第2章 线性表

知识要点

(1) 熟悉线性表的定义；
(2) 掌握线性表在顺序存储结构上实现的基本操作：插入和删除算法；
(3) 熟练掌握线性表在链式存储结构中的基本操作，并能在实际中选用适当的链表结构；
(4) 了解两种存储结构的特点及其适用场合；
(5) 了解特殊线性表广义表、字符串。

线性结构的基本特点：

线性结构是由数据元素组成的有限集合，在非空有限集中：

（1）存在唯一的被称做“**第一个**”的数据元素(**首元结点**)；

（2）存在唯一的被称做“**最后一个**”的数据元素(**尾元结点**)；

（3）除第一个元素外，集合中每个数据元素**均只有一个前驱**；

（4）除最后一个元素外，集合中每个数据元素**均只有一个后继**。

2.1 线性表的定义

线性表是由长度为n（n≥0)的一组结点$a_0, a_1, \ldots, a_{n-2}, a_{n-1}$组成的有限序列。

$$L= (a_0, a_1, \ldots, a_{n-2}, a_{n-1})$$

当n=0时，线性表为空，称为**空表**。L为**线性表名称**。

如果n＞0，线性表为非空表，在非空表中的每个数据元素都有一个确定的位置，a_0是线性表的**第一个数据元素**，表中的每一个结点a_i (i=0,…,n-1)都有一个**直接前驱结点**a_{i-1}和一个**直接后继结点**a_{i+1}，a_{n-1}是线性表的**最后一个数据元素**。

线性表的长度为数据元素的数目n。

根据线性表的定义，其形式化描述为：

长度为n的线性表是一种数据结构 L = (D, R)

其中：

D是数据集，即由n个数据元素组成的集合，$D=\{a_i \mid i=0,\ldots,n-1\}$；

R是关系集，即确定了D中数据元素的关系，$R=\{<a_{i-1}, a_i> \mid a_{i-1}, a_i \in D, i=1,\ldots,n-1\}$。

线性表用一个**抽象数据类型**(Abstract Data Type, **ADT**)定义如下：

算法3.1：线性表定义

```
template <class Elem> class List {
public:
    virtual void clear()=0;                  // 删除所有的数据元素
    virtual bool insert(const Elem&)=0;  // 在当前的位置之后插入数据元素
    virtual bool append(const Elem&)=0; // 在末尾插入数据元素
    virtual bool remove(Elem&)=0;       // 在当前的位置之后删除数据元素
    virtual void setStart()=0;               //设置当前位置到表头
    virtual void setEnd()=0;                 //设置当前位置到表尾
    virtual void next ()=0;                  //移动当前位置到其后继
    virtual void prev ()=0;                  //移动当前位置到其前驱
    virtual bool setPos(int pos)=0;          //指定当前数据元素的位置
    virtual bool getValue(Elem&) const=0;//返回当前数据元素的取值
    virtual bool IsEmpty()=0;                //判断线性表是否为空
    virtual bool IsFull()= 0;                //判断线性表是否为满
    virtual void print() const=0;            //输出表内容
};
```

2.2 线性表的顺序存储结构

2.2.1 顺序存储结构

顺序存储结构是用一组连续的存储单元来存储线性表，即利用数据元素之间的相对位置表示它们之间的次序关系。相邻元素的物理位置必定相邻，线性表$L=(a_0, a_1, \ldots, a_{n-2}, a_{n-1})$的顺序存储结构如图2-1所示。

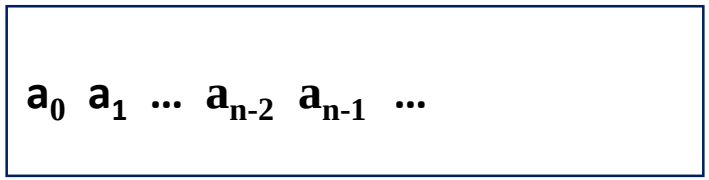


图2-1线性表L的顺序存储表示

采用顺序存储方式存储的线性表称为顺序表。其特点是：

(1) 线性表的逻辑顺序与物理顺序一致。

(2) 数据元素之间的关系采用物理位置的相邻关系来体现。

假设线性表的每个元素需占用c个存储单元，并以所占的第一个单元的存储地址作为数据元素的存储位置。那么线性表中第i个数据元素的存储位置$LOC(a_i)$和第i-1个数据元素的存储位置$LOC(a_{i-1})$之间满足下列关系：

$$LOC(a_i)= LOC(a_{i-1})+c \qquad (2\text{-}1)$$

假设线性表中第一个数据元素a_0所在的存储地址为$LOC(a_0)$，线性表中第i个数据元素的存储地址为：

$$LOC(a_i)= LOC(a_0)+i\times c \qquad (2\text{-}2)$$

例：设有一维数组M，下标的范围是0到9，每个数组元素用相邻的5个字节存储。存储器按字节编址，设存储数组元素M[0]的第一个字节的地址是98，则M[3]的第一个字节的地址是：LOC(M[3])=98+3×5=113。

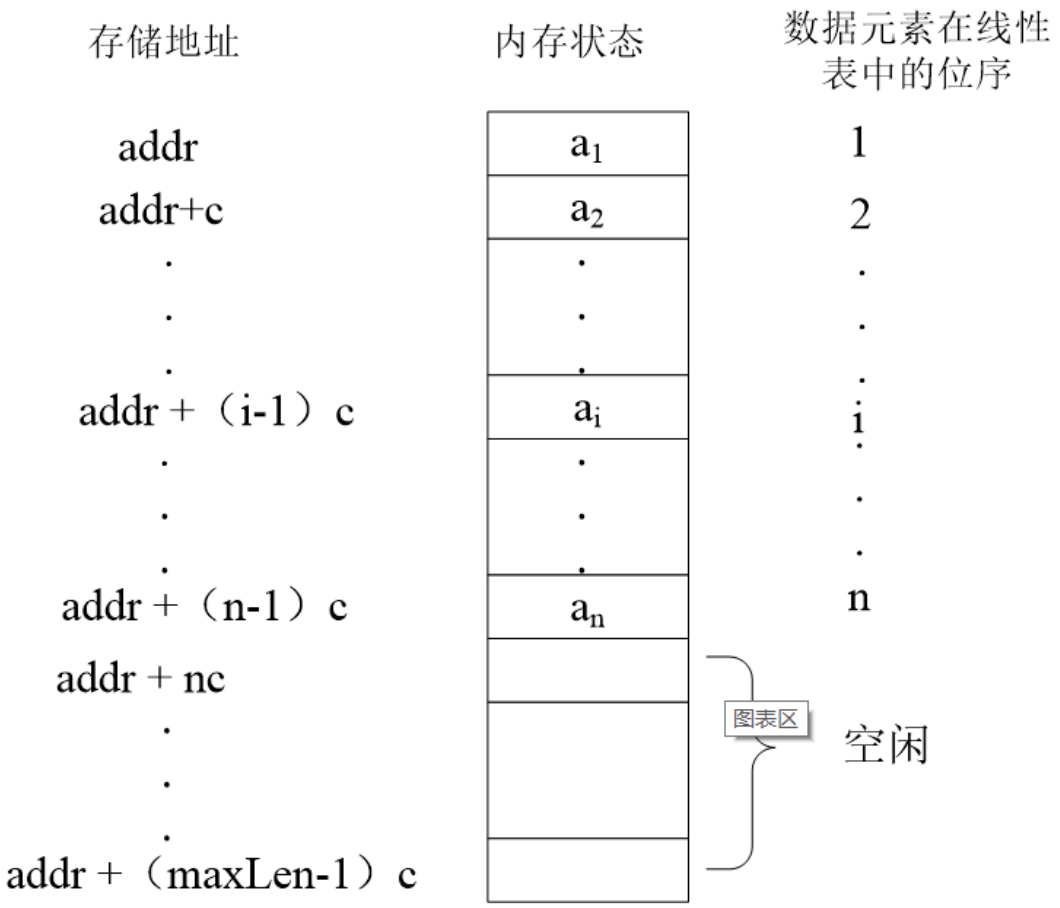


图 2-2　线性表的顺序存储结构示意图

线性表顺序存储结构的形式定义如下：

算法2.2：线性表顺序存储结构的定义

```
template <class Elem>
class  Alist: public list<Elem> {
    private:
        int maxSize, listSize, curr; // 线性表最大长度、表长、当前元素位置
        Elem * listArray;
    public:
        Alist(int size=DefaultListSize) { //构造函数
            maxsize=size; listSize = curr =0;
            listArray = new Elem[ maxSize ];
        }
        ~Alist() {  delete [ ] listArray; }          // 析构函数
        void clear(){ listSize = curr =0; }          //清空线性表
        void Prev( ){ if (curr>0) curr--; }          //当前位置curr前移到前驱
        void Next( ){ if (curr<listSize-1) curr++;}  //当前位置curr后移到后继
        bool setPos(int pos);                        //任意指定当前数据元素的位置
        bool insert(const Elem it);                  //在当前位置插入元素
        bool append(const Elem it);                  //在表尾插入元素
        bool remove( Elem &it);                      //删除当前位置元素并返回其值
        ….     //其它操作的实现
};
```

