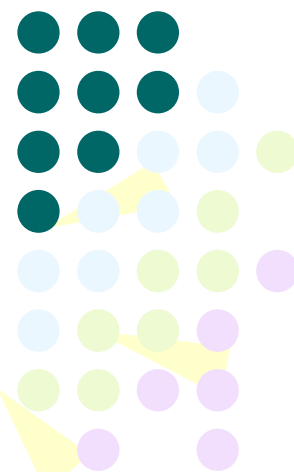


链表

吴清锋
2023年春

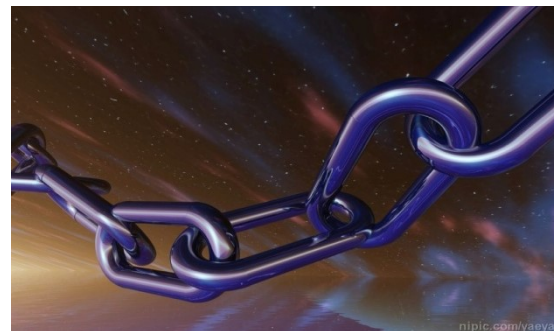




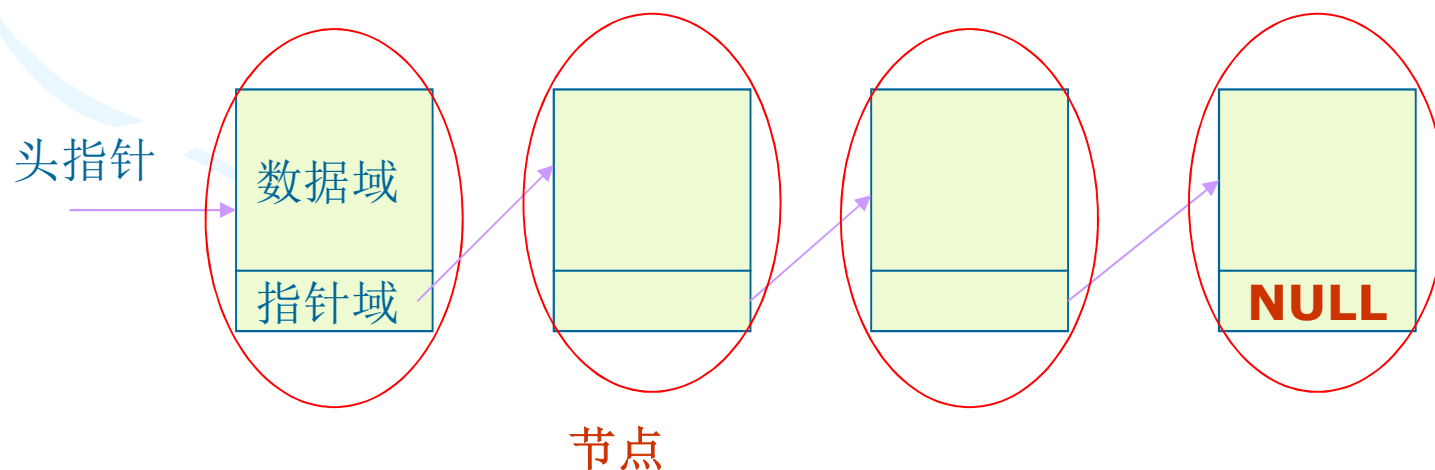
链式存储

- 链表概述
- 若干微操作
- 单链表基本运算及其实现
- 单链表应用

什么是链表



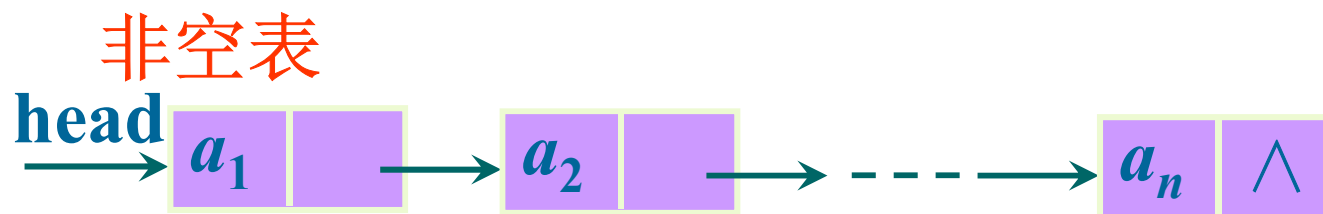
- 链表是由若干个节点组成。
- 节点通常是结构体变量或对象, 除包含有数据域外, 设置一个或多个指针域, 用以指向其后继结点
- 每个链表必定存在一个头指针, 指向链表的第一个节点



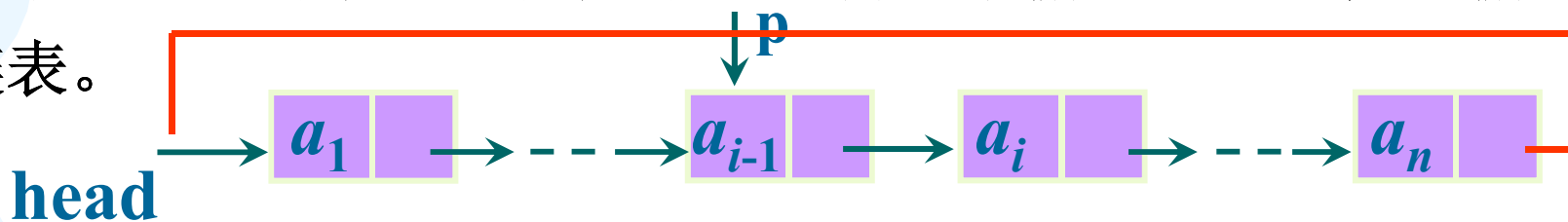
链表分类

空表 $\text{head} = \text{NULL}$

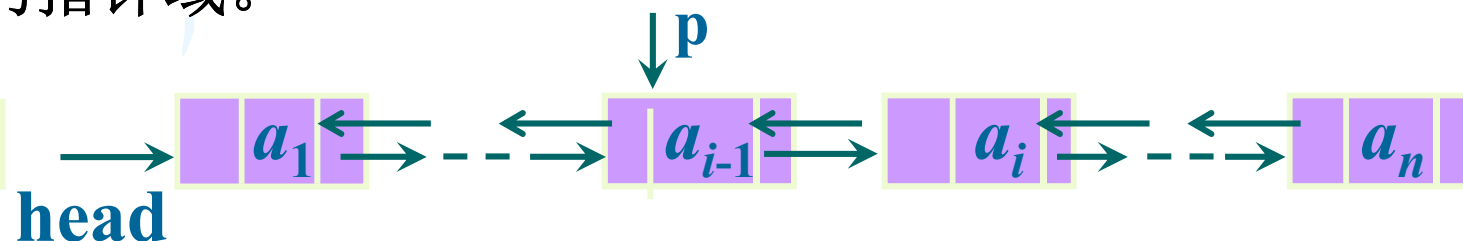
- 单链表



- 单循环链表：将单链表的首尾相接，将终端结点的指针域由空指针改为指向头结点，构成单循环链表，简称循环链表。



- 双链表：在单链表的每个结点中再设置一个指向其前驱结点的指针域。





链式存储

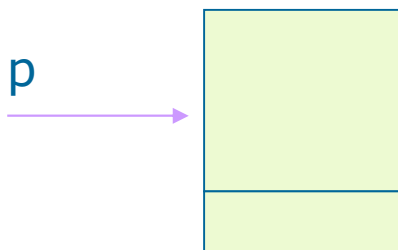
- 链表概述
- 若干微操作
- 单链表基本运算及其实现
- 单链表应用

链表微操作

- 生成一个节点
- 让一个指针指向存在的节点
- 让节点的指针指向另一个节点
- 让指向节点的指针指向下一个节点
- 判断是否为空

生成一个节点

- 用**new**分配一个节点，例如
p=new StudentNode;

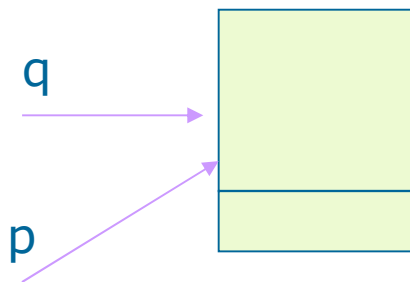


用**delete**释放一个节点，例如
delete p;

