第 2 章 线性表

线性表是软件设计中最常用且最简单的一种数据结构，是《数据结构》课程中所介绍的第一种数据结构。本章是整个课程的基础，首先给出线性表的定义及相关概念和基本运算；然后分别讨论采用顺序存储方式和链式存储方式存储线性表、实现前面所提出的基本运算的算法并对其进行性能比较，并通过实例说明了线性表的应用；最后介绍了串这一种特殊的线性表结构。

2.1 线性表的定义和运算

日常生活中，线性表的例子比比皆是，例如，英文 26 个字母表；在编写输出年历的程序时，需要用到一年中每个月的天数表；8086 系列中的中断向量表；某班级的学生成绩表等。所有这些表面上不完全相同的表都可抽象为本章所要介绍的线性表，其中各元素是依次排列的，这是最为常用的一类数据结构。

2.1.1 线性表的定义

定义：线性表 L 是 n 个元素 e_1，e_2，……，e_n 组成的有限序列，记作 L=(e_1,e_2,…,e_n)，其中 n.>=0 为**表长度**；当 n=0 时为**空表**，记作 L=（）。

对线性表中某个元素 e_i 来说，称其前面的元素 e_{i-1} 为 e_i 的直接前驱，称其后前面的元素 e_{i+1} 为 e_i 的直接后继。

对非空线性表，具有下列**特点**：

1) 有一个唯一的“第一个（首）”数据元素；
2) 有一个唯一的“最后一个（尾）”数据元素；
3) 除头元素外，表中其它元素有且仅有一个直接前驱；
4) 除尾元素外，表中其它元素有且仅有一个直接后继。

由前述可知，线性表是许多实际应用领域中表结构的抽象形式，因此，线性表中元素在不同的场合可以有不同的**含义**。例如，在字母表（A,B,C,……,Z）中，每个元素是一个字母；在一个学生信息表中，每个数据元素是一个学生的基本信息，其中可能包含学号、姓名、性别、专业、年级等数据项构成。但要注意，**在同一个表中的各元素的类型是一致的**。

2.1.2 线性表的运算

如前所述，线性表结构是许多实际应用中所用到的表结构的抽象，因此对线性表的实际运算可以有很多，例如对我们所熟知的工资表和成绩表就有很多运算的要求。为了便于研究，一般不会也不可能讨论所有的运算，取而代之的是只讨论其基本运算。在此基础上，可以较

方便地实现更多的复杂运算。

线性表常用的六个基本运算：

（1）初始化线性表：initialList (L)

建立线性表的初始结构，即建空表。这也是各种结构都可能要用的运算。

（2）求表长度：listLength(L)

即求表中的元素个数。

（3）按序号取元素：listGetElement(L,i)

取出表中序号为 i 的元素，i 的有效范围是 1<=i<=n。

（4）按值查询：listLocate(L,x)

在线性表 L 中查找指定值为 x 的元素，若存在该元素，则返回其地址；否则，返回一个能标记其不存在的地址值或标记。

（5）插入元素：listInsert(L,i,x)

在表 L 的第 i 个位置上插入值为 x 的元素。显然，若表中的元素个数为 n，则插入序号 i 应满足 1<=i<=n+1。

（6）删除元素：listDelete（L，i）

删除表 L 中序号为 i 的元素，显然，待删除元素的序号应满足 1<=i<=n。

虽然只给出了这六个基本运算，但借助于这些基本运算可以构造出其他更为复杂的运算。例如，如果要求删除线性表中值为 x 的元素，则可用上述运算中的两个运算来实现：先引用 ListLocate 求出元素的位置，再引用 ListDelete 来实现删除。尽管这一实现的时间性能不太好，但在讨论基本运算时，主要还是侧重于其逻辑上的实现，而不是具体程序上的实现。

另外，在特定的存储结构上实现基本运算时，可能会有一定的差异。

2.2 线性表的顺序存储结构—顺序表

2.2.1 顺序存储结构

为在计算机上实现数据结构，首先需要将数据结构存储到计算机中。对此可有许多不同的方法，其中最为简单的方法是**顺序存储方式**：假设有一个足够大的连续的存储空间，则可将表中元素按照其逻辑次序依次存储到这一存储区中，称由此而得到的线性表为**顺序表**。

在具体实现时，一般用高级语言中的数组来对应连续的存储空间。假设最多可存放 MAXLEN 个元素，则可用数组 data[MAXLEN]来存储表中元素。另外，由于线性表有插入和删除元素这类改变表中元素个数的运算，因此，为随时了解线性表当前的元素个数（即表长度），需要另外设置一个变量以记录其元素个数，此处不妨用 listLen 来记录元素个数。这样，一个线性表的顺序存储结构就有两个分量，将这两个分量合在一起构成了一个整体。如图 2-1 所示：

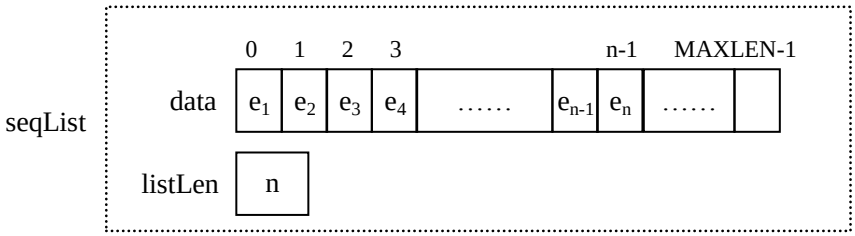

图 2-1 顺序存储结构示意图

对所设计的存储形式，需要让计算机能识别和处理，这就是数据结构的存储描述。在用 C 语言描述这一由两个分量组成的结构时，需要采用其所提供的结构类型（struct），在说明时，要给该类型起个名，在此不妨用 seqList。另外，由于用到的数组事先要规定最大的元素个数，因此，为通用起见而要先给出一个常量形式的最大长度；由于元素的类型也取决于具体应用的问题，故此处的元素类型用 elementType 来表示。

由上述分析可得顺序表类型的描述如下：

```
#define    MAXLEN 100          //不妨假设元素个数最大为 100
struct    sList
{
    elementType    data[MAXLEN];      //定义存储表中元素的数组
    int            listLen;           //定义表长度分量
};
typedef    struct sList    seqList;        //用 typedef 将 seqList 定义为顺序表类型
```

或者将上面两个部分集成描述为：

```
typedef    struct    sList
{
    elementType    data[MAXLEN];      //定义存储表中元素的数组
    int            listLen;           //定义表长度分量
} seqList;
```

由存储方式可知**顺序表的特点：逻辑上相邻的元素的存储地址也相邻，即逻辑上相邻的元素物理存储位置也相邻**。

实际使用时 elementType 可用 typedef 指定为需要的类型，比如需要元素类型为 int 型，可以使用 typedef int elementType 语句指定，这样 elementType 即为 int 类型。

在描述算法及上机实现时，需要对这种结构类型的具体变量进行运算，这样就涉及到表结构变量的定义，其格式如下：

```
seqList    L，*L1；
```

按 C 语言语法规定，引用顺序表结构变量的分量时，需要使用“.”符号，其含义相当于汉语中的“的”，其读音也相近。比如，上面的定义中，L 为顺序表结构变量，其分量的引用形式为 L.listLen 和 L.data[i]。如果变量为顺序表结构的指针变量，则需要使用“->”符号来引用结构的分量，如上面的 L1，分量的引用形式为 L1->listLen 和 L1->data[i]。

【注意】本书中线性表的元素编号都是从 1 开始，而数组的下标从 0 开始，两者差 1，请读者在阅读代码和上机实现时注意。

2.2.2 顺序表运算的实现

在线性表采用顺序表结构存储时，如何实现所给出的六个基本运算呢？各算法的性能又如何呢？下面对此展开讨论。

1．线性表初始化运算的实现

由顺序表结构的定义可知，其需要的连续存储空间在代码编译时已经确定，我们这里的初始化只是将表设置为空表，即表的 listLen=0。

【算法描述】

```
void initialList（seqList *L）
{    L->listLen=0;      }
```

【算法分析】算法的时间复杂度为 O(1)。

【思考问题】这个函数的参数 L，为什么要使用指针类型呢？有没有其他方式从子函数往主调函数传递初始化后的顺序表呢？

2．求表长度函数的实现

这一功能的实现也较简单，只需返回表 L 的分量 L.listLen 即可求出顺序表 L 的长度值。

【算法描述】

```
int   listLength（seqlist L）
{      return   L.listLen ;    }
```

【算法分析】算法的时间复杂度为 O(1)。

【思考问题】这个函数的参数为什么可以不用指针？使用指针行不行？还有其他方式往主调函数传递表长度值吗？

3．按序号求元素运算的实现

按前面的约定，顺序表 L 中序号为 i 的元素（即 e_i）存放在下标为 i-1 的数组单元中，因此，直接从该数组单元中取值即可。为使该函数有较大的应用范围，可用参数的形式返回所给定的元素值，因此，算法的实现形式与所给定的问题的运算要求略有差异。另外需要注意的是，作为一个算法，不仅要考虑参数正确情况下的求解，同时也要考虑参数不正确时的处理。就本题来说，需要判断元素的序号是否在合法的范围内，序号的正确范围是 1<=i<=n。

【算法描述】

```
void   getElement（seqlist L，int i, elementType *x）
{
      if（i<1 || I>L.listLen）  error(“超出范围”);
      else     *x= L.data[i-1];
}
```

【算法的一种实现】

```
//*********************************************//
//* 函数功能：按给定序号，取出表中元素          *//
//* 输入参数：seqList L，为当前顺序表           *//
//*           int i，指定元素序号               *//
```

```
//* 输出参数：elementType* x，返回获取的元素  *//
//* 返 回 值：int，0：取元素失败，1 成功       *//
//*******************************************//
int   getElement(seqList L, int i, elementType* x)
{
     if(i<1 || i>L.listLen)
          return 0;
     else
     {
          (*x)=L.data[i-1];
          return 1;
     }
}
```

【算法分析】算法时间复杂度为 O(1)。

【思考问题】本算法使用指针返回求得的目标元素，还有其他返回方法吗？

4．查找运算的实现

要确定值为 x 的元素在 L 表中的位置，需要依次比较各元素。因而需要用循环语句搜索。当搜索到该元素时，就返回 x 在 L 中的序号，否则返回 0。

【算法描述】

```
int   listLocate（seqlist L，elementType x）
{      int   i;
       for (i=0; i<L.listLen; i++)
            if (L.data[i]==x)   return   i+1;
       return 0;
}
```

【算法的一种实现】

```
//*************************************************//
//* 函数功能：给定元素 x，获取其在表 L 中的序号     *//
//* 输入参数：seqList L，为当前顺序表               *//
//*           elementType x，给定的元素             *//
//* 输出参数：无                                    *//
//* 返 回 值：元素序号，0 表示元素不在表中          *//
//*************************************************//
int listLocate(seqList L, elementType x)
{
       int i;
       for(i=0; i<L.listLen; i++)
            if(L.data[i]==x)
                 return i+1;   //元素 x 在 L 中，在此返回。
       return 0;    //循环结束，x 不在 L 中，在此返回。
```

}

【算法分析】算法的基本操作为 for 循环中的比较操作，所以只要统计出平均比较次数，即可得出算法的时间复杂度。查找成功，即 x 在表 L 中，假定 x 在 L 中的序号为 i，需要比较 i 次，而 1<=i<=n。假设 x 在 L 各个位置出现的概率相同，即出现概率平均为 1/n，成功查找的平均比较次数为 $\sum_{i=1}^{n} i \times \frac{1}{n} = \frac{n+1}{2}$。查找失败，即 x 不在表 L 中，比较次数为 n。所以，综合起来，算法的时间复杂度为 O(n)。

【思考问题】本算法使用函数返回值返回 x 在 L 中的序号，还有其他方法传回元素序号吗？

5．插入算法的实现

在顺序表 L 的第 i 个元素位置，即 e_i 位置插入一个元素 x 时，如果能插入，需要依次执行下列操作：

（1） 将 e_i～e_n 往后依次移一“格”；

（2） 将 x 插入到第 i 个位置上，实现插入；

（3） 修改表的长度，将长度 listLen 加 1，因为表长度是顺序表不可分割的一个分量。

在编程实现其中的批量移动元素这一操作时，需要注意移动的次序：初学者可能会考虑到两种不同的实现方法，其一是从前往后，也就是先将 e_i 的值送到 e_{i+1} 中，再将 e_{i+1} 的值送到 e_{i+2} 中，……，直到 e_n 也往后移一格为止。这种方法显然耗时费力。正确的移动只能是从后往前进行，即将 e_n 后移，以空出位置，再将 e_{n-1} 后移，……，e_i 后移。这种批量移动是软件设计中常见的基本操作。

实现批量移动的控制可用 for 循环语句来实现，若用 j **指示待移走元素的数组下标**，则第一个要移走的元素 e_n 在数组中的下标是 n-1（listLen-1），而最后一个要移走的元素 e_i 在数组中的下标是 i-1。移动元素的循环控制部分可为 for (j=n-1; j>=i-1; j--)。

下面再简要讨论插入的条件：在实际应用中，算法可能要被多处引用，某些引用未必能满足所需的条件，这就需要我们在算法中来检测相关的条件，以免造成不必要的错误。显然，如果表空间已经满了，则不能插入。另外，按定义，插入序号必须满足 1<=i<=n+1。

【算法描述】

```
void   listInsert（seqList *L，elementType x，int i）
{    int j;
     if （L->listLen==MAXLEN）error（“overflow”）;            //溢出，不能插入
     else if （i<1 || i>L->listLen+1） error（“position error”）;  //插入范围错，并结束
     else
     {      for（j=L->listLen-1;  j>=i-1;  j--）          //往后移动元素
                  L->data[j+1]=L->data[j];                 //注意元素编号与数组下标差 1。
            L->data[i-1]=x;          //填入插入内容
            L->listLen++;            //修改表长
     }
}
```