2.2.2 顺序存储结构的实现

1. 插入操作

线性表的插入操作是指在线性表L=(a_0, a_1, …, a_{i-1}, a_i, a_{i+1}, a_{n-1})中的第i-1个数据元素和第i个数据元素之间插入一个新的数据元素b，使长度为n的线性表变成长度为n+1的线性表L'=(a_0, a_1, …, a_{i-1}, b, a_i, a_{i+1}, a_{n-1})。一个线性表在进行插入操作前后其数据元素在存储空间中的位置变化如图2-3所示。

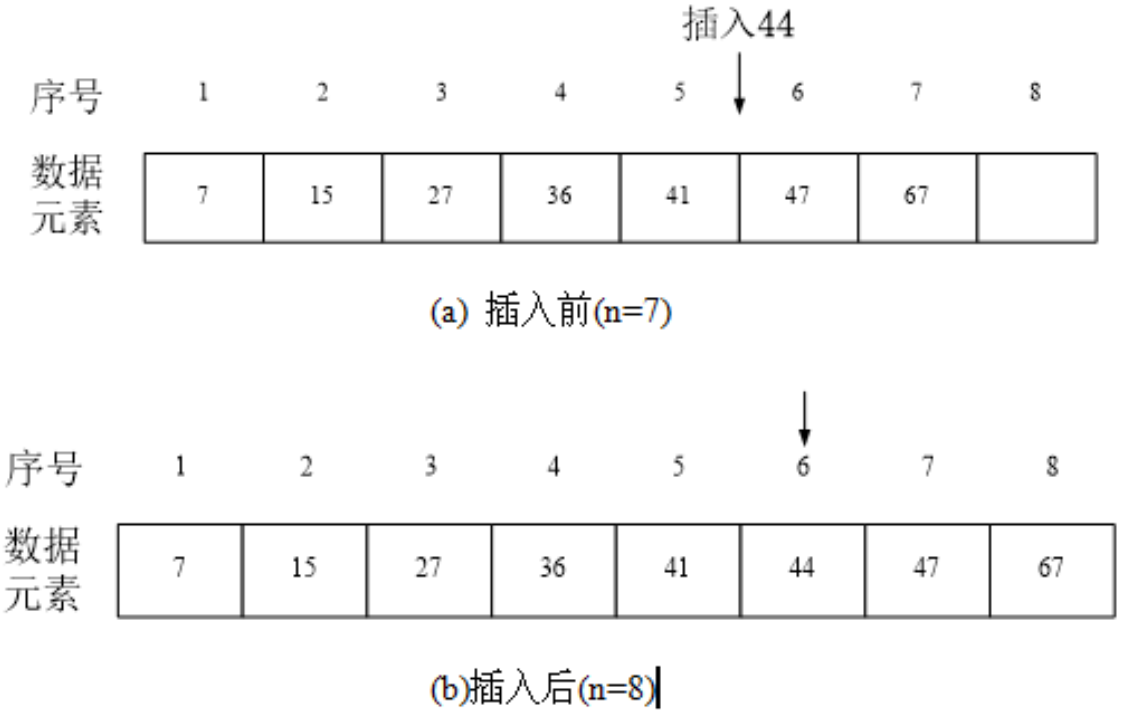


图 3-3 线性表插入前后的状况

插入操作的实现步骤：

（1） 将线性表中的第i个至第n-1个数据元素向后移动一个位置。

（2） 将数据元素b插入到数据元素a_{i-1}之后，即第i个位置上。

（3） 将线性表长度加1。

线性表的插入操作算法实现：

算法2.3：**线性表的指定位置插入（时间复杂度为O(n)）**

```
template <class Elem>    //在curr当前位置插入数据元素
bool Alist <Elem>:: insert(const Elem it) {
    if (listSize == maxSize) return false;  //判断线性表是否已满
    if ((curr <0) || ( curr > listSize-1) return false; //判断位置是否正确
    for (int i=listSize; i>curr; i--)          //将第i到n-1个元素后移
        listArray[i] = listArray[i-1];
    listArray[curr] = it;   //插入数据元素
    listSize++;             //增加线性表长度
    return true
}
```

2. 追加操作

追加操作是指在线性表的表尾插入一个数据元素。而前面的插入操作是指在线性表中的任一位置插入数据元素。由于追加操作不需移动数据元素，所以时间复杂度为O(1)。

线性表的追加操作算法实现：

算法2.4：线性表的追加插入

```
template <class Elem>
bool Alist<Elem>:: append(const Elem it) {
    if (listSize == maxSize) return false;
    listArray[listSize++] = it;
    return true
}
```

