数据结构—C++实现

沈 俊

jshen@shu.edu.cn

上海大学 计算机工程与科学学院

2021年12月

第3章　线 性 表

- 线性表的定义
- 线性表的顺序表示
- 线性表的链表表示
- 线性表的应用

3.1线性表的定义

线性表(linear-list)是最常用最简单的一种数据结构。一个线性表是n（n≥0)个相同类型数据元素的有限序列。记为： L=（a_1，a_2，…，a_n）。

其中，L是表名，a_1是第一个数据元素（也简称为首元素），无前驱，只有一个后继；a_n是最后一个数据元素(即第n个数据元素)，只有一个前驱，无后继。其余的每个数据元素a_i（i=2,3，…,n-1)都只有一个前驱，且只有一个后继。i（i=1,2，…,n)称为表中元素序号。n是数据元素的个数，也称为表的长度，若n=0，L称作空表。

线性表的例子

- 某班级学生的数据库课程的成绩：

  (72，65，83，94，87，98，57)

- 某车间职工的编号：

  (“0108”，“0110”，“0122”，"0132"，"0718")

在复杂的线性表中，一个数据元素可能是由若干个**数据项**组成的。

例如：在例1-1给出的“人事登记表”中，每一个职工的信息就是一个数据元素，它是由“编号”、“姓名”、“性别”、“出生日期”、“婚否”和“基本工资”六个数据项组成的。

表 1-1　人事登记表

编号	姓名	性别	出生日期	婚否	基本工资
0001	王　军	男	1960/5/30	已	650
0002	李　平	女	1953/6/2	已	710
0003	周丽娟	女	1948/7/8	已	980
0004	赵忠良	男	1950/12/2	已	950
0005	张国庆	男	1978/10/1	未	500
⋮	⋮	⋮	⋮	⋮	⋮

线性表的基本操作

(1) 初始化
(2) 求长度
(3) 取指定位置的元素
(4) 元素定位
(5) 修改指定元素的值
(6) 插入元素
(7) 删除元素
(8) 判是否为空表
(9) 表清空

3.2 线性表的顺序表示

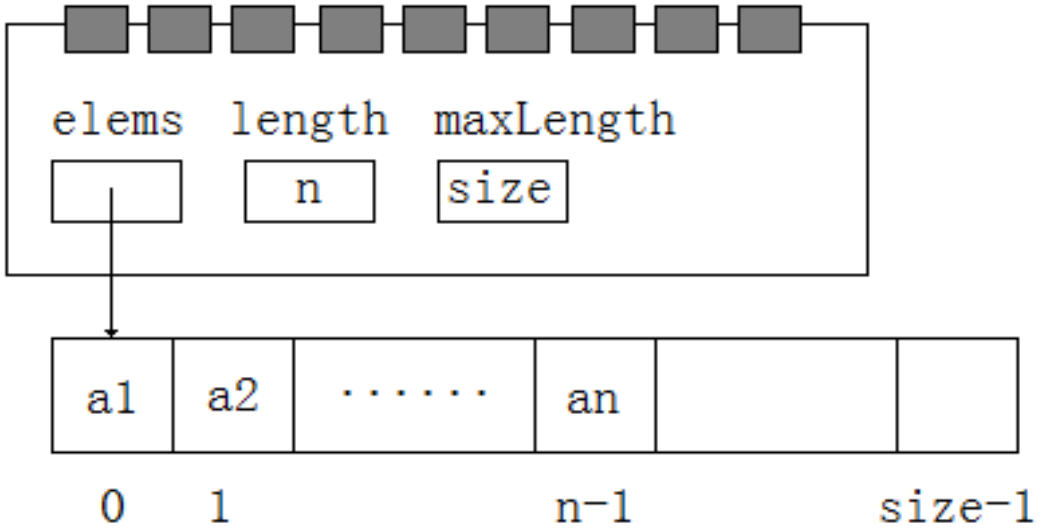
线性表的顺序存储方式是：用一组连续的有限空间依次存储线性表中的数据元素，简称为顺序表。

顺序表的特点是：

- 一块地址连续的空间存放线性表中的数据元素。
- 任意两个逻辑上相邻的数据元素在物理上也必然相邻。
- 顺序表可以随机访问。

顺序表的类定义与实现

```
template <class ElemType>
class SeqList {
protected:
        int length;
        int maxLength;
        ElemType *elems;
```



顺序表的类定义与实现

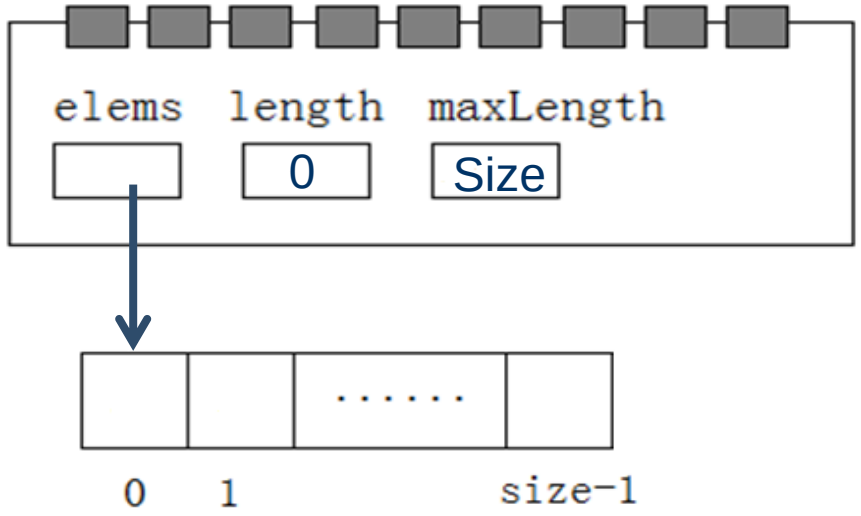
```
public:
    SeqList(int size=DEFAULT_SIZE);
    SeqList(ElemType v[], int n, int size=DEFAULT_SIZE);
    virtual ~SeqList();
    int GetLength() const;
    bool IsEmpty() const;
    void Clear();
    void Traverse(void (*Visit)(const ElemType &)) const;
    int LocateElem(const ElemType &e);
```

顺序表的类定义与实现

```
    Status GetElem(int i, ElemType &e) const;
    Status SetElem(int i, const ElemType &e);
    Status DeleteElem(int i, ElemType &e);
    Status InsertElem(int i, const ElemType &e);
    Status InsertElem(const ElemType &e);
    SeqList(const SeqList<ElemType> &sa);
    SeqList<ElemType> &operator =(const
            SeqList<ElemType>  &sa);
};
```

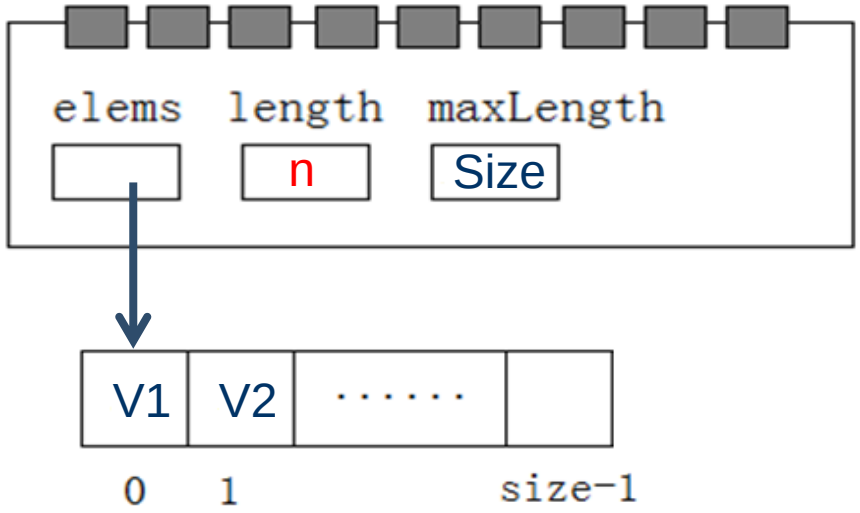
（1）构造空顺序表

```
template <class ElemType>
SeqList<ElemType>::SeqList(int size)
{
        elems=new ElemType[size];
        assert(elems);
        maxLength=size;
        length=0;
}
```



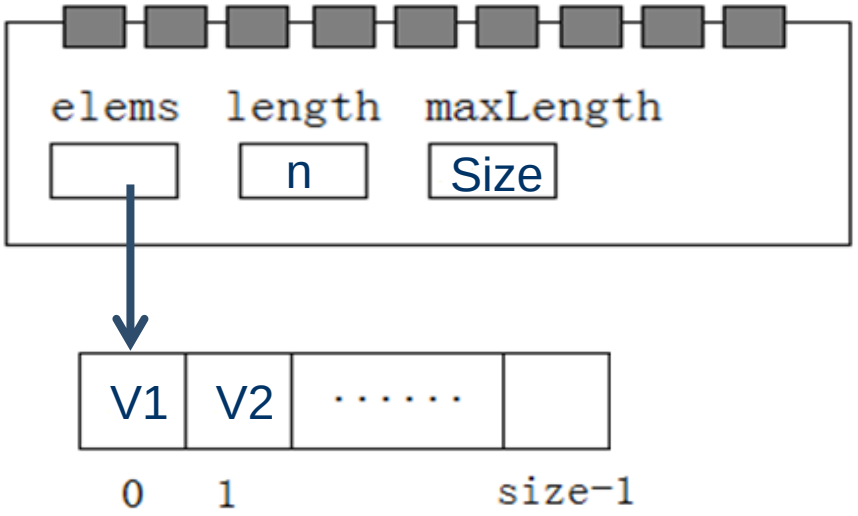
（2）根据数组内容构造顺序表

```
template <class ElemType>
SeqList<ElemType>::SeqList(ElemType v[], int n, int size) {
        elems=new ElemType[size];
        assert(elems);
        maxLength=size;
        length=n;
        for (int i=0; i < length; i++)
                elems[i]=v[i];
}
```



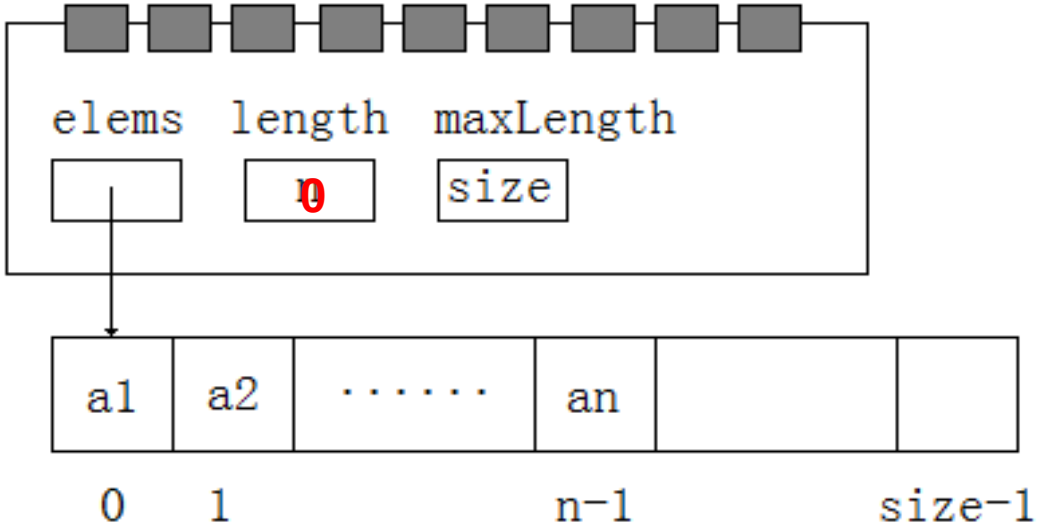
（3）析构函数

```
template <class ElemType>
SeqList<ElemType>::~SeqList()
{
        delete []elems;
}
```



