

C语言与程序设计

The C Programming Language



第14章 线性数据结构

毛伏兵

华中科技大学计算机学院



第14章 线性数据结构

- **C语言的动态存储分配函数**，进行动态数组设计。
- 然后从自引用结构、动态创建结点和相关操作等方面介绍单向链表、双向链表和十字交叉链表。
- 并进一步介绍堆栈、队列，以及基于线性数据结构的深度优先搜索、广度优先搜索等各种算法。
- 这些数据结构和算法在程序设计中起着基础性作用，需要熟练掌握。

14.1 动态存储分配

14.1.1 静态数据结构和动态数据结构

- 到目前为止，教材中介绍的各种基本类型和导出类型的数据结构都是静态数据结构。

数据结构是计算机存储、组织数据的方式。数据结构是指相互之间存在一种或多种特定关系的数据元素的集合。

- **静态数据结构**指在变量声明时建立的数据结构。在变量声明时变量的存储分配也确定了，并且在程序的执行过程中不能改变。----- **内存固定，连续**
- **动态数据结构**是在程序运行过程中通过调用系统提供的动态存储分配函数，向系统申请存储而逐步建立起来的。在程序运行过程中，动态数据结构所占存储的大小可以根据需要调节，使用完毕时可以通过释放操作将所获得的存储交还给系统供再次分配。----- **内存可变，不连续**

- 数据结构可以分为线性数据结构和非线性数据结构。
- 前者涉及链表、栈、队列等；
- 后者主要涉及树、图等。
- 简单情况下，链表、栈、队列、树、图等数据结构可以用数组等静态数据结构模拟，但作用有限。
- 更一般情况下，它们往往需要调用C的动态存储分配函数向系统申请存储而逐步建立的动态数据结构描述。
- 采用动态存储分配构造的链表、栈、队列等数据结构称为线性动态数据结构；
- 而采用动态存储分配构造的树、图等数据结构称为非线性动态数据结构。

14.1.2 C的动态存储分配函数

- 动态存储分配函数是C的标准函数，函数的原型声明在头文件**<stdlib.h>**中给出。
- 使用动态存储分配函数必须先使用**#include <stdlib.h>**编译预处理命令。
- C提供下列与动态存储分配相关的函数。

```
void * malloc(size_t size);  
void * calloc(size_t n, size_t size);  
void * realloc(void * p_block, size_t size);  
void free(void * p_block);
```

其中，**size_t**表示**unsigned int**，即无符号整型。它是在**<stdio.h>**中通过**typedef unsigned size_t;**定义的。



C的动态存储分配函数解释

void * malloc(size_t size);

void * calloc(size_t n, size_t size);

- **calloc**在动态分配完内存后，自动初始化该内存空间为零，而**malloc**不做初始化，分配到的空间中的数据是随机数据。**calloc**返回的是一个数组，而**malloc**返回的是一个对象

void * realloc(void * p_block, size_t size);

- 指针名 = (数据类型*) **realloc** (要改变内存大小的指针名, 新的大小)
- 传递给**realloc**的指针必须是先前通过**malloc()**, **calloc()**, 或**realloc()**分配的

void free(void * p_block);

Alignment(内存地址对齐)

- 一般情况下计算机对内存是按字进行读写的，32位计算机每次读写4字节的一个字，并且都是从0,4,8等4的倍数地址开始读写一个字。
- 一般地，将整型变量从4的倍数地址处对齐存放，其读写速度是最快的。



*14.2 动态数组设计

- 在没有学习动态存储分配之前，如果要设计某班的C语言课程成绩单，首先必须知道人数，并且要按照可能出现的最多人数进行相关数组的声明。**C99支持动态数组**，例如：

```
int n, i;
```

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

```
int a[n];
```

可以动态创建大小为n的数组。

- 但是，如果要求学号、姓名等数据项的长度也不必事先规定，而是根据各个班上的实际人数的多少、学号、姓名等数据项的实际长度，来恰当、无冗余地表示，使用动态存储分配函数创建动态数组也许更加合适。

例14.2 用动态存储分配方法设计某班的C语言课程成绩单。

```
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include "assert.h"
```

成绩单包含学号字符指针`num`和姓名字符指针`name`，它们将分别指向动态分配的学号存储区和姓名存储区。其优点是可以应用于不同长度的学号、姓名，并且根据学号、姓名实际长度分配存储，适应面广，且不会存在多余的空闲存储。

```
struct c_score_tab{
    char *num;           /* 学号 */
    char *name;          /* 姓名 */
    int    c;            /* C语言课程成绩 */
};
```

```
void dynamic_input(char *s1,char **s2)
```

```
/*字符指针形参s1指向提示性的实参字符串  
输入字符串到 (*s2) 指向的存储区 */
```

```
{ char pc[40];
```

```
int len;
```

```
printf(s1); /* 根据需要输出不同的提示 */
```

```
gets(pc); /* 从键盘接收输入的字符串 */
```

```
len=strlen(pc); /* 计算输入的字符串的长度 */
```

```
(*s2)=(char *)malloc(len*sizeof(char)+1);
```

```
/*根据输入串长度适当分配存储，  
动态分配len+1个字节，并由(*s2)指向  
实际上是有len+1个元素的动态字符数组*/
```

```
assert((*s2)); /* 断言 (*s) 非零 */
```

```
strcpy((*s2),pc); /* 将输入串拷贝到动态申请的存储区 */
```

```
}
```



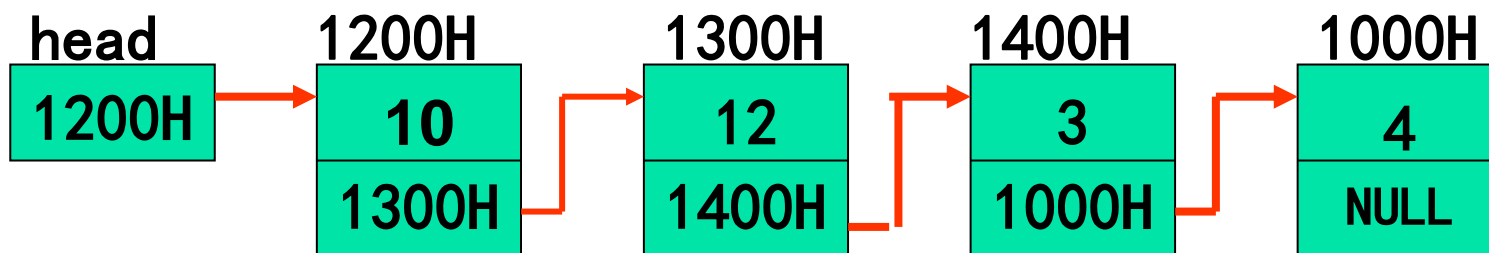
```

void main(void)
{
    int n,i;  struct c_score_tab *p;
    printf("input the number of students please!\n");
    scanf("%d",&n);
    getchar(); /* getchar用于读结束scanf输入的回车符 */
    p=(struct c_score_tab *)
        malloc(n*sizeof(struct c_score_tab));
    assert(p);
    for(i=0;i<n;i++){
        dynamic_input("input num!\n",&(p+i)->num );
        dynamic_input("input name!\n",&(p+i)->name);
        printf("input score!\n");
        scanf("%d",&(p+i)->c); /* 输入C语言课程成绩 */
        getchar();
    }
    printf("\n");
    for(i=0;i<n;i++) /* 打印成绩单 */
        printf("unm=%s\tname=%s\tscore=%d\n",
            (p+i)->num, (p+i)->name, (p+i)->c);
    printf("\n");
    free(p); /* 释放成绩单占据的存储 */
}

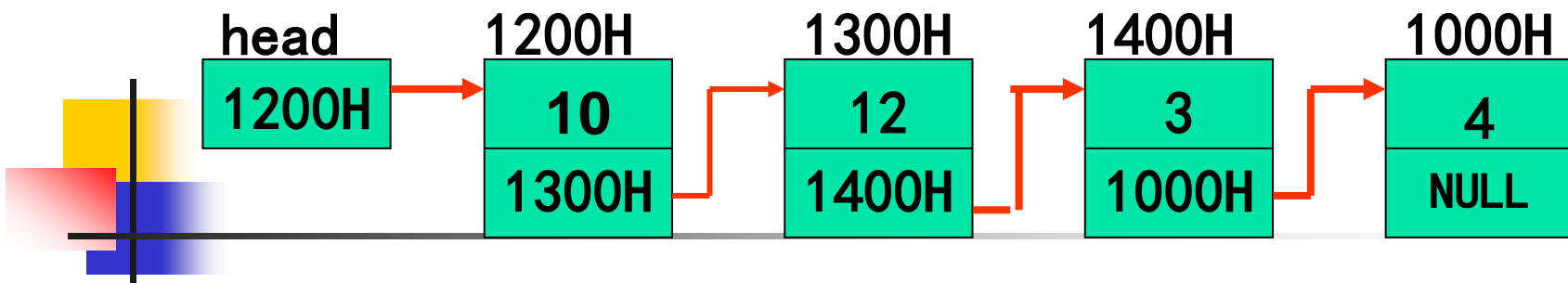
```

