一、线性表

线性表的逻辑结构

线性表L定义为：

$$L = \begin{cases} (a_0, a_1, \ldots, a_{n-1}), & n \geq 1 \\ (), & n = 0 \end{cases}$$

其中，a_i 是表项（或表中元素），n是表的长度。

- 线性表由n个元素构成
- 当n=0时，L=( )表示空线性表
- 当n>1时，表中第一个元素有唯一的后继，最后一个元素有唯一的前驱
- 其余元素有唯一的后继和前驱，因而呈现线性关系

线性表常见的操作包括：

（1）计算表的长度n
（2）从左到右（或从右到左）遍历表的元素
（3）访问指定位置（第i 个）元素
（4）将新值赋予指定位置（第i 个）元素
（5）在指定位置（第i 个）元素之前（或之后）插入新元素
（6）删除指定位置（第i 个）元素

线性表的存储结构--顺序存储

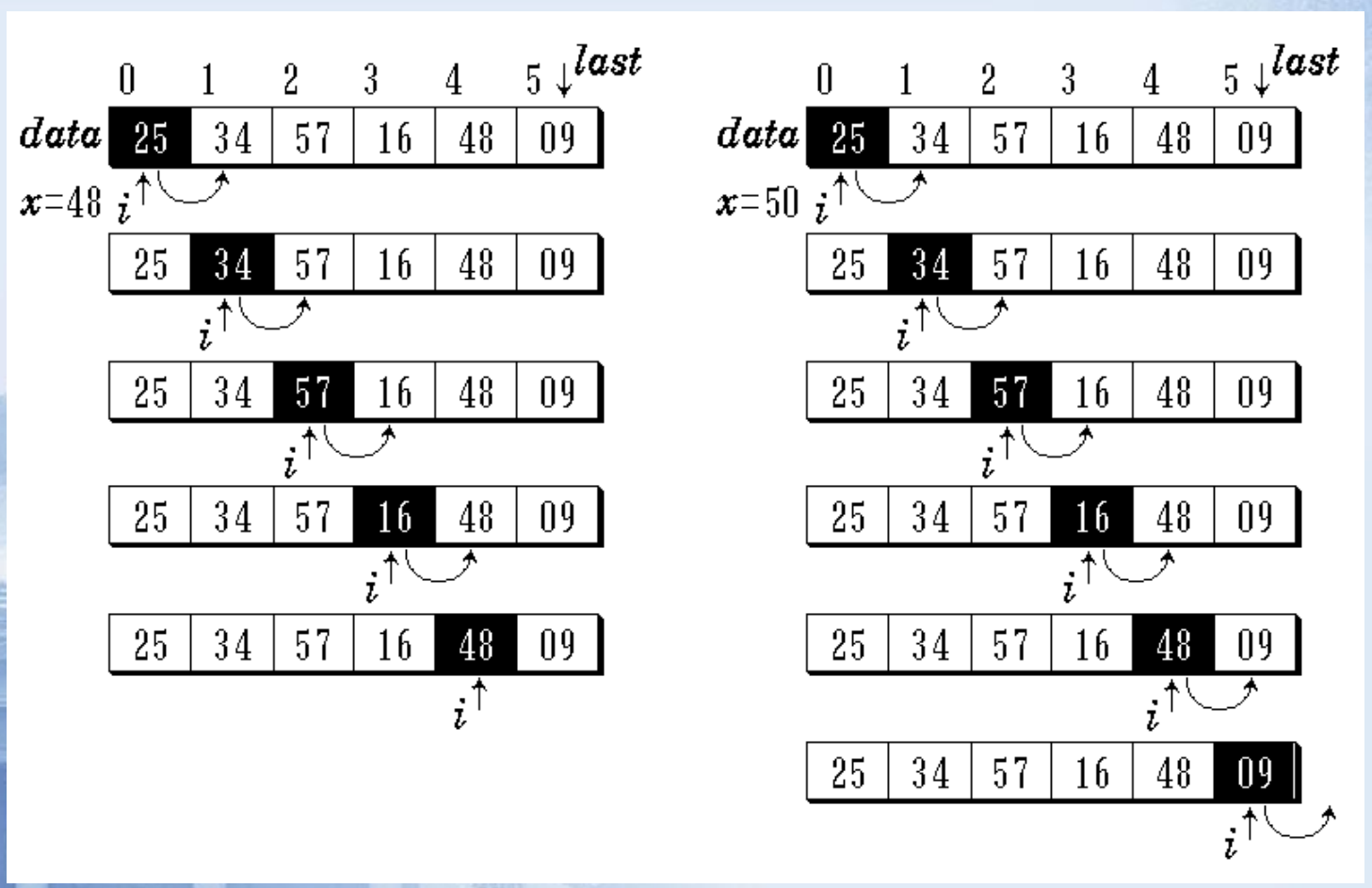
- 用程序设计语言提供的数组存放线性表。
- 数组第i 个单元与第i+1个单元在物理上是连续存放的，因此被称为顺序映射（**sequential mapping**）。

顺序表的特点：

- 各个表项的逻辑顺序和物理顺序一致；
- 对各个表项既可以顺序访问，也可以进行随机访问。

顺序表部分操作的实现与性能分析

顺序查找图示



$x = 48$

$x = 50$

平均比较次数：

$$ACN = \sum_{i=0}^{n-1} p_i \times c_i$$

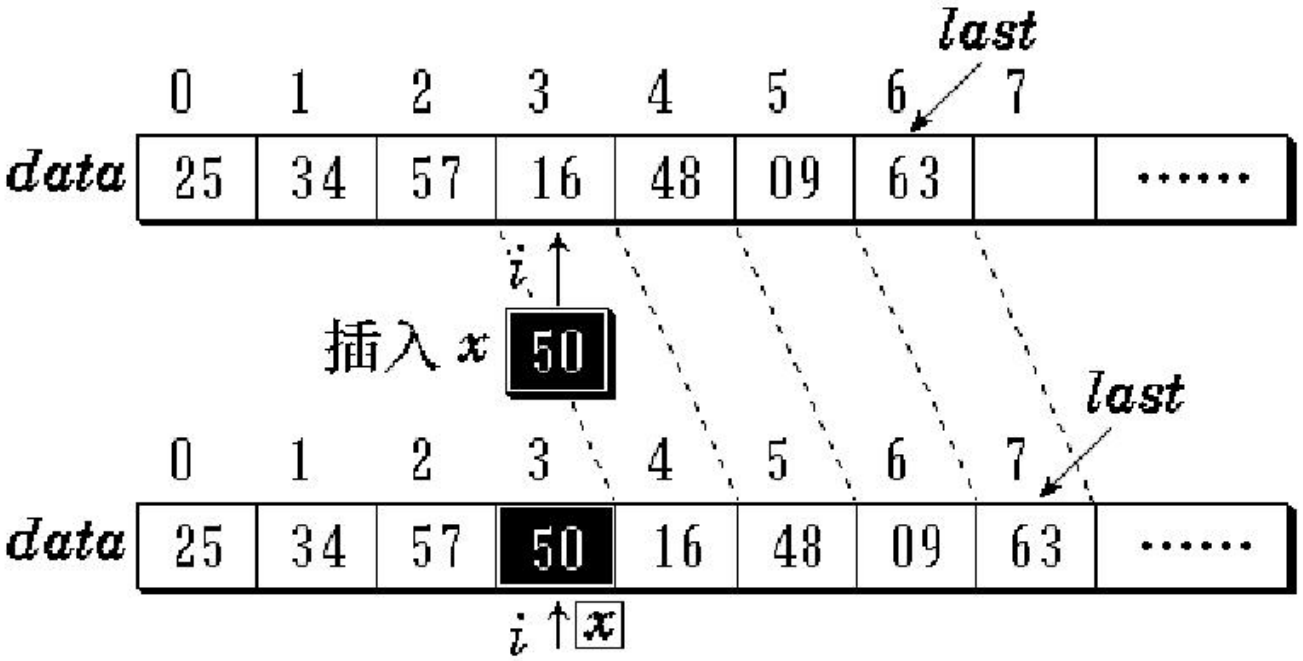
c_i为查找的是第i个元素时的比较次数，c_i= i+1。若搜索概率相等，即：$p_0=p_1=...=p_{n-1}=1/n$，则：

$$ACN = \frac{1}{n}\sum_{i=0}^{n-1}(i+1) = \frac{1}{n}(1+2+...+n)$$

$$= \frac{1}{n} \times \frac{(1+n)\times n}{2} = \frac{1+n}{2}$$

搜索不成功，数据比较 n 次

表项的插入

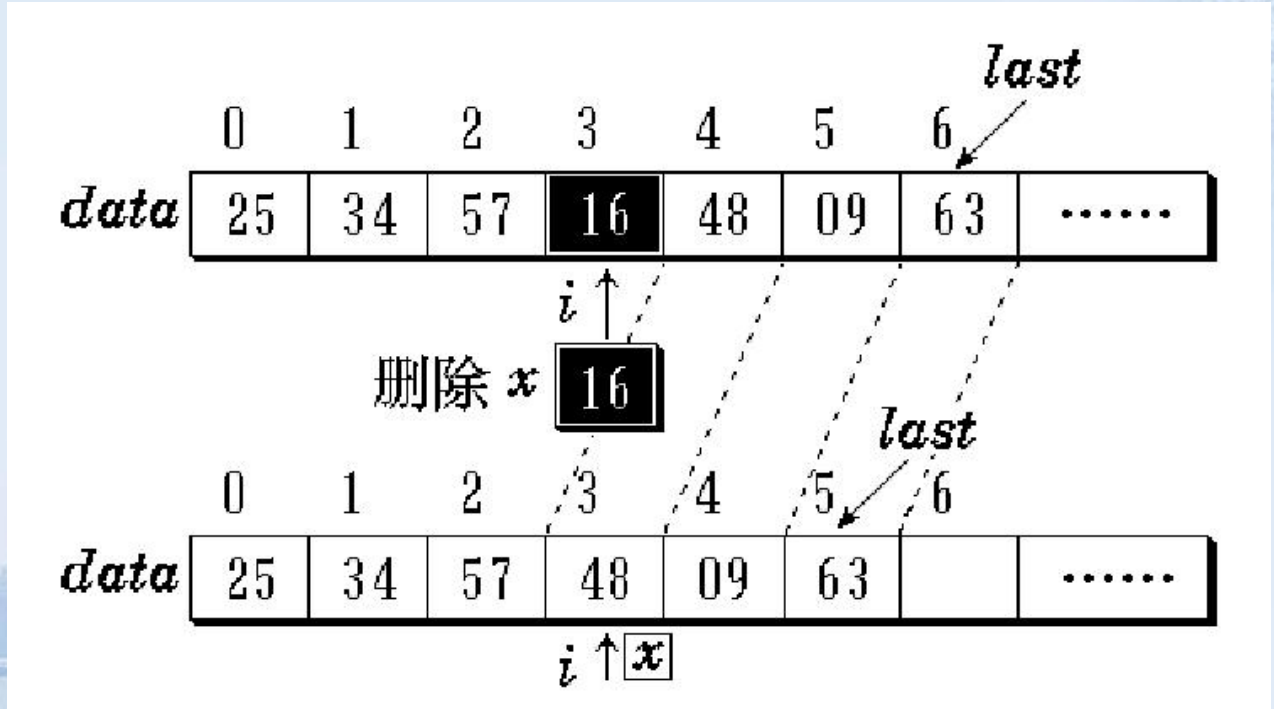


平均移动次数：

$$AMN = \frac{1}{n+1}\sum_{i=0}^{n}(n-i) = \frac{1}{n+1}(n+\ldots+1+0)$$

$$= \frac{1}{(n+1)}\frac{n(n+1)}{2} = \frac{n}{2}$$

表项的删除



平均移动次数：

$$AMN = \frac{1}{n}\sum_{i=0}^{n-1}(n-i-1) = \frac{1}{n}\frac{(n-1)n}{2} = \frac{n-1}{2}$$

- 对顺序存储的线性表做查找、插入和删除操作，其时间复杂度均为**O(n)**。
- 若对在做插入或删除操作时对表中原有元素排列顺序没有要求，则不必保持原来的顺序。
- 在插入时，把新的表项追加在表的尾部，再把表的长度加**1**；删除时，把表的最后一个表项填入被删除表项的位置，再把表的长度减**1**。
- 这样，插入、删除操作的时间复杂度为**O(1)**。

List数据类型的常用函数

1、**cmp(list1, list2)**　　比较两个列表的元素

2、**len(list)**　列表元素个数

3、**max(list)**　返回列表元素最大值

4、**min(list)**　返回列表元素最小值

5、**list(seq)**　将元组转换为列表

List数据类型的常用方法

1、**list.append(obj)** 在列表末尾添加新的对象

2、**list.count(obj)** 统计某个元素在列表中出现的次数

3、**list.extend(seq)** 在列表末尾一次性追加另一个序列中的多个值（用新列表扩展原来的列表）

4、**list.index(obj)** 从列表中找出某个值第一个匹配项的索引位置

5、**list.insert(index, obj)** 将对象插入列表

6、**list.pop(obj=list[-1])** 移除列表中的一个元素（默认最后一个元素），并且返回该元素的值

7、**list.remove(obj)** 移除列表中某个值的第一个匹配项

8、**list.reverse()** 反向列表中元素

9、**list.sort([func])** 对原列表进行排序

线性表的应用（集合运算）

```
class Set :  #集合的类定义
    def __init__(self ) :
        self.list = [ ]
    def  IsExist(self, x ) :  #判定元素是否在集合中存在
        if  self.list.count(x) > 0 : return True
        else : return False
    def  AllElements(self) :
        return self.list
    def  Insert(self, x ) :  #向集合中添加一个元素
        if  self.IsExist( x ) == False :
            self.list.insert( 0, x )
    def  Remove(self, x) :   #从集合中删除一个元素
        self.list.remove( x )
```

集合的“并”运算

```
def  Union(S1, S2) :
    #集合S1和S2做并运算，返回并运算结果
    S = Set( )
    #复制S1中所有元素到S中
    for  x in S1.AllElements( ) : S.Insert(x)
    for  y in S2.AllElements( ) :  #对S2中所有元素
        if  S1.IsExist( y ) == False :   #如果在S1中不存在
            S.Insert( y )              #将x插入到S中
    return S
```

线性表的应用：集合的“交”运算

```
def  Intersection(S1, S2) :
    #集合S1和S2做交运算，返回交运算结果
    S = Set( )
    for  x in S1.AllElements( ) :  #对S1集合中所有元素
        if  S2.IsExist(x) == True :   #如果在S2中存在
            S.Insert(x)        #将x插入到S中
    return S
```

【例】多项式(Polynomial)的表示及其计算

线性表可以用来表示和操作符号多项式。比如：两个多项式可以表示为：

$A(x)=3x^2+2x+4$　　$B(x)=x^4+10x^3+1$

一般形式为：

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \bullet\bullet\bullet + a_n x^n$$

$$= \sum_{i=0}^{n} a_i x^i$$

其中非零项的最大指数称为多项式的阶。

假定有两个多项式：

$$A(x)=\sum a_i x^i \qquad B(x)=\sum b_i x^i$$

它们的和与乘积的数学定义为：

$$A(x)+B(x)=\sum (a_i+b_i)x^i$$

$$A(x)\bullet B(x)=\sum (a_i x^i \bullet \sum (b_i x^i))$$

$$P_n(x) = a_0 x^{e_0} + a_1 x^{e_1} + \ldots + a_m x^{e_m}$$

	0	1	2		i		m
coef	a_0	a_1	a_2	……	a_i	……	a_m
exp	e_0	e_1	e_2	……	e_i	……	e_m

注意：这时数组可以存放不只一个多项式。这样，多项式数据类型的多个对象可以共享一个数组。

$$A(x) = 2.0x^{1000} + 1.8$$

$$B(x) = 1.2 + 51.3x^{5} + 3.7x^{101}$$

	A.start ↓	*A.finish* ↓	*B.start* ↓		*B.finish* ↓	*free* ↓ *maxTerms*
coef	1.8	2.0	1.2	51.3	3.7	……
exp	0	1000	0	50	101	……

两个多项式存放在***termArray***中

多项式的相加

结果多项式另存

扫描两个相加多项式，若都未检测完：

若当前被检测项指数相等，系数相加。若未变成0，则将结果加到结果多项式。

若当前被检测项指数不等，将指数大者加到结果多项式。

若有一个多项式已检测完，将另一个多项式剩余部分复制到结果多项式。

通过演示说明算法设计思想

算法思想：

初始化；

While（A、B两个多项式都没处理完）

{ 对A、B两个多项式的当前项的指数部分进行比较

if 指数部分相同

{ **if** 系数相加的结果不为零

在C多项式中增加新的项，其指数部分与A多项式的当前项指数部分相同，系数为两多项式当前项系数相加的结果。

}

if A多项式当前项的指数大于B多项式当前项的指数

复制A多项式的当前项到C多项式中作为一个新的项，改变A当前项

if B多项式当前项的指数大于A多项式当前项的指数

复制B多项式的当前项到C多项式中作为一个新的项，改变B当前项

}

While（ A多项式未处理完）{ 顺序将A多项式的剩余项复制到C多项式中}

While（ B多项式未处理完）{ 顺序将B多项式的剩余项复制到C多项式中}

例:

$$A(x) = 2x^{1000} + 1$$

$$B(x) = x^4 + 10x^3 + 3x^2 + 1$$

	a.start	a.finish	b.start			b.finish	c.start			c.finish		free
coef	2	1	1	10	3	1	2	1	10	3	2	
exp	1000	0	4	3	2	0	1000	4	3	2	0	
	0	1	2	3	4	5	6	7	8	9	10	11

项和多项式的类定义

```
class term :
    def __init__(self, e, c): #e是指数部分，c是系数
        self.exp = e
        self.coef = c
```

```
class Polynomial:  #指数项升序排列
    def __init__(self) : #创建空表达式
        self.list = [ ]
    def  IsEmpty (self) :  #若是零多项式返回Ture，否则返回False
        pass
    def Coef (self, e) : # 返回多项式中指数为e的项的系数
        pass
    def  LeadExp(self) : #返回多项式中最大指数
        pass
    def AddTerm (self, e, c) : #将 <e, c> 加入多项式
        pass
    def  Add (self, B) : #返回多项式与B之和
        pass
    def  Mult (self, B) : #返回多项式与B之积
        pass
    def  Eval (self, f) : #计算并返回x = f 时多项式的值
        pass
    def  Print(self) :
        pass
```

```
def  Add (self, B) : #返回多项式与B之和
        C = Polynomial( )
        a, b, c = 0, 0, 0
        while  a < len(self.list) and b < len(B.list) :
            if  self.list[a].exp == B.list[b].exp :  #指数相等
                c = self.list[a].coef + B.list[b].coef
                if  c is not 0 :
                    C.AddTerm(self.list[a].exp, c)
                a += 1; b += 1
            elif  self.list[a].exp < B.list[b].exp :
                C.AddTerm(self.list[a].exp, self.list[a].coef)
                a += 1;
            else :
                C.AddTerm(B.list[b].exp, B.list[b].coef)
                b += 1
```

```
while a < len(self.list) : #a未检测完时
    C.AddTerm(self.list[a].exp, self.list[a].coef)
    a += 1
while b < len(B.list) : #b未检测完时
    C.AddTerm(B.list[b].exp, B[b].coef)
    b += 1
return C
```

分析：
在最坏情况下，A和B中的各个指数部分均不相同，每次循环中仅改变一个多项式的当前项，循环的次数为最多，所以，循环的最坏情况时间复杂度为O(n+m)，其中n和m分别为A和B中非0项的个数。

两个多项式的相乘如何实现？

1、将A多项式与B多项式中的每一项相乘；

2、将步骤1中的结果逐个相加。

用数组表示多项式的缺点

当我们创建一个新的多项式的时候，**free**指针在没有达到**MaxTerm**时是连续增加的。如果**free**达到**MaxTerm**怎么办？或者是宣布创建失败，或者是检查数组中有无其它不再使用的多项式，将其删除以释放出一片连续的空间。在这个过程中，可能还需要对某些多项式进行移动。这些工作，都需要耗费额外的系统资源。

要从根本上解决这样的问题，须要采用链接式的线性表。

顺序存储结构的特点：

- 优点：结构简单、易于实现、随机存取、存储密度高。
- 缺点：插入、删除元素代价大，需要事先确定存储空间（或改变存储空间的大小代价大）。
- 适用场合：插入、删除操作少，数据元素数量相对固定。

线性表的存储结构—链接存储

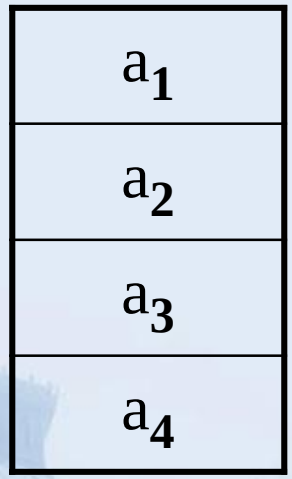
- 动态链表
- 静态链表。

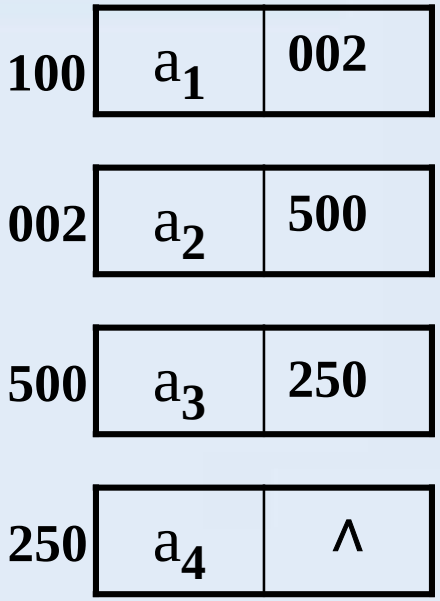
特点：使用一组任意存储单元存储数据元素，逻辑顺序和物理顺序可以相同，也可以不同。

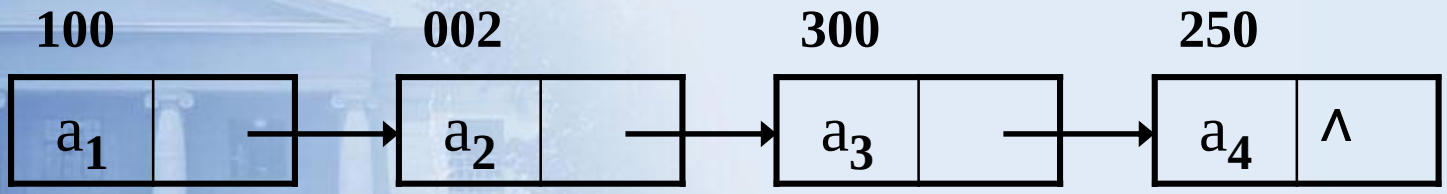
优点：

- 插入、删除方便，
- 共享空间好。

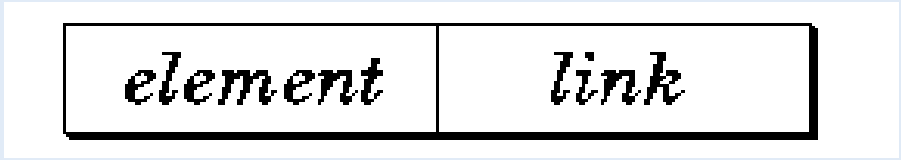
例如：线性表(a_1, a_2, a_3, a_4)



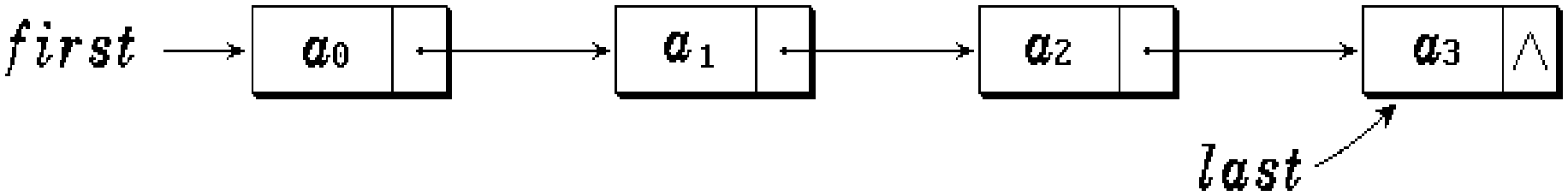




单链表：每个结点只含有一个链接域。
特点：每个元素(表项)由结点(*Node*)构成（数据域、链接域）。

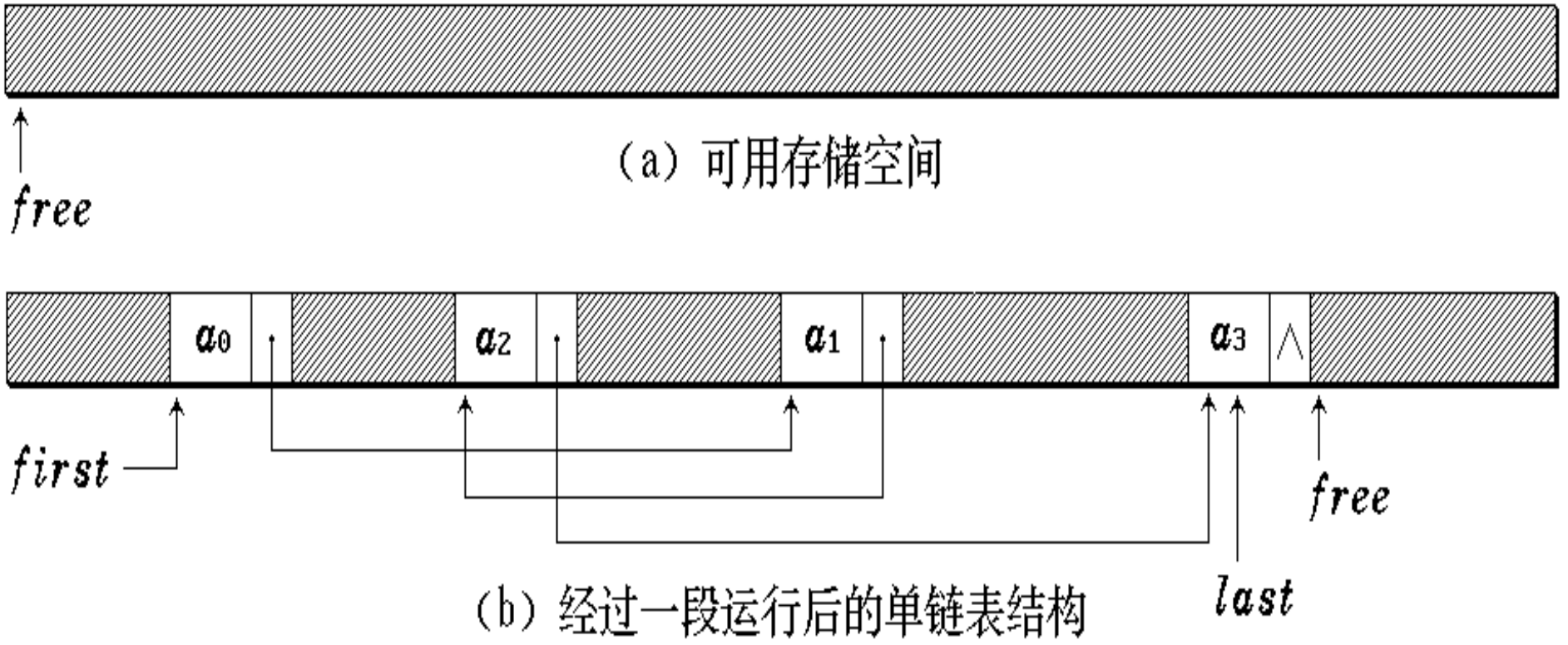


通过链接域反映数据元素之间的逻辑关系。
线性结构:



结点可以非顺序存储，数据元素顺序存取。
使用头指针和尾指针，表可扩充

单链表的存储映像



单链表结点的类定义

```
class Node:
    def __init__(self, data):
        self.data = data
        self.link = None
```

单链表的创建（在链表头部插入新结点）

```
def  CreateList(A):
        n = len(A)
        first = None
        for i in range(n-1, -1, -1 ) :
            t = Node(A[i])
            t.link = first
            first = t
        return first
```

单链表的输出

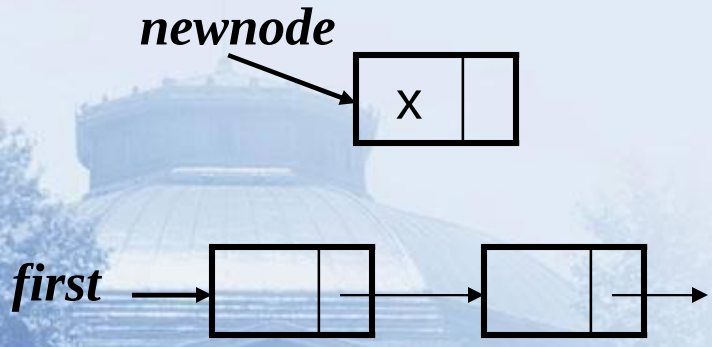
```
def  PrintList(L) :
    p = L
    while not p == None :
        print p.data,
        p = p.link
    print
```

单链表中的插入

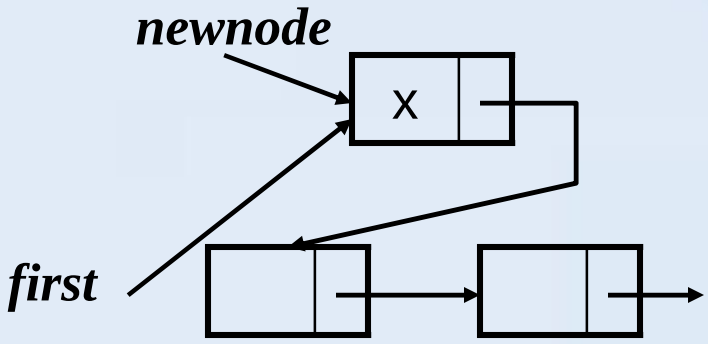
第一种情况：在第一个结点前插入

newnode.link = first ;

first = newnode;



（插入前）



（插入后）

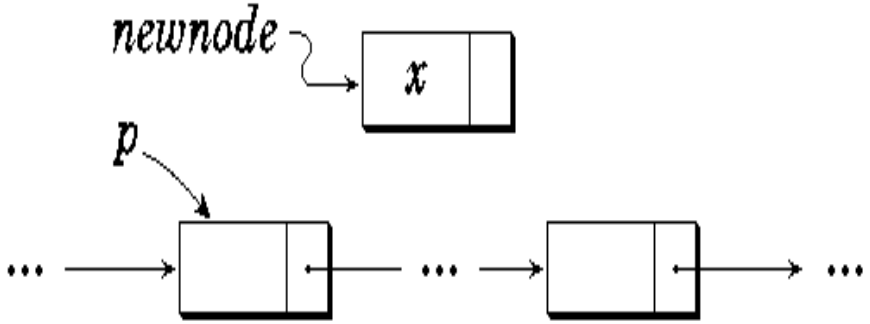
空表的条件：first==None

判断表尾的条件：P.link==None

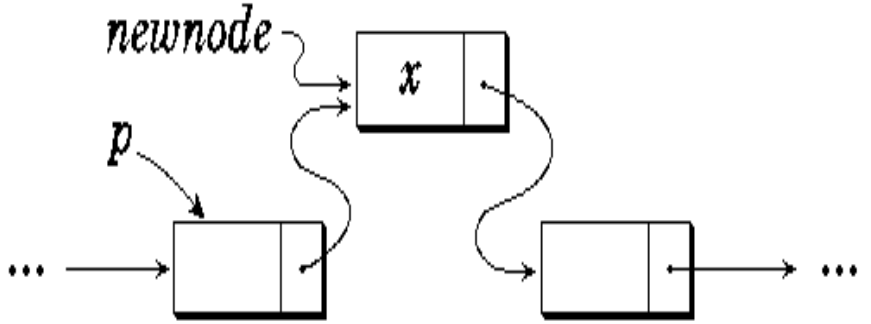
第二种情况：在链表中间插入

newnode.link = p.link;

p.link = newnode;



(插入前)

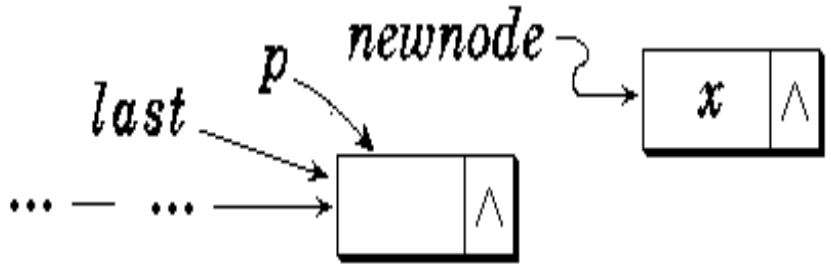


(插入后)

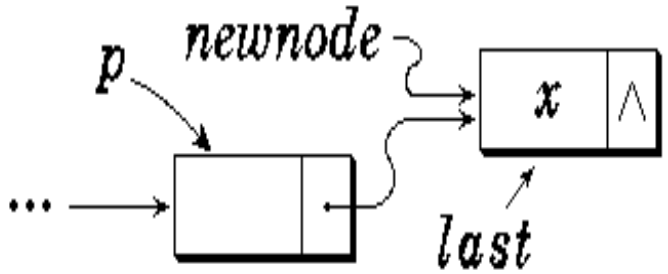
第三种情况：在链表末尾插入

newnode.link = p.link;

p.link = last = newnode;



（插入前）

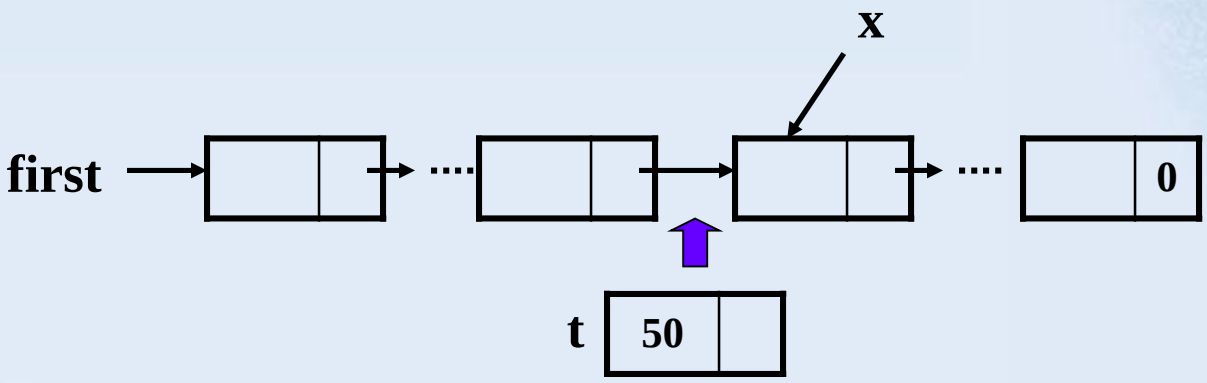


（插入后）

插入成功，返回1；

插入失败，返回0。

问题：如何在p指向的结点前插入结点？




```
temp = t.data;
t.data = x.data;
x.data = temp;
t.link = x.link;
x.link = t;
```

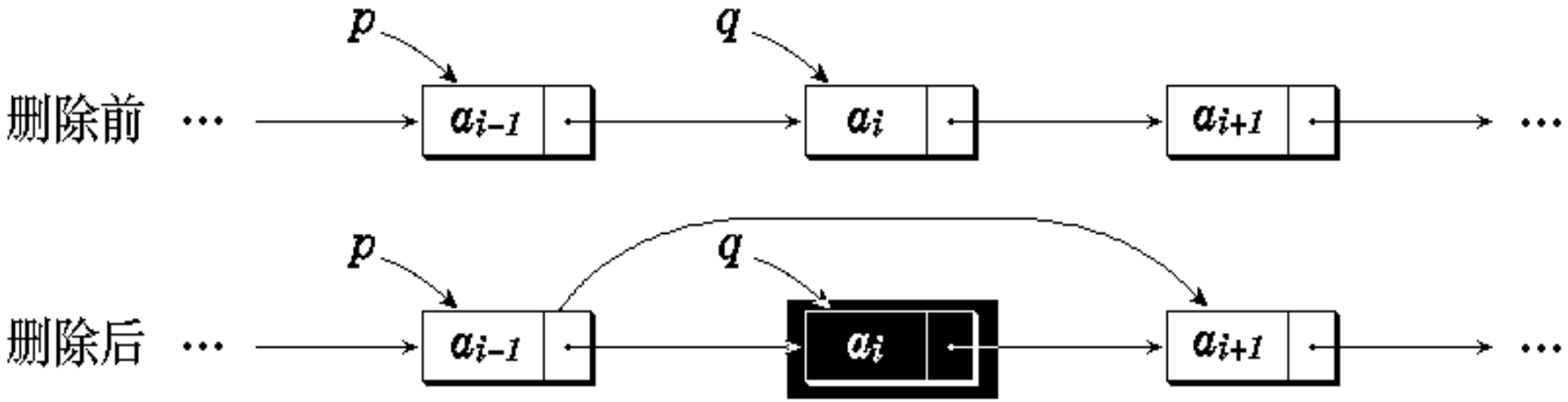
单链表中的删除

第一种情况：删除表中第一个元素

q = first; first = first.link

第二种情况：删除表中或表尾元素

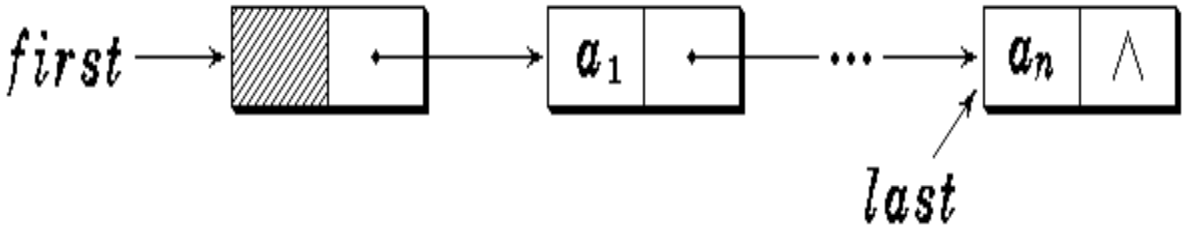
q = p.link; p.link = q.link;



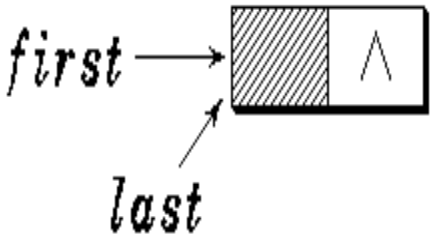
在单链表中删除含a_i的结点

带附加头结点的单链表

表头结点位于表的最前端，本身不带数据，仅标志表头。设置表头结点的目的是统一空表与非空表的操作，简化链表操作的实现。



非空表

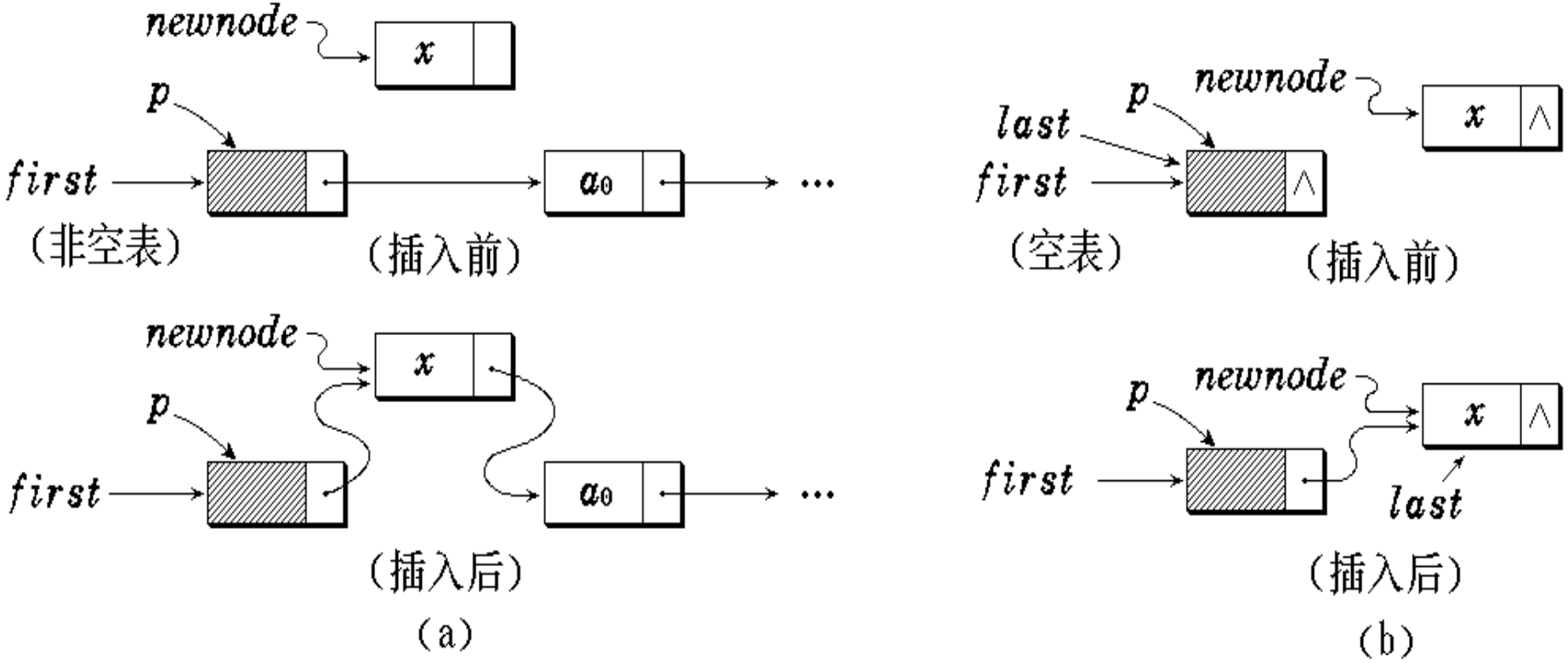


空表

空表的条件：**first.link==None**

判断表尾的条件：**P.link==None**

在带表头结点的单链表中
第一个结点前插入新结点

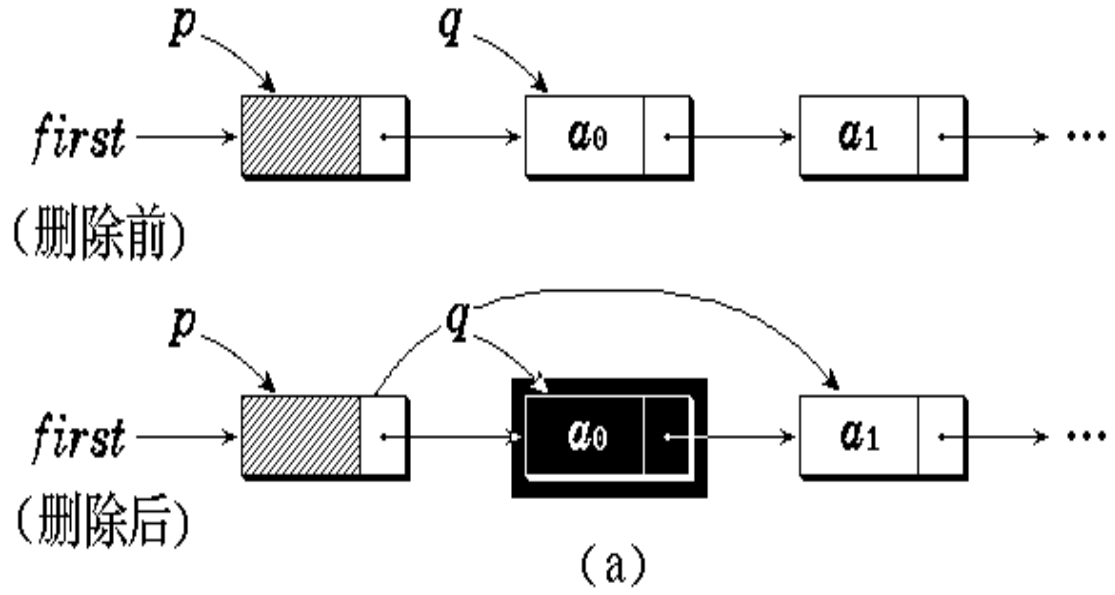


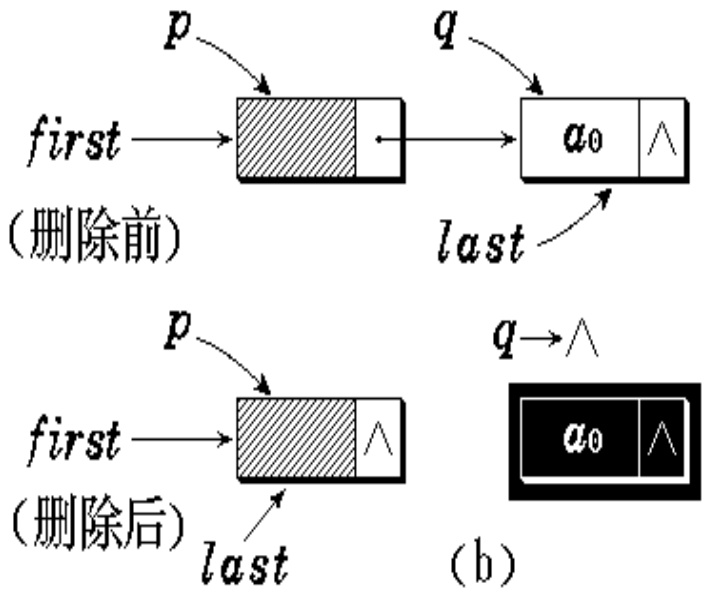
newnode.link = p.link;

if (*p.link ==None*) : *last = newnode;*

p.link = newnode;

从带表头结点的单链表中删除第一个结点






```
q = p.link;
p.link = q.link;
if ( p.link == NULL ) : last = p;
```

建立一个单链表

- 前（头）插法
- 后（尾）插法

前插法

```
……
first = None
……
val = input( “请输入")   #  将输入赋给变量val
while  val !=endTag :
    newNode = Node(val)  # 创建新结点
    newNode.link = first.link
    firs.link = newNode
    val = input( “请输入")
```

后插法

```
……
first = None
……
val = input( “请输入")   #  将输入赋给变量val
last = first
while  val !=endTag  :
    newNode = None(val)  # 创建新结点
    last.link = newNode
    last = newNode
    val = input( “请输入")
```

【例】两个单链表的连接

假设不存在数据成员last，Cancatenate将两个链表连接为一个，其计算时间与第一个链表的长度成线性关系

```
def  Concatenate(a, b) :  # 两个单链表头尾相连
    if  a == None :
        a = b
    else :
        p = a
        while p.link != None : #找到a的最后一个结点
            p = p.link
        p.link = b
```

【例】试编写在带头结点的单链表中删除（一个）最小值结点的算法。

```
def  delete(L) :
    # 单链表中删除最小值结点（不是最后一个结点）
    p = L   # p为工作指针,指向待处理的结点。
    q = p   # q 指向最小值结点，初始为第一个结点。
    while  p != None :
        if  p.data < q.data  :  q = p  # 记录最小值结点
        p = p.link     # 指针后移。
    q.data = q.link.data   # q结点与其后一结点交换值
    q.link = q.link.link    # 从链表上删除最小值结点
    return L
```

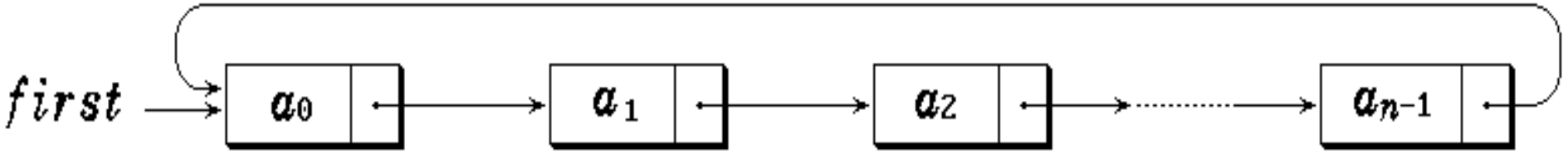
```
def  delete1(L) :  # 单链表中删除最小值结点
    p = L   # p为工作指针,指向待处理的结点。
    q = p   # q 指向最小值结点，初始为第一个结点。
    while  p != None :
        if  p.data < q.data  :  q = p  # 记录最小值结点
        p = p.link     # 指针后移。
    if  L== q : return L.link   # 第一个结点是最小值结点
    else :
        p, r = L, None
        while  p != q :   # r 定位到被删除结点的前一个结点
            r = p; p = p.link
        r.link = q.link    # 从链表上删除最小值结点
    return L
```

链表的其他变形

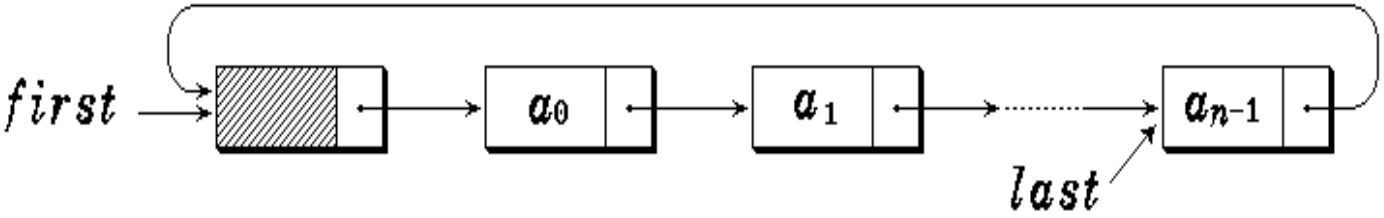
循环链表（*Circular List*）

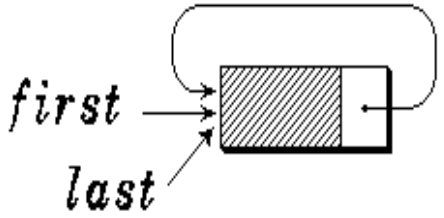
- 循环链表是单链表的变形。
- 循环链表最后一个结点的***link***指针不为(***NULL***)，而是指向了表的前端。
- 为简化操作，在循环链表中往往加入表头结点。
- 循环链表的特点是：只要知道表中某一结点的地址，就可搜寻到所有其他结点

