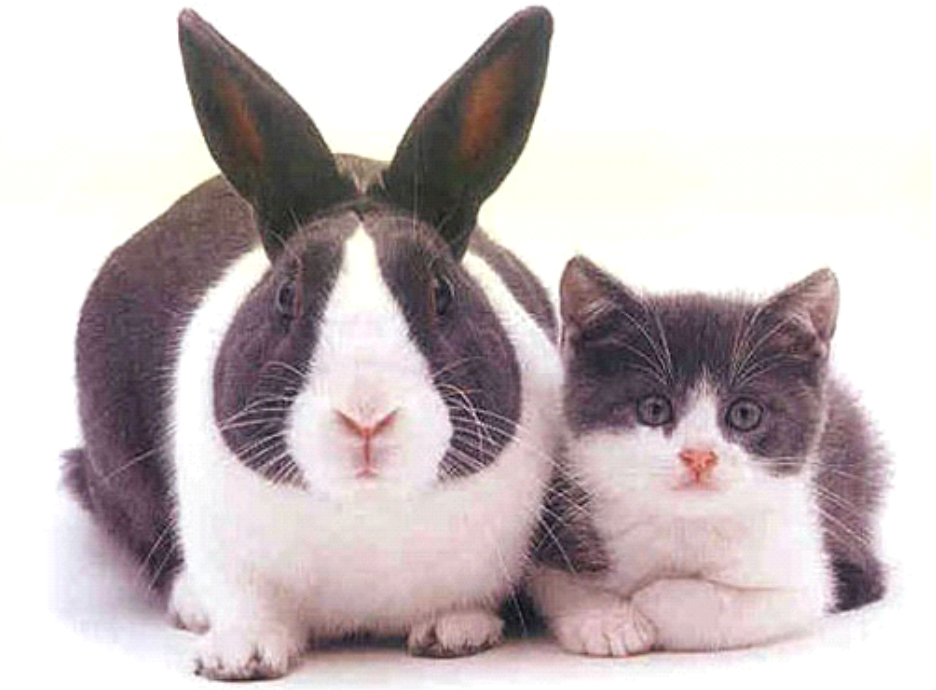
汇编语言数据结构

Data Structures In ASM



王增才 著

书名：汇编语言数据结构
作者：王增才
责任编辑：王增才
封面设计：王增才
定价：10 元
发行方式：电子版，PDF 格式
开本：16 开
出版日期：2010 年 12 月 7 日
字数：110 千字
版本：第 1 版
发行：增才网(http://www.zencai.com)
主题词：汇编语言 | 数据结构 | 计算机 | 编程 | 算法

内容简介

本书简明扼要地介绍了各种典型数据结构及其在汇编语言（编译器 MASM32v10）中的实现代码。主要内容包括：数组、栈与队列、链表、树、哈希表、图等。

本书可作为计算机类专业的本科生的汇编语言数据结构教材，也可以作为使用计算机的广大科技工作者的参考资料。

前言

本书所有示例的源代码均为汇编语言，编译环境：MASM32V10。MASM32V10 下载网址：http://www.masm32.com/

第一章：数组

第二章：栈与队列

第三章：链表

第四章：树

第五章：哈希表

第六章：图

有关更新和勘误可以在下面的网站上找到：

http://www.zencai.com/

源代码下载网址：http://www.zencai.com/Archive/ds.rar

如果由于某种原因造成上面的网址无法访问的话，可通过发邮件给作者来获取更新与勘误。作者的电子邮箱：wzc@zencai.com 或 wangzengcai@126.com

读者在阅读本书之前，应当已经能够看懂汇编语言编写的程序。

特别感谢自考中国网(http://www.chinazk.com)的站长 zwh 老师多年来对我的帮助。

目录

第一章 数组(Array)

本章要点：
数组的定义
一维数组
二维数组
数组的创建，读写操作

1.1 数组(Array)

1.1.1 简介

本章将揭示数组的概念及其汇编实现代码。数组是一种非常常见的数据结构，我相信，学过 C,C++,C#,JAVA 等编程语言的朋友对数组一定熟悉得不能再熟悉了。

1.1.2 定义

数组(Array)，是一系列数的组合，它们有同一个名称并以序号分别来标识其中的每个数（元素,Element）。

一维数组用数学符号可以表示为：

$$\text{Array}=\{a_1 \cdots a_n\}(n \in N_+) \quad (1)$$

Array 标识数组的名称，当然亦可是其他任意合法的名称 。（此处的合法指的是符合某一种具体的编程语言的变量命名规则）；n 标识数组 Array 中的元素个数；a_n 标识数组 Array 中的某一个元素。

二维数组用数学符号可以表示为：

$$Array_{mn}=\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \cdots & \cdots & \cdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}\ (\mathrm{m}\in \mathrm{N}_+, \mathrm{n}\in \mathrm{N}_+)$$

---

(1) N_+ 表示正整数。

在二维数组 $Array_{mn}$ 中，下标 m 表示行数，n 表示列数，其元素的个数为 m×n。超过二维的数组被称为多维数组，例如三维数组，四维数组，五维数组等等。

1.1.3 逻辑结构与存储结构

数据的逻辑结构可以看做是从具体问题抽象出来的数学模型。数据的存储结构可以看做是逻辑结构在内存中的存储模型。此处的存储结构用的存储器指的是虚拟存储器（ Virtual Mmory），汇编语言程序将其视为一个非常大的字节数组。虚拟存储器的每个字节都由一个唯一的数字来标识，称之为地址(address)，所有可能的地址的集合就称为虚拟地址空间(virtual address space)。虚拟地址空间只是一个展现给程序的概念性映像(image)。实际的实现，使用的是随机访问存储器 RAM、磁盘存储、特殊硬件和操作系统软件的结合，来为程序提供一个看上去统一的字节数组。一维数组的逻辑结构是线性的（直线），二维数组的逻辑结构是矩形的（矩阵），三维数组的逻辑结构是立体的（就好比空间向量坐标一样）。由于计算机的内存结构是线性的，因此所有数组的存储结构均为线性模式。

1.1.4 本节习题

难度：★

1. 数组的定义是什么？
2. 用数学符号描述一维数组。
3. 用数学符号描述二维数组。
4. 阐述数据的逻辑结构。
5. 阐述数据的存储结构。

难度：★★

6. 为什么要发明数组？

难度：★★★

7. 用立方体描述三维数组的数学模型。

难度：★★★★

8. 用平行空间或其他模型来描述四维数组，五维数组。

难度：★　　标识习题为基础题，可以从本书中直接找到答案，目的是复习学习内容。

难度：★★　标识习题为发散思维题，本书中没有直接答案，目的是培养创新发散思维。

难度：★★★　　标识习题为挑战思维题，本书中没有直接答案，难度比较大，作者自己可能也解答不了。

难度：★★★★　标识习题为极限思维题，本书中没有直接答案，难度不是一般的大，作者自己肯定解答不了。

1.2 一维数组

数组在使用前必须定义好并分配足够的内存存储空间。除了一些高级语言有动态数组外，一般情况下，数组在定义时就要明确其每个元素的大小与元素的个数。然后即可对数组进行读写操作。当然，在定义数组后，立即对数组进行初始化，或者在定义数组时就对其赋初始值，是个良好的习惯，以避免读取到脏数据。（即无意义的数据）

1.2.1 创建一维数组

声明并初始化一个一维数组的格式：

arrayName　type　　value

说明：

arrayName:数组名

Type:类型

Value:初始值列表(初始值之间用逗号分隔)

举例：

arrayA　dword　1,2,3,4,5

下面是该数组在内存中的图示，假设 arrayA 从地址 0 处开始，注意地址是以 4 字节递增的。

地址(Offset)	**值(Value)**
0000	1
0004	2
0008	3
000C	4

0010	5

除了上面例子中的显式初始化数组每个元素，亦可使用 dup 操作符创建数组，例如：

arrayB　byte 20 dup(0)

源代码

源代码 **ArrayA.asm**(路径:Data Structures In ASM Source Code\chapter 1\ArrayA):

```
.386
.model flat,stdcall
option casemap:none
include windows.inc
include user32.inc
includelib user32.lib
include kernel32.inc
includelib kernel32.lib
.data
arrayA  dword   1,2,3,4,5
szCaption db '消息框!',0
szText db 100 dup(0)
szCharsFormat db 'ArrayA:%d,%d,%d,%d,%d',0
.code
start:
invoke  wsprintf,addr szText, addr szCharsFormat, arrayA, arrayA+4,\
 arrayA+8, arrayA+12, arrayA+16
invoke MessageBox,NULL,offset szText,offset szCaption,MB_OK
invoke ExitProcess,NULL
end start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=ArrayA
OBJS=$(NAME).obj
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
 ml $(ML_FLAG) $<
clean:
    del *.obj
```

1.2.2 计算一维数组的大小

在使用数组时，我们经常需要知道数组的大小(size)。在下面这个例子中，我们创建了一个名为 arrayC 的数组并人工计算 arrayC 数组中的字节数：

arrayC byte 10,20,30,40

arrayCSize=4

如果以后我们要扩展 arrayC 数组，就必须手工的修正 arrayCSize，否则就可能导致程序出现严重的 bug。那可不可以让编译器自动为我们计算 arrayCSize 的值呢？MASM32 用$运算符（当前地址计算器,current location counter）来返回当前程序语句的地址偏移值。下例中，当前地址值($)减掉 arrayD 的地址偏移值就得到了 arrayDSize 的值：

arrayD byte 10,20,30,40

arrayDSize=($-arrayD)

注意，arrayDSize 必须紧跟在 arrayD 之后。

如果数组的每个元素都是 16 位的字(word),以字节(byte)计算的数组总长度必须除以 2 才能得到数组元素的个数。例如下例：

arrayE word 1000h,2000h,3000h,4000h

ArrayESize=($-arrayE)/2

与此类似，双字（doubleword）数组每个元素是 4 字节的，因此数组的总长度必须除以 4 才能得到数组元素的个数。如下例：

arrayF word 1000000h,2000000h,3000000h,4000000h

ArrayFSize=($-arrayF)/4

源代码

源代码 **ArrayF.asm**(路径:Data Structures In ASM Source Code\chapter 1\ArrayF):

```
.386
.model flat,stdcall
option casemap:none
include windows.inc
include user32.inc
includelib user32.lib
include kernel32.inc
includelib kernel32.lib
.data
arrayF  dword   1000000h,2000000h,3000000h,4000000h
arrayFSize=($-arrayF)/4
szCaption db '消息框!',0
szText db 100 dup(0)
szCharsFormat db 'ArrayF:%d,%d,%d,%d,Size:%d',0
.code
start:
invoke  wsprintf,addr szText, addr szCharsFormat, arrayF, arrayF+4,\
```

```
 arrayF+8, arrayF+12,addr arrayFSize
invoke MessageBox,NULL,offset szText,offset szCaption,MB_OK
invoke ExitProcess,NULL
end start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=ArrayF
OBJS=$(NAME).obj
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
 ml $(ML_FLAG) $<
clean:
    del *.obj
```

还有一个简单的办法来获取数组中的元素个数，即操作符 lengthof。以下面的数据为例：

```
.data
byte1    byte          10,20,30
array1   word      30 dup(?),0,0
array2   dword     1,2,3,4
```

下表列出了每个 lengthof 表达式的返回值：

表达式	值
lengthof byte1	3
lengthof array1	30+2
lengthof array2	4

注意：如果声明了一个跨多行的数组，lengthof 只把第一行的数据作为数组的组成部分。

源代码

源代码 **ArrayL.asm**(路径:Data Structures In ASM Source Code\chapter 1\ArrayL):

```
.386
.model flat,stdcall
option casemap:none
include windows.inc
include user32.inc
includelib user32.lib
include kernel32.inc
includelib kernel32.lib
.data
byte1   byte        10,20,30
array1  word    30 dup(?),0,0
```

```
array2  dword   1,2,3,4
szCaption db '消息框!',0
szText db 100 dup(0)
szCharsFormat db 'Array Length:%d',0
.code
start:
mov eax,lengthof    byte1
invoke  wsprintf,addr szText, addr szCharsFormat, eax
invoke MessageBox,NULL,offset szText,offset szCaption,MB_OK
mov eax,lengthof    array1
invoke  wsprintf,addr szText, addr szCharsFormat, eax
invoke MessageBox,NULL,offset szText,offset szCaption,MB_OK
mov eax,lengthof    array2
invoke  wsprintf,addr szText, addr szCharsFormat, eax
invoke MessageBox,NULL,offset szText,offset szCaption,MB_OK


invoke ExitProcess,NULL
end start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=ArrayL
OBJS=$(NAME).obj
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
 ml $(ML_FLAG) $<
clean:
    del *.obj
```

1.2.3 读写一维数组元素

读写一维数组元素，即读取数组某个元素的值与给某个数组元素赋值。对数组的读写操作是一种基本操作。在数组名称后面加上一个偏移值，即可读写某个元素。此处的偏移值必须是数组的数据类型的整数倍。我们以数组 arrayG 为例：

```
arrayG    byte      10h,20h,30h,40h,50h
```

如果 mov 指令使用 arrayG 作为源操作数，就可以将数组的第一个元素赋值给 AL：

```
mov       AL, arrayG                ;AL=10h
```

我们 可以在 arrayG 的偏移地址（offset）上直接加 1 来读取数组中的第二个元素：

```
mov     AL, [arrayG+1]              ;AL=20h
```

由此类推，通过加 2 可以读取第三个元素：

```
mov     AL, [arrayG+2]              ;AL=30h
```

像 arrayG+1 这样的表达式被称之为有效地址(effective address)，有效地址通过在变量偏移地址之后加上一个常数来标识。当我们将一个有效地址用方括号括起来时，表示用它来获取它的目标内存地址的内容。在 MASM32 中，这对方括号并不是必须的，指令也可以写做：

```
mov     AL, arrayG+1                ;AL=20h
```

MASM32 对有效地址没有内建的范围检查，因此我们必须小心地对数组进行范围检查(Range Checking)或范围控制，以防越界读写元素。一旦越界读写数组元素，将会导致不可预测的灾难性的逻辑错误，并且这个逻辑错误很难发现。所以我们在使用数组时必须格外小心。例如下面这个语句就是越界读取了数组元素：

```
mov     AL,[arrayG+20]          ;AL=??
```

此处的 AL 等于多少，也许只有上帝知道。

如果 mov 指令使用 arrayG 作为目的操作数，就可以将一个值写入数组的第一个元素：

```
mov      arrayG, 1                  ;arrayG=1
```

我们 可以在 arrayG 的偏移地址（offset）上直接加 1 来给数组中的第二个元素赋值：

```
mov     [arrayG+1],20       ;[arrayG+1]=2
```

由此类推，通过加 2 可以写入第三个元素：

```
mov      [arrayG+2],3           ;[arrayG+2]=3
```

字(word)和双字(Doubleword)数组 如果我们使用一个数据类型为 16 位字（16-bit ,word）的数组，请务必记牢，每个数组元素与前一个数组元素的偏移相差两个字节(8-bit ,byte)。如下例所示：

```
.data
arrayH    word      100h,200h,300h
.code
mov       ax,arrayH           ;AX=100h
mov       ax,[arrayH+2]       ;AX=200h
```

类似地，双字数组的每个元素与前一个元素的偏移相距 4 个字节(32-bit ,4 byte)

```
.data
arrayK    dword     10000h,20000h
.code
mov       eax,arrayK          ;eax=10000h
mov       eax,[arrayK+4]      ;eax=20000h
```

源代码

源代码 **ArrayK.asm**(路径:Data Structures In ASM Source Code\chapter 1\ArrayK):

```
.386
.model flat,stdcall
option casemap:none
include windows.inc
include user32.inc
includelib user32.lib
```

```
include kernel32.inc
includelib kernel32.lib
.data
arrayK  dword   10000h,20000h
szCaption db '消息框!',0
szText db 100 dup(0)
szCharsFormat db 'ArrayK:%d,%d',0
.code
start:
invoke  wsprintf,addr szText, addr szCharsFormat, arrayK, arrayK+4
invoke MessageBox,NULL,offset szText,offset szCaption,MB_OK
mov arrayK,1
mov arrayK+4,2
invoke  wsprintf,addr szText, addr szCharsFormat, arrayK, arrayK+4
invoke MessageBox,NULL,offset szText,offset szCaption,MB_OK
invoke ExitProcess,NULL
end start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=ArrayK
OBJS=$(NAME).obj
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
 ml $(ML_FLAG) $<
clean:
    del *.obj
```

1.2.4 间接操作数(indirect addressing)

也许你已经注意到在前面的源代码中，读写数组时非常笨拙，代码写得一点都不优美。那有没有更好的读写数组元素的办法呢？我们可以通过间接寻址的方法来优雅地读写数组元素。间接寻址，即用寄存器作为指针并操纵寄存器的值，存放地址的寄存器称为间接操作数（indirect operand）。

间接操作数可以是任何用方括号括起来的通用寄存器，例如eax,ebx,ecx,edx,esi,edi,ebp,esp。寄存器存放的是数据的偏移。一般都是用 esi 来存储数组的偏移。例如：

```
.data
abc     byte    10h
```

```
.code
mov     esi, offset     abc
```

在保护模式下，如果有效地址指向程序数据段之外的区域，CPU 就会产生一个通用保护错误（general protection fault）。即使指令并不修改内存，这种情况也可能发生。例如，如果 esi 未初始化，下面的指令就可能产生通用保护错误：

```
mov     ax, [esi]
```

避免这类错误的最好方法是认真地初始化作为间接操作数的寄存器。

在操作数的大小并不是很明确时，可以使用 ptr 操作符来明确地表示操作数的尺寸：

```
inc   byte     ptr   [esi]
```

跟数组下标类似，间接操作数可以指向数组的不同元素。例如 arrayM 由三个字节元素构成，我们可以增加 esi 的值逐个读取其元素：

```
.data
arrayB    byte      10h,20h,30h
.code
mov       esi,offset      arrayB
mov       al,[esi]              ;al=10h
inc       esi
mov       al,[esi]        ;al=20h
inc       esi
mov       al,[esi]        ;al=30h
```

如果使用 16 位字的整数数组，就需要给 esi 加 2：

```
.data
arrayN    word      1000h,2000h,3000h
.code
mov       esi,offset      arrayN
mov       ax,[esi]                ;ax=1000h
add       esi,2
mov       ax,[esi]                ;ax=2000h
add       esi,2
mov       ax,[esi]                ;ax=3000h
```

如果是 32 位双字数组，则每次偏移必须加 4 才能正确寻址数组的后续元素，下例累加数组元素：

```
.data
arrayP    dword     1,2,3
.code
mov       esi,offset arrayP
mov       eax,[esi]           ;第 1 个元素   eax=1
add       esi,4
add       eax,[esi]           ;加上第 2 个元素   eax=3
add       esi,4
add       eax,[esi]           ;加上第 3 个元素   eax=6
```

源代码

源代码 **ArrayP.asm**(路径:Data Structures In ASM Source Code\chapter 1\ArrayP):

```
.386
.model flat,stdcall
option casemap:none
include windows.inc
include user32.inc
includelib user32.lib
include kernel32.inc
includelib kernel32.lib
.data
arrayP  dword   1,2,3
szCaption db '消息框!',0
szText db 100 dup(0)
szCharsFormat db 'ArrayP Sum:%d',0
.code
start:
mov esi,offset arrayP
mov     eax,[esi]           ;第 1 个元素  eax=1
add     esi,4
add     eax,[esi]           ;加上第 2 个元素    eax=3
add     esi,4
add     eax,[esi]           ;加上第 3 个元素    eax=6
invoke  wsprintf,addr szText, addr szCharsFormat, eax
invoke MessageBox,NULL,offset szText,offset szCaption,MB_OK
invoke ExitProcess,NULL
end start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=ArrayP
OBJS=$(NAME).obj
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
 ml $(ML_FLAG) $<
clean:
    del *.obj
```

1.2.5 利用循环读取一维数组元素

上一小节的源代码 ArrayP.asm 有许多重复操作的代码,例如 mov，add 操作符。将这些重复的代码整合到循环里，可以有效地缩短程序的长度。

loop 指令提供了一种将程序块重复执行特定次数的简单方法。在这里，ecx 被自动用做计数器，在每次循环之后减 1，格式如下：

loop　　标号

loop 指令分两步执行：首先，ecx 减 1，接着与 0(zero)相比较，如果 ecx 不等于 0 则跳转到标号处；如果 ecx 等于 0 则不跳转，直接执行紧跟在 loop 后面的指令。

常见的编程错误是在循环开始之前随意将 ecx 初始化为 0，这种情形下，loop 指令执行后，ecx 减 1 的结果是 FFFFFFFFh，结果是循环将重复 4,294,967,296 次！

循环的目的地址与当前地址只能在相距-128 到+127 字节的范围之内。机器指令的平均大小是 3 字节左右，因此一个循环平均最多只能包含大约 42 条指令。

下面我们利用 loop 指令对上一小节中的源代码 ArrayP.asm 进行改造，步骤如下：

1. 将指针寄存器指向数组的起始偏移地址，用这个寄存器作为变址操作数。
2. 将 ecx 设置为数组中元素的数目。
3. 将一个寄存器清零用于保存累加和
4. 创建一个标号标识循环的开始。
5. 在循环体中，用间接寻址方式将数组的每个元素同用于存放累加和的寄存器相加。
6. 指针寄存器指向下一个数组元素。
7. 使用 loop 指令重复执行由开始标号标明的循环体。

（1~3 步的顺序不限）

源代码

源代码 **ArraySum.asm**(路径:Data Structures In ASM Source Code\chapter 1\ArraySum):

```
.386
.model flat,stdcall
option casemap:none
include windows.inc
include user32.inc
includelib user32.lib
include kernel32.inc
includelib kernel32.lib
.data
arraySum    dword   1,2,3
szCaption db '消息框!',0
szText db 100 dup(0)
szCharsFormat db 'ArraySum Sum:%d',0
.code
start:
mov esi,offset arraySum
```

```
mov ecx,lengthof    arraySum
mov     eax,0
L1:
add     eax,[esi]
add     esi,type arraySum   ;指向下一个元素
loop        L1              ;跳转到 L1,直到 ecx=0 时，才停止跳转
invoke  wsprintf,addr szText, addr szCharsFormat, eax
invoke MessageBox,NULL,offset szText,offset szCaption,MB_OK
invoke ExitProcess,NULL
end start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=ArraySum
OBJS=$(NAME).obj
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
 ml $(ML_FLAG) $<
clean:
    del *.obj
```

1.2.6 本节习题

难度：★

1. 创建一个一维数组
2. 计算一维数组的大小。
3. 读写一维数组元素。
4. 利用间接操作数读写一维数组。
5. 利用 loop 指令读写一维数组。

难度：★★

6. 读数组时，创建一个死循环。

难度：★★★

7. 创建一个循环体指令多于 42 条指令的循环，看看编译器会怎样报错。

难度：★★★★

8. 创建一个二维数组。

1.3 二维数组（Two-Dimensional Array）

基址变址与相对基址变址常常用来读写二维数组。

1.3.1 基址变址(base-index)操作数

基址变址操作数将两个寄存器的值相加（称为基址和变址）来产生偏移地址，操作数中可以使用任意两个 32 位通用寄存器。例如：

```
.data
array       word       1,2,3
.code
mov         ebx, offset        array
mov         esi,   2
mov         ax, [ebx+esi]          ;ax=2
mov         edi, offset        array
mov         ecx, 4
mov         ax, [edi+ecx]          ;ax=3
mov         ebp, offset        array
mov         esi,0
mov         ax,[ebp+esi]           ;ax=1
```

源代码

源代码 TDA**.asm**(路径:Data Structures In ASM Source Code\chapter 1\TDA):

```
.386
.model      flat,stdcall
option        casemap:none
include     windows.inc
include     user32.inc
includelib    user32.lib
include     kernel32.inc
includelib    kernel32.lib
.data
array       word        1,2,3
szCaption         db '  消息框!',0
szText       db  100  dup(0)
szCharsFormat         db    'Array:%d',0
.code
start:
mov ebx, offset    array
mov esi, 2
```

```
mov ax, [ebx+esi]          ;ax=2
movzx     eax,ax
invoke    wsprintf,addr   szText, addr   szCharsFormat, eax
invoke    MessageBox,NULL,offset   szText,offset   szCaption,MB_OK
mov edi, offset   array
mov ecx,4
mov ax,[edi+ecx]           ;ax=3
movzx     eax,ax
invoke    wsprintf,addr   szText, addr   szCharsFormat, eax
invoke    MessageBox,NULL,offset   szText,offset   szCaption,MB_OK
mov ebp, offset   array
mov esi,0
mov ax, [ebp+esi]          ;ax=1
movzx     eax,ax
invoke    wsprintf,addr   szText, addr   szCharsFormat, eax
invoke    MessageBox,NULL,offset   szText,offset   szCaption,MB_OK
invoke     ExitProcess,NULL
end  start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=TDA
OBJS=$(NAME).obj
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
 ml $(ML_FLAG) $<
clean:
    del *.obj
```

基址变址操作数在访问二维表格的时候特别有用，通常使用基址寄存器存放行偏移，变址寄存器存放列偏移。下面我们创建一个 3 行 5 列表格的数据定义以说明这种寻址方式：

```
tableB    byte      11,12,13,14,15
          byte      21,22,23,24,25
          byte      31,32,33,34,35
NumCols=5
```

在内存中，这张表仅仅是一系列连续的字节流，就跟一维数组一样，但是我们可以将其想像成一个具有 3 个逻辑行和 5 个逻辑列的二维数组。将数组中的每一行声明在单独的代码行上并不是必需的，但这样有助于清楚地表达表格的结构。数组的物理存储是按行顺序存储的，也就是说第一行的最后一个字节接着下一行的第一个字节，以此类推。我们以行和列的

坐标来定位表格中的一个特殊元素，坐标值从 0,0 开始，即第一行第一列的坐标为 0,0。假设 CurrentRow 表示当前行（0 行表示第一行），CurrentColumn 表示当前列（0 列表示第一列），ColumnNumber 表示每行的列数，ArraySize 表示元素的大小，则计算某一个元素的坐标的公式为：

CurrentRow*ColumnNumber*ArraySize+CurrentColumn*ArraySize

像其他间接寻址方式一样，如果有效地址超出了程序的数据区域，就会产生通用保护异常。

下面的程序计算表格例子中第二行的总和：

```
.data
CurrentRow    dword    1
ColumnNumber    dword    5
tableB    dword    11,12,13,14,15
          dword        21,22,23,24,25
          dword        31,32,33,34,35
ArraySize      dword    4
.code
mov       ecx,ColumnNumber
mov       ebx,offset tableB
mov       eax,CurrentRow
mul       ecx
mul       ArraySize
add       ebx,eax
mov       esi,0
mov       eax,0          ;eax 存求和的值
mov       edx,0          ;edx 存单个元素的值
L1:mov    edx,[ebx+esi]  ;获得某个元素的值
     add  eax,edx        ;累加
     add  esi,ArraySize       ;指向下一个元素
     loop      L1
```

