Part 1 线性表及其实现

例1 多项式的表示

$$f(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1} + a_n x^n \tag{1}$$

- 多项式的主要运算有 相加，相减，相乘等
- 多项式的关键数据有 多项式的项数n，各项的系数a_i及对应的指数i

• 方法1：顺序存储结构直接表示

`a[i]` 表示项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		-3x²			4x⁵	

多项式相加直接由两个数组对应位置相加即可

缺点：我们如果用这个方法表示一个系数为0的项数过多的多项式，会很浪费空间，例如$x + 9x^{9000}$

• 方法2：顺序存储结构只表示非零项

我们定义一种结构体（二元组）来表示多项式非零项，包含两个属性，即系数与指数；这样，定义一个对应的结构数组就可以表示整个多项式，但是要注意这个结构数组的存储需要按照指数的递增或递减顺序存放，以便进行加减

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) $P_2(x)$

这里假设是按照指数递减的顺序存储的结构体，两个多项式相加，只需要从头开始对比，指数大的直接放入结果，指数相同的系数相加再放入结果即可，例如

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)

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

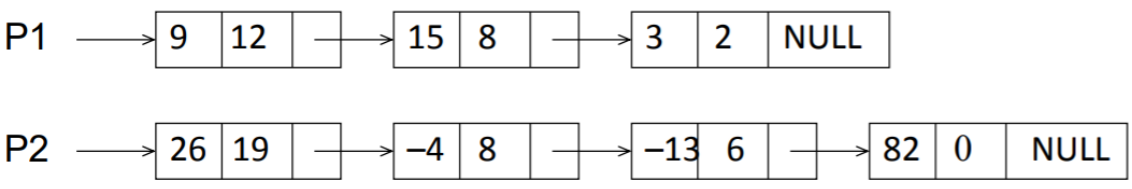
• 方法3：链表结构存储非零项

链表的每个节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和一个指数域

```
typedef struct PolyNode* Polynomial;
struct PolyNode{
    int coef;   // 系数
    int expon;  // 指数
    Polynomial link;    // 下一个节点的地址
}
```

例如

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



启示

- 同一个问题可以有不同的表示（存储）方法（数组，链表）
- 有一类共性的问题：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**
- 表起始位置称为**表头**，结束位置称为**表尾**

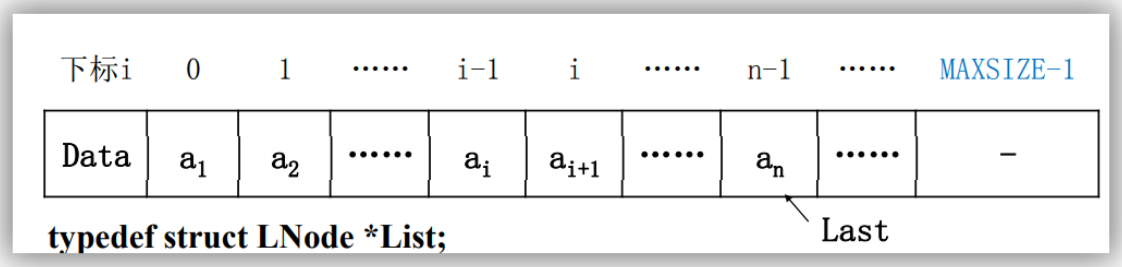
线性表抽象数据类型描述

- 类型名称：线性表 (List)
- 数据对象集：线性表是由$n \geq 0$个元素构成的有序序列
- 操作集：线性表L ∈ List，整数i表示位置，元素X ∈ ElementType，基本操作有

```
List MakeEmpty();    //初始化一个空线性表L
ElementType FindKth(int K, List L); //根据索引K，返回相应元素
int Find(ElementType X, List L);     //在L中查找X的第一次出现的位置
void Insert(ElementType X, int i, List L);  //在L的索引i前插入一个新元素X
void Delete(int i, List L); //删除索引为i的元素
int Length(List L); //返回线性表L的长度n
```

实现

• 顺序存储实现




```c
typedef struct LNode* List;
struct LNode{
    ElementType Data[MAXSIZE];  //存放线性表的数组
    int Last;   // 线性表的最后一个元素的索引
};
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`

• 主要操作实现

- 1. 初始化（建立空的顺序表）

