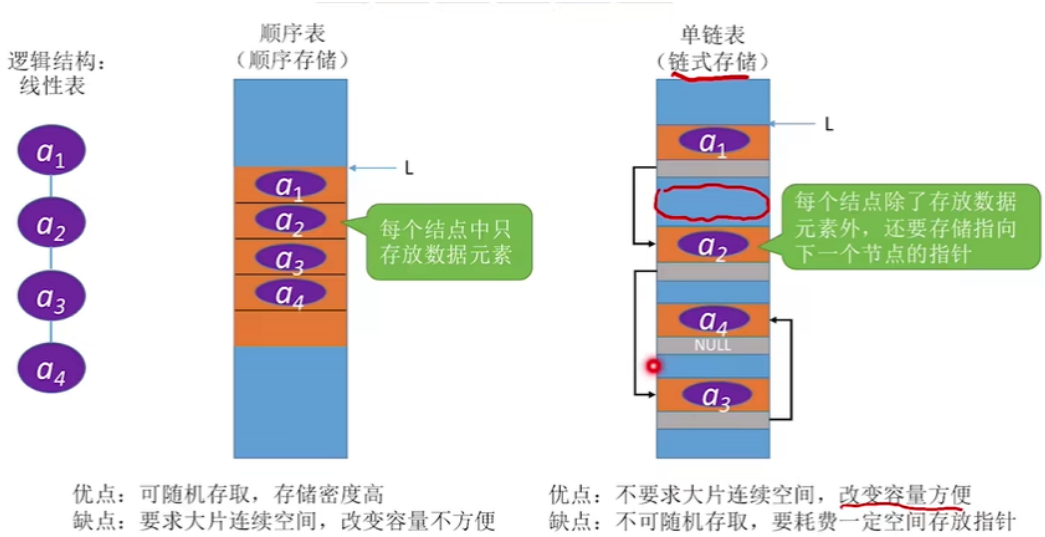
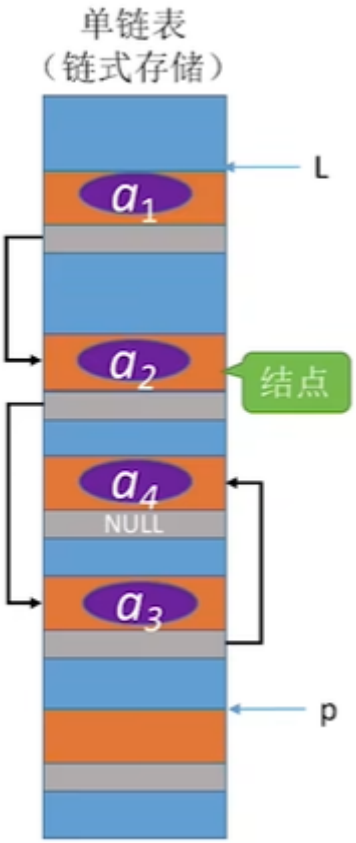
单链表

- ## 单链表的定义
- **什么是单链表**



- **用代码定义一个单链表**




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

```

```
//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
struct LNode * p = (struct LNode *)malloc(sizeof(struct LNode));

//用typedef重命名
typedef struct LNode LNode;
LNode * p = (LNode *)malloc(sizeof(LNode));

//更简洁的方式举例
typedef struct LNode
{
    ElemType data;
    struct LNode *next;
}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）
//上面代码等同于下面的代码
struct LNode
{
    ElemType data;
    struct LNode *next;
};
typedef struct LNode LNode;
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

```
LNode * L;    //声明一个指向单链表第一个结点的指针
//或者下面的表示方法也一样
LinkList L;   //声明一个指向单链表第一个结点的指针

//e.q:
LNode * GetElem(LinkList L,int i)
{
    int j = 1;
    LNode *p = L->next;

    if(i == 0)
        return L;
    if(i < 1)
        return NULL;

    while(p != NULL && j < i)
    {
        p = p->next;
        j++;
    }
    return p;
}
```