让一个指针指向存在的节点

假设 q 指向一个已存在的节点
执行语句：

$p=q;$

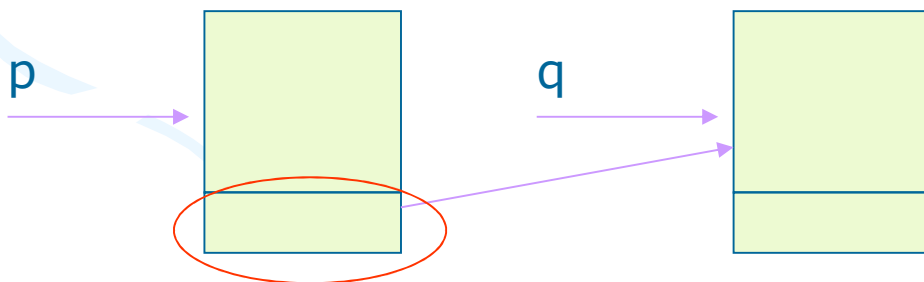


让节点的指针指向另一个节点

```
p=new StudentNode;
```

```
q=new StudentNode;
```

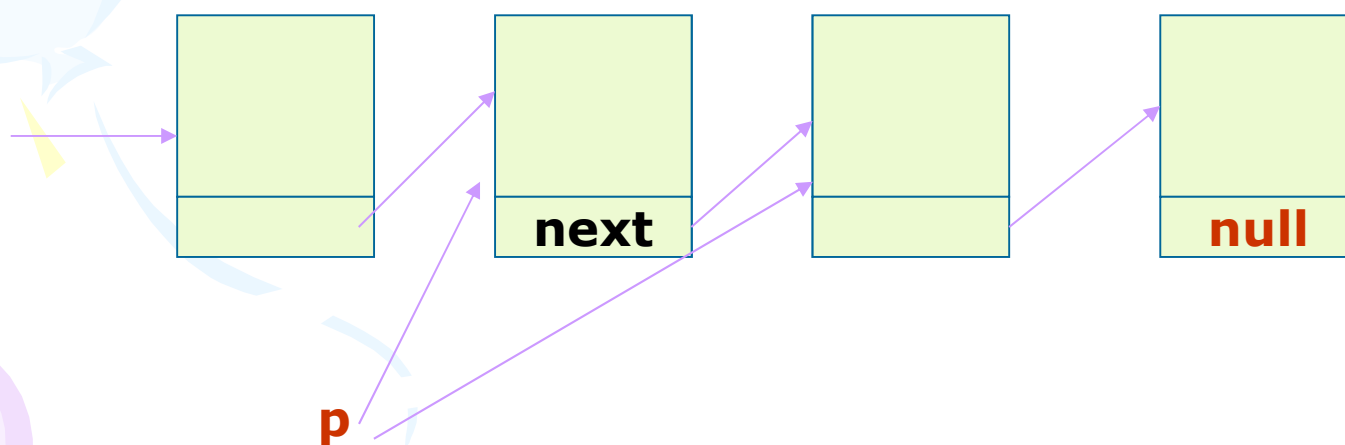
```
p->next=q
```



让指向节点的指针指向下一个节点

假设**p**指向链表中某一节点，执行语句

p=p->next;



A decorative graphic on the left side of the slide featuring three balloons: a green one at the top, a blue one in the middle, and a purple one at the bottom. Each balloon has a string and several small yellow triangular flags attached to it.

判断是否为空

1. 通过判断头指针是否为0来判断链表是否为空

`if(head==NULL)...`

2. 判断节点后面是否有节点

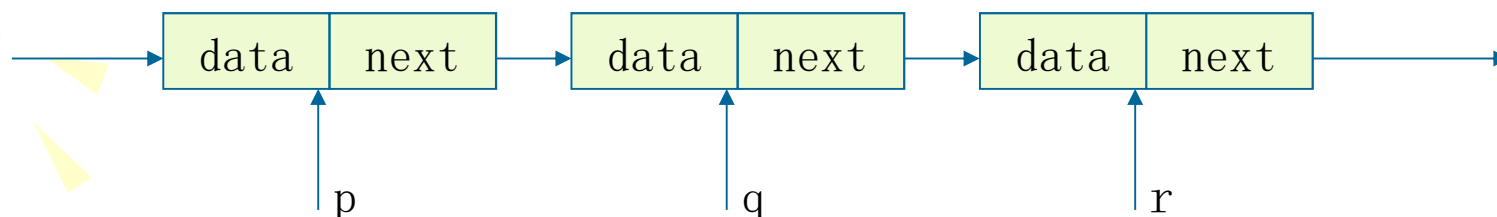
`if(r->next!=NULL) ...`



思考与练习

- 链表不具有的特点是_____。
 - A. 元素的存储地址可以不连续
 - B. 存储空间根据需要动态开辟，不会溢出
 - C. 可以直接随机访问元素
 - D. 插入和删除元素的时间开销与位置无关

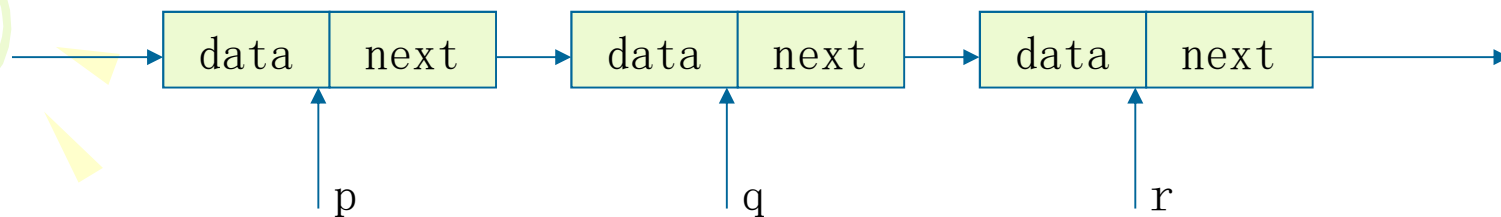
思考与练习



将上述链表当中q和r所指的节点交换位置，同时保持链表的连续，则下列语句当中无法胜任的是_____。

- A. `q->next=r->next; p->next=r; r->next=q;`
- B. `p->next=r; q->next=r->next; r->next=q;`
- C. `q->next=r->next; r->next=q; p->next=r;`
- D. `r->next=q; p->next=r; q->next=r->next;`

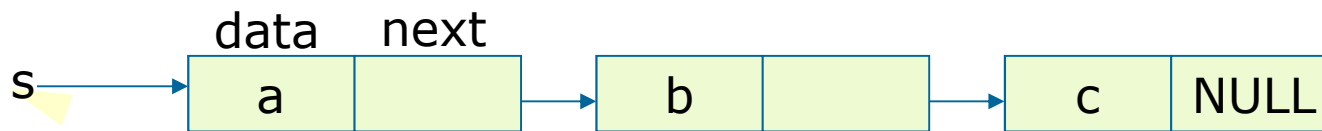
思考与练习



将上述链表当中q所指的节点删除，同时保持链表的连续，则下列语句当中无法胜任的是_____。

- A. $p \rightarrow next = q \rightarrow next;$
- B. $p \rightarrow next = p \rightarrow next \rightarrow next;$
- C. $p \rightarrow next = r;$
- D. $p = q \rightarrow next$

思考与练习



设有如上链表， s 、 p 、 q 为正确定义的指针。现有如下代码：

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

```
while(p->next) p=p->next;
```

```
p->next=q; q->next=NULL;
```

则该程序段的功能是_____。

- A) 首结点成为尾结点
- B) 尾结点成为首结点
- C) 删除首结点
- D) 删除尾结点



链式存储

- 链表概述
- 若干微操作
- 单链表基本运算及其实现
- 单链表应用

单链表类型声明

在单链表中,假定每个结点类型用**LinkedList**表示,它应包括存储元素的数据域,这里用**data**表示,其类型用通用类型标识符**ElemType**表示,还包括存储后继元素位置的指针域,这里用**next**表示。

LinkedList类型的定义如下:

```
typedef struct LNode /*定义单链表结点类型*/  
{  
    ElemType data;  
    struct LNode *next; /*指向后继结点*/  
} LinkedList;
```

