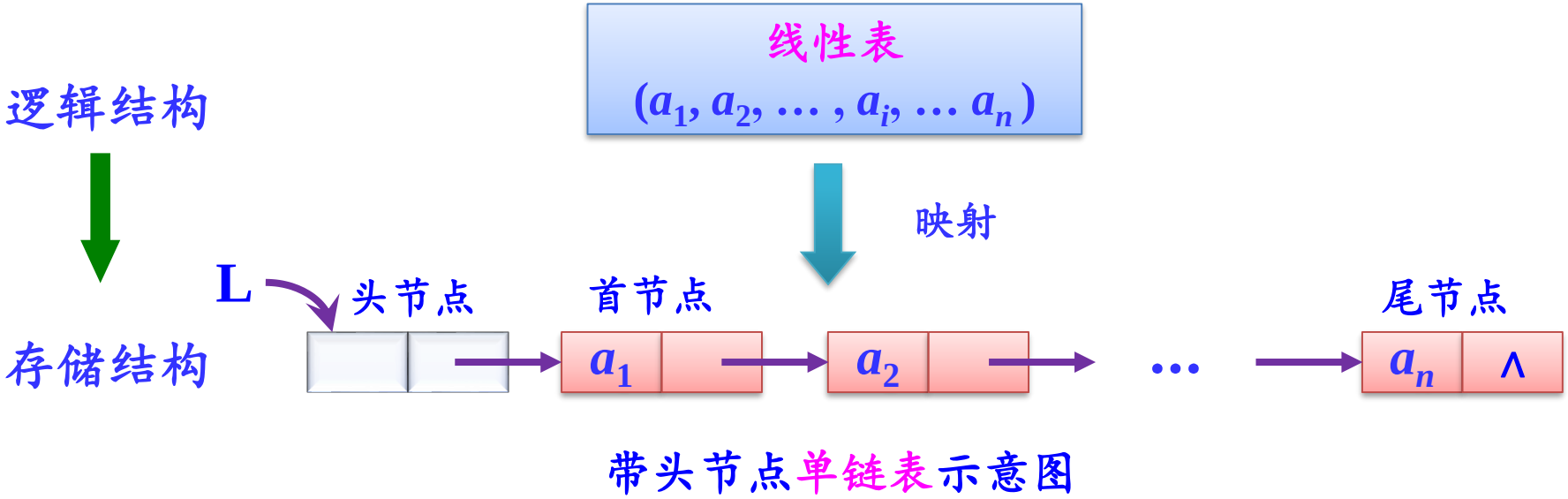
2.3 线性表的链式存储结构

2.3.1 线性表的链式存储—链表

线性表中每个节点有唯一的前趋节点和前趋节点。

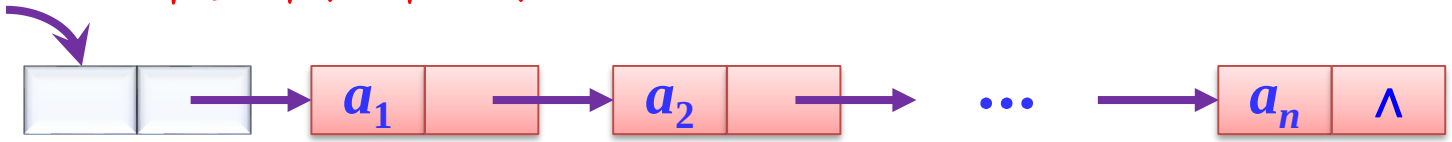
设计链式存储结构时，每个逻辑节点存储单独存储，为了表示逻辑关系，增加指针域。

- 每个物理节点增加一个指向后继节点的指针域 ⇨ 单链表。
- 每个物理节点增加一个指向后继节点的指针域和一个指向前趋节点的指针域 ⇨双链表。



带头节点单链表示意图

带头节点单链表



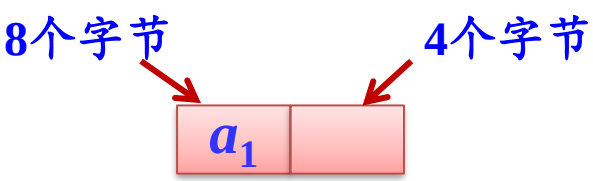
单链表增加一个头节点的优点如下：

- 第一个节点的操作和表中其他节点的操作相一致，无需进行特殊处理；
- 无论链表是否为空，都有一个头节点，因此空表和非空表的处理也就统一了。

存储密度是指节点数据本身所占的存储量和整个节点结构中所占的存储量之比，即：

$$\text{存储密度}=\frac{\text{节点数据本身占用的空间}}{\text{节点占用的空间总量}}$$

例如



存储密度 = 8/12 = 67%

