第二章 线性表

刘亮亮

2020 年 3 月 15 日

- 要点：本章主要介绍第一种线性结构——线性表，线性表的抽象数据类型定义、线性表的顺序存储结构（顺序表）的表示与实现、线性表的链式存储结构的表示与实现，以及介绍双向链表、循环链表，及其线性表的应用。
- 重点：1.顺序表的表示与实现；2.链表的表示与实现;
- 难点：1 .插入算法与删除算法

在现实生活中，有很多的线性结构，比如我们的各种表格、电话号码簿、排队等都是线性结构。线性结构的特点是一对一的关系，也就是说是除第一个元素外，每一个元素都有且仅有一个直接前驱，除最后一个元素外每个元素有且只有一个后继。称为第一个的元素没有前驱，称为最后一个的元素没有后继。

线性表是一种最简单灵活的线性结构，是零个或多个元素组成的有限序列。下面就介绍线性表的表示与实现。

1 线性表的抽象数据类型定义

线性表是$n(n >= 0)$个数据元素的有限序列。具体的数据元素的类型由具体的问题来决定，可以是基本类型（例如：整数、字符等），也可以是复合类型（如字符串）、自定义类型（如学生表里面的学生类型）。不论数据元素是什么类型，都具有如下形式：

$$SqList = (a_1, a_2, ..., a_{i-1}, a_i, a_{i+1}, a_n)$$

其中a_{i-1}是a_i的直接前驱，a_{i+1}是a_i的直接后继（注：如果没有特殊说明，前驱就是指直接前驱，后继就是指直接后继）

例 1 (整数序列)**.** *lst=(10,20,30,3,5,7,9)*

例 2 (句子的单词序列)**.** *sentence=(*“我”,“爱”,“中国”*)*

根据抽象数据类型的定义的描述，对线性表的抽象数据定义描述如下：

ADT SqList{

数据对象:$D = \{a_i | a_i \in DataType\}$

数据关系:$R = \{< a_i, a_{i+1} >\}$

基本操作：

$init(\&L)$:

操作结果：建立空线性表L，创建成功返回true,并且用L返回，否则返回false

$destroy(\&L)$:

初始条件：线性表L存在

操作结果：删除一个线性表L,删除成功返回true,否则返回false

$clear(\&L)$:

初始条件：线性表L存在

操作结果：清空线性表L，清空成功返回true,否则返回false

$isEmpty(\&L)$:

初始条件：线性表L存在

操作结果：如果线性表L为空，返回true,否则返回false.

$length(\&L)$:

初始条件：线性表L存在

操作结果：返回线性表L的长度

$get(\&L, \&e, index)$:

初始条件：线性表L存在

操作结果：返回线性表L的第index的元素

$find(\&L, e)$:

初始条件：线性表L存在

操作结果：在线性表L中查找e，返回e在线性表L的索引号

$proir(\&L, e, \&priorE)$:

初始条件：线性表L存在

操作结果：在线性表L中查找e，并用$priorE$返回e的前驱,否则操作失败

$next(\&L, e, \&nextE)$:

初始条件：线性表L存在

操作结果：在线性表L中查找e，并用$nextE$返回e的后继,否则操作失败

$insert(\&L, i, e)$:

初始条件：线性表L存在

操作结果：在线性表L中第i个元素之前插入元素e

$delete(\&L, i, \&e)$:

初始条件：线性表L存在

操作结果：删除线性表L中第i个元素，用e返回

}End SqList.

定义一个数据结构的时候，一般而言需要把基本操作都定义出来，因为不同的应用有不一样的操作，而这些操作可以通过这些基本操作组合来进行实现。看如下示例。

例 3. 两个集合用线性表来表示，分别表示La和Lb，现利用线性表的基本操作，实现将Lb合并到La中。

分析：大家想想现实生活中将两个集合合并你是怎么来实现的？比如说将两个篮子里面的水果归并在一起，相同的不放进去。依次取出Lb 中的元素，若Lb 中的元素在La 中存在，则不放进去。否则，将该元素放入在La 中，因此在整个合并操作过程中，可以使用线性表的基本操作，求长度$length$、取元素get、 查找元素$find$、 插入元素$insert$ 等基本操作来完成。大家注意到$insert$操作时候，在什么位置插入，因为集合是无序的，所以我们只需要简单在La的表末进行插入就可以。

算法思想：

Step1 分别求得La和Lb的长度；
Step2 依次取出Lb中的元素e
Step3 判断e是否在La中：
　　(1)如果不存在，则插入La的表尾；
　　(2)否则，继续；
Step4 重复Stp2 Step3,直到Lb中的元素取出来了。

算法实现如下：

Algorithm 1 利用线性表实现两个集合A和B的合并$union(\&La, \&Lb)$

Input: 用线性表La表示集合A; 用线性表Lb表示集合B;

Output: 合并后的线性表La;

```
1: lenA=length(La);
2: lenB=length(Lb);
3: for (i = 0; i < lenB; i++) do
4:     get(Lb,i,e); //取出Lb中的第i个元素
5:     if (!find(Lb, e)) then //如果e不在线性表La中进行插入
6:         insert(La, lenA++, e) //插入元素在线性表La的末尾
7:     end if
8: end for
9: return Status
```

例 4. 两个有序表的合并，合并后得到的线性表仍然是有序的。

分析：例如$La=\{1,5,10,21,30\}$，$Lb=\{3,5,15,21,40,53,70\}$,$La$和$Lb$，都分别有序，现将$La$和$Lb$进行合并生成一个新的线性表$Lc$，使得$Lc$也是有序的。大家想类似这个问题：两队人，都依次是从低到高有序，现需要将两队归并成一个队，并且使得两队人排成一队后，仍然从低到高有序，我想这种例子在站队的时候经常出现，问题是这种问题怎么来做？由于两队都是有序的，因此一开始的时候需要分别设两个指针分别指向这两个队，然后比较指向的人的身高，个子较矮的人先插入到新的队里，插入后插入元素所在的队的那个指针移动到下一个，没插入的保持位置，然后再比较，一直到有一个队列中的元素全部取出来了，这样还有人的队列每个人都取出来依次插入到表末就可以，因为剩余的人肯定都比已经插入的人的个子都要高。其实在设计这类算法的时候，需要找出操作的规律，然后让计算机重复来做，计算机最擅长的是循环做同样的事情（步骤）。

具体的La和Lb合并过程图示如下：

算法思想：

Step1 初始化一个空的列表Lc
Step2 分别求得La和Lb的长度；
Step3 设置两个整数变量i和j，初始值为0；
Step4 取得La的第i个元素a_i,Lb的第j个元素b_j
Step5 判断a_i和b_j的大小:
　　(1)若$a_i <= b_j$，则插入a_i到Lc的表末，并且i往后移，j保持不变；
　　(2)否则，插入b_j到Lc的表末，并且j往后移，i保持不变；
Step6 重复Step4 Step5,直到La或Lb达到了表末，表示每个元素都取出来;
Step7 如果La未到达表末，则依次取出La中的每个元素依次插入到Lc中;
Step8 如果Lb未到达表末，则依次取出Lb中的每个元素依次插入到Lc中。

