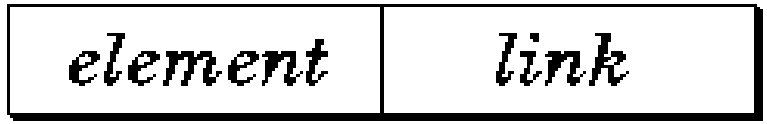
第三章　链表

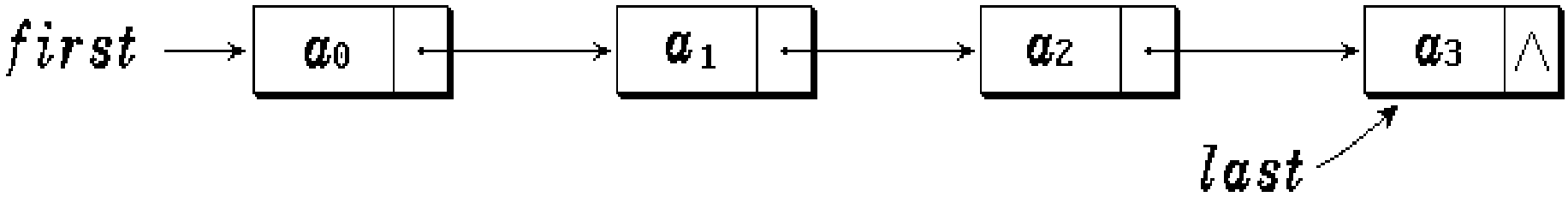
- 单链表 (*Singly Linked List*)
- 循环链表 (*Circular List*)
- 多项式及其相加
- 双向链表 (*Doubly Linked List*)
- 稀疏矩阵
- 小结

单链表 (*Singly Linked List*)

- 特点
  - 每个元素(表项)由结点(*Node*)构成。

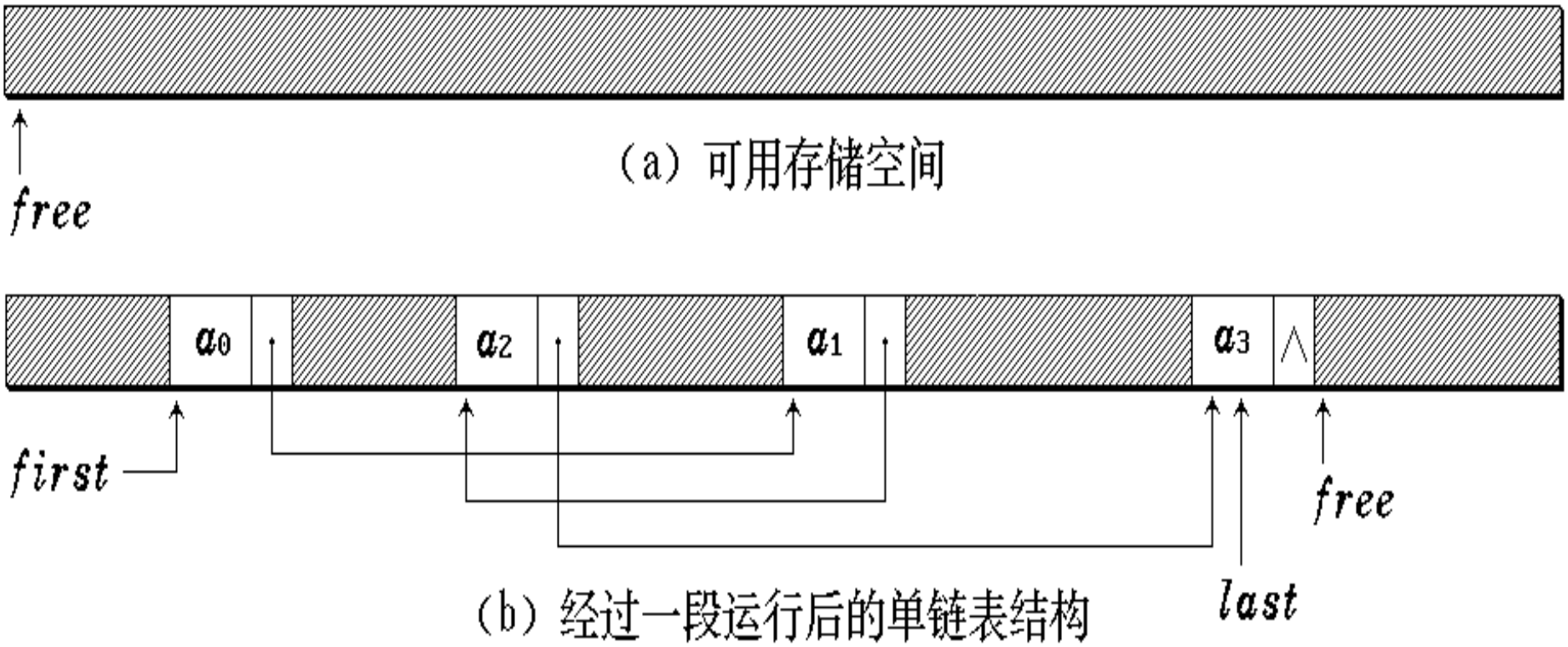


  - 线性结构



  - 结点可以不连续存储
  - 表可扩充

单链表的存储映像



单链表的类定义

- 多个类表达一个概念(单链表)。
  - 链表结点(*ListNode*)类
  - 链表(*List*)类
  - 链表游标(*Iterator*)类
- 定义方式
  - 复合方式
  - 嵌套方式

```
class List;                          //复合类定义
class ListNode {                     //链表结点类
friend class List;                   //链表类为其友元类
private:
    int data;                        //结点数据, 整型
    ListNode *link;                  //结点指针
};
class List {                         //链表类
public:
    //链表公共操作
    ……
private:
    ListNode *first, *last;          //表头和表尾指针
};
```

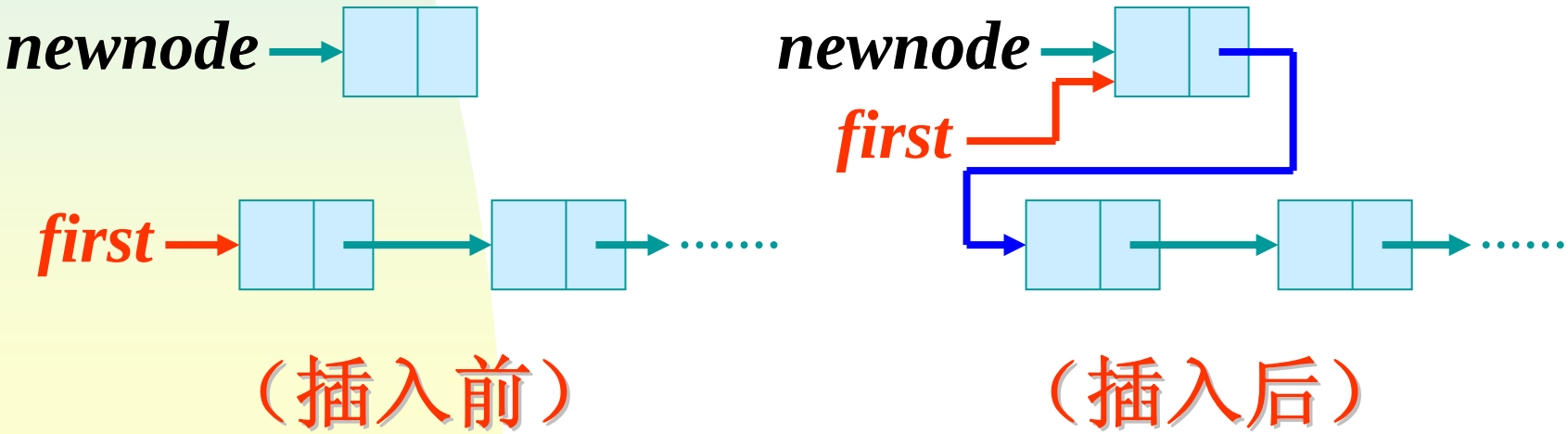
```
class List {          //链表类定义(嵌套方式)
public:
    //链表操作
    ……
private:
    class ListNode {              //嵌套链表结点类
    public:
        int data;
        ListNode *link;
    };
    ListNode *first, *last;       //表头和表尾指针
};
```

单链表中的插入与删除

- 插入
  - 第一种情况：在第一个结点前插入

newnode→link = first ;

first = newnode;

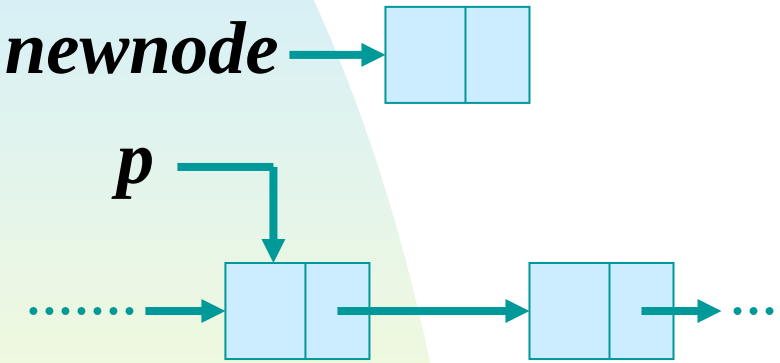


（插入前）　　（插入后）

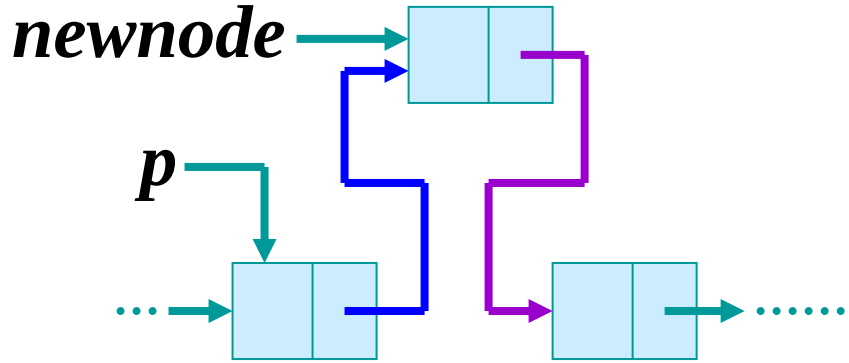
第二种情况：在链表中间插入

newnode→link = p→link;

p→link = newnode;



