哈尔滨工业大学计算机科学与技术学院

实验报告

课程名称：数据结构与算法

课程类型：必修

实验项目：线性表的链式存储结构与应用

实验题目：一元多项式计算器

实验日期：2020 年 10 月 20 日

班级：

学号：

姓名： Youngsc

设计成绩	报告成绩	指导老师
		张岩

一、 实验目的

1. 掌握线性表顺序存储结构的特点及线性表在顺序存储结构中各种基本操作的实现。
2. 掌握线性表链式存储结构的特点及线性表在链式存储结构中各种基本操作的实现。
3. 重点巩固和体会线性表在链式存储结构上的各种操作和应用。

二、实验要求及实验环境

（一）实验要求

以动态或者静态链表存储一元多项式，在此基础上按要求完成对一元多项式的运算。（为保证多项式的值的准确性，多项式的系数可以用分数表示，涉及到两个分数相除时，结果也可以用分数表示。）

1. 能够输入多项式（可以按各项的任意输入顺序，建立按指数降幂排列的多项式）和输出多项式（按指数降幂排列）,以文件形式输入和输出，并显示。
2. 能够给出计算两个多项式加法、减法、乘法和除法运算的结果多项式，除法运算的结果包括商多项式和余数多项式。
3. 能够计算一元多项式的 k 阶导函数。
4. 能够计算多项式在某一点 $x=x_0$ 的值，其中 x_0 是一个浮点型常量，返回结果为浮点数。
5. 要求尽量减少乘法和除法运算中间结果的空间占用和结点频繁的分配与回收操作。（提示：利用循环链表结构或者可用空间表的思想，把循环链表表示的多项式返还给系统或者可用空间表，从而解决上述问题）。

（二）实验环境

1. 硬件环境
   a) Legion Y7000P 2019 PG0
   b) CPU: Intel(R)_Core(TM)_i7-9750H_CPU_@_2.60GHz 2.59GHz
   c) 内存(RAM): 16GB DDR4
2. 系统环境
   a) Windows10 家庭中文版
3. 开发工具
   a) Dev C++5.11
   b) gcc (GCC) 9.2.0

（三）

三、设计思想（本程序中的用到的所有数据类型的定义，主程序的流程图及各程序模块之间的调用关系）

1．逻辑设计

为了实现多项式的四则运算，我们需要实现分数的四则运算以及约分通分，紧接着运用链表，将一个表达式不同次幂的项由高次幂向低次幂依次链接。接着按照多项式的四则运算法则分别重载四个运算符号，其中除法得到商多项式和余数多项式，故用二元组的形式同时将其返回。而求 k 阶导便用到了最直观的求导法则进行计算并返回。求 x 的不同取值下的浮点值也是模拟了带数求值的过程并返回结果实现。

2．物理设计

输入时我们首先输入一个多项式的项数，接着依次输入每一项的系数分子，系数分母和幂次，我们规定：

a) 若该系数小于 0，则我们将分子取为小于零，分母取为大于零。
b) 若系数为整数，则我们仍然需要输入原本为 1 的分母。
c) 我们输入的系数的分母分子可以是不经过约分的。
d) 我们输入时的多项式可以无序，也可以存在多个同次幂的项不合并同类项。

我们在输出多项式时有以下设计来保证多项式输出形式与我们日常书写多项式的形式尽可能接近：

a) 系数为零的项，初下文 b）提到的情况外，我们通常不予显示。
b) 若当前多项式在合并同类项后各系数均为 0，我们将输出一个单独的值 0，即此时零次项的系数虽然是 0，当我们仍然显示。
c) 若当前系数的分母化简后为 1，我们仅显示分子。
d) 若当前项的次数为 0，我们仅输出系数；若当前项的次数为 1，我们输出的形式，若当前项的次数大于 1，我们将输出 ax^b 的形式。

紧接着对于多项式的结构体内，我们设计一个 change 函数使该多项式加上一个 change 函数传入的项。通过这个核心的 change 函数，从而实现对多项式的四则运算。

