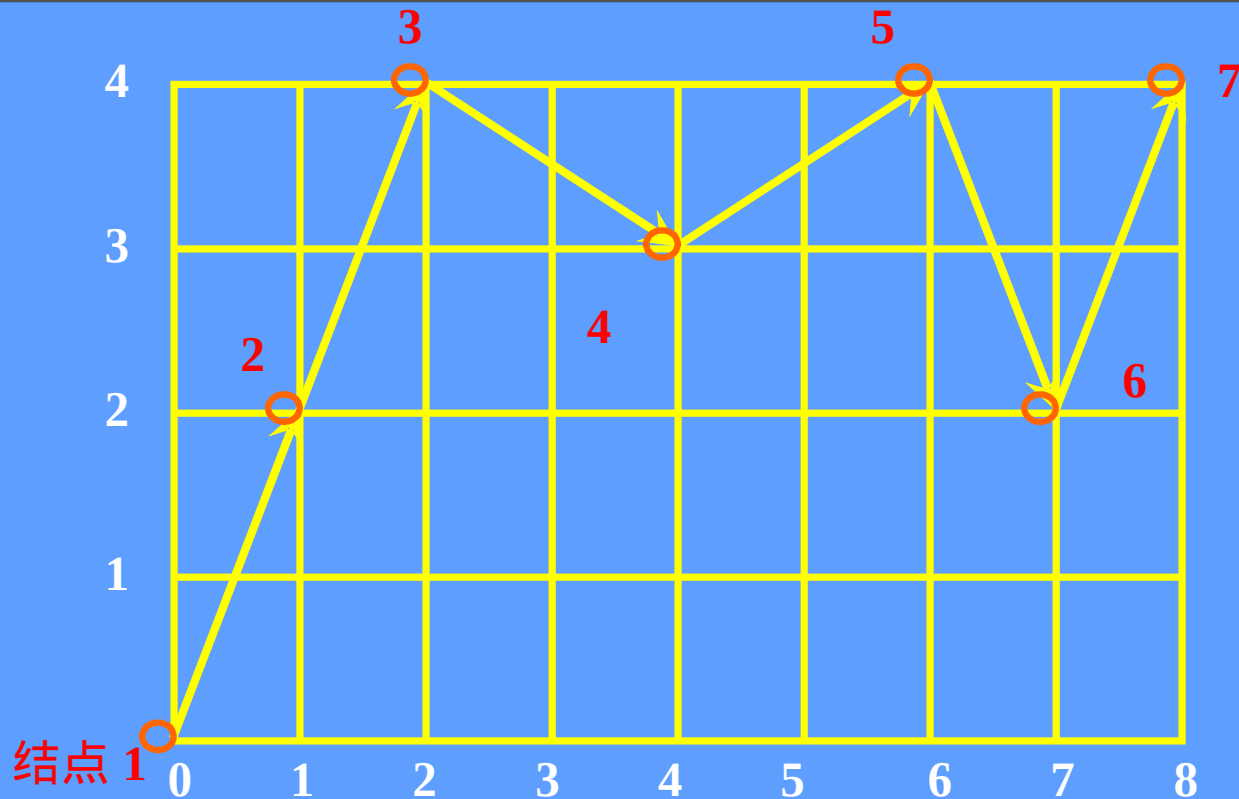
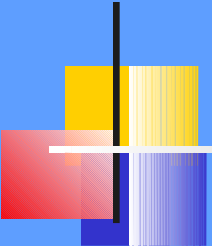


# 第十一章 链表

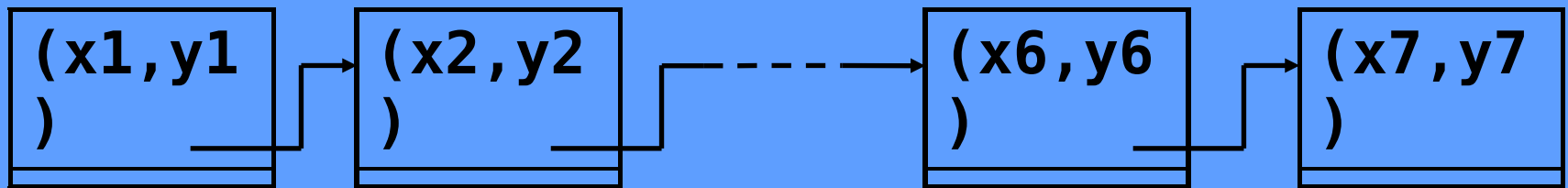
# 结构的概念与应用

例：跳马。依下图将每一步跳马之后的位置  $(x, y)$  放到一个“结点”里，再用“链子穿起来”，形成一条链，相邻两结点间用一个指针将两者连到一起。





依上图有 7 个结  
点



为了表示这种既有数据又有指针的情况，引入结构这种数据类型。



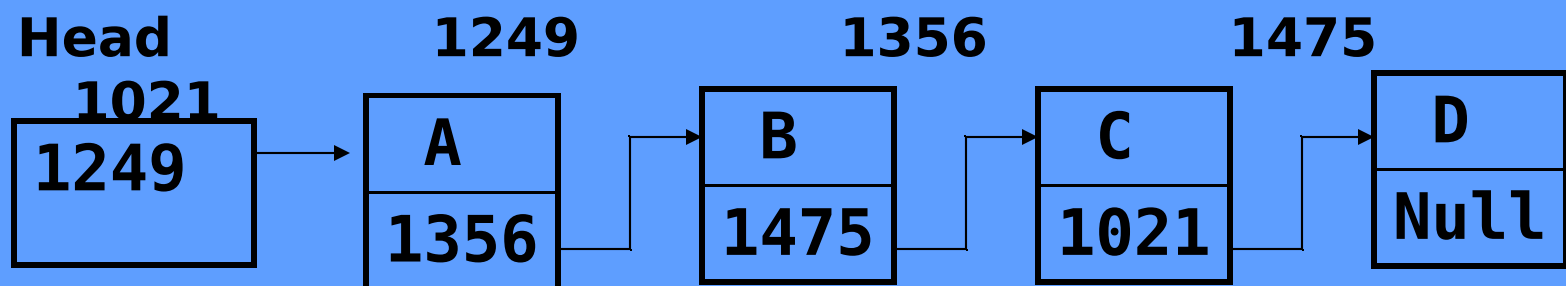
## 11.7 用指针处理链表

链表是程序设计中一种重要的动态数据结构，它是动态地进行存储分配的一种结构。

### 动态性体现为

- 链表中的元素个数可以根据需要增加和减少，不像数组，在声明之后就固定不变；
- 元素的位置可以变化，即可以从某个位置删除，然后再插入到一个新的地方；

结点里的指针是存放下一个结点的地址



- 1、链表中的元素称为“结点”，每个结点包括两个域：**数据域和指针域**；
- 2、单向链表通常由一个头指针（**head**），用于指向链表头；
- 3、单向链表有一个尾结点，该结点的指针部分指向一个空结点（**NULL**）。



# 链表中结点的定义

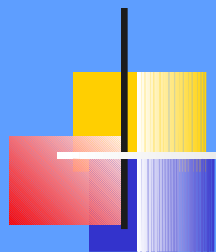
- ❑ 链表是由结点构成的， 关键是定义结点；
- ❑ 链表的结点定义打破了先定义再使用的限制，即可以用自己定义自己；
- ❑ 递归函数的定义也违反了先定义再使用；

这是 C 语言程序设计上的两大特例

## 链表的基本操作

对链表的基本操作有：

- ( 1 ) **创建链表**是指，从无到有地建立起一个链表，即往空链表中依次插入若干结点，并保持结点之间的前驱和后继关系。
- ( 2 ) **检索操作**是指，按给定的结点索引号或检索条件，查找某个结点。如果找到指定的结点，则称为检索成功；否则，称为检索失败。
- ( 3 ) **插入操作**是指，在结点  $k_{i-1}$  与  $k_i$  之间插入一个新的结点  $k'$ ，使线性表的长度增 1，且  $k_{i-1}$  与  $k_i$  的逻辑关系发生如下变化：  
插入前， $k_{i-1}$  是  $k_i$  的前驱， $k_i$  是  $k_{i-1}$  的后继；插入后，新插入的结点  $k'$  成为  $k_{i-1}$  的后继、 $k_i$  的前驱。



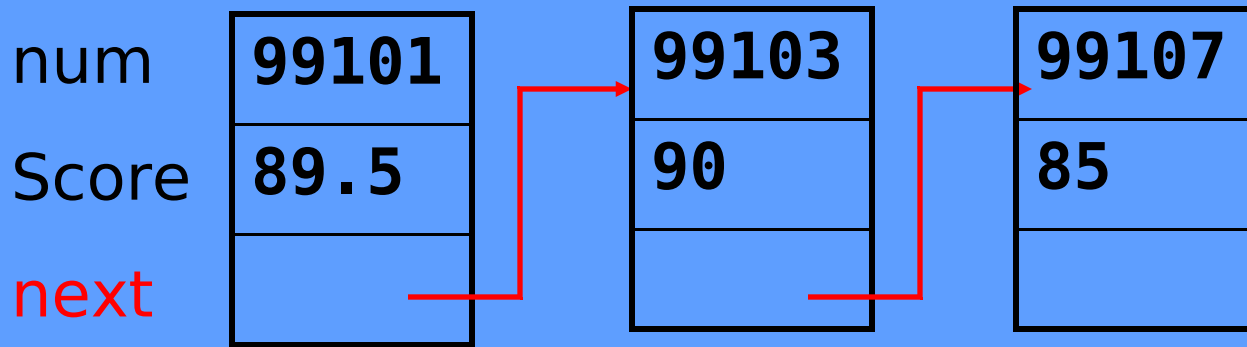
( 4 ) **删除操作**是指，删除结点  $k_i$ ，使线性表的长度减 1，且  $k_{i-1}$ 、 $k_i$  和  $k_{i+1}$  之间的逻辑关系发生如下变化：

删除前， $k_i$  是  $k_{i+1}$  的前驱、 $k_{i-1}$  的后继；删除后， $k_{i-1}$  成为  $k_{i+1}$  的前驱， $k_{i+1}$  成为  $k_{i-1}$  的后继。

(5) 打印输出



一个指针类型的成员既可指向其它类型的结构体数据，也可以指向自己所在的结构体类型的数据



**next** 是 **struct student** 类型中的一个成员，它又指向 **struct student** 类型的数据。

换名话说：**next** 存放下一个结点的地址

表

```
#define NULL 0

struct student
{ long num;      float score;
  struct student *next; };

main()
{ struct student a, b, c, *head, *p;
  a.num=99101; a.score=89.5;
  b. num=99103; b.score=90;
  c.num=99107 ; c.score=85;
  head=&a;  a.next=&b;  b.next=&c;  c.next=NULL;
  p=head;
do
{ printf(" %ld %5.1f\n",p->num,p->score);
  p=p->next; }while(p!=NULL); }
```

各结点在程序中定义，不是临时开辟的，始终占有内容不放，这种链表称为“静态链表”

## 11.7.3 处理动态链表所需的函数

C 语言使用系统函数动态开辟和释放存储单元

### 1.malloc 函数

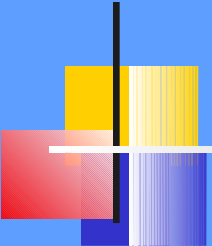
数

函数原形：**void \*malloc(unsigned int size);**

**作用：**在内存的动态存储区中分配 **一个** 长度为 **size** 的连续空间。

**返回值：**是一个指向分配域起始地址的指针（基本类型 **void**）。

**执行失败：**返回 **NULL**



## 2. calloc 函数

**函数原形** : `void *calloc(unsigned n,unsigned size);`

**作用**：在内存动态区中分配 **n 个** 长度为 **size** 的连续空间。

**函数返回值**：指向分配域起始地址的指针

**执行失败**：返回 **null**

**主要用途**：为一维数组开辟动态存储空间。

**n** 为数组元素个数，每个元素长度为 **size**

### 3. free 函数

函数原形：`void free(void *p);`

作用：释放由 `p` 指向的内存区。

`P`：是最近一次调用 `calloc` 或 `malloc` 函数时返回的值。

**free 函数无返回值**

动态分配的存储单元在用完后一定要释放，否则内存会因申请空间过多引起资源不足而出现故障。

# 结点的动态分配

ANSI C 的三个函数 (头文件 malloc.h) void

\*malloc(unsigned int size)

void \*calloc(unsigned n, unsigned  
size)

void free(void \*p)

C++ 的两个函数

new 类型 (初值)

delete [ ] 指针变量

/\*[ ] 表示释放数组, 可有可无) \*/

使用 new 的优点:

可以通过对象的大小直接分配, 而不管对象的具体长度是多少 ( p340 例 14.10 )

## 11.7.4 建立动态链表

基本方法：

三个结点（头结点 **head**、尾结点 **NULL** 和待插入结点 **P**）

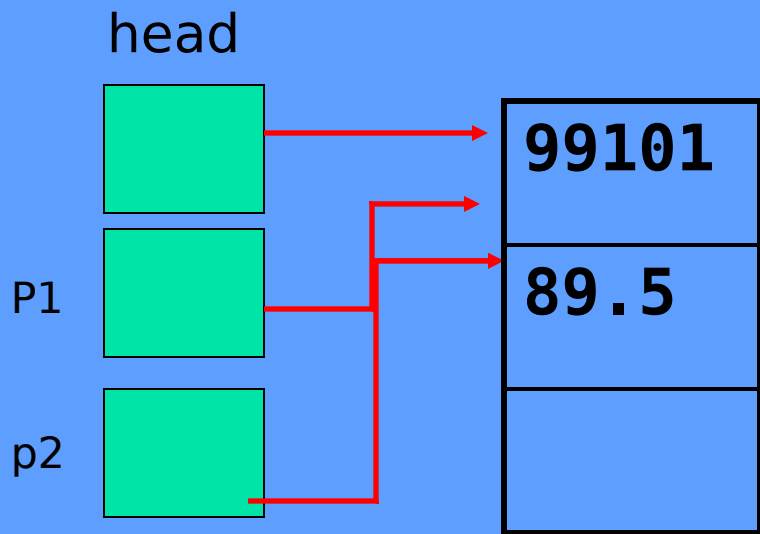
第一步：定义头结点 **head**、尾结点 **p2** 和待插入结点 **p1**，待插入的结点数据部分初始化；