14.3 链 表

- 链表是一种常用的动态数据结构，它由一系列包含数据域和指针域的结点组成。
- 如果结点的指针域中只包含一个指向后一个结点指针，这种链表称为单向链表。



单向链表结构



单向链表结构

head: 头指针，存放一个地址，指向第一个元素

结点: 链表中的元素，包括 **数据域**，用户数据
指针域，放下一个结点的地址

链尾: 最后一个结点，指针域为NULL

链表结点的结构类型定义

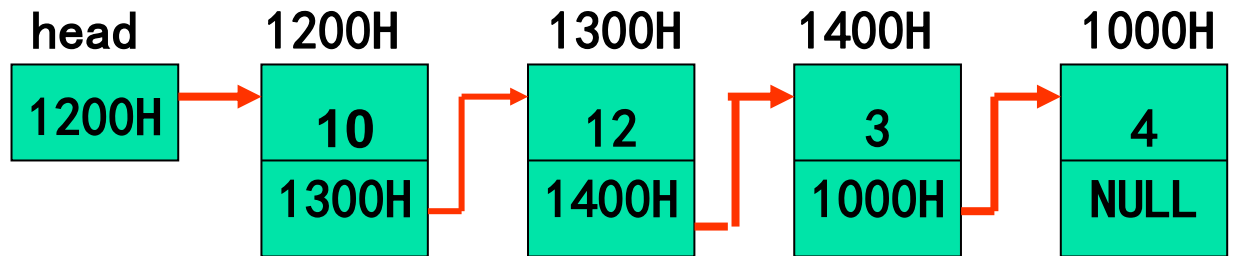
```
struct s_list{  
    int data;
```

```
    struct s_list *next; /*指向该结构自身的指针*/
```

```
};
```

头指针说明

```
struct s_list *head;
```



14.3.1 自引用结构

如果一个结构中**包含一个指向该结构自身的指针**，称该结构类型为**自引用结构类型**

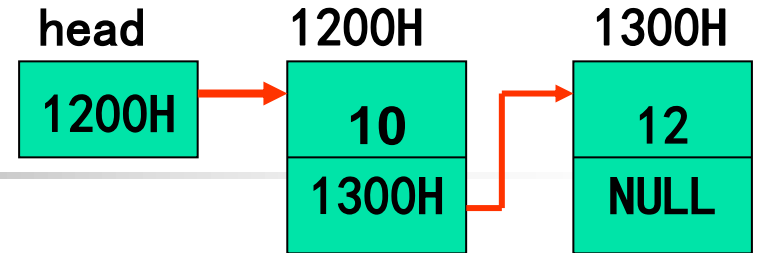
```
struct s_list node1={ 10,NULL};
```

```
/* node1为自引用结构变量 */
```

```
struct s_list *head;
```

```
/* head自引用结构类型的指针 */
```

14.3.2 动态创建结点



- 可以通过**malloc**的方式来动态创建结点。

- 如：

```
head=(struct s_list *)malloc(sizeof(struct s_list)) ;
```

```
head->data=10;
```

```
head->next=(struct s_list *)malloc(sizeof(struct s_list));
```

```
head->next->data=12;
```

```
head->next->next=NULL;
```




14.3.3 单向链表

链表的建立

先进后出链（用链表实现栈）：

结点的排列顺序和数的输入顺序相反

先进先出链（用链表实现队列）：

结点的排列顺序和数的输入顺序相同

1. 用循环方式建立先进先出链表

建立一个非空先进先出链表相关算法步骤如下：

(1) 声明头指针，尾指针。

```
struct s_list * loc_head=NULL,*tail;
```

(2) 创建第一个结点。包括：

① 给第一个结点动态分配存储并使头指针指向它。

```
loc_head=(struct s_list *)malloc(sizeof(struct s_list));
```

② 给第一个结点的数据域中成员赋值。

```
loc_head->data=*p++;
```

③ 使尾指针也指向第一个结点。

```
tail=loc_head;
```

(3) 循环建立后续结点

如果没遇到结束标志，进行下列操作：

① 给后继结点动态分配存储并使前驱结点的指针指向它。

```
tail->next=(struct s_list *)malloc(sizeof(struct s_list));
```

② 使尾指针指向新建立的后继结点

```
tail=tail->next;
```

③ 给后继结点的数据域中成员赋值。

```
tail->data=*p++;
```

(4) 给尾结点（最后一个结点）的指针赋**NULL**值。

```
tail->next=NULL;
```

/*create_list_v1: 用循环方式建立先进先出链表 *headp,
将指针p所指的整数依次存放到链表的每个节点。*/

void create_list_v1(struct s_list **headp, int *p)

{ struct s_list * loc_head=NULL,*tail;

~~if(p[0]==0) ; /* 相当于*p==0 */~~

else { /* loc_head指向动态分配的第一个结点 */

loc_head=(struct s_list *)malloc(sizeof(struct s_list));

loc_head->data=*p++; /* 对数据域赋值 */

tail=loc_head; /* tail指向第一个结点 */

while(*p){ /* tail所指结点的指针域指向动态创建的结点 */

tail->next=(struct s_list *)malloc(sizeof(struct s_list));

tail=tail->next; /* tail指向新创建的结点 */

tail->data=*p++; /* 向新创建的结点的数据域赋值 */

}

tail->next=NULL; /* 对指针域赋NULL值 */

}

***headp=loc_head; /* 使头指针headp指向新创建的链表 */**

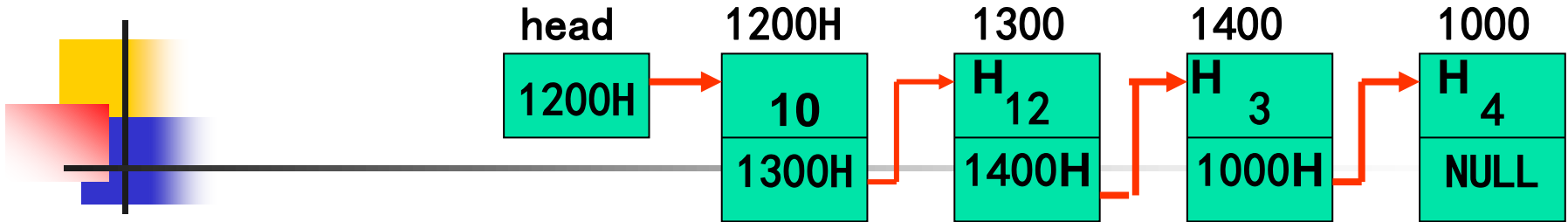
}

在main函数中调用create_list_v1

```
void main(void)
```

```
{  
    struct s_list *head=NULL,*p;  
    int s[]={1,2,3,4,5,6,7,8,0}; /* 0为结束标记 */  
    create_list_v1(&head,s); /* 创建新链表 */  
    print_list(head); /* 输出链表结点中的数据 */  
}
```

遍历链表的算法步骤



- (1) 初始化，使遍历指针 p 指向头结点。 $p = \text{head};$
- (2) 如果链表非空（即遍历指针 p 非空， $p \neq \text{NULL}$ ）

(2-1) 输出结点数据域中成员的值。

`printf("%d\t", p->data);`

(2-2) 使遍历指针 p 指向下一个结点。 $p = p \rightarrow \text{next};$

`void print_list(struct s_list *head)`

```
{ struct s_list * p=head; /*遍历指针p指向链头 */
  while(p){
    printf("%d\t", p->data);
    p=p->next; /*遍历指针p指向下一结点 */
  }
}
```

2. 用递归方式建立先进先出链表

/*create_list_v2: 用递归方式建立先进先出链表，将指针p所指的整数依次存放到链表的每个结点。返回该链表的头指针*/

struct s_list *create_list_v2(int *p)

{

struct s_list * loc_head=NULL;

if(p[0]==0) /* 遇到结束标记，返回NULL

return NULL;

else { /* loc_head指向新创建的结点 */

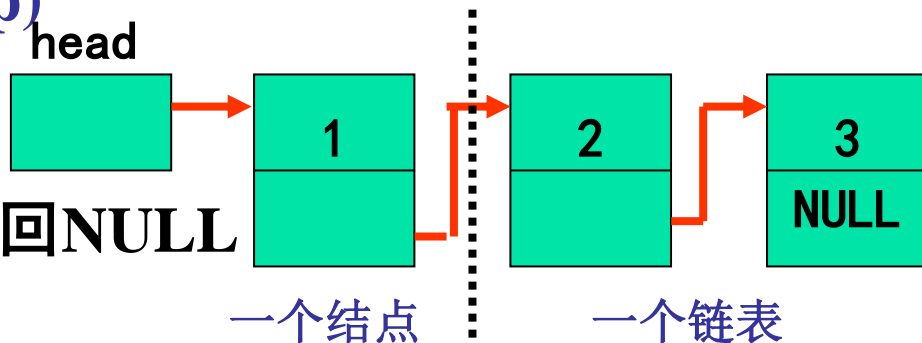
loc_head=(struct s_list *)malloc(sizeof(struct s_list));

loc_head->data=*p; /* 对新创建结点的数据域赋值 */

loc_head->next=create_list_v2(p+1); /* 递归创建下一结点 */

return loc_head; /* 返回链头地址 */

}
}





14.3.4 链表的相关操作

- 链表的相关操作包括：
- 创建链表; ✓
- 遍历链表;
- 在链表中插入一个结点;
- 删除链表中的一个结点;
- 同类型链表的归并;
- 链表的排序等。



1. 遍历链表

- 遍历链表可以进一步分为：
 - ◆ 输出链表中各个结点的数据域成员； ✓
 - ◆ 统计链表中结点的数目；
 - ◆ 查找链表中符合条件的某个结点等。

其中输出链表中各个结点的数据域成员已经介绍。

例 用循环遍历的方式统计链表中结点的数目（即链表的长度）。

/*count_nodes用循环遍历法统计链表head中结点的数目*/

```
int count_nodes(struct s_list * head)
```

```
{
```

```
    struct s_list *p=head;
```

```
    int num=0;
```

