第 2 章

线性结构

提纲

2.1 问题引入：一元多项式

例2.1 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

- 问题：如何在计算机中表示一元多项式并实现相关的运算？
- 多项式的关键数据：多项式项数、每一项系数（以及对应指数）

方法1：采用顺序存储结构直接表示一元多项式

用数组a存储多项式的相关数据：数组分量a[i]表示项x^i的系数

例如：$4x^5-3x^2+1$

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

- 这种表示的优点和缺点？

优点：多项式相加容易

缺点：如何表示 $1+2x^{30000}$

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。

$$\{(a_n,\ n),\ (a_{n-1},\ n-1),\ \ldots\ldots,\ (a_0,\ 0)\}$$

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

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

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

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

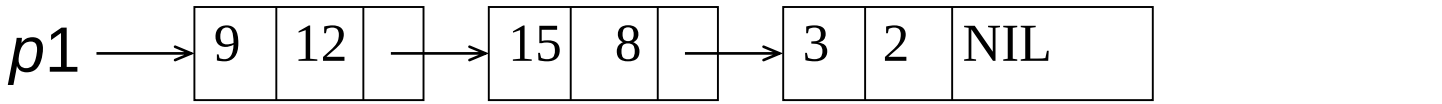
- 这种表示的优点和缺点？

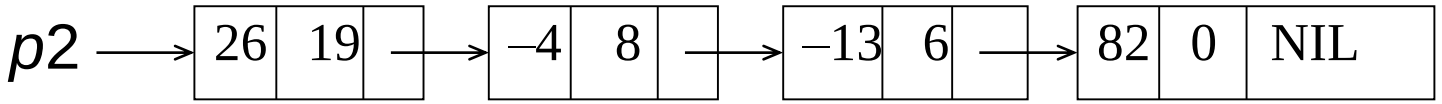
方法3：采用链表结构来存储多项式的非零项

链表表示多项式：链表结点对应一个非零项，包括：系数、指数、指针域

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

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





- 这种表示的优点和缺点？

2.1 问题引入：一元多项式

启示：数据结构的设计

- 往往需要在算法简洁、可理解性与时间、空间效率之间权衡
- 针对具体问题选择合适的数据结构及设计相应的算法

本章介绍：线性表的抽象定义，并分别讨论基于顺序存储和链接存储的线性表实现方法，以及线性表的基本操作，包括插入、删除等。

2.2.1 线性表的定义

线性表：由同一类型的数据元素构成的有序序列的线性结构

代码2-1：线性表的抽象数据类型定义

ADT List {

数据对象：

{ a_i| a_i∈ElemSet，i=1,2,…n, n > 0 } 或Φ，即空表；ElemSet为元素集合。

数据关系：

{<a_i, a_{i+1}>|a_i, a_{i+1}∈ElemSet，i=1,2,…n-1 }，a_1为表头元素，a_n为表尾元素。

基本操作：

InitList(*list*)：初始化一个空的线性表*list*。

DestroyList(*list*)：释放线性表*list*占用的所有空间。

Clear (*list*)：清空线性表*list*。

IsEmpty (*list*)：当线性表*list*为空时返回真值，否则返回假值。

Length (*list*)：返回线性表*list*中的元素个数，即表的长度。

Get (*list, i*)：返回线性表*list*中第*i*个元素的值。

Search(*list, x*)：在线性表*list*中查找元素*x*，查找成功，返回*x*的位置，否则返回NIL。

Insert (*list, i, x*)：在线性表*list*的第*i*个位置上插入元素*x*。

Remove (*list, i*)：从线性表*list*中删除第*i*个元素。

}

2.2.2 线性表的结构

线性表的逻辑结构：数据元素之间线性的序列关系，即数据元素之间的前驱和后继关系。
线性表的物理结构：线性表在计算机中的存储方式，又称为存储结构，即从程序实现的角度将逻辑结构映射到计算机存储单元中。

存储结构主要有两种形式：顺序存储结构和链式存储结构。

- **顺序存储结构**：数据元素被顺序地存储在连续的内存空间中，前驱和后继元素在物理空间上是相邻的。
- **链式存储结构**：可以动态地申请存储数据的结点空间，并使用类似指针这样的手段将结点按顺序前后链接起来

2.3.1 线性表的顺序存储实现

例如：数组list.data[kMaxSize]: 从list.data[0]开始依次顺序存放，kMaxSize代表数组最大容量，list.last记录当前线性表中最后一个元素在数组中的位置，表空时list.last= -1。

last

data	a_1	a_2	……	a_i	a_{i+1}	……	a_n	……	-
i	0	1	……	*i*-1	*i*	……	*n*-1	……	kMaxSize-1

2.3.2 顺序表的基本操作

1.初始化：顺序表的初始化即构造一个空表

(1)动态分配表结构所需要的存储空间

(2)将表中 list.last 指针置为-1，表示表中没有数据元素。

代码 2-2 产生一个初始空顺序表 InitList(*list*)

输入： 顺序表 *list*

输出： 完成了初始化的空顺序表 *list*

```
1   list.data ← new ElemSet[kMaxSize]   //申请顺序表空间
2   list.last ← -1;                     //空表中 list.last 值为-1
```

2.3.2 顺序表的基本操作

2.查找

查找：在线性表中查找与给定值x相等的数据元素

方法：从第一个元素 起依次和x比较，直到找到一个与x相等的数据元素，返回它在顺序表中的存储下标；或者查遍整个表都没有找到与x相等的元素，则返回NIL。

算法 2-1 在顺序表 *list* 中查找元素 *x*　Search(*list*, *x*)

输入： 顺序表 *list*，$x \in$ ElemSet

输出： 元素 *x* 在顺序表 *list* 中的位置 *i* (由于顺序表中的位置从 0 开始，*x* 的实际位序是 *i*+1)；
如果 *x* 不在顺序表中返回 NIL

```
1  i ← 0;
2  while i ≤ list.last 并且 list.data[i] ≠ x do
3  |  i←i+1
4  end
5  if  i > list.last then    // 如果没找到，返回 NIL
6      i ← NIL
7  end
8  return i
```

时间复杂性：

- 1~（n+1)/2
- 平均O(n)

2.3.2 顺序表的基本操作

3.插入

在表的第i个位序上插入一个值为x的新元素

$(a_1,a_2,\ldots,a_{i-1},a_i,a_{i+1},\ldots,a_n)$ -->

$(a_1,a_2,\ldots,a_{i-1},x,a_i,a_{i+1},\ldots,a_n)$

执行步骤：

（1）将$a_i \sim a_n$顺序向后移动（移动次序从后到前），为新元素让出位置；

（2）将x置入空出的第i个位序；

（3）修改 list.last 指针(相当于修改表长)，使之仍指向最后一个元素。

时间复杂度为O(n）

算法 2-2 在顺序表 *list* 的第 *i* 个位置上插入元素 *x*　Insert (*list*, *i*, *x*)

输入： 顺序表 *list*，*i* 是插入位置的序号（从 1 开始），*x*∈ ElemSet

输出： 完成插入后的顺序表 *list*

```
1   if  list.last = kMaxSize－1 then    // 表空间已满，不能插入
2   |   表满不能插入，退出
3   end
4   if  i < 1 或者 i > list.last + 2 then   //检查 i 的合法性。注意 i 代表位序，不是数组下标
5   |   插入位置不合法，退出
6   end
7   for  j ← list.last downto i-1 do
8   |   list.data[j+1] ← list.data[j]   //将 a_i~a_n 顺序向后移动
9   end
10  list.data[i-1] ← x   // 新元素插入
11  list.last ← list.last + 1 // list.last 仍指向最后元素
```

单选题 1分

设置

长度为n的顺序表中插入一个元素，需要移动表中已有元素。问在最好和最坏情况下，已有元素的移动总次数分别是：

A 1 和 n

B 0 和 n

C 0 和 1

D n 和 n

提交

2.3.2 顺序表的基本操作

4.删除

将线性表中的第i个位序上的元素删除

$(a_1,a_2,\ldots,a_{i-1},a_i,a_{i+1},\ldots,a_n)$ -->

$(a_1,a_2,\ldots,a_{i-1},a_{i+1},\ldots,a_n)$

■ 执行步骤：

(1)将～顺序向前移动，元素被覆盖；

(2)修改list.last指针(相当于修改表长)使之仍指向最后一个元素。

时间复杂度为O(n）

算法 2-3 从顺序表 *list* 中删除第 *i* 个元素　Remove (*list*, *i*)

输入： 顺序表 *list*，*i* 是删除元素的位置序号（从 1 开始）

输出： 完成删除后的顺序表 *list*

```
1  if  i<1 或 i > list.last + 1 then  // 检查空表及删除位置的合法性
2  |   不存在这个元素，退出
3  end
4  for  j←i to list.last do
5  |   list.data[j-1]← list.data[j]  // 将 a_{i+1} ~ a_n 顺序向前移动
6  end
7  list.last ←list.last – 1  // list.last 仍指向最后元素
```

单选题 1分

设置

如果顺序表不需要保持元素的次序，只需保持元素连续排放，
插入元素到某个位置或删除某个位置的元素需要多少时间?

