第 2 章　线　性　表

2.1　线性表的概念

2.1.1　线性表的定义及特征

在日常生活中，为了简明扼要地表示一批数据之间的内在联系，通常都会选择“表格”。为了能够深入理解线性表的概念以及线性结构的特点，先从实际问题着手，分析“表格”中每一项之间的关系。

【例 2.1】新生入学时，为了能够让辅导员老师全面掌握学生的信息，学校为每一位辅导员老师提供一份“学生基本信息表”，其表格形式如表 2-1 所示。

表 2-1　学生基本信息表

学号	姓名	所在院系	班级	性别	生源地	出生年月日
20100501	张三	计算机	C10	男	吉林省	1990.12.16
20100502	李四	计算机	C09	男	黑龙江省	1989.01.06
20100503	王五	计算机	C11	女	河南省	1990.12.03
⋮	⋮	⋮	⋮	⋮	⋮	⋮

表 2-1 描述的是学生基本信息之间的关系，可以分析一下该表的整体特点。

1）表中每一列所包含的数据元素都是相同的，只是数值不同而已；表中的每一行代表一个学生的基本信息，若把每一行看作构成表的基本元素，则表中的所有元素都具有相同的性质。

2）表中的各行之间存在某种特殊关系，这种关系决定了表中各元素排列的顺序，从而确定了元素在表中的位置。表 2-1 为按照“学号”从小到大的排序，也可以按照班级等排序，但是表的整体结构就会发生变化。所以说，表中的元素之间存在某种特殊的关系，这种特殊关系决定表中元素的顺序以及位置。

3）表中的每一列（除描述属性行以外）元素都是顺序的，而且都具有唯一的头元素和唯一的尾元素。

上述三点是日常生活中使用的表格的共性，这些共性也就是线性结构的特点，即：

- 具有唯一的头元素；
- 具有唯一的尾元素；
- 除头元素外，集合中的每一个元素均有一个直接的前驱；
- 除尾元素外，集合中的每一个元素均有一个直接的后继；
- 具有反对称性和传递性。

线性表（Linear List）是由具有相同数据类型的 n(n≥0)个数据元素组成的一种有限的而且有序的序列，这些元素也可以称为结点或者表目，通常记为

$$(a_0,a_1,\cdots,a_{i-2},a_{i-1},a_i,\cdots,a_{n-1})$$

其中，n 为表长，n=0 时称为空表。由线性结构特点可知，a_0 是线性表唯一的头元素，a_{n-1} 是唯一的尾元素，a_{i-1} 是第 i 个元素；a_{i-2} 是 a_{i-1} 的直接前驱，a_i 是 a_{i-1} 的直接后继；当 2≤i≤n 时，a_{i-1} 只有一个直接的前驱，当 1≤i≤n−1 时，a_{i-1} 只有一个直接的后继。

线性表在实际生活中的例子很多，如英文字母表(A, B, …, Z)是线性表，表中的每一个字母是一个数据元素；扑克牌(A, 2, …, K)也是顺序表，表中的每一个数据元素代表牌面的值等。

2.1.2 线性表的抽象数据类型

抽象数据类型（Abstract Data Type，ADT）是指用以表示应用问题的数据模型以及定义在该模型上的一组操作。从抽象数据类型的观点来看，一种数据结构即为一个抽象数据类型。一个抽象数据类型由数据部分和操作部分两方面来描述，数据部分描述数据元素和数据元素之间的关系，操作部分根据定义的抽象数据类型应用的需要来确定。

下面用 C++类模板的方法，给出线性表类（名字为 List，模板参数为元素类型 T）的一个抽象数据类型说明。其中，每一个运算都用函数的接口指出其输入/输出参数以及其返回值类型。

【例 2.2】线性表的数据类型定义。

```
template <class T>
class List
{
    void Clear();                  //置空线性表
    bool IsEmpty();                //线性表为空时,返回 true
    bool Append(const T value);//在表尾添加元素 value,表的长度增加 1
    bool Insert(const int p, const T value);//在位置 p 插入元素 value,
表的长度增加 1
    bool Delete(const int p);                    //删除位置 p 上的元素,
表的长度减 1
    bool GetValue(const int p, T& value);        //把位置 p 上的元素值返
回到变量 value 中
    bool SetValue(const int p, const T value); //把位置 p 的元素值修改
为 value
    bool GetPos(int &p, const T value);          //把值为 value 的元素
的位置返回到变量 p 中
};
```

线性表的抽象数据类型并不是唯一的，要根据实际的应用来进行抽象。线性表运算

的具体实现与线性表在计算机内的物理存储结构有密切的关系，运算效率也与存储结构密不可分。下面介绍顺序和链式这两种常用的线性表存储结构。

2.1.3 线性表的存储结构

线性表的存储结构是指为线性表开辟计算机存储空间以及所采用的程序实现方法，本质上是逻辑结构到存储结构的映射。

线性表的存储结构主要有两类：

1）定长的顺序存储结构，又称向量型的一维数组结构，简称顺序表。程序中通过创建数组来建立这种存储结构，它的特点是线性表元素被分配到一块连续的存储空间，元素顺序存储在这些地址连续的空间中，数据元素之间是“物理位置相邻”的。定长的顺序存储结构限定了线性表长度的变化不得超过该固定长度，这是顺序表的不足之处。

2）变长的线性存储结构，又称链接式存储结构，简称链表。链接式存储结构使用指针，按照线性表的前驱和后继关系将各元素用指针链接起来。变长的线性存储结构对线性表的长度不加限制，当有新的元素加入线性表时，可以通过 new 语句向操作系统申请新的存储空间，并通过指针把新元素链接到合适的位置上。

线性表的这两种存储结构以及运算的具体实现将在 2.2 和 2.3 节中详细介绍。

2.1.4 线性表运算分类

按照运算（operation）所具有的特性，线性表的运算可分为以下 5 类：

1）创建线性表的一个实例。

2）线性表的析构函数~List()，释放该线性表实例在计算机中所占有的存储空间。

3）获取有关当前线性表的信息，包括由内容寻找位置、由位置寻找内容等，不改变线性表的内容。

4）访问线性表并改变线性表的内容或结构，例如添加或删除元素、更改指定位置的元素内容，清空线性表等。

5）辅助管理操作，例如求线性表的长度等。

2.2 顺 序 表

线性表的顺序存储结构是指用一组地址连续的存储空间依次存储线性表的数据元素。假设线性表的第一个元素 a_0 的存储地址为 $LOC(a_0)$，每个元素占用l个存储单元，则第 i 个元素 a_{i-1} 的存储位置为

$$LOC(a_{i-1}) = LOC(a_0) + (i-1) \times l \qquad 1 \leqslant i \leqslant n \tag{2.1}$$

线性表中第 i+1 个元素的存储位置为

$$LOC(a_i) = LOC(a_{i-1}) + l \qquad 1 \leq i \leq n-1 \tag{2.2}$$

如图 2-1 所示。

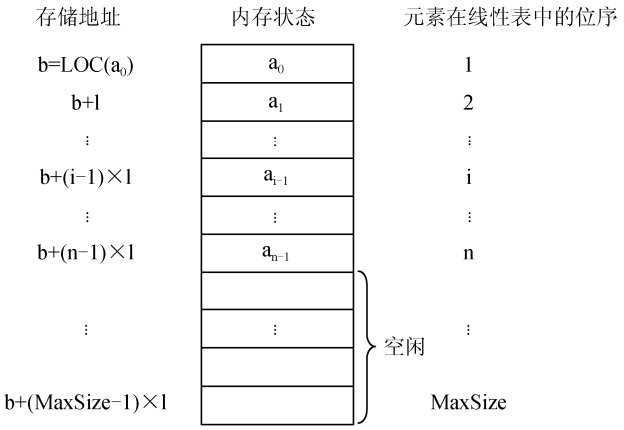


图 2-1　线性表的顺序存储结构示意图

把采用顺序存储结构的线性表简称为顺序表，也称为向量。顺序表有以下主要特征：

1）元素的数据类型是相同的。

2）元素顺序地存储在一片地址连续的存储空间中，一个元素按照存储顺序有唯一的索引值，又称为下标，可以随机存取表中的元素。

3）逻辑关系相邻的两个元素在物理位置上也相邻。

4）在程序中，数组变量说明语句一般使用常数作为向量长度，长度是静态常数，在程序执行时不变。

2.2.1　顺序表的实现

根据 2.1.2 节给出的线性表抽象数据类型，下面用 C++类模板描述线性表的一种顺序实现。

【例 2.3】顺序表的类定义。

```
template <class T>//线性表的元素类型为 T
class ArrayList : public List<T>    //定义顺序表 ArrayList
{
    public://顺序表的运算集
        ArrayList(const int size)   //创建顺序表,表长为最大长度
        {
            maxSize = size;
            arrayList = new T[maxSize];
            curLen = 0;
            position = 0;
        }

        ~ArrayList()                //析构函数,消除 ArrayList 的实例
        {
```

```
        delete [] arrayList;
    }

    void clear()                    //清空顺序表
    {
        delete [] arrayList;
        curLen = 0;
        position = 0;
        arrayList = new T[maxSize];
    }

    int Length();
    bool  Append(const  T  value);//在表尾添加元素 value,表的长度增加
1
    bool  Insert(const  int  p,  const  T  value);  //在位置 p 插入元素
value,表的长度增加 1
    bool  Delete(const int p); //删除位置 p 上的元素,表的长度减 1
    bool  GetValue(const int p, T& value);      //把位置 p 上的元素
值返回到变量 value 中
    bool  SetValue(const int p, const T value); //把位置 p 的元素值
修改为 value
    bool  GetPos(int &p, const T value);        //把值为 value 的
元素的位置返回到变量 p 中

private:                            //私有变量
    T *arrayList;                   //存储顺序表的实例
    int maxSize;                    //顺序表实例的最大长度
    int curLen;                     //顺序表实例的当前长度
    int position;                   //当前处理位置
};
```

（1）顺序表的插入

顺序表的插入操作是指在顺序表的第 $i-1$ 个数据元素和第 i 个数据元素之间插入一个新的数据元素 b，其结果使长度为 n 的顺序表 $(a_0,\cdots,a_{i-2},a_{i-1},\cdots,a_{n-1})$ 变为长度为 $n+1$ 的顺序表 $(a_0,\cdots,a_{i-2},b,a_{i-1},\cdots,a_{n-1})$，而且元素 a_{i-2} 和 a_{i-1} 在逻辑关系上也发生了变化。在顺序表的顺序存储结构中，逻辑上相邻的数据元素物理地址也是相邻的，所以需要将第 i～n（共 $n-i+1$）个位置上的元素向后移动一个位置，这样才能反映这个逻辑关系的变化。

由于顺序表是长度固定的线性结构，因此对顺序表进行插入运算时还需要检查顺序表中实际元素的个数，以免因为插入操作而发生溢出现象（即超过所允许的最大长度 maxSize 的值）。在顺序表的位置 p 上插入一个值为 value 的元素，其插入操作如图 2-2 所示，算法实现在例 2.3 中给出。

a_1	a_2	…	a_i	…	…	a_n	…	…
0	1	…	i–1	…	…	n–1	…	maxsize–1

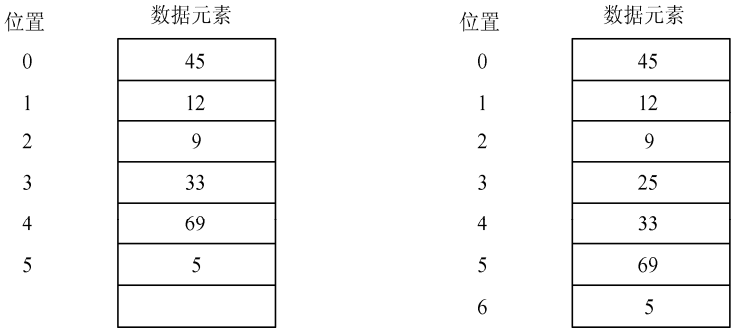
a_1	a_2	…	value	a_i	…	…	a_n	…	…
0	1	…	p	i	…	…	n	…	maxsize−1

图 2-2　顺序表元素插入示意图

【例 2.4】在顺序表的某个位置插入元素。

```
template <class T>                              //顺序表的元素类型为 T
bool ArrayList<T> :: Insert(const int p, const T value)
{
    if(curLen >= maxSize)                       //检查顺序表是否溢出
    {
        cout << "The List is overflow" << endl;
        return false;
    }

    if(p < 0 || p > curLen)                     //检查插入位置是否合法
    {
        cout << "Insertion point is illegal" << endl;
    }

    for(int i = curLen; i > p; i--)
    {
        arrayList[i] = arrayList[i-1];          //从表尾 curLen-1 处向后移
动一个位置直到插入位置 p
    }
    arrayList[p] = value;                       //位置 p 处插入新元素
    curLen++;                                   //表的实际长度增加 1
    return true;
}
```

如图 2-3，为了将元素 25 插入到元素 33 前面，则需要将位置 3，4，5 上的元素向后移一个位置，顺序表的长度变为 7。



（a）插入前 n=6　　（b）插入后 n=7

图 2-3　顺序表插入前后变化情况

该算法的执行时间主要消耗在元素的移动操作上。最好的情况下，插入位置为当前线性表的尾部，此时移动次数为 0；最坏的情况下，插入位置为线性表的首部，此时所

有 n 个元素均需移动。一般情况下，插入位置为 i 时需要移动其后的 n−i 个元素。假设在各个位置的插入概率相等，均为 $p=1/(n+1)$，则平均移动元素次数为：

$$\sum_{i=0}^{n} p\times(n-i)=\frac{1}{n+1}\sum_{i=0}^{n}(n-i)=\frac{n}{2}$$

即等概率情况下插入算法平均需要移动顺序表中元素个数的一半，总的时间开销与表中元素个数成正比，为 $O(n)$。

（2）顺序表的删除

与顺序表的插入操作相反，顺序表的删除操作是指在顺序表中删除第 i 个数据元素 a_{i-1}，使长度为 n 的顺序表 $(a_0,\cdots,a_{i-2},a_{i-1},a_i,\cdots,a_{n-1})$ 变为长度为 $n-1$ 的顺序表 $(a_0,\cdots,a_{i-2},a_i,\cdots,a_{n-1})$，并且 a_{i-2}、a_{i-1} 和 a_i 之间的逻辑关系也会发生变化，需要把第 $i+1\sim n$ 个元素（共 $n-i$）个元素依次向前移动一个位置。

删除运算需要事先检查顺序表是否为空，只有在非空表时才能进行删除操作。在顺序表中删除位置 i 上的元素，并且位置 $i+1$ 到 $curLen-1$ 的元素依次向前移动一个位置，其删除操作如图 2-4 所示，算法实现在例 2.4 中给出。

a_1	a_2	…	a_i	a_{i+1}	…	a_n	…	…
0	1	…	i−1	i	…	n−1	…	maxSize−1

a_1	a_2	…	a_{i+1}	…	a_n	…	…
0	1	…	i-1	…	n−2	…	maxSize−1

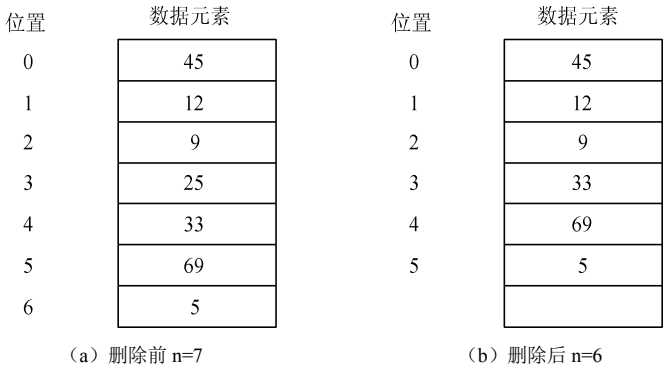
图 2-4　顺序表元素删除示意图

【例 2.5】删除顺序表中给定位置的元素。

```
template <class T>                              //顺序表的元素类型为 T
bool ArrayList<T> :: Delete(const int p)
{
    if(curLen <= 0)                             //检查顺序表是否为空
    {
        cout << "No element to delete" << endl;
        return false;
    }

    if(p < 0 || p > curLen -1)                  //检查删除位置的合法性
    {
        cout << "Deletion is illegal" << endl;
        return false;
    }

    for(int i = p; i < curLen - 1; i++)
    {
        arrayList[i] = arrayList[i+1];          //从删除位置 p 开始每个元素向前移动一个位置直到表尾
```

```
        }
        curLen--;                                    //表的实际长度减 1
        return true;
    }
```

如图 2-5 所示，删除位置 3 上的元素 25，需要将位置 4，5，6 上的元素依次向前移动一个位置，顺序表长度变为 6。



（a）删除前 n=7　　（b）删除后 n=6

图 2-5　顺序表删除前后变化情况

删除操作的时间复杂度分析与插入操作的时间复杂度分析类似，其平均时间复杂度为O(n)。顺序表按照位置读取元素非常方便，时间复杂度为O(l)，插入、删除和按内容查找的平均时间复杂度均为O(n)。

2.2.2　多维数组

从逻辑结构上看，多维数组可以认为是一维数组（向量）的扩充；但是从物理结构上看，一维数组是多维数组的特例。多维数组具有复杂的元素存储位置和计算公式。一般将多维数组简称为数组（Array）。

数组是由下标和值组成的序对集合。在数组中，一旦给定下标就存在一个与其对应的值，成为数组元素，每个数组元素都必须属于同一个数据类型。令 n 为数组的维数，当 n=1 时，n 维数组就退化为定长的线性表。反之，n 维数组可以看成是线性表的推广。因此可以得到如下定义：一维数组是一个向量，它的每一个元素是这个结构中不可分割的最小单位。n (n >1) 维数组是一个向量，它的每一个元素是 n−1 维数组，且具有相同的上限和下限。

由上述定义可以把二维数组看成是这样一个定长的线性表：它的每个元素也是一个定长的线性表。

图 2-6（a）是一个二维数组，以 m 行 n 列的矩阵形式表示，其中每个元素是一个列向量形式的线性表，如图 2-6（b）所示；或者每个元素是一个行向量形式的线性表，如图 2-6（c）所示。即二维数组中的每个元素 a_{ij} 都属于两个向量：第 i 行的行向量和第 j

列的列向量。a_{00} 是开始结点，没有前驱结点；$a_{m-1,n-1}$ 是终端结点，没有后继结点；边界上结点 $a_{0,j}(j=1,\cdots,n-1)$ 和 $a_{i,0}(i=1,\cdots,m-1)$ 只有一个前驱结点，$a_{m-1,j}(j=0,\cdots n-2)$ 和 $a_{i,n-1}(i=0,\cdots,m-2)$ 只有一个后继结点；其余每个元素 a_{ij} 有两个前驱结点 $a_{i-1,j}$ 和 $a_{i,j-1}$，两个后继结点 $a_{i+1,j}$ 和 $a_{i,j+1}$。

$$A_{m\times n}=\begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{m-1,0} & a_{m-1,1} & a_{m-1,2} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

（a）矩阵形式表示

$$A_{m\times n}=\begin{bmatrix}\begin{bmatrix} a_{00} \\ a_{10} \\ \vdots \\ a_{m-1,0} \end{bmatrix}\begin{bmatrix} a_{01} \\ a_{11} \\ \vdots \\ a_{m-1,1} \end{bmatrix}\cdots\begin{bmatrix} a_{0,n-1} \\ a_{1,n-1} \\ \vdots \\ a_{m-1,n-1} \end{bmatrix}\end{bmatrix}$$

（b）列向量的一维数组

$$A_{m\times n}=((a_{00},a_{01},\cdots,a_{0,n-1}),(a_{10},a_{11},\cdots,a_{1,n-1}),\cdots,(a_{m-1,0},a_{m-1,1},\cdots,a_{m-1,n-1}))$$

