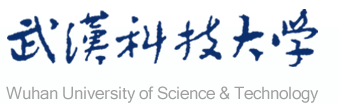
****

《数据结构》实验报告

**专 业：** 计算机科学与技术（卓越计划）

**班 级：** 计算机（卓越）1901

**学 号：** 201913137031

**姓 名：** 杨尔行

**实验2 一元多项式相加\***

**一、【实验目的】**

1、了解链式存储结构的基本知识；

2、掌握算法思想和数据结构的描述；

3、结合一元多项式相加的运算规则。

**二、【实验环境】**

**硬件**：Intel Core(TM) i5-8300H CPU 2.30GHZ

**软件**：Windows10 家庭中文版 64位操作系统；Visual Studio 2019 集成开发环境

**三、【实验内容】**

结合书上第41页的例子，采用链式存储结构，将两个线性链表表示的一元多项式相加，并输出。此一元多项式遵循多项式相加运算规则：对于两个一元多项式中存在指数相同的项时，其对应系数相加。合并后系数和为零时，删除“和多项式”中此项；合并后系数和不为零时，则构成“和多项式”中的一项。对于两个一元多项式中存在的指数不相同的项，则分别复抄到“和多项式”中去，原多项式保持不变。

**四、【实验程序】**

1)程序源代码

**文件名：main.c**

#include <stdio.h>

#include "polyn.h"

/// 从stdin取得多项式的参数，格式：项数N 系数1 指数1 系数2 指数2 ...... 系数N 指数N

void get\_polyn\_parameters(unsigned\* n, double coef[], int expn[]);

/// 主函数

int main()

{

// 缓存

int expn[10] = { 0 };

double coef[10] = { 0 };

unsigned n;

// 输入多项式1

printf("Enter polynomial 1. Format: N coef1 expn1 coef2 expn2 ... coefN expnN\n");

get\_polyn\_parameters(&n, coef, expn); // 从stdin取得参数，保存到缓存

Polyn\* p1 = init\_polyn(n, coef, expn); // 用缓存中的参数创建多项式链表p1

// 输入多项式2

printf("Enter polynomial 2. Format: N coef1 expn1 coef2 expn2 ... coefN expnN\n");

get\_polyn\_parameters(&n, coef, expn); // 从stdin取得参数，保存到缓存

Polyn\* p2 = init\_polyn(n, coef, expn); // 用缓存中的参数多项式链表p2

printf("Input:\nP1 = ");

print\_polyn(p1); // 输出相加前的p1

printf("P2 = ");

print\_polyn(p2); // 输出相加前的p2

printf("Computing...\n");

Polyn\* psum = add\_polyn(p1, p2); // 执行p1+p2

printf("Output:\nP1 + P2 =\n");

print\_polyn(p1);// 输出相加后的p1，证明其未被破坏

printf("+\n");

print\_polyn(p2);// 输出相加后的p2，证明其未被破坏

printf("=\n");

print\_polyn(psum);// 输出多项式和psum

deinit\_polyn(p1); // 回收内存

deinit\_polyn(p2);

deinit\_polyn(psum);

}

/// 从stdin取得多项式的参数，格式：项数N 系数1 指数1 系数2 指数2 ...... 系数N 指数N

void get\_polyn\_parameters(unsigned\* n, double coef[], int expn[])

{

scanf("%ud", n);

\*n = \*n % 10;

for (size\_t i = 0; i < \*n; i++) {

scanf("%lf %d", coef + i, expn + i);

}

}

**文件名：polyn.h**

#ifndef \_POLYN\_H\_

#define \_POLYN\_H\_

#include <stdlib.h>

#include <stdio.h>

/// 多项式节点

typedef struct Polyn

{

/// 后继节点

struct Polyn\* next;

/// 数据域

struct

{

/// 指数

int expo;

/// 系数

double coef;

} term;

} Polyn;

/// 给定数组，从中读取多项式。\*返回的内存要用deinit\_polyn回收\*

/// size - 多项式项数

/// expo - 指数组成的数组

/// coef - 系数组成的数组

Polyn\* init\_polyn(size\_t size, double coef[], int expo[]);

/// 回收多项式链表。

/// p - 多项式的指针

void deinit\_polyn(Polyn\* p);