3. 删除操作

线性表的删除操作是指将长度为n的线性表L中删除第i个数据元素，使其变成长度为n−1的线性表L’。

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

$$L' = (a_0, a_1, \cdots, a_{i-1}, a_{i+1}, \cdots, a_{n-1})$$

一个线性表在进行删除操作前后其数据元素在存储空间中的位置变化如图2−4所示。

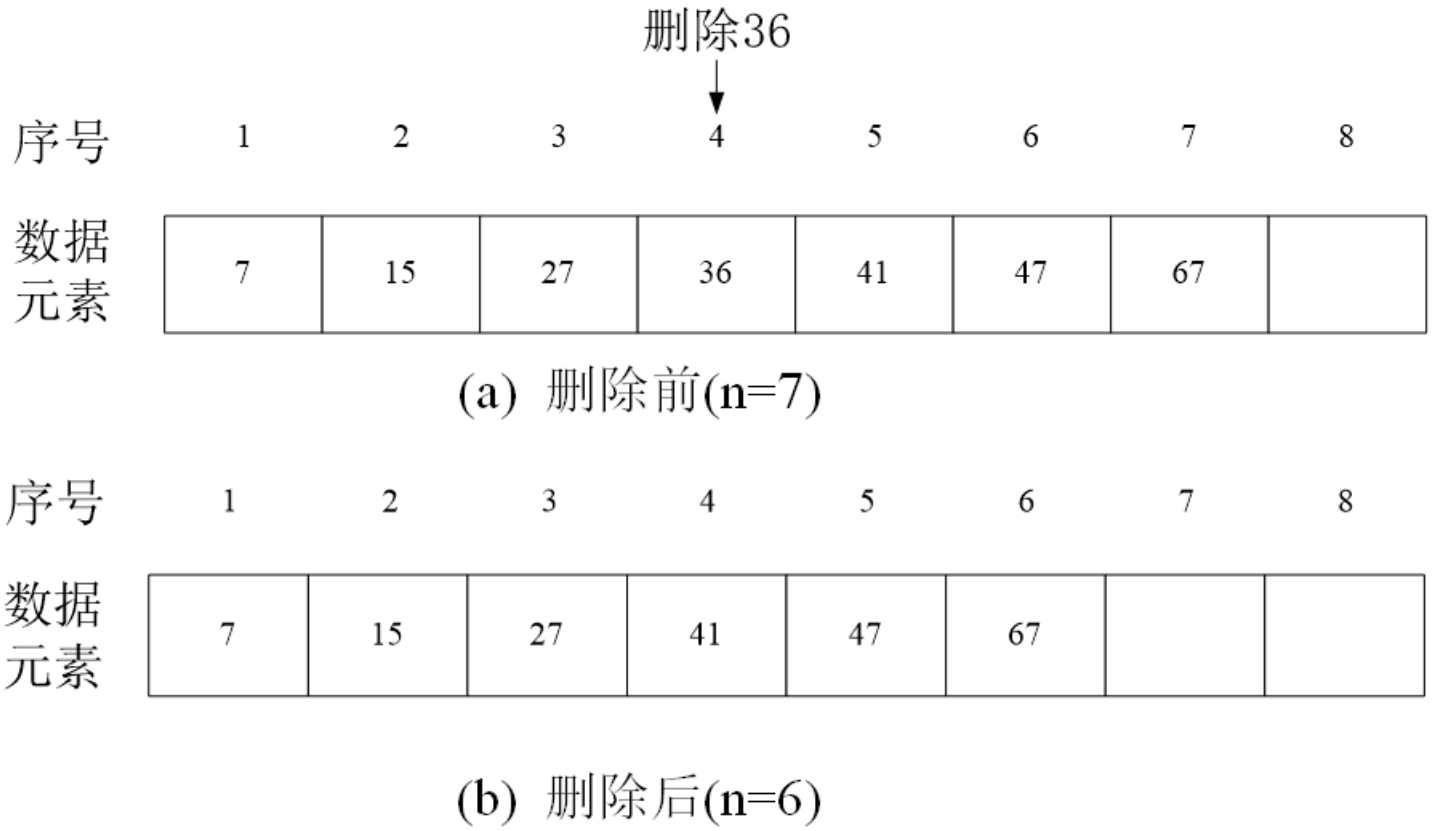


图2-4 线性表删除前后的状况

删除操作的实现步骤：

（1）在线性表L中，将第i+1个至第n-1个数据元素依次向前移动一个位置。

（2） 将线性表长度减1。

线性表的删除操作算法实现：

算法2.5：线性表的删除

```
template <class Elem>        // 删除并返回curr位置元素值
bool Alist<Elem>:: remove(Elem &it) {
    if (listSize == 0) return FALSE;
    if ((curr <0) || ( curr >= listSize)  return FALSE;
    it = listArray[curr];
    for (int i=curr; i<listSize-1; i++) listArray[i] = listArray[i+1];
    listSize--;
    return TURE;
}
```

4.时间复杂度分析

假设p_i是在线性表L中的第i个数据元素之前插入一个数据元素的概率，而在长度为n的线性表L中，在第i个数据元素之前插入一个数据元素的移动次数为：n-i+1。这样，在线性表L中插入数据元素时所需移动数据元素的期望值(平均次数)为：

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

假设q_i是删除线性表L中的第i个数据元素的概率，而在长度为n的线性表L中，删除第i个数据元素的移动次数为n-i。在线性表L中删除一个数据元素时所需移动数据元素的期望值(平均次数)为：

$$E_{delete} = \sum_{i=1}^{n} q_i \times (n - i)$$

在通常情况下，假定在线性表L中任何位置插入或删除数据元素是等概率的，则

$$p_i = \frac{1}{n+1}, \quad q_i = \frac{1}{n}$$

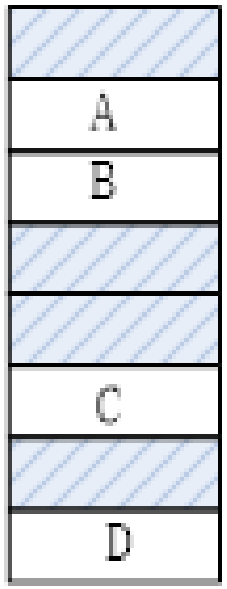
所以，

$$E_{insert} = \sum_{i=1}^{n+1}\left(\frac{1}{n+1} \times (n-i+1)\right) = \frac{1}{n+1}\sum_{i=1}^{n+1}(n-i+1) = \frac{n}{2}$$

$$E_{delete} = \sum_{i=1}^{n}\left(\frac{1}{n} \times (n-i)\right) = \frac{1}{n}\sum_{i=1}^{n}(n-i) = \frac{n-1}{2}$$

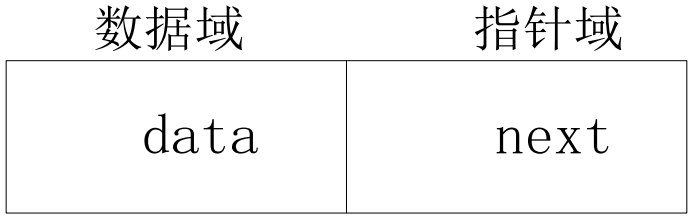
2.3 线性表的链式存储结构

- 链式存储结构**不要求**逻辑上相邻的数据元素在物理位置上也必须相邻，即不要求后继结点存放在固定间隔距离的位置上，在存储单元中的顺序可以是任意的，既可以是连续的，也可以是零散分布的。
- **插入或删除操作时不需要移动元素。**但它失去了顺序表可以随机存取的特点。
- 如线性表L(A,B,C,D)，存储分布如图所示。



2.3.1 单链表

单链表，也称为线性链表，是一种最简单的线性表的链式存储结构。用它来存储线性表时，每个数据元素用一个结点（Node）来存储，**每一个结点包含两个域，即存储数据元素信息data的域称为数据域，存储后继结点存放地址next的域称为指针域，指针域中存储的信息称为指针或链。**一般情况下，链表中每个结点可以包含若干个数据域和指针域。如果每个结点中只包含一个指针域，则称其为单链表。单链表的结点结构如图所示。



将n个结点链接起来就构成了链表(Linked list)。

一个线性表L(A,B,C,D)的单链表结构如图2-7所示。

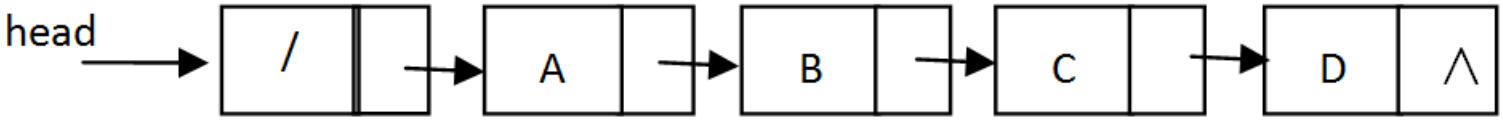


图2-7 带头结点的单链表结构示例图

访问链表中的任何结点都必须从链表的头指针开始，因此在图中的最前端增加了一个结点，这个结点没有存储任何数据元素，称为**头结点**。

单链表的头结点存储地址可从指针head找到，称指针head为**头指针**，其它结点的地址由前驱结点的指针next得到。

表中最后一个数据元素没有后继结点，所以它的指针域为“空”，用“∧”或“NULL”表示。

若单链表中只有一个头结点，而没有数据元素时，则此线性表为**空表**，头结点的指针域为“空”，如图2-8所示。

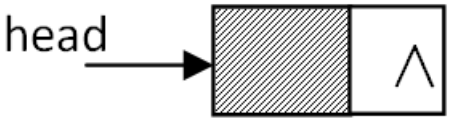


图2-8 带头结点的空单链表结构示例图

算法2.6： 单链表的结点类定义

```
template <class Elem>
class Link {
Public:
        Elem element;          // 当前结点的数据元素
        Link *next;            // 指向下一结点的指针

        Link(const Elem& item, Link * nextval=NULL)
                { element = item; next = nextval; }
        Link(Link* nextval=NULL)  { next = nextval; }
};
```

