

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： $f(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1} + a_nx^n$

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 $x+3x^{2000}$?

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！

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
------	-------	------

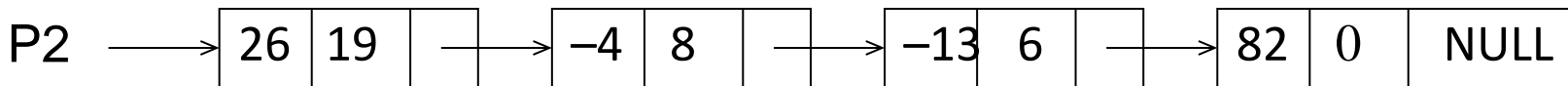
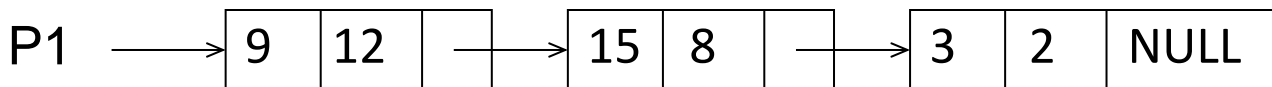
```
typedef struct PolyNode *Polynomial;  
typedef struct PolyNode {  
    int coef;  
    int expon;  
    Polynomial link;  
}
```

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**）

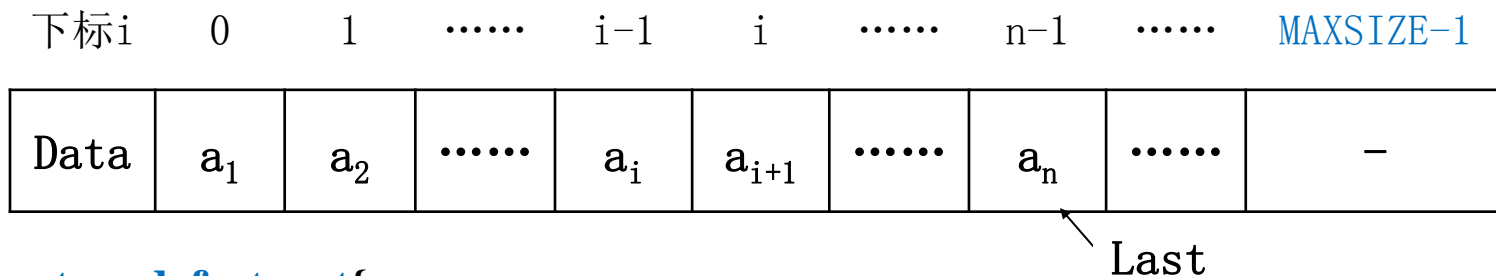
数据对象集：线性表是 n (≥ 0) 个元素构成的有序序列(a_1, a_2, \dots, a_n)

操作集：线性表 $L \in \text{List}$ ，整数 i 表示位置，元素 $X \in \text{ElementType}$ ，
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



```
typedef struct{
    ElementType Data[MAXSIZE];
    int Last;
} List;
```

List L, *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(List) );
    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
 $O(n)$ 。

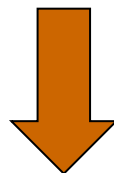
2. 查找

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

下标 i	0	1	$i-1$	i	$n-1$	$\text{MAXSIZE}-1$
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	$\text{SIZE}-1$
Data	a_1	a_2	X	a_i	a_{i+1}	a_n	-

↖
Last

3. 插入操作实现

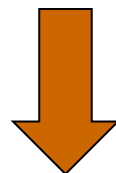
```
void Insert( ElementType X, int i, List *PtrL )
{
    int j;
    if ( PtrL->Last == MAXSIZE-1 ){           /* 表空间已满，不能插入*/
        printf( " 表满 " );
        return;
    }
    if ( i < 1 || i > PtrL->Last+2 ) {
        printf( " 位置不合法 " );
        return;
    }
    for ( j = PtrL->Last; j >= i-1; j-- )
        PtrL->Data[j+1] = PtrL->Data[j]; /*将  $a_i \sim a_n$  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;
}
```

