

第二章 线性表

- 2.1 线性表的类型定义
- 2.2 线性表的顺序表示和实现
- 2.3 线性表的链式表示和实现
- 2.4 一元多项式的表示和实现

上节复习

- 线性表是 n 个数据元素的有限序列
 - 数据同一性、数据顺序性
- 顺序表是用一组地址连续的存储单元依次存储线性表的数据元素
 - 采用C语言中动态分配的一维数组表示顺序表
 - 顺序表的数据结构包括`elem[]`、`length`、`listsize`
- 顺序表的创建：
 - 分配空间给`elem`，`length=0`，`listsize=初始设定值`
- 顺序表的插入：顺序表对象、位置、数据
 - 后移元素，插入，`length+1`， $n-i+1$ 个元素往后移动
 - 时间复杂度 $O(n)$
- 顺序表的删除：顺序表对象、位置
 - 前移元素（覆盖第 i 元素），`length-1`， $n-i$ 个元素往前移动
 - 时间复杂度 $O(n)$

顺序表的优缺点

■ 优点：

- ❑ 元素可以随机存取
- ❑ 元素位置可用一个简单、直观的公式表示并求取

■ 缺点：

- ❑ 在作插入或删除操作时，需要移动大量元素
- ❑ 因此引入链表，减少移动操作

2.3 链表

一. 链表的概念

- 链表是线性表的链式存储表示
- 链表中逻辑关系相邻的元素不一定在存储位置上相连，用一个链(指针)表示元素之间的邻接关系
- 线性表的链式存储表示主要有三种形式：
 - 线性链表（单链表）
 - 循环链表
 - 双向链表

2.3 链表

二. 线性链表

- 线性链表的元素称为结点 (node)
- N个结点 (a_i ($1 \leq i \leq n$) 的存储映像) 链结成一个链表, 即为线性表的**链式存储结构**, 这种存储结构是非顺序映像或链式映像。
- 结点除包含数据元素信息的**数据域**外, 还包含指示直接后继的**指针域**