【算法的一种实现】

```
//***********************************************//
//* 函数功能：在给定位置，插入给定元素              *//
//* 输入参数：seqList* L，为当前顺序表指针         *//
//*           elementType x，给定的插入元素        *//
//*           int i，给定的插入位置                *//
//* 输出参数：seqList* L，为当前顺序表指针         *//
//* 返 回 值：整型数，0：表满；1：插入位置非法；    *//
//*           2：插入成功                          *//
//***********************************************//
int listInsert(seqList *L, elementType x, int i)
{
    int j;
    if(L->listLen==MAXLEN)
        return 0;  //表满，返回 0
    else if(i<1 || i>L->listLen+1)
        return 1;  //序号超出范围，返回 1
    else
    {
        for(j=L->listLen-1; j>=i-1; j--)    //循环后移表元素
            L->data[j+1]=L->data[j];    //注意元素序号和数组下标差 1。
        L->data[i-1]=x;    //插入元素 x
        L->listLen++;       //表长度增 1
        return 2;           //成功插入，返回值 2
    }
}
```

【算法分析】由算法可知，该算法花费时间最多的操作是移动元素，移动元素的次数取决于序号 i。对 i 分别取值为 1，2，…和 n+1 时，移动次数分别为 n，n-1,……1，0。为便于讨论，通常是求出插入一个元素时的平均移动次数。虽然在各个位置上插入元素的概率不尽相同，但为了对算法的时间性能有大致的了解，我们假设各位置的插入概率相同，为 1/(n+1)，因此可求出其在插入一个元素时平均移动元素的次数如下：

E_{is}= （0+1+…+n）/(n+1) = n /2

所以，算法的时间复杂度为 O(n)。

【思考问题】本算法使用指针往主调函数回传插入后的顺序表，还有其他回传方式吗？

6．删除算法的实现

如果能从顺序表 L 中删除第 i 个元素的话，则可通过完成如下两个操作来实现：

（1）将 e_{i+1}～e_n 依次前移，从而将待删除元素 e_i“覆盖掉”，因为顺序表要求逻辑上相邻的元素在物理上也相邻。

（2）表长度减 1。

和插入操作类似的是，在批量前移元素时，同样要注意移动的次序。显然，此处不能象插入算法中那样从后往前进行。最好的做法是先前移 e_{i+1}，接着前移 e_{i+2}，…，最后前移 e_n。

关于删除运算能执行的条件：待删除元素应该存在，也就是说，表中不仅要有元素，并且待删除的第 i 个元素也存在，合法的 i 范围应为 1<=i<=n。

用 for 循环语句来实现批量元素前移，用 j **指示待移走元素的数组下标**，则第一个要移走的元素 e_{i+1} 在数组中的下标是 i，而最后一个要移走的元素 e_n 在数组中的下标是 n-1（listLen-1）。移动元素的循环控制部分为 for (j=i; j<=n-1; j++)。

【算法描述】

```
void   listDelete（seqlist *L，int i）
{    int j;
     if   (L->listLen<=0) error (“下溢出错”);          //空表不能删除元素
     if   (i<1 || i>L->listLen) error (“删除位置错”);   //序号错误，删除元素不存在
     else
     {    for （j=i；j<=L->listLen-1；j++）       //向前批量移动元素
               L->data[j-1]=L->data[j];
      L->listLen--;           //表长度减 1
}
```

【算法的一种实现】

```
//***********************************************//
//* 函数功能：删除表中指定位置的元素                *//
//* 输入参数：seqList* L，为当前顺序表指针          *//
//*           int i，给定的删除位置                 *//
//* 输出参数：seqList* L，为当前顺序表指针          *//
//* 返 回 值：整型数，0：表满；1：删除位置非法      *//
//*           2：删除成功                           *//
//***********************************************//
int listDelete(seqList *L, int i )
{
    int j;
    if(L->listLen<=0)
        return 0; //空表，返回值。
    else if(i<1 || i>L->listLen)
        return 1;   //删除的序号不在有效范围内，返回值 1.
    else
    {
        for(j=i; j<L->listLen; j++ )       //循环前移表元素。
            L->data[j-1]=L->data[j];     //元素编号与数组下标差 1。
        L->listLen--;   //修改表长度。
        return 2;          //成功删除，返回值 2。
    }
}
```

【算法分析】和插入算法类似的是，删除算法花费时间最多的操作也是移动元素，移动元素的次数同样取决于序号 i，为此，同样也要计算出移动一个元素的平均移动次数。由算

法可知，在 i 依次取值为 1，2，…和 n 时，分别要移动 n-1，…，1，0 次。同样假设各元素的删除概率相同，则删除一个元素时平均移动元素的次数为：

E_{is}= （0+1+…+(n-1)）/ n =(n-1) /2

可见，算法的时间复杂度为 O(n)。

【思考问题】

（1）算法中循环控制变量 j 的含义是____：

A. 要移走的线性表元素的序号；　　B. 要移走的数组元素的下标；

C. 将要移到的线性表元素的序号；　D. 将要移到到的数组元素的下标；

（2）本算法使用指针传回删除元素后的顺序表，还有其他方式传回吗？

2.2.3 顺序表的应用

前面讨论了顺序表结构及其基本运算的实现，虽然实际应用中许多问题的结构和运算比这些要复杂，但线性表中所讨论的基本运算及其实现方法有助于复杂问题的求解，下面给出几个求解实例。

【例 2.1】现有 2 个集合 A 和 B，求一个新集合 C=A∪B，3 个集合都用顺序表表示。比如 A={2, 4, 10, 5, 9}，B={1, 4, 6, 8}，则 C={2, 4, 10, 5, 9, 1, 6, 8}

【算法思想】这是较简单的顺序表合并问题，要注意的是，因为 A、B 和 C 都是集合，合并后 C 中不能出现重复的元素。基本步骤如下：

① 循环取出 A 中的元素，直接插入到 C 的最后，即 listLen+1 位置；

② 循环取出 B 中的元素，判断是否出现在 A 中，在 A 时跳过（集合元素不能重复），不在 A 中插入到 C 中。

【算法描述】

```
void ListMerge(seqList A, seqList B, seqList* pC)
{    int i;
     elementType x;
     for(i=0;i<A.listLen;i++)      //A 的元素写入 C
     {    getElement(A,i+1,x);
          listInsert(pC,x,pC->listLen+1);
     }
     for(i=0;i<B.listLen;i++)      //B 中与 A 不重复元素写入 C
     {    getElement(B,i+1,x);
          if(!listLocate(A,x))
               listInsert(pC,x,pC->listLen+1);
     }
}
```

【算法分析】假设 A 的元素个数为 m，B 的元素个数为 n，本例的时间性能主要取决于第二个 for 循环，事实上这是一个双重循环，因为 listLocate(A,x)中有循环，其时间性能为 O(m)，所以总的时间复杂度为 O(m×n)

【思考问题】

① 为什么 listInsert 中的插入位置是 pC->listLen+1，而不是 pC->listLen 呢？

② 本例使用指针从子函数往主函数传回新表 C，如果采用 C++的“引用”传递，程序该如何修改？还有其它传回 C 的方法吗？

③ 如果算法中的 A 和 B 都用“指针”或“引用”有什么不同呢？

【例 2.2】假设顺序表 L 中的元素递增有序，设计算法在顺序表中插入元素 x，要求插入后仍保持其递增有序特性，并要求算法时间尽可能少。

【算法思想】前面所介绍的基本插入算法是针对指定的插入位置进行的，此处则需要在算法中先搜索出插入的位置，然后将从该位置开始的元素往后移并执行插入。假设 L={5,10,15,20,30}，x=18，则插入后 L={5,10,15,18,20,30}。现在的问题是：如何搜索插入位置？如何移动元素？对此有两种基本的方法：

① 从表前往表后搜索插入位置，然后批量移动其后面的元素。这种方法比较费时间：假设搜索出的插入序号为 i，则搜索所需的比较次数为 i，批量移动后面元素的次数为 n-i+1。因此，对每次插入操作来说，需要比较或移动表中所有元素。

② 从后往前依次比较各元素，显然，在比较过程中，比 x 大的元素应该往后移，重复这样的操作，直到搜索到插入位置为止。这种方法相对容易实现，并且比较和移动操作同步，对每个元素来说，比较次数比移动总数多一次（除非是第一个位置）。下面的算法就是采用这种方法。

本例没有采用前面介绍的顺序表的常规插入算法，而是重新设计了一个算法。在插入前还需要做的一件事就是要判断插入的条件，显然，此处能够插入的条件就是原表中还没有满。

【算法描述】

```
void insert（seqlist *L，elementType x）
{ int i=L->listLen-1；//取表中最后元素的数组下标
  if（i>=MAXLEN-1）then error（“overflow”）；//表满，不能插入
  else { while（i>=0 && L->data[i]>x）        //往前搜索插入位置，并移动元素
         {
             L->data[i+1]=L->data[i]；//元素后移
             i--;
         }
         L->data[i+1]=x；
         L->listLen++；
       }
}
```

【算法分析】该算法花费的时间主要是比较和移动元素。在第 i 个位置插入时需移动 n-i+1 个元素，比较次数比移动次数多 1 次，即取决于插入位置。最好情况下，比较 1 次，移动 0 次，而最差情况下需要比较 n 次，移动 n 次。

【思考问题】

① 算法中的 while 循环结束时，空位置（插入位置）的下标是 i 还是 i+1?请模拟在表（4,6,10,15,20）中分别插入 25、8 和 2 时的实现过程。

② 如果 while 循环条件 L->data[i]>x 改为 L->data[i]>=x，结果会有何不同？

③ 用 for 循环能否完成相同功能呢？

【例 2.3】假设顺序表 A、B 分别表示一个集合，设计算法以判断集合 A 是否是集合 B

的子集，若是，则返回 TRUE，否则返回 FALSE，并要求算法时间尽可能少。

【算法思想】这是顺序表的一种应用形式。由题意可知，算法要采用布尔型函数形式返回求解结果。由集合的有关概念可知，判断集合 A 是否是集合 B 的子集，就是要判断 A 中每个元素是否均在 B 中出现，因此，这实际上变成了依次对 A 中每个元素，判断是否在 B 中出现（即是对 B 表的搜索问题）。其中，依次取 A 中每个元素需要以循环方式来实现控制，判断给定元素是否在 B 中出现也要以循环方式来实现控制。这样就可以得到两重循环形式的程序。

在内层循环中判断所取出的 A 中元素是否在 B 中时，若该元素不在 B 中，则整个求解结束，返回失败标志 FALSE。否则，若 A 中所有元素都判断成功，则返回成功标志 TRUE。由此得算法如下：

【算法描述】

```
BOOL subset(seqList *A, seqList *B)
{int ia,ib;   elementType x;   BOOL suc;      // ia,ib 分别指示 A、B 表中元素的数组下标
 for (ia=0; ia<A->listLen; ia++)
  { x=A->data[ia];                    //取出 A 中一个元素
    ib=0;   suc=FALSE;                //suc 为搜索成功与否的标志
    while (ib<B->listLen && suc==FALSE)
        if   (x==B->data[ib]) suc=TRUE;       //搜索到指定元素，设置成功标志
        else ib++;          //否则，继续搜索 B
    if   (suc==FALSE) return FALSE;    //若 A 表中当前元素未搜索到，立即结束
  }
 return TRUE;      //到此处时，一定是成功的
}
```

【算法分析】由于 A 中每个元素都要与 B 中每个元素比较，故该算法的时间复杂度为两表之长度的积的数量级，即为 O(|A|*|B|)。

本题也可这样求解：将判断指定元素是否在 B 表中的求解也设置为一个布尔函数的形式，在此不再赘述，有兴趣的读者可自己练习。

【思考问题】此算法函数参数 A 和 B 都是指针，不用指针可以吗？用指针有什么好处？使用 C++的“引用”是否可行？

【例 2.4】假设递增有序顺序表 A、B 分别表示一个集合，设计算法以判断集合 A 是否是集合 B 的子集，若是，返回 TRUE，否则返回 FALSE，并要求算法时间尽可能少。

【算法思想】本题虽然也可用前例算法来实现求解，但由于没有用到所给出的递增有序的条件，因而其时间性能不会得到改善，不符合对时间性能的要求。下面讨论运用所给出的递增有序这一条件来改进算法的时间性能的实现方法。

按定义，需要判断 A 表中每个元素是否在 B 表中出现，为此，可这样进行：不妨用变量 ia 依次指向 A 表中每个元素（即 A->data[ia]），然后判断该元素是否出现在 B 表中。而为了判断该元素是否在 B 表中，也需要用一个变量（不妨用 ib）依次指示 B 表中各元素，以指示搜索位置。当 ia 和 ib 均指向各自表中某一元素时，可能会出现如下几种情况：

（1）**A->data[ia]> B->data[ib]**: 即 A 表中当前元素大于 B 表中当前元素，因而需要继续在 B 表中搜索，即要执行 ib++以使 ib 指向 B 表中下一个元素并继续搜索。

（2）**A->data[ia]== B->data[ib]**: A 表中当前元素在 B 表中，即查找成功，因而可继续 A 表中下一个元素的判断，即要执行 ia++以使 ia 指向 A 表的下一个元素并继续搜索。与此相应，此时，指示 B 表中元素的 ib 应指示到哪儿？

一种简单的办法是从头开始，但那样的话，还是没有用上递增有序的条件，因而没有改善算法的时间复杂度。仔细分析可得到另一种办法，**即 ib 还是指示原来的位置，或者简单地往后移动一位，即执行 ib++也可以**。

（3）**A->data[ia]< B->data[ib]**: 即 A 表中当前元素小于 B 表中当前元素，因而肯定小于其后面的所有元素，所以该元素肯定不在 B 表中，即搜索失败。由于只要有一个元素不在 B 表中，就意味着 A 表不是 B 表的子集，故整个算法的求解结束，返回结果 FALSE。