第二步：该结点被头结点指向。使用 `malloc(LEN)`。  
`P1=p2=(struct student*)malloc(LEN);` 头指针部分为空，`head=NULL;`

第三步：重复申请待插入结点空间，对该结点的数据部分赋值（或输入值），将该结点插入在最前面，或者最后面（书上在尾部插入）。

`P2->next=P1;`  
`P2->next=NULL;`  
取后：`P2->next=NULL;`

## 11.7.4 建立动态链表

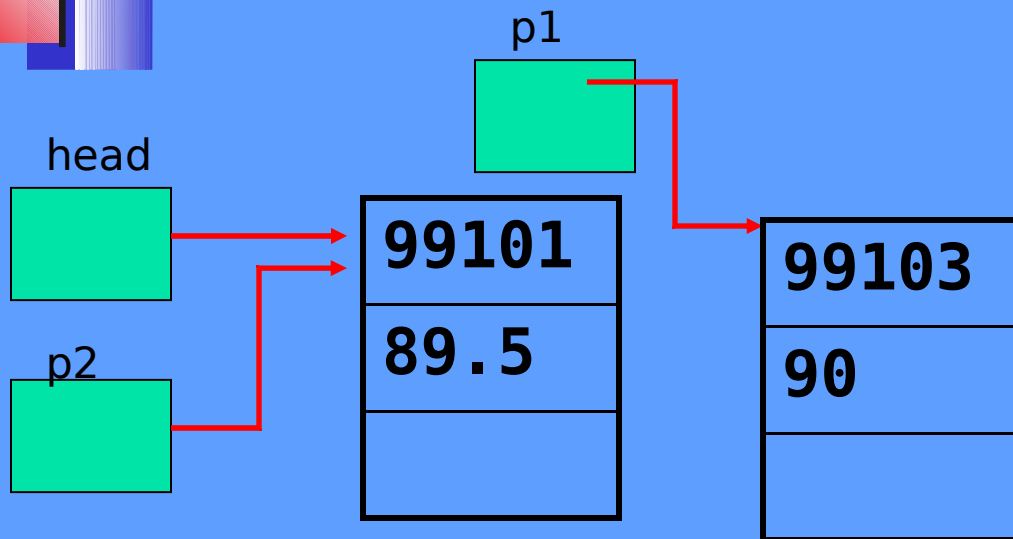


待插入的结点 **p1** 数据部分初始化，该结点被头结点 **head**、尾结点 **p2** 同时指向。

1. 任务是开辟结点和输入数据
2. 并建立前后相链的关系



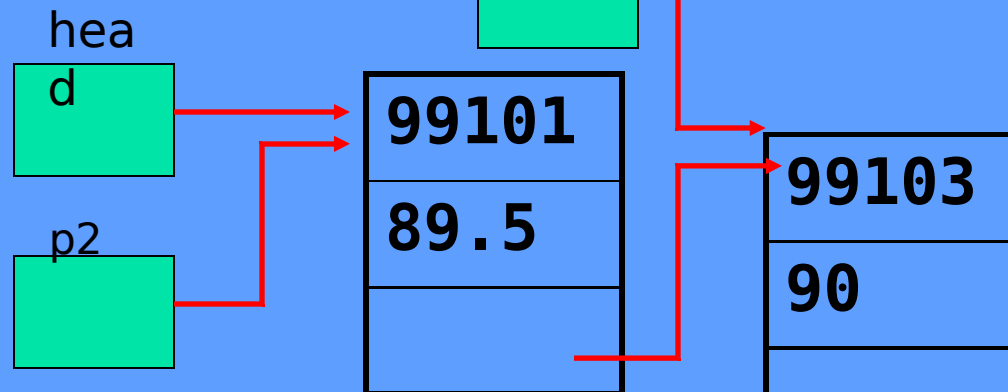
图 11.14



(a)

p1 重复申请待插入结点空间，对该结点的数据部分赋值（或输入值）

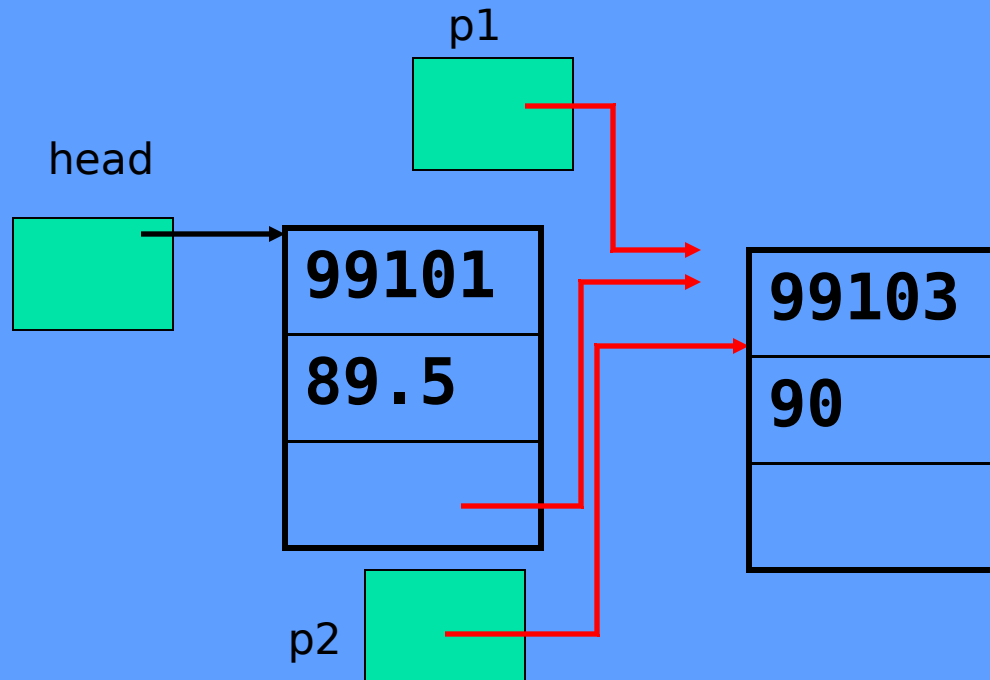
**P2->next** 指向 **p1** 新开辟的结点。



(b)



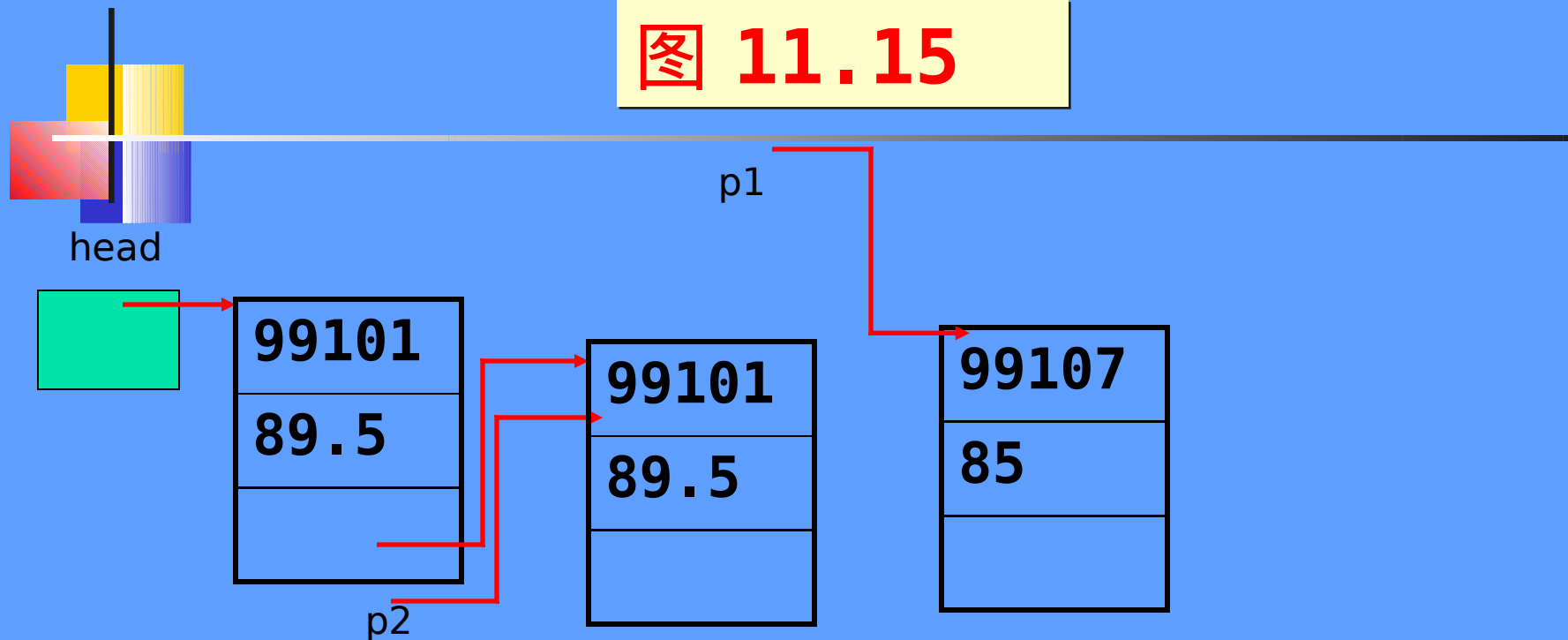
11.14



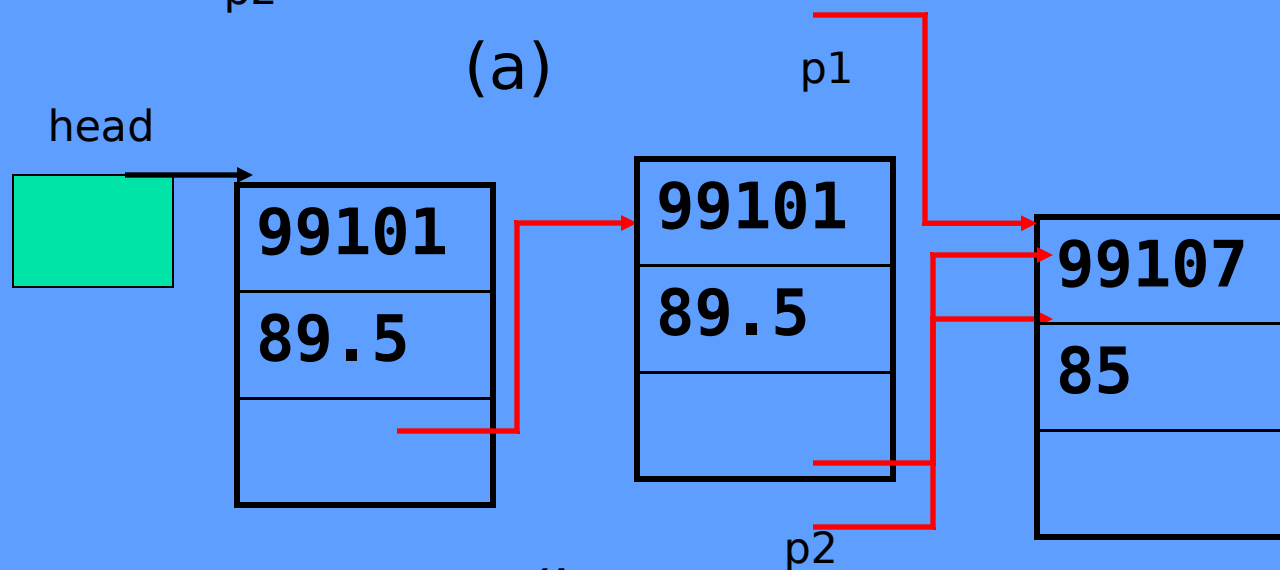
(c)