源代码

源代码 tableB**.asm**(路径:Data Structures In ASM Source Code\chapter 1\tableB):

```
.386
.model    flat,stdcall
option    casemap:none
include   windows.inc
include   user32.inc
includelib        user32.lib
include     kernel32.inc
includelib              kernel32.lib
.data
CurrentRow    dword    1
```

```
ColumnNumber        dword       5
ArraySize           dword       4
tableB      dword       11,12,13,14,15
            dword       21,22,23,24,25
            dword       31,32,33,34,35
szCaption   db  '消息框!',0
szText      db    100       dup(0)
szCharsFormat       db      '数组第二行的和:%d',0
.code
start:
mov         ebx,offset      tableB
mov         eax,CurrentRow
mul         ColumnNumber
mul         ArraySize
add         ebx,eax
mov         esi,0
mov         eax,0           ;eax存求和的值
mov         edx,0           ;edx存单个元素的值
mov         ecx,ColumnNumber
L1:mov      edx,[ebx+esi]   ;获得某个元素的值
      add   eax,edx         ;累加
      add   esi,4           ;指向下一个元素


      loop        L1

invoke      wsprintf,addr   szText, addr    szCharsFormat, eax
invoke      MessageBox,NULL,offset      szText,offset    szCaption,MB_OK

invoke      ExitProcess,NULL
end  start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=tableB
OBJS=$(NAME).obj
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
 ml $(ML_FLAG) $<
```

```
clean:
    del *.obj
```

1.3.2 相对基址变址操作数(Base-Index Displacement operand)

相对基址变址操作数将偏移、基址寄存器和变址寄存器组合起来产生一个有效地址，下面是该操作数最常见的两种格式：

[base+index+displacement]

displacement[base+index]

偏移(Displacement)可以是变量的名称或者常量表达式，基址和变址可以使用任意的 32 位寄存器。

下面的程序计算表格例子中第二行的总和：

```
.data
CurrentRow    dword    1
ColumnNumber    dword    5
tableC    dword    11,12,13,14,15
          dword        21,22,23,24,25
          dword        31,32,33,34,35
ArraySize      dword    4
.code
mov       ecx,ColumnNumber
mov       eax,CurrentRow
mul       ecx
mul       ArraySize
mov       ebx,eax
mov       esi,0
mov       eax,0          ;eax 存求和的值
mov       edx,0          ;edx 存单个元素的值
L1:
     add  eax, tableC[ebx+esi]    ;累加
     add  esi,ArraySize       ;指向下一个元素
     loop      L1
```

源代码

源代码 tableC**.asm**(路径:Data Structures In ASM Source Code\chapter 1\tableC):

```
.386
.model    flat,stdcall
option    casemap:none
```

```
include     windows.inc
include     user32.inc
includelib          user32.lib
include       kernel32.inc
includelib                kernel32.lib
.data
CurrentRow      dword       1
ColumnNumber        dword       5
ArraySize         dword       4
tableC      dword       11,12,13,14,15
            dword       21,22,23,24,25
            dword       31,32,33,34,35
szCaption   db  '消息框!',0
szText        db    100         dup(0)
szCharsFormat         db      '数组第二行的和:%d',0
.code
start:
mov         eax,CurrentRow
mul         ColumnNumber
mul         ArraySize
mov         ebx,eax
mov         esi,0
mov         eax,0               ;eax存求和的值
mov         edx,0               ;edx存单个元素的值
mov         ecx,ColumnNumber
L1:
      add   eax,tableC[ebx+esi]         ;累加
      add   esi,4               ;指向下一个元素


      loop          L1

invoke      wsprintf,addr     szText, addr      szCharsFormat, eax
invoke      MessageBox,NULL,offset        szText,offset     szCaption,MB_OK

invoke      ExitProcess,NULL
end   start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=tableC
OBJS=$(NAME).obj
```

```
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
 ml $(ML_FLAG) $<
clean:
    del *.obj
```

1.4 排序（Sort）

对数组进行排序是一种常规动作，排序的方法有千千万万种，每一种都有自己的优缺点，目前为止，还没有人发现一种超级排序的方法。Donald E. Knuth 在《The Art Of Computer Programming》第三卷中介绍了大约 25 种排序方法。本文只讨论了几种典型的排序方法。

1.4.1 冒泡排序(Bubble Sort)

冒泡排序(Bubble Sort)的基本思想：从位置 0 和 1 开始比较每对数组元素值，如果两个值的顺序不对则进行交换。这种方法之所以称为冒泡，是因为大的元素“上浮”到它们的适当的位置。下图显示了对数组 ArrayA={3,1,7,5,2}进行的一次完整的比较过程：

原始状态	3,1,7,5,2
第一对比较后	1,3,7,5,2
第二对比较后	1,3,7,5,2
第三对比较后	1,3,5,7,2
第四对比较后	1,3,5,2,7

在第一遍比较完成后，数组也许并未被排好序，因此必须另一轮的比较，假设 n 为数组的元素个数，则最多进行 n-1 遍比较后，就能保证数组是按顺序排列好了的。冒泡排序对小数组工作得很好，但对大数组就非常低效了。它的时间复杂度为 O(n^2)。

源代码

源代码 bubbleSort**.asm**(路径:Data Structures In ASM Source Code\chapter 1\bubbleSort):

```
.386
.model    flat,stdcall
option    casemap:none
include   windows.inc
include   user32.inc
includelib        user32.lib
include    kernel32.inc
```

```
includelib              kernel32.lib
.data
ArrayA      dword       3,1,7,5,2
szCaption   db  '消息框!',0
szText      db    100       dup(0)
szCharsFormat       db     '冒泡排序后的数组:%d,%d,%d,%d,%d',0
.code
;-------------------------------------------------------------
BubbleSort   proc   uses   eax   ecx   esi,
      pArray:ptr   dword,   ;指向数组的指针
      Count:dword           ;数组的大小
;按升序将一个32位的有符号整数数组排序
;使用冒泡排序法
;参数:pArray:ptr dword,Count:dword
;返回：无
mov   ecx,Count
dec   ecx
L1:push   ecx
      mov   esi,pArray
      L2:mov   eax,[esi]
            cmp   [esi+4],eax          ;比较相邻的两个元素
            jge   L3                   ;如果[esi]<=[esi+4]，则不交换两个元素
            xchg   eax,[esi+4]
            mov   [esi],eax
            L3:add   esi,4
      loop   L2                 ;内循环
      pop   ecx
loop   L1
L4:ret
BubbleSort   endp

start:
invoke   BubbleSort,addr   ArrayA,5
invoke     wsprintf,addr    szText, addr     szCharsFormat, ArrayA,ArrayA+4,ArrayA+8,\
      ArrayA+12,ArrayA+16
invoke     MessageBox,NULL,offset       szText,offset     szCaption,MB_OK

invoke     ExitProcess,NULL
end  start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=bubbleSort
OBJS=$(NAME).obj
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
 ml $(ML_FLAG) $<
clean:
    del *.obj
```

假设一种极端情况，数组 ArrayA 在排序前就已经是有序数组了，按上面的算法，程序依然要轮循 n-1 次（n 表示数组的元素个数）。我们可以对其进行改进，以提高执行效率。我们引入一个局部变量@exchange，在每轮循环前，先将其置为 0，若本轮循环过程中发生了交换，则将其置为 1，每轮循环结束时都检查@exchange，若@exchange=0，则终止循环。

源代码

源代码 bubbleSort**.asm**(路径:Data Structures In ASM Source Code\chapter 1\bubbleSort2):

```
.386
.model    flat,stdcall
option    casemap:none
include   windows.inc
include   user32.inc
includelib        user32.lib
include     kernel32.inc
includelib              kernel32.lib
.data
ArrayA    dword     3,1,7,5,2
szCaption  db  '消息框!',0
szText      db   100      dup(0)
szCharsFormat       db    '冒泡排序后的数组:%d,%d,%d,%d,%d',0
.code
;-------------------------------------------------------------
BubbleSort   proc   uses   eax   ecx   esi,
      pArray:ptr   dword,   ;指向数组的指针
      Count:dword           ;数组的大小
;按升序将一个32位的有符号整数数组排序
;使用冒泡排序法
;参数:pArray:ptr dword,Count:dword
;返回：无
local       @exchange:dword
mov   ecx,Count
```

```
dec   ecx
mov       @exchange,0
L1:push   ecx
      mov   esi,pArray
      mov       @exchange,0
      L2:mov   eax,[esi]
            cmp   [esi+4],eax          ;比较相邻的两个元素
            jge   L3                ;如果[esi]<=[esi+4]，则不交换两个元素
            xchg   eax,[esi+4]
            mov   [esi],eax
            mov       @exchange,1
            L3:add   esi,4
      loop   L2              ;内循环
      cmp       @exchange,0
      je    L4
      pop   ecx
loop   L1
L4:ret
BubbleSort   endp

start:
invoke   BubbleSort,addr   ArrayA,5
invoke     wsprintf,addr    szText, addr     szCharsFormat, ArrayA,ArrayA+4,ArrayA+8,\
      ArrayA+12,ArrayA+16
invoke     MessageBox,NULL,offset       szText,offset     szCaption,MB_OK

invoke     ExitProcess,NULL
end  start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=bubbleSort2
OBJS=$(NAME).obj
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
 ml $(ML_FLAG) $<
clean:
    del *.obj
```

1.4.2 快速排序(Quick Sort)

快速排序是 C.R.A.Hoare 于 1962 年提出的一种划分交换排序。它的基本思想为：将问题分解成若干个大小基本相等的子问题；递归地解各个子问题；最后将这些子问题的解组合成问题的解。假设待排序的 n 个元素存储在数组 ArrauA[0,...,n-1]中。快速排序的算法描述如下：

从 ArrayA[0,...n-1]中选取一个元素作为中心(pivot，有的论文或书将 pivot 翻译为基准或枢轴，笔者个人认为翻译为中心，更容易理解些，Hoare 的原论文将中心称为 bound)；以 pivot 为界，将数组划分为左右两个部分，左部分的元素为 ArrayA[0,...,pivot-1]，右部分的元素为 ArrayA[pivot+1,...n-1]；使得左部分的数组元素的值均小于等于 ArrayA[pivot]，右部分的数组元素的值均大于等于 ArrayA[pivot]，而 pivot 位于正确的位置，ArrayA[pivot]无须参加后续的排序；对左、右部分进行递归快速排序。

在快速排序算法中，有两个关键点：

（1）怎样选择中心元素？

（2）在确定中心元素后，怎样划分左、右部分？

由于在快速排序算法中，一个主要假设是将输入分成几乎大小相同的部分，因此所选择的 pivot 最好是数组中的中间值。理论上，好的选择方法可以找到中间值做 pivot，但由于开销过大，在实际中，几乎得不到实用。

有三种基本的方法可以选择 pivot。第一种是从数组 ArrayA 的固定位置选择 pivot 元素，例如第一个元素作为 pivot。这种选择方法，在数组元素随机时，效果较好；在数组有序时，每次递归过程中，数组元素将划分得极不均匀，算法性能会很差。第二种方法是计算数组 ArrayA 中的中间值作为中心。通常使用的方法是计算 ArrayA 中的第一个元素，中间的元素和最后一个元素的中间值作为中心。实践表明，采用三元素取中间值的规则可大大改善快速排序在最坏情况下的性能。第三种方法是采用随机数选择中心。在这种情况下，可以严格的证明递归的每一步以非常大的概率导致均匀划分，与初始数组元素的分布无关。

划分左右部分，其实就是通过交换左右部分的元素，使得左部分的元素均小于中心，右部分的元素均大于中心。

让我们来看一个基于第一种选择中心的方法简单的示例：

源代码

源代码 QuickSort**.asm**(路径:Data Structures In ASM Source Code\chapter 1\QuickSort):

```
.386
.model    flat,stdcall
option    casemap:none
include   windows.inc
include   user32.inc
includelib        user32.lib
include    kernel32.inc
includelib              kernel32.lib
.data
ArrayA    dword     3,1,7,5,2
```

```
szCaption   db  '消息框!',0
szText      db   100      dup(0)
szCharsFormat       db     '快速排序后的数组:%d,%d,%d,%d,%d',0
.code
;-------------------------------------------------------------

QuickSort       PROC   USES EAX EBX ECX EDX ESI EDI array:PTR   DWORD, lowitem:DWORD,
highitem:DWORD
        LOCAL     pivot:DWORD ;pivot  中心值
        MOV       EBX, array ;EBX  数组
        MOV       EDX, lowitem ;EDX  索引偏移
        MOV       ESI, lowitem ;ESI  左向偏移
        MOV       EDI, highitem ;EDI  右向偏移
        CMP       ESI, EDI
        JGE       TagEnd

        MOV       EAX, DWORD   PTR   [EBX+EDX*4]
        MOV       pivot, EAX
        MOV       EAX, EDI
        SUB       EAX, ESI
        MOV       ECX, EAX
        INC       EDX
    Tag1:
        MOV       EAX, pivot
        CMP       EAX, DWORD   PTR   [EBX+EDX*4]
        jge       Tag2
        JMP       Tag3
    Tag2:
        MOV       EAX, DWORD   PTR   [EBX+EDX*4]
        XCHG      EAX, DWORD   PTR   [EBX+ESI*4+4]
        MOV       DWORD   PTR   [EBX+EDX*4], EAX

        MOV       EAX, DWORD   PTR   [EBX+ESI*4]
        XCHG      EAX, DWORD   PTR   [EBX+ESI*4+4]
        MOV       DWORD       PTR             [EBX+ESI*4], EAX
        INC       ESI
    Tag3:
        INC       EDX
        LOOP      Tag1

        DEC       ESI
        PUSH      ESI
        INVOKE      QuickSort, array, lowitem, ESI
        POP       ESI
```

```
        INC     ESI
        INC     ESI
        INVOKE      QuickSort, array, ESI, highitem
    TagEnd:
        RET
QuickSort       ENDP
start:

invoke QuickSort,addr ArrayA,0,4
invoke      wsprintf,addr    szText, addr     szCharsFormat, ArrayA,ArrayA+4,ArrayA+8,\
      ArrayA+12,ArrayA+16
invoke      MessageBox,NULL,offset       szText,offset     szCaption,MB_OK

invoke      ExitProcess,NULL
end  start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=QuickSort
OBJS=$(NAME).obj
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
 ml $(ML_FLAG) $<
clean:
    del *.obj
```

1.4.3 直接插入排序（Straight Insertion Sort）

插入排序(Insertion Sort)的基本思想为：每次将一个待排序的元素，按其大小插入到前面已经排好序的数组元素中的适当位置，直到全部元素插入完毕。

直接插入排序就跟我们打扑克牌一样，摸来的第一张牌无须整理，此后每次从桌上的牌（无序）中摸最上面的一张牌并插入到手上的牌（有序）中的正确位置。为了找到这个正确的位置，我们可以从左向右将摸来的牌与手中插好的牌一一比较。

一种简单的直接插入方法为：首先在当前有序区 ArrayA[1..i-1]中查找 ArrayA[i]的正确插入位置 k(1≤k≤i-1)；然后将 ArrayA[k．．i-1]中的记录均后移一个位置，腾出 k 位置上的空间插入 Array[i]。若 ArrayA[i]的关键字大于等于 ArrayA[1．．i-1]中所有记录的关键字，则 ArrayA[i]就是插入原位置。

改进的方法：一种查找比较操作和记录移动操作交替地进行的方法。

具体做法：

将待插入记录 ArrayA[i]的关键字从右向左依次与有序区中记录 ArrayA[j](j=i-1，i-2，…，1)的关键字进行比较：

① 若 ArrayA[j]的关键字大于 ArrayA[i]的关键字，则将 ArrayA[j]后移一个位置；

②若 ArrayA[j]的关键字小于或等于 Array[i]的关键字，则查找过程结束，j+1 即为 Array[i]的插入位置。

关键字比 ArrayA[i]的关键字大的记录均已后移，所以 j+1 的位置已经腾空，只要将 ArrayA[i]直接插入此位置即可完成一趟直接插入排序。

源代码

源代码 SIS**.asm**(路径:Data Structures In ASM Source Code\chapter 1\SIS):

```
.
.386
.model      flat,stdcall
option      casemap:none
include     windows.inc
include     user32.inc
includelib          user32.lib
include       kernel32.inc
includelib                kernel32.lib
.data
ArrayA      dword       3,1,7,5,2
szCaption   db   '消息框!',0
szText      db    100         dup(0)
szCharsFormat         db      '直接插入排序后的数组:%d,%d,%d,%d,%d',0
.code
;-------------------------------------------------------------
InsertSort proc uses EAX EBX ECX EDX ESI EDI pArray:ptr dword,Count:dword
local       @i:dword
local       @j:dword
local       @sentinel:dword
mov         @i,1
mov  ebx,Count
mov         esi,pArray

L1:
cmp  @i,ebx
jge    L3
       mov  eax,[esi+@i]
       cmp  eax,[esi+@i-1]
       jge    L4
       mov  eax,[esi+@i]
```

```
        mov  @sentinel,eax
        mov  eax,@i
        dec   eax
        mov  @j,eax
        L5:
        mov  eax,[esi+@j]
        mov  [esi+@j+1],eax
        dec   @j
        mov  eax,[esi+@j]
        cmp  @sentinel,eax
        jl      L5
        mov  eax,@sentinel
        mov  [esi+@j+1],eax

L4:
inc @i
jmp   L1


L3:
ret

InsertSort              endp

start:
invoke      InsertSort,addr              ArrayA,5
invoke      wsprintf,addr    szText, addr     szCharsFormat, ArrayA,ArrayA+4,ArrayA+8,\
      ArrayA+12,ArrayA+16
invoke      MessageBox,NULL,offset        szText,offset     szCaption,MB_OK

invoke      ExitProcess,NULL
end   start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=SIS
OBJS=$(NAME).obj
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
```

```
 ml $(ML_FLAG) $<
clean:
    del *.obj
```

1.4.4 希尔排序(Shell Sort)

希尔排序(Shell Sort)是 D.L.Shell 于 1959 年提出的。其基本思想为：假设数组元素的个数为 n，先取一个小于 n 的整数 d_1 作为第一个增量，把数组中的元素分成 d_1 个组，所有距离为 d_1 的倍数的数组元素放在同一个组中，在各组内进行直接插入排序；然后，取第二个增量 d_2 ($d_2 < d_1$)，重复上述的分组和排序，直至所取的增量 d_t=1($d_t < d_{t-1} < \cdots < d_2 < d_1$)，即所有的元素放在同一个组中进行直接插入排序为止。希尔排序实质上是一种分组插入方法。

希尔排序是不稳定的，执行时间依赖于增量序列。如何选择增量序列使得比较和移动次数最小，至今未能从数学上加以解决。好的增量序列有如下特征：（1）最后一个增量必须是 1；（2）应该尽量避免增量序列中的值（尤其是相邻的值）互为倍数的情况。

希尔排序的源代码在此不列出来了，感兴趣的朋友可以自行编写。

1.5 查找(Search)

在数组中，查找操作是一个需要经常使用的动作。对于小数组（1000 个元素或更小），可以简单的使用顺序查找(Sequential Search)的办法来查找指定的元素。对于有上万或上十万元素的大数组，若是用顺序查找，则有点太浪费时间了。

二分查找法(Binary Search)在一个有序的大数组中查找一个元素效率比较高。假设数组 ArrayA 要查找的元素为 SearchElement，其基本思想为：

（1）查找范围由下标 first 和 last 表示，如果 ArrayA[first]>ArrayA[last]，则退出查找，表明没有匹配项。

（2）计算由下标 first 和 last 标识的数组的中点。

（3）将 SearchElement 值同数组中点处的元素的值进行比较，如果两个值相等则从过程中返回，eax 中包含中点值，这个返回值表明在数组中发现了匹配项；如果 SearchElement 大于中点处的数值，将第一个数组下标重设为中点之后的下一个位置；如果 SearchElement 小于中点处的数值，将最后一个数组下标重设为中点之前的一个位置。

每轮循环之后，查找范围都会缩小一半。虽然二分查找法效率高，但是排序本身是一种很费时的运算，因此，它适用于那种一经建立就很少改动，而又经常需要查找的数组。

源代码

源代码 SIS**.asm**(路径:Data Structures In ASM Source Code\chapter 1\SIS):

```
.
.386
.model      flat,stdcall
option      casemap:none
include     windows.inc
include     user32.inc
includelib          user32.lib
include       kernel32.inc
includelib                kernel32.lib
.data
ArrayA      dword       1,2,3,5,7
szCaption   db   '消息框!',0
szText      db    100         dup(0)
szCharsFormat         db      '数组:%d,%d,%d,%d,%d,查找到位置:%d',0
.code
;-------------------------------------------------------------
;二分查找法
BinarySearch    proc        uses    EBX EDX ESI    pArray:ptr dword,
Count:dword,    ;数组的元素个数
searchVal:dword      ;待查找的值
local       first:dword
local       last:dword
local       mid:dword
mov  first,0
mov  eax,Count
dec   eax
mov  last,eax
mov  edi,searchVal
mov  ebx,pArray
L1:   ;while first<=last
      mov  eax,first
      cmp  eax,last
      jg     L5
;mid=(last+first)/2
      mov  eax,last
      add   eax,first
      shr    eax,1
      mov  mid,eax
      ;edx=[mid]
      mov  esi,mid
      shl    esi,2
```

```
        mov  edx,[ebx+esi]
        ;if edx<searchVal
        ;first=mid+1
        cmp  edx,edi
        jge   L2
        ;first=mid+1
        mov  eax,mid
        inc    eax
        mov  first,eax
        jmp   L4
        ;else if edx>searchVal
        ;last=mid-1
        L2:cmp     edx,edi
        jle     L3
        mov  eax,mid
        dec   eax
        mov  last,eax
        jmp   L4
        ;else return mid
        L3:mov     eax,mid          ;找到了
        jmp   L9    ;返回
        L4:jmp     L1    ;继续while循环
        L5:mov     eax,-1      ;没有找到
        L9:ret

BinarySearch    endp

start:
invoke      BinarySearch,addr    ArrayA,5,5
invoke      wsprintf,addr    szText, addr     szCharsFormat, ArrayA,ArrayA+4,ArrayA+8,\
        ArrayA+12,ArrayA+16,eax
invoke      MessageBox,NULL,offset        szText,offset     szCaption,MB_OK

invoke      ExitProcess,NULL
end  start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=BinarySearch
OBJS=$(NAME).obj
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
```

```
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
 ml $(ML_FLAG) $<
clean:
    del *.obj
```

第二章 堆栈与队列(Stack and Queue)

本章要点

堆栈的定义
入栈、出栈
队列的定义
入队、出队

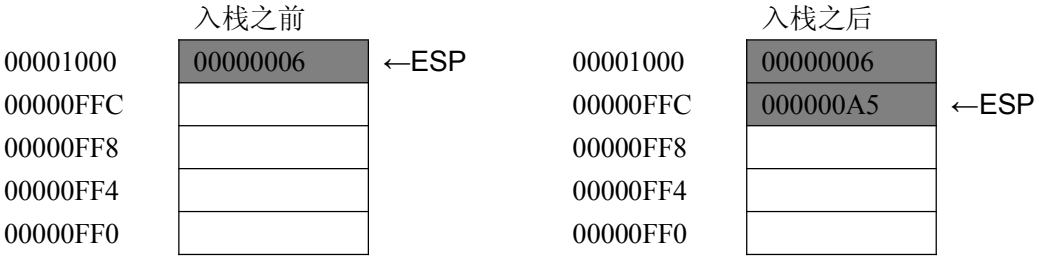
2.1 堆栈(Stack)

堆栈是一种遵循后进先出(LIFO,Last In ,First Out)原则的线性表。该线性表的末尾，叫做堆栈的栈顶(Top)，该线性表的头端，叫做堆栈的栈底(Base)。对堆栈的基本操作只有入栈(push)和出栈(pop)两种。

堆栈使用两个寄存器：SS 和 ESP(Extended Stack Pointer)。在保护模式下，SS 寄存器存放的是段选择器，用户模式程序不应对其进行修改。ESP 寄存器存放的是指向堆栈内特定位置的一个 32 位偏移值。ESP 指向最后压入到堆栈上的数据。

2.1.1 入栈(Push)

32 位的入栈(Push)操作将堆栈指针减 4，并将值复制到堆栈指针所指向的位置。下图的例子中，堆栈上被压入了值 000000A5：



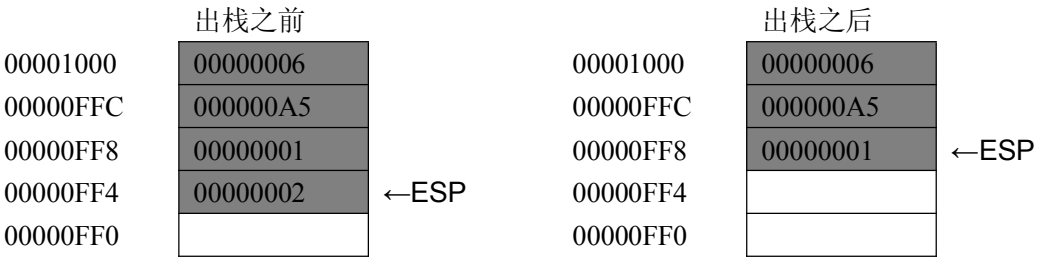
为什么堆栈指针在内存中不是向上扩展呢？这是因为 Intel 的设计者决定了堆栈要向下扩展，其实，不管堆栈的扩展方向如何，都必须遵循后进先出原则。

入栈是通过 PUSH 指令来实现的。PUSH 指令首先减少 ESP 的值，然后将一个 32 位的源操作数复制至堆栈上。对于 32 位操作数，ESP 值每次将减 4。PUSH 指令格式如下：

```
push    r/m32
push    imm32
```

2.1.2 出栈(Pop)

出栈(Pop)操作从栈顶移走一个值并将其置于寄存器或变量中，在值从栈顶弹出之后，堆栈指针相应增加，并指向栈中与弹出数据相邻的最高位置。下图演示了堆栈在讲数值 00000002 出栈前后的变化：



堆栈中 ESP 之下的区域从逻辑上讲是空的，当程序下次执行压栈指令时该区域会被覆盖重写。

出栈是通过 POP 指令来实现的。POP 指令首先将 ESP 所指的堆栈元素复制到 32 位的目的操作数中，然后增加 ESP 的值，对于 32 位操作数，ESP 值将加 4。其格式如下：

```
pop     r/m32
```

