第二章　线性表

- 线性表
- 顺序表
- 单链表
- 循环链表
- 双向链表
- 多项式

线性表的物理实现

单链表的变化形式

链表的应用

第二章　线性表

§2.1 线性表

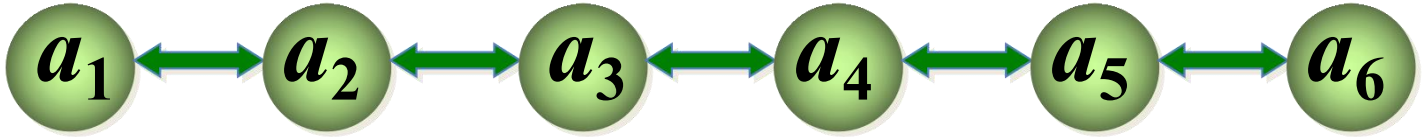
- 定义

  n（≥ 0）个表项的有限序列

  $$\mathbf{L}=(a_1, a_2, \ldots, a_n)$$

  - a_i是表项，n是表长度。
  - 第一个表项是表头，最后一个是表尾

- 线性表的特点
  - 为简单起见，假定表中元素的数据类型相同
  - 线性表中，结点和结点间的关系是一对一的
  - 有序表和无序表

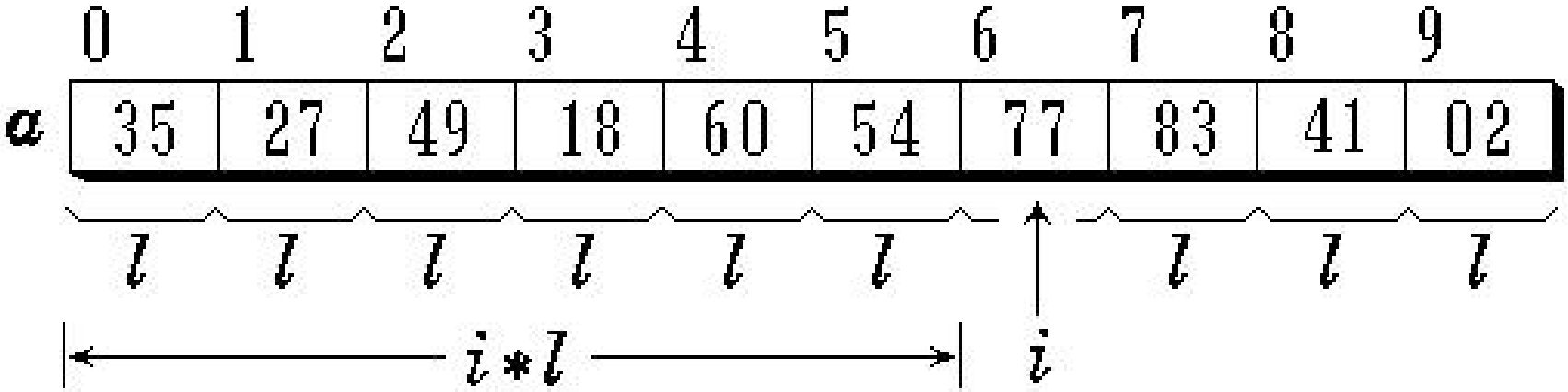


- 线性表的存储方式
  - 顺序存储方式 —— 顺序表
  - 链表存储方式 —— 链表

§2.2 顺序表

- 顺序表可用一维数组实现

$$LOC(i)=\begin{cases}\alpha, & i=0 \text{ 时}\\ LOC(i-1)+l, & i>0 \text{ 时}\end{cases}$$



$$LOC(i)=LOC(i-1)+l=\alpha+i*l$$

顺序表的定义

- 将线性表中的元素相继存放在一个连续的存储空间中

顺序表的特点

- 各表项的逻辑顺序与物理顺序一致
- 对各个表项可以顺序访问，也可以随机访问

结点的变体（异质数据）

25	‘s’	3.3	62	74	‘t’	1.0	‘6’

- 若想在线性表中存放不同类型的数据，可采用等价定义union：

```
typedef union {
    int val;              //按data.val引用
    char ch;              //按data.ch引用
    float dir;            //按data.dir引用
    union data *link;     //按data.link引用
} data;                   //整体上是同一类型data
```

结点的变体（异质数据）

- 实际使用中，可以增加一个枚举类型，提前判定数据对应的类型

```
enum ValueType{INT_, CHAR_, FLOAT_, LINK_};

struct TaggedValue{
    enumValueType type;
    typedef union {
            int val;                  //按data.val引用
             char ch;                 //按data.ch引用
            float dir;                //按data.dir引用
            union data *link;    //按data.link引用
        } data;              //整体上是同一类型data
    };
```

顺序表(SeqList)类的定义

```
#include <iostream.h>          //定义在“seqList.h”中
#include <stdlib.h>
#include “linearList.h"
const int defaultSize = 100;
template < class E>
class SeqList: public LinearList<E> {
  protected:
    E *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
    void reSize(int newSize);  //改变数组空间大小
```

```
public:
   SeqList(int sz = defaultSize);          //构造函数
   SeqList(SeqList<E>& L);                 //复制构造函数
   ～SeqList() {delete[ ] data;}           //析构函数
   int Size() const {return maxSize;}      //求表最大容量
   int Length() const {return last+1;}     //计算表长度
   int Search(E& 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
   bool getData(int i, E& x) const;  //取第i个表项的值
   bool Insert(int i, E x);                //插入
   bool Remove(int i, E& x);               //删除
};
```

顺序表的构造函数

```
#include <stdlib.h>      //操作“exit”存放在此
#include “seqList.h”     //操作实现放在“seqList.cpp”
template <class E>
SeqList<E>::SeqList(int sz) {
    if (sz > 0) {
        maxSize = sz;  last = -1;
        data = new E[maxSize];     //创建表存储数组
        if (data == NULL){   //动态分配失败
            cerr << "存储分配错误！ " << endl;
            exit(1);
        }
   }
};
```

复制构造函数

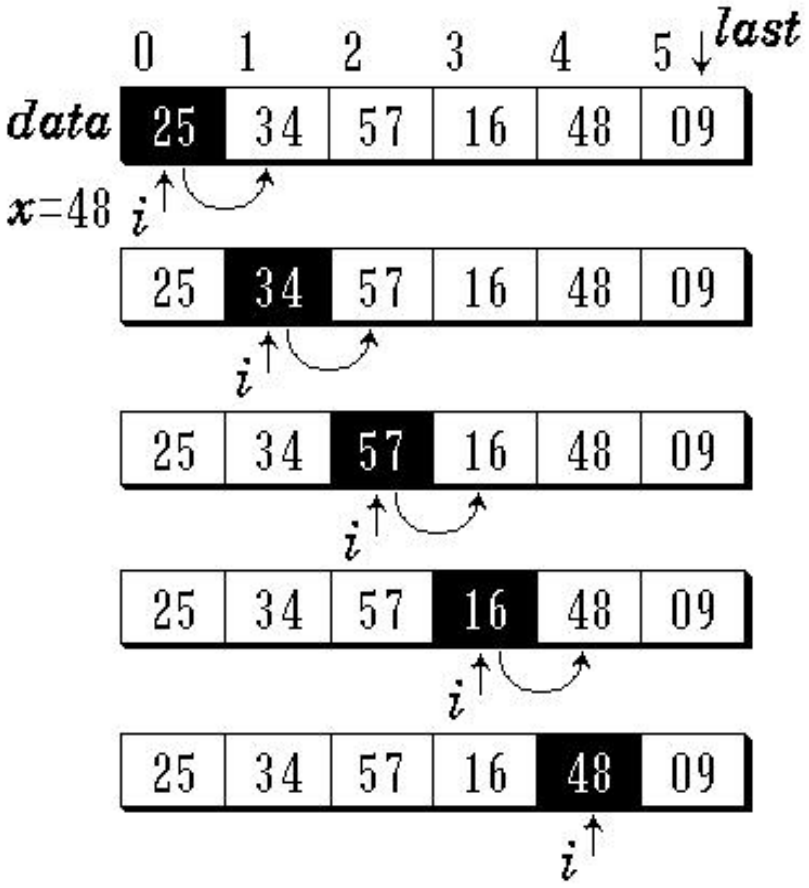
```
template <class E>
SeqList<E>::SeqList ( SeqList<E>& L ) {
    maxSize = L.Size();   last = L.Length()-1;
    E value;
    data = new E[maxSize];          //创建存储数组
    if (data == NULL)               //动态分配失败
        {cerr << "存储分配错误！ " << endl;     exit(1);}
    for (int i = 1; i <= last+1; i++)    //传送各个表项
        {L.getData(i, value); data[i-1] = value;}//该处复制整个数组元素，注意不要用浅复制
};
```

顺序表的搜索算法

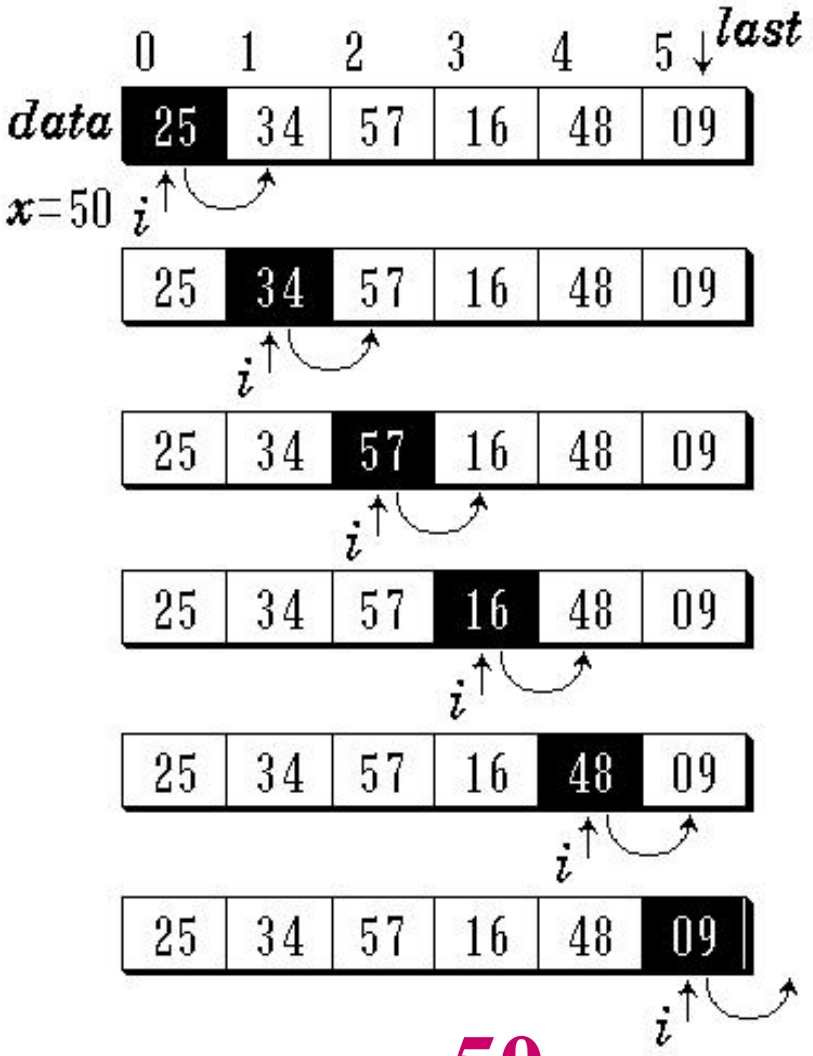
```
template <class E>
int SeqList<E>::Search(E & x) const {
//在表中顺序搜索与给定值 x 匹配的表项，找到则
//函数返回该表项是第几个元素，否则函数返回0
    for (int i = 0; i <= last; i++)      //顺序搜索
        if ( data[i] == x )
            return i+1; //表项序号和表项位置差1，逻
  辑序号和数组下标不可混用
    return 0;                            //搜索失败
};
```

在等概率条件下，成功查找的平均比较次数是？

顺序表的搜索（查找）



$x = 48$



$x = 50$

顺序查找数据的时间代价（比较次数分析）

ACN(Average Comparing Number)

(假设表的长度为n,即$n = last + 1$)

搜索成功：表项i的查找概率p_i，比较次数c_i

平均比较次数 $ACN = \sum_{i=1}^{n} p_i * c_i$

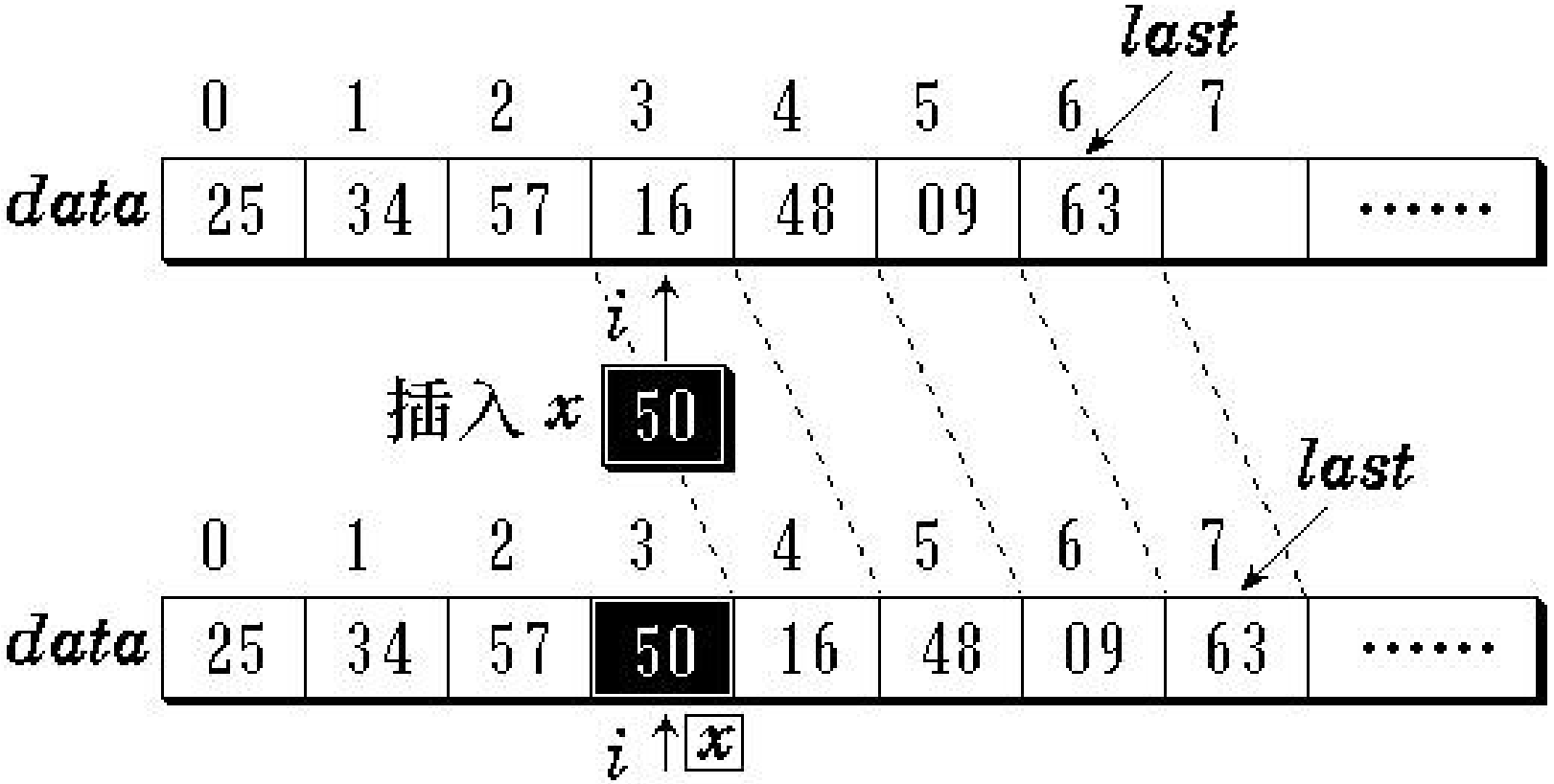
若搜索概率p_i相等，则

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

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

搜索不成功：数据比较n次

顺序表的插入


last
0 1 2 3 4 5 6 7
data
25 34 57 16 48 09 63 ……
i
插入 x
50
last
0 1 2 3 4 5 6 7
data
25 34 57 50 16 48 09 63 ……
i
x