算法2.7：带头结点的单链表类定义

```
template <class Elem>
class  LList : public List<Elem> {
Private:  Link<Elem>* head;          // 头指针
          Link<Elem>* tail;          // 指向表尾的指针
          Link<Elem>* curr;          // 指向当前数据元素的指针
          void init(){
                    curr= tail = head = new Link<Elem>; //创建一个结点
          } //初始化一个空表
          void removeall(){
                    while(head != NULL){
                              curr=head;
                              head=head->next;
                              delete curr;
                    }
          } // 释放所有的结点
//接下页
```

```
Public:
    LList( ) { init(); }
    ~LLink( ) { removeall(); }
    void Create (int n);                    //创建长度为n的单链表
    bool getValue (Elem &e);                //读取当前位置curr元素的值
    Link * Locate(Elem e);                  //返回第一个与e匹配的元素位置
    bool IsEmpty(){return (head->next = =NULL);} //判断是否为空表
    void Prev( );                           //当前位置指针curr前移到前驱
    void Next( );                           //当前位置指针curr后移到后继
    bool setPos(int pos);                   //任意指定当前数据元素的位置
    bool Insert(Elem x);                    //在当前位置curr之后插入元素
    bool remove(Elem &e);                   //删除当前位置curr之后的元素
    void clear() { removeall(); init(); } //清空
    void Print();                           //输出
}
```

单链表上所实现的基本操作：

1. 读取数据元素

在单链表中，任何两个数据元素的存储位置之间没有固定的联系。但是，每个元素的存储位置都包含在其直接前驱结点的信息之中。

假设p是指向线性表中第i个数据元素（结点a_i）的指针，则p->next是指向第i+1个数据元素（结点a_{i+1}）的指针，即p->data=a_i，p->next->data= a_{i+1}。

算法2.8：读取数据元素

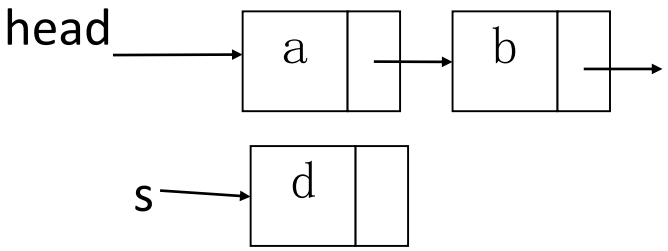
```
bool LList:: getValue(Elem &e) { //读取当前位置curr的值
    if ( curr==NULL || head -> next = = NULL) return false;
    e = curr->element;
    return true;
}
```

这个算法的时间复杂度为O(1)。

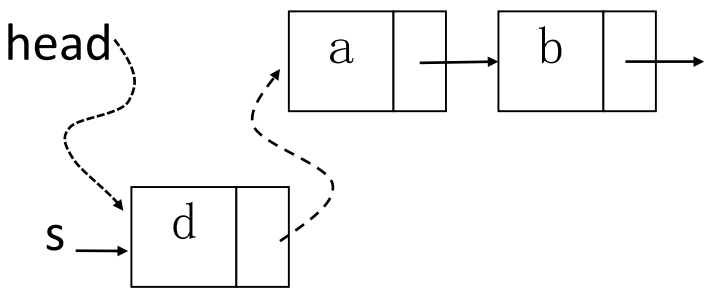
2. 单链表的插入

假设在**不带头结点**的单链表中插入结点d，s为指向结点d的指针，则有以下两种情况：

（1）将结点d插入到链表的头部。



(a) 插入前的单链表

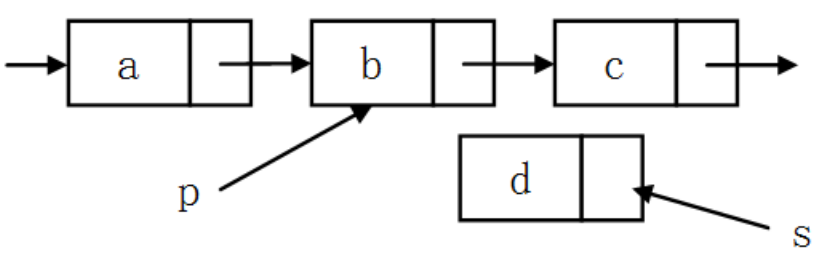


(b) 插入后的单链表

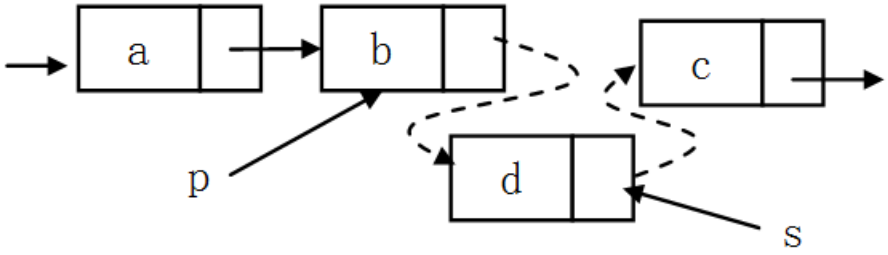
指针的修改用语句描述为：

```
s->next = head;
head= s;
```

（2）将结点d插入到某个结点之后，如插在b和c两个结点之间。



(a) 插入前的单链表



(b) 插入后的单链表

指针的修改用语句描述为：

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

插入算法的实现如下：

算法2.9：插入算法 （思考：头上插入、尾部插入算法？）

```
void LList :: Insert ( Elem x) {//在当前结点之后插入结点x
    Link *s = new Link;
    if(!s) { count<<"空间分配失败"<< end1；return false; }
    s->element = x;
    s-> next = curr-> next;
    curr-> next = s;
    return true;
}
```

3. 单链表的删除

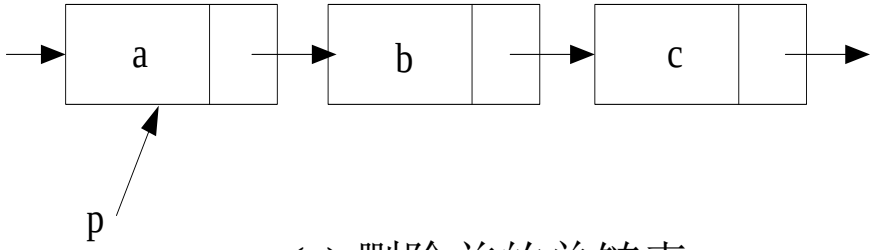
- 如要删除图中结点b，仅需修改结点a中的指针域即可。
- 如果指针p指向元素值为a的结点，则修改指针的语句为：

q = p-> next;

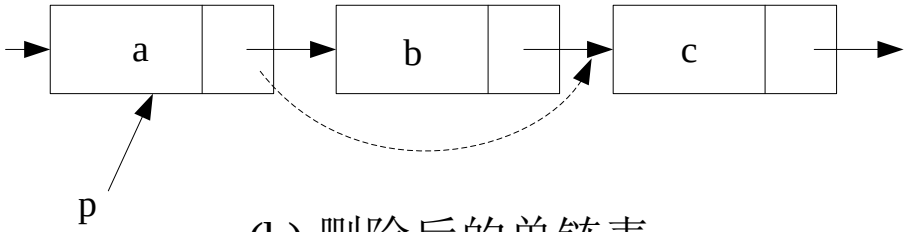
p-> next = p-> next-> next;

或p-> next = q-> next;

free(q);



(a) 删除前的单链表



(b) 删除后的单链表

单链表的删除算法如下：

算法2.10：单链表的删除（思考：删除首元素？）

```
bool LList :: remove( Elem &e ) {//删除当前结点之后的元素
    if ( curr->next == NULL) return false;      //返回失败
    Link *q = curr->next;                       //暂存删除节点指针
    curr->next = q->next;                       //删除结点链表关系
    e = q->element;
    delete q;
    return true;
}
```

4. 线性表实现方法的比较

（1）在空间上

顺序表的优点是对于表中的每一个元素没有浪费空间，而链表需要在每个结点上附加一个指针。如果结点的数据域占据的空间较小，则链表的结构性开销就占去了整个存储空间的大部分。当顺序表被填满时，存储上没有结构性开销。在这种情况下，顺序表有更高的空间效率。