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

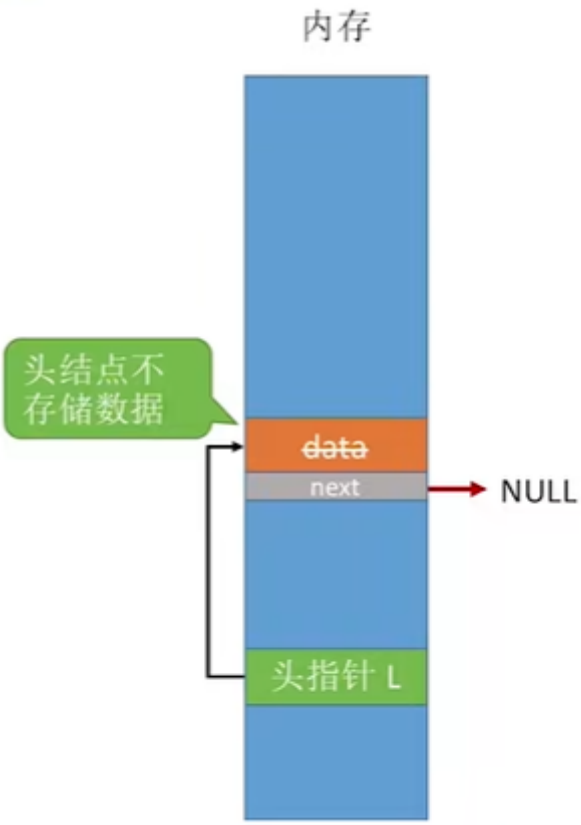
- **不带头结点的单链表**

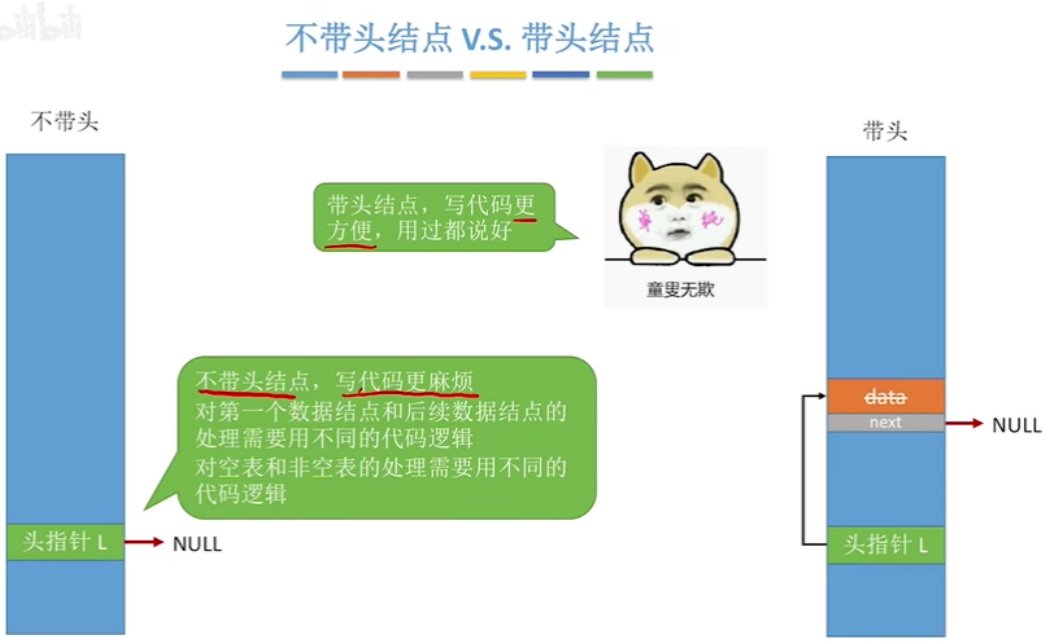
```
typedef struct LNode
{
    ElemType data;
    struct LNode *next;
}LNode,*LinkList;

//初始化一个空的单链表
bool InitList(LinkList &L)
{
    L = NULL;     //空表，暂时还没有任何结点，防止脏数据
    return true;
}

//判断单链表是否为空
bool Empty(LinkList L)
{
    if (L == NULL)
        return true;
    else
        return false;
}
//或者
bool Empty(LinkList L)     //这个布尔函数的值本身就是TRUE或FLASE
{
    return (L == NULL);
}

void test()
{
    LinkList L;     //注意，此处并没有创建一个结点
    //初始化一个空表
    InitList(L);
    //...后续代码...
}
```

- **带头结点的单链表**

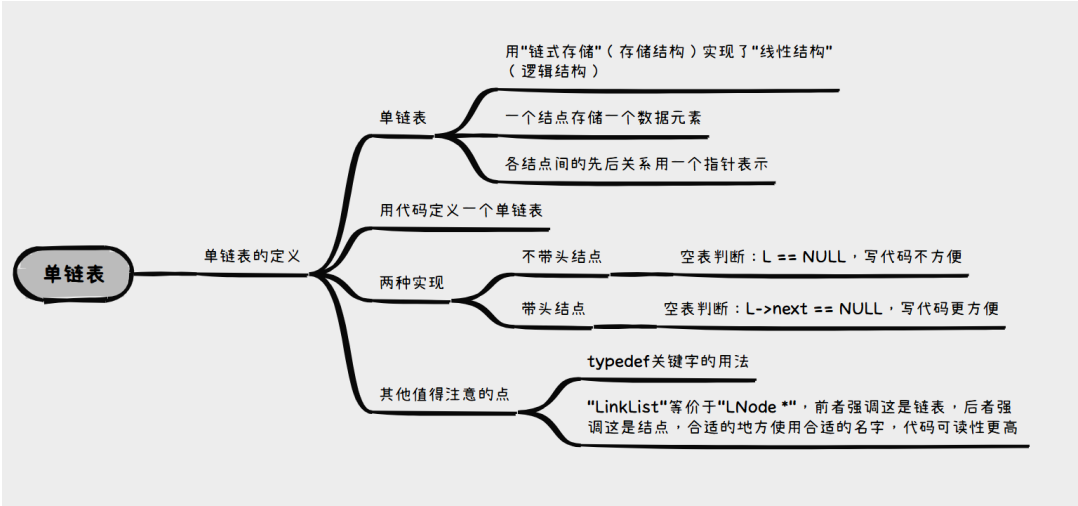
```
typedef struct LNode
{
    ElemType data;
    struct LNode *next;
}LNode,*LinkList;

//初始化一个单链表（带头结点）
bool InitList(LinkList &L)
{
    L = (LNode *)malloc(sizeof(LNode));    //分配一个头结点

    if(L == NULL)
        return false;

    L->next = NULL;    //头结点之后暂时还没有结点
    return true;
}

//判断单链表是否为空（带头结点）
bool Empty(LinkList L)
{
    return (L->next == NULL);
}

void test()
{
    LinkList L;    //注意，此处并没有创建一个结点
    //初始化一个空表
    InitList(L);
    //...后续代码...
```

```
31 }
```


内存
头结点不
存储数据
data
next
NULL
头指针 L

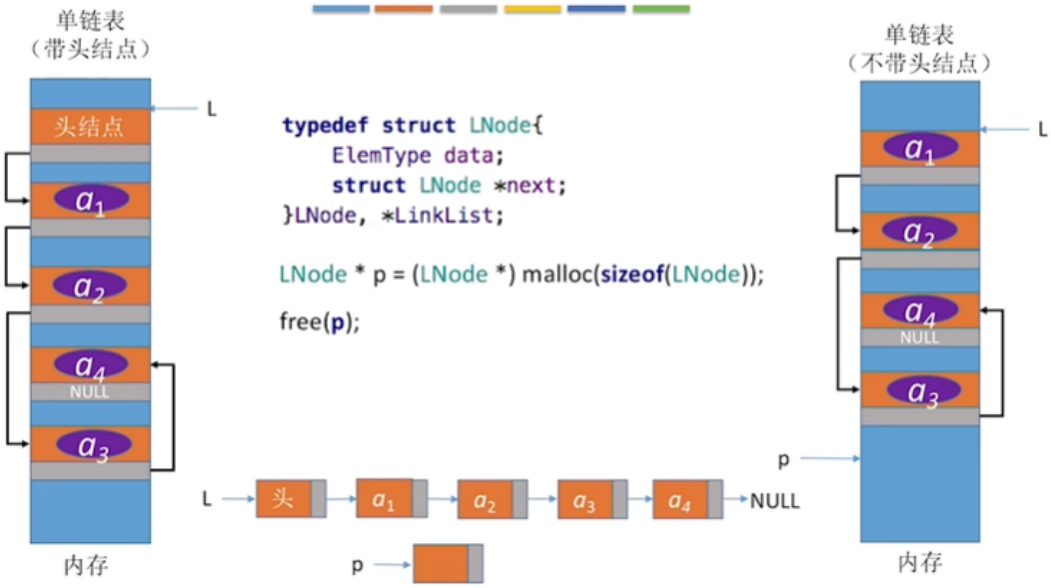

不带头结点 V.S. 带头结点
不带头
带头结点，写代码更
方便，用过都说好
童叟无欺
带头
不带头结点，写代码更麻烦
对第一个数据结点和后续数据结点的
处理需要用不同的代码逻辑
对空表和非空表的处理需要用不同的
代码逻辑
头指针 L
NULL
data
next
NULL
头指针 L

