单链表程序设计实例

【例7.11】写一个函数，输入n个整数，并按整数的输入顺序建立一个整数单链表。

【解题思路】

从空单链表开始，每输入一个整数，就向系统申请一个表元的存储空间，将输入整数存入新表元，并将新表元接在单链表末尾。当n个整数全部输入，并插入到单链表后，函数返回生成的单链表的头指针。

为使新表元能方便地接在单链表的末尾，另引入一个指针变量tail总是指向单链表的末尾表元。建立空链表时，头指针h和末尾表元指针tail都置NULL。将新表元接在链表末尾要分两种情况。一是原单链表为空单链表；二是原单链表有表元。图7-11是在非空单链表末尾表元之后接入一个由指针p所指新表元的示意图。接入新表元，首先是让tail所指的尾表元的后继指针由NULL改为指向p所指的新表元，接着是让tail指向新的尾表元。这两项改动可用代码tail = tail->next = p实现。

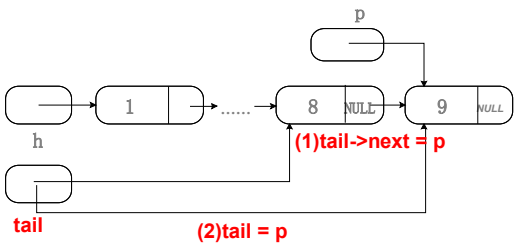


图7-11在链表末尾表元之后接新表元，链表状态变化图

```
intNode *createList(int n)
{ intNode *h, /* 链表的头指针 */
             *tail, /* 链表末尾表元的指针 */
             *p;
  int  k;
  h = tail = NULL;
  printf("Input data.\n");
  for(k = 0; k < n; k++) {
      p = (intNode *)malloc(sizeof(intNode));
      scanf("%d", &p->value);
      if (h == NULL) h = tail = p;
      else  tail = tail->next = p;
  }
  if(tail) tail->next  = NULL;/*末表元之后为空*/
  return h;
}
```

输入n个整数，并按整数的输入顺序建立一个整数单链表

```
#include <stdio.h>
#include <malloc.h>
struct intNode
{ int value;
  intNode *next; //采用C++句法，定义结构成员
};
/* 建立正整数链表的函数可以放在这里 */
```

```
/* 主函数 */
void main()
{ intNode *p, *q; //采用C++句法，定义结构指针
  int n;
  printf("输入链表的表元个数!\n");
  scanf("%d", &n);
  q = createList( (n); /* 当函数返回时，q为头指针 */
  while (q) {
   printf("%d\t",q->value);//依次显示链表中的表元值
   p = q->next;  /*保存后继表元指针*/
   free(q);/*删除当前表元*/
   q = p;  /*后继表元成为当前表元*/
  }
}
```

【例7.12】编写一个函数，输入n个整数，建立一个按值从小到大顺序链接的链表。

【解题思路】

输入整数构建有序链表的过程是一个循环。这里采用以下算法思想：

(1)要构建的链表的初始状态为空链表，即头指针head为NULL；

(2)构建有序链表的过程是一个循环，循环完成以下工作：

●输入一个整数，就向系统申请一个表元的存储空间并赋给指针变量p，将输入的整数存入新表元；

●寻找新表元插入位置；

●将新表元插入在插入处的前面。

n个整数全部输入后，函数返回生成的单链表的头指针。

寻找插入位置也是一个循环：

从链表的首表元开始考察，因越大的数插在越后面，如果输入整数大于当前表元的值，需继续考察下一个表元；否则，新表元插在当前表元之前。

由于单链表上无法简单地完成插在某表元之前的工作，寻找循环引入指向当前要考察的表元的指针变量v和指向当前要考察的表元前一个表元的指针变量u；

插入在v所指表元之前，改为插入在u所指表元之后。

```
intNode *createSortList(int n)
{ intNode *u, *v, *p, *head=NULL;/*u指向前驱，v指向后继 */
  int  k;
  printf("Input data.\n");
  for(k = 0; k < n; k++) {
    p = (intNode *)malloc (sizeof(intNode));
    printf ("请输入一个整型数据： ");
    scanf ("%d", & p->value);/* 输入的整数存入新表元 */
    v = head;  /* 从首表元开始寻找插入位置 */
    while(v != NULL && p->value > v->value) //寻找要插入的位置
    {u = v; v = v->next; }//保存前驱表元指针，并考察下一个表元
    if(v == head)  head = p;/* 新表元插在首表元之前 */
    else  u->next=p;//新表元插在u所指表元之后，v所指表元之前
    p->next = v;
  }
  return head;/* 返回链表的头指针head */
}
```

【例7.13】编制一个函数，实现将已知单链表的表元链接顺序颠倒。即使单链表的第1个表元变为最末一个表元，第2个表元变为最后第2个表元，……，最后一个表元变为第1个表元。

依旧设链表为整数链表，且设链表是带辅助表元的。图7-12 表示链表颠倒前和颠倒后的链表状态。

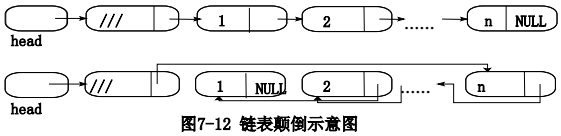
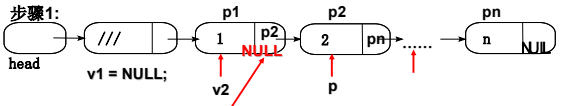


