第 2 章　线性表

线性表是最基本、最简单，也是最常用的一种数据结构。线性表中数据元素之间是一对一的关系，即除了第一个和最后一个数据元素之外，其他数据元素都是首尾相接的。线性表的逻辑结构简单，便于实现和操作。通过本章的学习，读者应该掌握以下内容:

- 线性表的逻辑结构;
- 线性表的顺序存储及运算实现;
- 线性表的链式存储及运算实现;
- 顺序表和链表的比较。

2.1　线性表的逻辑结构

2.1.1　线性表的定义

线性表是由 n（n≥0）个类型相同的数据元素组成的有限序列，通常表示为下列形式：

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

其中：L 为线性表名称，a_i 为组成该线性表的数据元素。

线性表中数据元素的个数称为线性表的长度，当 n=0 时，线性表为空，又称为空线性表。

设 a_i 是表中第 i 个元素，其中 i=1，…，n-1，称 a_i 是 a_{i+1} 的**直接前驱元素**，a_{i+1} 是 a_i 的**直接后继元素**。在线性表中，除 a_1 无直接前驱外，其余元素有且仅有一个直接前驱；除 a_n 无直接后继外，其余元素有且仅有一个直接后继。在不引起混淆的情况下，简称直接前驱为“**前驱**”，简称直接后继为“**后继**”。

数据元素的含义广泛，在不同的具体情况下，可以有不同的含义。

例如，英文字母表(A, B, C,…, Z)是一个长度为 26 的线性表，其中每个数据元素为一个字母。

再如，某公司 2000 年每月产值表(400,420,500,…,600,650)（单位：万元）是一个长度为 12 的线性表，其中每个数据元素为整数。

上述两例中的每一个数据元素都是不可分割的，在一些复杂的线性表中，每一个数据元素又可以由若干个数据项组成，在这种情况下，通常将数据元素称为记录（record）。例如，表 2-1 所示的某学校的学生成绩表就是一个线性表，表中每一个学生的成绩就是一个记录，每个记录包含 6 个数据项：学号、姓名、数据结构……

表 2-1 学生成绩表

学号	姓名	数据结构	大学物理	高等数学	平均成绩
0232101	王刚	95	90	85	90
0232102	李娟	90	80	85	85
0232103	赵平	99	95	91	95
0232104	王强	86	70	84	80
0232105	张雪	92	91	84	89

矩阵也是一个线性表，但它是一个比较复杂的线性表。在矩阵中，可以把每行看成是一个数据元素，也可以把每列看成是一个数据元素，而其中的每一个数据元素又是一个线性表。

线性表是一个相当灵活的数据结构，它的长度可根据需要增长或缩短，即对线性表的数据元素不仅可以进行访问，还可以进行插入和删除等。

2.1.2 线性表的 ADT 定义

线性表的抽象数据类型定义如下：

ADT List{

数据元素：$D=\{a_i|a_i\in D_0,i=1,2,\cdots,n，n\geq 0，D_0$ 为某一数据对象$\}$

关系：$R1=\{<a_{i-1},a_i>|a_i,a_{i-1}\in D，i=2,\cdots,n\}$

基本操作：

（1）InitList(*L)。

操作前提：L 为未初始化线性表。

操作结果：将 L 初始化为空表。

（2）DestroyList(*L)。

操作前提：线性表 L 已存在。

操作结果：将 L 销毁。

（3）ClearList(*L)。

操作前提：线性表 L 已存在。

操作结果：将表 L 置为空表。

（4）EmptyList(L)。

操作前提：线性表 L 已存在。

操作结果：如果 L 为空表则返回真，否则返回假。

（5）ListLength(L)。

操作前提：线性表 L 已存在。

操作结果：如果 L 为空表则返回 0，否则返回表中的数据元素个数。

（6）Locate(L,e)。

操作前提：表 L 已存在，e 是给定的一个数据元素。

操作结果：返回线性表 L 中第一个与 e 相等的数据元素的位序，若这样的数据元素不存在，则返回 0。

（7）GetData(L,i,*e)。

操作前提：表 L 存在，且 i 值合法，即 1≤i≤ListLength(L)

操作结果：用 e 返回线性表 L 中第 i 个数据元素的值。

（8）ListInsert (*L,i,e)。

操作前提：表 L 已存在，e 为合法数据元素值，且 1≤i≤ListLength(L)+1。

操作结果：在 L 中第 i 个位置上插入新的数据元素 e，L 的长度加 1。

（9）ListDelete(*L,i,*e)。

操作前提：表 L 已存在且非空，1≤i≤ListLength(L)。

操作结果：删除 L 的第 i 个数据元素，并用 e 返回其值，L 的长度减 1。

}ADT　List

上面我们定义了线性表的逻辑结构和基本操作。在计算机内，线性表有两种基本的存储结构：顺序存储结构和链式存储结构。下面分别讨论两种存储结构以及对应存储结构下实现各操作的算法。

2.2　线性表的顺序存储和实现

2.2.1　线性表顺序存储结构

在线性表的顺序存储结构中，其前后两个元素在存储空间中是紧邻的，且前驱元素一定存储在后继元素的前面。由于线性表的所有数据元素属于同一数据类型，所以每个元素在存储器中占用的空间大小相同，因此要在该线性表中查找某一个元素是很方便的。

假设线性表中的第一个数据元素的存储地址为 $Loc(a_1)$，每一个数据元素占 L 字节，则线性表中第 i 个元素 a_i 在计算机存储空间中的存储地址为：

$$Loc(a_i)=Loc(a_1)+(i-1)*L$$

线性表的顺序存储结构的特点是：线性表中逻辑上相邻的结点在存储结构中也相邻，如图 2-1 所示。只要确定了线性表存储的起始位置，就可以随机存取表中的任一数据元素。因此，线性表的顺序存储结构是一种随机存取的存储结构。

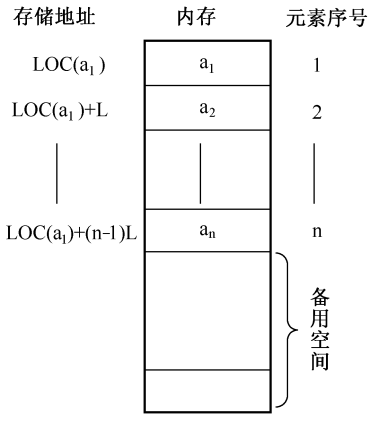


图 2-1　线性表的顺序存储结构示意图

由于程序语言中的向量（一维数组）也是采用顺序存储表示，故可以用向量这种数组类型来描述线性表的顺序存储。顺序表示的线性表也称为**顺序表**。

2.2.2 线性表在顺序存储结构下的运算

可以用 C 语言描述顺序表如下：

```
//线性表的顺序存储结构
#define ERROR 0
#define OK    1
#define   LIST_INIT_SIZE   100    //存储空间的初始分配量
typedef   struct{
     ElemType   elem[LIST_INIT_SIZE];    //存储空间基址
     int   length;                       //当前长度
}SqList;
```

在顺序表中，线性表的有些运算很容易实现。例如，设 L 是指向某一顺序表的指针，则表的初始化操作是将表的长度置 0，即 L->length=0；求表长和取表中第 i 个结点的操作只需分别返回 L->length 和 L->data[i-1]即可。以下主要讨论插入和删除两种运算。

1. 顺序表的插入操作

线性表的插入运算是指在表的第 i（1≤i≤n＋1）个位置上，插入一个新结点 e，使长度为 n 的线性表：

$(a_1,\cdots,a_{i-1},a_i,\cdots,a_n)$

变为长度为 n+1 的线性表：

$(a_1,\cdots,a_{i-1},e,a_i,\cdots,a_n)$

用顺序表作为线性表的存储结构时，由于结点的物理顺序必须和结点的逻辑顺序保持一致，因此必须将表中位置 n，n-1，…，i 上的结点，依次后移到位置 n+l，n，…，i+l 上，空出第 i 个位置，然后在该位置上插入新结点 e。仅当插入位置 i=n+l 时，才无需移动结点，直接将 e 插入表的末尾。

