# 栈和队列

## 栈和队列的定义和特点

### 栈（stacks）

定义：仅在表尾进行插入或删除操作的线性表。

结构：栈顶、栈底

特点：后进先出（LIFO）、先进后出（FILO）

存储结构：顺序栈、链栈（顺序栈常见）

逻辑结构：一对一

主要算法：入栈、出栈、读栈顶元素值、建栈、判断栈满、栈空等。

### 队列（queues）

定义：在队尾插入，在队头删除。

特点：先进先出。

存储结构：循环队列（顺序队列），链队列。

逻辑结构：一对一

实现方式：入队、出队函数是重点

### 栈的实际用途

- 数值转换
- 括号匹配的检验
- 表达式求值

## 栈的表示和操作的实现

**前提：一定要预设栈顶指针 `top` ！**

`base==top` 是空栈的标志

## 顺序栈

### 顺序栈的表示

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

### 顺序栈的初始化

```
Status InitStack(SqStack &S){
    S.base=new SElemType[MAXSIZE];
    if(!S.base)
        return OVERFLOW;//Judge if it exsit
    S.top=S.base;
    S.stackSize=MAXSIZE;
    return OK;
}
```

### 判断顺序栈是否为空

```
bool StackEmpty(SqStack S){
    if(S.top==S.base)
        return true;
    else
        return false;
}
```

### 求顺序栈长度

```
int StackLength(SqStack S){
    return S.top-S.base;
}
```

### 清空顺序栈

```
Status ClearStack(SqStack S){
    if(S.base)
        S.top=S.base;
    return OK;
}
```

### 顺序栈入栈(重点)

```
Status Push(SqStack &S,SElemTpe e){
    if(S.top-S.base==S.stackSize)//Stackfull
        return ERROR;
    *S.top++=e;//指针*S.top指向的是原栈顶的下一个位置
    return OK;
}
```

### 顺序栈出栈（重点）

```
Status Pop(SqStack &S,SElemType &e){
    if(S.top==S.base)//栈空
        return ERROR;
    e=*--S.top;
    return OK;
}
```

### 取顺序栈栈顶元素

```
Status GetTop(SqStack S,SElemType &e){
    if(S.top==S.base)
        return ERROR;
    e=*(S.top-1);
    return OK;
}
//e=*(S.top--)不可以，尽管自减的运算优先级高，但后自减返回的是原始值参加下一步的运算，相当于*(S.top),取得是栈顶的下一个值
//e=*(--S.top)可以取到正确的栈顶元素，但栈顶元素会被修改，若有后续操作的需求结果会受到影响
```

## 链栈

### 链栈的表示

> 智能在链表头指针位置进行操作，故没有必要加头结点，栈顶指针就是链表的头指针。

```
typedef struct StackNude{
    SElemType data;
    struct StackNode *next;
}StackNode,*LinkStack;
LinkStack S;
```

### 链栈的初始化

```
void InitStack(LinkStack &S){
    S=NULL;
}
```

### 判断链栈是否为空

```
Status StackEmpty(LinkStack S){
    if(S==NULL)
        return TRUE;
    else
        return FALSE;
}
```

### 链栈的入栈

```
Status Push(LinkStack &S,SElemType e){
    p=new StackNode;//生成新结点p
    if(!p)
        exit(OVERFLOW);
    p->data=e;
    p->next=S;
    S=p;
    return OK;
}
```

### 链栈的出栈

```
Status Pop(LinkStack &S,SElemType &e){
    if(S==NULL)
        return ERROR;
    e=S->data;
    p=S;
    S=S->next;
    delete p;
    return OK;
}
```

取链栈栈顶元素

```
SElemType GetTop(LinkStack S){
    if(S==NULL)
        exit(1);
    else
        return S→data;
}
```

# 队列的表示和操作的实现

## 顺序队列（循环队列）

队列的循环表示

```
#define M 100
typrdef struct{
    QElemType *base;
    int front;//队头指针
    int rear;//队尾指针
}SqQueue;
//空队的标志Q.front==Q.rear
//入队base[rear++]=x;
//出队x=base[front++];
```

循环队列初始化

```
Status InitQueue(SqQueue &Q){
    Q.base = new QElemType[MAXQSIZE];
    if(!Q.base)
        exit(OVERFLOW);
    Q.front=Q.rear=0;
    return OK;
}
```