四、测试结果

输入样例 1	输出样例 1
4 -3 1 1 3 1 2 -1 1 3 2 1 0 2 1 1 1 -1 1 0	不等式 A 为：-x^3 + 3x^2 - 3x + 2 不等式 B 为：x - 1 A+B 为：-x^3 + 3x^2 - 2x + 1 A-B 为：-x^3 + 3x^2 - 4x + 3 A*B 为：-x^4 + 4x^3 - 6x^2 + 5x - 2 A/B 的商多项式为：-x^2 + 2x - 1 A/B 的余数多项式为：1 验证结果：-x^3 + 3x^2 - 3x + 2 A 多项式的 0 阶导数为：-x^3 + 3x^2 - 3x + 2 A 多项式的 1 阶导数为：-3x^2 + 6x - 3 A 多项式的 2 阶导数为：-6x + 6 A 多项式的 3 阶导数为：-6 A 多项式的 4 阶导数为：0 A 多项式的 5 阶导数为：0 当原函数中自变量取值为 1 时，多项式 A 取值为 1.000 当原函数中自变量取值为 2 时，多项式 A 取值为 0.000 当原函数中自变量取值为 3 时，多项式 A 取值为-7.000
输入样例 2	输出样例 2
10 1 1 0 1 1 3 3 2 1 -4 2 2 -6 1 4 9 4 2 7 4 6 3 6 5 2 8 2 1 1 7	不等式 A 为：x^7 + 7/4x^6 + 1/2x^5 - 6x^4 + x^3 + 1/2x^2 + 3/2x + 1 不等式 B 为：x^2 - x + 1 A+B 为：x^7 + 7/4x^6 + 1/2x^5 - 6x^4 + x^3 + 3/2x^2 + 1/2x + 2 A-B 为：x^7 + 7/4x^6 + 1/2x^5 - 6x^4 + x^3 - 1/2x^2 + 5/2x A*B 为：x^9 + 3/4x^8 - 1/4x^7 - 19/4x^6

3 1 1 0 1 1 2 -1 1 1	+ 15/2x^5 - 13/2x^4 + 2x^3 + 1/2x + 1 A/B 的商多项式为：x^5 + 11/4x^4 + 9/4x^3 - 13/2x^2 - 31/4x - 3/4 A/B 的余数多项式为：17/2x + 7/4 验证结果：x^7 + 7/4x^6 + 1/2x^5 - 6x^4 + x^3 + 1/2x^2 + 3/2x + 1 A 多项式的 0 阶导数为：x^7 + 7/4x^6 + 1/2x^5 - 6x^4 + x^3 + 1/2x^2 + 3/2x + 1 A 多项式的 1 阶导数为：7x^6 + 21/2x^5 + 5/2x^4 - 24x^3 + 3x^2 + x + 3/2 A 多项式的 2 阶导数为：42x^5 + 105/2x^4 + 10x^3 - 72x^2 + 6x + 1 A 多项式的 3 阶导数为：210x^4 + 210x^3 + 30x^2 - 144x + 6 A 多项式的 4 阶导数为：840x^3 + 630x^2 + 60x - 144 A 多项式的 5 阶导数为：2520x^2 + 1260x + 60 当原函数中自变量取值为 1 时，多项式 A 取值为 1.250 当原函数中自变量取值为 2 时，多项式 A 取值为 174.000 当原函数中自变量取值为 3 时，多项式 A 取值为 3135.250