其插入过程见图 2-2，具体算法描述如下：

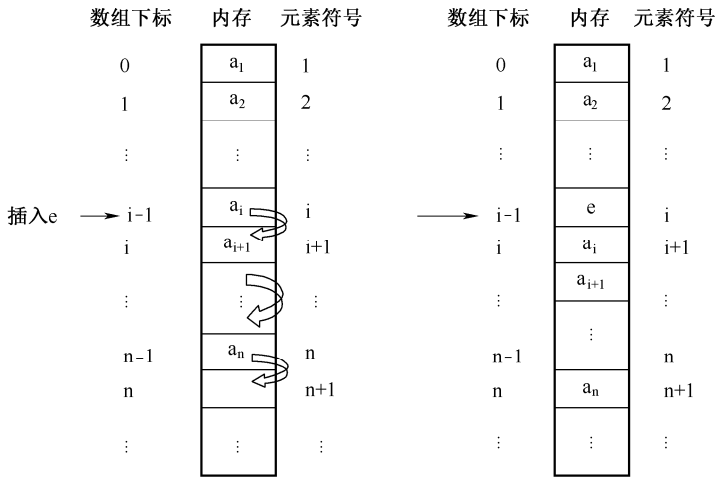


图 2-2 顺序表中插入结点的过程

【算法 2-1】

```
int ListInsert_Sq(SqList *L,int i,ElemType e)
{
    int j;
    if(i<1||i>L->length+1)              //非法位置，退出运行
        return ERROR;
    if(L->length>=LIST_INIT_SIZE)       //当前存储已满，退出运行
        return ERROR;
    for(j=L->length-1;j>=i-1;j--)
        L->elem[j+1]=L->elem[j];
    L->elem[i-1]=e;
    L->length++;
    return   OK;
}
```

现在分析算法 2-1 的时间复杂度。该问题的规模是表的长度 L->lengh，设它的值为 n。显然该算法的时间主要花费在 for 循环中的结点后移语句上，该语句的执行次数（即移动结点的次数）是 n-i+l。由此可看出，所需移动结点的次数不仅依赖于表的长度 n，而且还与插入位置 i 有关。当 i=n+l 时，由于循环变量的终值大于初值，结点后移语句将不执行，无需移动结点；若 i=1，则结点后移语句将循环执行 n 次，需移动表中所有结点。也就是说该算法在最好情况下的时间复杂度是 O(1)；在最坏时间下的复杂度是 O(n)。由于插入可能在表中任意位置上进行，因此需分析算法的平均性能。

在长度为 n 的线性表中插入一个结点，令 $E_{IS}(n)$表示移动结点次数的期望值（即移动结点的平均次数），在表中第 i 个位置上插入一个结点的移动次数为 n-i+l，故

$$E_{IS}(n)=\sum_{i=1}^{n+1}p_i(n-i+1)$$

式中，p_i 表示在表中第 i 个位置上插入一个结点的概率。不失一般性，假设在表中任意合法位置（1≤i≤n+1）上插入结点的机会是均等的，则

$$p_1=p_2=\cdots=p_{n+1}=1/(n+1)$$

因此，在等概率插入的情况下：

$$E_{IS}(n)=1/(n+1)\sum_{i=1}^{n+1}(n-i+1)=n/2$$

也就是说，在顺序表上做插入运算，平均要移动表中的一半结点。当表长 n 较大时，算法的效率相当低。虽然 $E_{IS}(n)$中 n 的系数较小，但就数量级而言，它仍然是线性阶的，因此算法的平均时间复杂度是 O(n)。

2. 顺序表的删除操作

线性表的删除运算是指将表的第 i（1≤i≤n）个结点删去，使长度为 n 的线性表$(a_1,\cdots,a_{i-1},a_i,\cdots,a_n)$，变成长度为 n-1 的线性表$(a_1,\cdots,a_{i-1},a_{i+1},\cdots,a_n)$。

和插入运算类似，在顺序表上实现删除运算也必须移动结点，才能反映出结点间逻辑关系的变化。若 i=n，则只要简单地删除终端结点，无需移动结点；若 1≤i≤n-1，则必须将表中位置 i+1，i+2，…，n 上的结点，依次前移到位置 i，i+1，…，n-1 上，以填补删除操作造成的空缺。其删除过程见图 2-3。

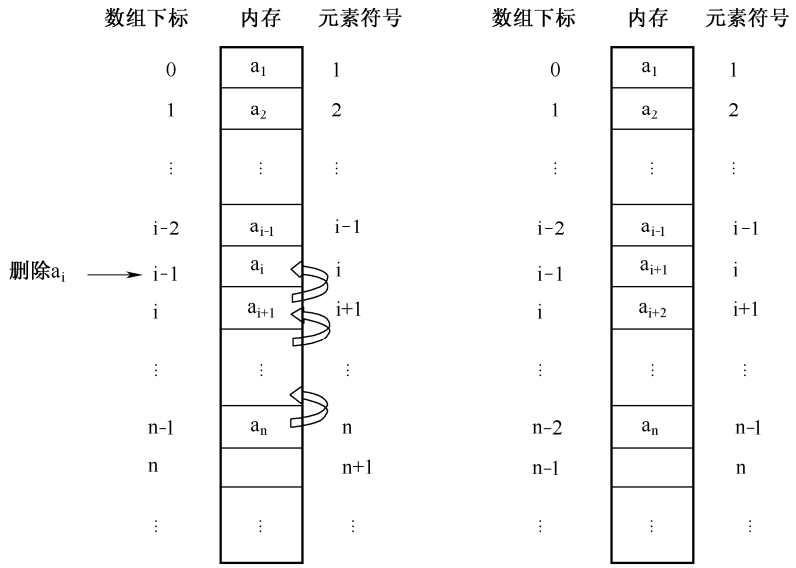


图 2-3　顺序表中删除结点的过程

【算法 2-2】

```
int ListDelete_Sq(SqList *L,int i,ElemType *e)
{
    int j;
    if(i<1||i>L->length)
        return ERROR;                  //非法位置
    for(j=i;j<=L->length-1;j++)
        L->elem[j-1]=L->elem[j];       //结点前移
    *e=L->elem[i];                     //被删除元素的值赋给 e
    L->length--;
    return OK;
}
```

算法 2-2 的时间分析与算法 2-1 类似，结点的移动次数也是由表长 n 和位置 i 决定的。若 i=n，则由于循环变量的初值大于终值，前移语句将不执行，不需要移动结点；若 i=1 则前移语句将循环执行 n-1 次，需移动表中除开始结点外的所有结点。这两种情况下算法的时间复杂度分别是 O(1)和 O(n)。

删除算法的平均性能分析与插入算法相似。在长度为 n 的线性表中删除一个结点，令 $E_{DE}(n)$ 表示所需移动结点的平均次数，删除表中第 i 个结点的移动次数为 n-i，故：

$$E_{DE}(n)=\sum_{i=1}^{n+1}p_i(n-i)$$

式中，p_i 表示删除表中第 i 个结点的概率，在等概率的假设下：

$$p_1=p_2=\cdots=p_n=1/n$$

由此可得：

$$E_{DE}(n)=1/n\sum_{i=1}^{n+1}(n-i)=(n-1)/2$$

即在顺序表上做删除运算，平均要移动表中约一半的结点，平均时间复杂度为 O(n)。

3. *顺序表存储结构的特点*

线性表的顺序存储结构中任意数据元素的存储地址可由公式直接导出，因此顺序存储结构的线性表可以随机存取其中的任意元素。

但是，顺序存储结构也有一些不方便之处，主要表现在：

（1）数据元素最大个数需要预先确定，使得高级程序设计语言编译系统需要预先分配相应的存储空间。

（2）插入与删除运算的效率很低。为了保持线性表中的数据元素的顺序，在插入操作和删除操作时需要移动大量数据。对于插入或删除操作很频繁的线性表来说，若线性表的数据元素占字节较多，这些操作将影响系统的运行速度。

（3）顺序存储结构的线性表的存储空间不便于扩充。当一个线性表分配顺序存储空间后，如果线性表的存储空间已满，但还需要插入新的元素，则会发生“上溢”错误。在这种情况下，如果在原线性表的存储空间后找不到与之连续的可用空间，则会导致运算的失败或中断。

