

Lecture 5: 线性表的链式实现

5.1 单链表的实现

5.1.1 单链表的定义

5.1.1.1 定义

- 线性表的优缺点：
 - 优点： 可随机存取，存储密度高
 - 缺点： 要求大片连续空间，改变容量不方便
- 单链表的优缺点：
 - 优点： 不要求大片连续空间，改变容量方便
 - 缺点： 不可随机存取，要耗费一定空间存放指针
- 用**代码定义**一个单链表

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

```
//重命名用法
typedef <数据类型> <别名>
```

```
//增加一个新的结点:在内存中申请一个结点所需空间，并用指针P指向这个结点
struct LNode *p=(struct LNode *)malloc(sizeof(struct LNode ));
```

```
//教材中的定义方法如下
typedef struct LNode{
    ElemType data;
    struct LNode *next;
}LNode, *LinkList;
//解释:相当于重命名为LNode,并声明了指针 *LinkList
```

```
//指针的两种定义方式
LNode * L; //声明一个指向单链表第一个结点的指针(强调返回结点)
LinkList L; //同上，但是代码可读性更强(强调这是单链表)
```

5.1.1.2 初始化

概念：带头结点优点

- 由于第一个数据结点的位置被存放在头结点的指针域中，因此在链表的第一个位置上的操作和在表的其他位置上的操作一致，无须进行特殊处理。

- 无论链表是否为空，其头指针都是指向头结点的非空指针（空表中头结点的指针域为空），因此空表和非空表的处理也统一。

```
//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;        //空表暂时没有任何结点
    return true;
}

//判断单链表是否为空
bool Empty(LinkList L){
    if (L == NULL)
        return true;
    else
        return false;
}

//判空的另一种写法
bool Empty(LinkList L){
    return (L==NULL);
}
```

```
//带头结点的单链表
//初始化一个单链表(带头节点)
bool InitList(LinkList &L){
    L = (LNode *) malloc (sizeof (LNode ));
    if(L==NULL)    //内存不足，分配失败
        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(带头节点)
bool Empty(LinkList L){
    if(L->next == NULL)
        return true;
    else
        return false;
}
```

面试题：头指针和头节点的区别

- 头指针：
 - 是指向第一个节点存储位置的指针，具有标识作用；

- 头指针是链表的必要元素，无论链表是否为空，头指针都存在；
- 头结点：
 - 是放在第一个元素节点之前，便于在第一个元素节点之前进行插入和删除的操作；
 - 头结点不是链表的必须元素，可有可无，头结点的数据域也可以不存储任何信息；

5.2 单链表的插入与删除

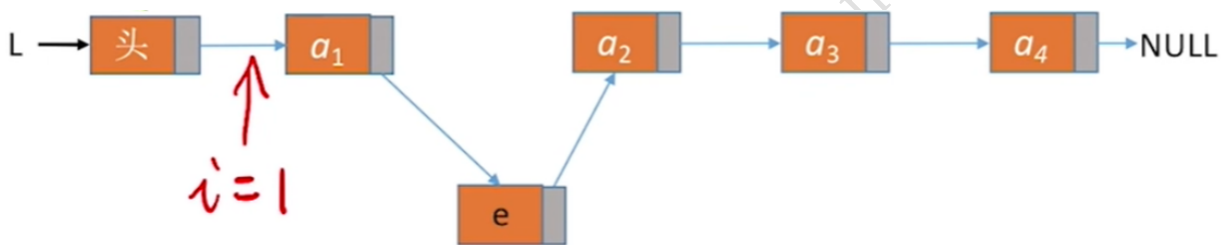
5.2.1 单链表的插入

5.2.1.1 按照位序插入节点

概念：按位序插入（带头结点）

- `ListInsert (&L, i, e)`：插入 操作
 - 在表 L 中的第 i 个位置上指定元素 e
 - 找到第 i-1 个结点，将新结点插入其后

• 图示：



代码：在第 i 个位置插入元素 e：带头结点

```
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;
    LNode *s= (LNode *)malloc(sizeof(LNode));
    s->data = e;
    s->next=p->next; //注意:和下面一行不能点到
    p->next=s;      //将结点s连接到p之后
    return true;    //插入成功
}
```

代码：按位序插入：不带头结点

```
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; //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;
    LNode *s = (LNode *)malloc(sizeof(LNode));
    s->data = e;
    s->next=p->next; //注意:和下面一行不能点到
    p->next=s; //将结点s连接到p之后
    return true; //插入成功
}

