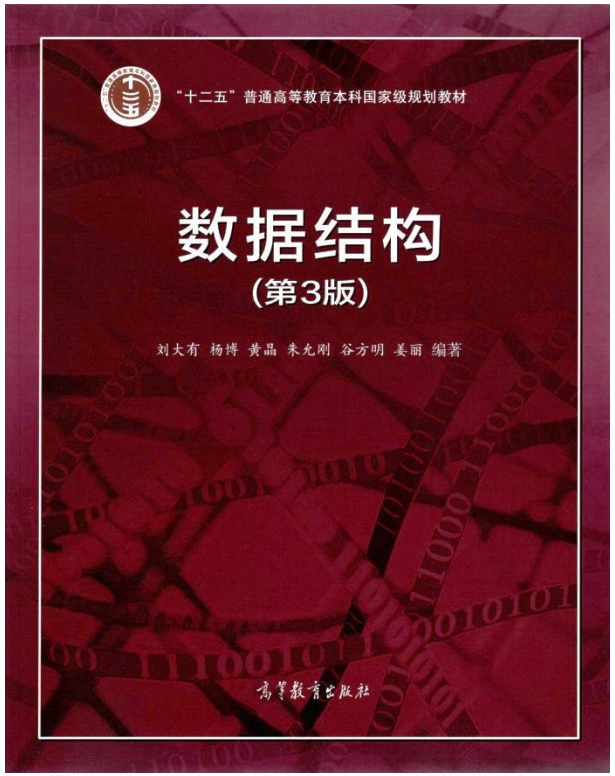
课堂实录



线性表的链接存储

- 单链表
- 循环链表
- 双向链表
- 静态链表
- 侵入式链表
- 链表的双指针技巧

数据之法
结构之美
算法之道

zhuyungang@jlu.edu.cn

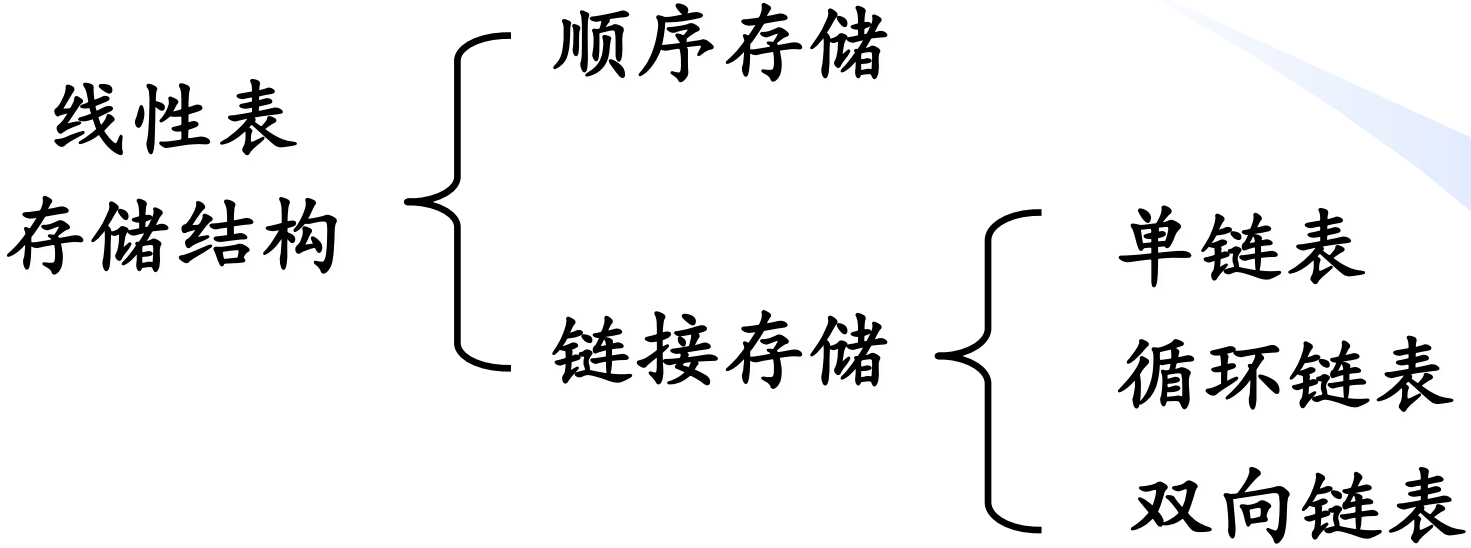
Last updated on 2023.9

The best way to learn swimming is swimming,
the best way to learn programming is programming.

下周课之前慕课自学内容

- 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

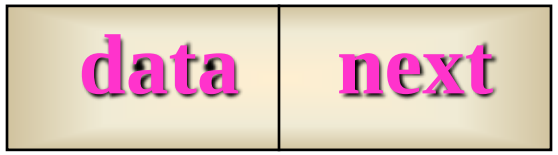
线性表的存储结构



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。

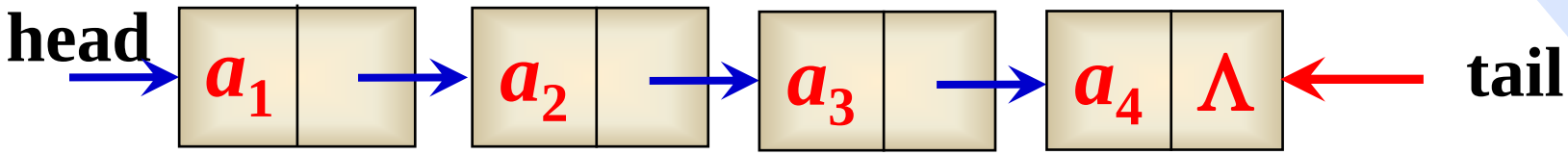
单链表（Singly Linked List）

◆单链表的结点结构：




```
struct Node{
    int data;
    Node* next;
};
```

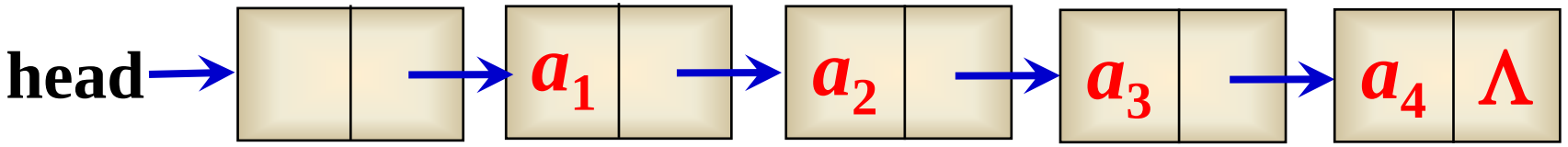
单链表的定义：每个结点只含一个链接域的链表叫单链表。



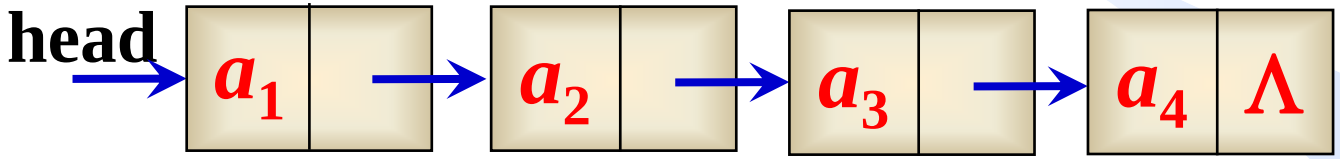
◆链表的第一个结点被称为头结点(也称为表头)，指向头结点的指针被称为头指针(head).

◆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).

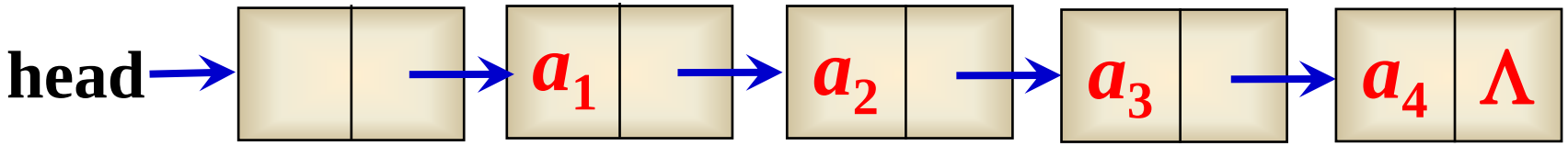
- 为了对表头结点插入、删除等操作的方便，通常在表的前端增加一个特殊的表头结点，称其为哨位（哨兵）结点 。
- 哨位结点不被看作表中的实际结点，我们在讨论链表中第k个结点时均指第k个实际的表结点。
- 表的长度：非哨位结点的个数。若表中只有哨位结点，则称其为空链表，即表长度为0



- 哨位结点作用:简化边界条件的处理。
- 例：删除结点
  - ✓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. ▌
// 若i=k则p即为所求，返回p;
// 若p=Λ 表示链表长度不足k（无第k个结点），返回Λ;
```

```
//返回链表第k个结点的地址
Node* Find (Node* head, int k){
    if(k<0) return NULL;  //输入的k位置不合法
    Node* p=head; //令指针p指向哨位结点
    for(int i=0; p!=NULL && i<k; i++) //找第k个结点
        p=p->next;
    return p;
}
//若i==k 则p即为所求，返回p;
//若p==NULL 表示链表长度不足k（无第k个结点），返回NULL;
```

```
Node* Search (Node* head, int item){
    //在链表中查找字段值为item的结点并返回其指针
    Node* p=head->next;  //令指针p指向第1个结点
    //遍历
    while(p!=NULL && p->data!=item)
        p=p->next; //扫描下一个结点
    return p;
}
//若p->data==item则p即为所求，返回p
//若p==NULL表示无结点item，返回NULL
```

```
Node* Search (Node* head, int item){
    //另一种写法，用for循环遍历链表
    for(Node* p=head->next; p!=NULL; p=p->next)
        if(p->data==item) return p;
    return NULL;
}
```

```
void Delete (Node* head, int k){
    // 删除链表中第k个结点
    if(k < 1) return;    // 输入k不合法，不能删哨位
    Node* p = Find(head, k-1);//找第k-1结点，由p指向
    if(p==NULL || p->next==NULL)
        return;    //无第k-1个结点或只有k-1个结点
    //删除第k个结点，未完待续
```

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。

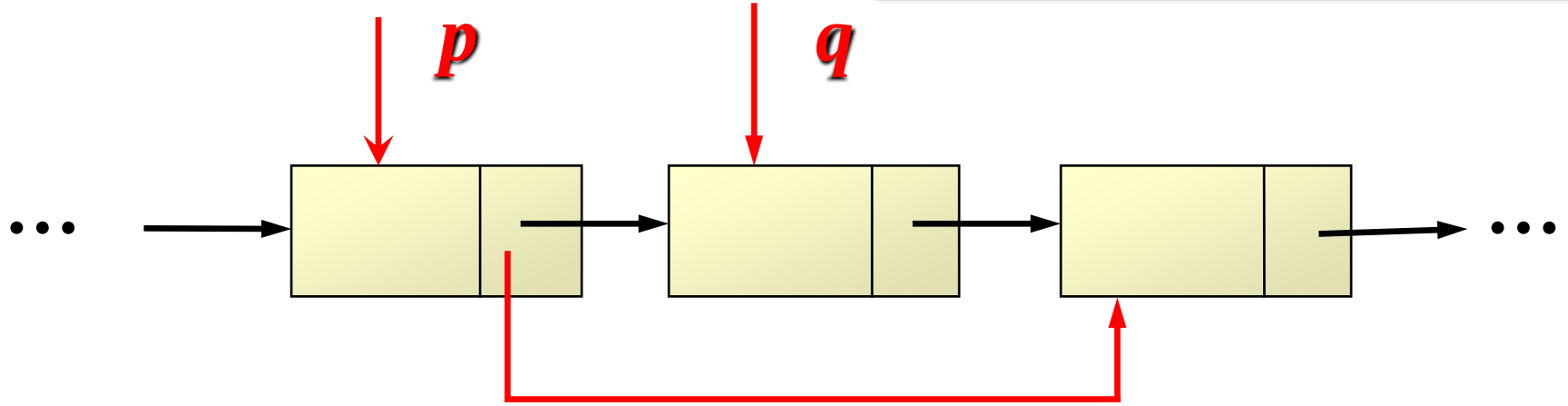
```
q = p->next;

p->next = q->next;

delete q;
```

- 在ADL语言中，释放不用的空间则用语句“AVAIL⇐q”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“q⇐AVAIL”来描述。

- 删除：



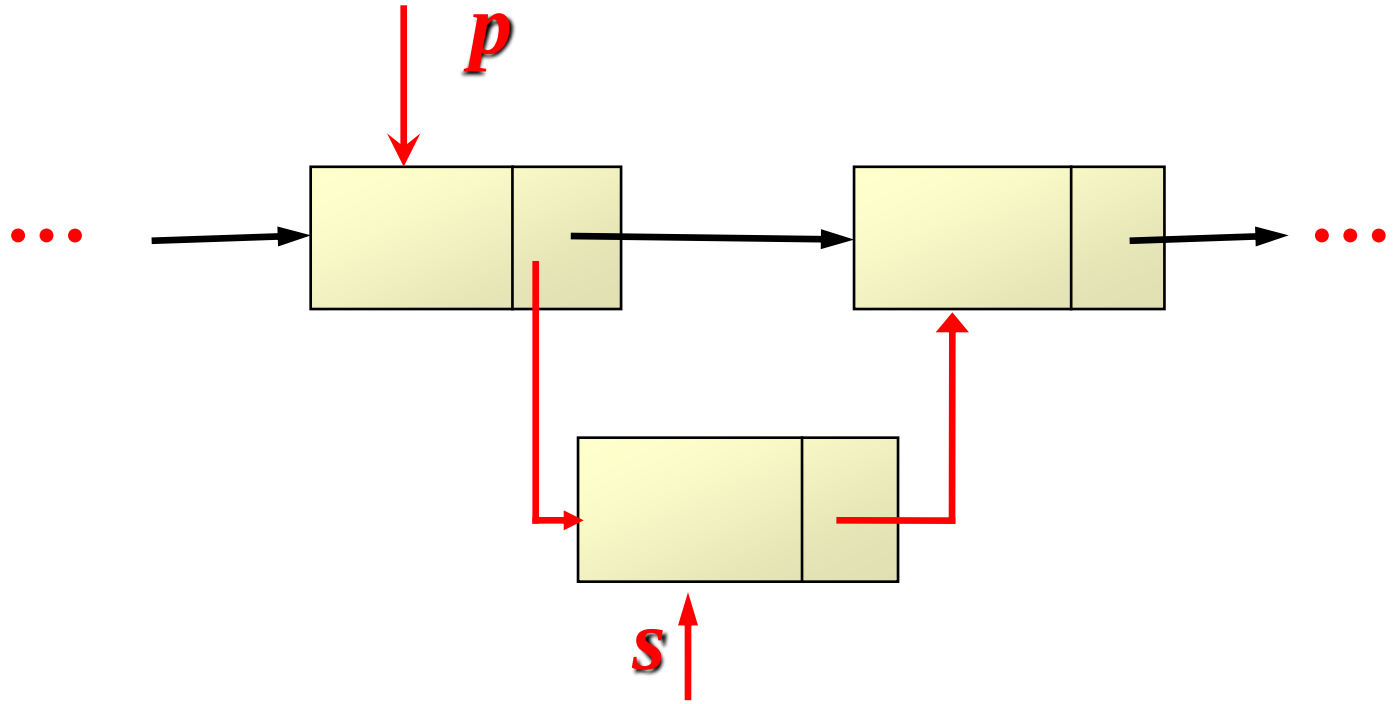
```
void Delete (Node* head, int k){
    //删除链表中第k个结点
    if(k < 1) return;    //输入k不合法，不能删哨位
    Node* p = Find(head, k-1);//找第k-1结点，由p指向
    if(p==NULL || p->next==NULL)
        return;                  //无第k-1个结点或只有k-1个结点
    //删除第k个结点
    Node* q = p->next;
    p->next = q->next;   // 修改p的next指针
    delete q;                    //释放q存储空间
}
```

```
void Insert (Node* head, int k, int 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) return;                 //插入位置不合法
    Node* p=Find(head, k);          //找第k结点
    if(p==NULL) return;             //无第k个结点
    //插入新结点
    Node* s=new Node;               //生成新结点s
    s->data=item;                   //未完待续.....
```

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。

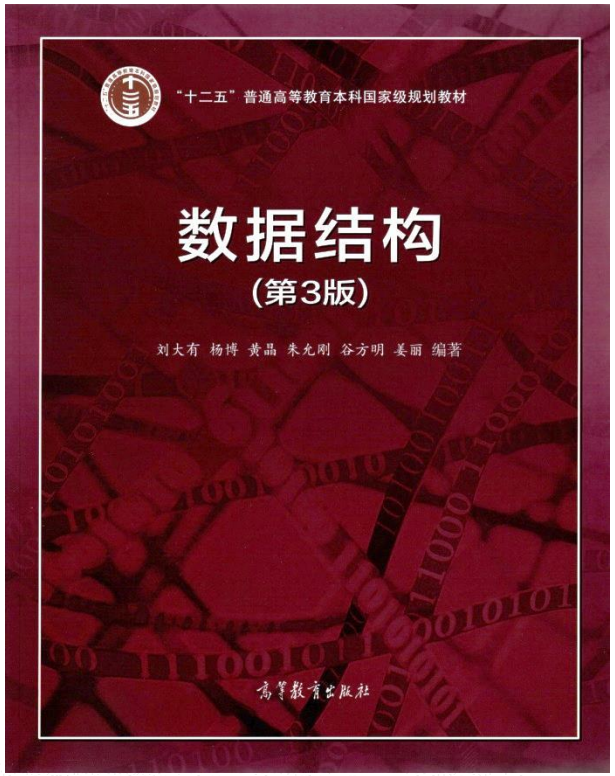
- 插入：