源代码

源代码 Stack**.asm**(路径:Data Structures In ASM Source Code\chapter 2\Stack):

```
.
.386
.model    flat,stdcall
option    casemap:none
include   windows.inc
include   user32.inc
includelib        user32.lib
include     kernel32.inc
includelib              kernel32.lib
.data
ArrayA          dword           1,2,3,5,7
szCaption       db          '消息框!',0
szText      db   100       dup(0)
szCharsFormat       db     '利用堆栈反转数组:%d,%d,%d,%d,%d',0
```

```
.code
;-------------------------------------------------------
start:
mov         ecx,5
mov         esi,0
;入栈
L1:
push        ArrayA[esi]
add         esi,4
loop          L1
mov         ecx,5
mov         esi,0
;出栈
L2:
pop         eax
mov         ArrayA[esi],eax
add         esi,4
loop        L2
invoke      wsprintf,addr    szText, addr     szCharsFormat, ArrayA,ArrayA+4,ArrayA+8,\
      ArrayA+12,ArrayA+16
invoke      MessageBox,NULL,offset        szText,offset     szCaption,MB_OK

invoke      ExitProcess,NULL
end   start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=Stack
OBJS=$(NAME).obj
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
 ml $(ML_FLAG) $<
clean:
    del *.obj
```

2.2 队列(Queue)

队列是一种遵循先进先出(FIFO,First In ,First Out)原则的线性表。该线性表的末尾，叫

做队列的队尾(Rear)，允许插入元素；该线性表的头端，叫做队列的队头(Front)。对队列的基本操作只有入队(enqueue)和出队(dequeue)两种。

在队尾插入一个元素，叫做入队。在队头删除一个元素并返回该元素，叫做出队。

源代码

源代码 Queue**.asm**(路径:Data Structures In ASM Source Code\chapter 2\Queue):

```
.
.386
.model      flat,stdcall
option      casemap:none
include     windows.inc
include     user32.inc
includelib          user32.lib
include       kernel32.inc
includelib                kernel32.lib
.data
QueueA   dword      5      dup(0)      ;初始化队列
szCaption   db  '消息框!',0
szText       db    100        dup(0)
szCharsFormat        db      '第%d出队:%d',0
front dword      0      ;队头
rear  dword      0      ;队尾
.code
;-------------------------------------------------------------
start:
mov         eax,rear
mov         QueueA[eax*4],1             ;入队
inc         rear
mov         eax,rear
mov         QueueA[eax*4],2
inc         rear
mov         eax,rear
mov         QueueA[eax*4],3
inc         rear
mov         eax,rear
mov         QueueA[eax*4],5
inc         rear
mov         eax,rear
mov         QueueA[eax*4],7
inc         rear
;出队
mov         ecx,5
L1:
```

```
mov     ebx,front
mov     eax,QueueA[ebx*4]
push    ecx
invoke  wsprintf,addr   szText, addr    szCharsFormat, front,eax
invoke  MessageBox,NULL,offset      szText,offset   szCaption,MB_OK
pop     ecx
inc     front
loop    L1

invoke  ExitProcess,NULL
end  start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=Queue
OBJS=$(NAME).obj
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
 ml $(ML_FLAG) $<
clean:
    del *.obj
```

2.3 结构(Structure)

结构(Structure)是一种构造类型，它是由若干“成员”（又称为域,field）组成的。每个成员可以是一个基本数据类型或者又是一种构造类型。结构是不同类型的数据项的集合。因为结构是一种构造而成的数据类型，所以在声明和使用之前必须先定义它，也就是构造它。如同在调用过程之前先定义过程一样。程序语句可以将结构变量(structure variable)作为单个实体进行访问，也可以对单个域进行访问。

结构的使用包含三个按顺序进行的步骤：

1. 定义结构。
2. 声明一个或多个该结构类型的变量，称为结构变量。
3. 写运行时指令访问结构或结构的域。

2.3.1 定义结构

定义一个结构的一般格式为：

结构名　struct

域的声明

结构名　ends

结构中可包含任意数量的域。

如果为结构的域提供初始化值，在定义结构变量时这些初始化值就成了域的默认值，结构中可使用多种类型的初始化值：

1. 未定义：使用“？”表示域内容未定义。
2. 字符串：用引号括起来的字符序列初始化字符串域
3. 整数：使用整数常量或整数表达式初始化整数域。
4. 数组：当域是一个数组时，使用 dup 操作符初始化数组元素。

例如要定义一个用于描述雇员信息的 Employee 结构，结构中的域有身份证号、姓名、工作年数以及工资的历史值等。在程序中，下面的结构定义应该放在任何 Employee 类型变量的声明之前：

```
Employee    struct
    IdNum          byte      "00000000000000000"
    PersonName     byte      30   dup(0)
    Years          dword     0
    SalaryHistory  dword     0,0,0,0
Employee    ends
```

2.3.2 声明结构变量

我们可以声明一个结构的变量。如果声明的时候使用尖括号 <>，编译器将保留默认的域初始值。若尖括号内包含数据，则可向特定的域中插入新值。例如：

```
worker      Employee <>
worker      Employee <"421734198001011234">
```

当字符串域的初始化值比域短时，剩余的位置将用空格填充，空字符并不会被自动插入到字符串域的末尾。

可以使用逗号来忽略掉对结构中某些域的初始化。例如，下面的语句忽略了 IdNum 域并初始化 PersonName 域：

```
worker      Employee <,"Wzc">
```

如果某个域包含一个数组，可以使用 dup 操作符初始化某些或全部数组元素，如果初始化值比域短，那么剩余位置将用 0 填充。例如，下面的语句初始化 SalaryHistory 的前两个值并将其余设为 0：

```
worker      Employee <,,,2 dup(20000)>
```

可以像下例一样声明结构数组：

```
AllWorker   Employee 3 dup(,,,2 dup(20000))
```

直接引用单个域时要求使用结构变量作为修饰词。例如：

```
.data
```

```
worker Employee <>
.code
mov     edx,worker.Years
mov     worker.SalaryHistory,20000
mov     worker.SalaryHistory+4,30000
mov     edx,offset      worker.PersonName
```

间接操作数允许使用寄存器来寻址结构数据，这种方法在向过程传递结构地址或使用结构数组时提供了特别的灵活性。引用间接操作数时要求使用那个 ptr 操作符：

```
mov     esi,offset      worker
mov eax,(Employee       ptr   [esi]).Years
```

2.3.3 队列的结构

在前面的介绍队列的小节中源代码，并不优美，极易上溢或下溢。队尾指针超越了队列空间的上界（即队满）而不能做入队操作，称之为上溢。队头指针小于了队列空间的下界（即队空）而不能做出队操作，称之为下溢。

为了防止队列的上溢、下溢，统计队列的元素总数以及队列的大小，我们可以定义如下结构：

```
Queue           strcut
    front       dword       0
    rear        dword       0
    count           dword       0
    queueSize           dword   5
    element             dword   5 dup(0)
Queue           ends
```

源代码

源代码 Queue2**.asm**(路径:Data Structures In ASM Source Code\chapter 2\Queue2):

```
.
.386
.model    flat,stdcall
option    casemap:none
include   windows.inc
include   user32.inc
includelib        user32.lib
include    kernel32.inc
includelib              kernel32.lib
include         queue.inc
.data
```

```
QueueA          queue<>  ;声明队列结构的变量
szCaption       db          '消息框!',0
szText      db    100       dup(0)
szCharsFormat       db      '第%d出队:%d',0
.code
;-------------------------------------------------------------
;置空队
InitQueue       proc        uses        esi    q:ptr              queue
        mov         esi,q
        mov         (queue      ptr   [esi]).front,0
        mov         (queue      ptr    [esi]).rear,0
        mov         (queue      ptr      [esi]).count,0
        ret
InitQueue       endp

;判队空,队空返回1，队非空返回0
QueueEmpty   proc           uses       esi    q:ptr              queue
        mov         esi,q
        mov         eax,(queue      ptr    [esi]).count
        cmp         eax,0
        jg          LE1
        mov         eax,1
        jmp         LE2
LE1:
        mov             eax,0
        jmp         LE2
LE2:
        ret

QueueEmpty    endp

;判队满，队满返回1，队未满返回0
QueueFull       proc          uses       esi    q:ptr          queue
        mov         esi,q
        mov         eax,(queue      ptr     [esi]).count
        cmp         eax,(queue      ptr            [esi]).queueSize
        jl          LF1
        mov         eax,1
        jmp         LF2
LF1:
        mov             eax,0
        jmp         LF2
LF2:
        ret
```

```
QueueFull       endp

;入队 ,队满返回-1，入队成功返回1
EnQueue         proc            uses        esi ebx   q:ptr           queue,newElement:dword
        mov         esi,q
        ;队满上溢
        invoke          QueueFull,esi
        cmp         eax,1
        je          L1
        ;队列元素个数加1
        inc          (queue         ptr         [esi]).count
        ;新元素插入到队尾
        mov         eax,newElement
        mov         ebx,(queue      ptr         [esi]).rear
        mov         (queue      ptr         [esi]).element[ebx*4],eax
        ;将队尾指针加1
        inc         (queue    ptr   [esi]).rear
        mov         eax,1
        jmp         L2

L1:
        mov         eax,-1
        jmp         L2
L2:
        ret
EnQueue         endp

;出队,队空则返回-1，出队成功返回元素
DeQueue             proc            uses        esi   ebx  q:ptr        queue
        mov         esi,q
        invoke          QueueEmpty,q
        ;队空下溢
        cmp         eax,1
        je          LD1
        ;取队头元素
        mov         ebx,(queue      ptr   [esi]).front
        mov         eax,(queue      ptr    [esi]).element[ebx*4]
        ;队列元素个数减1
        dec          (queue   ptr   [esi]).count
        ;将队头指针加1
        inc         (queue      ptr   [esi]).front
        jmp         LD2

LD1:
```

```
    mov     eax,-1
    jmp     LD2
LD2:
    ret

DeQueue     endp
start:
;入队
invoke      EnQueue,addr        QueueA,1
cmp     eax,-1
je      LM2
invoke      EnQueue,addr        QueueA,2
invoke      EnQueue,addr        QueueA,3
invoke      EnQueue,addr      QueueA,5
invoke      EnQueue,addr        QueueA,7


;出队
mov     ecx,5
LM1:
invoke      DeQueue,addr        QueueA
push    ecx
invoke  wsprintf,addr       szText, addr        szCharsFormat, QueueA.front,eax
invoke  MessageBox,NULL,offset      szText,offset   szCaption,MB_OK
pop     ecx
loop    LM1

invoke      ExitProcess,NULL
LM2:
end     start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=Queue2
OBJS=$(NAME).obj
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
 ml $(ML_FLAG) $<
clean:
```

```
del *.obj
```

定义队列结构的文件如下：

queue.inc

```
queue           struct
    front       dword       0
    rear        dword       0
    count           dword       0
    queueSize           dword   5
    element             dword   5   dup(0)
queue           ends
```

笔者实测发现，过程 EnQueue 内部声明了标号 L1,L2，则过程 QueueFull 内部不能声明一样的 L1,L2，因为 EnQueue 调用了 QueueFull。对于没有嵌用关系的过程，则其标号可以一样。此处的标号指的是标号名加冒号的格式。

第三章 链表（Linked List）

本章要点

链表的定义
单链表的实现
双链表的实现
循环链表的实现
链栈的实现
链队列的实现

在顺序表（例如数组）中，插入和删除元素操作都需要移动大量的结点，为了减少这样的时间浪费，人们设计链式存储的线性表，简称为链表(Linked List)。链表是用一系列连续或不连续的存储单元来存放结点的线性表。

3.1 单链表(Single Linked List)

我们将每个结点只有一个指针(pointer)的链表称为单链表。为了能正确表示结点之间的逻辑关系，在存储每个结点元素的同时，还必须存储其后继结点的地址信息。单链表的结构如下：

data	next

显然，单链表中的每个结点的存储地址是存放在其前趋结点的 next 中，开始结点无前趋结点，所以单链表指向开始结点，尾端结点无后继结点，故尾端结点的 next 指向 NULL。例如下图是单链表{1,2,3,5,7}的示意图（为了便于演示，这里假设指针占两个字节）：

存储地址	元素(data)	指针(next)
…	…	…

110	7	NULL
…	…	…
130	2	135
135	3	170
…	…	…
165	1	130
170	5	110
…	…	…
	head	
→	165	

单链表由头指针(head)来确定。
单链表的结构如下：

```
SingleLinkedList        struct
    data        dword       0
    next        dword       0
SingleLinkedList        ends
```

3.1.1 创建结点

创建一个结点前，要调用函数 Alloc(笔者注：MASM32的帮助文档 masmlib.chm 将分配内存空间的函数 Alloc 错写为函数 Malloc)分配一个足够大的内存空间(单链表结构大小)，然后创建新结点：

```
local       p:ptr       SingleLinkedList
invoke          Alloc,type      SingleLinkedList
mov         (SingleLinkedList   ptr [eax]).data,0
```

注意，在实用的程序中，分配内存空间时，一定要加入错误处理以防止系统没有足够的空间可以分配。当单链表的结点删除时，或整个单链表不再使用时，一定要调用函数 free 释放内存空间。

3.1.2 查找结点

在链表中，即使知道被访问结点的序号 i，也不能像顺序表中那样直接按序号 i 访问结点，而只能从链表的头指针出发，顺指针 next 逐个结点往下搜索，直至搜索到第 i 个结点为止。因此，链表不是随机存取结构。若线性表的操作主要是进行查找，很少做插入和删除操作时，采用顺序表做存储结构为宜。若线性表的要频繁地进行插入和删除，宜采用链表做存户结构。

设单链表的长度为 n，要查找表中的第 i 个结点，仅当0<=i<n 时，i 值是合法的。但有时需要找头结点的位置，故我们把头结点看做是第0个结点。我们从头结点开始顺着指针扫描，用指针指向当前扫描到的结点，用 j 作计数器，累计当前扫描的结点数。指针的初始值指向头结点，j 的初始值为0，当指针扫描下一个结点时，计数器 j 相应地加1。因此当 j=i 时，指

针所指的结点就是要找的第 i 个结点。

```
GetNode proc uses esi head:ptr SingleLinkedList,i:dword
    local j:dword
    ;从头结点开始扫描
    mov esi,head
    mov j,0
    ;顺指针向后扫描，直到指针 next 为 NULL 或 j=i 为止
    L3:
    cmp (SingleLinkedList ptr [esi]).next,0
    je L4
    mov eax,j
    cmp e ax,i
    jge L4
    mov esi,(SingleLinkedList ptr [esi]).next
    inc j
    jmp L3
    ;判断是否找到了第 i 个结点
    L4:
    mov eax,j
    cmp eax,i
    je L1
    ;没找到第 i 个结点
    mov eax,0
    jmp L2

L1:
    mov eax,esi
    jmp L2
L2:
    ret
GetNode endp
```

按值查找是在链表中，查找是否有结点值等于给定值 key 的结点，若找到了，则返回首次找到的其值为 key 的结点的存储位置；否则返回 NULL。查找过程从开始结点出发，顺着指针逐个将结点的值与给定值 key 作比较。其算法如下：

```
LocateNode proc uses esi head:ptr SingleLinkedList,key:dword
    mov esi,head
    ;从开始结点比较
    mov esi,(SingleLinkedList ptr [esi]).next
    ;执行循环，直到 next=0或找到了值为止
    LD3:
    cmp esi,0
    je LD1
    mov eax,key
```

```
    cmp     eax,(SingleLinkedList ptr [esi]).data
    je      LD1
    ;扫描下一个结点
    mov     esi,(SingleLinkedList ptr [esi]).next
    jmp     LD3

LD1:
    mov     eax,esi
    jmp     LD2
LD2:
    ret
LocateNode      endp
```

3.1.3 插入结点

插入结点操作是将值 x 的新结点插入到表的第 i（0<=i<n）个结点的位置上，即插入到 a_{i-1} 与 a_i 之间。插入结点无须移动结点，仅仅只要修改指针即可。因此，我们必须首先找到 a_{i-1} 的存储位置 p，然后创建一个结点元素为 x 的新结点 s，并令结点 a_{i-1} 的指针指向新结点，新结点的指针指向 a_i。具体算法如下：

```
;将值为 x 的新结点插入到带头结点的单链表 head 的第 i 个结点位置上
InsertNode  proc    uses    esi     ebx  head:ptr
SingleLinkedList,x:dword,i:dword
    ;找到第 i-1个结点
    dec     i
    invoke      GetNode,head,i
    cmp     eax,0
    je      LN1
    mov     esi,eax
    ;分配新结点的内存空间
    invoke      Alloc,type  SingleLinkedList
    ;判断分配内存是否成功
    cmp     eax,0
    je      LN3
    mov     ebx,eax
    mov     eax,x
    mov     (SingleLinkedList   ptr     [ebx]).data,eax
    mov     eax,(SingleLinkedList   ptr     [esi]).next
    mov     (SingleLinkedList   ptr     [ebx]).next,eax
    mov     (SingleLinkedList   ptr     [esi]).next,ebx
    ;插入成功，返回1
```

```
    mov     eax,1
    jmp     LN2

LN1:
    ;没有找到第 i 个结点的位置
    mov     eax,-1
    jmp     LN2
LN3:
    ;分配内存失败
    mov     eax,-3
    jmp     LN2

LN2:
    ret
InsertNode      endp
```

3.1.4 删除结点

删除操作是将表的第 i 个结点删除。因为在单链表中结点 a_i 的存储地址是在其直接前趋结点 a_{i-1} 的指针 next 中，所以我们必须找到 a_{i-1} 的存储位置 p。然后 a_{i-1} 的指针指向 a_i 的直接后继结点。最后，调用函数 free 释放结点 a_i 的空间。链表的删除操作，无须移动结点，修改指针即可。

```
DeleteNode  proc    uses        esi  ebx  head:ptr  SingleLinkedList,i:dword
    ;找到第 i-1个结点
    dec     i
    invoke      GetNode,head,i
    cmp     eax,0
    je      returnLine
    mov     ebx,eax
    cmp     (SingleLinkedList   ptr  [ebx]).next,0
    je      returnLine
    mov     esi,(SingleLinkedList       ptr      [ebx]).next
    mov     eax,(SingleLinkedList       ptr       [esi]).next
    mov     (SingleLinkedList   ptr   [ebx]).next,eax
    invoke      Free,esi
    jmp     returnLine
LD1:
    ;没有找到第 i 个结点的位置
    mov     eax,-1
    jmp     returnLine
```

```
returnLine:
        ret
DeleteNode          endp
```

源代码

源代码 SLL**.asm**(路径:Data Structures In ASM Source Code\chapter 3\SLL):

```
.386
.model      flat,stdcall
option      casemap:none
include     windows.inc
include     user32.inc
includelib          user32.lib
include       kernel32.inc
includelib                kernel32.lib
include             SingleLinkedList.inc
include     masm32.inc
includelib          masm32.lib
include             ole32.inc
includelib          ole32.lib



.data
szCaption          db          '消息框!',0
szText        db    100         dup(0)
szCharsFormat          db      '结点:%d',0
szCharsFormat4         db      '依次在单链表的链尾插入4个结点:%d,%d,%d,%d',0
szCharsFormat5         db      'Insert结点后:%d,%d,%d,%d,%d',0
szCharsFormat24        db      '删除一个结点后:%d,%d,%d,%d',0


headNode         dword      0
node1       dword         0
node2       dword       0
node3       dword       0
node4       dword       0
node5       dword       0
node6       dword       0
node7       dword       0
.code
;-------------------------------------------------------------
```

```
;按序号查找，找到了返回存储位置，没找到返回0
GetNode     proc        uses    esi   head:ptr     SingleLinkedList,    i:dword
    local    j:dword
    ;从头结点开始扫描
    mov      esi,head
    mov      j,0
    ;顺指针向后扫描，直到指针next为NULL或j=i为止
    L3:
    cmp      (SingleLinkedList    ptr    [esi]).next,0
    je   L4
    mov      eax,j
    cmp      eax, i
    jge  L4
    mov      esi,  (SingleLinkedList    ptr   [esi]).next
    inc  j
    jmp  L3
    ;判断是否找到了第i个结点
    L4:
    mov      eax,j
    cmp      eax,i
    je       L1
    ;没找到第i个结点
    mov      eax,0
    jmp      L2

L1:
    mov      eax,esi
    jmp      L2
L2:
    ret
GetNode     endp

;找链尾结点，找到了返回存储位置，没找到返回0
GetLastNode  proc        uses    esi       head:ptr  SingleLinkedList
    ;从头结点开始扫描
    mov      esi,head
    cmp      esi,0
    je   NotFind
    ;顺指针向后扫描，直到指针next为NULL为止
    L3:
    cmp      (SingleLinkedList    ptr    [esi]).next,0
    je       L1
    mov      esi,(SingleLinkedList    ptr    [esi]).next
    jmp      L3
```

```
L1:
        mov         eax,esi
        jmp         L2
NotFind:
        mov         eax,0
        jmp          L2
L2:
        ret
GetLastNode          endp

;按值查找，找到返回存储位置，没找到返回0
LocateNode      proc            uses      esi    head:ptr          SingleLinkedList,      key:dword
        mov         esi,head
        ;从开始结点比较
        mov         esi,(SingleLinkedList       ptr          [esi]).next
        ;执行循环，直到next=0或找到了值为止
        LD3:
        cmp         esi,0
        je          LD1
        mov         eax,key
        cmp         eax,(SingleLinkedList       ptr          [esi]).data
        je          LD1
        ;扫描下一个结点
        mov         esi,(SingleLinkedList       ptr     [esi]).next
        jmp         LD3

LD1:
        mov         eax,esi
        jmp   LD2
LD2:
        ret

LocateNode     endp

;插入操作，将值为x的新结点插入到带头结点的单链表head的第i个结点的位置上
InsertNode      proc      uses      esi       ebx          head:ptr    SingleLinkedList,x:dword,i:dword
        ;找到第i-1个结点
        dec         i
        invoke          GetNode,head,i
        cmp         eax,0
        je          LN1
        mov         esi,eax
```

```
    ;分配新结点的内存空间
    invoke      Alloc,type      SingleLinkedList
    ;判断分配内存是否成功
    cmp     eax,0
    je      LN3
    mov     ebx,eax
    mov     eax,x
    mov     (SingleLinkedList   ptr   [ebx]).data,eax
    mov     eax,(SingleLinkedList   ptr   [esi]).next
    mov     (SingleLinkedList   ptr   [ebx]).next,eax
    mov     (SingleLinkedList   ptr   [esi]).next,ebx
    ;插入成功，返回1
    mov     eax,1
    jmp     LN2

LN1:
    ;没有找到第i个结点的位置
    mov     eax,-1
    jmp     LN2
LN3:
    ;分配内存失败
    mov     eax,-3
    jmp     LN2

LN2:
    ret
InsertNode      endp

;删除结点操作
;删除带头结点的单链表head上的第i个结点
DeleteNode    proc    uses    esi   ebx   head:ptr   SingleLinkedList,i:dword
    ;找到第i-1个结点
    dec     i
    invoke      GetNode,head,i
    cmp     eax,0
    je      returnLine
    mov     ebx,eax
    cmp     (SingleLinkedList   ptr   [ebx]).next,0
    je      returnLine
    mov     esi,(SingleLinkedList   ptr   [ebx]).next
    mov     eax,(SingleLinkedList   ptr   [esi]).next
    mov     (SingleLinkedList   ptr   [ebx]).next,eax
    invoke      Free,esi
    jmp     returnLine
```

```
LD1:
    ;没有找到第i-1个结点的位置
    mov     eax,-1
    jmp     returnLine
returnLine:
    ret

DeleteNode      endp

;在单链表链尾创建新结点
;若创建头结点，请将head置0
CreateNode      proc        uses    esi ebx head:ptr SingleLinkedList,x:dword
    ;若head=0，则创建头指针
    cmp     head,0
    je      createHead

    invoke      GetLastNode,head
    cmp     eax,0
    je      returnLine
    mov     esi,eax
    ;分配新结点的内存空间
    invoke      Alloc,type  SingleLinkedList
    ;判断分配内存是否成功
    cmp     eax,0
    je      mallocFail
    mov     ebx,eax
    mov     eax,x
    mov     (SingleLinkedList   ptr   [ebx]).data,eax
    mov     (SingleLinkedList   ptr   [ebx]).next,0
    mov     (SingleLinkedList   ptr   [esi]).next,ebx
    jmp     returnLine
createHead:
    ;分配新结点的内存空间
    mov     eax,type SingleLinkedList
    invoke      Alloc,eax
    ;判断分配内存是否成功
    cmp     eax,0
    je      mallocFail
    mov     ebx,eax
    mov     eax,x
    mov     (SingleLinkedList   ptr   [ebx]).data,eax
    mov     (SingleLinkedList   ptr   [ebx]).next,0
    mov     eax,ebx
    jmp     returnLine
```

```
mallocFail:
        ;分配内存失败
        mov         eax,-3
        jmp         returnLine
returnLine:
        ret

CreateNode      endp
start:
invoke          CreateNode,0,1
mov         headNode,eax
invoke      wsprintf,addr    szText, addr     szCharsFormat, (SingleLinkedList    ptr     [eax]).data


invoke          MessageBox,NULL,offset       szText,offset     szCaption,MB_OK
invoke          CreateNode,headNode,2
invoke          CreateNode,headNode,5
invoke          CreateNode,headNode,7
;显示前面插入的4个结点
mov         eax,headNode
mov         ebx,(SingleLinkedList       ptr     [eax]).data
mov         node1,ebx
mov         eax,(SingleLinkedList       ptr     [eax]).next
mov         ebx,(SingleLinkedList       ptr     [eax]).data
mov         node2,ebx
mov         eax,(SingleLinkedList       ptr     [eax]).next
mov         ebx,(SingleLinkedList       ptr     [eax]).data
mov         node3,ebx
mov         eax,(SingleLinkedList       ptr     [eax]).next
mov         ebx,(SingleLinkedList       ptr    [eax]).data
mov         node4,ebx
invoke      wsprintf,addr    szText, addr     szCharsFormat4, node1,node2,node3,node4

invoke      MessageBox,NULL,offset       szText,offset     szCaption,MB_OK
invoke      InsertNode,headNode,11,3
mov         eax,headNode
mov         ebx,(SingleLinkedList       ptr     [eax]).data
mov         node1,ebx
mov         eax,(SingleLinkedList       ptr    [eax]).next
mov         ebx,(SingleLinkedList       ptr    [eax]).data
mov         node2,ebx
mov         eax,(SingleLinkedList       ptr     [eax]).next
mov         ebx,(SingleLinkedList       ptr     [eax]).data
```

```
mov     node3,ebx
mov     eax,(SingleLinkedList   ptr   [eax]).next
mov     ebx,(SingleLinkedList   ptr   [eax]).data
mov     node4,ebx
mov     eax,(SingleLinkedList   ptr   [eax]).next
mov     ebx,(SingleLinkedList   ptr   [eax]).data
mov     node5,ebx
invoke      wsprintf,addr   szText, addr   szCharsFormat5, node1,node2,node3,node4,node5
invoke      MessageBox,NULL,offset      szText,offset   szCaption,MB_OK



invoke  DeleteNode,headNode,3
mov     eax,headNode
mov     ebx,(SingleLinkedList   ptr   [eax]).data
mov     node1,ebx
mov     eax,(SingleLinkedList   ptr   [eax]).next
mov     ebx,(SingleLinkedList   ptr   [eax]).data
mov     node2,ebx
mov     eax,(SingleLinkedList   ptr   [eax]).next
mov     ebx,(SingleLinkedList   ptr   [eax]).data
mov     node3,ebx
mov     eax,(SingleLinkedList   ptr   [eax]).next
mov     ebx,(SingleLinkedList   ptr   [eax]).data
mov     node4,ebx
invoke  wsprintf,addr   szText, addr   szCharsFormat24, node1,node2,node3,node4
invoke  MessageBox,NULL,offset      szText,offset   szCaption,MB_OK

invoke      GetNode,headNode,2
invoke  wsprintf,addr   szText, addr   szCharsFormat, (SingleLinkedList   ptr   [eax]).data
invoke  MessageBox,NULL,offset      szText,offset   szCaption,MB_OK
invoke  LocateNode,headNode,2
invoke  wsprintf,addr   szText, addr   szCharsFormat, (SingleLinkedList   ptr   [eax]).data
invoke  MessageBox,NULL,offset      szText,offset   szCaption,MB_OK


invoke  ExitProcess,NULL
end  start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=SLL
OBJS=$(NAME).obj
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
 ml $(ML_FLAG) $<
clean:
    del *.obj
```

