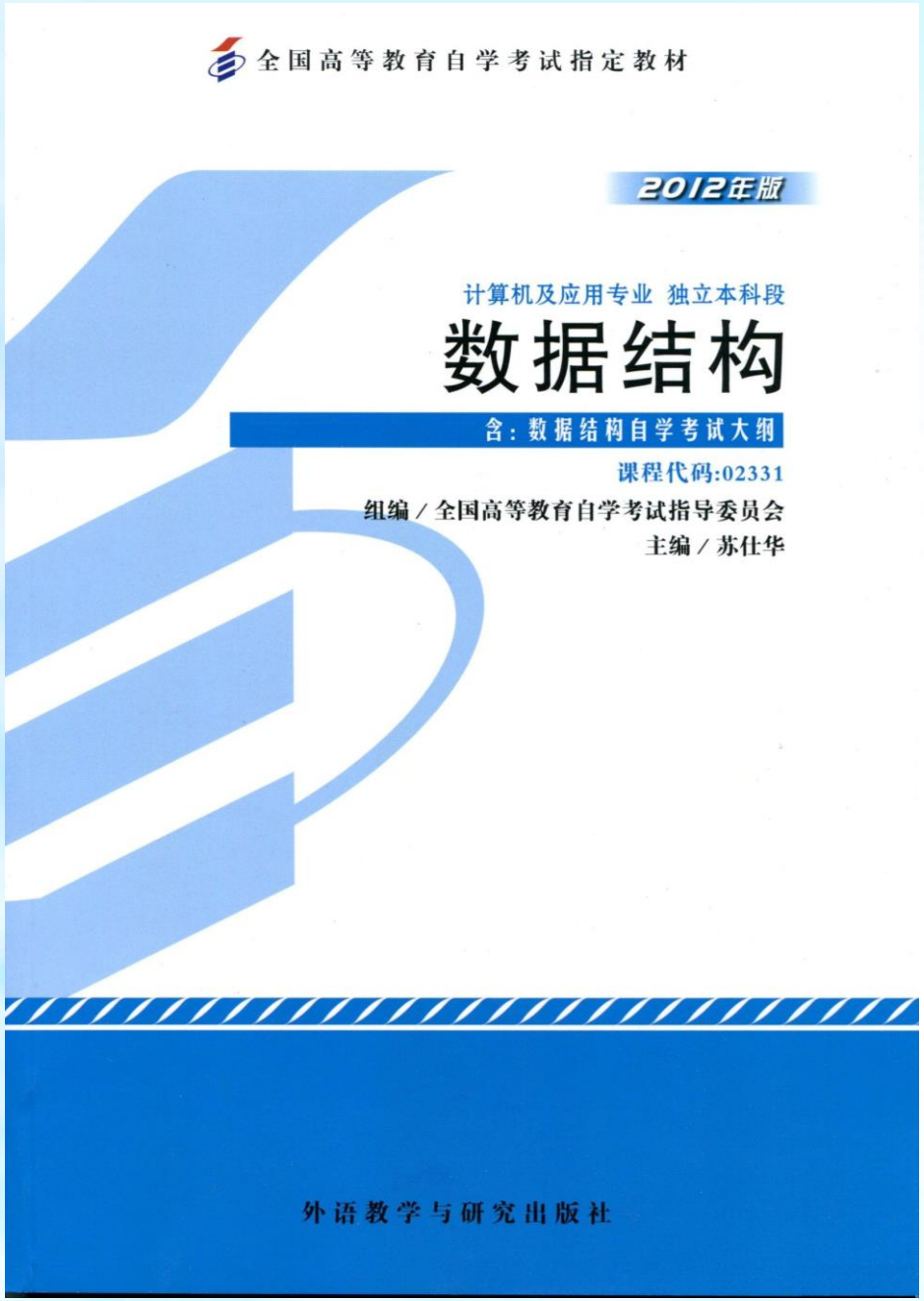
尚德机构

数据结构

主讲：王老师

学习是一种信仰！ IN LEARING WE TRUST

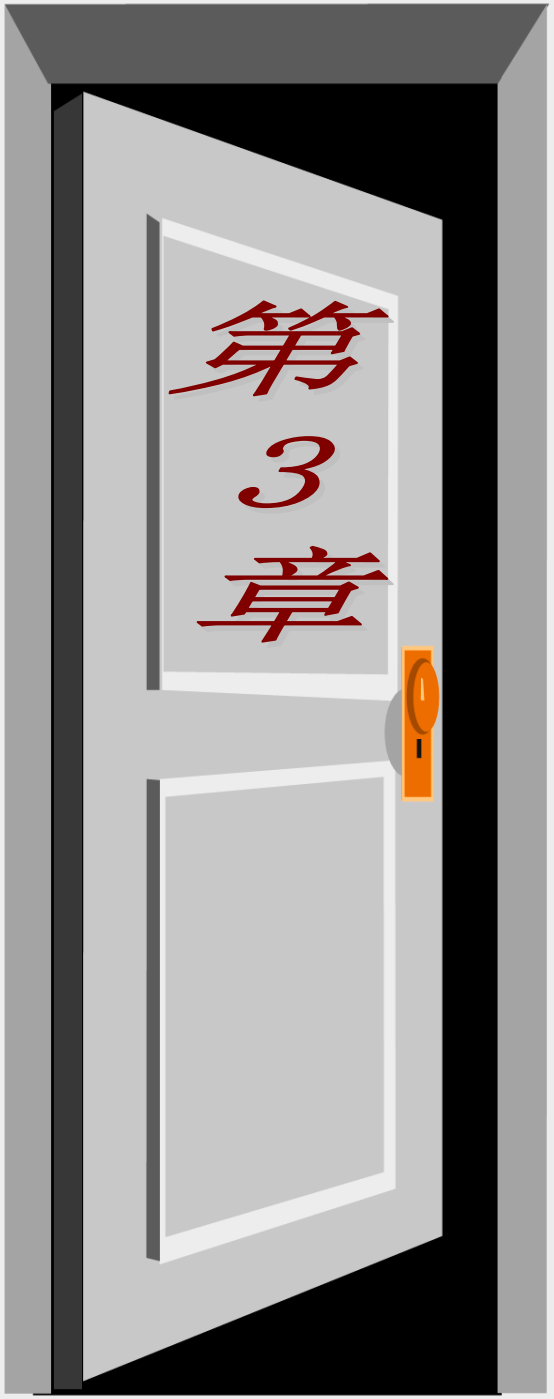


全国高等教育自学考试指定教材 2331数据结构

主　编　　苏仕华

出版社　　外语教学与研究出版社

出版时间　2012年3月



栈和队列

栈（Stack）是限定在表的一端进行插入和删除运算的线性表，通常将插入、删除的一端称为栈顶（top），另一端称为栈底（bottom）。不含元素的空表称为空栈。

根据上述栈的定义，每次删除（退栈）的总是当前栈中最后插入（进栈）的元素，而最先进栈的元素在栈底，要到最后才能删除。假设栈S=(a_1, a_2, ... a_n)，若栈中元素按a_1, a_2, ... a_n的次序进栈，其中a_1为栈底元素，a_n为栈顶元素，而退栈的次序却是a_n, a_{n-1}, ... a_1 。也就是说，栈的修改是按后进先出的原则进行的，如图3.1所示。因此，栈又称为**后进先出**（Last In First Out)的线性表，简称为LIFO表。