```
s->next = p->next;
p->next = s;
```



```
void Insert (Node* head, int k, int 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) return;                  //插入位置不合法
    Node* p=Find(head, k);           //找第k结点
    if(p==NULL) return;              //无第k个结点
    //插入新结点
    Node* s=new Node;                //生成新结点s
    s->data=item;
    s->next = p->next;               //s的next指针指向p的后继
    p->next = s;                     //p的next指针指向s
}
```

课堂实录



线性表的链接存储

- 单链表
- 循环链表
- 双向链表
- 静态链表
- 侵入式链表
- 链表的双指针技巧

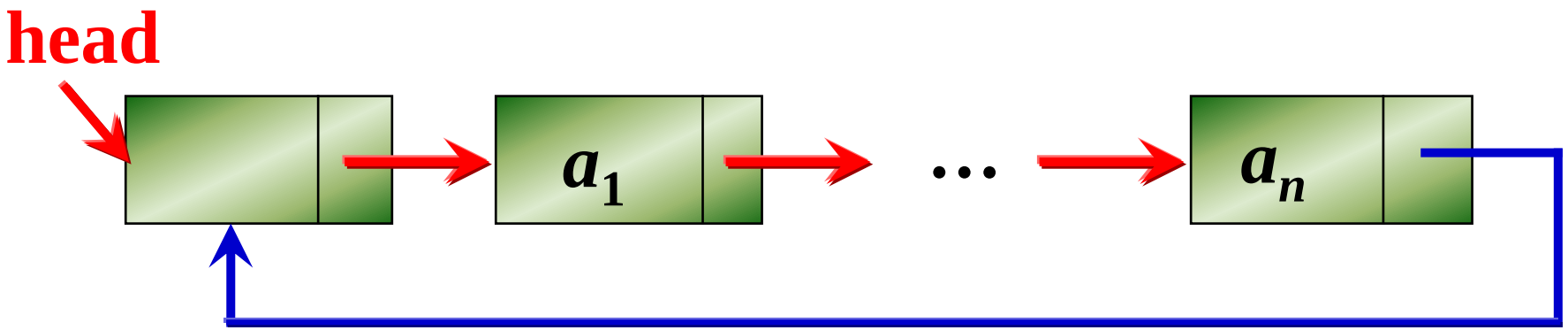
数据之法 结构之美 算法之道

zhuyungang@jlu.edu.cn

Last updated on 2023.9

循环链表（Circular Linked List）

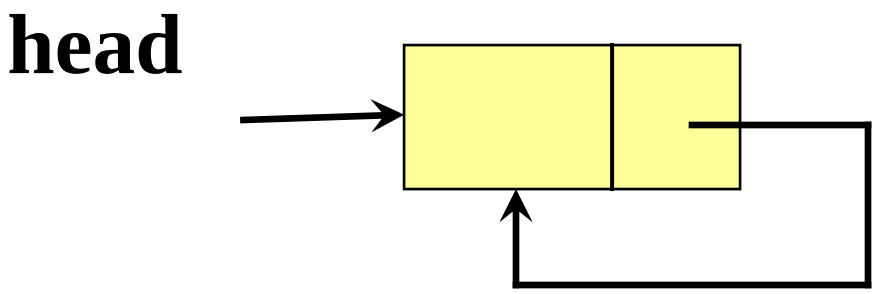
- 把链接结构“循环化”，即把表尾结点的next域存放指向哨位结点的指针，而不是存放空指针NULL（即Λ），这样的单链表被称为循环链表。
- 循环链表使我们可从链表的任何位置开始，访问链表中的任一结点。



判断空表的条件（假设包含哨位结点）：

单链表： `head->next == NULL`

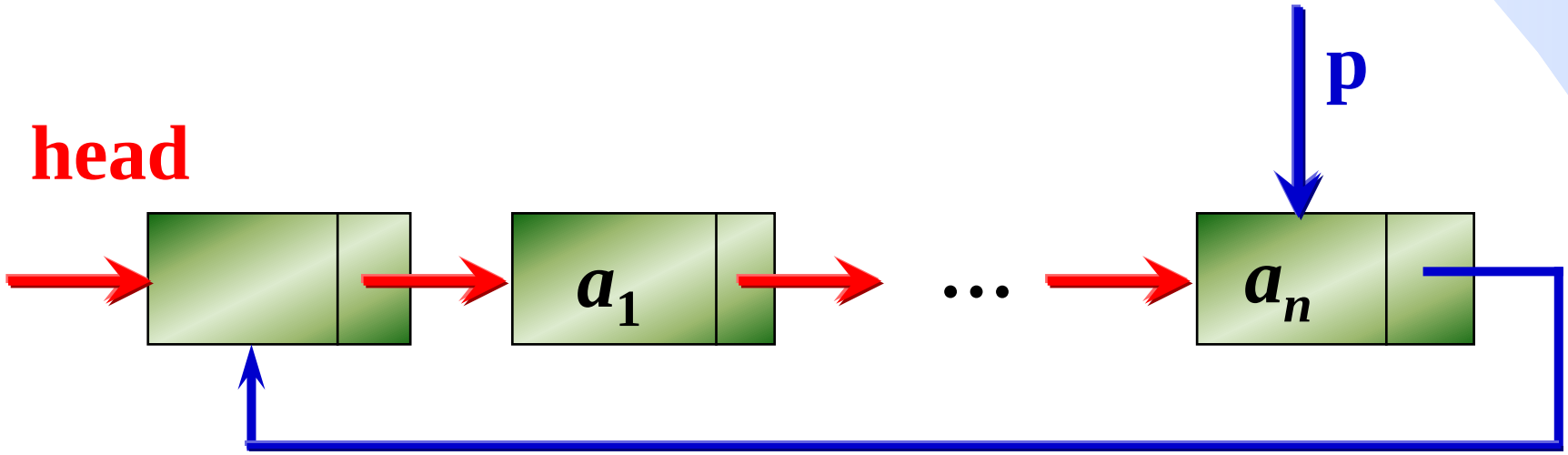
循环链表： `head->next == head`

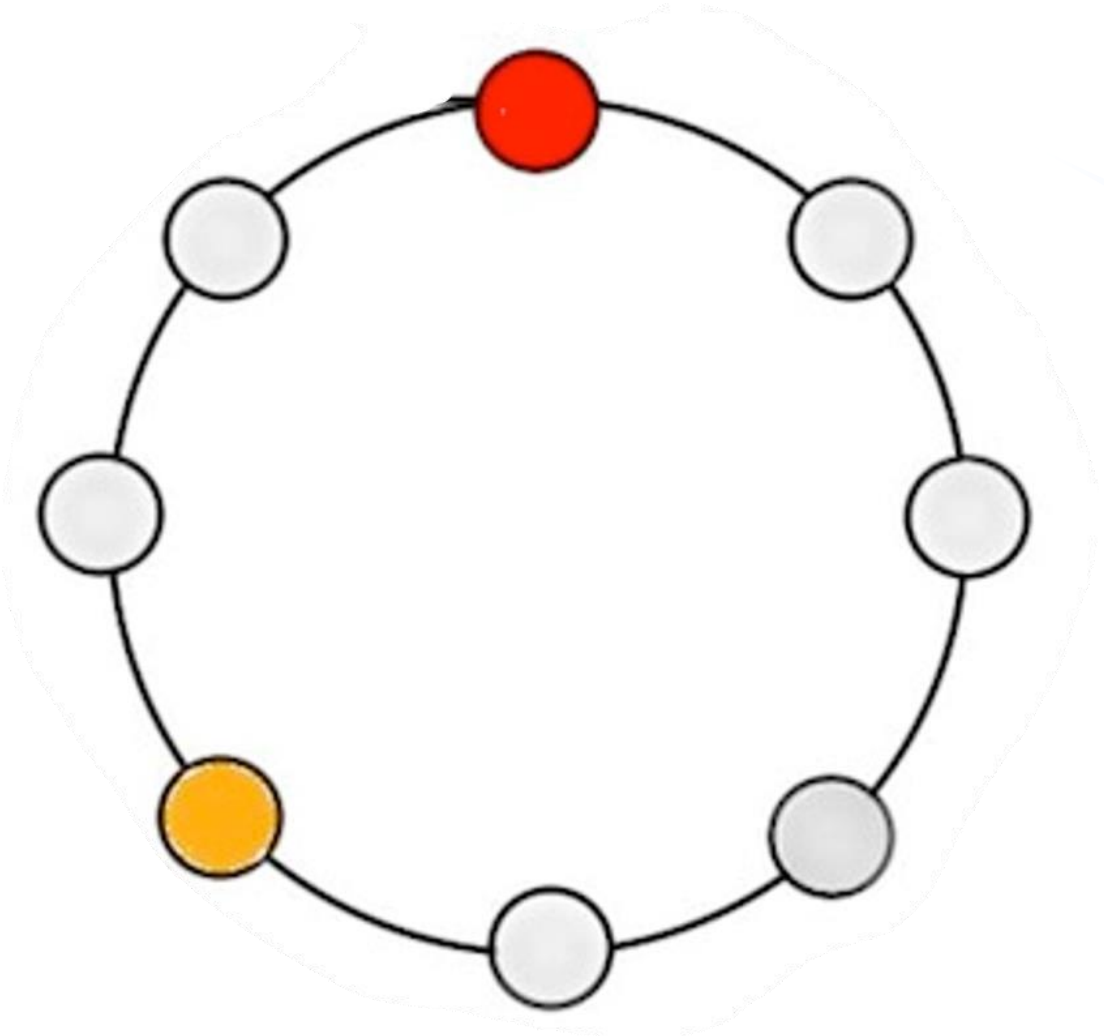


判断表尾的条件：

单链表： `p->next == NULL`

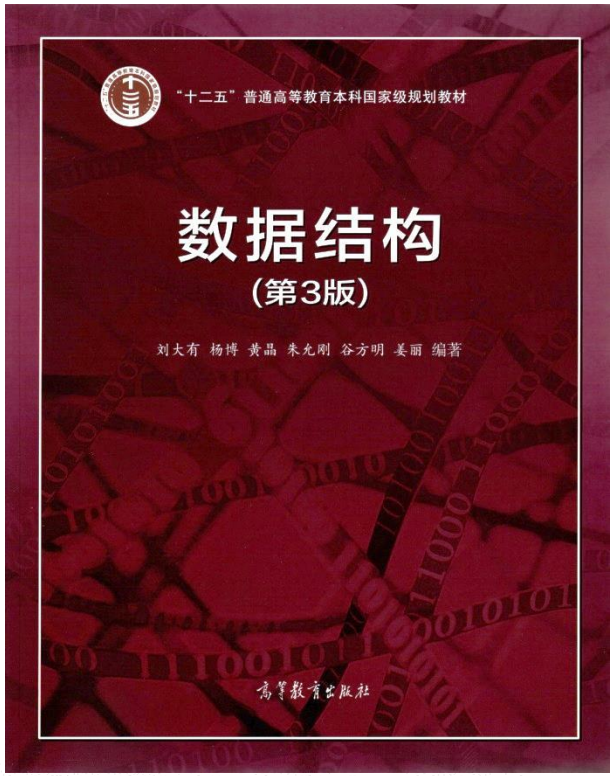
循环链表： `p->next == head`





若待解决的问题所涉及的线性结构是环形的，可以考虑循环链表

课堂实录



线性表的链接存储

- 单链表
- 循环链表
- **双向链表**
- 静态链表
- 侵入式链表
- 链表的双指针技巧

数据之法 结构之美 算法之道

zhuyungang@jlu.edu.cn

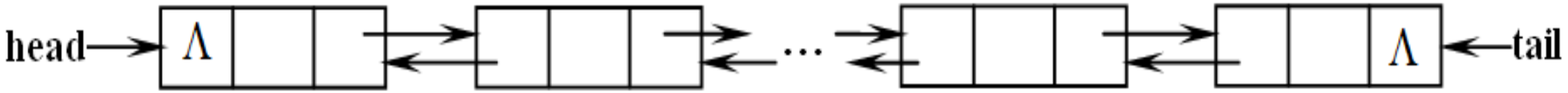
Last updated on 2023.9

双向链表（Doubly Linked List）

- 每个结点有两个指针域
- *prev*指针指向其前驱，*next*指针指向其后继；
- 优点：方便找结点的前驱。

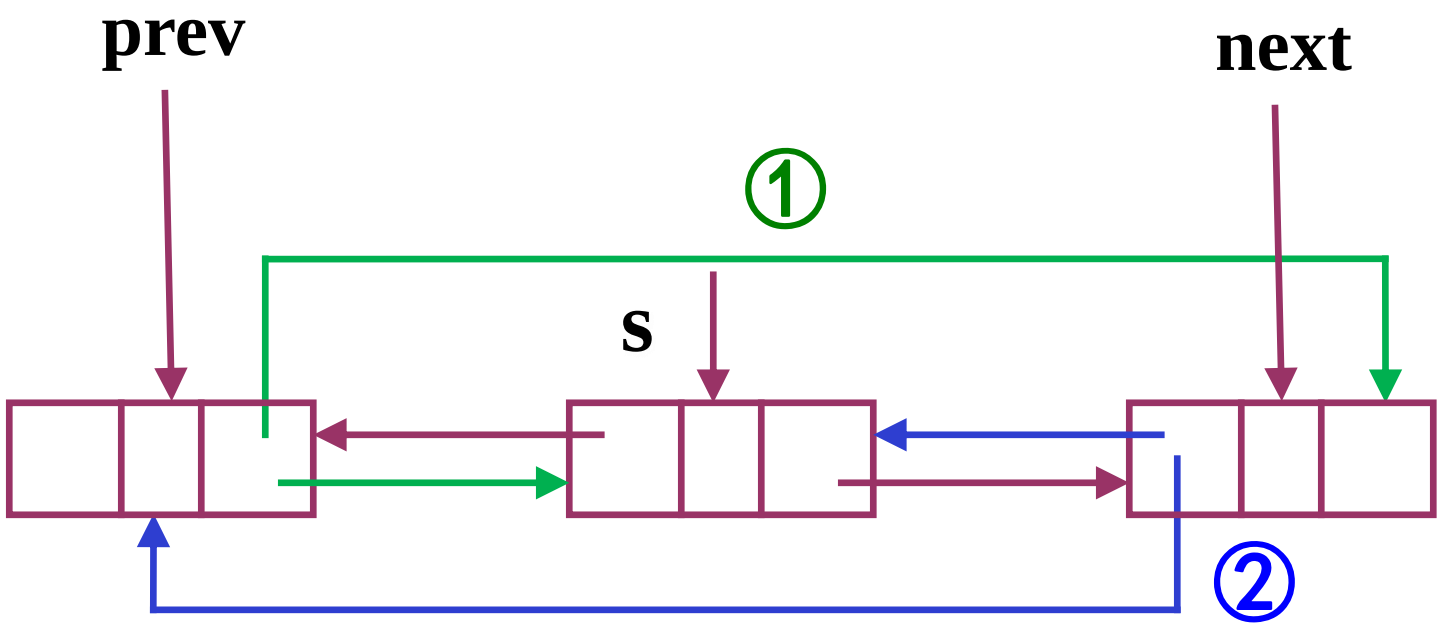
prev	*data*	*next*

```
struct Node{
    int data;
    Node* prev;
    Node* next;
};
```



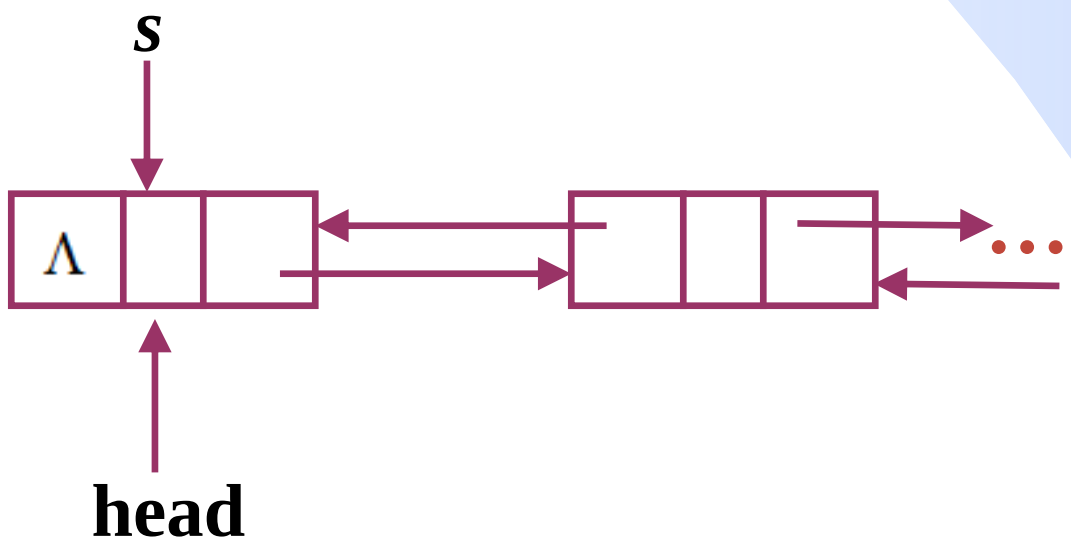
删除结点s 常规情况