重复执行上述判断过程，直到 ia 和 ib 中至少有一个指向表尾之后为止，此时，根据不同的情况可以分别得出如下结论：

（1）ia>A->=listLen: 意味着 A 表中每个元素都已经被判断过了，并且都在 B 表中（为什么？），因而可以返回结论 TRUE。

（2）否则，意味着 A 表中的当前元素肯定不在 B 表中，因而返回结论 FALSE。

【算法描述】

```
BOOL subset(Seqlist *A, seqList *B)
{   int ia=0, ib=0;
    while   (ia<A->listLen && ib<B->listLen)
    {
          if   (A->data[ia]== B->data[ib])   { ia++; ib++; }
          else   if   (A->data[ia]> B->data[ib])   ib++;
          else   return   FALSE;
    }
    if   (ia>=A->listLen)   return   TRUE;
                    else     return   FALSE;
}
```

【算法分析】由于 ia、ib 从头开始依次指示 A、B 表中每个元素一次（严格地说，由于停顿可能使某个元素被比较几次，但每次比较至少要通过一个元素），故算法的时间复杂度为两表长度之和的数量级，即 O(|A|+|B|)。

【思考问题】

①那个表会先搜索结束？

②本算法在什么情况下会出现 ia 和 ib 同时等于对应表的长度？

【例 2.5】设计算法将递增有序顺序表 A、B 中的元素值合并为一个递增有序顺序表 C，并要求算法时间尽可能少。

【算法思想】如果没有要求递增有序，而是简单合并，则可分别将两个表中的元素依次放置到 C 表中即完成了问题要求。然而，由于有递增有序这一要求，因而必须要重新构思求解方法。

假设用 ia 和 ib 依次指向 A、B 两表中的元素（一般都是从前往后，即初值都为 0），则可能会出现如下情况之一：

（1）**A->data[ia]< B->data[ib]**: 则 A 表中当前元素应先于 B 表中当前元素放到 C 表

中，然后再用 A 表中下一元素来与 B 表中当前元素比较，即要执行 ia++以使 ia 指示下一元素。

（2）**A->data[ia]==B->data[ib]**: 则可将 A、B 两表中当前元素同时放在 C 表中（也可仅放其中一个），然后再对 A、B 表中下一元素来比较。

（3）**A->data[ia]> B->data[ib]**: 则 B 表中当前元素应先于 A 表中当前元素放到 C 表中，然后再用 B 表中下一元素来与 A 表中当前元素比较，即要执行 ib++以使 ib 指示下一元素。

重复执行上述比较操作，直到当 ia 和 ib 中的一个指向表尾之后时，可以将另一个表中的剩余元素放到 C 表中。另外，还需要解决以下的问题：

（1）C 表中元素存放位置的指示；（2）C 表的长度值的设置（许多初学者容易忽略）

【算法描述】

```
void merge(Seqlist *A, seqList *B, seqList *C)
{   int ia=0, ib=0, ic=0;   elementType x;
    while   (ia<A->listLen && ib<B->listLen)
    {
        if   (A->data[ia]== B->data[ib])
        {
            C->data[ic++]=A->data[ia++];
            C->data[ic++]=B->data[ib++];
        }
        else   if   (A->data[ia]> B->data[ib])   C->data[ic++]=B->data[ib++];
        else   C->data[ic++]=A->data[ia++];

        while   (ia<A->listLen)   C->data[ic++]=A->data[ia++];
        while   (ib<B->listLen)   C->data[ic++]=B->data[ib++];
        C->listLen=ic;
}
```

【算法分析】由于每次循环都要取出 A 或者 B 中的一个元素插入 C 中，所以算法的时间复杂度为两表长度之和的数量级，即 O(|A|+|B|)。

【思考问题】

① 若要求 C 为递减顺序表，如何实现？

② 若已知顺序表 A 递增，B 递减，要求 C 递增，如何实现？

2.3 链表

由前面讨论可知，在顺序表中插入和删除一个元素时，在等概率情况下，需要移动表中约一半的元素。当表中元素较多时，显然是费时的。此外，顺序表需要使用连续的内存空间，且空间大小要按最大需求分配，可能导致内存空间利用率不高。为此，需要重新讨论线性表的存储结构问题。下面所讨论的链表就是一种新的存储结构形式。

2.3.1 链表基本结构

1. 链表的概念

链表使用不连续的，或连续的存储空间来存放线性表的数据元素。那么在不连续的内存空间上怎样实现线性表元素间的逻辑次序呢？即，从当前元素出发，怎样找到它的直接后继在什么地方呢？链表的做法的是在每个数据元素后面附加一个地址域（指针域），用来存放当前元素直接后继的地址。这个由数据域和指针域形成的结构作为一个整体，叫做**结点（节点）**。结点的构成如图 2-2 所示：

设要在表 L 中的第 i 个位置上插入值为 x 的元素。为了实现插入，并且不移动元素，需要另外开辟存储单元来存放要插入的元素 x 的值。在这种情况下，为了能体现出 x 与其新的前驱、后继元素之间的逻辑关系（次序），可这样实现：为每个元素增加一个变量以记录元素的地址，从而构成了称为**结点**的一个整体，其形式如图 2-2 所示：

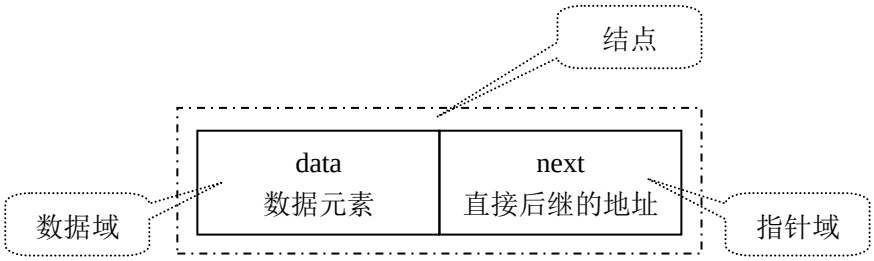

图 2-2 结点结构

链表中，结点是存取的基本单位，是一个整体。结点的数据域存放数据元素；指针域存放直接后继结点的地址，也叫指针或链。一个有 n 个元素的线性表，构成 n 个结点存储到内存不连续，或连续的存储空间中，逻辑上相邻的元素在物理位置上不一定相邻，元素之间的逻辑次序通过指针进行链接，故称以这种存储形式所存储的表为**链表**。

比如，由 e_1 元素结点的 next 即可找到 e_2，由 e_2 结点的 next 即可找到 e_3 结点，这样“顺藤摸瓜”，一直可以找到 e_n，不管他们存储在内存的什么位置。

我们怎么知道线性表的第一个结点存放在内存的什么地方呢？显然我们需要知道其在内存中的地址，这个地址叫做**头指针（head）**。有了头指针就可找到第一个结点，由第一个结点的 next 指针即可找到第二个结点，如此可以找到所有结点。所以，链表中头指针是至关重要的，因为头指针唯一确定整个链表。我们以后在定义链表时，事实上只要定义出其头指针即可。由于头指针也是指向一个结点，所以 head 指针与 next 指针具有相同的数据类型。链表的存储结构如图 2-3 所示：

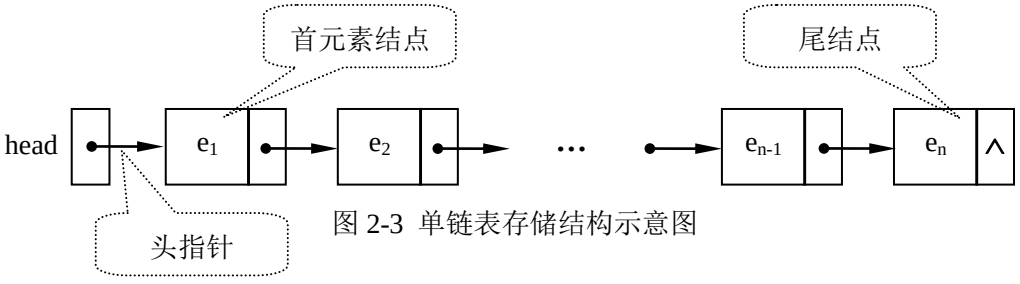

图 2-3 单链表存储结构示意图

图 2-3 所示链表的结点由一个数据域和一个指针域构成，称这样的链表叫做**单链表**（Single Linked List）。

2. 单链表的存储描述

描述单链表，其实只要定义出结点结构，有了结点结构，再给出结点类型的头指针即可以确定整个链表。由上面的讨论可知，单链表结点结构由数据域和指针域两个分量构成，借助 C 语言的结构体很容易定义出结点结构，描述如下：

```
struct   slNode
{        elementType     data;    // 数据域
         struct slNode*   next;    // 指针域，结构（结点）自身引用
};
typedef struct   slNode   node, *linkList;   //或
typedef slNode   node, *linkList;
```

或者将上面两部分集成描述为：

```
typedef struct   slNode
{        elementType      data;   // 数据域
         struct slNode*     next;   // 指针域，结构（结点）自身引用
} node, *linkList;
```

上面的描述中使用 typedef 将结点类型重新命名为 node，和结点指针类型 linkList。以后我们可以使用类型 node 来定义结点变量或结点指针变量，也可用 linkList 直接定义结点指针变量。

这里讨论的链表要求按需分配结点的存储空间，即要求程序在运行期间可以动态地向操作系统申请需要的存储空间，使用完毕立即释放空间，这样的链表叫做**动态链表**。C 语言为我们提供了这样的库函数，**malloc()**库函数让我们在程序执行期间向操作系统申请内存空间，**free()**库函数让我们动态的释放内存空间。C++提供了两个操作符 new 和 delete，分别用来动态申请和释放空间。本书后面在描述申请和释放结点时更多使用 new 和 delete 操作符。

需要注意的是，C 或 C++中，用户动态申请的内存空间，使用完毕后必须由用户负责释放这部分空间，还给操作系统。否则，操作系统会一直认为这些空间仍为用户程序使用，这样的内存叫做垃圾内存（Garbage Memory），或叫产生了内存泄漏（Memory Leak）。C 和 C++没有垃圾内存自动回收机制，用户申请的内存，需要用户自行负责回收。所以，malloc 和 free，new 和 delete 必须成对使用，申请了多少内存，用完后就释放它，否则即造成内存泄漏。Java 和 C#都提供自动垃圾回收机制，动态申请的内存即使不立即手工释放，系统会根据需要启动垃圾回收机制，自动回收垃圾内存。

申请结点和释放结点的简单用法如下：

malloc()申请结点：

```
node* p;
p=( node* )malloc( sizeof(node) );   //动态申请一个结点的内存空间，返回结点指针
```

free()释放结点：

```
free(p);
```

new 申请结点：

```
node* p;
```

p=new node; //动态申请一个结点的内存空间，返回结点指针

delete 释放结点：

delete p;

有了结点指针，怎样引用结点结构的分量呢？比如有下面的代码：

node* p;

p=new node;

上面的代码定义了一个结点指针 p 并申请了结点的内存空间，用“->”符号来引用结点的分量，即：p->data; p->next。

前面讨论过，一个链表由头指针唯一确定，那么头指针怎样定义呢？因为头指针也是指向结点结构的指针，所以头指针类型与上面定义的指针 p 或 next 指针类型相同，所以头指针可以定义为：**node* head**。以后我们会常常用头指针表示整个链表。同时，为简单见，也常用单个字符来表示头指针，如，node* H 或 node* L。

头指针的一些用法如图 2-4 所示：

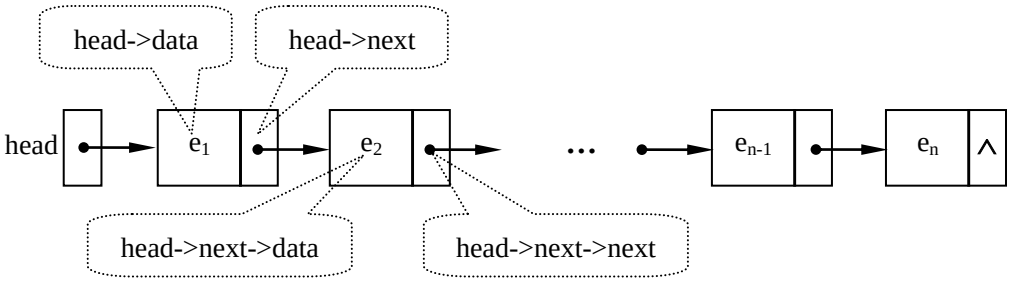

图 2-4 头指针用法示意图

可见，head->data 为首元素结点的元素值 e_1；head->next 为指向 e_2 结点的指针。因为 head->next 为 e_2 结点指针，所以，head->next->data 即为 e_2；head->next->next 为指向下一个结点 e_3 的指针，如此继续下去，可以直到尾结点。

3. 有关链表结构的进一步讨论 –– 带头结点的单链表

本小节我们从探讨单链表插入运算开始，引出**带头结点的单链表**概念。假设要在链表的第 i 个元素结点前插入值为 x 的新结点，要完成此操作，首先要用 s=new node 申请一个新的结点，s 为指向新结点的指针；将元素值 x 赋给结点的数据域，即执行 s->data=x；还要知道前一个结点 e_{i-1} 的指针，假设为 p；最后修改 x 结点和 e_{i-1} 结点的 next 指针，将新结点链接到链表中。由 2.2 节顺序表的插入操作可知元素的有效插入范围为：1<=i<=n+1。我们可以把此范围划分为三种情况分别加以讨论，情况一：2<=i<=n，即新结点 x 插入后不是第一个结点，也不是最后一个结点；情况二：i=n+1，即新结点 x 插入后为最后一个结点；情况三：i=1，即新结点 x 插入后为第一个结点（首元素结点）。

情况一：2<=i<=n

此情况下插入结点 x，首先要修改 x 结点的 next 指针，使其指向 e_i 结点，如图 2-5 所示，可通过代码 s->next=p->next 实现。

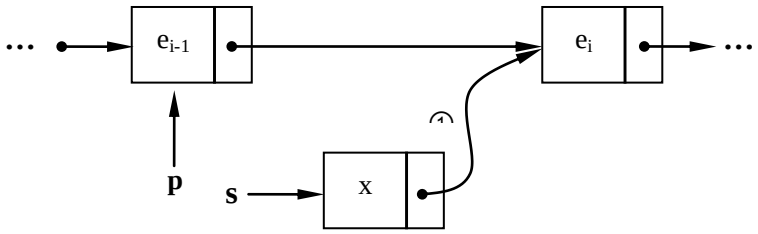

图 2-5 更新 x 结点的 next 指针后情况

接下来，通过执行 p->next=s 修改 e_{i-1} 结点的 next 指针，使指向 x 结点，即为 s，修改后新结点 x 即链接到了链表，插入工作完成，如图 2-6 所示。

