第4章 串和数组

提纲

CONTENTS

4.1 串

4.1.1 串的基本概念

- 串是由零个或多个字符组成的有限序列。记作str="$a_0a_1 \ldots a_{n-1}$"（$n \geqslant 0$）。
- 串中所包含的字符个数n称为串长度，当$n=0$时，称为空串。
- 一个串中任意连续的字符组成的子序列称为该串的子串。
- 包含子串的串相应地称为主串。
- 若两个串的长度相等且对应字符都相等，则称两个串相等。

【例4.1】设s是一个长度为n的串，其中的字符各不相同，则s中的所有子串个数是多少？

- 空串是其子串，计1个。
- 每个字符构成的串是其子串，计n个。
- 每2个连续的字符构成的串是其子串，计$n-1$个。
- 每3个连续的字符构成的串是其子串，计$n-2$个。
- …
- 每$n-1$个连续的字符构成的串是其子串，计2个。
- s是其自身的子串，计1个。

子串个数

$=1+n+(n-1)+\cdots+2+1$

$=n(n+1)/2+1$

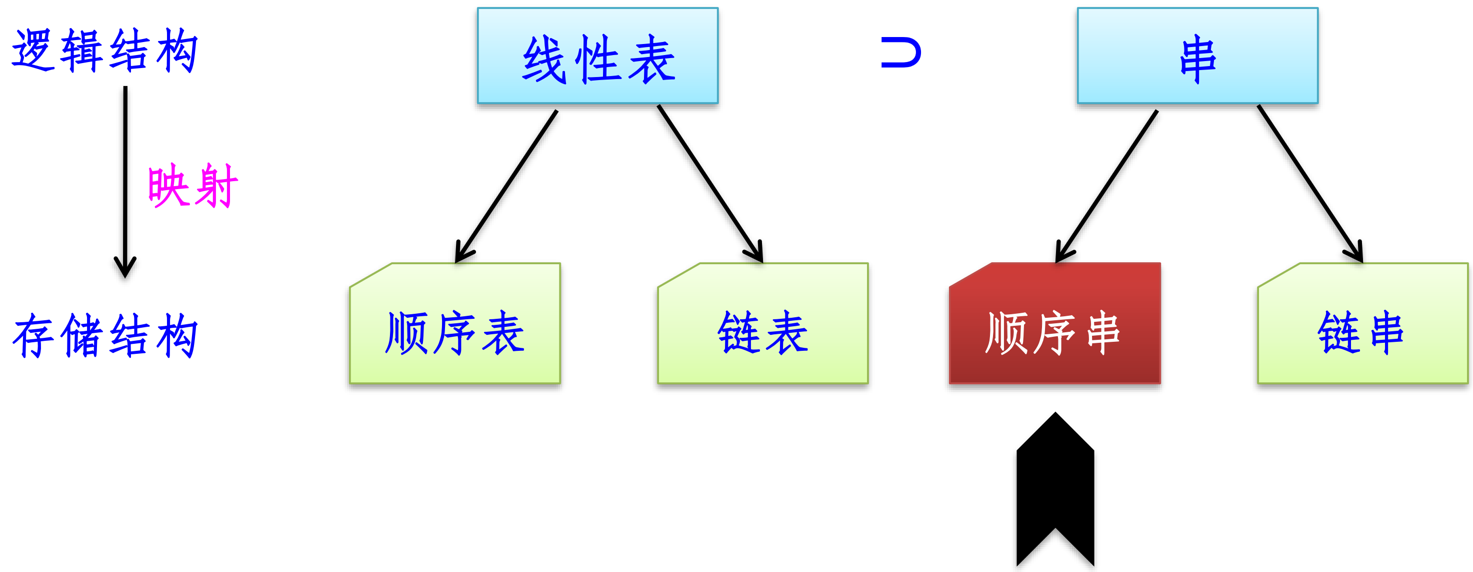
例如，s="software"的子串个数=(8×9)/2+1=37。

4.1.2 串的抽象数据类型

```
ADT String
{
数据对象：
   D={a_i | 0≤i≤n-1，n≥0，a_i为字符类型}
数据关系：
   R={r}
   r={<a_i，a_{i+1}> | a_i，a_{i+1}∈D， i=0，…，n-2}
基本运算：
   StrAssign(cstr)：由字符串常量cstr创建一个串，即生成其值等于cstr的串。
   StrCopy()：串复制，返回由当前串复制产生一个串。
   getsize()：求串长，返回当前串中字符个数。
   geti(i)：返回序号i的字符。
   seti( i，x)：设置序号i的字符为x。
   Concat(t)：串连接，返回一个当前串和串t连接后的结果。
   SubStr(i，j)：求子串，返回当前串中从第i个字符开始的j个连续字符组成的子串。
   InsStr(i，t)：串插入，返回串t插入到当前串的第i个位置后的子串。
   DelStr(i，j)：串删除，返回当前串中删去从第i个字符开始的j个字符后的结果。
   RepStr(i，j，t)：串替换，返回用串t替换当前串中第i个字符开始的j个字符后的结果。
   DispStr()：输出字符串。
}
```

4.1.2 串的存储结构

串的实现方式



1. 串的顺序存储结构–顺序串

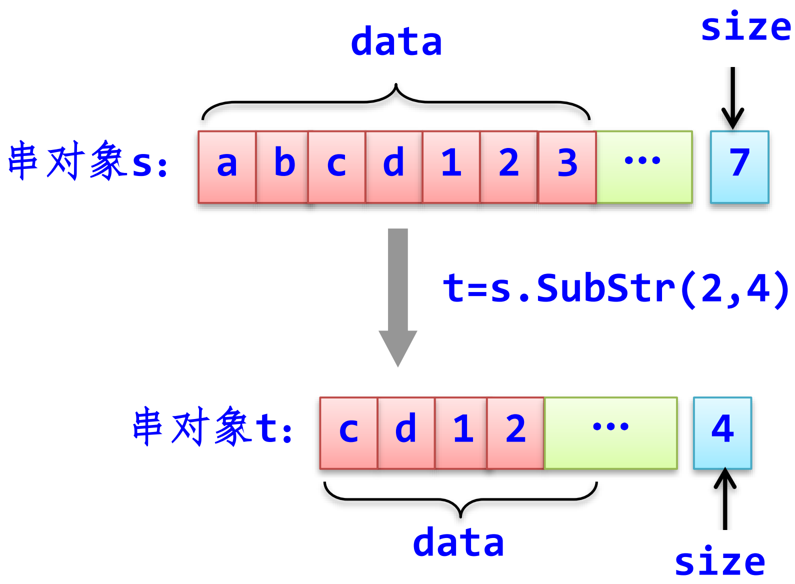
- 和顺序表一样，用一个data数组和一个整型变量size来表示一个顺序串，size表示data数组中实际字符的个数。
- 为了简单，data数组采用固定容量为MaxSize（可以模仿顺序表改为动态容量方式）。

顺序串类SqString

```
MaxSize=100                                 #假设容量为100
class SqString:                             #顺序串类
  def __init__(self):                       #构造方法
    self.data=[None]*MaxSize                #存放串中字符
    self.size=0                             #串中字符个数
  #串的基本运算算法
```

顺序串上的基本运算算法设计与顺序表类似，仅以求子串为例说明。

求子串：对于一个顺序串求序号*i*开始长度为*j*的子串。



实现： 先创建一个空串*s*，当参数正确时，*s*子串的字符序列为data[*i*..*i*+*j*-1]，共*j*个字符，当*i*和*i*+*j*-1不在有效序序号0～size-1范围内时，则参数错误，此时返回空串。

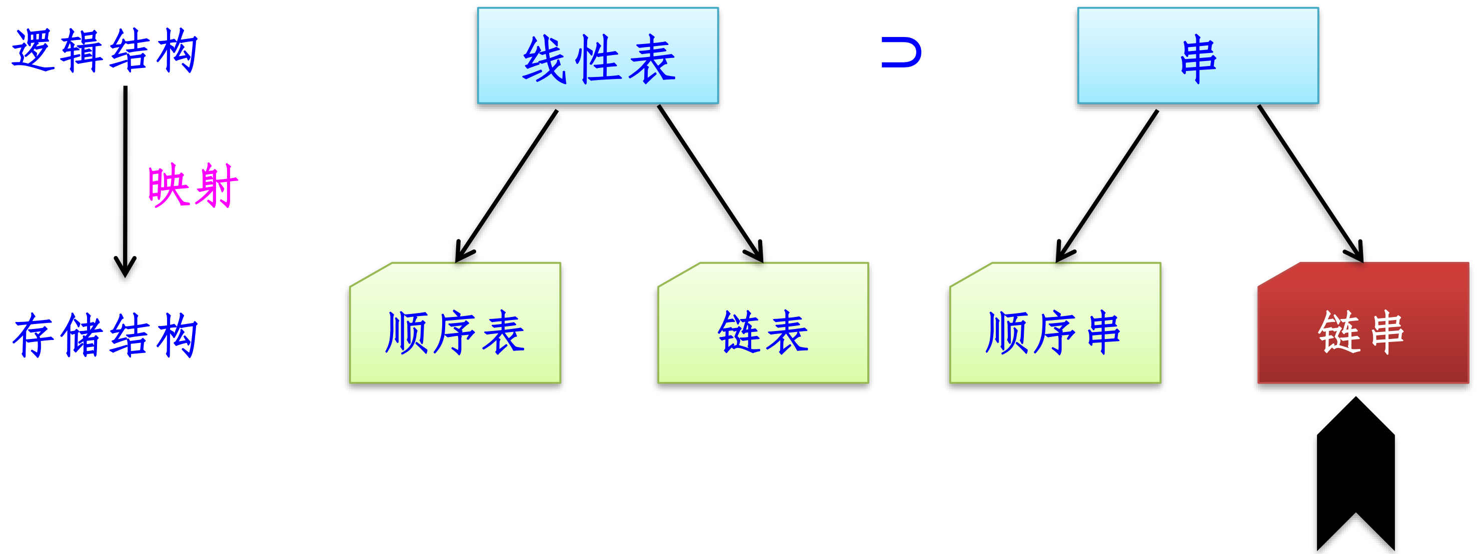
```
def SubStr(self,i,j):                                  #求子串的运算算法
  s=SqString()                                          #新建一个空串
  assert i>=0 and i<self.size and j>0 and i+j<=self.size   #检测参数
  for k in range(i,i+j):                                #将data[i..i+j-1]->s
    s.data[k-i]=self.data[k]
  s.size=j
  return s                                              #返回新建的顺序串
```

