

第二章 线性表

自然情况登记表

序号	姓名	专业	性别	通信地址	联系方式
1	马云	计算机	男	阿里	112233
2	李彦宏	计算机	男	百度	556688
3	马化腾	视觉	男	腾讯	
4	扎克伯格	计算机	男	facebook	776633
5	盖茨	软件	男	微软	
6					
7					



本章节目录

[2.1 线性表的抽象数据类型](#)

[2.2 线性表的顺序存储结构](#)

[2.3 线性表的链式存储结构](#)

2.1 线性表的抽象数据类型

- ❖ 一个线性表(Linear List) 是由 $n(n \geq 0)$ 个相同类型的数据元素(结点) 组成的有限序列。
- ❖ 数据元素可以是一个字符、一个数或一个记录，也可以是更复杂的信息。
- ❖ 线性表一：26个英文字母组成的字母表（A，B，C，...，Z）。表中的数据元素是单个字母字符。
- ❖ 线性表二：某学生五门课的成绩列表（76，87，88，80，92）。表中的数据元素是整数。

36个同学的成绩，如何存储？

❖线性表：某班级学生**信息表**。表中的数据元素是一个**记录**，包括姓名、学号、性别和年龄4个数据项。

姓 名	学 号	性 别	年 龄
王 琳	021108101	女	19
马小明	021108102	男	20
李晓俊	021108103	男	19
⋮	⋮	⋮	⋮

◆线性表中的元素可以是各种各样的，但**同一线性表**中的元素必定具有**相同特性**，即属同一数据对象。

❖ 一个线性表可记为：

$$(a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n), n \geq 0 \square$$

其中， n 为表的长度，当 $n=0$ 时，称为空表。

称 i 为 a_i 在线性表中的**位序**。

❖ 表中 a_{i-1} 领先于 a_i ，称 a_{i-1} 是 a_i 的**直接前驱**， a_{i+1} 是 a_i 的**直接后继**。

❖ 线性表的抽象数据类型定义：

线性表的抽象数据类型

ADT List {

数据对象: $D = \{a_i \mid a_i \in \text{ElemSet}, i=1,2,\dots,n, n \geq 0\}$

数据关系: $R = \{ \langle a_i, a_{i+1} \rangle \mid a_i, a_{i+1} \in D \}$

基本操作:

InitList(&L); 创建一个空的线性表

Destroy(&L); 销毁一个存在的线性表

ClearList(&L); 清空一个已经存在的线性表

ListEmpty (L) ; 判断线性表是否为空

ListLength (L)

GetList (L, i)

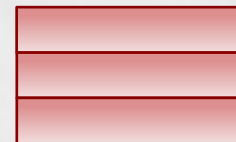
ListInsert (&L, i, e)

ListDelete(&L, i) }

线性表在计算机中如何存储？

❖ 顺序存储：顺序表

姓 名	学 号	性 别	年 龄
王 琳	021108101	女	19
马小明	021108102	男	20
李晓俊	021108103	男	19
⋮	⋮	⋮	⋮



❖ 链式存储：链表

2.2 线性表的顺序存储结构

- ❖ 线性表的顺序存储是指在内存中用**地址连续**的一块存储空间**依次存放**线性表的数据元素，用这种存储形式存储的线性表称其为**顺序表**。
- ❖ 假设**每个数据元素占d个存储单元**，且将 a_i 的存储地址表示为 $Loc(a_i)$ ，则有如下关系：

$$Loc(a_i) = Loc(a_1) + (i-1) * d$$

$Loc(a_1)$ 是线性表的第一个数据元素 a_1 的存储地址，通常称作线性表的**基地址**。

- ❖ 顺序表一般用**数组**表示。
- ❖ 顺序表的存储结构如下图所示，其中b为线性表的基地址。

存储地址	内存状态	元素位序
b	a_1	1
$b+d$	a_2	2
\vdots	\vdots	\vdots
$b+(i-1)d$	a_i	i
\vdots	\vdots	\vdots
$b+(n-1)d$	a_n	n
$b+nd$		空闲
\vdots		
$b+(\text{MAXSIZE}-1)d$		

每个数据元素
占 d 个字节

适合随机存取

2.2.1 顺序表的类型定义

```
#define MAXSIZE 100 // 顺序表的容量
```

typedef?

```
typedef struct
```

```
{
```

```
    ElemType data[MAXSIZE]; // 存放顺序表的元素
```

```
    int len;                // 顺序表的实际长度
```

```
} SqList;
```

数据类型因
应用而异

- ◆ MAXSIZE为顺序表的容量，表示线性表可能达到的最大长度
- ◆ len表示顺序表当前的长度
- ◆ ElemType表示存储数据的类型

举例

- ❖ 一个同学5门课程的成绩（整数，最多10门课）

SqList x;

- ❖ 36名同学，每名同学5门课成绩

```
#define MAXSIZE 50 // 顺序表的容量
typedef struct
{
    ElemType data[MAXSIZE][5]; // 存放顺序表的元素
    int len; // 顺序表的实际长度
} SqList;
```

SqList x;

2.2.2 线性表基本运算在顺序表上的实现

- ❖ C语言中数组的下标从”0”开始，因此，若L是Sqlist类型的顺序表，则表中第i个元素是L.data[i-1]。 □

1. 初始化线性表运算

```
void InitList(Sqlist &sq)
{
    sq.len=0;
}
```

2.求线性表长度运算

```
int ListLength (Sqlist sq)
{
    return(sq.len);
}
```

3.求线性表中第i个元素运算

```
ElemType GetElem(SqList sq, int i)
{
    if (i<1 || i>sq.len)    // i 值不合法
        return 0;
    else
        return(sq.data[i-1]);
}
```

4.按值查找运算(找出值等于e的数据元素，返回他的位置)

```
int LocateElem(SqList sq, ElemType e)
```

```
{
```

```
    int i=0;
```

```
    while ( (i<sq.len) && (sq.data[i]!=e) ) i++;
```

```
        // 将表中元素逐个与e比较
```

```
    if (i>=sq.len)
```

```
        return 0;
```

```
    else
```

```
        return i+1;
```

```
}
```


5. 插入一个数据元素e，使之成为第i个元素

```
int ListInsert(SqList &sq, int i, ElemType e)
```

```
{
```

```
    int j;
```

```
    if (i<1 || i>sq.len+1) return 0; // i 值不合法
```

```
    for (j=sq.len; j>=i; j--)    // 插入位置及之后的元素右移
```

```
        sq.data[j] = sq.data[j-1];
```

```
    sq.data[i-1] = e;    // 插入e
```

```
    sq.len++;    // 表长增1
```

```
    return 1;
```

```
}
```

i

data[0] data[1].....data[i-1] data[i]data[n-1]

data[0] data[1]..... data[i]data[i+1]data[n]