A O(1)

B O(log(n))

C $O(\sqrt{n})$

D O(n)

提交

顺序表的反转

反转：

$(a_1,a_2,\ldots,a_{i-1},a_i,a_{i+1},\ldots,a_{n-1},a_n)$

$(a_n,a_{n-1},\ldots,a_{i+1},a_i,a_{i-1},\ldots,a_2,a_1)$

■ 算法1：删除+插入

从顺序表的末尾开始处理，每次删除表中最后一个元素，并把第k次删除的元素插入到顺序表的第k个位置$(1 \le k < n)$；重复该操作n-1次。

```
算法：Reverse(list)

输入：顺序表list
输出：反转顺序表中所有元素的位置

1. n ← list.last   //元素个数
2. for i ← 1 to n-1 do   //循环n-1次
3. |   dat ← list.data[n]   //取出最后一个元素
4. |   Remove(list, n)  //删除最后一个元素
5. |   Insert(list, i, dat)  //插入到第i个位置
6. end
```

时间复杂度为$O(n^2)$

顺序表的反转

反转：

$(a_1,a_2,\ldots,a_{i-1},a_i,a_{i+1},\ldots,a_{n-1},a_n)$

$(a_n,a_{n-1},\ldots,a_{i+1},a_i,a_{i-1},\ldots,a_2,a_1)$

■ 算法2：顺序交换

执行n/2次交换，第i次交换顺序表第i个元素与第n-i+1个元素

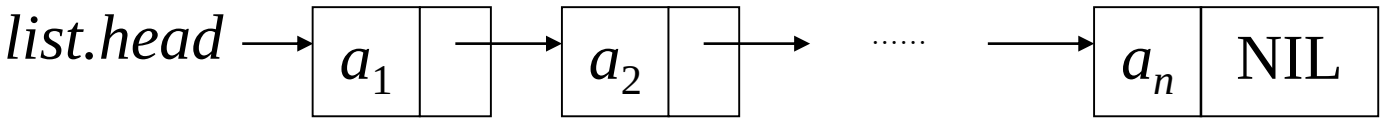
$$(1 \le i \le \frac{n}{2})$$

```
算法：Reverse(list)

输入：顺序表list
输出：反转顺序表中所有元素的位置

1. n ← list.last   //元素个数
2. for i ← 1 to n/2 do   //循环n/2次
3. |   j ← n – i + 1  //交换第i和第n-i+1个元素
4. |   dat ← list.data[i]
5. |   list.data[i] ← list.data[j]
6. |   list.data[j] ← dat
7. end
```

时间复杂度为O(n)

2.4.1 线性表的链接存储实现

单向链表（简称单链表）：每个数据单元由数据域data和链接域next两部分组成，n个数据单元通过链接域next串联起来。



链式存储：

- 不要求逻辑上相邻的两个数据元素物理上也相邻
- 通过“链”建立起数据元素之间的逻辑关系
- 对线性表的插入、删除不需要移动数据元素，只需要修改“链”。

2.4.2 单链表的基本操作

1. **求表长**：求单链表元素个数

（1）设一个移动指针p和计数器counter，初始化

（2）p逐步往后移，同时计数器counter加1

（3）当后面不再有结点时，counter的值就是结点个数，即表长。

算法 2-4 求单链表 *list* 中的元素个数，即表长　Length (*list*)

输入： 单链表 *list*，其中 *list.head* 指向链表头结点

输出： 单链表长度

```
1   p ← list.head
2   counter ← 0
3   while  p ≠ NIL  do
4   |   counter ← counter + 1
5   |   p ← p.next
6   end
7   return counter
```

时间复杂性O(n)

2.4.2 单链表的基本操作

2. **查找**：分按序号查找Get (list, i)、按值查找Search(list, x)

（1）按序号查找步骤：

- 从链表的第一个元素结点起，判断当前结点是否是第i个；
- 若是，则返回该结点的值，否则继续后一个，直到表结束为止。
- 如果没有第i个结点则返回错误码(ErrorCode)。

时间复杂性：O(n)

算法 2-5 返回单链表 *list* 中第 *i* 个元素值　Get (*list*, *i*)

输入： 单链表 *list*，其中 *list.head* 指向链表头结点；*i* 是待查找元素在链表中的位序（从 1 开始）

输出： 第 *i* 个元素值；如果不存在则返回错误码（ErrorCode）

```
1   if list.head = NIL 或 i = 0 then //空表或查找位置不合法
2   |  return ErrorCode
3   end
4   p ← list.head
5   counter ← 1
6   while  p ≠ NIL 且 counter < i  do
7   |  p ← p.next
8   |  counter ← counter +1
9   end
10  if  p ≠ NIL  then
11  |  return p.data
12  else
13  |  return ErrorCode //不存在第 i 个元素
14  end
```

2.4.2 单链表的基本操作

2.查找

（2）按值查找步骤：

- 从链表的第一个元素结点起，判断当前结点其值是否等于x；
- 若是，返回该结点的位置（即指向该结点的指针），否则继续后一个，直到表结束为止。
- 找不到时返回空（NIL）。

算法 2-6 在单链表 *list* 中查找元素 *x* 所在结点　Search(*list*, *x*)

输入： 单链表 *list*，其中 *list.head* 指向链表头结点；*x*∈ ElemSet

输出： 元素 *x* 在单链表 *list* 中的位置，即指向该结点的指针；如果 *x* 不在单链表中返回 NIL

```
1   p ← list.head
2   while  p ≠ NIL 且 p.data ≠ x  do
3       p ← p.next
4   end
5   return p
```

时间复杂性：O(n)

2.4.2 单链表的基本操作

3. **插入**：在list的第i个位置上插入元素x

步骤：

（1）找到第i-1个结点；

（2）若存在，则申请一个新结点的空间并填上相应值x，然后将新结点插到第i-1个结点之后；

（3）如果不存在则直接退出：

注意：表空和不空时候的不同处理方式

时间复杂度为O(n）

算法 2-7 在单链表 *list* 的第 *i* 个位置上插入元素 *x*　Insert (*list*, *i*, *x*)

输入： 单链表 *list*，*i* 是插入位置的序号（从 1 开始），*x* ∈ ElemSet

输出： 完成插入后的单链表 *list*

```
1  if i < 1 then
2  |  插入位置不合法，退出
3  end
4  if i =1 then  //插入第 1 个结点
5  |  new_node ← new ListNode //创建新的结点
6  |  new_node.data ← x
7  |  new_node.next ← list.head //插入表头
8  |  list.head ← new_node
9  else  // i > 1，寻找第 i-1 个结点并插入其后
10 |  p ← list.head
11 |  counter ← 1
12 |  while p ≠ NIL 且 counter < (i-1) do
13 |  |  p ← p.next
14 |  |  counter ← counter +1
15 |  end
16 |  if p ≠ NIL then //p 指向第 (i-1)个结点
17 |  |  new_node ← new ListNode //创建新的结点
18 |  |  new_node.data ← x
19 |  |  new_node.next ← p.next
20 |  |  p.next ← new_node
21 |  else
22 |  |  插入位置不合法，退出
23 |  end
24 end
```

2.4.2 单链表的基本操作

4. 删除

在单链表中删除指定位序i的元素

步骤：

- 找到被删除结点的前一个元素
- 再删除结点并释放空间。

时间复杂度为O(n）

算法 2-8 从单链表 *list* 中删除第 *i* 个元素 Remove (*list*, *i*)

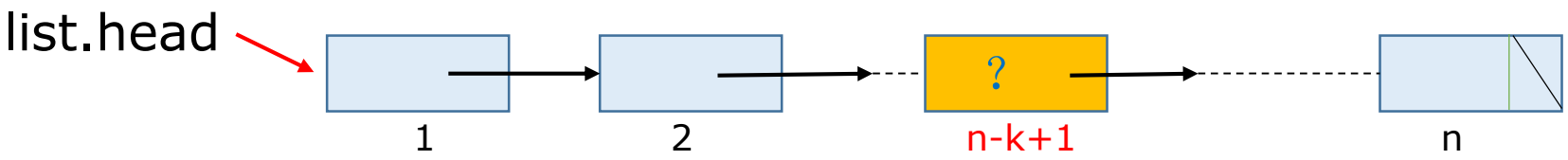
输入： 单链表 *list*，*i* 是删除元素的位置序号（从 1 开始）

输出： 完成删除后的单链表 *list*

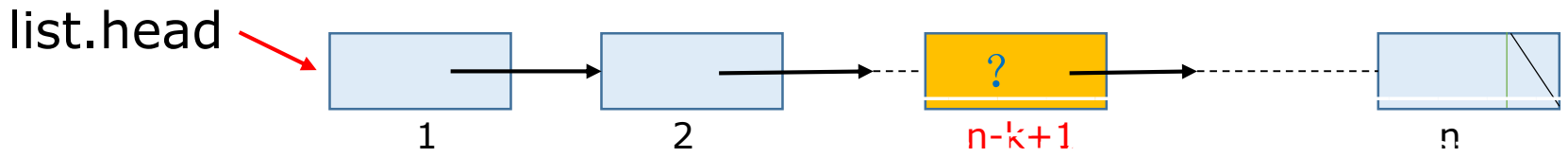
```
1  if i < 1 then
2  |  删除位置不合法，退出
3  end
4  p ← list.head
5  if p ≠ NIL 且 i =1 then //删除第 1 个结点
6  |  list.head ← p.next
7  |  delete p
8  else // i > 1，寻找第 i-1 个结点
9  |  counter ← 1
10 |  while p ≠ NIL 且 counter < (i-1) do
11 |  |  p ← p.next
12 |  |  counter ← counter +1
13 |  end
14 |  if p ≠ NIL 且 p.next ≠ NIL then //p 指向第 (i-1)个结点，且待删除结点存在
15 |  |  deleted_node ← p.next
16 |  |  p.next ← deleted_node.next
17 |  |  delete deleted_node
18 |  else
19 |  |  删除位置不合法，退出
20 |  end
21 end
```