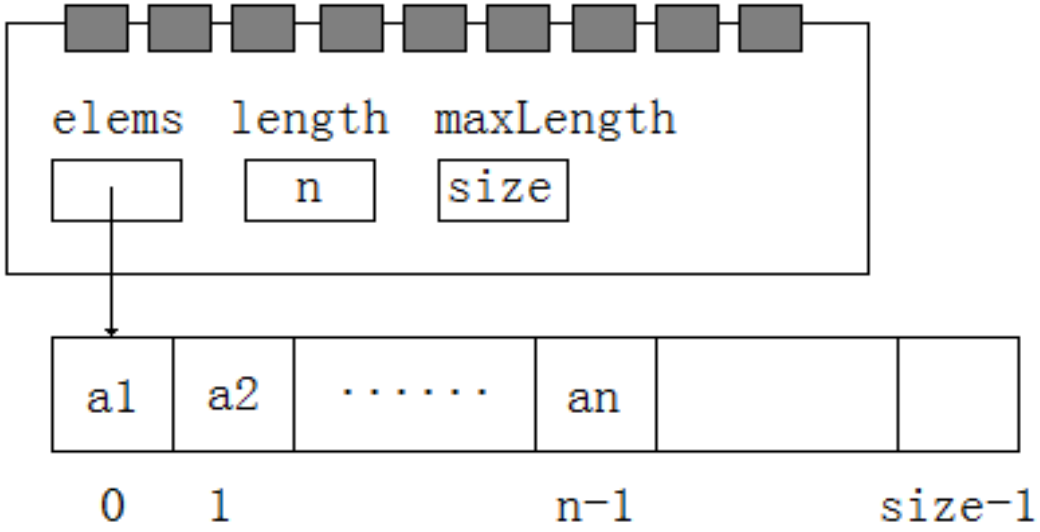
（4）清空顺序表

```
template <class ElemType>
void SqList<ElemType>::Clear()
{
        length = 0;
}
```



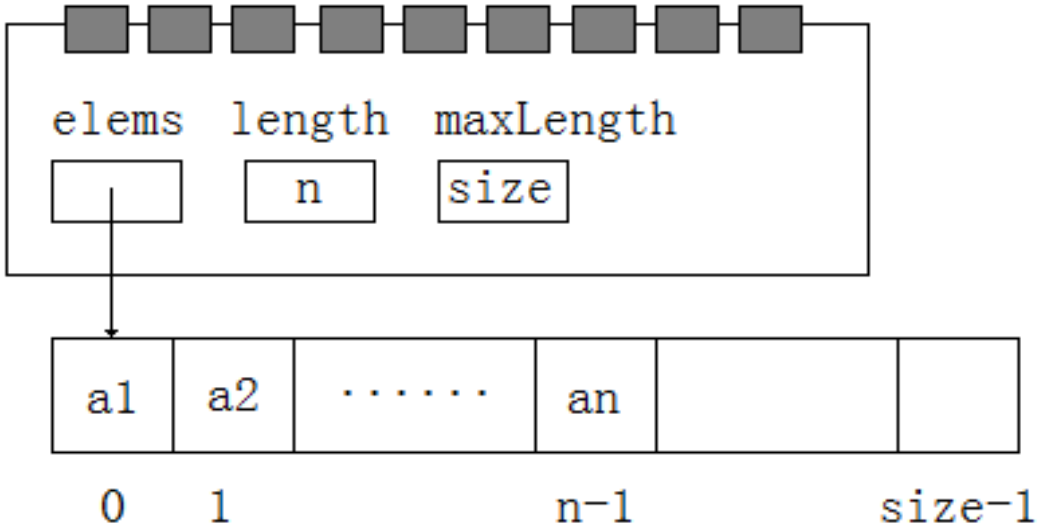
（5）遍历顺序表

```
template <class ElemType>
void SqList<ElemType>::Traverse(void (*visit)(const
    ElemType &)) const  {
    for (int i = 1; i <= length; i++)
            (*visit)(elems[i-1]);
}
```



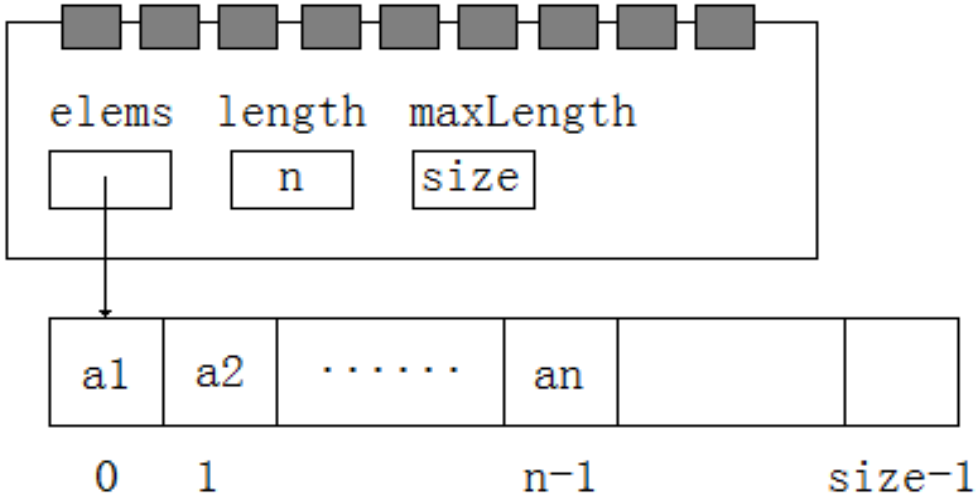
（6）定位函数

```
template <class ElemType>
int SeqList<ElemType>::LocateElem(const ElemType &e) const
{
        int i=0;
        while (i < length && elems[i] != e)        i++;
        return i<length ? i+1 : 0;
}
```



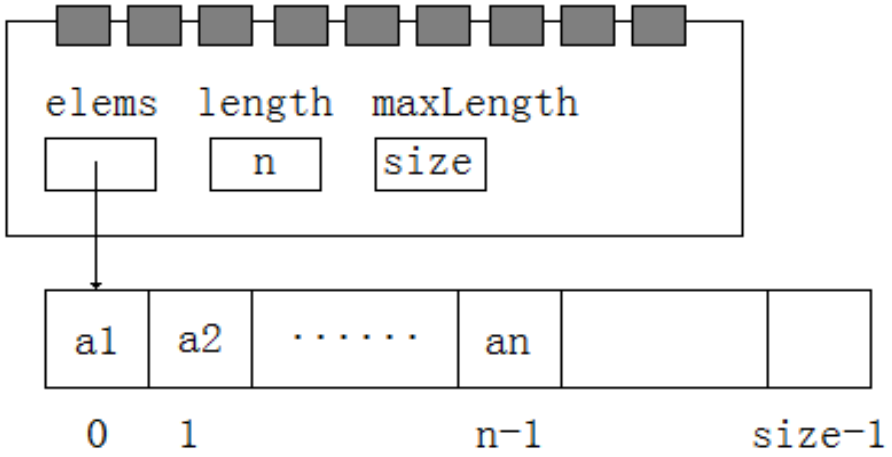
（7）取指定元素的值

```
template <class ElemType>
Status SeqList<ElemType>::GetElem(int i, ElemType &e) const{
        if (i < 1 || i > length)  return NOT_PRESENT;
        else    {
                e=elems[i - 1];
                return ENTRY_FOUND;
        }
}
```



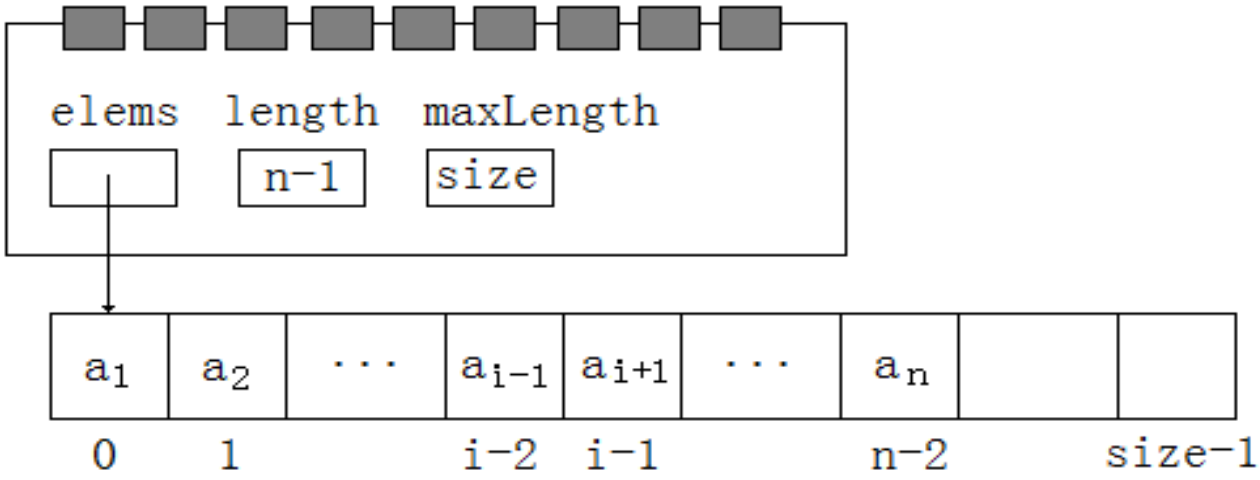
（8）修改指定元素的值

```
template <class ElemType>
Status SeqList<ElemType>::SetElem(int i, const ElemType &e){
        if (i < 1 || i > length)  return RANGE_ERROR;
        else    {
                elems[i - 1]=e;
                return SUCCESS;
        }
}
```



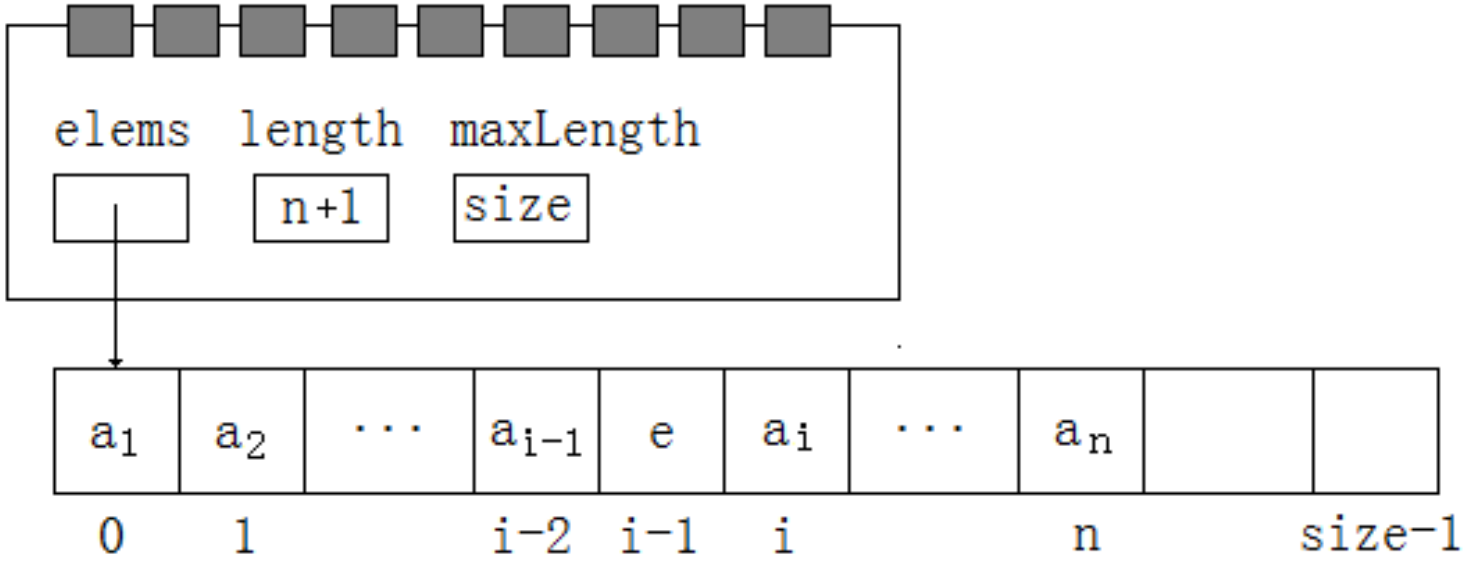
（9）删除指定元素

```
template <class ElemType>
Status SeqList<ElemType>::DeleteElem(int i, ElemType &e)
{
        if (i < 1 || i > length)
                return RANGE_ERROR;
        else    {
                e=elems[i - 1];
                for (int j=i; j<length; j++)
        }
}
```



