计算机程序设计基础(1)
--- C语言程序设计(11B)

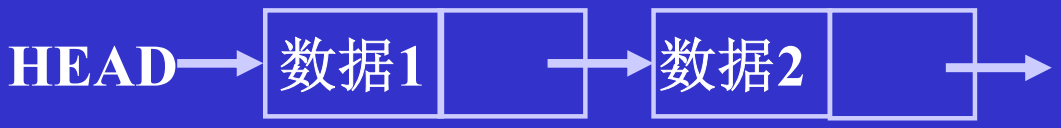
孙甲松
sunjiasong@tsinghua.edu.cn
电子工程系 信息认知与智能系统研究所
罗姆楼6-104
电话: 13901216180/62796193
2022.12.

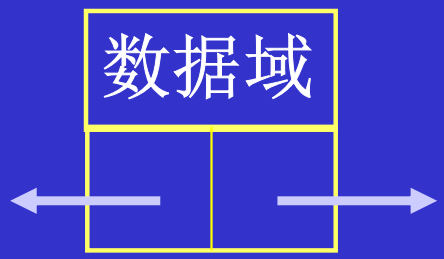
第11章　结构体与联合体 第2部分

目录

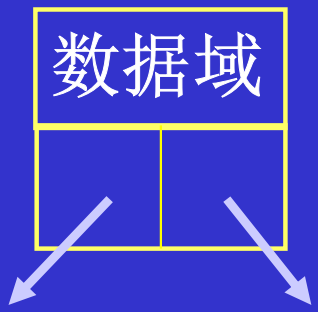
- 如果在结构体中包含与结构体同类型（或不同类型的）的一个指针或多个指针或指针数组，就可以构成链表、树、图等各种复杂的数据结构。

链表，单链表，环形链表



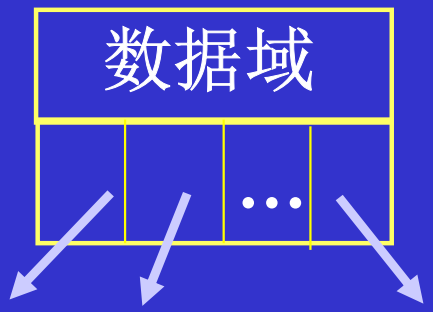


双向链表



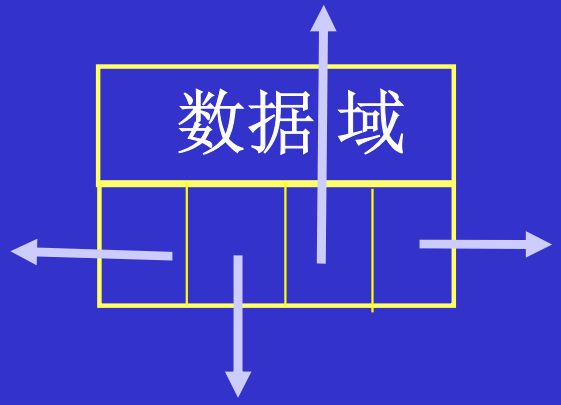
二叉树

二叉平衡树，
二叉索引树



多叉树

B树，B+树



十字链表，图

11.4 链表

11.4.1 链表的基本概念

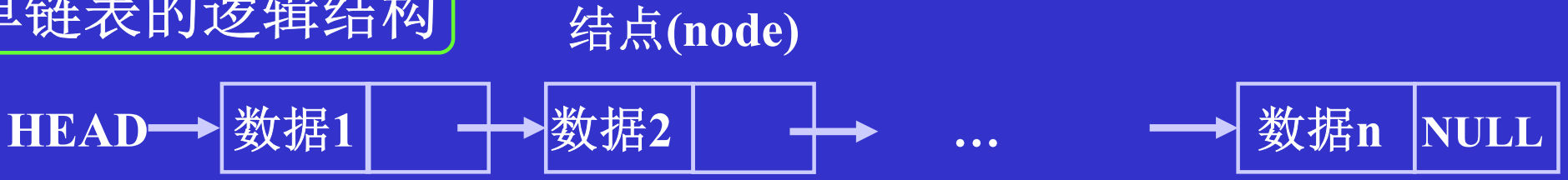
提醒：要理解掌握空指针NULL的概念

1. 链表的一般结构

存储节点

数据域	存放数据元素
指针域	存放下一个结点元素的地址

单链表的逻辑结构



- HEAD被称为头指针，当HEAD=NULL(或0)时称为空表

- 在链表中，各数据结点的存储序号是不连续的，并且各结点在存储空间中的位置关系与逻辑关系也不一致，各数据元素之间的前后件关系是由各结点的指针域来指示的。链表是线性表，但不同于同样是线性表的数组，数组是连续存放的。

2. 结点结构体类型的定义

链表结点结构
的一般形式：

```
struct  结构体名
{ 数据成员表;
    struct  结构体名  *指针变量名;
};
```

3. 结点的动态内存申请

形式是：

```
struct  结构体名  *p;
p=(struct  结构体名  *)malloc(结构体字节数);
```

p指向malloc函数所返回动态内存块的首地址

（当然也可以使用calloc函数，大同小异）

4. 结点的动态内存释放

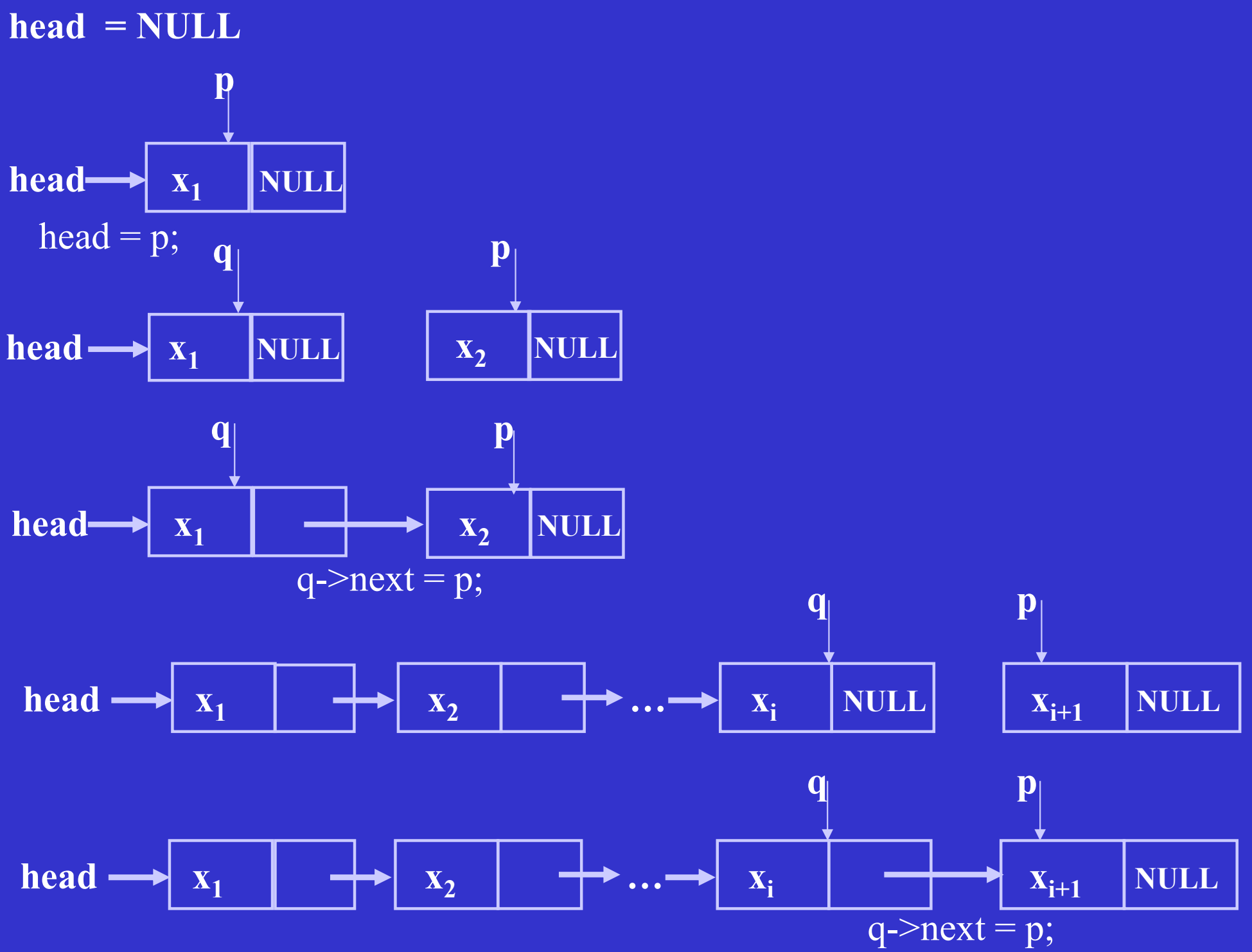
释放结点内存块用如下函数： **free(p);**

它表示释放由p所指的动态存储内存块。

【例11-10】读入一个正整数序列建立链表，以非正整数结束。

```
#include   <stdio.h>
#include   <stdlib.h>
struct  node   /*定义结点类型*/
{  int  data;
   struct  node  *next;
};
main( )
{  int  x;
   struct  node  *head, *p, *q;
   head = NULL;     /*置链表头指针为空*/
   q = NULL;
   scanf("%d", &x);  /*读入一个正整数*/
```

```
while (x＞0)   /*若输入值大于0 */
{ p=(struct node *)malloc(sizeof(struct node));
        /* 申请一个结点动态内存 */
  if (p == NULL)
  { printf("can't get memory!\n");
    exit(1);
  }
  p->data = x;   /* 置当前结点的数据域为输入的正整数x */
  p->next = NULL;       /* 置当前结点的指针域为空 */
  if (head == NULL)
        head = p;  /* 若链表为空,则将头指针指向当前结点p */
  else  q->next = p;  /* 将当前结点链接在链表的最后 */
  q=p;                  /* 递推，置当前结点q为链表最后一个结点 */
  scanf("%d", &x);   /*再读入一个正整数，为下一次循环做好准备 */
}
```


