第三章 栈和队列

制作：数据结构在线课程课题组

南京审计大学 信息工程学院

2022.10

内容概要

1、栈的类型定义

2、栈类型的实现

3、栈的应用举例

4、队列的类型定义

5、队列类型的实现

栈的例子

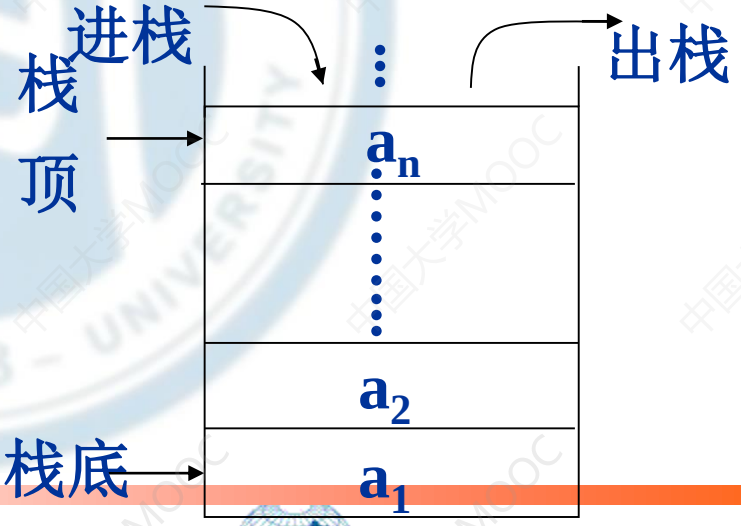
认识——栈

- 栈是在程序设计中被广泛使用的线性数据结构。

- 与线性表相比，它们的插入和删除操作受更多的约束和限定，故又称为限定性的线性表结构。
  - 线性表允许在表内任一位置进行插入和删除；
  - 栈只允许在表尾一端进行插入和删除；

3.1 栈

- 栈
  - 限定只能在表的一端进行插入和删除操作的线性表。
  - 栈顶(top)：允许插入和删除的一端。
  - 栈底(bottom)：不允许插入和删除的一端。
  - 空栈：不含元素的空表。
- 特点
  - 先进后出（FILO）
  - 后进先出（LIFO）



例3.1　设有4个元素a、b、c、d进栈,给出它们所有可能的出栈次序。

答:所有可能的出栈次序如下:　　卡特兰数

abcd　abdc　acbd　acdb

adcb　bacd　badc　bcad

bcda　bdca　cbad　cbda

cdba　dcba

例3.2 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　大小中

(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。

例3.3 已知一个栈的进栈序列是1,2,3,…,n,其输出序列是$p_1,p_2,\ldots,p_n$,若$p_1=n$,则p_i的值__。

(A) i　　(B) n-i

(C) n-i+1　　(D) 不确定

答:当$p_1=n$时,输出序列必是n,n-1,…,3,2,1,则有:

$p_2=n-1$, $p_3=n-2$, …, $p_n=1$

推断出$p_i=n-i+1$,所以本题答案为C。

栈的抽象数据类型定义

ADT Stack {

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

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

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。

栈的抽象数据类型定义

ClearStack(&S)
初始条件：栈 S 已存在。
操作结果：将 S 清为空栈

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。

栈的抽象数据类型定义

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。

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。

栈的抽象数据类型定义

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

写出下列程序段的输出结果

```
void main( ){
Stack S;
Char x,y;
InitStack(S);
x=’c’;y=’k’;
Push(S,x); Push(S,’a’);  Push(S,y);
Pop(S,x); Push(S,’t’); Push(S,x);
Pop(S,x); Push(S,’s’);
while(!StackEmpty(S)){ Pop(S,y);printf(y); };
Printf(x);
}
```