【例4.2】设计一个算法Strcmp(*s*, *t*)，以字典顺序比较两个英文字母串*s*和*t*的大小，假设两个串均以顺序串存储。

```
def Strcmp(s,t):                          #比较串s和t的算法
  minl=min(s.getsize(),t.getsize())       #求s和t中最小长度
  for i in range(minl):                   #在共同长度内逐个字符比较
    if s[i]>t[i]: return 1
    elif s[i]<t[i]: return -1
  if s.getsize()==t.getsize():            #s==t
    return 0
  elif s.getsize()>t.getsize():           #s>t
    return 1
  else: return -1                         #s<t
```

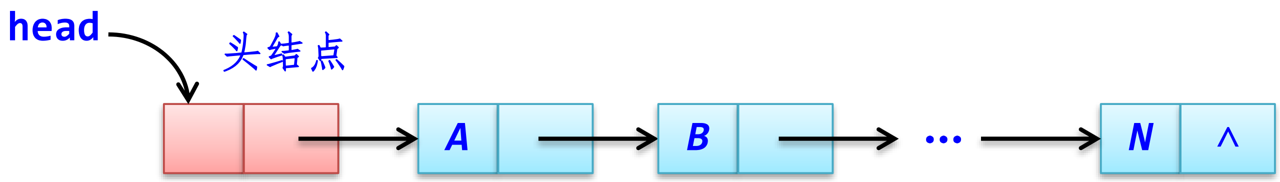
2. 串的链式存储结构–链串

串的实现方式

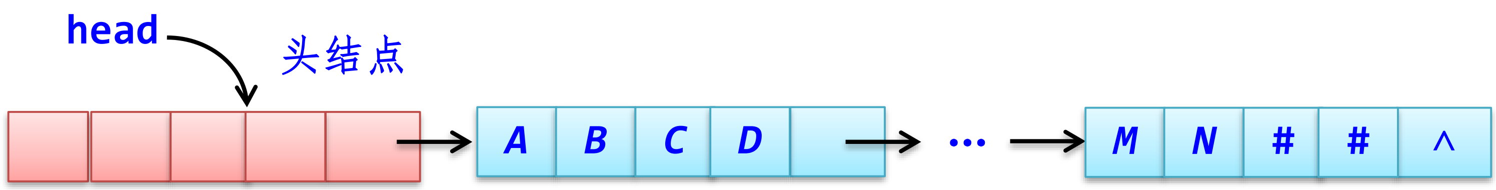


用带头结点的单链表表示链串

例如，*s*= "ABCDEFGHIJKLMN"，共14个字符。



结点大小=1



结点大小=4

链串的结点类型LinkNode（结点大小为1）

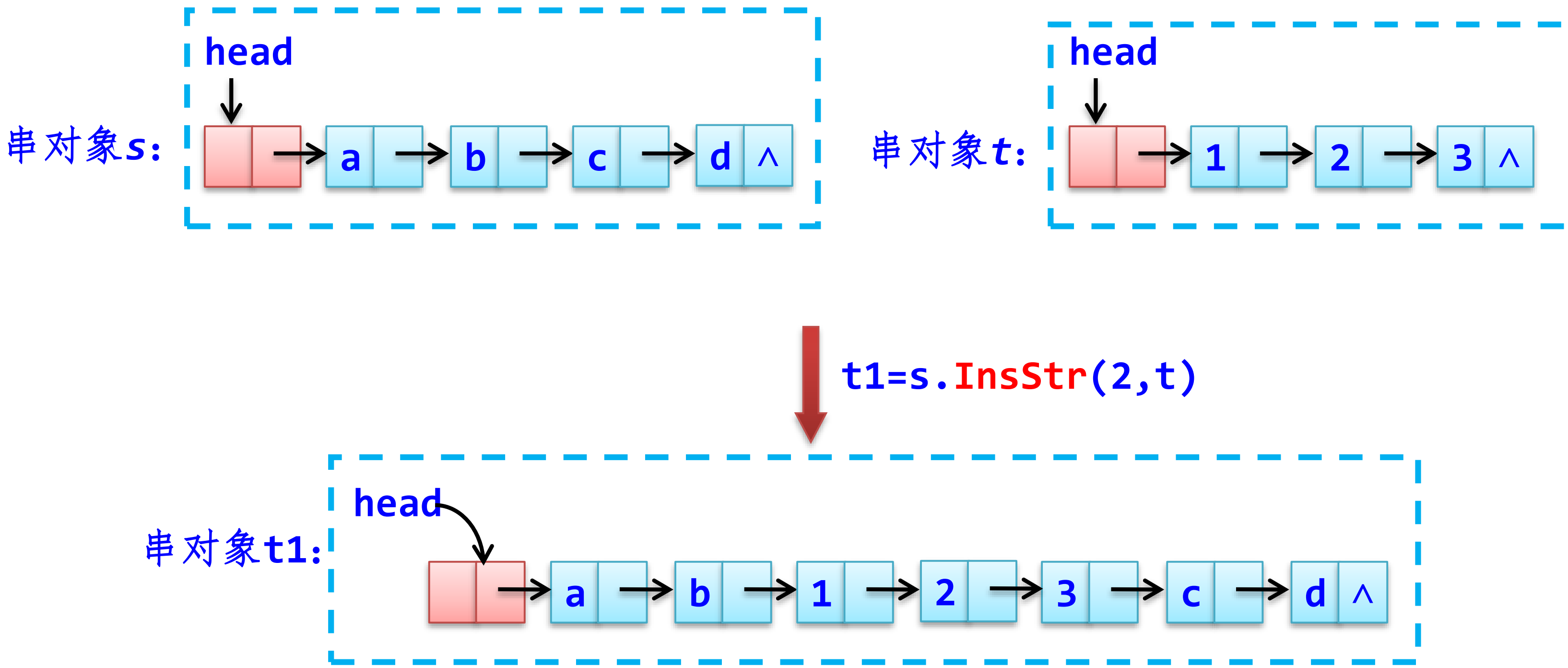
```
class LinkNode:                         #链串结点类型
  def __init__(self,d=None):            #构造方法
    self.data=d                         #存放一个字符
    self.next=None                      #指向下一个结点的指针
```

一个链串用一个头结点head来唯一标识，链串类LinkString

```
class LinkString:                    #链串类
  def __init__(self):                #构造方法
    self.head=LinkNode()             #建立头结点
    self.size=0
  #串的基本运算算法
```

链串上的基本运算算法设计与单链表类似，仅以串插入算法为例说明。

串插入：链串在序号i位置插入串t



实现：先创建一个空串s，当参数正确时，采用尾插法建立结果串s：

（1）将当前链串的前i个结点复制到s中。

（2）将t中所有结点复制到s中。

（3）再将当前串的余下结点复制到s中。

```
def InsStr(self,i,t):                              #串插入运算的算法
  s=LinkString()                                   #新建一个空串
  assert i>=0 and i<self.size                      #检测参数
  p,p1=self.head.next, t.head.next
  r=s.head                                         #r指向新建链表的尾结点
  for k in range(i):                               #将当前链串的前i个结点复制到s
    q=LinkNode(p.data)
    r.next=q; r=q                                  #将q结点插入到尾部
    p=p.next
```

```
  while p1!=None:                          #将t中所有结点复制到s
    q=LinkNode(p1.data)
    r.next=q;                              #将q结点插入到尾部
    p1=p1.next

  while p!=None:                           #将p及其后的结点复制到s
    q=LinkNode(p.data)
    r.next=q; r=q                          #将q结点插入到尾部
    p=p.next

s.size=self.size+t.size
r.next=None                                #尾结点的next置为空
return s                                   #返回新建的链串
```

4.1.3 串的模式匹配

- 设有两个串*s*和*t*，串*t*定位操作就是在串*s*中查找与子串*t*相等的子串。
- 通常把串*s*称为目标串，把串*t*称为模式串，因此定位也称作模式匹配。
- 模式匹配成功是指在目标串*s*中找到一个模式串*t*。
- 不成功则指目标串*s*中不存在模式串*t*。

1. BF算法

思路

目标串s="$s_0s_1\cdots s_{n-1}$"，模式串t="$t_0t_1\cdots t_{m-1}$"

- 第1趟：从s_0/t_0开始比较，若相等，则继续逐个比较后续字符。如果对应的字符全部相同且t的字符比较完，说明t是s的子串，返回t在s中的起始位置，表示匹配成功；如果对应的字符不相同，说明第一趟匹配失败。
- 第2趟：从s_1/t_0开始比较，若相等，则继续逐个比较后续字符。如果对应的字符全部相同且t的字符比较完，说明t是s的子串，返回t在s中的起始位置，表示匹配成功；如果对应的字符不相同，说明第一趟匹配失败。
- 依次类推。只要有一趟匹配成功，则说明t是s的子串，返回t在s中的起始位置。如果i超界都没有匹配成功，说明t不是s的子串，返回-1。

例如，设目标串s="aaaaab"，模式串t="aaab"

第1趟匹配

s="a a a a a b"

t="a a a b"

i=3

j=3

失败,修改为 i=i-j+1=1 j=0

比较4次

第2趟匹配

s="a a a a a b"

t="a a a b"

i=4

j=3

失败,修改为 i=i-j+1=2 j=0

比较4次

第3趟匹配

s="a a a a a b"

t="a a a b"

i=6

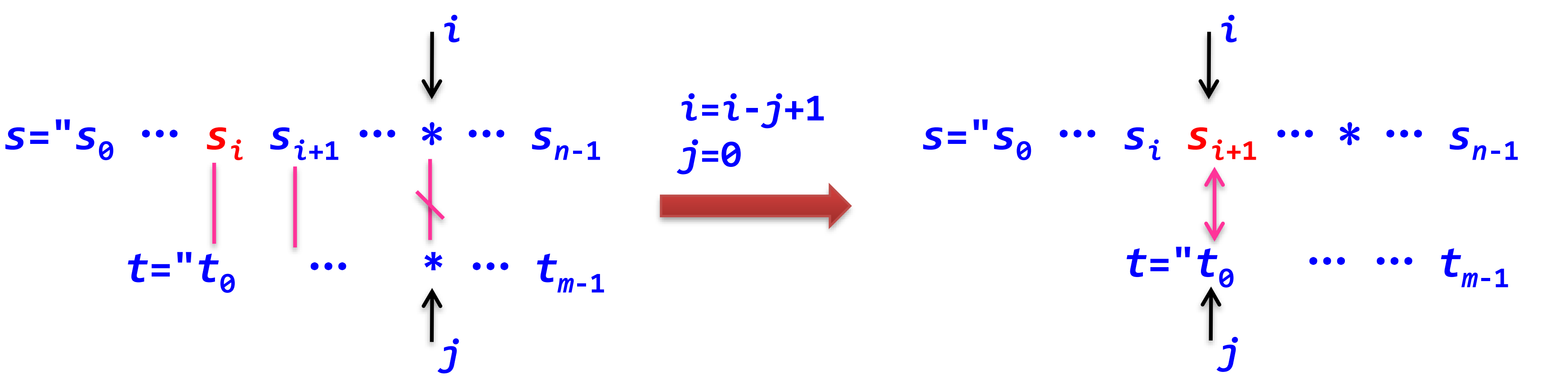
j=4

成功，返回i-t.getsize()=2

比较4次

共比较12次

```
def BF(s,t):                                          #BF算法
  i,j=0,0
  while i<s.getsize() and  j<t.getsize():             #两串未遍历完时循环
    if s[i]==t[j]:                                    #两个字符相同
      i,j=i+1,j+1                                     #继续比较下一对字符
    else:
      i,j=i-j+1,0                                     #i从下个位置，j从头开始匹配

  if j>=t.getsize():
    return (i-t.getsize())                            #返回匹配的首位置
  else:
    return (-1)                                       #模式匹配不成功
```