2.3　线性表的链式存储和实现

从上一节的讨论中可见，线性表的顺序存储结构的特点是，逻辑关系上相邻的两个元素在物理位置上也相邻，因此可以随机存取表中任一元素，它的存储位置可用一个简单、直观的公式来表示。然而，从另一方面来看，这个特点也造成了这种存储结构的弱点：其一，在做插入或删除操作时，需移动大量元素；其二，在给长度变化较大的线性表预先分配空间时，必须按最大空间分配，使存储空间不能得到充分利用；其三，表的容量难以根据实际需要扩充。本节将讨论线性表的另一种表示方法——链式存储结构，由于它不要求逻辑上相邻的元素在物理位置上也相邻，因此它没有顺序存储结构所具有的弱点，但同时也失去了顺序表可随机存取的优点。

2.3.1　线性链表

线性表的链式存储结构特点是，用一组任意的存储单元来存放线性表的结点，这组存储单元既可以是连续的，也可以是不连续的，甚至是零散分布在内存中的任意位置上。链表中结点的逻辑次序和物理次序不一定相同。为了能正确表示结点间的逻辑关系，在存储每个结点值的同时，还必须存储指示其后继结点的地址（或位置）信息，这个信息称为**指针**（pointer）或**链**（link）。这两部分信息组成了链表中的结点结构，如图 2-4 所示。

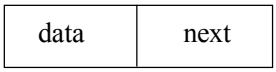
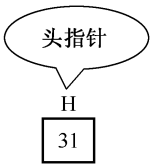


图 2-4　线性链表中的结点结构

图 2-4 中 data 域是数据域，用来存放结点的值；next 域是指针域（亦称链域），用来存放结点的直接后继的地址（或位置）。链表正是通过每个结点的链域将线性表的 n 个结点按其逻辑顺序链接在一起。由于上述链表的每个结点只有一个链域，故将这种链表称为**单链表**（single linked list）。

显然，单链表中每个结点的存储地址是存放在其前驱结点 next 域中，而开始结点无前驱，故应设头指针 head 指向开始结点。同时，由于终端结点无后继，故终端结点的指针域为“空”，即 NULL（图示中也可用∧表示）。例如，图 2-5 是线性表(ZHAO,QIAN,SUN,LI,ZHOU,WU,ZHENG,WANG)的单链表示意图。



存储地址	数据域	指针域
1	LI	43
7	QIAN	13
13	SUN	1
19	WANG	NULL
25	WU	37
31	ZHAO	7
37	ZHENG	19
43	ZHEOU	25

图 2-5　单链表示意图

由于单链表只注重结点间的逻辑顺序，并不关心每个结点的实际存储位置，因此通常用箭头来表示链域中的指针，于是链表就可以直观地画成用箭头链接起来的结点序列。例如图 2-5 可以画成图 2-6 的形式。

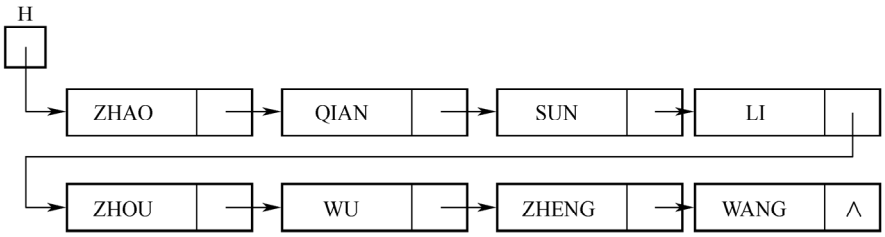


图 2-6　单链表的一般图示法

由上述可见，单链表可由头指针唯一确定，在 C 语言中可用“结构指针”来描述。

```
typedef struct LNode{
    ElemType data;
    struct LNode *next;
}LNode,*LinkList;
LinkList L,p;
```

假设 L 是 LinkList 型的变量，则 L 为单链表的头指针，它指向表中的第一个结点。若 L 为“空”（L==NULL），则表示的线性表为“空”表，其长度 n 为“零”。有时，在单链表的第一个结点之前附设一个结点，称之为**头结点**。头结点的数据域可以不存储任何信息，也可存储如线性表的长度等类的附加信息，头结点的指针域存储指向第一个结点的指针（即第一个元素结点的存储位置）。如图 2-7（a）所示，单链表的头指针指向头结点。若线性表为空表，则头结点的指针域为“空”，如图 2-7（b）所示。

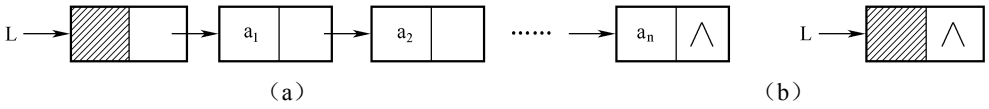


图 2-7　带头结点的单链表

在这里我们一定要严格区分指针变量和结点变量这两个概念。例如，以上定义的变量 p 是类型为 LNode *的指针变量，若 p 的值非空（p!=NULL），则它的值是类型为 LNode 的某一个结点的首地址。通常 p 所指的结点变量并非在变量说明部分明显地定义，而是在程序执行过程中，当需要时才产生，故称为动态变量。实际上，它是通过标准函数生成的，即：

```
p=(LNode *)malloc(sizeof(LNode));
```

或

```
p=(LinkList)malloc(sizeof(LNode));
```

函数 malloc 分配一个类型为 LNode 的结点变量的空间，并将其首地址放入指针变量 p 中。一旦 p 所指的结点变量不再需要，又可通过标准函数：

```
free(p);
```

释放 p 所指的结点变量空间。

由于无法通过预先定义的标识符去访问这种动态的结点变量，而只能通过指针 p 来访问它，即用*p 作为该结点变量的名称来访问。p 和*p 之间的关系见图 2-8。根据所学的 C 语言程序设计的知识，可以得出以下关系：

- (*p)表示 p 所指向的结点；
- (*p).data⇔p->data 表示 p 指向结点的数据域；
- (*p).next⇔p->next 表示 p 指向结点的指针域。

图 2-8　指针 p 和结点*p 的关系

1. 单链表的建立

假设线性表中结点的数据类型是字符，逐个输入这些字符的结点，并以换行符'\n'为输入结束符。动态建立带头结点的单链表的常用方法有如下两种。

（1）头插法建表。

从一个带头结点的空表开始，重复读入数据，生成新结点，将读入数据存放到新结点的数据域中，然后将新结点插入到当前链表的表头结点之后，直至读入结束标志为止。头插法建单链表是将链表右端看成固定，链表不断向左延伸得到的。头插法最先得到的是尾结点。图 2-9 表示在空链表 head 中依次插入 d、c、b 后，将 a 插入到当前链表表头时指针的修改情况。

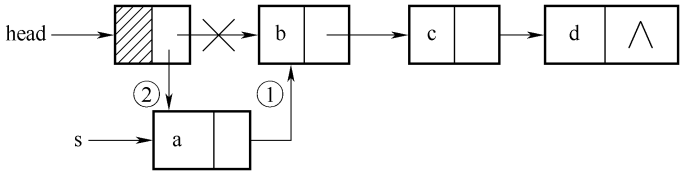


图 2-9　结点*s 插入到单链表 head 的头上

【算法 2-3】

```
LinkList CreateListH(void)
{
    char ch;
    LinkList head=(LinkList)malloc(sizeof(LNode));
    LinkList s;
    head->next=NULL;
    ch=getchar();
    while(ch!='\n')
    {
        s=(LinkList)malloc(sizeof(LNode));
        s->data=ch;
        s->next=head->next;//对应图 2-9 的①
        head->next=s;      //对应图 2-9 的②
        ch=getchar();
    }
    return head;
}
```

（2）尾插法建表。

头插法建立链表虽然算法简单，但生成的链表中结点的次序和输入的顺序相反。若希望二者次序一致，可采用尾插法建表。该方法是将新结点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。例如，在空链表 head 中插入 a、b、c 后，将 d 插入到当前链表的表尾，其指针修改情况如图 2-10 所示。