（c）行向量的一维数组

图 2-6　二维数组

以此类推，三维数组 A 可以视为以二维数组为元素的向量，四维数组可以视为以三维数组为元素的向量，……。

多维数组的逻辑特征是：一个元素可能有多个直接前驱和多个直接后继。

（1）数组顺序表的定义

把数组中的元素按照逻辑次序存放在一组地址连续的存储单元的方式称为数组的顺序存储结构，采用这种存储结构的数组称为数组顺序表。

由于内存单元是一维结构，而数组是个多维结构，因此用一组连续存储单元存放数组的元素存在一个次序问题。例如图 2-6（a）所示的二维数组可以看成图 2-6（b）所示的一维数组也可以看成图 2-6（c）所示的一维数组。所以，对二维数组有两种顺序存储方式：列优先顺序表和行优先顺序表。

（2）列优先顺序表

以列为主序的数组顺序表，是将数组元素按照列向量排序，第 i+1 个列向量紧接在第 i 个列向量的后面，即按列优先，逐列顺序存储。它又称为列优先数组顺序表，简称列优先顺序表。

列优先顺序表推广到 n 维数组，可以规定为最左的下标优先存储，从左到右。

（3）行优先顺序表

以行为主序的数组顺序表，是将数组元素按照行向量排序，第 i+1 个行向量紧接在第 i 个行向量的后面，即按行优先，逐行顺序存储。它又称为行优先数组顺序表，简称行优先顺序表。

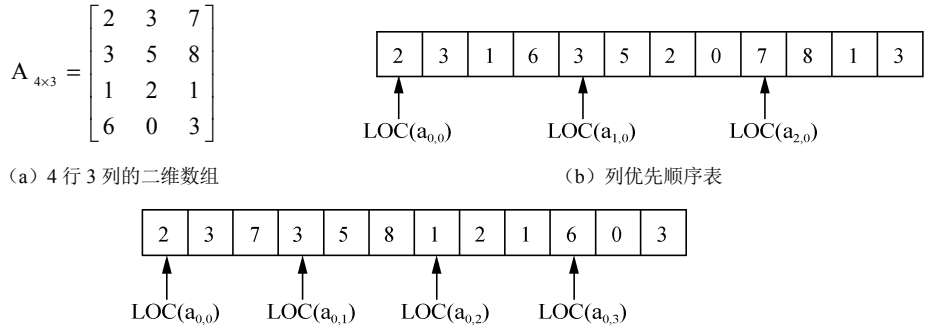
行优先顺序表推广到 n 维数组，可以规定为最右的下标优先存储，从右到左。

图 2-7（a）给出数组 A 的一个 4 行 3 列的二维数组，图 2-7（b）给出 A 的列优先

顺序表，图 2-7（c）为 A 的行优先顺序表。

（4）数组顺序表的定位公式

对于数组，一旦规定了其维数和各维的长度，便可以为它分配存储空间。反之，数组存放的起始地址、数组行号和列号，以及每个数组元素所占用的存储单元，便可以求得给定下标的数组元素存储位置的起始位置。下面给出行优先顺序表的定位公式。



（a）4 行 3 列的二维数组　　（b）列优先顺序表

（c）行优先顺序表

图 2-7　二维数组的两种存储方式

假设每个元素占 l 个存储单元，则二维数组 A 中的任一元素 a_{ij} 的存储地址可以定义如下：

$$LOC(a_{ij}) = LOC(a_{00}) + (b_2 \times i + j) \times l \tag{2.3}$$

其中，$LOC(a_{ij})$ 是 a_{ij} 的存储地址；$LOC(a_{00})$ 是 a_{00} 的存储地址，即二维数组 A 的起始存储地址，也称为基地址或基址；b_2 是数组第二维的长度。

同理，可以推出 n 维行优先顺序表的元素存储地址的计算公式如下：

$$\begin{aligned} LOC(a_{j_1,j_2,\cdots,j_n}) &= LOC(a_{0,0,\cdots,0}) + (b_2 \times \cdots \times b_n \times j_1 + b_3 \times \cdots \times b_n \times j_2 + \cdots + b_n \times j_{n-1} + j_n) \\ &= LOC(a_{0,0,\cdots,0}) + \left(\sum_{i=1}^{n-1} j_i \prod_{k=i+1}^{n} b_k + j_n \right) \times l \end{aligned} \tag{2.4}$$

式（2.4）可以缩写成如下形式：

$$LOC(a_{j_1,j_2,\cdots,j_n}) = LOC(a_{0,0,\cdots,0}) + \sum_{i=1}^{n} c_i j_i \tag{2.5}$$

其中，$c_n = l$，$c_{i-1} = b_i \times c_i$，$1 < i \leqslant n$。

由式（2.5）很容易看出，一旦确定了数组各维的长度，c_i 就是常数，因此，数组元素的存储地址是其下标的线性函数。

（5）数组顺序表的特点

如果计算各个元素存储地址的时间相等，则存取数组中的任一元素的时间也相等。具有这一特点的存储结构称为随机存储结构。因此，数组顺序表就是一个随机存储结构。

2.3 链　　表

尽管顺序表是一种非常有用的数据结构，但是其至少存在以下两个方面的局限：

1）改变顺序表的大小需要重新创建一个新的顺序表并把原有的数据都复制过去。

2）顺序表通过物理位置上的相邻关系来表示线性结构的逻辑关系，插入、删除元素平均需要移动一半的元素。

为了克服顺序表无法改变长度的缺点，并满足许多应用经常插入新结点或删除结点的需要，产生了链表这样的数据结构。

链表可以看成一组既存储数据又存储相互连接信息的结点集合。这样，各结点不必如顺序表那样存放在地址连续的存储空间，可以散放在存储空间的各处，而由称为指针的域来按照线性的后继关系链接结点。链表的特点是可以动态地申请内存空间，根据线性表元素的数目动态地改变其所需要的存储空间。在插入元素时申请新的内存空间，删除元素时释放其占有的存储空间。

链接存储是最常用的存储方法之一，它不仅可以用来表示线性表，也常常用于其他非线性的数据结构。例如，后几章中讨论的树结构和图结构，其中很多是使用结点的链接存储方式。本节主要讨论几种用于线性表的链接存储结构：单链表、双链表、循环链表等，统称为链表。

2.3.1 链表的实现

（1）单链表

单链表是通过指针把它的一串内存结点链接成一个链，这些内存结点两两之间地址不必相邻，如图 2-8 所示。为此，它的存储结点由两部分组成：第一部分存放线性表结点的数据，称为 data 字段；另一部分存放指向后继结点的指针，称为 link 字段。对于没有后继结点的终止结点而言，其 link 域为空指针 NULL（在图中用“∧”表示）。

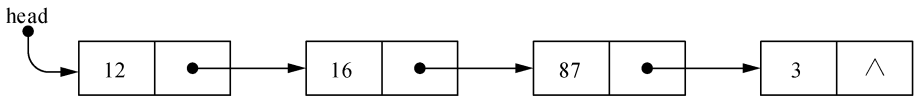


图 2-8　单链表示例

【例 2.6】单链表的结点定义。

```
template<class T>
class LinkNode
{
    public:
        T data;                              //数据域
        LinkNode<T>*link;                    //指向后继指针的结点
        LinkNode(const T&el, LinkNode<T>*ptr = 0){ //构造函数
```

```
            data=el;
            link=ptr;
        }
    };
```

由于单链表中的结点是一个独立的对象，为了方便复用，故将其定义为一个独立的类。LinkNode 是由其自身来定义的，因为其中的 link 域指向正在定义的类型本身，这种类型称为自引用型。由于在 2.4 节栈和 2.5 节队列中也要用到 LinkNode 类，故将其数据成员声明为公有的。

用一个指向表首的变量 head 存放指向单链头结点的指针。由于单链表的每个结点存储地址并不连续，因此在访问的时候，只能从头指针开始沿着结点的 link 域来进行。例如图 2-8 中，head ->data = 12；而 head-> link->data = 16。一个线性表的元素个数越多，则这个单链表越长。为了加速对表尾元素的访问，往往会使用一个表尾变量 tail 来存放指向单链表尾结点的指针。如图 2-9 所示，对于单链表的访问只可通过头、尾指针来进行。

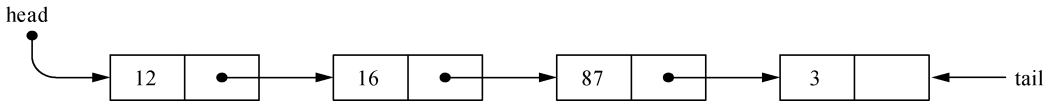
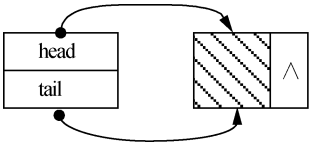
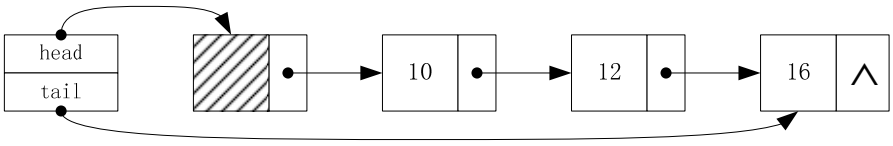


图 2-9　具有头、尾指针的单链表

为了便于实现，为每个单链表加上一个“头结点”，它位于单链表的第一个结点之前。头结点的 data 域可以不存储任何信息，也可以存放一个特殊标志或表长。如图 2-10（a）所示为一个带头结点的空链表，图 2-10（b）则为一个带有头结点的非空链表。只要表存在，它必须至少有一个头结点。



（a）空链表



（b）非空链表

图 2-10　带有头结点的单链表

使用带头结点的单链表，将有以下两个好处：①对带头结点的单链表，在表的任何

结点之前插入结点或者删除表中的任何结点，所要做的都是修改前一结点的指针域。若单链表没有头结点，则首结点没有前驱结点，在其前插入结点或删除该结点时要作为特殊情况专门处理；②对带头结点的单链表，表头指针是指向头结点的非空指针，因此空链表与非空链表的处理是一样的。

下面给出这种带有头结点的单链表的定义。根据链表的实际应用特点，在其中增加了一个尾指针 tail，一个返回链表长度的函数 getSize ()和一个返回指定结点位置的指针的函数 setPos()。

【例 2.7】单链表的类型定义。

```
template<class T>
class LinkList
{
    private:
        LinkNode<T> *head, *tail;       //表头和表尾指针
        LinkNode<T> *prevPtr, *currPtr;//记录表当前遍历位置的指针，由插入和删除操作更新
        int position;                   //当前元素在表中的位置序号，由函数 reset 使用
    public:
        LinkList();
        ~LinkList();
        int getSize()const;             //返回链表中的元素个数
        bool isEmpty()const;            //链表是否为空
        void reset(int pos = 0);//初始化指针的位置（第一位数的位置设为 0)
        void next();                    //使指针移动到下一个结点
        bool endOfList()const;          //指针是否到了链尾
        int currentPosition(void);      //返回指针当前的位置
        void insertHead(const T&item); //在表头插入结点
        void insertTail(const T&item); //在表尾添加结点
        void insertAt(const T&item);    //在当前结点之前插入结点
        void insertAfter(const T&item);//在当前结点之后插入结点
        T deleteHead();                 //删除头结点
        void deleteCurrent();           //删除当前结点
        T&data();                       //返回对当前结点成员数据的引用
        const T&data()const;    //返回对当前结点成员数据的常引用
        void clear();           //清空链表：释放所有结点的内存空间
        LinkNode<T>* setPos(int pos);  //返回指定位置 pos 的指针
        bool insertPos(const int i, const T value);//在指定位置插入结点
        bool deletePos(const int i);             //删除指定位置的结点
};
```

对于单链表，最常用的运算为检索、插入和删除，下面分别来介绍这 3 种算法。

1）单链表的检索。由于单链表存储地址空间的不连续，单链表无法像顺序表那样直接通过结点的位置来确定其地址，需要从头指针 head 所指的首结点开始沿 link 域，逐个结点进行访问。

根据单链表的结构特点，只要有指向某一个结点的指针，便可通过该指针访问此结点。也就是说，按照位置检索只要返回指向该位置的指针即可。代码 2.7 给出了在单链表中查找第 i 个结点的代码，并返回指向该结点的指针。

【例 2.8】返回指定位置 pos 的指针。

```
template<class T>
LinkNode<T> * LinkList<T>::setPos(int pos)
{
    if(pos == -1)       //i 为-1 则定位到头结点
        return head;
    int count = 0;
    LinkNode<T> *p = head->link;
    while(p != NULL && count < pos)
    {
        p = p->link;
        count++;
    }
    return p;           //指向第 i 个结点，当链表长度小于 i 时返回 NULL
}
```

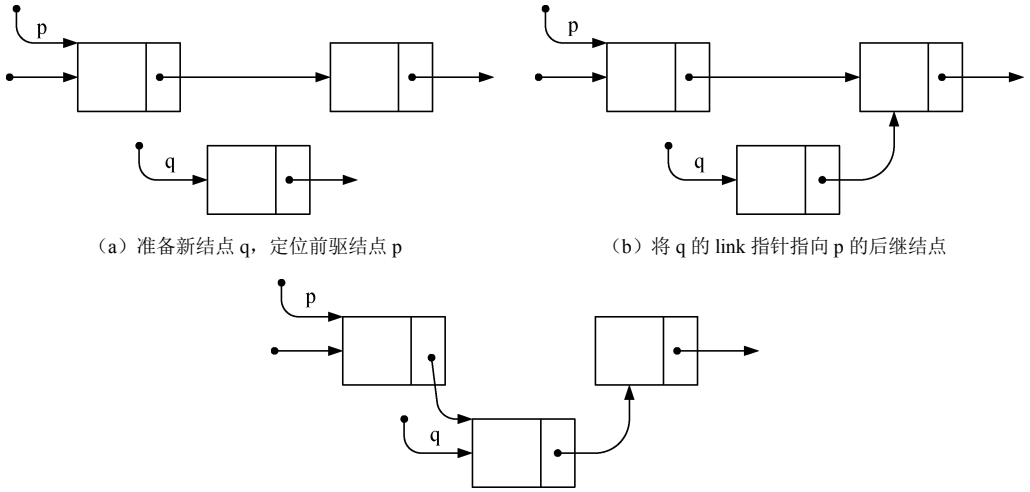

链表中第 i 个结点是按照 C/C++的数组下标编号规则，从 0 到 n−1，头结点的编号为−1。当单链表实际长度小于给定的 i 时，返回 NULL，当 i 为−1 时返回指向头结点的指针。在链表上基于位置检索需要从链表的第一个结点开始移动，直到找到第 i 个位置，所以平均需要 O(n)的时间。

2）单链表的插入。由于单链表的结点之间的前驱关系和后继关系由指针来表示，因此在插入或删除结点时，维护结点之间逻辑关系只需要改变相关结点的指针域，而不必像顺序表那样进行大量的数据元素的移动。

在单链表中插入结点还涉及到存储管理的问题。在单链表中插入一个新的结点时，可以使用 new 命令为新结点开辟存储空间。与 new 对应，可使用 delete 命令释放从单链表中删除的结点所占用的空间，否则这些被占用的空间会变成存储空间中无法利用的垃圾，影响再利用。new 和 delete 是 C++程序语言为动态存储管理提供的两个重要的命令，C 语言标准函数库中提供的 malloc()和 free()函数具有同样的功能。

向单链表中插入一个新元素的操作，具体包括创建一个新结点（并赋值）和修改相关结点的链接信息以维护原有的前驱后继关系。由于单链表没有指向前驱的指针，因此

在第 i 个位置插入结点时，必须先获得位置 i−1 的指针。对于 n 个结点的线性表，插入点可以有 n+1 个；i = 0 表示在表头插入，setPos(i−1)将返回头结点 head，新结点直接插入到头结点 head 之后，成为表中第一个结点；i = n 表示在表尾插入。相应的插入过程如图 2-11 所示，例 2.8 为单链表插入运算的实现方法。



（a）准备新结点 q，定位前驱结点 p　　（b）将 q 的 link 指针指向 p 的后继结点

（c）将 p 的 link 指针指向 q

图 2-11　单链表插入运算

【例 2.9】插入单链表的第 i 个结点。

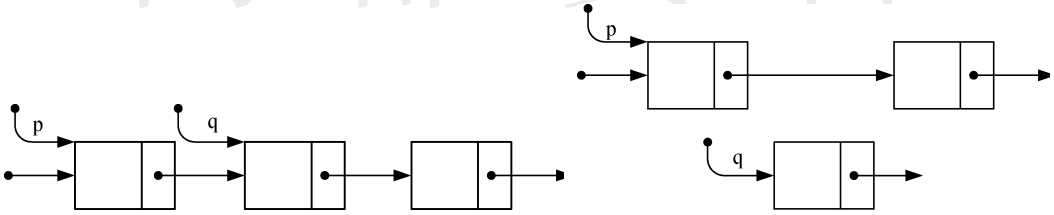
```
template<class T>
bool LinkList<T>::insertPos(const int i, const T value)
{
    LinkNode<T> *p, *q;
    if((p = setPos(i - 1)) == NULL)     //p 是第 i 个结点的前驱
    {
        cout << "插入操作不允许" << endl;
        return false;
    }

    q = new LinkNode<T>(value, p->link);
    p->link = q;
    if(p == tail)                       //在表尾进行插入操作
        tail = q;
    return true;
}
```

3）单链表的删除。与插入算法相同，从表中删除一个结点也需要修改该结点的前驱结点的指针域来维持结点间的线性关系，同时需要释放被删结点所占用的内存，以免发

生“丢失”。例 2.9 给出了删除单链表第 i 个结点的代码，相应的操作见图 2-12。

【例 2.10】删除单链表的第 i 个结点。

```
template<class T>
bool LinkList<T>::deletePos(const int i)
{
    LinkNode<T> *p, *q;
    if((p = setPos(i - 1)) == NULL || p == tail)   //待删除点不存在
    {
        cout << "非法删除点" << endl;
        return false;
    }

    q = p->link;                                   //q 为真正待删除点
    if(q == tail)                                  //删除点为表尾，修改尾指针
    {
        tail = p;
        p->link = NULL;
        delete q;
    }
    else if(q != NULL)                             //删除结点 q，并修改指针
    {
        p->link = q->link;
        delete q;
    }
    return true;
}
```

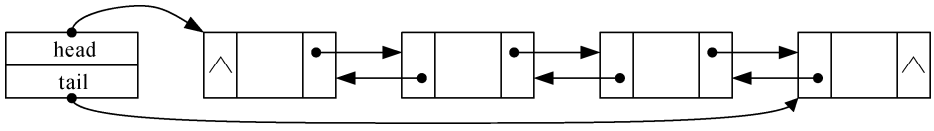
（a）删除前指针情况　　（b）删除后指针情况

图 2-12　单链表删除运算

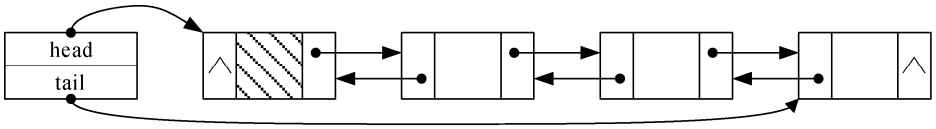
从例 2.8 和例 2.9 可以看出，尽管插入（删除）操作本身可在常数时间内完成结点的创建（释放）和链接信息的修改，但在位置 i 上进行插入（删除）操作的时候，首先需要检索到位置 i-1 的结点，而定位操作的平均时间复杂度为 O(n)，这也是在某些应用中需要在抽象数据类型中增加一个表示当前位置的成员的原因，因为通常插入和删除均在当前位置上进行。