```c
List MakeEmpty()
{
    List PtrL;
    PtrL = (List)malloc(sizeof(struct LNode));
    PtrL->Last = -1;    //顺序表中存在一个元素是这里才是0
    return PtrL;
}
```

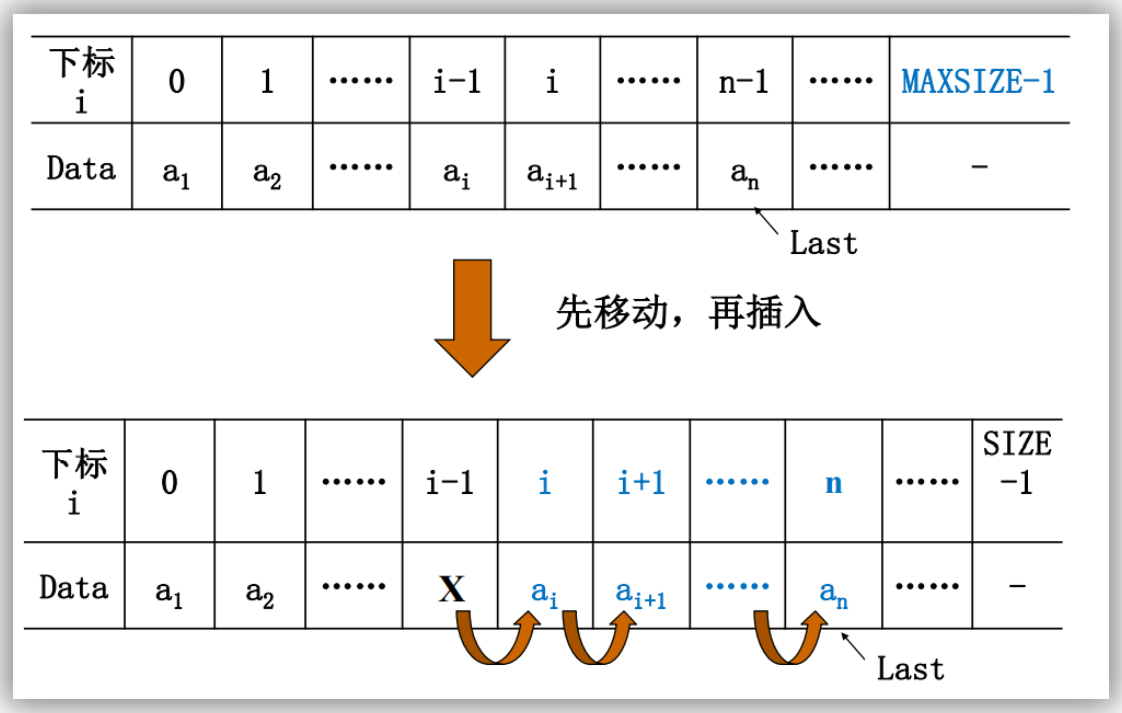
- 2. 查找（查找值为 X 的元素的位置）

