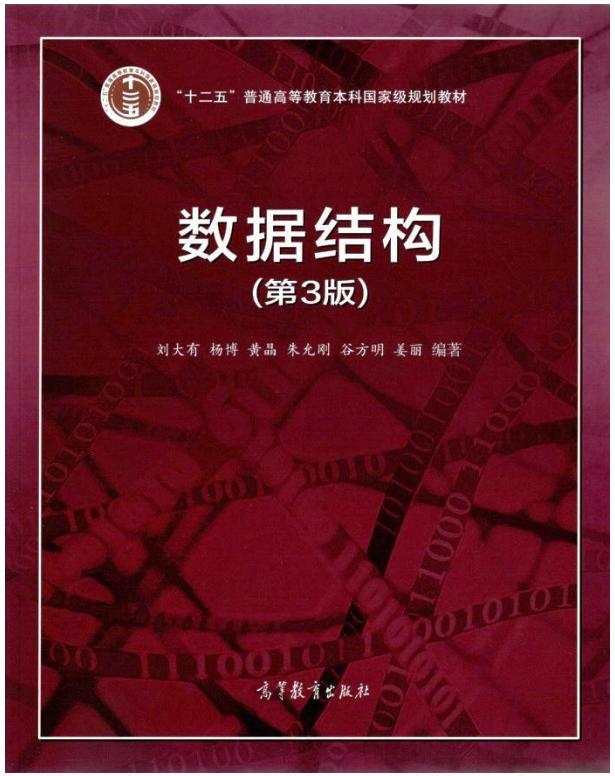


线性表的链接存储

- 单链表
- 循环链表
- 双向链表
- 静态链表
- 链表的双指针技巧
- 复杂链表的拷贝
- 跳跃表

数据之法
结构之美
算法之道

zhuyungang@jlu.edu.cn

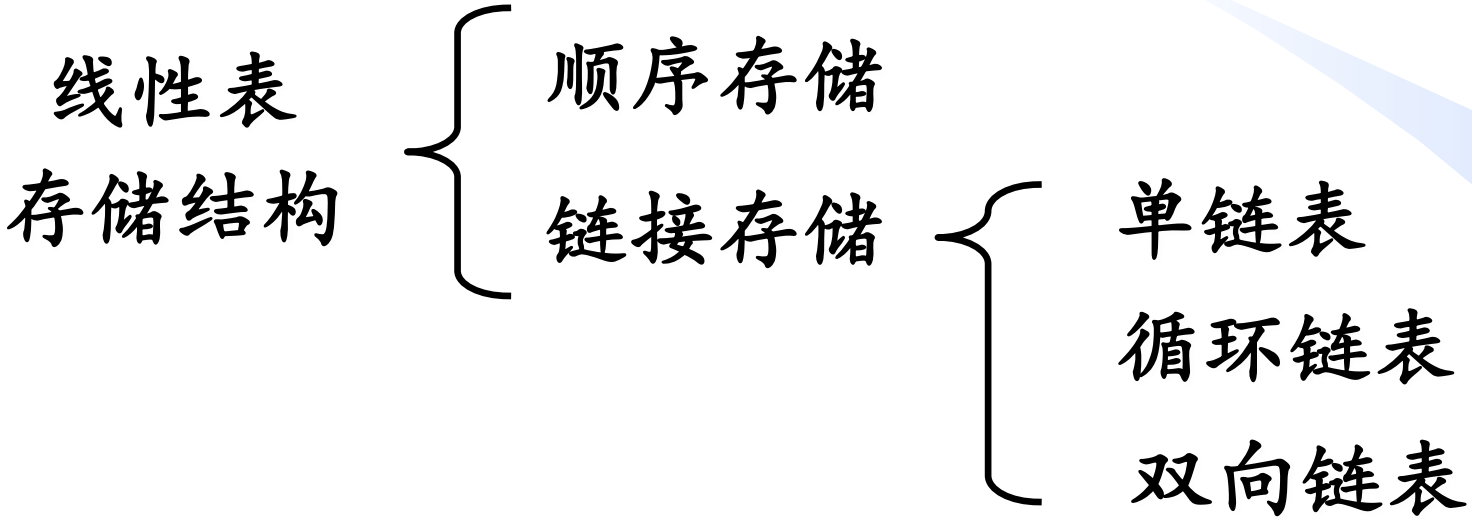
Last updated on 2022.9

流水不争先，争滔滔不绝。

慕课自学内容

- 线性表的定义和基本操作
- 线性表的顺序存储结构
- 堆栈的定义和主要操作
- 顺序栈
- 链式栈
- 顺序栈与链式栈的比较
- 队列的定义和主要操作
- 顺序队列
- 链式队列

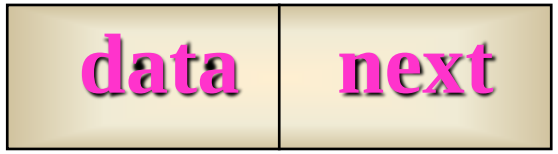
线性表的存储结构



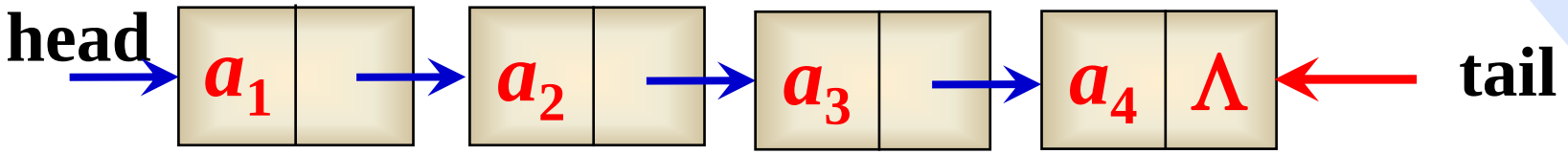
■ **链接存储**：用**任意**一组存储单元存储线性表，一个存储单元除包含结点数据字段的值，还必须存放其逻辑相邻结点（前驱或后继结点）的地址信息，即指针字段。

线性表的链接存储结构—单链表

- 单链表的**结点结构**：



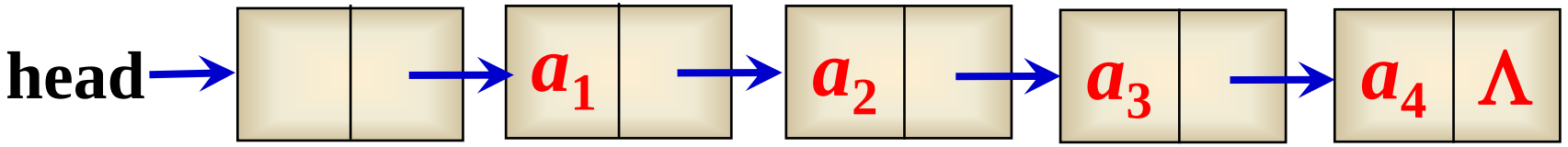
单链表的定义：每个结点只含**一个链接域**的链表叫单链表。



◆链表的第一个结点被称为头结点(也称为表头)，指向头结点的指针被称为头指针(head).

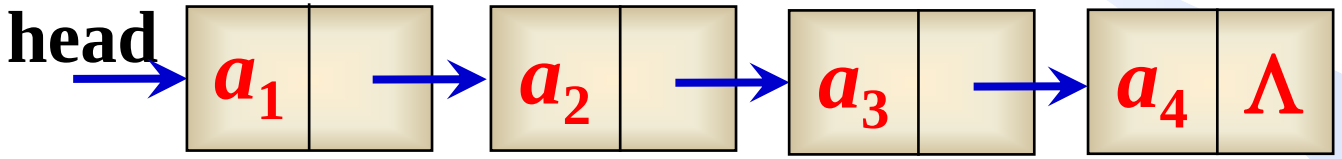
◆链表的最后一个结点被称为尾结点(也称为表尾)，指向尾结点的指针被称为尾指针(tail).

- 为了对表头结点插入、删除等操作的方便，通常在表的前端增加一个特殊的表头结点，称其为<u>哨位（哨兵）结点</u>。
- 哨位结点不被看作表中的实际结点，我们在讨论链表中第k个结点时均指第k个实际的表结点。
- 表的长度：非哨位结点的个数。若表中只有哨位结点，则称其为空链表，即表长度为0

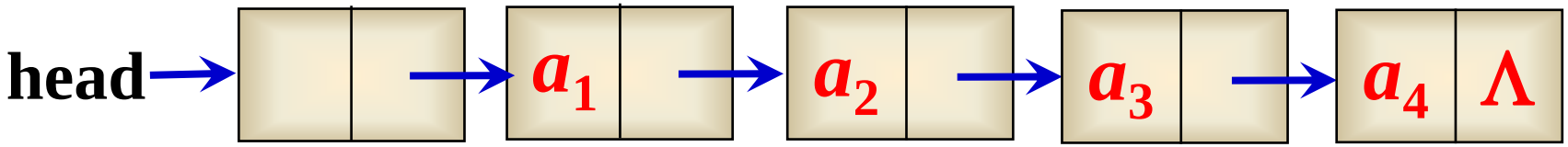


- 哨位结点作用:简化边界条件的处理。

- 如果没有哨位结点……



- 有了哨位结点后……



哨位结点

3、单链表主要操作举例

```
算法 Find (head, k. p) // 链表第k个结点的地址赋给p
  IF  k < 0 THEN RETURN Λ.  //输入的k位置不合法
  //初始化
   p←head .  i←0.  //令指针p指向哨位结点，计数器初始值为0
  //找第k个结点
  WHILE p≠Λ AND i<k DO // 若找到第k个结点或已到达
  ( p←next(p) .  i←i+1. )      // 表尾，则循环终止
    RETURN p.  // p≠Λ则p即为所求，返回p；p=Λ 表示链表
                //长度不足k（无第k个结点），返回Λ;
```

算法 Search (*head, item . p*) // 在链表中查找字段值为*item*的结点并返回其指针

```
//初始化
p←next(head) .  // 令指针p指向第1个结点
//遍历
WHILE  p ≠ Λ AND data(p) ≠ item  DO
      p←next(p) .  // 扫描下一个结点
 RETURN p .
```

```
算法Delete (head, k) // 删除链表中第k个结点
  IF  k < 1 THEN RETURN.   // 输入k不合法，不能删哨位
  p ←Find (head, k-1) .        //找第k–1结点，由p指向
  IF p = Λ OR next(p) = Λ THEN
      RETURN.                        //无第k-1个结点或只有k-1个结点
  //删除第k个结点
```

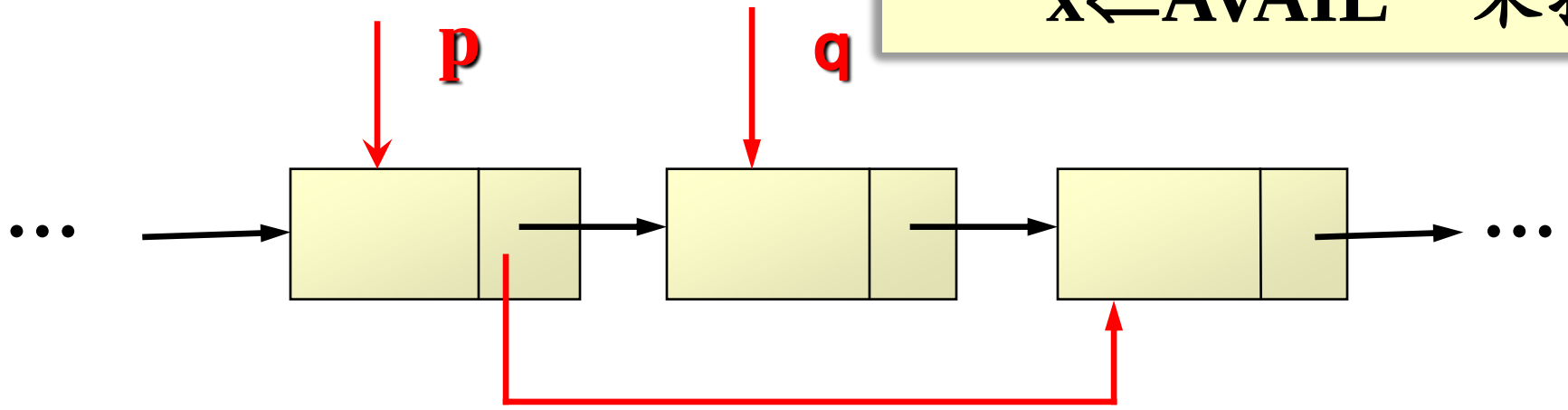
在链表中，插入或删除一个结点，只需改变一个或两个相关结点的指针，不对其它结点产生影响。

q ← next (p).

next (p) ← next (q) .

AVAIL ⇐ q .

- 删除：



- 在ADL语言中，释放不用的空间则用语句“AVAIL⇐x”来描述，其中AVAIL是一个可利用空间表。
- 申请新存储空间的操作用语句“x⇐AVAIL”来描述。

```
算法Delete (head, k) // 删除链表中第k个结点
  IF  k < 1 THEN RETURN.   // 输入k不合法，不能删哨位
  p ←Find (head, k-1) .        //找第k–1结点，由p指向
  IF p = Λ OR next(p) = Λ THEN
     RETURN.                    //无第k-1个结点或只有k-1个结点
  //删除第k个结点
  q←next(p).
  next(p) ←next(q).            // 修改p的next指针
  AVAIL⇐q.                     //释放q存储空间
```

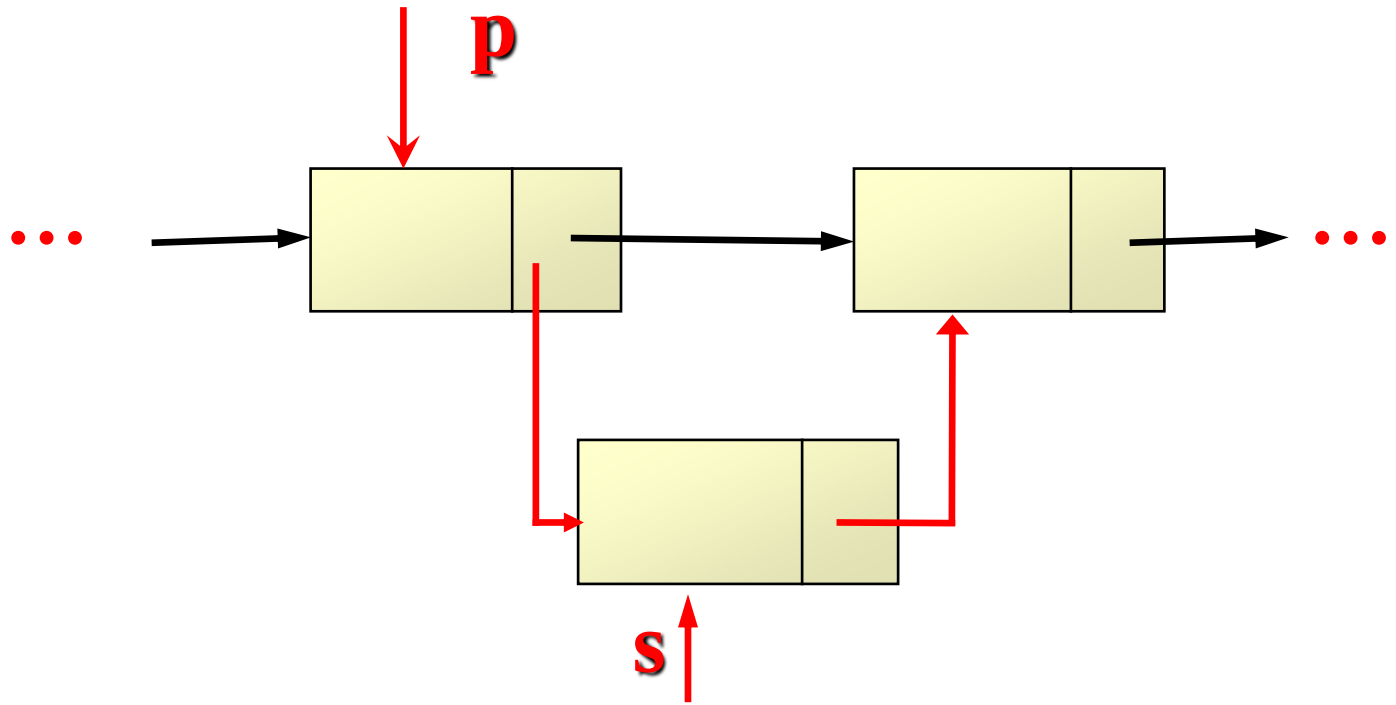
```
算法Insert (head, k, item )
  // 在链表head中 第k个结点后插入字段值为item的结点
  IF k < 0 THEN RETURN.      //插入不合法
  p ← Find (head, k) .       //找第k结点
  IF p= Λ THEN RETURN.       //无第k个结点
  //插入
 s⇐AVAIL. data(s) ←item.     // 生成新结点s
```

在链表中，插入或删除一个结点，只需改变一个或两个相关结点的指针，不对其它结点产生影响。

- 插入：

next(s) ← next(p)

next(p) ← s



算法Insert (head, *k*, *item*)

// 在链表*head*中 第*k*个结点后插入字段值为*item*的结点

IF *k* < 0 THEN RETURN.　　//插入不合法

p ← Find (head, *k*) .　　//找第*k*结点

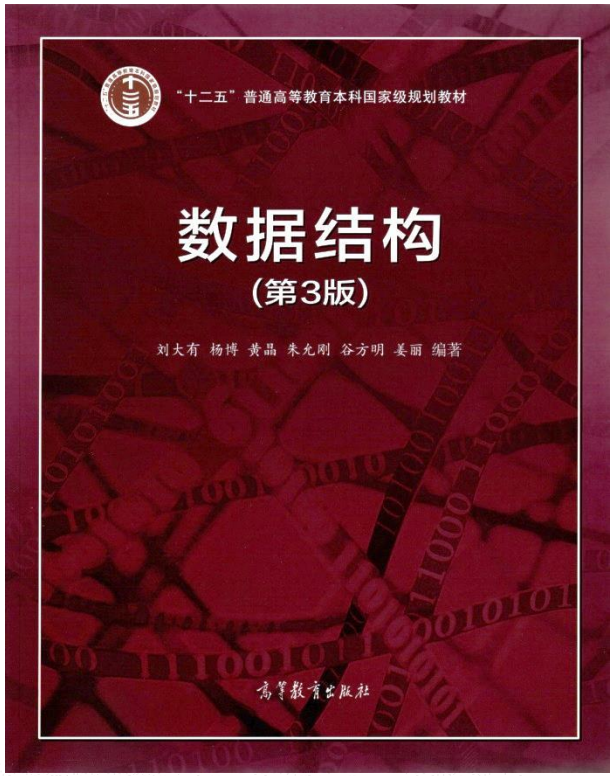
IF *p*= Λ THEN RETURN.　　//无第*k*个结点

//插入

s⇐AVAIL. data(s) ←*item*.　　// 生成新结点s

next(s) ← next(p).　　//s的next指针指向*p*的后继结点

next(p) ← s.　　// 修改*p*的next指针，令其指向*s*



线性表的链接存储

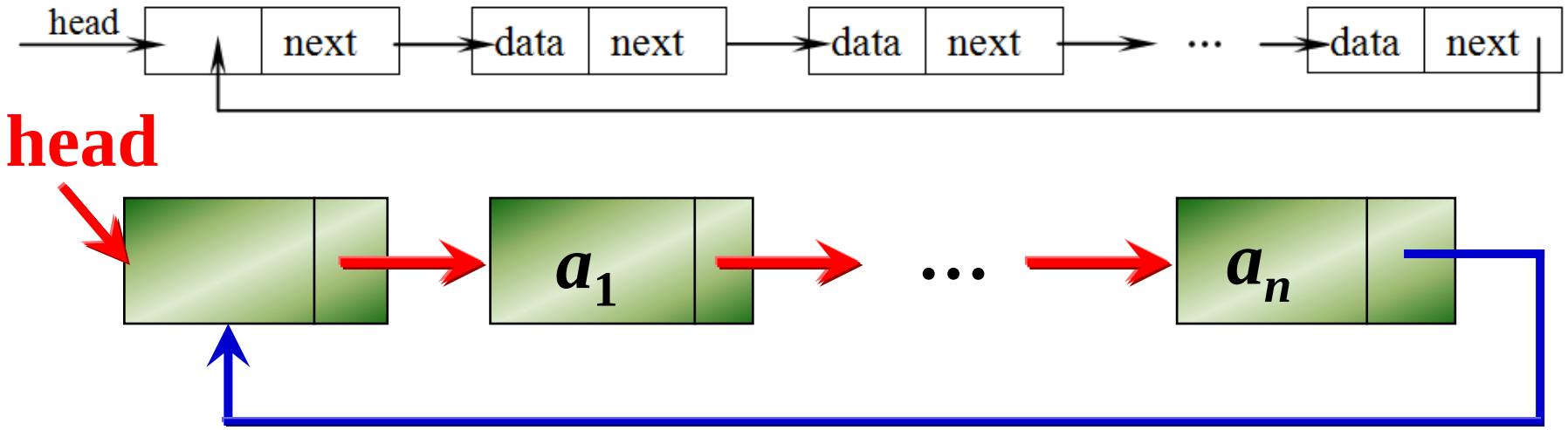
- 单链表
- **循环链表**
- 双向链表
- 静态链表
- 链表的双指针技巧
- 复杂链表的拷贝
- 跳跃表

数据之法
结构之美
算法之道

zhuyungang@jlu.edu.cn

Last updated on 2022.9

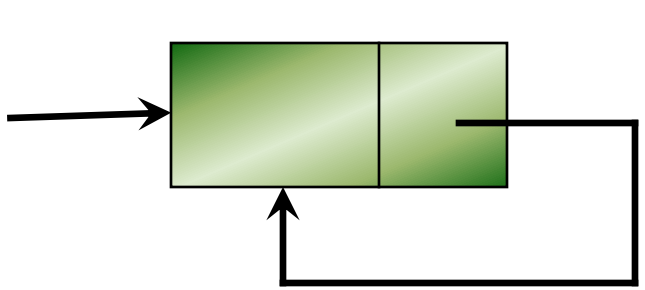
- 把链接结构“循环化”，即把表尾结点的next域存放指向哨位结点的指针，而不是存放空指针NULL（即Λ），这样的单链表被称为循环链表。
- 循环链表使我们可从链表的任何位置开始，访问链表中的任一结点。



判断空表的的条件:

单链表： **next(head) = Λ**

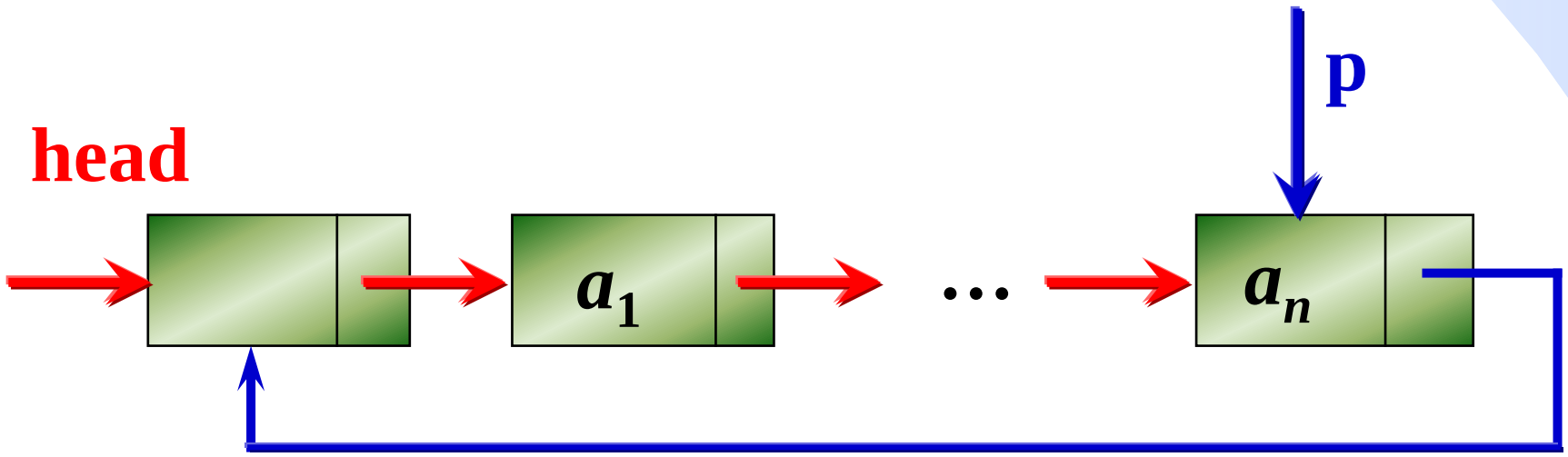
循环链表： **next(head) = head**



判断表尾的条件：

单链表：next (p) = Λ

循环链表：next (p) = head



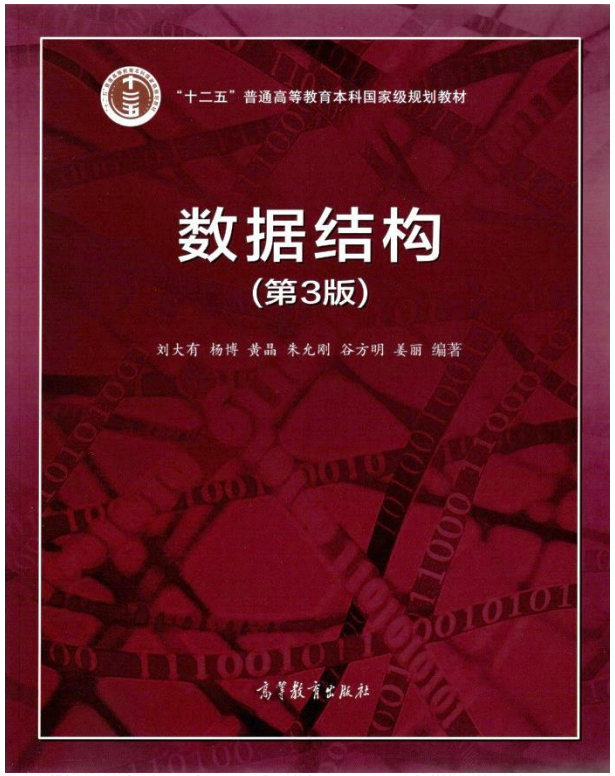
约瑟夫问题

- 著名犹太历史学家 Josephus。
- 罗马人占领乔塔帕特，39 个犹太人与Josephus及其朋友躲到一个洞中。39个犹太人宁愿死也不要被敌人抓到，于是决定了一个自杀方式。 41个人排成一个圆圈，由第1个人开始报数，每报数到第3人该人就必须自杀，然后再由下一个重新报数，直到所有人都自杀身亡为止。
- 然而Josephus 及其朋友并不想遵从。他将朋友与自己安排在第16个与第31个位置。

简化版本

- N个人围成一圈，从第一个开始报数，第M个人出列，不断循环，按顺序输出出列人的编号。例如N=6，M=5，出列人的序号为5，4，6，2，3。最后剩下1号。

- 留作作业



线性表的链接存储

- 单链表
- 循环链表
- **双向链表**
- 静态链表
- 链表的双指针技巧
- 复杂链表的拷贝
- 跳跃表

数据之法
结构之美
算法之道

zhuyungang@jlu.edu.cn

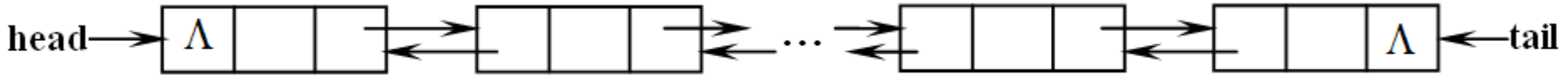
Last updated on 2022.9

- 双向链表（Double-Linked List）

每个结点有两个指针域

左指针指向其前驱，右指针指向其后继；

优点：方便找结点的前驱。



双向链表结点的结构体定义

```
struct DLNode{
    int data;
    DLNode *left, *right ;
};
```

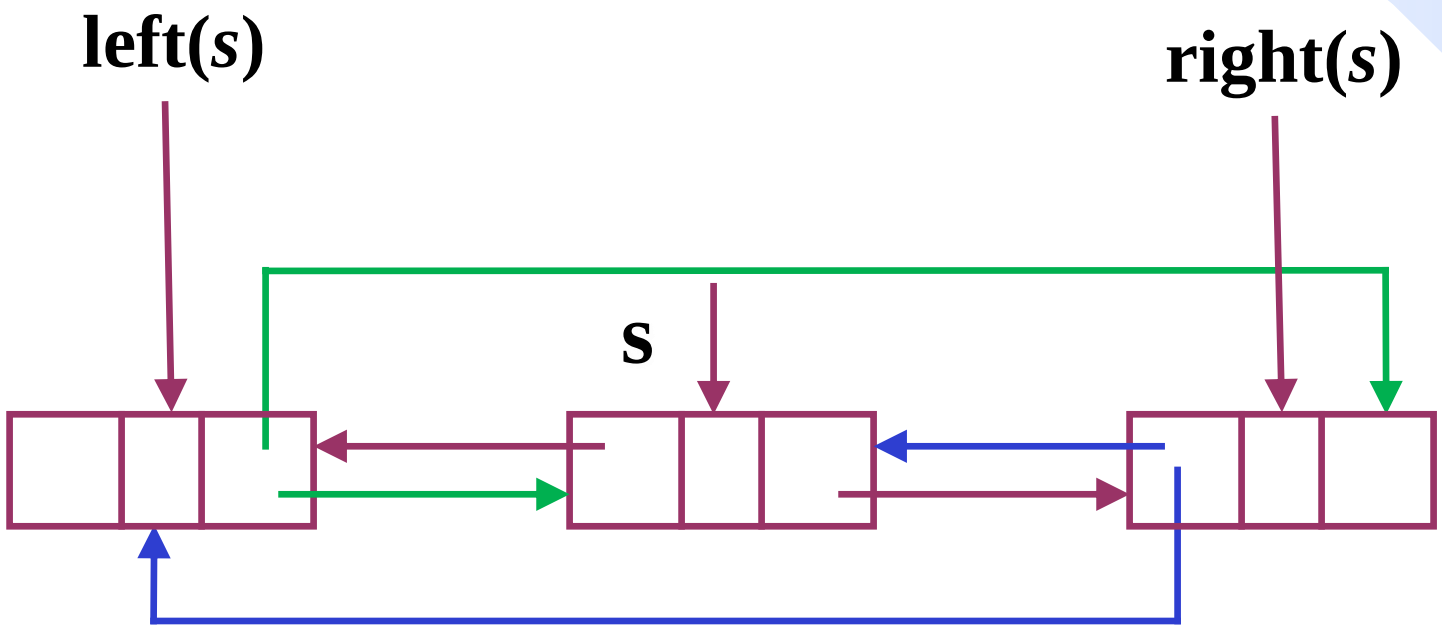
删除结点s

right(left(s)) ← right(s).

left(right(s)) ← left(s).

AVAIL⇐ s.

常规情况



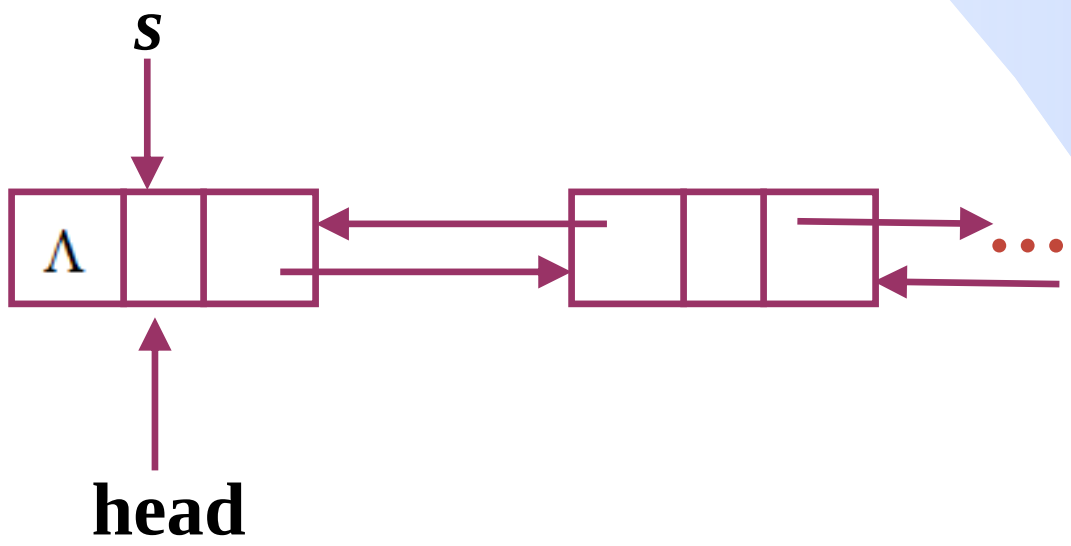
删除结点s

right(left(s)) ← right(s).

left(right(s)) ← left(s).

AVAIL⇐ s.

如果s是第1个结点?



删除结点s

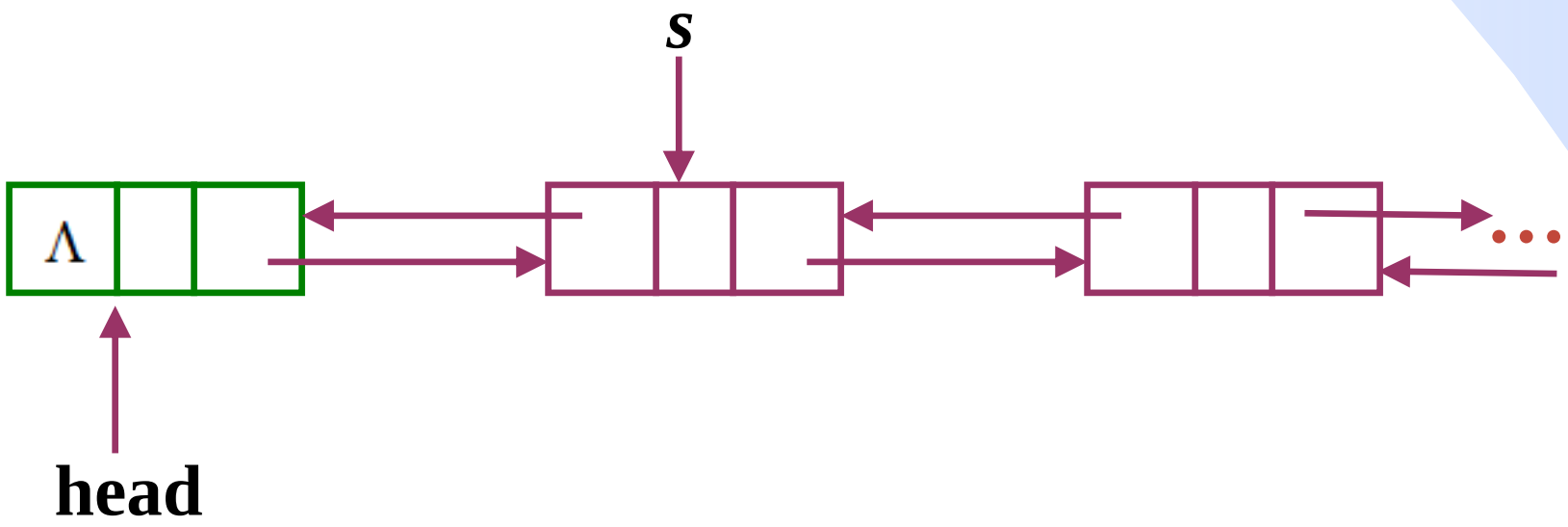
right(left(s)) ← right(s).

left(right(s)) ← left(s).

AVAIL⇐ s.

引入表头
哨位结点

如果s是第1个结点?



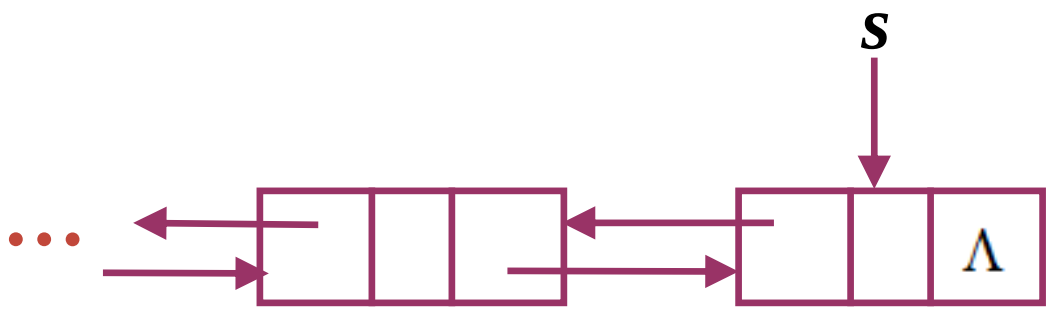
删除结点s

right(left(s)) ← right(s).

left(right(s)) ← left(s).

AVAIL⇐ s.

如果s是最后1个结点?



删除结点s

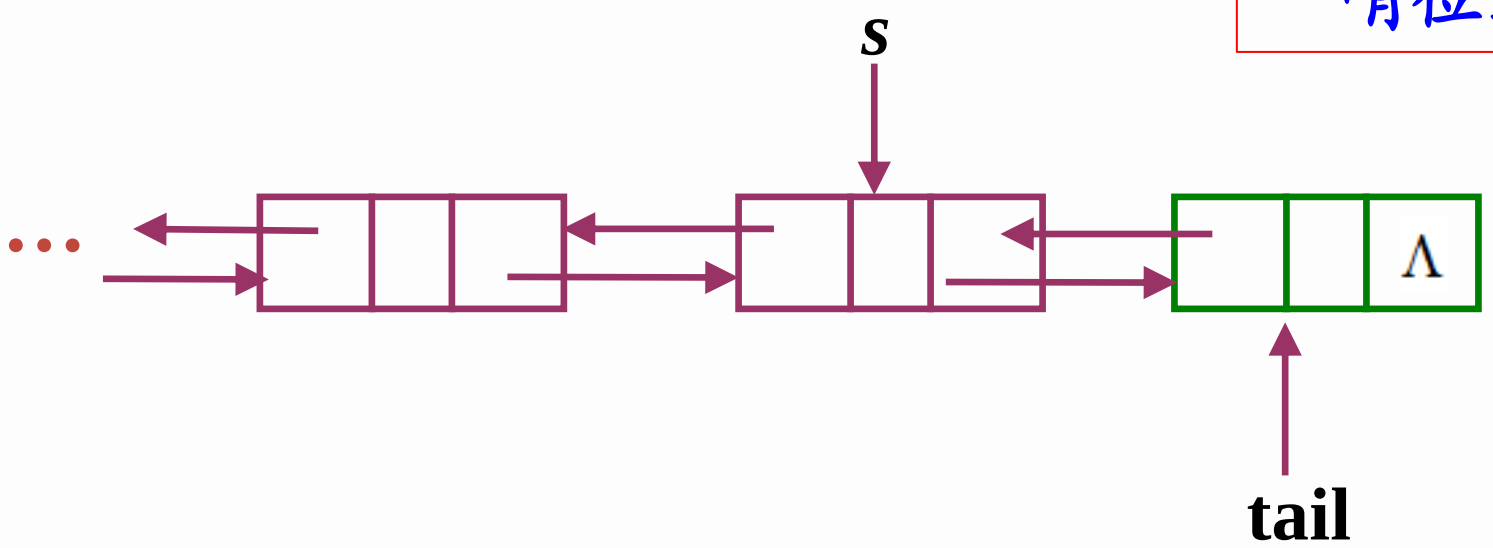
right(left(s)) ← right(s).

left(right(s)) ← left(s).

AVAIL⇐ s.

如果s是最后1个结点?

表尾引入
哨位结点



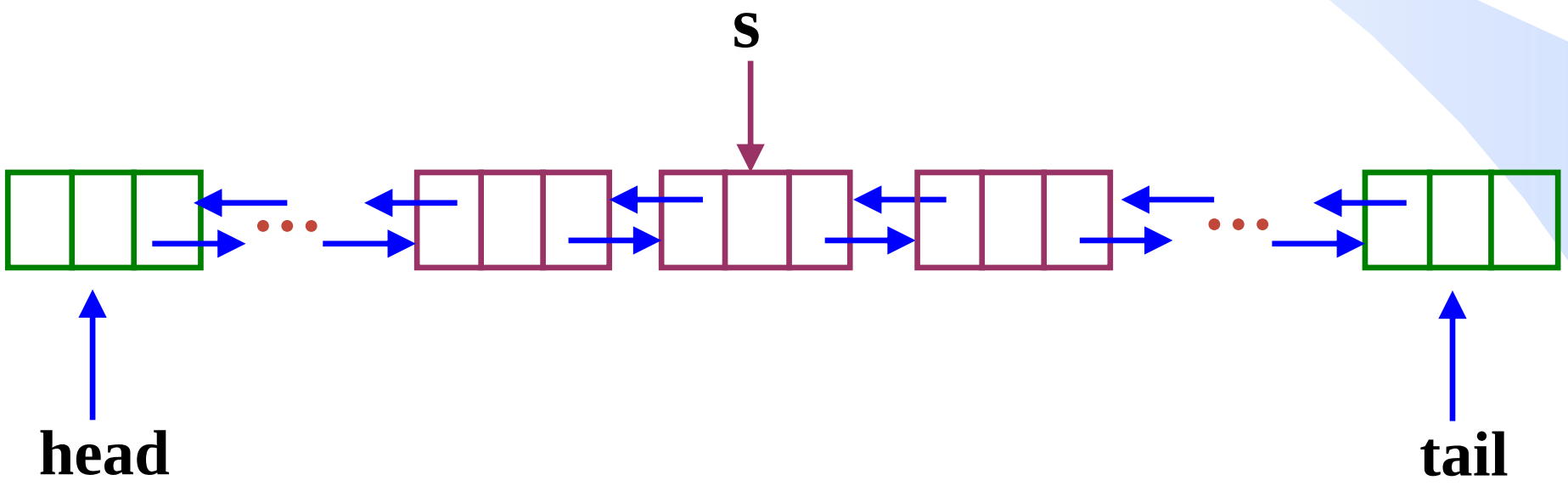
算法DeleteNode(*head, tail, s.*)

// 删除双向链表（带双哨位结点）中的非哨位结点*s*

right(left(*s*))← right(*s*).

left(right(*s*))← left(*s*).

AVAIL⇐ *s*.



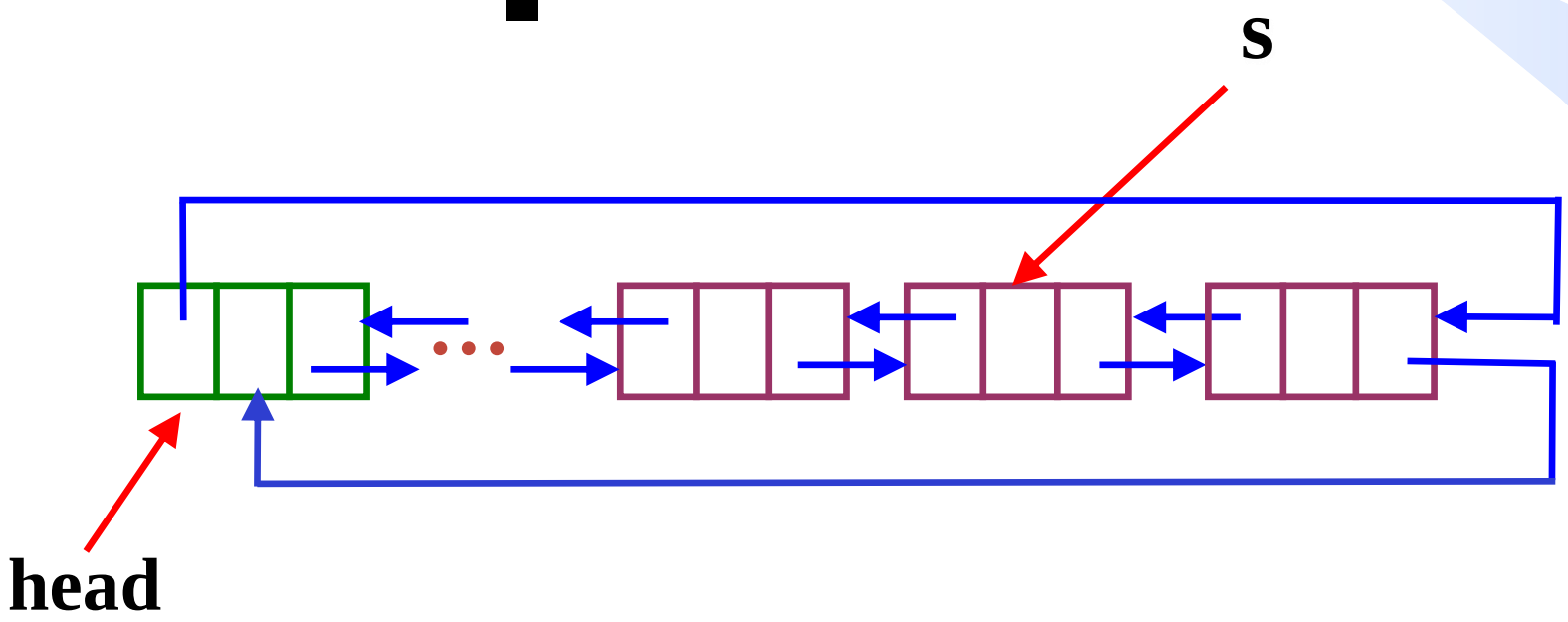
算法DeleteNode(*head, s. head, tail*)

// 删除双向循环链表（带哨位结点）中的非哨位结点*s*

right(left(*s*))← right (*s*).

