第二章 线性表

- 线性表
- 顺序表
- 单链表
- 线性链表的其它变形
- 单链表的应用：多项式及其运算
- 静态链表
- 本章小结

2.1 线性表 (Linear List)

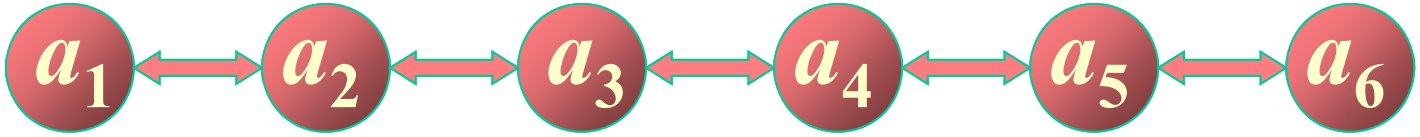
- **线性表的定义和特点**
  - 定义 n (≥ 0) 个数据元素的有限序列，记作

    $$(a_1, a_2, \ldots, a_n)$$

    a_i 是表中数据元素， n 是表长度。
  - 原则上讲，线性表中表元素的数据类型可以不相同，但采用的存储表示可能会对其有限制。

线性表的特点

- 除第一个元素外，其它每一个元素有一个且仅有一个**直接前驱**。
- 除最后一个元素外，其它每一个元素有一个且仅有一个**直接后继**。



线性表的操作

- 计算表的长度n；
- 从左到右（或从右到左）遍历表的元素；
- 访问第i个元素，$0 \le i < n$；
- 将新值赋予第i个元素，$0 \le i < n$；
- 将新元素插入第i个位置，$0 \le i < n$，使原来的第i, i+1, …, n-1个元素变为第i+1, i+2, …, n个元素；
- 删除第i个元素，$0 \le i < n$，使原来的第i+1, i+2, …, n-1个元素变为第i, i+1, …, n-2个元素。

线性表的类定义

```
enum bool { flase, true };
template <class Type>
class LinearList {
public:
    LinearList ( );  //构造函数
    ~LinearList ( );  //析构函数
    virtual int Size ( ) const = 0;  //求表最大体积
    virtual int Length ( ) const = 0;  //求表长度
    virtual int Search ( Type &x ) const = 0;
    //在表中搜索给定值 x
    virtual int Locate ( int i ) const = 0;
    //在表中定位第 i 个元素位置
    virtual Type * GetData ( int i ) const = 0;  //取第 i 个表项的值
```

```
    virtual void SetData ( int i, Type &x ) = 0;
    //修改第 i 个表项的值为 x
    virtual bool Insert ( int i, Type &x ) = 0;
    //在第 i 个表项后插入 x
    virtual bool Remove ( int i, Type &x ) = 0;
    //删除第 i 个表项，通过 x 返回
    virtual bool IsEmpty ( ) const = 0;  //判表空
    virtual bool IsFull ( ) const = 0;  //判表满
    virtual void Sort ( ) = 0;  //排序
    virtual void Input ( ) = 0;  //输入
    virtual void Output ( ) = 0;  //输出
    virtual LinearList <Type> operator =
                ( LinearList <Type> &L ) = 0;  //复制
};
```

- 如何表示线性表的结构，从而高效实现这些操作?
- 最通常的方法是用程序设计语言提供的**数组**，即用数组的第i个单元表示线性表的a_i元素。
- 数组第i个单元与第i+1个单元在物理上是连续存放的，因此称上述方法为**顺序映射**。
  - 顺序映射使随机存取表中的任何元素的时间是O(1)，但插入和删除第i个元素将导致其后续元素的迁移。

2.2 顺序表 (Sequential List)

- **顺序表的定义和特点**
  - **定义** 将线性表中的元素相继存放在一个连续的存储空间中。
  - 可利用一维数组描述存储结构。
  - **特点** 线性表的数组存储方式。
  - **限制** 所有元素有相同数据类型。

	0	1	2	3	4	5
data	25	34	57	16	48	09

顺序表的静态存储表示

```
#define maxSize 100
typedef int Type;
typedef struct {
      Type data[maxSize];
       int n;
} SeqList;
```

顺序表的动态存储表示

```
typedef int Type;
typedef struct {
      Type *data;
       int maxSize, n;
} SeqList;
```

顺序表(SeqList)类定义

```
const int defaultSize = 100;
template <class Type>
class SeqList {
      Type *data;  //存放数组
      int maxSize;  //最大可容纳表项的项数
      int last;  //当前已存表项的最后位置（从 0 开始）
      void ReSize( int newSize );  //改变 data 数组空间大小
public:
      SeqList ( int sz = defaultSize );  //构造函数
      SeqList ( SeqList <Type> &L );  //复制构造函数
      ~SeqList ( ) { delete [ ] data; }  //析构函数
      int Size( ) const { return maxSize; }
      //计算表最大可容纳表项个数
```

```
int Length ( ) const { return last+1; } //析构函数
int Search ( Type &x ) const;
//搜索 x 在表中位置，函数返回表项序号
int Locate ( int i ) const;
//定位第 i 个表项，函数返回表项序号
Type * GetData ( int i ) const  //取第 i 个表项的值
    { return ( i > 0 && i <= last+1 ) ? &data[i-1] : NULL; }
Type * SetData ( int i, Type &x ) const
 //用 x 修改第 i 个表项的值
    { if ( i > 0 && i < =last+1 ) data[i-1]=x; }
```

```
    bool Insert ( int i, Type &x );  //插入 x 在第 i 个表项之后
    bool Remove ( int i, Type &x );
    //删除第 i 个表项，通过 x 返回表项的值
    bool IsEmpty ( ) { return ( last == -1 ) ? true : false; }
    //判表空否，空则返回 true；否则返回 false
    bool IsFull ( ) { return ( last == maxSize-1 ) ? true : false; }
    //判表满否，空则返回 true；否则返回 false
    void input ( );  //输入
    void output ( );  //输出
    SeqList <Type> operator = ( SeqList <Type> &L );
    //表整体赋值
};
```

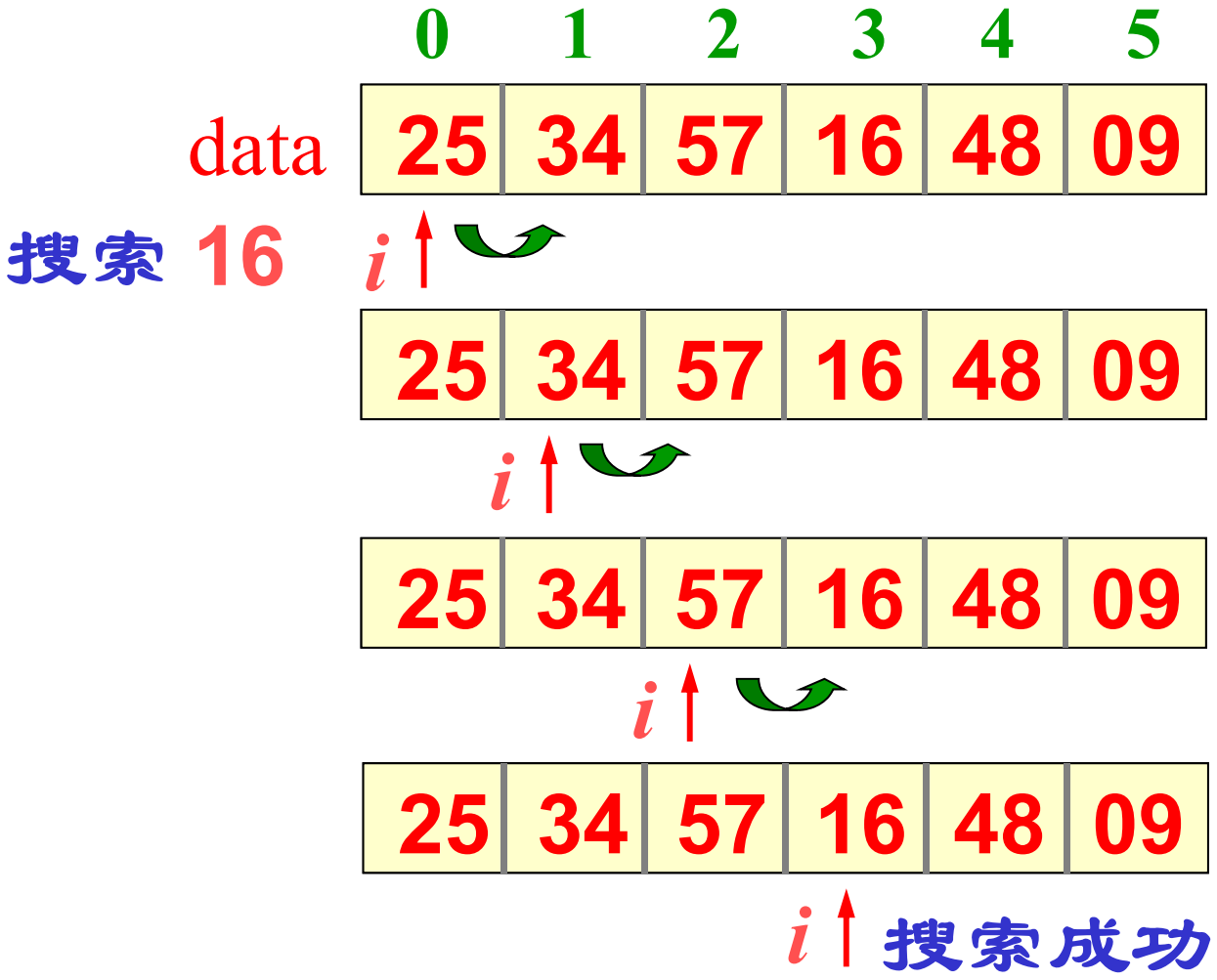
顺序表部分公共操作的实现

```
template <class Type>
SeqList <Type> :: SeqList ( int sz ) { //构造函数
    if ( sz > 0 ) {
        maxSize = sz;  last = -1;
        data = new Type [maxSize];
        if ( data == NULL ) {
            cerr << "存储分配错误！ " << endl;
            exit (1);
        }
    }
}
```

```
template <class Type>
SeqList <Type> :: SeqList ( SeqList <Type> &L ) {
    MaxSize = L.Size( );  last = L.Length( )-1;
    data = new Type [maxSize];
    if ( data == NULL ) {
        cerr << "存储分配错误！ " << endl;
        exit (1);
    }
    for ( int i = 1; i <= last+1; i++ )
        data[i-1] = L.GetData(i);
}
```

```
template <class Type>
SeqList <Type> :: ReSize ( int newSize ) {
    if ( newSize <= 0 )
       { cerr << "无效的数组大小" << endl; return; }
    if ( newSize != maxSize ) {
        Type *NewArray = new Type [newSize];
        if ( NewArray = = NULL )
           { cerr << "存储分配错误" << endl; exit(1); }
        int n = last+1;
        Type *srcptr = data;
        Type *destptr = NewArray;
        while ( n-- ) *destptr++ = *srcptr++;
        delete [ ] data;
        data = newArray;  maxSize = newSize;
    }
}
```

顺序搜索图示


0 1 2 3 4 5
data
25 34 57 16 48 09
搜索 16
i
25 34 57 16 48 09
i
25 34 57 16 48 09
i
25 34 57 16 48 09
i
搜索成功


0 1 2 3 4
data
25 34 57 16 48
搜索 50
i
25 34 57 16 48
i
25 34 57 16 48
i
25 34 57 16 48
i
25 34 57 16 48
i
搜索失败

```
template <class Type>
int SeqList <Type> :: Search ( Type &x ) const {
//搜索函数：在表中顺序搜索与给定值 x 匹配的表项
//找到则函数返回该表项是第几个元素
//否则函数返回 0，表示搜索失败
    for ( int i = 0; i <= last; i++ )
        if ( data[i] == x ) return i+1;
    return 0;
}
```

搜索成功的平均比较次数

$$\mathrm{ACN} = \sum_{i=0}^{n-1} p_i \times c_i$$

若搜索概率相等，则

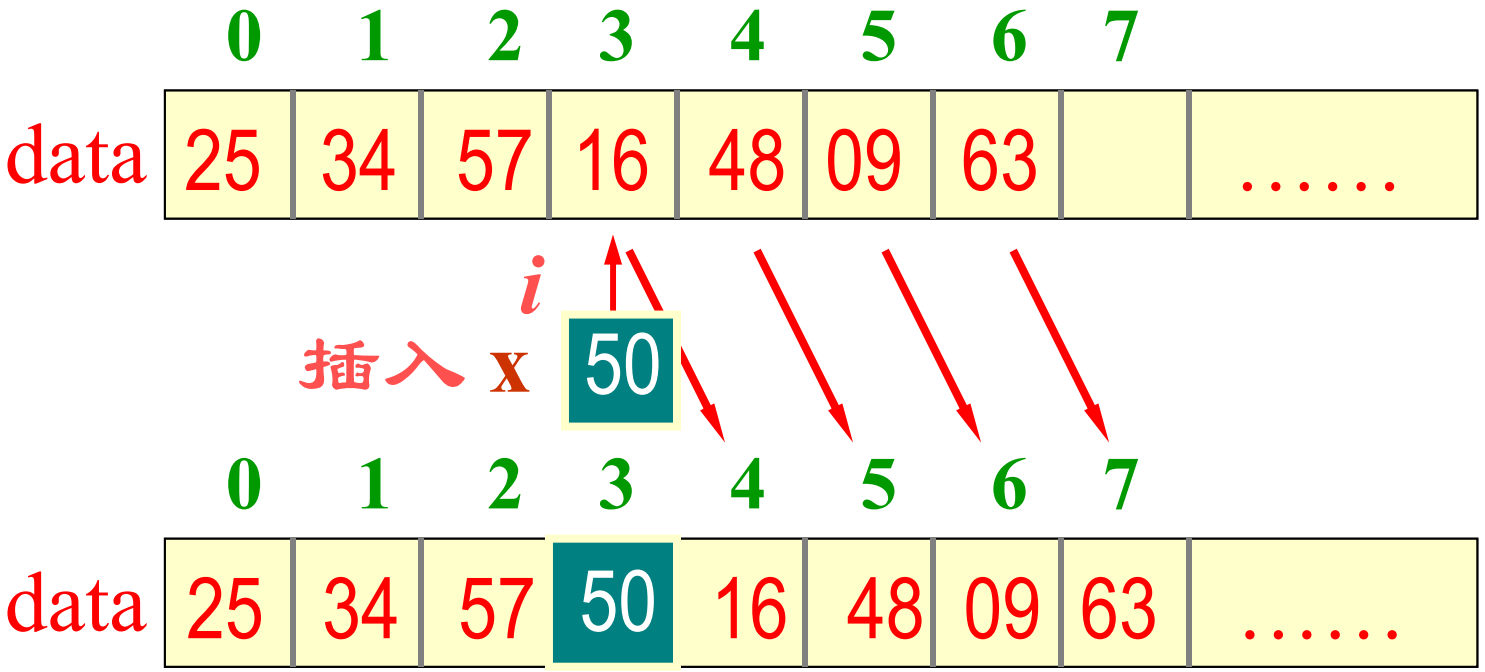
$$\mathrm{ACN} = \frac{1}{n}\sum_{i=0}^{n-1}(i+1) = \frac{1}{n}(1+2+\cdots+n) =$$

$$= \frac{1}{n} * \frac{(1+n)*n}{2} = \frac{1+n}{2}$$

搜索不成功　数据比较 n 次。

```
template <class Type>
int SeqList <Type> :: Locate ( int i ) const {
//定位函数：函数返回第 i 个表项的位置
//否则函数返回 0，表示定位失败
    if ( i >= 1 && i <= last+1 ) return i;
    else return 0;
}
```

表项的插入

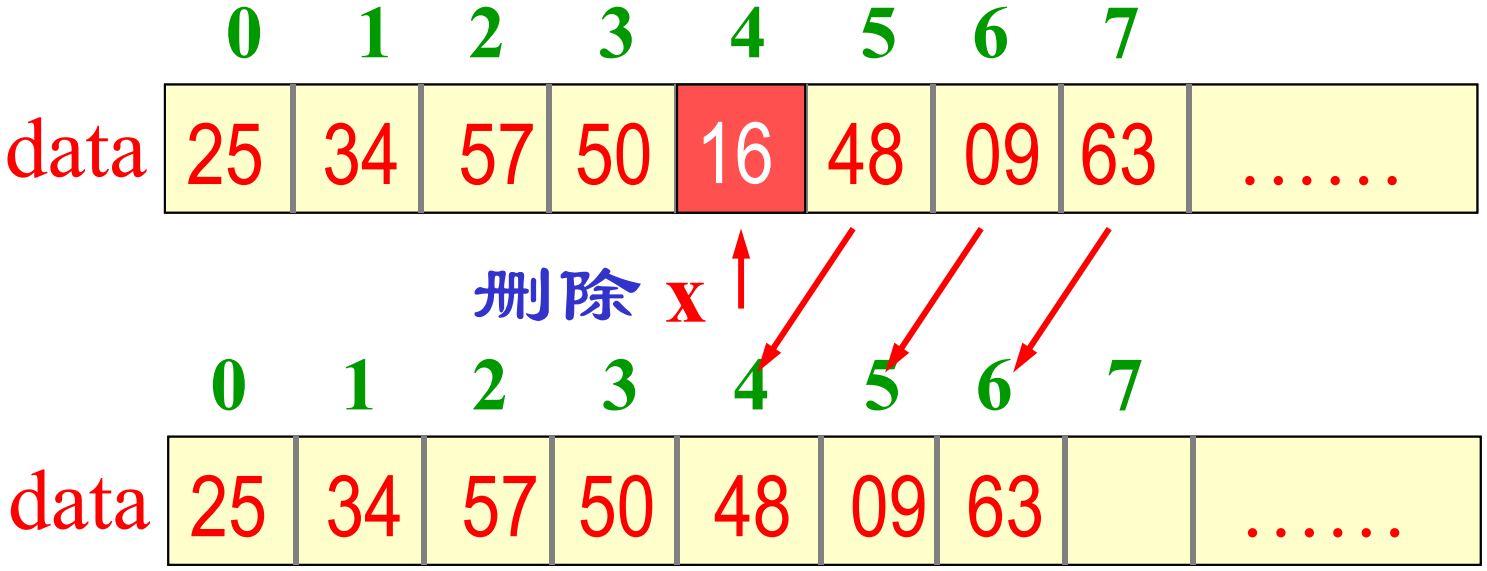


$$\text{AMN} = \frac{1}{n+1}\sum_{i=0}^{n}(n-i) = \frac{1}{n+1}(n+\cdots+1+0)$$

$$= \frac{1}{(n+1)}\frac{n(n+1)}{2} = \frac{n}{2}$$

```
template <class Type>
bool SeqList <Type> :: Insert ( int i, Type &x ) {
//在表中第 i 个表项之后插入新元素 x
    if ( i < 0 || i > last+1 || last == maxSize-1 )
        return false;  //插入不成功
    else {
        for ( int j = last; j > i; j-- )
            data[j+1] = data[j];
        data[i] = x;
        last++;
        return true;  //插入成功
    }
}
```

表项的删除



$$\mathrm{AMN} = \frac{1}{n}\sum_{i=0}^{n-1}(n-i-1) = \frac{1}{n}\frac{(n-1)n}{2} = \frac{n-1}{2}$$

```
template <class Type>
int SeqList <Type> :: Remove ( Type &x, int i ) {
//从表中删除第 i 个表项
//通过引用型参数 x 返回删除的元素值
    if ( last == -1 ) return false;
    if ( i < 0 || i > last+1 ) return false;
    x = data[i-1];
    for ( int j = i; j <= last; j++ )
            data[j-1] = data[j];
    last--;
    return true;
}
```

顺序表的应用：集合的“并”运算

```
void Union ( SeqList <int> &LA,
                    SeqList <int> &LB ) {
    int n = LA.Length ( );
    int m = LB.Length ( );
    for ( int i = 0; i < m; i++ ) {
        int x = LB.GetData (i);
        int k = LA.Search (x);
        if ( k == 0 )
          { LA.Insert ( n, x);  n++; }
    }
}
```

顺序表的应用：集合的“交”运算

```
void Intersection ( SeqList <int> &LA,
                    SeqList <int> &LB ) {
    int n = LA.Length ( );
    int m = LB.Length ( );
    int i = 0;
    while ( i < n ) {
        int x = LA.GetData (i);
        int k = LB.Search (x);
        if ( k == 0 ) { LA.Remove (i, x);  n--; }
        else i++;
    }
}
```

随堂练习

例1：用线性表的顺序存储结构来描述一个城市的设计和规划是否合适？为什么？

例2：在包含n个元素的顺序表中删除一个元素，需要平均移动_____个元素，其中具体移动的元素个数与_____有关。

例3：线性表的顺序存储结构具有三个弱点：其一，在作插入或删除操作时，需移动大量元素；其二，由于难以估计，必须预先分配较大的空间，往往使存储空间不能得到充分利用；其三，表的容量难以扩充。线性表的链式存储结构是否一定都能够克服上述三个弱点。试讨论之。

例1：用线性表的顺序存储结构来描述一个城市的设计和规划是否合适？为什么？

不合适。因为一个城市的设计和规划涉及非常多的项目，比较复杂，需要经常改动、扩充和删除各种信息，这样才适应不断发展的需要，所以顺序表不能很好地适应其需要。

例2：在包含n个元素的顺序表中删除一个元素，需要平均移动(n-1)/2 个元素，其中具体移动的元素个数与具体删除位置有关。

例3：线性表的顺序存储结构具有三个弱点：其一，在作插入或删除操作时，需移动大量元素；其二，由于难以估计，必须预先分配较大的空间，往往使存储空间不能得到充分利用；其三，表的容量难以扩充。线性表的链式存储结构是否一定都能够克服上述三个弱点。试讨论之。

不一定。由于链式存储需要额外的空间来存储指针，所以要比顺序存储多占用空间。在空间允许的情况下，链式存储结构可以克服顺序存储结构的弱点，但空间不允许时，链式存储结构会出现新问题。

为什么学习链表来表示线性表?