单链表基本运算

1. 建立单链表

先考虑如何建立单链表。建立单链表的常用方法有如下两种：

(1) 头插法建表

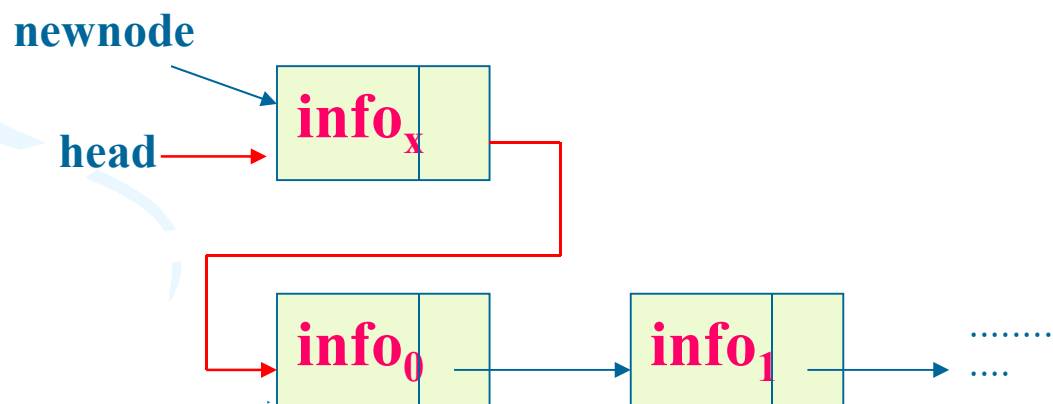
(2) 尾插法建表

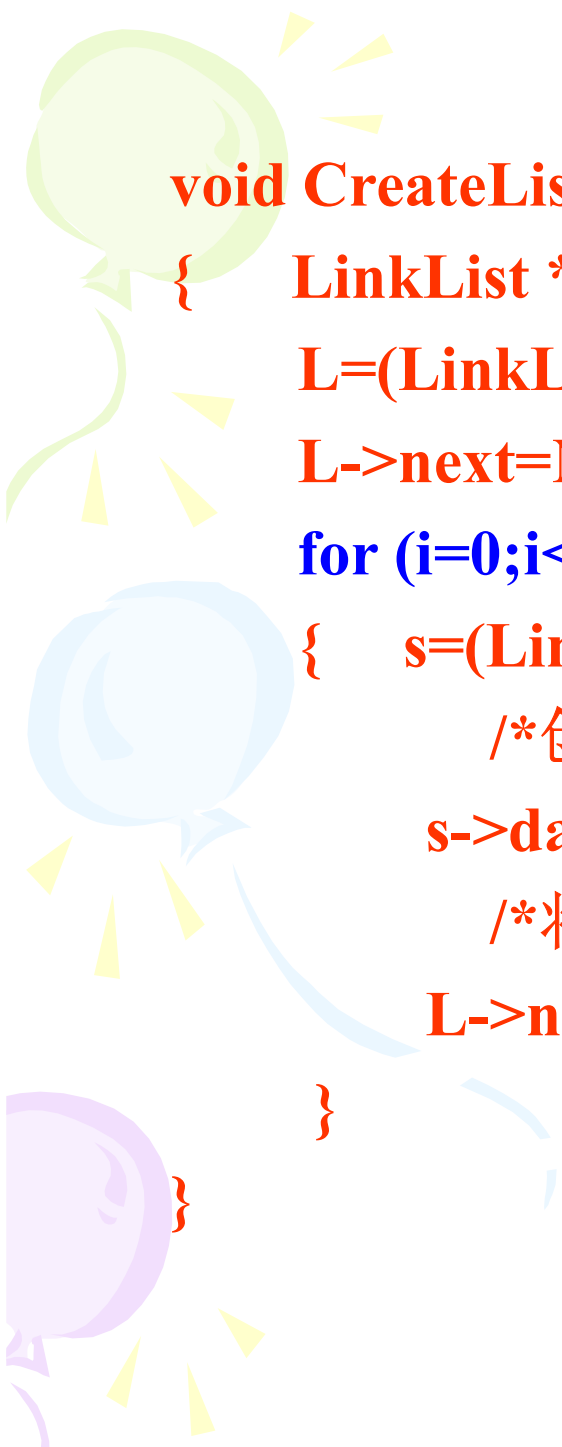
【假设】通过一个含有 n 个数据的数组来建立单链表。

(1) 头插法建表【节点插入在链首】

该方法从一个空表开始,读取字符数组a中的字符,生成新结点,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上,直到结束为止。

newnode→next=head; //注意: 链表操作次序非常重要
head=newnode;





```
void CreateListF(LinkList *&L,ElemType a[],int n)  
{   LinkList *s;int i;  
      L=(LinkList *)malloc(sizeof(LinkList)); /*创建头结点*/  
      L->next=NULL;  
      for (i=0;i<n;i++) //从数组中获取数据  
      {   s=(LinkList *)malloc(sizeof(LinkList));  
          /*创建新结点*/  
          s->data=a[i]; s->next=L->next;  
          /*将*s插在原开始结点之前,头结点之后*/  
          L->next=s;  
      }  
}
```

采用头插法建立单链表的过程

第1步: 建头结点



第2步: $i=0$, 新建a结点, 插入到头结点之后



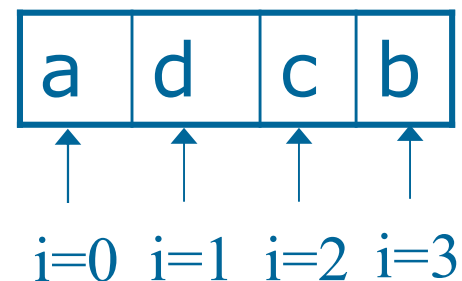
第3步: $i=1$, 新建d结点, 插入到头结点之后



第4步: $i=2$, 新建c结点, 插入到头结点之后



第5步: $i=3$, 新建b结点, 插入到头结点之后

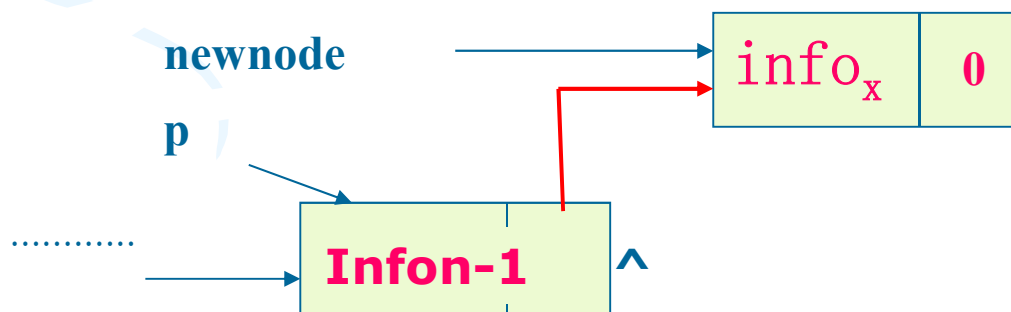


(2) 尾插法建表【节点插在队尾】

头插法建立链表虽然算法简单,但生成的链表中结点的次序和原数组元素的顺序相反。若希望两者次序一致,可采用尾插法建立。该方法是将新结点插到当前链表的表尾上,为此必须增加一个指针r,使其能找到并指向当前链表的尾结点。采用尾插法建表的算法如下:

```
newnode.next=NULL;
```

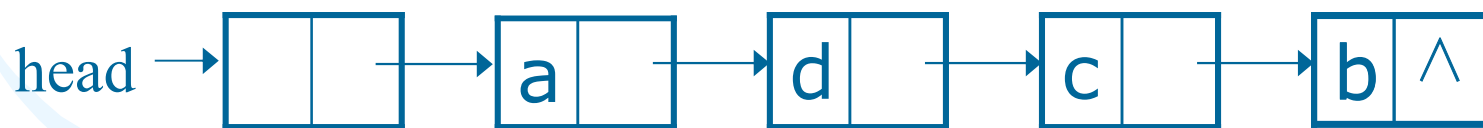
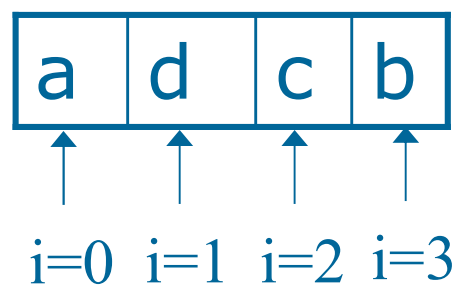
```
p→next=newnode;
```





```
void CreateListR(LinkList *&L,ElemType a[],int n)
{   LinkList *s,*r;int i;
    L=(LinkList *)malloc(sizeof(LinkList));
    /*创建头结点*/
    r=L;  /*r始终指向终端结点,开始时指向头结点*/
    for (i=0;i<n;i++)
    {   s=(LinkList *)malloc(sizeof(LinkList));
        /*创建新结点*/
        s->data=a[i];r->next=s; /*将*s插入*r之后*/
        r=s;
    }
    r->next=NULL; /*终端结点next域置为NULL*/
}
```

