第9章 建立自己的数据类型（3）

复习回顾

- 上次课的内容：
  - ◆递归输出全排列
  - ◆结构体变量初始化
  - ◆结构体数组
  - ◆结构体指针
  - ◆结构体作函数参数
  - ◆一个学期快过去了，你的学习热情是否一如开学往昔？

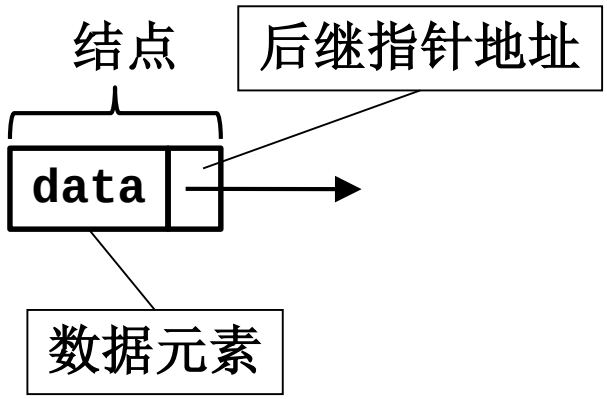
你中枪了吗？

单向链表与猴子捞月

- 猴子之间的连接关系
  - 最上面的猴子倒挂在树枝上
  - 上面的猴子紧紧抓住下面一只猴子的腿
  - 最下面的一只猴子两手悬空
- 链表结点的连接关系
  - 表头指针（树上的猴子腿）
  - 上一个节点保存下一节点的地址（抓住）
  - 表尾节点的地址域为空（两手空空）

链表结点的结构

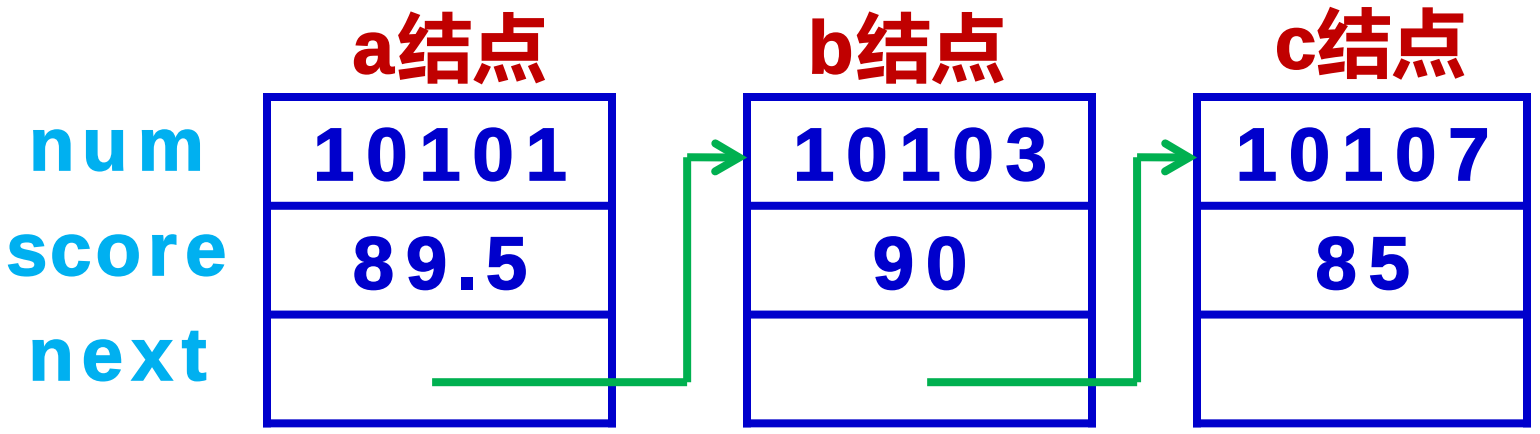
```
struct Student
{
    int num;
    float score;
    struct Student *next;
}a,b,c;
```



	a结点	b结点	c结点
num	?	?	?
score	?	?	?
next	?	?	?

建立简单的静态链表

例： 建立一个如图所示的简单链表，它由3个学生数据的结点组成，要求输出各结点中的数据。

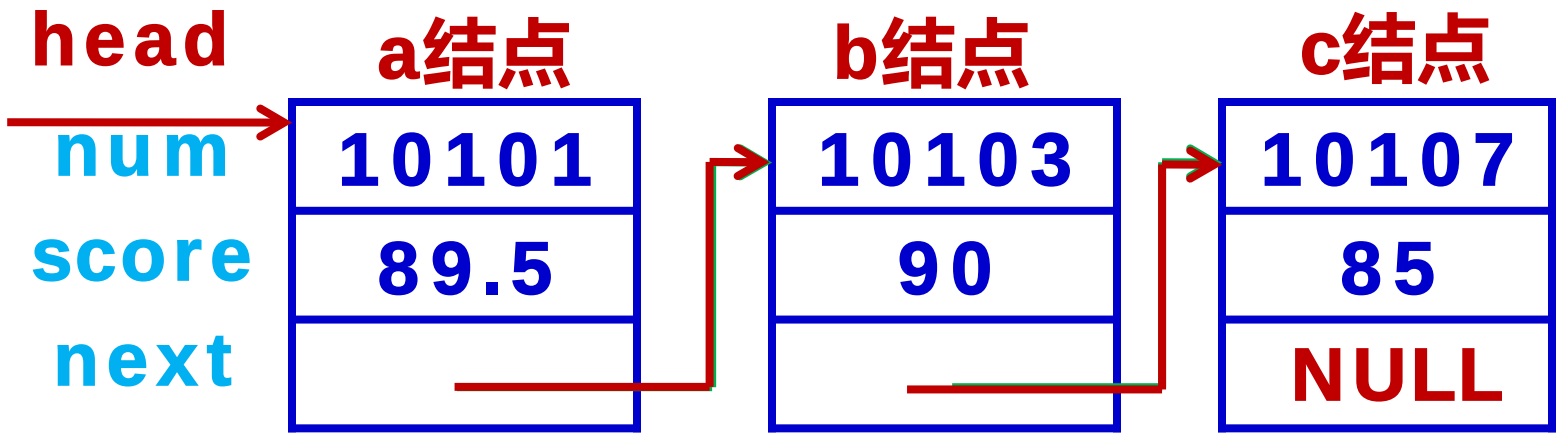


建立简单的静态链表

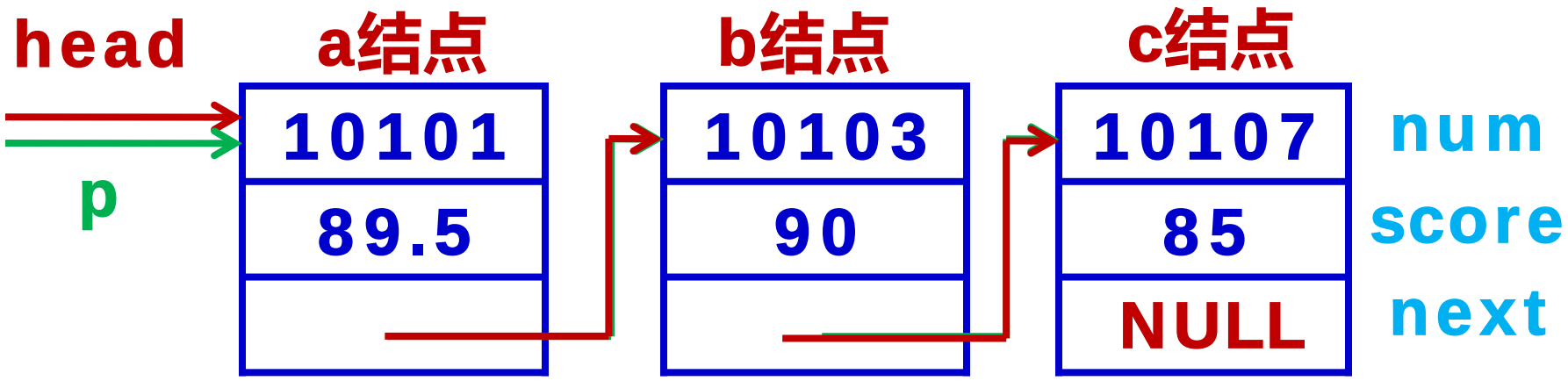
➢解题思路：

head=&a;　a.next=&b;

b.next=&c;　c.next=NULL;



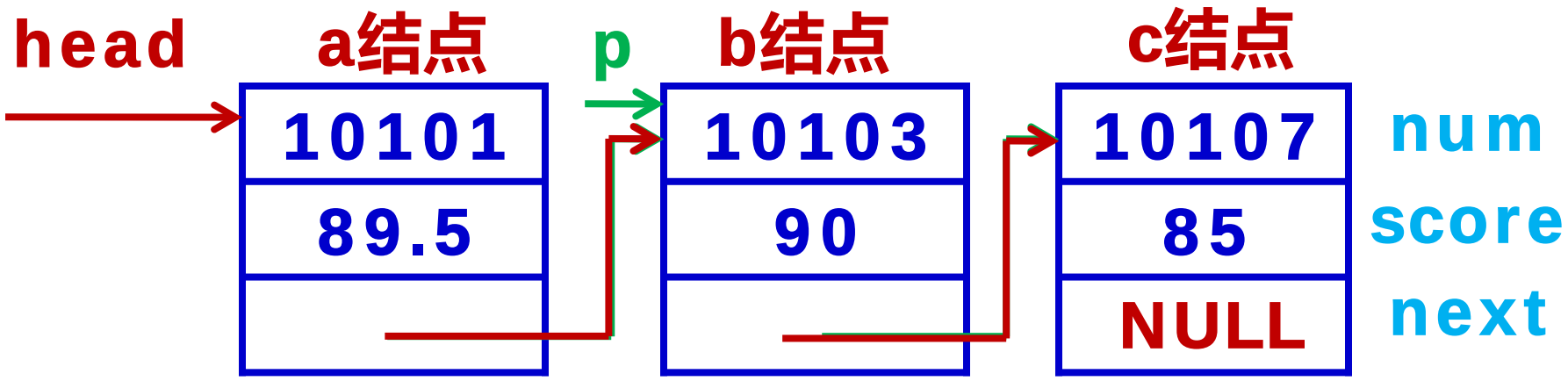
```
1. #include <stdio.h>
2. struct Student
3. { int num; float score; struct Student *next; };

4. int main()
5. {
6.     struct Student a, b, c, *head, *p;

7.     a.num = 10101;  a.score = 89.5;
8.     b.num = 10103;  b.score = 90;
9.     c.num = 10107;  c.score = 85;
10.
11.    head = &a;       a.next = &b;
12.    b.next = &c;     c.next = NULL;

13.    p=head;    //实现链表输出
14.    do {
15.        printf("%ld%5.1f\n",p->num,p->score);
16.        p = p->next;
17.    } while (p!=NULL);
18.
19.    return 0;
20.}
```




```
p=head;
do {
  printf("%ld%5.1f\n",p->num,p->score);
  p=p->next;
}while(p!=NULL);
```

