数据结构与算法

陈越编写

高等教育出版社出版

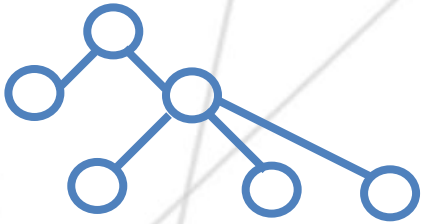
讲 解 人：张家琦

日　　期：2022.09

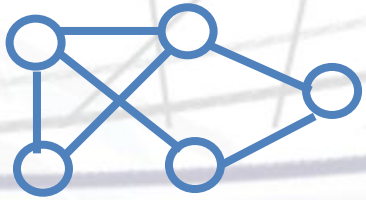
- 集合——数据元素间除“同属于一个集合”外，无其它关系

- 线性结构——一个对一个，如线性表、栈、队列

- 树形结构——一个对多个，如树



- 图形结构——多个对多个，如图



[例3.1] 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】多项式的关键数据是：多项式项数**n** 、每一项的系数a_i（及相应指数 **i**）。有3种不同的方法。

方法1：采用顺序存储结构直接表示

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	……

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 $a_i x^i$ 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。

例如： $P_1(x)=9x^{12}+15x^8+3x^2$ 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) $P_2(x)$

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	Expon	Next

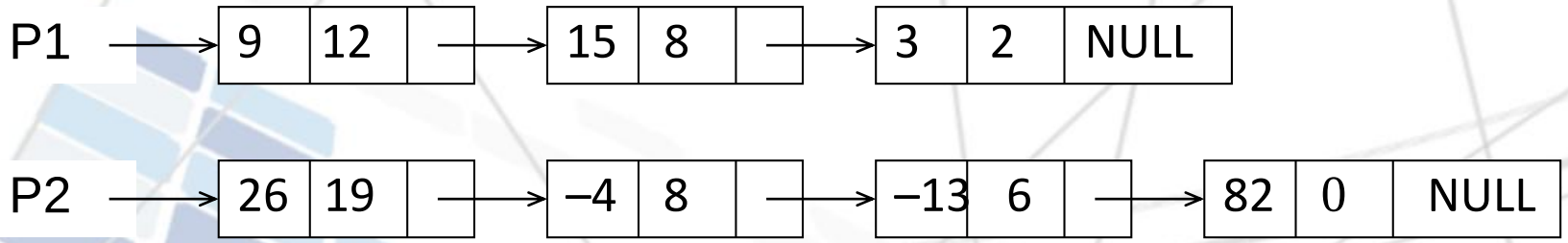
```
typedef struct PolyNode *PtrToPolyNode;
typedef struct PolyNode {
        int Coef;
        int Expon;
        PtrToPolyNode Next;
}
typedef PtrToPolyNode Polynomial;
```