6. 删除第i个数据元素运算

```
int ListDelete (SqList &sq ,int i)
{
    int j;

    if (i<1 || i>sq.len) return 0; // i 值不合法

    for (j=i; j<sq.len; j++)      // 被删除元素之后的元素左移

        sq.data[j-1]= sq.data[j];

    sq.len--;                    // 表长减1

    return 1;
}
```

i

data[0] data[1].....data[i-1] data[i]data[n-1]

data[0] data[1].....data[i]data[i+1]data[n-1]

2.2.3 顺序表操作的算法分析

1. 插入

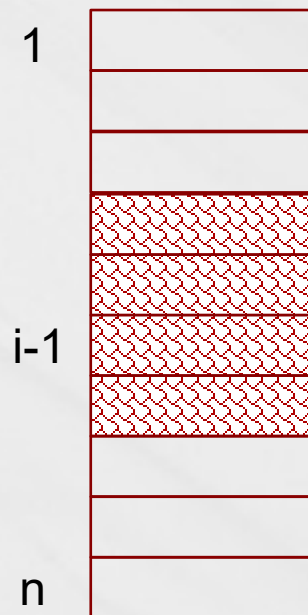
- ❖ 假设 p_i 是在第 i 个位置插入一个元素的概率，则在长度为 n 的线性表中插入一个元素时所需**移动元素**的平均次数为(有 $n+1$ 个插入位置)：

$$\sum_{i=1}^{n+1} p_i(n-i+1)$$

$$i=1,2,\dots,n,n+1$$

$$p_i=1/(n+1)$$

◆ 插入算法的平均时间复杂度为 $O(n)$



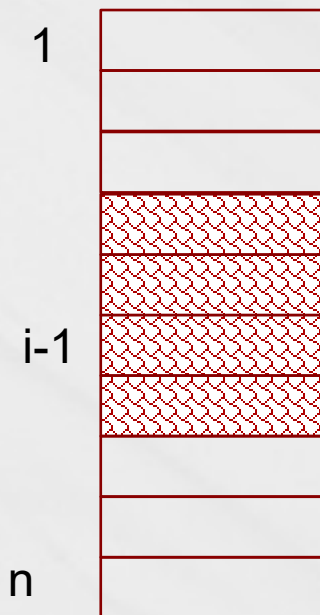
2. 删除

- ❖ 假设 q_i 是在第 i 个位置删除一个元素的概率，则在长度为 n 的线性表中删除一个元素时所需移动元素的平均次数为(有 n 个元素可以删除)：

$$\sum_{i=1}^n q_i(n-i)$$

$$q_i=1/n$$

- ◆ 删除算法的平均时间复杂度为 $O(n)$ 。



2.2.4 顺序表的应用举例

- ❖ 【例1】编写一算法，从顺序表中删除自第 $i+1$ 个元素开始的 k 个元素。

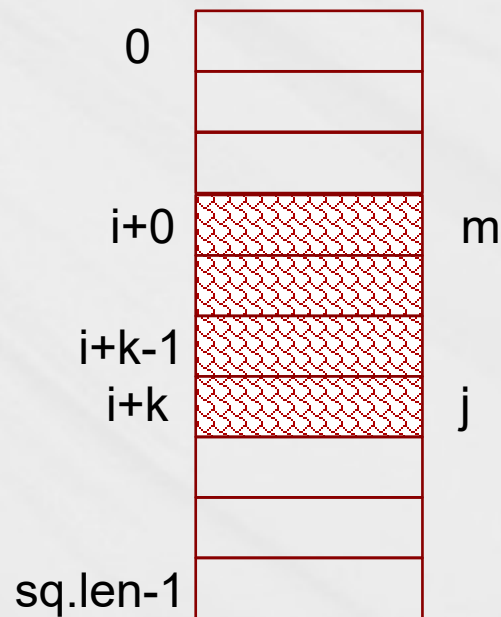
算法思路：为保持顺序表的逻辑特性，需将 $i+k \sim n$ 位置的所有元素依次前移 k 个位置。算法如下：

```
int deleteK(Sqlist &sq, int i, int k)
{
    if (i<1 || k<1 || i+k-1>sq.len) return 0;

    for (int m=i, j=i+k; j<=sq.len-1; j++,m++)
        sq.data[m]=sq.data[j];

    sq.len-=k;

    return 1;
} // delete K
```



【例2】已知有两个按元素值**升序排列**的顺序表La和Lb，设计一个算法将表La和表Lb的全部元素**归并**为一个按元素值**升序**排列的顺序表Lc。

❖ 算法思路：扫描顺序表La，顺序表Lb。

(1) 当表La和表Lb都未扫描完时，比较两者的当前元素，将较小者插入表Lc的表尾

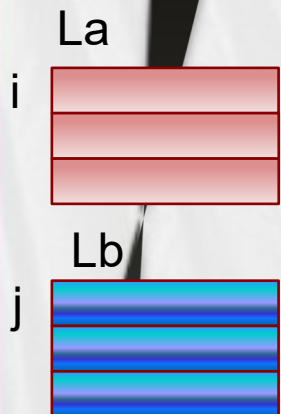
(2) 若两者的当前元素相等，则将这两个元素依次插入表Lc的表尾

(3) 最后，**将尚为扫描完的顺序表的余下部分元素依次插入表Lc的表尾**。算法如下：

❖ void MergeList_Sq(SqList La, SqList Lb, SqList &Lc)

{

int i=0, j=0, k=0;



```

while (i < La.len && j < Lb.len)    // 表La和表Lb都未扫描完时
{
    if (La.data[i] < Lb.data[j])
    {
        Lc.data[k] = La.data[i];    i++;    k++;
    }
    else if (La.data[i] > Lb.data[j])
    {
        Lc.data[k] = Lb.data[j];    j++;    k++;
    }
    else {
        Lc.data[k] = La.data[i];    i++;    k++;
        Lc.data[k] = Lb.data[j];    j++;    k++;
    }
}

while (i < La.len) {
    Lc.data[k] = La.data[i];    i++;    k++;
}

while (j < Lb.len) {
    Lc.data[k] = Lb.data[j];    j++;    k++;
}

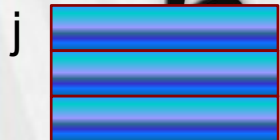
Lc.len = k;    // 置表Lc的实际长度
} // MergeList_Sq

```

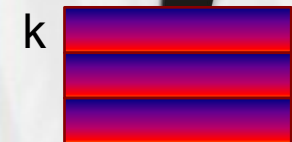
La



Lb



Lc



【例3】已知线性表 (a_1, a_2, \dots, a_n) 按顺序存储，且每个元素都是互不相等的整数。设计算法，把所有是奇数的数据元素移到所有偶数数据元素的前边，并给出完整的程序。

