**数据结构课程设计**

班级：信管一班

姓名：高夕毓

学号：1513490428

日期：2017年5月25日

1. 各题目的编译、运行情况
2. 编译情况：
3. 运行情况：
4. 课程设计内容描述
5. 一元多项式的表示和相加（链表、建立、相加、输出）

<1>解题思路：先建立一个结构体，结构体中包含系数与指数的数据域以及next的指针域。将每个结构体定义为一个节点，在通过指针域链接来建立链表。表示部分，通过从头结点遍历至尾节点的数据域来确定多项式。而相加部分则先利用冒泡排序以指数为基准由小到大将多项式排序，然后由小到大同时遍历两个链表，相同指数则系数相加插入新的链表，不同指数则指数小的先插入链表，大的后插入。最后将剩余节点全部插入新的链表，即完成相加。

<2>函数调用图：

开始

相加放入L3

冒泡排序

表示L1，L2

建立链表L1，L2，L3

结束程序

表示L3

各函数功能:

int createPoly(Poly \*L) 创建多项式链表

int printPolynoimal(Poly \*L) 通过从头结点遍历至尾节点输出链表

int sortPolynoimal(Poly \*L) 对节点进行冒泡排序，由小至大。

int addPolynoimal(Poly \*L1, Poly \*L2, Poly \*L3) 将L1链表和L2链表相加组合成L3链表

int main() 主函数,用来流程控制，建立L1，L2，L3链表

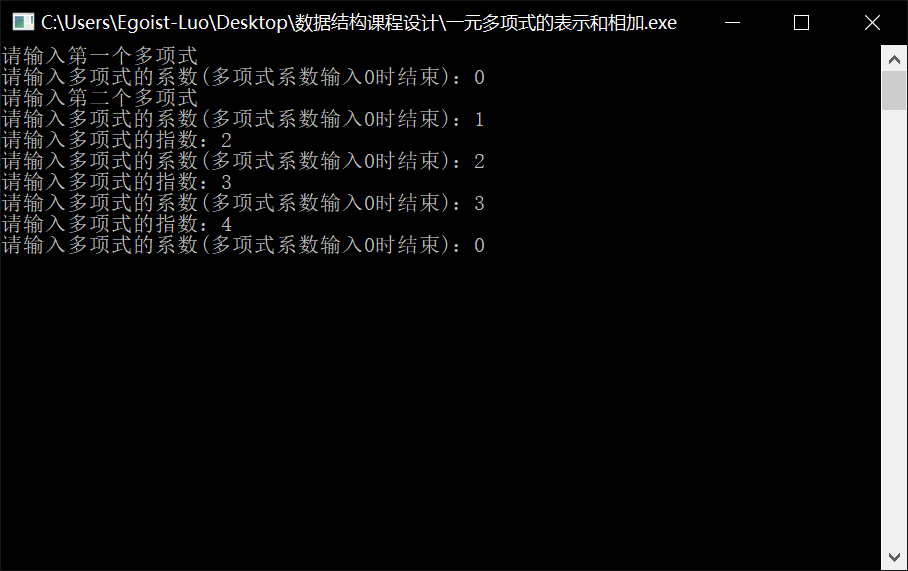
<3>三组测试数据

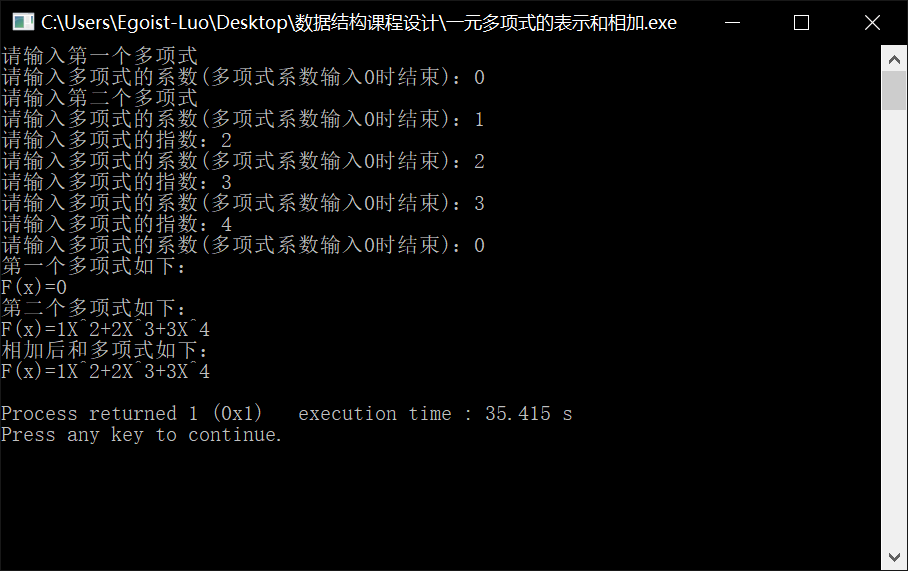
1)原始数据：L1：F(x) = 0

L2：F(x) = 1x^2+2x^3+3x^4

预期结果：L3: F(x) = 1x^2+2x^3+3x^4

实际结果：



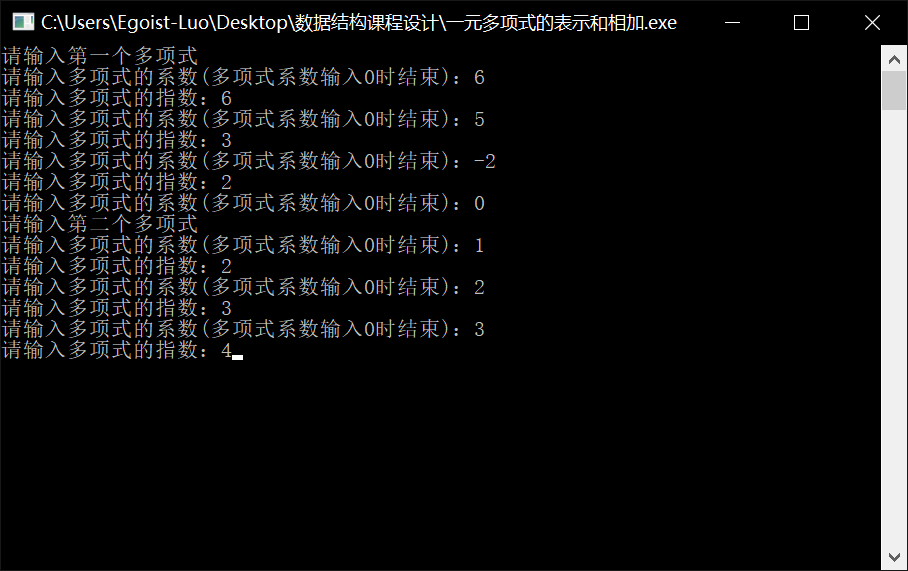


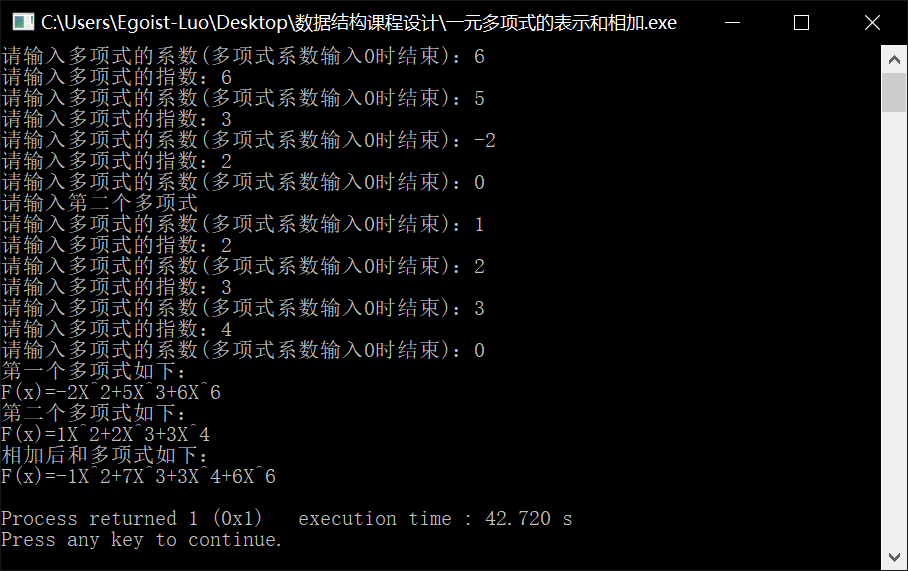
2)原始数据：L1：F(x) = 6x^6+5X^3-2x^2

L2：F(x) = 1x^2+2x^3+3x^4

预期结果：L3: F(x) = -x^2+7x^3+3x^4+6x^6

实际结果：



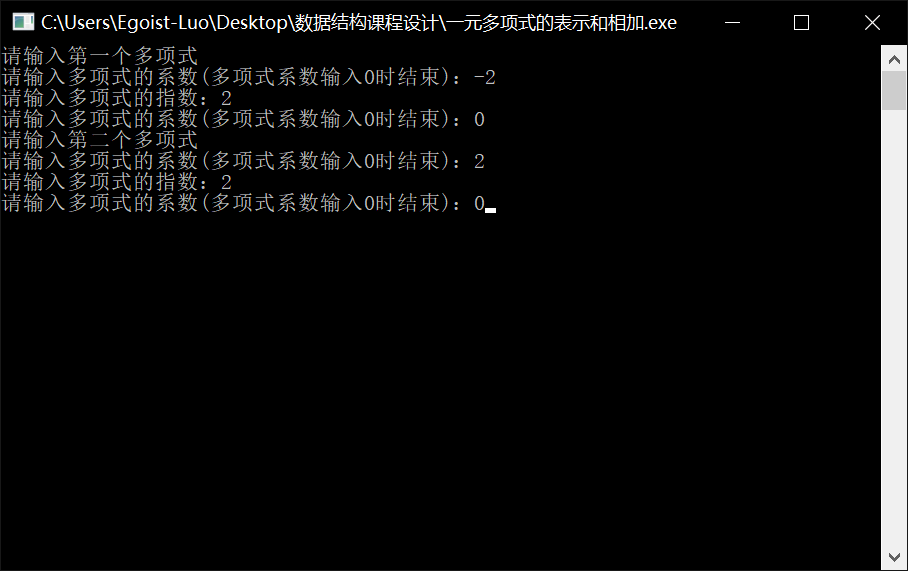


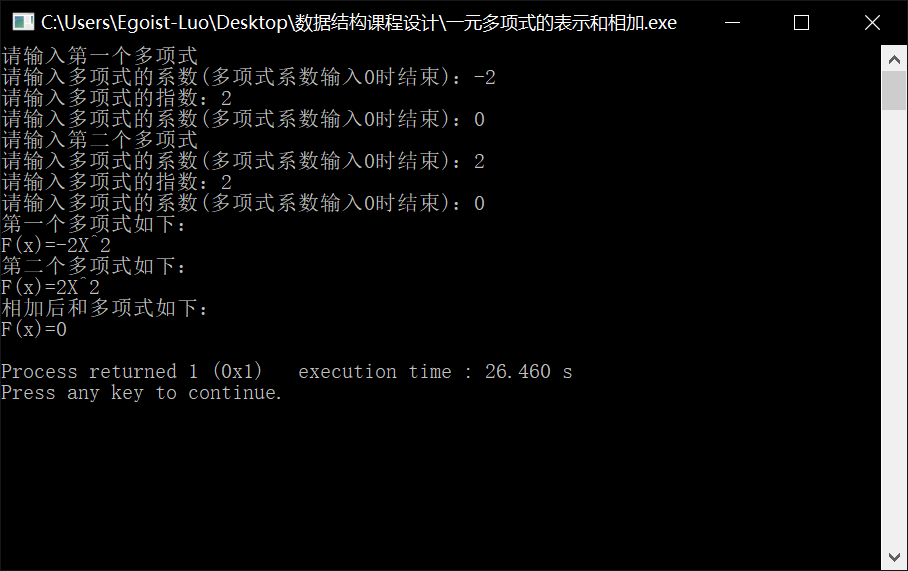
3)原始数据：L1：F(x) = -2x^2

L2：F(x) = 2x^2

预期结果：L3: F(x) = 0

实际结果：





<4>含注释的源程序：

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

typedef struct {

int cof; //系数

int exp; //指数

struct Node \*next; /\*指向下一项的指针\*/

}Poly;

int createPoly(Poly \*L) { /\*创建多项式链表\*/

int x, y; /\*x表示系数，y表示指数\*/

Poly \*q; /\*定义新节点\*/

do {

printf("请输入多项式的系数(多项式系数输入0时结束)：");

scanf("%d", &x);

if (x == 0) { /\*如果系数为0就将指数填入0\*/

y = 0;

} else {

printf("请输入多项式的指数：");

scanf("%d", &y);

}

q = (Poly\*)malloc(sizeof(Poly));

q -> cof = x;

q -> exp = y;

L -> next = q;

L = q;

}while(L -> cof != 0); /\*系数为0就跳出\*/

L -> next = NULL; /\*尾标识为NULL\*/

return 1;

}

int printPolynoimal(Poly \*L) { /\*遍历链表并且输出\*/

Poly \*head;

head = L;

head = head -> next;

if (head -> cof == 0) { /\*如果第一个节点系数是0就说明整个链表为空\*/

printf("F(x)=0\n");

return;

} else {

printf("F(x)=%dX^%d", head -> cof, head -> exp);

head = head -> next;

}

while (head -> cof != 0) {

if (head -> cof > 0) {

printf("+%dX^%d", head -> cof, head -> exp);

head = head -> next;

} else {

printf("%dX^%d", head -> cof, head -> exp);

head = head -> next;

}

}

printf("\n");

return 1;

}

int sortPolynoimal(Poly \*L) { /\*对节点进行排序，仅交换数据\*/

int i, j, tempCof, tempExp, length = 0;

