三

算法与数据结构

第三章 线性结构

CONTENTS

目录

3.1 引子

- **【定义】线性结构是一个数据元素的有序（次序）集合**
  - **基本特征：**
    - **集合中必存在唯一的一个“第一元素”；**
    - **集合中必存在唯一的一个“最后元素”；**
    - **除最后元素之外，均有唯一的后续；**
    - **除第一元素之外，均有唯一的前驱。**
  - **线性结构包括线性表、栈、队列、字符串、数组、广义表等。**

【例】一元多项式及其运算。

一元多项式： $f(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1} + a_n x^n$

主要运算：多项式相加、相减、相乘等

【分析】多项式的关键数据是：

- **多项式是一个线性结构**
- **多项式项数从0到*n*，它由 *n*+1 个系数唯一确定，每一项的系数a_i及相应指数 *i***

表示
存储

方法1：采用顺序存储结构直接表示

数组元素对应多项式各项：$a[i]$表示项x^i的系数，下标 i 表示指数；

例如： $f(x) = 4x^5 - 3x^2 + 1$

表示：

下标i	0	1	2	3	4	5	……
a[i]	1	0	–3	0	0	4	……

两个多项式加/减运算：两个数组对应分量相加/相减

? **如何表示稀疏多项式，如：** $f(x) = 4x^{10000} - 3x^{1000} + 1$

方法2：采用顺序存储结构表示多项式的非零项。

每个非零项 a_i 涉及两个信息：指数 i 和系数 a_i，可以将一个多项式看成是一个 (a_i ，i) 二元组的集合，使用结构数组存储。

【例】 $P_1(x) = 9x^{12} + 15x^8 + 3x^2$ 和 $P_2(x) = 26x^{19} - 4x^8 - 13x^6 + 82$

数组下标i	0	1	2	……
系数	9	15	3	–
指数	12	8	2	–

(a) $P_1(x)$

数组下标i	0	1	2	3	……
系数	26	–4	–13	82	–
指数	19	8	6	0	–

(b) $P_2(x)$

按照指数大小升序/降序存储

【例】 $P_1(x)=9x^{12}+15x^8+3x^2$ 和 $P_2(x)=26x^{19}-4x^8-13x^6+82$

$P_1(x)$：(9,12),(15,8),(3,2)

$P_2(x)$：(26,19), (-4,8), (–13,6), (82,0)

数组下标i	0	1	2	……
系数	9	15	3	–
指数	12	8	2	–

(a) $P_1(x)$

数组下标i	0	1	2	3	……
系数	26	–4	–13	82	–
指数	19	8	6	0	–

(b) $P_2(x)$

- **相加过程：从头开始，比较两个多项式当前对应项的指数**
  - **$P_1(x)$的指数>$P_2(x)$的指数，将$P_1(x)$该项二元组信息移到结果多项式**
  - **$P_1(x)$的指数<$P_2(x)$的指数，将$P_2(x)$该项二元组信息移到结果多项式**
  - **$P_1(x)$的指数=$P_2(x)$的指数， $P_1(x)$ 与$P_2(x)$系数相加，将新项移到结果多项式**
- **最后得到的结果多项式是：((26,19), (9,12), (11,8), (–13,6), (3,2), (82,0))**

表示成： $P_2(x)=26x^{19}+9x^{12}+11x^8-13x^6+3x^2+82$

方法3：采用链表结构来存储多项式的非零项。

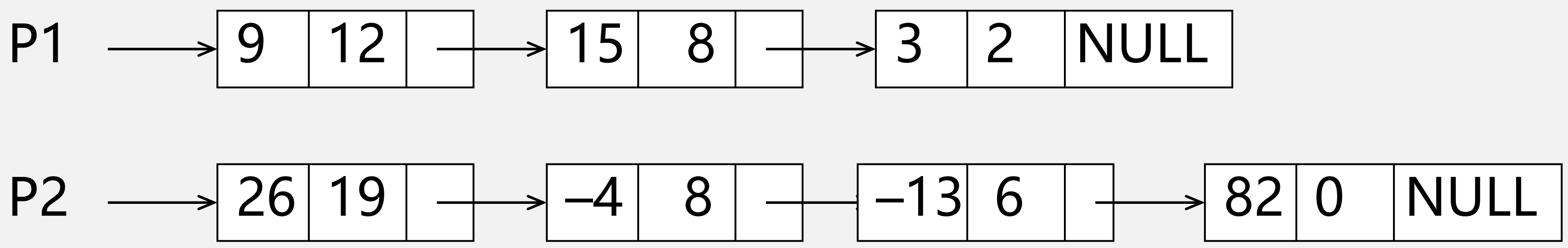
每个链表结点存储多项式中的一个非零项，包括系数和指数两个数据域以及一个指针域，表示为：

coef	exp	link

【定义】

```
typedef struct PolyNode *Polynomial;
typedef struct PolyNode {
        int coef;
        int exp;
        Polynomial link;
}
```

【例】



➢数据结构的**操作**与数据结构的**存储方式**是密切相关的，**不同的数据存储方式**，相应的**操作实现方法**是**不一样**的，**时间空间效率也不同**。

	方法一	方法二	方法三
存储方式	数组	数组	链表
操作	简单	简单	复杂
时间效率	稀疏时效率低	高	高
空间效率	浪费空间	高/灵活性不足，需预先确定大小	高

