

第9章 结构与联合

结构与联合都属于C的构造类型。

- 某种具体的结构说明（关键字**struct**）
- 结构变量的定义
- 结构成员的引用（.运算符）
- 通过结构指针引用结构成员（->运算符）
- 字段结构
- 联合类型（**union**）
- 链表



结构的声明和引用

■ **struct 结构类型名{**
 成员声明表
 };

```
struct plant{  
char name[10];  
double diameter;  
int moons;  
};
```

例1、输入平面上两个点，求它们之间的长度。

```
#include "stdio.h"
```

```
#include "math.h"
```

```
struct point{ /* 平面上点的结构类型说明 */
```

```
    int    x;
```

```
    int    y; /* x,y是点的坐标 */
```

```
};
```

```
int main(void)
```

```
{ struct point start, end; /* 声明点结构变量 */
```

```
    double dx,dy,length;
```

```
    printf("输入线段的起点和终点坐标: \n");
```

```
    scanf("%d%d%d%d",&start.x,&start.y, &end.x,&end.y);
```

```
    dx=(end.x-start.x)*(end.x-start.x);
```

```
    dy=(end.y-start.y)*(end.y-start.y);
```

```
    length=sqrt(dx+dy); /*计算线段长度*/
```

```
    printf("the length is %f\n",length); /*输出线段长度*/
```

```
    return 0;
```

```
}
```

结构成员的访问:

结构变量名. 成员名

例2、改写例1将求两点间的长度定义为函数

```
double  linelen(struct point  pt1,struct point  pt2)
{
    double dx,dy,length;
    dx=(pt2.x-pt1.x)*(pt2.x-pt1.x);
    dy=(pt2.y-pt1.y)*(pt2.y-pt1.y);
    length=sqrt(dx+dy);      /*计算线段长度*/
    return length;
}
```

结构变量作为函数的参数

```
#include "stdio.h"
#include "math.h"
struct point{ /* 平面上点的结构类型 */
    int    x;
    int    y; /* x,y是点的坐标 */
};
int main(void)
{ struct point start, end; /* 声明点结构变量 */
  double length;
  printf("输入线段的起点和终点坐标: \n");
  scanf("%d%d%d%d",&start.x,&start.y, &end.x,&end.y);
  length=linelen(start, end); /*计算线段长度*/
  printf("the length is %f\n",length); /*输出线段长度*/
  return 0;
}
```

例3、改写例2将函数参数改为结构类型的指针

```
double  linelen(struct point *p1,struct point *p2)
{
    double dx,dy,length;
    dx=(p2->x-p1->x)*(p2->x-p1->x);
    dy=(p2->y-p1->y)*(p2->y-p1->y);
    length=sqrt(dx+dy); /*计算线段长度*/
    return length;
}
```

- 结构类型的指针作为函数的参数
- 结构类型的指针也可以作为函数的返回值

```
#include "stdio.h"
#include "math.h"
struct point{ /* 平面上点的结构类型 */
    int    x;
    int    y; /* x,y是点的坐标 */
};
int main(void)
{ struct point start, end; /* 声明点结构变量 */
  double length;
  printf("输入线段的起点和终点坐标: \n");
  scanf("%d%d%d%d",&start.x,&start.y, &end.x,&end.y);
  length=linelen(&start, &end); /*计算线段长度*/
  printf("the length is %f\n",length); /*输出线段长度*/
  return 0;
}
```

结构类型的变量作为函数的参数时：



**1、将实参结构拷贝给形参，占用内存较多，
耗费时间较长。**

—— 适用于较小的结构

2、值传递，形参结构不影响实参结构的值。

结构类型的指针作为函数的参数时:

1、只将实参指针的值拷贝到形参指针单元。
占用内存少，耗费时间短。

—— 适用于较大的结构

2、在函数内部对形参指针的间访操作会影响
实参指针所指结构变量的值。

—— 适合于要修改实参指针所
指的结构变量值的情况。

通过结构指针访问结构成员

- `struct point a, b={10, 20}, *p=&a;`
 `*p=b;` `/* 与a=b;等价 */`
- 通过“*”用结构指针访问结构变量的成员
 $(\text{*结构指针}).\text{成员名}$
 $(\text{*p}).x \iff a.x$
- 通过运算符“->”访问结构变量的成员
 $\text{结构指针名} \rightarrow \text{结构成员名}$
 $p \rightarrow y \iff (\text{*p}).y$

结构变量的赋值操作

- 当两个结构变量的类型相同时，它们之间可以直接相互赋值。例如：

```
static struct point {
```

```
    int x;
```

```
    int y;
```

```
} a={1,2}, b;
```

则 **b=a;** /* 合法,对应的各个成员赋值 */

b={1,2}; /* 错 */

定义一个初始化点结构变量的函数

- 结构成员作为函数的参数
- 结构变量作为函数的返回值

```
point makepoint(int x, int y)
{
    struct point temp;
    temp.x = x;
    temp.y = y;
    return temp;
}
```

```

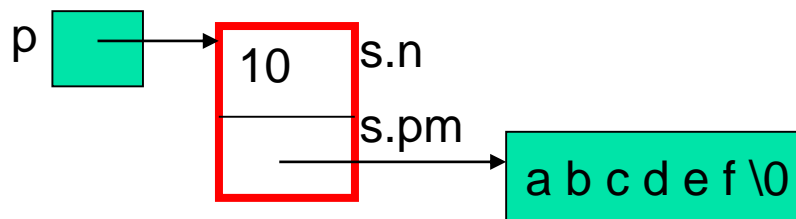
#include "stdio.h"
#include "math.h"
struct point{ /* 平面上点的结构类型 */
    int    x;
    int    y;    /* x,y是点的坐标 */
};
int main(void)
{ struct point start, end; /* 声明点结构变量 */
  double length;
  start=makepoint (0, 0) ;
  end=makepoint (2, 3) ;
  length=linelen(&start, &end); /*计算线段长度*/
  printf("the length is %f\n",length); /*输出线段长度*/
  return 0;
}

```

设有：

```
struct T{  
    int n;  
    char *pm;
```

```
}s={10, "abcdef" },*p=&s; 写出以下表达式的值和类型。  
（各表达式相互无关）
```



- 1) `++p->n` 11, int 访问n并使其增1
- 2) `p->n++` 10, int 访问n，取其原值参与运算，再使n增1
- 3) `*p->pm` 'a', char 访问pm所指字符 'a'
- 4) `*p->pm++` 'a', char 访问pm所指字符 'a'后pm增1指向 'b'
- 5) `*++p->pm` 'b', char 先访问pm，然后pm增1，再访问pm所指字符 'b'

- 例4 编程输入商品信息（包括商品编码、名称、价格），并且按照价格排序并显示排序后的结果。

商品编码	名称	价格（元）
1	笔	2.0
2	毛巾	10.5

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

```
#define N 3
```

```
struct GOODS {
```

```
    long code;          /* 货物编码 */
```

```
    char name[20];      /* 名称 */
```

```
    float price;        /* 价格 */
```

```
};
```

```
/* 输入n件物品的信息 */
```

```
void input(struct GOODS *p,int n);
```

```
/* 对n件物品按价格降序排序 */
```

```
void sort(struct GOODS *p,int n);
```

```
/*显示n件物品的信息 */
```

```
void display(struct GOODS *p,int n);
```



```
int main(void)
```

```
{
```

```
    struct GOODS g[N];
```

```
    input(g,N);    /* 结构数组名作为实参 */
```

```
    display(&g[0],N);
```

```
    sort (g,N);
```

```
    display(g,N);
```

```
    return 0;
```

```
}
```

/* 输入n件物品的信息 */

void input(struct GOODS *p,int n)

{

int i;

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

scanf("%ld",&p[i].code);

scanf("%s",(p+i)->name);

scanf("%f",&(p+i)->price;

}

}

/*显示n件物品的信息 */

void display(struct GOODS *p,int n)

```
{  
    int i;  
    for(i=0;i<n;i++){  
        printf("%ld\t",(*(p+i)).code);  
        printf("%s\t",(p+i)->name);  
        printf("%f\n",p[i].price);  
    }  
}
```

/* 对n件物品按价格降序排序 */

void sort(struct GOODS *p,int n)

{ int i,j; struct GOODS t;

for(i=0;i<n-1;i++)

for(j=i+1;j<n;j++)

if((p+i)->price<(p+j)->price){

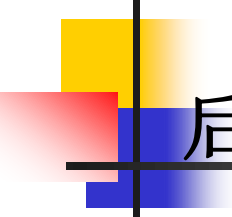
t=*(p+i);