（10）在任意位置插入元素

```
template <class ElemType>
Status SeqList<ElemType>::InsertElem(int i, const ElemType &e){
    if (length == maxLength)    return OVER_FLOW;
    else if (i < 1 || i > length + 1)    return RANGE_ERROR;
    e
    }
}
```

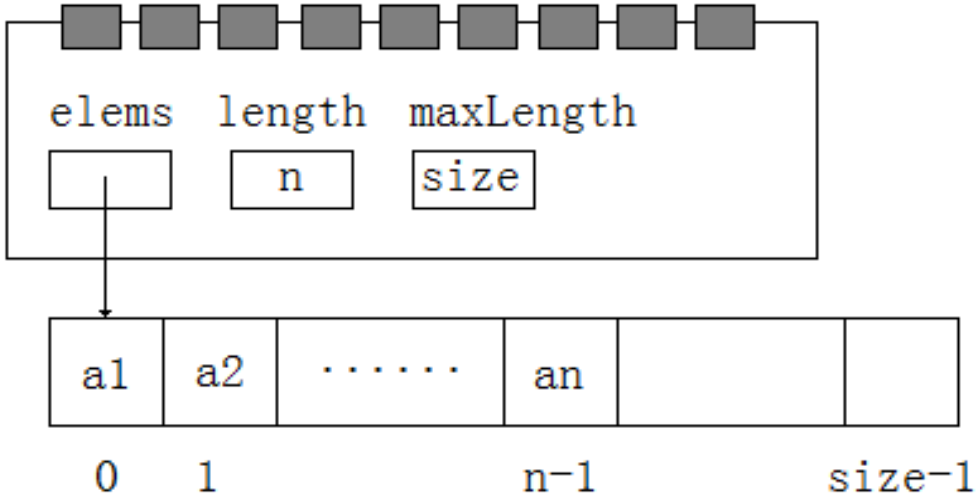


（11）在表尾插入元素

```
template <class ElemType>
Status SeqList<ElemType>::InsertElem(const ElemType &e)
{
        if (length==maxLength)
                return OVER_FLOW;
        else    {
                elems[length]=e;
                length++;
                return SUCCESS;
        }
}
```

（12）判断顺序表是否为空

```
template <class ElemType>
bool SqList<ElemType>::IsEmpty() const
{
        return length == 0;
}
```



顺序表插入、删除算法的复杂度分析

举例来看顺序表上的插入和删除。

在原来已有7个元素的表的第4个元素前插入数据元素x=24的过程。

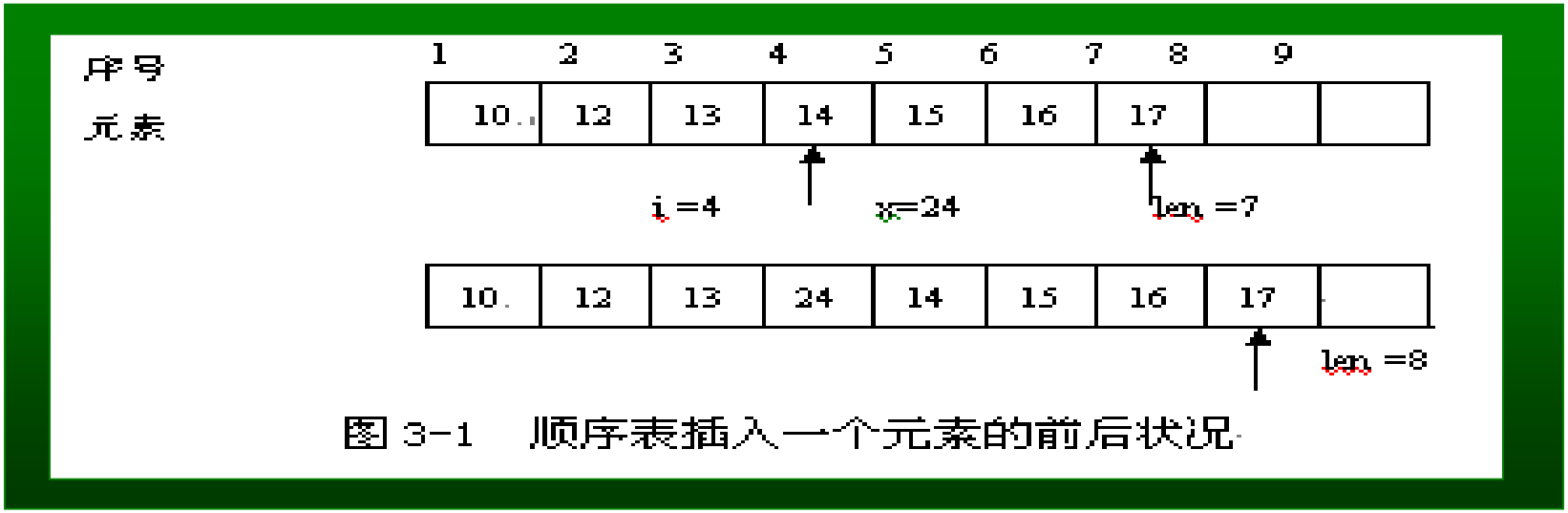


图 3-1 顺序表插入一个元素的前后状况

顺序表插入、删除算法的复杂度分析

图3-2是在原来已有8个元素的顺序表中删除第4个元素的过程。

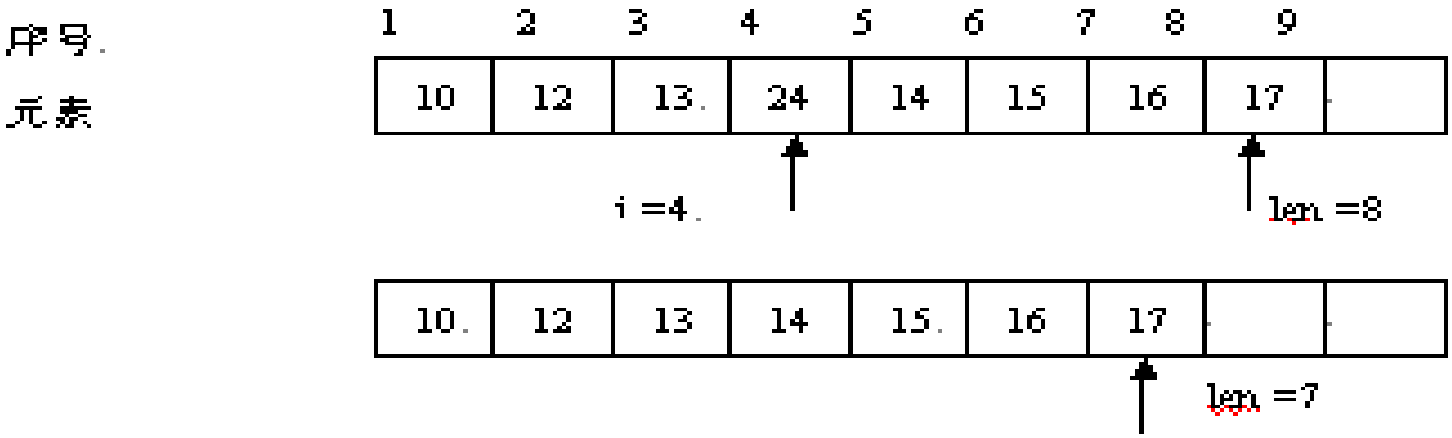


图 3-2　顺序表删除一个元素的前后状况

顺序表插入、删除算法的复杂度分析

在顺序表中第i个数据元素之前插入一个元素时，需要移动n−i+1个元素。若在顺序表的任何位置上插入数据元素的概率相等，即为1/(n+1)，则数据元素移动的平均次数为：

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

顺序表插入、删除算法的复杂度分析

在顺序表中删除第i个数据元素时，需要将删除元素之后的n-i个数据元素依次向前移动。若在顺序表的任何位置上删除数据元素的概率相等(为1/n)，则数据移动的次数平均为：

$$E_{dl} = \sum_{i=1}^{n} \frac{1}{n}(n-i) = \frac{n-1}{2}$$

因此，顺序表中插入和删除一个数据元素的时间复杂度为O(n)。

3.3 线性表的链表表示

在链表存储方式中，用结点存储线性表数据元素。结点通常有一个数据域，另外还有一个或一个以上的指针域。元素之间的关系通过指针来表示。

单链表

1. 单链表的结构

采用链接存储方式存储的线性表称为线性链表，又称单链表(linked list)，或简称为链表。在单链表中，每一个数据元素占用一个结点。如图3-5所示。一个结点由两个域组成，一个域存放数据元素data，一个域存放指向该链表中下一个结点的指针next，它给出下一个结点的开始存储地址。

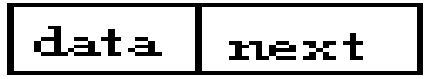


图3-5 单链表的结点结构

单链表

在单链表的表尾结点中，指针域为空以“∧”表示之。

设线性表存有某系99级学生的学号如下：

(99101，99104，99110，99201，99208)

可用如图3-6所示的单链表表示。

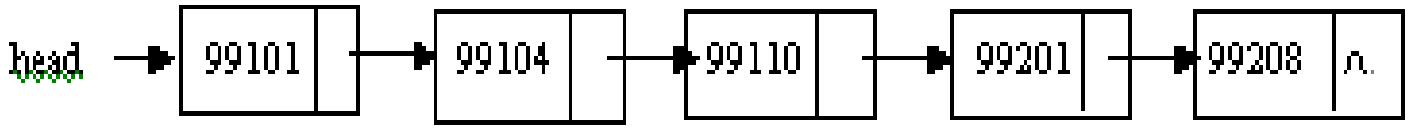
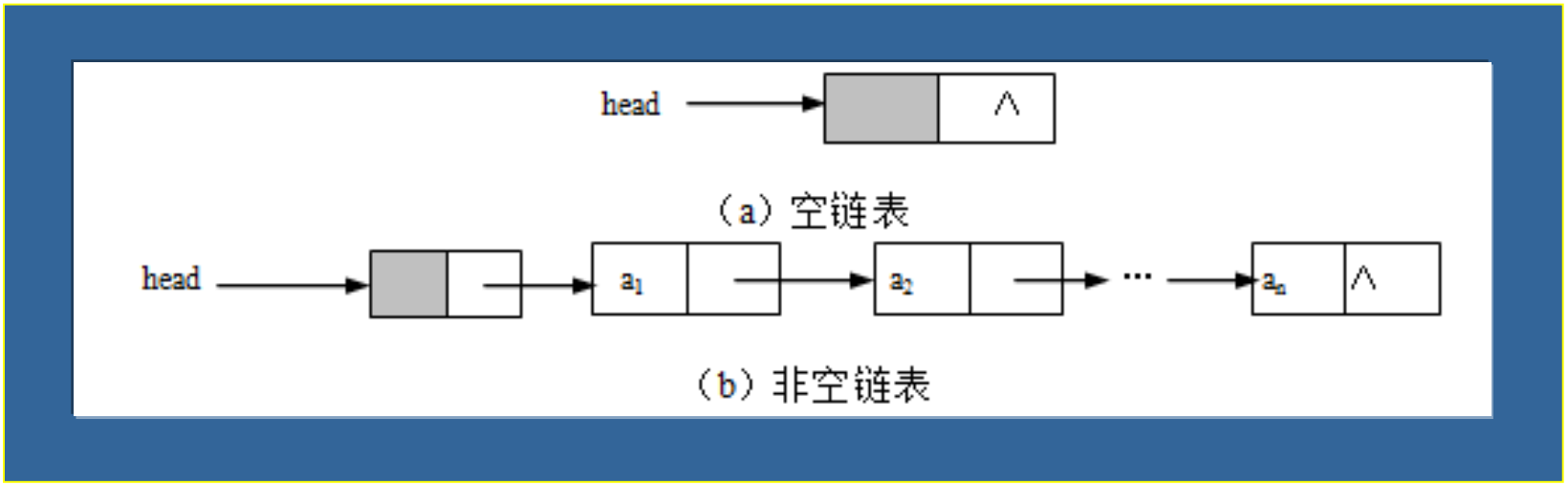