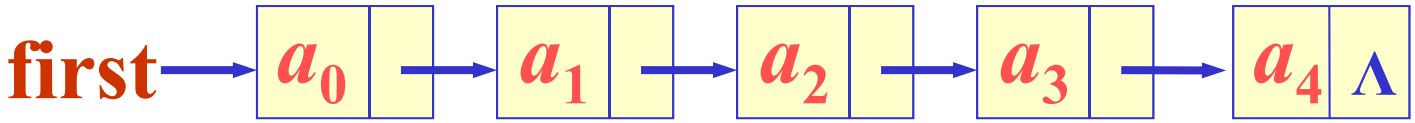
- 用**数组**或**顺序映射**表示线性表，相邻元素存储地址的间距是常数。因此，可用O(1)的时间访问任何元素。但是，删除或往其中插入元素将导致表中其它元素的移动，代价太高O(n)。
- **链表表示**可以有效地解决上述数据移动问题。在链表表示中，表的数据元素可存放在存储器中的任何位置，**每一个元素都与其下一个元素的地址（指针）相关联。**

- 利用数组或顺序方式来组织数据结构
  - 优点
    - 存储利用率高，存取速度快。
  - 缺点
    - 数组元素个数不能自由扩充（除动态数组外）
    - 效率很低
    - 造成空间很大浪费
- 链表(*Linked List*)
  - 适用于插入删除频繁、存储空间需求不定的情形。
  - 不但可表示线性聚集，还可表示非线性聚集。

2.3 单链表 (Singly Linked List)

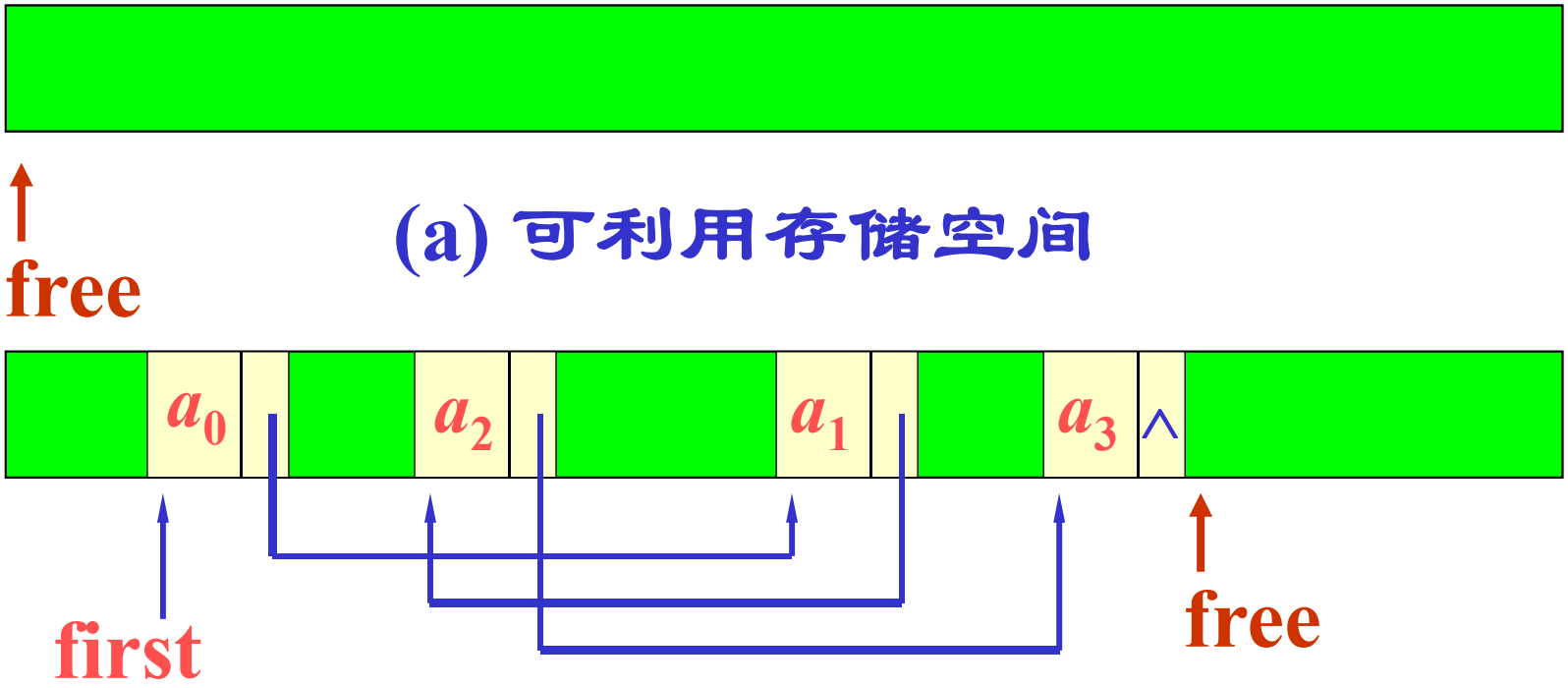
- **特点**
  - 每个元素（表项）由结点(*Node*)构成。

  - 线性结构



  - 结点之间可以连续，可以不连续存储;
  - 结点的逻辑顺序与物理顺序可以不一致;
  - 表可扩充。

单链表的存储映像



单链表类定义

- 多个类表达一个概念（单链表）
  - 链表结点(LinkNode)类
  - 链表(List)类
- 定义方式
  - 复合方式
  - 嵌套方式
  - 继承方式
  - 结构方式

```
class List; //链表类定义（复合方式）
class LinkNode { //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
   LinkNode *link; //结点指针
};
class List { //链表类
 private:
   LinkNode *first , *current; //表头指针
 public:
   ..... //链表操作
};
```

```
class List {  //链表类定义（嵌套方式）
 private:
    class LinkNode {  //嵌套链表结点类
    public:
        int data;
        LinkNode *link;
    };
    LinkNode *first , *current;  //表头指针
 public:
    ……  //链表操作
};
```

```
class LinkNode {  //链表结点类
 protected:
    int data;
    LinkNode *link;
};

class List : public LinkNode
{  //链表类定义（继承方式）
    //链表类，  继承链表结点类的数据和操作
  private:
     LinkNode *first , *current;  //表头指针
  public:
     ..... //链表操作
};
```

```
struct LinkNode {  //用结构定义链表结点类
    int data;
    LinkNode *link;
};

class List
{
 private:
     LinkNode *first , *current;  //表头指针
 public:
     ..... //链表操作
};
```

- 在**复合方式**中，链表结点类中声明链表类是它的友元类，这样可以“奉献”它的私有成员给链表类，这种方式灵活。

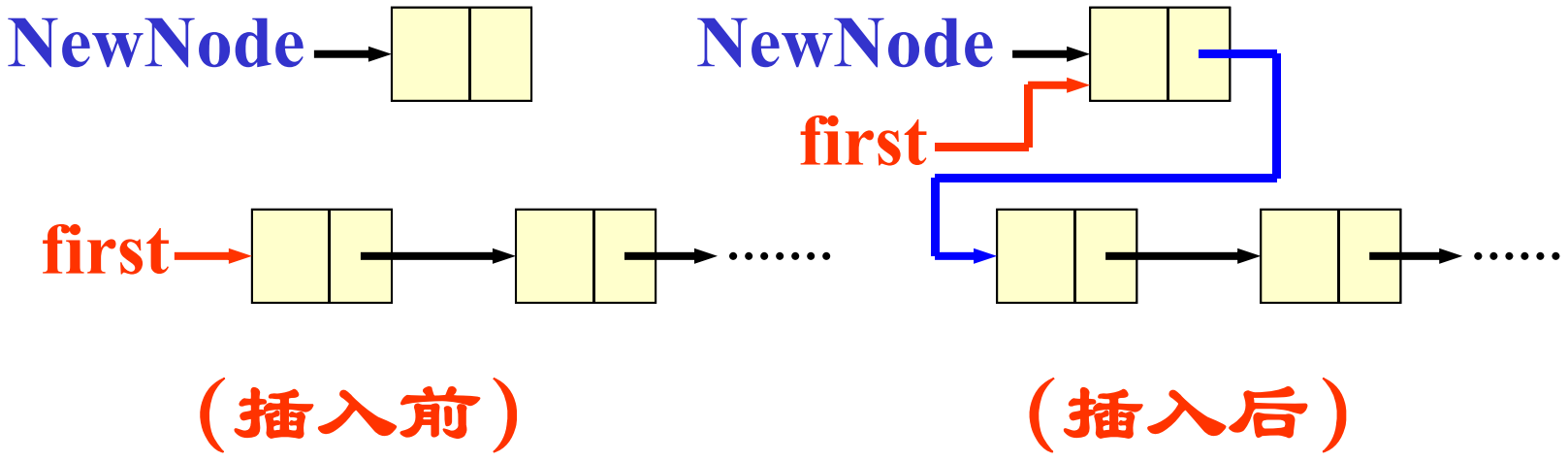
- 在**嵌套方式**中，链表结点类是链表类的私有成员，这样限制了链表结点类的应用范围。

- 在**继承方式**中，链表类声明为链表结点类的派生类，这在实现上是可行的，但在逻辑上是有问题的。如
  - ✓三角形 is 多边形（继承关系）
  - × 链表 is 链表结点（显然概念有误）

单链表中的插入与删除

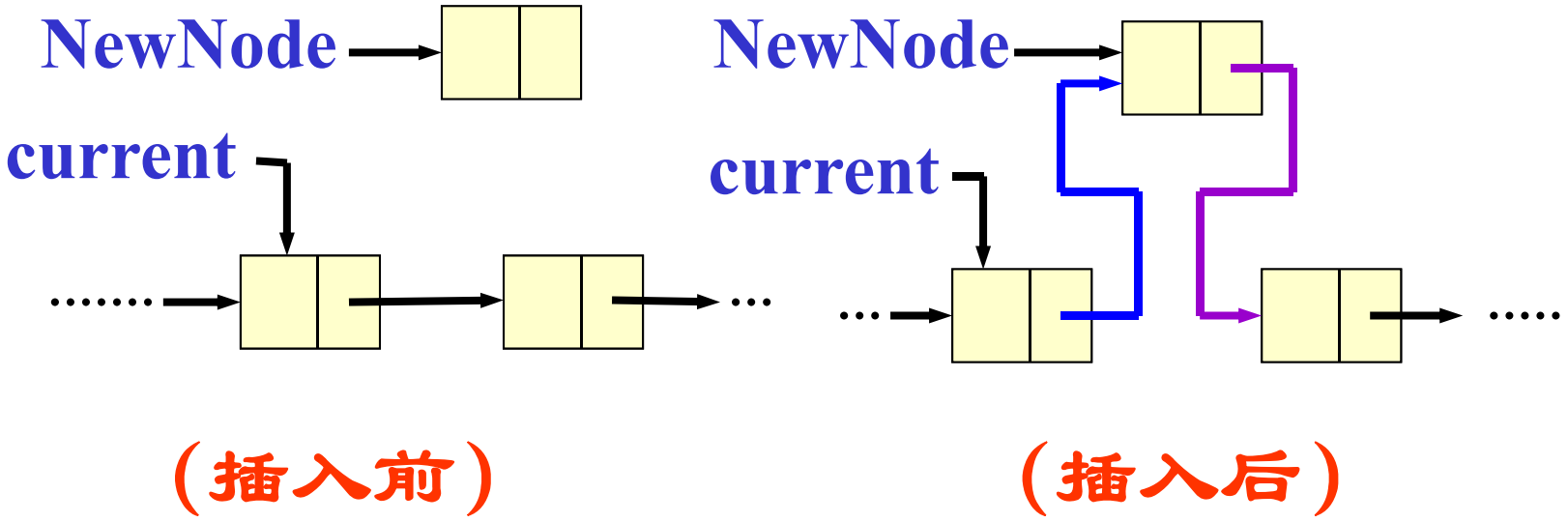
- 插入
  - 第一种情况：在链表最前端插入

```
NewNode->link = first;
first = NewNode;
```



第二种情况：在链表中间插入

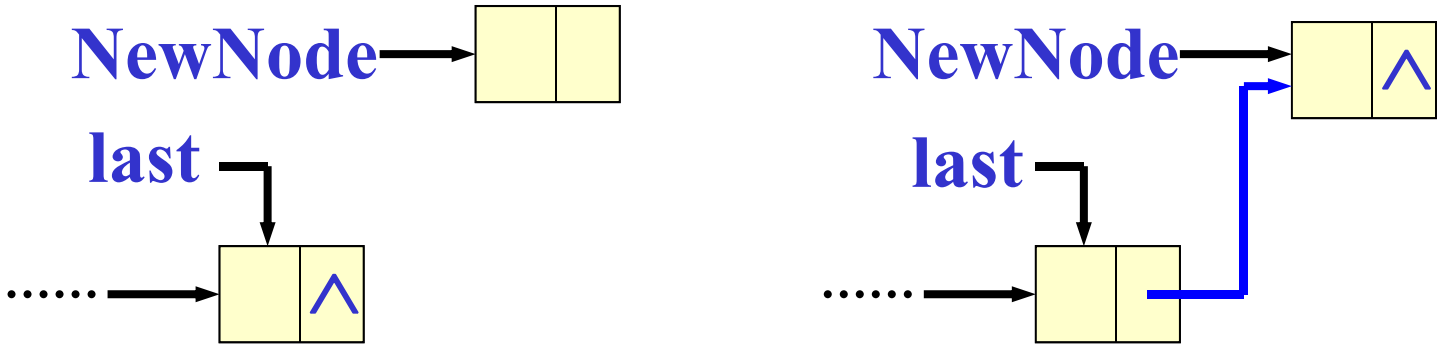
```
NewNode->link = current->link;
current->link = NewNode;
```



（插入前）　　（插入后）

- **第三种情况：在链表末尾插入**

```
NewNode->link = last->link;
last->link = NewNode;
```



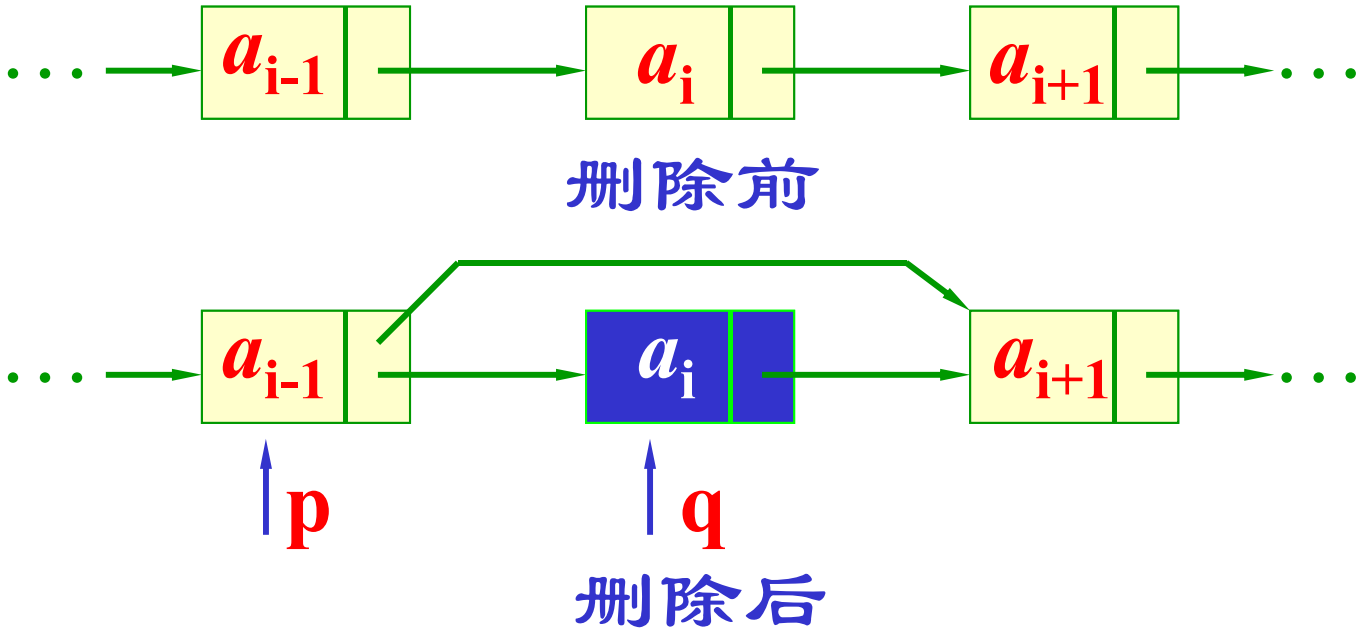
（插入前）　　（插入后）

last (≡ current)

```
bool List :: Insert ( int i, int &x ) {
//将新元素 x 插入到第 i 个结点之后
// i 从 1 开始， i = 0 表示插入到第一个结点之前
   if ( first == NULL || i == 0 ) {  //插在表前
        LinkNode *NewNode = new LinkNode (x);
        if  ( NewNode == NULL )
              { cerr << "存储分配错误！ \n" ;  exit(1); }
        NewNode->link = first;
        first = NewNode;    }
    else {
        LinkNode *current = first;
        for ( k = 1; k < i; k++ )
              if ( current == NULL ) break;
              else current = current->link;
```

```
            if ( current == NULL && first != NULL )
              { cout << "无效的插入位置！ \n" ; return false; }
            else {
                  LinkNode *NewNode = new LinkNode(x);
                  if ( NewNode == NULL )
                    { cerr << "存储分配错误！ \n" ;  return false; }
                  NewNode->link = current->link;
                  current = current->link = NewNode;
            }
      }
      return true;
}
```

- **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**

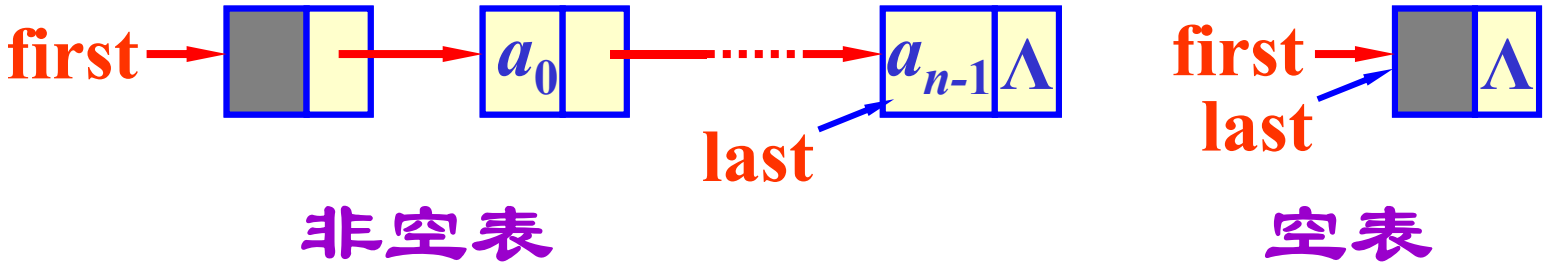


在单链表中删除含a_i的结点

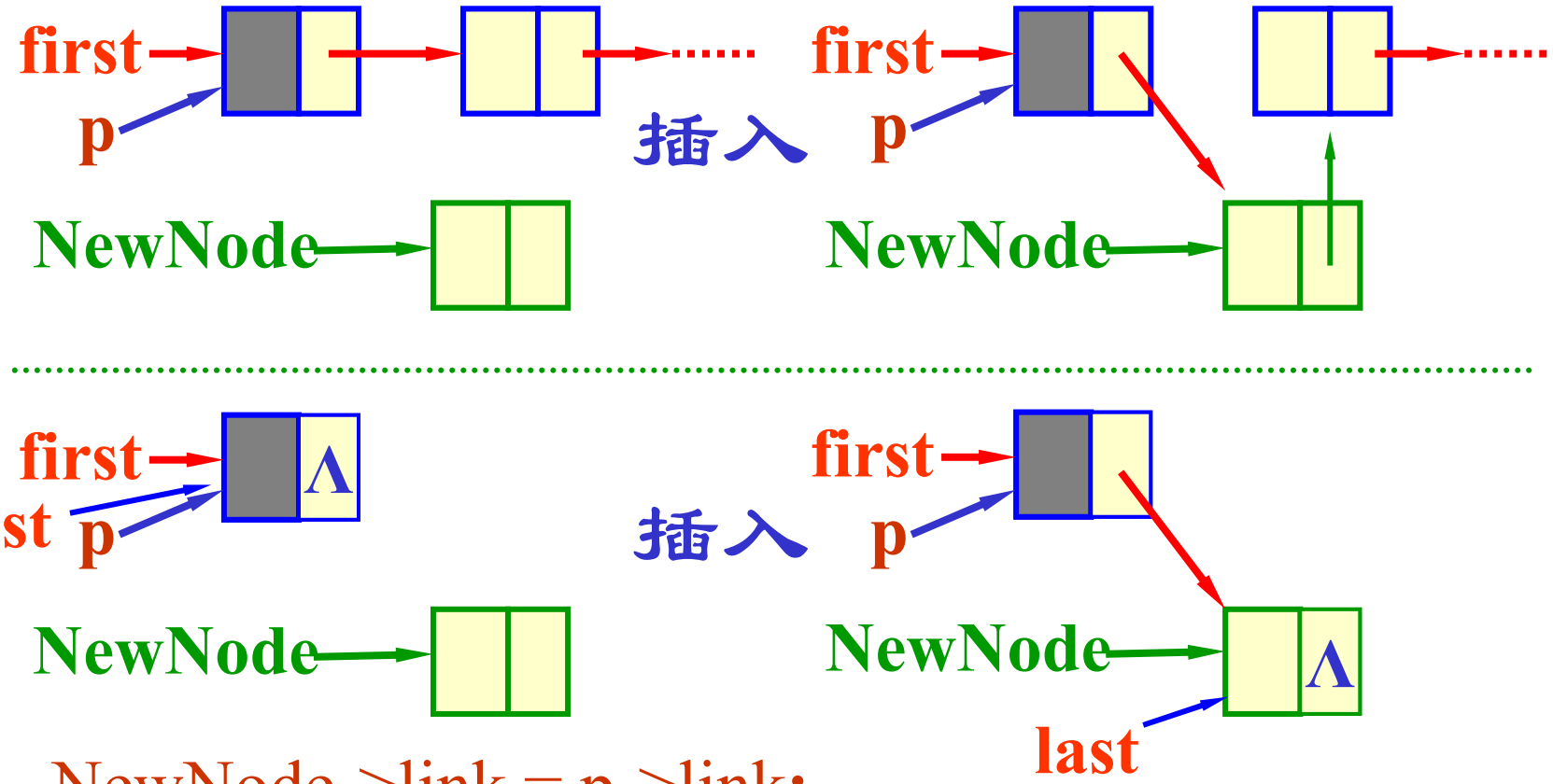
```
bool List :: Remove ( int i, int &x ) {
//在链表中删除第 i 个结点，i 从 1 开始
   LinkNode *del, *current;
   if ( i <= 1 ) { del = first; first = first->link; }
   else {
        current = first;
        for ( int k=0; k<i-1; k++ )
            if ( current == NULL ) break;
            else current = current->link;
        if ( current == NULL || current->link == NULL )
          { cout << "无效的删除位置！ \n" ; return false; }
        del = current->link;  current->link = del->link;
    }
    x = del->data;  delete del;  return true;
}
```

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。

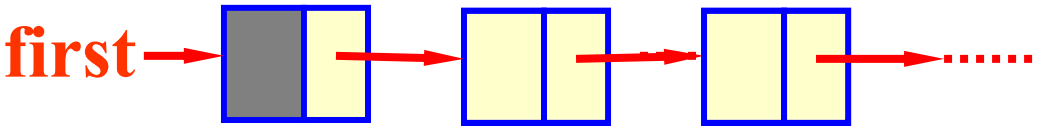


在带表头结点的单链表最前端插入新结点

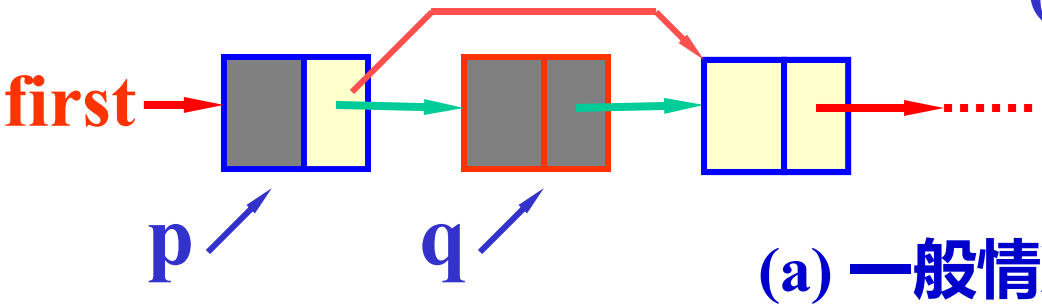



```
NewNode->link = p->link;
if ( p->link == NULL ) last = NewNode;
p->link = NewNode;
```