- **知识总结**



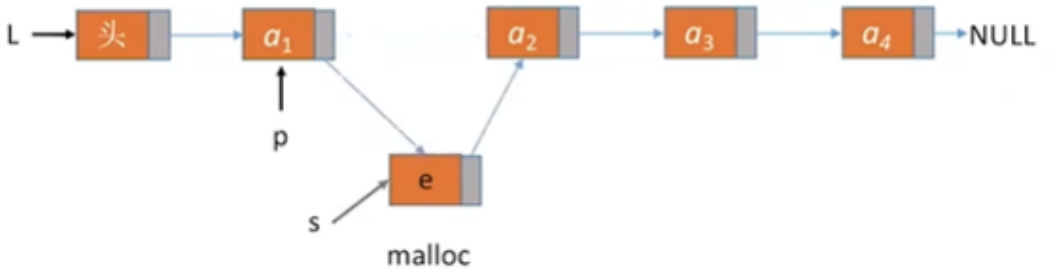
- # 单链表的插入删除

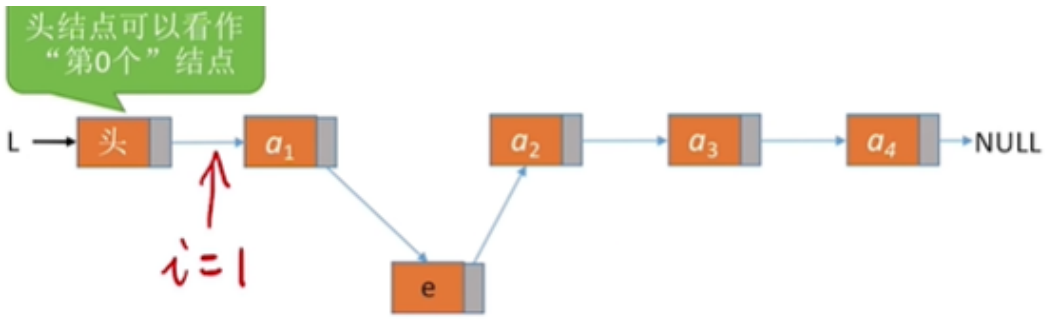
- **关于简化图的说明**



- **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。

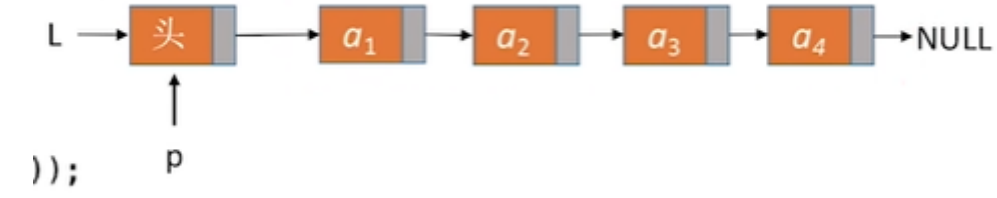


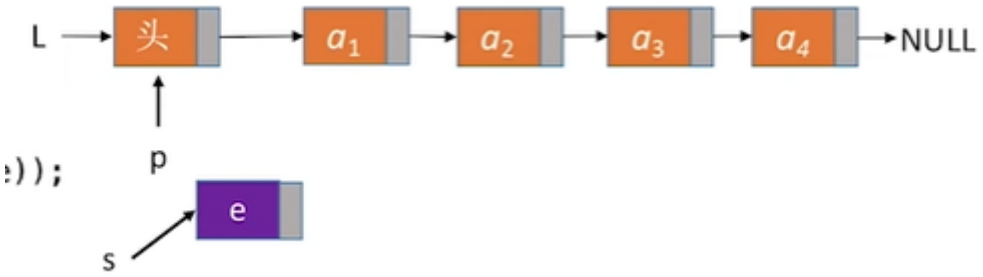


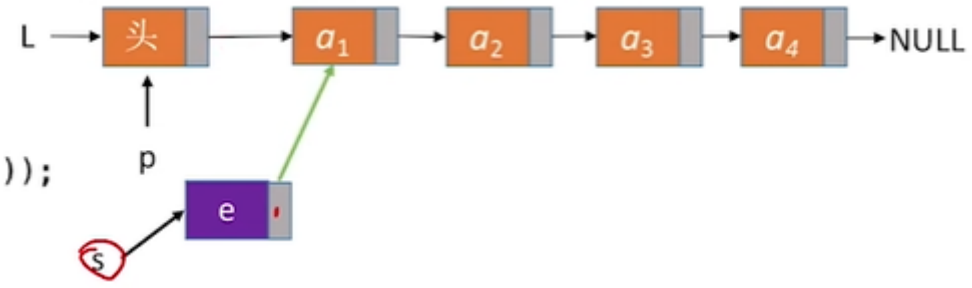

```
//在第i个位置插入元素e（带头结点）

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

bool ListInsert(LinkList &L,int i,ElemType e)
{
    if(i < 1)
        return false;

    LNode *p;     //指针p指向当前扫描到的结点
    int j = 0;    //当前p指向的是第几个结点
    p = L;        //L指向头节点，头结点是第0个结点（不存数据）

    while(p != NULL && j < i - 1)     //循环找到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;     //将结点s连到p之后
    return true;     //插入成功
}
//30和31不可颠倒!
```