一般地，存储密度越大，存储空间的利用率就越高。显然，顺序表的存储密度为1（100%），而链表的存储密度小于1。

思考题：

线性表的顺序存储结构和链式存储结构的差异？

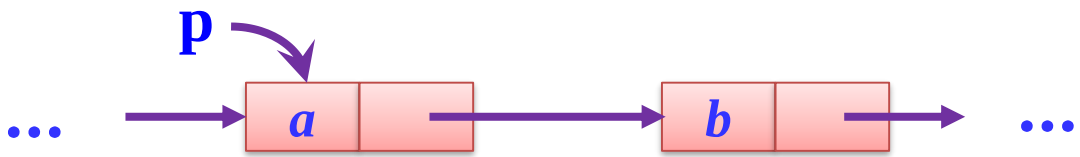
2.3.2 单链表

单链表中节点类型LinkList的定义如下：

```
typedef struct LNode          //定义单链表节点类型
{     ElemType data;
      struct LNode *next;     //指向后继节点
} LinkList;
```

单链表的特点

当访问过一个节点后，只能接着访问它的后继节点，而无法访问它的前趋节点。



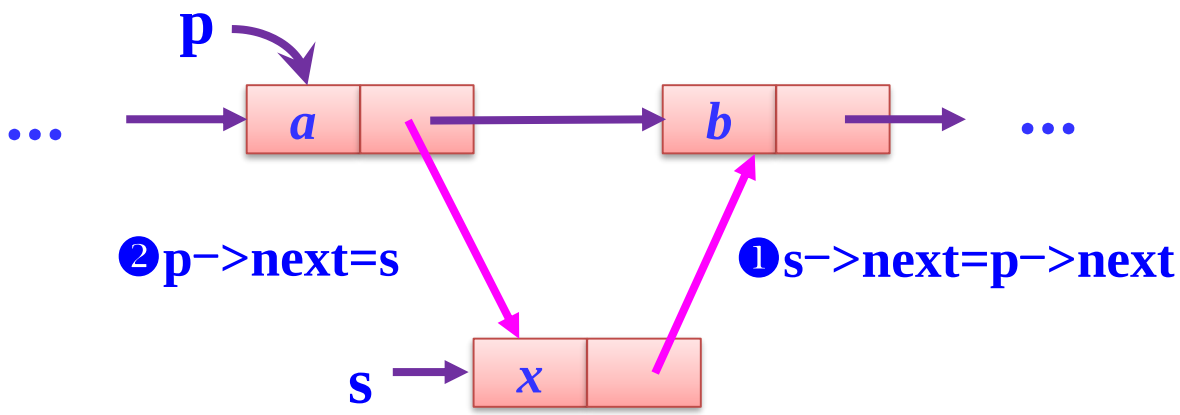
1、插入节点和删除节点操作

（1）插入节点

插入操作：将值为x的新节点*s插入到*p节点之后。

特点：只需修改相关节点的指针域，不需要移动节点。

单链表插入节点演示



插入操作语句描述如下：

❶ s->next = p->next;

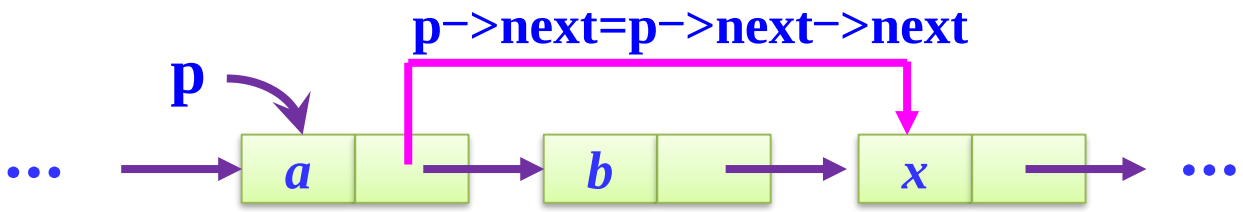
❷ p->next = s;