（插入前）

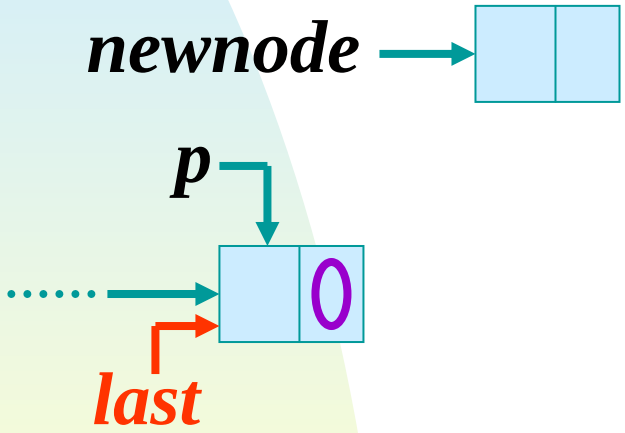


（插入后）

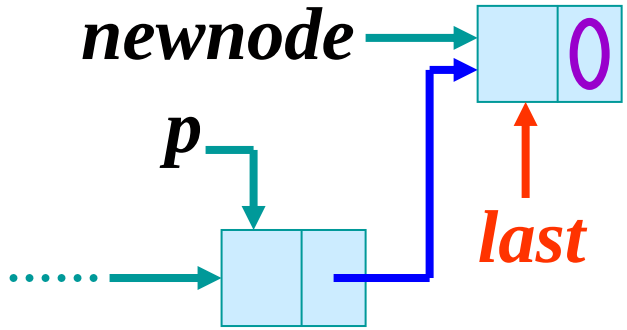
- 第三种情况：在链表末尾插入

newnode→link = p→link;

p→link = last = newnode;



（插入前）

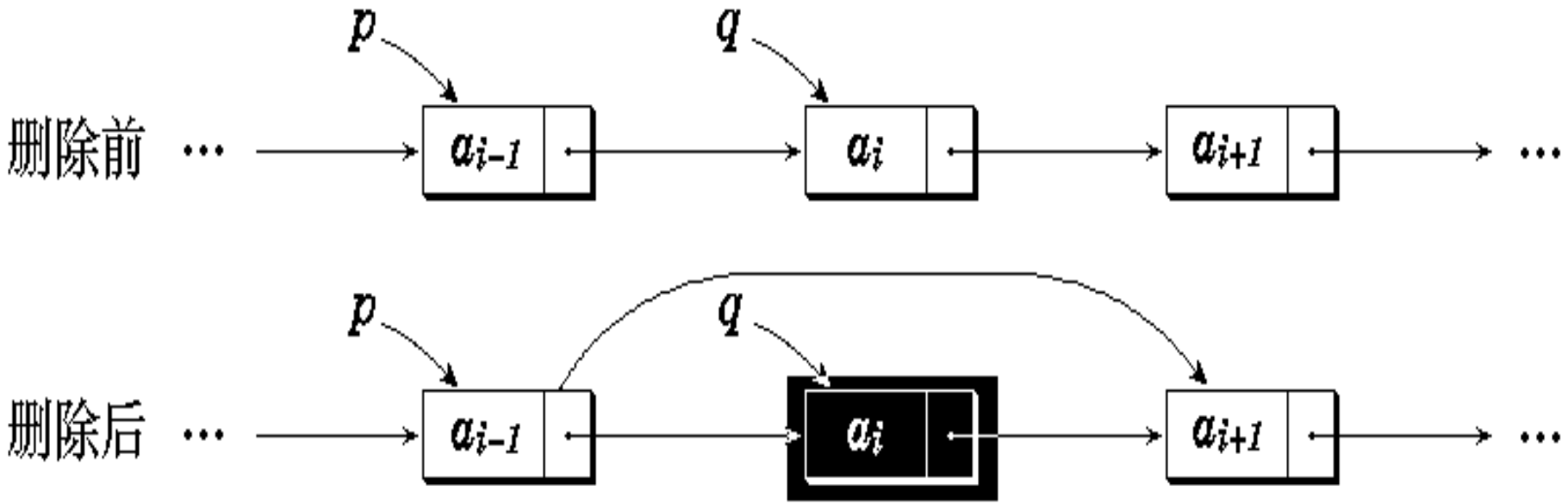


（插入后）

```
int List::Insert ( const int x, const int i ) {
//在链表第 i 个结点处插入新元素 x
   ListNode *p = first;  int k = 0;
   while ( p != NULL && k < i −1 )
       { p = p→link;  k++; }   //找第i−1个结点
   if ( p == NULL && first != NULL ) {
      cout << “无效的插入位置!\n”;
      return 0;
   }
   ListNode *newnode= new ListNode(x, NULL);
    //创建新结点,其数据为x,指针为0
```

```
    if ( first == NULL || i == 0 ) {    //插在表前
      newnode→link = first;
      if ( first == NULL ) last = newnode;
      first = newnode;
    }
    else {                              //插在表中或末尾
      newnode→link = p→link;
       if ( p→link == NULL ) last = newnode;
      p→link = newnode;
    }
    return 1;
}
```

- 删除
  - 第一种情况：删除表中第一个元素
  - 第二种情况：删除表中或表尾元素



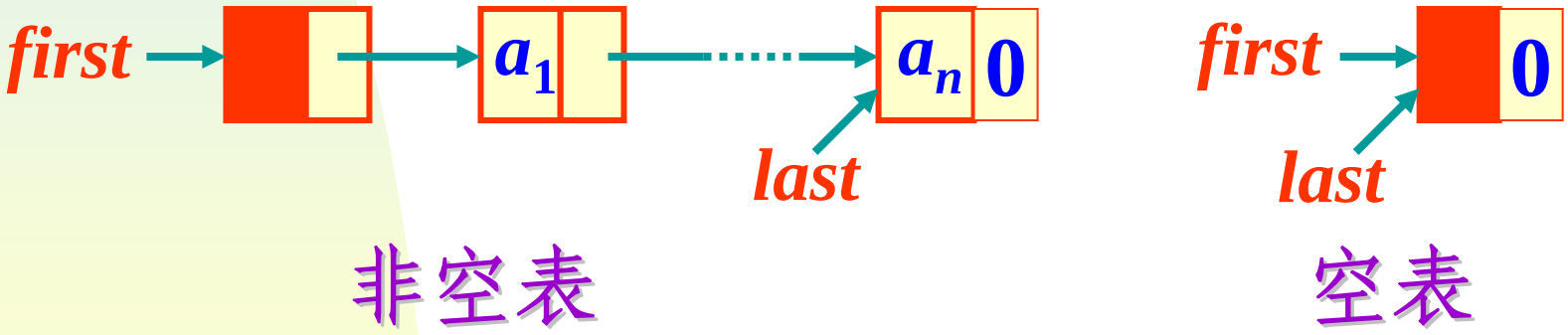
在单链表中删除含a_i的结点

```
int List::Remove ( int i ) {
//在链表中删除第i个结点
    Node *p = first, *q;  int k = 0;
    while ( p != NULL && k< i−1 )
      { p = p→link;  k++; }    //找第i−1个结点
    if ( p == NULL || p→link == NULL ) {
      cout << “无效的删除位置!\n”;
      return 0;
    }
    if ( i == 0 ) {                    //删除表中第 1 个结点
       q = first;                      //q 保存被删结点地址
       p = first = first→link;         //修改first
    }
```

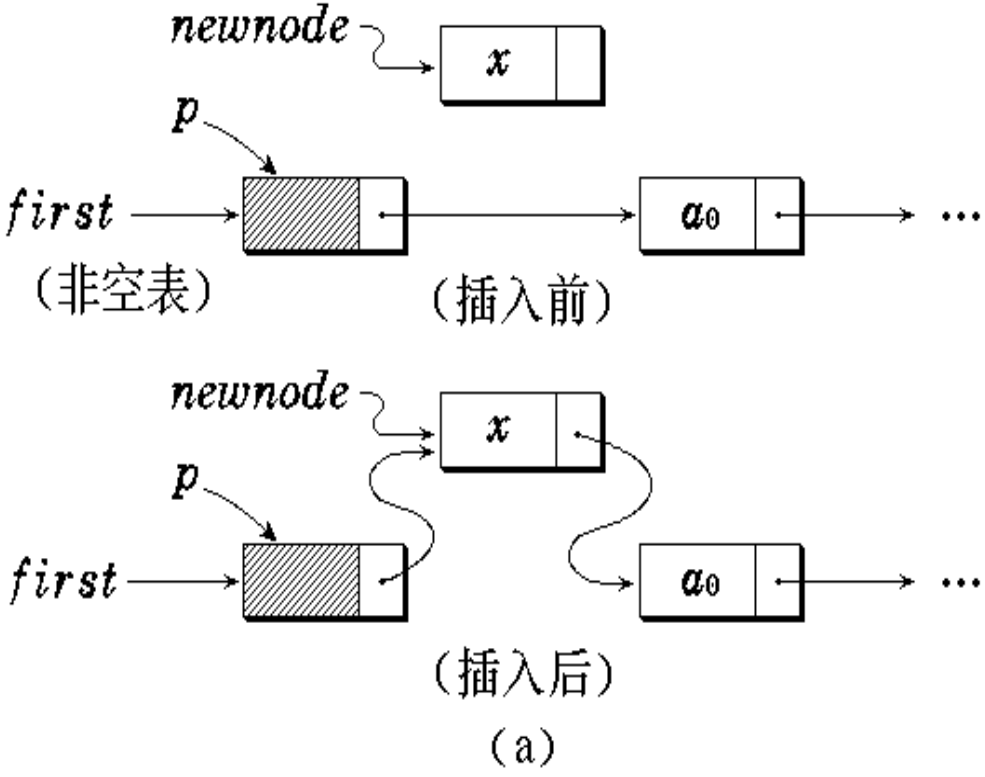
```
    else {                                  //删除表中或表尾元素
        q = p→link;
        p→link = q→link;                    //重新链接
    }
    if ( q == last ) last = p;              //可能修改last
    k = q→data;
    delete q;                               //删除q
    return k;
}
```