循环链表的示例



带表头结点的循环链表



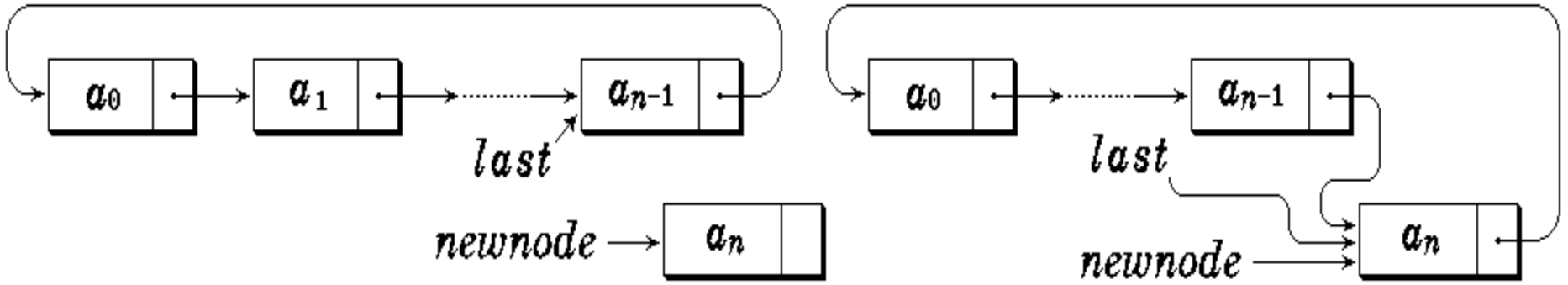


空表的条件：first.link==first

判断表尾的条件：current.link==first

在表头和表尾插入新结点非常方便

在表尾插入新结点



Newnode.**link = last.link**

Last.link = newnode

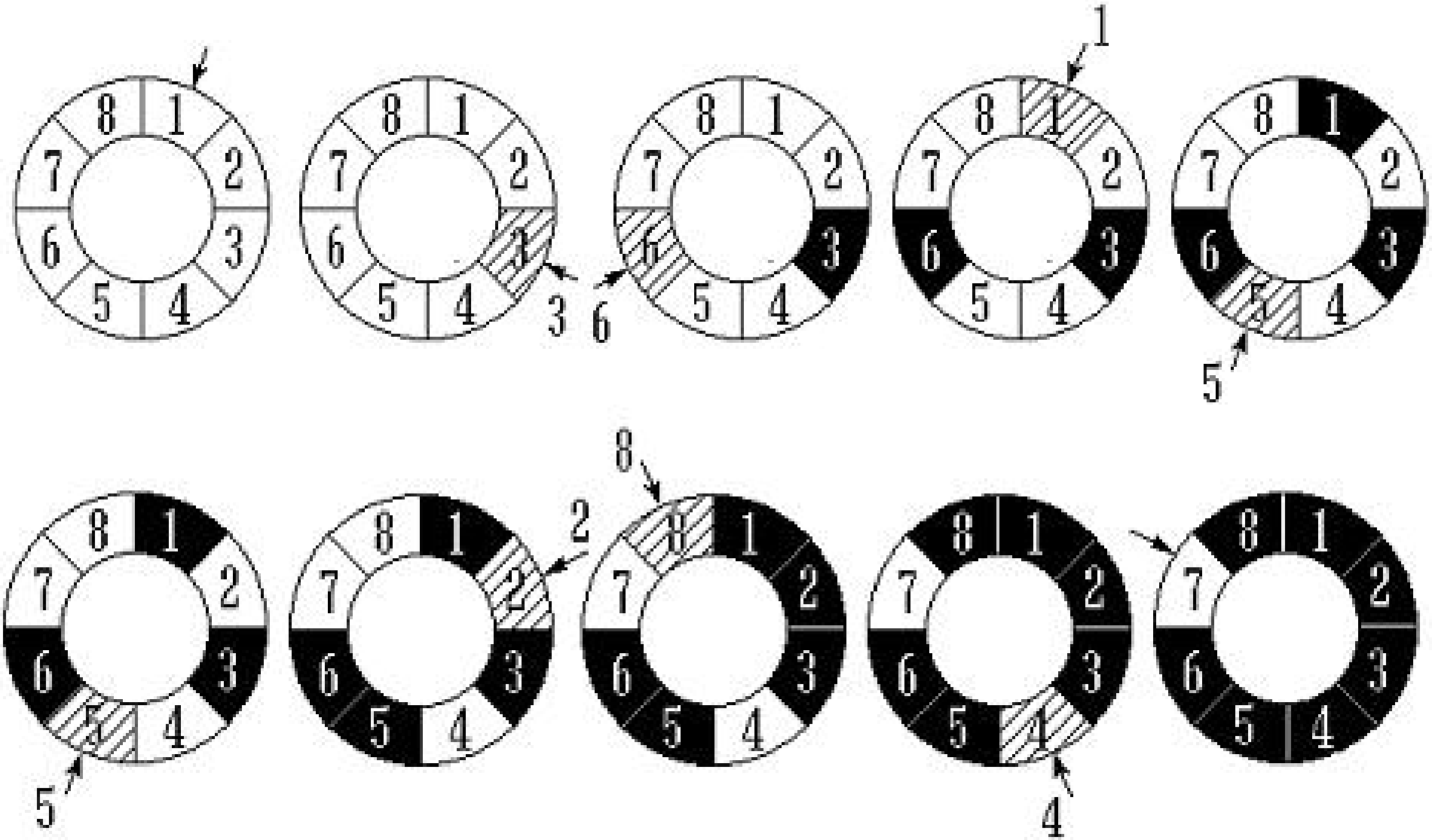
last = newnode

【例】用循环链表求解约瑟夫问题

约瑟夫问题的提法

n个人围成一个圆圈，首先第1个人从1开始一个人一个人顺时针报数， 报到第m个人，令其出列。然后再从下一个人开始，从1顺时针报数，报到第m个人，再令其出列，...，如此下去，直到圆圈中只剩一个人为止。此人即为优胜者。

例如　n = 8　m = 3



约瑟夫问题的解法

```
def  Josephus (CircList, n, m) :
    p, pre = CircList, None
    for i in range(n-1) : #执行n-1次
        for j in range(m) :
            pre = p
            p = p.link
        print "Delete ", p.data
        pre.link = p.link
    print "The Last Node is : ", pre.data
```