***(p+i)=*(p+j);**

***(p+j)=t;**

}

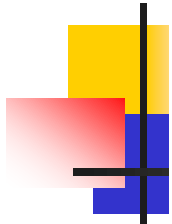
}



设计一个能够描述学生成绩的结构类型，然后声明对应的结构数组，描述以下成绩表。

成绩表

学号	姓名	性别	入学时间			计算机原理	英语	数学	音乐
			年	月	日				



定义

```
struct date
```

```
{
```

```
    int month;
```

```
    int da
```

```
    int year;
```

```
};
```

```
struct STUDENT
```

```
{
```

```
    int ID;
```

```
    char Name[10];
```

```
    char Sex[4];
```

```
    struct date timeOfEnter;
```

```
    int Computer;
```

```
    int English;
```

```
    int Math;
```

```
    int Music;
```

```
};
```

```
struct STUDENT stu[30];
```



初始化

```
struct STUDENT stu[30] = {  
    {1, " Tom  ", 'M', {1999, 12, 20}, 90, 83, 72, 82},  
    {2, " Nick  ", 'M', {1999, 07, 06}, 78, 92, 88, 78},  
    {3, " Sue   ", 'F', {1999, 07, 06}, 89, 72, 98, 66},  
    {4, " May   ", 'M', {1999, 07, 06}, 78, 95, 87, 90}  
};
```



嵌套结构中结构成员的访问

- **结构变量名.结构成员名.成员名**

- **理解为:**

(结构变量名.结构成员名) .成员名

stu[0].timeofenter.year

- 注意 “.” 、 “->” 、 “*” 、 “++或--” 运算符 优先级和结合性

- 设有说明：

```
char u[]="abcde";
```

```
char v[]="xyz";
```

```
struct T{
```

```
    int x;
```

```
    char c;
```

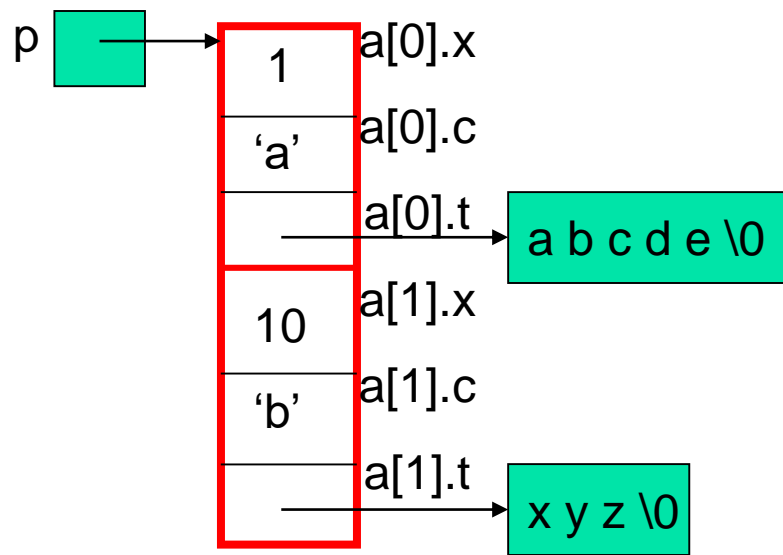
```
    char *t;
```

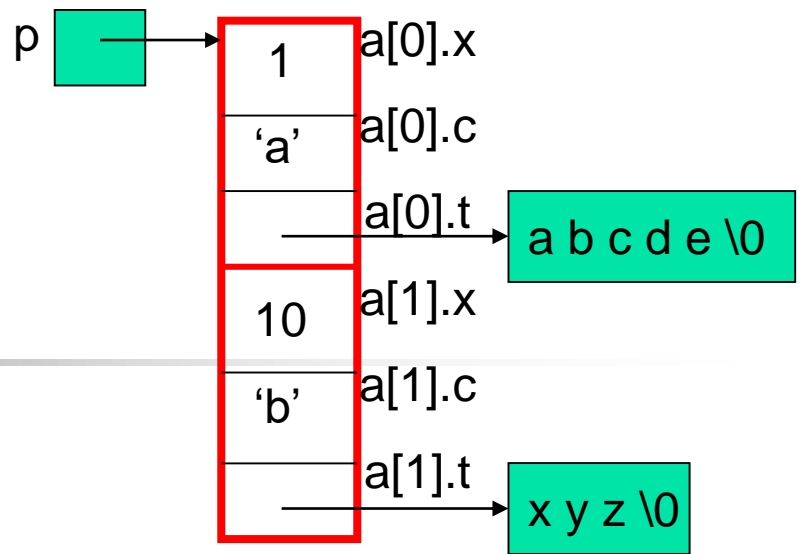
```
}a[]={ {1, 'a', u}, {10, 'b', v}}, *p=a;
```

- 若执行代码片段：

```
printf( "%d\n" ,(++p)->x);    /* 输出 10 */
```

```
printf( "%c\n",p->c);        /* 输出 b */
```





■ 若执行代码片段:

```
p=a;
```

```
printf("%d\n",(p++)->x);    /* 输出 1 */
```

```
printf("%d\n",p->x);        /* 输出 10 */
```

若执行代码片段:

```
p=a;
```

```
printf("%c\n",*p++->t);    /* 输出 a */
```

```
printf("%c\n",*p->t);      /* 输出 x */
```

比较下面各表达式的异同:

↗ X自增

不同

■ $(++p) \rightarrow x$ 与 $++p \rightarrow x$

↗ 先访问x, p增1

相同

■ $(p++) \rightarrow x$ 与 $p++ \rightarrow x$

↗ *作用在t上, ++作用在p上

相同

■ $*(p++) \rightarrow t$ 与 $*p++ \rightarrow t$

↗ ++作用在t上, 访问的是t[1]

不同

■ $*(++p) \rightarrow t$ 与 $*++p \rightarrow t$

相同

■ $*(++p \rightarrow t)$ 与 $*++p \rightarrow t$

不同

■ $*++p \rightarrow t$ 与 $++*p \rightarrow t$

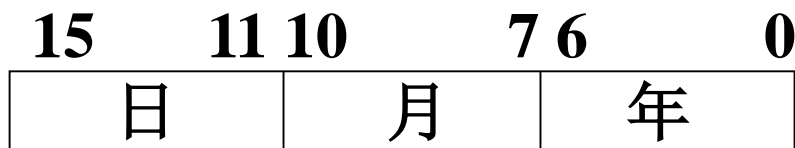
↗ ++作用在t上, 访问的是t[1]

↘ 访问t[0], ++作用在t[0]上

*9.8 字段 (bit field) 结构

【例2.16】 压缩和解压。把表示21世纪日期的日、月和年3个整数压缩成1个16位的整数。

分析： 因为日有31个值，月有12个值，年有100个值，所以可以在一个整数中用5b表示日，用4b表示月，用7b表示年。



用字段结构实现

字段结构的声明:

```
struct date {  
    unsigned short year : 7 ;    /* 年 */  
    unsigned short month : 4 ;   /* 月 */  
    unsigned short day : 5 ;     /* 日 */  
};
```

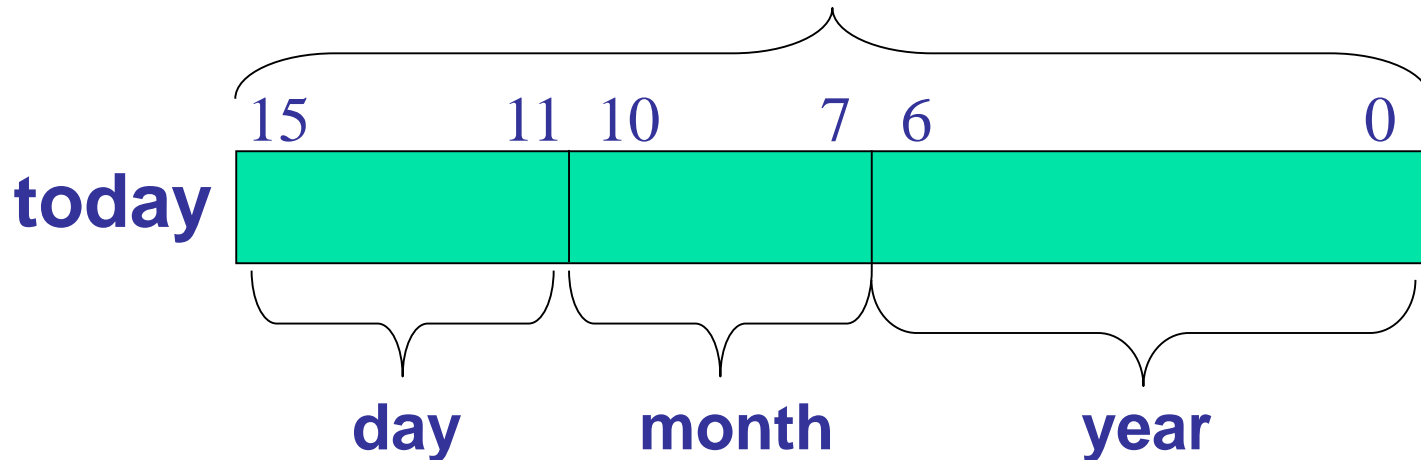
↑ ↑
字段名 字段宽度

字段的存储结构

```
struct date {  
    unsigned short year : 7 ;    /* 年 */  
    unsigned short month : 4 ;   /* 月 */  
    unsigned short day : 5 ;     /* 日 */  
};
```

struct date today; /* today是date 字段结构变量 */

2个字节



引用字段

- 与结构完全相同： . 或 ->
- 字段就是一个小整数，它可以出现在其它整数可以出现的任何地方。字段在参与运算时被自动转换为 int 或 unsigned int 类型的整数。

```
today. year=2013;  
today. month=5;  
today. day=9;
```