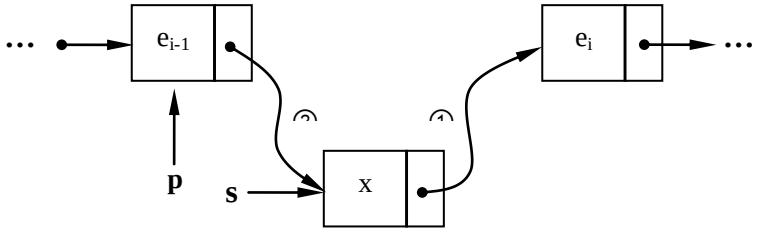

图 2-6 更新 e_{i-1} 结点的 next 指针后情况

总结起来，情况一下插入结点经过了两个步骤，即：

① s->next=p->next；

② p->next=s；

【思考问题】上述插入操作的两个步骤是否可以调换？

情况二：i=n+1

此情况下，新结点 x 插入后成为最后一个结点（尾结点），与情况一相同，第一步更新 x 结点的 next 指针为空指针，即执行 s->next=NULL，因为 e_n 结点的 next 也为 NULL，所以也可以执行 s->next=p->next；第二步更新 e_n 结点的 next 指针，执行 p->next=s，如图 2-7 所示。可见情况二与情况一操作相同，可归为情况一。执行的两个步骤如下：

①s->next=p->next；或 s->next=NULL;

② p->next=s；

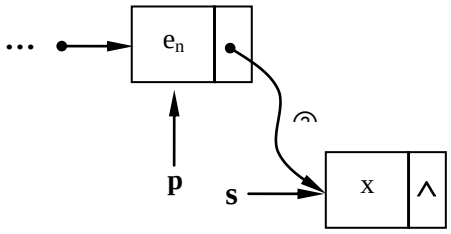

图 2-7 更新 x 结点和 e_n 结点的 next 指针后情况

情况三：i=1

此情况下，插入的新结点 x 成为首元素结点，第一步更新 x 结点的 next 指针要执行 s->next=head；第二步，链表的头指针发生变化，要改为 s，即 x 结点的指针成为链表的头指针，即执行 head=s，如图 2-8 所示，此情况下，两个操作步骤为：

① s->next=head；

② head=s；

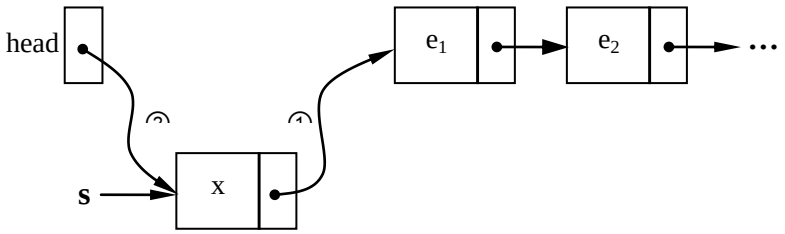

图 2-8 x 结点插入为首元素结点情况

可见，情况三下插入结点的操作与前两种情况不同，这显然会给结点的插入操作带来不便。此外，链表的头指针会随着在头部插入结点而变化。不仅插入结点存在这个问题，链表删除操作也有这个问题。为了解决上述问题，我们在链表的最前面人为增加一个结点，称为**头结点**。头结点的类型与元素结点相同，让链表的头指针指向头结点，头结点的 next 指针，即 head->next，指向首元素结点，如图 2-9 所示。我们称加了头结点的单链表为**带头结点的单链表**。

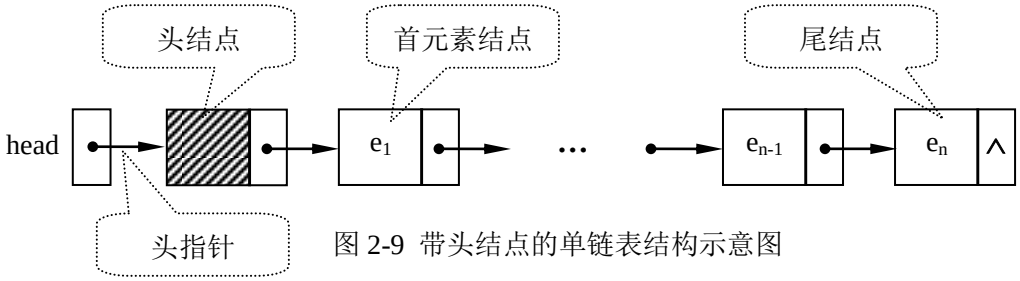

图 2-9 带头结点的单链表结构示意图

我们称没有元素结点的链表为**空链表**，对**带头结点的空单链表**，我们只需将头结点的 next 置为空指针即可，如图 2-10 所示。

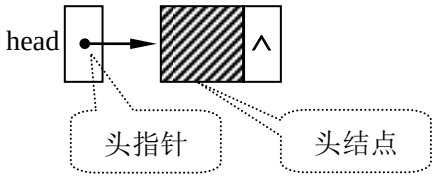

图 2-10 带头结点的空单链表意示图

加了头结点后，链表的插入和删除操作只能在头结点之后进行，这使得链表所有位置的插入和删除操作步骤相同。此外，只要申请了头结点，则整个链表在存续期间头指针始终不变。在后面的讨论中，若不特别声明，所述单链表均为带头结点的单链表。

2.3.2 链表基本运算的实现

下面讨论采用链表结构存储时，线性表各基本运算的实现。掌握和理解这些基本运算的实现方法有助于将其应用于更为复杂的问题求解中。在此基础上，我们将在下一小节讨论链表结构的应用实例。

1．初始化链表运算的实现

初始化链表即建立一个不含元素结点的空链表，对于带头结点的单链表来说，就是要编写一个函数，在此函数中申请头结点，头结点的 next 指针置为空，并将头结点的指针（头指针）返回给主调函数。

【算法描述：方法一】使用 C++的引用回传初始化后的链表

```
void   initialList（node * &L）
{   L=new node;          //产生头结点。也可用下面语句产生头结点：
                         //L=（node *）malloc（sizeof（node））;
    L->next=NULL;        //设置后继指针为空
}
```

此算法使用了 C++的**引用**往主调函数回传头结点指针（头指针）。那么如果这里不使用**引用**，初始化函数参数只定义为结点指针，即定义为 initialList(node* L)，是否可以正确回传头指针呢？**答案：不行！**为什么不行呢？假设主调函数如下面代码片段所示，当执行语句①申明头指针变量 L 时，系统会给 L 赋一个初始值，Windows 下这个初始值常为“0xCCCCCCCC”，这是 L 初始指向的内存地址。主调函数执行语句②，把 L 作为实参传递到初始化函数。在初始化函数中将执行 L=new node 语句，这句执行的结果是操作系统根据当前的内存情况，在合适的位置分配一个结点的空间，并把这个结点在内存中的首地址赋值给 L，比如为“0x00441150”，可见主函数和子函数中 L 的值发生了改变，指向的内存地址不同，所以子函数不能回传已经申请的头结点指针。C++的引用使用“别名”机制可以实现本算法的回传，事实上引用是使用了 L 的地址（指针的地址）实现传递，在此调用过程中，L 在主函数和子函数中的地址是相同的、不变的，比如皆为“0x0012FF7C”。关于引用的更多知识请您参阅 C++的相关材料。

【算法描述：方法二】使用指针的指针回传初始化后的链表

```
void   initialList（node ** pL ）
{   (*pL)=new node;          //pL 为结点指针的指针，所以(*pL)为结点指针。
                             //产生头结点。也可用下面语句产生头结点：
                             //(*pL)=（node *）malloc（sizeof（node））;
    (*pL)->next=NULL;        //设置后继指针为空
}
```

这个实现使用了指针的指针，即 L 的指针，在传递过程中 L 的地址（指针）是相同的，不变的，比如皆为“0x0012FF7C”，所以可以用双重指针回传初始化后的链表。主调函数中的调用方式如下面代码片段中的语句③。

【算法描述：方法三】使用函数返回值回传初始化后的链表

```
node* initialList()
{
```

```
    node* p;     //申明结点指针变量
    p=new node;        //产生一个结点
    p->next=NULL;   //结点的 next 指针置为空。
    return P;          //返回已申请结点的指针
}
```

这可能是初学者最容易理解的一种实现方法，主调函数中的调用方式如下面代码片段中的语句④。

【主调函数代码片段】不妨设主调函数即为 C 语言主函数 main，当然也可为其他自己编写的函数。

```
void   main()
{    node* L;              //① 申明链表头指针变量
       …
     initialList(L);       //② 调用方法一，初始化链表
     //initialList(&L);    //③ 调用方法二，初始化链表
     //L=initialList();    //④ 调用方法三，初始化链表
       …
}
```

【算法分析】算法时间复杂度 O(1)。

【思考问题】读者可自己设计出链表不带头结点时的初始化算法。

这里我们用了较多的篇幅讨论头指针的回传问题，是因为不仅链表初始化会遇到这个问题，后面的按序号求元素运算、元素定位运算、链表创建运算都会涉及到链表头指针或结点指针的回传问题，凡是在主函数和子函数中结点指针值发生改变，又要回传此指针，都存在这个问题。当然学习数据结构，您只要会使用其中一种方法即可。但我还是希望您能举一反三，彻底搞清楚这个问题，特别是您要成为 C 和 C++高手，您就必须搞清楚这些东西，在后面的内容中将不再重述。

2. 求链表长度的实现

按定义，求链表 L 的长度就是求出链表 L 中元素的个数，而链表中没有存储其长度值，因此需要逐个“数”出其结点个数。“数”结点时可用一指针（不妨用 p）依次指示每个元素结点（显然，其初值应为 L->next，指向首元素结点），p 每指到一个结点就作一次计数（因此需要设置一个计数变量 len，其初值为 0），直到搜索到最后，即 p 移出链表。由此得流程图如图 2-11 和算法如下：

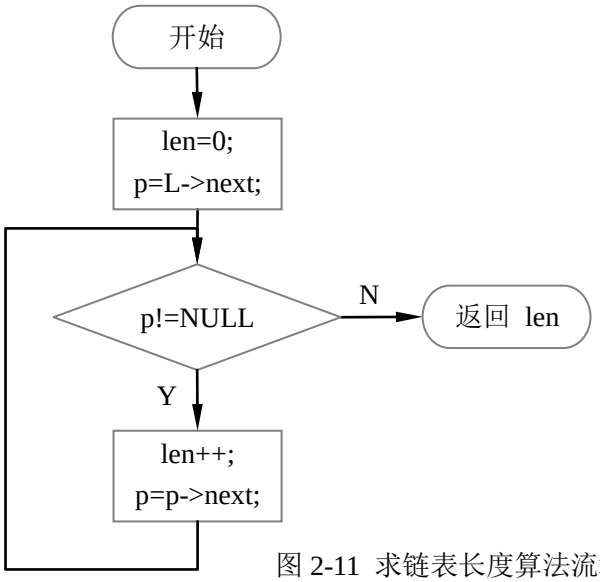

图 2-11 求链表长度算法流程图

【算法描述】

```
int   listLength（node *L）
{     int   len=0；node *p=L->next；//p 初始指向首元素结点
      while（p!=NULL）
      {   len++；              //p 指向元素结点，计数加 1
          p=p->next；          //p 移到下一个结点，继续后续结点的计数
      }
      return   len；             //返回结果
}
```

【算法分析】本算法的主要操作是计数元素结点个数，设线性表有 n 个元素，则时间复杂度为 O(n)。而顺序表中有一个长度分量 listLen，可直接读出，求长度算法的复杂度为 O(1)。

【思考问题】

① 算法初始化时，若 p 指向头结点，即 p=L，而不是首元素结点，算法要怎样修改才能计数出结点个数？这个问题在插入结点时将会遇到。

② 若用指针或引用来返回长度值，算法如何实现？

3. 按序号取元素结点的实现

该运算可以变成求链表 L 中指定序号 i 的元素结点的指针，为此，需要从首元素结点开始依次“数”过去，因而要用循环语句来控制这一搜索过程。在搜索过程中，要用一个指针（不妨用 p）依次指向所数到的结点（即*p），显然其初值应为 L->next，指向首元素结点。并需要设置一个计数变量（不妨设为 j）以记录所指结点的序号。当 j==i 时，p 指向的结点即为目标结点。算法中需要注意控制循环的条件。另外，需要考虑在所指定结点不存在时所返回的值。流程图如图 2-12 所示。

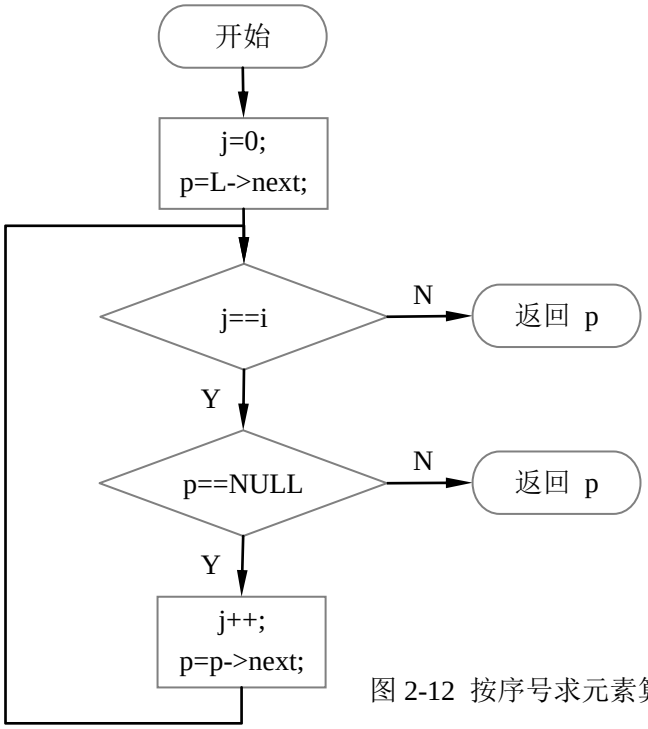

图 2-12 按序号求元素算法流程图

这一流程图中的循环有两个条件，而 C 语言中的循环条件只有一个，因此，需要将这两个条件合并为一个条件，这是较常见的，因而应注意掌握。由此得算法如下。

【算法描述】

```
node * getElement（ node *L，int  i ）
{    node  * p=L->next；  int  j=0；
     while（ j!=i && p!=NULL ）   //当前结点不是目标结点，并且不空时就继续搜索
     {
          j++；
          p=p->next；
     }
     return p;                 //返回结果
}
```