经典面试题

查找单链表list倒数第k个结点，时间复杂度O(n)，n是单链表长度（预先未知）



经典面试题



1. 单指针法（2次遍历）

（1）第一次遍历，记录单链表长度n

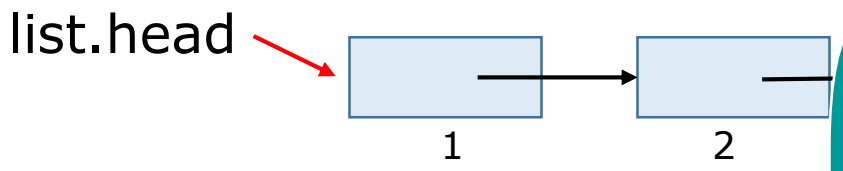
（2）第二次遍历，查找第n-k+1个结点

时间复杂度为O(n)

输入：单链表，整数k (k>0)
输出：单链表第k个结点

```
1.  ptr ← head.list
2.  n ← 0                          //长度的初始值
3.  while ptr ≠ NIL do
4.  |  n ← n + 1                   //数量增1
5.  |  ptr ← ptr.next              //移到下一个结点
6.  end
7.  if n ≥ k then                  //存在倒数第k个结点
8.  |  ptr ← list.head
9.  |  for i ← 1 to n-k do         //移动n-k次
10. |  |  ptr ← ptr.next
11. |  end
12. end
13. return ptr
```

经典面试题



2. 双指针法（1次遍历）

(1) 用第一个指针找到第k个结点

(2) 第二个指针指向头结点

(3) 两个指针同步右移，直到第一个指针走到末尾结点为止

时间复杂度为O(n)

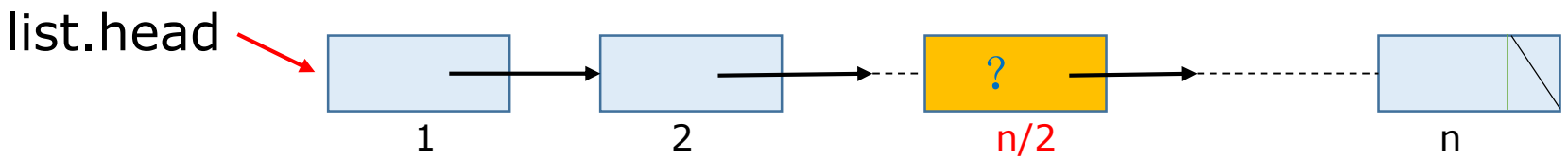
算法：FindKthNodeFromBottom(list, k)

输入：单链表，整数k (k>0)； 输出：单链表第k个结点

```
1.  ptr1 ← head.list          //第一个指针初始化
2.  ptr2 ← NIL                //第二个指针初始化
3.  for j ← 1 to k-1 do       //第一个指针移动k-1次
4.  | if ptr1 ≠ NIL then
5.  | | ptr1 ← ptr1.next
6.  | end
7.  end
8.  if ptr1 ≠ NIL  then  //存在倒数第k个结点
9.  | ptr2 ← head.list   //第二个指针从头开始
10.|  while ptr1.next ≠ NIL  do
11. | | ptr1 ← ptr1.next
12. | | ptr2 ← ptr2.next   //两个指针同时右移
13. | end
14. end
15.return ptr2    //返回第二个指针
```

思考题

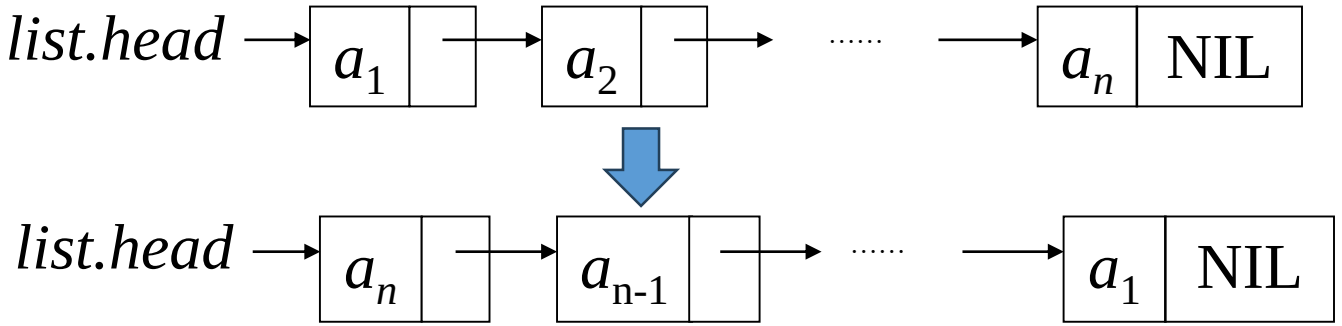
查找单链表list第n/2个结点，时间复杂度O(n)，n是单链表长度（预先未知）



查找双链表list第n/2个结点?

单链表的反转

反转：



■ 算法1：删除+插入

从单链表的末尾开始处理，每次删除表中最后一个结点，并把第k次删除的结点（数据）插入到单链表的第k个位置$(1 \le k < n)$；重复该操作n-1次。

时间复杂度为$O(n^2)$

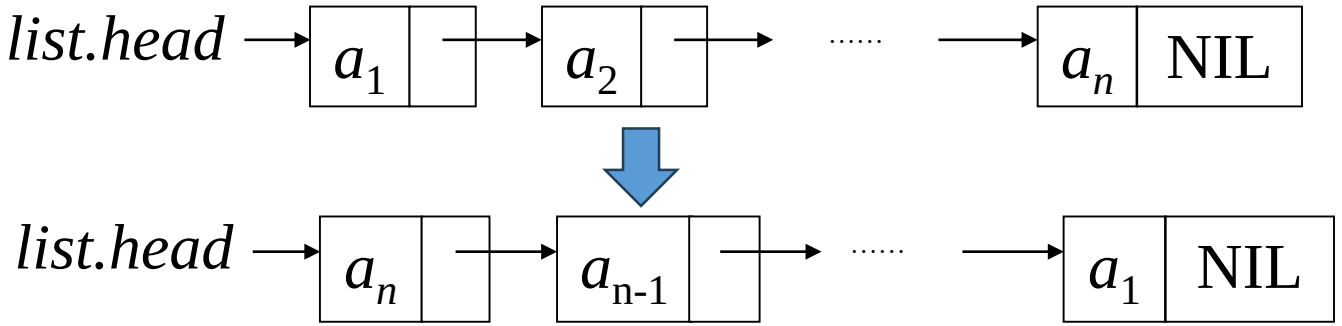
算法：Reverse(list)

输入：单链表list
输出：反转单链表中所有结点的次序

```
1. n ← Length(list)          //元素个数
2. for i ← 1 to n-1 do       //循环n-1次
3. |   dat ← Get(list, n)    //取出最后一个结点数据
4. |   Remove(list, n)       //删除最后一个结点
5. |   Insert(list, i, dat)  //插入到第i个位置
6. | end
7. end
```

单链表的反转

反转：



■ 算法2：指针重排

时间复杂度为O(n)

算法：Reverse(list)