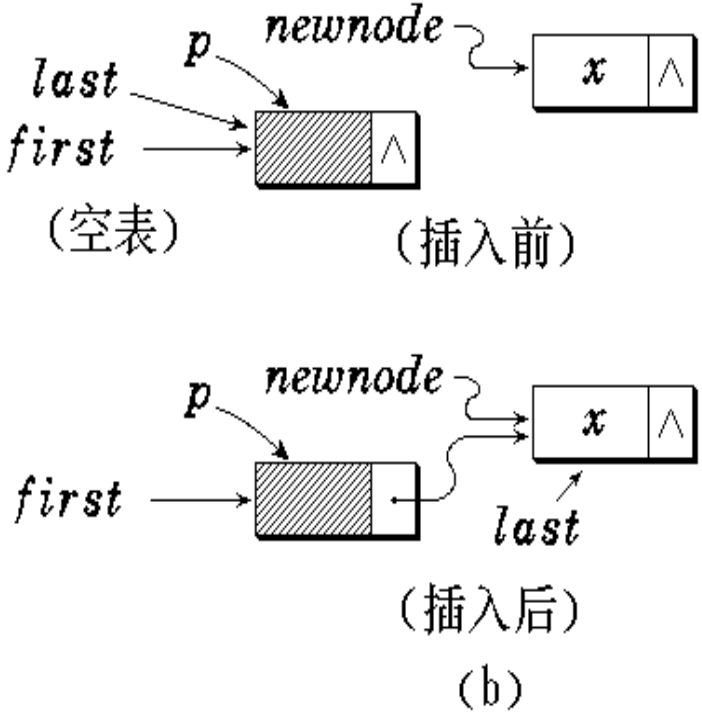
带表头结点的单链表

- 表头结点位于表的最前端，本身不带数据，仅标志表头。
- 设置表头结点的目的是统一空表与非空表的操作，简化链表操作的实现。



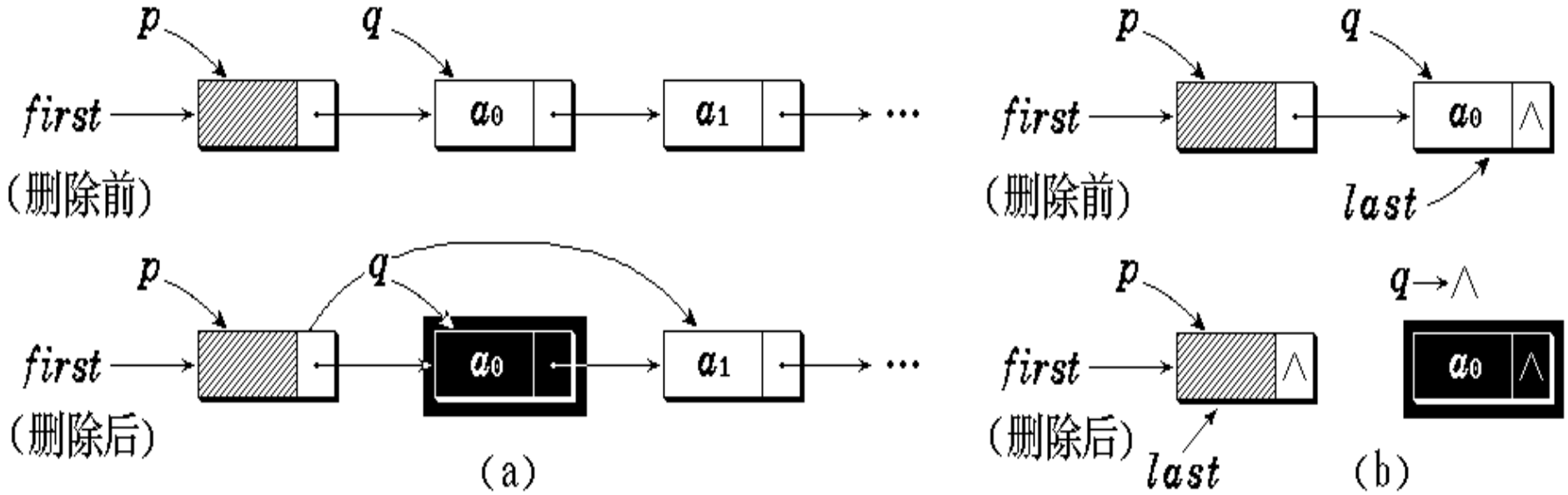
在带表头结点的单链表
第一个结点前插入新结点






```
newnode→link = p→link;
if ( p→link == NULL ) last = newnode;
p→link = newnode;
```

从带表头结点的单链表中删除第一个结点




```
q = p→link;
p→link = q→link;
delete q;
if ( p→link == NULL ) last = p;
```

单链表的类模板

- 类模板将类的数据成员和成员函数设计得更完整、更灵活。
- 类模板更易于复用。
- 在单链表的类模板定义中，增加了表头结点。

用模板定义的单链表类

```
template <class Type> class List;

template <class Type> class ListNode {
friend class List<Type>;
    Type data;                          //结点数据
    ListNode<Type> *link;               //结点链接指针
public:
    ListNode ( );                       //链表结点构造函数
    ListNode ( const Type& item );
    ListNode<Type> *NextNode ( ) { return link; }
    //给出当前结点的下一结点地址
```

```
    ListNode<Type> *GetNode ( const Type&
        item, ListNode<Type> *next );
    //创建数据为item，指针为next的新结点
    void InsertAfter ( ListNode<Type> *p );
    //在当前结点后插入结点p
    ListNode<Type> *RemoveAfter ( );
    //摘下当前结点的下一结点
};

template <class Type> class List {
    ListNode<Type> *first, *last;
public:
```

```
    List ( const Type & value ) {
        last =first = new ListNode<Type>( value );
    }                                   //构造函数
    ~List ( );                          //析构函数
    void MakeEmpty ( );                 //链表置空
    int Length ( ) const;               //求链表长度
    ListNode<Type> *Find ( Type value );
    ListNode<Type> *Find ( int i );
    int Insert ( Type value, int i );
    Type *Remove ( int i );
    Type *Get ( int i );
};
```

链表结点部分操作的实现

```
template <class Type>
ListNode<Type> :: ListNode ( ) : link (NULL) { }

template <class Type>
ListNode<Type>::
ListNode( const Type& item ) :
            data (item), link (NULL) { }

template <class Type>
void ListNode<Type>::
InsertAfter ( ListNode<Type> *p )
            { p→link = link;  link = p; }
```

```
template <class Type> ListNode<Type>
*ListNode<Type>::GetNode ( const Type
& item,  ListNode<Type> *next = NULL ) {
    ListNode<Type> *newnode =
          new ListNode<Type> ( item );
    newnode →link = next;
    return newnode;
}

template <class Type> ListNode<Type>
*ListNode<Type>::RemoveAfter ( ) {
//摘下当前结点的下一结点
```

```
    ListNode<Type> *tempptr = link;
    if ( link == NULL ) return NULL;
    //没有下一结点则返回空指针
    link = tempptr→link;        //重新链接
    return tempptr;
    //返回下一结点地址
}

template <class Type> List<Type> :: ~List ( ){
//析构函数 (链表的公共操作)
    MakeEmpty ( );  delete first;
    //链表置空，再删去表头结点
}
```

```
template <class Type>
void List<Type> :: MakeEmpty ( ) {
//删去链表中除表头结点外的所有其他结点
    ListNode<Type> *q;
    while ( first→link != NULL ) {
        q = first→link;  first→link = q→link;
        //将表头结点后第一个结点从链中摘下
        delete q;        //释放它
    }
    last = first;        //修改表尾指针
}
```

```
template <class Type>
int List<Type>::Length ( ) const {
//求单链表的长度
    ListNode<Type> *p = first→link;
    //检测指针p指示第一个结点
    int count = 0;
    while ( p != NULL ) {      //逐个结点检测
        p = p→link;  count++;
    }
    return count;
}
```

```
template <class Type>
ListNode<Type>*List <Type>::
Find ( Type value ) {
//在链表中从头搜索其数据值为value的结点
    ListNode<Type> *p = first→link;
    //检测指针 p 指示第一个结点
    while ( p != NULL && p→data != value )
        p = p→link;
    return p;
    // p 在搜索成功时返回找到的结点地址
    // p 在搜索不成功时返回空值
}
```

```
template <class Type>
ListNode<Type> *List<Type> :: Find ( int i ) {
//在链表中从头搜索第 i 个结点，不计头结点
    if ( i < −1 ) return NULL;
    if ( i == −1 ) return first;                    // i 应 ≥ 0
    ListNode<Type> *p = first→link;
    int j = 0;
    while ( p != NULL && j < i )                    // j = i 停
        { p = p→link;  j = j++; }
    return p;
}
```

```
template <class Type>
int List<Type> :: Insert ( Type value, int i ) {
//将含value的新元素插入到链表第 i 个位置
    ListNode<Type> *p = Find ( i−1 );
    // p 指向链表第 i−1个结点
    if ( p == NULL ) return 0;
    ListNode<Type> *newnode =    //创建结点
                GetNode ( value, p→link );
    if ( p→link == NULL ) last = newnode;
    p→link = newnode;                    //重新链接
    return 1;
}
```

```
template <class Type>
Type *List<Type>::Remove ( int i ) {
//从链表中删去第 i 个结点
    ListNode<Type> *p = Find (i-1), *q;
    if ( p == NULL || p→link == NULL )
        return NULL;
    q = p→link;  p→link = q→link;    //重新链接
    Type value = new Type ( q→data );
    if ( q == last ) last = p;
    delete q;
    return &value;
}
```

```
template <class Type>
Type *List<Type>::Get ( int i ) {
//提取第 i 个结点的数据
    ListNode<Type> *p = Find ( i );
    // p 指向链表第 i 个结点
    if ( p == NULL || p == first )
        return NULL;
    else return & p→data;
}
```

链表的游标类 (*Iterator*)

- 游标类主要用于单链表的搜索。
- 游标类的定义原则：
  - ***Iterator***类是***List***类和***ListNode***类的友元类。
  - ***Iterator***对象引用已有的***List***类对象。
  - ***Iterator***类有一数据成员***current***，记录对单链表最近处理到哪一个结点。
  - ***Iterator***类提供若干测试和搜索操作

表示链表的三个类的模板定义

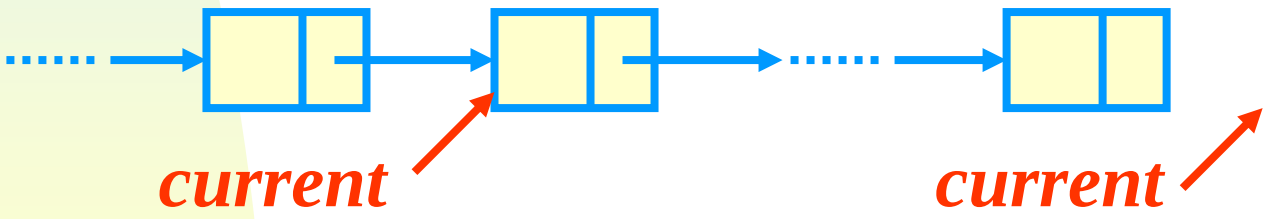
```
enum Boolean { False, True };
template <class Type> class List;
template <class Type> class ListIterator;

template <class Type> class ListNode {  //表结点
friend class List <Type>;
friend class ListIterator <Type>;
public:
      ……
private:
   Type data;
   ListNode<Type> *link;
};
```

```
template <class Type> class List {        //链表类
public:
    ……
private:
    ListNode<Type> *first, *last;
};

template <class Type> class ListIterator {
public:
    ListIterator ( const List<Type> & l )
                    : list ( l ), current ( l.first ) { }
    //构造函数: 引用链表 l, 表头为当前结点
```

```
    Boolean NotNull ( );
    //检查链表中当前指针是否非空
    Boolean NextNotNull ( );
    //检查链表中下一结点是否非空
    ListNode <Type> *First ( );
    //返回链表表头指针
    ListNode <Type> *Next ( );
    //返回链表当前结点的下一个结点的地址
private:
    const List<Type> & list;      //引用已有链表
    ListNode<Type> *current;   //当前结点指针
}
```