**P2 指向新  
结点  
 $p2 = p1$**

图 11.15

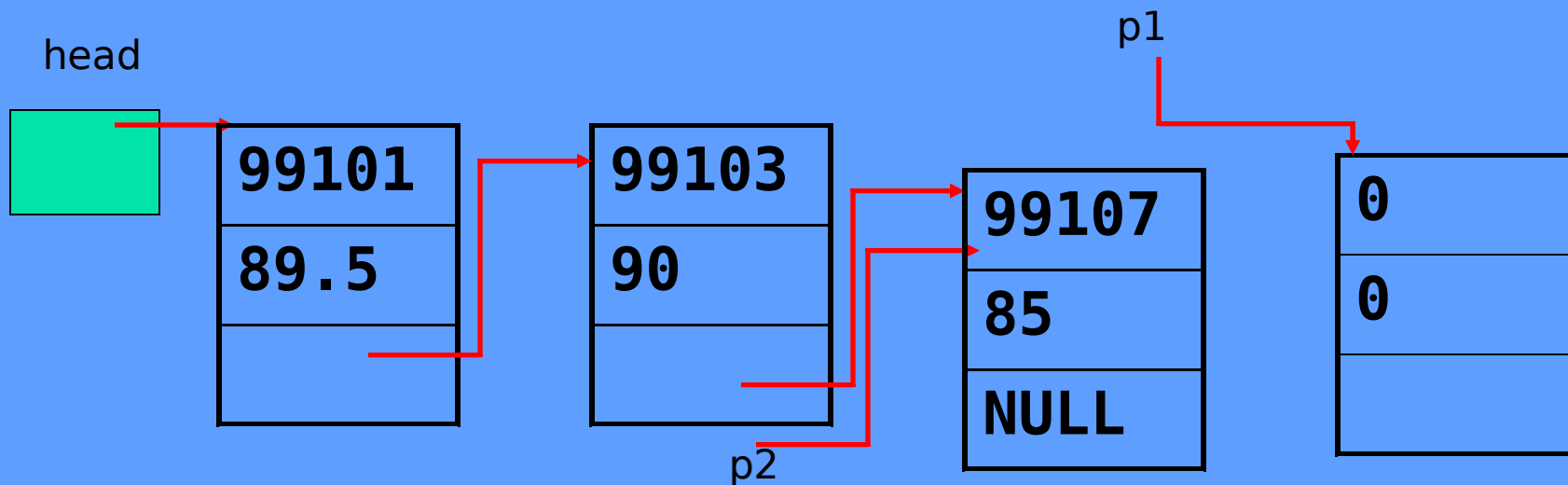
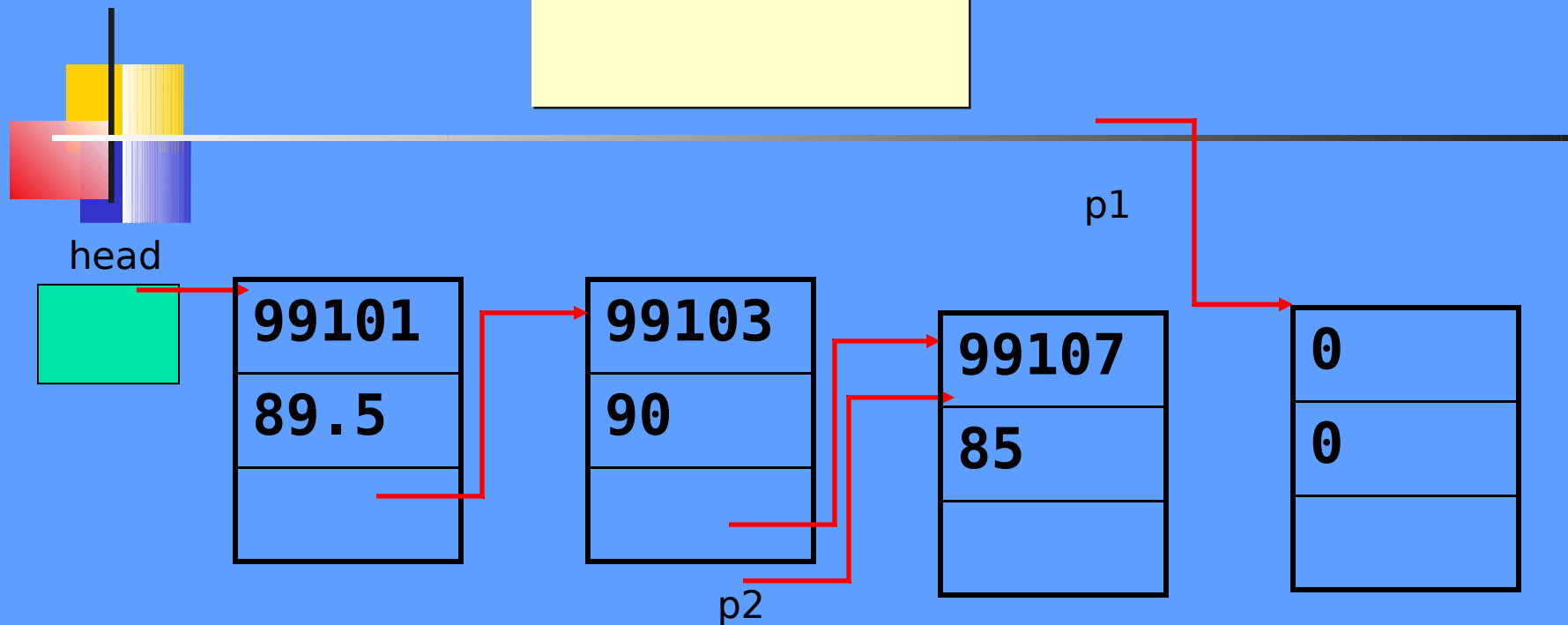


(a)



(b)

图 11.16



# 例 11.8 建立一个有 3 名学生数据的单向动态链表

```
#define NULL 0
```

```
#define LEN sizeof 结构体类型数据的长度， sizeof 是“字节数运算符”
```

```
struct student
```

```
{ long num; float score; struct student *next; };
```

```
int n;
```

```
struct student *creat(void) 定义指针类型的函数。带回链表的起始地址
```

```
{ struct student *head; struct student *p1, *p2;
```

p1, p2 是指向结构体类型数据的指针变量，强行转换成结构体类型

malloc 开辟长度为 LEN 的内存区

```
scanf(" %1d,%f",&p1->num,&p1->score),
```

```
head=NULL; 假设头指向空结点
```

```
while(p1->num!=0)
```

```
{ n=n+1; /*n 是结点的个数 */
```

```
if(n==1)head=p1; 头指针指向 p1 结点
```

```
else p2->next=p1; p2=p1; P1 开辟的新结点链到了 p2 的后面
```

```
p1=(struct student*)malloc(LEN); P1 继续开辟新结点
```

```
scanf("%1d,%f",&p1->num,&p1->s); 给新结点赋值此
```

```
p2->next=NULL; return(head); }// 返回链表的头指针
```

算法： **p1** 指向新开的结点： `p1=(struct student*)malloc(LEN);`

**p1** 的所指向的结点连接在 **p2** 所指向结点后面，用 `p2->next=p1` 来实现。

**p2** 指向链表中最后建立的结点，：`p2=p1;`

## 链表遍历

1. 单向链表总是从头结点开始的；
2. 每访问一个结点，就将当前指针向该结点的下一个结点移动：

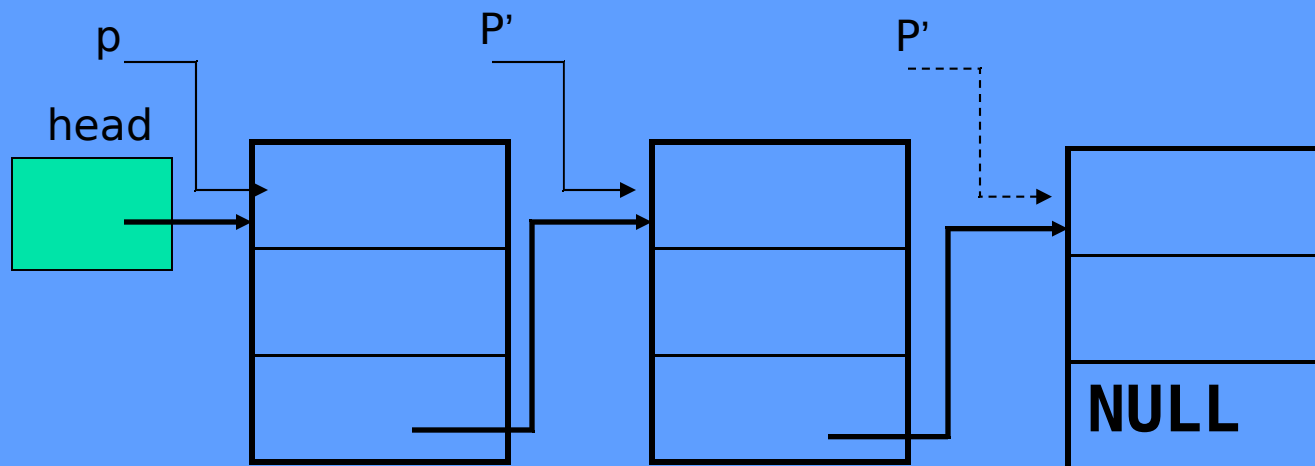
**`p=p->next;`**

3. 直至下一结点为空


**`P=NULL`**



11.18







```
void print (struct student *head)
{ struct student * p;
  printf("\nNow,These %d records are:\n",n);
  p=head;
  if(head!=NULL)
  do
  { printf(" %ld %5.1f\n",p -> num,p -> score);
    p=p -> next;
  }while(p!=NULL);
}
```



## 11.7.6 对链表的删除操作

### 删除结点原则：

不改变原来的排列顺序，只是从链表中分离开来，撤消原来的链接关系。

### 两种情况：

- 1、要删的结点是头指针所指的结点则直接操作；
- 2、不是头结点，要依次往下找。

另外要考虑：空表和找不到要删除的结点



## 链表中结点删除

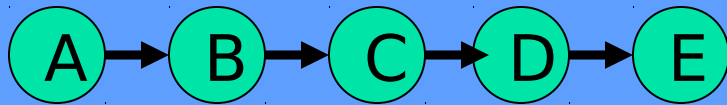
需要由两个临时指针：

**P1：** 判断指向的结点是不是要删除的  
结点（用于寻找）；

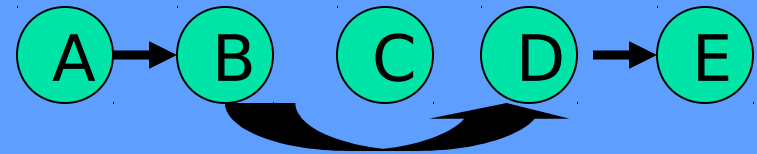
**P2：** 始终指向 **P1** 的前面一个结点；



11.19

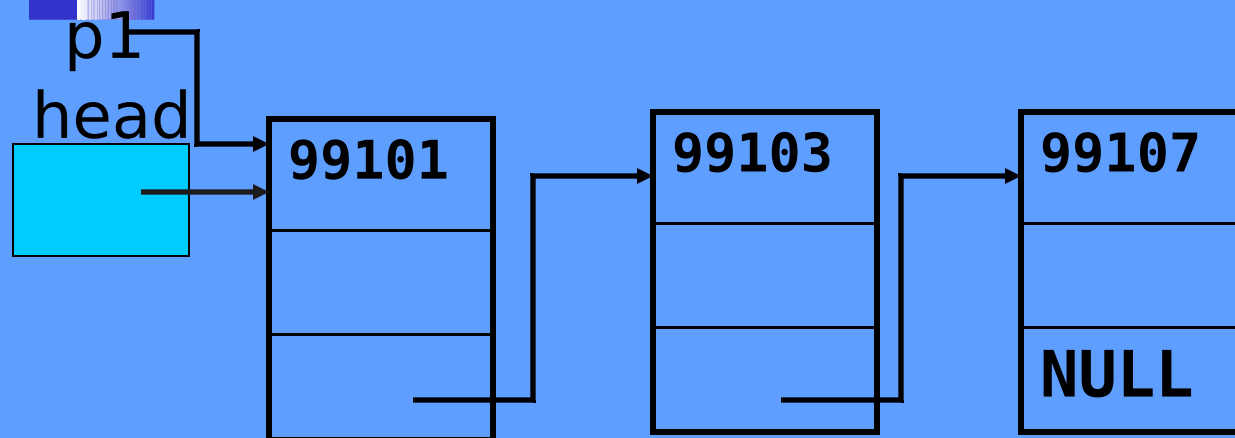


(a)

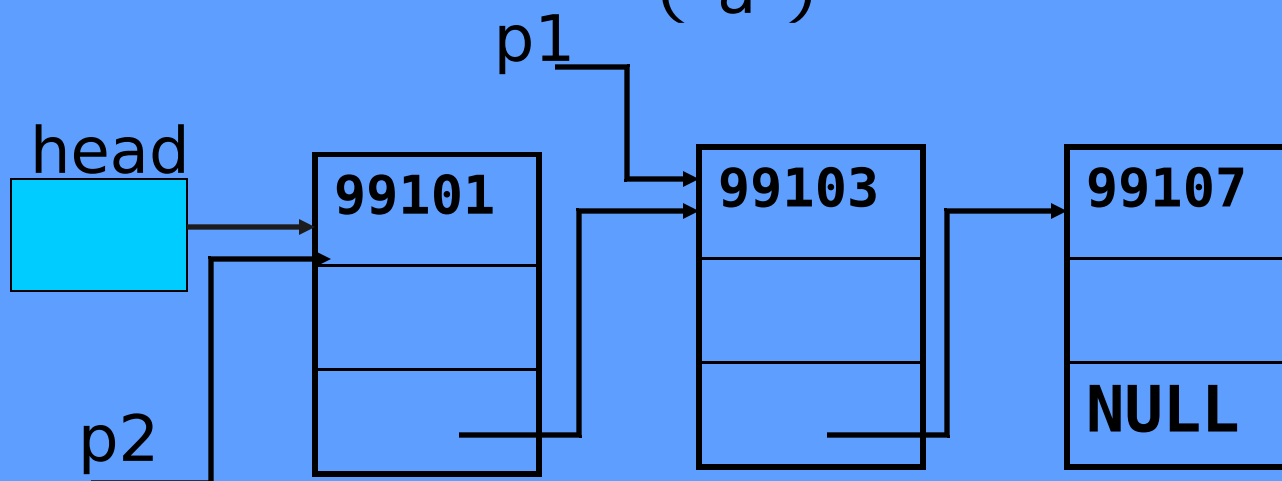


(B)

图 11.20



( a )



( b )