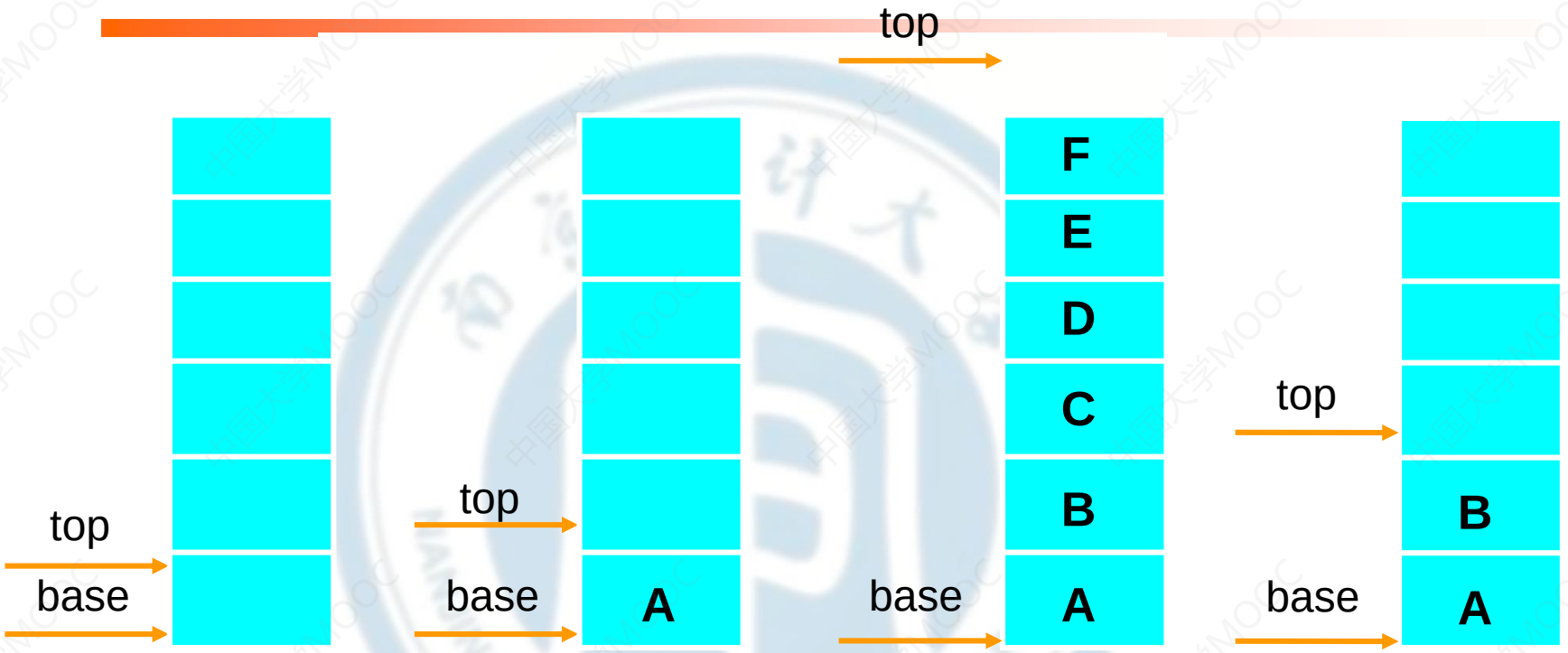
stack

3.1.2 栈的表示和实现

一、 顺序栈：

```
#define STACK_INIT_SIZE 100
#define STACKINCREMENT 10
typedef struct{
    SElemType   *base;
    SElemType   *top;
    int         stacksize;
}SqStack;
```

栈顶指针始终指向栈顶元素的下一位置



若top初始值=base

为空栈

入栈 top=top+1

出栈 top=top-1

2、特点：简单、方便，但易产生溢出。

上溢(overflow):栈已经满，又要压入元素；

下溢(underflow):栈已经空，还要弹出元素；

注：上溢是一种错误，使问题的处理无法进行下去；而下溢一般认为是一种结束条件，即问题处理结束。

基本操作的算法描述

```
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;
} //InitStack
```

时间复杂度0（1）

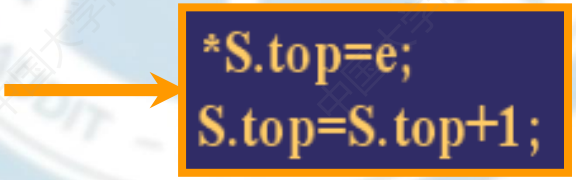
基本操作的算法描述

```
Status GetTop (SqStack S, SElemType &e)
{   // 若栈不空，则用 e 返回S的栈顶元素，并返回OK,否则返回ERROR
    if (S.top == S.base ) return ERROR; // 空栈
    e = *(S.top-1);          // 返回非空栈中栈顶元素
    return OK;
} //GetTop
```

时间复杂度0（1）

基本操作的算法描述

```
Status Push (SqStack &S, SElemType e)
{ // 插入元素 e 为新的栈顶元素
    if(S.top-S.base>=S.stacksize)      {//栈满，追加存储空间
    S.base=(SElemType *)realloc(S.base,
    (S.stacksize+STACKINCREMENT)*sizeof(SElemType));
 if(!S.base) exit(OVERFLOW);       // 存储分配失败
    S.top=S.base+S.stacksize;
    S.stacksize +=STACKINCREMENT;
    }
    *(S.top++) = e;                                  // 插入新的元素
    return OK;
} //Push
```



时间复杂度0（1）

基本操作的算法描述

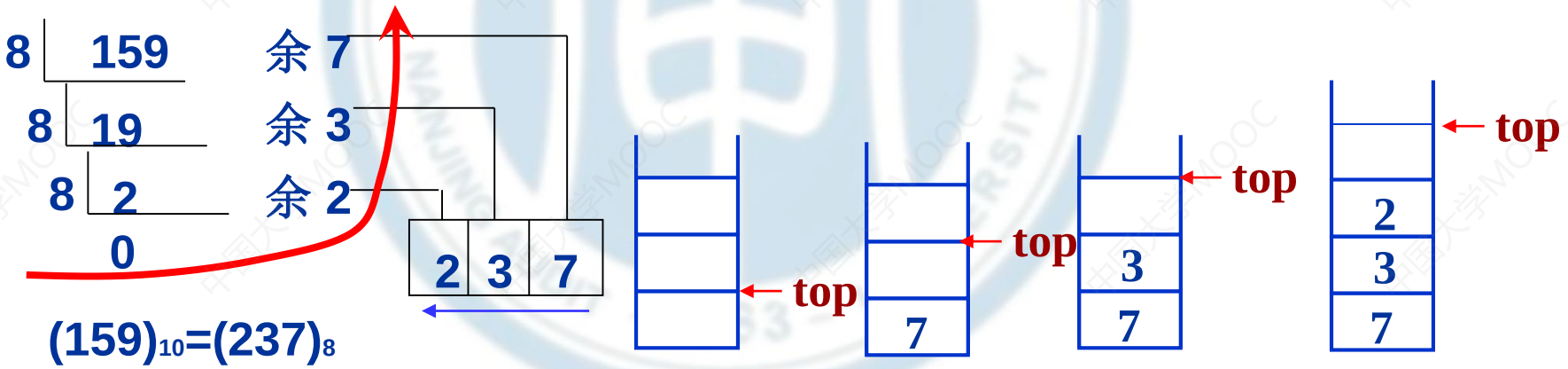
```
Status Pop (SqStack &S, SElemType &e)
{ // 栈不空，删除S的栈顶元素，用e返回其值，并返回 OK；否则返回 ERROR
    if (S.top == S.base) return ERROR;          //空栈
    e = *(--S.top);                             //返回非空栈中栈顶元素
    return OK;
} //Pop
```