相当于p=&b;

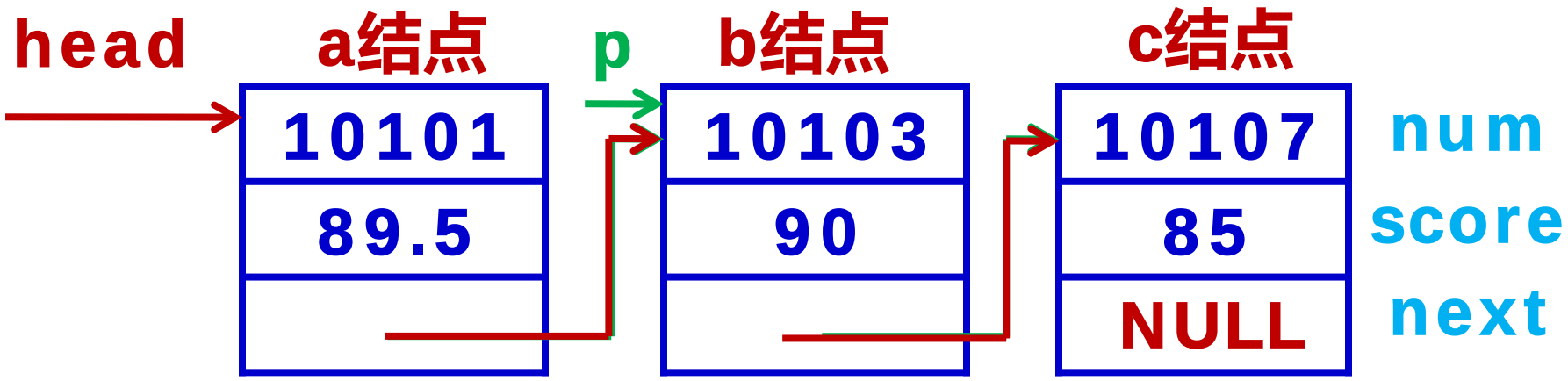
```
10101  89.5
```




```
p=head;
do {
  printf("%ld%5.1f\n",p->num,p->score);
  p=p->next;
}while(p!=NULL);
```

相当于p=&b;

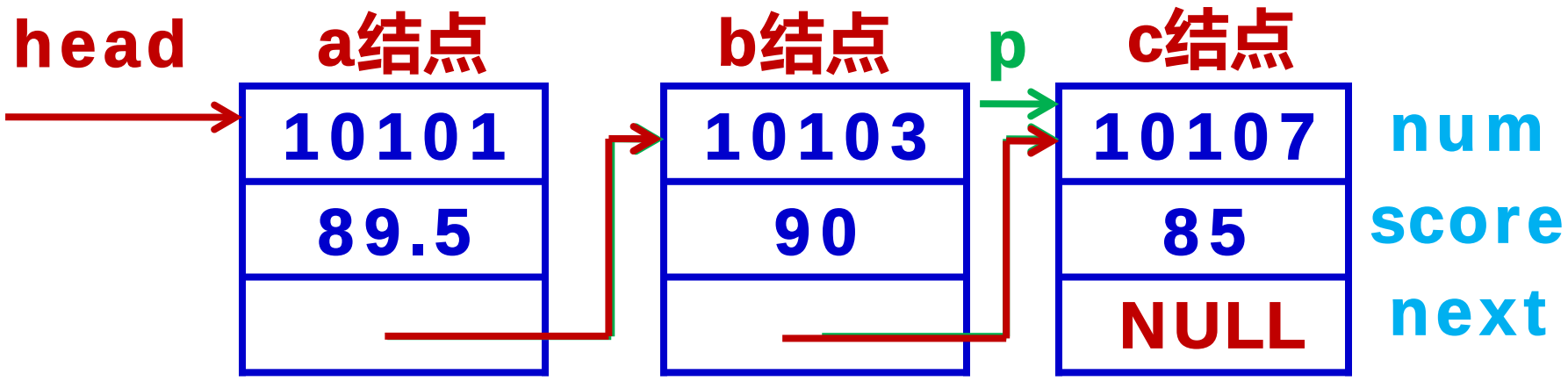
```
10101  89.5
```




```
p=head;
do {
  printf("%ld%5.1f\n",p->num,p->score);
  p=p->next;
}while(p!=NULL);
```

相当于p=&c;

```
10101   89.5
10103   90.0
```

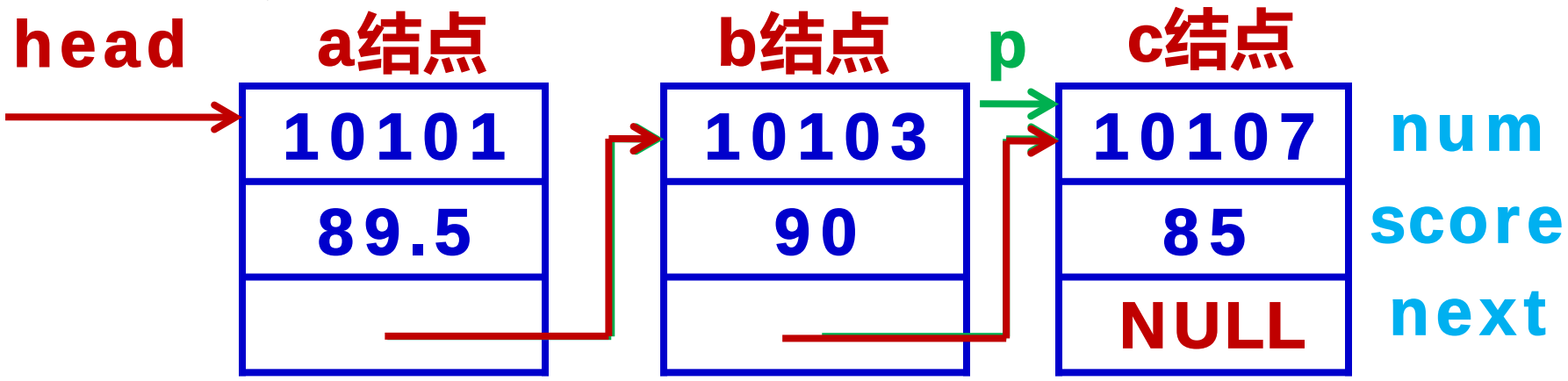



```
p=head;
do {
  printf("%ld%5.1f\n",p->num,p->score);
  p=p->next;
}while(p!=NULL);
```

相当于p=&c;

```
10101   89.5
10103   90.0
```

静态链表




```
p=head;
do {
  printf("%ld%5.1f\n",p->num,p->score);
  p=p->next; 相当于p=NULL;
 }while(p!=NULL);
```

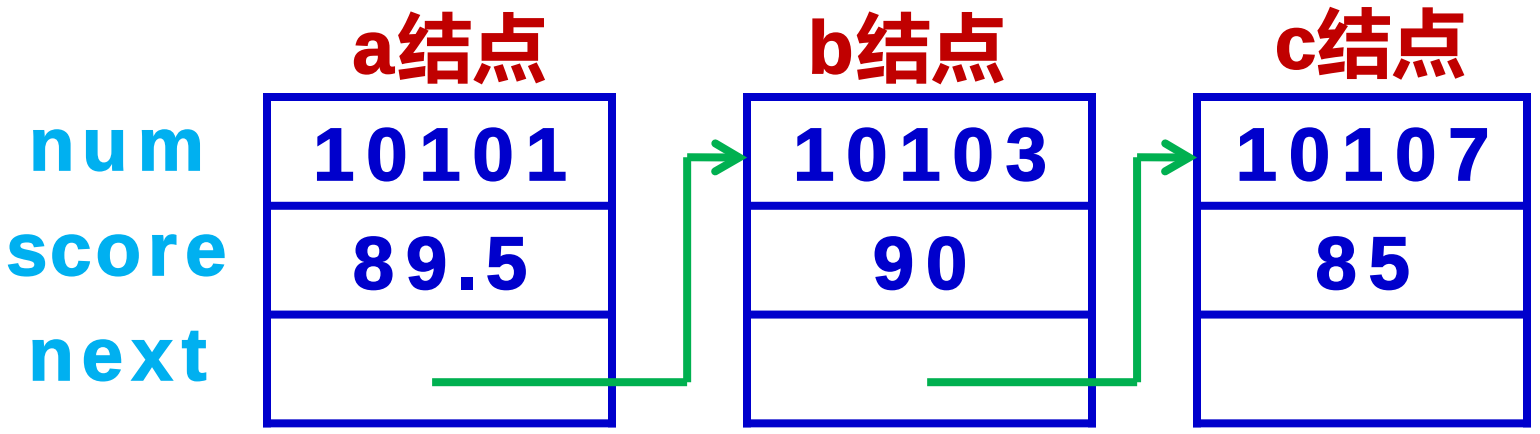
```
10101   89.5
10103   90.0
10107   85.0
```