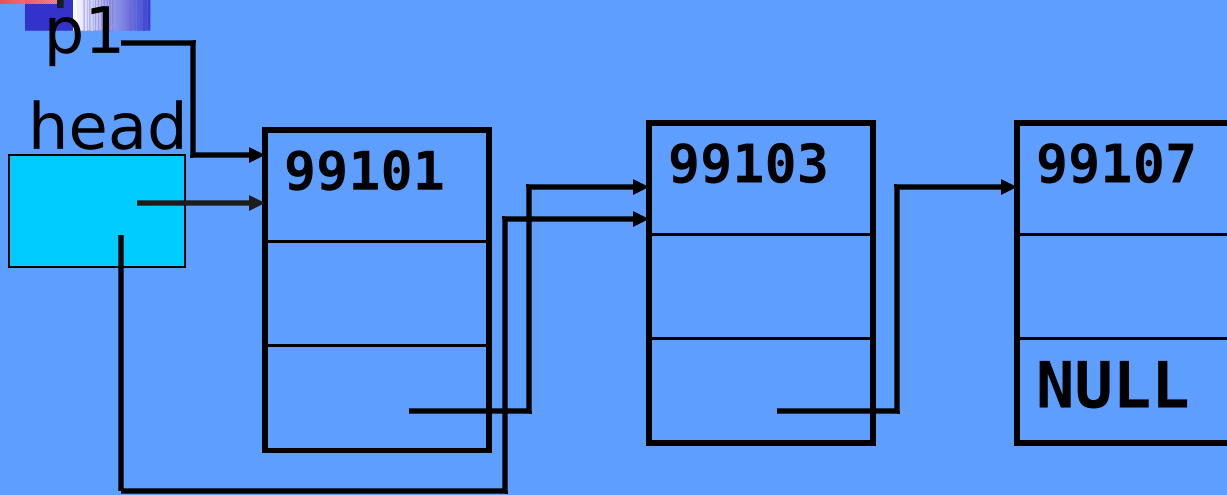
原链表

**P1** 指向头结点

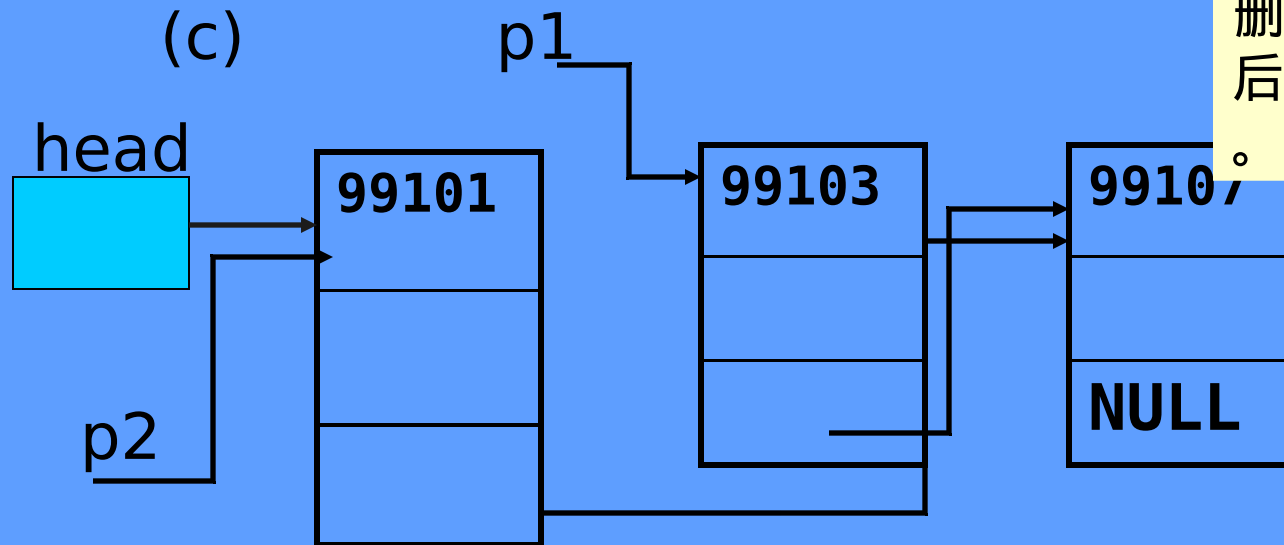
**P2** 指向 **p1** 指向的结点。**P1** 指向下移一个结点。



11.20



(c)



(d)

经判断后，第 1 个结点是要删除的结点，**head** 指向第 2 个结点，第 1 个结点脱离。经 **P1** 找到要删除的结点后使之脱离。

```

struct student *del( struct student *head, long num )
{ struct student  *p1, *p2;
  if(head==NULL) {printf("\nlist null!\n"); goto end; }
  p1=head;
  while(num!=p1->num&& p1->next!=NULL)
  { p2=p1; p1=p1->next; }
  if(num==p1->num)
  { if(p1==head) head=p1->next;
    else p2->next=p1->next;
    printf("delete:%ld\n",num);
    n=n-1; }
  else printf(" %ld not been found!\n",num);
end: return(head); }

```

找到了

没找到

例

题

1

0

1



## 11.7.7 对链表的插入操作

**插入结点：** 将一个结点插入到已有的链表中

**插入原则：**

- 1、插入操作不应破坏原链接关系
- 2、插入的结点应该在它该在的位置

**实现方法：**

应该有一个插入位置的查找子过程  
共有三种情况：

- 1、插入的结最小
- 2、插入的结点最大
- 3、插入的结在中间



## 操作分析

同删除一样，需要几个临时指针：

**P0**：指向待插的结点；初始化：**p0=数组 stu**；

**P1**：指向要在 **P1** 之前插入结点；初始化：**p1=head**；

**P2**：指向要在 **P2** 之后插入结点；

插入操作：当符合以下条件时：**p0->num** 与 **p1->num** 比较找位置

**if (p0->num>p1->num)&&(p1->next!=NULL)** 则插入的结点不在 **p1** 所指结点之前；指针后移，交给 **p2**；

**p1=p1->next;**      **p2=p1;**

则继续与 **p0** 指向的数组去比，直到 (**p1->next!=NULL**) 为止。

否则有两种情况发生：

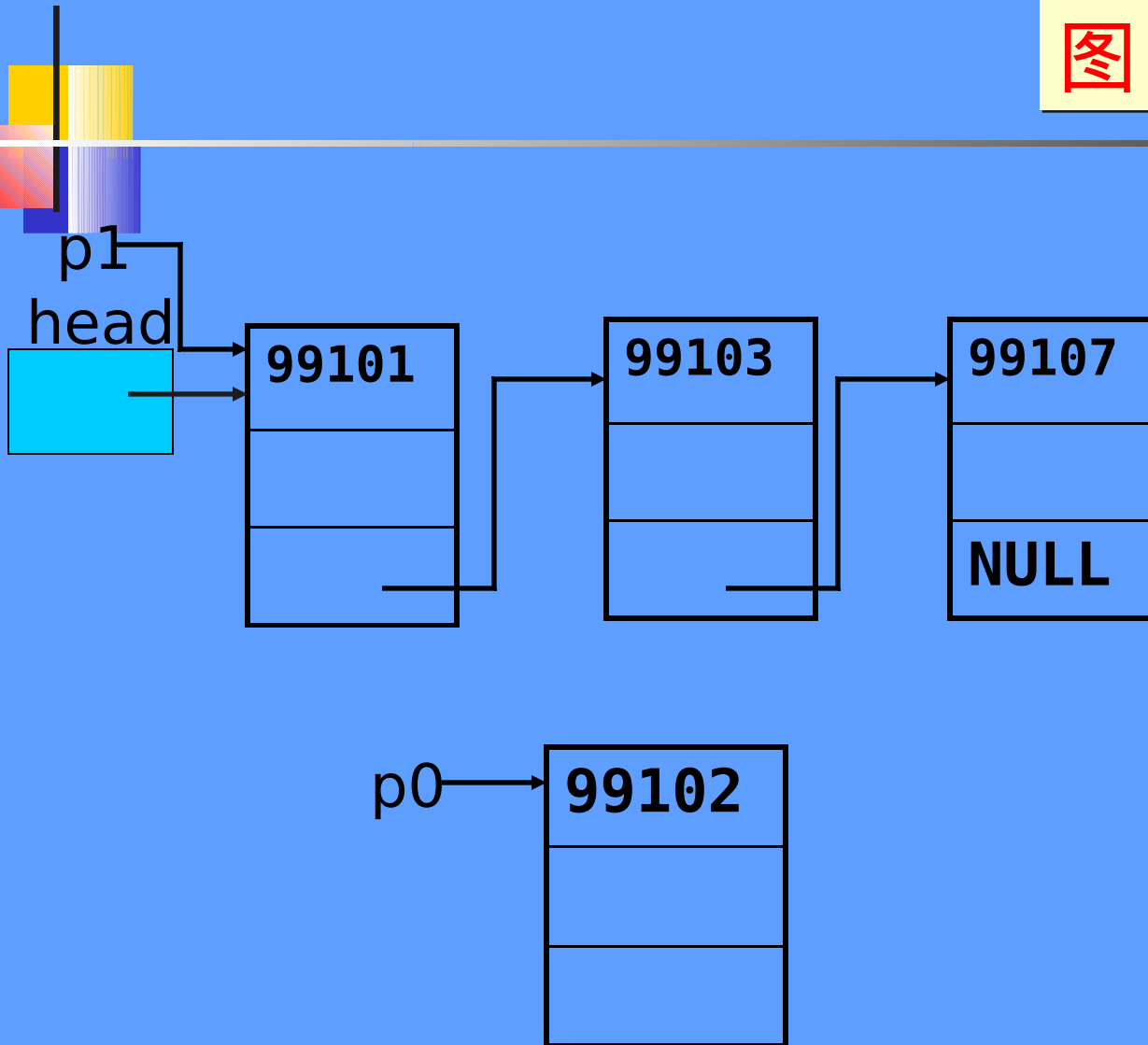
**if (head==p1)** **head=p0;** **p0->next=p1** 插到原来第一个结点之前；

**else p2->next=p0;** **p0->next=p1;** 插到 **p2** 指向的结点之后；

还有另外一种情况：插到最后的结点之后；

**p1->next=p0;** **p0->next=NULL;**

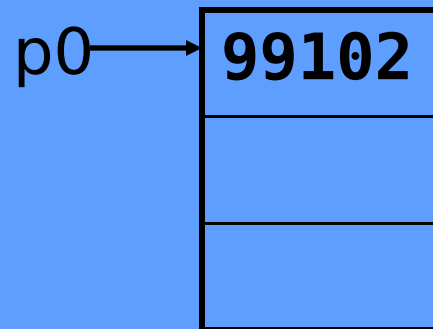
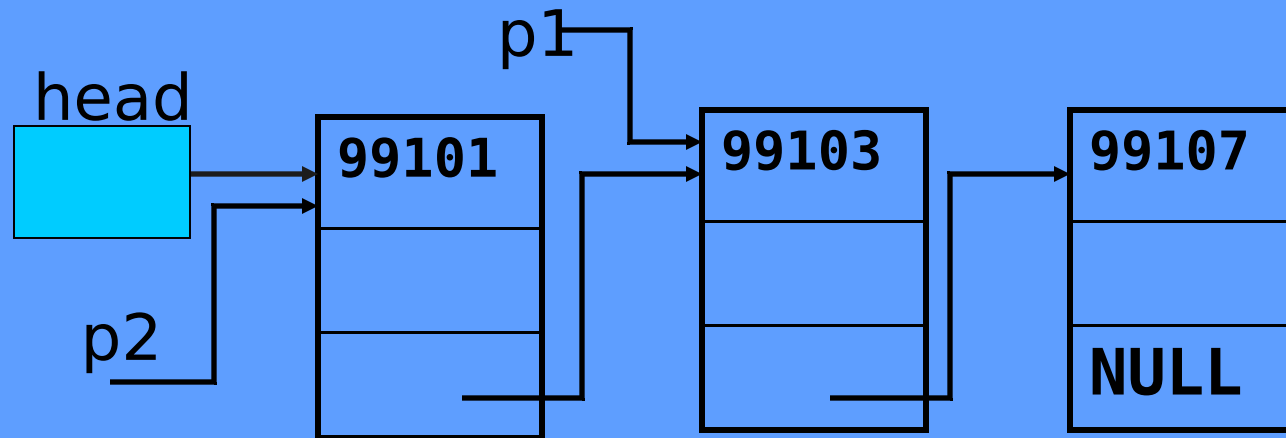
图 11.22



( a )



11.22



(b)

```
struct student insert(struct student *head, struct student *stud)
```

```
{struct student *p0  P0 指向要插的结点
```

```
    p1=head; p0=stud;
```

```
    if( head==NULL; )
```

原来的链表是空  
表

```
        { head=p0;p0->next=NULL; }
```

使 p0 指向的结点作为头结点

```
    else
```

```
        while(( p0->num>p1->num)&&(p1->next!=NULL))
```

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

使 p2 指向刚才 p1 指向的  
结点

```
    if( p0->num<=p1->num)
```

插到原来第一个结点之前

```
        { if(head==p1) head=p0;
```

```
            else p2->next=p0;    p0->next=p1;
```

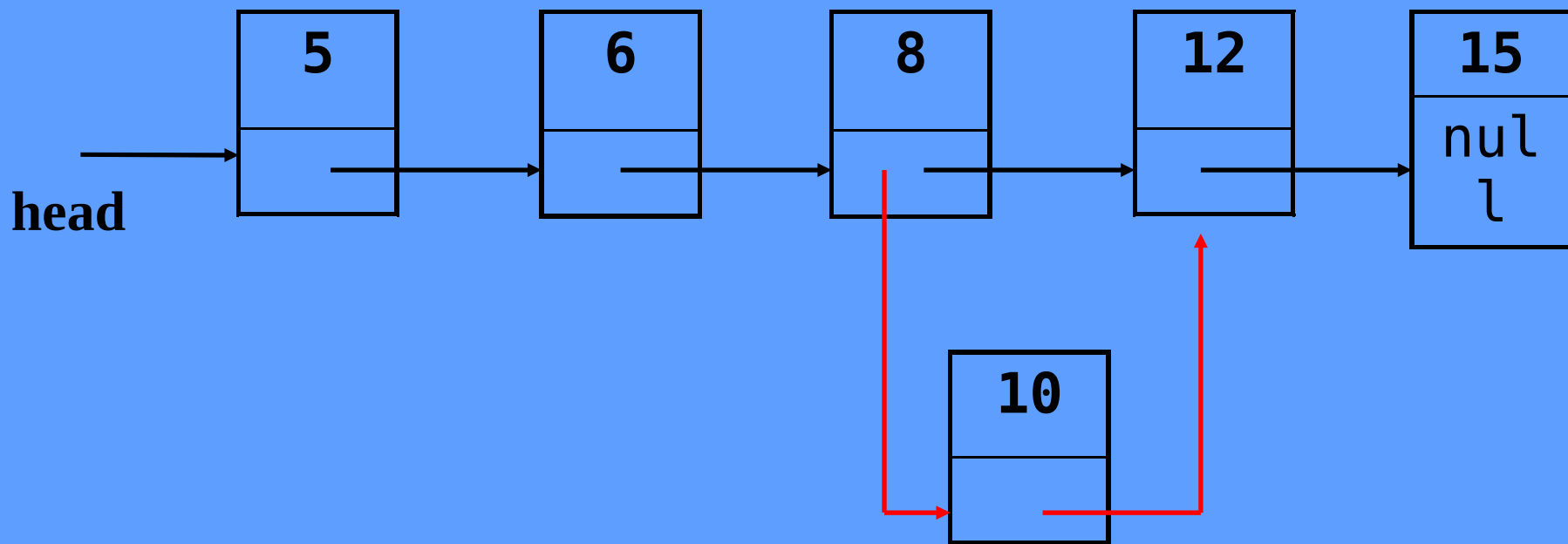
插到 p2 指向的结点  
之后

```
        else {p1->next=p0;p0->next=NULL;}
```