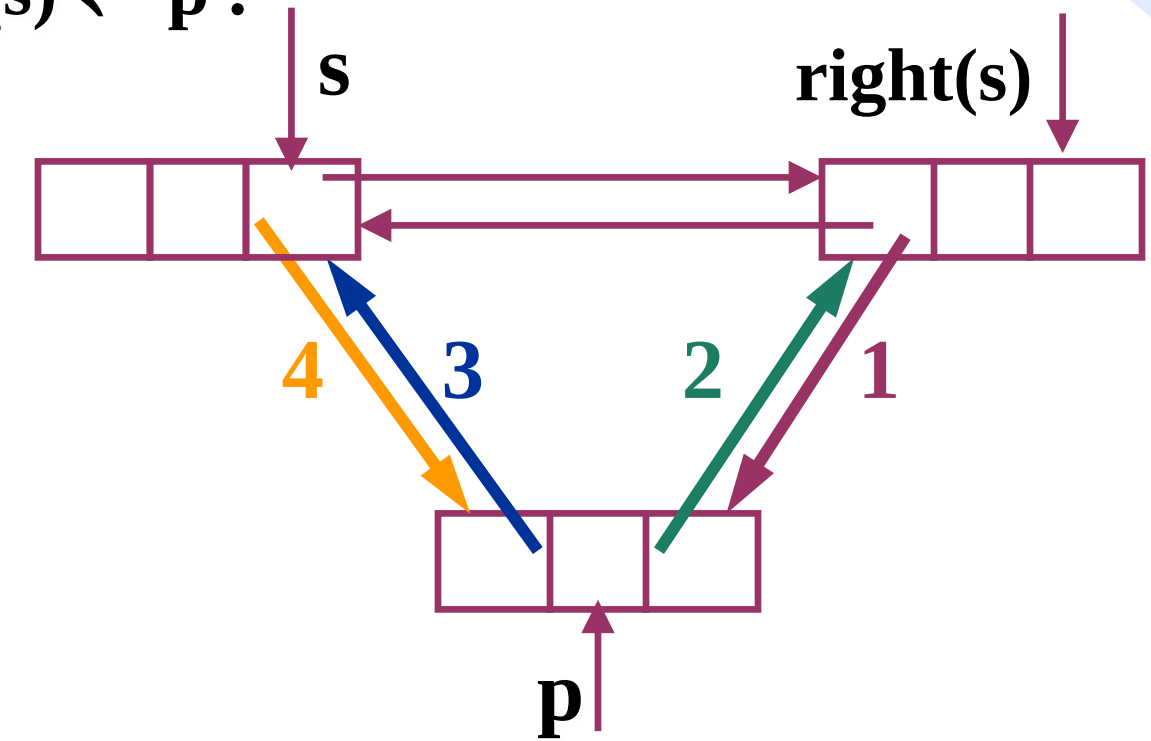
left(right(*s*))← left(*s*).

AVAIL⇐ *s*.



在带哨位结点的双向循环链表中的结点s之后插入结点 p

1. left (right (s))←p.
2. right (p) ← right (s).
3. left (p) ← s.
4. right (s) ← p .



算法DLInsert(*head, s , p. head, tail*)

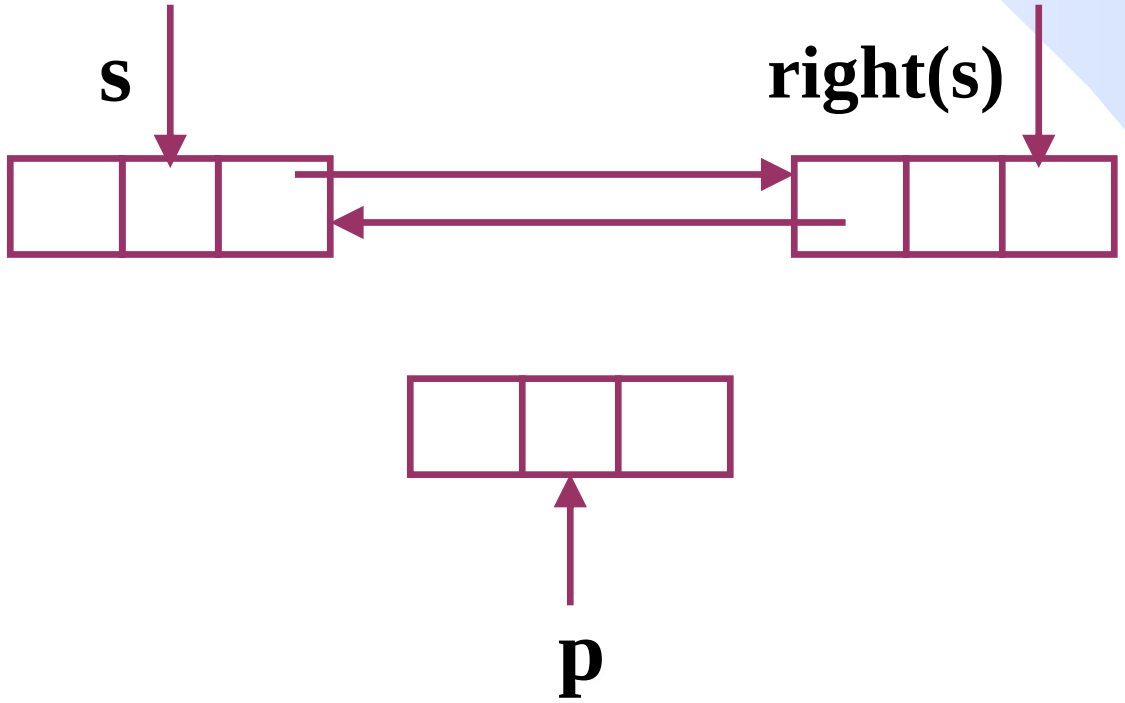
//在带哨位结点的双向循环链表中结点s的右边插入新结点p

right (*p*) ← right (*s*).

left (*p*) ← *s*.

left (right (*s*)) ← *p* .

right (*s*) ← *p* .



顺序存储和链式存储的比较

1、空间效率的比较

- 顺序表所占用的空间来自于申请的数组空间，数组大小是事先确定的，当表中的元素较少时，顺序表中的很多空间处于闲置状态，造成了空间的浪费；
- 链表所占用的空间是根据需要动态申请的，不存在空间浪费问题，但链表需要在每个结点上附加一个指针，从而产生额外开销。

2、时间复杂性的比较

- 线性表的基本操作是查找、插入和删除。

	基于下标的查找	插入删除
顺序表	O(1) 按下标直接查找	O(n) 需要移动若干元素
链　表	O(n) 从表头开始遍历链表	O(1) 只需修改几个指针值

- 当线性表经常需要进行插入、删除操作时，链表的时间复杂性较小，效率较高；
- 当线性表经常需要基于下标的查找，且查找操作比插入删除操作频繁的情况下，则顺序表的效率较高。

链表编程时留意边界条件处理

以下情况代码是否能正常工作：

- 链表为空时
- 链表只包含一个结点时
- 在处理头结点和尾结点时

链表应用场景举例

子弹链表
敌机链表

- 本方发射子弹：子弹链表插入新结点
- 子弹飞出边界、打中敌机：删除子弹结点
- 敌机飞出边界、被子弹打中：删除敌机结点
- 随机生成敌机：敌机链表插入新结点

```
struct Node{
    int x, y;
    Node *next;
    ……
};
Node *pBullet, *pEnemy;
```

链表应用场景举例

核心逻辑：

```
WHILE pBullet ≠Λ DO
(    WHILE pEnemy ≠Λ DO
      (     IF pBullet和pEnemy相遇 THEN
            (  //子弹打中敌机
                 删除pBullet所指结点.
                 删除pEnemy所指结点.
                 BREAK.
             )
            pEnemy← next(pEnemy).
       )
      pBullet ← next(pBullet).
)
```

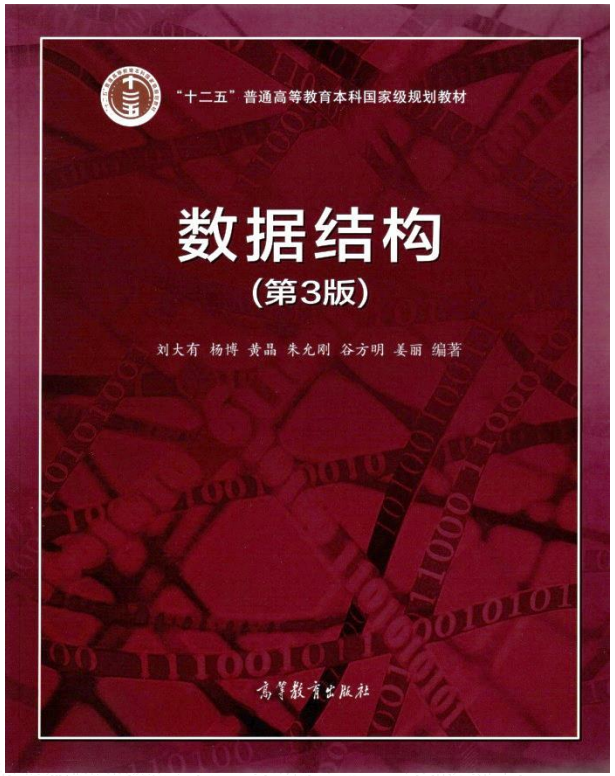
链表应用场景举例

```
...... ......
typedef struct LOS_DL_LIST {//双向循环链表，鸿蒙内核最重要结构体之一
    struct LOS_DL_LIST *pstPrev; /**< Current node's pointer to the previous node *///前驱结点
    struct LOS_DL_LIST *pstNext; /**< Current node's pointer to the next node *///后继结点
} LOS_DL_LIST;

//将指定结点初始化为双向链表结点
LITE_OS_SEC_ALW_INLINE STATIC INLINE VOID LOS_ListInit(LOS_DL_LIST *list)
{
    list->pstNext = list;
    list->pstPrev = list;
}

//将指定结点挂到双向链表头部
LITE_OS_SEC_ALW_INLINE STATIC INLINE VOID LOS_ListAdd(
{
    node->pstNext = list->pstNext;
    node->pstPrev = list;
    list->pstNext->pstPrev = node;
    list->pstNext = node;
}
//将指定结点从链表中删除,自己把自己摘掉
LITE_OS_SEC_ALW_INLINE STATIC INLINE VOID LOS_Li
{
    node->pstNext->pstPrev = node->pstPrev;
    node->pstPrev->pstNext = node->pstNext;
    node->pstNext = NULL;
    node->pstPrev = NULL;
}

...... ......
```



线性表的链接存储

- 单链表
- 循环链表
- 双向链表
- **静态链表**
- 链表的双指针技巧
- 复杂链表的拷贝
- 跳跃表

数据之法
结构之美
算法之道

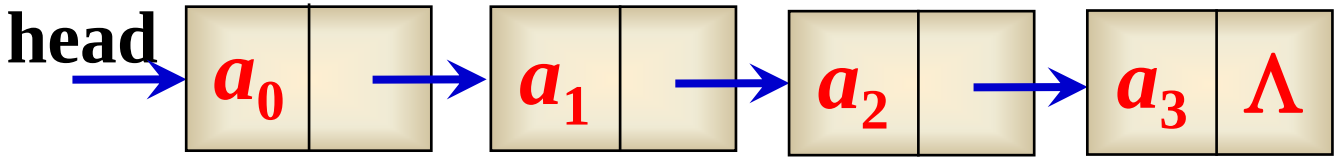
zhuyungang@jlu.edu.cn

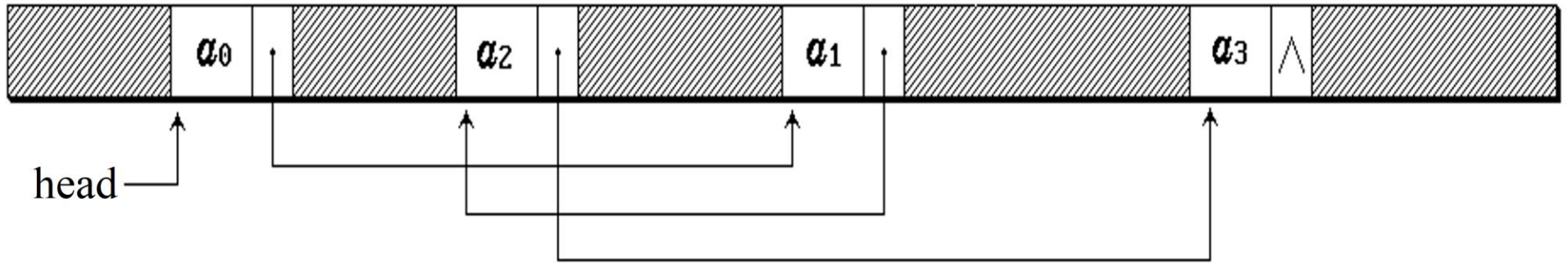
Last updated on 2022.9

静态链表

有些程序设计语言没有指针类型，如何实现链表？

回顾链表在内存的存储方式

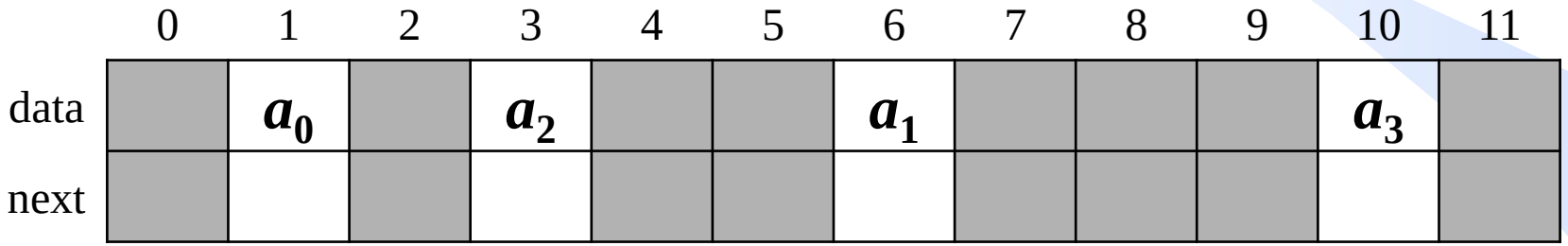


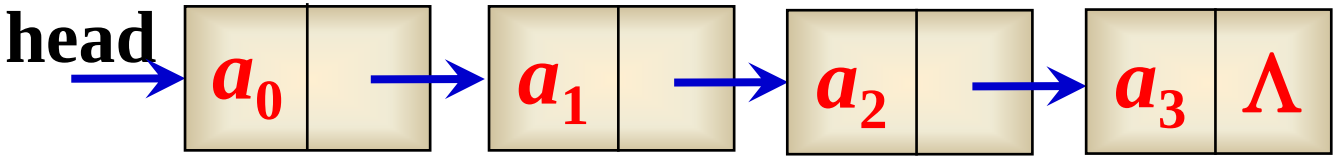


静态链表

```
int data[12];
int next[12];
```

链表的元素用数组存储，
用数组的下标模拟指针。

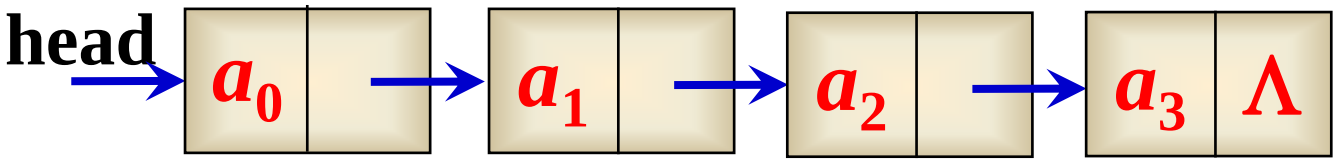




静态链表

```
int data[12];
int next[12];
```

	0	1	2	3	4	5	6	7	8	9	10	11
data		a_0		a_2			a_1				a_3	
next		6		10			3				-1	



静态链表作为一种编程技巧，在有指针的程序设计语言中，也有广泛的应用。

例题：有若干个盒子，从左至右依次编号为1,2,3,...,n。可执行以下指令（保证X不等于Y）：

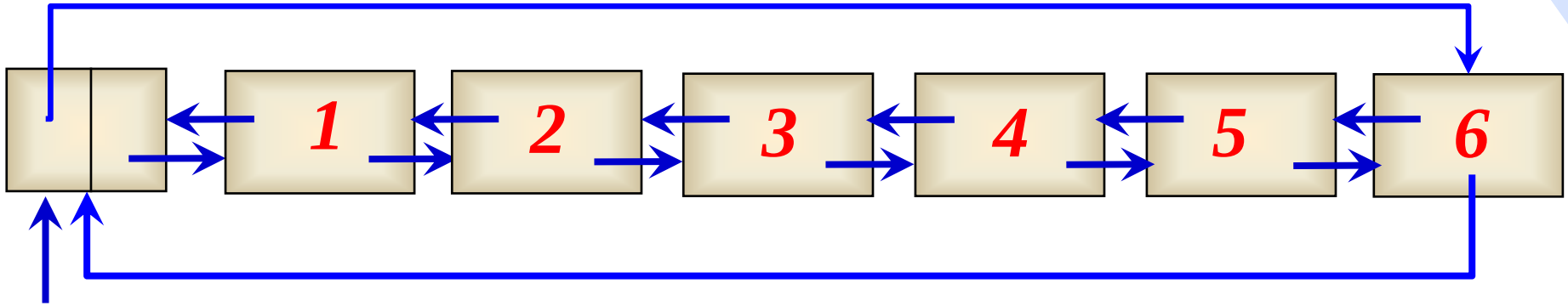
- L X Y表示把盒子X移动到盒子Y左边（如果X已在Y左边，则忽略该指令）。
- R X Y表示把盒子X移动到盒子Y右边（如果X已在Y右边，则忽略该指令）。

例如n=6时,初始：	**1**	**2**	**3**	**4**	**5**	**6**
执行指令L 1 4：	**2**	**3**	**1**	**4**	**5**	**6**
执行指令R 3 5：	**2**	**1**	**4**	**5**	**3**	**6**

静态双向循环链表

```
int data[7];
int left[7];
int right[7];
```

	0	1	2	3	4	5	6
left							
data		1	2	3	4	5	6
right							

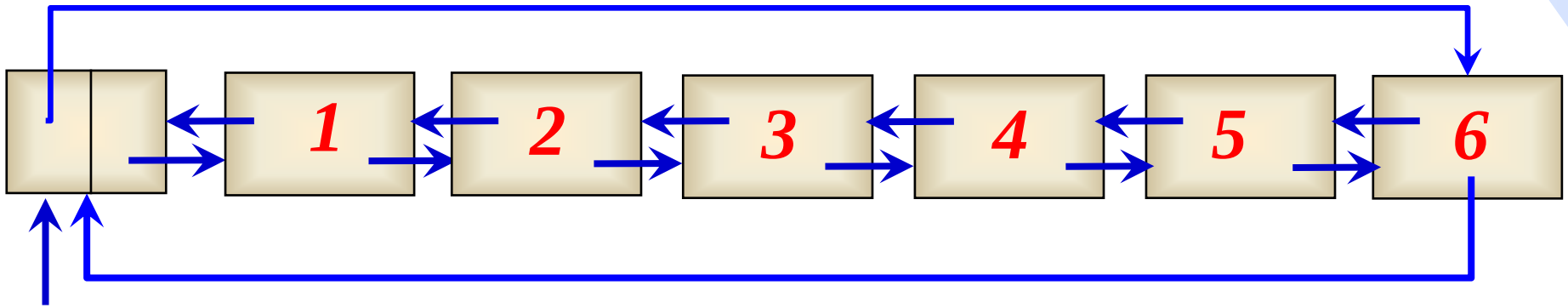


head

静态双向循环链表

```
int data[7];
int left[7];
int right[7];
```

	0	1	2	3	4	5	6
left	6	0	1	2	3	4	5
data		**1**	**2**	**3**	**4**	**5**	**6**
right	1	2	3	4	5	6	0



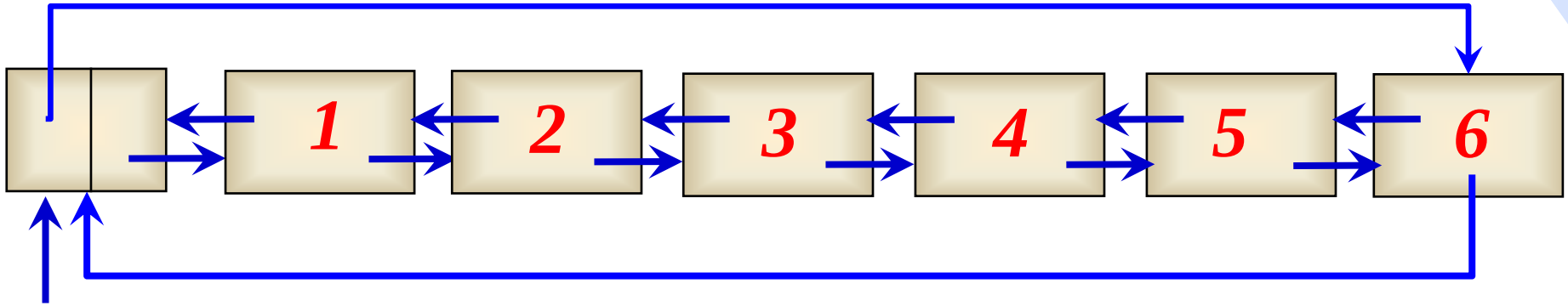
head

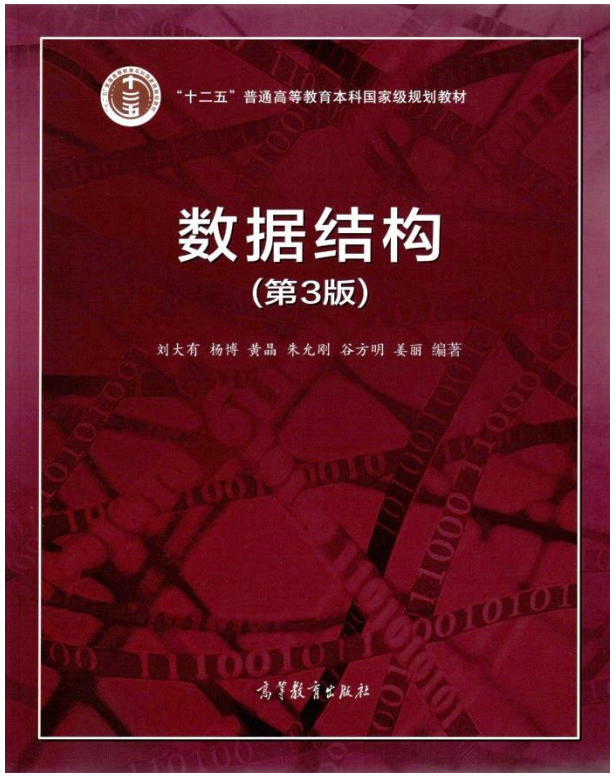
删除盒子x：
right[left[x]]=right[x];
left[right[x]]=left[x];