3.2 线性表

- 线性表的定义及操作
- 顺序存储结构
- 链式存储结构
- 广义表与多重链表

3.2.1 线性表的定义及操作

- **【定义】用数据元素的有限序列表示，由 n ($n\geq0$) 个数据元素 $a_1, a_2, \ldots, a_n$ 组成的有限并且有序的序列。**

$$(a_1, \underbrace{a_2, \ldots a_{i-1}, a_i, a_{i+1}, \ldots,}_{\text{数据元素}} a_n)$$

线性起点

a_i的直接前趋

a_i的直接后继

线性终点

n为元素总个数，即表长

下标，是元素的序号，表示元素在表中的位置

$n=0$时称为 **空表**

线性表中的数据元素之间存在着序偶关系 $<a_{i-1}, a_i>$

➢ 【例】26 个英文字母组成的字母表：

$$(A, B, C, \ldots, Z)$$

数据元素都是字母; 元素间关系是线性

➢ 【例】 分析学生情况登记表

学号	姓名	性别	年龄	班级
2016011810205	于春梅	女	18	2016级电信161班
2016011810260	何仕鹏	男	18	2016级电信161班
2016011810284	王爽	女	18	2016级通信161班
2016011810360	王亚武	男	18	2016级通信161班
:	:	:	:	:

数据元素都是记录; 元素间关系是线性

抽象数据类型线性表的定义：

ADT List {

数据对象：D = { a_i | $a_i \in$ ElemSet, i = 1, 2, ..., n, $n \geq 0$ }

数据关系：R1 = { <a_{i-1}, a_i > | a_{i-1}, $a_i \in$ D, i = 2, ..., n }

基本操作：

1、List MakeEmpty()：初始化一个新的空线性表L；

2、ElementType FindKth(int K, List L)：根据指定的位序K，返回相应元素；

3、int Find(ElementType X, List L)：已知X，返回线性表L中与X相同的第一个元素的相应位序i；若不存在则返回空；

4、void Insert(ElementType X, int i, List L)：指定位序i前插入一个新元素X；

5、void Delete(int i, List L)：删除指定位序i的元素；

6、int Length(List L)：返回线性表L的长度n。

3.2.2 线性表的顺序存储结构

- 按顺序方式存储数据元素称为顺序表
- 顺序表示是把逻辑上相邻的数据元素存储在物理上相邻的存储单元中的存储结构。

- 【例】一维数组在内存中占用的存储空间是一组连续的存储区域，一维数组是一种顺序存储结构

- 假设线性表的每个元素需占用l个存储单元，线性表中第$i+1$个数据元素的存储位置LOC(a_{i+1})和第i个数据元素的存储位置LOC(a_i)之间满足下列关系：

$$LOC(a_{i+1}) = LOC(a_i) + l$$

- 线性表的第i个数据元素a_i的存储位置为：

$$LOC(a_i) = LOC(a_1) + (i-1)*l$$

线性表的顺序存储结构示意图

存储地址	内存状态	数据元素在线性表中的位序
$LOC(a_1)$	a_1	1
$LOC(a_1)+l$	a_2	2
⋮	⋮	⋮
$LOC(a_1)+(i-1)\times l$	a_i	i
⋮	⋮	⋮
$LOC(a_1)+(n-1)\times l$	a_n	n
$LOC(a_1)+n\times l$		空闲
$LOC(a_1)+(\text{maxlen}-1)\times l$		

顺序表的特点：

以物理位置相邻表示逻辑关系。

任一元素均可随机存取。

在内存中用**地址连续的**一块存储空间**顺序存放**线性表的各元素。

下标i	0	1	…	i-1	i	…	n-1	…	MAXSIZE-1
Data	a_1	a_2	…	a_i	a_{i+1}	…	a_n	…	-

Last

```
typedef  struct {
        int  Data[MAXSIZE];
        int  Last;    /*当前的最后一个元素的下标*/
} LNode;
typedef LNode *List;
List L;
```

访问下标为 i 的元素：L->Data[i]

线性表的长度：L->Last+1

主要操作的实现

1. 初始化（建立空的顺序表)

```
List MakeEmpty( )
{     List L;
      L = (List)malloc(sizeof(struct LNode));
      L->Last = -1;
      return L;
}
```

2. 查找 (找给定值X相等的数据元素)

```
int Find( ElementType X, List L )
{   int i = 0;
    while(i<=L->Last && L->Data[i]!= X )
        i++;
    if (i>L->Last)  return ERROR; /*如果没找到，返回错误 */
    else  return i;  /* 找到后返回的是存储位置 */
}
```

查找算法性能分析

- **查找成功的平均比较次数**

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

- P_i是查找某个元素的概率
- C_i 比较次数

若查找概率相等，则查找成功的比较次数为

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

- **查找不成功,数据比较 n 次。**

3. 插入

第 i $(1 \le i \le n+1)$个位置上插入一个值为X的新元素

下标i	0	1	…	i-1	i	…	n-1	…	MAXSIZE-1
Data	a_1	a_2	…	a_i	a_{i+1}	…	a_n	…	-

Last

先移动，再插入

下标i	0	1	…	i-1	i	i+1	…	n	…	MAXSIZE-1
Data	a_1	a_2	…	X	a_i	a_{i+1}	…	a_n	…	-

Last

算法思想

1）检查 i 值**是否超出**所允许的范围 $(1 \le i \le n+1)$，若超出，则进行“表满/位置不合法”错误处理；

2）将线性表的第 i 个元素和它后面的所有元素均**后移一个位置**；

3）将新元素**X写入**到空出的第 i 个位置上；

4）**修改Last指针**，指向最后一个元素。

1	2	3	4	5	6	7

1	2	3	4	7	5	6

7	1	2	3	4	5	6

插入算法

```
bool Insert( ElementType X, int i, List L )
{   int j;
    if (L->Last == MAXSIZE-1 ){/*表空间已满，不能插入*/
        printf("表满");
        return FALSE;
    }
    if (i<1||i>L->Last+2) { /*检查插入位置的合法性*/
        printf("位置不合法");
        return FALSE;
    }
    for (j =L->Last;j>=i-1;j--)
        L->Data[j+1]=L->Data[j]; /*将 ai ~ an倒序向后移动*/
    L->Data[i-1] = X;    /*新元素插入*/
    L->Last++;           /*Last仍指向最后元素*/
    return TRUE;
}
```

