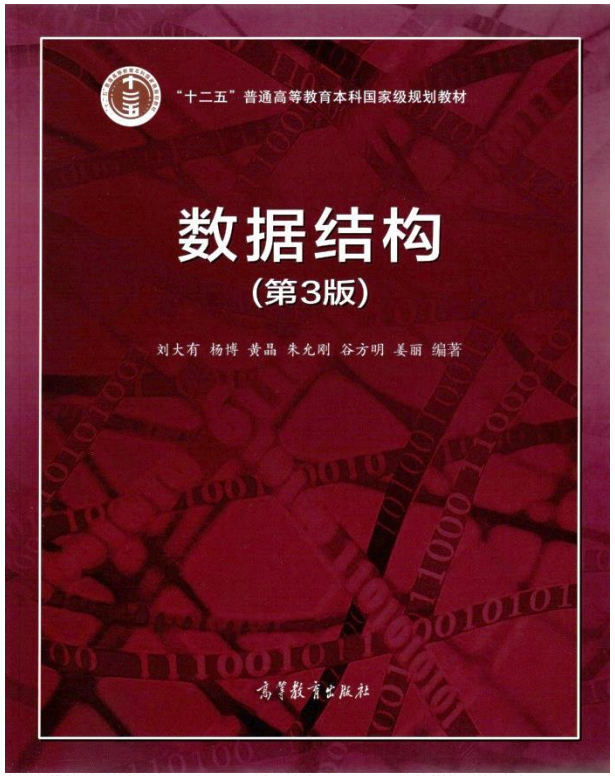


线性表的链接存储

- 单链表
- 循环链表
- 双向链表

数据之法
结构之美
算法之道

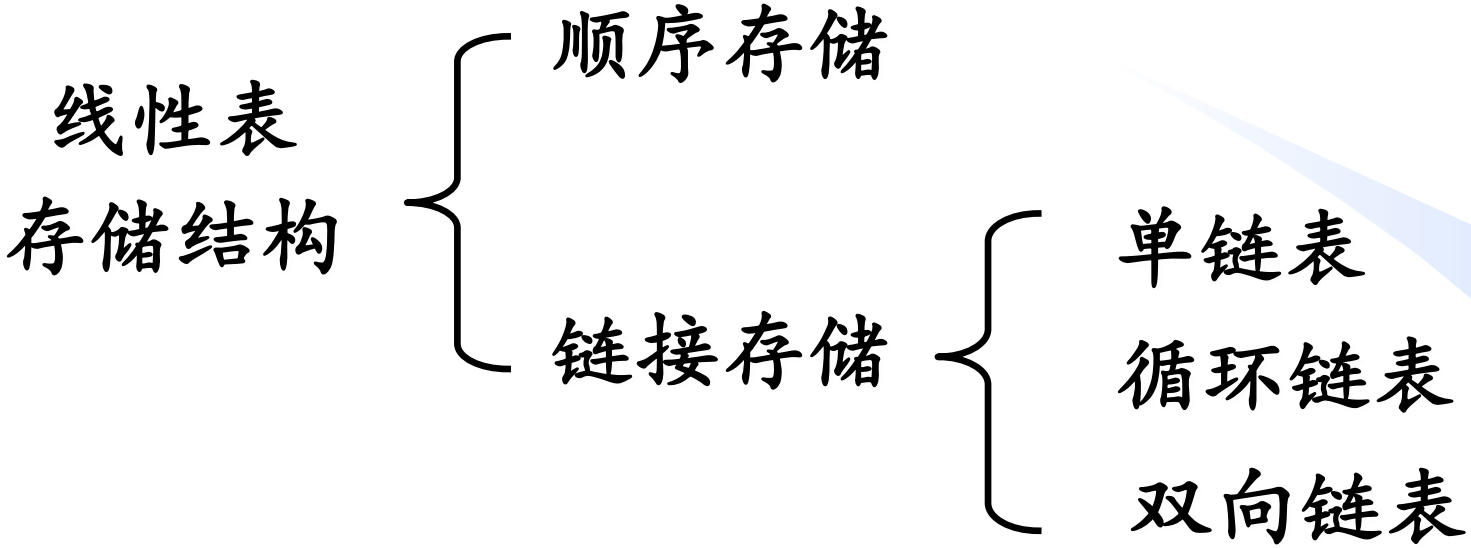
Last updated on 2024.9

zhuyungang@jlu.edu.cn

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

线性表的存储结构

A

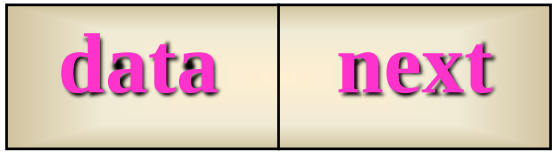


链接存储：用**任意**一组存储单元存储线性表，一个存储单元除包含结点数据字段的值，还必须存放其逻辑相邻结点（前驱或后继结点）的地址信息，即指针字段。

单链表（Singly Linked List）

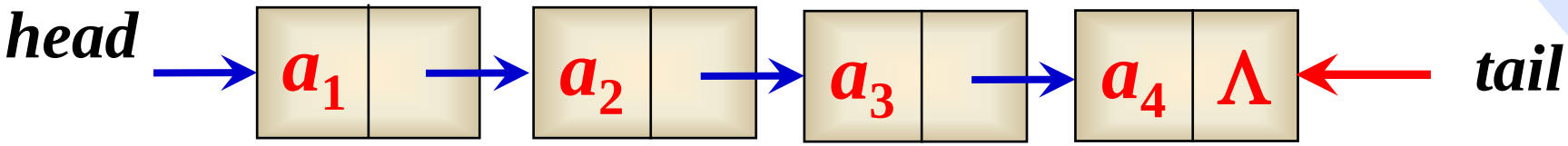
A

- 单链表的**结点结构**：




```
struct ListNode{
    int data;
    ListNode* next;
    ListNode(int d){data=d; next=NULL;}
};
```

单链表的定义：每个结点只含**一个链接域**的链表叫单链表。

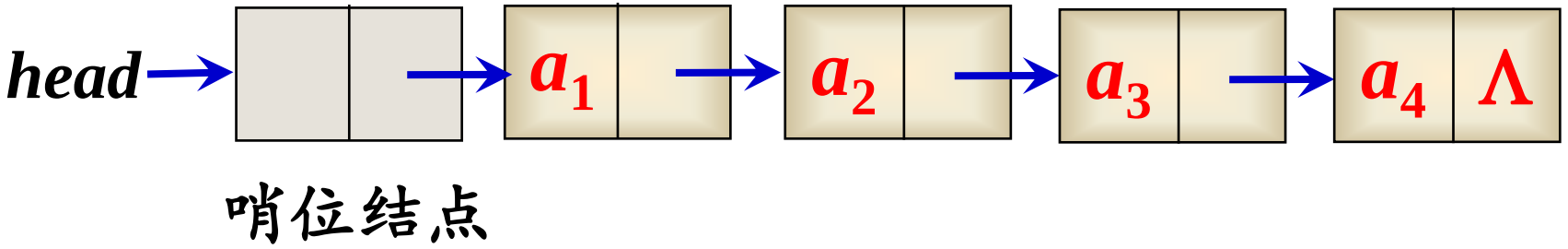


- 链表的第一个结点被称为**头结点**（也称为表头），指向头结点的指针被称为**头指针**（*head*）.
- 链表的最后一个结点被称为**尾结点**（也称为表尾），指向尾结点的指针被称为**尾指针**（*tail*）.

单链表的哨位结点

B

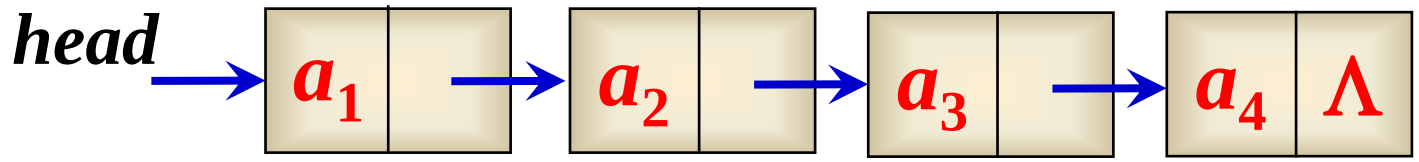
- 为便于在表头进行插入、删除操作，通常在表的前端增加一个特殊的表头结点，称其为哨位（哨兵）结点 。
- 哨位结点不被看作表中的实际结点，我们在讨论链表中第k个结点时均指第k个实际的结点。
- 表的长度：非哨位结点的个数。若表中只有哨位结点，则称其为空链表，即表长度为0



单链表的哨位结点

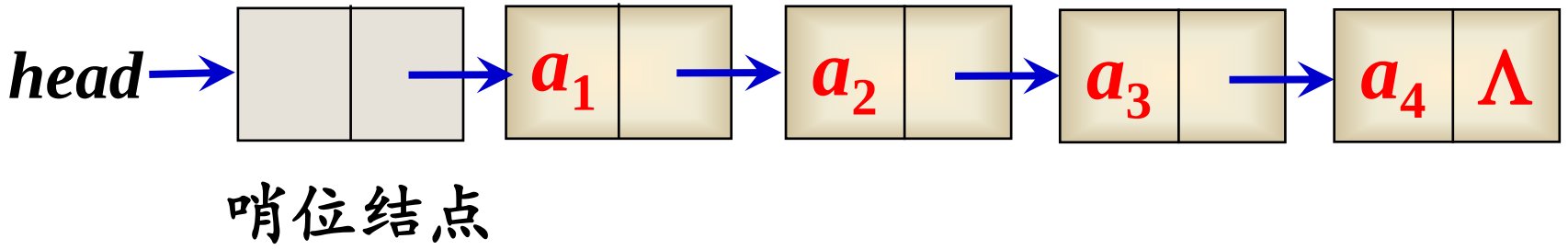
B

- 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……



删除第一个结点和删除中间结点过程不同，需分别处理

  - ✓ 有了哨位结点后……



删除第一个结点和删除中间结点过程相同

单链表主要操作举例——查找

A

```
算法 FindKth (head, k)                    //找链表第k个结点，并返回指针
  IF  k < 0 THEN RETURN Λ.                //输入的k位置不合法
   p←head . i←0.  //初始化，令指针p指向哨位结点，计数器初值为0
  //找第k个结点
   WHILE p≠Λ AND i<k DO  //若找到第k个结点或已达表尾则循环终止
   (    p←next(p) .
        i←i+1.
   )
   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;
```