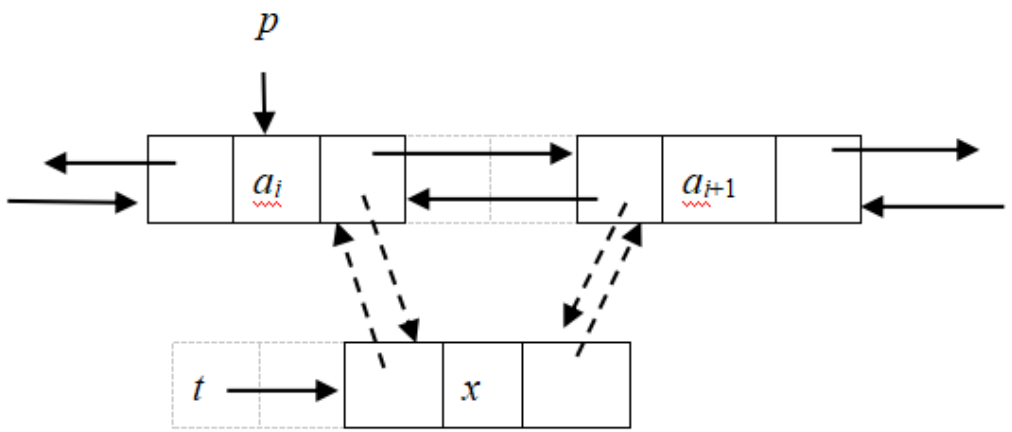
输入：单链表list
输出：反转单链表中所有结点的次序

```
1.  ptr ← list.head    //用ptr指向单链表
2.  list.head ← NIL    //list初始化为空链表
3.  while ptr ≠ NIL do
4.  |   t_head ← ptr    //t_head指向单链表的头结点
5.  |   ptr ← ptr.next  //指针ptr后移
6.  |   t_head.next ← list.head  //t_head所指结点插入到list先头
7.  |   list.head ← t_head
8.  |  end
9.  end
```

2.4.3 双向链表

双向链表：结点前后之间实现双向链接，即每个结点都有两个指针，一个next指向直接后继，另一个prior指向直接前驱。

1. 插入



代码 2-3 在双向链表中 *p* 所指结点后面插入 *t* 所指的结点 DLLInsert(*p*, *t*)

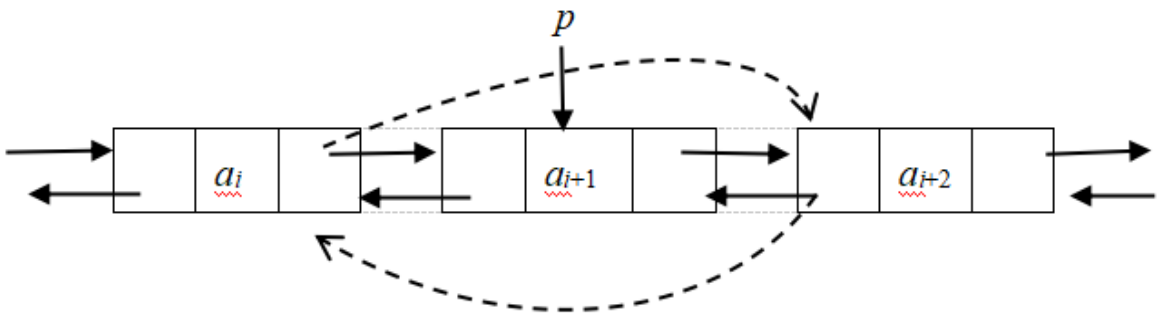
输入： 双向链表中某结点指针 *p*，待插入结点的指针 *t*

输出： 插入完成后的双向链表 *list*

```
1  p.next.prior ← t
2  t.next ← p.next
3  p.next ← t
4  t.prior ← p
```

2.4.3 双向链表的基本操作

2、删除：双向链表中p所指向结点删除



代码 2-4 将双向链表中 *p* 所指的结点删除 DDLDelete(*p*)

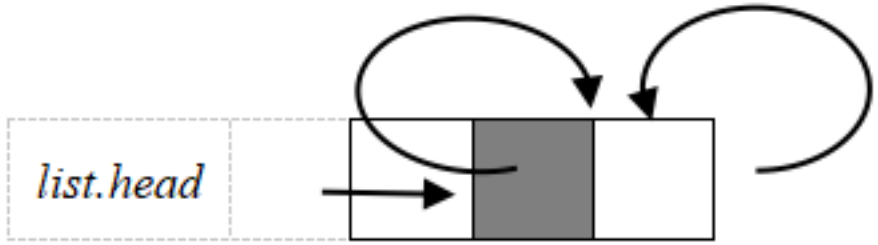
输入： 双向链表中待删除结点指针 *p*

输出： 完成删除后的双向链表 *list*

```
1    p.next.prior ← p.prior
2    p.prior.next ← p.next
3    delete p
```

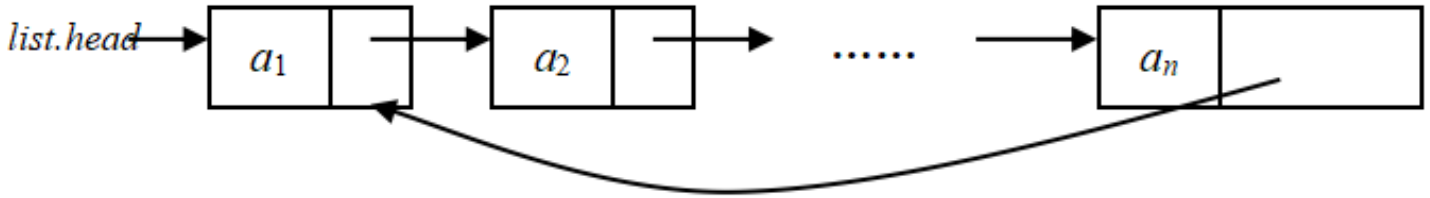
2.4.3 双向链表的基本操作

带头结点的双向链表：可以设置一个空的“头结点”，真正的元素链接在这个空结点之后。

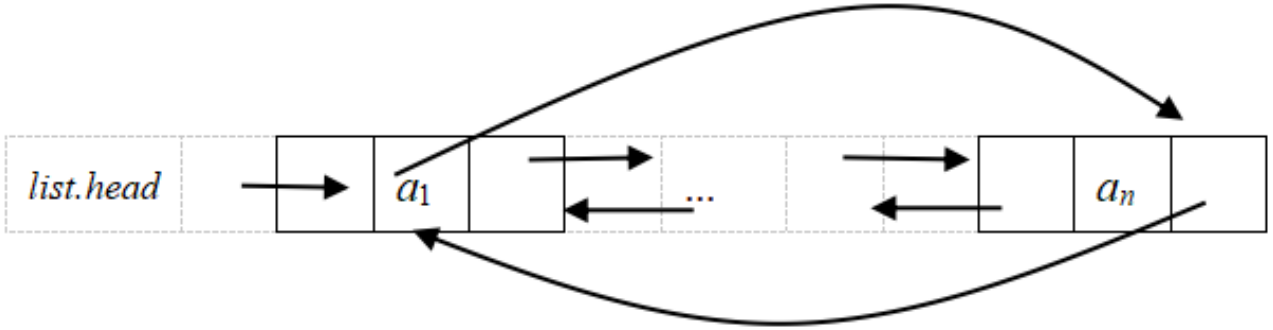


2.4.4 循环链表

循环单链表：链表终端结点的指针指向链表的起始结点



循环双向链表：让终端结点的后继指针指向链表的起始结点，同时让起始结点的前驱指针指向链表的终端结点



2.4.4 循环链表

单向循环链表的遍历：可以从任意结点start开始将整个链表遍历一遍

代码 2-5 从单向循环链表中 *start* 指向的结点开始，遍历每个结点 CLLTraverse(*start*)

输入： 起始结点的指针 *start*

输出： Visit 函数处理每个访问结点的结果

```
1   p ← start //通过指针 p 遍历
2   do
3   |   Visit(p)
4   |   p ← p.next
5   while   p ≠ start
```

2.4.5 静态链表

静态链表：用数组存放线性表中的元素，但并不按照元素顺序在数组中依序存放，而是给每个数组元素增加一个域，用于指示线性表中下一个元素的位置（即它在数组中的下标）

- 物理存储空间上依赖于数组
- 元素逻辑链接关系是采用了链表的思想

data		g	d	a	e	b	f		h	c	
next	3	8	4	5	6	9	1		0	2	
下标	0	1	2	3	4	5	6	7	8	9	10

（a,b,c,d,e,f,g,h）

代码 2-6 将静态链表 *list* 的每个结点访问一遍 SLTraverse(*list*)