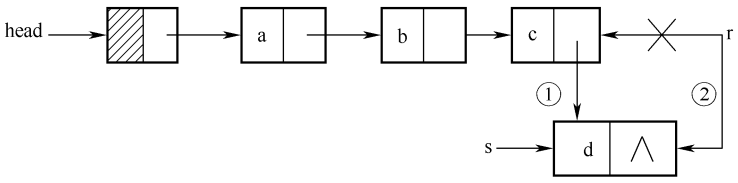


图 2-10　结点*s 插入到单链表 head 的尾上

【算法 2-4】

```
LinkList CreateListT(void)
{
    char ch;
    LinkList head=(LinkList)malloc(sizeof(LNode));
    LinkList s,r;
    r=head;
    ch=getchar();
    while(ch!='\n')
    {
        s=(LinkList)malloc(sizeof(LNode));
        s->data=ch;
        r->next=s;          //对应图 2-10 的①
        r=s;                //对应图 2-10 的②
```

```
        ch=getchar();
    }
    r->next=NULL;
    return head;
}
```

2. 查找

（1）按序号查找。

在单链表中，由于每个结点的存储位置都放在其前一结点的 next 域中，所以即使知道被访问结点的序号 i，也不能像顺序表那样直接按序号 i 访问一维数组中的相应元素，实现随机存取，只能从链表的头指针出发，顺链域 next 逐个结点往下搜索，直至搜索到第 i 个结点为止。

算法描述：设带头结点的单链表的长度为 n，要查找表中第 i 个结点，则需要从单链表的头指针 head 出发，从第一个结点（head->next）开始顺着链域扫描，用指针 p 指向当前扫描到的结点，初值指向第一个结点（head->next），用 j 做记数器，累计当前扫描过的结点数（初值为 0），当 j=i 时，指针 p 所指的结点就是要找的第 i 个结点。

【算法 2-5】

```
LinkList GetNode(LinkList head,int i)  //在带头结点的单链表 head 中查找第 i 个结点，若找到（1≤i≤n），则返回该结点的存储位置，否则返回 NULL
{
    int j;
    LinkList p;
    p=head;j=0;                 //从头结点开始扫描
    while( p->next && j<i)
    {
        p=p->next;              //扫描下一结点
        j++;                    //已扫描结点计数器
    }
    if(i==j)
        return p;               //找到了第 i 个结点
    else
        return NULL;            //找不到，i≤0 或 i>n
}
```

算法中，while 语句的终止条件是搜索到表尾或者满足 j≥i，其频度最多为 i，它和被搜索的位置有关。在等概率假设下，平均时间复杂度为：

$$\sum_{i=0}^{n} i/(n+1) = 1/(n+1) \times \sum_{i=1}^{n} i = n/2 = O(n)$$

（2）按值查找。

算法描述：按值查找是指在单链表中查找是否有结点值等于 e 的结点，若有的话，则返回首次找到其值为 e 的结点的存储位置，否则返回 NULL。查找过程从单链表的头指针指向的第一个结点出发，顺着链逐个将结点的值与给定值 e 做比较。

【算法 2-6】

```
LinkList LocateNode(LinkList head,ElemType e)  //在带头结点的单链表 head 中查找其结点值等于 e 的结点，若找到则返回该结点的位置，否则返回 NULL
```

```
{
    LinkList p;
    p=head->next;                  //从开始结点比较，p 初始值指向第一个结点
    while(p && p->data!=e)         //直到 p 为 NULL 或 p->data 等于 e 停止
        p=p->next;
    return p;                      //若 p=NULL，则查找失败
}
```

该算法的执行时间与查找值 e 有关，其平均时间复杂度分析类似于按序号查找，也为 O(n)。

3. 插入

算法描述：要在带头结点的单链表 head 中的第 i 个数据元素前插入一个值为 e 的新结点，首先在单链表中找到第 i-1 个结点并由指针 p 指示，然后申请一个新的结点并由指针 s 指示，其数据域的值为 e，并修改第 i-1 个结点的指针使其指向 s，然后使 s 结点的指针域指向第 i 个结点。插入结点的过程如图 2-11 所示。

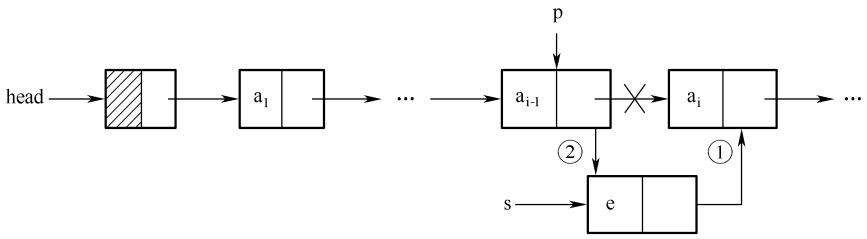


图 2-11　在单链表第 i 个结点前插入一个结点的过程

【算法 2-7】

```
int InsertList(LinkList head,int i,ElemType e)   //在带头结点的单链表 L 中的第 i 个位置插入值为 e 的新结点
{
    LinkList p,s;
    p=GetNode(head,i-1);                  //寻找第 i-1 个结点，并返回指针
    if(p==NULL)
        return ERROR;                     //插入位置不合理
    s=(LinkList)malloc(sizeof(LNode));
    s->data=e;
    s->next=p->next;                      //对应图 2-11 中的①
    p->next=s;                            //对应图 2-11 中的②
    return OK;
}
```

当单链表中有 n 个结点时，则前插操作的插入位置有 n+1 个，即 1≤i≤n+1。当 i=n+1 时，则认为是在单链表的尾部插入一个结点。因此，用 i-1 做实参调用 GetNode 时可完成插入位置的合法性检查。算法的时间复杂度主要耗费在查找操作 GetNode()函数上，故时间复杂度也为 O(n)。

4. 删除

算法描述：欲在带头结点的单链表 head 中删除第 i 个结点，则首先要找到第 i-1 个结点并使 p 指向第 i-1 个结点，而后删除第 i 个结点并释放结点空间，删除过程如图 2-12 所示。

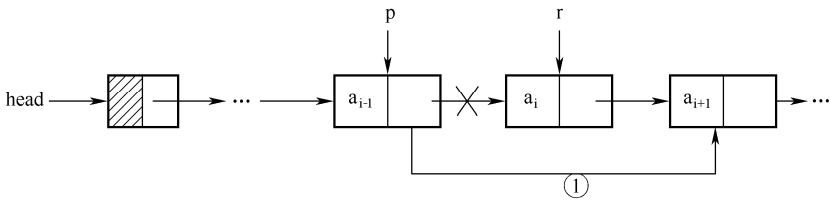


图 2-12　在单链表中将第 i 个结点删除的过程

【算法 2-8】

```
int DeleteList(LinkList head,ElemType *e,int i)  //在带头结点的单链表 L 中删除第 i 个元素，并将删除的元素保存到变量*e 中
{
    LinkList p,r;
    p=GetNode(head,i-1);  //找第 i-1 个结点，并返回指针
    if(p==NULL || p->next==NULL)  //i<1 或 i>n 时删除位置有错
        return ERROR;
    r=p->next;
    p->next=r->next;
    *e=r->data;
    free(r);
    return OK;
}
```

设单链表的长度为 n，则删去第 i 个结点仅当 1≤i≤n 时是合法的。算法的时间复杂度主要耗费在查找操作 GetNode()函数上，故时间复杂度也为 O(n)。

【例 2-1】编写一个算法，求单链表的长度。

算法描述：可以采用“数”结点的方法来求出单链表的长度，用指针 p 依次指向各个结点，从第一个结点开始“数”，一直“数”到最后一个结点（p->next=NULL）。

【算法 2-9】

```
int ListLength(LinkList head)  //本算法用来求带头结点的单链表 head 的单链表的长度
{
    int j;
    LinkList p;
    p=head->next;
    j=0;  //用来存放单链表的长度
    while(p!=NULL)
    {
        p=p->next;
        j++;
    }
    return j;
}
```

【例 2-2】如果以单链表表示集合，假设集合 A 用单链表 LA 表示，集合 B 用单链表 LB 表示，设计算法求两个集合的差，即 A-B。

算法思想：由集合运算的规则可知，集合的差 A-B 中包含所有属于集合 A 而不属于集合

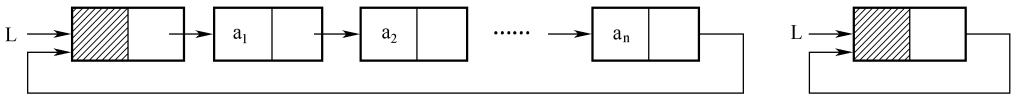
B 的元素。具体做法是：对于集合 A 中的每个元素 e，在集合 B 的链表 LB 中进行查找，若存在与 e 相同的元素，则从 LA 中将其删除。

