第3章 栈与队列

特殊线性表或
受限制的线性表

栈和队列

本章重点与难点

- **重点：**

  (1)栈、队列的定义、特点、性质和应用；

  (2)ADT栈、ADT队列的设计和实现以及基本操作及相关算法。

- **难点：**

  (1)循环队列中对边界条件的处理；

  (2)利用栈和队列解决实际问题的应用水平。

第三章 栈和队列

3.1 栈

3.1.1 抽象数据类型栈的定义

3.1.2 栈的表示和实现

3.2 栈的应用举例

3.2.1 数制转换

3.2.2 括号匹配的检验

3.2.3 行编辑程序问题

3.2.4 表达式求值

3.3 栈与递归的实现

3.4 队列

3.1.1 抽象数据类型栈的定义

• 线性表

线性表是具有相同数据类型的n（n≥0）个数据元素的有限 序列，其中n为表长，当n = 0时线性表是一个空表。若用L命名线性表，则其一般表示为

$$L = (a_1, a_2, \dots, a_i, a_{i+1}, \dots, a_n)$$

• 栈的定义

- 栈(Stack)是一种特殊的线性表，其插入和删除操作均在表的一端进行，是一种运算受限的线性表。

3.1.1 抽象数据类型栈的定义

- 栈的定义
  - 栈(Stack)是一种特殊的线性表，其插入和删除操作均在表的一端进行，是一种运算受限的线性表。

3.1.1 抽象数据类型栈的定义

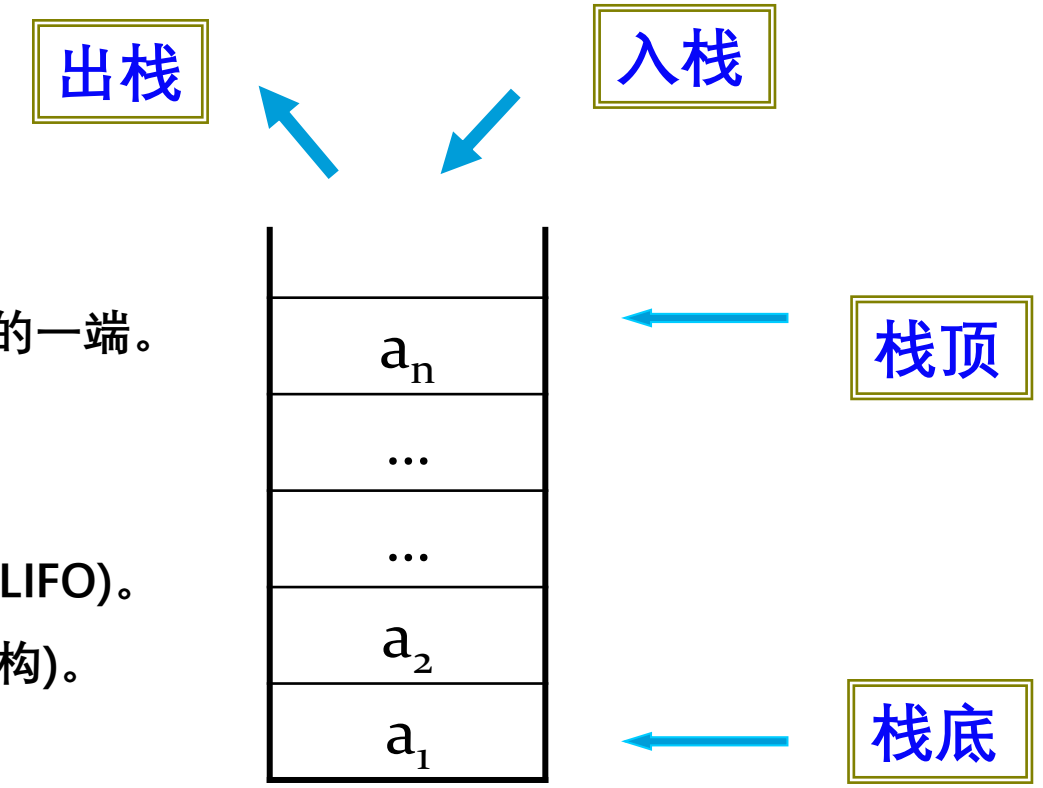
栈的相关术语

栈顶(top)是栈中允许插入和删除的一端。

栈底(bottom)是栈顶的另一端。

后进先出(Last In First Out，简称LIFO)。

又称栈为后进先出表(简称LIFO结构)。



图例

3.1.1 抽象数据类型栈的定义

• 抽象数据类型栈

ADT Stack {

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a_n 端为栈顶，a_1 端为栈底。

基本操作： 见下页

} ADT Stack

3.1.1 抽象数据类型栈的定义

- 栈的基本操作

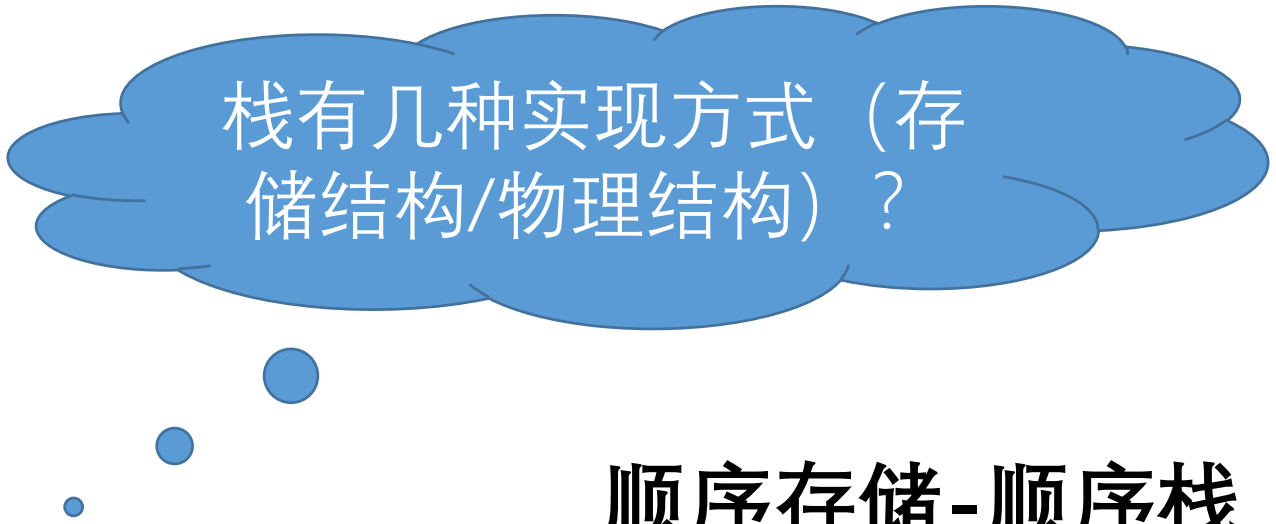
```
InitStack(&S)          //初始化栈
DestroyStack(&S)       //销毁栈

Push(&S, e)            //入栈
Pop(&S, &e)            //出栈
GetTop(S, &e)          //取栈顶元素

StackEmpty(S)          //判栈空
StackLength(S)         //求栈长度
```

3.1.2 栈的表示和实现

- 思考：



顺序存储-顺序栈

链式存储-链栈

3.1.2 栈的表示和实现

- Recall线性表的顺序存储与实现

```
顺序表的类型定义

#define MAXSIZE 100    //  数组容量

Typedef struct

{  DataType  data[MAXSIZE];  //  数组域

    int  last;              //  线性表长域

} Seqlist;                  //  结构体类型名
```

3.1.2 栈的表示和实现

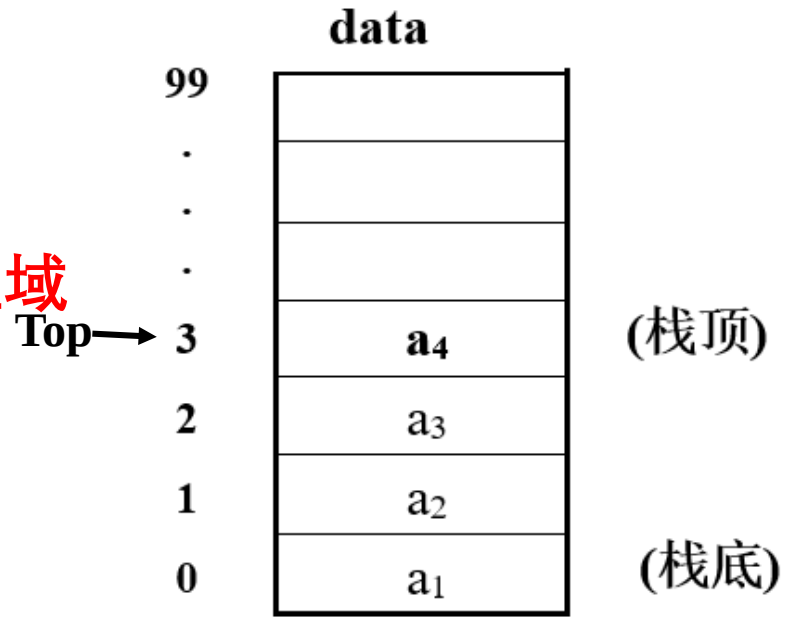
• 顺序栈的C语言实现（静态分配）

```
define MAXSIZE 100    //  数组容量

Typedef struct

{  DataType  data[MAXSIZE];  //  数组域
    int  top ;//栈顶指针

} SqStack;              //  结构体类型名
```



3.1.2 栈的表示和实现

(1)置空栈

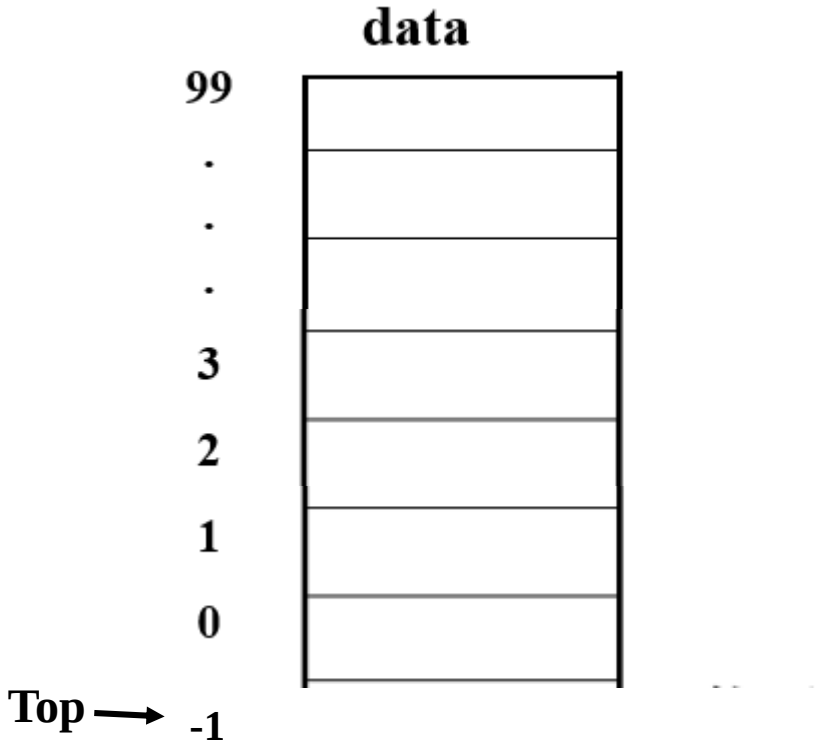
```
void InitStack(SeqStack *s)
{s->top=-1;}
```