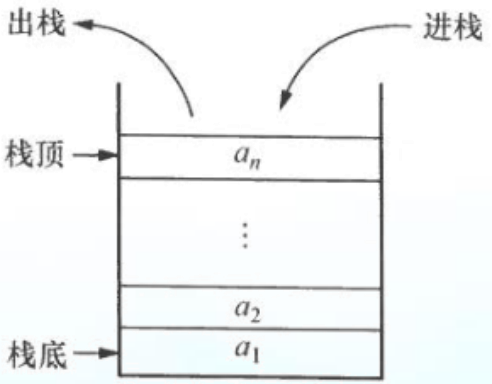
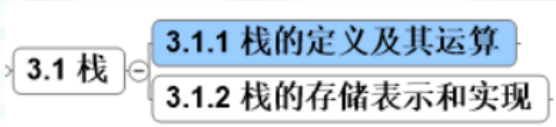


图 3.1　栈的示意图



栈的基本运算除了在栈顶进行插入或删除运算外，还有栈的初始化、判栈空及取栈顶元素等运算。栈的运算主要有以下几种。

①置空栈 InitStack (&S): 构造一个空栈S。

②判栈空 StackEmpty (S): 若栈S为空栈，则返回TRUE，否则返回FALSE。

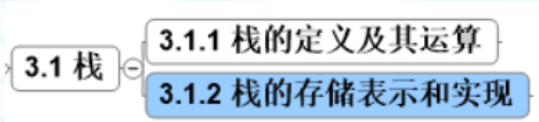
③判找满 StackFull (S): 若栈S为满找，则返回TRUE，否则返回FALSE。

④进栈（又称入栈或插入） Push (&S，x): 将元素x插入S栈的栈顶。

⑤退栈（又称出栈或删除） Pop (&S): 若栈S为非空，则将S的栈顶元素删除，并返回栈顶元素。

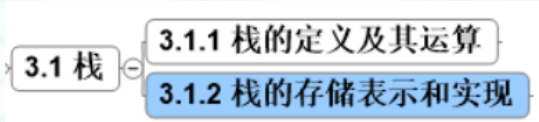
⑥取栈顶元素 GetTop (S): 若S栈为非空，则返回栈顶元素，但不改变栈的状态。同线性表一样，利用以上六种找的基本运算，就可以实现有关的应用要求。

3.1.2二、栈的存储表示和实现 1. 栈的顺序存储结构

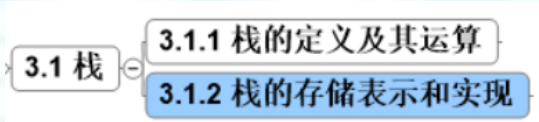


顺序栈定义如下：

```
# define StackSize 100     //找空间的大小应根据实际需要来定义，这里假设为100
typedef char DataType ;  //DataType的类型可根据实际情况而定，这里假设为char
typedef struct {
    DataType data [ StackSize ] ;      //数组data用来存放表结点
    int top ;                          //表示栈顶指针
} SeqStack ;
SeqStack S ;
```



设S是SeqStack类型的顺序栈。S.data[0]是栈底元素，那么栈顶S.data[top]是正向增长的，即进栈时需将S.top加1，退找时需将S.top减1。因此，S.top<0表示空栈，S.top=StackSize-1表示栈满。当栈满时再做进栈运算必定产生空间溢出，简称“上溢”；当栈空时再做退栈运算也将产生溢出，简称“下溢”。



【例3.1】对于一个栈，给出输入序列为abc，试写出全部可能的输出序列。

分析：因为栈是受限在一端输入或输出，而且有“后进先出”的特点，所以本题有如下几种情况：

（1）a进 a出 b进 b出 c进 c出　　产生输出序列abc

（2）a进 a出 b进 c进 c出 b出　　产生输出序列acb

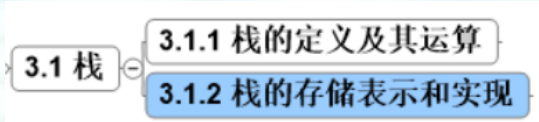
（3）a进 b进 b出 a出 c进 c出　　产生输出序列bac

（4）a进 b进 b出 c进 c出 a出　　产生输出序列bca

（5）a进 b进 c进 c出 b出 a出　　产生输出序列cba

不可能产生的输出序列cab。

本题的答案是：abc, acb, bac, bca, cba。



【例3.2】设一个栈的输入序列为：1, 2, 3, 4, 5, 则下列序列中不可能是栈的输出序列的是（　　）

A. 1, 2, 3, 4, 5

B. 5, 4, 3, 2, 1

C. 2, 3, 4, 5, 1

D. 4, 1, 2, 3, 5

分析：像这类题目，一是要知道找的操作规则，即后进先出；再一个就是排除可能的，那么剩下的自然是不可能的。例如，该题就是按照入栈退栈的规则，一一排除可能的序列：序列A很显然不可能是，因为1进1出，2进2出，…，故得到出栈序列1, 2, 3, 4, 5；序列B也显然不可能是, 先1, 2, 3, 4, 5入栈, 出栈是5, 4, 3, 2, 1；序列C是1, 2进栈，2出栈，3进3出，4进4出， 5进5出, 最后1出栈。因此本题的答案是D。

真题演练

设栈S的输入序列为1，2，3，4，5，则下列选项中不可能是S的输出序列的是（　　）。

A: 2,3,4,1,5

B: 5,4,1,3,2

C: 2,3,1,4,5

D: 1,5,4,3,2

真题演练

设栈S的输入序列为1，2，3，4，5，则下列选项中不可能是S的输出序列的是（　　）。

A: 2,3,4,1,5

B: 5,4,1,3,2

C: 2,3,1,4,5

D: 1,5,4,3,2

答案：B

真题演练

若进栈次序为a，b，c，且进栈和出栈可以穿插进行，则可能出现的含3个元素的出栈序列个数是（　）。

A: 3

B: 5

C: 6

D: 7

真题演练

若进栈次序为a，b，c，且进栈和出栈可以穿插进行，则可能出现的含3个元素的出栈序列个数是（　）。

A: 3

B: 5

C: 6

D: 7

答案：B

真题演练

设栈的初始状态为空，元素1，2，3，4，5依次入栈，不能得到的出栈序列是（　）。

A: 1，2，3，4，5

B: 4，5，3，2，1

C: 1，2，5，4，3

D: 1，2，5，3，4

真题演练

设栈的初始状态为空，元素1，2，3，4，5依次入栈，不能得到的出栈序列是（　）。

A: 1，2，3，4，5

B: 4，5，3，2，1

C: 1，2，5，4，3

D: 1，2，5，3，4

答案：D

真题演练

栈中有a、b和c三个元素,a是栈底元素,c是栈顶元素,元素d等待进栈,则不可能的出栈序列是(　　)。

A: dcba

B: cbda

C: cadb

D: cdba

真题演练

栈中有a、b和c三个元素,a是栈底元素,c是栈顶元素,元素d等待进栈,则不可能的出栈序列是(　　)。

A: dcba

B: cbda

C: cadb

D: cdba

答案：C

真题演练

设栈的进栈序列为a，b，c，d，e，经过合理的出入栈操作后，不能得到的出栈序列是（　　）。

A: d，c，e，a，b

B: d，e，c，b，a

C: a，b，c，d，e

D: e，d，c，b，a

真题演练

设栈的进栈序列为a，b，c，d，e，经过合理的出入栈操作后，不能得到的出栈序列是（　　）。

A: d，c，e，a，b

B: d，e，c，b，a

C: a，b，c，d，e

D: e，d，c，b，a

答案：A

真题演练

设栈的初始状态为空，元素1、2、3、4、5、6依次入栈，得到的出栈序列是（2,4,3,6,5,1）,则栈的容量至少是（　　）。

A: 2

B: 3

C: 4

D: 6

真题演练

设栈的初始状态为空，元素1、2、3、4、5、6依次入栈，得到的出栈序列是（2,4,3,6,5,1）,则栈的容量至少是（　　）。

A: 2

B: 3

C: 4

D: 6

答案：B

真题演练

设栈的初始状态为空，元素1,2,3,4,5,6依次入栈，栈的容量是3，能够得到的出栈序列是（　　）。

A: 1,2,6,4,3,5

B: 2,4,3,6,5,1

C: 3,1,2,5,4,6

D: 3,2,6,5,1,4

真题演练

设栈的初始状态为空，元素1,2,3,4,5,6依次入栈，栈的容量是3，能够得到的出栈序列是（　　）。

A: 1,2,6,4,3,5

B: 2,4,3,6,5,1

C: 3,1,2,5,4,6

D: 3,2,6,5,1,4

答案：B

真题演练

已知一个栈的入栈序列是1，2，3，…，n，其输出序列为p1，p2，p3，…，p_n，若p1是n，则pi是（　　）。

A: i

B: n-i

C: n-i+1

D: 不确定

真题演练

已知一个栈的入栈序列是1，2，3，…，n，其输出序列为p1，p2，p3，…，p_n，若p1是n，则pi是（　　）。

A: i

B: n-i

C: n-i+1

D: 不确定

答案：C

真题演练

若元素的入栈顺序为1，2，3，…，n，如果第2个出栈的元素是n，则输出的第i(1<=i<=n)个元素是( )