```
Node* prev = s->prev;
Node* next = s->next;
prev->next = next;   //步骤①
next->prev = prev;   //步骤②
delete s;
```



删除结点s 如果s是第1个结点?

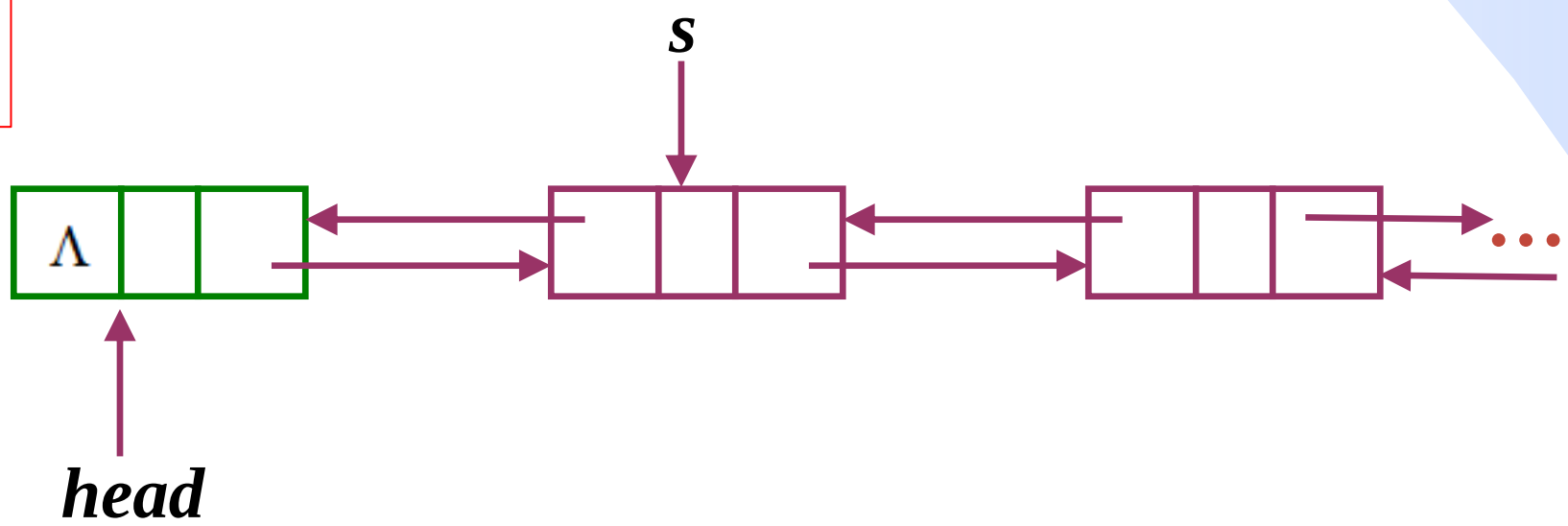
```
Node* prev = s->prev;
Node* next = s->next;
prev->next = next;
next->prev = prev;
delete s;
```



删除结点s 如果s是第1个结点?

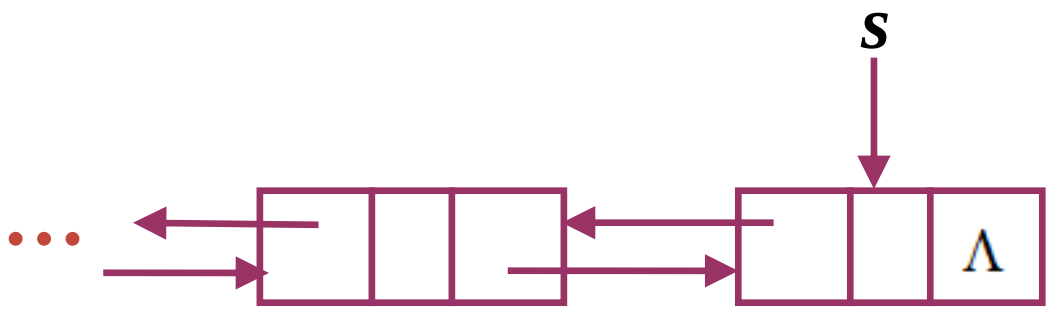
```
Node* prev = s->prev;
Node* next = s->next;
prev->next = next;
next->prev = prev;
delete s;
```

引入表头
哨位结点



删除结点s 如果s是最后1个结点?

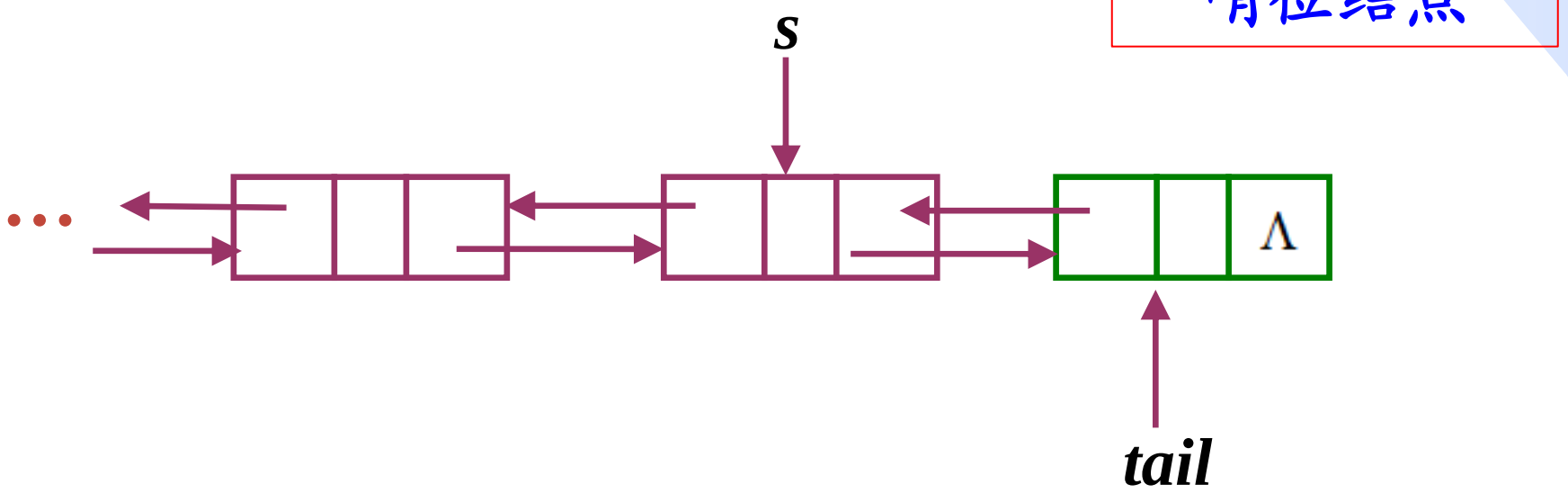
```
Node* prev = s->prev;
Node* next = s->next;
prev->next = next;
next->prev = prev;
delete s;
```



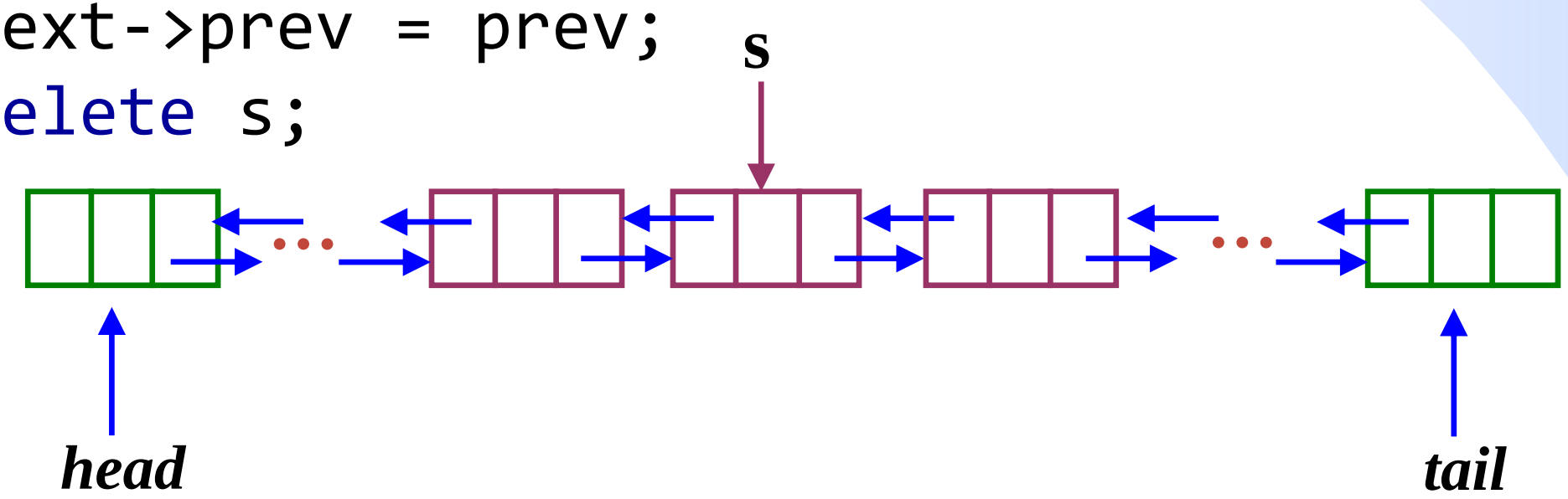
删除结点s 如果s是最后1个结点?

```
Node* prev = s->prev;
Node* next = s->next;
prev->next = next;
next->prev = prev;
delete s;
```

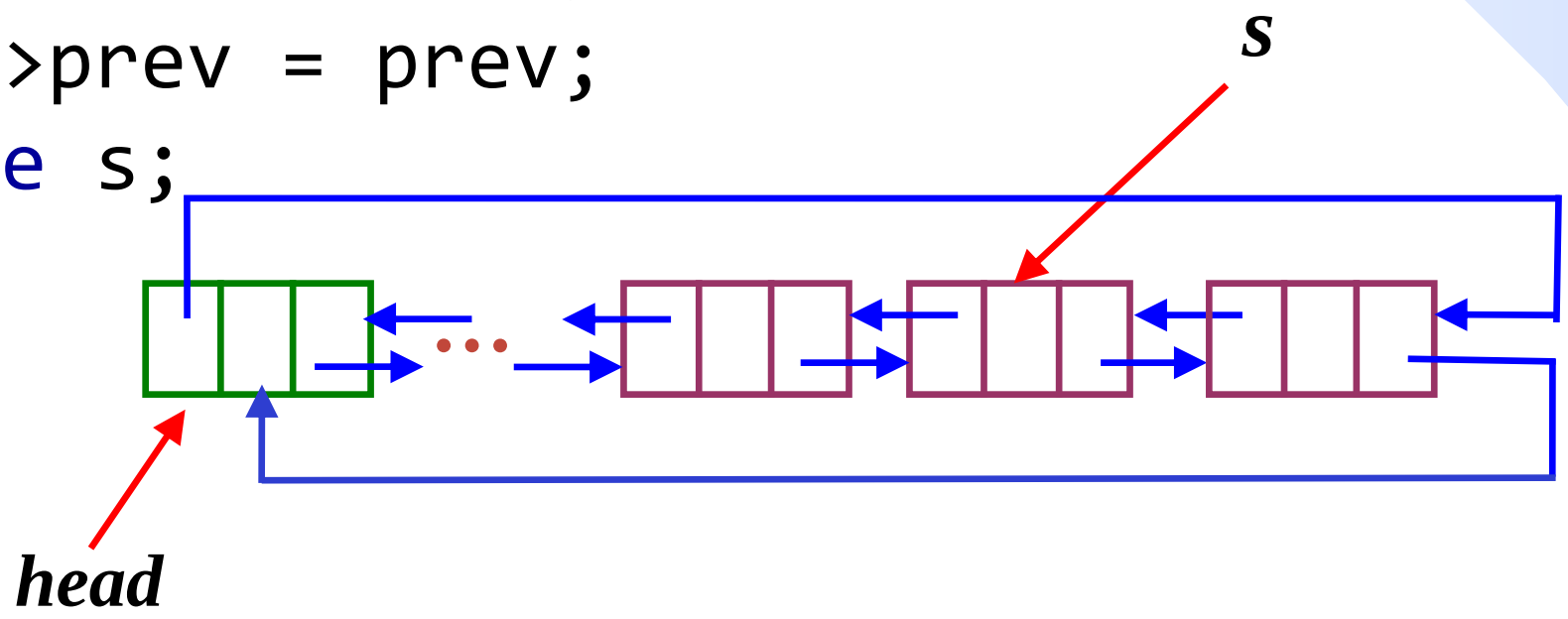
表尾引入
哨位结点



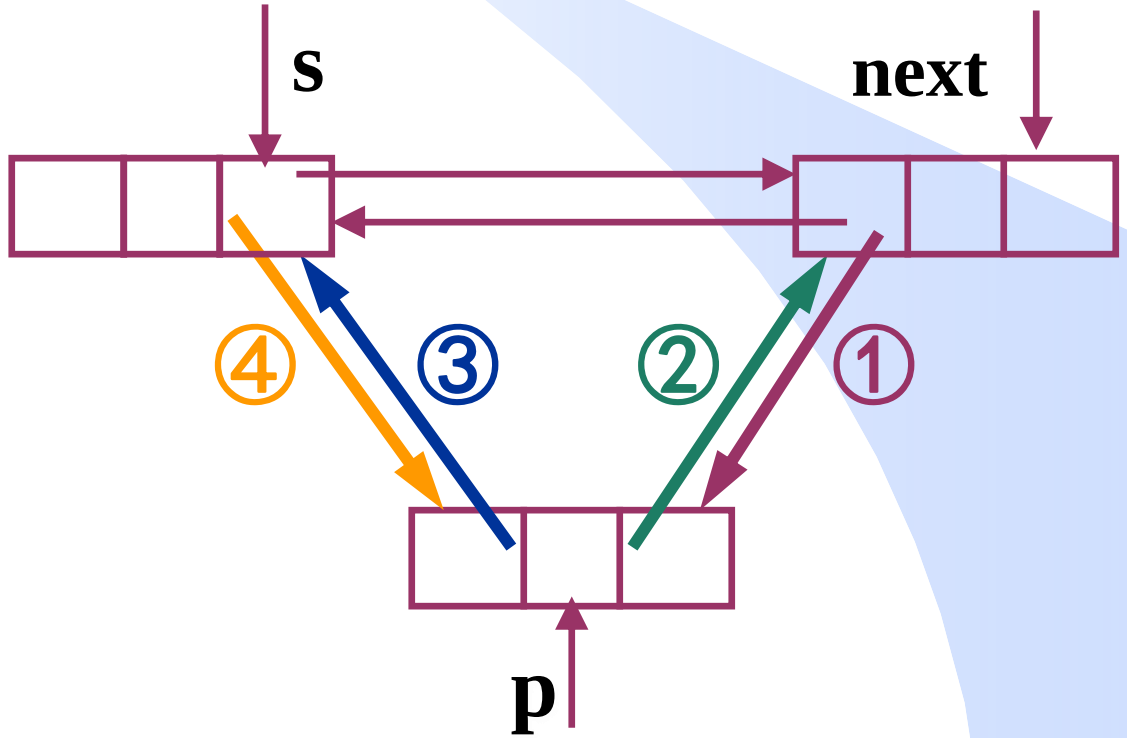
```
void DeleteNode(Node* head, Node* tail, Node* s){
    // 删除双向链表（带双哨位结点）中的非哨位结点s
    //假定head、tail和s非空
    Node* prev = s->prev;
    Node* next = s->next;
    prev->next = next;
    next->prev = prev;
    delete s;
}
```



```
void DeleteNode(Node* head, Node* s){
    //删除双向循环链表（带哨位结点）中的非哨位结点s
    //假定head和s非空
    Node* prev = s->prev;
    Node* next = s->next;
    prev->next = next;
    next->prev = prev;
    delete s;
}
```



```
void InsertNode(Node* head, Node* s, Node* p){
    //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
    //假定head、s、p非空
    Node* next = s->next;
    next->prev = p;  //步骤①
    p->next = next;  //步骤②
    p->prev = s;     //步骤③
    s->next = p;     //步骤④
}
```



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。

顺序存储和链式存储的比较

2、时间效率的比较

- 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低，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

本方子弹链表，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

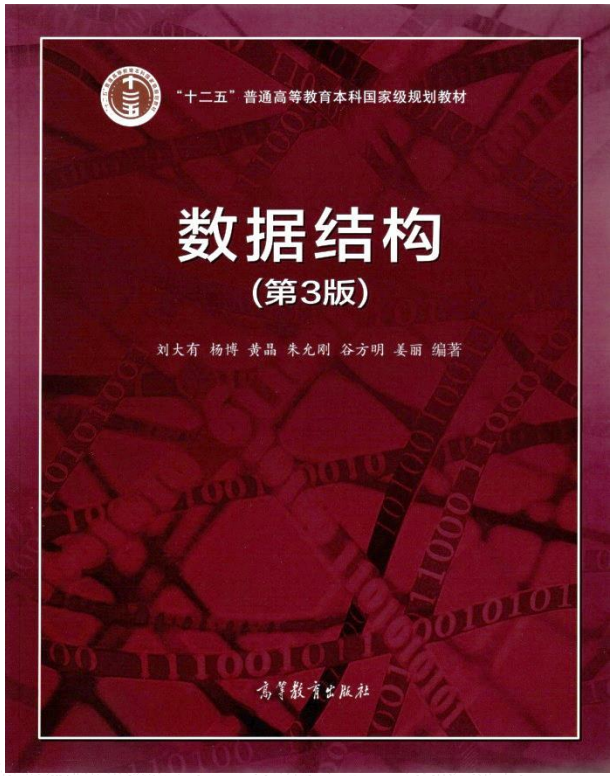
```
struct Node{
    int x, y;
    Node *next;
    ……
};
Node *pBullet, *pEnemy;
```

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).
)
```