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

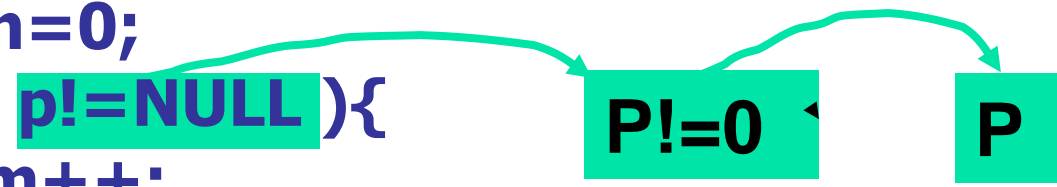
```
        num++;
```

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

```
    }
```

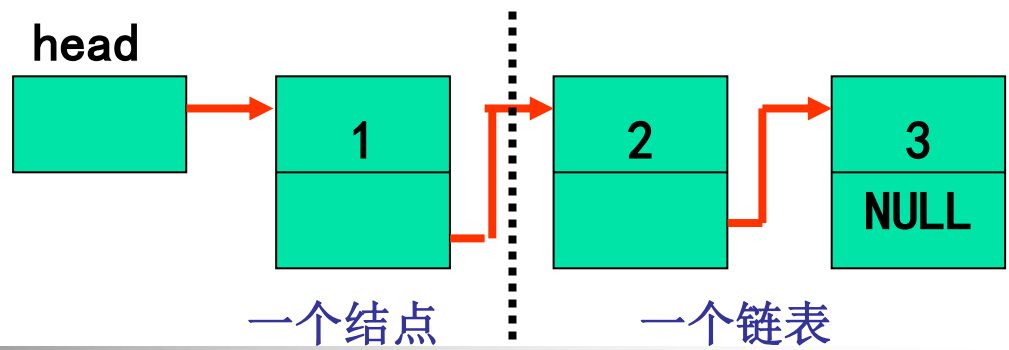
```
    return num;
```

```
}
```



P!=0

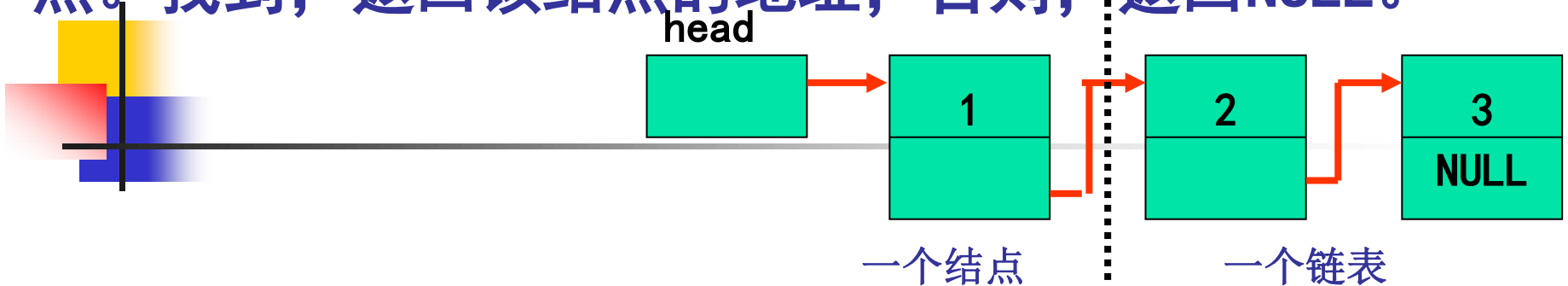
P



/*用递归法统计链表head中结点的数目 */

```
int count_nodes_recursive(struct s_list * head)  
{  
    struct s_list * p=head;  
    if(p)  
        return (1+count_nodes_recursive(p->next));  
    else  
        return 0;  
}
```

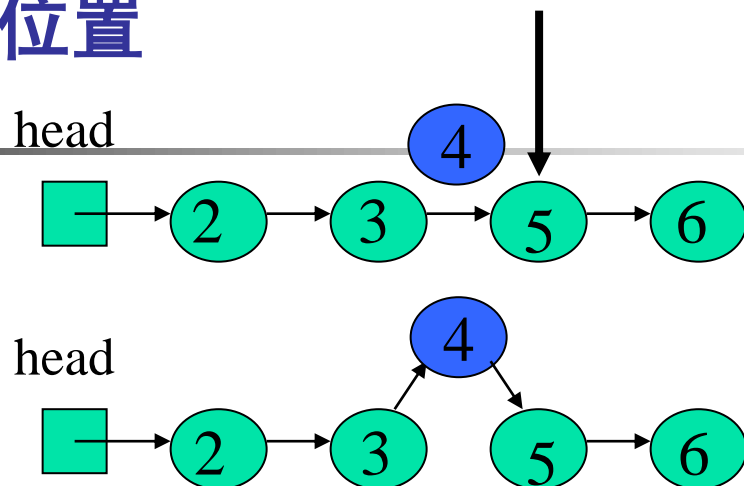
例 用递归法查找**链表**中数据成员值与形参**n**相同的结点。找到，返回该结点的地址，否则，返回NULL。



```
struct s_list * find_nodes_recursive(struct s_list * head, int n)
{
    struct s_list * p=head;
    if(p) { /* 链表非空，查找 */
        if(p->data==n)
            return p; /* 找到，返回该结点的地址 */
        else
            return (find_nodes_recursive(p->next,n)); /* 递归查找 */
    }
    else return NULL;
}
```

2、插入结点

(1) 寻找插入位置



插入点的位置可能是链头、链尾、或者链中。

插入方式有将加入结点作为插入点的新后继结点和插入点的新前驱结点两种。

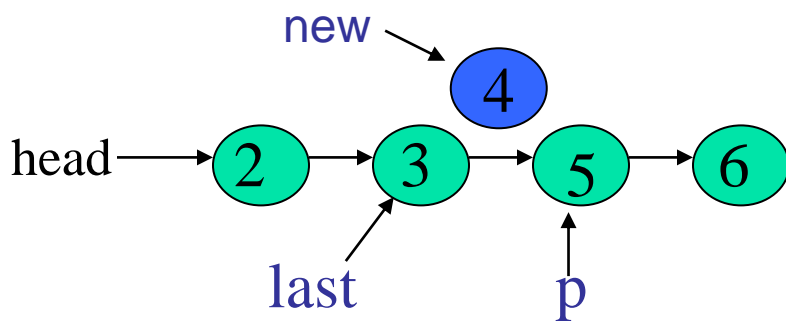
教材介绍将加入结点作为插入点的新后继结点（自学）

本节介绍将加入结点作为插入点的新前驱结点。

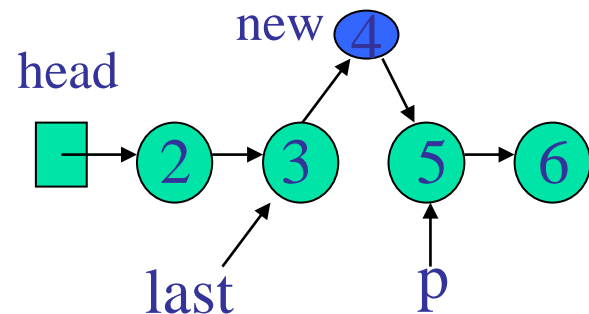
(2) 插入结点

插入点是链中 (p 和 $last$ 之间)

需要两个遍历指针，一个是指向插入点的遍历指针 p ，另一个是指向插入点前驱结点的指针 $last$ 。设新结点由 new 指针所指，它将插入在 p 和 $last$ 所指结点之间。