A: n-i

B: n-i+1

C: n-i+2

D: 无法确定

真题演练

若元素的入栈顺序为1，2，3，…，n，如果第2个出栈的元素是n，则输出的第i(1<=i<=n)个元素是( )

A: n-i

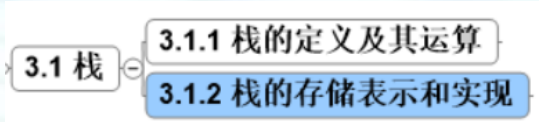
B: n-i+1

C: n-i+2

D: 无法确定

答案：D

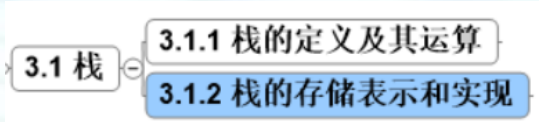
2. 顺序栈基本运算的实现



（1）置空栈

```
void InitStack ( SeqStack * S )
{       //置空顺序栈。由于c语言数组下标是从0开始，所以栈中元素亦从0开始
        //存储，因此空栈时栈顶指针不能是0，而只能是-1
    S->top=-1 ;
 }
```

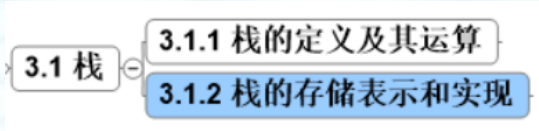
2. 顺序栈基本运算的实现



(2) 判栈空

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

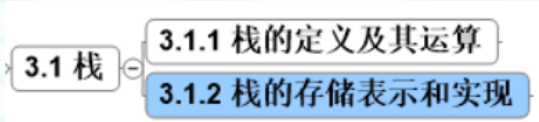
2. 顺序栈基本运算的实现



（3）判栈满

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

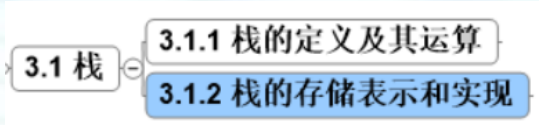
2. 顺序栈基本运算的实现



(4) 进栈（入栈）

```
void Push ( SeqStack * S, DataType x )
{
    if ( StackFull ( S ) )
    printf ( "stack overflow" ) ;
    else {
        S->top=S->top+1 ;          //栈顶指针加1
        S->data[S->top]=x ;        //将x入栈
    }
}
```

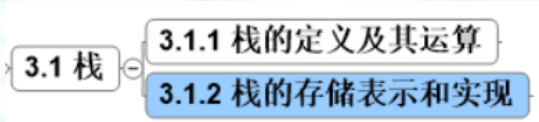
2. 顺序栈基本运算的实现



（5）退栈（出栈）

```
DataType Pop ( SeqStack *S )
{
    if ( StackEmpty ( S ) ) {
        printf ( "stack underflow" ) ;
        exit ( 0 );                          //出错退出处理
     }
    else
        return S->data [ S->top--] ;         //返回栈顶元素后栈顶指针减1
 }
```

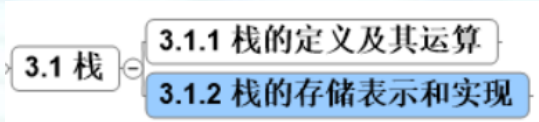
2. 顺序栈基本运算的实现



（6）取栈顶元素（不改变栈顶指针）

```
DataType GetTop ( SeqStack *S )
{
    if ( StackEmpty ( S ) ) {
        printf ( "stack empty" ) ;
        exit ( 0 ) ;                          //出错退出处理
    }
    else
        return S->data[S->top] ;              //返回栈顶元素
}
```

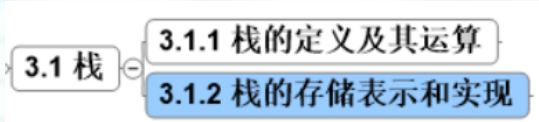
2. 顺序栈基本运算的实现



由于顺序栈必须预先分配存储空间，因此在应用中要考虑溢出问题。另外，在实际应用中还可能同时使用多个栈，为了防止溢出，需要为每个栈分配一个较大的空间，这样做往往会产生空间上的浪费，因为当某一个栈发生溢出的同时，其余的栈还可能有很多的未用空间。如果将多个栈分配在同一个顺序存储空间内，即让多个栈共享存储空间，则可以相互进行调节，既节约了空间，又可降低发生溢出的频率。

当程序中同时使用两个栈时，可以将两个栈的栈底分别设在顺序存储空间的两端，让两个栈顶各自向中间延伸。当一个栈中的元素较多而栈使用的空间超过共享空间的一半时，只要另一个栈中的元素不多，就可以让第一个栈占用第二个栈的部分存储空间。只有当整个存储空间被两个找占满时（即两栈顶相遇），才会产生溢出。

3. 栈的链式存储结构及基本操作



为了克服这种由顺序存储分配固定空间所产生的溢出和空间浪费问题，可以采用链式存储结构来存储栈。栈的链式存储结构称为链栈，它是运算受限的单链表，其插入和删除操作仅限制在表头位置上（栈顶）进行，因此不必设置头结点，将单链表的头指针head改为栈顶指针top即可。

3. 栈的链式存储结构及基本操作

链栈的类型定义如下：

```
typedef struct stacknode {
        DataType data ;
        struct stacknode * next ;
} StackNode ;
typedef StackNode * LinkStack ;
LinkStack top ;
```

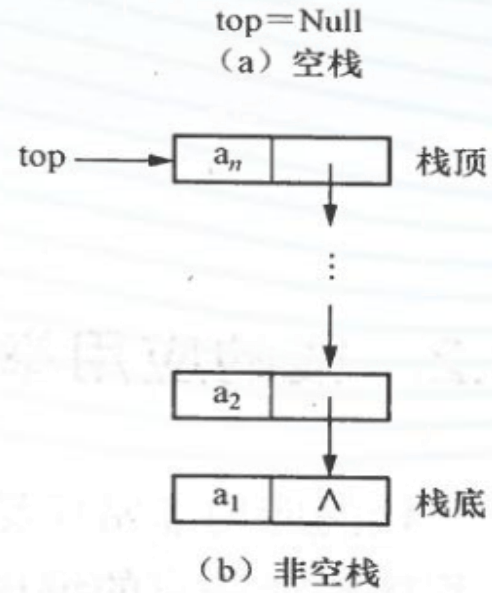
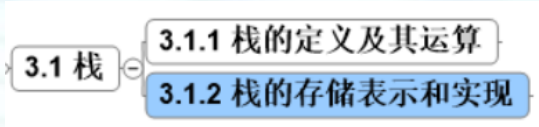