时间复杂度0（1）

```
S.top=S.top-1;
e=*S.top;
```

3.2 栈的应用——数制转换

- 应用一：数制转换
  - 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。
  - 例　把十进制数159转换成八进制数。



$(159)_{10}=(237)_8$

栈的应用——数制转换

算法3.1 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

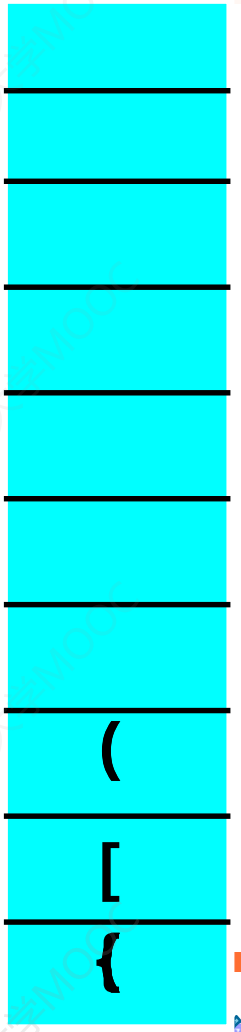
```
void conversion (){
    InitStack(S);
    scanf("%d",N);
    while(N){
        Push(S,N%8);
        N = N/8;
    }
    while (!StackEmpty(s)){
        Pop(S,e);
        printf("%d",e);
    }
}
```

栈的应用——括号匹配算法

(1+3)*{ [(3/4) +(4%8)] } +(4/6)

用一个栈stack进行判断，将“（”，“[”，“{”入栈；当遇到“）”,”]”或”}”时，检查当前的栈顶元素是否是对应的“（”，“[”，“{”；若是则退栈，否则返回提示“不配对”。

当整个算术表达式检查完毕时，若栈为空则表示括号配对正确，否则不正确。



算法的设计思想：

1）凡出现左括号，则进栈；

2）凡出现右括号，首先检查栈是否空：

若栈空　则表明该“右括号”多余不匹配

否则和栈顶元素比较，

若相匹配，则“左括号出栈”

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“左括号”有余不匹配

3.3 栈与递归

递归：在定义自身的同时又出现了对自身的调用。

使用递归的三种环境 ：

1、用递归定义的数学函数，如阶乘函数；

2、有的数据结构，如二叉树、广义表，由于结构本身固有的递归特性，则它们的操作可递归地描述；

3、虽然问题本身没有明显的递归结构，但是用递归求解比迭代求解更简单，如Honoi塔问题。

函数调用时的处理

- 当一个函数在运行期间调用另一个函数时，在运行该被调用函数之前，需先完成三件事：
  - 将所有的实在参数、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。
- 而从被调用函数返回调用函数之前，应该完成：
  - 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。

多个函数嵌套调用的规则是：

后调用的先返回 ！

此时的内存管理实行“栈式管理”

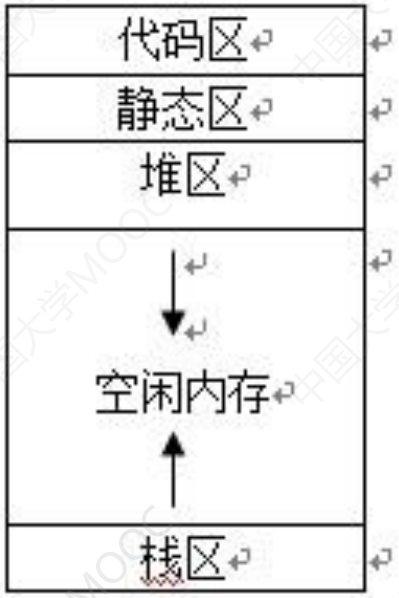
例如：

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

Main的数据区

运行时存储空间的划分

- 一个程序在运行时内存会被划分成以下部分
  - 代码区，静态数据区，堆区，栈区
- 栈可以用来传递函数参数、存储局部变量、以及存储返回值的信息，还可以用于保存寄存器的值以供恢复之用