- ❖ 解题思路：先定义顺序表的类型，并根据题意将 ElemType 设为 int 型；然后设计一个算法 move 来把所有奇数移到所有的偶数前边；最后在主函数中调用实现移动算法的函数。
- ❖ 算法 move 的思路： $i=0$ ， $j=\text{sq.len}-1$ ，从左向右找到偶数 $\text{sq.data}[i]$ ，从右向左找到奇数 $\text{sq.data}[j]$ ，将两者交换； $i++$ ， $j++$ ，重复此过程直到 i 大于 j 为止。
- ❖ 完整的程序参见教材！

$i=0$

2 6 33 5 6 7 8 9 23 10 11

$j=\text{sq.len}-1$

11

2.3 线性表的链式存储结构

- ❖ **线性表的顺序存储结构的特点**：逻辑关系上相邻的数据元素在物理位置上也相邻，即必须使用**一块连续的存储单元**存储一个线性表。
- ❖ 线性表的链式存储结构（简称**链表**）是指用一些**任意的存储单元**来存放线性表的各个数据元素（每个数据元素称为**结点**），
- ❖ **线性表的链式存储结构特点**：这些存储单元即可以是连续的，也可以是不连续的，甚至是零散分布在内存中的任意位置上的。
- ❖ 种类：单链表，双向链表，循环链表

2.3.1 单链表

❖ 单链表的结点结构如下：



- ❖ 其中：data域是数据域，用来存放结点的值。next是指针域（亦称链域），用来存放结点的直接后继的地址（或位置）。
- ❖ 由于链表的每一个结点只有一个链域，故将这种链表称为单链表。

❖ 单链表的结点类型定义如下：

```
typedef struct LNode
{
    ElemType data; // 数据域
    struct LNode *next; // 指针
} LNode, *LinkList;
```

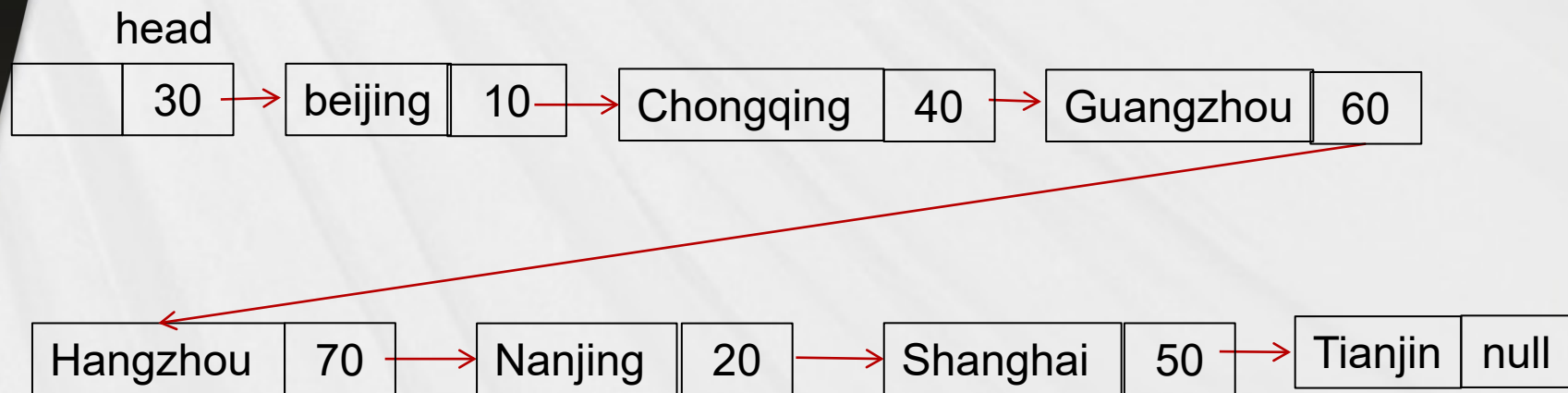
为了便于一些运算的实现，在单链表的首结点之前附设一个结点，称之为“头结点”。

有了上述的类型定义，就可以用LNode 或 LinkList来声明单链表的表头指针：

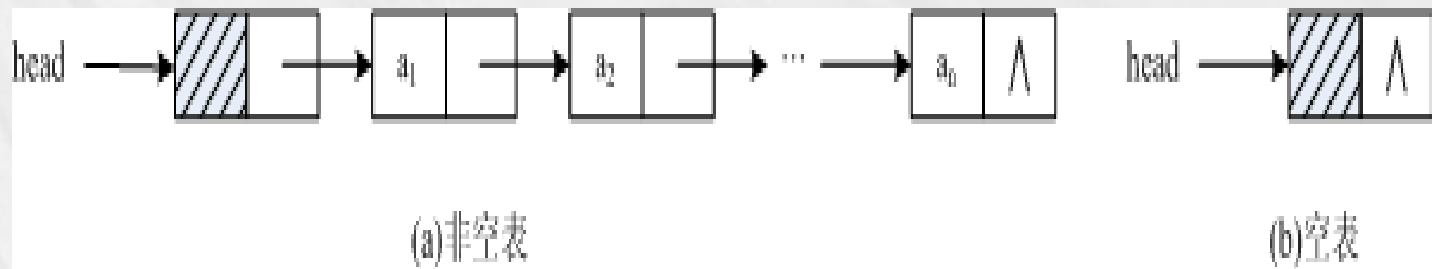
```
LNode *head;           或           LinkList head;
```



	存储地址	数据域	指针域
head <div><div></div><div>30</div></div> →	10	Chongqing	40
	20	Shanghai	50
	30	beijing	10
	40	Guangzhou	60
	50	Tianjin	Null
	60	Hangzhou	70
	70	Nanjing	20



❖ 带头结点的单链表 和空链表：



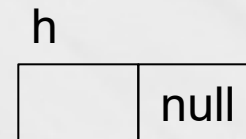
◆在单链表中，任何两个元素的存储位置之间没有固定的联系，每个数据元素的存储位置都包含在其**直接前驱结点的指针（链域）**中。因此，在单链表中，要想取得第*i*个数据元素，必须从头指针出发寻找，即只能**顺序存取**。

1. 线性表基本运算在单链表上的实现

LinkList h -----指向单链表头节点的指针

(1) 初始化线性表----仅有头节点

```
void InitList(LinkList h)
{
    h=(LNode *)malloc(sizeof(LNode)); //创建头结点
    h->next=NULL;
}
```



(2) 求线性表长度

```
int ListLength(LinkList h)
{
    int i=0;
    LNode *p=h;
    while (p->next!=NULL)
        //当p指向最后一个数据结点时，循环停止
    {
        i++; //增1
        p=p->next; //指针p沿着next域移动一次
    }
    return i;
}
```



(3) 找到线性表中第i个数据元素-----返回何值?