【算法 2-10】

```
void Difference(LinkList LA,LinkList LB)   //此算法求两个集合的差集
{
    LinkList   pre,p,q,r;
    pre=LA;p=LA->next;    //p 指向 LA 表中的某一结点，而 pre 始终指向 p 的前驱
    while(p!=NULL)
    {
        q=LB->next;
        while(q!=NULL && q->data!=p->data)   //依次扫描 LB 中的结点，看有否
            q=q->next;                       //与 LA 中*P 结点的值相同的结点
        if(q!=NULL)
        {
            r=p;
            pre->next=p->next;
            p=p->next;
            free(r);
        }
        else
        {
            pre=p;
            p=p->next;
        }
    }
}
```

2.3.2 循环链表

循环链表（circular linked list）是单链表的另一种形式，它是一个首尾相接的链表。其特点是将单链表最后一个结点的指针域由 NULL 改为指向头结点或线性表中的第一个结点，从而得到单链形式的循环链表，并称为循环单链表。类似地，还有多重链的循环链表。在循环单链表中，表中所有结点被链在一个环上，多重循环链表则是将表中的结点链在多个环上。为了使某些操作方便实现，在循环单链表中也可设置一个头结点。这样，空循环链表仅由一个自成循环的头结点表示。带头结点的单循环链表如图 2-13 所示。



（a）非空表　　（b）空表

图 2-13　单循环链表示意图

带头结点的循环单链表的各种操作的实现算法与带头结点的单链表的实现算法类似，差别仅在于算法中的循环条件不是 p!=NULL 或 p->next !=NULL，而是 p!=L 或 p->next!=L。

在循环单链表中附设尾指针有时比附设头指针会使操作变得更简单。如在用头指针表示

的循环单链表中，找开始结点 a_1 的时间复杂度是 O(1)，然而要找到终端结点 a_n，则需要从头指针开始遍历整个链表，其时间复杂度是 O(n)。如果用尾指针 rear 来表示循环单链表，则查找开始结点和终端结点都很方便，它们的存储位置分别是 rear->next->next 和 rear，显然，查找时间复杂度都是 O(1)。因此，实际使用中多采用尾指针表示循环单链表，如图 2-14 所示。

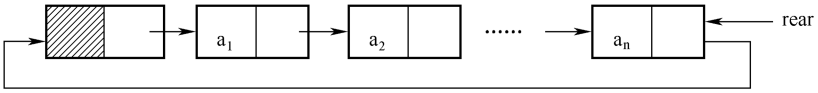


图 2-14　带尾指针 rear 的单循环链表

【例 2-3】有两个带头结点的循环单链表 LA、LB，编写一个算法，将两个循环单链表合并为一个循环单链表，其头指针为 LA。

算法思想：先找到两个链表的尾，并分别由指针 p、q 指向它们，然后将第一个链表的尾与第二个链表的第一个结点链接起来，并修改第二个链表的尾 q，使它的链域指向第一个链表的头结点，如图 2-15 所示。

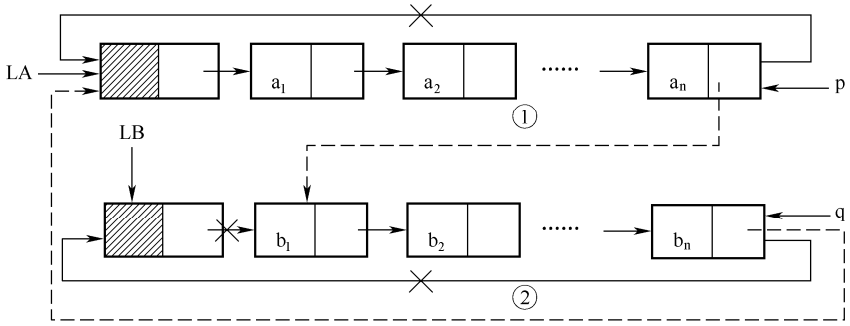


图 2-15　带头结点两个单循环链表的链接

【算法 2-11】

```
LinkList ConnectH(LinkList LA,LinkList LB)  //此算法将两个采用头指针的循环单链表的首尾连接起来
{
    LinkList p,q;
    p=LA;
    q=LB;
    while(p->next!=LA)
        p=p->next;              //找到表 LA 的表尾，用 p 指向它
    while(q->next!=LB)
        q=q->next;              //找到表 LB 的表尾，用 q 指向它
    q->next=LA;                 //修改表 LB 的尾指针，使之指向表 LA 的头结点
    p->next=LB->next;           //修改表 LA 的尾指针，使之指向表 LB 中的第一个结点
    free(LB);
    return(LA);
}
```

采用上面的方法，需要遍历链表，找到表尾，其执行时间是 O(n)。若在尾指针表示的单循环链表上实现，则只需要修改指针，无需遍历，其执行时间是 O(1)，如图 2-16 所示。

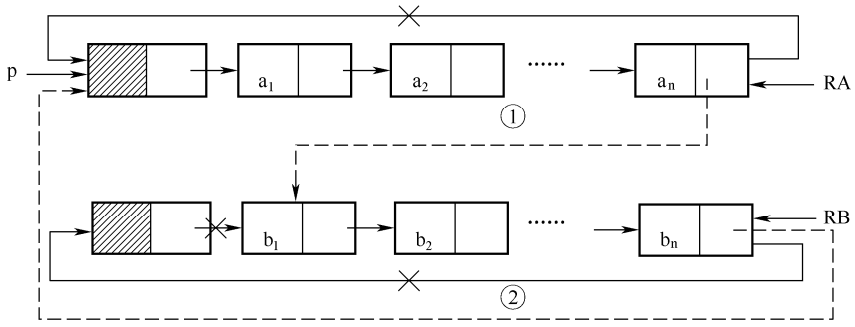


图 2-16　带尾结点两个单循环链表的链接

【算法 2-12】

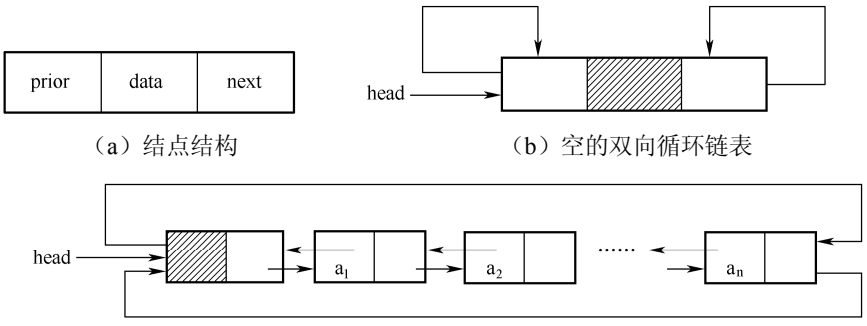
```
LinkList ConnectR(LinkList RA,LinkList RB)  //此算法将两个采用尾指针的循环链表首尾连接起来
{
    LinkList p;
    p=RA->next;                  //保存链表 RA 的头结点地址
    RA->next=RB->next->next; //链表 RB 的开始结点链到链表 RA 的终端结点之后
    free(RB->next);              //释放链表 RB 的头结点
    RB->next=p;                  //链表 RA 的头结点链到链表 RB 的终端结点之后
    return RB;                   //返回新循环链表的尾指针
}
```

2.3.3　双向循环链表

单向循环单链表的出现，虽然能够实现从任一结点出发沿着链找到其前驱结点，但时间耗费是 O(n)。如果希望从表中快速确定某一个结点的前驱，另一个解决方法就是在单链表的每个结点中再增加一个指向其前驱的指针域 prior，这样形成的链表中就有两条方向不同的链，称之为**双（向）链表**（double linked list）。双链表的结构定义如下：

```
typedef struct DuLNode{
    ElemType data;
    struct DuLNode *prior,*next;
}DuLNode,*DuLinkList;
```

与单链表类似，双链表一般也是由头指针唯一确定的，增加头结点也能使双链表的某些运算变得方便。同时双向链表也可以有循环表，称为双向循环链表，其结构如图 2-17 所示。