定义单链表结点结构的文件如下：

SingleLinkedList.inc

```
SingleLinkedList        struct
    data        dword       0
    next        dword       0
SingleLinkedList        ends
```

3.2 双链表(Double Linked List)

为了方便的查找结点的直接前趋（某结点的前一个结点称为直接前趋），我们在单链表的每个结点里再增加一个指向其直接前趋的指针 prior，这样的链表被称为双链表。

其结点的数据结构为：

```
DoubleLinkedList        struct
    data        dword       0
    prior       dword       0
    next        dword       0
DoubleLinkedList        ends
```

在双链表中，有些操作如：GetNode,LocateNode 等等仅需涉及一个方向的指针，它们的算法描述和单链表的操作相同，在此就省略了。但是在插入、删除时有很大的不同，本节重点阐述插入、删除操作。注意，在双链表中，，若有直接前趋和直接后继，插入结点操作和删除结点操作必须同时修改两个方向上的指针。

3.2.1 插入结点

插入结点操作是将值为 x 的新结点插入到表的第 i（0<=i<n）个结点的位置上，即插入到 a_{i-1} 和 a_i 之间。

双链表的前插操作算法如下：

```
InsertNode          proc
```

InsertNode endp

3.2.2 删除结点

删除结点操作是将第 i 个结点删除并释放内存空间。

DeleteNode proc

DeleteNode endp

源代码

源代码 DLL**.asm**(路径:Data Structures In ASM Source Code\chapter 3\DLL):

```
.386
.model flat,stdcall
option casemap:none
include windows.inc
include user32.inc
includelib user32.lib
include kernel32.inc
includelib kernel32.lib
include DoubleLinkedList.inc
include masm32.inc
includelib masm32.lib
include ole32.inc
includelib ole32.lib



.data
szCaption db '消息框!',0
szText db 100 dup(0)
szCharsFormat db '结点:%d',0
szCharsFormat4 db '依次在双链表的链尾插入4个结点:%d,%d,%d,%d',0
szCharsFormat5 db 'Insert结点后:%d,%d,%d,%d,%d',0
szCharsFormat24 db '删除一个结点后:%d,%d,%d,%d',0


headNode dword 0
node1 dword 0
node2 dword 0
node3 dword 0
node4 dword 0
```

```
node5     dword     0
node6     dword     0
node7     dword     0
.code
;-------------------------------------------------------------

;按序号查找，找到了返回存储位置，没找到返回0
GetNode       proc          uses      esi   head:ptr      DoubleLinkedList,   i:dword
      local       j:dword
      ;从头结点开始扫描
      mov         esi,head
      mov         j,0
      ;顺指针向后扫描，直到指针next为NULL或j=i为止
      L3:
      cmp         (DoubleLinkedList   ptr    [esi]).next,0
      je    L4
      mov         eax,j
      cmp         eax, i
      jge   L4
      mov         esi,  (DoubleLinkedList   ptr   [esi]).next
      inc   j
      jmp   L3
      ;判断是否找到了第i个结点
      L4:
      mov         eax,j
      cmp         eax,i
      je          L1
      ;没找到第i个结点
      mov         eax,0
      jmp         L2

L1:
      mov         eax,esi
      jmp         L2
L2:
      ret
GetNode       endp

;找链尾结点，找到了返回存储位置，没找到返回0
GetLastNode   proc          uses      esi       head:ptr   DoubleLinkedList
      ;从头结点开始扫描
      mov         esi,head
      cmp         esi,0
      je    NotFind
```

```
        ;顺指针向后扫描，直到指针next为NULL为止
        L3:
        cmp     (DoubleLinkedList   ptr     [esi]).next,0
        je      L1
        mov     esi,(DoubleLinkedList     ptr     [esi]).next
        jmp     L3


L1:
        mov     eax,esi
        jmp     L2
NotFind:
        mov     eax,0
        jmp      L2
L2:
        ret
GetLastNode         endp

;按值查找，找到返回存储位置，没找到返回0
LocateNode      proc            uses    esi   head:ptr       DoubleLinkedList,    key:dword
        mov     esi,head
        ;从开始结点比较
        mov     esi,(DoubleLinkedList     ptr         [esi]).next
        ;执行循环，直到next=0或找到了值为止
        LD3:
        cmp     esi,0
        je      LD1
        mov     eax,key
        cmp     eax,(DoubleLinkedList     ptr         [esi]).data
        je      LD1
        ;扫描下一个结点
        mov     esi,(DoubleLinkedList     ptr     [esi]).next
        jmp     LD3

LD1:
        mov     eax,esi
        jmp  LD2
LD2:
        ret

LocateNode    endp

;插入操作，将值为x的新结点插入到带头结点的双链表head的第i个结点的位置上
InsertNode      proc  uses      esi   ebx  head:ptr   DoubleLinkedList,x:dword,i:dword
```

```
        ;找到第i个结点

        invoke      GetNode,head,i
        cmp         eax,0
        je          LN1
        mov         esi,eax
        ;分配新结点的内存空间
        invoke          Alloc,type      DoubleLinkedList
        ;判断分配内存是否成功
        cmp         eax,0
        je          LN3
        mov         ebx,eax
        mov         eax,x
        mov         (DoubleLinkedList   ptr     [ebx]).data,eax
        ;将原来第i个结点的前趋指针赋值给新结点的前趋指针
        mov         eax,(DoubleLinkedList     ptr     [esi]).prior
        mov         (DoubleLinkedList   ptr   [ebx]).prior,eax
        ;新结点的后继指针指向原来的第i个结点
        mov         (DoubleLinkedList   ptr     [ebx]).next,esi
        mov         eax,(DoubleLinkedList     ptr     [esi]).prior
        ;原来的第i个结点的前趋结点的后继指针指向新结点
        mov         (DoubleLinkedList   ptr   [eax]).next,ebx
        ;原来的第i个结点的前趋指针指向新结点
        mov         (DoubleLinkedList   ptr   [esi]).prior,ebx
        ;插入成功，返回1
        mov         eax,1
        jmp         LN2

LN1:
        ;没有找到第i个结点的位置
        mov         eax,-1
        jmp         LN2
LN3:
        ;分配内存失败
        mov         eax,-3
        jmp         LN2

LN2:
        ret
InsertNode      endp

;删除结点操作
;删除带头结点的双链表head上的第i个结点
DeleteNode      proc        uses        esi   ebx   head:ptr     DoubleLinkedList,i:dword
```

```
    ;找到第i个结点
    invoke    GetNode,head,i
    cmp       eax,0
    je    returnLine
    mov       ebx,eax
    ;若等于0，则是尾结点
    cmp       (DoubleLinkedList    ptr    [ebx]).next,0
    je    EndNode
    ;要删除结点的后继结点的指针
    mov       esi,(DoubleLinkedList      ptr    [ebx]).next
    ;要删除结点的前趋结点的指针
    mov       eax,(DoubleLinkedList      ptr    [ebx]).prior
    mov       (DoubleLinkedList    ptr    [eax]).next,esi
    mov       (DoubleLinkedList    ptr    [esi]).prior,eax
    invoke    Free,ebx
    mov       eax,1
    jmp       returnLine
LD1:
    ;没有找到第i个结点的位置
    mov       eax,-1
    jmp       returnLine
;删除尾结点
EndNode:
    mov       eax,(DoubleLinkedList      ptr    [ebx]).prior
    ;将第i个结点的前趋结点的后继指针赋值为NULL
    mov       (DoubleLinkedList    ptr    [eax]).next,0
    invoke    Free,ebx
    mov       eax,1
    jmp       returnLine

returnLine:
    ret

DeleteNode    endp

;在双链表链尾创建新结点
;若创建头结点，请将head置0
CreateNode    proc    uses    esi    ebx    head:ptr    DoubleLinkedList,x:dword
    ;若head=0，则创建头指针
    cmp       head,0
    je    createHead

    invoke    GetLastNode,head
    cmp       eax,0
```

```
        je          returnLine
        mov         esi,eax
        ;分配新结点的内存空间
        invoke          Alloc,type      DoubleLinkedList
        ;判断分配内存是否成功
        cmp         eax,0
        je      mallocFail
        mov         ebx,eax
        mov         eax,x
        mov         (DoubleLinkedList   ptr     [ebx]).data,eax
        mov         (DoubleLinkedList   ptr     [ebx]).prior,esi
        mov         (DoubleLinkedList   ptr     [ebx]).next,0
        mov         (DoubleLinkedList   ptr   [esi]).next,ebx
        jmp         returnLine
createHead:
        ;分配新结点的内存空间
        mov         eax,type   DoubleLinkedList
        invoke          Alloc,eax
        ;判断分配内存是否成功
        cmp         eax,0
        je      mallocFail
        mov         ebx,eax
        mov         eax,x
        mov         (DoubleLinkedList   ptr     [ebx]).data,eax
        mov         (DoubleLinkedList   ptr     [ebx]).prior,0
        mov         (DoubleLinkedList   ptr   [ebx]).next,0
        mov         eax,ebx
        jmp         returnLine

mallocFail:
        ;分配内存失败
        mov         eax,-3
        jmp         returnLine
returnLine:
        ret

CreateNode      endp
start:
invoke      CreateNode,0,1
mov         headNode,eax
invoke      wsprintf,addr    szText, addr     szCharsFormat, (DoubleLinkedList   ptr    [eax]).data


invoke      MessageBox,NULL,offset       szText,offset     szCaption,MB_OK
```

```
invoke      CreateNode,headNode,2
invoke      CreateNode,headNode,5
invoke      CreateNode,headNode,7
;显示前面插入的4个结点
mov         eax,headNode
mov         ebx,(DoubleLinkedList      ptr    [eax]).data
mov         node1,ebx
mov         eax,(DoubleLinkedList      ptr    [eax]).next
mov         ebx,(DoubleLinkedList      ptr     [eax]).data
mov         node2,ebx
mov         eax,(DoubleLinkedList      ptr    [eax]).next
mov         ebx,(DoubleLinkedList      ptr     [eax]).data
mov         node3,ebx
mov         eax,(DoubleLinkedList      ptr    [eax]).next
mov         ebx,(DoubleLinkedList      ptr    [eax]).data
mov         node4,ebx
invoke      wsprintf,addr     szText, addr     szCharsFormat4, node1,node2,node3,node4

invoke      MessageBox,NULL,offset        szText,offset     szCaption,MB_OK
invoke      InsertNode,headNode,11,3
mov         eax,headNode
mov         ebx,(DoubleLinkedList      ptr    [eax]).data
mov         node1,ebx
mov         eax,(DoubleLinkedList      ptr     [eax]).next
mov         ebx,(DoubleLinkedList      ptr    [eax]).data
mov         node2,ebx
mov         eax,(DoubleLinkedList      ptr    [eax]).next
mov         ebx,(DoubleLinkedList      ptr     [eax]).data
mov         node3,ebx
mov         eax,(DoubleLinkedList      ptr     [eax]).next
mov         ebx,(DoubleLinkedList      ptr    [eax]).data
mov         node4,ebx
mov         eax,(DoubleLinkedList      ptr    [eax]).next
mov         ebx,(DoubleLinkedList      ptr     [eax]).data
mov         node5,ebx
invoke      wsprintf,addr     szText, addr     szCharsFormat5, node1,node2,node3,node4,node5
invoke      MessageBox,NULL,offset        szText,offset     szCaption,MB_OK



invoke      DeleteNode,headNode,3
mov         eax,headNode
mov         ebx,(DoubleLinkedList      ptr     [eax]).data
mov         node1,ebx
```

```
mov     eax,(DoubleLinkedList   ptr   [eax]).next
mov     ebx,(DoubleLinkedList   ptr   [eax]).data
mov     node2,ebx
mov     eax,(DoubleLinkedList   ptr   [eax]).next
mov     ebx,(DoubleLinkedList   ptr   [eax]).data
mov     node3,ebx
mov     eax,(DoubleLinkedList   ptr   [eax]).next
mov     ebx,(DoubleLinkedList   ptr   [eax]).data
mov     node4,ebx
invoke  wsprintf,addr   szText, addr   szCharsFormat24, node1,node2,node3,node4
invoke  MessageBox,NULL,offset      szText,offset    szCaption,MB_OK

invoke  GetNode,headNode,2
invoke  wsprintf,addr   szText, addr   szCharsFormat, (DoubleLinkedList   ptr    [eax]).data
invoke  MessageBox,NULL,offset      szText,offset    szCaption,MB_OK
invoke  LocateNode,headNode,2
invoke  wsprintf,addr   szText, addr   szCharsFormat, (DoubleLinkedList   ptr   [eax]).data
invoke  MessageBox,NULL,offset      szText,offset    szCaption,MB_OK


invoke  ExitProcess,NULL
end  start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=DLL
OBJS=$(NAME).obj
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
 ml $(ML_FLAG) $<
clean:
    del *.obj
```

定义双链表结点结构的文件如下：

DoubleLinkedList.inc

```
DoubleLinkedList        struct
    data        dword       0
    prior       dword       0
```

```
    next        dword           0
DoubleLinkedList        ends
```

3.3 循环链表(Circular Linked List)

循环链表是一种首尾相接的链表。它的特点是表中最后一个结点的指针指向头结点，整个链表构成一个圆环。从任意结点出发均可找到表中其他结点。循环链表的操作和非循环的线性链表基本一致，差别在于算法中的循环条件不是判断尾结点的指针是否为空(NULL)，而是它是否等于头指针，另外插入尾结点时，其指针要指向头结点。

3.3.1 单循环链表

在单链表中，将尾结点的指针指向 NULL 改为指向表头结点，就得到了单链形式的循环链表，简称为单循环链表。在单循环链表中，所有的结点都被链在一个圆环上。

源代码

源代码 CSLL**.asm**(路径:Data Structures In ASM Source Code\chapter 3\CSLL):

```
.386
.model    flat,stdcall
option    casemap:none
include    windows.inc
include    user32.inc
includelib        user32.lib
include     kernel32.inc
includelib              kernel32.lib
include         SingleLinkedList.inc
include    masm32.inc
includelib        masm32.lib
include         ole32.inc
includelib        ole32.lib



.data
szCaption       db          '消息框!',0
szText      db   100       dup(0)
szCharsFormat       db     '结点:%d',0
szCharsFormat4      db     '依次在单链表的链尾插入4个结点:%d,%d,%d,%d',0
```

```
szCharsFormat5      db      'Insert结点后:%d,%d,%d,%d,%d',0
szCharsFormat24     db      '删除一个结点后:%d,%d,%d,%d',0


headNode        dword       0
rearNode        dword       0
node1       dword       0
node2       dword       0
node3       dword       0
node4       dword       0
node5       dword       0
node6       dword       0
node7       dword       0
.code
;-------------------------------------------------------------

;按序号查找，找到了返回存储位置，没找到返回0
GetNode         proc            uses        esi    head:ptr         SingleLinkedList,      i:dword
        local       j:dword
        ;从头结点开始扫描
        mov         esi,head
        mov         j,0
        ;顺指针向后扫描，直到指针next为头指针或j=i为止
        L3:
        mov         eax,head
        cmp         (SingleLinkedList      ptr      [esi]).next,eax
        je      L4
        mov         eax,j
        cmp         eax,  i
        jge    L4
        mov         esi,   (SingleLinkedList      ptr     [esi]).next
        inc    j
        jmp    L3
        ;判断是否找到了第i个结点
        L4:
        mov         eax,j
        cmp         eax,i
        je          L1
        ;没找到第i个结点
        mov         eax,0
        jmp         L2

L1:
        mov         eax,esi
```

```
        jmp         L2
L2:
        ret
GetNode         endp

;找链尾结点，找到了返回存储位置，没找到返回0
GetLastNode     proc            uses        esi         head:ptr    SingleLinkedList
        ;从头结点开始扫描
        mov         esi,head
        cmp         esi,0
        je      NotFind
        ;顺指针向后扫描，直到指针next为头结点为止
        L3:
        mov         eax,head
        cmp         (SingleLinkedList       ptr     [esi]).next,eax
        je          L1
        mov         esi,(SingleLinkedList       ptr     [esi]).next
        jmp         L3


L1:
        mov         eax,esi
        jmp         L2
NotFind:
        mov         eax,0
        jmp          L2
L2:
        ret
GetLastNode         endp

;按值查找，找到返回存储位置，没找到返回0
LocateNode      proc            uses        esi     head:ptr        SingleLinkedList,       key:dword
        mov         esi,head
        ;从开始结点比较
        mov         esi,(SingleLinkedList       ptr         [esi]).next
        ;执行循环，直到next=0或找到了值为止
        LD3:
        cmp         esi,0
        je          LD1
        mov         eax,key
        cmp         eax,(SingleLinkedList       ptr         [esi]).data
        je          LD1
        ;扫描下一个结点
        mov         esi,(SingleLinkedList       ptr     [esi]).next
```

```
        cmp     esi,head
        je      LD4
        jmp     LD3

LD1:
        mov     eax,esi
        jmp  LD2
LD4:
        mov     eax,0
        jmp     LD2
LD2:
        ret

LocateNode      endp

;插入操作，将值为x的新结点插入到带头结点的单链表head的第i个结点的位置上
InsertNode      proc    uses    esi     ebx     head:ptr  SingleLinkedList,x:dword,i:dword
        ;找到第i-1个结点
        dec     i
        invoke          GetNode,head,i
        cmp     eax,0
        je      LN1
        mov     esi,eax
        ;分配新结点的内存空间
        invoke          Alloc,type      SingleLinkedList
        ;判断分配内存是否成功
        cmp     eax,0
        je      LN3
        mov     ebx,eax
        mov     eax,x
        mov     (SingleLinkedList   ptr    [ebx]).data,eax
        mov     eax,(SingleLinkedList     ptr   [esi]).next
        mov     (SingleLinkedList   ptr        [ebx]).next,eax
        mov     (SingleLinkedList   ptr  [esi]).next,ebx
        ;插入成功，返回1
        mov     eax,1
        jmp     LN2

LN1:
        ;没有找到第i个结点的位置
        mov     eax,-1
        jmp     LN2
LN3:
        ;分配内存失败
```

```
        mov     eax,-3
        jmp     LN2

LN2:
        ret
InsertNode          endp

;删除结点操作
;删除带头结点的单链表head上的第i个结点
DeleteNode      proc    uses    esi ebx head:ptr SingleLinkedList,i:dword
        ;找到第i-1个结点
        dec     i
        invoke      GetNode,head,i
        cmp     eax,0
        je      returnLine
        mov     ebx,eax
        cmp     (SingleLinkedList   ptr   [ebx]).next,0
        je      returnLine
        mov     esi,(SingleLinkedList   ptr   [ebx]).next
        mov     eax,(SingleLinkedList   ptr   [esi]).next
        mov     (SingleLinkedList   ptr   [ebx]).next,eax
        invoke      Free,esi
        jmp     returnLine
LD1:
        ;没有找到第i-1个结点的位置
        mov     eax,-1
        jmp     returnLine
returnLine:
        ret

DeleteNode          endp

;在单链表链尾创建新结点
;若创建头结点，请将head置0
CreateNode          proc        uses    esi ebx head:ptr SingleLinkedList,x:dword
        ;若head=0，则创建头指针
        cmp     head,0
        je      createHead

        invoke      GetLastNode,head
        cmp     eax,0
        je      returnLine
        mov     esi,eax
        ;分配新结点的内存空间
```

```
        invoke          Alloc,type      SingleLinkedList
        ;判断分配内存是否成功
        cmp         eax,0
        je          mallocFail
        mov         ebx,eax
        mov         eax,x
        mov         (SingleLinkedList   ptr     [ebx]).data,eax
        mov         eax,head
        mov         (SingleLinkedList   ptr   [ebx]).next,eax
        mov         (SingleLinkedList   ptr     [esi]).next,ebx
        jmp         returnLine
createHead:
        ;分配新结点的内存空间
        mov         eax,type  SingleLinkedList
        invoke          Alloc,eax
        ;判断分配内存是否成功
        cmp         eax,0
        je          mallocFail
        mov         ebx,eax
        mov         eax,x
        mov         (SingleLinkedList   ptr   [ebx]).data,eax
        mov         (SingleLinkedList   ptr     [ebx]).next,ebx
        mov         eax,ebx
        jmp         returnLine

mallocFail:
        ;分配内存失败
        mov         eax,-3
        jmp         returnLine
returnLine:
        ret

CreateNode      endp
start:
invoke          CreateNode,0,1
mov       headNode,eax
invoke    wsprintf,addr   szText, addr    szCharsFormat, (SingleLinkedList    ptr     [eax]).data


invoke          MessageBox,NULL,offset      szText,offset    szCaption,MB_OK
invoke          CreateNode,headNode,2
invoke          CreateNode,headNode,5
invoke          CreateNode,headNode,7
;显示前面插入的4个结点
```

```
mov     eax,headNode
mov     ebx,(SingleLinkedList   ptr   [eax]).data
mov     node1,ebx
mov     eax,(SingleLinkedList   ptr   [eax]).next
mov     ebx,(SingleLinkedList   ptr   [eax]).data
mov     node2,ebx
mov     eax,(SingleLinkedList   ptr   [eax]).next
mov     ebx,(SingleLinkedList   ptr   [eax]).data
mov     node3,ebx
mov     eax,(SingleLinkedList   ptr   [eax]).next
mov     ebx,(SingleLinkedList   ptr  [eax]).data
mov     node4,ebx
invoke  wsprintf,addr   szText, addr   szCharsFormat4, node1,node2,node3,node4

invoke  MessageBox,NULL,offset      szText,offset   szCaption,MB_OK
invoke  InsertNode,headNode,11,3
mov     eax,headNode
mov     ebx,(SingleLinkedList   ptr   [eax]).data
mov     node1,ebx
mov     eax,(SingleLinkedList   ptr  [eax]).next
mov     ebx,(SingleLinkedList   ptr  [eax]).data
mov     node2,ebx
mov     eax,(SingleLinkedList   ptr   [eax]).next
mov     ebx,(SingleLinkedList   ptr   [eax]).data
mov     node3,ebx
mov     eax,(SingleLinkedList   ptr   [eax]).next
mov     ebx,(SingleLinkedList   ptr  [eax]).data
mov     node4,ebx
mov     eax,(SingleLinkedList   ptr   [eax]).next
mov     ebx,(SingleLinkedList   ptr  [eax]).data
mov     node5,ebx
invoke      wsprintf,addr   szText, addr   szCharsFormat5, node1,node2,node3,node4,node5
invoke      MessageBox,NULL,offset      szText,offset   szCaption,MB_OK



invoke  DeleteNode,headNode,3
mov     eax,headNode
mov     ebx,(SingleLinkedList   ptr  [eax]).data
mov     node1,ebx
mov     eax,(SingleLinkedList   ptr   [eax]).next
mov     ebx,(SingleLinkedList   ptr   [eax]).data
mov     node2,ebx
mov     eax,(SingleLinkedList   ptr   [eax]).next
```

```
mov     ebx,(SingleLinkedList   ptr   [eax]).data
mov     node3,ebx
mov     eax,(SingleLinkedList   ptr    [eax]).next
mov     ebx,(SingleLinkedList   ptr   [eax]).data
mov     node4,ebx
invoke  wsprintf,addr   szText, addr   szCharsFormat24, node1,node2,node3,node4
invoke  MessageBox,NULL,offset      szText,offset    szCaption,MB_OK

invoke          GetNode,headNode,2
invoke  wsprintf,addr   szText, addr   szCharsFormat, (SingleLinkedList   ptr    [eax]).data
invoke  MessageBox,NULL,offset      szText,offset    szCaption,MB_OK
invoke  LocateNode,headNode,2
invoke  wsprintf,addr   szText, addr   szCharsFormat, (SingleLinkedList   ptr    [eax]).data
invoke  MessageBox,NULL,offset      szText,offset    szCaption,MB_OK


invoke  ExitProcess,NULL
end  start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=CSLL
OBJS=$(NAME).obj
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
 ml $(ML_FLAG) $<
clean:
    del *.obj
```

定义单链表结点结构的文件如下：

SingleLinkedList.inc

```
SingleLinkedList        struct
    data        dword       0
    next        dword       0
SingleLinkedList        ends
```