```

- 结论:
 - 不带头结点写代码更不方便，推荐用带头结点
- 注意:
 - 考试中两种情况都可能会考，考试时注意审题；

5.2.2.2 按照指定节点插入节点

后插

```

//后插操作:在p结点之后插入元素 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;
    s->next = p->next;
    p->next=s; //将结点s连接p之后
    return true;
}

```

前插

```

//前插操作: 在p结点之前插入元素 e
bool InsertPriorNode(LNode *p,ElemType e){

```

```

if(p==NULL)
    return false;
LNode *s =(LNode *)malloc(sizeof(LNode));
if(s==NULL) //分配内存失败
    return false;

//新分配的节点和p节点交换
s->next=p->next;
p->next=s;          //新结点s连接到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;
}

```

5.2.2 删除

5.1.3.1 指定位序后删除

概念: 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。

```

//按位序删除(带头结点)
bool ListDelete(LinkList &L,int i,ElemType &e){
    if(i<1)
        return false;
    LNode *p;          //指针p指向当前扫描的结点
    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个结点之后已无其他结点
        return false;
    //第i个节点不为null时
    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;         //删除成功
}

```

5.1.3.2 指定结点后删除

```

//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; //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;
}

```

- 分析：时间复杂度
 - 如果 p 是最后一个结点，只能从表头开始一次寻找 p 的前去，时间复杂度 $O(n)$;
 - 但是，在中间时，时间复杂度为 $O(1)$ (因为结点已知)

5.3 单链表的查找和建立

5.3.1 单链表的查找

按位查找

```

//按位查找，返回第i个元素(带头结点)
LNode* GetElem(LinkList L,int i){
    if(i<0)
        return NULL;
    LNode *p;    //指针p指向当前扫描到的结点
    int j=0;      //当前p指向的第几个结点
    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){ //循环找到第i个结点
        p=p->next;
        j++;
    }
    return p;
}

```

```

//王道书上的代码
LNode* GetElem(LinkList L,int i){
    int j=1;      //计数，初始为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;      //返回第i个结点的指针，若i大于表长，则返回NULL
}

```

按值查找

```

//按值查找，找到数据域==e的结点
LNode* LocateElem(LinkList L, ElemType e){
    LNode *p = L->next;
    //从第1个结点开始查找数据域为e的结点
    while(p != NULL && p->data != e)
        p = p->next;
    return p; //找到后返回该结点指针，否则返回NULL
}
//时间复杂度:O(n)

```

表的长度

```

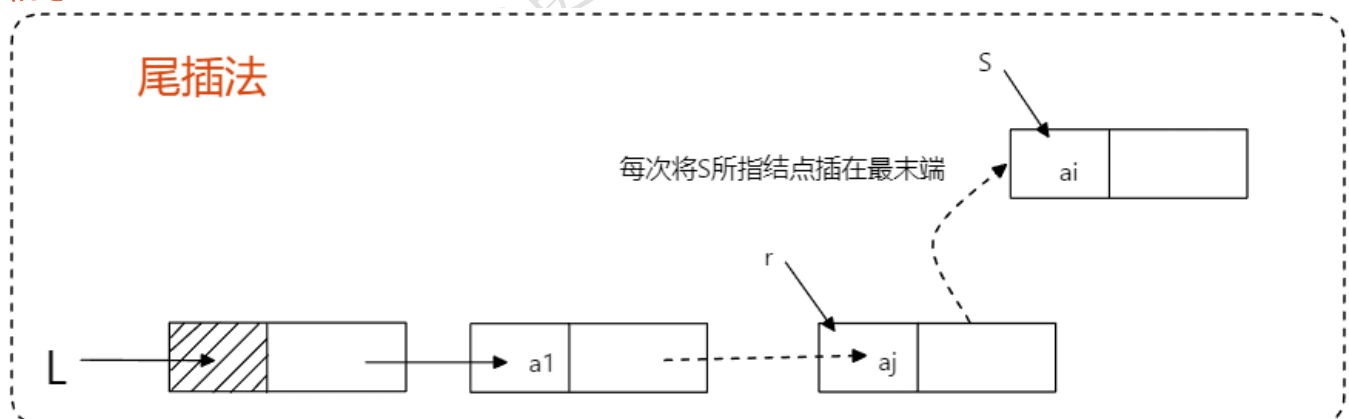
//求表的长度
int Length(LinkList L){
    int len=0; //统计表长
    LNode *p = L;
    while(p->next != NULL){
        p = p->next;
        len++;
    }
    return len;
}