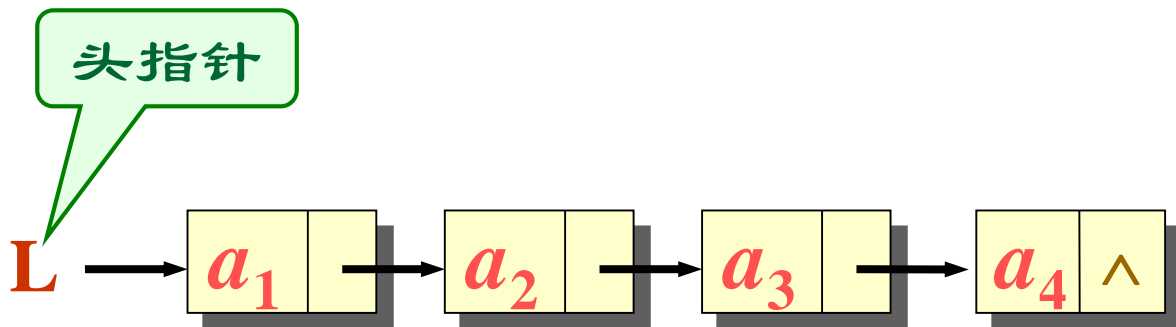


- 每个结点, 在需要时动态生成, 在删除时释放

2.3 链表

三. 单链表

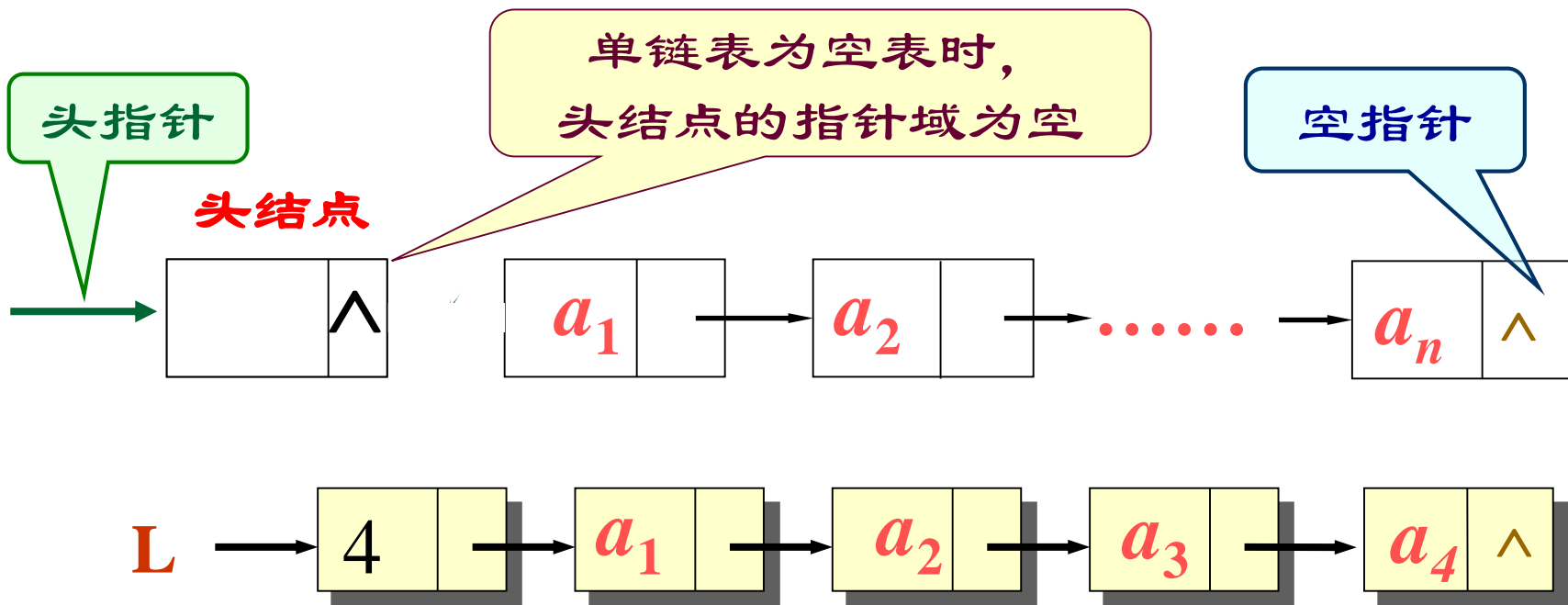
- 每个结点只包含一个指针域的链表称为线性链表（单链表）
- 以单链表中第一个结点的存储地址作为单链表的地址，称作单链表的头指针。单链表可由头指针惟一确定。
- 单链表最后一个结点的指针域为空（NULL）
- 指针是数据元素之间的逻辑关系的映像