(2)判栈空

```
int StackEmpty(SeqStack *s)
{return s->top==-1;}
```

(3)判栈满

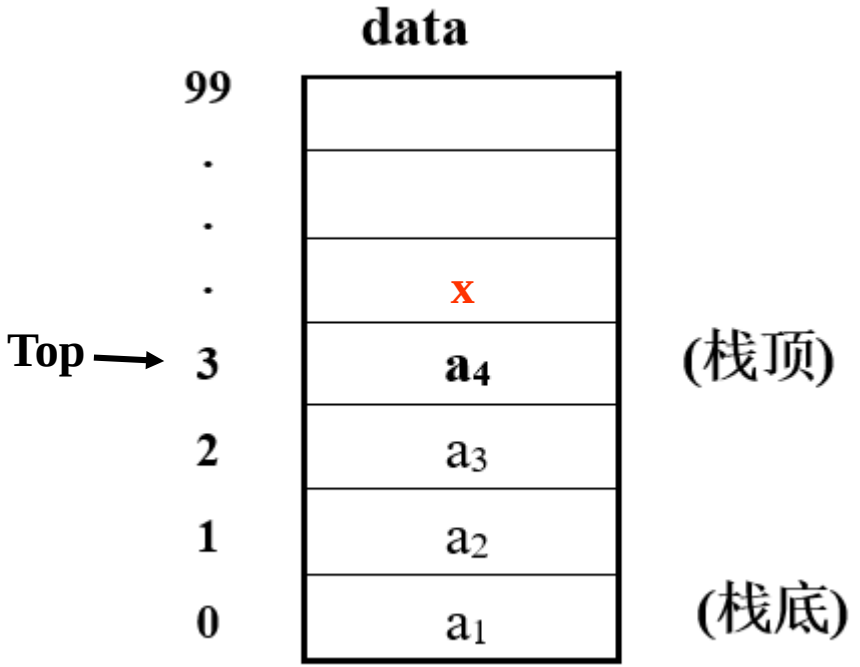
```
int StackFull(SeqStack *s)
{return s->top==MAXSIZE-1;}
```



3.1.2 栈的表示和实现

4) 进栈操作

```
void Push(SeqStack *s,DataType x )
{if (StackFull(s)   Error(“ overflow” );
  s->top ++;
  s->data[s->top]=x; }
}
```



进栈操作：栈不满时，栈顶指针先加1，再送值到栈顶元素。

3.1.2 栈的表示和实现

(1)置空栈

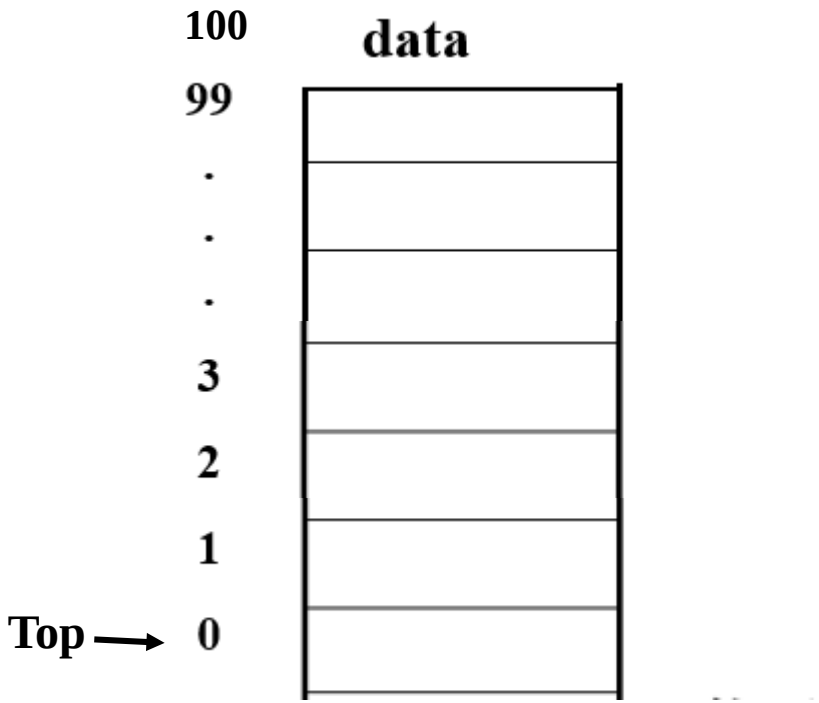
```
void InitStack(SeqStack *s)
{s->top=0;}
```

(2)判栈空

```
int StackEmpty(SeqStack *s)
{return s->top==0;}
```

(3)判栈满

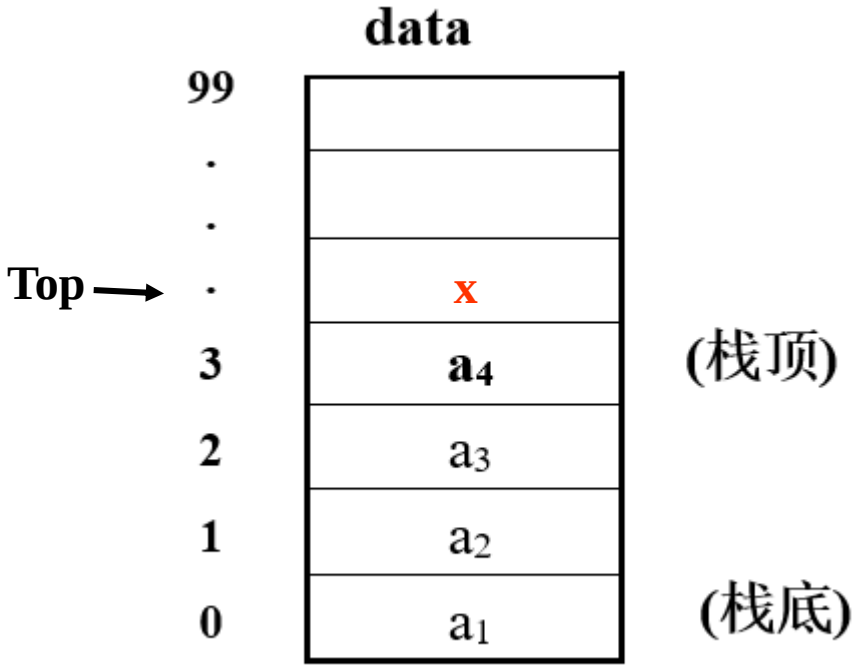
```
int StackFull(SeqStack *s)
{return s->top==MAXSIZE;}
```



3.1.2 栈的表示和实现

4) 进栈操作

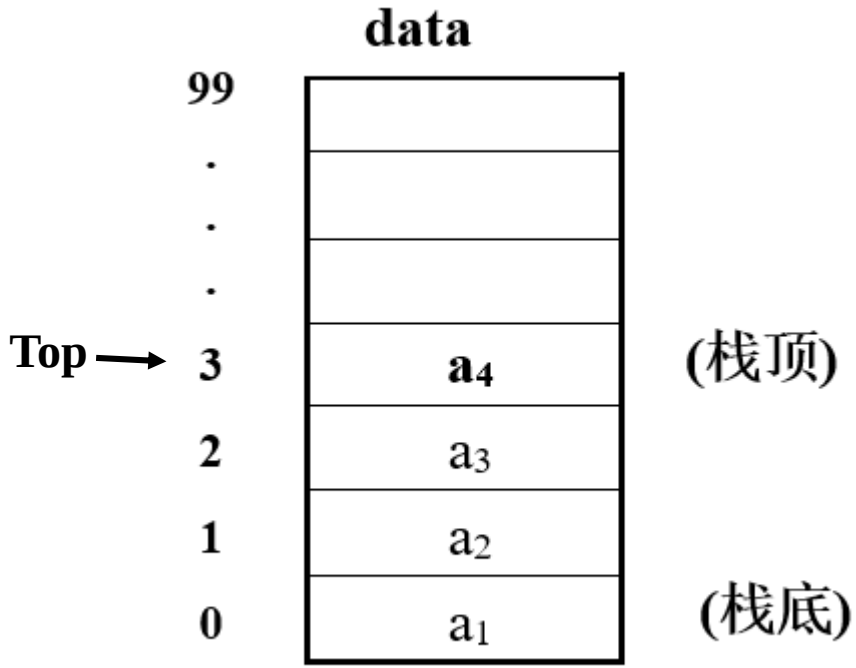
```
void Push(SeqStack *s,DataType x )
{if (StackFull(s)   Error(“ overflow” );
   s->data[s->top]=x;
   s->top ++;
}
```



3.1.2 栈的表示和实现

5）出栈操作

```
DataType  Pop(SeqStack  *s)
   {if (Stackempty(s))
Error("underflow");
   DataType x =s->data[s->top] ;   top --;
   return x;//返回当前栈顶元素
}
```



出栈操作：栈非空时，先取栈顶元素值，再将栈顶指针减1.

3.1.2 栈的表示和实现

- 顺序栈的C语言实现（动态分配）

```
//----- 栈的顺序存储表示 -----
#define  STACK_INIT_SIZE  100; //栈容量
#define  STACKINCREMENT   10;  //栈增量
typedef struct {
  ElemType  *base;    //基地址
  ElemType  *top;     //栈顶指针
  int  stacksize;           //栈容量
} SqStack;
```

3.1.2 栈的表示和实现

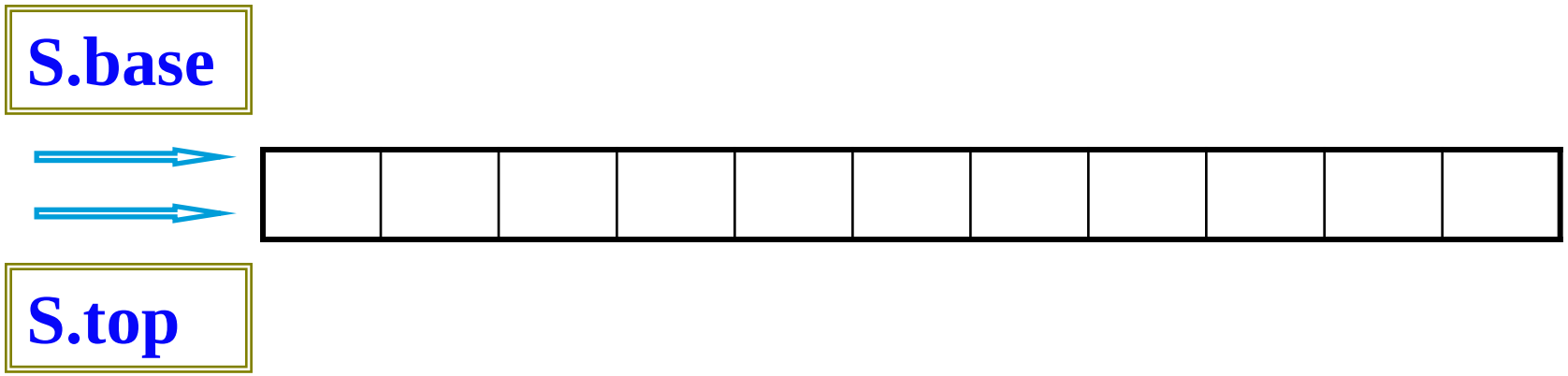
- 栈初始化过程演示

```
#define  STACK_INIT_SIZE  100; //栈容量
 #define  STACKINCREMENT   10;  //栈增量
 typedef struct {
   ElemType  *base;    //基地址
   ElemType  *top;     //栈顶指针
   int  stacksize;          //栈容量
 } SqStack;
```