从带表头结点的单链表中删除最前端的结点

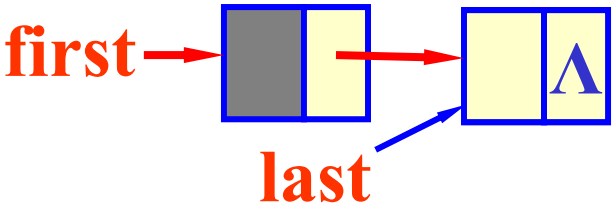


(非空表)



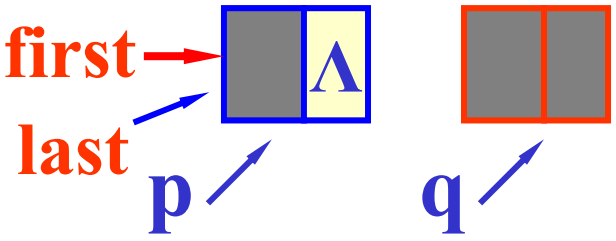
(a) 一般情况:

```
q = p->link;
p->link = q->link;
delete q;
```



(b) 有可能删除后链表变为空表:

```
if ( p->link == NULL ) last = p;
```



(空表)

单链表的模板类

- 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 LinkNode {
friend class List <Type>;
private:
    Type data;  //结点数据
    LinkNode <Type> *link;  //结点链接指针
public:
    LinkNode ( ) : link (NULL) { }  //构造函数
    LinkNode ( Type item ) : data (item), link (NULL) { }
```

```
    LinkNode <Type> * NextNode ( ) { return link; }
    //取得结点的下一结点地址
    Type InsertAfter ( ListNode <Type> *p );
    //当前结点插入
    LinkNode <Type> * GetNode ( const Type &item,
        LinkNode <Type> *next );
    //以 item 和 next 建立一个新结点
    LinkNode <Type> * RemoveAfter ( );
    //删除当前结点的下一结点
};
```

```
template <class Type> class List {
//链表类
private:
   LinkNode <Type> *first, *last, *current;
   //链表的表头指针、尾指针和当前元素指针
public:
   List ( ) { first = new LinkNode <Type>; }
   List ( const Type &value )
      { last = first = new LinkNode <Type> ( value ); }
   List ( List <Type> &L );
   ~List ( ) { MakeEmpty ( );  delete first; }
   void MakeEmpty ( );
   int Length ( ) const;
   LinkNode <Type> * GetHead ( ) const { return first; }
   void SetHead ( LinkNode <Type> *p ) { first = p; }
```

```
    LinkNode <Type> * Search ( Type x );
    LinkNode <Type> * Locate ( int i );
    Type * GetData ( int i );
     void SetData ( int i, Type &x );
    bool Insert ( int i, Type &x);
    bool Remove ( int i, Type &x );
    bool IsEmpty ( ) const
          { return first->link == NULL ? true : false; }
    bool IsFull ( ) const  { return false; }
    void Sort ( );
    void Input ( );
    void Output ( );
    List <Type> operator = ( List <Type> &L );
};
```

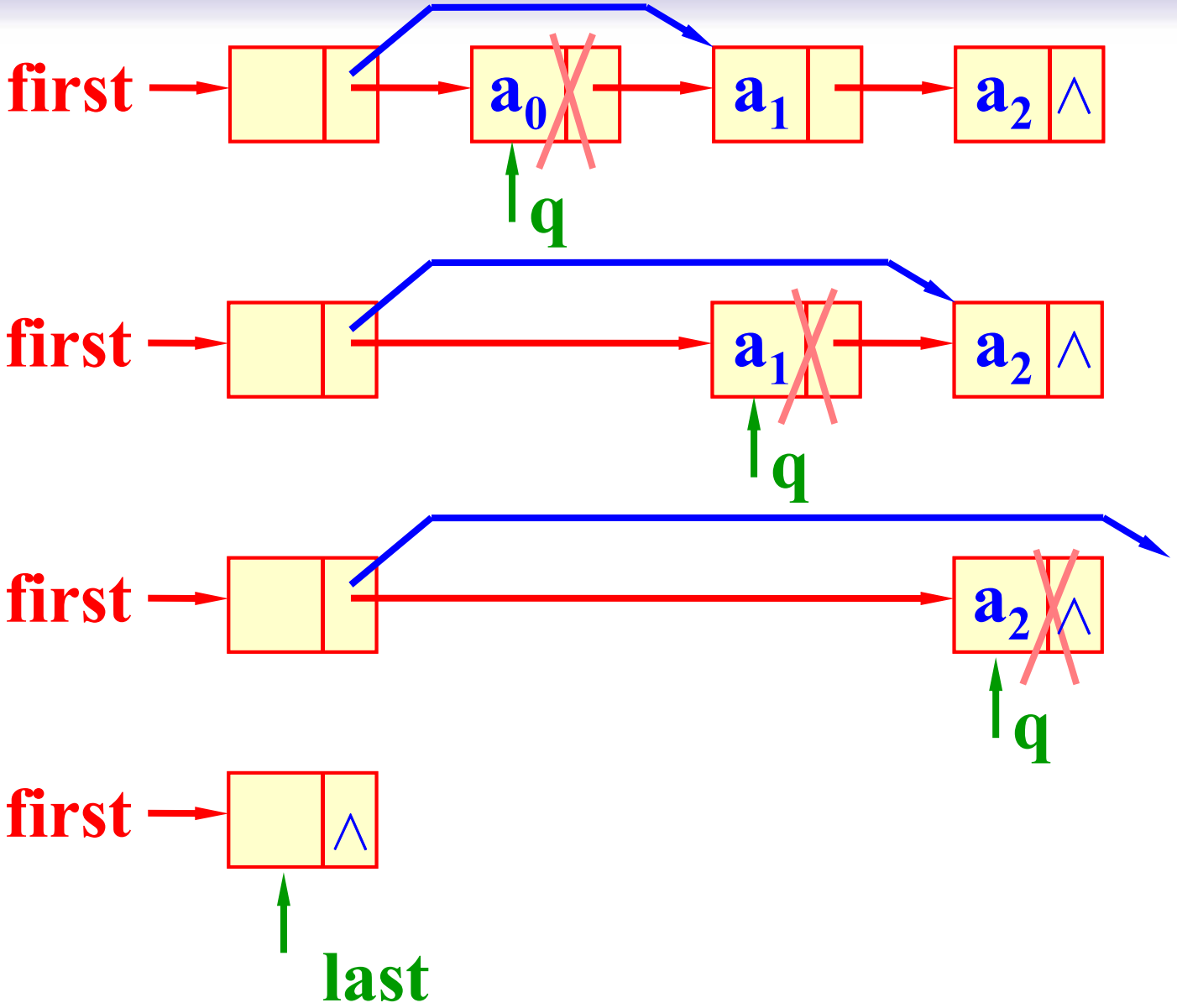
链表类部分操作的实现

```
template <class Type> void LinkNode <Type> ::
    InsertAfter ( LinkNode <Type> *p ) {
//将 p 所指示结点链接成为当前结点(*this)的后继结点
    p->link = link;  link = p;
}

template <class Type> LinkNode <Type>
        * LinkNode <Type> :: GetNode ( const Type &item,
                LinkNode <Type> *next = NULL) {
//以 item 和 next 为参数，建立新结点并返回新结点地址
    LinkNode <Type> *NewNode =
                        new LinkNode <Type> (item);
    NewNode->link = next;  return NewNode;
}
```

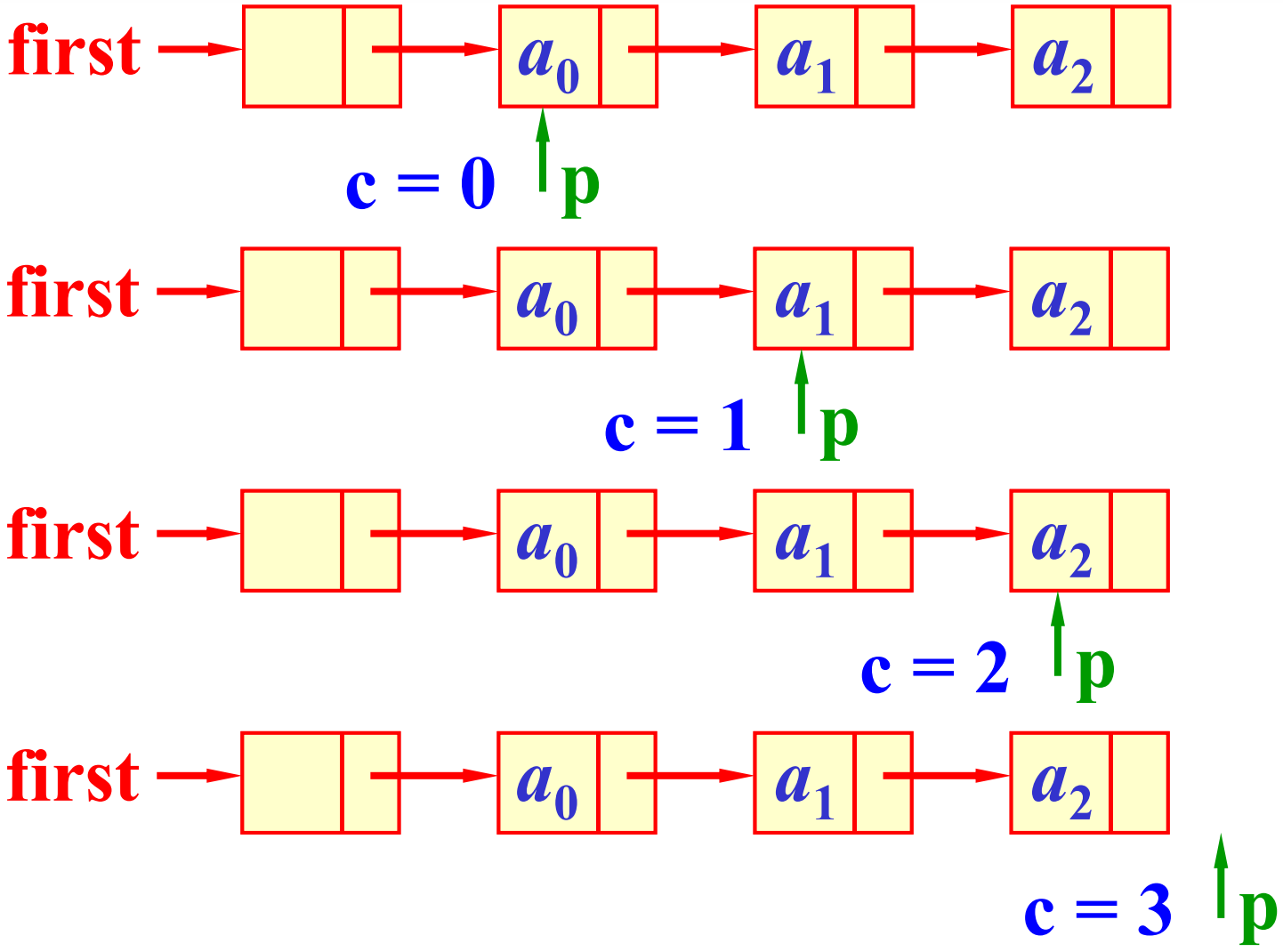
```
template <class Type> * LinkNode <Type> ::
        RemoveAfter ( ) {
//从链中摘下当前结点的下一结点删除
//并返回其地址
   LinkNode <Type> *tempptr = link;
   //保存被删除结点的地址
   if ( link == NULL ) return NULL;
   //当结点无后继，返回NULL
   link = tempptr->link; //将被删结点从链中摘下
   return tempptr;
}
```

```
template <class Type>
        List <Type> :: List ( List <Type> &L ) {
    Type value;
    LinkNode <Type> *srcptr = L.GetHead ( );
    LinkNode <Type> *destptr = first
                                = new LinkNode <Type>;
    while ( srcptr->Link != NULL ) {
        value = srcptr->link->data;
        destptr->link = new LinkNode <Type> ( value );
        destptr = destptr->link;
        srcptr = srcptr->link;
    }
    destptr->link = NULL;
}
```

```
template <class Type>
        void List <Type> :: MakeEmpty ( ) {
//删去链表中除表头结点外的所有其它结点
//将链表置为空表
    LinkNode <Type> *q;
    while ( first->link != NULL ) {
    //当链不空时，删去链中所有结点
        q = first->link;  first->link = q->link;
        delete q;  //循链逐个删除，保留一个表头结点
    }
    last = first;  //表尾指针指向表头结点
}
```


first
a_0
a_1
a_2 ∧
q
first
a_1
a_2 ∧
q
first
a_2 ∧
q
first
∧
last

```
template <class Type>
        int List <Type> :: Length ( ) const {
//求单链表的长度
    LinkNode <Type> *p = first->link;
    //检测指针 p 指示第一个数据结点
    int count = 0;
    while ( p != NULL ) {  //循链扫描，寻找链尾
        p = p->link;  count++;
    }
    return count;
}
```


first
a_0
a_1
a_2
c = 0
p
first
a_0
a_1
a_2
c = 1
p
first
a_0
a_1
a_2
c = 2
p
first
a_0
a_1
a_2
c = 3
p

```
template <class Type> LinkNode <Type>
            * List <Type> :: Search ( Type x ) {
//在链表中从头搜索其数据值为 x 的结点
//搜索成功时函数返回该结点地址，否则返回NULL
    LinkNode <Type> *p = first->link;
    while ( p != NULL && p->data != x )
          p = p->link;
    //循链找含 x 结点
    return p;
}
```

```
template <class Type> LinkNode <Type>
        * List <Type> :: Locate ( int i ) {
//定位函数：返回表中第 i 个元素的地址
//若 i<0 或 i 超出表中结点个数，则返回 NULL
   if ( i < -1 ) return NULL; // i 值不合理
   LinkNode <Type> *p = first;  int j = 0;
   //检测指针 p 指向表中第一个结点
   while ( p != NULL && j < i )
   { p = p->link;  j++; }  //寻找第 i 个结点的地址
   return p;
   //返回第 i 个结点地址，若返回 NULL，表示 i 值太大
}
```

```
template <class Type>
        bool List <Type> :: Insert ( int i, Type &x ) {
//将新元素 x 插入在链表中第 i 个位置
    LinkNode <Type> *p = Locate ( i );
    //定位第 i 个元素
    if ( p == NULL ) return false;
    //参数 i 的值不合理，函数返回 false
    ListNode <Type> *NewNode = p->GetNode ( x, p->link );
    //创建含 x 的结点
    if ( p->link == NULL ) last = NewNode;
    p->link = NewNode;
    return true; //插入成功，函数返回 true
}
```

```
template <class Type>
        bool  List <Type> :: Remove ( int i, Type &x ) {
//将链表中的第 i 个元素删去
    LinkNode <Type> *p = Locate ( i-1 ), *q;
    // p 定位于第 i-1 个元素
    if ( p == NULL || p->link == NULL ) return false;
    // i 的值不合理或空表，返回 flase
    q = p->link;  p->link = q->link;
    // q 指向被删结点，重新拉链
    x = q->data;
    //取出被删结点中的数据值
    if ( q == last ) last = p;  //删除尾结点时，表尾指针修改
    delete q;
    return true;
}
```

```
template <class Type> List <Type> & List <Type> ::
operator = ( List <Type> &L ) {
    Type value;
    LinkNode <Type> *srcptr = L.GetHead ( );
    LinkNode <Type> *destptr = first
                                = new LinkNode <Type>;
    while ( srcptr->Link != NULL ) {
        value = srcptr->link->data;
        destptr->link = new LinkNode <Type> ( value );
        destptr = destptr->link;
        srcptr = srcptr->link;
    }
    destptr->link = NULL;
    return *this;
}
```

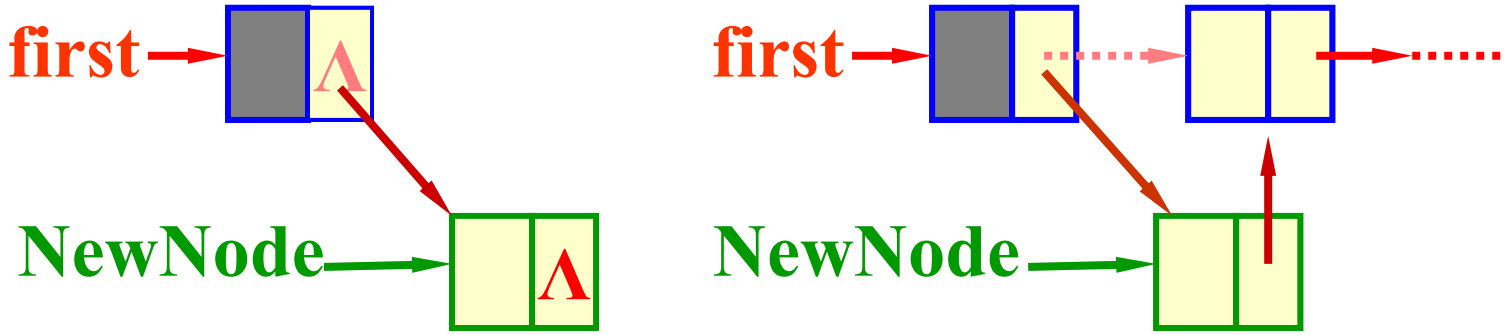
创建power类，计算x的幂

```
#include <iostream.h>
class Power {
    double x;
    int e;
    double mul;
public:
    Power ( double val, int exp );  //构造函数
    double Get_Power ( ) { return mul; }  //取幂值
};
```

```
Power :: Power ( double val, int exp ) {
//按 val 值计算乘幂
    x = val;  e = exp;  mul = 1.0;
    if ( exp == 0 ) return;
    for ( ; exp > 0; exp-- ) mul = mul * x;
}

main ( ) {
    Power pwr ( 1.5, 2 );
    cout << pwr.Get_Power ( ) << "\n";
}
```

前插法建立单链表

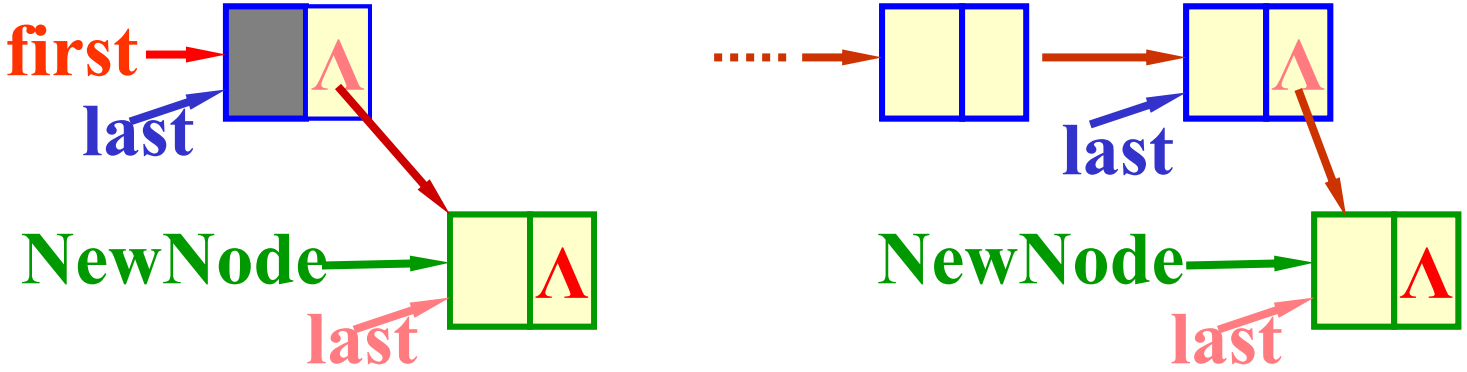
- 从一个空表开始，重复读入数据：
  - 生成新结点；
  - 将读入数据存放到新结点的数据域中；
  - 将该新结点插入到链表的前端。
- 直到读入结束符为止。



```
template <class Type>
void List <Type> :: CreateListF ( Type endTag )
{
    Type val;  LinkNode <Type> *NewNode;
    first = new LinkNode <Type>;  //表头结点
    cin >> val;
    while ( val != endTag ) {
        NewNode = new LinkNode <Type> (val);
        NewNode->link = first->link;
        first->link = NewNode;  //插入到表前端
        cin >> val;
    }
    current = first;
}
```

后插法建立单链表

- 每次将新结点插到链表的表尾；
- 设置一个尾指针 last，总是指向表中最后一个结点，新结点插在它的后面；
- 尾指针 last 初始时置为指向表头结点地址。



```
template <class Type>
void List <Type> :: CreateListR ( Type endTag ) {
    Type val;  LinkNode <Type> *NewNode, *last;
    first = new LinkNode <Type>;  //表头结点
    cin >> val;  current = last = first;
    while ( val != endTag ) {  // last 指向表尾
        NewNode = new LinkNode <Type> ( val );
        last->link = NewNode;  last = NewNode;
        cin >> val;  //插入到表末端
    }
    last->link = NULL;  //表收尾
}
```

- 单链表用于表示一个顺序表时，经常需要执行的操作包括：
  - 打印一个单链表中所有数据的值；
  - 当链表中所有的数据为整数、浮点数或双精度数时，计算所有数据的累加和、乘积，求它们最大值、最小值、中值、平均值、方差；
  - 检索链表中所有满足某种条件的数据，以及对它们进行排序等。
- 为实现上述操作，都需要前后搜索整个链表。
  - 如果将所有操作都加入到链表类中，会使类的成员函数集变得很大。
  - 对于将来出现的大量新的需求，必须不断地修改类的定义，不利于建立类库以及类的复用。

链表的游标类 (Iterator)

链表游标(Iterator)是一种用于遍历链表的全部元素的对象。从 first 出发，很容易遍历链表的全部元素，为什么还需要游标呢?

- 设链表L的元素都是整型数，考虑下列操作：
  - 打印L中的所有元素。
  - 计算L中所有元素的最大值、最小值和平均值。
  - 计算L中所有元素的和与积。
  - 求L中满足谓词P(x)的元素（例如，P(x)为：x是某个整数的平方）。
  - 求L中的元素x，使得对于某个函数f，f(x)达到最大值。

- 这启发我们将遍历(Traversal)移到List <Type>定义之外。
- 进一步观察，以上操作都不改动链表内容，但需要访问List <Type>和LinkNode <Type>的私有数据成员。为此，我们定义第三个类ListIterator <Type>。
- ListIterator <Type>处理遍历链表的细节并对外提供链表中元素的值。

- 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

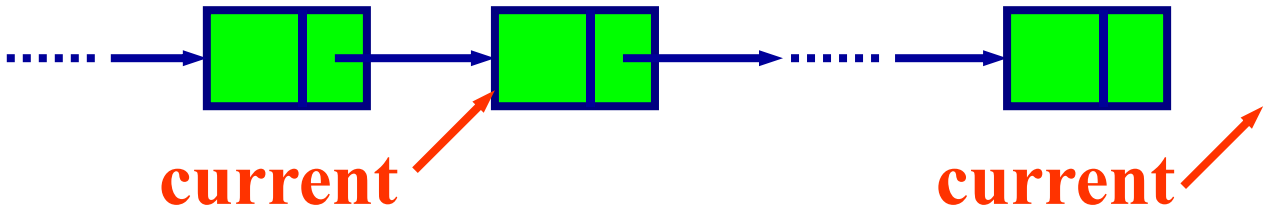
```
template <class Type> class List;
template <class Type> class ListIterator;
template <class Type> class LinkNode {  //表结点
friend class List <Type>;
friend class ListIterator <Type>;
public:
      ……
private:
   Type data;
   LinkNode <Type> *link;
};
```

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

template <class Type> class ListIterator {
private:
    const List <Type> &list;  //引用已有链表
    LinkNode <Type> *current;  //当前结点指针
```

```
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
    LinkNode <Type> * Firster ( )
      { current = first;  return first; }
    //当前指针置于表头，返回表头结点地址
    bool NotNull ( );  //检查当前指针空否
    bool NextNotNull ( );
    //检查链表中下一结点是否非空
    LinkNode <Type> *First ( );  //返回第一个结点
    LinkNode <Type> *Next ( );
    //返回链表当前结点的下一个结点的地址
};
```