LNode *GetElem(LinkList h, int i)

```
{int j=1;
```

```
LNode *p=h->next; //p指向第一个数据节点
```

```
if (i<1 || i> ListLength(h)) return NULL; //i值不合法
```

```
while (j<i) //从第1个结点开始, 查找第i个结点
```

```
{
```

```
    p=p->next; j++;
```

```
}
```

```
return p; // 返回第i个结点的指针
```

```
}
```



◆本算法的时间复杂度为 $O(n)$ 。

(4) 按值e查找节点-----返回何值?

```
LNNode *LocateElem(LinkList h, ElemType e)
```

```
{
```

```
    LNode *p=h->next;
```

```
    while (p!=NULL && p->data!=e)
```

//从第1个结点开始, 查找data域为e的结点

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

```
    return p;
```

```
}
```



本算法的时间复杂度为 $O(n)$

(5) 插入值为e的结点到位置i-----返回何值?

```
int ListInsert(LinkList h, ElemType e, int i)
{
    int j=0;
    LNode *p=h, *s;

    if (i<1 || i> ListLength(h)+1) return 0; //i值不合法

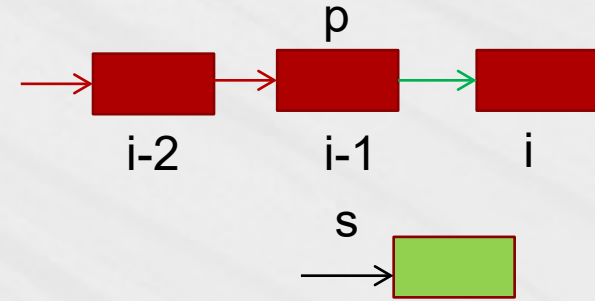
    while (j<i-1) { p=p->next; j++; } //从头结点开始, 查找第i-1个结点

    s=( LNode *)malloc(sizeof(LNode)); //创建新结点

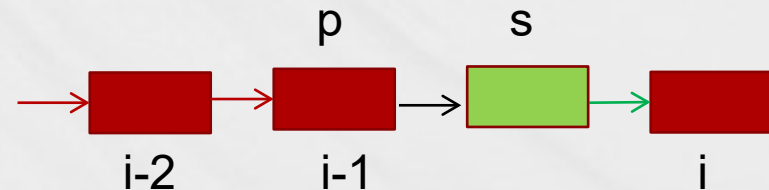
    s->data=e;

    s->next=p->next; p->next=s; //插入链表中

    return 1;
}
```



交换顺序
可以吗?



本算法的时间复杂度为 $O(n)$:找到第 $i-1$ 个节点所需要的时间

(6) 删除第i个结点

```
int ListDelete(LinkList h, int i)
{
    int j=0;

    LNode *p=h, *q;

    if ( i<1 || i> ListLength(h)) return 0;  //i值不合法

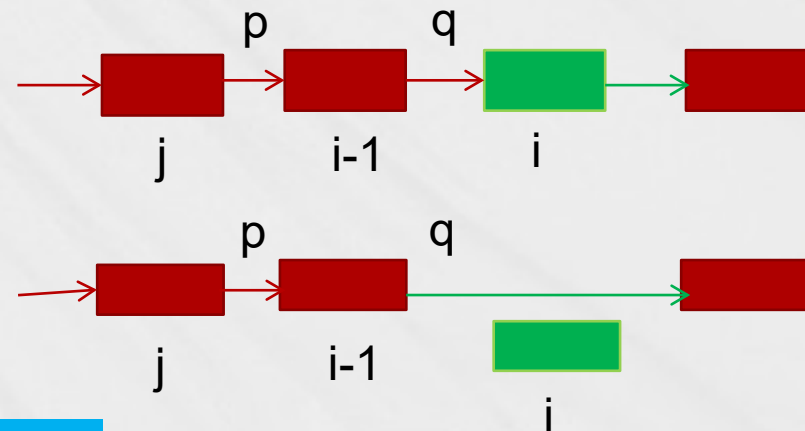
    while (j<i-1) {p=p->next; j++;}  //从头结点开始, 查找第i-1个结点

    q=p->next;  //删除并释放结点

    p->next=q->next;

    free(q);

    return 1;
}
```



本算法的时间复杂度为 $O(n)$

(7) 输出链表各个数据元素的值

```
void ListOutput(LinkList h)
{
    LNode *p=h->next;
    while (p!=NULL)
    {
        printf("%5d ", p->data); //输出结点的data域
        p=p->next;
    }
}
```



本算法的时间复杂度为 $O(n)$

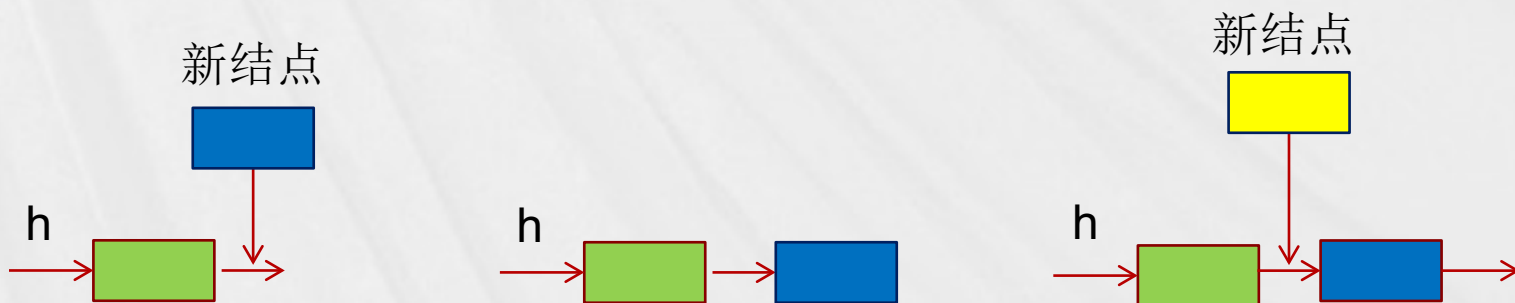
2. 建立单链表：头插法和尾插法

(1) 头插法建表

算法思路：从一个空表开始，读取数据，申请新结点，将读取的数据存放到新结点的数据域中，然后将新结点插入到当前链表的表头节点后面，如此重复，直到读入结束标志为止

原始数据存在数组中,共 n 个数据,存储在数组 $a[]$ 中

❖ 算法如下：




```
LNode *CreateListF(LinkList h, ElemType a[ ], int n)
```

```
{
```

```
LNode *s; int i;
```

```
h=( LNode *)malloc(sizeof(LNode)); // 创建头结点：空表
```

```
h->next=NULL;
```

```
for (i=0; i<n; i++)
```

```
{ s=( LNode *)malloc(sizeof(LNode)); // 创建新结点
```

```
s->data=a[i];
```