（2）删除节点

删除操作：删除*p节点之后的一个节点。

特点：只需修改相关节点的指针域，不需要移动节点。

单链表删除节点演示



删除操作语句描述如下：

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

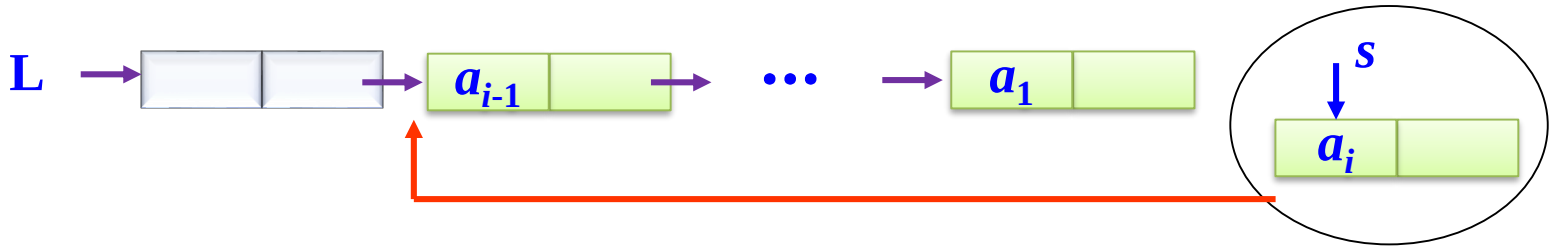
2、建立单链表

先考虑如何整体建立单链表。

建立单链表的常用方法有两种。

（1）头插法建表

- 从一个空表开始，创建一个头节点。
- 依次读取字符数组a中的元素，生成新节点
- 将新节点插入到当前链表的表头上，直到结束为止。

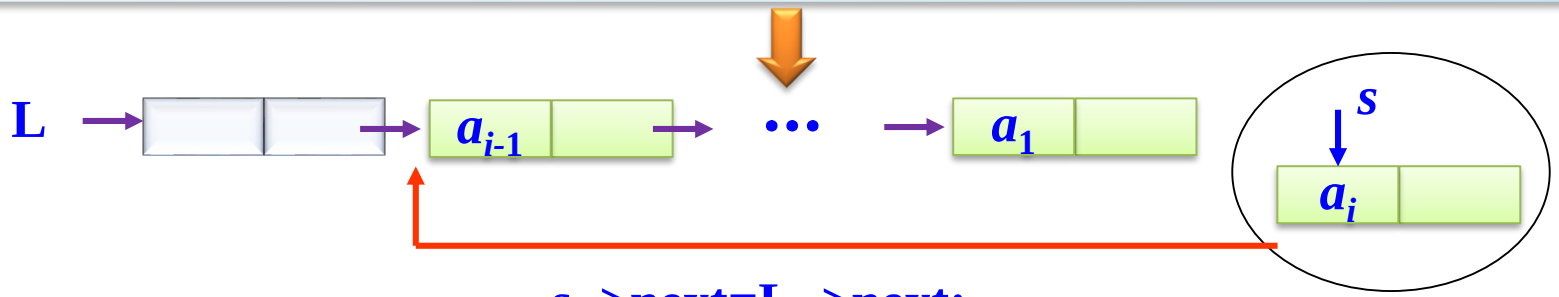


注意：链表的节点顺序与逻辑次序相反。

头插法建表算法如下：

```
void CreateListF(LinkList *&L,ElemType a[],int n)
{   LinkList *s;
    int i;
    L=(LinkList *)malloc(sizeof(LinkList));
    L->next=NULL;                    //创建头节点,其next域置为NULL
```

```
    for (i=0;i<n;i++)                    //循环建立数据节点
    {     s=(LinkList *)malloc(sizeof(LinkList));
          s->data=a[i];                  //创建数据节点*s
          s->next=L->next;               //将*s插在原开始节点之前,头节点之后
          L->next=s;
    }
}
```

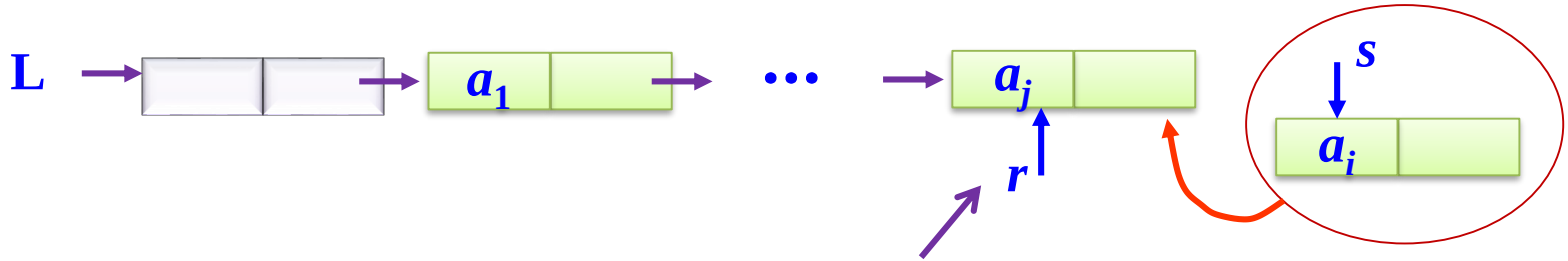


s->next=L->next;

L->next=s;