head = NULL
p
head
x1
NULL
head = p;
q
p
head
x1
NULL
x2
NULL
q
p
head
x1
x2
NULL
q->next = p;
q
p
head
x1
x2
…
xi
NULL
xi+1
NULL
q
p
head
x1
x2
…
xi
xi+1
NULL
q->next = p;

```
    p=head;
    /*从链表的第一个结点开始,打印各结点的元素值,并删除*/
    while(p != NULL)
    {   printf("%d ", p->data);  /* 打印当前结点中的数据 */
        q=p;
        p=p->next;     /* p指向下一个结点 */
        free(q); /* 释放当前结点q 的动态内存，也就删除了q 结点*/
    }
    printf("\n");
}
```

运行结果：

5 4 3 1 2 -1

5 4 3 1 2

11.4.2 链表的基本运算

1. 在链表中查找指定元素

在对链表进行插入或删除的运算中，首先需要找到插入或删除的位置，这就需要对链表进行扫描查找，在链表中寻找包含指定元素值的前一个结点。当找到包含指定元素的前一个结点后，就可以在该结点后插入新结点或删除该结点后的一个结点。提示：链表的插入删除不需要移动数据。

在非空链表中寻找包含指定元素值的前一个结点的C语言描述。

```
struct  node          /*定义结点类型*/
{    ET  data;      /*ET为数据元素类型名,下同*/
     struct  node  *next;
};
```

```
/* 在头指针为head的非空链表中寻找包含元素x的前一个结点p，结点p的值作为函数值返回 */
   struct  node  *lookst(struct node *head, ET x)
   {  struct  node  *p;
      p=head;
      while( (p->next != NULL) && ( p->next->data != x) )
          p=p->next;   /*注意:上面条件表达式中的项不能写颠倒 */
      return p;   /* 如果找不到元素x，则返回指向链尾结点的指针 */
   }
```

2. 链表的插入

在头指针为head的链表中包含元素x的结点之前插入新元素b

插入过程

- 用malloc( )函数申请取得新结点p, 并让该结点的数据域为 b，即令 p->data=b；p->next=NULL;
- 在链表中寻找包含元素x的前一个结点，设该结点的存储地址为q;
- 最后将结点p插入到结点q之后。为了实现这一步，只要改变以下两个结点的指针域内容:

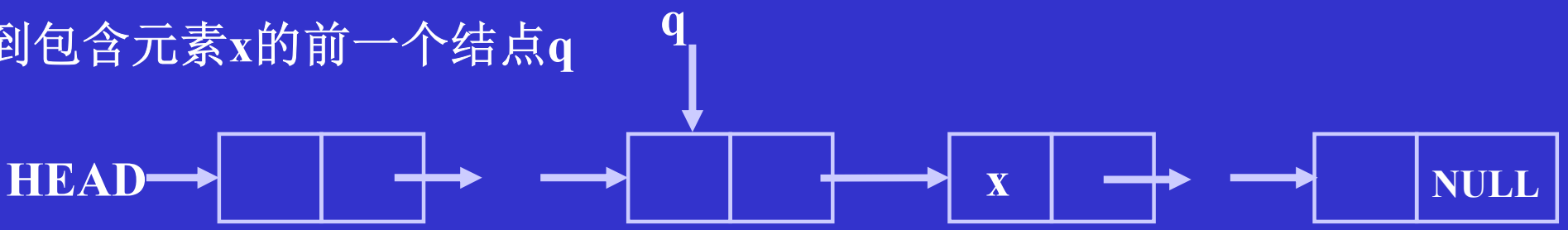
① 使结点p指向包含元素x的结点（即结点q的后件结点）, 即令 p->next = q->next;

② 使结点q的指针域内容改为指向结点p，即令

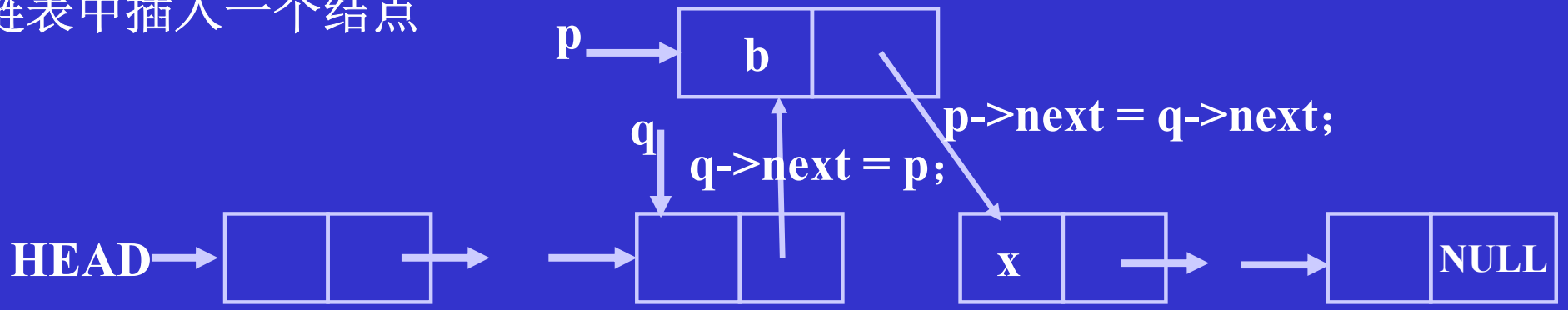
q->next = p;

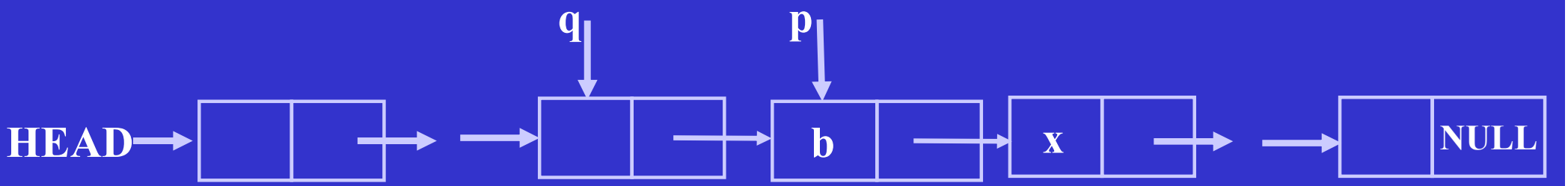
申请取得新结点p

找到包含元素x的前一个结点q



链表中插入一个结点





```
#include   <stdio.h>
#include   <stdlib.h>
struct  node          /*定义结点类型*/
{  ET  data;          /*数据元素类型*/
   struct  node  *next;
};
```

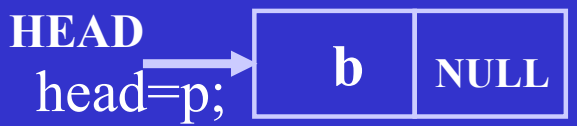
```
/* 在头指针为head的链表中包含元素x的结点之前插入新元素b */
/* 注意：因为函数inslst中要修改head的值，因此传来的是指针
head的内存地址，所以出现了形参 struct node **head，后面的删
除函数也是如此 */
```

```
void inslst(struct node  **head, ET x, ET b)
{   struct  node   *p, *q;
    p=(struct node *)malloc(sizeof(struct node)); /*申请一个新结点p */
    if (p == NULL)
    { printf("can't get memory!\n");
      exit(1);
    }
    p->data=b;          /*设置结点的数据域为b */
    p->next=NULL; /*设置结点的指针域为NULL */
    if (*head==NULL) /* 链表为空 */
    {  *head=p;
       return;
    }
    if ((*head)->data==x)    /*在第一个结点前插入*/
    {  p->next=*head;
       *head=p;
       return;
    }
    q=lookst(*head, x) ; /*寻找包含元素x的前一个结点q*/
    p->next=q->next ;
    q->next=p;  /*结点p插入到结点q之后*/
    return; /* 如果找不到元素x，则将结点p插入到链尾，此时 q->next为NULL */
}
```