算法实现如下：

```
Status union(SqList& Lc,SqList& La,SqList& Lb){
    Init(Lc);
    lenA=length(La);
    lenB=length(Lb);
    lenC=length(Lc);
    i=0,j=0;
```

```
    while(i<lenA&&j<lenB){
        get(La,ai,i);
        get(Lb,bj,j);
        if(ai<=bj){
            insert(Lc,lenC++,ai);//插入元素ai在Lc的末尾
            i++; //i往后移
        }
        else{
            insert(Lc,lenC++,bj);//插入元素bj在Lc的末尾
            j++;
        }
    }
    while(i<lenA){//插入La剩余的部分
        get(La,ai,i);
        insert(Lc,lenC++,ai);
        i++;
    }
    while(j<lenB){//插入La剩余的部分
        get(Lb,bj,j);
        insert(Lc,lenC++,bj);
        j++;
    }
    return OK;
}
```

从上面的算法可知，可以采用基本操作*length*、*get*、*insert*等就可以实现这个两个有序表的合并更为复杂一点的操作。

因此，通过线性表的基本操作可以实现一些更复杂的操作。因此在数据结构定义的时候，需要定义这个数据结构所有的基本操作。

以上只是线性表的抽象数据类型的定义与实现，但是线性表的存储结构是怎么样的？如何在计算机中存储的，不同的存储结构操作的实现是不是相同？下面我们分别介绍线性表的两种存储结构及实现。根据存储结构的分类，分为顺序存储结构和链式存储结构，那么，线性表也可以分别采用这两种存储方式来进行存储。我们利用顺序存储结构来存储的线性表称

为顺序表，利用链式存储结构来存储的线性表称为链表。下面我们分别来介绍这两种结构。

2 线性表的顺序表示与实现

定义 1 (线性表的顺序存储结构). 利用一组地址连续的存储单元来依次存放线性表中的数据元素，称为线性表的顺序表示。

定义 2 (顺序表). 这种利用顺序存储结构来表示的线性表，简称为顺序表。

2.1 线性表的顺序表示

线性表的顺序存储表示是用物理上相邻来表示其逻辑上相邻。假如一个线性表$List = (a_1, a_2, ..., a_n)$，如果每个元素所占的存储单元为$L$个，如果第一个元素占用的起始地址为$Loc(a_1)$，那么$a_2$的起始地址为$Loc(a_1) + L$，依次类推，$a_i$的起始地址为$Loc(a_1) + (i-1) * L$，而$a_{i+1}$的起始地址为$Loc(a_1) + i * L$，可以看到$a_i$与$a_{i+1}$逻辑上相邻，在物理上也是相邻的，从而通过这种物理上相邻来反映逻辑上相邻。

下面我们首先来看看线性表的顺序结构的结构定义，也可以分为两种，一种是静态分配，一种是动态分配，静态分配其大小是固定的，动态分配是可以进行扩充线性表的容量。

```
#define MAXSIZE 50   /*分配的最大值*/
typedef int ElemType;//根据具体应用来定义类型，这里定义成int
typedef struct SqList{
    ElemType data[MAXSIZE]; /*利用数组来进行存放*/
    int length; /*线性表的当前长度，表示实际长度*/
}SqList;
```

上面这个表示是静态分配的数组，线性表的最大容量是固定的。也可以定义动态分配，在运行的过程中可以对线性表的容量进行扩充。我们后面的算法的介绍以下面这个定义来进行实现。

```
#define INITSIZE 50   /*分配的最大值*/
```

```
#define INCREMENT 20 /*分配增量*/
typedef int ElemType;//根据具体应用来定义类型，这里定义成int
typedef struct SqList{
    ElemType *data; /*存储的起始地址*/
    int length;    /*线性表的当前长度，表示实际长度*/
    int listSize; /*当前分配的容量*/
}SqList;
```

根据上面的描述，定义一个线性表的顺序存储结构的结构类型需要定义以下几个属性：

- 存储空间的起始地址：*data表示当前存储空间的起始地址。
- 线性表的当前长度：线性表实际存储了多少个数据元素。
- 当前分配的存储大小：当前线性表的容量大小。

下面分别介绍顺序存储结构下的基本操作的实现。

2.2 基本操作的实现

在介绍算法之前，先定义一些预定义的常量（没有特殊说明，后面的例子不再定义这些常量和符号）。

```
#define OK 1
#define ERROR 0
#define OVERFLOW -1
#define TRUE 1
#define FALSE 0
#define INITSIZE 50   /*分配的最大值*/
#define INCREMENT 20 /*分配增量*/
typedef int Status;
```

2.2.1 初始化顺序表

首先看第一个基本操作，构建一个空的顺序表，就是定义一个初始分配空间大小的线性表，该线性表当前长度为0，当前分配的大小为初始容量大小。具体算法实现如下：

```
Status init(SqList &L){
    //构造一个空的线性表
    L.data =(ElemType*)malloc(INITSIZE*sizeof(ElemType
        ));
    if(!L.data){
        exit(OVERFLOW);//分配失败
    }
    //分配成功，初始化结构的基本属性
    L.length = 0;
    L.listSize=INITSIZE;
    return OK;
}
```

注：*malloc*函数是*C*语言的动态分配的函数，用于申请一块连续的指定大小的内存块区域以*void**类型返回分配的内存区域地址，当无法知道内存具体位置的时候，想要绑定真正的内存空间，就需要用到动态的分配内存，且分配的大小就是程序要求的大小。

上面初始化的操作*init*主要是初始化一个空的顺序表，需要分配存储空间，如果分配存储空间失败，初始化也就不成功。只有初始化成功了，才可以将数据元素放入到线性表中。

2.2.2 取元素操作

取元素操作是取得线性表中第i个元素,因为采用的顺序存储结构，取元素操作很方便，直接通过下标就可以取元素。但是算法的要求需要健壮性，因此在设计算法的时候需要考虑一些参数是否满足条件，对参数进行合法性检查。在取元素操作，参数i必须在表长范围内，否则取元素就会出现越界错误。具体的算法实现如下：

```
Status get(SqList &L,int i,ElemType& e){
    /*如果是空表*/
    if(L.length==0){
        return FALSE;
    }
    /*判断参数是否合法i*/
    if(i<0||i>=L.length){
        return FALSE;
    }
    e=*(L.data+(i-1));
    return OK;
}
```

2.2.3 查找元素操作

查找是一种最基本的操作，给定一个元素，查找该元素是否在线性表中出现，如果出现返回该元素在线性表中的位置，如果线性表中不存在该元素就返回−1，表示查找失败。

该操作比较简单，从第一个元素开始依次往后比较，如果相等就表示查找成功，返回其位置，如果超过表长，表示线性表中没有该元素。

```
Status find(SqList &L,ElemType e){
    int i=0;
    while(i<L.length){
        if(L.data[i]==e){
            return i;
        }
        i++;
    }
    return -1;
}
```