失配处：s_i/t_j

BF算法性能

- 该算法在最好情况下的时间复杂度为O(m)，即主串的前m个字符正好等于模式串的m个字符。
- 最坏情况下的时间复杂度为O($n \times m$)。
- 平均情况下的时间复杂度为O($n \times m$)。

2. KMP算法

基本KMP算法

主要是消除了目标串指针的回溯，从而使算法效率有了某种程度的提高。

第1趟匹配

s="a a a a a b"　　i=3

t="a a a b"　　j=3

第2趟匹配

s="a a a a a b"　　i=4

t="a a a b"　　j=3

失配处

第1趟匹配

s="a a a a a b"　　i=3

t="a a a b"　　j=3

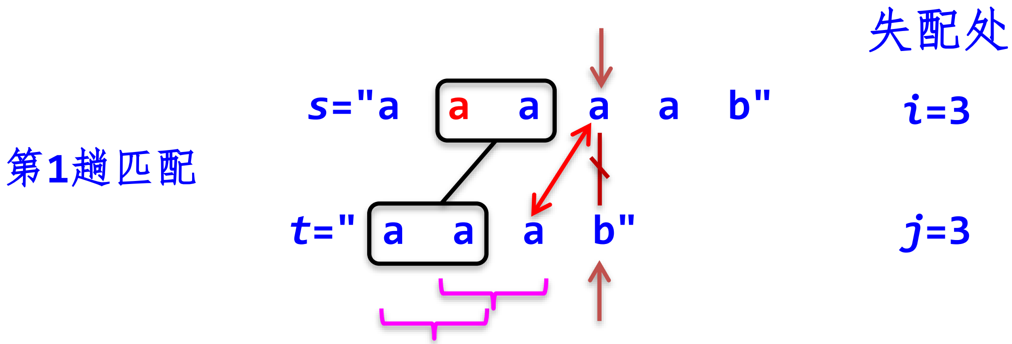
这种信息可以匹配之前获取

第2趟匹配

s="a a a a a b"

t="a a a b"

跳过第2趟匹配前面2个字符的比较



这种信息可以匹配之前获取

- t_j的前面有多少个连续字符（不含t_0）和t开头的连续字符相同！
- 用**next**数组存放，这里**next[3]=2**。
- 下一次做s_i/t_j比较。

归纳起来，求模式t的next[j]（$0 \leq j \leq m-1$）数组的公式如下：

后缀不含t_j，同时$j-k \geq 1$，即后缀至多从t_1开始而不能从t_0开始

$$
\text{next}[j]=\begin{cases}
\text{MAX}\{k \mid 0<k<j \text{ 且 } \text{"}\overbrace{t_0 t_1 \ldots t_{k-1}}^{\text{前缀}}\text{"}=\text{"}\overbrace{t_{j-k} t_{j-k+1} \ldots t_{j-1}}^{t_j\text{的后缀}}\text{"}\} & \text{当前缀非空时} \\
-1 & \text{当}j=0\text{时} \\
0 & \text{其他情况}
\end{cases}
$$

均含k个字符

模式串*t*="abcac"

j	t[j]	t[j]前面的子串	前缀	后缀	相同串	next[j]
0	a					-1
1	b	a				0
2	c	ab	a	b		0
3	a	abc	a,ab	c,bc		0
4	c	abca	a,ab,abc	a,ca,bca	a	1

↓

j	0	1	2	3	4
t[j]	a	b	c	a	c
next[j]	-1	0	0	0	1

```python
def GetNext(t,next):                          #由模式串t求出next值
  j,k=0,-1
  next[0]=-1
  while j<t.getsize()-1:
    if k==-1 or t[j]==t[k]:                   #j遍历后缀，k遍历前缀
      j,k=j+1,k+1
      next[j]=k

    else:
      k=next[k]                               #k置为next[k]
```

next[*k*] ⇨ next[*k*+1]

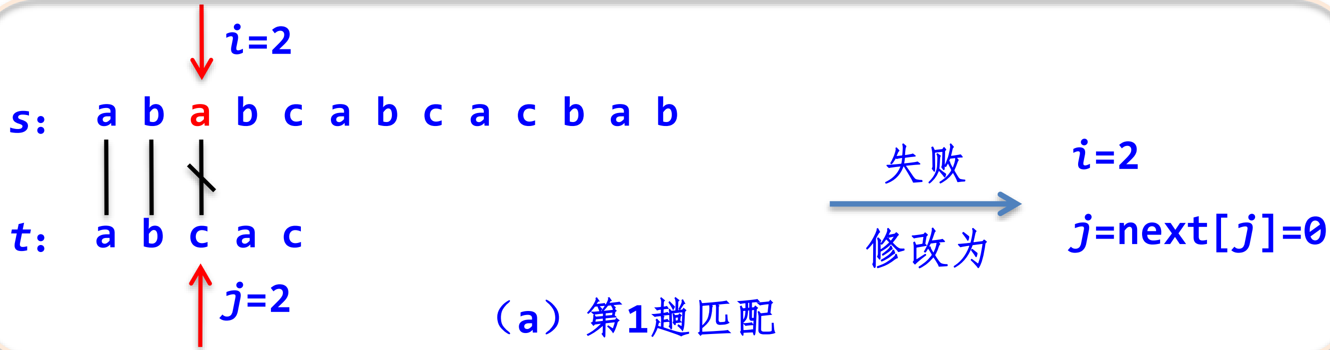
```python
def KMP(s,t):                              #KMP算法
  next=[None]*MaxSize
  GetNext(t,next)                          #求next数组

  i,j=0,0
  while i<s.getsize() and j<t.getsize():
    if j==-1 or s[i]==t[j]:
      i,j=i+1,j+1                          #i,j各增1
    else:
      j=next[j]                            #i不变,j回退

  if j>=t.getsize():
    return(i-t.getsize())                  #返回起始序号
  else:
    return(-1)                             #返回-1
```

KMP算法性能

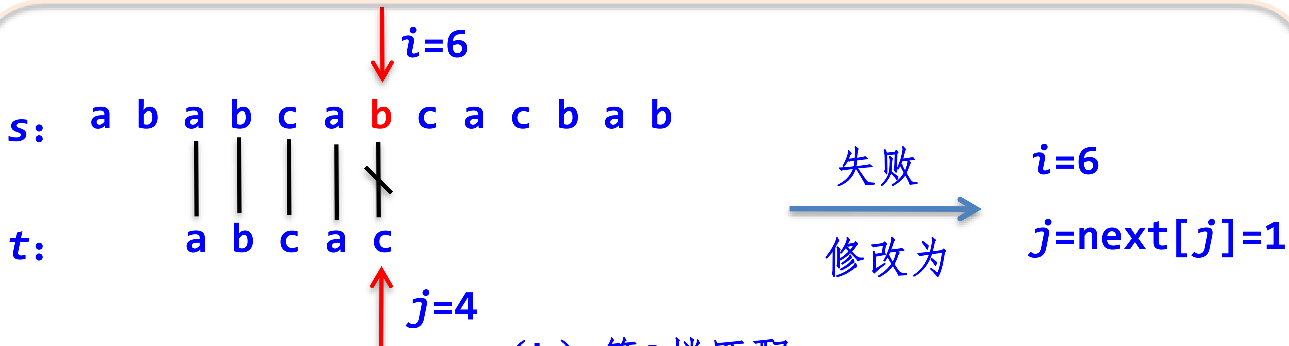
- 设目标串*s*的长度为*n*，模式串*t*长度为*m*。
- 在KMP算法中求next数组的时间复杂度为O(*m*)。
- 在后面的匹配中因主串*s*的下标*i*不减即不回溯，比较次数可记为*n*。
- KMP算法总的时间复杂度为O(*n*+*m*)。

【例4.6】设目标串s="ababcabcacbab"，模式串t="abcac"。给出KMP进行模式匹配的过程。

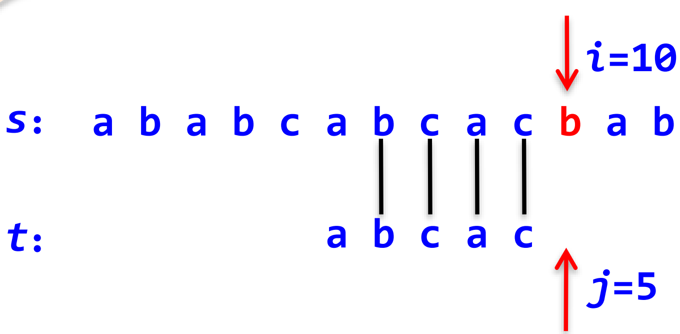
j	0	1	2	3	4
t[j]	a	b	c	a	c
next[j]	-1	0	0	0	1



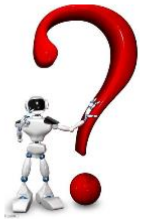
（a）第1趟匹配



（b）第2趟匹配



（c）第3趟匹配成功，返回i-t.getsize()=5



- KMP算法的性能提高了吗?
- KMP算法跳过了中间一些趟，正确吗?