Poly \*head, \*point; /\*辅助指针point帮助指向head的下一个节点\*/

head = L;

head = head -> next;

while (head -> next != NULL) { /\*计算链表长度\*/

length++;

head = head -> next;

}

for (i = 0; i < length - 1; i++) { /\*冒泡排序\*/

head = L; /\*将head复位\*/

head = head -> next;

for(j = i; j < length - 1; j++) {

point = head -> next;

if(head -> exp > point -> exp) {

tempCof = head -> cof;

tempExp = head -> exp;

head -> cof = point -> cof;

head -> exp = point -> exp;

point -> cof = tempCof;

point -> exp = tempExp;

}

head = head -> next;

}

}

return 1;

}

int addPolynoimal(Poly \*L1, Poly \*L2, Poly \*L3) {

int x = 0; /\*x为和多项式每一项的系数\*/

Poly \*headL1, \*headL2, \*headL3; /\*定义和多项式的指针和节点\*/

headL1 = L1;

headL2 = L2;

headL1 = headL1 -> next;

headL2 = headL2 -> next;

while (headL1 -> cof && headL2 -> cof) {

if (headL1 -> exp == headL2 -> exp) { /\*两指数相同时，系数相加\*/

x = headL1 -> cof + headL2 -> cof;

if (x) { /\*系数和不为零就插入到和多项式\*/

headL3 = (Poly\*)malloc(sizeof(Poly));

headL3 -> cof = x;

headL3 -> exp = headL1 -> exp;

L3 -> next = headL3;

L3 = headL3;

headL1 = headL1 -> next;

headL2 = headL2 -> next;

} else { /\*系数和为零就向后移一个节点\*/

headL1 = headL1 -> next;

headL2 = headL2 -> next;

}

} else if (headL1 -> exp < headL2 -> exp){ /\*两指数不相同时，将小的节点插入L3\*/

headL3 = (Poly\*)malloc(sizeof(Poly));

headL3 -> cof = headL1 -> cof;

headL3 -> exp = headL1 -> exp;

L3 -> next = headL3;

L3 = headL3;

headL1 = headL1 -> next;

} else {

headL3 = (Poly\*)malloc(sizeof(Poly));

headL3 -> cof = headL2 -> cof;

headL3 -> exp = headL2 -> exp;

L3 -> next = headL3;

L3 = headL3;

headL2 = headL2 -> next;

}

}

while (headL1 -> cof != 0) { /\*如果L1有剩余即当前节点系数不为0，则将剩下的全部插入L3\*/

headL3 = (Poly\*)malloc(sizeof(Poly));

headL3 -> cof = headL1 -> cof;

headL3 -> exp = headL1 -> exp;

L3 -> next = headL3;

L3 = headL3;

headL1 = headL1 -> next;

}

while (headL2 -> cof != 0) { /\*如果L2有剩余即当前节点系数不为0，则将剩下的全部插入L3\*/

headL3 = (Poly\*)malloc(sizeof(Poly));

headL3 -> cof = headL2 -> cof;

headL3 -> exp = headL2 -> exp;

L3 -> next = headL3;

L3 = headL3;

headL2 = headL2 -> next;

}

headL3 = (Poly\*)malloc(sizeof(Poly)); /\*制造出cof值为0的节点，插入到链表末尾作为标志节点\*/

headL3 -> cof = 0;

L3 -> next = headL3;

return;

}

int main() {

Poly \*L1, \*L2, \*L3;

L1 = (Poly\*)malloc(sizeof(Poly));

L2 = (Poly\*)malloc(sizeof(Poly));