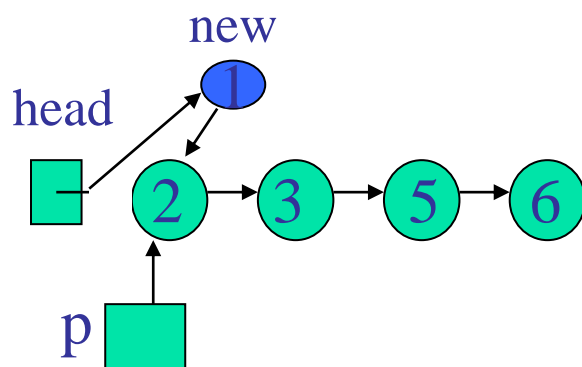
插入前结点间的指向关系



插入后结点间的指向关系

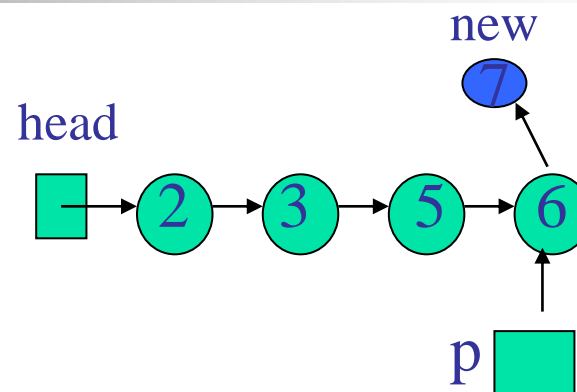
$last \rightarrow next = new;$
 $new \rightarrow next = p;$

插入点是链头



```
head=new;  
new->next=p;
```

插入点是链尾



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

**/* insertnode : 在一个升序链表head中插入值为x的结点,
返回链表第一个结点的地址。 */**

#define LEN sizeof(struct s_list)

struct s_list **insertnode* (struct s_list *head, int x)

{

struct s_list *p, *last, *new;

new = (struct s_list *)malloc(LEN); */* 创建一个新节点 */*

new->data=x;

p=head;

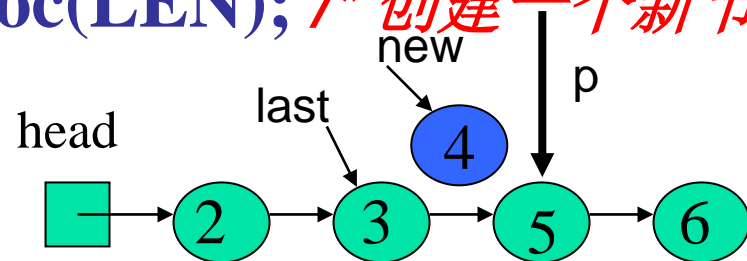
/ 寻找插入位置 */*

while(x > p->data && p->next != NULL) {

last=p;

p=p->next;

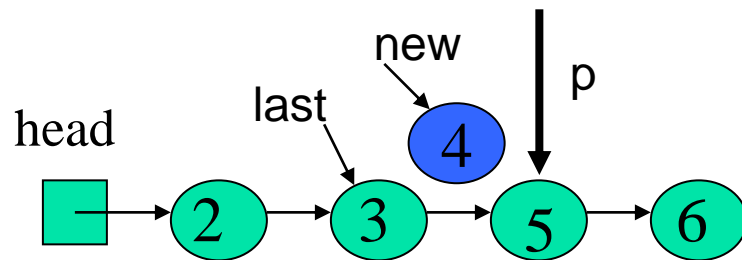
}




```

if ( x < p->data ) { /* 插入点不是链尾 */
    if ( p == head ) head = new; /* 新节点为链头 */
    else last->next = new; /* 链中 */
    new->next = p;
}
else { /* 新节点为链尾 */
    new->next = NULL;
    p->next = new;
}
return head;
}

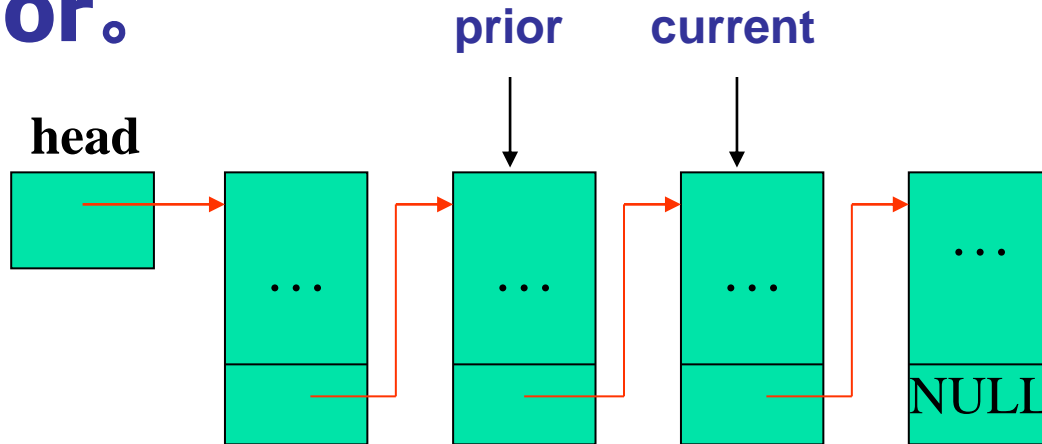
```