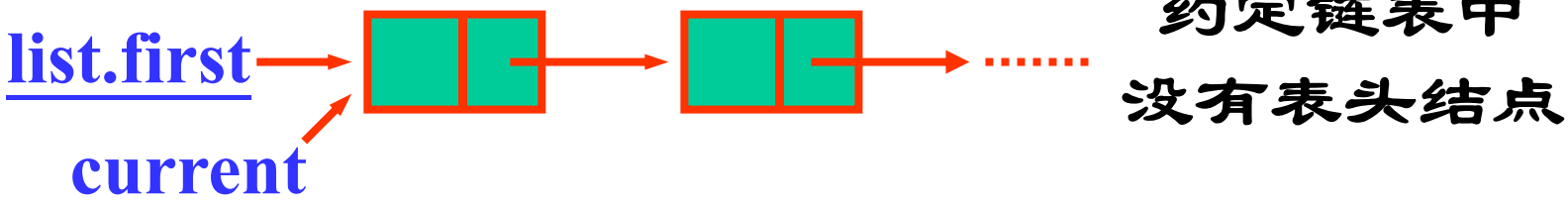
链表的游标类成员函数的实现

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

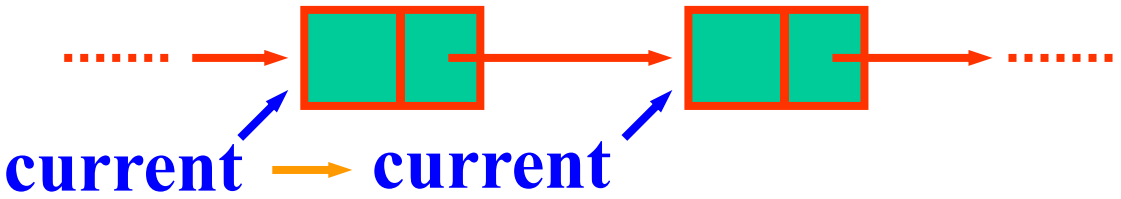


情况 1 返回true　　情况 2 返回false

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



```
template <class Type>
LinkNode <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( );  //第一个元素
    while ( li.nextNotNull ( ) )  //链表未扫描完
        retval += *li.Next( );  //累加
    return retval;
}
```

随堂练习

例1：(1) 对一个线性表分别进行遍历和逆置运算，求解其最好的渐进时间复杂度表示。

(2) 求解求顺序表和单链表长度的渐进时间复杂度表示。

(3) 若给定n个元素的向量，则求解建立一个有序单链表的渐进时间复杂度表示。

例2：设有头结点的单链表L，编程对表中任一值只保留一个结点，删除其余值相同的结点。

例3：有一带头结点的单链表，编程将链表颠倒过来，要求不用另外的数组或结点完成。

例1：(1) 对一个线性表分别进行遍历和逆置运算，求解其最好的渐进时间复杂度表示。

遍历时间的渐进时间复杂度表示为O(n)；进行逆置运算时，顺序表的时间要少于单链表时间，其量级也为O(n)。

(2) 求解求顺序表和单链表长度的渐进时间复杂度表示。

顺序表的长度可以直接从最后一个元素的位置获得，而单链表的长度则必须遍历整个表后获得，即O(1)与O(n)。

(3) 若给定n个元素的向量，则求解建立一个有序单链表的渐进时间复杂度表示。

单纯建立单链表的渐进时间复杂度为O(n)，而建立一个有序单链表还涉及查找插入到其正确位置，故渐进时间复杂度表示为$O(n^2)$。

例 2：设有头结点的单链表 L，编程对表中任一值只保留一个结点，删除其余值相同的结点。

(1) 首先用指针 p 指向链表中第一个数据结点，然后用指针 t 搜索整个链表以寻找值相同的结点直到链尾；在搜索中，指针 s 指向 t 所指结点前驱结点，当 t->data=p->data 时则删除 t 所指结点，即 s->link=t->link。

(2) 修改指针 p，使 p 指向链表的下一个结点（即 p=p->link），然后重复(1)的操作直至 p=NULL。

```
DelElem(lklist  *L)
{
    Pointer *p, *t, *pre;
    p=L->link;
    t=p;
    while (p!=NULL)
    {
        pre=t;
        t=t->link;
        do
        {
            while ((t!=NULL) && (t->data!=p->data))
            {
                pre=t;
                t=t->link;
            }
            if (t!=NULL)
            {
                pre->link=t->link;
                free(t);
                t=pre->link;
            }
        }while (t!=NULL);
        p=p->link;
        t=p;
    }
}
```

例 3：有一带头结点的单链表，编程将链表颠倒过来，要求不用另外的数组或结点完成。

在遍历原单链表每个结点的同时摘下该结点插入到新链表的表头。这样，先遍历到的结点先插入，后遍历到的结点后插入；由于插入是在表头进行，所以先插入的结点成为表尾，后插入的结点成为表头；也即实现了链表的逆置。

```
void reverse (lklist *h)
{
    lklist *p, *q;
    p=h->link;      /*p 指向原链表的第一个数据结点*/
    h->link=NULL;     /*新链表初始为空*/
    while (p)
    {
        q=p;       /*q 指向将摘下来插入到新链表的结点*/
        p=p->link;     /*p 指向下一个待摘下来的原链表结点*/
        q->link=h->link;          /*将*q 插入到新链表表头*/
        h->link=q;
    }
}
```

2.4 线性链表的其他变形

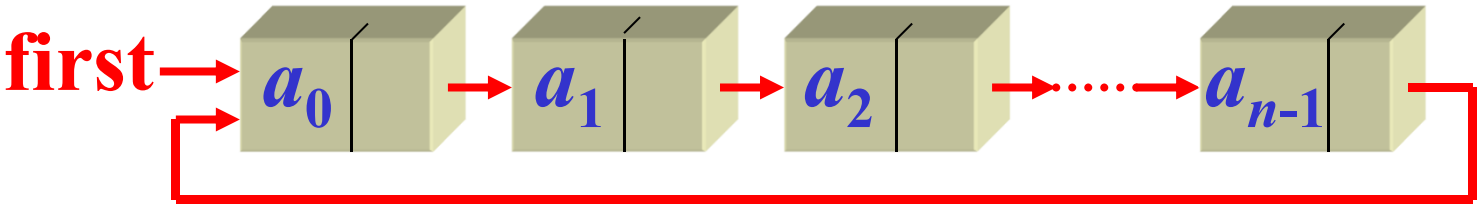
2.4.1 循环链表 (Circular List)

- 循环链表是单链表的变形。
- 循环链表的最后一个结点的 link 指针不为 NULL，而是指向表的前端。
- 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，就可搜寻到所有其它结点的地址。
- 检查指针 current 是否到达链表的链尾时：

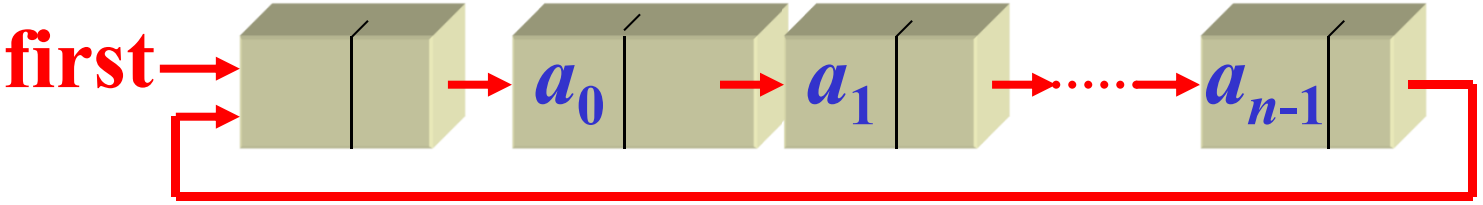
  **不是判断是否current->link == NULL；

  **而是判断是否current->link == first。

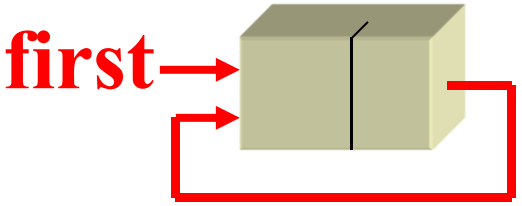
- **循环链表的示例**



- **带表头结点的循环链表**



（非空表）



（空表）

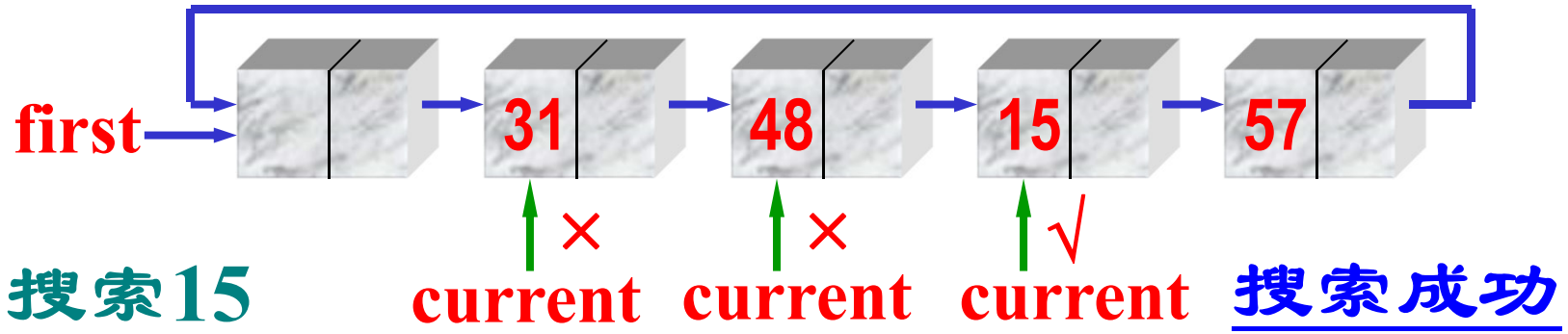
循环链表类定义

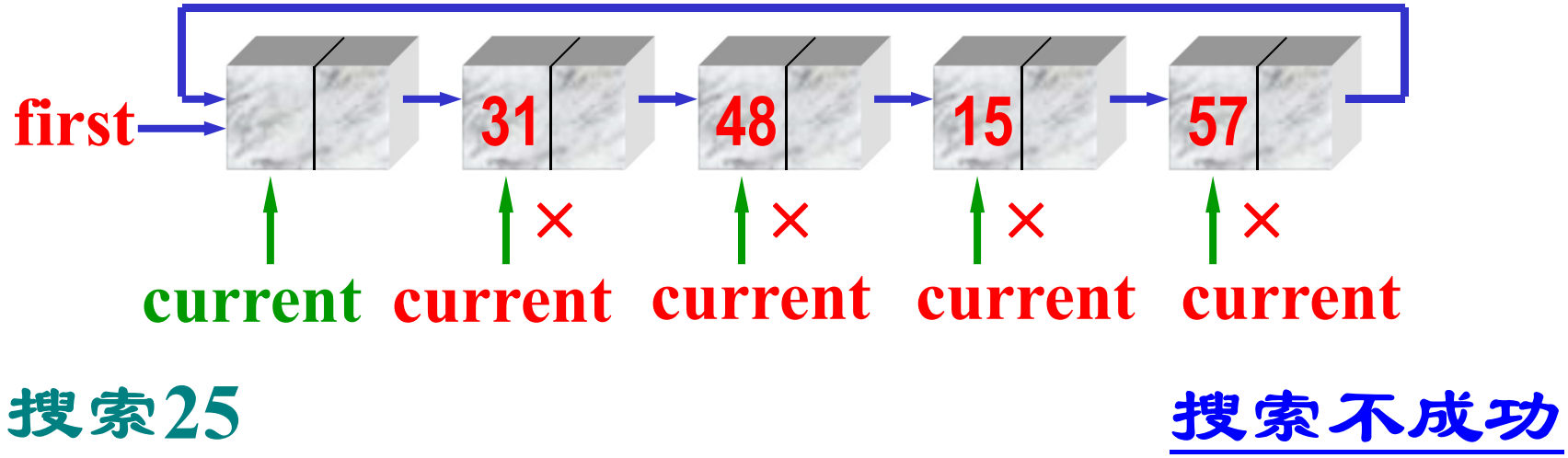
```
template <class Type> class CircList;

template <class Type> class CircLinkNode {
friend class CircList <Type>;
public:
    CircLinkNode ( Type d = 0,
        CircLinkNode <Type> *next = NULL )
        : data (d), link (next) { }  //构造函数
private:
    Type data;  //结点数据
    CircLinkNode <Type> *link;  //链接指针
};
```

```
template <class Type> class CircList {
//循环链表类
private:
    LinkNode <Type> *first, *last, *current;
    //循环链表的表头指针、尾指针和当前元素指针
public:
    CircList ( const Type &value );
    CircList ( CircList <Type> &L );
    ~CircList ( );
    int Length ( ) const;
    CircLinkNode <Type> * GetHead ( ) const;
    void SetHead ( CircLinkNode <Type> *p ) { first = p; }
```

```
    CircLinkNode <Type> * Search ( Type value );
    CircLinkNode <Type> * Locate ( int i );
    Type * GetData ( int i );
    void SetData ( int i, Type &x );
    bool Insert ( int i, Type &x );
    bool Remove ( int i, Type &x );
    bool IsEmpty ( ) const
            { return first->link == first ? true : false; }
    void Sort ( );
    void Input ( );
    void Output ( );
    CircList <Type> operator = ( CircList <Type> &L );
};
```

循环链表的搜索算法


first
31
48
15
57
×
×
√
current
current
current
搜索15
搜索成功

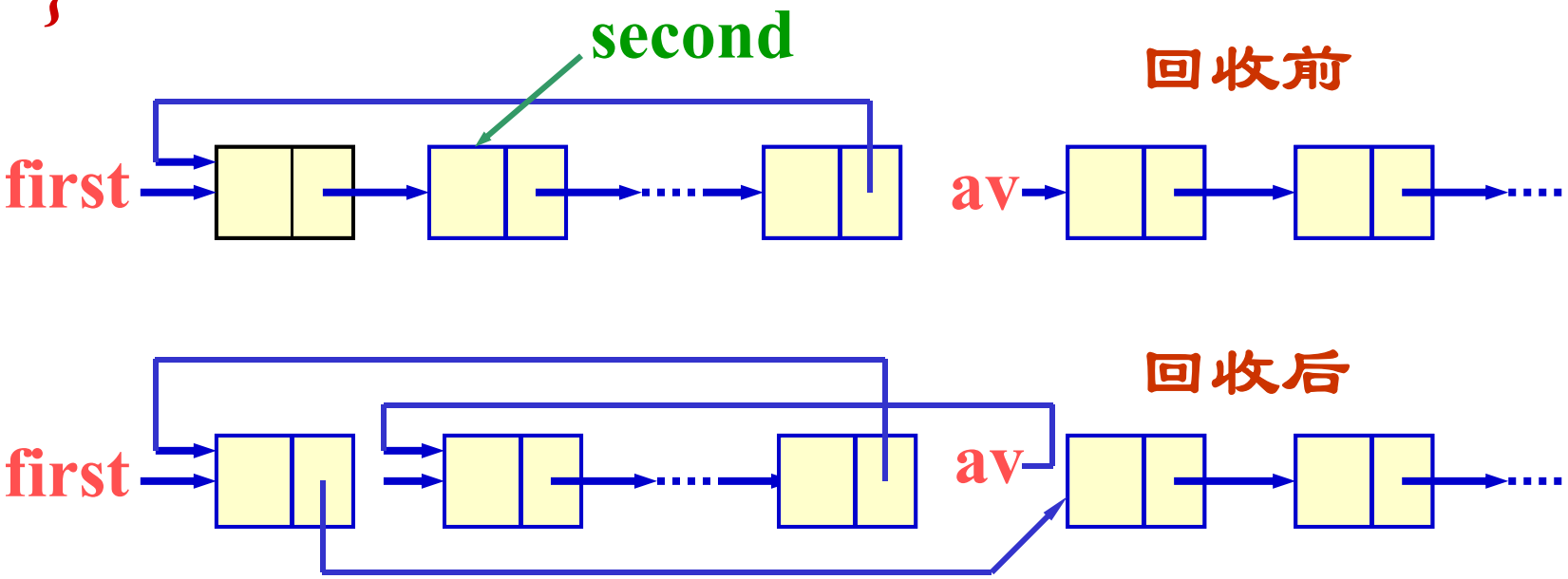

first
31
48
15
57
current
×
×
×
×
current
current
current
current
搜索25
搜索不成功

循环链表的搜索算法

```
template <class Type> CircLinkNode <Type>
        * CircList <Type> :: Search ( Type value )
{  //在链表中从头搜索其数据值为value的结点
    current = first->link;
    while ( current != first &&
                    current->data != value )
       current = current->link;
    return current;
}
```

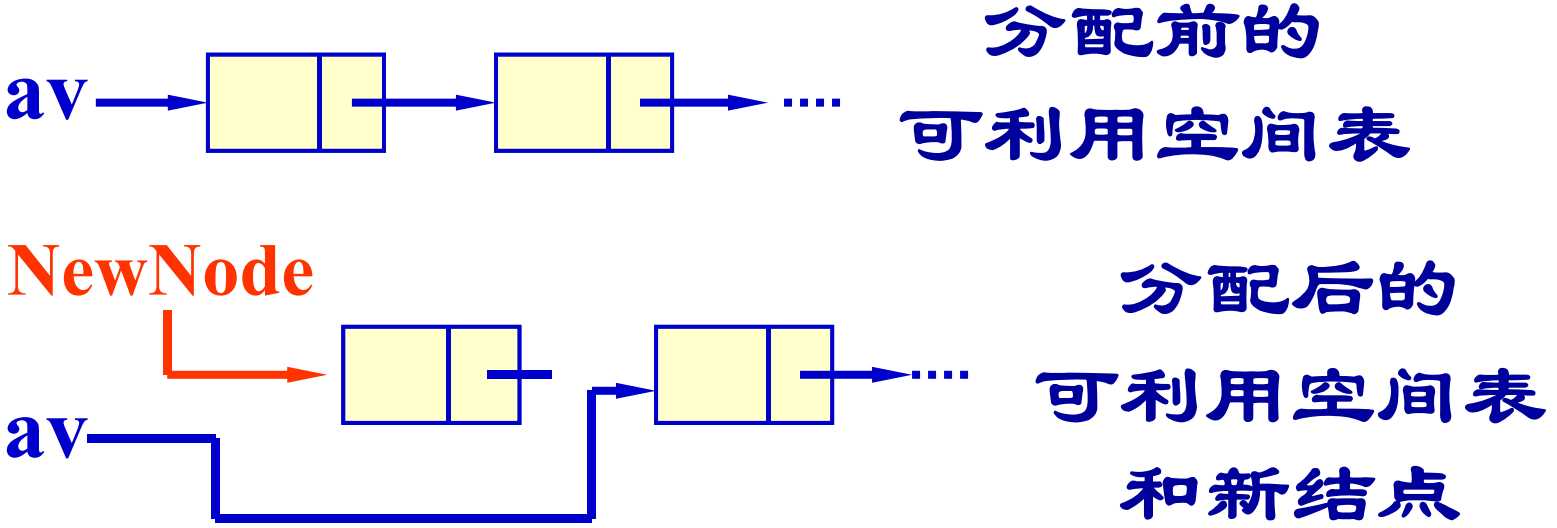
利用可利用空间表回收循环链表

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

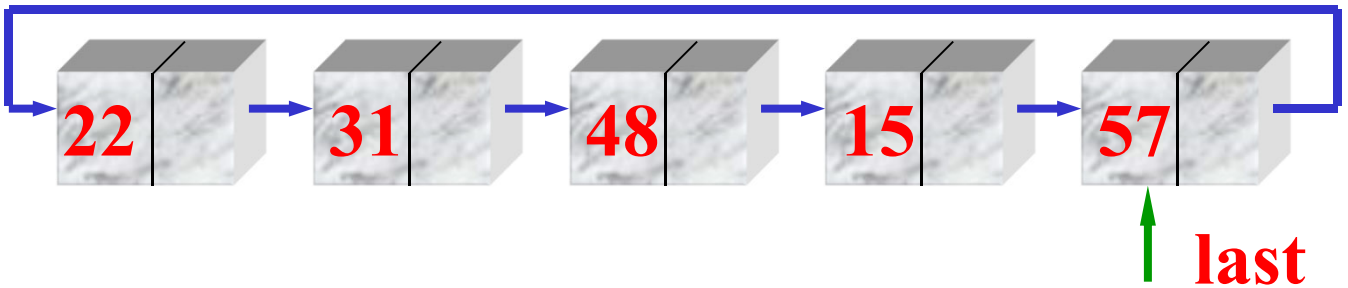


从可利用空间表分配结点

```
if ( av == NULL )
   Newnode = new CircLinkNode <Type>;
else { NewNode = av;  av = av->link; }
```