L3 = (Poly\*)malloc(sizeof(Poly));

printf("请输入第一个多项式\n");

createPoly(&L1);

printf("请输入第二个多项式\n");

createPoly(&L2);

sortPolynoimal(&L1);

sortPolynoimal(&L2);

printf("第一个多项式如下：\n");

printPolynoimal(&L1);

printf("第二个多项式如下：\n");

printPolynoimal(&L2);

printf("相加后和多项式如下：\n");

addPolynoimal(&L1, &L2, &L3);

printPolynoimal(&L3);

return 1;

}

<5>编译与运行情况：

编译正常通过，运行正常。

(2)表达式括号匹配检验（压栈、出栈）

<1>解题思路：构造包含顶指针，底指针和增量的结构体。以此建立一个空栈。然后依次读入输入的字符，存放至栈中。最后从栈中依次读出字符。分别设置三种括号的标志。当右括号读出时标志加一，当，左括号读出时标志减一。如果表达式括号匹配，则三种标志位都等于0，如果括号不匹配则标志位不等于0。最后输出结果。

<2>函数调用图：

开始

建立空栈S

是否是回车

入栈

输入字符

不是

是

出栈

是否是括号

得出结论

计算计数器

不是

是

不是

是否出栈完

计数器加/减1

是

结束

各函数功能:

int InitStack (SqStack \*S) 构造一个空栈

int Push (SqStack \*S, char e) 让e入栈

int Pop (SqStack \*S, char \*e) 让e出栈

int isEmpty (SqStack \*S) 判断栈是否为空，空返回0，非空返回1

int match (SqStack \*S, int counter) 判断括号是否匹配，通过计数器计算前后括号数量。如果三个计数器都为0，括号匹配。

int main() 流程控制，并且建立栈S。

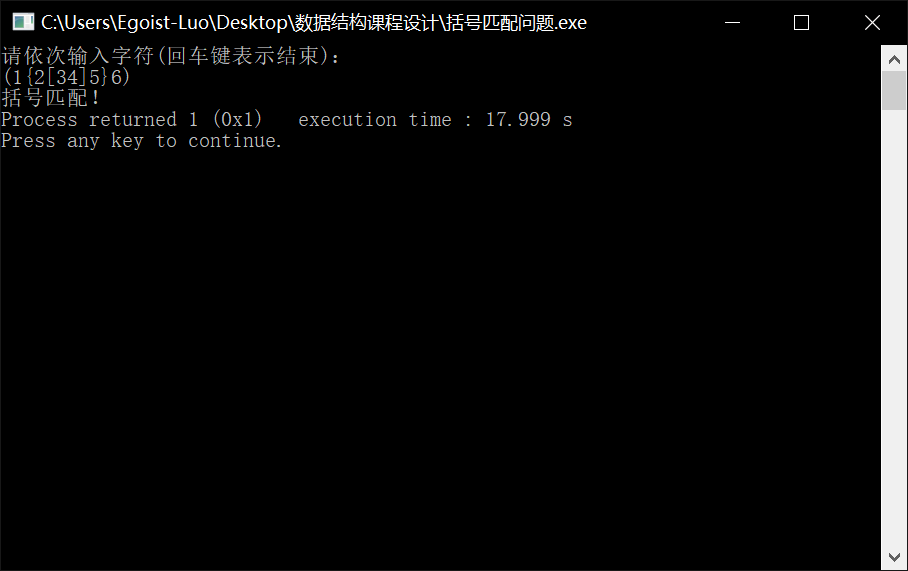
<3>三组测试数据

1)原始数据：(1{2[34]5}6)

预期结果：匹配

实际结果：



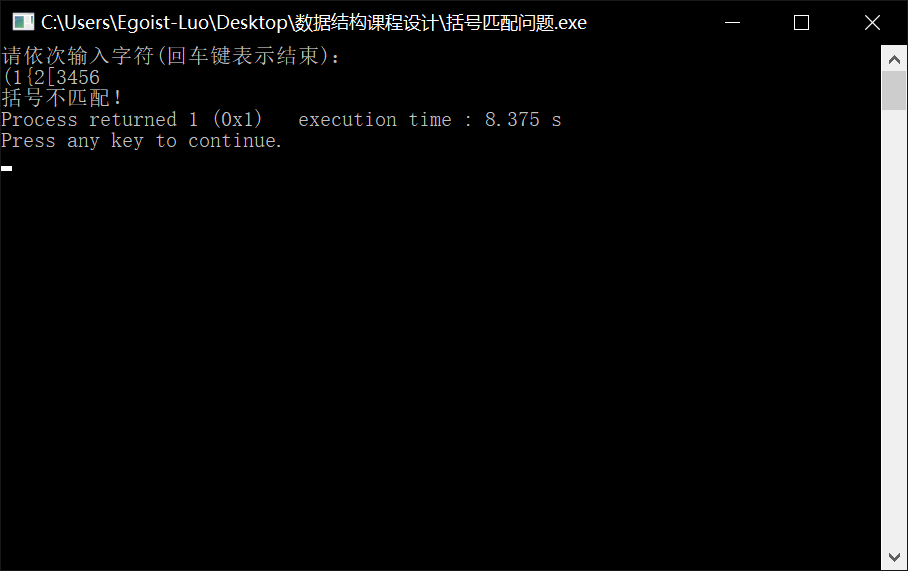


2)原始数据：(1{2[3456

预期结果：不匹配

实际结果：



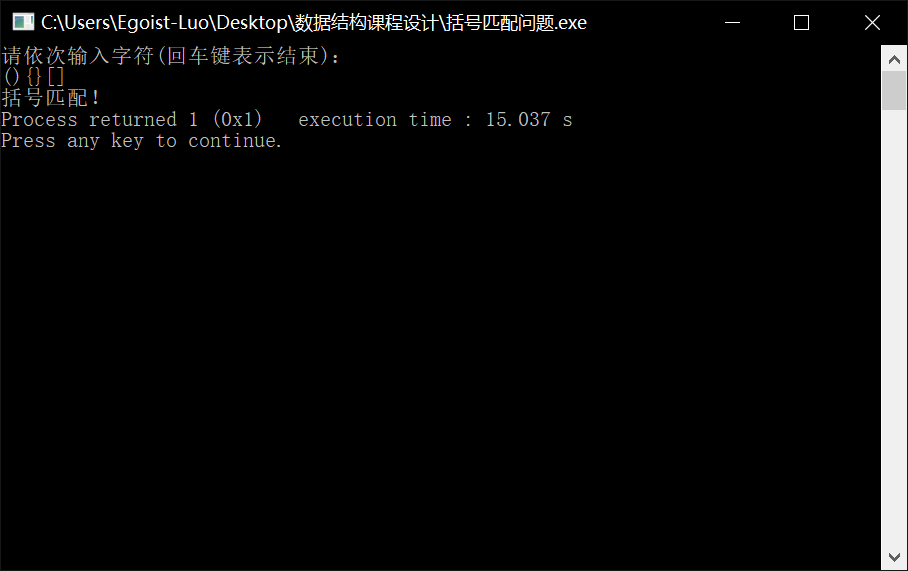


3)原始数据：(){}[]

预期结果：匹配

实际结果：





<4>含注释的源程序：

#include <stdio.h>

#include <stdlib.h>

#include <malloc.h>

#define STACK\_INIT\_SIZE 100 /\*存储空间初始分配量\*/

#define STACKINCREMENT 20 /\*存储空间分配增量\*/

typedef struct {

char \*top;

char \*base;

int stacksize;

}SqStack;

int InitStack (SqStack \*S) { /\*构造一个空栈\*/

S -> base = (char\*)malloc(STACK\_INIT\_SIZE\*sizeof(char));

S -> top = S -> base;

S -> stacksize = STACK\_INIT\_SIZE;

return 1;

}

int Push (SqStack \*S, char e) { /\*插入元素e为新的栈顶元素\*/

if (S -> top - S -> base >= S -> stacksize) {

S -> base = (char\*)realloc(S -> base, (S -> stacksize + STACKINCREMENT)\*sizeof(char));

if (!S -> base) { /\*栈溢出\*/

return -1;

}

S -> top = S -> base + S -> stacksize;

S -> stacksize += STACKINCREMENT;

}

\*(S -> top) = e; /\*先赋值在将指针加1\*/

S -> top ++;

return 1;

}

int Pop (SqStack \*S, char \*e) { /\*退栈\*/

if (S -> top == S -> base) { /\*栈空\*/

return 0;

} else {

S -> top --; /\*先将指针减1在出栈\*/

\*e = \*(S -> top);

}

return 1;

}

int isEmpty (SqStack \*S) { /\*判断栈是否为空\*/

if (S -> top == S -> base) {

return 0;

} else {

return 1;