建立动态链表

- 所谓建立动态链表是指在程序执行过程中从无到有地建立起一个链表，即一个一个地开辟结点和输入各结点数据，并建立起前后相链的关系。
- 两个步骤
  1. 利用动态内存管理函数分配结点空间
  2. 将已经生成的结点彼此连接起来

建立简单动态链表

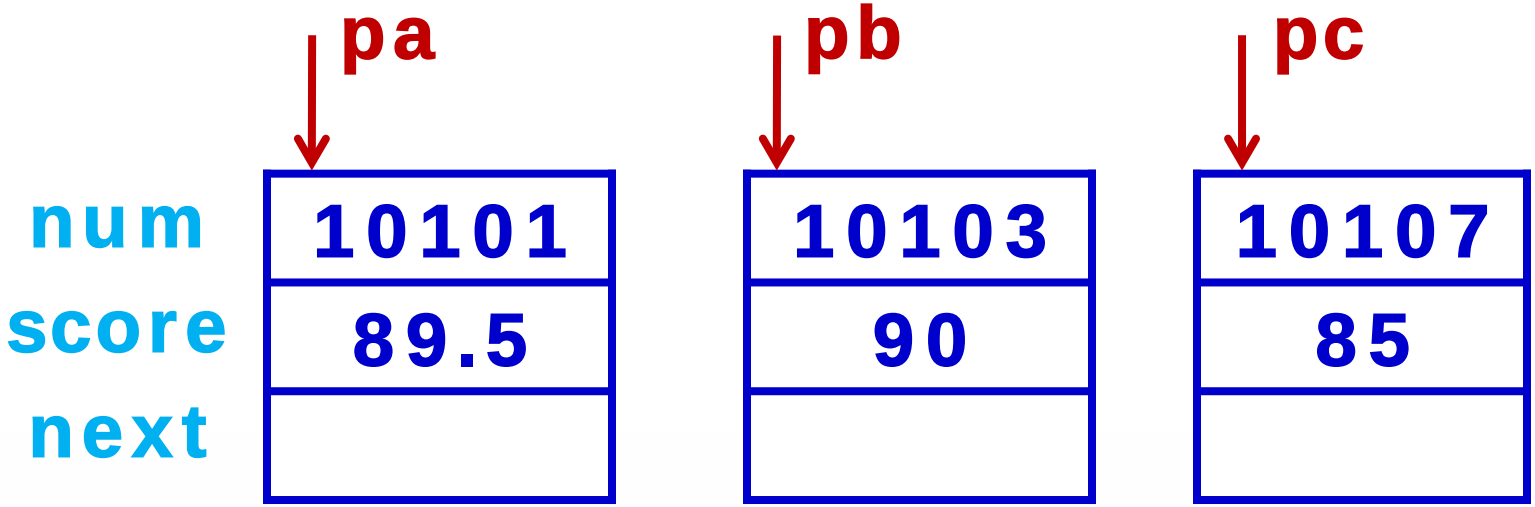
例： 动态建立一个如图所示的简单链表，它由3个学生数据的结点组成



建立简单的动态链表

- 解题思路：

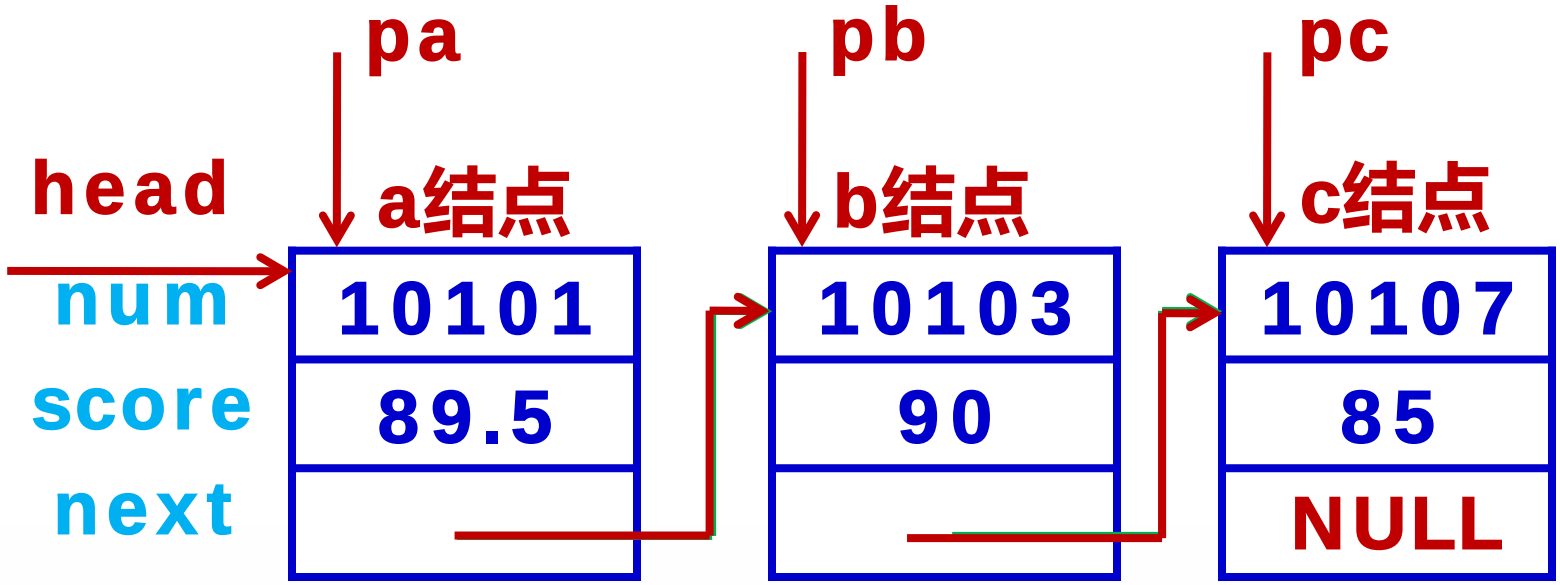
```
pa=(struct Student*)malloc(sizeof(struct Student));
pa->num=10101; pa->score = 89.5;

pb=(struct Student*)malloc(sizeof(struct Student));
pb->num=10103; pb->score = 90;

pc=(struct Student*)malloc(sizeof(struct Student));
pc->num=10107; pa->score = 85;
```



建立简单的动态链表

➢解题思路：

```
head=pa;        pa->next=pb;
pb->next=pc;    pc->next=NULL;
```



重复利用指针来建立链表

- 例：写一函数建立一个有3名学生数据的单向动态链表。（输入学号为0时结束创建）
- 基本思路：
  - 实际中链表的长度无法固定，初始化时为每个结点都建立一个指针变量显然不经济。
  - 更多的时候，往往是少数几个指针交替使用来建立链表。一般情况下，需要用到3个指针变量。

➢解题思路：

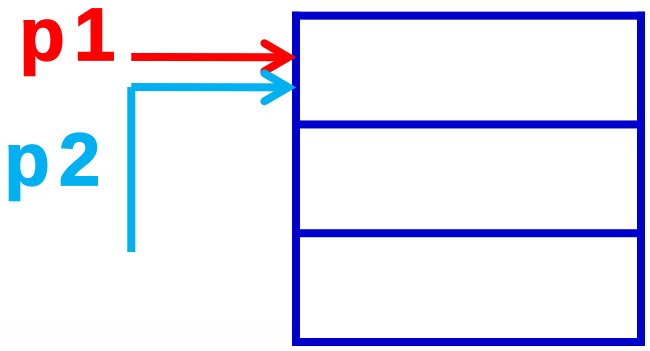
◆定义3个指针变量：head,p1和p2，它们都是用来指向struct Student类型数据

```
struct Student *head,*p1,*p2;
```

➢解题思路：

◆用malloc函数开辟第一个结点，并使p1和p2指向它

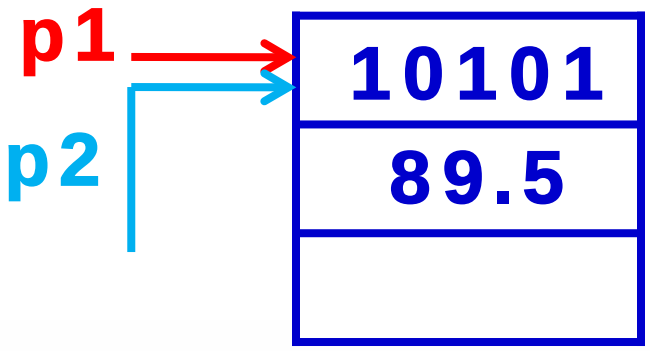
```
p1=p2=(struct Student*)malloc(LEN);
```