表项的插入算法

```
template <class E>
bool SeqList<E>::Insert (int i, E x) {
//将新元素x插入到表中第i (1≤i≤last+2) 个表项位
//置。即在第i个之前插入
    if (last == maxSize-1) return false;        //表满
    if (i < 1 || i > last+2) return false;  //参数i不合理
    for (int j = last; j >= i-1; j--)      //从后往前，依
  次后移
        data[j+1] = data[j];
    data[i-1] = x;        //插入(第 i 表项在data[i-1]处)
    last++;
    return true;                          //插入成功
};
```

顺序表插入的时间代价（移动次数）

在表中第 i 个位置插入，从data[i-1] 到data [last] 成块后移，移动n-i+1项 (假设表的长度为n,即n = last + 1)

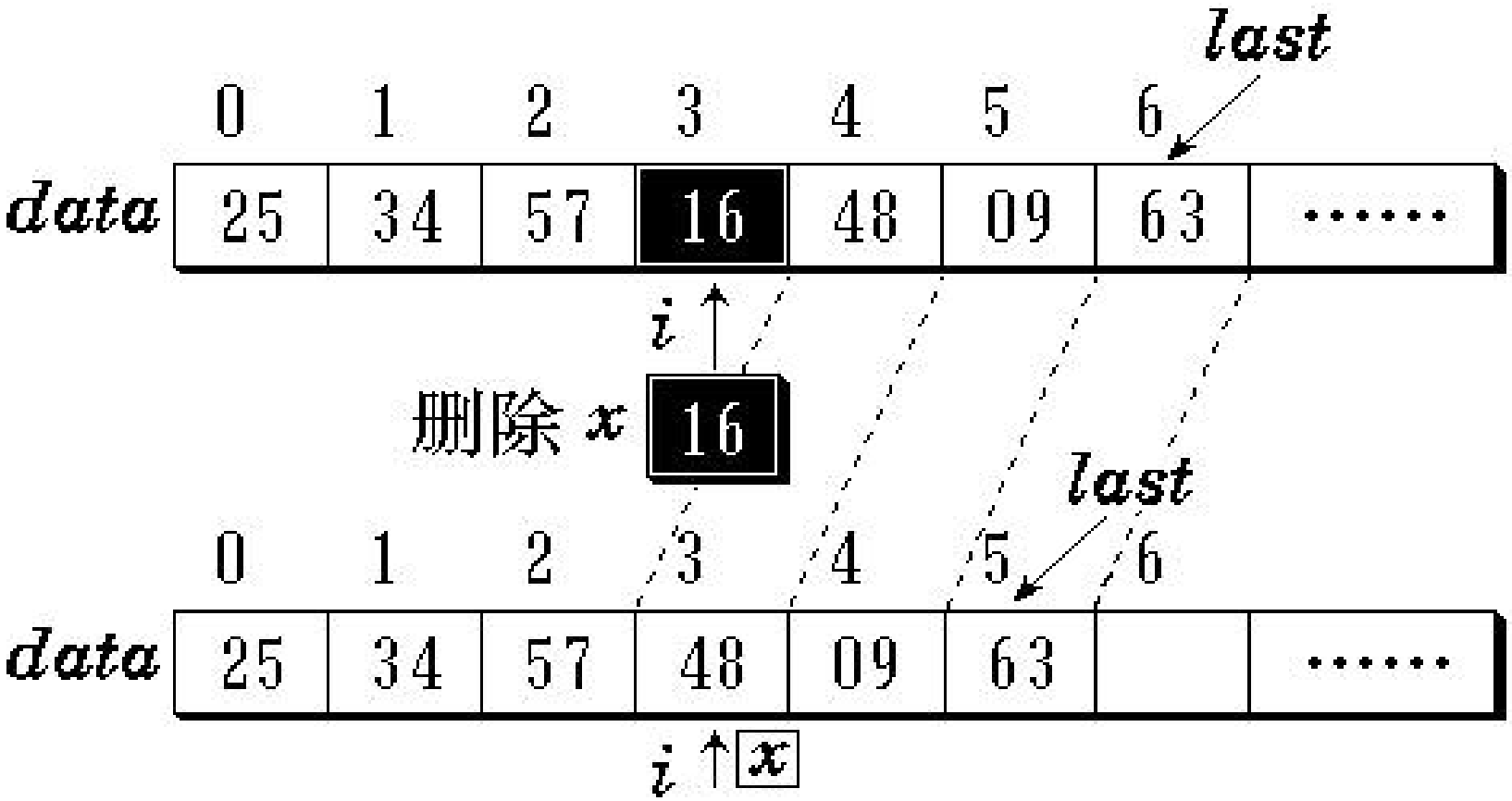
平均数据移动次数*AMN*（*Average Moving Number*）在各表项插入概率相等时为

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

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

在插入时有n+1个插入位置，平均移动n/2项

顺序表的表项删除


last
0 1 2 3 4 5 6
data
25 34 57 16 48 09 63 ……
i
删除 x
16
last
0 1 2 3 4 5 6
data
25 34 57 48 09 63 ……
i
x

表项的删除算法

```
template <class E>
bool SeqList<E>::Remove (int i, E& x) {
//从表中删除第 i (1≤i≤last+1) 个表项，通过引用型
//参数 x 返回被删元素。
    if (last == -1) return false;                    //表空
    if (i < 1 || i > last+1) return false;//参数i不合理
    x = data[i-1];
    for (int j = i; j <= last; j++)      //从前往后，依次
  前移填补
        data[j-1] = data[j];
    last--;
    return true;
};
```

顺序表表项删除的时间代价(移动次数)

删除第 *i* 个表项，需将第 *i*+1 项到第 *last+1*项全部前移，需前移的项数为 *n-i* (假设表的长度为n,即n = last + 1)

平均数据移动次数*AMN*（*Average Moving Number*）在*n*个表项删除概率相等时为

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

在删除时有n个删除位置，平均移动(n-1)/2项

在表头插入/删除最慢，在表尾插入/删除最快

顺序表的应用：用顺序表实现集合的“并”运算

```
void Union ( SeqList<int> & LA,
                          SeqList<int> & LB ) {
    int n = LA.Length ( );
    int m = LB.Length ( );
    int x;
    for ( int i = 1; i <= m; i++ ) {
        LB.getData(i, x);     //在LB中取一元素
        int k = LA.Search (x);      //在LA中搜索它
        if ( k == 0 )                  //若未找到插入它
          { LA.Insert (n, x);  n++; }
    }
}
```

用顺序表实现集合的“交”运算

```
void Intersection ( SeqList<int> & LA,
                    SeqList<int> & LB ) {
    int n = LA.Length ( );
    int m = LB.Length ( );  int i = 1; int x;
    while ( i <= n ) {
        LA.getData (i, x);    //在LA中取一元素
        int k = LB.Search (x);     //在LB中搜索它
        if ( k == 0 ) { LA.Remove (i,x);  n--;}
                          //未找到,在LA中删除它
        else i++;
    }
}
```

若线性表有序，如何优化并与交的运算？

某顺序表使用数组 data[] 存储，表项序号从1开始，数组下标从0开始，`last` 指向最后一个有效元素,当前状态为：

data = [12, 25, 31, 48, 60, _, _, _]

last = 4

maxSize = 8

依次执行以下操作：

Insert(3, 40); // 在第3个表项之前插入40

Remove(5, x); // 删除当前第5个表项，并通过x返回其值

1. 写出每次操作完成后的有效元素序列和 `last` 的值。
2. 写出 `x` 的值，并分别计算插入、删除时移动的元素个数。
3. 写出两个操作执行前至少需要检查的边界条件，并说明它们的最坏时间复杂度。

执行 `Insert(3, 40)` 后：

data = [12, 25, 40, 31, 48, 60, _, _]; last = 5

需要移动 `n-i+1=3` 个元素，即 `31、48、60`。

执行 `Remove(5, x)` 后，删除的是当前第5个表项 `48`：

x = 48; data = [12, 25, 40, 31, 60, _, _, _]; last = 4

需要移动 `n-i=1` 个元素，即将 `60` 前移。

- 插入前检查：表是否已满，即 `last == maxSize - 1`；位置是否合法，即 `1 <= i <= last + 2`。
- 删除前检查：表是否为空，即 `last == -1`；位置是否合法，即 `1 <= i <= last + 1`。
- 插入和删除的最坏时间复杂度均为 `O(n)`；在表尾插入或删除时不需要移动元素，移动部分可视为 `O(1)`。

§2.3 单链表

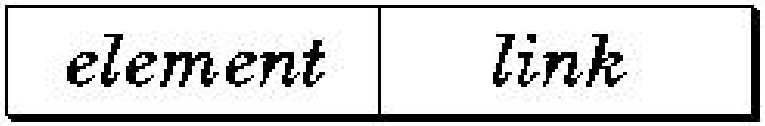
连续存储方式（顺序表）

- 特点：存储利用率高，存取速度快
- 缺点：插入、删除等操作时需要移动大量数据

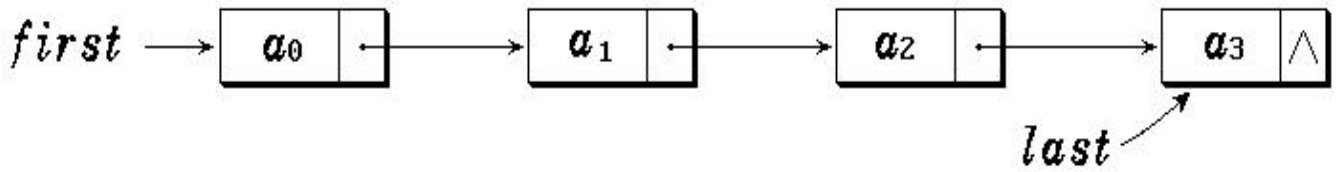
链式存储方式（链表）

- 特点：适应表的动态增长和删除
- 缺点：需要额外的指针存储空间

- 单链表的特点
  - 每个元素(表项)由结点(*Node*)构成。

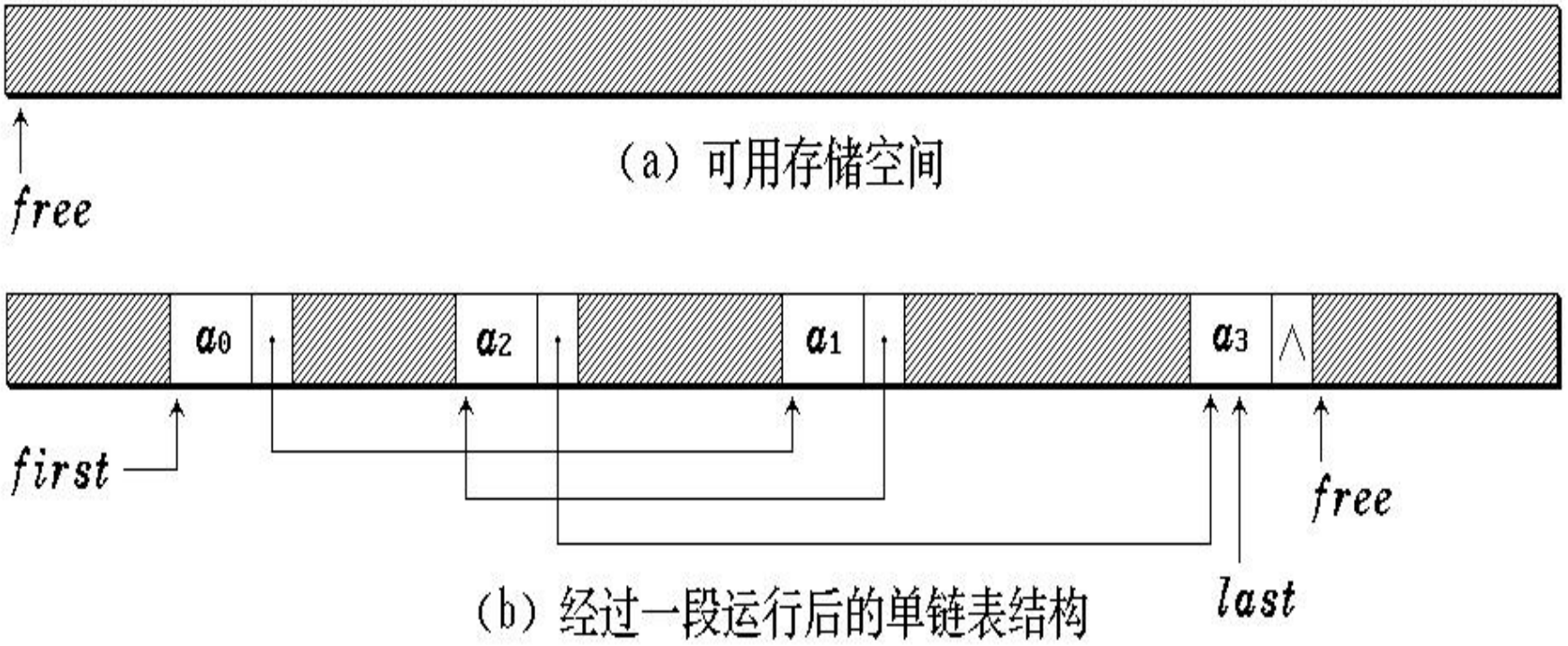


  - 线性结构



  - 结点可以不连续存储
  - 表长度可方便地扩充

单链表的存储映像



如何访问第i个元素？

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

- 定义方式
  - 结构方式
  - 复合方式
  - 嵌套方式
  - 继承方式

链表类定义（结构方式）

```
struct ListNode {                  //链表结点结构
    int data;                      //结点数据，整型
    ListNode * link;               //结点指针
};
```

```
class List {                       //链表类
  public:
    …
  private:
    ListNode *first ;              //表头指针
};
```

链表类定义（复合方式）

```
class List;                          //复合方式

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
private:
    ListNode *first ;                //表头指针
};
```

链表类定义（嵌套方式）

```
class List {                          //链表类
 public:
      …

 private:
   class ListNode {                   //链表结点类
    public:
       int data;                      //结点数据, 整型
       ListNode * link;               //结点指针
};

   ListNode *first ;                  //表头指针
};
```

链表类定义（继承方式）

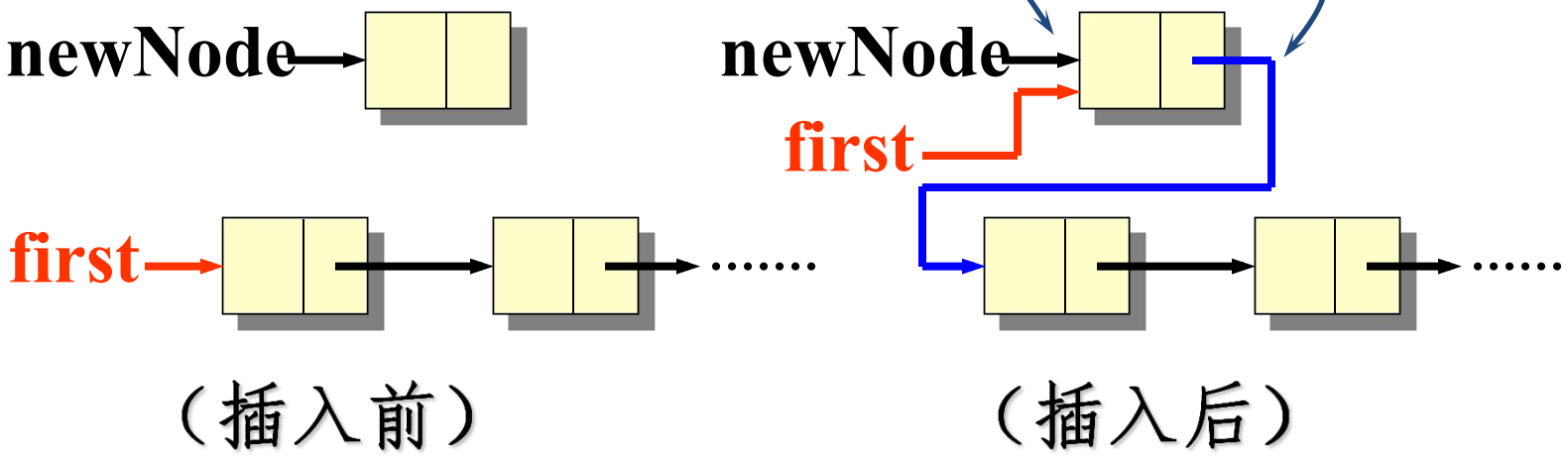
```
class ListNode {                    //链表结点类
  protected:
    int data;                       //结点数据, 整型
    ListNode * link;                //结点指针
};
```

```
class List: pubic class LinkNode {          //链表类
  public:
    …
  private:
    ListNode *first ;                //表头指针
};
```