```c
int Find(ElementType X, List PtrL)
{
    int i = 0;
    while(i <= PtrL->Last && PtrL->Data[i] != X)
        i++;
    if(PtrL->Last <= i) // 没找到，返回-1
        return -1;
    else
        return i;    // 找到了，返回对应位置（索引）
}
```

查找成功的平均比较次数是 `(n + 1) / 2`，平均时间性能为 `O(n)`

- 3. 插入（在第 i（1 ≤ i ≤ n + 1）个位置（即 Data[n - 1]）插入一个值为 X 的新元素）

先把要插入位置及其之后的元素统一向后移动一位之后，再在该位置插入元素



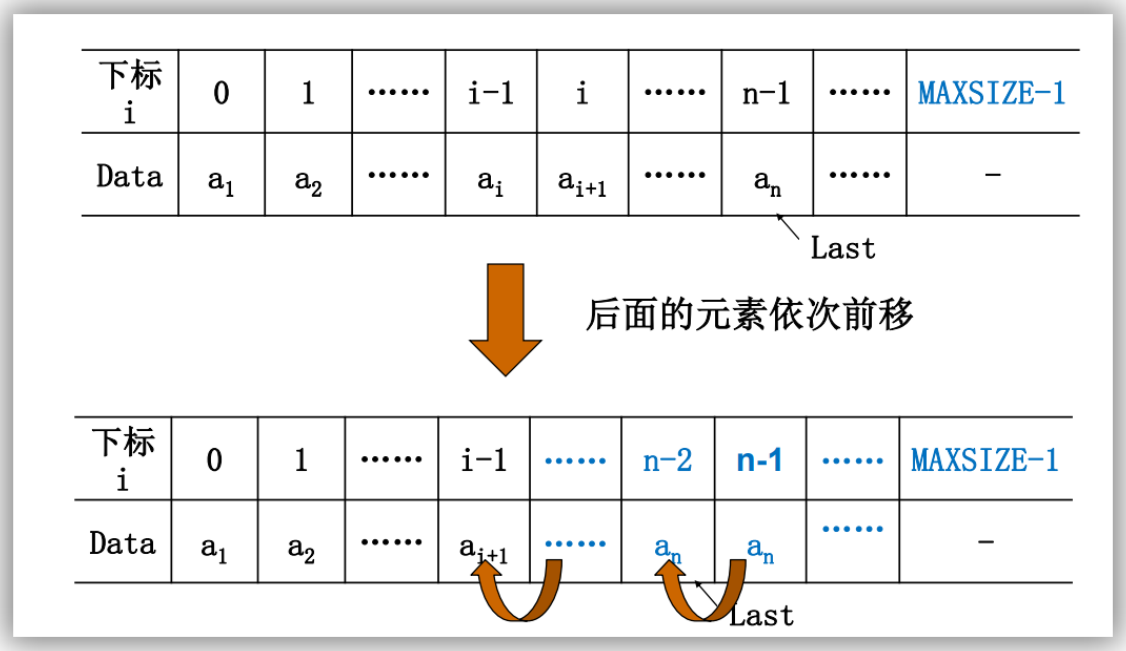

```c
void Insert(ElementType X, int i, List PtrL)
{
    if(PtrL->Last == MAXSIZE - 1)    // 表空间已满，不能插入
    {
```

```
5           printf("表满，无法插入");  return;
6       }
7       if(i < 1 || PtrL->Last + 2 < i)
8       {
9           printf("位置不合法，无法插入");    return;
10      }
11      // 将 a[i]~a[n]倒序向后移动，倒序是为了避免覆盖
12      for(int j = PtrL->Last; i - 1 <= j; j--)
13          PtrL->Data[j + 1] = PtrL->Data[j];
14      PtrL->Data[i - 1] = X;  // 新元素插入
15      PtrL->Last++;    // 保证Last依然指向最后元素
16  }
```

平均移动次数为 `n / 2`，平均时间性能为 `O(n)`

- 4. 删除（删除第 `i`（`1 ≤ i ≤ n`）个位置上的元素）

将要删除位置后面的元素统一向前移动一位覆盖要删除的位置即可，这里需要顺序移动

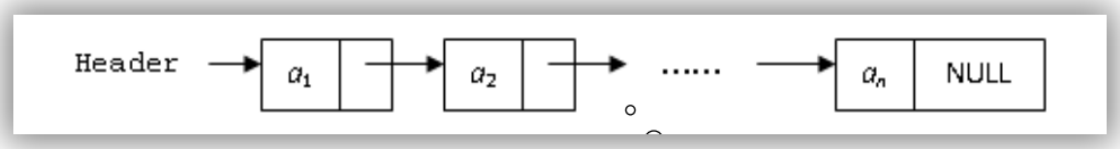


```
void Delete(int i, List PtrL)
{
    if(i < 1 || PtrL->Last + 1 < i)
    {
        printf("不存在第%d个元素", i); return;
    }
    for(int j = i; j <= PtrL->Last; j++)
        PtrL->Data[j - 1] = PtrL->Data[j];
    PtrL->Last--;    // 保证Last依然指向最后元素
}
```

平均移动次数为 `(n - 1) / 2`，平均时间性能为 `O(n)`

• 链式存储实现

顺序存储结构中的元素逻辑相邻的同时存放地址也是相邻的；而链式存储则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，这种存储方式的主要优势就体现在插入和删除不需要移动数据元素，而只是修改 **链**