图 3.2　链栈结构示意图

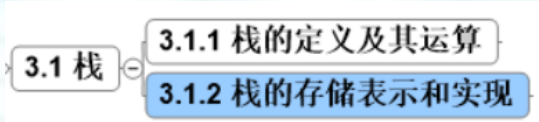
3. 栈的链式存储结构及基本操作



(1) 判栈空

```
int StackEmpty ( LinkStack top )
{
    return top==NULL ;
}
```

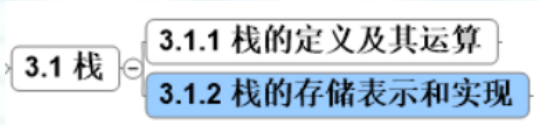
3. 栈的链式存储结构及基本操作



（2）进找（入栈）

```
LinkStack Push ( LinkStack top , DataType x )
{   //将元素x插入栈顶
    StackNode * p ;
    p=(StackNode * ) malloc ( sizeof ( StackNode ) ) ; //申请新结点
    p->data=x ;
    p->next=top ;                    //将新结点*p插入栈顶
    top=p ;                          //使top指向新的栈顶
    return  top;                     //返回新栈顶指针
}
```

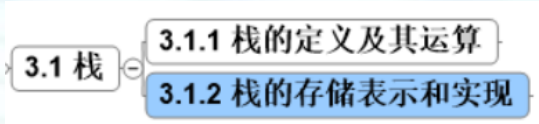
3. 栈的链式存储结构及基本操作



（3）退栈（出栈）

```
LinkStack Pop ( LinkStack top, DataType *x )
{   StackNode * p=top ;                          //保存栈顶指针
    if ( StackEmpty ( top ) ) {
          printf (  “stack empty") ;             //栈为空
          exit ( 0 ) ; }                         //出错退出处理
    else {
          *x=p->data ;                           //保存删除结点值，并带回
          top=p->next ;                          //栈顶指针指向下一个结点
          free ( p ) ;                           //删除P指向的结点
          return top ;                           //并返删除后的栈顶指针
     }
}
```

3. 栈的链式存储结构及基本操作



（4）取栈顶元素

```
DataType GetTop ( LinkStack top )
{                                                    //取栈顶元素
    if ( StackEmpty ( top ) ) {
        printf ( "stack empty" ) ;                   //栈为空
        exit ( 0 ) ;                                 //出错退出处理
    }
    else
        return top->data ;                           //返回找顶结点值
}
```

可利用栈的操作来实现：循环读入表达式中的字符，如遇左括号就进找；遇右括号则判断栈是否为空，若为空，则返回FALSE，否则退栈；循环结束后再判断栈是否为空，若栈空则说明括号匹配，否则说明不匹配。其实现算法如下：

```
int Expr( )
{   SeqStack S ;
    DataType ch , x ;
    InitStack ( &S ) ;                  //初始化栈S
    ch=getchar ( ) ;
    while ( ch!= '\n' ) {
        if ( ch== '(' )
            Push ( &S, ch ) ;           //遇左括号进钱
        else
            if ( ch==')' )
                if ( StackEmpty ( &S ) ) return 0;
                 else x=Pop ( &S ) ;    //遇右括号退栈
            ch=getchar ( ) ;            //读入下一个字符
    } // end of while
    if ( StackEmpty ( &S ) ) return 1;
    else return 0 ;
}
```

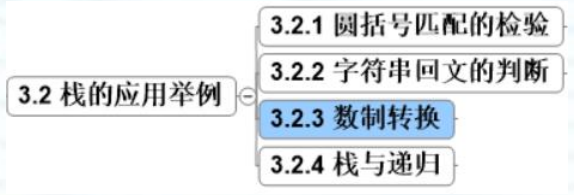
3.2.2二、字符串回文的判断

利用顺序栈的基本运算，试设计一个算法，判断一个输入字符串是否具有中心对称，例如abababa和abcba都是中心对称的字符串。

分析：所谓“中心对称”，首先要知道中心在哪儿，有了中心位置之后，就可以从中间向两头进行比较，若完全相同，则该字符串是中心对称，否则不是。这就要首先求出字符串串的长度，然后将前一半字符入栈，再利用退栈操作将其与后一半字符进行比较。可设计算法如下：

```
int symmetry (char str[ ] )
{   SeqStack S ;
    int j, k, i=0 ;
    InitStack ( &S ) ;
    while ( str [ i ]!=’ \0’ ) i++;                    //求串长度
    for ( j=0 ; j<i/2 ; j++ )
        Push ( &S,str [ j ] ) ;                        //前一半字符入栈
    k= ( i+1 )/2;                                      //后一半字符在串中的起始位置
    for ( j=k; j<i; j++ )                              //后一半字符与栈中字符比较
    if ( str [ j ]!=Pop ( &S ) )
        return 0 ;                                     //有不相同字符，即不对称
    return 1 ;                                         //完全相同，即对称
}
```

3.2.3三、数制转换




```
void conversion ( int N, int d )
{   //将一个非负的十进制数N转换成任意的d进制数
    SeqStack S ;
    InitStack ( &S )
    while ( N ) {
        Push ( &S, N % d ) ;
            N=N/d;
    }
    while ( !StackEmpty ( &S ) ) {
        i=Pop ( &S ) ;
        printf ( “%d”, i ) ;
    }
}
```

例如，要将十进制数1348转换成八进制数，按上述算法，栈的变化情况如图3.3所示。

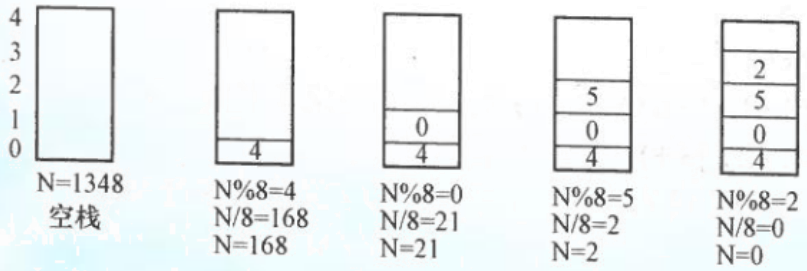
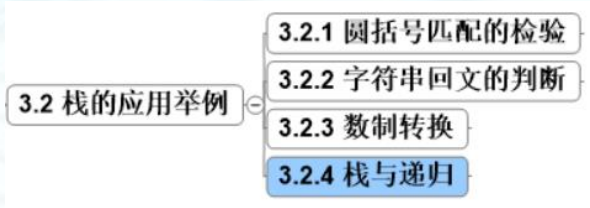


图 3.3　栈变化示意图

3.2.4四、栈与递归



【例3.3】试分析求阶乘的递归函数。

解：求阶乘的递归函数定义如下：

```
long int fact ( int n )
{    int temp;
     if ( n==0 )
        return1;
     else
        temp=n*fact ( n-1 ) ;
     r12: return temp ;
}
```

调用层次	调用	参数 n	返回地址	temp 结果	退栈时计算结果	
↑ 5	fact(0)	0	rl2	1		↓
↑ 4	fact(1)	1	rl2	1*fact(0)	1*1=1	↓
↑ 3	fact(2)	2	rl2	2*fact(1)	2*1=2	↓
↑ 2	fact(3)	3	rl2	3*fact(2)	2*3=6	↓
↑ 1	fact(4)	4	rl2	4*fact(3)	4*6=24	↓
↑ 0	fact(5)	5	rl1	5*fact(4)	5*24=120	返回
进栈				主函数打印 n=120		退栈

图 3.4　系统工作栈递归的变化示意图

如果用一个如下所示的C语言主函数来调用上述递归函数：

```
void main ( )
{    long int n ;
     n=fact ( 5 ) ;
     r11: printf ( “5!=%ld”, n ) ;
}
```