查找——按序查找

A

```
ListNode* FindKth(ListNode* head, int k){
      //找链表中第k个结点，并返回指针
      if(k<0) return NULL;      //输入的k位置不合法
      ListNode *p=head;         //令指针p指向哨位结点
      int i=0;
      while(p!=NULL && i<k){ //找第k个结点
            p=p->next;
            i++;
      }
      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;

查找——按值查找

A

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

查找——按值查找

A

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

查找——找前驱

A

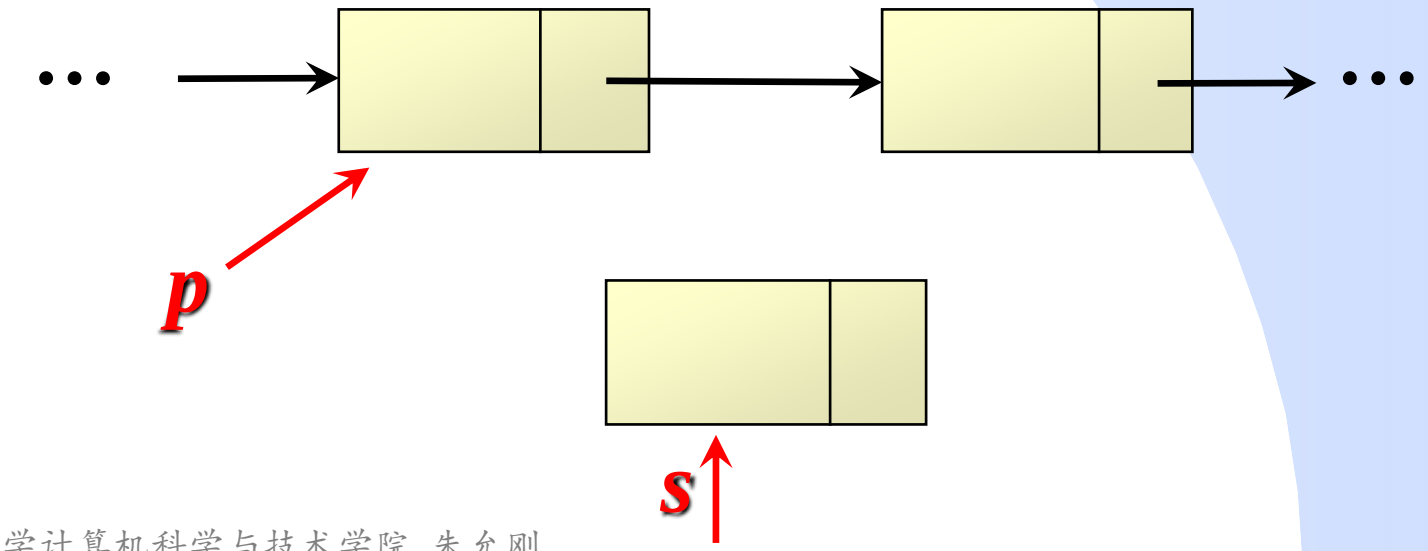
```
ListNode* FindPrev(ListNode* head, ListNode* p) {
      //找p指向的结点的前驱结点，并返回指针
      if(p==NULL) return NULL;
      for(ListNode* q=head; q!=NULL; q=q->next)
            if(q->next == p) return q;
      return NULL;
}
```

插入——在p指向的结点右侧插入结点

A

```
void Insert(ListNode* head, ListNode* p, int K){
    //指针p指向的结点右侧插入数据域为K的新结点
    if(p==NULL) return;
    ListNode* s=new ListNode(K);      //生成新结点s
```

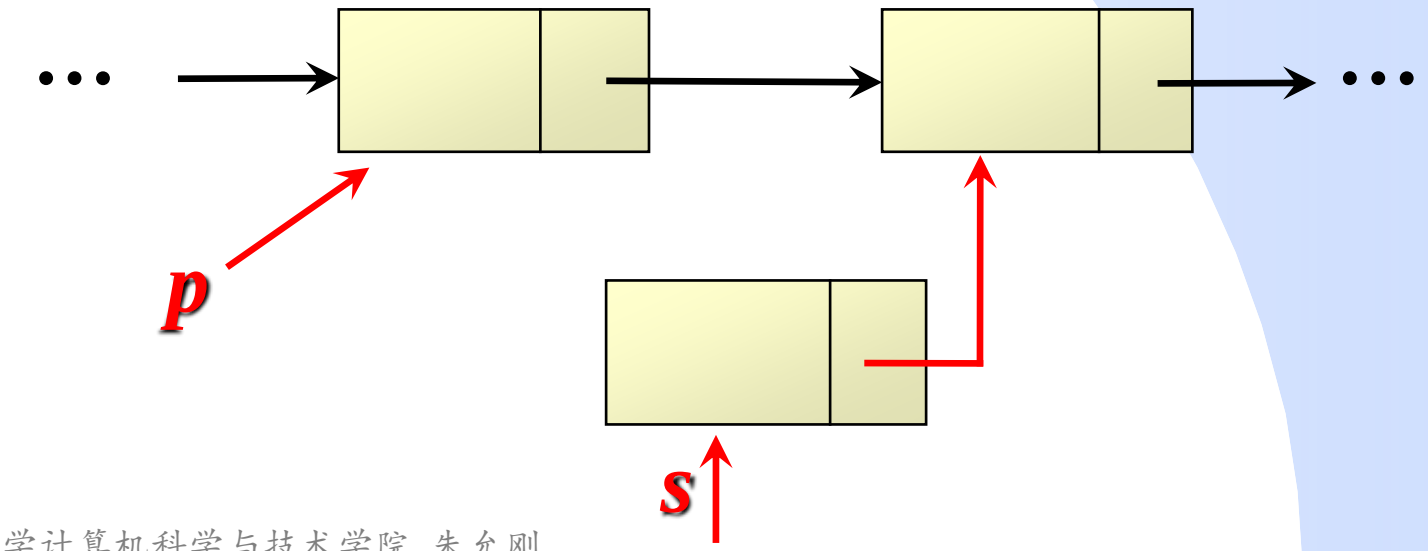
- 在ADL语言中，申请新存储空间的操作用语句“s⇐AVAIL”来描述。
- 释放不用的空间则用语句“AVAIL⇐s”来描述。



插入——在p指向的结点右侧插入结点

A

```
void Insert(ListNode* head, ListNode* p, int K){
    //指针p指向的结点右侧插入数据域为K的新结点
    if(p==NULL) return;
    ListNode* s=new ListNode(K);      //生成新结点s
    s->next = p->next; //s的next指针指向p的后继
```

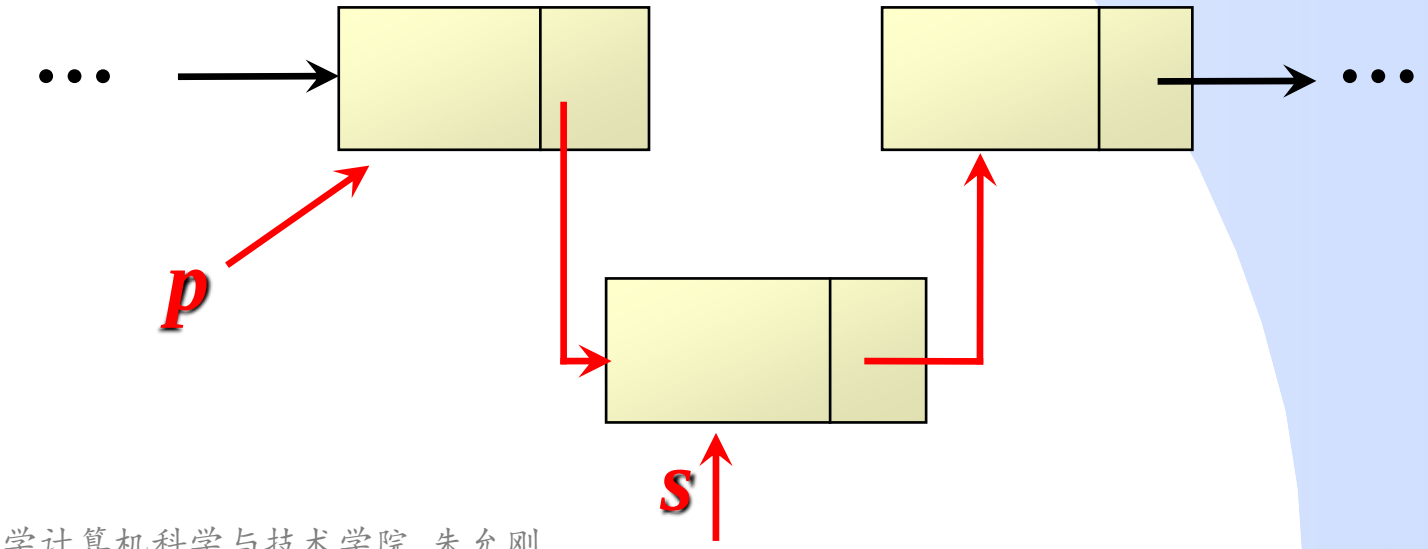


插入——在p指向的结点右侧插入结点

A

```
void Insert(ListNode* head, ListNode* p, int K){
      //指针p指向的结点右侧插入数据域为K的新结点
      if(p==NULL) return;
      ListNode* s=new ListNode(K);      //生成新结点s
      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
}
```

创建链表
连续插入结点



链表的创建——头插法

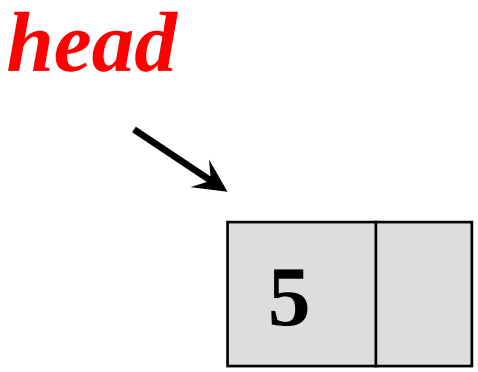
A