3. 删除结点

(1)查找被删结点

需要指向被删除结点的遍历指针**current**和
指向被删除结点的前驱结点的遍历指针
prior。

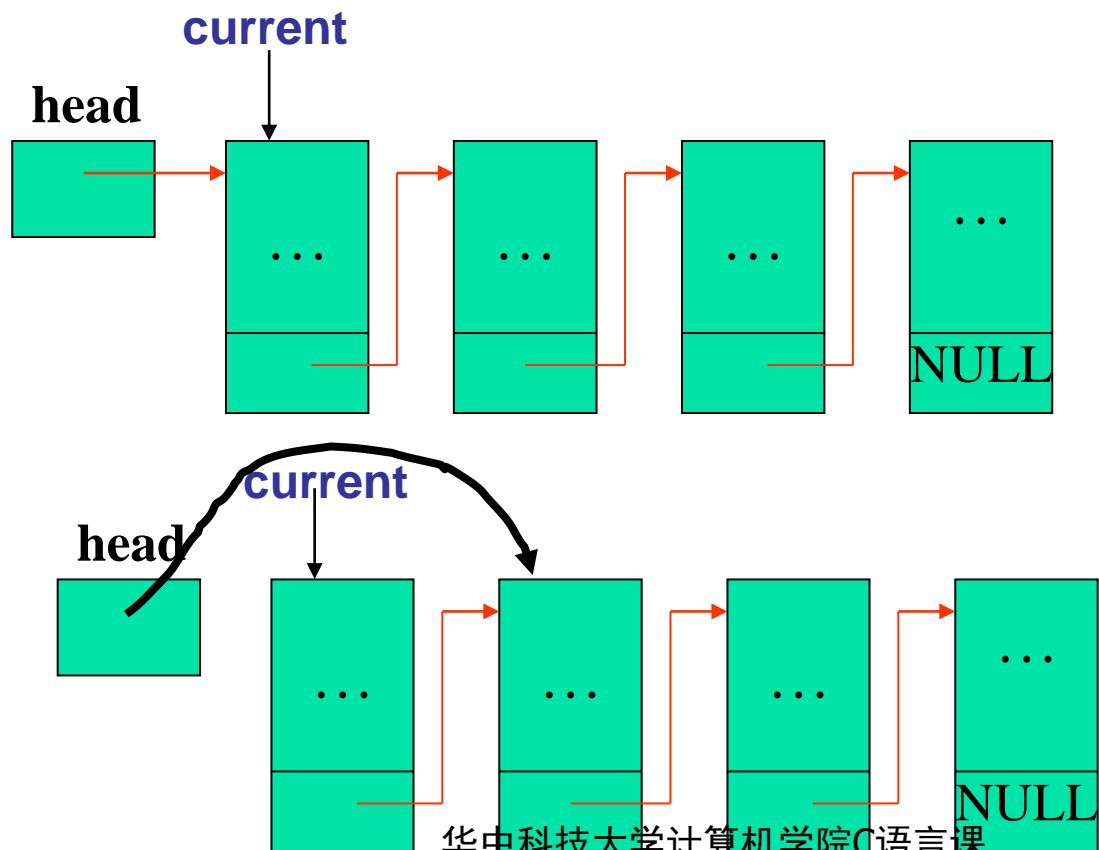


(2) 删除current所指结点

即改变连接关系

链头

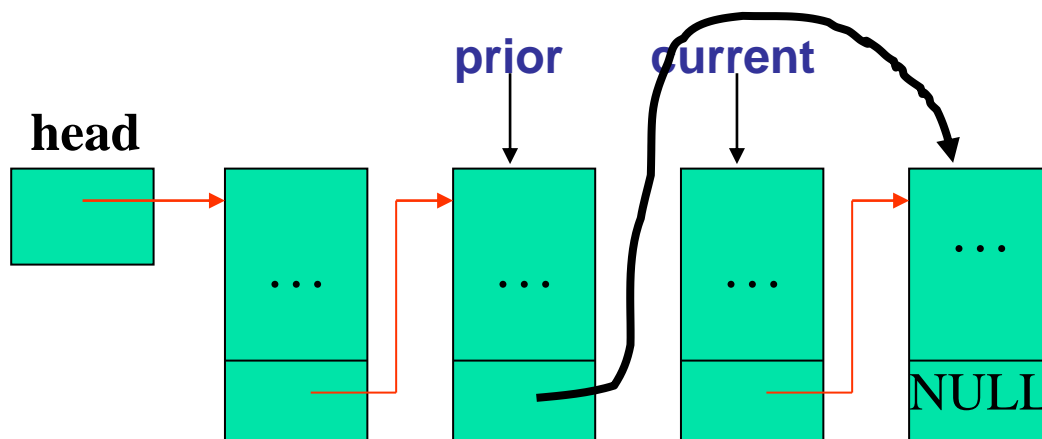
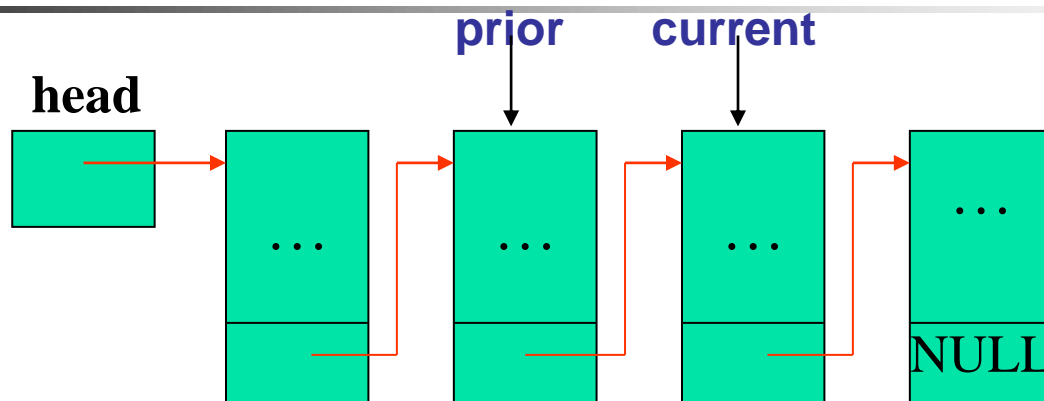
修改 *head*: $head = current \rightarrow next$



非链头

修改前驱结点的指针域:

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



(3) 释放 `current` 所指的存储区



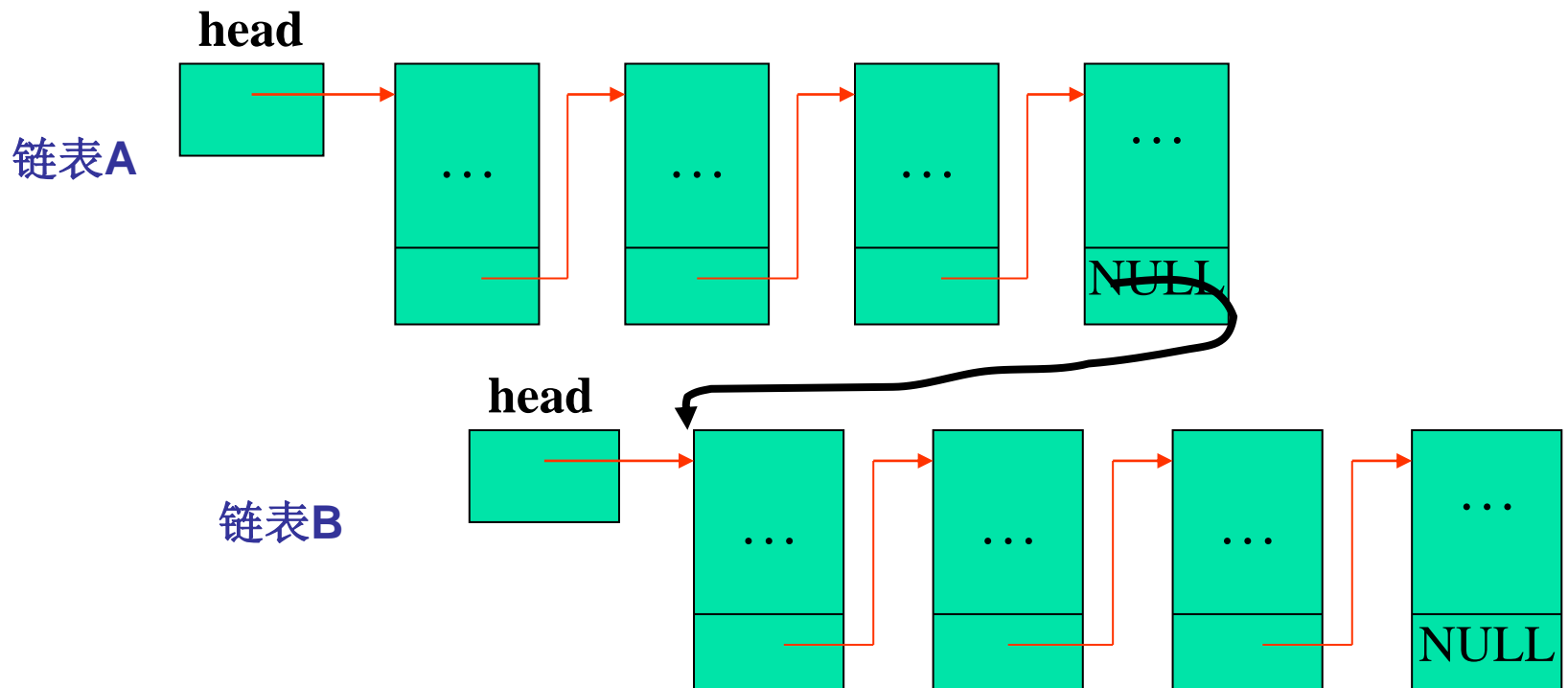
```
free(current);
```

/* 查找链表中数据成员的值与形参n相等的结点，如果找到，删除该结点，并返回插入点的地址；如果没有找到，返回NULL。*/

```
struct s_list *delete_nodes(struct s_list **headp,int n)
{
    struct s_list *current=*headp,*prior=*headp;
    /* 查找成员值与n相等的结点 */
    while(current !=NULL&&current->data!=n){
        prior=current; /* prior指向当前结点 */
        current=current->next; /* current指向下一结点 */
    }
    if(!current) /* current==NULL，没有符合条件的结点 */
        return NULL;
    if(current==*headp) /* 被删结点是链头*/
        *headp=current->next;
    else /* 被删结点不是链头 */
        prior->next= current->next;
    free(current); /* 释放被删结点的存储 */
    return current;
}
```

4. 归并链表

对于非空链表A、B,将链表B归并到链表A, 指将链表A的链尾指向链表B的链头所形成的一个新的链表。



归并链表的算法

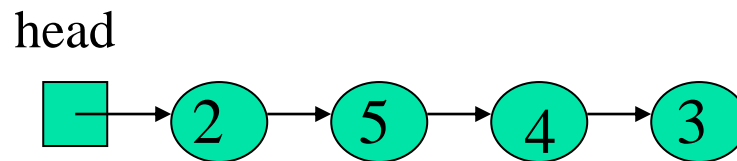
(1) 遍历链表A找到其链尾

(2) 将链表B的头指针值赋给链表A链尾的指针域

```
void concatenate_lists ( struct s_list *headA,  
                          struct s_list *headB  )  
{  
    struct s_list *current=headA;  
    while(current-> next != NULL ) {  
        current=current->next;  
    }  
    current->next=headB;  
}
```


5. 链表排序

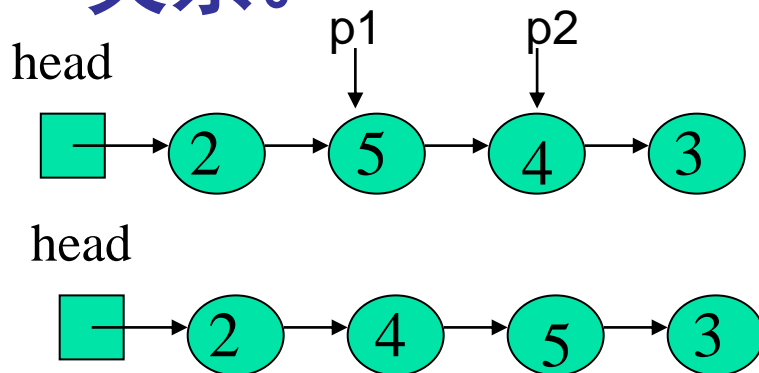
- 链表排序是指将链表的结点按某个数据域从小到大（升序）或从大到小（降序）的顺序连接。



