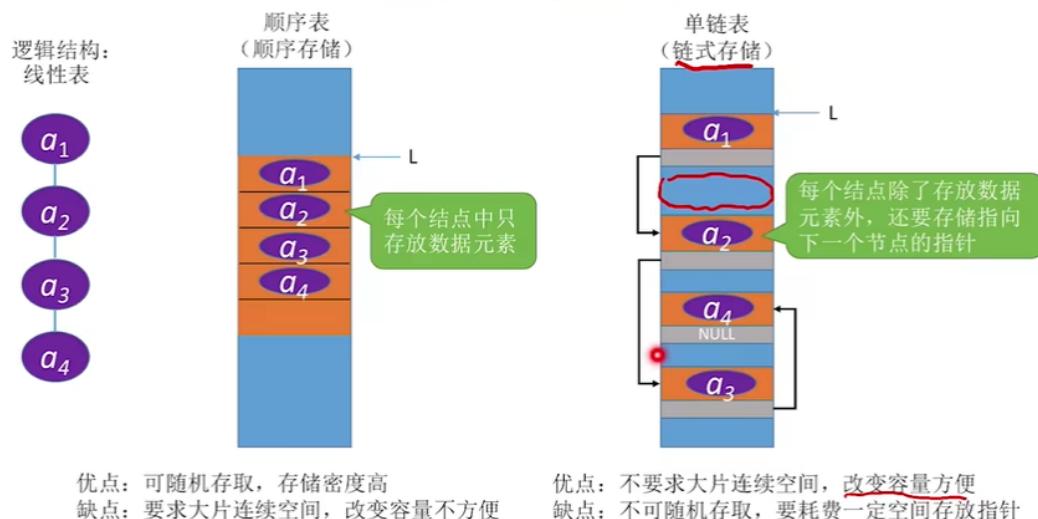


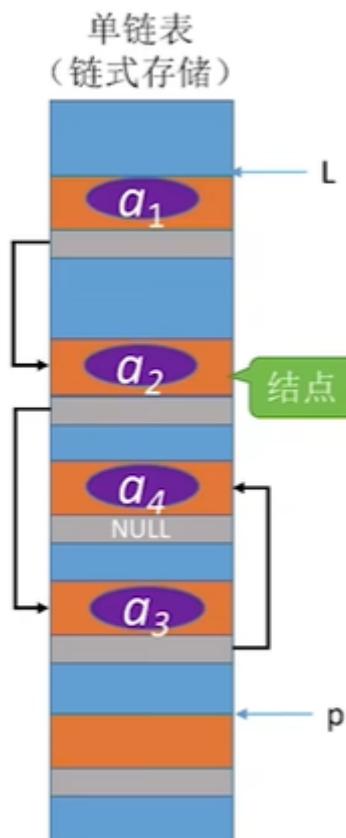
单链表

• 单链表的定义

• 什么是单链表



• 用代码定义一个单链表



```
1 struct LNode           // 定义单链表结点类型
2 {
3     ELEMTYPE data;      // 每个节点存放一个数据元素
4     struct LNode *next; // 指针指向下一个节点
5 };
6
```

```

7 //增加一个新结点：在内存中申请一个结点所需空间，并用指针p指向这个点
8 struct LNode * p = (struct LNode *)malloc(sizeof(struct LNode));
9
10 //用typedef重命名
11 typedef struct LNode LNode;
12 LNode * p = (LNode *)malloc(sizeof(LNode));
13
14 //更简洁的方式举例
15 typedef struct LNode
16 {
17     ELEMType data;
18     struct LNode *next;
19 }LNode,*LinkList;           //将struct LNode重命名为LNode，并且用
LinkList表示指向struct LNode的指针（下同）
20 //上面代码等同于下面的代码
21 struct LNode
22 {
23     ELEMType data;
24     struct LNode *next;
25 };
26 typedef struct LNode LNode;
27 typedef struct LNode *LinkList;

```

要表示一个单链表时，只需声明一个头指针L，指向单链表的第一个结点

```

1 LNode * L;      //声明一个指向单链表第一个结点的指针
2 //或者下面的表示方法也一样
3 LinkList L;    //声明一个指向单链表第一个结点的指针
4
5 //e.q:
6 LNode * GetElem(LinkList L,int i)
7 {
8     int j = 1;
9     LNode *p = L->next;
10
11    if(i == 0)
12        return L;
13    if(i < 1)
14        return NULL;
15
16    while(p != NULL && j < i)
17    {
18        p = p->next;
19        j++;
20    }
21    return p;
22 }

```

强调这是一个单链表 使用LinkList

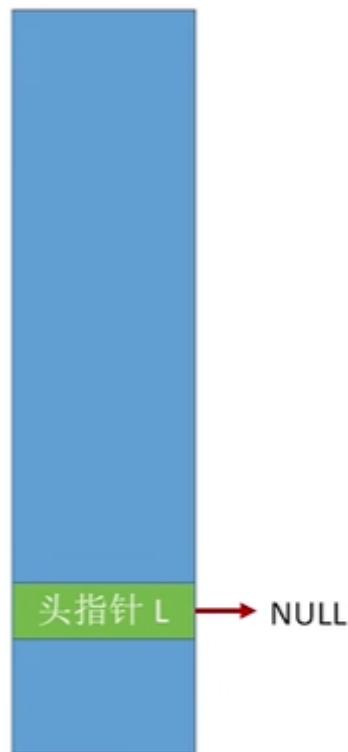
强调这是一个结点使用LNode *

但是这两种表达都是表示指向struct LNode的指针，只是强调的点不一样

- 不带头结点的单链表

```
1 | typedef struct LNode
2 | {
3 |     ELEMTYPE data;
4 |     struct LNode *next;
5 | }LNode,*LinkList;
6 |
7 | //初始化一个空的单链表
8 | bool InitList(LinkList &L)
9 | {
10 |     L = NULL;      //空表，暂时还没有任何结点，防止脏数据
11 |     return true;
12 | }
13 |
14 | //判断单链表是否为空
15 | bool Empty(LinkList L)
16 | {
17 |     if (L == NULL)
18 |         return true;
19 |     else
20 |         return false;
21 | }
22 | //或者
23 | bool Empty(LinkList L)      //这个布尔函数的值本身就是TRUE或FALSE
24 | {
25 |     return (L == NULL);
26 | }
27 |
28 | void test()
29 | {
30 |     LinkList L;      //注意，此处并没有创建一个结点
31 |     //初始化一个空表
32 |     InitList(L);
33 |     //...后续代码...
34 | }
```