```
ListNode* BuildList(int n) {
    //读入n个数，以“头插法”创建包含n个结点的链表，返回表头指针
    ListNode* head = new ListNode(n); //哨位结点可存链表长度
    int K;
    while(n--){
        scanf("%d", &K);
        Insert(head, head, K); //连续在表头插入新结点
    }
    return head;
}
```

链表的创建——头插法

A

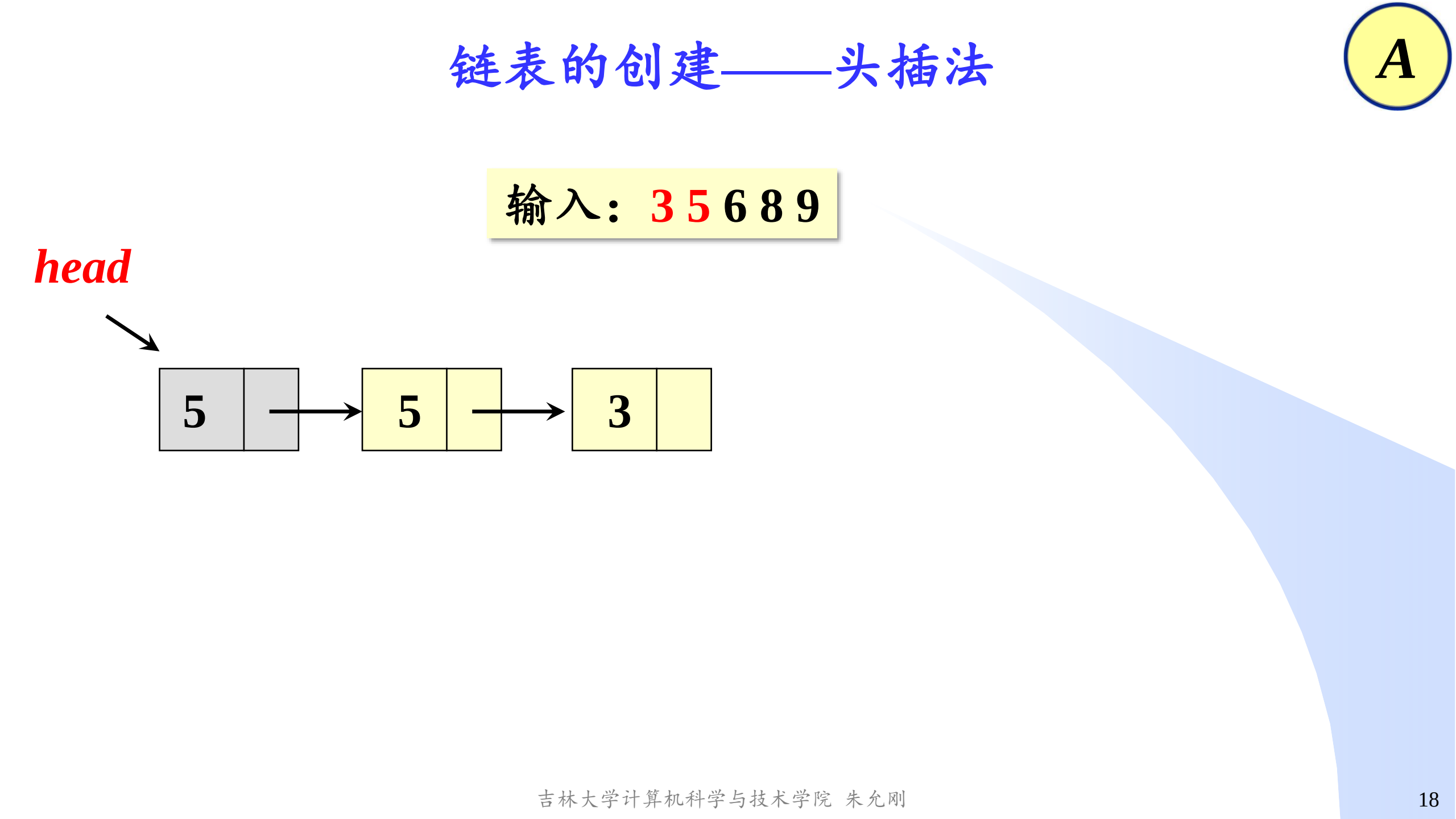
输入：3 5 6 8 9



链表的创建——头插法

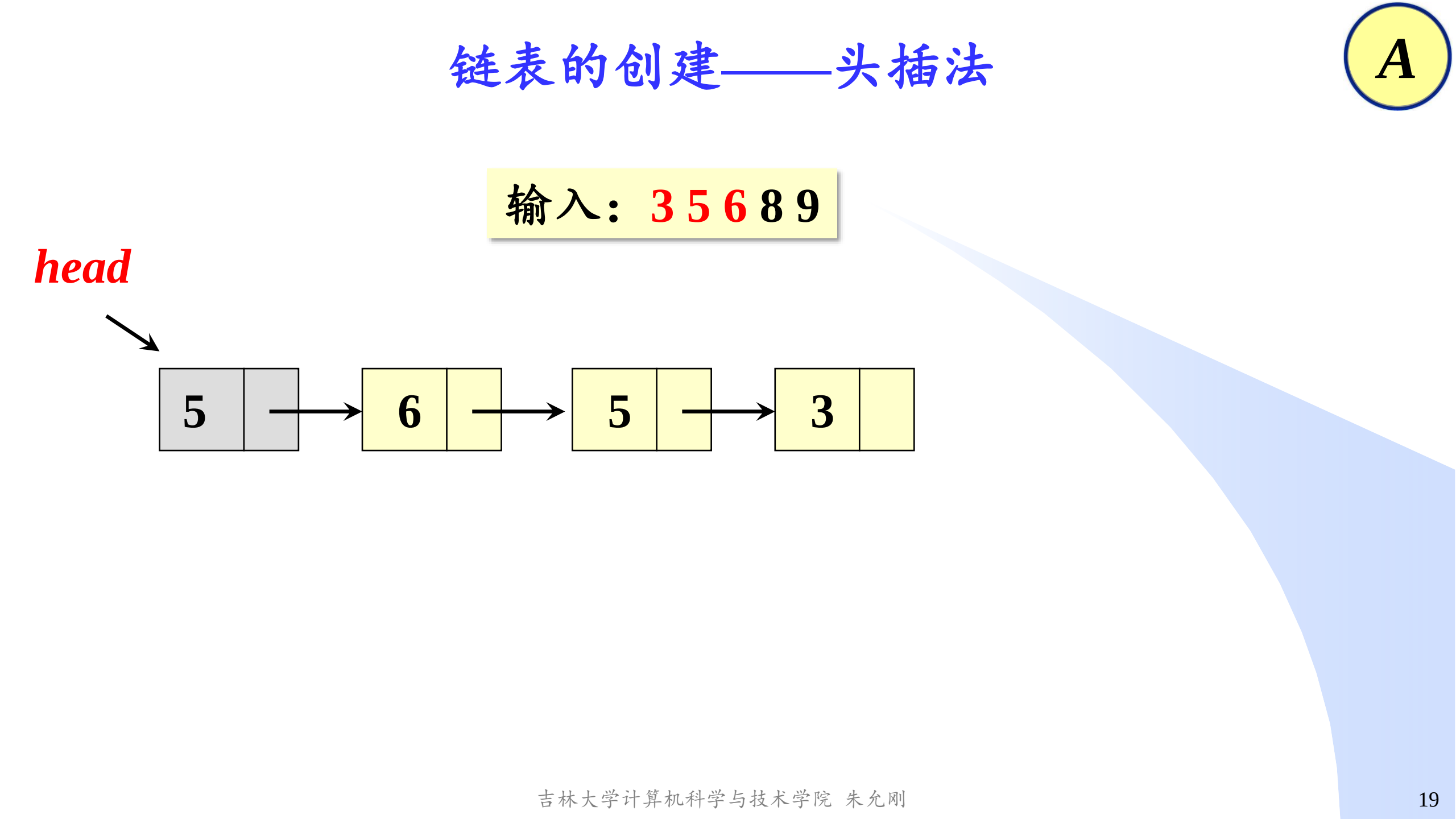
输入：3 5 6 8 9

链表的创建——头插法



输入：3 5 6 8 9

链表的创建——头插法

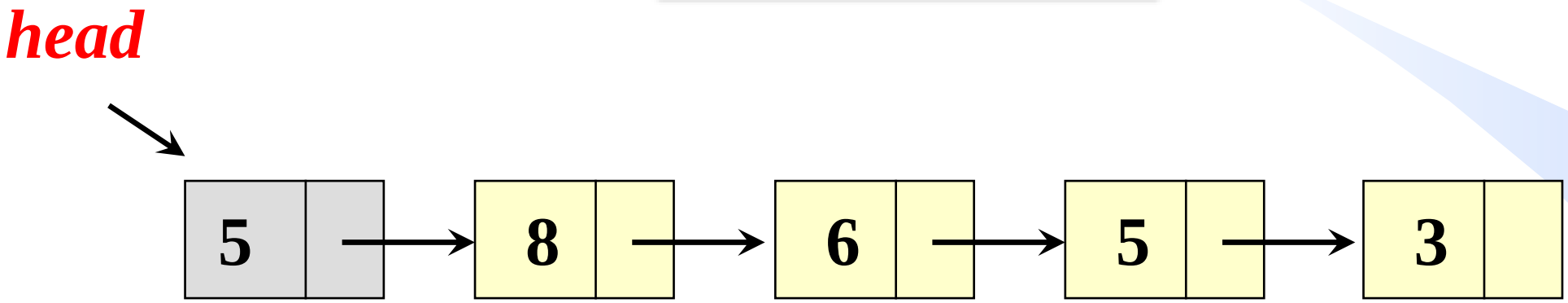


输入：3 5 6 8 9

链表的创建——头插法

A

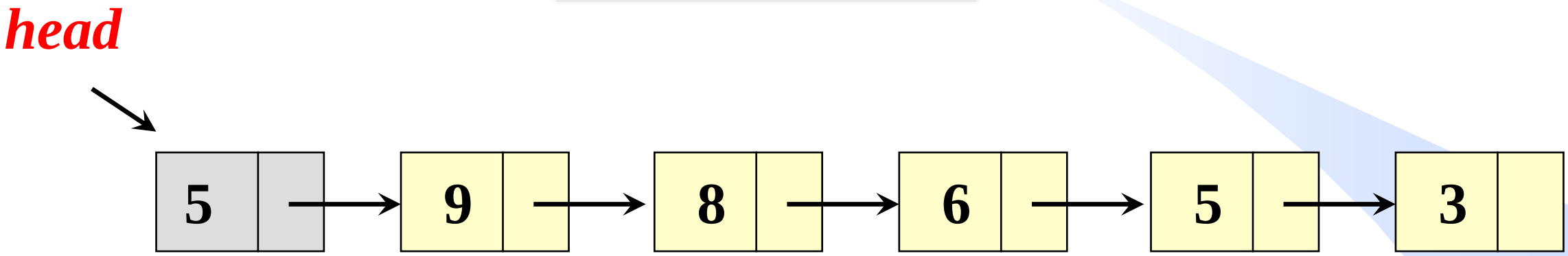
输入：3 5 6 8 9



链表的创建——头插法

A

输入：3 5 6 8 9

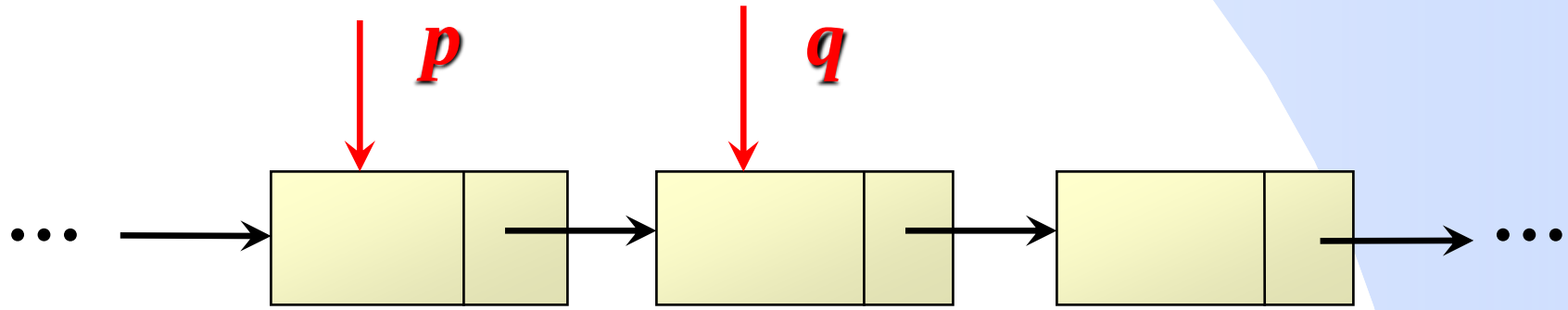


如何按输入的
正序创建链表？

删除——删除p指向的结点的后继结点

A

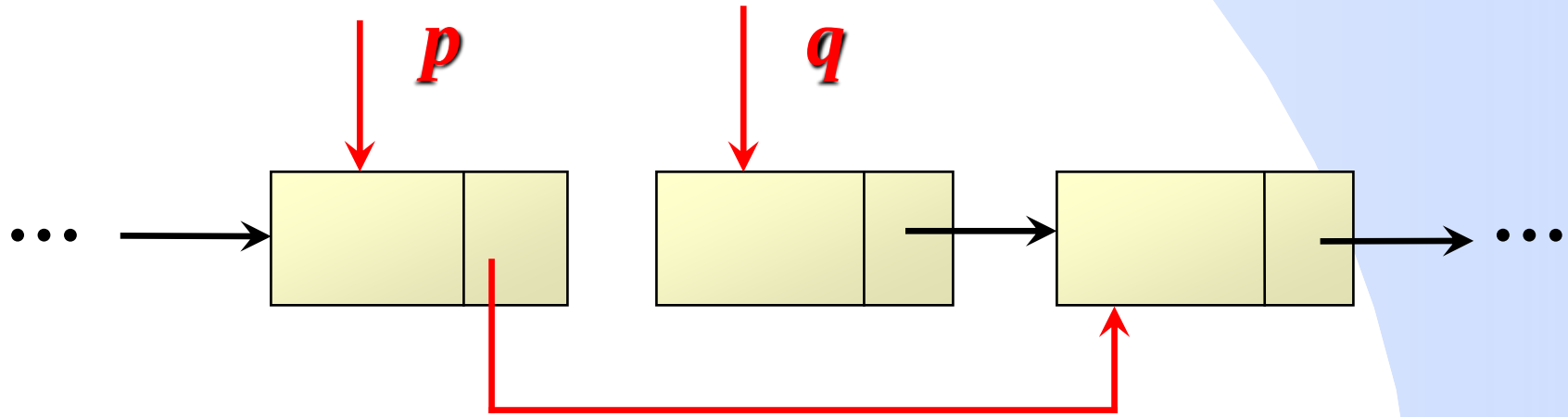
```
void Delete (ListNode* head, ListNode* p){
    if(p==NULL||p->next==NULL)
        return;
    ListNode* q = p->next;
```



删除——删除p指向的结点的后继结点

A