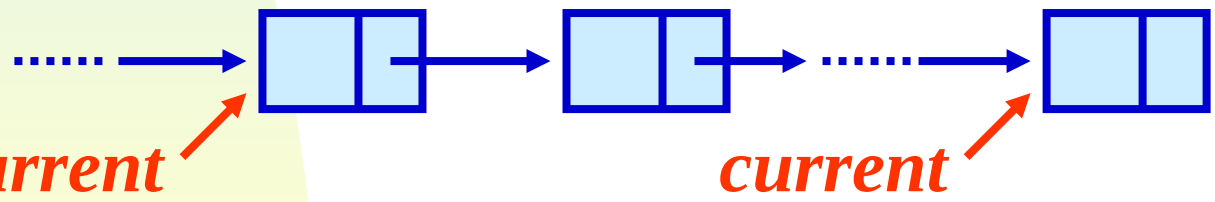
链表的游标类成员函数的实现

```
template <class Type>
Boolean ListIterator<Type> :: NotNull ( ) {
//检查链表中当前元素是否非空
    if ( current != NULL ) return True;
    else return False;
}
```



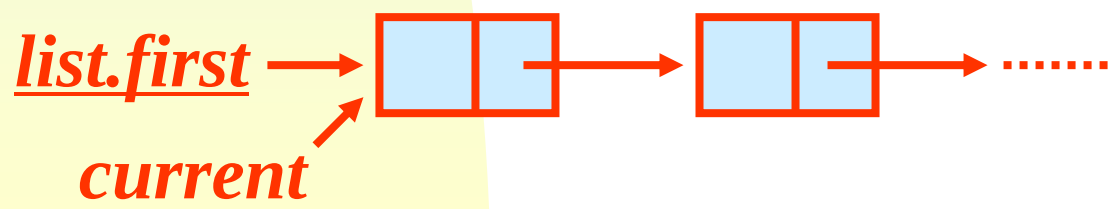
情况 1 返回*True*　　情况 2 返回*False*

```
template <class Type>
Boolean ListIterator<Type>::NextNotNull ( ) {
//检查链表中下一元素是否非空
    if ( current != NULL &&
        current→link != NULL ) return True;
    else return False;
}
```



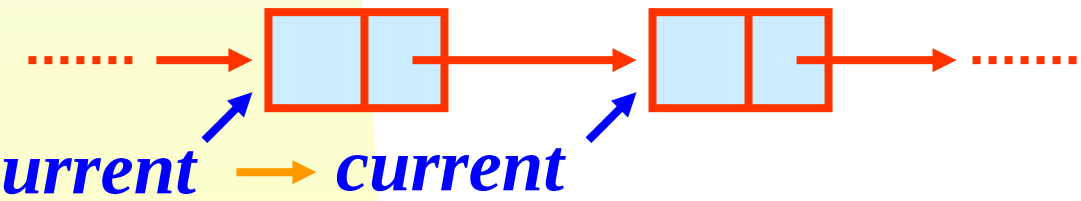
情况 1 返回 True　　情况 2 返回 False

```
template <class Type>
ListNode<Type>* ListIterator<Type> :: First ( ) {
//返回链表中第一个结点的地址
    if ( list.first→link != NULL ){
        current = list.first;
        return &current→data;
    }
    else { current = NULL;  return NULL; }
}
```



约定链表中
没有表头结点

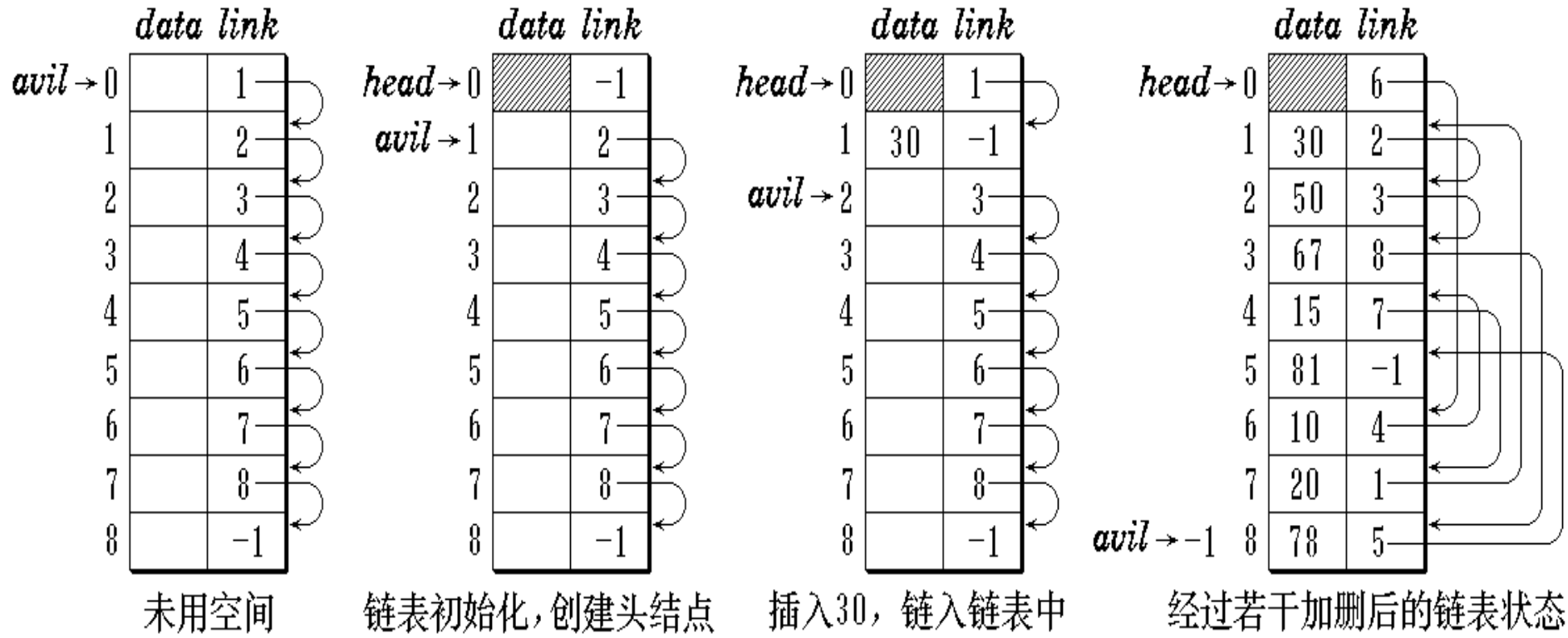
```
template <class Type>
ListNode<Type>* ListIterator<Type> :: Next ( ) {
//返回链表中当前结点的下一个结点的地址
    if ( current != NULL
            && current→link != NULL ) {
        current = current→link;
        return & current→data;
    }
    else { current = NULL;  return NULL; }
}
```



利用游标类 (*iterator*) 计算元素的和

```
int sum ( const List<int> &l ) {
    ListIterator<int> li ( l );
    //定义游标对象, current 指向 li.first
    if ( ! li.NotNull ( ) ) return 0;
    //链表为空时返回0
    int retval = * li.First( )→data;   //第一个元素
    while ( li.nextNotNull ( ) )          //链表未扫描完
          retval += * li.Next( )→data;      //累加
    return retval;
}
```

静态链表结构　利用数组定义，运算过程中存储空间大小不变



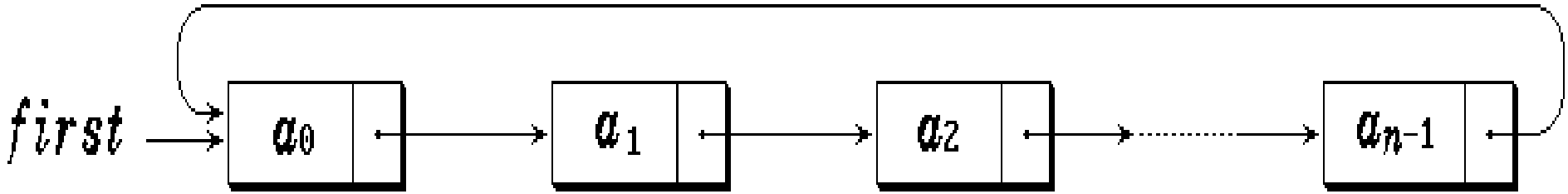
分配节点：　*j = avil; avil = A[avil].link;*

释放节点：　*A[i].link = avil; avil = i;*

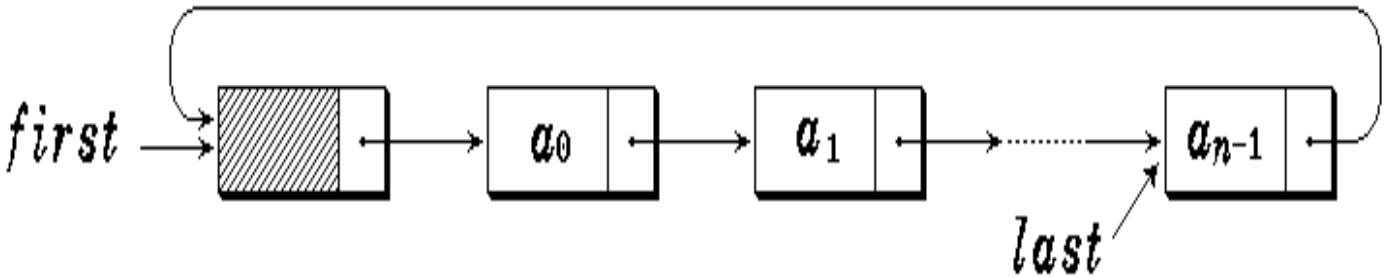
循环链表 (*Circular List*)

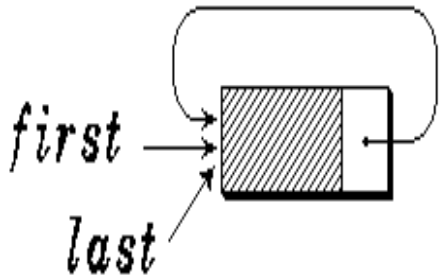
- 循环链表是单链表的变形。
- 循环链表最后一个结点的 ***link*** 指针不为 **0 (*NULL*)**，而是指向了表的前端。
- 为简化操作，在循环链表中往往加入表头结点。
- 循环链表的特点是：只要知道表中某一结点的地址，就可搜寻到所有其他结点的地址。