两种方法

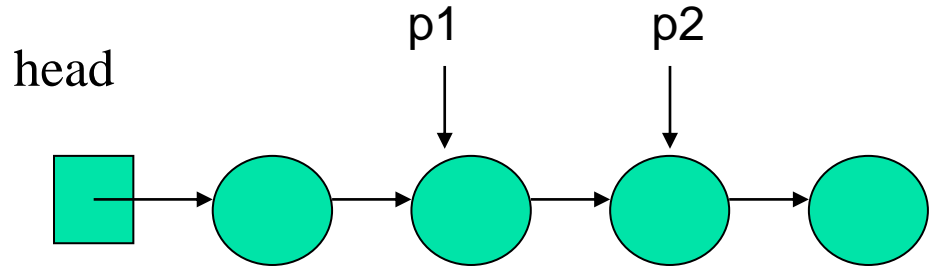
排序中对链表结点的交换有**两种方法**,

(1) 交换结点的数据域, 不改变结点间的指向关系。



```
if(p1->data > p2->data){  
    int t;  
    t=p1->data; /* 交换数据域 */  
    p1->data=p2->data;  
    p2->data=t;  
}
```

```
struct std {  
    char num[12];  
    char name[9];  
    int score;  
};
```



```
struct s_list {  
    struct std data;  
    struct s_list *next;  
};  
  
if(p1->data->score > p2->data->score) {  
    struct std t;  
    t=p1->data; /* 交换数据域 */  
    p1->data =p2->data;  
    p2->data =t;  
}
```

例 写一个函数，采用交换结点数据域的方法实现对链表的升序排序。

```
void sort_lists(struct s_list *head)
```

```
{
```

```
    struct s_list *p1=head,*p2;
```

```
    int t;
```

```
    for(p1=head;p1!=NULL;p1=p1->next)
```

```
        for(p2=p1->next;p2!=NULL;p2=p2->next)
```

```
            if(p1->data > p2->data){
```

```
                t=p1->data;
```

```
                p1->data=p2->data;
```

```
                p2->data=t;
```

```
            }
```

```
}
```

head



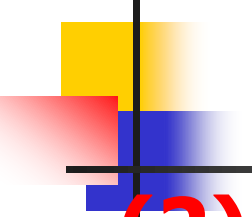
p1

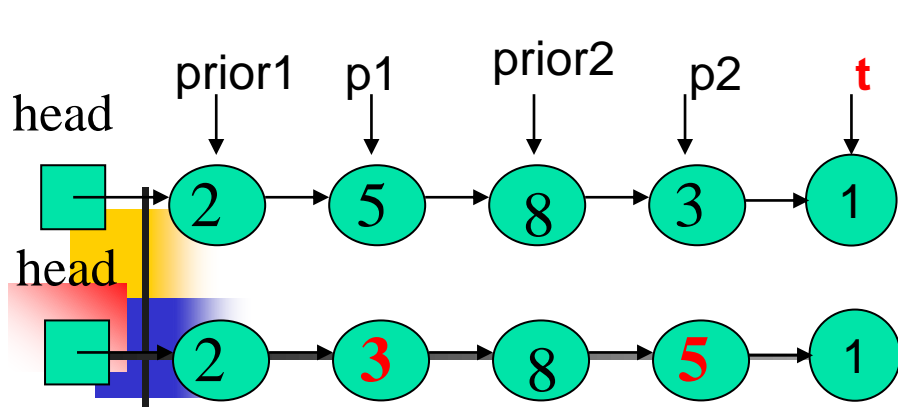
p2

p2

/* 交换数据域 */

每轮得到最小的值（在最开始的位置上）

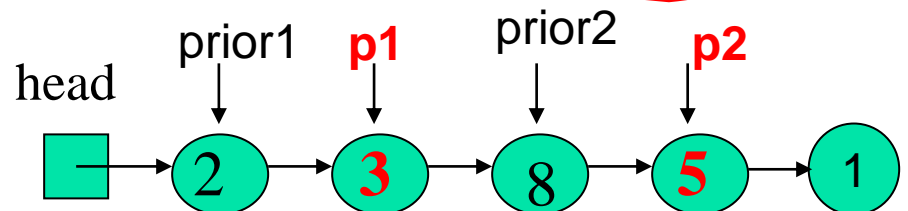
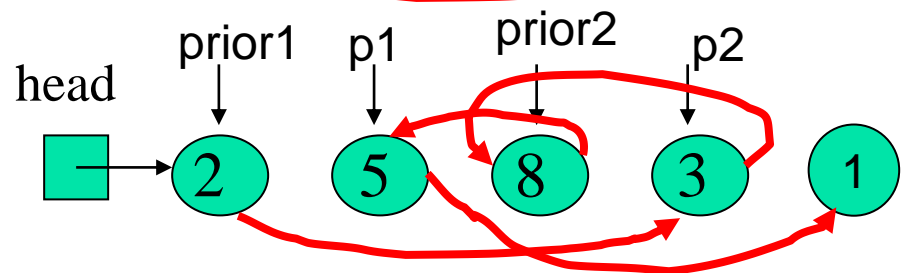
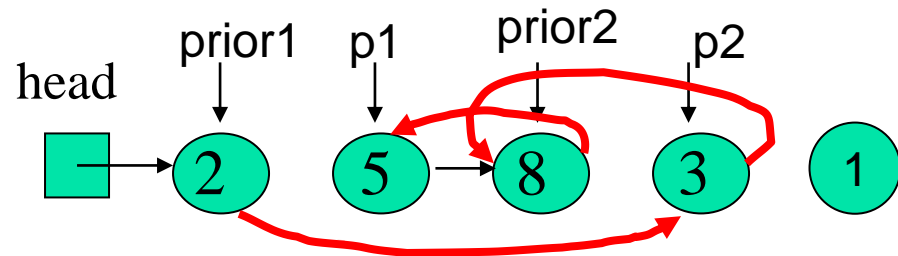
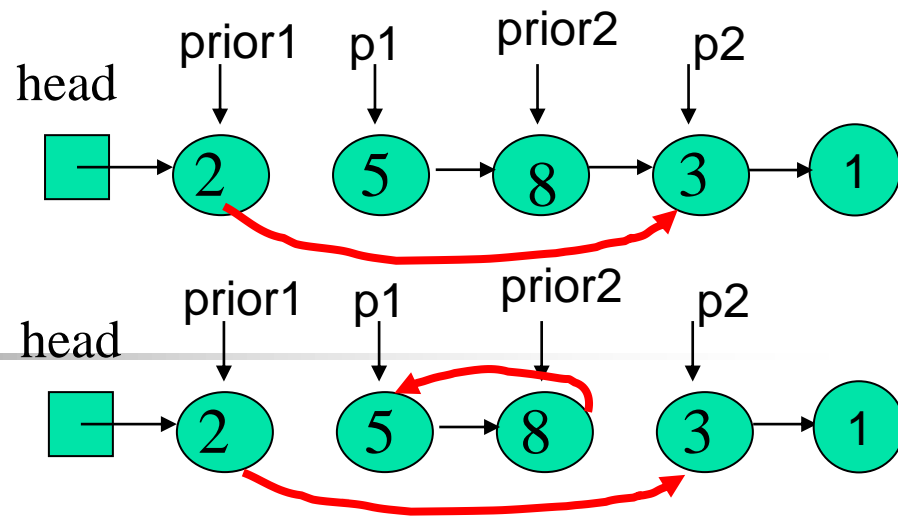
- 
- **(2) 改变结点的连接，操作较为复杂。**
 - 在数据域较为简单的情况下，多采用交换结点的数据域的方法进行链表排序。
 - 在数据域较为复杂，成员较多，采用改变链表结点之间的连接实现结点的交换的排序方法则效率较高。



```

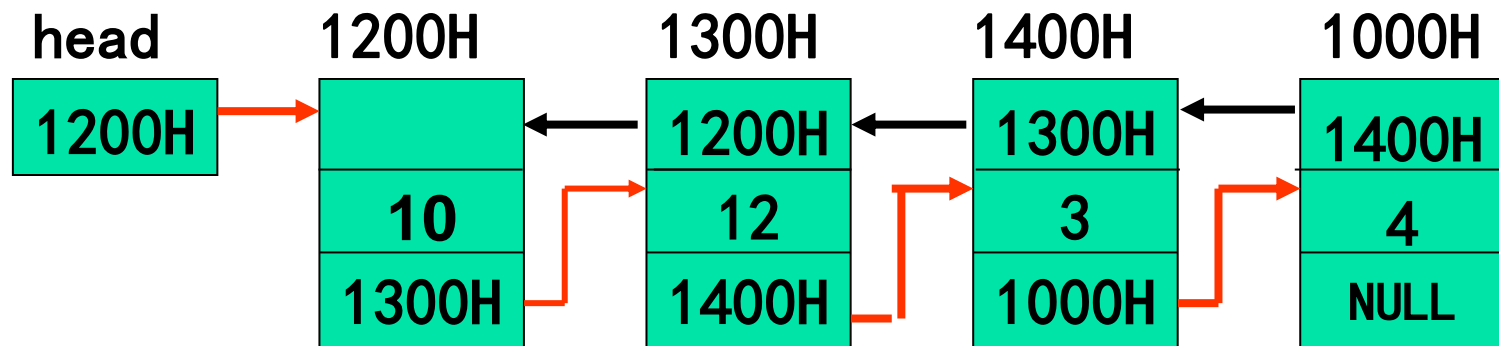
if(p1->data>p2->data){
    t=p2->next;
    prior1->next=p2;
    prior2->next=p1;
    p2->next=p1->next;
    p1->next=t;
    p2=p1;
    p1=prior1->next;
}