采用尾插法建立单链表的过程



头结点

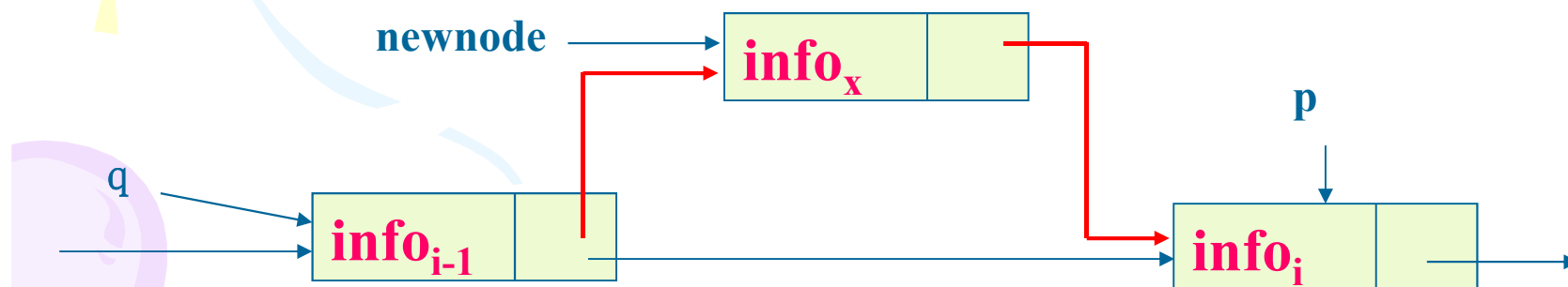
2. 插入结点运算

插入运算是将值为 x 的新结点插入到单链表的第 i 个结点（可以有其他特殊需求）的位置上。先在单链表中找到第 $i-1$ 个结点,再在其后插入新结点。

```
newnode→next=p;
```

```
//或newnode→next=q→next; 可用于插入某结点之后
```

```
q→next=newnode;
```





```
int ListInsert(LinkList *&L,int i,ElemType e)
```

```
{   int j=0;      LinkList *p=L,*s;
```

```
    while (j<i-1 && p!=NULL)  /*查找第i-1个结点*/
```

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

```
    if (p==NULL) return 0; /*未找到位序为i-1的结点*/
```

```
    else      /*找到位序为i-1的结点*p*/
```

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

```
        s->data=e;
```

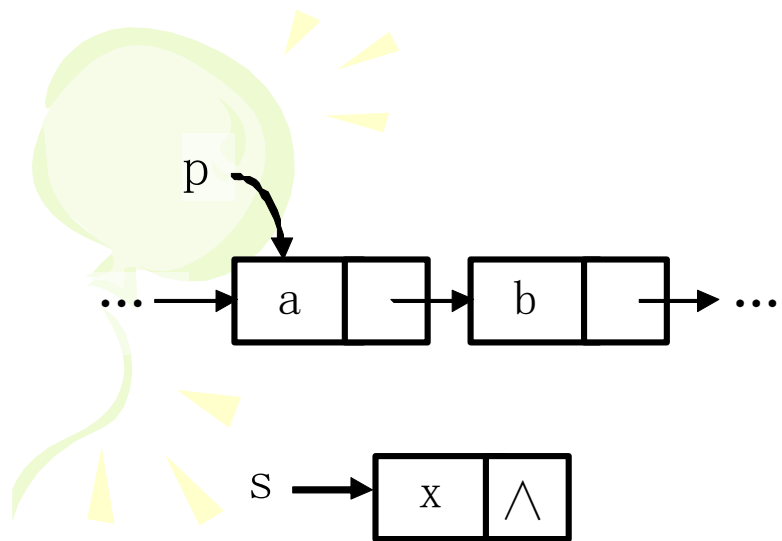
```
        s->next=p->next; /*将*s插入到*p之后*/
```

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

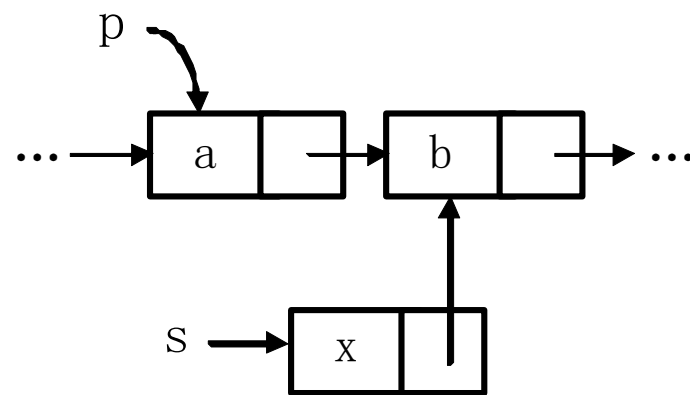
```
        return 1;
```

```
    }
```

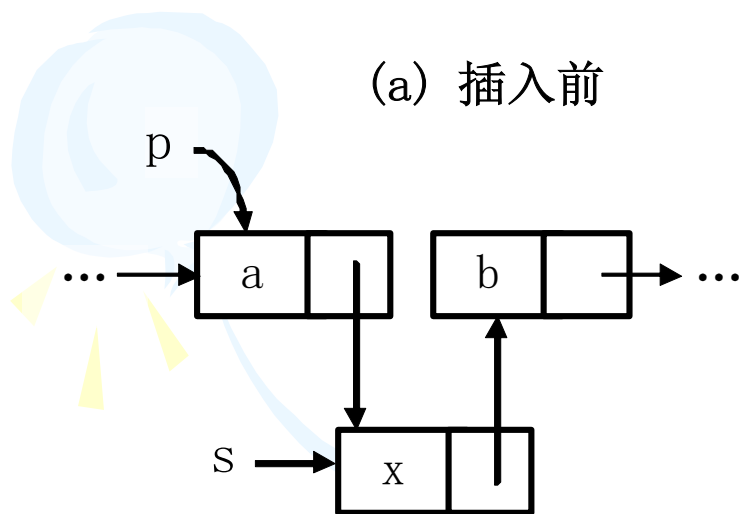
```
}
```



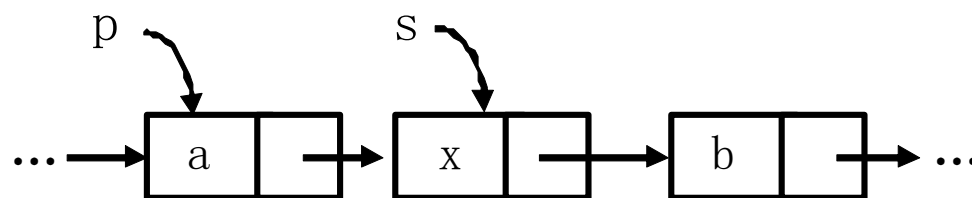
(a) 插入前



(b) $s \rightarrow next = p \rightarrow next$



(c) $p \rightarrow next = s$



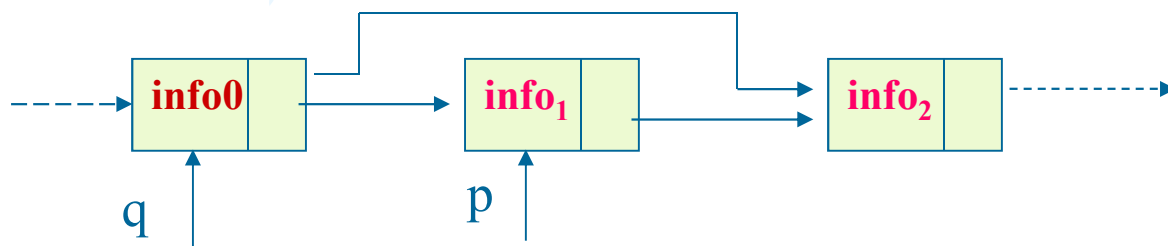
(d) 插入后

插入结点示意图

3. 删除结点运算

删除运算是将单链表的第*i*个结点删去。先在单链表中找到第*i-1*个结点,再删除其后的结点。删除单链表结点的过程如下所示。

- 令一个指针 q 指向要删除节点的前一个节点
- 令一个指针 p 指向要删除的节点
 - $p = q \rightarrow next;$
- 令删除节点的前一个节点的 $next$ 指针指向被删除节点的后一个节点。
 - $q \rightarrow next = q \rightarrow next \rightarrow next$ 或 $q \rightarrow next = p \rightarrow next;$
- 删除该节点 **delete p**





```
int ListDelete(LinkList *&L,int i,ElemType &e)
```

```
{   int j=0; LinkList *p=L,*q;
```

```
    while (j<i-1 && p!=NULL) /*查找第i-1个结点*/
```

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

```
    if (p==NULL) return 0; /*未找到位序为i-1的结点*/
```

```
    else    /*找到位序为i-1的结点*p*/
```

```
    {   q=p->next;                /*q指向要删除的结点*/
```

```
        if (q==NULL) return 0; /*若不存在第i个结点,返回0*/
```

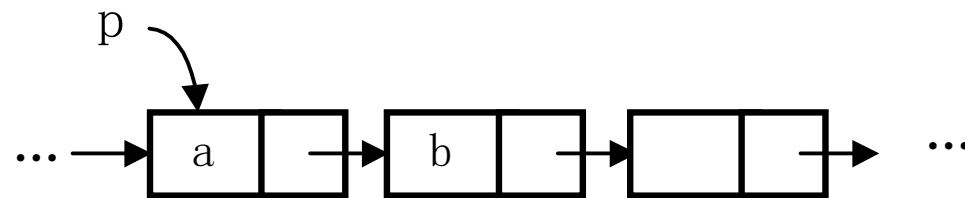
```
        p->next=q->next;        /*从单链表中删除*q结点*/
```

```
        free(q);                /*释放*q结点*/
```