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：



[例3.2] 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

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

所以，上述二元多项式可以用“复杂”链表表示为：

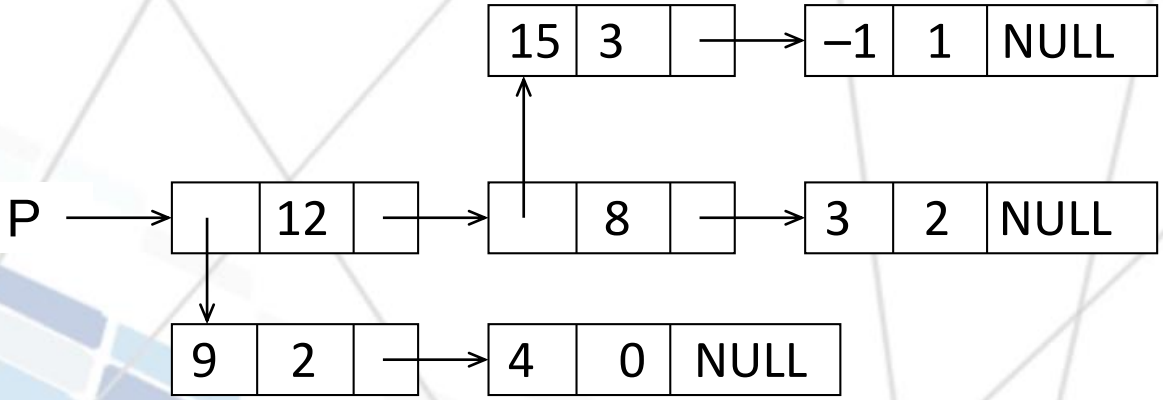


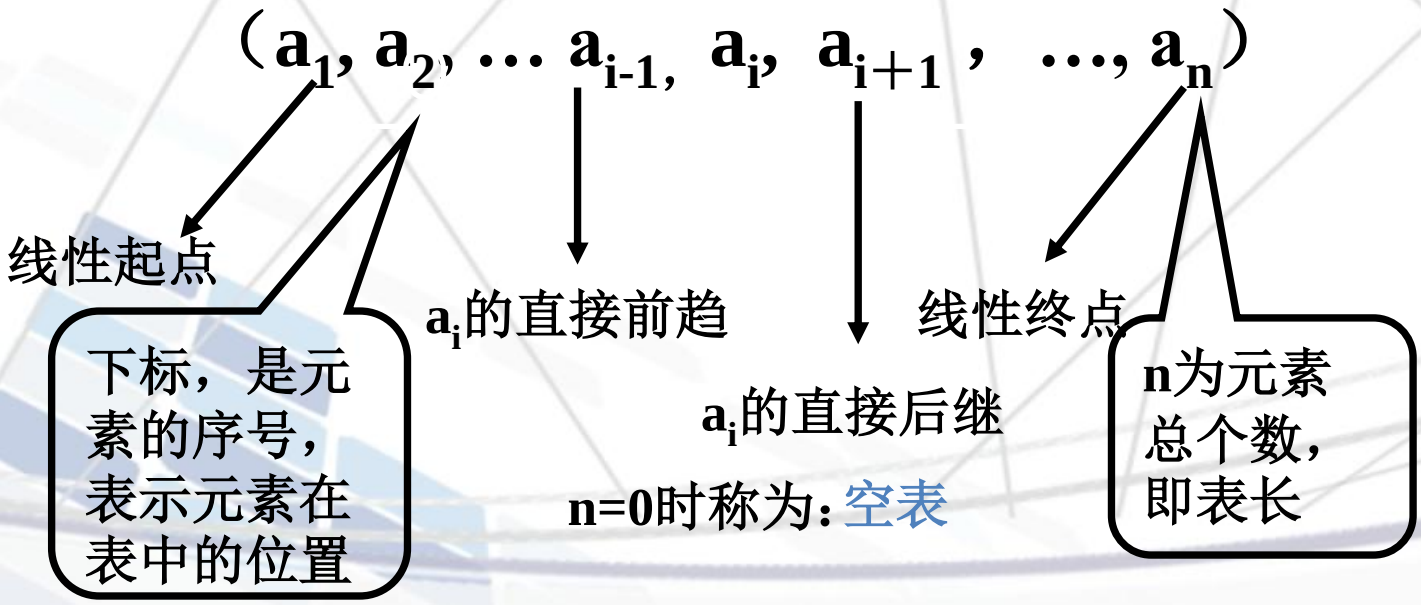
图3.4 二元多项式非零项的链表表示

【定义】“线性表(Linear List)”是由同一类型的数据元素构成的有序序列的线性结构。

- 线性表中元素的个数称为线性表的长度；
- 当一个线性表中没有元素（长度为0）时，称为空表；
- 表的起始位置称表头，表的结束位置称表尾。

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($a_1, a_2, \ldots, a_n$)； a_{i+1}称为 a_i 的直接后继， a_{i-1}为 a_i的直接前驱；直接前驱和直接后继反映了元素之间一对一的邻接逻辑关系。



操作集：对于一个具体的线性表L ∈ List，一个表示位置的整数i，一个元素X ∈ ElementType，线性表的基本操作主要有：

1、List MakeEmpty()：初始化一个新的空线性表L；

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

3、Position Find(List L, ElementType X)：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、bool Insert(List L, ElementType X, int i)：指定位序i前插入一个新元素X；成功则返回true，否则返回false；

5、bool Delete(List L, int i)：删除指定位序i的元素；成功则返回true，否则返回false；

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

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。一维数组在内存中占用的存储空间就是一组连续的存储区域。

```
typedef  int Position;
```

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 LNode *PtrToLNode;
struct LNode{
   ElementType  Data[MAXSIZE];
   Position  Last;
};
typedef  PtrToLNode 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;
}
```

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

```
#define ERROR -1
Position Find( List L, ElementType X )
{     Position i = 0;
      while( i <= L->Last && L->Data[i]!= X )  i++;
      if ( i > L->Last )
            return ERROR; /* 如果没找到，返回错误信息 */
      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