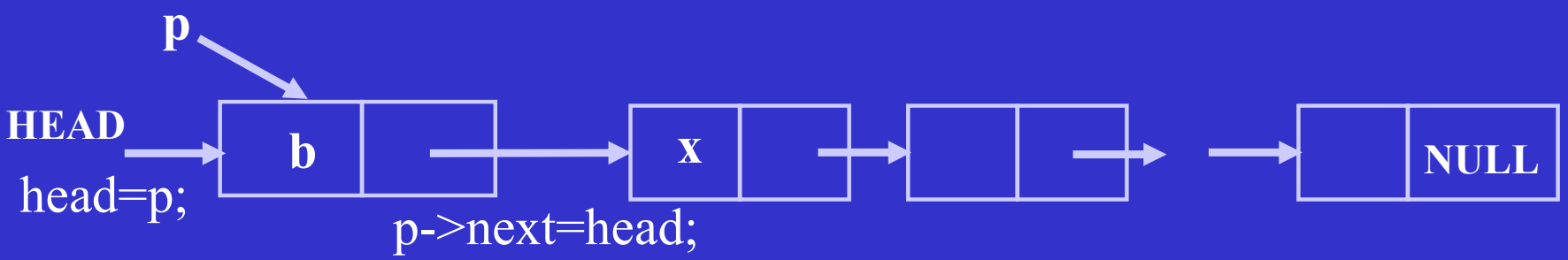
申请取得新结点p

向链表中插入一个结点的几种特殊情况：

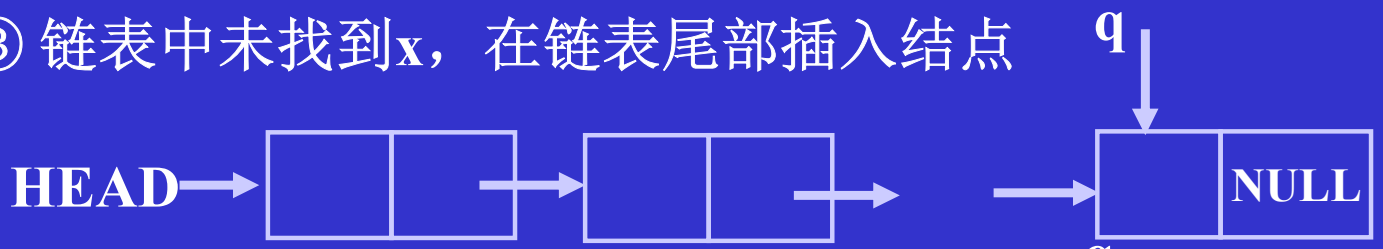
① 链表为空 HEAD=NULL

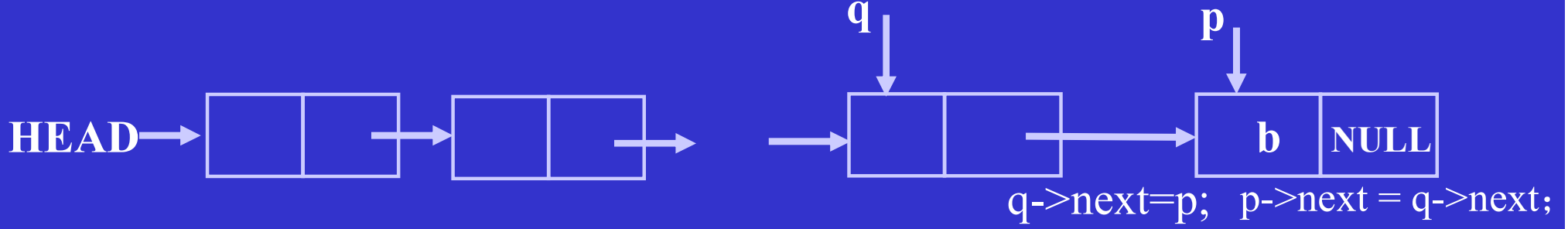


② 链表的第一个结点的值为x，在第一个结点前插入



③ 链表中未找到x，在链表尾部插入结点

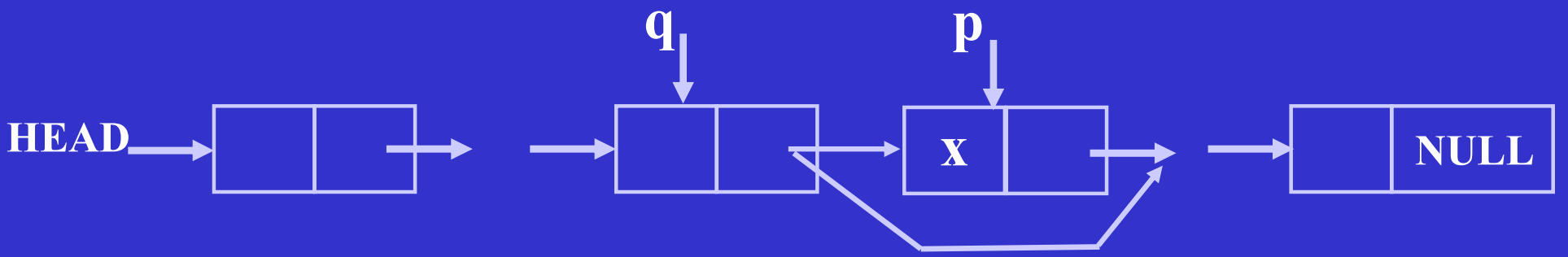




3. 链表的删除

删除过程

- 在链表中寻找包含元素x的前一个结点，设该结点地址为q。则包含元素x的结点地址 p=q->next;
- 将结点q后的结点p从链表中删除，即让结点q的指针指向包含元素x的结点p的指针指向的结点，即令q->next=p->next;
- 将包含元素x的结点p释放。



C语言描述：

```
#include   <stdio.h>
#include   <stdlib.h>
struct  node          /*定义结点类型*/
{  ET  data;          /*数据元素类型*/
   struct  node  *next;
};
```

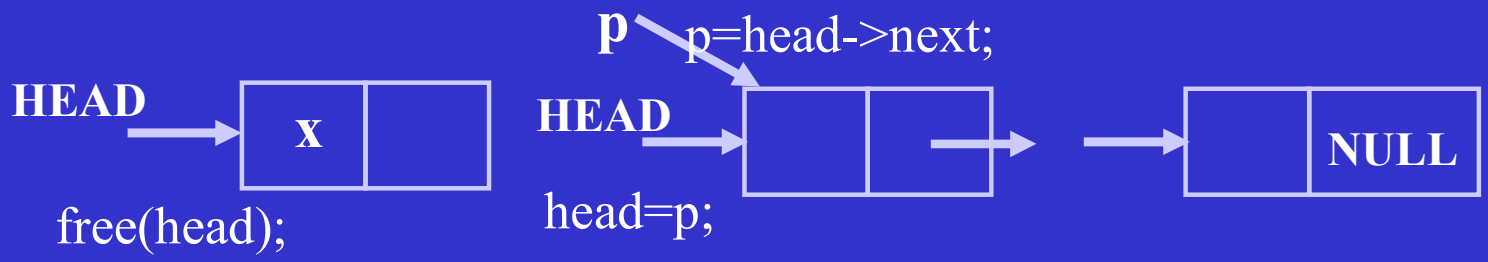
```
/*在头指针为head的链表中删除包含元素x的结点*/
void delst(struct node  **head,  ET  x)
{   struct  node   *p, *q;
    if (*head==NULL) /* 链表为空 */
    {   printf("This is a empty list!\n");  return; }
    if ((*head)->data==x)   /* 删除第一个结点 */
    {  p=(*head)->next;
       free(*head) ;    /* 释放表头结点 */
       *head=p;
       return;
    }
    q=lookst(*head, x) ; /*寻找包含元素x的前一个结点q */
    if (q->next==NULL) /*链表中没有包含元素x的结点 */
    {  printf("No this node in the list!\n");  return; }
    p=q->next ;   q->next=p->next;  /* 删除结点p */
    free(p) ;   /* 释放结点p */
    return;
}
```