2.2.4 插入元素操作

插入元素操作是线性表的一个重点操作，首先看操作的定义，在线性表L的i个位置之前插入一个元素e,定义是$insert(SqList\ \&L, int\ i, ElemType\ e)$。下面我们来分析插入过程。

线性表$L = (a_1...,a_{i-1},a_i...a_n)$，经过在第$i$个位置前面插入一个元素$e$后，得到$L = (a_1,...a_{i-1},e,a_i,...,a_n)$, 也就是在$a_{i-1}$和$a_i$之间插入元素$e$。

分析：正如我们在一个排列的队伍中插入一个人一样，首先我们需要从第i个元素到最后的元素进行移动，也就是说把第i个位置腾出来，然后将插入的人站在那个位置上，我们怎么移动呢？肯定是最后的那个人先往后移动一个位置，依次类推。另外参数i是否是合法的位置也需要在插入前进行判断。线性表的容量也必须进行判断，是否有空间来进行插入新的元素。根据这些分析，算法思想如下：

算法思想：

Step1 判断i的位置是否合法，如果不合法退出；
Step2 判断表L的存储空间是否满了，如果满了需要重新分配
Step3 如果重新分配失败，则返回；
Step4 从顺序表的最后一个元素到第i个元素依次向后移动一个位置
Step5 将待插入的元素赋值给第i个位置
Step6 表长增1。

算法描述如下：

```
1  Status insert(SqList &L,int i,ElemType e){
2      //判断i的位置是否合法
3      if(i<0||i>L.length){
4          return FALSE;
5      }
6      /*判断存储空间是否满了，满了则重新分配*/
7      if(L.length>=listSize){
8          newbase=(ElemType*)realloc(L.data,(listSize+
               INCREMENT)*sizeof(ElemType));
9          /*如果分配失败*/
10         if(!newbase){
11             exit(OVERFLOW);
```

```
12            }
13            L.data=newbase;
14            L.listSize += INCREMENT;
15        }
16        p = &(L.data[i-1]);//利用p保存插入位置
17        for(q=&(L.data[L.length-1]);q>=p;q--){
18            *(q+1)=*q;
19        }
20        *p = e;//插入e
21        L.length++;//表长加1
22        return OK;
23    }
```

注：*realloc*函数是*C*语言函数，可以重用或扩展以前用*malloc()*、*calloc()*、*realloc()*函数自身分配的内存。函数需两个参数：一个是包含地址的指针（该地址由之前的*malloc()*、*calloc()*或*realloc()*函数返回），另一个是要新分配的内存字节数。

设计完一个算法后，需要对算法进行算法分析，根据第一章的内容，算法分析主要做时间分析和空间分析，在这里我们分析算法的时间复杂度。

对于插入算法，主要的时间耗费在移动元素的操作上，对于在指定的位置i前插入一个元素，那么需要移动$n-i+1$次，最好的情况，在表末进行插入，移动次数为0，时间复杂度为$O(1)$；最坏的情况在表头插入，这样每一个元素都需要进行移动，移动的次数为n 次，时间复杂度为$O(n)$。因此移动次数和i有关系，我们不能用$O(n-i+1)$来表示插入的时间复杂度，那怎么度量呢？

这里，我们定义‘平均移动次数‘来衡量算法的时间复杂度，平均移动次数是这样来定义的：假如在第i个(i取：$1...n+1$)位置插入的概率是p_i,那么移动次数的期望值是：

$$E=\sum_{i=1}^{n+1} p_i*(n-i+1)$$

如果我们假设在每个位置插入的概率相等，也就是$p_i=\frac{1}{n+1}$,代入上面的式子，进行整理后，可以得到

$$E = \frac{n}{2}$$

这就是插入算法的平均移动次数，那么插入算法的时间复杂度为$O(n)$。根据算法分析，我们可知，插入算法的时间主要在移动元素上，如果表中数据很多，而需要反复进行插入操作，很显然选择顺序表不是一个很好的存储结构，因为元素的大量移动（平均需要移动一半的元素），实际上是在做大量的赋值操作，这个赋值操作本质上挺耗费时间。

2.2.5 删除元素操作

删除线性表的指定的元素也是线性表中的重点操作之一，先通过现实生活中的例子来看一下删除操作的定义，一队人在排队买票，某个人（如不失一般性，第i个）不想排了，他就离开了队伍，他后面的人会移动到他的位置，依次类推，这样队列仍然保持线性关系，而他之前的人的位置没有发生改变。因此，删除第i个元素的操作只需要第$i+1$个元素到第n个元素依次向前移动一个位置，就可以实现删除操作。算法思想如下：

算法思想：

Step1 判断i是否在表长范围之内，如果不是退出，无法进行删除；
Step2 保存待删除的元素
Step3 从顺序表的第$i+1$个元素到第n个元素依次向前移动一个位置
Step4 表长减1。

算法描述如下：

```
Status delete(SqList &L,int i,ElemType& e){
    //判断i的位置是否合法
    if(i<1||i>L.length){
        return FALSE;
    }
    p = &(L.data[i-1]);//利用p指向待删除的元素
    e = *p;//保存待删除的值返回
    q=L.data+L.length-1;   //q指向最后一个元素
    //从i+1个元素到第n个元素向前移动
    for(++p;p<=q;++p){
        *(p-1)=*p;
```

```
    }
    L.length--;//表长减1
    return OK;
}
```

和顺序表的插入算法一样，删除算法最主要的操作也是在移动元素，因此算法分析和插入算法分析类似，也是采用平均移动次数来计算，删除第i个元素的需要移动$n-i$次，假如任何位置上删除是等概率，则平均移动次数为：

$$E=\sum_{i=1}^{n}p_i*(n-i)=\frac{1}{n}\sum_{i=1}^{n}(n-i)=\frac{n-1}{2}$$

因此，删除操作的时间复杂度为$O(n)$。

2.2.6 应用举例

例 5. 利用顺序表来实现两个集合的合并

分析：回顾上一节两个集合A和B的合并，将B中的元素合并到A中去，在上一节里使用基本操作来实现，现在A和B是采用顺序存储结构来，那如何实现A和B的合并。

具体的算法思想如下：

Step1 判断La当前的容量是否大于或等于La的当前长度加上Lb的当前长度，否则进行重新分配；
Step2 依次取出Lb的元素，在La中进行查找，
Step3 如果相等，则不进行插入，否则插入在表末
Step4 重复执行Step2 Step3。

```
Status union2Set(SqList &La,SqList &Lb){
    //判断La的空间是否有空间来进行插入
    if(La.listSize<La.length+Lb.length){
        newbase = (ElemType*)realloc(La.data,(La.
            length+Lb.length)*sizeof(ElemType));
        if(!newbase){
            exit(OVERFLOW);
        }
        La.data=newbase;
```

```
9       }
10      //将Lb的元素依次取出来进行判断
11      for(int j=0;j<Lb.length;j++){
12          bj=Lb.data[j];
13          for(int i=0;i<La.length;i++){
14              ai=La.data[i];
15              //如果bj和ai相等，则不进行插入
16              if(bj==ai){
17                  break;
18              }
19          }
20          //比较完后，La中没有元素和bj相等,则在La的表末进行插入，
                插入完后表长增1
21          if(i>=La.length){
22              La.data[La.length++]=bj;
23          }
24      }
25      return OK;
26  }
```