问题1

以目标串*s*="aaaaab"，模式串*t*="aaab"为例。

j	0	1	2	3
t[j]	a	a	a	b
next[j]	**-1**	**0**	**1**	**2**

j	0	1	2	3
t[j]	a	a	a	b
next[j]	**-1**	**0**	**1**	**2**

第1趟匹配

s="a a a a a b"

t="a a a b"

$i=3$

$j=3$

失败,修改为 → i不变 j=next[3]=2

比较4次

第2趟匹配

s="a a a a a b"

t="a a a b"

$i=4$

$j=3$

失败,修改为 → i不变 j=next[3]=2

比较2次

第3趟匹配

s="a a a a a b"

t="a a a b"

$i=6$

$j=4$

成功, 返回2

比较2次

共比较8次<12次

问题2 以目标串s="ababcabcacbab"，模式串t="abcac"为例。

j	0	1	2	3	4
t[j]	a	b	c	a	c
next[j]	-1	0	0	0	1

i=2

s: a b a b c a b c a c b a b

t: a b c a c

j=2

→ i=2

j=next[j]=0

跳过了s_1的那一趟？

s: a b a b c a b c a c b a b

t: a b c a c

- 失配处为s_2/t_2。有"s_1"="t_1"
- BF下一趟"$s_1s_2s_3$…"/"$t_0t_1t_2$…"
- next[2]=0 ⇨ "t_1"≠"t_0"
- 即"s_1"≠"t_0",所有s_1开始的匹配是没有必要的！

改进KMP算法

基本KMP算法存在的问题

设目标串*s*="aaabaaaab"，模式串*t*="aaaab"。

j	0	1	2	3	4
t[*j*]	a	a	a	a	b
next[*j*]	-1	0	1	2	3

j	0	1	2	3	4
$t[j]$	a	a	a	a	b
next[j]	-1	0	1	2	3

第1趟匹配

i=3

s: a a a b a a a a b

t: a a a a b

j=3

失败 → 修改为

i=3

j=next[j]=2

第2趟匹配

i=3

s: a a a b a a a a b

t: a a a a b

j=2

失败 → 修改为

i=3

j=next[j]=1

j	0	1	2	3	4
$t[j]$	a	a	a	a	b
next[j]	-1	0	1	2	3

第3趟匹配

i=3

s: a a a b a a a a b

t: a a a a b

j=1

失败 → 修改为

i=3

j=next[j]=0

第4趟匹配

i=3

s: a a a b a a a a b

t: a a a a b

j=0

失败 → 修改为

i=3

j=next[j]=-1

修改为 →

i=i+1=4

j=j+1=0

第5趟匹配

i=9

s: a a a b a a a a b

t: a a a a b

j=5

返回i-t.getsize()=4

设主串s="aaabaaaab"，模式串t="aaaab"。

j	0	1	2	3	4
$t[j]$	a	a	a	a	b
next[j]	-1	0	1	2	3

i=3/j=3

s_i=b

next[j]=2

$t_j=t_{next[j]}$

⇨ $s_i \neq t_{next[j]}$

i=3/j=next[j]=2

s_i=b

next[j]=1

$t_j=t_{next[j]}$

⇨ $s_i \neq t_{next[j]}$

i=3/j=next[j]=1

s_i=b

next[j]=0

$t_j=t_{next[j]}$

⇨ $s_i \neq t_{next[j]}$

i=3/j=next[j]=0

s_i=b

next[j]=-1

j	0	1	2	3	4
$t[j]$	a	a	a	a	b
next[j]	-1	0	1	2	3
nextval[j]	-1	-1	-1	-1	3

- 首先，**nextval[0]=-1**
- j**=1**：失配处为s_i/t_1，则$s_i \neq t_1$。**KMP**算法的下一次比较$s_i/t_{next[1]}$，而**next[1]=0**，并且$t_0=t_1$，说明一定有$s_i \neq t_{next[1]}$ ⇨ **nextval[j]=nextval[next[j]]=-1**

j	0	1	2	3	4
$t[j]$	a	a	a	a	b
next[j]	-1	0	1	2	3
nextval[j]	-1	-1	-1	-1	3

将next数组改为nextval数组，与next[0]一样，先置nextval[0]=-1。假设求出next[j]=k，现在失配处为s_i/t_j，即$s_i \neq t_j$，

（1）如果有$t_j=t_k$成立，可以直接推出$s_i \neq t_k$成立，没有必要再做s_i/t_k的比较，直接置nextval[j]=nextval[k]（nextval[next[j]]），即下一步做$s_i/t_{\text{nextval}[j]}$的比较。

（2）如果有$t_j \neq t_k$，没有改进的，置nextval[j]=next[j]。

求出next：

- $t_j=t_k$：nextval[j]=nextval[k]
- 否则：nextval[j]=next[j]=k

```
def GetNextval(t,nextval):          #由模式串t求出nextval值
  j,k=0,-1
  nextval[0]=-1
  while j<t.getsize()-1:
    if k==-1 or t[j]==t[k]:
      j,k=j+1,k+1
      if t[j]!=t[k]:
        nextval[j]=k
      else:                         #t[j]=t[k]
        nextval[j]=nextval[k]
    else: k=nextval[k]
```

将next改为nextval即可

```
def KMPval(s,t):                              #改进后的KMP算法
  nextval=[None]*MaxSize
  GetNextval(t,nextval)                       #求nextval数组
  i,j=0,0
  while i<s.getsize() and j<t.getsize():
    if j==-1 or s[i]==t[j]:
      i,j=i+1,j+1                             #i,j各增1
    else: j=nextval[j]                        #i不变,j回退

  if j>=t.getsize():
     return(i-t.getsize())                    #返回起始序号
  else:
     return(-1)                               #返回-1
```

本算法的时间复杂度也为O(n+m)。

【例4.7】设*s*="aaabaaaab"，*t*="aaaab"。计算模式串*t*的nextval函数值。并画出利用改进KMP算法进行模式匹配时每一趟的匹配过程。

j	0	1	2	3	4
t[*j*]	a	a	a	a	b
next[*j*]	-1	0	1	2	3
nextval[*j*]	-1	-1	-1	-1	3

j	0	1	2	3	4
$t[j]$	a	a	a	a	b
next[j]	-1	0	1	2	3
nextval[j]	-1	-1	-1	-1	3

第1趟匹配

i=3

s: a a a b a a a a b

t: a a a a b

j=3

失败 修改为 → i=3, j=nextval[j]=-1

修改为 → i=i+1=4, j=j+1=0

第2趟匹配

i=9

s: a a a b a a a a b

t: a a a a b

j=5

返回i-t.length=4

【例】设目标串为s="abcaabbabcabaacbacba"，模式串t="abcabaa"。计算模式串t的nextval函数值。并画出利用KMP算法进行模式匹配时每一趟的匹配过程。

j	0	1	2	3	4	5	6
$t[j]$	a	b	c	a	b	a	a
next[j]	-1	0	0	0	1	2	1
nextval[j]	-1	0	0	-1	0	2	1

j	0	1	2	3	4	5	6
nextval[*j*]	-1	0	0	-1	0	2	1

第1趟匹配

s="a b c a a b b a b c a b a a c b a c b a"

t="a b c a b a a"

i=4 $\xrightarrow{\text{失败}}$ *i*=4

j=4 修改为 *j*=nextval[4]=0

第2趟匹配

s="a b c a a b b a b c a b a a c b a c b a"

t="a b c a b a a"

i=6 $\xrightarrow{\text{失败}}$ *i*=6

j=2 修改为 *j*=nextval[2]=0

j	0	1	2	3	4	5	6
nextval[*j*]	-1	0	0	-1	0	2	1

第3趟匹配

s="a b c a a b b a b c a b a a c b a c b a"

t="a b c a b a a"

i=6 *j*=0 $\xrightarrow{\text{失败}}$ 修改为 *i*=6 *j*=nextval[0]=-1 $\longrightarrow$ 修改为 *i*=*i*+1=7 *j*=*j*+1=0

第4趟匹配

s="a b c a a b b a b c a b a a c b a c b a"

t="a b c a b a a"

i=14 *j*=7 成功，返回*i*-t.getsize()=7

4.2 数 组

4.2.1 数组的基本概念

- 数组是一个二元组（**idx**，**value**）的集合，对每个**idx**，都有一个**value**值与之对应。**idx**称为下标，可以由一个整数、两个整数或多个整数构成，下标含有d（d≥1）个整数称为维数是d。
- 数组按维数分为一维、二维和多维数组。
- 一维数组A是n（n>1）个相同类型元素a_0，a_1，…，a_{n-1}构成的有限序列，其逻辑表示为A=（a_0，a_1，…，a_{n-1}），其中，A是数组名，a_i（0≤i≤n-1）是数组A中序号为i的元素。
- 一个二维数组可以看作是每个数据元素都是相同类型的一维数组的一维数组。
- 以此类推。

二维数组的逻辑关系用二元组表示

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

B=(D, R)

R={r_1, r_2}

r_1={<1, 2>, <2, 3>, <3, 4>, <5, 6>, <6, 7>, <7, 8>, <9, 10>,
<10, 11>, <11, 12>}　　//同行关系

r_2={<1, 5>, <5, 9>, <2, 6>, <6, 10>, <3, 7>, <7, 11>, <4, 8>,
<8, 12>}　　//同列关系

数组具有以下**特点**

（1）数组中各元素都具有统一的数据类型。

（2）d（$d \geqslant 1$）维数组中的非边界元素具有d个前驱元素和d个后继元素。

（3）数组维数确定后，数据元素个数和元素之间的关系不再发生改变，特别适合于顺序存储。

（4）每个有意义的下标都存在一个与其相对应的数组元素值。

d维数组抽象数据类型

```
ADT Array
{
数据对象:
    D={ 数组中所有元素 }
数据关系:
    R={r₁, r₂, …, r_d}
    r_i={ 元素之间第i维的线性关系 | i=1, …, d}
基本运算:
   Value(A, i₁, i₂, …, i_d): A是已存在的d维数组，其运算结果是返回
                                A[i₁, i₂, …, i_d]值。
   Assign(A, e, i₁, i₂, …, i_d): A是已存在的d维数组，其运算结果是
                                置A[i₁, i₂, …, i_d]=e。
    …
}
```

数组的主要操作是存取元素值，没有插入和删除操作，所以数组通常采用顺序存储方式来实现。

1. 一维数组

- 一维数组的所有元素依逻辑次序存放在一片连续的内存存储单元中。
- 其起始地址为第一个元素a_0的地址即LOC(a_0)。
- 假设每个数据元素占用k个存储单元。
- 则任一数据元素a_i的存储地址LOC(a_i)就可由以下公式求出

$$\mathrm{LOC}(a_i)=\mathrm{LOC}(a_0)+i\times k \qquad (1\leqslant i<n)$$

↓

一维数组具有随机存储特性

- 在Python中长度为n的一维数组{a_0，a_1，…，a_{n-1}}通常采用形如[a_0，a_1，…，a_{n-1}]的列表表示。
- 例如，以下语句创建一个长度为MAXN的一维数组a，初始元素值均为None：

```
MAXN=10
a=[None]*MAXN
```

2．*d*维数组

以*m*行*n*列的二维数组$A_{m\times n}=(a_{i,j})$为例讨论（二维数组也称为矩阵）。

$$\begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

$$\begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

按行优先存储

假设每个元素占k个存储单元，LOC($a_{0,0}$)表示$a_{0,0}$元素的存储地址。对于元素$a_{i,j}$:

- $a_{i,j}$前面有0～i-1共i行，每行n个元素，共有$i \times n$个元素。
- 在第i行中前面有$a[i,0..j-1]$，共j个元素。
- 合起来，$a_{i,j}$前面有$i \times n+j$个元素。

$$\mathrm{LOC}(a_{i,j})=\mathrm{LOC}(a_{0,0}) + (i \times n + j) \times k$$

$$\begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

按列优先存储

假设每个元素占k个存储单元，LOC($a_{0,0}$)表示$a_{0,0}$元素的存储地址。对于元素$a_{i,j}$:

- $a_{i,j}$前面有0～j-1共j列，每列m个元素，共有$j\times m$个元素。
- 在第j列中前面有a[0..i-1,j]，共i个元素。
- 合起来，$a_{i,j}$前面有$j\times m+i$个元素。则：

LOC($a_{i,j}$)=LOC($a_{0,0}$) + ($j\times m$ + i)$\times k$

二维数组也具有随机存储特性，以此类推。

更一般地，数组$A[c_1..d_1, c_2..d_2]$，则该数组按行优先存储时有：

$$LOC(a_{i,j})=LOC(a_{c1,c2})+[(i-c_1)\times(d_2-c_2+1)+(j-c_2)]\times k$$

按按列优先存储时有：

$$LOC(a_{i,j})=LOC(a_{c1,c2})+[(j-c_2)\times(d_1-c_1+1)+(i-c_1)]\times k$$

- 在Python中m行n列的二维数组$\{\{a_{0,0}, a_{0,1}, \cdots, a_{0,n-1}\}, \cdots, \{a_{m-1,0}, a_{m-1,1}, \cdots, a_{m-1,n-1}\}\}$通常采用形如$[[a_{0,0}, a_{0,1}, \cdots, a_{0,n-1}], \cdots, [a_{m-1,0}, a_{m-1,1}, \cdots, a_{m-1,n-1}]]$的嵌套列表表示。
- 例如，以下语句创建一个MAXM行MAXN列的二维数组a，初始元素值均为None：

```
MAXM,MAXN=3,4
a=[[None]*MAXN for i in range(MAXM)]
```

【例4.8】设有二维数组a[1..50，1..80]，其a[1][1]元素的地址为2000，每个元素占2个存储单元，若按行优先存储，则元素a[45][68]的存储地址为多少?若按列优先存储，则元素a[45][68]的存储地址为多少？

按行优先存储

- 元素a[45][68]前面有1～44行，每行80个元素，计44×80个元素。
- 在第45行中，元素a[45][68]前面有a[45][1..67]计67个元素，这样元素a[45][68]前面存储的元素个数=44×80+67。
- LOC(a[45][68])=2000+(44×80+67)×2=9174。

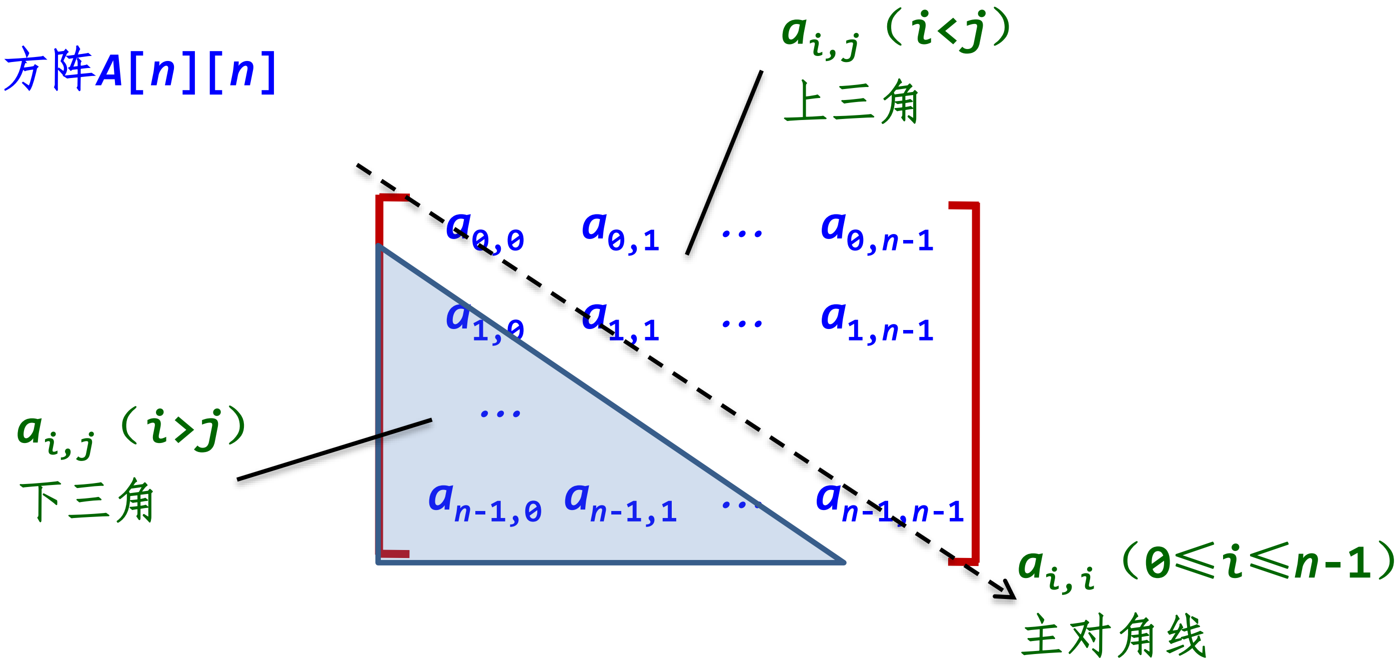
【例4.8】设有二维数组a[1..50，1..80]，其a[1][1]元素的地址为2000，每个元素占2个存储单元，若按行优先存储，则元素a[45][68]的存储地址为多少?若按列优先存储，则元素a[45][68]的存储地址为多少？

按列优先存储

- 元素a[45][68]前面有1～67列，每列50个元素，计67×50个元素。
- 在第68列中，元素a[45][68]前面有a[1..44][68]计44个元素，这样元素a[45][68]前面存储的元素个数=67×50+44。
- LOC(a[45][68])=2000+(67×50+44)×2=8788。

4.2.2 特殊矩阵的压缩存储

n阶方阵$A[n][n]$



1．对称矩阵的压缩存储

若一个n阶方阵A的元素满足$a_{i,j}=a_{j,i}$（$0\leqslant i，j\leqslant n-1$），则称其为$n$阶对称矩阵。

$$\begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \cdots & & & \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix}$$

$i\geqslant j$

下三角+主对角线

$$a_{0,0},\ a_{1,0},\ a_{1,1},\ \dots,\ a_{n-1,0},\ a_{n-1,1},\ \dots,\ a_{n-1,n-1}$$

$$B=\underbrace{(b_0,\ b_1,\ b_2,\ \dots\dots,\ \dots\dots,\ b_s)}_{n(n+1)/2\text{个元素}}$$

$$a_{i,j} \longrightarrow b_k \qquad k = ?$$

$$B=(\underbrace{\underbrace{a_{0,0}}_{\text{1个元素}},\underbrace{a_{1,0},a_{1,1}}_{\text{2个元素}},\ldots,\underbrace{a_{i-1,0},\ldots,a_{i-1,i-1}}_{i\text{个元素}},\underbrace{a_{i,0},\ldots,a_{i,j-1}}_{j\text{个元素}}}_{\text{共计}i(i+1)/2+j\text{个元素}},\overset{b_k}{\updownarrow}\!\!\!\!a_{i,j},\ldots,a_{n-1,n-1})$$

$$k=\begin{cases}\dfrac{i(i+1)}{2}+j & \text{当}i\geqslant j\text{时（下三角+主对角线的元素）}\\[2ex] \dfrac{j(j+1)}{2}+i & \text{当}i<j\text{时（}a_{i,j}=a_{j,i}\text{）}\end{cases}$$

n^2个元素 ⟷ $n(n+1)/2$个元素

$A[0..n-1,0..n-1]$ ⟷ $B[0..n(n+1)/2-1]$

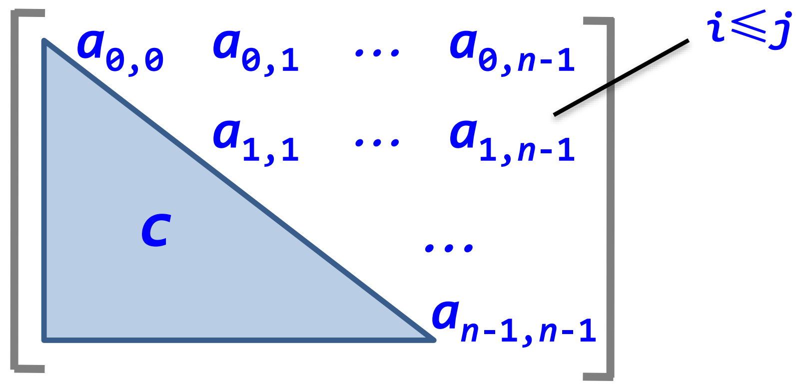
$a[i][j]$ ⟷ $b[k]$

$$k=\begin{cases}\dfrac{i(i+1)}{2}+j & \text{当}i\geq j\text{时（下三角+主对角线的元素）}\\[2ex] \dfrac{j(j+1)}{2}+i & \text{当}i<j\text{时（}a_{i,j}=a_{j,i}\text{）}\end{cases}$$

对于对称矩阵A，采用一维数组B存储，并提供A的所有运算。

2. 三角矩阵的压缩存储

上三角矩阵



对于上三角部分的元素$a_{i,j}$

$$\begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ & a_{1,1} & \cdots & a_{1,n-1} \\ & c & \cdots & \\ & & & a_{n-1,n-1} \end{bmatrix}$$

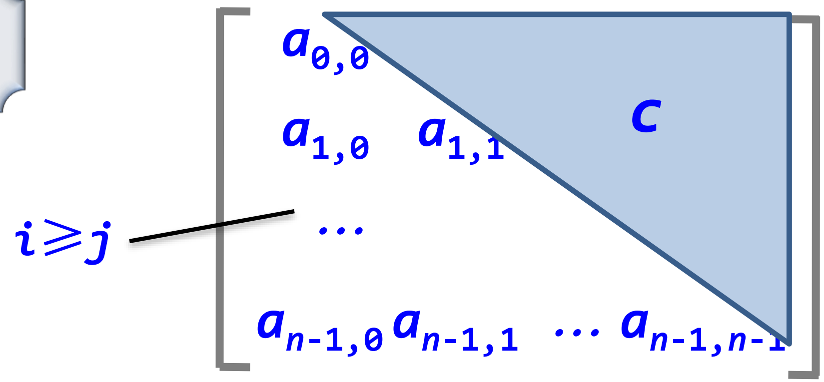
$$\left.\begin{array}{l} \text{第0行：存储}n\text{个元素} \\ \text{第1行：存储}n-1\text{个元素} \\ \cdots \\ \text{第}i-1\text{行：存储}n-i\text{个元素} \end{array}\right\} i(2n-i+1)/2\text{个元素}$$