五、经验体会与不足

通过本实验，我了解到了如何通过动态链式存储结构实现链表，掌握了线性表链序存储结构的特点及线性表在链序存储结构中各种基本操作的实现，意识到了抽象数据类型的作用。

但是在这次实验的过程中我的代码实现过长，显得比较冗余，下次应当改进

六、附录：源代码（带注释）

```
# include <bits/stdc++.h>

using namespace std;

int gcd(int x,int y){return y? gcd(y,x%y):x;} // 通过辗转相除法求分子和分母的最大公约数从而实现约分通分
```

```
struct fraction{ //  分数的结构体
	int numerator,denominator,flag; //  分别为分子、分母、正负号。
	fraction(){numerator = 0, denominator = 1; flag=0;} //  构造函数初始化
	fraction(int _nume,int _deno){numerator = abs(_nume),denominator = _deno,flag = numerator ? _nume/numerator : 0;}

	fraction simplify() //  约分函数
	{
		if (!numerator) denominator = 1;
		else
		{
			int p = gcd(numerator,denominator); //  求得分子分母的最大公约数
			numerator /= p;
			denominator /= p;
		}
		return *this;
	}

	fraction operator + (const fraction& a) const{ //  重载两个分数的加法
		int p = gcd(denominator,a.denominator); //  求出两个分数的分母的最大公约数用来实现通分
		fraction ret = fraction(numerator*(a.denominator/p)*flag+a.numerator*(denominator/p)*a.flag,denominator*a.denominator/p);
		return ret.simplify();
	}

	fraction operator - (const fraction& a) const{//  重载两个分数的减法
		int p = gcd(denominator,a.denominator);
		fraction ret = fraction(numerator*(a.denominator/p)*flag-a.numerator*(denominator/p)*a.flag,denominator*a.denominator/p);
		return ret.simplify();
	}

	fraction operator * (const fraction& a) const{//重载两个分数的乘法
		fraction ret = fraction(numerator*a.numerator*flag*a.flag,denominator*a.denominator);
		return ret.simplify();
	}

	fraction operator / (const fraction& a) const{//重载两个分数的除法
		fraction ret = fraction(numerator*a.denominator*flag*a.flag,denominator*a.numerator);
		return ret.simplify();
	}

	double operator * (const double& a) const { //  重载一个分数和一个双精度浮点数的乘法
		return 1.0*numerator*a/denominator*flag;
```

```
        }
};

struct ele{ // 定义多项式中每一项的结构体
        fraction coefficient; // 分数形式的系数
        int power; // 次幂
        ele *nxt; // 该项所在多项式中的下一项
        ele(){};
        ele(int _numer,int _denom,int _power){coefficient = fraction(_numer,_denom),power = _power; nxt = NULL;}
        ele(fraction _ceof,int _power){coefficient = _ceof,power = _power; nxt = NULL;}
        // 两种不同参数的构造函数
};

struct polynomial{ // 定义多项式的结构体
        ele *base;// 该多项式的最高次项
        polynomial(){base = new ele(0,1,0);} // 构造函数初始化

        void change(ele to) // 向当前多项式加上 to 这一项
        {
                if (!to.coefficient.numerator) return; // 加的项为 0 则返回
                ele *now = this -> base,*pre = NULL;
                while (now != NULL && now->power > to.power) pre = now,now = now->nxt;
                if (now == NULL || now -> power < to.power) // 若原本不存在与 to 同次幂的项
                {
                        ele *it = new ele; // 新建项
                        *it = to;
                        if (pre != NULL) pre -> nxt = it;
                        else base = it;
                        it -> nxt = now;
                        it -> coefficient.simplify();
                }

                else // 若原本就存在该次幂的项
                {
                        now -> coefficient = now -> coefficient + to.coefficient;
                        if (!now -> coefficient.numerator && now -> power)
                        {
                                if (pre == NULL) this -> base    = now -> nxt;
                                else pre -> nxt = now -> nxt;
                                delete(now);
                        }
                }
        }
```

```
polynomial operator + (const polynomial& a) const // 重载多项式加法
{
	polynomial ret;
	ele *nowa = base,*nowb = a.base,*now = new ele;
	while (nowa != NULL && nowb != NULL) // 按照将次分别加入
	{
		if (nowa->power > nowb->power) *now = *nowa,nowa = nowa->nxt;
		else if (nowa->power < nowb->power) *now = *nowb,nowb = nowb->nxt;
		else *now = ele(nowa->coefficient+nowb->coefficient,nowa->power),nowa = nowa->nxt,nowb = nowb->nxt;
		ret.change(*now);
	}
	delete(now);
	while (nowa != NULL) ret.change(*nowa),nowa = nowa->nxt;
	while (nowb != NULL) ret.change(*nowb),nowb = nowb->nxt;
	return ret;
}

polynomial operator - (const polynomial& a) const //重载多项式的减法
{
	polynomial ret;
	ele *nowa = base,*nowb = a.base,*now = new ele;
	while (nowa != NULL && nowb != NULL)
	{
		if (nowa->power > nowb->power) ret.change(*nowa),nowa = nowa->nxt;