```
void Delete (ListNode* head, ListNode* p){
    if(p==NULL||p->next==NULL)
        return;
    ListNode* q = p->next;
    p->next = q->next;          // 修改p的next指针
```

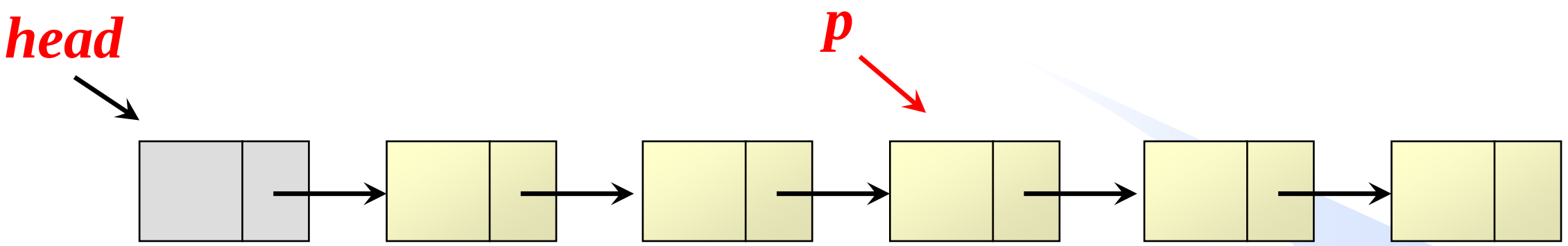


删除——删除p指向的结点的后继结点

A

```
void Delete (ListNode* head, ListNode* p){
     if(p==NULL||p->next==NULL)
          return;
     ListNode* q = p->next;
     p->next = q->next;          // 修改p的next指针
     delete q;                   // 释放q存储空间
}
```

删除——删除p指向的结点



正常做法

先找到p的前驱

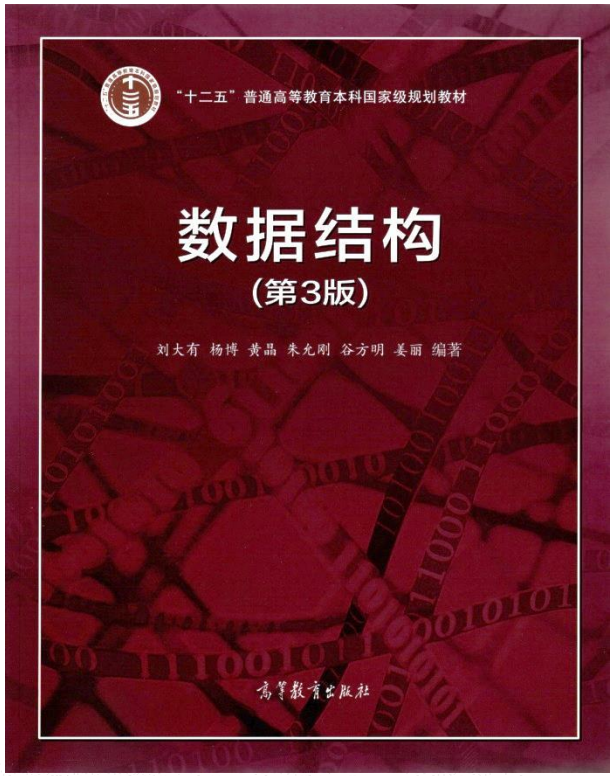
最坏时间复杂度O(n)

变通做法

若p非表尾,将p后继的数据值赋给p,然后删除p后继

最坏时间复杂度O(1)

LeetCode 237



线性表的链接存储

- 单链表
- **循环链表**
- 双向链表

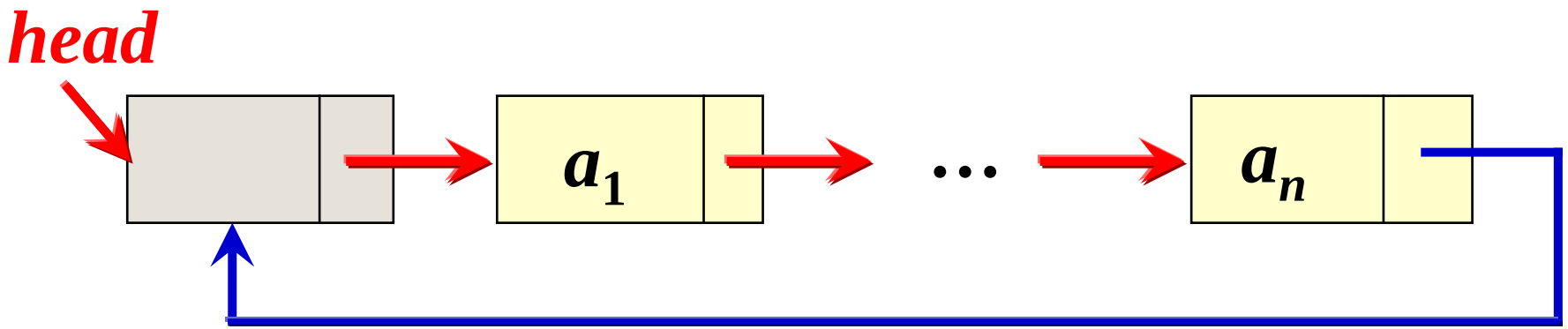
数据之法 结构之美 算法之道

zhuyungang@jlu.edu.cn

循环链表（Circular Linked List）

A

- 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，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。

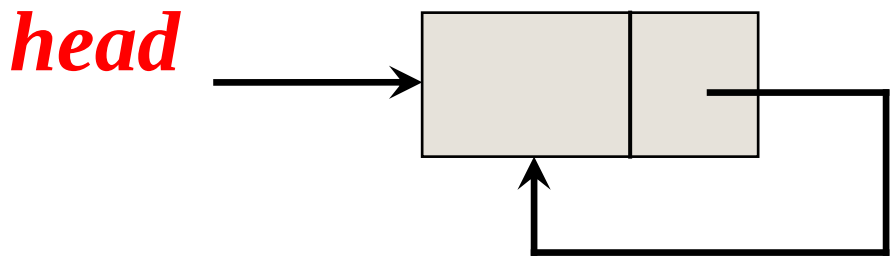


判断表空（假设包含哨位结点）

A

单链表： head->next == NULL

循环链表： head->next == head

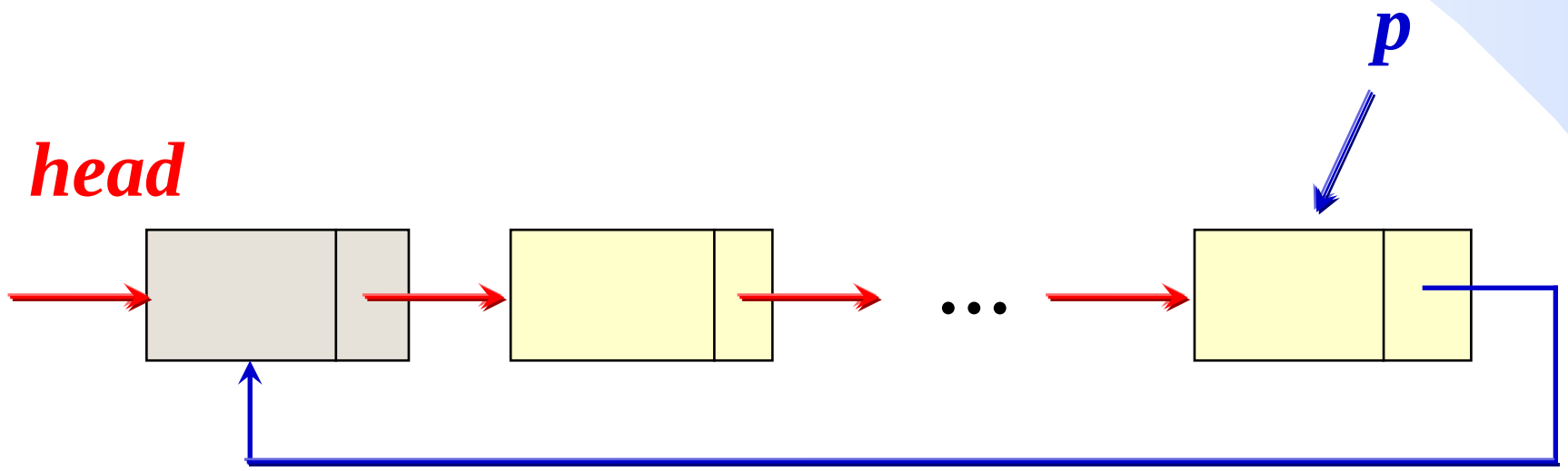


判断表尾（假设包含哨位结点）

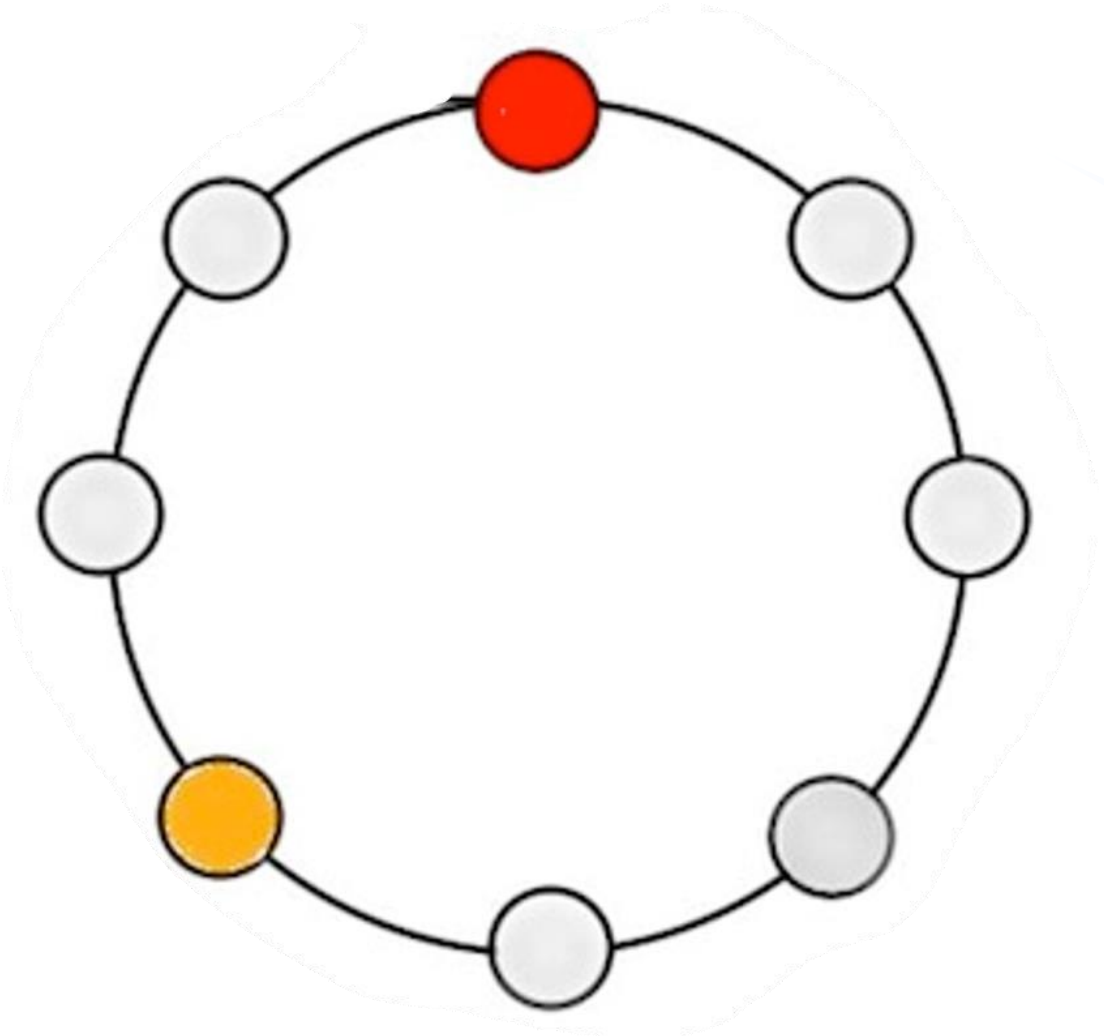
A

单链表： p->next == NULL

循环链表： p->next == head



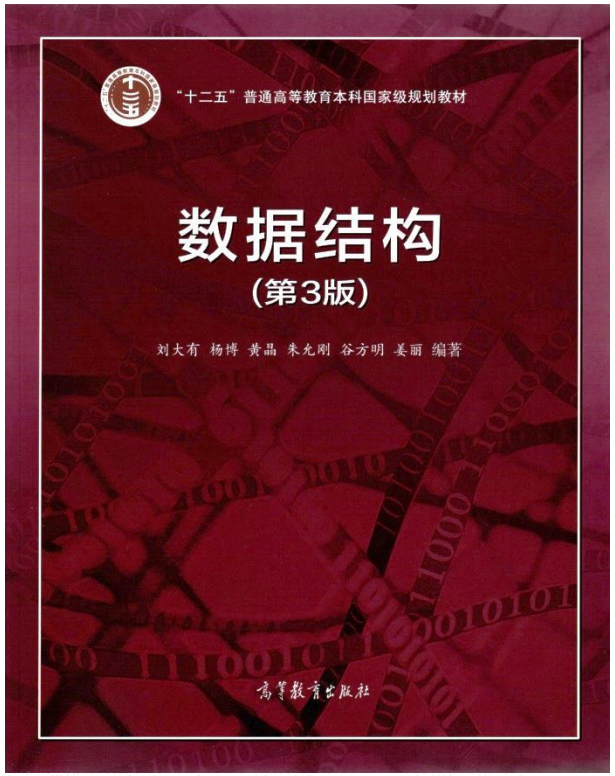
B



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

A

约瑟夫问题



线性表的链接存储

- 单链表
- 循环链表
- **双向链表**

数据之法 结构之美 算法之道

zhuyungang@jlu.edu.cn

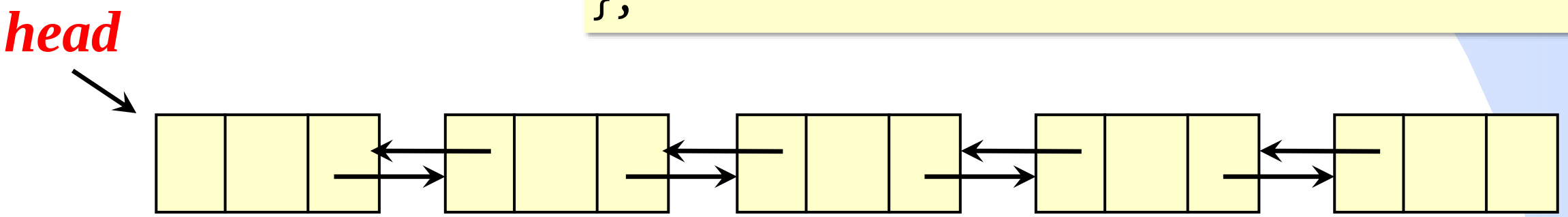
Last updated on 2024.9

双向链表（Doubly Linked List）

B

- 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。