图 3-6　99 级学生学号的单链表

带头结点的单链表



单链表的插入

（1） 若i =1即a_i是链表中第一个结点中的数据，则新结点newnode应插入在第一个结点之前。

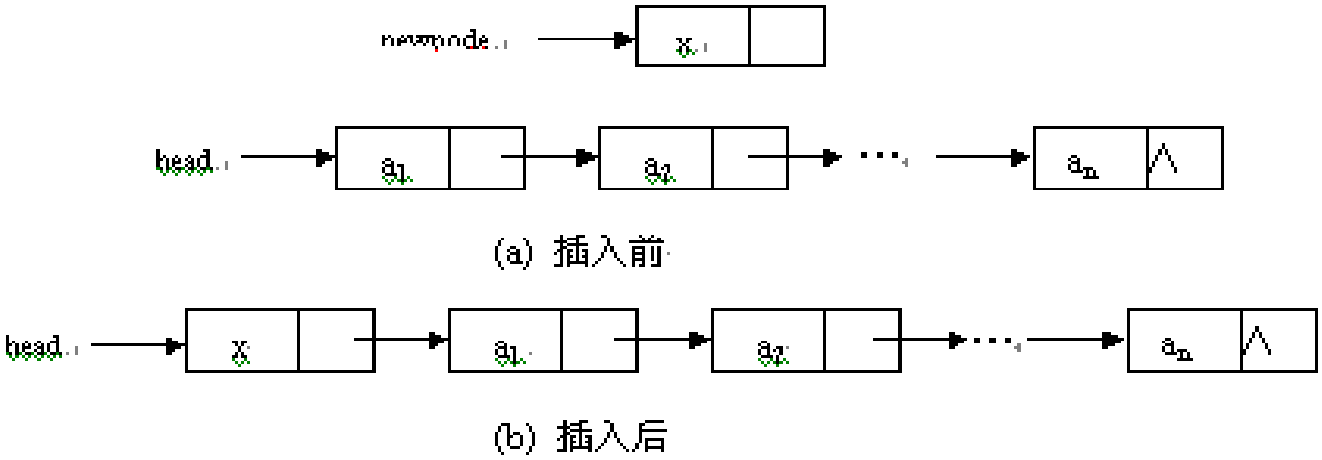


图 3-8 不带头结点单链表在第一个结点前插入结点的过程

单链表的插入

（2）若1<i≤n，即a_i不是链表中第一个结点中的数据，则新结点newnode插入在a_{i-1}与a_i之间。

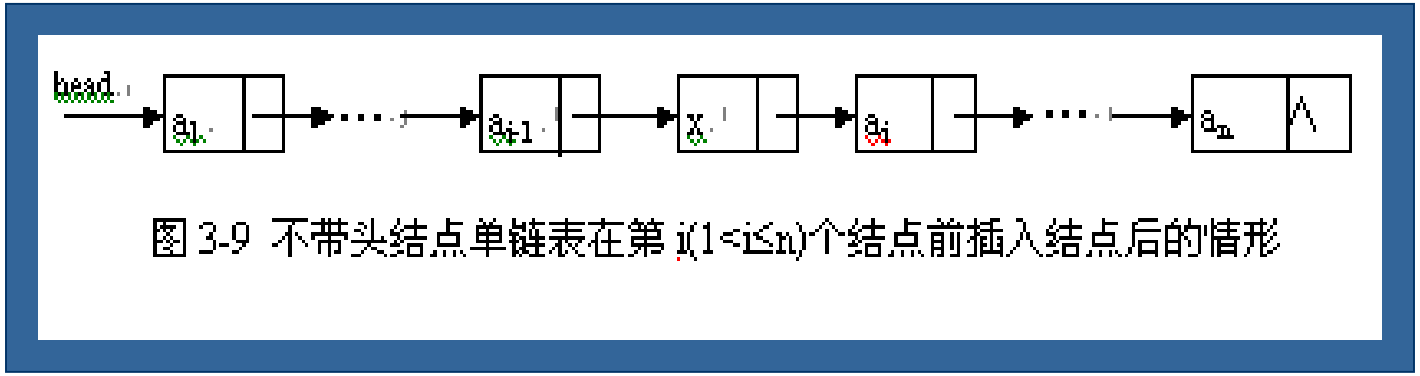


图 3-9 不带头结点单链表在第 i(1<i≤n)个结点前插入结点后的情形

单链表的插入

(3) 若i = n+1，即在线性表的表尾后插入新结点newnode，相当于表尾追加newnode。

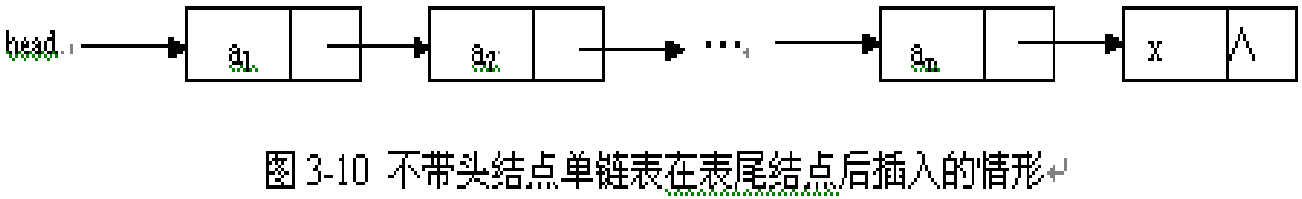


图 3-10 不带头结点单链表在表尾结点后插入的情形

单链表的删除

不带头结点的单链表的删除：删除链表中第i个结点。

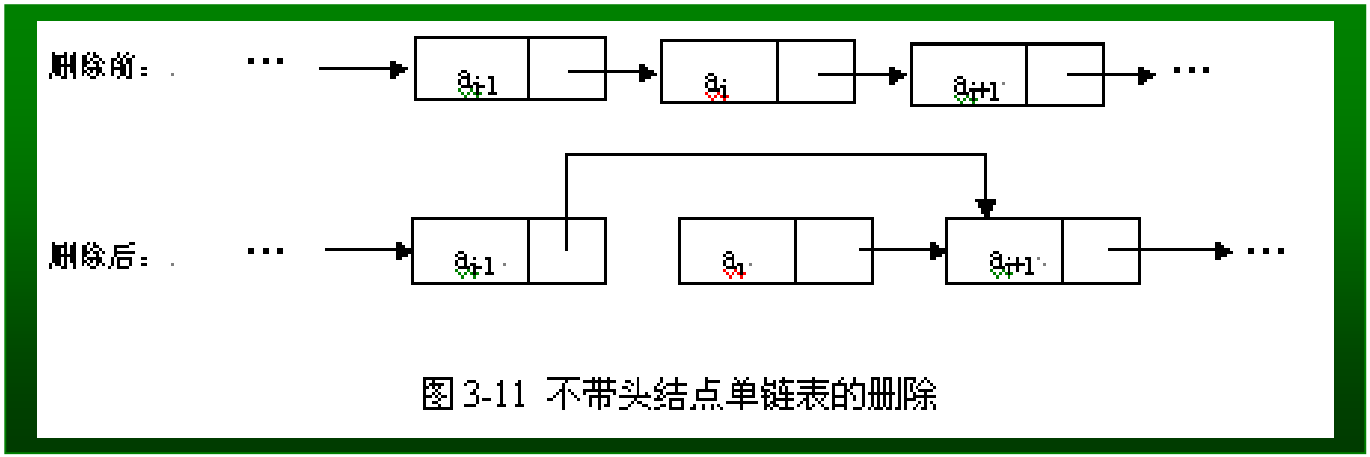


图 3-11 不带头结点单链表的删除

单链表中结点的类模板

```
template <class ElemType>
struct Node
{
        ElemType data;
        Node<ElemType> *next;

        Node();
        Node(ElemType e, Node<ElemType> *link=NULL);
};
```

单链表中结点的类模板

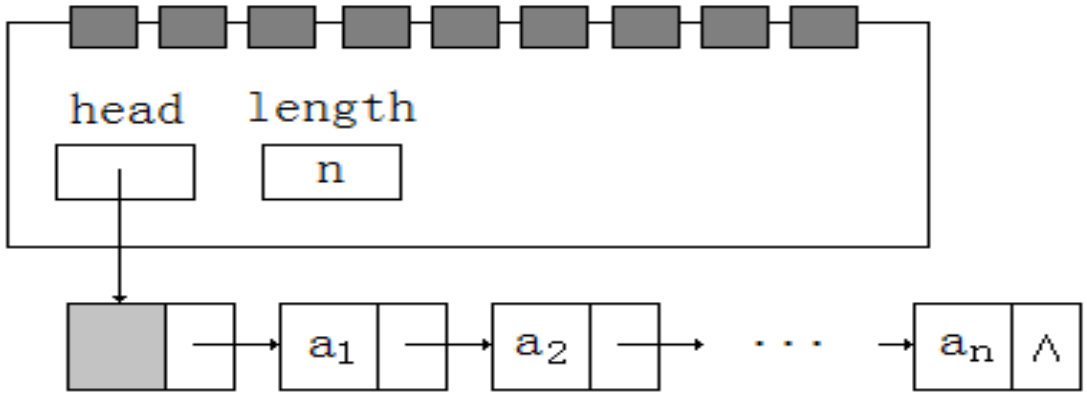
```
template<class ElemType>
Node<ElemType>::Node()
{
        next=NULL;
}
```

单链表中结点的类模板

```
template<class ElemType>
Node<ElemType>::Node(ElemType e, Node<ElemType> *link)
{
        data=e;
        next=link;
}
```

单链表的类模板

```
template <class ElemType>
class LinkList
{
protected:
        Node<ElemType> *head;
        int length;
```