（2）双链表

单链表的主要不足之处在于其指针域仅指向其后继结点，因此从一个结点不能有效地找到其前驱，而必须从表首开始顺着 link 域逐一查找。这对于长链表而言，代价相当可观。为此，引入双链表，其基本结构是在每个结点中增加一个指向前驱结点的指针，形成的双链表如图 2-13 所示。



（a）不带头结点的双链表



（b）带有头结点的双链表

图 2-13　双链表

图 2-13 中指针 head 用于指向表首结点，指针 tail 用于指向表尾结点。图 2-14 给出了相应结点的图示，其中 next 表示指向后继结点的指针，prev 表示指向前驱结点的指针。线性表的链式实现究竟采用单链表还是双链表对 List 类的用户而言应该是透明的。下面给出双链表结点 DLLNode 类的定义和实现。

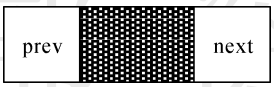


图 2-14　双链表的结点

【例 2.11】双链表的结点定义和实现。

```
template<class T>
class DLLNode
{
    public:
        T data;              //保存结点元素的内容
        DLLNode<T> *next;    //指向后继结点的指针
        DLLNode<T> *prev;    //指向前驱结点的指针

           DLLNode(const T info, DLLNode<T> *prevVal = NULL, DLLNode<T>
*nextVal = NULL)                    //构造函数
            {
                data = info;
                prev = prevVal;
                next = nextVal;
            }
            DLLNode(DLLNode<T> *prevVal = NULL, DLLNode<T> *nextVal =
```

```
NULL)
                                        //给定前后指针的构造函数
        {
            prev = prevVal;
            next = nextVal;
        }
};
```

与单链表的基本运算相比，双链表的运算更复杂一些，因为双链表需要维护两个指针域。下面以插入和删除两种基本运算为例进行讨论。

双链表中要在 p 所指结点后插入一个新的结点 q，具体操作步骤如下：

① 执行 new q 开辟结点空间。

② 填写结点的数据域信息。

③ 填写结点在双链表中的链接关系，即

```
q->prev = p;
q->next = p->next;
```

④ 修改 p 所指结点及后继结点在新结点插入后的链接信息，即把新结点的地址填入原 p 所指结点的 next 域以及新结点后继的 prev 域，即

```
p->next = q;
q->next->prev = q;
```

插入过程示意图见图 2-15。

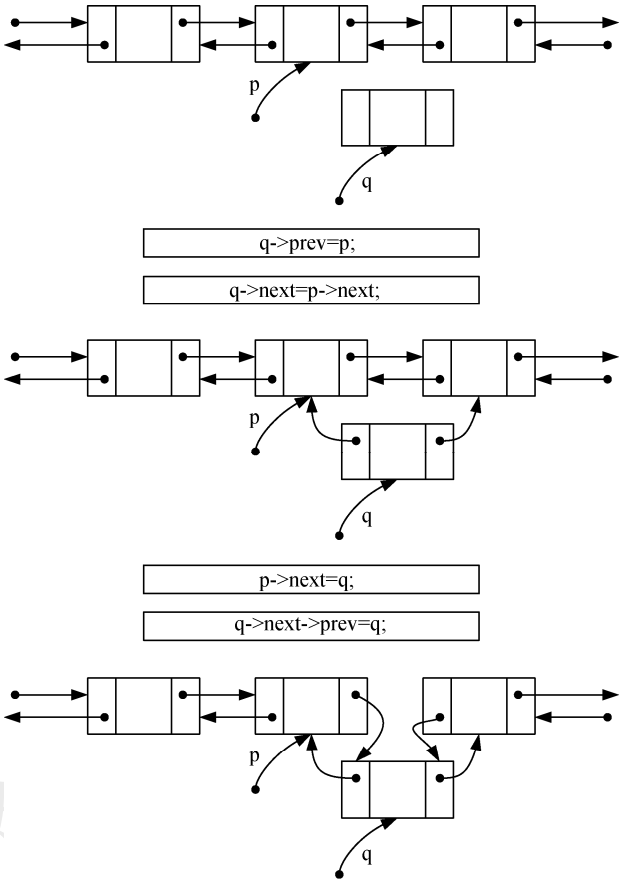


图 2-15　双链表插入运算

同样，如果要删除双链表中的一个结点，只需要修改该结点的前驱 prev 域和该结点的后继 next 域。例如，要删除指针变量 p 所指的结点，需要按如下操作进行。

①
```
p->prev->next = p->next;
p->next->prev = p->prev;
```

② 然后把变量 p 的前驱和后继置空，再释放 p 所指空间即可。

```
p->next = NULL;
p->prev = NULL;
delete p;
```

删除操作示意图见图 2-16。

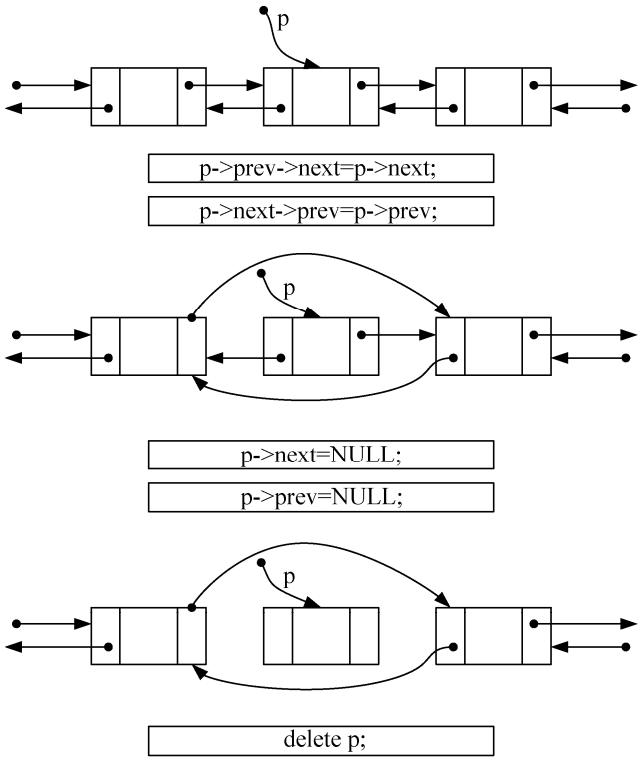
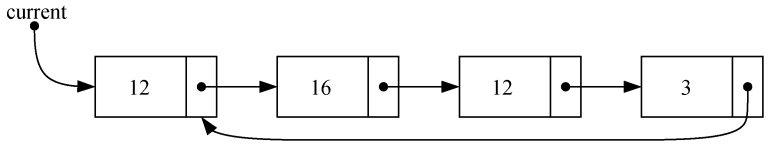


图 2-16　双链表删除运算

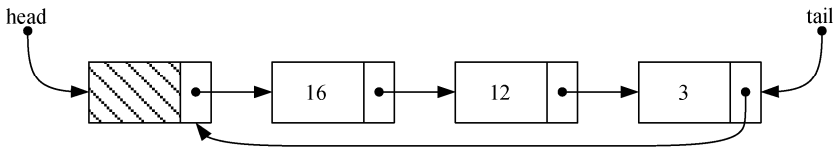
（3）循环链表

在某些情况下，需要把结点组成环形链表。例如，有多个进程在同一段时间内访问同样的资源，为了保证每一个进程可以公平地分享这个资源，可以把这些进程组织在如图 2-17（a）所示的称为循环链表的结构中。可以通过指针 current 访问该结构，current 指向的结点即为将要激活的进程。随着 current 指针的移动，可以激活每一个进程。

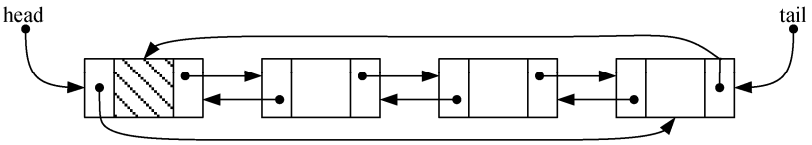
只要将单链表的形式稍作修改，使其最后一个结点的指针不为 NULL，而是指向表首结点，就成为一个循环链表。类似地，也可以将双链表的头结点和尾结点链接起来，这样不会增加额外存储开销，却给很多操作带来方便。使用循环链表的主要优点是：从循环链表中的任一点出发，都能访问到表中的其他结点。下面给出几种循环链表示意图，如图 2-17 所示。



（a）不带头结点的循环链表



（b）带有头结点的循环链表



（c）带有头结点的循环双链表

图 2-17　几种循环链表

2.3.2　线性表实现方法的比较

1．顺序表的主要优点

1）顺序表的存储密度为 1，存储利用率很高，因为表示数据元素之间的逻辑关系无需占用附加空间。

2）顺序表存取操作方便，可以从前向后顺序存取，也可以从后向前顺序存取，还可以按照元素序号（下标）直接存取，因此顺序表查找速度快，其时间复杂度为 O(1)。

2．线性链表的主要优点

1）线性链表无需事先定义链表的最大长度，只要存储有空间可供分配，就可以增加链表长度，不存在存储溢出的问题。

2）线性链表插入和删除操作方便，只需要修改链接指针，无需大量元素移动，其时间复杂度为 O(1)。

当线性表中经常插入、删除内部元素时，不宜使用顺序表，因为顺序表的插入和删除操作涉及大量的元素移动，其平均时间复杂度为 O(n)，n 为顺序表长度。当线性表中经常需要按位置访问内部元素时，不宜选择链表，因为对链表的检索需要从表头开始，其时间复杂度为 O(n)。总之，两种存储结构各有其优缺点，应根据应用程序的实际需要来选择使用哪一种。

2.4　栈

栈是一种限制访问端口的线性表，即栈的所有操作都限定在线性表的一端进行。线性表的元素插入（称为栈的“压入”）和删除（称为栈的“弹出”,）都限制在表首进行。表首被称为“栈顶”，而栈的另一端称为“栈底”。栈的特点是，每次取出（并被删除）的总是刚刚压进的元素，即在时间上最后压入的元素。而最先压入的元素则被放在栈的

底部，要到最后才能取出。因而，栈又称为“后进先出表”、“下推表”，从元素的先后顺序来看，从栈里取出的元素正是压入元素的逆序。

下面给出栈的一个抽象数据类型定义。

【例 2.12】栈的类定义。

```
template<class T>
class Stack
{
    public:
        void Clear();                //清空栈
        bool Push(const T item);     //栈的压入操作
        bool Pop(T & item);          //读取栈顶元素的值并删除
        bool Top(T & item);          //读取栈顶元素的值但不删除
        bool IsEmpty();              //判断栈是否为空
        bool IsFull();               //判断栈是否已满
};
```

2.4.1 顺序栈

采用顺序存储结构的栈称为顺序栈，需要一块地址连续的存储单元来存储栈中的元素，因此需要事先知道或估计栈的大小。

顺序栈本质上是简化的顺序表。对元素数目为 n 的栈，首先需要确定数组的哪一端表示栈顶。如果把数组的第 0 个位置作为栈顶，按照栈的定义，所有的插入和删除操作都在第 0 个位置上进行，即意味着每次的 push 和 pop 操作都需要把当前栈的所有元素在数组中后移或者前移一个位置，时间复杂度为 O(n)。反之，如果把最后一个元素的位置 n-1 作为栈顶，那么只需要将新元素添加在表尾，出栈操作也只需要删除表尾的元素，每次操作的时间复杂度仅为 O(1)。图 2-18 所示为按照后一种方案实现的栈，其中 top 表示栈顶。

图 2-18 栈的顺序存储结构示意图

顺序存储实现时，用一个整形变量 top（通常为栈顶指针）来指示当前栈顶位置，同时也可以表示当前栈中元素的个数。下面给出一个顺序栈类及其部分成员函数的实现方法。

【例 2.13】栈的顺序实现。

```
template<class T>
```

```
class ArrayStack:public Stack<T>
{
    private:
        int maxSize;                //栈的最大值
        int top;                    //栈顶位置
        T *st;                      //存放栈元素的数组

    public:
        ArrayStack(int size)        //创建一个给定长度的顺序栈实例
        {
            maxSize = size;
            top = -1;
            st = new T[maxSize];
        }

        ArrayStack()                //创建一个顺序栈实例
        {
            top = -1;
        }

        ~ArrayStack()               //析构函数
        {
            delete [] st;
        }

        void Clear()                //清空栈的内容
        {
            top = -1;
        }

        bool Push(const T item)     //入栈操作
        {
            if(top = maxSize - 1)   //栈已满
            {
                cout << "栈满溢出" << endl;
                return false;
            }
            else                    //入栈，并修改栈顶指针
            {
                st[++top] = item;
                return true;
            }
        }

        bool Pop(T & item)          //出栈操作
        {
```

```
        if(top == -1)            //栈为空
        {
            cout << "栈为空，不能进行删除操作" << endll;
            return false;
        }
        else
        {
            item = st[top--];    //读取栈顶元素并修改栈顶指针
            return true;
        }
    }

    bool Top(T & item)           //读取栈顶元素，但不删除
    {
        if(top == -1)            //栈为空
        {
            cout << "栈为空，不能读取栈顶元素" << endl;
            return false;
        }
        else
        {
            item = st[top];
            return true;
        }
    }
};
```


图 2-19 所示为 maxSize 取值为 4 的顺序栈 s 中数据元素和栈顶指针的变化。其中，图 2-19（a）表示空栈状态，此时 s.top = −1；图 2-19（b）表示栈中具有一个元素的情况，此时栈顶 top 所指为栈顶元素所在位置 0；图 2-19（c）表示进行若干次进栈操作后栈满的情况，此时 top = MaxSize – 1；图 2-19（d）表示出栈一次后栈的状态。

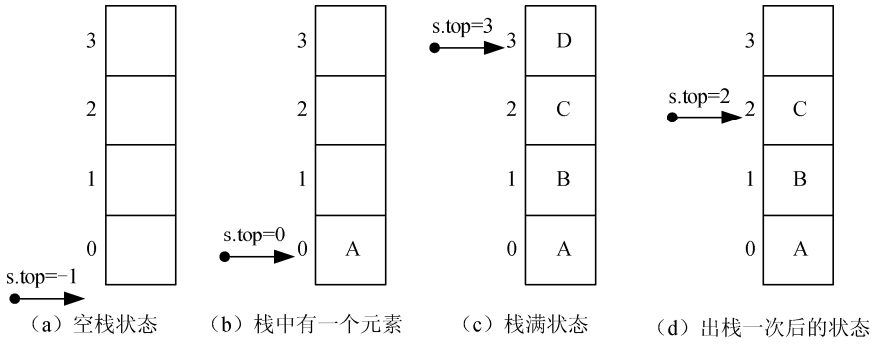


图 2-19　顺序栈的存储结构

栈中的元素是动态变化的，当栈中已有 maxSize 个元素时，进栈操作会产生溢出现象，这种溢出称之为“上溢”；相应地，在空栈上进行出栈操作也会发生溢出，这种溢出称为“下溢”。为了避免溢出，在对栈进行插入和删除操作之前要检查栈是否已满或者是否为空。

例 2.14 是改进的进栈操作。如果出现上溢出时仍然希望对顺序栈进行进栈操作，可以考虑适当地扩充当前顺序栈的容量。例 2.14 申请一个扩大一倍的数组，把顺序栈的原有内容移到新的数组中，按照正常的方式进行进栈操作。

【例 2.14】改进的进栈操作。

```
template<class T>
bool ArrayStack<T>::push(const T item)
{
    if(top == maxSize - 1)
    {
        T * newSt = new T [maxSize * 2];
        for(int i = 0; i <= top; i++)
        {
            newSt[i] = st[i];
        }
        delete [] st;     //释放原栈
        st = newSt;
        maxSize *= 2;
    }
    st[++top] = item;
    return true;
}
```


2.4.2 链式栈

链式栈本质上是简化的链表，为了方便存取，栈顶元素设置为链表首结点，变量 top 设置为指向栈顶的指针。图 2-20 为一个链式栈的示意图。

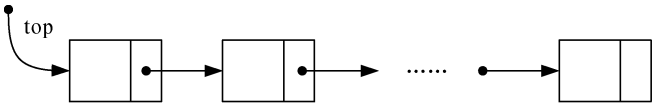


图 2-20　链式栈示意图

下面给出链式栈的一个简单实现。其中，数据成员 top 为指向链式栈首结点的指针，链表的结点类型为 2.3 节中定义过的 LinkNode 类。进栈操作 push 在链表首结点之前插入元素，出栈操作 pop 删除链表首结点元素并释放其空间，进栈操作和出栈操作的时间复杂度均为 O(1)。

【例 2.15】栈的链式实现。

```
template<class T>
class LinkStack:public Stack<T>
{
    private:
        LinkNode <T> *top;              //指向栈顶的指针
        int size;                       //存放元素的个数

    public:
        LinkStack(int s)                //构造函数
        {
            top = NULL;
            size = 0;
        }
        ~LinkStack()                    //析构函数
        {
            Clear();
        }
        void Clear()                    //清空栈内容
        {
            while(top != NULL)
            {
                LinkNode <T> *tmp = top;
                top = top->link;
                delete tmp;
            }
            size = 0;
        }
        bool Push(const T item)         //入栈操作的链式实现
        {
            LinkNode <T> *tmp = new LinkNode<T>(item, top);
            top = tmp;
            size++;
            return true;
        }
        bool Pop(T & item)              //出栈操作的链式实现
        {
            LinkNode<T> *tmp;
            if(size == 0)
            {
                cout << "栈为空，不能执行出栈操作" << endl;
                return false;
            }
            item = top->data;
            tmp = top->link;
            delete top;
            top = tmp;
            size--;
```

```
            return true;
        }
        bool Top(T & item)              //读取栈顶元素，但不删除
        {
            if(size == 0)
            {
                cout << "栈为空，不能读取栈顶元素" << endl;
                return false;
            }
            item = top->data;
            return true;
        }
};
```

2.4.3 栈与递归

递归是计算机科学的一个重要概念，是一种常用的算法设计技术。许多程序设计语言都支持递归，这些支持本质上是通过栈来实现的。在数学及程序设计方法学中递归的定义是：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。本节将以阶乘函数的计算为例，分析函数的递归调用在程序运行阶段的工作过程。

（1）阶乘函数的递归定义

阶乘 n!的递归定义如下：

$$n!=\begin{cases}1, & n\leqslant 0\\ n\times(n-1)!, & n>0\end{cases}$$

为了定义整数 n 的阶乘，必须先定义（n−1）的阶乘，而为了定义（n−1）的阶乘，还要定义（n−2）的阶乘，如此直到 n=0 为止，这时 0 的阶乘为 1。由此可以看出，递归定义由两部分组成。第一部分是递归基础，也称为递归出口，是保证递归结束的前提；第二部分是递归规则，确定了由简单情况求解复杂情况需要遵循的规则。在阶乘的定义中，递归基础为 $n\leqslant 0$，此时阶乘定义为 1；递归规则为 $n\times(n-1)!$，即 n 的阶乘由（n−1）的阶乘来求解。例 2.16 给出了阶乘函数。

【例 2.16】阶乘函数。

```
long factorial(long n)
{
    if(n <= 0)
        return 1;
    return n * factorial(n-1);
}
```