带尾指针的循环链表



如果插入与删除仅在链表的两端发生，可采用**带表尾指针的循环链表结构**。

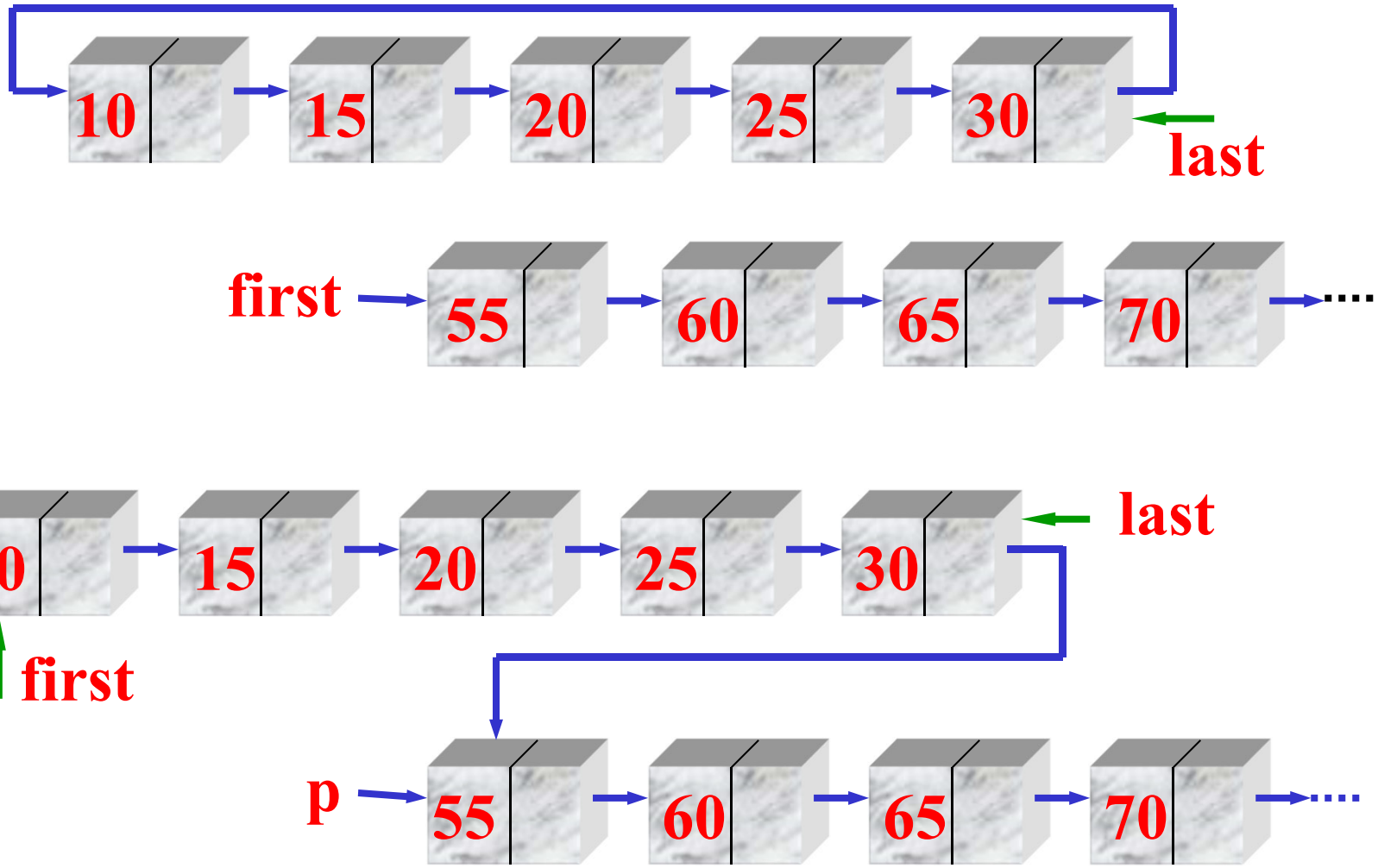
在表尾插入，时间复杂性 O(1)；

在表尾删除，时间复杂性 O(n)；

在表头插入，相当于在表尾插入；

在表头删除，时间复杂性 O(1)。

将循环链表链入单链表链头

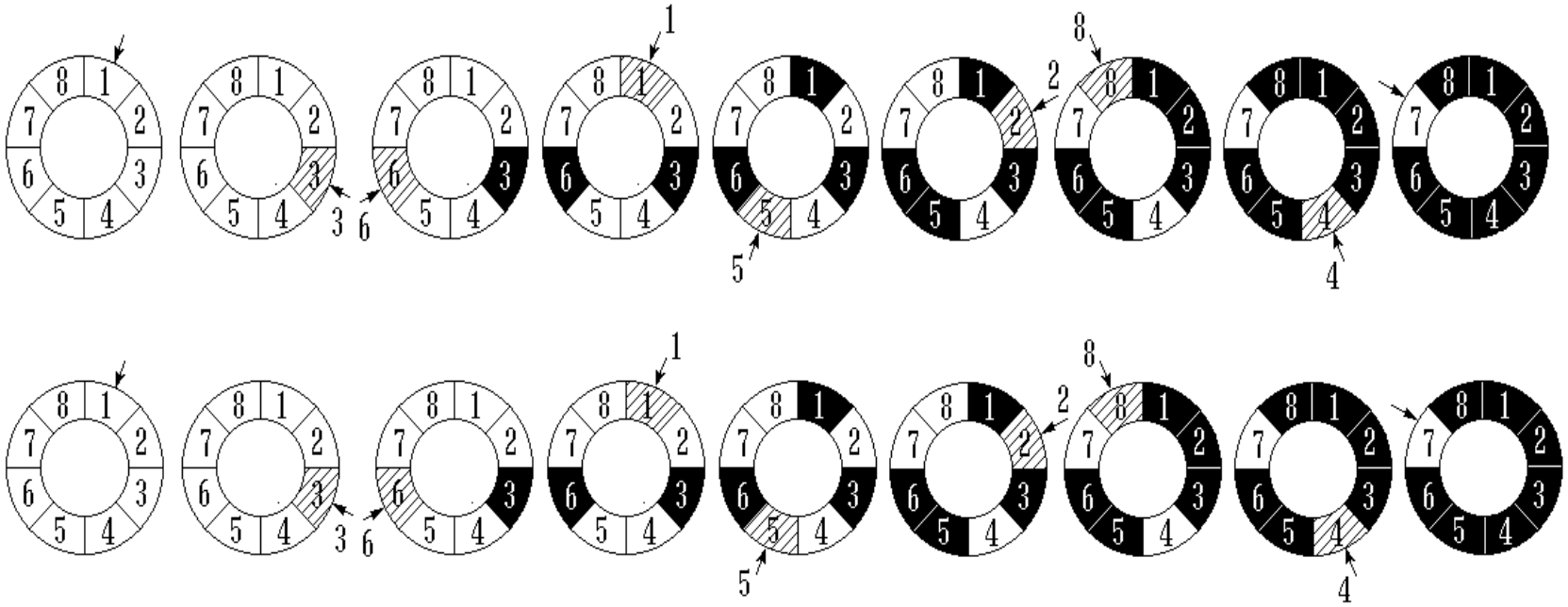


用循环链表求解约瑟夫问题

- **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，$n = 8 \quad m = 3$



约瑟夫问题的解法

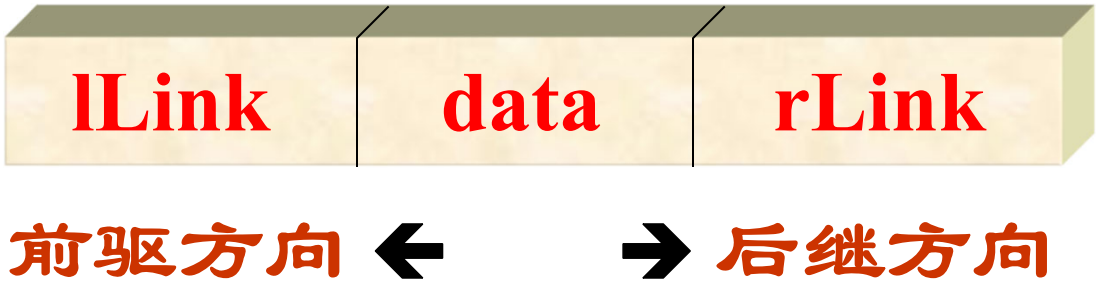
```
#include <iostream.h>
#include "CircList.h"
template <Type>
void Josephus ( CircList <Type> &Js, int n, int m )
{
   CircLinkNode <Type> *p = Js.GetHead ( ), *pre=NULL;
   for ( int i = 0; i < n-1; i++ ) {  //执行 n-1 次
      for ( int j = 0; j < m-1; j++ )
           { pre=p;  p = p->link; }
      cout << "出列的人是" << p->data << endl;
      pre->link = p->link;  delete p;
      p = pre->link;
   }
}
```

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

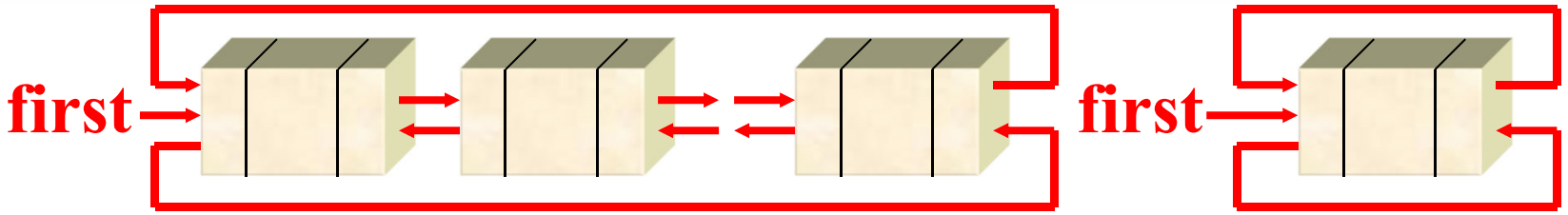
- 在单链表中，搜索一个指定结点的后继结点非常方便。
  - 只要该结点的*link*域的内容不为空，就可通过*link*域找到该结点的后继结点地址。
- 在单链表中，搜索一个指定结点的前驱结点非常不容易。
  - 必须从链头开始，沿*link*链顺序检测，直到某一结点的后继结点为该指定结点，则此结点即为该指定结点的前驱结点。
- 在一个应用问题中，经常要求检测指针向前驱和后继方向移动。
  - 需要保证移动的时间复杂度达到最小。

2.4.2 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



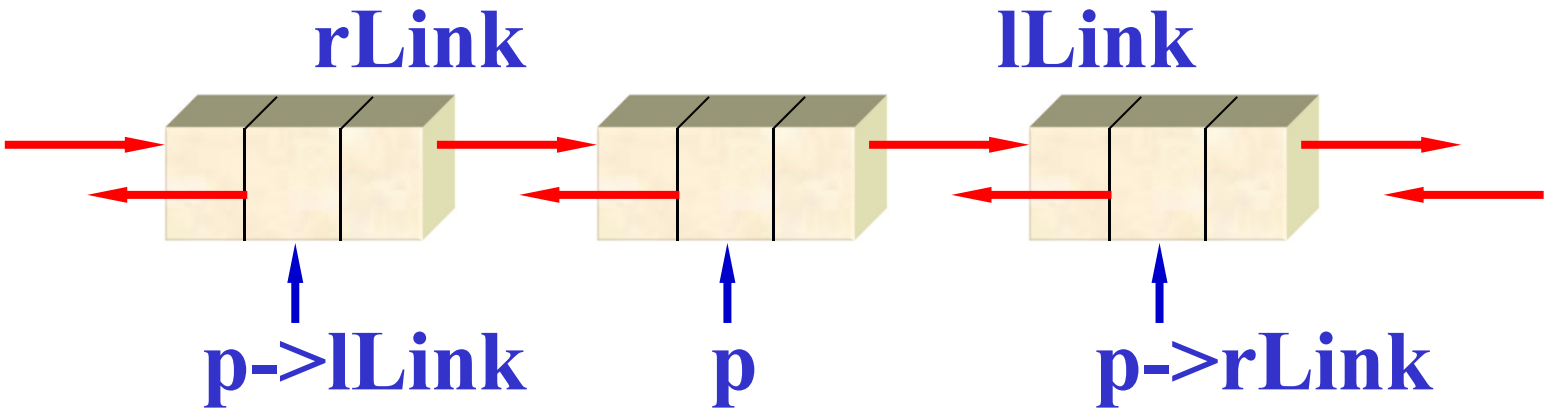
- 双向链表通常采用带表头结点的循环链表形式。



非空表　　空表

- **结点指向**

  p = p->lLink->rLink = p->rLink->lLink



双向循环链表类定义

```
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
public:
   DblNode ( Type value, DblNode <Type> *left,
                         DblNode <Type> *right ) :
      data (value), lLink (left), rLink (right) { }
```

```
    DblNode ( Type value ) : data (value),
        lLink (NULL), rLink (NULL) { }
    DblNode ( ) : data(0), lLink(NULL), rLink(NULL) { }
    DblNode <Type> * GetLeftLink ( )
      { return lLink; }  //取得左链指针值
    DblNode <Type> * GetRightLink ( )
      { return rLink; }  //取得右链指针值
    Type GetData ( ) { return data; }
    void SetLeftLink ( DblNode <Type> *Left )
      { lLink = Left; }  //修改左链指针值
    void SetRightLink ( DblNode <Type> *Right )
      { rLink = Right; }  //修改右链指针值
    void SetData ( Type value ) { data = value; }
};
```

```
template <class Type> class DblList {
private:
   DblNode <Type> *first, *current;
public:
   DblList ( Type uniqueVal );  //构造函数
   DblList ( DblList <Type> &RL );  //复制构造函数
   ~DblList ( );  //析构函数
   int Length ( ) const;  //计算长度
   bool IsEmpty ( )  { return first->rLink == first; }
   DblNode <Type> * GetHead const { first = ptr; }
   void SetHead ( DblNode <Type> *ptr )  { first = ptr; }
   DblLNode <Type> * Search ( const Type &x );
   DblLNode <Type> * Locate ( int i, int d );
   bool Insert ( int i, const Type &x, int d );
   bool Remove ( int i, Type &x, int d );
};
```

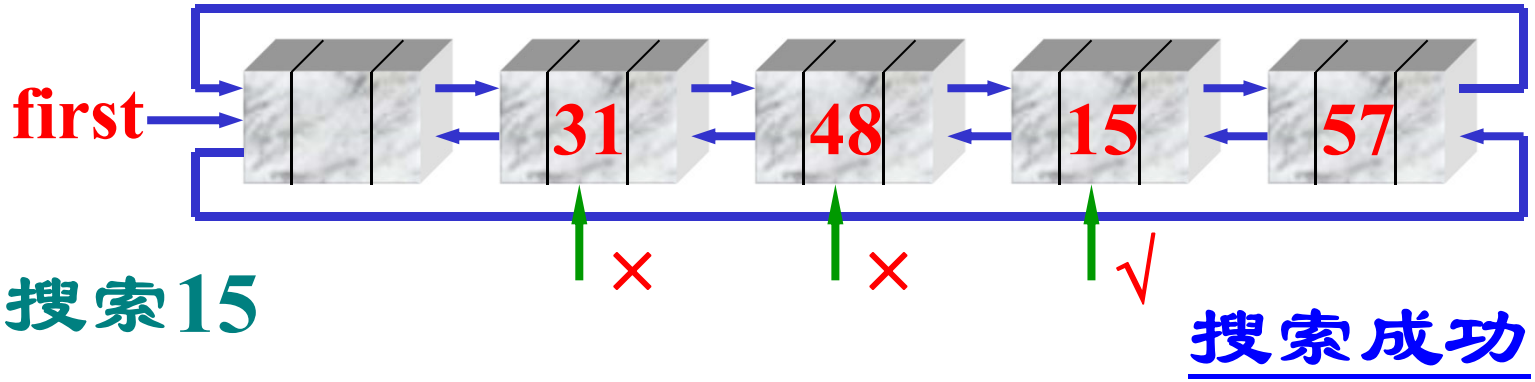
```
template <class Type>
DblList <Type> :: DblList ( Type uniqueVal ) {
//构造函数：建立双向循环链表的表头结点
    first = current =
          new DblNode <Type> ( uniqueVal );
    if (first == NULL )
       { cerr << "存储分配错！ \n" ; exit(1); }
    first->rLink = first->lLink = first;
}
```

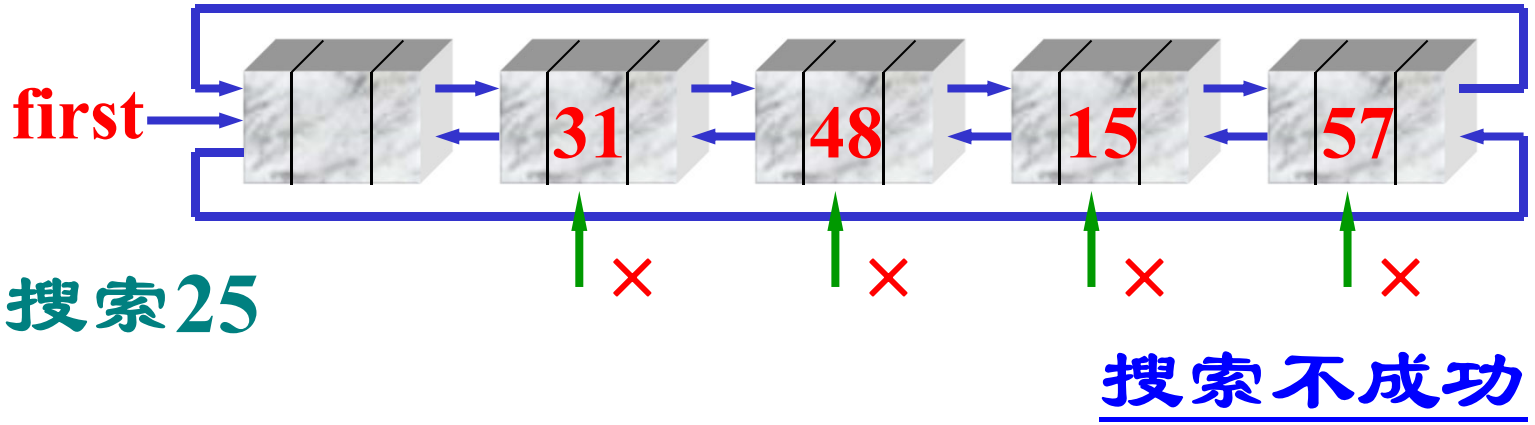
```
template <class Type>
DblList <Type> :: DblLIst ( DblList <Type> &RL )
{ //复制构造函数：用已有链表 RL 初始化
   first = new DblNode <Type> ( RL->data );
   if ( first == NULL )
     { cerr << "存储分配错!\n" ; exit(1); }
   first->lLink = first->rLink = first;
   if ( RL->current == RL->first ) current = first;
   ListNode <Type> *p = RL->first->rLink;
   ListNode <Type> *last = first;
```

```
while ( p != RL->first )
{  //逐个结点复制
   last->rLink = new DblNode <Type>
                              ( p->data, last, last->rLink );
   if ( last->rLink == NULL )  //分配新结点
     { cerr << "存储分配错!\n" ; exit(1); }
   last = last->rLink;
   last->rLink->lLink = last;  //链结完成
   if ( RL->current == p ) current = last;
   p = p->rLink;
}
}
```

```
template <class Type>
int DblList <Type> :: Length ( ) const {
//计算带表头结点的双向循环链表的长度
    DblNode <Type> *p = first->rLink;
    int count = 0;
    while ( p != first )
        { p = p->rLink;  count++; }
    return count;
}
//按前驱方向计算只需左、右互换即可
```

双向循环链表的搜索算法

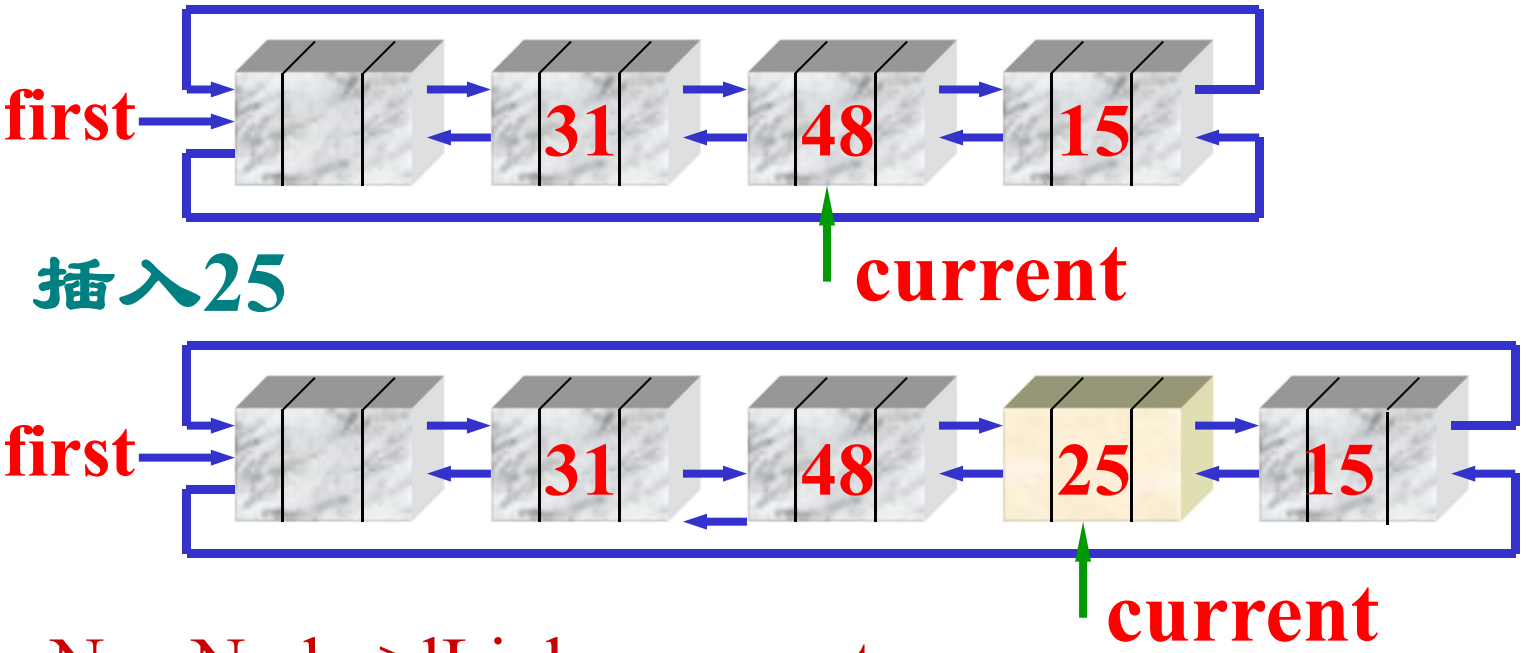




```
template <class Type> DblNode <Type>
        * DblList <Type> :: Search ( const Type &x ) {
//在双向循环链表中搜索含 x 的结点
   current = first->rLink;
   while ( current != first &&
                  current->data != x )
       current = current->rLink;
   if ( current != first ) return current;
   else return NULL;
}
```

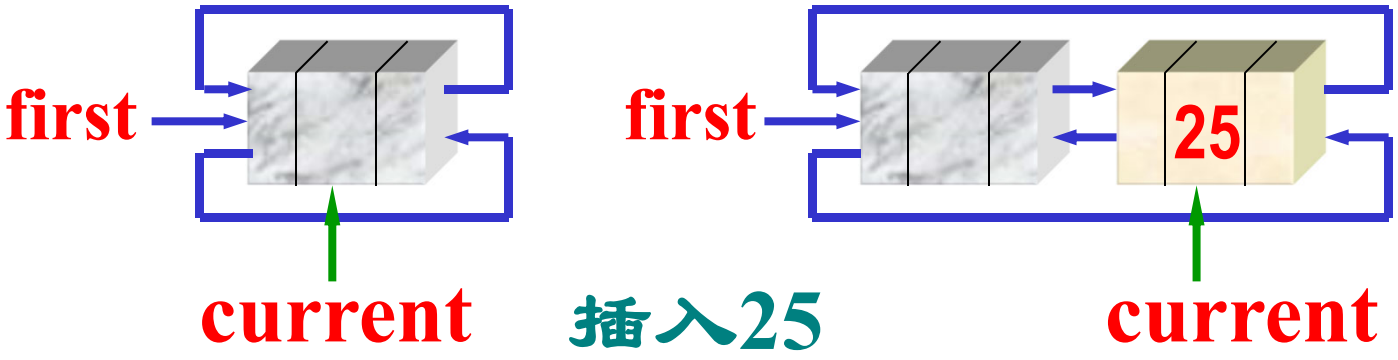
```
template <class Type> DblNode <Type>
        * DblList <Type> :: Locate ( int i, int d ) {
    if ( first->rlink == first || i == 0 ) return first;
    if ( d==0 ) current = first->lLink;
    else current = first->rLink;
    for ( int j=1; j<i; j++ )
        if ( current == first ) break;
        else if ( d==0 ) current = current->lLink;
            else current = current->rLink;
    if ( current != first ) return current;
    else return NULL;
}
```

双向循环链表的插入算法（非空表）




```
NewNode->lLink = current;
NewNode->rLink = current->rLink;
current->rLink = NewNode;
current = current->rLink;
current->rLink->lLink = current;
```