插入算法的时间复杂度分析：

- 问题规模是表的长度，设它的值为 n。
- 算法的时间主要花费在向后移动元素的 for 循环语句上。该语句的循环次数为 $(n-i+1)$。由此可看出，所需移动结点的次数不仅依赖于表的长度 n，而且还与插入位置 i 有关。
- 当插入位置在表尾 $(i=n+1)$ 时，不需要移动任何元素；这是最好情况，其时间复杂度 $O(1)$。
- 当插入位置在表头 $(i=1)$ 时，所有元素都要向后移动，循环语句执行 n 次，这是最坏情况，其时间复杂度 $O(n)$。

- **算法的平均时间复杂度：**
  - **设 p_i 为在第 i 个元素之前插入一个元素的概率，假设在表中任何位置($1 \le i \le n+1$) 插入结点的机会是均等**

$$p_i = \frac{1}{n+1}$$

  - **则插入一个元素时所需移动次数的平均期望值为：**

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

在顺序表上做插入运算，平均要移动表上一半元素。当表长 n 较大时，算法的效率相当低。算法的平均时间复杂度为 $O(n)$。

4. 删除（删除表的第 i $(1\leq i\leq n)$个位置上的元素)

下标i	0	1	…	i-1	i	…	n-1	…	MAXSIZE-1
Data	a_1	a_2	…	a_i	a_{i+1}	…	a_n	…	-

Last

后面的元素依次前移

下标i	0	1	…	i-1	…	n-2	n-1	…	MAXSIZE-1
Data	a_1	a_2	…	a_{i+1}	…	a_n		…	-

Last

删除算法

```
bool Delete( int  i, List L )
{   int  j;
    if( i<1||i>L->Last+1) {/*检查空表及删除位置的合法性*/
         printf (“第%d个位置不存在元素”, i );
         return false;
    }
    for (j=i;j<=L->Last;j++)
         L->Data[j-1]=L->Data[j]; /*将 ai+1~ an顺序向前移动*/
        L->Last--;    /*Last仍指向最后元素*/
        return true;
}
```

删除算法的复杂度分析

- 算法的平均时间复杂度：设 q_i 为删除第 i 个元素的概率，假设在表中任何位置($1 \le i \le n$)删除结点的机会均等：

$$q_i = \frac{1}{n}$$

在长度为 n 的线性表中删除一个元素时所需移动元素次数的期望值为

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

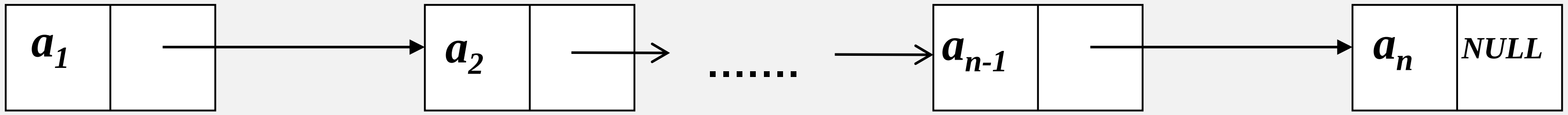
由此可见，在顺序表上做删除运算，平均约要移动表上一半元素。当表长 n 较大时，算法的效率相当低。算法的平均时间复杂度为 $O(n)$。

线性表的顺序表示

- 特征：逻辑上相邻，物理上也相邻；
- 优点：随机查找快
- 缺点：进行插入和删除操作时，需移动大量的元素。

3.2.3 线性表的链式存储结构

- 为避免大量结点的移动，线性表的另一种存储方式，**链式存储结构**，简称为链表
- 链表：插入、删除不需要移动数据元素，只需要修改链



逻辑上相邻的两个元素不要求物理上也相邻：通过“链”建立起数据元素之间的逻辑关系

相关概念

- 用一组地址任意的存储单元存放线性表中的数据元素
- 结点（表示数据元素）＝
  元素(数据元素的映像)+指针(指示后续存储位置)
- 以“结点的序列”表示线性表——链表

【例】线性表：（赵，钱，孙，李，周，吴，郑，王）

顺序表

存储地址	存储状态
0031	赵
0033	钱
0035	孙
0037	李
0039	周
0041	吴
0043	郑
0045	王

链表

结点

存储地址	数据域	指针域
0001	李	0043
0007	钱	0013
0013	孙	0001
0019	王	NULL
0025	吴	0037
0031	赵	0007
0037	郑	0019
0043	周	0025