图7-12 链表颠倒示意图

为实现从颠倒前状态变为颠倒后的状态，可用以下四个赋值操作实现：

```
p = v2->next; //v2:当前指针,保护v2所指表元的后继指针,
v2->next = v1;//使v2所指表元的后继表元是v1所指表元
v1 = v2;        /* 调整v1 */
v2 = p;         /* 调整v2 */
```

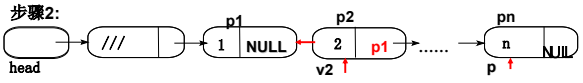
步骤1:



从首节点开始为v2节点, v2的下一个节点为p2节点,若v2是首节点,则v2的next为NULL.

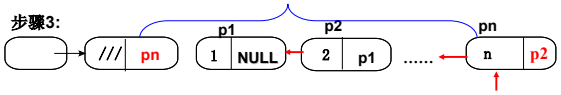
```
p=v2->next   //保留下一个节点地址p2;
v2->next = v1;
v1 = v2 ; // 保留当前指针p1;
v2 =  p;  // 取下一个节点p2;
```

步骤2:




```
p=v2->next;        //pn
v2->next = v1;     //p1
v1 = v2;           //p2
v2=p;              //pn
```

步骤3:




```
p=v2->next;//NULL  v2->next = v1;//p2;
v1 = v2;//pn;  v2=p;//NULL;
head->next=v1=pn
```

```
void reverse(intNode *h)
{  intNode *p, *v1, *v2;
   v1 = NULL; /* 开始颠倒时，已颠倒部分为空 */
   v2 = h->next;   /* v2 指向链表的首表元 */
   while(v2 != NULL) { /* 还未颠倒完，循环 */
     p = v2->next; //保护v2所指表元的后继指针
     v2->next = v1;//使v2所指表元的后继表元是v1所指表元
     v1 = v2; /* 调整v1 */
     v2 = p; /* 调整v2 */
     }
   h->next = v1;
   return;
}
```

如果颠倒的单链表没有辅助表元，则函数应返回颠倒后的链表头指针，单链表颠倒函数可以改写如下：

```
intNode *reverse (intNode *h)
{  intNode *p, *v1, *v2;
   v1 = NULL;/* 开始颠倒时，已颠倒部分为空 */
   v2 = h;   /* v2 指向链表的首表元 */
   while(v2 != NULL) { /* 还未颠倒完，循环 */
         p = v2->next; v2->next = v1;
         v1 = v2;      v2 = p;
   }
   return v1;
}
```

【例7.14】设有顺序编号为1至n的n个人，按顺时针顺序站成一个圆圈。首先从第1个人开始，从1开始顺时针报数，报到第m（<n）个人，令其出列。然后再从出列的下一个人开始，从1顺时针报数，报到第m个人，再令其出列，------，如此下去，直到圆圈不再有人为止。求这n个人出列顺序。

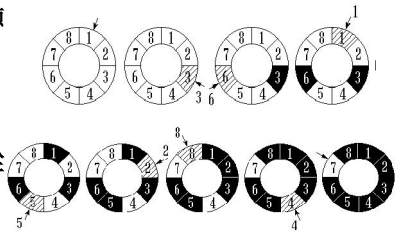


图7-14 n = 8，m = 3,8个人站成圆圈，从第一个人开始报数，每次第3人出列的示意图。

【解题思路】

程序首先输入n和m，接着构造一个有n个表元组成的循环链表。下面的程序将构造循环链表的工作交由函数makeLoop()完成，函数返回循环链表的首表元指针。主函数的主要工作是一个循环，每次循环完成报数和出列的工作。为了控制报数到第m人出列，报数循环只能重复向前m-2次，直至第m-1人报数，第m-1人的下一个人出列。出列时要考虑到最后一个人出列时，循环链表变成空链表。

```
#include  <stdio.h>
#include <malloc.h>
struct Node{
    int num;
    Node *next;
};
/*构造含n个表元的循环链表，函数返回循环链表的末表元指针*/
Node *makeLoop(int n)
{   Node *h, *tail, *p;
    int i;
    for(h = tail = NULL, i = 1; i <= n; i++) {
        p = (Node *)malloc(sizeof(Node));
        p->num = i; /*生成第i号表元*/
        if(h == NULL) h = tail = p; /*插入首表元*/
        else tail = tail->next = p; /*第i号表元接在链表末尾*/
    }
    tail->next = h; /*构成循环链表*/
    return h; /*返回首表元指针*/
}
```

```
void main()
{ int n, i, m;   Node *h, *p, *t;
   printf("Enter n & m\n");    scanf("%d%d", &n, &m);
   if(m == 1){ for(i = 1; i <= n; i++)  printf("%4d", i);
      return;
   }
   p = h = makeLoop(n);
   while(p) {//输出
      for(i=1;i<m-1;i++)p=p->next;/*向前m-2次，报数m-1人*/
      t = p->next; /*下一个表元出列*/
      printf("%4d", t->num);  p = p->next = t->next;
      if(p == t) p = NULL; /*最后一个表元出列，变成空链表*/
      free(t); /*释放出列表元的空间*/
   }
   printf("\n");
}
```