}

}

int match (SqStack \*S, int counter) { /\*判断括号是否匹配\*/

int i, flag1 = 0, flag2 = 0, flag3 = 0; /\*标志位flag1,2,3, 记录三种括号()[]{}的次数\*/

char e;

if (isEmpty(S) == 0) { /\*判断栈是否为空\*/

printf("输入的字符为空！");

} else { /\*不为空就弹栈\*/

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

Pop(S, &e);

switch (e) { /\*判断括号是否匹配\*/

case '(': flag1--;break;

case ')': flag1++;break;

case '[': flag2--;break;

case ']': flag2++;break;

case '{': flag3--;break;

case '}': flag3++;break;

default: break;

}

}

if ((flag1 == 0 && flag2 == 0) && (flag3 == 0 && (counter != 0))) {

printf("括号匹配！");

} else {

printf("括号不匹配！");

}

}

return 1;

}

int main() {

int counter; /\*定义计数器，记录入栈次数\*/

char c = 0;

SqStack \*S;

InitStack(&S);

printf("请依次输入字符(回车键表示结束)：\n");

while (c != '\n') { /\*当输入的字符是回车时,停止输入\*/

c = getchar();

if (c != '\n') { /\*当输入的字符不是回车时,将其放入栈\*/

Push(&S, c);

counter++;

}

}

match(&S, counter);

return 1;

}

<5>编译与运行情况：

编译正常通过，运行正常。

(3)数制的转换（十进制到二进制转换）

<1>解题思路: 构造包含顶指针，底指针和增量的结构体。以此建立一个空栈。然后输入数据，通过除法和求模运算分别求出整数商和余数，将余数放入栈，用商作为数据，循环上一步的的运算直到商为0，最后将数据出栈，就是转换为二进制的结果。

<2>函数调用图:

开始

建立空栈S

输入数据

将余数入栈

做除法，求模运算

不是

商是否为0

是

依次出栈

输出

结束

各函数功能:

int InitStack (SqStack \*S) 构造一个空栈

int Push (SqStack \*S, char e) 让e入栈

int Pop (SqStack \*S, char \*e) 让e出栈

int Convert (SqStack \*S, int data) 转换函数，将data除2和取模运算，即分别计算出商和余数。接下来继续计算，以商作为data，重复运算。

int main() 流程控制，并且建立栈S。

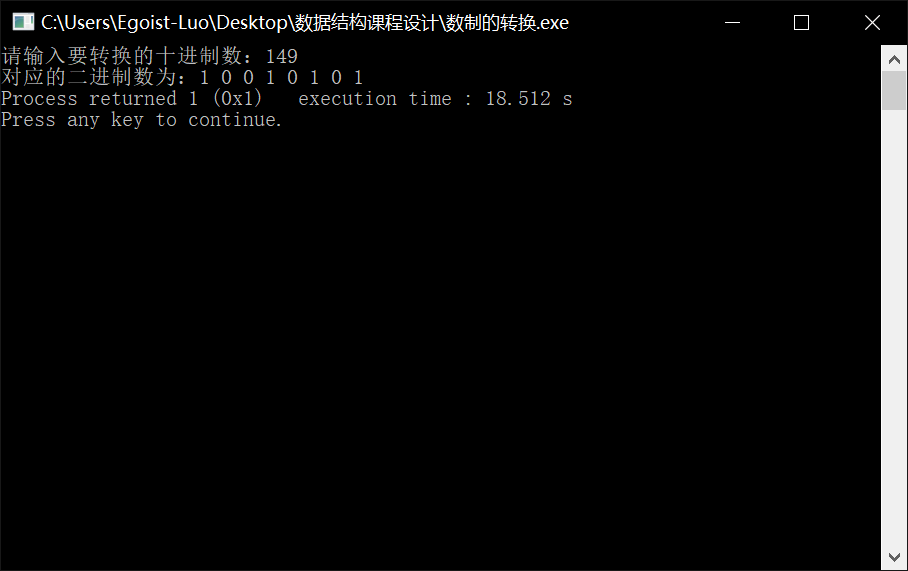
<3>三组测试数据

1)原始数据：149

预期结果：1001 0101

实际结果：



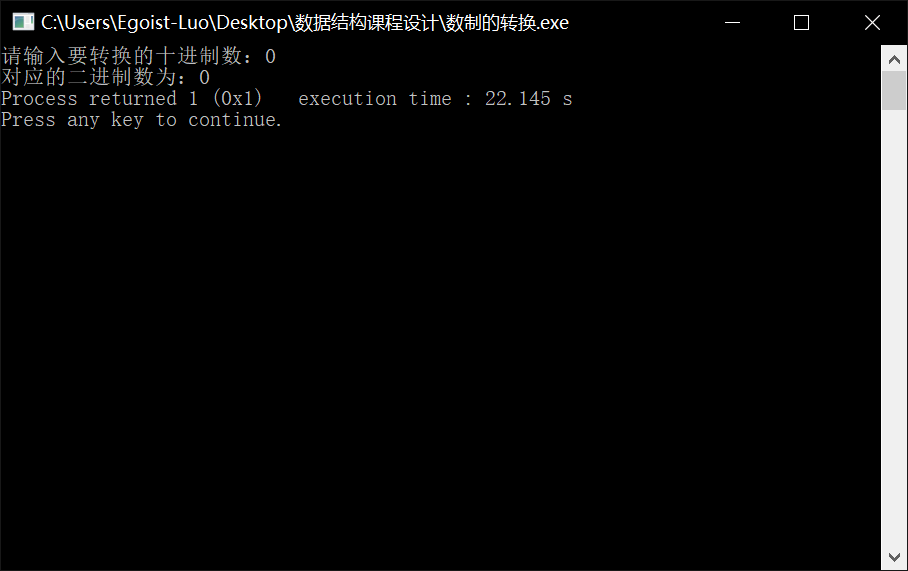


2)原始数据：0

预期结果：0

实际结果：



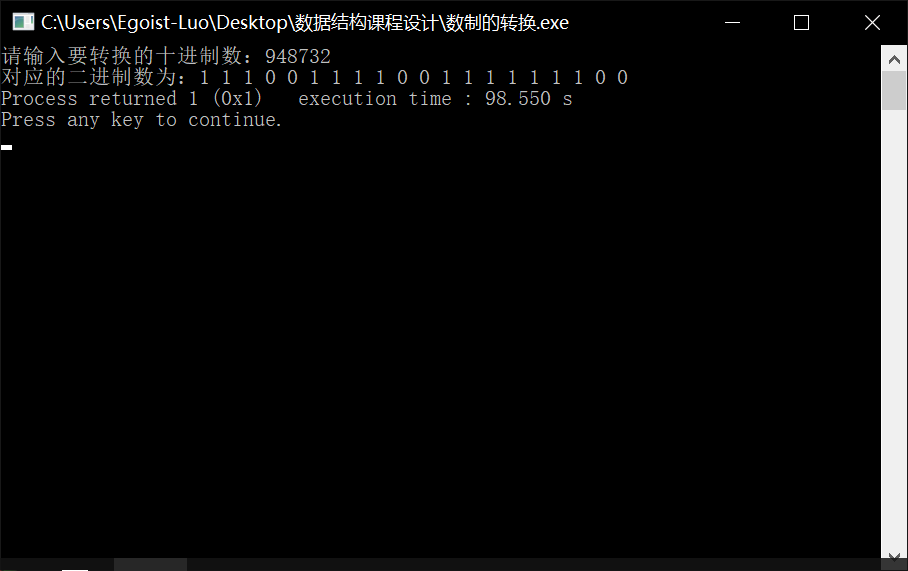


3)原始数据：948732

预期结果：1110 0111 1001 1111 1100

实际结果：





<5>编译与运行情况：

编译正常通过，运行正常。

(4)约瑟夫环（循环单链表）

<1>解题思路：先定义包含一个数字域以及next的指针域的结构体。然后输入节点总数创建链表，最后将链表尾指针指向第一个数据节点使之闭合成为约瑟夫环。接下来输入开始节点和间距。如果间距为1就是依次输出即可。如果间距大于一就相隔间距输出，并且输出后将断环闭合。最后直到约瑟夫环中没有节点即结束。