字段：字中一组相邻的二进制位，是指定了存储位数的，**unsigned int**（或**unsigned short**）的结构成员。

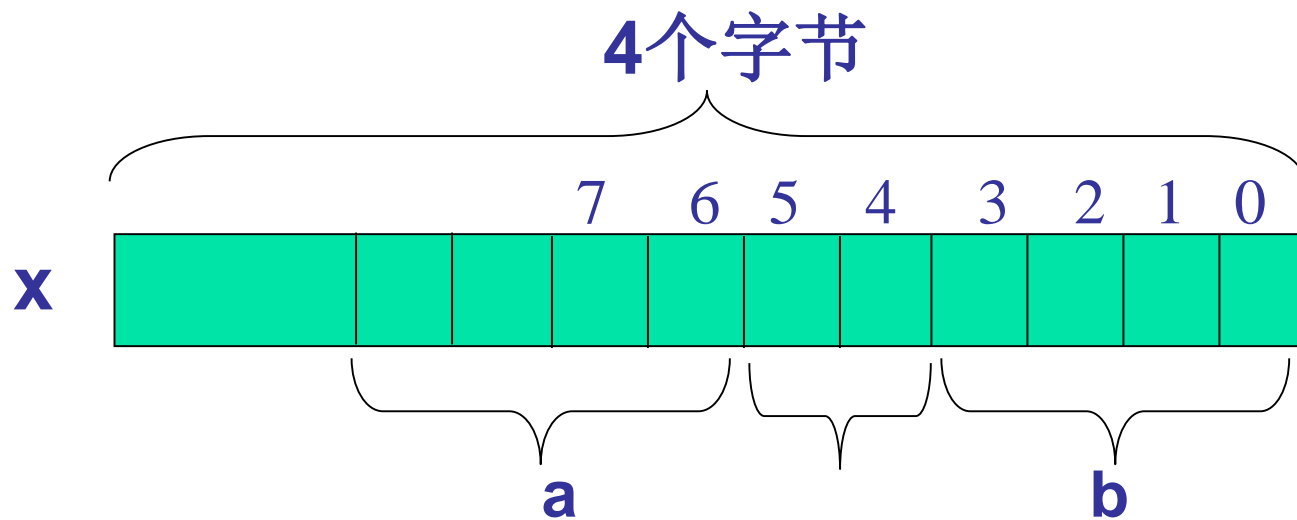
字段的宽度：组成字段的二进制位的数目，是一个非负的整型常量表达式。

字段结构在操作系统，编译程序，计算机接口的C语言编程方面使用较多。

可以用一个字段结构变量表示多个值很小的整数，通过字段名对相应的小整数进行操作。

无名位段（匿名）

```
struct {  
    unsigned a: 4;  
    : 2;  
    unsigned b: 4;  
} x;
```



9.7 联合

- **结构变量**是占据各自不同空间的各成员变量的集合。
- **如何让几个变量共享同一存储区？**
- **联合变量**是占用同一块内存空间的各成员变量的集合。

- 与结构类似，联合类型也是一种构造类型。一个联合类型中包含有多个成员，**这些成员共享共同的存储区域**，但这些成员并不同时存在，联合**存储区域的大小由各个成员中所占字节数最大的成员决定**，在任何时刻，各个成员中**只能有一个成员拥有该存储**。
- 除了用**关键字union**取代**struct**之外，联合类型的定义、联合变量的声明、以及联合成员的引用在语法上**与结构完全相同**。

- 联合变量的声明、初始化及联合成员的引用与结构完全相同，但是只能对联合的第1个成员进行初始化（早期C标准）。

```
union chl {  
    char c;  
    short h;  
    long l;  
} v = {'9'};  
union chl w = {'a'};
```

C99：可以通过指定成员实现对联合变量的任意成员进行初始化

字段结构与联合的应用



如何访问**16位字**中的高低字节和各二进制位？

- ◆ 定义**8位宽**的字段**byte0**、**byte1**
 - 表示一个**16位字**中的高/低字节
- ◆ 定义**1位宽**的字段**b0~b15**
 - 表示一个**16位字**中的**bit**
- ◆ 定义联合类型
 - 使一个**short变量**、**2个byte字段**与**16个bit字段**共享存储。

例9.12 用字段和联合访问一个16位字中的高低字节和各二进制位。

```
#include<stdio.h>
#define CHAR_BIT 8
struct w16_bytes{
    unsigned short byte0:8; /* byte0: 低字节*/
    unsigned short byte1:8; /* byte1: 高字节 */
};
struct w16_bits{
    unsigned short b0:1,b1:1,b2:1,b3:1,b4:1,b5:1,b6:1,b7:1,
        b8:1,b9:1,b10:1,b11:1,b12:1,b13:1,b14:1,b15:1;
};
```

union w16{ /* 短整型变量、结构成员byte、结构成员bit共享共同的存储 */

short

i;

struct w16_bytes

byte;

struct w16_bits

bit;

};

void main(void)

{

union w16 w={0}; /* i为0; byte0和byte1也为0; b0~b15皆为0 */

void bit_print(short); /* 函数原型 */

w.bit.b9=1; /* 相当于byte1为2 */

w.bit.b10=1; /* 相当于byte1为6 */

w.byte.byte0='b';

printf("w.i=0x%x\n",w.i); /* 按整型解释、输出共享存储的内容 */

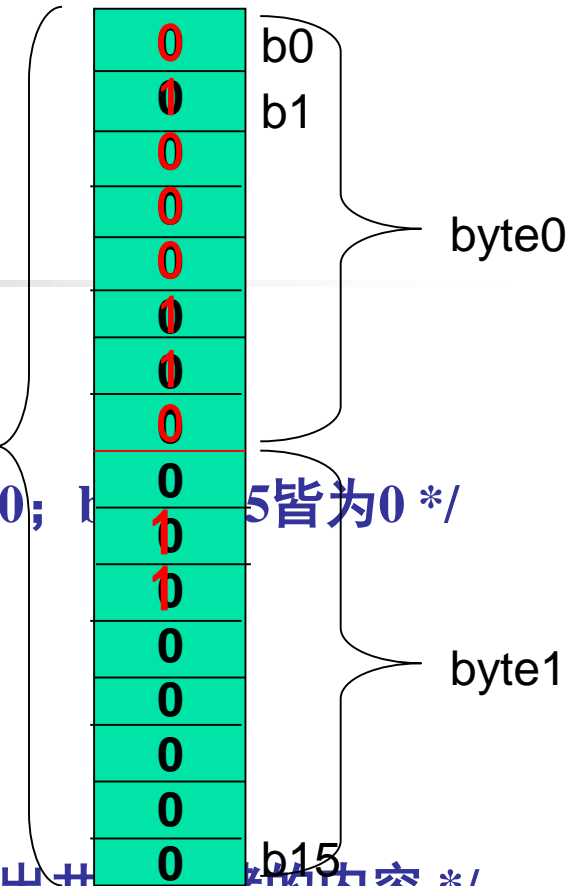
bit_print(w.i); /* 从高到低，逐位输出共享存储的各bit值 */

}

w.i

w.byte

w.bit



运行结果:

w.i=0x662

00000110 01100010

9.8 动态存储分配

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**



C的动态存储分配函数

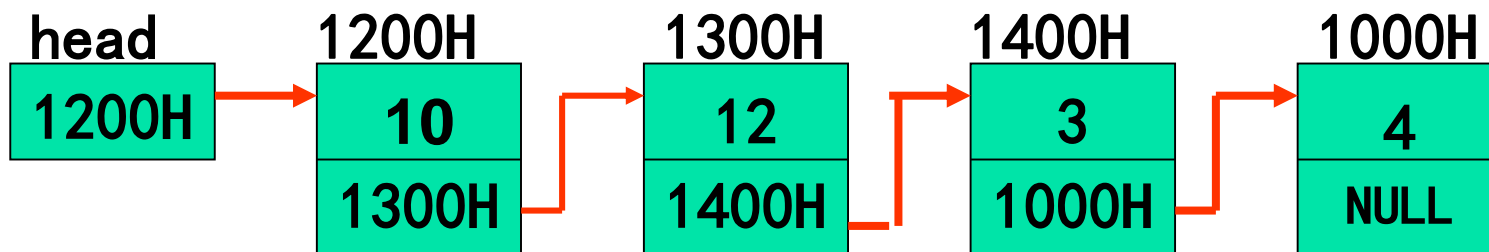
- 动态存储分配函数是C的标准函数，函数的原型声明在头文件**<stdlib.h>**中给出。
- 使用动态存储分配函数必须先使用**#include <stdlib.h>**编译预处理命令。
- C提供下列与动态存储分配相关的函数。

```
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);
```