单链表中的插入与删除操作

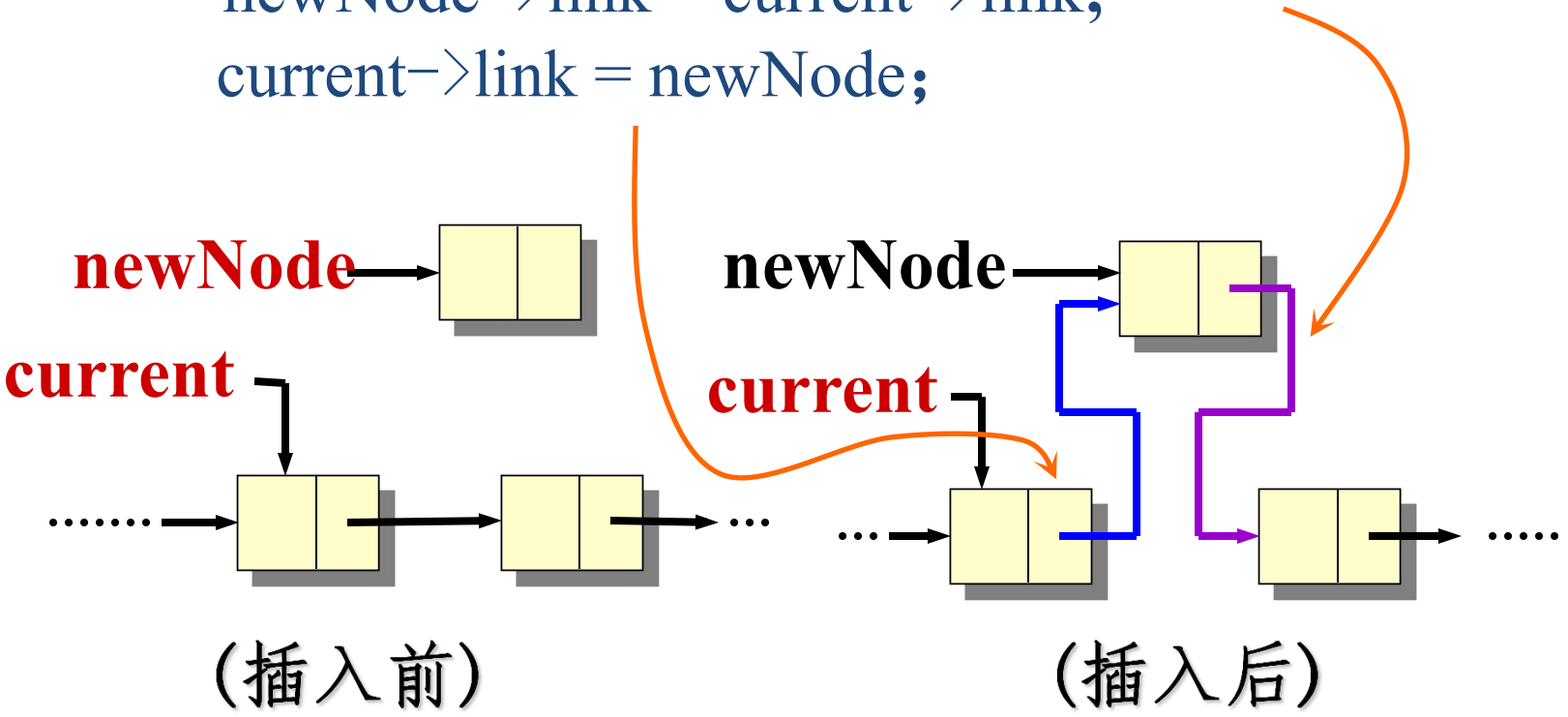
- 插入
  - 第一种情况：在链表最前端插入

```
newNode->link = first ;    //first存的是第一个元素的地址
first = newNode;
```



- 第二种情况：在链表中间插入

```
newNode->link = current->link;
current->link = newNode;
```

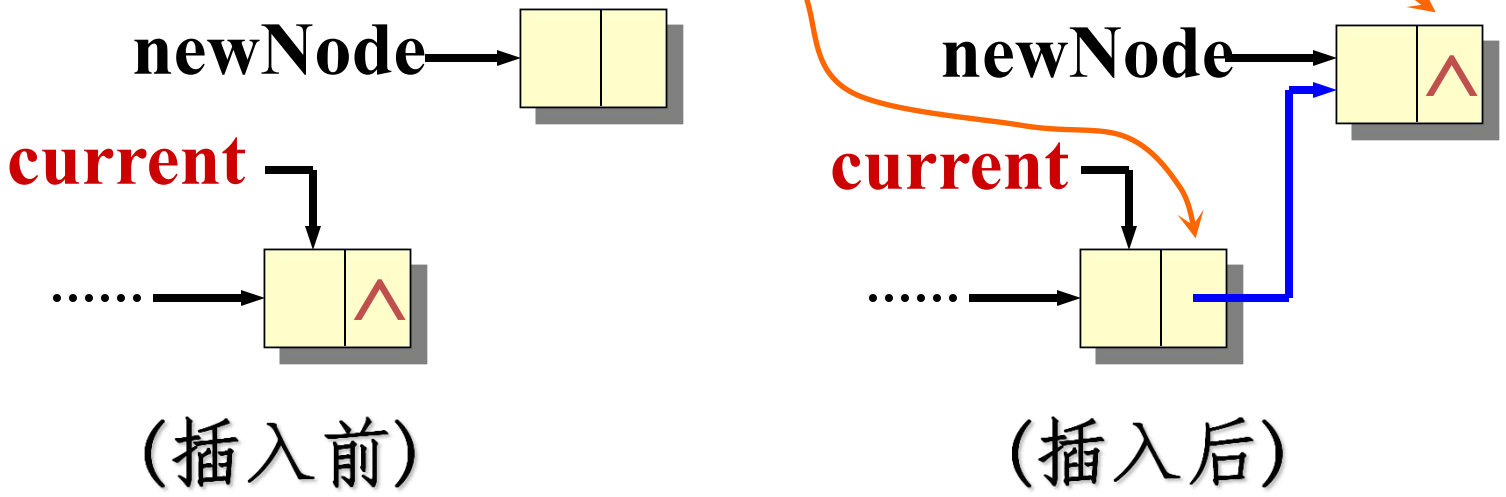


赋值顺序一定不能写反

- 第三种情况：在链表末尾插入

  newNode->link = current->link;

  current->link = newNode;



newNode->link默认是NULL

单链表的插入算法

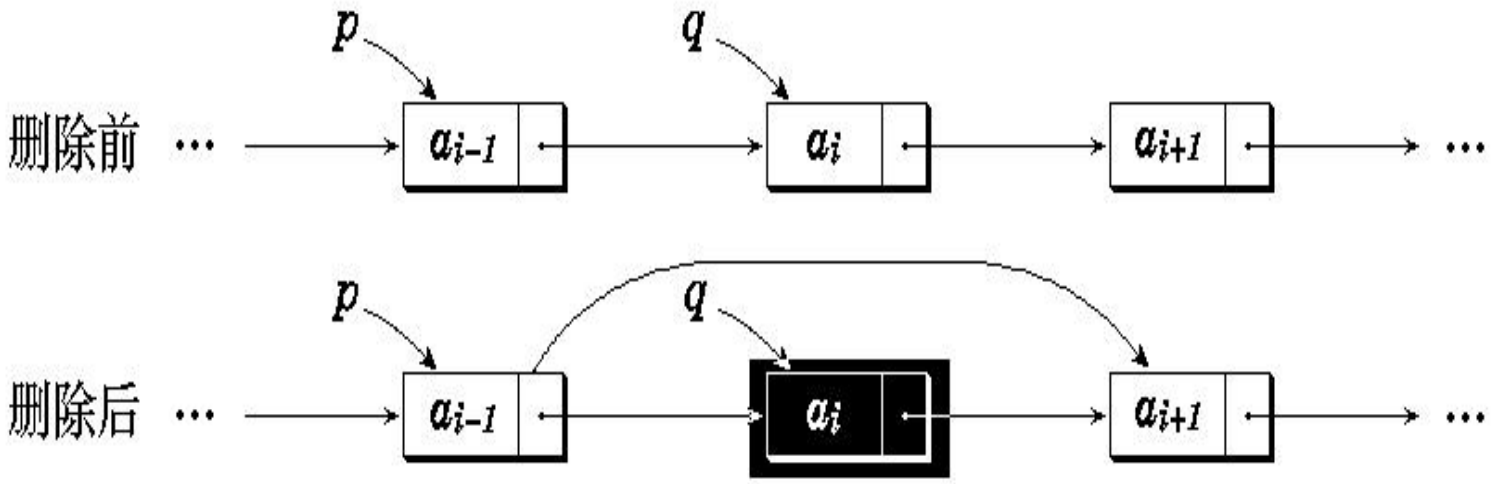
```
bool List::Insert(int i, int x) {
//将新元素 x 插入到第 i 个结点之后。i 从1开始，
//i = 0 表示插入到首元结点之前。
   if (first == NULL || i == 0) {    //空表或首元结点前
      LinkNode *newNode = new LinkNode(x);
         //建立一个新结点
      newNode->link = first;  first = newNode;
         //新结点成为首元结点
   }
   else {                    //否则，寻找插入位置
      LinkNode *current = first;   int k = 1;     //指向第1
   个位置
```

```
        while (k < i && current != NULL)    //找第i结点
            { current = current->link;  k++; }//指向第i个
        if (current == NULL && first != NULL)    //链短
            {cerr << "无效的插入位置!\n";  return false;}
        else {                  //插入在链表的中间
            LinkNode *newNode = new LinkNode(x);
            newNode->link = current->link;
            current->link = newNode;
        }
    }
    return true;
};
```

顺序表在第i个位置插入

单链表在第i个结点后插入

- 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



在单链表中删除第i个结点a_i

单链表的删除算法

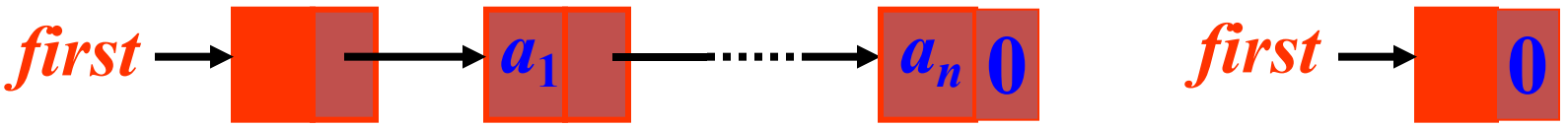
```
bool List::Remove (int i, int& x) {
//将链表中的第 i 个元素删去, i 从1开始。
    LinkNode *del;              //暂存删除结点指针
    if (i <= 1)  { del = first;  first = first->link; }
    else {
        LinkNode *current = first;  k = 1;   //找i-1号结点
        while (k < i-1 && current != NULL)
            { current = current->link;   k++; }//指向第i-1个
        if (current == NULL || current->link == NULL) {
            cout << "无效的删除位置!\n";  return false;
        }
```

```
        del = current->link;          //删中间/尾结点
        current->link = del->link;
    }
    x = del->data; delete del;   //取出被删结点数据，注意最后删除节点
    return true;
};
```

- 实现单链表的插入和删除算法，不需要移动元素，只需修改结点指针，比顺序表方便。
- 情况复杂，要专门讨论空表和在表头插入的特殊情形。
- 寻找插入或删除位置只能沿着链顺序检测。