<2>函数调用图：

开始

创建链表

闭合约瑟夫环

判断间距是否为1

是

不是

间隔输出

连续输出

判断约瑟夫环中是否空

闭环

不是

是

结束

各函数功能:

int create(Node \*L) 创建链表

int process(Node \*L) 实现约瑟夫环，先进行闭环操作，然后根据开始节点和间距进行输出节点操作。

int main() 流程控制，并且建立列表L

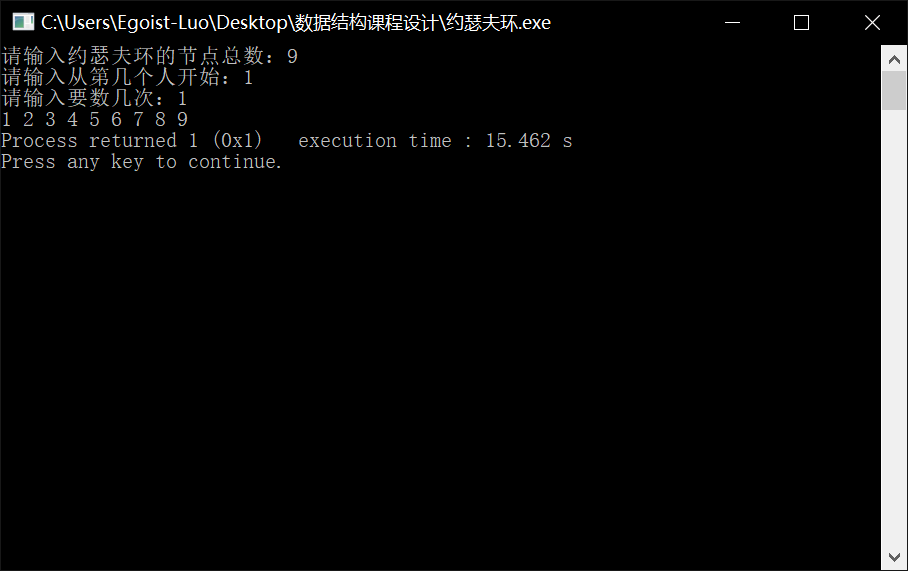
<3>三组测试数据

1)原始数据：节点总数9即123456789 起始1，间隔1

预期结果：123456789

实际结果：



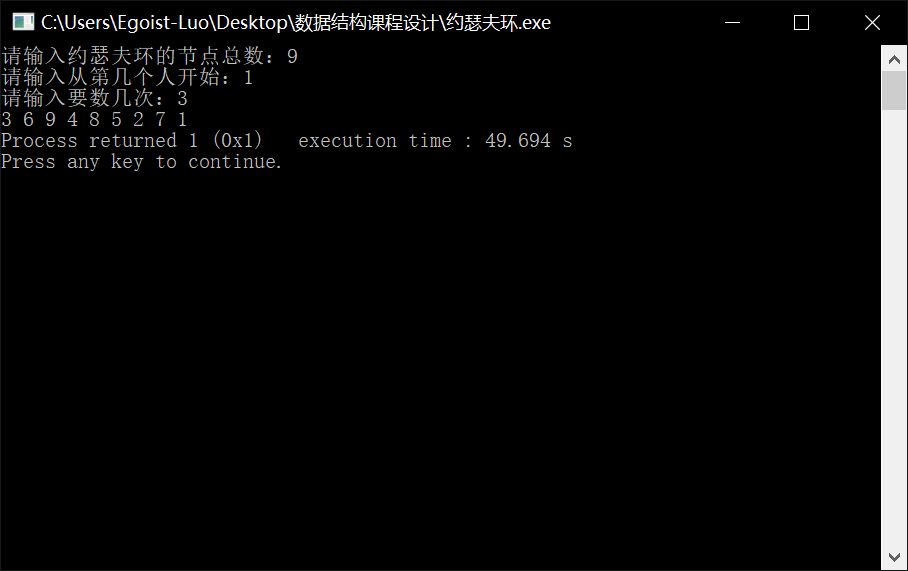


2)原始数据：节点总数9即123456789 起始1，间隔3

预期结果：369485271

实际结果：



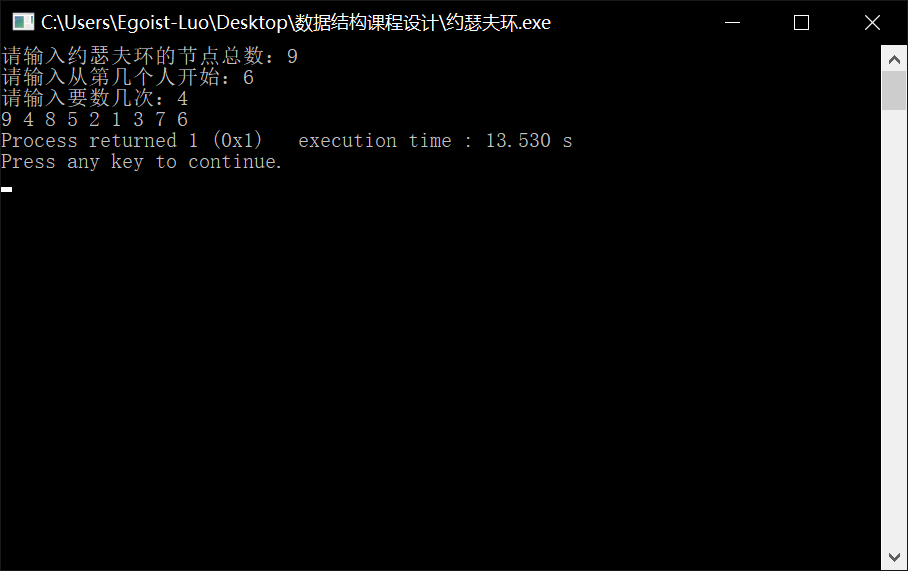


3)原始数据：节点总数9即123456789 起始6，间隔4

预期结果：648521376

实际结果：





<4>含注释的源程序

#include <stdio.h>

#include <stdlib.h>

#include <malloc.h>

typedef struct {

int num;

struct Node \*next;

}Node;

int create(Node \*L) { /\*创建约瑟夫环\*/

int i; /\*计数器\*/

int num; /\*每个节点中的数字及约瑟夫环的节点总数\*/

Node \*q; /\*定义新节点\*/

printf("请输入约瑟夫环的节点总数：");

scanf("%d", &num);

for (i = 1; i <= num; i++) { /\*建立数据节点\*/

q = (Node\*)malloc(sizeof(Node));

q -> num = i;

L -> next = q;

L = q;

}

L -> next = NULL; /\*尾标识为NULL\*/

return 1;

}

int process(Node \*L) { /\*实现约瑟夫环\*/

int m, n, i; /\*定义从第m个人开始，数n次\*/

Node \*head, \*point; /\*定义头结点，point节点(用以删除节点)\*/

head = L;

head = head -> next; /\*将头结点和L移到有数值的第一个节点\*/

L = head;

while (head -> next != NULL) { /\*遍历到最后一个节点\*/

head = head -> next;

}

head -> next = L; /\*闭环\*/

printf("请输入从第几个人开始：");

scanf("%d", &m);

printf("请输入要数几次：");

scanf("%d", &n);

for (i = 1; i < m; i++) { /\*将指针移到第m个人处\*/

L = L -> next;

}

if (n == 1) { /\*如果第m个人数1次，就依次出列\*/

while(L -> next != L) { /\*如果最后只有L指向的节点自己形成环，就跳出\*/

head = L; /\*L自身就是要输出的节点\*/

while(head -> next != L) { /\*将head指向L的前节点\*/

head = head -> next;

}

printf("%d ", L -> num);

point = L; /\*将point指针指向已经输出的节点\*/

L = L -> next;

head -> next = L; /\*将断环闭合\*/

free(point);

}

printf("%d", L -> num);

free(L);

return 1;

} else { /\*如果第m个人数n次\*/

while(L -> next != L) {

for (i = 1; i < n; i++) { /\*将L指向要输出的节点\*/

L = L -> next;

}

while(head -> next != L) {

head = head -> next;

}

printf("%d ", L -> num);

point = L;

L = L -> next;

head -> next = L;

free(point);

}

printf("%d", L -> num);

free(L);

return 1;

}

}

int main() {

Node \*Circle;

Circle = (Node\*)malloc(sizeof(Node));

create(&Circle);

process(&Circle);

}

<5>编译与运行情况：

编译正常通过，运行正常。

(5)表达式求解

<1>解题思路: 构造包含顶指针，底指针和增量的结构体。然后分别构造一个只包含运算符的栈（OPTR）和只包含数字的栈（OPND）。之后依次读入所输入的表达式。判断是不是数字，如果是数字就将数字放入数字栈（OPND）。如果不是即运算符，让运算符栈栈顶元素和读入的运算符进行比较。如果优先级小于将读入的运算符入栈，优先级相等的就让栈顶元素出栈，优先级的大于的就让栈顶元素弹栈，并且连续两次让数字栈弹栈，得到一个运算符和两个数字，进行计算，得到的结果放入数字栈。循环以上过程直到读入的表达式字符为#为止。最后将数字栈出栈，即得到结果。