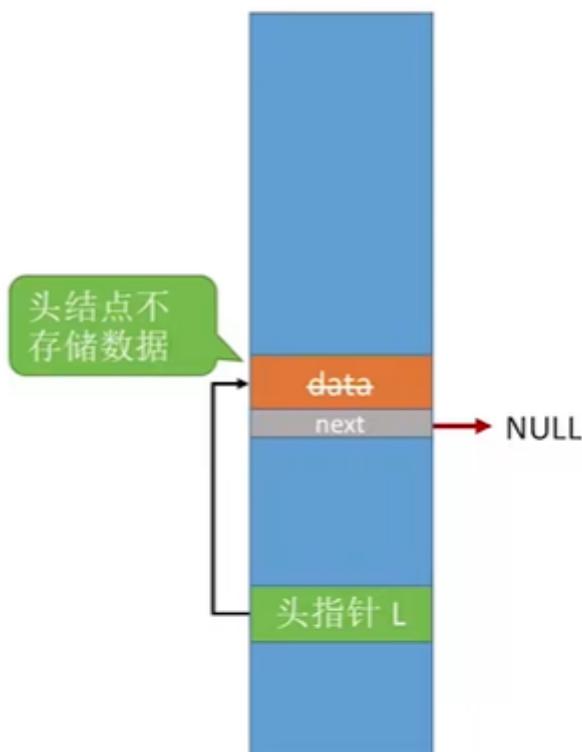
内存



- 带头结点的单链表

```
1  typedef struct LNode
2  {
3      ELEMTYPE data;
4      struct LNode *next;
5  }LNode,*LinkList;
6
7 //初始化一个单链表（带头结点）
8 bool InitList(LinkList &L)
9 {
10     L = (LNode *)malloc(sizeof(LNode));      //分配一个头结点
11
12     if(L == NULL)
13         return false;
14
15     L->next = NULL;      //头结点之后暂时还没有结点
16     return true;
17 }
18
19 //判断单链表是否为空（带头结点）
20 bool Empty(LinkList L)
21 {
22     return (L->next == NULL);
23 }
24
25 void test()
26 {
27     LinkList L;      //注意，此处并没有创建一个结点
28     //初始化一个空表
29     InitList(L);
30     //...后续代码...
}
```