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 ($1 \leq i \leq n$) 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	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”, i);
        return ;
    }
    for ( j = i; j <= PtrL->Last; j++)
        PtrL->Data[j-1] = PtrL->Data[j];
    PtrL->Last--;
    return;
}
```

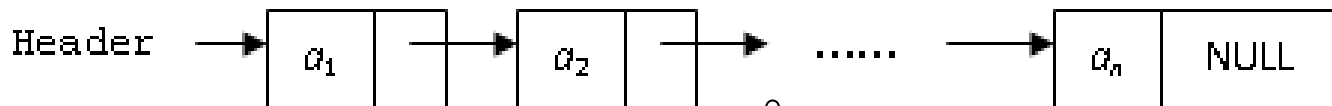
平均移动次数为 $(n-1)/2$,
平均时间性能为 $O(n)$

/*将 $a_{i+1} \sim a_n$ 顺序向前移动*/
/*Last仍指向最后元素*/

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 Node{  
    ElementType Data;  
    struct Node *Next;  
} List;
```

```
List L, *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NULL */
}
```

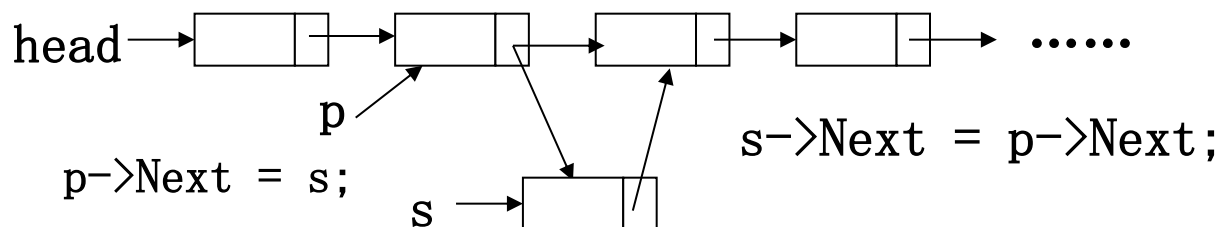
(2) 按值查找: Find

```
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 $i-1$ ($1 \leq i \leq n+1$) 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 ) {
        s = (List *)malloc(sizeof(List));
        s->Data = X;
        s->Next = PtrL;
        return s;
    }
    p = FindKth( i-1, PtrL );
    if ( p == NULL ) {
        printf( " 参数i错 " );
        return NULL;
    }
    else {
        s = (List *)malloc(sizeof(List));
        s->Data = X;
        s->Next = p->Next;
        p->Next = s;
        return PtrL;
    }
}
```