真题演练

下到选项中，不宜通过栈求解的问题是（　　）。

A: 判断字符串是否是回文

B: 检验圆括号是否匹配

C: 不同数制之间进行转换

D: 图的广度优先搜索遍历

真题演练

下到选项中，不宜通过栈求解的问题是（　　）。

A: 判断字符串是否是回文

B: 检验圆括号是否匹配

C: 不同数制之间进行转换

D: 图的广度优先搜索遍历

答案：D

真题演练

递归实现或函数调用时，处理参数及返回地址，应采用的数据结构是（　　）。

A: 堆栈

B: 多维数组

C: 队列

D: 线性表

真题演练

递归实现或函数调用时，处理参数及返回地址，应采用的数据结构是（　　）。

A: 堆栈

B: 多维数组

C: 队列

D: 线性表

答案：A

队列（Queue）也是一种操作受限的线性表，它只允许在表的一端进行元素插入，而在另一端进行元素删除。允许插入的一端称为队尾（rear），允许删除的一端称为队头（front）。

在队列中，通常把元素的插入称为入队，而元素的删除称为出队。队列的概念与现实生活中的排队相似，新来的成员总是加入队尾，排在队列最前面的总是最先离开队列，即先进先出，因此又称队列为先进先出（FIFO）表。

假设队列q=(a_1, a_2, … a_n)，是在空队列情况下依次加入元素a_1, a_2, … a_n 之后形成的，a_1是队头元素，a_n是队尾元素，则退出队列也必须按此次序进行；也就是说，只有在a_1, a_2, … a_{n-1}都出队之后，a_n才能出队列。队列的示意图如图3.7所示。

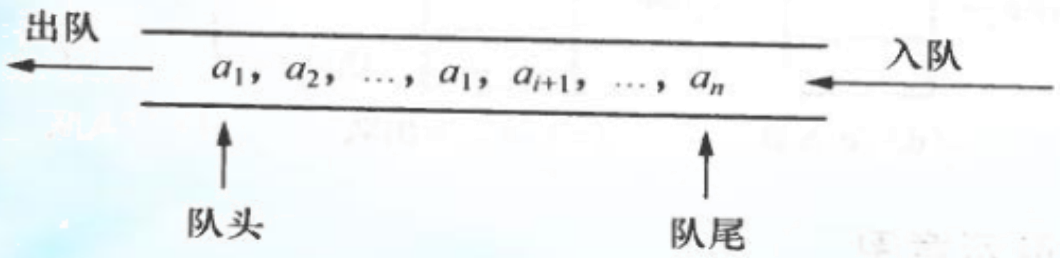


图 3.7　队列示意图

队列的操作运算与栈类似，有关队列的基本运算如下：

（1）置空队列 InitQueue(Q)，构造一个空队列Q。

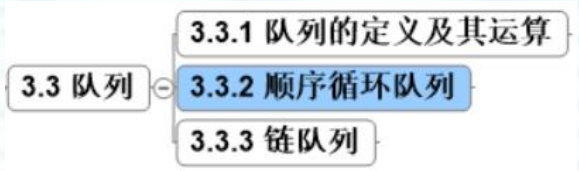
（2）判队空 QueueEmpty(Q)，若Q为空队列，则返回TRUE，否则返回FALSE。

（3）入队列 EnQueue(Q，x)，若队列不满，则将数据x插入到Q的队尾。

（4）出队列 DeQueue(Q)，若队列不空，则删除队头元素，并返回该元素。

（5）取队头 GetFmm(Q)，若队列不空，则返回队头元素。

3.3.2二、顺序循环队列



顺序队列的类型定义如下：

```
# define QueueSize 100
typedef struct {
    DataType data [QueueSize];
    int front, rear;
} SeqQueue;
SeqQueue Q;                          //定义一个顺序队列Q
```

入队运算可描述为：

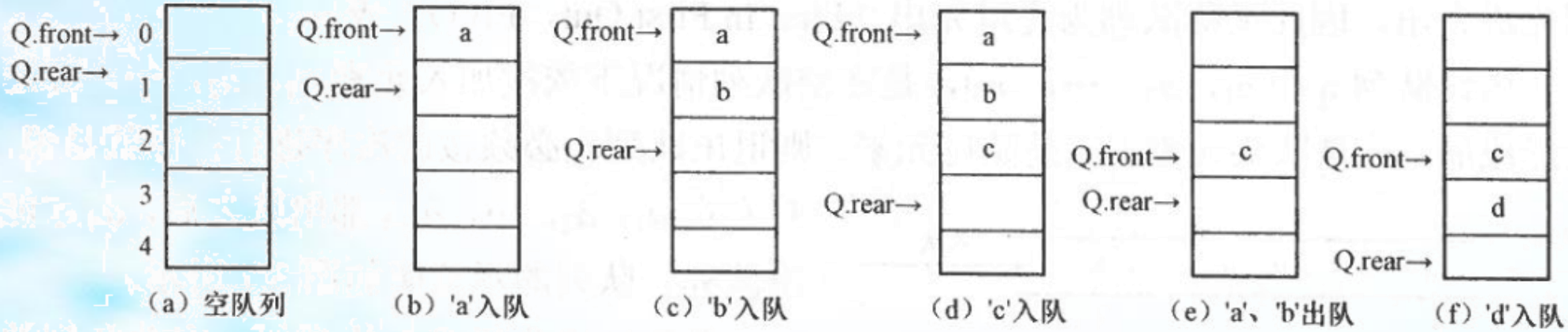
```
Q.data[Q.rear]=x; Q.rear=Q.rear+1;
```

而出队运算为：

```
x=Q.data[Q.front]; Q.front=Q.front+1; return x;
```

3.3.2 顺序循环队列

图3.8给出了一个入队和出队操作的例子，可说明头、尾指针和队列中元素之间的关系。假设队列分配的最大空间为5，当队列处于如图3.8(f)所示的状态时，如果再继续插入新的元素就会产生上溢，而出队时空出的一些存储单元无法使用；如将队列的存储空间定义得太大，则会产生存储空间的浪费。



3.3.2 顺序循环队列

为了充分利用数组空间，克服上溢，可将数组空间想象为一个环状空间，如图3.9所示，并称这种环状数组表示的队列为循环队列。

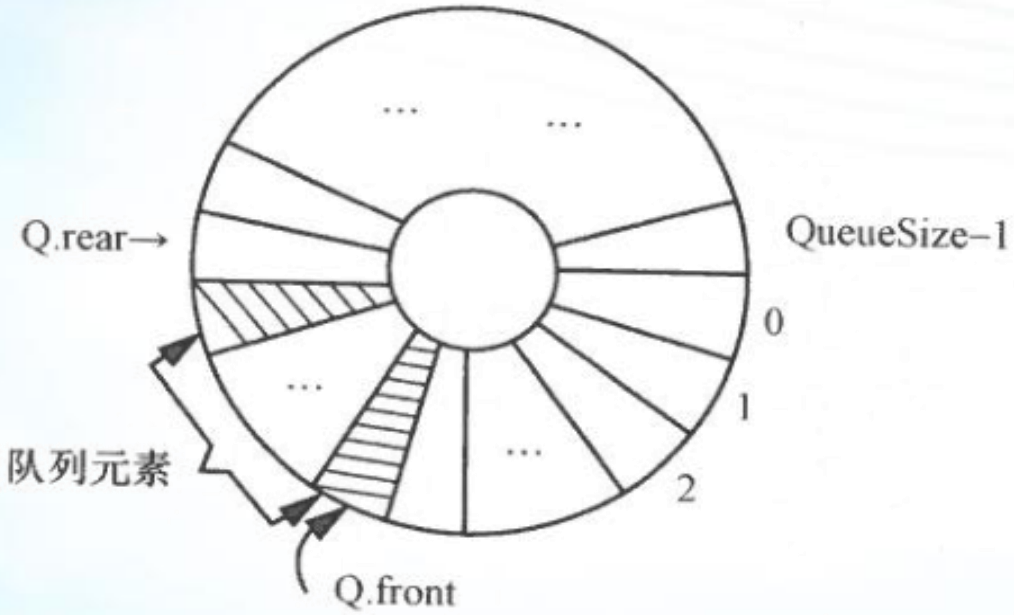
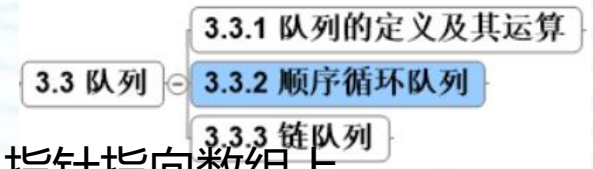