内存



不带头结点 V.S. 带头结点

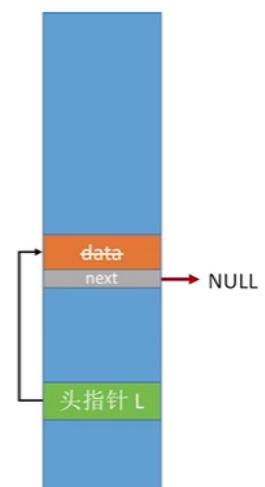
不带头



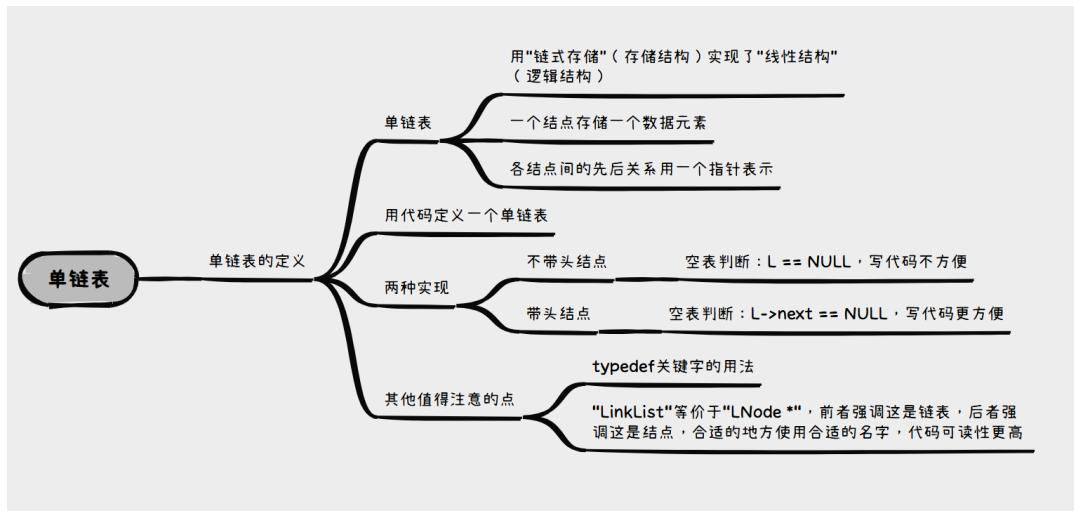
带头



童叟无欺

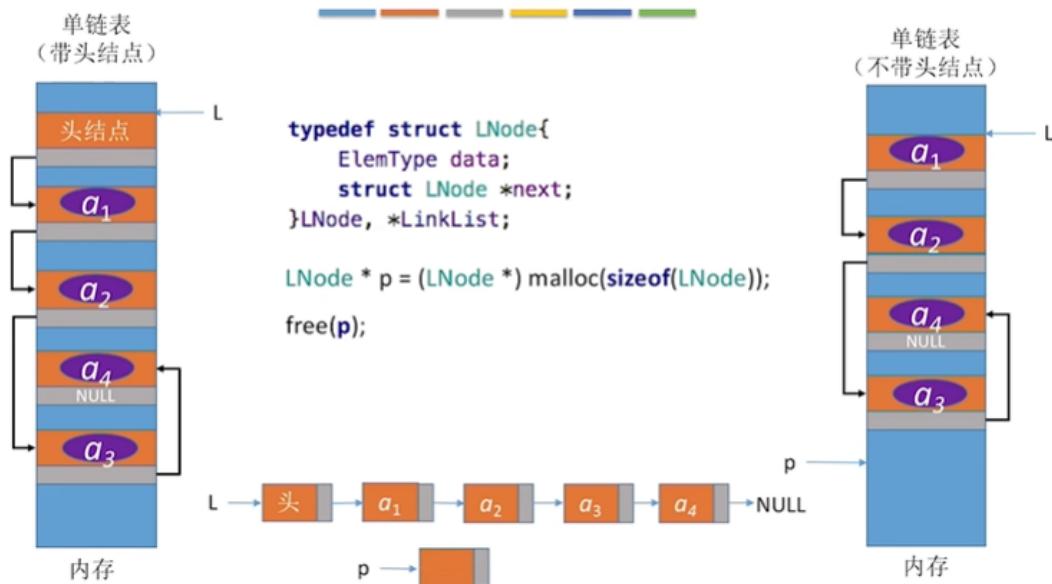


- 知识总结



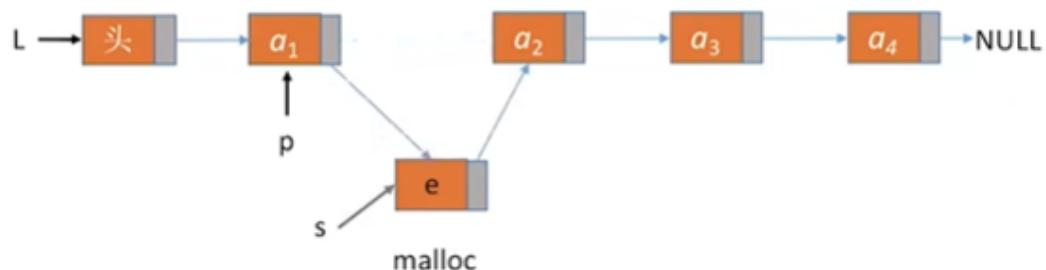
• 单链表的插入删除

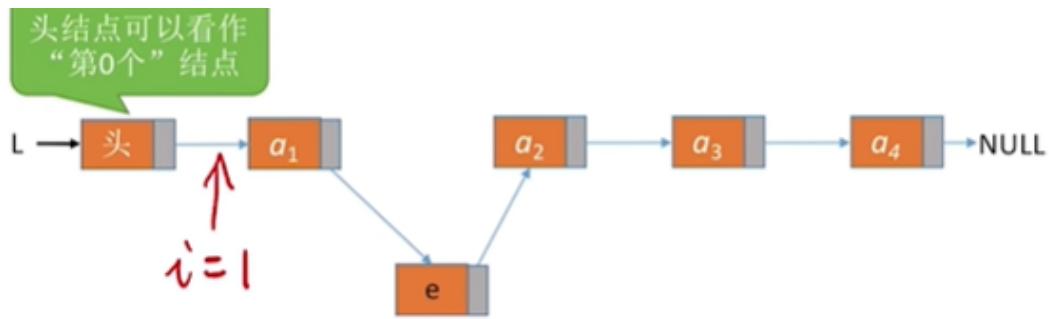
• 关于简化图的说明



• 按位序插入 (带头结点)

ListInsert(&L,i,e): 插入操作。在表L中的第i个位置上插入指定元素e。





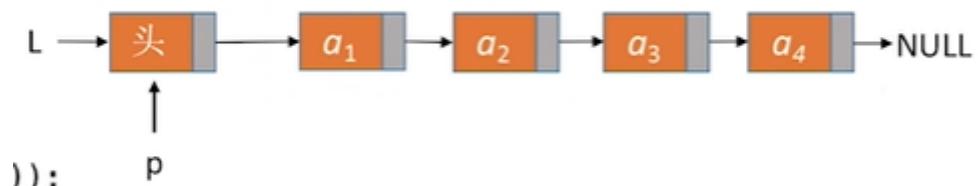
```

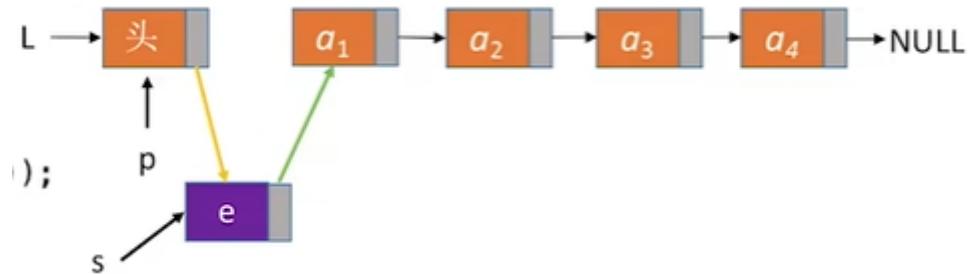
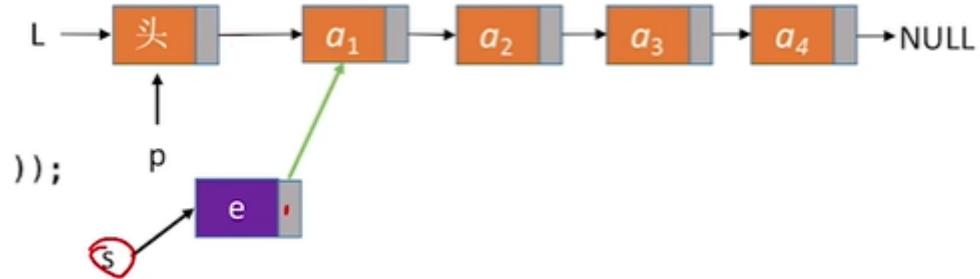
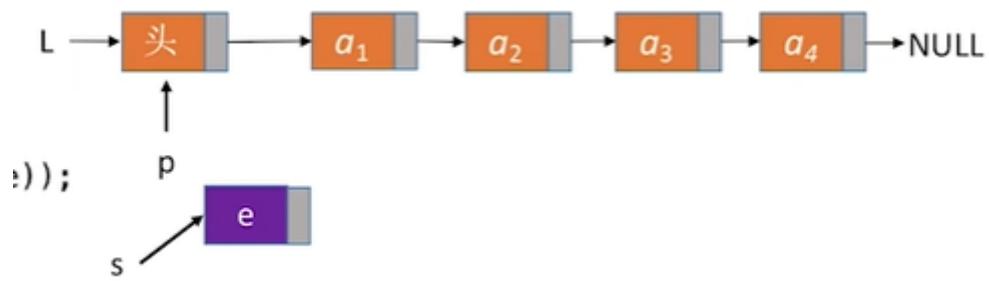
1 //在第i个位置插入元素e（带头结点）
2
3 typedef struct LNode
4 {
5     ELEMTYPE data;
6     struct LNode *next;
7 }LNode,*LinkList;
8
9 bool ListInsert(LinkList &L,int i,ELEMTYPE e)
10 {
11     if(i < 1)
12         return false;
13
14     LNode *p;      //指针p指向当前扫描到的结点
15     int j = 0;    //当前p指向的是第几个结点
16     p = L;        //L指向头节点，头结点是第0个结点（不存数据）
17
18     while(p != NULL && j < i - 1)      //循环找到i - 1个结点
19     {
20         p = p->next;
21         j++;
22     }
23
24     if(p == NULL)      //i值不合法
25         return false;
26
27     LNode *s = (LNode *)malloc(sizeof(LNode));
28
29     s->data = e;
30     s->next = p->next;
31     p->next = s;      //将结点s连到p之后
32     return true;      //插入成功
33 }
34 //30和31不可颠倒！