【算法分析】分析方法同顺序表按序号求元素算法，时间复杂度为 O(n)。

【思考问题】

① 本算法若取元素失败，返回的 p 值为什么？

② 用引用或双重指针返回指针 p，如何实现？

4. 按值查询元素的实现

分析：在链表 L 中查找值为 x 的元素结点，并返回其指针作为结果，显然要采用逐个比较的方法来实现。具体方法是：设置一个指针（不妨用 p）依次指示各元素结点，每指向一个结点时，就判断其值是否等于 x，若是，则返回该结点的指针；否则继续往后搜索，直

到表尾；若没有找到这样的结点，则返回空指针（这是常见的要求）。算法流程图如图 2-13。

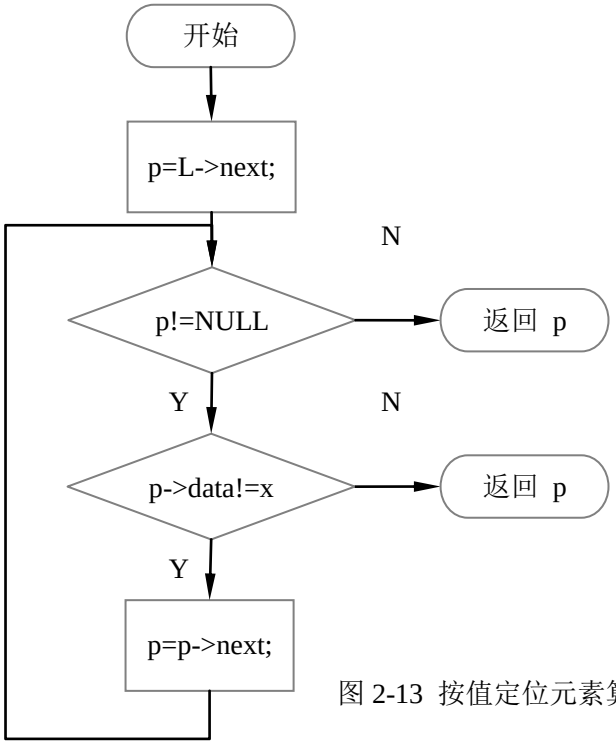

图 2-13 按值定位元素算法流程图

【算法描述】

```
node * listLocate（ node *L，elementType x ）
{    node   *p=L->next；    //p 初始指向首元素结点
     while（ p!=NULL && p->data!=x ）
     {                    //p 指元素结点，又不是目标结点，继续搜索下一个结点
          p=p->next；
     }
     return   p；
}
```

【算法分析】分析方法同顺序表按元素值定位元素，时间复杂度为 O(n)。

【思考问题】

① 若查找失败，返回的 p 值为什么？

② 若用引用和双重指针返回目标结点指针，如何实现？

③ 若要求同时返回目标结点的序号，应如何修改算法？

5. 插入算法的实现

分析：结点的插入操作我们在 2.3.1.3 小节已经进行了讨论，我们知道加了头结点后，链表所有位置的插入操作步骤都是相同的，并给出了插入操作的关键步骤。插入操作过程如图 2-14 所示。这里再对插入操作的其它内容做详细分析。在链表 L 的第 i 个元素结点前插入值为 x 的结点，即新结点插入到结点 e_{i-1} 和 e_i 之间，需要完成下列工作：

（1） 搜索 e_{i-1} 结点并给出 e_{i-1} 结点的指针 p

算法中给出插入位置 i，i 为元素序号，我们现在需要根据给出的序号 i，搜索 e_{i-1} 结点，并给出其指针 p。我们在前面按序号求元素算法 getElement(L, i)中遇到过类似的问题，但此算法中给出的是 e_i 结点的指针。而我们现在要求在给定 i 情况下，给出 e_{i-1} 结点的指针；此外，插入位置可能是 i=1，这样其前驱就是头结点，p 即为头指针，所以 p 只能初始化为 p=L，为此，我们需要对先前的代码进行改造如下：

```
p=L;  //p 初始指向头结点
j=0;
while( j!=i-1 && p!=NULL)  //未找到 e(i-1) 结点，且未到表尾，继续搜索下一结点
{
    j++;
    p=p->next;
}
```

此循环在两种情况下退出，一种情况是 j==i-1 退出，此时 p 指向 e_{i-1} 结点，正确搜索到插入位置，这正式我们要求的；第二种情况是 p=NULL 退出，说明插入的元素序号 i 超出有效范围。

与顺序表插入一样我们也要检查插入位置 i 的合法性，i 的有效范围为：1<=i<=n+1。在顺序表中我们用表长度参数 listLen 可以直接判断 i 是否超出范围，那么，这里我们怎样检查呢？直接的想法可能是先利用 listLength(L)运算，求出链表的长度 n，再判断 i 是否在 1<=i<=n+1 范围内。有没有更为巧妙的方法呢？答案是肯定的。我们可以借助上面搜索 e_{i-1} 结点指针 p 的过程，顺便判断 i 是否超出有效范围。我们先来分析 i>n+1 情况，即 i-1>n。上面的 while 循环，当 j=n 时，p 指在尾结点 e_n 上，此时 j=n，i-1>n，所以 j!=i-1，又 p 不为空，将继续执行循环体 j++和 p=p->next，此后 p 变为空，退出循环，可见 i>n+1 时，while 循环因 p==NULL 退出。再来分析 i<1 情况，即 i-1<0。因 while 循环控制变量从 j=0 开始，循环一开始就有 j>i-1，此后每次循环 j 加 1，所以永远不可能出现 j==i-1 情况，while 循环只能在 p==NULL 时退出，可见 i<1 时，while 循环也以 p==NULL 退出。综上可知，while 循环结束后，如果 p==NULL，则肯定是因为 i 超出有效范围。在 while 循环后面，增加下面代码即可检查 i 的合法性。

```
if( p==NULL ) error("序号错");
```

顺便提一下，利用上面的方法判断 i 的有效性，在 i<1 和 i>n+1 两种无效序号情况下，都要搜索链表的所有结点，直至表尾。事实上，对 i<1 无效情况，我们可以在 while 循环前直接加一条 if(i<1) error("序号下越界") 语句排除，不需要搜索整条链表，效率会更高。

（2） 申请一个新结点，结点指针 s

```
s=new node;  //或：s=( node * ) malloc( sizeof( node ));
```

（3） 将元素值 x 赋给新结点的数据域

```
s->data=x;
```

（4） 更新新结点的 next 指针，使指向结点 e_i

```
s->next=p->next;
```

（5） 更新 e_{i-1} 结点的 next 指针，使指向结点 x。

```
p->next=s;
```

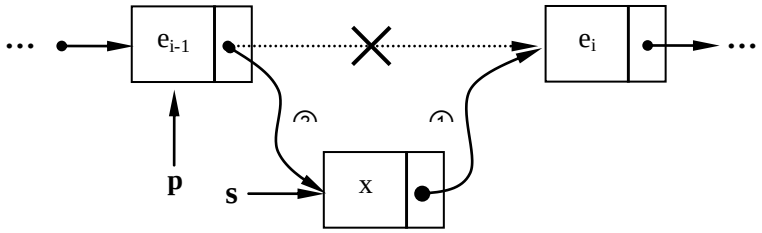

图 2-14 结点插入操作示意图

【算法描述】

```
void   listInsert（ node *L，int i，elementType x ）
{     node * p=L；int j=0；
      while（ j!=i-1 && p!=NULL ）        //搜索 a_{i-1} 结点
      {
            p=p->next；
            j++；
      }
      if（p==NULL ） error（“序号错”）；       //等价于判断插入序号是否正确
      else
      {     s=new node;          //产生新结点。或用下面语句申请新结点：
                                 //s=( node* )malloc( sizeof( node ));
            s->data=x;           //装入数据 x
            s->next=p->next;     //插入（链接）新结点
            p->next=s;
      }
}
```

【算法的一种实现】

```
//*************************************************************//
//* 函数功能：表中指定位置 i 插入 x 元素                          *//
//*           插入成功返回 true；失败返回 false                   *//
//* 输入参数：node *L, int i, elementType x                     *//
//* 输出参数：node *L                                           *//
//* 返 回 值：bool，插入成功返回 true，失败返回 false            *//
//*************************************************************//
bool listInsert(node* L, int i, elementType x )
{
     node* p=L;   //p 指向头结点
     node* S;
     int k=0;
     while(k!=i-1 && p!=NULL)   //搜索 e_{i-1} 节点，并取得指向 e_{i-1} 的指针 p
     {
```

```
            p=p->next;   //p 指向下一个节点
            k++;
        }

    if( p==NULL )
        return false;   //p 为空指针，说明插入位置 i 无效，返回 false
    else
    {
            //此时，k=i-1，p 为 e_{i-1} 结点的指针
        S=new node;    //动态申请内存, 创建一个新节点，即：要插入的节点
        S->data=x; //装入数据
        S->next=p->next;
        p->next=S;

      return true;   //正确插入返回 true
    }
}
```

【算法分析】算法的主要费时操作在循环搜索 e_{i-1} 结点的指针 p，可见时间复杂度为 O(n)。

【思考问题】主调函数中也需要知道插入结点情况，为什么这里函数参数用结点指针（单指针）就可以了呢？

6．删除算法的实现

分析：删除链表中第 i 个元素结点，e_i，就是要让其前驱 e_{i-1} 的 next 指针绕过该结点指向 e_{i+1} 结点，释放 e_i 结点。结点删除操作过程如图 2-15 所示。

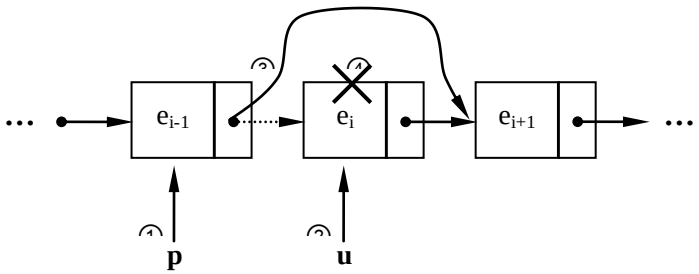

图 2-15 结点删除操作示意图

可见要删除一个结点，我们需要完成以下工作：

（1） 搜索 e_{i-1} 结点，给出此结点指针 p

p 指针的搜索过程与前面结点插入算法的搜索过程相同，当给出的删除元素序号 i 有效时，while 循环由 j==i-1 条件控制结束，此时 p 指向 e_{i-1} 结点，p->next 指向 e_i 结点，即要删除的目标结点。

删除操作也要检查序号 i 的合法性，i 的有效范围为：1<=i<=n。对与 i<1 无效情况，

i 的无效判定方法与插入操作的判定方法完全相同。对与 i>n 无效情况，与插入算法判定稍有不同。当 i>n+1 时，while 循环由 p==NULL 控制退出，这与插入算法判定一样。但是当 i=n+1 时，两者不同，对插入是合法位置，但对删除就是无效序号。当 i=n+1 时，j=n，此时 p 指在 e_n 结点上，即有 j==i-1 和 p!=NULL，while 循环因 j=i-1 条件满足而退出，退出后 p 指向 e_n 结点，不为空，但 **p->next 为空**。而我们要删除的目标结点指针正是 p->next，p->next 为空则没有结点可供删除，即序号 i=n+1 无效。综上分析，while 循环结束时，p==NULL 或者 p->next==NULL，都可判定为删除序号 i 无效。即：

if(p==NULL ‖ p->next==NULL) error(“删除序号错”);

（2） 获取 e_i 结点指针 u，用以后面释放此结点

u=p->next;

（3） 更新 e_{i-1} 结点的 next 指针，使绕过 e_i 结点，指向 e_{i+1} 结点

p->next=u->next; 或者 p->next=p->next->next;

（4） 释放 e_i 结点的存储空间

delete u; 或者 free(u);

前面已经介绍过，C 或 C++中用户申请的内存，不用时必须用 free()或 delete 显示地释放，系统不能自动回收这部分内存空间。

【算法描述】

```
void listDelete（node *L，int i）
{
    node *p，*u;
    int j=0;
    p=L;
    while（j!=i-1 && p!=NULL）      //搜索 e(i-1) 结点
    {
        p=p->next;
        j++;
    }
    if（ p==NULL ‖ p->next==NULL ） //判定删除序号 i 是否有效
        error（“删除序号错”）;        //报错并退出运行
    else
    {
        u=p->next;                  //指向待删除的结点
        p->next=u->next;            //绕过待删除的结点
        delete u;                   //释放结点的存储空间。或用 free（u）;
    }
}
```

【算法的一种实现】

```
//*********************************************************//
//* 函数功能：删除表中指定位置 i 处的元素（结点）              *//
//*           删除成功返回 true；失败返回 false               *//
//* 输入参数：node* L, int i                                *//
```

```
//* 输出参数：node* L，返回删除后的链表                          *//
//* 返 回 值：bool，删除成功 true；失败 false                    *//
//************************************************************//
bool listDelete(node* L,int i)
{
    node* u;
    node* p=L;   //指向头结点
    int k=0;
    while(k!=i-1 && p!=NULL)   //搜索 e(i-1) 结点
    {
        p=p->next;
        k++;
    }
    if(p==NULL || p->next==NULL)
        return false;     //删除位置 i 超出范围，删除失败，返回 false
    else
    {                     //此时，p 指向 e(i-1)
        u=p->next;        //u 指向待删除结点 e(i)
        p->next=u->next;     //e(i-1) 的 next 指向 e(i+1) 结点，或为空（e(i-1) 为最后结点）
        delete u;         //释放 e(i) 结点
        return true;      //成功删除，返回 true
    }
}
```