2.3 链表

三. 单链表

- 为了方便，有时在线性链表的第一个结点之前附设一个**头结点**，其数据域可以为空，也可以为线性链表的长度信息。



2.3 链表

三. 单链表

■ 单链表的定义

//定义一个结点，也是定义一个链表

```
typedef struct LNode {
```

```
    ElemType    data;
```

// 数据域

```
    struct LNode *next;
```

// 后继指针

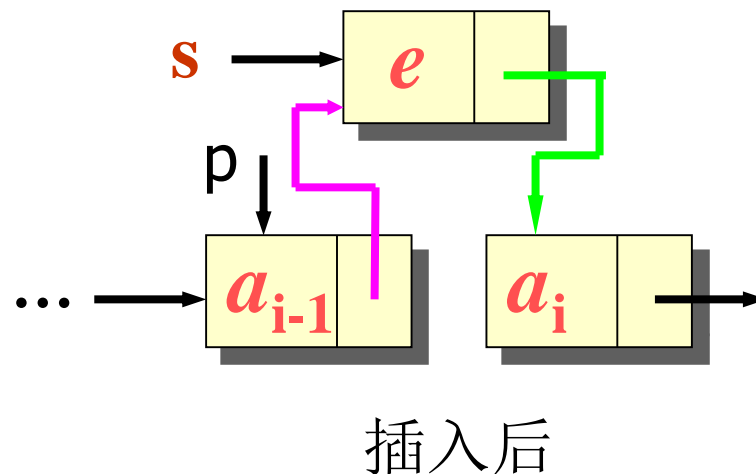
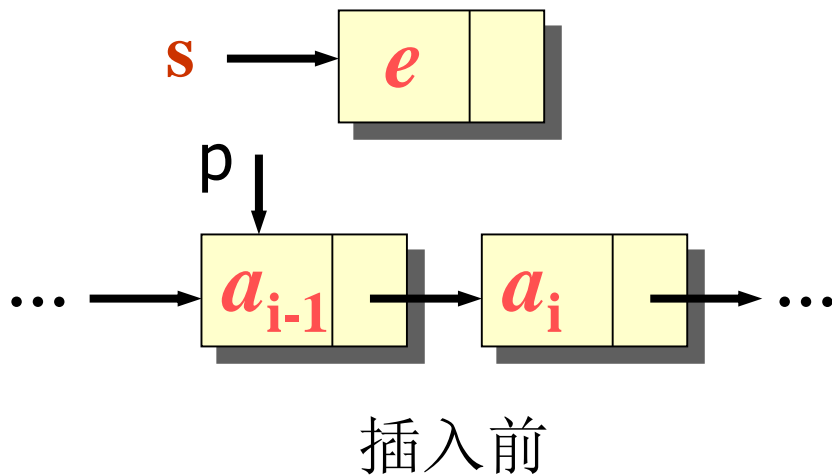
```
} LNode, *LinkList;
```



2.3 链表

三. 单链表

- 单链表的插入是在链表的第 $i-1$ 元素与第 i 元素之间插入一个新元素



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

2.3 链表

三. 单链表

■ 单链表插入的代码

```
Status ListInsert_L(LinkList &L, int i, ElemType e) {  
    // 在带头结点的单链表L中第i个位置之前插入元素e  
    p = L; j = 0;  
    while ( p && j<i-1 ) { p = p->next; j++; }           // 寻找第i-1结点  
    if (!p || j>i-1) return ERROR;  
    s = (LinkList) malloc(sizeof(LNode));                // 生成新结点  
    s->data = e;      // 插入L中  
    s->next = p->next;  
    p->next = s;  
    return OK;  
} // ListInsert_L
```

2.3 链表

三. 单链表

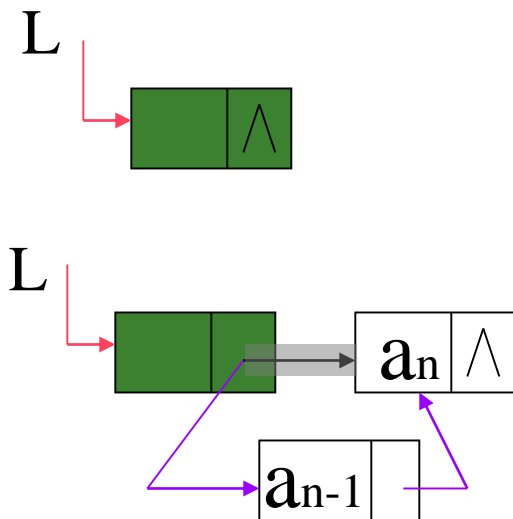
- 单链表插入的算法时间复杂度主要取决于while循环中的语句频度
- 频度与在线性链表中的元素插入位置有关，因此线性链表插入的时间复杂度为 $O(n)$