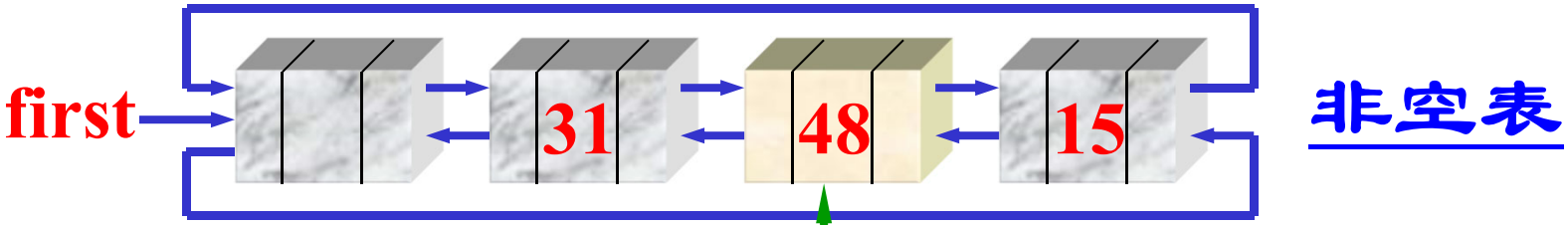
双向循环链表的插入算法（空表）




```
NewNode->lLink = current;
NewNode->rLink = current->rLink;  ( = first )
current->rLink = NewNode;
current = current->rLink;
current->rLink->lLink = current;
      ( first->lLink = current )
```

```
template <class Type> int DblList <Type> ::
Insert ( int i, const Type &x, int d ) {
    current = Locate ( i, d );
    if ( current == NULL ) return false;
    DblNode <Type> *NewNode = new DblNode <Type> (x);
    if ( NewNode == NULL )
      { cerr << "存储分配失败！" << endl; exit(1); }
    if ( d == 0 ) {
       NewNode->lLink = current->lLink;
       current->lLink = NewNode;
       NewNode->lLink->rLink = NewNode;
       NewNode->rLink = current;   }
    else {
       NewNode->rLink = current->rLink;
       current->rLink = NewNode;
       NewNode->rLink->lLink = NewNode;
       NewNode->lLink = current;   }
    return true;
}
```

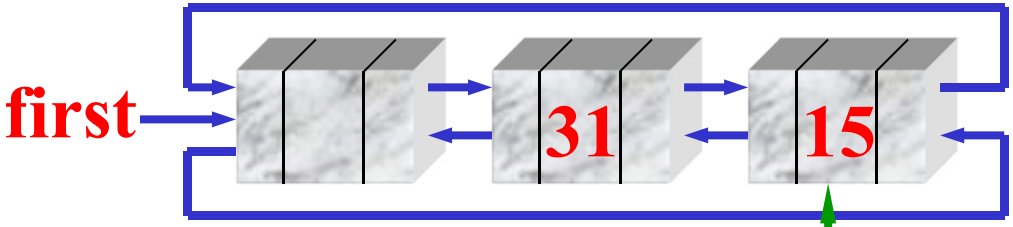
双向循环链表的删除算法



非空表

删除48

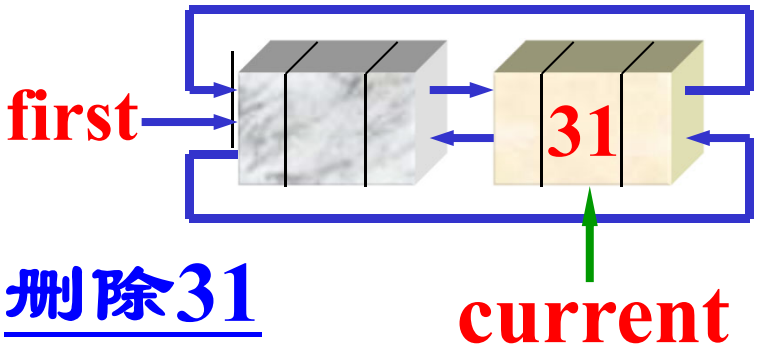
current

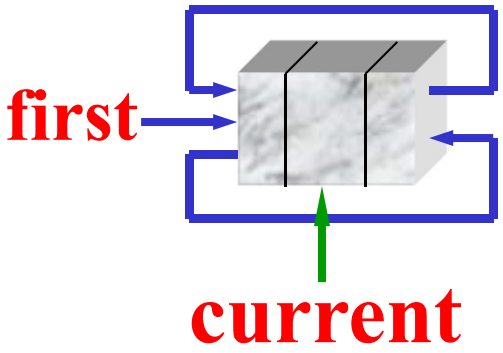


current

```
current->rLink->lLink = current->lLink;
current->lLink->rLink = current->rLink;
```

双向循环链表的删除算法






```
current->rLink->lLink = current->lLink;
current->lLink->rLink = current->rLink;
```

```
template <class Type> bool DblList <Type> ::
Remove ( int i, Type &x, int d) {
    DblNode <Type> *current = Locate ( i, d );
    if ( current == NULL ) return false;
    current->rLink->lLink = current->lLink;  //摘下
    current->lLink->rLink = current->rLink;
    x = current->data;  delete current;
    return true;
}
```

多项式 (Polynomial)

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

$$= \sum_{i=0}^{n} a_i x^i$$

- n阶多项式 $P_n(x)$ 有 $n+1$ 项
  - 系数 $a_0, a_1, a_2, \ldots, a_n$
  - 指数 $0, 1, 2, \ldots, n$ 按升幂排列

多项式(*Polynomial*)类定义

```
class Polynomial {
public:
    Polynomial ( );  //构造函数
    int operator ! ( );  //判是否零多项式
    float Coef ( int e );
    int LeadExp ( );  //返回最大指数
    Polynomial Add ( Polynomial poly );
    Polynomial Mult ( Polynomial poly );
    float Eval ( float x );  //求值
};
```

多项式的存储表示

第一种： 定义一个有**maxDegree+1**个元素的数组表示系数，数组下标表示相应的指数：

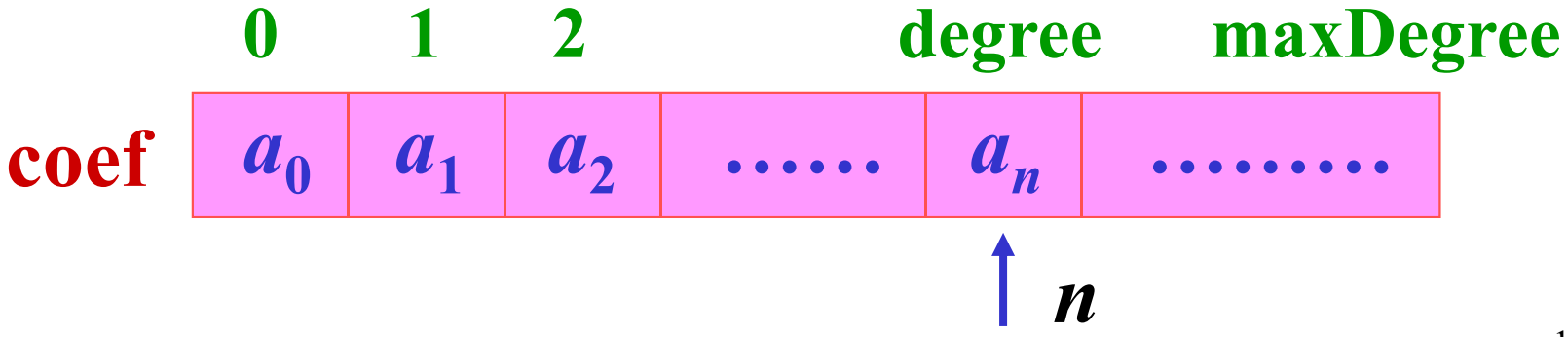
静态数组表示

```
private:
    int degree;
    float coef [maxDegree+1];
```

$P_n(x)$可以表示为：

pl.degree = n

pl.coef[i] = a_i, $0 \le i \le n$



第二种： 当**p.degree << maxDegree**时，方法1很浪费空间，可动态定义**coef**数组的元素个数：

动态数组表示

```
private:
    int degree;
    float *coef;
Polynomial :: Polynomial ( int sz ) {
    degree = sz;
    coef = new float [degree + 1];
}
```

以上两种存储表示适用于指数连续排列的多项式，但对于指数不全的多项式如 $P_{101}(x) = 3 + 5x^{50} - 14x^{101}$，不经济。

第三种：对于有很多零项的稀疏多项式，方法2仍然很浪费空间。例如，$x^{800}+3$仅有2个非零项，其余799项都是零项。这时，只存储非零项更为有效。不仅需要显式存储系数，而且需要显式存储指数。

```
class Polynomial;
class Term {  //多项式的项定义
   friend Polynomial;
 private:
   float coef;  //系数
   int exp;  //指数
};
```

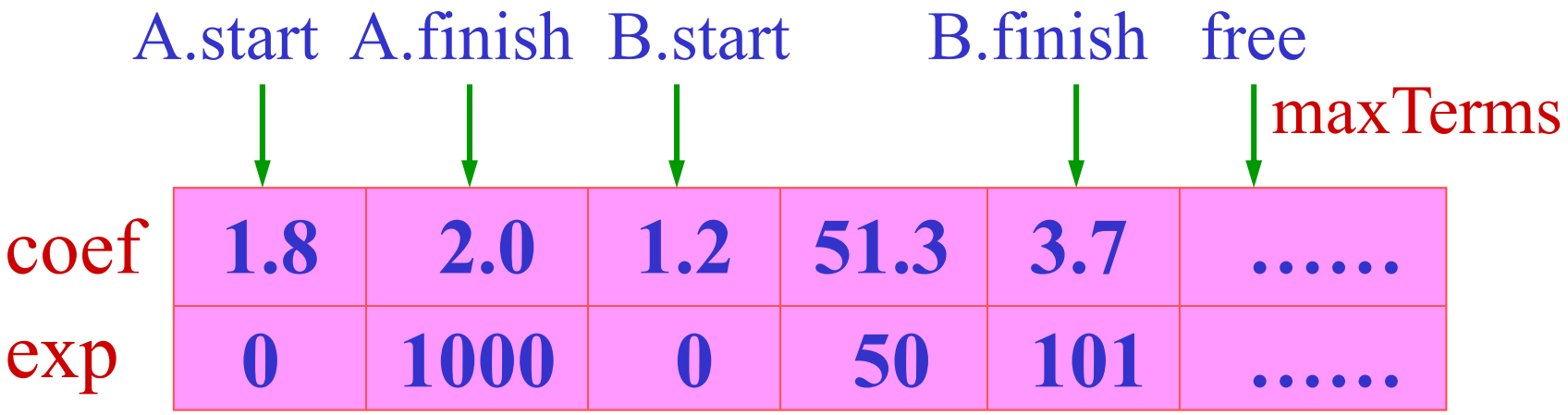
	0	1	2		*i*		*m*
coef	a_0	a_1	a_2	……	a_i	……	a_m
exp	e_0	e_1	e_2	……	e_i	……	e_m

```
class Polynomial { //多项式类定义
public:
   ……
private:
   static Term TermArray[maxTerms]; //项数组
   static int free; //当前空闲位置指针
   // Term Polynomial :: TermArray[maxTerms];
   // int Polynomial :: free = 0;
   int start, finish; //多项式始末位置
};
```

其中，**maxTerms**是常数。由于类中的静态成员声明不构成其定义，还必须在类定义之外定义静态成员如下：

```
Term Polynomial :: TermArray[maxTerms];
int Polynomial :: free = 0; //指示TermArray中的下一个可用单元
```

$$A(x) = 1.8 + 2.0x^{1000}$$
$$B(x) = 1.2 + 51.3x^{50} + 3.7x^{101}$$



两个多项式存放在TermArray中

一个多项式p(x)的项数为p.finish–p.Start+1。
当多项式的零项很多时，方法3明显好于方法2。但当绝大多数都是非零项时，方法3所用空间大约是方法2的两倍。

两个多项式的相加

- 结果多项式另存。
- 扫描两个相加多项式，若都未检测完：
  - 若当前被检测项指数相等，系数相加。若未变成 0，则将结果加到结果多项式。
  - 若当前被检测项指数不等，将指数小者加到结果多项式。
- 若有一个多项式已检测完，将另一个多项式剩余部分复制到结果多项式。

用方法3表示多项式A和B。由于多项式的项是按指数递增的顺序排列，因而通过对A和B逐项扫描，比较指数，很容易实现 C=A+B。

```
Polynomial Polynomial ::
operator + ( Polynomial B ) {
    Polynomial C;
    int a = start;  int b = B.start;  C.start = free;
    float c;
    while ( a <= finish && b <= B.finish )
        switch ( compare ( TermArray[a].exp,
          TermArray[b].exp) ) {  //比较对应项指数
        case '=' :  //指数相等
          c = TermArray[a].coef  +  //系数相加
                              TermArray[b].coef;
          if ( c ) NewTerm ( c, TermArray[a].exp );
          a++;  b++;  break;
```

```
case '>' : // b 指数小，建立新项
    NewTerm ( TermArray[b].coef,
                        TermArray[b].exp );
    b++; break;
case '<' : // a 指数小，建立新项
    NewTerm ( TermArray[a].coef,
                        TermArray[a].exp );
    a++;
}
for (  ; a <= finish; a++ )  // a 未检测完时
    NewTerm ( TermArray[a].coef,
                        TermArray[a].exp );
```

```
    for (  ; b <= B.finish; b++ )  // b 未检测完时
        NewTerm ( TermArray[b].coef,
                                TermArray[b].exp );
    C.finish = free-1;
    return C;
}
```

在多项式中加入新的项

```
void Polynomial :: NewTerm ( float c, int e ) {
//把一个新的项加到多项式 C(x) 中
   if ( free >= maxTerms ) {
      cout << "Too many terms in polynomials" << endl;
      return;
   }
   TermArray[free].coef = c;
   TermArray[free].exp = e;
   free++;
}
```

用函数 **NewTerm** 将新的属于 **C** 的项存入 **free** 所指的 **TermArray** 可用单元。

多项式的链表表示

- 在多项式的链表表示中每个结点三个数据成员：

Data ≡ Term

- 优点
  - 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。

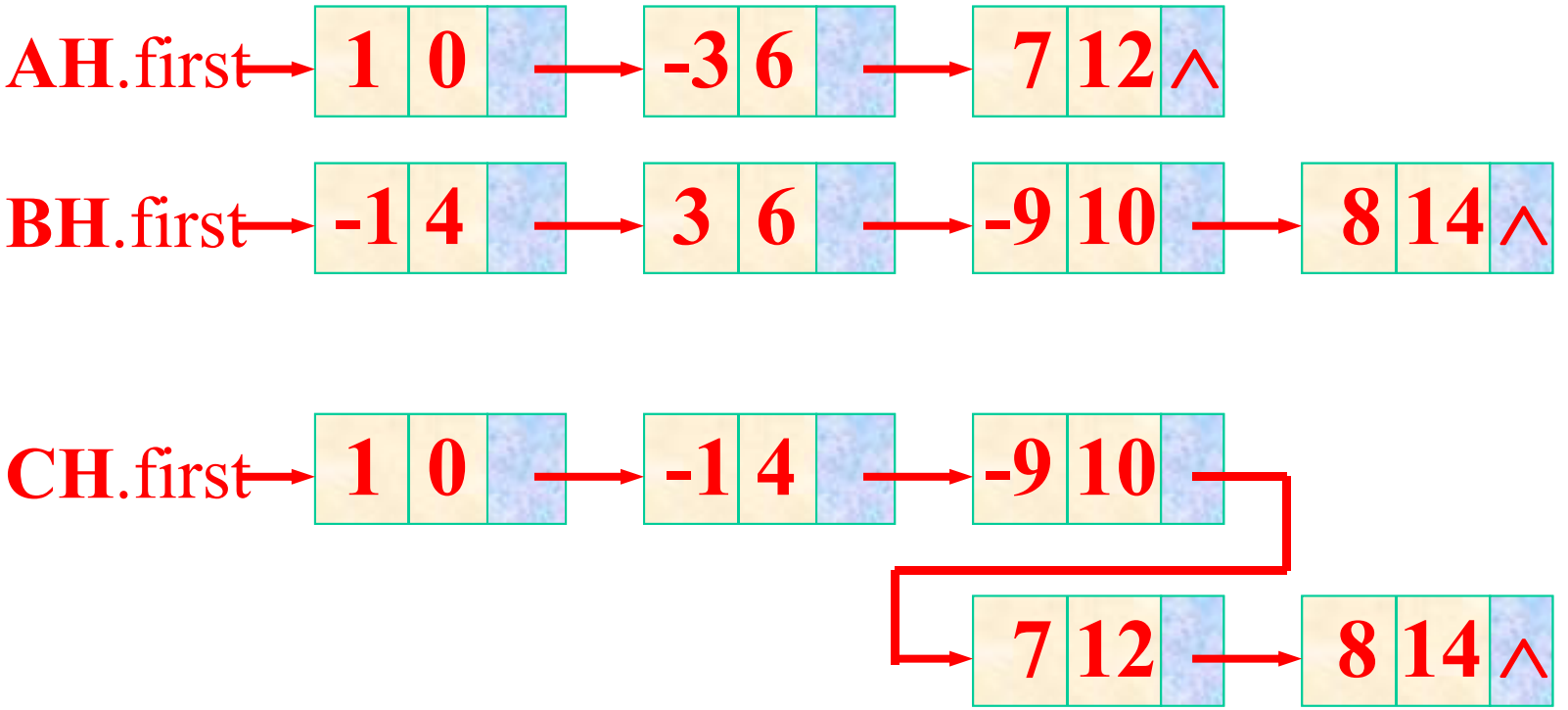
多项式(Polynomial)类的链表定义

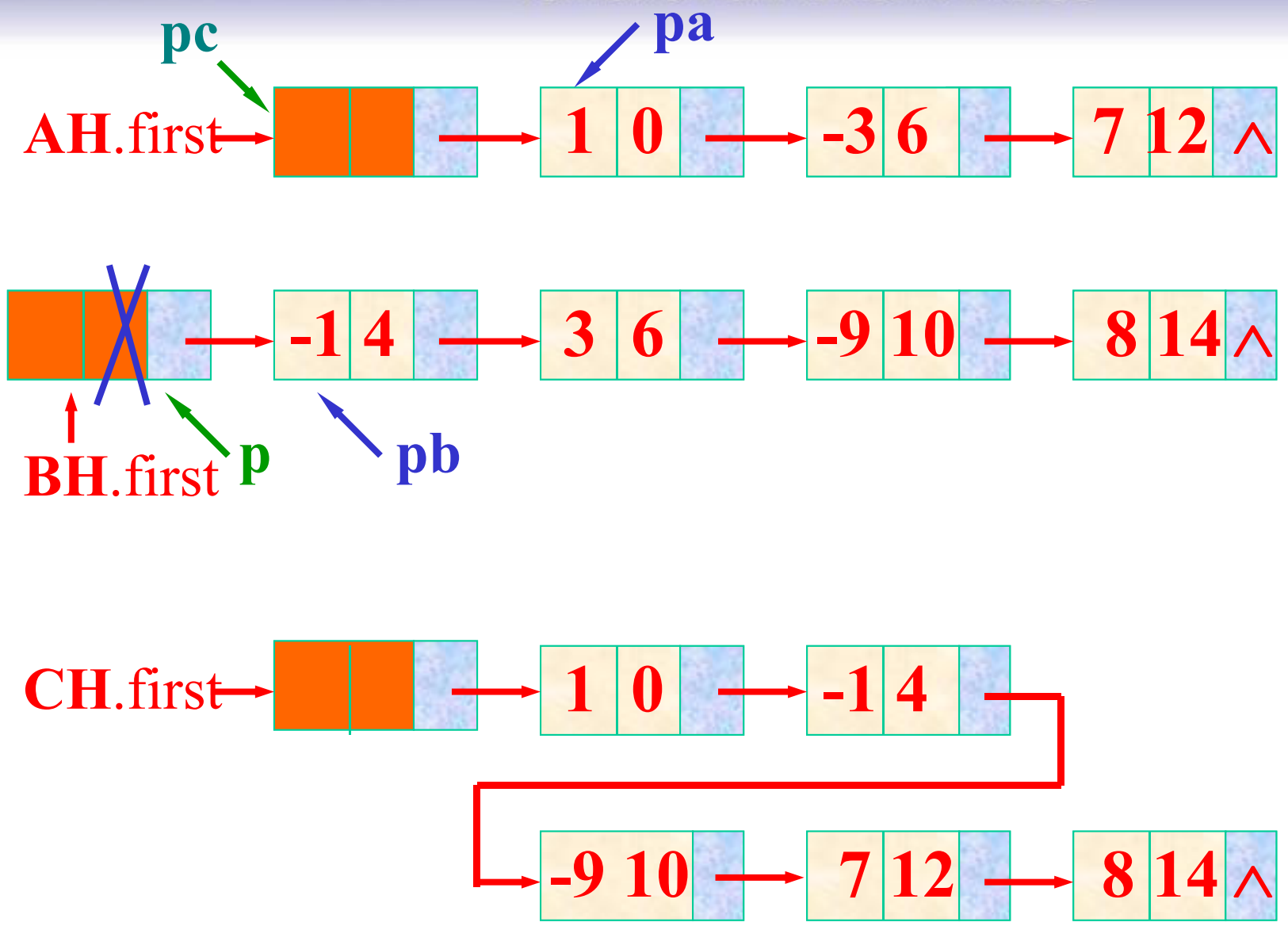
```
struct Term {  //多项式结点定义
    float coef;  //系数
    int exp;  //指数
    Term *link;
    Term ( float c, int e ) { coef = c;  exp = e; }
};
class Polynomial  {  //多项式类的定义
    List <Term> poly;
    friend Polynomial operator +
        ( Polynomial &, Polynomial &);  //加法
};
```