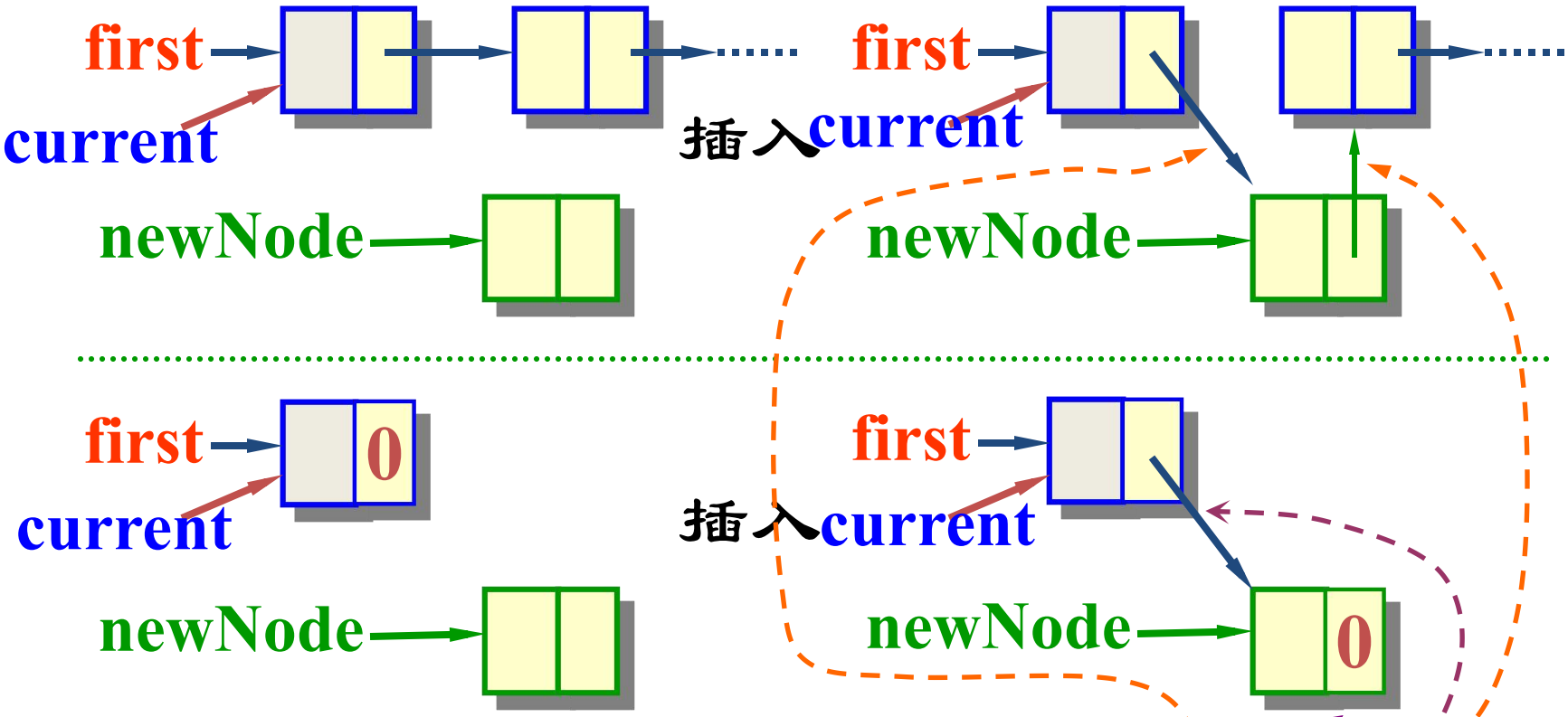
带附加头结点（表头结点）的单链表

- 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。



非空表　　空表

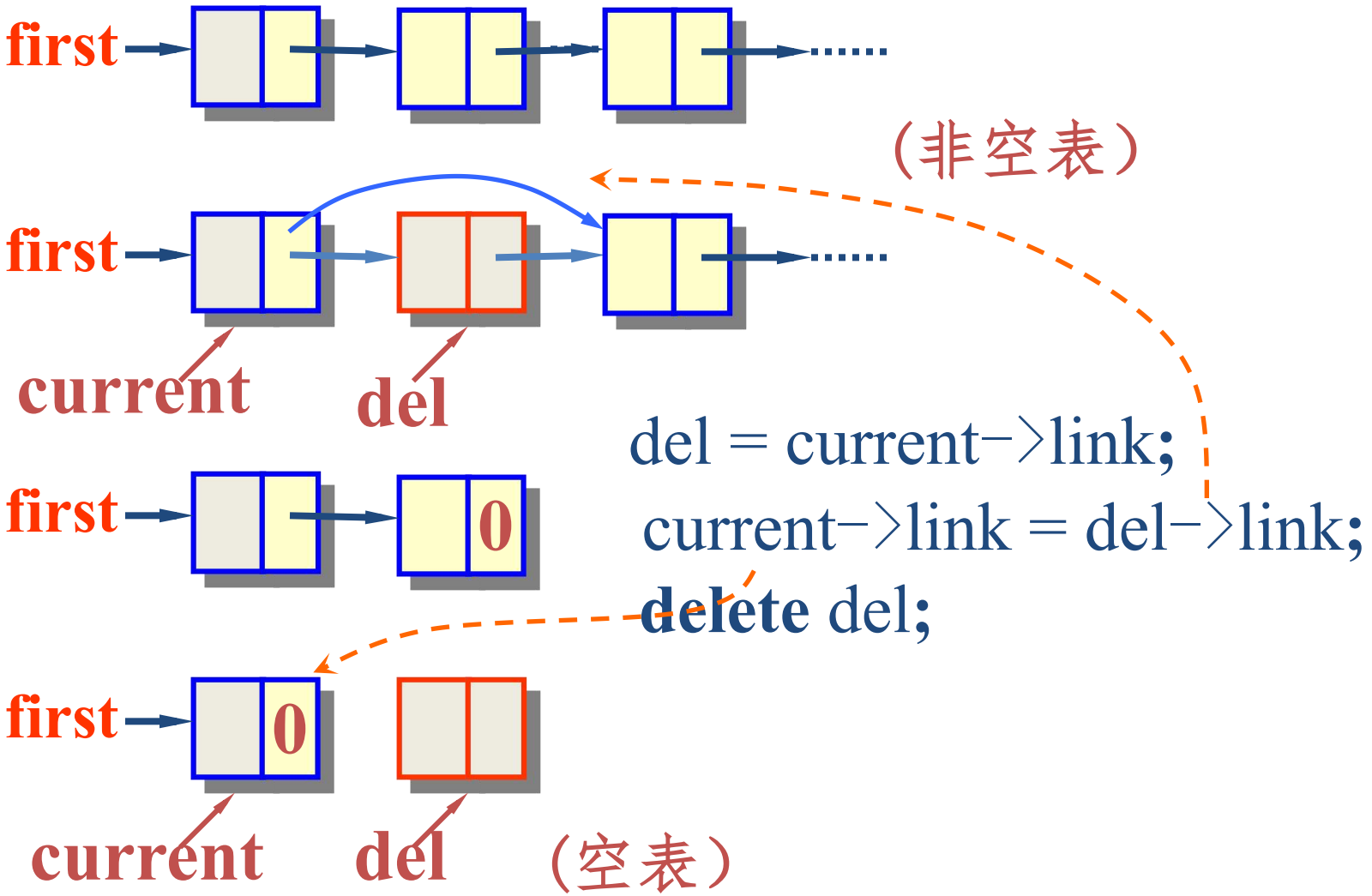
在带表头结点的单链表最前端插入新结点



newNode->link = current->link;

current->link = newNode;

从带表头结点的单链表中删除最前端的结点



一个不带头结点的单链表当前为：

first → A → B → C → NULL

已知 `current` 指向结点 `B`，`newNode` 指向已经创建、数据域为 `X` 的新结点。现在依次完成：

1. 将 `X` 插入到 `B` 之后；
2. 删除链表的第一个数据结点。

1. 写出上述两个操作的核心指针语句，并画出最终链表。
2. 插入时的两条赋值语句能否交换顺序？说明原因。
3. 若采用带头结点的单链表，删除第一个数据结点时应如何改写核心语句？这种设计简化了什么边界情况？
4. 在已经获得 `current` 或待删结点前驱的情况下，插入和删除的时间复杂度是多少？若还需要从表头定位操作位置，总时间复杂度又是多少？

插入：

```
newNode->link = current->link;
current->link = newNode;
```

first → A → B → X → C → NULL

删除：

```
LinkNode *del = first;
first = first->link;
delete del;
```

first → B → X → C → NULL

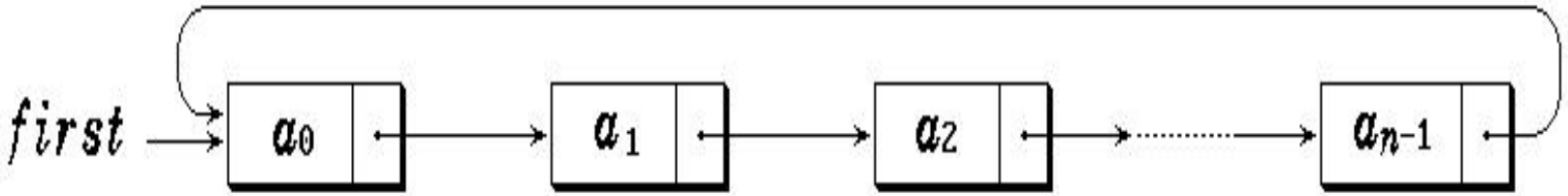
```
LinkNode *h = first;
LinkNode *del = h->link;
h->link = del->link;
delete del;
```

- 已经获得插入位置或待删结点前驱时，修改指针本身为 `O(1)`。
- 若需要从表头定位第 `i` 个位置，定位为 `O(n)`，因此整个操作为 `O(n)`。

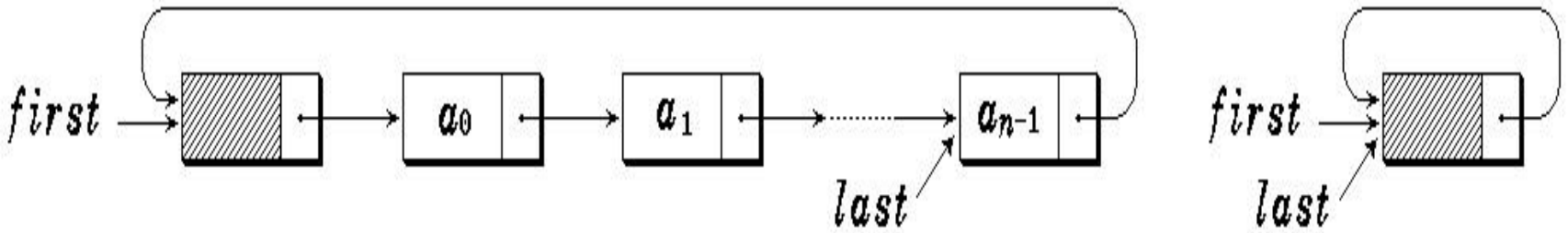
§2.4 线性链表的其他变形

- 循环链表
- 双向链表

循环链表 (*Circular List*)



循环链表



带表头结点的循环链表

如何判断表为空？？

循环链表的特点：

- 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。

- 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。

循环链表类的定义

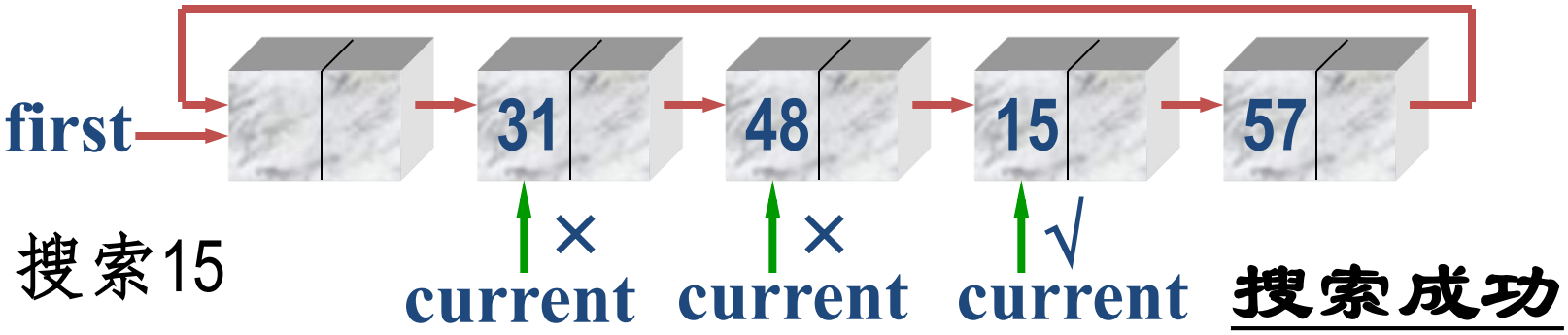
```
template <class E>
struct CircLinkNode {                    //链表结点类定义
    E data;
    CircLinkNode<E> *link;
    CircLinkNode ( CircLinkNode<E> *next =
          NULL ) { link = next; }
    CircLinkNode ( E x, CircLinkNode<E> *next =
          NULL ) { data = x;  link = next; }
    bool Operator==(CircLinkNode<E> & node) {
        return data == node.data; }
    bool Operator!=(CircLinkNode<E> & node) {
        return data != node.data; }
};
```

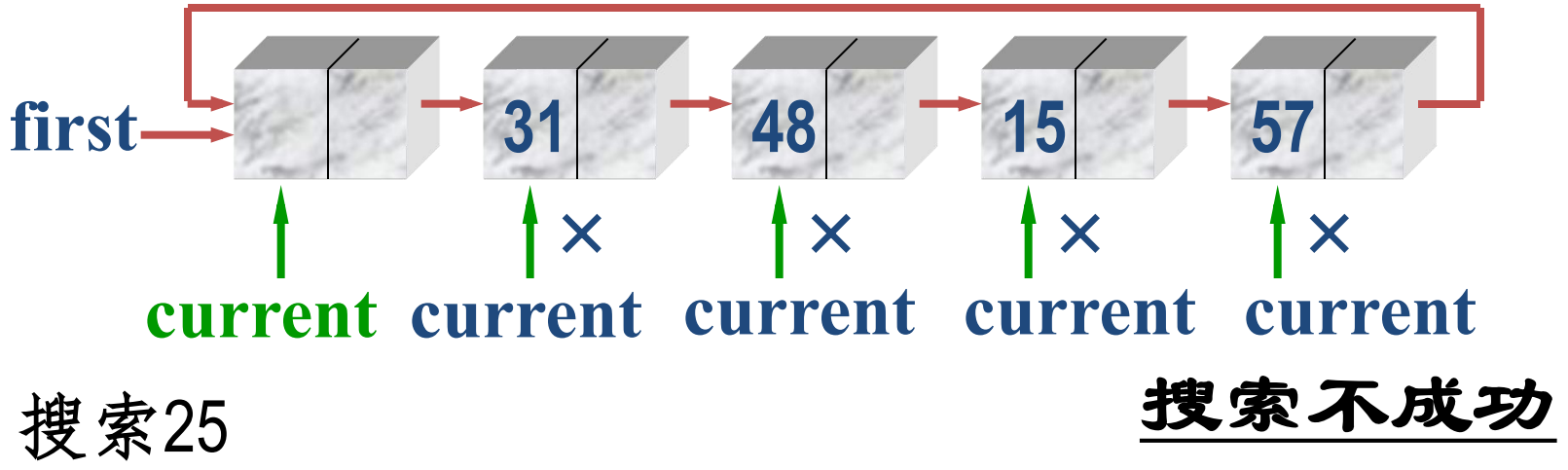
```
template <class E>      //链表类定义
class CircList : public LinearList<E> {
private:
   CircLinkNode<E> *first, *last;  //头指针, 尾指针
public:
   CircList(const E x);                    //构造函数
   CircList(CircList<E>& L);               //复制构造函数
   ~CircList();                            //析构函数
   int Length() const;                     //计算链表长度
   bool IsEmpty() { return first->link == first; }
                                           //判表空否
   CircLinkNode<E> *getHead() const;
                                           //返回表头结点地址
```

```
    void setHead ( CircLinkNode<E> *p );
                                        //设置表头结点地址
    CircLinkNode<E> *Search ( E x ); //搜索
    CircLinkNode<E> *Locate ( int i );//定位
    E *getData ( int i );                       //提取
    void setData ( int i, E x );                //修改
    bool Insert ( int i, E x );                 //插入
    bool Remove ( int i, E& x);                 //删除
};
```

- 循环链表与单链表的操作实现，最主要的不同就是扫描到链尾，遇到的不是**NULL**，而是表头。

循环链表的搜索算法


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

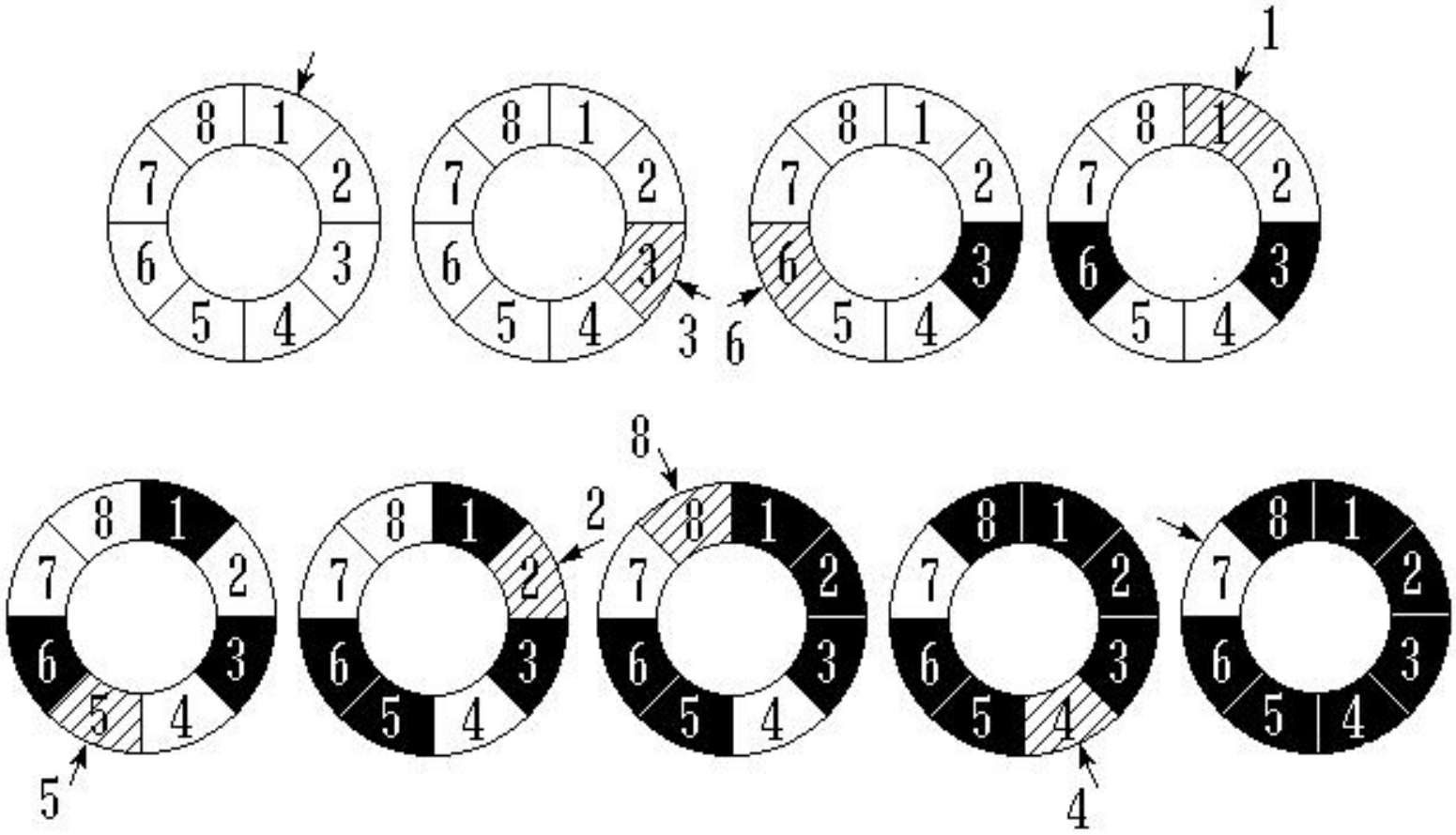

first
31
48
15
57
current
current
current
current
current
搜索25
搜索不成功

循环链表的搜索算法

```
template <class E>
CircListNode<E> * CircList<E>::Search( E x )
{
//在链表中从头搜索其数据值为 x 的结点
    current = first->link;//已经指向第1个非空元素
    while ( current != first && current->data != x )
        current = current->link;
    return current;
}
```