2.3 链表

三. 单链表

■ 单链表的创建—头插法

头插法，即从表头不断插入新结点。需逆序输入数据值。



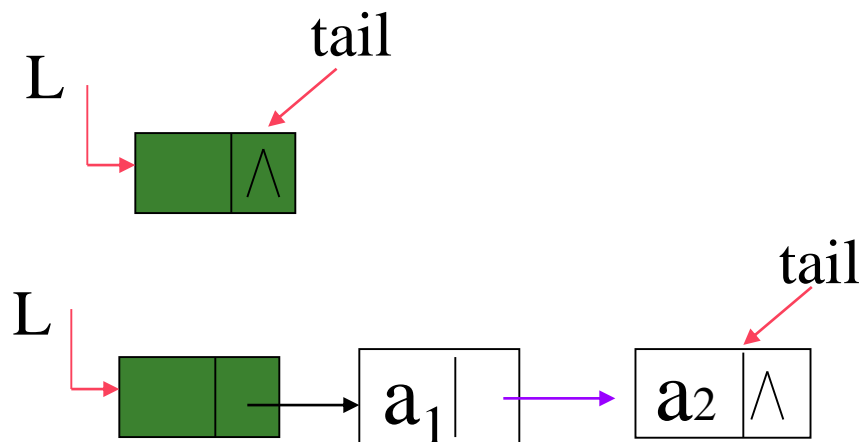
2.3 链表

三. 单链表

■ 单链表的创建—尾插法

在表尾不断插入新结点。按链表序输入数据值。

为记录尾结点，增加一个尾指针tail，指向最后一个结点。



2.3 链表

三. 单链表

■ 单链表的创建—尾插法

```
void LinkList::CreateListInTail(int n) //尾插法
{
    int i, e;
    LNode *tail = head, *s;           //初始，尾指针指向头结点，表中无数据

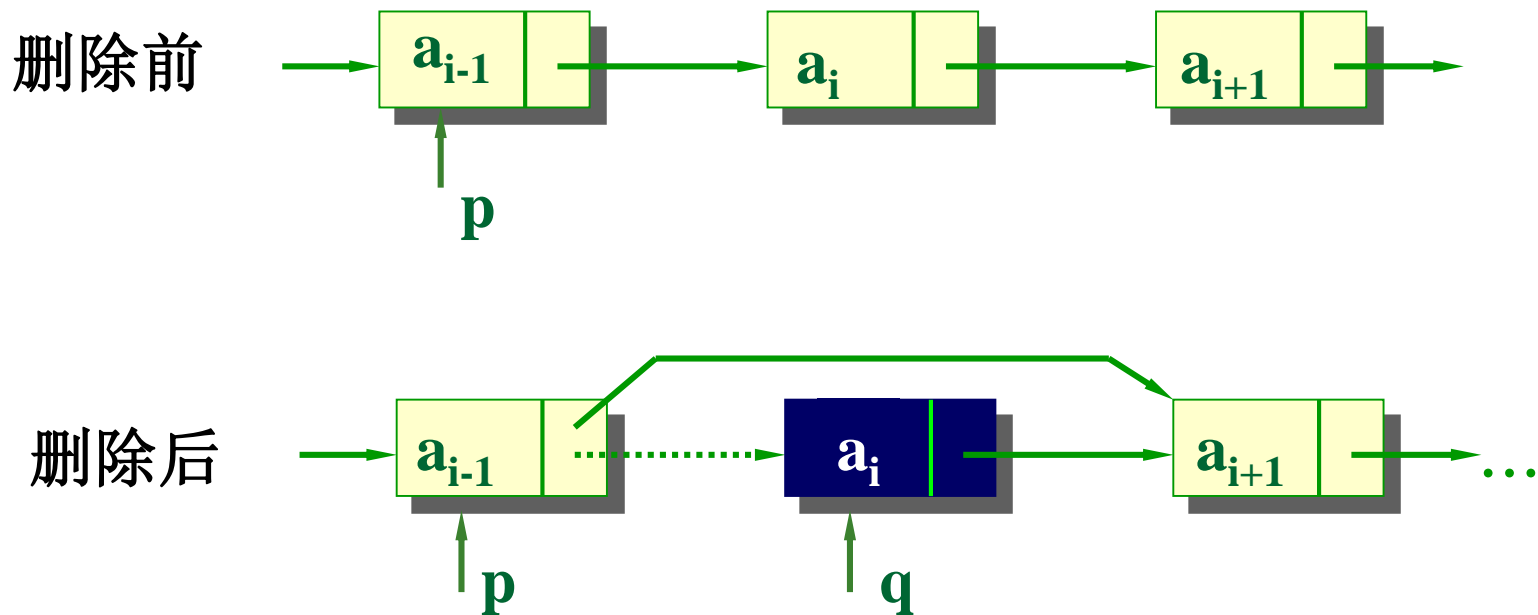
    for(i=0; i<n; i++)
    {
        cin>>e;

        s = new LNode(e);             //创建结点s,next为空
        tail->next = s;                //s链在尾结点之后
        tail = s;                     //新的尾结点是s
    }
}
```