➢解题思路：

◆读入一个学生的数据给p1所指的第一个结点

```
scanf("%ld,%f",&p1->num,&p1->score);
```

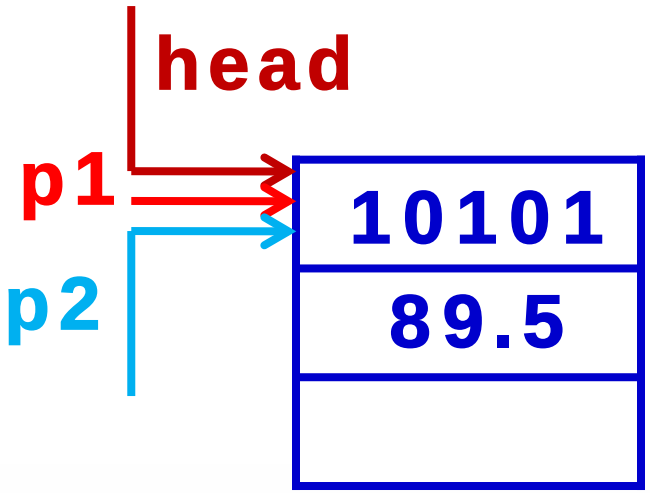


➢解题思路：

◆读入一个学生的数据给p1所指的第一个结点

```
scanf("%ld,%f",&p1->num,&p1->score);
```

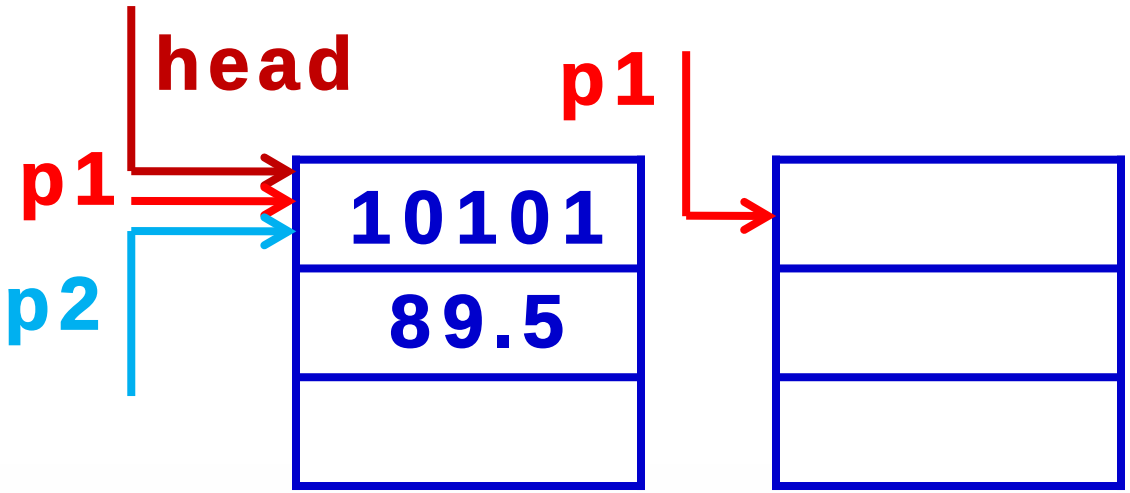
◆使head也指向新开辟的结点



➢解题思路：

◆再开辟另一个结点并使p1指向它，接着输入该结点的数据

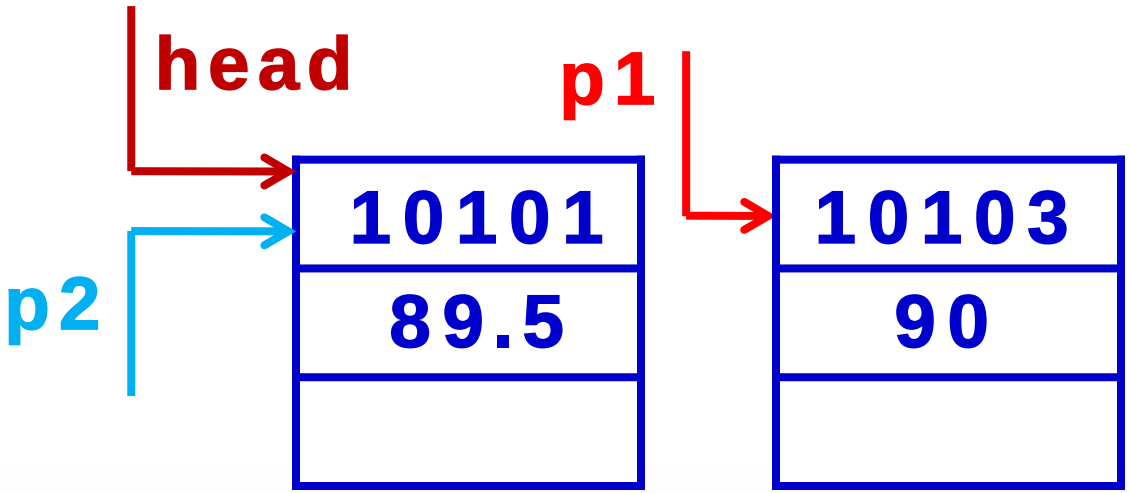
```
p1=(struct Student*)malloc(LEN);
```