图 3.9　循环队列示意图

3.3.2 顺序循环队列



在这种循环队列中进行入队、出队运算时，头尾指针仍然要加1，只不过当头尾指针指向数组上界（QueueSize-1）时，如果按正常的加1运算，数组就会产生越界溢出，因此，需要判断加1后是否超过数组上界，若是则使其指向数组下界0。如果用i来表示Q.front或Q.rear，那么，这种循环意义上的加1运算可描述为：

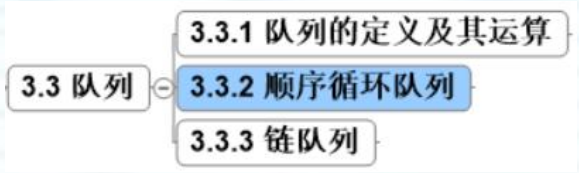
```
if ( i+1==QueueSize)                    //i表示 Q.rear或Q.front
      i=0;
else
      i=i+1;
```

可利用求余（%）运算将上述的操作简化为：

```
      i=(i+1)%  QueueSize;
```

在这样定义的循环队列中，出队元素的空间可以重新被利用。所以，一般情况下真正实用的顺序队列是循环队列。在循环队列的运算中，要涉及一些边界条件的处理问题。如图3.9所示的循环队列中，由于入队时的尾指针Q.rear向前追赶队头指针Q.front，出队时头指针向前追赶尾指针，如果是尾指针追赶上头指针，说明队满，否则若头指针追赶上尾指针，说明队空。因此，队列无论是空还是满，Q.rear==Q.front都成立。由此可见，仅凭队列的头尾指针是否相等是无法判断队列是“空”还是“满”的。解决这个问题有多种方法，常用的方法一般有三种：其一是另设一个标志位，以区别队列是“空”还是“满”；其二是设置一个计数器记录队列中元素个数；第三种方法是少用一个元素空间，约定入队前，测试尾指针在循环意义下加1后是否等于头指针，若相等则认为队列满，即尾指针Q.rear所指向的单元始终为空。

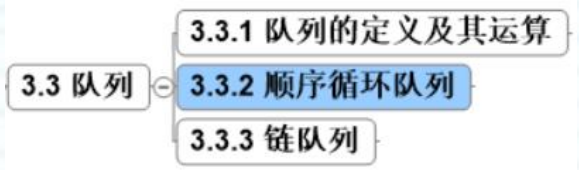
3.3.2 顺序循环队列



循环队列的顺序存储类型定义如下：

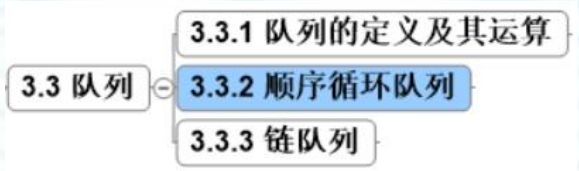
```
#define QueueSize 100
typedef char DataType ;                    //假设数据为字符型
typedef struct {
    DataType data [QueueSize];
    int front, rear;
} CirQueue;
```

3.3.2 顺序循环队列



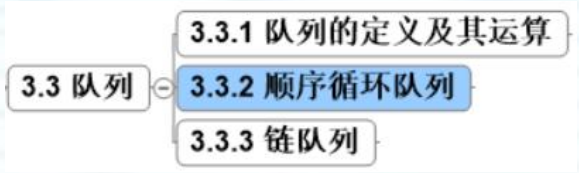
（1）置空队列

```
void InitQueue ( CirQueue *Q )
{
        Q->front=Q->rear=0;
}
```



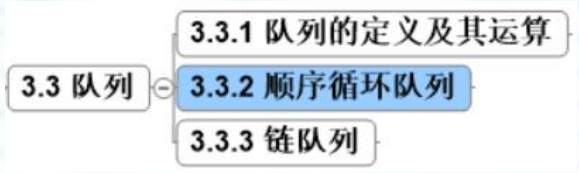
(2) 判队空

```
int QueueEmpty ( CirQueue *Q )
{
    return Q->rear==Q->front;
}
```



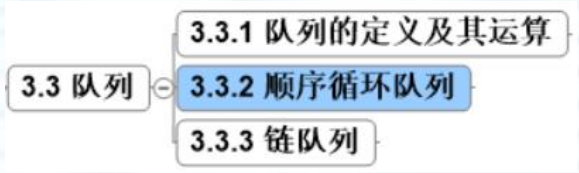
(3) 判队满

```
int QueueFull ( CirQueue *Q )
{
    return (Q->rear+1) % QueueSize == Q->front;
}
```



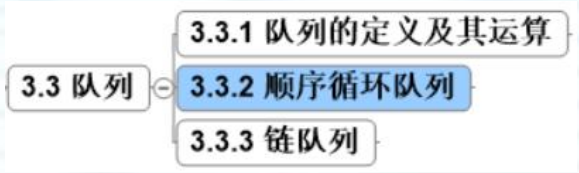
(4) 入队列

```
void EnQueue ( CirQueue *Q, DataType x)
{  //插入元素x为队列Q新的队尾元素
if ( QueueFull ( Q ) )
      printf ("Queue overflow");
else {
      Q->data[Q->rear]=x;
      Q->rear=(Q->rear+1) % QueueSize;  //循环意义下的加1
      }
}
```



（5）取队头元素

```
DataType GetFront ( CirQueue *Q )
{            //获取Q的队头元素值
    if ( QueueEmpty ( Q ) ) {
        printf ("Queue empty");
        exit ( 0 );                          //出错退出处理
}
    else
        return Q->data[Q->front]             //返回队头元素值
}
```



（6）出队列

```
DataType DeQueue ( CirQueue *Q )
{ //删除Q的队头元素，并返回其值
    DataType x;
    if ( QueueEmpty ( Q ) ) {
        printf ("Queue empty");
        exit(0);                          //出错退出处理
      }
    else {
        x=Q->data[Q->front];     //保存待删除元素值
        Q->front=(Q->front+1) % QueueSize;      //头指针加1
        return x;                         //返回删除元素值
      }
}
```