```
struct ListNode{
    int data;
    ListNode* prev;
    ListNode* next;
    ListNode(int d){data=d; prev=next=NULL;}
};
```

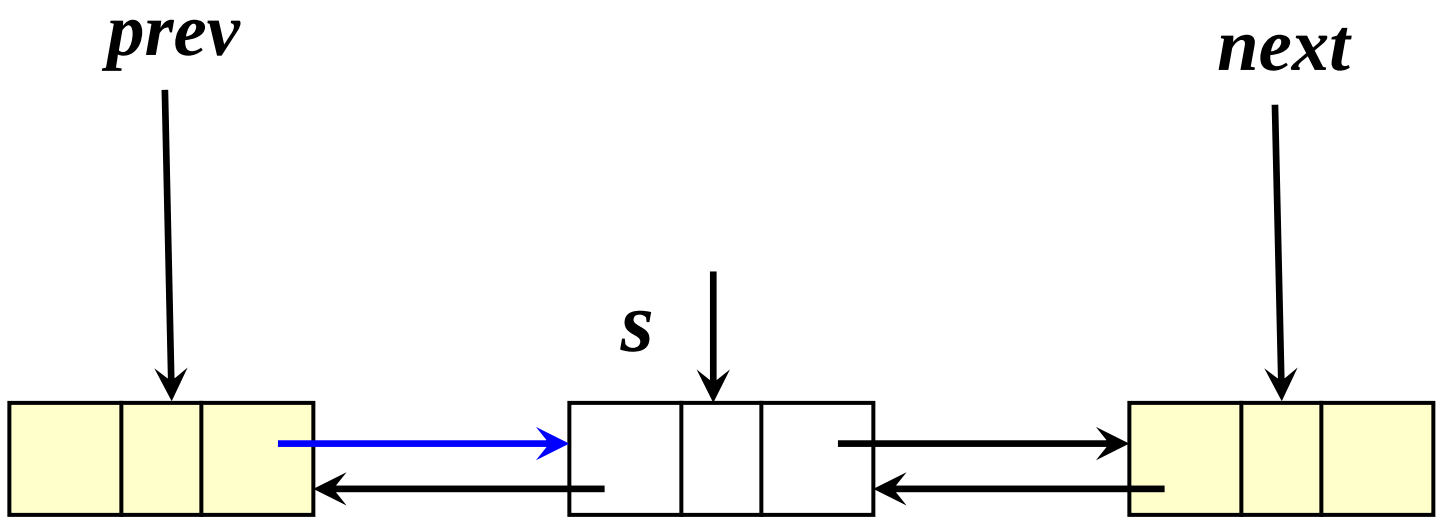


删除——删除指针s指向的结点

B

```
ListNode* prev = s->prev;
ListNode* next = s->next;
```

常规情况

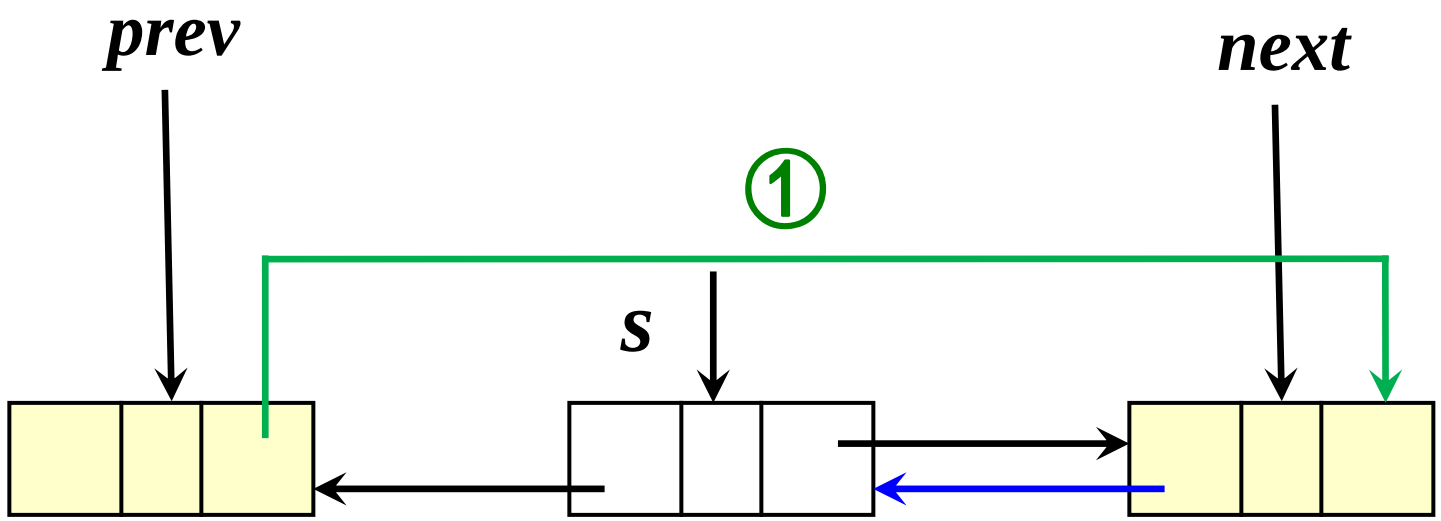


删除——删除指针s指向的结点

B