(1) 给栈S申请栈空间

(2) 设置基地址S.base和栈顶地址S.top

(3) 设置栈容量S.stacksize=STACK_INIT_SIZE



3.1.2 栈的表示和实现

- 栈初始化算法

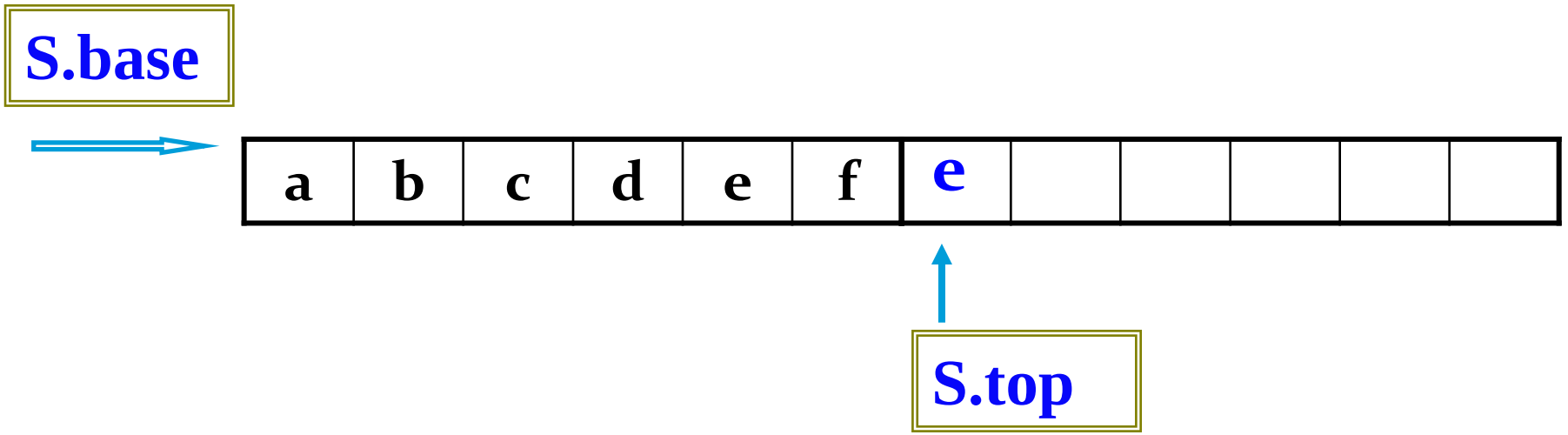
```
Status InitStack (SqStack &S) // 构造一个空栈S
{
  S.base=(ElemType*)malloc(STACK_INIT_SIZE*
                           sizeof(ElemType));
  if (!S.base) exit (OVERFLOW); //存储分配失败
  S.top = S.base;
  S.stacksize = STACK_INIT_SIZE;
  return OK;
}
```

3.1.2 栈的表示和实现

- 栈满情况下入栈操作演示

(1) 如果栈满，给栈增加容量

(2) 将数据存入栈顶位置，栈顶后移一位



3.1.2 栈的表示和实现

• 入栈操作演示

```
Status Push (SqStack &S, SElemType e) {
  if (S.top - S.base >= S.stacksize) //栈满，追加存储空间
    { S.base = (ElemType *) realloc ( S.base,
       (S.stacksize + STACKINCREMENT) *
                                    sizeof (ElemType));
      if (!S.base) exit (OVERFLOW); //存储分配失败
      S.top = S.base + S.stacksize;
      S.stacksize += STACKINCREMENT;
    }
  S.top = e; S.top++; //先传数据再移动指针
return OK;
}
```

3.1.2 栈的表示和实现

顺序栈小结：

- 出栈和入栈都只能在栈顶进行
- 设立栈顶指针top，既可以指向栈顶元素的位置，也可以指向栈顶元素的下一个位置，但是在实现具体操作的时候要保持一致
- 创建动态分配的顺序栈

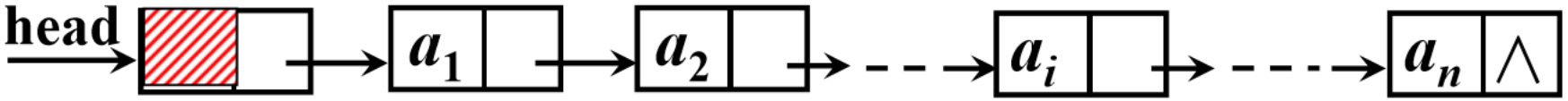
3.1.2 栈的表示和实现

- 栈的指针实现---链栈

✓栈的链式存储结构及实现

■链栈：栈的链接存储结构

■如何改造链表实现栈的链接存储?



■将哪一端作为栈顶?

■将链表首端作为栈顶，方便操作。

■链栈需要加头结点吗?

■通常采用单链表实现，并规定所有操作都是在单链表的表头进行的。如果链栈没有头结点，head指向栈顶元素。

3.1.2 栈的表示和实现

- 讨论链栈基本操作的实现

InitStack(&S) //初始化栈

DestroyStack(&S) //销毁栈

Push(&S, e) //入栈

Pop(&S, &e) //出栈

GetTop(S, &e) //取栈顶元素

StackEmpty(S) //判栈空

StackLength(S) //求栈长度

第三章 栈和队列

3.2.1 数制转换

- **数制转换**----是计算机实现计算的基本问题。

例 将10进制1348转换成8进制

图例

✓方法：除留余数法。 $(1348)_{10}=(2504)_{8}$

计算顺序

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

输出顺序

输出的时候是从高位到低位进行，恰好和计算相反。

转换的过程满足栈的后进先出的特点

3.2.1 数制转换

- 10进制数N转换成8进制的算法

```
void conversion () {
   InitStack(S); scanf (“%d”,N);//输入一个任意非负十进制整数
   while (N) {
     Push(S, N % 8); N = N/8;        //余数入栈
   }
   while (!StackEmpty(S)) {
     Pop(S,e);//若栈不空，则删除S的栈顶元素，用e返回其值
     printf ( "%d", e );
   }
} // conversion
```

3.2.2 括号匹配的检验

- **问题描述**
  - 一个表达式中，包含三种括号“(”和“)”，“[”和“]”和“{”和“}”，这三种括号可以按任意的合法次序使用。设计算法检验表达式中所使用括号的合法性。

例在表达式中正确的格式：

（[]（））或[（[] []）]

例在表达式中不正确的格式

- [（]）----左右不匹配
- （[（）]---左括号多了
- [（）]）----右括号多了

3.2.1 括号匹配的检验

- 算法过程

[([ ] [ ])]

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。

3.2.1 括号匹配的检验

- 括号匹配检验算法

```
Status  check( ) {
   char ch;  InitStack(S);
   while ((ch=getchar())!='#') {
      switch (ch) {
         case (ch=='('||ch=='['||ch=='{'):Push(S,ch);break;
         case (ch== ')'):
            if (StackEmpty(S)) return FALSE;
            else
               {Pop(S,e);if(e!= '(') return FALSE;}
            break;
```

3.2.1 括号匹配的检验

- 括号匹配检验算法

```
case (ch== ‘]'):
  if (StackEmpty(S)) retrun FALSE;
  else
  {Pop(S,e);if(e!= ‘[') return FALSE;}
   break;…….
 default:break;
}
}
if (StackEmpty(S))  return TRUE;
else return FALSE;
}
```

3.2.3 行编辑程序问题

- 问题描述
  - 在用户输入一行字符的过程中，允许用户输入出差错，并在发现有误时可以及时更正。

- 解决方法

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（当用户发现刚刚键入的一个字符是错的，可补进一个退格符，以表示前一个字符无效），“@”为退行符（当用户发现当前键入的行内差错较多或难以补救时）。

3.2.3 行编辑程序问题

例 假设从终端接受了这样两行字符：

```
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 行编辑程序问题

- 行编辑问题算法

```
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);
    ch=getchar();
    while (ch != EOF) { //EOF为全文结束符
      while (ch != '\n') {……}
     …… }

    DestroyStack(S);
}
```

3.2.3 行编辑程序问题

- 行编辑问题算法

假设从终端接受了这样两行字符：

whli##ilr#e （s#*s)

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

```
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
```

3.2.3 行编辑程序问题

- 行编辑问题算法