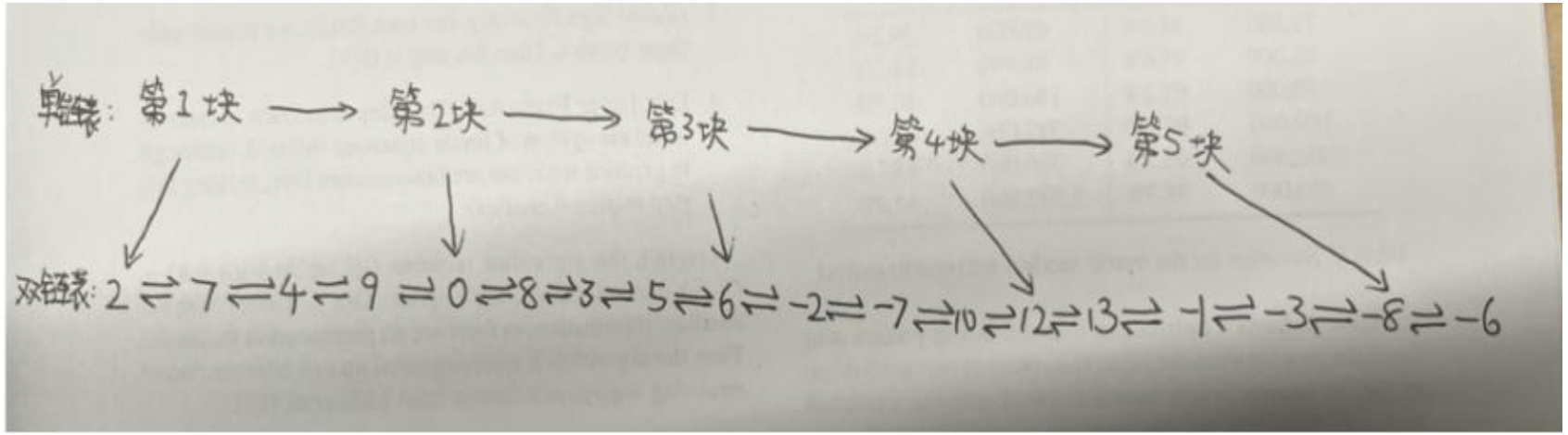
输入： 静态链表 *list*

输出： Visit 函数处理每个访问结点的结果

```
1  p ← list[0].next
2  while p ≠ 0 do
3  |  Visit(p)
4  |  p ← list[p].next
5  end
```

2.4.6 块状链表

块状链表：采用双向链表维护总长度不超过n的线性表的各元素，并将元素分成若干个块，每个块所包含的元素个数为$\sqrt{n}$（最后一个块内的元素个数可能不满）。



时间复杂度：

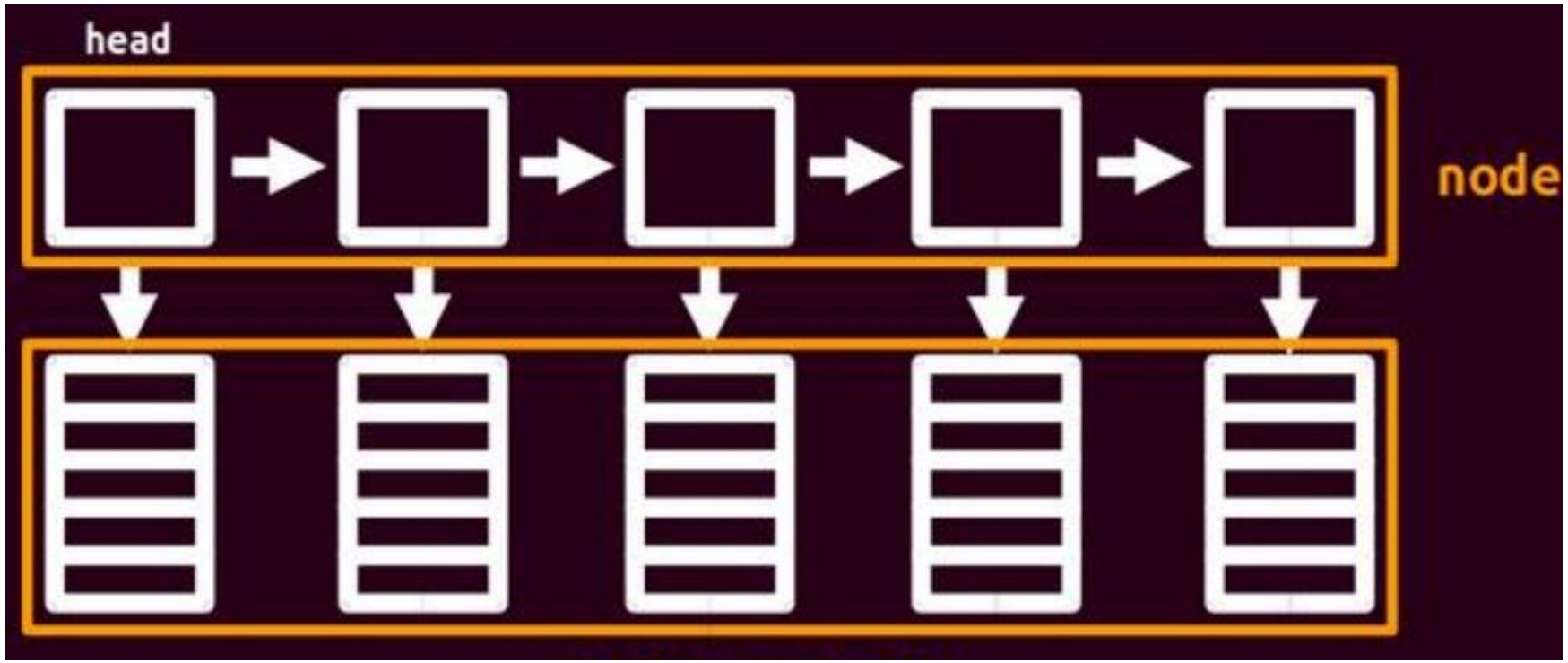
$O(\sqrt{n})$

1. 查找操作

查找第10个元素的查找路径：

（1）在单向链表中遍历第1块、第2块、第3块；

（2）进入第3块对应的双向链表继续查找结点6、结点-2，即第10个元素为-2。


head
node

2.4.6 块状链表

2. 插入操作：在第i个元素之后插入新元素

（1）基于查找操作找到第个元素所对应的双向链表结点

（2）在双向链表上执行插入操作

（3）由于第i个元素所在块多了一个元素，因此该块之后的所有块指向的双向链表结点将变为原指向结点的前驱结点。

例如，如果向第10个元素-2之后插入新元素20

1）在结点-2和-7之间插入新元素20，变为-2、20、-7。

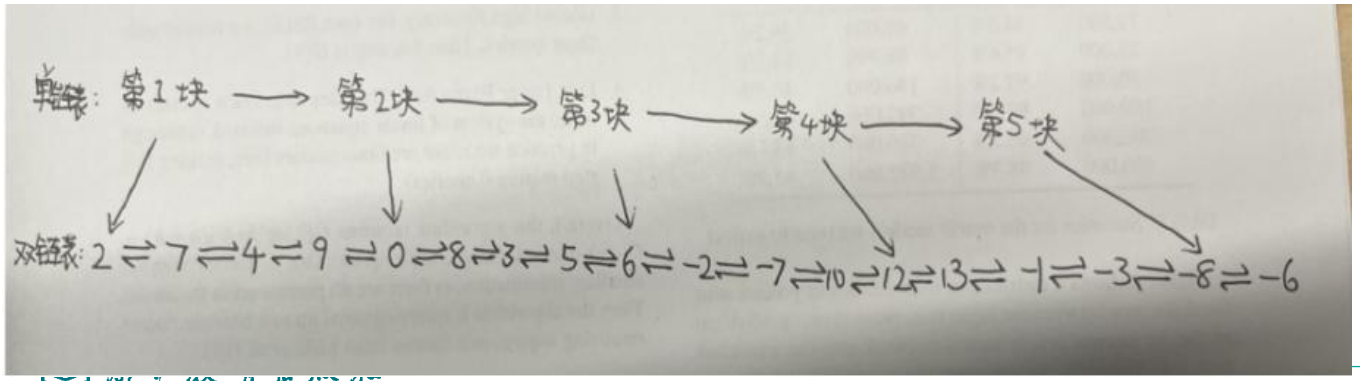
2）原有元素次序产生变化：原第11个元素变为现第12个、原第12个元素变为现第13个……。