循环链表示例：求解约瑟夫问题

- 约瑟夫问题（例：$n=8$ $m=3$）



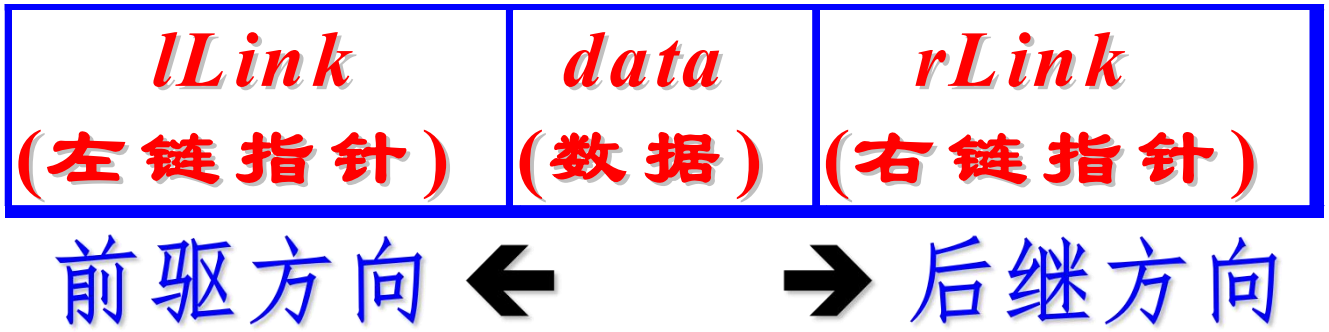
求解Josephus问题的算法

```
#include <iostream.h>
#include "CircList.h"
template <class E>
void Josephus(CircList<E>& Js, int n, int m) {
    CircLinkNode<E> *p = Js.getHead(),
                    *pre = NULL;
    int i, j;
    for ( i = 0; i < n-1; i++ ) {              //执行n-1次
        for ( j = 1; j < m; j++)               //数m-1个人
            { pre = p;  p = p->link; }//pre是前一个！
        cout << "出列的人是"  << p->data << endl;
```

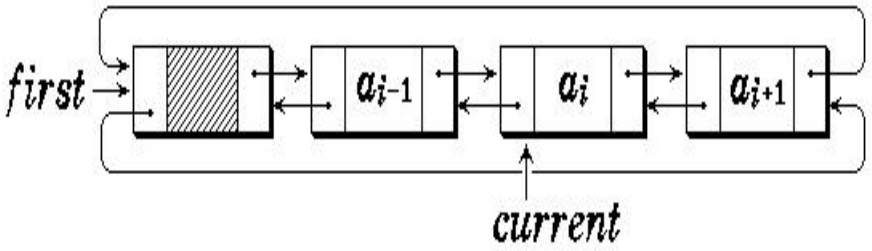
```
        pre->link = p->link;  delete p;          //删去
        p = pre->link;       //p再次变为报数第1个
    }
};
void main() {
    CircList< int> clist;
    int i, n, m;
    cout <<"输入游戏者人数和报数间隔：";
    cin >> n >> m;
    for (i = 1; i <= n; i++ ) clist.insert(i, i);   //约瑟夫环
    Josephus(clist, n, m);                  //解决约瑟夫问题
}
```

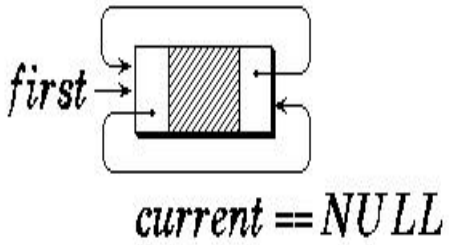
双向链表（doubly linked list）

- 每个结点的结构



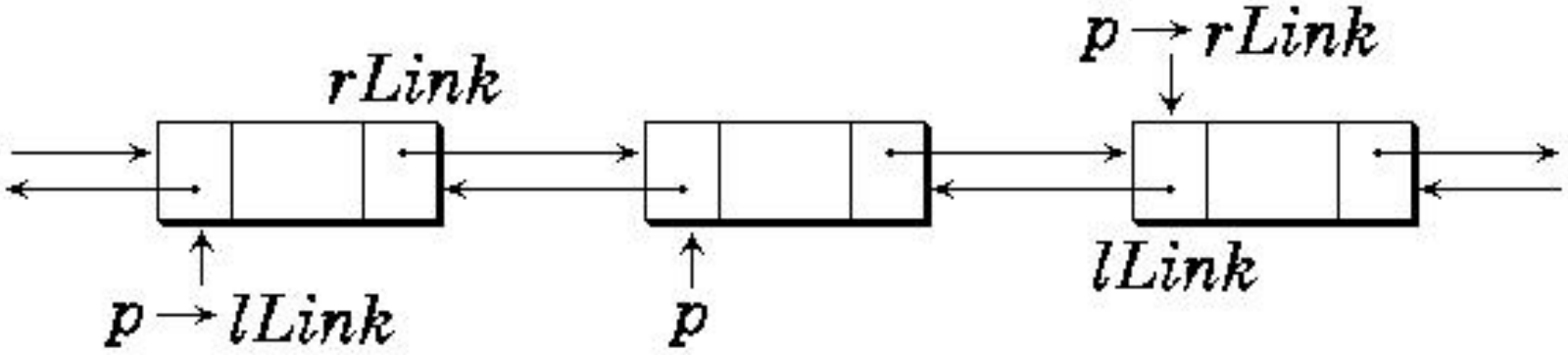
- 带头结点的双向链表表示





- 链表中的结点指针关系

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



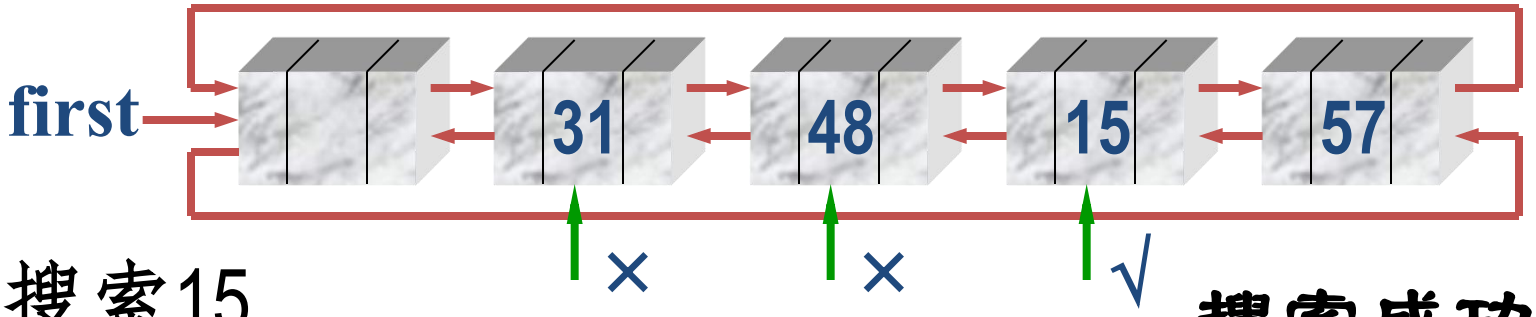
双向循环链表类的定义

```
template <class E>
struct DblNode {                    //链表结点类定义
    E data;                         //链表结点数据
    DblNode<E> *lLink, *rLink;//前驱、后继指针
    DblNode ( DblNode<E> *l = NULL,
    DblNode<E> *r = NULL )
       { lLink = l;  rLink = r; }          //构造函数
    DblNode ( E value, DblNode< E> *l = NULL,
    DblNode<E> *r = NULL)
      { data = value;  lLink = l;  rLink = r; }  //构造函数
};
```

```
template <class E>
class DblList {                    //链表类定义
   public:
      DblList ( E uniqueVal ) {          //构造函数
         first = new DblNode<E> (uniqueVal);
         first->rLink = first->lLink = first;
      };
      DblNode<E> *getFirst () const { return first; }
      void setFirst ( DblNode<E> *ptr ) { first = ptr; }
      DblNode<E> *Search ( E x, int d);
     //在链表中按d指示方向寻找等于给定值x的结点,
     //d=0按前驱方向,d≠0按后继方向
```

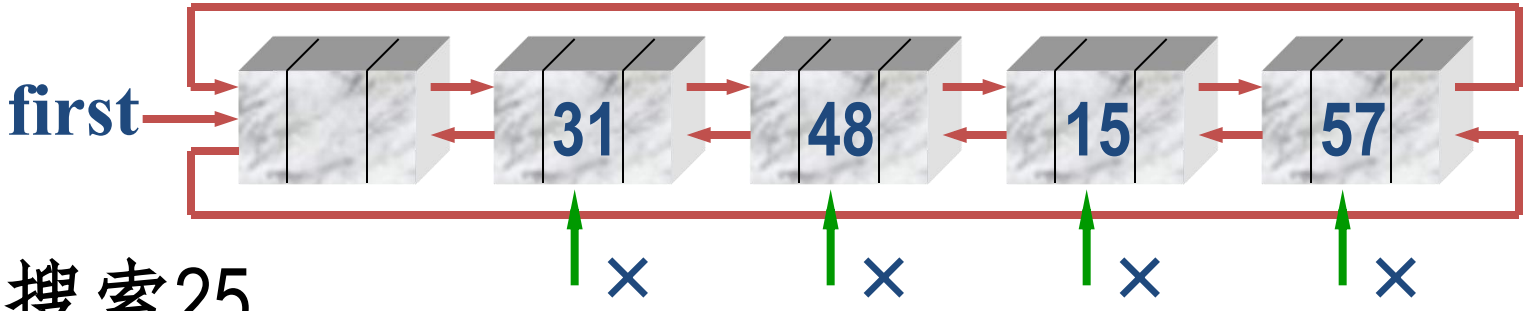
```
  DblNode<E> *Locate ( int i, int d );
 //在链表中定位序号为i(≥0)的结点, d=0按前驱方
 //向,d≠0按后继方向
  bool Insert ( int i, E x, int d );
 //在第i个结点后插入一个包含有值x的新结点,d=0
 //按前驱方向,d≠0按后继方向
  bool Remove ( int i, E& x, int d );   //删除第i个结点
  bool IsEmpty() { return first->rlink == first; }
 //判双链表空否
 private:
  DblNode<E> *first;                    //表头指针
};
```

双向循环链表的搜索算法



搜索15

搜索成功



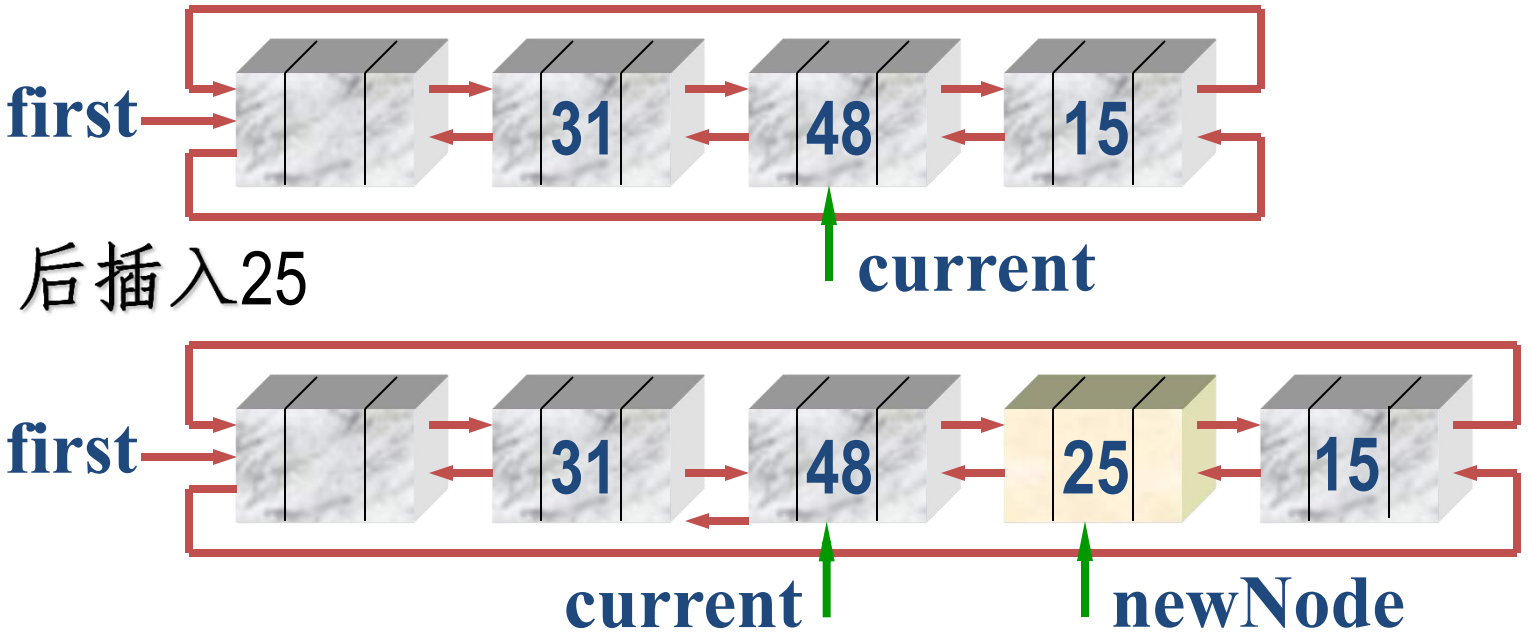
搜索25

搜索不成功

双向循环链表的搜索算法

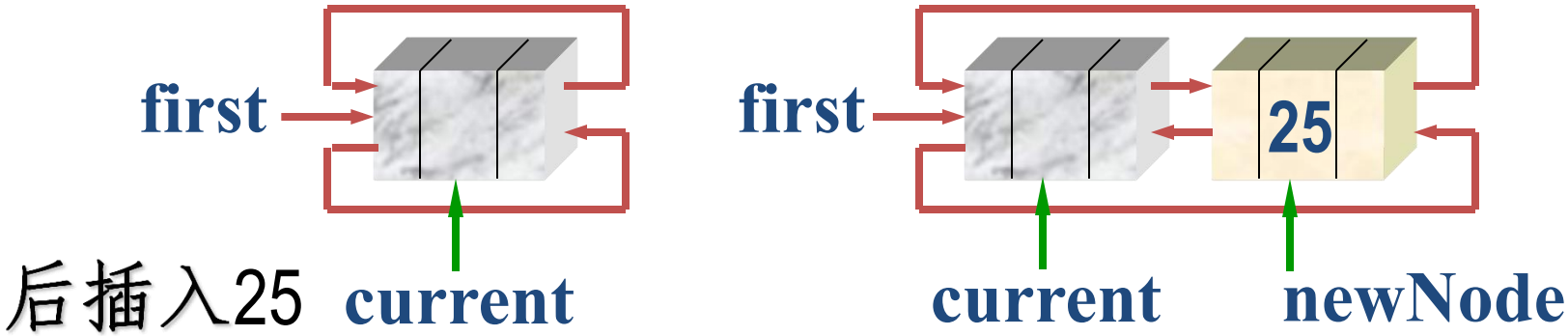
```
template <class E>
DblNode< E> *DblList<E>::Search (E x, int d) {
//在双向循环链表中寻找其值等于x的结点。
   DblNode<E> *current = (d == 0)?
        first->lLink : first->rLink;   //按d确定搜索方向
   while ( current != first && current->data != x )
        current = (d == 0) ?
             current->lLink : current->rLink;
   if ( current != first ) return current;    //搜索成功
   else return NULL;                          //搜索失败
};
```