- 循环链表的示例



- 带表头结点的循环链表





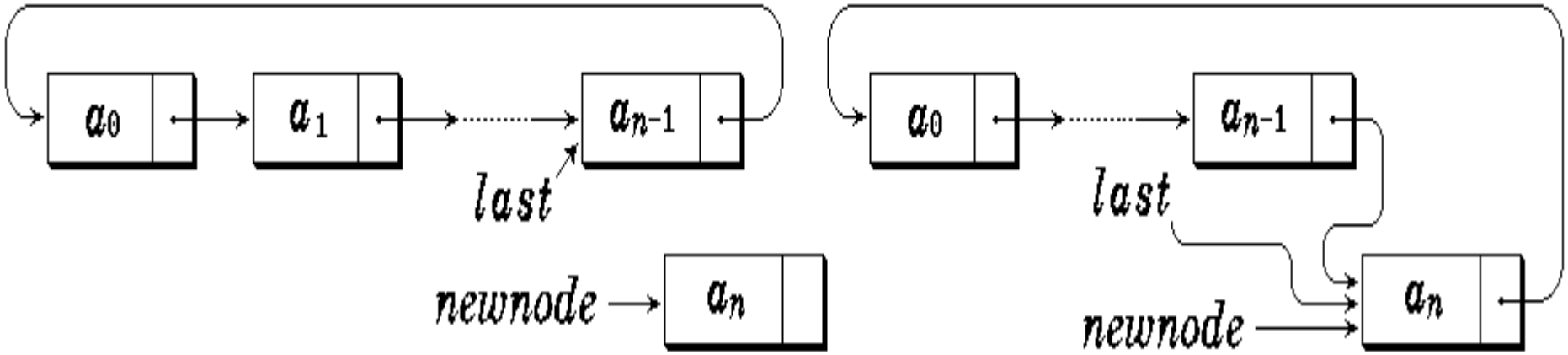
循环链表类的定义

```
template <class Type> class CircList;

template <class Type> class CircListNode {
friend class CircList;
public:
    CircListNode ( Type d = 0,
        CircListNode<Type> *next = NULL ) :
            data ( d ), link ( next ) { }    //构造函数
private:
    Type data;
    CircListNode<Type> *link;
}
```

```
template <class Type> class CircList {
public:
    CircList ( Type value );
    ~CircList ( );
    int Length ( ) const;
    Boolean IsEmpty ( )
        { return first→link == first; }
    Boolean Find ( const Type & value );
    Type getData ( ) const;
    void Firster ( ) { current = first; }
    Boolean First ( );
    Boolean Next ( );
```

```
    Boolean Prior ( );
    void Insert ( const Type & value );
    void Remove ( );
private:
    CircListNode<Type> *first, *current, *last;
};
```

插入

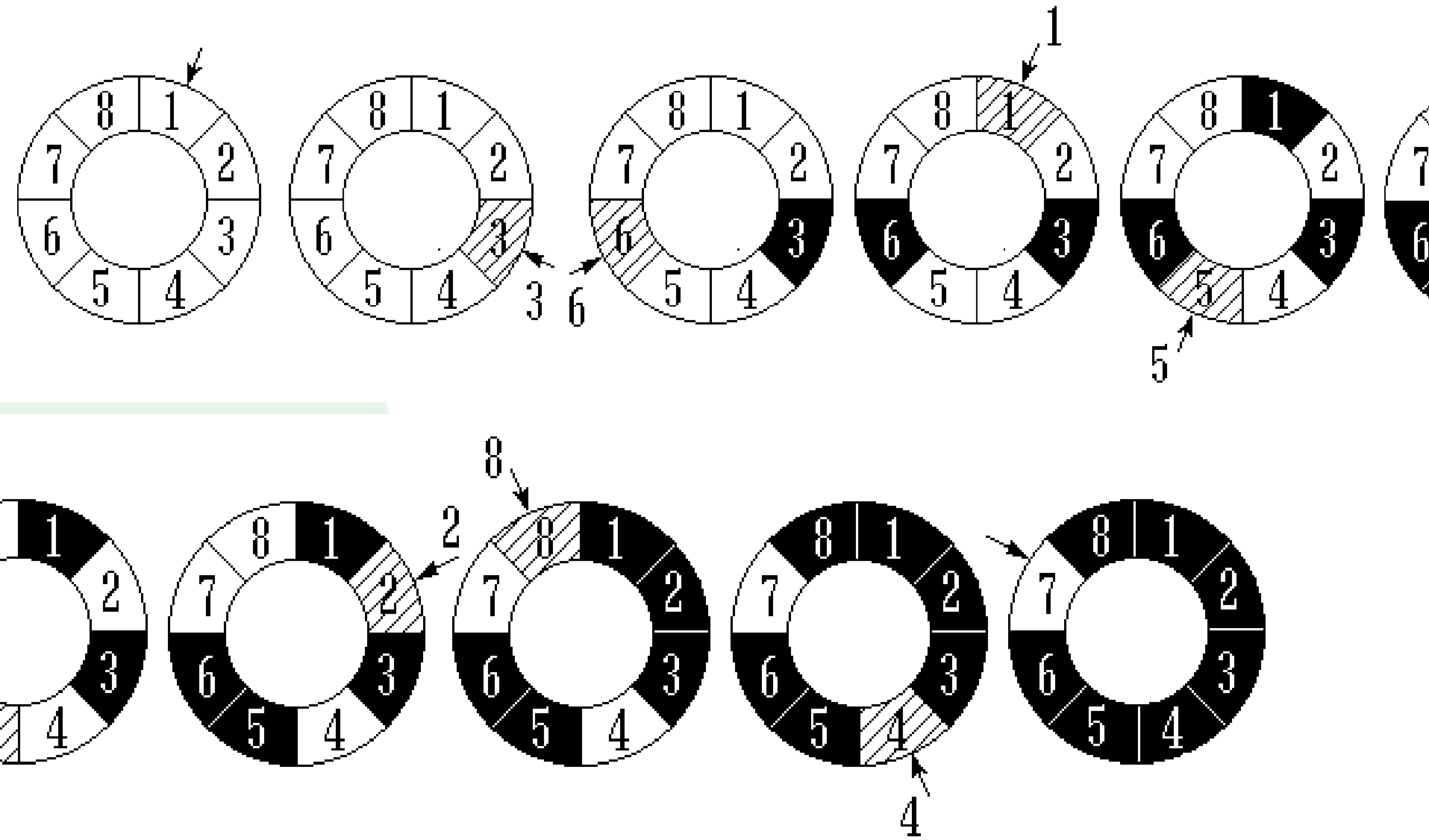


用循环链表求解约瑟夫问题

- 约瑟夫问题的提法

  n 个人围成一个圆圈，首先第 1 个人从 1 开始一个人一个人顺时针报数，报到第 *m* 个人，令其出列。然后再从下一个人开始，从 1 顺时针报数，报到第 *m* 个人，再令其出列，…，如此下去，直到圆圈中只剩一个人为止。此人即为优胜者。

- 例如 $n = 8$ $m = 3$



约瑟夫问题的解法

```
#include <iostream.h>
#include "CircList.h"
Template<Type>
void CircList<Type>:: Josephus ( int n, int m ) {
    Firster ( );
    for ( int i = 0; i < n−1; i++ ) {     //执行n−1次
        for ( int j = 0; j < m−1; j++ ) Next ( );
        cout << "出列的人是" << getData ( )
             << endl;                      //数m−1个人
        Remove ( );                        //删去
    }
```

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

多项式及其相加

- 在多项式的链表表示中每个结点增加了一个数据成员***link***，作为链接指针。

$data \equiv Term$

coef	*exp*	*link*

- 优点是：
  - 多项式的项数可以动态地增长，不存在存储溢出问题。
  - 插入、删除方便，不移动元素。

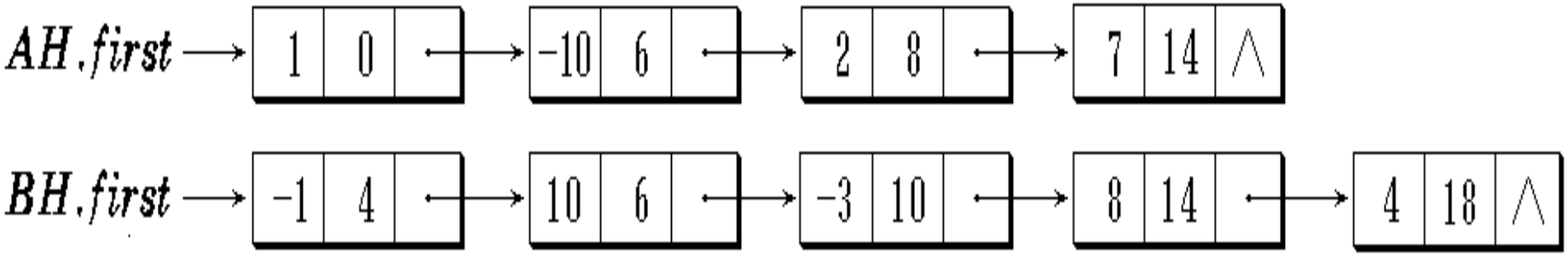
多项式(*polynomial*)类的链表定义

```
struct Term {
    int coef;
    int exp;
    Term ( int c, int e ) { coef = c;  exp = e; }
};

class Polynomial     {
    List<Term> poly;                          //构造函数
    friend Polynomial & operator +
         ( Polynomial &, Polynomial &);     //加法
};
```

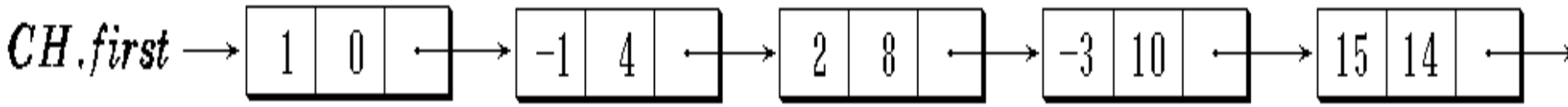
多项式链表的相加

$AH = 1 - 10x^6 + 2x^8 + 7x^{14}$

$BH = -x^4 + 10x^6 - 3x^{10} + 8x^{14} + 4x^{18}$



（a）两个相加的多项式



（b）相加结果的多项式

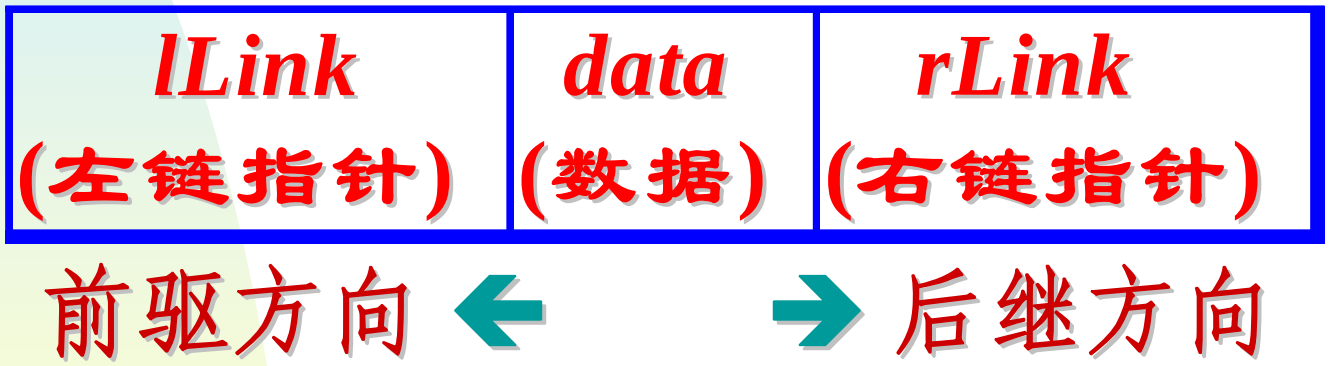
```
Polynomial & operator + ( Polynomial & ah,
                Polynomial & bh ) {
//多项式加法, 结果由ah表头结点指示
   ListNode<Term> *pa, *pb, *pc, *p;
   ListIterator<Term> Aiter ( ah.poly );
   ListIterator<Term> Biter ( bh.poly );
   //建立两个多项式对象 Aiter、Biter
   pa = pc = Aiter.First ( );   // ah 检测指针
   pb = p = Biter.First ( );    // bh 检测指针
   pa = Aiter.Next ( );  pb = Biter.Next ( );
   // pa, pb 越过表头结点
   delete p;
```

```
while ( Aiter.NotNull ( ) && Biter.NotNull ( ) )
    switch ( compare ( pa→exp, pb→exp ) ) {
    case '=' :                          // pa指数等于pb指数
        pa→coef = pa→coef + pb→coef;
        p = pb;   pb = Biter.Next ( );  delete p;
        if ( !pa→coef ) {       //指数相加结果为0
            p = pa;  pa = Aiter.Next ( );
            delete p;
        }
        else {                          //指数相加结果不为0
            pc→link = pa;   pc = pa;   //链入结果链
            pa = Aiter.Next ( );
        }
```