（2）尾插法建表

- 从一个空表开始，创建一个头节点。
- 依次读取字符数组a中的元素，生成新节点
- 将新节点插入到当前链表的表尾上，直到结束为止。

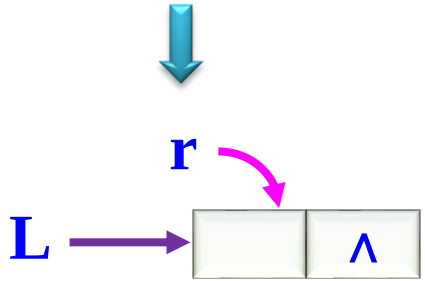


增加一个尾指针r，使其始终指向当前链表的尾节点

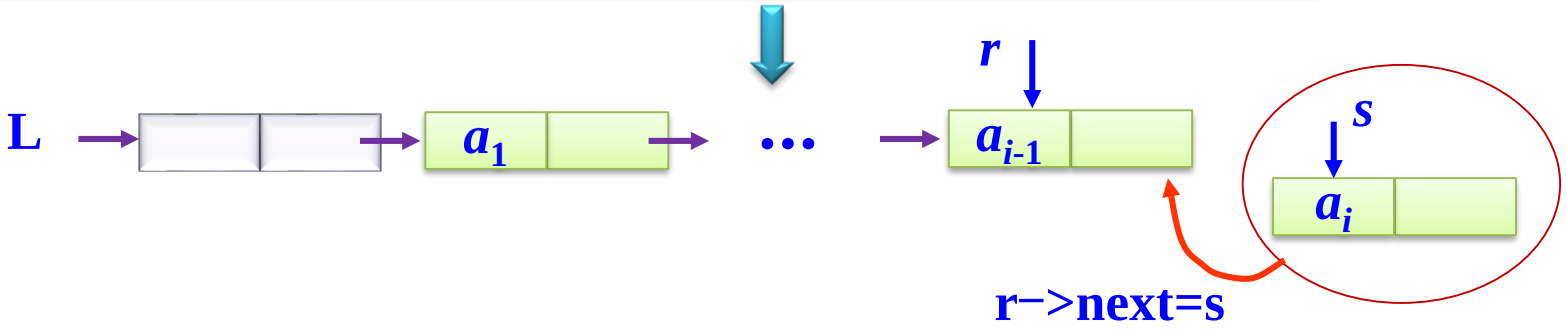
注意：链表的节点顺序与逻辑次序相同。

尾插法建表算法如下：

```
void CreateListR(LinkList *&L,ElemType a[],int n)
{    LinkList *s,*r;
     int i;
     L=(LinkList *)malloc(sizeof(LinkList)); //创建头节点
     r=L;                                    //r始终指向尾节点,开始时指向头节点
```



```
    for (i=0;i<n;i++)            //循环建立数据节点
    {     s=(LinkList *)malloc(sizeof(LinkList));
          s->data=a[i];          //创建数据节点*s
          r->next=s;             //将*s插入*r之后
          r=s;
    }
    r->next=NULL;                //尾节点next域置为NULL
}
```

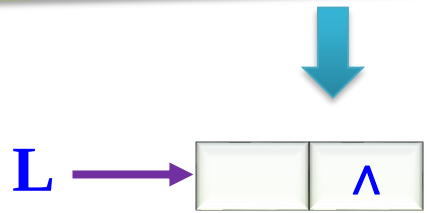


3、线性表基本运算在单链表上的实现

（1）初始化线性表InitList(L)

该运算建立一个空的单链表，即创建一个头节点。

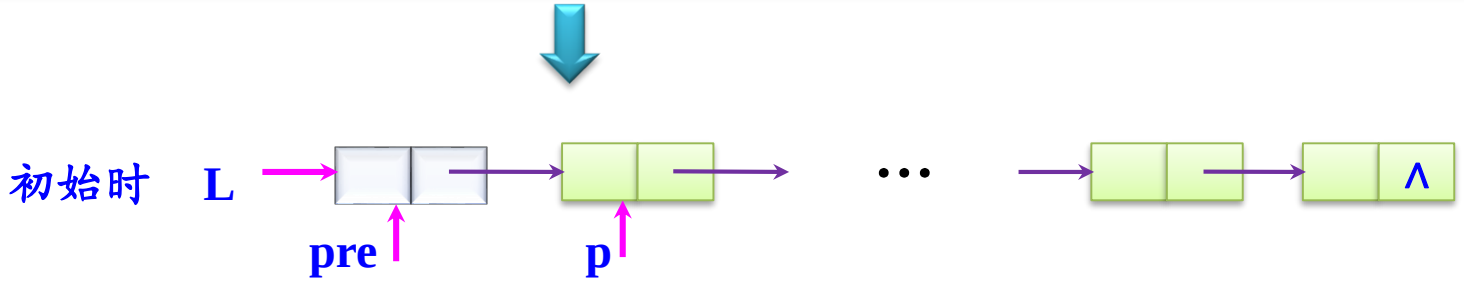
```
void InitList(LinkList *&L)
{
    L=(LinkList *)malloc(sizeof(LinkList));   //创建头节点
    L->next=NULL;
}
```