课堂实录



线性表的链接存储

- 单链表
- 循环链表
- 双向链表
- **静态链表**
- 侵入式链表
- 链表的双指针技巧

数据之法
结构之美
算法之道

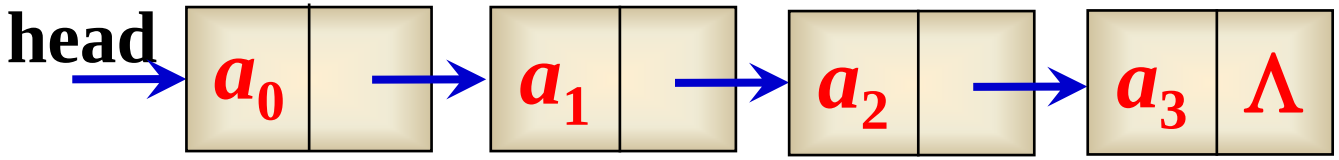
zhuyungang@jlu.edu.cn

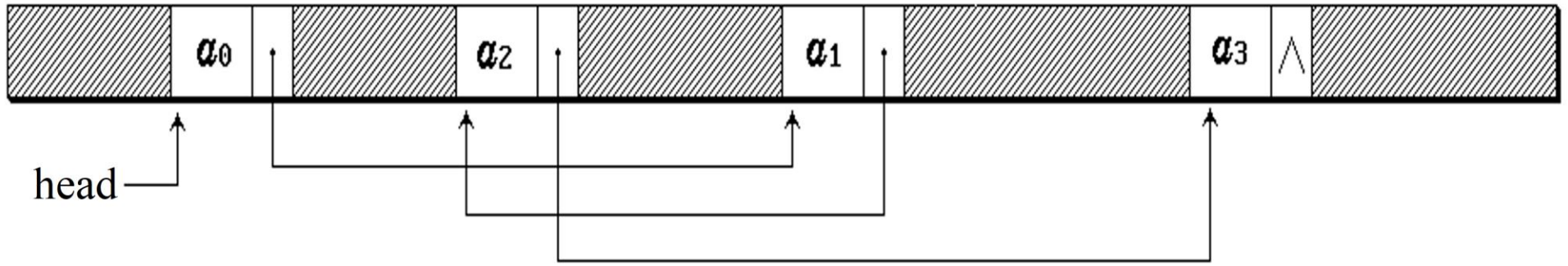
Last updated on 2023.9

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

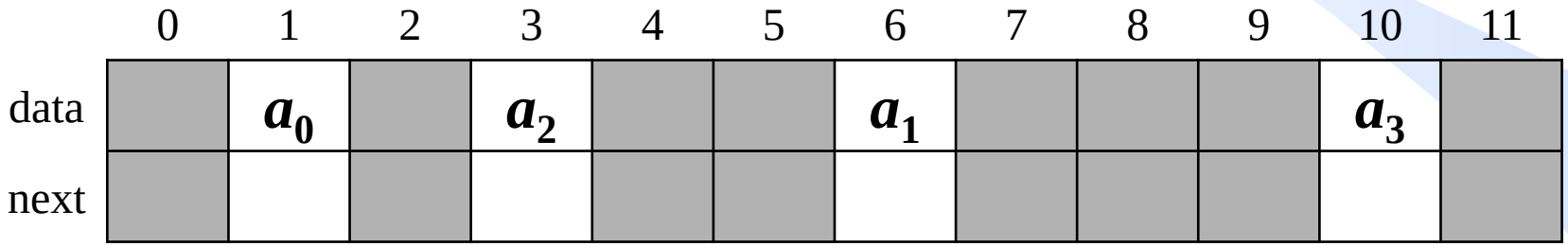


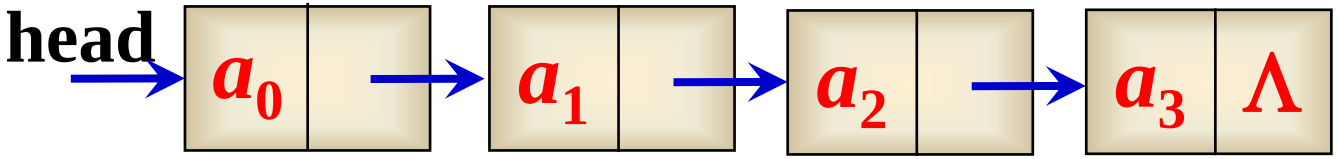


静态链表

```
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。

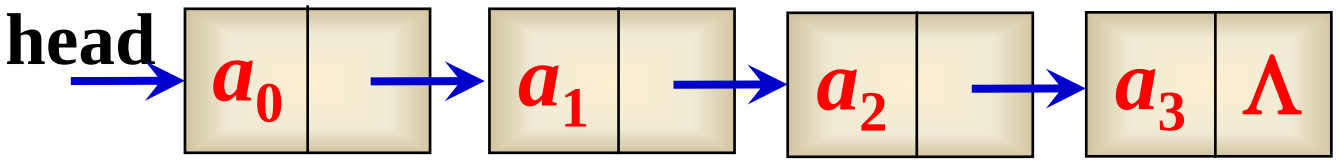




静态链表

```
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，在有指针的编程语言中，也有广泛的应用

例：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）：

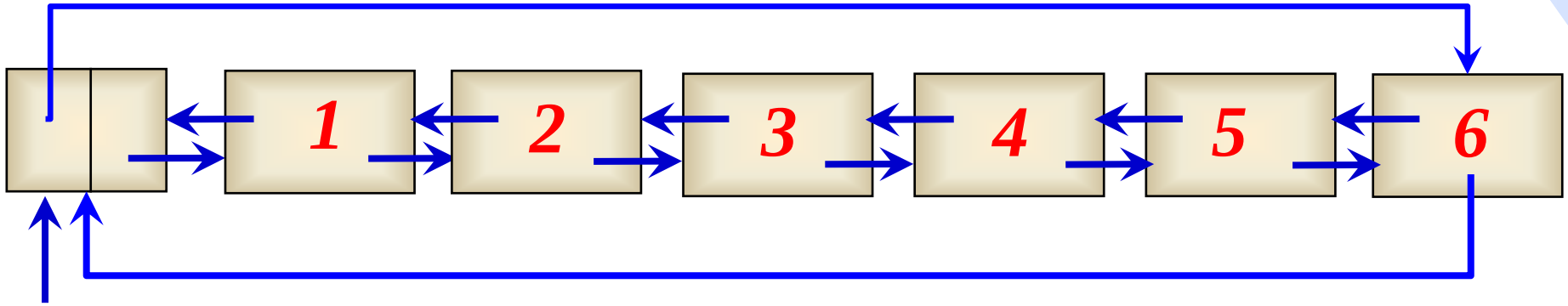
- L X Y表示把盒子X移动到盒子Y左边（如果X已在Y左边，则忽略该指令）。
- R X Y表示把盒子X移动到盒子Y右边（如果X已在Y右边，则忽略该指令）。

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 prev[7];
int next[7];
```

	0	1	2	3	4	5	6
prev							
data		1	2	3	4	5	6
next							



head

静态双向循环链表

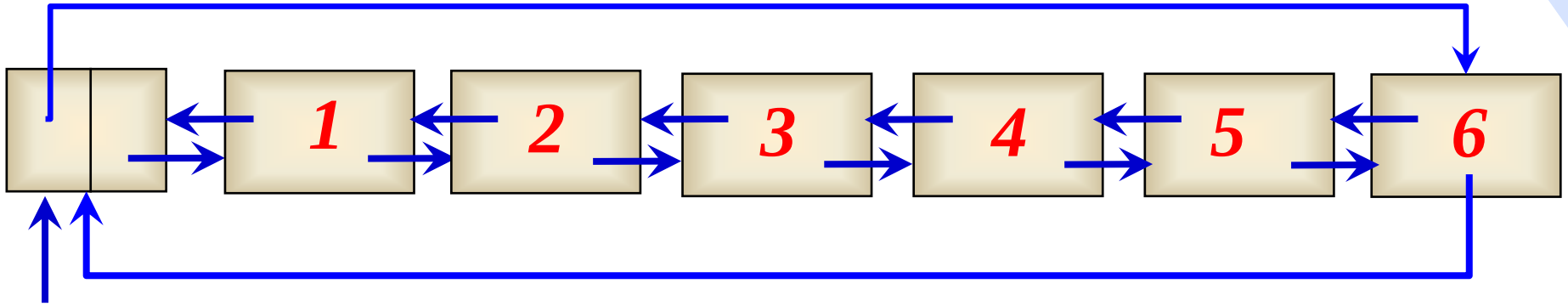
```
int data[7];
int prev[7];
int next[7];
```

盒子不动，指针动

在盒子移动过程中仅prev和next“指针”变化。

	0	1	2	3	4	5	6
prev	6	0	1	2	3	4	5
data		**1**	**2**	**3**	**4**	**5**	**6**
next	1	2	3	4	5	6	0

移动过程中盒子在数组中的位置始终不变，第*i*个盒子始终在数组第*i*位。所以可以在O(1)时间内找到任意盒子。



head

删除盒子x：

```
next[prev[x]]=next[x];
prev[next[x]]=prev[x];
```

盒子x插到y右边：

```
prev[next[y]]=x;
next[x]=next[y];
prev[x]=y;
next[y]=x;
```

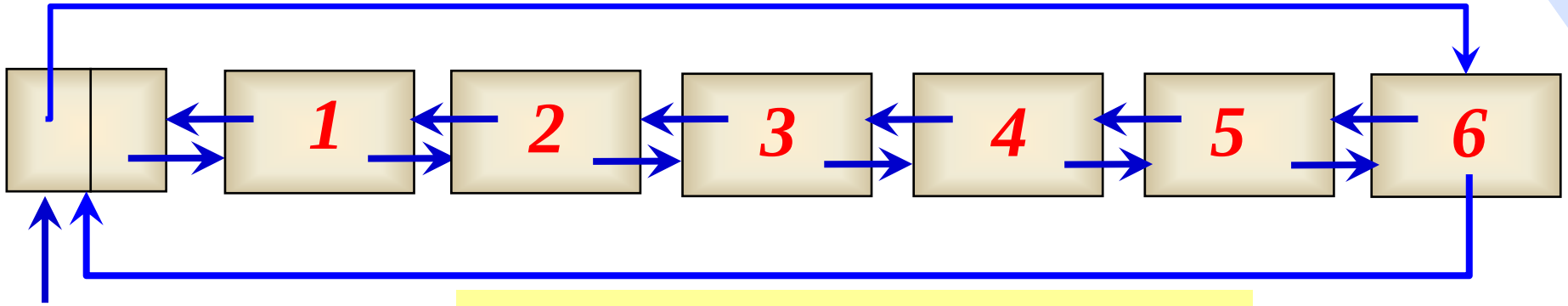
盒子x插到y左边：

```
next[prev[y]]=x;
prev[x]=prev[y];
next[x]=y;
prev[y]=x;
```

L X Y 操作：
①删除盒子X
②将X插到Y左边

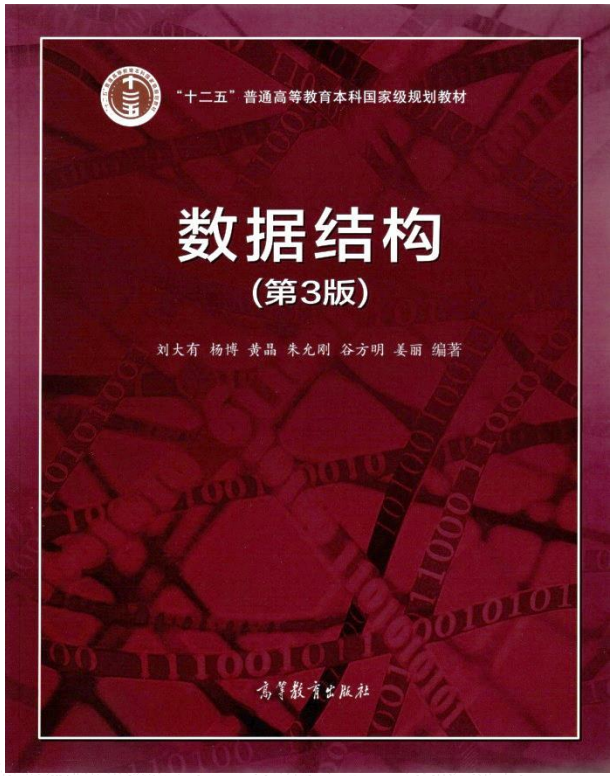
	0	1	2	3	4	5	6
prev	6	0	1	2	3	4	5
data		1	2	3	4	5	6
next	1	2	3	4	5	6	0

R X Y操作：
①删除盒子X
②将X插到Y右边



盒子不动，动的是“指针”

课堂实录



线性表的链接存储

- 单链表
- 循环链表
- 双向链表
- 静态链表
- 侵入式链表
- 链表的双指针技巧

数据之法 结构之美 算法之道

zhuyungang@jlu.edu.cn

Last updated on 2023.9

普通链表（非侵入式）

```
struct Student{
    char name[20];
    double GPA;
    Student* next;
};
```

插入 InsertStudent(…)

删除 DeleteStudent(…)

```
struct Teacher{
    char name[20];
    char title[50];
    Teacher* next;
};
```

插入 InsertTeacher(…)

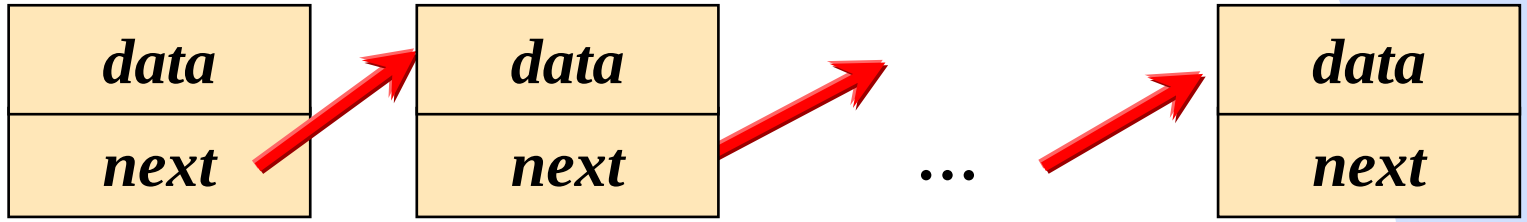
删除 DeleteTeacher(…)

```
struct Car{
    int ID;
    char ower[20];
    Car* next;
};
```

插入 InsertCar(…)

删除 DeleteCar(…)

数据与结构深度耦合，一种类型的链表只能应对一种业务场景，当业务场景改变之后，需要重新定义链表结构，重新实现链表的各种操作（如插入、删除、查找等）。



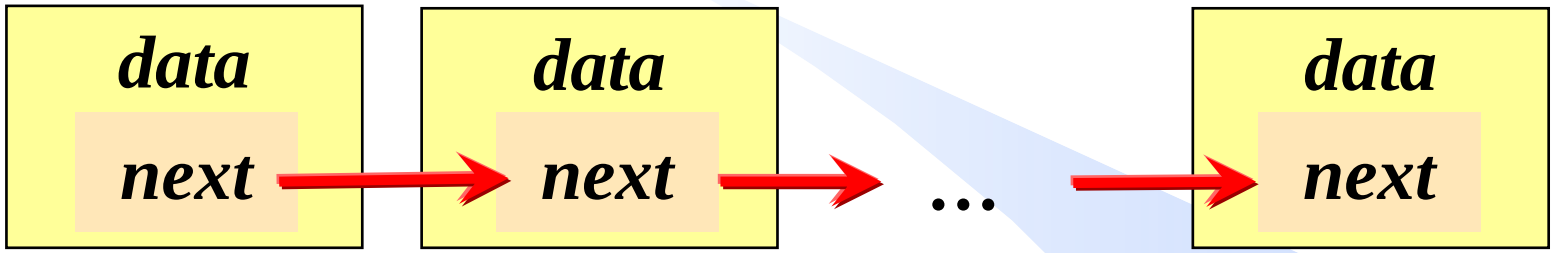
侵入式链表

一套基本的链表结构
不含数据只含指针

```
struct ListPtr{
    ListPtr* next;
};
```

插入 InsertList(…)

删除 DeleteList(…)




```
struct Student{
    char name[20];
    double GPA;
    ListPtr list;
};
```

```
struct Teacher{
    char name[20];
    char title[50];
    ListPtr list;
};
```