盒子x插到y右边：
left[right[y]]=x;
right[x]=right[y];
left[x]=y;
right[y]=x;

盒子x插到y左边：
right[left[y]]=x;
left[x]=left[y];
right[x]=y;
left[y]=x;

	0	1	2	3	4	5	6
int left[7]	6	0	1	2	3	4	5
int data[7]		1	2	3	4	5	6
int right[7]	1	2	3	4	5	6	0





线性表的链接存储

- 单链表
- 循环链表
- 双向链表
- 静态链表
- **链表的双指针技巧**
- 复杂链表的拷贝
- 跳跃表

数据之法
结构之美
算法之道

zhuyungang@jlu.edu.cn

Last updated on 2022.9

找单链表倒数第k个结点

- 已知一个带有表头结点的单链表，假设该链表只给出了头指针list。在不改变链表的前提下，请设计一个尽可能高效的算法，查找链表中倒数第k个位置上的结点（k为正整数）。若查找成功，算法输出该结点的data值，并返回1；否则，只返回0。（考研题全国卷，滴滴面试题）

- 解法1：

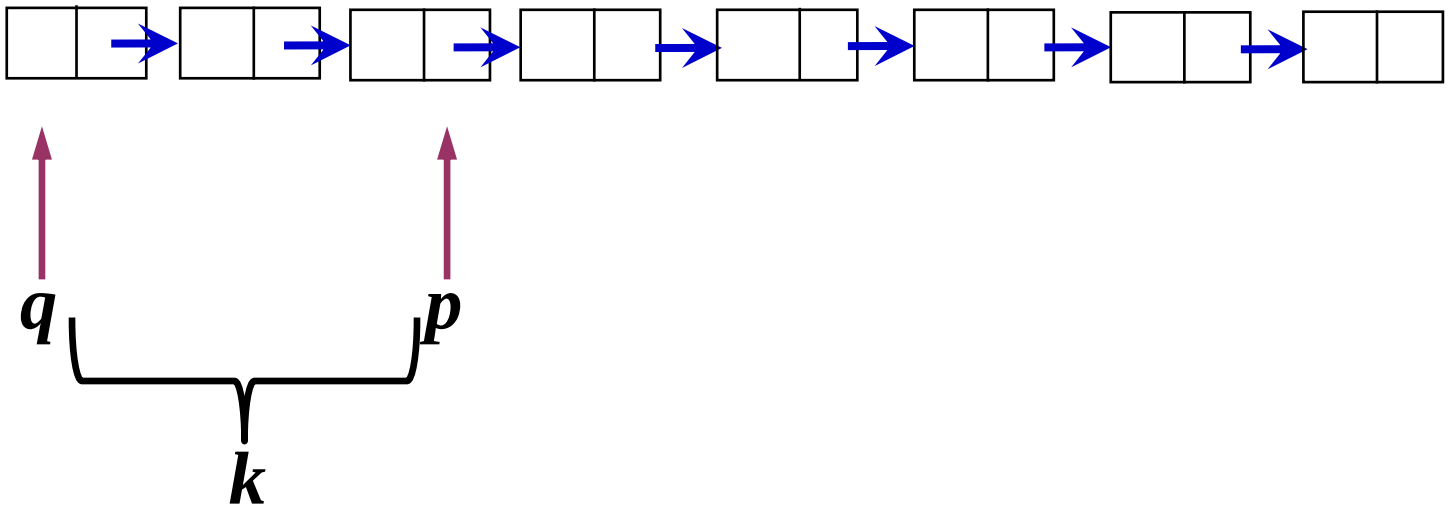
- 找到最后一个结点，往前找*k*-1次前驱
- 找某个结点的前驱结点：O(n)
- 整个算法$O(n^2)$

- 解法2：

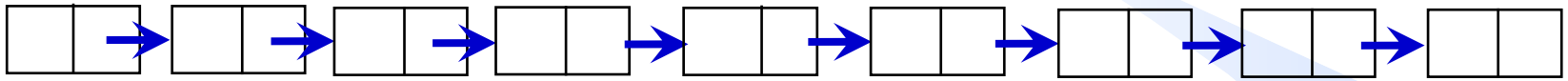
- 倒数第k个，即正数第n-k+1个
- 时间复杂度O(n)，遍历2次链表

解法3：

- 使用两个指针:*p*和*q*，先把*p*指向第*k*个元素，*q*指向第1个元素。然后*p*和*q*同时向后遍历，当*p*遍历到结尾时，*q*正好遍历到倒数第*k*个。
- 两个指针并行遍历，遍历一次，时间复杂度O(n)。



找单链表中间位置的结点，要求只遍历一次链表。【腾讯面试题】



维护两个指针fast和slow

fast每次移动2步，slow每次移动1步

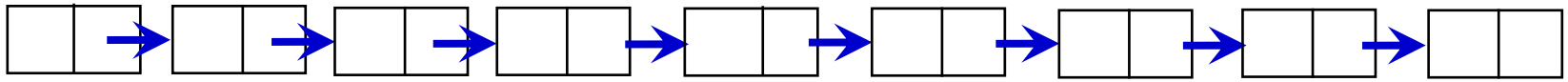
fast←next(next(fast)).

slow ← next(slow).

C/C++:

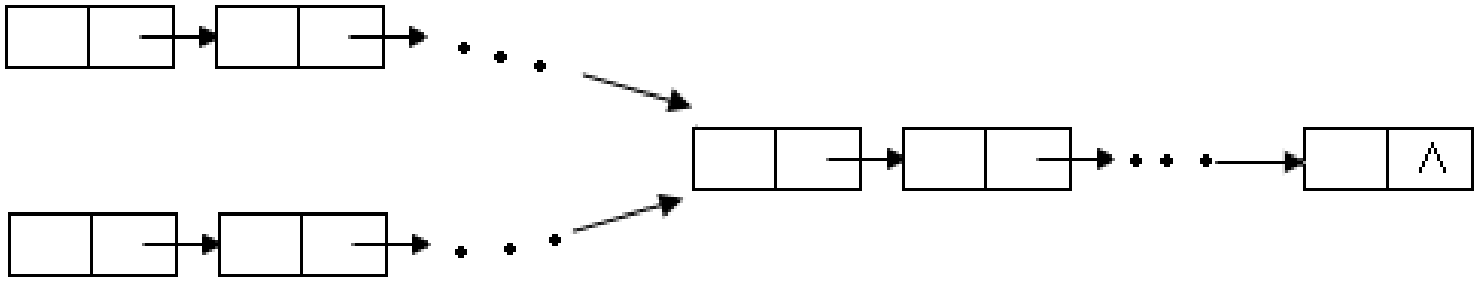
```
ListNode* middleNode(ListNode* head) {
    ListNode *fast = head, *slow = head;
    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
    }
    return slow;
}
```

课下思考：对于长度为偶数的链表，若要返回第1个中间结点（即两个中间结点中靠左的那个结点），该如何修改上述代码？

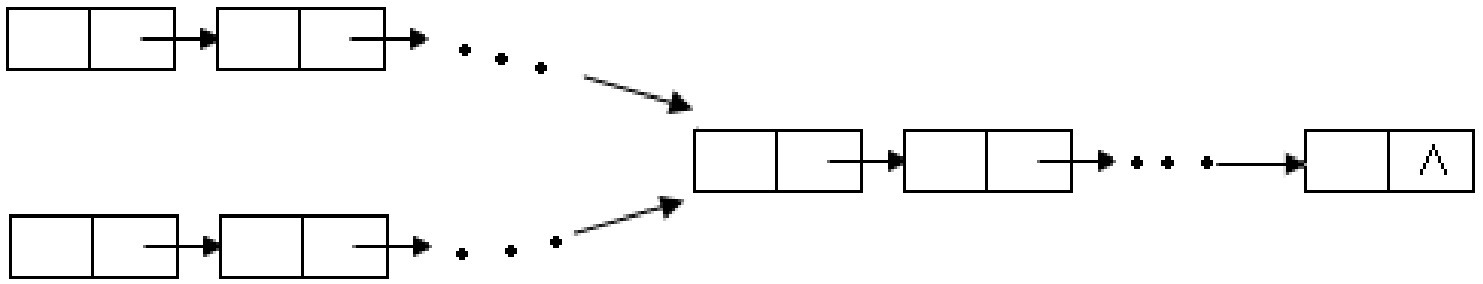


链表相交问题

给定两个单链表的头指针head1和head2，设计一个算法判断这两个链表是否相交,如果相交则返回第一个交点，要求时间复杂度为O(L1+L2)，L1、L2分别为两个链表的长度。为了简化问题，这里我们假设两个链表均不含有环.【微软、腾讯、小米面试题】



链表相交问题



方案1:

- **FOR** ∀结点*i* ∈ 链表1 **DO**

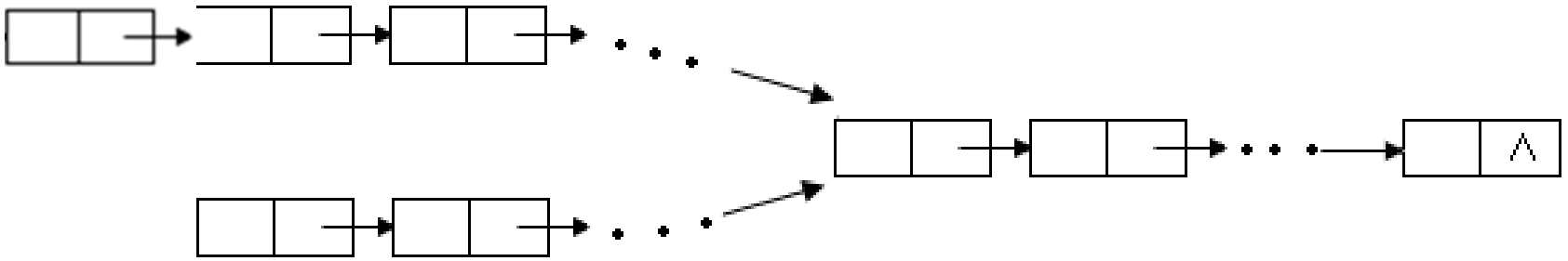
  (

  判断结点*i*是否在链表2中.

)

- **O(L1*L2)**

链表相交问题

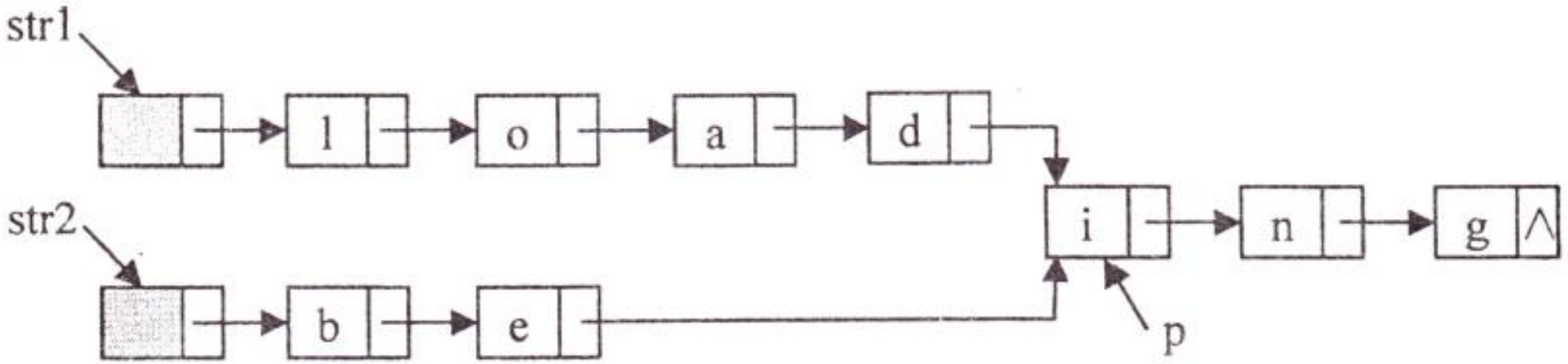


- 思路：
- 是否相交？如果两个链表相交，则最后一个结点一定是共有的，可以分别遍历2个链表，记录其最后一个结点和链表长度。若2个链表最后一个结点相等，则相交，否则不相交。
- 找相交结点？用指针p1指向较长的那个链表，p2指向较短的那个链表，p1先向后移动|L1-L2|步，然后p1和p2同时向后移动，每移动一步比较p1和p2是否相等，当二者相等时，其指向的结点即为交点。

```
ListNode* cross(ListNode *head1, ListNode *head2) {
//若两个链表相交，返回交点指针，否则返回NULL
        if (head1==NULL || head2==NULL) return NULL;
        if (head1==head2) return head1;
        ListNode *p1=head1,*p2=head2;
        int L1=1,L2=1;
        while (p1->next!=NULL) { L1++;  p1=p1->next;}
        while (p2->next!=NULL) { L2++;  p2=p2->next;}
        if (p1!=p2) return NULL; //不相交
        if (L1>=L2)  { p1=head1; p2=head2;}
        else { p1=head2; p2=head1;}
        for(int i=0;i<abs(L1-L2);i++)  p1=p1->next; //p1和p2对齐
        while (p1!=p2) {
                p1=p1->next;
                p2=p2->next;
        }
        return p1;
}
```

考研题全国卷

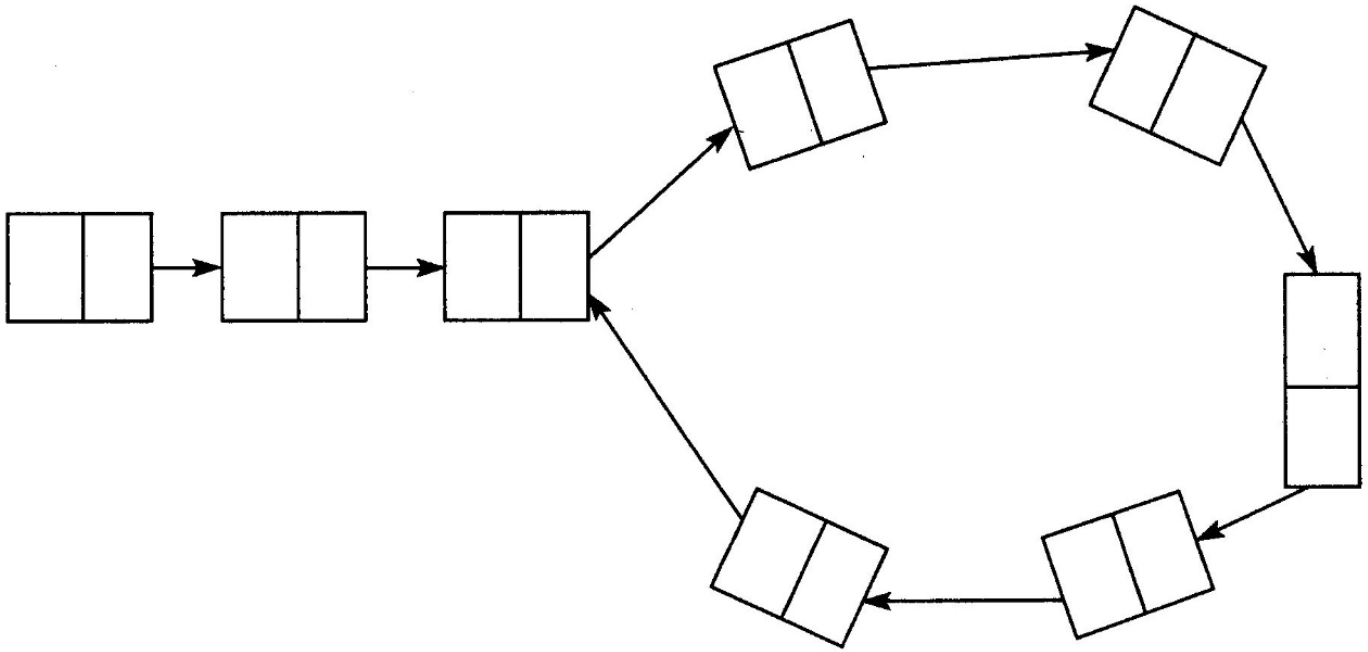
（13 分）假定采用带头结点的单链表保存单词，当两个单词有相同的后缀时，则可共享相同的后缀存储空间。例如，“loading”和“being”的存储映像如下图所示。



设 *str1* 和 *str2* 分别指向两个单词所在单链表的头结点，链表结点结构为 | data | next |，请设计一个时间上尽可能高效的算法，找出由 *str1* 和 *str2* 所指的两个链表共同后缀的起始位置（如图中字符 i 所在结点的位置 *p*）。

单链表判环问题

编写算法（1）判断一个单链表中是否含有环。如果有环的话：（2）找出从头结点进入环的第一个结点。【微软、腾讯、百度、小米面试题】



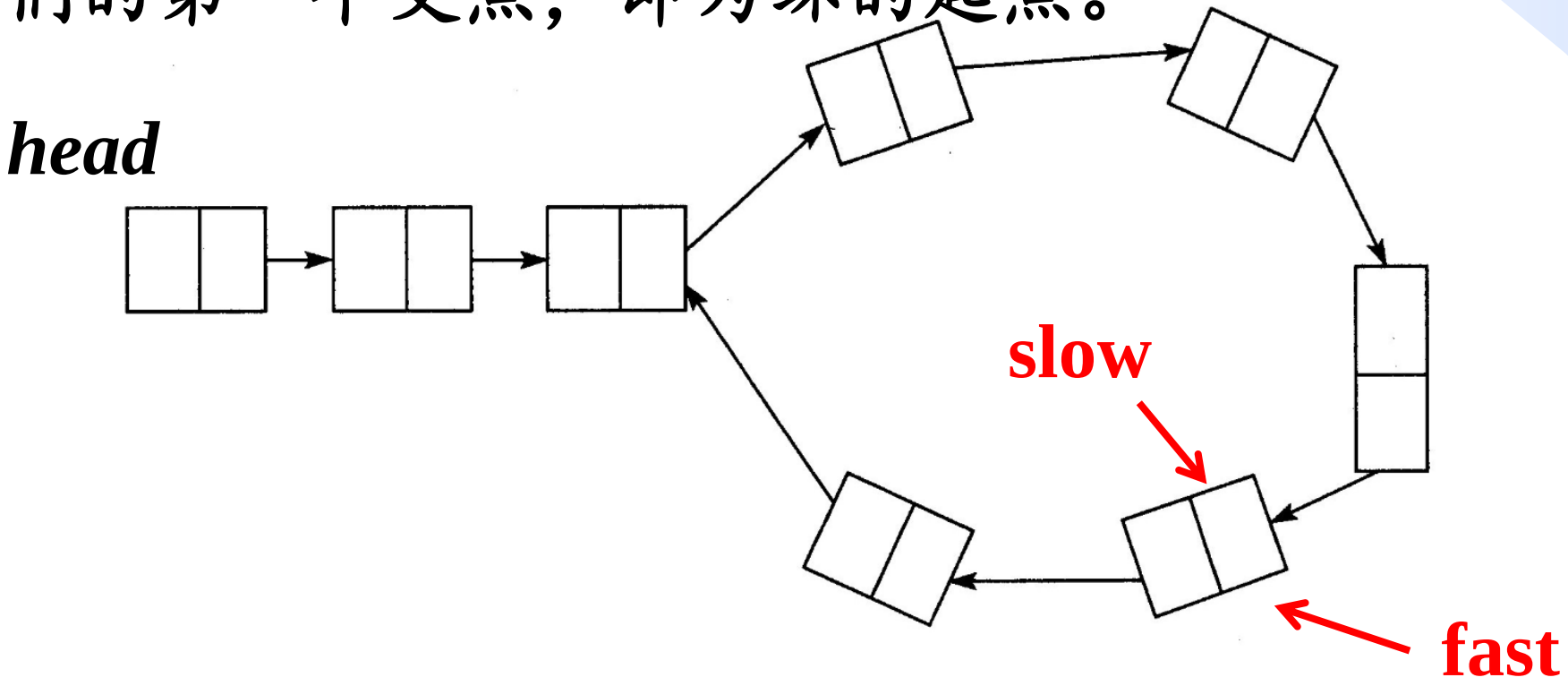
单链表判环问题

- 使用两个指针slow和fast从链表头开始遍历，slow每次前进1步，fast每次前进2步。
- 如果含有环，fast和slow必然会在某个时刻相遇（fast==slow），好比在环形跑道上赛跑时运动员的套圈。
- 如果遍历过程中，最终fast达到NULL，则说明无环。

```
bool hasCycle(ListNode *head) {
    ListNode *slow = head, *fast = head;
    while (fast != NULL && fast->next != NULL){
        slow = slow->next;
        fast = fast->next->next;
        if (fast == slow) return true;
    }
    return false;
}
```

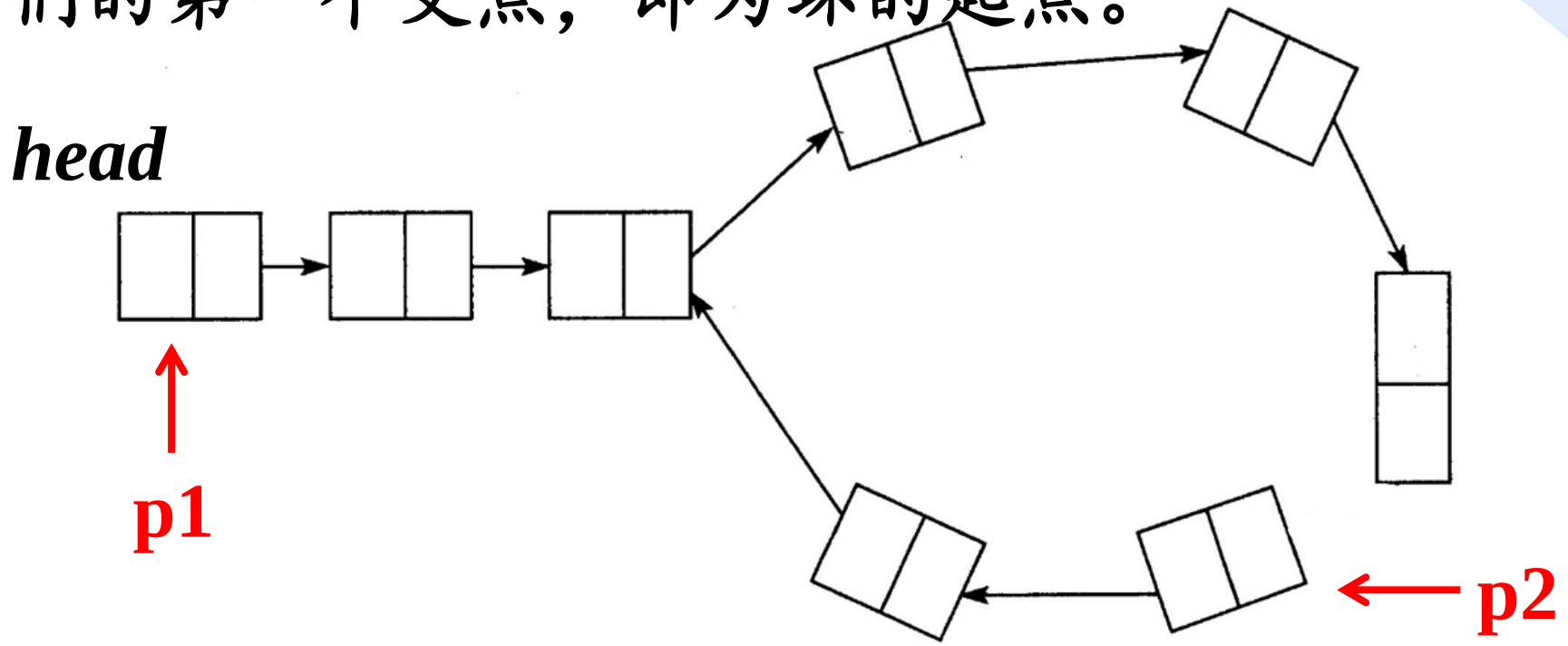
单链表判环问题

从相遇结点（fast==slow所对应的结点）处断开环，令p1←head，p2←fast。此时，原单链表可以看作两条单链表，一条从p1开始，另一条从p2开始，结合链表相交问题，找到它们的第一个交点，即为环的起点。