算法分析：从算法中看，主要在依次取出Lb中的元素在La中依次进行比较，假设表长都是n，则最大的比较次数为$n*n$，因此时间复杂度为$O(n^2)$。

例 6. 利用顺序表来实现两个有序表的合并。

分析：两个有序表La和Lb合并到一个新的表Lc中，新的表仍然要保持有序。想想现实生活中的例子，你是如何来实现的？比如两个队列都按从身高从矮到高排列，现在要将两个队伍合并成一个新的队伍。因为是有序，我们依次从两个队列中各取出一个人，来比较两个人的个子，如果矮那么其他人肯定都比他高，我们只需要把矮的人插入在新的队列中的尾部，然后插入过的那个队伍取下一个人再和另外一个队伍的刚才那人进行比较，一直重复这样的操作，是不是可以将人从矮到高插入到新的队伍中。当一个队伍都插入完了，另一个队伍还有人时候，表示这些剩下的人都不比已插入的矮，我们只需要依次取出进行插入就可以。

$$La = (1, 3, 5, 7, 9)$$

$$Lb = (2, 4, 5, 9, 11)$$

首先取出La中的第一个的元素$a_i = 1(i = 0)$,Lb中的第一个元素$b_j = 2(j = 0)$,进行比较，a_i小，则插入a_i到表Lc中，此时$Lc = (1)$, 然后取出La 的第二个元素$a_i = 3$和$b_j = 2$进行比较，b_j小，插入b_j到表Lc中，则$Lc = (1, 2)$，依次类推.....

具体的算法思想如下：

Step1 初始化Lc，对Lc分配$La.length + Lb.length$存储空间；
Step2 如果分配失败，退出合并，返回-1；
Step3 设立指针i指向La的第一个元素，指针j指向Lb的第一个元素;
Step4 依次取出a_i,b_j，进行比较。
Step5 如果$a_i <= b_j$，插入a_i到Lc中，然后i指针后移，j保持不变；
Step6 否则插入b_j到Lc中，然后j指针后移，i保持不变；
Step7 重复Step4 Step6,直到某个表到达表末（$i >= L.length$或者$j >= L.length$）
Step8 如果La还有元素($i < L.length$)，则将La中的元素依次取出插入到表Lc中
Step9 如果Lb还有元素($j < L.length$)，则将Lb中的元素依次取出插入到表Lc中

算法描述如下：

```
Status union(SqList &Lc,SqList &La,SqList &Lb){
    //分配Lc的空间为La和Lb的长度之和
    Lc.data = (ElemType*)malloc((La.length+Lb.length)*
        sizeof(ElemType));
    if(!Lc.data){
        exit(OVERFLOW);
    }
    Lc.length=0;
    Lc.listSize=La.length+Lb.length;

    i=0;
    j=0;

```

```
13      while(i<La.length&&j<Lb.length){
14          ai=La.data[i];
15          bj=Lb.data[j];
16
17          if(ai<=bj){
18              Lc.data[Lc.length++]=ai;
19              i++;
20          }
21          else{
22              Lc.data[Lc.length++]=bj;
23              j++;
24          }
25      }
26      //如果La未到表末,依次取出插入到Lc的表末
27      while(i<La.length){
28          ai=La.data[i];
29          Lc.data[Lc.length++]=ai;
30          i++;
31      }
32      //如果Lb未到表末，依次取出插入到Lc的表末
33      while(j<Lb.length){
34          bj=Lb.data[j];
35          Lc.data[Lc.length++]=bj;
36          j++;
37      }
38      return OK;
39  }
```

3 线性表的链式表示与实现

本节介绍线性表的链式存储结构，链式存储结构是开辟一组任意的存储单元（不一定连续）来存放，那如何表示顺序表呢，并且要反映线性表的“一对一”的关系？线性表的链式表示简称单链表。

3.1 单链表的表示与实现

单链表是一种线性表的链式存储结构的数据结构，链表中的数据是以结点来表示的，每个结点的构成：元素域data(数据元素的映象) + 指针域next(指示后继元素存储位置)，元素就是存储数据的存储单元，指针就是连接每个结点的地址数据。以“结点的序列”表示的线性表称作线性链表（单链表），单链表是链式存取的结构。

其中数据域data,可以存储想要存储的任何数据类型，可以是整型等简单标准类型，也可以是自定义的结构类型。next是一个指针，其代表了可以指向的一个地址，通常指向下一个结点（当前节点的后继），利用指针指向来表示逻辑上相邻，区别于顺序表逻辑上相邻物理上也相邻，单链表的元素的地址可以连续也可以不连续。链表的尾部的指针域next指向空(NULL)，因为尾部后面已经没有元素了。

单链表的组成都是结点结构组成，因此结点的代码描述如下：

```
typedef struct Node{
    ElemType data; /*数据域*/
    struct Node* next;    /*单链表的指针域*/
}Node,*LinkedList;
//Node表示结点的类型，LinkedList表示指向Node结点类型的指针类型
```

为了操作方便，通常在第一个结点前面附设一个结点，该结点数据域可以不存储信息，也可以存储链表长度等信息，该结点的指针域指向单链表的第一个结点。因此只需要知道头结点的地址，沿着指针可以访问单链表的所有结点。没有特殊说明一般链表都带头结点。

加入L为头指针，头指针指向头结点，那么带结点的单链表为空的条件是：$L->next == NULL$。

如果p指向当前节点，则$p->data$表示当前节点中存储的数据域，而$p->next$表示其后继结点的地址，则$p->next->data$表示后继结点的数据域，$p->next->next$表示后继结点的后继，依次类推。

3.2 基本操作的实现

3.2.1 初始化链表

任何的结构，类型一样，链表也需要初始化操作，初始化是创建一个

单链表的头结点，一般来说，我们所谓的初始化单链表一般指的是申请结点的空间，同时对头结点的指针域辅以空值(NULL)，其代码可以表示为：

```
LinkedList listInit(){
    Node* L;
    L=(Node*)malloc(sizeof(Node));
    if(!L){//申请空间失败
        exit(OVERFLOW);
    }
    L->next=NULL;//头结点的指针域为空
    return L;
}
```

3.3 取元素算法

顺序表取第i个元素比较方便，直接通过下标就可以取的，而在单链表中取第i个元素，相对来说复杂一点，需要通过遍历来取得，我们无法知道第i个元素存储在哪个存储单元，取第i个，需要首先找到第i-1个，因为第i个的存储位置，第i-1个元素的指针域next里面存储了，依次类推，第i-1个需要先找到第i-2个.....，那么我们可以通过头结点一直往后找，一直到第i个就可以。因此算法思想如下：

Step1:初始化一个指针p指向头结点以及初始化一个计数器j;