递归实例-n！

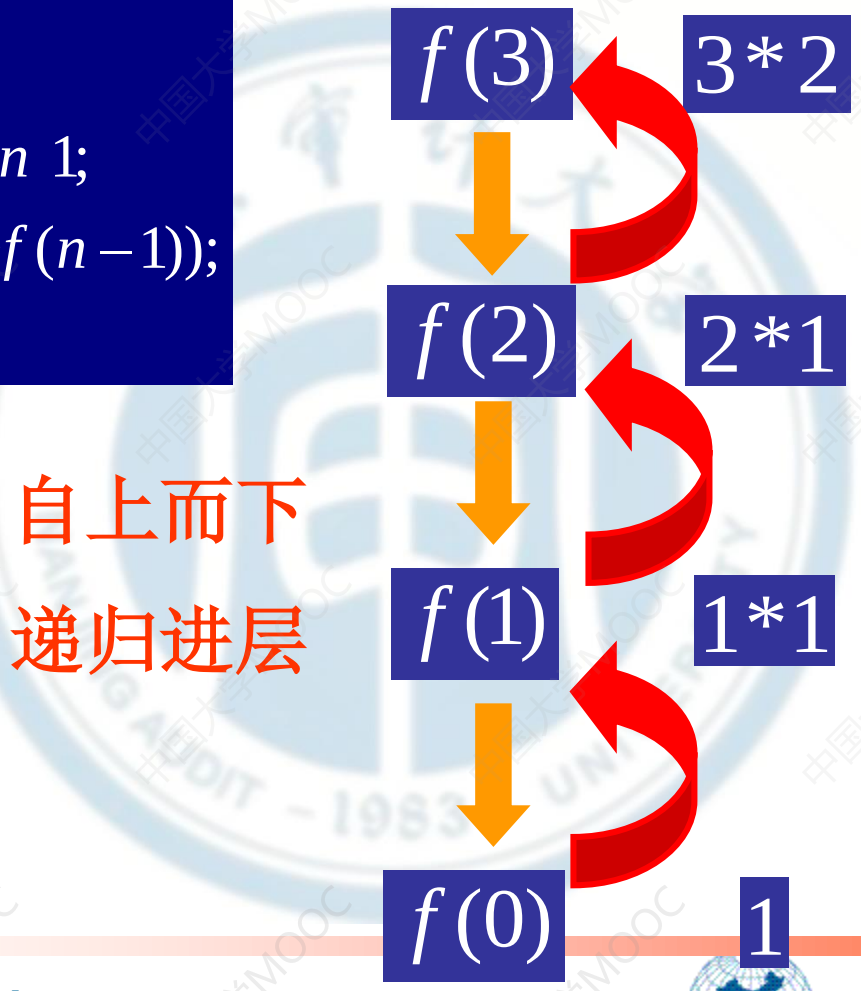
$$n! = \begin{cases} 1 & n = 0 \\ n*(n\text{-}1)! & n \geq 1 \end{cases}$$

其递归算法如下：

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

递归实例-n！

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



递归实例–斐波那契数列

F(0)=0，　F(1)=1，　F(n)=F(n-1)+F(n-2)　（n>=2）

```
int fibonacci(int n)
{
    if(n==0)
        return 0;
    else if (n==1)
        return 1;
    else
        return fibonacci(n-1)+fibonacci(n-2);
}
```

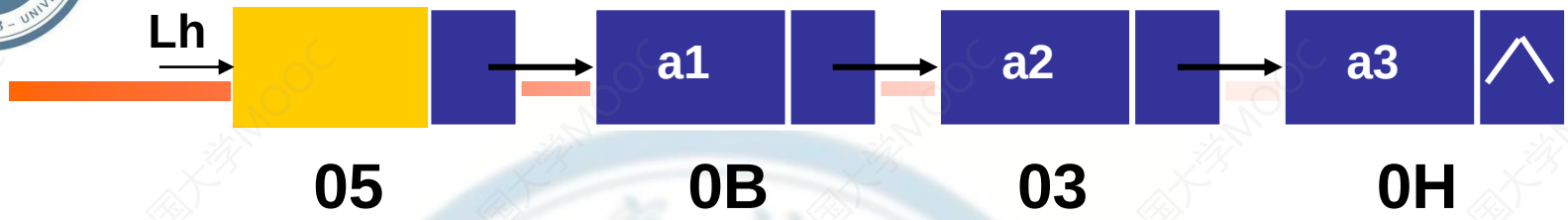
递归的特点：

1、递归执行时需要系统提供隐式栈来实现，效率低，费时；

2、有些计算机语言不支持递归，如FORTRAN、BASIC；

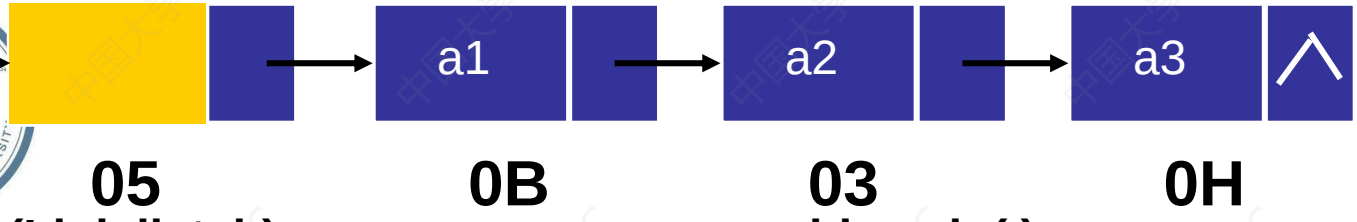
3、递归是一种分而治之，把复杂问题分解为简单问题的求解方法。对求解某些复杂问题，递归是有效的。

递归实例-单链表输出1




```
Status ListTraverse(LinkList L)
 { if (L==NULL||L->next==NULL)
     return ERROR;
   else
     {    LinkList p=L->next;
          printf("%d ",p->data);
         ListTraverse(p);
     }
   return OK;
   }
```

86627
44513




```
void  rev(Link list  h)
{   if (h->next!=NULL)
       rev(h->next);
    printf("%d",h->data);
 }
```

```
void  main( )
{   //创建带头节点的单链表
    rev(lh->next);
    printf("\n");
}
```

```
rev(0B);  void  rev(0B)
          {   if (03!=NULL)
                 rev(03);
              printf("%d",0B->data);
           }
```

```
void  rev(03)
{   if (0H!=NULL)
       rev(0H);
    printf("%d",03->data);
}
```

递归实例-单链表输出2

递归实例-单链表输出3

```
void PrintList (LinkList h)
{
    if(h!=NULL)
    {
        printf("%d",h->data);
        PrintList (h->next);
    }
}
```