### 求循环队列的长度

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

### 顺序队列入队

```
Status 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;
}
```

### 循环队列出队

```
Status DeQueue(LinkQueue &Q,QElemType &e){
    if(Q.front==Q.rear)
        return ERROR;
    e=Q.base[Q.front];
    Q.front=(Q.front+1)%MAXQSIZE;
    return OK;
}
```

## 链队列

### 链队列的表示

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

```
typedef struct{
    QueuePtr front;
    QueuePtr rear;
}LinkQueue;
```

### 链队列初始化

```
Status InitQueue(LinkQueue &Q){
    Q.front=Q.rear=new QNode;
    Q.front->next=NULL;
    return OK;
}
```

### 销毁链队列

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

### 判断链队列是否为空

```
Status QueueEmpty(LinkQueue Q){
    return (Q.front==Q.rear);
}
```

### 求链队列的队头元素

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

### 链队列入队

```
Status EnQueue(LinkQueue &Q,QElemType e){
    p = new QNode;
    p->data=e;
    p->next=NULL;
    Q.rear->next=p;
    Q.rear=p;
    return OK;
}
```

### 链队列出队

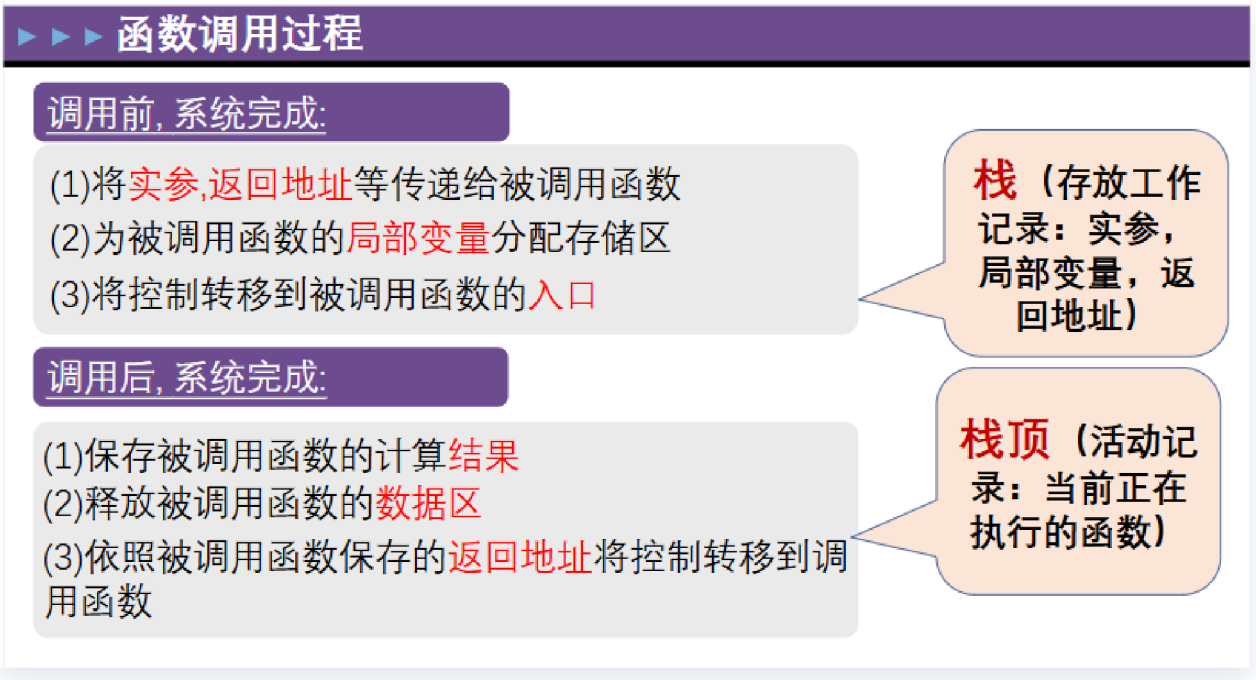
```
Status DeQueue(LinkQueue &Q,QElemType &e){
    if(Q.front==Q.rear)
        return ERROR;
    p=Q.front->next;
    e=p->data;
    Q.front->next=p->next;
    if(Q.rear==p)
        Q.rear=Q.front;
    delete p;
    return OK;
}
```