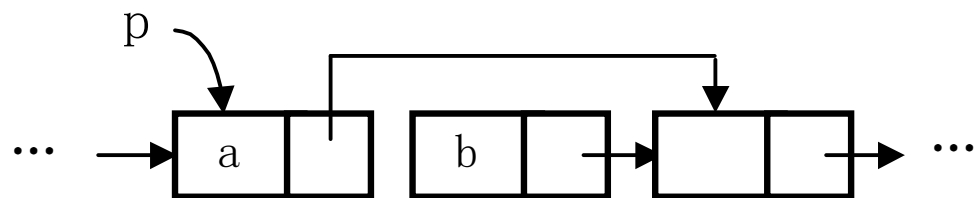
```
        return 1;
```

```
    }
```

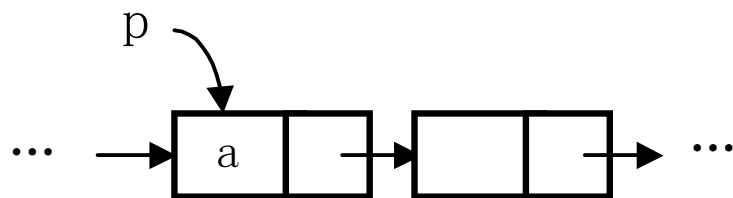
```
}
```



(a) 删除前



(b) $p \rightarrow \text{next} = p \rightarrow \text{next} \rightarrow \text{next}$



(c) 删除后

删除结点示意图



4. 线性表基本其他运算实现

(1) 初始化线性表InitList(L)

该运算建立一个空的单链表,即创建一个头结点。

```
void InitList(LinkList *&L) //指针的引用
```

```
{
```

```
    L=(LinkList *)malloc(sizeof(LinkList));    /*创建头结点*/
```

```
    L->next=NULL;
```

```
}
```



(2) 销毁线性表DestroyList(L)

释放单链表L占用的内存空间。即逐一释放全部结点的空间。



```
void DestroyList(LinkList *&L)
```

```
{   LinkList *p=L,*q=p->next;
```

```
   while (q!=NULL)
```

```
   {   free(p);
```

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

```
   }
```

```
   free(p);
```



```
}
```




(3) 判线性表是否为空表 **ListEmpty(L)**

若单链表L没有数据结点,则返回真,否则返回假。

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



(4) 求线性表的长度ListLength(L)

返回单链表L中数据结点的个数。

```
int ListLength(LinkList *L)
{
    LinkList *p=L;int i=0;
    while (p->next!=NULL)
    {
        i++;
        p=p->next;
    }
    return(i);
}
```

(5) 输出线性表DispList(L)

逐一扫描单链表L的每个数据结点,并显示各结点的data域值。

```
void DispList(LinkList *L)
```

```
{   LinkList *p=L->next;
```

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

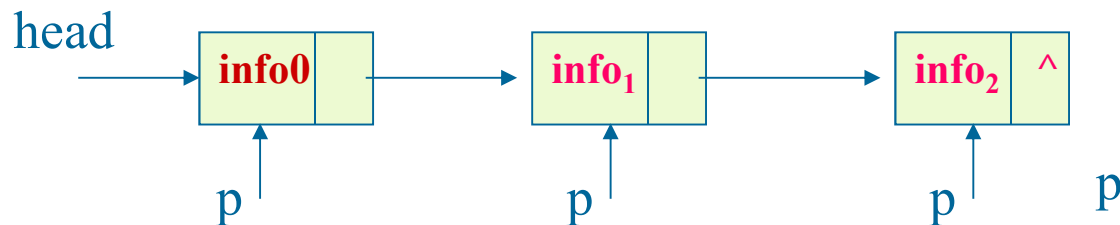
```
   {   cout<<p->data;
```

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

```
   }
```

```
   cout<<"\n";
```

```
}
```



(6) 返回线性表L中指定位置的某个数据元素

GetElem(L,i,&e)

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

```
int GetElem(LinkList *L,int i,ElemType &e)
```

```
{   int j=0;
```

```
    LinkList *p=L;
```

```
    while (j<i && p!=NULL)
```

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

```
    }
```

```
    if (p==NULL) return 0; /*不存在第i个数据结点*/
```

```
    else /*存在第i个数据结点*/
```

```
    {   e=p->data;    return 1;
```

```
    }
```

```
}
```

(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相等的结点,若存在这样的结点,则返回位置,否则返回0。

```
int LocateElem(LinkList *L,ElemType e)
```

```
{   LinkList *p=L->next;int n=1;
    while (p!=NULL && p->data!=e)
    {   p=p->next; n++; }
    if (p==NULL) return(0);
    else return(n);
```

```
}
```

修改返回值，要求：若能找到第**1**个值域与**e**相等的结点，返回结点指针；



作业：完成链表基本操作

- 用C++完成，代码可运行

- 着重分析：

- 1 是否有头结点，对算法的影响

- 2 函数的返回值

若函数返回值是**bool**型，且函数功能将新生成一个新链表，此时头指针是如何返回？

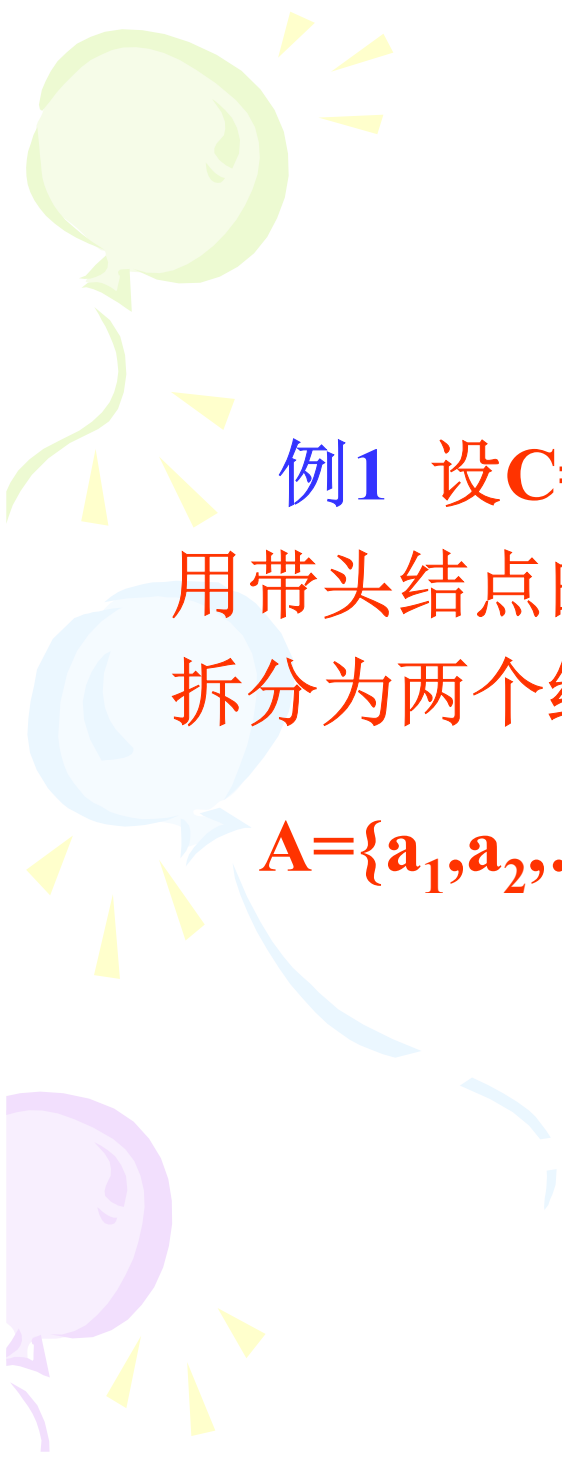
- 3 函数的形式参数

SqList *L 和 **SqList * &L**



链式存储

- 链表概述
- 若干微操作
- 单链表基本运算及其实现
- 单链表应用

The background features three stylized balloons: a green one at the top left, a light blue one in the middle left, and a purple one at the bottom left. Each balloon has a string and several small yellow triangular flags attached to it.