Step2:循环移动指针，进行计数，直到计数器j的值等于i或者p指向表尾了。

Step3:如果p指向表尾或者计数器大于i，查找失败；

Step4:否则查找成功，返回值。

```
Status get(LinkedList L,ElemType& e,int i){
    //L为带头结点的单链表的头结点
    p=L;
    j=0;
    //遍历
    while(p&&j<i){
```

```
7           p=p->next;
8           j++;
9       }
10      //如果元素不存在
11      if(!p||j>i){
12          return ERROR;
13      }
14      //查找成功，返回第i个元素
15      e=p->data;
16      return OK;
17  }
```

算法分析：查找算法主要的时间耗费在遍历查找上，如果表长为n,如果$1 <= i <= n$,查找成功的次数为$i-1$，否则查找失败，查找次数为n，则时间复杂度为$O(n)$。

3.4 查找算法

查找算法是给定一个元素，查找该元素从指定的位置起是否在链表中出现，这个也是采用遍历的思想，从头结点开始进行查找。不同的时候，需要从指定的起始位置进行查找。

具体的算法实现如下：

```
1   Status find(LinkedList L,ElemType e,int pos){
2       //L为带头结点的单链表的头结点,e为待查找的元素，pos为指定的位置
3       p=L;//指向头结点
4       j=0;//计数器
5       //遍历,找到第pos位置
6       while(p&&j<pos){
7           p=p->next;
8           j++;
9       }
10      //起始位置不符合要求，超过表长范围
11      if(!p||j>pos){
```

```
12          return ERROR;
13      }
14      //继续查找
15      while(p&&p->data!=e){
16          p=p->next;
17          j++;
18      }
19      //查找失败
20      if(!p){
21          return ERROR;
22      }
23      return j;//返回位置
24  }
```

这个算法的复杂度为$O(n)$。

3.5 插入算法

首先回顾一下插入操作，是在第i个位置前面插入一个元素。根据单链表的特点，插入操作要分成两步，第一步需要找到插入位置，第二步将插入的元素插入到位置上。第一步找到插入位置，因为是在第i个元素前面插入，因此我们需要找到第$i-1$个元素（想想为何不找第i个元素），因此需要通过遍历思想，从头结点依次找到第i-1个结点。第二步是插入一个结点，首先要对待插入的节点开辟存储空间，然后将该结点连接到单链表上，所谓的连接就是让结点的指针域next进行修改。算法思想如下：

Step1:依次查找第i-1个元素

Step2:查找失败，则返回——插入失败

Step3:生成新的待插入的结点

Step4:插入到链表中

```
1   Status insert(LinkedList L,int i,ElemType e){
2       //L为带头结点的单链表的头结点
3       p=L;
4       j=0;     //j为计数器
```

```
5         //遍历，沿指针后移,找第i-1个结点
6         while(p&&j<i-1){
7             p=p->next;
8             ++j;
9         }
10        //i小于1或者大于表长+1
11        if(!p||j>i-1){
12            return ERROR;
13        }
14        //申请结点
15        s=(LinkList)malloc(sizeof(LNode));
16        s->data=e;
17        //插入s
18        s->next=p->next;
19        p->next=s;
20        return OK;
21    }
```

根据算法描述可以看到，插入算法的时间在查找第i-1个结点上，最快是在第1个位置前面插入（查找1次），最慢是表末插入（查找n 次），而插入查找只需要执行1次，因此时间复杂度为O(n)次。通过算法可知，单链表的插入算法不需要大量的移动操作，但是每次从头结点进行查找。

3.6 删除算法

根据单链表的特点，删除第i个元素，要找到第$i-1$个结点，然后建立第$i-1$个结点的后继为第$i+1$个结点。当找到了需要删除的数据时，直接使用前驱结点跳过要删除的结点指向要删除结点的后一个结点，再将原有的结点通过free函数释放掉。

算法思想：

Step1：依次查找第$i-1$个元素

Step2：查找失败，则返回，表示删除失败

Step3:从链表中删除节点（改变指针方向），在删除前需要保存待删除的结点；

Step4:释放结点

算法实现如下：

```
Status  delete(SqList&  L,int  i,ElemType& e){
    //L为带头结点的单链表的头结点
    p=L;
    j=0;     //j为计数器
    //遍历，沿指针后移,找第i-1个结点
    while(p->next&&j<i-1){
        p=p->next;
        ++j;
    }
    //删除位置不合理
    if(!p->next||j>i-1){
        return ERROR;
    }
    q=p->next;  //保存待删除的结点
    //删除q
    p->next=q->next;
    e=q->data;
    free(q);
    return OK;
}
```

3.7 建立单链表

在初始化之后，可以开始创建单链表了，单链表的创建分为头插入法和尾插入法两种，两者并无本质上的不同，都是利用指针指向下一个结点元素的方式进行逐个创建，只不过使用头插入法需要逆序建立，否则得到的序列是逆序的。

头插入法建单链表：从空表开始，生成新结点，并读取数据赋值给新结点的数据域中，然后将新结点插入到当前链表的表头，也就是头结点之后，反复做这样的操作就可以建立单链表。(这里以建立数据类型为整数,通过控制台输入，直到输入结束)

```
LinkedList createByHead(){
    Node *L;
    L = (Node *)malloc(sizeof(Node));   //申请头结点空间
    L->next = NULL;          //初始化一个空链表

    int x;       //x为链表数据域中的数据
    while(scanf("%d",&x) != EOF) {
        Node *p;
        p = (Node *)malloc(sizeof(Node)); //申请新的结点
        p->data = x;          //结点数据域赋值
        p->next = L->next;       //将结点插入到表头
        L->next = p;
    }
    return L;
}
```

尾部插入法：头插法建立单链表的算法虽然简单，但生成的链表中结点的次序和输入数据的顺序不一致。若希望两者次序一致，可采用尾插法。该方法是将新结点逐个插入到当前链表的表尾上，为此必须增加一个尾指针r, 使其始终指向当前链表的尾结点，否则就无法正确的表达链表。

```
LinkedList createByTail(){
    Node *L;
    L = (Node *)malloc(sizeof(Node));//申请头结点空间
    L->next = NULL; //初始化一个空链表
    Node *r;
    r = L;         //r始终指向终端结点，开始时指向头结点
    int x;       //x为链表数据域中的数据
    while(scanf("%d",&x) != EOF) {
        Node *p;
        p = (Node *)malloc(sizeof(Node));//申请新的结点
        p->data = x; //结点数据域赋值
        r->next = p; //将结点插入到表尾
```

```
13            r = p;
14        }
15        r->next = NULL;
16        return L;
17    }
```

4 双向链表

单链表可以实现很多的功能与操作，但是，单链表仍然存在不足，所谓‘单链表’，是指结点中只有一个指向其后继的指针，具有单向性，当找当前的结点的后继时间复杂度为$O(1)$，但是如果找当前节点的前驱，就必须要多次进行从头开始的遍历，时间复杂度为$O(n)$, 这样的搜索不是很便利。