（a）结点结构　　（b）空的双向循环链表

（c）非空的双向循环表

图 2-17　双向循环链表示意图

由于在双向链表中既有前向链又有后向链，寻找任一个结点的直接前驱结点与直接后继结点变得非常方便。设指针 p 指向双链表中的某一结点，则有下式成立：

p->prior->next=p=p->next->prior

在双向链表中，那些只涉及后继指针的算法，如求表长度、取元素、元素定位等，与单链表中相应的算法相同，但对于前插和删除操作则涉及到前驱和后继两个方向的指针变化，因此与单链表中的算法不同。

1. 双向链表的插入操作

算法描述：欲在双向链表第 i 个结点（指针 p 所指）之前插入一个的新的结点（指针 s 所指），则指针的变化情况如图 2-18 所示。

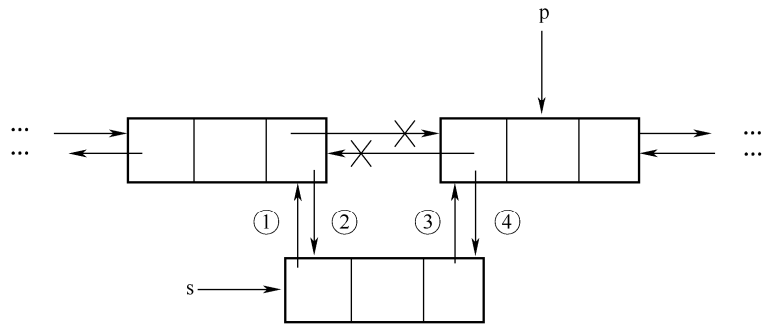


图 2-18　双链表的前插操作

【算法 2-13】

```
int DInserLinkList(DuLinkList L,int i,ElemType e)
{
    DuLinkList s,p;
    //……先检查待插入的位置 i 是否合法（实现方法同单向链表的前插操作）
    //……若位置 i 合法，则让指针 p 指向它
    s=(DuLinkList)malloc(sizeof(DuLNode));
    if(s)
    {
        s->data=e;
        s->prior=p->prior;      //对应图 2-18 中的①
        p->prior->next=s;       //对应图 2-18 中的②
        s->next=p;              //对应图 2-18 中的③
        p->prior=s;             //对应图 2-18 中的④
        return OK;
    }
    else
        return ERROR;
}
```

2. 双向链表的删除操作

算法描述：欲删除双向链表中的第 i 个结点，则指针的变化情况如图 2-19 所示。

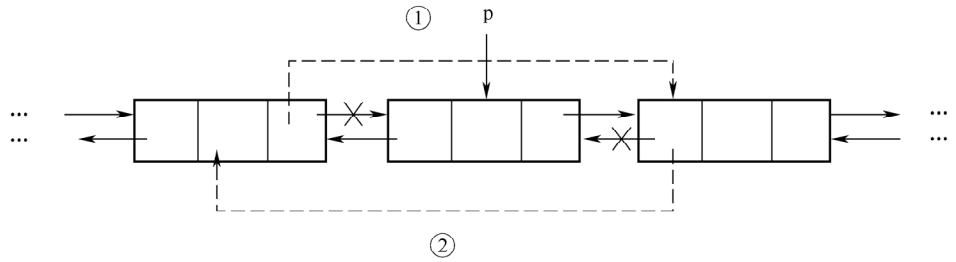


图 2-19　双向链表的删除操作

【算法 2-14】

```
int DDeletelinkList(DuLinkList L,int i,ElemType *e)
{
    DuLinkList p;
    //……先检查待插入的位置 i 是否合法（实现方法同单向链表的删除操作）
    //……若位置 i 合法，则让指针 p 指向它
    *e=p->data;
    p->prior->next=p->next;      //对应图 2-19 中的①
    p->next->prior=p->prior;     //对应图 2-19 中的②
    free(p);
    return OK;
}
```

2.3.4　循环链表

1. 基于空间的考虑

顺序表的存储空间是静态分配的，在程序执行之前必须明确规定它的存储规模。若线性表的长度 n 变化较大，则存储规模难以预先确定。估计过大将造成空间浪费，估计太小又将使空间溢出的机会增多。在静态链表中，初始存储池虽然也是静态分配的，但若同时存在若干个结点类型相同的链表，则它们可以共享空间，使各链表之间能够相互调节余缺，减少溢出机会。动态链表的存储空间是动态分配的，只要内存空间尚有空闲，就不会产生溢出。因此，当线性表的长度变化较大，难以估计其存储规模时，采用动态链表作为存储结构较好。

在链表中的每个结点，除了数据域外，还要额外设置指针域（或光标），从存储密度来讲，这是不经济的。所谓**存储密度**（storage density）是指结点数据本身所占的存储量与整个结点结构所占的存储量之比，即：

存储密度=结点数据本身所占的存储量/结点结构所占的存储总量

一般地，存储密度越大，存储空间的利用率就高。显然，顺序表的存储密度为 1，而链表的存储密度小于 1。例如单链表的结点的数据均为整数，指针所占空间和整型量相同，则单链表的存储密度为 50%。因此若不考虑顺序表中的备用结点空间，则顺序表的存储空间利用率为 100%，而单链表的存储空间利用率为 50%。由此可知，当线性表的长度变化不大，易于事先确定其大小时，为了节约存储空间，宜采用顺序表作为存储结构。

2. 基于时间的考虑

顺序表是由向量实现的，它是一种随机存取结构，对表中任一结点都可以在 O(1)时间内直接地存取，而链表中的结点，需从头指针起顺着链找才能取得。因此，若线性表的操作主要

是进行查找，很少做插入和删除时，采用顺序表做存储结构为宜。

在链表中的任意位置上进行插入和删除，都只需要修改指针。而在顺序表中进行插入和删除，平均要移动表中近一半的结点，尤其是当每个结点的信息量较大时，移动结点的时间开销就相当可观。因此，对于频繁进行插入和删除的线性表，宜采用链表做存储结构。若表的插入和删除主要发生在表的首尾两端，则采用尾指针表示的单循环链表为宜。

3. 基于语言的考虑

对于没有提供指针类型的高级语言，若要采用链表结构，则可以使用光标实现的静态链表。虽然静态链表在存储分配上有不足之处，但它和动态链表一样，具有插入和删除方便的特点。

值得指出的是：即使是对那些具有指针类型的语言，静态链表也有其用武之地。特别是当线性表的长度不变，仅需改变结点之间的相对关系时，静态链表比动态链表可能更方便。

2.4　一元多项式的表示及相加

对于符号多项式的各种操作，实际上都可以利用线性表来处理。比较典型的是关于一元多项式的处理。在数学上，一个一元多项式 $P_n(x)$可按升幂的形式写成：

$$P_n(x)=p_0+p_1x+p_2x^2+p_3x^3+\cdots+p_nx^n$$

它实际上可以由 n+1 个系数唯一确定。因此，在计算机中，可以用一个线性表 P 来表示：

$$P=(p_0,p_1,p_2,\cdots,p_n)$$

其中每一项的指数隐含在其系数的序号中。

假设 $Q_m(x)$是一个一元多项式，则它也可以用一个线性表 Q 来表示，即：

$$Q=(q_0,q_1,q_2,\cdots,q_m)$$

若假设 m<n，则两个多项式相加的结果 $R_n(x)=P_n(x)+Q_m(x)$，也可以用线性表 R 来表示：

$$R=(p_0+q_0,p_1+q_1,p_2+q_2,\cdots,p_m+q_m,p_{m+1},\cdots,p_n)$$

我们可以采用顺序存储结构来实现，顺序表的方法，使得多项式的相加的算法定义十分简单，即 p[0]存系数 p_0，p[1]存系数 p_1……p[n]存系数 p_n，对应单元的内容相加即可。但是在通常的应用中，多项式的指数有时可能会很高并且变化很大，例如：

$$R(x)=1+5x^{10000}+7x^{20000}$$

若采用顺序存储，则需要 20001 个空间，而存储的有用数据只有三个，这无疑是一种浪费。若只存储非零系数项，则必须存储相应的指数信息才行。

假设一元多项式 $P_n(x)=p_1x^{e1}+p_2x^{e2}+\cdots+p_mx^{em}$，其中 p_i 是指数为 ei 的项的系数（且 $0\leqslant e1\leqslant e2\leqslant\cdots\leqslant em=n$），若只存非零系数，则多项式中每一项由两项构成（指数项和系数项），用线性表来表示，即：

$$((p_1,e1),(p_2,e2),\cdots(p_m,em))$$

采用这样的方法存储，在最坏情况下，即 n+1 个系数都不为零，则比只存储系数的方法多存储一倍的数据。对于非零系数多的多项式则不宜采用这种表示。

对于线性表的两种存储结构，一元多项式也有两种存储表示方法。在实际应用中，可以视具体情况而定。下面给出用单链表实现一元多项式相加运算的方法。

（1）用单链表存储多项式的结点结构如下：

```
typedef struct PolyNode
{
    int coef;
    int exp;
    struct PolyNode *next;
}PolyNode,*PolyList;
```