写递归算法的建议

① 找准最小情况

② 确保问题的规模不断变小

③ 特殊情况尽量放在递归函数外处理，因为递归执行的单位是整个函数

④ 注意参数传递方式

⑤ 考虑问题全面

⑥ 大胆地写！谨慎地演算，测试！

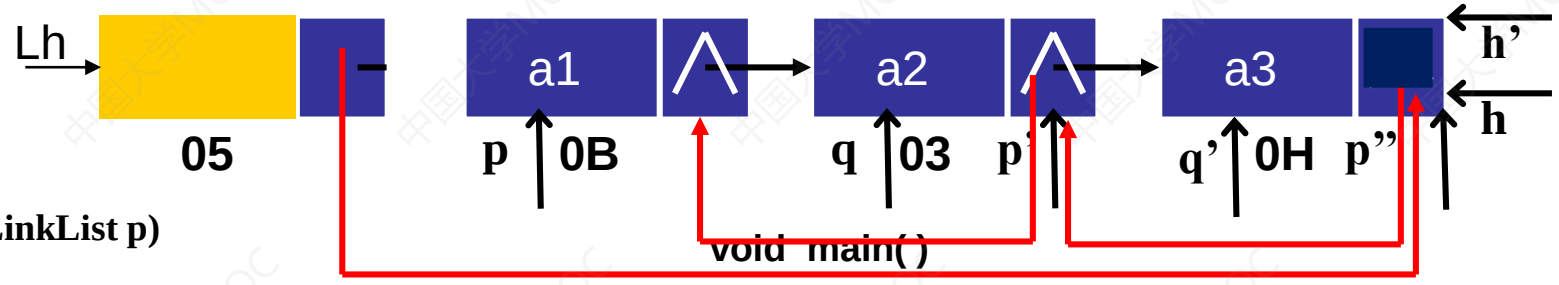
递归实例-单链表的长度

```
int   Len (LinkList L)
{
    int i=0;
    LinkList p=L->next;
    while(p)
    {
        i++;
        p=p->next;
    }
    return i;
}
```

```
int   Len (LinkList L)
{
    if    (L->next==NULL)
        return 0;
    else
        return 1+Len (L->next);
}
```

递归实例–单链表的逆置

```
LinkList reverse(LinkList p)
{   LinkList q,h;
    if(p->next == NULL)
        return p;
    else{
        q = p->next;
        h = reverse(q);
        q->next = p;
        p->next = NULL;
        return h; }
}
```




```
LinkList rev(LinkList p)
{
    if(p->next == NULL)
        return p;
    else{
        q = p->next;
        h = rev(q);
        q->next = p;
        p->next = NULL;
        return h; }
}
```

```
void main( )
{  //创建带头节点的单链表
   lh->next=rev(lh->next);
}
```

```
LinkList  rev(0B)
{   if (03= =NULL)

   else{
        q = p->next;
        h = rev(q);
```

```
LinkList  rev(03)
{   if (0H= =NULL)

   else{
        q = p->next;
        h = rev(q);
```

```
LinkList  rev(0H)
{   if (NULL= =NULL)
        return p;

}
```

递归实例-单链表的逆置

```
void reverse(LinkList pc,LinkList &pd)
{
        LinkList p;
        if (pc->next == NULL)
                pd=pc;
        else
        {
                p=pc->next;
                reverse(p,pd); //递归逆置后继结点
                p->next=pc;          //将后继结点指向当前结点。
                pc->next=NULL;
        }
}
```

递归实例-单链表的顺序归并

```
LinkList MergeList(LinkList pa,LinkList pb)
{
    LinkList pc;
    if  (pa==NULL)
        return pb;
    if  (pb==NULL)
        return pa;
    if (pa->data <= pb->data)
    {  pc=pa;
       pc->next = MergeList(pa->next,pb);
    }
    else
    {
       pc=pb;
       pc->next = MergeList(pa,pb->next);
    }
    return pc;
}
```

现实生活中的队列

认识栈和队列

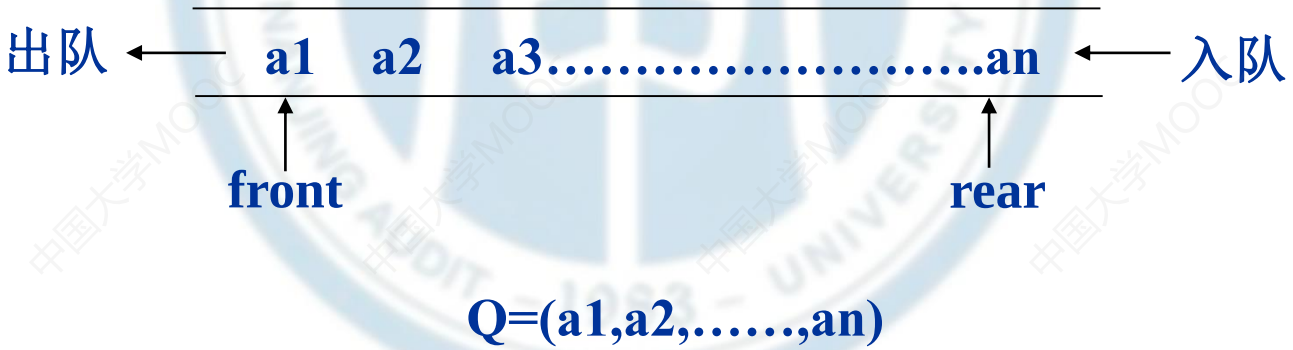
- 栈和队列是在程序设计中被广泛使用的两种线性数据结构。

- 与线性表相比，它们的插入和删除操作受更多的约束和限定，故又称为限定性的线性表结构。
  - 线性表允许在表内任一位置进行插入和删除；
  - 栈只允许在表尾一端进行插入和删除；
  - 队列只允许在表尾一端进行插入，在表头一端进行删除。