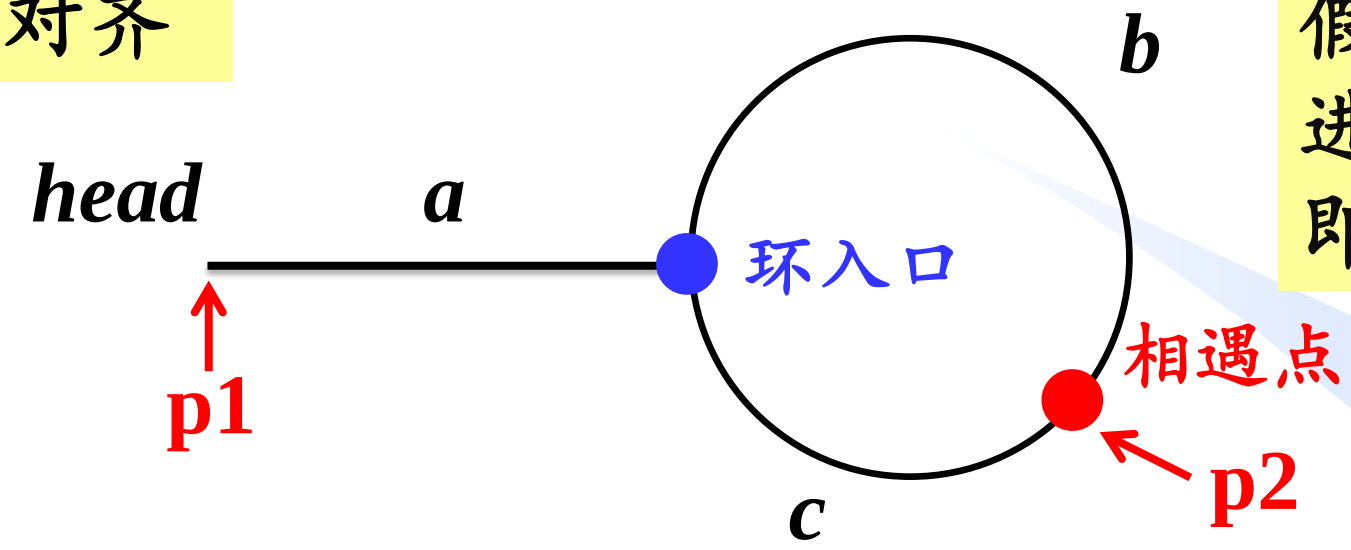
单链表判环问题

从相遇结点（fast==slow所对应的结点）处断开环，令p1←head，p2←fast。此时，原单链表可以看作两条单链表，一条从p1开始，另一条从p2开始，结合链表相交问题，找到它们的第一个交点，即为环的起点。



单链表判环问题

p1和p2无需对齐



假定fast指针进入环1圈后即与slow相遇

2*slow指针走的步数= fast指针走的步数

$$2(a+b) = a+b+c+b$$

$$a = c$$

若fast在环内走了很多圈才与slow相遇，刚才的策略是否还可行？

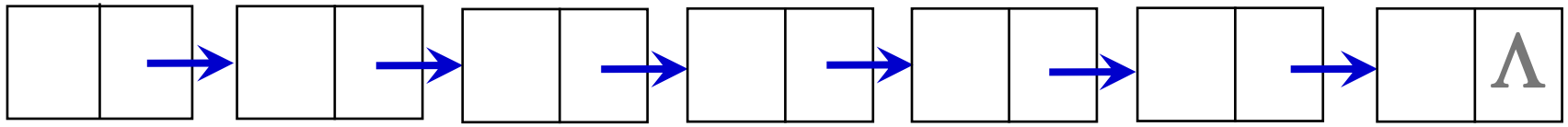
单链表判环问题

```
ListNode* detectCycleEntrance(ListNode *head) {
        ListNode *slow = head, *fast = head;
        while (fast != NULL && fast->next != NULL){
            fast = fast->next->next;
            slow = slow->next;
            if (fast == slow) {     //有环，找环的入口
                ListNode *p1 = head, *p2=fast;
                while (p1 != p2) {
                    p1 = p1->next;
                    p2 = p2->next;
                }
                return p1;
            }
        }
        return NULL;
}
```

判断一个单链表是否是回文

要求时间复杂度O(n)，空间复杂度O(1)

【华为、字节跳动、微软、谷歌、苹果、快手面试题】



回文：

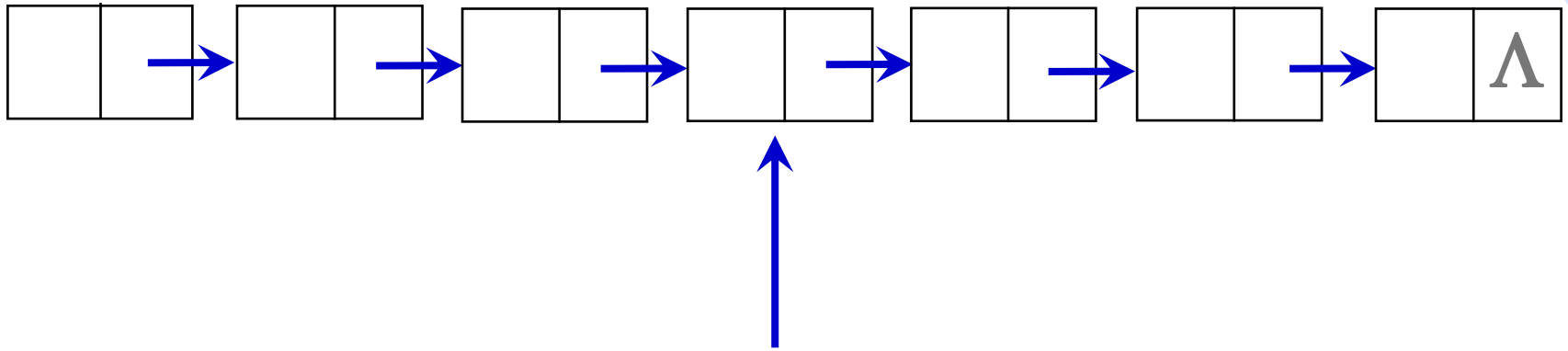
l e v e l

r e f e r

上海自来水来自海上

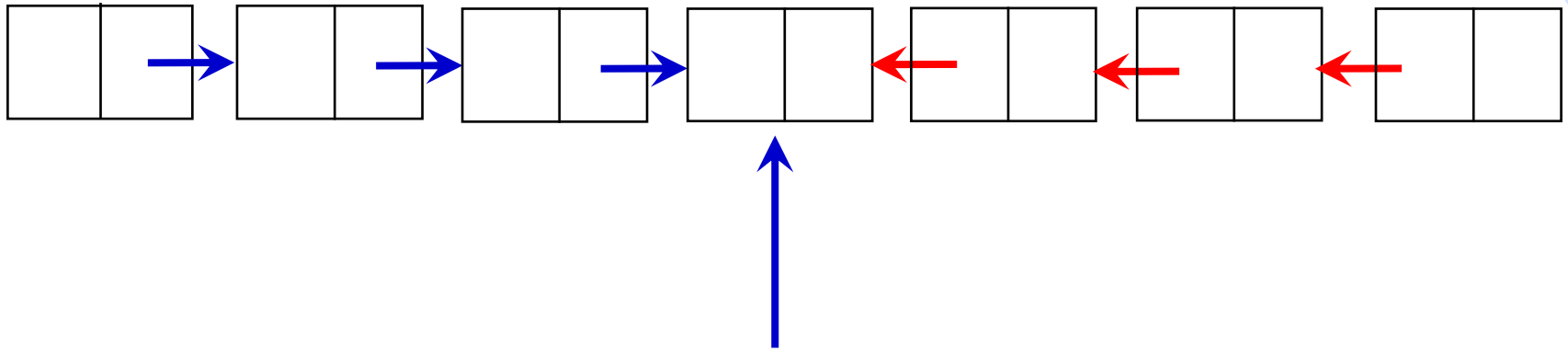
判断一个单链表是否是回文

要求时间复杂度O(n)，空间复杂度O(1)



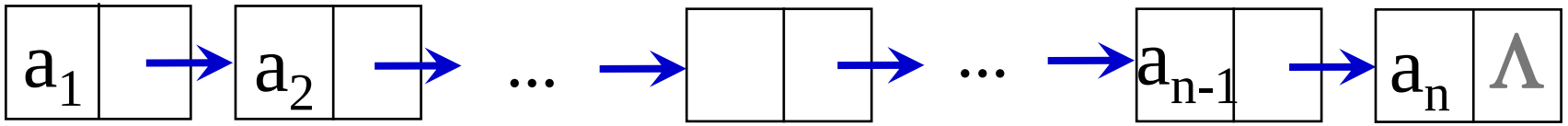
判断一个单链表是否是回文

要求时间复杂度O(n)，空间复杂度O(1)



重排链表结点

设线性表L=(a_1, a_2, a_3, …, a_{n-2}, a_{n-1}, a_n)采用带哨位结点的单链表保存，请设计一个空间复杂度为O(1)且时间上尽可能高效的算法，重新排列L中的各结点，得到线性表L'= (a_1, a_n, a_2, a_{n-1}, a_3, a_{n-2}…)。**【2019年考研题全国卷（13分），字节跳动、美团、小米、腾讯、百度、华为、阿里、拼多多、快手面试题】**



一元多项式及其操作

$$p(x)=a_n x^n + a_{n-1}x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0$$

一元多项式的链表结点结构如下，每个结点包含两个数据域（系数和指数）和一个链接域。

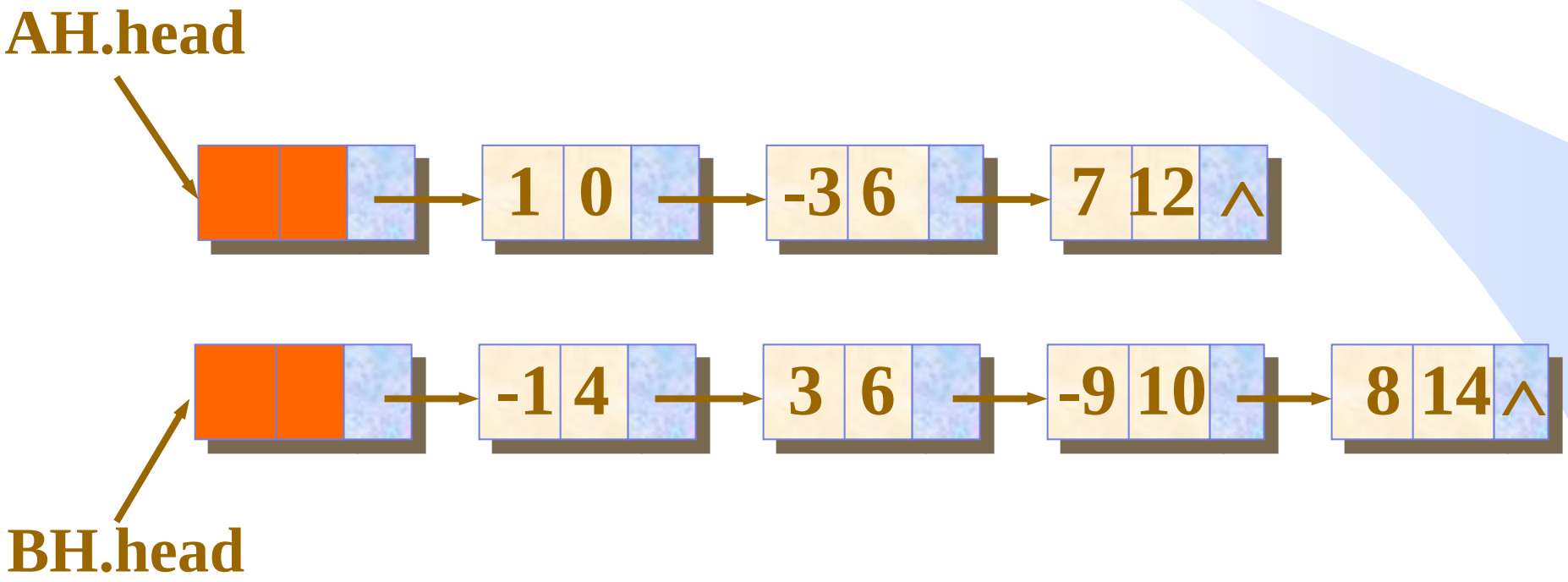
coef	exp	link

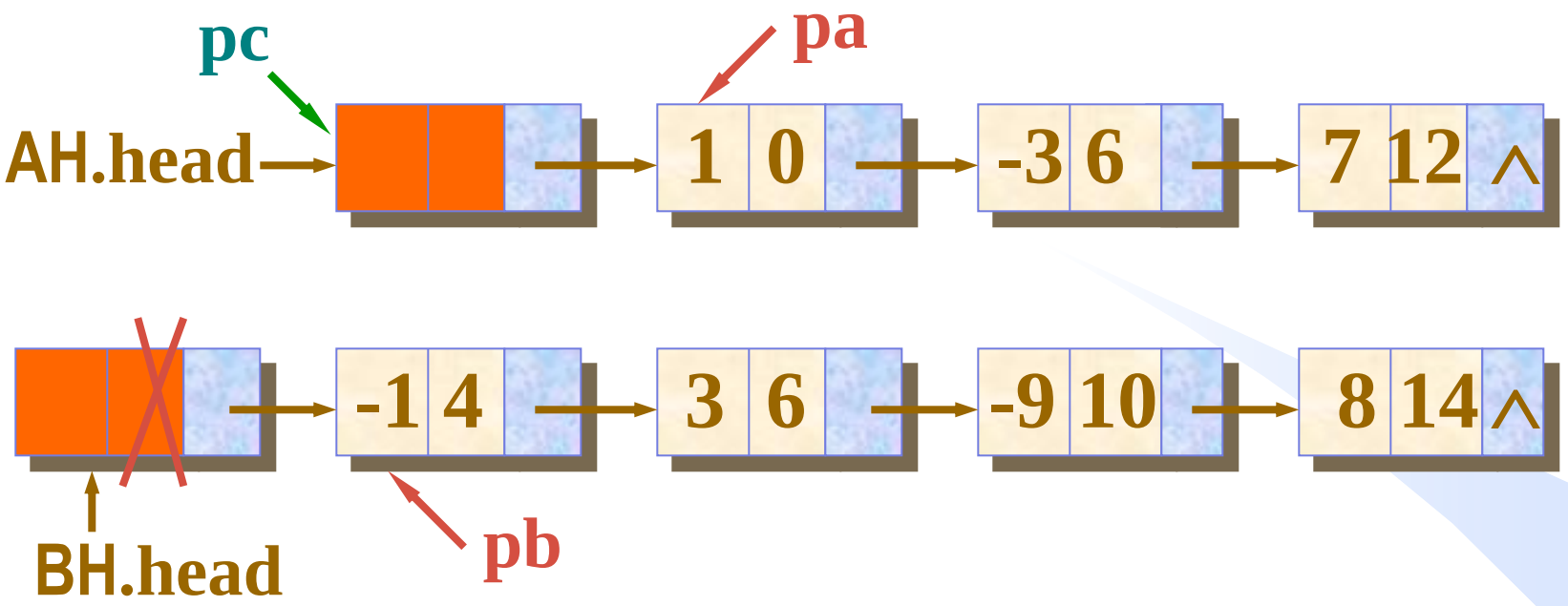
优点：多项式的项数可以动态地增长，不会出现存储溢出问题。插入、删除方便，不移动元素，只需“穿针引线”。

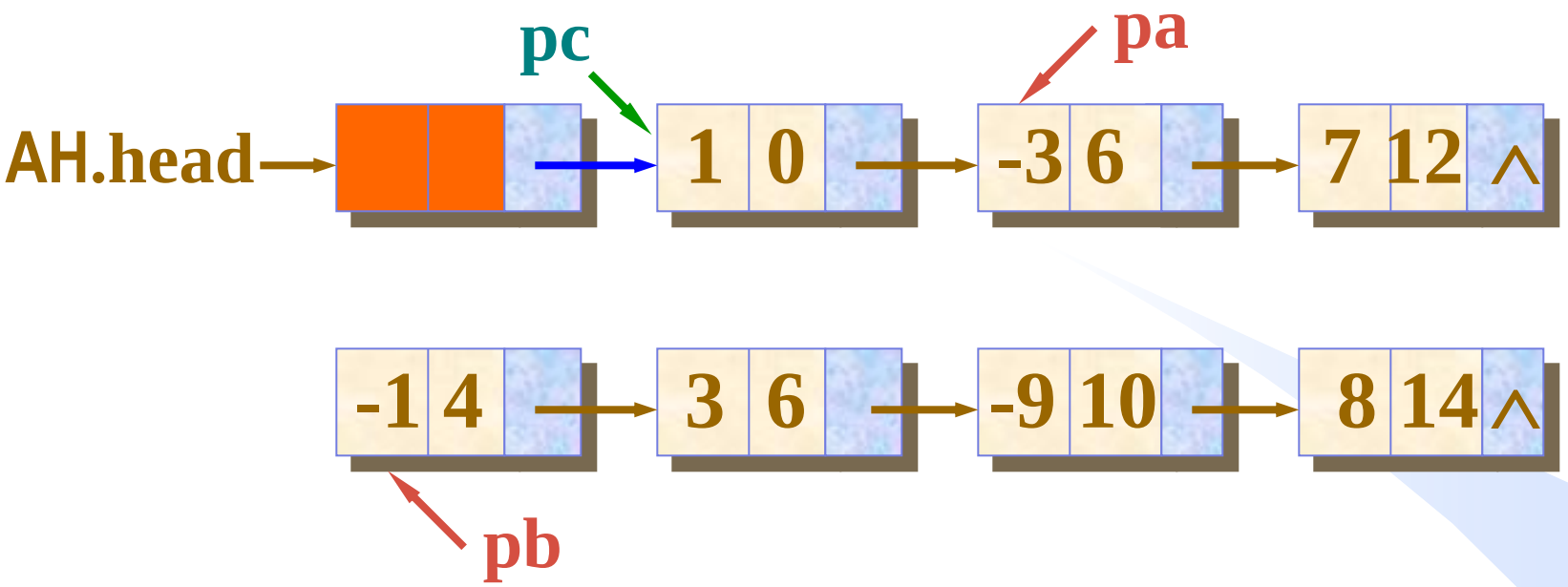
多项式链表相加示例

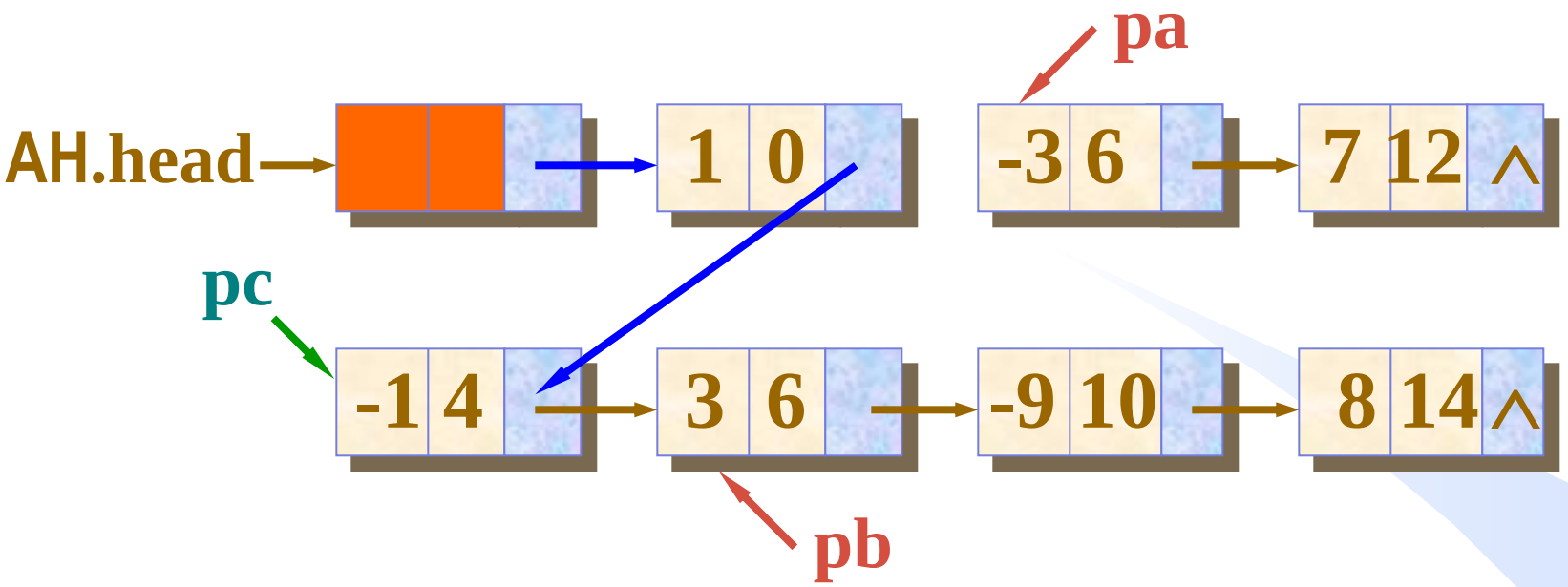
$$AH = 1 - 3x^6 + 7x^{12}$$

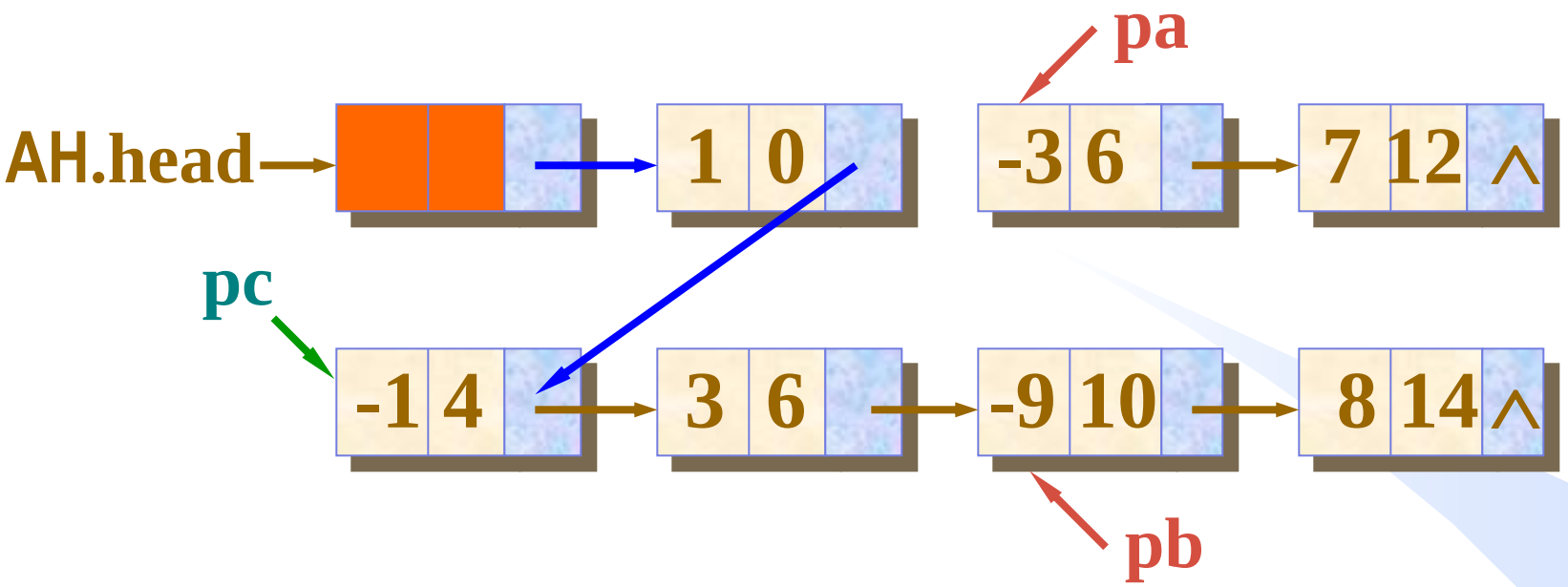
$$BH = -x^4 + 3x^6 - 9x^{10} + 8x^{14}$$