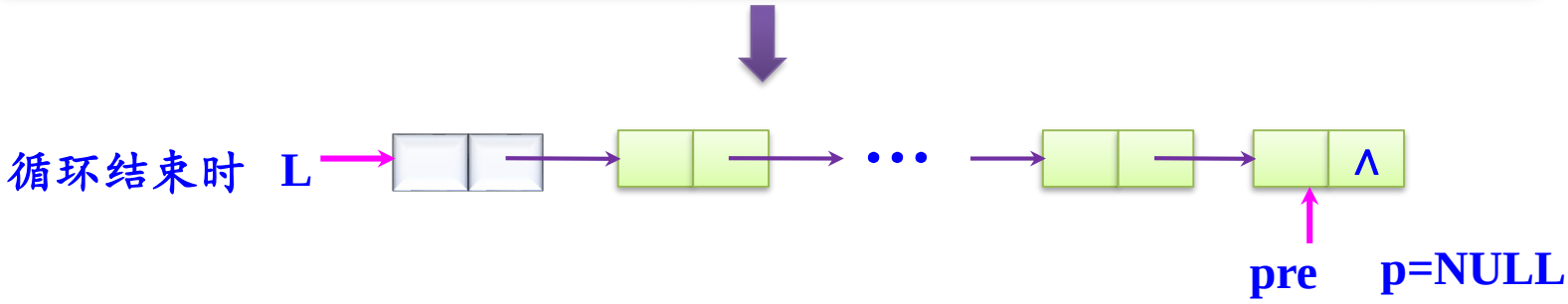
（2）销毁线性表DestroyList(L)

释放单链表L占用的内存空间。即逐一释放全部节点的空间。

```
void DestroyList(LinkList *&L)
{
    LinkList *pre=L, *p=L->next;    //pre指向*p的前趋节点
```



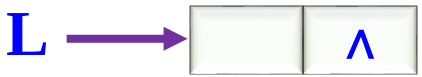
```
    while (p!=NULL)        //扫描单链表L
    {     free(pre);       //释放*pre节点
          pre=p;           //pre、p同步后移一个节点
          p=pre->next;
    }
    free(pre);    //循环结束时,p为NULL,pre指向尾节点,释放它
}
```



（3）判线性表是否为空表ListEmpty(L)

若单链表L没有数据节点，则返回真，否则返回假。

```
bool ListEmpty(LinkList *L)
{
    return(L->next==NULL);
}
```

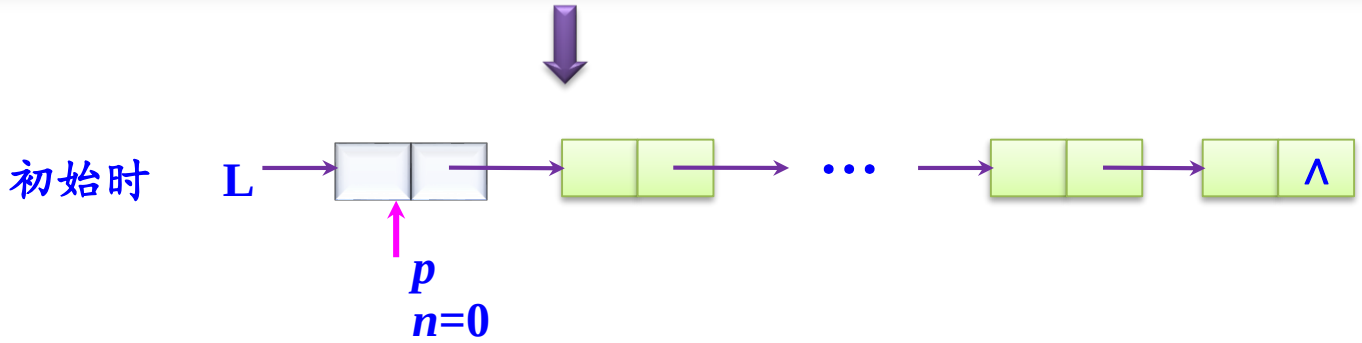


空表的情况

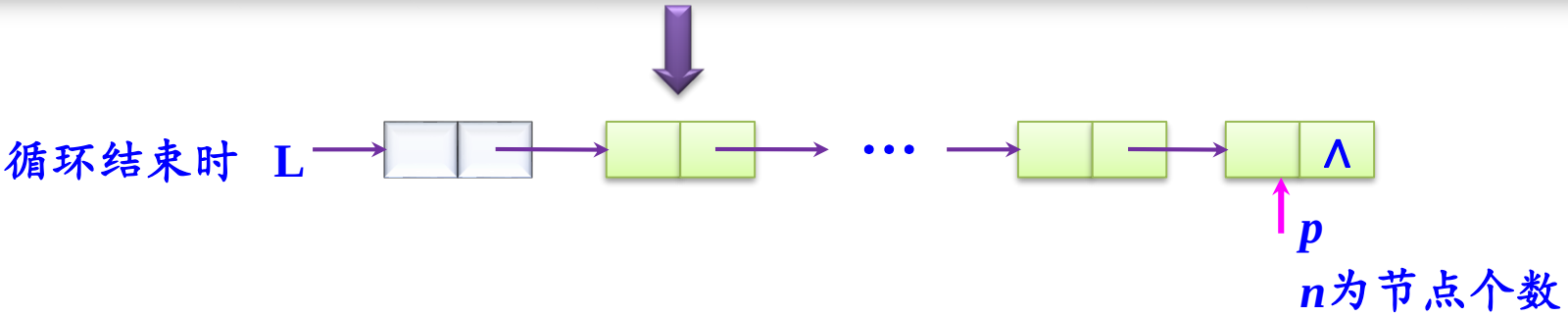
（4）求线性表的长度ListLength(L)