多项式及其相加

在多项式的链表表示中每个结点增加了一个数据成员 *link*，作为链接指针。

$data \equiv Term$

coef	*exp*	*link*

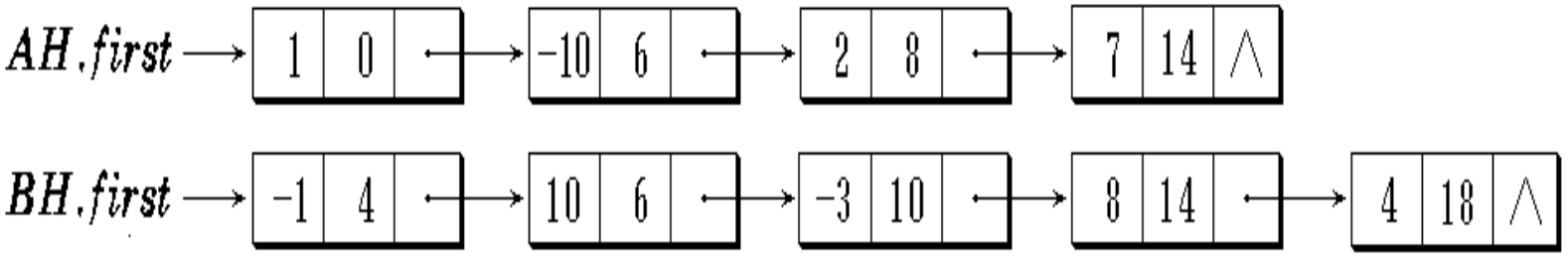
优点：

- 多项式的项数可以动态地增长，不存在存储溢出问题。
- 插入、删除方便，不移动元素。

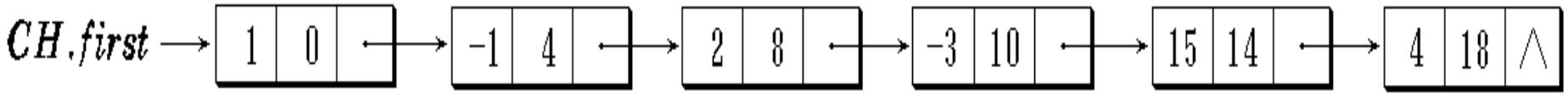
多项式链表的相加

$$AH = 1 - 10x^6 + 2x^8 + 7x^{14}$$

$$BH = -x^4 + 10x^6 - 3x^{10} + 8x^{14} + 4x^{18}$$



(a) 两个相加的多项式



(b) 相加结果的多项式

算法

初始化；

While（两个链都没处理完）

　　{ **if** （指针指向当前节点的指数项相同）

　　{系数相加，在C链中添加新的节点；

　　A、B链的指针均前移；}

　　else

　　{以指数小的项的系数添入C链中的新节点；

　　指数小的相应链指针前移；}

　　}

While（A链处理完）

　　{ 顺序处理B链；}

While（B链处理完）

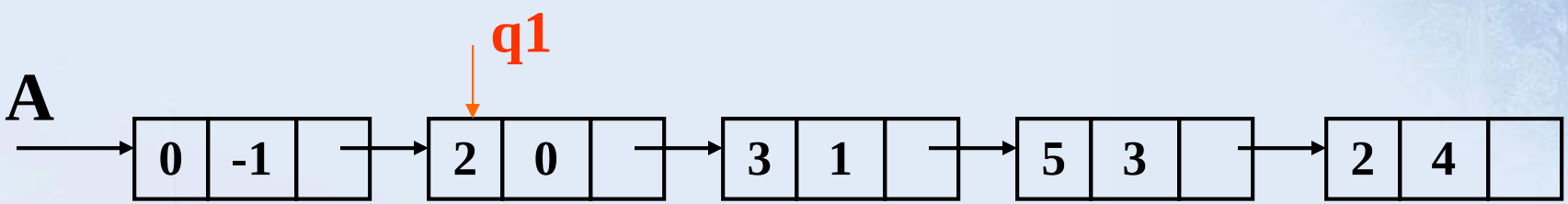
　　{ 顺序处理A链；}

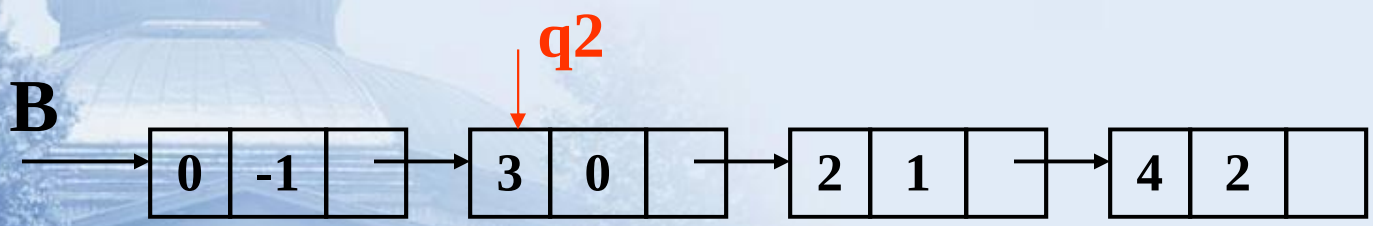
多项式

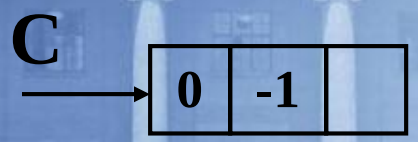
$2 + 3X + 5X^3 + 2X^4$

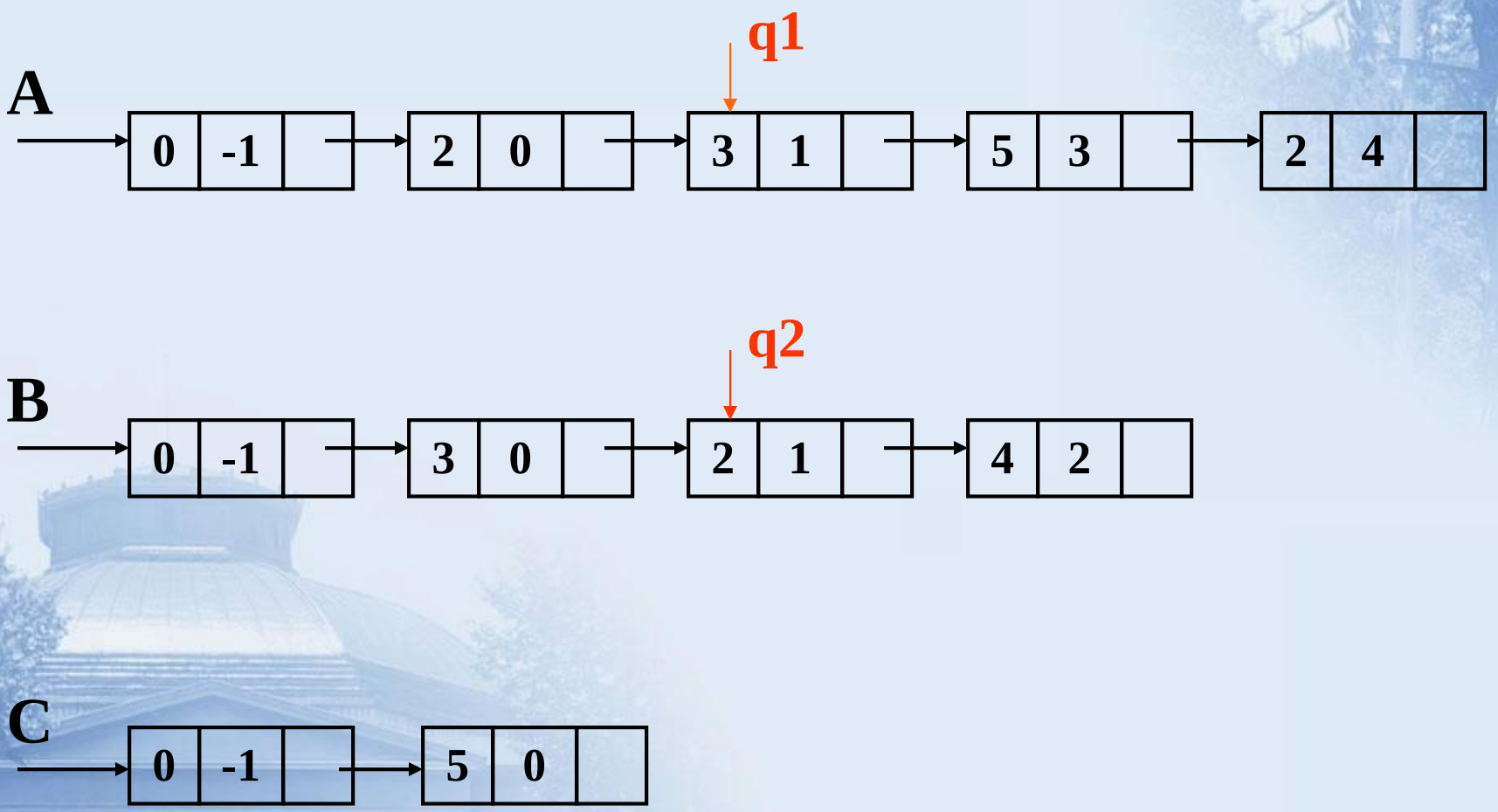
$3 + 2X + 4X^2$

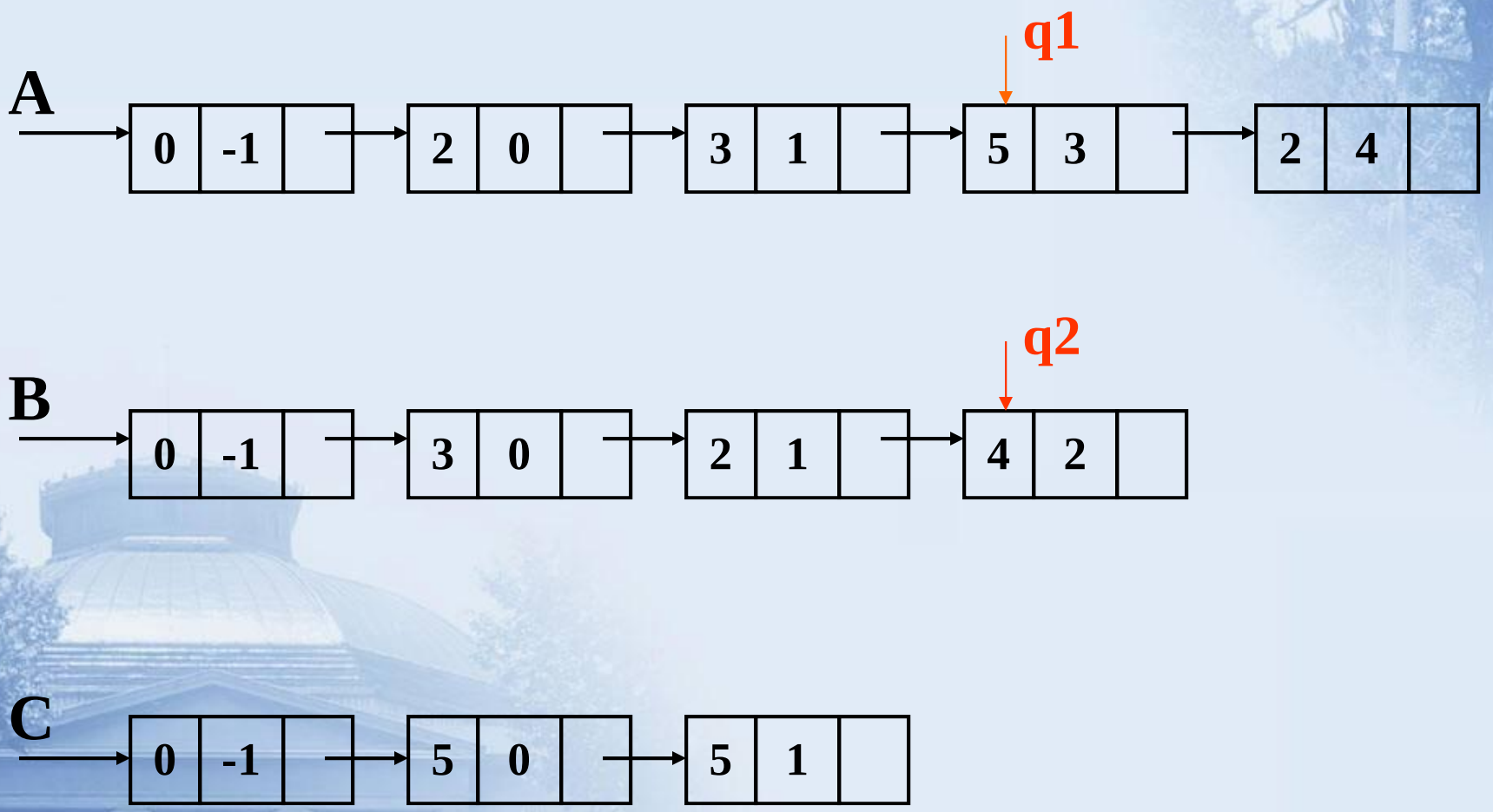
相加，结果为：$5 + 5X + 4X^2 + 5X^3 + 2X^4$

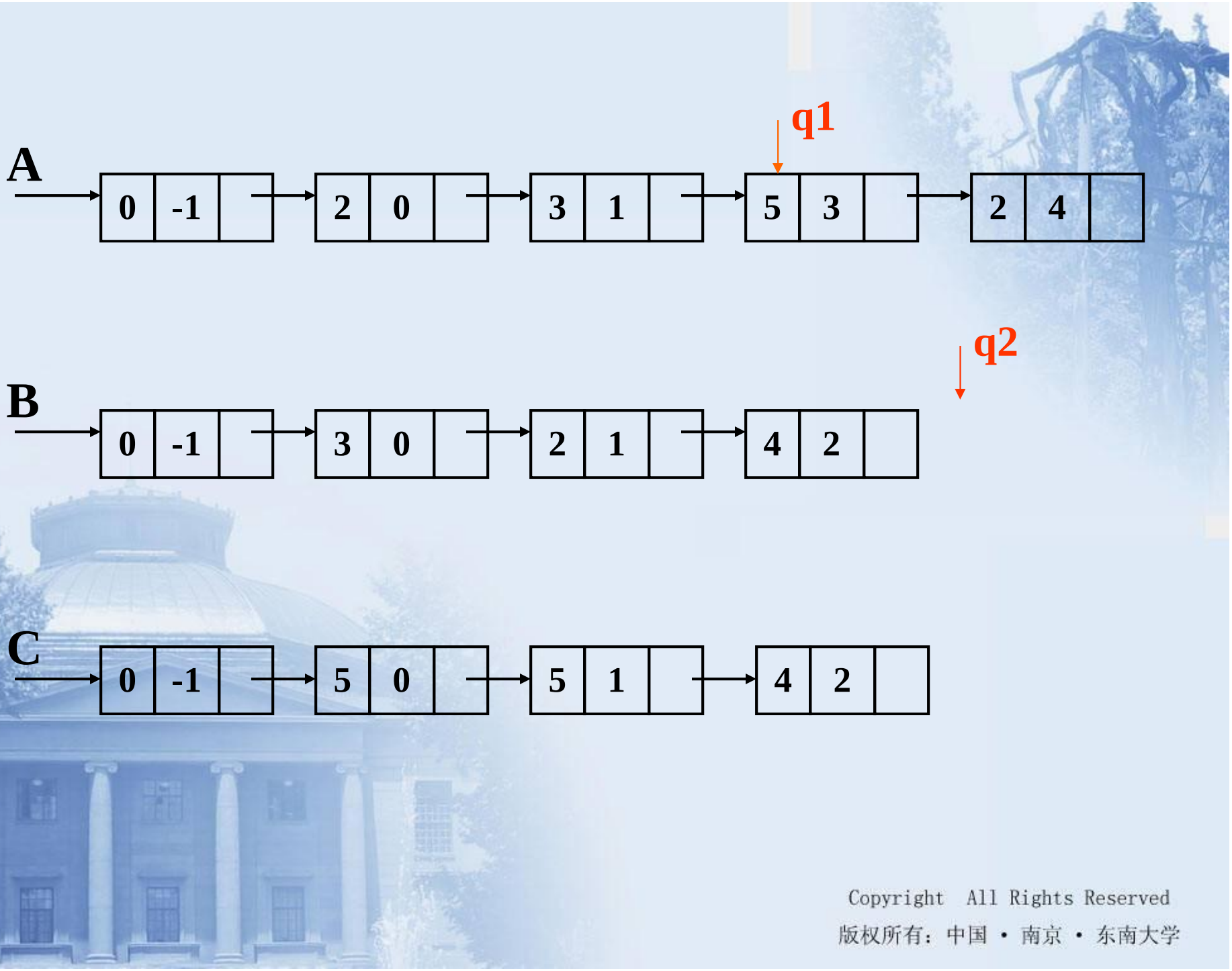




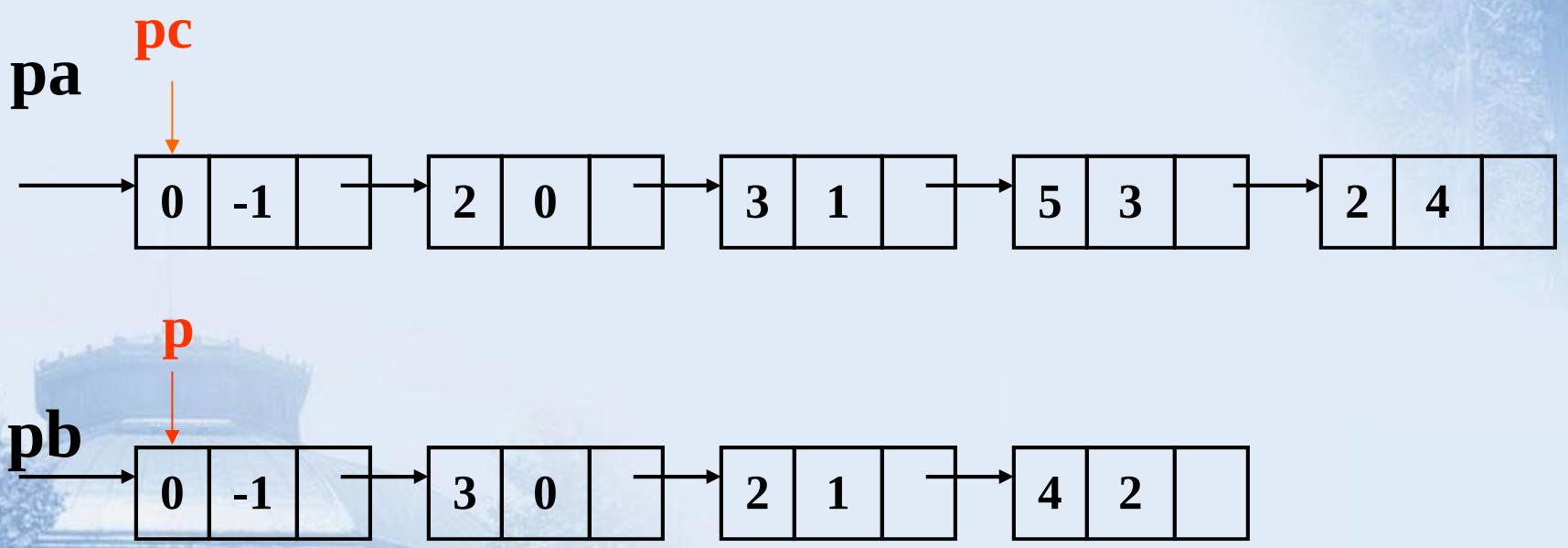


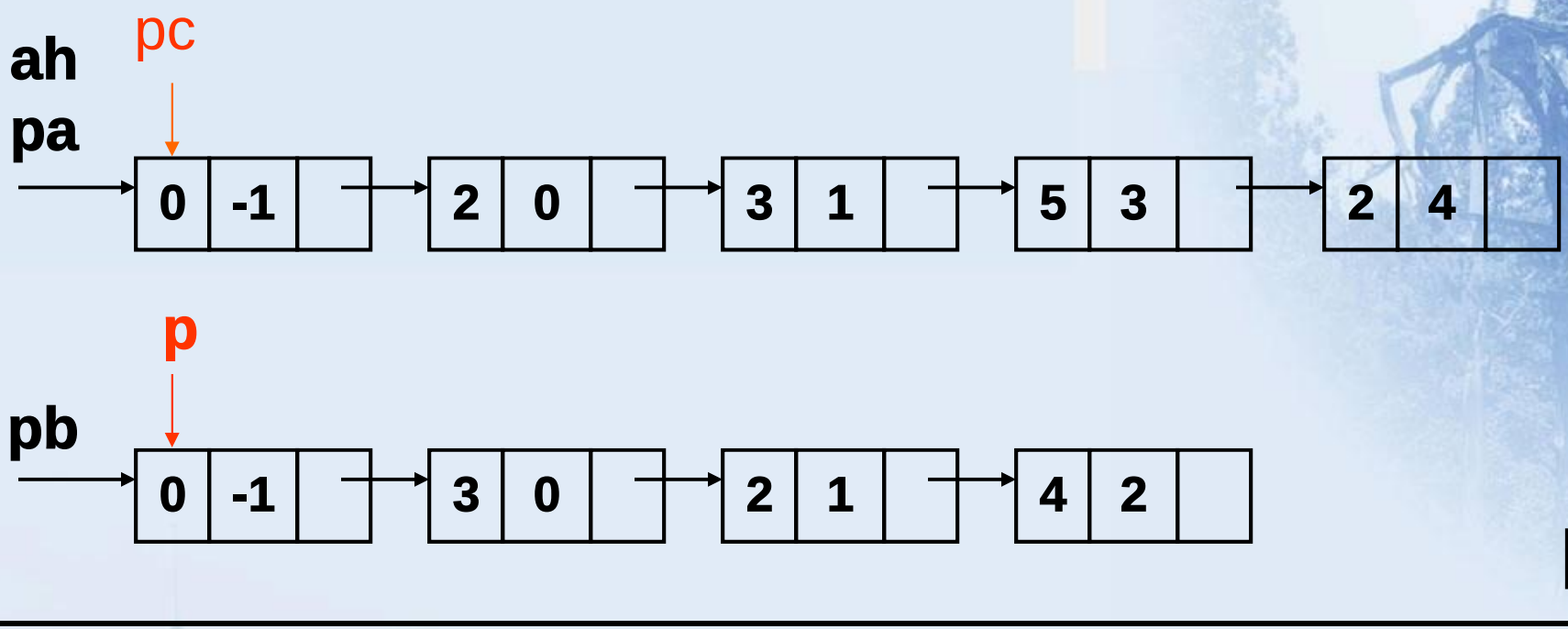




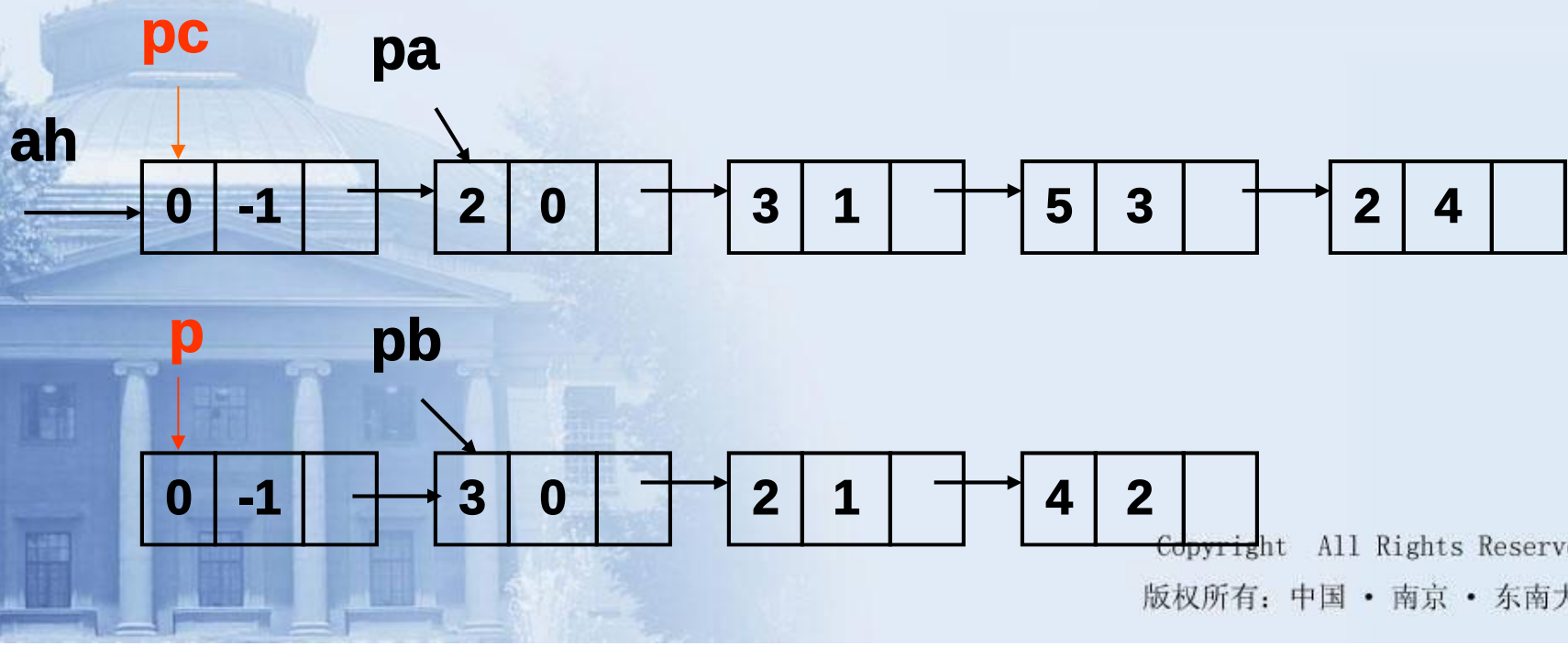


（若不附加任何新的存储空间）：

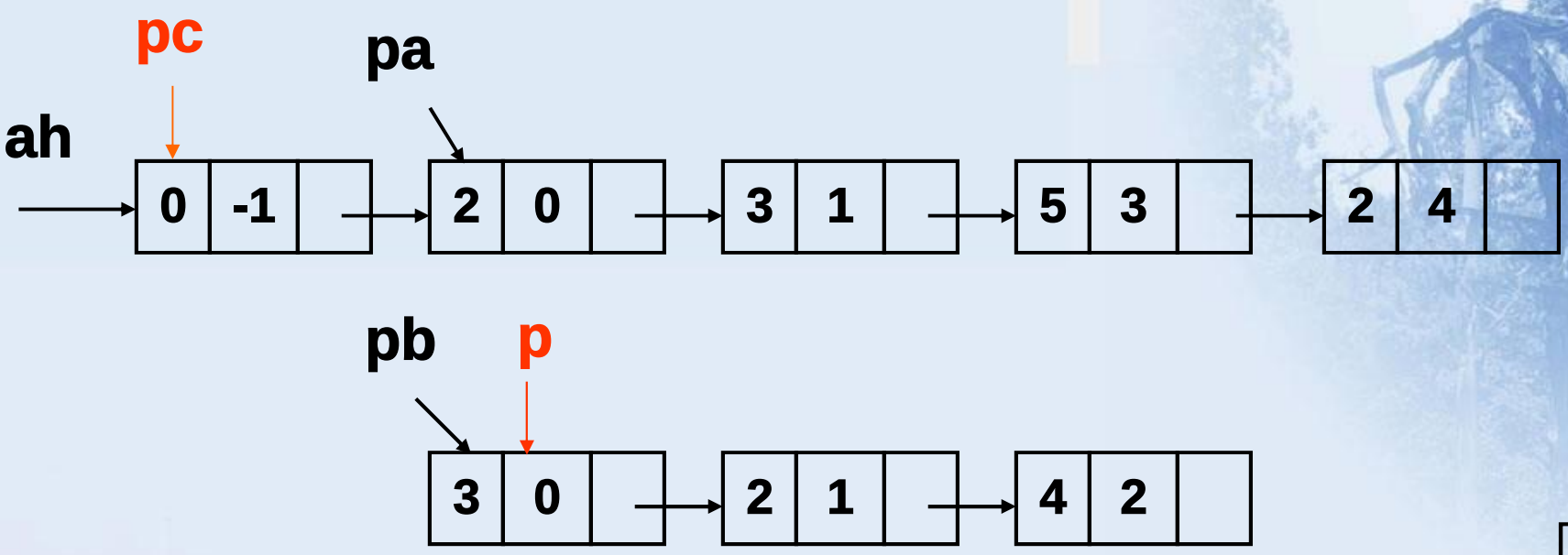




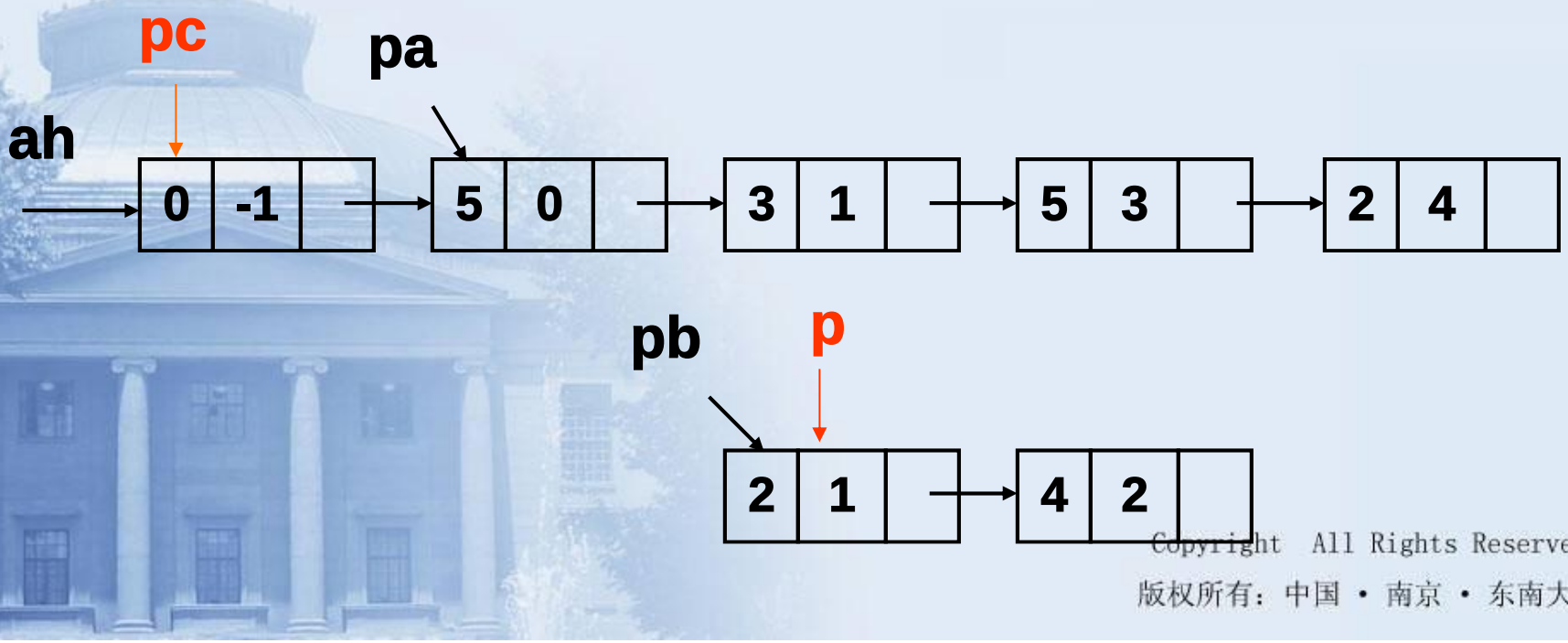
[1]



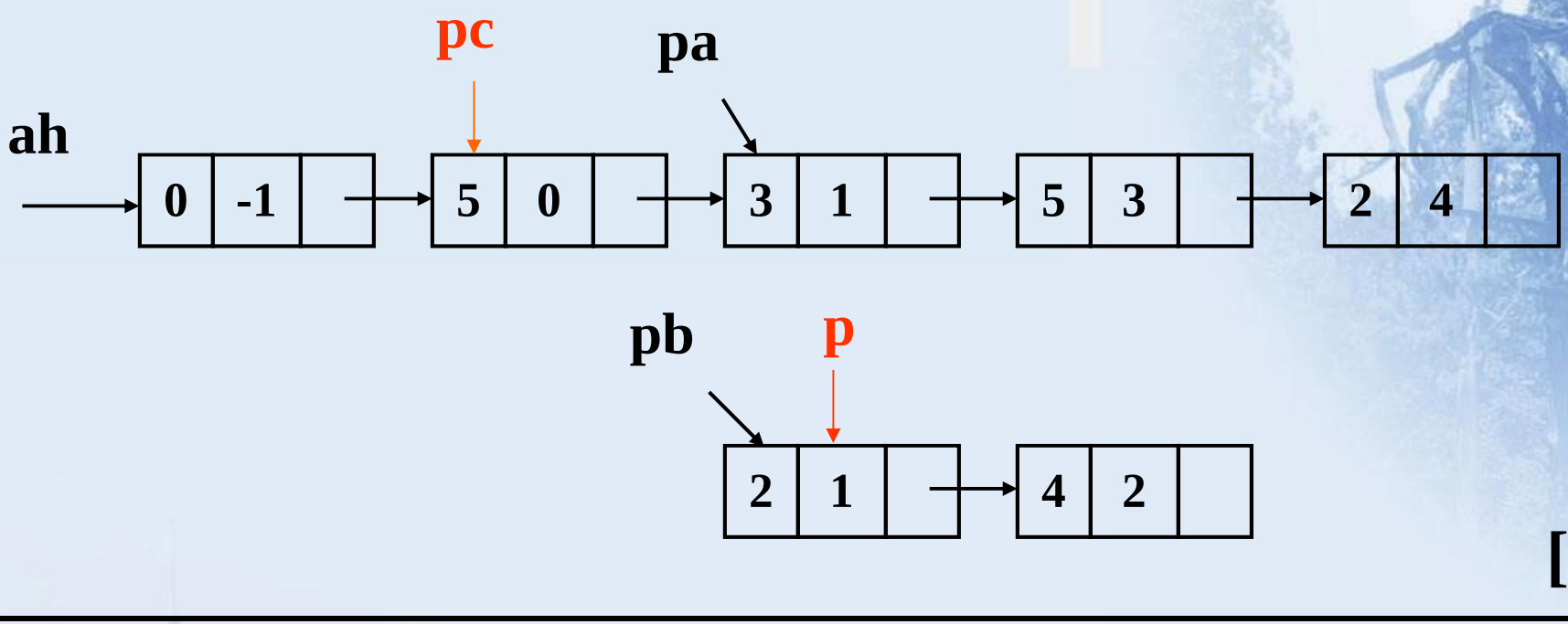
[2]



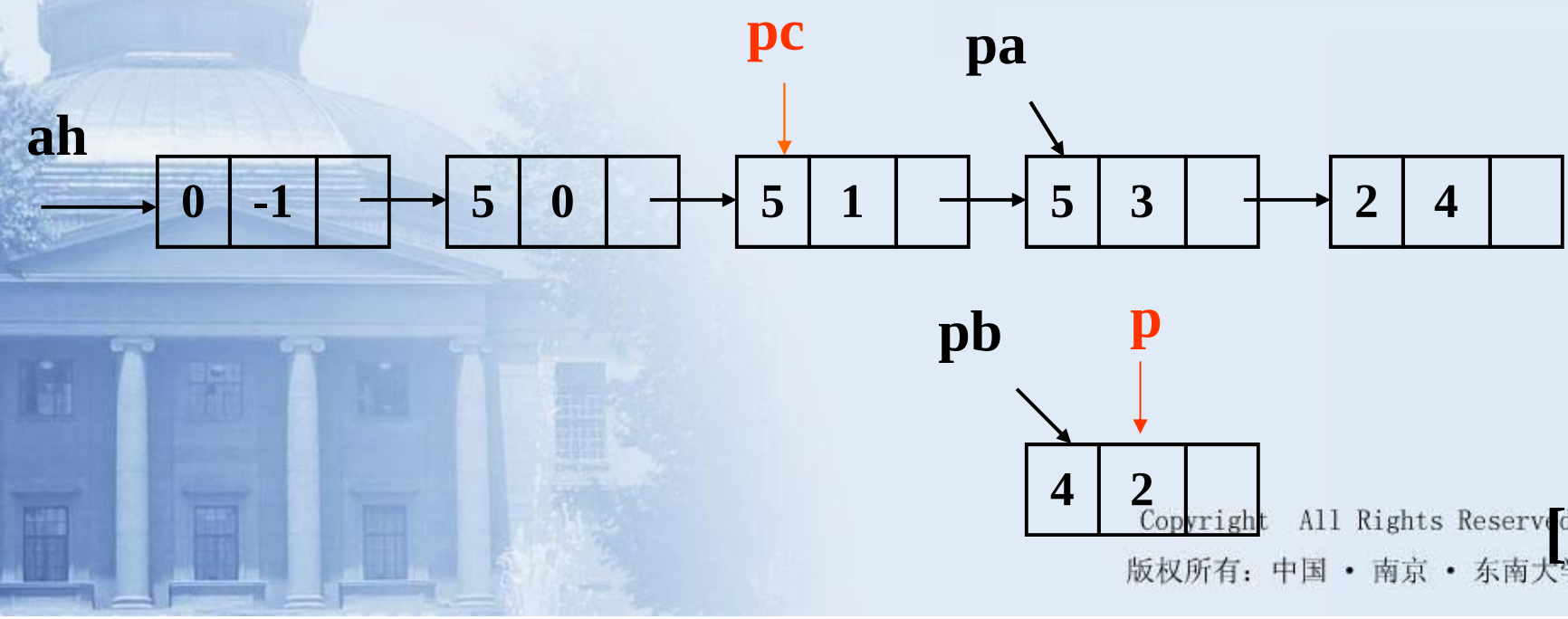
[3]



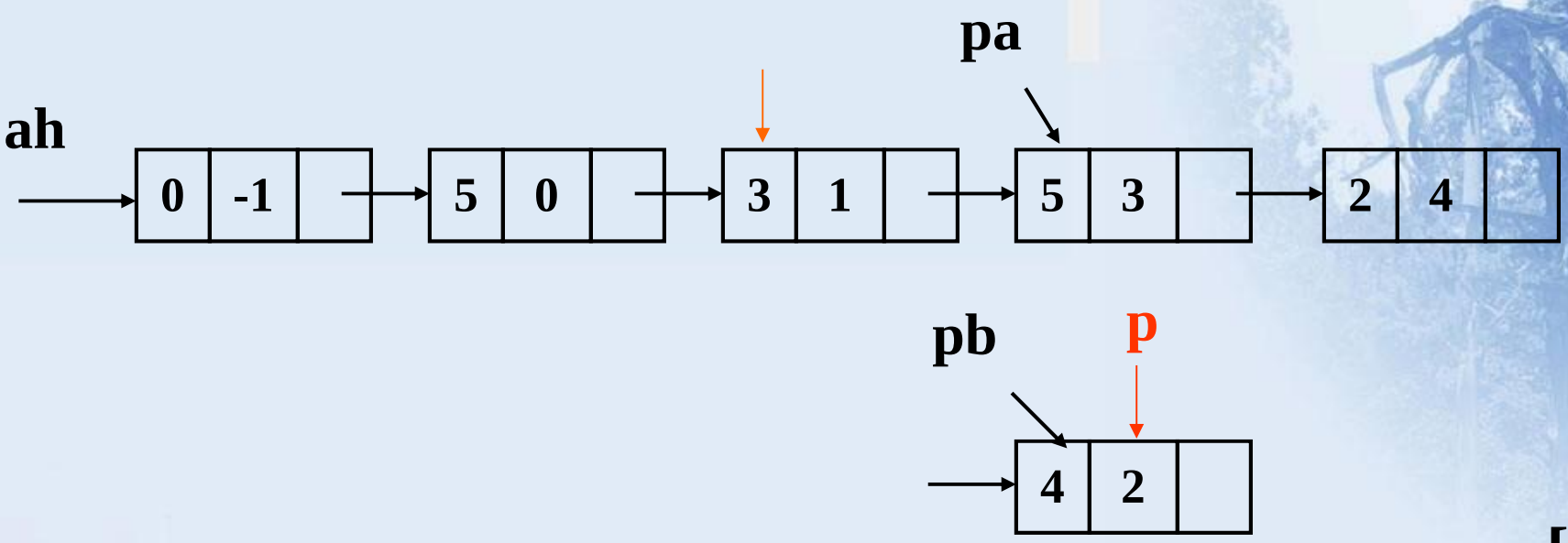
[4]



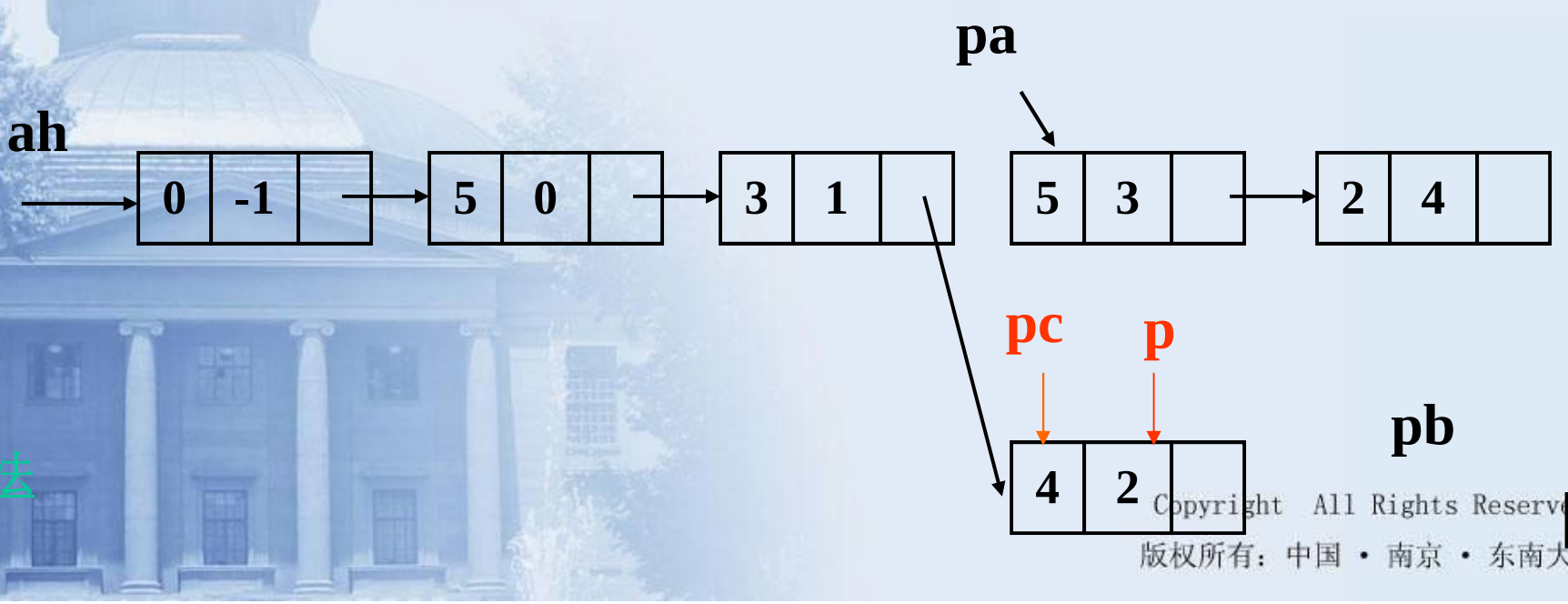
[5]



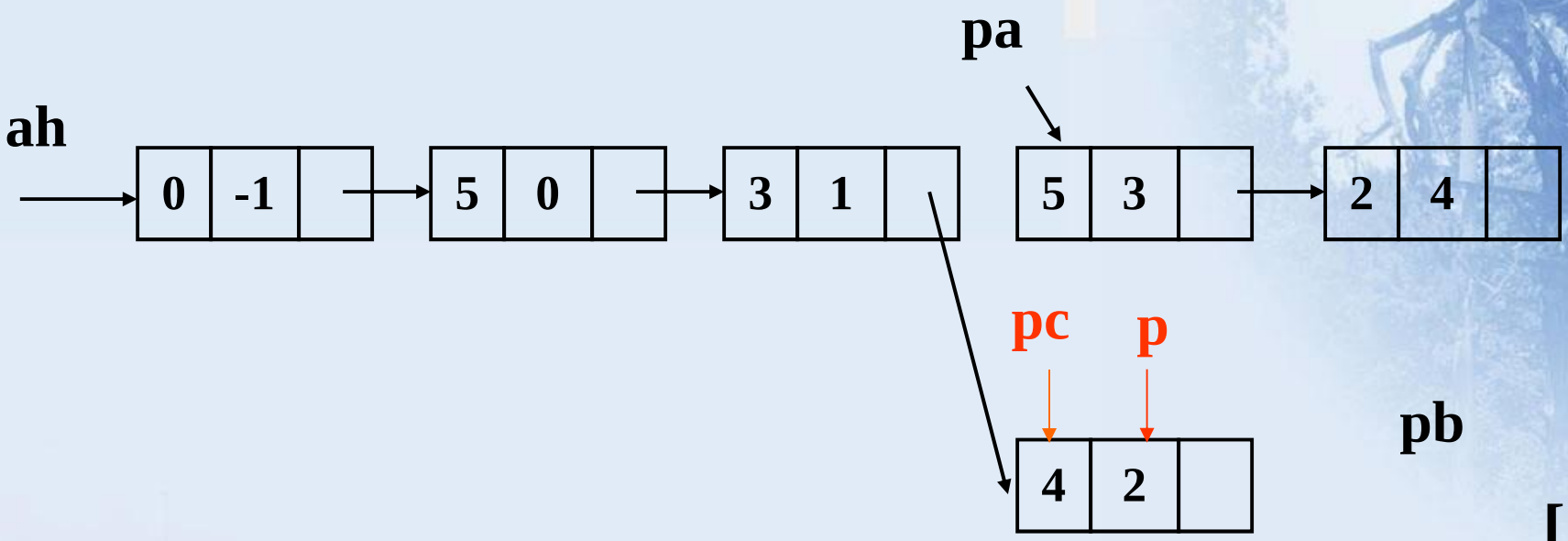
[6]