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

❖ 插入算法

```
bool Insert( List L, ElementType X, int i )
{   Position 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 ) {
 /* 检查插入位序的合法性：是否在1~n+1。n为当前元素个数，即Last+1 */
            printf("位序不合法");
            return false;
    }
    for( j=L->Last; j>=i-1; j-- ) /*Last指向序列最后元素 */
            L->Data[j+1] = L->Data[j]; /* 将位序i及以后的元素顺序向后移动 */
    L->Data[i-1] = X;  /* 新元素插入第i位序，其数组下标为i-1 */
    L->Last++;      /* Last仍指向最后元素 */
    return true;
}
```

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

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

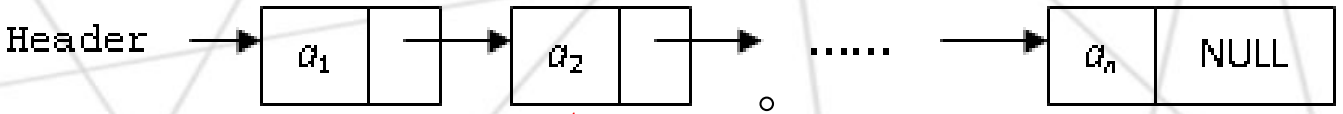
❖ 删除算法

```
bool Delete( List L, int i )
{   Position j;
    if( i<1 || i>L->Last+1 ) { /* 检查空表及删除位序的合法性 */
        printf("位序%d不存在元素", i );
        return false;
    }
    for( j=i; j<=L->Last; j++ )
        L->Data[j-1] = L->Data[j]; /*将位序i+1及以后的元素顺序向前移动*/
    L->Last--; /* Last仍指向最后元素 */
    return true;
}
```

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数据元素物理上也相邻，它是通过“链”建立起数据元素之间的逻辑关系。因此对线性表的插入、删除不需要移动数据元素，只需要修改“链”。




```
typedef  struct LNode *PtrToLNode;
struct LNode{
     ElementType  Data;
     PtrToLNode Next;
};
typedef  PtrToLNode Position;
typedef  PtrToLNode List;
List L;
```

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 L )
{
        Position p;
        int cnt = 0; /* 初始化计数器 */

        p = L;  /* p指向表的第一个结点 */
        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;

```
#define ERROR -1
ElementType FindKth( List L, int K )
{   Position p;
    int cnt = 1; /* 位序从1开始 */
    p = L; /* p指向L的第1个结点 */
    while ( p && cnt<K ) {
            p = p->Next;
            cnt++;
    }
    if ( (cnt==K) && p )
            return p->Data; /* 找到第K个 */
    else
            return ERROR;   /* 否则返回错误信息 */
}
```

（2）按值查找: Find

```
#define ERROR NULL
Position Find( List L, ElementType X )
{   Position p = L;
    /* p指向L的第1个结点 */
    while ( p && p->Data!=X )
            p = p->Next;
    /* 下列语句可以用 return p; 替换 */
    if ( p )
            return p;
    else
            return ERROR;
}
```

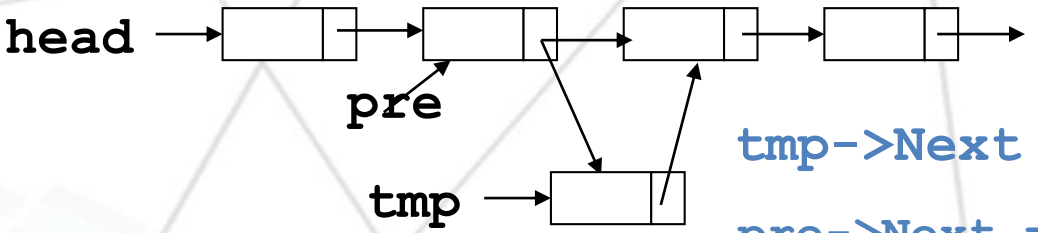
平均时间性能为 O(n)。

3. 插入（在链表的第 *i(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pre指向；

（3）然后修改指针，插入结点 (pre之后插入新结点是 tmp)




```
tmp->Next = pre->Next;
pre->Next = tmp;
```