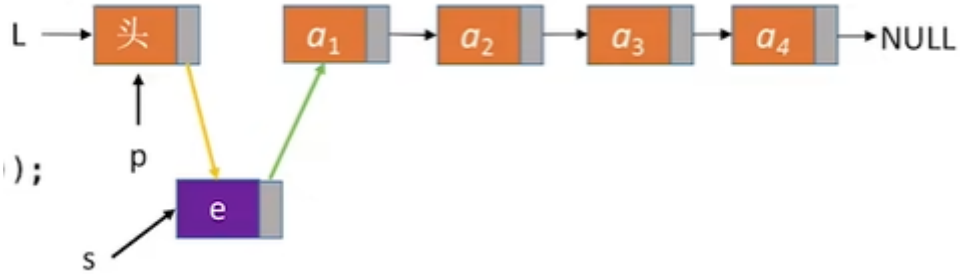
分析：

1.如果i = 1（插在表头）



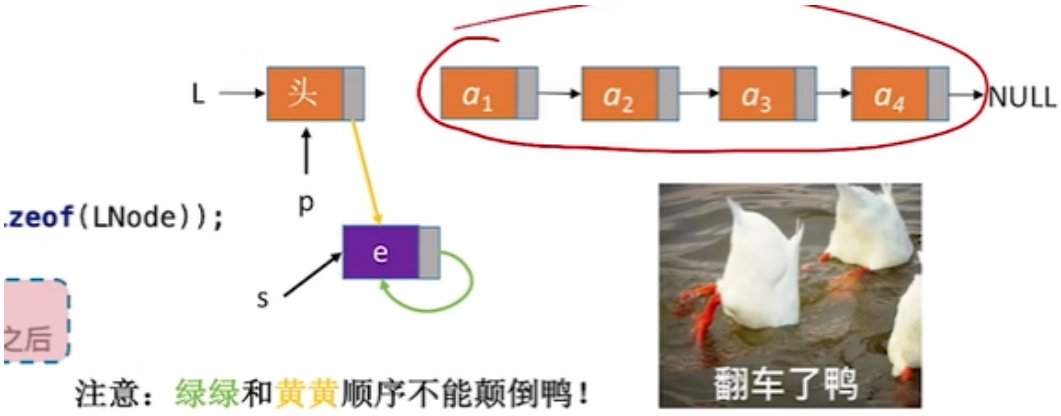




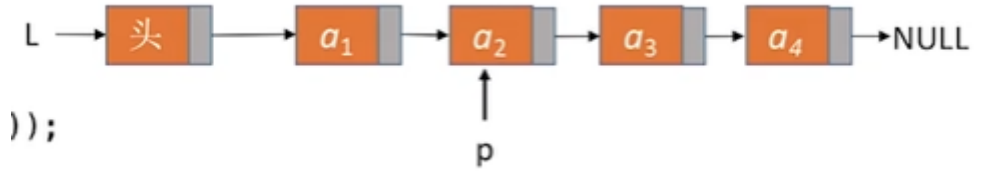


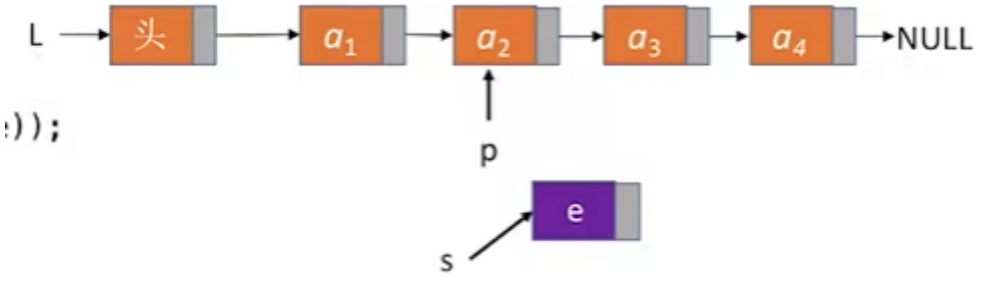
最好的时间复杂度：O(1)