对此在单链表的基础上，产生了双向链表的概念，即： 在单链表的基础上，对于每一个结点设计一个前驱结点，前驱结点与前一个结点相互连接，构成一个链表。

双向链表可以简称为双链表，是链表的一种，它的每个数据结点中都有两个指针，分别指向直接后继和直接前驱。所以，从双向链表中的任意一个结点开始，都可以很方便地访问它的前驱结点和后继结点。

其结点结构如下：

```
1  typedef struct DNode{
2      ElemType data;          //数据域
3      struct DNode *prior;//前驱
4      struct DNode *next;  //后继
5  }DNode,*DuLinkList;
```

由于增加了指向前驱的指针，则：$p \to next \to prior == p \to prior \to next == p$

双向链表查找某个结点的后继的时间复杂度为$O(1)$，查找其前驱的时间复杂度也为$O(1)$。

在p结点前面插入一个结点s，执行如下操作：

$$s \to prior = p \to prior;$$

$$p \to prior \to next = s;$$

$$s \to next = p$$

$$p \to prior = s;$$

删除结点p，执行如下操作：

$$p \to prior \to next = p \to next;$$

$$p \to next \to prior = p \to prior;$$

5 循环链表

对于单链表以及双向链表，其就像一个小巷，无论怎么样最终都能从一端走到另一端，然而循环链表则像一个有传送门的小巷，因为循环链表当你以为你走到结尾的时候，其实你又回到了开头。

循环链表和非循环链表其实创建的过程以及思路几乎完全一样，唯一不同的是，非循环链表的尾结点指向空（NULL），而循环链表的尾指针指向的是链表的开头。通过将单链表的尾结点指向头结点的链表称之为循环单链表（Circular linkedlist）。

循环链表空表的条件是：$L \to next == H$。

6 双向循环链表

将双向链表的头结点的前驱指针指向最后一个结点，最后一个结点的后继指针指向头结点，这样就构成了双向循环链表。操作和双向链表及循环链表类似，既具备双向链表的特点，又具备循环链表的特点。

空的双向循环链表的条件是：

$$H \to next == H \ or \ H \to prior == H$$

7 应用

应用举例：利用线性表实现一元多项式的加法和乘法。

多项式$P_n(x)$按升幂可写成：

$$P_n(x) = P_0 + P_1 * x + p_2 * x^2 + ... + P_n * x^n$$

它由n+1个系数唯一确定，因此可用一个线性表$p=(p_0,p_1,p_2,...,p_n)$,每一项的指数i隐含在系数p_i的序号里。设$Q_m(x)$ 是一元m次多项式，则可用线性表$Q=(q_0,q_1,q_2,...,q_m)$来表示。不失一般性，设$m<n$，则两个多项式相加的结果多项式$R_n(x)=P_n(x)+Q_m(x)$可用线性表：

$$(p_0+q_0,p_1+q_1,...,p_m+q_m,p_{m+1},,p_n)$$

分析：

对P、Q、R可用顺序存储结构，多项式相加的算法定义将十分简单，但通常应用中次数可能很高，使得顺序存储结构的最大长度很难确定；另外很多项的系数也可能为0，比如$P(x)=1+3x^{10000}+2x^{20000}$,如果是像这样仅有3个非0元素，则顺序存储结构的线性表的存储空间将非常浪费，故采用单链表结构，但是存储指数。

一元n次多项式$Pn(x)=p_1*x^{e_1}+p_2*x^{e_2}+...+p_m*x^{e_m}$,其中$p_i$是指数为$e_i$的项的非零系数，且满足$0<=e_1<e_2<<e_m=n$, 若用一个长度为m且每个元素有两个数据项（系数项+指数项）的线性表$((p_1,e_1),(p_2,e_2),...,(p_m,e_m))$便可唯一确定多项式$P_n(x)$。最坏情况下，$n+1(=m)$个系数都不为0，这种表示将大大节省空间。

实现：

”和多项式”链表中的节点无需另生成，而应该从两个多项式的链表中获取。运算规则如下：假设指针qa、qb分别指向多项式P 和Q中当前进行比较的某个节点，则比较两个节点中的指数项，有下列3种情况：

情形1.qa所指节点的指数值小于qb所指节点的指数值，则应摘取qa所指节点插入到“和多项式”中；

情形2.qa所指节点的指数值大于qb所指节点的指数值，则应摘取qb所指节点插入到“和多项式”中；

情形3.qa所指节点的指数值等于qb所指节点的指数值，则将两节点的系数相加

3.1若系数相加和数不为0，则修改qa所指节点的系数值，同时释放qb所指节点；

3.2若系数相加和数为0，从多项式P的单链表中删除相应节点，并释放qa和qb所指节点。

注意：表示一元多项式的应该是有序链表。

结点结构如下，分成三个域，一个存放系数、一个存放指数，另外一

个是指针域，指向下一个结点。

```
//一元多项式的存储结构
Typedef struct PNode{
      float coef;//系数
       int expn;//指数
      struct Lnode *next;   //指针域
}PNode, *PolyLinkList;
```

```
//两个一元多项式的求和
Status  AddPloy(PolyLinkList& pl1, PolyLinkList& pl2){
      p=pl1->next;
      q=pl2->next;
      s=pl1;//用s保存为p的前驱
      while(p&&q){
          if(p->expn<q->expn){
              s->next=p;
              s=p;
              p=p->next;
          }else if(p->expn>q->expn){
               s->next=q;
               s=q;
               q=q->next;
          }
          else{   //指数相等，系数求和
                 p->coef=p->coef+q->coef;
                 if(p->coef==0){
                        t1=p;
                        t2=q;
                        p=p->next;
                        q=q->next;
                        free(t1);
```

```
24                          free(t2);
25                      }
26                  }
27              }
28              if(p){
29                  s->next=p;
30              }
31              if(q){
32                  s->next=q;
33              }
34  }
```

算法分析：利用链表实现，相当于两个有序表的合并，因此需要运行$m+n$次，时间复杂度为O(m+n)。

8 小结

本章介绍了线性表这种简单常用的线性结构，主要内容包括：

- 线性表的定义与性质：一对一的关系
- 线性表的顺序存储结构与链式存储结构
- 顺序表的抽象数据类型定义与操作
  顺序表的插入和删除操作，算法复杂度为O(n)，需要移动元素，取第i个元素的操作为O(1)
- 链表的抽象数据类型定义与操作
  链表的插入和删除操作，算法复杂度为O(n)，不需要移动元素，查找第i元素的操作为O(n)
- 循环链表
- 双向链表