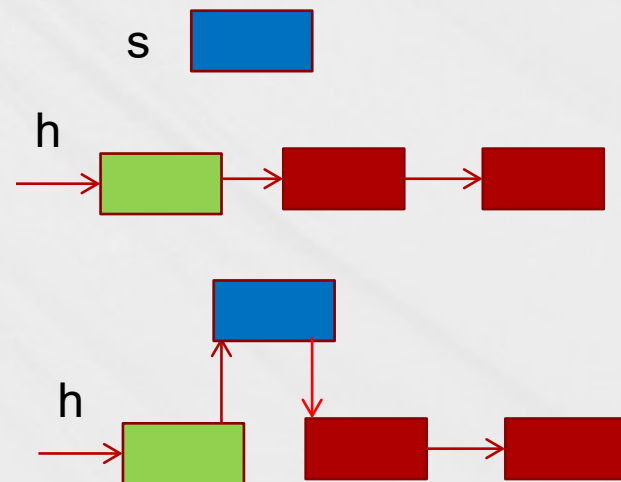
```
s->next=h->next; // 将新结点插入到头结点之后
```

```
h->next=s;
```

```
}
```

```
return(h);
```

```
}// CreateListF
```



(2) 尾插法建表

算法思路：将新结点插入到当前链表的表尾上，为此必须增加一个**尾指针r**，使其始终指向当前链表的尾结点。n个数据存储在数组a[]中。

算法如下：

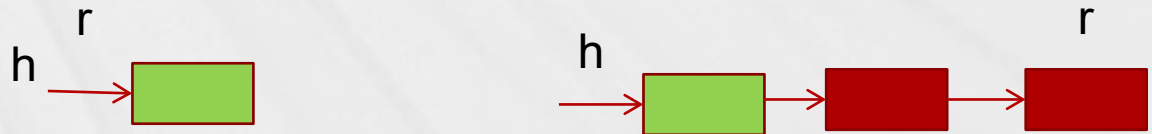
```
LNode * CreateListR(LinkList h, ElemType a[ ], int n)
```

```
{  LNode *s, *r; int i;
```

```
  h=( LNode *)malloc(sizeof(LNode)); // 创建头结点
```

```
  h->next=NULL;  r=h;
```

```
  // r 始终指向尾结点，开始时指向头结点
```



```
for (i=0;i<n;i++)
```

```
{ s=( LNode *)malloc(sizeof(LNode)); // 创建新结点
```

```
s->data=a[i];
```

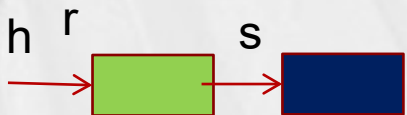
```
r->next=s;  r=s; // 将新结点插入到尾结点之后
```

```
}
```

```
r->next=NULL; // 将尾结点的next域置为空
```

```
}// CreateListR
```

◆头插法建表和尾插法建表算法的时间复杂度都是 $O(n)$ 。



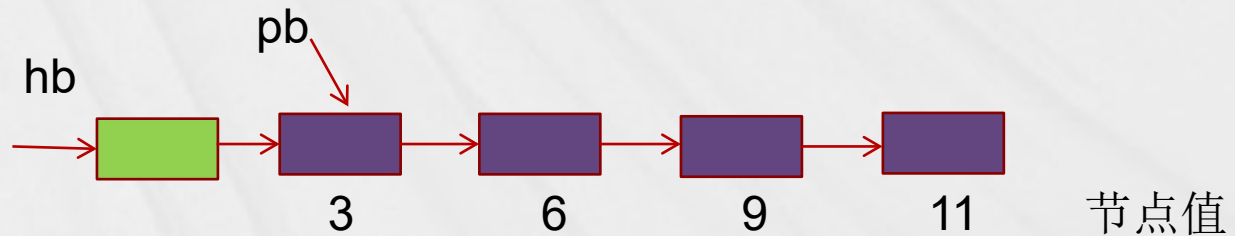
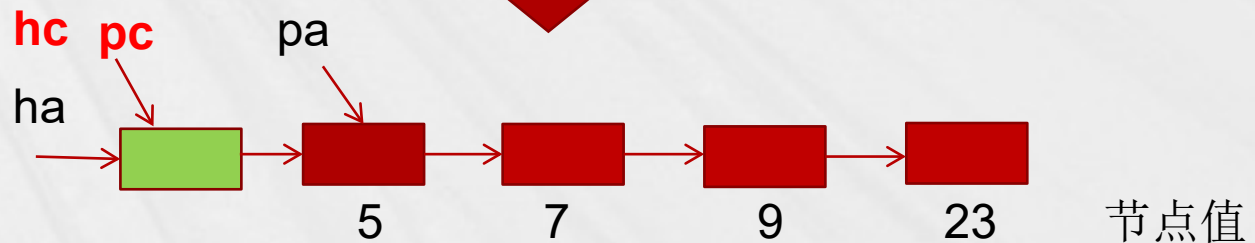
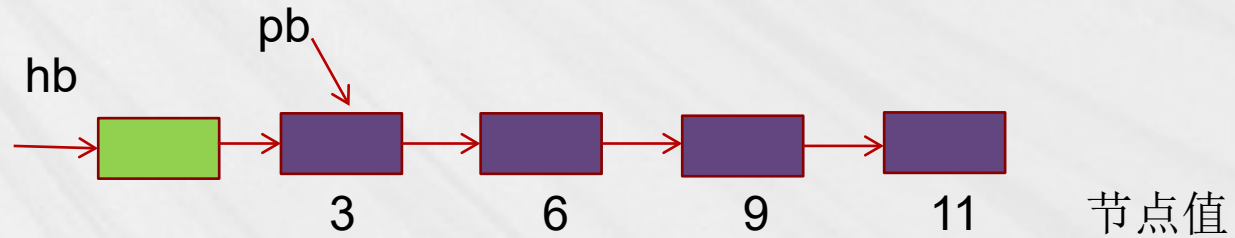
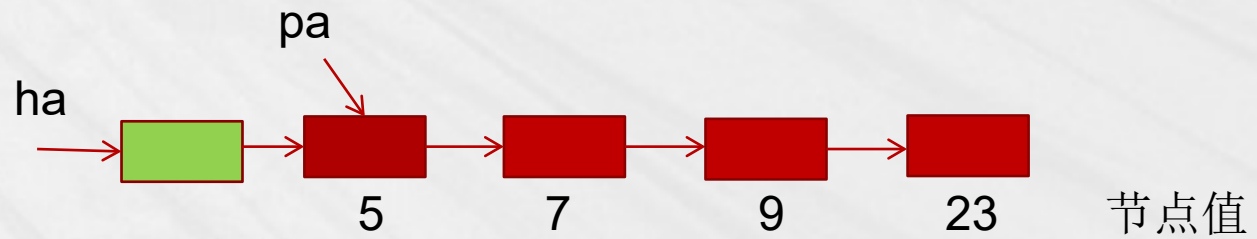
3. 单链表的应用举例