头指针 H

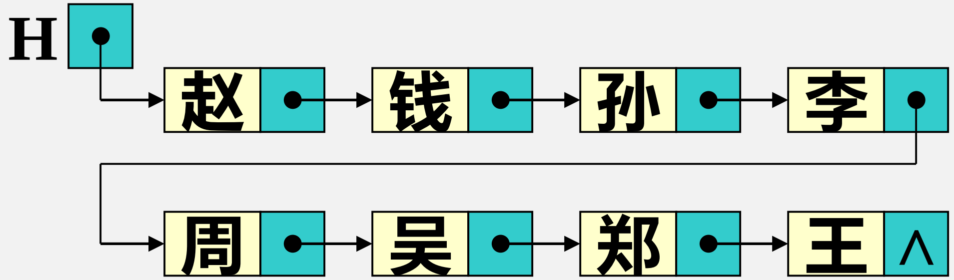
0031

链表

单链表

链

指针



单链表的表示

单链表在 C 语言中可用“结构指针”来描述：

```
typedef  struct  LNode{
        DataType    data;   //数据元素的类型
        struct   LNode  *Next;   //指示结点地址的指针
}
typedef LNode *List;
List L;
```

单链表的基本操作

- 求表长——int Length(List L)，返回线性表*L*的长度*n*；
- 查找；
  - 按值查找——ElementType ElementType FindKth(int K, List L)，根据指定的位序*K*，返回相应元素；
  - 定位——int Find(ElementType X, List L)，返回线性表L中与X相同的第一个元素的相应位序 *i*；若不存在则返回空；
- 插入—— Insert(ElementType X, int i, List L)，指定位序 *i* 前插入一个新元素X；
- 删除——Delete(int i, List L)，删除指定位序 *i* 的元素

主要操作的实现

1.求表长

```
int   Length ( List L )
{     List  p = L;          /* p指向表的第一个结点*/
      int  cnt = 0;
      while ( p ) {
            p = p->Next;
            cnt++;          /* 当前p指向的是第cnt个结点*/
      }
      return  cnt;
}
```

时间性能为 O(n)。

2. 查找

(1) 按序号查找: FindKth;

```
ElementType FindKth( int K, List L )
{    List  p = L;
     int  cnt = 1;  /*位序从1开始*/
     while (p!=NULL && cnt < K ) {
         p = p->Next;
         cnt++;
     }
    if (cnt==K)
         return p->data;/* 找到第K个, 返回指针 */
    else
         return -1;     /* 否则返回空 */

}
```

算法的时间复杂度为: $O(n)$

（2）按值查找: Find

```c
LNode *Find( ElementType X, List L )
{
    List  p = L;
    while ( p!=NULL && p->Data != X )
        p = p->Next;
    if(p)
        return p;
    else
        return NULL;
}
```

该算法的执行时间与 X有关，时间复杂度为：$O(n)$

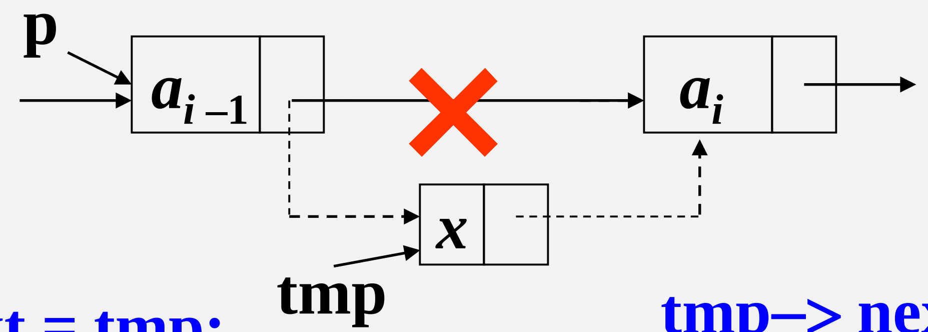
3. 插入（在链表的第 i-1(1≤i≤n+1)个结点后插入值为X的新结点)

步骤： 1、生成一个数据域为 x 的新结点tmp。

2、首先找到 a_{i-1} 的存储位置 p。

3、插入新结点：①、结点 a_{i-1} 的指针域指向新结点。

②、新结点的指针域指向结点 a_i。



p –> next = tmp;

tmp–> next = p –> next;

插入算法（无头结点）

```
List Insert( ElementType X, int i, List L )
{     List  p, tmp;
      tmp = (List )malloc(sizeof(List)); /*申请、填装结点*/
      tmp->Data = X;
      if ( i == 1 ) {       /* 新结点插入在表头 */
         tmp->Next = L;
         return tmp;        /*返回新表头指针*/
       }
      else{
         int cnt=1;
         p=L;
         while (p && cnt<i-1) { /*查找位置i-1的结点*/
           p=p->next;
           cnt++;
         }
        if ( p == NULL || cnt!=i-1) {       /* 第i-1个不存在，不能插入 */
           printf("参数i错");
           free(tmp);  return NULL;
        }else {
           tmp->Next = p->Next;    /*新结点插入在第i-1个结点的后面*/
           p->Next = tmp;
           return L;
       }
}}
```