9 Next

介绍另外两种线性结构－－－－栈和队列，又称为操作受限的线性表。主要内容如下：

(1)栈的表示与实现

(2)队列的表示与实现

(3)栈和队列的应用举例。

10 习题

（一）填空题

1. 带头结点L的单链表为空表的条件是：_____

2. 删除结点s的后继的语句是_____、_____、_____。

3.顺序表的插入算法时间复杂度为_____、单链表的插入算法时间复杂度为_____

4. 带头结点循环单链表，为空表的条件是_____

5.如果线性表的操作主要是取元素和查找等操作，最好选择_____存储结构。

6.顺序表插入算法的平均移动次数是_____，删除算法的平均移动次数是_____，时间复杂度都是_____。

7.链表的插入和删除算法主要在_____操作上。

8.顺序表是通过_____来反映逻辑相邻，链表是通过_____来反映相邻。

（二）选择题

1. 线性表的顺序存储结构地址是（ ）,链式存储结构地址是（ ）。

A.连续的、不连续的； B.连续的、连续的； C.连续的，可连续可不连续； D.任意的、任意的

2.关于顺序表和链表的插入算法，下面表述正确的是（ ）

A.顺序表和链表都需要移动元素； B.顺序表需要移动，链表不需要；C.顺序表的时间复杂度比链表的差； D.链表需要移动元素，顺序表不要移动

3.在p结点后插入一个结点s的语句是（ ）

A.$p \rightarrow next = s; s \rightarrow next = p \rightarrow next$; B.$s \rightarrow next = p; p \rightarrow next = s$;

C.$s \to next = p \to next; p \to next = s \to next;$ D.$s \to next = p \to next; p \to next = s;$

4. 双向链表的插入和删除算法的时间复杂度为：（ ）

A. O(1),O(1) B.O(n)、O(n)

C.O(n),O(1) D.O(1),O(n)

5. 双向链表在p结点之前插入一个结点s的语句序列是（ ）

A.$p \to next = s; s \to prior = p \to prior;$

B.$p \to prior = s; s \to next = p; s \to prior = p \to prior;$

C.$p \to prior \to next = s; s \to prior = p \to prior; s \to next = p; p \to prior = s;$

D.$s \to next = p; p \to prior = s; s \to prior = p \to prior; p \to prior \to next = s;$

（三）算法题

1. 已有La、Lb两个链表，每个链表中的结点包括学号、成绩。要求把两个链表合并，按学号升序排列。

输入：

La的数据：学号 成绩（按空格隔开），如：

18005 98

18001 69

....

Lb的数据；学号 成绩(按空格隔开)

18006 98

18007 80

18009 80

输出：新的链表按照学号升序排列的数据，并输入学号和成绩。

2. 编程实现两个一元多项式的加法和乘法。

11 附录

11.1 顺序表的实现代码

```
/****predefine.h*********/
#pragma once
#include <iostream>

#define OK 1
#define ERROR 0
#define MAX 100
#define INCREMENT 10

typedef int Status;

using namespace std;
```

```
/****SqList.h*********/
#pragma once
#include "predefine.h"
typedef int ElemType;
class SqList
{
public:
        SqList();
        SqList(int elem[], int n);
        ~SqList();

        Status insert(ElemType e, int pos);//在pos前面插入一个e
        Status del(ElemType& e, int pos);    //删除第pos个元素，并且用e返回
        Status find(ElemType e);//查找第一次出现的e的位置
```

```
15          Status get(ElemType& e, int pos);//取第pos个元素
16          void print();  //打印输出里面的元素
17          Status clear();//清空
18          int getLength();//取长度
19  private:
20          ElemType* data;//数据数组
21          int length;//顺序表的长度
22          int listSize;//当前的容量
23  };
```

```
1   /****SqList.cpp*********/
2   #include "SqList.h"
3
4   SqList::SqList()
5   {
6           data=(ElemType*)malloc(sizeof(ElemType)*MAX);
7           if (!data)
8           {
9                   exit(OVERFLOW);
10          }
11          length = 0;
12          listSize = MAX;
13  }
14  SqList::SqList(int elem[], int n)
15  {
16          data = (ElemType*)malloc(sizeof(ElemType)*n);
17          if (!data)
18          {
19                  exit(OVERFLOW);
20          }
21          length = n;
22          listSize = n;
23          //赋值
```

```
24          for (int i = 0; i < n; i++)
25          {
26                  data[i] = elem[i];
27          }
28  }
29  SqList::~SqList()
30  {
31          free(data);//delete[] data;
32  }
33
34  Status SqList::insert(ElemType e, int pos)
35  {
36          //判断插入位置是否合法
37          if (pos<1 || pos>length + 1)
38          {
39                  return ERROR;
40          }
41          //判断空间是否为满
42          if (length >= listSize)
43          {
44                  ElemType* newbase = (ElemType*)realloc
                        (data, (listSize + INCREMENT) *
                        sizeof(ElemType));
45                  if (!newbase)
46                  {
47                          exit(OVERFLOW);
48                  }
49                  data = newbase;
50          }
51
52          //插入
53          ElemType* q = &data[pos - 1];
54          for (ElemType* p = &data[length - 1]; p >= q;
```

```
            p--)
        {
                *(p + 1) = *p;
        }
        //插入元素
        *q = e;
        /*和上面代码一样
        for (int i = length - 1; i >= pos - 1; i--)
        {
                data[i + 1] = data[i];
        }
        data[pos - 1] = e;
        */
        //表长增1
        length++;
        return OK;
}
Status SqList::del(ElemType& e, int pos)
{
        //判断删除位置是否合法
        if (pos<1 || pos>length)
        {
                return ERROR;
        }
        //保存删除的元素
        e = data[pos - 1];
        //移动元素
        for (int i = pos; i < length; i++)
        {
                data[i - 1] = data[i];
        }
        //表长减1
        length--;
```

```
87          return OK;
88  }
89  Status SqList::find(ElemType e)
90  {
91          for (int i = 0; i < length; i++)
92          {
93                  if (data[i] == e)
94                  {
95                          return i;
96                  }
97          }
98          return ERROR;
99  }
100 Status SqList::get(ElemType& e, int pos)
101 {
102         //判断位置是否合法
103         if (pos<1 || pos>length)
104         {
105                 return ERROR;
106         }
107         e = data[pos];
108         return OK;
109 }
110 void SqList::print()
111 {
112         /*输出每个元素*/
113
114         //10,20,30
115         if (length > 0)
116         {
117                 cout << "当前线性表的长度为:
                        " << length << ", 元素为: ";
118                 cout << data[0];
119                 for (int i = 1; i < length; i++)
```

```
120                 {
121                         cout << "," << data[i];
122                 }
123                 cout << endl;
124         }
125         else
126         {
127                 cout << "当前线性表为空! " << endl;
128         }
129 }
130 Status SqList::clear()
131 {
132         length = 0;
133         return OK;
134 }
135 int SqList::getLength()
136 {
137         return length;
138 }
```

```
1  /****Test.cpp*********/
2  #include "SqList.h"
3
4  int main()
5  {
6          int a[5] = { 10,0,5,20,30 };
7          SqList lst(a,5);
8          lst.print();
9
10         int e, pos;
11         while (true)
12         {
13                 cout << "插入(0) or 删除(1) or 退出（其
```

```
    它)：";
14          char ch;
15          cin >> ch;
16          if (ch == '0')
17          {
18                  cout << "请输入插入位置和元素：";
19                  cin >> pos >> e;
20                  Status flag=lst.insert(e, pos)
                        ;
21                  if (flag)
22                  {
23                          cout << "插入成功! 插入
                            后:" ;
24                          lst.print();
25                  }
26                  else
27                  {
28                          cout << "插入失败，可能是
                            位置不合法!
                            " << endl;
29                  }
30          }
31          else if (ch == '1')
32          {
33                  cout << "请输入删除位置：";
34                  cin >> pos;
35                  Status flag = lst.del(e, pos);
36                  if (flag)
37                  {
38                          cout << "删除成功，删除元
                            素是:"<<e<<"删除后:
                            ";
39                          lst.print();
40                  }
41                  else
42                  {
```

```
                                        cout << "删除失败，请检查
                                            删除位置! " << endl;
                        }
                }
                else
                {
                        break;
                }
        }
        system("pause");
        return 0;
}
```