插到最后的结点之后

```
        n=n+1; return(head); }
```

**课堂举例：**已有一个如图所示的链表；  
它是按结点中的整数域从小到大排序的，现在  
要插入一个结点，该结点中的数为 **10**

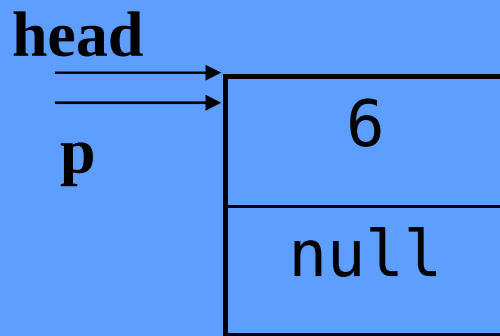


此结点已插入链表

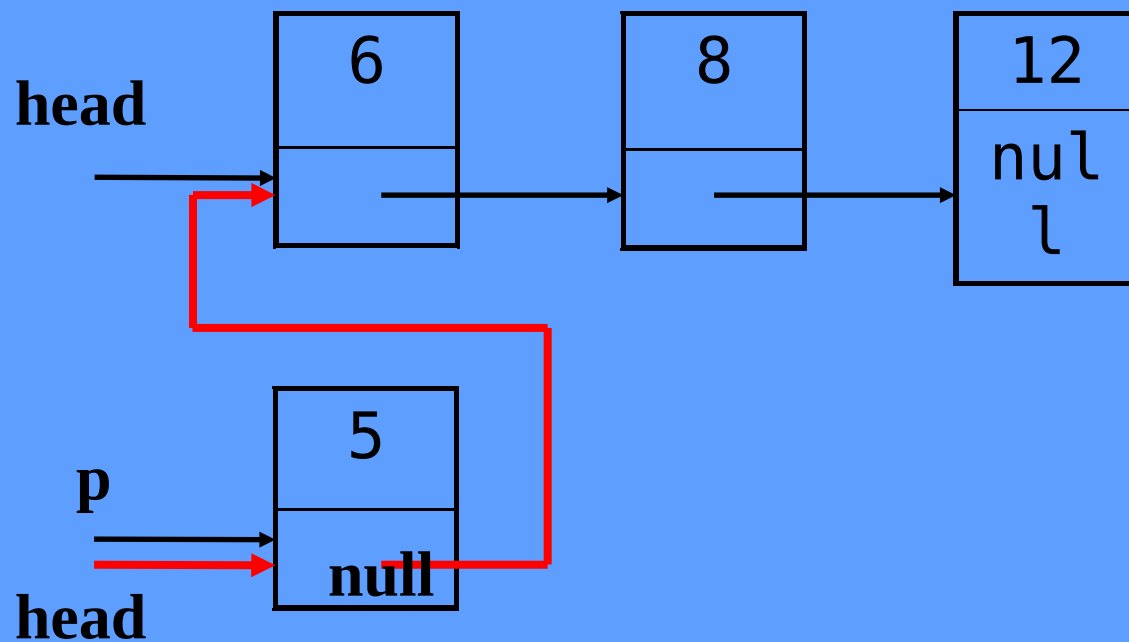
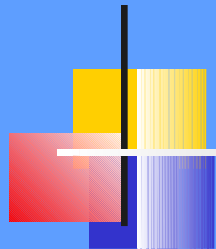
**分析：**按三种情况

**1、**第一种情况，链表还未建成（空链表），待插入结点 **p** 实际上是第一个结点。这时必然有 **head==null**。只要让头指针指向 **p** 就可以了。  
语句为

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

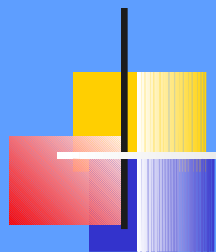


**2、**第二种情况，链表已建成，待插入结点 **p** 的数据要比头结点的数据还要小，这时有  
**(p->num ) < (head->num)**  
当然 **p** 结点要插在 **head** 结点前。



语句为

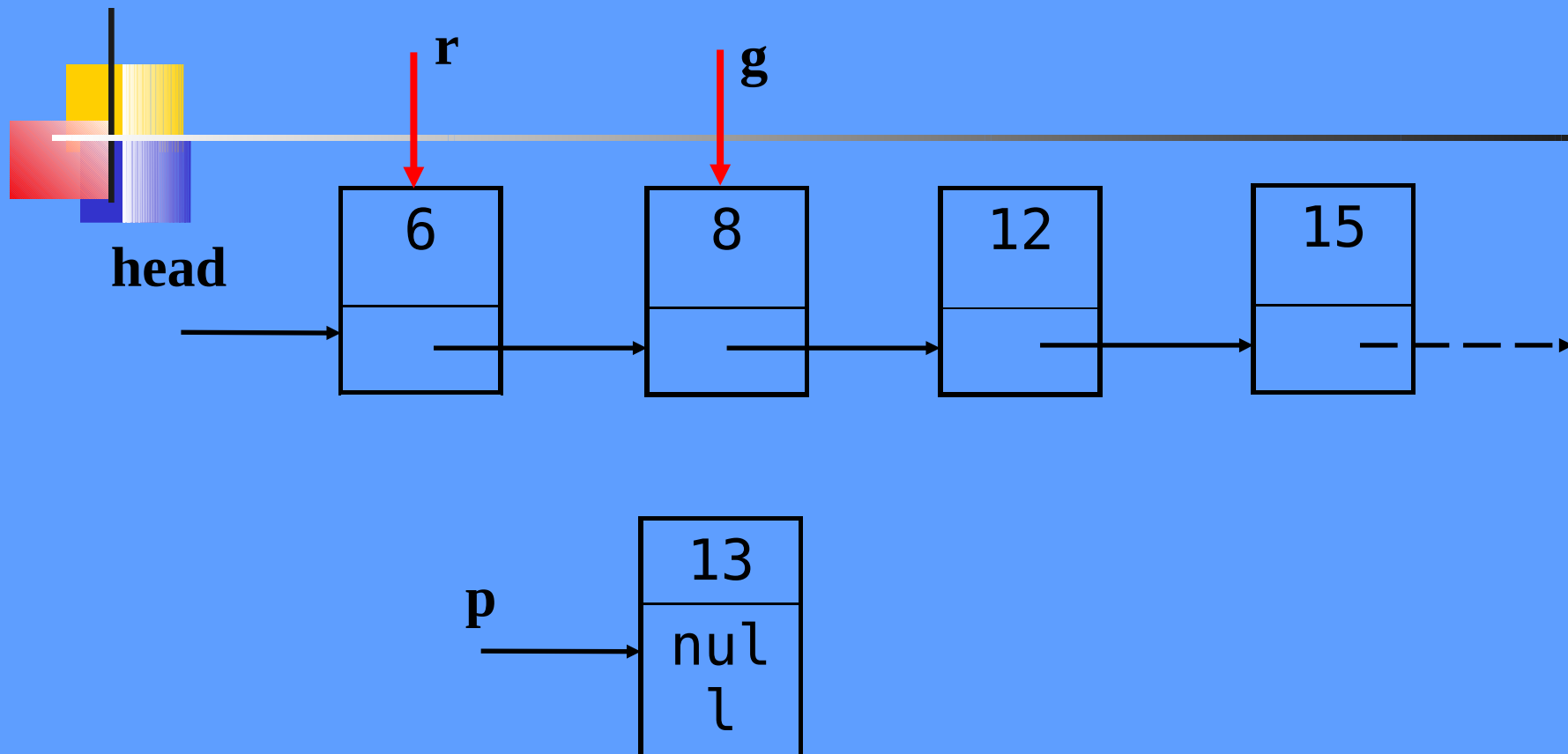
```
p->next=head;  
head=p;
```



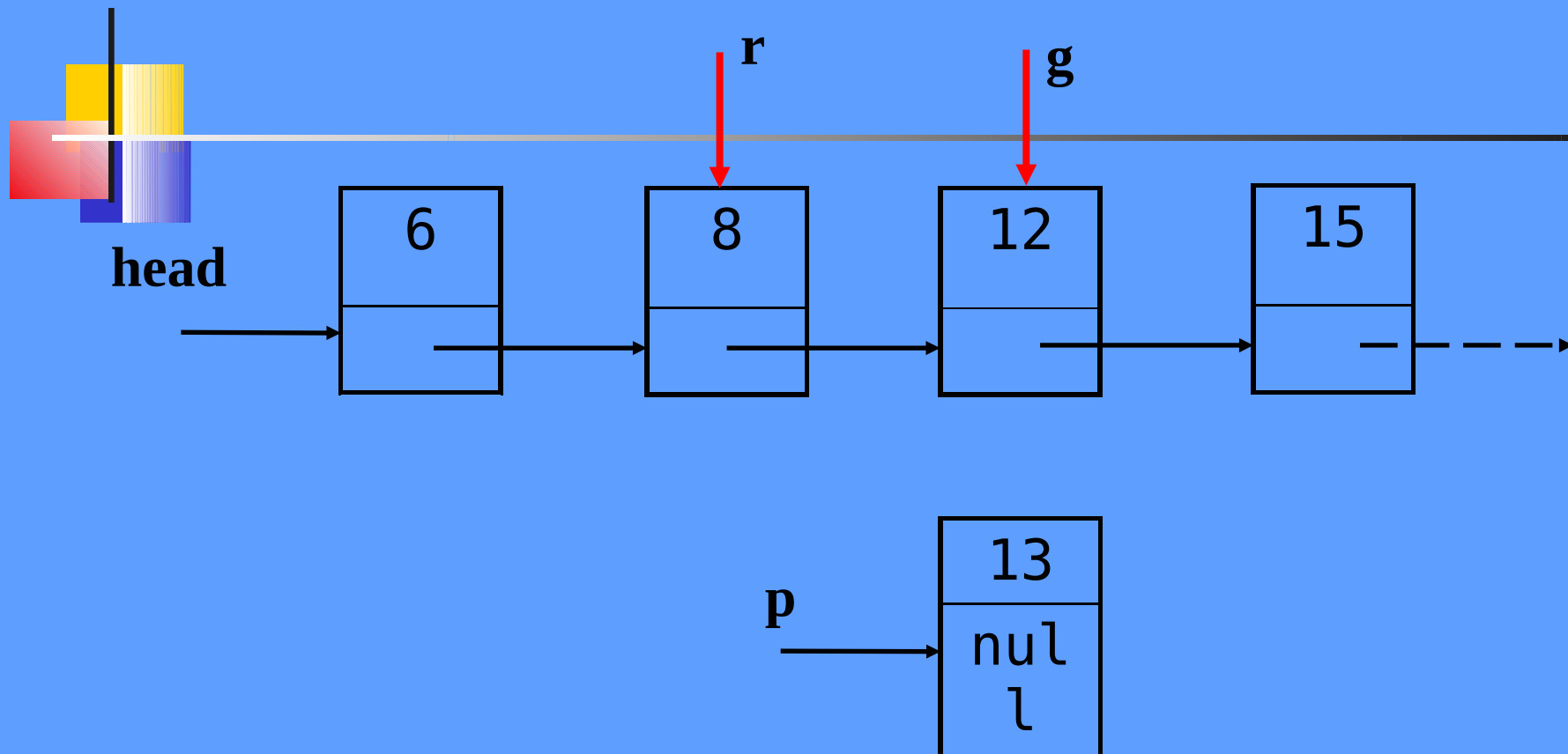
3、第三种情况，链表已建成，待插入结点 **p** 的数据比头结点的数据大，需要找到正确的插入位置。这时，可以借助两个结构指针 **r** 和 **g**，利用循环比较来找到正确位置。然后将结点 **p** 插入到链表中正确的位置。