从链表中删除一个结点的几种特殊情况：

① 链表为空 **HEAD=NULL**

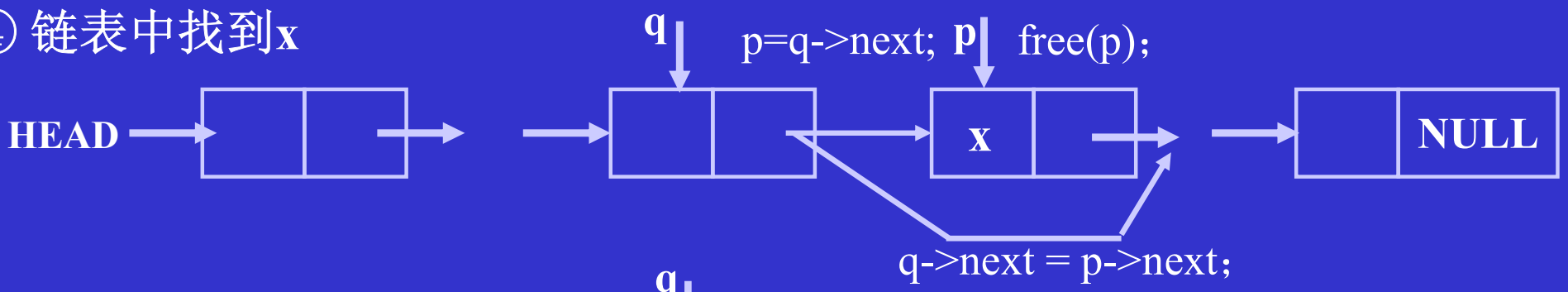
打印链表空信息，函数返回终止执行

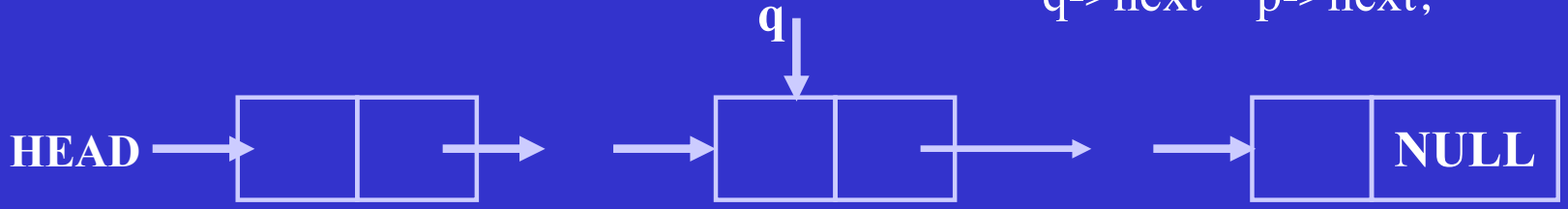
② 链表的第一个结点的值为x，删除第一个结点



③ 链表中未找到x，打印出错信息，函数返回终止执行

④ 链表中找到x





4. 链表的打印

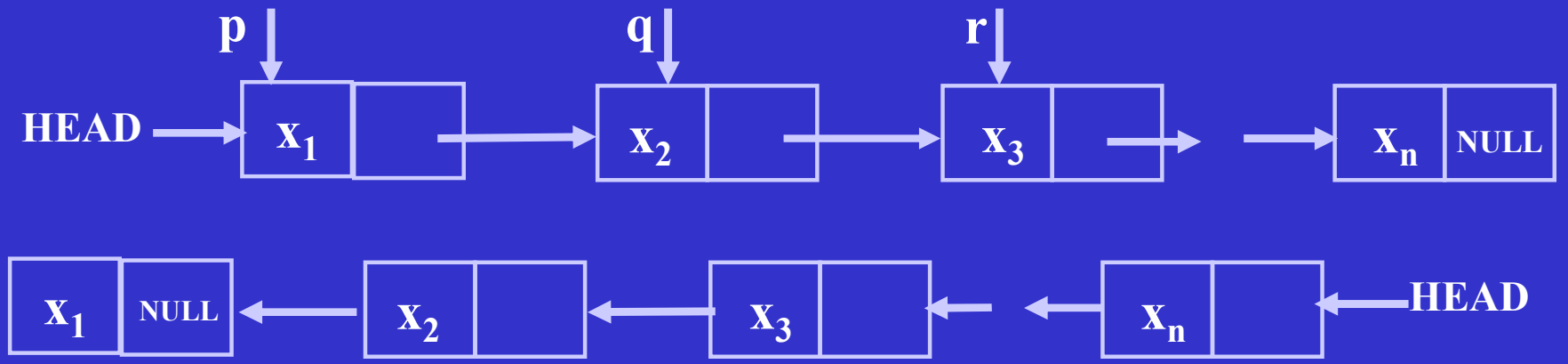
```
void printlst(struct node  *head)
{      struct node  *p=head;
       /*从链表的第一个结点开始,打印各结点的元素值 */
       while(p != NULL)
       {   printf("%d  ", p->data);  /* 打印当前结点中的数据 */
           p=p->next;
       }
       printf("\n");
}
```

简化版：

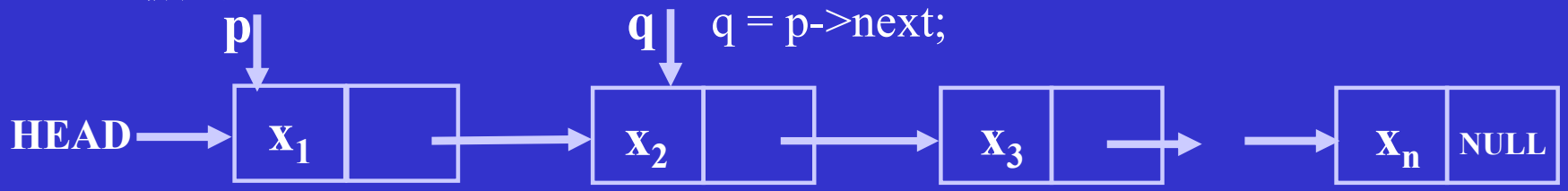
```
void printlst(struct node  *head)
{      while(head != NULL)
       {   printf("%d  ", head ->data);  /* 打印当前结点中的数据 */
           head = head ->next;
       }
       printf("\n");
}
```

5. 链表的逆转

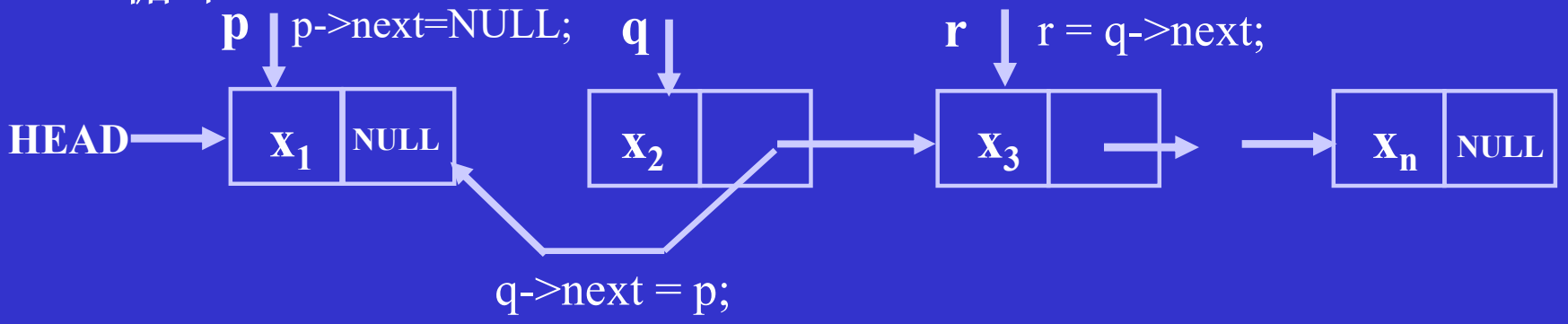
```
void reverselst(struct node  **head)
{   struct node  *p,*q, *r;
    p=*head;
    if (p == NULL) return;
    q=p->next;
    p->next = NULL;    /* 设置链表尾结点指针为NULL */
    while (q != NULL)
    {   r = q->next;   /* r指向q的下一个结点 */
        q->next = p;   /* q的指针域指向前一个结点p */
        p = q;  q = r;   /* 三个指针依此递推,为下一次循环做好准备*/
    }
    *head = p;
    return;
}
```



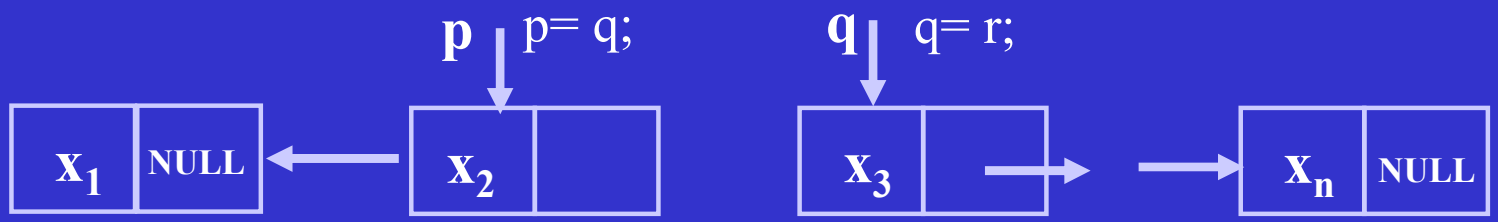
进入while循环之前:



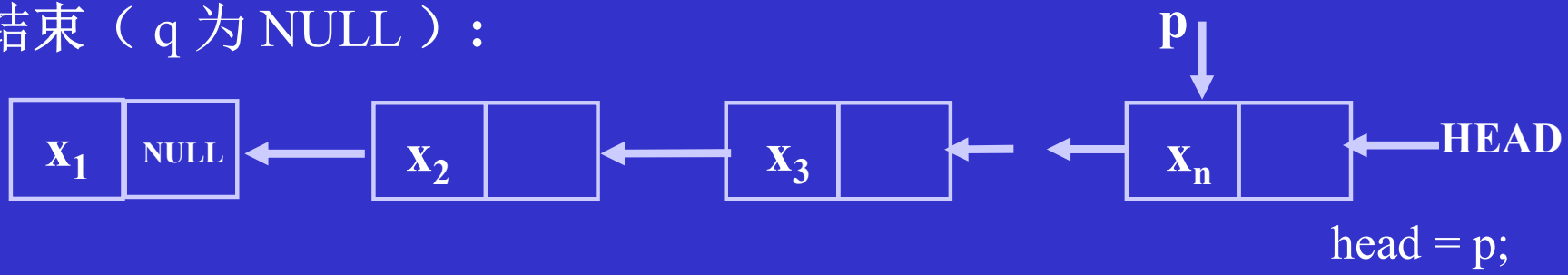
进入while循环:



while循环执行一次结束(指针依此向前递推）:



逆转结束（ q 为 NULL ）:



补充实例：读入一个正整数序列，以非正整数结束。按照其反序建立链表，打印出链表，并逐个删除链表中的元素。

```
#include   <stdio.h>
#include   <stdlib.h>
struct  node   /*定义结点类型*/
{  int  data;
   struct  node  *next;
};
struct  node  *lookst(struct node *head, int x) /* 在链表中查找 */
{  struct  node  *p;
   p=head;
   while( (p->next != NULL) && ( p->next->data != x) )
        p=p->next;
   return(p);
}
void inslst(struct node  **head, int x, int b) /* 在链表中插入 */
{   struct  node   *p, *q;
    p=(struct node *)malloc(sizeof(struct node)); /*申请1个新结点*/
    if (p == NULL)
    { printf("can't get memory!\n");  exit(1); }
    p->data=b;     /*设置结点的数据域*/
    if (*head==NULL) /*链表为空*/
    { *head=p;      p->next=NULL;      return; }
```

```
    if ((*head)->data==x)    /*在第一个结点前插入*/
    {   p->next=*head;     *head=p;       return; }
    q=lookst(*head, x) ; /*寻找包含元素x的前一个结点q*/
    p->next=q->next ; q->next=p;  /*结点p插入到结点q之后*/
    return;
}
void delst(struct node  **head,  int  x) /* 在链表中删除 */
{   struct  node   *p, *q;
    if (*head==NULL) /*链表为空*/
    { printf("This is a empty list!\n"); return; }
    if ((*head)->data==x) /*删除第一个结点*/
    { p=(*head)->next;
      free(*head) ;    *head=p;     return;
    }
    q=lookst(*head, x) ;  /*寻找包含元素x的前一个结点q */
    if (q->next==NULL)  /*链表中没有包含元素x的结点 */
    { printf("No this node in the list!\n"); return; }
    p=q->next ;   q->next=p->next;  /*删除结点p */
    free(p) ;   /*释放结点p*/
    return;
}
```

```
void printlst(struct node  *head) /* 打印链表中元素 */
{    while(head != NULL)
     {    printf("%d  ", head ->data);  /*打印当前结点中的数据*/
          head = head ->next;
     }
     printf("\n");
}
main( )
{  int  x, y=0;
   struct  node  *head;
   head=NULL;           /* 置链表头指针为空 */
   scanf("%d", &x);     /* 输入一个正整数 */
   while (x > 0)
   {     inslst(&head, y, x);  /* 在前一个数y之前插入x */
        /*注意:插入时会修改head指针值,因此必须传head的地址 */
         y = x;
        scanf("%d", &x);
   }
   printlst(head);
```

```
    while (head != NULL)
    {       delst(&head, head->data);
     /*注意:删除时也会修改head指针值，因此必须传head的地址 */
            printlst(head);
    }
}
```

运行结果：

1 2 3 4 5 -1

5 4 3 2 1

4 3 2 1

3 2 1

2 1

1

请按任意键继续…

再次强调：调用函数时，如果仅仅传递的是指针，那么在函数中只能修改指针所指单元的内容。如果要在函数中修改指针的值，则必须给函数传递指针单元的地址！

修改一下主程序，每删除一个节点后，逆转一次链表：

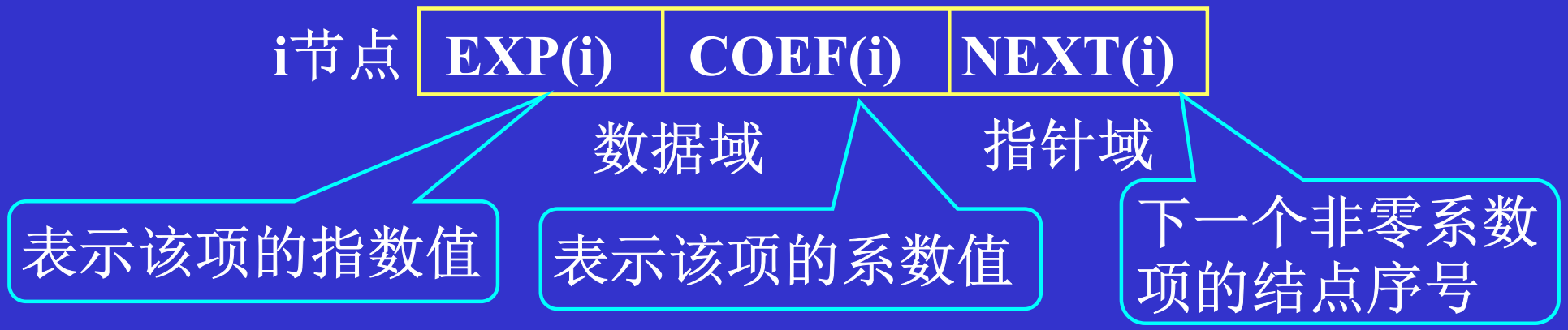
```
main( )
{  int  x, y=0;
   struct  node  *head=NULL;
   scanf("%d", &x);
   while (x > 0)
   {  inslst(&head, y, x);
      y = x;
      scanf("%d", &x);
   }
   printlst(head);
   while (head != NULL)
   {  delst(&head, head->data);
      reverselst(&head);
      /* 删除一个节点后，逆转一次链表 */
      printlst(head);
   }
}
```

运行结果：

```
1  2  3  4  5 -1
5  4  3  2  1
1  2  3  4
4  3  2
2  3
3
请按任意键继续...
```

11.4.3 多项式的表示与运算

设多项式为： $P_n(x)=a_nx^n+a_{n-1}x^{n-1}+\ldots+a_1x+a_0$



C语言描述：

```
#include   <stdio.h>
#include   <stdlib.h>
struct  node                    /* 定义结点类型 */
{     int  exp;                 /* 指数为正整数 */
      double  coef;             /* 系数为双精度型 */
      struct  node  *next;      /* 指针域 */
};
```

题外话，此种描述并不一定最好，因为在**32**位编译器上：

sizeof(struct node) 的结果是**24**，每个**node**节点占**24**个字节的内存。

若改为：

```
struct node                    /* 定义结点类型 */
{    double coef;              /* 系数为双精度型 */
     int exp;                  /* 指数为正整数 */
     struct node *next;        /* 指针域 */
}
```

因为在**32**位编译器上，exp和next都占4个字节，恰好为8字节。

sizeof(struct node) 的结果将是**16**。

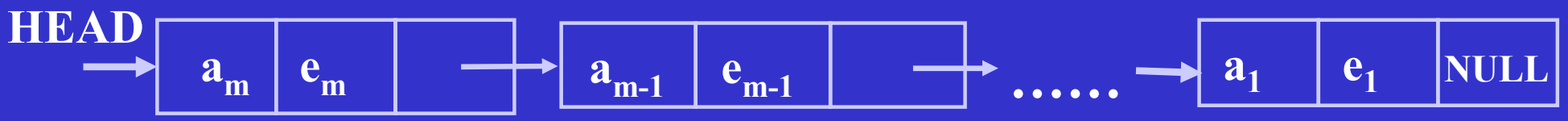
而且不需要用#pragma pack(4) 强制对齐。

所以大家以后在定义**struct**时，要多留意一下各个结构体成员的排列顺序！

非零系数项的多项式为：

$$P_m(x) = a_m x^{e_m} + a_{m-1} x^{e_{m-1}} + \cdots + a_1 x^{e_1}$$

其中 $a_k \neq 0(k=1,2,\ldots,m)$, $e_m > e_{m-1} > \ldots > e_1 \geq 0$



1. 多项式链表的生成

过程：

按降幂顺序以数对形式依次输入多项式中非零系数项的指数e_k和系数a_k (k=m,m-1,…,1)，最后以输入指数值-1为结束。对于每一次的输入，申请一个新结点，填入输入的指数值与系数值后，将该结点链接到链表的末尾。