当线性表元素数目变化较大或者未知时，最好使用链表实现。而如果用户事先知道线性表的大致长度，使用顺序表的空间效率会更高。

（2）在访问上

像取出线性表中第i个元素这样的按位置的随机访问，使用顺序表更快一些；通过前驱和后继可以很容易调整当前位置向前或者向后，这两种操作需要的时间复杂度为O(1)。

相比之下，单链表不能直接访问表中任意的第i个元素，按位置访问只能从表头开始，直到找到指定的位置。这两种操作需要的平均时间复杂度和最差时间复杂度均为O(n)。

（3）插入和删除操作上

给出指向链表中合适位置的指针后，只是对部分结点指针进行更新，插入和删除函数所需要的时间仅为O(1)。而顺序表必须在数组内将其余的元素向前或者向后移动，这种方法所需要的平均时间和最差时间均为O(n)。

对于许多应用，插入和删除是最主要的操作，因此它们的时间效率是很重要的。仅就这个原因而言，链表经常比顺序表效率更高。

5. 单链表的改进方法

（1）空闲链（Freelists），设立一个空闲链管理空闲的结点空间。

（2）静态链表（Static Linked List），利用一组地址连续的内存空间来描述线性链表，是顺序表和链表两者的结合，把数组元素作为存储结点，数组元素类型包含数值域data和游标指示器cur。游标定义为整型，指示结点在数组中的相对位置。

在静态链表中，插入与删除元素的算法为修改游标，与单链表中要通过修改指针实现插入与删除操作不同的是，new和deletefree两个函数使用的是静态链表本身的已声明的空间，即静态链表中未使用的部分，静态链表的这个部分称为“备用链表”。

静态链表的方法：

（1）结点空间必须预先分配；

（2）元素之间的位置变化反映在表示关系的游标值的变化。

next指针是数组中元素的下标索引号；

（3）要同时管理已建立的链表和空闲链。

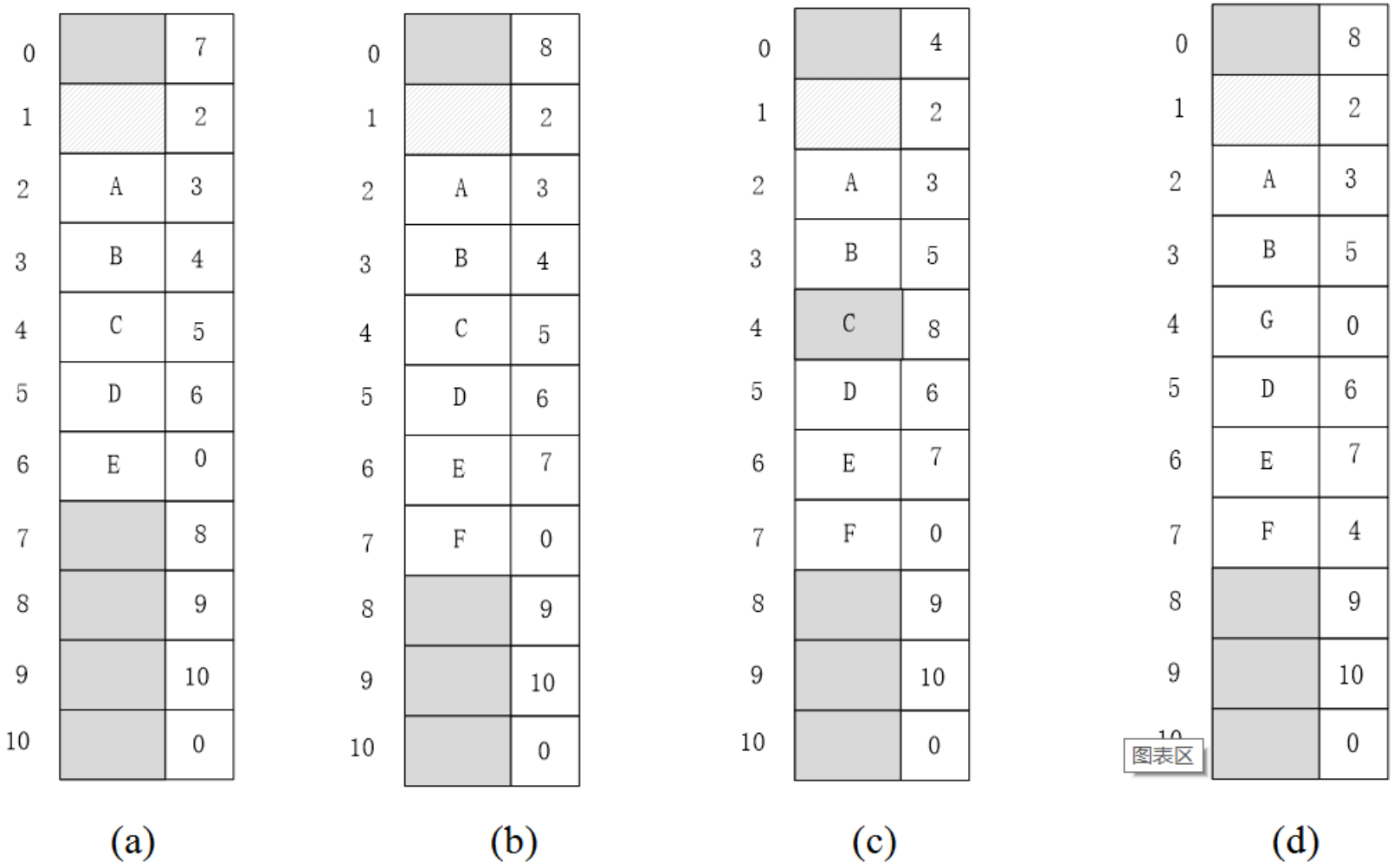

0
1
2 A 3
3 B 4
4 C 5
5 D 6
6 E 0
7 8
8 9
9 10
10 0
(a)
0
1
2 A 3
3 B 4
4 C 5
5 D 6
6 E 7
7 F 0
8 9
9 10
10 0
(b)
0
1
2 A 3
3 B 5
4 C 8
5 D 6
6 E 7
7 F 0
8 9
9 10
10 0
(c)
0
1
2 A 3
3 B 5
4 G 0
5 D 6
6 E 7
7 F 4
8 9
9 10
10 0
图表区
(d)

图2-12 静态链表示例

例：有一组连续的内存空间，用Array[0..11]来表示。其中，

①Array[0]为头结点，其游标指示器指示空闲链的第一个结点，游标值为7，Array[1]为已建立的线性链表的头结点，Array[2..6]分放了5个数据元素，如图2-12(a)所示。

②在静态链表末尾插入数据元素F，将Array[0]所指的空闲链的第一个结点的游标值修改为8，其存储变化情况如图2-12(b)所示。

③将静态链表中的数据元素C删除，Array[0]的游标值修改为4，其存储变化情况如图2-12(c)所示。

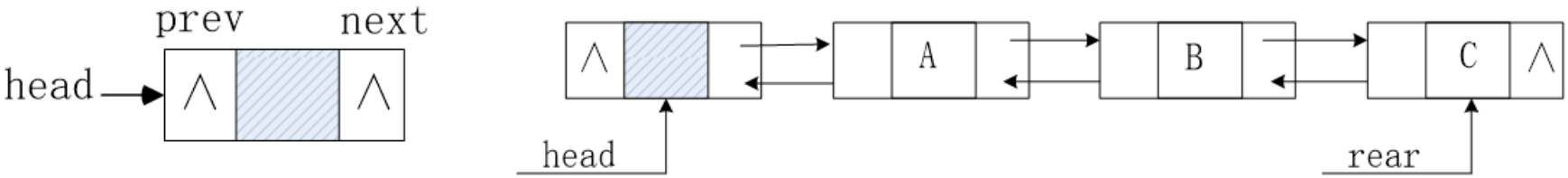
④在静态链表中的插入数据元素G，Array[0]的游标值修改为8，其存储变化情况如图2-12(d)所示。