单链表的类模板

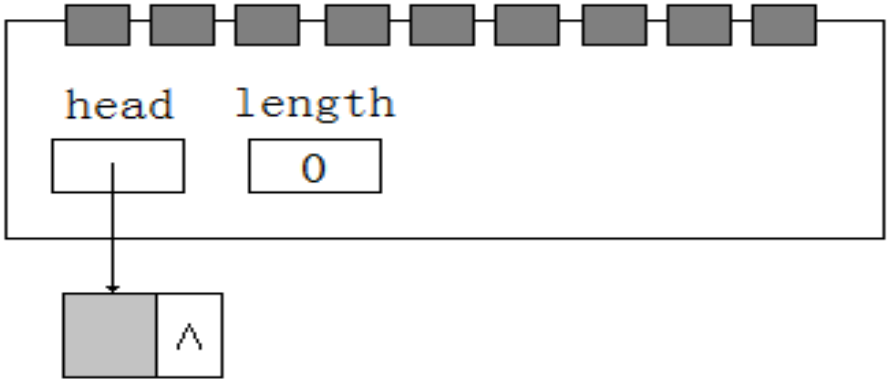
```
public:
        LinkList();
        LinkList(ElemType v[], int n);
        virtual ~LinkList();
        int GetLength() const;
        bool IsEmpty() const;
        void Clear();
        void Traverse(void (*Visit)(const ElemType &)) const;
        int LocateElem(const ElemType &e);
```

单链表的类模板

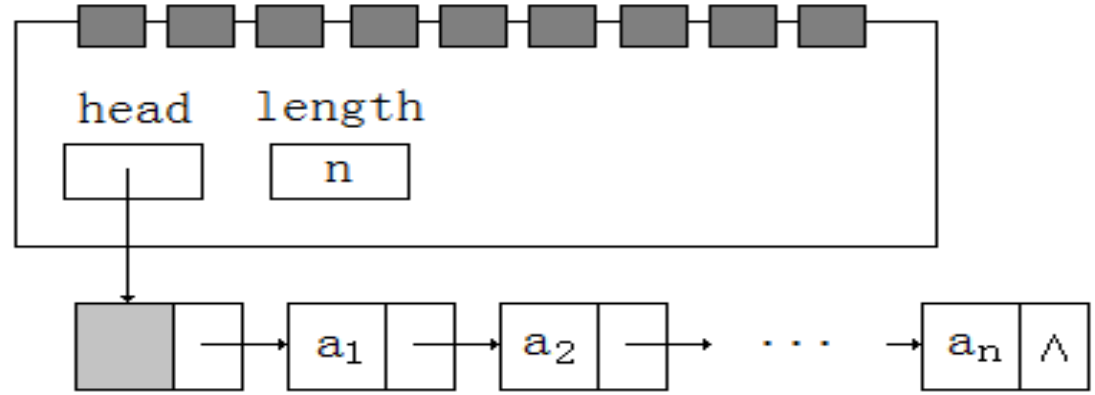
```
    Status GetElem(int i, ElemType &e) const;
    Status SetElem(int i, const ElemType &e);
    Status DeleteElem(int i, ElemType &e);
    Status InsertElem(int i, const ElemType &e);
    Status InsertElem(const ElemType &e);
    LinkList(const LinkList<ElemType> &la);
    LinkList<ElemType> &operator =(const LinkList
            <ElemType>  &la);
};
```

（1）无参数的构造函数

```
template <class ElemType>
linkList<ElemType>::linkList()
{
        head=new Node<ElemType>;
        assert(head);
        length=0;
}
```



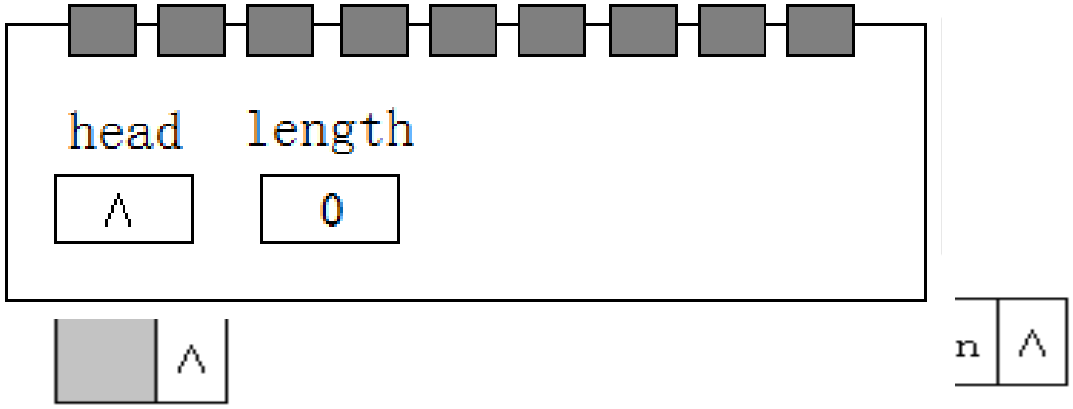
（2）根据数组内容构造链表




```
template <class Ele
linkList<ElemType>
    Node<ElemTyp
    p=head=new N
    assert(head);
    for (int i=0; i < n; i++) {
        p->next=new Node<ElemType>(v[i], NULL);
        assert(p->next);
        p=p->next;
    }
    length=n;
}
```

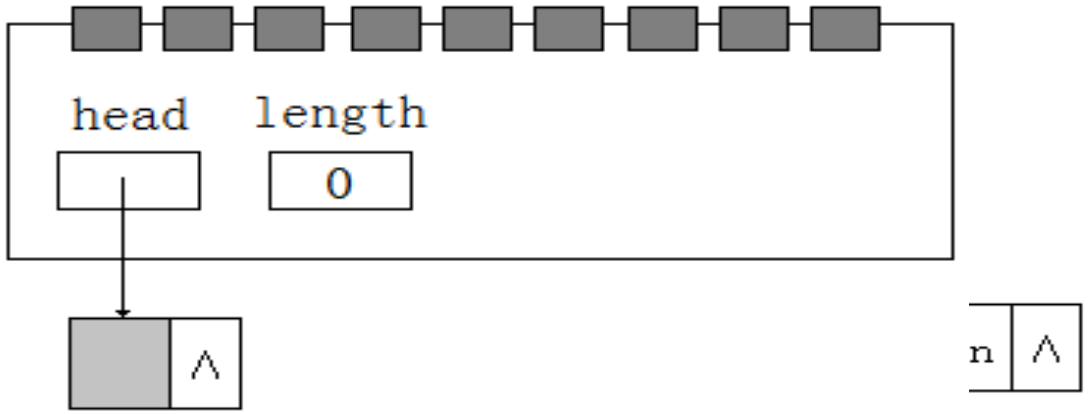
（3）析构函数

```
template <class ElemType>
linkList<ElemType>::~linkList()
{
        Clear();
        delete head;
}
```



（4）清空单链表

```
template <class ElemType>
void linkList<ElemType>::Clear()
{
    Node<ElemType> *p=head->next;
    while (p != NULL) {
        head->next=p->next;
        delete p;
        p=head->next;
     }
    length=0;
}
```

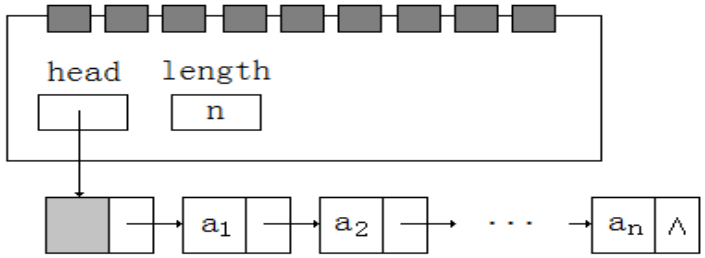


（5）遍历链表

```
template <class ElemType>
void LinkList<ElemType>::Traverse(void (*Visit)(const ElemType &)) const
{
    Node<ElemType> *p=head->next;
        while (p != NULL) {
                (*Visit)(p->data);
                p=p->next;
        }
}
```

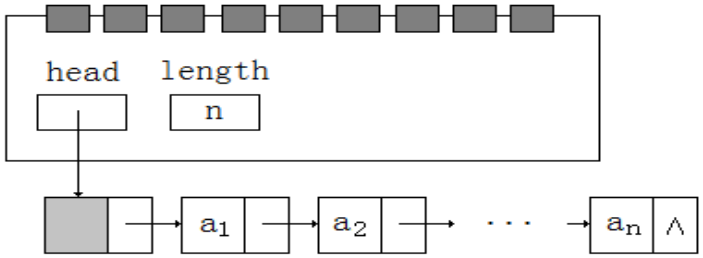
（6）元素定位

```
template <class ElemType>
int LinkList<ElemType>::LocateElem(const ElemType &e) const
{
        Node<ElemType> *p=head->next;
        int count=1;
        while (p != NULL && p->data != e) {
                count++;
                p=p->next;
        }
        return (p != NULL) ? count : 0;
}
```



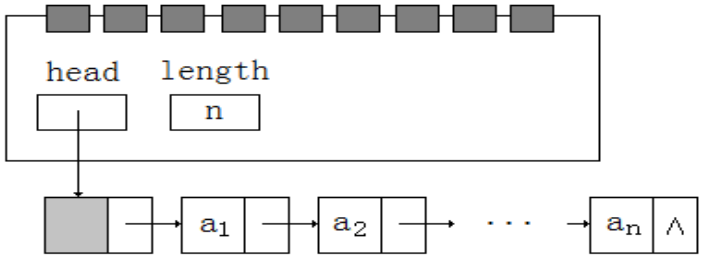
（7）取指定元素的值

```
template <class ElemType>
Status LinkList<ElemType>::GetElem(int i, ElemType &e) const {
    if (i < 1 || i > length)   return RANGE_ERROR;
    else  {
        Node<ElemType> *p=head->next;
        int count;
        for (count=1; count < i; count++)
            p=p->next;
        e=p->data;
        return ENTRY_FOUND;
    }
}
```



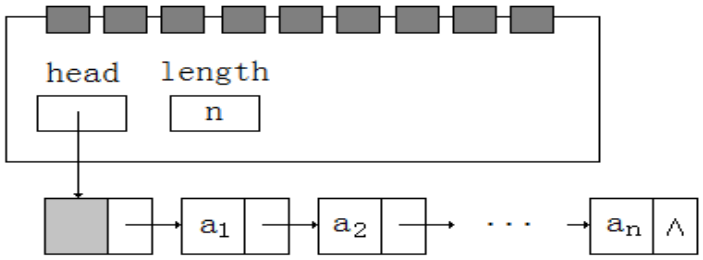
（8）修改指定元素的值

```
template <class ElemType>
Status LinkList<ElemType>::SetElem(int i, const ElemType &e) {
        if (i < 1 || i > length)    return RANGE_ERROR;
        else    {
                Node<ElemType> *p=head->next;
                int count;
                for (count=1; count < i; count++)
                        p=p->next;
                p->data=e;
                return SUCCESS;
        }
}
```



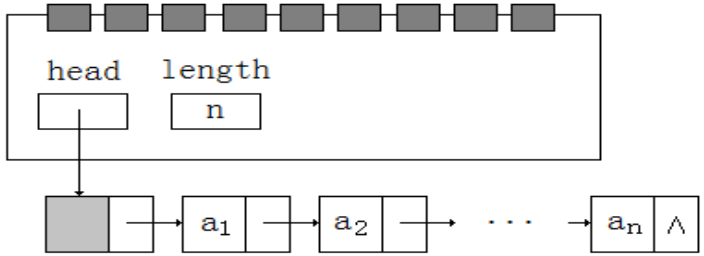
（9）删除指定元素

```
template <class ElemType>
Status LinkList<ElemType>::DeleteElem(int i, ElemType &e) {
    if (i < 1 || i > length)   return RANGE_ERROR;
    else   {
        Node<ElemType> *p=head, *q;
        int count;
        for (count=1; count < i; count++)  p=p->next;
        q=p->next;    p->next=q->next;    e=q->data;
        length--;       delete q;       return SUCCESS;
    }
}
```