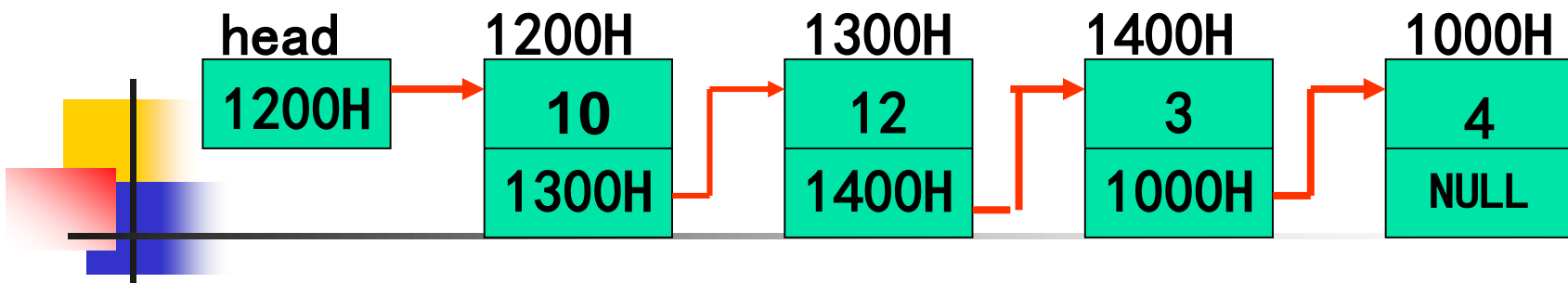
其中，**size_t**表示**unsigned int**，即无符号整型。它是在**<stdio.h>**中通过**typedef unsigned size_t;**定义的。

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

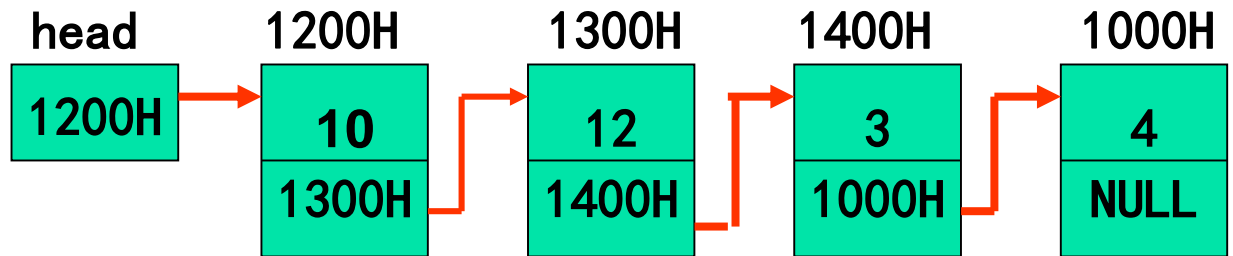
```
struct s_list{  
    int data;
```

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

```
};
```

头指针说明

```
struct s_list *head;
```



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**

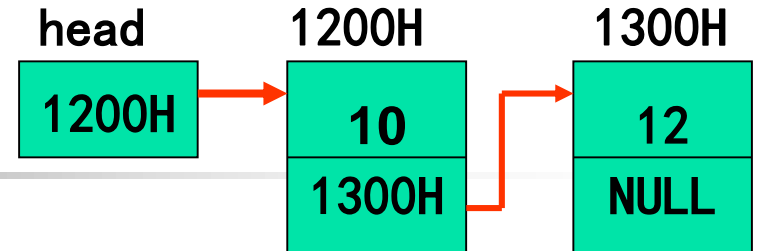
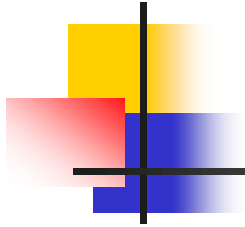
```
struct s_list node1={ 10,NULL};
```

```
/* node1为自引用结构变量 */
```

```
struct s_list *head;
```

```
/* head自引用结构类型的指针 */
```

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;
```



9.8.3 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 */**

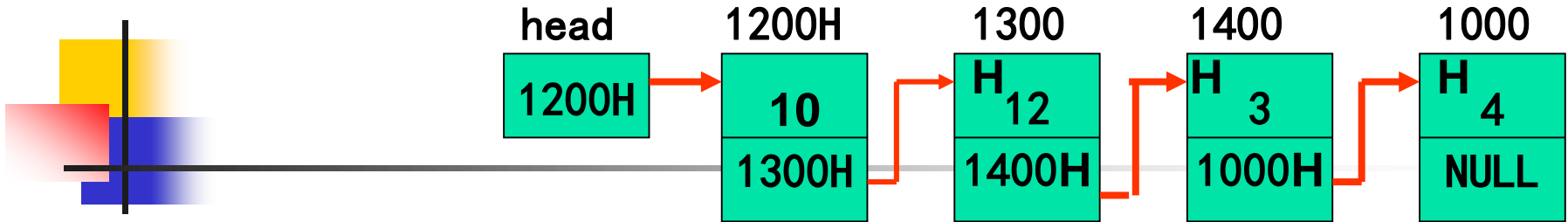
}

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，传递  
    头指针的地址&head给形参headp */  
    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。 $p = head;$
- (2) 如果链表非空（即遍历指针 p 非空， $p \neq NULL$ ）

(2-1) 输出结点数据域中成员的值。

$\text{printf}(\text{"\%d\\t"}, p \rightarrow \text{data});$

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

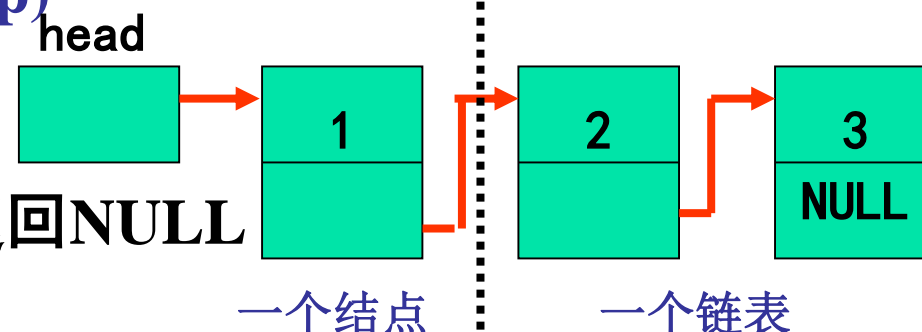
void print_list(struct s_list *head)

```
{ 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 */  
  }  
}
```



一个结点 一个链表



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

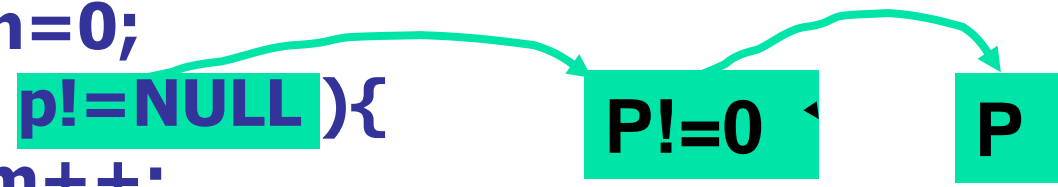
- 遍历链表可以进一步分为:

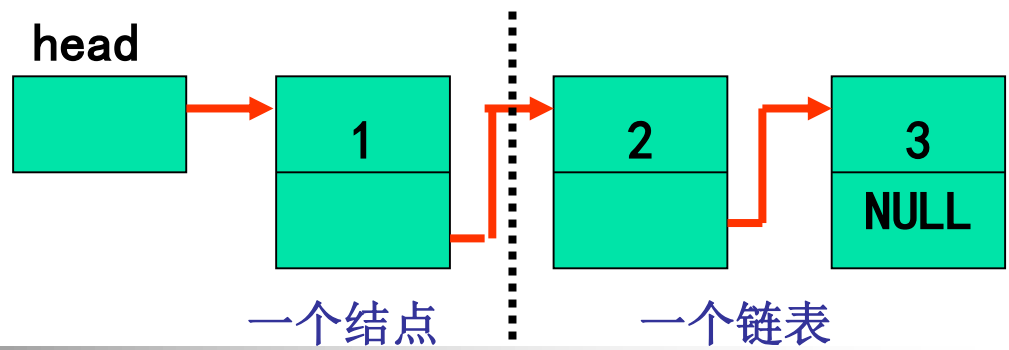
- ◆ 输出链表中各个结点的数据域成员; ✓
- ◆ 统计链表中结点的数目;
- ◆ 查找链表中符合条件的某个结点等。

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;  
}
```