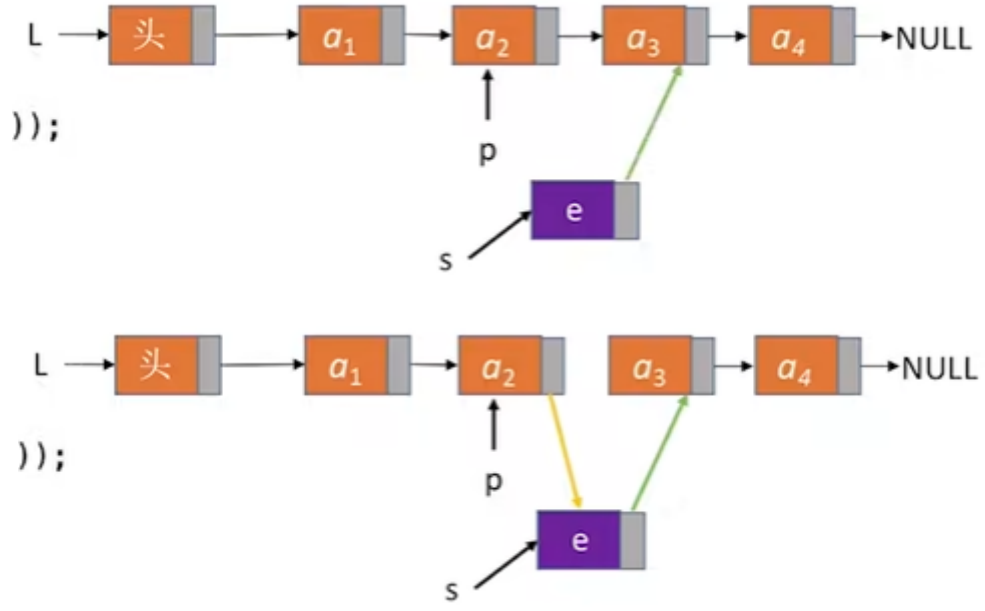
若30和31颠倒则会导致断链：



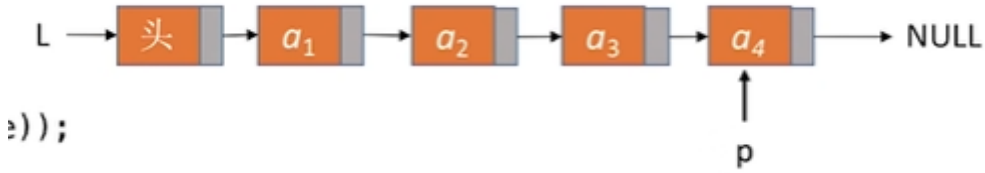
2.如果i = 3（插在表中）

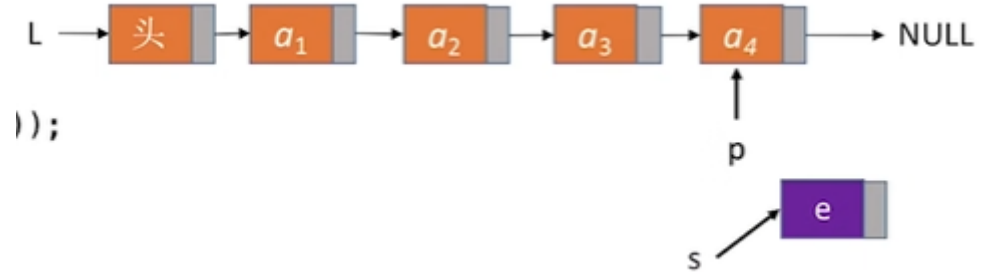


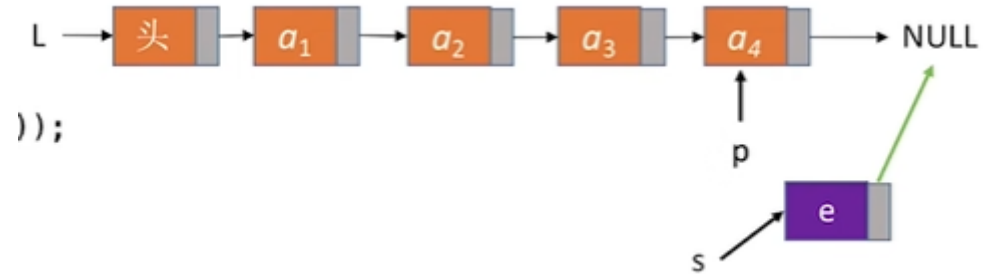


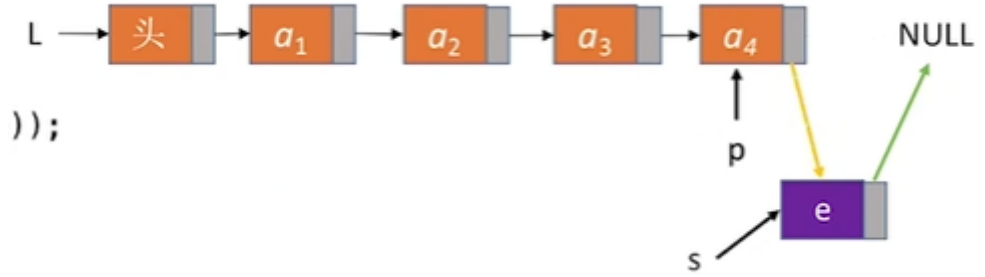


3.如果i = 5（插在表尾）





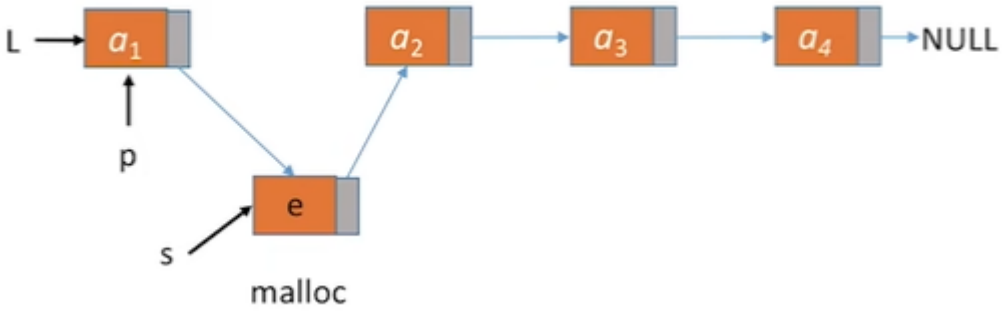




最坏的时间复杂度：O(n)

- **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。

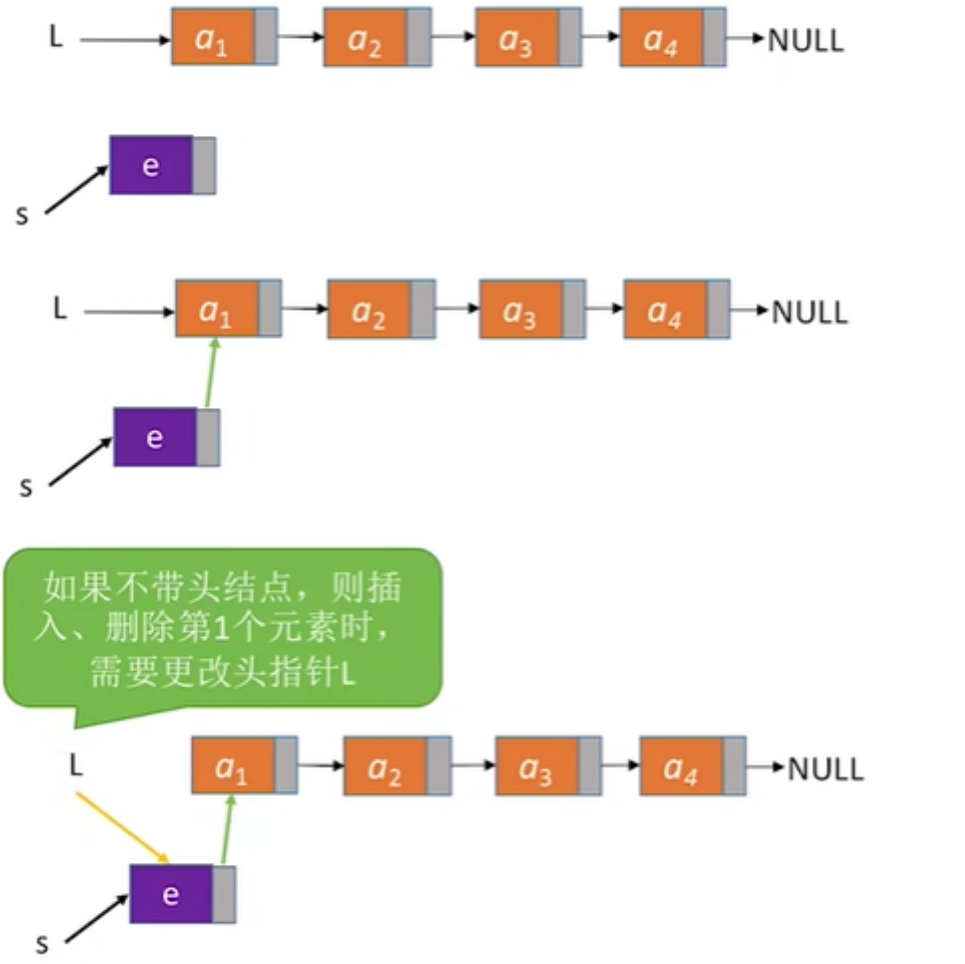


不存在“第0个”结点，因此i = 1时需要特殊处理

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

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
        reutnr false;