（2）通过键盘输入一组多项式的系数和指数，以输入系数 0 为结束标志，并约定建立多项式链表时，总是按指数从小到大的顺序排列。

算法描述：从键盘接受输入的系数和指数，用尾插法建立一元多项式的链表。

【算法 2-15】

```
PolyList PolyCreate()
{
    PolyList head, rear, s;
    int c,e;
    head=(PolyList)malloc(sizeof(PolyNode));  //建立多项式的头结点
    rear=head;                  //rear 始终指向单链表的尾，便于尾插法建表
    scanf("%d,%d",&c,&e);  //键入多项式的系数和指数项
    while(c!=0)                 //若 c=0，则代表多项式的输入结束
    {
        s=(PolyList)malloc(sizeof(PolyNode));  //申请新的结点
        s->coef=c;
        s->exp=e;rear->next=s;  //在当前表尾做插入
        rear=s;
        scanf("%d,%d",&c,&e);
    }
    rear->next=NULL;  //将表的最后一个结点的 next 置 NULL，以示表结束
    return(head);
}
```

（3）图 2-20 所示为两个多项式的单链表，分别表示多项式 $A(x)=7+3x+9x^8+5x^{17}$ 和多项式 $B(x)=8x+22x^7-9x^8$。

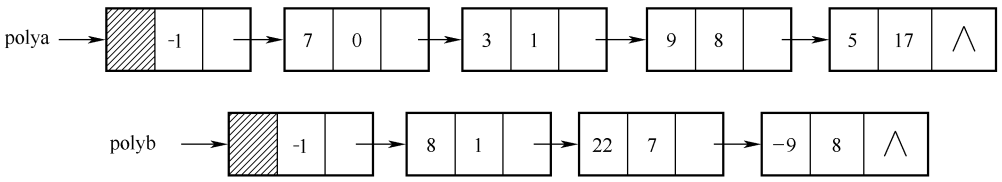


图 2-20　多项式的单链表表示法

多项式相加的运算规则是：两个多项式中所有指数相同的项的对应系数相加，若和不为零，则构成“和多项式”中的一项；所有指数不相同的项均复制到“和多项式”中。以单链表作为存储结构，并且“和多项式”中的结点无需另生成，则可看成是将多项式 B 加到多项式 A 中，由此得到下列运算规则（设 p、q 分别指向多项式 A，B 的一项，比较结点的指数项）。

（1）若 p->exp<q->exp，则结点 p 所指的结点应是“和多项式”中的一项，令指针 p 后移。

（2）若 p->exp>q->exp，则结点 q 所指的结点应是“和多项式”中的一项，将结点 q 插

入在结点 p 之前，且令指针 q 在原来的链表上后移。

（3）若 p->exp=q->exp，则将两个结点中的系数相加，当和不为零时修改结点 p 的系数域，释放 q 结点；若和为零，则和多项式中无此项，从 A 中删去 p 结点，同时释放 p 和 q 结点。

【算法 2-16】

```
void PolyAdd(PolyList polya,PolyList polyb)   //此函数用于将两个多项式相加，然后将和多项式存放在多项式 polya 中，并将多项式 ployb 删除
{
    PolyList p,q,pre,temp;
    int sum;
    p=polya->next;   //令 p 指向 polya 多项式链表中的第一个结点
    q=polyb->next;   //令 q 指向 polyb 多项式链表中的第一个结点
    pre=polya;       //pre 指向和多项式的尾结点
    while(p!=NULL && q!=NULL)//当两个多项式均未扫描结束时
    {
        if(p->exp<q->exp)//如果 p 指向的多项式项的指数小于 q 的指数，将 p 结点加入到和多项式中
        {
            pre->next=p;pre=pre->next;
            p=p->next;
        }
        else if(p->exp==q->exp)//若指数相等，则相应的系数相加
        {
            sum=p->coef+q->coef;
            if(sum!=0)
            {
                p->coef=sum;pre->next=p;pre=pre->next;
                p=p->next;temp=q;q=q->next;free(temp);
            }
            else //若系数和为零，则删除结点 p 与 q，并将指针指向下一个结点
            {
                temp=p->next;free(p);p=temp;
                temp=q->next;free(q);q=temp;
            }
        }
        else
        {
            pre->next=q;pre=pre->next;   //将 q 结点加入到和多项式中
            q=q->next;
        }
    }
    if(p!=NULL)     //多项式 A 中还有剩余，则将剩余的结点加入到和多项式中
        pre->next=p;
    else            //否则，将 B 中的结点加入到和多项式中
        pre->next=q;
}
```

假设 A 多项式有 M 项，B 多项式有 N 项，则上述算法的时间复杂度为 O(M+N)。图 2-21 所示为图 2-20 中两个多项式的和，其中孤立的结点代表被释放的结点。