➢解题思路：

◆再开辟另一个结点并使p1指向它，接着输入该结点的数据

```
scanf("%ld,%f",&p1->num,&p1->score);
```

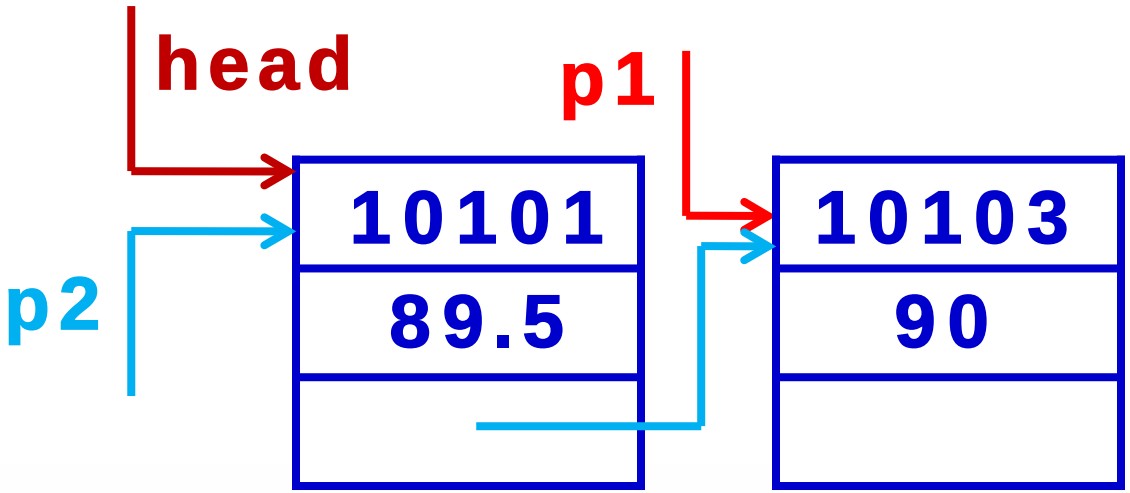


➢解题思路：

◆使第一个结点的next成员指向第二个结点，即连接第一个结点与第二个结点

```
p2->next=p1;
```

◆使p2指向刚才建立的结点

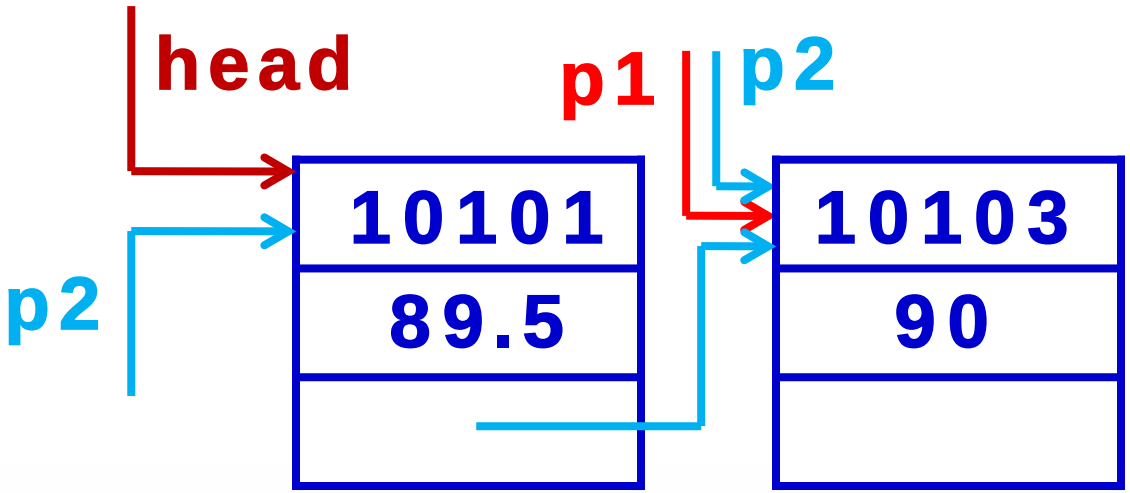


➢解题思路：

◆使第一个结点的next成员指向第二个结点，即连接第一个结点与第二个结点

```
p2->next=p1;
```

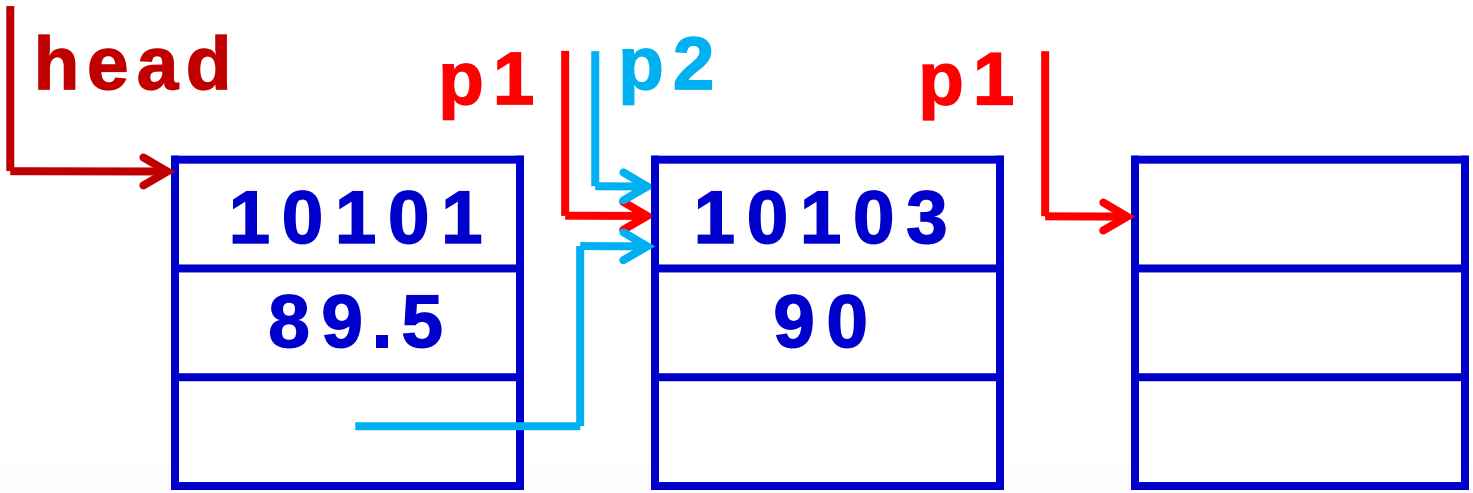
◆使p2指向刚才建立的结点 `p2=p1;`



➢解题思路：

◆再开辟另一个结点并使p1指向它，接着输入该结点的数据

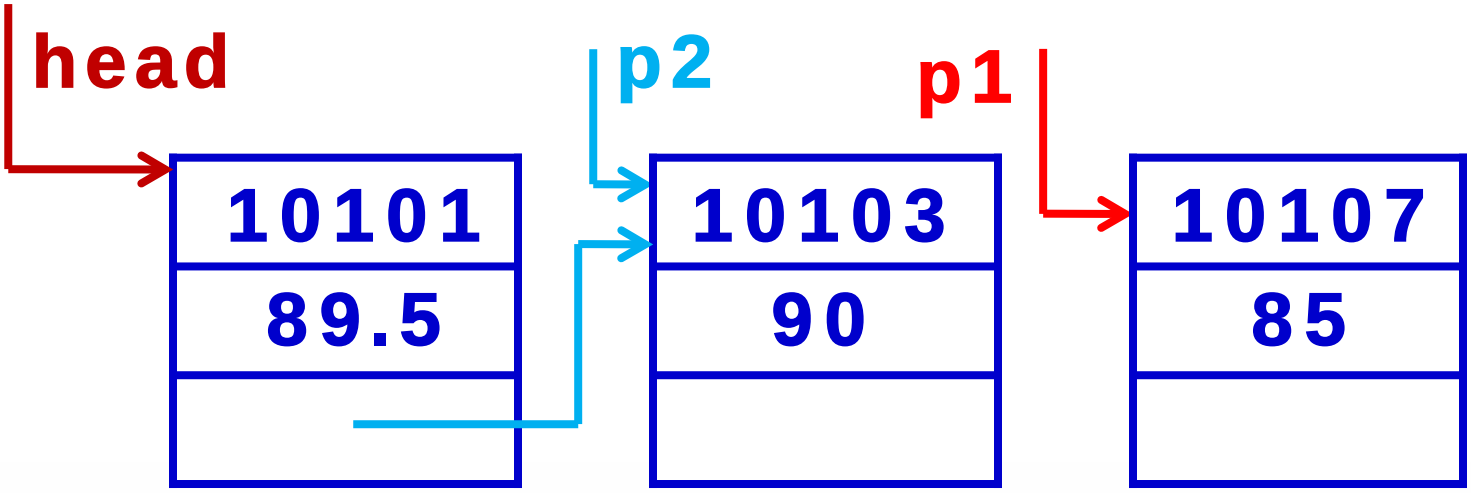
```
p1=(struct Student*)malloc(LEN);
```