7.8 类型定义(typedef)

除了可以直接使用C提供的标准类型名（如int、char、float、double、long等）和前面介绍的结构类型、指针、类型外，C语言允许由用户自己定义类型说明符，也就是说允许用户为数据类型取“别名”。这一功能要求用类型定义符typedef来完成。

例如，整型变量说明符int取自单词integer的前三个字母，为了增加程序的可读性，可把整型说明符用typedef定义为：

typedef int INTEGER;

以后就可用INTEGER来代替int作整型变量的类型说明。 同样地，也可以用语句：

typedef float REAL;

来使REAL代替float作为实型变量的类型说明。经过用typedef说明后，语句

INTEGER a,b;就等效于语句int a,b;

而语句

REAL x,y等效于语句float x,y;。

typedef定义的一般形式为：

typedef 原类型名 新类型名;

其中原类型名是系统提供的类型符，新类型名一般用大写表示，以便于区别。

另外，用typedef定义数组、指针、结构等类型将带来很大的方便，不仅使程序书写简单，而且使意义更为明确，因而增强了可读性。

例如：

```
typedef int NUM[100]; /*定义NUM为整型数组，该数组元素有100个*/
NUM a, b, c;          /*定义了三个数组元素达100的整型数组*/
typedef char *STRING;  /*定义STRING为字符指针类型*/
STRING p,s[10];       /*p为字符指针变量，s为指针数组 */
typedef int (*POINTER)( )/*定义POINTER为指向函数的指针类型，函数返回整型值*/
POINTER p1,p2;      /*  p1和p2为指向函数的指针变量  */
```

又例如语句：

```
typedef struct date
{    int month;
     int day;
     int year;
}DATE;
```

定义了一个新类型名DATE，它代表所定义的一个结构类型。这时就可以用DATE定义变量：

```
DATE birthday;  /*birthday是结构变量*/
DATE *p;        /*p为指向此结构类型数据的指针*/
```

例: typedef int INTARRAY[20];
/*含20个整数的数组类型INTARRAY */

利用以上类型定义，可定义变量：
INTARRAY v1,v2; /*定义两个各含20个整数的数组*/
在以上变量定义中，对于结构，枚举等类型，不必再冠相应的类型关键字。对于数组类型，当有多个数组类型相同且元素个数也相同时，先用 typedef 定义一个数组类型，然后再定义数组变量就比较方便，简洁。

类型定义符typedef的几点说明：

(1) 用typedef可以定义各种类型名，但不能用来定义变量。

(2) 用typedef只是对已经存在的类型增加一个类型名，而没有创造新的类型。

(3) typedef与#define有相似之处，如：typedef int COUNT;和#define COUNT int 的作用都是用COUNT代表int。但事实上，它们两者是不同的。#define是在预编译时处理的，它只能作简单的字符串替换，而typedef是在编译时处理的。实际上它并不是作简单的字符串替换，例如：

typedef int NUM[10];

并不是用NUM[10]去代替int，而是采用如同定义变量的方法那样来定义一个类型。当用typedef定义一些数据类型（尤其是象数组、指针、结构、共用类型等类型数据）时，可把它们单独放在一个文件中，然后在需要用到它们的文件中用#include命令把它们包含进来。

(4) 使用typedef有利于程序的通用与移植。有时程序会依赖于硬件特性，用typedef便于移植。例如，有些计算机系统int型数据用两个字节，数值范围为 -32768～32767；而另外一些机器则以4个字节存放一个整数，相当于前一种机型的long int型。若要把一个C程序从一个以4个字节存放整数的计算机系统移植到以2个字节存放整数的系统，一般需要将程序中的每个int变量都改为long int变量，即把“int a,b,c;”改为“long int a,b,c;”。现在可只加一行“typedef int INTEGER;”语句，而在程序中用INTEGER定义变量。当对程序进行移植时，只需将typedef定义体改为“typedef long INTEGER;”即可。

7.9 变量定义

形式化地描述变量定义或变量说明的句法，有以下形式：

存储类 类型限定符 类型区分 数据区分

其中存储类有：

auto、register、extern、static和缺省

见5.8存储类别和作用域。

类型限定符有三种形式：

类型名、typedef和typedef 类型名。

类型区分有简单类型和构造类型之分。简单类型有：

char、 int、 float、double、short、unsigned、long、void；或在int之前有short、unsigned或long；在char之前有unsigned；在double之前有long。以及用typedef定义的简单类型名。

构造类型有：

struct 结构类型名、结构类型定义

union 联合类型名、联合类型定义

enum 枚举类型名、枚举类型定义

用typedef定义的构造类型名，包括数组类型的类型名。

数据区分由一个或多个"数据说明符 初值符"对组成，其中初值符可以没有。当有多个"数据说明符 = 初值"时，它们之间用逗号分隔。

数据说明符又有多种形式，最简单的是一个标识符；另有多种构造形式，它们是：

（数据说明符）、* 数据说明符、数据说明符()、数据说明符[常量表达式]、数据说明符[]。

假设把类型区分符说明的类型记为Type。如果数据说明符是一个未加任何修饰的标识符，则它说明标识符的类型是Type的。例如

int j;