```



14.3.5 双向链表

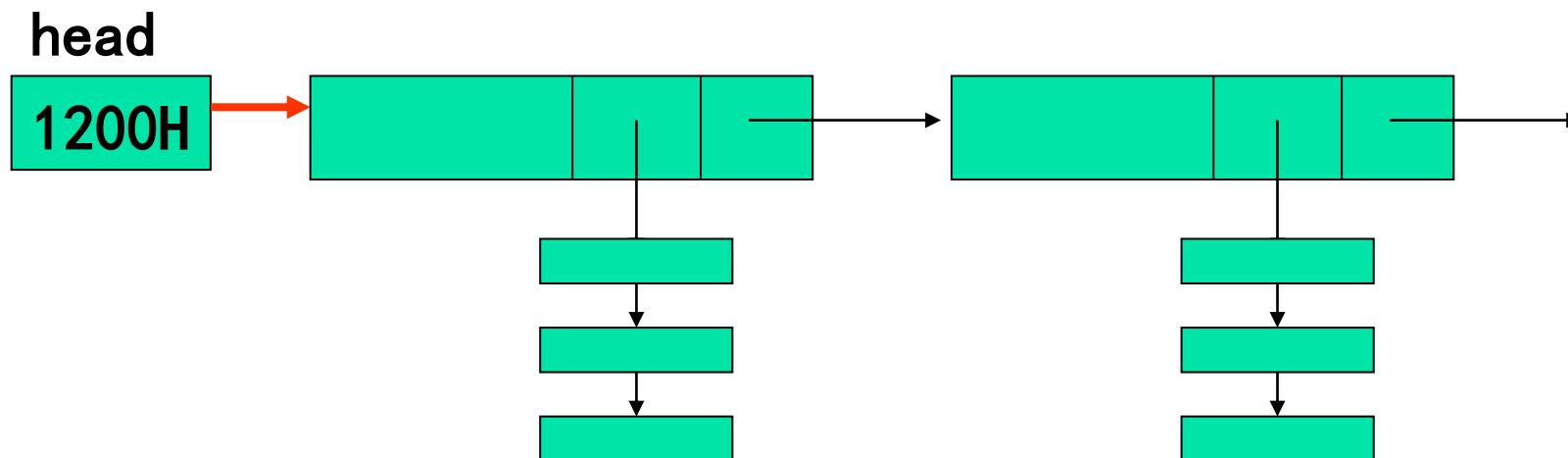
- 如果结点的指针域包含两个指针，且一个指向前一个结点，另一个指向后一个结点，这种链表称为**双向链表**。



双向链表结构

14.3.6 十字交叉链表

- 如果结点的指针域包含两个指针，且一个指向后一个结点，另一个指向另外一个链表，这种链表称为**十字交叉链表**。



十字交叉链表结构

例14.18 利用十字交叉链表和fread、fwrite函数实现学生基本信息和学生学习成绩的录入，遍历，存盘，加载等操作功能。

学生基本信息表

学号	姓名	性别	年龄	家庭住址	联系电话	备注
0001	aaa	m	18	hubei, wuhan	12345678	Abc def
0001	bbb	f	18	hunan, changsha	76545678	Aaaaaa bbb
...						
00xx	zzz	m	19	hubei, honghu	32145678	Nnn kkk

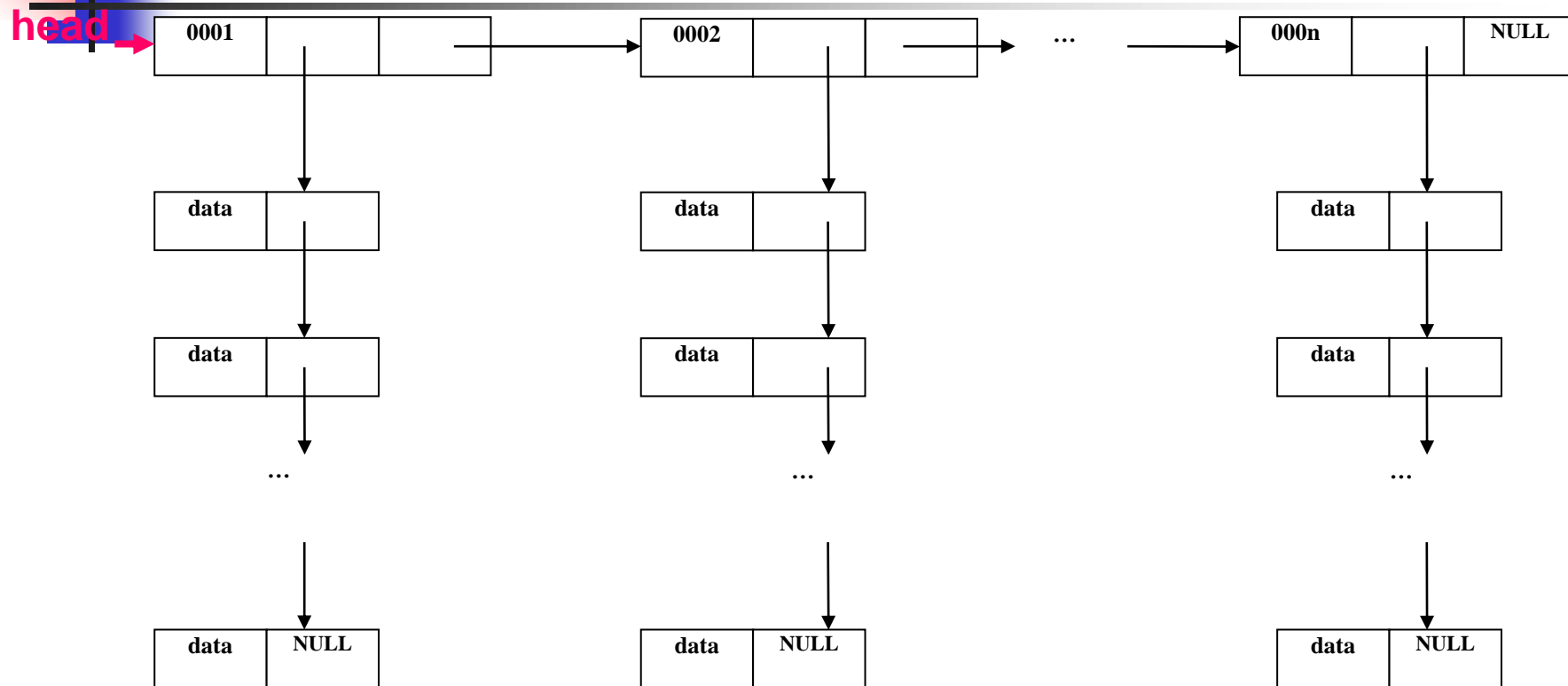
学生学习成绩表

学号	高等数学	普通物理	电工基础	演讲与口才	欧洲文化	论语初探
0001	86	85	73	×	80	×
0002	77	83	76	82	87	75
0003	87	82	81	×	×	86
...						
00xx	89	87	85	×	×	×

数据结构需求分析

- **用结构数组，编程处理比较方便。但存在明显不足：**
- 1. 学生基本信息表中的备注项长度越长，能够记载的内容就越丰富，但多数人没有备注项就会浪费大量存储。如果长度取得过短，描述能力有限，有时不能满足记载要求。
- 2. 学生选修的课程不一样，有的选课门数多，有的选课门数少；如果采用数组就必须按照选课最多的同学考虑数组的大小。同时，对课程设置的变化缺乏适应能力。编好的程序这学期可能能够满足要求，如果下个学年课程设置变了，程序也许需要进行修改。
- 3. 处理学生的增减、课程的增减效率很低。
- 综上所述，**用十字交叉链表比较合理**。十字交叉链表中由学生基本信息构成一个单向链表，链表中的每个结点描述一个同学的基本信息；同时，每个结点都有一个头指针，它指向该生的学习成绩链。

十字交叉链表



水平方向:学生基本信息链;

垂直方向:各学生课程成绩链



学生课程成绩结构类型courses的定义

该类型的结构变量可以构成纵向单向链表的结点。

```
typedef struct score_tab{  
    char    num[5];          /* 学号 */  
    char    course[20];      /* 课程名称 */  
    int     score;           /* 成绩 */  
    struct  score_tab * next;  
} courses;
```

学生基本信息结构类型studs的定义

该类型的结构变量可以构成十字交叉链表中横向学生基本信息链的结点。

```
typedef struct student_tab{  
    char    num[5];           /* 学号 */  
    char    name[10];        /* 姓名 */  
    char    sex;              /* 性别 */  
    int     age;              /* 年龄 */  
    char    addr[30];         /* 家庭住址 */  
    char    phone[12];        /* 联系电话 */  
    char*    memo;            /* 备注字段 */  
    courses *head_score;      /* 指向成绩链的头指针 */  
    struct  student_tab  *next; /* 指向下一个结点的自引用指针 */  
} studs;
```



