2.1 线性表及其实现

多项式的表示

[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】如何表示多项式?

多项式的关键数据：

- 多项式项数**n**
- 各项系数**a_i** 及指数 **i**

方法1：顺序存储结构直接表示

数组各分量对应多项式各项：

a[i]: 项x^i的系数a_i

例如： $$f(x) = 4x^5 - 3x^2 + 1$$

表示成：

下标i	0	1	2	3	4	5	……
a[i]	**1**	**0**	**–3**	**0**	**0**	**4**	**……**
	1		$-3x^2$			$4x^5$	

两个多项式相加： 两个数组对应分量相加

问题：如何表示多项式 $x+3x^{2000}$？

如果用数组表示的话，需要2001项，但是只有两项系数非零
(用一元数组实现会造成空间的浪费)

方法2：顺序存储结构表示非零项

每个非零项 $a_i x^i$ 涉及两个信息：系数 a_i 和指数 i
可以将一个多项式看成是一个 (a_i, i) 二元组的集合。

用结构数组表示：数组分量是由系数a_i、指数i组成的结构，对应一个非零项

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

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

(a) $P_1(x)$

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

(b) $P_2(x)$

按指数大小有序存储！

在这里按照递增的方式有序存储

方法2：顺序存储结构表示非零项

相加过程：从头开始，比较两个多项式当前对应项的指数

P1: (9,12), (15,8), (3,2)

P2: (26,19), (-4,8), (-13,6), (82,0)

P3: (26,19) (9,12) (11,8) (-13,6) (3,2) (82,0)

$$P_3(x) = 26x^{19} + 9x^{12} + 11x^8 - 13x^6 + 3x^2 + 82$$

方法3：链表结构存储非零项

链表中每个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

coef	expon	link

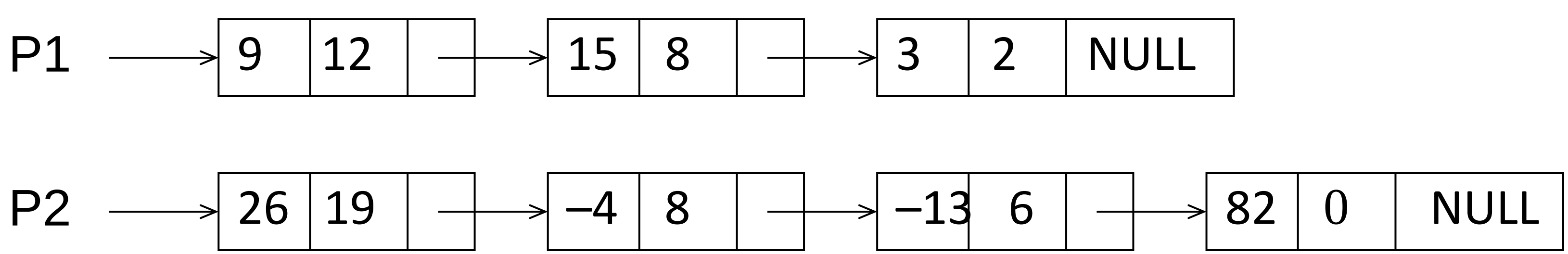
```
typedef struct PolyNode *Polynomial;
struct PolyNode {
        int coef;
        int expon;
        Polynomial link;
}
```

例如：

$$P_1(x) = 9x^{12} + 15x^8 + 3x^2$$

$$P_2(x) = 26x^{19} - 4x^8 - 13x^6 + 82$$

链表存储形式为：



什么是线性表

多项式表示问题的启示：

1. 同一个问题可以有不同的表示（存储）方法
2. 有一类共性问题：有序线性序列的组织和管理

"线性表(Linear List)"：由同类型数据元素构成有序序列的线性结构

- 表中元素个数称为线性表的长度
- 线性表没有元素时，称为空表
- 表起始位置称表头，表结束位置称表尾

线性表的抽象数据类型描述

类型名称：线性表（List）　　数据类型的两个关键要素:数据对象集+操作集

数据对象集：线性表是 n (≥0)个元素构成的有序序列($a_1, a_2, \ldots, a_n$)