第i行有$a[i,i..j-1]$：$j-i$个元素

$$k=\begin{cases} \dfrac{i(2n-i+1)}{2}+j-i & \text{当}i\le j\text{时} \\ \dfrac{n(n+1)}{2} & \text{当}i>j\text{时} \end{cases}$$

存放常量c

下三角矩阵



$$k=\begin{cases} \dfrac{i(i+1)}{2}+j & \text{当} i \geqslant j \text{时} \\ \dfrac{n(n+1)}{2} & \text{当} i < j \text{时} \end{cases}$$

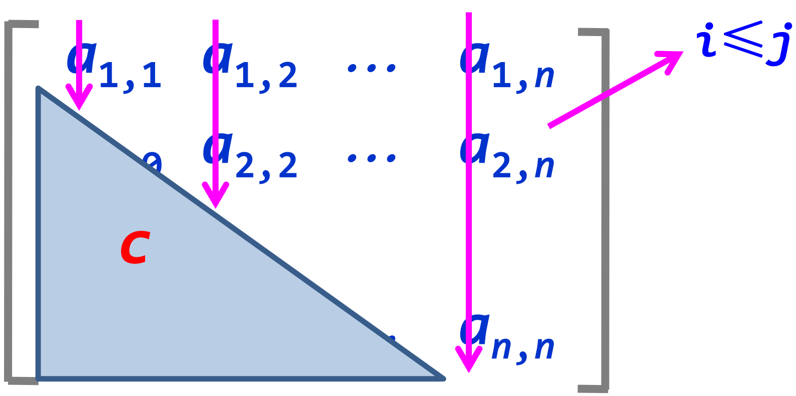
存放一个常量c

示例

若将n阶上三角矩阵A按列优先顺序压缩存放在一维数组$B[1..n(n+1)/2]$中，A中第一个非零元素$a_{1,1}$存于B数组的b_1中，则应存放到b_k中的非零元素$a_{i,j}$（$i≤j$）的下标i、j与k的对应关系是（　）。

A. $i(i+1)/2+j$　　　　B. $i(i-1)/2+j$

C. $j(j+1)/2+i$　　　　D. $j(j-1)/2+i$



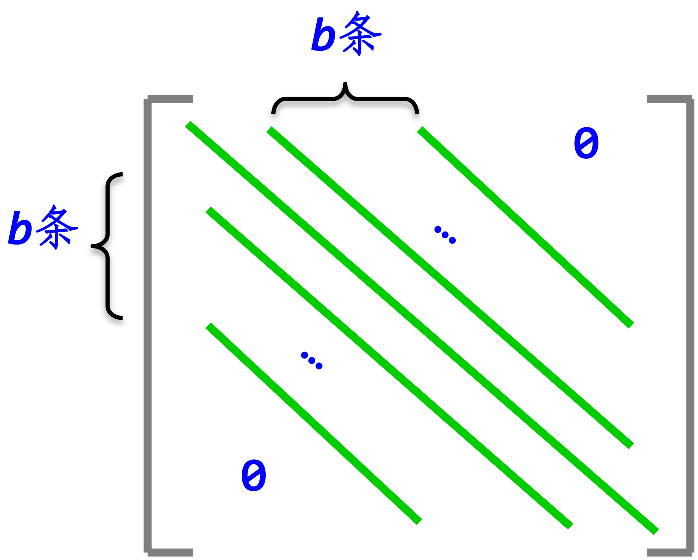
1～j-1列的元素个数：$j(j-1)/2$

第j列a_{ij}之前的元素个数：i-1

$k=j(j-1)/2+i-1+1= j(j-1)/2+i$

- **按行还是按列**
- **初始下标从0还是从1开始**

3. 对角矩阵的压缩存储



半带宽为*b*的对角矩阵

对角矩阵 ⟷ 压缩存储

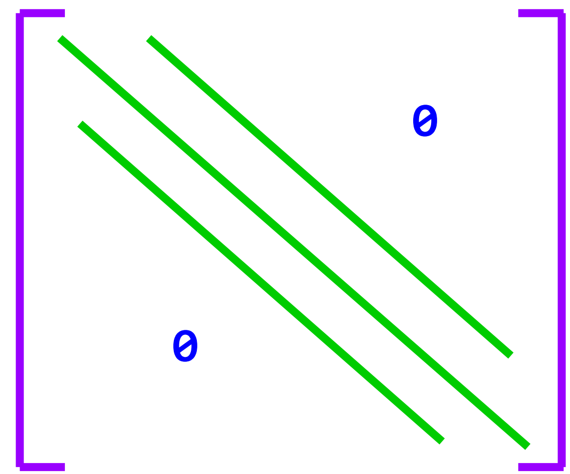
$A \longleftrightarrow B$

$a[i][j] \longleftrightarrow b[k]$

当$b=1$时称为三对角矩阵

其压缩地址计算公式如下:

$$k = 2i + j$$



4.2.3 稀疏矩阵

一个阶数较大的矩阵中的非零元素个数**s**相对于矩阵元素的总个数**t**十分小时，即**s<<t**时，称该矩阵为**稀疏矩阵**。

例如一个**100**×**100**的矩阵，若其中只有**100**个非零元素，就可称其为稀疏矩阵。

定性的描述

稀疏矩阵和特殊矩阵的不同点：

- 特殊矩阵的特殊元素（值相同元素、常量元素）分布有规律。
- 稀疏矩阵的特殊元素（非0元素）分布没有规律。

1．稀疏矩阵的三元组表示

$$A_{6\times 7}=\begin{bmatrix}0&0&1&0&0&0&0\\0&2&0&0&0&0&0\\3&0&0&0&0&0&0\\0&0&0&5&0&0&0\\0&0&0&0&6&0&0\\0&0&0&0&0&7&4\end{bmatrix}$$

i	j	$a_{i,j}$
0	2	1
1	1	2
2	0	3
3	3	5
4	4	6
5	5	7
5	6	4

通常按行优先顺序排列

三元组表示中每个元素的类定义如下：

```
class TupElem:                      #三元组元素类
  def __init__(self,r1,c1,d1):      #构造方法
    self.r=r1                       #行号
    self.c=c1                       #列号
    self.d=d1                       #元素值
```

设计稀疏矩阵三元组存储结构类TupClass如下：

```
class TupClass:                         #三元组表示类
  def __init__(self,rs,cs,ns):          #构造方法
    self.rows=rs                        #行数
    self.cols=cs                        #列数
    self.nums=ns                        #非零元素个数
    self.data=[]                        #稀疏矩阵对应的三元组顺序表
```

TupClass类中包含如下基本运算方法：

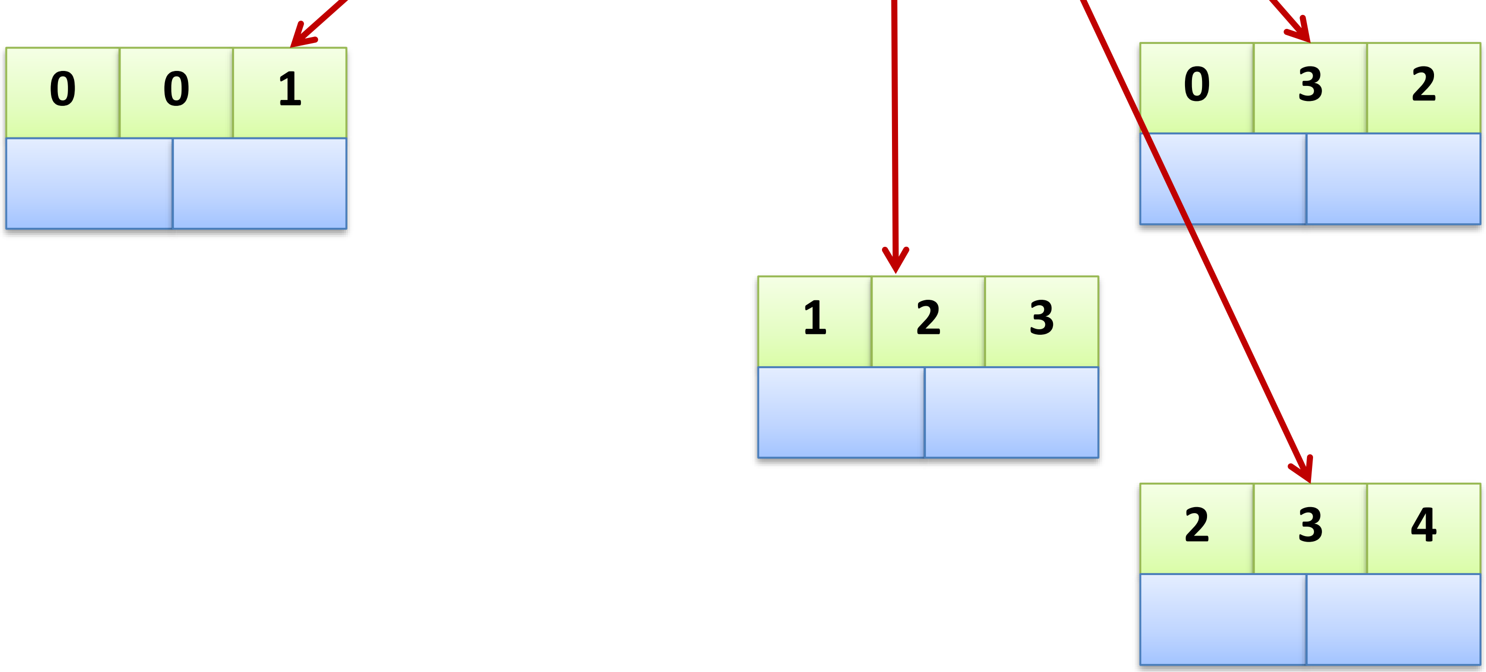
- **CreateTup(A,m,n)：** 由*m*行*n*列的稀疏矩阵*A*创建其三元组表示。
- **Setvalue(i,j,x)：** 利用三元组给稀疏矩阵的元素赋值即执行*A*[*i*][*j*]=*x*。
- **GetValue(i, j)：** 利用三元组取稀疏矩阵的元素值即执行*x*=*A*[*i*][*j*]。
- **DispTup()：** 输出稀疏矩阵的三元组表示。