➢解题思路：

◆再开辟另一个结点并使p1指向它，接着输入该结点的数据

```
scanf("%ld,%f",&p1->num,&p1->score);
```

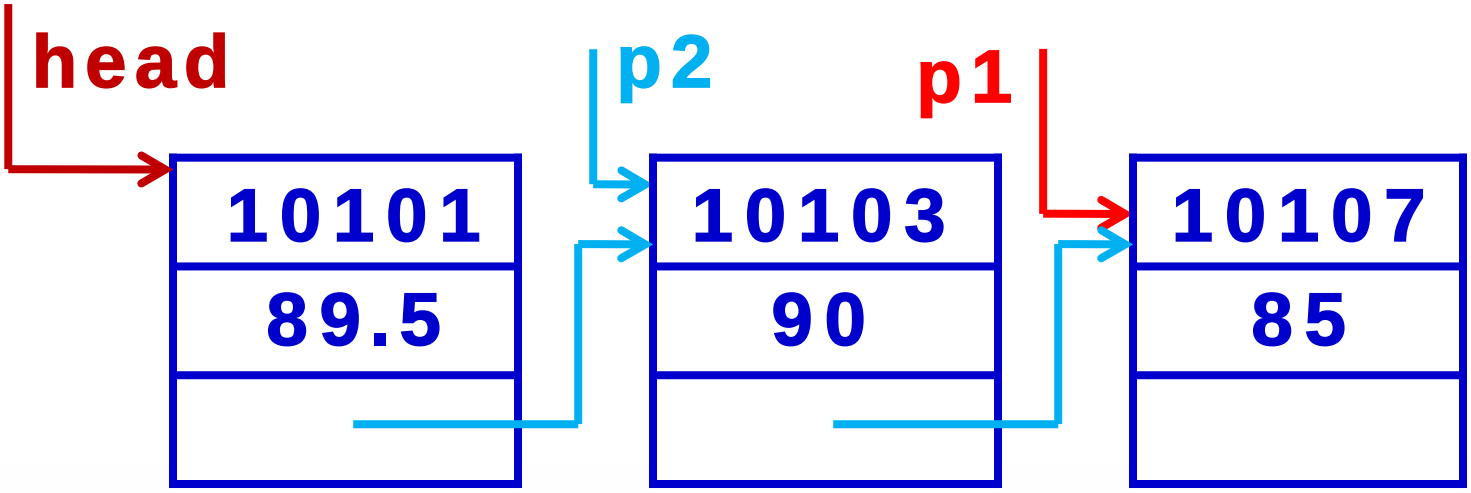


➢解题思路：

◆使第二个结点的next成员指向第三个结点，即连接第二个结点与第三个结点

```
p2->next=p1;
```

◆使p2指向刚才建立的结点

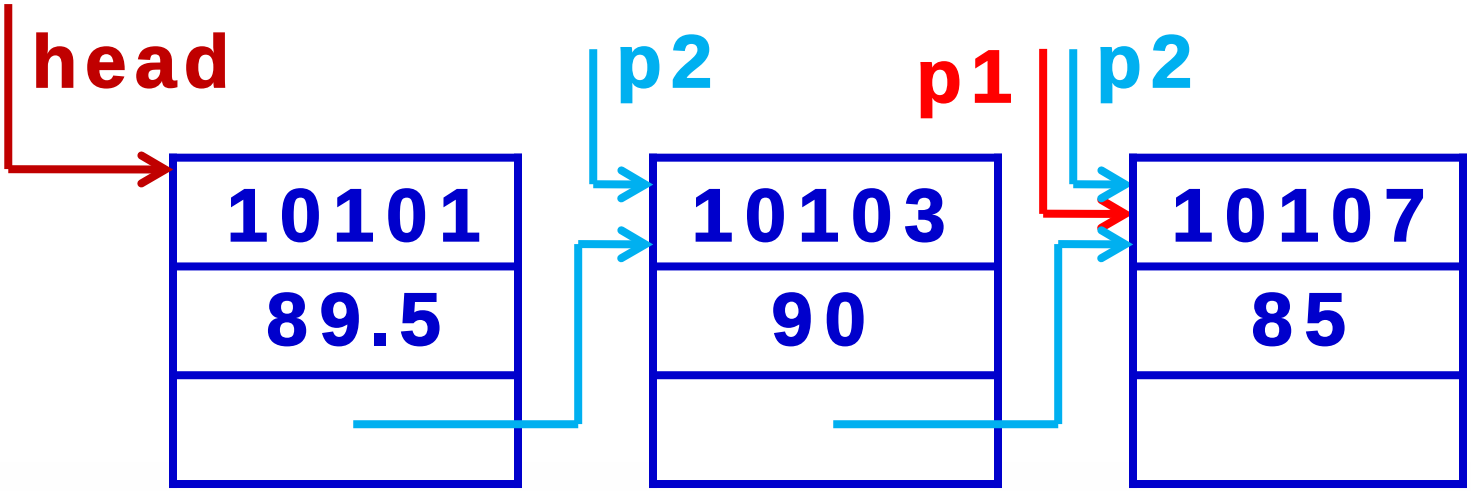


➢解题思路：

◆使第二个结点的next成员指向第三个结点，即连接第二个结点与第三个结点

```
p2->next=p1;
```

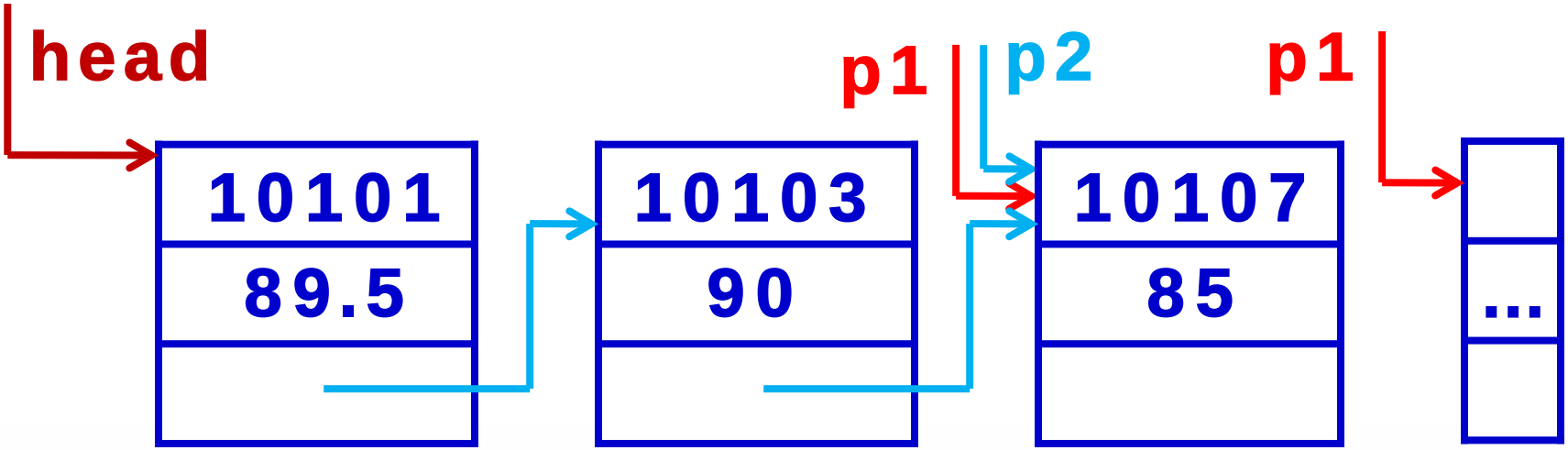
◆使p2指向刚才建立的结点 `p2=p1;`



➢解题思路：

◆再开辟另一个结点并使p1指向它，接着输入该结点的数据

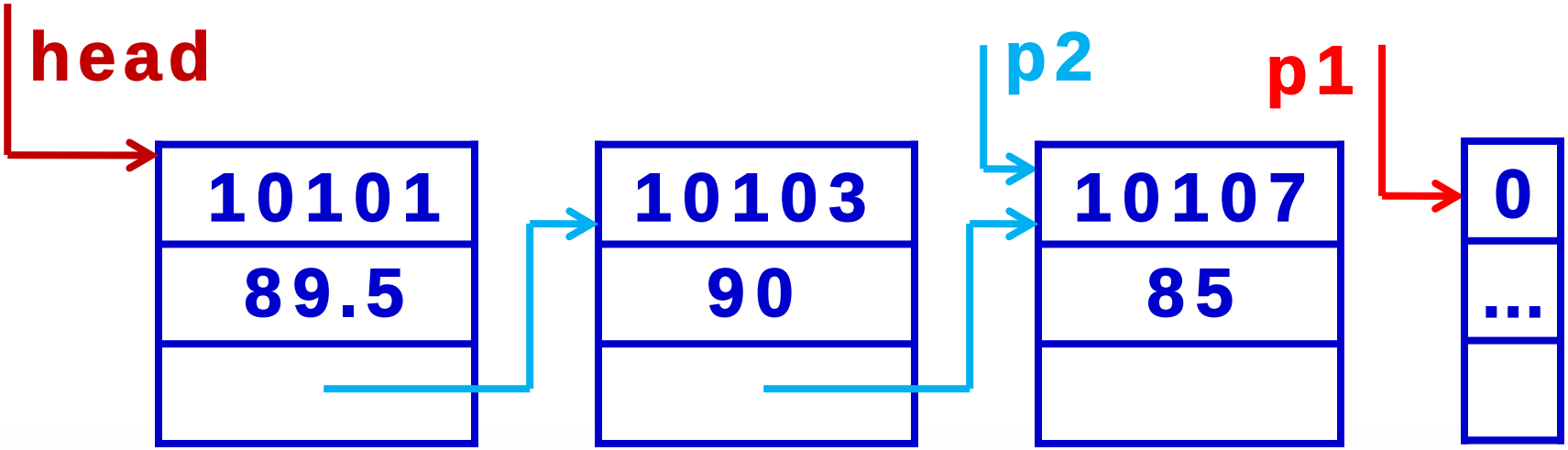
```
p1=(struct Student*)malloc(LEN);
```



➢解题思路：

◆再开辟另一个结点并使p1指向它，接着输入该结点的数据

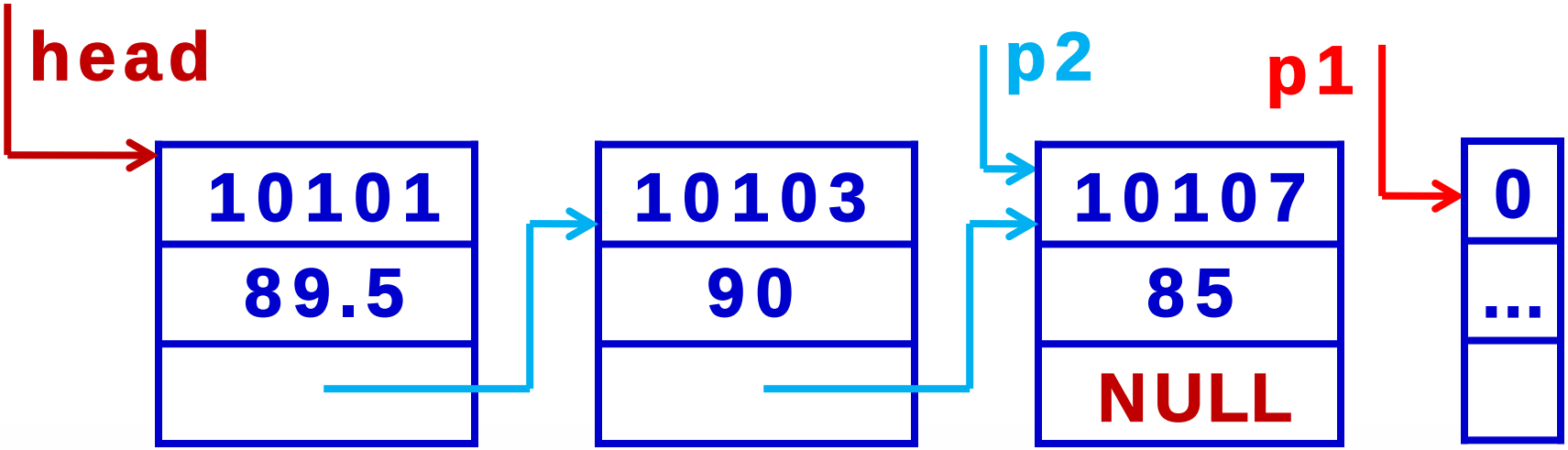
```
scanf("%ld,%f",&p1->num,&p1->score);
```



➢解题思路：

◆输入的学号为0，表示建立链表的过程完成，该结点不应连接到链表中

```
p2->next=NULL;
```



```
#include <stdio.h>
#include <stdlib.h>
#define LEN sizeof(struct Student)
struct Student
{
    long num;
    float score;
    struct Student *next;
};
int n;
```

struct Student类型数据的长度

```
struct Student *creat(void)
{
    struct Student *head,*p1,*p2;
    n = 0;
    p1 = p2 = (struct Student*) malloc(LEN);
    scanf("%ld,%f",&p1->num,&p1->score);
    head = NULL;
    while (p1->num!=0)
    {
        n = n+1;
        if (n==1)
            head = p1;
        else
            p2->next = p1;
        p2 = p1;
        p1 = (struct Student*)malloc(LEN);
        scanf("%ld,%f", &p1->num, &p1->score);
    }
    p2->next = NULL;
    return(head);
}
```

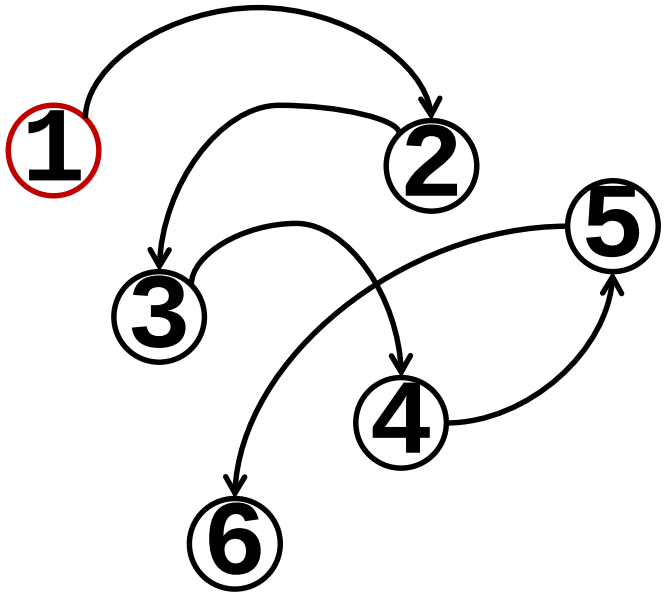
- 用p2和p1连接两个结点
- p1总是开辟新结点
- p2总是指向当前最后结点
- 为链表不断链入新结点的过程就是不断将p1和p2向后移动的过程

```
int main()
{
    struct Student *pt;
    pt = creat();
    printf("%ld %5.1f\n",
                    pt->num,pt->score);
    return 0;
}
```

```
10101  89.5
```

