

数据结构

Data Structure

2017年秋季学期
刘鹏远

线性结构---链表---续

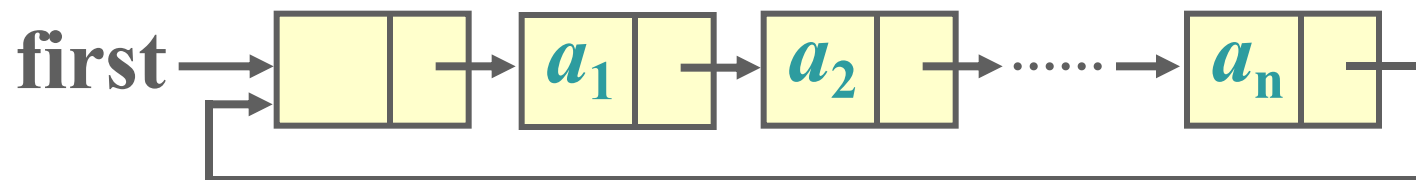
○上节课最后提出的问题

OUTLINE

循环链表
双向链表
静态链表

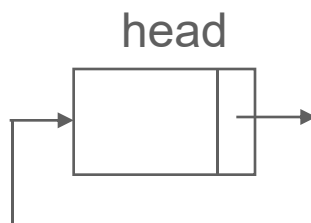
循环链表

○与普通单链表逻辑结构的差异



○逻辑上线性结构，元素间关系未变。形象上--->环

○操作等与单链表基本类似，除了头尾元素处。



循环链表

带头节点循环链表的判空条件是：

$\text{first} \rightarrow \text{next} == \text{first}$

只要知道表中某一结点的地址，就可从此节点开始，搜寻到所有其他结点的地址。

在搜寻过程中，没有一个结点的 next 域为空。

```
for (p = first->next; p != first; p = p->next)
    do S;
```

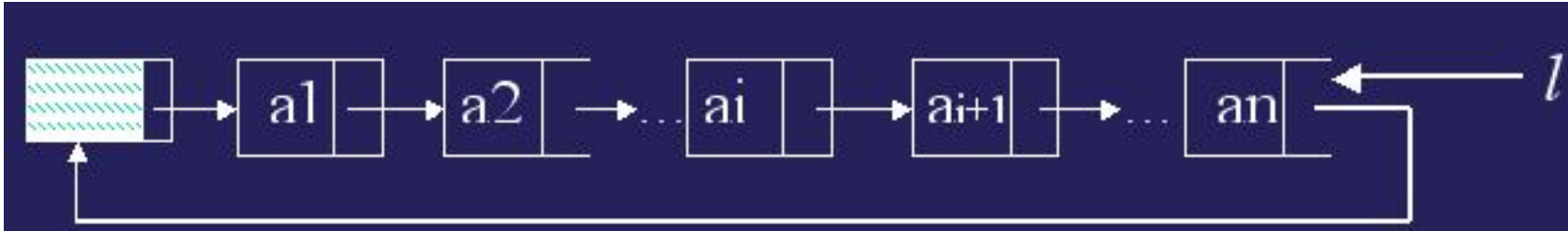
循环链表

循环链表的所有操作的实现类似于单链表，差别在于检测到链尾，指针不为NULL，而是链头。

应用时，一般需要对任意节点（非头结点）的next操作

`link_node* next(link_list L, link_list p)`

循环链表



对带有头结点的循环链表，采用指向尾元素的指针表示该链表也比较常见。why?

判断空：tail->next == tail 头结点：tail->next

非空表的首元素节点：tail->next->next

问题：

给定任意元素位置，如何实现 $O(1)$ 找直接前驱？

双（向）链表

双向链表(Double Linked List)：

构成链表的每个结点中设立两个指针域：

一个指向其直接前趋的指针域prior

一个指向其直接后继的指针域next。

这样形成的链表中有两个方向不同的链，故称为双向链表。

双（向）链表

双向链表一般增加头指针也能使双链表上的某些运算/操作变得方便。

类似地，也有双向循环链表。

虽然可以前后遍历，但是又增加了一个指针域

时间与空间的折衷。

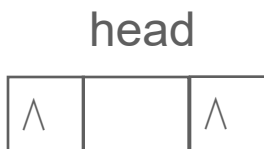
双（向）链表

○ 存储结构与实现

```
typedef struct dbl_list_node{  
    Elem_type data;  
    struct dbl_list_node *prior, *next;  
}d_list_node;
```



```
typedef d_list_node* d_link_list;
```