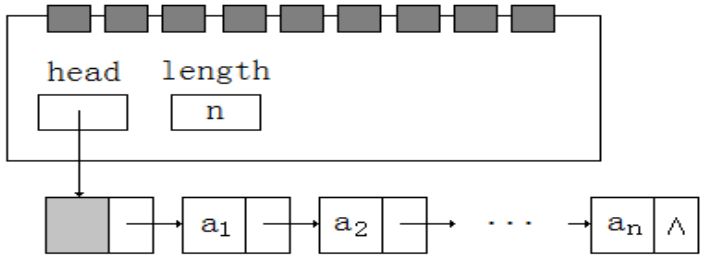
（10）在任意位置插入元素

```
template <class ElemType>
Status LinkList<ElemType>::InsertElem(int i, const ElemType &e){
    if (i < 1 || i > length+1)   return RANGE_ERROR;
    else {
        Node<ElemType> *p=head, *q;
        int count;
        for (count=1; count < i; count++)  p=p->next;
        q=new Node<ElemType>(e, p->next);
        assert(q);
        p->next=q;    length++; return SUCCESS;
    }
}
```

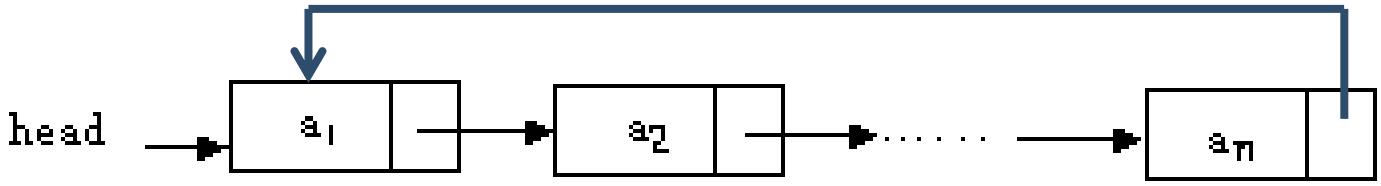


（11）在表尾插入元素

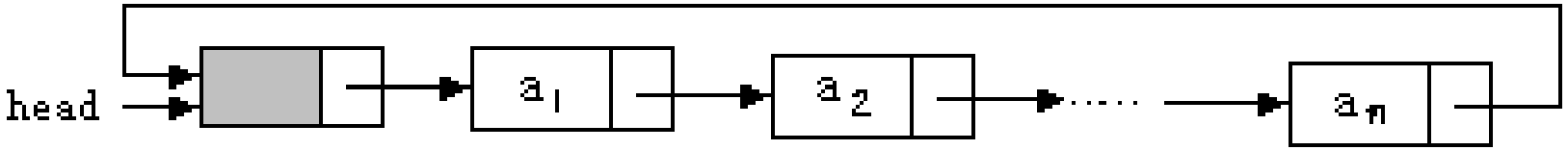
```
template <class ElemType>
Status LinkList<ElemType>::InsertElem(const ElemType &e)
{
        Node<ElemType> *p, *q;
        q=new Node<ElemType>(e, NULL);
        assert(q);
        for (p=head; p->next != NULL; p=p->next) ;
        p->next=q;
        length++;
        return SUCCESS;
}
```

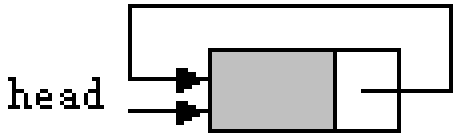


不带头结点的单循环链表



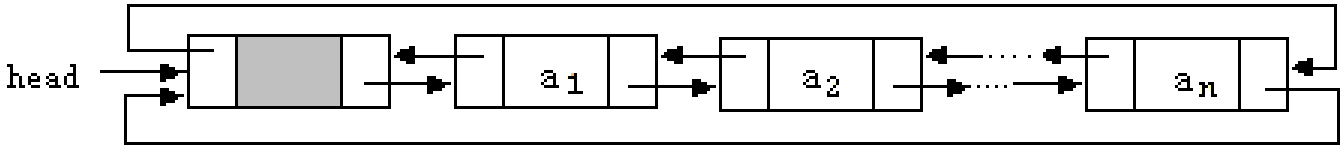
带头结点的单循环链表



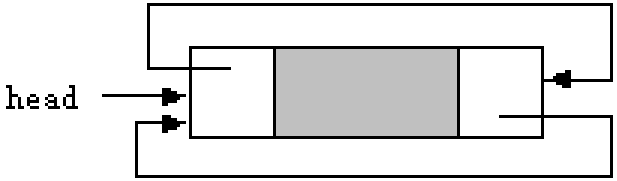


双向循环链表

双向链表结点



(a) 非空表



(b) 空表

图3- 21 带头结点的双向链表

双向循环链表中结点的类模板

```
template <class ElemType>
struct DblNode
{
        ElemType data;
        DblNode<ElemType> *prior ;
        DblNode<ElemType> *next ;

        DblNode();
        DblNode(ElemType e,
                DblNode<ElemType> *priorlink=NULL,
                DblNode<ElemType> *nextlink=NULL);
};
```

双向循环链表中结点的类模板

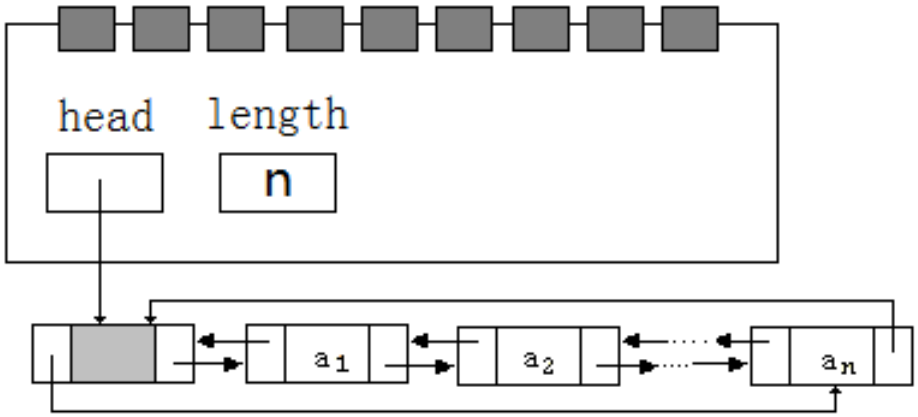
```
template<class ElemType>
DblNode<ElemType>::DblNode()
{
        prior=NULL;
        next=NULL;
}
```

双向循环链表中结点的类模板

```
template<class ElemType>
DblNode<ElemType>::DblNode(ElemType e,
                           DblNode<ElemType> *priorlink,
                           DblNode<ElemType> *nextlink)
{
        data=e;
        prior=priorlink;
        next=nextlink;
}
```

双向循环链表的类模板

```
template <class ElemType>
class DblLinkList
{
protected:
        DblNode<ElemType> *head;
        int length;
```



双向循环链表的类模板

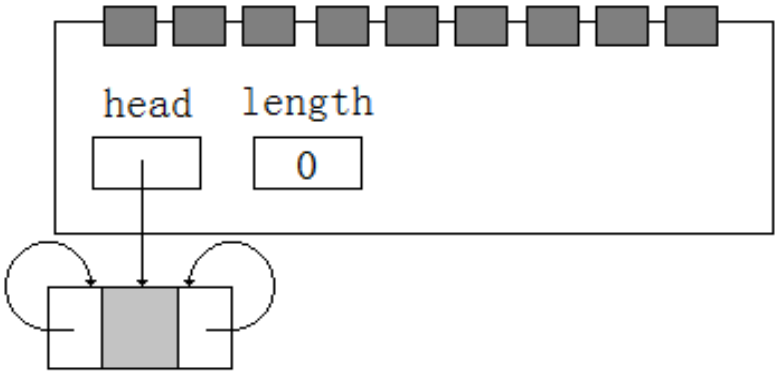
```
public:
        DblLinkList();
        DblLinkList(ElemType v[], int n);
        virtual ~DblLinkList();
        int GetLength() const;
        bool IsEmpty() const;
        void Clear();
        void Traverse(void (*Visit)(const ElemType &)) const;
        int LocateElem(const ElemType &e);
```

双向循环链表的类模板

```
    Status GetElem(int i, ElemType &e) const;
    Status GetElem(int i, ElemType &e) const;
    Status SetElem(int i, const ElemType &e);
    Status DeleteElem(int i, ElemType &e);
    Status InsertElem(int i, const ElemType &e);
    Status InsertElem(const ElemType &e);
    DblLinkList(const DblLinkList<ElemType> &la);
    DblLinkList<ElemType> &operator =(const
            DblLinkList<ElemType> &la);
};
```

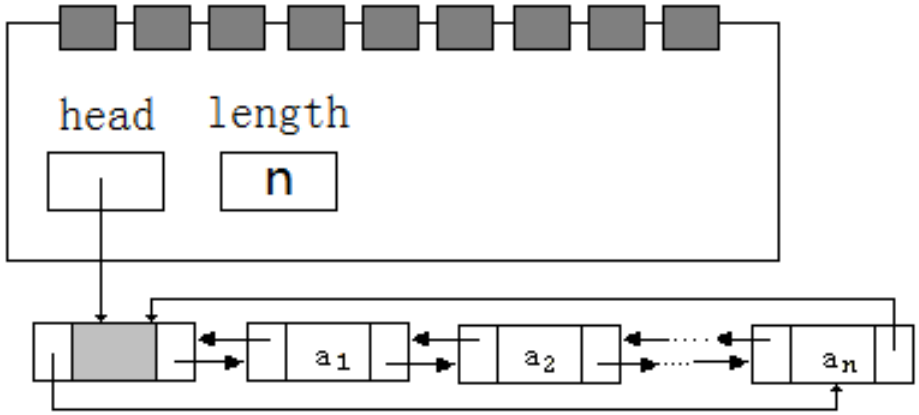
（1）无参数的构造函数

```
template <class ElemType>
DblLinkList<ElemType>::DblLinkList()
{
        head=new DblNode<ElemType>;
        assert(head);
        head->prior=head->next=head;
        length=0;
}
```



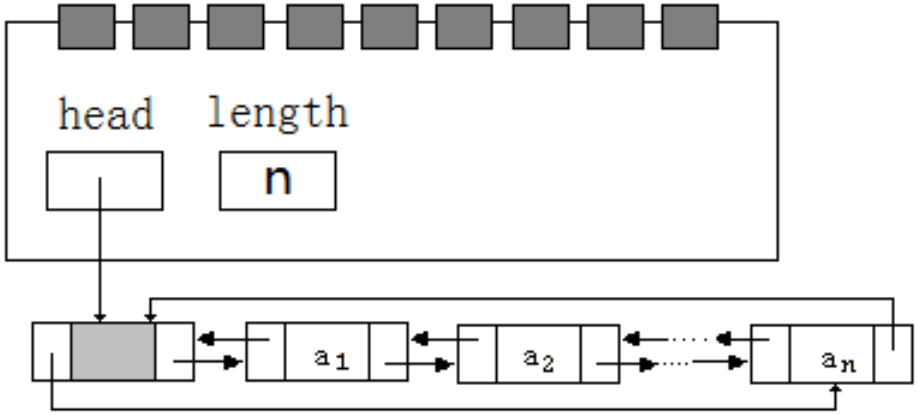
（2）根据数组内容构造链表

```
template <class ElemType>
DblLinkList<ElemType>::DblLi
{
        DblNode<ElemType> *
        p=head=new DblNode
        assert(head);
        for (int i=0; i < n; i++) {
                p->next=new DblNode<ElemType>(v[i], p);
                p=p->next;
        }
        length=n;      head->prior=p;      p->next=head;
}
```