```
struct Car{
    int number;
    int ower[20];
    ListPtr list;
};
```

均可使用InsertList/DeleteList，如：

```
ListPtr head;
for(int i=0; i<10; i++){
    Student* s1 = new Student;
    InsertList(&head, &s1->list);
}
Teacher* t1 = new Teacher;
InsertList(&head, &t1->list);
```

将数据与结构解耦合。只需要定义一种链表结构*ListPtr*，完成一套链表操作，即可实现对所有类型链表的统一管理

侵入式链表的应用——操作系统内核链表

Microsoft® Windows®

```
typedef struct _LIST_ENTRY {
   struct _LIST_ENTRY *Flink; // 前向指针
   struct _LIST_ENTRY *Blink; // 后向指针
} LIST_ENTRY;
```

Linux™

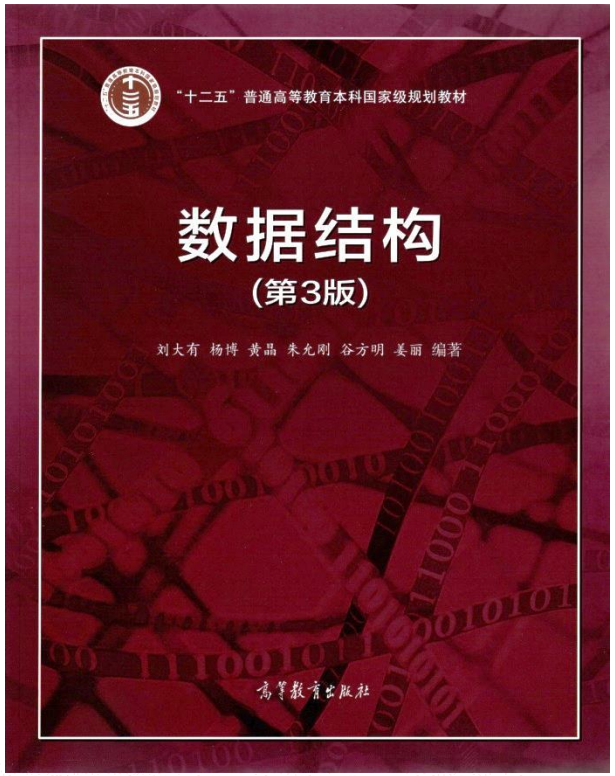
```
struct list_head {
        struct list_head *next, *prev;
};
#define list_entry(ptr, type, member) \
                container_of(ptr, type, membe
#define container_of(ptr, type, member) ({                              \
        const typeof(((type *)0)->member ) *__mptr = (ptr);         \
        (type *)( (char *)__mptr - offsetof(type,member) );})

#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)

#define LIST_HEAD_INIT(name) { &(name), &(name) }
#define LIST_HEAD(name) \
        struct list_head name = LIST_HEAD_INIT(name)
```


```
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; //前驱结点
    struct LOS_DL_LIST *pstNext; //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(LOS_DL_LIST *list, LOS
{
    node->pstNext = list->pstNext;
    node->pstPrev = list;
    list->pstNext->pstPrev = node;
    list->pstNext = node;
}
```

课堂实录



线性表的链接存储

- 单链表
- 循环链表
- 双向链表
- 静态链表
- 侵入式链表
- **链表的双指针技巧**

数据之法 结构之美 算法之道

zhuyungang@jlu.edu.cn

Last updated on 2023.9

找单链表倒数第k个结点

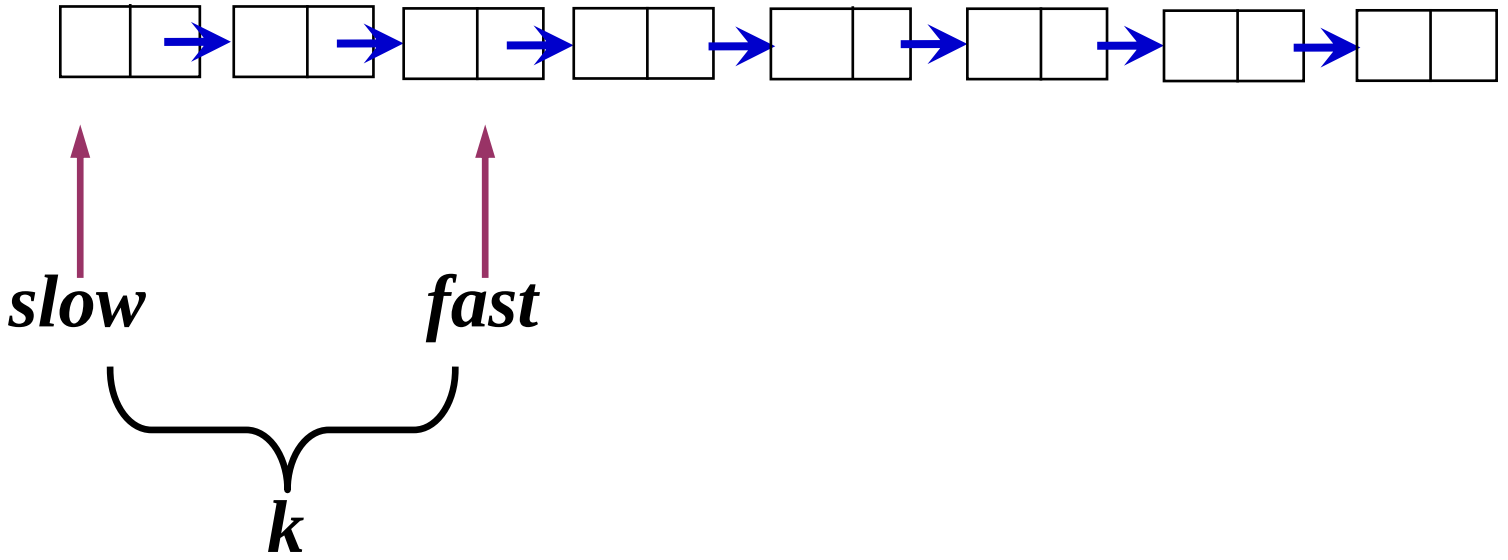
给定一个不含哨位结点的单链表，请设计一个尽可能高效的算法，查找链表中倒数第k个结点（k为正整数）。若查找成功，算法返回该结点的数据域之值；否则，返回0。（考研题全国卷，大厂面试题LeetCode）

```
struct ListNode{
    int val;
    ListNode* next;
};
```

- 解法1：
  - ✓ 先找到最后一个结点，再往前找k-1次前驱
  - ✓ 找某个结点的前驱结点：时间$O(n)$
  - ✓ 整个算法时间复杂度$O(n^2)$

- 解法2：
  - ✓事实：倒数第k个，即正数第n-k+1个，n为链表长度
  - ✓先遍历一遍链表，确定链表长度n，再遍历一遍链表找第n-k+1个结点
  - ✓遍历2次链表，时间复杂度$O(n)$

- 解法3：
  - ✓使用两个指针:*fast*和*slow*，先把*fast*指向第*k*个结点，*slow*指向第1个结点。然后*fast*和*slow*同时并行向后移动，当*fast*移动到最后一个结点时，*slow*正好指向倒数第*k*个结点。
  - ✓只遍历1次链表，时间复杂度O(*n*)。

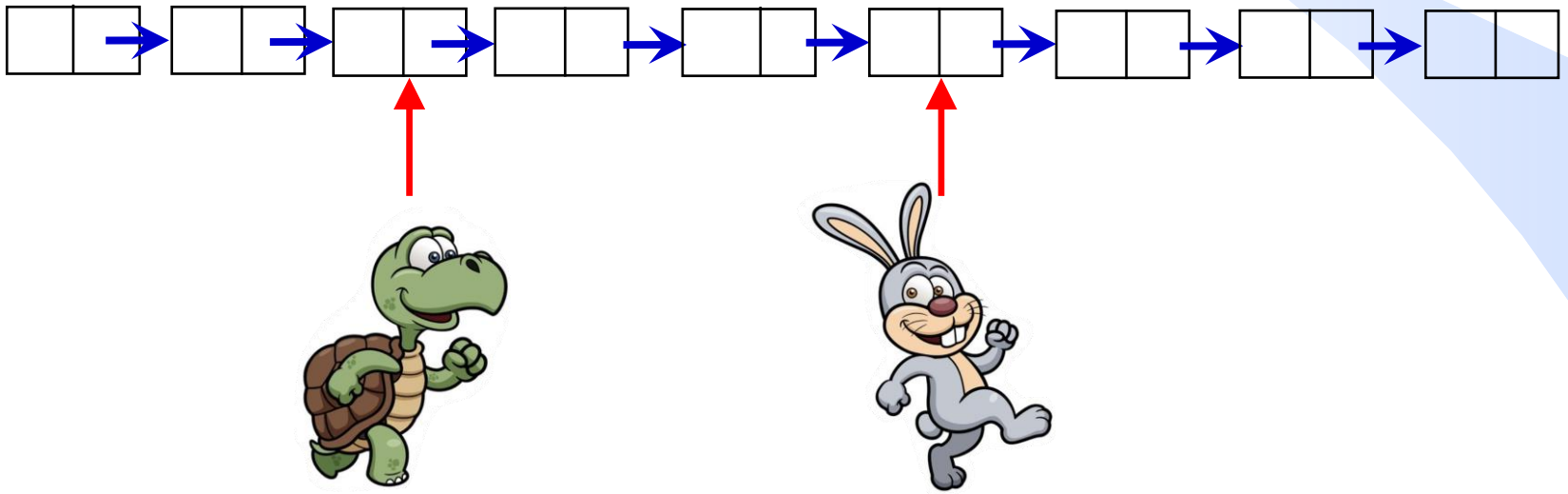


Talk is cheap, show me the code

```
int kthToLast(ListNode* head, int k) {
    ListNode *slow = head, *fast = head;
    for(int i = 1; fast != NULL && i < k; i++)
        fast = fast->next;      //让fast指向正数第k个结点
    if(fast==NULL) return 0;    //链表长度不足k
    while(fast->next != NULL) {
        slow = slow->next;
        fast = fast->next;
    }
    return slow->val;
}
```

找单链表中间结点

找单链表中间位置的结点，要求只遍历一次链表。若链表长度为偶数，返回两个中间结点中靠右的那个结点。【大厂面试题 LeetCode 876】

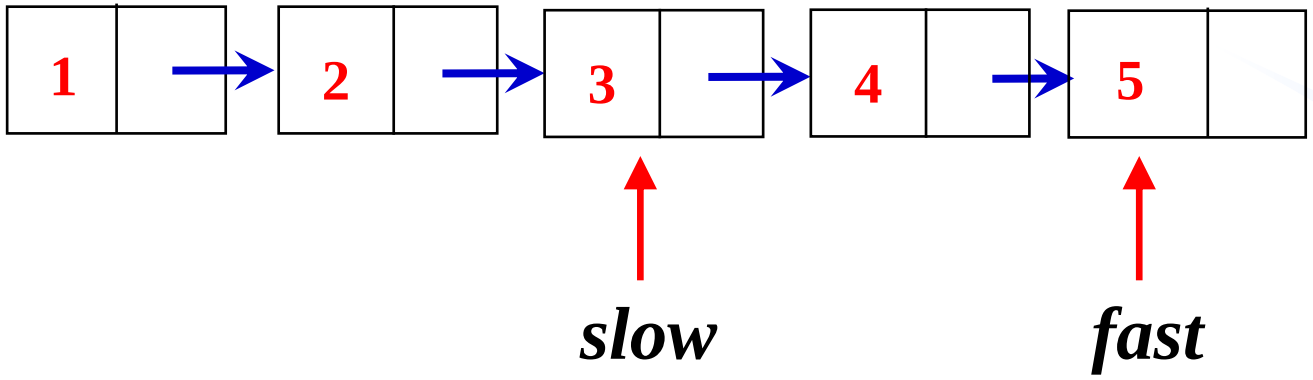


✓维护两个指针fast和slow

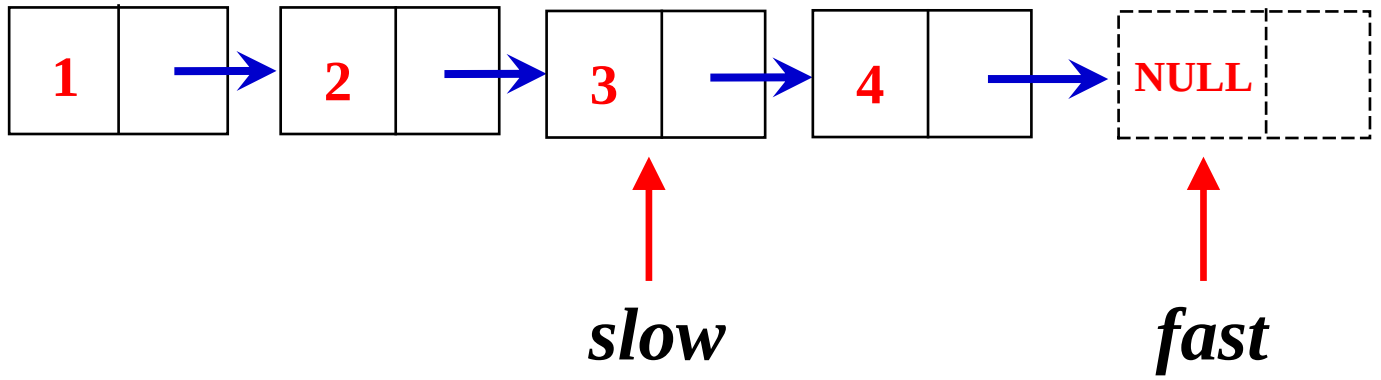
✓slow每次移动1步，fast每次移动2步