【例3.5】设Q是一个有11个元素存储空间的顺序循环队列，初始状态Q.front=Q.rear=0,写出下列操作后头、尾指针的变化情况，若不能入队，请说明理由。

d,e,b,g,h入队；　　d,e出队；

i,j,k,l,m入队；　　b出队；

n,o,p,q,r入队。

分析：本题的入队和出队的变化情况是这样的：

当元素d，e，b，g，h入队后，Q.rear=5，Q.front=0；

元素d.e出队，Q.rear=5，Q.front=2；

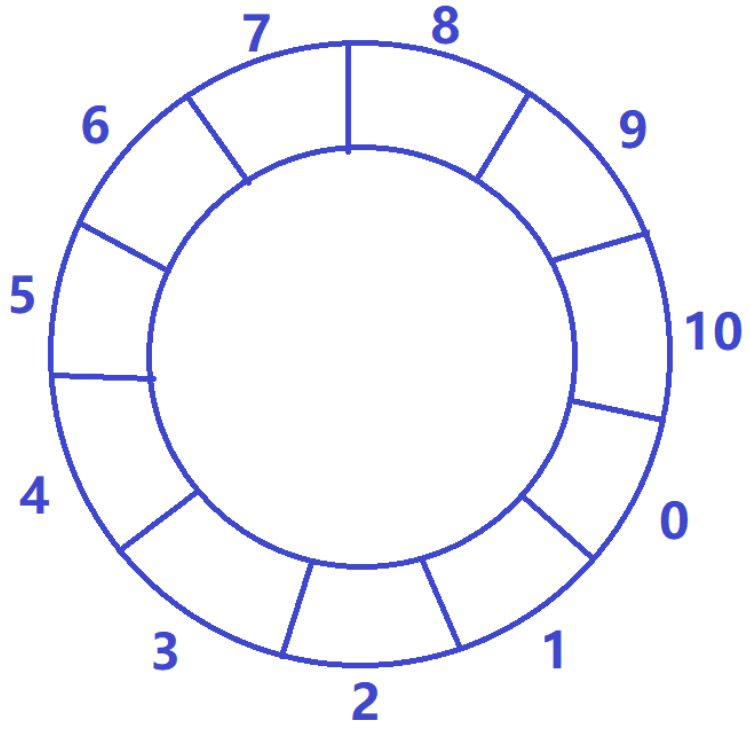
元素i，j，k，1，m入队后，Q.rear=10，Q.front=2;

元素b出队后，Q.rear=10，Q.front=3；

此时n，o，p入队，由于Q.rear=2，Q.front=3，

当q入队时，（Q.rear+1）% QueueSize==Q.front。

故队列满将产生溢出。



3.3.3三、链队列

链队列的类型定义如下：

```
typedef struct qnode {
    DataType data;
    struct qnode * next;
} QueueNode;                    //链队列结点类型
typedef struct {
    QueueNode * front;          //队头指针
    QueueNode * rear;           //队尾指针
} LinkQueue;                    //链队列类型
LinkQueue Q;                    //定义一链队列Q
```

3.3.3 链队列

链队列一般是不带头结点的，但和单链表类似，为了简化边界条件的处理，在队头结点之前也附加一个头结点，并设队头指针指向此结点。因此，空的链队列和非空链队列的结构如图3.10所示。

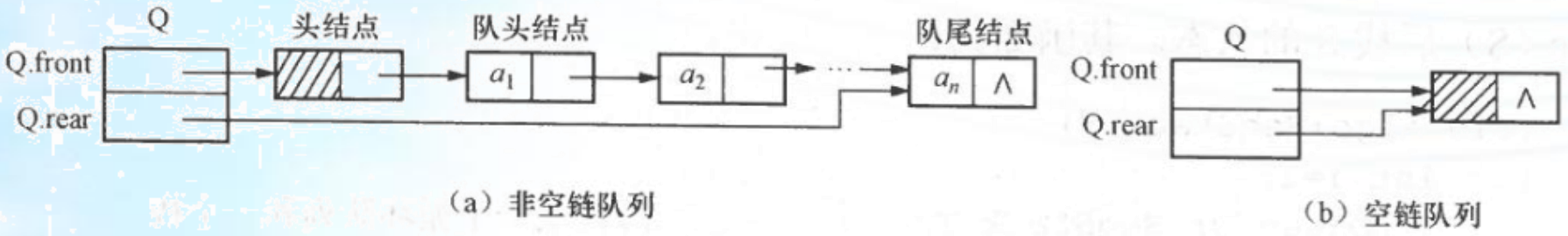


图 3.10　链队列和非空链队列结构示意图

3.3.3 链队列

下面给出的都是带头结点链队列的基本运算。

(1) 构造空队列

```
void InitQueue (LinkQueue * Q)
{
    Q->front=( QueueNode * ) malloc ( sizeof ( QueueNode ) ) ;  //申请头结点
    Q->rear=Q->front;                                  //尾指针也指向头结点
    Q->rear->next=NULL;
}
```

下面给出的都是带头结点链队列的基本运算。

（2）判队空

```
int QueueEmpty ( LinkQueue *Q )
{
    return Q->rear==Q->front;          //头尾指针相等队列为空
}
```

下面给出的都是带头结点链队列的基本运算。

（3）入队列

```
void EnQueue ( LinkQueue * Q, DataType x )
{   //将元素x插入链队列尾部
    QueueNode * p=(QueueNode * ) malloc ( sizeof ( QueueNode ) )        //申请新结点
    p->data=x;
    p->next=NULL;
    Q->rear->next=p;                          //*P链到原队尾结点之后
    Q->rear=p;                                //队尾指针指向新的队尾结点
}
```

下面给出的都是带头结点链队列的基本运算。

(4) 取队头元素

```
DataType GetFront ( LinkQueue * Q )
{   //取链队列的队头元素值
    if ( QueueEmpty ( Q ) ) {
        printf ("Queue underflow");
        exit(0);                                    //出错退出处理
    }
    else
        return Q->front->next->data;        //返回原队头元素值
}
```

（5）出队列

链队列的出队操作有两种不同情况要分别考虑。

①当队列的长度大于1时，则出队操作只需要修改头结点的指针域即可，尾指针不变，操作步骤如下：

```
s=Q->front->next;
Q->front->next=s->next;
x=s->data;
free(s);return x;                    //释放队头结点，并返回其值
```

（5）出队列

②若列队长度等于1，则出队时不仅要修改头结点指针域，而且还需要修改尾指针。

```
s=Q->front->next;
Q->front->next=NULL;
Q->rear=Q->front;
x=s->data;
free(s);return x;                                    //释放队头结点，并返回其值
```

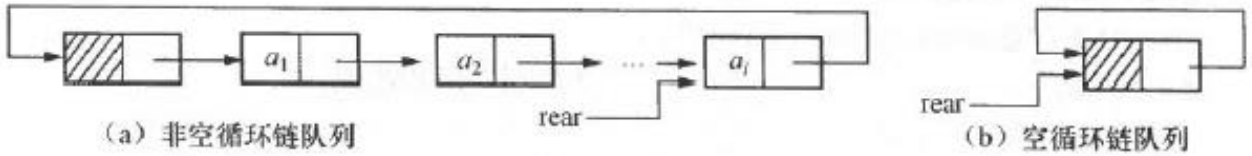
这样，在写算法时对于长度等于1和长度大于1的情况要分别处理。为了使得在长度等于1和长度大于1的情况下处理操作一致，可以改进出队算法，使得出队时只修改头指针，删除队列头结点（不是队头结点），使链队列的队头结点成为新的链队列的头结点。

3.3.3 链队列

链队列的出队算法描述如下：

```
DataType DeQueue ( LinkQueue * Q )
{   //删除链队列的头结点，并返回头结点的元素值
    QueueNode * p;
    if ( QueueEmpty ( Q ) ) {
        printf ("Queue underflow");
        exit(0);                                    //出错退出处理
    }
    else {
        p=Q->front;                                 //p指向头结点
        Q->front=Q->front->next;                    //头指针指向原队头结点
        free (p);                                   //删除释放原头结点
        return (Q->front->data);                    //返回原队头结点的数据值
    }
}
```

【例 3.7】 假设用一个带头结点的循环单链表表示队列（称为循环链队列），该队列只设一个指向队尾结点的指针 rear，不设头指针，试编写相应的初始化队列、入队（即插入）和出队（即删除）算法。循环链队列的结构如图 3.11 所示。



（a）非空循环链队列　　（b）空循环链队列

图 3.11　只设尾指针的循环链队列

按题目的已知条件和假设，该循环链队列的类型定义为：

```
typedef struct queuenode {
    DataType  data;
    struct  queuenode  * next ;
 }QueueNode;
QueueNode  * rear ;
```