    LNode *s = (LNode *)malloc(sizeof(LNode));
    s->data = e;
    s->next = p->next;
    p->next = s;
    return true;    //插入成功
}
```

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...

后续逻辑和带头结点的一样

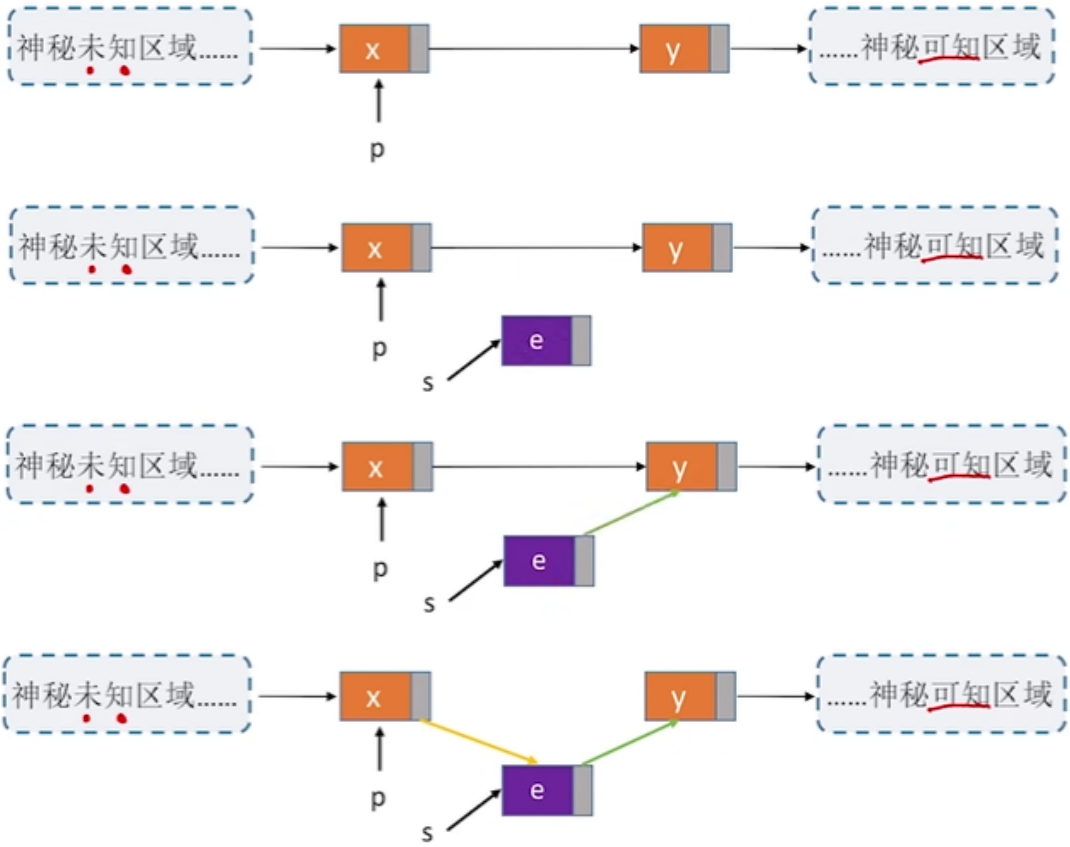
结论：不带头结点写代码更不方便，推荐用带头结点
注意：考试中带头、不带头都有可能考察，注意审题

王道考研/CSKAOYAN.COM

- **指定结点的后插操作**

```
//后插操作：在p结点之后插入元素e
typedef struct LNode
{
    ElemType data;
    struct LNode *next;
}LNode,*LinkList;

bool InsertNextNode(LNode *p,ElemType e)
```

```
{
    if(p == NULL)
        return false;

    LNode *s = (LNode *)malloc(sizeof(LNode));

    if(s == NULL)          //内存分配失败
        return false;

    s->data = e;           //用结点s保存数据元素e
    s->next = p->next;
    p->next = s;           //将结点s连到p之后
    return true;
}
```



- 时间复杂度：O(1)

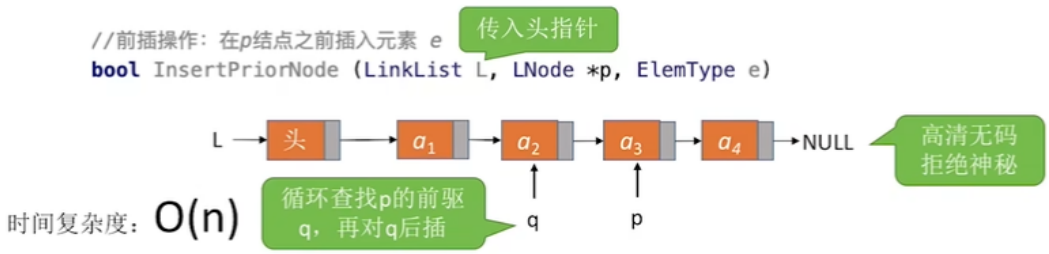
```
//在第i个位置插入元素e（带头结点）可更改为一下代码

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

bool ListInsert(LinkList &L,int i,ElemType e)
{
    if(i < 1)
        return false;

    LNode *p;     //指针p指向当前扫描到的结点
    int j = 0;    //当前p指向的是第几个结点
    p = L;        //L指向头节点，头结点是第0个结点（不存数据）
```

```
    while(p != NULL && j < i - 1)    //循环找到i - 1个结点
    {
        p = p->next;
        j++;
    }

   return InsertNextNode(p,e);
}

bool InsertNextNode(LNode *p,ElemType e)
{
    if(p == NULL)
        return false;