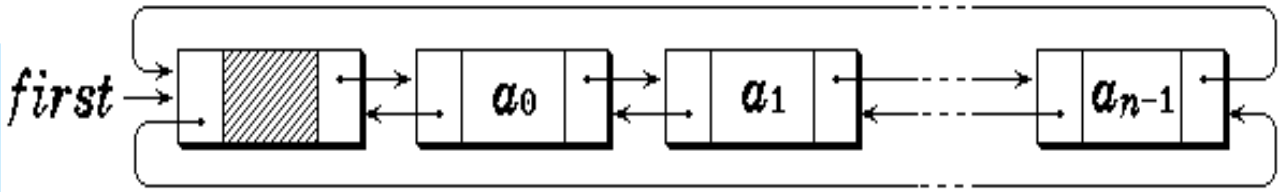
```
            break;
        case '>' :                          // pa指数大于pb指数
            pc→link = pb;   pc = pb;
            pb = Biter.Next ( );   break;
        case '>' :                          // pa指数小于pb指数
            pc→link = pa;   pc = pa;
            pa = Aiter.Next ( );
    }
    if ( Aiter.NotNull ( ) )  pc→link = pa;
    else  pc→link = pb;
}
```

双向链表 (*Doubly Linked List*)

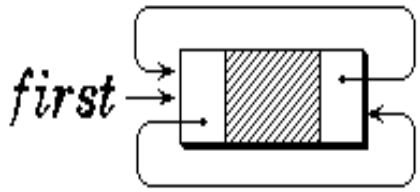
- 双向链表是指在前驱和后继方向都能游历(遍历)的线性链表。
- 双向链表每个结点结构:



- 双向链表通常采用带表头结点的循环链表形式。



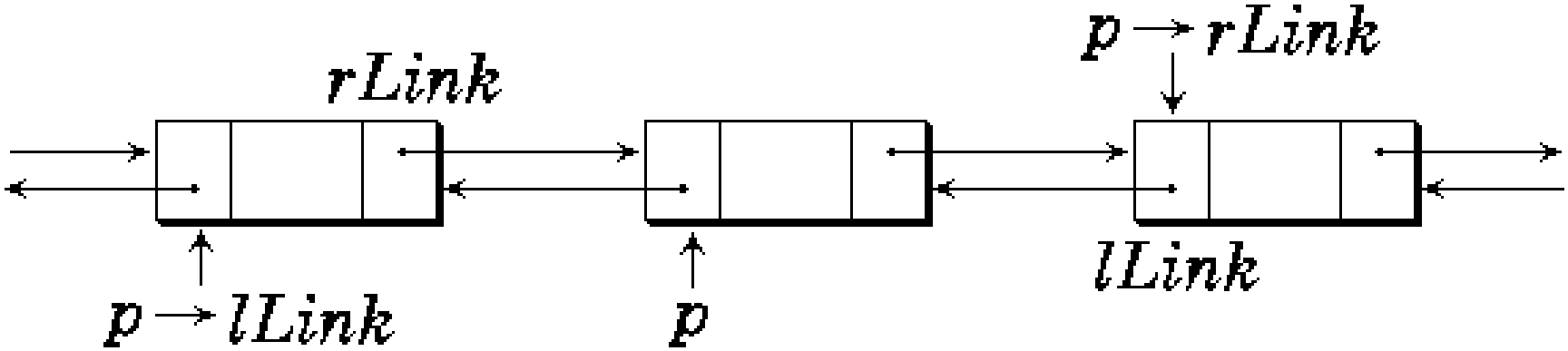
非空表



空表

- 结点指向

$$p == p \rightarrow lLink \rightarrow rLink == p \rightarrow rLink \rightarrow lLink$$



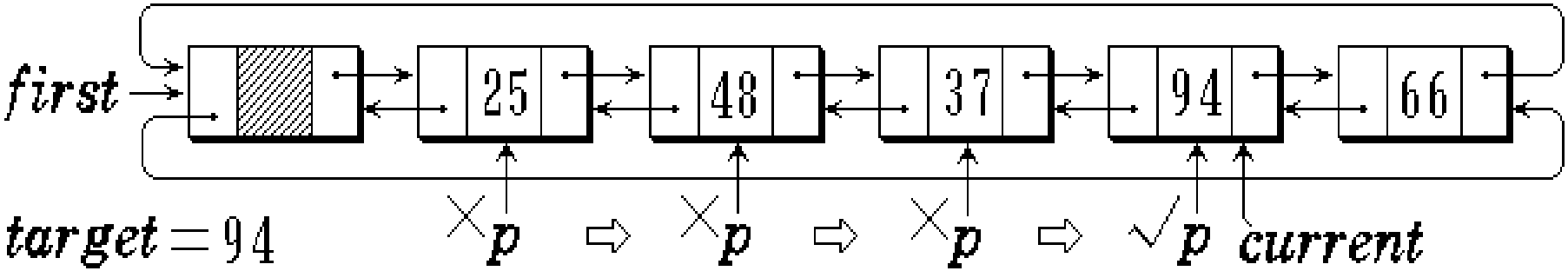
双向循环链表类的定义

```
template <class Type> class DblList;

template <class Type> class DblNode {
friend class DblList<Type>;
private:
    Type data;                                  //数据
    DblNode<Type> *lLink, *rLink;               //指针
    DblNode ( Type value,                       //构造函数
        DblNode<Type> *left,
        DblNode<Type> *right ) :
        data (value), lLink (left), rLink (right) { }
```

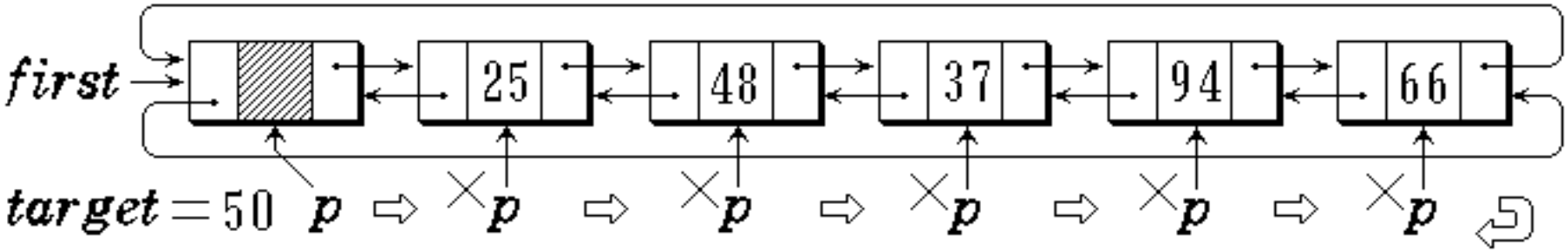
```
    DblNode ( Type value ) : data (value),
        lLink (NULL), rLink (NULL) { }
};

template <class Type> class DblList {
public:
    DblLIst ( Type uniqueVal );
    ~DblList ( );
    int Length ( ) const;
    int IsEmpty ( ) { return first→rlink == first; }
    int Find ( const Type & target );
    Type getData ( ) const;
```

```
    void Firster ( ) { current = first; }
    int First ( );
    int Next ( );
    int Prior ( );
    int operator ! ( )
        { return current != NULL; }
    void Insert ( const Type & value );
    void Remove ( );
private:
    DblNode<Type> *first, *current;
};
```

双向循环链表的搜索算法



搜索成功

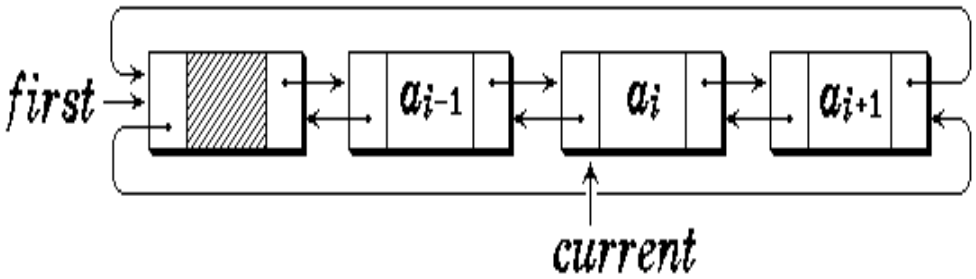


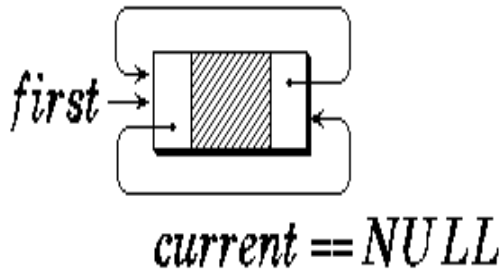
搜索不成功

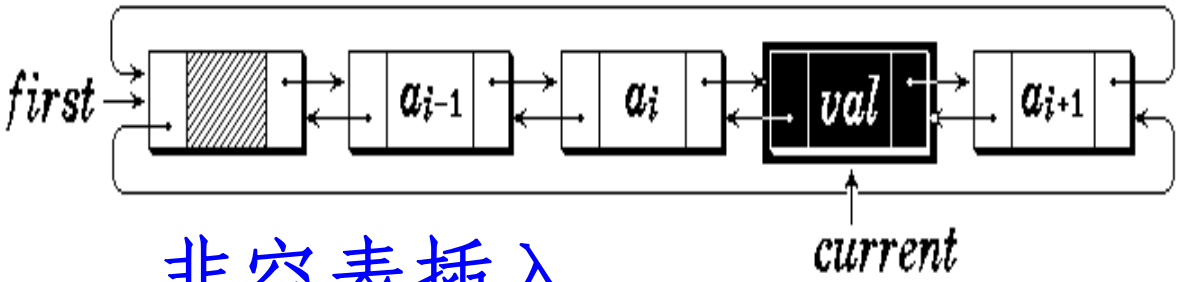
```
template <class Type> int DblList<Type>::
Find ( const Type & target ) {
//在双向循环链表中搜索含target的结点，
//搜索成功返回1，否则返回0。
    DblNode<Type> *p = first→rLink;
    while ( p != first && p→data != target )
          p = p→rLink;                    //循链搜索
    if ( p != first ) { current = p;  return 1; }
    return 0;
}
```