函数原型声明

/* 输入数据并创建十字交叉链表 */

void create_cross_list(studs **head);

/* 遍历十字交叉链表 */

void traverse_cross_list(studs *head);

/* 将十字交叉链表中结点数据存盘 */

void save_cross_list(studs *head);

/* 读入已经保存的数据，创建新的十字交叉链表 */

void load_cross_list(studs **head);


十字交叉链表的创建

- **算法思路：**先创建学生基本信息链，创建完毕后通过(*head) = hp使main函数中的头指针指向学生基本信息链；然后遍历学生基本信息链，对链每个结点，询问用户是否创建该结点对应学生的课程成绩链，如果得到肯定应答，则以该结点的head_score指针为该生课程成绩链的头指针，并创建该生的课程成绩链。因此，当遍历学生基本信息链后，每个学生的课程成绩链也相应创建完毕，十字交叉链表动态创建完毕。学生基本信息链和课程成绩链都是后进先出链（栈式链），即头指针始终指向最后加入的结点。另外，由于需要修改头指针，因此必须传递头指针的地址。

```

void create_cross_list(studs **head)  /* 创建十字交叉链表的函数 */
{
    studs *hp=NULL,*p;
    courses *pcrs;
    char s[80]="",ch;
loop:
    p = (studs *)malloc(sizeof(studs)); /* 创建学生基本信息的第一个结点 */
    printf("input num,name,sex,age,addr,phone\n");
    scanf("%s %s %c %d %s %s",p->num,p->name,&p->sex,&p->age,
        p->addr,p->phone);/* 输入各项基本数据 */
    getchar(); /* 专门用于读scanf 输入中的换行符 */
    p->head_score = NULL; /* 置成绩链头指针为NULL */
    p->next = hp; /* 学生基本信息链头指针值赋给结点的next 域 */
    hp = p; /* 头指针指向新创建的结点 */
    printf("continue input data of students? yes or no\n");
    ch = getchar();    getchar(); /* 是否继续创建学生基本信息结点 */
    if(ch == 'y' || ch == 'Y')
        goto loop; /* 是，创建下一个学生基本信息结点 */
    (*head) = hp; /* 调用函数中的head 头指针指向新创建的学生基本信息链 */
}

```

```

while(p != NULL){ /* 循环创建每个学生的学习成绩链 */
    printf("input %s student's score? yes or no?\n",p->num);
    ch = getchar(); getchar();
    while(ch == 'y' || ch == 'Y'){
        pcrs = (courses *)malloc(sizeof(courses));/* 创建成绩结点 */
        printf("input course and score\n");
        scanf("%s %d", pcrs->course,&pcrs->score);/* 输入课程名和成绩 */
        getchar(); /* fflush(stdin); */
        strcpy(pcrs->num, p->num); /* 学号成员赋值 */
        pcrs->next = p->head_score; /* 头指针值赋给新结点的指针域 */
        p->head_score = pcrs; /* 头指针指向新结点 */
        printf("input %s student's score? yes or no?\n",p->num);
        ch = getchar(); getchar();/*是否继续创建成绩结点 */
    }
    p = p->next; /* p指向学生基本信息链的下一个学生 */
}
}

```



十字交叉链表的遍历

算法思路：首先将头指针值赋给学生基本信息链的遍历指针p；先遍历学生基本信息链，在遍历过程中，对链中每个结点，先输出结点数据域的内容，然后从该结点的head_score开始，将p->head_score赋给学生课程成绩链的遍历指针pcrs，让pcrs从课程成绩链的链头开始遍历课程成绩链的每个结点并输出课程成绩链结点数据域的内容。因此，当学生基本信息链遍历完毕时，链中每个结点指向的课程成绩链也全部遍历完毕。

```
void traverse_cross_list(studs *head) /* 遍历十字交叉链表的函数 */
```

```
{
```

```
    studs * p = head;
```

```
    courses * pcrs;
```

```
    while(p != NULL){ /* 遍历学生基本信息链 */
```

```
        printf("%s\t%s\t%c\t%d\t%s\t%s\n",p->num,p->name,p->sex,  
                p->age,p->addr,p->phone);
```

```
        pcrs = p->head_score;
```

```
        while(pcrs != NULL){ /* 遍历学生成绩链 */
```

```
            printf("%s\t%s\t%d\n",pcrs->num,pcrs->course,pcrs->score);
```

```
            pcrs = pcrs->next;
```

```
        }
```

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

```
    }
```

```
}
```



学生基本信息和学生课程成绩的保存

创建两个数据文件`student.dat`和`score.dat`,
学生基本信息保存在`student.dat`中,
学生课程成绩保存在`score.dat`中

(1) 说明两个文件指针`out1` 和 `out2`

两个遍历指针`p` (遍历学生基本信息链)

`pcrs` (遍历学生课程成绩链)

(2) 以二进制写方式打开两个数据文件



(3) 对十字交叉链表进行遍历。

对学生基本信息链的每个结点，

①将p指向的结点内容写入文件student.dat;

②pcrs = p->head_score将该结点指向的学生课程成绩链的头指针赋给pcrs

③让pcrs遍历学生课程成绩链，将pcrs指向的学生课程成绩链结点内容写入到score.dat文件

(4) 关闭两个文件

```

void save_cross_list(studs *head) /* 保存十字交叉链表的函数 */
{
    FILE *out1,*out2;
    studs * p = head;
    courses * pcrs;
    if((out1 = fopen("c:\\student.dat","wb")) == NULL) /* 打开基本信息文件 */
        exit(-1);
    if((out2 = fopen("c:\\score.dat","wb")) == NULL) /* 打开成绩文件 */
        exit(-1);
    while(p != NULL){
        fwrite(p,sizeof(studs),1,out1);/* 写学生基本信息记录 */
        pcrs = p->head_score; /* 成绩遍历指针指向成绩链链头 */
        while(pcrs != NULL){ /* 遍历学生成绩链 */
            fwrite(pcrs,sizeof(courses),1,out2); /* 写成绩记录 */
            pcrs = pcrs->next; /* 指向下一个成绩结点 */
        }
        p =p->next; /* 指向下一个学生基本信息结点 */
    }
    fclose(out1); /* 关闭基本信息文件 */
    fclose(out2); /* 关闭成绩文件 */
}

```

读取student.dat和score.dat, 构造十字交叉链表

- (1) 说明两个文件指针in1 和 in2
两个遍历指针p（遍历学生基本信息链）
pcrs（遍历学生课程成绩链）
- (2) 以二进制读方式打开两个数据文件
- (3) 读学生基本信息文件student.dat，创建横向的学生基本信息链
- (4) 创建课程成绩链的结点并使pcrs指向它，再从score.dat文件中读一条记录并将其赋给pcrs所指向的结点；然后遍历学生基本信息链，查找链中学号值与pcrs所指向结点的学号值相等的结点。找到后，以该结点的head_score指针成员为课程成绩链的头指针，以后进先出的方式将课程成绩新结点加入到以head_score为头指针的课程成绩链中。

```

void load_cross_list(studs **head) /* 读入十字交叉链表的函数 */
{
    FILE *in1,*in2;
    studs *hp=NULL,*p;
    courses * pcrs;
    if((in1 = fopen("c:\\student.dat","rb")) == NULL) /* 打开基本信息文件 */
    {
        exit(-1);
    }
    if((in2 = fopen("c:\\score.dat","rb")) == NULL) /* 打开成绩文件 */
    {
        exit(-1);
    }
    while(!feof(in1)){
        p = (studs *)malloc(sizeof(studs)); /* 创建基本信息结点 */
        fread(p,sizeof(studs),1,in1); /* 读一条基本信息记录到结点中 */
        if(!feof(in1)){
            p->head_score = NULL; /* 置成绩链头指针为NULL */
            p->next = hp; /* 学生基本信息链头指针值赋给结点的next 域 */
            hp = p; /* 头指针指向新创建的结点 */
        }
    }
}

```



```

(*head) = hp; /* 调用函数中的head 头指针指向新创建的学生基本信息链 */
while(!feof(in2)){
    pcrs = (courses *)malloc(sizeof(courses)); /* 创建新成绩结点 */
    fread(pcrs,sizeof(courses),1,in2); /* 读一条成绩记录到新结点 */
    if(!feof(in2)){
        p = (*head);/*遍历指针p指向学生基本信息链的链头 */
        while(p != NULL){ /*遍历学生基本信息链 */
            if(!strcmp(p->num , pcrs->num)){ /* 查找相同学号的结点 */
                pcrs->next = p->head_score; /* 找到，新结点加入成绩链 */
                p->head_score = pcrs;
                break;
            }
            else
                p = p->next; /* 没找到，转下一结点 */
        }
    }
}
fclose(in1); /* 关闭基本信息文件 */
fclose(in2); /* 关闭成绩文件 */
}

```