2.3.2 双向链表

双向链表的每个结点包含**一个数据域和两个指针域**，其中一个指针为前驱指针prev，指向它的前驱结点；另一个指针为后继指针next，指向它的后继结点。如图2-13所示。简化

图2-13 双向链表结点结构

对于双向链表，若要查找一个结点的前驱结点，可以很容易通过前驱指针prev找到。双向链表通常为双向循环链表。这样，无论是插入还是删除，对链表中的第一个结点、最后一个结点和中间任意结点的操作过程相同。图2-14是带头结点的双向链表。



（a）空表　　(b) 非空的双向链表

图2-14 带头结点的双向链表

算法2.11：双向链表的类定义

```
template <class Elem>
class DLink {              //双向链表的结点类定义
public:
    Elem element;
    DLink *prev;
    DLink *next;

    DLink (const Elem& it, DLink *p=NULL, DLink *n=NULL)
          { element = it; prev = p; next = n;}
    DLink (DLink *p=NULL ,DLink *n=NULL)
          {prev = p; next = n;}
    ~ DLink();
}
```

```
template <class Elem>
class DList : public List<Elem> {        //双向链表类的定义
private:
    DLink *head, *tail, *curr;  // 双向链表的头指针、尾指针、当前结点
public:
    DList ( ) { curr= tail = head = new DLink<Elem>; //创建一个头结点
    }
    ~ DList(){
        while(head != NULL){          // 释放所有的结点
                curr=head;  head=head->next; delete curr;
        }
    }
    void Prev( );                     //当前位置指针curr前移到前驱
    void Next( );                     //当前位置指针curr后移到后继
    bool setPos(int pos);             //任意指定当前数据元素的位置
    void Create(int n);               //创建长度为n的双链表
    bool getValue(Elem &x);           //取表中当前位置元素的值
    bool insert( const Elem x);       //在当前结点之后插入元素
    bool remove( Elem &x);            //删除当前结点之后的元素
}
```

双向链表是一种对称结构，若p为指向表中某一结点的指针，则有：

$$p->next->prev = p->prev->next = p$$

双向链表与单链表在实现插入或删除操作时有很大的不同，因为在双向链表中实施插入或删除操作时，不但要修改指向其直接后继的指针next，而且还要修改指向其直接前驱的指针prev。

（1）在当前结点之后插入一个新结点，如图2-15所示。

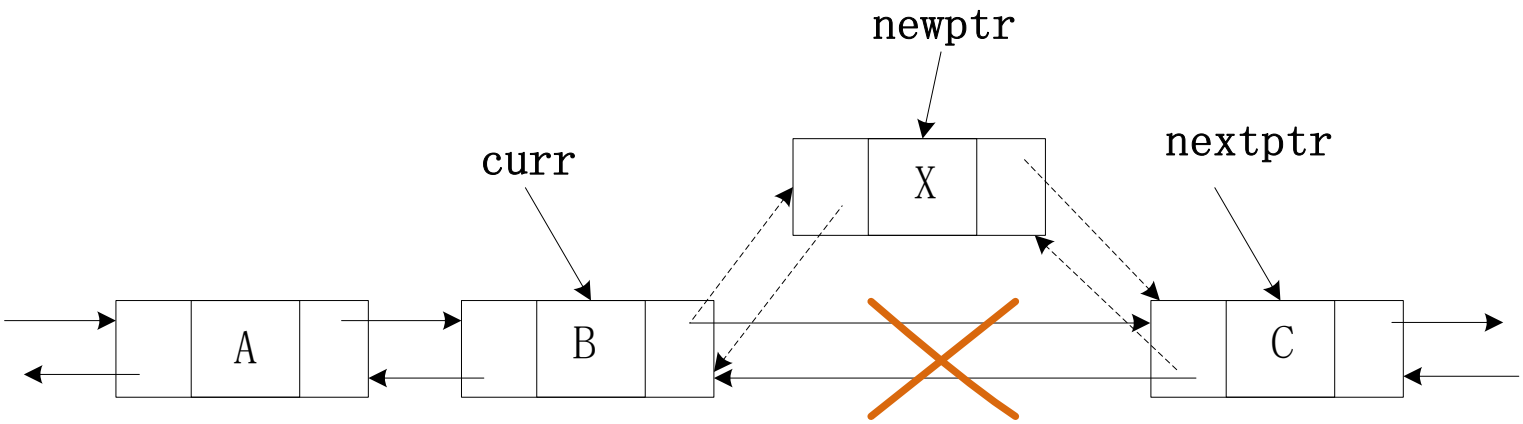


图2-15 当前结点之后插入一个新结点

指针的修改用语句描述为：

```
nextptr  = curr->next;
newptr ->next = nextptr;
newptr ->prev = curr;
curr->next       = newptr;
nextptr->prev  = newptr;
```

算法2.12：双向链表的插入算法

```
template <class Elem> //双向链表在当前结点之后插入元素
bool DList<Elem>::Insert( const Elem x){
    DLink *nextptr, *newptr;
    nextptr = curr->next;
    newptr = new DLink<Elem>(x, curr, nextptr);
    if (newptr == NULL) return false;
    curr->next = newptr;
    nextptr->prev = newptr;
    return true;
}
```

（2）在当前结点之前插入一个新结点，如图2-16所示。

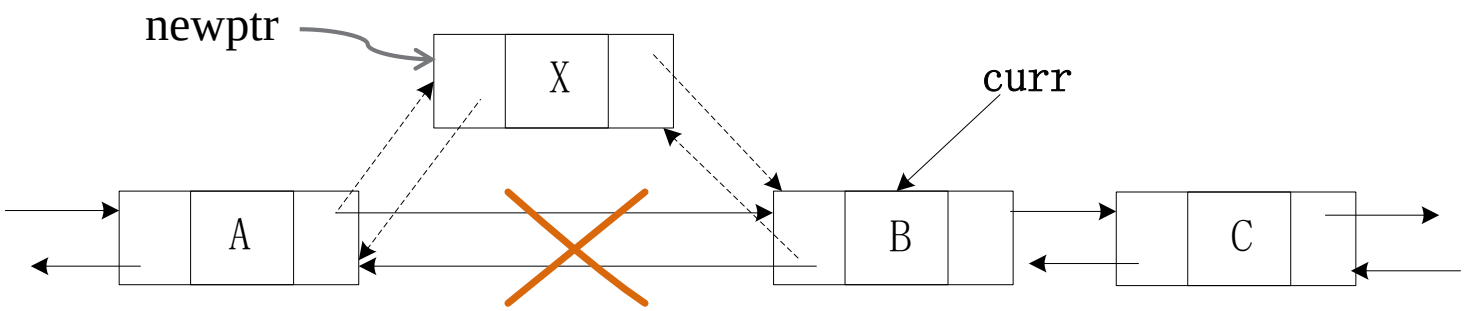


图2-16 当前结点之前插入一个新结点

指针的修改用语句描述为：

```
p = curr->prev;
newptr ->next = curr;
newptr ->prev = p;
p->next      = newptr;
curr->prev = newptr;
```