```
ListNode* prev = s->prev;
ListNode* next = s->next;
prev->next = next;   //步骤①
```

常规情况

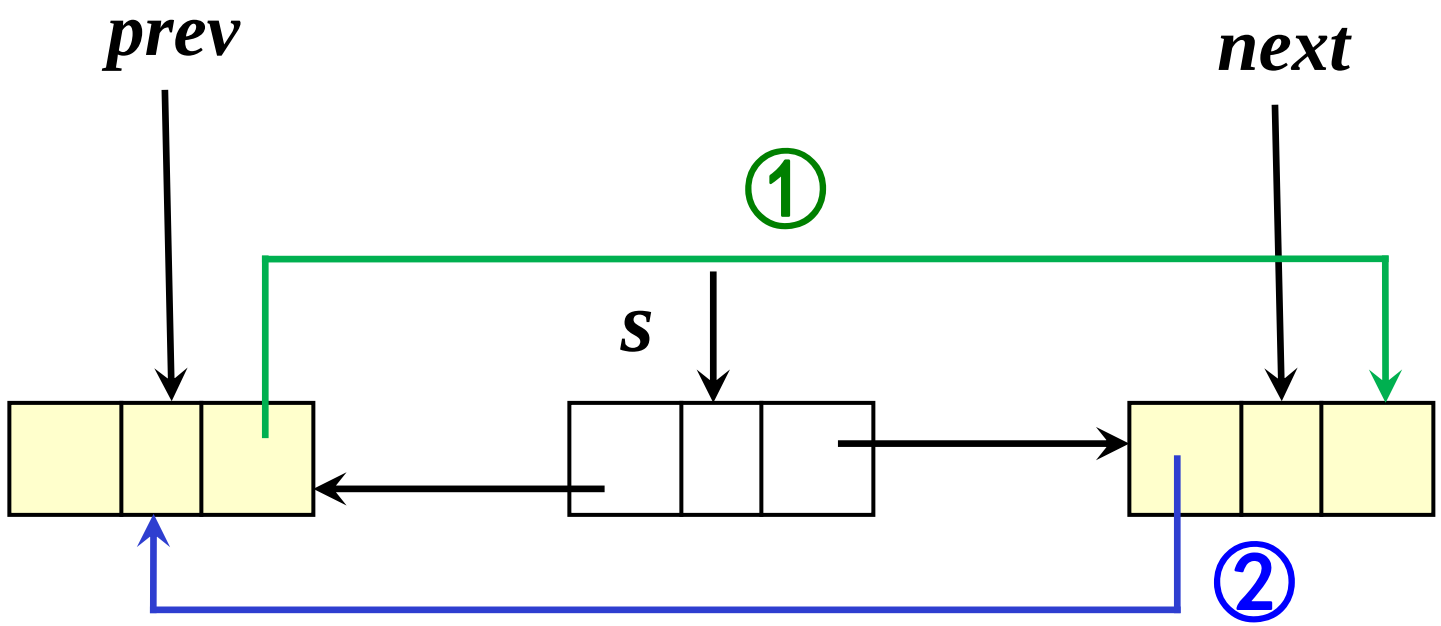


删除——删除指针s指向的结点

B

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

常规情况

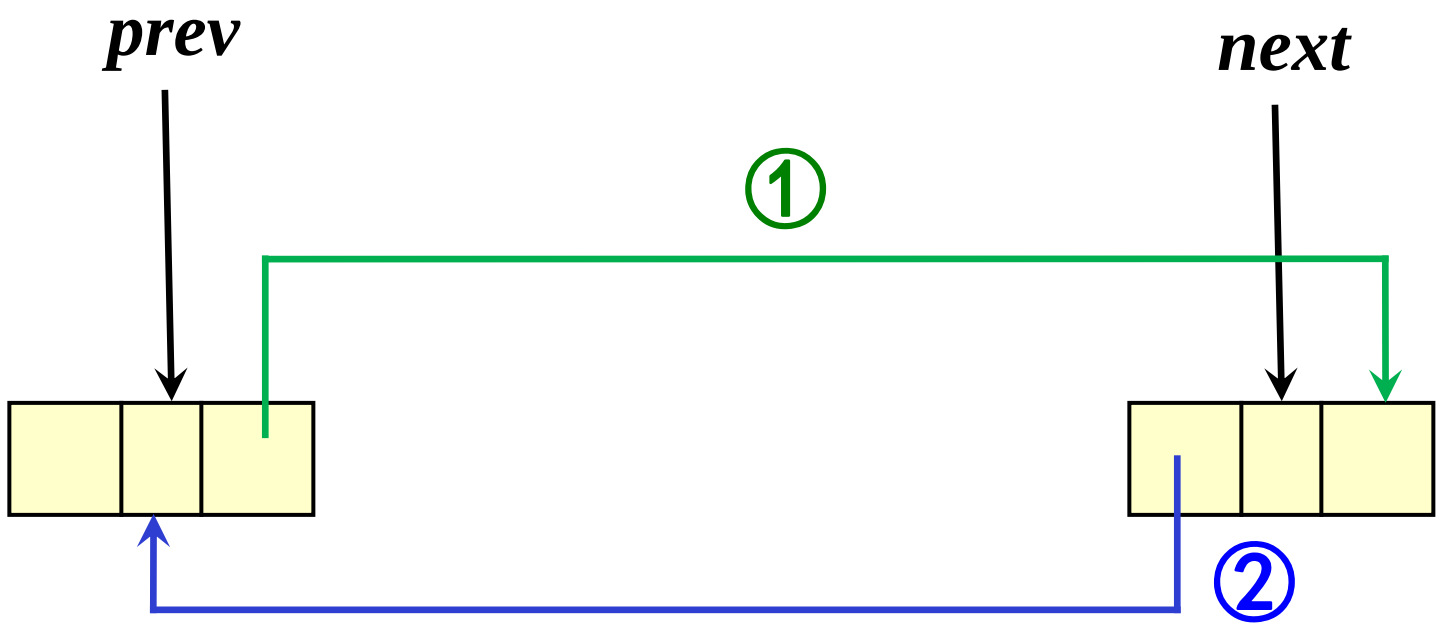


删除——删除指针s指向的结点

B

常规情况

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



删除——删除指针s指向的结点

B

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

s无前驱，导致该指针为空

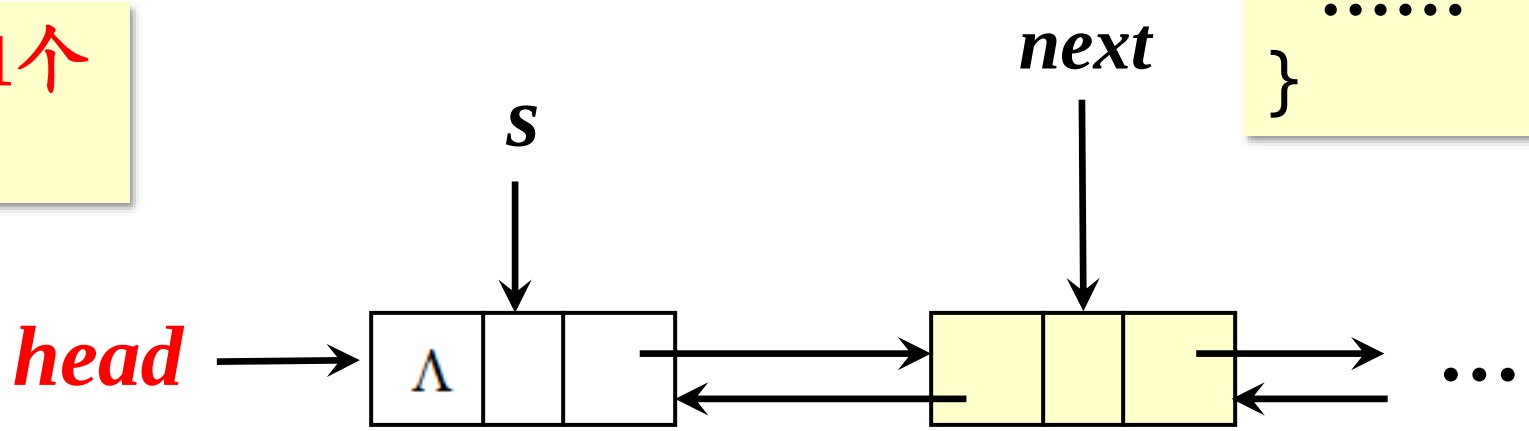
空指针

特殊情况

若s是第1个结点？

方案1：再加一段代码对这种情况单独处理

```
if(s是第1个结点){
  ……
}
```



删除——删除指针s指向的结点

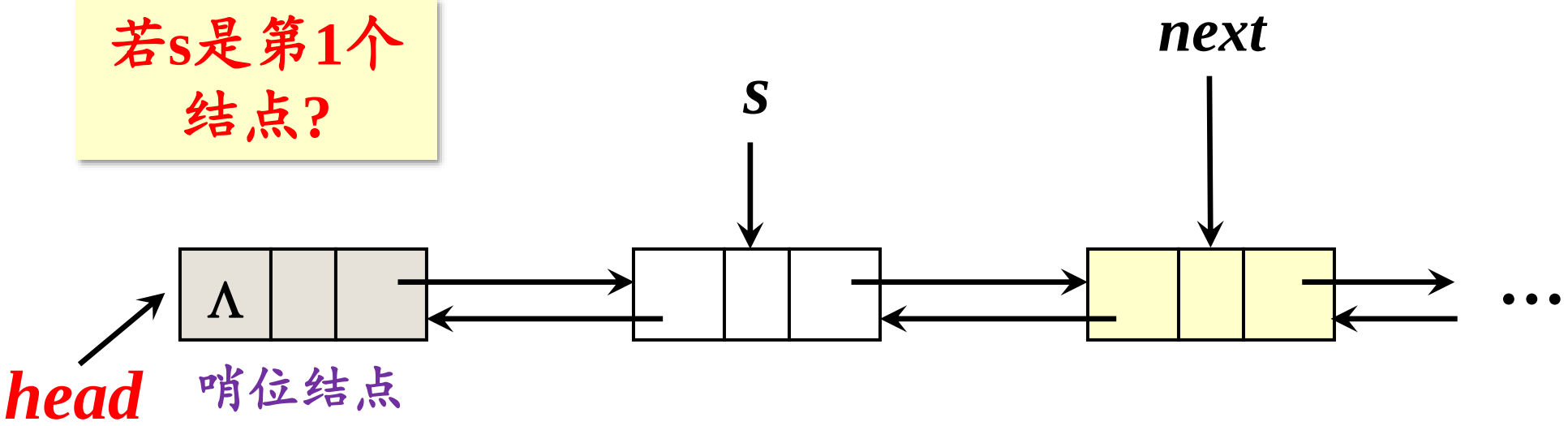
B

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

方案2：表头引入哨位结点，使s始终有前驱，原代码仍然有效

特殊情况

若s是第1个结点?



删除——删除指针s指向的结点

B

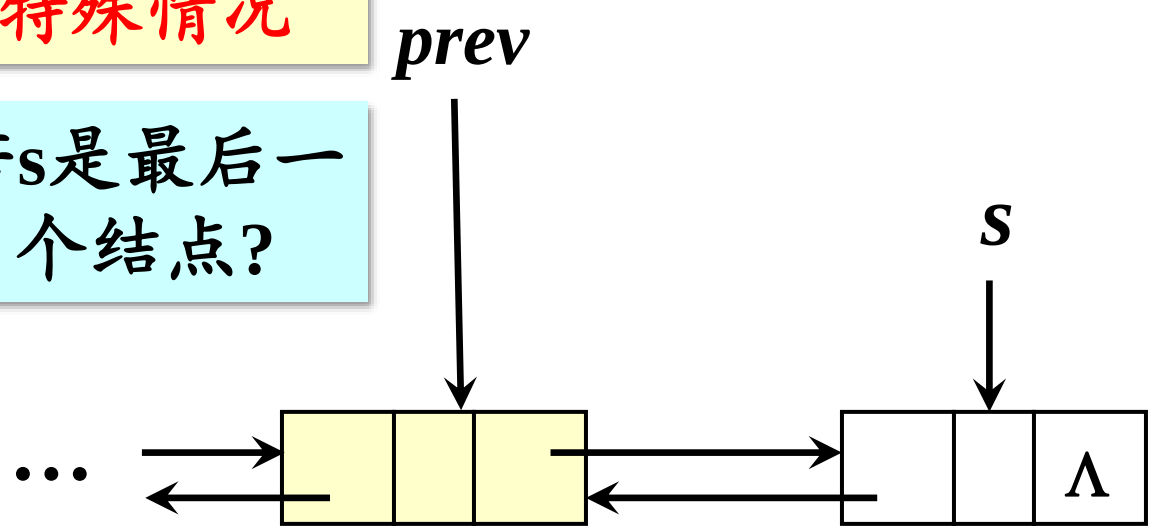
```
ListNode* prev = s->prev;
ListNode* next = s->next;
prev->next = next;
next->prev = prev;
delete s;
```

s无后继，导致该指针为空

空指针

特殊情况

若s是最后一个结点？



方案1：再加一段代码对这种情况单独处理