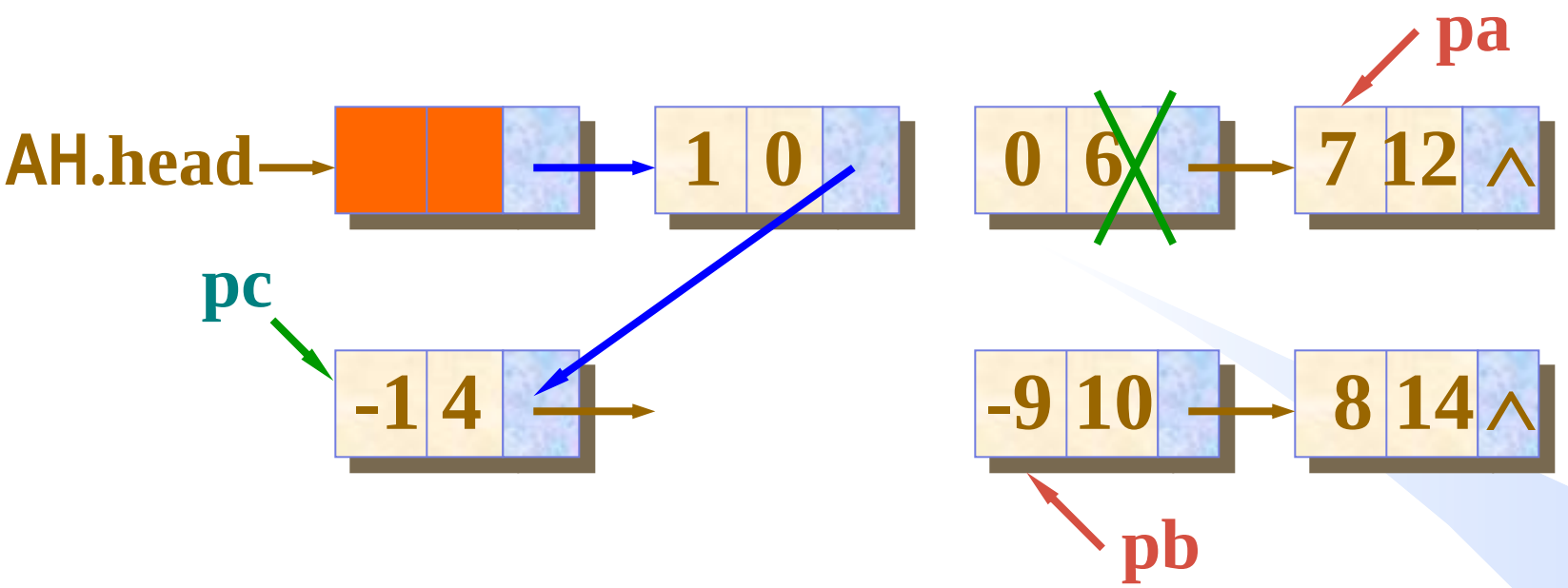


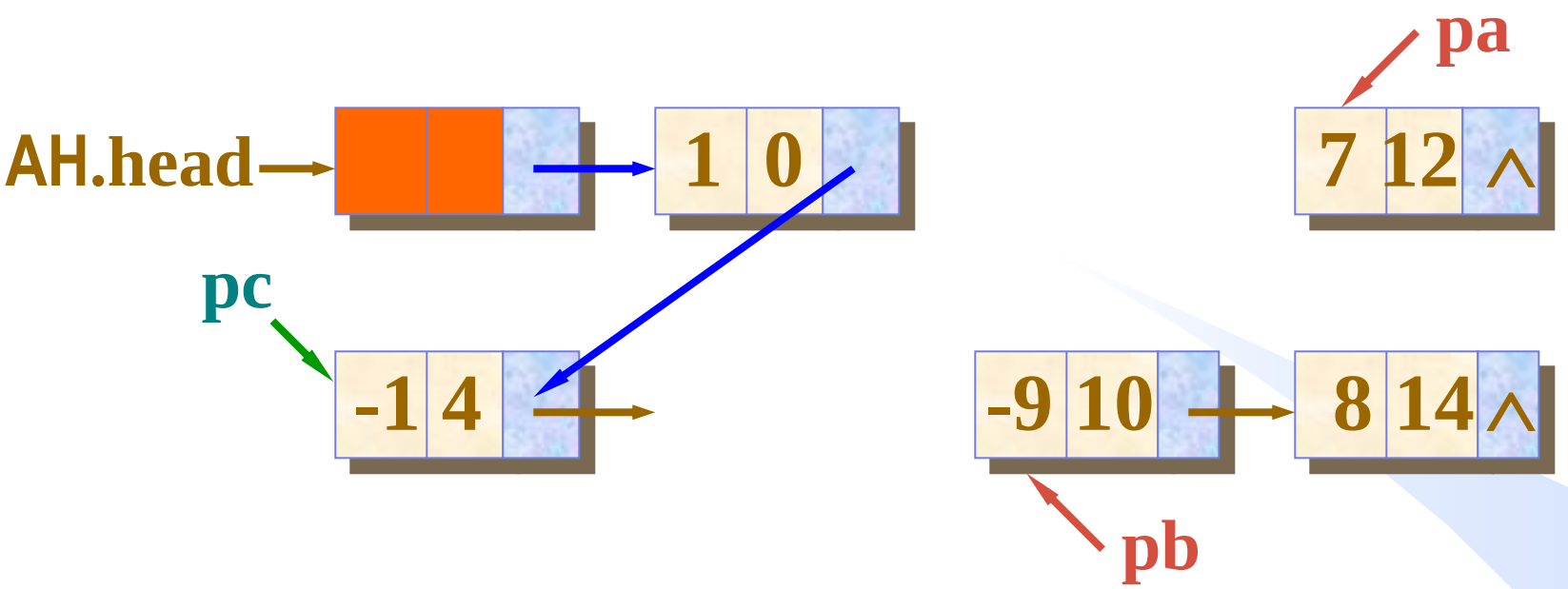

pc
pa
AH.head
1 0
-3 6
7 12 ∧
-1 4
3 6
-9 10
8 14 ∧
BH.head
pb

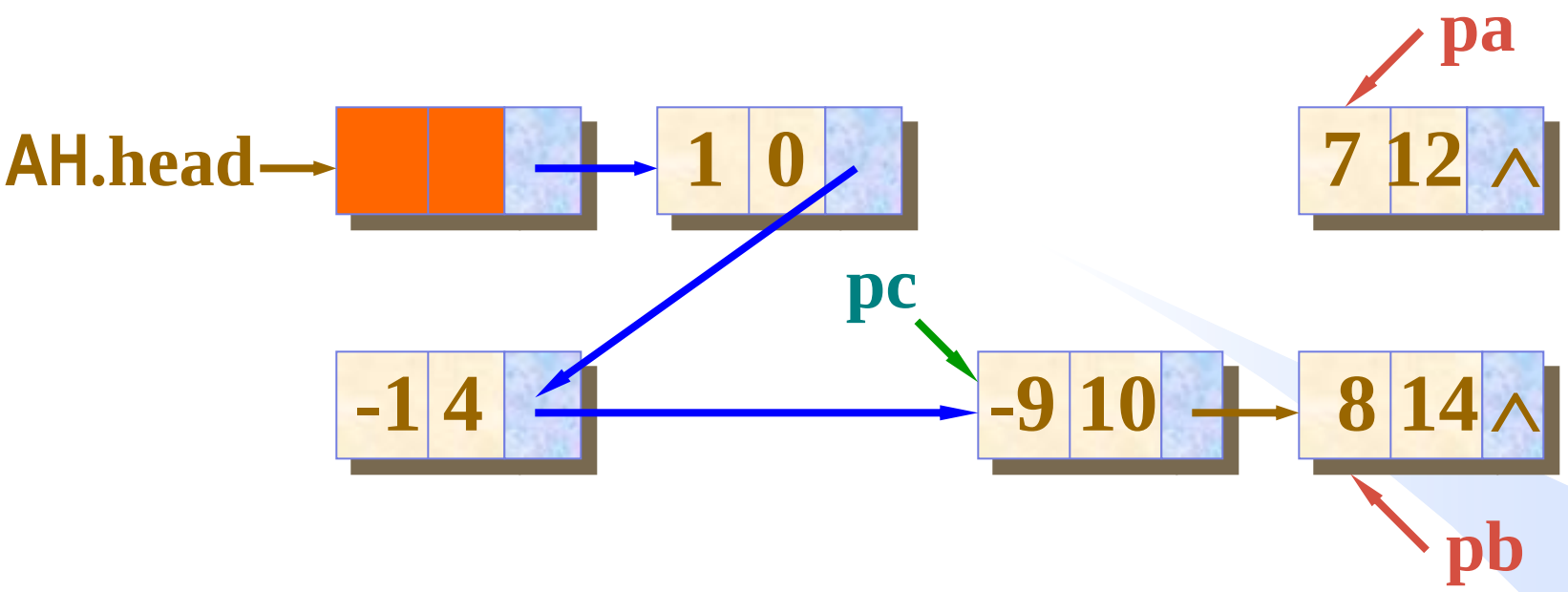

pc
pa
AH.head
1 0
-3 6
7 12 ∧
-1 4
3 6
-9 10
8 14 ∧
pb

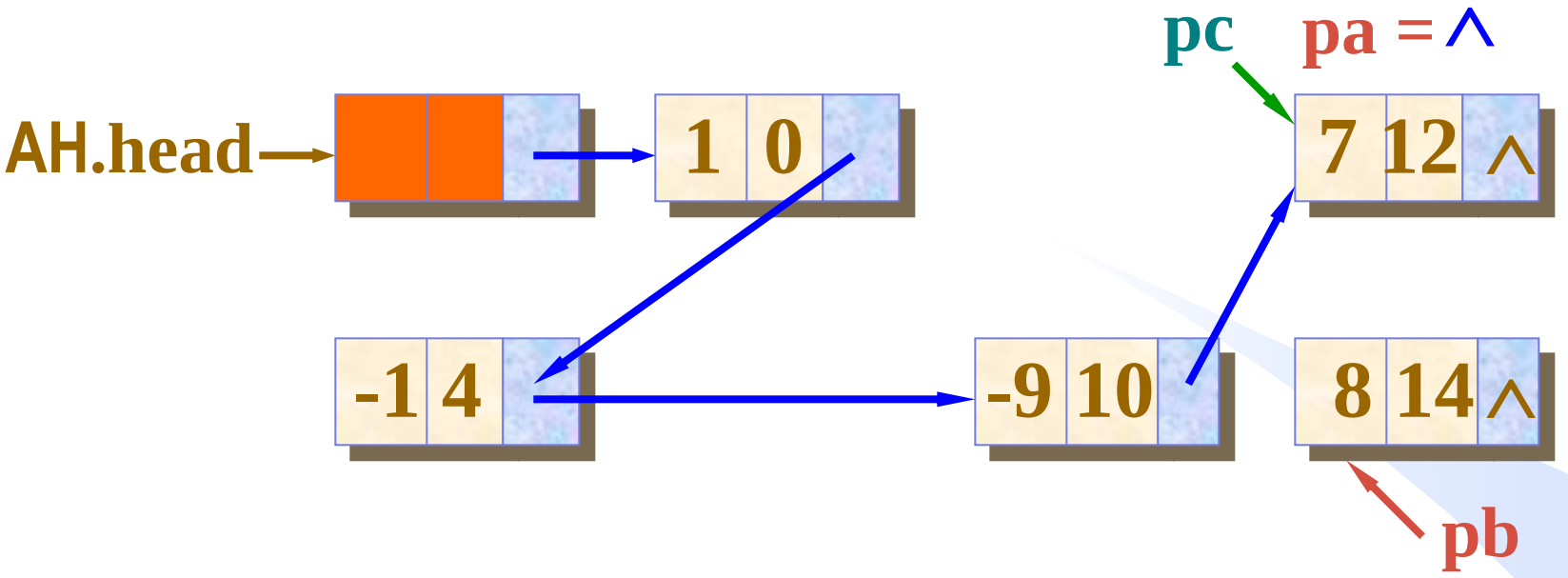

pa
AH.head
1 0
-3 6
7 12 ∧
pc
-1 4
3 6
-9 10
8 14 ∧
pb

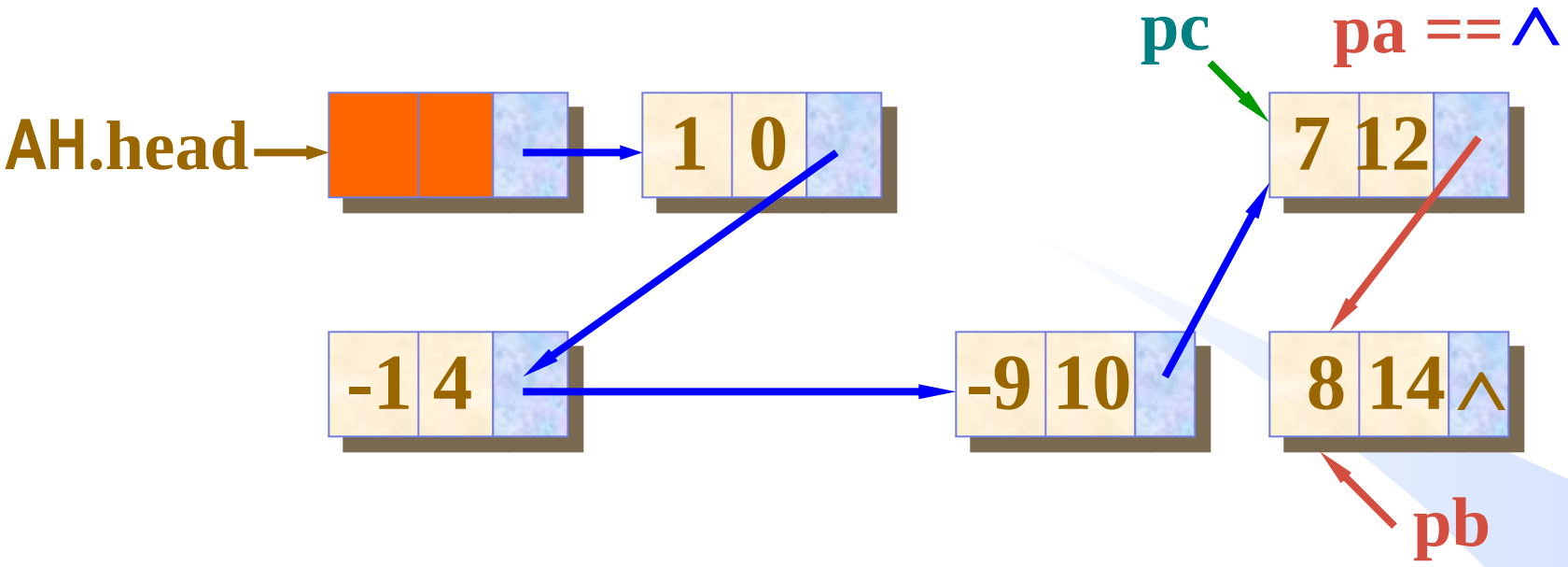

pa
AH.head
1 0
-3 6
7 12 ∧
pc
-1 4
3 6
-9 10
8 14 ∧
pb

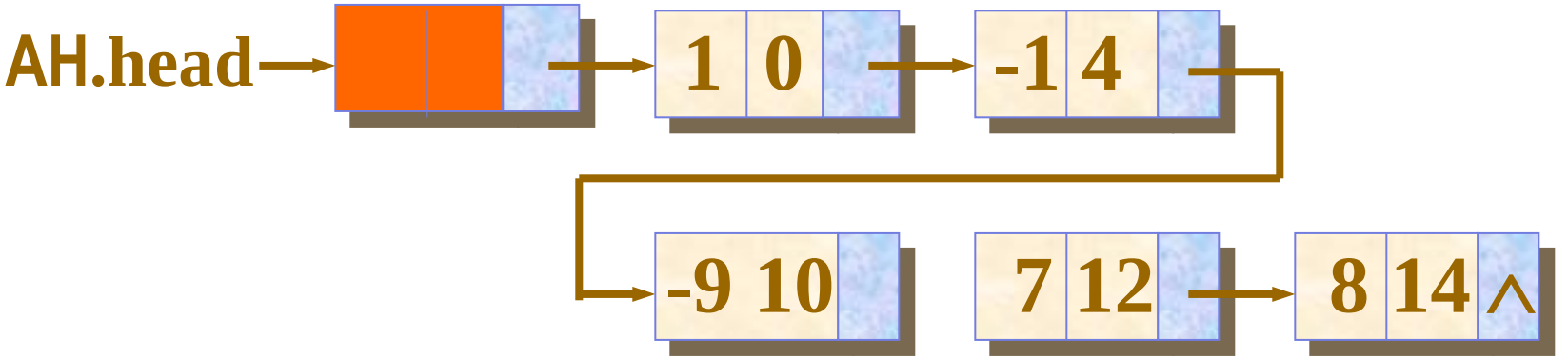

AH.head
1 0
0 6
7 12 ∧
pa
pc
-1 4
-9 10
8 14 ∧
pb

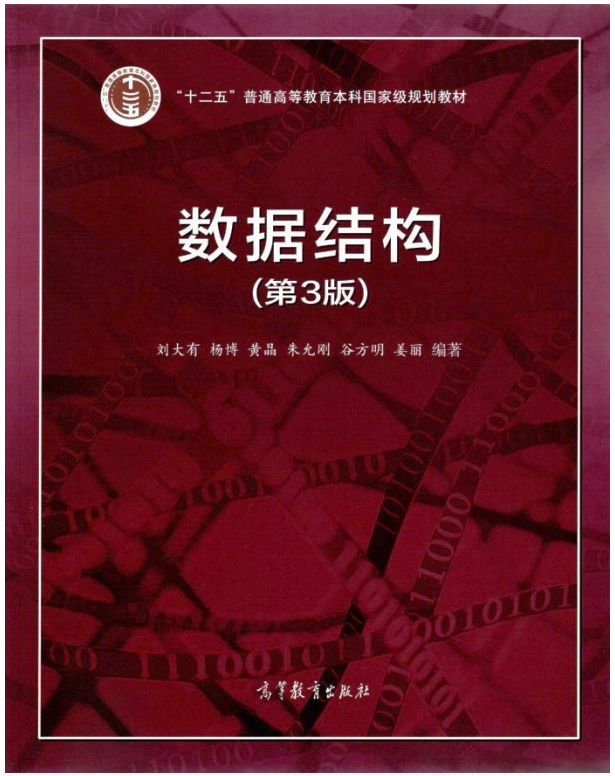

AH.head
1 0
pa
7 12 ∧
pc
-1 4
-9 10
8 14 ∧
pb


AH.head
1 0
pa
7 12 ∧
pc
-1 4
-9 10
8 14 ∧
pb


pc
pa = ∧
AH.head
1 0
7 12 ∧
-1 4
-9 10
8 14 ∧
pb


pc
pa == ∧
AH.head
1 0
7 12
-1 4
-9 10
8 14 ∧
pb


AH.head
1 0
-1 4
-9 10
7 12
8 14 ∧



线性表的链接存储

- 单链表
- 循环链表
- 双向链表
- 静态链表
- 链表的双指针技巧
- **复杂链表的拷贝**
- 跳跃表

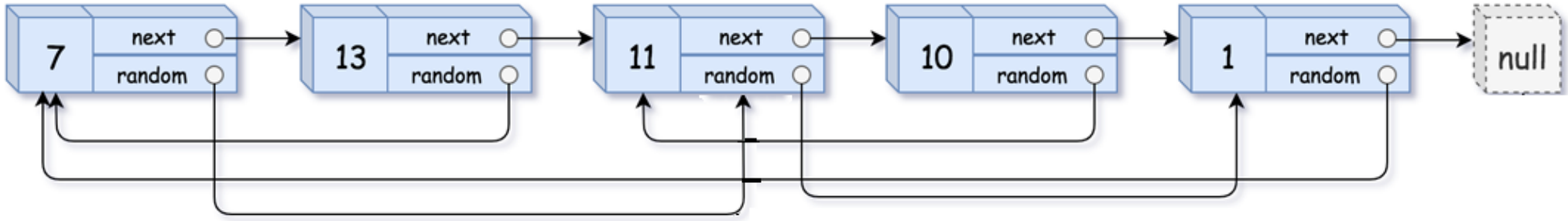
数据之法
结构之美
算法之道

zhuyungang@jlu.edu.cn

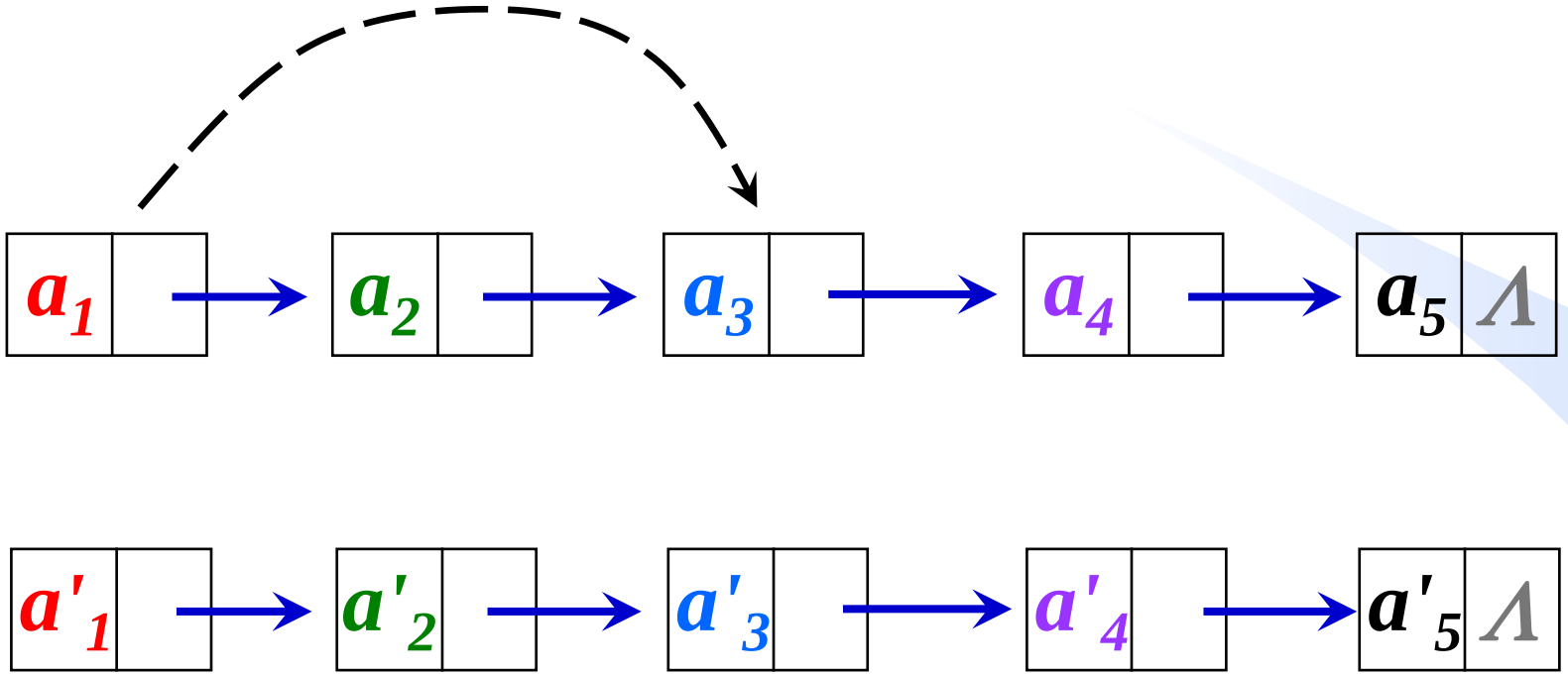
Last updated on 2022.9

复制带随机指针的链表

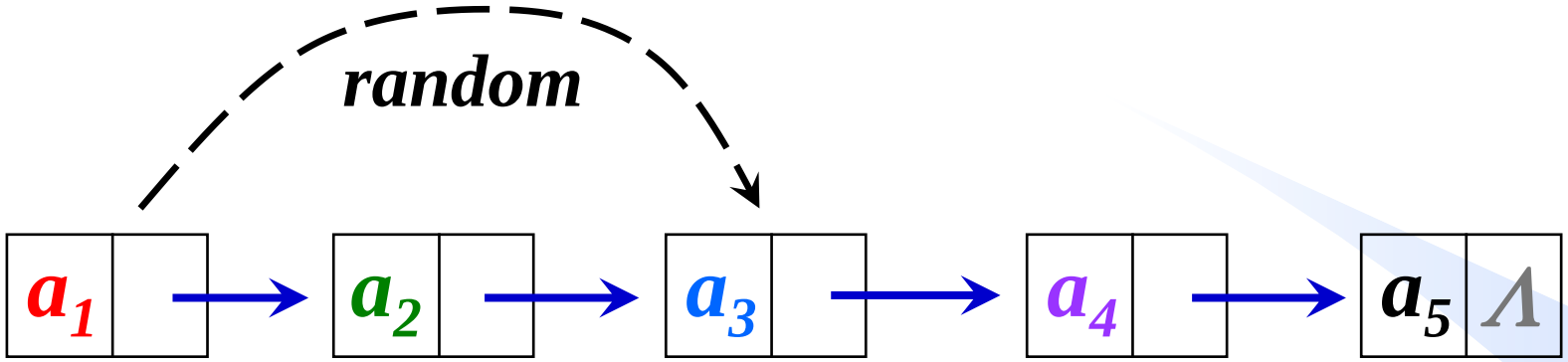
给定一个长度为 n 的链表，每个结点包含一个额外增加的随机指针 random，该指针可以指向链表中的任何结点或空结点。请构造这个链表的深拷贝。深拷贝应该正好由 n 个全新结点组成，其中每个新结点的数据域值都为其对应的原结点的数据值。新结点的 next 指针和 random 指针也都应指向复制链表中的新结点。复制链表中的指针都不应指向原链表中的节点。【字节跳动、微软、谷歌面试题】



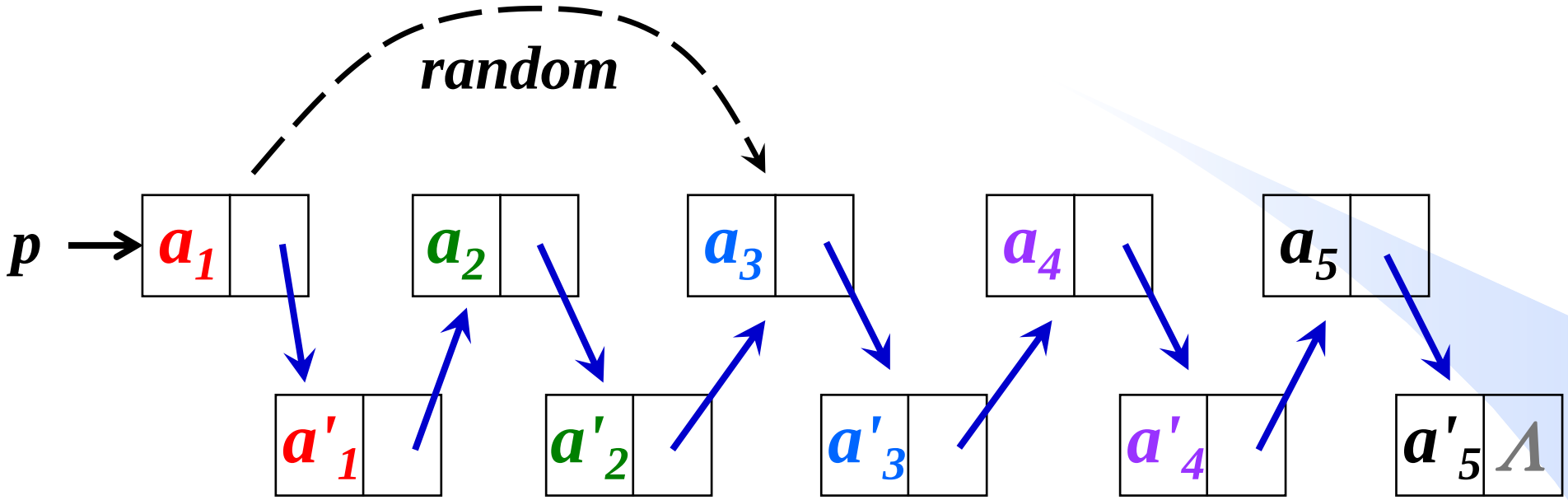
复制带随机指针的链表



复制带随机指针的链表



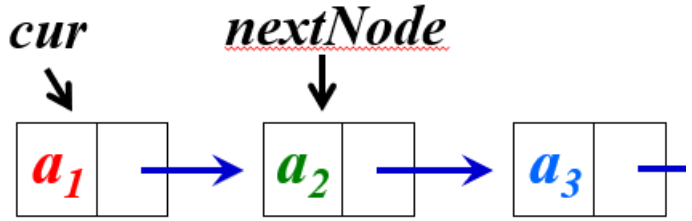
复制带随机指针的链表




```
p->next->random = p->random->next;   p->random可能为空
```

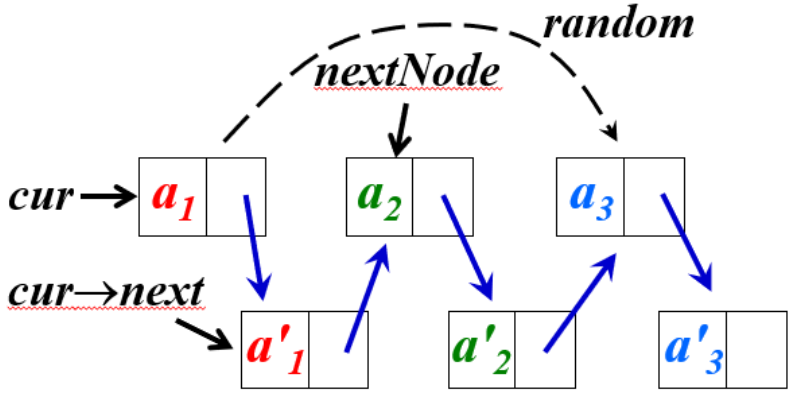
```
p->next->random =(p->random==NULL)?NULL:p->random->next;
```

复制带随机指针的链表




```
Node* copyRandomList(Node* head) {
    if (head == NULL) {return NULL; }
    Node* cur = head;
    while(cur != NULL){          //拷贝结点，并将拷贝出的结点作为原结点的后继