返回单链表L中数据节点的个数。

```
int ListLength(LinkList *L)
{
    int n=0;
    LinkList *p=L;          //p指向头节点，n置为0（即头节点的序号为0）
```



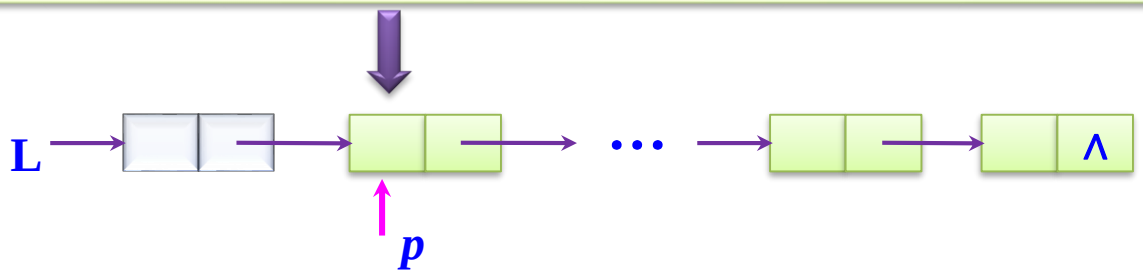
```
    while (p->next!=NULL)
    {     n++;
          p=p->next;
    }
    return(n);     //循环结束,p指向尾节点,其序号n为节点个数
}
```



（5）输出线性表DispList(L)

逐一扫描单链表L的每个数据节点，并显示各节点的data域值。

```
void DispList(LinkList *L)
{    LinkList *p=L->next;                  //p指向开始节点
     while (p!=NULL)                       //p不为NULL,输出*p节点的data域
     {     printf("%d ",p->data);
           p=p->next;                      //p移向下一个节点
     }
     printf("\n");
}
```



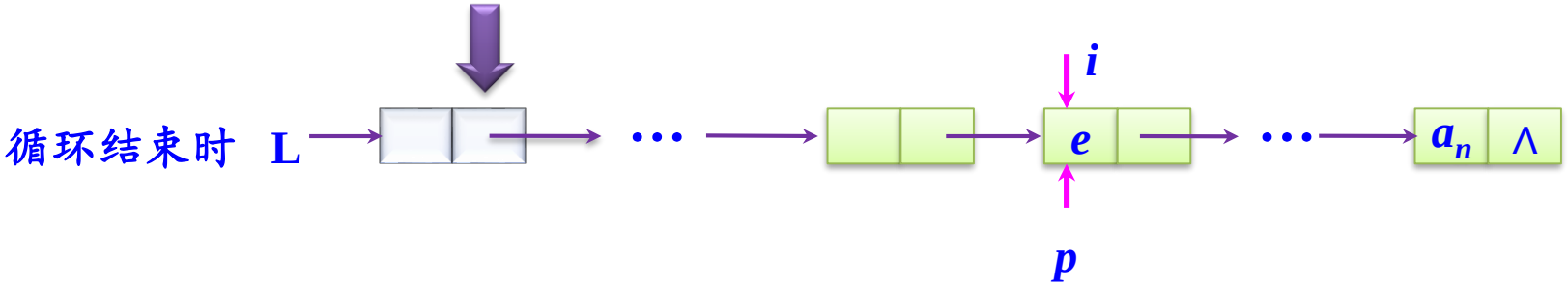
（6）求线性表L中位置*i*的数据元素GetElem(L,i,&e)

思路：在单链表L中从头开始找到第*i*个节点，若存在第*i*个数据节点，则将其data域值赋给变量*e*。

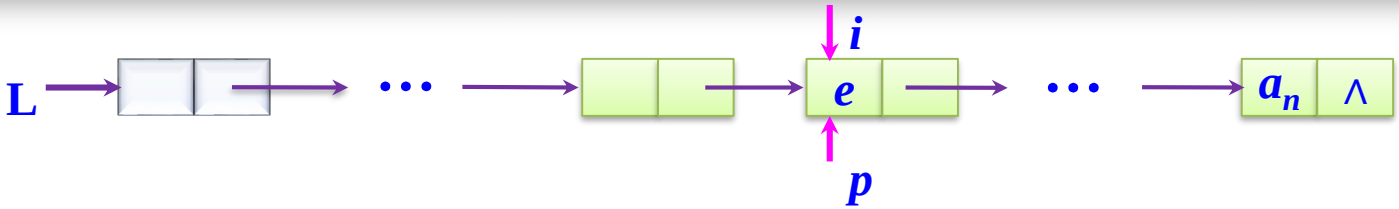
```
bool GetElem(LinkList *L,int i,ElemType &e)
{    int j=0;
     LinkList *p=L;          //p指向头节点，j置为0（即头节点的序号为0）

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

找第*i*个节点*p