双向循环链表的插入算法（非空表）




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

双向循环链表的插入算法（空表）




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

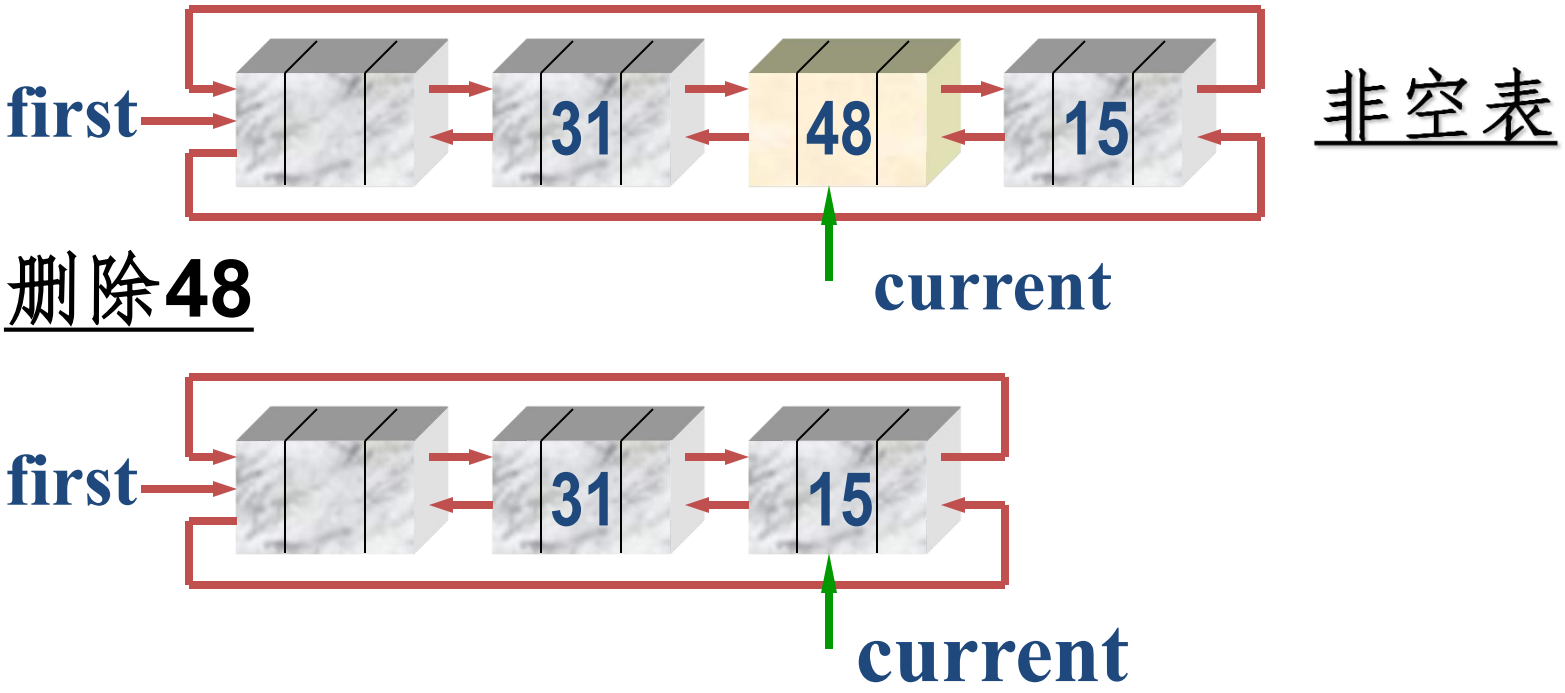
两种方式完全统一

双向循环链表的插入算法

```
template <class E>
bool DblList<E>::Insert ( int i, E x, int d ) {
//建立一个包含有值x的新结点, 并将其按 d 指定的
//方向插入到第i个结点的左侧或右侧。
    DblNode< E> *current = Locate(i, d);
      //按d指示方向查找第i个结点
    if ( current == NULL ) return false;    //插入失败
    DblNode<E> *newNd = new DblNode<E>(x);
    if (d == 0) {        //前驱方向:插在第i个结点左侧
        newNd->lLink = current->lLink; //链入lLink链
        current->lLink = newNd;
```

```
        newNd->lLink->rLink = newNd;  //链入rLink链
        newNd->rLink = current;
    } else {                    //后继方向:插在第i个结点后面
        newNd->rLink = current->rLink;  //链入rLink链
        current->rLink = newNd;
        newNd->rLink->lLink = newNd;   //链入lLink链
        newNd->lLink = current;
    }
    return true;            //插入成功
};
```

双向循环链表的删除算法




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

双向循环链表的删除算法

```
template < class E>
bool DblList< E>::Remove( int i, E& x, int d ) {
//在双向循环链表中按d所指方向删除第i个结点。
    DblNode< E> *current = Locate (i, d);
    if (current == NULL) return false;        //删除失败
    current->rLink->lLink = current->lLink;
    current->lLink->rLink = current->rLink;
        //从lLink链和rLink链中摘下
    x = current->data;  delete current;        //删除
    return true;                               //删除成功
};
```

§2.5 多项式及其运算

$$A(x) = 1 - 10x^6 + 2x^8 + 7x^{14}$$

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

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

- 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

多项式的顺序存储表示

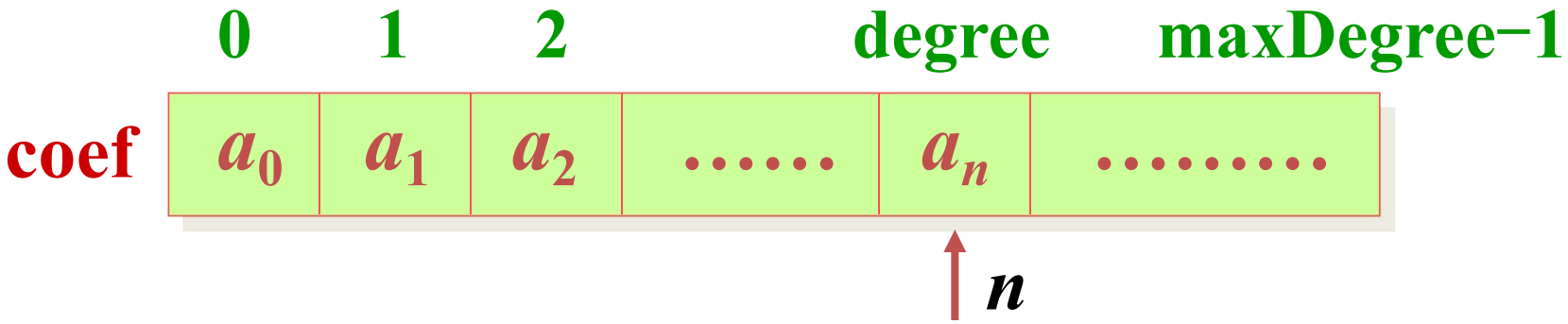
第一种：(静态数组表示)

```
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： (动态数组表示)

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

- 以上两种存储表示适用于指数连续排列的多项式。但对于绝大多数项的系数为零的多项式，如 $P_{101}(x) = 3+5x^{50}-4x^{101}$, 不经济。

第三种：

	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

```
struct term {          //多项式的项定义
        float coef;          //系数
        int exp;             //指数
};

static term termArray[maxTerms];     //项数组
static int free, maxTerms;           //当前空闲位置指针
```

初始化：

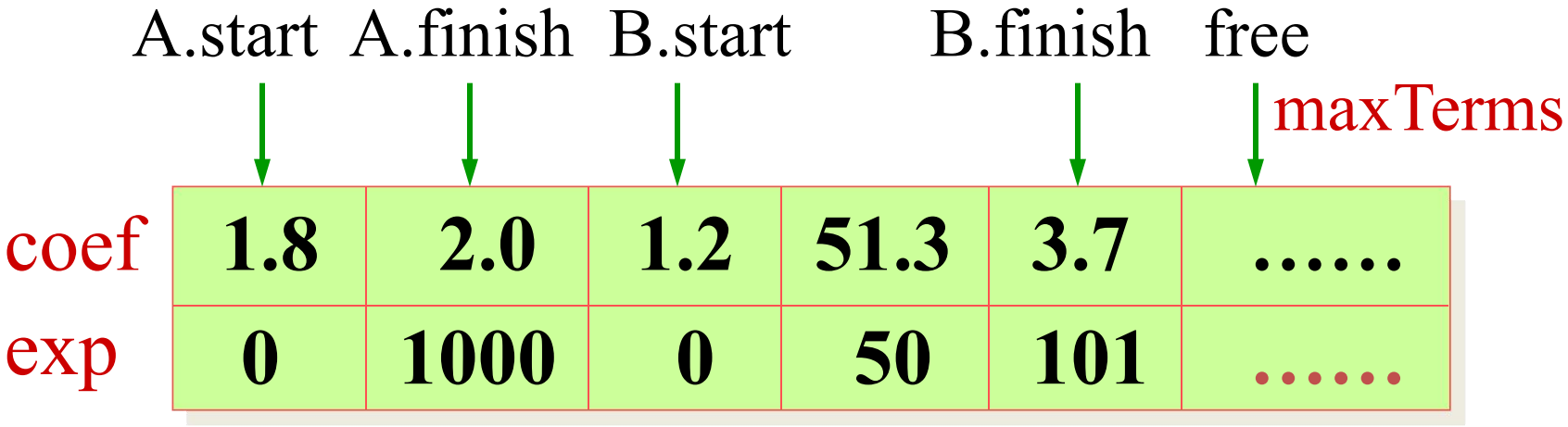
```
// term Polynomial::termArray[maxTerms];
// int Polynomial::free = 0;

class Polynomial {          //多项式定义
   public:
        ……
   private:
      int start, finish;          //多项式始末位置
}
```

两个多项式存储的例子

$A(x) = 2.0x^{1000}+1.8$

$B(x) = 1.2 + 51.3x^{50} + 3.7x^{101}$



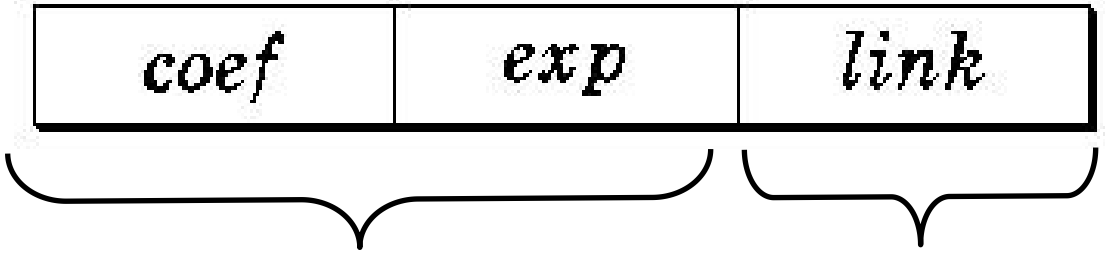
两个多项式存放在termArray中

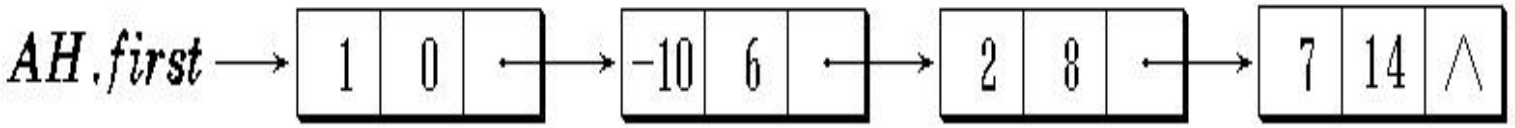
第四种：多项式的链表存储表示

$$A(x) = 1 - 10x^6 + 2x^8 + 7x^{14}$$

每个结点的结构：

$data \equiv Term$





- 多项式顺序存储表示的缺点
  - 插入和删除时项数可能有较大变化，因此要移动大量数据
  - 不利于多个多项式的同时处理

- 多项式链表存储表示的优点
  - 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;                        //链接指针
    Term (float c, int e, Term *next = NULL)
        { coef = c;  exp = e;  link = next;}
    Term *InsertAfter ( float c, int e);
    friend ostream& operator << (ostream&,
                const Term& );
};
```

```
class Polynomial {                          //多项式类的定义
public:
    Polynomal() { first = new Term(0, -1); }  //构造函数
    Polynomal (Polynomal& R);               //复制构造函数
    int maxOrder();                         //计算最大阶数
private:
    Term *first;
    friend ostream& operator << (ostream&,
          const Polynomal& );
    friend istream& operator >> ( istream&,
         Polynomal& );
```

```
    friend void Add ( Polynomial& A, Polynomial& B,
            Polynomial& C );
    friend void Mul ( Polynomial& A, Polynomial& B,
            Polynomial& C );
};
```

两个多项式的相加算法

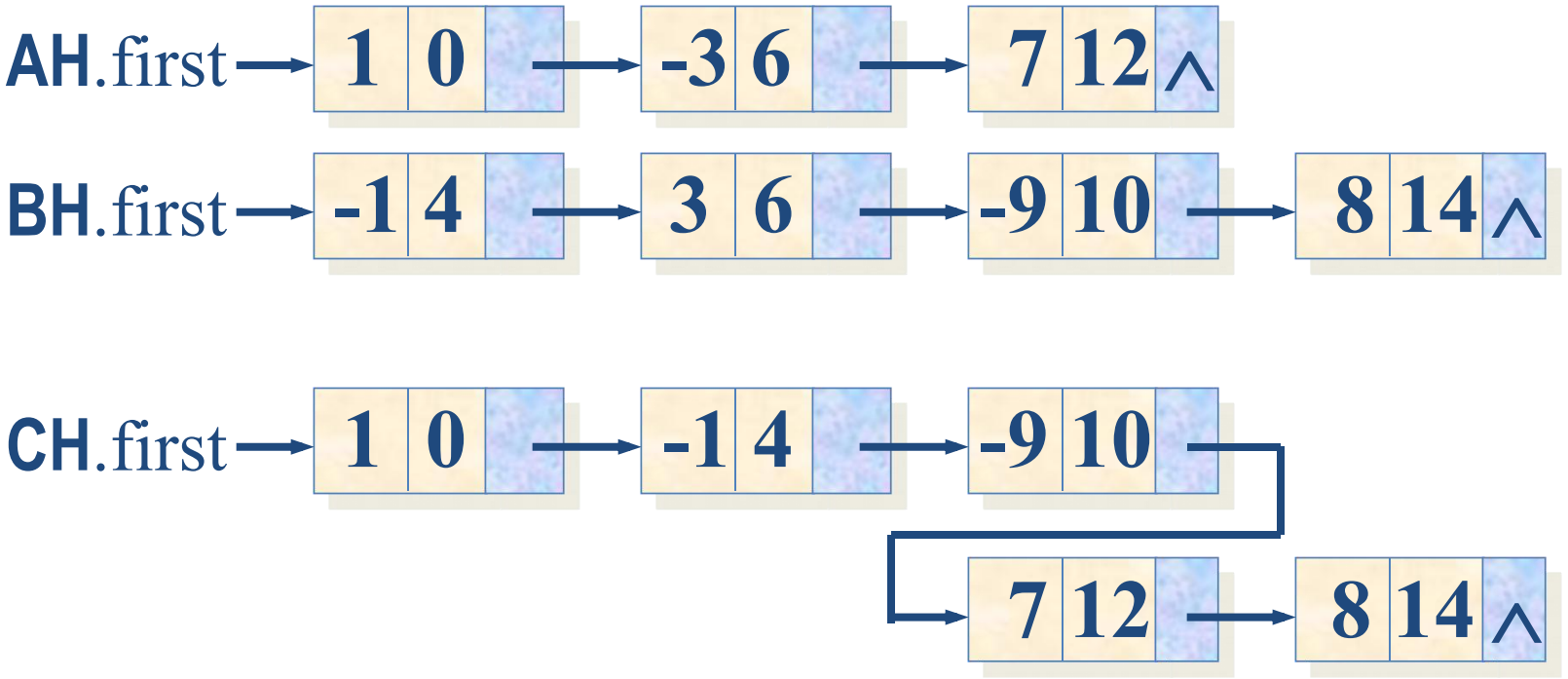
- 设两个多项式都带表头结点，检测指针pa和pb分别指示两个链表当前检测结点，并设结果多项式的链表为C，存放指针为pc，初始位置在C的表头结点。
- 当pa和pb没有检测完各自的链表时，比较当前检测结点的指数域：
  - 指数不等：小者加入C链，相应检测指针pa或者pb进1；
  - 指数相等：对应项系数相加。若相加结果不为零，则结果加入C链，pa与pb进1。
  - 当pa或pb指针中有一个为NULL，则把另一个链表的剩余部分加入到C链。