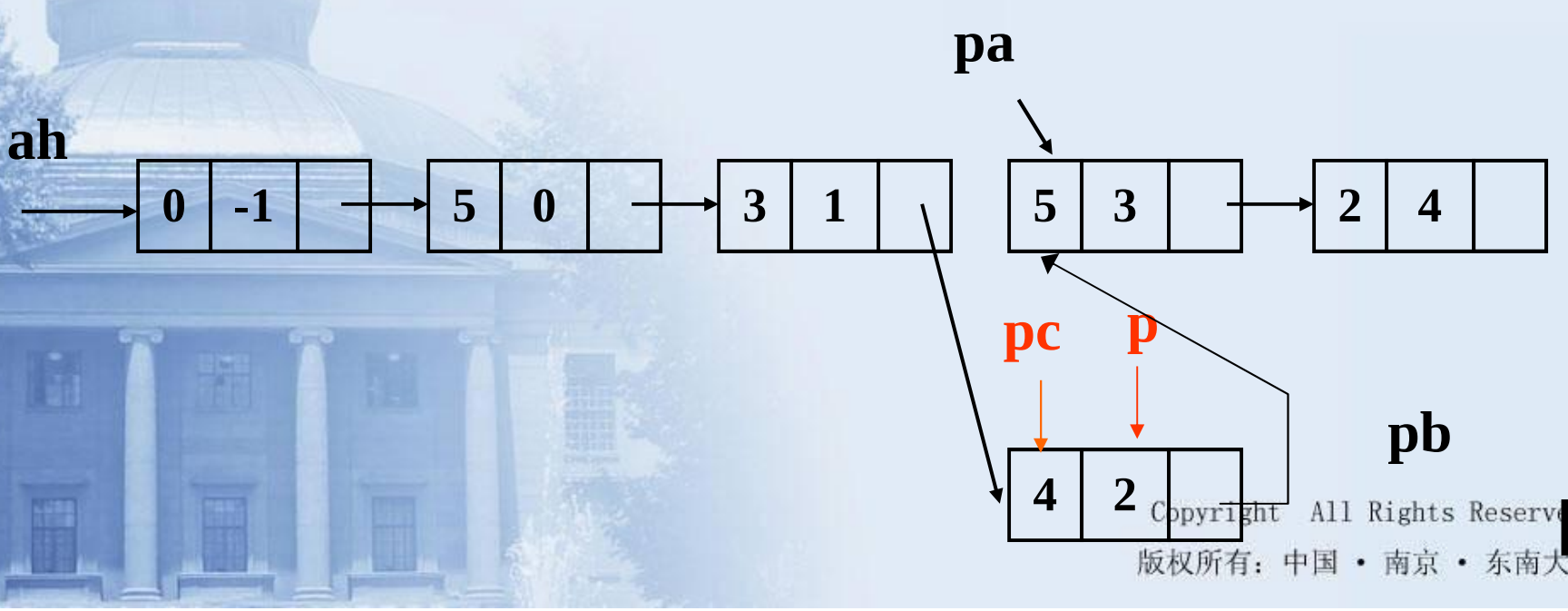
[6]



[7]



[7]



[8]

分析：

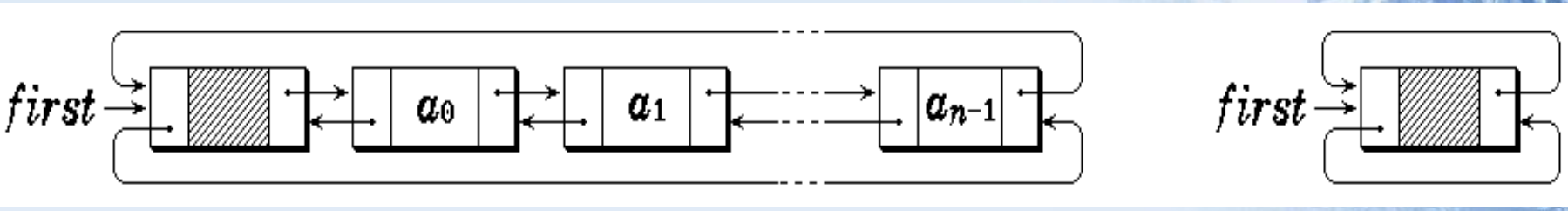
设A有m项， B有n项。第一个while循环中中，游标Aiter或Biter的当前结点指针至少有一个沿链表移动到下一个结点，此循环迭代次数最多是m + n – 1。第二和第三个while循环的总次数不超过m + n。整个算法的时间复杂性是O(m + n)。

双向链表 (*Doubly Linked List*)

- 双向链表是指在前驱和后继方向都能游历(遍历)的线性链表。
- 双向链表每个结点结构

前驱方向 ← → 后继方向

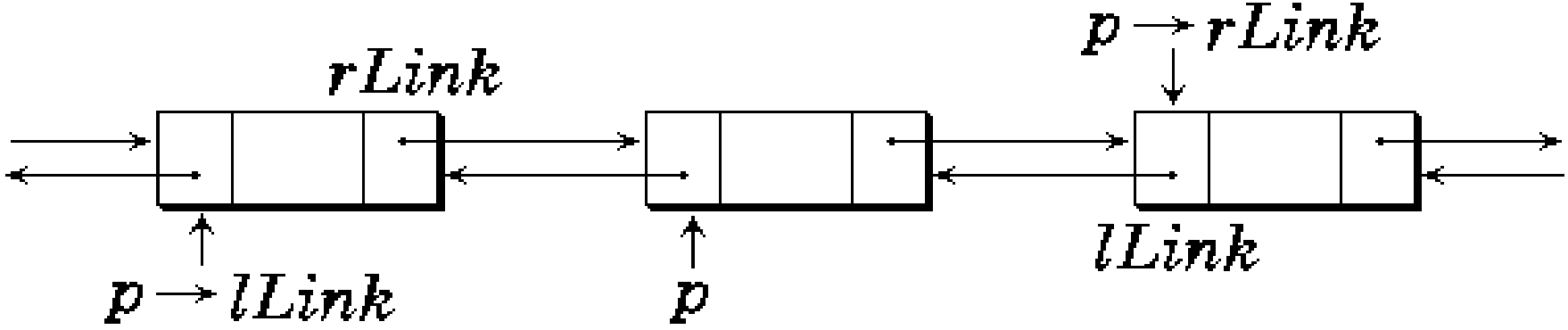
- 双向链表通常采用带表头结点的循环链表形式。



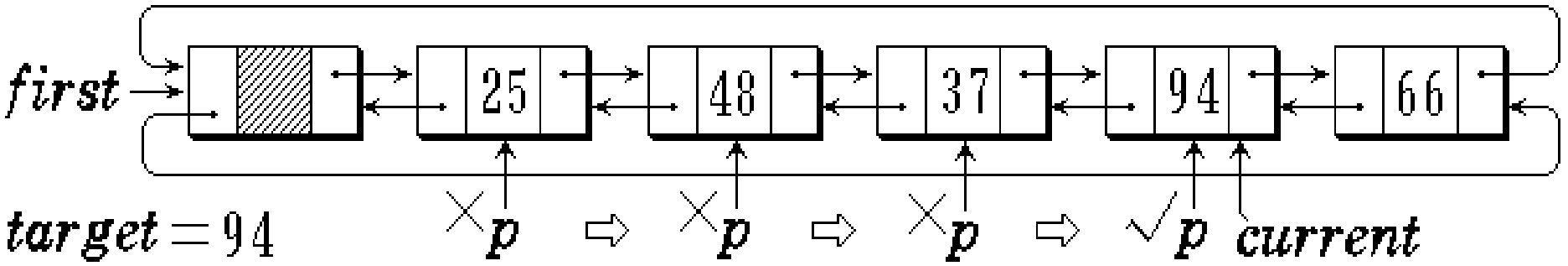
非空表　　空表

结点指向

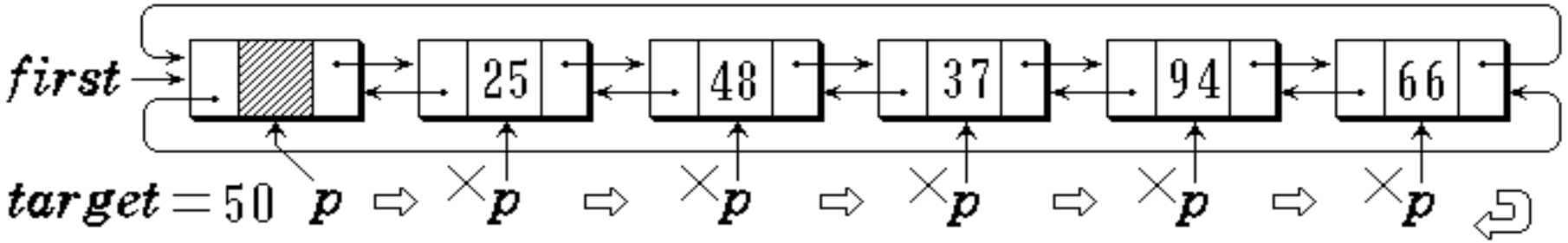
$p == p\rightarrow lLink\rightarrow rLink == p\rightarrow rLink\rightarrow lLink$



双向循环链表的搜索



搜索成功

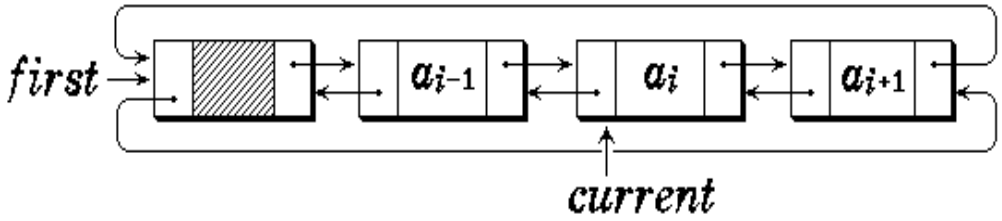


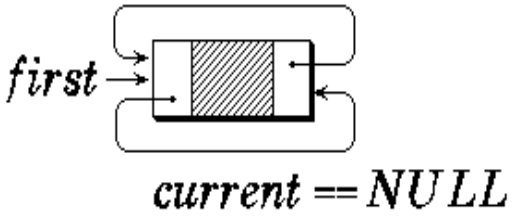
搜索不成功

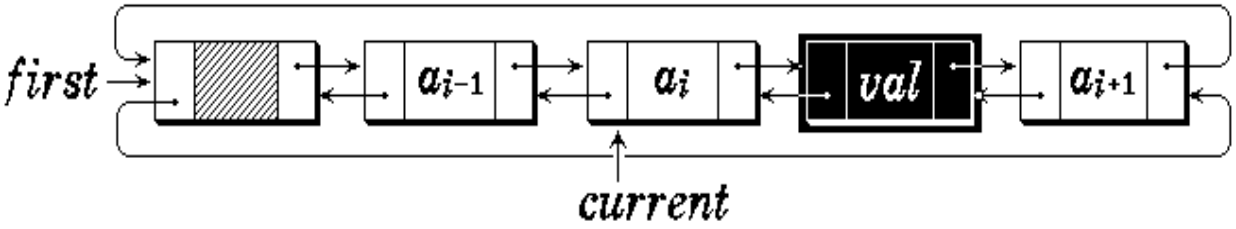
双向循环链表的插入算法(p指向新结点)

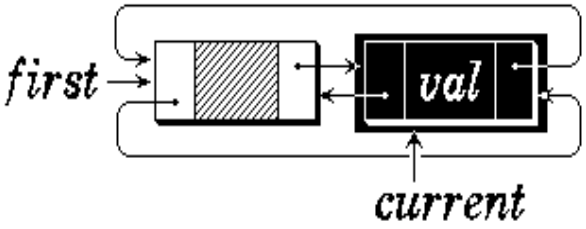
p.lLink = current
p.rLink =current→rLink
Current.rLink = p
current = current.rLink
Current.rLink.lLink = current

结果, **current**指向新结点



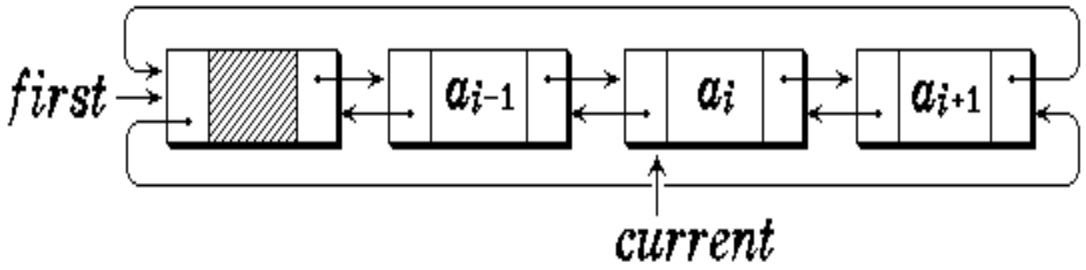


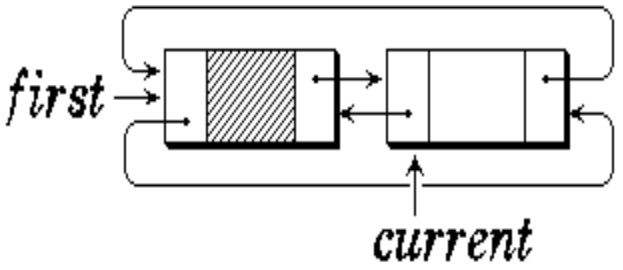


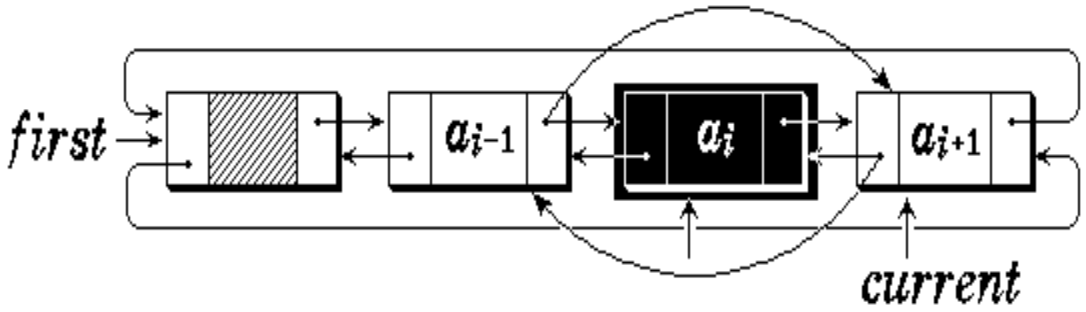


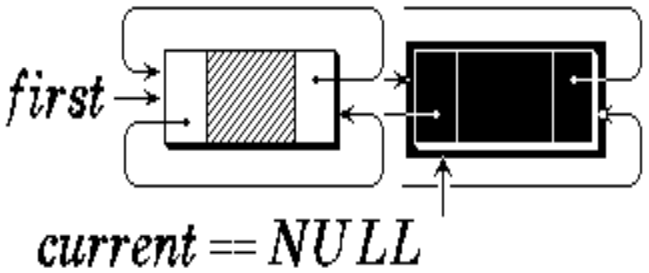
双向循环链表的结点删除

Current.rLink.lLink = current.lLink
current.lLink.rLink = current.rLink



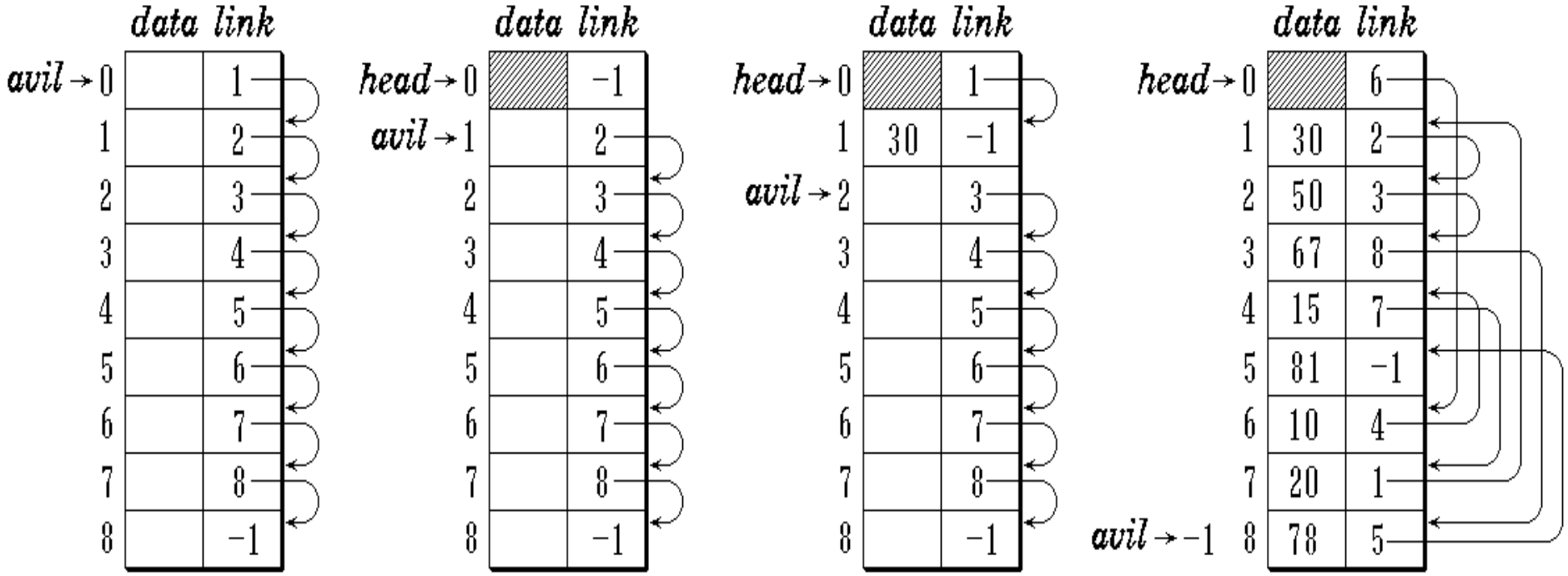






静态链表结构

利用数组定义，运算过程中存储空间大小不变



分配节点：　*j = avil; avil = A[avil].link;*

释放节点：　*A[i].link = avil; avil = i;*

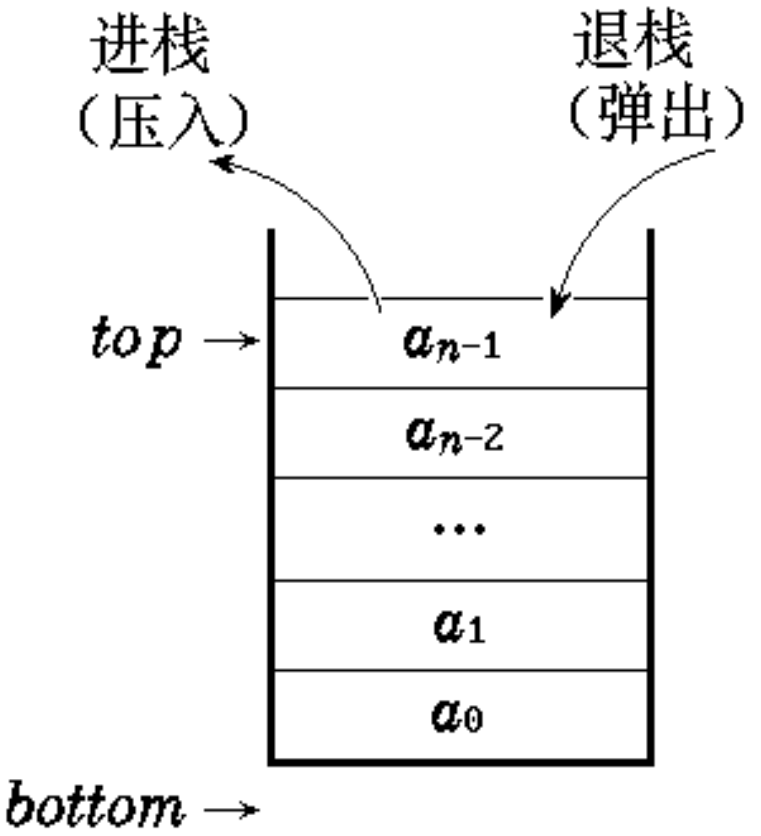
二、栈和队列

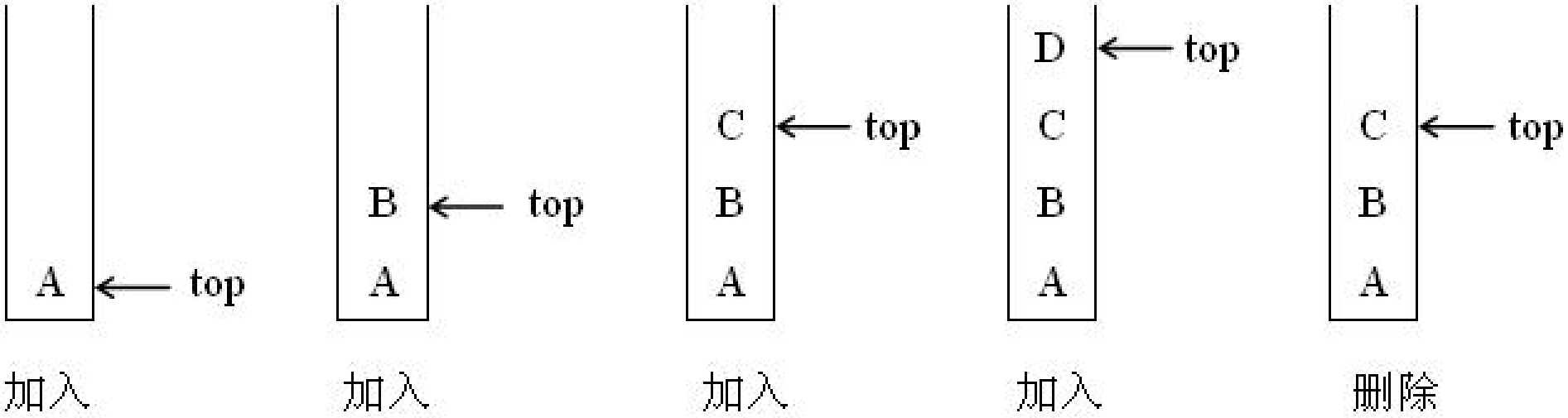
- 栈的表示和实现
- 应用实例
- 队列的表示和实现
- 应用实例

栈（ Stack ）

只允许在一端插入和删除的线性表。允许插入和删除的一端称为栈顶(*top*)，另一端称为栈底(*bottom*)

后进先出 (*LIFO*)





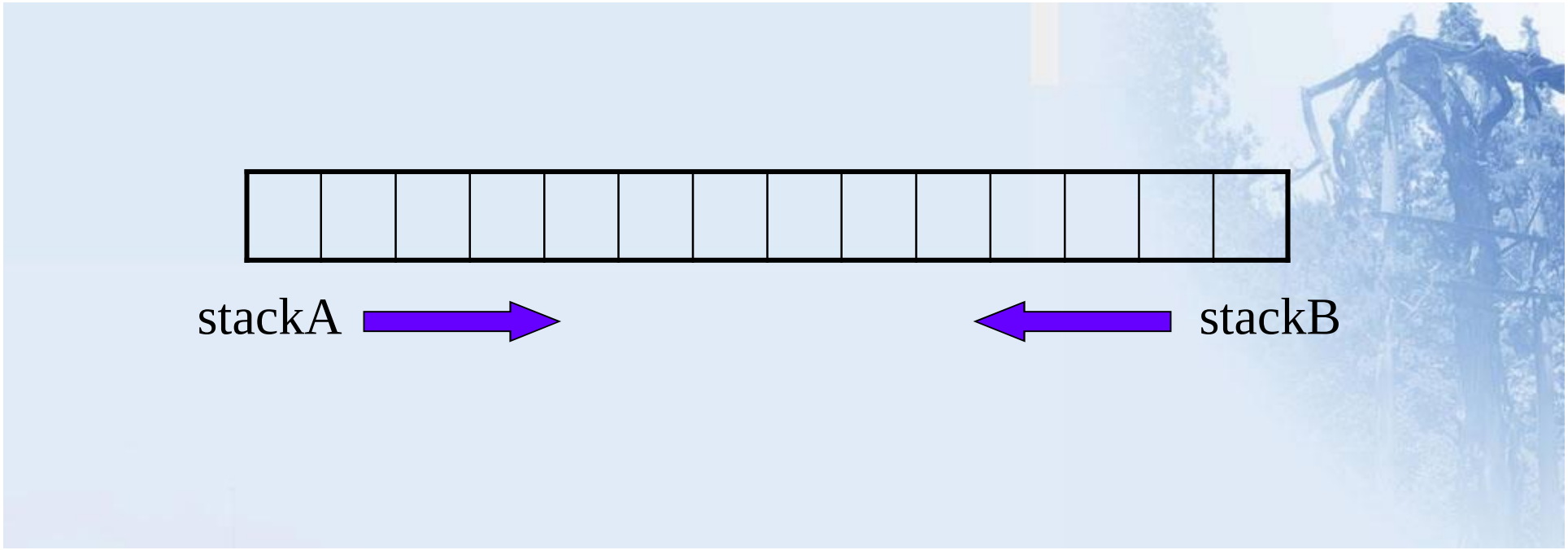
顺序栈 — 用数组表示的栈

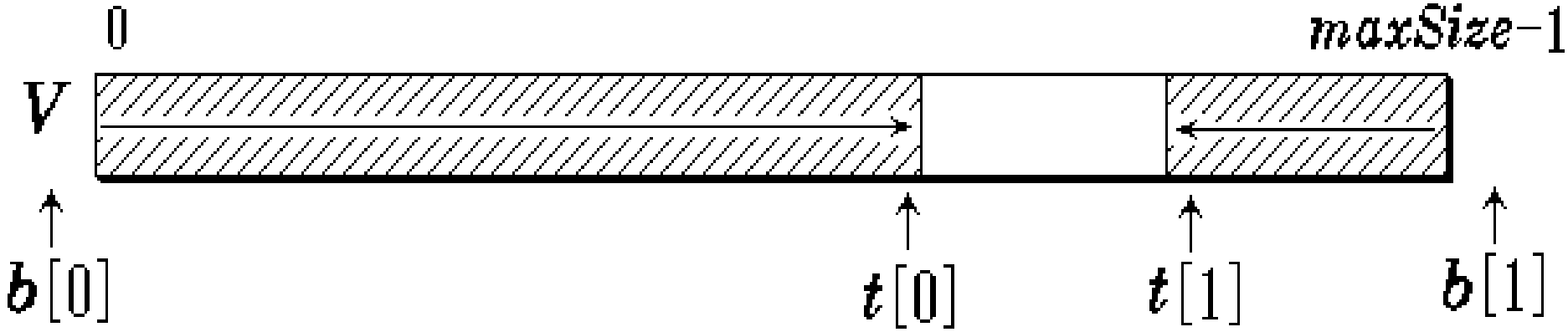
- 与顺序存储结构的线性表一样，利用一组地址连续的存储单元依次存放自栈底到栈顶的数据元素(称为顺序栈)；
- 可以使用一维数组来作为栈的顺序存储空间；
- 设指针**top**指向栈顶元素的当前位置；
- 以数组小下标的一端作为栈底，通常以**top=-1**时为空栈；
- 在元素进栈时指针**top**加**1**；
- 当**top**等于数组的最大下标值时则栈满(上溢)。