        Node *nextNode = cur->next;      cur->next = new Node(cur->val);
        cur->next->next = nextNode;      cur = nextNode;
    }
```

复制带随机指针的链表




```
Node* copyRandomList(Node* head) {
    if (head == NULL) {return NULL; }
    Node* cur = head;
    while(cur != NULL){
        Node *nextNode = cur->next;
        cur->next->next = nextNode;
    }
    cur = head;
    while(cur != NULL){          //拷贝random指针
        Node * nextNode = cur->next->next;
        cur->next->random = (cur->random==NULL)? NULL : cur->random->next;
        cur = nextNode;
    }
    Node *headCopy = head->next; cur = head;
    while(cur != NULL){          //拆分
        Node * nextNode = cur->next->next;
        cur->next->next =(nextNode ==NULL)? NULL : nextNode->next;
        cur->next = nextNode;
    }
    return headCopy;
}
```

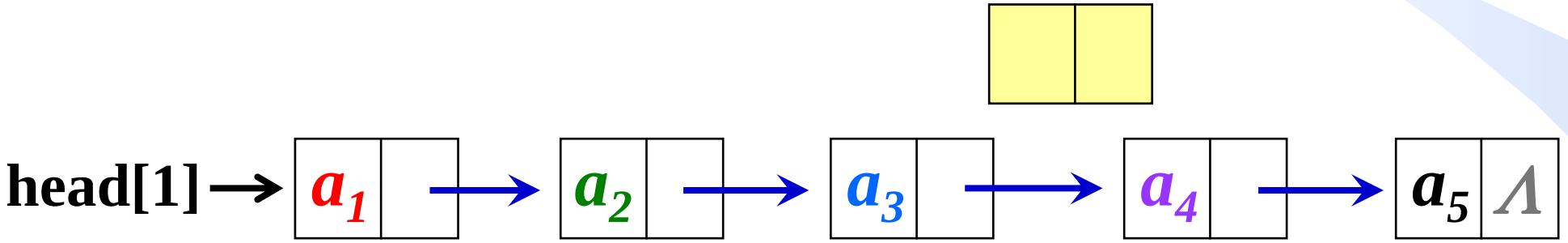
```
        cur->next = new Node(cur->val);
        cur = nextNode;
```

可持久化数据结构

保存数据结构的历史版本，使数据结构可以恢复到任意历史状态。

实现方法：在对数据结构进行修改前，对当前版本拷贝（备份）。

增量式备份：只拷贝新版本与当前版本不同的部分。

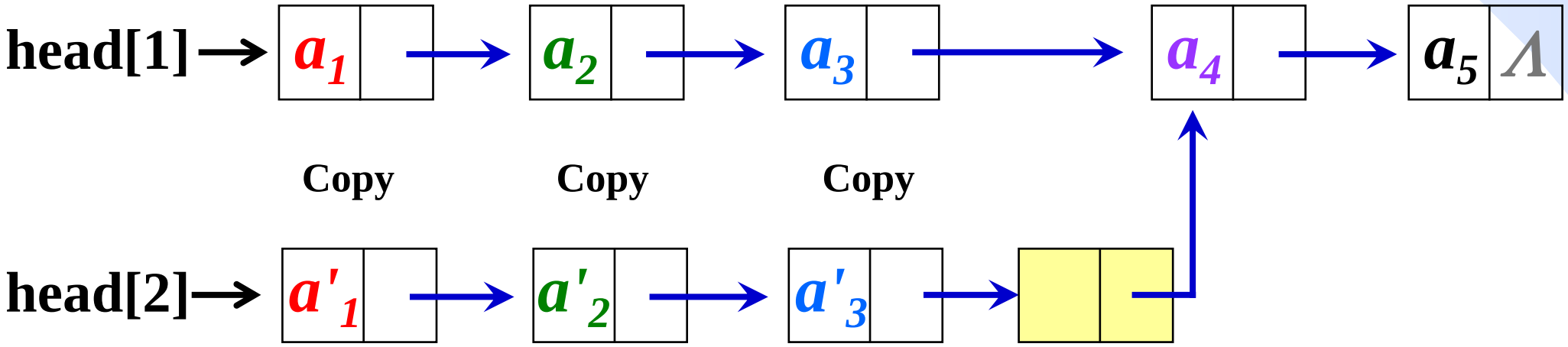


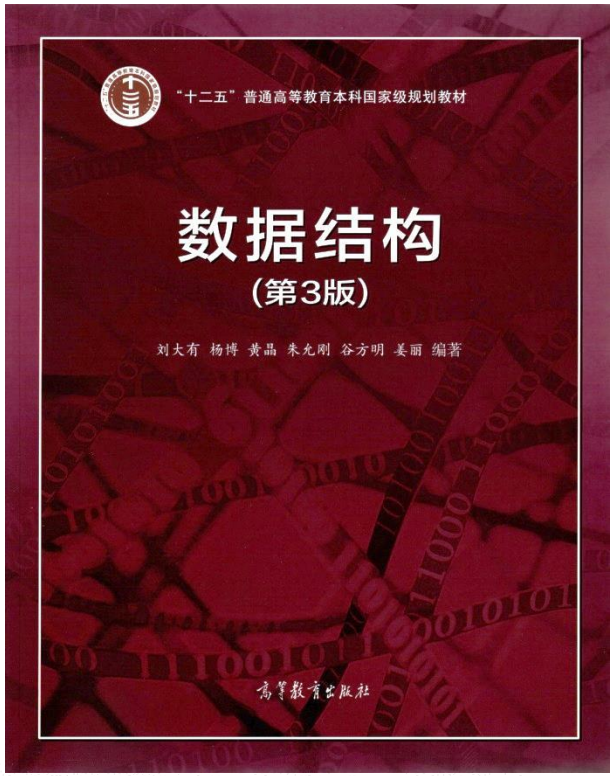
可持久化数据结构

保存数据结构的历史版本，使数据结构可以恢复到任意历史状态。

实现方法：在对数据结构进行修改前，对当前版本拷贝（备份）。

增量式备份：只拷贝新版本与当前版本不同的部分。





线性表的链接存储

- 单链表
- 循环链表
- 双向链表
- 静态链表
- 链表的双指针技巧
- 复杂链表的拷贝
- 跳跃表

数据之法
结构之美
算法之道

zhuyungang@jlu.edu.cn

Last updated on 2022.9

有序顺序表的二分查找

算法BinarySearch (*R*, *n*, *K*)

/*针对有序数组*R*的对半查找算法，R[1]≤R[2]≤…≤R[n]*/

s←1 . *e*←*n* .

WHILE *s* ≤ *e* **DO**

(*mid*←⌊*s*+(*e*-*s*)/2⌋ . // 找中间位置

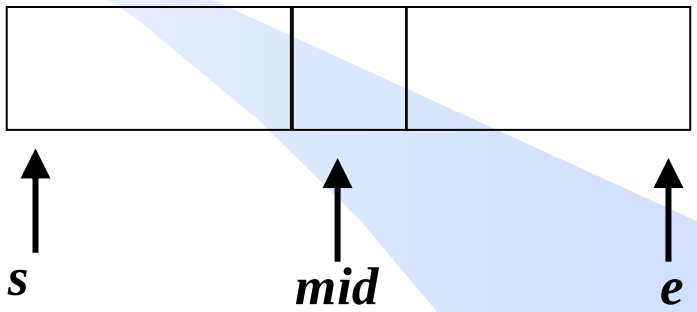
  IF *K*＜*R*[*mid*] **THEN** *e*←*mid* -1 .

  IF *K*＝*R*[*mid*] **THEN RETURN TRUE.**

  IF *K*＞*R*[*mid*] **THEN** *s*←*mid* +1

)

RETURN FALSE.

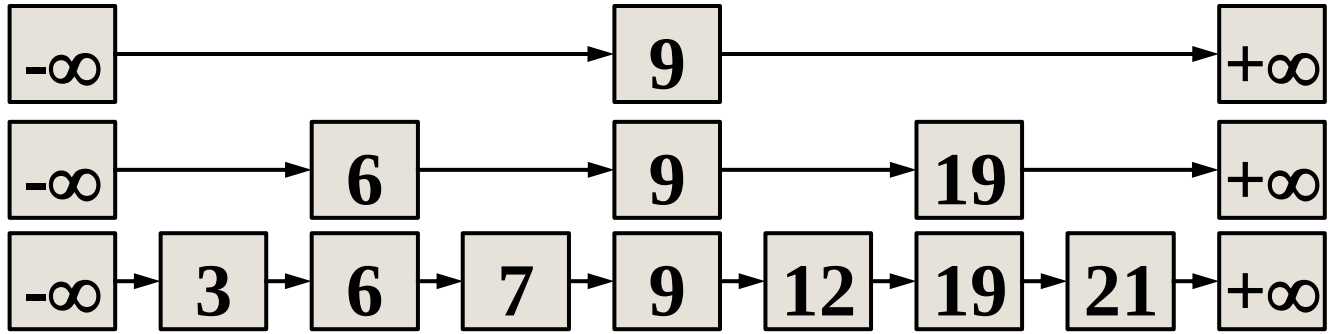


O(logn)