思考：修改指针的两个步骤如果交换一下，将会发生什么？

```
#define ERROR NULL /* 用空地址表示错误 */
List Insert( List L, ElementType X, int i )
{   Position tmp, pre;
    tmp = (Position)malloc(sizeof(struct LNode)); /* 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 { /* 查找位序为i-1的结点 */
        int cnt = 1;  /* 位序从1开始 */
        pre = L;      /* pre指向L的第1个结点 */
        while ( pre && cnt<i-1 ) {    pre = pre->Next;    cnt++;
        }
        if ( pre==NULL || cnt!=i-1) { /* 所找结点不在L中 */
                printf("插入位置参数错误\n");
                free(tmp);    return ERROR;
        }
        else { /* 找到了待插结点的前一个结点pre */
                tmp->Next = pre->Next;
                pre->Next = tmp;
               return L;
        } }
}
```

空表时的插入要作为特例，
除非单链表带虚头结点。

❖ 插入算法（带头结点）

```
bool Insert( List L, ElementType X, int i )
{ /* 这里默认L有头结点 */
   Position tmp, pre;
   int cnt = 0;
   /* 查找位序为i-1的结点 */
   pre = L;     /* pre指向表头 */
   while ( pre && cnt<i-1 ) {    pre = pre->Next;    cnt++;    }
   if ( pre==NULL || cnt!=i-1) { /* 所找结点不在L中 */
            printf("插入位置参数错误\n");
            return false;
   }
   else { /* 找到了待插结点的前一个结点pre；若i为1，pre就指向表头 */
            /* 插入新结点 */
            tmp=(Position)malloc(sizeof(struct LNode)); /*申请、填装结点*/
            tmp->Data = X;
            tmp->Next = pre->Next;
            pre->Next = tmp;
            return true;
   }
}
```

4. 删除（删除链表的第 *i (1≤i≤n)*个位置上的结点)

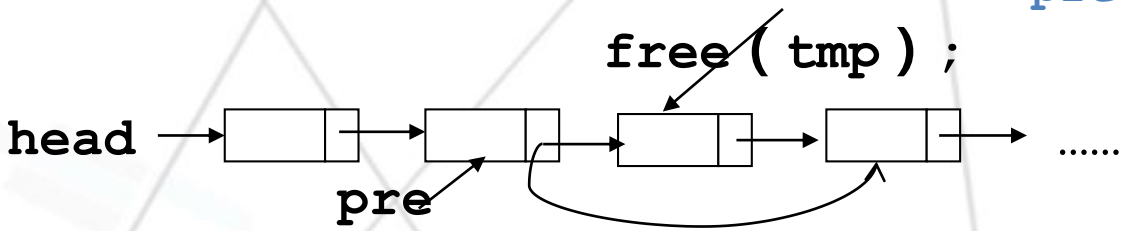
（1）先找到链表的第 *i-1*个结点，用pre指向；

（2）再用指针tmp指向要被删除的结点（pre的下一个结点）；

（3）然后修改指针，删除tmp所指结点;

（4）最后释放tmp所指结点的空间。

```
tmp = pre->Next;
pre->Next = tmp->Next;
```



思考：操作指针的几个步骤如果随意改变，将会发生什么？

❖ 删除算法（带头结点）

```
bool Delete( List L, int i )
{  /* 这里默认L有头结点 */
   Position tmp, pre;
   int cnt = 0;
   /* 查找位序为i-1的结点 */
   pre = L;     /* pre指向表头 */
   while ( pre && cnt<i-1 ) {
      pre = pre->Next;
      cnt++;
   }
   if ( pre==NULL || cnt!=i-1 ||pre->Next==NULL) {
   /* 所找结点或位序为i的结点不在L中 */
      printf("删除位置参数错误\n");
      return false;
   } else { /* 找到了待删结点的前一个结点pre */
      /* 将结点删除 */
      tmp=pre->Next;
      pre->Next=tmp->Next;
      free(tmp);
      return true;
   }
}
```

平均查找次数为 n /2，
平均时间性能为 O(n)。

❖ 广义表

【例】如何表示一个单位的人员情况。一种简单的表示方法是用一个线性表来表示，其先后顺序按照进单位的时间顺序排列：

（张三，李四，王五，钱六，孙七，……）

- 如何体现三个不同部门？比如办公室、生产部、销售部。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，……））

【定义】上述这类表就是一种“广义表(Generalized List)”。

- 广义表是线性表的推广。
- 广义表与线性表一样，也是 由n个元素组成的有序序列。
- 不同点在于，对于线性表而言， n个元素都是基本的单元素；
- 而在广义表中，这些元素不仅可以是单元素也可以是另一个广义表。