数据说明符为标识符j，它的类型为int型的。

数据说明符"（数据说明符）"，相当于未加修饰的数据说明符，使用括号只是用来改变复杂数据说明符中有关内容的结合顺序。

数据说明符"* 数据说明符"，使数据说明符所含标识符的类型是"指向Type类型的指针"。例如

int *ip

使标识符ip的类型是指向int型的指针。

数据说明符"数据说明符()"，使数据说明符所含标识符具有"返回Type 类型值的函数"的类型。例如

int f()

则标识符f具有"返回 int 型值的函数"的类型。

数据说明符"数据说明符[常量表达式]"或"数据说明符[]"，则数据说明符所含的标识符具有"元素类型为Type 的数组"的类型。例如

int a[100] 或 int a[]

则标识符a就具有"元素类型为int的数组"的类型，或简单地说成a是int型的数组。

"= 初值"用于对定义的变量设置初值，可以缺省。对于静态变量或全局变量如果未指定初值，系统自动置二进位全是0的值。对于自动的变量或寄存器类的变量，它们的初值可以是一般的可计算表达式。如果没有给定初值，则变量的初始值是不确定的。

静态变量或全局变量的初值可以有下面多种形式：

常量表达式、字符串常量、&变量、&变量+整常量、&变量 - 整常量、{初值表}。

其中初值表中的初值之间用逗号分隔，初值表中的表达式必须是常量表达式，或者是前面说明过的变量的地址表达式，或是字符串。

对静态变量或全局变量来说，初始化工作只做一次，一般在程序执行之前进行。

对于结构或数组，初值是由花括号括起，用逗号分隔每个成分的初值表。

初值表中的初值按数组元素的下标或结构成分的定义顺序给出。

初值表中的初值个数比成分个数多是错误的；而比成分个数少时，系统用零填补。

字符串初值是用于对字符数组或字符串指针初始化。特别地，当数组定义中的元素个数省略时，编译程序通过计数初值个数确定数组的元素个数。

7.9 变量定义

形式化地描述变量定义或变量说明的句法，有以下形式：

存储类 类型限定符 类型区分 数据区分

其中存储类有：

auto、register、extern、static和缺省

见5.8存储类别和作用域。

类型限定符有三种形式：

类型名、typedef和typedef 类型名。

类型区分有简单类型和构造类型之分。简单类型有：

char、 int、 float、double、short、unsigned、long、void；或在int之前有short、unsigned或long；在char之前有unsigned；在double之前有long。以及用typedef定义的简单类型名。

构造类型有：

struct 结构类型名、结构类型定义

union 联合类型名、联合类型定义

enum 枚举类型名、枚举类型定义

用typedef定义的构造类型名，包括数组类型的类型名。

数据区分由一个或多个"数据说明符 初值符"对组成，其中初值符可以没有。当有多个"数据说明符 = 初值"时，它们之间用逗号分隔。

数据说明符又有多种形式，最简单的是一个标识符；另有多种构造形式，它们是：

（数据说明符）、* 数据说明符、数据说明符()、数据说明符[常量表达式]、数据说明符[]。

假设把类型区分符说明的类型记为Type。如果数据说明符是一个未加任何修饰的标识符，则它说明标识符的类型是Type的。例如

```
int j;
```

数据说明符为标识符j，它的类型为int型的。

数据说明符"（数据说明符）"，相当于未加修饰的数据说明符，使用括号只是用来改变复杂数据说明符中有关内容的结合顺序。

数据说明符"* 数据说明符"，使数据说明符所含标识符的类型是"指向Type类型的指针"。例如

```
int *ip
```

使标识符ip的类型是指向int型的指针。

数据说明符"数据说明符()"，使数据说明符所含标识符具有"返回Type 类型值的函数"的类型。例如

```
int f()
```

则标识符f具有"返回 int 型值的函数"的类型。

数据说明符"数据说明符[常量表达式]"或"数据说明符[]"，则数据说明符所含的标识符具有"元素类型为Type 的数组"的类型。例如

```
int a[100]      或      int a[]
```

则标识符a就具有"元素类型为int的数组"的类型，或简单地说成a是int型的数组。

"= 初值"用于对定义的变量设置初值，可以缺省。对于静态变量或全局变量如果未指定初值，系统自动置二进位全是0的值。对于自动的变量或寄存器类的变量，它们的初值可以是一般的可计算表达式。如果没有给定初值，则变量的初始值是不确定的。

静态变量或全局变量的初值可以有下面多种形式：

常量表达式、字符串常量、&变量、&变量+整常量、&变量 - 整常量、{初值表}。

其中初值表中的初值之间用逗号分隔，初值表中的表达式必须是常量表达式，或者是前面说明过的变量的地址表达式，或是字符串。

对静态变量或全局变量来说，初始化工作只做一次，一般在程序执行之前进行。

对于结构或数组，初值是由花括号括起，用逗号分隔每个成分的初值表。

初值表中的初值按数组元素的下标或结构成分的定义顺序给出。

初值表中的初值个数比成分个数多是错误的；而比成分个数少时，系统用零填补。

字符串初值是用于对字符数组或字符串指针初始化。特别地，当数组定义中的元素个数省略时，编译程序通过计数初值个数确定数组的元素个数。

基本算法

- 排序算法
  - 插入排序算法
  - shell(希尔)排序
  - 冒泡排序算法
  - 选择排序算法
- 查找算法
  - 顺序查找算法
  - 二分查找算法