跳跃表 Skip list

- 动机：能否借鉴顺序表二分查找思想，提高有序单链表中查找元素的时间效率

William Pugh
康奈尔大学博士
马里兰大学教授



跳跃表 —查找

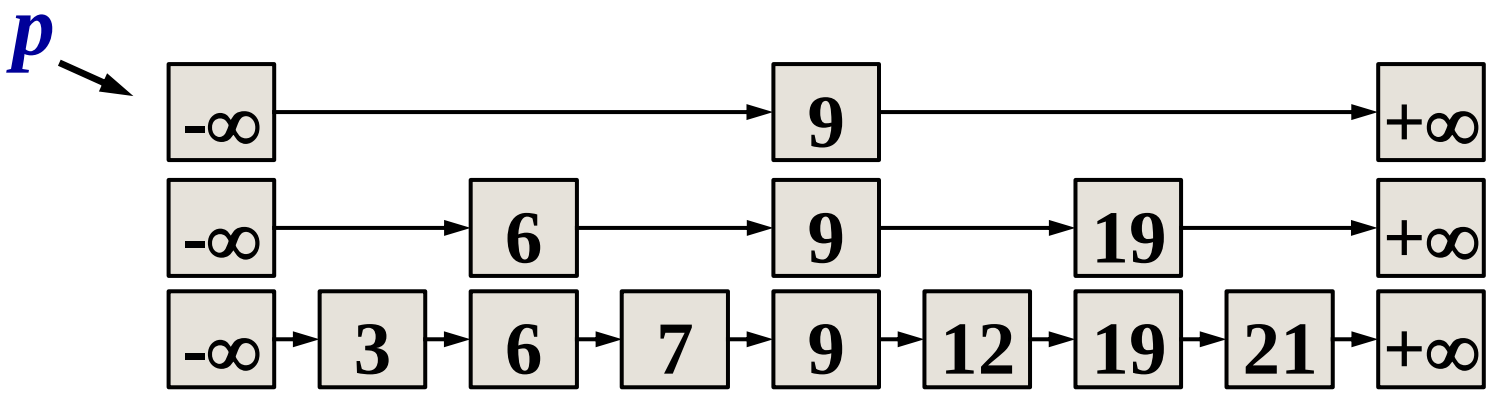
- 在跳表中查找元素*X*（例查找12）

令*p*指向最上层第一个结点，将*X*与data(next(*p*))比较

(1) *X* = data(next(*p*))： 查找成功

(2) *X* > data(next(*p*))： *p*向右移动

(3) *X* < data(next(*p*))： *p*向下移动

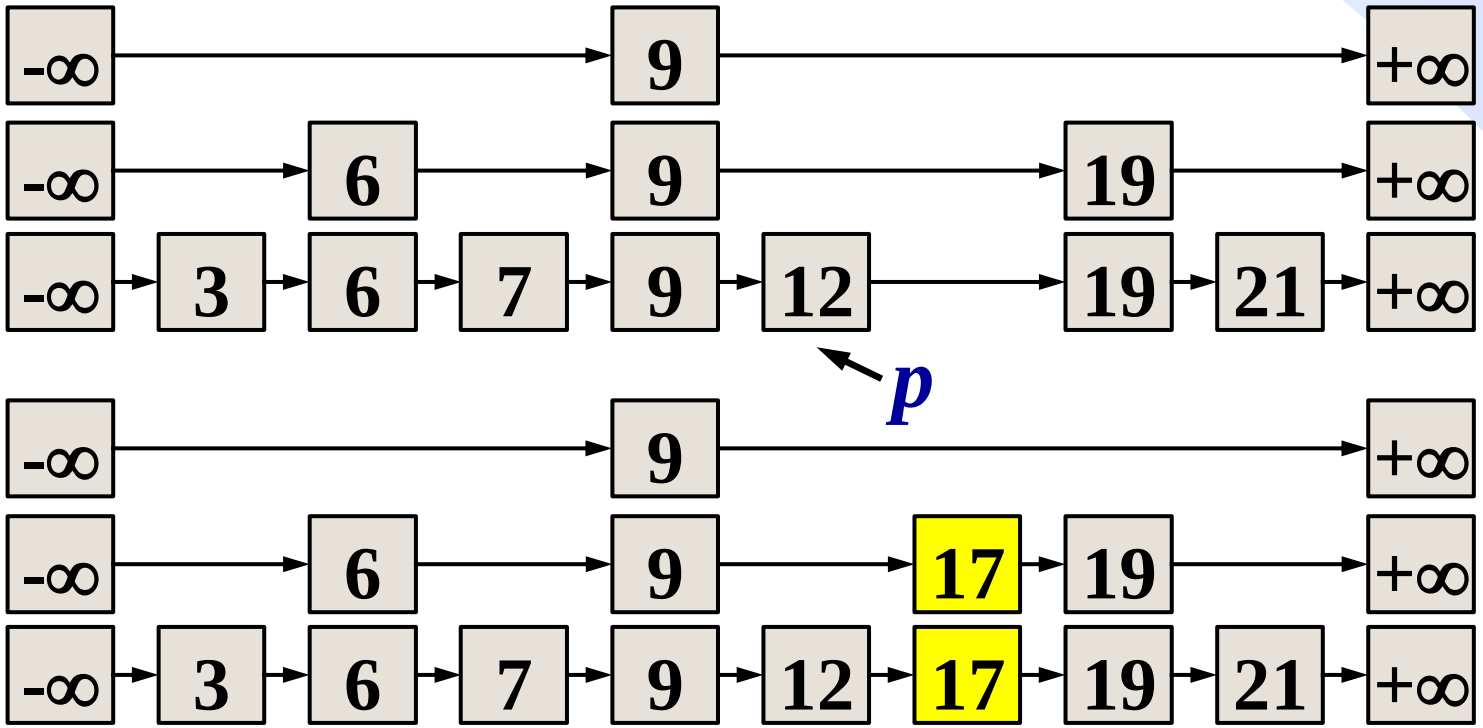


跳跃表 —插入

- 在跳表中插入元素X（例插入17）

(1) 查找X，在查找失败的位置p插入X

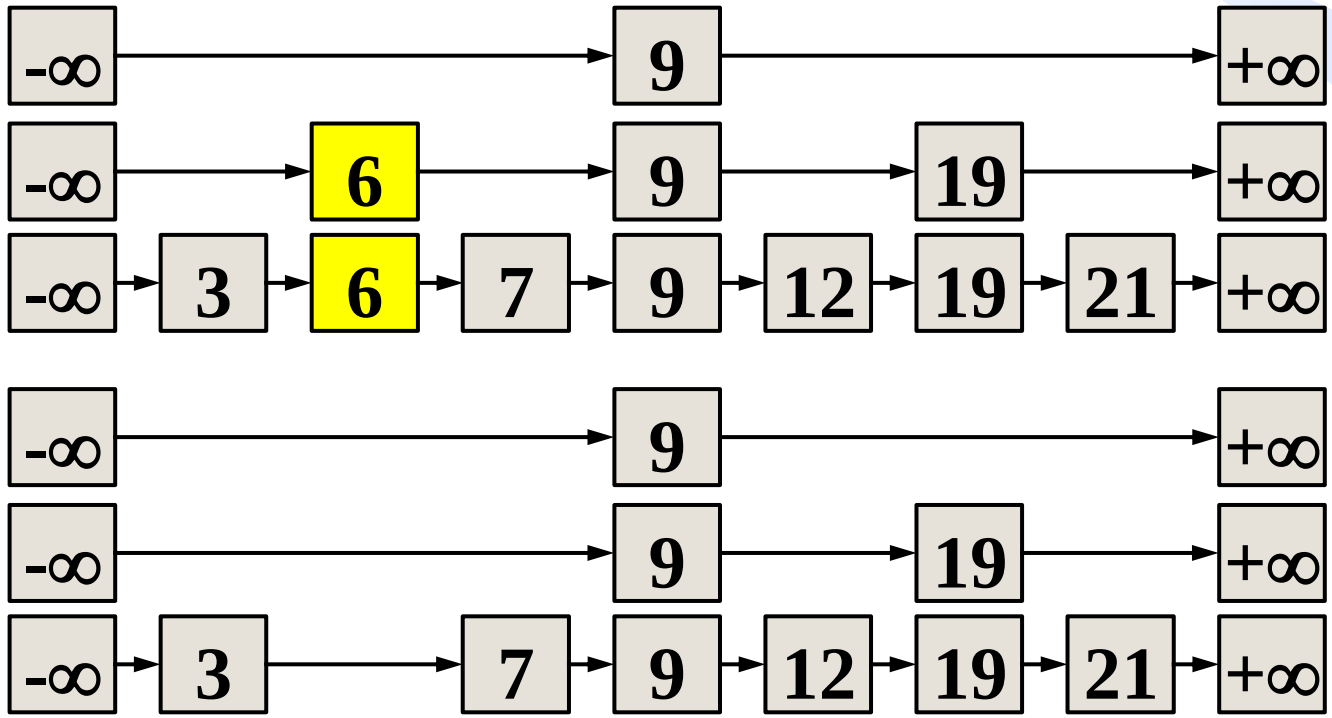
(2) 以1/2的概率（可通过抛硬币实现）向上生长一层。生长概率逐层减半



跳跃表 —删除

- 在跳表中删除元素X（例删除6）

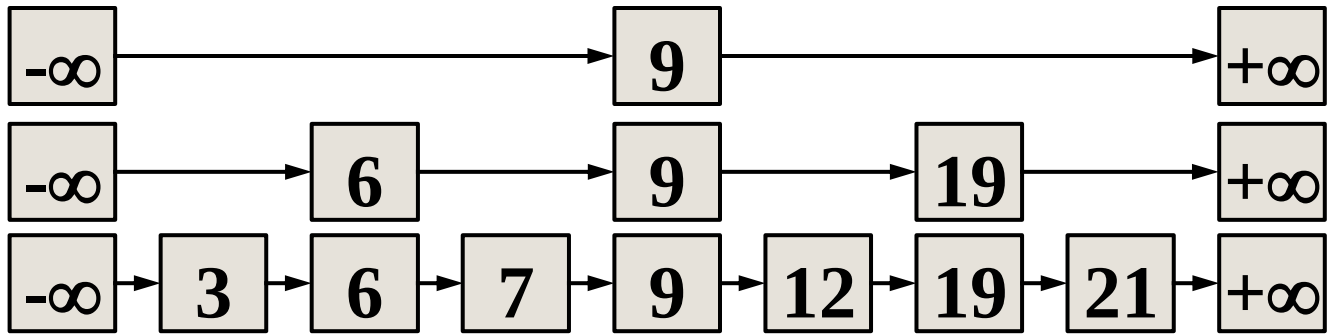
查找X，删除每一层的X。



跳跃表 —空间复杂度

- 第0层有n个结点
- 第1层期望结点个数：$n/2$
- 第2层期望结点个数：$n/4$
- 第i层期望结点个数：$\frac{n}{2^i}$
- 总结点个数

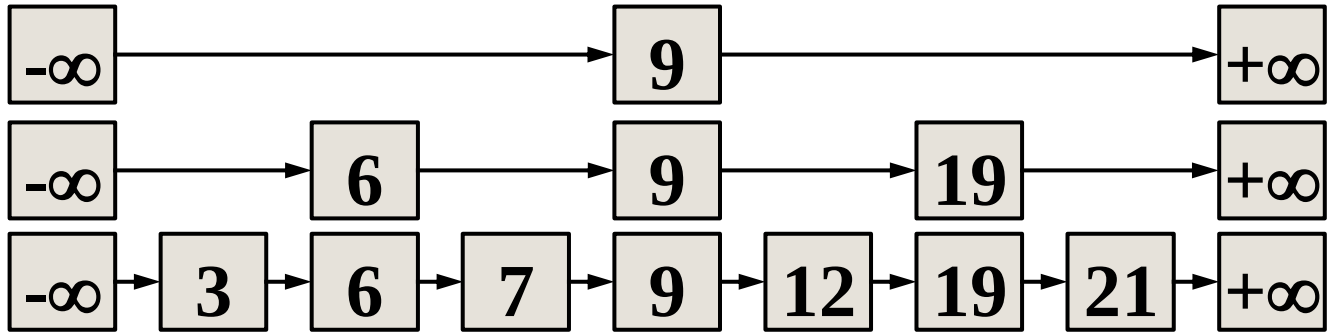
$$\sum_{i=0}^{h}\frac{n}{2^i}=n\sum_{i=0}^{h}\frac{1}{2^i}=n\frac{1-\left(\frac{1}{2}\right)^{h+1}}{1-\frac{1}{2}}=2n\left(1-\frac{1}{2^{h+1}}\right)<2n=O(n)$$



跳跃表 —时间复杂度

- 查找过程：由顶层到底层，每层元素比较次数为常数
- 时间复杂度取决于跳跃表的层数

$$h = O(\log n)$$



跳跃表 —有关性质

(1) 跳跃表的每一层都是一个有序的链表；

(2) 最底层的链表包含所有元素；

(3) 跳跃表的查找、插入、删除时间复杂度为O(logn)；

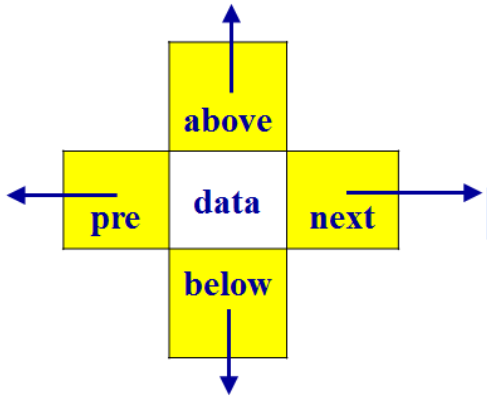
(4) 跳跃表是一种随机化的数据结构（通过抛硬币来决定层数）；

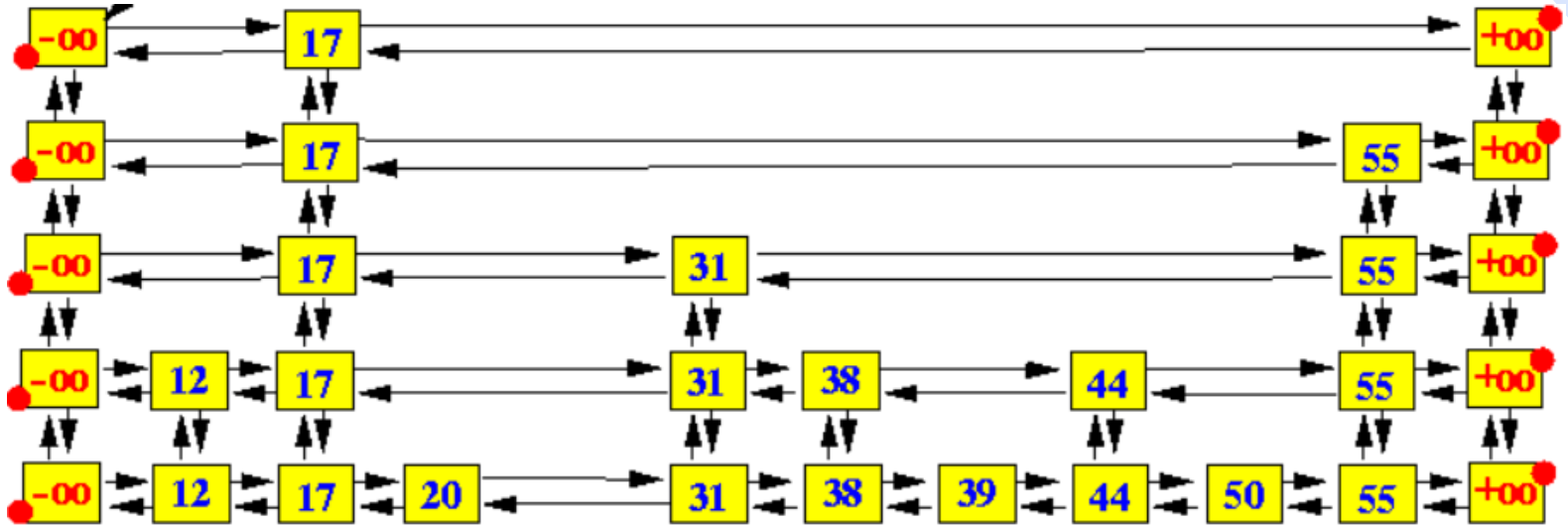
(5) 跳跃表的空间复杂度为 O(n)。

维护一组有序的数据，并且希望在查找、插入、删除等操作上尽可能快，那么跳跃表会是不错的选择。

跳跃表 —实现

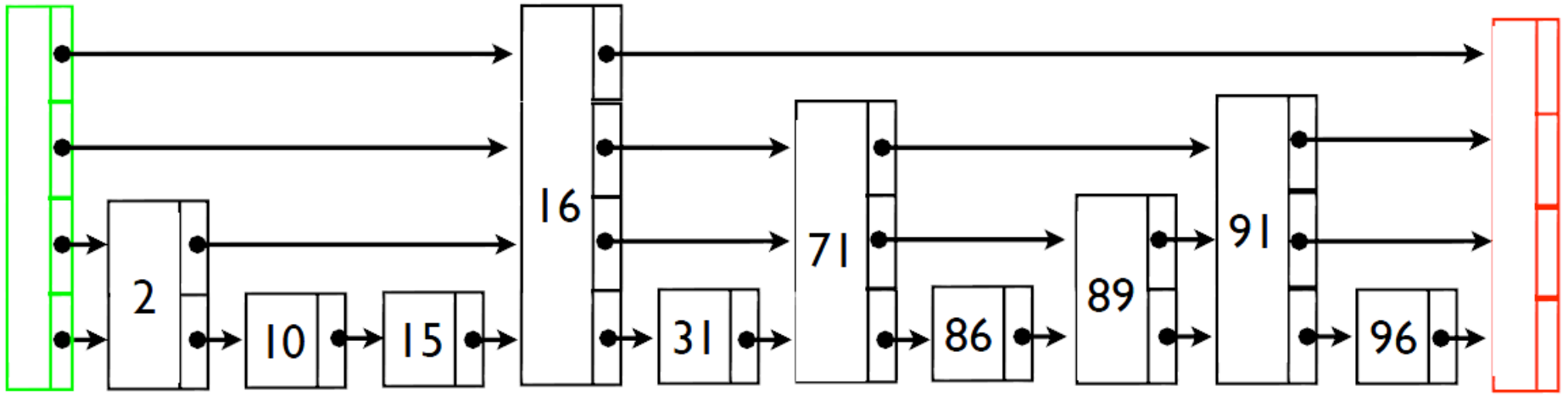
- 实现方式一：四联表





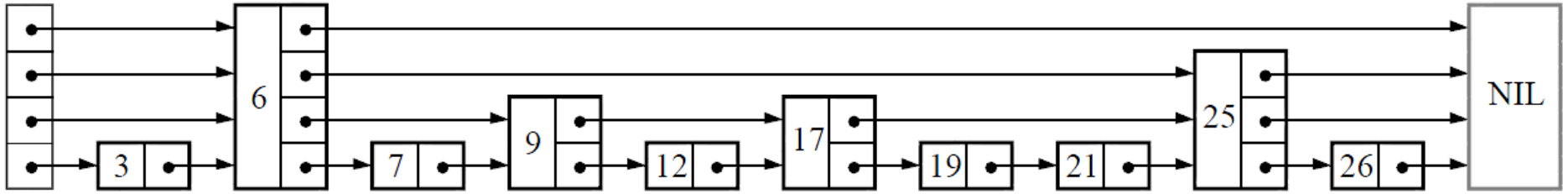
跳跃表 —实现

- 实现方式二：
- struct Skip_Node
  { int data;
    Skip_Node *next[ ];
  }

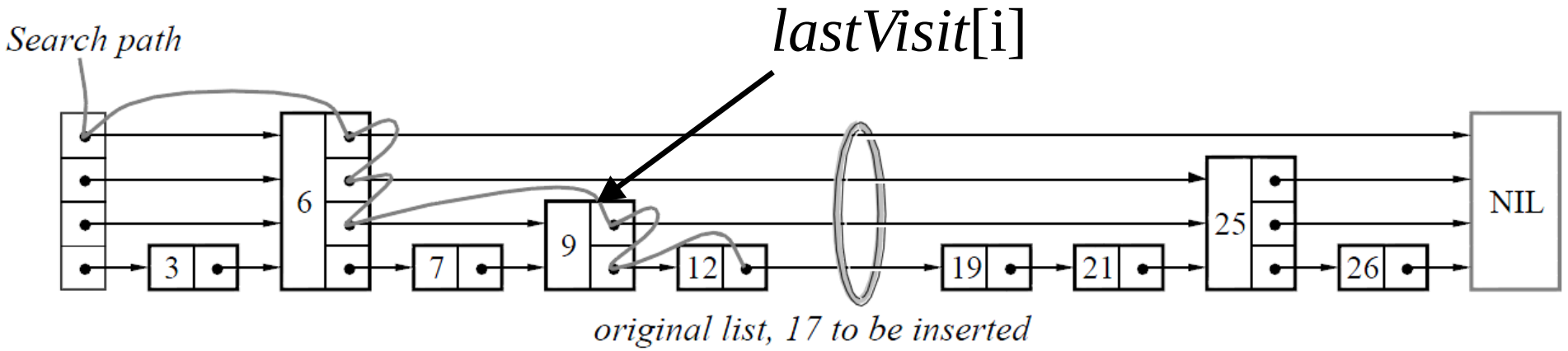


课下阅读：跳跃表—查找

```
算法 Search(head, K. p)  //查找K, 返回指针
p ← head;
FOR i= LEVEL TO 0 STEP -1 DO          //控制p下降
        WHILE data(next[i](p))< K DO  //控制p右移
                p ← next[i](p).
p ← next[0](p).
IF data(p)=K THEN RETURN p.
RETURN Λ.
```



课下阅读：跳跃表—插入



算法 Search2(head, K. p, *lastVisit*)

p ← head;

FOR i= LEVEL **TO** 0 **STEP** -1 **DO**

(**WHILE** data(next[i](p))< K **DO**

p ← next[i](p).

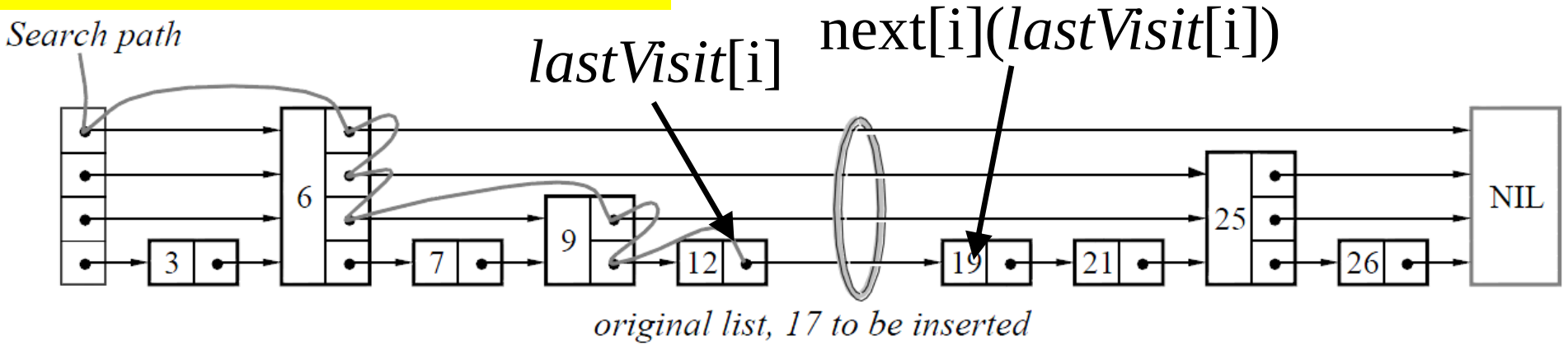
lastVisit[i] ← p.

)

p ← next[0](p). ▌

记录查找过程中每层（下降前）经过的最后一个结点

课下阅读：跳跃表—插入



算法 Insert(head, K. p)

Search2(head, K. p, *lastVisit*).

IF data(p)=K **THEN RETURN** p.

i ← 0. p ⇐**AVAIL**. data(p) ← K.

next[i](p) ← next[i](*lastVisit*[i]).

next[i](*lastVisit*[i]) ← p.

WHILE RAND( ) MOD 2=1 **AND** i < MaxLevel **DO**

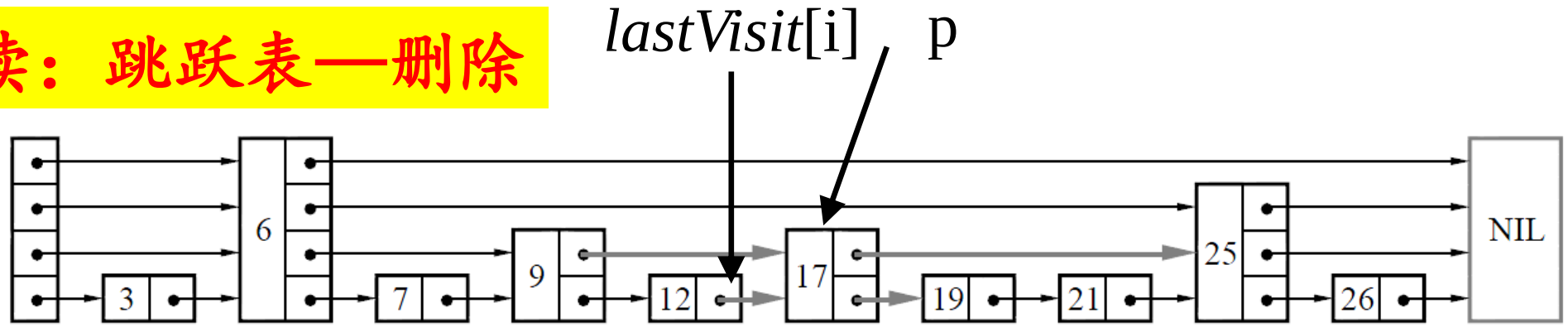
(i ← i+1.

next[i](p) ← next[i](*lastVisit*[i]).

next[i](*lastVisit*[i]) ← p.

) ▌

课下阅读：跳跃表—删除




```
算法 Delete (head, K. p)
    Search2(head, K. p, lastVisit).
    IF data(p)=K THEN
    (       i ← 0.
            WHILE next[i](lastVisit[i]) = p DO
            (           next[i](lastVisit[i]) ← next[i](p).
                        i ← i+1.
            )
            AVAIL⇐ p.
            WHILE next[i](head)=NIL DO
             (          删去第i层.
                        i ← i-1.
             )
    )
```

跳跃表应用

redis

LEVELDB

APACHE HBASE

自愿性质OJ练习题

- ✓ LeetCode 206（反转链表）
- ✓ LeetCode 92（反转链表II）
- ✓ LeetCode 19（删除链表倒数第K个结点）
- ✓ LeetCode 876（找链表的中间结点）
- ✓ LeetCode 160（链表相交及交点）
- ✓ LeetCode 141（链表判环）
- ✓ LeetCode 142（链表找环的入口）
- ✓ LeetCode 143（重排链表）
- ✓ LeetCode 234（判断链表是否是回文）
- ✓ LeetCode 138（复制带随机指针的链表）