（2）递归函数的实现

大多数程序设计语言运行环境所提供的函数调用机制是由底层的编译栈支持的，编

译栈中的“运行时环境”指的是目标计算机上用来管理存储器，并保存执行过程所需信息的寄存器及存储器的结构。

在非递归调用的情况下，数据区的分配可以在程序运行前进行，直到整个程序运行结束再释放，这种分配称为静态分配，采用静态分配时，函数的调用和返回处理比较简单，不需要每次分配和释放被调用函数的数据区。在递归调用的情况下，被调用函数的局部变量不能静态地分配某些固定单元，而必须每调用一次就分配一次，以存放当前所使用的数据，当返回时随即释放，这种只有在执行调用时才能进行的存储分配称为“动态分配”，此时需要在内存中开辟一个足够大的称之为运行栈的动态区域。

用作动态数据分配的存储区可以按照多种方式组织。典型的组织如图 2-21 所示，将存储器分为栈区和堆区，栈区用于分配具有后进先出 LIFO 特征的数据（如函数的调用），而堆区则用于不符合 LIFO 的数据（如指针的分配）的动态分配。

运行栈中元素的类型（即被调函数需要的数据类型）涉及动态存储分配中的一个重要概念——活动记录。过程或函数的一次执行所需要的信息用一块连续的存储区来管理，这块存储区叫做活动记录。它由图 2-22 的各个域组成。

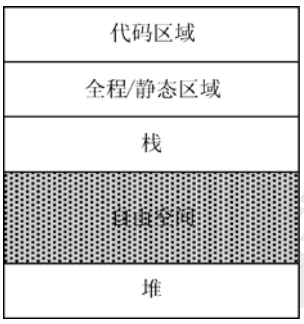


图 2-21　运行时存储器的组织形式

图 2-22　一般的活动记录

活动记录各个域的用途如下：

① 局部临时变量空间：用来存放目标程序临时变量的值，如计算表达式时所产生的中间结果。

② 局部变量空间：用于保存过程或函数的局部数据。

③ 机器状态：用来保存过程或函数调用前的机器状态信息，其中包括各种寄存器的当前值和返回地址等。

④ 控制链：用来指向调用方的活动记录。

⑤ 参数：用于存放调用方提供的实在参数。

⑥ 返回值：用于存放被调用方返回给调用方的值。

每次调用一个函数时，执行进栈操作，把被调函数所对应的活动记录分配在栈的顶部；而在每次从函数返回时，执行出栈操作，释放该函数的活动记录，恢复到上次调用所分配的数据区域中。因为运行栈中存放的是被调函数的活动记录，所以运行栈又称为

活动记录栈。同时，由于运行栈按照函数的调用序列来组织，因此也称为调用栈。

一个函数在运行栈中可以有若干不同的活动记录，每个活动记录代表了一个不同的调用。对于递归函数来说，递归深度就决定了其在运行栈中活动记录的数目。当函数进行递归调用时，函数体的同一个局部变量在不同的递归层次被分配给不同的存储空间，放在运行栈的不同位置。

概括来讲，函数调用可以分解成以下 3 个步骤：

1）调用函数发送调用信息，包括调用方要传送给被调用方的信息，如传给形式参数（简称形参）的实在参数（简称实参）的值、函数返回地址等。

2）分配被调用方需要的局部数据区，用来存放被调用方定义的局部变量、形参变量（存放实参的值）、返回地址等，并接受调用方传送来的调用信息。

3）调用方暂停，把计算控制转移到被调方，即自动转移到被调函数的程序入口。

当被调方结束运行，返回到调用方时，其返回处理一般也分解为 3 个步骤进行：

1）传送返回信息，包括被调方要传回给调用方的信息，诸如计算结果等。

2）释放分配给被调方的数据区。

3）按返回地址把控制转回调用方。

下面以例 2.17 阶乘函数为例，分析递归计算的过程中递归工作栈和活动记录是如何工作的。

【例 2.17】阶乘函数主程序。

```
#include<iostream>
using namespace std;
int main()
{
    cout << factorial(4) << endl;
    return 0;
}
```


主程序通过 factorial(4)这个语句向阶乘函数 factorial()的形参 n 提供了实参 4，建立阶乘函数 factorial()的一个活动记录，把过程调用所需的必要信息，包括返回地址、参数（4）、局部变量存入栈中，如图 2-23（a）所示。在计算 factorial(4)时，需要计算 factorial(3)的值，需要将 factorial(3)的活动记录压入栈中，如此进行下去，栈顶不断更新，直到最终调用 factorial(0)，此时 factorial(0)的活动记录成为新的栈顶。如图 2-23（b）所示。由于 factorial(0)满足递归的出口条件，可以直接得到计算结果，其活动记录从栈顶弹出，并将计算结果和控制权返回给其调用方 factorial(1)。factorial(1)根据 factorial(0)的返回结果 1 可以计算出 1! = 1，执行结束后，也从栈顶弹出其活动记录，继续将控制权转移给它的调用方 factorial(2)，依次类推，按进栈顺序的反序依次从栈中删除每个活动记录，把计算结果和控制权逐层返回，直至最后 factorial(4)把控制连同计算结果 24 返回给它的调用方 main()函数。这样，当在 main()中执行 cout 语句时，只在运行环境中保留了 main()和全局静态区域的活动记录。

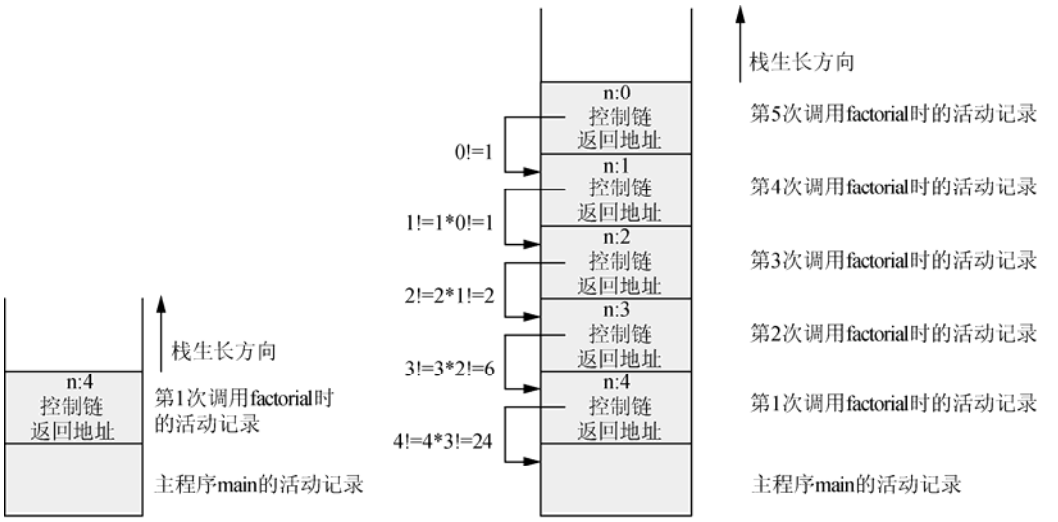


图 2-23　递归调用时栈的变化状态

2.5　队　　列

与栈类似，队列（queue）也是一种限制访问端口的线性表。队列的元素只能从表的一端插入，从表的另一端删除。按照习惯，通常会把只允许删除的一端称为队列的头，简称队头（front），把删除操作称为出队；而把表的另一端称为队列的尾，简称队尾（rear），这一端只允许进行插入操作，并将其操作称为入队。如同现实生活中的排队购物一样，在没有“插队”的情况下，新来的成员总是加入到队列的尾部，而每次取出的成员都是队列的前端，即先来先服务，因此队列通常也被称为先进先出（first in first out，FIFO）线性表。如图 2-24 所示为队列的一个示意图。

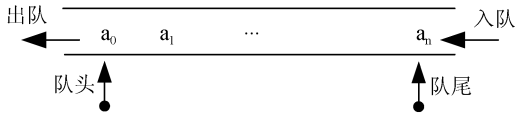


图 2-24　队列的示意图

下面给出队列的抽象数据类型，包括队列的入队、出队、查看队头元素等常用操作。

【例 2.18】队列的抽象数据类型定义。

```
template<class T>
class Queue
{
public:
    void Clear();                    //清空队列
```

```
    bool EnQueue(const T item);     //队列的尾部加入元素 item
    bool DeQueue(T & item);         //取出队列的第一个元素，并删除
    bool IsEmpty();                 //判断队列是否为空
    bool IsFull();                  //判断队列是否已满
    bool GetFront(T & item);        //读取队头元素，但不删除
};
```

队列的存储结构和实现方法主要包括顺序结构和链式结构两种。

2.5.1 顺序队列

用顺序存储结构来实现队列就形成了顺序队列。与顺序表一样，顺序队列需要分配一块连续的区域来存储队列的元素，需要事先定义队列的大小。与栈类似，顺序队列也存在溢出问题，队列满时入队会产生上溢现象，而当队列为空时出队会产生下溢现象。在队列出现上溢的现象时，如果需要继续操作，可以考虑为队列适当开辟新的存储空间。

为了有效的实现顺序队列，如果只沿用顺序表的实现方法，很难取得良好的效率。假设队列中有 n 个元素，顺序表的实现需要把所有元素都存储在数组的前 n 个位置上。如果选择把队列的尾部元素放在位置 0，则出队操作的时间复杂度是 O(1)，因为队列最前面的元素是数组最后面的元素，但是此时的入队操作时间复杂度为 O(n)，因为需要把队列中当前元素都向后移动一个位置。反之，如果把队列的尾放在 n−1 的位置上，就会出现相反的情况，出队时间复杂度为 O(n)，入队的时间复杂度为 O(1)。

如果可以保证队列元素在连续存储的同时允许队列的首尾位置在数组中移动，则可以提高队列的效率。如图 2-25 所示，将元素 3 和 6 分别出队列之后，队头元素变成了 8，将元素 12 入队列之后，队尾元素变成了新入队的 12。在经过多次的入队和出队操作之后，队头元素 3 变成了 8，队尾元素也变成了新入队的 12。随着出队操作的执行，队头 front 不断的后移，同时，随着入队元素的增加，队尾 rear 也是在不断的增加。

当队尾 rear 达到了数组的最末端，即 rear 等于 maxSize-1，即使数组的前端可能还有空闲的位置，再进行入队操作也会产生溢出，这种数组实际上尚有空闲位置而发生上溢的现象称为“假溢出”。解决假溢出的方法是采用循环的方式来组织存放队列元素的数组，在逻辑上将数组看成一个环，即把数组中的下标编号最低的位置，看成是编号最高位置的直接后继，这可以通过取模运算实现。即数组位置 x 的后继位置为(x+1) % maxSize，这样就形成了循环队列，也成为环形队列。图 2-26 所示为一个循环队列的示意图。

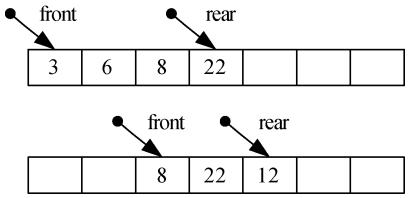


图 2-25　顺序队列的出队、入队示意图

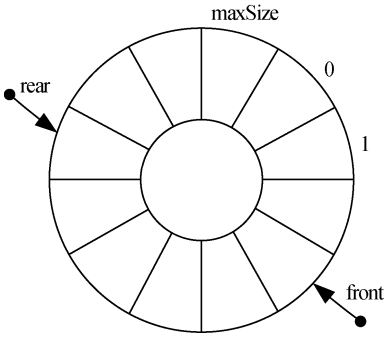


图 2-26　循环队列示意图

下面介绍如何表示一个空队列，以及如何表示一个队列已被元素填满。首先，忽略队头 front 的实际位置和其内容时，队列中的可能没有元素（空队列）、有一个元素、有两个元素等。如果数组有 n 个位置，则队列中最多有 n 个元素。因此，队列有 n+1 种不同的状态。如果把队头 front 的位置固定下来，则 rear 应该有 n+1 种不同的取值来区分这 n+1 种状态，但实际上 rear 只有 n 种可能的取值，除非有表示空队列的特殊情形。换言之，如果用位置 0~n−1 间的相对取值来表示 front 和 rear，则 n+1 种状态中必有两种不能区分。因此，需寻求其他途径来区分队列的空与满。

一种方法是记录队列中元素的个数，或者用至少一个布尔变量来指示队列是否为空。此方法需要每次执行入队或出队操作时设置这些变量。另一种方法，也是顺序队列通常采用的方法，是把存储 n 个元素的数组的大小设置为 n+1，即牺牲一个元素的空间来简化操作和提高效率。如图 2-27（a）表示队列为空的状态，此时 front＝rear；图 2-27（b）表示队列的一般状态，入队操作时，rear=(rear+1)%(n+1)；出队操作时，front=(front+1)%(n+1)；图 2-27（c）则表示队列为满的状态，此时(rear + 1) % (n + 1) = front。

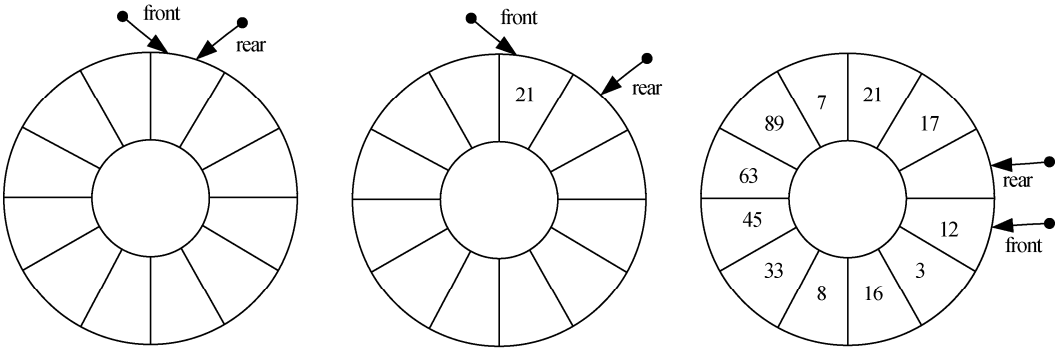
下面给出顺序队列的实现方法。

【例 2.19】队列的顺序实现。

```
template<class T>
class ArrayQueue:public Queue<T>
{
    private:
        int maxSize;                    //存放队列数组的大小
        int front;                      //表示队头所在位置的下标
        int rear;                       //表示队尾所在位置的下标
        int *queue;                     //存放类型为 T 的队列元素的数组

    public:
        ArrayQueue(int size)                //创建队列的实例
        {
            maxSize = size + 1;         //多出一个空间，区分队列空与满
```

```
    queue = new T[maxSize];
    front = rear = 0;
}

~ArrayQueue()                //析构函数
{
    delete [] queue;
}

void Clear()                 //清空队列
{
    front = rear;
}

bool EnQueue(const T item)  //item 入队，插入队尾
{
    if((rear + 1) % maxSize == front)
    {
        cout << "队列已满，溢出" << endl;
        return false;
    }
    queue[rear] = item;
    rear = (rear + 1) % maxSize;
    return true;
}

bool DeQueue(T & item)      //返回队头元素，并删除该元素
{
    if(front == rear)
    {
        cout << "队列为空" << endl;
        return false;
    }
    item = queue[front];
    front = (front + 1) % maxSize;
    return true;
}

bool GetFront(T & item)     //返回队头元素，但不删除
{
    if(front == rear)
    {
        cout << "队列为空" << endl;
        return false;
    }
    item = queue[item];
    return true;
}
```

```
};
```



（a）队列为空的状态　　（b）队列的一般状态　　（c）队列为满的状态

图 2-27　循环队列的几种状态示意图

2.5.2　链式队列

链式队列是队列的基于单链表的存储表示。如图 2-28 所示，在单链表的每个结点中有两个域，一个是存放数据元素的 data 域，另一个是存放单链表下一个结点地址的 link 域。在此单链表中设置了两个指针：队头指针 front 指向队首结点，队尾指针 rear 指向队尾结点。链表中的所有结点都必须通过这两个指针才能访问到，且队头端只能用来删除结点，队尾端用来插入新结点。例 2.19 给出了链式队列的实现，其中链表结点类型为 2.3 节中已经定义过的 LinkNode 类。

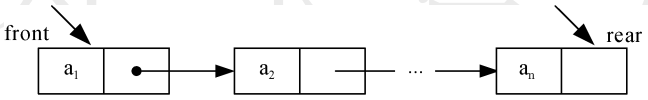


图 2-28　链式队列

【例 2.20】队列的链式实现。

```
template<class T>
class LinkQueue:public Queue<T>
{
private:
    int size;                    //队列中当前元素的个数
    LinkNode<T> * front;         //表示队列的头指针
    LinkNode<T> * rear;          //表示队列的尾指针

public:
    LinkQueue(int size)          //构造函数，创建队列的实例
    {
        size = 0;
        front = rear = NULL;
    }
```

```
~LinkQueue()                //析构函数
{
    Clear();
}

void Clear()                //清空队列
{
    while(front != NULL)
    {
        rear = front;
        front = front->link;
        delete rear;
    }
    rear = NULL;
    size = 0;
}

bool EnQueue(const T item) //item 入队，插入队尾
{
    if(rear == NULL)
    {
        front = rear = new LinkNode<T>(item, NULL);
    }
    else
    {
        rear->next = new LinkNode<T>(item, NULL);
        rear = rear->next;
    }
    size++;
    return true;
}

bool DeQueue(T & item)      //读取队头元素并删除
{
    LinkNode<T> * temp;
    if(size == 0)
    {
        cout << "队列为空" << endl;
        return false;
    }
    &item = front->data;
    temp = front;
    front = front->link;
    delete temp;
    if(front == NULL)
    {
```

```
            rear = NULL;
        }
        size--;
        return true;
    }

    bool GetFront(T & item)          //返回队头元素，但不删除
    {
        if(size == 0)
        {
            cout << "队列为空" << endl;
            return false;
        }
        item = front->data;
        return true;
    }
};
```

队列的应用十分广泛，在计算机硬设备之间可以作为数据通信缓冲器，在网络服务方面可以用作邮件缓冲器，在操作系统中用来解决 CPU 资源竞争的问题等。

2.6 字 符 串

2.6.1 基本概念

串（字符串）是计算机非数值处理的主要对象之一。在早期程序设计语言中，串仅作为输入和输出的常量出现，并不参与运算。随着计算机的发展，串在文字编辑、语法扫描、符号处理及定理证明等许多领域得到越来越广泛的应用。在高级程序设计语言中开始引入串变量的概念，如同整型、实型变量一样，串变量也可以参加各种运算，并建立了一组串运算的基本函数和过程。在汇编语言和高级语言编译程序中，源程序和目标程序都是串数据。在事务处理程序中，顾客的姓名和地址及货物的名称、产地和规格等一般也是用字符串处理的。在信息检索系统、文字编辑系统、问答系统、自然语言翻译系统、音乐分析程序以及多媒体应用系统中，也都是以串数据作为处理对象的。

现今使用的计算机硬件结构主要反映计算的需要，因此串数据的处理比整数和浮点数的处理要复杂得多。而且在不同类型的应用中，所处理的串具有不同的特点，要有效地实现串的处理，就必须依据实际情况，使用适合处理的存储结构。

（1）串的定义

串（或字符串）是由零个或多个字符组成的有限序列。含零个字符的串称为空串，用Φ表示。串中所含字符的个数称为该串的长度（或串长）。组成字符串的基本单位是字符。通常将一个串表示成“$a_1a_2\cdots a_n$”的形式。其中，最外边的双引号本身不是串的内容，它们是串的标志，以便将串与标识符（如变量名等）加以区别。每个$a_i(1\leqslant i\leqslant n)$

代表一个字符，不同的机器和编译语言对合法字符（即允许使用的字符）有不同的规定。但在一般情况下，英文字母、数字(0,1,…,9)和常用标点符号以及空格符等都是合法字符。

（2）子串、主串和位置

串中任意连续的字符组成的子序列称为该串的子串。包含子串的串称为主串。字符在序列中的序号称为该字符在串中的位置，子串在主串中的位置则以子串的第一个字符在主串中的位置来表示。