参见下面的图示

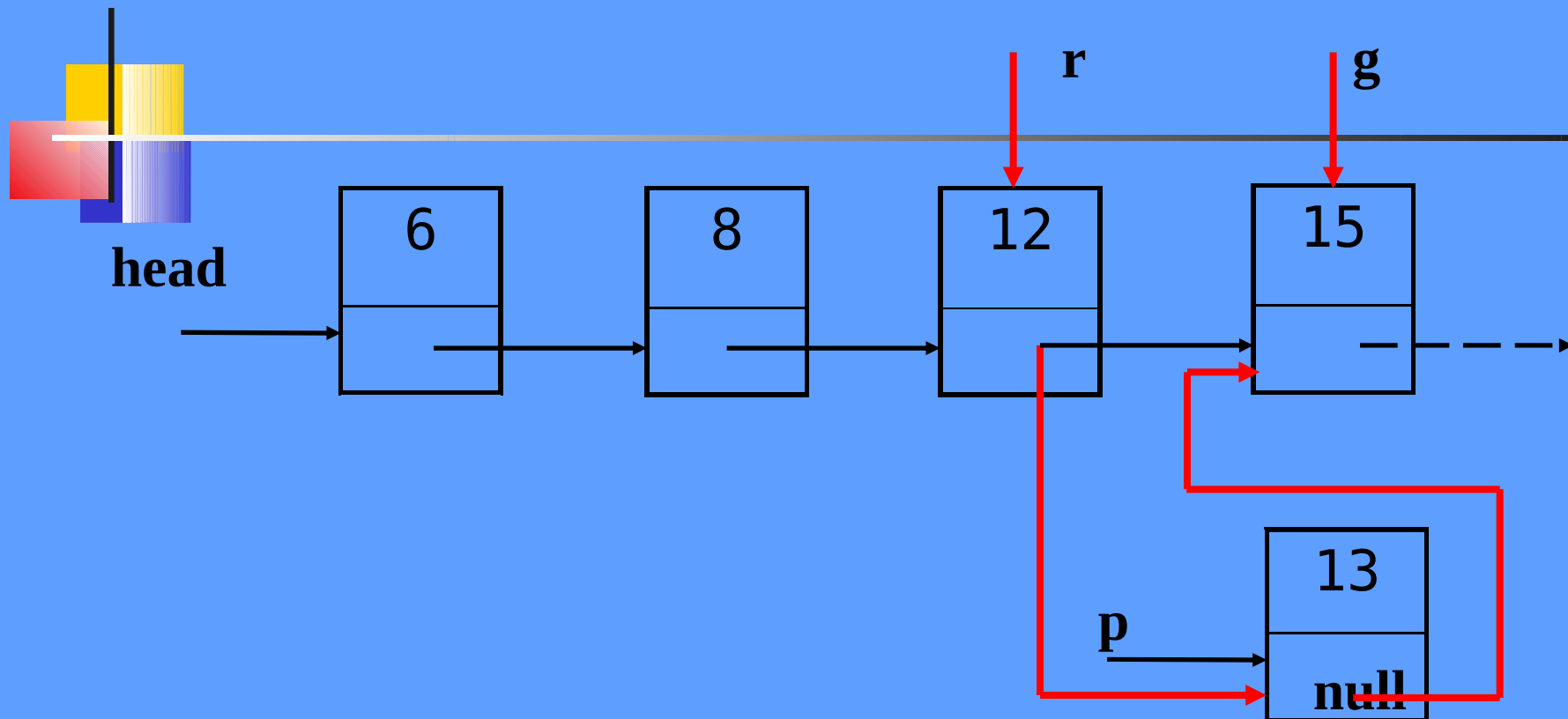




**说明：**这种情况下，**p** 结点已经与链表的第一个结点比较过了，所以从链表的下一个结点开始比较。**13>8**，继续比较。



说明：  $13 > 12$ ，继续比较。



**说明：**  $13 < 15$ ，找到了正确的插入位置，则插入结点 **p**；语句为：

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

# 参考程序

// 结构 7.c

```
#include <stdio.h>           // 预编译命令
#include <malloc.h>          // 内存空间分配
#define null 0               // 定义空指针常量
#define LEN sizeof(struct numST) // 定义常量，表示结构长度

struct numST                 // 结构声明
{
    int num;                 // 整型数
```

// 被调用函数 insert()，两个形参分别表示链表和待插入的结点

```
void insert (struct numST **phead, struct numST *p)
{
    struct numST *q,*r;
    if ((*phead)==null)
    {
        *phead = p;
        return;
    }
    else
    {
        // 第二种情况， p 结点 num 值小于链表头结点的 num 值
        if ( (*phead)->num > p->num)
        {
            // 将 p 结点插到链表头部
            p->next = *phead; // 将 p 的 next 指针指向链表头
            (*phead)
            // 将链表头赋值给 phead
```



```
// 第三种情况，循环查找正确位置
```

```
r = *phead; // r 赋值为链表头
```

```
q = (*phead)->next; // q 赋值为链表的下一个结点
```

```
while (q!=null) // 利用循环查找正确位置
```

```
{
```

```
    // 判断当前结点 num 是否小于 p 结点的 num
```

```
    if (q->num < p->num)
```

```
    {
```

```
        r = q; // r 赋值为 q，即指向 q 所指的结点
```

```
        q = q->next; // q 指向链表中相邻的下一个
```

结点

```
    }
```

```
    else // 找到了正确的位置
```

```
        break; // 退出循环
```


```
}
```

```
// 将 p 结点插入正确的位置
```

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

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

```
}
```



// 被调用函数，形参为 ST 结构指针，用于输出链表内容

```
void print(struct numST *head)
```

```
{
```

```
    int k=0;
```

```
    // 整型变量，用于计数
```

```
    struct numST * r;
```

```
    // 声明 r 为 ST 结构指针
```

```
    r=head;
```

```
    // r 赋值为 head，即指向链表头
```

```
    while(r != null)
```

```
    // 当型循环，链表指针不为空则继续
```

```
    续
```

```
    {
```

```
    // 循环体开始
```

```
        k=k+1;
```

```
    // 计数加 1
```

```
        printf("%d %d\n",k,r->num);
```

```
    r=r->next;
```

```
    // 取链表中相邻的下一个结点
```

```
    }
```

```
    // 循环体结束
```

```
}
```

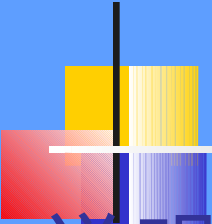
```
void main()                                // 主函数开始
{                                           // 函数体开始
    struct numST *head, *p;               // ST 型结构指针
    head = null;

    // 分配两个 ST 结构的内存空间，用于构造链表
    head = (struct numST *) malloc(LEN);
    head->next = (struct numST *) malloc(LEN);

    // 为链表中的两个结点中的 num 赋值为 5 和 10
    head->num = 5;
    head->next->num = 10;
    head->next->next = null;              // 链表尾赋值为空

    // 构造一个结点 p，用于插入链表
    p = (struct numST *) malloc(LEN);
    p->num = 8;
    p->next = null;
    insert(&head, p);                    // 调用 create 函数建立链表，
    print(head);                          // 调用 print 函数，输出链表内容
}                                           // 主函数结束
```





---

**说明：**函数 `insert()` 的第一个形参为 `struct numST**` 类型，即“指针的指针”。调用时送入的实参是链表头指针的地址，即程序中的 `&head`。这样对 `head` 的修改才会在函数返回后仍有效。如果形参为 `struct numST*`，则传入的为指针，当函数返回后，`head` 无法改变。

# 11.8 共用体

## 构造类型之二——联合

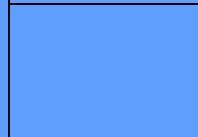
在同一存储单元里，根据需要放不同类型的数据，  
使用覆盖技术。

## 11.8.1 概念

单元起始地址：**1000** 。三个变量（数据）占  
用同一单元：**1000** —— **1003**



整型（ **2Byte** ）



字符型（ **1 byte** ）



浮点型（ **4 byte** ）

# 共用体变量的定义

格式（一般形式）：

```
union 联合类型名  
{ 成员列表  
} 变量列表；
```

## 11.8.2 共用体变量的引用方式

同结构类型变量的引用格式：

**变量名 . 成员名**



---

格式与结构类型的定义和变量声明形式上类似，但实质上有区别：

1. 结构类型的长度 = 各成员的长度和；  
各成员占独立的存储单元，不共享；
2. 联合类型的长度为成员中长度的最大者，各成员共享长度最大的存储单元；

## 11.8.3 共用体类型数据的特点

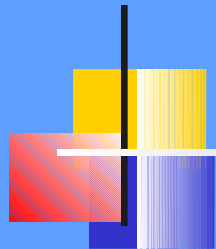
- 虽然同一内存单元内可以存放不同类型（同一地址）、不同长度的数据，但任一时刻，只有一种类型数据（最后赋值的）起作用；其它的都没有意义；
- 不能对共用体变量整体赋值，也不能对其初始化。
- 共用变量不可作为函数的参数，但可以通过指针指向；
- 共用体类型可以和结构类型 / 数组类型互为基类型； **p289**

```
struct
{ int num;
  char name[10];
  char sex; char job;
  union
    {int class;  char position[10];
     }category;
}person[2];
main()
{ int n,i;
  for(i=0;i<2 ;i++);
    {scanf("%d , %s , %c , %c",
&person[i].num,person[i].name,&person
[i].sex,&person[i].job);
```

续

```
if(person[i].job=='s')
    scanf("%d",&person[i].category.class);
else if(person[i].job=='t')
    scanf("%s",person[i].category.position);
else printf("input error!");    }
printf("\n");
printf("no.          name          sex    job
class/position\n");
for(i=0;i<2;i++)
{ if(person[i].job=='s')
    printf("%-6d %-10s %-3c %-3c %-
6d\n",person[i].num,person[i].name,person[i].
sex,person[i].job,person[i].category.class);
else
    printf("%-6d %-10s %-3c %-3c %-
6s\n",person[i].num,person[i].name,person[i].
sex,person[i].job,person[i].category.position);
}
}
```





# 枚举类型

----- 构造类型之三



## 11.9 枚举类型

枚举类型是指能将类型所包含的值一一列举出来。枚举值称为枚举常量

定义枚举类型的关键字是 **enum**。其类型的定义以及变量的声明同结构类型和联合类型；

声明格式：

```
enum  
weekday(sum,mon,tue,wed,thu,fri,sat);
```

定义变量：

```
enum weekday workday,week_end;
```



## 关于枚举类型变量

- 在 C 编译中，对枚举元素按常量处理；
- 对枚举型变量的赋值（枚举型变量的取值）只能取该变量所属枚举类型的枚举常量值；
- 一个整数不能直接赋给一个枚举变量。进行强制性转换；

# 说 明

( 1 ) 枚举型仅适应于取值有限的数。

例如，根据现行的历法规定，1 周 7 天，1 年 12 个月。

( 2 ) 取值表中的值称为枚举元素，其含义由程序解释。

例如，不是因为写成“Sun”就自动代表“星期天”。事实上，枚举元素用什么表示都可以。

( 3 ) 枚举元素作为常量是有值的——定义时的顺序号（从 0 开始），所以枚举元素可以进行比较，比较规则是：序号大者为大！

例如，上例中的 Sun=0、Mon=1、.....、Sat=6，所以 Mon>Sun、Sat 最大。

( 4 ) 枚举元素的值也是可以人为改变的：在定义时由程序指定。

例如，如果 enum weekdays {Sun= 7, Mon = 1, Tue, Wed, Thu, Fri, Sat}; 则 Sun= 7, Mon= 1, 从 Tue=2 开始，依次增 1。

## 例 题 13

**/\*file1.c 文件 1\*/**

**main()**

```
{ extern enter-string(char str[80]);  
  extern delete-string(char str[],char ch);  
  extern print-string(char str[]);  
  char c; char str[80];  
  enter_string(str); scanf("%c",&c);  
  delete_string(str,c); print_string(str);}
```

**/\* file2.c 文件 2\*/**

**#include<stdio.h>**

**enter\_string(char str[80])**

**{gets(str);}**



续

```
for(i=0;i<2;i++)
{if(person[i].job=='s')
    printf("%-6d  %-10s  %-3c  %-3c  %-6d\n",person[i].num,person[i].name,person[i].sex,person[i].job,person[i].category.class);
    else
        printf("%-6d  %-10s  %-3c  %-3c  %-6s\n",person[i].num,person[i].name,person[i].sex,person[i].job,person[i].category.position);
    } }
```

## 11.10 用 typedef 为类型定义新名字

除可直接使用 C 提供的标准类型和自定义的类型（结构、共用、枚举）外，也可使用 **typedef** 定义已有类型的别名。该别名与标准类型名一样，可用来定义相应的变量。

定义已有类型别名的方法如下：

- （ 1 ）按定义变量的方法，写出定义体；
- （ 2 ）将变量名换成别名；
- （ 3 ）在定义体最前面加上 **typedef**。

任何已有的类型可以重新命名

```
typedef long integer;
```

```
// 将 long 重新命名为 integer，使得 integer  
和 long 同等使用
```

可以和新类型定义一起定义名字

```
typedef int ARR[10] ;
```

```
// 定义了一个数组名 ARR，它是具有 10 个元素的  
整型数组类型
```

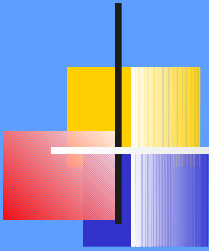
```
typedef struct{ int num;
```

```
float score;
```

```
} S; /* 定义结构体别名为 S*/
```

```
STUDENT stu1;
```





## 讨论： **typedef** 和 **#define**

### 说明：

- ( 1 ) 用 **typedef** 只是给已有类型增加 1 个别名，并不能创造 1 个新的类型。就如同人一样，除学名外，可以再取一个小名（或雅号），但并不能创造出另一个人来。
- ( 2 ) **typedef** 与 **#define** 有相似之处，但二者是不同的：前者是由编译器在编译时处理的；后者是由编译预处理器在编译预处理时处理的，而且只能作简单的字符串替换。