3.3.2 双循环链表

将双链表的头结点与尾结点链接起来构成的链表，称之为双循环链表。

源代码

源代码 CDLL**.asm**(路径:Data Structures In ASM Source Code\chapter 3\CDLL):

```
.386
.model      flat,stdcall
option      casemap:none
include     windows.inc
include     user32.inc
includelib          user32.lib
include       kernel32.inc
includelib                kernel32.lib
include             DoubleLinkedList.inc
include     masm32.inc
includelib          masm32.lib
include             ole32.inc
includelib          ole32.lib



.data
szCaption          db           '消息框!',0
szText        db    100        dup(0)
szCharsFormat          db      '结点:%d',0
szCharsFormat4         db      '依次在双链表的链尾插入4个结点:%d,%d,%d,%d',0
szCharsFormat5         db      'Insert结点后:%d,%d,%d,%d,%d',0
szCharsFormat24        db      '删除一个结点后:%d,%d,%d,%d',0


headNode         dword      0
node1       dword         0
node2       dword       0
node3       dword       0
node4       dword       0
node5       dword       0
node6       dword       0
node7       dword       0
.code
;-------------------------------------------------------------
```

```
;按序号查找，找到了返回存储位置，没找到返回0
GetNode     proc        uses    esi   head:ptr     DoubleLinkedList,   i:dword
    local   j:dword
    ;从头结点开始扫描
    mov     esi,head
    mov     j,0
    ;顺指针向后扫描，直到指针next为头指针或j=i为止
    L3:
    mov     eax,head
    cmp     (DoubleLinkedList   ptr   [esi]).next,eax
    je   L4
    mov     eax,j
    cmp     eax, i
    jge  L4
    mov     esi,  (DoubleLinkedList   ptr   [esi]).next
    inc  j
    jmp  L3
    ;判断是否找到了第i个结点
    L4:
    mov     eax,j
    cmp     eax,i
    je      L1
    ;没找到第i个结点
    mov     eax,0
    jmp     L2

L1:
    mov     eax,esi
    jmp     L2
L2:
    ret
GetNode     endp

;找链尾结点，找到了返回存储位置，没找到返回0
GetLastNode  proc       uses    esi       head:ptr  DoubleLinkedList
    ;从头结点开始扫描
    mov     esi,head
    cmp     esi,0
    je   NotFind
    ;顺指针向后扫描，直到指针next为头指针  为止
    L3:
    mov     eax,head
    cmp     (DoubleLinkedList   ptr   [esi]).next,eax
```

```
        je          L1
        mov         esi,(DoubleLinkedList       ptr     [esi]).next
        jmp         L3


L1:
        mov         eax,esi
        jmp         L2
NotFind:
        mov         eax,0
        jmp          L2
L2:
        ret
GetLastNode           endp

;按值查找，找到返回存储位置，没找到返回0
LocateNode      proc            uses        esi    head:ptr          DoubleLinkedList,     key:dword
        mov         esi,head
        ;从开始结点比较
        mov         esi,(DoubleLinkedList       ptr          [esi]).next
        ;执行循环，直到next=头结点或找到了值为止
        LD3:
        cmp         esi,0
        je          LD1
        mov         eax,key
        cmp         eax,(DoubleLinkedList       ptr          [esi]).data
        je          LD1
        ;扫描下一个结点
        mov         esi,(DoubleLinkedList       ptr     [esi]).next
        cmp         esi,head
        je          LD4
        jmp         LD3

LD1:
        mov         eax,esi
        jmp   LD2
LD4:
        mov         eax,0
        jmp   LD2
LD2:
        ret

LocateNode      endp
```

```
;插入操作，将值为x的新结点插入到带头结点的双链表head的第i个结点的位置上
InsertNode      proc  uses     esi   ebx  head:ptr  DoubleLinkedList,x:dword,i:dword
        ;找到第i个结点

        invoke     GetNode,head,i
        cmp        eax,0
        je         LN1
        mov        esi,eax
        ;分配新结点的内存空间
        invoke          Alloc,type        DoubleLinkedList
        ;判断分配内存是否成功
        cmp        eax,0
        je         LN3
        mov        ebx,eax
        mov        eax,x
        mov        (DoubleLinkedList   ptr    [ebx]).data,eax
        ;将原来第i个结点的前趋指针赋值给新结点的前趋指针
        mov        eax,(DoubleLinkedList     ptr     [esi]).prior
        mov        (DoubleLinkedList   ptr   [ebx]).prior,eax
        ;新结点的后继指针指向原来的第i个结点
        mov        (DoubleLinkedList   ptr    [ebx]).next,esi
        mov        eax,(DoubleLinkedList     ptr     [esi]).prior
        ;原来的第i个结点的前趋结点的后继指针指向新结点
        mov        (DoubleLinkedList   ptr   [eax]).next,ebx
        ;原来的第i个结点的前趋指针指向新结点
        mov        (DoubleLinkedList   ptr   [esi]).prior,ebx
        ;插入成功，返回1
        mov        eax,1
        jmp        LN2

LN1:
        ;没有找到第i个结点的位置
        mov        eax,-1
        jmp        LN2
LN3:
        ;分配内存失败
        mov        eax,-3
        jmp        LN2

LN2:
        ret
InsertNode      endp

;删除结点操作
```

```
;删除带头结点的双链表head上的第i个结点
DeleteNode      proc      uses      esi   ebx   head:ptr    DoubleLinkedList,i:dword
        ;找到第i个结点
        invoke      GetNode,head,i
        cmp         eax,0
        je      returnLine
        mov         ebx,eax

        ;要删除结点的后继结点的指针
        mov         esi,(DoubleLinkedList       ptr     [ebx]).next
        ;要删除结点的前趋结点的指针
        mov         eax,(DoubleLinkedList       ptr     [ebx]).prior
        mov         (DoubleLinkedList   ptr   [eax]).next,esi
        mov         (DoubleLinkedList   ptr   [esi]).prior,eax
        invoke      Free,ebx
        mov         eax,1
        jmp         returnLine
LD1:
        ;没有找到第i个结点的位置
        mov         eax,-1
        jmp         returnLine
;删除尾结点

returnLine:
        ret

DeleteNode      endp

;在双链表链尾创建新结点
;若创建头结点，请将head置0
CreateNode      proc      uses      esi   ebx   head:ptr    DoubleLinkedList,x:dword
        ;若head=0，则创建头指针
        cmp         head,0
        je      createHead

        invoke      GetLastNode,head
        cmp         eax,0
        je          returnLine
        mov         esi,eax
        ;分配新结点的内存空间
        invoke              Alloc,type      DoubleLinkedList
        ;判断分配内存是否成功
        cmp         eax,0
        je      mallocFail
```

```
        mov         ebx,eax
        mov         eax,x
        mov         (DoubleLinkedList    ptr     [ebx]).data,eax
        mov         (DoubleLinkedList    ptr     [ebx]).prior,esi
        mov         eax,head
        mov         (DoubleLinkedList    ptr     [ebx]).next,eax
        mov         (DoubleLinkedList    ptr    [esi]).next,ebx
        jmp         returnLine
createHead:
        ;分配新结点的内存空间
        mov         eax,type   DoubleLinkedList
        invoke            Alloc,eax
        ;判断分配内存是否成功
        cmp         eax,0
        je     mallocFail
        mov         ebx,eax
        mov         eax,x
        mov         (DoubleLinkedList    ptr     [ebx]).data,eax
        mov         (DoubleLinkedList    ptr     [ebx]).prior,ebx
        mov         (DoubleLinkedList    ptr    [ebx]).next,ebx
        mov         eax,ebx
        jmp         returnLine

mallocFail:
        ;分配内存失败
        mov         eax,-3
        jmp         returnLine
returnLine:
        ret

CreateNode      endp
start:
invoke      CreateNode,0,1
mov         headNode,eax
invoke      wsprintf,addr     szText, addr     szCharsFormat, (DoubleLinkedList    ptr    [eax]).data


invoke      MessageBox,NULL,offset        szText,offset     szCaption,MB_OK
invoke      CreateNode,headNode,2
invoke      CreateNode,headNode,5
invoke      CreateNode,headNode,7
;显示前面插入的4个结点
mov         eax,headNode
mov         ebx,(DoubleLinkedList      ptr    [eax]).data
```

```
mov     node1,ebx
mov     eax,(DoubleLinkedList   ptr   [eax]).next
mov     ebx,(DoubleLinkedList   ptr    [eax]).data
mov     node2,ebx
mov     eax,(DoubleLinkedList   ptr   [eax]).next
mov     ebx,(DoubleLinkedList   ptr    [eax]).data
mov     node3,ebx
mov     eax,(DoubleLinkedList   ptr   [eax]).next
mov     ebx,(DoubleLinkedList   ptr   [eax]).data
mov     node4,ebx
invoke  wsprintf,addr   szText, addr   szCharsFormat4, node1,node2,node3,node4

invoke  MessageBox,NULL,offset      szText,offset    szCaption,MB_OK
invoke  InsertNode,headNode,11,3
mov     eax,headNode
mov     ebx,(DoubleLinkedList   ptr   [eax]).data
mov     node1,ebx
mov     eax,(DoubleLinkedList   ptr    [eax]).next
mov     ebx,(DoubleLinkedList   ptr   [eax]).data
mov     node2,ebx
mov     eax,(DoubleLinkedList   ptr   [eax]).next
mov     ebx,(DoubleLinkedList   ptr    [eax]).data
mov     node3,ebx
mov     eax,(DoubleLinkedList   ptr    [eax]).next
mov     ebx,(DoubleLinkedList   ptr   [eax]).data
mov     node4,ebx
mov     eax,(DoubleLinkedList   ptr   [eax]).next
mov     ebx,(DoubleLinkedList   ptr    [eax]).data
mov     node5,ebx
invoke  wsprintf,addr   szText, addr   szCharsFormat5, node1,node2,node3,node4,node5
invoke  MessageBox,NULL,offset      szText,offset    szCaption,MB_OK


invoke  DeleteNode,headNode,3
mov     eax,headNode
mov     ebx,(DoubleLinkedList   ptr    [eax]).data
mov     node1,ebx
mov     eax,(DoubleLinkedList   ptr    [eax]).next
mov     ebx,(DoubleLinkedList   ptr   [eax]).data
mov     node2,ebx
mov     eax,(DoubleLinkedList   ptr   [eax]).next
mov     ebx,(DoubleLinkedList   ptr   [eax]).data
mov     node3,ebx
```

```
mov     eax,(DoubleLinkedList   ptr   [eax]).next
mov     ebx,(DoubleLinkedList   ptr   [eax]).data
mov     node4,ebx
invoke  wsprintf,addr   szText, addr   szCharsFormat24, node1,node2,node3,node4
invoke  MessageBox,NULL,offset      szText,offset    szCaption,MB_OK

invoke  GetNode,headNode,2
invoke  wsprintf,addr   szText, addr   szCharsFormat, (DoubleLinkedList   ptr    [eax]).data
invoke  MessageBox,NULL,offset      szText,offset    szCaption,MB_OK
invoke  LocateNode,headNode,2
invoke  wsprintf,addr   szText, addr   szCharsFormat, (DoubleLinkedList   ptr   [eax]).data
invoke  MessageBox,NULL,offset      szText,offset    szCaption,MB_OK


invoke  ExitProcess,NULL
end  start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=CDLL
OBJS=$(NAME).obj
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
 ml $(ML_FLAG) $<
clean:
    del *.obj
```

定义双链表结点结构的文件如下：

DoubleLinkedList.inc

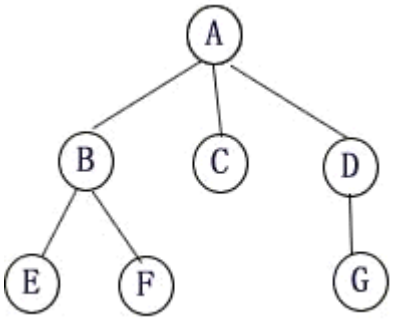
```
DoubleLinkedList        struct
    data        dword       0
    prior       dword       0
    next        dword       0
DoubleLinkedList        ends
```

第四章 树(Tree)

本章重点
树的概念
二叉树的各种运算
一般树与二叉树的转换

4.1 树的概念

树型结构是一种非常重要的非线性数据结构。树型结构是结点之间有分支，并且具有层次关系的结构，它类似于客观世界中的树。
树的示例 4.1(a)：



4.1.1 树的定义

树(Tree)是 n(n>=0) 个结点的有限集 T。T 为空时称为空树。在任一非空树中，必须满足如下两个条件：
（1）有且仅有一个特定的称为根(Root)的结点。

（2）其余结点可分为 m(m>=0)个互不相交的子集 T_1， T_2，…， T_m，其中每个子集本身又是一棵树，并称其为根的子树(Subtree)。

4.1.2 基本术语

一个结点拥有的子树数称为该结点的**度**(Degree)。一棵树的度是指该树中结点的最大度数。度为零的结点称为**叶子**(Leaf)。如图 4.1(a)中，结点 A、B、E 的度分别为 3,2,0，树的度为 3。E、F、C、G 均为叶子。度不为零的结点称为**分支结点**，除根结点之外的分支结点统称为**内部结点**，根结点又称为**开始结点**。

树中某个结点的子树之根称为该结点的**孩子**(Child)，相应地，该结点称为孩子的**双亲**(Parents)。如图 4.1(a)中，B 是结点 A 的孩子，而 A 是 B 的双亲。同一个双亲的孩子称为**兄弟**(Sibling)。如图 4.1(a)中，B、C、D 互为兄弟。结点的**祖先**(Ancestor)是从根到该结点所经

分支上的所有结点。如图 4.1(a) 中，E 的祖先为 A、B。反之，以某结点为根的子树中的任一结点均称为该结点的**子孙**(Descendant)。如图 4.1(a) 中，A 的子孙为 B、E。

结点的**层数**(Level)是从根算起的，根为第一层，其余结点的层数等于其双亲结点的层数加 1。双亲在同一层的结点互为堂兄弟。

若将树中每个结点的各子树看成是从左到右有次序的（即不能互换），则称该树为**有序树**(Ordered Tree)，否则称为**无序树**(Unordered Tree)。**森林**(Forest)是 m(m>=0)棵互不相交的树的集合。

树的逻辑特征可以用树中结点之间的父子关系来描述：树中任一结点都可以有零个或多个孩子，但最多只能有一个双亲。树中只有根没有双亲，它是开始结点。叶子没有孩子。

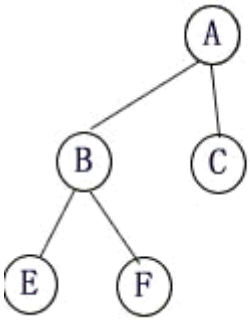
4.2 二叉树(Binary Tree)

二叉树是树形结构的一个重要类型，许多实际问题抽象出来的数据结构往往是二叉树的形式，即使一般树也能简单地转换为二叉树，而且二叉树的存储结构及其算法都较为简单。

4.2.1 二叉树的定义

二叉树是一种每个结点最多只有两棵子树的树形结构。二叉树的子树有左右之分，其次序不能任意颠倒。每个结点最多只能有两棵子树，并且有左右之分。在二叉树中，即使是一个孩子，也有左右之分。

二叉树的示意图 4.2(a)



4.2.2 二叉树的数据结构

由上面的介绍可知，二叉树的每个结点最多只有两个孩子，用链接方式存储二叉树时，每个结点除了存储结点本身的数据外，还应设置两个指针 lchild 和 rchild，分别指向该结点的左孩子和右孩子，结点的结构为：

lchild	data	rchild

二叉树的结点的结构定义为：

```
BinaryTree    struct
    data       dword      0
    lchild     dword      0
    rchild     dword      0
BinaryTree    ends
```

有时，为了便于找到结点的双亲，则还可在结点结构中增加一个指向其双亲结点的指针，结点的数据结构为：

lchild	data	parent	rchild

带双亲指针的二叉树的结点的结构定义为：

```
BinaryTree    struct
    data       dword      0
    lchild     dword      0
    rchild     dword      0
    parent     dword      0
BinaryTree    ends
```

4.2.3 二叉树的遍历

在二叉树的一些应用中，常常需要在树中查找具有某种特征的结点，或者对树中全部结点逐一进行某种处理，这就提出了一个**遍历二叉树**(Traversing Binary Tree)的问题。遍历是指沿着某条搜索路线，依次对树中每个结点均做一次且仅做一次访问。

从二叉树的递归定义可知，一棵非空的二叉树由根结点、左子树、右子树这三个基本部分组成。若能依次遍历这三个部分，即可实现遍历整个二叉树。假如以 N、L、R 分别表示访问根结点，遍历左子树，遍历右子树，则可有六种遍历方案：NLR、LNR、LRN、NRL、RNL、RLN。前三种方案是左子树总是先于右子树被遍历，而后三种方案则正好相反。由于二者是对称的，所以我们只讨论先左后右的前三种方案。

这三种遍历的差别在于访问根结点的次序不同，NLR、LNR 和 LRN 分别表示访问根结点的操作是发生在遍历其左右子树之前，之中（间）和之后。分别称之为前（根）序遍历(Preorder Traversal)、中（根）序遍历(Inorder Traversal)、后（根）序遍历(Postorder Traversal)。

前序遍历的递归算法为：

若二叉树非空，则依次执行如下操作:

（1）访问根结点

（2）前序遍历左子树

（3）前序遍历右子树

```
PreOrder    proc    uses    esi    root:ptr    BinaryTree
    local       j:dword
    ;从根结点开始扫描
    mov     esi,root
    mov     j,0
    ;前序遍历，直到 root 为 NULL
    L3:
    cmp     esi,0
```

```
    je          returnLine
    ;访问根结点
    mov         eax,(BinaryTree     ptr     [esi]).data
    ;显示根结点
    invoke      wsprintf,addr       szText, addr        szCharsFormat, eax
    invoke      MessageBox,NULL,offset      szText,offset   szCaption,MB_OK

    ;前序遍历左子树
    invoke      PreOrder,(BinaryTree    ptr   [esi]).lchild
    ;前序遍历右子树
    invoke      PreOrder,(BinaryTree    ptr   [esi]).rchild
returnLine:
    ret
PreOrder    endp
```

中序遍历的递归算法为：

若二叉树非空，则依次执行如下操作：

（1）中序遍历左子树

（2）访问根结点

（3）中序遍历右子树

```
nOrder  proc    uses        esi   root:ptr   BinaryTree
    local   j:dword
    ;从根结点开始扫描
    mov     esi,root
    mov     j,0
    ;中序遍历，直到 root 为 NULL
    L3:
    cmp     esi,0
    je    returnLine
    ;中序遍历左子树
    invoke    InOrder,(BinaryTree     ptr     [esi]).lchild

    ;访问根结点
    mov     eax,(BinaryTree     ptr   [esi]).data
    ;显示根结点
    invoke    wsprintf,addr       szText, addr        szCharsFormat, eax
    invoke    MessageBox,NULL,offset      szText,offset   szCaption,MB_OK


    ;中序遍历右子树
    invoke    InOrder,(BinaryTree     ptr     [esi]).rchild
returnLine:
    ret
InOrder   endp
```

后序遍历的递归算法为：
若二叉树非空，则依次执行如下操作：
（1）后序遍历左子树
（2）后序遍历右子树
（3）访问根结点

```
PostOrder    proc    uses    esi    root:ptr    BinaryTree
    local    j:dword
    ;从根结点开始扫描
    mov    esi,root
    mov    j,0
    ;后序遍历，直到 root 为 NULL
    L3:
    cmp    esi,0
    je    returnLine
    ;后序遍历左子树
    invoke    PostOrder,(BinaryTree    ptr    [esi]).lchild

    ;后序遍历右子树
    invoke    PostOrder,(BinaryTree    ptr    [esi]).rchild

    ;访问根结点
    mov    eax,(BinaryTree    ptr    [esi]).data
    ;显示根结点
    invoke    wsprintf,addr  szText, addr    szCharsFormat, eax
    invoke    MessageBox,NULL,offset    szText,offset    szCaption,MB_OK

returnLine:
    ret
PostOrder    endp
```

源代码

源代码 BTree**.asm**(路径:Data Structures In ASM Source Code\chapter 4\BinaryTree):

```
.386
.model    flat,stdcall
option    casemap:none
include    windows.inc
include    user32.inc
includelib        user32.lib
include     kernel32.inc
includelib            kernel32.lib
include        BinaryTree.inc
include    masm32.inc
```

```
includelib      masm32.lib
include         ole32.inc
includelib      ole32.lib



.data
szCaption       db          '消息框!',0
szText      db  100     dup(0)
szCharsFormat       db      '结点:%d',0



rootNode        dword   0
parentNode      dword   0
node1       dword   0
node2       dword   0
node3       dword   0
node4       dword   0
node5       dword   0
node6       dword   0
node7       dword   0
.code
;-------------------------------------------------------------

;前序遍历二叉树
PreOrder  proc              uses        esi    root:ptr     BinaryTree
    local       j:dword
    ;从根结点开始扫描
    mov         esi,root
    mov         j,0
    ;前序遍历，直到root为NULL
    L3:
    cmp         esi,0
    je      returnLine
    ;访问根结点
    mov         eax,(BinaryTree       ptr    [esi]).data
    ;显示根结点
    invoke      wsprintf,addr     szText, addr      szCharsFormat, eax
    invoke      MessageBox,NULL,offset        szText,offset     szCaption,MB_OK

    ;前序遍历左子树
    invoke      PreOrder,(BinaryTree      ptr    [esi]).lchild
    ;前序遍历右子树
```

```
        invoke      PreOrder,(BinaryTree      ptr     [esi]).rchild
returnLine:
        ret
PreOrder        endp

;中序遍历二叉树
InOrder    proc              uses       esi    root:ptr     BinaryTree
        local       j:dword
        ;从根结点开始扫描
        mov         esi,root
        mov         j,0
        ;中序遍历，直到root为NULL
        L3:
        cmp         esi,0
        je      returnLine
        ;中序遍历左子树
        invoke      InOrder,(BinaryTree          ptr          [esi]).lchild

        ;访问根结点
        mov         eax,(BinaryTree         ptr  [esi]).data
        ;显示根结点
        invoke      wsprintf,addr    szText, addr     szCharsFormat, eax
        invoke      MessageBox,NULL,offset       szText,offset     szCaption,MB_OK


        ;中序遍历右子树
        invoke      InOrder,(BinaryTree         ptr          [esi]).rchild
returnLine:
        ret
InOrder         endp

;后序遍历二叉树
PostOrder       proc              uses         esi  root:ptr      BinaryTree
        local       j:dword
        ;从根结点开始扫描
        mov         esi,root
        mov         j,0
        ;后序遍历，直到root为NULL
        L3:
        cmp         esi,0
        je          returnLine
        ;后序遍历左子树
        invoke      PostOrder,(BinaryTree       ptr    [esi]).lchild
```

```
    ;后序遍历右子树
    invoke    PostOrder,(BinaryTree    ptr    [esi]).rchild

    ;访问根结点
    mov       eax,(BinaryTree      ptr    [esi]).data
    ;显示根结点
    invoke    wsprintf,addr    szText, addr    szCharsFormat, eax
    invoke    MessageBox,NULL,offset       szText,offset    szCaption,MB_OK

returnLine:
    ret
PostOrder      endp

;创建非根结点
;输入参数:parent--双亲结点,data--数据,child=0，左子结点
;child=1,右子结点
;成功，返回新结点的内存地址，内存分配失败返回-3，
;父节点无效，返回0
CreateNode    proc      uses      esi   ebx   parent:ptr   BinaryTree,data:dword,child:dword
    mov       esi,parent
    cmp       esi,0
    je        parentIsZeroLine
    ;分配新结点的内存空间
    invoke          Alloc,type              BinaryTree
    ;判断分配内存是否成功
    cmp       eax,0
    je    mallocFail
    mov       ebx,eax
    ;将数据写入新结点
    mov       eax,data
    mov       (BinaryTree    ptr   [ebx]).data,eax
    ;判断是添加到左子结点，还是右子结点
    cmp       child,1
    je        rightChild
    mov       (BinaryTree    ptr    [esi]).lchild,ebx
    mov       eax,ebx
    jmp       returnLine
rightChild:
    mov       (BinaryTree    ptr   [esi]).rchild,ebx
    mov       eax,ebx
    jmp       returnLine


;父结点=0
```

```
parentIsZeroLine:
        mov         eax,0
        jmp         returnLine

mallocFail:
        ;分配内存失败
        mov         eax,-3
        jmp         returnLine

returnLine:
        ret
CreateNode      endp

;创建非根结点
;输入参数:data--数据
;成功，返回根结点的内存地址，内存分配失败返回-3
CreateRootNode      proc        uses        esi    ebx         data:dword

        ;分配新结点的内存空间
        invoke      Alloc,type          BinaryTree
        ;判断分配内存是否成功
        cmp         eax,0
        je      mallocFail
        mov         ebx,eax
        ;将数据写入新结点
        mov         eax,data
        mov         (BinaryTree      ptr    [ebx]).data,eax
        mov         eax,ebx
        jmp         returnLine

mallocFail:
        ;分配内存失败
        mov         eax,-3
        jmp         returnLine

returnLine:
        ret
CreateRootNode      endp


start:
;构建一棵二叉树
;           1
;           / \
```

```
;       2   3
;      / \
;     5   7

;创建根结点
invoke      CreateRootNode,1
mov         rootNode,eax
;插入左子树
invoke      CreateNode,rootNode,2,0
mov         parentNode,eax
;插入右子树
invoke      CreateNode,rootNode,3,1
invoke      CreateNode,parentNode,5,0
invoke      CreateNode,parentNode,7,1

;前序遍历
invoke       PreOrder,rootNode
;中序遍历
invoke       InOrder,rootNode
;后序遍历
invoke      PostOrder,rootNode

invoke      ExitProcess,NULL
end         start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=BTree
OBJS=$(NAME).obj
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
 ml $(ML_FLAG) $<
clean:
    del *.obj
```

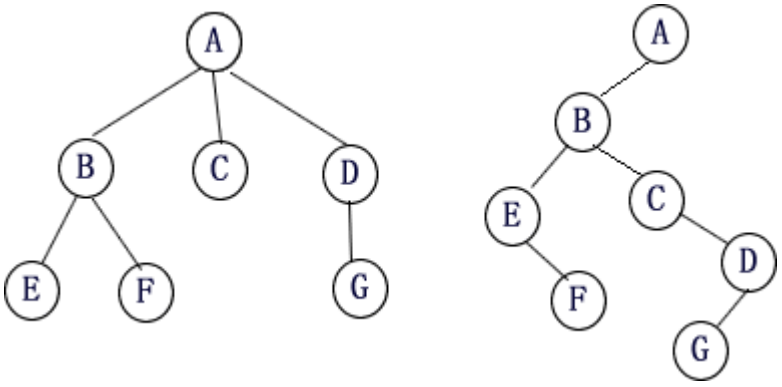
定义二叉树结点结构的文件如下：

BinaryTree.inc

```
BinaryTree          struct
    data        dword       0
    lchild      dword       0
    rchild          dword           0
BinaryTree          ends
```

4.3 多叉树转换为二叉树

现实中，我们遇到的树可能很多都是多叉树，而多叉树在算法上并不容易实现。人们想了一个变通的办法，即在多叉树与二叉树之间建立一种一一对应的关系。简单的说，将一个多叉树的第一个孩子结点作为二叉树的左结点，将其兄弟结点作为二叉树的右结点，即可简单方便地将多叉树转换为二叉树。如下图所示：