（3）删除当前结点，如图2-17所示。

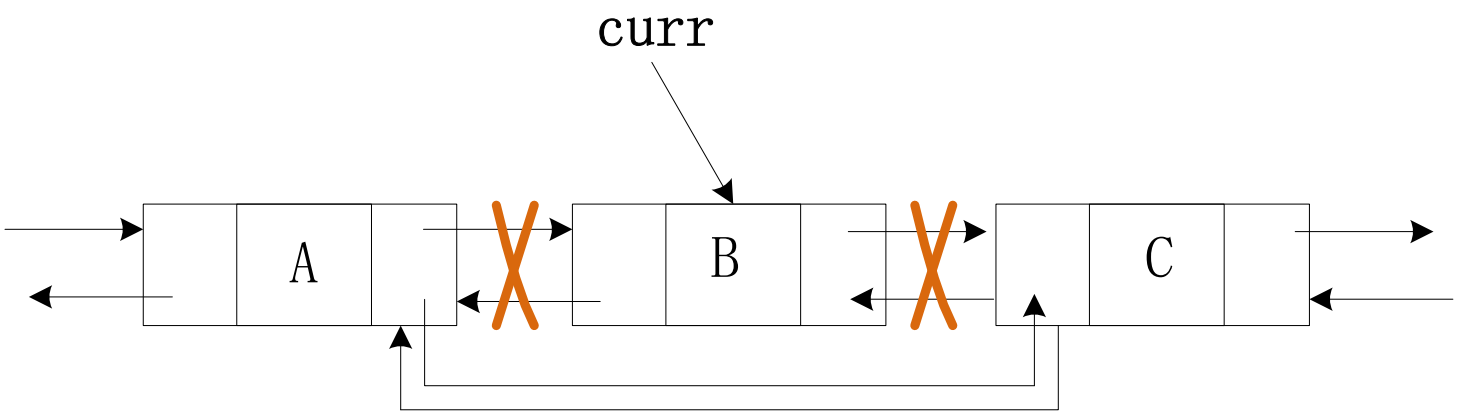


图2-17 删除当前结点

指针的修改用语句描述为：

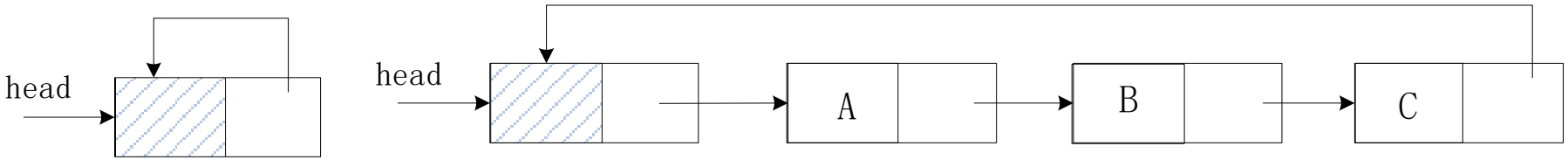
```
p = curr;
curr->prev->next = curr->next;
curr->next->prev = curr->prev;
curr = curr->prev;
free(p);
```

算法2.13：双向链表的删除

```
template <class Elem>  //双链表中删除当前结点元素
bool DList<Elem> :: remove( Elem &x ){
    if ( curr==NULL || head->next == NULL) return false;
    DLink *temp=curr;
    curr->prev->next = curr->next;
    curr->next->prev = curr->prev;
    curr = curr->next;
    x=temp->element;
    delete temp;
    return true;
}
```

2.3.3 循环链表

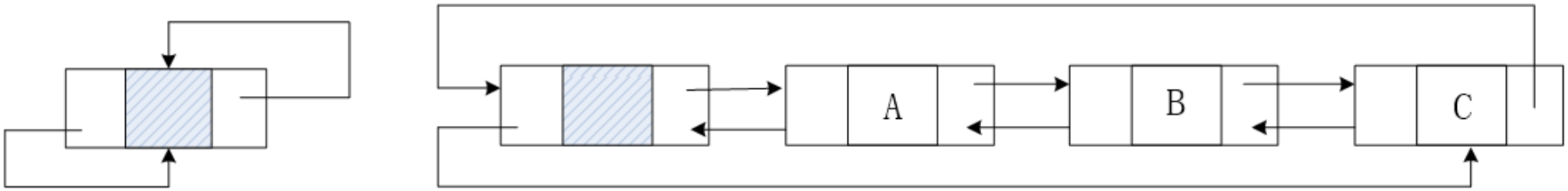
循环链表是线性链表的一种变形。在线性链表中，每个结点的指针都指向它的下一个结点，最后一个结点的指针域为空，表示链表结束。而循环链表则将表中最后一个结点的指针域指向头结点，整个链表形成一个环。由此，从表中任一结点出发均可找到表中其它结点。如图2-18所示为单向循环链表。



(a) 空表　　(b) 非空表

图2-18 单向循环链表

图2-19所示为双向循环链表。



(a)空表　　(b) 非空表

注：循环链表的定义和操作与单链表相似，只是循环结束条件有所不同，单链表中算法的循环条件是判定p或p->next是否为空，而在循环链表算法中的循环条件是判定p或p->next是否等于头指针head。

算法2.14：单循环链表的类定义

```
template <class Elem>
class CLinkNode {      //单循环链表的结点类定义
    Elem element;
    CLinkNode<Elem> *link;

    CLinkNode(CLinkNode<Elem> *next = NULL) {link=next;}
    CLinkNode(Elem d, CLinkNode<Elem> *next = NULL)
    :element(d), link(next){ }
};
```

```
template <class Elem>
class CList : public List<Elem> {        //带头结点的单循环链表类的定义
private:
    CLinkNode<Elem> *head, *tail, *curr;  //头指针, 尾指针, 当前位置
public:
    CList(){          //构造函数
        curr= tail = head = new DLink<Elem>; //创建一个头结点
        head->link = head
    }
    CList(CList<Elem> &L);              //复制构造函数
    ~CList();                           //析构函数
    int Length();                       //计算循环链表长度
    bool IsEmpty() {return head->link == head ? true : false; }//判表空否
    void Prev( );                       //当前位置指针curr前移到前驱
    void Next( );                       //当前位置指针curr后移到后继
    bool setPos(int pos);               //任意设定当前数据元素的位置
    bool getValue(Elem &x);             //取出当前位置元素的值
    bool insert( Elem x);               //在当前位置之后插入元素
    bool remove( Elem &x);              //删除当前位置之后的元素, 并返回其值
    CLinkNode<Elem> * Search(Elem x);  //搜索含数据x 的元素
};
```

约瑟夫问题是循环链表实际应用的一个典型例子。

约瑟夫问题：假设编号为1，2，…，n(n＞0)的n个人，按顺时针方向围坐一圈，每人持有一个正整数密码。开始时先选一个正整数m，从第1个人开始，自1起顺序报数，报到m时停止报数，报到m的人出列，将他的密码作为新的m值，然后从出列人的下一个人重新从1顺序报数。如此下去，直到所有人全部出列为止。

根据问题的描述，采用循环链表结构来表示约瑟夫问题，结点的定义如图2-20所示。

No	pwd	next

图2-20 约瑟夫问题的结点结构

其中： No表示每个人的编号；
pwd表示每个人的密码；
next为指向下一个结点的指针。

解决约瑟夫问题的步骤如下：

（1）建立一个带头结点head的具有n个结点的约瑟夫问题循环链表；

（2）在循环链表中查找、输出和删除密码为m的结点。具体来说，在带头结点的约瑟夫问题循环链表中，循环查找密码为m的结点，将其输出，并取出该接点的密码赋值给m，最后将该结点从约瑟夫问题循环链表中删除。直到输出循环链表中的所有元素为止。

创建约瑟夫问题的循环链表的具体实现算法如下。

算法2.15： 约瑟夫问题的循环链表

```
#include <iostream.h>
#include “CList.h”
template <T>
void Josephus(CircList<T>& Js, int n, int m) {
    CLinkNode<T> *p = Js.getHead(), *pre = NULL;
    int i, j;
    for (i = 0;  i < n;  i++ ) {//执行n 次
        for (j = 1;  j < m;  j++){ //数m-1 个人
            pre = p;
            p = p-> next;
        }
        cout << “出列的人是” << p -> no << endl;  //输出出列人的编号
        m = p -> pwd; //将出列人的编号赋给m
        pre->next = p->next;  delete p;  //删去
        p = pre->next;
    }
}
```

```
void main() {
    CList<int> clist;
    int i, n, m;
    cout << "输入游戏者人数和报数间隔:"; cin >> n >> m;
    for (i = 1; i <= n; i++ ) clist.insert(i); //形成约瑟夫环
    Josephus(clist, n, m); //解决约瑟夫问题
}
```