空双向链表



非空双向链表

双（向）链表

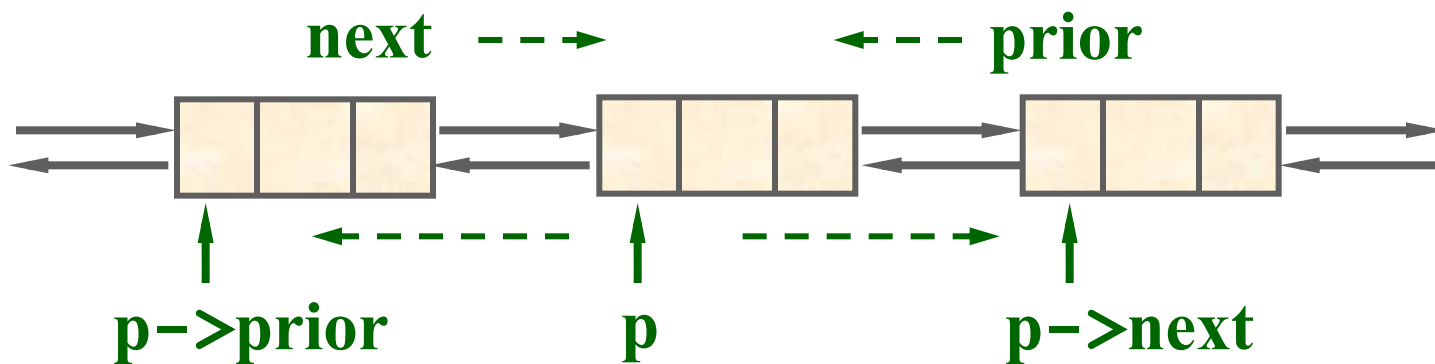
结点指向

$p \rightarrow \text{prior}$ 指示结点 p 的直接前驱结点

$p \rightarrow \text{next}$ 指示结点 p 的直接后继结点

$p \rightarrow \text{prior} \rightarrow \text{next}$ 指示结点 p 的前驱结点的后继结点，即结点 p 本身

$p \rightarrow \text{next} \rightarrow \text{prior}$ 指示结点 p 的后继结点的前驱结点，即结点 p 本身

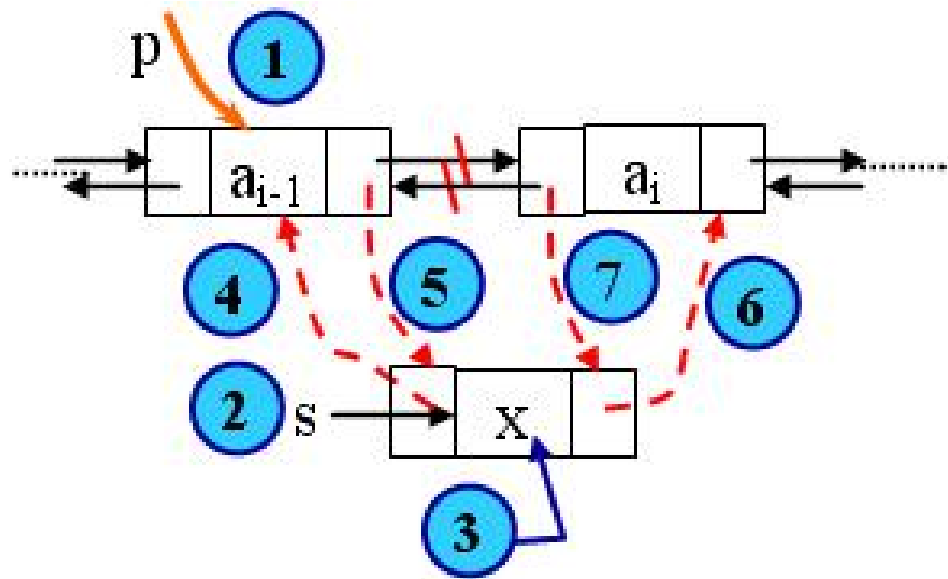


双（向）链表

双向链表基本操作

插入元素

1. 找到 $i-1$ 位置的 p 指针（找 i 位置也可）
2. $s =$
 $(d_link_list^*)malloc(sizeof(d_link_node));$
3. $s->data = x;$
4. $s->prior = p;$
5. $s->next = p->next;$
6. $p->next->prior = s;$
7. $p->next = s;$

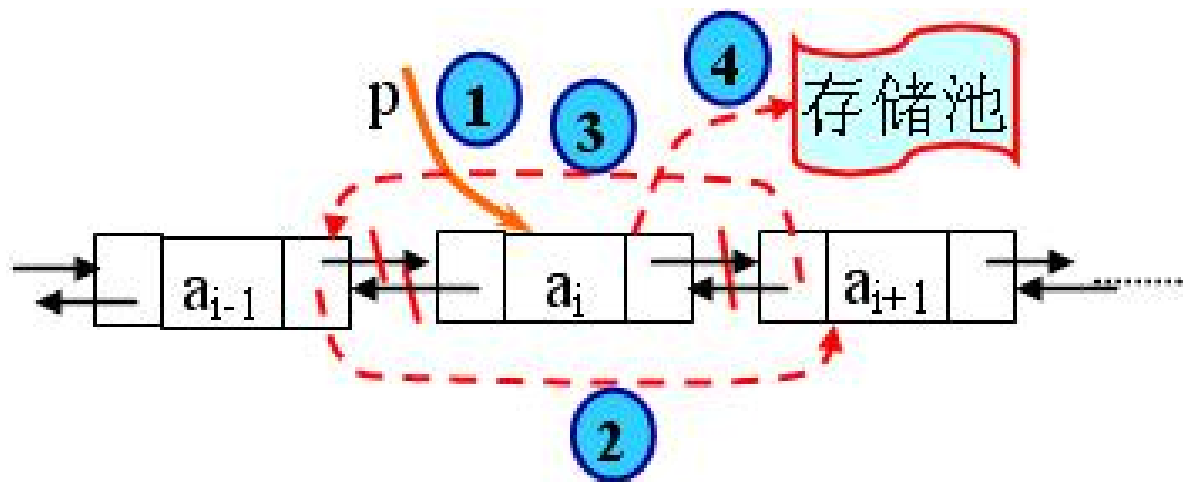


与单链类似步骤：
找前一个节点
分配新节点
新节点赋值
前/后节点链接

双（向）链表

删除元素

1. 找到 i 位置的 p 指针
2. $p \rightarrow \text{prior} \rightarrow \text{next} = p \rightarrow \text{next};$
3. $p \rightarrow \text{next} \rightarrow \text{prior} = p \rightarrow \text{prior};$
4. $\text{free}(p);$