```
    if (p==NULL)            //不存在第i个数据节点，返回false
        return false;
    else                    //存在第i个数据节点，返回true
    {   e=p->data;
        return true;
    }
}
```

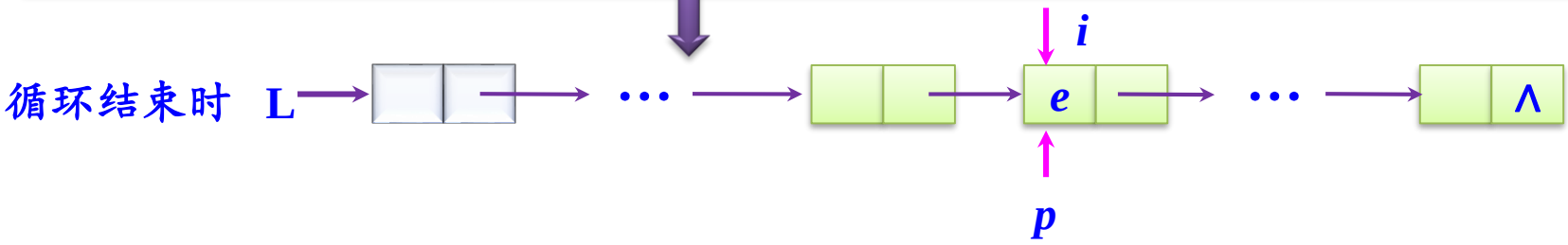


（7）按元素值查找LocateElem(L,e)

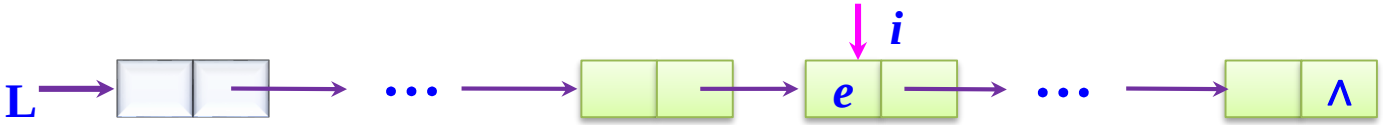
思路：在单链表L中从头开始找第1个值域与*e*相等的节点，若存在这样的节点，则返回位置，否则返回0。

```
int LocateElem(LinkList *L,ElemType e)
{    int i=1;
     LinkList *p=L->next;          //p指向开始节点,i置为1

   while (p!=NULL && p->data!=e)
   {    p=p->next;                 //查找data值为e的节点,其序号为i
        i++;
   }
```



```
    if (p==NULL)              //不存在元素值为e的节点,返回0
        return(0);
    else                      //存在元素值为e的节点,返回其逻辑序号i
        return(i);
}
```



算法的时间复杂度为O(*n*) ⇨ 不具有随机存取特性

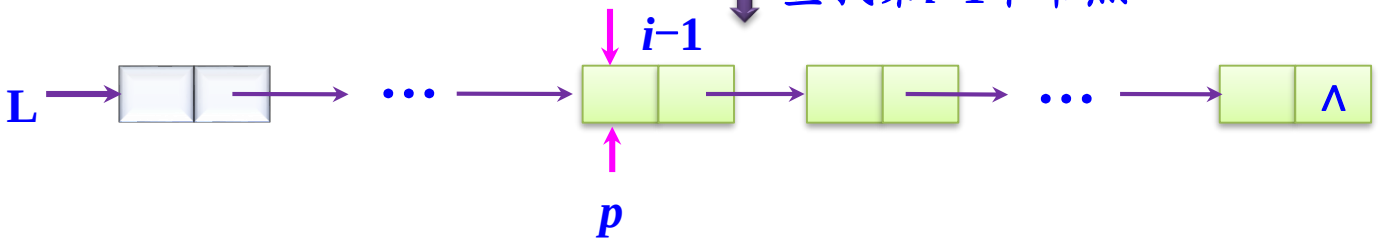
（8）插入数据元素ListInsert(&L,i,e)

思路：先在单链表L中找到第$i-1$个节点*p，若存在这样的节点，将值为e的节点*s插入到其后。

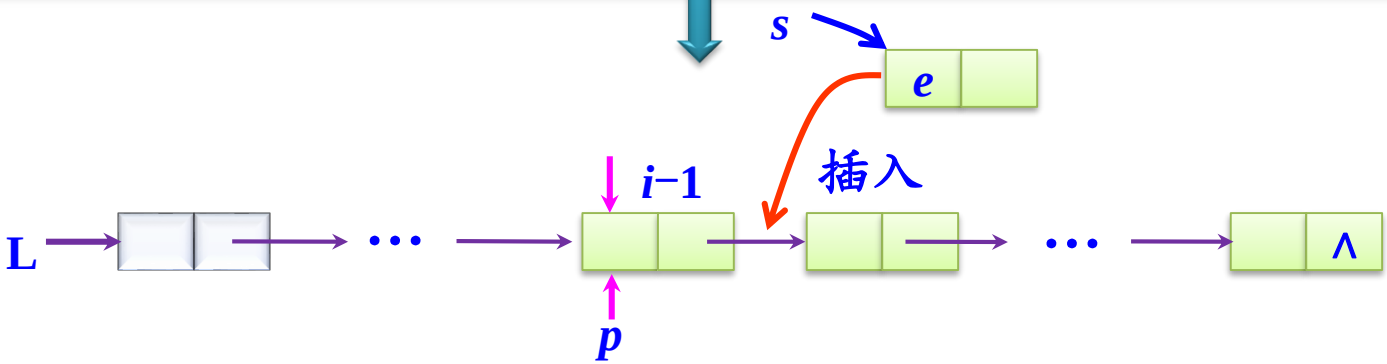
```
bool ListInsert(LinkList *&L,int i,ElemType e)
{    int j=0;
     LinkList *p=L,*s;                    //p指向头节点，j置为0