3.4 队列

队列

- 只允许在一端进行插入而在另一端进行删除的线性表。
- 队尾：允许插入的一端。
- 队头：允许删除的一端。
- 特点：先进先出（FIFO）。



Q=(a1,a2,……,an)

队列的抽象数据类型定义

ADT Queue {

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

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

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。

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。

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

```
void main( ){
Queue Q;  Init Queue (Q);
Char x=’e’; y=’c’;
EnQueue (Q,’h’); EnQueue (Q,’r’);  EnQueue (Q, y);
DeQueue (Q,x); EnQueue (Q,x);
DeQueue (Q,x); EnQueue (Q,’a’);
while(!QueueEmpty(Q)){
    DeQueue (Q,y);printf(y); };
Printf(x);
}
```

char

队列类型的实现

- 链 队 列——链式存储
- **循环队列——顺序存储**

循环队列的结构定义

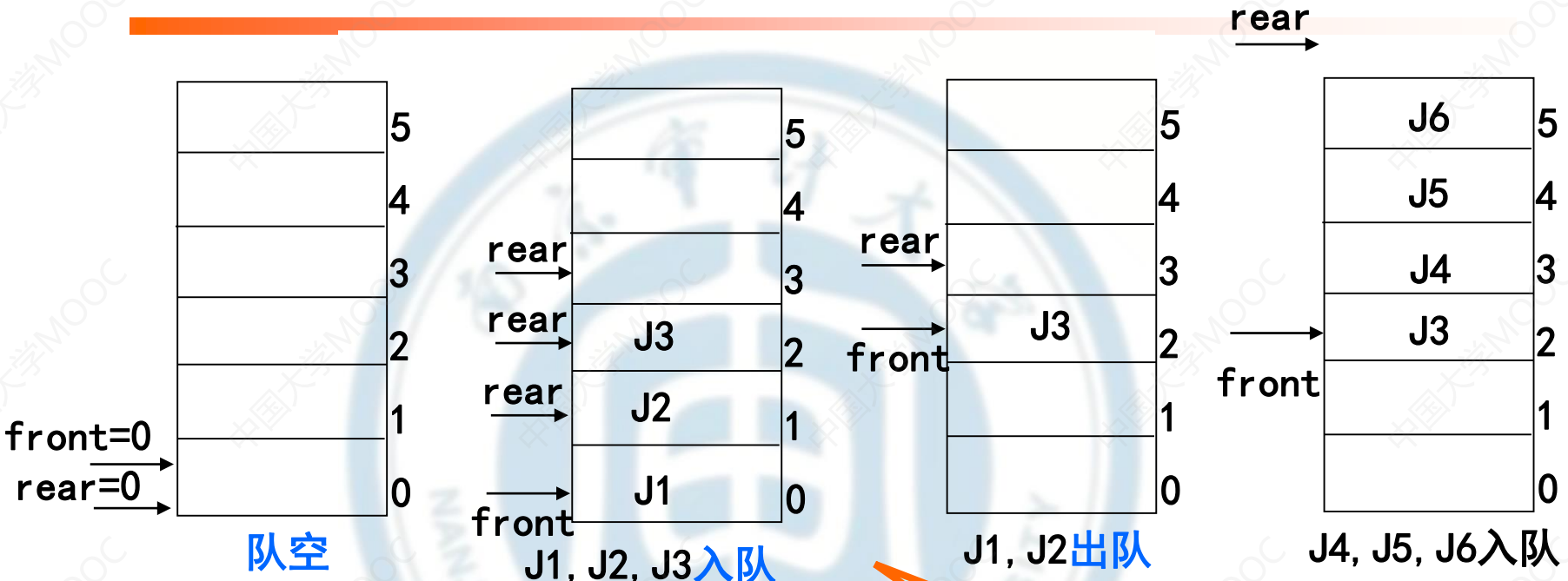
```
#define MAXQSIZE 100  // 最大队列长度
typedef struct {
        QElemType *base; // 初始化的动态分配存储空间
        int rear;        // 队尾指针，指向队尾元素的下一个位置
        int front;       // 队头指针，指向队头元素的位置
} SqQueue;
```

队列的顺序存储结构

1、存储方式：用一组地址连续的存储单元依次存放从队头到队尾的元素。但是由于在两端操作，设两个指示器，（rear和front）分别指示队列的尾和首。

特别约定头指针始终指向队列首部，而尾指针始终指向队列尾元素的下一位置。

队列的顺序存储结构



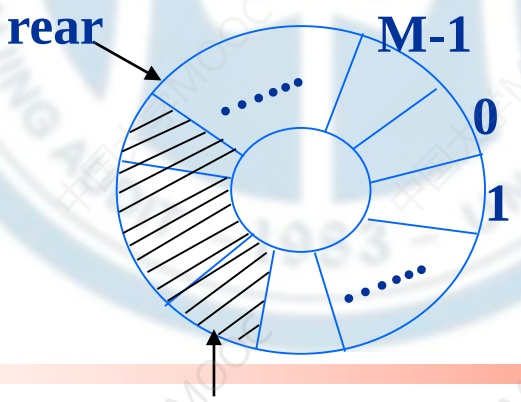
- 存在问题（设数组大小为M），则：
  - 当front=0，rear=M时，再有元素入队发生溢出——真溢出。
  - 当front≠0，rear=M时，再有元素入队发生溢出——假溢出。

设两个指针front,rear,约定非空时：
rear指示队尾元素的下一个位置；
front指示队头元素
初值front=rear=0

空队列条件：front==rear
入队列：~~sq[rear++]=x;~~
出队列：x=sq[front++];