两个堆栈共享空间

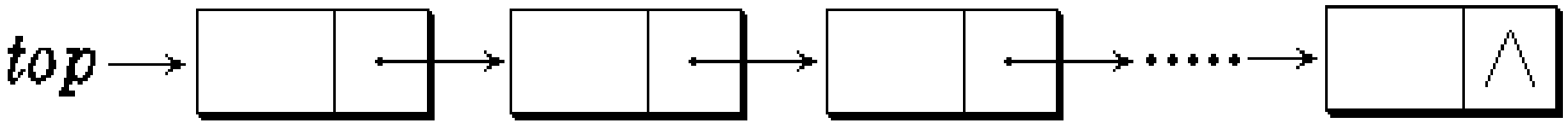
在计算机系统软件中，各种高级语言的编译系统都离不开栈的使用。常常一个程序中要用到多个栈，为了不发生上溢错误，就必须给每个栈预先分配一个足够大的存储空间，但实际中很难准确地估计。另一面方面，若每个栈都预分配过大的存储空间，势必会造成系统空间紧张。若让多个栈共用一个足够大的连续存储空间，则可利用栈的动态特性使它们的存储空间互补。这就是栈的共享邻接空间。

栈的共享中最常见的是两栈的共享。假设两个栈共享一维数组**stack[MAXNUM]**，则可以利用栈的“栈底位置不变，栈顶位置动态变化”的特性，两个栈底分别为**-1**和**MAXNUM**，而它们的栈顶都往中间方向延伸。因此，只要整个数组**stack[MAXNUM]**未被占满，无论哪个栈的入栈都不会发生上溢。





栈的链接表示 — 链式栈



栈也可以采用链式存储结构表示，这种结构的栈简称为链栈。在一个链栈中，栈底就是链表的最后一个结点，而栈顶总是链表的第一个结点。因此，新入栈的元素即为链表新的第一个结点，只要系统还有存储空间，就不会有栈满的情况发生。一个链栈可由栈顶指针**top**唯一确定，当**top**为**NULL**时，是一个空栈。

- 链式栈基本无栈满问题，空间可扩充
- 插入与删除仅在栈顶处执行
- 链式栈的栈顶在链头
- 适合于多栈操作

【例】对于一个栈，给出输入项A、B、C，如果输入项序列是ABC，试给出所有可能的输出序列。

A进 A出 B进 B出 C进 C出	ABC
A进 A出 B进 C进 C出 B出	ACB
A进 B进 B出 A出 C进 C出	BAC
A进 B进 B出 C进 C出 A出	BCA
A进 B进 C进 C出 B出 A出	CBA

不可能产生输出序列CAB

【例】用S表示入栈操作，X表示出栈操作，若元素入栈顺序为1234，为了得到1342出栈顺序，相应的S和X操作顺序为：

S X S S X S X X

问题：如何判断一个入栈和出栈的操作序列是否合法？

【例】若让元素1，2，3依次进栈，则出栈次序不可能出现____的情况。

（A） 3，2，1

（B） 2，1，3

√ （C） 3，1，2

（D） 1，3，2

栈的数学性质：当多个编号元素依某种顺序压入，且可任意时刻弹出时，所获得的编号元素排列的数目，恰好满足卡塔南数列的计算，即：

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

$$= \frac{1}{n+1} \times \frac{(2n)!}{(2n-n)! \times n!}$$

$$= \frac{1}{n+1} \times \frac{(2n)!}{n! \times n!}$$

其中，**n**为编号元素的个数，$\mathbf{C_n}$是可能的排列数目。例如，**n＝3**时，有：

$$C_3 = \frac{1}{n+1}\binom{2n}{n} = \frac{1}{3+1} \times \frac{6!}{3! \times 3!} = \frac{1}{4} \times \frac{6 \times 5 \times 4}{3 \times 2 \times 1} = 5$$

用列表实现栈

```
class Stack:
    def __init__(self): self.list=[]
    def IsEmpty(self):
        if len(self.list)==0 : return True
        else : return False
    def Push(self, x):
        self.list.append(x)
    def Pop(self):
        if not self.IsEmpty( ) :
            return self.list.pop(len(self.list)-1)
        else : return None
    def Top(self):
        if not self.IsEmpty( ) :
            return self.list[len(self.list)-1]
        else : return None
```

【例】假设一个算术表达式中包含圆括号、方括号和花括号，编写一个判断表达式中括号是否匹配的程序。

算法思想：
建立一个栈。顺序对表达式进行扫描，如遇到“(”、“[”、“{”，将其入栈，当遇到“)”、“]”、“}”时，检查当前栈顶元素是否为对应的“(”、“[”、“{”，若是则退栈，否则表示不配对。整个算术表达式检查完毕时栈为空，表示括号正确配对，否则不配对。

【例】所谓回文，是指从前向后顺读和从后向前倒读都一样的不含空白字符的串。例如**did**、**madamimada m**、**pop**即是回文。试编写一个算法，以判断一个串是否是回文。

【算法1】将字符串中全部字符进栈，然后将栈中的字符逐个与原字符串中的字符进行比较。算法如下：

```
def  palindrome(str) : #用栈
    S= Stack( )
    for  i in range(len(str)) :  # 扫描，所有字符进栈
        S.Push(str[i])
    for  i in range(len(str)) :  # 比较字符串
        if  str[i] != S.Pop() : return 'No'
    return 'Yes'
```

【其他算法】采用首尾比较的方式，判断从字符串是否回文，返回是或不是。

```
def  palindrome(str) : #不用栈，首尾比较
    n = len(str)
    for  i in range(n) :
        if  str[i] != str[n-i-1] : return 'No'
    return 'Yes'
```

```
def  palindrome(str, i, j) : #不用栈，递归首尾比较
    if  i < j :
        if  str[i] == str[j] : return palindrome(str, i+1, j-1)
        else : return False
    return True
```

栈的典型应用 --- 表达式计算

表达式求值是程序设计语言编译中的一个最基本问题之一。它的实现方法是栈的一个典型的应用实例。

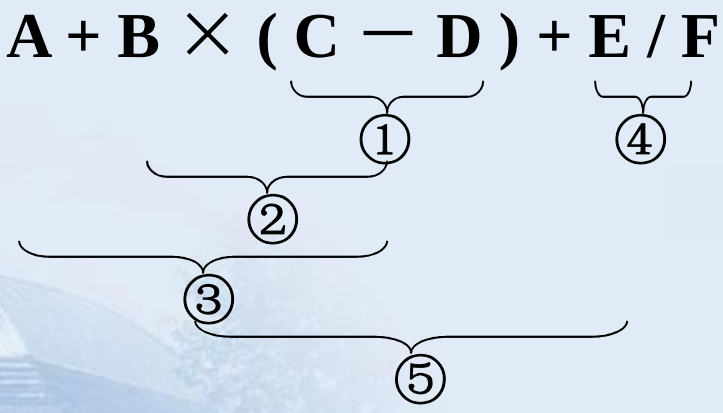
在计算机中，任何一个表达式都是由操作数（**operand**）、操作符（**operator**）和分隔符（**delimiter**）组成的。其中操作数可以是常数，也可以是变量或常量的标识符；操作符可以是算术运算符、关系运算符和逻辑运算符；分隔符为左右括号和标识表达式结束的结束符。在本节中，仅讨论简单算术表达式的求值问题。在这种表达式中只含加、减、乘、除四则运算，所有的运算对象均为单变量。表达式的结束符为“#”。

算术四则运算的规则为：

（1）先乘除、后加减；

（2）同级运算时先左后右；

（3）先括号内，后括号外。



计算表达式时是如何确定计算顺序的呢？是通过操作符及其分隔符所决定的运算优先级。下面是C++操作符的优先级，其中1级最高：

优先级	操作符
1	负号(-)，！
2	*，/，%
3	+，-
4	<，<=，>=，>
5	==，!=
6	&&
7	||

计算机是如何接收一个表达式并生成正确的可执行指令呢？回答是计算机将输入的表达式重写为一个称之为后缀形式的表达式。对于由操作符和操作数组成的表达式，我们通常的写法称作中缀形式，这是因为操作符在操作数中间。类似的理由，如果把操作符放在操作数后边则被称作后缀形式。

一种解决表达式计算的有效方法可分为两步：

（1）将中缀转化为后缀；

（2）计算后缀。

我们书写的表达式一般都是运算符在两个操作数中间（除单目运算符外），这种表达式被称为中缀表达式。中缀表达式有时必须借助括号才能将运算顺序表达清楚，处理起来比较复杂。在编译系统中，对表达式的处理采用的是另外一种方法，即将中缀表达式转变为后缀表达式，然后对后缀式表达式进行处理，后缀表达式也称为逆波兰式。

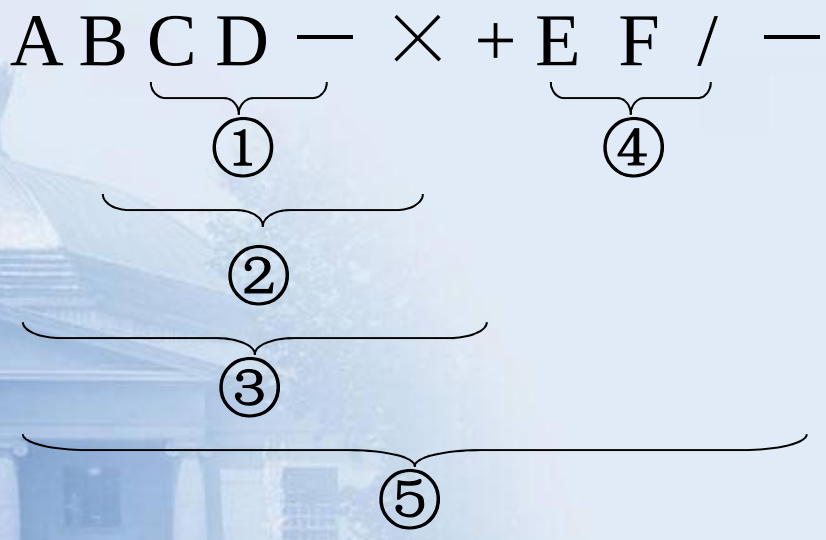
波兰表示法（也称为前缀表达式）是由波兰逻辑学家（**Lukasiewicz**）提出的，其特点是将运算符置于运算对象的前面，如**a+b**表示为**+ab**；逆波兰式则是将运算符置于运算对象的后面，如**a+b**表示为**ab+**。后缀表达式运算时按从左到右的顺序进行，不需要括号。

后缀表达式的计算规则：

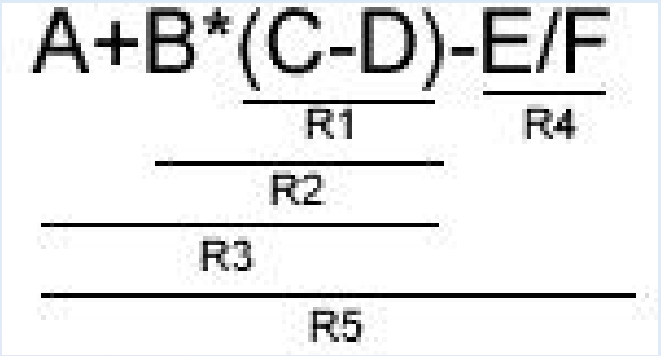
从左向右，先找操作符，再找操作数

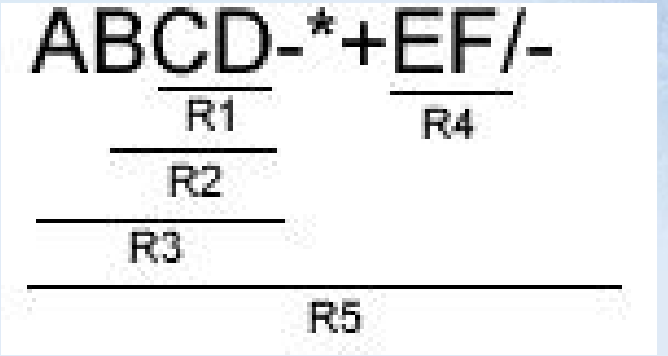
中缀：　**A+B*(C-D)-E/F**

后缀：　**ABCD-*+EF/-**



通过演示观察处理过程





后缀表达式特点

1、与相应的中缀表达式中的操作数次序相同

2、没有括号

3、操作符的次序决定了表达式的计算顺序

后缀表达式的计算

算法实现：得到后缀表达式后，我们在计算表达式时，可以设置一个栈，从左到右扫描后缀表达式，每读到一个操作数就将其压入栈中；每读到一个操作符时，则从栈顶取出两个操作数进行运算，并将结果压入栈中，一直到后缀表达式读完。最后栈顶就是计算结果。

处理规则1：扫描到操作数时，操作数入栈；
处理规则2：扫描到操作符时，从栈顶相应操作数，操作结果入栈。

A B C D － × + E F / －

步骤	扫描项	栈中内容	说明
1		空	置空栈
2	A	A	
3	B	AB	
4	C	ABC	
5	D	ABCD	
6	－	ABR_1	$R_1 = C－D$
7	×	AR_2	$R_2 = B \times R_1$
8	＋	R_3	$R_3 = A + R_2$
9	E	R_3E	
10	F	R_3EF	
11	/	R_3R_4	$R_4 = E / F$
12	－	R_5	$R_5 = R_3 －R_4$

通过演示观察计算过程

```
def  eval (e)  # e是表达式
    S = Stack();  //initialize stack
    在表达式e中取一个操作数或操作符x
    while x is not “#” :
        if  x是操作数 : S.Push(x)
        else :
            从栈中托出与x操作相对应的操作数
            完成x指定的操作，结果存入y
             S.Push(y)
        在表达式e中取一个操作数或操作符x
    return S.Pop( )
#end of eval
```

```
def  eval(E) : # 计算后缀表达式
    S = Stack( )  # initialize stack
    for a in E :
        if  a == '#' : return S.Pop( )
        elif  a[0] >= '0' and a[0] <= '9' : # 数字
            S.Push(int(a))
        else :
            x = S.Pop( );  y = S.Pop( )
            if  a == '+' : S. Push(y + x)
            elif  a == '-' : S.Push(y - x)
            elif  a == '*' : S.Push(y * x)
            elif  a == '/' : S.Push(y / x)}
# Evaluating postfix expressions
```

中缀化为后缀

首先讨论如何用手工的方式将表达式的中缀形式化为后缀形式。具体步骤如下：

1. 在表达式的每一步计算的子表达式前后加一对括号；
2. 移动所有的操作符并用其替换完成此操作符运算的子表达式的右括号；
3. 删除所有的左括号；

A / B – C + D * E - A * C

((((A / B) – C) + (D * E)) - (A * C))

((((A B / C – (D E * + (A C * -

A B / C – D E * + A C * -

例子

中缀: A + B * C

后缀: A B C * +

实际的算法设计中，为方便起见，仍然利用栈。

处理规则1：x为操作数时，直接输出。

处理规则2：x为操作符时，如栈为空，x入栈；如栈不为空，从栈顶取出操作符y，将其优先级与x的优先级进行比较，如果x的优先级大于y的优先级，x入栈，否则，y输出，x继续与栈顶元素比较。

处理规则3：扫描完毕，如栈不为空，重复退栈并输出，直到栈为空。

A + B * C

扫描字符	栈的内容	输出
none	empty	none
A	empty	A
+	+	A
B	+	AB
*	+*	AB
C	+*	ABC
		ABC*
		ABC*+

操作符	优先数
unary minus,!	1
*, /, %	2
+, -	3
<, <=, >=, >	4
==, !=	5
&&	6
||	7
#	8

例子

中缀：　A/B-C+D*E-A*C

后缀：　AB/C-DE*+AC*-

扫描字符	栈的内容	输出
none	empty	none
A	empty	A
/	/	A
B	/	AB
-	-	AB/
C	-	AB/C
+	+	AB/C-
D	+	AB/C-D
*	+ *	AB/C-D
E	+ *	AB/C-DE
-	-	AB/C-DE*+
A	-	AB/C-DE*+A
*	- *	AB/C-DE*+A
C	- *	AB/C-DE*+AC
		AB/C-DE*+AC*-

表达式中的括号使得表达式的处理稍微复杂一些，需要对操作符赋予两个优先数：栈内优先数和栈外优先数。

操作符	栈内优先数	栈外优先数
unary minus,!	1	1
*, /, %	2	2
+, -	3	3
<, <=, >=, >	4	4
==, !=	5	5
&&	6	6
||	7	7
(	8	0
#	8	

处理规则4：当栈顶操作符的栈内优先数小于或等于当前扫描的操作符的栈外优先数时（注意：这里优先数越小，优先级越高），栈顶元素出栈。

处理规则5：x为‘)’时，逐次输出栈中操作符，直到栈中操作符为‘(’为止。最后再做一次退栈操作(取掉栈顶元素‘(’)。

例子

中缀: **A * (B + C) * D**
后缀: **A B C + * D ***

当前字符	栈的内容	输出
none	#	none
A	#	A
*	#*	A
(	#*(	A
B	#*(	AB
+	#*(+	AB
C	#*(+	ABC
)	#*	ABC+
*	#*	ABC+*
D	#*	ABC+*D
none	#	ABC+*D*

操作符	栈内	栈外
unary minus,!	1	1
*, /, %	2	2
+, -	3	3
<, <=, >=, >	4	4
==, !=	5	5
&&	6	6
||	7	7
(	8	0
#	8	

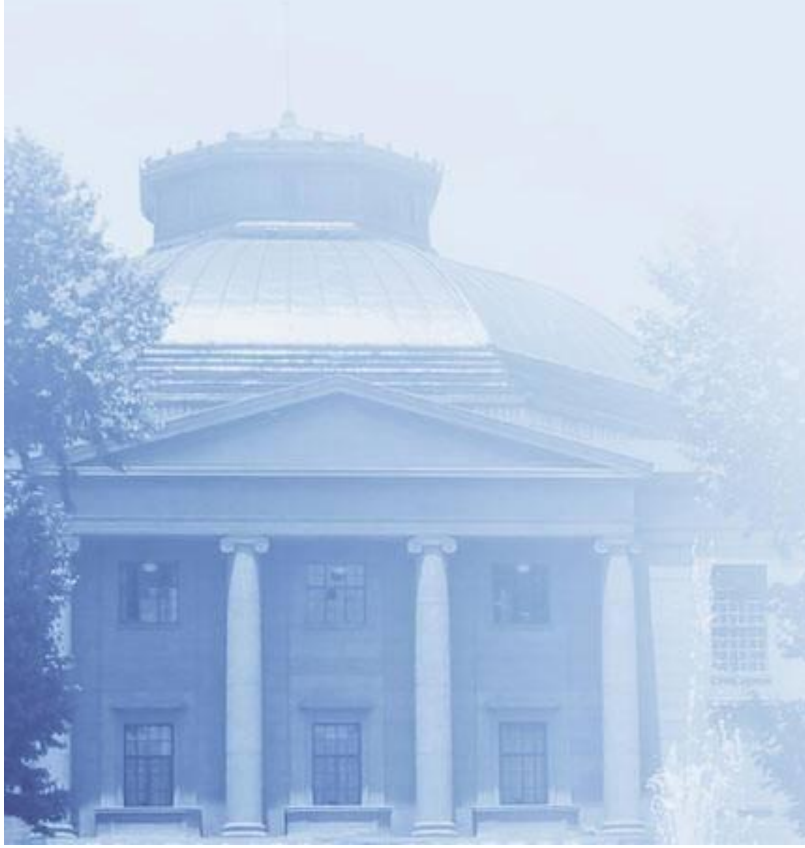
通过演示观察算法执行的过程

```
def  postfix(e)
# 假定表达式的最后一个符号及栈底均为 '#'
  S=Stack( )
  S.Push('#')
  for  e的每一个符号 x, x!='#'
      if   x是操作数 ： 输出 x
      elif  x是 ')' :
          y=S.Pop()  #栈中符号出栈
          while  y is not '(' ： # 直到出栈符号是 '(' 为止
              输出y
              y=S.Pop()  #继续取栈中符号
      else ： # x是操作符
          y=S.Pop()
          while  isp(y)<=osp(x) :  #比较y的栈内优先数和x的栈外优先数
              输出y
              y=S.Pop()
          S.Push(y),  S.Push(x)
  while  not S.IsEmpty( ) : 表达式处理完后栈依然不空
      输出S.Pop()
#Converting from infix to postfix form
```

```
def  Postfix(E) : #中缀转化为后缀
    R = [ ]
    isp = {'*':2, '/':2, '+':3, '-':3, '(': 8, '#':8}
    osp = {'*':2, '/':2, '+':3, '-':3, '(': 0}
    S = Stack()  # initialize stack
    S.Push('#')
    for  x in E :
        if  x[0] >='0' and x[0] <='9' : # x is an operand
            R.append(x)
        elif  x == ')' :
            y = S.Pop( )
            while  y != '(' :
                R.append(y);  y = S.Pop( )
        else :
            y = S.Pop( )
            while  isp[y] <= osp[x] :
                R.append(y);  y = S.Pop( )
            S.Push(y); S.Push(x)
    while  not S.IsEmpty( ) :
        R.append(S.Pop( ))
    return R
```

算法分析

算法对表达式从左到右扫描一遍，对每个操作数处理的时间为**O(1)**；每个操作符最多各入栈和出栈一次，因此每个操作符处理的时间也是**O(1)**，所以，总的时间复杂度为**O(n)**。这里，**n**为表达式中的符号数。



将字符串解析为表达式

```
def resolve( Str ) : # 解析表达式字符串，返回结果列表
    R = [] # 存放解析结果
    tmp = ''
    for s in Str:
        if  s < '0' and s != '.' : # 运算符或分隔符
            if  len(tmp) > 0 :
                R.append(tmp)
                tmp = ''
            R.append(s)
        else :  # 数字
            tmp += s
    if len(tmp) > 0 :
        R.append(tmp)
    return R
```

栈的典型应用 --- 迷宫问题求解

我们将迷宫问题表示为两维数组**maze[m][n]**，数组中下标为**i，j（1≤i≤m，1≤j≤n）**的元素为**1**时表示此路不通，为**0**时表示有路可走。假定开始点为**maze[1][1]**，出口为**maze[m][n]**。

入口

0	1	1	1	0	1	1	1
1	0	1	0	1	0	1	0
0	1	0	0	1	1	1	1
0	1	1	1	0	0	1	1
1	0	0	1	1	0	0	0
0	1	1	0	0	1	1	0

出口

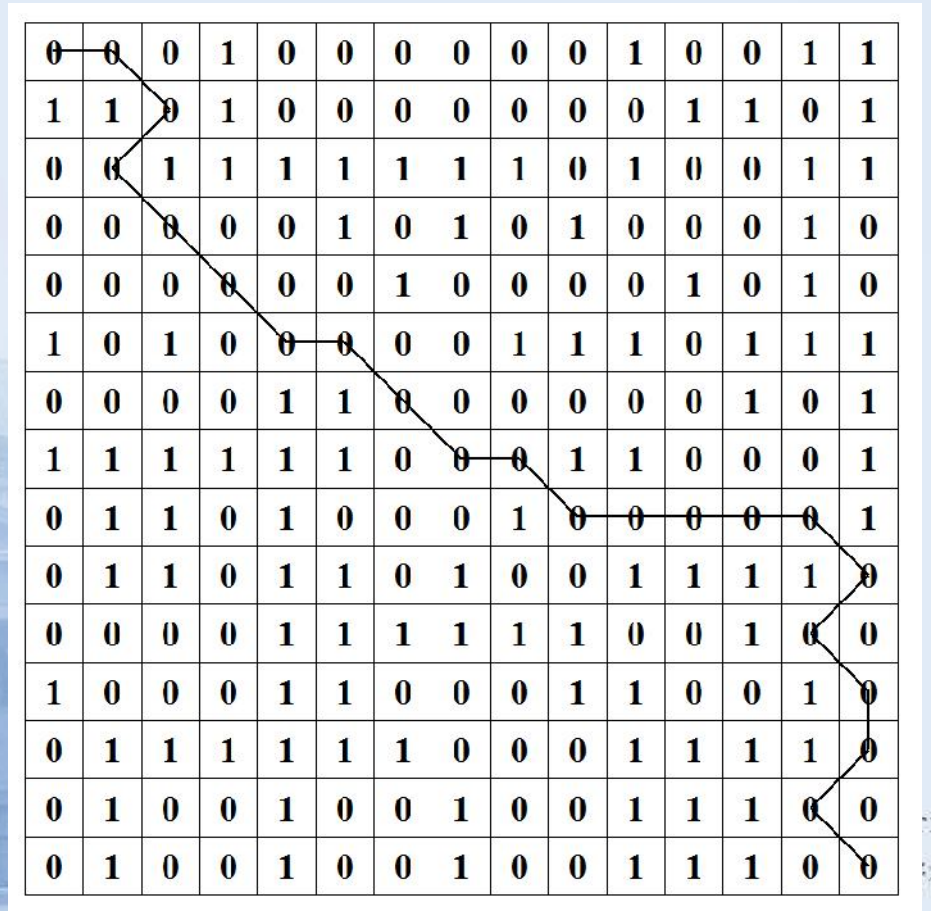
通过演示观察迷宫的搜索过程

需要解决的问题：

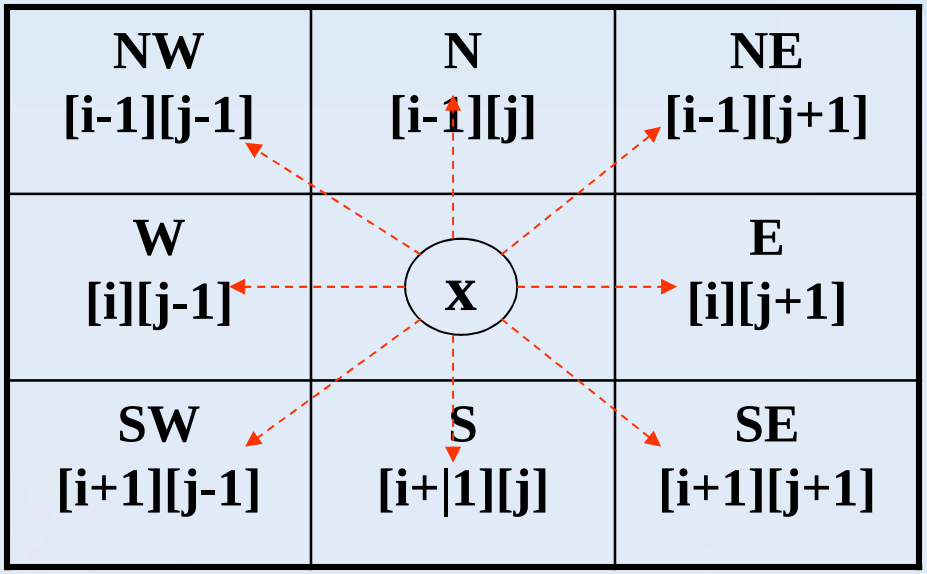
1、在任意位置如何搜索周围的8个方向？

2、如何避免重复走老路？

3、如何保存走过的路径以便回朔时利用？



1、解决问题1




```
# 定义方向数组
move = [(1,1),(1,0),(1,-1),(0,-1),(-1,-1),(-1,0),(-1,1),(0,1)]
……
while d < 8 :
    x, y = i + move[d][0], j + move[d][1]
    ……
```

```
# 定义方向数组
move = [(1,1),(1,0),(1,-1),(0,-1),(-1,-1),(-1,0),(-1,1),(0,1)]
……
while d < 8 :
    x, y = i + move[d][0], j + move[d][1]
    ……
```

d	move[q].[0]	move[d].[1]
N	-1	0
NE	-1	1
E	0	1
SE	1	1
S	1	0
SW	1	-1
W	0	-1
NW	-1	-1