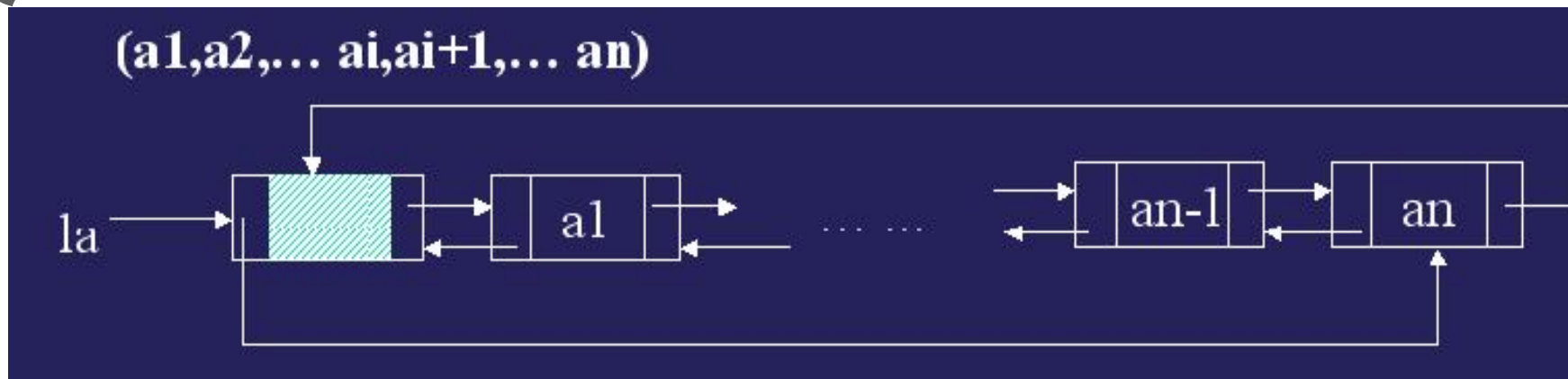


与单链类似步骤：

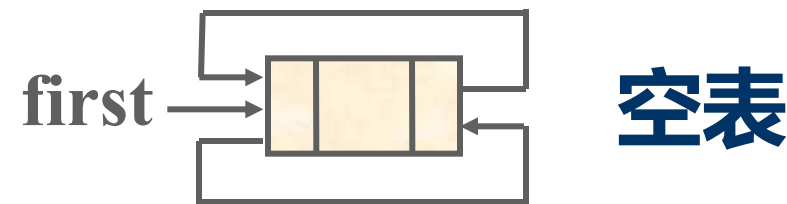
由当前节点可找前一个节点
更改前/后节点链接
释放节点

双向链表

○双向循环链表



```
typedef struct d_node {           //结点定义
    Elem_type data;               //数据
    struct d_node *prior, *next;  //指针
    //struct d_node *llink, *rlink;
} *d_list;
```



初始化

```
Status init( d_list *first ) {  
    *first = (d_node *) malloc(sizeof(d_node ));  
    if ( *first == NULL ) {                //如果分配失败  
        printf (“存储分配错!\n” );  
        return ERROR;    }  
    (*first)->prior= (*first)->next= *first;  
        //表头结点的链指针都指向自己  
    return OK;  
}
```


查找

在以 first 为头结点的双向循环链表中，从头搜寻含 x 的结点

区分是在后继方向还是在先驱方向找。

当搜索结果 p 指向表头，则搜索失败

否则返回找到的结点地址。

查找

后继方向

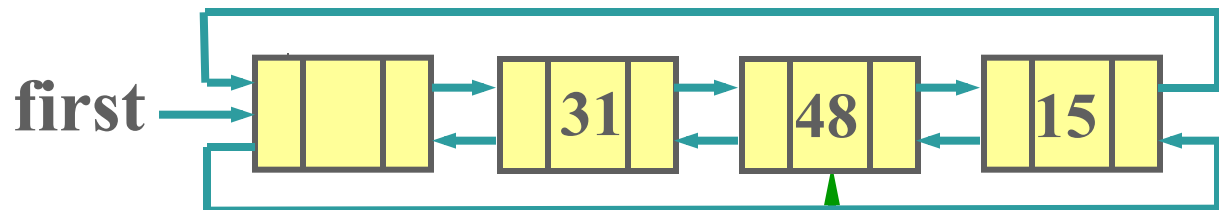
```
d_node *p = first->next;  
while (p != first && p->data != x) p = p->next;  
return p;
```