按照存储方法来实现算法

- 顺序存储方法（采用数组）
  - 顺序存储的顺序查找算法
  - 顺序存储的二分查找算法
  - 顺序存储的插入排序算法
  - 顺序存储的冒泡排序算法
  - 顺序存储的选择排序算法
- 链接存储方法（采用链表）
  - 链接存储的顺序查找算法
  - 链接存储的插入排序算法

存储方法定义

- 顺序存储定义（采用数组）

```
#define N  20   /* 数据个数 */
int a[N]; /* 数据存储 */
int key; /* 待查找数据 */
```

- 链接存储定义（采用链表）

```
#defien NODE struct node
NODE
{       int value;
        NODE *next;
};
int key; /* 待查找数据 */
```

排序算法

◆顺序存储的插入排序算法

e.g: 36、24、10、6、12存放在 r 数组的下标为 1 至 5 的元素用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中

0	1	2	3	4	5
	36	24	10	6	12

r[0] 用作哨兵。共执行 5 遍操作。

每遍操作：先将元素复制内容放入r[0]，再将本元素同已排序的序列，从尾开始进行比较。在已排序的序列中寻找自己的位置，进行插入。或者寻找不到，则一直进行到哨兵为止。意味着本元素最小，应该放在 r[1] 。

每一遍，排序的序列将增加一个元素。如果序列中有 n 个元素，那么最多进行n 遍即可。

e.g: 36、24、10、6、12存放在 r 数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。

0	1	2	3	4	5
	36	24	10	6	12
		i			
24	36	24			

	36	24	10	6	12
		i			
24		36			

e.g: 36、24、10、6、12存放在 r 数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。

0	1	2	3	4	5
	36	24	10	6	12
		i			
24		36			

e.g: 36、24、10、6、12存放在 r 数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。

0	1	2	3	4	5
	36	24	10	6	12
		i			
24	24	36			

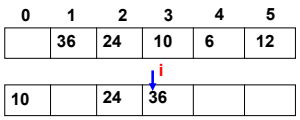
e.g: 36、24、10、6、12存放在 r 数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。

0	1	2	3	4	5
	36	24	10	6	12
			i		
10	24	36	10		

e.g: 36、24、10、6、12存放在 r 数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。

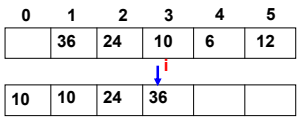
0	1	2	3	4	5
	36	24	10	6	12
			i		
10	24		36		

e.g: 36、24、10、6、12存放在 r 数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。



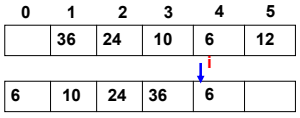
43

e.g: 36、24、10、6、12存放在 r 数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。



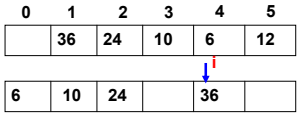
44

e.g: 36、24、10、6、12存放在 r 数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。



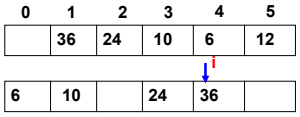
45

e.g: 36、24、10、6、12存放在 r 数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。



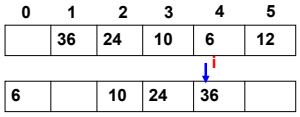
46

e.g: 36、24、10、6、12存放在 r 数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。



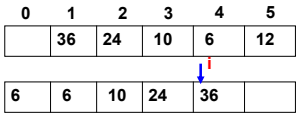
47

e.g: 36、24、10、6、12存放在 r 数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。



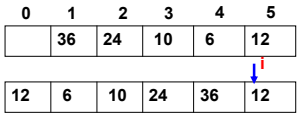
48

e.g: 36、24、10、6、12存放在 r 数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。



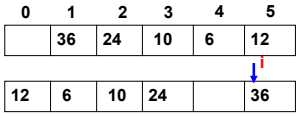
49

e.g: 36、24、10、6、12存放在 r 数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。



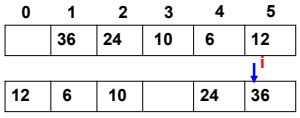
50

e.g: 36、24、10、6、12存放在 r 数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。



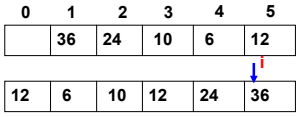
51

e.g: 36、24、10、6、12存放在 r 数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。



52

e.g: 36、24、10、6、12存放在 r 数组的下标为 1 至 5 的元素之中，用直接插入法将其排序。结果仍保存在下标为 1 至 5 的元素之中。



53

(1) 直接插入排序:用结构数组实现程序

```
//*PROGRAM  :直接插入排序 *
#include <stdlib.h>
#include <time.h>
#include <stdio.h>
typedef struct          //定义排序表的结构
{
        int *value; //数据元素序列
        int count;    //表中元素的个数
}SqList;
void InitialSqList(SqList *);  //初始化排序表
void InsertSort(SqList *);     //直接插入排序
void PrintSqList(SqList *);    //显示表中的所有元素
```