例如，串“eij”是串“Beijing”的子串，“Beijing”称为主串；字符‘n’在串“Beijing”中的位置为6；子串“eij”在串“Beijing”中的位置为2。

（3）空格串

由一个或多个空格组成的串称为空格串。由“ ”组成。

（4）串的比较

当且仅当两个字符串的值相等时，即只有当两个字符串的长度相等，且各个对应位置的字符都相等时，称这两个串相等。

当两个串不相等时，可以按照“字典顺序”分大小，令

$S = "s_1 s_2 \cdots s_m"$ $(m \geqslant 1)$

$T = "t_1 t_2 \cdots t_n"$ $(n \geqslant 1)$

① 比较两个串的第一个字符。如果$'s_1' < 't_1'$，则串S小于串T；反之，如果$'s_1' > 't_1'$，则串S大于串T。

② 确定两个串的最大相等前缀子串，$"s_1 s_1 \cdots s_k" = "t_1 t_1 \cdots t_k"$（其中$1 \leqslant k \leqslant m$，$1 \leqslant k \leqslant n$）。如果$k \neq m$且$k \neq n$，则由是$'s_{k+1}'$大还是$'t_{k+1}'$大来确定是串S大还是串T大；如果$k = m$且$k < n$，则此时串T大于串S；如果$k = n$且$k < m$，则此时串S大于串T。

例如，"axyz"<"bxyz"，"ab"<"abcde"，"abcde"<"abcdef"，$\Phi \neq$" "。

（5）串与线性表的区别

① 串的数据对象约束为字符集。

② 串的基本操作与线性表有很大差别：

线性表的基本操作中，大多以“单个元素”作为操作对象，如查找某个元素、在某个位置上插入一个元素或删除一个元素。

串的基本操作中，通常以“串的整体”作为操作对象。如在串中查找某个子串、在串的某个位置上插入一个子串以及删除一个子串。

例如，a = "BEI"，b = "JING"，c = "BEIJING"，d = "BEI JING"，长度分别为3、4、7、8；a和b都是c和d的子串；a在c和d中的位置都是1；b在c和d中的位置是4和5；a、b、c、d彼此不相等。

2.6.2 存储结构和实现

考虑到字符串变长的特点，合理选择字符串存储结构是很必要的，同时需要结合具

体的应用分析各种存储方案的利弊，再进行合适的选择。在实际应用中字符串长度变化非常显著，长如文件，短为单词。通过统计分布来看，字符串长度分布的方差较大。针对这种情况，用静态长度的向量作为存储结构是不恰当的。串的变长特点是无法回避的，字符串的拼接、查找、置换和模式匹配等操作本身都涉及变长串操作。这些操作开销时间大，必须精心设计算法，选择恰当的字符串存储结构。

1．字符串的顺序存储

对于字符串的长度分布变化不大的情况，字符串采用顺序存储比较合适。串的顺序表示就是把串中的字符顺序地存储在一组地址连续的存储单元中。为了便于编程和节省空间，一般的顺序存储方案使用类型为 char 的一维定长数组。

顺序存储的字符串适合访问串中连续的一组字符或者单个字符，但是进行插入或者删除操作就不是很方便，需要移动插入或者删除点后面的所有字符。此外，存储串的数组为静态定长的，程序运行中一旦产生更长的字符串，就会造成数组溢出，这给编写程序和调试程序带来不便。

C/C++语言的标准字符串是采用顺序存储方案的典型代表。其特点就是在程序中采用“ char str[Max] ”的形式定义字符串变量，其中 Max 是整形常数，表示字符数组的长度。标准字符串需要在其末尾带一个结束标记'\0'来表示串的结束，因此字符串的最大长度不能超过 Max−1。例如：

```
char str1[13] = "Hello World!";
char str2[7] = "2010";
char str3[4];
```

这 3 个数组 str1、str2 和 str3 在 C++中的存储示意图如图 2-29 所示。

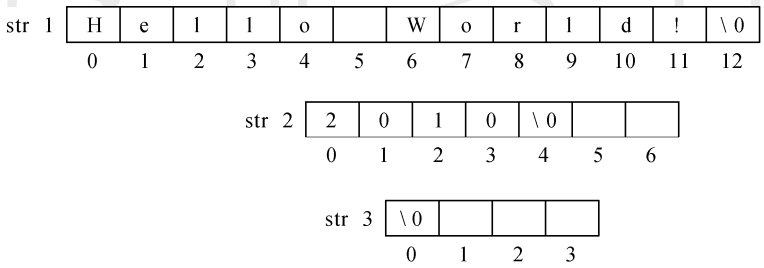


图 2-29　C++标准字符串的变量说明示意图

作为字符串的结束标志，'\0'是 ASCII 码中 8 位全 0 码，又称为 NULL 符。这个 NULL 符专门用作结束标记符。对于字符串常数，C/C++也是采用这种存储方式。在说明字符串变量时，字符串可以使用字符串常数作为初始值，也可以不赋初值。图 2-29 中的 str3 就没有给初值，意味着它存储空字符串。

C++中的字符数组是用字符指针定义的，指向字符数组的初始地址。赋值语句 str1 = str2 不能被理解为字符串 str2 的内容复制到字符串 str1。这是在字符串处理中非常容易犯的错误。标准库<string.h>提供了若干处理字符串的常用函数，表 2-2 列出了几个常

用的函数。

表 2-2　标准串函数

函数名	函数功能说明
int strlen(char *s)	求字符串 s 的当前长度，不计结束符。空串的长度为 0
char * strcpy(char * s1, const char *s2)	将字符串 s2 复制到 s1，并返回一个指针，指向 s1 的开始位置
char * strcat(char * s1, const char *s2)	将字符串 s2 拼接到 s1 尾部
int strcmp(const char *s1, const char *s2)	比较字符串 s2 和 s1，若 s1 和 s2 完全相同则返回 0；s1 大于 s2 则返回正数；s1 小于 s2 时返回负数
char * strchr(char *s, char c)	返回字符串中第一次出现字符 c 的位置。若 s 中不含 c，则返回空指针
char * strrchr(char *s, char c)	从字符串 s 的尾部查起，返回第一次出现字符 c 的位置。若 s 中不含 c，则返回空指针

字符串顺序存储方式简单易实现，C++的标准串及其标准库函数提供了若干处理字符串的方法，但并没有避免静态定长的局限。在实际应用中。大多数的字符串变量具有动态变化的长度。为此，下面介绍能够适应动态变化的字符串存储结构。

2．字符串类 class String 的存储结构

本节将讨论 String 类的存储结构，通过实例来分析存储空间是如何动态管理的。分别列举创建字符串 String::String(char *s)、赋值运算符 String String::operator = (String& s)、拼接运算符 String String::operator + (String& s)和抽取子串函数 String String::Substr(int index, int count)等 4 种操作。在 2.6.3 节中给出算法的具体实现代码。

（1）构造函数 String::String(char *s)

例如语句

```
String str1 = "Love";
```

此句隐含的调用构造函数 String::String(char *s)，s 所对应的实参为“Love”。功能是在动态存储区开辟一个长度为 5 的字符数组，并将初始值“Love”存入，末尾添加结束符。如图 2-30 所示。

（2）赋值运算符 String String::operator = (String& s)

例如语句

```
String str2 = "Love China";
String str1 = str2;
```

先创建字符串实例 str2，然后调用赋值语句 str1 = str2，相当于调用赋值运算符 str1 =，其实参是(str2)。为了把较长的字符串拷贝到 str1 中，必须将原有数组的空间释放，在动态存储区开辟新的数组，把 str1 的内容拷贝到新的数组中，如图 2-31 所示。

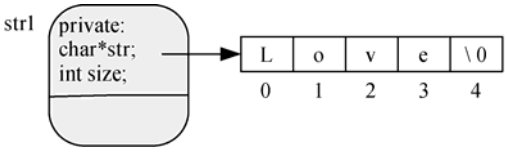


图 2-30　构造函数示意图

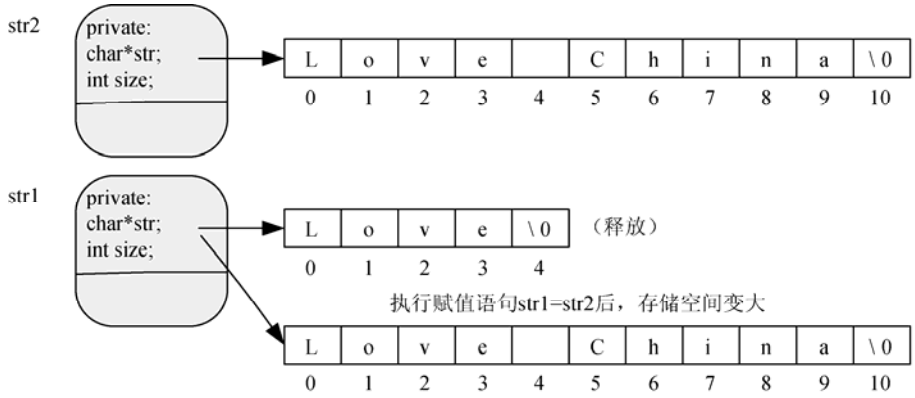


图 2-31　赋值运算示意图

（3）拼接运算符 String String::operator + (String& s)

例如下列语句

```
String str1 ="Love";
String str2 ="China", str3 = "China";
str3 = str1 + str2;
```

字符串实例 str1 有初值"Love"，str2 和 str3 有初值"China"。赋值语句 str3 = str1 + str2 拼接运算符的功能是把 str1 字符串尾部拼接 str2 字符串，结果为一个长的新字符串。由于 str3 没有足够的存储空间来存放结果，因此必须在动态存储区开辟新的空间来存储数组，同时释放原有空间，如图 2-32 所示。

（4）抽取子串函数 String String::Substr(int index, int count)

以下列语句为例

```
String str1 = "Love";
String str2 = "China";
str2 = str1.Substr(1,3);
```

抽取子串函数 str1.Substr(1, 3)是把字符串 str1 的一部分抽取出来，从下标 index 开始抽取长度为 count 的子串，形成新的子串。由于变量 str2 有足够的存储空间来存放赋值号右边的结果字符串，因此无需再动态开辟新的存储数组。如图 2-33 所示。

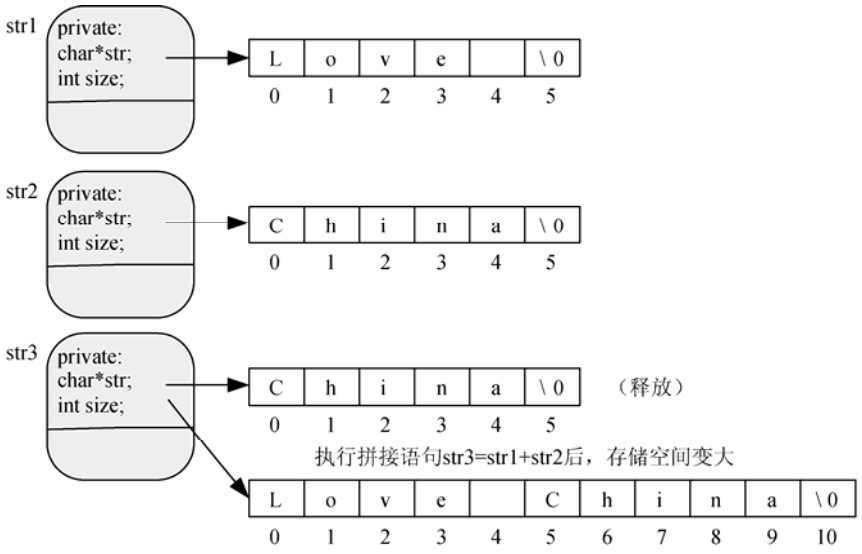


图 2-32　字符串拼接示意图

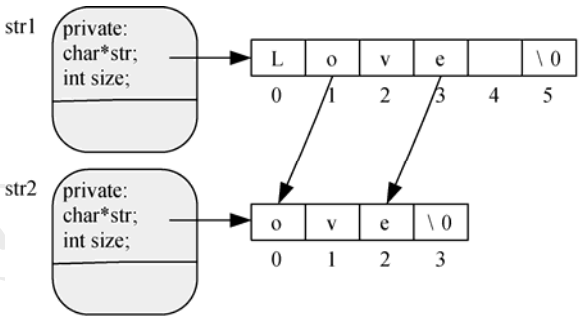


图 2-33　抽取子串实例

2.6.3　字符串运算的算法实现

本节所给出的程序功能说明请参照 2.6.2 节。

（1）C++标准串运算的实现

【例 2.21】求字符串的当前长度。

```
int strlen(char s[])
{
    int count = 0;
    while(s[count] != '\0')
        count++;
    return count;
}
```

【例 2.22】字符串的复制函数。

```
char * strcpy(char * s1, char * s2)
{
    int count = 0;
    while(s[i] != '\0')
    {
        s1[count] = s2[count];
        count++;
    }
    d[i] = '\0';
    return s1;
}
```

例 2.21 中，并没有比较 s1 和 s2 的长度，若 s2 比 s1 长，则会出现拷贝出界问题，可能会丢失字符串。

【例 2.23】字符串的比较函数。

```
int strcmp(char * s1, char * s2)
{
    int count = 0;
    while(si[count] != '\0' && s2[count] != '\0')
    {
        if(s1[count] > s2[count])
            return 1;
        else if(s1[count] < s2[count])
            return -1;
        i++;
    }
    if(s1[count] == '\0' && s2[count] != '\0')
        return -1;
    else if(s1[count] != '\0' && s2[count] == '\0')
        return 1;
    return 0;
}
```


【例 2.24】正向寻找字符函数。

```
char * strchr(char * s, char c)
{
    int count = 0;
    while(s[count] != '\0' && s[count] != c)
        count++;
    if(s[count] == '\0')
        return '\0';
    else
        return &s[count];
```

```
}
```

【例 2.25】逆向寻找字符函数。

```
char * strrchr(char * s, char c)
{
    int count = 0;
    while(s[count] != '\0')
        count++;
    while(count >= 0 && s[count] != c)
        count--;
    if(count < 0)
        return '\0';
    else
        return &s[count];
}
```

（2）String 串运算的实现

本节内的程序使用了<string.h>库函数来简化编程工作，注意，在类定义程序前要书写#include<string.h>包含语句。

【例 2.26】String 串构造函数。

```
String::String(char * s)
{
    size = strlen(s);           //求出串的长度
    str = new char[size + 1];   //动态开辟一块空间，需要存储结束符'\0'，所以长度为 size+1
    assert(str != '\0');        //当开辟动态区域不成功时，运行异常，退出
    strcpy(str, s);             //将初值 s 复制到 str 所指的存储空间
}
```

【例 2.27】赋值运算符的实现。

```
String String::operator = (String& s)
{
    if(size != s.size)      //长度不同，则释放本串存储空间并开辟新的空间
    {
        delete [] str;                  //释放原空间
        str = new char[s.size + 1];     //若开辟空间失败，则退出
        assert(str != '\0');
        size = s.size;
    }
    strcpy(str, s.str);
    return *this;
}
```

【例 2.28】拼接运算符的实现。

```
String String::operator + (String& s)
```

```
{
    String temp;                        //创建一个串 temp
    int len;
    len = size + s.size;                //拼接长串的长度
    delete [] temp.str;                 //释放 temp 创建时 new 申请的存储空间
    temp.str = new char[len + 1];       /为长串开辟存储空间
    assert(temp.str != NULL);           //若开辟存储空间不成功，则退出
    temp.size = len;
    strcpy(temp.str, str);              //先把本实例的私有项 str 存到 temp
    strcat(temp.str, s.str);            //进行字符串的拼接
    return temp;
}
```

【例 2.29】抽取字符串函数的实现。

```
String String::Substr(int index, int count)
{
    int left = size - index;    //自下标 index 向右计数到串尾,长度为 left
    String temp;
    char *p, *q;
    if(index >= size)           //若下标值 index 超过本串的实际长度，则返
                                    回空串
        return temp;
    if(count > left)            //若 count 超过自 index 以右的子符串长度，
                                    则 count 变小
        count = left;
    delete [] temp.str;         //释放原来的存储空间
    temp.str = new char[count + 1];
    assert(temp.str != NULL);   //若开辟存储空间不成功，则退出
    p = temp.str;               //p 指向暂无内容的空串
    q = &str[index];            //q 指向本实例串 str 数组下标 index 处
    for(int i = 0; i < count; i++)
        *p++ = *q++;            //将 q 所指的内容赋值给 p，同时后移
    *p = '\0';                  //加入结束标志'\0'
    temp.size = count;
    return temp;
}
```


2.6.4 字符串的模式匹配

字符串的模式匹配是一种常用的运算。所谓模式匹配，可以简单地理解为在目标（字符串）中寻找一个给定的模式（也是字符串），返回目标和模式匹配的第一个子串的首字符位置。通常目标串比较大，而模式串则比较短小。典型的例子包括文本编辑和 DNA 分析。在文本编辑时，人们经常会使用“替换”命令来对文本中的某个字符或者是字符串甚至是语句进行替换，此时便需要找到被替换的内容，再进行修改或替换。这个要查找的内容即为模式，在文中查找被替换内容的过程就是一个字符串模式匹配的过程。字符串模式匹配算法在分子生物学中越来越重要，人们使用该算法从 DNA 序列中提取信

息，在其中定位某种模式，并比较序列，获得共有的子序列。

模式匹配有精确匹配和近似匹配两类。

（1）精确匹配

如果在目标 T 中至少一处存在模式 P，则称匹配成功，否则即使目标与模式只有一个字符不同也不能称为匹配成功，即匹配失败。给定一个字符或符号组成的字符串目标对象 T 和一个字符串模式 P，模式匹配的目的是在目标 T 中搜索与模式 P 完全相同的子串，返回 T 和 P 匹配的第一个字符串的首字母位置。

（2）近似匹配

如果模式 P 与目标 T（或其子串）存在某种程度的相似，则认为匹配成功。常用的衡量字符串相似度的方法是根据一个串转换成另一个串所需的基本操作数目来确定。基本操作由字符串的插入、删除和替换来组成。

下面介绍关于精确匹配的两种经典算法。

1．朴素的模式匹配算法

模式匹配的一个简单方法就是把模式 P 的字符依次与目标 T 的相应字符进行比较。首先从首字母开始，依次将两个字符串相应位置上的字符进行比较，如图 2-34 的第（1）步所示。当某次比较失败时，则把模式 P 相对于 T 向右移动一个字符位置，重新开始下一次匹配，如图 2-34 的步骤（2）所示。如此不断重复，直到某一次匹配成功返回，如图 2-34 的步骤（3）所示；或者比较到目标串的结束也没有出现“配串”的情况，则匹配失败，如图 2-34 的步骤（4）所示。

图 2-34 的这种匹配方法称为朴素的模式匹配算法。其算法实现见例 2.29。函数 int NaiveStrMatching(String T, String P)从目标 T 的首位置开始进行匹配，用模式 P 匹配 T，寻找首个模式 P 子串并返回其下标位置。若整个过程匹配失败，则返回负值。

【例 2.30】朴素的字符串模式匹配算法。

```
#include<string.h>
#include<assert.h>
int NaiveStrMatching(const String & T, const String & P)
{
    int p = 0;                          //模式的下标变量
    int t = 0;                          //目标的下标变量
    int plen = P.length();      //模式的长度
    int tlen = T.length();      //目标的长度
    if(tlen < plen)             //如果目标比模式短，匹配无法成功
        return -1;
    while(i < plen && j < tlen)//反复比较对应字符进行匹配
    {
        if(T[t] == P[p])
        {
            p++;
```

```
                t++;
            }
            else
            {
                t = t - p + 1;
                p = 0;
            }
        }
        if(p >= plen)
            return (t - plen + 1);
        else
            return -1;
    }
```