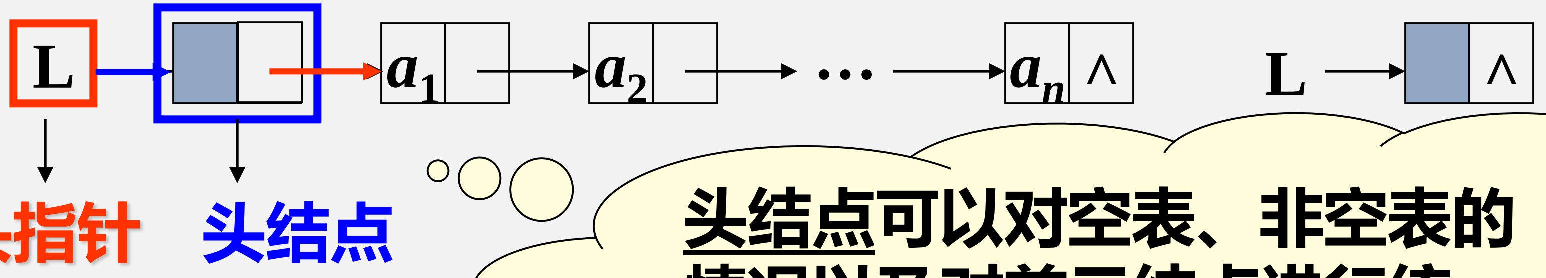
平均查找次数n/2，平均时间性能：$O(n)$

头结点：在单链表的第一个结点之前人为地附设的一个结点。

头结点
- **数据域**
  - 不存放任何数据
  - 存放附加信息（链表的结点个数等）。
- **指针域** 存放第一个结点的地址

（若线性表为空表，则“空”，用 ^ 表示。）

头指针存放**头结点**的地址。



头结点可以对空表、非空表的情况以及对首元结点进行统一处理，编程更方便。

插入算法 （带头结点）

```
Bool Insert( ElementType X, int i, List L )
{    List p, tmp;   /*p指向表头，tmp临时结点*/
     int cnt i=0;
     p=L;
     while (p && cnt<i-1) { /*查找位置i-1的结点*/
       p=p->next;
       cnt++;
    }
    if (p==NULL)||cnt!=i-1) {/*所找的结点不在L中*/
         print("插入位置参数错误\n");
         return false;
    }
    else{
       tmp =(List)malloc(sizeof(struct LNode)); /*申请、填装结点*/
       tmp->Data = X;
       tmp->Next = p->Next;      /*新结点插入在第i-1个结点的后面*/
       p->Next =tmp;
       return true;
      }
}
```

4. 删除（删除链表的第 i $(1 \le i \le n)$ 个位置上的结点)

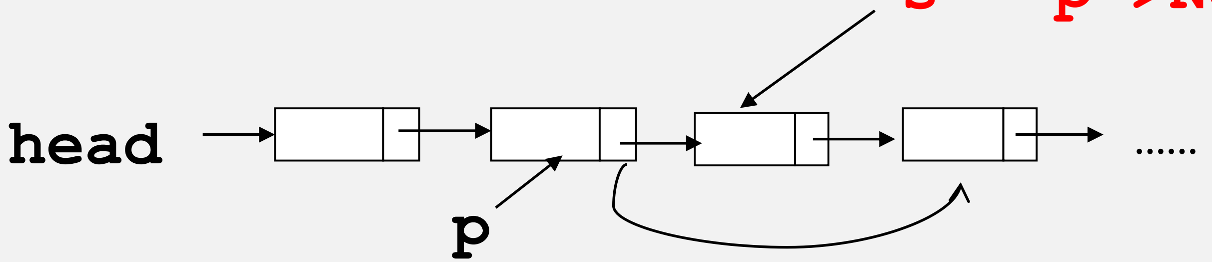
（1）先找到链表的第 i-1个结点，指针p指向该结点;

（2）再用指针s指向要被删除的结点（p的下一个结点）；

（3）然后修改指针，删除s所指结点;

（4）最后释放s所指结点的空间。

```
s = p->Next;  free（s）;
```




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

删除算法(带头结点)

```
bool Delete( int  i, List L )
{  List  p, s;  //p指向被删除结点前一结点，s指向被删除结点
   int cnt=0;
   p=L;
   while (p && cnt<i-1){ /*查找第i-1个结点*/
       p=p->next;
       cnt++;
   }
   if (p==NULL)||cnt!=i-1) {/*所找的结点不在L中*/
       print("插入位置参数错误\n");
       return false;
   }
   else{
       s = p->Next;             /*s指向第i个结点*/
       p->Next = s->Next;       /*从链表中删除*/
       free(s);                 /*释放被删除结点 */
       return true;}
}
```

顺序表与链表的比较

	顺序存储	链式存储
存储空间	静态分配，程序执行之前必须明确规定它的存储规模	存储空间是动态分配的。
存储密度	大	小
访问方式	随机存取，易于查找和修改	顺序存取，查找和修改需要遍历整个链表
插入/删除时移动元素个数	平均需要移动近一半元素	不需要移动元素，只需要修改指针