54

```
void main()
{       SqList L;  //声明表L
        char j='y';
        while(j!='n'&&j!='N')
        {       InitialSqList( &L);
                printf("没有排序的序列如下：\n");
                PrintSqList(&L);
                InsertSort(&L);
                printf("已排好序的序列如下：\n");
                PrintSqList(&L);
                printf("继续进行下一次排序吗?(Y/N)");
                scanf(" %c",&j);
        }
}
```

```
void InitialSqList(SqList *L)//表初始化
{
    int i;
    printf("请输入待排序的记录的个数:");
    scanf("%d",&L->count);
    L->value=(int *)malloc(L->count*sizeof(int));
    //申请结构数组成员value的个数
    srand(time(NULL));
    for(i=1;i<=L->count;i++)
        L->value[i]=rand()%100;
    //rand()%100：生成n个小于100的随机数
}
```

```
void InsertSort(SqList *L)
{
   int i,j;
   for(i=1;i<L->count;i++)
    if(L->value[i]<L->value[i-1])
    {//"<",需将L-value[i]插入有序子表
        L->value[0]=L->value[i];//复制为哨兵
        for(j=i-1;L->value[0]<L->value[j];--j)
          L->value[j+1]=L->value[j];//记录后移
          L->value[j+1]=L->value[0];//插入到正确的位置
       }
}
```

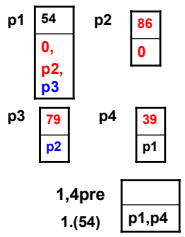
```
void PrintSqList(SqList *L)//显示表所有元素
{
        int i;
        for(i=1;i<=L->count;i++)
          printf("%d%c",L->value[i],i%6?'\t':'\n');
        printf("\n");
}
```

(2)直接插入排序:用链接存储的插入排序算法实现程序

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
typedef struct  S     //定义链表结构
{
        int value;
        S   *next;
}SqList;

void PrintSqList(SqList *); //显示表中的所有元素
SqList  *ptr;
```

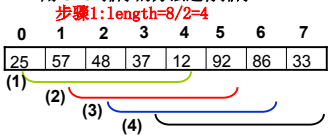
```
void append(SqList *ptr,int data)
{  //有序链表的形成, 输入54,86,79,39
   SqList *pre,*suc,*p;
   p =(SqList *)malloc(sizeof(SqList));
   p->value=data;
   pre=ptr;//h,h,h,h
   suc=ptr->next;//0,p1,p1,p1
   while(suc )
   { // 2.  (54 ,86),   3.(54,79),   (86,79),   4. (54,39)
     if(suc->value>=p->value)
        break;//如果新插入的小于当前节点,退出
        pre=suc;//保留前驱:p1,p1,
        suc=suc->next;//取下一个节点地址0,p2,
   }
    pre->next=p; //新节点插入到前驱地址后面
    p->next=suc;//将后继地址放到新节点的地址成员
}
```



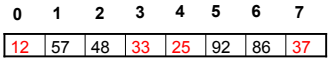
shell (希尔)排序

将要排序的数值按照某个间隔长度分成若干个数列的集合,再针对各个数列进行插入排序,重复进行数列分割,每次分割的间隔长度缩小为上一次的二分之一,直到分割长度为0,排序完成.e.g:
将序列 25、57、48、37、12、92、86、33
用 shell排序的方法进行排序.

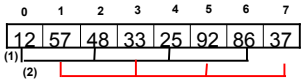
步骤1:length=8/2=4



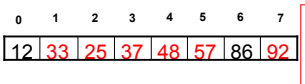
经过对集合(1)∽(4)分别进行"插入排序",得到下图



步骤2:length=4/2=2

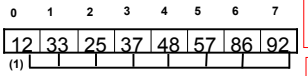


经过对集合(1)∽(2)分别进行"插入排序",得到下图



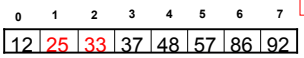
shell排序的优点是:

以插入的方法排序,方法简单.由于插入法对已经排好序的部分会快速的处理,故最后几次的排序速度会提高很多.



程序实现:类似于直接插入排序的程序。注意修改步长。另外,shell 排序的分析非常困难,原因是何种步长序列最优难以断定。

经过对集合(1)进行"插入排序",得到下图



查找算法

◆顺序存储的顺序查找算法

在数组 a 中，查找数据值等于key 的数组元素a[i]，返回i 值。如果查找失败，返回-1。