思路一：扩大矩阵，可以不检查是否越界

1	1	1	1	1	1	1	1	1	1
1	0	1	1	1	0	1	1	1	1
1	1	0	1	0	1	0	1	0	1
1	0	1	0	0	1	1	1	1	1
1	0	1	1	1	0	0	1	1	1
1	1	0	0	1	1	0	0	0	1
1	0	1	1	0	0	1	1	0	1
1	1	1	1	1	1	1	1	1	1

两维数组定义为： **maze[m+2][n+2].**

思路二：在移动每一步前，检查是否越界

NW [i-1][j-1]	N [i-1][j]	NE [i-1][j+1]
W [i][j-1]		E [i][j+1]
SW [i+1][j-1]	S [i+|1][j]	SE [i+1][j+1]

```
if  x<0 or x==n or y<0 or y==n or
    A[x][y] == 1 or A[x][y] == -1 :  # 越界
    d = d+1    # 转到下一方向
    continue
```

2、解决问题2

为了防止重复走老路，我们定义一个与maze数组同样大小的数组mark，当位置i，j走过后，置mark[i][j]为1。另一个思路是将maze[i][j]值为-1，只要能与0相区别即可

3、解决问题3

定义一个栈来保存走过的路径。栈的元素为一个三元组**(i, j, d)**。栈的大小定义至多为**m×p**，因为数组**maze**中每个位置至多只走一次。

step	i	j	d
1	1	1	SE
2	2	2	SE
3	3	4	E
4	3	4	SE
…	…	…	…

	1	2	3	4	5	6	7	8
1	0	1	1	1	0	1	1	1
2	1	0	1	0	1	0	1	0
3	0	1	0	0	1	1	1	1
4	0	1	1	1	0	0	1	1
5	1	0	0	1	1	0	0	0
6	0	1	1	0	0	1	1	0

利用栈进行回朔时，首先需要判断出栈元素所标示的位置的八个方向是否都被搜索过，即三元组(i, j, d)中d是否为方向数组下标的最大值。

1、如果其八个方向未被全部搜索完，(d不是方向数组下标的最大值)，则调整搜索方向（d=d+1），将调整过的三元组(i,j,d)入栈，然后按新的方向继续搜索。所以，一个坐标位置最多可能8次入栈。

2、否则，下一个栈顶元素出栈，继续进行检查。

迷宫求解算法框架

```
初始化栈S, 将入口坐标和第一个方向（东）加入S;
while  stack 不空 :
    ( i, j, dir ) = S.Pop( )  # 出栈元素的下标和方向;
    while  还有方向可试 :
        ( g, h ) = 下一个移动点的坐标;
        if   g == m and  h == n : # 成功，输出路径;
        if  !maze[g][h] and !mark[g][h] :  # 合法且未被访问过
            mark[g][h] = 1  # 或 maze[g][h] = -1
            dir = 下一个可试的方向;
            将( i, j, dir ) 加入S;
             i = g; j = h; dir = “北”;
输出 “不存在路径”;
```

这里，栈S的元素是三元组，可用元组类型定义：

(row, col, dir)

由此可得迷宫路径算法FindPath

```
def FindPath( A, move, S ) :
    n = len(A)
    while not S.IsEmpty( ) :
        tup = S.Pop( )
        i, j, d = tup[0], tup[1], tup[2]+1
        while d < 8 :
            x, y = i + move[d][0], j + move[d][1]
            if  x<0 or x==n or y<0 or y==n or A[x][y] == 1
                    or A[x][y] == -1 :
                d = d+1
                continue
```

```
        if  A[x][y] == 0 : #该点可达
            S.Push((i, j, d)) #新点前一点入栈
            i, j = x, y #重置新点
            A[i][j] = -1 #新点前一点标志已经到达
            if  i == n-1 and j == n-1 : #到达出口
                Output(A, S)
                return
            else : d = 0
        else : d += 1  #改变方向
print 'No Ptath existed! '
```

迷宫路径算法FindPath（递归）

```
def  FindPathRecursion(A, move, S, x, y) :  #递归方式
    n = len(A)
    S.Push((x, y, 0))
    if  x == n-1 and y == n-1 : #到达出口
        print 'The Path Is Exist!'
        Output(A, S)
        return
    A[x][y] = -1
    for u in move :
        x1, y1 = x + u[0], y + u[1]
        if  x1>= 0 and x1<n and y1>=0 and y1<n :
            if  A[x1][y1] == 0 :
                FindPathRecursion(A, move, S, x1, y1)
                S.Pop( )
```

探索成功时，栈S的内容构成了所求路径的主要部分，需要输出。方式一为根据S的内容重构矩阵，然后输出矩阵。方式二为直接输出S中的路径坐标信息。

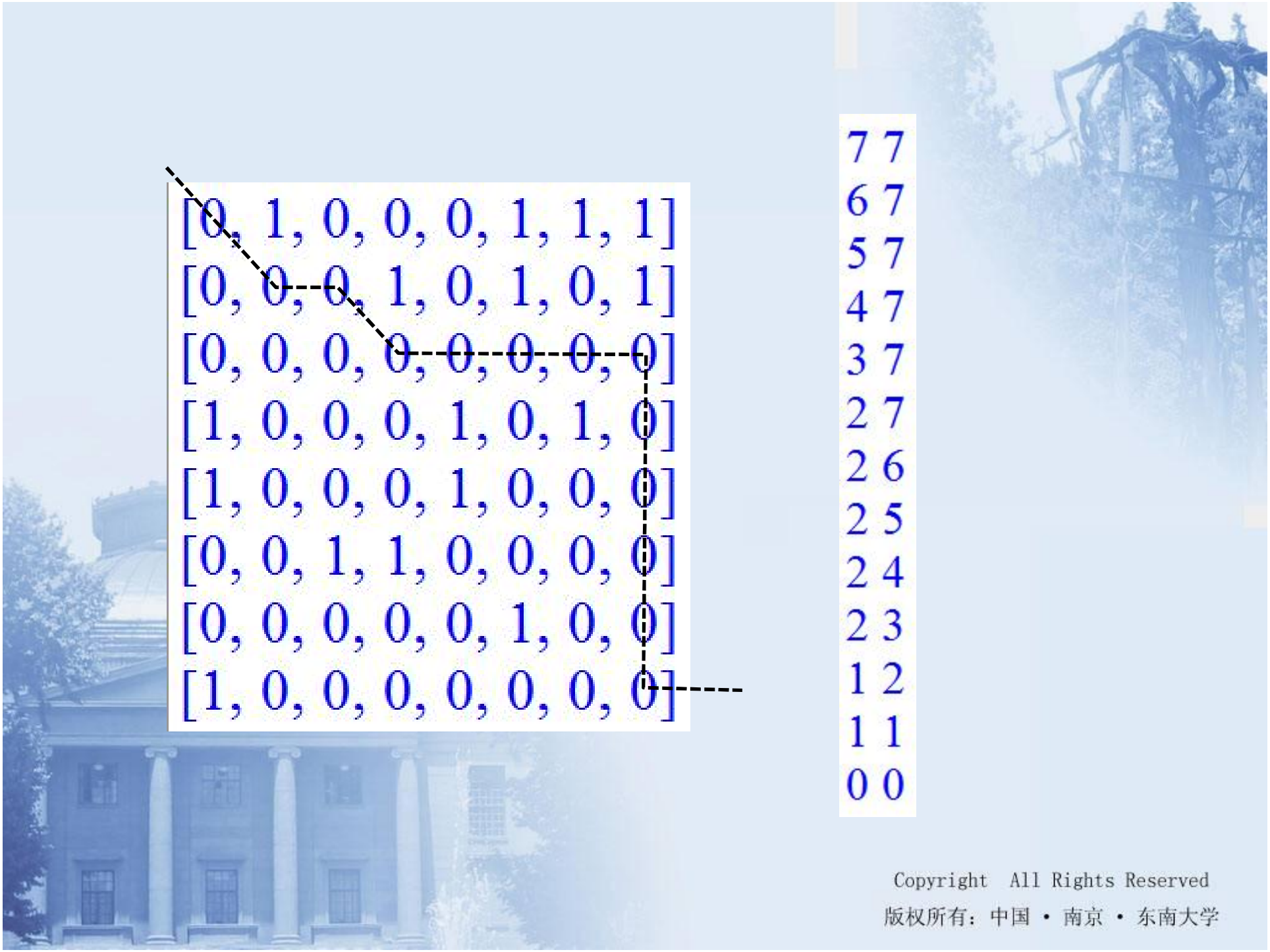
```
def Output(A, s) : # 输出形式为矩阵
    n = len(A)
    for i in range(n) :
        for j in range(n) :
            if A[i][j] == -1 : A[i][j] = 0
    A[0][0], A[n-1][n-1] = 8, 8
    while not s.IsEmpty() :
        tup = s.Pop()
        x, y = tup[0], tup[1]
        A[x][y] = 8
    for i in A : print i
```

[0, 0, 1, 1, 0, 0, 1, 0]
[1, 0, 1, 0, 0, 1, 0, 0]
[1, 0, 0, 1, 1, 1, 0, 0]
[1, 1, 0, 1, 0, 0, 0, 1]
[0, 1, 0, 1, 1, 0, 1, 1]
[0, 0, 0, 0, 1, 1, 1, 1]
[0, 0, 1, 1, 0, 0, 0, 1]
[0, 0, 0, 1, 1, 1, 0, 0]

[8, 8, 1, 1, 0, 0, 1, 0]
[1, 8, 1, 0, 0, 1, 0, 0]
[1, 0, 8, 1, 1, 1, 0, 0]
[1, 1, 8, 1, 0, 0, 0, 1]
[0, 1, 8, 1, 1, 0, 1, 1]
[0, 0, 0, 8, 1, 1, 1, 1]
[0, 0, 1, 1, 8, 8, 8, 1]
[0, 0, 0, 1, 1, 1, 0, 8]

另外一种输出形式

```
def Output1(A, s) : # 输出形式为路径
    n = len(A)
    print n-1,n-1
    while not s.IsEmpty() :
        tup = s.Pop()
        print tup[0], tup[1]
```



调用形式

```
n = 8
A = [ [0 for i in range(0, n)] for j in range(0, n)]
move = [(0,1),(1,1),(1,0),(1,-1),(0,-1),(-1,-1),(-1,0),(-1,1)]
S = Stack( )
S.Push((0, 0, -1))  #起点
for i in range(0, n) :
    for j in range(0, n) :
        A[i][j] = int(random.uniform(0, 1.6))
A[0][0], A[n-1][n-1] = 0, 0
for i in A : print i
print
FindPath(A, move, S)
```

递归算法的调用形式

```
n = 8
B = [ [0 for i in range(0, n)] for j in range(0, n)]
move = [(0,1),(1,1),(1,0),(1,-1),(0,-1),(-1,-1),(-1,0),(-1,1)]
S = Stack( )
for i in range(0, n) :
    for j in range(0, n) :
        B[i][j] = int(random.uniform(0, 1.6))
B[0][0], B[n-1][n-1] = 0, 0
S = Stack( )
for i in B : print i
FindPathRecursion(B, move, S, 0, 0)
```

说明：

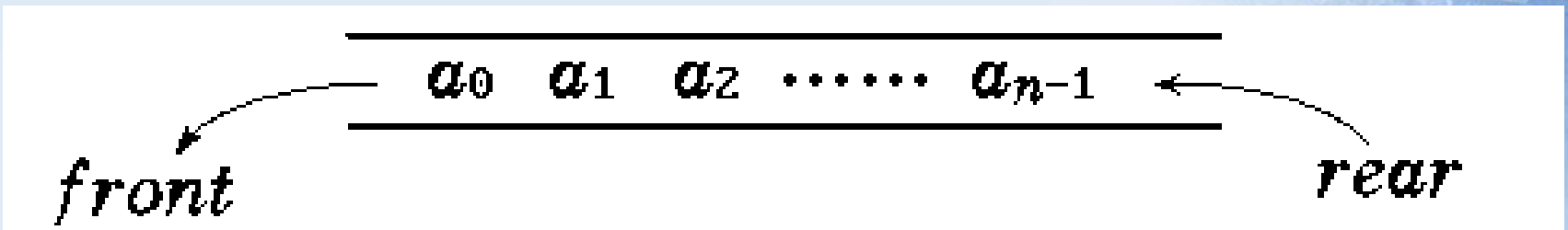
- 栈的大小为**m×n**，栈中每个元素由**x**，**y**坐标和与下一搜索方向组成。
- 任何出栈元素，在改变方向后重新入栈（即栈内存储的是写一次搜索的方向），除非以它为中心的**8**个方向都已搜索过。
- 栈中元素为已经走过的路径的位置（回朔部分不在其中），当前所在位置不在栈中。因此，输出搜索路径时，除了输出栈中元素外，还需输出当前位置和出口位置。

分析：

- **path的计算时间依赖于给定迷宫的内容。**
- **主while循环的迭代次数由入栈元素个数决定。每个新点[i][j]被访问后即被标记，且一个点不可能被标记两次，未被标记的点不可能入栈。设z是迷宫中可通行点数，则最多有z个点被标记，被标记的点入栈常数次。由于z≤mòn，所以入栈元素的数量最多是O(mòn)。**
- **内while循环最多迭代8次，其时间为O(1)。**
- **整个算法的最坏时间复杂性是O(mòn)。**

队列（Queue）

- 日常生活中体现“先来先服务”原则的实例很多；
- 操作系统中的作业排队（体现先来先服务），必要时形成多个队列；
- 计算机系统中输入输出缓冲区的结构也是队列，用以解决速度不匹配的矛盾。

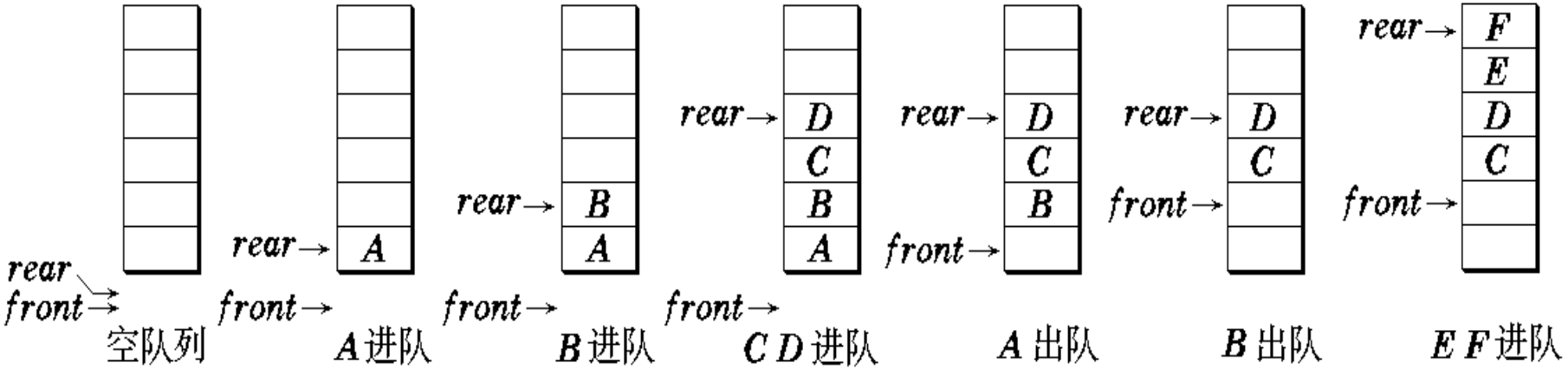


定义　队列是只允许在一端删除，在另一端插入的顺序表。允许删除的一端叫做队头(front)，允许插入的一端叫做队尾(rear)。

特性

先进先出(FIFO, First In First Out)

队列的进队和出队



队列是一种特殊的线性表，因此队列可采用顺序存储结构存储，也可以使用链式存储结构存储。

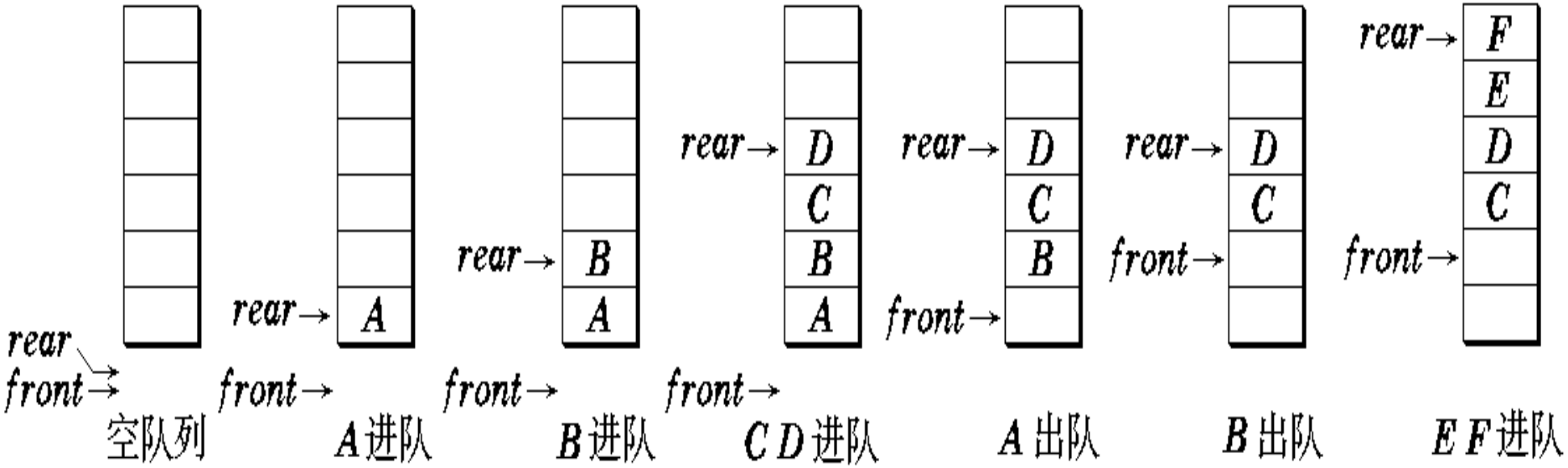
- 进队时队尾指针先进一， **rear = rear + 1**，再将新元素按 **rear** 指示位置加入(队列不空时，队尾指针指向最后一个元素)。
- 出队时队头指针先进一，**front = front + 1**，再将下标为 **front** 的元素取出。 (队列不空时，队头指针指向第一个元素的前面)
- 队满时再进队将溢出出错；队空时再出队将做队空处理。
- 队列为空时，队尾指针＝对头指针。

队列的基本运算

- 初始化：初始化一个新的队列。
- 队列非空判断　**IsEmpty( )**：若队列不空，则返回**TRUE**；否则，返回**FALSE**。
- 入队列 **EnQueue(x)**：在队列的尾部插入元素**x**，使元素**x**成为新的队尾。若队列满，则返回**FALSE**；否则，返回出队列
- 出队列 **DeQueue( )**：若队列不空，则返回队头元素，并从队头删除该元素，队头指针指向原队头的后继元素；否则，返回空元素**NULL**。
- 取队头元素　**GetFront( )**：若队列不空，则返回队头元素；否则返回空元素**NULL**。
- 求队列长度 **Length( )**：返回队列的长度。

循环队列 (Circular Queue)

在顺序队列中，当队尾指针已经指向了队列的最后一个位置时，此时若有元素入列，就会发生“溢出”。也就是说，队列的存储空间并没有满，但队列却发生了溢出，我们称这种现象为假溢出。解决这个问题有两种可行的方法：



（1）采用平移元素的方法，当发生假溢出时，就把整个队列的元素平移到存储区的首部，然后再插入新元素。这种方法需移动大量的元素，因而效率是很低的。

（2）将顺序队列的存储区假想为一个环状的空间。我们可假想**q->queue[0]**接在**q->queue[MAXNUM-1]**的后面。当发生假溢出时，将新元素插入到第一个位置上，这样做，虽然物理上队尾在队首之前，但逻辑上队首仍然在前。入队和出队仍按“先进先出”的原则进行，这就是循环队列。

很显然，方法二中不需要移动元素，操作效率高，空间的利用率也很高。

存储队列的数组被当作首尾相接的表处理。

队头、队尾指针加**1**时从***maxSize*** **−1**直接进到**0**，可用语言的取模**(**余数**)**运算实现。

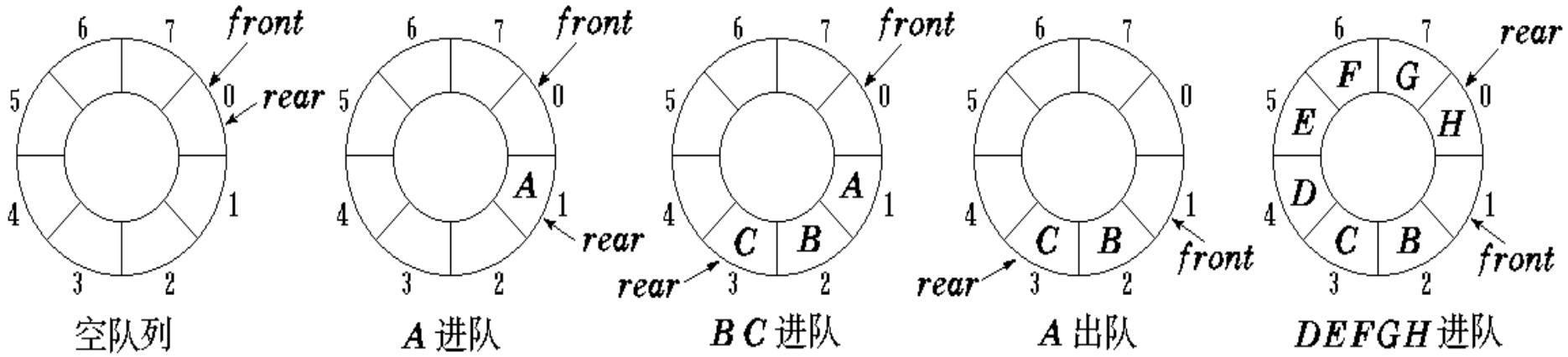
队头指针进**1:** *front* = (*front* + 1) % *maxSize*;
队尾指针进**1:** *rear* = (*rear* + 1) % *maxSize***;**