❖ 广义表的数据结构可以定义如下：

```
typedef struct GNode *PtrToGNode;
typedef PtrToGNode GList;
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 {  /* 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;
   PtrPtrToGNode Next; /* 指向后继结点 */
};
```

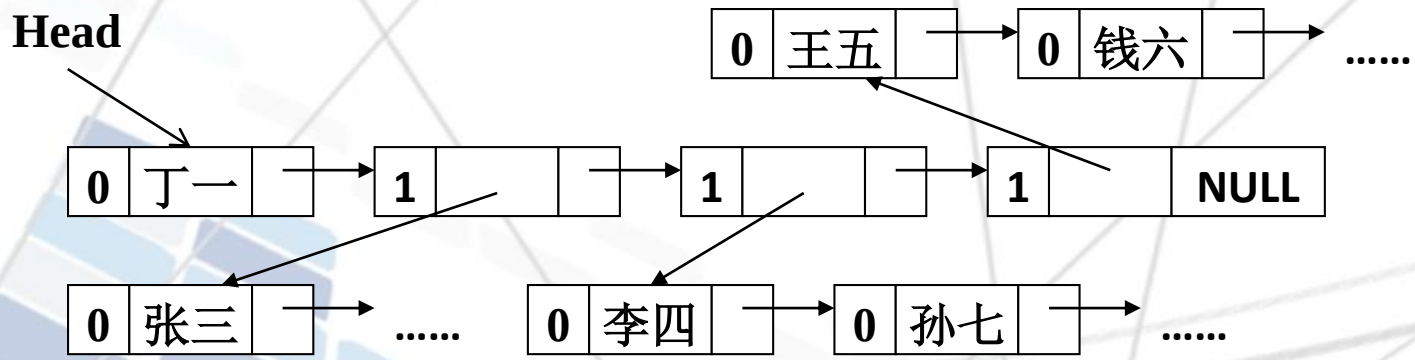


图3.7 广义表结构

❖ 多重链表

【定义】在图 3 .7的例子中，广义表采用链表存储的方式实现。像这种链表，其元素可能还是另一个子链表的起点指针，叫“多重链表”。

- 一般来说，多重链表中每个结点的指针域会有多个，如前面的例子包含了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。

- 多重链表在数据结构实现中有广泛的用途，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的方式实现存储。

[例3.3] 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，
- 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。
- 例如，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？

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

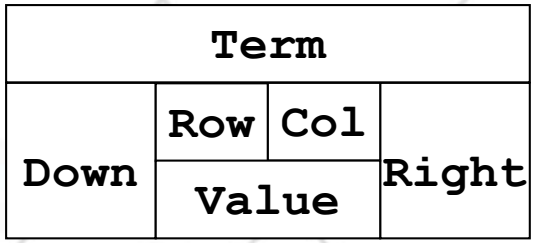
$$B=\begin{bmatrix}0 & 2 & 11 & 0 & 0 & 0\\ 3 & -4 & -1 & 0 & 0 & 0\\ 0 & 0 & 0 & 9 & 13 & 0\\ 0 & -2 & 0 & 0 & 10 & 7\\ 6 & 0 & 0 & 5 & 0 & 0\end{bmatrix}$$

【分析】 采用一种典型的多重链表——十字链表来存储稀疏矩阵。

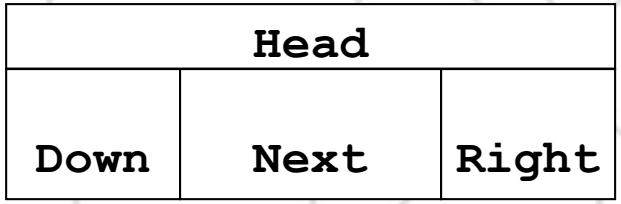
- 链表中用于存放矩阵非0元素的每个结点有两个指针域;
- 一个是行指针(或称为向右指针)Right，
- 另一个是列指针（或称为向下指针）Down，
- 结点的数据域存放元素的行坐标Row、列坐标Col和数值Value。