```

分析：

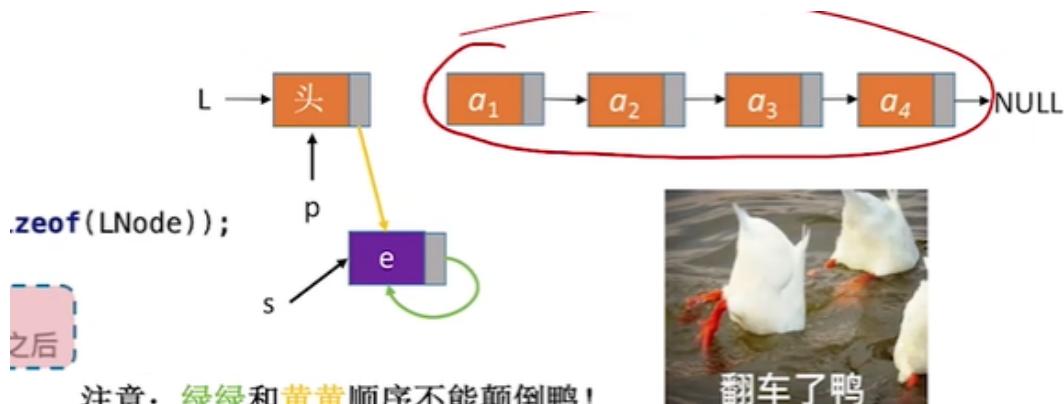
1. 如果 $i = 1$ (插在表头)





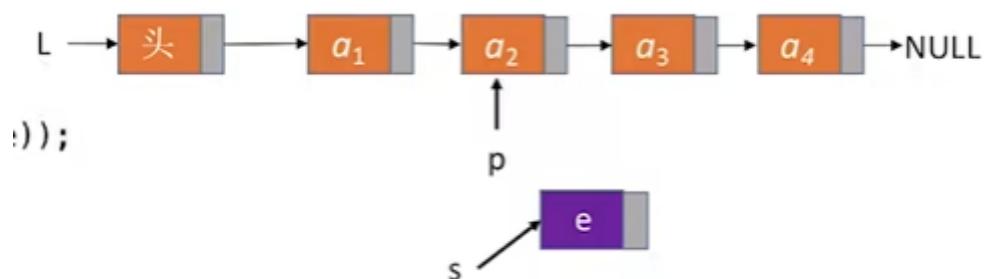
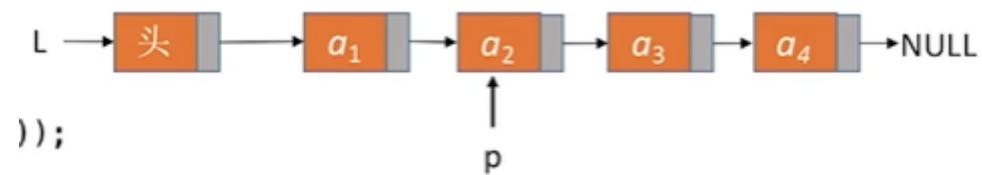
最好的时间复杂度: O(1)

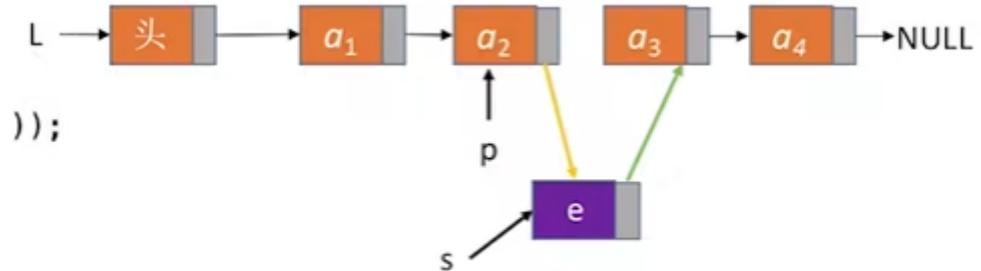
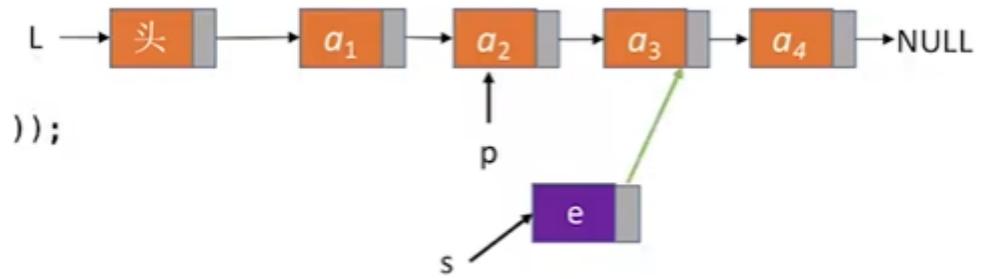
若30和31颠倒则会导致断链:



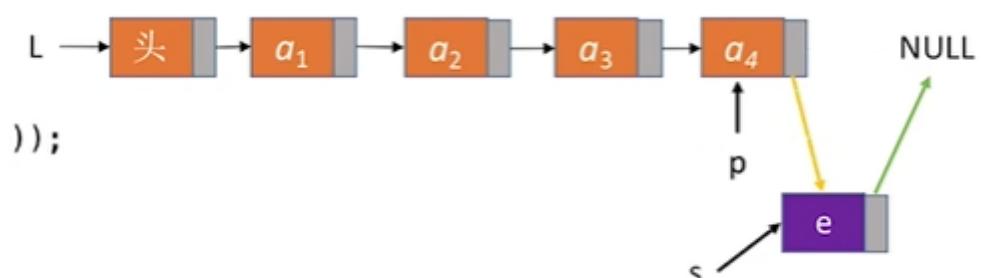
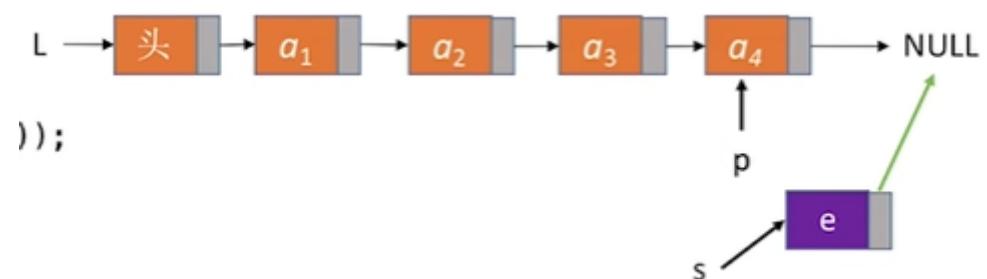
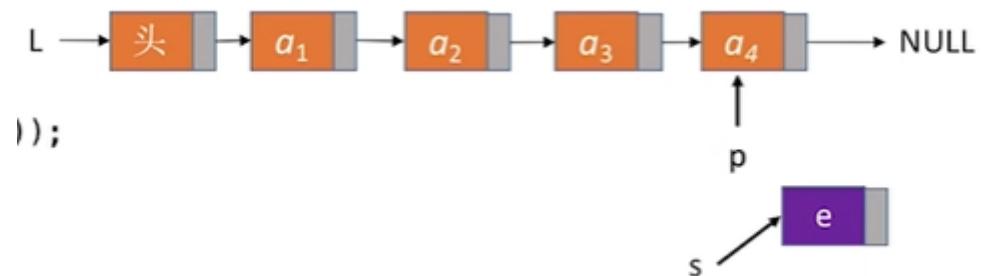
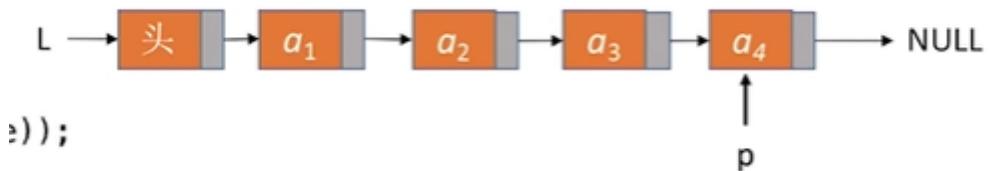
注意: 绿绿和黄黄顺序不能颠倒鸭!

2.如果i = 3 (插在表中)





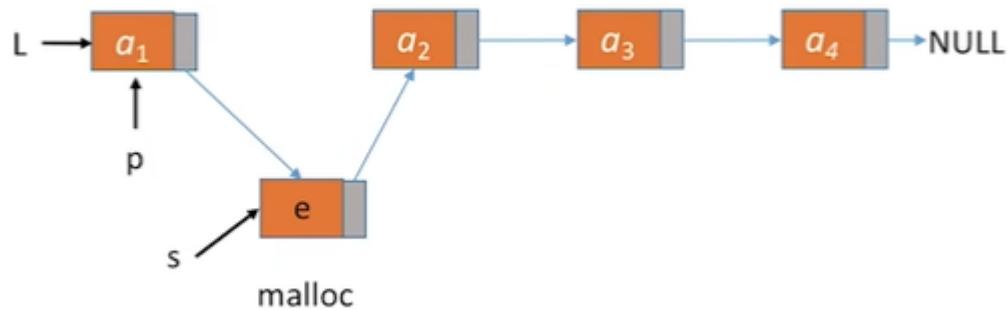
3.如果*i* = 5 (插在表尾)



最坏的时间复杂度: O(n)

- 按位序插入（不带头结点）

ListInsert(&L,i,e): 插入操作。在表L中的第i个位置上插入指定元素e。



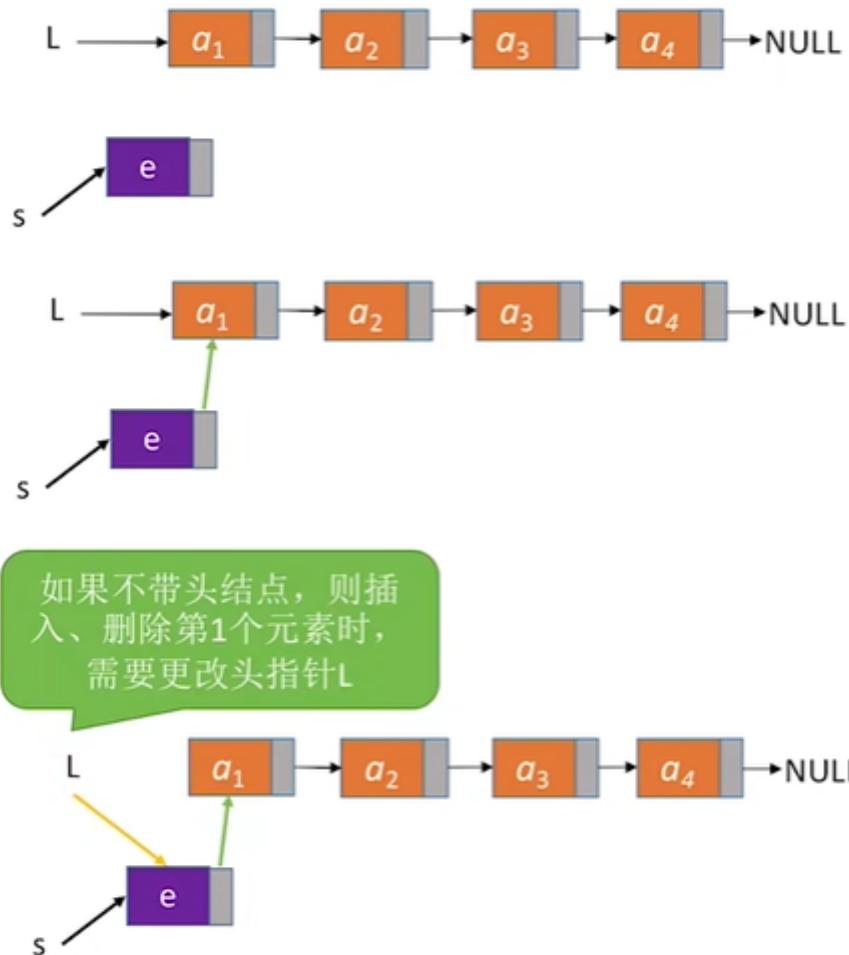
不存在“第0个”结点，因此*i* = 1时需要特殊处理

```

1  typedef struct LNode
2  {
3      ELEMTYPE data;
4      struct LNode *next;
5 }LNode,*LinkList;
6
7 bool ListInsert(LinkList &L,int i,ELEMTYPE e)
8 {
9     if(i < 1)
10        return false;
11     if(i == 1)    //插入第1个结点的操作与其他结点操作不同
12     {
13         LNode *s = (LNode *)malloc(sizeof(LNode));
14         s->data = e;
15         s->next = L;
16         L = s;           //头指针指向新结点
17         return true;
18     }
19
20     LNode *p;      //指针p指向当前扫描的结点
21     int j = 1;    //当前p指向的是第几个结点
22     p = L;        //p指向第1个结点（注意：不是头结点）
23
24     while(p != NULL && j < i - 1) //循环找到第i - 1个结点
25     {
26         p = p->next;
27         j++;
28     }
29
30     if(p == NULL)    //i值不合法
31         return false;
32
33     LNode *s = (LNode *)malloc(sizeof(LNode));
34     s->data = e;
35     s->next = p->next;
36     p->next = s;
37     return true;    //插入成功
38 }
```