前驱方向的操作类似，只需把 next 换成 prior。

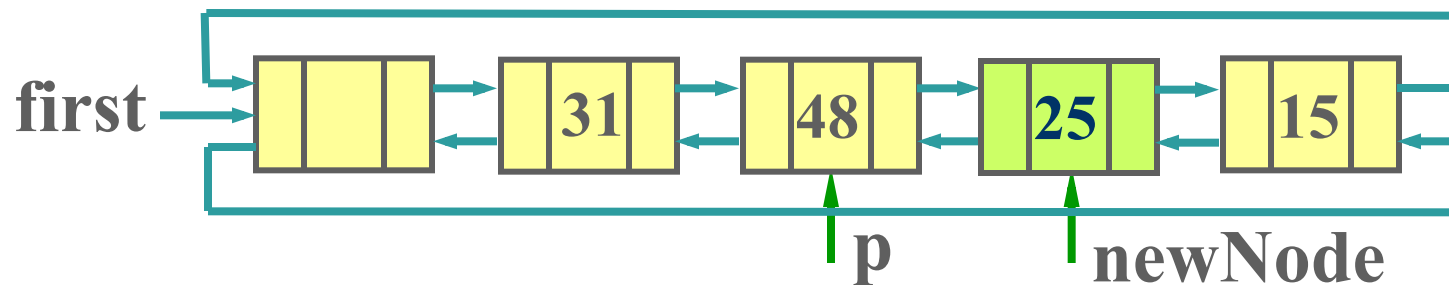
■ 定位:查找第 i 个结点在链中的位置

```
d_node* Locate ( d_list first, int i, int d ) { //伪码
    if ( i < 0 ) return NULL/ERROR;
    d_node *p = first; int j = 0;
    while ( j < i ) {
        p = (!d) ? p->prior: p->next;
        j++;
        //d = 0前驱方向, d = 1后继方向
        if ( p == first ) return NULL;    // i太大
    }
    return p;}    //返回第 i 个结点地址
```

双向循环链表的插入（非空表）



○在结点 *p 后插入25



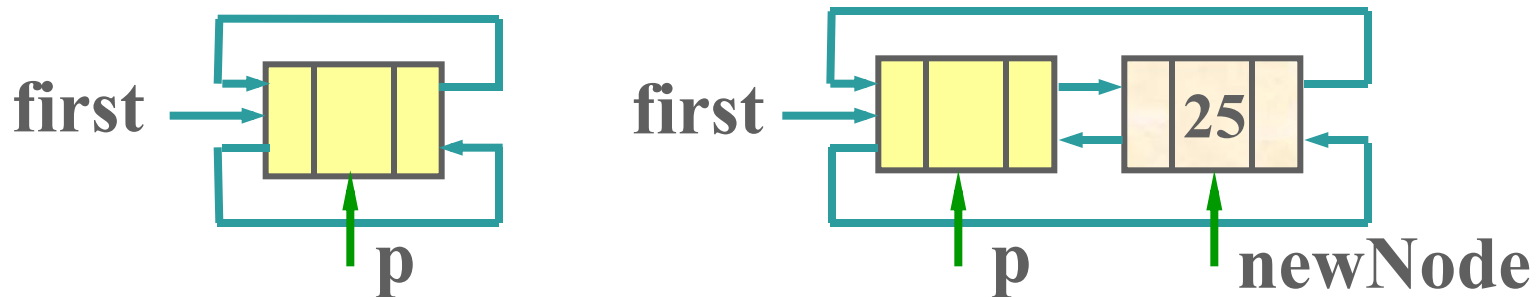
```
newNode->next = p->next;
```

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

```
newNode->prior = p;
```

```
newNode->next->prior = newNode;
```

双向循环链表的插入 (空表)



○在结点 *p 后插入25

`newNode->next = p->next; (= first)`

`p->next = newNode;`

`newNode->prior = p;`

`newNode->next->prior = newNode;`

`(first->next = newNode)`



Status Insert (d_list first, ElemType x, int i, int d) {**//伪码**

d_node *p = Locate (first, i-1, d);

//指针定位于插入位置i-1节点，节点有方向

if (p == NULL) return 0;

d_node *newNode = (d_node *) malloc(sizeof(d_node));

newNode->data = x;

if (d) { **//在p后继方向插在右方**

newNode->next = p->next;

p->next = newNode;