<2>函数调用图

开始

创建运算符栈和数字栈，并初始化

读入表达式

不是

是

判断是否为数字

小于

放入数字栈

判断优先级

大于

等于

出栈

出栈

入栈

计算

入栈

判断是否结束读入

不是

是

数字栈出栈

结束

各函数功能:

char InitStack (SqStack \*S) 构造空栈

char Push (SqStack \*S, char e) 让e入栈

char Pop (SqStack \*S, char \*e) 让e出栈

char getTop(SqStack S) 返回栈顶第一个元素

char Precede(char a, char b) 比较两个运算符的优先级,a，b中存放待比较的运算符,'>'表示a>b,'0'表示不可能出现的比较，这是通过二维数组建立比较表实现的。

int in(char c, char OP[8]) 判断输入的某个字符是否是运算符

char Operate(char a, char n, char b) 四则运算计算

char evaluateExpression(SqStack OPTR, SqStack OPND) 表达式计算，流程控制

int main() 主函数，建立OPTR与OPND栈

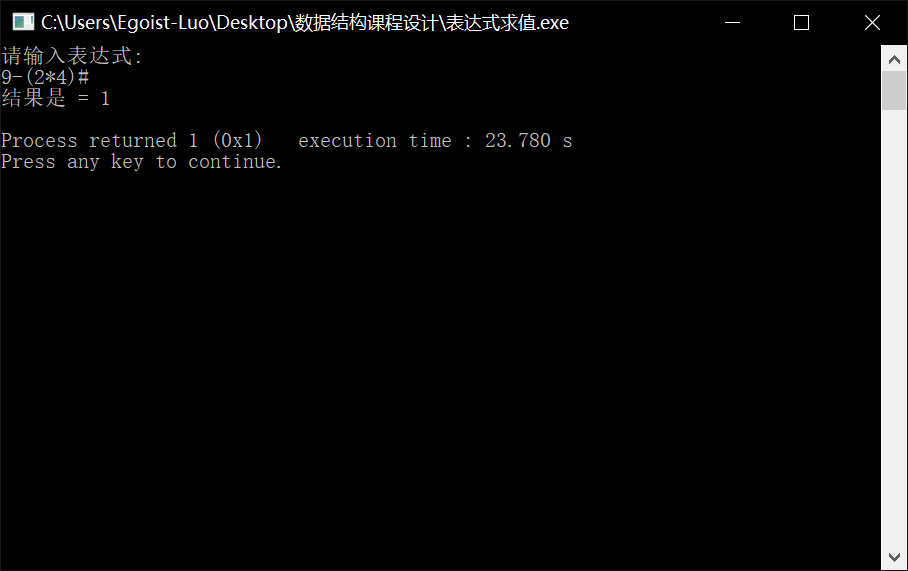
<3>三组测试数据

1)原始数据：9-(2\*4)#

预期结果：1

实际结果：



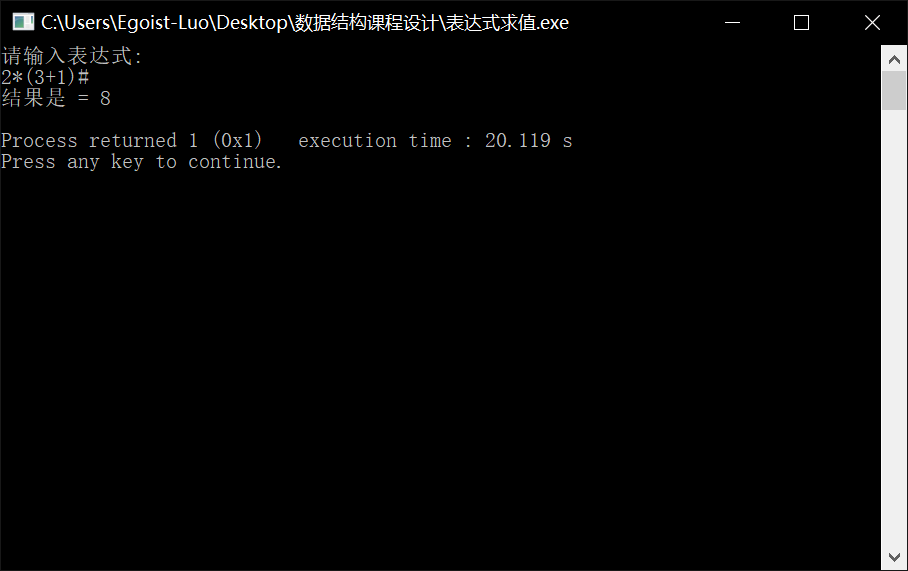


2)原始数据：2\*(3+1)#

预期结果：8

实际结果：



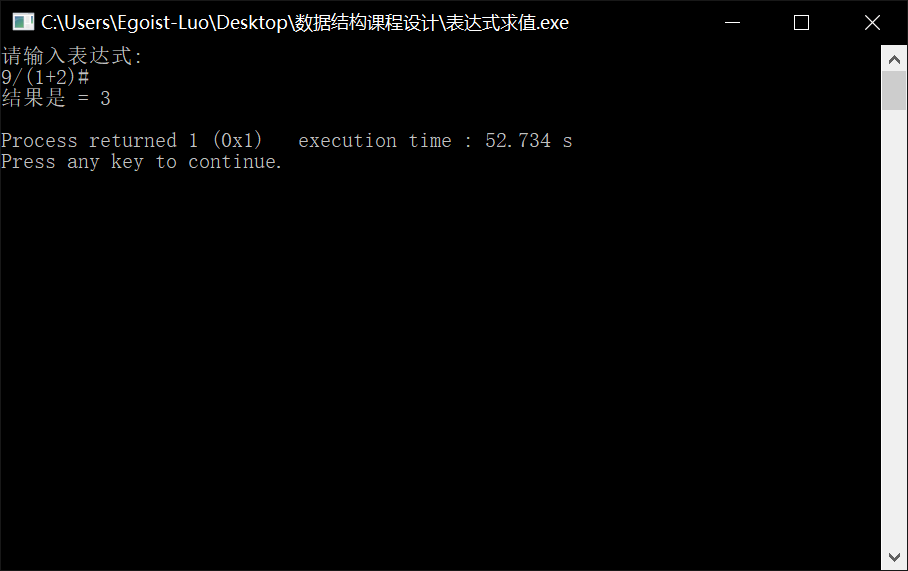


3)原始数据：9/(1+2)#

预期结果：3

实际结果：





<4>含注释的源程序

#include <stdio.h>

#include <stdlib.h>

#include <malloc.h>

#define STACK\_INIT\_SIZE 100 /\*存储空间初始分配量\*/

#define STACKINCREMENT 10 /\*存储空间分配增量\*/

typedef struct {

char \*top;

char \*base;

int stacksize;

}SqStack;

int InitStack (SqStack \*S) { /\*构造一个空栈\*/

S -> base = (char\*)malloc(STACK\_INIT\_SIZE\*sizeof(char));

S -> top = S -> base;

S -> stacksize = STACK\_INIT\_SIZE;

return 1;

}

int Push (SqStack \*S, char e) { /\*插入元素e为新的栈顶元素\*/

if (S -> top - S -> base >= S -> stacksize) {

S -> base = (char\*)realloc(S -> base, (S -> stacksize + STACKINCREMENT)\*sizeof(char));

if (!S -> base) { /\*栈溢出\*/

return -1;

}

S -> top = S -> base + S -> stacksize;

S -> stacksize += STACKINCREMENT;

}

\*(S -> top) = e; /\*先赋值在将指针加1\*/

S -> top ++;

return 1;

}

int Pop (SqStack \*S, char \*e) { /\*退栈\*/

if (S -> top == S -> base) { /\*栈空\*/

return 0;

} else {

S -> top --; /\*先将指针减1在出栈\*/

\*e = \*(S -> top);

}

return 1;

}

char getTop(SqStack S){ /\*返回栈顶第一个元素\*/

char e;

if(S.top == S.base)

return 0;

e = \*(S.top - 1);

return e;

}

char Precede(char a, char b) { /\*比较两个运算符的优先级,a，b中存放待比较的运算符,

'>'表示a>b,'0'表示不可能出现的比较 \*/

int i,j;

char pre[][7]={ /\*运算符之间的优先级制作成一张表格\*/

{'>','>','<','<','<','>','>'},

{'>','>','<','<','<','>','>'},

{'>','>','>','>','<','>','>'},

{'>','>','>','>','<','>','>'},

{'<','<','<','<','<','=','0'},

{'>','>','>','>','0','>','>'},

{'<','<','<','<','<','0','='}};

switch(a) {

case '+': i = 0; break;

case '-': i = 1; break;

case '\*': i = 2; break;

case '/': i = 3; break;

case '(': i = 4; break;

case ')': i = 5; break;

case '#': i = 6; break;

}

switch(b) {

case '+': j = 0; break;

case '-': j = 1; break;

case '\*': j = 2; break;

case '/': j = 3; break;

case '(': j = 4; break;

case ')': j = 5; break;

case '#': j = 6; break;