例1 设 $C=\{a_1, b_1, a_2, b_2, \dots, a_n, b_n\}$ 为一线性表,采用带头结点的**hc**单链表存放,编写一个算法,将其拆分为两个线性表,使得:

$$A=\{a_1, a_2, \dots, a_n\}, B=\{b_1, b_2, \dots, b_n\}$$



解：设拆分后的两个线性表都用带头结点的单链表存放。

先建立两个头结点***ha**和***hb**,它们用于存放拆分后的线性表**A**和**B**,**ra**和**rb**分别指向这两个单链表的表尾,用**p**指针扫描单链表**hc**,将当前结点***p**链到**ha**末尾,**p**沿**next**域下移一个结点,若不为空,则当前结点***p**链到**hb**末尾,**p**沿**next**域下移一个结点,如此这样,直到**p**为空。最后将两个尾结点的**next**域置空。



对应算法如下：



```
void fun(LinkList *hc, LinkList *&ha, LinkList *&hb)
```

```
{
```


```
    LinkList *p=hc->next,*ra,*rb;
```

```
    ha=hc;          /*ha的头结点利用hc的头结点*/
```

```
    ra=ha;          /*ra始终指向ha的末尾结点*/
```

```
    hb=(LinkList *)malloc(sizeof(LinkList)); /*创建hb头结点*/
```

```
    rb=hb;          /*rb始终指向hb的末尾结点*/
```



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

```
{    ra->next=p;ra=p; /*将*p链到ha单链表末尾*/
```

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

```
    if (p!=NULL)
```

```
    {    rb->next=p;
```

```
        rb=p;        /*将*p链到hb单链表末尾*/
```

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

```
    }
```

```
}
```

```
ra->next=rb->next=NULL; /*两个尾结点的next域置空*/
```

```
}
```

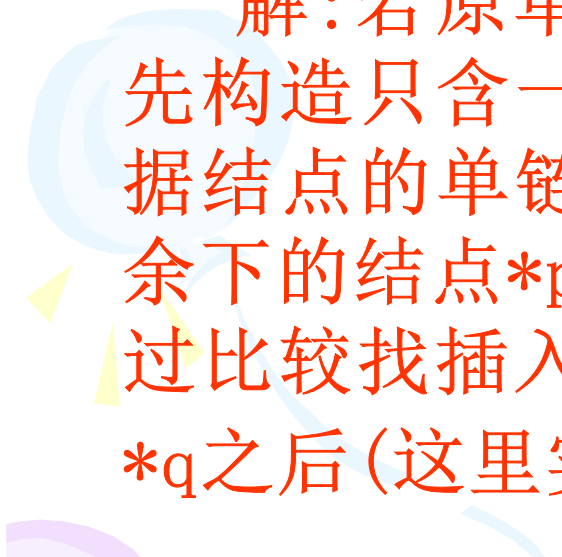


本算法实际上是采用尾插法建立两个新表。


所以, 尾插法建表算法是很多类似习题的基础!



例 2: 有一个带头结点的单链表 **head**, 其 **ElemType** 类型为 **char**, 设计一个算法使其元素递增有序。



解: 若原单链表中有一个或以上的数据结点, 先构造只含一个数据结点的有序表(只含一个数据结点的单链表一定是有序表)。扫描原单链表余下的结点 $*p$ (直到 $p == \text{NULL}$ 为止), 在有序表中通过比较找插入 $*p$ 的前驱结点 $*q$, 然后将 $*p$ 插入到 $*q$ 之后(这里实际上采用的是直接插入排序方法)。





```
void Sort(LinkList *&head)
```

```
{    LinkList *p=head->next,*q,*r;
```

```
    if (p!=NULL) /*head有一个或以上的数据结点*/
```

```
    {    r=p->next; /*r保存*p结点后继结点的指针*/
```

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

```
        /*构造只含一个数据结点的有序表*/
```

```
        p=r;
```

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

```
        {    r=p->next;
```

```
            /*r保存*p结点后继结点的指针*/
```



```
q=head;
```

```
while (q->next!=NULL && q->next->data<p->data)
```

```
    q=q->next; /*在有序表中找插入*p的前驱结点*q*/
```

```
    p->next=q->next; /*将*p插入到*q之后*/
```

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

```
    p=r;          /*扫描原单链表余下的结点*/
```

```
}
```

```
}
```

```
}
```

5

链表的综合实例1

手机通讯录

问题描述:

- 1、显示一个菜单
- 2、根据选择的菜单项完成相应的操作

欢迎光临使用手机通讯录

- 1、输入联系人
- 2、查询联系人
- 3、删除联系人
- 0、退出

请选择需要的操作

5

链表的实例

分析

定义一个结构体，可以保存保存联系人信息的姓名、手机号、住址

姓名	手机	住址
----	----	----

结构体：联系人

5

链表的实例 分析

Why?

因为手机联系人经常变，不好使用数组
使用链表



5

链表的实例

1、结构体定义

lesson19_01.c

```
struct node {  
    char name[20];  
    char phone[12];  
    char address[50];  
    struct node * next;  
} head={"", "", "", NULL};
```

head作为头
结点变量

5

链表的实例

2、主函数

```
void showMenu();  
void selectMenu();  
void add(struct node * head);  
void find(struct node * head);  
void del(struct node * head);  
int main(){  
    showMenu();  
    selectMenu();  
    return 0;
```

2023/4/12

5

链表的实例

3、输出菜单

```
void showMenu(){  
    printf("欢迎光临使用手机通讯录\n");  
    printf(" 1、输入联系人\n");  
    printf(" 2、查询联系人\n");  
    printf(" 3、删除联系人\n");  
    printf(" 0、退出\n");  
    return ;  
}
```

2023/4/12

5

链表的实例

```
void selectMenu(){
```

```
int select;
```

```
while(1){
```

```
printf("请选择需要的操作\n");
```

```
scanf("%d",&select);
```

```
getchar();/*过滤后面的回车*/
```

```
switch(select){
```

```
case 1: add(&head);break;
```

```
case 2: find(&head);break;
```

```
case 3: del(&head);break;
```

```
case 0: return;
```

```
}
```

```
}
```

```
}
```

2023/4/12

3、菜单选择

5

链表的实例

4、增加联系人

```
void add(struct node * head){  
    struct node *newNode;  
    newNode=(struct node *)malloc(sizeof(struct node ));  
    printf("姓名: ");  
    gets(newNode->name);  
    printf("电话: ");  
    gets(newNode->phone);  
    printf("住址: ");  
    gets(newNode->address);  
    newNode->next=head->next;  
    head->next= newNode;  
    return ;  
}
```

2023/4/12

5

链表的实例

5、查找联系人

```
void find(struct node * head){  
    struct node *p=head->next;  
    char name[20];  
    printf("输入查找的姓名: ");gets(name);  
    while(p!=NULL){  
        if(strcmp(name,p->name)==0){  
            printf("电话是:%s\n", p->phone);  
            break;  
        }  
        p=p->next;  
    }  
    if(p==NULL){printf("没找到!\n");}  
    return ;  
}
```

2023/4/12

5

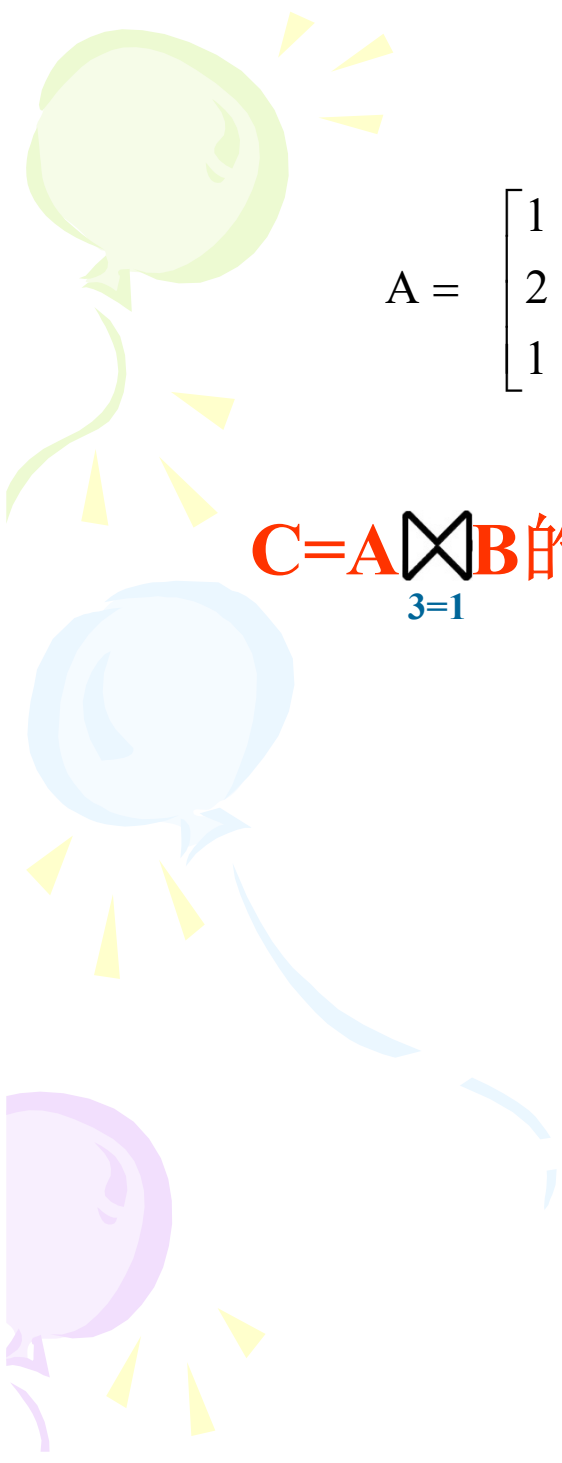
链表的实例

6、删除联系人

```
void del(struct node * head){
    struct node *pre=head,*p=head->next;
    char name[20];
    printf("输入删除的姓名: ");gets(name);
    while(p!=NULL){
        if(strcmp(name,p->name)==0){
            printf("删除的电话是:%s\n", p->phone);
            pre->next=p->next;
            free(p);
            break;
        }
        p=p->next;
    }
    if(p==NULL){printf("没找到!\n");}
    return ;
}
```