循环链表的优点：

（1）从表中任一结点出发均可访问到表中其它结点，这使得某些操作在循环链表上容易实现；

（2）插入、删除操作中不需区分尾结点还是中间结点，使操作简化。

2 .4 线性表应用举例

2.4.1 —元多项式的表示

多项式的表示和求和是线性表应用的典型实例。在数学上，一元多项式的书写形式为：

$$P_n(x) = P_0(x^0) + P_1(x^1) + P_2(x^2) \ldots + P_n(x^n)$$

对于多项式中所有项的系数可用一个线性表来表示：

$$L = \{p_0, p_1, p_2, \ldots p_n\}$$

将n次多项式的次数考虑进去，则可采用两种线性表表示方法：

（1）把变量x的指数i隐含在系数p_i中，设$Q_1=\{p_0,p_1,p_2,\cdots p_n\}$；

（2）把变量x的指数e_i和系数p_i分开来描述，设$Q_2=\{(p_0,e_0),(p_1,e_1),(p_2,e_2),\cdots(p_m,e_m)\}$，其中$m\leq n$。线性表可以使用顺序存储结构或链式存储结构来存储，从而进行多项式的相关运算。

若采用顺序存储结构，可使用一个一维数组存储线性表Q_1，数组中存储的数据元素为多项式的系数，数组下标表示相应的次数。多项式$P_n(x)$的顺序存储结构如图2-21所示。

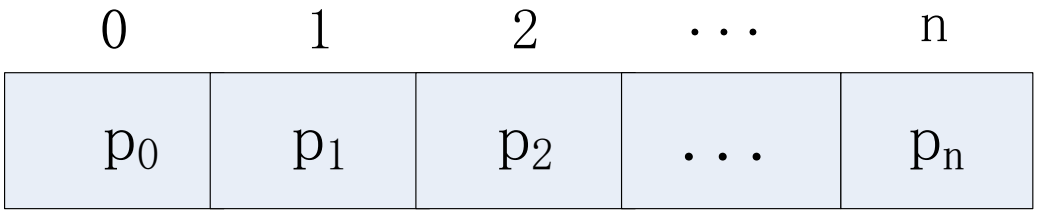


图2-21 多项式Pn(x)的顺序存储结构

算法2.16：多项式的顺序存储结构类定义

```
Const int PolySize = 100;
Class Poly{
    Elem  element[PolySize];
    int size;
    public:
        Poly(){size = 0;} //构造一个空多项式
        ~Poly();
        void Clear() {size = 0;}//清空
        Poly sum(Poly  p); //多项式相加
        Poly sub(Poly  p); //多项式相减
}
```

通常多项式的次数可能很高，非零项数目却很少，如多项式：

$$P(x) = 1+3x^{10}+6x^{1000}+9x^{100000}$$

如用一维数组来存放，只有四个非零项，其余均为0，因此导致存储空间的极大浪费。因此，一般采用链式存储结构来描述一元多项式，将线性表用Q_2表示。线性表的每个元素包括两个数据成员：coef（系数）和exp（指数）。对于稀疏多项式（次数较大，非零项个数较少）的多项式通常采用这种链式存储结构。上述多项式P(x)用线性链表表示如图2-22所示。

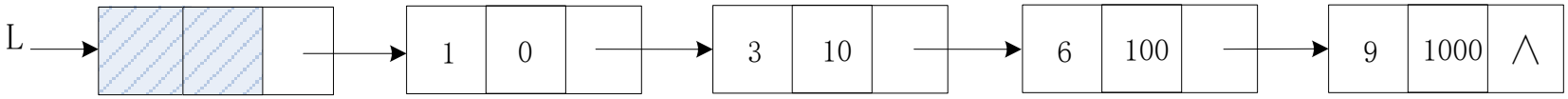


图2-22 多项式P(x)的链式存储结构

算法2.17：多项式P(x)用线性链表

```
struct term{  //数据部分定义
        double  coef;
        int  exp;
};
struct PNode{    // 链表结构定义
        term data;
        PNode *next;
};
class  Poly{
PNode * phead;
public:
        Poly(){phead = new PNode;  phead->next=NULL;}
        ~Poly(){delete phead;}
        void clear(){phead->next =NULL;}
        Poly PolyAdd(Poly Pb);
        void print();
};
```

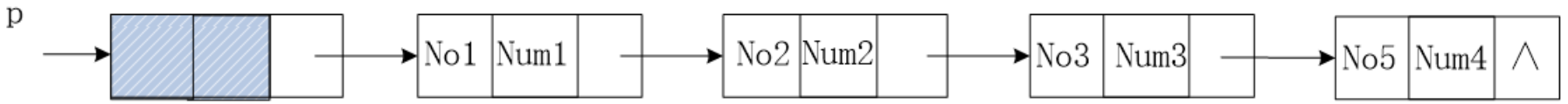
2.4.2 商品链更新

某仓库中各商品的库存数量按商品编号从小到大存储在一个带头结点的单链表中。链表的结点由商品编号(No)、数量(Num)和链指针(next)三个域组成如下：

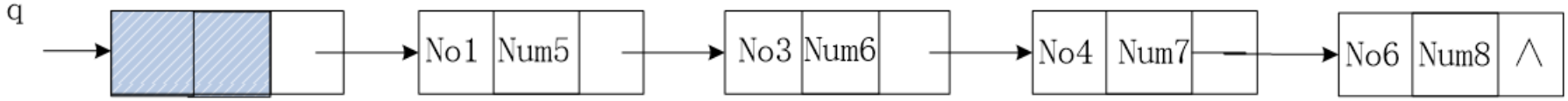
No	Num	next

现新进一批商品需要入库。在这些入库商品中，有部分商品是库存中已有商品，其商品是新增商品。若各入库商品的数量也是按商品编号从小到存储在一个带头结点的单链表中，根据入库商品的数量更新各库存商品的数量。

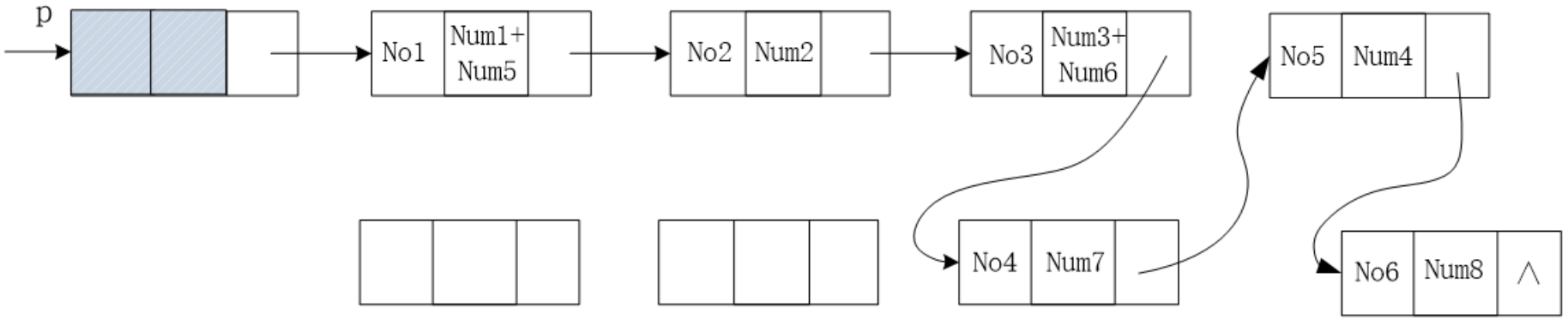
已有商品用单链表L表示，其结构如图2-23所示：



新进商品用单链表Lin表示，其结构如图2-24所示：



则入库后商品更新链表Lnew表示，其结构如图2-25所示：



算法2.18： 商品更新链表

```
typedef struct Node {
    int No, Num;
    struct Node *next;
}  Node, *Llist;
void update(Llist L, Llist Lin ){ //L为库存链表，Lin为入库链表
    Llist  p,q;
    p=L;
    while (Lin->next!=NULL ){
        q=Lin->next;
        while (p->next!=NULL && p->next->NO<q->NO) p=p->next;
        if (p->next==NULL) { p->next=q; free(Lin); return; }
        p=p->next; Lin->next=q->next;
        if (p->NO==q->NO) { p->Num+=q->Num; free(q); }
        else { q->next=p->next; p->next=q; p=q;}
    }
    free(Lin);
    return;
}   //结束update()
```