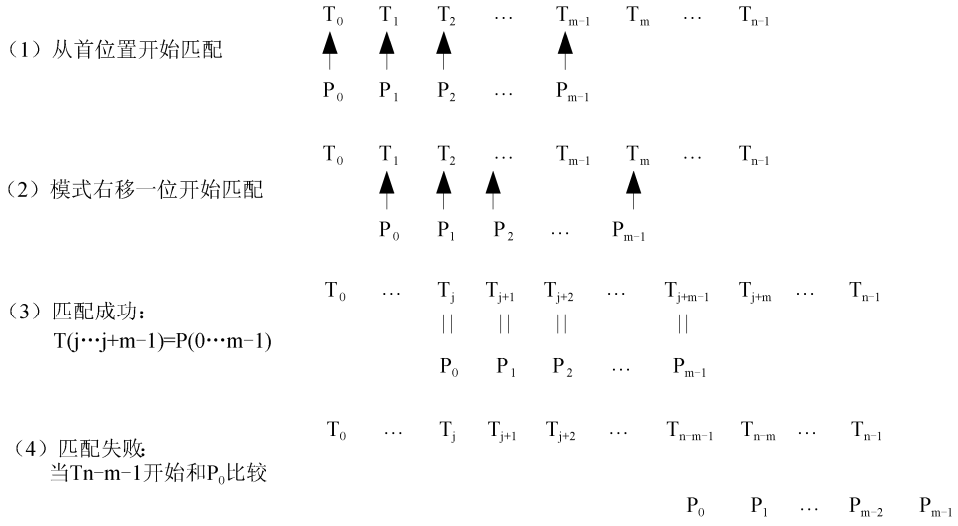


图 2-34　朴素的模式匹配方法示意图

图 2-35 是一个采用朴素的模式匹配算法的例子，目标字符串为"acabaabaabcacab"，模式为"abaabcac"，匹配过程中若模式的字符与当前目标字符不等则用下划线及黑体标出。在第 1 次匹配中，目标字符串中字符'c'与模式中的字符'b'不匹配，此时模式右移到下一个字符开始第 2 次匹配，发现对应的字符'c'与模式中的字符'a'不匹配，模式继续右移，直到第 6 次匹配时匹配成功，此时返回匹配位置 5，作为模式匹配成功的结果。若匹配不成功，则返回一个负数。

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	a	c	a	b	a	a	b	a	a	b	c	a	c	a	b
1	a	b	a	a	b	c	a	c							
2		a	b	a	a	b	c	a	c						
3			a	b	a	a	b	c	a	c					
4				a	b	a	a	b	c	a	c				
5					a	b	a	a	b	c	a	c			
6						a	b	a	a	b	c	a	c		

图 2-35　朴素的模式匹配示例

朴素的模式匹配算法最差的情况需要$O(|T||P|)$的代价，其中|T|和|P|分别是目标串和模式串的长度。例如目标串 T 形如a^n（即由 n 个字符 a 组成的字符串），而模式 P 形如$a^{m-1}b$，每次都是模式的最后一个字符处不匹配，即每次都需要进行$|P|$次比较，再把模式右移一位，再次从模式的第一个字符开始比较，最多需要（$|T|-|P|+1$）次，因此总共需要比较$|P|(|T|-|P|+1)$次。

Knuth、Morris、Pratt 等人发现，其实每次右移的位数存在且与目标串无关，仅依赖于模式本身，因此他们对朴素的模式匹配算法进行了改进。改进后的算法简称为 KMP 算法，其基本思想为：预先处理模式本身，分析其字符分布情况，并为模式中的每一个字符计算失配时应该右移的位数。

2．字符串的特征向量

如图 2-36，在模式 P 与目标 T 的匹配过程中，当某次比较出现$P_i \neq T_j$时，意味着在此前的匹配历史中有下述匹配的子串满足：$P(0\cdots i-1)=T(j-i\cdots j-1)$。即在$P_i \neq T_j$之前，P 从首字母开始的子串$P(0\cdots i-1)$已与目标 T 中的$T(j-i\cdots j-1)$相等，如果$P(0\cdots i-2)=P(1\cdots i-1)$成立，即表示模式右移一位后，$P(0\cdots i-2)$与$T(j-i+1\cdots j-1)$必相等。所以，此时直接比较$P_{i-1}$和$T_j$即可。由此可知，利用之前的比较结果减少了在目标串上的回溯。若$P(0\cdots i-2)=P(1\cdots i-1)$不成立，则可以继续查看$P(0\cdots i-3)$与$P(2\cdots i-1)$是否相等，若相等，则把模式右移两位，因为此时$P(0\cdots i-3)$与$T(j-i+2\cdots j-1)$必相同。依次类推，必然可以找到某一个 k 值，使得$P(0\cdots i-k-1)=P(k\cdots i-1)$成立，此时把模式右移 *k* 位开始下一次匹配，既不会丢失配串，又不会产生目标回溯的情况。

通常，把模式 P 的前 k 个字符组成的子串$P(0\cdots k-1)$称为 P 的前缀子串，把 P 在 i 之前的前 k 个字符组成的子串$P(i-k\cdots i-1)$称为位置 i 的后缀子串。对于模式位置 i 上的字符P_i而言，一旦与目标T_j失配时，模式的下标从位置 i 移动到位置 k，P_k直接与T_j比较，而跳过了$P(0\cdots k-1)$。如果把目标T_j

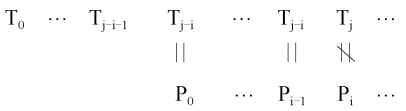


图 2-36　$P_i \neq T_j$时的状态

看做是不动的，则模式向右滑动了 $i-k$ 位。把这个 k 值称为 P_i 的特征数 n_i，即 $P(0\cdots i-1)$ 中最大相同前缀子串和后缀子串。所有的特征数组成了模式的一个特征向量，表示模式 P 的字符分布特征。

特征数 $n_i(0 \leqslant n_i \leqslant i)$ 是递归定义的，其定义如下：

1）$n_0=0$，对于 $i>0$ 的 n_i，假定已知前一位置的特征数 $n_{i-1}=k$；

2）如果 $i>0$ 且 $q_i=q_k$，则 $n_i=k+1$；

3）当 $q_i \neq q_k$ 且 $k \neq 0$ 时，则令 $k=n_{k-1}$，并让（3）循环直到条件不满足；

4）当 $q_i \neq q_k$ 且 k=0 时，则 $n_i=0$。

下面给出计算特征向量 N 的程序 int * Next(String P)。

【例 2.31】计算特征向量的算法。

```
int * Next(String P)
{
    int m = P.strlen();              //模式 P 的长度
    assert(m > 0);                   //若 m = 0，则退出
    int * N = new int[m];            //在动态存储区开辟新的数组
    assert(N != 0);
    N[0] = 0;
    for(int i = 1; i < m; i++)       //对 P 的每一个位置进行分析
    {
        int k = N[i - 1];            //第 i-1 位置的最长前缀串长度
        while(k > 0 && P[i] != P[k])
            k = N[k - 1];
        if(P[i] == P[k])
            N[i] = k + 1;
        else
            N[i] = 0;
    }
    return N;
}
```


3．KMP 模式匹配算法

求得模式的特征向量之后，基于特征分析的快速模式匹配算法（KMP 模式匹配算法）与朴素匹配算法类似，只是在每次匹配过程中发生某次失配时，不再单纯地把模式后移一位，而是根据当前字符的特征数来决定模式右移的位数。具体实现方法见例 2.31。

【例 2.32】KMP 模式匹配算法。

```
#include "String.h"
#include<assert.h>
int KMPStrMatching(String T, String P, int * N, int startIndex)
```

```
{
    int lastIndex = T.strlen() - P.strlen();
    if((lastIndex - startIndex) < 0) //若 startIndex 过大，则无法匹配成功
        return (-1);
    int i;                          //指向 T 内部字符的游标
    int j = 0;                      //指向 P 内部字符的游标
    for( i = startIndex, i < T.strlen(); i++)
    {
        while(P[j] != T[i] && j > 0)
            j = N[j - 1];
        if(P[j] == T[i])            //当 P 的第 j 位和 T 的第 i 位相同时，则继续
            j++;
        if(j == P.strlen())
            return (i -j + 1);  //匹配成功，返回该 T 子串的开始位置
    }
    return (-1);
}
```

由于计算字符串特征向量数组的算法本身采用的也是 KMP 模式匹配，因此时间代价也与模式长度成正比，为O(|P|)。所以整个 KMP 匹配的时间代价为O(|P|+|T|)。在模式长度远小于目标长度时，基本上为O(|T|)。此外，对于同一模式，由于特征向量是预先计算的，其计算结果可以在 KMP 算法中被多次使用。

2.7 线性表的应用

2.7.1 栈：简易计算器

1．需求描述

栈是一种常见的数据结构，其应用非常广泛，比如文件编辑器中保存 undo 序列，记录网页历史访问记录，编译器中函数的递归调用以及二叉树的深度遍历等。实际上，在能够计算算术表达式（一般为中缀表达式）的计算器中，栈的功能也得到了充分的体现。

一个简易计算器至少要完成下述功能：①可以识别加减乘除以及括号的中缀表达式并计算；②如果表达式有误，应给出相应的提示，比如“除数不能为 0”等。计算器的一个正常的计算流程如图 2-37 所示。

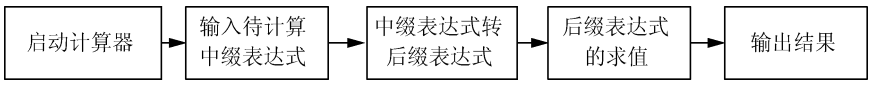


图 2-37　表达式计算流程图

2．问题分析

一个表达式通常由变量、常量、运算符以及函数调用按照一定规则组合而成，为方便讨论，在不失一般性的前提下，下面以只包含加减乘除括号的四则运算表达式为例，展开说明栈在表达式计算中的作用。

（1）表达式的定义

基本符号集 由数字0~9以及加、减、乘、除、括号组成的16个基本符号组成，各个符号之间用“,”间隔开来：

{ 0, 1,…, 9, +, −, *, /, (,) }

语法成分集 共包含5个语法成分：

{ <表达式>, <项>, <因子>, <常数>, <数字> }

语法公式集 语法公式用于定义语法成分，又称产生式规则。为方便起见，同时在不影响应用的前提下，下面给出要讨论的经过简化的表达式：

<表达式> :: = <项> + <项> | <项> − <项> | <项>

<项> :: = <因子> * <因子> | <因子> / <因子> |<因子>

<因子> :: = <常数> | (<表达式>)

<常数> :: = <数字> | <数字> <数字>

<数字> :: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

在语法公式中，用:: =表示规则定义符，其左侧部分表示一个被定义的语法成分的名称，右侧部分则用来定义左侧部分的语法规则。语法公式包含若干基本符号或若干语法成分作为其组成部分。其中，连接符“|”用于分隔其他语法成分，不作为语法成分，其含义为“或者”。例如“<数字> :: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9”表示数字的符号是0~9中的任意一个。

公式右部常由若干语法成分顺序拼接而成。例如，语法公式：

<项> :: = <因子> * <因子> | <因子> / <因子> | <因子>

其含义为项是由两个因子中间插上一个运算符*或/所组成的，或者也可以为一个单独的因子。语法公式还可以进行递归定义，如：

<常数> :: = <数字> | <数字> <数字>

其含义为常数由若干数字顺序排列而成。此外，公式：

<因子> :: = <常数> | (<表达式>)

其中右部被圆括号括起来的<表达式>成分所涉及的几个语法成分之间相互引用，形成递归定义，因为前面的语法公式集中<表达式>的定义右部出现<项>，而<项>的右部又出现了<因子>。这种递归定义的语法公式类似数学上的递推公式，如果把他们的左部不短带入到右部相同语法成分的相应位置，就可以组成很复杂的结构。

在上述公式中，运算符都是放在两个运算对象中间，故称其为中缀表达式。中缀表达法的算术表达式在进行具体求值计算时，一般需按照如下计算次序：

1）先执行括号内的计算，后执行括号外的计算。在具有多层括号时，按层次反复地去括号，左右括号必须配对。

2）在无括号或同层括号时，先乘（*）、除（/），后加（+）、减（-）。

3）在同一个层次，若有多个乘除（*、/）或加减（+、/）的运算，就按照自左向右顺序执行。

括号可以改变运算符的优先级。例如，表达式“23 + (34 * 45) / (5 + 6 + 7)”的计算过程是：

1）34 * 45 = 1530；

2）5 + 6 + 7 = 18；

3）1530 / 18 = 85；

4）23 + 85 = 108；

为了符合上述语法公式，该公式中 34 * 45 采用了括号的形式，与通用的“23 + 34 * 45/(5+6+7)”形式的语义相同。

后缀表达式　又称逆波兰表达式，不含括号，运算符放在两个参与运算的语法成分的后面。按照后缀表达式求值时，所有的求值计算皆按运算符出现的顺序，严格从左向右进行。下面给出与上述中缀表达式对应的后缀表达式的语法公式：

<表达式> :: = <项> <项> + | <项> <项> - | <项>

<项> :: = <因子> <因子> * | <因子> <因子> / | <因子>

<因子> :: = <常数>

<常数> :: = <数字> | <数字> <数字>

<数字> :: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

可以看出，后两个语法公式与中缀表达式的形式相同。参照前面中缀表达式例子“23 + (34 * 45) / (5 + 6 + 7)”，给出其后缀表达式：

23　34　45　*　5　6　+　7　+　/　+

通过观察可以看出，后缀表达式中所有操作数的出现次序与其在等价的中缀表达式中的出现次序完全相同，不同之处仅在于运算符按照实际计算的顺序出现在对应的操作数后面，且去除了括号。

（2）中缀表达式转换位后缀表达式

将中缀表达式转换为等价的后缀表达式是栈的一个典型的应用场景。在高级程序设计语言的编译软件中，就是使用这种算法对算术运算公式进行变换的，最终把它们转变为计算机可以直接执行的机器指令序列。

中缀表达式的运算次序受到运算符优先级和括号的影响，因此，在中缀表达式转换为与其等价的后缀表达式的过程中，关键在于如何利用栈来恰当的去掉括号，然后在必要的时候按照先乘除后加减的优先规则调换运算符的先后次序。在去括号的过程中用栈来存储有关的元素。

其实现的基本思路为：从左至右扫描中缀表达式，用栈存放表达式中的操作符，开括号以及在这个括号后面连带的暂时不能确定计算次序的内容。

设算法的中缀表达式用字符串 InfixExp 表示；输出结果，即后缀表达式用字符串 PostfixExp 表示。算法大致流程如下（不考虑一些异常情况，自左向右扫描 InfixExp，依次读入数字或者符号）：

1）当读入的是操作数时，直接输出到后缀表达式 PostfixExp 序列。

2）当读入的是开括号时，直接将其压栈。

3）当读入的是闭括号时，先判断栈是否为空。

若为空，则表示括号不匹配，应作为错误异常处理，清栈退出。

若非空，则把栈中元素依次弹出，直到遇到第一个开括号为止，将弹出的元素输出到后缀表达式 PostfixExp 序列。由于后缀表达式不需要括号，因此不应将弹出的开括号输出到后缀表达式 PostfixExp 序列。若没有遇到开括号，则说明括号匹配不成功，清栈退出。

4）当读入的是运算符 op（即为四则运算“+ - * /”之一）时：

① 当栈非空 && 栈顶不是开括号 && 运算符 op 的优先级不高于栈顶运算符的优先级时，反复进行将栈顶元素弹出，并输出到后缀表达式 PostfixExp 序列。

② 把输入的运算符 op 压栈。

5）最后，当中缀表达式 InfixExp 的符号序列全部读入时，若栈内仍有元素，则把他们全部依次弹出并输出到后缀表达式 PostfixExp 序列。若弹出的元素遇到开括号，则说明括号不匹配，进行错误异常处理，清栈退出。

这样，通过对中缀表达式 InfixExp 进行一次扫描就可以将其转换成等价的后缀表达式。上面仅给出了算法的核心思想和流程，并没有对涉及到的字符符号的读入、字符串的拼接、语法检查以及语法错误处理等细节进行展开。

（3）后缀表达式的求值

由于后缀表达式的计算按照运算符出现的次序进行，其计算相对简单。假设待计算的表达式以“=”结束。下面讨论后缀表达式的求值算法，用一个栈来存放操作数，自左向右顺序扫描后缀表达式，对该序列中的符号依次进行分析：

1）若遇到一个操作数，则直接将其压栈。

2）若遇到一个运算符，则从栈中弹出两个操作数，按照运算符对着两个操作数进行相应的计算，然后将计算结果压栈。

如此继续，直到遇到符号“=”，此时栈中应包含且只包含一个数，即为表达式的值。例如，对后缀表达式

23 34 45 * 5 6 + 7 + / +

的实际求值过程。从左向右扫描，首先遇到操作数 23,34,45，将他们压入栈中，遇到第一个运算符“*”时，弹栈两次得到两个操作数 45 和 34，进行“*”运算，即计算“34*45”，然后将运算结果 1530 压入栈中。继续向右扫描，把遇到的操作数 5,6 分别压

栈，当遇到运算符“+”时，同样两次弹栈得到操作数 6,5，然后计算“5+6”并将其结果 11 压入栈中。然后，把遇到的操作数 7 压栈，当再遇到加号时，从栈中弹出 7,11，并计算“11+7”并将结果 18 压入栈中。接着，遇到运算符“/”时，同样两次弹栈得到操作数 18,1530，这时候要引起注意，先弹出的为第 2 操作数，后弹出的为第 1 操作数，因此计算“1530/18”并将其结果 85 压栈。最后，遇到运算符“+”，弹出栈中剩下的两个操作数 23,85，计算“23+85”，其值 108 即为表达式求值的最终结果。

（4）应用实现

简易计算器的实现主要涉及到栈的应用以及字符串处理的一些知识。通常来讲，人们最习惯、最便捷的输入待计算的表达式为中缀表达式，但是事实上使用栈来实现计算表达式功能的时候，后缀表达式更容易计算。因此，可以结合栈和字符串的相关知识，定义一个计算器类，利用此类对计算器的各个步骤进行统一的管理。整个计算过程可以分成两大步：第一，先将输入的中缀表达式转换为后缀表达式；第二，计算后缀表达式的值。

3．概要设计

简易计算器主要包含一个计算器类 Calculator，主要用来管理计算表达式的各个步骤，首先接收一个后缀表达式字符串，然后将中缀表达式转换为后缀表达式，接着对后缀表达式进行计算，最后将得到的值返回。