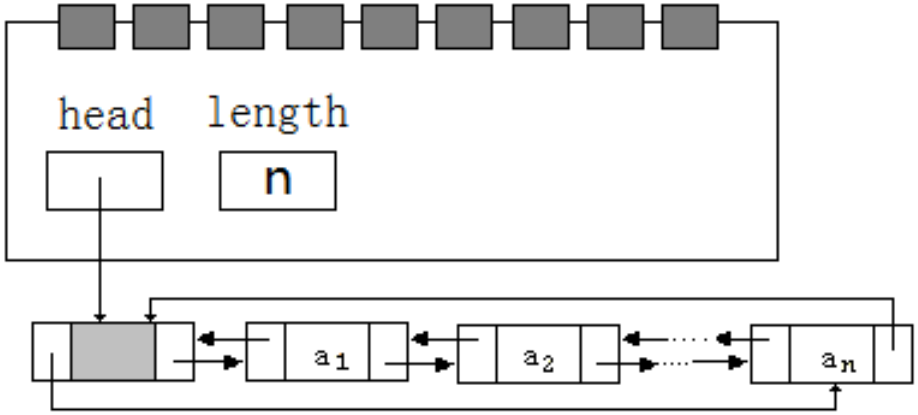
（3）析构函数

```
template <class ElemType>
DblLinkList<ElemType>::~DblLinkList()
{
        Clear();
        delete head;
}
```



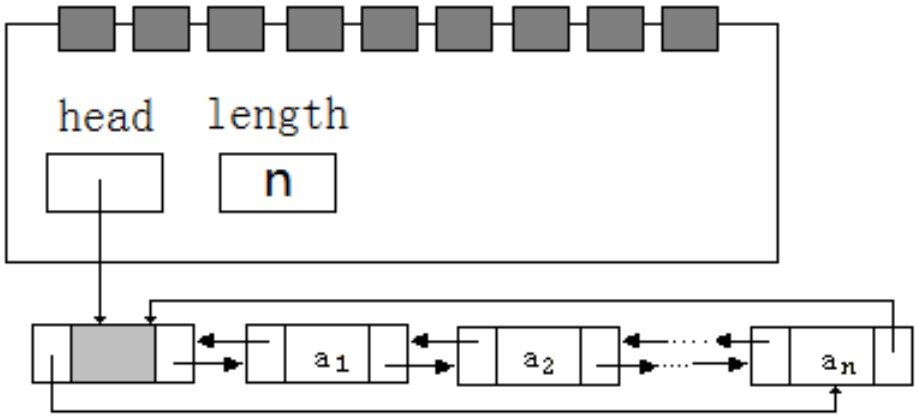
（4）清空单链表

```
template <class ElemType>
void DblLinkList<ElemType>::Clear()
{
        ElemType tmpElem;
        while (length > 0)
                DeleteElem(1, tmpElem);
}
```



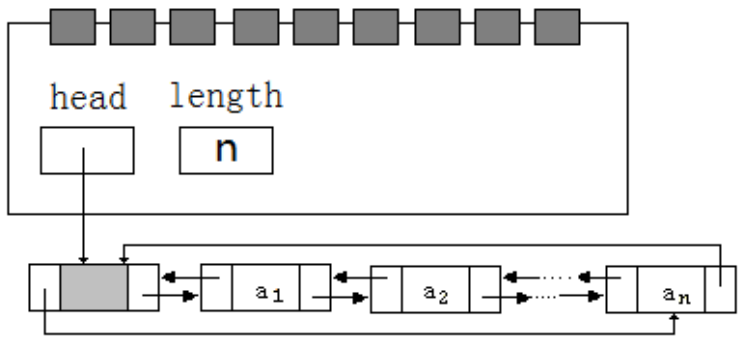
（5）遍历链表

```
template <class ElemType>
void DblLinkList<ElemType>::Traverse(void
(*Visit)(const ElemType &)) const
{
        DblNode<ElemType> *p;
        for (p=head->next; p != head; p=p->next)
                (*Visit)(p->data);
}
```



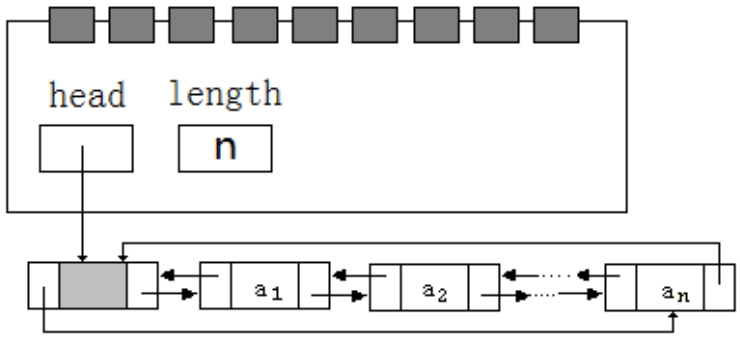
（6）元素定位

```
template <class ElemType>
int DblLinkList<ElemType>::LocateElem(const ElemType &e)  {
        DblNode<ElemType> *p=head->next;
        int count=1;
        while (p != head && p->data != e) {
                count++;        p=p->next;
        }
        if (p != head)    return count;
        else                  return 0;
}
```



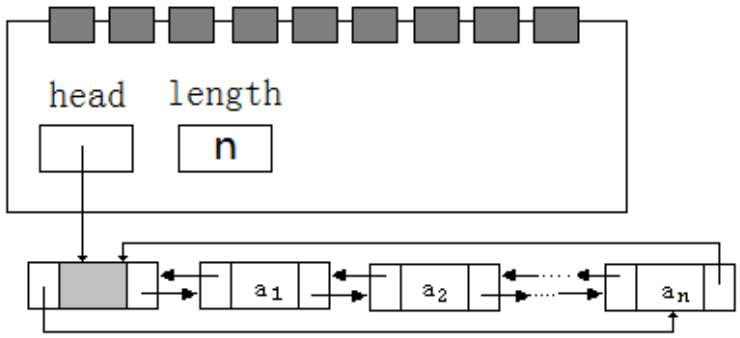
（7）取指定元素的值

```
template <class ElemType>
Status DblLinkList<ElemType>::GetElem(int i, ElemType &e) const
{       DblNode<ElemType> *p=head->next;
        int count;
        if (i < 1 || i > length)   return NOT_PRESENT;
        else    {
                for (count=1; count < i; count++)  p=p->next;
                e=p->data;    return ENTRY_FOUND;
        }
}
```



（8）修改指定元素的值

```
template <class ElemType>
Status DblLinkList<ElemType>::SetElem(int i, const ElemType &e)
{       DblNode<ElemType> *p=head->next;
        int count;
        if (i < 1 || i > length)  return RANGE_ERROR;
        else    {
                for (count=1; count < i; count++)  p=p->next;
                p->data=e;    return SUCCESS;
        }
}
```



（9）删除指定元素

templa

Status , ElemType &e)

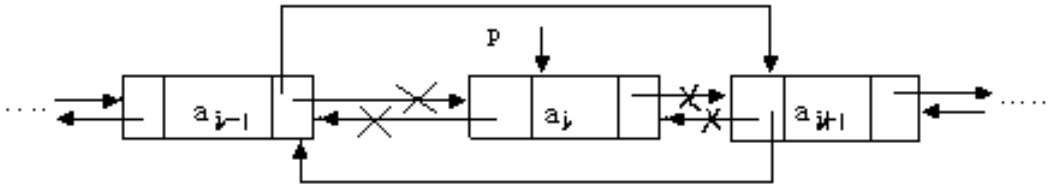


图3-23 双向链表的删除

{

```
    int count;
    if (i < 1 || i > length)   return RANGE_ERROR;
    else    {
        for (count=1; count < i; count++)   p=p->next;
        p->prior->next=p->next;  p->next->prior=p->prior;
        e=p->data;  length--;
        delete p;   return SUCCESS;
    }
```

}

（10）在任意位置插入元素

```
template <
Status DblL                                                    t ElemType &e)
{       DblN
        int c
        if (i                                                  ROR;
        else    {
            for (count=1; count < i; count++) p=p->next;
            q=new DblNode<ElemType>(e, p->prior, p);
            p->prior->next=q;   p->prior=q;
            length++;  return SUCCESS;
        }
}
```

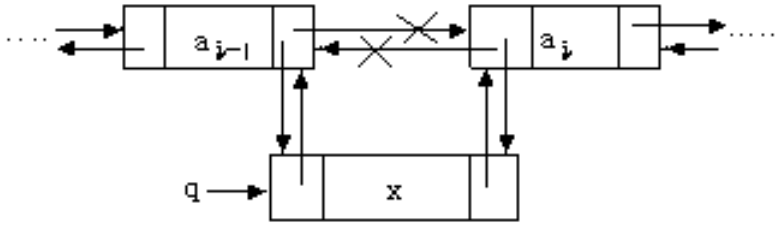
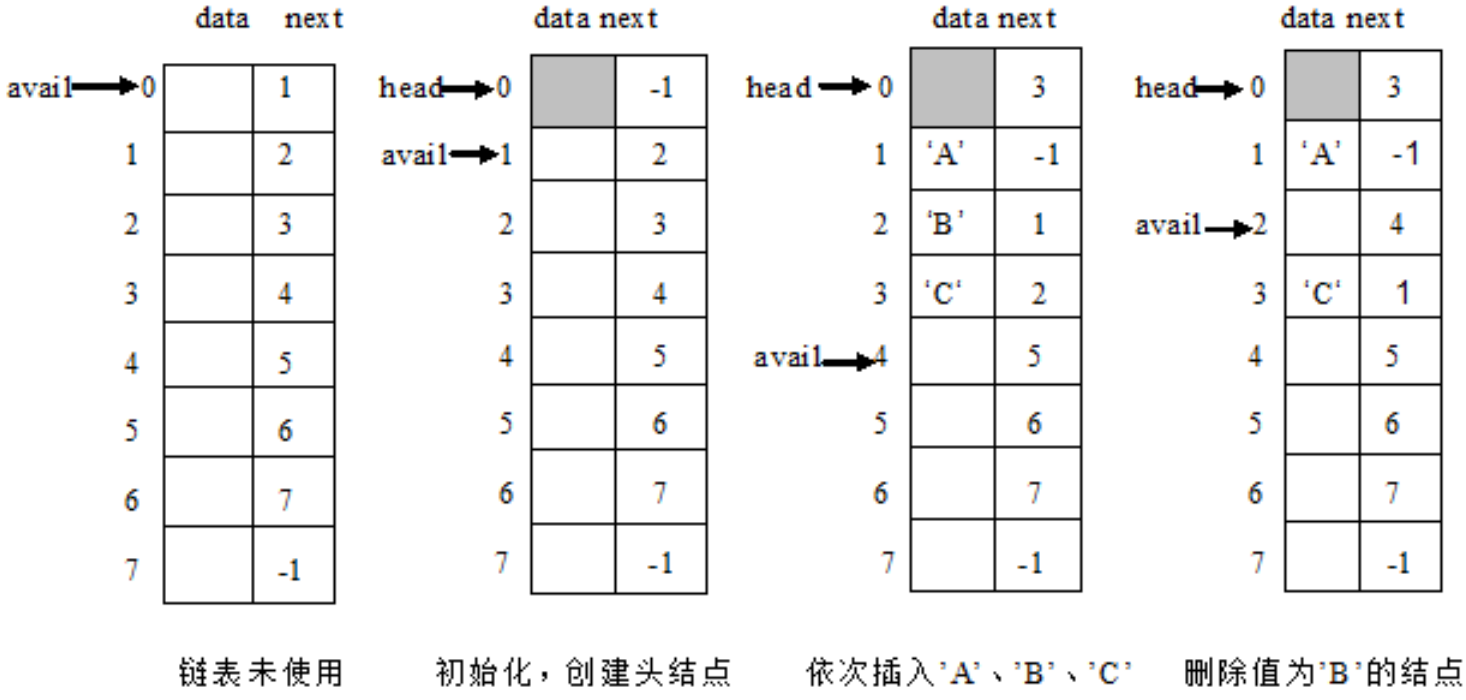