    LNode *s = (LNode *)malloc(sizeof(LNode));

    if(s == NULL)          //内存分配失败
        return false;

    s->data = e;           //用结点s保存数据元素e
    s->next = p->next;
    p->next = s;           //将结点s连到p之后
    return true;
}
```

- **指定节点的前插操作**

1.

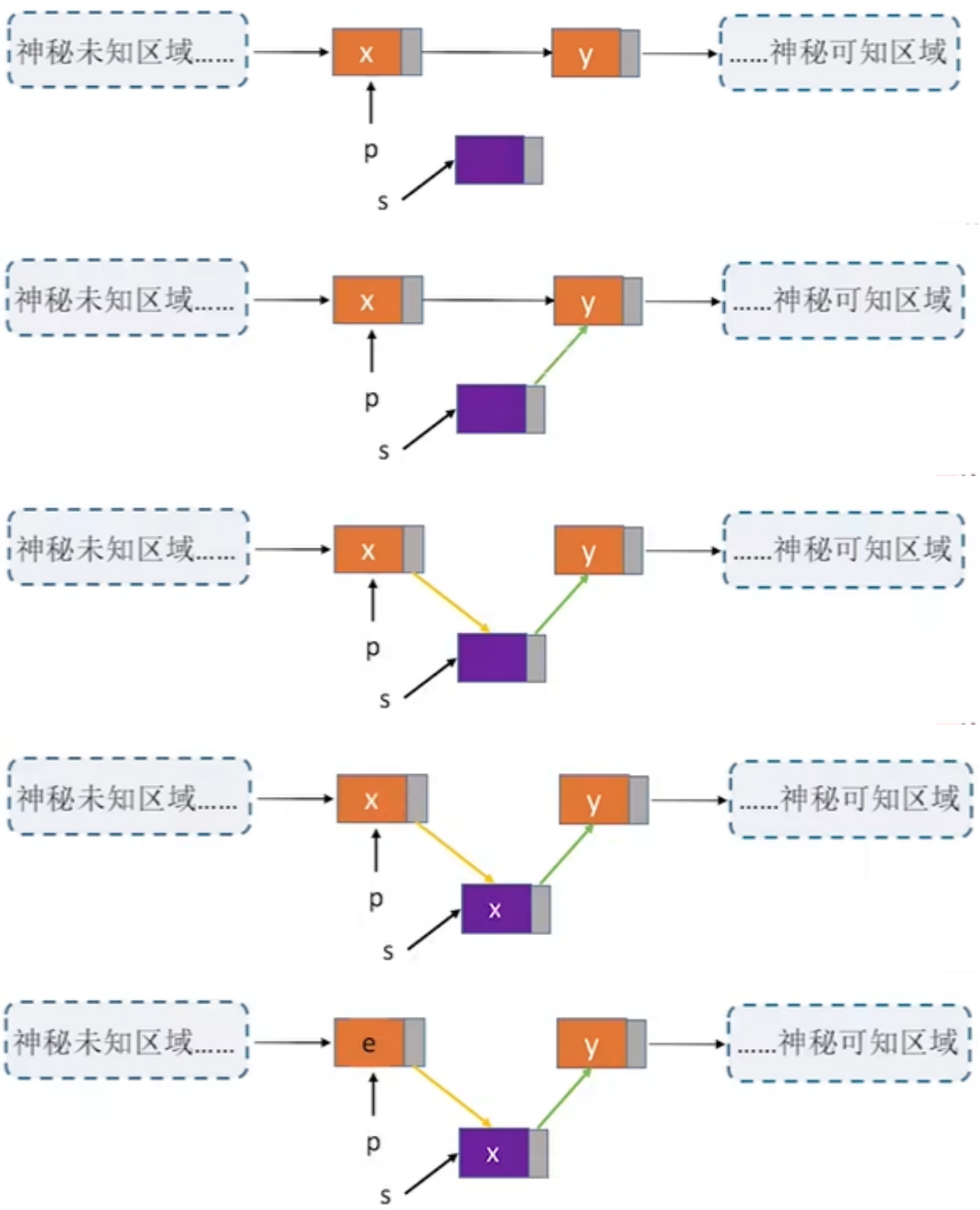
```
//前程操作：在p结点前插入元素e
bool InsertPrioNode(LinkList L,LNode *p,ElemType e)
```



2.

```
//前程操作：在p结点前插入元素e
bool InsertPrioNode(LNode *p,ElemType e)
{
    if(p == NULL)
        return false;

    LNode *s = (LNode *)malloc(sizeof(LNode));

    if(s == NULL)    //内存分配失败
        return false;

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

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

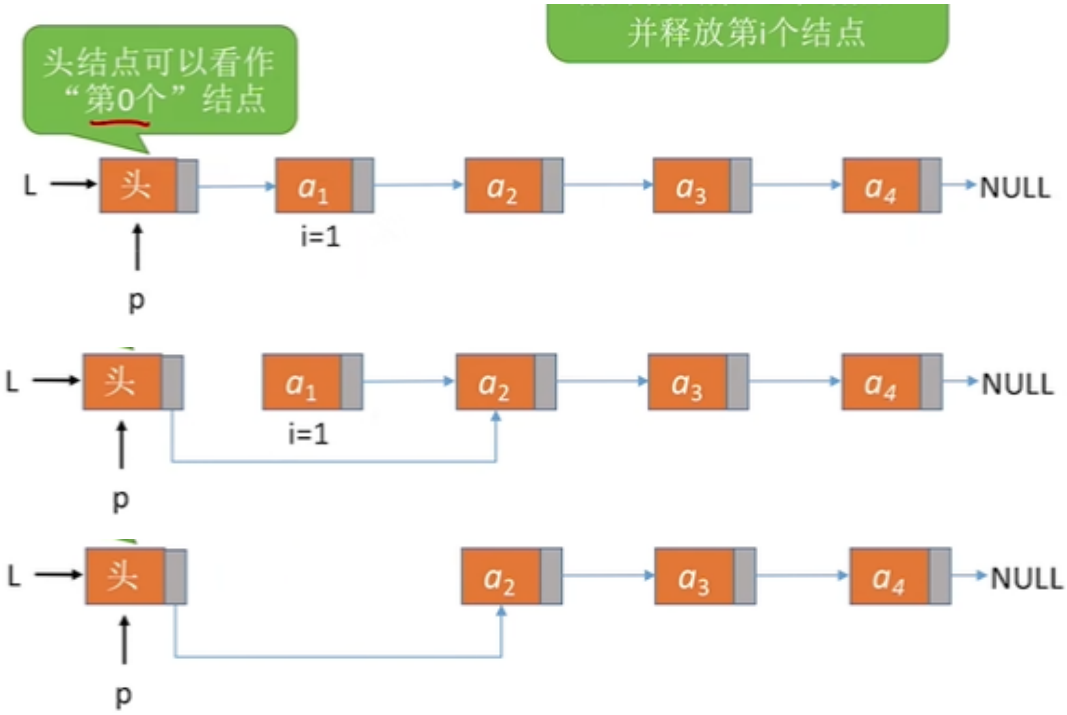


时间复杂度为：O(1)

```
//前程操作：在p结点前插入元素s
bool InsertPriorNode(LNode *p,LNode *s)
{
    if(p == NULL || s == NULL)
        return false;

    s->next = p->next;
    p->next = s;                //s连到p之后
    ElemType temp = p->data;  //交换数据域部分
    p->data = s->data;
    s->data = temp;
    return true;
}
```