下面给出设计类图，如图 2-38。

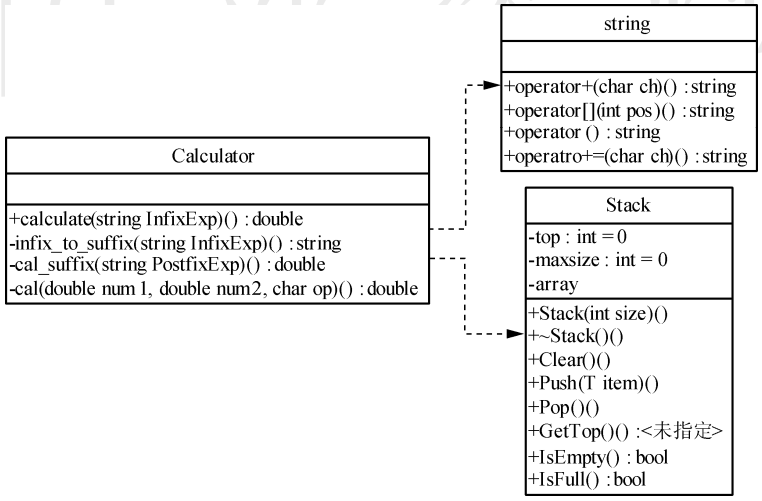


图 2-38　简易计算器类图

4．详细设计

根据上文的分析，中缀表达式转后缀表达式的流程如图 2-39 所示。

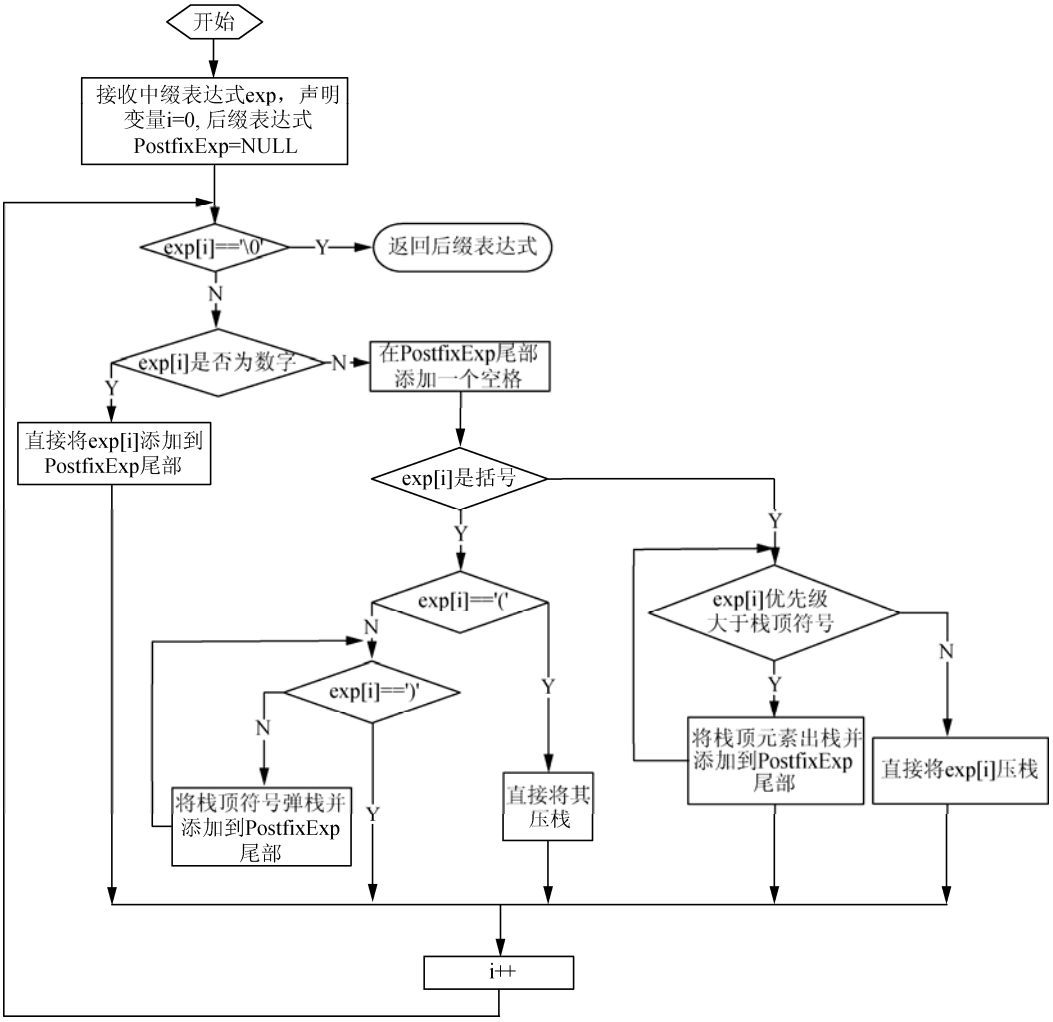


图 2-39　中缀表达式转后缀表达式流程图

在此过程中，首先由函数接收一个用户的输入，即一个中缀表达式字符串，然后对输入的中缀表达式从左向右依次进行扫描，在扫描的过程中，根据上文提到的中缀转后缀表达式的过程的五个步骤对其进行转换。

图 2-40 是后缀表达式的计算过程流程图。

在此过程中，也是首先由函数接收一个字符串类型的后缀表达式，然后从左向右扫描字符串，依次对字符串进行表达式的计算，计算过程参考上文的“后缀表达式”求值过程。

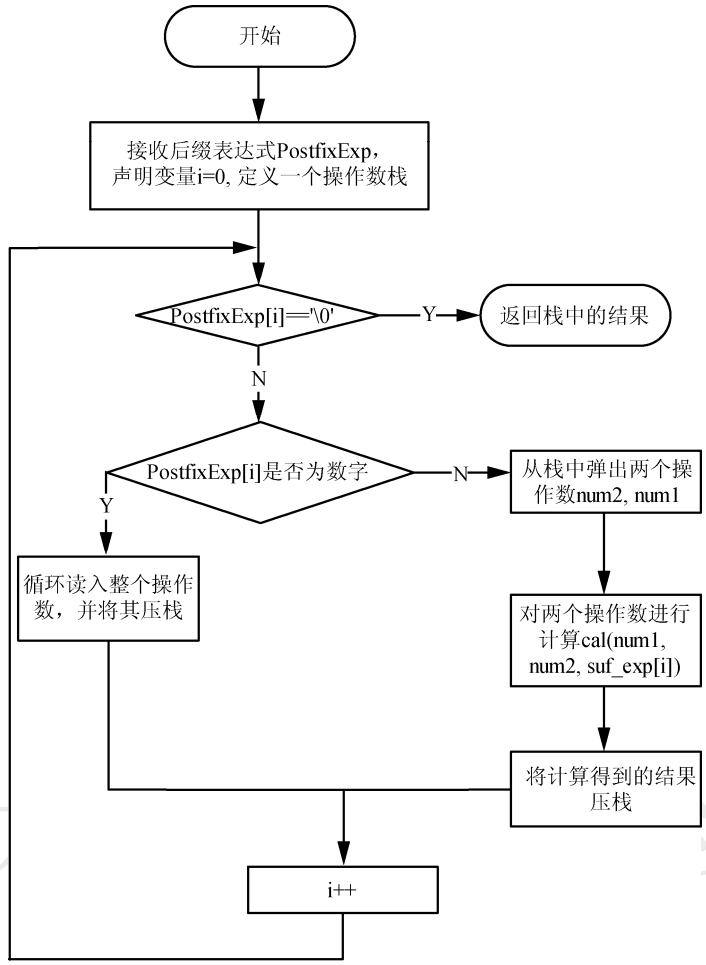


图 2-40　后缀表达式计算流程图

2.7.2　队列：银行叫号系统的实现

1．需求描述

银行取号机（叫号机）到处可见，也就是在生活中经常看到的排队机，它给用户带来了很大的方便。

使用取号排队系统，一方面可消除用户长时间“站队”的辛苦、对“站错队”、“插队”的抱怨，避免发生排错队和混乱嘈杂的现象，减少许多不必要的纠纷，全面改善服务质量和企业形象，另一方面更可以依据统计数据调整业务分配、挖掘潜力、合理安排窗口服务，减少群众的等候时间，提高办事效率。同时，排队系统支持多种形式的排队，可依照业务的种类或用户种类进行排队，支持对特殊对象（会员）的优先服务。

编写一个程序模拟银行取号系统，该程序只是模拟了取号排队的功能，至于完全的

模拟需要硬件与软件的结合。本模拟系统的最终目标是：实现叫号机的基本功能，即用户到达后可以叫号，工作人员登录进入系统后可以对用户进行办理业务，有如下规则：

1）银行客户有多个优先级普通业务用户，VIP 业务用户，对公业务用户。

2）银行有多个窗口，分为普通窗口，VIP 窗口和对公窗口，窗口数任意，三种业务窗口的比例为 3∶1∶1。

3）各类型客户在其对应窗口按顺序依次办理业务。

4）VIP 窗口和对公业务窗口，在空闲的时候可以处理普通业务窗口，而一旦有对应的客户等待办理业务的时候，则优先处理对应客户的业务。

5）随机生成各种类型的客户到达银行取票的过程，各类型业务用户数量可以预先设定。

6）自行设定业务办理时间，以及业务操作，可设置为三种用户设置相同的服务时间。

7）办理业务的过程通过一个时间变量来模拟。无需图形界面，只需要控制台以命令行形式模拟，窗口叫号过程，输出用户的类型还有该类型的编号，如果窗口由用户在接受服务，则输出窗口类型和编号，并且输出用户类型及编号。

2．问题分析

本题目是队列的应用，主要需要模拟的是，用户入队列，办理业务用户出队列，优先级比较选择队列。

模拟用户入队列，也就是模拟不同类型的用户取号的过程，为简化代码做出如下分析。用户类型分为三种类型，一种普通用户数量较多，两种特殊用户类型较少，故可以将这三种用户设置成为三个队列。根据用户比例，可以简化程序构造，预先设置这三类用户分别由 n1 位，n2 位，n3 位，预先存储于数组之中。取号时间即入队时间亦可预先设定，但是需要出现条件中预设的特殊情况，例如特殊队列空闲，普通用户队列有人的情况。

另外两种情况的模拟，也就是模拟个窗口在处理用户业务之后的叫号过程。同样做一下简化，银行窗口的处理用户业务的时间设定为固定的时间 t 秒，窗口处理业务完成之后，立刻从对应的队列中取得下个客户，并且在窗口中显示窗口号以及其叫的用户号。

这里假设普通用户窗口一直有人办理业务，不会出现空闲情况。VIP 和对公窗口客户较少，如果出现空闲情况，可以从普通用户队列用户中取得下一个用户，如果处理业务的过程中，一旦有对应的客户等待办理业务的时候，则处理完之后优先处理对应客户的业务。最后根据三种窗口的比例，不妨假设三种类型的窗口个数分别为普通用户窗口 3 个，VIP 和对公窗口各一个。

综上所述，需要三个用户队列来模拟存储不同类型的用户，需要三种对象来模拟存储不同类型的银行窗口，还需要编写一个入队模拟器来模拟用户的到达情况以及窗口叫号的过程，最重要的还需要预设好用户到达的场景以达到简化程序，覆盖所有题目要求

的条件。

3．概要设计

首先根据上一部分的分析，先将程序的逻辑结构分析如下：整个模拟系统可以分为三大类：用户类，银行窗口类和模拟程序类。其中用户类为基类下面可以分为普通用户，VIP 用户，还有对公用户。

银行窗口类按照三种用户类型可以抽取公有成员作为基类，然后针对不同的用户类型分为普通窗口，VIP 窗口，还有对公窗口。模拟程序类，负责三种情况的模拟，首先模拟程序主结构也就是时间的变化，模拟用户来到银行入队列的过程，和银行窗口服务和叫号的过程，整个程序的类图如图 2-41 所示。

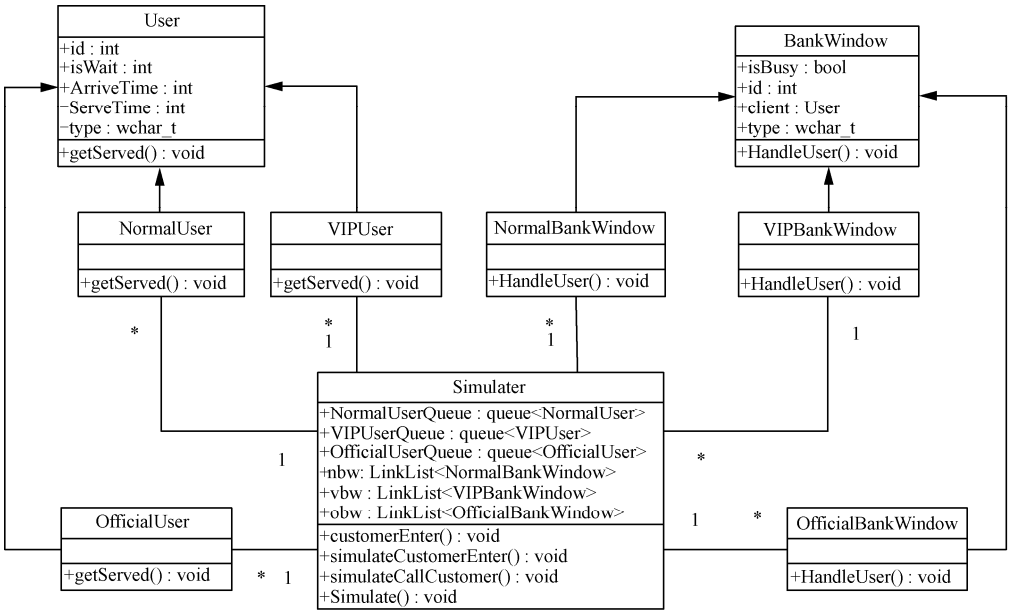


图 2-41　银行叫号系统类图

程序主体通过调用 Simulater 模拟器类的 Simulate 方法，来模拟整个叫号过程。该类是整个程序的核心类，数据成员包括三个用户队列来标识三类用户，还有三个银行窗口链表用来模拟存储银行窗口，方法包括模拟用户入队，银行处理用户队列，以及呼叫用户，程序结构如图 2-42 所示。

然后设置用户到达的场景。业务的处理时间是 t 秒，那么可以假设用户入队的时间间隔是 t1 秒，显然需要 t1<t，才会出现等待用户，故不妨设 t=4s，t1=2s。然后假设初始用户就可以将窗口占满，也就是初始情况下有 3 位普通用户，VIP 和对公用户各一位,然后模拟一个 12s 的叫号过程，会在 0s，4s 和 8s 的时候会有叫号的过程，12s 的时候没有用户，没有叫号，程序结束，如表 2-3 所示。

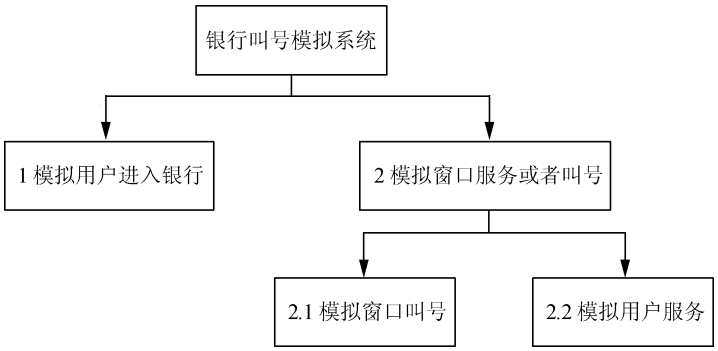


图 2-42　程序结构图

表 2-3　12s 各类型用户总数以及等待用户总数表

时间	进入银行用户（人）			等待用户（人）		
	普通用户	VIP 用户	对公用户	普通用户	VIP 用户	对公用户
0s	3	1	1	0	0	0
2s	5	0	0	5	0	0
4s	0	0	0	0	0	0
6s	5	1	0	5	1	0
8s	0	0	0	1	0	0
10s	0	0	0	0	0	0
12s	0	0	0	0	0	0

表 2-3 分为两部分，进入银行用户人数以及等待用户人数。其中进入银行用户人数，用于模拟不同用户的入队情况，例如普通用户，在 0s，2s，6s 的时间点上有新的用户进入银行，也就是进入普通用户队列。等待用户，用于描述各个时刻不同用户队列中的用户个数，以及出队列情况，例如 VIP 用户在 0s 的时候，有入队用户，但是可以办理业务，所以入队之后立即出队，但是在 6s 的时候，VIP 有用户入队，但是该时间点上，没有空闲窗口，所以需要等待到第 8s 的时候，VIP 窗口有空，才能出队。

4．详细设计

首先需要预先设置一下系统参数，包括时间，窗口类型，用户类型，以及用于表示用户到达场景的矩阵。

有了这些预定义参数之后，就可以直接定义 Simulater 中的主要模拟程序，模拟整个时间过程中的用户进入银行，接受服务，叫号的过程，流程如图 2-43 所示。

在模拟用户进入银行的过程中，将表格中的数据抽象成是一个二维数组定义为 CustomerEnerMatrix。 根据 CustomerEnerMatrix 每一行的三个元素表示在时间 t 的时间点上，三类用户进入银行的数量，将不同用户进入银行的过程抽象为三种用户在同一个时间点上依次进入各自的队列的过程，流程图如图 2-44 所示。

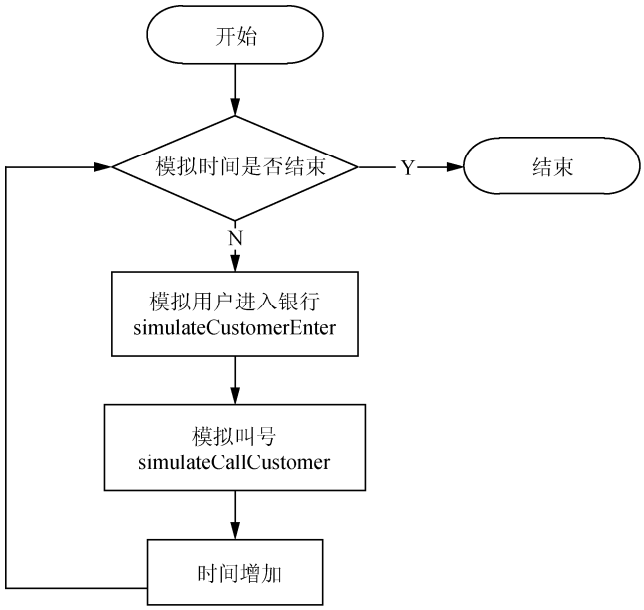


图 2-43　系统流程图

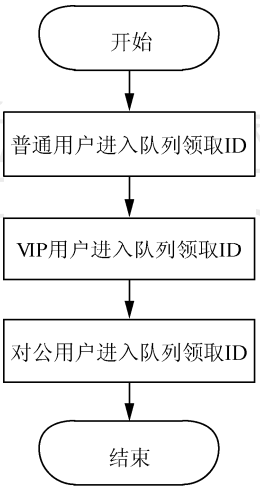


图 2-44　流程图

接下来介绍模拟叫号过程，假设一个用户被叫号之后，立刻开始接受窗口服务，处理事务，并且窗口一旦处理完用户立刻呼叫下一位用户，中间没有时间间隔。和模拟进入的结构类似，分别遍历银行里面的三类窗口，调用函数来处理用户队列，如果窗口上有正在接受服务的顾客，则显示窗口信息以及接受服务的客户信息，如果窗口上顾客结束服务后立即叫号。其流程如图 2-45 所示。