11.2 链表的实现代码

```
/****LinkList.h**********/
#pragma once
#include "predefine.h"
typedef int DataType;
//结点类型
class Node
{
public:
        DataType data;//数据域
        Node *next;//指针域
};
class LinkList
{
public:
        LinkList();
        LinkList(DataType a[], int n);
        ~LinkList();

```

```
public:
        Status create(int size);  //初始化链表
        Status get(DataType &e,int pos); //取元素
        Status insert(DataType e, int pos);//插入操作
        Status del(DataType &e, int pos);//删除操作
        int getLength();
        bool isEmpty();
        void print();
        Status find(DataType e);
        Status clear();//清空
private:
        Node* head;//头结点
        int size;//链表长度
};
```

```
/****LinkList.cpp*********/
#include "LinkList.h"
LinkList::LinkList()
{
    head = new Node;
    head->next = nullptr;
    size = 0;
}
LinkList::~LinkList()
{
        Node *ptr;
        while (head != NULL)
        {
                ptr = head;
                head = head->next;
                delete ptr;
        }
}
```

```
19 Status LinkList::create(int size)  //初始化链表：顺序
20 {
21         //判断size是否符合
22         if (size < 0)
23         {
24                 cout << "Create Error..." << endl;
25                 return ERROR;
26         }
27         Node *pNew = nullptr;
28         Node * pTemp = head;
29         for (int i = 0; i < size; i++)
30         {
31                 pNew = new Node;
32                 pNew->next = nullptr;//因为我们需要把它查
                       到末尾
33                 cout << "输入第" << i + 1 << "个结点的值：
                       " ;
34                 cin >> pNew->data;
35                 pTemp->next = pNew;
36                 //保留下一次插入的位置
37                 pTemp = pNew;
38         }
39         //别忘了：
40         this->size = size;
41         cout << "创建完毕! " << endl;
42         return OK;
43 }
44 Status LinkList::get(DataType &e, int pos)//取元素
45 {
46         //判断条件
47         if (pos<1 || pos>size)
48         {
49                 return ERROR;
50         }
```

```
51          //遍历去找pos
52          Node* ptr = head;
53          for (int i = 1; i <= pos; i++)
54          {
55                  ptr = ptr->next;
56          }
57          e = ptr->data;
58          return OK;
59  }
60  Status LinkList::insert(DataType e, int pos)//插入操作
61  {
62          //判断条件
63          if (pos<1 || pos>size + 1)
64          {
65                  return ERROR;
66          }
67          Node *pNew = new Node;
68          if (pNew == NULL)
69          {
70                  cout << "Error create Node..." << endl
                        ;
71                  exit(OVERFLOW);
72          }
73          pNew->data = e;
74          Node *ptr = head;
75          //遍历去找pos
76          for (int i = 1; i < pos; i++)
77          {
78                  ptr = ptr->next;
79          }
80          pNew->next = ptr->next;
81          ptr->next = pNew;
82          //别忘了:
```

```
83          size++;
84          return OK;
85  }
86  Status LinkList::del(DataType &e, int pos)//删除操作
87  {
88          //判断条件
89          if (pos<1 || pos>size)
90          {
91                  return ERROR;
92          }
93          Node *ptr = head;
94          //遍历去找pos
95          for (int i = 1; i < pos; i++)
96          {
97                  ptr = ptr->next;
98          }
99          //删除
100         Node *q = ptr->next;
101         ptr->next = ptr->next->next;
102         e = q->data;
103         delete q;
104         //别忘了
105         size--;
106         return OK;
107 }
108 int LinkList::getLength()
109 {
110         return size;
111 }
112 bool LinkList::isEmpty()
113 {
114         if (size == 0)
115         {
```

```
116                     return true;
117             }
118             return false;
119     }
120     void LinkList::print()
121     {
122             if (size > 0)
123             {
124                     cout << "链表长度为: " << size << "元素为:
                            ,";
125                     Node* ptr = head->next;
126                     cout << ptr->data;
127                     ptr = ptr->next;
128                     while (ptr != NULL)
129                     {
130                             cout << "," << ptr->data;
131                             ptr = ptr->next;
132                     }
133                     cout << endl;
134             }
135             else
136             {
137                     cout << "当前为空表! " << endl;
138             }
139     }
140     Status LinkList::find(DataType e)
141     {
142         Node* p=head->next;
143         int i=1;
144         while(!p)
145         {
146             //相等，返回其位置
147             if(p->data==e)
148             {
```

```
149                 return i;
150             }
151         }
152             return ERROR;
153 }
154 Status LinkList::clear()
155 {
156             //释放存储空间
157             Node* ptr = head->next;
158             while (ptr != NULL)
159             {
160                     Node *q = ptr;
161                     ptr = ptr->next;
162                     delete q;
163             }
164             size = 0;
165             return OK;
166 }
```

```
1  /****LinkListTest.cpp*********/
2  #include "LinkList.h"
3  int main()
4  {
5      //构建一个链表对象
6      LinkList lst;
7      int n;
8      cout << "输入创建链表的大小：";
9      cin >> n;
10     lst.create(n);
11     lst.print();
12     int e, pos;
13     while (true)
14     {
```

```
15          cout << "插入(0) or 删除(1) or 退出其它(): ";
16          char ch;
17          cin >> ch;
18          if (ch == '0')
19          {
20                  cout << "请输入插入位置和元素: ";
21                  cin >> pos >> e;
22                  Status flag = lst.insert(e, pos);
23                  if (flag)
24                  {
25                          cout << "插入成功! 插入后:";
26                          lst.print();
27                  }
28                  else
29                  {
30                          cout << "插入失败, 可能是位置不合
                               法! " << endl;
31                  }
32          }
33          else if (ch == '1')
34          {
35                  cout << "请输入删除位置: ";
36                  cin >> pos;
37                  Status flag = lst.del(e, pos);
38                  if (flag)
39                  {
40                          cout << "删除成功, 删除的元素
                               是" << e << "删除后: ,";
41                          lst.print();
42                  }
43                  else
44                  {
45                          cout << "删除失败, 请检查删除位置!
                               " << endl;
46                  }
```

```
47          }
48          else
49          {
50                  break;
51          }
52      }
53      system("pause");
54      return 0;
55  }
```