```
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);
    ch=getchar();
    while (ch != EOF) { //EOF为全文结束符
       while (ch != '\n') {……}
      将从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
       ch = getchar();
     }
    DestroyStack(S);
}
```

3.2.4 表达式求值

例 求表达式3*(2+3*5)+6的值

四则运算规则：

（1）先乘除，后加减；

（2）从左算到右；

（3）先括号内，后括号外。

- 由三个部分组成：操作数、运算符、界限符（界限符是必不可少的，反映了计算的先后顺序）

- 不使用界限符可以无歧义的表达运算顺序么？

表达式：
- 前缀表达式（波兰式）
- 中缀表达式
- 后缀表达式（逆波兰式）

3.2.4 表达式求值

规则运算符在两个操作数中间

规则：运算符在两个操作数后面

规则：运算符在两个操作数前面

中缀表达式	后缀表达式	前缀表达式
a+b	ab+	+ab
a+b-c	a b+c-	-+abc
a+b-c* d	ab+cd* -	-+ab* cd

3.2.4 表达式求值

- 高级语言中，采用类似自然语言的中缀表达式，但计算机对中缀表达式的处理是很困难的，而对后缀或前缀表达式则显得非常简单。

后缀表达式的特点：

① 在后缀表达式中，变量（操作数）出现的顺序与中缀表达式顺序相同。

② 后缀表达式中不需要括弧（界限符）定义计算顺序，而由运算（操作符）的位置来确定运算顺序。

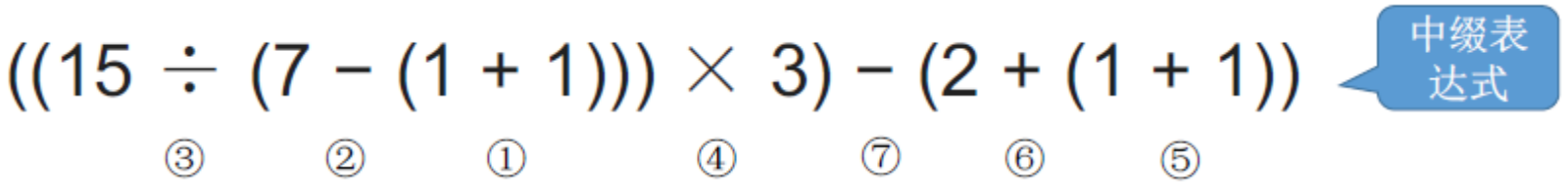
3.2.4 表达式求值

中缀转后缀的手算方法：

1、确定中缀表达式中各个运算符的运算顺序

2、选择下一个运算符，按照「左操作数 右操作数 运算符」的方式组合成一个新的操作数

3、如果还有运算符没被处理，就继续



3.2.4 表达式求值

中缀转后缀的手算方法：

1、确定中缀表达式中各个运算符的运算顺序

运算顺序不唯一，因此对应的 后缀表达式也不唯一

2、选择下一个运算符，按照「左操作数 右操作数 运算符」的方式组合成一个新的操作数

3、如果还有运算符没被处理，就继续

A+ B * (C - D) - E / F

A+ B * (C - D) - E / F

AB CD - * + EF / -

ABCD-*EF/-+

“左优先”：只要左边的运算符能先计算，就优先算左边的

可保证运算顺序唯一；

后缀表达式：运算符在式中出现的顺序恰好为表达式的运算顺序

前缀表达式不具有

3.2.4 表达式求值

A+ B * (C - D) - E / F

A B C D - * + E F / -

中缀转后缀的机算方法：

初始化一个栈，用于保存暂时还不能确定运算顺序的运算符。

从左到右处理各个元素，直到末尾。可能遇到三种情况：

① 遇到操作数。直接加入后缀表达式。

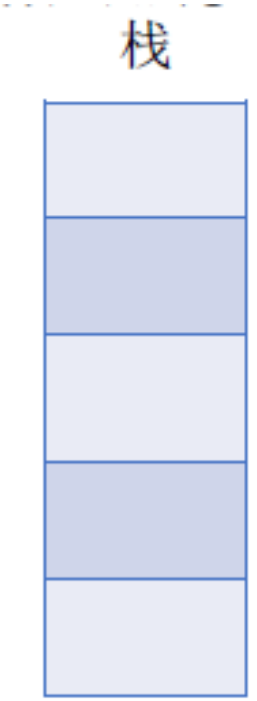
② 遇到运算符。考虑当前栈的情况：

a.栈空或者栈顶元素为“（”，运算符直接入栈；

b.栈顶元素为运算符：依次弹出栈中优先级高于或等于当前运算符的所有运算符，并加入后缀表达式，若碰到“(”或栈空则停止。之后再把当前运算符入栈。

③遇到界限符。遇到“(”直接入栈；遇到“)”则依次弹出栈内运算符并加入后缀表达式，直到弹出“(”为止。注意：“(”不加入后缀表达式。

按上述方法处理完所有字符后，将栈中剩余运算符依次弹出，并加入后缀表达式。



3.2.4 表达式求值

后缀表达式的计算方法（手算）：

- 从左往右扫描，每遇到一个运算符，就让运算符前面最近的两个操作数执行对应运算，合体为一个操作数 （注意两个操作数的左右顺序）

AB CD -* + EF/ -

A+ B * (C - D) - E / F

填空题 1分

设置

15 7 1 1 + - ÷ 3 × 2 1 1 + + - =

[填空1]

作答

3.2.4 表达式求值

后缀表达式的计算方法（手算）：

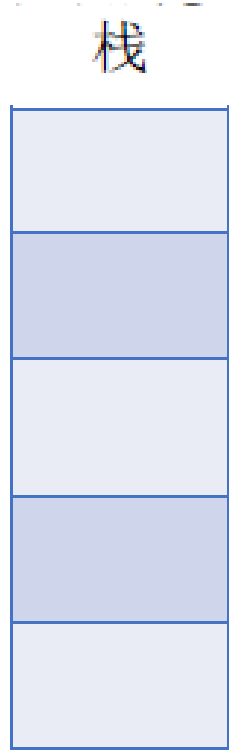
- 从左往右扫描，每遇到一个运算符，就让运算符前面最近的两个操作数执行对应运算，合体为一个操作数 （注意两个操作数的左右顺序）

AB CD -* + E F/ -

A+ B * (C - D) - E / F

后缀表达式的计算方法（机算）：

- ①从左往右扫描下一个元素，直到处理完所有元素
- ②若扫描到操作数则压入栈，并回到①；否则执行③
- ③若扫描到运算符，则弹出两个栈顶元素，执行相应运算，运算结果压回栈顶，回到①
- 表达式结束，栈中唯一元素即为表达式的值。



第三章 栈和队列

3.3 栈与递归的实现

- **递归调用的定义：**
  - 子程序（或函数）直接调用自己或通过一系列调用语句间接调用自己。是一种描述问题和解决问题的基本方法。
- **递归的基本思想：**
  - 把一个不能或不好求解的大问题转化为一个或几个小问题，再把这些小问题进一步分解成更小的小问题，直至每个小问题都可以直接求解。
- **递归的要素：**
  - 递归边界条件：确定递归到何时终止，也称为递归出口；
  - 递归模式：大问题是如何分解为小问题的，也称为递归体

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

3.3 栈与递归的实现

- 什么问题可以应用递归：
  - 一类是递归函数；
  - 第二类是，有的数据结构，如二叉树，广义表等，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；
  - 还有一类问题，虽然问题本身没有明显的递归结构，但用递归求解比迭代求解更简单，如八皇后问题、Hanoi塔问题等。

3.3 栈与递归的实现

- **函数调用：**
  - 当在一个函数的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；
    - 将控制转移到被调函数的入口。

3.3 栈与递归的实现

- **函数调用的背后过程：**

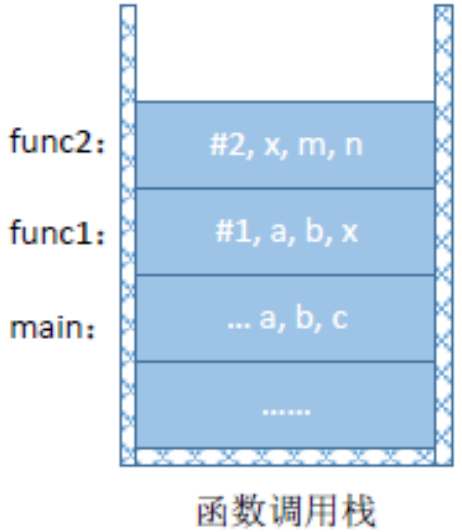
```
void main() {
  int a, b, c;
  ...
  func1 (a, b);
#1 c=a+b;
  ...
}
```

```
void func1 (int a, int b) {
  int x;
  ...
  func2 (x);
#2 x=x+10086;
  ...
}
```

```
void func2 (int x) {
  int m, n;
  ...
}
```

函数调用的特点：最后被调用的函数最先执行结束（LIFO）

函数调用时，需要用一个栈存储：
① 调用返回地址
② 实参
③ 局部变量



函数调用栈

3.3 栈与递归的实现

- **函数调用：**
  - 从被调用函数返回调用函数之前，应该完成：
    - 保持被调用函数的计算结果；
    - 释放被调用函数的数据区；
    - 依照被调用函数保存的返回地址将控制转移到调用函数。

3.3 栈与递归的实现

- **函数调用的背后过程：**

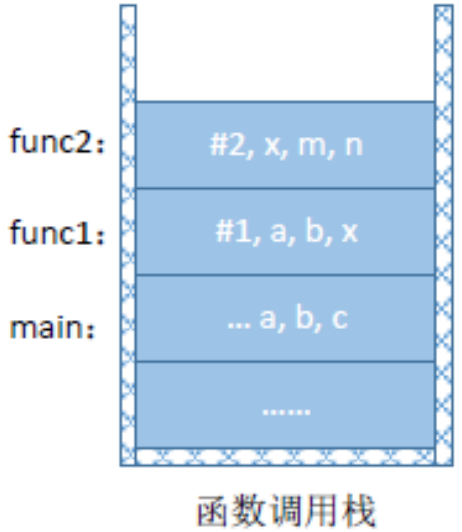
```
void main() {
  int a, b, c;
  ...
  func1 (a, b);
#1 c=a+b;
  ...
}
```

```
void func1 (int a, int b) {
  int x;
  ...
  func2 (x);
#2 x=x+10086;
  ...
}
```

```
void func2 (int x) {
  int m, n;
  ...
}
```

函数调用的特点：最后被调用的函数最先执行结束（LIFO）

函数返回时：
① 返回函数计算结果（若有）
② 弹出相应函数信息
③ 根据保存地址转移到调用函数



函数调用栈

3.3 栈与递归的实现

- **函数调用：**
  - 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”

3.3 栈与递归的实现

【例】求阶乘的函数

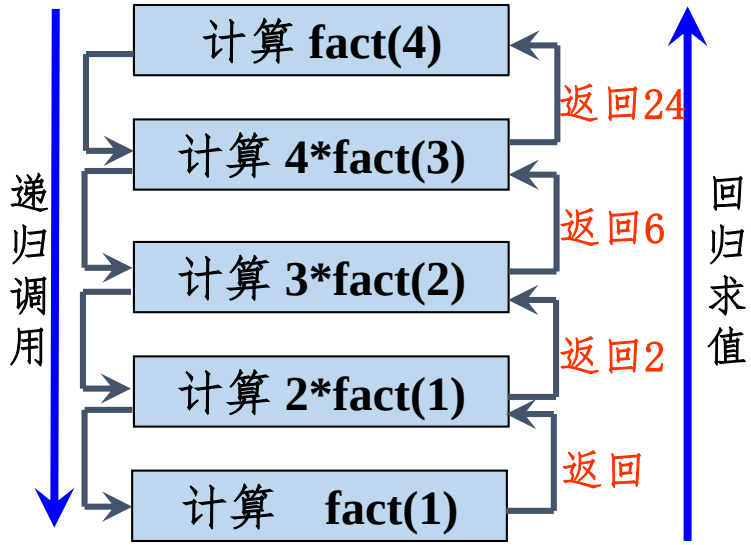
$$n!=\begin{cases}1 & \text{当}n=0\text{或}n=1\\ n*(n-1)! & \text{当}n>1\end{cases}$$

(1) 非递归算法

```
long fact( int n ）
{
    long f=2;    int i;
    for( i=2; i <= n; i++ )
          f = f*i;
    return(f);
}
```

(2) 递归算法

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



T(n)=o(n)

T(n)= ？

3.3 栈与递归的实现