源代码

源代码 BTree2**.asm**(路径:Data Structures In ASM Source Code\chapter 4\BinaryTree2):

```
.386
.model      flat,stdcall
option      casemap:none
include     windows.inc
include     user32.inc
includelib          user32.lib
include      kernel32.inc
includelib              kernel32.lib
include             BinaryTree.inc
include     masm32.inc
includelib          masm32.lib
include             ole32.inc
includelib          ole32.lib
```

```
.data
szCaption       db          '消息框!',0
szText      db    100       dup(0)
szCharsFormat       db      '结点:%d',0



rootNode        dword       0
parentNode      dword       0
node1       dword       0
node2       dword       0
node3       dword       0
node4       dword       0
node5       dword       0
node6       dword       0
node7       dword       0
.code
;-------------------------------------------------------------

;前序遍历二叉树
PreOrder  proc              uses        esi    root:ptr     BinaryTree
        local       j:dword
        ;从根结点开始扫描
        mov         esi,root
        mov         j,0
        ;前序遍历，直到root为NULL
        L3:
        cmp         esi,0
        je      returnLine
        ;访问根结点
        mov         eax,(BinaryTree       ptr    [esi]).data
        ;显示根结点
        invoke      wsprintf,addr     szText, addr      szCharsFormat, eax
        invoke      MessageBox,NULL,offset        szText,offset     szCaption,MB_OK

        ;前序遍历左子树
        invoke      PreOrder,(BinaryTree      ptr    [esi]).lchild
        ;前序遍历右子树
        invoke      PreOrder,(BinaryTree      ptr     [esi]).rchild
returnLine:
        ret
PreOrder        endp
```

```
;中序遍历二叉树
InOrder    proc              uses       esi    root:ptr     BinaryTree
        local       j:dword
        ;从根结点开始扫描
        mov         esi,root
        mov         j,0
        ;中序遍历，直到root为NULL
        L3:
        cmp         esi,0
        je      returnLine
        ;中序遍历左子树
        invoke      InOrder,(BinaryTree          ptr          [esi]).lchild

        ;访问根结点
        mov         eax,(BinaryTree        ptr  [esi]).data
        ;显示根结点
        invoke      wsprintf,addr    szText, addr     szCharsFormat, eax
        invoke      MessageBox,NULL,offset       szText,offset     szCaption,MB_OK


        ;中序遍历右子树
        invoke      InOrder,(BinaryTree          ptr          [esi]).rchild
returnLine:
        ret
InOrder          endp

;后序遍历二叉树
PostOrder        proc              uses       esi  root:ptr      BinaryTree
        local       j:dword
        ;从根结点开始扫描
        mov         esi,root
        mov         j,0
        ;后序遍历，直到root为NULL
        L3:
        cmp         esi,0
        je          returnLine
        ;后序遍历左子树
        invoke      PostOrder,(BinaryTree        ptr    [esi]).lchild

        ;后序遍历右子树
        invoke      PostOrder,(BinaryTree     ptr     [esi]).rchild

        ;访问根结点
        mov         eax,(BinaryTree       ptr     [esi]).data
```

```
    ;显示根结点
    invoke    wsprintf,addr    szText, addr    szCharsFormat, eax
    invoke    MessageBox,NULL,offset    szText,offset    szCaption,MB_OK

returnLine:
    ret
PostOrder    endp

;创建非根结点
;输入参数:parent--双亲结点,data--数据,child=0，左子结点
;child=1,右子结点
;成功，返回新结点的内存地址，内存分配失败返回-3，
;父节点无效，返回0
CreateNode    proc    uses    esi    ebx    parent:ptr    BinaryTree,data:dword,child:dword
    mov    esi,parent
    cmp    esi,0
    je    parentIsZeroLine
    ;分配新结点的内存空间
    invoke    Alloc,type    BinaryTree
    ;判断分配内存是否成功
    cmp    eax,0
    je    mallocFail
    mov    ebx,eax
    ;将数据写入新结点
    mov    eax,data
    mov    (BinaryTree    ptr    [ebx]).data,eax
    ;判断是添加到左子结点，还是右子结点
    cmp    child,1
    je    rightChild
    mov    (BinaryTree    ptr    [esi]).lchild,ebx
    mov    eax,ebx
    jmp    returnLine
rightChild:
    mov    (BinaryTree    ptr    [esi]).rchild,ebx
    mov    eax,ebx
    jmp    returnLine


;父结点=0
parentIsZeroLine:
    mov    eax,0
    jmp    returnLine

mallocFail:
```

```
        ;分配内存失败
        mov         eax,-3
        jmp         returnLine

returnLine:
        ret
CreateNode      endp

;创建非根结点
;输入参数:data--数据
;成功，返回根结点的内存地址，内存分配失败返回-3
CreateRootNode      proc        uses        esi    ebx         data:dword

        ;分配新结点的内存空间
        invoke      Alloc,type          BinaryTree
        ;判断分配内存是否成功
        cmp         eax,0
        je      mallocFail
        mov         ebx,eax
        ;将数据写入新结点
        mov         eax,data
        mov         (BinaryTree      ptr    [ebx]).data,eax
        mov         eax,ebx
        jmp         returnLine

mallocFail:
        ;分配内存失败
        mov         eax,-3
        jmp         returnLine

returnLine:
        ret
CreateRootNode      endp


start:
;多叉树
;               1
;          /    |    \
;       2    3     5
;       /
;      7
;转换为二叉树
;       1
```

```
;      /\
;     2  3
;     /   \
;     7    5
;
;创建根结点
invoke      CreateRootNode,1
mov         rootNode,eax
;插入左子树
invoke      CreateNode,rootNode,2,0
mov         parentNode,eax
;插入右子树
invoke      CreateNode,rootNode,3,1
invoke      CreateNode,eax,5,1
invoke      CreateNode,parentNode,7,0


;前序遍历
invoke       PreOrder,rootNode
;中序遍历
invoke       InOrder,rootNode
;后序遍历
invoke      PostOrder,rootNode

invoke      ExitProcess,NULL
end         start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=BTree2
OBJS=$(NAME).obj
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
 ml $(ML_FLAG) $<
clean:
    del *.obj
```

定义二叉树结点结构的文件如下：

BinaryTree.inc

```
BinaryTree          struct
    data        dword          0
    lchild      dword          0
    rchild          dword          0
BinaryTree          ends
```

4.4 二叉排序树(Binary Sort Tree)

二叉排序树(Binary Sort Tree)又称二叉查找(搜索)树(Binary Search Tree)。二叉排序树或者是一棵空树；或者是具有如下性质的二叉树：

（1）若它的左子树不为空，则左子树上所有结点的值均小于它的根结点的值。

（2）若它的右子树不为空，则右子树上所有结点的值均大于它的根结点的值。

（3）它的左、右子树本身也是二叉排序树。

二叉排序树的结点的数据结构为：

```
BinarySortTree      struct
    ;关键字项，在简化的二叉排序树结构中，有时可用 data 字段代替
    key         dword     0
    data        dword     0
    lchild      dword     0
    rchild      dword     0
BinarySortTree      ends
```

4.4.1 二叉排序树的插入

在二叉排序树中，插入新结点，要保证插入后仍满足二叉排序树的性质。其插入过程如下：若二叉排序树为空，则为待插入的关键字申请一个新结点，并令其为根；否则将关键字和根的关键字比较，若二者相等，则说明树中已有此关键字，无须插入；若关键字比根的关键字小，则将关键字插入根的左子树中，否则将它插入右子树中。而子树中的插入过程与上述的树中插入过程相同，如此进行下去，直到将关键字作为一个新的叶子结点的关键字插入到二叉排序树中，或者直到发现树中已有此关键字为止。算法如下：

```
InsertBST       proc uses esi ebx root:ptr BinarySortTree,key:dword,data:dword
    ;esi 指向根结点
    mov esi,root
    ;查找插入位置
L1:
    cmp esi,0
    je    createNewNode
    ;若树中已有 key，无须插入
```

```
        mov eax,key
        cmp (BinarySortTree     ptr   [esi]).key,eax
        je    noInsert
        ;保存当前查找的结点
        mov ebx,esi
        cmp eax,(BinarySortTree      ptr   [esi]).key
        jl    leftChild
        ;在右子树中查找
        mov esi,(BinarySortTree ptr   [esi]).rchild
        jmp  L1
leftChild:
        ;在左子树中查找
        mov esi,(BinarySortTree ptr   [esi]).lchild
        jmp  L1

createNewNode:
        ;分配新结点的内存空间
        invoke    Alloc,type BinarySortTree
        ;判断分配内存是否成功
        cmp eax,0
        je    mallocFail
        mov esi,eax
        ;将数据写入新结点
        mov eax,key
        mov (BinarySortTree ptr [esi]).key,eax
        mov eax,data
        mov (BinarySortTree ptr [esi]).data,eax
        ;原树为空
        cmp root,0
        je    rootNodeLine
        ;原树非空，将新结点作为 ebx 结点的左孩子或右孩子插入
        mov eax,key
        cmp eax,(BinarySortTree ptr [ebx]).key
        jl    L2
        mov (BinarySortTree ptr [ebx]).rchild,esi
        mov eax,esi
        jmp  returnLine

L2:
        mov (BinarySortTree ptr [ebx]).lchild,esi
        mov eax,esi
        jmp  returnLine
```

```
mallocFail:
    ;分配内存失败
    mov eax,-3
    jmp  returnLine
rootNodeLine:
    mov eax,esi
    jmp  returnLine
noInsert:
    ;无须插入
    mov eax,0
    jmp  returnLine

returnLine:
    ret
InsertBST    endp
```

4.4.2 二叉排序树上的查找

为了提高查找效率，本小节提出了一种基于递归的查找算法：

```
SearchBST    proc uses esi r:ptr BinarySortTree,key:dword
    mov esi,r
    cmp esi,0
    je    findEnd
    mov eax,key
    cmp eax,(BinarySortTree ptr [esi]).key
    je    findEnd
    cmp eax,(BinarySortTree ptr [esi]).key
    jl    leftChild
    invoke    SearchBST,(BinarySortTree ptr [esi]).rchild,eax
    ret
leftChild:
    invoke    SearchBST,(BinarySortTree ptr [esi]).lchild,eax
    ret
findEnd:
    mov eax,esi
    ret
SearchBST    endp
```

源代码

源代码 BTree2**.asm**(路径:Data Structures In ASM Source Code\chapter 4\BinaryTree2):

```
.386
.model      flat,stdcall
option      casemap:none
include     windows.inc
include     user32.inc
includelib          user32.lib
include       kernel32.inc
includelib                kernel32.lib
include             BinarySortTree.inc
include     masm32.inc
includelib          masm32.lib
include             ole32.inc
includelib          ole32.lib



.data
szCaption          db          '消息框!',0
szText        db    100         dup(0)
szCharsFormat          db      '结点:%d',0



rootNode          dword       0
parentNode        dword       0
node1       dword       0
node2       dword       0
node3       dword       0
node4       dword       0
node5       dword       0
node6       dword       0
node7       dword       0
.code
;-------------------------------------------------------------

;前序遍历二叉树
PreOrder  proc              uses        esi    root:ptr     BinarySortTree
        local         j:dword
        ;从根结点开始扫描
        mov           esi,root
        mov           j,0
```

```
    ;前序遍历，直到root为NULL
    L3:
    cmp     esi,0
    je    returnLine
    ;访问根结点
    mov     eax,(BinarySortTree     ptr   [esi]).data
    ;显示根结点
    invoke    wsprintf,addr   szText, addr    szCharsFormat, eax
    invoke    MessageBox,NULL,offset      szText,offset    szCaption,MB_OK

    ;前序遍历左子树
    invoke    PreOrder,(BinarySortTree      ptr   [esi]).lchild
    ;前序遍历右子树
    invoke    PreOrder,(BinarySortTree      ptr    [esi]).rchild
returnLine:
    ret
PreOrder      endp

;中序遍历二叉树
InOrder   proc          uses      esi   root:ptr    BinarySortTree
    local     j:dword
    ;从根结点开始扫描
    mov     esi,root
    mov     j,0
    ;中序遍历，直到root为NULL
    L3:
    cmp     esi,0
    je    returnLine
    ;中序遍历左子树
    invoke    InOrder,(BinarySortTree        ptr        [esi]).lchild

    ;访问根结点
    mov     eax,(BinarySortTree   ptr  [esi]).data
    ;显示根结点
    invoke    wsprintf,addr   szText, addr    szCharsFormat, eax
    invoke    MessageBox,NULL,offset      szText,offset    szCaption,MB_OK


    ;中序遍历右子树
    invoke    InOrder,(BinarySortTree        ptr        [esi]).rchild
returnLine:
    ret
InOrder       endp
```

```
;后序遍历二叉树
PostOrder       proc            uses        esi  root:ptr      BinarySortTree
        local       j:dword
        ;从根结点开始扫描
        mov         esi,root
        mov         j,0
        ;后序遍历，直到root为NULL
        L3:
        cmp         esi,0
        je          returnLine
        ;后序遍历左子树
        invoke      PostOrder,(BinarySortTree       ptr    [esi]).lchild

        ;后序遍历右子树
        invoke      PostOrder,(BinarySortTree       ptr    [esi]).rchild

        ;访问根结点
        mov         eax,(BinarySortTree  ptr     [esi]).data
        ;显示根结点
        invoke      wsprintf,addr    szText, addr    szCharsFormat, eax
        invoke      MessageBox,NULL,offset      szText,offset    szCaption,MB_OK

returnLine:
        ret
PostOrder       endp

;插入结点
;输入参数:root--根结点,key--关键字,data--数据
;成功，返回新结点的内存地址，内存分配失败返回-3
;无须插入，返回0
InsertBST       proc        uses        esi        ebx   root:ptr     BinarySortTree,key:dword,data:dword
        ;esi指向根结点
        mov         esi,root
        ;查找插入位置
L1:
        cmp         esi,0
        je          createNewNode
        ;若树中已有key，无须插入
        mov         eax,key
        cmp         (BinarySortTree       ptr    [esi]).key,eax
        je     noInsert
        ;保存当前查找的结点
        mov         ebx,esi
        cmp         eax,(BinarySortTree       ptr    [esi]).key
```

```
        jl      leftChild
        ;在右子树中查找
        mov     esi,(BinarySortTree     ptr     [esi]).rchild
        jmp     L1
leftChild:
        ;在左子树中查找
        mov     esi,(BinarySortTree     ptr     [esi]).lchild
        jmp     L1

createNewNode:
        ;分配新结点的内存空间
        invoke  Alloc,type      BinarySortTree
        ;判断分配内存是否成功
        cmp     eax,0
        je      mallocFail
        mov     esi,eax
        ;将数据写入新结点
        mov     eax,key
        mov     (BinarySortTree     ptr     [esi]).key,eax
        mov     eax,data
        mov     (BinarySortTree     ptr     [esi]).data,eax
        ;原树为空
        cmp     root,0
        je      rootNodeLine
        ;原树非空，将新结点作为ebx结点的左孩子或右孩子插入
        mov     eax,key
        cmp     eax,(BinarySortTree     ptr     [ebx]).key
        jl      L2
        mov     (BinarySortTree     ptr     [ebx]).rchild,esi
        mov     eax,esi
        jmp     returnLine

L2:
        mov     (BinarySortTree     ptr     [ebx]).lchild,esi
        mov     eax,esi
        jmp     returnLine



mallocFail:
        ;分配内存失败
        mov     eax,-3
        jmp     returnLine
```

```
rootNodeLine:
        mov         eax,esi
        jmp         returnLine
noInsert:
        ;无须插入
        mov         eax,0
        jmp         returnLine

returnLine:
        ret
InsertBST       endp

;递归查找结点
SearchBST       proc        uses        esi    r:ptr          BinarySortTree,key:dword
        mov         esi,r
        cmp         esi,0
        je      findEnd
        mov         eax,key
        cmp         eax,(BinarySortTree                ptr    [esi]).key
        je      findEnd
        cmp         eax,(BinarySortTree                ptr     [esi]).key
        jl      leftChild
        invoke      SearchBST,(BinarySortTree      ptr    [esi]).rchild,eax
        ret
leftChild:
        invoke      SearchBST,(BinarySortTree      ptr     [esi]).lchild,eax
        ret
findEnd:
        mov         eax,esi
        ret
SearchBST       endp


start:

;创建根结点
invoke      InsertBST,rootNode,1,1
mov         rootNode,eax
;插入结点
invoke          InsertBST,rootNode,2,2

invoke          InsertBST,rootNode,3,3
invoke          InsertBST,rootNode,5,5
invoke          InsertBST,rootNode,7,7
```

```
;前序遍历
;invoke     PreOrder,rootNode
;中序遍历
invoke      InOrder,rootNode
;后序遍历
;invoke          PostOrder,rootNode
invoke          SearchBST,rootNode,7
mov         eax,(BinarySortTree             ptr     [eax]).data
invoke      wsprintf,addr   szText, addr    szCharsFormat, eax
invoke      MessageBox,NULL,offset      szText,offset   szCaption,MB_OK


invoke      ExitProcess,NULL
end         start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=BST
OBJS=$(NAME).obj
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
 ml $(ML_FLAG) $<
clean:
    del *.obj
```

定义二叉树结点结构的文件如下：

BinarySortTree.inc

```
BinaryTree          struct
    key         dword           0
    data        dword           0
    lchild      dword           0
    rchild          dword           0
BinaryTree          ends
```

第五章 哈希表(Hash Table)

本章重点
哈希表的概念
哈希函数的构造方法
哈希表的冲突处理
哈希表的查找

5.1 哈希表的概念

哈希表(Hash Table)，又称为散列表，是根据关键字而直接进行访问的数据结构。换个说法，它通过映射函数把关键字的值映射到表中的一个位置来直接访问记录。它的优势是查找速度特别的快。这个映射函数叫做**哈希函数**（散列函数,Hash Function）。

我们可以举一个哈希表的最简单的例子。假设有 10 个结点，每结点的关键字恰好为 0-9 之间互不相同的整数，则我们可以将关键字看做是下标，将这 10 个结点存储到一个大小为 10 的数组中。此时，查找任一结点都无须比较，可以直接使用关键字作为下标对数组进行直接寻址，其时间为 O(1)。

对不同的关键字可能得到同一散列地址，即 key1<>key2，而 f(key1)=f(key2)，我们将这种现象称为**冲突**(Collision)或碰撞。而发生冲突的这两个关键字称为该散列函数的 **同义词**(Synonym)。

若对于关键字集合中的任一关键字，经散列函数映射到地址集合中任何一个地址的概率是相等的，则称此类散列函数为**均匀散列函数**(Uniform Hash Table)。

冲突的频繁程度除了与散列函数相关外，还与表的填满程度相关。设 m 和 n 分别表示表长和表中填入的结点数，则将 a=m/n 定义为散列表的**装填因子**(Load Factor)。a 越大，表越满，冲突的机会也越大。通常取 a<=1。

哈希表的结点的数据结构为：

```
HashTable     struct
    key  dword    0
    data dword    0
HashTable     ends
```

5.2 哈希函数的构造方法

哈希函数的选择有两个标准：简单和均匀。简单是指哈希函数的计算简单快速。均匀是指对于关键字集合中的任一关键字，经散列函数映射到地址集合中任何一个地址的概率是相等的，以使冲突最小化。

常用的构造哈希函数的方法有：

5.2.1 直接寻址法

直接寻址法指的是取关键字或关键字的某个线性函数值为哈希地址。即：
f(key)=key 或者 f(key)=a*key+b
其中 a 和 b 为常数（这种哈希函数叫做自身函数）。

例如：有一个 5 个学生的班的语文成绩表，其中，学号作为关键字，哈希函数取关键字自身。如下表所示：

地址	1	2	3	4	5
学号	1	2	3	4	5
语文成绩	25	70	81	63	90

由于直接寻址所得地址集合和关键字集合的大小相同。因此，对于不同的关键字不会发生冲突。

源代码

源代码 HashA.**asm**(路径:Data Structures In ASM Source Code\chapter 5\HashA):

```
.386
.model    flat,stdcall
option    casemap:none
include    windows.inc
include     user32.inc
includelib          user32.lib
include    kernel32.inc
includelib              kernel32.lib
include    HashTable.inc
.data
;哈希表的初始化
HashA    HashTable        7    dup(<0,0>)
szCaption        db     '消息框!',0
szText       db         100  dup(0)
szCharsFormat        db     '哈希表查找:%d,%d',0
.code
;直接寻址法创建哈希表的一个结点
CreateNode    proc              uses        esi    ht:ptr       HashTable,key:dword,data:dword
      mov             esi,ht
      mov          eax,key
      mov             ebx,type          HashTable
      mul         ebx
      add         esi,eax
      ;直接寻址
      mov         eax,key
```

```
        mov         (HashTable    ptr    [esi]).key,eax
        mov             eax,data
        mov         (HashTable      ptr  [esi]).data,eax
        ret
CreateNode          endp

;直接寻址法查找某个结点
;返回data
HashSearch          proc        uses      esi   ht:ptr       HashTable,key:dword
        mov             esi,ht
        mov             eax,key

        mov             ebx,type    HashTable
        mul         ebx
        add         esi,eax
        ;直接寻址
        mov          eax,(HashTable       ptr          [esi]).data
        ret
HashSearch    endp
start:

;初始化
invoke      CreateNode, addr    HashA,1,25
invoke       CreateNode, addr   HashA,2,25
invoke      CreateNode, addr     HashA,3,81
invoke       CreateNode, addr   HashA,4,63
invoke      CreateNode, addr     HashA,5,90
invoke      HashSearch, addr    HashA,3
invoke      wsprintf,addr    szText,addr       szCharsFormat,3,eax
invoke      MessageBox,NULL,offset              szText,offset          szCaption,MB_OK


invoke       ExitProcess,NULL
end         start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=HashA
```

```
OBJS=$(NAME).obj
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
 ml $(ML_FLAG) $<
clean:
    del *.obj
```

定义哈希表结点结构的文件如下：

HashTable.inc

```
HashTable       struct
    key     dword       0
    data    dword       0
HashTable       ends
```

5.2.2 平方取中法

平方取中法指的是取关键字平方之后的值的中间几位做哈希地址。平方的目的是为了扩大近似数的差别。因为一个乘积的中间几位数和乘数的每一位都相关，所以由此产生的哈希地址较为均匀。这是一种较常用的构造哈希函数的方法。

5.2.3 除余法

除余法指的是取关键字被某个不大于哈希表表长 m 的数 p 除后所得余数为哈希地址。即：

f(key)=key　mod　p(p<=m)

这是一种最简单，也最常用的构造哈希函数的方法。它不仅可以对关键字直接取模(mod)，也可在平方取中等运算之后取模。经验表明：一般情况下，可以选 p 为素数，这样可以减少冲突。

源代码

源代码 HashB.**asm**(路径:Data Structures In ASM Source Code\chapter 5\HashB):

```
.386
.model    flat,stdcall
option    casemap:none
```

```
include     windows.inc
include      user32.inc
includelib          user32.lib
include     kernel32.inc
includelib              kernel32.lib
include     HashTable.inc
.data
;哈希表的初始化
HashA     HashTable         7     dup(<0,0>)
szCaption         db      '消息框!',0
szText       db          100   dup(0)
szCharsFormat         db      '哈希表查找:%d,%d',0
.code
;除余法创建哈希表的一个结点
CreateNode     proc              uses        esi    ht:ptr       HashTable,key:dword,data:dword
       mov              esi,ht
       mov          eax,key
       mov         edx,0
       mov         ebx,5
       div         ebx
       ;取余
       mov         eax,edx
       mov              ebx,type          HashTable
       mul         ebx
       add         esi,eax
       ;除余法寻址
       mov         eax,key
       mov          (HashTable     ptr    [esi]).key,eax
       mov              eax,data
       mov          (HashTable      ptr   [esi]).data,eax
       ret
CreateNode           endp

;除余法查找某个结点
;返回data
HashSearch           proc         uses       esi    ht:ptr       HashTable,key:dword
       mov              esi,ht
       mov              eax,key
       mov         edx,0
       mov         ebx,5
       div         ebx
       mov         eax,edx

       mov              ebx,type     HashTable
```

```
    mul     ebx
    add     esi,eax
    ;除余法寻址
    mov     eax,(HashTable ptr [esi]).data
    ret
HashSearch  endp
start:

;初始化
invoke  CreateNode, addr  HashA,1,25
invoke   CreateNode, addr  HashA,2,25
invoke  CreateNode, addr   HashA,3,81
invoke   CreateNode, addr  HashA,4,63
invoke  CreateNode, addr   HashA,5,90
invoke  HashSearch, addr  HashA,3
invoke  wsprintf,addr  szText,addr  szCharsFormat,3,eax
invoke  MessageBox,NULL,offset  szText,offset  szCaption,MB_OK


invoke   ExitProcess,NULL
end     start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=HashB
OBJS=$(NAME).obj
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
 ml $(ML_FLAG) $<
clean:
    del *.obj
```

定义哈希表结点结构的文件如下：

HashTable.inc

```
HashTable       struct
```

```
    key     dword     0
    data    dword     0
HashTable       ends
```

5.3 处理冲突的方法

前文提到均匀的哈希函数可以减少冲突，但不能避免冲突，因此，如何处理冲突是哈希表中一个不可缺少的内容。

常用的处理冲突的方法有三个：开放定址法、再哈希法和拉链法。

5.3.1 开放定址法(Open Addressing)

用开放定址法处理冲突的方法是：当冲突发生时，使用某种探测技术在散列表中形成一个探测序列，沿此序列逐个单元地查找，直到找到给定的关键字、或者碰到一个开放的地址为止（即该地址单元为空，没有被占用）。插入时，探测到开放的地址，则可将待插入的新结点存入该地址单元。查找时，探测到开放的地址，则表明哈希表中没有待查的关键字，即查找失败。

开放定址法的一般形式为：

h_i=(h(key)+ d_i) MOD m　　(1≤i≤m-1)

其中：h(key)为哈希函数，m 为哈希表表长，d_i为增量序列。d_i可以有以下取法：

（1）d_i=1,2,3,…，m-1，称为线性探测法(Linear Probing)。

（2）d_i=1^2, 2^2,…, k^2　　(1≤k^2≤m-1)，称为二次探测法(Quadratic Probing)。

（3）d_i=伪随机数序列，称为伪随机数探测法

通过以上定义中，我们可以推出，在开放定址法中，可能会有多个散列地址不同的结点争夺同一个后继散列地址的现象出现，这种现象被称为堆积(Clustering)。这将造成不是同义词的结点也处在同一个探测序列之中，从而增加了探测序列的长度，也就是增加了查找时间。

5.3.2 再哈希法(Double Hashing)