分析：

（1）初始化空队列：

```
QueueNode * InitQueue(QueueNode * rear)
{
    rear=(QueueNode *)malloc(sizeof(QueueNode));          //申请头结点
    rear->next=NULL;
    return rear;
}
```

3.3.3 链队列

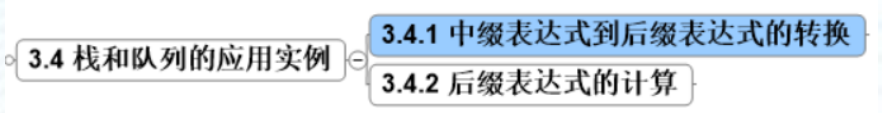
（2）入队列（插入一个结点）：在队列中插入一个结点操作是在队尾进行的，所以应在该循环链队列的尾部插入一个结点。插入的过程应该是：首先生成一个新结点 S，因为链表带头结点，所以队空与否对插入没有影响。插入操作是简单的：将尾的指针域值赋给新结点的指针域（即 s->next=rear->next）；把新结点指针 S 赋给原尾指针 rear 所指结点的指针域（即 rear->next=s）；再把 S 赋给 rear（即 rear=s）。因此，该入队算法如下：

```
void EnQueue(QueueNode * rear, DataType x)
{
    QueueNode * s=(QueueNode *)malloc(sizeof(QueueNode));//申请新结点
    s->data=x;
    s->next=rear->next;
    rear->next=s;
    rear=s;
}
```

（3）出队列（删除一个结点）：在队列中删除一个结点，首先要判断队列是否为空，若不为空，则可进行删除操作；否则，显示出错。删除的思想是将原表头结点删掉，把队头结点作为新的头结点，实现算法如下（要特别注意头结点和队头结点的区别）：

```
DataType DelQueue(QueueNode * rear)
{   QueueNode * s,*t;
    DataType x;
```

3.4第四节 栈和队列的应用实例　3.4.1一、中缀表达式到后缀表达式的转换



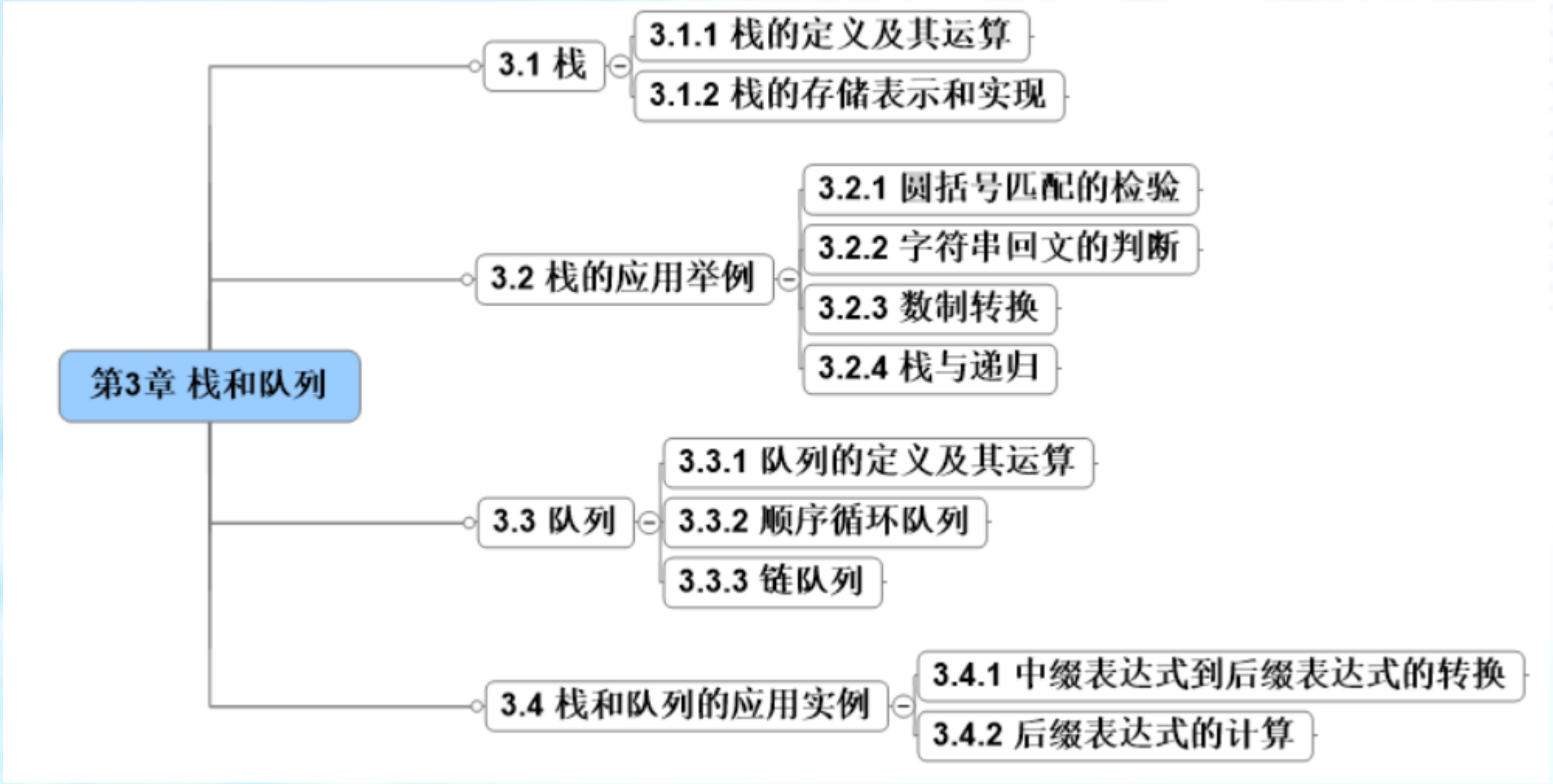
通过栈将中缀表达式转换为后缀表达式的算法思想如下：顺序扫描中缀算术表达式，当读到数字时，直接将其送至输出队列中；当读到运算符时，将栈中所有优先级高于或等于该运算符的运算符弹出，送至输出队列中，再将当前运算符入栈；当读入左括号时，即入栈；当读到右括号时，将靠近栈顶的第一个左括号上面的运算符全部依次弹出，送至输出队列中，再删除找中的左括号。

3.4.2二、后缀表达式的计算

在后缀表达式中，不仅不需要括号，而且还能完全免除算符优先规则。对于后缀表达式来说，仅仅使用一个自然规则，即从左到右顺序完成计算，这个规则对计算机而言是很容易实现的。下面将讨论如何用计算机来实现计算后缀表达式的算法。

如果在表达式中仅仅只有一个运算符，如像53*这样的表达式，显然计算过程非常简单，可立即进行。但后缀表达式在多数情况下都多于一个运算符，因此必须要像保存输入数字一样保存其中间结果。我们知道，在计算后缀表达式时，最后保存的值最先取出参与运算，所以要用到栈。利用前面生成的后缀表达式队列，很容易写出计算后缀表达式的算法。在算法中使用了整型栈S来存储读入的操作数和运算结果，因为在生成的后缀表达式队列中存放的是字符序列，因此在算法中要有一个数字字符到数值的转换。

本章总结


第3章 栈和队列
3.1 栈
3.1.1 栈的定义及其运算
3.1.2 栈的存储表示和实现
3.2 栈的应用举例
3.2.1 圆括号匹配的检验
3.2.2 字符串回文的判断
3.2.3 数制转换
3.2.4 栈与递归
3.3 队列
3.3.1 队列的定义及其运算
3.3.2 顺序循环队列
3.3.3 链队列
3.4 栈和队列的应用实例
3.4.1 中缀表达式到后缀表达式的转换
3.4.2 后缀表达式的计算

祝大家顺利通过考试！