3）将第4块所指向的双向链表结点从12更新为前驱10，第5块的双向链表结点从-8更新为前驱-3。

值得额外注意的是，插入操作有可能导致新的块产生，在实现代码中需要处理好这种特殊情况。

3. 删除操作: 与插入类似

插入和删除时间复杂度：$O(\sqrt{n})$



2.5 线性表的应用

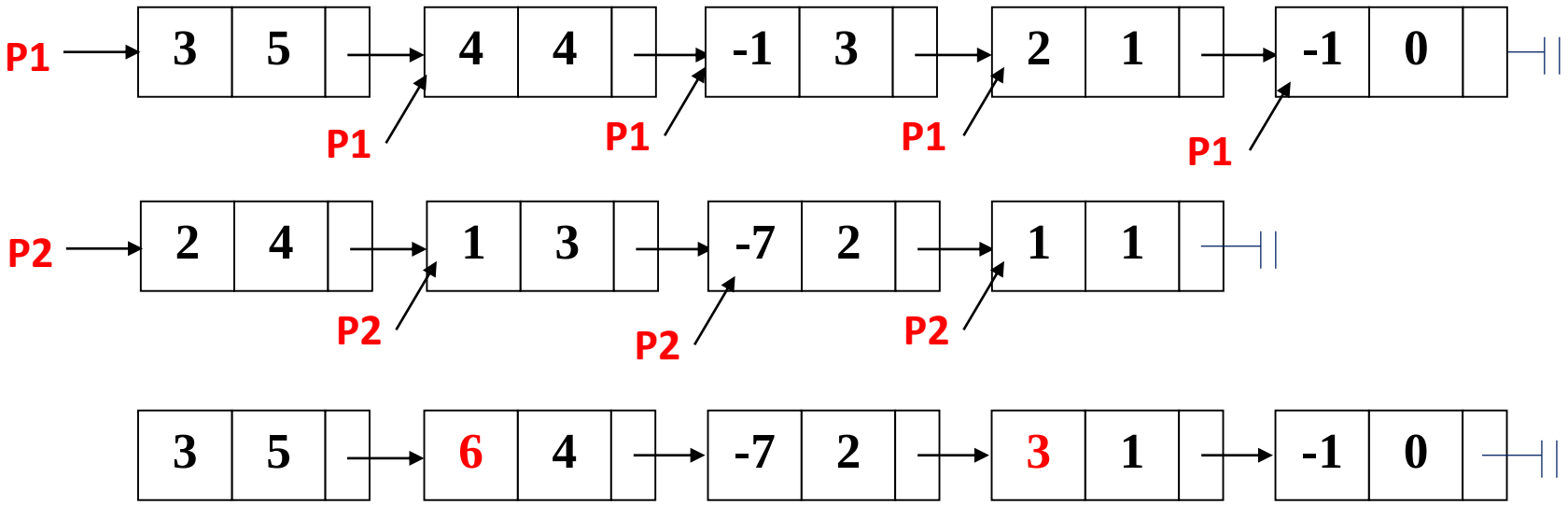
一元多项式的加法:

p1和p2分别指向两个多项式第一个结点，不断循环比较p1和p2所指的各结点，做不同处理

- P1.expon==P2.expon，
  插入求和结点，
  P1=P1.next;
  P2=P2.next

- P1.expon>P2.expon，
  插入P1，P1=P1.next;

- P1.expon<P2.expon，
  插入P2，P2=P.next;



2.5.2 大整数处理

大整数表示：用顺序表中的元素依次表示该大整数的个位数、十位数、百位数……

n = -23456所对应的digits[] = {6, 5, 4, 3, 2}，length = 5，sign = -1

1. 加法运算：假设a+b>0,基本过程是：

(1) 首先将这两个大整数的位数对齐（位数较少的大整数的高位补0）

(2) 将两个大整数对应的位数依次相加或相减，同时处理进位或借位，并将结果存入一个新的大整数*c*中。

(3) 处理加法导致的最高位进位，或减法导致的前导0问题。

时间复杂度是O(*n*)

- 处理一个正的大整数*a*加一个负的大整数-*b*（*a*<*b*）：计算*b*和-*a*相加的结果，并将最终结果的符号位取反；
- 处理一个负的大整数-*a*加一个负的大整数-*b*：计算*a*和*b*相加的结果，并将最终结果的符号位取反。

2.5.2 大整数处理

2. 乘法运算：两个正的大整数a和b相乘

（1）用i和j的二重循环分别枚举大整数a和b的每一位（数组下标从0开始）

（2）对于a的第i位数字和b的第j位数字（按从低位到高位的顺序），将它们相乘并累加至表示计算结果的大整数c的第i+j位上。

（3）对于得到的大整数c，从最低位到最高位依次处理进位问题，得到最终的计算结果。

时间复杂度是$O(n^2)$

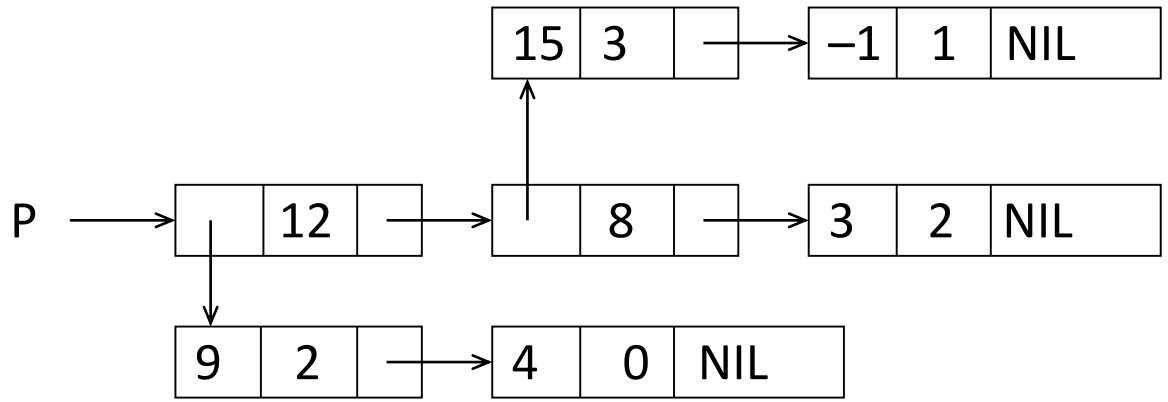
2.6 拓展延伸

1.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；
- 广义表中，这些元素不仅可以是单元素也可以是另一个广义表。

例如：二元多项式的表示

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



2.6.2多维数组和特殊矩阵

1. 多维数组的顺序存储：按照行优先的顺序存放，即先存放第0行的元素，再存放第1行的元素，……，其中每一行中的元素再按照列的顺序存放。

例如：二维整形数组 a[3][4]的存放顺序：

a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[2][0]	a[2][1]	a[2][2]	a[2][3]

一般来说，二维数组元素$a[i][j]$的存储位置（地址）l_{ij}计算方法是：

$$l_{ij} = l_{00} + (i \times n_c + j) \times size$$

设n维数组各维大小是（s_1，s_2，……，s_n），第一个元素的地址是$l_{(0,0,\ldots,0)}$，元素占用空间为$size$个字节，则下标为（i_1,i_2，...,i_n）的元素位置是：

$$l_{(i1,i2,\ldots in)} = l_{(0,0,\ldots,0)} + (i_1 \times s_2 \times s_3 \times .. \times .s_n + i_2 \times s_3 \times \ldots s_n + \ldots i_{n-1} \times s_n + i_n) \times size$$

2.6.2多维数组和特殊矩阵

2. 特殊矩阵：上三角矩阵和下三角矩阵

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix} \qquad \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

压缩空间存储：如果只存储上三角（或者下三角）元素，则将近减少了一半的存储空间

对于下三角矩阵，设单个元素所占空间为$size$，则$a[i][j]$的存储位置l_{ij}与矩阵首个元素$a[0][0]$的地址l_{00}的关系是：

$l_{ij} = l_{00} + (i(i+1)/2+j) \times size$，$i \geq j$

2.6.3稀疏矩阵和舞蹈链

1. 稀疏矩阵：数值为0的元素数目远远多于非0元素的数目，并且非0元素分布没有规律

■ 存储：非0项

1) 三元组表的顺序存储

row	0	0	1	2	3	3	3
col	0	3	1	3	0	1	4
value	18	2	27	-4	23	-1	12
	0	1	2	3	4	5	6

2) 三元组表的链式存储：十字链表

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

稀疏矩阵

	col0	col1	col2	col3	col4	col5
row0	15	0	0	22	0	−15
row1	0	11	3	0	0	0
row2	0	0	0	−6	0	0
row3	0	0	0	0	0	0
row4	91	0	0	0	0	0
row5	0	0	28	0	0	0

nonzero terms

rows columns

transpose

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	−15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	−6
[7]	4	0	91
[8]	5	2	28

(a)

	row	col	value
b[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	−6
[8]	5	0	−15

(b)

稀疏矩阵转置

Assign
A[i][j] to B[j][i]

place element <i, j, value> in
element <j, i, value>

For all columns i

For all elements in column j

Scan the array “columns” times.The array has “elements” elements.

```
void transpose(term a[], term b[])
/* b is set to the transpose of a */
{
   int n,i,j, currentb;
   n = a[0].value;        /* total number of elements */
   b[0].row = a[0].col; /* rows in b = columns in a */
   b[0].col = a[0].row; /* columns in b = rows in a */
   b[0].value = n;
   if (n > 0 )  { /* non zero matrix */
      currentb = 1;
      for (i = 0; i < a[0].col; i++)
      /* transpose by the columns in a */
         for (j = 1; j <= n; j++)
         /* find elements from the current column */
            if (a[j].col == i) {
            /* element is in current column, add it to b */
               b[currentb].row = a[j].col;
               b[currentb].col = a[j].row;
               b[currentb].value = a[j].value;
               currentb++;
            }
   }
}
```

==> O(columns*elements)

A[6][6] 转置成B[6][6]

i=1 j=8
a[i].col = 2 != i

Matrix A

	Row	Col	Value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

Row Col Value

0	6	6	8
1	0	0	15
2	0	4	91
3	1	1	11

```
void transpose(term a[], term b[])
/* b is set to the transpose of a */
{
   int n,i,j, currentb;
   n = a[0].value;        /* total number of elements */
   b[0].row = a[0].col; /* rows in b = columns in a */
   b[0].col = a[0].row; /* columns in b = rows in a */
   b[0].value = n;
   if (n > 0 )  { /* non zero matrix */
      currentb = 1;
      for (i = 0; i < a[0].col; i++)
      /* transpose by the columns in a */
        for (j = 1; j <= n; j++)
        /* find elements from the current column */
          if (a[j].col == i) {
          /* element is in current column, add it to b */
            b[currentb].row = a[j].col;
            b[currentb].col = a[j].row;
            b[currentb].value = a[j].value;
            currentb++;
          }
   }
}
```

Set Up row & column in B[6][6]

And So on...