操作集：线性表L ∈ List，整数i表示位置，元素X ∈ ElementType，
线性表基本操作主要有：

1、List MakeEmpty()：初始化一个空线性表L；

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

3、int Find(ElementType X, List L)：在线性表L中查找X的第一次出现位置；

4、void Insert(ElementType X, int i, List L)：在位序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。

线性表的顺序存储实现

利用数组的连续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

last用于指示最后一个元素所在的位置

```
typedef struct LNode *List;
struct LNode{
        ElementType  Data[MAXSIZE];
        int  Last;
} ;
struct LNode L;
List  PtrL;
```

访问下标为 i 的元素：L.Data[i] 或 PtrL->Data[i]

线性表的长度：L.Last+1 或 PtrL->Last+1

❖ 主要操作的实现

1. 初始化（建立空的顺序表）

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

> 查找成功的平均比较次数为(n +1)/2，平均时间性能为O(n)。

2. 查找

```
int Find( ElementType X, List PtrL )
{   int i = 0;
    while( i <= PtrL->Last && PtrL->Data[i]!= X )
         i++;
    if (i > PtrL->Last)  return -1; /* 如果没找到，返回-1 */
    else  return i;              /* 找到后返回的是存储位置 */
}
```

3. 插入（第 *i (1≤i≤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

注意:移动的顺序应该是从后往前移动

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

Last

3. 插入操作实现

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

代码的健壮性与严谨性，值得学习

平均移动次数为 $n/2$，
平均时间性能为 $O(n)$

4. 删除（删除表的第 i $(1\le i\le n)$ 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

注意:移动的顺序应该是从前往后移动

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

Last

4. 删除操作实现

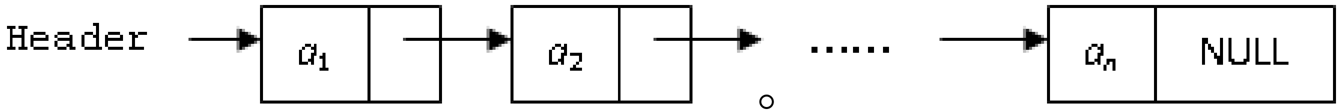
```
void Delete( int  i, List PtrL )
{     int  j;
      if( i < 1 || i > PtrL->Last+1 ) {
           printf ("不存在第%d个元素",
           return ;
      }
      for ( j = i; j <= PtrL->Last; j++ )
           PtrL->Data[j-1] = PtrL->Data[j];      /*将 a_{i+1}～ a_n顺序向前移动*/
      PtrL->Last--;                              /*Last仍指向最后元素*/
      return;
}
```

平均移动次数为 (n-1) /2，
平均时间性能为 O(n)

线性表的链式存储实现

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

- 插入、删除不需要移动数据元素，只需要修改“链”。




```
typedef struct LNode *List;
struct LNode{
      ElementType  Data;
      List  Next;
};
struct Lnode L;
List    PtrL;
```

访问序号为 i 的元素？
求线性表的长度？

❖ 主要操作的实现

1.求表长

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

时间性能为 O(n)。

2. 查找

（1）按序号查找: FindKth;

```
List  FindKth( int K, List PtrL )
{     List  p = PtrL;
      int  i = 1;
      while (p !=NULL && i < K ){
            p = p->Next;
            i++;
      }
      if ( i == K ) return p;
                /* 找到第K个，返回指针 */
      else  return NULL;
                     /* 否则返回
}
```

（2）按值查找: Find

```
List Find( ElementType X, List PtrL )
{
    List  p = PtrL;
    while ( p!=NULL && p->Data != X )
         p = p->Next;
    return p;
}
```