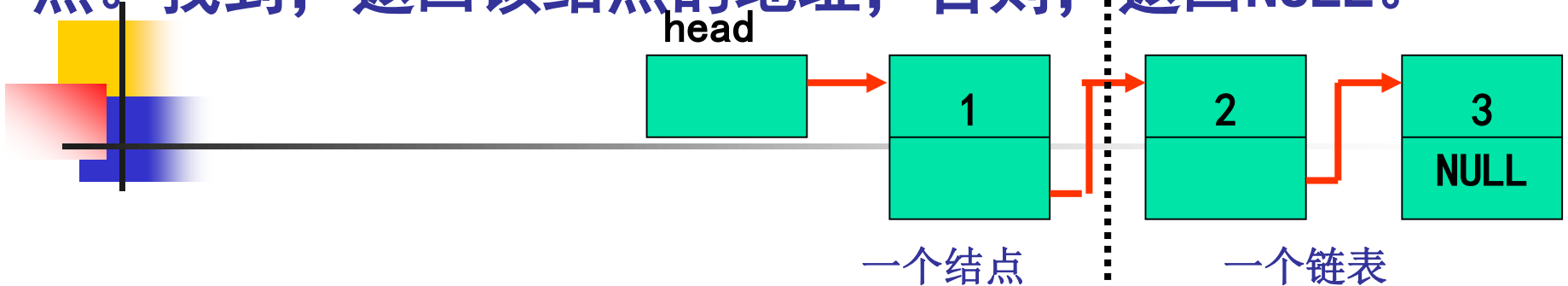




/*用递归法统计链表head中结点的数目 */

```
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;  
}
```

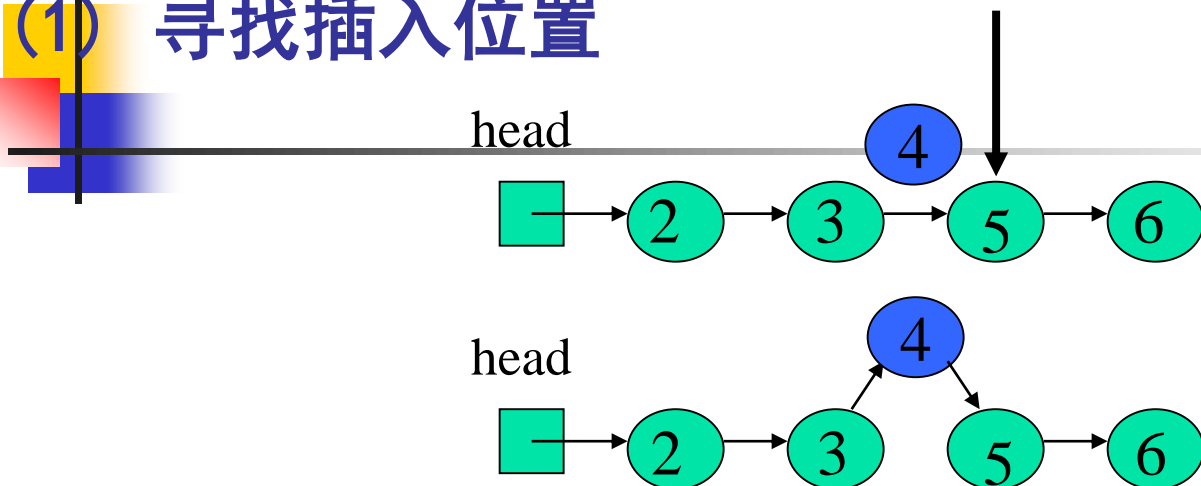
例 用递归法查找**链表**中数据成员值与形参**n**相同的结点。找到，返回该结点的地址，否则，返回NULL。



```
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。

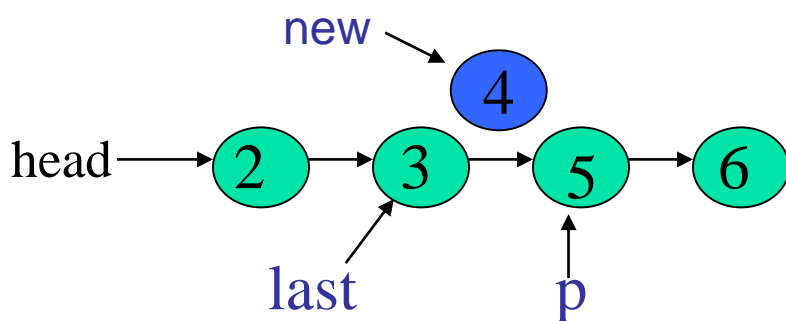
教材介绍将加入结点作为插入点的新后继结点（自学）

本节介绍将加入结点作为插入点的新前驱结点。

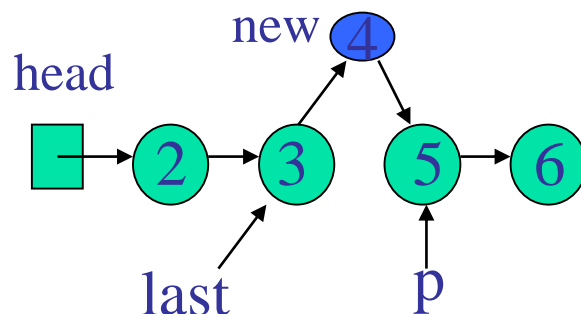
(2) 插入结点

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。



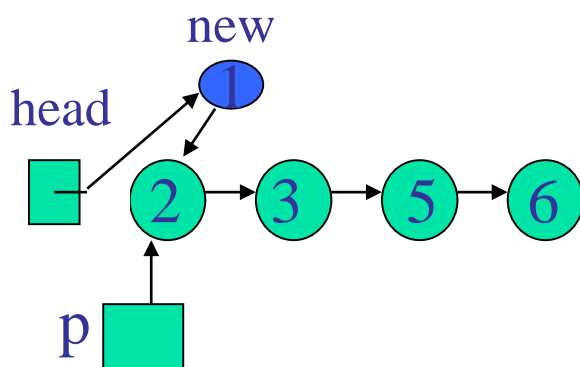
插入前结点间的指向关系



插入后结点间的指向关系

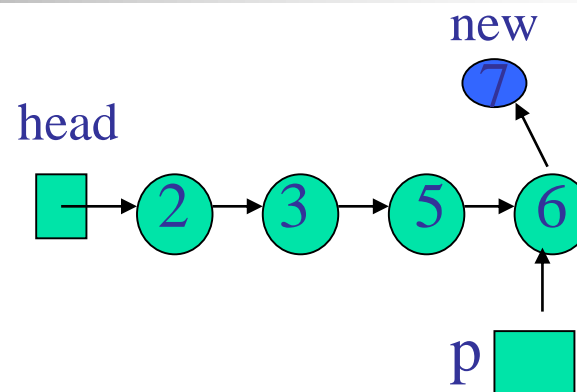
$last \rightarrow next = new;$
 $new \rightarrow next = p;$

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;

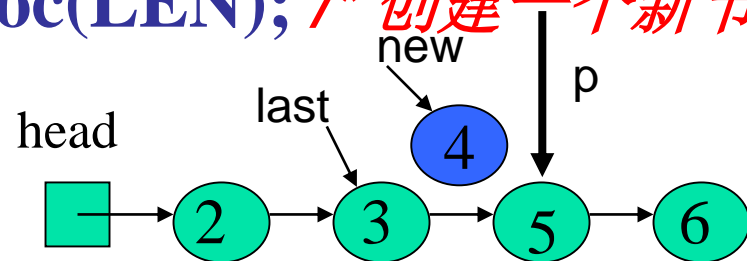
/ 寻找插入位置 */*

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

last=p;

p=p->next;

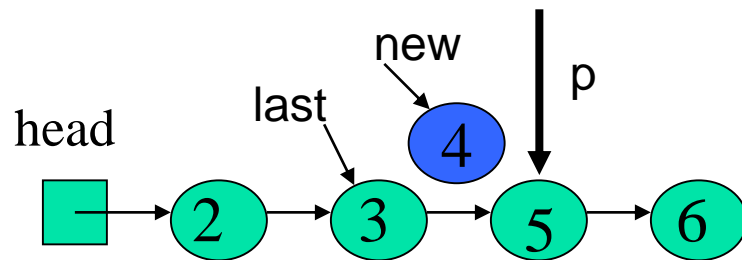
}