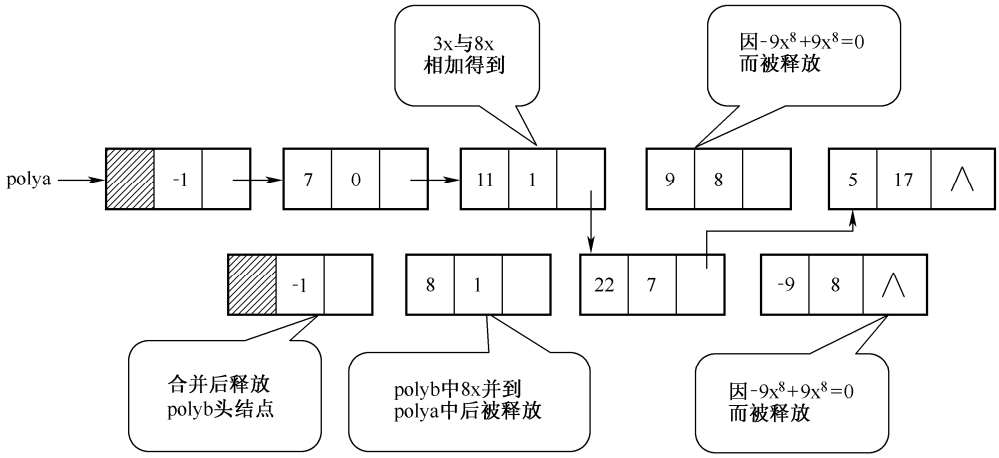


图 2-21　多项式相加得到的多项式和

通过对多项式加法的介绍，我们可以将其推广到实现两个多项式的相乘，因为乘法可以分解为一系列的加法运算。

一、选择题

1．下述_____是顺序存储结构的优点。

A．存储密度大　　B．插入运算方便

C．删除运算方便　　D．可方便地用于各种逻辑结构的存储表示

2．下面关于线性表的叙述中，_____是错误的。

A．线性表采用顺序存储，必须占用一片连续的存储单元

B．线性表采用顺序存储，便于进行插入和删除操作

C．线性表采用链式存储，不必占用一片连续的存储单元

D．线性表采用链式存储，便于插入和删除操作

3．线性表是具有 n 个_____的有限序列（n>0）。

A．表元素　　B．字符　　C．数据元素

D．数据项　　E．信息项

4．若某线性表最常用的操作是存取任一指定序号的元素和在最后进行插入和删除运算，则利用_____存储方式最节省时间。

A．顺序表　　B．双链表

C．带头结点的双循环链表　　D．单循环链表

5．若线性表中有 n 个元素，算法_____在单链表上实现要比在顺序表上实现效率更高。

A．删除所有值为 x 的元素　　B．在最后一个元素的后面插入一个新元素

C．顺序输出前 k 个元素　　D．交换其中某两个元素的值

6．某线性表中最常用的操作是在最后一个元素之后插入一个元素和删除第一个元素，则采用_____存

储方式最节省运算时间。

A．单链表　　B．仅有头指针的单循环链表

C．双链表　　D．仅有尾指针的单循环链表

7．设一个链表最常用的操作是在末尾插入结点和删除尾结点，则选用_____最节省时间。

A．单链表　　B．单循环链表

C．带尾指针的单循环链表　　D．带头结点的双循环链表

8．静态链表中指针表示的是_____。

A．内存地址　　B．数组下标　　C．下一元素地址　　D．左、右孩子地址

9．链表不具有的特点是_____。

A．插入、删除不需要移动元素　　B．可随机访问任一元素

C．不必事先估计存储空间　　D．所需空间与线性长度成正比

10．若长度为 n 的线性表采用顺序存储结构，在其第 i 个位置插入一个新元素的算法的时间复杂度为_____（1≤i≤n+1）。

A．O(0)　　B．O(1)　　C．O(n)　　D．$O(n^2)$

11．对于顺序存储的线性表，访问结点和增加、删除结点的时间复杂度分别为_____。

A．O(n)　O(n)　　B．O(n)　O(1)　　C．O(1)　O(n)　　D．O(1)　O(1)

12．线性表$(a_1,a_2,\cdots,a_n)$以链式方式存储时，访问第 i 位置元素的时间复杂度为_____。

A．O(i)　　B．O(1)　　C．O(n)　　D．O(i-1)

13．非空的循环单链表 head（头指针）的尾结点指针 p 满足_____。

A．p->next==head　　B．p->next==NULL

C．p==NULL　　D．p==head

14．在一个单链表中，已知指针 q 所指结点是指针 p 所指结点的前驱结点，若在 q 和 p 之间插入结点 s，则执行_____。

A．s->next=p->next;p->next=s;　　B．p->next=s->next;s->next=p;

C．q->next=s;s->next=p;　　D．p->next=s;s->next=q;

15．在一个单链表中，若删除指针 p 所指结点的后续结点，则执行_____。

A．p->next=p->next->next;　　B．p=p->next;p->next=p->next->next;

C．p->next=p->next　　D．p=p->next->next

16．在双向链表指针 p 所指的结点前插入指针 s 所指的新结点的操作为_____。

A．p->prior=s;s->next=p;p->prior->next=s;s->prior=p->prior;

B．p->prior=s;p->prior->next=s;s->next=p;s->prior=p->prior;

C．s->next=p;s->prior=p->prior;p->prior=s;p->prior->next=s;

D．s->next=p;s->prior=p->prior;p->prior->next=s;p->prior=s;

二、判断题

1．链表中的头结点仅起到标识的作用。（　　）

2．顺序存储结构的主要缺点是不利于插入或删除操作。（　　）

3．线性表采用链表存储时，结点和结点内部的存储空间可以是不连续的。（　　）

4．顺序存储方式插入和删除结点时效率太低，因此它不如链式存储方式好。（　　）

5．对任何数据结构，链式存储结构一定优于顺序存储结构。（　　）

6．顺序存储方式只能用于存储线性结构。（　　）

7．集合与线性表的区别在于是否按关键字排序。（　　）

8．所谓静态链表就是一直不发生变化的链表。（　　）

9．线性表的特点是每个元素都有一个前驱和一个后继。（　　）

10．取得线性表的第 i 个元素的时间同 i 的大小有关。（　　）

11．线性表只能用顺序存储结构实现。（　　）

12．线性表就是顺序存储的表。（　　）

13．为了很方便的插入和删除数据，可以使用双向链表存放数据。（　　）

14．顺序存储方式的优点是存储密度大，且插入、删除运算效率高。（　　）

15．链表是采用链式存储结构的线性表，进行插入、删除操作时，在链表中比在顺序存储结构中效率高。（　　）

三、填空题

1．当线性表的元素总数基本稳定，且很少进行插入和删除操作，但要求以最快的速度存取线性表中的元素时，应采用_____存储结构。

2．线性表 L=(a_1,a_2,…,a_n)用数组表示，假定删除表中任一元素的概率相同，则删除一个元素平均需要移动元素的个数是_____。

3．设单链表的结点结构为(data,next)，next 为指针域，已知指针 px 指向单链表中 data 为 x 的结点，指针 py 指向 data 为 y 的新结点，若将结点 y 插入结点 x 之后，则需要执行以下语句_____、_____。

4．在一个长度为 n 的顺序表中第 i 个元素（1≤i≤n）之前插入一个元素时，需向后移动_____个元素。

5．在单链表中设置头结点的作用是_____。

6．对于一个具有 n 个结点的单链表，在已知的结点*p 后插入一个新结点的时间复杂度为_____，在给定值为 x 的结点后插入一个新结点的时间复杂度为_____。

7．根据线性表的链式存储结构中每一个结点包含的指针个数，将线性链表分成_____和_____；根据指针的连接方式，链表又可分成_____和_____。

8．在双向循环链表中，向 p 所指的结点后插入指针 f 所指的结点，其操作是_____、_____、_____、_____。

9．链式存储的特点是利用_____来表示数据元素之间的逻辑关系。

10．顺序存储结构是通过_____表示元素之间的关系的；链式存储结构是通过_____表示元素之间的关系的。

11．对于双向链表，在两个结点之间插入一个新结点需修改的指针共_____个，单链表为_____个。

12．循环单链表的最大优点是_____。

13．已知指针 p 指向单链表 L 中的某结点，则删除其后继结点的语句是_____、_____、_____。

14．带头结点的双循环链表 L 为空表的条件是_____。

四、编程题

1．已知两个顺序表 La 和 Lb 中的元素递增有序，试利用顺序表的基本操作实现将 La 与 Lb 合并为一个新的顺序表 Lc，且 Lc 中的元素亦递增有序。

2．已知一个顺序表 L 中的元素递增有序，编写一个算法，将元素 e 插入到顺序表中，且插入 e 后该表仍保持递增有序。

3．已知一个顺序表 L 中的数据元素为整数类型，编写一个算法，将该表中的所有奇数排在偶数之前，即表的前面为奇数，后面为偶数。

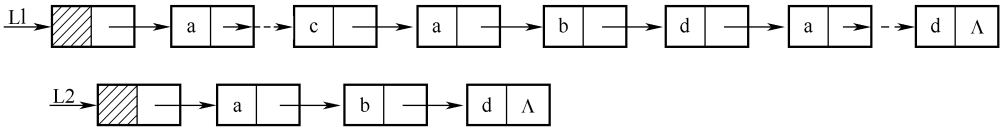
4．编写一个算法，实现顺序表就地逆置操作，即在原顺序表存储空间上将元素按位序逆转。

5．编写一个算法，实现带头结点的单链表就地逆置操作，即利用原链表结点空间实现逆转。

6．编写一个算法，计算带头结点的单链表 L 中数据域值为 x 的结点个数。

7．编写一个算法，将带有头结点单链表 L 中数据域值最小的那个结点移到链表的最前面。

8．L1 与 L2 分别为两单链表头结点地址指针，且两表中数据结点的数据域均为一个字母。设计把 L1 中与 L2 中数据相同的连续结点顺序完全倒置的算法。



9．设线性表 $A=(a_1,a_2,\cdots,a_m)$，$B=(b_1,b_2,\cdots,b_n)$，试写一个按下列规则合并 A、B 为线性表 C 的算法，使得：当 $m\leqslant n$ 时，$C=(a_1,b_1,\cdots,a_m,b_m,b_{m+1},\cdots,b_n)$；或者当 $m>n$ 时，$C=(a_1,b_1,\cdots,a_n,b_n,a_{n+1},\cdots,a_m)$。

线性表 A、B、C 均以单链表作为存储结构，且 C 表利用 A 表和 B 表中的结点空间构成。注意：单链表的长度值 m 和 n 均未显式存储。

10．已知有单链表表示的线性表中含有三类字符的数据元素（如字母字符、数字字符和其他字符），试编写算法来构造三个以循环链表表示的线性表，使每个表中只含同一类的字符，且利用原表中的结点空间作为这三个表的结点空间，头结点可另辟空间。