【算法分析】算法时间复杂度 O(n)。

7．链表的构造

前面讨论的链表六种基本运算，除了初始化链表外，其他五种运算，我们均假设链表已经存在，且可能已经有若干个元素结点。那么，这个链表从哪来呢？我们知道链表是在程序运行期间，通过动态申请内存逐步构建的，所以要使用链表，必须先创建它。

创建链表即先初始化一个链表，然后连续地插入若干个元素结点。您可能立马想到，这很简单，先调用基本运算（1）初始化一个链表，然后反复调用基本运算（5）的插入结点运算就可以完成这个任务。按这个思路的确可以完成任务，但运算（5）每次都要搜索插入位置 i，显然时间性能不理想。因此，如果在构造算法中能记住上一次的插入位置，按这个位置直接插入当前结点，则可解决其时间性能问题。下面所讨论的算法即是基于这一基本思想的。我们约定从键盘输入数据元素。那么如何控制创建结束呢？一般有两种方法，一种是循环创建结点，当键盘输入一个特殊符号（结束符），退出循环，结束创建；另一种是先指定结点个数，由此控制循环，当插入的结点数等于这个值时，退出循环，创建结束。

算法的基本框架如下：

① 初始化链表 L；

② 读入元素==>x；

③ 如果 x 是结束符，结束构造；

④ 否则，产生结点并装入 x；

⑤ 插入新结点到链表 L 中；

⑥ 转 ②。

在这一框架中，需要明确操作 ⑤ 中插入结点的位置。可有几种不同的选择：

① 插入到表尾，简称**尾插法**，即：每次插入的结点都成为为节点，从而使所建链表中元素的次序与输入的顺序一致。

② 插入到表头，简称**头插法**，即，插入到头结点之后，每次插入的结点都成为首元素结点，从而使所建链表中元素的次序与输入的顺序相反。

③ 其它要求的插入，例如按递增、递减的要求插入等。

下面分别讨论前两种方法，即尾插法和头插法建表的算法。这两个算法也是许多链表问题求解的基础。

（1）尾插法创建链表的算法

如前所述，所谓尾插法建表是指在创建链表的过程中，将每次所读入的数据装入结点后插入到链表的尾部，使成为新的尾结点。分析如下:

① 为使创建算法相对独立，这里不调用基本运算（1）的链表初始化操作。我们直接在创建算法中申请头结点，给出头指针，即执行 L=new node 即可。

② 为能很快找到插入的位置（即原来的尾结点），提高插入的速度，可设一指针 R，始终指向尾结点,这样每个新结点可直接链接到 R 所指的结点上。

③ 插入每个结点时, 除将其链接到链表尾部外, 还要移动原指针 R，使其指向新的尾结点, 以便后续结点的插入操作。

依此分析得尾插法建表算法的基本框架如下：

① 产生链表 L 的头结点，头结点 next 置空，并让指针 R 指向头结点，即：R=L。

② 读入元素==>x。

③ 如果 x 是结束符，结束构造。

④ 否则，产生结点并装入 x。

⑤ 将新结点链接到指针 R 所指结点的后面，并让指针 R 指向新的尾结点。

⑥ 转②。

【算法描述】

```
void   createList( node * &L );          //尾插法建表
{
    L=new node;        //申请产生头结点，头指针为 L。或用下面语句申请：
                       //L=（node *）malloc（sizeof（node））;
    R=L;               //设置尾指针
    cin>>x;            //读入键盘输入的第一个数据到变量 x
    while ( x!=End_of_Symbol )          //x 不是结束符时，循环插入。
    {                                   // End_of_Symbol 表示结束符
        u=new node;        //申请新结点
        u->data=x;         //装入数据
        u-> next=NULL      //新结点 next 指针置空。或用 u->next=R->next;
        R->next=u;         //将新结点链接到链尾
```

```
        R=u;                    //尾指针后移，以指向新的尾结点
        cin>>x;                 //读入下一个键盘输入数据到变量 x
    }
}灌
```

【算法的一种实现 — 结束符控制创建结束】

```
void createListR( node *& L )
{
    elementType x;    //保存键盘输入的数据元素值
    node *u, *R;      //L 为头结点指针（头指针），R 为尾结点的指针
    L=new node;       //产生头结点，头指针为 L
    L->next=NULL;  //头结点的指针域为空
    R=L;  //设置尾指针，对空链表：头、尾指针相同

    cout<<"请输入元素数据（整数，9999 退出）:"<<endl;    //本例以 9999 为结束符
    cin>>x;      //读入第一个键盘输入数据到变量 x

    while(   x!=9999   )
    {
        u=new node;          //动态申请内存，产生新节点
        u->data=x;           //装入元素数据
        u->next=NULL;     //新结点 next 指针置空
        R->next=u;           //新结点链接到表尾
        R=u;                 //修改尾指针，使指向新的尾结点

        cin>>x;              //读入下一个键盘输入数据
    }
}
```

【算法的一种实现 — 结点个数控制创建结束】

```
void createListR( node *& L )
{
    int i,   n;           //n 为元素结点个数，不含头结点
    elementType x;    //数据元素值
    node *u, *R;      //R 为尾结点指针

    L=new node;       //产生头结点
    L->next=NULL;  //头结点的指针域为空
    R=L;              //设置尾指针，对空链表头、尾指针相同

    cout<<"请输入元素结点个数（整数）：n=";      //输入元素结点个数，保存到 n
    cin>>n;
```

```
    cout<<"请输入元素数据（整数）:"<<endl;
    for( i=n; i>0; i--)       //尾插法循环插入节点
    {
        u=new node;     //动态申请内存，产生新结点
        cin>>x;         //键盘输入元素数据
        u->data=x;      //元素数据装入新结点
        u->next=NULL;
        R->next=u;      //新结点链接到表尾。
        R=u;            //后移尾指针，使指向新的尾结点。
    }
}
```

【算法分析】时间复杂度 O(n)。

【思考问题】创建链表也存在往主调函数回传创建好的链表问题，即回传头指针，这里我们使用了 C++的“引用”实现回传。如果用“双重指针”或函数返回值的方法实现回传，怎样实现呢？

（2）头插法建立链表的算法

头插法建表也是链表算法中的一个重要算法，其基本方法是指在创建链表的过程中，将每次所读入的数据装入结点后插入到链表的表头，成为首元素结点。分析如下:

① 头插法初始化链表过程类似尾插法。

② 由于每次新结点都插入到头结点之后，有头指针 L 即可记忆插入位置，故不必象尾插法那样要专门设置一个指针来指示插入位置。

③ 插入每个结点的操作一致。

依此分析得头插法建表算法的基本框架如下：

① 产生链表 L 的头结点，并设置其 next 指针为空。

② 读入元素==>x。

③ 如果 x 是结束符，结束构造。

④ 否则，产生结点并装入 x。

⑤ 将新结点链接到链表的头结点之后。

⑥ 转②。

【算法描述】

```
void   createListH(   node *& L   );        //头插法建表
{
    L=new node;         //产生头结点
    L->next=NULL;     //设置头结点 next 指针为空
    cin>>x;             //读入键盘输入的第一个数值
    while ( x!=End_of_Symbol )          //x 不是结束符时，循环插入新结点
    {                                   // End_of_Symbol  表示结束符
        u=new node;      //产生新结点
        u->data=x;       //元素数据装入新结点
        u->next=L->next;
```

```
            L->next=u;          //将新结点链接到表头，使成为首元素结点
            cin>>x;             //读入键盘输入的下一个数据
        }
    }灌
【算法的一种实现 – 结束符控制创建结束】
void createListH(    node *& L    )
{
    node *u;
    elementType x;     //存放元素数值
    L=new node;        //产生头节点
    L->next=NULL;   //头结点的指针域置为空

    cout<<"请输入元素数据（整数，9999 退出）:"<<endl;     //本例以 9999 为结束符
    cin>>x;      //读入第一个键盘输入数据到变量 x

    while(    x!=9999    )
    {
        u=new node;          //动态申请内存，产生新节点
        u->data=x;           //装入元素数据
        u->next=L->next;   //新结点链接到表头，使成为首元素结点
        L->next=u;
        cin>>x;              //读入下一个键盘输入数据
    }
}
```

【算法分析】时间复杂度 O(1)。

【思考问题】

① 如果采用结点数控制创建结束，如何实现？

② 如果采用“双重指针”，或函数返回值回传创建的链表，如何实现？

2.3.3 链表结构的应用

【例 2.5】设计算法以判断链表 L 中的元素是否是递增的，若递增，则返回 TRUE，否则返回 FALSE。

分析：判断是否递增就是要判断表中每个元素是否小于其直接后继，因而是表的搜索问题的变形。为此，不妨设一个指针 P 依次指向各元素结点，当 P 所指元素小于其直接后继时，即满足 p->data < p->next->data 时，需要继续其后续元素的判断，否则，可直接返回失败标志 FALSE。这样，当整个表搜索结束时，可直接返回成功标志。由此可得到流程图及算法如下：

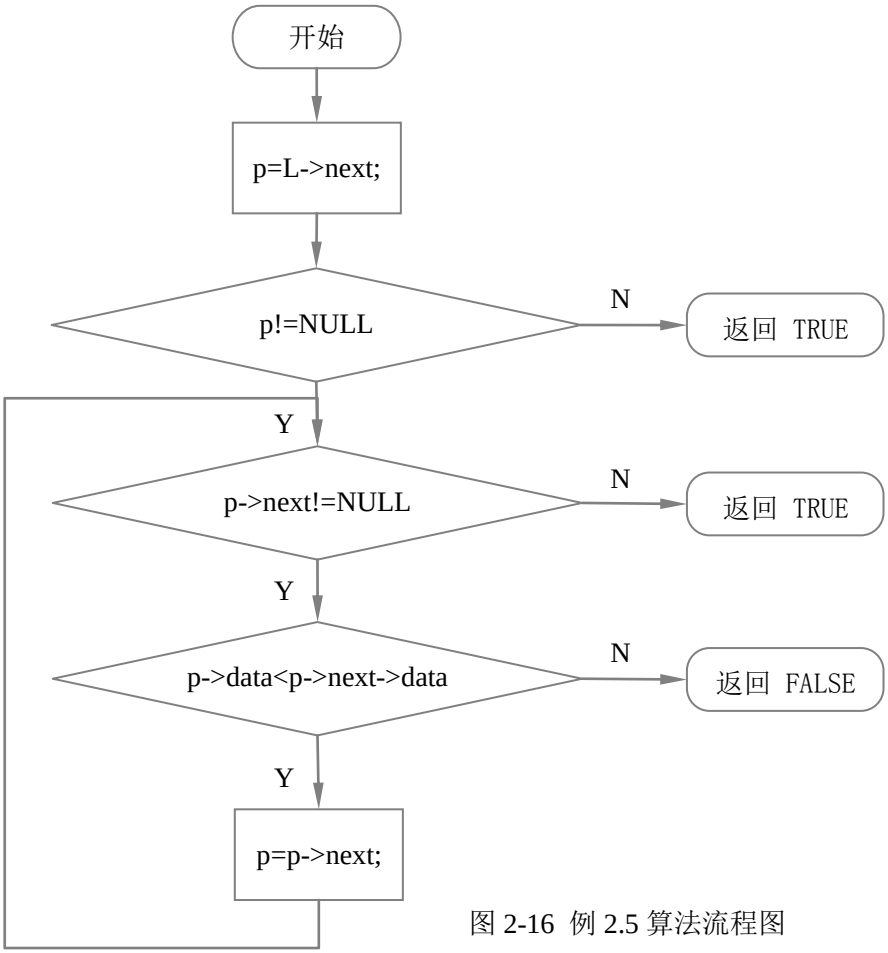

图 2-16 例 2.5 算法流程图

【算法描述】

```
bool Judge(node* L )
{
    node* P=L->next;   //P 指向第一个元素节点
    if(P==NULL)
        return true;     //空表返回 true
    while(P->next!=NULL)
    {
        if(P->data<P->next->data)
            P=P->next;   //当前 2 个相邻结点递增，P 指向下一个节点
        else
            return false;   //后一个元素值大于或等于前驱的值，非递增，返回 false
    }
    return true;   //所有元素都相邻递增，则 L 递增；一个节点情况也在这返回。
```

```
}
```

【例 2.6】假设递增有序的链表 L 表示一个集合，试设计算法在表中插入一个值为 x 的元素结点，使其保持递增有序。

分析：显然，这是插入算法的一个变形——给出了插入要满足的条件，但没有给定插入位置。因此，需要搜索满足这一条件的插入位置的前驱结点的指针 P。有效的插入位置分为两种情况，一种是 x 结点插入中间某位置，必须满足：P->data<x，同时，p->next->data>x；另一种情况是 x 最大，要插入到表的最后，此时，P 指向表尾，而 P->next==NULL。另外，由于给定链表表示一个集合，这意味着不能有重复的元素，在算法中应能判断出来。请读者自己构造算法的流程图。

【算法描述】

```
void setInsert(node* L, elementType x)
{
    node* u;
    node* P=L;
    while(P->next!=NULL && P->next->data<x)   //搜索插入位置 P
    {
        P=P->next;   //P 后移一个结点
    }
    if(P->next==NULL || P->next->data>x)
    {           //P 指向表尾，或 P->next 的元素比 x 大，P->next 即为插入位置
        u=new node;   //产生新节点
        u->data=x;
        u->next=P->next;   //插入新结点
        P->next=u;
    }
}
```