其中，**data**列表用于存放稀疏矩阵中所有非零元素，通常按行优先顺序排列。这种有序结构可简化大多数稀疏矩阵运算算法。

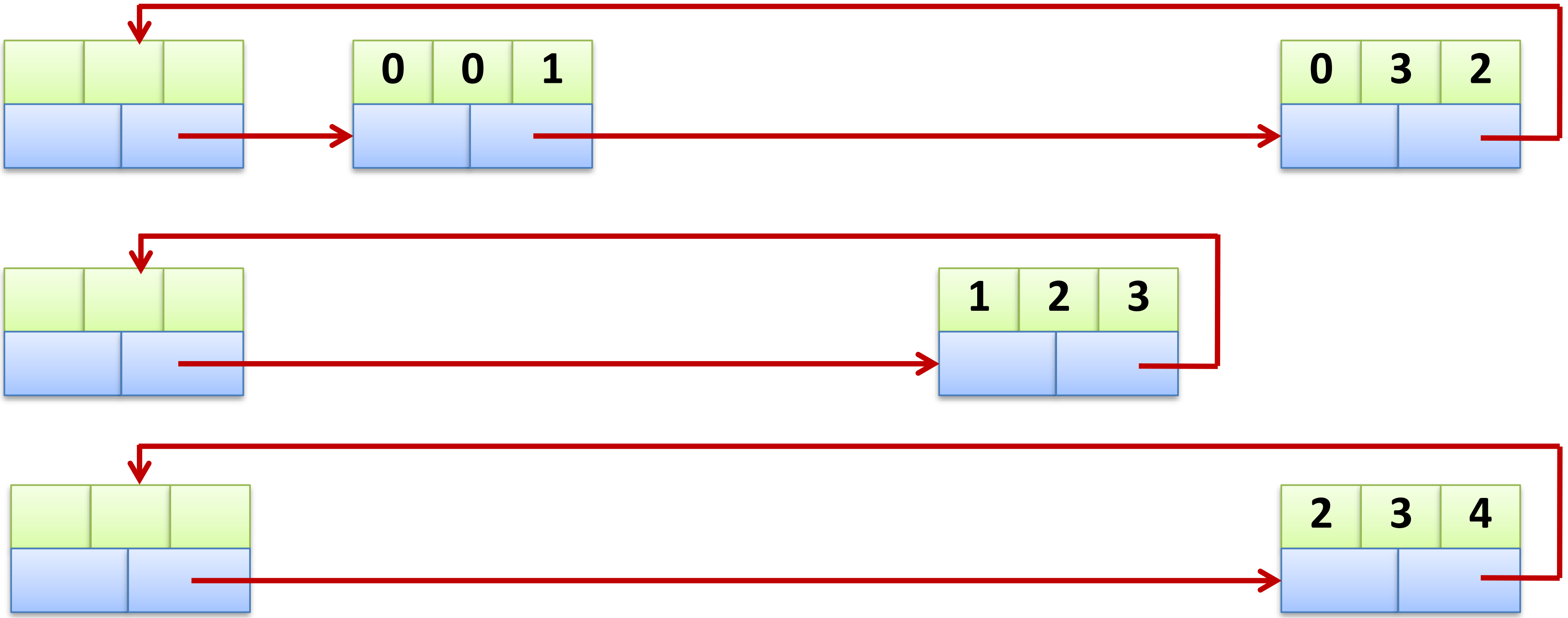
2．稀疏矩阵的十字链表表示

- 每个非零元素对应一个结点。

$$B_{3\times4}=\begin{bmatrix}1 & 0 & 0 & 2\\ 0 & 0 & 3 & 0\\ 0 & 0 & 0 & 4\end{bmatrix}$$