2.3 链表

三. 单链表

- 单链表的删除是将第 i 元素删除



- $p \rightarrow \text{next} = p \rightarrow \text{next} \rightarrow \text{next}$

2.3 链表

三. 单链表

■ 单链表删除的代码

```
Status ListDelete_L(LinkList &L, int i, ElemType &e) {  
    // 在带头结点的单链表L中, 删除第i个位置的元素  
    p = L; j = 0;  
    while ( p->next && j<i-1) { p = p->next; j++; } // 寻找第i-1结点  
    if (!p->next || j>i-1) return ERROR;  
    q = p->next;  
    p->next = q->next;           // 删除i结点  
    e = q->data;                 // 取值  
    free(q);                    // 释放结点  
    return OK;  
} // ListDelete_L
```


2.3 链表

三. 单链表

- 单链表删除的算法时间复杂度主要取决于while循环中的语句频度
- 频度与在线性链表中的删除位置有关，因此线性链表删除的时间复杂度为 $O(n)$

2.3 链表

三. 单链表

■ 单链表的查找

■ 按序号查找，取单链表中的第 i 个元素

不能象顺序表中那样直接按序号 i 访问结点，而只能从链表的头结点出发，沿链域`next`逐个结点往下搜索，直到搜索到第 i 个结点为止。

因此，链表不是随机存取结构

■ 按值查找，在链表中查找是否有结点值等于给定值 k ，若有，返回结点地址

2.3 链表

三. 单链表

■ 单链表按位置查找的代码

- 从头结点开始，顺链一步步查找
- 查找第 i 个数据元素的基本操作为：移动指针 p ，比较 j 和 i 。（ j 为当前指针所指向的结点序号）

ElemType Get_Elem(LNode *L , int i)

{ //在单链表L中找第 i 个元素，并返回其值

int j ; LNode *p;

p=L->next; j=1; // 初始化，使 p 指向第一个结点， j 为计数器

while ($p \neq \text{NULL}$ && $j < i$)

{ p=p->next; j++; } // 移动指针 p 向后查找，直到指向第 i 个元素或 p 为空

if ($j \neq i$) return ERROR ;

else return(p->data);

}

2.3 链表

三. 单链表

■ 单链表的检索

LNode *Locate_Node(LNode *L, int key)

{//在单链表L中找指定值key的数据元素，并返回其指针

LNode *p=L->next;

while (p!=NULL && p->data!=key) // 移动指针p向后查找，直到找到值为key的数据元素或p为空

p=p->next;

if (p->data==key) return p;

else

{ printf(“所要查找的结点不存在!!\n”);

retutn(NULL);