其算法的C语言描述为：

```
struct node *inpoly( )  /*多项式链表的生成 */
{       struct node *head = NULL, *p, *k=NULL;
        int e;    double a;
        printf("input exp and coef:");
        scanf("%d%lf", &e, &a);
        while (e >= 0)
        {       p = (struct node *)malloc(sizeof(struct node));
                p->exp = e;  p->coef = a;  p->next = NULL;
                if (head == NULL)     head = p;
                else      k->next = p;
                k = p;  /* 向前递推一步，k始终指向链表尾 */
                printf("input exp and coef:");
                scanf("%d%lf", &e, &a);
        }
        return head;  /* 返回链表的头指针 */
}
```

2. 多项式链表的释放

过程: 从表头开始,逐步释放链表中的各结点

其算法的C语言描述为:

```
void delpoly(struct node *head)  /*多项式链表的释放 */
{       struct node *p, *k = head;
        while (k != NULL)
        {       p = k->next;
                free(k);
                k = p;
        }
}
```

3. 多项式的输出

过程：

从表头结点后的第一个结点开始,以(a,b)数对形式顺链输出各结点中的指数域与系数域的内容。

其算法的C语言描述为：

```
void outpoly(struct node *head)  /*多项式链表的输出*/
{       struct node *p = head;
        while (p != NULL)
        {       printf("(%d, %lf)\n", p->exp, p->coef);
                p = p->next;
        }
}
```

outpoly函数还可以简化为：

```
void outpoly(struct node *head)  /*多项式链表的输出*/
{    while (head != NULL)
     {   printf("(%d, %lf)\n", head->exp, head->coef);
         head = head->next;
     }
}
```

4. 多项式的相加

过程：

假设多项式$A_m(x)$与$B_n(x)$已经用链表表示，其头指针分别为AH与BH，和多项式C(x)用另一个链表表示，其头指针为CH。多项式相加的运算规则：从两个多项式链表的第一个元素结点开始检测，对每一次的检测结果做如下运算：

- 若两个多项式中对应结点的指数值相等，则将它们的系数值相加。如果相加结果不为零，则形成一个新结点后链入头指针为CH的链表末尾。然后再检测两个链表中的下一个结点。
- 若两个多项式中对应结点的指数值不相等，则复抄指数值大的那个结点中的指数值与系数值，形成一个新结点后链入头指针为CH的链表末尾，并把指数值大的那个结点所在的链表的当前指针指向下一个结点。
- 直至两个链表中的一个链表为空终止循环。
- 最后把不为空的链表的剩余部分复制到CH链表的尾部。

其算法的C语言描述为：

```
struct node *addpoly(struct node *ah, struct node *bh)
{       struct node *k=NULL, *p, *m, *n, *ch=NULL;
        int e;  double d;
        m = ah; n = bh;
        while ( m != NULL && n != NULL)
        {       if (m->exp == n->exp) /* 指数相同， 系数求和 */
                {       d = m->coef + n->coef;  e = m->exp;
                        m = m->next;   n = n->next;
                }
                else if (m ->exp > n->exp) /* 复制指数大的结点 */
                {       d = m->coef;  e = m->exp;
                        m = m->next;
                }
                else    /*  m->exp < n->exp  */
                {       d = n->coef;  e = n->exp;
                        n = n->next;
                }
```

```
            if (d != 0)  /* 系数不为0，生成新结点插入ch链表 */
            {   p = (struct node *)malloc(sizeof(struct node));
                p->exp = e;    p->coef = d;
                p->next = NULL;
                if (ch == NULL) ch = p;
                else   k->next = p;
                k = p;  /* 向前递推一步，k始终指向链表尾 */
            }
    }
    while ( m != NULL)  /* 复制ah链表剩余部分 */
    {   p = (struct node *)malloc(sizeof(struct node));
        p->exp = m->exp;  p->coef = m->coef;  p->next = NULL;
        m = m->next;
        if (ch == NULL)  ch = p;
        else  k->next = p;
        k = p;    /* 向前递推一步，k始终指向链表尾 */
    }
```

```
        while ( n != NULL)    /* 复制bh链表剩余部分 */
        {   p = (struct node *)malloc(sizeof(struct node));
            p->exp = n->exp;  p->coef = n->coef;  p->next = NULL;
            n = n->next;
            if (ch == NULL)  ch = p;
            else  k->next = p;
            k = p;    /* 向前递推一步，k始终指向链表尾 */
        }
        return ch;
}
main( )
{       struct node *ah, *bh, *ch;
        ah = inpoly( );  /* 生成A多项式链表 */
        bh = inpoly( );  /* 生成B多项式链表 */
        ch = addpoly(ah, bh);  /* 多项式求和生成C多项式链表 */
        printf("A is: \n");     outpoly(ah);  /* 输出A多项式链表 */
        printf("B is: \n");     outpoly(bh);  /* 输出B多项式链表 */
        printf("C is: \n");     outpoly(ch);  /* 输出C多项式链表 */
        delpoly(ch);    delpoly(bh);   delpoly(ah);
}    /* 依次释放C、B、A多项式链表所占动态内存 */
```

运行结果：

```
input exp and coef : 3 5
input exp and coef : 1 3
input exp and coef : 0 2
input exp and coef : -1 -1
input exp and coef : 3 -5
input exp and coef : 2 4
input exp and coef : 1 2
input exp and coef : -1 -1
A is:
(3, 5.000000)
(1, 3.000000)
(0, 2.000000)
B is:
(3, -5.000000)
(2, 4.000000)
(1, 2.000000)
C is:
(2, 4.000000)
(1, 5.000000)
(0, 2.000000)
请按任意键继续…
```

$$A(x) = 5x^3 + 3x + 2$$

$$B(x) = -5x^3 + 4x^2 + 2x$$

$$C(x) = 4x^2 + 5x + 2$$

11.5 联合体

联合体 又称为共用体，各种不同数据共用同一段存储空间。

一般形式

```
union 联合体名
{ 成员表 };
```

- 虽然联合体与结构体在定义形式上类似，但它们在存储空间的分配上是有本质区别的。

结构体

按照定义中各个成员所需要的存储空间的总和来分配存储单元，其中各成员的存储位置是不同的。

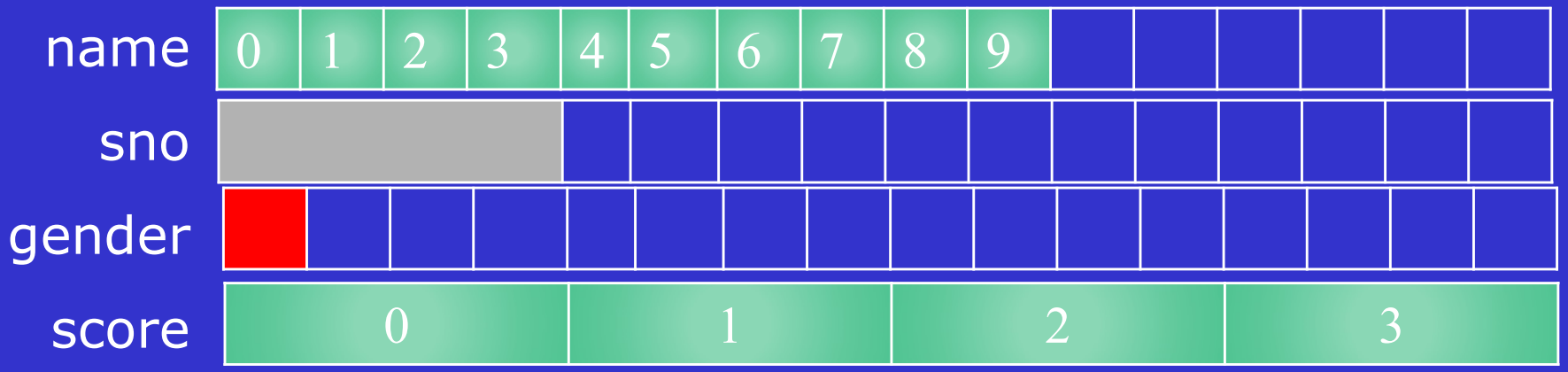
联合体

按定义中需要存储空间最大的成员来分配存储单元，其他成员也使用共享该空间，它们的首地址是相同的。

联合体 union 和 struct 的定义和使用是相同的。但 union 中的各个数据之间是共享同一个单元的，所占内存单元的大小就是几个数据项中长度最大的一个。

例如，对于联合体：

```
union student {
    char name[10];
    long sno;
    char gender;
    float score[4];
} std[100], *p, a, b;
```



- 则sizeof (union student)为16
- (=max(10*1, 4, 1, 4*4)),
  大小是最长一项 float score[4] 的长度。
- 由于数据是共享内存，而不是独立占有各自的内存，所以使用时，要注意赋值和使用的时序。
- 若有以下语句：
- a.sno=12345;
- a.gender = 'f';
- 则此时a.sno的值已经不是12345了。