矩阵快速转置

- 第三个循环后*row_terms* and *starting_pos*的值如下:

	[0]	[1]	[2]	[3]	[4]	[5]
row_terms =	2	1	2	2	0	1
starting_pos =	1	3	4	6	8	8

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	−15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	−6
[7]	4	0	91
[8]	5	2	28

(a)

transpose →

	row	col	value
b[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	−6
[8]	5	0	−15

(b)

矩阵快速转置

- Buildup row_term & starting_pos

starting_pos[0]=1;

starting_pos[i]=starting_pos[i-1]
+row_term[i-1]

- transpose

For columns

For elements

For columns

For elements

```
void fast_transpose(term a[], term b[])
{
/* the transpose of a is placed in b */
   int row_terms[MAX_COL], starting_pos[MAX_COL];
   int i,j, num_cols = a[0].col, num_terms = a[0].value;
   b[0].row = num_cols;  b[0].col = a[0].row;
   b[0].value = num_terms;
   if (num_terms > 0) { /* nonzero matrix */
      for (i = 0; i < num_cols; i++)
         row_terms[i] = 0;
      for (i = 1; i <= num_terms; i++)
         row_terms[a[i].col]++;
      starting_pos[0] = 1;
      for (i = 1; i < num_cols; i++)
         starting_pos[i] =
                     starting_pos[i-1] + row_terms[i-1];
      for (i = 1; i <= num_terms; i++) {
         j = starting_pos[a[i].col]++;
         b[j].row = a[i].col;   b[j].col = a[i].row;
         b[j].value = a[i].value;
      }
   }
}
```

	[0]	[1]	[2]	[3]	[4]	[5]	
row_terms	2	1	2	2	0	1	#col = 6
starting_pos	1	3	4	6	8	8	#term = 6

Matrix A :

	Row	Col	Value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

```
void fast_transpose(term a[], term b[])
{
/* the transpose of a is placed in b */
   int row_terms[MAX_COL], starting_pos[MAX_COL];
   int i,j, num_cols = a[0].col, num_terms = a[0].value;
   b[0].row = num_cols;  b[0].col = a[0].row;
   b[0].value = num_terms;
   if (num_terms > 0) { /* nonzero matrix */
     for (i = 0; i < num_cols; i++)
       row_terms[i] = 0;
     for (i = 1; i <= num_terms; i++)
       row_terms[a[i].col]++;
     starting_pos[0] = 1;
     for (i = 1; i < num_cols; i++)
       starting_pos[i] =
                    starting_pos[i-1] + row_terms[i-1];
     for (i = 1; i <= num_terms; i++) {
       j = starting_pos[a[i].col]++;
       b[j].row = a[i].col;   b[j].col = a[i].row;
       b[j].value = a[i].value;
     }
   }
}
```

I = 8

Matrix A

	Row	Col	Value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

Row Col Value

0	6	6	8
1	0	0	15
2	0	4	91
3	1	1	11
4	2	1	3
5	2	5	28
6	3	0	22
7	3	2	-6
8	5	0	-15

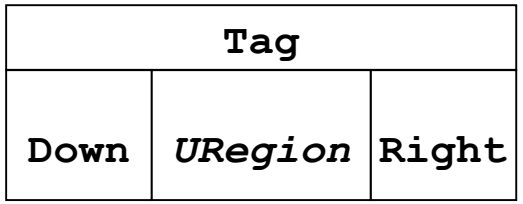
```
                [0] [1] [2] [3] [4] [5]
row_terms    = 2   1   2   2   0   1
starting_pos = 3   4   6   8   8   9
```

```
void fast_transpose(term a[], term b[])
{
/* the transpose of a is placed in b */
   int row_terms[MAX_COL], starting_pos[MAX_COL];
   int i,j, num_cols = a[0].col, num_terms = a[0].value;
   b[0].row = num_cols;  b[0].col = a[0].row;
   b[0].value = num_terms;
   if (num_terms > 0) { /* nonzero matrix */
      for (i = 0; i < num_cols; i++)
         row_terms[i] = 0;
      for (i = 1; i <= num_terms; i++)
         row_terms[a[i].col]++;
      starting_pos[0] = 1;
      for (i = 1; i < num_cols; i++)
         starting_pos[i] =
                     starting_pos[i-1] + row_terms[i-1];
      for (i = 1; i <= num_terms; i++) {
         j = starting_pos[a[i].col]++;
         b[j].row = a[i].col;   b[j].col = a[i].row;
         b[j].value = a[i].value;
      }
   }
}
```

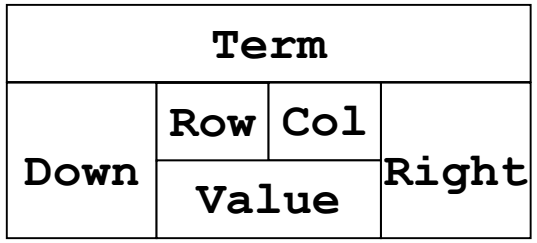
2.6.3稀疏矩阵和舞蹈链

稀疏矩阵的十字链表结构

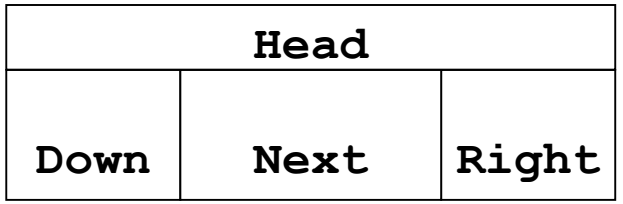
- 用一个标识域Tag来区分头结点和非0元素结点：
- 头节点的标识值为“Head”，矩阵非0元素结点的标识值为“Term”。