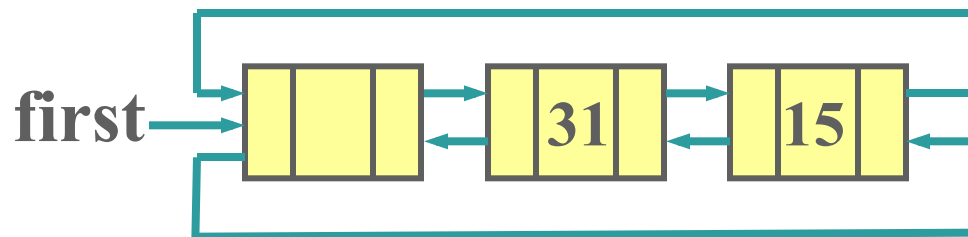
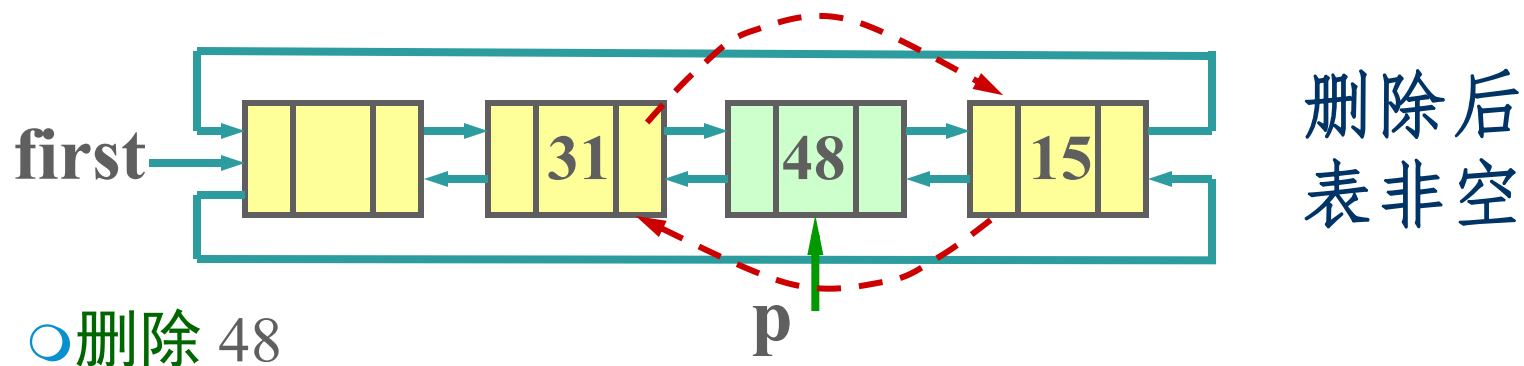
newNode->prior = p;

newNode->next->prior = newNode;

```
}  
else {           //在前驱方向插入到 p 左方  
    newNode->prior = p->prior;  
    p->prior = newNode;  
    newNode->next = p;  
    newNode->prior->next = newNode;  
}  
return OK;  
}
```

- 两个方向的插入语句类似，只是 **prior** 与 **next** 互换了一下。
- 判断内存是否申请成功自行加入

双向循环链表的删除

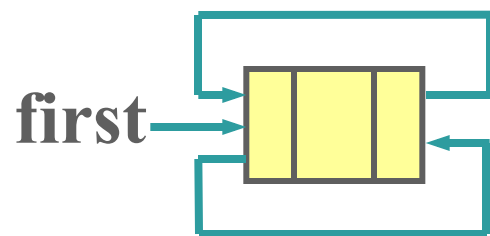
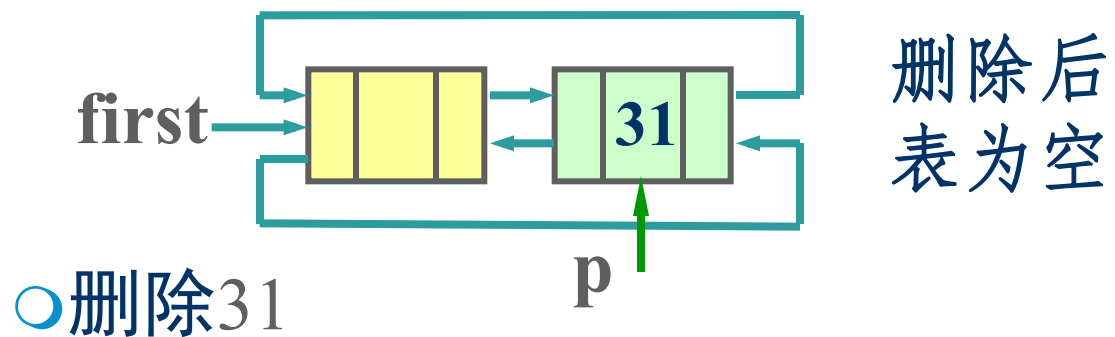


$p \rightarrow \text{next} \rightarrow \text{prior} = p \rightarrow \text{prior};$

$p \rightarrow \text{prior} \rightarrow \text{next} = p \rightarrow \text{next};$

双向链表删除可不用找该节点前一个位置

双向循环链表的删除



$p \rightarrow next \rightarrow prior = p \rightarrow prior;$

$p \rightarrow prior \rightarrow next = p \rightarrow next;$



Status Remove (DblList first, int i, int d, ElemType& x)//伪码

{ //删除在 d 指明方向的第 i 个结点, x 返回其值

 DblNode *p = Locate (first, i, d);

 //指针定位于删除结点位置

 if (p == NULL||p==first) return 0; //不能删除

 p->next->prior = p->prior;

 p->prior->next = p->next;

 //将被删结点 p 从链上摘下

 x = p->data; free(p); //删去

 return OK;}



思考：

有很多编程语言，没有指针类型，如何实现链式结构？

整个内存可视为一个大数组，因此...

静态链表

利用一维数组，对每个节点，牺牲一点儿存储空间，用来存放next域，next域存放的是相邻（后继）元素的（索引）地址。

将内存空间转换到一个数组空间，将内存物理地址转换为数组下标索引。

物理相邻的元素，逻辑上可能不相邻，反之也可。

保留链表一个优点：插入删除无需移动元素

静态链表

- 仍然是线性结构，元素的线性（相邻）关系不是用物理的相邻表示，而是用特别的“指针”来表示。
- 由于利用一维数组，实现时元素节点存储空间非真正动态申请（只是对动态申请的模拟），因此称为静态链表。

静态链表

静态链表的（静态）存储结构

```
typedef struct static_list_node
```

```
{
```

```
    ElemType data;
```

```
    int cursor; //游标cursor, 指针, 指向下一元  
    素的下标,继续用next也可
```

```
}s_list;
```