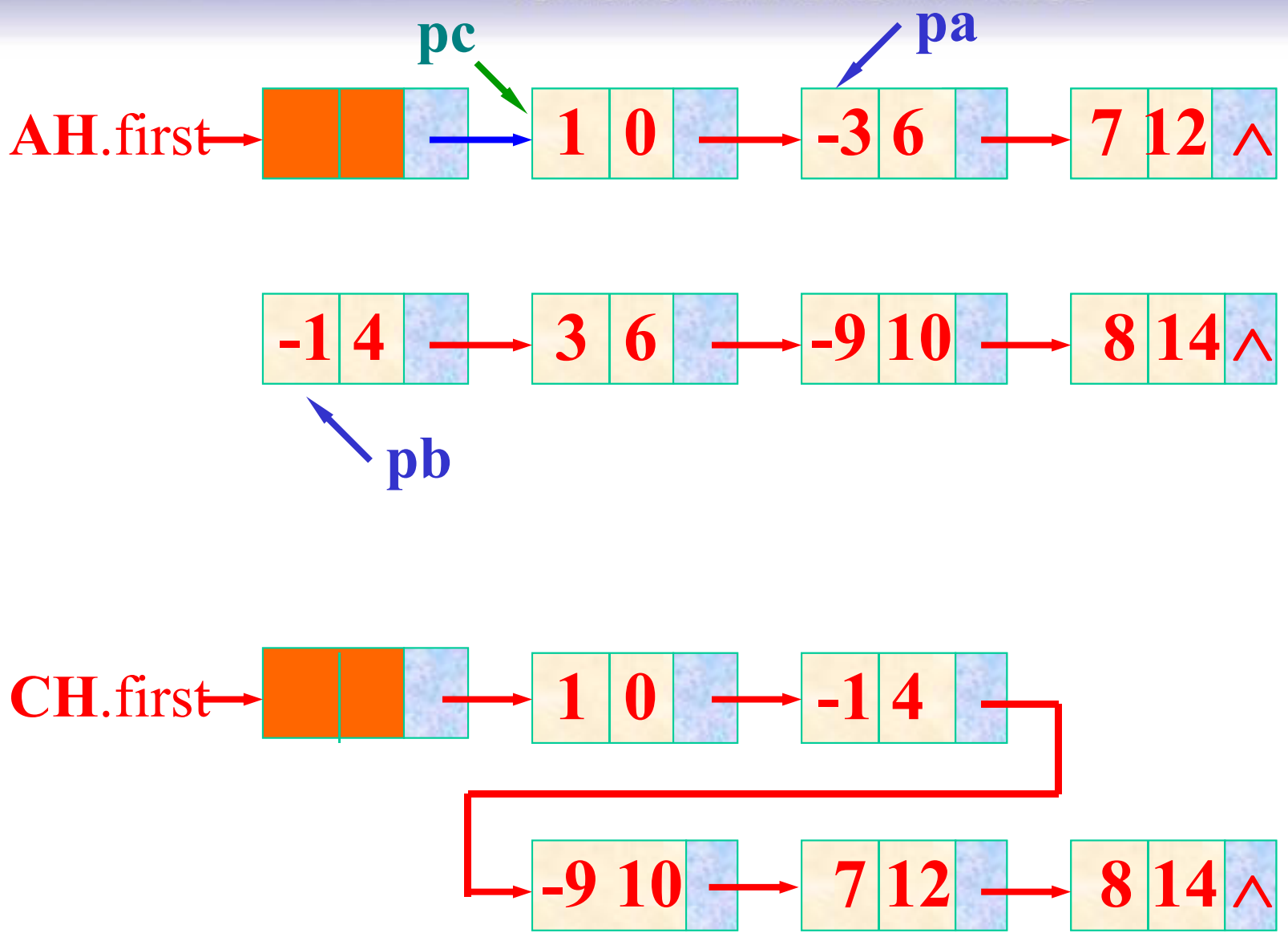
多项式链表的相加

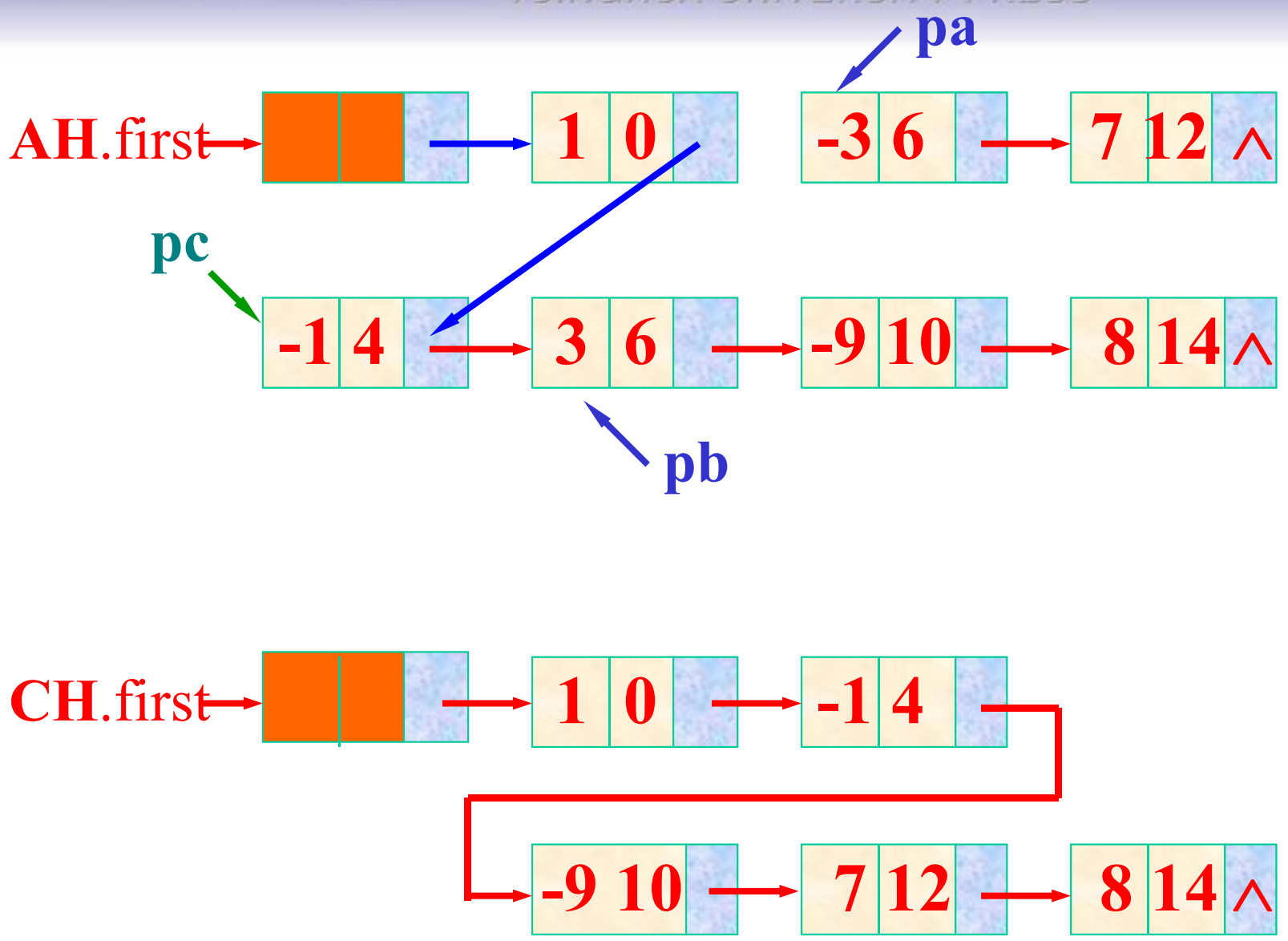
$$AH = 1 - 3x^6 + 7x^{12}$$

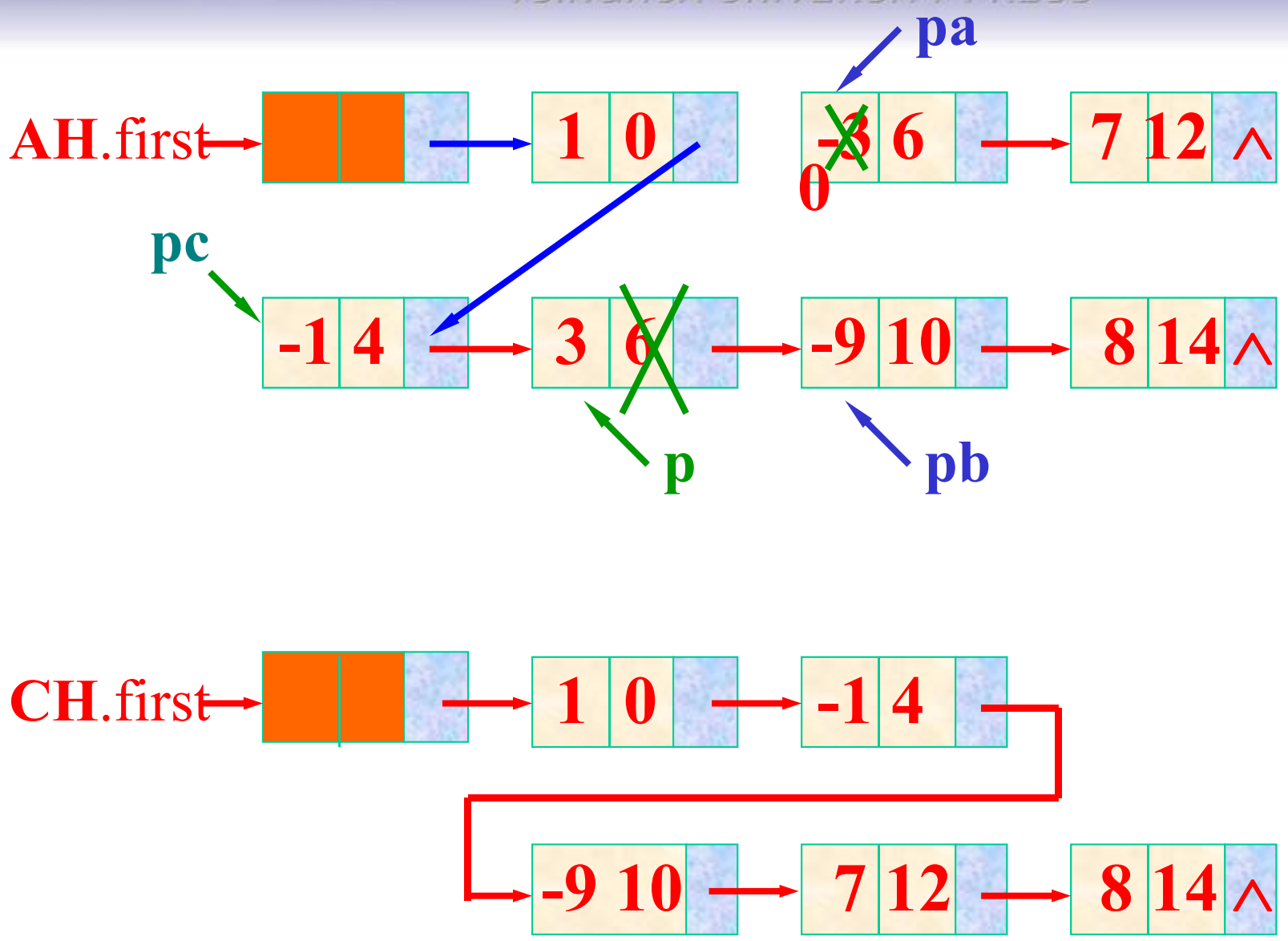
$$BH = -x^4 + 3x^6 - 9x^{10} + 8x^{14}$$

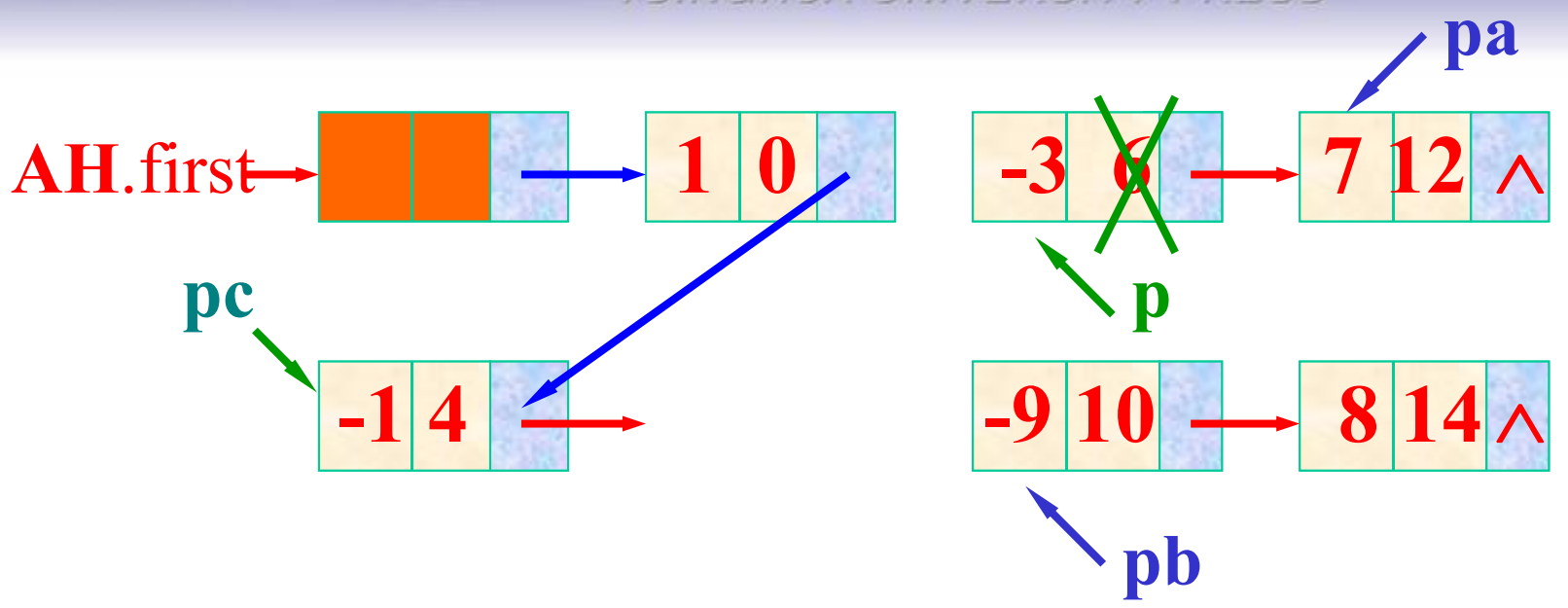


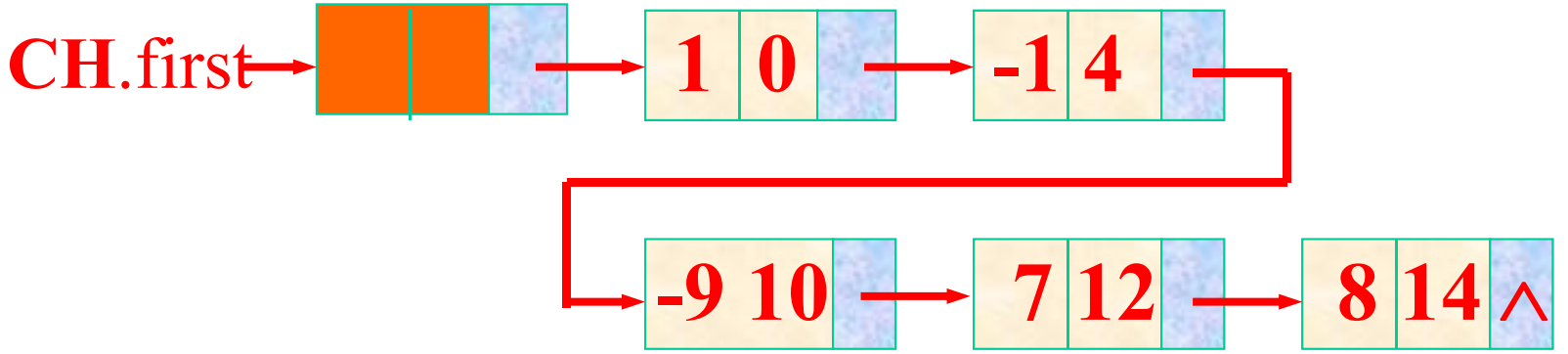

pc
pa
AH.first
1 0
-3 6
7 12 ∧
-1 4
3 6
-9 10
8 14 ∧
BH.first
p
pb
CH.first
1 0
-1 4
-9 10
7 12
8 14 ∧

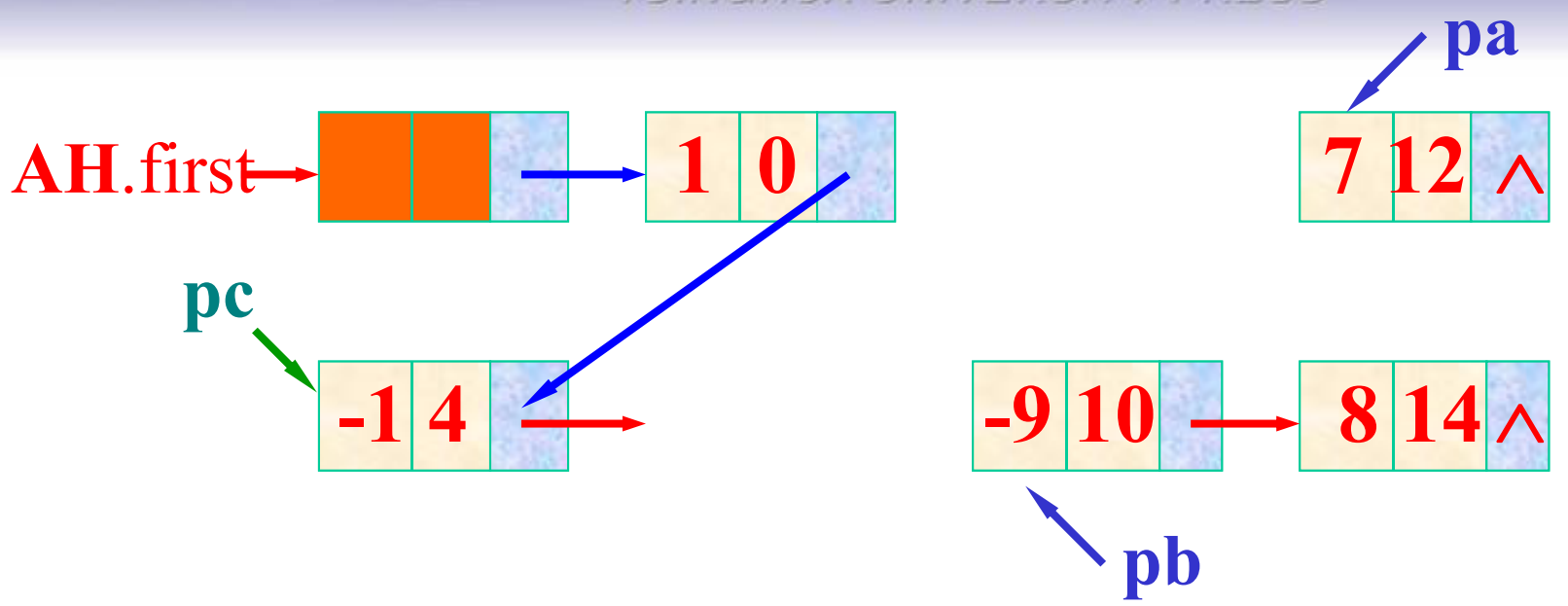

pc
pa
AH.first
1 0
-3 6
7 12 ∧
-1 4
3 6
-9 10
8 14 ∧
pb
CH.first
1 0
-1 4
-9 10
7 12
8 14 ∧

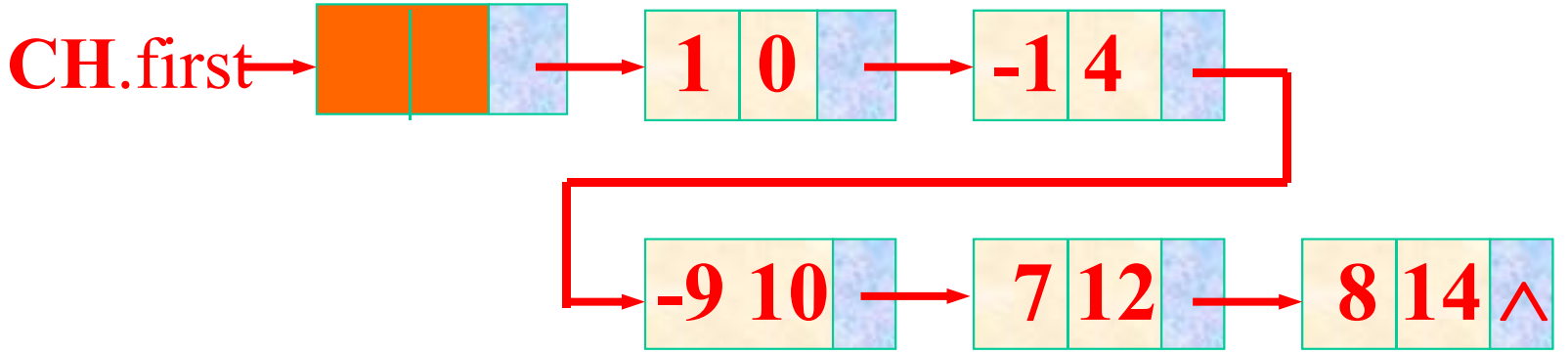

pa
AH.first
1 0
-3 6
7 12 ∧
pc
-1 4
3 6
-9 10
8 14 ∧
pb
CH.first
1 0
-1 4
-9 10
7 12
8 14 ∧

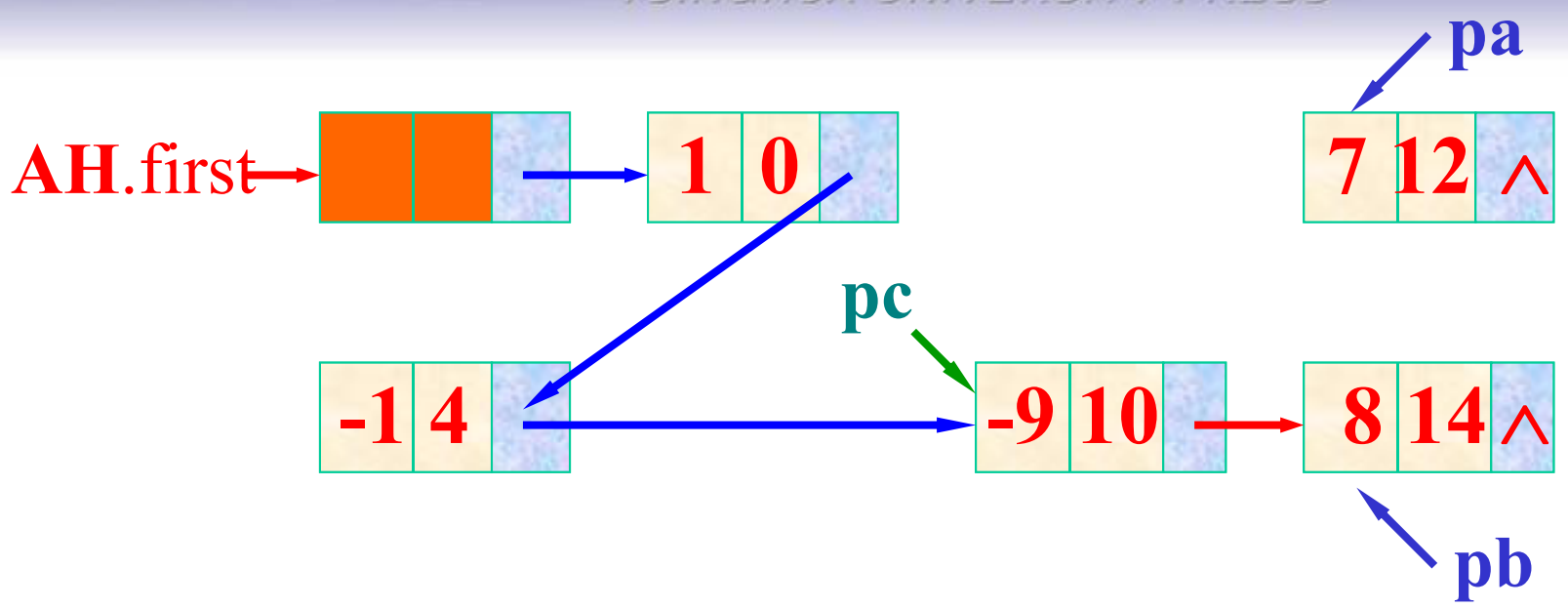

pa
AH.first
1 0
-3 6
0
7 12 ∧
pc
-1 4
3 6
-9 10
8 14 ∧
p
pb
CH.first
1 0
-1 4
-9 10
7 12
8 14 ∧