- **按位序删除（带头结点）**

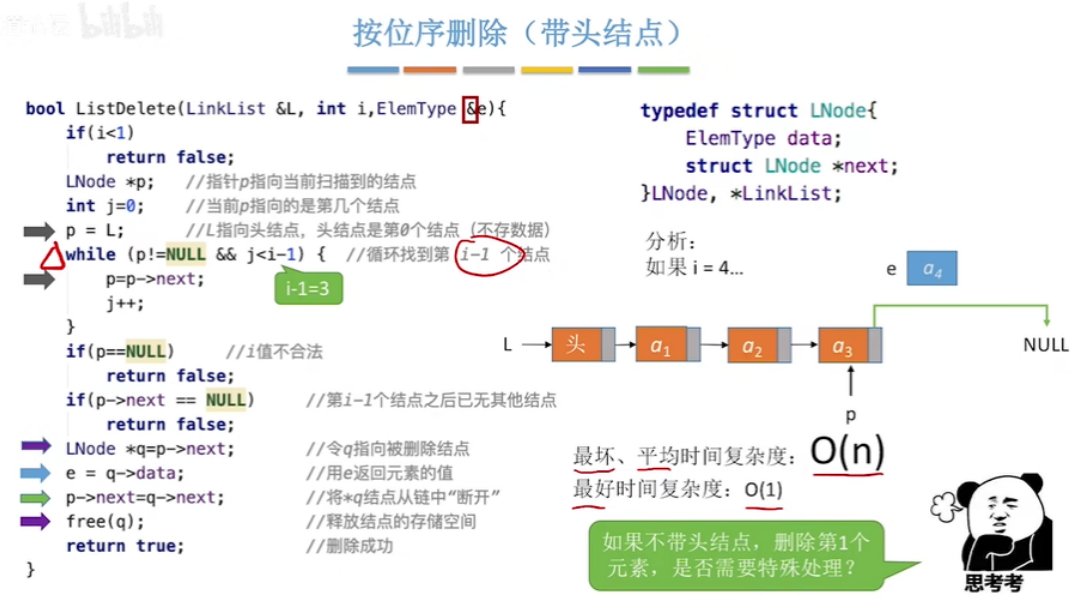
ListDelete(&L,i,&e)：删除操作。删除表L中第i个位置的元素，并用e返回删除元素的值。



free()释放第二个结点

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

bool ListDelete(LinkList &L,int i,ElemType &e)
{
    if(i < 1)
        return false;

    LNode *p;          //指针p指向当前扫描到的结点
    int j = 0;         //当前p指向的是第几个结点
    p = L;             //L指向头结点，头结点是第0个结点（不存数据）

```

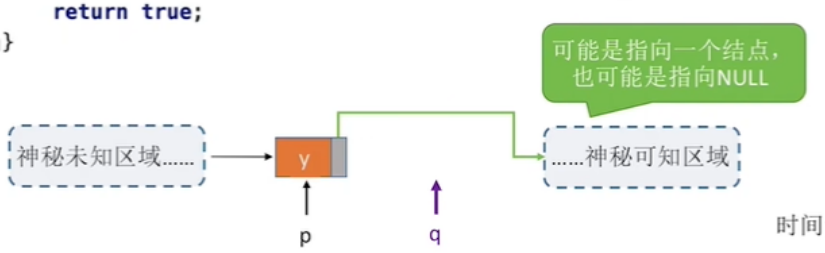
```
    while(p != NULL && j < i - 1)  //循环找到第i - 1个结点
    {
        p = p->next;
        j++;
    }

    if(p == NULL)          //i值不合法
        return false;

    if(p->next == NULL)  //第i - 1个结点之后无其他结点
        return false;

    LNode *q = p->next;  //令q指向被删除结点
    e = q->data;          //用e返回元素的值
    p->next = q->next;    //将*q结点从链中“断开”
    free(q);              //释放结点的存储空间
    return true;
}
```



- **指定结点的删除**

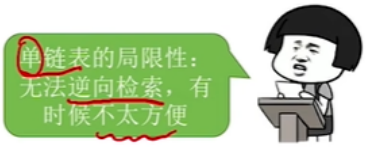
```
//删除指定结点p
bool DeleteNode(LNode *p)
{
    if(p == NULL)
        return false;

    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;
}
```

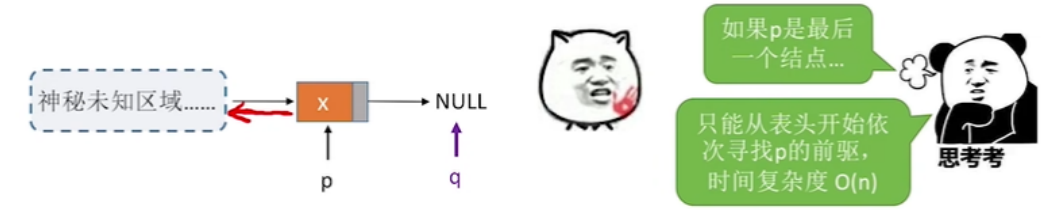
```
//删除指定结点 p
bool DeleteNode (LNode *p){
    if (p==NULL)
        return false;
    LNode *q=p->next;          //令q指向*p的后继结点
    p->data=p->next->data;     //和后继结点交换数据域
    p->next=q->next;           //将*q结点从链中“断开”
    free(q);                   //释放后继结点的存储空间
    return true;
}
```



时间复杂度：O(1)

```
//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;
}
```





- **知识总结**