分析：

1. 如果 $i = 1$ (插在表头)



```

bool ListInsert(LinkList &L, int i, ElemType e){
    if(i<1)
        return false;
    if(i==1){ //插入第1个结点的操作与其他结点操作不同
        LNode *s = (LNode *)malloc(sizeof(LNode));
        s->data = e;
        s->next=L;
        L=s; //头指针指向新结点
        return true;
    }
    LNode *p; //指针p指向当前扫描到的结点
    int j=1; //当前p指向的是第几个结点
    p = L; //p指向第1个结点(注意：不是头结点)
    while (p!=NULL && j<i-1) { //循环找到第 i-1 个结点
        p=p->next;
        j++;
    }
    if(p==NULL) //i值不合法
        return false;
    LNode *s = (LNode *)malloc(sizeof(LNode));
    s->data = e;
    s->next=p->next;
    p->next=s;
    return true; //插入成功
}

```

typedef struct LNode{
 ElemType data;
 struct LNode *next;
}LNode, *LinkList;

分析：
②如果 $i > 1 \dots$

L → a₁ → a₂ → a₃ → a₄ → NULL
p

后续逻辑和带头结点的一样

结论：不带头结点写代码更不方便，推荐用带头结点
注意：考试中带头、不带头都有可能考察，注意审题
王道考研/CSKAOYAN.COM

• 指定结点的后插操作

```

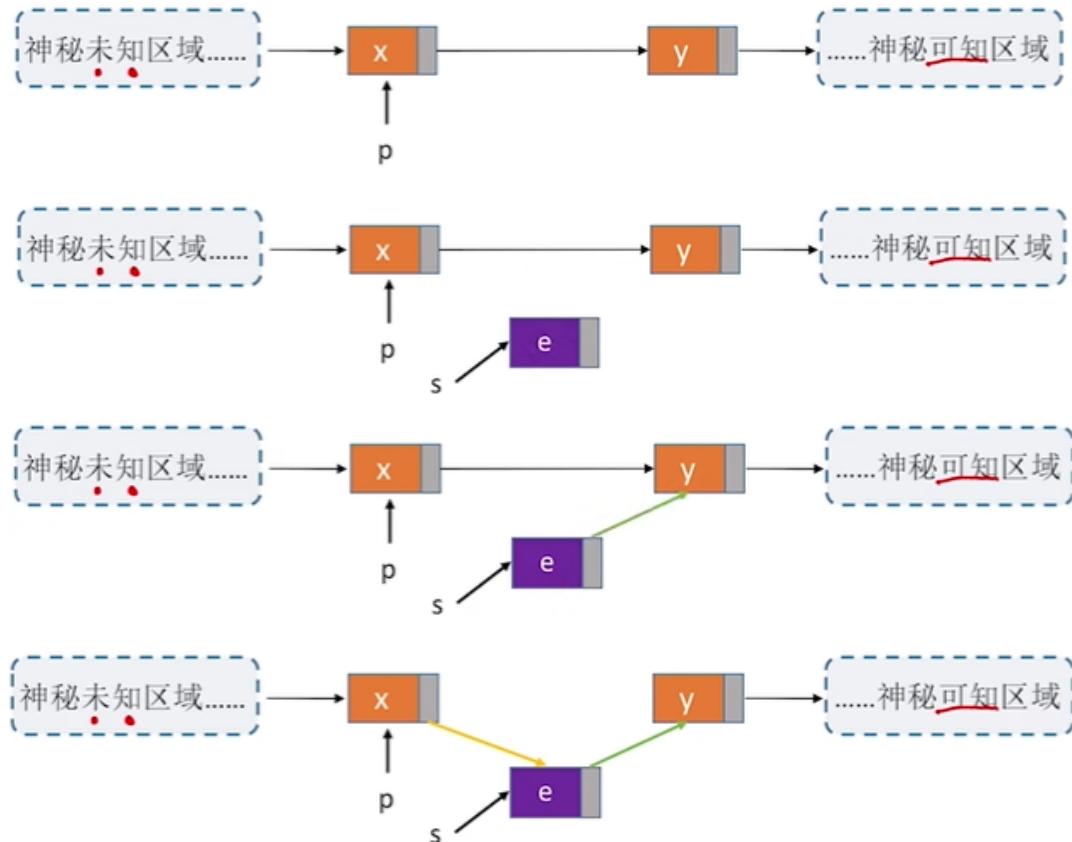
1 //后插操作：在p结点之后插入元素e
2 typedef struct LNode
3 {
4     ElemType data;
5     struct LNode *next;
6 }LNode,*LinkList;
7
8 bool InsertNextNode(LNode *p,ElemType e)

```

```

9  {
10     if(p == NULL)
11         return false;
12
13     LNode *s = (LNode *)malloc(sizeof(LNode));
14
15     if(s == NULL)          //内存分配失败
16         return false;
17
18     s->data = e;          //用结点s保存数据元素e
19     s->next = p->next;
20     p->next = s;          //将结点s连到p之后
21
22     return true;
23 }

```



- 时间复杂度: $O(1)$

```

1 //在第i个位置插入元素e（带头结点）可更改为一下代码
2
3 typedef struct LNode
4 {
5     ELEMTYPE data;
6     struct LNode *next;
7 }LNode,*LinkList;
8
9 bool ListInsert(LinkList &L,int i,ELEMTYPE e)
10 {
11     if(i < 1)
12         return false;
13
14     LNode *p;      //指针p指向当前扫描到的结点
15     int j = 0;      //当前p指向的是第几个结点
16     p = L;          //L指向头结点，头结点是第0个结点（不存数据）

```

```

17
18     while(p != NULL && j < i - 1)      //循环找到i - 1个结点
19     {
20         p = p->next;
21         j++;
22     }
23
24     return InsertNextNode(p, e);
25 }
26
27 bool InsertNextNode(LNode *p, ElemType e)
28 {
29     if(p == NULL)
30         return false;
31
32     LNode *s = (LNode *)malloc(sizeof(LNode));
33
34     if(s == NULL)          //内存分配失败
35         return false;
36
37     s->data = e;          //用结点s保存数据元素e
38     s->next = p->next;
39     p->next = s;          //将结点s连到p之后
40
41     return true;
42 }