/* 新结点插入在表头 */
/* 申请、填装结点 */

平均查找次数为 $n/2$ ，平均
时间性能为 $O(n)$

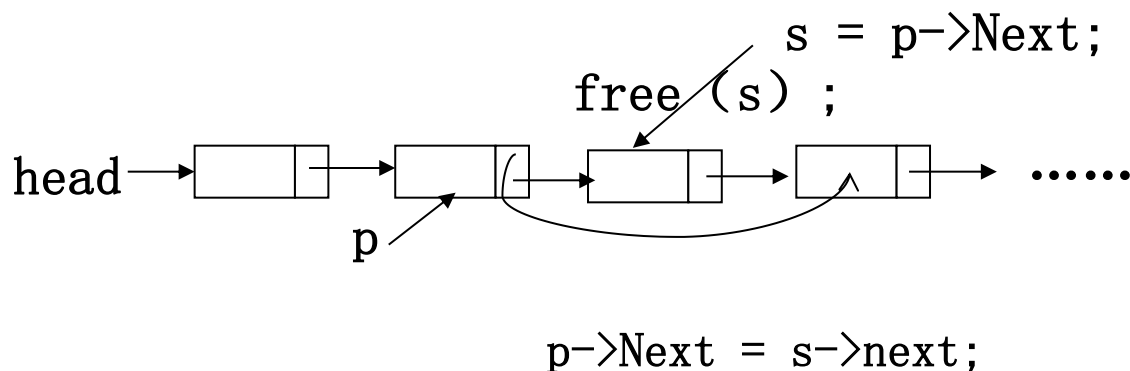
/* 查找第i-1个结点 */
/* 第i-1个不存在，不能插入 */

/* 申请、填装结点 */

/* 新结点插入在第i-1个结点的后面 */

4. 删除 (删除链表的第 i ($1 \leq i \leq n$) 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。



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 ) {
        s = PtrL;
        if (PtrL!=NULL) PtrL = PtrL->Next;
        else return NULL;
        free(s);
        return PtrL;
    }
    p = FindKth( i-1, PtrL );
    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;
        p->Next = s->Next;
        free(s);
        return PtrL;
    }
}
```

/* 若要删除的是表的第一个结点 */
/*s指向第1个结点*/
/*从链表中删除*/

平均查找次数为 $n/2$,
平均时间性能为 $O(n)$

/*查找第i-1个结点*/

/*s指向第i个结点*/
/*从链表中删除*/
/*释放被删除结点 */

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？

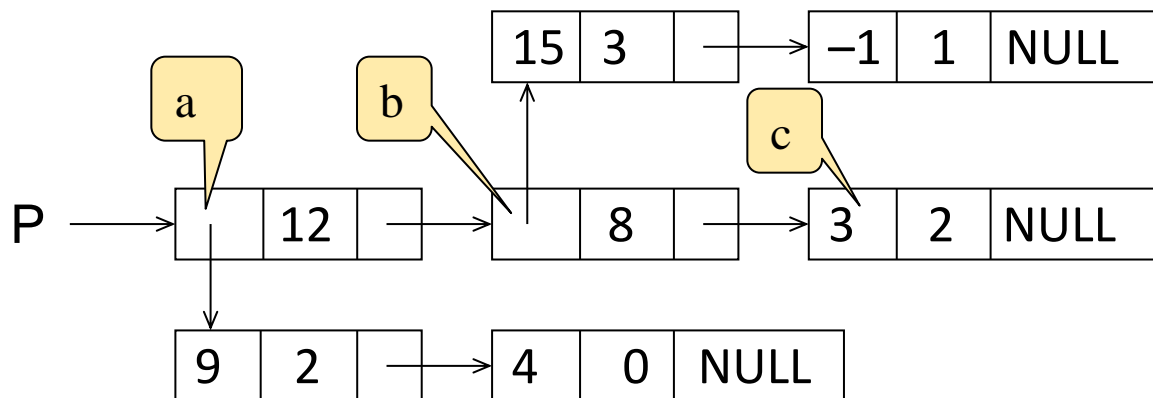
比如，给定二元多项式： $P(x, y) = 9x^{12}y^2 + 4x^{12} + 15x^8y^3 - x^8y + 3x^2$

【分析】 可以将上述二元多项式看成关于 x 的一元多项式

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

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

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{  
    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;  
        struct GNode *SubList;  
    } URegion;  
    struct GNode *Next; /*指向后继结点*/  
} GList;
```