【例1】ha和hb: 非递减有序单链表的表头指针，试设计一个算法，将这两个有序链表合并成一个非递减有序的单链表hc。要求结果链表仍使用原来两个链表的空间。表中允许有重复的数据。

算法思路：

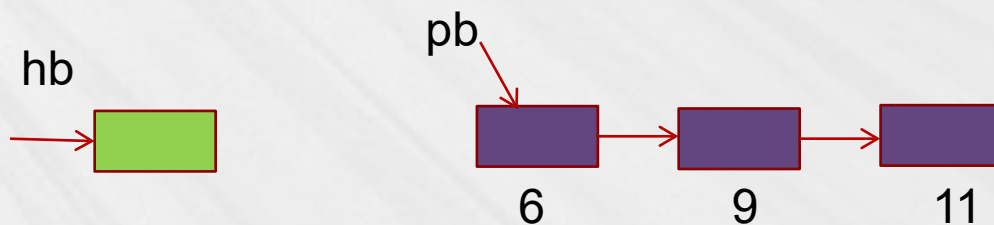
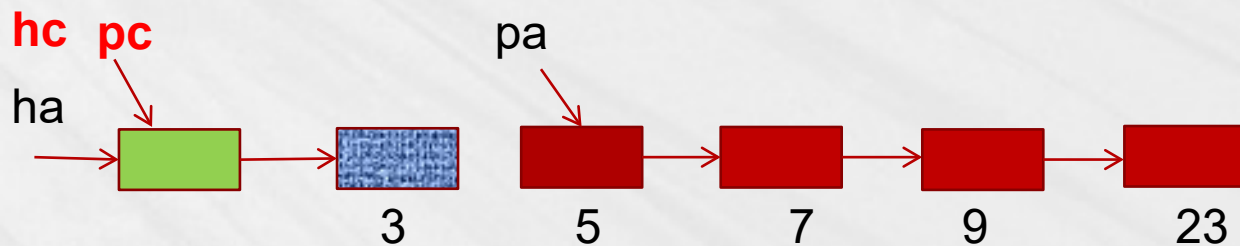
- ①3个表：设立3个指针pa、pb和pc，其中pa和pb分别指向ha和hb表中当前待比较的结点，而pc指向hc表的最后一个结点。
- ②比较pa->data和pb->data，将较小者插入hc的表尾，即链到pc所指结点之后。若pa->data和pb->data相等，则将两个结点均链到pc所指结点之后。
- ③如此反复，直到有一个表的数据元素已归并完（pa或pb为空）为止，再将另一个表的剩余段链接到pc所指结点之后。