```

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;
}

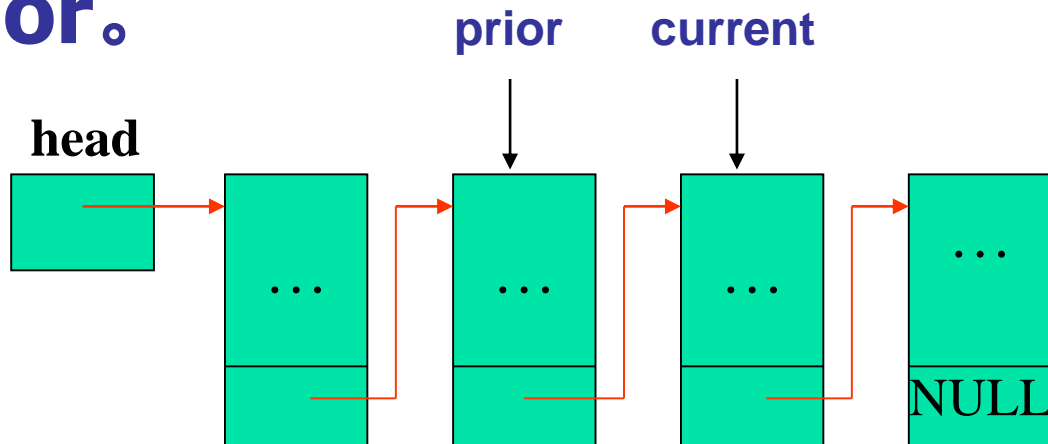
```



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**。

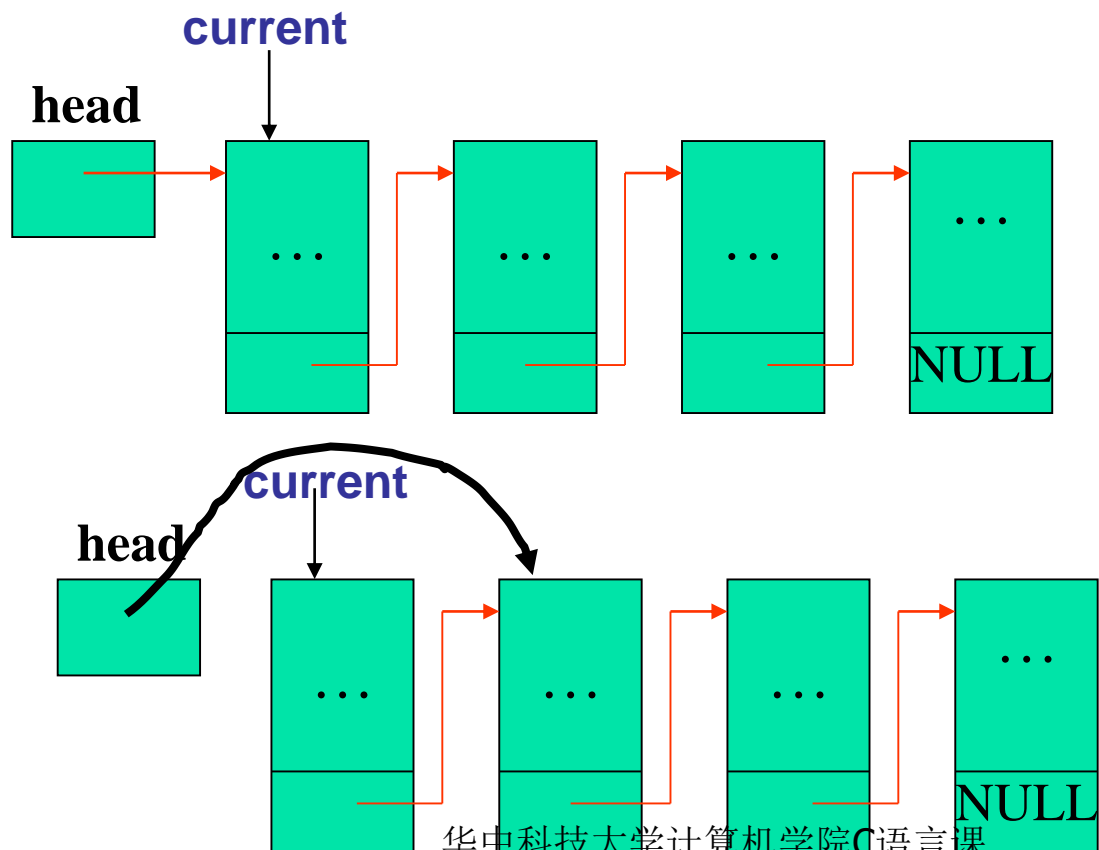


(2) 删除current所指结点

即改变连接关系

链头

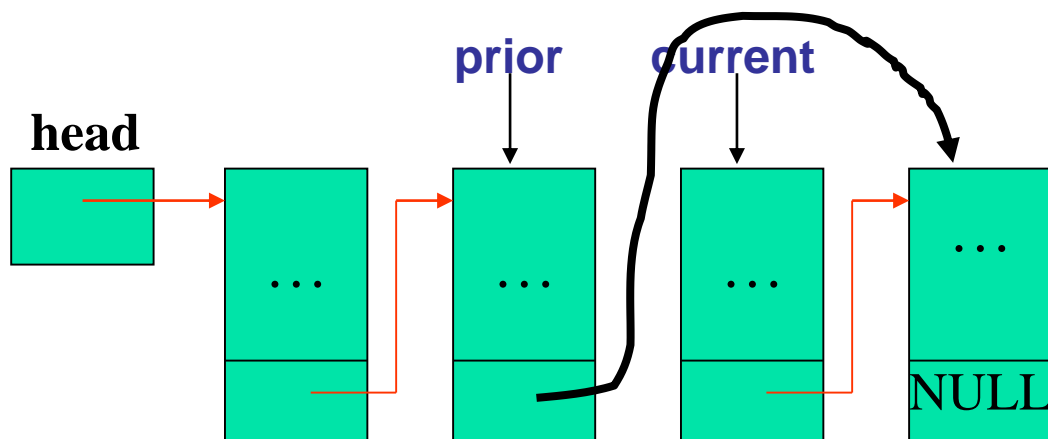
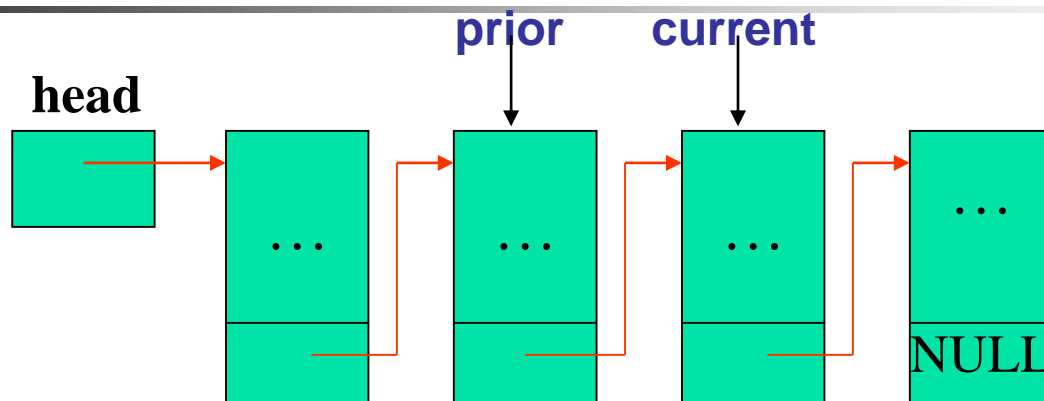
修改 *head*: *head* = *current* -> *next*



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



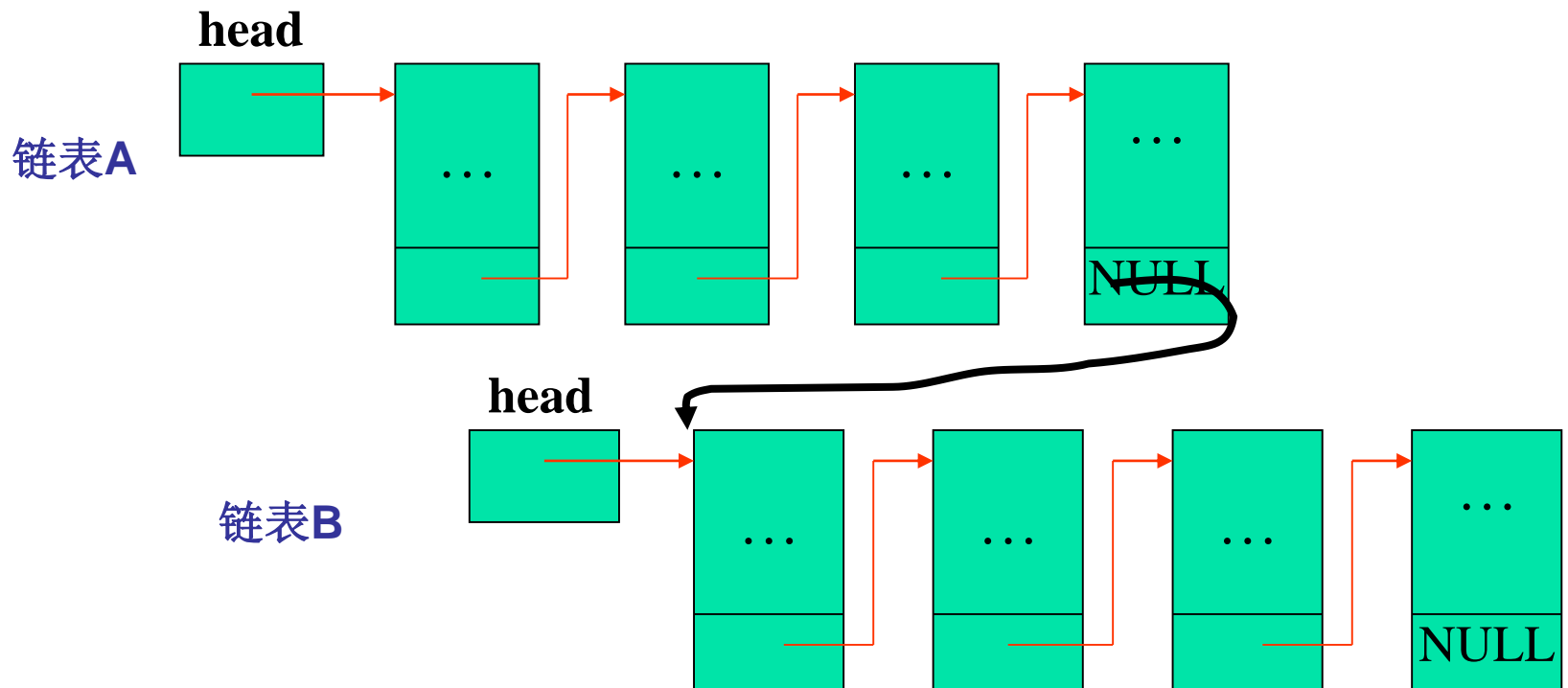
```
free(current);
```

/* 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

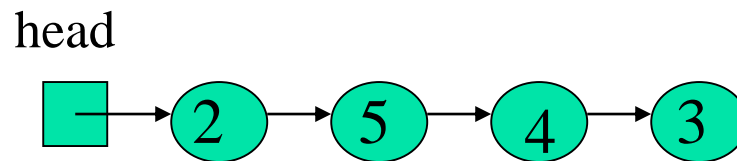
(1) 遍历链表A找到其链尾

(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;  
}
```