```

- 时间复杂度: $O(n)$

5.3.2 单链表的建立

概念



- 头插法，尾插法：核心就是初始化操作，指定结点的后插操作；

后插

```

//后插操作：在p结点之后插入元素 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;
}

```

尾插

```

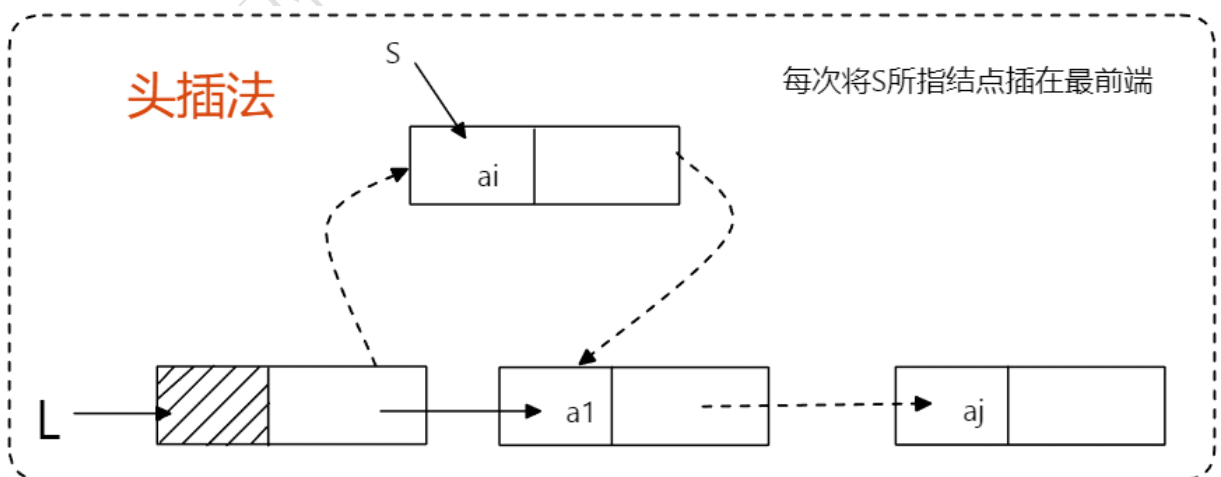
//尾插法建立单链表（顺序）
LinkList List_TailInsert(LinkList &L){ //正向建立单链表
    int x; //设ElemType为整型
    L=(LinkList)malloc(sizeof(LNode)); //建立头结点
    LNode *s,*r=L; //r为表尾指针
    scanf("%d",&x); //输入结点的值
    while(x!=9999){ //输入9999表示结束
        s=(LNode *)malloc(sizeof(LNode));
        s->data=x;
        r->next=s;
        r=s;
        scanf("%d",&x);
    }
    r->next=NULL; //尾节点指针置空
    return L;
}

```

- 核心： r 指针一直指向链表尾部最后一个节点，当输入新的数值、插入新的节点时，通过 r 将当前尾节点指向新的插入节点；

概念：头插法

- 图示：



- 代码：

```

//头插法建立单链表（逆序）
LinkList List_HeadInsert(LinkList &L){ //逆向建立单链表
    LNode *s;
    int x;

```

```

L=(LinkedList)malloc(sizeof(LNode)); //创建头结点
L->next=NULL; //初始为空链表
scanf("%d",&x); //输入结点的值
while(x!=9999){ //输入9999表示结束
    s=(LNode*)malloc(sizeof(LNode)); //创建新结点
    s->data=x;
    s->next=L->next;
    L->next=s; //将新结点插入表中，L为头指针
    scanf("%d",&x);
}
return L;
}
//时间复杂度:O(n)

```

- 注意：这里是链表的逆置

5.4 双链表

5.4.1 双链表的初始化

带头节点

```

//定义双链表
typedef struct DNode{ //注意:这里的D表示Double
    ElemType data;
    struct DNode *prior,*next; //prior:先, 先前的
}DNode,*DLinklist;