图3- 24 双向链表前插入算法过程

（11）在表尾插入元素

```
template <class ElemType>
Status DblLinkList<ElemType>::InsertElem(const ElemType &e)
{
        DblNode<ElemType> *p;
        p=new DblNode<ElemType>(e, head->prior, head);
        head->prior->next=p;
        head->prior=p;
        length++;
        return SUCCESS;
}
```

静态链表


data next
avail 0 1
1 2
2 3
3 4
4 5
5 6
6 7
7 -1
链表未使用
data next
head 0 -1
avail 1 2
2 3
3 4
4 5
5 6
6 7
7 -1
初始化，创建头结点
data next
head 0 3
1 'A' -1
2 'B' 1
3 'C' 2
avail 4 5
5 6
6 7
7 -1
依次插入'A'、'B'、'C'
data next
head 0 3
1 'A' -1
avail 2 4
3 'C' 1
4 5
5 6
6 7
7 -1
删除值为'B'的结点

3.4 线性表的应用

例3-1利用顺序表表示集合，并求两个集合的交。

数据结构：可以考虑利用两个顺序表la和lb分别表示两个集合。

算法思想：为了简化算法实现，先定义两个数组a和b存放两个集合的元素，再利用这两个数组初始化构造两个顺序表la、lb，定义一个空的顺序表lc以存放两个集合交的结果。通过一个循环，依次取出la中的每一个元素e，再用定位函数确定元素e是否在lb中。如果元素e在lb中（定位函数返回序号大于0），则把它插入lc中。算法最后再分别遍历三个顺序表，输出三个集合的内容。

求两个集合的交集

```
int main(void)  {
    char a[]={'A', 'C', 'E', 'G', 'I'}, b[]={'A', 'B', 'C', 'D', 'H', 'I', 'J'}, e;
    SeqList<char> la(a, 5, 50), lb(b, 7, 50), lc(50);
    for (int i=1; i <= la.GetLength(); i++)   {
        la.GetElem(i, e);
        if (lb.LocateElem(e))     lc.InsertElem(e);
     }
     cout << "集合A:";        la.Traverse(Write);    cout << endl;
     cout << "集合B:";        lb.Traverse(Write);    cout << endl;
     cout << "集合C:";        lc.Traverse(Write);    cout << endl;
     system("PAUSE");       return 0;
}
```

一元多项式表示和相关运算的实现

例3-2 一元多项式表示

一般情况下的多项式可写成：

$$A(x) = a_m x^{e_m} + \cdots + a_2 x^{e_2} + a_1 x^{e_1}$$

其中a_i是非零系数，指数e_i是非负整数，且$e_m > e_{m-1} > \ldots > e_2 > e_1 \geqslant 0$。若用一个长度为m且每个元素有两个数据项的线性表

$$((a_m, e_m), \ldots, (a_2, e_2), (a_1, e_1))$$

便可唯一确定多项式A(x)。对此线性表可以有两种存储结构，其一是顺序存储结构；其二是链表存储结构。

项的定义

```
struct PolyItem
{
// 数据成员:
   double  coef;                          // 系数
   int     expn;                          // 指数
// 构造函数:
    PolyItem() ;                          // 无参构造函数
    PolyItem(double cf, int en);          // 有参构造函数
};
```

用单链表表示一元多项式的类模板定义

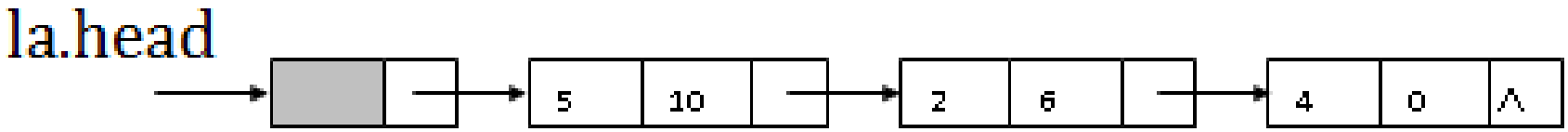
```
class Polynomial
{
protected:
    LinkList<PolyItem> polyList;
public:
    Polynomial(){};
    ~Polynomial(){};
    int Length() const;
    bool IsZero() const;
    void SetZero();
```

用单链表表示一元多项式的类模板定义

```
    void Display();
    void InsItem( const PolyItem &item);
    Polynomial operator +(const Polynomial &p) const;
    Polynomial operator -(const Polynomial &p) const;
    Polynomial operator *(const Polynomial &p) const;
    Polynomial(const Polynomial &copy);
    Polynomial(const LinkList<PolyItem> &copyLinkList);
    Polynomial &operator =(const Polynomial &copy);
    Polynomial &operator =(const LinkList<PolyItem>
                  &copyLinkList);
};
```

用单链表表示一元多项式的类模板定义

$$a = 5x^{10} + 2x^{6} + 4$$



多项式相加运算的实现

数据结构：当上面的多项式a和b相加时，设置两个链表la和lb分别存放两个多项式的二元组线性表，并设结果多项式链表为lc。

算法思想：开始时先从la、lb两个链表取第一个元素（即多项式a、b的第一项的系数和指数），进行步骤一。

步骤一：当两个链表la和lb都能取到元素时，比较检测结点的指数域；否则进行步骤二。

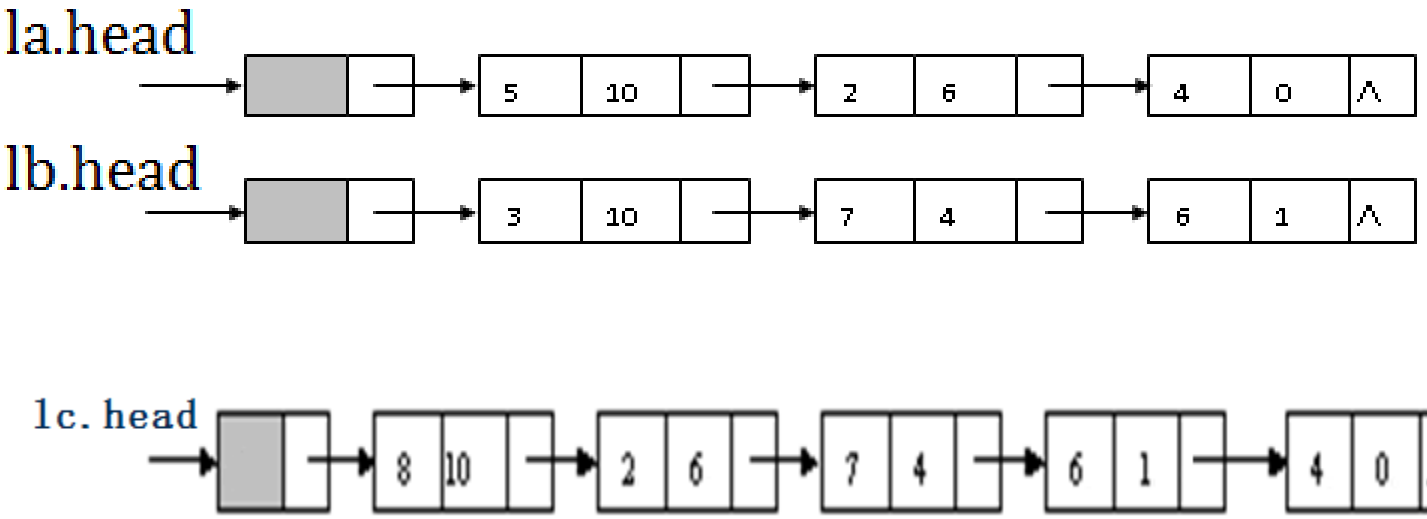
（1）指数不等：指数大者插入lc链表，且在指数大的链表中取下一个元素；

（2）指数相等：对应项系数相加。若相加结果不为零，则结果插入lc链表中；且在两个链表la和lb中分别取下一个元素。

步骤二：把la或lb链表中剩余部分加入到lc链表中。

多项式相加运算的实现

$$a = 5x^{10} + 2x^{6} + 4 \qquad b = 3x^{10} + 7x^{4} + 6x$$



多项式相加运算的实现

```
Polynomial Polynomial::operator +(const Polynomial &p) const {
        LinkList<PolyItem> la=polyList;
        LinkList<PolyItem> lb=p.polyList;
        LinkList<PolyItem> lc;
        int aPos=1, bPos=1;
        PolyItem aItem, bItem;
        Status aStatus, bStatus;
        aStatus=la.GetElem(aPos++, aItem);
        bStatus=lb.GetElem(bPos++, bItem);
```

多项式相加运算的实现

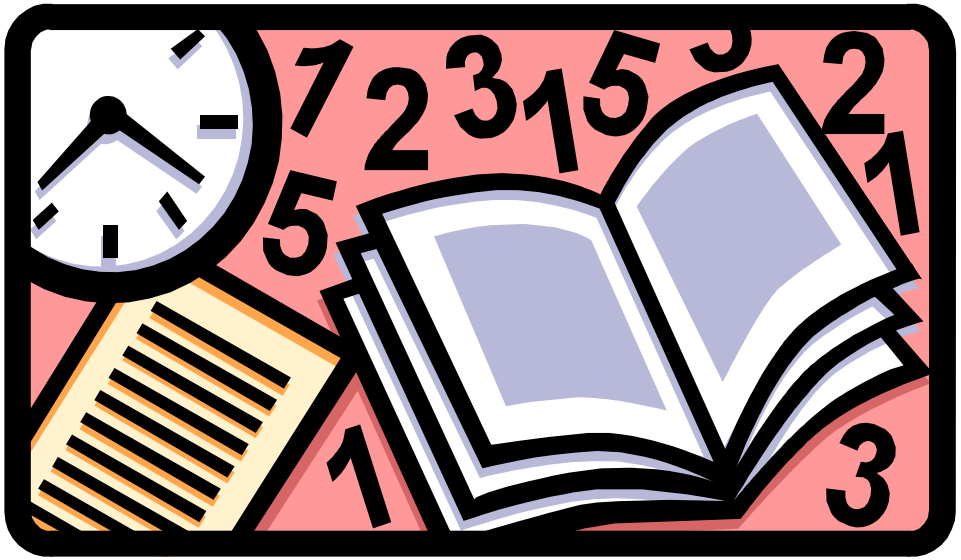
```
while (aStatus == ENTRY_FOUND  && bStatus == ENTRY_FOUND )          {
    if (aItem.expn > bItem.expn) {
            lc.InsertElem(aItem);       aStatus=la.GetElem(aPos++, aItem);
     }
     else if (aItem.expn < bItem.expn) {
            lc.InsertElem(bItem);       bStatus=lb.GetElem(bPos++, bItem);
      }
     else {
            PolyItem sumItem(aItem.coef + bItem.coef, aItem.expn);
            if (sumItem.coef != 0)      lc.InsertElem(sumItem);
            aStatus=la.GetElem(aPos++, aItem);
            bStatus=lb.GetElem(bPos++, bItem);
     }
}
```

多项式相加运算的实现

```
        while (aStatus == ENTRY_FOUND) {
                lc.InsertElem(aItem);
                aStatus=la.GetElem(aPos++, aItem);
        }
        while (bStatus == ENTRY_FOUND) {
                lc.InsertElem(bItem);
                bStatus=lb.GetElem(bPos++, bItem);
        }
        Polynomial fc;
        fc.polyList=lc;
        return fc;
}
```

多项式相加运算的实现

设两个多项式链表的长度分别为m和n，则总的比较指数的次数为O(m+n)。对于多项式的减法、乘法也可以类似地定义。



学习进步！