}

}

2.3 链表

三. 单链表

■ 遍历整个单链表的代码框架

```
ElemType  Traverse(LNode *L)
```

```
{  LNode *p;
```

```
    p=L->next;    /* 使p指向第一个结点 */
```

```
    while (p!=NULL) {
```

```
        //.....你要执行的代码写这里
```

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

```
    // ..... 后续操作代码写这里
```

```
    return 0;
```

```
}
```

2.3 链表

三. 单链表

- 单链表查找的算法时间复杂度主要取决于while循环中的语句频度
- 频度与被查找元素在单链表中的位置有关，若 $1 \leq i \leq n$ ，则频度为 $i-1$ ，否则为 n ，因此时间复杂度为 $O(n)$

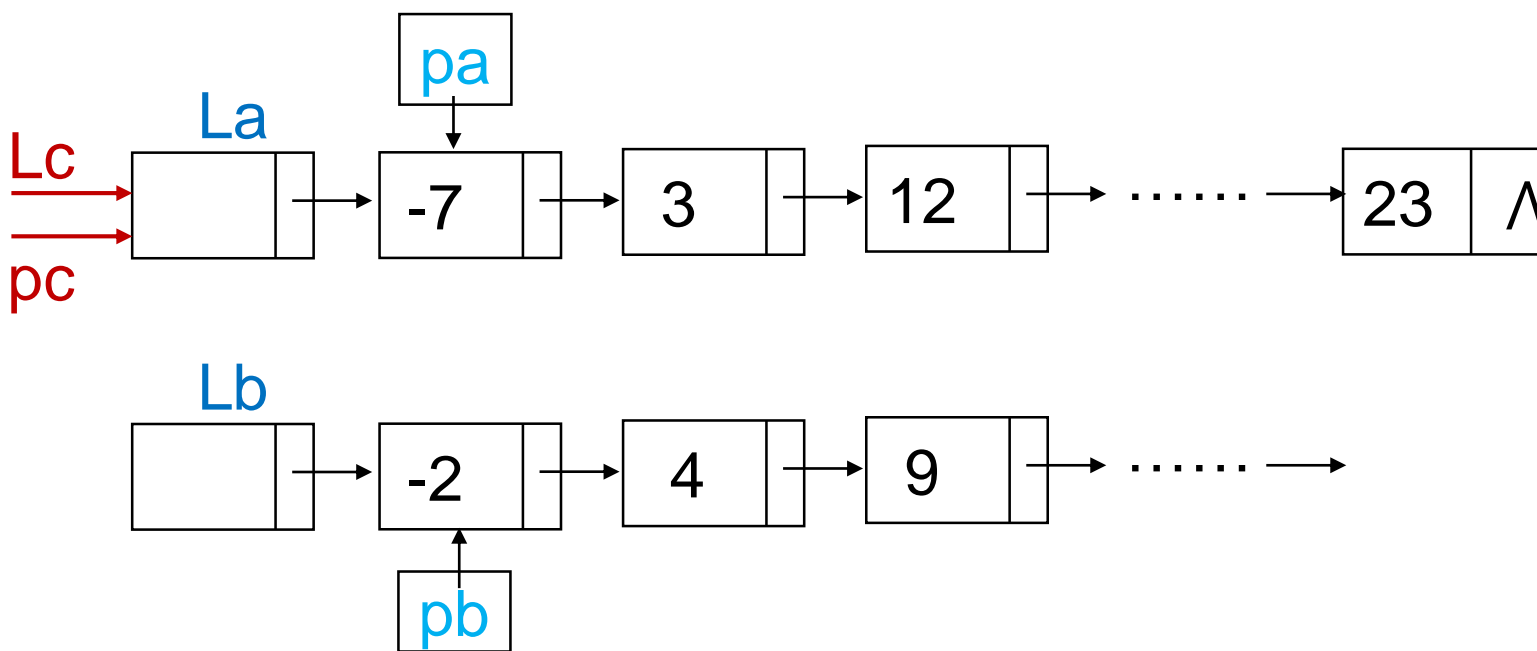
2.3 链表

三. 单链表

■ 单链表的合并

- 把两个递增有序的单链表La和Lb合并成Lc，保持递增有序

两个有序的单链表La，Lb的初始状态



三. 单链表

■ 把两个递增有序的单链表La和Lb合并成Lc，保持递增有序



2.3 链表

三. 单链表

■ 单链表合并代码

```
void MergeList_L(LinkList &La, LinkList &Lb, LinkList &Lc) {  
    LinkList pa, pb, pc;  
    pa = La->next;    pb = Lb->next;  
    Lc = pc = La;      // 用La的头结点作为Lc的头结点  
    while ( pa && pb ) {  
        if (pa->data <= pb->data)  
            { pc->next = pa; pc = pa; pa = pa->next; }  
        else { pc->next = pb; pc = pb; pb = pb->next; }  
    }  
    pc->next = pa ? pa : pb; // 插入剩余段  
    free(Lb);               // 释放Lb的头结点  