存储密度=结点数据本身所占的存储量/结点结构所占的存储总量

【例】如何表示一个单位的人员情况。

- 简单方法：**线性表**，按照进单位的时间先后顺序排列：

  （张三，李四，王五，钱六，孙七，……）

- 如何体现三个不同部门？确保同一个部门放在一起。那么可以用**三个有序序列的子表**构成的**线性表**来表示：

  （（张三，……），（李四，孙七，……），（王五，钱六，……））

- 如果想表示这个单位的负责人是谁，可将负责人作为表的第一元素：

  （丁一，（张三，……），（李四，孙七，……），（王五，钱六，……））

3.2.4 广义表与多重链表

（1）广义表

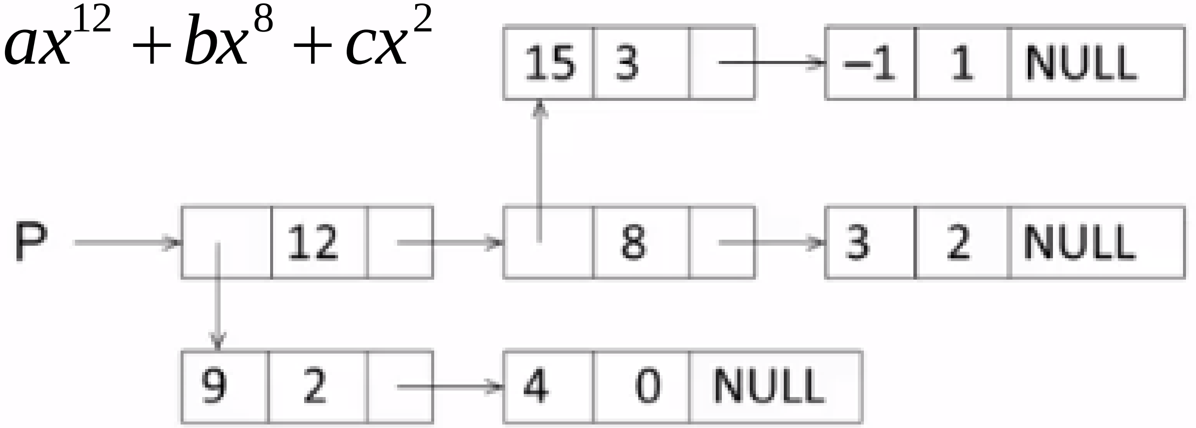
【例】已知一元多项式的表示，那么二元多项式又该如何表示？

$$P(x,y)=9x^{12}y^{2}+4x^{12}+15x^{8}y^{3}-x^{8}y+3x^{2}$$

【分析】可以将上述二元多项式看成关于x的一元多项式

$$P(x,y)=\left(9y^{2}+4\right)x^{12}+\left(15y^{3}-y\right)x^{8}+3x^{2}$$

$$=ax^{12}+bx^{8}+cx^{2}$$



- 广义表是**线性表的推广**。
- 相同点：也是由n个元素组成的**有序序列**。
- 不同点：
  - 线性表： n个元素都是基本的**单元素**。
  - 广义表：元素不仅可以是单元素也可以是**另一个广义表**。
- 广义表记为：GList=(a_1,a_2,...,a_i,...a_n)， a_i是单元素或广义表。

广义表的数据结构定义如下：

```
typedef struct GNode{
    int Tag;  /*标志域：0表示该结点是单元素，1表示该结点是广义表 */
    union { /* 公用域：子表指针域Sublist与单元素数据域Data复用*/
      ElementType   Data;
      struct GNode *SubList;
    } URegion;
    struct GNode *Next;   /* 指向后继结点 */
} GList;
```

<table>
<tr><td rowspan="2">Tag</td><td>Data</td><td rowspan="2">Next</td></tr>
<tr><td>SubList</td></tr>
</table>

多重链表

【定义】广义表采用链表存储的方式实现时，其元素可能还是另一个子链表的起点指针，这类链表称为“多重链表”。

- 一般来说，多重链表中每个结点的有多个指针域。
- 但包含两个指针域的链表并不一定是多重链表，比如双向链表不是多重链表。

- 多重链表在数据结构实现中有广泛的用途，基本上如树、图等相对复杂的数据结构都可以采用多重链表的方式实现存储。

【例】矩阵的表示——方法一：采用二维数组表示

但二维数组表示矩阵有两个缺陷：

- 一是数组的**大小需要事先确定**；
- 另一个是当矩阵包含许多0元素时，将造成大量的**存储空间浪费**。

【例】对于下面A和B这样的“**稀疏矩阵**”最好是**只存储非0元素**。如何用多重链表方式实现存储?

$$A=\begin{bmatrix} 18 & 0 & 0 & 2 & 0 \\ 0 & 27 & 0 & 0 & 0 \\ 0 & 0 & 0 & -4 & 0 \\ 23 & -1 & 0 & 0 & 12 \end{bmatrix} \qquad B=\begin{bmatrix} 0 & 2 & 11 & 0 & 0 & 0 \\ 3 & -4 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 9 & 13 & 0 \\ 0 & -2 & 0 & 0 & 10 & 7 \\ 6 & 0 & 0 & 5 & 0 & 0 \end{bmatrix}$$

【分析】方法二：十字链表来存储稀疏矩阵。（只存储非零结点）

- 结点总体结构：用一个标识域Tag来区分头结点和非0元素结点：头节点的标识值为“Head”，矩阵非0元素结点的标识值为“Term”。
- **矩阵非0元素的结点:**
  - **指针域**：一个是行指针(或称为向右指针)Right，另一个是列指针（或称为向下指针）Down。
  - **数据域**：行坐标Row、列坐标Col和数值Value。

<table>
<tr><td colspan="3">Tag</td></tr>
<tr><td>Down</td><td>URegion</td><td>Right</td></tr>
</table>

(a) 结点的总体结构

<table>
<tr><td colspan="4">Term</td></tr>
<tr><td rowspan="2">Down</td><td>Row</td><td>Col</td><td rowspan="2">Right</td></tr>
<tr><td colspan="2">Value</td></tr>
</table>

(b) 矩阵非0元素结点

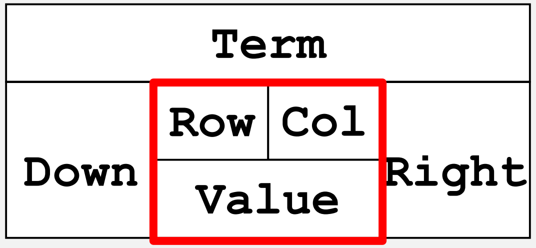
<table>
<tr><td colspan="3">Head</td></tr>
<tr><td>Down</td><td>Next</td><td>Right</td></tr>
</table>