该方法是开放定址法中最好的方法之一，它的探测序列为：

h_i=(h(key)+i*h1(key)) MOD m　　(0≤i≤m-1)

该方法使用了两个散列函数 h(key)和 h1(key)。

5.3.3 拉链法(Chaining)

拉链法处理冲突的方法是：将所有关键字为同义词的结点链接在同一个单链表中。若选定的散列表长度为 m，则可将散列表定义为一个由 m 个头指针组成的指针数组 T[0,…, m−1]，凡是散列地址为 i 的结点，均插入到以 T[i]为头指针的单链表中。

拉链法有如下优点：

（1）拉链法处理冲突简单，而且没有堆积现象，即非同义词绝不会发生冲突，所以平均查找长度较短，查找速度较快。

（2）由于拉链法中各链表上的结点空间是动态申请的，故它更适合于造表前无法确定表长的情况。

（3）拉链法在结点较大时，与开放定址法相比，要节省空间。

（4）删除结点的操作易于实现。

第六章 图(Graph)

本章重点

图的概念

图的遍历

6.1 图的概念

在语言学、逻辑学、物理、化学、工程、计算机科学、数学等等领域中，图结构有着广泛的应用。

6.1.1 图的定义

图(Graph)是一种复杂的非线性结构。在线性表中，数据元素之间仅有线性关系，每个数据元素只有一个直接前驱和一个直接后继；在树形结构中，数据元素之间是层次关系，即每一层上的数据元素可能和下一层中多个元素（即其孩子结点）相关，但只能和上一层中一个元素（即其双亲结点）相关；在图中，结点之间的关系可以是任意的，图中任意两个数据元素之间都可能相关。

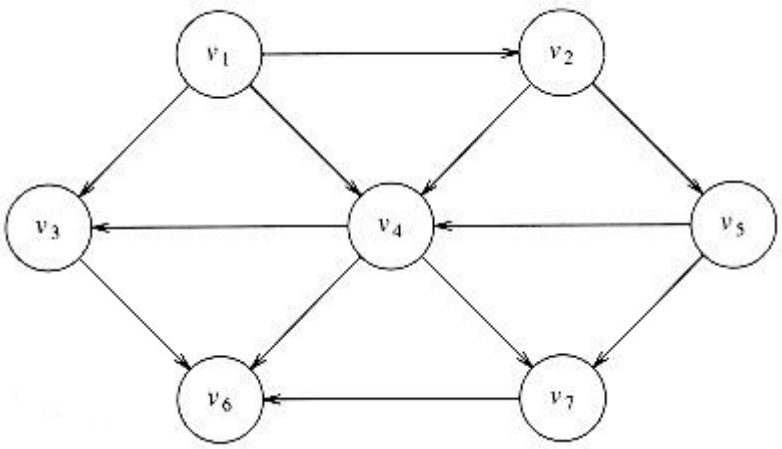
6.1.2 图的术语

假设图为 G，则图 G 由两个集合 V 和 E 组成，记为 G=(V,E)，其中 V 是顶点的有穷非空集合，E 是 V 中顶点偶对（称为边）的有穷集合。通常，也将图 G 的顶点集合和边集合分别记为 V(G)和 E(G)。V(G)不能是空集，E(G)可以是空集，若 E(G)为空，则图 G 只有顶

点而没有边。

在图中的数据元素通常称为**顶点**(Vertex)。将两个顶点关联起来称为**边**(Edge，也称为两个顶点的偶对)。

两个顶点组成的有序对，称为**有向边**，有序对通常用尖括号表示。有向边也称为**弧**(Arc)，边的起点称为**弧尾**(Tail)，边的终点称为**弧头**(Head)。例如，$<v_i, v_j>$表示一条有向边，v_i表示边的起点，v_j表示边的终点。所以$<v_i, v_j>$和$<v_j, v_i>$是两条不同的有向边。若图中每条边均为有向边，则称图为**有向图**(Digraph)。下图G_1是一个有向图，图中边的方向是用从起点指向终点的箭头表示的。

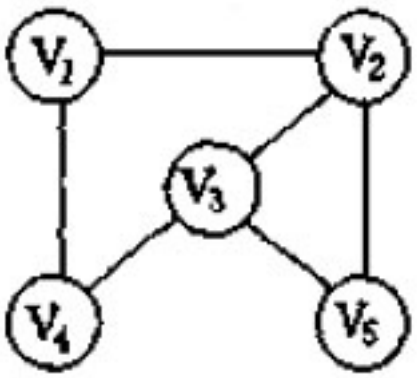


该图的顶点集合和边集合分别为：

$V(G_1)=\{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$

$E(G_1)=\{<v_1, v_2>,<v_1, v_3>,<v_1, v_4>,<v_2, v_4>,<v_2, v_5>,<v_3, v_6>,<v_4, v_3>,<v_4, v_6>,<v_4, v_7>,<v_5, v_4>,<v_5, v_7>,<v_7, v_6>\}$

两个顶点组成的无序对，称为**无向边**，无序对通常用圆括号表示。所以，无序对(v_i, v_j)和(v_j, v_i)表示同一条边。若图中的每一条边都没有方向，则称为**无向图**(Undigraph)。下图G_2是一个无向图。



该图的顶点集合和边集合分别为：

$V(G_2)=\{v_1, v_2, v_3, v_4, v_5\}$

$E(G_2)=\{(v_1, v_2),(v_1, v_4),(v_2, v_3),(v_2, v_5),(v_3, v_4),(v_3, v_5)\}$

我们用 n 表示图中顶点数目，用 e 表示边或弧的数目。在下面的讨论中，我们不考虑顶点到其自身的边，换句话说，我们只讨论简单的图。若 G 是无向图，则 0≤e≤n(n-1)/2。若 G 是有向图，则 0≤e≤n(n-1)。恰好有 n(n-1)/2 条边的无向图称为**无向完全图**(Undirected Complete Graph)。恰好有 n(n-1)条边的有向图称为**有向完全图**(Directed Complete Graph)。若图中边的数目远远小于 n^2，此类图称为**稀疏图**(Sparse Graph)。若图中边的数目接近于 n^2 (准确地说，对于无向图，e 接近于 n(n-1)/2；对于有向图，e 接近于 n(n-1))，此类图称为稠密图(Dense Graph)。

假设有两个图 G=(V,E)和 $G'=(V', E')$，如果 $V' \subseteq V$，而且 $E' \subseteq E$，则 G' 是 G 的**子图**(Subgraph)。

有时，图的边具有与它相关的数，这种与图的边相关的数叫做权(Weight)。权具有意义，比如可以表示从一个顶点到另一个顶点的距离或耗费等等。这种带权的图通常称为**网络**(Network)。

对于无向图，边(v_i, v_j)是一条无向边，则称顶点 v_i 和 v_j 互为**邻接点**(Adjacent)，或称 v_i 和 v_j 相邻接。并称边(v_i, v_j)**依附**或**关联**(Incident)于顶点 v_i 和 v_j，或称边(v_i, v_j)和顶点 v_i 和 v_j 相关联。无向图中，顶点 v 的**度**(Degree)是关联于该顶点的边的数目，记为 D(v)。

对于有向图，边$<v_i, v_j>$是一条有向边，则称顶点 v_i 邻接到 v_j，顶点 v_j 邻接于 v_i。并称边$<v_i, v_j>$关联于 v_i 和 v_j，或称边$<v_i, v_j>$与 v_i 和 v_j 相关联。有向图中，把以顶点 v 为终点的边的数目，称为 v 的**入度**(Indegree)，记为 ID(v)；把以顶点 v 为起点的边的数目，称为 v 的**出度**(Outdegree)，记为 OD(v)。顶点 v 的度则定义为该顶点的入度与出度之和，即

D(v)=ID(v)+OD(v)。

在无向图 G 中，若存在一个顶点序列 v_p，v_{i1}，v_{i2}，…，v_{im}，v_q，使得（v_p，v_{i1}），（v_{i1}，v_{i2}），…，（v_{im}，v_q）均属于 E(G)，则称顶点 v_p 到 v_q 存在一条**路径**(Path)。若 G 是有向的，则路径也是有向的，它由 E(G)中的有向边<v_p，v_{i1}>，<v_{i1}，v_{i2}>，…，<v_{im}，v_q>组成。路径长度定义为该路径上边的数目。若一条路径上，除了 v_p 和 v_q 可以相同外，其余顶点均不相同，则称此路径为一条**简单路径**。起点和终点相同的简单路径称为**简单回路**或**简单环**(Cycle)。

在一个有向图中，若存在一个顶点 v，从该顶点有路径可以到达图中其他所有顶点，则称此有向图为**有根图**，v 称作图的根。

在无向图 G 中，若从顶点 v_i 到 v_j 有路径，则称 v_i 和 v_j 是**连通**的。若 V(G)中任意两个不同的顶点都连通，则称 G 为连通图(Connected Graph)。

无向图 G 的极大连通子图称为 G 的**连通分量**(Connected Component)。显然，任何连通图的连通分量只有一个，即其自身，而非连通的无向图有多个连通分量。

在有向图 G 中，若对于 V(G)中任意两个不同的顶点 v_i 和 v_j，都存在从 v_i 到 v_j 以及从 v_j 到 v_i 的路径，则称 G 是**强连通图**。有向图的极大强连通子图称为 G 的**强连通分量**。

6.2 图的存储结构

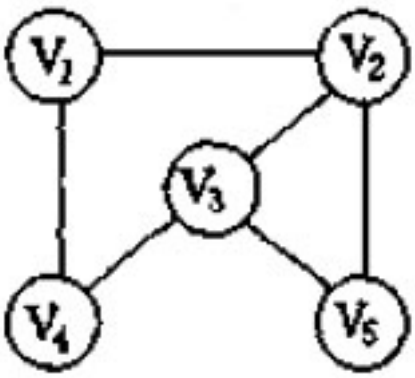
本节介绍两种常用的图的存储结构：邻接矩阵和邻接表

6.2.1 邻接矩阵(Adjacency Matrix)

邻接矩阵(Adjacency Matrix)是表示顶点之间相邻关系的矩阵。设 G=(V,E)是具有 n 个顶点的图，则 G 的邻接矩阵是具有如下性质的 n 阶方阵：

$A[i,j]=\left\{\begin{matrix}1\\0\end{matrix}\right\}$（若($v_i,v_j$)或<$v_i,v_j$>是 E(G)中的边，则 A[i,j]=1；若($v_i,v_j$)或<$v_i,v_j$>不是 E(G)中的边，则 A[i,j]=0）

如下图 G_1：



其邻接矩阵如下：

	v_1	v_2	v_3	v_4	v_5
v_1	0	1	0	1	0
v_2	1	0	1	0	1
v_3	0	1	0	1	1
v_4	1	0	1	0	0
v_5	0	1	1	0	0

也可用矩阵公式表示为：

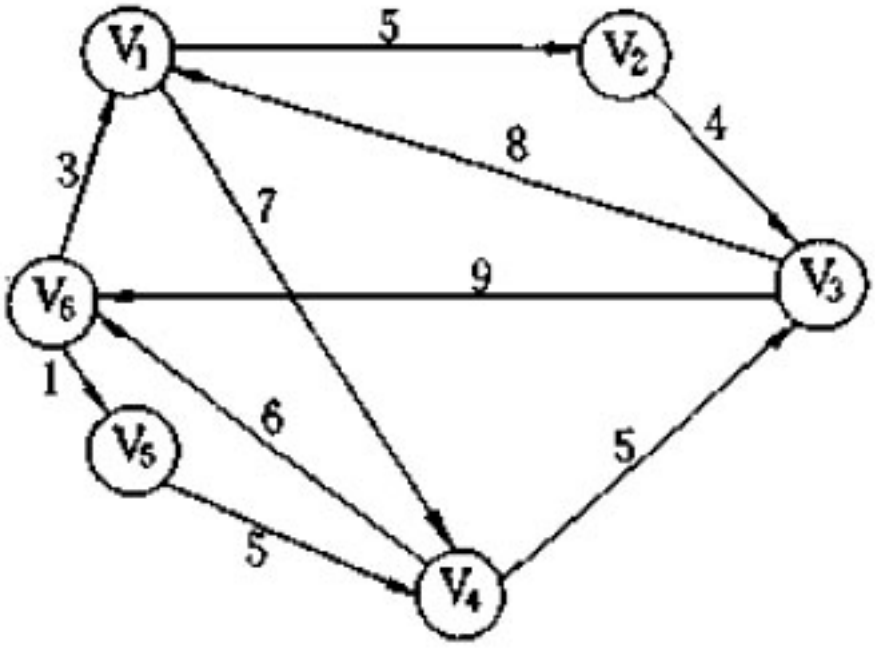
$$A=\begin{Bmatrix}0&1&0&1&0\\1&0&1&0&1\\0&1&0&1&1\\1&0&1&0&0\\0&1&1&0&0\end{Bmatrix}$$

若 G 是网络(Network)，则邻接矩阵可定义为：

A[i,j]=$\begin{Bmatrix} w_{ij} \\ 0或\infty \end{Bmatrix}$（若($v_i, v_j$)或<$v_i, v_j$>是 E(G)中的边，则 A[i,j]= w_{ij}；若(v_i, v_j)或<v_i, v_j>不是 E(G)中的边，则 A[i,j]=0 或∞）

其中， w_{ij} 表示边上的权值；∞表示一个计算机允许的、大于所有边上权值的数。

如下有向网络图：



其邻接矩阵如下：

	v_1	v_2	v_3	v_4	v_5	v_6
v_1	0	0	8	0	0	3
v_2	5	0	0	0	0	0
v_3	0	4	0	5	0	0
v_4	7	0	0	0	5	0
v_5	0	0	0	0	0	1
v_6	0	0	9	6	0	0

也可用矩阵公式表示为：

$$A=\begin{Bmatrix}0&0&8&0&0&3\\5&0&0&0&0&0\\0&4&0&5&0&0\\7&0&0&0&5&0\\0&0&0&0&0&1\\0&0&9&6&0&0\end{Bmatrix}$$

源代码

前节无向图 G_1 源代码 AM1.**asm**(路径:Data Structures In ASM Source Code\chapter 6\AM1):

```
.386
.model    flat,stdcall
option     casemap:none
include    windows.inc
include    user32.inc
```

```
includelib              user32.lib
include     kernel32.inc
includelib              kernel32.lib
include     AdjacencyMatrix.inc
.data
szCaption          db    '消息框!',0
szText      db    100  dup(0)
szCharsFormat          db          '查找边:%d,%d,%d',0
a      AdjacencyMatrix       <100        dup(0)>
.code
;创建边
;am--指向矩阵的指针
;i--行坐标
;j--  列坐标
;w--权重
;maxv--顶点数
CreateEdge      proc                uses          esi  ebx\
 am:ptr    AdjacencyMatrix,i:dword,j:dword,w:dword,maxv:dword
       mov           esi,am
       ;计算edge[i,j]
       mov         eax,i
       mov         ebx,maxv
       mul         ebx
       mov         ebx,j
       add         eax,ebx
       mov         ebx,4
       mul         ebx
       mov         ebx,eax
       mov         eax,w
       mov               (AdjacencyMatrix       ptr    [esi]).edge[ebx],eax

       ret
CreateEdge      endp


start:

;初始化
;第1行
invoke      CreateEdge, addr     a,1,1,0,5
invoke      CreateEdge, addr     a,1,2,1,5
invoke      CreateEdge, addr      a,1,3,0,5
invoke      CreateEdge, addr     a,1,4,1,5
invoke      CreateEdge, addr     a,1,5,0,5
```

```
;第2行
invoke      CreateEdge, addr   a,2,1,1,5
invoke      CreateEdge, addr   a,2,2,0,5
invoke      CreateEdge, addr   a,2,3,1,5
invoke     CreateEdge, addr    a,2,4,0,5
invoke      CreateEdge, addr   a,2,5,1,5

;第3行
invoke      CreateEdge, addr   a,3,1,0,5
invoke      CreateEdge, addr   a,3,2,1,5
invoke      CreateEdge, addr   a,3,3,0,5
invoke     CreateEdge, addr    a,3,4,1,5
invoke      CreateEdge, addr   a,3,5,1,5

;第4行
invoke     CreateEdge, addr     a,4,1,1,5
invoke      CreateEdge, addr   a,4,2,0,5
invoke     CreateEdge, addr    a,4,3,1,5
invoke      CreateEdge, addr   a,4,4,0,5
invoke     CreateEdge, addr    a,4,5,0,5

;第5行
invoke     CreateEdge, addr     a,5,1,0,5
invoke      CreateEdge, addr   a,5,2,1,5
invoke     CreateEdge, addr    a,5,3,1,5
invoke     CreateEdge, addr    a,5,4,0,5
invoke     CreateEdge, addr    a,5,5,0,5
mov             esi,offset          a
;行号
mov        eax,1
;顶点数
mov        ebx,5
mul        ebx
;列号
add        eax,2
mov        ebx,4
mul        ebx
mov        ebx,eax

invoke     wsprintf,addr    szText,addr      szCharsFormat,1,2,\
(AdjacencyMatrix       ptr    [esi]).edge[ebx]
invoke     MessageBox,NULL,offset              szText,offset     szCaption,MB_OK
```

```
invoke      ExitProcess,NULL
end  start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=AM1
OBJS=$(NAME).obj
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
 ml $(ML_FLAG) $<
clean:
    del *.obj
```

定义图结构的文件如下：

AdjacencyMatrix.inc

```
AdjacencyMatrix          struct
    ;邻接矩阵，即边表
    edge      dword          100  dup(0)
    data      dword          0
AdjacencyMatrix          ends
```

有向图的源代码与无向图的一样，故省略。

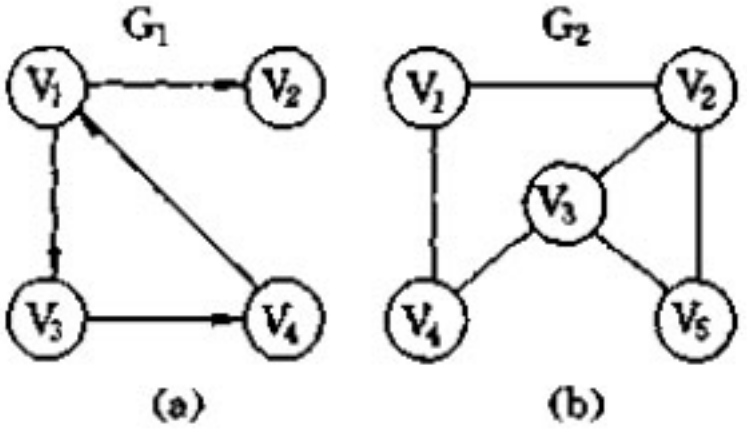
6.2.2 邻接表(Adjacency List)

邻接表(Adjacency List)是图的一种链式存储结构。在邻接表中，对图中的每个顶点建立一个单链表，第 i 个单链表中的结点表示依附于顶点 v_i 的边（对于有向图是以顶点 v_i 为尾的弧）。邻接表中每个表结点均有两个域，其一是邻接点域 adjvex，用以存放与 v_i 相邻接的顶点 v_j 的序号 j。其二是链域 next，用来将邻接表的所有表结点链接在一起。若要表示边上的信息（如权值），则在表结点中还应增加一个数据域。再为每个顶点 v_i 的邻接表设置一个头结点，头结点包含两个域，其中一个是顶点域 vertex，用来存放顶点 v_i 的信息，另一个是指针域 firstedge，它是 v_i 的邻接表的头指针。为了便于随机访问任一顶点的邻接表，我们可以将所有头结点顺序存储在一个数组中。

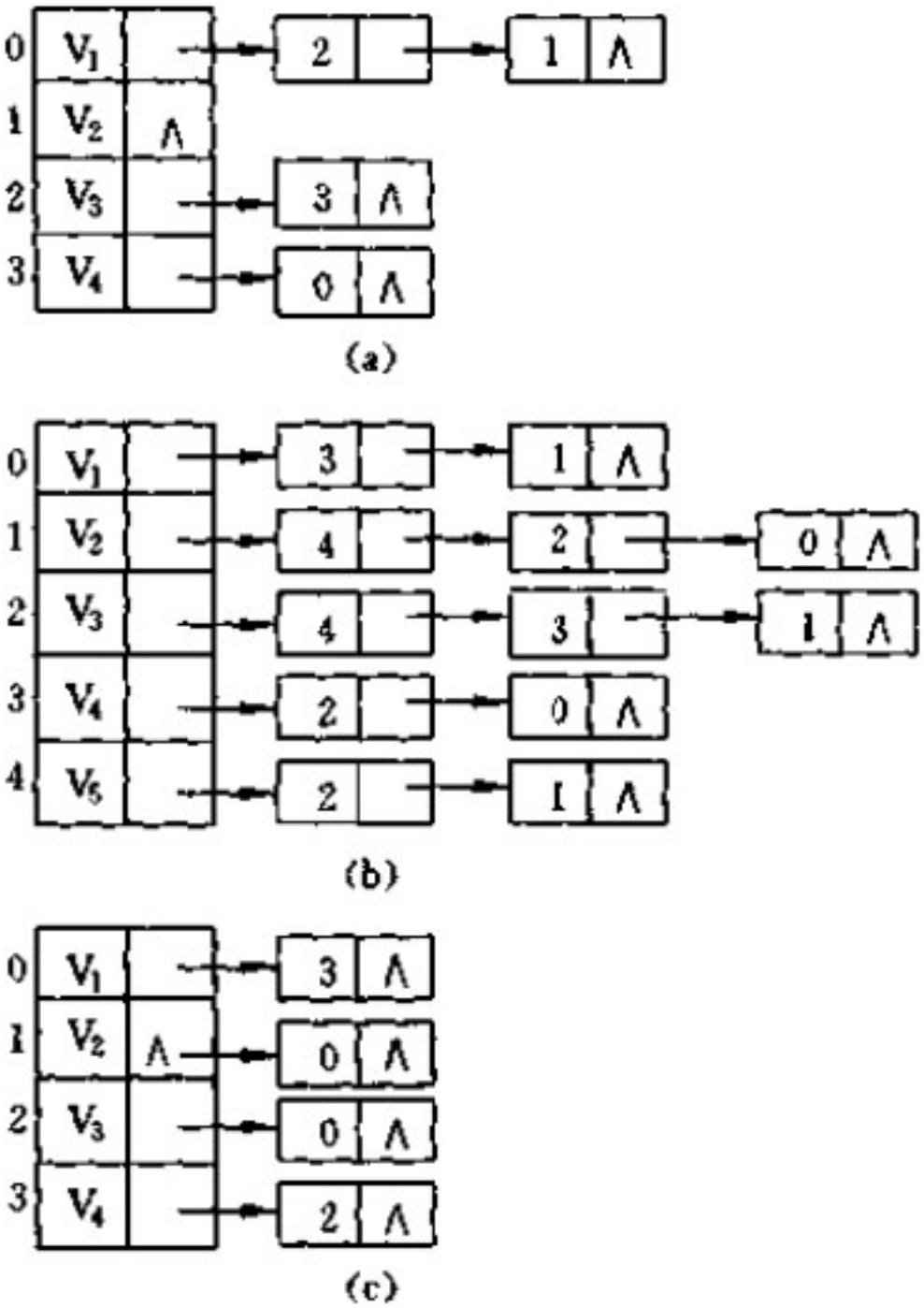
在无向图中，顶点 v_i 的邻接表中每个表结点都对应于与 v_i 相关联的一条边，我们将无向图的邻接表称为**边表**。在有向图中， v_i 的邻接表中每个表结点都对应于以 v_i 为起点射出的一条边，我们将有向图的邻接表称为**出边表**。我们将邻接表的表头数组称为**顶点表**。

有向图还有一种称为逆邻接表的表示法，该方法为图中每个顶点 v_i 建立一个**入边表**，入边表中的每个表结点均对应一条以 v_i 为终点（即射入 v_i ）的边。

下图给出了 G_1 和 G_2 :



下面是 G_1 的邻接表(a)和逆邻接表(c)， G_2 的邻接表(b)：



源代码

G_1 源代码 AL1.**asm**(路径:Data Structures In ASM Source Code\chapter 6\AL1):

```
.386
.model    flat,stdcall
option    casemap:none
```

```
include     windows.inc
include     user32.inc
includelib              user32.lib
include     kernel32.inc
includelib              kernel32.lib
include     AdjacencyList.inc
include     masm32.inc
includelib          masm32.lib
include             ole32.inc
includelib          ole32.lib
.data
szCaption           db      '消息框!',0
szText      db      100     dup(0)
szCharsFormat           db      '查找顶点:%d,%d',0
a       vnode       100   dup(<0,0>)
.code
;创建顶点表
;al--指向邻接表顶点表数组的指针
;i--第i个顶点
;data--  顶点信息

CreateVertex    proc                uses        esi     ebx     alist:ptr   vnode,i:dword,data:dword
        mov             esi,alist
        ;计算vnode[i]
        mov         eax,i
        mov         ebx,type    vnode
        mul         ebx
        add         esi,eax
        mov         eax,data
        ;写入顶点信息
        mov             (vnode      ptr             [esi]).vertex,eax
        ;边表置为空表
        mov             (vnode      ptr      [esi]).firstedge,eax
        ret
CreateVertex      endp

;创建边表结点表
;alist--指向邻接表顶点表数组的指针
;i--第i行
;j--第j列
;maxv--顶点数
CreateEdge      proc                uses        esi     ebx   alist:ptr           vnode,i:dword,j:dword,maxv:dword
        local       newNode:dword
        ;分配新结点的内存空间
```

```
        invoke      Alloc,type          node
        ;判断分配内存是否成功
        cmp         eax,0
        je      mallocFail
        mov         ebx,eax
        ;将数据写入新结点
        mov         eax,j
        mov         (node       ptr     [ebx]).adjvex,eax
        mov         newNode,ebx

        mov                 esi,alist
        ;计算vnode[i]
        mov         eax,i
        mov         ebx,type    vnode
        mul         ebx
        add         esi,eax
        ;读边表的头指针
        mov          eax,(vnode     ptr    [esi]).firstedge
        mov                 ebx,newNode
        mov         (node       ptr     [ebx]).next,eax
        ;将新结点插入顶点i的边表的头部
        mov                 (vnode      ptr     [esi]).firstedge,ebx
        mov         eax,1
        jmp         returnLine

        mallocFail:
        ;分配内存失败
        mov         eax,-3
        jmp         returnLine

returnLine:
        ret
CreateEdge      endp




start:

;初始化顶点
invoke      CreateVertex, addr   a,0,1
invoke       CreateVertex, addr              a,1,2
invoke      CreateVertex, addr   a,2,3
invoke      CreateVertex, addr   a,3,4
```

```
;建立边表
invoke      CreateEdge, addr   a,0,1,4
invoke     CreateEdge, addr     a,0,2,4
invoke     CreateEdge, addr    a,2,3,4
invoke      CreateEdge, addr   a,3,0,4
mov        esi,offset  a

invoke     wsprintf,addr    szText,addr     szCharsFormat,0,(vnode  ptr   [esi]).vertex
invoke      MessageBox,NULL,offset      szText,offset     szCaption,MB_OK


invoke     ExitProcess,NULL
end        start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=AL1
OBJS=$(NAME).obj
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
 ml $(ML_FLAG) $<
clean:
    del *.obj
```