} // MergeList_L
```

时间复杂度?

2.3 链表

三. 单链表

- 静态链表，采用静态数组方式实现链表，为了与一般链表区分，称为静态链表。

	0	1	2	3	4	5	6	7	8	9	10
数据	...	6	D	C	F		B	A		E	
指针	5	7	9	2	^	8	3	6	10	4	11

2.3 链表

三. 单链表

- 可借用一维数组来描述线性链表
- 类型说明

```
#define MAXSIZE 100
```

```
Class Node{
```

```
    ElemType data;
```

```
    int cur; //游标，替代next指针，指示下一个结点的位置
```

```
};
```

```
Class SList{
```

```
    Node list[MAXSIZE];
```

```
public:
```

```
    ...
```

```
};
```

静态链表示例

0		1
1	Zhao	2
2	Qian	3
3	Sun	4
4	Li	5
5	Zhou	6
6	Wu	0
7		

修改前

0		1
1	Zhao	2
2	Qian	3
3	Sun	7
4	Li	5
5	Zhou	6
6	Wu	0
7	Shi	4

在第3个结点后
增加结点"shi"

0		1
1	Zhao	3
2	Qian	3
3	Sun	7
4	Li	5
5	Zhou	6
6	Wu	0
7	Shi	4

删除结点"qian"

2.3 链表

三. 单链表

◆ 静态链表与单链表区别：

- 1、静态链表将暂时不用的结点链成一个备用链表；
- 2、插入时，从备用链表中申请结点。
- 3、删除结点时，将结点放入备用链表。

2.3 链表

三. 单链表

■ 静态链表实现策略:

● 备用链表

✎ 为了辨明数组中那些分量未被使用，将所有未被使用过以及被删除的分量用游标链成备用链。

● 一种策略（浪费两个分量）

✎ 将数组的第一个分量用来做备用链表的头结点，串起整个备用分量。

✎ 将数组的第二个分量用来做静态链表的头结点。

■ 静态链表的实现

(1) 初始化备用链

0		1
1		2
2		3
3		4
4		5
5		6
6		7
7		0

//建立备用链表,
//list[0]为其头结点
For (i=0;i<MAXSIZE-1;i++)
 list[i].cur =i+1;
 list[MAXSIZE-1]=0