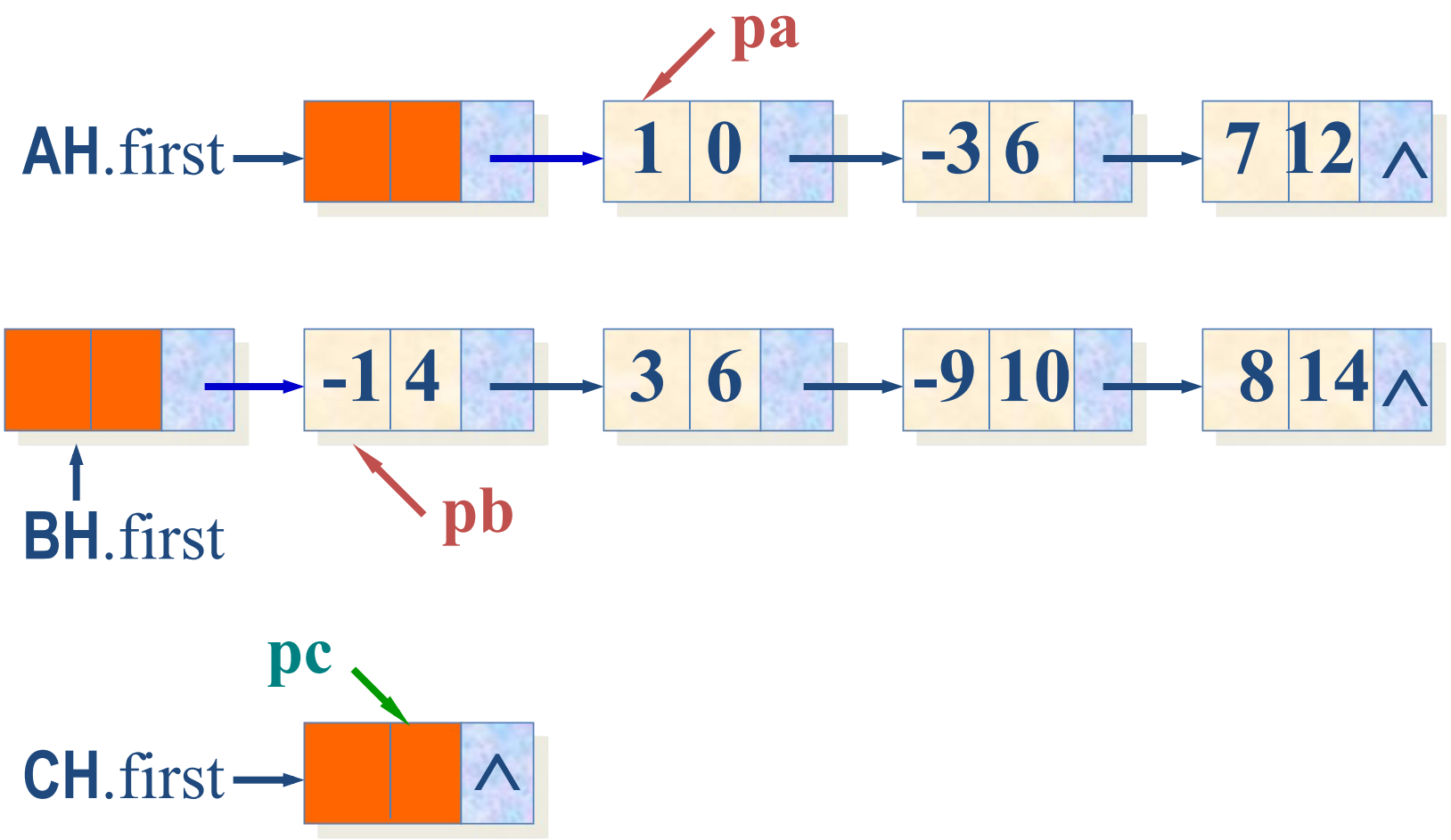
多项式链表的相加

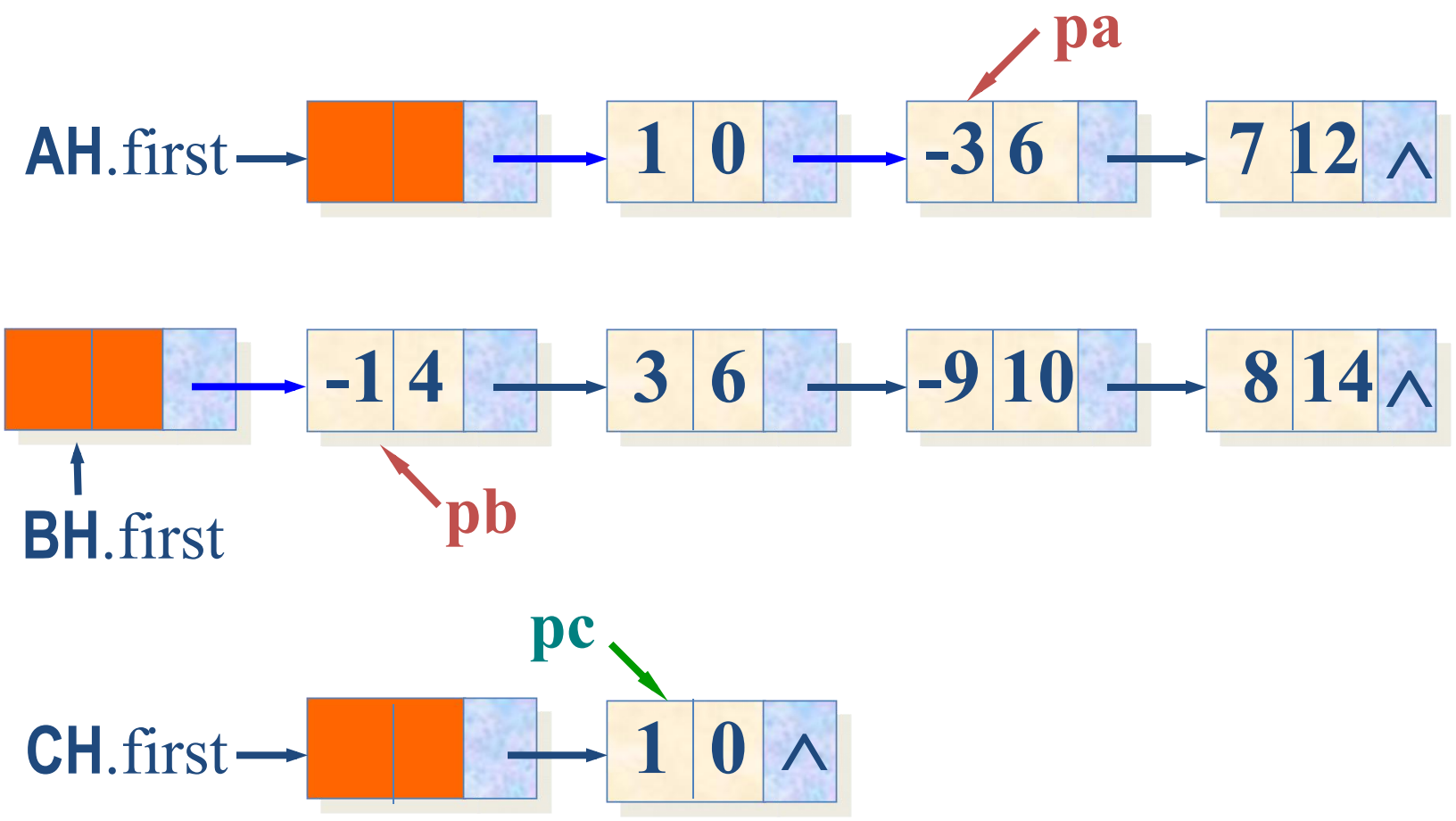
$AH = 1 - 3x^6 + 7x^{12}$

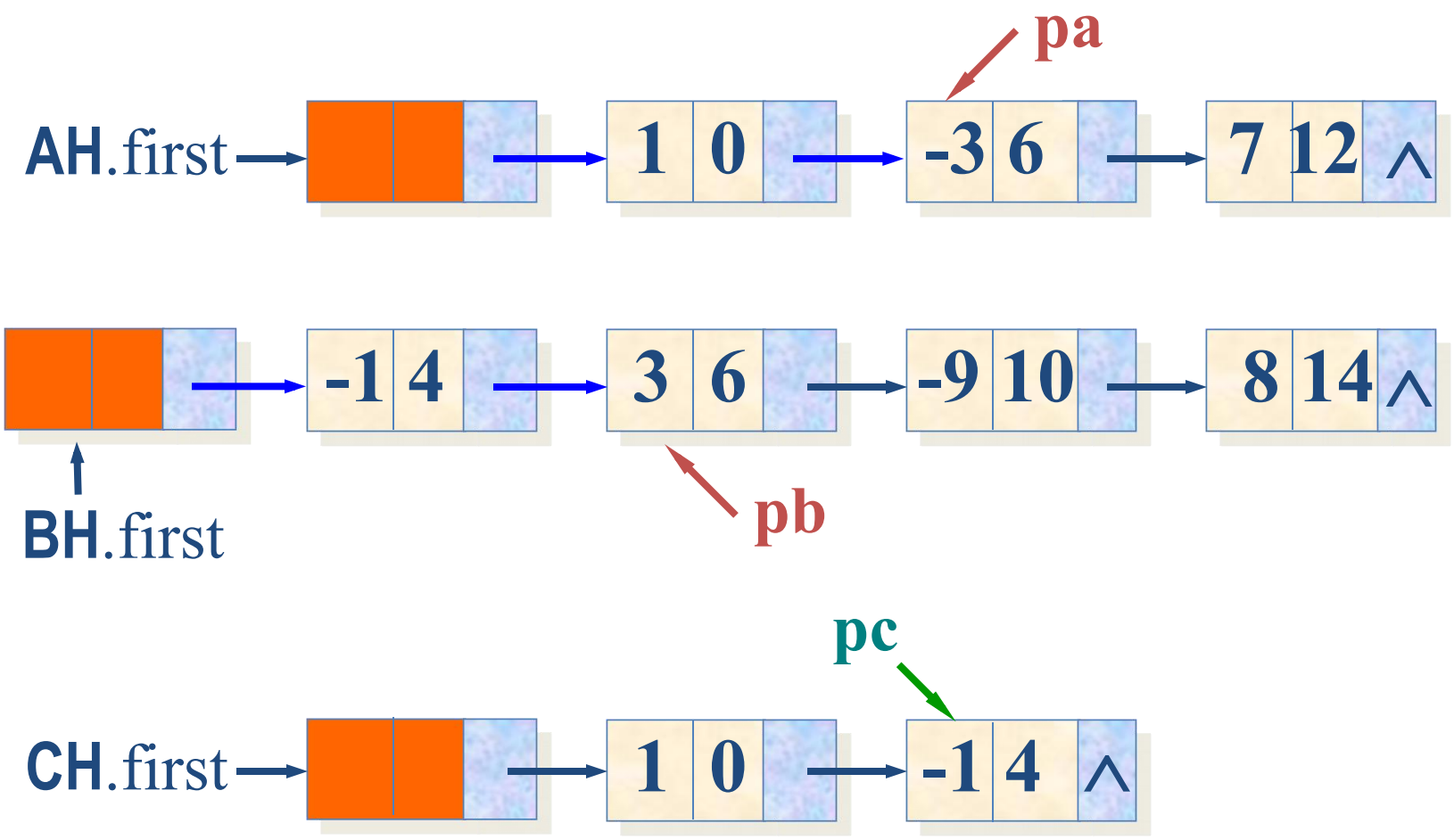
$BH = -x^4 + 3x^6 - 9x^{10} + 8x^{14}$

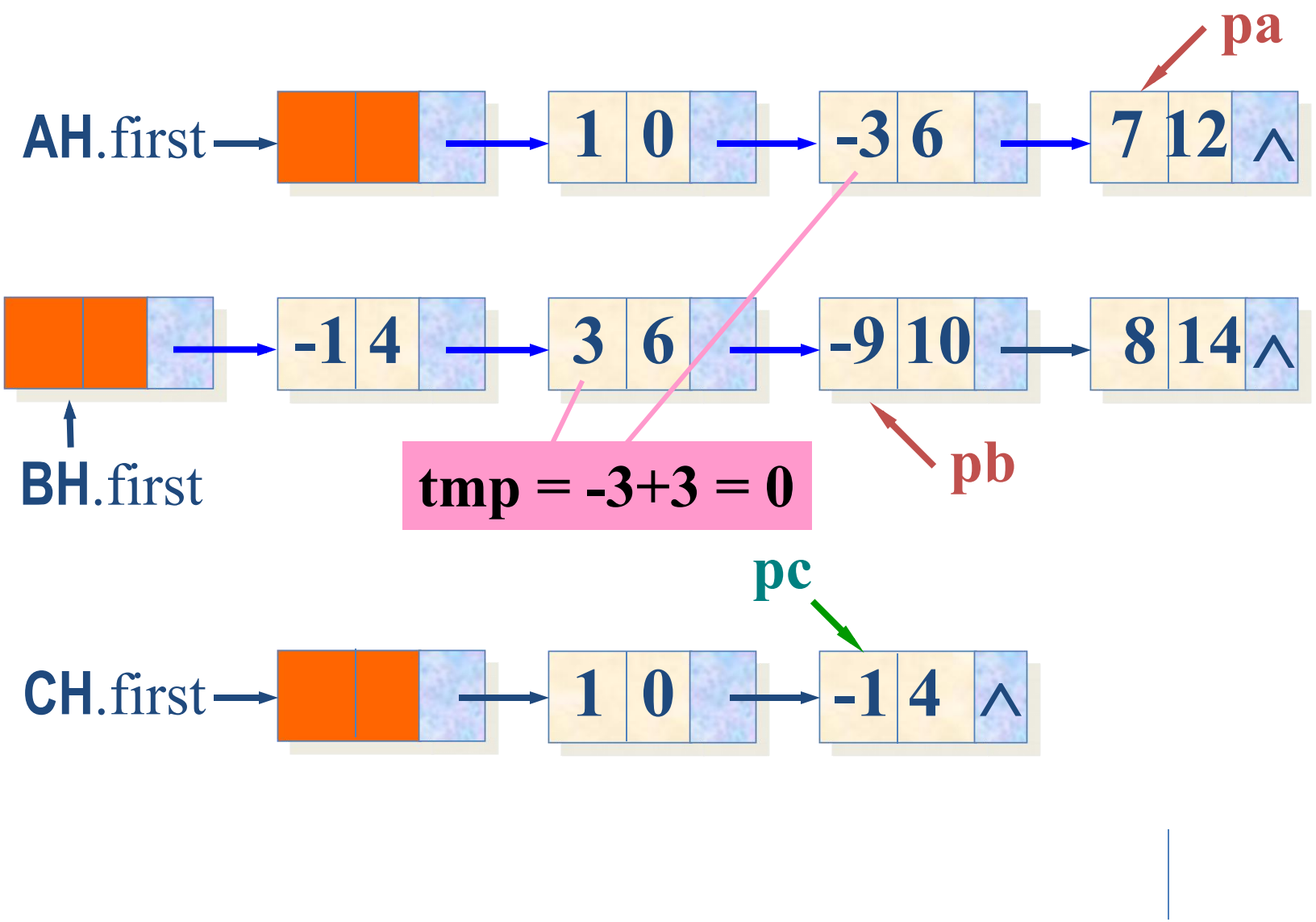
$CH = 1 - x^4 - 9x^{10} + 7x^{12} + 8x^{14}$