用ha的头结点作为hc的头结点， $hc=ha$ ， $pc=ha$

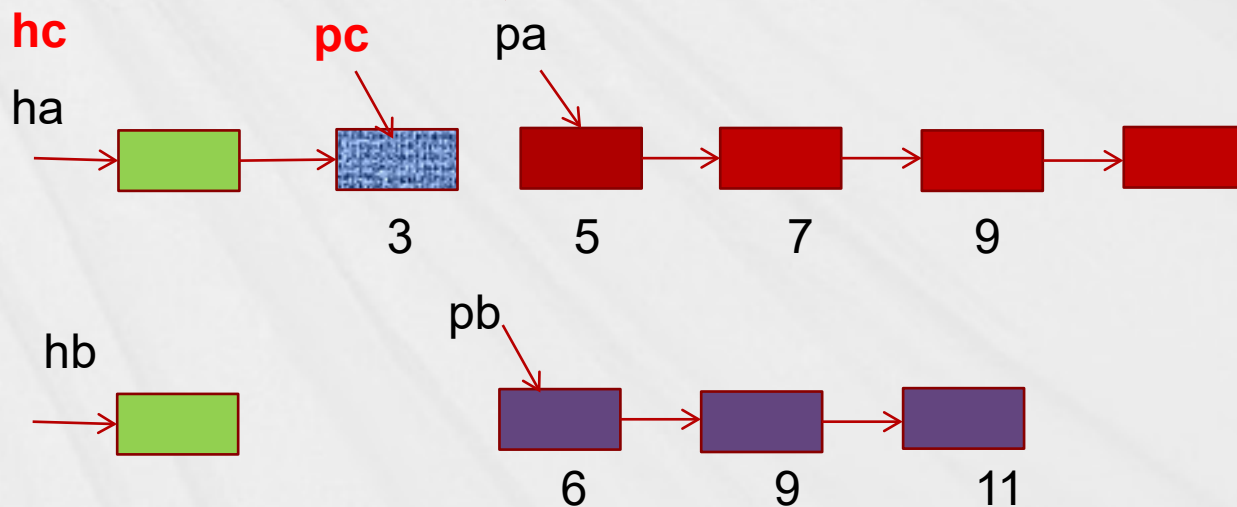


第一步：
确定hc表头

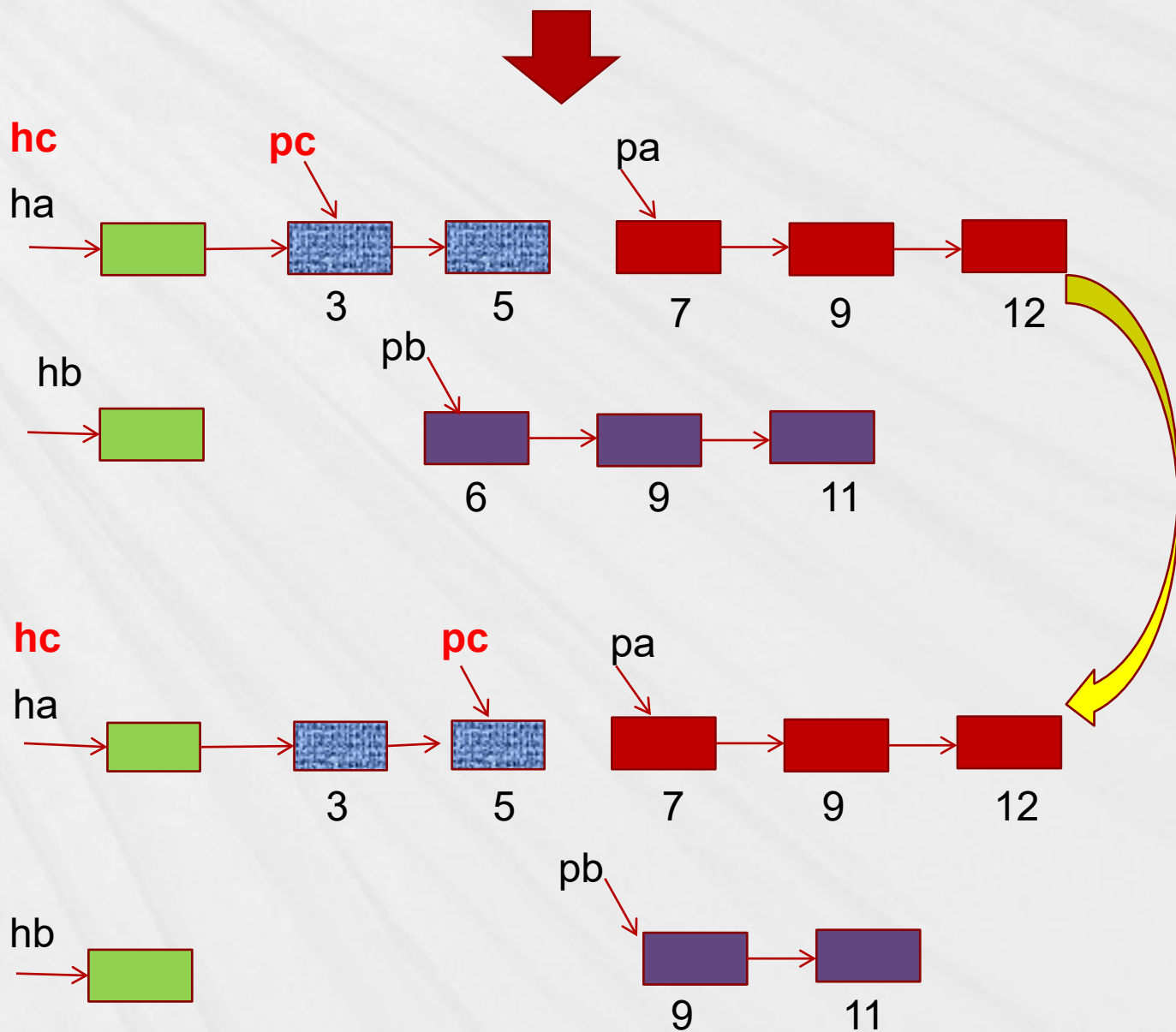
第二步：
插入一个
节点到hc



第三步：
移动pc到
c表尾



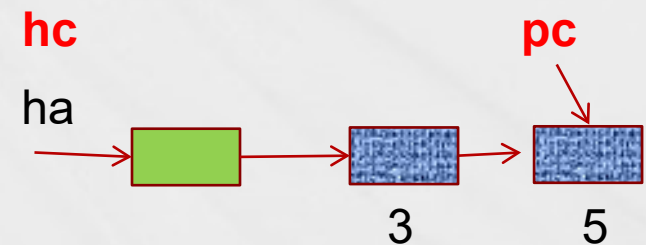
第四步




❖ 具体算法:

// 用ha的头结点作为hc的头结点, pc始终指向hc的表尾结点


```
void MergeList_L(LinkList ha, LinkList hb, LinkList hc)
{
    LNode *pa, *pb, *pc;
    pa=ha->next; pb=hb->next;
    hc=pc=ha;
    while(pa&&pb){
        if(pa->data<pb->data) {pc->next=pa; pc=pa; pa=pa->next;}
        else if(pa->data>pb->data) {pc->next=pb; pc=pb; pb=pb->next;}
        else{ pc->next=pa; pc=pa; pa=pa->next;
              pc->next=pb; pc=pb; pb=pb->next;}
    }
}
```





```
pc->next=pa ? pa : pb;    // 插入剩余段  
free(hb);                // 释放hb的头结点  
} // MergeList_L
```

- ❖ 本算法的**基本操作**是结点**数据的比较和结点的链入**，在最坏情况下，对每个结点均需进行上述操作，因此，若表ha和表hb的长度分别是m和n，则本算法的**时间复杂度为 $O(m+n)$** 。

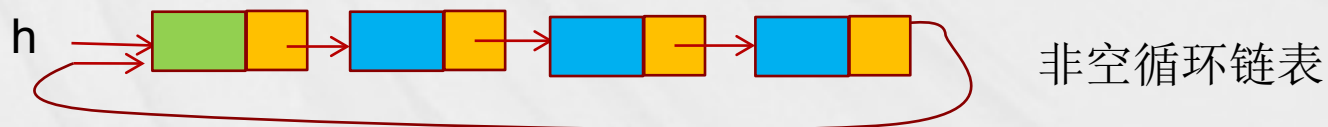


【例2】设计算法，根据输入的学生人数、姓名和成绩建立一个单链表，并累计其中成绩优秀的人数。要求给出完整的程序。

- ❖ 解题思路：先定义单链表结点的类型，并根据题意将ElemType设为int型，char name[12]；
- ❖ 然后设计一个算法create，用于输入学生人数，根据人数输入姓名和成绩，并建立相应的单链表；
- ❖ 设计一个算法count，用于计算成绩优秀的人数；
- ❖ 最后在主函数中调用实现上述两个算法的函数。
- ❖ 完整的程序参见教材，自学！

2.3.2 单循环链表

- ❖ **循环链表的特点**：表中最后一个结点的指针域指向头结点，整个链表形成一个环。
- ❖ 单循环链表的操作和单链表基本一致，差别仅在于算法中的循环条件不是 p 或 $p \rightarrow next$ 是否为空，而是它们是否等于头指针。



例如，求线性表的长度运算在单循环链表上的实现算法如下：

```
int ListLength(LinkList h)
```

```
{
```

```
    int i=0;
```

```
    LNode *p=h;
```

```
    while (p->next!=h) //当p指向最后一个数据结点时，循环停止
```

```
    {
```