【思考问题】此算法能处理 x 已经在集合中（链表 L 中）的情况吗？

【例 2.7】 设计算法复制链表 A 中的内容到 B 表中。

分析：对 B 表来说，事实上是一个创建链表的问题。与前面创建链表算法不同的只是数据来源不同，这里要将 A 表的数据元素作为 B 表的输入数据，即循环读出 A 表的元素数据，申请新结点，插入 B 表。又题目隐含要求 B 表元素的逻辑顺序与 A 表相同，所以 B 表要采用**尾插法**进行创建。

【算法描述】

```
void listCopy(node* A, node* &B)
{
    node* u;
    node* Pa=A->next;   //Pa 指向 A 的首元素结点
    node* Rb;           //B 的尾指针
```

```
        B=new node;          //初始化 B 表，产生 B 的头结点
        B->next=NULL;
        Rb=B;       //B 表空表，尾指针和头指针相同
        while(Pa!=NULL)
        {           //循环取出 A 中的所有元素，创建新结点插入 B 中
            u=new node;
            u->data=Pa->data;   //复制节点的值
            u->next=NULL;
            Rb->next=u;       //u 插入 B 表尾
            Rb=u;             //Rb 重新指向 B 表尾
            Pa=Pa->next;      //取 A 的下一个元素
        }
    }
```

【例 2.8】已知递增有序链表 A、B 分别表示一个集合，设计算法以实现 C＝A∩B，要求求解结果以相同方式存储，并要求时间尽可能少。

分析：对 C 表来说，这也是一个按尾插法建表算法的一种变形，所不同的是数据来源。因此，对 C 表来说，应包括：

设置头结点；设置尾指针；每找到一个符合条件的元素，即将其插入到表尾。

这些操作的形式相对固定些，因而易于理解。下面重点讨论符合条件的元素的搜索的实现。

由给定条件可知，要插入到 C 表的元素应是在 A、B 两表中都出现的元素。如何求出这些元素？

容易想到的方法是使用两层循环，第一层：依次取 A 的一个元素；第二层：将 A 的这个元素与 B 的元素依次进行比较 — 若 B 中有相同元素，加入 C 中，继续取 A 的下一个元素，直到 A 的元素取完。对 A 的每个元素，B 每次从第一个结点开始比较。时间复杂度：O(|A|*|B|)。这不是性能最好的实现方法。

如果运用所给出的递增有序特点，可以提高算法的时间性能。用两个指针 Pa 和 Pb 分别指向 A 和 B 的结点，初始时分别指向 A、B 的首元素结点，即 Pa=A->next，Pb=B->next，则比较 Pa 和 Pb 当前指向结点的元素大小，会有下述三种情况：

① Pa->data==Pb->data：搜索到公共元素，加入 C 中，即在 C 表表尾插入一个新结点，其值为 Pa->data，或 Pb->next。然后继续取 A 表中下一个元素，即 Pa=Pa->next，同时 Pb 也往后移一个结点，即 Pb=Pb->next。

② Pa->data>Pb->data：表明 A 中当前元素可能在 B 表当前元素的后面，因此要往 B 表的后面搜索，即执行 Pb=Pb->next，继续搜索。

③ Pa->data<Pb->data：A 当前元素值小于 B 当前元素值，说明 A 当前元素不可能在 B 中，取 A 的下一个元素，即：Pa=Pa->next，继续搜索；

循环搜索，直到 A，或者 B 元素取完，即 Pa，Pb 至少有一个为空为止。此后不可能再有交集元素存在，搜索结束。

这样在判定 A 的下一个元素是否交集元素时，B 表不必再从头开始，而只要从上次 Pb 指示的结点继续往后搜索即可。算法完成下来，A 表和 B 表都最多只需要遍历（搜索）一

遍，所以时间复杂度为 O(|A|+|B|)。时间复杂度是两表长之和而不是积，从而要快得多。

由此分析可得流程图如下：

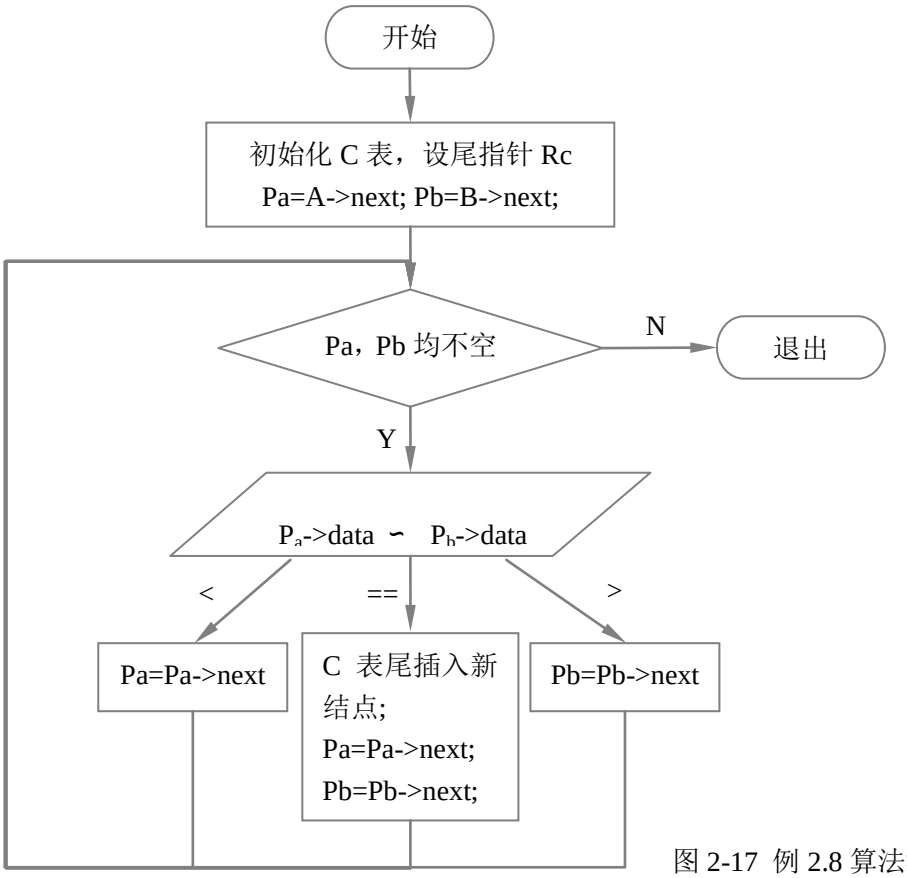

图 2-17 例 2.8 算法流程图

【算法描述】

```
void interSet( node* A, node* B, node* & C )
{
    node* Pa, *Pb, *Rc, *u;
    C=new node;   //生成 C 的头结点
    C->next=NULL;
    Rc=C;              //设置 C 表尾指针
    Pa=A->next;Pb=B->next;   //Pa 和 Pb 分别指向 A 和 B 表的第一个元素节点
    while( Pa!=NULL && Pb!=NULL )   //A 或 B 中一个元素取完，退出循环
    {
        if( Pa->data<Pb->data )   //B 中没有 A 的当前元素，即 Pa->data
            Pa=Pa->next;          //取 A 的下一个元素，回去循环
        else if (Pa->data>Pb->data)   //A 元素值大于 B 元素值，移动 Pb，继续搜索 B
            Pb=Pb->next;
        else   //Pa->data=Pb->data，即为交集元素，在 C 插入新节点，Pa,Pb 同时后移
        {
```

```
                u=new node;
                u->data=Pa->data;   //或 u->data=Pb->data
                u->next=NULL;
                Rc->next=u;      // 尾插法在 C 中插入 u，
                Rc=u;            //修改 C 的尾指针 Rc，指向 u
                Pa=Pa->next;     //Pa 和 Pb 同时后移，分别取 A 和 B 的下一个元素
                Pb=Pb->next;
            }
        }
    }
```

2.4 其它结构形式的链表

前面所讨论的链表结构只是一种基本形式，由于每个结点中仅有一个指针，故称之为单链表。在实际应用中，可能会根据实际问题的特点和需要，对链表结构作必要的修改，从而得到不同结构形式的链表。下面给出一些常见链表结构形式及其运用的变化。

2.4.1 单循环链表

如果将单链表的表尾结点中的后继指针（next）改为指向表头结点，即构成单循环链表，如图 2-18 为带头结点的单循环链表结构的示意图：

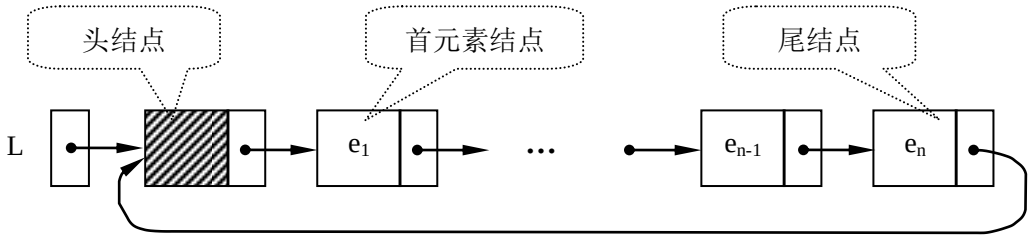

图 2-18 带头结点的单循环链表示意

单循环链表也可以不带头结点。其特点是尾结点的 next 指针指向头结点，整个链表形成一个环，可从任一结点出发搜索到其它各结点。

由于尾结点后继指针的变化，使有关单循环链表的运算的实现必须作相应的调整。首先，链表初始化时要建立循环，即：L->next=L。其次，在遍历链表结点时，单链表中我们常用 p==NULL，或 p->next==NULL 来判定遍历结束，但对循环链表，当 p 指向尾结点时，p->next 指向标有，即 p->next==L，所以单循环链表要用 p==L，或 p->next==L 来判定是否搜索到表尾，否则会造成死循环。第三，在表尾插入和删除结点时要注意保持链表的循环。除此以外，单循环链表的基本操作和单链表基本一致。

2.4.2 带尾指针的单循环链表

在许多情况下，要求能方便地搜索到链表的表头和表尾结点。为此，可采用带尾指针的单循环链表结构。如图 2-19 所示。这类结构也可以不带头结点。

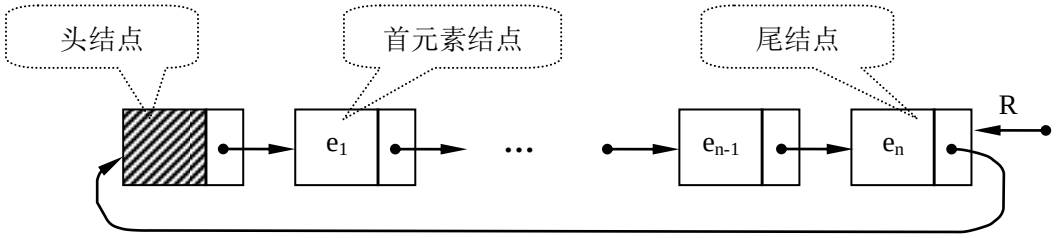

图 2-19 带尾指针的单循环链表示意图

对单循环链表，有了尾指针 R 后，头指针可以用 R->next 表示，可以取消专门的头指针变量 L。与此对应，其基本操作也应作相应调整。初始化链表改为 R=new node，R->next=R。遍历链表时，用指针 p 依次指向各结点，p 初始化指向头结点应为 p=R->next；指向首元素结点应为 p=R->next->next；判定是否指向尾结点应用 p==R。在表尾插入和删除结点时，要记得移动尾指针 R，使其始终指向尾结点；同时注意保持链表的循环特性。

【例 2.9】 已知 A 和 B 分别是如图 2-20 所示的两个带尾指针的单循环链表的尾指针，设计程序段将 A,B 这两个表首尾相接，合并后以 A 表的头结点为头结点，B 表的尾结点为尾结点，要求尽可能节省时间。

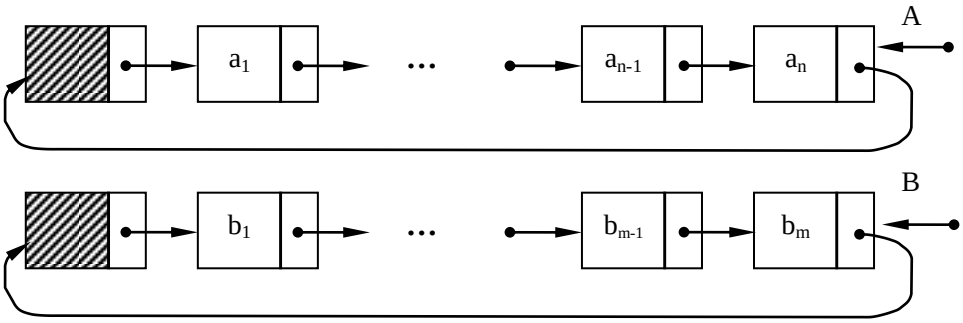

图 2-20 例 2.9 示意图

分析：按题意，由于要求时间性能好，可以利用两表的原有结点，重新链接构成一个合并表，而不能通过申请新结点重新创建一个新表。即，将 B 表的首元素结点链接到 A 表的尾结点上；更新 B 表尾结点的 next 指针，即 B->next，使指向原 A 表的头结点，形成一个大环；释放不需要的原 B 表头结点。

如果不是带尾指针的单循环链表，而是一般的带头结点的单链表或循环链表，则在求解本题时，需要先搜索到 A 表的表尾结点，然后才能将 B 表的首元素结点连接到其后继指针中。然而，本题所给出的带尾指针的单循环链表结构已经给出了尾指针，因而不必搜索。因此，本题只需集中考虑重新链接问题，下面是实现步骤：

① 连接到 A 表表尾的应是 B 表的首元素结点，而不是头结点，故需执行下面语句：

A->next=B->next->next;

然而，这一操作将丢失 A 表头结点指针，故应先保存其指针：u=A->next;

② 由于 B 表头结点已经不需要了，故应执行释放操作。

③ 最后，应将 B 表尾结点的 next 指针指向新的表头，即原 A 表头结点。并让 A 也指向新的表尾。

操作序列如下：

```
u=A->next;                  //保存 A 表头指针
A->next=B->next->next;      //B 表首元素结点链接到 A 表尾结点
delete B->next;             //释放 B 表头结点
B->next=u;                  //B 表尾结点的 next 指向新的表头，形成“大环”
A=B;                        //让 A、B 同时指向新表的尾结点，即新表尾指针
```