```
typedef struct LNode* List;
struct LNode{
    ElementType Data;    // 节点数据
    List Next;  // 下一个节点的位置
};
struct LNode L;
List PtrL;
```

• 主要操作实现

1. 求表长

```
int Length(List PtrL)
{
    List p = PtrL;  // 指向第一个节点
    int j = 0;
    while(p)
    {
        p = p->Next;
        j++;
    }
    return j;
}
```

时间性能为 `O(n)`

2. 查找

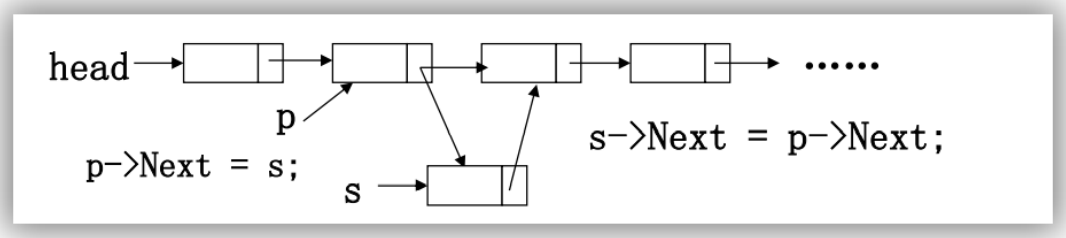
```
// 按序号查找，找第K个元素
List FindKth(int K, List PtrL)
{
    List p = PtrL;
    int i = 1;
    while(p != NULL && i < K)
    {
        p = p->Next;
        i++;
    }
    if(i == K)  return p;    // 找到第K个元素，返回指针
    else return NULL;    // 找不到，返回NULL

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

两种查找平均时间性能均为 `O(n)`

- 3. 插入（在第 `i` 个节点后插入一个值为X的新节点，返回链表 `head` ）

1. 构造一个新节点，用 `s` 指向
2. 找到链表的第 `i- 1` 个节点，用 `p` 指向
3. 修改指针，先将 `p->next` 赋给 `s->next` ，再将 `s` 赋给 `p->next` ，注意这个过程不能颠倒

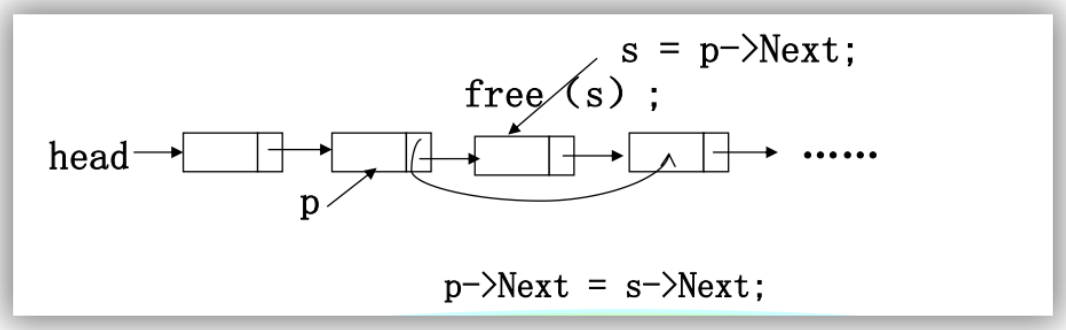



```
List Insert(ElementType X, int i, List PtrL)
{
    List p, s = (List)malloc(sizeof(struct LNode)); // 申请，填装节点；
    // 新节点插在表头，新节点前面没有第i-1个节点，因此不需要p->Next = s
    if(i == 1)
    {
        s->Data = X;
        s->Next = PtrL;
        return s;
    }
    p = FindKth(i - 1, PtrL);    //查找第i-1个节点
    if(p == NULL)    // 第i-1个节点不存在，则不能插入
    {
        printf("参数i有误"); return NULL;
    }
    else
    {
        s->Data = X;
        s->Next = p->Next;
        p->Next = s;
        return PtrL;
    }
}
```