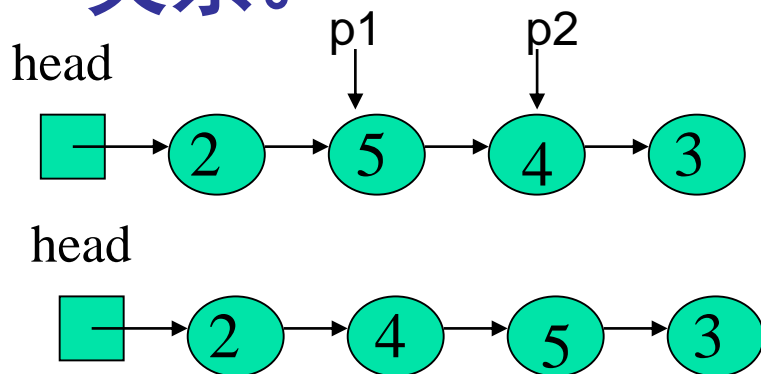
5. 链表排序

- 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。



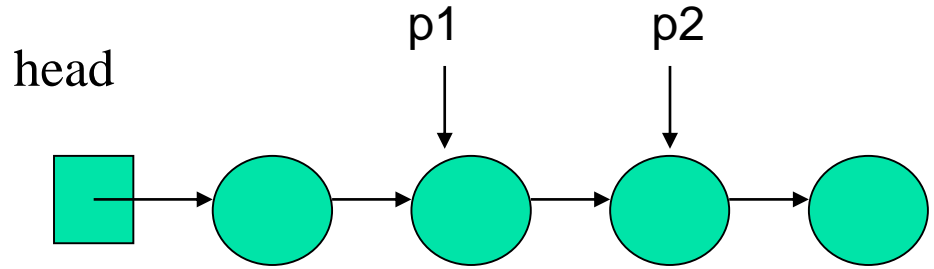
```
if(p1->data > p2->data){  
    int t;  
    t=p1->data; /* 交换数据域 */  
    p1->data=p2->data;  
    p2->data=t;  
}
```



```
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;
```

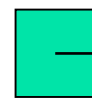
```
                p1->data=p2->data;
```

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

```
            }
```

```
}
```

head

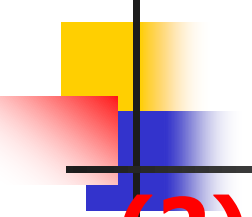


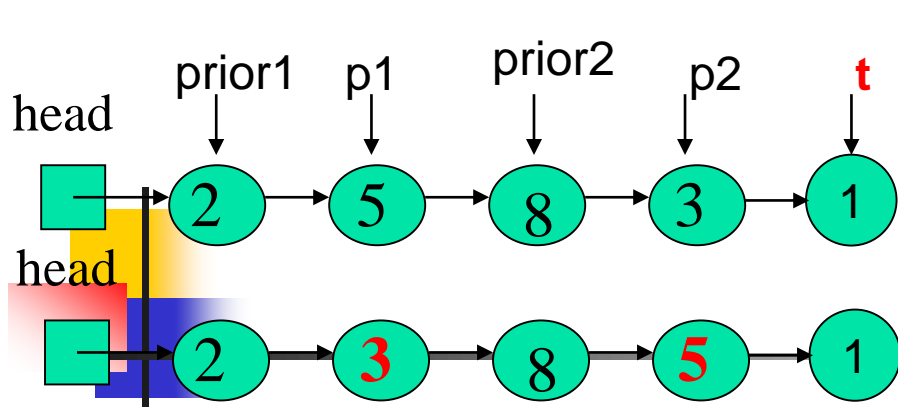
p1

p2

p2

/* 交换数据域 */

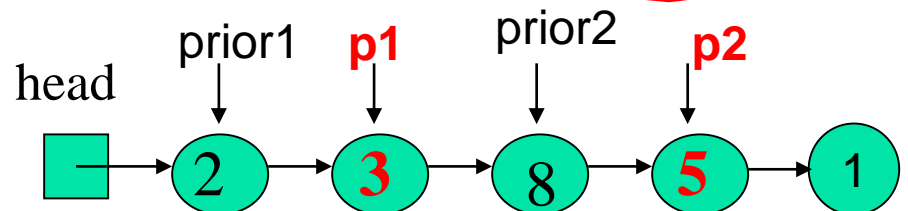
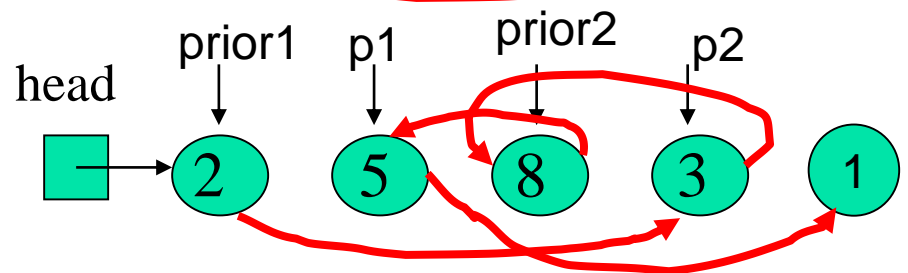
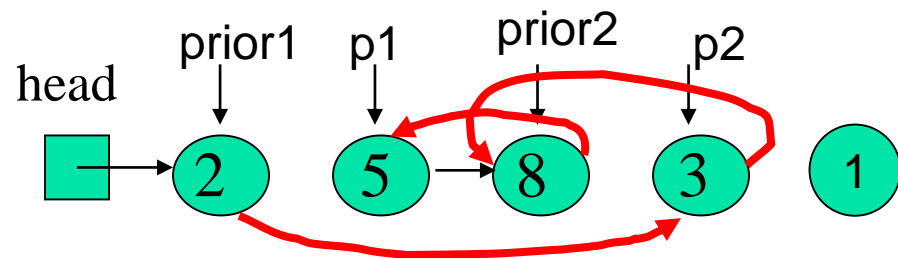
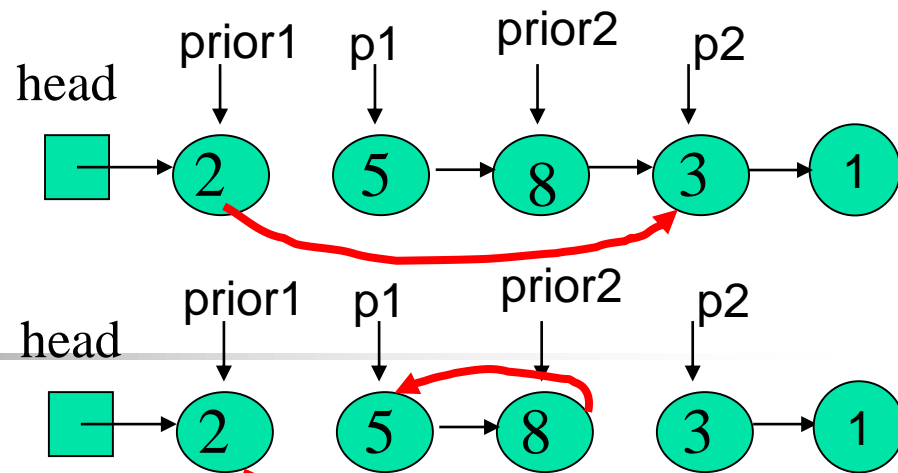
- 
- **(2) 改变结点的连接，操作较为复杂。**
 - 在数据域较为简单的情况下，多采用交换结点的数据域的方法进行链表排序。
 - 在数据域较为复杂，成员较多，采用改变链表结点之间的连接实现结点的交换的排序方法则效率较高。



```

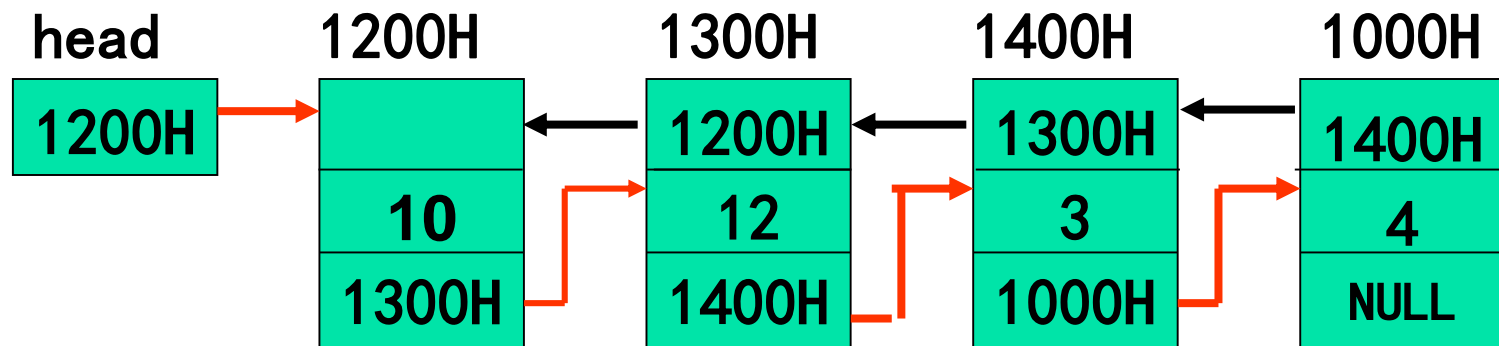
if(p1->data>p2->data){
    t=p2->next;
    prior1->next=p2;
    prior2->next=p1;
    p2->next=p1->next;
    p1->next=t;
    p2=p1;
    p1=prior1->next;
}