合并后的链表如图 2-21 所示：

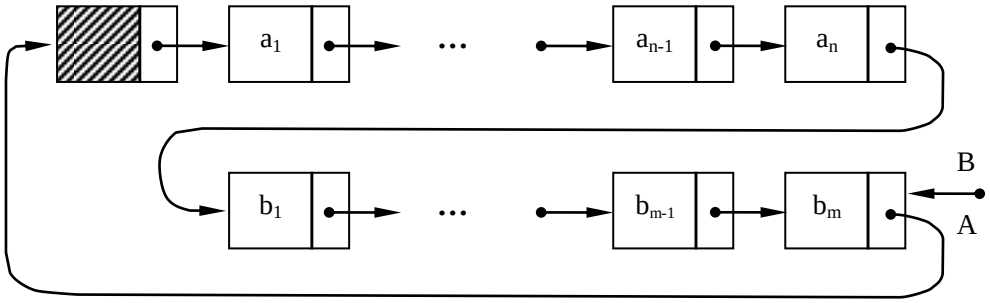

图 2-21 例 2.9 合并后的链表

2.4.3 双链表结构

如果要求能快速地求出任一链表结点的前驱结点，则需要用到双链表结构。双链表中每个结点除了后继指针外，还增加了指向其直接前驱结点的指针。双链表结点结构如图 2-22 所示。

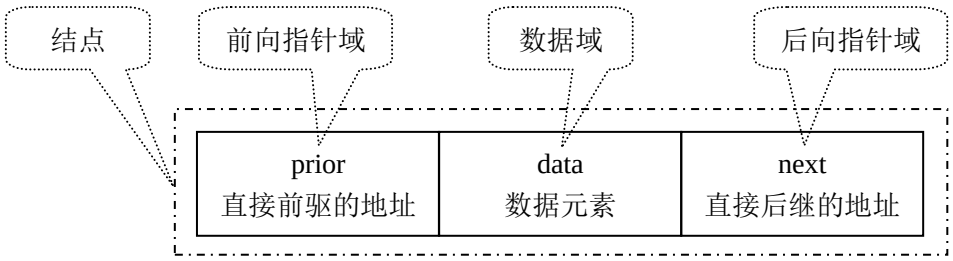

图 2-22 双链表结点结构示意图

双链表也可以带头结点，也可以是循环的。带头结点的双循环链表结构形式如图 2-23 所示：

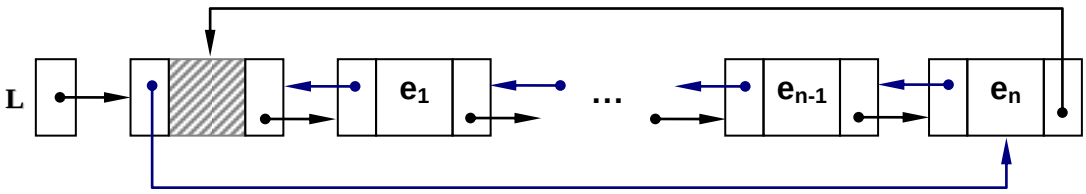

图 2-23 带头结点的双循环链表结点结构示意图

双向链表从任意一个结点开始，既可前向搜索，又可以后向搜索。如果把单链表比作城市中放入单行车道，那么双链表就是双向车道。双链表结点之间的指针关系相对复杂，下面列举几个指针间的关系，如图 2-24 所示：

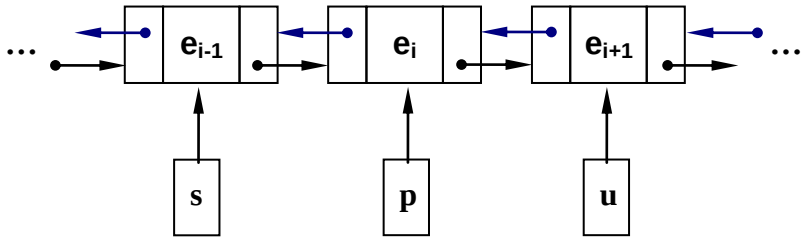

图 2-24 双循环链表结点间指针关系示意图

由图可见，p->next==u，指向 e_{i+1} 结点；p->prior==s，指向 e_{i-1} 结点；p->next->prior==p，即，u->prior==p，指向 e_i 结点；p->prior->next==p，即，s->next==p，指向 e_i 结点。

双链表的优点是访问前驱、后继方便（单链表访问前驱不便）。双链表的缺点是有两个指针域，占用更多内存。

【双链表结点结构描述】

假设双链表中的结点的前驱和后继指针分别为 prior 和 next，数据字段为 data，结点类型可描述如下：

```
typedef struct DLNode
{
    elementType     data;          //数据域
    struct DLNode   *prior;        //前向指针
    struct DLNode   *next;         //后向指针
}dnode, *dLinkedList;
```

由于增加了一个前驱指针，因此，与单链表结构相比，双链表的某些基本运算可能要作一些变化，但象按序号搜索元素、按值查找元素等运算基本上没有多大变化。下面重点讨论在双循环链表中插入和删除结点的操作的实现。

1.双循环链表的初始化

申请头结点，建立双向循环链，即它的 prior 和 next 指针都指向头结点，等于头指针，如图 2-25 所示：

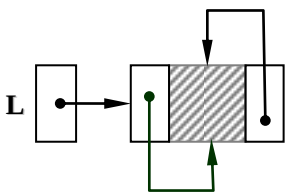

图 2-25 空双循环链表示意图

【算法描述】

```
void initialList(dnode* & L)
```

```
{
    L=new dnode;
    L->prior=L;
    L->next=L;
}
```

2.双循环链表中插入结点

现在讨论在双循环链表中的第 i 个位置上插入一个值为 x 的结点的实现。插入操作首先要搜索到目标位置结点，并用指针 p 指示；然后申请一个新结点，指针为 s；最后要更新四个指针把新结点链接到表中，即新结点的 prior 和 next 指针，p 指向结点前驱的 next 指针，p 指向结点的 prior 指针，插入操作过程如图 2-26 所示，由图可见要完成插入操作需要经过以下步骤：

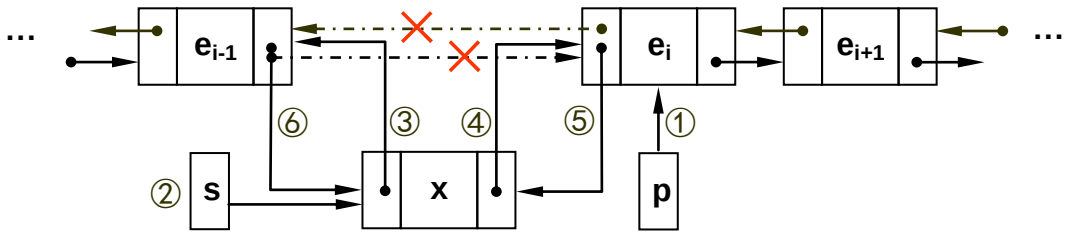

图 2-26 双循环链表插入操作示意图

① 搜索插入位置，获取 a_i 结点的指针 p；
② 申请新结点，装入数据；s=new node; s->data=x;
③ s->prior=p->prior; //新结点前向链接到 a_{i-1} 结点
④ s->next=p; //新结点后向链接到 a_i 结点
⑤ p->prior =s; // a_i 结点前向链接到新结点
⑥ s->prior->next =s; //a_{i-1} 结点后向链接到新结点

在这一序列中，许多初学者容易将其中⑥的语句错误地写成 p->prior->next=s，这是由于没有注意到指针的动态变化所造成的，因为执行⑤语句后，p->prior 已经为 s。

还有，这个操作序列不是唯一的。比如，保持其它步骤不便，上述的⑤、⑥两步可以改变为：

⑤ p->prior->next =s;
⑥ p->prior =s;

【算法描述】

```
bool listInsert(dnode* L, int i, elementType x )
{
    dnode* p=L->next;   //p 指向首元素结点
    dnode* S;
    int k=1;
    while(k!=i && p!=L)   //搜索 ai 结点，获取其指针 p。当 p=L 即又回到头结点
```

```
    {
        p=p->next;   //p 移到下一个结点
        k++;
    }
    if(p==L && k!=i)   //当 p==L 且 K==i 时，插入位置仍然合法，结点要插在最后
        return false;    //p 指向头结点，说明插入位置 i 不对，返回 false
    else
    {           //此时，k==i，p 为 ai 结点的指针
                //或者 p==L 且 k==i，新结点要插入最后
        S=new dnode;          //申请新结点
        S->data=x;            //装入数据
        S->prior=p->prior;    //S 的前驱为 ai-1
        S->next=p;            //S 后继为 p
        p->prior=S;           //p 的前驱变为 S
        S->prior->next=S;     //ai-1 的后继为 S
        return true;          //正确插入返回 true
    }
}
```

3.双循环链表中删除结点

下面讨论删除双循环链表中的第 i 个元素结点的实现。删除操作首先要搜索到目标结点 a_i，取得其指针 p；然后，修改 p 指向结点前驱的 next 指针和后继的 prior 指针；最后释放目标结点即可。操作过程如图 2-27 所示，具体步骤如下：

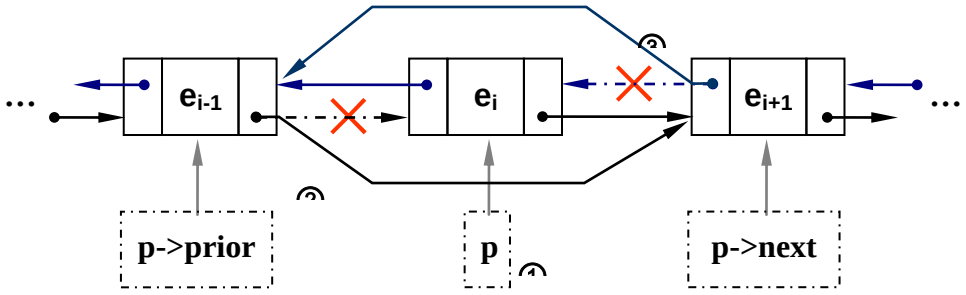

图 2-27 双循环链表删除操作示意图

① 搜索 a_i 结点，指针 p；则 a_{i-1} 结点指针 p->prior；a_{i+1} 结点指针 p->next；
② (p->prior)->next = p->next ; //a_{i-1} 结点的 next 指向 a_{i+1} 结点
③ (p->next)->prior = p->prior; //a_{i+1} 结点的 prior 指向 a_{i-1} 结点
④ delete p; // free(p)，释放结点 p

【算法描述】

```
bool listDelete(dnode* L,int i)
{
```

```
        dnode* p=L->next;     //p 初始化指向首元素结点
        int k=1;
        while(k!=i && p!=L)   //搜索 ai 结点，p=L 说明又回到头结点
        {
            p=p->next;
            k++;
        }
        if(p==L)
            return false;   //删除位置 i 超出范围，删除失败，返回 false
        else
        {                       //此时，p 指向 ai 结点
            p->next->prior=p->prior;     //p 的后继 ai+1 的 prior 指向 p 的前驱 ai-1
            p->prior->next=p->next;      //p 的前驱 ai-1 的 next 指向 p 的后继 ai+1
            delete p;         //释放结点
            return true;      //成功删除，返回 true
        }
    }
```

本章小结

线性表是本课程后续其它数据结构的重要基础。线性表也是软件设计中最常用的数据结构，是实际应用领域中许多具体数据的抽象表示形式，在实际应用时可赋予不同的实际含义。

每种数据结构都涉及到**逻辑结构、运算定义、存储结构及其上的运算的实现**几个方面。

线性表是有限个元素的序列，元素之间逻辑上是线性关系，其特点是每个元素最多有一个前驱和一个后继。

对线性表结构可概括出六个基本运算，许多复杂的运算可通过调用这六个基本算法来实现。

线性表的存储实现有顺序存储和链式存储两类。

采用顺序存储方式存储线性表，得到**顺序表**结构。在顺序表结构中，**逻辑上相邻的元素的存储地址也相邻**。通过对在顺序表结构上实现所给出的基本运算及算法分析可知，在顺序表结构上插入和删除运算时，需要移动较多的元素（在等概率情况下，插入或删除一个元素平均需要移动一半的元素），且需要按最大空间需求预分配存储空间，因而提出了线性表的链式存储结构，即链表结构。

在**链表结构**中，逻辑上相邻的元素的存储位置不一定相邻，**元素之间的逻辑次序是通过指针（链）来描述的**。链表有静态链表和动态链表两种实现方式，本章仅讨论了采用指针和动态变量实现的动态链表。动态链表可以在程序执行期间动态按需申请内存，用完释放。

为了运算及描述的一致性，在链表中设置了一个**头结点**，从而得到**带头结点的单链表结构**形式。在（带头结点的）单链表上实现各种基本运算是本章的重要内容，许多更为复杂的运算都是建立在对这些基本运算的真正理解的基础之上的；

在有些情况下，为了实现特定问题的求解，需要将链表的首尾结点相连接，即让尾结点的后继指针指示表头结点，从而得到**循环链表结构**。对循环链表的有关运算的实现与前面链表结构上的运算实现基本类似，所不同的是，循环链表中尾结点的判断，因此要求算法中能注意到这一点，以免出现“死循环”。对某些特定问题，例如要实现链表的首尾相接，需要能比较方便地知道链表的首尾结点，为此可采用**带尾指针的单循环链表**形式。

如果需要能方便地指示每个结点的前驱结点的话，可在每个结点中再增设一个指示其前驱的指针，从而得到**双（向）链表结构**。双链表也可以带头结点，也可以设置为循环形式。在双链表上的大多数运算的实现与单链表上的运算实现有许多相似之处，明显不同的是插入和删除结点的运算。

在各种链表结构上搜索结点及插入和删除运算是最基本的运算要求，因为这是复杂运算的基础；理解线性表的各种存储结构及其特点是基本的学习要求；根据实际问题的需要设计合理的数据结构、有效的算法是重点也是难点。