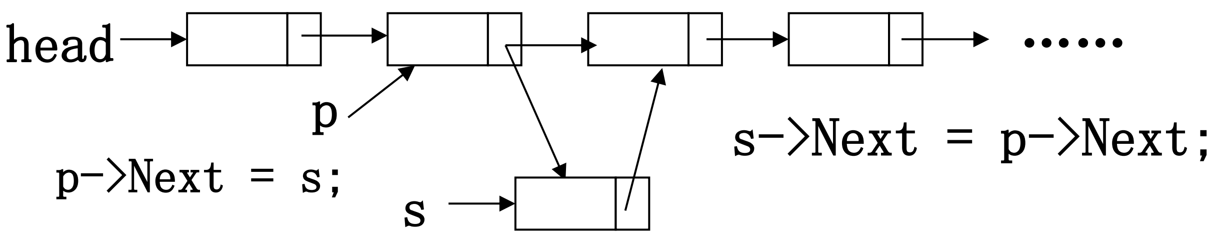
平均时间性能为 O(n)

3. 插入（在第 *i-1(1≤i≤n+1)*个结点后插入一个值为X的新结点)

（1）先构造一个新结点，用s指向；

（2）再找到链表的第 *i-1*个结点，用p指向；

（3）然后修改指针，插入结点 (p之后插入新结点是 s)



思考：修改指针的两个步骤如果交换一下，将会发生什么？

3. 插入操作实现

```
List Insert( ElementType X, int i, List PtrL )
{       List  p, s;
        if ( i == 1 ) {  头节点特殊处理                        /* 新结点插入在表头 */
            s = (List)malloc(sizeof(struct LNode));            /*申请、填装结点*/
            s->Data = X;
            s->Next = PtrL;
            return s;
        }
        p = FindKth( i-1, PtrL );                              /* 查找第i-1个结点 */
        if ( p == NULL ) {                                     /* 第i-1个不存在，不能插入 */
                printf( " 参数i错 " );
                return NULL;
        }else {
              s = (List)malloc(sizeof(struct LNode));          /*申请、填装结点*/
              s->Data = X;
              s->Next = p->Next;                               /*新结点插入在第i-1个结点的后面*/
              p->Next = s;
              return PtrL;
        }
}
```

平均查找次数为 n /2，平均时间性能为 O(n)

4. 删除（删除链表的第 *i (1≤i≤n)*个位置上的结点)

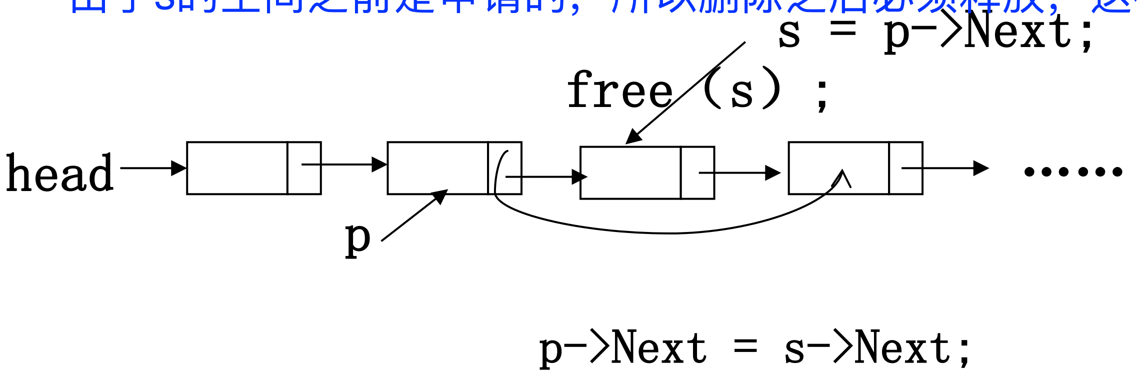
（1）先找到链表的第 *i-1*个结点，用p指向；

（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。

由于s的空间之前是申请的，所以删除之后必须释放，这样不会造成内存的泄露



思考：操作指针的几个步骤如果随意改变，将会发生什么?