双向循环链表的插入算法

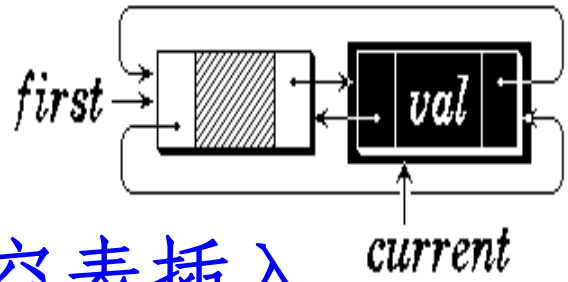
```
p→lLink = current;
p→rLink =current→rLink;
current→rLink = p;
current = current→rLink;
current→rLink→lLink = current;
```







非空表插入



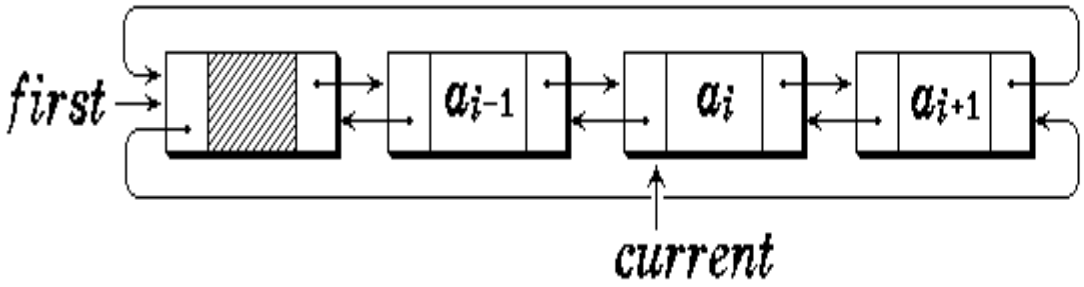
空表插入

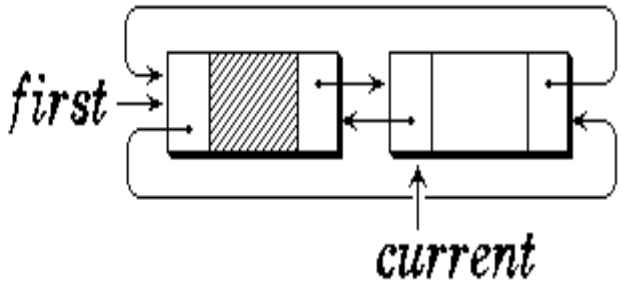
```
template <class Type> void DblList<Type>::
Insert ( const Type & value ) {
    if ( current == NULL )          //空表情形
        current = first→rLink =
                new DblNode ( value, first, first );
    else {                                  //非空表情形
        current→rLink =new DblNode
                ( value, current, current→rLink );
         current = current→rLink;
    }
    current→rLink→lLink = current;
}
```

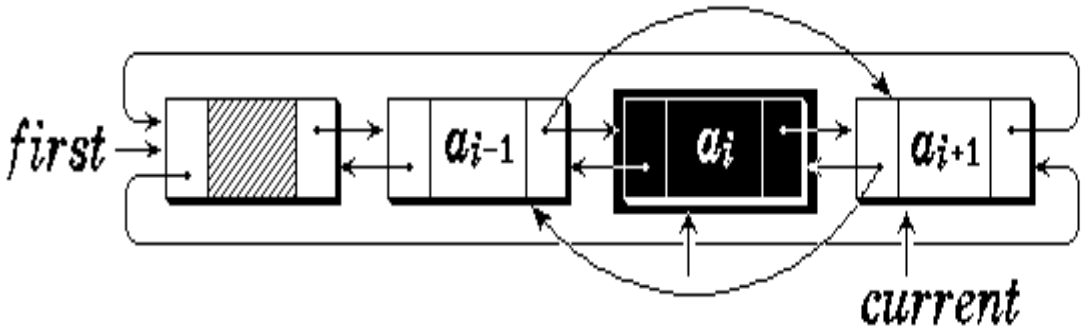
双向循环链表的删除算法

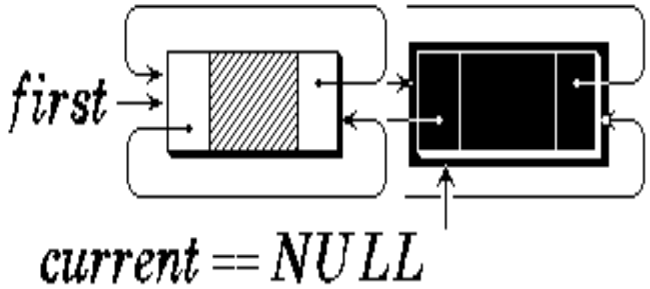
current→rLink→lLink = current→lLink;

current→lLink→rLink = current→rLink;









```
template <class Type>
void DblList<Type>::Remove ( ) {
    if ( current != NULL ) {
        DblNode *temp = current;          //被删结点
        current = current→rLink;          //下一结点
        current→lLink = temp→lLink;       //从链中摘下
        temp→lLink→rLink = current;
        delete temp;                      //删去
        if ( current == first )
            if ( IsEmpty ( ) ) current = NULL;
            else  current = current→rLink;
    }
}
```

其他双向循环链表的公共操作

```
template <class Type>
DblList<Type>::DblLIst ( Type uniqueVal ) {
//双向循环链表的构造函数,创建表头结点
    first = new DblNode<Type> ( uniqueVal );
    first→rLink = first→lLink = first;
    current = NULL;
}
```

```
template <class Type>
int DblList<Type>::Length ( ) const {
//求双向循环链表的长度(不计表头结点)
    DblNode<Type> * p = first→rLink;
    int count = 0;
    while ( p != first )
        { p = p→rLink;  count++; }
    return count;
}
```

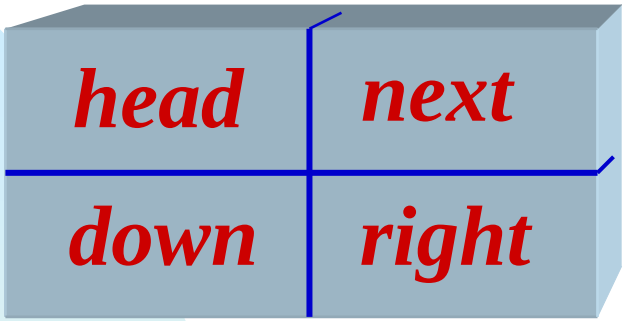
```
template <class Type>
int DblList<Type>::First ( ) {
    if ( !IsEmpty ( ) )      //跳过表头结点的第一个
        { current = first→rLink;  return 1; }
    current = NULL;  return 0;
}

template <class Type>
int DblList<Type>::Next ( ) {
    if ( current→rLink == first )
            { current = NULL;  return 0; }
    current = current→rLink;  return 1;
}
```

```
template <class Type>
int DblList<Type>::Prior ( ) {
    if ( current→lLink == first )
        { current = NULL;  return 0; }
    current = current→lLink;  return 1;
}
```

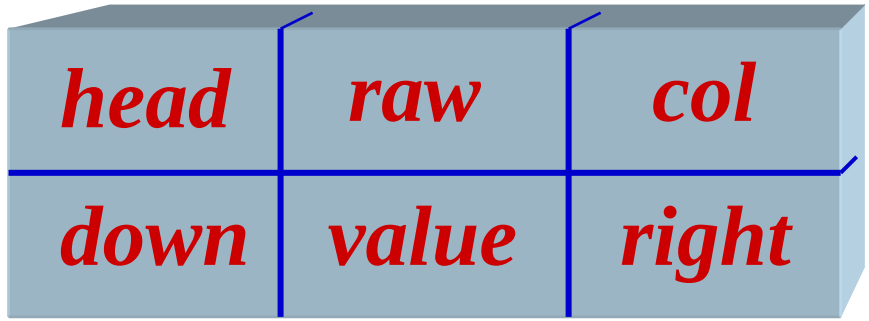
稀疏矩阵

- 在矩阵操作(+、-、*、/)时矩阵非零元素会发生动态变化，用稀疏矩阵的链接表示可适应这种情况。
- 稀疏矩阵的链接表示采用正交链表：行链表与列链表十字交叉。
- 行链表与列链表都是带表头结点的循环链表。用表头结点表征是第几行，第几列。