/// 将两个升幂排列的多项式相加，不破坏原来的多项式，返回结果多项式。\*返回的内存要用deinit\_polyn回收\*

/// p1 - 多项式1

/// p2 - 多项式2

Polyn\* add\_polyn(const Polyn\* p1, const Polyn\* p2);

/// 输出多项式。

/// p - 多项式

void print\_polyn(const Polyn\* p);

#endif // \_POLYN\_H\_

**文件名：polyn.c**

#include "polyn.h"

#include <stdio.h>

/// 给定数组，从中读取多项式。返回的Polyn\*要用deinit\_polyn回收

/// size - 多项式项数，即数组的长度

/// expo - 指数组成的数组

/// coef - 系数组成的数组

Polyn\* init\_polyn(size\_t size, double coef[], int expo[])

{

Polyn\* head = malloc(sizeof(Polyn)); // 头节点

Polyn\* p = head;

for (size\_t i = 0; i < size; i++) { // 遍历数组添加节点

p->next = malloc(sizeof(Polyn));

p = p->next;

p->term.coef = coef[i];

p->term.expo = expo[i];

}

p->next = NULL;

return head;

}

/// 回收多项式链表。

/// p - 多项式的指针

void deinit\_polyn(Polyn\* p)

{

while (p->next) { // 不断删除第一个

Polyn\* saved = p->next;

p->next = p->next->next;

free(saved);

}

free(p); // 回收头节点

}

/// 将两个升幂排列的多项式相加，不破坏原来的多项式，返回结果多项式。返回的Polyn\*要用deinit\_polyn回收。

/// p1 - 多项式1

/// p2 - 多项式2

Polyn\* add\_polyn(const Polyn\* p1\_head, const Polyn\* p2\_head)

{

Polyn\* r\_head = malloc(sizeof(Polyn)); // 结果多项式的头

Polyn\* r = r\_head; // 指向结果的尾节点

Polyn\* p1 = p1\_head->next, \* p2 = p2\_head->next; // 指向p1p2的节点

while (p1 || p2) { // p1 p2不全为空

int take\_who = 0; //表示选择p1(值为1)还是p2(值为2)还是都要(值为3)的flag

if (!p1) {// p1空且p2不空

take\_who = 2; // 选择p2

}

else if (!p2) {// p1不空且p2空

take\_who = 1; // 选择p1

}

else { // p1 p2 全不为空

if (p1->term.expo > p2->term.expo) { // p2 指数小

take\_who = 2; // 选p2

}

else if (p1->term.expo < p2->term.expo) { // p1 指数小

take\_who = 1;// 选p1

}

else take\_who = 3; // 指数一样大，都要

}

if (take\_who == 1) { // 如果选p1

r->next = malloc(sizeof(Polyn));

r = r->next;

r->term = p1->term;

p1 = p1->next;

}

else if (take\_who == 2) {// 如果选p2

r->next = malloc(sizeof(Polyn));

r = r->next;

r->term = p2->term;

p2 = p2->next;

}

else if (take\_who == 3) {// 如果都要

double sum = p1->term.coef + p2->term.coef;

if (sum) {

r->next = malloc(sizeof(Polyn));

r = r->next;

r->term.coef = p1->term.coef + p2->term.coef;

r->term.expo = p1->term.expo;

}

p1 = p1->next;

p2 = p2->next;

}

else abort();

}

r->next = NULL; // 尾节点置空

return r\_head;

}

/// 输出多项式。

/// p - 多项式

void print\_polyn(const Polyn\* p\_h)

{

int first = 1;

Polyn\* p = p\_h->next;

while (p) {

if (first) first = 0;

else printf(" + "); // 不是第一次，就需要输出加号

switch (p->term.expo) {

case 0:

printf("%4.2lf ", p->term.coef);// 指数为0

break;

case 1:

printf("%4.2lf x ", p->term.coef);// 指数为1

break;

default: // 指数非0非1

printf("%4.2lf x ^ %-3d", p->term.coef, p->term.expo);