```
slow = slow->next;
fast = fast->next->next;
```

扫描结束的条件




```
fast->next==NULL
```

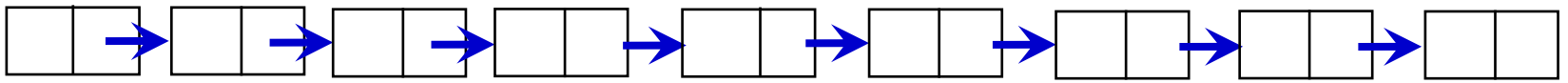



```
fast==NULL
```

Talk is cheap, show me the code

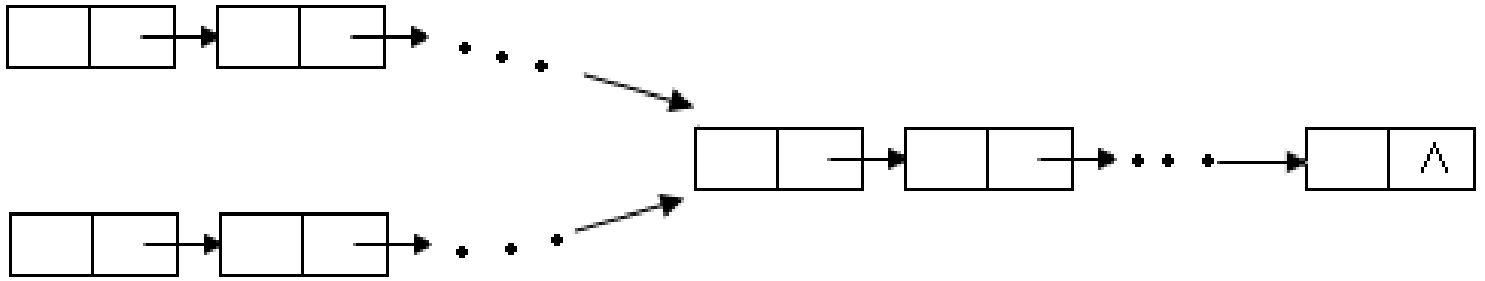
```
ListNode* middleNode(ListNode* head) {
    ListNode *fast = head, *slow = head;
    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
    }
    return slow;
}
```

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？

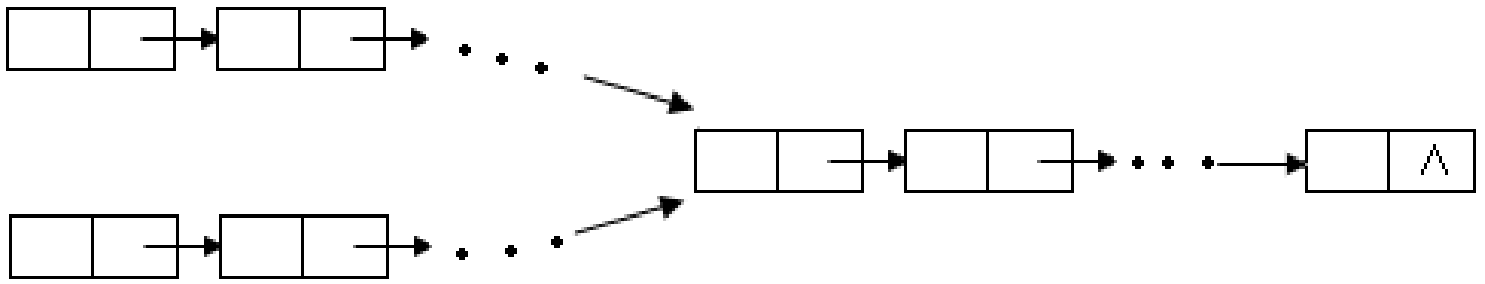


链表相交问题

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$O(L_1+L_2)$，L_1、L_2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【大厂面试题 LeetCode 160】



链表相交问题



方案1:

FOR ∀结点*i* ∈ 链表1 DO

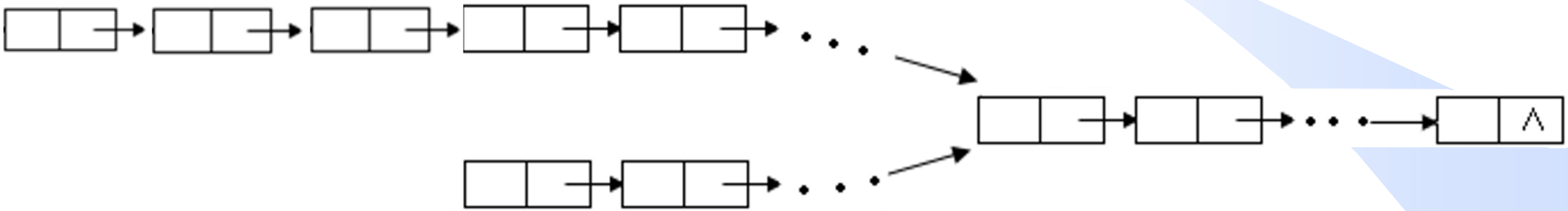
(

判断“结点*i*”是否在链表2中.

)

时间复杂度：O(L_1*L_2)

链表相交问题



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。

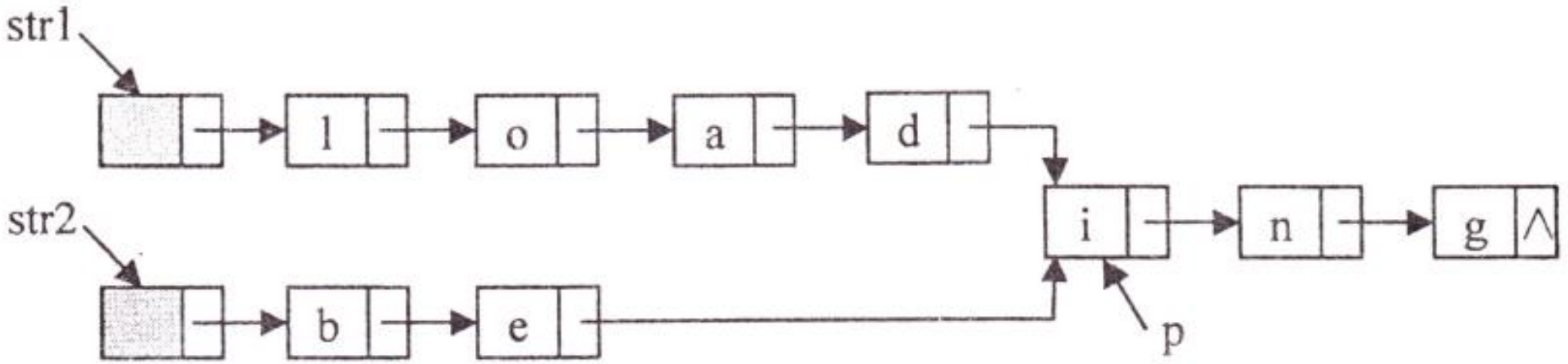
- **找交点？**用指针p1指向较长的那个链表，p2指向较短的那个链表，p1先向后移动$|L_1-L_2|$步，使p1和p2“对齐”，然后p1和p2同时并行向右移动，每移动一步比较p1和p2是否相等，当二者相等时，其指向的结点即为交点。

```
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;
}
```

Talk is cheap,
show me the code

考研题全国卷

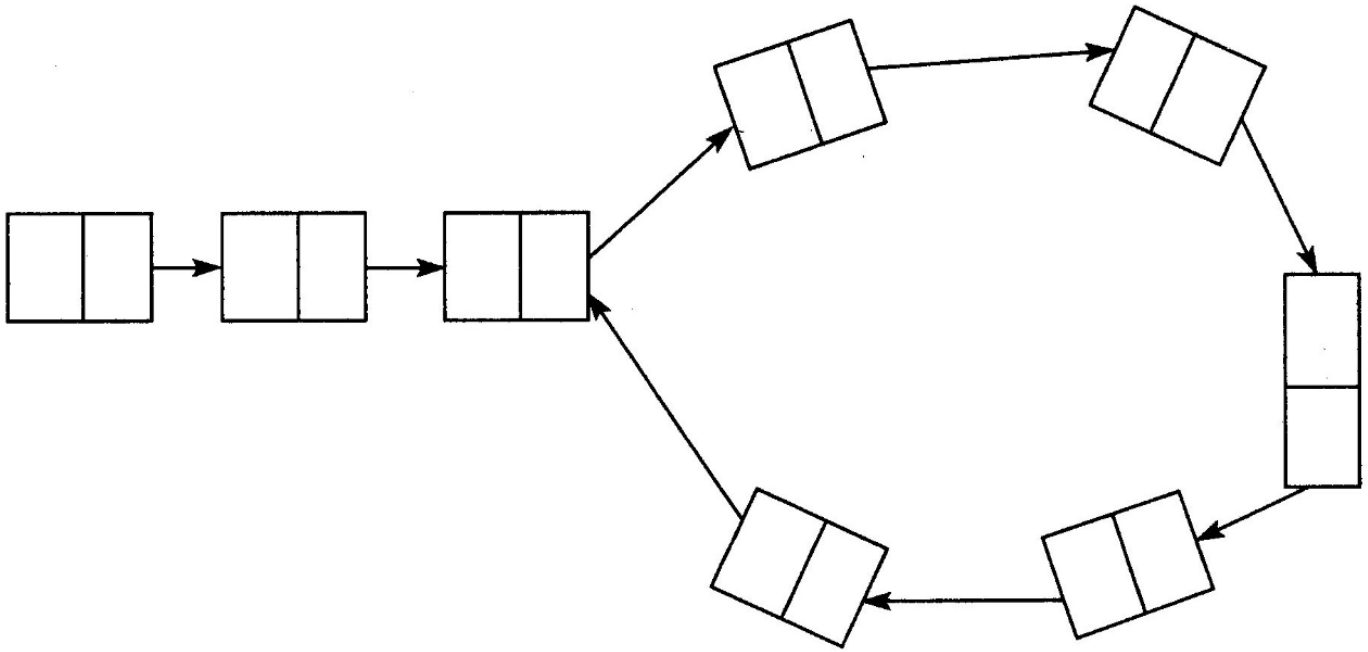
（13 分）假定采用带头结点的单链表保存单词，当两个单词有相同的后缀时，则可共享相同的后缀存储空间。例如，“loading”和“being”的存储映像如下图所示。



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

编写算法判断一个单链表中是否含有环。如果有环的话，找出从头结点进入环的第一个结点。【大厂面试题LeetCode 141、LeetCode 142】



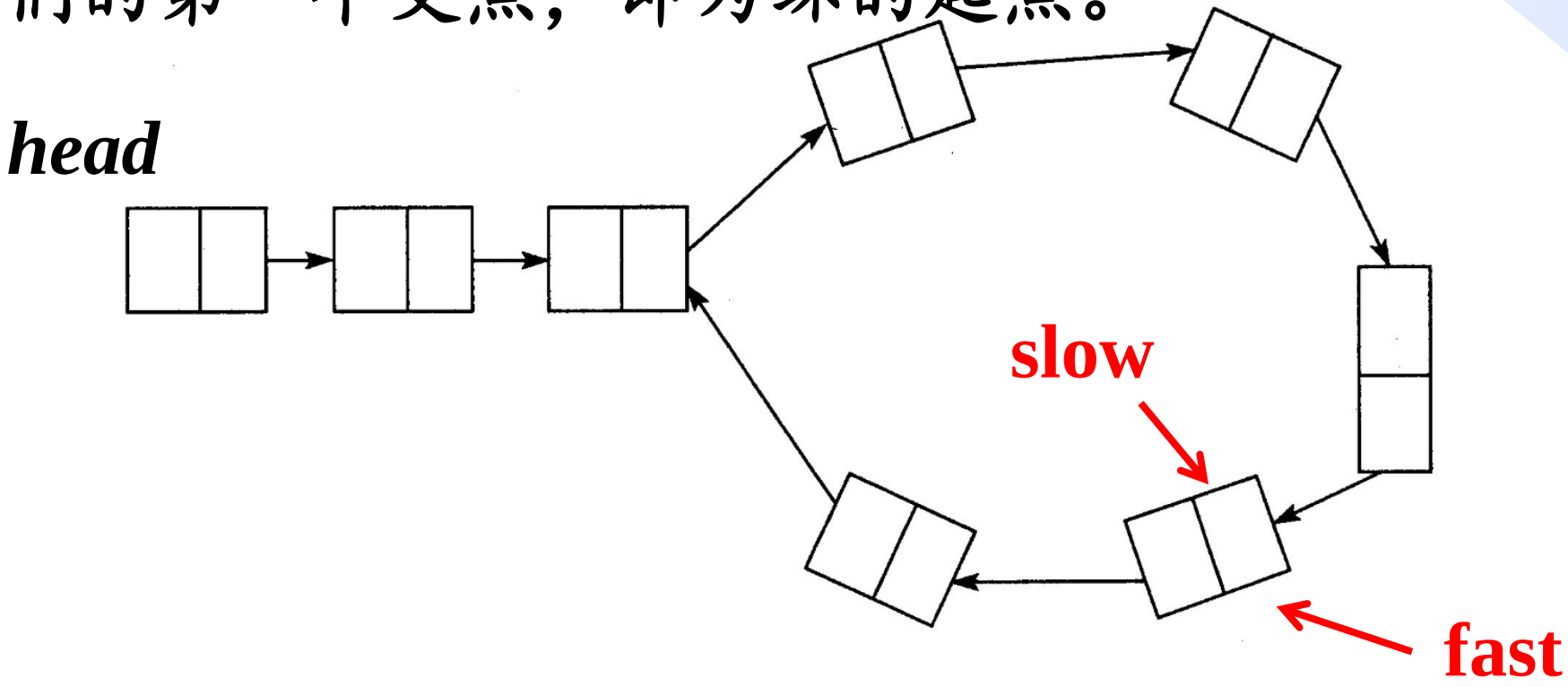
单链表判环问题

- 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;
}
```