```

• 指定节点的前插操作

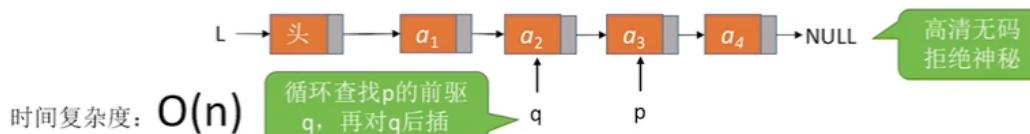
1.

```

1 //前程操作：在p结点前插入元素e
2 bool InsertPrioNode(LinkList L, LNode *p, ElemType e)

```

//前插操作：在p结点之前插入元素 e 传入头指针
bool InsertPriorNode (LinkList L, LNode *p, ElemType e)



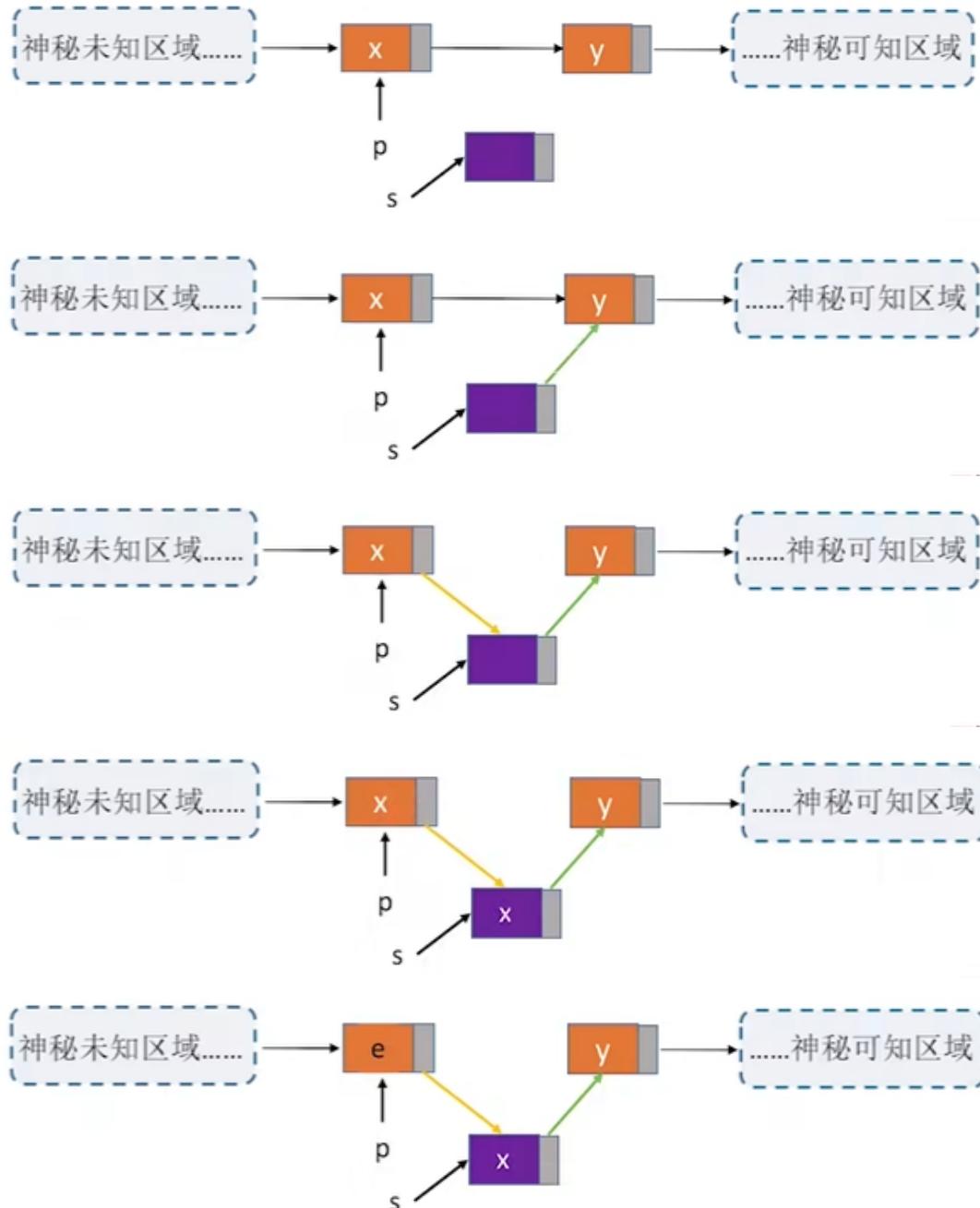
2.

```

1 //前程操作：在p结点前插入元素e
2 bool InsertPrioNode(LNode *p, ElemType e)
{
3
4     if(p == NULL)
5         return false;
6
7     LNode *s = (LNode *)malloc(sizeof(LNode));
8
9     if(s == NULL)          //内存分配失败
10        return false;
11
12     s->next = p->next;

```

```
13     p->next = s;           //新结点s连到p之后  
14     s->data = p->data;   //将p中元素复制到s中  
15     p->data = e;         //p中元素覆盖为e  
16     return true;  
17 }
```



时间复杂度为: O(1)