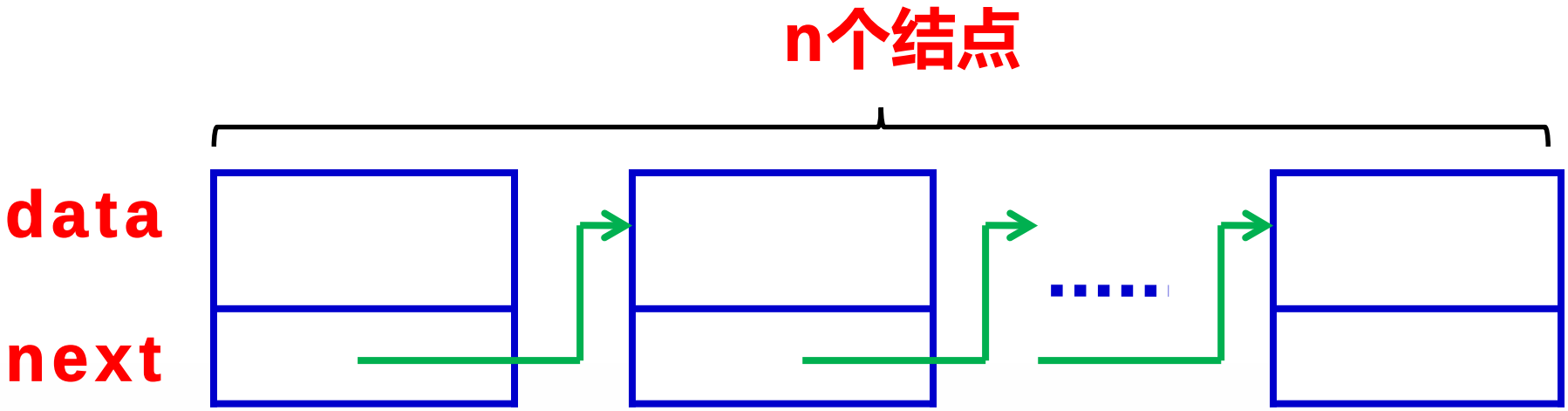
更多的链表操作

- 创建n个结点的动态链表
- 往链表插入结点
- 从链表删除结点
- 两个有序链表的合并
- ……



建立含有n个结点的动态链表

例： 建立一个如图所示的简单链表，它由n个结点组成，结点数据是一个整数，由用户输入。链表建立完成后，要求输出各结点中的数据。

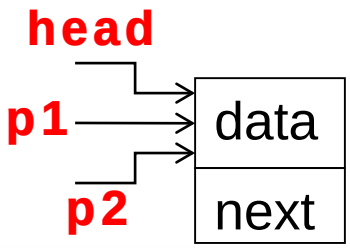


建立含有n个结点的动态链表

解题思路：一般需三个指针，各司其职

- head，指向表头结点（领头羊）
- p1，临时指针变量（负责收集新结点）
- p2，临时指针变量（断后，负责链接新结点）

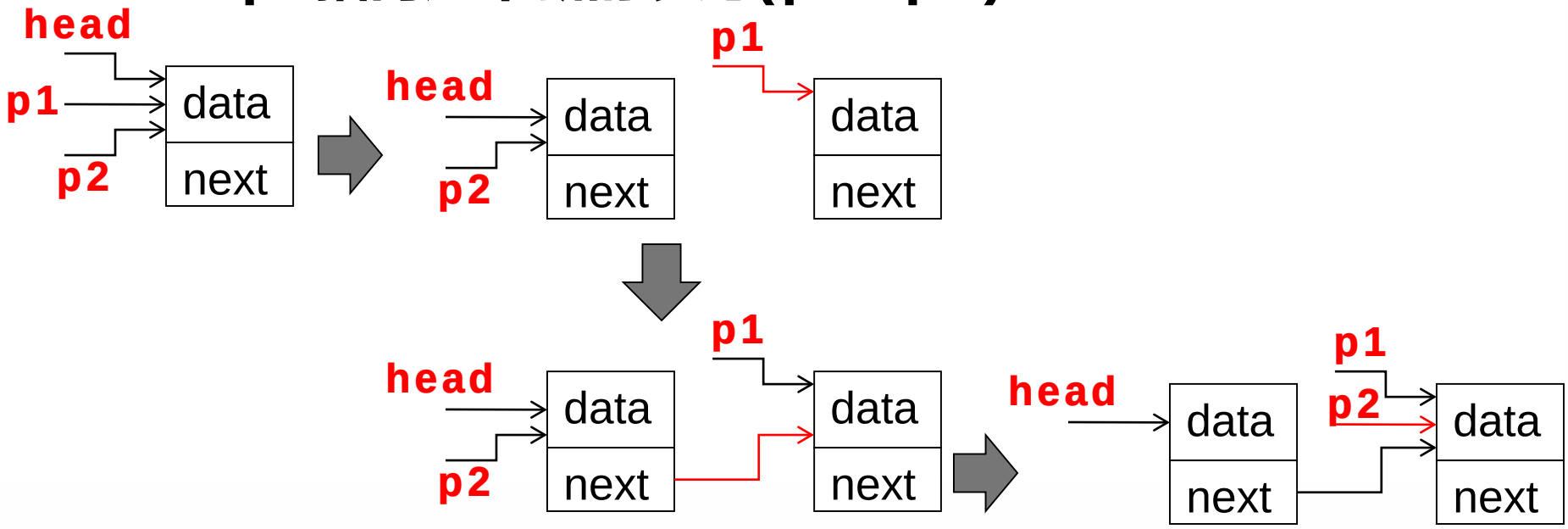
1. 若n=0，空链表，直接head=NULL。
2. 若n=1，由malloc新建一个结点
   - ✓ p1收集这个结点，故指向它
   - ✓ 这个结点是表头，故head也指向它
   - ✓ 这个结点还是表尾，故p2也指向它



建立含有n个结点的动态链表

解题思路：一般需三个指针，各司其职

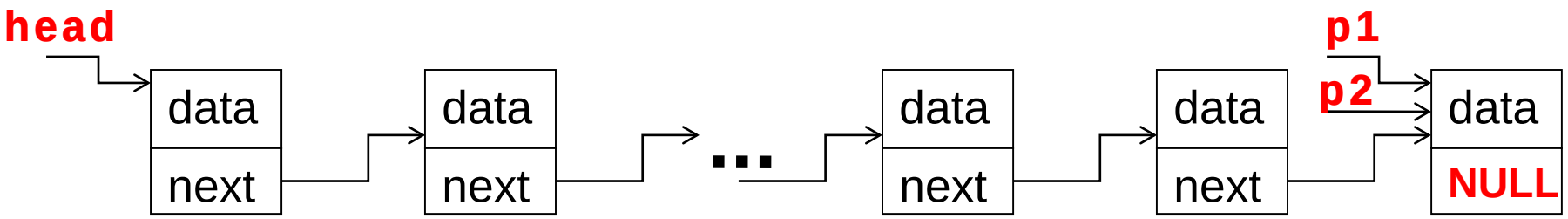
3. 若n ≥ 2，重复：由p1指向malloc新建的结点，p2将这个结点连进链表(p2->next=p1)，然后p2指向这个新的表尾(p2=p1)



建立含有n个结点的动态链表

解题思路：一般需三个指针，各司其职

4. 如果链表已经满足结束条件（即达到事前预定的长度，或接收到一个结束标记），那么循环操作就将终止。新的结点不再被链入结点中。这个时候，将NULL赋给p2->next，表示该结点为整个链表的尾结点。



建立含有n个结点的动态链表

- 源代码

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #define LEN sizeof(struct node)

4. struct node
5. {
6.     int data;
7.     struct node * next;
8. };
```

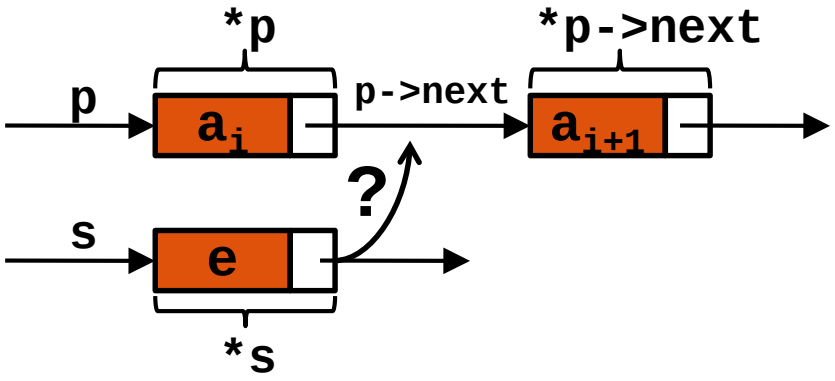
```
9.  struct node * initialize(int num)
10. {
11.     struct node * head = NULL;
12.     struct node * p1, * p2;
13.     if (num > 0)
14.     {
15.         p1 = p2 = (struct node*)malloc(LEN); //表头结点
16.         scanf("%d", &p1->data);
17.         head = p1;
18.         while (num > 1)
19.         {
20.             p1 = (struct node*)malloc(LEN);
21.             scanf("%d", &p1->data);
22.             p2->next = p1;
23.             p2 = p1;
24.             num--;
25.         }
26.         p2->next = NULL;
27.     }
28.     return head;
29. }
```

```
30. void traverse(struct node * head)
31. {
32.     struct node *p;
33.     printf("The link nodes are: ");
34.     p = head;
35.     if (head != NULL)
36.     {
37.         do {
38.             printf("%d ", p->data);
39.             p = p->next;
40.         } while (p != NULL);
41.     }
42.     printf("\n");
43. }
44. int main()
45. {
46.     int numOfNodes = 0;
47.     struct node * head
48.     scanf("%d", &numOfNodes);
49.     head = initialize(numOfNodes);
50.     traverse(head);
51.     return 0;
52. }
```