```

初始化

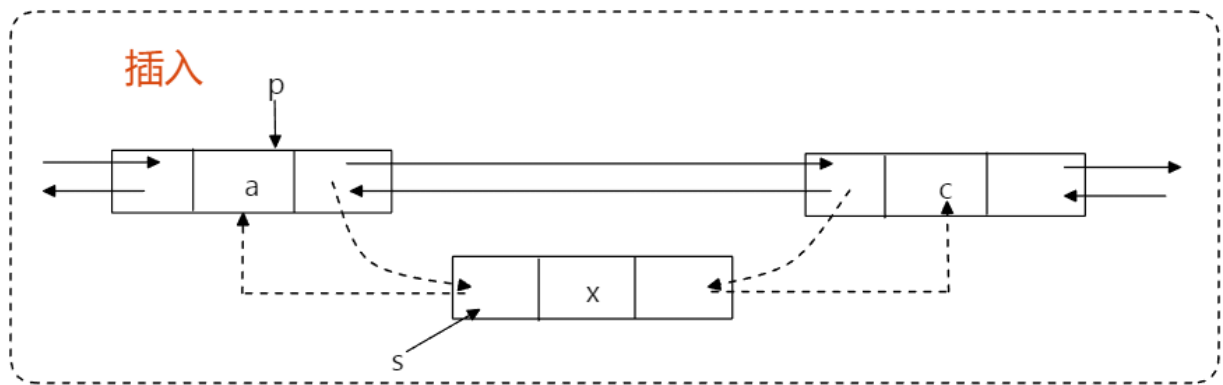
```

//初始化双链表
bool InitDLinkList(DLinklist &L){
    L=(DNode *)malloc(sizeof(DNode)); //分配一个头结点
    if(L==NULL) //内存不足, 分配失败
        return false;
    L->prior =NULL; //头结点的prior永远指向NULL
    L->next =NULL; //头结点之后暂时还没有结点
    return true;
}

```

插入

- 图示:

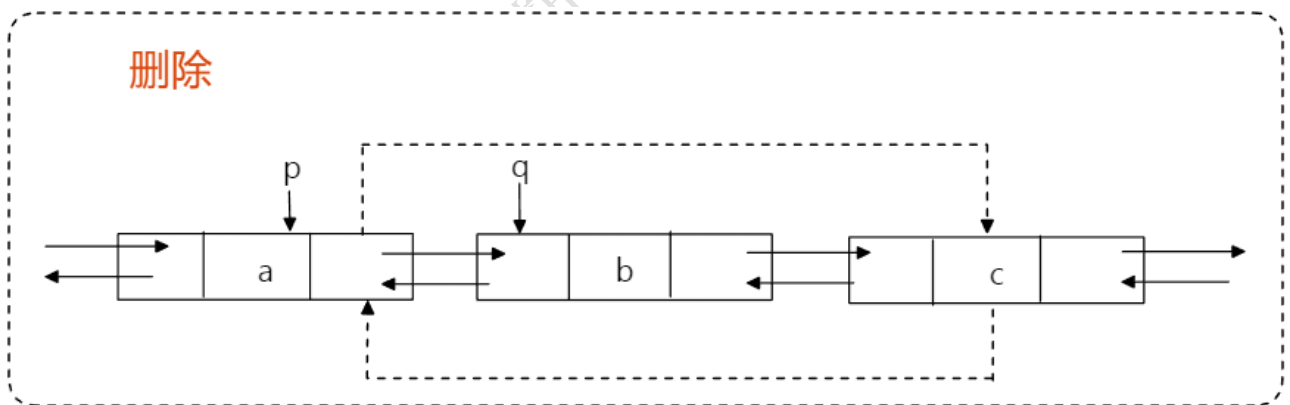


- 代码:

```
//在p结点之后插入s结点
bool InsertNextDNode(DNode *p, DNode *s){
    if(p==NULL || s==NULL) //非法参数
        return false;
    s->next=p->next;        //将结点*s插入到结点*p之后
    if(p->next!=NULL) //如果p结点有后继结点
        p->next->prior=s;
    s->prior=p;              //注意代码顺序
    p->next=s;
}
```

5.4.2 双链表的删除与遍历

概念: 双链表的删除与遍历



删除节点

```
//删除p结点的后继结点
bool DeleteNextDNode(DNode *p){
    if(p==NULL)
        return false;
    DNode *q = p->next; //找到p的后继结点q
    if(q==NULL)
        return false;
    p->next=q->next;
    if(q->next!=NULL) //q结点不是最后一个结点
        q->next->prior=p;
    free(q);
}
```

```
    return true;
}
```

删除链表

```
void DestoryList(DLinklist &L){
    //循环释放各个数据结点
    while(L->next != NULL)
        DeleteNextDNode(L);
    free(L); //释放头结点
    L=NULL; //头指针指向NULL
}
```