(a) 结点的总体结构

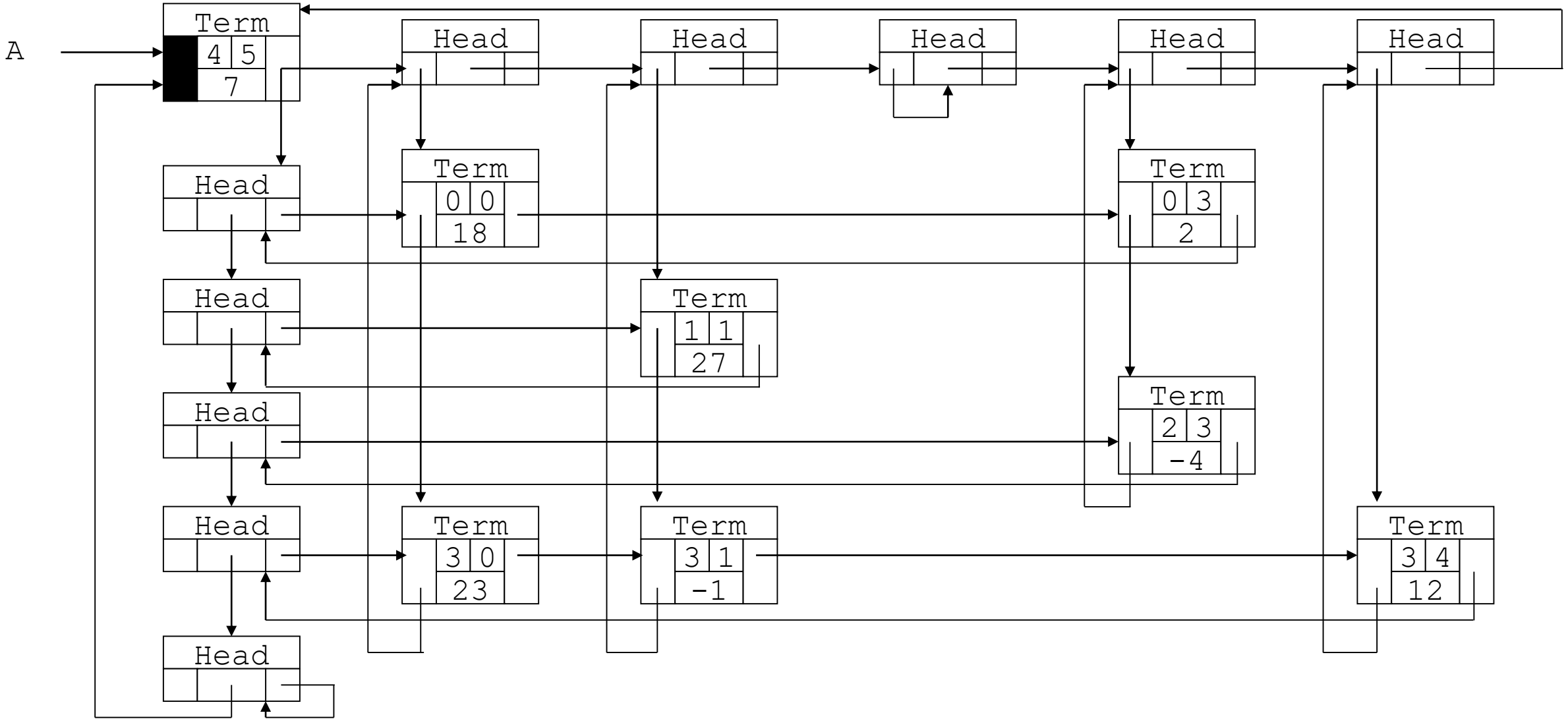


(b) 矩阵非0元素结点



(c) 头结点

稀疏矩阵的十字链表实现的一个例子



2.6.3稀疏矩阵和舞蹈链

2. 舞蹈链

- **拟解决问题**：给定一个元素值为1或者0的n行m列的矩阵，需要从中挑选若干行形成新矩阵，要求新矩阵的每一列有且仅有一个元素为1。
- **精确覆盖问题**：从矩阵中选取若干行，使得这些行中的1“精确地覆盖”原矩阵的每一列。

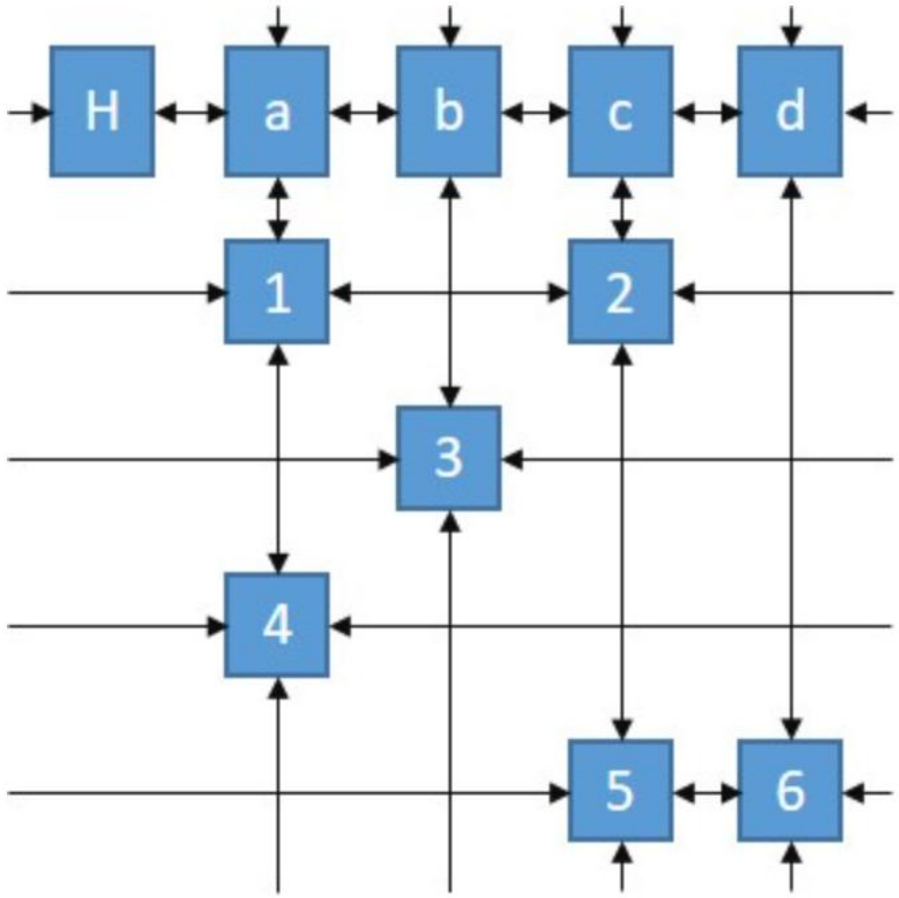
$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

- D.E. Knuth提出X算法：采用回溯搜索在矩阵中寻找合适的行实现“精确覆盖”
- X算法的执行需要大量的矩阵行和列的删除以及恢复操作，需要良好的矩阵数据结构支持。因此，提出**舞蹈链**：双向循环十字链表
  - 例如，在舞蹈链上，可以快速删除矩阵的某一行：只需要找到该行的行首结点，随后从行首结点开始遍历该行所有结点，将这些结点从所在的列链表中删除即可。
  - 同样，可以快速地向矩阵第r行、第c列插入一个元素。

2.6.3稀疏矩阵和舞蹈链

2. 舞蹈链

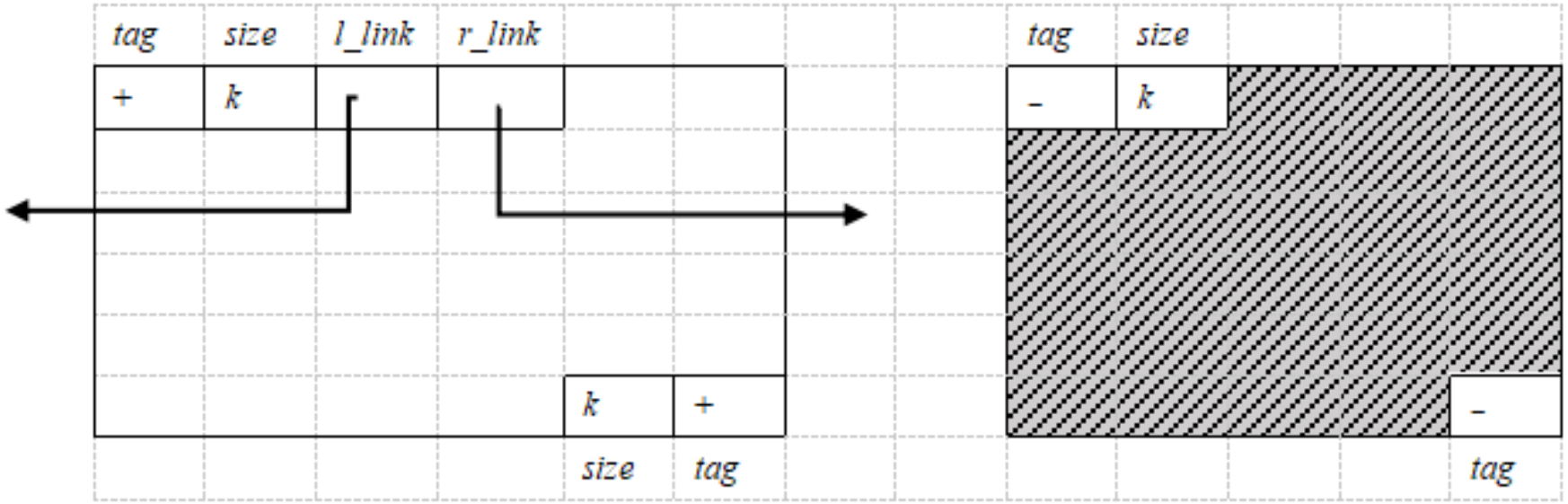
$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$



- **舞蹈链**：双向循环十字链表

2.7 应用场景：内存管理

- 操作系统空闲内存管理：空闲块的管理，处理申请和回收；
- 内存空闲块组织成双向链表
- 空闲块分配策略：首次适配、最佳适配
- 空闲块回收：空闲块合并



(a) 空闲块结构　　（b）已分配块结构

2.8 小结

- **线性表**：若干数据元素组成的有序序列，其基本操作有插入、删除、查找等
- **线性表实现方法**：顺序存储、链式存储
- **双向链表**：通过设置分别指向后继和前驱结点的两个指针，实现从前向后和从后向前两个方向的访问
- **循环链表**：最后结点的向后指针指向链表的头结点；双向链表头结点的向前指针还指向链表最后结点
- **静态链表**：物理存储依赖于顺序存储结构、逻辑链接关系采用链表思想
- **块状链表**：在传统的链表结构之上采用“分块”的思想进行改进，平衡了查找操作的时间复杂度和插入/删除操作的时间复杂度
- **广义表**：对一般线性表的推广，是一种“表中有表”的数据元素组织方式
- **多维数组**：将元素在空间上按行顺序存储，实现类似一维数组的随机访问
- **特殊矩阵和稀疏矩阵**可以进行压缩存储，减少所需要的存储空间；稀疏矩阵通过十字链表记录非零元素
- **舞蹈链**：一种采用双向链表链接的十字链表，可以较好地支持矩阵的整行删除、整列删除以及恢复操作

谢谢观看