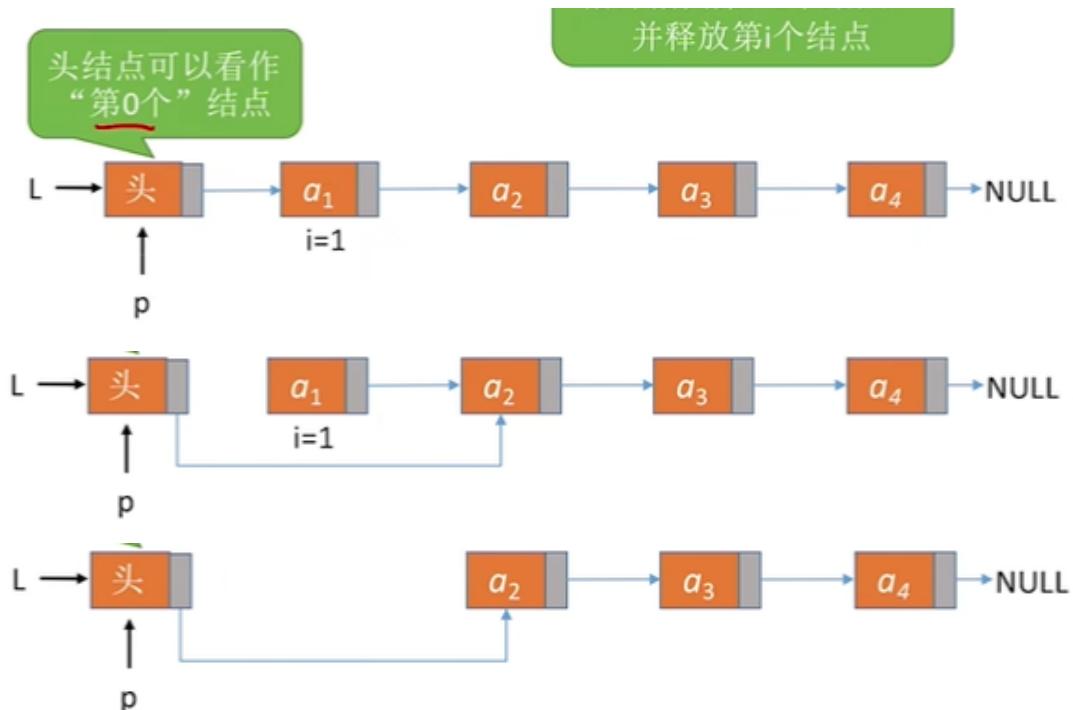
```

1 //前程操作：在p结点前插入元素s
2 bool InsertPriorNode(LNode *p, LNode *s)
3 {
4     if(p == NULL || s == NULL)
5         return false;
6
7     s->next = p->next;
8     p->next = s;           //s连到p之后
9     ElemType temp = p->data; //交换数据域部分
10    p->data = s->data;
11    s->data = temp;
12    return true;
13 }

```

- 按位序删除（带头结点）

ListDelete(&L,i,&e): 删除操作。删除表L中第i个位置的元素，并用e返回删除元素的值。



free()释放第二个结点

```

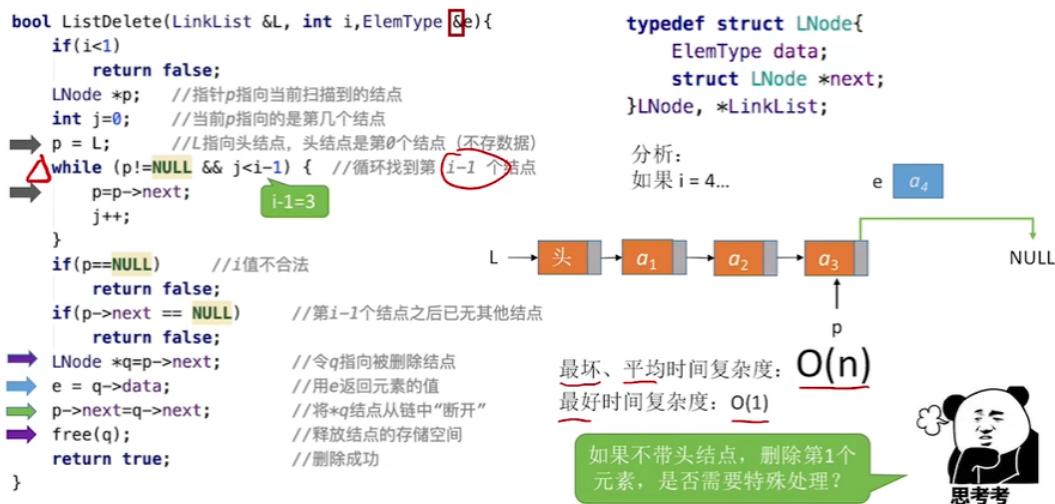
1 typedef struct LNode
2 {
3     ELEMTYPE data;
4     struct LNode *next;
5 } LNode,*LinkList;
6
7 bool ListDelete(LinkList &L,int i,ELEMTYPE &e)
8 {
9     if(i < 1)
10         return false;
11
12     LNode *p;           //指针p指向当前扫描到的结点
13     int j = 0;          //当前p指向的是第几个结点
14     p = L;             //L指向头结点，头结点是第0个结点（不存数据）
15

```

```

16     while(p != NULL && j < i - 1) //循环找到第i - 1个结点
17     {
18         p = p->next;
19         j++;
20     }
21
22     if(p == NULL)           //i值不合法
23         return false;
24
25     if(p->next == NULL)   //第i - 1个结点之后无其他结点
26         return false;
27
28     LNode *q = p->next;    //令q指向被删除结点
29     e = q->data;          //用e返回元素的值
30     p->next = q->next;    //将*q结点从链中“断开”
31     free(q);              //释放结点的存储空间
32     return true;
33 }
```

按位序删除（带头结点）



- 指定结点的删除

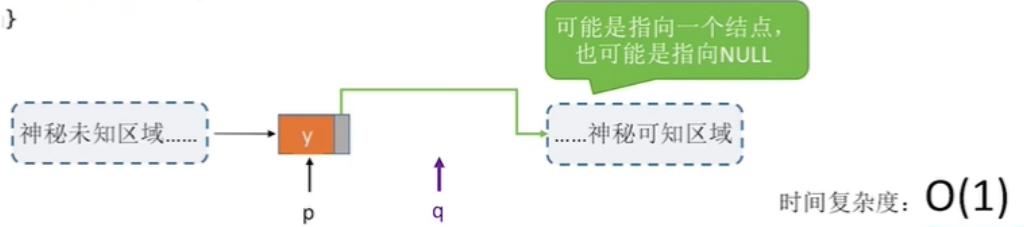
```

1 //删除指定结点p
2 bool DeleteNode(LNode *p)
3 {
4     if(p == NULL)
5         return false;
6
7     LNode *q = p->next;      //令q指向*p的后继结点
8     p->data = p->next->data; //和后继点交换数据域
9     p->next = q->next;      //将*q结点从链中“断开”
10    free(q);                //释放后继结点的存储空间
11    return true;
12 }
```

```

//删除指定结点 p
bool DeleteNode (LNode *p){
    if (p==NULL)
        return false;
    LNode *q=p->next;           //令q指向*p的后继结点
    p->data=p->next->data;     //和后继结点交换数据域
    p->next=q->next;           //将*q结点从链中“断开”
    free(q);                    //释放后继结点的存储空间
    return true;
}

```



```

//删除指定结点 p
bool DeleteNode (LNode *p){
    if (p==NULL)
        return false;
    LNode *q=p->next;           //令q指向*p的后继结点
    p->data=p->next->data;     //和后继结点交换数据域
    p->next=q->next;           //将*q结点从链中“断开”
    free(q);                    //释放后继结点的存储空间
    return true;
}

```

单链表的局限性：
无法逆向检索，有时候不太方便



• 知识总结