4. 删除操作实现

```
List Delete( int  i, List PtrL )
{       List  p, s;
        if ( i == 1 ) {                          /* 若要删除的是表的第一个结点 */
             s = PtrL;                                        /*s指向第1个结点*/
             if (PtrL!=NULL)  PtrL = PtrL->Next;              /*从链表中删除*/
             else return NULL;
             free(s);
             return PtrL;
        }
        p = FindKth( i-1, PtrL );                         /*查找第i-1个结点*/
        if ( p == NULL ) {
              printf(“第%d个结点不存在”, i-1);     return NULL;
       } else  if ( p->Next == NULL ){
              printf(“第%d个结点不存在”, i);      return NULL;
       } 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 PtrL;
       }
}
```

平均查找次数为 $n/2$，
平均时间性能为 $O(n)$

广义表

[例] 我们知道了一元多项式的表示，那么二元多项式又该如何表示？

比如，给定二元多项式：$P(x,y)=9x^{12}y^2+4x^{12}+15x^8y^3-x^8y+3x^2$

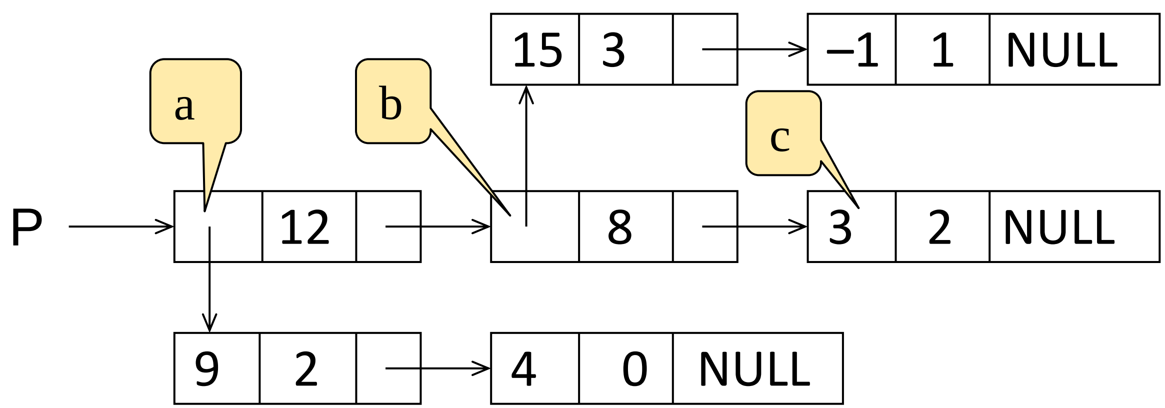
【分析】 可以将上述二元多项式看成关于x 的一元多项式

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

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

此时x的系数又可以表示成新的线性表

所以，上述二元多项式可以用“复杂”链表表示为：



广义表(Generalized List)

- 广义表是线性表的推广
- 对于线性表而言， n个元素都是基本的单元素；
- 广义表中，这些元素不仅可以是单元素也可以是另一个广义表。

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

<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

多重链表：链表中的节点可能同时隶属于多个链

- 多重链表中结点的指针域会有多个，如前面例子包含了**Next**和**SubList**两个指针域；
- 但包含两个指针域的链表并不一定是多重链表，比如在双向链表不是多重链表。