```



14.3.5 双向链表

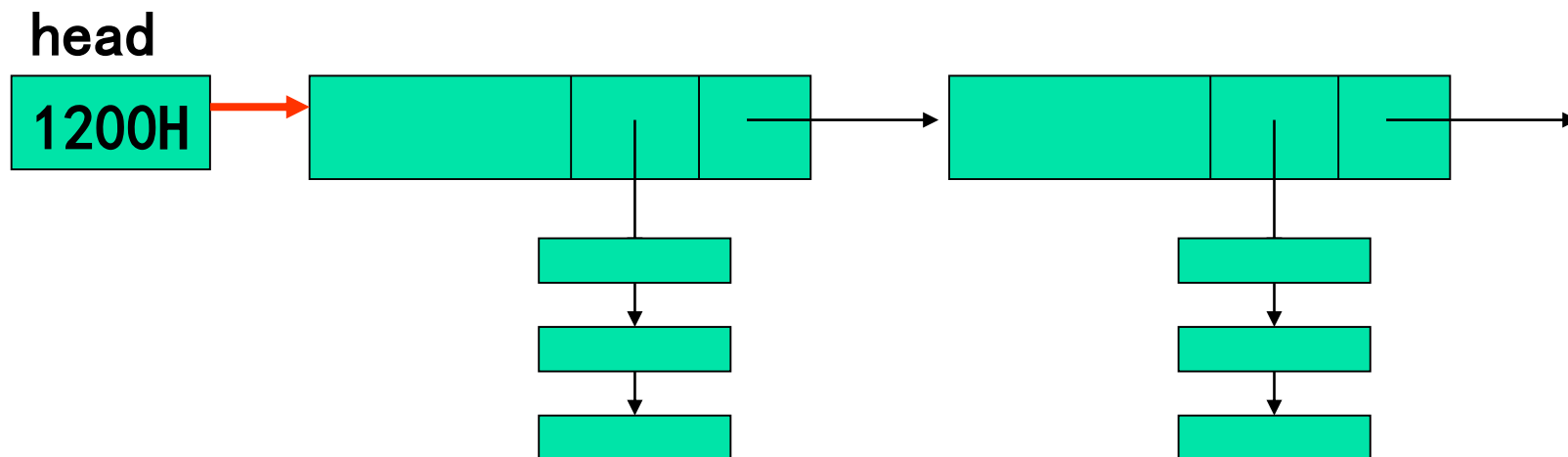
- 如果结点的指针域包含两个指针，且一个指向前一个结点，另一个指向后一个结点，这种链表称为**双向链表**。



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

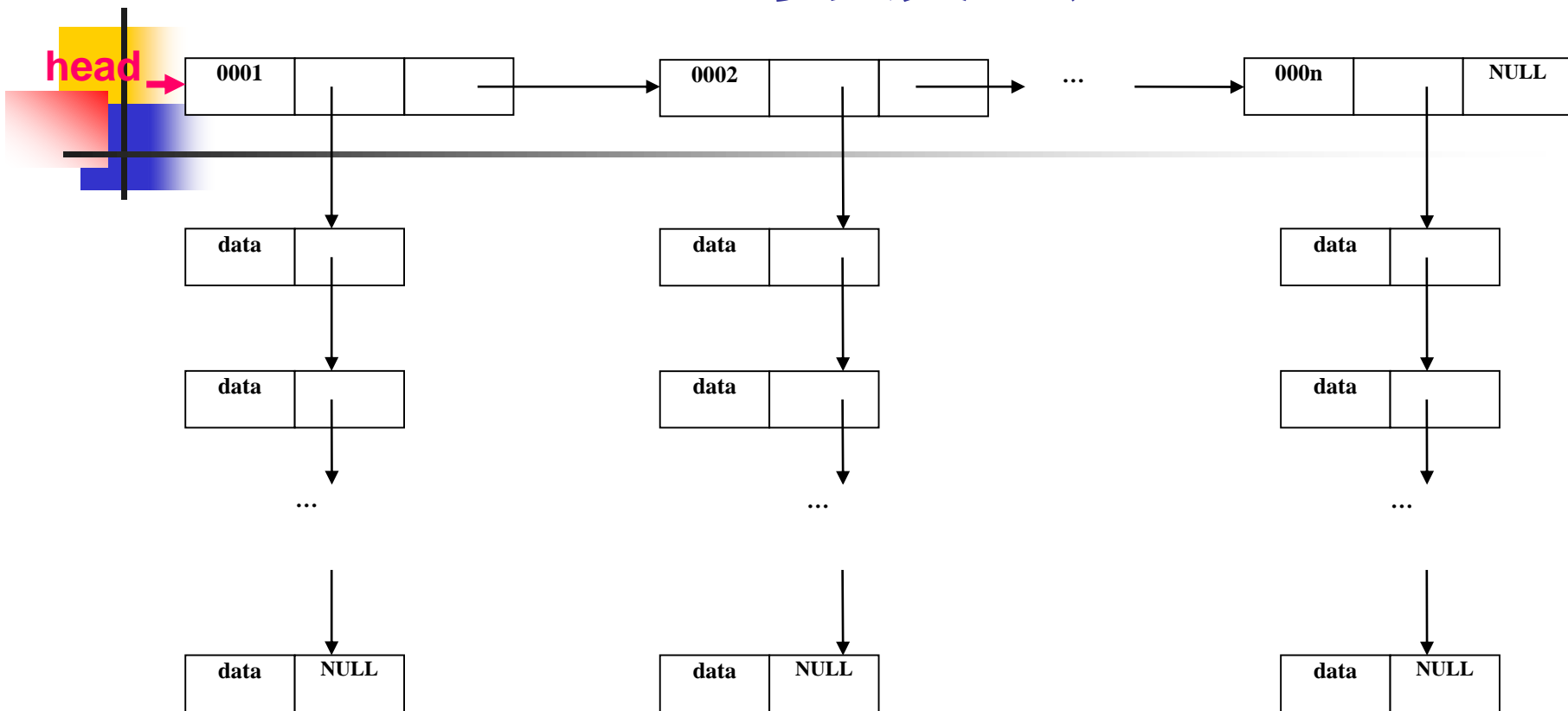
学生学习成绩表

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，创建完并通过(*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。