## 栈与递归（重点）

> 递归：若存在一个函数、过程或者数据结构的定义，在内部直接或简洁出现定义本身的应用，则称它们是递归的，或是递归定义的。

```
//exzample
long Fact (long n){//n的阶乘递归函数
    if(n==0)
        return 1;
    else return
        n*Fact(n-1);
}
```

**栈与递归关系：在程序设计语言编译系统内部用应栈实现递归。**




## 用到递归的情况

- 递归定义数学函数
- 具有递归特性的数据结构
- 可递归求解的问题
  - `hanoi` 塔问题
  - 迷宫问题

# 分治法

分治法求解递归问题

分治法：对于一个较为复杂的问题，发呢结成几个相对简单且揭发相同或类似的子问题来求解。

条件：

1. 将问题转变成一个新问题，新问题与原问题，解法相同或类似，仅是处理对象的不同，且处理对象有变化的规律，从而实现问题简化。
2. 必须要有一个明确的**递归出口**，或称**递归地边界**。

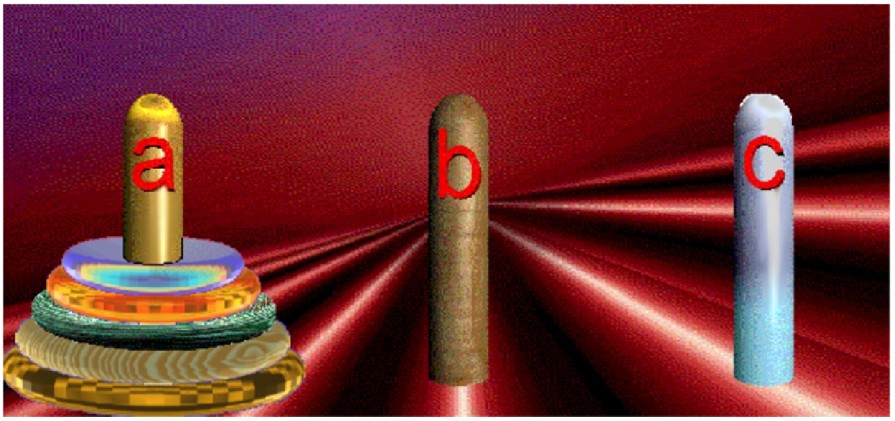
```
//NORMAL
void p(参数表){
    if(递归结束的条件)
        可直接求解步骤;//----基本项
    else
        p(较小的参数);//----归纳项
}
```

```
//EXZAMPLE
long Fact(long n){//n的阶乘递归函数
    if(n==0)
        return 1;//基本项
    else
        return n*Fact(n-1);//归纳项
}
//时间复杂度O(n)
//空间复杂度O(n)
```

# Hanoi汉诺塔问题

## Hanoi塔问题

汉诺塔




在印度圣庙里，一块黄铜板上插着三根宝石针。
主神梵天在创造世界时，在其中一根针上穿好了由大到小的64片金片，这就是汉诺塔。
任务：要求把所有的金片都移到另外一个针上时，一次只移动一片，小片必在大片上面。

```
#include<iostream.h>
using namespace std;

void move(char x,char z){
    cout<<x<<'→'<<z<<endl;
}

void Hanoi(int n,char A,char B,char C){
    if(n==1)
        move(A,C);
    else{
        Hanoi(n-1,A,B,C);
        move(A,C);
        Hanoi(n-1,B,A,C);
    }
}

void main(){
    Hanoi(3,'a','b','c');
}
/*
```

```
OUTPUT:
a→c
a→b
c→b
a→c
b→a
b→c
a→c
实现了从a柱到b柱的迁移
*/
```

> 空间复杂度：O(n)——与递归树的深度成正比
>
> 时间复杂度：$O(2^n)$——与递归树的调用函数结点数成正比

## 递归的优缺点

1. 优点：结构清晰，程序易读。
2. 缺点：
   - 每次调用要生成工作记录，保存状态信息，入栈。
   - 返回时要出栈，恢复状态信息。
   - 时间空间开销可能变大。