```
int search(int a[], int n, int key)
{
        int i;
        for(i=0; i<n; i++)
        if(a[i] == key)
        return i; /* 查找成功, 返回i */
        return -1; /* 查找失败, 返回-1 */
}
```

◆顺序存储的二分查找算法

二分查找法的算法（4.2.4节）

假定数组 a 的元素已按它们的值从小到大的顺序存放（称为已排序）。则二分法是更好的查找方法。其算法基本思想是对任意a[begin]到a[end](begin<=end)的数组元素，试探元素a[middle]，middle=(begin+end)/2，根据比较a[middle]与key 的结果分别采取不同的对策。算法描述为：

初始，令查找的下界begin 和上界end 分别等于数组元素的下界值0 和上界值n-1。令

middle=(begin+end)/2，并在区间[begin,end]中进行循环查找。在每轮循环中进行测试：

若 end > begin； 表示查找失败，返回n。

若 key > a[middle]； 则查找区间改为 [middle+1, end]， 继续循环。

若 key < a[middle]； 则查找区间为 [begin, middle-1]， 继续循环。

若 key == a[middle]；表示查找完成，返回middle。

由于在每轮循环中进行查找后，使查找区间减半， 因此称此查找方法为二分查找法。

顺序存储的二分查找程序

```
int search(int a[], int n, int key)
{
        int begin=0, end=n-1, middle;
        while(begin <= end)
        {
                middle = (begin + end) / 2;
                if (key > a[middle])
        begin = middle + 1; /* 改区间为[middle+1, end] */
        else if (key < a[middle])
        end = middle - 1; /* 改区间为[begin, middle-1] */
        else /* key == a[middle] */
        return middle; /* 查找成功，返回middle */
}
        return -1; /* 查找失败，返回-1 */
}
```

◆链接存储的顺序查找算法

```
NODE *search(NODE *head, int key)
{
        NODE *node;
        for(node=head; node; node=node->next)
                if(node->num == key)
                        return node;
                /* 查找成功, 返回node */
                        return NULL;
  /* 查找失败, 返回NULL */
}
```

*7.5 联合

在某些特殊应用中，要求将不同的数据对象存放在同一个存储区域。例如，可把一个整型变量、一个字符变量、一个浮点型变量存放在同一个地址开始的内存单元中。在C语言中使用这种方法称为联合（或称共用体）。

联合是一种覆盖技术，即任一时刻只存储其中一种数据，而不是同时存放多种类型的数据。

分配给联合的存储区域大小，要求至少能存储其中最大的一种数据。

联合类型与联合变量的定义

联合类型：

```
union 联合类型名{
        成员说明表
};
```

例如：

```
union  Data {
    int    ival;
    char   chval;
    float  fval;
};
```

联合变量：

```
联合类型  变量名表;

union Data x, y, z;
union  Data{
  int    ival;
  char   chval;
  float  fval;
}x, y, z;
```

说明：定义联合类型 union Data，能存储整型，或字符型，或浮点型的数据。

注意：联合与结构的定义形式非常相似。但它们的含义是不相同的。

联合变量的引用

如上例：x.ival（引用联合变量a中整型变量ival）

x.chval（引用联合变量a中字符变量chval）

x.fval （引用联合变量a中浮点变量fval）

联合的特点：

1. 一个联合可存放多种不同类型的数据，但在每一瞬间只能存放其中一种数据，不是同时存放多种数据。

2. 联合变量中起作用的成员是最后一次存放的成员。

例如： x.ival = 1; x.fval = 2.0; x.chval= '?';

说明：只有 x.chval 是有效的，而 x.ival 及 x.fval 引用其值已经变成不确定的了。

3. 联合变量的开始地址和它的各成员变量的开始地址都是相同。例如：&x，&x.ival，&x.chval 都是同一地址值。

4. 对联合的初始化只能对其成员表中列举的第一个成员置初值。

5. 函数的形参不能是联合类型，函数的结果也不能是联合类型。但指向联合的指针可以作为函数形参，函数也可以返回指向联合的指针。

6. 联合可以嵌套在结构中。

**7.6 位域

用二进制的一位或连续若干位代表不同属性的状态。例如：某台计算机配置的磁盘机中的控制状态寄存器的字长为16位(自右至左，第0位至第15位)。设其中某些位的意义如下：

第15位：置 1 表示数据传送发生错误；

第7位 ：置 1 表示设备已准备好，可传送数据；

第6位 ：置 1 允许响应中断；

第2位 ：置 1 表示读；

第1位 ：置 1 表示写。

实现上述要求可以给对应字中的某些二进位定义一系列表示特征的代码。

例如：

```
#define  ERROR 0100000    /* 对应第 15 位错误标志 */
#define  READY  0200      /* 对应第 7 位准备好 */
#define  IENABLE 0100     /* 对应第 6 位允许中断 */
#define  READ    04       /* 对应第 2 位读 */
#define  WRITE   02       /* 对应第 1 位写 */
```

又如：符号表中，为了区分每个标识符的类别属性，可在描述类别属性字符中的某些二进位作为标志位使用。例如：

```
#define  VARIABLE  01   /* 第 0 位表示变量名 */
#define  FUNCTION  02   /* 第 1 位表示函数名 */
#define  TYPE      04   /* 第 2 位表示类型名 */
#define  EXTERNAL  010  /* 第 3 位表示外部的 */
#define  STATICAL  020  /* 第 4 位表示静态的 */
```

通常称这种表示法为字位标志法，所有的数字必定是 2 的若干次幂。对这些位可以进行移位、屏蔽、求补等运算，就能实现对属性值的测试，存储等。

例1： flg = VARIABLE | EXTERNAL | STATICAL;

表示：将flg置成变量，全局，静态的。

例2： flg &= ~(VARIABLE | EXTERNAL | STATICAL);

表示：将flg置成非变量，非全局，非静态的。

说明：利用C提供的位域机制，能直接定义和存取字中的字段。字段是机器字存储单元中的一串连续的二进位。字段的定义和存取的方法建立在结构基础上。