```
if(s是尾结点){
  ……
}
```

删除——删除指针s指向的结点

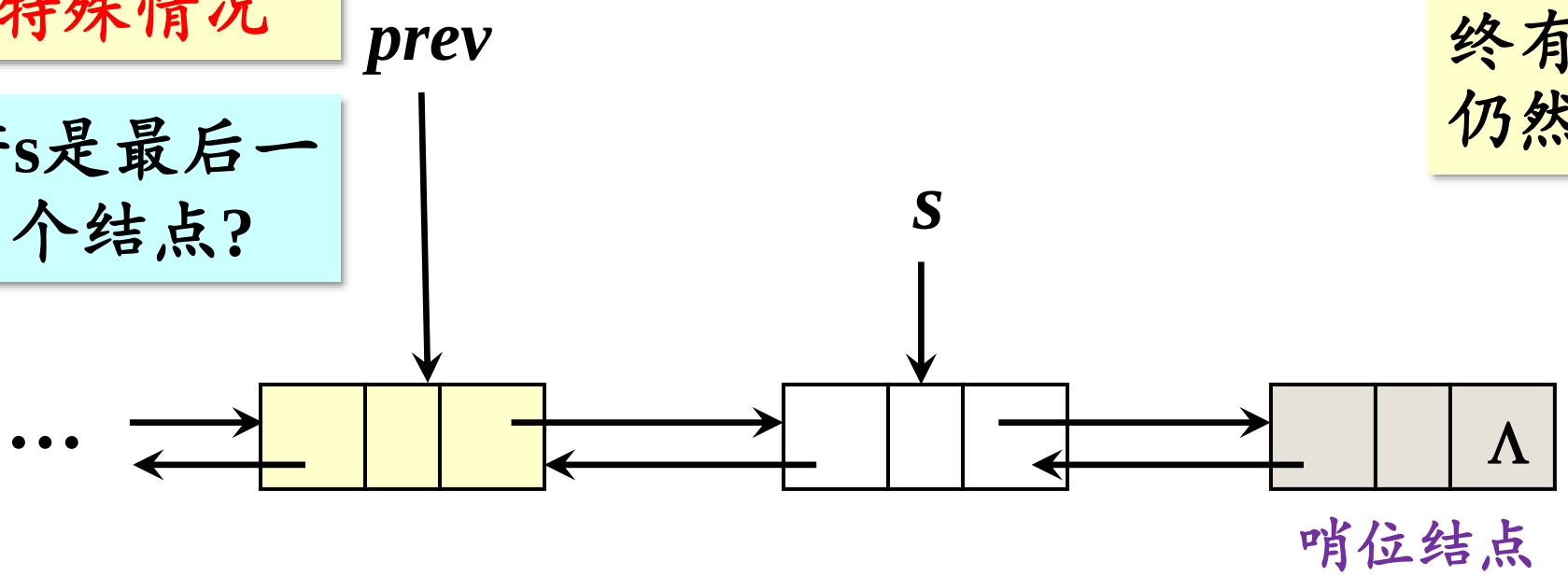
B

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

特殊情况

若s是最后一个结点?

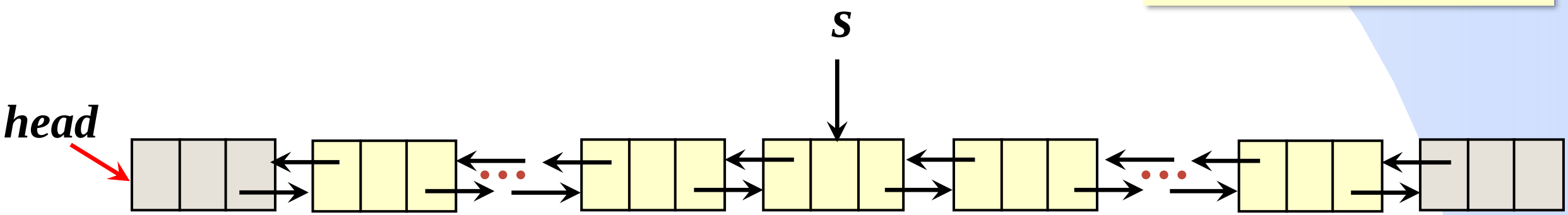
方案2：表尾引入哨位结点，使s始终有后继，原代码仍然有效



删除——删除指针s指向的结点

B

缺点：需2个哨位结点

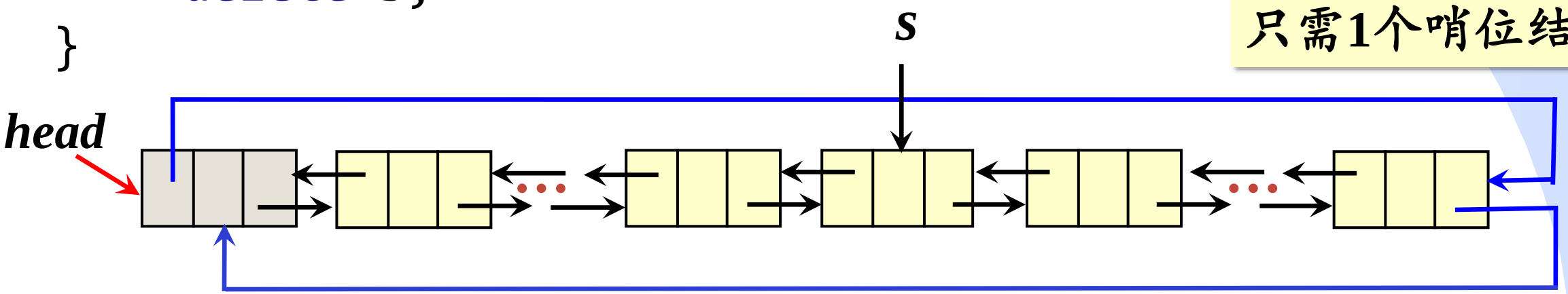


删除——删除指针s指向的结点

B

```
void DeleteNode(ListNode* head, ListNode* s){
    // 删除带哨位结点的双向循环链表中的结点s
    if(s==NULL||s==head) return;        //不能删哨位结点
    ListNode* prev = s->prev;
    ListNode* next = s->next;
    prev->next = next;
    next->prev = prev;
    delete s;
}
```

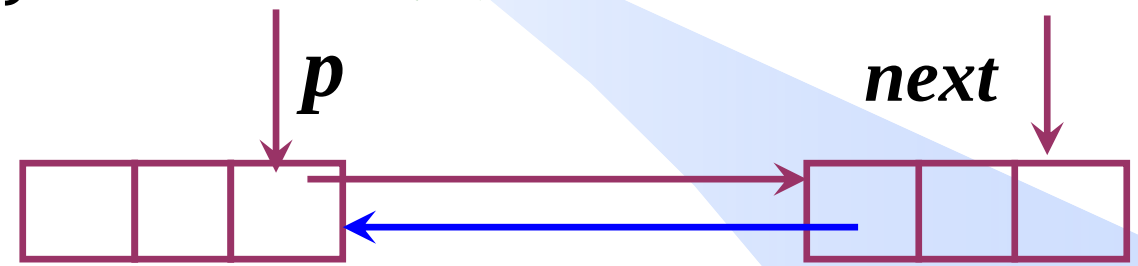
解决方案：采用双向循环链表，从而只需1个哨位结点

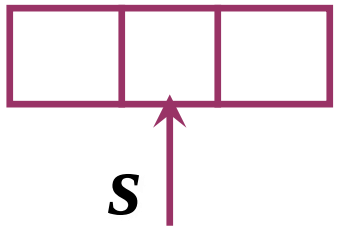


插入——在指针p指向的结点右侧插入结点

B