(c) 头结点

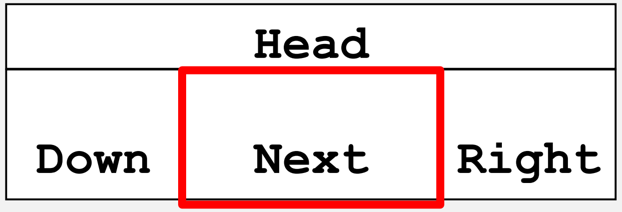
➢ 稀疏矩阵的数据结构可定义为：

```
#define MAXSIZE 50       /* 矩阵最大非0元素个数 */
typedef enum  { Head, Term } NodeTag;

typedef struct  TermNode {  //非零元素结点
        int  Row;
        int  Col;
        ElementType  Value;
};

typedef struct MNode  *PtrToNode;
typedef struct  MNode {  //矩阵结点
        PtrMatrix Down, Right;
        NodeTag Tag;
        union {
                PtrMatrix  Next;
                Struct TermNode  Term;
        }  URegion;
};
```

(a) 结点的总体结构



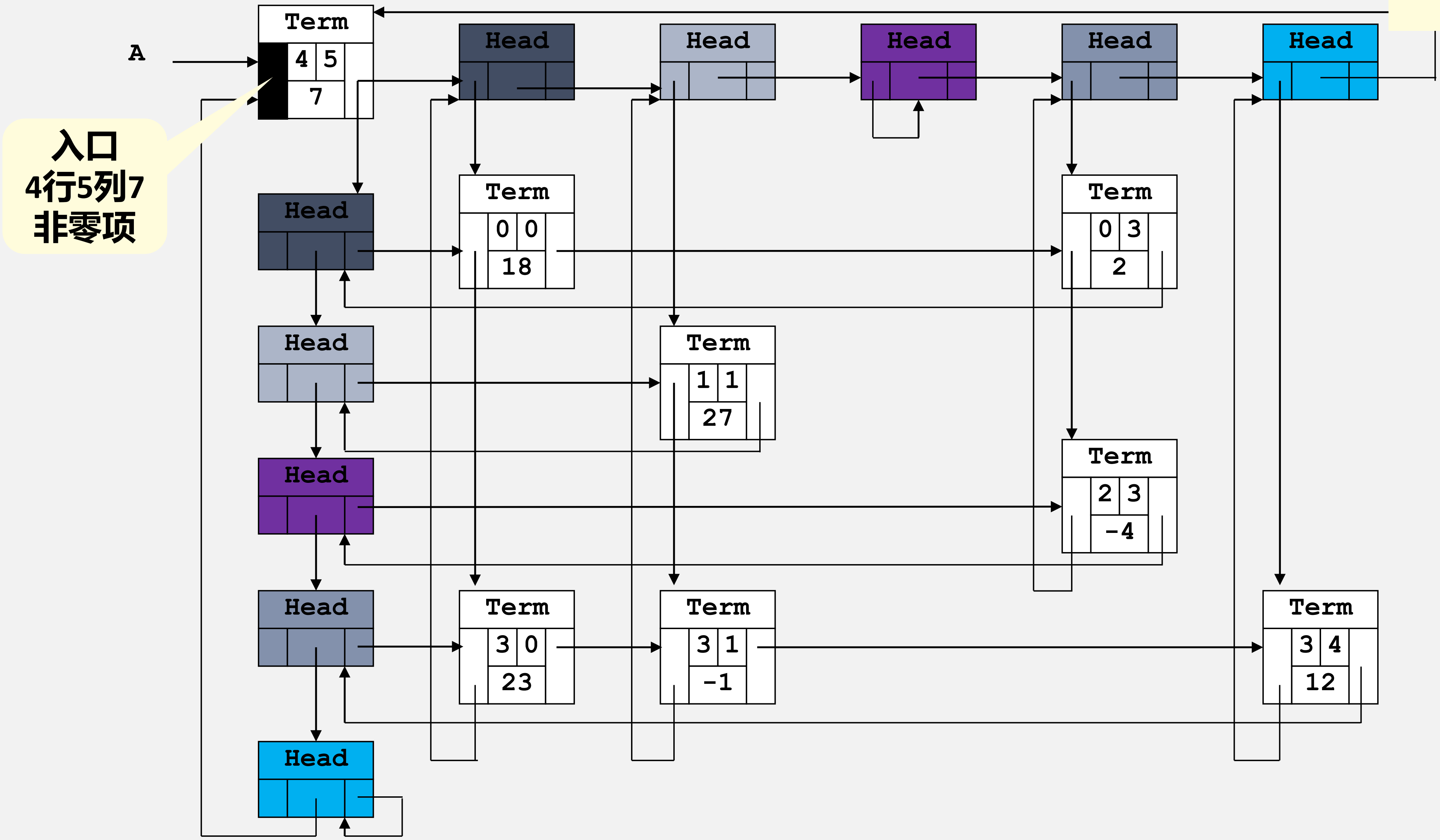
(b) 矩阵非0元素结点



(c) 头结点

❖ 矩阵A的多重链表图

$$A=\begin{bmatrix}18 & 0 & 0 & 2 & 0\\ 0 & 27 & 0 & 0 & 0\\ 0 & 0 & 0 & -4 & 0\\ 23 & -1 & 0 & 0 & 12\end{bmatrix}$$