有程序：

```
#include <stdio.h>
union EXAMPLE
{
    struct
    {
        int x, y;
    } in;
    int a, b;
} e;
main( )
{
    e.a=1;
    e.b=2;
    e.in.x=e.a*e.b;
    e.in.y=e.a+e.b;
    printf("%d,%d\n",e.in.x,e.in.y);
}
```

运行结果是什么？ **4,8**

```
union EXAMPLE
 {
    struct
    {
       int x, y;
    } in;
    int a, b;
 } e;
```

分析：对于联合体e，实际上是a和b和结构体in共享内存，再进一步，因为结构体in中有x和y，实际上是a和b和in.x共享内存。in.y的内存单元在in.x之后。

因此执行

```
    e.a=1;  e.b=2;
```

后，实际上a和b的值都是2。

执行 e.in.x=e.a*e.b; 之后，a和b和in.x的值都变成4。

执行 e.in.y=e.a+e.b;后，in.y的值为8。所以运行结果是4,8 。

在定义联合体类型变量时，不仅可以将类型的定义与变量的定义分开（如上面的定义），也可以在定义联合体类型的同时定义该类型的变量，或者直接定义联合体类型变量。

程序中不能直接引用联合体变量本身，而只能引用联合体变量中的各成员。引用的联合体变量成员的一般形式为：　　联合体变量名.成员名

说明

- 由于一个联合体变量中的各成员共用一段存储空间，因此，在任一时刻，只能有一种类型的数据存放在该变量中，即在任一时刻，只有一个成员的数据有意义，其他成员的数据是没有意义的。

说明

- 在引用联合体变量中的成员时，必须保证数据的一致。例如，如果最近一次存入到联合体变量中的是整型成员的数据，则在下一次取数时，也只能取该变量中整型成员中的数据，而取该变量中的其他类型成员中的数据一般是没有意义的。
- 在定义联合体变量时不能为其初始化，并且，联合体变量不能作为函数参数。
- 联合体类型与结构体类型可以互相嵌套，即联合体类型可以作为结构体类型的成员，结构体类型也可以作为联合体类型的成员。

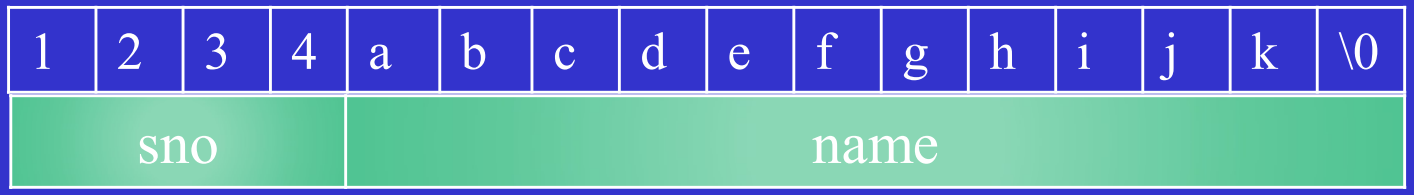
- 关于无名结构体/联合体

```
struct {
    long  sno;
    char  name[12];
    float  score[4];
};

union {
    long  sno;
    char  name[12];
    float  score[4];
};
```

- 无名结构体应用举例（可以快速分解收到的数据包）：

```
#include <stdio.h>
#include <string.h>
union {
   struct {
    long  sno;
    char  name[12];
    float  score[4];
  };  /* sno, name, score复合为一个整体，与 buf 数组形成联合体 */
  char buf[32];
} t;
void main( )
{ strcpy(t.buf, "1234abcdefghijk\0");
  printf("%d %s", t.sno, t.name);
} /* 因为是无名结构体，所以sno, name可以直接当做t的成员 */
```



运行结果：875770417 abcdefghijk

因为：875770417 = '4'*256^3+ '3' *256^2 +'2'*256+'1' 其中'1'=49

11.6 枚举类型与自定义类型名

11.6.1 枚举类型

① 可以先定义枚举类型名，然后定义该枚举类型的变量。

定义枚举类型名的一般形式为:

enum 枚举类型名{ 枚举元素列表 };

其中在枚举元素列表中依次列出了该类型中所有的元素（即枚举常量）, 如果在定义中没有显式地给出这些元素的值,则这些元素依次取值为0,1,2,…。

定义枚举类型名以后就可以定义该枚举类型的变量，其定义形式为:

enum 枚举类型名 变量表;

11.6 枚举类型与自定义类型名

11.6.1 枚举类型

例如： **enum week {sun, mon, tue, wed, thu, fri, sat};**

定义了枚举类型**week**，在这个类型中，共有七个常量枚举元素，分别为: **sun(值为0), mon(值为1), tue(值为2), wed(值为3), thu(值为4), fri(值为5), sat(值为6)** 。

定义了该枚举类型**week**后，就可以用该枚举类型**week**定义该类型的变量:

```
enum week a, b;
```

定义了枚举类型**week**的两个变量**a**与**b**，这两个变量中只能赋值为**sun**，**mon**，**tue**，**wed**，**thu**，**fri**，**sat**这七个常量元素之一，否则会有编译警告错误信息提示。

② 也可以在定义枚举类型的同时定义该枚举类型的变量。一般形式为：

enum 枚举类型名{ 枚举元素列表 } 变量表;

例如：**enum week {sun, mon, tue, wed, thu, fri, sat} a, b;**

③ 无名枚举类型，直接定义枚举类型变量。这种定义方法的一般形式为：

enum { 枚举元素列表 } 变量表;

例如：**enum {sun, mon, tue, wed, thu, fri, sat} a, b;**

- 无论是哪种方式定义的枚举变量，定义后，可以给枚举变量a，b赋值，例如：

a=mon; b=fri;

提示

- 不能对枚举元素赋值，因为枚举元素本身就是常量(即枚举常量)。在上面的定义中，下列赋值语句都是错误的：

mon=1;　fri=5;

- 虽然在程序中不能对枚举元素赋值，但实际上，每个枚举元素都有一个确定的整型值。如果在定义枚举类型时没有显式地给出各枚举元素的值，则这些元素的值按列出的顺序依次取值为**0, 1, 2, …**。但C语言还允许在对枚举类型定义时显式地给出各枚举元素的值。

例如：

```
enum  logic { FALSE, TRUE } flag;
```

- flag的取值是FALSE, TRUE 两个枚举常量，两个枚举常量实际值分别是FALSE为0，TRUE为1，系统总是默认从0开始给枚举常量赋值。
- `flag = TRUE;  printf("%d  ", flag);`
- `flag = FALSE; printf("%d\n", flag);`
- 结果将是：1 0

- 又例如：
- `enum Week {Sunday, Monday, Tuesday, Wednesday,`
- `Thursday, Friday, Saturday} week;`
- 则定义了枚举变量Week和7个枚举常量。其中Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday这7个枚举常量的值依次为0,1,2,3,4,5,6 。

可以自己指定enum常量的值：

```
enum  logic { TRUE=1, FALSE=-1} flag;
```

- 2个枚举常量实际值分别是TRUE为1，FALSE为-1。

- enum Week

```
{  Sunday=1, Monday=3, Tuesday=7, Wednesday=100,
   Thursday=1000, Friday=10000, Saturday= -1
} week;
```

- 7个枚举常量的值就是相应的指定值。这些枚举常量等同于宏定义所定义的常量。

- **C语言允许将一个整型值经强制类型转换后赋给枚举类型变量。 例如：a=(enum week)1; b=(enum week)6;**

【例11-11】 根据键盘输入的一周中的星期几（整数值）,输出其英文名称。

```
#include  <stdio.h>
main( )
{ int  n;
   enum week {sun ,mon, tue, wed, thu, fri, sat} weekday;
   printf("input n：  ");
   scanf("%d",&n);
   if ((n>=0) && (n<=6))
   {   weekday=(enum week)n;
```

```
    switch(weekday)
    {  case sun：  printf("Sunday\n"); break;
       case mon: printf("Monday\n"); break;
       case tue：  printf("Tuesday\n"); break;
       case wed: printf("Wednesday\n"); break;
       case thu：  printf("Thursday\n"); break;
       case fri：  printf("Friday\n"); break;
       case sat：  printf("Saturday\n");
    }
  }
  else printf("ERR!\n");
}
```

程序的运行结果是：

input n:2

Tuesday

请按任意键继续…

实际上，【例11-11】可以用字符指针数组重写程序为：

```
#include  <stdio.h>
main( )
{  char *Weeks[ ]={"Sunday", "Monday", "Tuesday",
                   "Wendesday", "Thursday", "Friday", "Saturday"};
    int n;
    printf("input n：  ");
    scanf("%d", &n);
    if ((n>=0) && (n<=6))
        printf("%s\n", Weeks[n]);
    else
        printf("ERR!\n");
}
```

程序的运行结果是：

```
input n:2
Tuesday
请按任意键继续…
```

11.6.2 自定义类型名

自定义类型名 用typedef声明新的类型名来代表已有的类型名

一般形式

typedef 原类型名 新类型名;