     while (j<i-1 && p!=NULL)
     {    j++;
          p=p->next;
     }
```

查找第$i-1$个节点



```
    if (p==NULL)                //未找到第i-1个节点,返回false
        return false;
    else                        //找到第i-1个节点*p,插入新节点并返回true
    {
        s=(LinkList *)malloc(sizeof(LinkList));
        s->data=e;              //创建新节点*s,其data域置为e
        s->next=p->next;        //将*s插入到*p之后
        p->next=s;
        return true;
    }
}
```

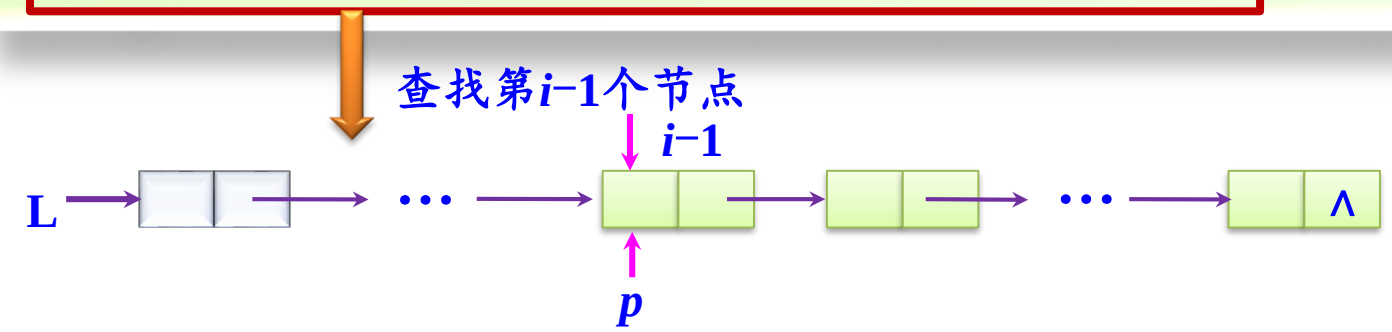


（9）删除数据元素ListDelete(&L,i,&e)

思路：先在单链表L中找到第$i-1$个节点*p，若存在这样的节点，且也存在后继节点，则删除该后继节点。

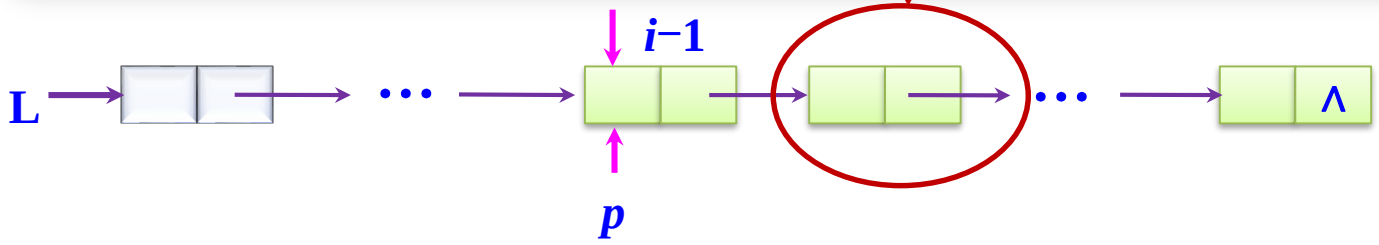
```
bool ListDelete(LinkList *&L,int i,ElemType &e)
{    int j=0;
     LinkList *p=L,*q;                    //p指向头节点,j置为0

     while (j<i-1 && p!=NULL)   //查找第i-1个节点
     {    j++;
          p=p->next;
     }
```



```
    if (p==NULL)                    //未找到第i-1个节点,返回false
        return false;
    else                            //找到第i-1个节点*p
    {   q=p->next;                  //q指向第i个节点
        if (q==NULL)                //若不存在第i个节点,返回false
            return false;
        e=q->data;
        p->next=q->next;            //从单链表中删除*q节点
        free(q);                    //释放*q节点
        return true;                //返回true表示成功删除第i个节点
    }
}
```

删除第i个节点



——本讲完——