除了查找之外的操作都是常数时间完成，因此插入的平均时间性能为 `O(n)`

- 4. 删除（删除链表第 `i` 个位置上的节点，返回链表 `head` ）

1. 找到第 `i - 1` 个节点，用 `p` 指向
2. `s` 指向要被删除的节点，即 `p->next`
3. 修改 `p->next` 为 `s->next` （或者是 `p->next->next` ）
4. 释放 `s` 所指节点的空间




```
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);    // 寻找第i-1个节点
    // 第i个节点或第i-1个节点不存在都无法删除
    if(p == NULL || p->Next == NULL)
    {
        printf("第%d个节点不存在", i); return NULL;
    }
    else
    {
        s = p->next;
        p->next = s->next;
        free(s);
```

```
            return PtrL;
        }
    }
```

除了查找之外的操作都是常数时间完成，因此删除的平均时间性能为 `O(n)`

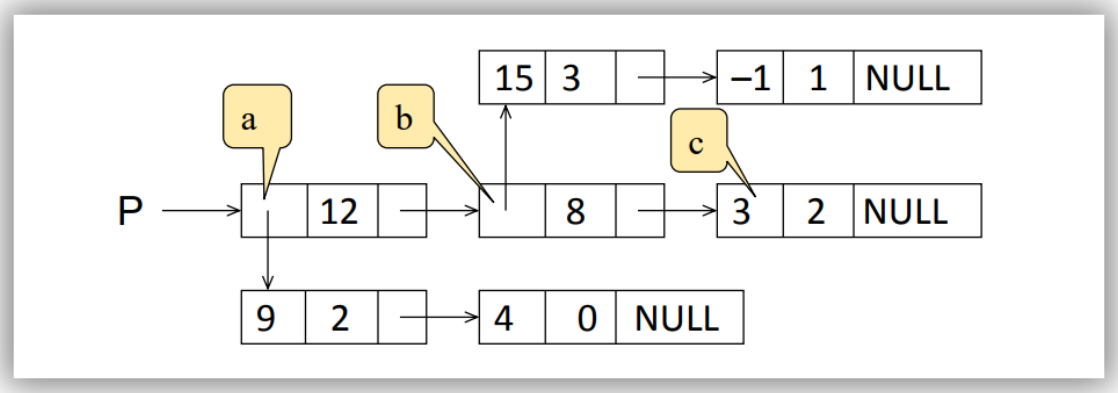
广义表 Generalized List：表中元素可以是单元素或广义表

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?

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

一种方法是将其看成是关于x的一元多项式，而这个一元多项式的系数中又包含了关于y的一元多项式，$P(x,y)=\left(9y^2+4\right)x^{12}+\left(15y^3-y\right)x^8+3x^2$

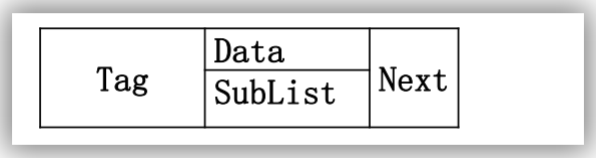
我们用一种复杂的链表来表示这种二元多项式，这就是一种广义表



- 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
- 广义表的元素可以是单元素，也可以是另一个广义表（包括线性表，及其无限套娃）

例如：(1, 2, 3, (4, 5, (6, 0, (99, 94))), (88, 99, 32))是一个长度为5的广义表，它的后两个元素是它的子表

```
typedef struct GNode* GList;
struct GNode
{
    int Tag;     // 标志域，0表示节点是单元素，1表示节点是广义表
    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; // 后继节点
}
```



多重链表：链表中节点可能同时隶属于多个链

- 多重链表中节点的**指针域会有多个**，实际上广义表中的例子就是一个多重链表，包含了 `Next` 和 `SubList` 两个指针域
- **含有多个指针域的链表不一定是多重链表**，比如双向链表，指针域中的指针必须指向不同的链才行
- 多重链表应用广泛，例如树，图这种复杂的数据结构都可以使用多重链表来存储