}

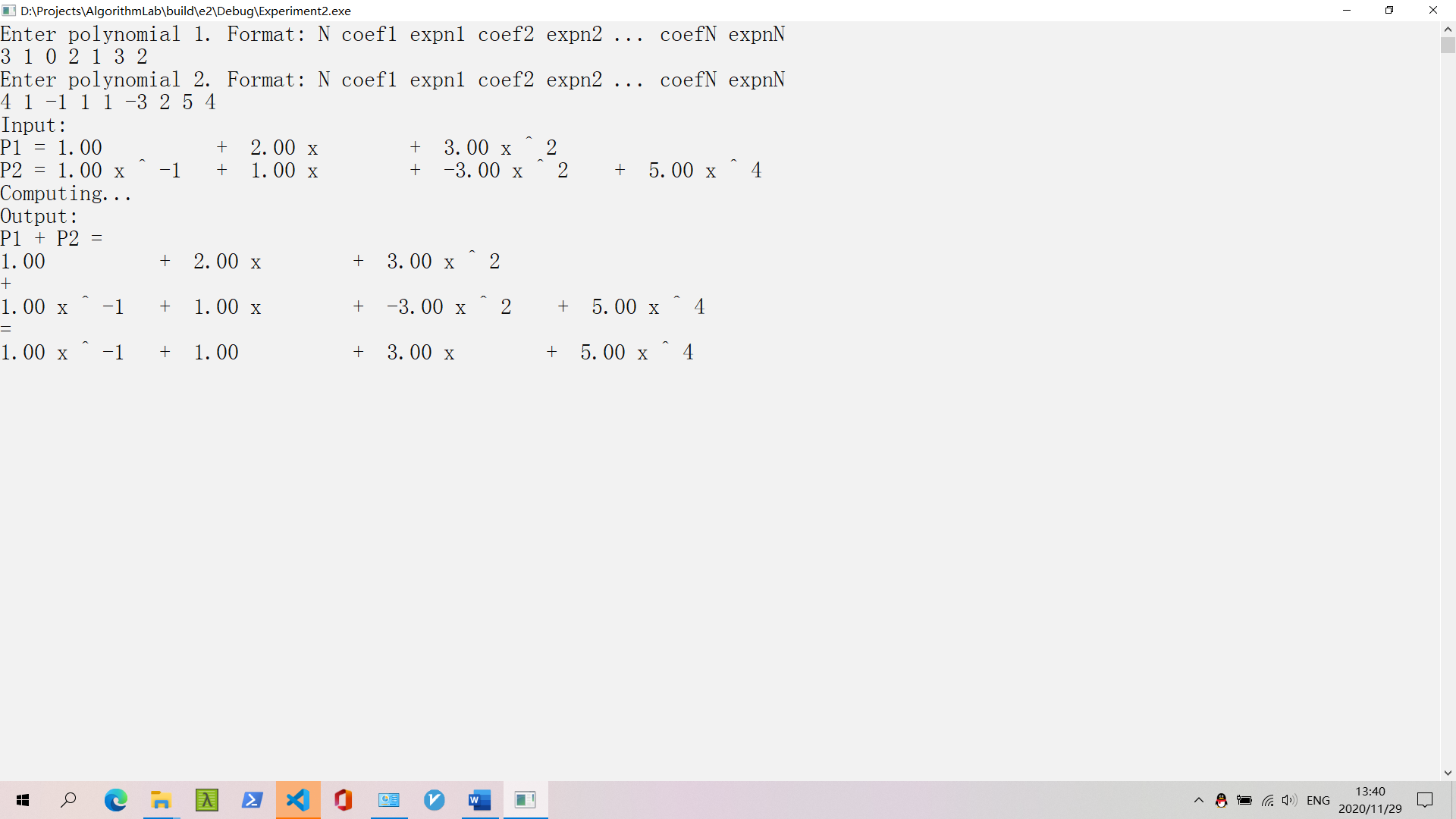
p = p->next;

}

putchar('\n');

}

2)程序运行结果截图



**六、【实验小结】**

本次实验使用了第2章线性表的知识，实现了链表的一些操作。一元多项式的相加本质上是对两个有序链表进行合并的过程。维护两个指针，分别指向两个加数链表。在循环中，不断取较低幂项加入结果链表，如果等幂，则系数相加再加入结果链表。由于仅需要遍历两个加数链表各一次，算法的时间复杂度为O(n)。

由于算法涉及较多的线性表插入和删除操作，使用链式存储结构显然是最合适的。