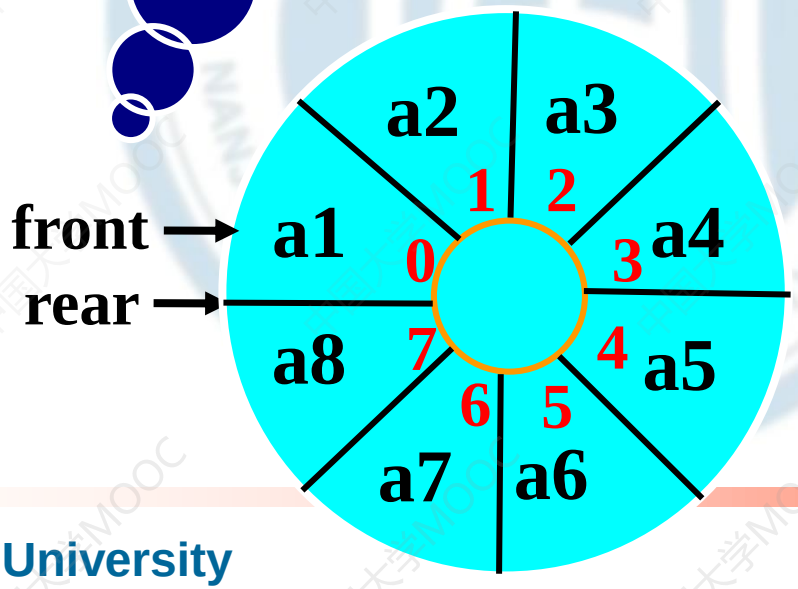
解决方案

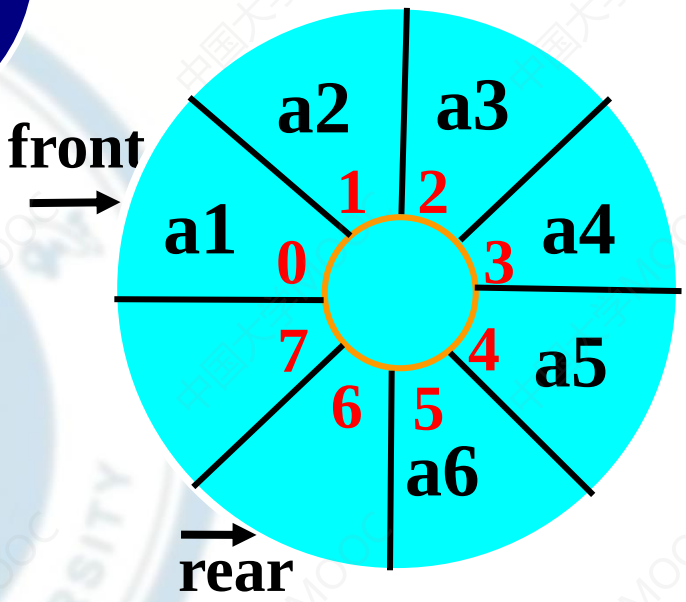
- 队首固定，每次出队剩余元素向下移动——浪费时间。
- 循环队列
  - 基本思想：把队列臆想成环形，让sq[0]接在sq[M-1]之后，若rear+1==M,则令rear=0;其余情况rear=rear+1
  - 入队： sq[rear]=x; rear=(rear+1)%M;
  - 出队： x=sq[front]; front=(front+1)%M;



即队列为空或者队列满时，都是front=rear，如何区分？

顺序队列
然存在问
我们知道
front=rear；
右图中，继续入
front=rear





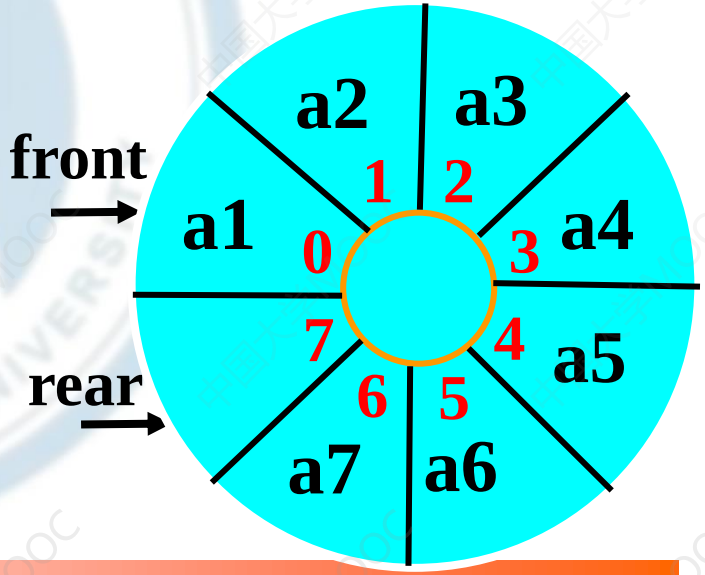
两种方法：

1）设置标志位以区别队列是“空”还是“满”