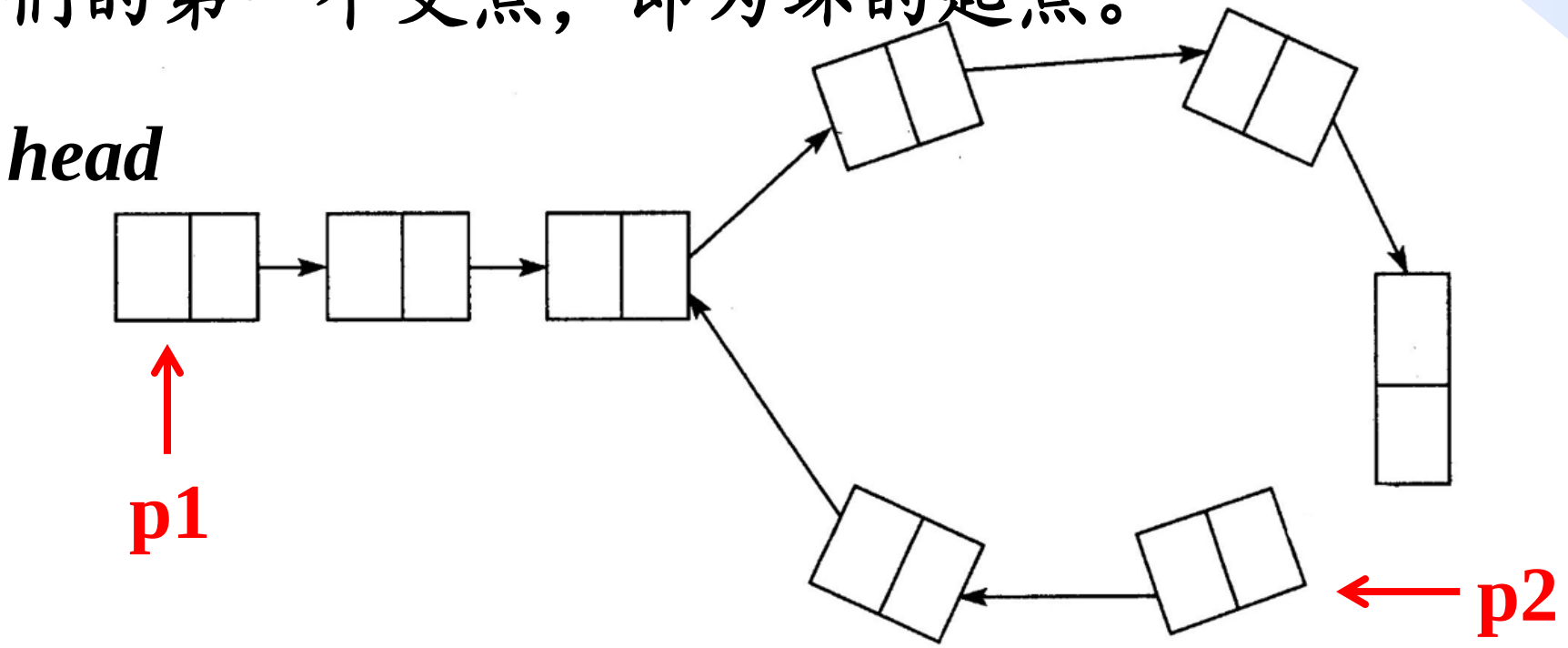
单链表判环问题

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。



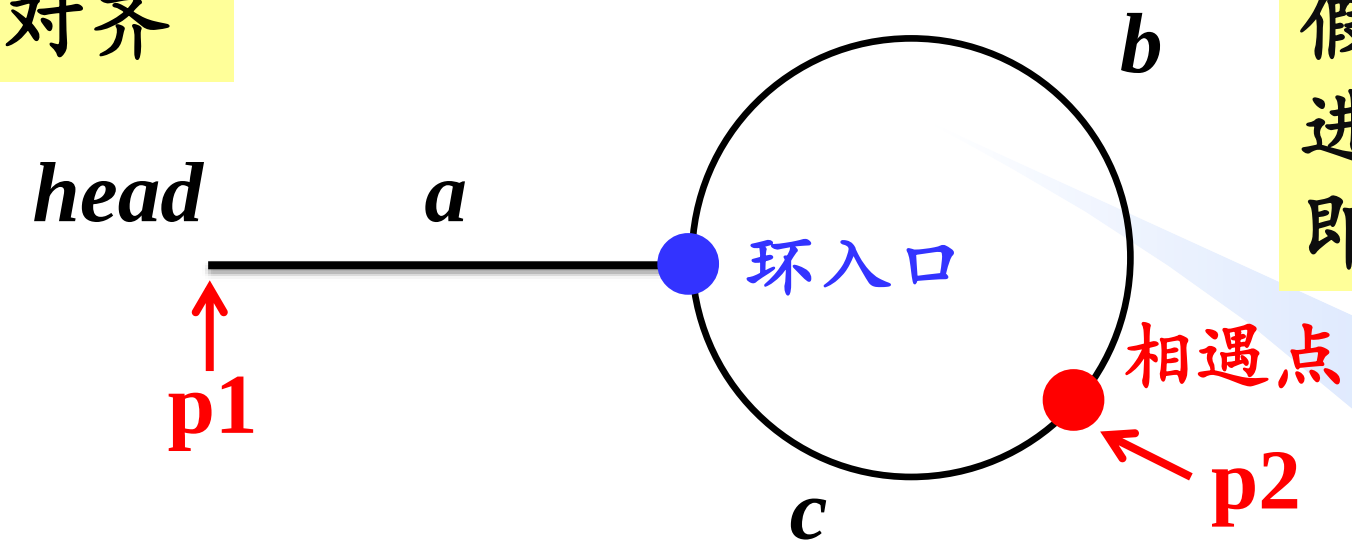
单链表判环问题

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？

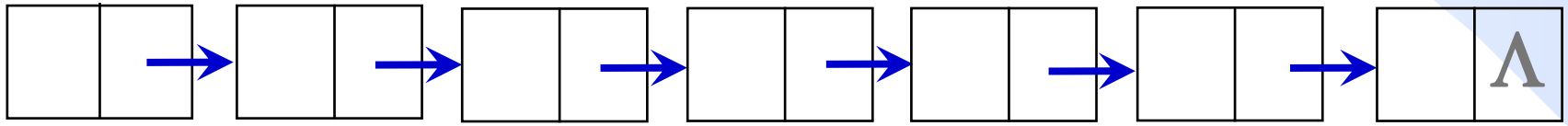
单链表判环问题

```
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)

【大厂面试题LeetCode 234】



回文：中文对称

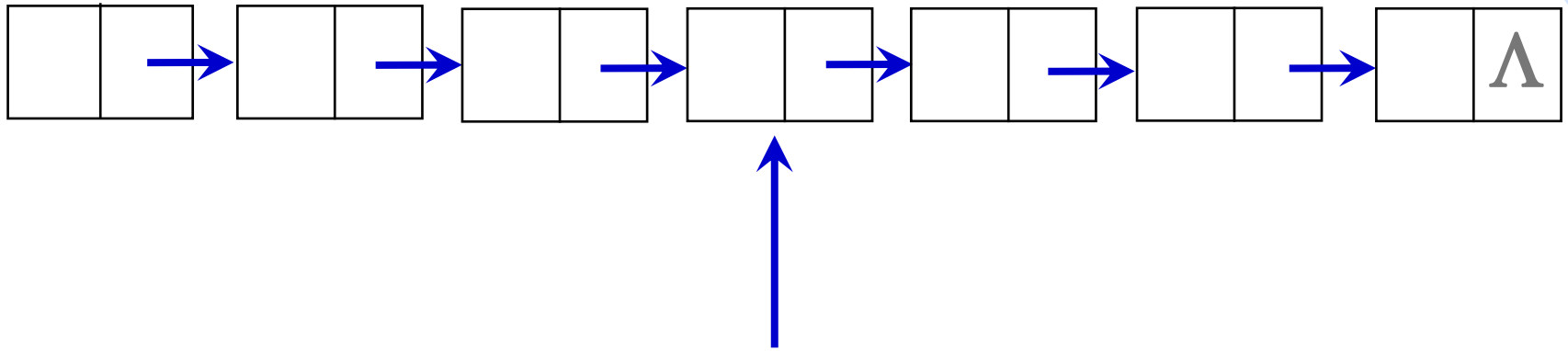
level

refer

上海自来水来自海上

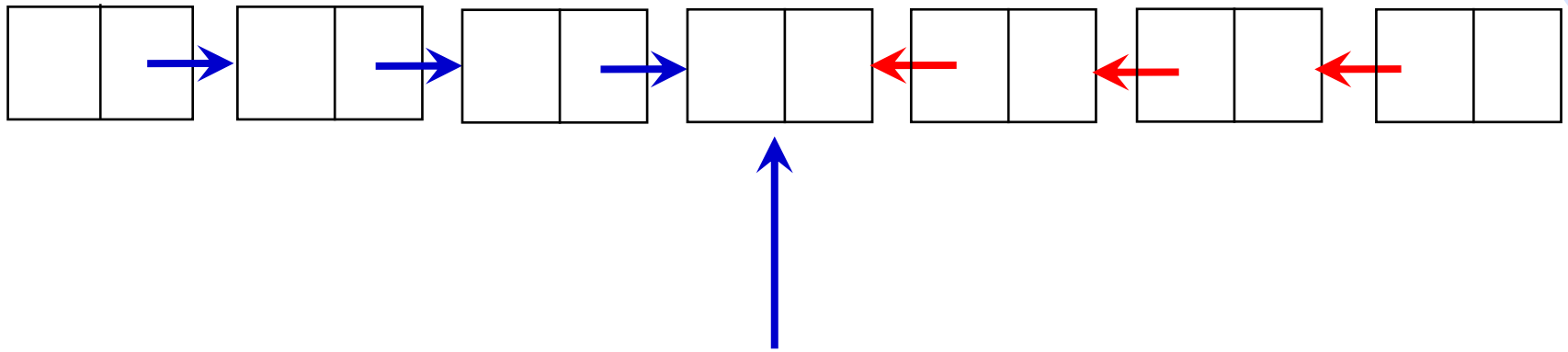
判断一个单链表是否是回文

找到中间结点，将后半部分链表反转



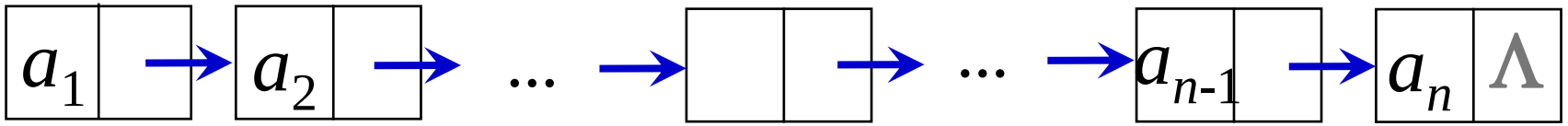
判断一个单链表是否是回文

找到中间结点，将后半部分链表反转



重排链表结点

设线性表$L=(a_1, a_2, a_3, \ldots, a_{n-2}, a_{n-1}, a_n)$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$L'=(a_1, a_n, a_2, a_{n-1}, a_3, a_{n-2}\ldots)$。**【2019年考研题全国卷（13分），大厂面试题** LeetCode 143 **】**



重排链表结点

设线性表$L=(a_1, a_2, a_3, \ldots, a_{n-2}, a_{n-1}, a_n)$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$L'=(a_1, a_n, a_2, a_{n-1}, a_3, a_{n-2}\ldots)$。【2019年考研题全国卷（13分），大厂面试题 LeetCode 143】

一元多项式及其操作

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

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*	*next*