- 多重链表有广泛的用途：
  基本上如树、图这样相对
  复杂的数据结构都可以采
  用多重链表方式实现存储。

[例] 矩阵可以用二维数组表示，但二维数组表示有两个缺陷：

- 一是数组的大小需要事先确定，
- 对于“稀疏矩阵”，将造成大量的存储空间浪费。

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

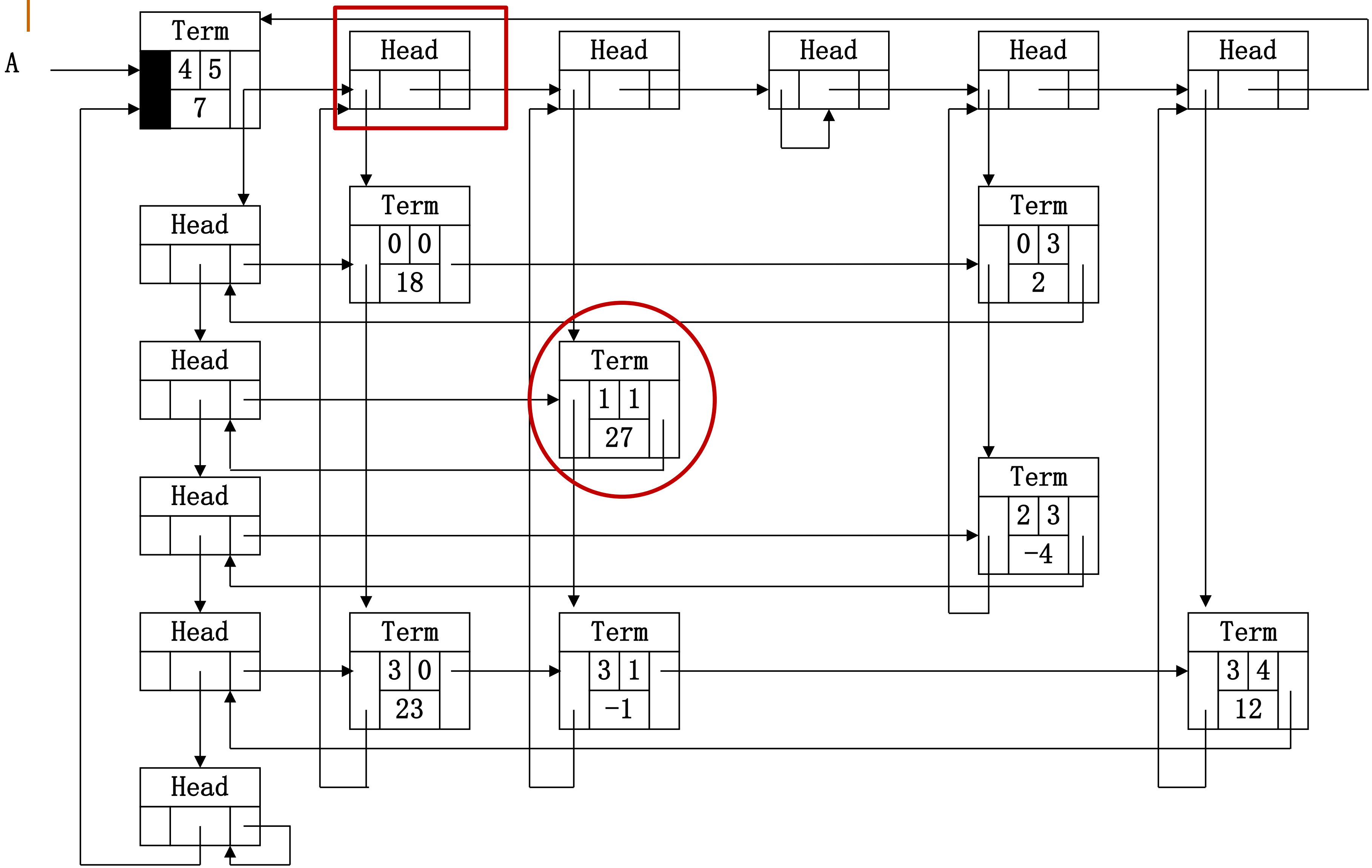
【分析】 采用一种典型的多重链表——十字链表来存储稀疏矩阵

- 只存储矩阵非0元素项

  结点的数据域：行坐标Row、列坐标Col、数值Value

- 每个结点通过两个指针域，把同行、同列串起来；
  - 行指针(或称为向右指针)Right
  - 列指针（或称为向下指针）Down

❖ 矩阵A的多重链表图


A
Term
4 5
7
Head
Head
Head
Head
Head
Head
Term
0 0
18
Term
0 3
2
Head
Term
1 1
27
Head
Term
2 3
−4
Head
Term
3 0
23
Term
3 1
−1
Term
3 4
12
Head

- 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”。

<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