定义图结构的文件如下：

AdjacencyList.inc

```
;邻接表边表结点
nodestruct
    ;邻接点域
    adjvex dword   0
    ;链域
    next  dword     0
node       ends

;顶点表的结点
vnode      struct
    ;顶点域
    vertex      dword      0
    ;边表头指针
```

```
    firstedge   dword      0
vnode       ends
```

6.2.3 邻接矩阵和邻接表的比较

在稀疏图中，用邻接表表示图比用邻接矩阵节省存储空间。在稠密图中，应当用邻接矩阵为宜。在邻接矩阵中，很容易判断(v_i, v_j)和<v_i, v_j>是否是图的一条边，只要判定矩阵中的第 i 行第 j 列上的那个元素是否为零即可。但是在邻接表中，需扫描第 i 个边表，最坏情况下要耗费 O(n)时间。

6.3 图的遍历(Traversing Graph)

和树的遍历类似，图的**遍历**（Traversing Graph）也是从某个顶点出发，沿着某条搜索路径对图中每个顶点各做一次且仅做一次访问。图的遍历算法是求解图的连通性问题、拓扑排序和求关键路径等等算法的基础。

图的遍历比树遍历复杂得多。因为图的任一顶点都可能和其余的顶点相邻接。所以在访问了某个顶点之后，可能沿着某条路径搜索之后，又回到该顶点上。为了避免同一丁点被访问多次，在遍历图的过程中，必须记下每个已访问过的顶点。为此可以设计一个数组 visited[0…n-1]，它的初始值置为“假”或者零，一旦访问了顶点 v_i之后，便将 visited[i]置为真或者被访问时的次序号。

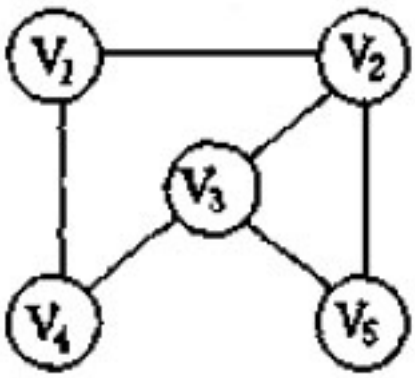
通常有两种遍历图的方法：深度优先遍历和广度优先遍历。它们对无向图和有向图都适用。

6.3.1 深度优先遍历(Depth-First Traversal)

图的**深度优先遍历**类似于树的前序遍历。是树的前序遍历的推广。

假设图 G 的初始状态是所有顶点均未曾访问过，在 G 中任选一顶点 v 为初始出发点（源点），则深度优先遍历可定义为如下：首先访问出发点 v，并将其标记为已访问过；然后依次从 v 出发搜索 v 的每个邻接点 w，若 w 未曾访问过，则以 w 为新的出发点继续进行深度优先遍历，直至图中所有和源点 v 有路径相通的顶点均已被访问为止；若此时图中仍有未访问的顶点，则另选一个尚未访问的顶点作为新的源点重复上述过程，直至图中所有顶点均已被访问为止。

如下图：



其深度优先遍历的源代码如下：

源代码

源代码 DT.**asm**(路径:Data Structures In ASM Source Code\chapter 6\DT):

```
.386
.model      flat,stdcall
option        casemap:none
include     windows.inc
include     user32.inc
includelib                  user32.lib
include     kernel32.inc
includelib                  kernel32.lib
include     AdjacencyMatrix.inc
.data
szCaption          db    '消息框!',0
szText      db    100   dup(0)
szCharsFormat          db          '深度优先遍历顶点:%d',0

a      AdjacencyMatrix      <100        dup(0)>
visited      byte  10    dup(0)

.code
;创建边
;am--指向矩阵的指针
;i--行坐标
;j--  列坐标
;w--权重
;maxv--顶点数
CreateEdge      proc                uses          esi  ebx\
 am:ptr     AdjacencyMatrix,i:dword,j:dword,w:dword,maxv:dword
```

```
        mov         esi,am
        ;计算edge[i,j]
        mov         eax,i
        mov         ebx,maxv
        mul         ebx
        mov         ebx,j
        add         eax,ebx
        mov         ebx,4
        mul         ebx
        mov         ebx,eax
        mov         eax,w
        mov             (AdjacencyMatrix      ptr    [esi]).edge[ebx],eax

        ret
CreateEdge      endp

;以i为出发点对邻接矩阵进行深度优先遍历
;am--指向矩阵的指针
;i--源点
;maxv--顶点数
DFSM    proc            uses        esi    ebx  am:ptr     AdjacencyMatrix,i:dword,maxv:dword
        local       j:dword
        mov         eax,0
        mov         j,eax
        ;显示顶点i,这是简单的矩阵，故省略了顶点表
        invoke      wsprintf,addr    szText, addr     szCharsFormat, i
        invoke      MessageBox,NULL,offset       szText,offset     szCaption,MB_OK
        mov         eax,1
        mov         ebx,i
        mov         esi,offset        visited
        mov         [esi+ebx],eax
        ;依次搜索i的邻接点
        DFSMforLine:
        mov         eax,j
        cmp         eax,maxv
        je      DFSMreturnLine
        mov             esi,am
        ;计算edge[i,j]
        mov         eax,i
        mov         ebx,maxv
        mul         ebx
        mov         ebx,j
        add         eax,ebx
        mov         ebx,4
```

```
        mul         ebx
        mov         ebx,eax
        mov         eax,0
        cmp          (AdjacencyMatrix     ptr         [esi]).edge[ebx],eax
        je     DFSMAddOne
        mov         eax,1
        mov         ebx,j
        mov         esi,offset          visited
        cmp         [esi+ebx],eax
        je     DFSMAddOne
        invoke      DFSM,am,j,maxv
DFSMAddOne:
        inc         j
        jmp         DFSMforLine
DFSMreturnLine:
        ret
DFSM    endp

;深度优先遍历
;am--指向矩阵的指针
;maxv--顶点数
DFSTraverse   proc        uses        esi    ebx   am:ptr      AdjacencyMatrix,maxv:dword
        local       i:dword
        mov         eax,0
        mov         i,eax
        mov               esi,am
        forLine:
        mov         eax,i
        cmp         eax,maxv
        je          DFSTreturnLine
        mov         esi,offset          visited
        add         esi,eax
        mov         ebx,[esi]
        ;访问过该顶点，则跳转到forLine
        cmp         ebx,1
        je     addOne
        invoke      DFSM,esi,i,maxv
        addOne:
        inc     i
        jmp         forLine

DFSTreturnLine:
        ret
```

```
DFSTraverse    endp



start:

;初始化
;第1行
invoke      CreateEdge, addr     a,0,1,0,5
invoke      CreateEdge, addr     a,0,2,1,5
invoke      CreateEdge, addr      a,0,3,0,5
invoke      CreateEdge, addr     a,0,4,1,5
invoke      CreateEdge, addr     a,0,5,0,5

;第2行
invoke       CreateEdge, addr    a,1,1,1,5
invoke       CreateEdge, addr    a,1,2,0,5
invoke       CreateEdge, addr    a,1,3,1,5
invoke      CreateEdge, addr     a,1,4,0,5
invoke       CreateEdge, addr    a,1,5,1,5

;第3行
invoke       CreateEdge, addr    a,2,1,0,5
invoke       CreateEdge, addr    a,2,2,1,5
invoke       CreateEdge, addr    a,2,3,0,5
invoke      CreateEdge, addr     a,2,4,1,5
invoke       CreateEdge, addr    a,2,5,1,5

;第4行
invoke      CreateEdge, addr      a,3,1,1,5
invoke       CreateEdge, addr    a,3,2,0,5
invoke      CreateEdge, addr     a,3,3,1,5
invoke       CreateEdge, addr    a,3,4,0,5
invoke      CreateEdge, addr     a,3,5,0,5

;第5行
invoke      CreateEdge, addr      a,4,1,0,5
invoke       CreateEdge, addr    a,4,2,1,5
invoke      CreateEdge, addr     a,4,3,1,5
invoke      CreateEdge, addr     a,4,4,0,5
invoke      CreateEdge, addr     a,4,5,0,5
invoke      DFSTraverse,addr     a,5
```

```
invoke      ExitProcess,NULL
end  start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=DT
OBJS=$(NAME).obj
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
 ml $(ML_FLAG) $<
clean:
    del *.obj
```

定义图结构的文件如下：

AdjacencyMatrix.inc

```
AdjacencyMatrix          struct
    ;邻接矩阵，即边表
    edge      dword          100  dup(0)
    data      dword          0
AdjacencyMatrix          ends
```

6.3.2 广度优先遍历(Breadth-First Traversal)

图的**广度优先遍历**类似于树的按层次遍历。

假设图 G 的初始状态是所有顶点均未曾访问过，在 G 中任选一顶点 v 为源点，则广度优先遍历可定义为：首先访问源点 v，接着依次访问 v 的所有邻接点 w_1， w_2，…， w_t，然后再依次访问与 w_1， w_2，…， w_t 邻接的所有未曾访问过的顶点，依此类推，直至图中所有和源点 v 有路径相通的顶点都已访问到为止。若此时图中仍有未访问的顶点，则另选一个尚未访问的顶点作为新的源点重复上述过程，直至图中所有顶点均已被访问为止。

源代码

源代码 BT.**asm**(路径:Data Structures In ASM Source Code\chapter 6\BT):

```
.386
.model      flat,stdcall
option      casemap:none
include     windows.inc
include     user32.inc
includelib          user32.lib
include     kernel32.inc
includelib          kernel32.lib
include             AdjacencyMatrix.inc
include             queue.inc
.data
szCaption           db      '消息框!',0
szText      db      100         dup(0)
szCharsFormat       db      '广度优先遍历顶点:%d',0
a       AdjacencyMatrix <100        dup(0)>
visited             byte  10          dup(0)

.code
;置空队
InitQueue           proc        uses        esi         q:ptr       queue
        mov         esi,q
        mov         (queue      ptr     [esi]).front,0
        mov         (queue      ptr     [esi]).rear,0
        mov         (queue      ptr     [esi]).count,0
        ret
InitQueue           endp

;判队空,队空返回1，队非空返回0
QueueEmpty      proc        uses        esi     q:ptr       queue
        mov         esi,q
        mov         eax,(queue      ptr     [esi]).count
        cmp         eax,0
        jg      LE1
        mov         eax,1
        jmp         LE2
LE1:
        mov         eax,0
        jmp         LE2
LE2:
        ret

QueueEmpty          endp

;判队满，队满返回1，队未满返回0
```

```
QueueFull       proc        uses        esi    q:ptr         queue
        mov         esi,q
        mov         eax,(queue        ptr   [esi]).count
        cmp         eax,(queue       ptr     [esi]).queueSize
        jl      LF1
        mov         eax,1
        jmp         LF2
LF1:
        mov          eax,0
        jmp         LF2
LF2:
        ret
QueueFull       endp

;入队 ,队满返回-1，入队成功返回1
EnQueue         proc        uses        esi    ebx   q:ptr        queue,newElement:dword
        mov         esi,q
        ;队满上溢
        invoke      QueueFull,esi
        cmp         eax,1
        je      L1
        ;队列元素个数加1
        inc     (queue      ptr   [esi]).count
        ;新元素插入到队尾
        mov         eax,newElement
        mov         ebx,(queue       ptr     [esi]).rear
        mov         (queue      ptr          [esi]).element[ebx*4],eax
        ;将队尾指针加1
        inc         (queue      ptr    [esi]).rear
        mov         eax,1
        jmp         L2

L1:
        mov         eax,-1
        jmp         L2
L2:
        ret
EnQueue         endp

;出队,队空则返回-1，出队成功返回元素
DeQueue         proc        uses        esi    ebx   q:ptr  queue
        mov         esi,q
        invoke      QueueEmpty,q
        ;队空下溢
```

```
        cmp         eax,1
        je      LD1
        ;取队头元素
        mov         ebx,(queue      ptr     [esi]).front
        mov         eax,(queue      ptr     [esi]).element[ebx*4]
        ;队列元素个数减1
        dec         (queue      ptr     [esi]).count
        ;将队头指针加1
        inc         (queue      ptr     [esi]).front
        jmp         LD2

LD1:
        mov         eax,-1
        jmp         LD2
LD2:
        ret

DeQueue         endp
;创建边
;am--指向矩阵的指针
;i--行坐标
;j-- 列坐标
;w--权重
;maxv--顶点数
CreateEdge      proc        uses        esi     ebx \
am:ptr      AdjacencyMatrix,i:dword,j:dword,w:dword,maxv:dword
        mov         esi,am
        ;计算edge[i,j]
        mov         eax,i
        mov         ebx,maxv
        mul         ebx
        mov         ebx,j
        add         eax,ebx
        mov         ebx,4
        mul         ebx
        mov         ebx,eax
        mov         eax,w
        mov         (AdjacencyMatrix    ptr     [esi]).edge[ebx],eax

        ret
CreateEdge          endp

;以i为出发点对邻接矩阵进行广度优先遍历
;am--指向矩阵的指针
```

```
;i--源点
;maxv--顶点数
BFSM    proc    uses    esi    ebx   am:ptr    AdjacencyMatrix,i:dword,maxv:dword
    local    j:dword
    local    k:dword
    local    QueueA:queue ;声明队列结构的变量

    invoke    InitQueue,addr    QueueA
    mov    eax,0
    mov    j,eax
    ;显示顶点i,这是简单的矩阵，故省略了顶点表
    invoke    wsprintf,addr    szText, addr    szCharsFormat, i
    invoke    MessageBox,NULL,offset    szText,offset    szCaption,MB_OK
    ;设置已访问标记
    mov    eax,1
    mov    ebx,i
    mov    esi,offset    visited
    mov    [esi+ebx],eax

    invoke    EnQueue,addr    QueueA,i


    BFSMWhileLine:
    invoke    QueueEmpty,addr    QueueA
    cmp    eax,1
    je    BFSMreturnLine
    ;出队
    invoke    DeQueue,addr    QueueA
    mov    k,eax
    BFSMforLine:
    mov    ebx,maxv
    cmp    j,ebx
    je    BFSMWhileLine
    ;未访问

    mov    esi,am
    ;计算edge[i,j]
    mov    eax,i
    mov    ebx,maxv
    mul    ebx
    mov    ebx,j
    add    eax,ebx
    mov    ebx,4
    mul    ebx
```

```
        mov     ebx,eax
        mov     eax,0
        cmp     (AdjacencyMatrix    ptr    [esi]).edge[ebx],eax
        je    BFSMAddOne
        mov     eax,1
        mov     ebx,j
        mov     esi,offset      visited
        cmp     [esi+ebx],eax
        je    BFSMforLine
        mov     [esi+ebx],eax
        ;显示顶点j,这是简单的矩阵，故省略了顶点表
        invoke    wsprintf,addr    szText, addr    szCharsFormat, j
        invoke    MessageBox,NULL,offset      szText,offset    szCaption,MB_OK
        invoke    EnQueue,addr      QueueA,j



BFSMAddOne:
        inc     j
        jmp     BFSMforLine
        jmp     BFSMWhileLine
BFSMreturnLine:
        ret
BFSM    endp

;广度优先遍历
;am--指向矩阵的指针
;maxv--顶点数
BFSTraverse   proc      uses      esi   ebx  am:ptr    AdjacencyMatrix,maxv:dword
        local     i:dword
        mov     eax,0
        mov     i,eax
        mov     esi,am
        forLine:
        mov     eax,i
        cmp     eax,maxv
        je    BFSTreturnLine
        mov     esi,offset      visited
        add     esi,eax
        mov     ebx,[esi]
        ;访问过该顶点，则跳转到forLine
        cmp     ebx,1
        je      addOne
        invoke    BFSM,esi,i,maxv
```

```
    addOne:
    inc    i
    jmp        forLine

BFSTreturnLine:
    ret

BFSTraverse        endp



start:

;初始化
;第1行
invoke          CreateEdge, addr    a,0,1,0,5
invoke          CreateEdge, addr    a,0,2,1,5
invoke          CreateEdge, addr    a,0,3,0,5
invoke      CreateEdge, addr    a,0,4,1,5
invoke      CreateEdge, addr    a,0,5,0,5

;第2行
invoke      CreateEdge, addr    a,1,1,1,5
invoke      CreateEdge, addr    a,1,2,0,5
invoke      CreateEdge, addr    a,1,3,1,5
invoke      CreateEdge, addr    a,1,4,0,5
invoke      CreateEdge, addr    a,1,5,1,5

;第3行
invoke      CreateEdge, addr    a,2,1,0,5
invoke      CreateEdge, addr    a,2,2,1,5
invoke      CreateEdge, addr    a,2,3,0,5
invoke      CreateEdge, addr    a,2,4,1,5
invoke      CreateEdge, addr    a,2,5,1,5

;第4行
invoke      CreateEdge, addr    a,3,1,1,5
invoke      CreateEdge, addr    a,3,2,0,5
invoke      CreateEdge, addr    a,3,3,1,5
invoke      CreateEdge, addr    a,3,4,0,5
invoke      CreateEdge, addr    a,3,5,0,5

;第5行
invoke      CreateEdge, addr        a,4,1,0,5
```

```
invoke      CreateEdge, addr     a,4,2,1,5
invoke      CreateEdge, addr     a,4,3,1,5
invoke      CreateEdge, addr     a,4,4,0,5
invoke      CreateEdge, addr     a,4,5,0,5

invoke      BFSTraverse,addr a,5


invoke      ExitProcess,NULL
end         start
```

注：蓝色--关键字 橙色--函数 绿色--变量，运算符，字符串，常量等等 褐色--类型，寄存器 灰色(50%的灰)--注释

MakeFile:

```
NAME=BT
OBJS=$(NAME).obj
LINK_FLAG=/subsystem:windows
ML_FLAG=/c /coff
$(NAME).exe:$(OBJS)
    Link    $(LINK_FLAG) $(OBJS)
.asm.obj:
 ml $(ML_FLAG) $<
clean:
    del *.obj
```

定义图结构的文件如下：

AdjacencyMatrix.inc

```
AdjacencyMatrix          struct
    ;邻接矩阵，即边表
    edge      dword          100  dup(0)
    data      dword          0
AdjacencyMatrix          ends
```

定义队列的文件如下：

queue.inc

```
queue          struct
    front      dword          0
    rear       dword          0
    count          dword          0
    queueSize          dword      5
    element            dword      5      dup(0)
queue          ends
```

参考资料
1. The MASM32 project,MASM32 Library Reference,2007
2. 黄刘生，《数据结构》，经济科学出版社，2000 年 4 月第 1 版
3. Kip R.Irvine,Assembly Language for Intel-Based Computers(Fourth Edition),Pearson Education,Inc. 2003
4. 严蔚敏，吴伟民，《数据结构（C 语言版）》，清华大学出版社，1997 年 4 月第 1 版
5. Randal E. Bryant, David O'Hallaron,Computer Systems: A Programmer's Perspective, Prentice Hall,Inc, 2003
6. Donald E. Knuth, The Art Of Computer Programming,2nd, Addison Wesley Longman,1998
7. 霍红卫，许进，《快速排序算法研究》，《微电子学与计算机》，2002 年第 6 期
8. C.A.R.Hoare,Quicksort,The Computer J.,1962,15(1):10~15
9. 汇编网,http://www.asmedu.net
10. 王增才，《C 语言自学教程》，2009 年
11. 百度百科哈希表词条

附录

32 位通用段寄存器（32-bit General-Purpose Registers）

寄存器名称	特殊用法	英文名称
EAX	1. EAX 在乘法和除法指令中被自动使用。通常称之为扩展累加寄存器。 2. 存储过程的返回值。	Extended accumulator register
EBX		
ECX	CPU 自动使用 ECX 作为循环计数器	
EDX		

寄存器名称	特殊用法	英文名称
EBP		
ESP	ESP 寻址堆栈(一种系统内存结构）上的数据，ESP 寄存器一般绝不应该用于算术运算和数据传送,通常称之为扩展堆栈指针寄存器	Extended stack pointer register
ESI	ESI 和 EDI 通常用于内存数据的高速传送，ESI 称之为扩展源指针寄存器	Extended source index register
EDI	EDI 称之为扩展目的指针寄存器	Extended destination index register

基数后缀(radix)

基数后缀不区分大小写，如果整数常量后面没有后缀，就被认为是十进制的。基数后缀列表如下：

基数后缀	进制
h	十六进制
q/o	八进制
d	十进制
b	二进制
r	编码实数
t	十进制（可选）
y	二进制（可选）

以字母开头的十六进制常量前面必须加一个数字 0，以防止汇编编译器将其解释为标识符。

标号与变量的命名规范

（1）可以用字母、数字、下划线及符号 @、$和?。
（2）第一个符号不能是数字。
（3）长度不能超过 247 个字符。
（4）不能使用指令名等关键字
（5）在作用域内必须是唯一的。
请尽量避免使用@符号作为标号和变量的首字符，因为@符号被编译器扩展用于预定义的符号。

内部数据类型(Intrinsic Data Types)

类型(Type)	缩写	长度(字节)	用途	Usage
BYTE	db	1	8 位无符号整数	8-bit unsigned integer
SBYTE			8 位有符号整数	8-bit signed integer
WORD	dw	2	16 位无符号整数（也可在实地址模式下用做近指针）	16-bit unsigned integer(can also be a Near pointer in Real-address mode)
SWORD			16 位有符号整数	16-bit signed integer

DWORD	dd	4	32 位无符号整数（也可在保护模式下用做近指针）	32-bit unsigned integer(can also be a Near pointer in Protected mode)
SDWORD			32 位有符号整数	32-bit signed integer
FWORD			48 位整数（保护模式下用做远指针）	48-bit integer(Far pointer in Protected mode)
QWORD			64 位整数	64-bit integer
TBYTE			80 位（10 字节）整数	80-bit(10-byte) integer
REAL4			32 位（4 字节）IEEE 短实数	32-bit(4-byte) IEEE short real
REAL8			64 位（8 字节）IEEE 长实数	64-bit(8-byte) IEEE long real
REAL10			80 位（10 字节）IEEE 扩展精度实数	80-bit(10-byte) IEEE extended real

所有使用到变量类型的情况中，只有定义全局变量的时候，类型才可以用缩写。

构建汇编编程环境(MASM32V10)

本文介绍在 Windows 操作系统里怎样搭建 32 位的汇编语言编程环境。
MASM32 是一种非常流行的集成了微软的 MASM 汇编语言编译器的软件包，目前最新版本为 10。0 版。MASM32version10 的下载地址：http://www.masm32.com/masmdl.htm 注意：MASM32 不支持 Win9x 或者 Me。（我想，现在大概也没有人用 Win9x 了吧！）

第一步：

选择一个驱动器安装 MASM32 软件包，例如 C 盘，安装好的目录是 c:\Masm32 目录，对我们来说，整个软件包中重要的只有 3 个目录：bin 目录中有汇编编译器 ml.exe，资源编译器 rc.exe 和链接器 Link.exe 等执行文件；include 目录中有各种头文件；lib 目录中有全部导入库。

第二步：

由于 MASM32 软件包中没有 nmake.exe 文件，所以要单独寻找 nmake.exe 并拷贝到 c:\Masm32\bin 目录中。注意：若我们电脑上安装了 VC++6.0,VC++2005,VC++2008 等多个版本的微软的 C++软件，则应注意 nmake.exe 的版本。

第三步：

为这个环境建立一个设置环境变量的批处理文件，假设文件名为 Var.bat，那么这个文件内容如下：

```
@echo off
```

```
set include=c:\masm32\Include
set lib=c:\masm32\lib
set path=c:\masm32\bin;%path%
echo on
```

文件中设置了 3 个环境变量：

- include 变量指定头文件的搜索目录。在 asm 和 rc 文件中可以根据这个变量寻找 include 语句指定的文件名，避免了使用头文件的全路径名，这样以后移动了 MASM32 的安装位置就不必修改每个源文件中的 include 语句。如果使用 Visual C++的集成环境来建立 rc 文件的话，为了使 rc.exe 能找到头文件，还要把 VC++安装目录下的 Include 和 MFC\Include 目录包含进来（注意：VC++2005,VC++2008 只需要把把 VC++安装目录下的 Include 目录包含进来即可），中间用“;”隔开：

```
set include=c:\masm32\Include;VC 目录\Include;VC 目录\MFC\Include
```

VC++安装目录一般为 C:\Program Files\Microsoft Visual Studio\VC98\。

- lib 变量指定导入库文件的搜索目录。在 asm 文件中可以根据这个变量寻找 includelib 语句指定的导入库文件，Link.exe 也根据这个变量寻找库文件的位置。
- path 变量就不必多解释了。它只是使我们不必在键入命令时带长长的路径而已。

按照上面的步骤安装完成后，下面来编译一个程序测试一下。打开一个文件浏览窗口，切换到源文件目录 c:\Source。打开一个 MS-DOS 窗口，并键入 Var 执行已建立的 Var.bat，这时环境变量和路径已经设置好了，可以键入 SET 命令验证一下 include 和 path 等环境串是否正确，然后键入 c:以及 cd \Source 切换到要工作的目录中，并键入 nmake，当屏幕上出现如下所示的正确的编译链接信息后，Test.exe 就建立完成了。(注：该实例源代码系《Windows 环境下 32 位汇编语言程序设计（第 2 版）》中的源代码)

参考资料

1.罗云彬，《Windows 环境下 32 位汇编语言程序设计（第 2 版）》，电子工业出版社，2003 年

2.MASM32 官网，www.masm32.com

附录

Vc++6.0 环境下的 var.bat

```
@echo off
set include=c:\masm32\Include;d:\Program Files\Microsoft Visual Studio\VC98\include;d:\Program Files\Microsoft Visual Studio\VC98\MFC\Include
set lib=c:\masm32\lib
set path=c:\masm32\bin;%path%
echo on
```

VC++2005 环境下的 var.bat

```
@echo off
set include=c:\masm32\Include;D:\Program Files\Microsoft Visual Studio 8\VC\include
set lib=c:\masm32\lib
set path=c:\masm32\bin;%path%
echo on
```

VC++2008 环境下的 var.bat

```
@echo off
set include=c:\masm32\Include;D:\Program Files\Microsoft Visual Studio 9.0\VC\include
set lib=c:\masm32\lib
set path=c:\masm32\bin;%path%
echo on
```