```
5
1 2 3 4 5
The link nodes are: 1 2 3 4 5
```

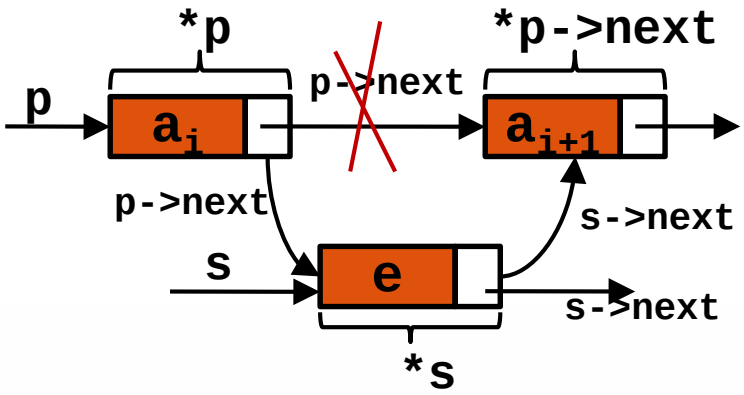
在链表中插入结点（一般情况）

- 假设存储元素 e 的结点为 s，要插入到结点 p 和 p->next 之间



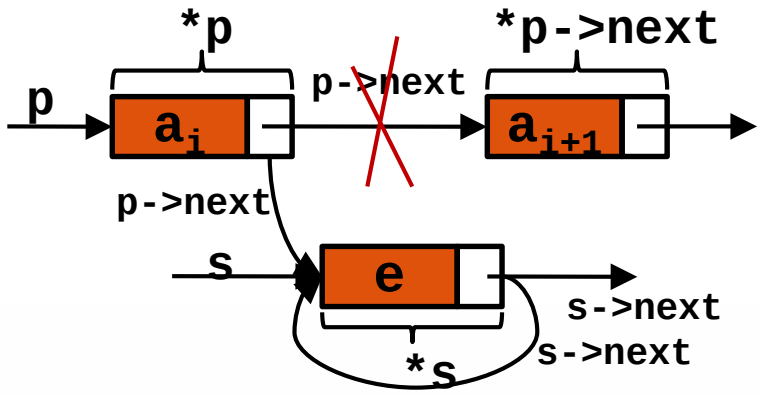
正确的做法：

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



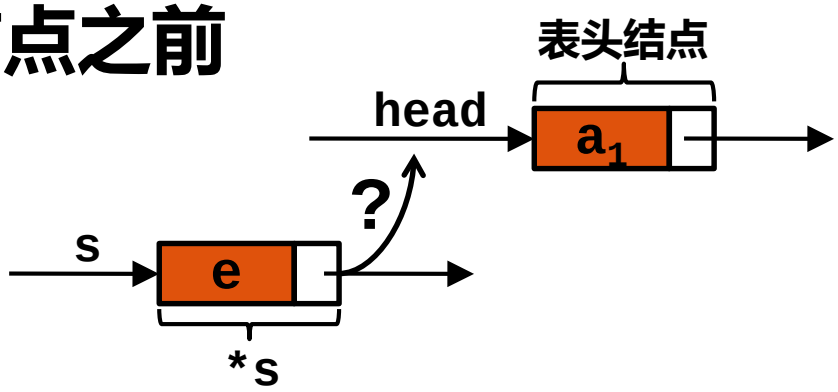
错误的做法：

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



在链表头插入结点（特殊情况）

- 假设存储元素 e 的结点为 s，要插入到指针 head指向的表头结点之前

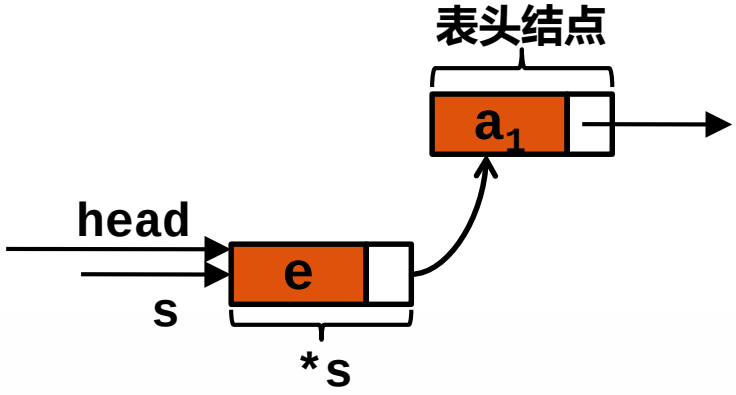


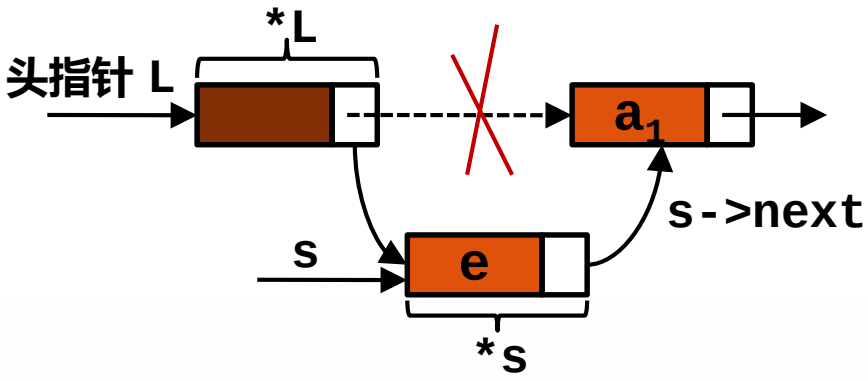
方法一：

```
s->next = head; head = s;
```

方法二（增加头结点L）

```
s->next=L->next; L->next=s;
```

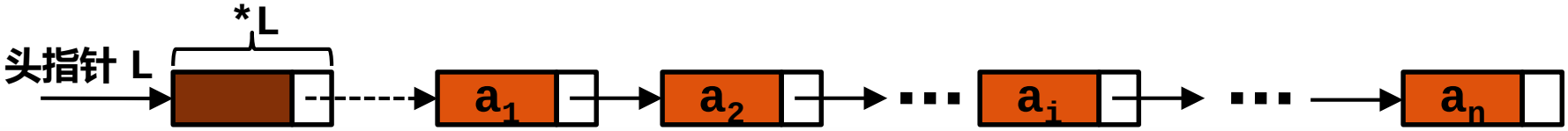




```
1.  struct node * insertNode(struct node * head, int position, int value)
2.  {
3.      struct node *p, *s;
4.      p = head;
5.      if (position == 0) //在表头插入，方法一需要这个分支，方法二则不需要
6.      {
7.          s = (struct node*)malloc(LEN);
8.          s->data = value;
9.          s->next = p;
10.         head = s;
11.     }
12.     else
13.     {
14.         while (position>1 && p->next != NULL) //找到插入位置
15.         {
16.             position--;
17.             p = p->next;
18.         }
19.         s = (struct node*)malloc(LEN);
20.         s->data = value;
21.         s->next = p->next;
22.         p->next = s;
23.     }
24.     return head;
25. }
```

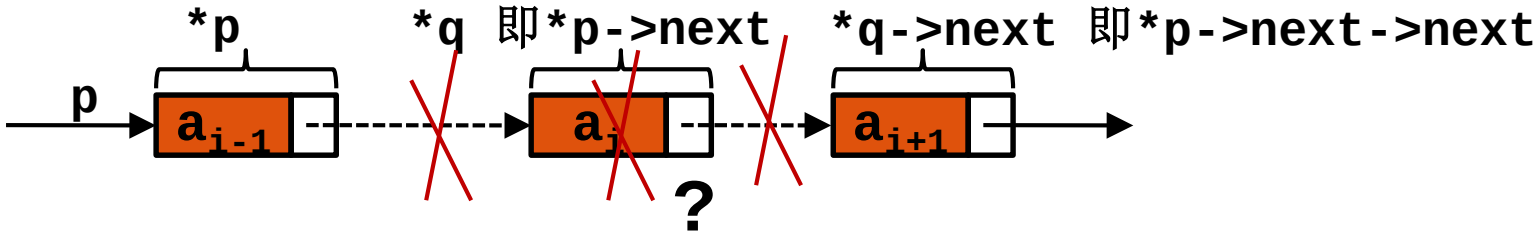
头结点

- **头结点的特点**
  - 位于表头结点之前；
  - 链表中第一个数据仍存放在表头结点中，头结点的数据域不存放信息或存放其他信息。
- **作用**：使链表中所有有效结点都有前驱结点，可以统一链表的插入和删除操作。



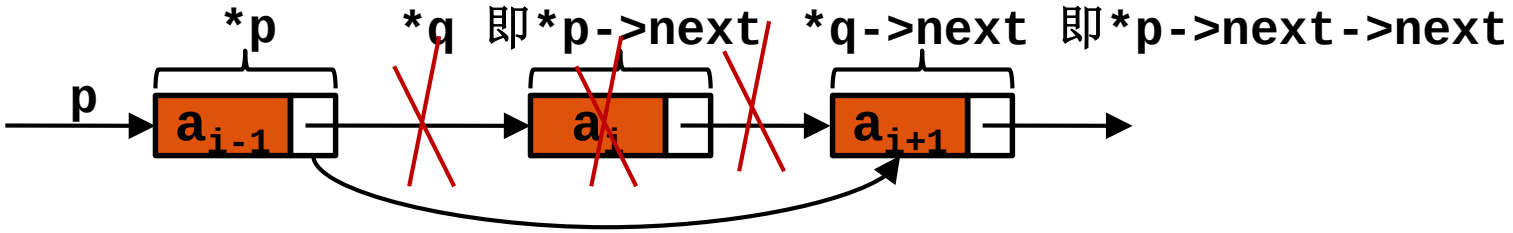
在带头结点的链表中删去结点

- 假设要删除单链表中 q 指向的结点，其中 p 是 q 的前继结点



正确的做法：p->next = p->next->next;

或者：q = p->next; p->next = q->next;

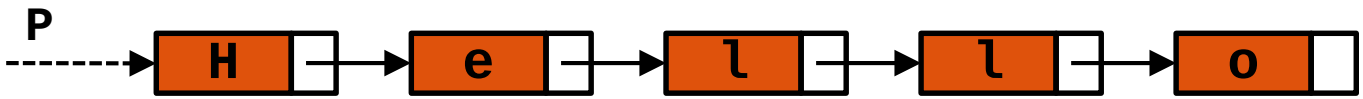


```
1.  struct node * removeNode(struct node * L, int value)
2.  {
3.      struct node *p, *q;
4.      p = L;
5.      q = L->next;
6.      while (q != NULL && q->data != value) //寻找删除结点
7.      {
8.          p = q;
9.          q = q->next;
10.     }
11.     if (q != NULL)
12.     {
13.         p->next = q->next;
14.     }
15.     else
16.     {
17.         printf ("The node is not found!\n");
18.     }
19.     free(q);
20.     return L;
21. }
```

作业 2017/12/20

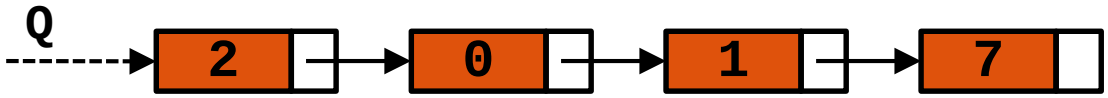
- 按下列要求编写程序，提交手写源代码

1. (1) 输入一个字符串，用链表存储，如输入"Hello"表示为。



(2) 定义函数Sum(P)，求出链表P中各结点的字符所对应的ASCII码的和，即一个整数n，返回这个整数。

(3) 定义函数int2list(n) 把(2)求得的整数用链表存储，返回该链表的头指针，如2017存储为如下链表，返回Q



注意：程序必须添加必要的注释，以便批改人员理解！