静态链表

```
#define      MAXSIZE      1000
```

```
s_list my_list[MAXSIZE];
```

//如果没有结构体，可用pair，或用数组
嵌套等各种方式均可实现。

索引类似于节点指针（地址）

索引元素游标类似于next

静态链表

实现如下操作

```
Status init(static_list[]);
```

```
int s_malloc(static_list[]);
```

```
Status s_free(static_list[], int);
```

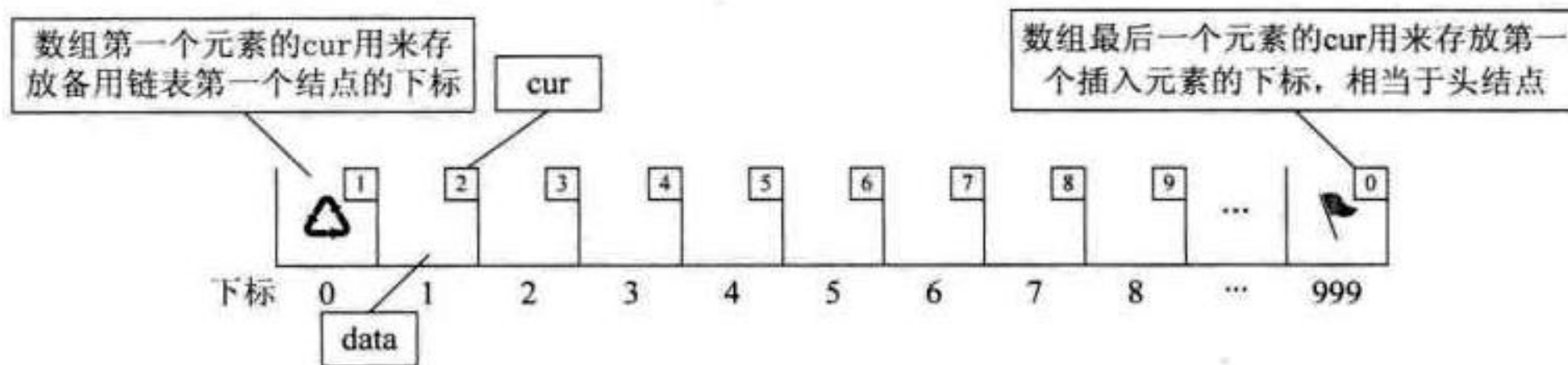
```
Status insert(static_list[], int, Elem_type);
```

```
Status del(static_list[], int, Elem_type*);
```

```
int Loc(static_list[], int);
```

```
Status traverse(static_list[]);
```


静态链表

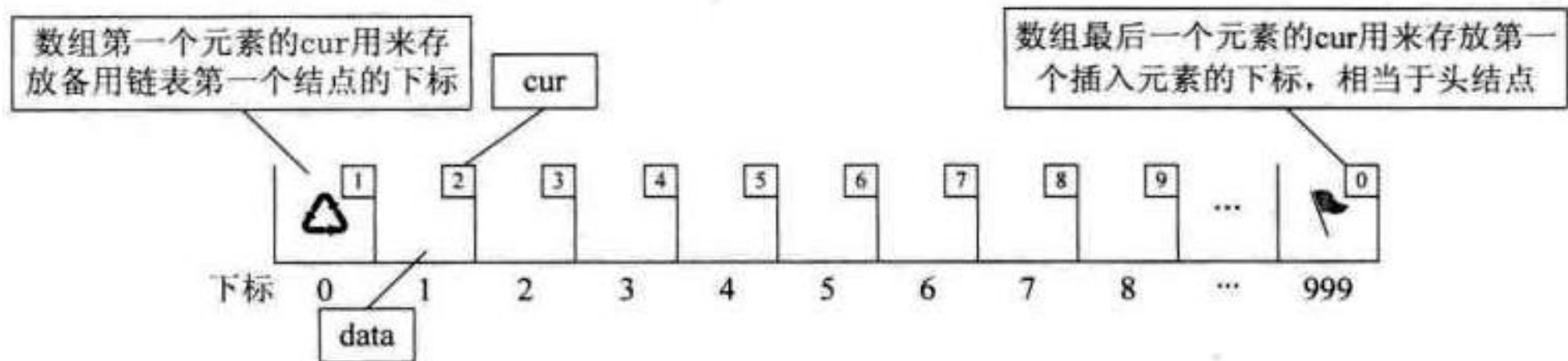


操作相当于两个带头节点的链表，节点位置可自定

备用链表(存储空间)-- Head[0]

当前链表-- Head-[MAXSIZE-1]