它指定用新类型名代表原类型名。原类型名仍可继续使用。

特别注意：利用 typedef 声明只是对已经存在的类型增加了一个类型别名，而没有定义新的类型。另外，在用 typedef 指定新类型名时，习惯上将新类型名用大写字母表示或下划线开头，以便与系统提供的标准类型标识符相区别。

- 自定义类型 typedef 只是为了用户书写程序的方便，把书写复杂的类型另起一个别名，这也可以使程序简洁易读。例如：

struct student 定义后，我们可以用这个struct student类型定义变量，申请动态内存：

```
struct student *p, a;
p=(struct student *)malloc(sizeof(struct student));
```

- 若有定义：

```
typedef struct student STU;
```

则上面的两个语句就可以书写成：

```
STU *p, a;
p = (STU *)malloc(sizeof(STU));
```

注意：

- 1. typedef 只是使书写简单并且易读。
- 2. typedef 与宏定义一样，只是一种符号的代换，并不能提高程序的执行效率。

- VS2008中的 _int8, _int16, _int32, _int64 类型来自：
- typedef signed char _int8;
- typedef signed short _int16;
- typedef signed int _int32;
- typedef signed long long _int64;

- 另外，还可以用typedef来声明数组类型。例如，

  typedef int NUM[100];

- 指定用NUM代表具有100个整型元素的整型数组类型，NUM是int [100]类型，自定义该整型数组类型NUM后，就可以用该整型数组类型NUM定义该数组类型的变量，例如

  NUM x, y;

- 定义了整型数组类型的两个变量x与y，在这两个变量中均包含100个整型元素，即实际上定义了两个长度均为100的整型一维数组x与y。

- 因此 NUM x, y; 等价于：

  int x[100], y[100];

如果：　　typedef int *NUM[100];

那么：　　NUM x, y;

等价于：　int *x[100], *y[100];

NUM是 int * [100] 指针数组类型

```
typedef struct node {
    int value;
    struct node *left, *right;
} NODE, *PNODE;
NODE   a;          /* NODE 是：   struct node  类型  */
PNODE  p, *q;    /* PNODE 是：  struct node * 类型 */
```

- a是一个struct node变量， 可以 a.value=5;
- p是一个struct node指针， 可以 p=&a; p->value=10;
  PNODE p; 等价于： struct node *p;

- q是一个指向struct node指针的指针， 可以
  q=&p; (*q)->value=10;
  PNODE *q; 等价于： struct node **q;

如果：

```
typedef struct node {
        int value;
        struct node *left, *right;
} node;
node a;
struct node b;
```

自定义类型名可以和结构体类型同名。
并可以同时使用！

有一种写法：

```
typedef struct node {
    int value;
    struct node *left, *right;
};
node a;          /* 错误！ */
struct node a; /* 正确！ */
```

因此上述定义中的 typedef 无任何作用，等价于：

```
struct node {
    int value;
    struct node *left, *right;
};
```

补充：浅复制(浅拷贝) (Shallow Copy) 与 深复制(深拷贝) (Deep Copy)

结构体可以整体赋值，例如：

```
struct student {
   char name[10];
   int  sno;
   float grade;
} a={"ZhangSan", 201101011, 85.5}, b;
b = a;          /* a的各成员都对应赋值给b */
```

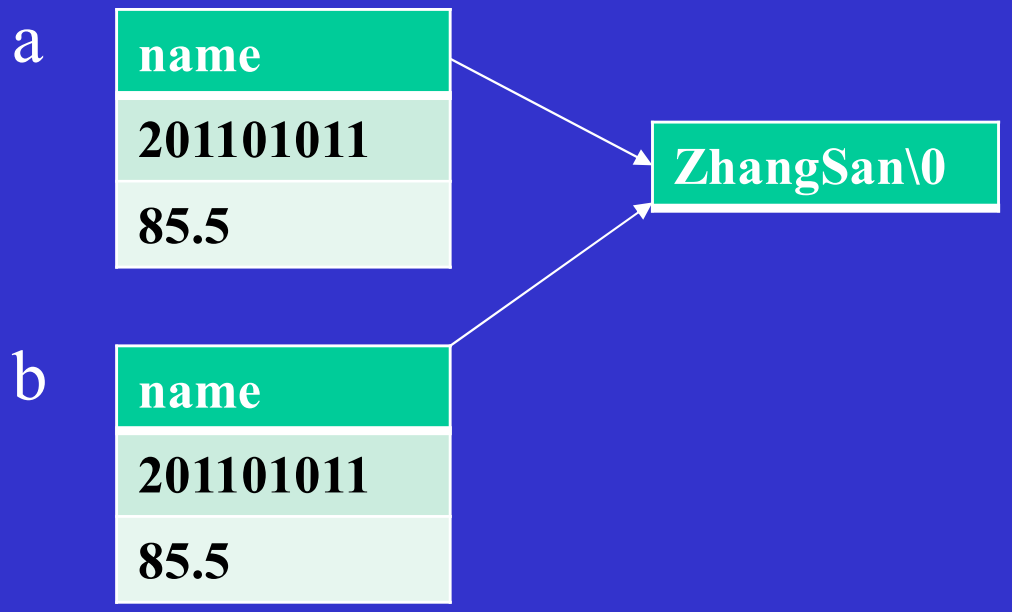
即结构整体复制，将一个结构体变量的各成员的值对应赋值给另一个结构体变量的各成员：

整型、实型、数组、字符串、指针。

但指针赋值会使不同的结构变量的指针指向同一块内存。

若结构体定义改为：

```
struct student {
   char *name;
   int  sno;
   float grade;
} a, b;
a.name = (char *)malloc(sizeof(char)*10);
strcpy(a.name, "ZhangSan");
a.sno= 201101011;  a.grade=85.5;
b = a;    // a的各成员都对应赋值给b
```

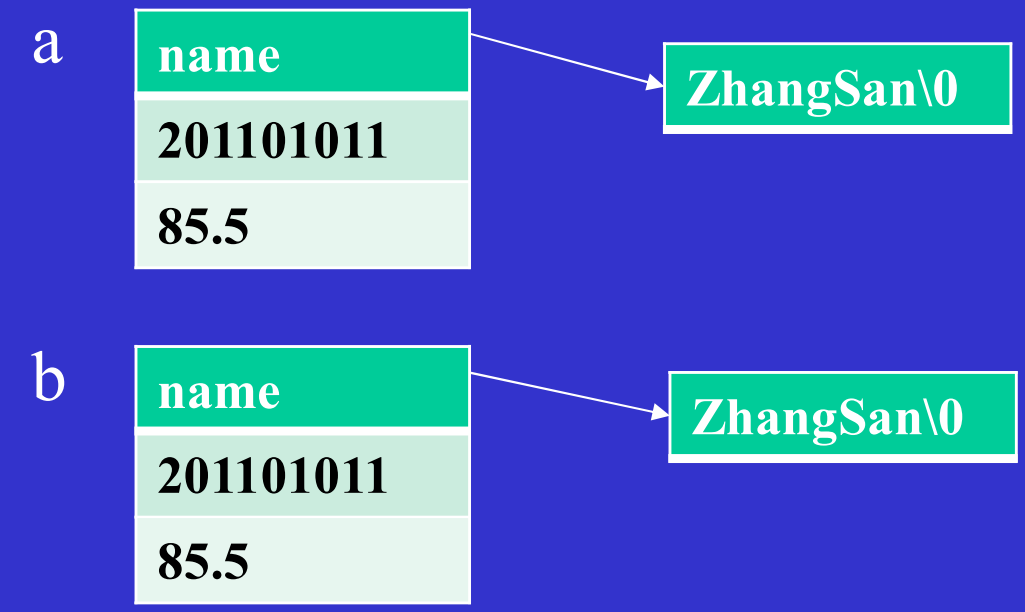


则a.name指针和b.name指针指向同一块内存。若要修改b.name的内容则同时也修改了a.name的内容，而且后患无穷。

提示: 结构体整体赋值或参数传递属于 浅复制(浅拷贝)。

若程序改为：

```
struct student {
   char *name;
   int  sno;
   float grade;
} a, b;
a.name = (char *)malloc(sizeof(char)*10);
strcpy(a.name, "ZhangSan");
a.sno=201101011; a.grade=85.5;
b = a;      // a的各成员都对应赋值给b
b.name = (char *)malloc(sizeof(char)*10);
strcpy(b.name, a.name);
```



则a.name指针和b.name指针指向不同的动态内存。修改b.name的内容与a.name无关。这就是 深复制(深拷贝)。

第10次作业(第11章)

p.315-317 习题 1， 2，11，12* (选做）

探究题：用环形链表解约瑟夫问题。

所谓环形链表，就是单链表尾指针又指向了链表头，形成环状。

所谓约瑟夫问题，就是，m个人每人一个编号：1, 2, 3, …, m，排成一个圆圈，由第1个人开始报数，每报数到第n人则该人就出列，然后再由下一个重新报数，直到剩下最后1个人，并打印出其编号。