附录：线性表的应用

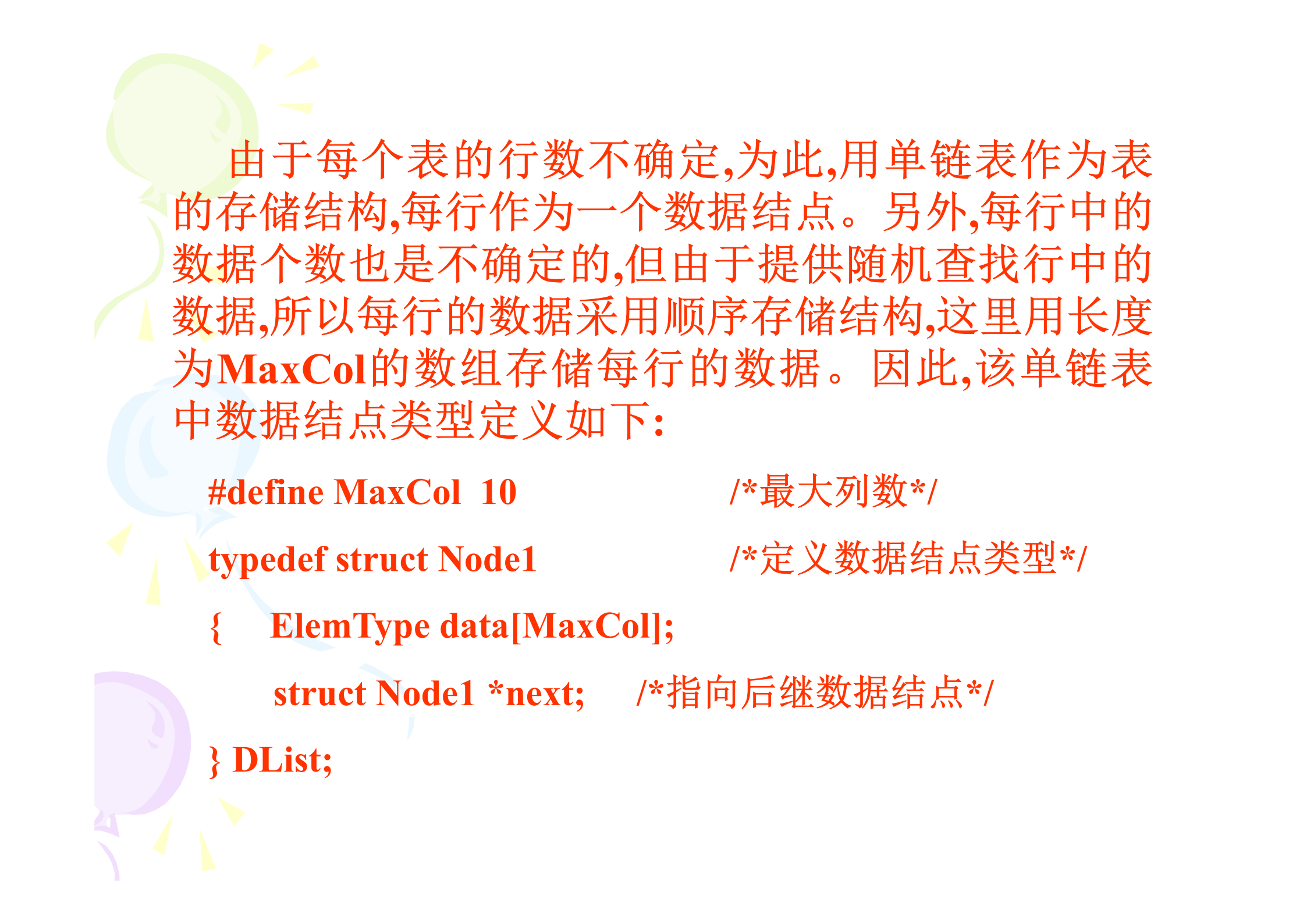
计算任意两个表的简单自然连接过程讨论线性表的应用。假设有两个表A和B,分别是m1行、n1列和m2行、n2列,它们简单自然连接结果 $C=A \bowtie_{i=j} B$,其中i表示表A中列号,j表示表B中的列号,C为A和B的笛卡儿积中满足指定连接条件的所有记录组,该连接条件为表A的第i列与表B的第j列相等。例如:


$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 3 \\ 1 & 1 & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} 3 & 5 \\ 1 & 6 \\ 3 & 4 \end{bmatrix}$$

C=A₃₌₁⊗B的计算结果如下:

$$\begin{bmatrix} 1 & 2 & 3 & 3 & 5 \\ 1 & 2 & 3 & 3 & 4 \\ 2 & 3 & 3 & 3 & 5 \\ 2 & 3 & 3 & 3 & 4 \\ 1 & 1 & 1 & 1 & 6 \end{bmatrix}$$



由于每个表的行数不确定,为此,用单链表作为表的存储结构,每行作为一个数据结点。另外,每行中的数据个数也是不确定的,但由于提供随机查找行中的数据,所以每行的数据采用顺序存储结构,这里用长度为**MaxCol**的数组存储每行的数据。因此,该单链表中数据结点类型定义如下:

```
#define MaxCol 10                /*最大列数*/  
  
typedef struct Node1             /*定义数据结点类型*/  
{  
    ElemType data[MaxCol];  
  
    struct Node1 *next;    /*指向后继数据结点*/  
} DList;
```

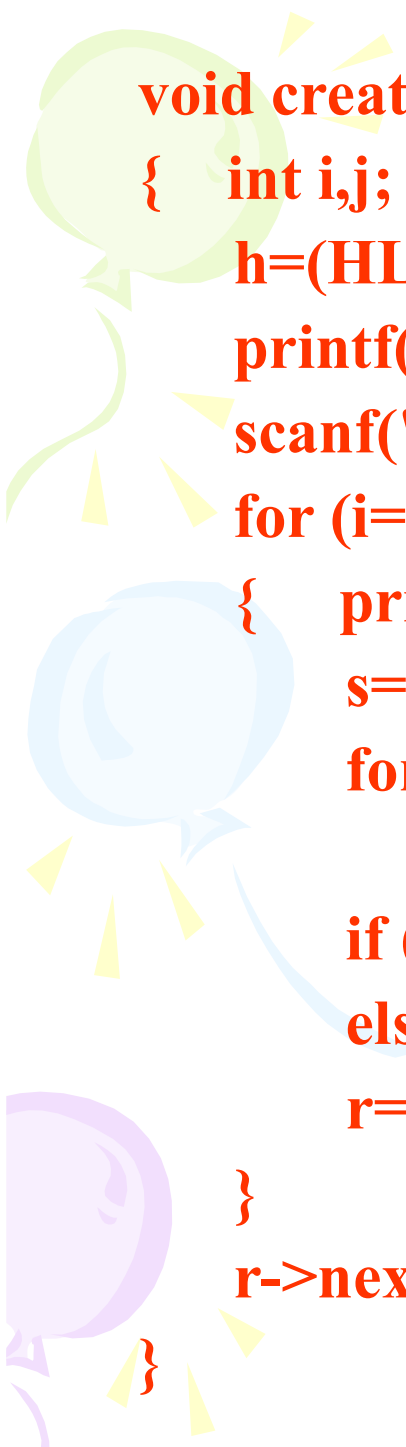
另外,需要指定每个表的行数和列数,为此将单链表的头结点类型定义如下:

```
typedef struct Node2    /*定义头结点类型*/  
{   int Row,Col; /*行数和列数*/  
    DList *next; /*指向第一个数据结点*/  
} HList;
```

采用尾插法建表方法创建单链表,用户先输入表的行数和列数,然后输入各行的数据,为了简便,假设表中数据为int型,因此定义:

```
typedef int ElemType;
```