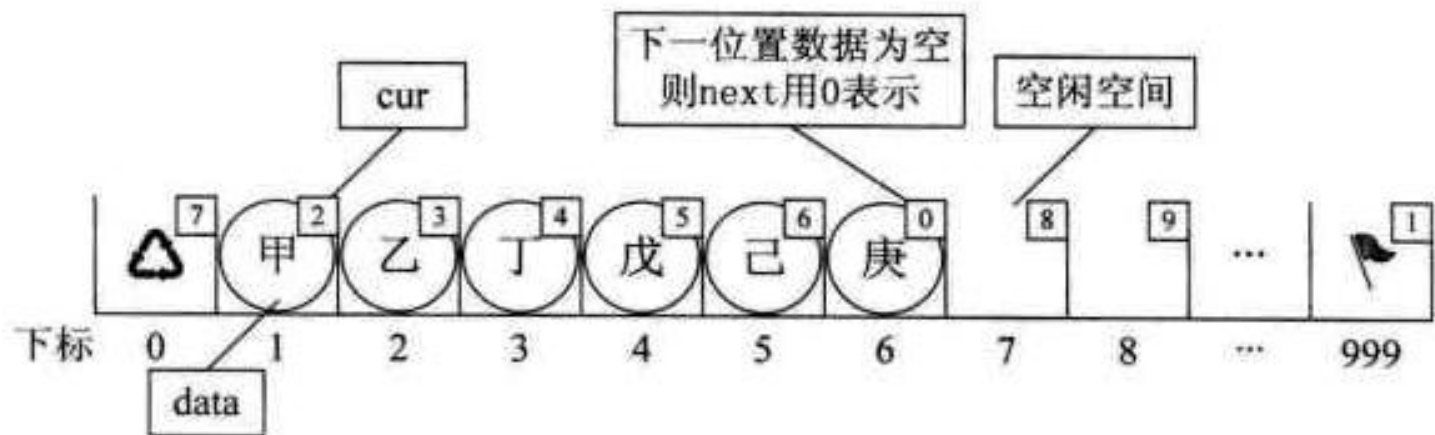
静态链表



```
Status init(static_list L[])
{
    L[MAX_SIZE-1].cursor = 0;
    L[MAX_SIZE-2].cursor = 0;
    int i;
```

```
    for(i=0;i<MAX_SIZE-2;i++)
        L[i].cursor = i+1;
    return OK;
}
//教材少一个头结点，因而不同
```