```
182  //计算正整数 n!
183  int factorial (int n){
184      if (n==0 || n==1)
185          return 1;
186      else
187          return n*factorial(n-1);
188  }
189
190  int main() {
191      //... 其他代码
192      int x=factorial(10);
193      printf("奥利给! ");
194  }
```

再次思考：递归算法的空间复杂度

函数调用栈

递归调用时，函数调用栈可称为“递归工作栈”
每进入一层递归，就将递归调用所需信息压入栈顶
每退出一层递归，就从栈顶弹出相应信息

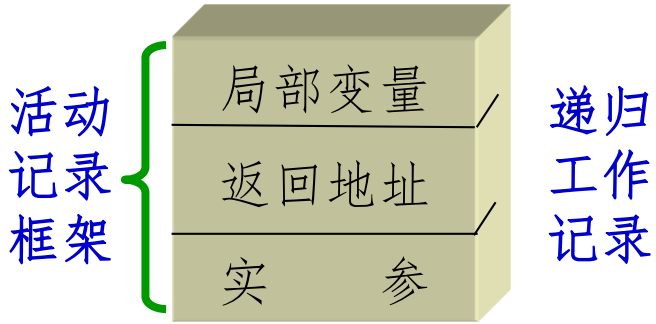
缺点：太多层递归可能会导致栈溢出

3.3 栈与递归的实现

递归过程与递归工作记录

①每一次递归调用时，需要为过程中使用的参数、局部变量和返回地址等另外分配存储空间；

②每层递归调用需分配的空间形成递归工作记录，按栈结构组织，即LIFO。



递归函数的内部执行过程

①建栈：运行开始时，首先为递归调用建立一个工作栈，其结构包括值参、局部变量和返回地址；

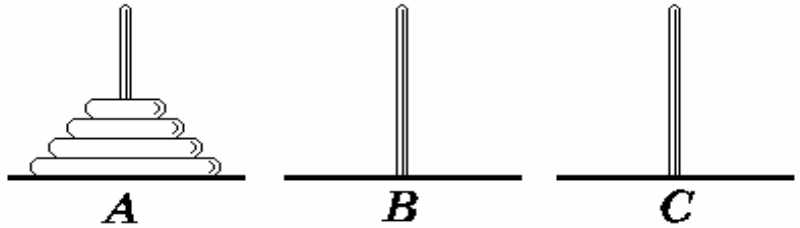
②压栈：每次执行递归调用之前，把递归函数的值参和局部变量的当前值以及调用后的返回地址压栈；

③出栈：每次递归调用结束后，将栈顶元素出栈，使相应的值参和局部变量恢复为调用前的值，然后转向返回地址指定的位置继续执行。

3.3 栈与递归的实现

【例2-13】汉诺塔问题：递归的经典问题

在世界刚被创建的时候有一座钻石宝塔（塔A），其上有64个金碟。所有碟子按从大到小的次序从塔底堆放至塔顶。紧挨着这座塔有另外两个钻石宝塔（塔B和塔C）。从世界创始之日起，婆罗门的牧师们就一直在试图把塔A上的碟子移动到塔C上去，其间借助于塔B的帮助。每次只能移动一个碟子，任何时候都不能把一个碟子放在比它小的碟子上面。当牧师们完成任务时，世界末日也就到了。



汉诺塔问题的递归求解:

如果 n = 1，则将这一个盘子直接从 塔A移到塔 C 上，

否则，执行以下三步：

① 将塔A上的n-1个碟子借助塔C先移到塔B上；

② 把塔A上剩下的一个碟子移到塔C上；

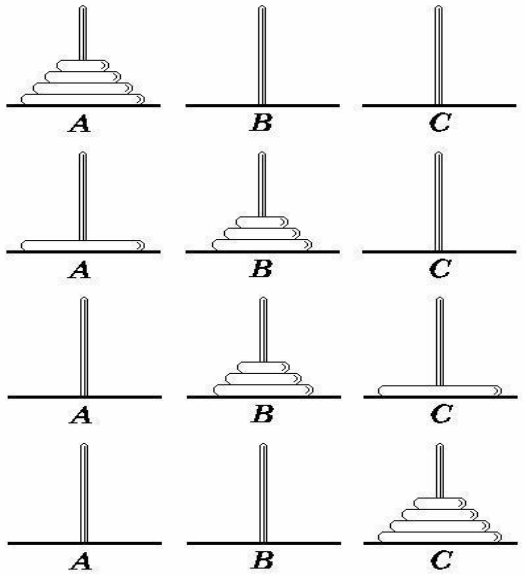
③ 将n-1个碟子从塔B借助于塔A移到塔C上。

3.3 栈与递归的实现

设：

函数：Hanoi(int n, char A, char B, char C);
功能：将n个盘子从塔A移到塔C,塔B为临时；

函数：Move(A, C);
功能：将塔A上的1个盘子直接移到塔C；



汉诺塔问题的递归求解:
如 n = 1,则将这1个盘子直接从塔A移到塔 C 上。
Move(A, C);
否则，执行以下三步：
(1)将n-1个盘子从塔A借助于塔C移到塔B上；
Hanoi(n-1, A, C, B);
(2)把塔A上剩下的一个碟子移到塔C上；
Move(A, C);
(3)将n-1个碟子从塔B借助于塔A移到塔C上。
Hanoi(n-1, B, A, C);

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

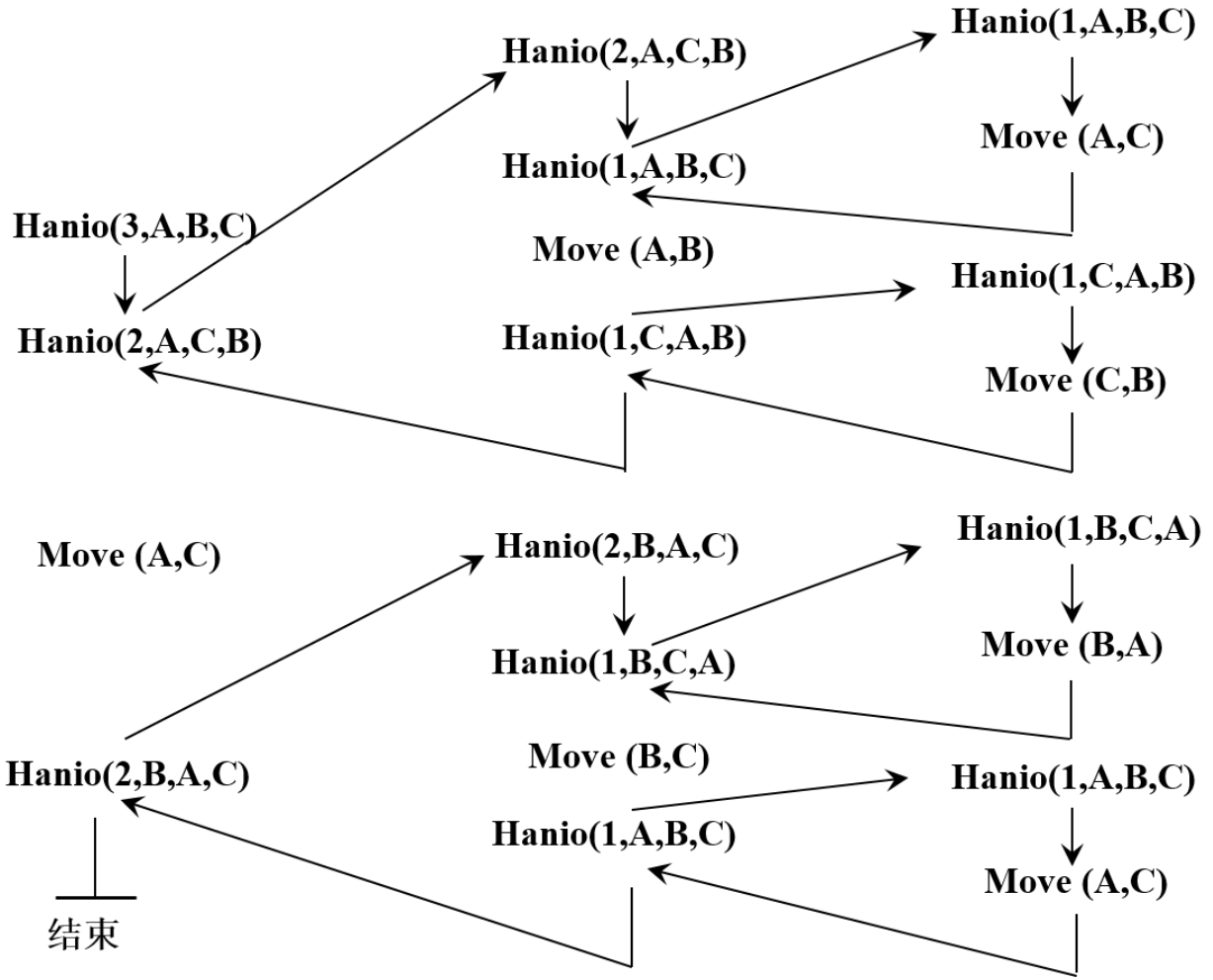
3.3 栈与递归的实现

递归函数的运行轨迹

①写出函数当前调用层执行的各语句，并用有向弧表示语句的执行次序；

②对函数的每个递归调用，写出对应的函数调用，从调用处画一条有向弧指向被调用函数入口，表示调用路线，从被调用函数末尾处画一条有向弧指向调用语句的下面，表示返回路线；

③在返回路线上标出本层调用所得的函数值。



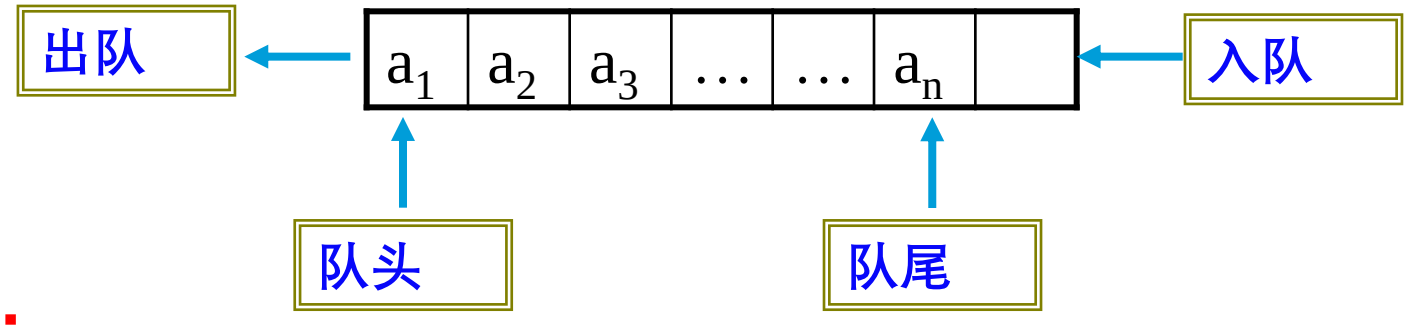
第三章 栈和队列

3.4 队列

- **队列的定义：**
  - 队列(Queue)——是一种运算受限的特殊的线性表，它只允许在表的一端进行插入，而在表的另一端进行删除。

3.4 队列

- **队列的定义：**
  - 队列(Queue)——是一种运算受限的特殊的线性表，它只允许在表的一端进行插入，而在表的另一端进行删除。
- **队列的术语：**
  - 队头(front)是队列中允许删除的一端。
  - 队尾(rear)是队列中允许插入的一端。
- **队列示意图：**



- **队列的特点：**
  - 先进先出(First In First Out ，简称FIFO)。又称队列为先进先出表。

3.4.1 队列的基本操作

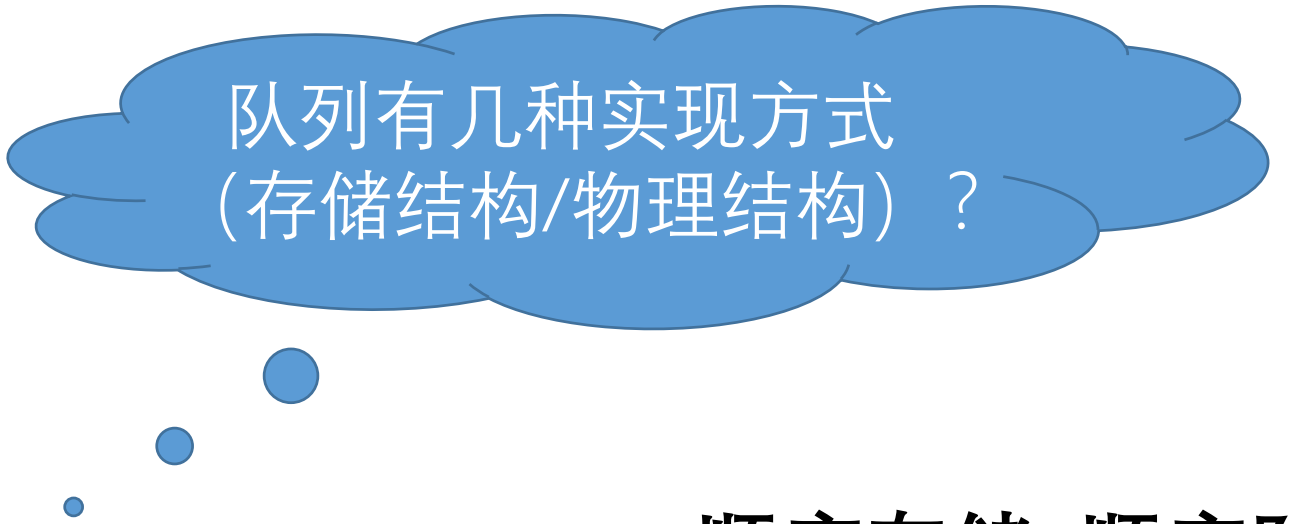
InitQueue(&Q) //初始化队列
DestroyQueue(&Q) //销毁队列

EnQueue(&Q, e) //入队列
DeQueue(&Q, &e) //出队列
GetHead(Q, &e) //取队头元素

QueueEmpty(Q) //判队列是否空
QueueLength(Q) //求队列长度

3.1.2 队列的存储结构

- 思考：



顺序存储-顺序队列

链式存储-链队列

3.4.3 队列的顺序表示及实现

- 顺序队列数据类型实现：

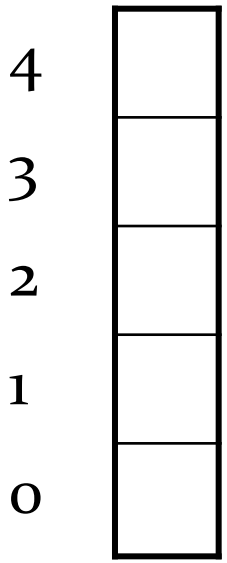
```
#define MAXQSIZE  10  //最大队列长度
  typedef struct {
    ElemType  base[MAXQSIZE];  //初始化分配存储空间
    int  front;     // 头指针
    int  rear;     // 尾指针
  } SqQueue;
```

我们约定初始化建空队列时，令front=rear=0。每当插入新的队列尾元素时，尾指针增加1，每当删除队列头元素时，头指针增加1

3.4.3 队列的顺序表示及实现

- 顺序队列讨论：

我们约定初始化建空队列时，令front=rear=0



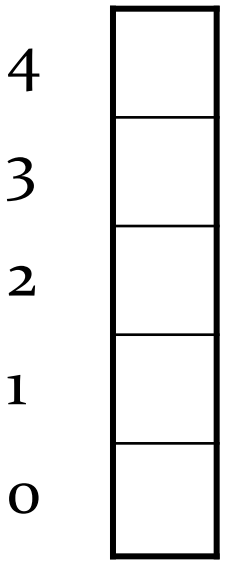
Sq.rear=0
Sq.front=0