因出队而相等，则为空；

因入队而相等，则为满；

2）少用一个元素的空间，约定（rear+1）maxsize==front，就认为队满

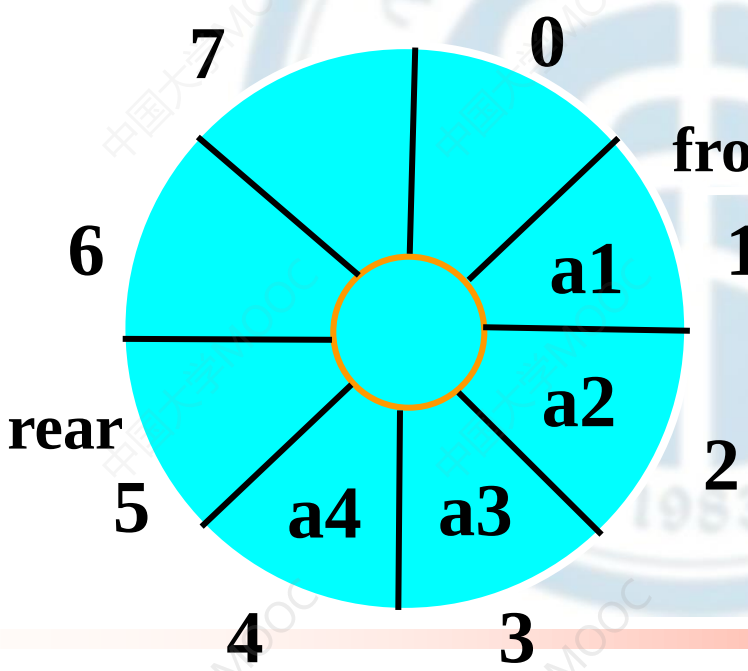


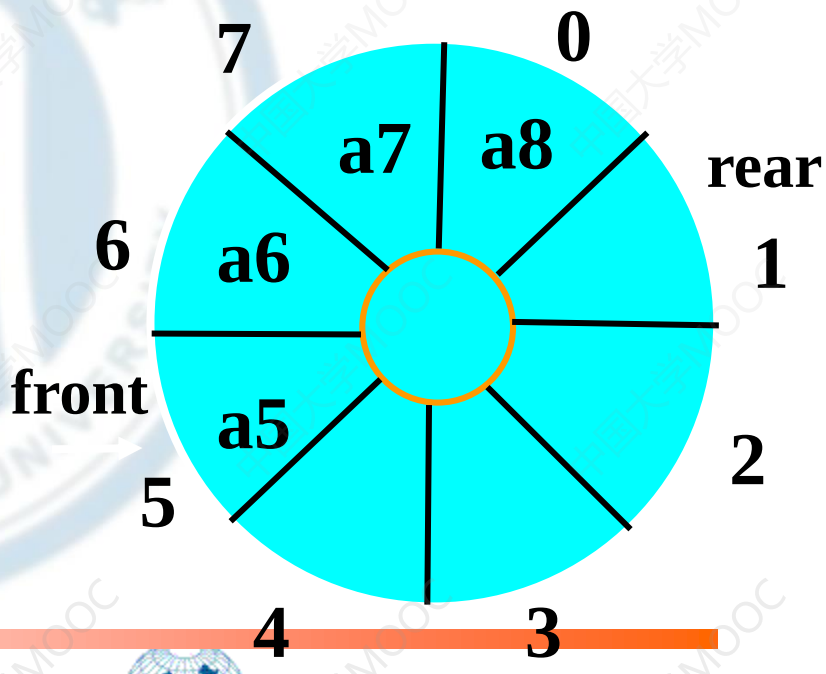
循环队列的基本操作描述

```
Status InitQueue (SqQueue &Q){
    // 构造一个空队列 Q
    Q.base = (QElemType *)malloc(MAXQSIZE
                    *sizeof(QElemType)); // 为循环队列分配存储空间
    if (!Q.base) exit(OVERFLOW);                    // 存储分配失败
    Q.front = Q.rear = 0;
    return OK;
} // InitQueue
```

循环队列的基本操作描述

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





循环队列的基本操作描述

```
Status EnQueue (SqQueue &Q, QElemType e){
    // 插入元素 e 为新的队列尾元素
    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;
}
```

循环队列的基本操作描述

```
Status DeQueue (SqQueue &Q, QElemType &e){
    // 若队列不空，则删除当前队列Q中的队头元素，用 e 返回其值,并返回OK
    if (Q.front == Q.rear)    return ERROR;
    e = Q.base[Q.front];
    Q.front = (Q.front+1) % MAXQSIZE;
    return OK;
}
```

本章小结

- 栈和队列都属线性结构，因此他们的存储结构和线性表非常类似，同时由于他们的基本操作要比线性表简单得多，因此它们在相应的存储结构中实现的算法都比较简单，相信对大家来说都不是难点。
- 这一章的重点则在于栈和队列的应用。通过本章所举的例子学习分析应用问题的特点，在算法中适时应用栈和队列。

实验五作业——括号匹配算法

```
Status check(char *str){
InitStack(s);
int i=0;
while (str[i]!='\0'){
  switch (str[i]){
      case'(':Push(s,'(');break;
      case'[':Push(s,'[');break;
      case'{':Push(s,'{');break;
      case')':Pop(s,e);
            if (e!='(')  return ERROR;
      case']':Pop(s,e);
            if (e!='[')  return ERROR;
      case'}':Pop(s,e);
            if (e!='{')  return ERROR;
      default;
       }
i++;
}
if (IsEmpty(s) )
          return   OK;
else
          return ERROR;
}
```

实验六作业——数值转换递归实现

```
#include "public1.h"
#define N 8
void conversion(int n)
{
  if(n/N>0)
     conversion(n/N);
  printf("%d",n%N);
}
```

```
void main()
{
    int n;
    printf("将十进制整数n转换为%d进制
                数，请输入：n(≥0)=",N);
    scanf("%u",&n);
    conversion(n);
    printf("\n");
}
```

关于《数据结构》思政课程项目

目标：

① 树立科学辩证唯物的世界观；

② 树立爱国、爱党、爱岗、爱专业的价值观

③ 树立把计算机成为促进社会进步发展、造福人类的工具的专业观