对应的建表算法如下:



```
void create(HList *&h)
{   int i,j; DList *r,*s;
    h=(HList *)malloc(sizeof(HList));h->next=NULL;
    printf("表的行数,列数:");
    scanf("%d%d",&h->Row,&h->Col);
    for (i=0;i<h->Row;i++)
    {   printf(" 第%d行:",i+1);
        s=(DList *)malloc(sizeof(DList));
        for (j=0;j<h->Col;j++)
            scanf("%d",&s->data[j]);
        if (h->next==NULL) h->next=s;
        else r->next=s;
        r=s; /*r始终指向最后一个数据结点*/
    }
    r->next=NULL;
}
```



采用尾插法建表



对应的输出表的算法如下：

```
void display(HList *h)
```

```
{   int j;
```

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

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

```
    {   for (j=0;j<h->Col;j++)
```

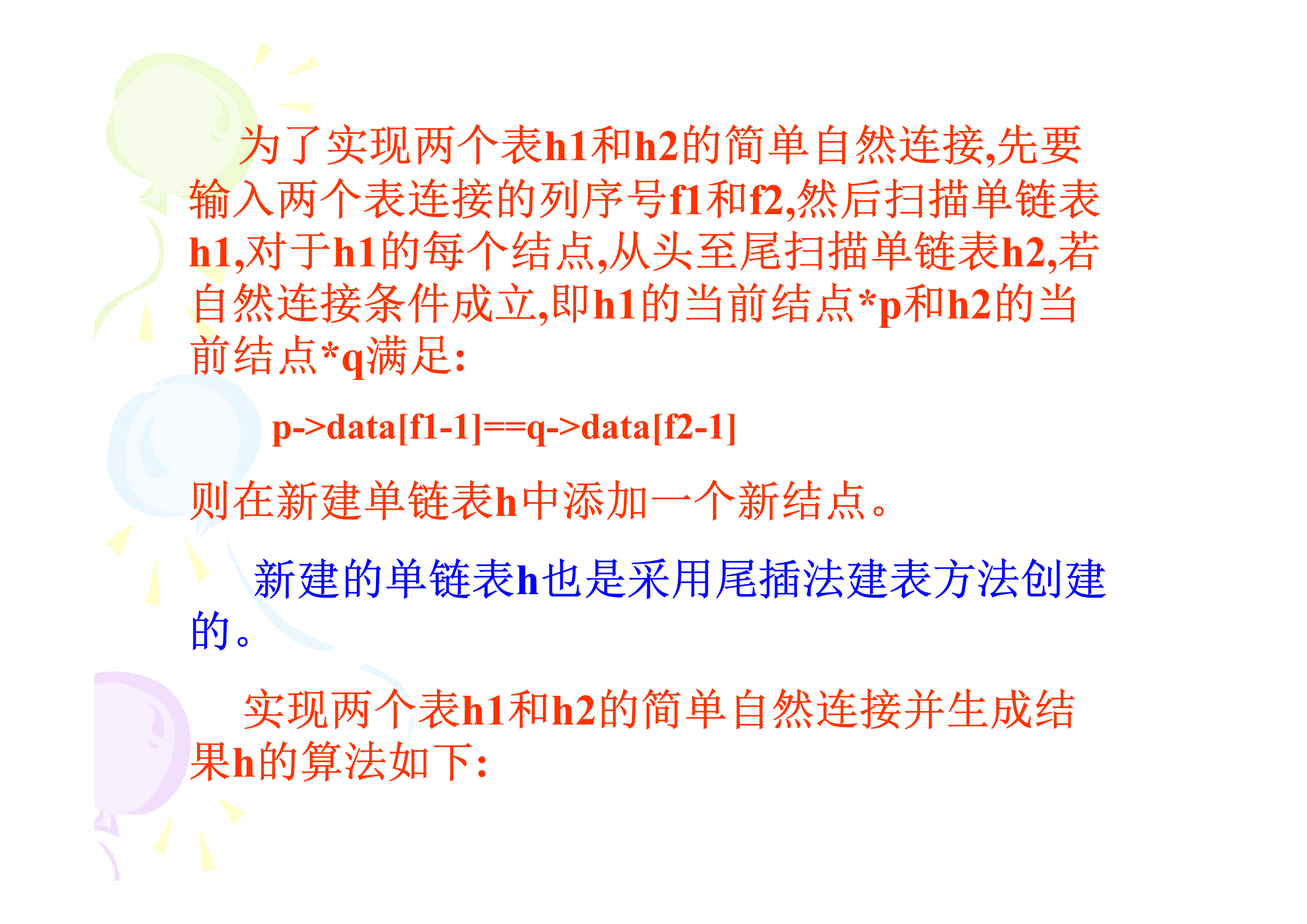
```
        printf("%4d",p->data[j]);
```

```
        printf("\n");
```

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

```
    }
```

```
}
```



为了实现两个表**h1**和**h2**的简单自然连接,先要输入两个表连接的列序号**f1**和**f2**,然后扫描单链表**h1**,对于**h1**的每个结点,从头至尾扫描单链表**h2**,若自然连接条件成立,即**h1**的当前结点***p**和**h2**的当前结点***q**满足:

$p \rightarrow data[f1-1] == q \rightarrow data[f2-1]$

则在新建单链表**h**中添加一个新结点。

新建的单链表**h**也是采用尾插法建表方法创建的。

实现两个表**h1**和**h2**的简单自然连接并生成结果**h**的算法如下:



```
void link(HList *h1,HList *h2,HList *&h)
```

```
{    int f1,f2,i;DList *p=h1->next,*q,*s,*r;
```

```
    printf("连接字段是:第1个表位序,第2个表位序:");
```

```
    scanf("%d%d",&f1,&f2);
```

```
    h=(HList *)malloc(sizeof(HList));
```

```
    h->Row=0;
```

```
    h->Col=h1->Col+h2->Col;
```

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

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

```
    {    q=h2->next;
```

尾插法建表

```
while (q!=NULL)
{
    if (p->data[f1-1]==q->data[f2-1]) /*对应字段值相等*/
    {
        s=(DList *)malloc(sizeof(DList));
        /*创建一个数据结点*/
        for (i=0;i<h1->Col;i++) /*复制表1的当前行*/
            s->data[i]=p->data[i];
        for (i=0;i<h2->Col;i++)
            s->data[h1->Col+i]=q->data[i];/*复制表2的当前行*/
        if (h->next==NULL) h->next=s;
        else r->next=s;
        r=s;          /*r始终指向最后数据结点*/
        h->Row++;      /*表行数增1*/
    }
    q=q->next; /*表2下移一个记录*/
}
p=p->next;      /*表1下移一个记录*/
}
r->next=NULL; /*表尾结点next域置空*/
}
```



一个应用：有序表

所谓**有序表**,是指这样的线性表,其中所有元素以递增或递减方式排列,并规定有序表中不存在元素值相同的元素。在这里仍以顺序表进行存储。

其中只有**ListInsert()**基本运算与前面的顺序表对应的运算有所差异,其余都是相同的。有序表的**ListInsert()**运算对应的算法如下:



```
int ListInsert(SqList &L,ElemType e)
```

```
{    int i=0,j;
```

```
    while (i<L.length && L.data[i]<e) i++;
```

```
    if (L.data[i]==e) return 0;
```

```
    for (j=ListLength(L);j>i;j--)
```

```
        /*将data[i]及后面元素后移一个位置*/
```

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

```
    L.data[i]=e;
```

```
    L.length++; /*顺序表长度增1*/
```

```
    return 1;
```

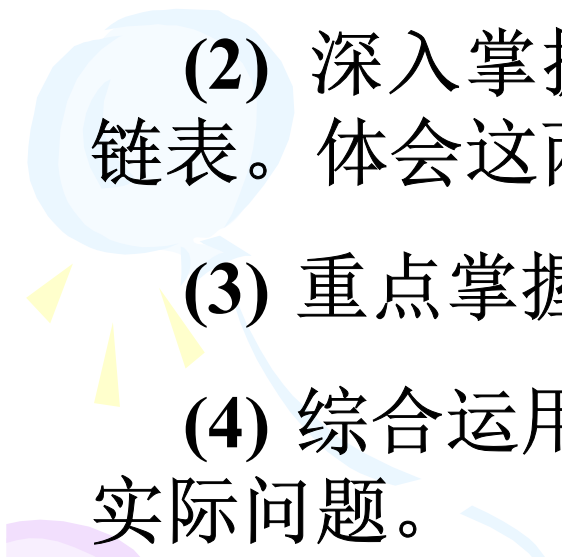
```
}
```

注意：这里用的不是
顺序表指针，直接就是
顺序表



本章小结

本章的基本学习要点如下：

- (1) 理解线性表的逻辑结构特性。
 - (2) 深入掌握线性表的两种存储方法,即顺序表和链表。体会这两种存储结构之间的差异。
 - (3) 重点掌握顺序表和链表上各种基本运算的实现。
 - (4) 综合运用线性表这种数据结构解决一些复杂的实际问题。
- 
- 