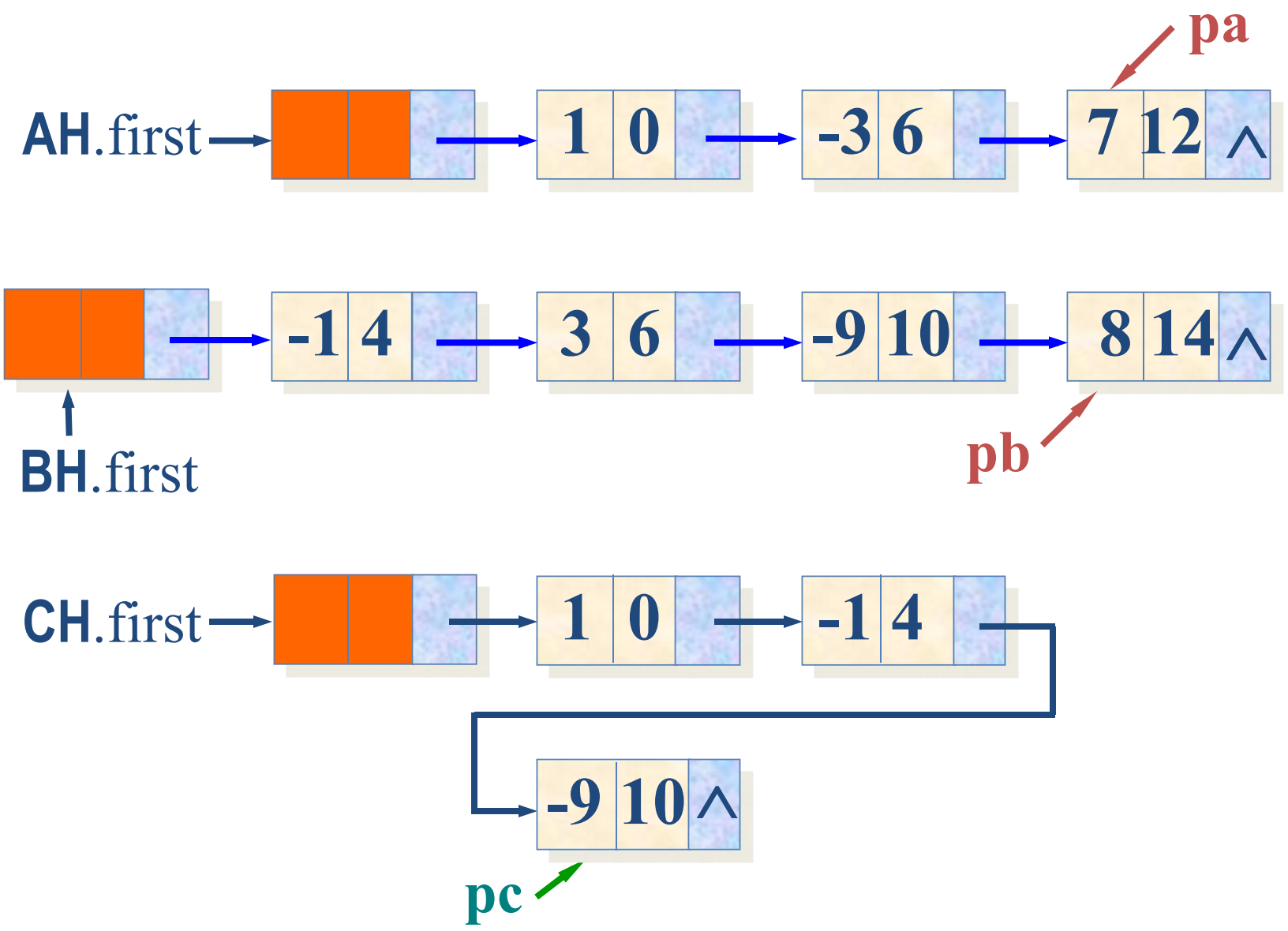


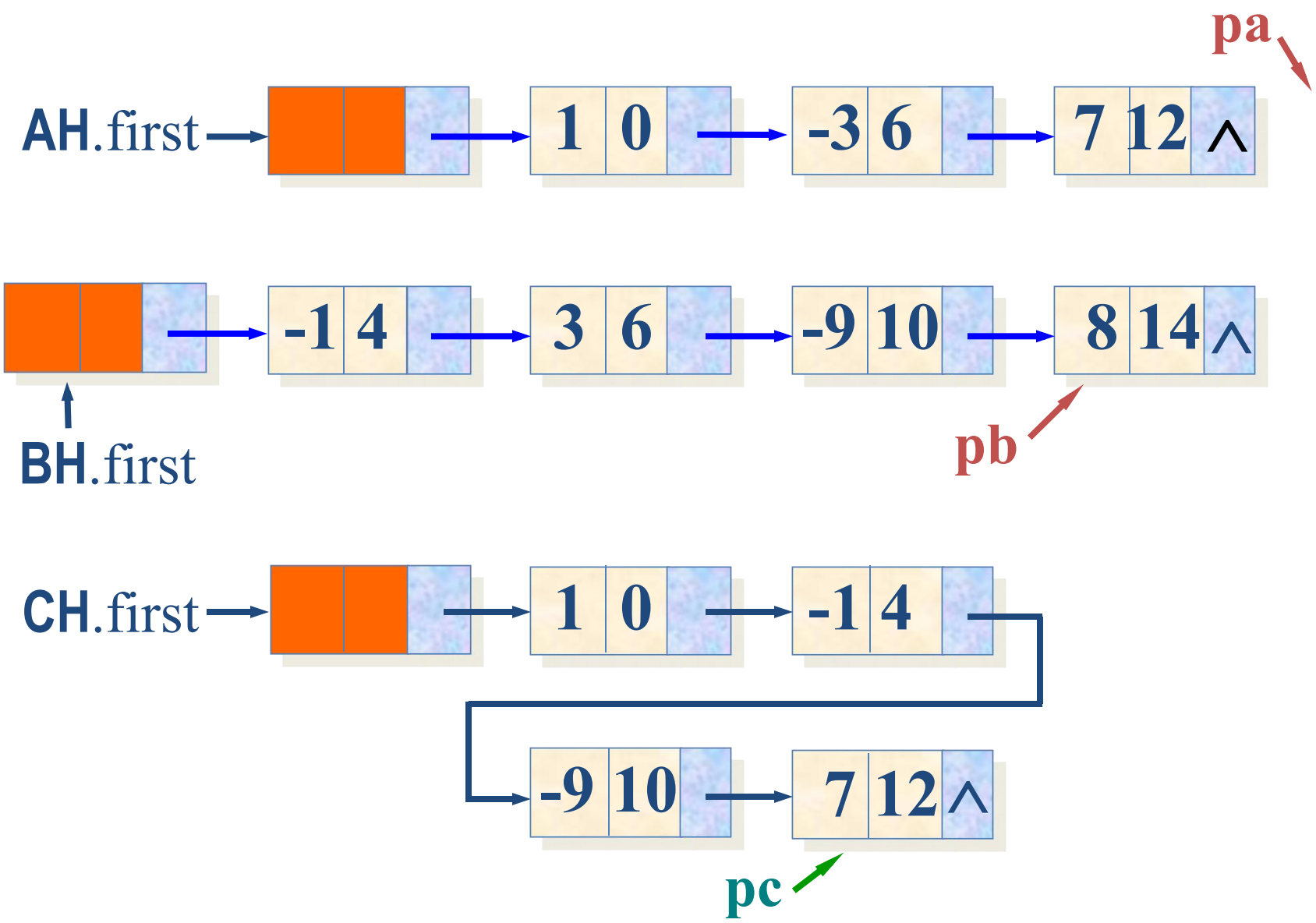

pa
AH.first
1 0
-3 6
7 12 ^
-1 4
3 6
-9 10
8 14 ^
BH.first
pb
pc
CH.first
^

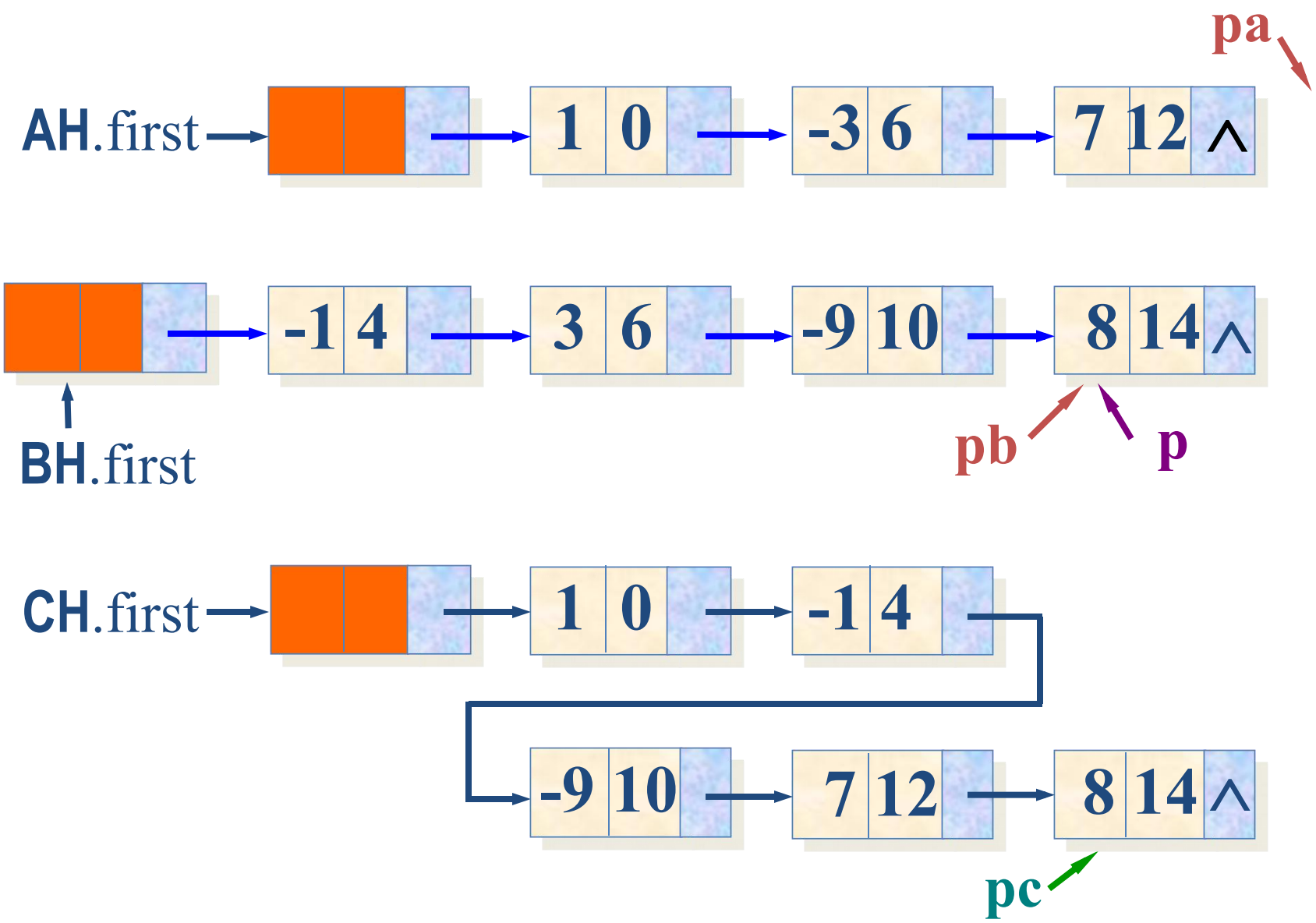

pa
AH.first
1 0
-3 6
7 12 ^
-1 4
3 6
-9 10
8 14 ^
BH.first
pb
pc
CH.first
1 0 ^


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


pa
AH.first
1 0
-3 6
7 12 ^
-1 4
3 6
-9 10
8 14 ^
BH.first
tmp = -3+3 = 0
pb
pc
CH.first
1 0
-1 4 ^


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


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


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

两个多项式均按指数升序存储在带头结点的单链表中：

A(x) = 4 - 2x^3 + 5x^8 + 6x^13+8x^15

B(x) = -3x + 2x^3 - 7x^8 + 9x^10

使用指针 `pa`、`pb` 分别扫描两个链表，结果写入链表 `C`。

1. 按照归并过程，依次写出结果链表中保留的各项，并给出最终的 `C(x)`。

2. 当 `pa` 与 `pb` 指向的项指数相等时应如何处理？本题中的 `x^3` 项和 `x^8` 项分别怎样处理？

3. 当一个链表先扫描结束时，应如何处理另一个链表中的剩余项？

4. 若两个多项式分别有 `m` 项和 `n` 项，说明该相加算法的时间复杂度；并指出算法正确执行所依赖的一项关键存储约定。

1. `A` 的常数项指数0较小，保留 `4`；
2. `B` 的指数1较小，保留 `-3x`；
3. 两边指数均为3，系数相加为 `-2+2=0`，不建立结果结点；
4. 两边指数均为8，系数相加为 `5-7=-2`，建立结果项 `-2x^8`；
5. `B` 的指数10较小，保留 `9x^10`；
6. `B` 扫描结束，将 `A` 的剩余项 `6x^13+8x^15` 接入结果。

C(x) = 4 - 3x - 2x^8 + 9x^10 + 6x^13+8x^15

- 每个结点至多被扫描一次，因此时间复杂度为 `O(m+n)`。
- 关键约定是两个输入链表都必须按指数采用同一方向有序排列（本题为升序）；否则不能使用一次线性归并完成相加。

§2.6 静态链表

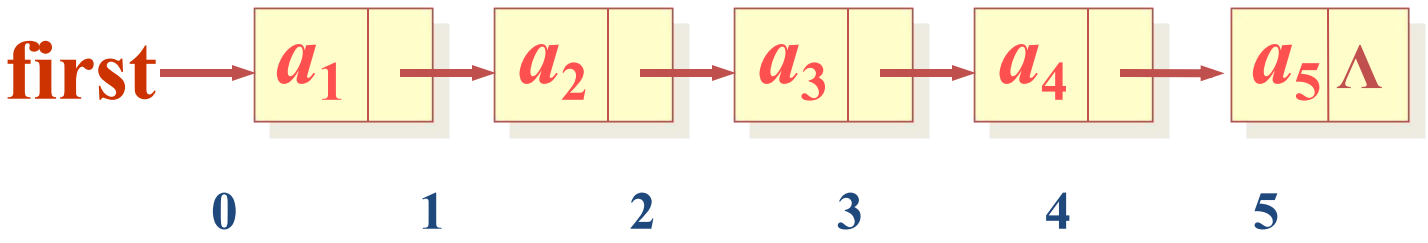
- 为数组中每一个元素附加一个链接指针，就形成静态链表结构。

- 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。

- 处理时可以不改变各元素的物理位置，只要重新链接就能改变这些元素的逻辑顺序。

- 它是利用数组定义的，在整个运算过程中存储空间的大小不会变化。

静态链表的结构



data		a_3	a_4	a_1	a_5	a_2
link	3	2	4	5	−1(0)	1

- 0号是表头结点，link给出首元结点地址。
- 循环链表收尾时link＝0，回到表头结点。如果不是循环链表，收尾结点指针link＝−1。
- link指针是数组下标，因此是整数。