例：矩阵存储

用二维数组存储矩阵有两个缺点

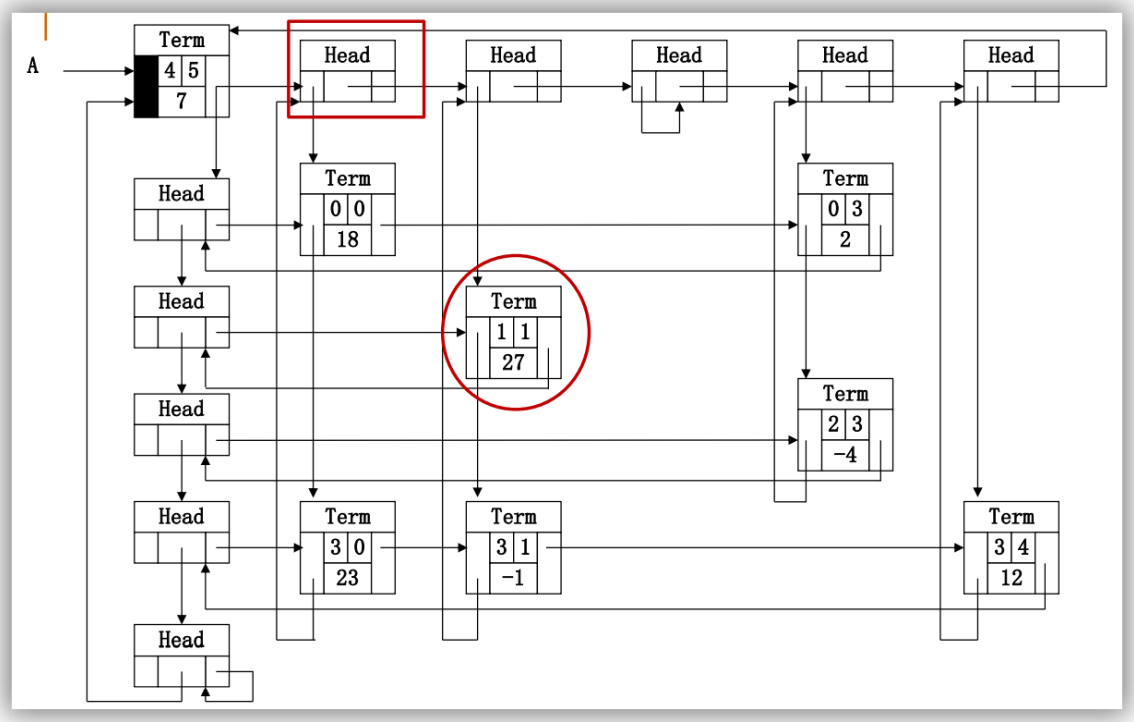
- 事先需要确认数组的大小
- 如果是稀疏矩阵，则会造成大量的存储空间浪费

我们采用一种典型的多重链表---十字链表来存储稀疏矩阵

- 只存储矩阵的非0元素项
- 每个非零元素项节点的数据域包括：行坐标Row，列坐标Col，数值Value

- 每个节点通过两个指针域：行（向右）指针 Right，列（向下）指针 Down，把同行、同列串起来

下图是矩阵 $\begin{bmatrix} 18 & 0 & 0 & 2 & 0 \\ 0 & 27 & 0 & 0 & 0 \\ 0 & 0 & 0 & -4 & 0 \\ 23 & -1 & 0 & 0 & 12 \end{bmatrix}$ 的十字链表表示



- 一共存在两种节点，一种是头节点标识节点，标识值为 `Head`，一种是非零元素节点，标识值为 `Term`
- 注意左上角为矩阵的入口节点，它的Row是矩阵行数，Col是矩阵列数，Value是矩阵非零元素个数，它通过两个指针域连接了行的头节点和列的头节点
- 每一行和每一列对于的链表都是循环链表

节点总体结构

Tag		
Down	*URegion*	Right

矩阵非零元素节点

Term			
Down	Row	Col	Right
	Value		

头节点

Head		
Down	Next	Right

Part 2 堆栈

酷酷酷