```
Status InitQueue (SqQueue &Q) {// 构造一个空队列Q
    Q.front = Q.rear = 0;
    return OK;
}
bool QueueEmpty (SqQueue &Q) {// 判断队列是否为空
if  (Q.front == Q.rear)
   return true
else
    return false;
}
```

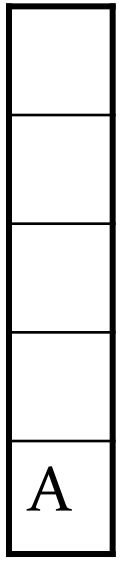
3.4.3 队列的顺序表示及实现

- 顺序队列讨论：

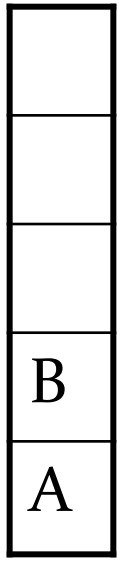
我们约定每当插入新的队列尾元素时，尾指针增加1，尾指针始终指向队列尾元素的下一个位置。



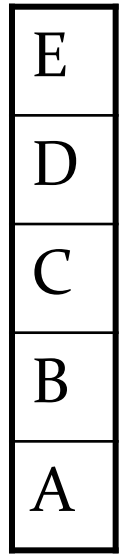
Sq.rear=0
Sq.front=0



Sq.rear=1
Sq.front=0



Sq.rear=2
Sq.front=0

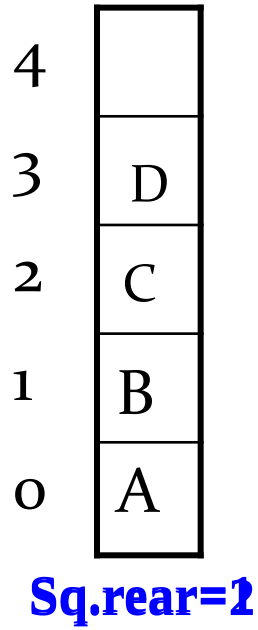


Sq.rear=5
Sq.front=0

3.4.3 队列的顺序表示及实现

- 入队列算法：

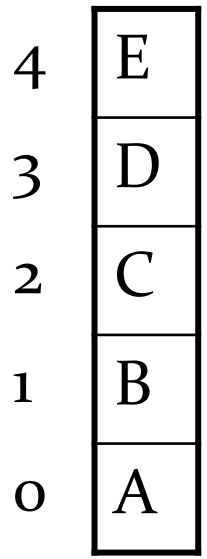
```
Status EnQueue (SqQueue &Q, ElemType e) {
                        // 插入元素e为Q的新的队尾元素
    if (队列满)
        return ERROR;
    Q.base[Q.rear] = e;
    Q.rear = Q.rear+1;
    return OK;
}
```



3.4.3 队列的顺序表示及实现

- 入队列算法：

```
Status EnQueue (SqQueue &Q, ElemType e) {
                        // 插入元素e为Q的新的队尾元素
    if (队列满)  Q.rear == MAXQSIZE?
        return ERROR;
    Q.base[Q.rear] = e;
    Q.rear = Q.rear+1;
    return OK;
}
```

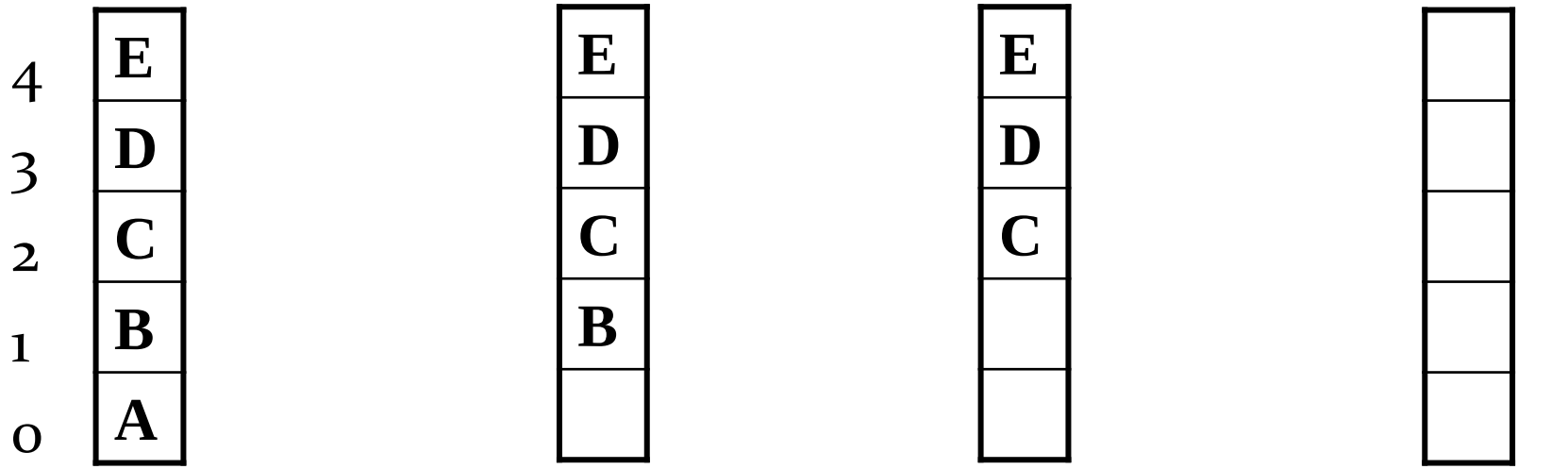


Sq.rear=5

3.4.3 队列的顺序表示及实现

• 顺序队列讨论：

我们约定每当删除队列头元素时，头指针增加1.因此，在非空队列中，头指针始终指向队列头元素。

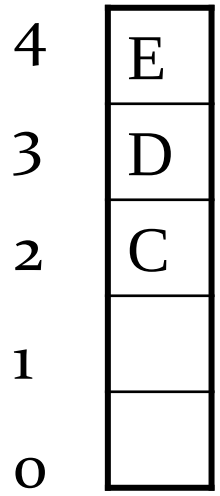


当元素被插入到数组中下标最大的位置上之后，队列的空间就用尽了，但此时数组的低端还有空闲空间，这种现象叫做假溢出。

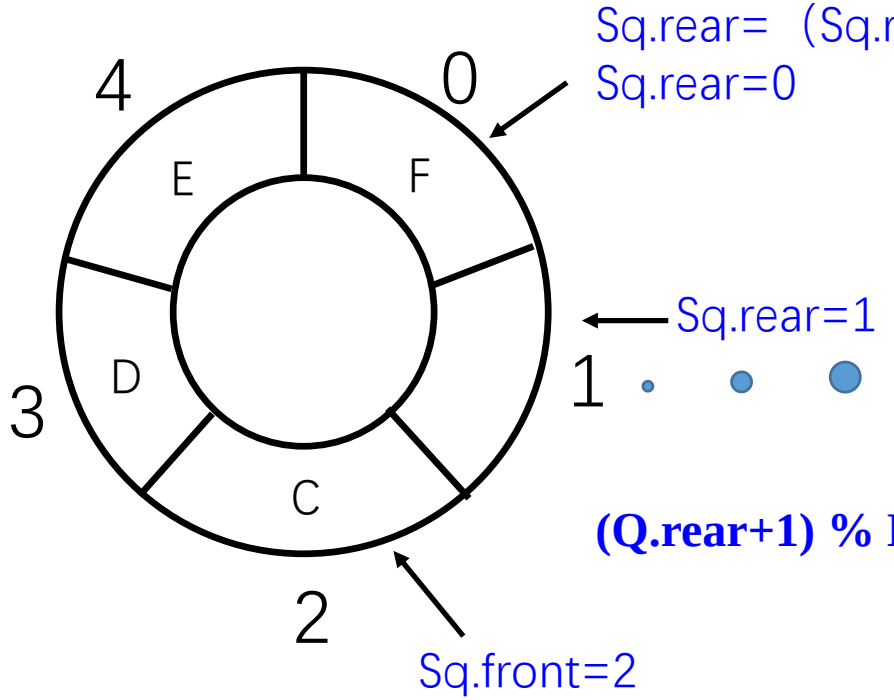
3.4.3 队列的顺序表示及实现---循环队列

- 循环队列的定义：

循环队列是顺序队列的一种特例，它是把顺序队列构造成一个首尾相连的循环表。指针和队列元素之间关系不变。



Sq.rear=5
Sq.front=2



(Q.rear+1) % MAXQSIZE == Q.front

3.4.3 队列的顺序表示及实现---循环队列

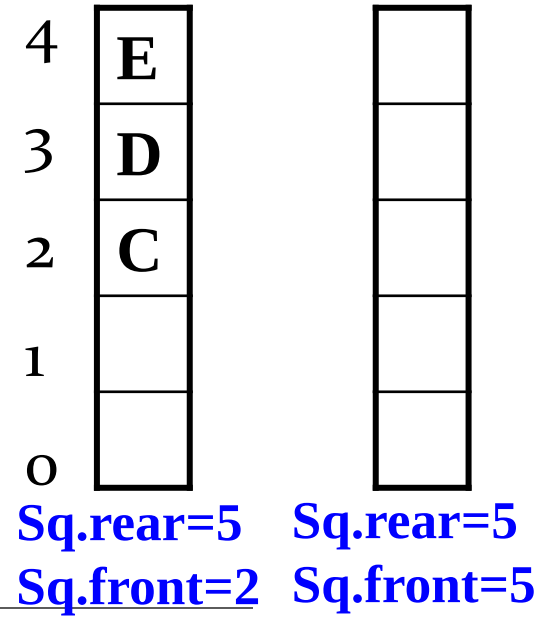
- 入队列算法：

```
Status EnQueue (SqQueue &Q, ElemType 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;
}
```

3.4.3 队列的顺序表示及实现---循环队列

- 出队列算法：