# 结构体与共体例子

```
struct TM
```

```
{
```

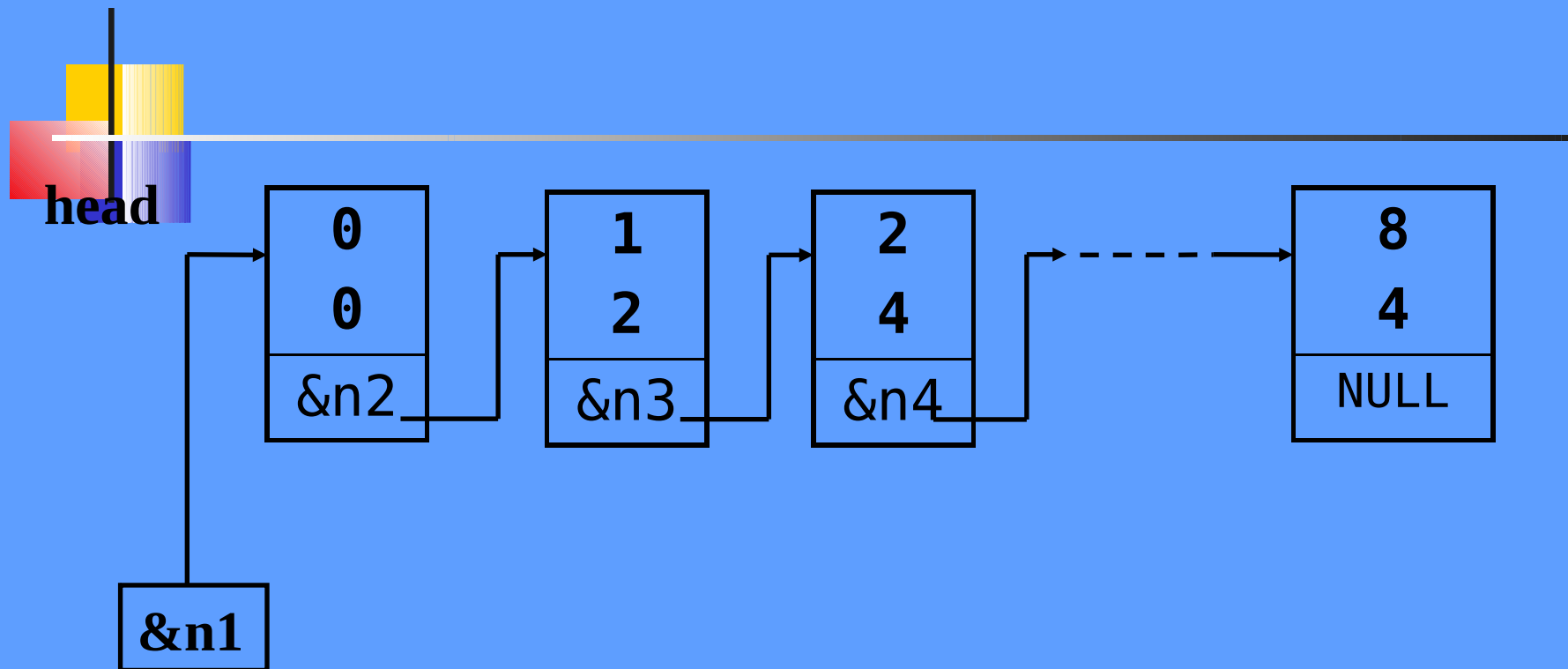
```
    int x,y;  
    数型
```

// 结构 TM 的成员， x,y 为整

```
    struct TM * next // 结构 TM 的成员，属 TM
```

型  
下面的表是马的跳步方案，从左下角跳到右  
上角  
}

结点	n1	n2	n3	n4	n5	n6	n7
x	0	1	2	4	6	7	8
y	0	2	4	3	4	2	4



**NULL** 为空地址

下面是形成链表的一个参考程序



## // 结构 1.c

```
#include <stdio.h>
```

```
#define null 0
```

```
struct TM
```

```
{
```

```
    int x,y;
```

```
    struct TM * next;
```

```
};
```

```
void main()
```

```
{
```

```
    int i;
```

```
// 声明 TM 结构 n1~n7, 结构指针 head,p
```

```
struct TM n1,n2,n3,n4,n5,n6,n7, * head, * p;
```

```
// 预编译命令
```

```
// 定义空指针常量
```

```
// 定义结构 TM
```

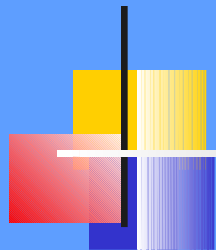
```
// 整型变量 x,y
```

```
// 指向 TM 结构的指针
```

```
// 主函数
```

```
// 主函数开始
```

```
// 声明整型变量
```



```
// 分别对 TM 结构 n1~n7 中的  
x,y 赋值
```

```
n1.x=0;n1.y=0;  
n2.x=1;n2.y=2;  
n3.x=2;n3.y=4;  
n4.x=4;n4.y=4;  
n5.x=6;n5.y=4;  
n6.x=7;n6.y=2;  
n7.x=8;n7.y=4;
```

```
// head 赋值为 n1 , 即 head 指  
向 n1
```

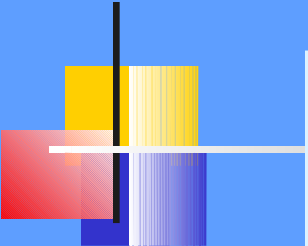
```
head = n1;
```

```
// n1~n7 构成链表
```

```
n1.next=&n2;  
n2.next=&n3;  
n3.next=&n4;  
n4.next=&n5;  
n5.next=&n6;  
n6.next=&n7;
```

```
// n7 的 next 指针赋值为空指  
针
```

```
n7.next=null;
```



```
p=head;      // p 赋值为 head , 即 p 指向  
head 所指的内容  
i=1;      // i 赋值为 1
```

```
do          // 直到型循环  
{          // 循环体开始
```

```
    // 输出结点信息  
    printf(" 结点 %d: x=%d, y=  
%d\n",i,p->x,p->y);  
    p=p->next;          // p 指向下一个  
结点  
    i=i+1;          // 计数加 1
```

```
} while(p!=null); // 未到达链表尾部, 则  
继续循环
```



用结构数组，利用键盘输入结点中的数据。

重点看

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

```
n[i].x=a;
```

结构数组，数组中的元素为结构类型的数据  
，如 **n[8]**

// 结构 2.c

```
#include <stdio.h>
```

```
#define null 0
```

```
struct TM
```

```
{
```

```
    int x,y;
```

```
    struct TM *next;
```

```
};
```

```
// 预编译命令
```

```
// 定义空指针常量
```

```
// 定义 TM 结构
```

```
// 整型变量 x,y
```

```
// 指向 TM 结构的指针
```

```
void main()
```

```
{
```

```
    int i,a,b;
```

```
// 主函数
```

```
// 主函数开始
```

```
// 声明整型变量 i,a,b
```

```
// 声明 TM 型结构数组 n[8] , TM 结构指针 head,p
```

```
struct TM n[8],*head,*p;
```

```
for(i=1;i<=7;i=i+1)
```

```
// 循环
```

```
{
```

```
// 循环体开始
```

```
    printf(" 输入 n[%d] 的 x\n",i);    // 提示输入第 i 个结构的  
    的 x 值
```

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

```
// 输入 a
```

```
    n[i].x=a;
```

```
// 将 a 的值赋给结构 n[i] 的元
```

```
    素 x
```

```
    printf(" 输入 n[%d] 的 y\n",i);    // 提示输入第 i 个结构的  
    的 y 值
```


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

```
// 输入 b
```

```
    n[i].y=b;
```

```
// 将 b 的值赋给结构 n[i] 的
```





```
head=&n[1];           // 链表头部指向 n[1]
for(i=1;i<=6;i=i+1)   // 将结构数组 n 形成链表
{
    n[i].next=&n[i+1]; // n[i] 的 next 指针指向下一个结构
    n[i+1]
}
n[7].next=null;       // 链表尾部指向空

p=head;               // p 指向链表头部 head
i=1;                  // i 赋值为 1
do                    // 直到型循环，用于输出链表内容
{
    // 输出结点内容
    printf(" 结点 %d: x=%d, y=%d\n",i,p->x,p->y);

    p=p->next;         // p 指向相邻的下一个结点
    i=i+1;             // 计数 i 加 1
} while(p!=null);     // 未到链表尾部，则继续循环
```

下面的程序与上面的程序区别仅在

```
scanf("%d",&(n[i].x));
```

去替换

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

```
n[i].x=a;
```

**// 结构 3.c**

```
#include <stdio.h>
```

```
#define null 0
```

```
struct TM
```

```
{
```

```
    int x,y;
```

```
    struct TM *next;
```

```
};
```

```
// 预编译命令
```

```
// 定义空指针常量
```

```
// 定义 TM 结构
```

```
// 整型变量 x,y
```

```
// 指向 TM 结构的指针
```



```
void main()
```

```
{
```

```
    int i,a,b;
```

```
// 主函数
```

```
// 主函数开始
```

```
// 声明整型变量 i,a,b
```

```
// 声明 TM 型结构数组 n[8] , TM 结构指针 head,p
```

```
struct TM n[8],*head,*p;
```

```
for(i=1;i<=7;i=i+1) // 循环
```

```
{
```

```
// 循环体开始
```

```
    printf(" 输入 n[%d] 的 x\n",i);    // 提示输入第 i 个结构的  
    的 x 值
```

```
    scanf("%d",&(n[i].x));    // 输入 n[i].x
```

```
    printf(" 输入 n[%d] 的 y\n",i);    // 提示输入第 i 个结构的  
    的 y 值
```

```
    scanf("%d",&(n[i].y));    // 输入 n[i].y
```

```
}
```

```
// 循环体结束
```

```
head=&n[1];           // 链表头部指向 n[1]
for(i=1;i<=6;i=i+1)  // 循环
{                    // 循环体开始
    n[i].next=&n[i+1]; // n[i].next 指向 n[i+1]
}                  // 循环体结束
n[7].next=null;     // 链表尾部赋值为空指针

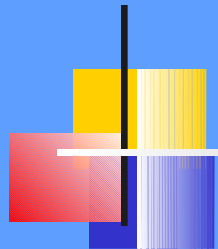
p=head;             // p 指向链表头部 head
i=1;                // i 赋值为 1

do                  // 直到型循环
{                  // 循环体开始
    // 提示输入结点信息
    printf(" 结点 %d: x=%d, y=%d\n",i,(*p).x,(*p).y);
    p=(*p).next;   // p 指向相邻的下一个结点
    i=i+1;          // i 加 1
} while(p!=null);  // 未到达链表尾部，则继续循环
}                  // 主函数结束
```

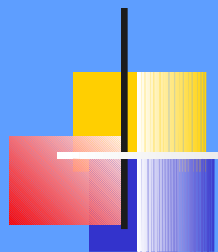
## 任 务

我们要作一张登记表，登记排队求职信息，包括：姓名、年龄、性别、电话四个参数。希望便于管理，即可以插入和删除，这时可用队列，采用结构类型变量。

```
struct ST
{
    char name[20];    // 字符串，姓名
    int  age;         // 整数，年龄
    char sex;         // 字符，性别
    long num;         // 电话号码
    struct ST *next;  // ST 结构的指针
};                  // 注意，这里必须有分号
```



# 循 环 链 表



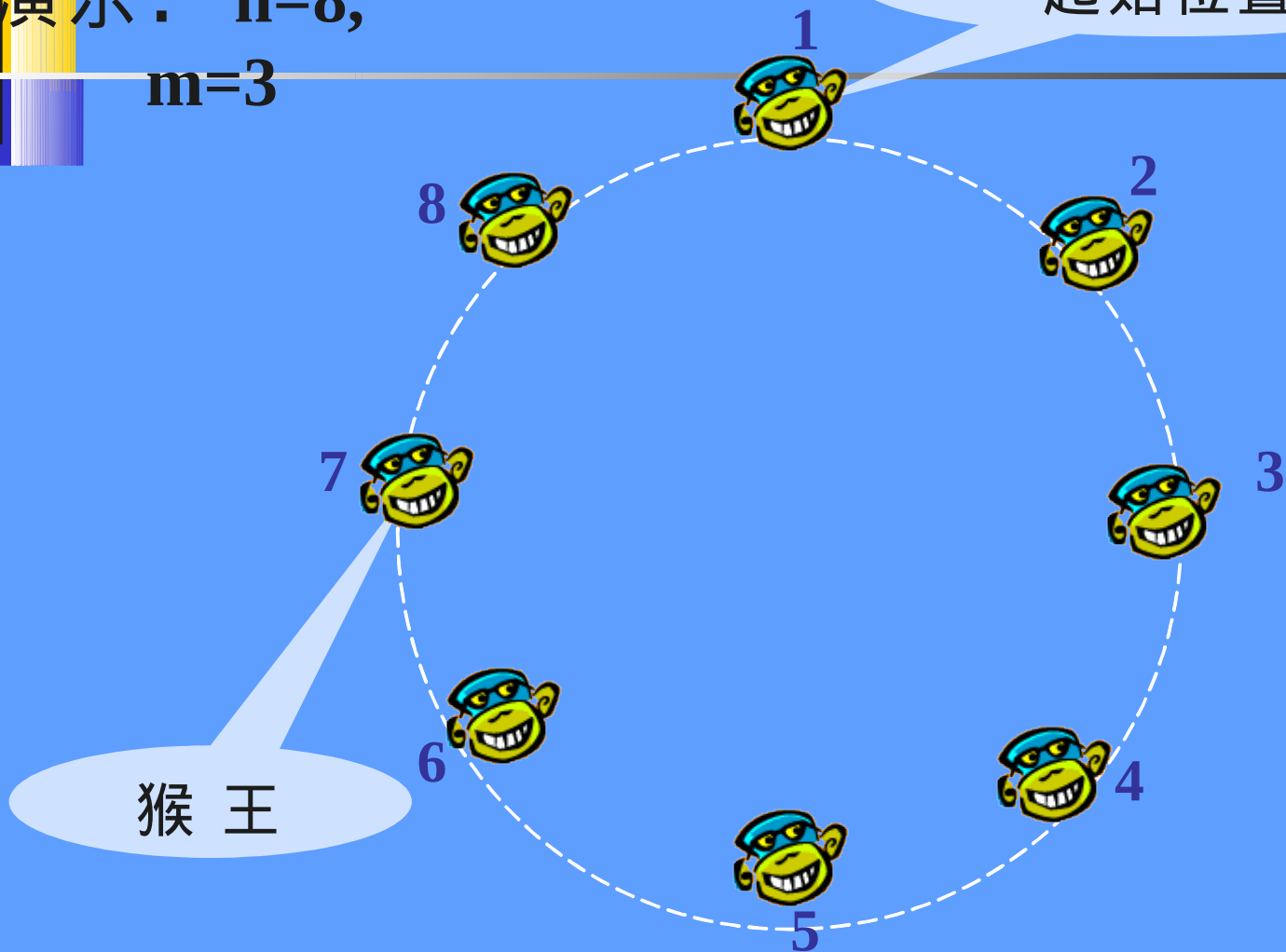
例：猴子选大王。

$n$  只猴子围成一圈，顺时针方向从 1 到  $n$  编号。之后从 1 号开始沿顺时针方向让猴子从 1, 2, ...,  $m$  依次报数，凡报到  $m$  的猴子，都让其出圈，取消候选资格。然后不停地按顺时针方向逐一让报出  $m$  者出圈，最后剩下一个就是猴王。