遍历

```
//后向遍历
while(p!=NULL){
    //对结点p做相应处理，如打印
    p=p->next;
}

//前向遍历
while(p!=NULL){
    //对结点p做相应处理
    p=p->prior;
}

//前向遍历(跳过头结点)
while(p->prior !=NULL){
    p=p->prior;
}
```

- 双链表不可随机存取，按位查找，按值查找操作都只能用遍历的方式实现。
- 时间复杂度： $O(n)$

面试题：单双链表的区别

- 1. 方向不同：单链表只有指向下一个结点的指针；
 - 双链表既有指向下一个结点的指针，也有指向上一个结点的指针
- 2. 适用情况不同：
 - 单向链表更适合元素的增加与删除；
 - 双向链表更适合元素的查询工作
- 3. 读取不同：单链表只能单向读取。双链表可以双向读取

5.4.3 循环单链表

概念：循环单链表



单链表：表尾结点的next指针指向 NULL



循环单链表：表尾结点的next指针指向头结点

代码

```
typedef 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; //指针指向下一个结点
}LNode, *LinkList;

//初始化一个循环单链表
bool InitList(LinkList &L){
    L = (LNode *)malloc(sizeof(LNode)); //分配一个头结点
    if(L==NULL)                          //内存不足，分配失败
        return false;
    L->next = L;                          //头结点next指向头结点
    return true;
}

//判断循环单链表是否为空
bool Empty(LinkList L){
    if(L->next == L) // 判断是否为自己
        return true;
    else
        return false;
}

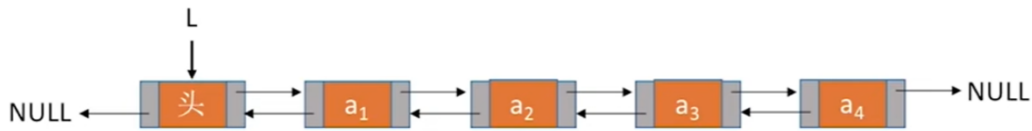
//判断结点p是否为循环单链表的表尾结点
bool isTail(LinkList L, LNode *p){
    if(p->next==L)
        return true;
    else
        return false;
}
```

- 单链表： 从一个结点出发只能找到后续的各个结点
- 循环单链表： 从一个结点出发 可以 找到其他任何一个结点

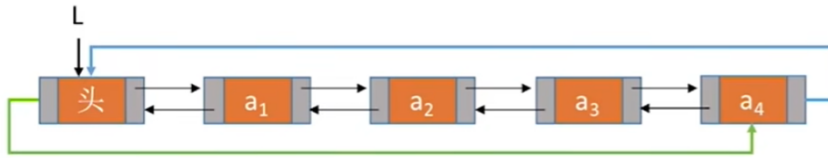
5.4.4 循环双链表

5.4.4.1 基本概念

图示



双链表：
表头结点的 prior 指向 NULL；
表尾结点的 next 指向 NULL



循环双链表：
表头结点的 prior 指向表尾结点；
表尾结点的 next 指向头结点

代码

```
typedef struct LNode{
    ElemType data;
    struct DLNode *prior, *next;
}DLNode, *DLinkList;

//初始化空的循环双链表
bool InitDLinkList(DLinklist &L){
    L = (DNode *)malloc(sizeof(DNode)); //分配一个头结点
    if(L==NULL) //内存不足，分配失败
        return false;
    L->prior = L; //头结点的prior指向头结点
    L->next = L; //头结点的next指向头结点
    return true;
}

//判断循环双链表是否为空
bool Empty(DLinklist L){
    if(L->next == L)
        return true;
    else
        return false;
}

//判断结点p是否为循环双链表的表尾结点
bool isTail(DLinklist L, DNode *p){
    if(L->next==L)
        return true;
    else
        return false;
}
```

5.4.4.1 循环双链表的插入与删除

插入

```

//循环双链表的插入
//在p结点之后插入s结点
bool InsertNextDNode(DNode *p, DNode *s){
    s->next=p->next;    //将结点*s插入到结点*p之后
    p->next->prior =s;
    s->prior=p;
    p->next=s;
}

```

删除

```

//循环双链表的删除
p->next=q->next;
q->next->prior =p;
free(q);

```

分析：需要关注的核心问题

- 1. 如何判断节点是否为空；
- 2. 如何判断节点 p 是否是表头/表尾节点；
- 3. 如何在：
 - 表头；
 - 表尾；
 - 中间；
 - 这三个位置进行插入、删除一个节点；

