就是**优先级队列（Priority Queue）**。
  - 最小优先级队列（min Priority Queue）：数字越小优先权越高
  - 最大优先级队列（max Priority Queue）：数字越大优先权越高
  - 优先权相同时先来先服务。

任务编号	1	2	3	4	5
优先权	20	0	40	30	10
执行顺序	3	1	5	4	2

栈和队列 --作业

- 3.20
- 3.27
- 3.31