Tag	Data	Next
	SubList	

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**。

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

【分析】 采用一种典型的多重链表——**十字链表**来存储稀疏矩阵

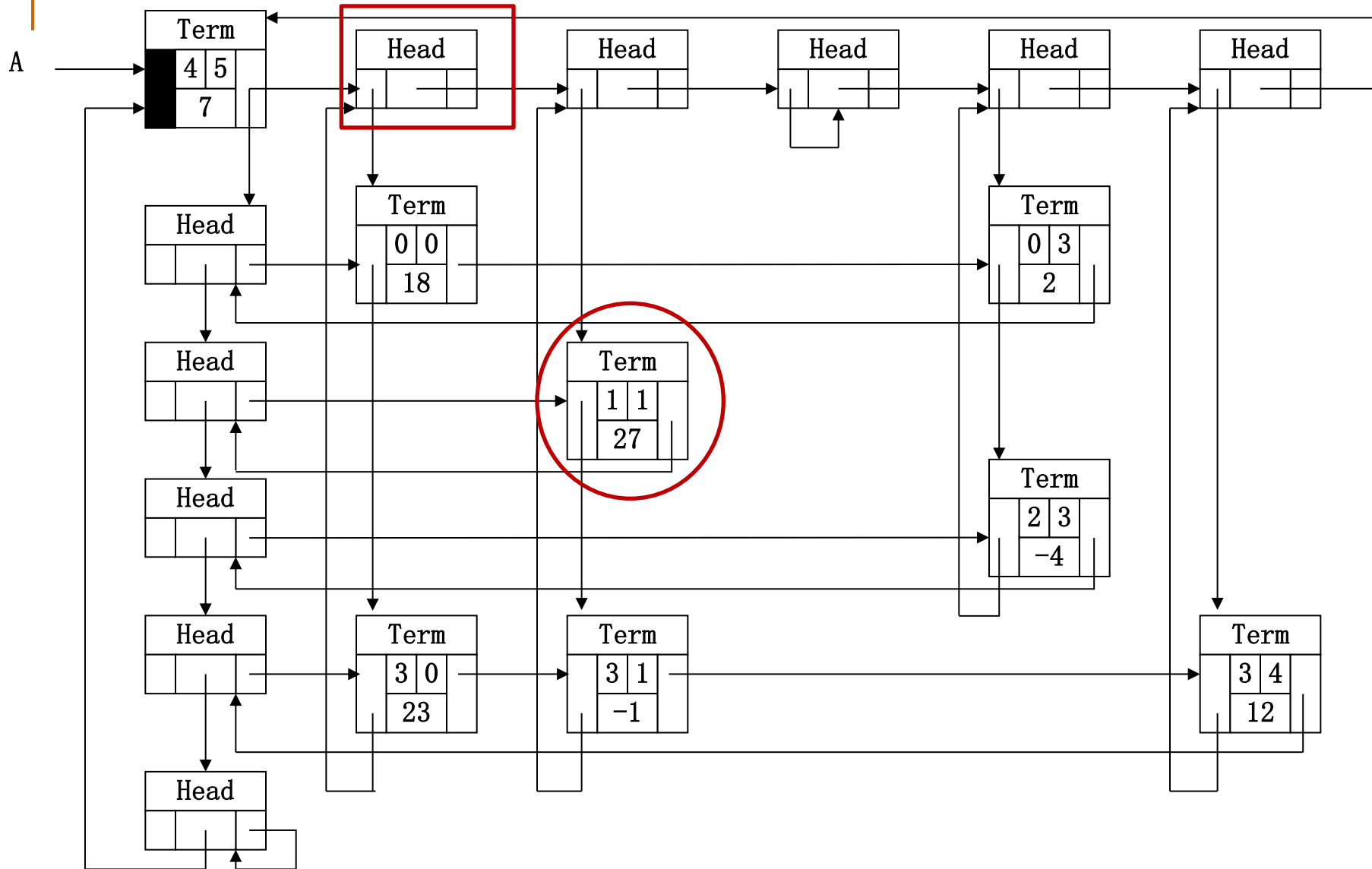
□ 只存储矩阵非0元素项

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



□ 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”。

Tag		
Down	<i>URegion</i>	Right

(a) 结点的总体结构

Term			
Down	Row	Col	Right
	Value		

(b) 矩阵非0元素结点

Head		
Down	Next	Right

(c) 头结点