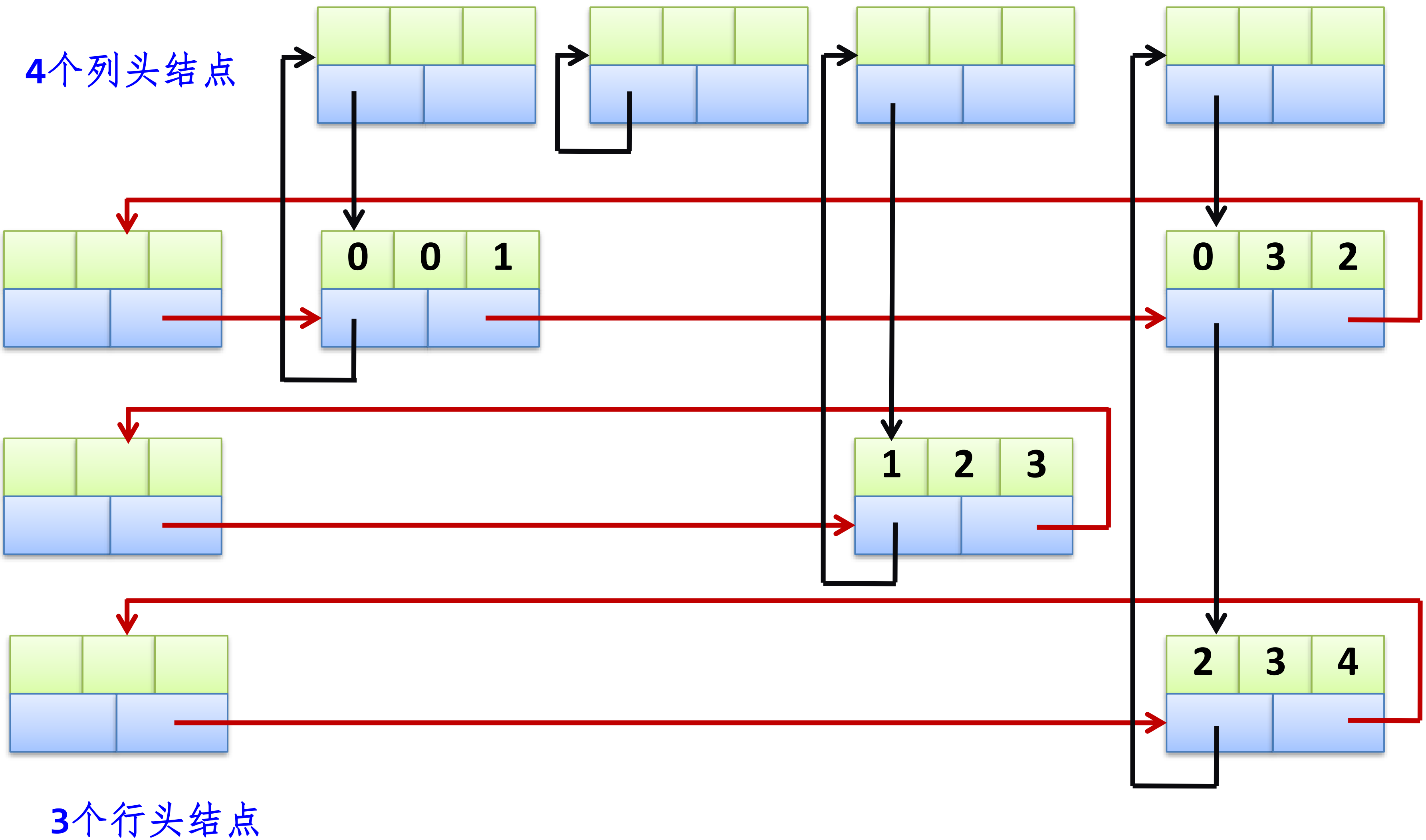


- 每行的所有结点链起来构成一个带行头结点的循环单链表。以$h[i]$（$0 \leqslant i \leqslant m-1$）作为第$i$行的头结点。

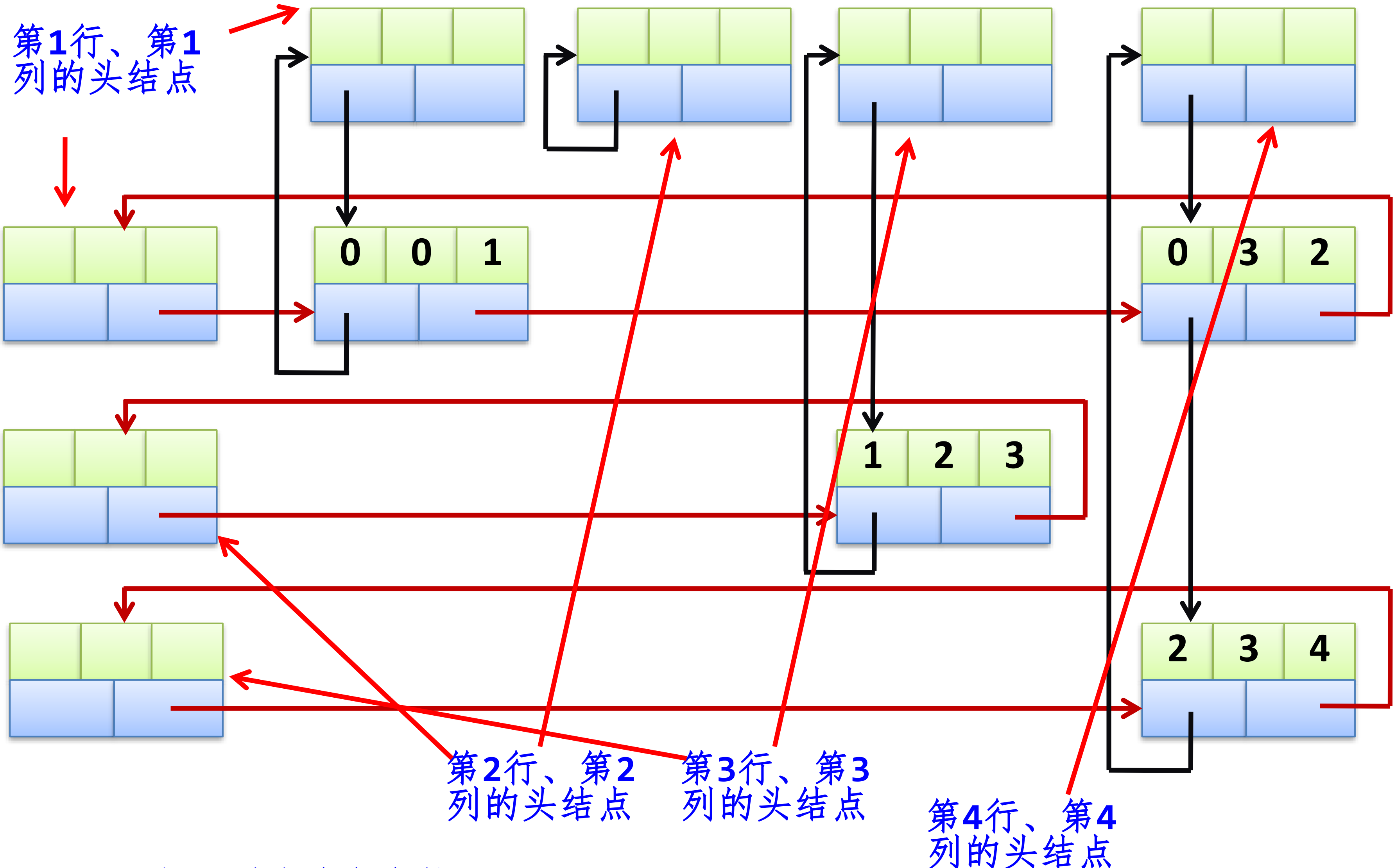


3个行头结点

- 每列的所有结点链起来构成一个带列头结点的循环单链表。以$h[i]$（$0 \leqslant i \leqslant m-1$）作为第$i$列的头结点。

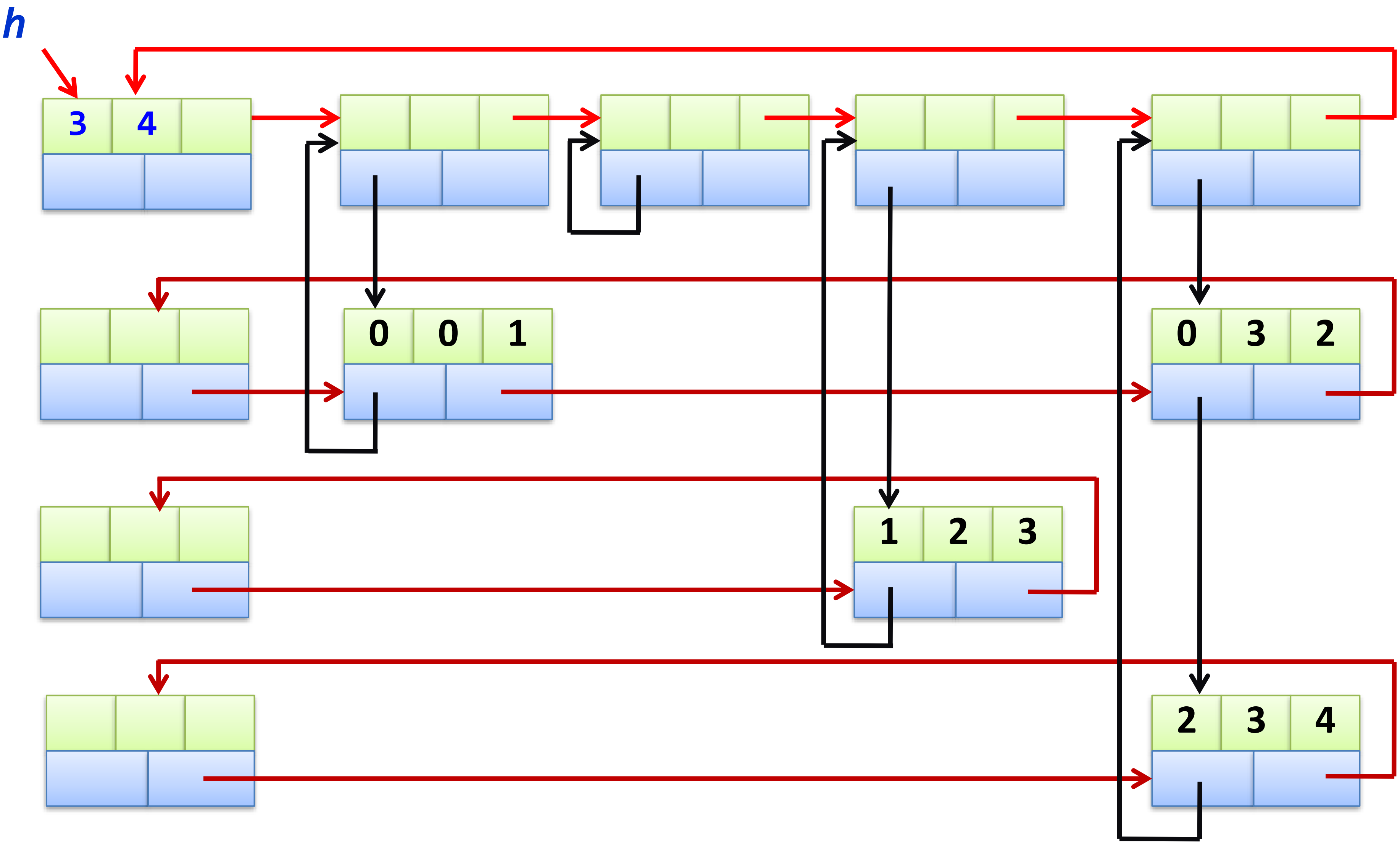


行、列头结点可以共享



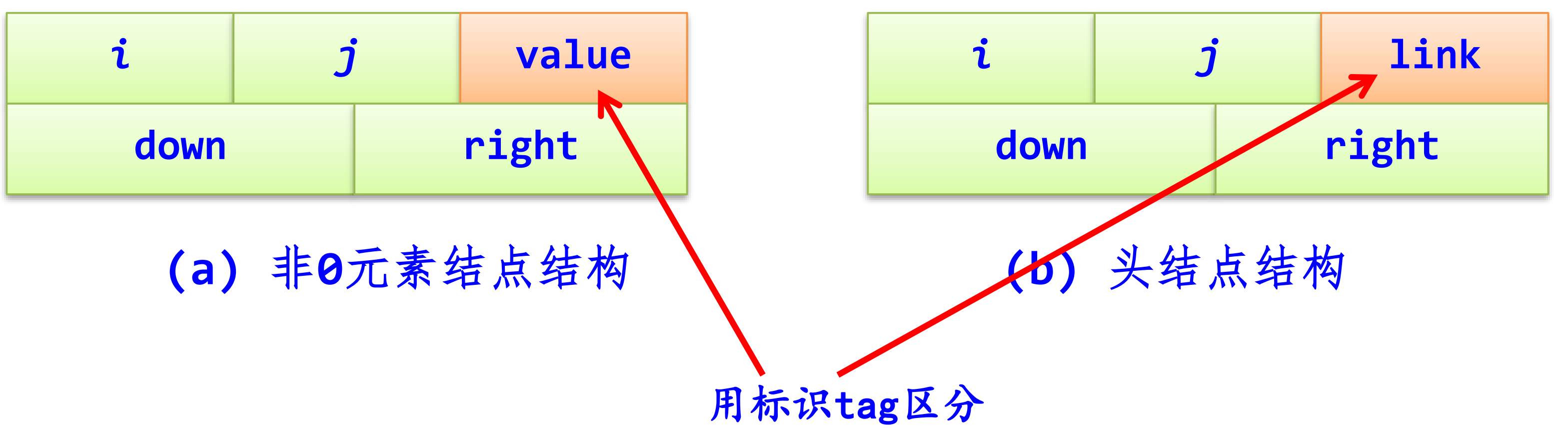
行、列头结点个数=MAX(*m*，*n*)

增加一个总头结点，并把所有行、列头结点链起来构成一个循环单链表



总的头结点个数=MAX(m,n)+1

为了统一，设计结点类型如下：

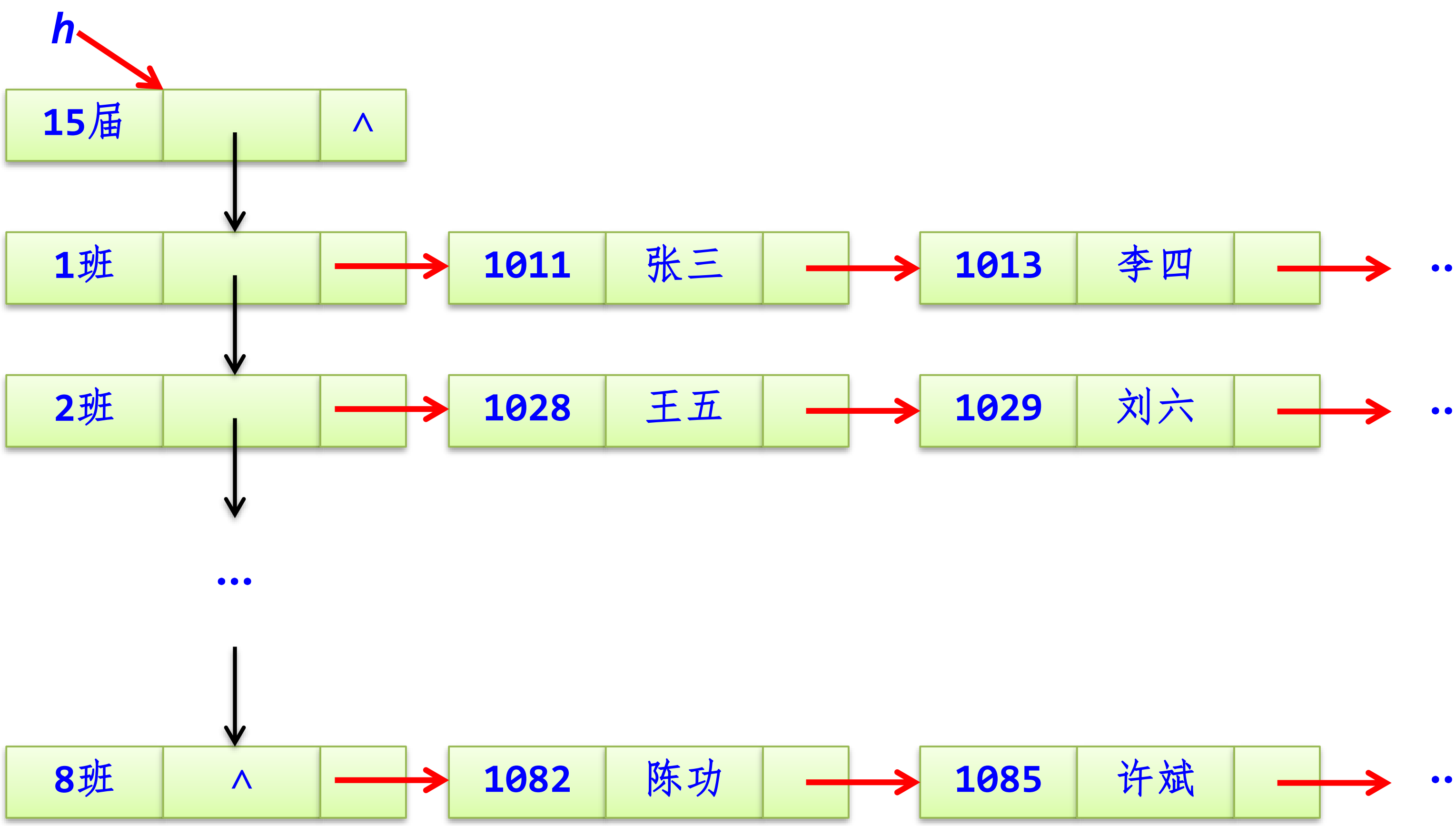


(a) 非0元素结点结构

(b) 头结点结构

示例

十字链表的启示：设计存储某年级所有学生的存储结构。



通过*h*来唯一标识学生存储结构。


成功路上并不拥挤，
因为坚持的人不多！