}

return pre[i][j];

}

int in(char c, char OP[8]) { /\*判断输入的某个字符是否是运算符\*/

int i;

for(i = 0; OP[i] != '\0'; i++) {

if(OP[i] == c) {

return 1;

}

}

return 0;

}

char Operate(char a, char n, char b) { /\*四则运算计算\*/

int i, j, result;

char back;

i = a - '0'; /\*将char转换为int\*/

j = b - '0';

switch(n) {

case '+': result = i + j; break;

case '-': result = i - j; break;

case '\*': result = i \* j; break;

case '/': result = i / j; break;

}

back = result + '0';

return back;

}

char evaluateExpression(SqStack OPTR, SqStack OPND) {

char a, b, c, x, n, temp;

temp = '#';

Push(&OPTR, temp);

char OP[8] = {'+','-','\*','/','(',')','#','\0'};

c = getchar(); /\*输入字符\*/

while(c != '#' || getTop(OPTR) != '#') { /\*判断是不是运算符\*/

if(!in(c, OP)) { /\*不是运算符则进栈\*/

Push(&OPND, c);

c = getchar();

} else { /\*是运算符就和前面的比较\*/

x = getTop(OPTR);

switch(Precede(x, c)) {

case '<': /\*栈顶元素优先级低\*/

Push(&OPTR, c);

c = getchar();

break;

case '=': /\*脱括号并接受下一字符\*/

Pop(&OPTR, &x);

c = getchar();

break;

case '>': /\*退栈并将运算结果入栈\*/

Pop(&OPTR, &n);

Pop(&OPND, &b);

Pop(&OPND, &a);

Push(&OPND, Operate(a, n, b));

break;

}

}

}

return getTop(OPND);

}

int main(){

char c;

SqStack OPTR; /\*OPTR为运算符栈，OPND为数字栈\*/

SqStack OPND;

InitStack(&OPTR);

InitStack(&OPND);

printf("请输入表达式:\n");

c = evaluateExpression(OPTR, OPND);

printf("结果是 = %c\n",c);

return 1;

}

<5>编译与运行情况：

编译正常通过。运行时如果计算中出现负数就无法运算，而且输入表达式的时候因为用的是getchar所以无法读出两位数以上的数字进行计算，这是该程序缺陷的地方。

(6) 将两个有序线性表合并成一个有序线性表，并去掉重复元素。

<1>解题思路:先建立一个结构体，结构体中包含数据域以及next的指针域。将每个结构体定义为一个节点，在通过指针域链接来建立链表。选择升序或者降序排序，分别输入节点数据，建立L1，L2的链表。然后根据选择升序或者降序调用不同的合并函数。如果是升序则数据小的先插入链表，大的后插入，如果是降序则大的先插入链表，小的后插入。最后输出新链表L3。

<2>函数调用图

开始

建立链表L1,L2

选择升降序

升序合并

降序合并

输出

结束

各函数功能:

int createList (LNode \*L) 建立线性表

int mergeListSL(LNode \*L1, LNode \*L2, LNode \*L3) 将L1，L2合并成L3，升序并输出

int mergeListLS(LNode \*L1, LNode \*L2, LNode \*L3) 将L1，L2合并成L3，

降序并输出

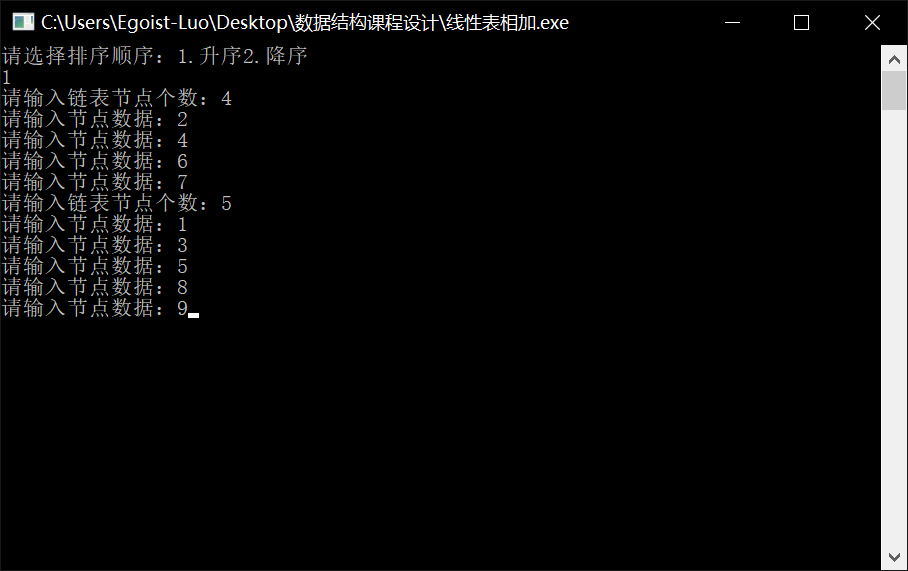
int main() 主函数，流程控制。

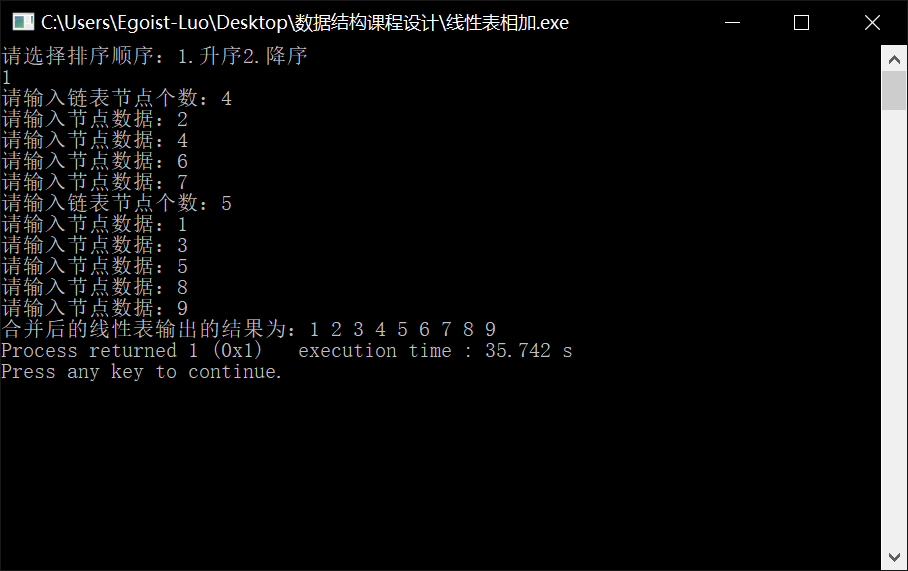
<3>三组测试数据

1)原始数据：升序（1） L1：2467 L2： 13589

预期结果：123456789

实际结果：

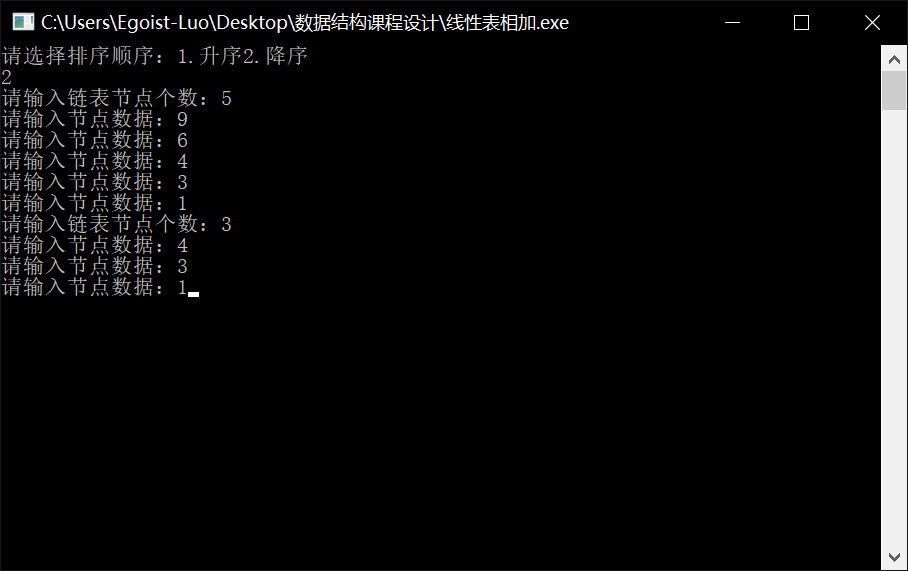


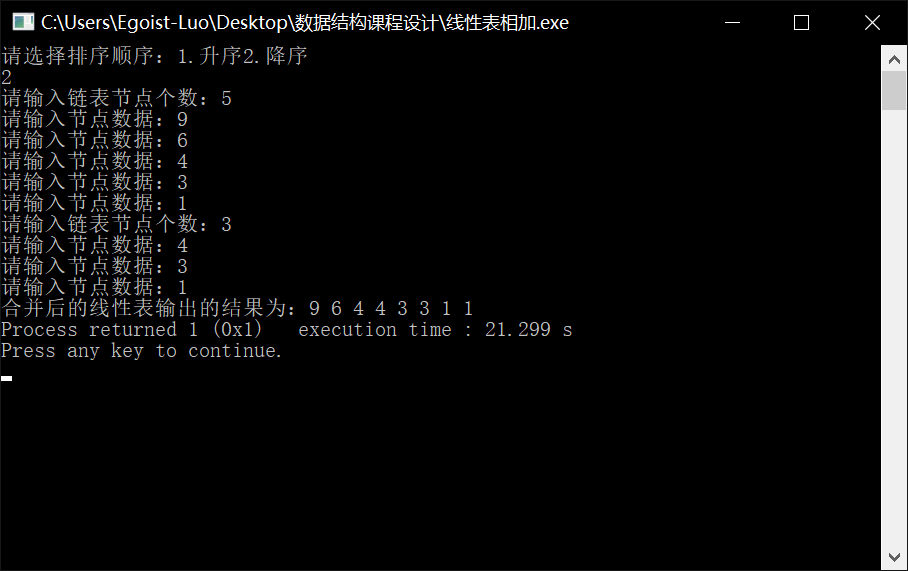


2)原始数据：降序（2） L1：96431 L2：431

预期结果：96443311

实际结果：

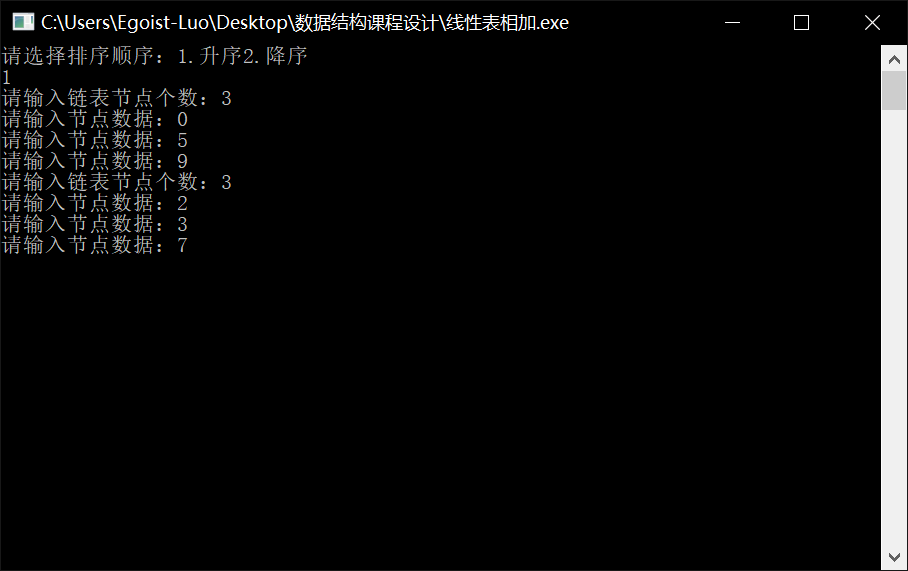


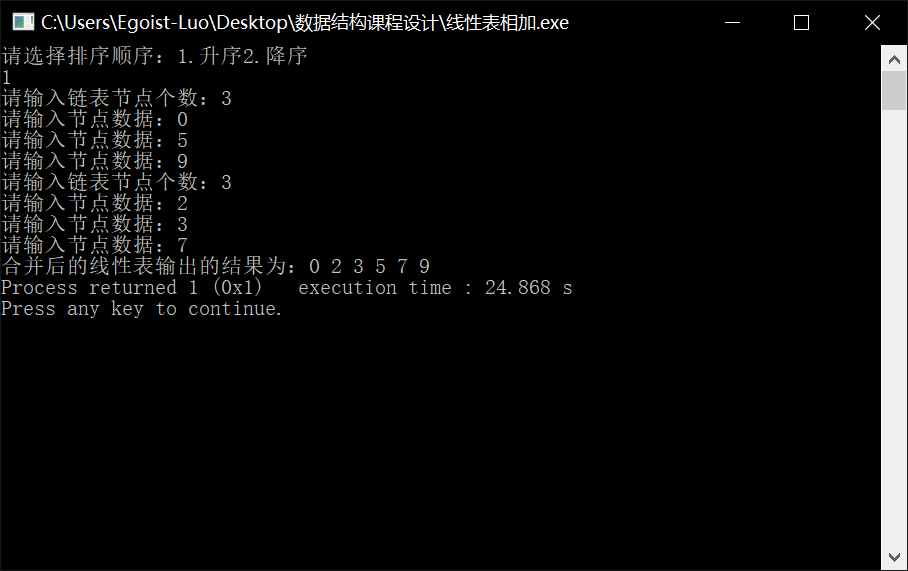


3)原始数据：升序（1） L1：059 L2：237

预期结果：023579

实际结果：





<4>含注释的源程序

#include <stdio.h>

#include <stdlib.h>

#include <malloc.h>

typedef struct {

int data;

struct Node \*next;

}LNode;

int createList (LNode \*L) { /\*建立线性表\*/

LNode \*q;

int m, n, i; /\*节点个数为m，节点数据为n\*/

printf("请输入链表节点个数：");

scanf("%d", &m);

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

q = (LNode\*)malloc(sizeof(LNode));

printf("请输入节点数据：");

scanf("%d", &n);

q -> data = n;

L -> next = q;

L = q;

}

L -> next = NULL; /\*尾标识为NULL\*/

return 1;

}

int mergeListSL(LNode \*L1, LNode \*L2, LNode \*L3) { /\*将L1，L2合并成L3，升序\*/

LNode \*p1, \*p2, \*p3;

p1 = L1;

p2 = L2;

p3 = L3;

p1 = p1 -> next;

p2 = p2 -> next;

while (p1 != NULL && p2 != NULL) {

if (p1 -> data <= p2 -> data) { /\*如果p1的data小于p2的data，就将p1插入L3\*/

p3 -> next = p1;

p3 = p1;

p1 = p1 -> next;

} else { /\*如果p2的data小于p1的data，就将p2插入L3\*/

p3 -> next = p2;

p3 = p2;

p2 = p2 -> next;

}

}

p3 -> next = (p1 != NULL)? p1: p2; /\*如果p1还有节点，就将p1剩余节点全部插入L3，p2同理\*/

p3 = L3; /\*指针回到L3开头\*/

p3 = p3 -> next;

printf("合并后的线性表输出的结果为：");

while (p3 != NULL) { /\*将L3线性表显示出来\*/

printf("%d ", p3 -> data);

p3 = p3 -> next;

}

return 1;

}

int mergeListLS(LNode \*L1, LNode \*L2, LNode \*L3) { /\*将L1，L2合并成L3，降序\*/

LNode \*p1, \*p2, \*p3;

p1 = L1;

p2 = L2;

p3 = L3;

p1 = p1 -> next;

p2 = p2 -> next;

while (p1 != NULL && p2 != NULL) {

if (p1 -> data >= p2 -> data) { /\*如果p1的data大于p2的data，就将p1插入L3\*/

p3 -> next = p1;

p3 = p1;

p1 = p1 -> next;

} else { /\*如果p2的data大于p1的data，就将p2插入L3\*/

p3 -> next = p2;

p3 = p2;

p2 = p2 -> next;

}

}

p3 -> next = (p1 != NULL)? p1: p2; /\*如果p1还有节点，就将p1剩余节点全部插入L3，p2同理\*/

p3 = L3; /\*指针回到L3开头\*/

p3 = p3 -> next;

printf("合并后的线性表输出的结果为：");

while (p3 != NULL) { /\*将L3线性表显示出来\*/

printf("%d ", p3 -> data);

p3 = p3 -> next;

}

return 1;

}

int main() {

LNode \*L1, \*L2, \*L3;

int ch;

L1 = (LNode\*)malloc(sizeof(LNode));

L2 = (LNode\*)malloc(sizeof(LNode));

L3 = (LNode\*)malloc(sizeof(LNode));

printf("请选择排序顺序：");

printf("1.升序");

printf("2.降序\n");

scanf("%d", &ch);

createList(&L1);

createList(&L2);

switch(ch) {

case 1:mergeListSL(&L1, &L2, &L3);break;

case 2:mergeListLS(&L1, &L2, &L3);break;

default: printf("输入错误");break;

}

return 1;

}

<5>编译与运行情况：

编译正常通过，运行正常。