```
Status DeQueue (SqQueue &Q, ElemType &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;
}
```



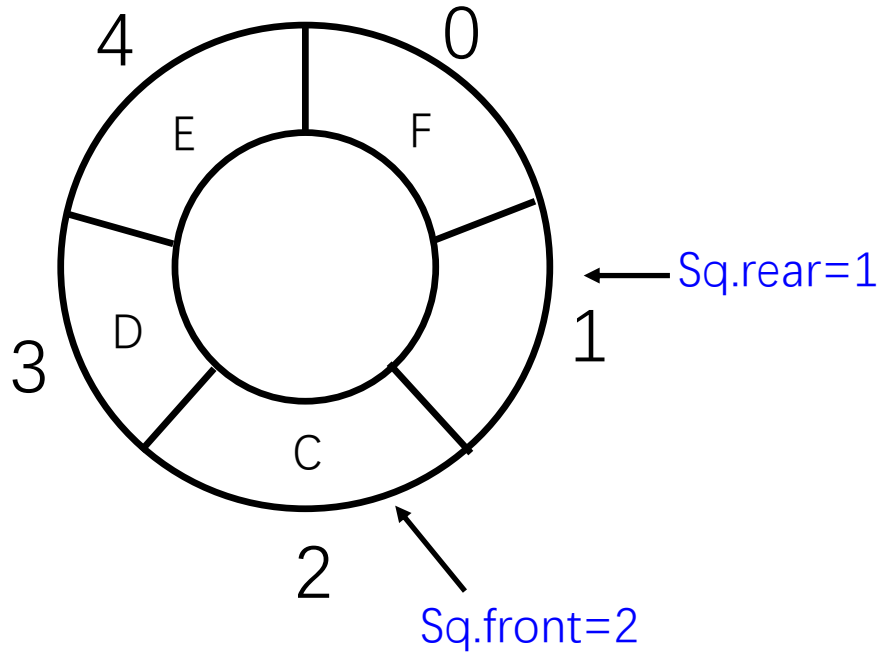
3.4.3 队列的顺序表示及实现

- 求队列长度算法：

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

3.4.3 队列的顺序表示及实现---循环队列

- 判断队列已满/已空：



判空条件：

Q.front == Q.rear

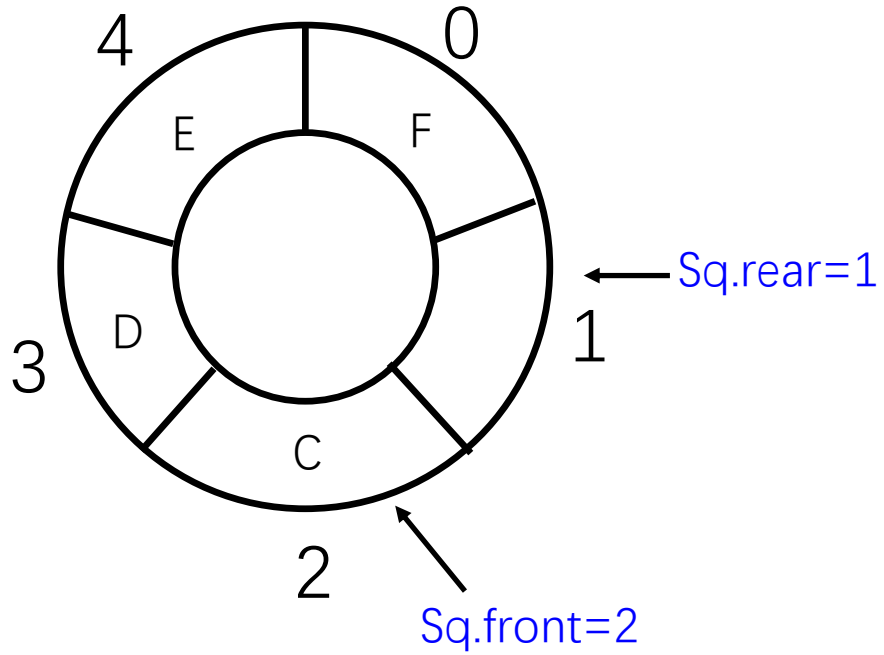
判满条件：

(Q.rear+1) % MAXQSIZE == Q.front

思考：能不能不浪费这个孤独可怜的空间？

3.4.3 队列的顺序表示及实现---循环队列

- 判断队列已满/已空：



解决方案1：

加设标志位，让删除动作使其为1,插入动作使其为0

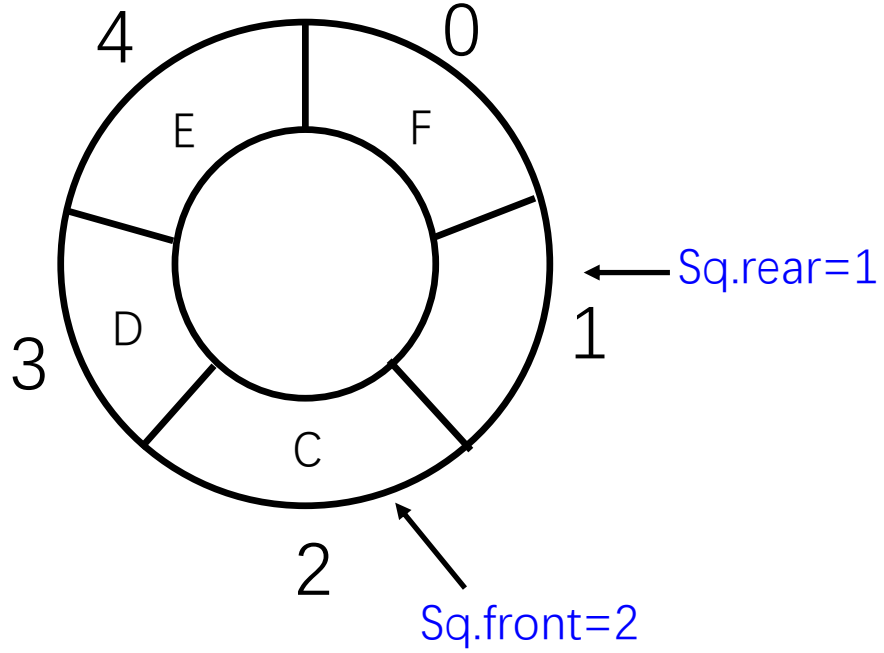
```
typedef struct {
   ElemType  base[MAXQSIZE];
   int  front，  rear;
   int  tag;
} SqQueue;
```

判空：Q.front == Q.rear && tag == 1

判满：Q.front == Q.rear && tag == 0

3.4.3 队列的顺序表示及实现---循环队列

- 判断队列已满/已空：



解决方案2：

使用一个计数器记录队列中元素个数（即队列长度）

```
typedef struct {
   ElemType  base[MAXQSIZE];
   int  front，  rear;
   int size;
} SqQueue;
```

判空：size ==0

判满：size == maxsize

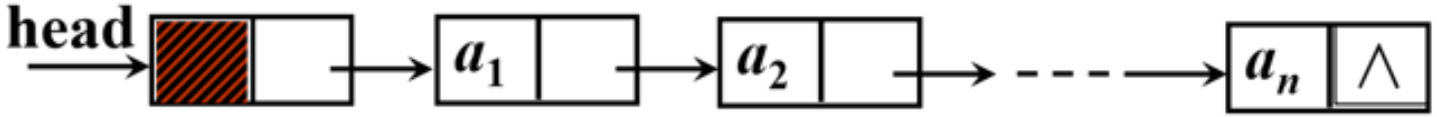
3.4.3 队列的顺序表示及实现

- **顺序队列小结：**
  - 定义和基本操作跟顺序表类似
  - 增加头尾指针方便操作
  - 通常采用循环队列的方式来解决假溢出的问题

3.4.4 队列的链式存储和实现

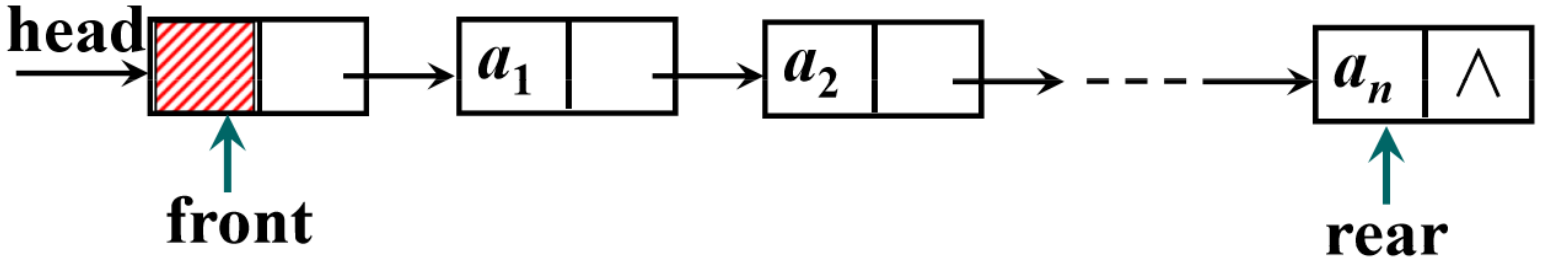
- **Recall 线性链表的结构定义：**

```
typedef struct node
{     DataType data;          //数据域
      struct node *next;      //指针域
} ListNode，*LinkList；
```



3.4.4 队列的链式存储和实现

- 队列的链接存储结构及实现：
  - 链队列：队列的链接存储结构
  - 如何改造单链表实现队列的链接存储?



  - 队首指针即为链表的头结点指针
  - 增加一个指向队尾结点的指针

3.4.4 队列的链式存储和实现

- 链队列的结点实现：

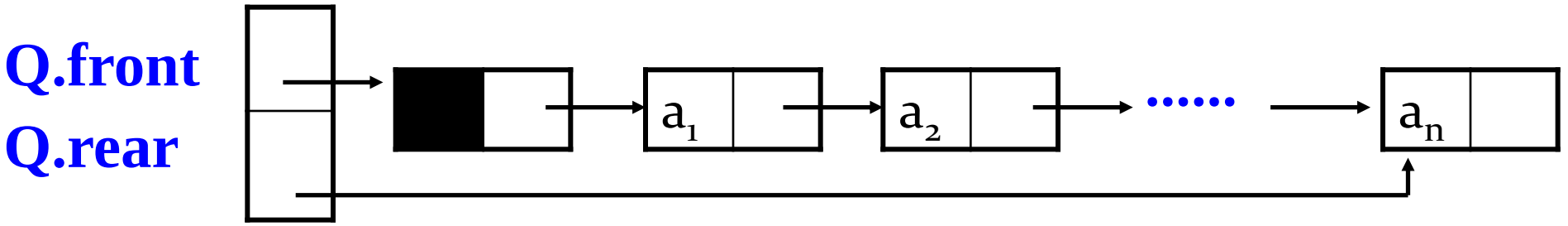
```
typedef struct { // 结点类型
  QElemType      data;
  struct QNode  *next;
} QNode, *QueuePtr;
```

- 链队列数据类型实现：

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

3.4.4 队列的链式存储和实现

- 带头结点的链队列示意图：



3.4.4 队列的链式存储和实现

- 带头结点的链队列初始化：

```
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;
}
```

3.4.4 队列的链式存储和实现

- 带头结点的链队列入队算法：

图例

```
Status EnQueue (LinkQueue &Q, QElemType e) {
// 插入元素e为Q的新的队尾元素
  QNode* 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;
}//Q.rear->next = p,表示原来的尾结点的next域指向新的结点e;
//Q.rear=p，表示rear移动到新的结点p上
```

3.4.4 队列的链式存储和实现

- 带头结点的链队列出队算法：

图例