演示:  $n=8$ ,  
 $m=3$

起始位置



猴子被淘汰的顺序 3 6 1 5 2 8 4





## 说明：

---

如图 1 所示有 8 只猴子围成一圈，  $m=3$ 。从 1# 猴的位置开始，顺时针 1 至 3 报数，第一个出圈的是 3#；第二个出圈的是 6#，第 3 个出圈的是 1#；第 4 个出圈的是 5#；第 5 个是 2#，第 6 个是 8#；第 7 个是 4#。最后剩下一个是 7#，它就是猴王。

我们用循环链表来模拟这个选择过程。

## 1、定义一个名为 **mon** 的结构

```
struct mon
```

```
{  
    int num;           // 整数，表示猴子的编号  
    struct mon *next; // 指针，指向相邻的下一只猴子  
}
```

## 2、将链表的头指针 **head** 定义为全局变量。

```
struct mon*head;
```

## 3、主函数

用键盘输入猴子数 **n**，输入数 **m**，调用函数 **create** 建立一个循环链表，模拟众猴围成一圈的情况。该函数的实参为 **n**。调用函数 **select**，模拟 1 至 **m** 报数，让 **n-1** 只猴子逐一出列的过程。即在具有 **n** 个结点的循环链表按报数 **m** 删除结点的过程。该函数的实参为 **head** 和 **m**。最后输出剩下的结点的编号。

#### 4、建立循环链表的函数 `create(int nn)`

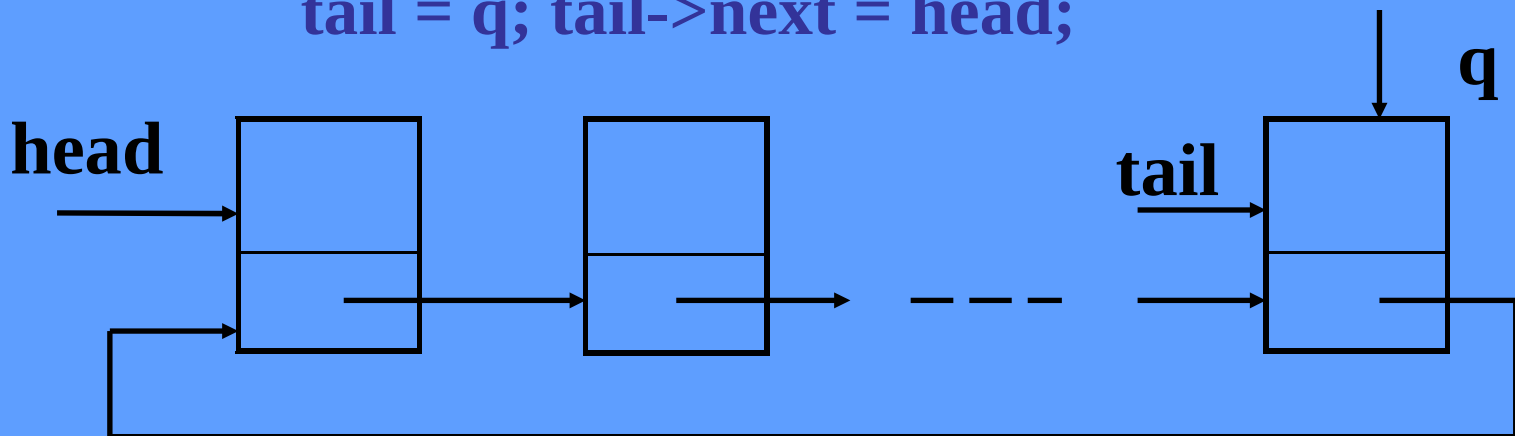
其中 `nn` 为形式参数。要从编号 1 到编号 `nn`。思路是

(1) 先做第 1 个结点，让其中的数据域 `p->num` 赋值为 1，让指针域赋值为 `null`。之后让链头指针 `head` 指向第 1 个结点。利用指针 `q` 记住这个结点，以便让指针 `p` 去生成下面的结点。

(2) 利用一个计数循环结构，做出第 2 个结点到第 `nn` 个结点。并将相邻结点一个接一个链接到一起。

(3) 最后一个结点要和头结点用下一语句链接到一起

`tail = q; tail->next = head;`



## 5、删结点的函数 select(int mm)

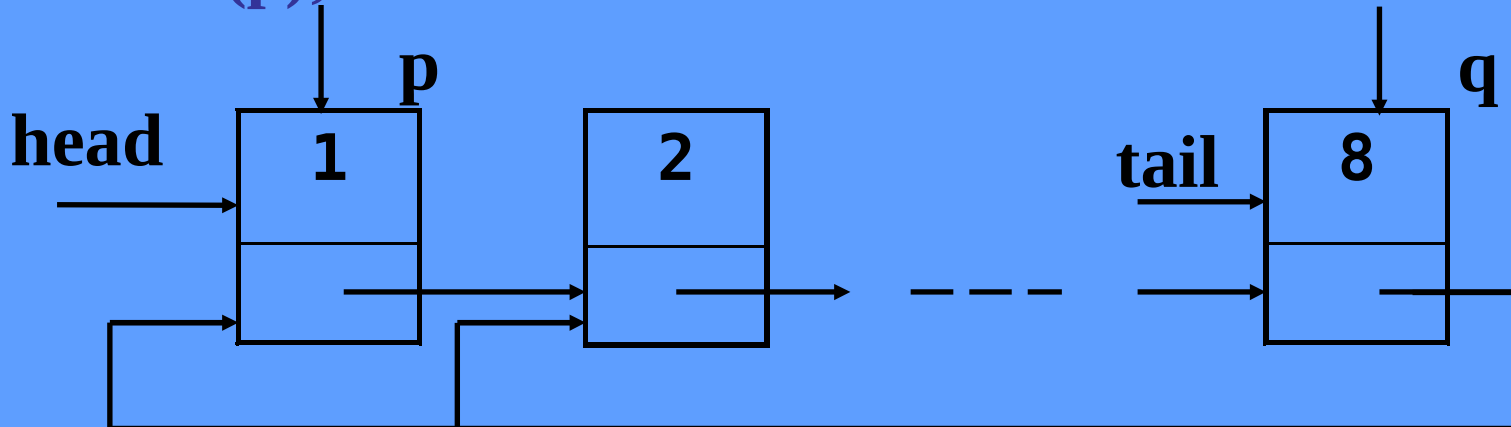
mm 为形式参数，从 1 至 m 报数，凡报到 mm 者删除其所在的结点。

设计两个指针 p 和 q。一开始让 q 指向链表的尾部 q=tail。让 p 指向 q 的下一个结点。开始时让 p 指向 1# 猴所在的结点。用一个累加器 x，初始时 x=0，从 1# 猴所在结点开始让 x=x+1=1，如果 mm 是 1 的话，1# 猴所在的 p 结点就要被删除。有三条语句

```
printf(" 被删掉的猴子号为 %d 号 \n",p->num);
```

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

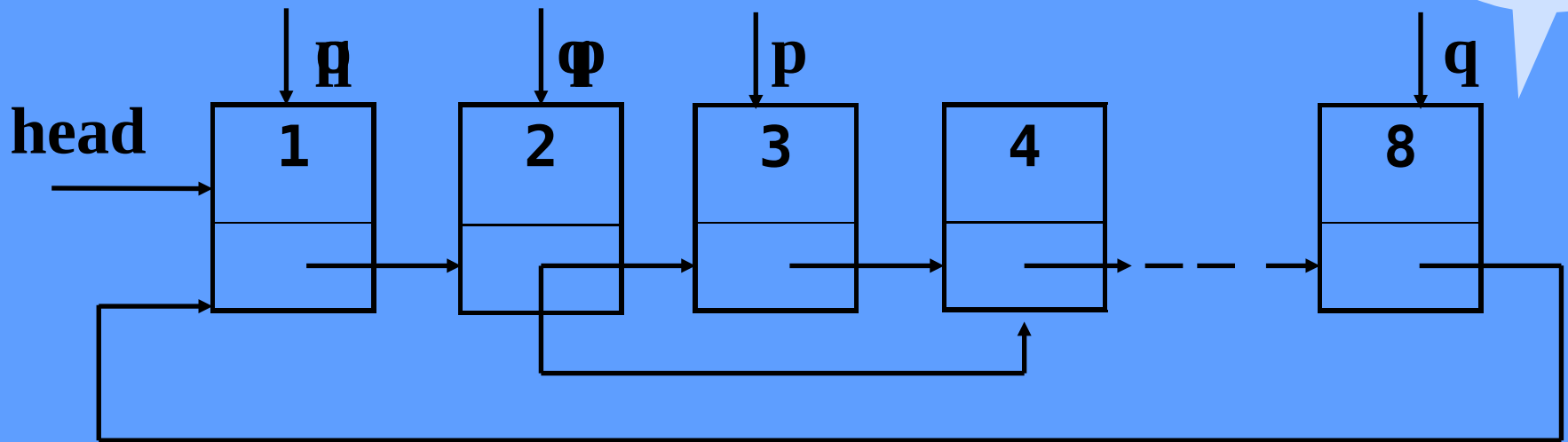
```
free(p);
```



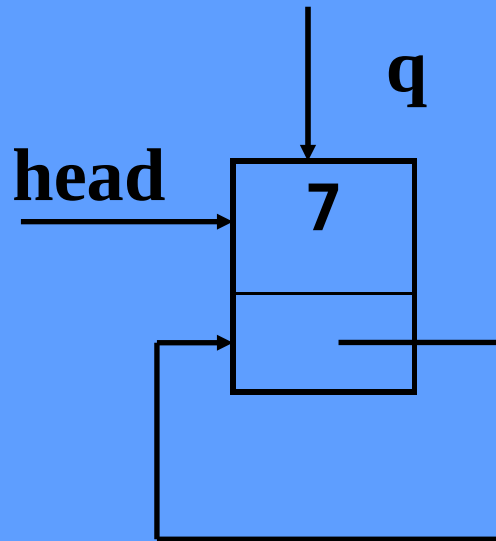
演示

这里 `free(p)` 是释放 `p` 结点所占用的内存空间的语句。如果 `mm` 不是 1 而是 3，程序会在 `do-while` 循环中，让 `x` 加两次 1，`q` 和 `p` 一起移动两次，`p` 指向 3# 所在结点，`q` 指向 2# 所在结点，之后仍然用上述三条语句删去 3# 所在的结点。

演示



这个 **do-while** 循环的退出条件是  **$q == q \rightarrow next$** 。即当只剩下一个结点时才退出循环。当然猴王非其莫属了。这时，让头指针 **head** 指向 **q**，**head** 是全局变量，在主程序最后输出猴王时要用 **head->num**。



参考程序如下：



```
#include <stdio.h>
```

```
// 预编译命令
```

```
#include <malloc.h>
```

```
// 内存空间分配
```

```
#define null 0
```

```
// 定义空指针常量
```

```
// 定义常量，表示结构长度
```

```
#define LEN sizeof(struct mon)
```

```
struct mon // 结构声明
```

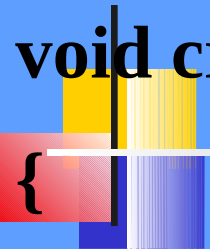
```
{
```

```
    int num; // 整型数，用于记录猴子号
```

```
    struct mon *next; // mon 结构指针
```

```
};
```

```
struct mon *head, *tail; // mon 结构指针，全局变量
```



```
void create(int nn)      // 被调用函数
{                        // 函数体开始
    int i;               // 整型变量 i，用于计数
    struct mon *p,*q;    // 声明 mon 结构指针 p， q

    // 为 p 分配内存空间
    p=(struct mon *) malloc(LEN);

    p->num=1;            // 初始化 p 结点 num 域为 1
    p->next=null;        // 初始化 p 结点 next 域为空
    head=p;             // 链表头指针 head 赋值为 p
    q=p;                // q 赋值为 p
```



**for(i=2;i<=nn;i=i+1) // 利用循环结构构造链表**

**{ // 循环体开始**

**p=(struct mon \*)malloc(LEN); // 为 p 分配内存空间**

**p->num=i; // 初始化 p 结点 num 域为 i，表示猴子号**

**q->next=p; // 将 p 结点加到链表尾部**

**q=p; // 让 q 指向链表尾部结点**

**p->next=null; // 链表尾部指向空**

**} // 循环体结束**

**tail = q; // 链表尾**

**tail->next=head; // 链表尾部指向链表头，**

**// 形成循环链表**

// 被调用函数 select , mm 表示结点删除间隔

void select(int mm)

{

int x=0;

struct mon \*p,\*q;

q=tail;

do

结点

{

p=q->next;

x=x+1;

if(x % mm==0)

{

// 输出被删掉的猴子号

printf(" 被删掉的猴子号为 %d 号 \n",p-

>num);

q->next=p->next; // 删除此结点

free(p); // 释放空间

}

else q=p;

}while(q!=q->next);

head = q;

// 函数体开始

// 声明整型值 x , 并初始化为 0

// 声明结构指针 p , q

// q 赋值为 tail , 指向循环链表尾部

// 直到型循环, 用于循环删除指定间隔的

// 循环体开始

// p 赋值为 q 相邻的下一个结点

// x 加 1

// x 是否整除 mm ,

// 表示是否跳过指定间隔

// q 指向相邻的下一个结点 p

// 剩余结点数不为 1 , 则继续循环

// head 指向结点 head 为链表中剩余一个

```

void main()                                // 主函数开始
{                                           // 函数体开始
    int n,m;                               // 声明整型变量 n,m
    head = null;                           // 初始化 head 为空

    printf(" 请输入猴子数 \n");           // 提示信息
    scanf("%d",&n);                       // 输入待插入结点数据

    printf(" 请输入间隔 m\n");            // 提示信息
    scanf("%d",&m);                       // 输入间隔
    create(n);                             // 调用函数 create 建立循环链表

    select(m);                             // 调用函数 select , 找出剩下的猴子
    printf(" 猴王是 %d 号 \n",head->num); // 输出
    猴王
}                                           // 函数体结束

```