队列初始化：*front* = *rear* = **0;**

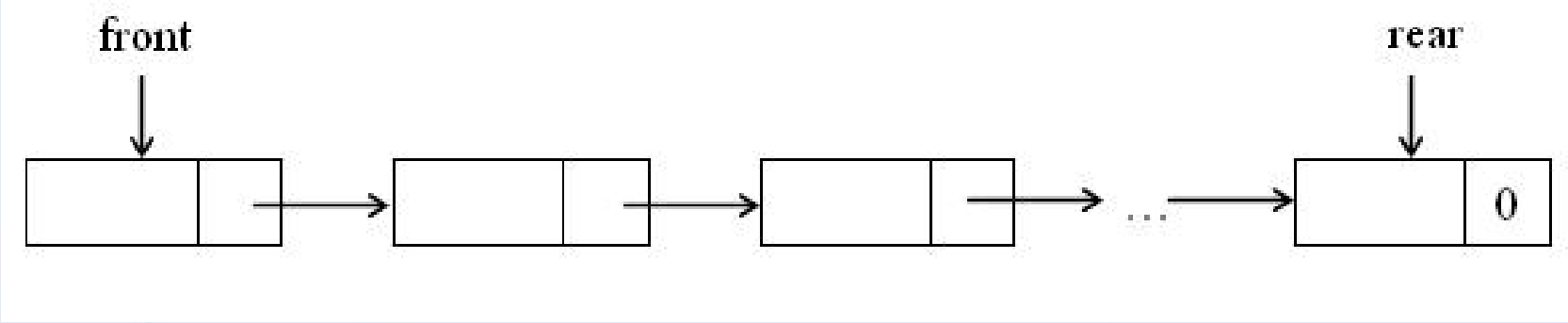
队空条件：*front* == *rear***;**

队满条件：(*rear* + 1) % *maxSize* == *front*

循环队列的进队和出队



链式队列：



链式队列的存储结构定义类似于链式栈。

用列表实现队列

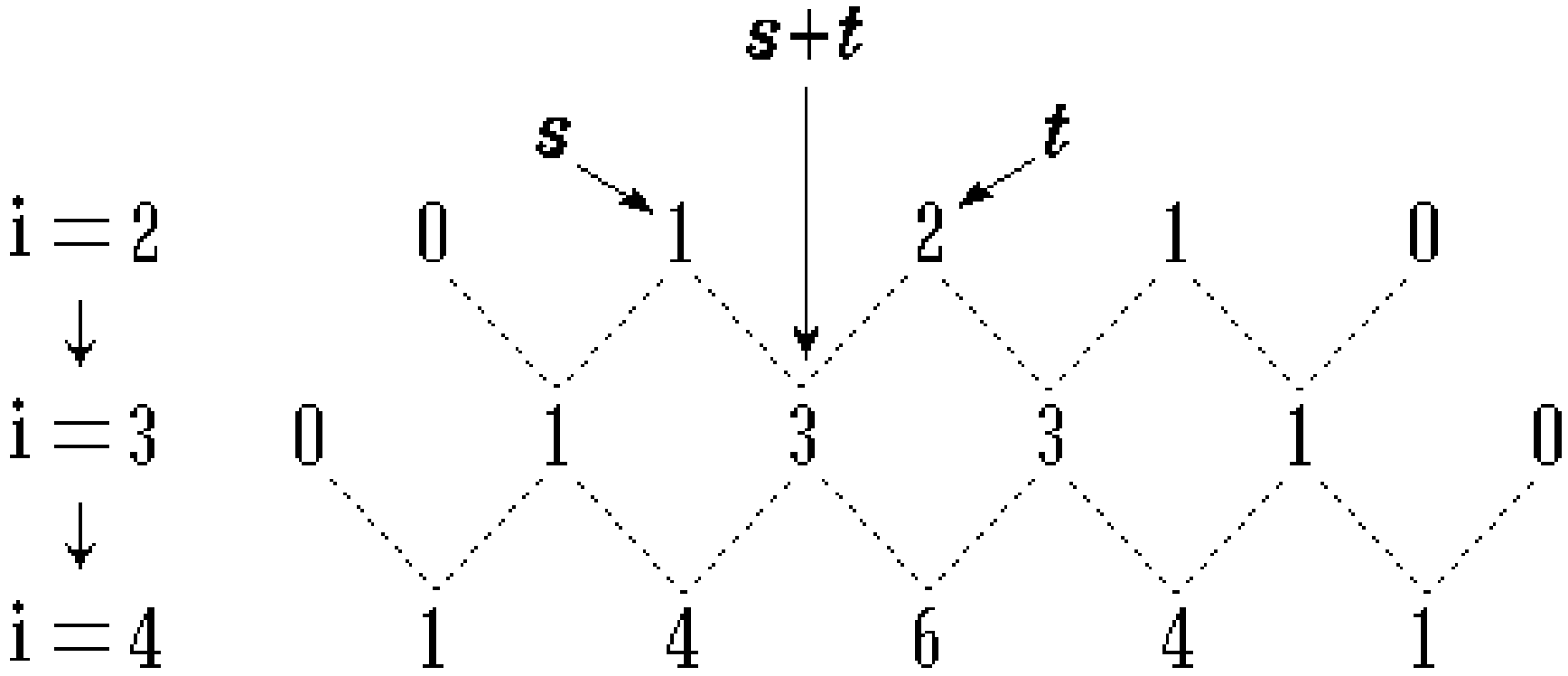
```
class Queue:
    def __init__(self): self.list=[]
    def Enqueue(self, x): self.list.append(x)
    def Dequeue(self) :
        if len(self.list)>0: return self.list.pop(0)
    def First(self):
        if len(self.list)>0: return self.list[0]
    def IsEmpty(self):
        if len(self.list)==0:
            return True
        else: return False
    def Output(self) :
        print “Head ->:”, ; print self.list
```

队列的应用举例
— 逐行打印二项展开式 $(a + b)^i$ 的系数

杨辉三角形 (Pascal's triangle)

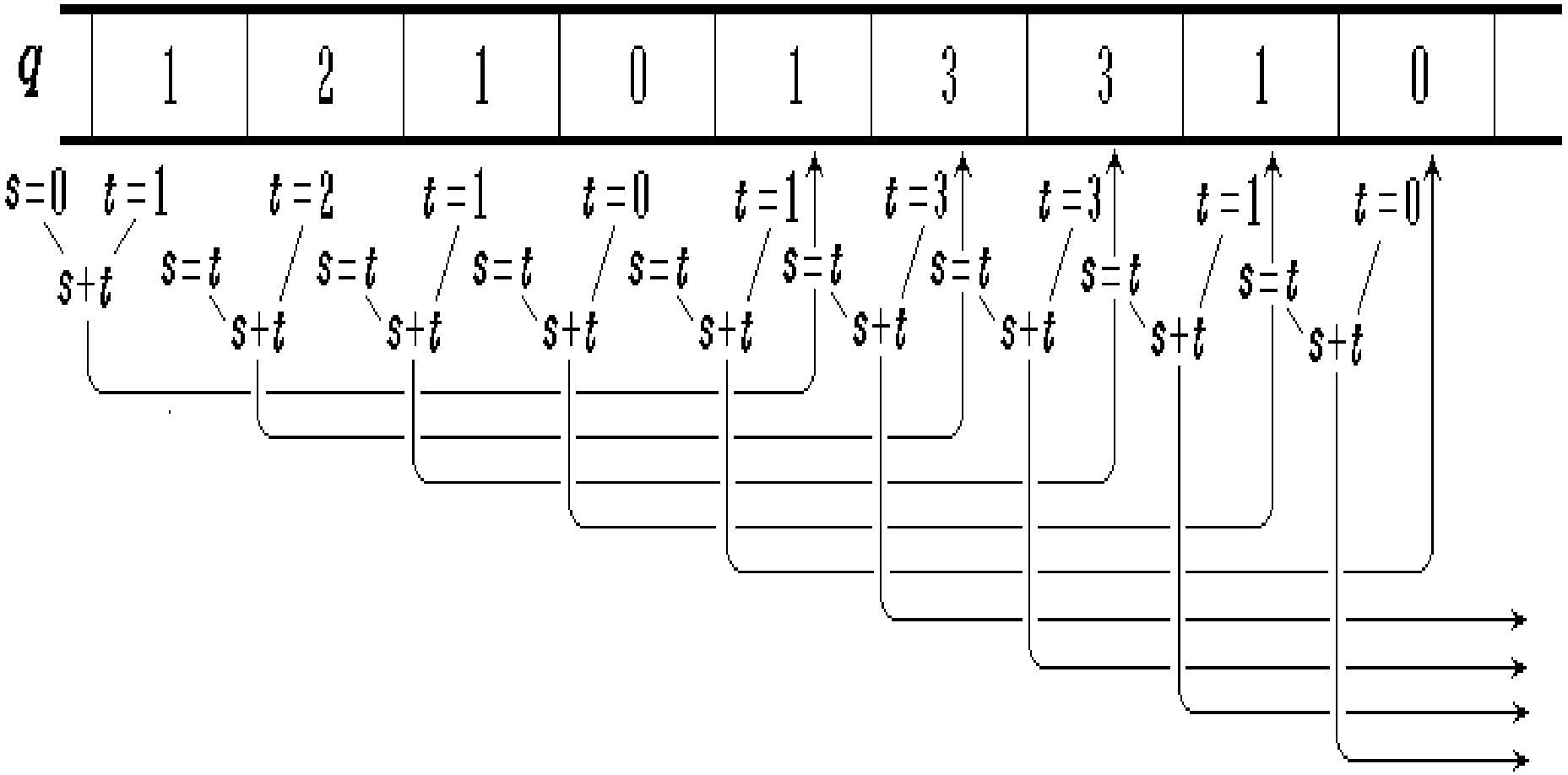
1 1	i = 1
1 2 1	2
1 3 3 1	3
1 4 6 4 1	4
1 5 10 10 5 1	5
1 6 15 20 15 6 1	6

分析第 *i* 行元素与第 *i*+1行元素的关系



目的是从前一行的数据可以计算下一行的数据

从第 *i* 行数据计算并存放第 *i*+1 行数据



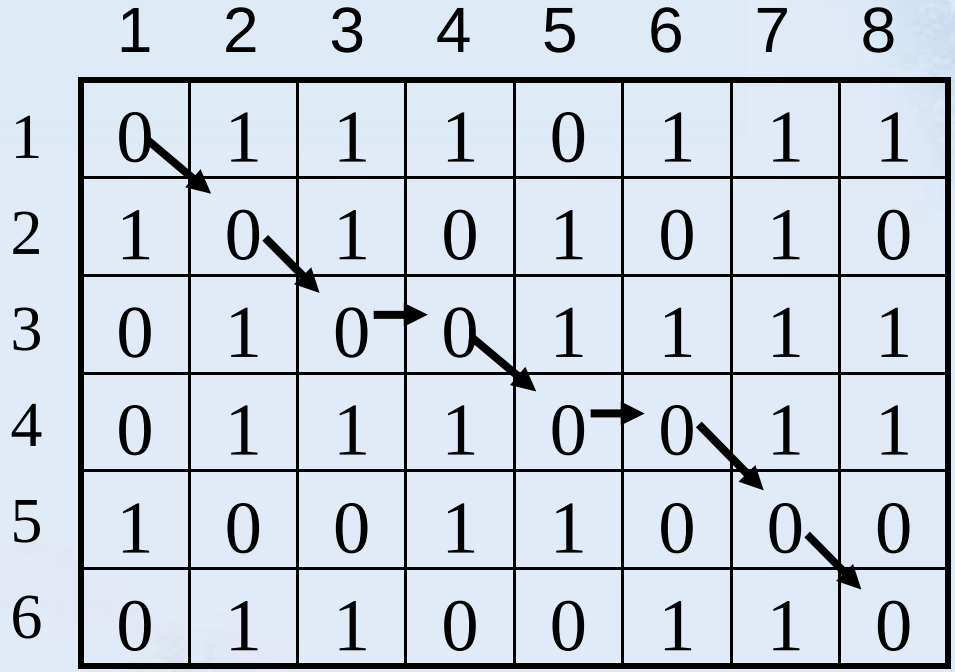
```
def  Yanghui( n ) :
    Q= Queue()                              #/队列初始化
    Q.Enqueue (1);  Q.Enqueue (1)
    s = 0
    for i in range(1, n+1):   #逐行计算
        print
        Q.Enqueue (0)
        for  j in range(1, i+3) :  #处理第i行
            t = Q.Dequeue( )
            Q.Enqueue( s+t )
            s = t
            if  j != i+2 : print s,  # 仅输出非0值
```

输出

```
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
1 10 45 120 210 252 210 120 45 10 1
```

队列的应用举例—求解迷宫问题

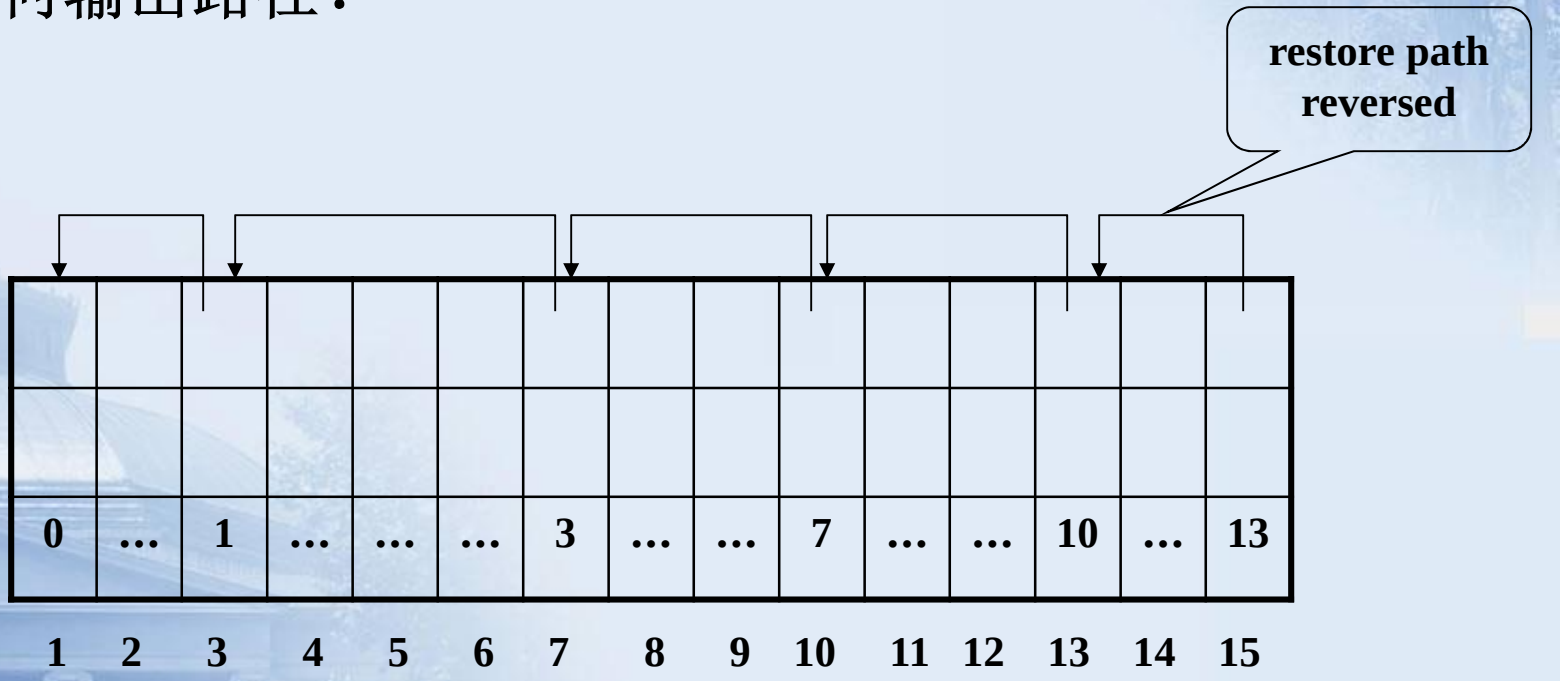
初始化队列并将 入口坐标及方向加入队列Q;
while Q is not empty :
    (i,j,pre) = 从队列中取出坐标和方向;
    for 当前位置的**8**个方向 :
        (g, h, p) = 下一步的坐标及当前位置的方向;
        if g == m and h == n :
            到达出口，输出路径，结束;
        if 可以移动到 **maze[g][h]**
            and 未经过**mark[g][h] :**
            mark[g][h] = 1
            将 **(g, h, p)** 加入队列;
输出“路径不存在”;



0	1	2	3	4	5	6	7	8
-1	0	1	1	2	2	3	……	
1	2	3	1	2	3	4	……	
1	2	3	3	4	4	1	……	

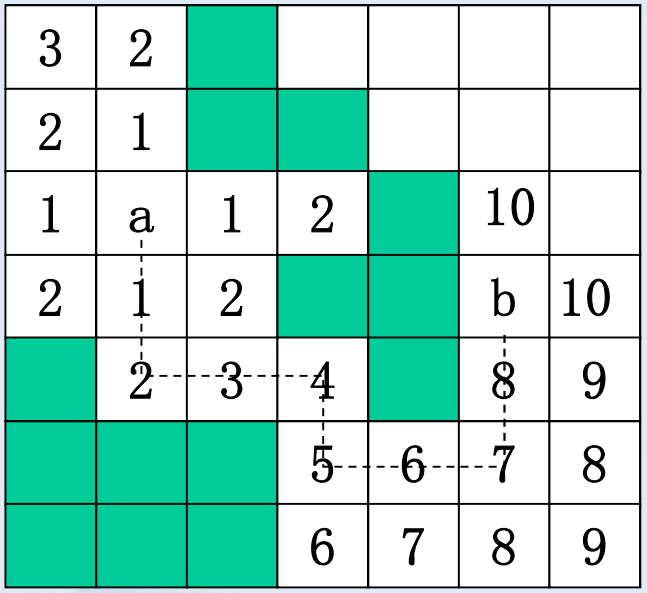
问题：

1）在此方案中有无回朔？
2）如何输出路径？



Depth first search: stack (need go back)
Width first search: queue (do not need go back)

队列的应用举例 — 电路布线



构造路径：从终点作为开始的检查点，搜索周围网格，检查其值是否比检查点的值小1，如是，则以此网格作为新的检查点，继续查找。

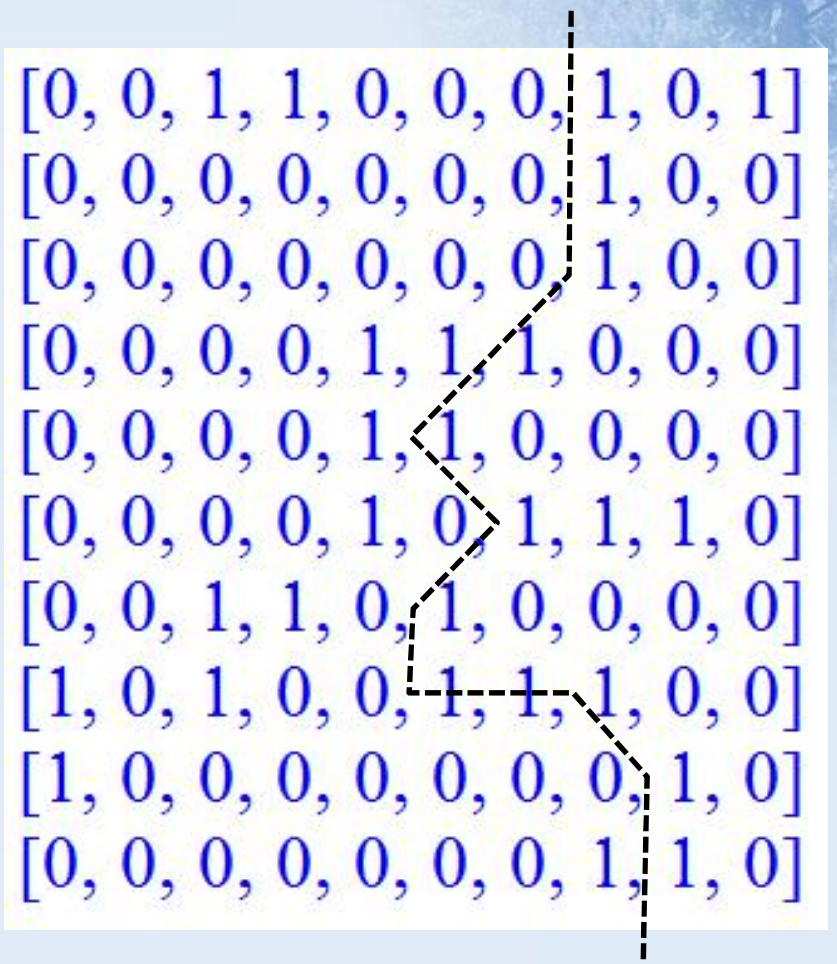
因为0表示可走线，1表示不可走线。为方便区别，起点以2表示。因此，路径的长度为终点数字-2。

```
def FindPath( A, a, b, move ) :
    n = len(A)
    A[a/n][a%n] = 2 #设置开始位置
    Q = Queue();  Q.Enqueue(0)
    while not Q.IsEmpty() :
        p = Q.Dequeue()
        i, j = p/n, p%n   # 一维向二维转换
        c = A[i][j]
        for  k in move :
            x, y = i + k[0], j + k[1]
            if  x<0 or x==n or y<0 or y==n or A[x][y] == 1 : continue
            if  x*n+y == b : # 检查是否到达结束位置
                A[x][y] = c+1
                return True
            if  A[x][y] == 0 :
                A[x][y] = c+1
                Q.Enqueue(x*n+y)
    return False
```

```
def Output(A, B, move) :
    n = len(A)
    path = []
    i, j = n-1, n-1
    while A[i][j] > 2 :
        for  k in move :
            x, y = i + k[0], j + k[1]
            if  x<0 or x==n or y<0 or y==n or A[x][y] < 2 : continue
            if  A[x][y] == A[i][j] - 1 :
                path.append((x, y))
                i, j = x, y
    for p in path : B[p[0]][p[1]] = 8
    B[0][0], B[n-1][n-1] = 8, 8
    for i in B : print i
```

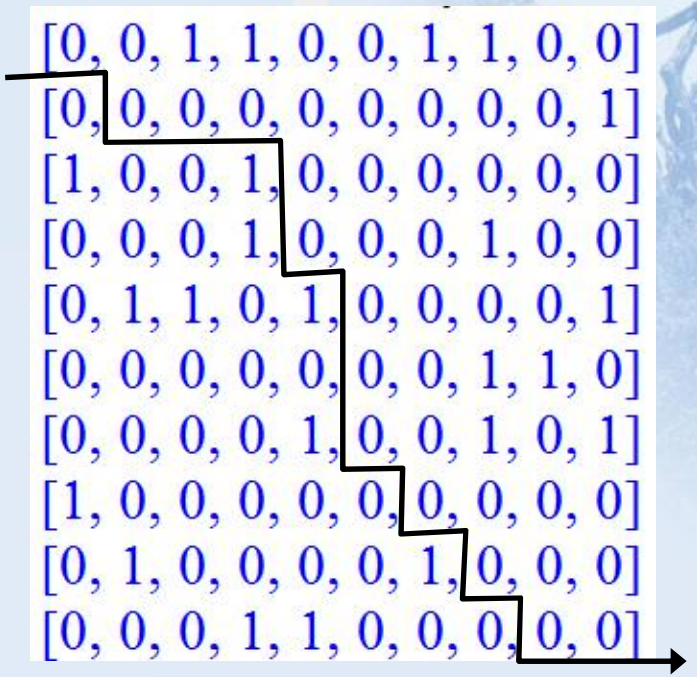
没有路的情况

[0, 0, 1, 1, 0, 0, 0, 1, 0, 1]
[0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 0, 1, 1, 1, 0, 0, 0]
[0, 0, 0, 0, 1, 1, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 1, 1, 1, 0]
[0, 0, 1, 1, 0, 1, 0, 0, 0, 0]
[1, 0, 1, 0, 0, 1, 1, 1, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 0, 0, 0, 1, 1, 0]



找到了路径

[0, 0, 1, 1, 0, 0, 1, 1, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
[1, 0, 0, 1, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 0, 0, 0, 1, 0, 0]
[0, 1, 1, 0, 1, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 0, 1, 1, 0]
[0, 0, 0, 0, 1, 0, 0, 1, 0, 1]
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 0, 0, 0, 0, 1, 0, 0, 0]
[0, 0, 0, 1, 1, 0, 0, 0, 0, 0]



[8, 8, 1, 1, 0, 0, 1, 1, 0, 0]
[0, 8, 8, 8, 8, 0, 0, 0, 0, 1]
[1, 0, 0, 1, 8, 0, 0, 0, 0, 0]
[0, 0, 0, 1, 8, 8, 0, 1, 0, 0]
[0, 1, 1, 0, 1, 8, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 8, 0, 1, 1, 0]
[0, 0, 0, 0, 1, 8, 8, 1, 0, 1]
[1, 0, 0, 0, 0, 0, 8, 8, 0, 0]
[0, 1, 0, 0, 0, 0, 1, 8, 8, 0]
[0, 0, 0, 1, 1, 0, 0, 0, 8, 8]

初始化及函数调用

```
n = 8
A = [ [0 for i in range(0, n)] for j in range(0, n)]
B = [ [0 for i in range(0, n)] for j in range(0, n)]
move = [(0,1),(1,0),(0,-1),(-1,0)]
for i in range(0, n) :  # A、B初始值
    for j in range(0, n) :
        A[i][j] = int(random.uniform(0, 1.4))
        B[i][j] = A[i][j]
begin, end = 0, n*n-1   # 确定起点和终点
A[begin/n][begin%n], A[end//n][end%n] = 0, 0
for i in A : print i
print
if  FindPath(A, begin, end, move) == True :   # 查找路径
    Output(A, B, move)  # 输出路径
else :
    print 'No Ptath existed! '
```

分析：

1、搜索过程（完成网格编号的过程）的时间开销为**O(m^2)**（或**O(m×n)**）；

2、重构路径过程的时间开销为**O(PathLen)**，最坏情况下为**O(m)**（或**O(m+n)**）；

2、所以，算法的总的时间复杂度为**O(m^2)**（或**O(m×n)**） 。

优先级队列 (Priority Queue)

优先级队列　是不同于先进先出队列的另一种队列。每次从队列中取出的是具有最高优先权的元素。例如下表：任务的优先权及执行顺序的关系

任务编号	1	2	3	4	5
优先权	20	0	40	30	10
执行顺序	3	1	5	4	2

数字越小，优先权越高

优先级队列的存储表示和实现方式

存储表示：顺序存储方式；链接存储方式。

实现方式：1、队尾入队，通过搜索确定出队元素。

2、队头出队，元素入队时调整排列次序。

双端队列(Double-ended Queue)

- 队列的两端各提供3个存取函数：读、插入、删除
- 两种实现方式：顺序、链接

【例】利用双栈（设为S1和S2）模拟一个队列，写出入队和出队的算法。

入队算法：

```
def enqueue( a ) :
    if  not S1.IsFull( ) :
        S1.push(a)
   else : print "overflow"
```

出队算法：

```
def  dequeue( ) :
   if  not S2.IsEmpty( ) :
       return S2.pop( )
   elif  not S1.IsEmpty( )  :
       while not S1.IsEmpty( ) :
            S2.push(S1.pop( ))
       return (S2.pop( )
   else :
       print "Infeasible"
```