3.3 一元多项式的加法运算

问题描述：求一元多项式 $Pa(x)$和$Pb(x)$相加的结果$Pc(x)$ 。

方法：采用带头结点的单链表表示多项式，按照指数递减的方式顺序排列各项。

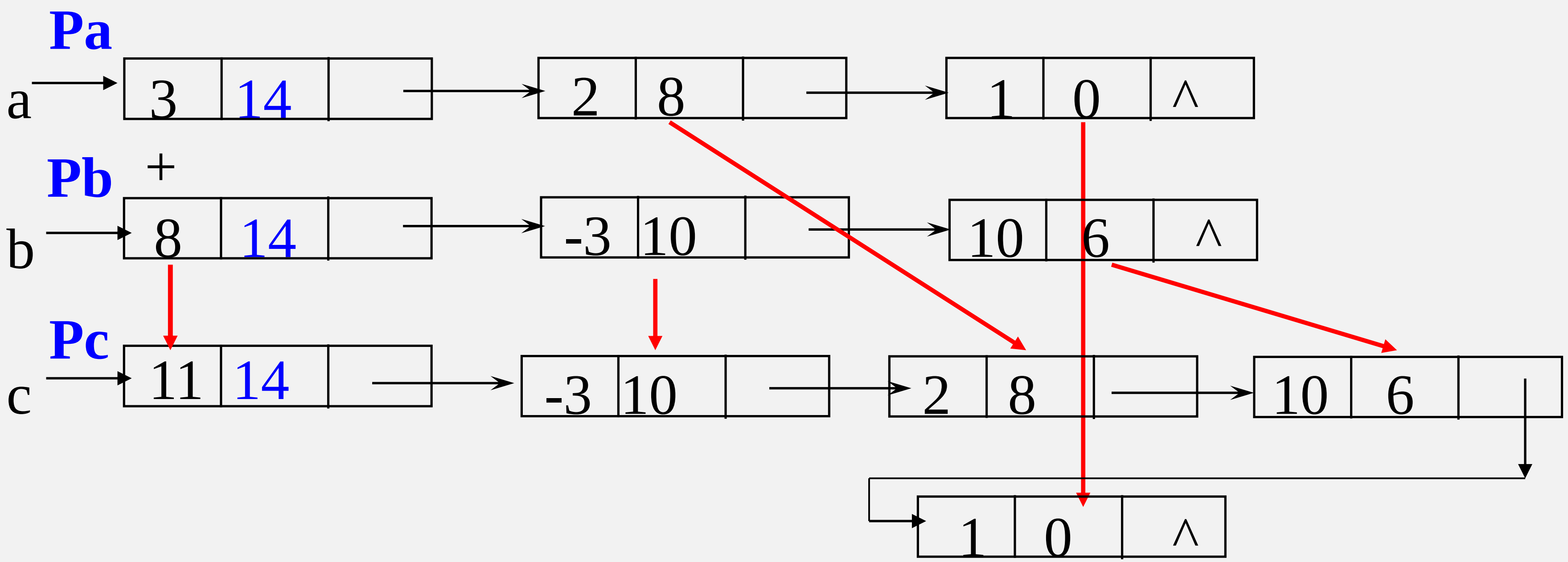
```
Struct  Term
{   float  coef;                          //系数域
    int    exp;                            //指数域
    struct Term   *link;      //指针域
} * Polynomial;
```

coef	exp	link

设计思想：

- 初始情况： Pa和Pb指向两个多项式的第一个结点,Pc为结果多项式
- 依次比较Pa和Pb所指结点的指数项，判断Pa->exp和Pb->exp大小
  - Pa->exp>Pb->exp 摘取Pa 插入Pc中，Pa指针指向下一项
  - Pa->exp<Pb->exp 摘取Pb 插入Pc中，Pb指针指向下一项
  - Pa->exp=Pb->exp Pa Pb的系数相加
    - 如果和不为0，修改Pa的系数，插入Pc，删除Pb结点
    - 为0，删除相应Pa结点，指针指向下一项

【例】 $Pa(x)=3x^{14}+2x^{8}+1$ 和 $Pb(x)=8x^{14}-3x^{10}+10x^{6}$



```
void Add ( Polynomial &A, Polynomial &B, Polynomial& C ) {
  //两个带头结点的按降幂排列的多项式相加，返回结果多项式
  //链表的表头指针 C，结果不另外占用存储，覆盖 A 和 B 链表
    Term *pa, *pb, *pc, *p;   Term a, b;
    C = pc = A;                         //结果存放指针
    pa = A->link;                     //多项式 A 的检测指针
    pb = B->link;                     //多项式 B 的检测指针
    delete B;                           //删去 B 的表头结点
    while ( pa != NULL && pb != NULL ) {
       a = pa->data;  b = pb->data;
       if ( a.exp = = b.exp ) {        //两个多项式指数相等情况
         a.coef = a.coef + b.coef;  //系数相加
            p = pb;   pb = pb->link;   delete p; //删除b结点
        if ( a.coef ) {         //相加不为零, 加入 C 链
                  pa->data = a;   pc->link = pa;  pc = pa; pa = pa->link;}
              else                          //相加为零,该项不要
                  { p = pa;  pa = pa->link;  delete p; }
             }
```

```
else if ( pa->exp > pb->exp ) {//A多项式指数大于B多项式的情况
      pc->link = pa;
      pc = pa;
      pa = pa->link; }
else {                       // A多项式指数小于B多项式的情况
      pc->link = pb;
      pc = pb;
      pb = pb->link;
      }
  }
//剩余部分链入 C 链
 if ( pa != NULL )
     pc->link = pa;
 else
    pc->link = pb;
}
```