```
struct  id_atr  {
     unsigned  variable   :1;
     unsigned  function  :1;
     unsigned  type         :1;
     unsigned  external   :1;
     unsigned  statical    :1;
};
   struct  id_atr  flg;
```

说明：flg 包含五个字段，其中每个字段的长度均为1。为特别强调它们是无正负号的量，定义它们是 unsigned 型的。

```
struct  ins_type   {
  unsigned  op  :6;        /* 操作码占前 6 位 */
  unsigned  flg  :2;        /* 特征码占   2 位 */
  unsigned  operand1 :4; /* 第一操作数占 4 位 */
  unsigned  operand2 :4; /* 第二操作数占 4 位 */
}  instruction  ;
```

引用字段的方法（类似于引用结构的成员）：

如：instruction.op表示引用instruction的op字段。另外，字段可以认为是一个无符号整数，像别的整数一样可出现在算术表达式中。

如：

```
flg.variable = flg.external = flg.statical = 1;
```

条件：

```
if (flg.extermal==0 && flg.statical==0) ...;
```

表示：测试flg的相应位是否都为0。

说明：

1. 一个字段只能在同一个整数字中，即限制字段不能跨越整数字的边界。如果剩余的位太少不够下一个字段时，下一个字段将占用下一个整数字。
2. 字段可以不命名，称作无名字段，但无名字段仅用于填充。
3. 使用字段时，要注意具体机器分配字段的方向，有的从左向右，也有的从右向左。
4. 不能对字段施取地址运算(&)。

**7.7 枚举

所谓"枚举"是指将所有的值一一列出，枚举变量只能取列举出来值的范围。

枚举类型的定义：

```
  enum 类型名 {标识符1, 标识符2, ..., 标识符n};
```

例如：

```
 enum weekday {SUN, MON, TUE, WED, THU, FRI, SAT };
 enum weekday  today, yesterday, tomorrow;
```

表示：定义枚举类型 enum weekday

并定义枚举变量 today、yesterday、tomorrow

1. 枚举变量 today、yesterday 和 tomorrow 只能取 SUN 到 SAT 之一的值。

例如：today = SUN;

tomorrow = MON;

yesterday = SAT;

2. 可以在定义枚举类型同时，定义枚举类型变量。

   如: enum { RED, YELLOW, BLUE } color;

3. 枚举类型中的标识符称为枚举常量，每个标识符都表示一个有意义的值。C语言编译按定义时的顺序依次使它们的值为0，1，2，...。如在上面的定义中，SUN 值为0，MON 值为1，...，SAT 值为6。

注意：程序不能对它们赋值。

4.枚举变量的赋值。

例如： today = SUN; 是正确的。

而： SUN = 0 或 SAT = 6 都是错误的。

5. 枚举常量的对应整数也可由程序直接指定。如:

```
enum weekday {SUN=7,MON=1,TUE,WED,THU,FRI,SAT};
```

表示：指定SUN为7，MON为1，后面未指定对应整数的枚举常量所代表的整数，是前一个枚举常量代表的整数加1。所以，TUE为2，...，SAT为6。

6.枚举类型变量与常量或整数可以作关系比较。

例如： if (today == SUN) yesterday = SAT;

【例7.16】输出枚举常量值和变量值。

```
#include  <stdio.h>
void main()
{ int i = 2, j = 1;
  enum  en {RED = 5, GREEN, BLUE} x, y, z;
  x = (enum en)(i-j);
  y = (enum en)(j+i);
  z = (enum en)(j-i);
  printf("Red=%d, Green=%d, x=%d, y=%d, z=%d\n",
              RED, GREEN, x, y, z);
}
```

输出结果为：Red = 5, Green = 6, x = 1, y = 3, z = -1

【例7.17】输入月收入，求出年收入总金额。

```
#include <stdio.h>
void main()
{ enum {Jan = 1, Feb, Mar, Apr, May, Jun,
               Jul, Aug, Sep, Oct, Nov, Dec} month;
  int  yearearn=0, monthearn;
  for ( month=Jan; month<=Dec; month++)
  {  printf ("Enter the monthly earning for: ");
     switch  (month) {
     case Jan : printf("January.\n");     break;
     case Feb : printf("February.\n");    break;
     case Mar : printf("March.\n");       break;
     case Apr : printf("April.\n");             break;
     case May : printf("May.\n");         break;
     case Jun : printf("June.\n");          break;
     case Jul : printf("July.\n");         break;
     case Aug : printf("August.\n");       break;
     case Sep : printf("September.\n");    break;
     case Oct : printf("October.\n");      break;
     case Nov : printf("November.\n");     break;
     case Dec : printf("December.\n");     break;
   }
   scanf ( "%d", &monthearn);
   yearearn += monthearn;
   printf("\n");
  }
  printf ("The total earnings for the year are %d\n",
               yearearn);
}
```