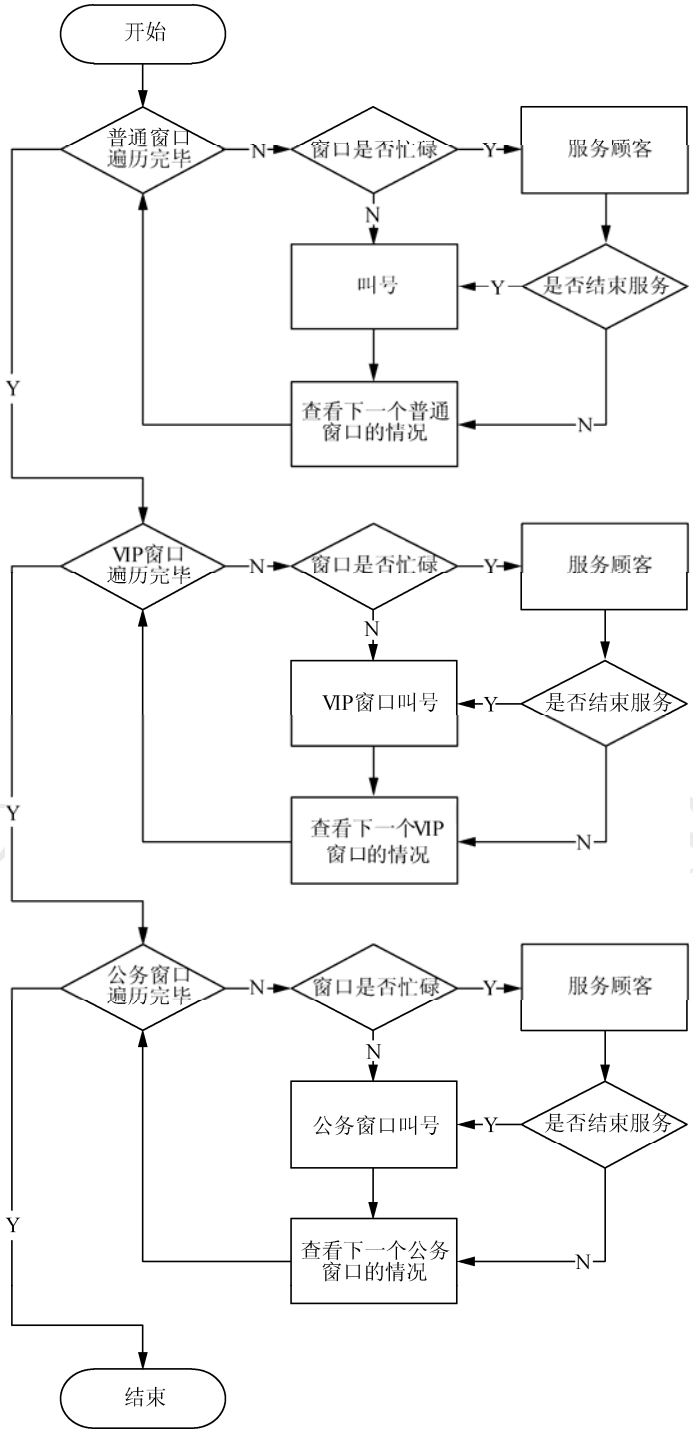

开始
普通窗口遍历完毕
N
窗口是否忙碌
Y
服务顾客
N
叫号
Y
是否结束服务
Y
查看下一个普通窗口的情况
N
Y
VIP窗口遍历完毕
N
窗口是否忙碌
Y
服务顾客
N
VIP窗口叫号
Y
是否结束服务
Y
查看下一个VIP窗口的情况
N
Y
公务窗口遍历完毕
N
窗口是否忙碌
Y
服务顾客
N
公务窗口叫号
Y
是否结束服务
Y
查看下一个公务窗口的情况
N
结束

图 2-45　流程图

不同的用户都需要接受服务，只是办理的事务不一样，因此对于用户接受服务这个事件，需要用到C++面向对象中的多态的策略，也就是在基类中定义通用方法，在各子类中，根据自身情况进行不同地定义，编写代码中注意使用这种方法。

2.7.3 字符串及链表：简易文本编辑器

1．需求描述

在使用计算机的过程中，能接触到各种各样的文字编辑软件，从系统附带的记事本和写字板到功能丰富的Word，以及Visual Studio等IDE。用户已经很熟悉这些软件的功能。

编写一个应用程序，实现一个具有常用功能的英文文本编辑软件。这个软件应当至少支持下述功能：①获取或者设置文档的名字。②移动文档的光标（插入点）并可以显示文档中当前光标的位置。③向文档中追加文本，新加入的文本将被存储在文档的末端。④在指定的位置插入文本。⑤从当前位置开始添加文本，这样原来的文本将会被覆盖。⑥在文档中查找文本，得到所查找文本在文档中首次出现的位置，如果没有找到应当返回EOF。⑦在文档中删除文本。⑧分别统计文档中不同字符各类的个数，包括字母，数字，标点符号，空白字符及总字符个数。

2．问题分析

本应用是对英文文本进行处理，故可以利用之前学习到的字符串的相关知识。另外，为了有效的实现文本的插入修改和删除功能，使用链表分块存储文本不同的部分。这里提出文本行的概念，指的是从文件开始或者一个回车之后的部分开始直至遇见新的回车符或者文件末尾结束的一段文本。链表的每个节点存储一个文本行。所有的文本行按照先后顺序进行连接形成一个链表，即链表中存储着文档中的所有文本内容。

建立一个类称之为文本编辑器，在其中添加存储着文档的链表，以及文档的大小（字符个数），文档的当前光标位置，光标所在的行列以及文档的名字。

对文档的输入输出可以通过重载流输入和输出运算符实现，也可以提供相应的方法。附加文本行的功能需要新建结点将新加入的文本行添加到文档末尾。插入文本的功能则需要判断当前是否在一个行结尾的位置，若在行结尾的位置，则将新输入的文本作为一个新文本行添加到文档链表中，否则就在当前所在的文本行中加入给定的文本。重载流输入运算符时与插入文本的逻辑基本相同。但是注意在读入文件的时候应当注意文档中包含的换行符，在遇到换行符时应当新建一个结点。在向文档加入新内容之后，都应将光标置于加入的文本之后。

文档输出可以使用重载流输出运算符实现，实现此功能时需要注意当输出一个文本行之后需要人为添加一个回车符，输出运算不应当改变文档的光标位置。

移动光标功能需要两个参数，一个是参考位置，典型的参考位置包括三个值：文档

开始，文档结束，及文档中当前光标的位置；另一个参数是偏移。最终的位置由参考位置与偏移的代数和确定。

在文档中查找过程即遍历每一个文本行节点，在每个节点中查找给定的字符串是否出现。如果找到返回匹配文本的开始位置；否则返回 EOF。查找过程应当修改光标的当前位置。

删除文本功能需要用户提供删除的起始位置和删除的文本长度。注意当用户删除的内容位于两个不同的行上时，要将删除后的两个行合并为一个新文本行。删除操作同样需要修改当前光标位置到删除的起始位置处。同时，可以提供删除功能的另一种实现，参数为需要删除的文本，可以结合上述查找功能及删除功能进行实现。

统计字符的功能可以通过判断 ASCII 码是否在特定的范围内进行实现。

根据以上分析，得到系统架构图如图 2-46 所示。

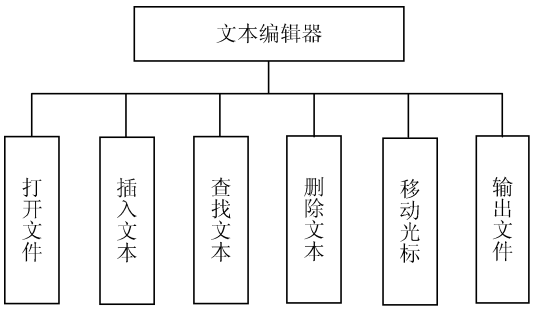


图 2-46 系统架构图

3．概要设计

首先确定系统需要哪些类，从此前的描述过程中可以提取出文本编辑器，文档，位置，节点，链表，字符串，文本等主要概念。再考虑到一个文档是一个文本编辑器的实例，因此文档类并不需要；文本也可以用字符串类表示；位置参考文件流中使用 int 类型或者 size_t 类型实现。

故本系统中有文本编辑器、节点、链表、字符串 4 个类。其中节点、链表、字符串三个类在此前的学习中已经实现过。此处只考虑文本编辑器（TextEditor）类的属性及方法。

根据上文中的分析 ，TextEditor 的成员变量至少包含：名字（name）、光标（cursor）、文章（article）；而成员函数至少包含获得名字函数 GetName、设置名字函数 SetName、获取光标位置函数 GetCursor、移动光标函数（MoveCursor）、添加文本函数 AddText、删除文本函数 DeleteText、插入文本函数 InsertText、查找文本函数 FindText、统计文本函数 WordStat 以及对流输入运算符>>和流输出运算符<<的重载。系统类图如图 2-47 所示。

4．详细设计

下面将对应用程序涉及到的关键技术进行详细解释。

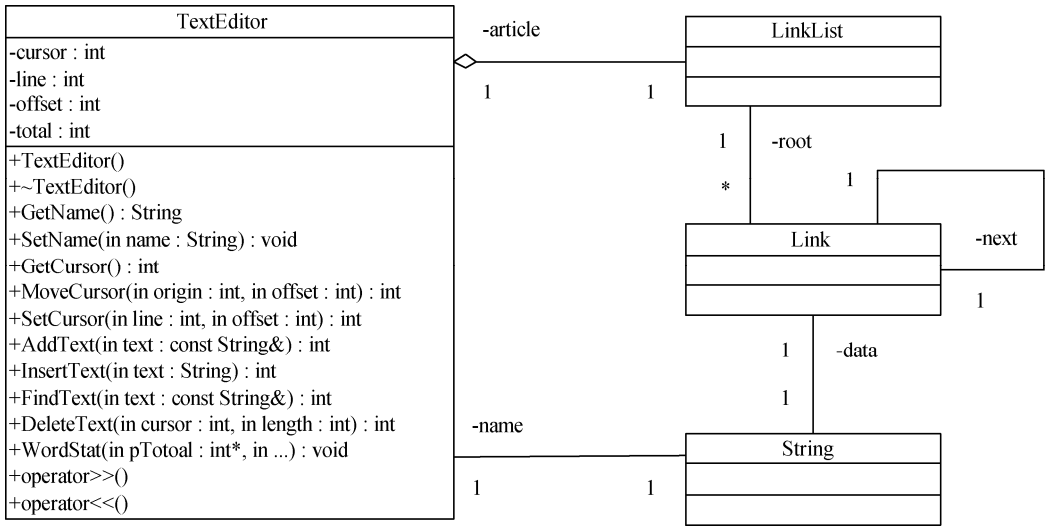


图 2-47　系统类图

下面探讨 TextEditor 类的具体实现方式。选取其中较为复杂的 InsertText 函数及 DeleteText 函数进行实现。TextEditor 的类声明如下所示：

```
class TextEditor
{
private:
    Linklist<String> article;
    int cursor;
    int line;
    int offset;
    int total;
    String name;

public:
    const string& GetName() const;
    void SetName(const String& name);

    int GetCursor(int *pLine=NULL, int *pOffset=NULL) const;
    int MoveCursor(int origin,int offset);
    int SetCursor(int line,int offset);

    int AddText(const String& text);
    int InsertText(String text);

    int FindText(const String& text) const;

    int DeleteText(const String& text);
    int DeleteText(int cursor, int length);

    void WordStat(int* pTotal,int* pLetter=NULL,int* pDigit=NULL,
        int* pSpace=NULL, int* pQuot=NULL) const;

    friend ostream& operator<<(ostream& out, TextEditor& editor);
    friend istream& operator>>(TextEditor& editor, istream& in);
```

```
    TextEditor(void);
    ~TextEditor(void);
};
```

首先介绍 MoveCursor 函数的逻辑，如图 2-48 所示。

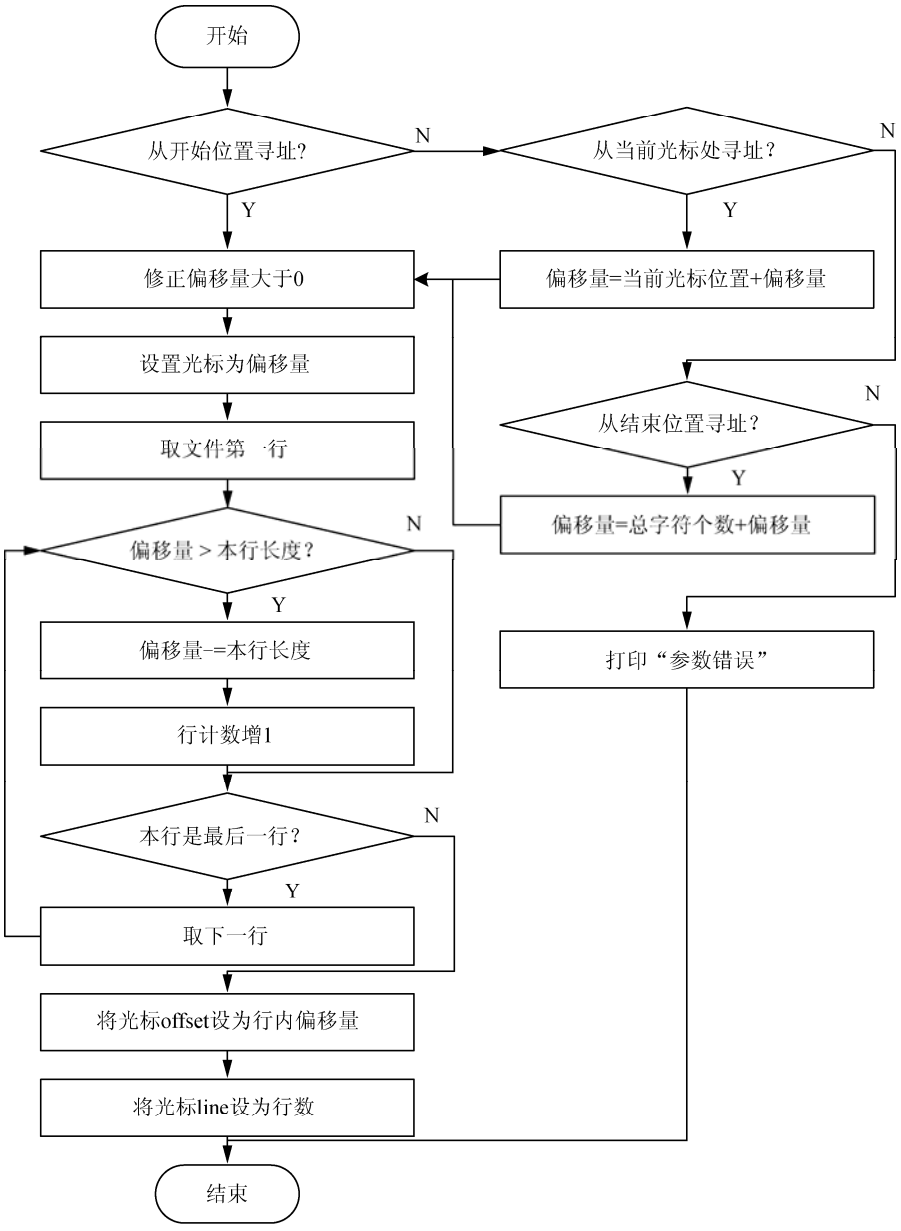


图 2-48　MoveCursor 函数逻辑图

图 2-49 介绍了 InsertText 函数的实现方式。

图 2-50 介绍了 DeleteText 的实现方式。

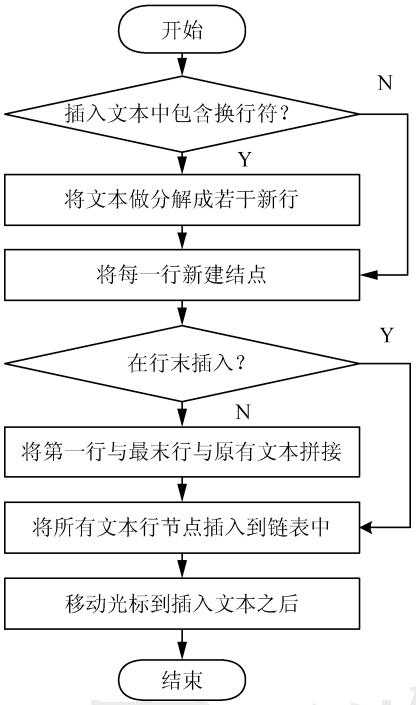


图 2-49　InsertText 函数逻辑图

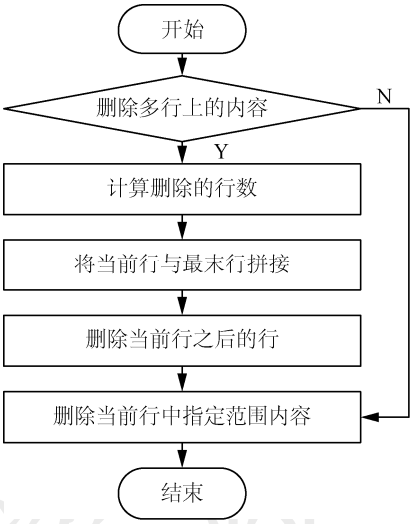


图 2-50　DeleteText 函数逻辑图

其他函数可以仿照以上几个函数的实现方式进行实现。

习　　题

1．线性表可用于顺序表或链表存储，试问：

（1）两种存储表示各有哪些主要优缺点？

（2）如果有 n 个表同时并存，并且在处理过程中各表的长度会动态发生变化，表的总数也可能自动改变，在此情况下，应选用哪种存储表示？为什么？

（3）若表的总数基本稳定，且很少进行插入和删除，但要求以最快的速度存取表中的元素，应采用哪种存储表示？为什么？

2．利用顺序表的操作，实现以下函数：

（1）从顺序表中删除具有最小值的元素并由函数返回被删元素的值，空出的位置由最后一个元素填补。

（2）从顺序表中删除具有给定值 x 的所有元素。

（3）从有序顺序表中删除其值在给定值 s 与 t 之间（s<t）的所有元素。

3．给定一个不带头结点的单链表，写出将链表倒置的算法。

4．已知 head 为单链表的表头指针，链表中存储的都是整形数据，实现下列运算的递归算法：

（1）求链表中的最大值。

（2）求链表中的结点个数。

（3）求所有整数的平均值。

5．设 A 和 B 是两个单链表，其表中元素递增有序。试写一算法将 A 和 B 归并成一个按元素值递减有序的单链表 C，并要求辅助空间为 O(1)，试分析算法的时间复杂度。

6．设 ha 和 hb 分别是两个带头结点的非递减有序单链表的表头指针，试设计一个算法，将这两个有序链表合并成一个非递减有序的单链表。要求结果链表仍使用原来两个链表的存储空间，不另外占用其他的存储空间。表中允许有重复的数据。

7．设双链表表示的线性表 $L=(a_1,a_2,\ldots,a_n)$，试写一时间复杂度为 O(n)的算法，将 L 改造为 $L=(a_1,a_3,\ldots,a_n,\ldots,a_4,a_2)$。

8．已知有一个循环双链表，p 指向第一个元素之为 x 的结点，设计一个算法从该循环双链表中删除*p 结点。

9．内存中一片连续空间（不妨假设地址从 1 到 m），提供给两个栈 S_1 和 S_2 使用，怎样分配这部分存储空间，使得对任一个栈，仅当这部分空间全满时才发生上溢。

10．简述线性表、栈和队列的异同？

11．设有一个数列的输入顺序为 123456，若采用栈结构，并以 A 和 D 分别表示进栈和出栈操作，试问通过进栈和出栈操作的合法序列有多少种？能否得到输出顺序为 325641 的序列？能否得到输出顺序为 154623 的序列？

12．设计算法把一个十进制整数转换为二至九进制之间的任意进制数输出。

13．假设表达式中允许包含 3 种括号：圆括号、方括号和大括号。设计一个算法采用顺序栈判断表达式中的括号是否正确配对。

14．由用户输入 n 个 10 以内的数，每输入 i($0\leqslant i\leqslant 9$)，就把它插入到第 i 号队列中。最后把 10 个队列中非空队列，按队列号从小到大的顺序串接成一条链，并输出该链的所有元素。

15．设计一个环形队列，用 front 和 rear 分别作为队头和队尾指针，另外用一个 tag 表示队列是空（0）还是不空（1），这样就可以用 front==rear 作为队满的条件。要求设计队列的相关基本运算算法。

16．已知 t="abcaabbcabcaabdab"，求模式串的 next 数组值。

17．设计 Strcmp(s,t)算法，实现两个字符串 s 和 t 的比较。

18．设计一个算法，在串 str 中查找字串 substr 最后一次出现的位置（不能使用 STL）。