```
void InsertNode(ListNode* head, ListNode* p, int K){
    //在带哨位结点的双向循环链表中结点p右侧插入数据值为K的结点
    if(p==NULL) return;
    ListNode* s=new ListNode(K);        //生成新结点s
    ListNode* next = p->next;
```

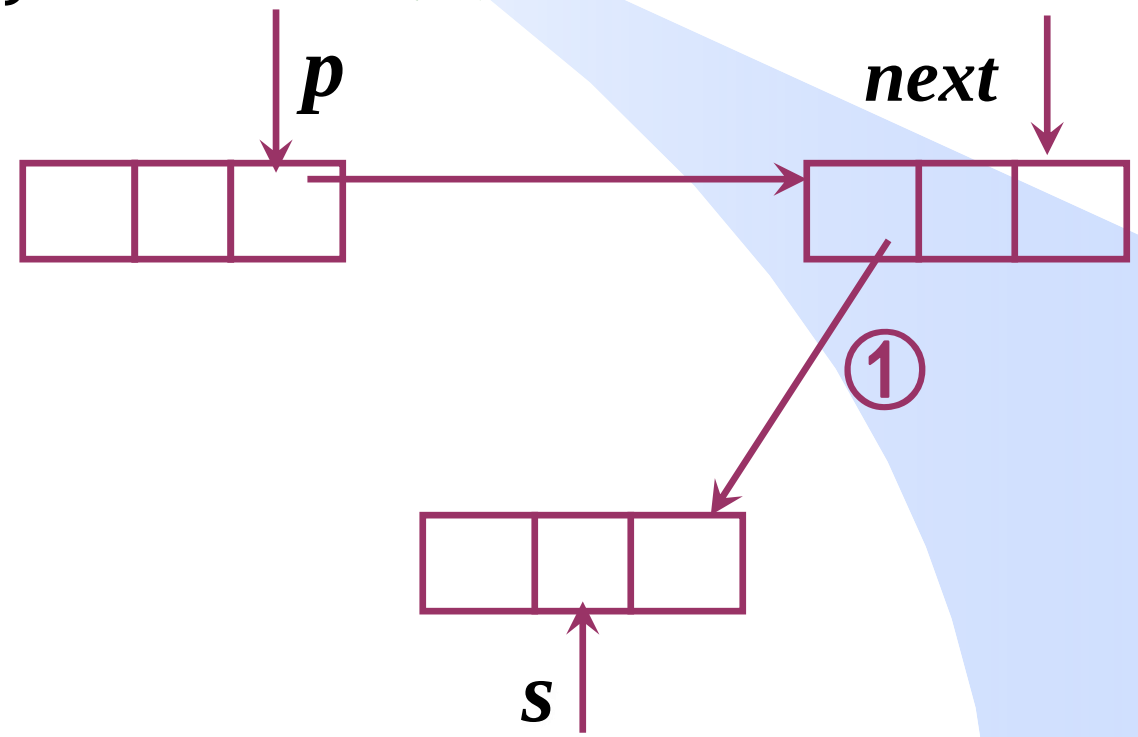




插入——在指针p指向的结点右侧插入结点

B

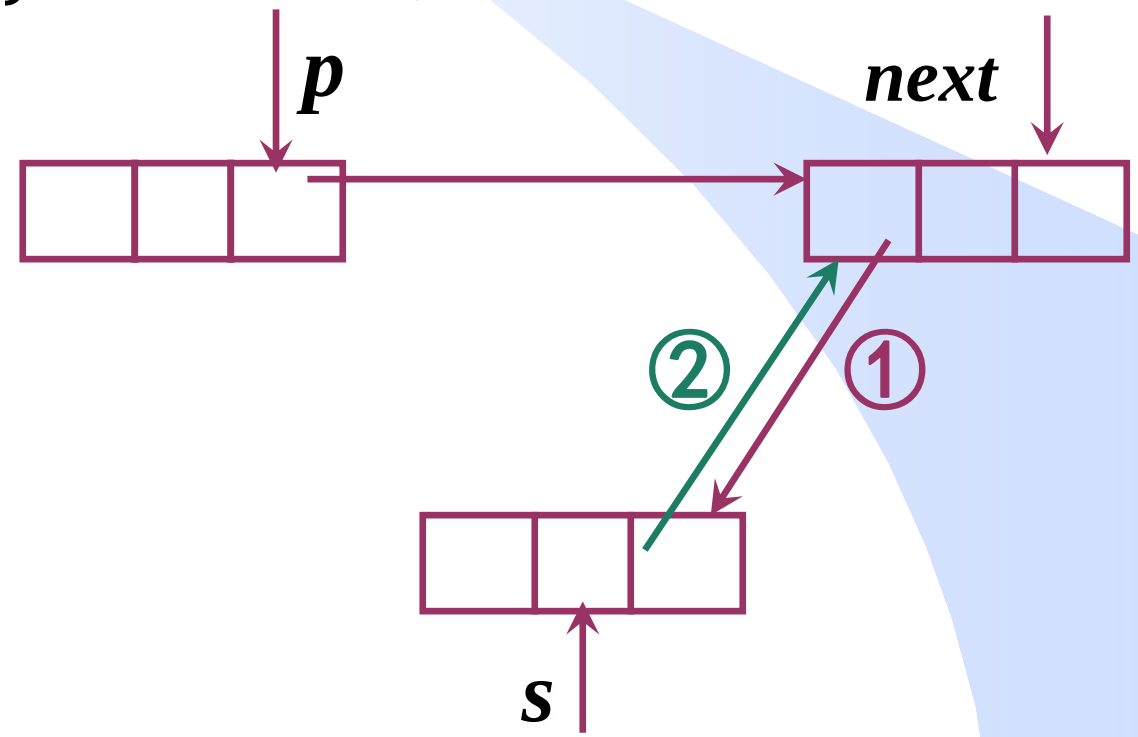
```
void InsertNode(ListNode* head, ListNode* p, int K){
    //在带哨位结点的双向循环链表中结点p右侧插入数据值为K的结点
    if(p==NULL) return;
    ListNode* s=new ListNode(K);        //生成新结点s
    ListNode* next = p->next;
    next->prev = s;      //步骤①
```



插入——在指针p指向的结点右侧插入结点

B

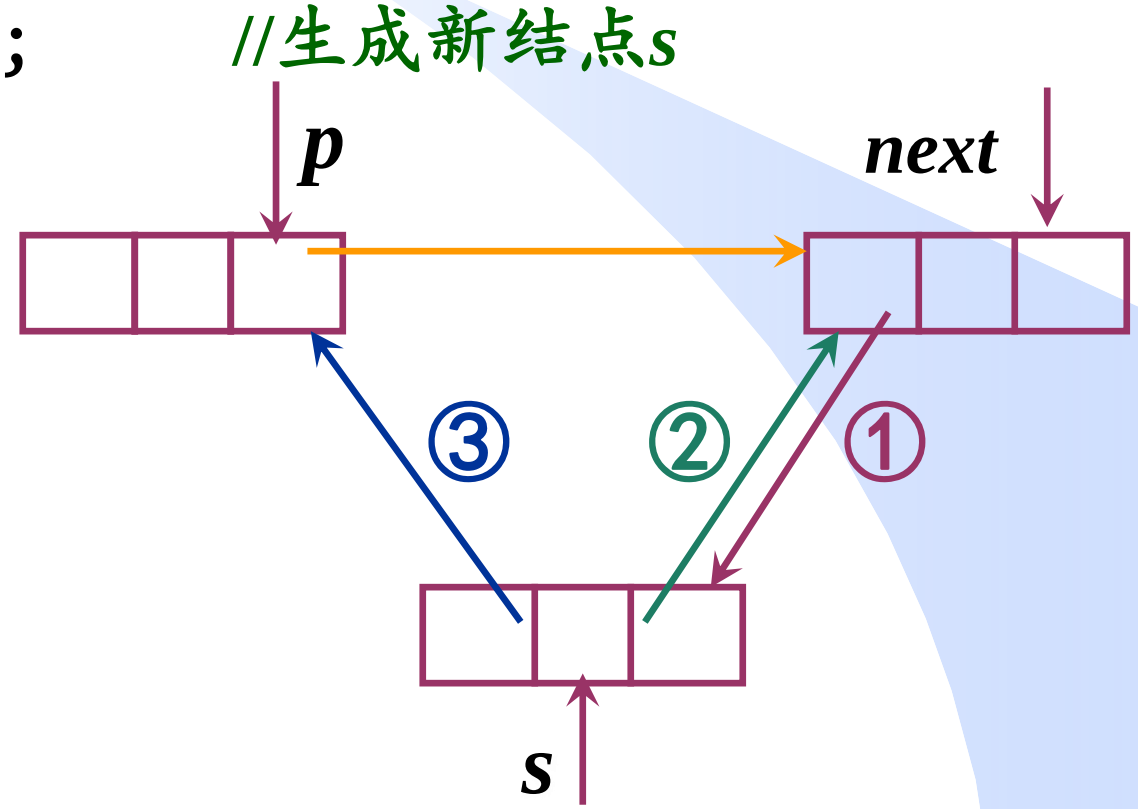
```
void InsertNode(ListNode* head, ListNode* p, int K){
    //在带哨位结点的双向循环链表中结点p右侧插入数据值为K的结点
    if(p==NULL) return;
    ListNode* s=new ListNode(K);        //生成新结点s
    ListNode* next = p->next;
    next->prev = s;                     //步骤①
    s->next = next;                     //步骤②
```



插入——在指针p指向的结点右侧插入结点

B

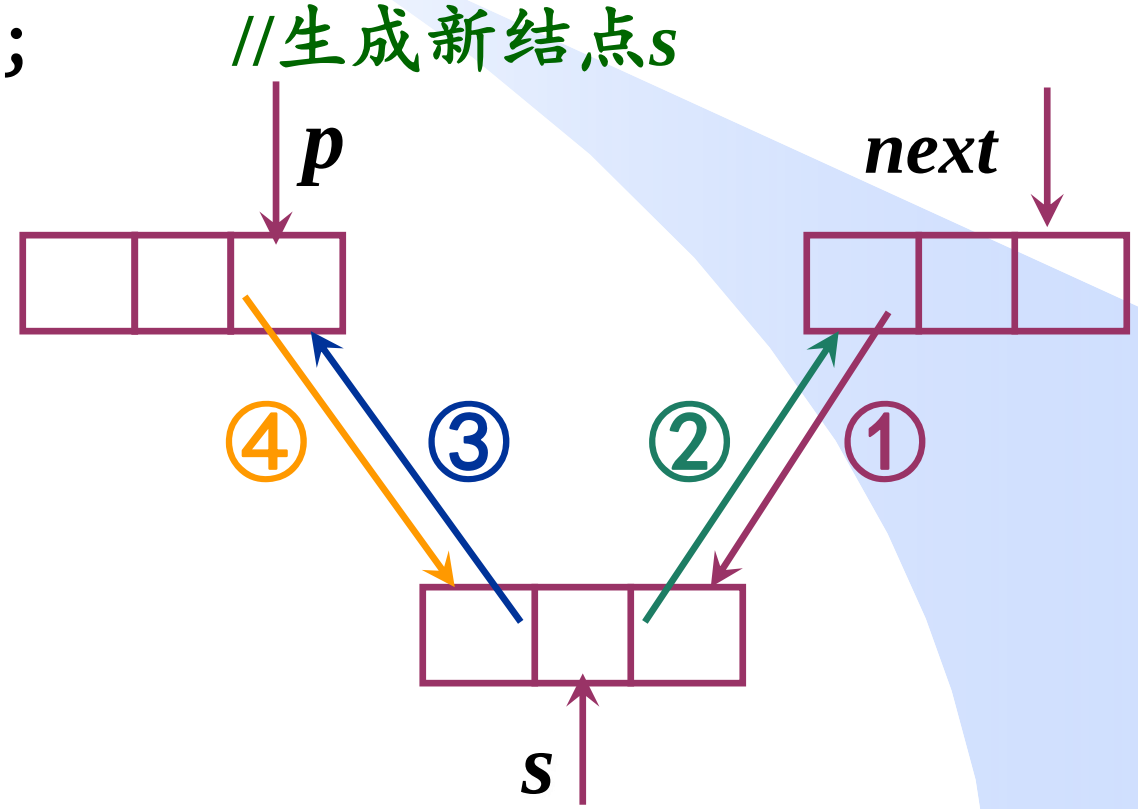
```
void InsertNode(ListNode* head, ListNode* p, int K){
    //在带哨位结点的双向循环链表中结点p右侧插入数据值为K的结点
    if(p==NULL) return;
    ListNode* s=new ListNode(K);        //生成新结点s
    ListNode* next = p->next;
    next->prev = s;                     //步骤①
    s->next = next;                     //步骤②
    s->prev = p;                        //步骤③
```



插入——在指针p指向的结点右侧插入结点

B

```
void InsertNode(ListNode* head, ListNode* p, int K){
    //在带哨位结点的双向循环链表中结点p右侧插入数据值为K的结点
    if(p==NULL) return;
    ListNode* s=new ListNode(K);        //生成新结点s
    ListNode* next = p->next;
    next->prev = s;                     //步骤①
    s->next = next;                     //步骤②
    s->prev = p;                        //步骤③
    p->next = s;                        //步骤④
}
```



顺序存储和链式存储的比较

B

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

B

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

- 当需要在某特定位置频繁插入、删除时，链表的效率较高；
- 当需要频繁进行基于下标的查找时，则顺序表的效率较高。

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;
}
```

链表编程时留意边界条件处理

B

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

D

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

链表应用场景举例

核心逻辑：

```
while(pBullet != NULL){
    while(pEnemy != NULL){
        if(pBullet和pEnemy相遇){
            //子弹打中敌机
            删除pBullet所指结点;
            删除pEnemy所指结点;
            break;
        }
        pEnemy = pEnemy->next;
    }
    pBullet = pBullet->next;
}
```