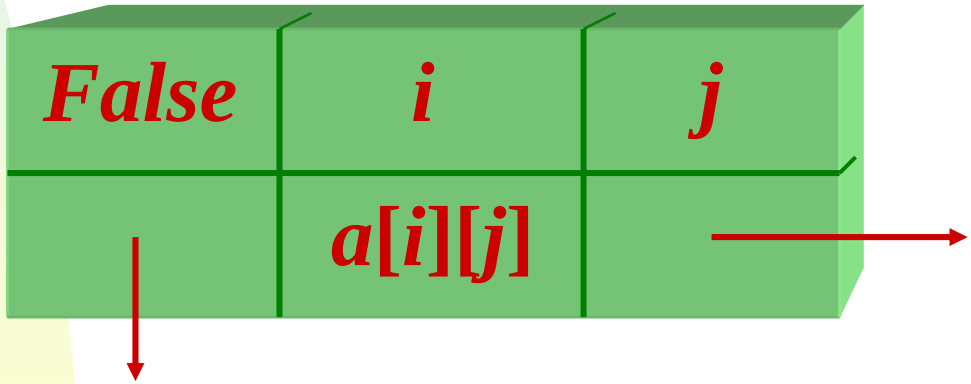
稀疏矩阵的结点



(a) 表头结点

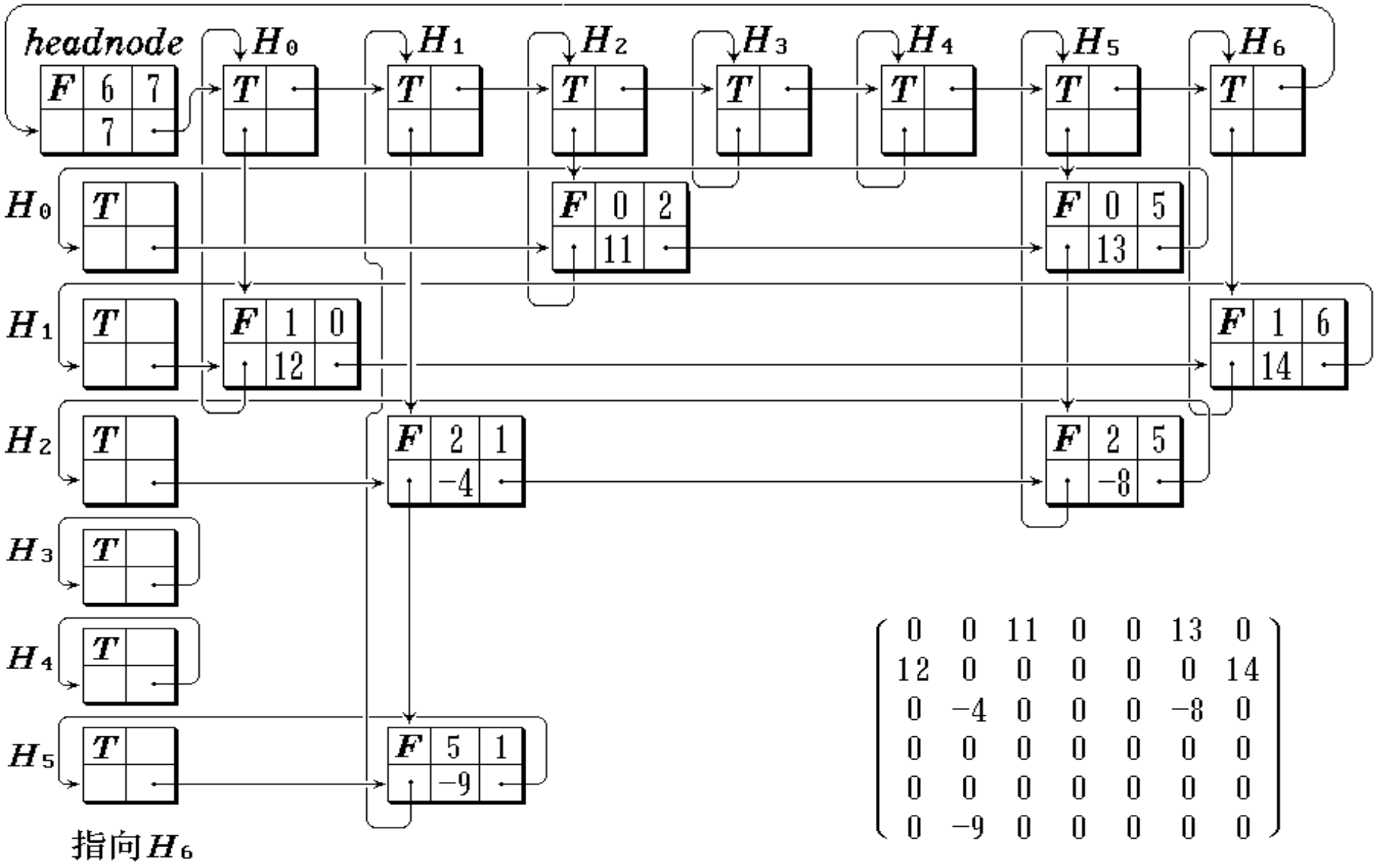


(b) 非零元素结点



(c) 建立*a*[*i*][*j*]结点

稀疏矩阵的正交链表表示的示例



$$\begin{pmatrix} 0 & 0 & 11 & 0 & 0 & 13 & 0 \\ 12 & 0 & 0 & 0 & 0 & 0 & 14 \\ 0 & -4 & 0 & 0 & 0 & -8 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -9 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

稀疏矩阵的链表表示的类定义

```
enum Boolean { False, True };
struct Triple { int row, col;  float value; };

class Matrix;

class MatrixNode {          //矩阵结点定义
friend class Matrix;
friend istream &operator >> ( istream &,
     Matrix & );            //矩阵输入重载函数
private:
    MatrixNode *down, *right;      //列/行链指针
    Boolean head;                  //结点类型
```

```
    Union { Triple triple;  MatrixNode *next; }
    //矩阵元素结点(False)或链头结点(True)
    MatrixNode ( Boolean, Triple* );
    //结点构造函数
}

MatrixNode::MatrixNode ( Boolean b, Triple *t ) {
//矩阵结点构造函数
    head = b;                    //结点类型
    if ( b ) { right = next = this; }
    else triple = *t;
}
```

```
typedef MatrixNode *MatrixNodePtr;
//一个指针数组,用于建立稀疏矩阵

class Matrix {
friend istream& operator >> ( istream &,
        Matrix & );                 //矩阵输入
public:
    ~Matrix ( );                    //析构函数
private:
    MatrixNode *headnode;    //稀疏矩阵的表头
};
```

用正交链表表示的稀疏矩阵的建立

```
istream & operator
    >> ( istream & is, Matrix & matrix ) {
  Triple s;  int p;
  is >> s.row >> s.col >> s.value;
  //输入矩阵的行数,列数和非零元素个数
  if ( s.row > s.col ) p = s.row;
  else p = s.col;                    //取行、列数大者
  matrix.headnode =                  //整个矩阵表头结点
          new MatrixNode ( False, &s );
  if ( !p ) {                        //零矩阵时
    matrix.headnode→right = matrix.headnode;
```

```
    return is;
}
MatrixNodePtr *H = new MatrixNodePtr ( p );
//建立表头指针数组，指向各链表的表头
for ( int i = 0; i < p; i++ )
    H[i] = new MatrixNode ( True, 0 );
int CurrentRow = 0;
MatrixNode *last = H[0];   //当前行最后结点
for ( i = 0; i < s.value; i++ ) {   //建立矩阵
   Triple t;
   is >> t.row >> t.col >> t.value;
   //输入非零元素的三元组
```

```
if ( t.row > CurrentRow ) {
//如果行号大于当前行,闭合当前行
    last→right = H[CurrentRow];
    CurrentRow = t.row;
    last = H[CurrentRow];
}
last = last→right =          //链入当前行
      new MatrixNode ( False, &t );
H[t.col]→next = H[t.col]→next→down
    =  last;                 //链入列链表
}
last→right = H[CurrentRow];  //闭合最后一行
```

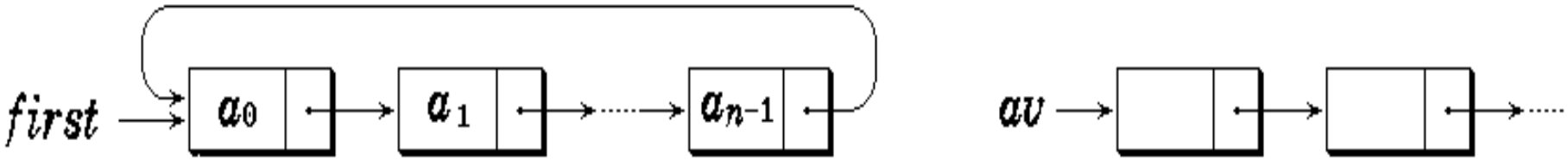
```
    //闭合各列链表
    for ( i = 0; i < s.col; i++ )
        H[i]→next→down = H[i];
    //链接所有表头结点
    for ( i = 0; i < p−1; i++ )
        H[i]→next =H[i+1];
    H[p−1]→next = matrix.headnode;
    matrix.headnode→right = H[0];
    delete [ ] H;
    return is;
}
```

稀疏矩阵的删除

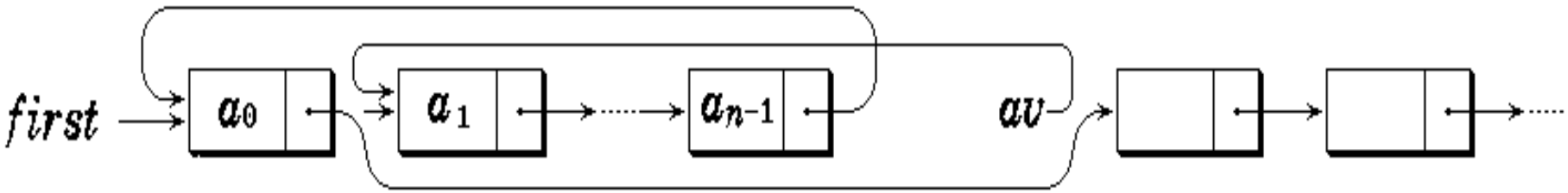
- 为执行稀疏矩阵的删除，需要使用可利用空间表来管理回收的空间。
- 可利用空间表是单链表结构，只允许在表头插入或删除，其表头指针为*av*。
- 使用可利用空间表，可以高效地回收循环链表。
- 如果需要建立新的稀疏矩阵，还可以从可利用空间表中分配结点。

利用可利用空间表回收循环链表

```
if ( first != NULL ) {
    CircListNode<Type> *second = first→link;
    first→link = av;    av = second;
    first = NULL;
}
```



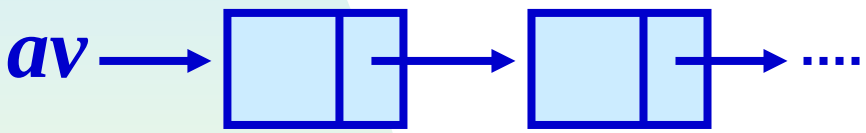
（a）回收前的循环链表和可利用空间表



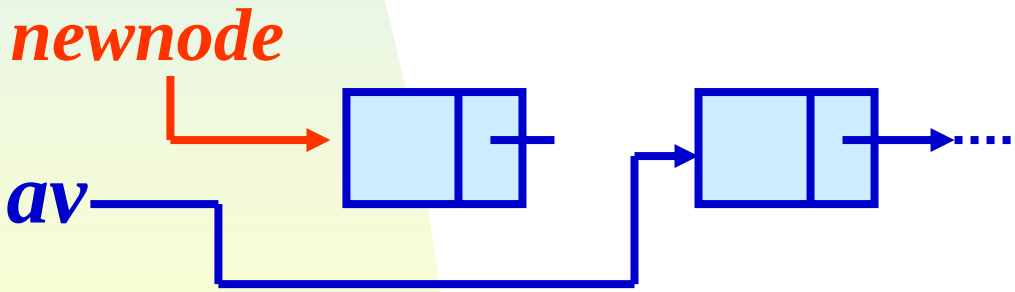
（b）回收后的可利用空间表

从可利用空间表分配结点

```
if ( av == NULL ) newnode =
      new CircListNode<Type>;
else { newnode = av;  av = av→link; }
```



分配前的可利用空间表



分配后的可利用空间表和新结点

用正交链表表示的稀疏矩阵的删除

```
Matrix::~Matrix ( ) {
    if ( headnode == NULL ) return;
    MatrixNode *x = headnode→right, *y;
    headnode→right = av;  av = headnode;
    while ( x != headnode ) {
        y = x→right;  x→right = av;  av = y;
        x = x→next;
    }
    headnode = NULL;
}
```

小结 需要复习的知识点

- 单链表
  - 单链表的结构和类定义;
  - 单链表中的插入与删除;
  - 带表头结点的单链表;
  - 用模板定义的单链表类;
  - 静态链表
- 单链表的递归算法
  - 搜索含 *x* 结点

- 删除含 *x* 结点
- 统计单链表中结点个数

- 循环链表
  - 循环链表的类定义
  - 用循环链表解约瑟夫问题;
- 多项式及其相加
  - 多项式的类定义
  - 多项式的加法
- 双向链表
  - 双向循环链表的插入和删除算法