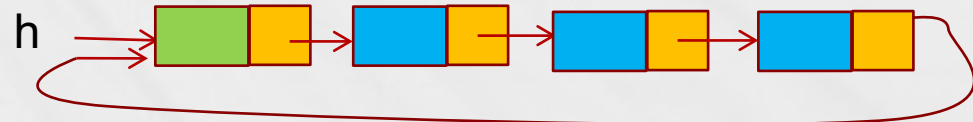
```
        p=p->next; i++; //指针p沿着next域移动一次，i值增1
```

```
    }
```

```
    return i;
```

```
} // ListLength
```

p->next!=NULL

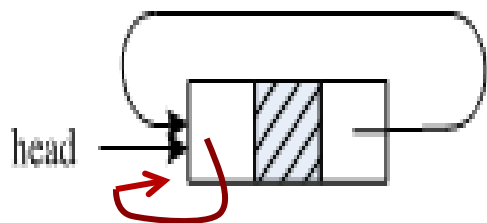


2.3.3 双向链表

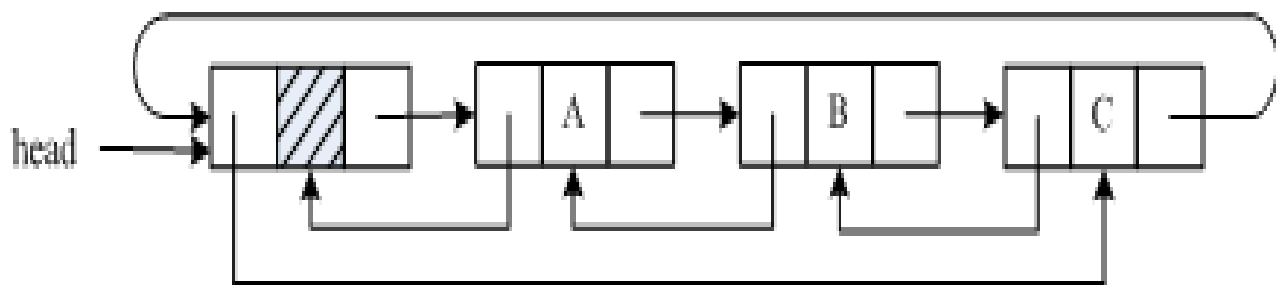
在双向链表的结点中有**两个指针域**，其一指向**直接后继**，另一指向**直接前驱**。



(a)



(b)



(c)

(a)双向链表的结点结构

(b) 空的双向循环链表

(c) 非空的双向循环链表

双向链表节点的数据结构：

```
typedef  struct DuNode
{
    ElemType data;
    struct DuNode *prior;
    struct DuNode *next;
} DuLNode, *DuLinkList;
```


1. 删除第*i*个节点

时间复杂度如何?

```
int ListDelete_DuL(DuLinkList Dh, int i)
{
    int j=0;

    DuLNode *p=Dh,*q;

    if (i<1 || i> ListLength_DuL(Dh)) return 0;  //i值不合法

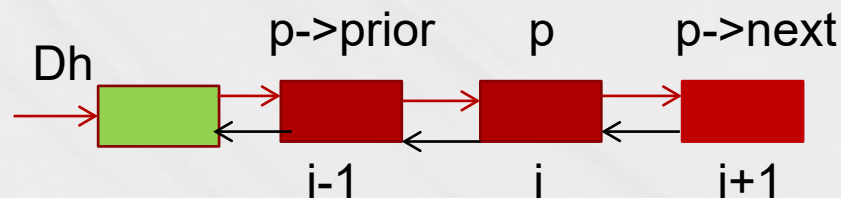
    while (j<i) {p=p->next; j++;}  //从头结点开始, 查找第i个结点p

    p->prior->next=p->next;  //删除并释放结点

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

    free(p);

    return 1;
} // ListDelete_DuL
```



2. 插入值 (第i个节点, 值为e)

```
int ListInsert_DuL(DuLinkList Dh, ElemType e, int i)

{ int j=0;

  LNode *p=Dh, *s;

  if (i<1 || i> ListLength_DuL(Dh)+1) return 0; //i值不合法

  while (j<i) { p=p->next; j++; } //从头结点开始, 查找第i个结点p

  s==( DuLNode *)malloc(sizeof(DuLNode)); //创建新结点

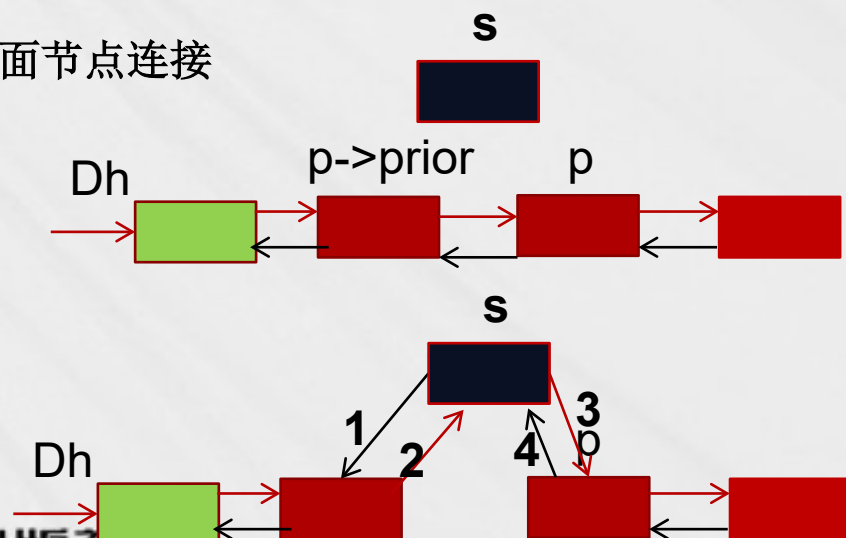
  s->data=e;

  s->prior=p->prior; p->prior->next=s; //s与前面节点连接

  s->next=p; p->prior=s;; //s与后面节点连接

  return 1;

} // ListInsert_DuL
```



本章小结

- ❖ 线性表是一种典型的线性结构。线性表中的数据元素可以是各种各样的，但同一线性表中的数据元素必**具有相同的特性**。
- ❖ 顺序表是以“物理位置相邻”来表示线性表中数据元素之间的逻辑关系的，通常用数组来描述。
- ❖ 链表是通过每个结点的链域将线性表的各个结点按其逻辑次序链接在一起的。
- ❖ 单链表： 一个指针
- ❖ 单循环链表的特点是表中最后一个结点的指针域指向头结点，整个链表形成一个环。
- ❖ 双向链表的结点中有两个分别指向直接后继和指向直接前驱的指针域，在双向链表中寻查结点的前驱和后继都很方便。

本章习题

1. 对于表长为 n 的顺序表，在任何位置上插入或删除一个元素的概率相等时，插入一个元素所需要移动的元素平均个数为多少？删除一个元素所需要移动的元素平均个数为多少？
2. 设线性表 A 采用顺序存储且元素按值递增有序排列。试编一算法，将 x 插入到线性表的适当位置上，并保持线性表的有序性。分析算法的时间复杂度。
3. 设线性表 A 采用链式存储且元素按值递增有序排列。试编一算法，将 x 插入到线性表的适当位置上，并保持线性表的有序性。分析算法的时间复杂度。
4. 已知 La 是带头结点的单链表，编写一算法，从表 La 中删除自第 i 个元素起，共 m 个元素。
5. 分别画出顺序表、单链表、双链表和单循环链表的结构图。