- 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”。

(a) 结点的总体结构



(b) 矩阵非0元素结点



(c) 头结点

- 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”。

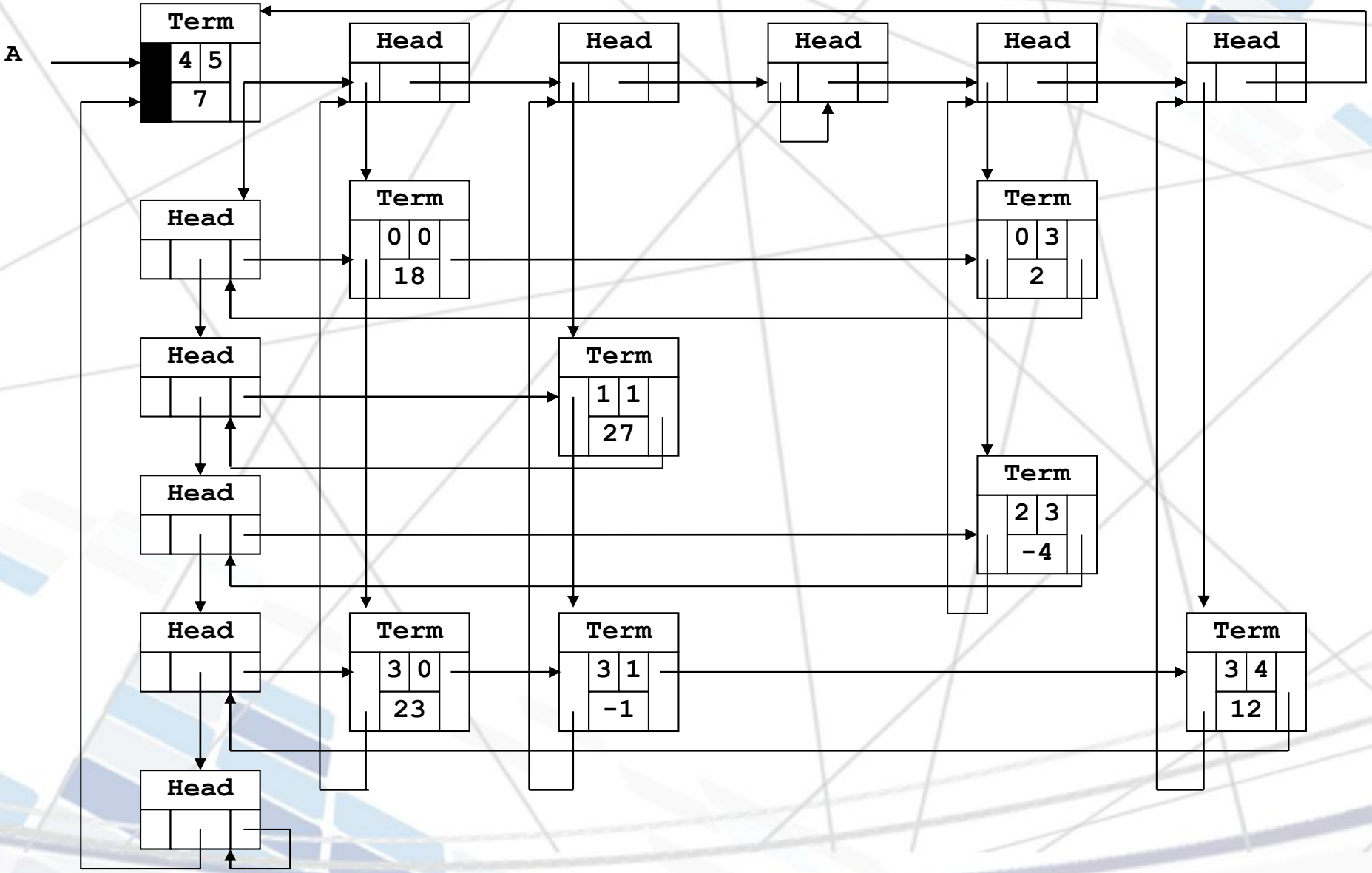
- 稀疏矩阵的数据结构可定义为：

```
typedef enum {Head, Term} NodeTag;

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

typedef struct MNode *PtrToMNode;
struct MNode { /* 矩阵结点定义 */
        PtrToMNode Down, Right;
        NodeTag Tag;
        union { /* Head对应Next指针；Term对应非零元素结点 */
                PtrToMNode Next;
                struct TermNode Term;
        } URegion;
};
typedef PtrToMNode Matrix; /* 稀疏矩阵类型定义 */
Matrix  HeadNode [MAXSIZE]; /* MAXSIZE是矩阵最大非0元素个数 */
```

❖ 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
Head
Term
3 0
23
Term
3 1
-1
Term
3 4
12
Head