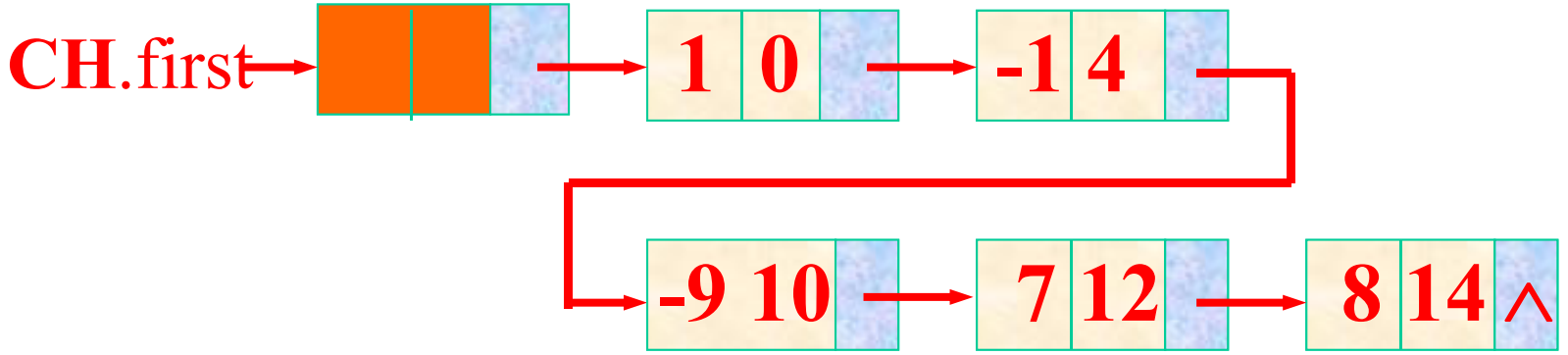

pa
AH.first
1 0
-3 6
7 12 ∧
pc
p
-1 4
-9 10
8 14 ∧
pb

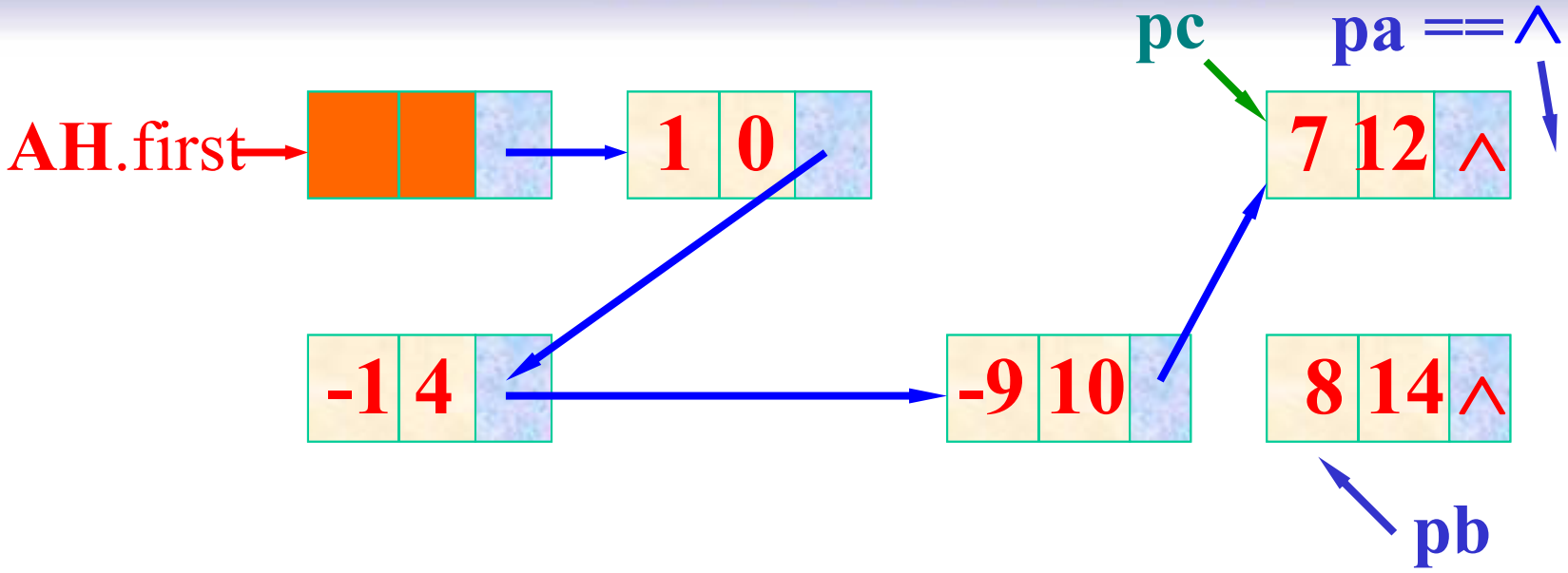

CH.first
1 0
-1 4
-9 10
7 12
8 14 ∧

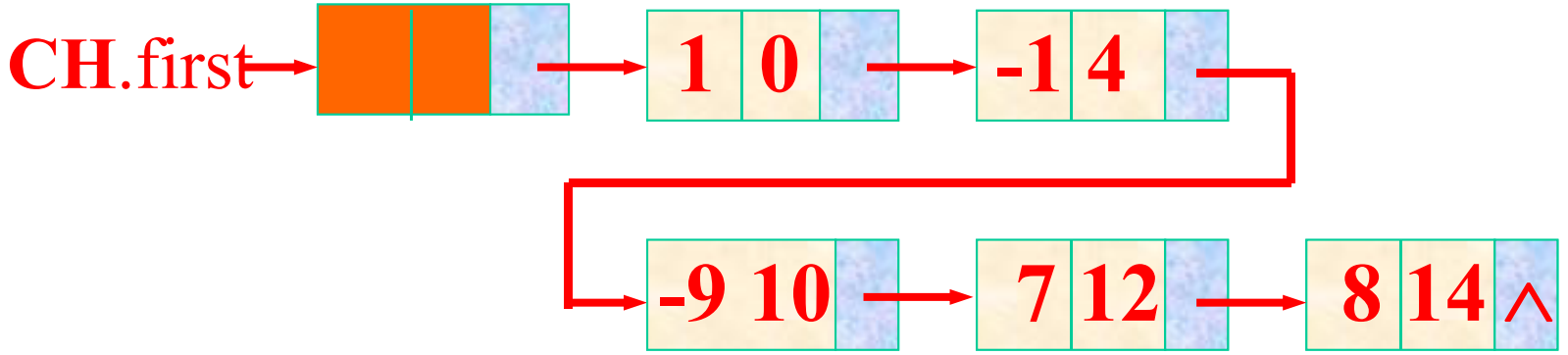

pa
AH.first
1 0
7 12 ∧
pc
-1 4
-9 10
8 14 ∧
pb

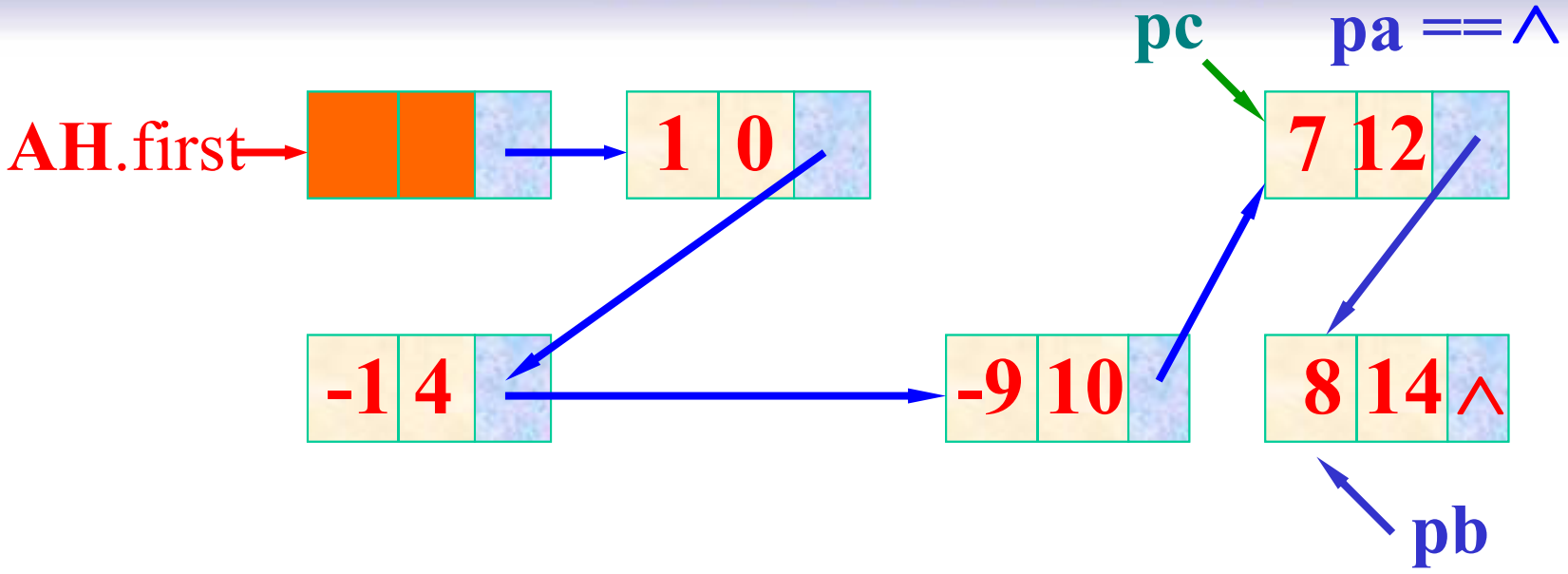

CH.first
1 0
-1 4
-9 10
7 12
8 14 ∧

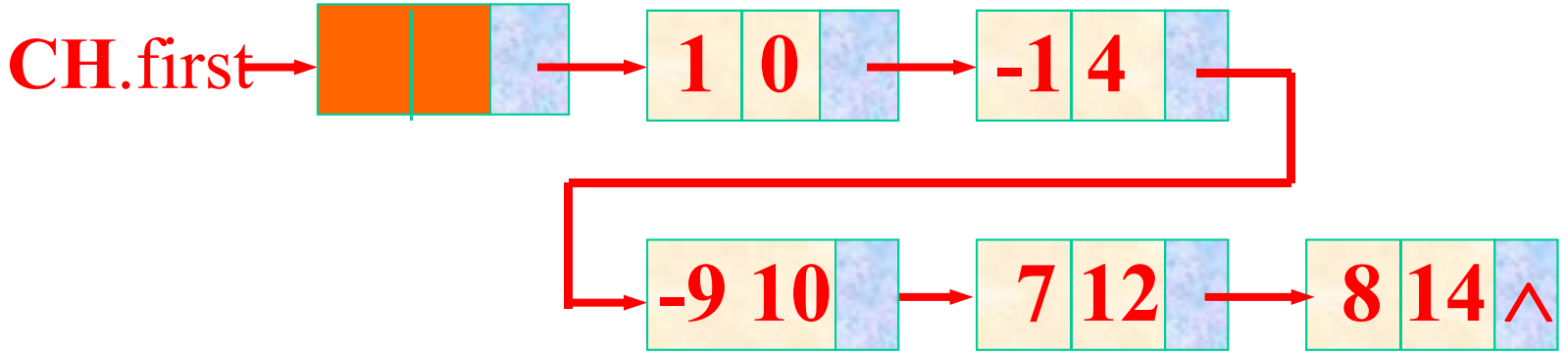

pa
AH.first
1 0
7 12 ∧
pc
-1 4
-9 10
8 14 ∧
pb


CH.first
1 0
-1 4
-9 10
7 12
8 14 ∧


pc
pa == ^
AH.first
1 0
7 12 ^
-1 4
-9 10
8 14 ^
pb


CH.first
1 0
-1 4
-9 10
7 12
8 14 ^


pc
pa == ∧
AH.first
1 0
7 12
-1 4
-9 10
8 14 ∧
pb


CH.first
1 0
-1 4
-9 10
7 12
8 14 ∧

```
Polynomial operator +
    ( Polynomial &ah, Polynomial &bh ) {
//两个带头结点的按升幂排列的多项式相加
//返回结果多项式链表表头指针
//结果不另外占用存储，覆盖 ah 和 bh 链表
    LinkNode <Term> *pa, *pb, *pc, *p;
    Term a, b;
    pc = ah.poly.Firster ( );  //结果存放指针
    p = bh.poly.Firster ( );
    pa = ah.poly.First ( );  //多项式检测指针
    pb = bh.poly.First ( );  //多项式检测指针
    delete p;  //删去 bh 的表头结点
```

```
while ( pa != NULL && pb != NULL ) {
  a = pa->data;  b = pb->data;
  switch ( compare ( a.exp, b.exp ) ) {
  case '=' :  // pa->exp == pb->exp
       a.coef = a.coef + b.coef;  //系数相加
       p = pb;  pb = pb->link;
       delete p;  //删去原 pb 所指结点
       if ( abs(a.coef ) < 0.0001 ) {
            p = pa;  pa = pa->link;  delete p;
       } //相加为零, 该项不要
       else {  //相加不为零, 加入 ch 链
            pa->data = a;  pc->link = pa;
```

```
                pc = pa;  pa = pa->link;  }
            break;
        case '>' :  // pa->exp > pb->exp
              pc->link = pb;  pc = pb;
              pb = pb->link;
              break;
        case '<' :  // pa->exp < pb->exp
             pc->link = pa;
             pc = pa;  pa = pa->link;    }    }
    if ( pa != NULL ) pc->link = pa;
    else pc->link = pb;  //剩余部分链入 ch 链
    return ah;
}
```

```
Polynomial operator + ( const Polynomial &ah,
                        const Polynomial &bh ) {
  LinkNode <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 ( );
  pb = p = Biter.First ( );
  //取得 ah 与 bh 的表头结点
  pa = Aiter.Next ( );  pb = Biter.Next ( );
  // pa 与 pb 是链表 ah、bh 的检测指针
  delete p;  //保留 ah 的表头结点，删去 bh 的表头结点
```

```
while ( Aiter.NotNull ( ) && Biter.NotNull ( ) )
 //两两比较
   switch ( compare ( pa->exp, pb->exp ) ) {
   case '=' :  // pa->exp == pb->exp
        pa->coef = pa->coef + pb->coef;  //系数相加
        p = pb;   pb = Biter.Next ( );  delete p;
        //删去原 pb 所指结点
        if ( !pa->coef ) {
           p = pa;  pa = Aiter.Next ( );
           delete p;  }  //相加为 0，不要
        else {
           pc->link = pa;  pc = pa;
           pa = Aiter.Next ( );  }  //相加不为 0，加入 ch 链
        break;
```

```
        case '>' :
        // pa->exp > pb->exp
              pc->link = pb;  pc = pb;
              pb = Biter.Next ( );   break;
        case '<' :
        // pa->exp < pb->exp
             pc->link = pa;  pc = pa;
             pa = Aiter.Next ( );
    }
    if ( Aiter.NotNull ( ) )  pc->link = pa;
    //剩余部分链入 ch 链
    else  pc->link = pb;
}
```

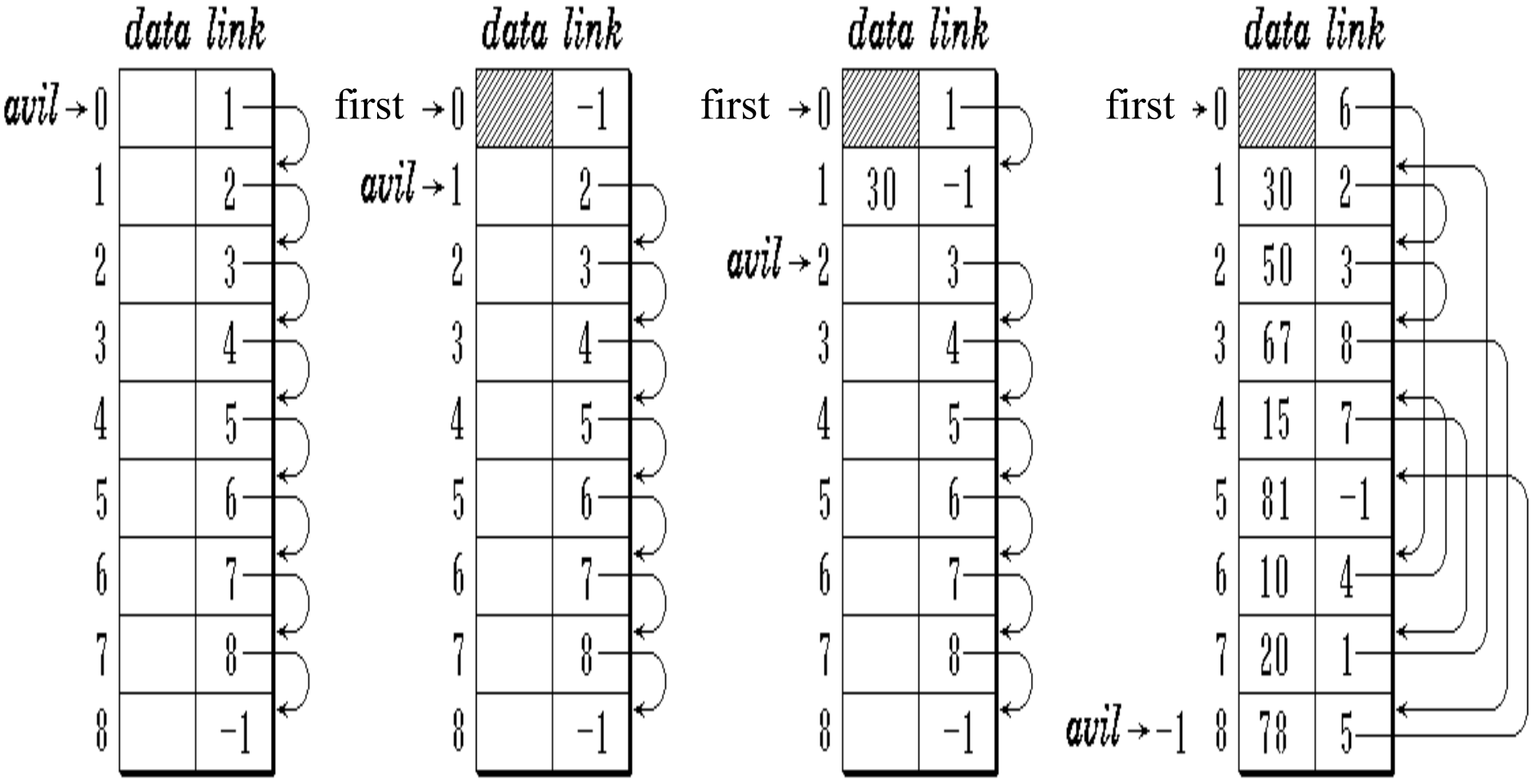
- 设两个多项式链表的长度分别为m与n，则总的比较次数为$O(m+n)$。

- 有了多项式的加法，则可容易地定义多项式的减法、乘法和除法。
  - 两个多项式的乘法可以转化为一系列多项式的加法来实现。

静态链表

- 为数组中每一个元素附加一个链接指针。
- 允许不改变各元素的物理位置，只要重新链接就能够改变这些元素的逻辑顺序。
- 利用数组定义，在整个运算过程中存储空间的大小不会变化——静态链表。
  - 每个结点由两个数据成员构成；
    - data域存储数据
    - link域存放链接指针
  - 所有结点形成一个结点数组；
  - 可带有表头结点。

静态链表结构


data link
avil → 0 1
1 2
2 3
3 4
4 5
5 6
6 7
7 8
8 -1
未用空间
data link
first → 0 -1
avil → 1 2
2 3
3 4
4 5
5 6
6 7
7 8
8 -1
链表初始化，创建头结点
data link
first → 0 1
1 30 -1
avil → 2 3
3 4
4 5
5 6
6 7
7 8
8 -1
插入30，链入链表中
data link
first → 0 6
1 30 2
2 50 3
3 67 8
4 15 7
5 81 -1
6 10 4
7 20 1
avil → -1 8 78 5
经过若干加删后的链表状态

- 需要分配一个结点时，从avil为头指针的链中摘下第一个结点分配出去，并用指针j指示；
- 头指针avil退到可利用空间表的第二个结点，即下一次可分配的结点地址。

  j = avil; avil = elem[avil].link;
- 需要释放由指针i指示的结点时，将其链入avil所指示的链中的最前端，并由avil指示。

  elem[i].link = avil; avil = i;
- 表头结点一般在第0个位置，其link指针指向链表中第一个结点。
- 最后一个结点的link指针为-1，表示链表终止。

静态链表类定义

```
const int maxSize = 100;  //静态链表大小
template <class Type> class StaticList;

template <class Type> class SLinkNode {
friend class StaticList <Type>;
private:
    Type data;  //结点数据
    int link;  //结点链接指针
};
```

```
template <class Type> class StaticList {
    SLinkNode <Type> elem[maxSize];
    int avil;  //当前可分配空间首地址
public:
    void InitList ( );
    int length ( );
    int Search ( Type x );
    int Locate ( int i );
    bool Append ( Type x );
    bool Insert ( int i, Type x);
    bool Remove ( int i );
    bool IsEmpty ( );
};
```

将链表空间初始化

```
template <class Type>
void StaticList <Type> :: InitList ( ) {
    elem[0].link = -1;
    avil = 1;  //当前可分配空间从 1 开始
               //建立带表头结点的空链表
    for ( int i = 1; i < maxSize-1; i++ )
        elem[i].link = i+1;  //构成空闲链接表
    elem[maxSize-1].link = -1;  //链表收尾
}
```

```
template <class Type>
int StaticList <Type> :: Length ( ) {
//计算静态链表的长度
   int p = elem[0].link;  int i = 0;
   while ( p != -1 ) {
        p = elem[p].link;  i++;   }
   return i;
}
template <class Type>
bool StaticList <Type> :: IsEmpty ( ) {
//判断表空否
   if ( elem[0].link == -1 )  return true;
   else return false;
}
```

在静态链表中查找具有给定值的结点

```
template <class Type>
int StaticList <Type> :: Search ( Type x ) {
    int p = elem[0].link;
    //指针 p 指向链表第一个结点
    while ( p != -1 )
        if ( elem[p].data == x) break;
        else p = elem[p].link;
    //逐个结点检测查找具有给定值的结点
    return p;
}
```

在静态链表中查找第 i 个结点

```
template <class Type>
int StaticList <Type> :: Locate ( int i ) {
    if ( i < 0 ) return -1;  //参数不合理
    if ( i == 0 ) return 0;
    int j = 1, p = elem[0].link;
    while ( p != -1 && j < i )
        { p = elem[p].link;  j++; }
        //循链查找第 i 号结点
    return p;
}
```

在静态链表的表尾追加一个新结点

```
template <class Type>
bool StaticList <Type> :: Append ( Type x ) {
    if ( avil == -1 ) return false;  //追加失败
    int q = avil;  //分配结点
    avil = elem[avil].link;
    elem[q].data = x;   elem[q].link = -1;
    int p = 0;  //查找表尾
    while (elem[p].link != -1) p = elem[p].link;
    elem[p].link = q;  //追加
    return true;
}
```

在静态链表第 i 个结点处插入新结点

```
template <class Type>
bool StaticList <Type> :: Insert ( int i, Type x ) {
    int p = Locate (i);
    if ( p == -1 ) return false;  //找不到结点
    int q = avil;  //分配结点
    avil = elem[avil].link;
    elem[q].data = x;
    elem[q].link = elem[p].link;  //链入
    elem[p].link = q;
    return true;
}
```

在静态链表中释放第 i 个结点

```
template <class Type>
bool StaticList <Type> :: Remove ( int i ) {
    int p = Locate (i-1);
    if ( p == -1 ) return false;  //找不到结点
    int q = elem[p].link;  //第 i 号结点
    elem[p].link = elem[q].link;
    elem[q].link = avil;  //释放
    avil = q;
    return true;
}
```

本章小结

- 知识点
  - 线性表
  - 顺序表及其三种重要操作
  - 单链表、带表头结点的单链表
  - 循环链表、双向链表与静态链表
  - 多项式的应用

- **课程习题**

  - 笔做题——2.6, 2.11, 2.12, 2.16, 2.17, 2.19
    （以作业形式提交）
  - 上机题——2.10, 2.15, 2.28
  - 思考题——2.2, 2.4, 2.5, 2.18, 2.24