```
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;
  QNode* p = Q.front->next;   e = p->data;
  Q.front->next = p->next;
  if (Q.rear == p)  Q.rear = Q.front;
  free (p);       return OK;
}//注意当只有一个结点时，记得队尾指针要指向队头
```

3.4.4 队列的应用

- **队列使用的原则：** 凡是符合先进先出原则的
  - 服务窗口和排号机、打印机的缓冲区、分时系统、树型 结构的层次遍历、图的广度优先搜索等等 结构的层次遍历、图的广度优先搜索等等
- **举例**
  - 约瑟夫出圈问题：n个人排成一圈，从第一个开始报数， 报到m的人出圈，剩下的人继续开始从1报数，直到所有的人都出圈为止。
  - 舞伴问题：假设在周末舞会上，男士们和女士们进入舞厅 时，各自排成一队。跳舞开始时，依次从男队和女队的队 头上各出一人配成舞伴。若两队初始人数不相同，则较长 的那一队中未配对者等待下一轮舞曲。现要求写算法模拟 上述舞伴配对问题。

例：多项式的代数运算

$$P(x) = \sum_{i=n-1}^{0} a_i \, x^{\,i}$$

方案1：数组一

n-1	a_{n-1}
…	…
i	a_i
…	…
3	a_3
2	a_2
1	a_1
0	a_0

coeftype p[N];

14	**3**
13	0
12	0
11	0
10	0
9	0
8	**2**
7	0
6	0
5	0
4	0
3	0
2	0
1	0
0	**1**

$P(x)=3x^{14}+2x^{8}+1$

方案2：数组二

$coef_{n-1}$	exp_{n-1}
$coef_{n-2}$	exp_{n-2}
…	…
$coef_i$	exp_i
…	…
$coef_2$	exp_2
$coef_1$	exp_1
$coef_0$	exp_0
…	…

```
Struct {
        coeftype  coef;
        exptype exp;
} p[N]
```

$P(x)=3x^{14}+2x^{8}+1$

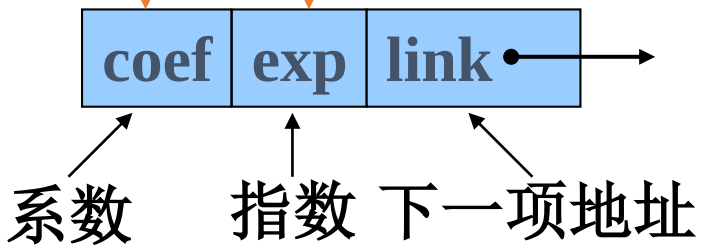
3	14
2	8
1	0
	…

方案3：链表

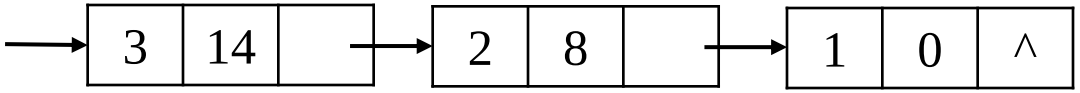
$$P(x) = \sum_{i=n}^{0} a_i x^i$$

结点类型：

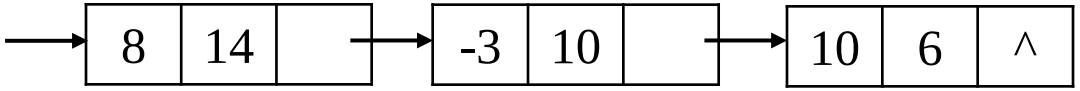
```
struct PolyNode {
    int   coef ;
    int   exp ;
    polylink  *link ;
}
typedef PolyNode
*PolyPointer ;
```



$a(x) = 3x^{14} + 2x^8 + 1$



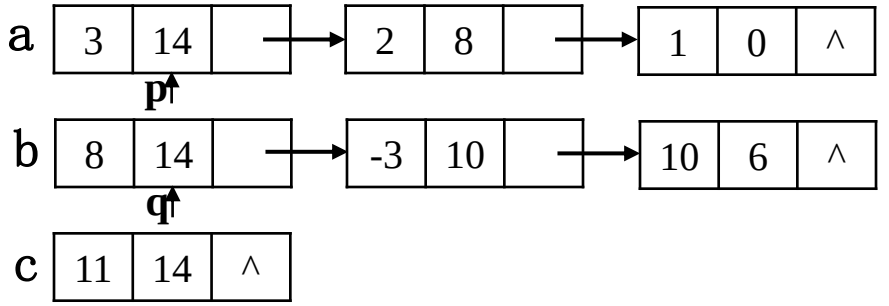
$b(x) = 8x^{14} - 3x^{10} + 10x^6$



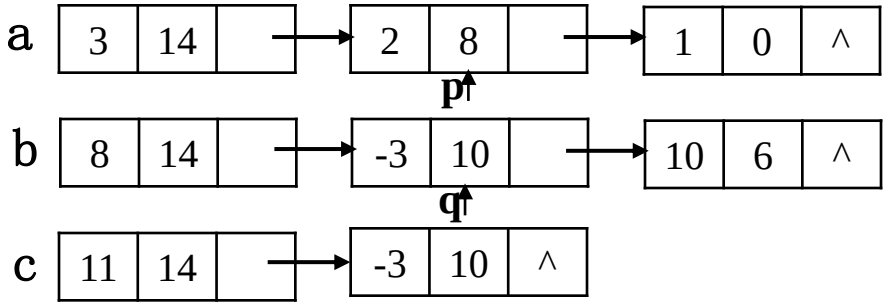
$c(x) = a(x) + b(x)$

$= 11x^{14} - 3x^{10} + 2x^8 + 10x^6 + 1$

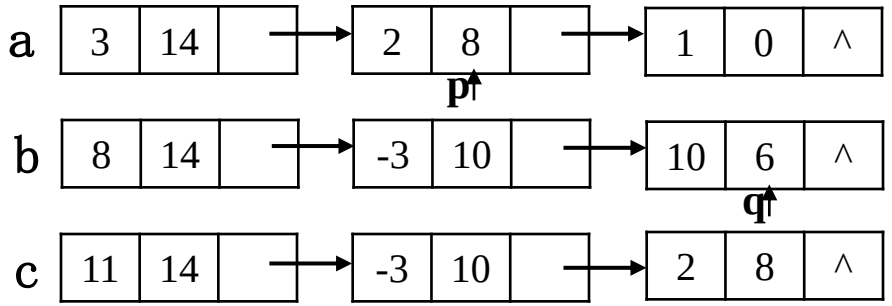
(1) p->exp == q->exp



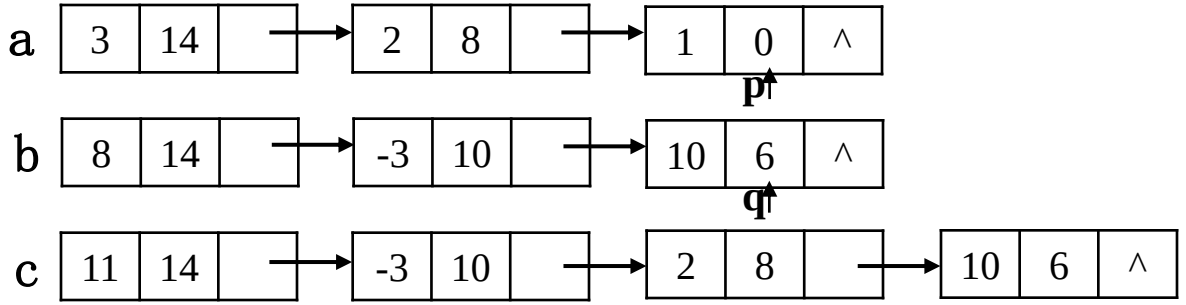
(2) p->exp < q->exp



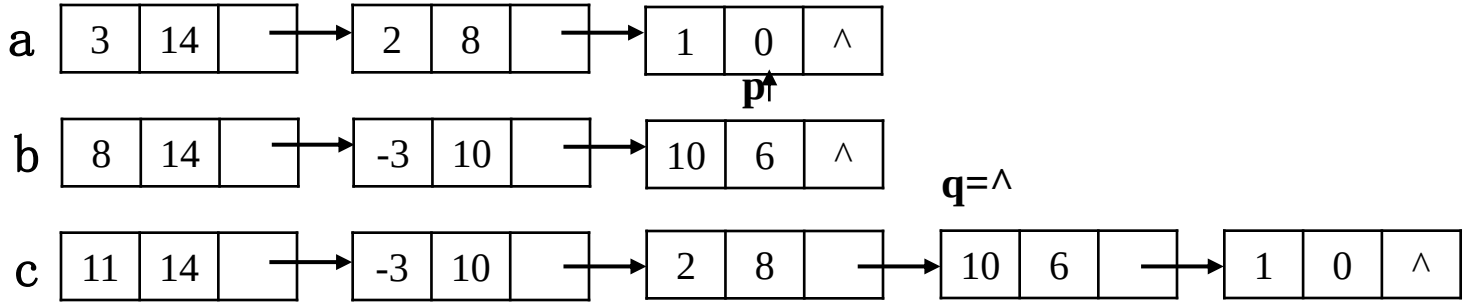
(3) **p->exp > q->exp**



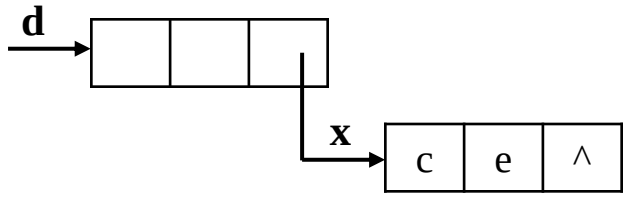
(4) **p->exp < q->exp**



(5) **p != null**

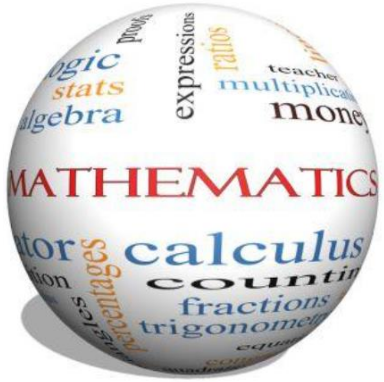


Attch(int c, int e, PolyPointer d)




```
PolyPointer  Attch ( int  c , int e ,
PolyPointer  d )
{     PolyPointer   x  ;
      x = new   PolyNode ;
      x->coef = c ;
      x->exp = e ;
      d->link = x ;
      return  x ;
}
```

Compare(int x , int y) =
- ‘=’ 当x=y
- ‘>’ 当x>y
- ‘<’ 当x<y




```
char  Compare ( int x, int y)
{
   char c;
   if( x == y )
        c = ‘=’;
    else if( x > y )
        c = ‘>’;
    else
        c = ‘<’;
    return( c );
}
```

表达式加法算法：

```
PolyPointer PolyAdd ( PolyPointer a ,
                                  PolyPointer b )
{ PolyPointer p, q, d, c;
  int y ;
  p = a->link;  q = b->link;
  c = new PolyNode; d = c ;
  while ( (p != NULL) && (q != NULL) )
      switch ( Compare ( p->exp, q->exp ) )
      {   case '=' :
            y = p->coef + q->coef ;
            if ( y )  d = Attch( y, p->exp, d );
            p = p->link ; q = q->link ;
            break;
          case '>':
            d = Attch( p->coef, p->exp, d );
            p = p->link ;
           break;
          case '<':
            d = Attch( q->coef, q->exp, d );
            q = q->link ;
             break;
      }
      while ( p != NULL )
      { d = Attch( p->coef, p->exp, d );
        p = p->link ;
       }
       while ( q !=NULL )
       { d = Attch( q->coef, q->exp, d );
         q = q->link ;
       }
       d->link = NULL ;
       p = c;  c = c->link;
       delete  p;
       return  c;
}
```

本章小结

- ✓ 熟练掌握：
  - (1)栈、队列的定义、特点和性质；
  - (2)ADT栈、ADT队列的设计和实现以及基本操作及相关算法。
- ✓ 重点学习：
  - ADT栈和队列在递归、表达式求值、括号匹配、数制转换、迷宫求解中的应用，提高利用栈和队列解决实际问题的应用水平。