(2) 分配静态链表头结点

0		2
1		1
2		3
3		4
4		5
5		6
6		7
7		0

// 从备用链上取得第一个备用结点的下标

k = List[0].cur; // k=1

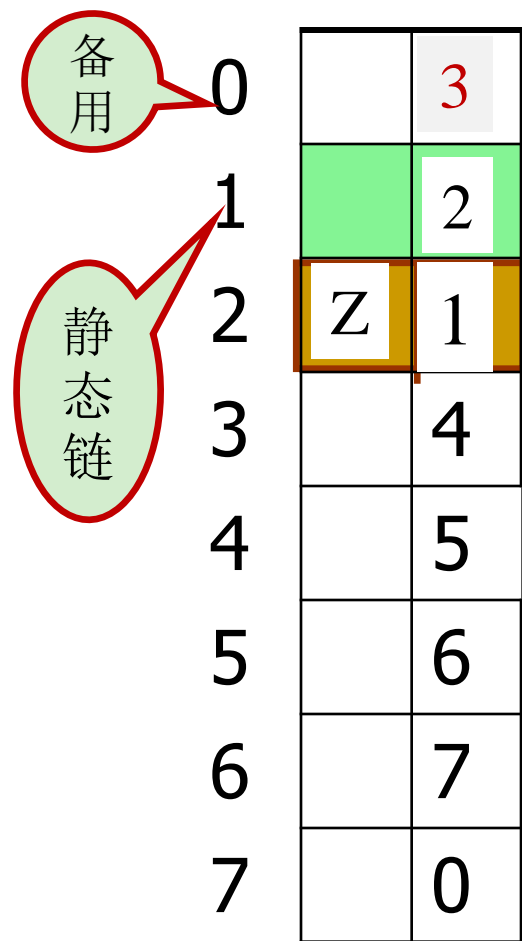
//修改备用链头结点的游标指向下一个备用
结点

List[0].cur=list[k].cur;

//k为当前获得的结点的下标，在此为静态
链表分配头结点，并将链表末尾的游标
初始化为头结点下标。

List[k].cur=1;

(3) 静态链表插入结点（头插法）



0		3
1		2
2	z	1
3		4
4		5
5		6
6		7
7		0

//首先获得一个备用结点

```
k = list[0].cur;
```

// 并修改备用链头结点的游标

```
List[0].cur=list[k].cur;
```

//数据元素赋值为‘z’

```
List[k].data = 'z';
```

// 在静态链表中插入第一个元素

```
List[k].cur=list[1].cur;
```

```
List[1].cur=k;
```

(4) 删除静态链表结点

0		5
1		4
2	z	1
3	h	2
4	a	3
5		6
6		7
7		0

// 删除静态链表中的第*i*个结点,
假设它在数组中的下标为

p

,它
的前驱结点的下标为

q

。

//修改

q

的cur指向

p

的cur

List[**q**].cur=list[**p**].cur;

//将

p

结点插入到备用链表的头
结点之后

List[**p**].cur=list[0].cur;

List[0].cur=**p**;

0		3
1		4
2	z	1
3	h	5
4	a	2
5		6
6		7
7		0

← **p**

← **q**

小结

- 链表是线性表的链式存储表示，逻辑上相邻的元素不一定在存储位置上相连
 - 链表由结点组成，每个结点包含数据域Data和指针Next
 - 带头结点的单链表
 - 单链表空：head->next == NULL；单链表末尾p->next=NULL
- 链表的运算
 - 单链表的查找要从头结点开始往后搜索，时间复杂度为 $O(n)$
 - 单链表的插入和删除的时间复杂度都为 $O(n)$
 - 插入修改指针：S->next=P_{i-1}->next；P_{i-1}->next=S；
 - 删除修改指针：P_{i-1}->next=P_i->next；free(P_i)；

练习

习题1. 已知某链表的元素结构如下所示，求第3号元素的内存地址是多少？若删除第2号元素，第1号元素的指针域是多少？

元素编号	数据域 Data	指针域 Next
Head	439043903867	A0C700
1	76	ABC358
2	35	BD6222
3	64	CC235F
4	55	NULL

习题2. 在一个单链表中，若p所指结点是q所指结点的前驱结点，则在结点p、q之间插入结点s的代码是？

练习

习题3. 写一算法将单链表中值重复的结点删除，使所得的结果链表中所有结点的值均不相同，并分析算法时间复杂度。

习题4. 已知带头结点`head`的单链表，请写出判断链表元素是否递减的算法，并分析算法时间复杂度。