5.5 静态链表

概念：静态链表

- 查找：
 - 从头结点出发挨个往后遍历结点
- 操作：
 - 插入位序为 i 的结点：
 1. 找到一个空的结点，存入数据元素
 2. 从头结点出发找到位序为 i-1 的结点
 3. 修改新结点的 next
 4. 修改 i-1 号结点的 next
 - 删除某个结点：
 1. 从头结点出发找到前驱结点
 2. 修改前驱结点的游标
 3. 被删除结点 next 设为-2
- 特点：
 - 优点：
 - 增，删操作不需要大量移动元素

- 缺点：
 - 不能随机存取，只能从头结点开始依次往后查找：容量固定不变

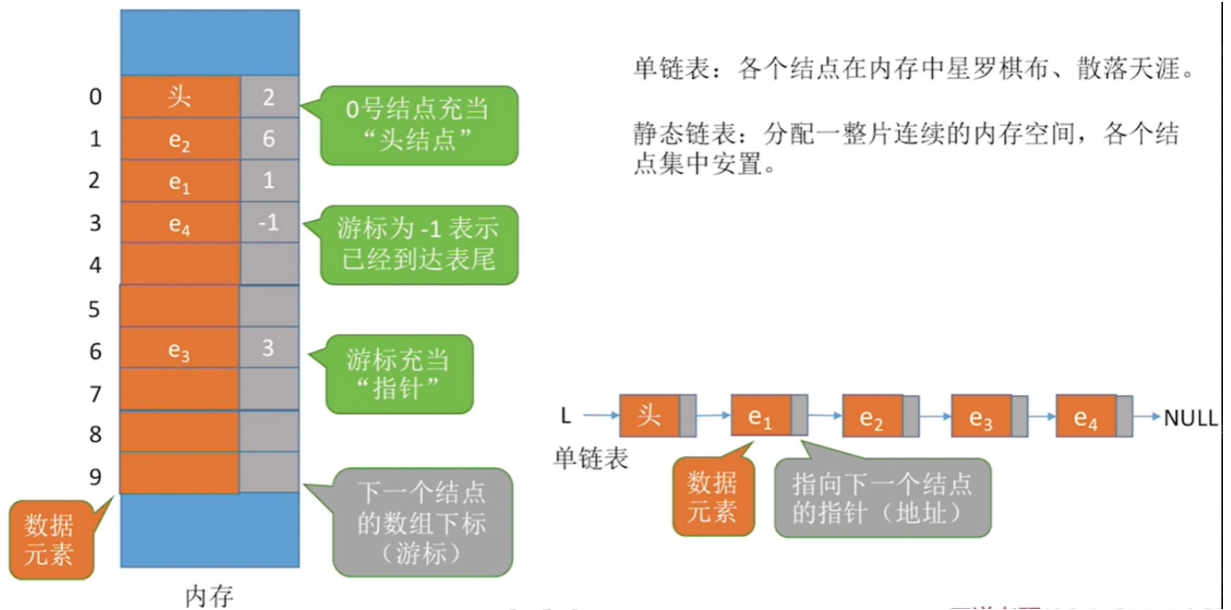
- 适用场景：

- 1. 不支持指针的低级语言
- 2. 数据元素数量固定不变的场景 (如操作系统的文件分配表 FAT)

- 注意：

- 脏数据的初始值，需要在初始化的时候设为一个固定的、不会被使用的值，比如-2，这样可以用于识别当前地址处的数据是否是空数据，能否被分配；

- 图示：



- 代码：

```
//用代码定义一个静态链表
#define MaxSize 10 //静态链表的最大长度
struct Node{       //静态链表结构类型的定义
    ElemType data; //存储数据元素
    int next;      //下一个元素的数据下标
}
```

```
//书上的写法
#define MaxSize 10 //静态链表的最大长度
struct struct{     //静态链表结构类型的定义
    ElemType data; //存储数据元素
    int next;      //下一个元素的数组下标
}SLinkList[MaxSize];
```

```
//等价于下面
#define MaxSize 10 //静态链表的最大长度
struct Node{       //静态链表结构类型的定义
    ElemType data; //存储数据元素
    int next;      //下一个元素的数组下标
};
```

```
typedef struct Node SLinkList[MaxSize];  
//相当于：用SLinkList定义"一个长度为MaxSize"的Node型数组
```

获取笔记持续更新可加微信：WandD2277