静态链表



模拟申请空间malloc及模拟释放空间free

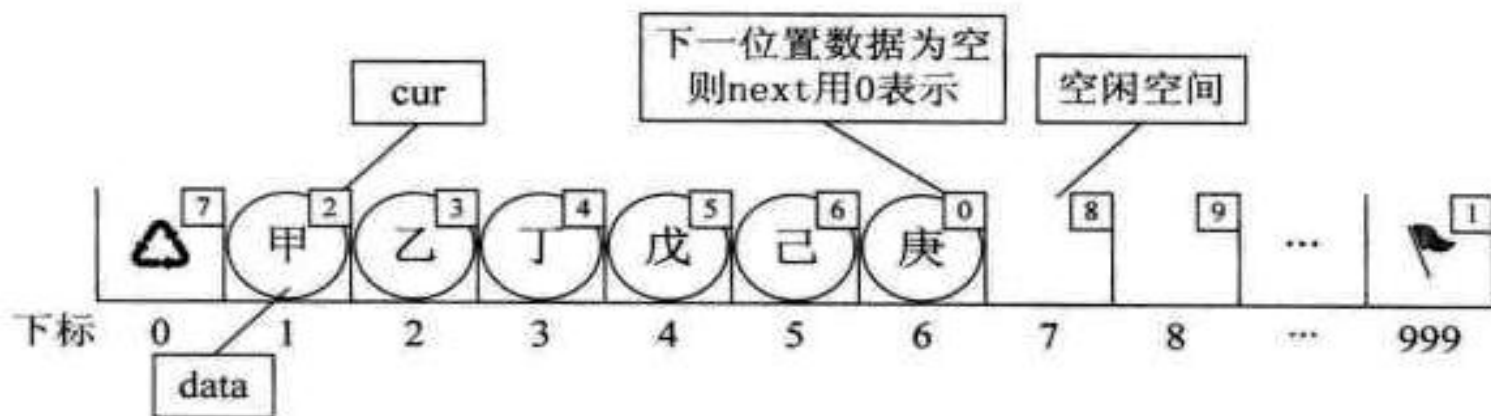
均对以0下标节点为表头的（备用）链表操作

malloc类似删除节点，但不考虑节点释放，返回节点下标。

free类似已经申请了一个新节点，要插入到链表头结点之后。

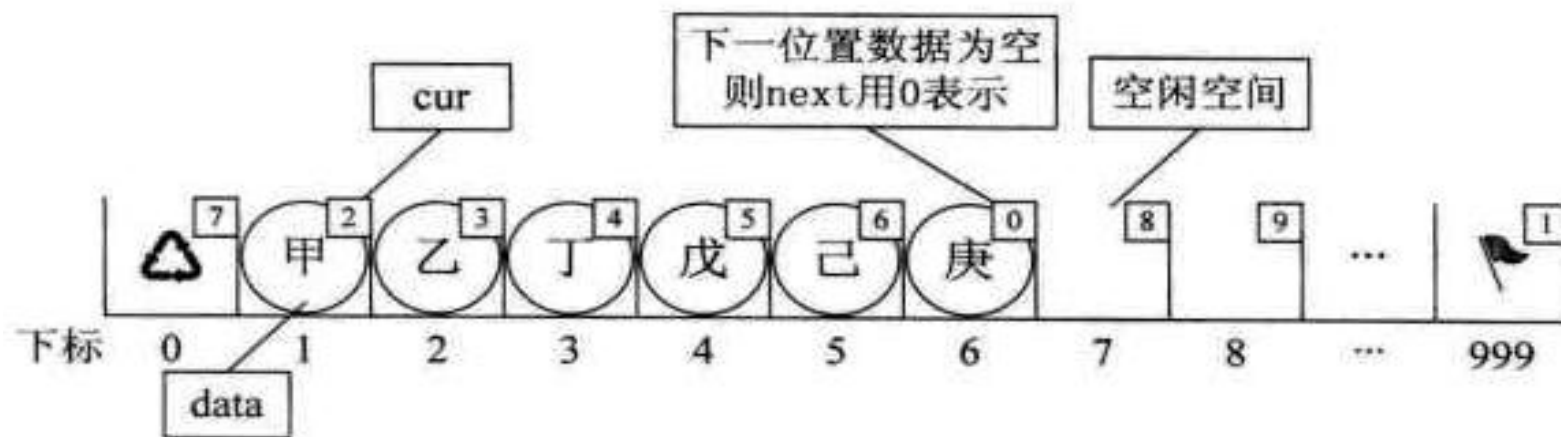
其其他操作基本针对以MAXSIZE-1节点为表头的链表操作。

静态链表



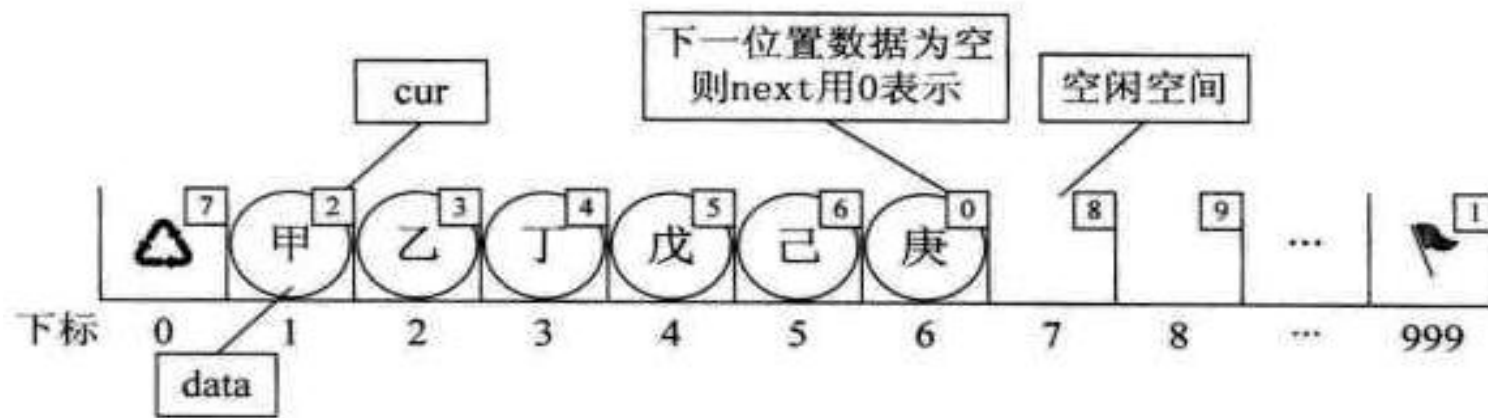
```
int s_malloc(static_list L[])
{
    int i;
    i=L[0].cursor;
    if(i) L[0].cursor = L[i].cursor;
    return i;
}
```

静态链表



```
Status s_free(static_list L[], int i)
{
    L[i].cursor=L[0].cursor;
    L[0].cursor = i;
    return OK;
}
```

静态链表



插入（在L中第i个节点之前）

- 1、为新节点申请空间，赋值
- 2、找到L中i-1位置元素
- 3、更改节点指向
- 4、返回

静态链表

```
Status insert(static_list L[], int k , Elem_type e)
{
    int j=s_malloc(L); int i;          int p=MAX_SIZE-1;
    for(i=0;i<k-1;i++)
        p=L[p].cursor;
    L[j].cursor = L[p].cursor;
    L[p].cursor = j;    L[j].data = e;
    return OK;
} //边界条件及申请是否成功等判定(已略, 自行补充)
```

静态链表

```
Status del(static_list L[], int k, Elem_type* e)
{
    int p=MAX_SIZE-1; int i;
    for(i=0;i<k-1;i++)
        p=L[p].cursor;
    int q = L[p].cursor;//保留欲删节点下标
    L[p].cursor = L[q].cursor; *e = L[q].data; s_free(L, q);
    return OK;
} //边界条件及申请是否成功等判定(自行补充)
```

静态链表

- 其余操作自行补充
- 可先实现find操作，寻值/址查找某元素位置（索引），这样如插入、删除等操作可得到进一步简化。
- 静态链表虽然整体模拟单链表，但由于一般要在一个一维数组内操作两个模拟单链表，且游标与下标在操作时容易混淆，因此需要自行实现单链表的操作并配合画图才容易掌握。

静态链表


- 优点：
 - 插入删除操作无需移动元素，给定位置时间复杂度为 $O(1)$ 。
 - 新节点空间分配快，无指针操作，较为安全。
- 缺点：
 - 无法随机存取。
 - 数组空间浪费，动态性差。

静态链表

以往考题

画出类似前面的图

初始化，插入、删除元素后的结果



请务必实现好能够正确运行的：
顺序表及链表，因为下周二解决问题要以这两个为基础。

下周二预期进度：
线性结构的应用：利用现行结构解决集合/有序表合并、一元多项式表示求值、约瑟夫问题等。