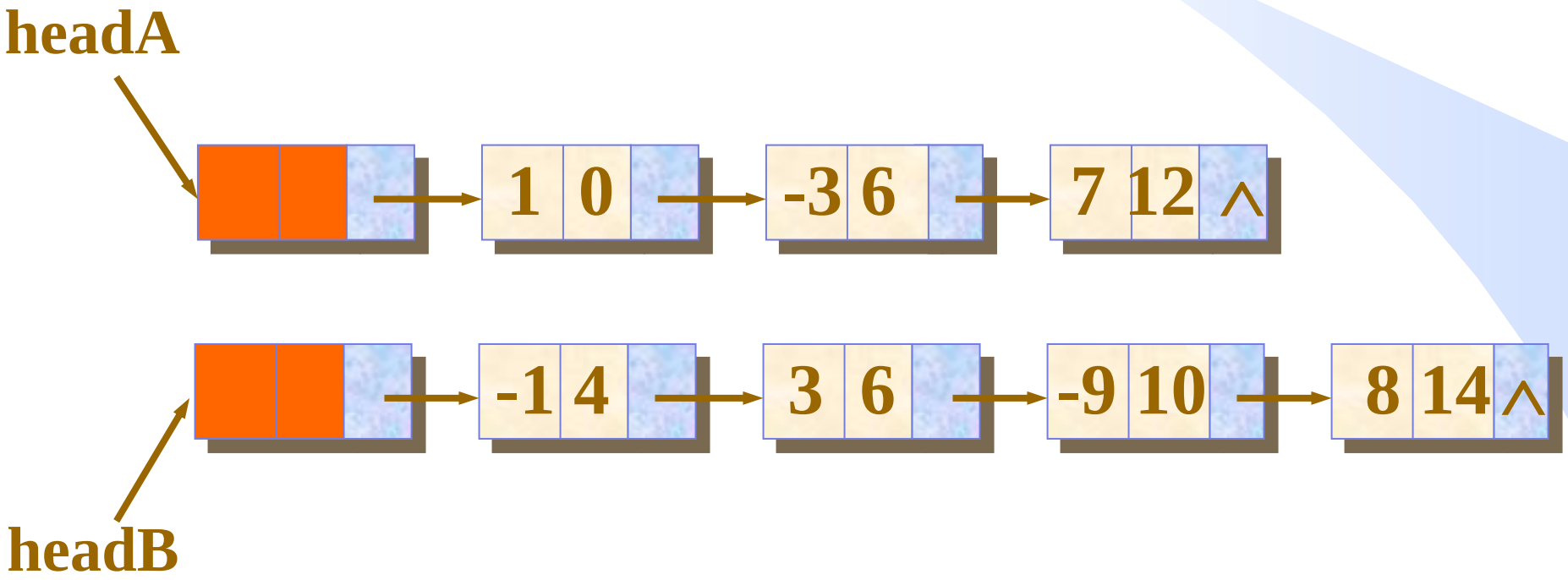
优点：多项式的项数可以动态地增长，不会出现存储溢出问题。

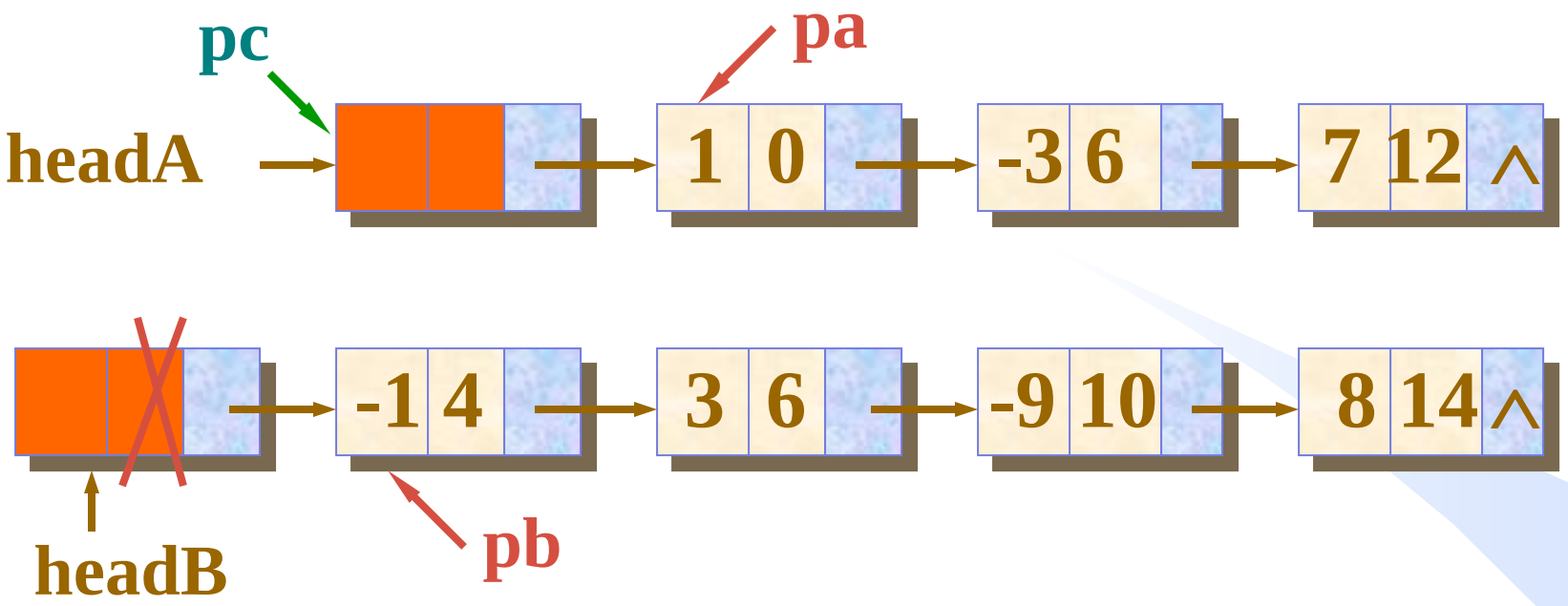
插入、删除方便，不移动元素，只需“穿针引线”。

多项式链表相加示例

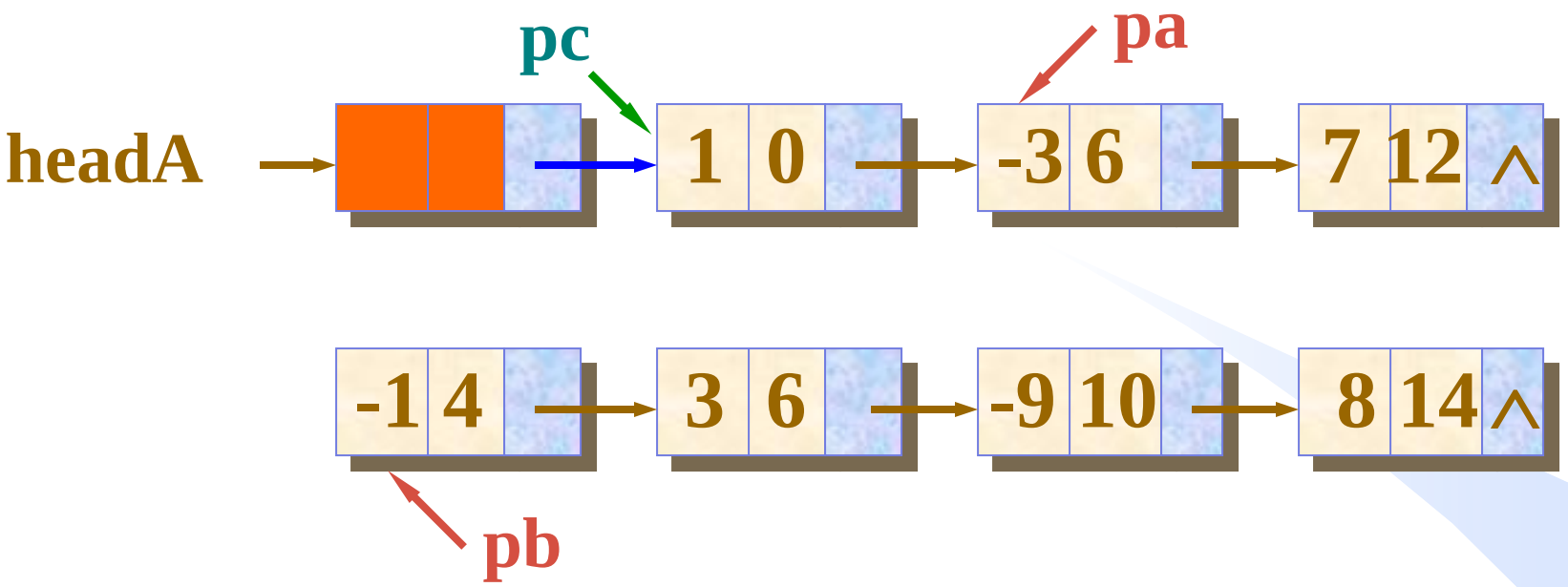
$$A = 1 - 3x^6 + 7x^{12}$$

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

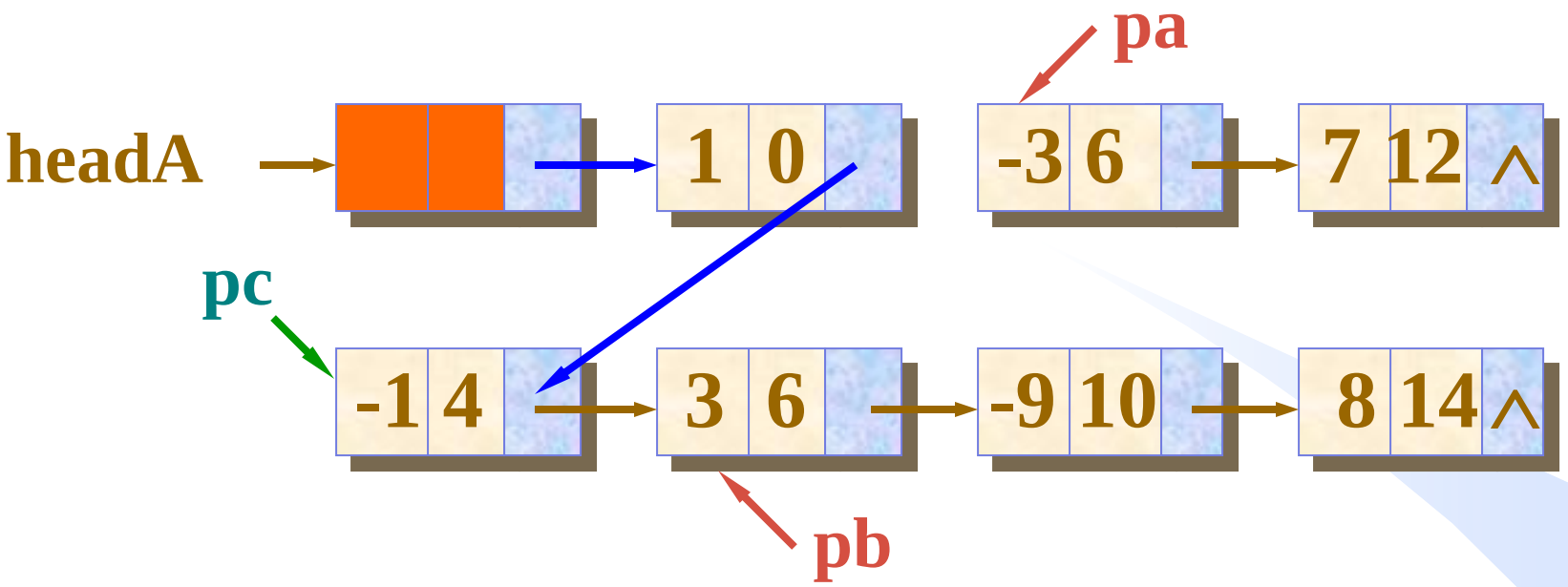




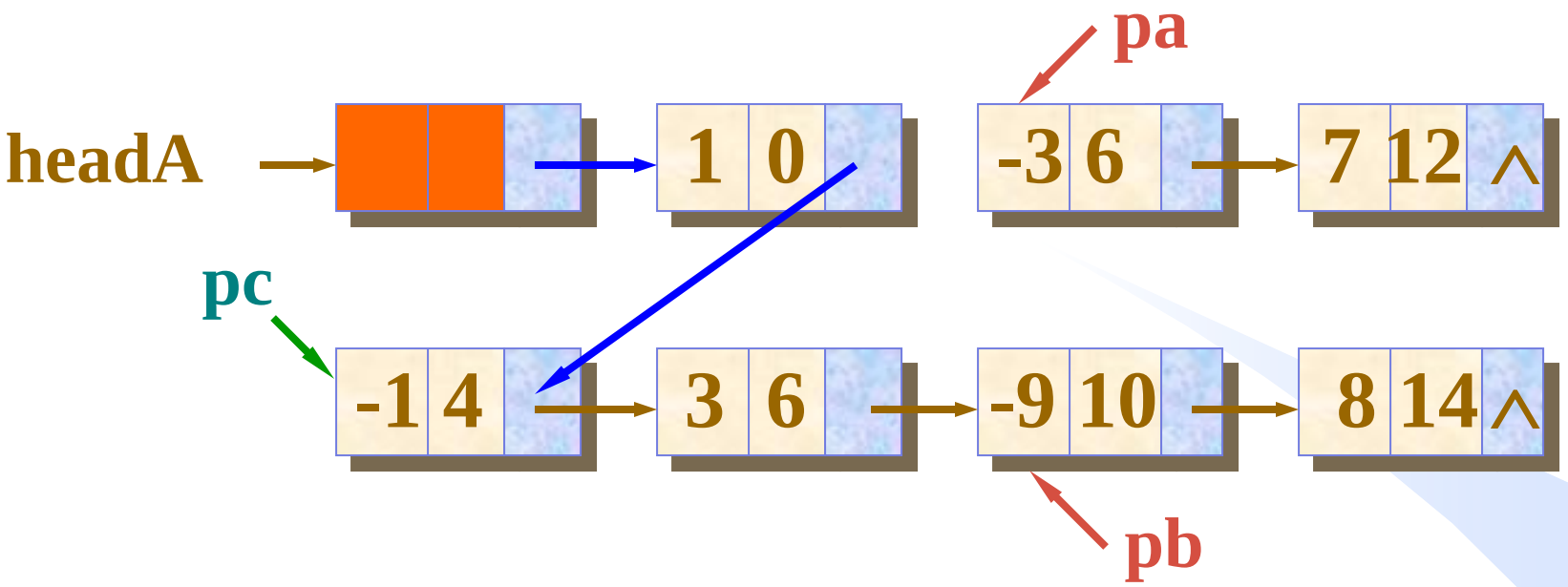
通过指针*pa*扫描链表A，通过指针*pb*扫描链表B
指针*pc*指向当前计算结果的最后一个结点



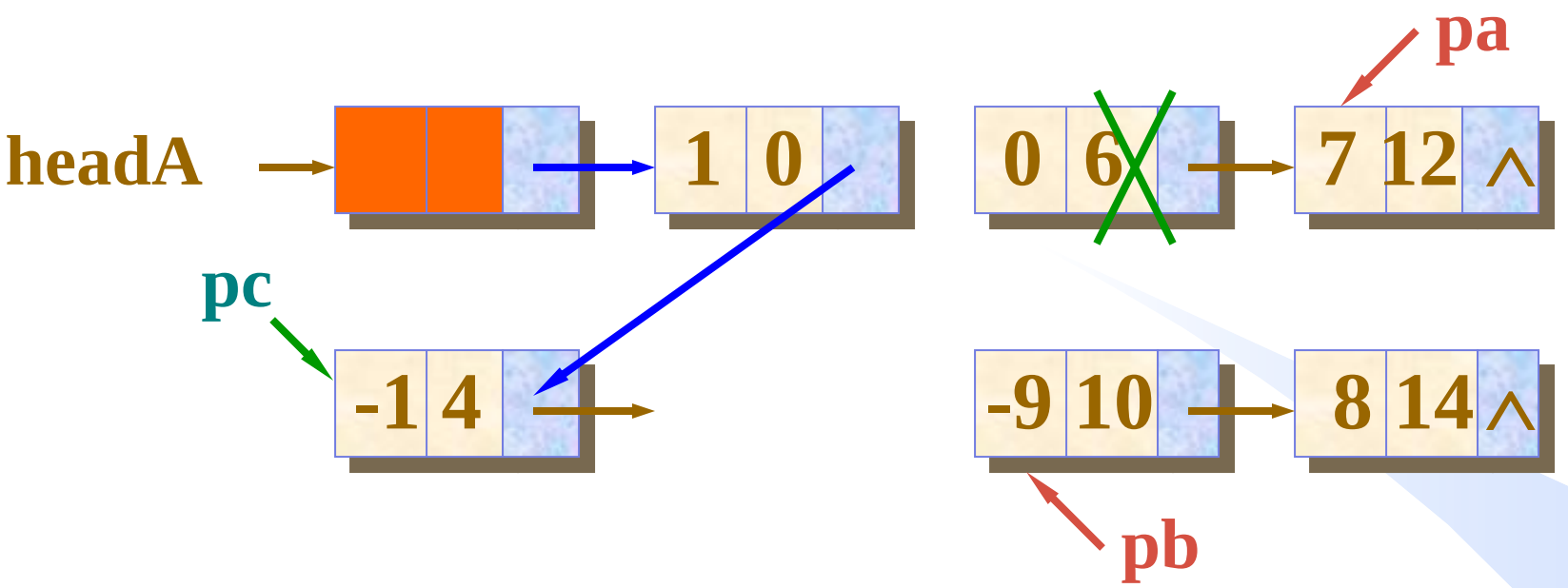
通过指针*pa*扫描链表A，通过指针*pb*扫描链表B
指针*pc*指向当前计算结果的最后一个结点



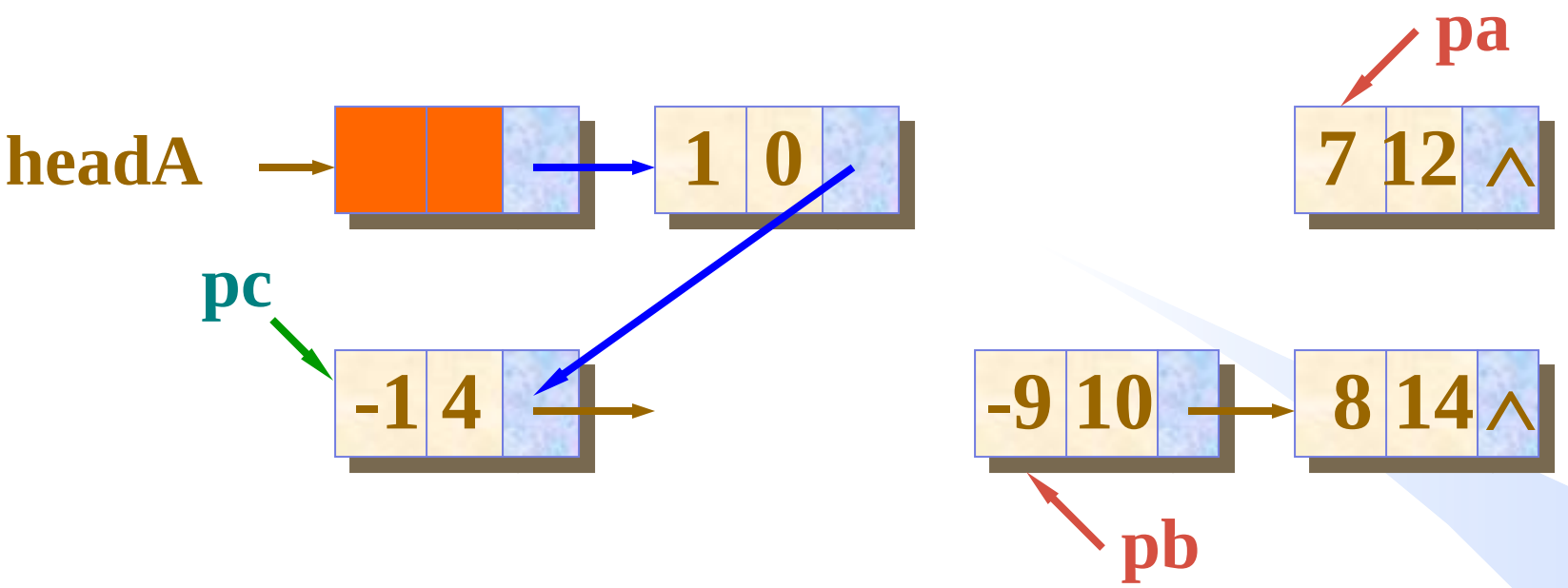
通过指针*pa*扫描链表A，通过指针*pb*扫描链表B
指针*pc*指向当前计算结果的最后一个结点



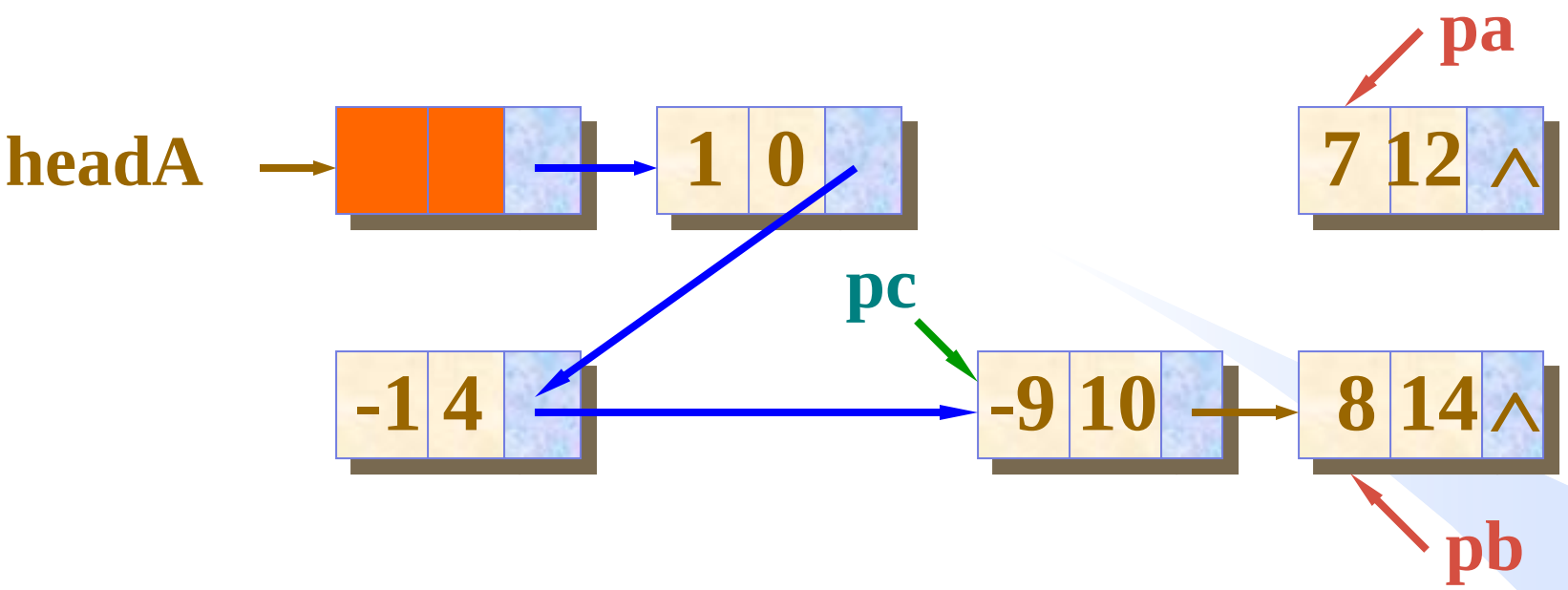
通过指针*pa*扫描链表A，通过指针*pb*扫描链表B
指针*pc*指向当前计算结果的最后一个结点



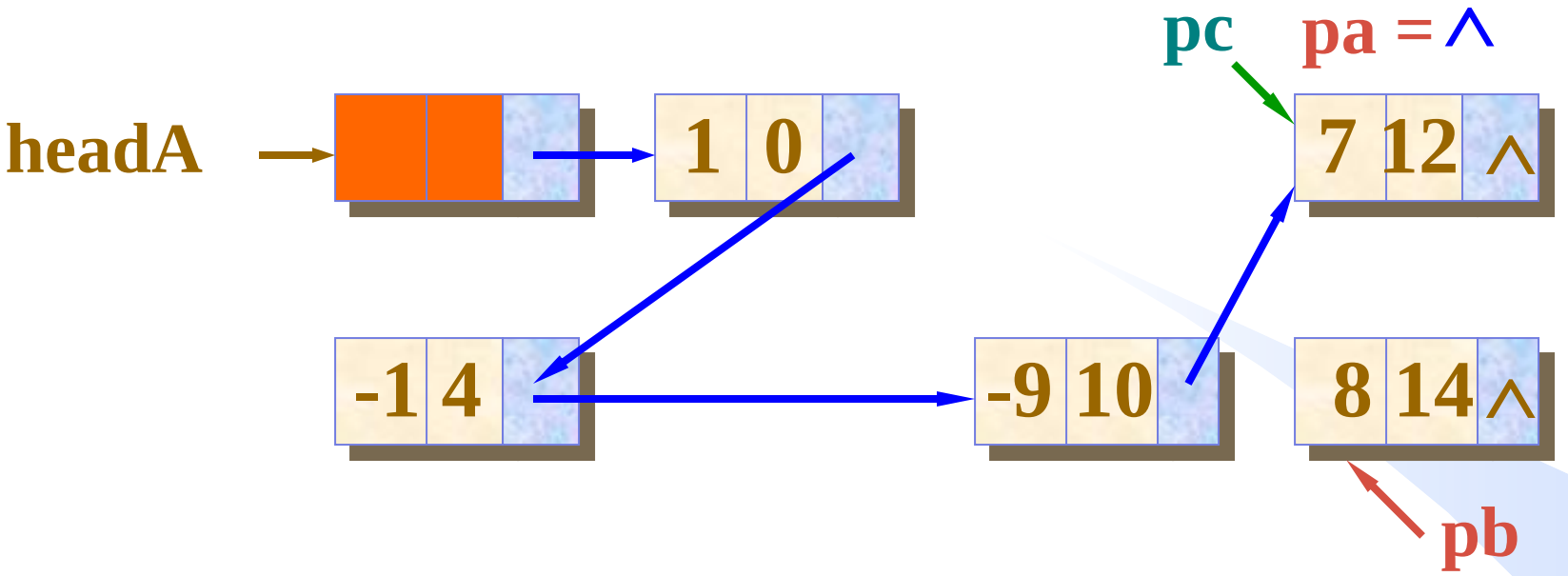
通过指针*pa*扫描链表A，通过指针*pb*扫描链表B
指针*pc*指向当前计算结果的最后一个结点



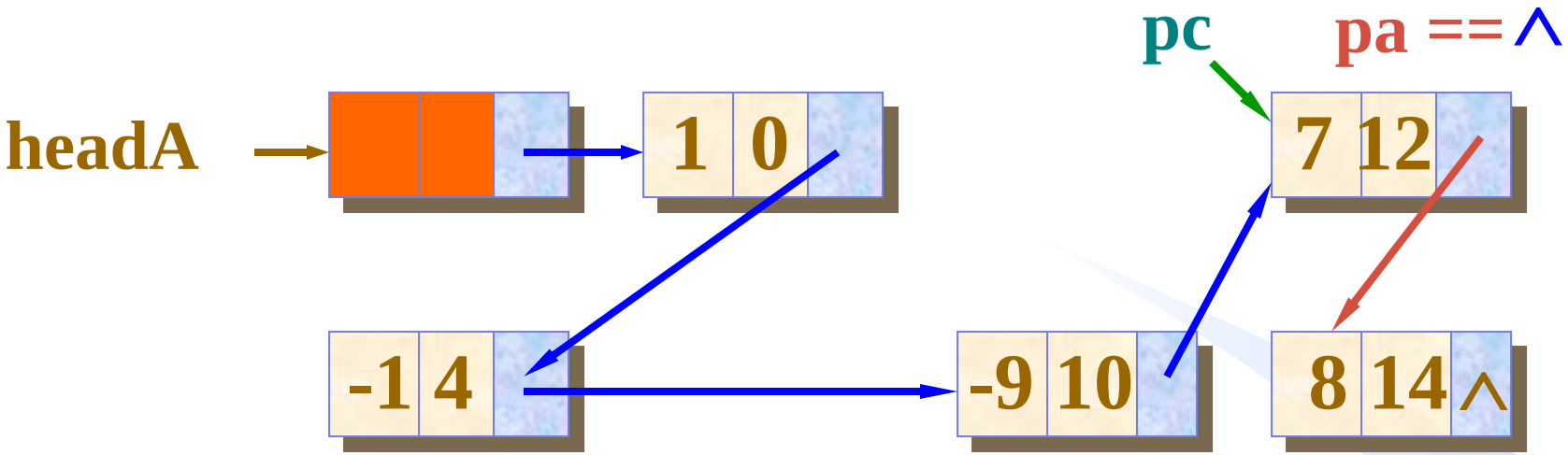
通过指针*pa*扫描链表A，通过指针*pb*扫描链表B
指针*pc*指向当前计算结果的最后一个结点

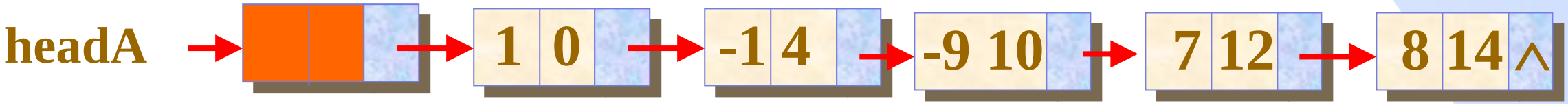


通过指针*pa*扫描链表A，通过指针*pb*扫描链表B
指针*pc*指向当前计算结果的最后一个结点



通过指针*pa*扫描链表A，通过指针*pb*扫描链表B
指针*pc*指向当前计算结果的最后一个结点





$$A = 1 - x^4 - 9x^{10} + 7x^{12} + 8x^{14}$$