		else if (nowa->power < nowb->power)
		{
			*now = *nowb;
			now -> coefficient.flag *= -1;
			ret.change(*now);
			nowb = nowb->nxt;
		}
		else
		{
			*now = ele(nowa->coefficient-nowb->coefficient,nowa->power),nowa = nowa->nxt,nowb = nowb->nxt;
			ret.change(*now);
		}
	}

	while (nowa != NULL) ret.change(*nowa),nowa = nowa->nxt;
	while (nowb != NULL)
	{
		*now = *nowb;
		now->coefficient.flag *= -1;
		ret.change(*now);
```

```
            nowb = nowb->nxt;
        }
        delete now;
        return ret;
    }

    polynomial operator * (const polynomial& a) const // 重载多项式的乘法
    {
        polynomial ret;
        ele *nowa = base,*nowb,*now = new ele;
        while (nowa != NULL)
        {
            nowb = a.base;
            while (nowb != NULL) ret.change(ele(nowa->coefficient*nowb->coefficient,nowa->power+nowb->power)),nowb =
nowb->nxt;
            // 使得两多项式中的任意两项均相乘后加入结果
            nowa = nowa->nxt;
        }
        return ret;
    }

    pair <polynomial,polynomial> operator / (const polynomial& a) const // 使用二元组的形式重载多项式除法，从而可以同时返回商多项
式和余数多项式
    {
        polynomial ret,res=*this,*b = new polynomial;
        while (res.base -> power >= a.base -> power)
        {
            ele *now;
            *b = a;
            polynomial ex;
            ex.change(ele(res.base->coefficient/b->base->coefficient,res.base->power-b->base->power));
            *b = *b*ex;
            res = res-(*b);
            ret.change(*(ex.base));
        }
        // 按照多项式除法的法则计算结果
        return make_pair(ret,res);
    }

    polynomial derivative(int k){ //求 k 次导函数
        polynomial ret;
        ele *now = this -> base;
        while (now != NULL && now->power >= k)
        {
```

```
                fraction x=now->coefficient;
                for (int i=now->power; i>=now->power-k+1; -    -i) x = x*fraction(i,1);
                ret.change(ele(x,now->power-k));
                now = now->nxt;
            }// 按照多项式求导法则求导
            return ret;
        }

        void print(){ // 打印多项式
            ele *now = base;
            if (now -> power == 0) //
            {
                if (now->coefficient.denominator != 1)
printf("%d/%d",now->coefficient.numerator*now->coefficient.flag,now->coefficient.denominator);
                else printf("%d",now->coefficient.numerator*now->coefficient.flag); // 识别整数或分数
                printf("\n");
                return;
            }
            bool flag = 0;
            while (now != NULL)
            {
                if (now->coefficient.numerator == 0 && flag) break;
                if (!flag) // 判断是不是要输出的第一项
                {
                    flag = 1;
                    if (now -> coefficient.flag < 0) printf("-"); // 第一项为正不管，为负输出符号
                }
                else
                {
                    if (now -> coefficient.flag > 0) printf("+ "); // 除第一项外，根据正负确定输出的符号为加号减号。
                    else printf("- ");
                }

                if (now->coefficient.denominator != 1)
printf("%d/%d",abs(now->coefficient.numerator),now->coefficient.denominator); // 系数是分数的输出形式
                else if (!now->power) printf("%d",abs(now->coefficient.numerator)); //
                else if (now->coefficient.numerator != 1) printf("%d",abs(now->coefficient.numerator));      // 系数为整数的输出形
式。

                if (now->power > 1) printf("x^%d ",now->power); // 当前项次数高于 1 的输出形式
                else if (now->power == 1) printf("x ",now->power); // 当前项次数等于 1 时只输出 x
                else printf(" "); //常数项输出形式
                now = now -> nxt;
            }
```

```
        printf("\n");
    }

    void scan(){ // 输入多项式
        int n,x,y,z;
        scanf("%d",&n);
        for (int i=1; i<=n; ++i) scanf("%d%d%d",&x,&y,&z),change(ele(x,y,z));
    }

    double calc(double x){ // 计算 x 不同取值下的函数值
        double ret = 0;
        ele *now = this->base;
        while (now != NULL)
        {
            double sum = 1;
            for (int i=1; i<=now->power; ++i) sum *= x;
            ret += now->coefficient*sum;
            now = now->nxt;
        }
        return ret;
    }
};

int main(){
    freopen("data.in","r",stdin);
    freopen("data.out","w",stdout);
    polynomial x,y;
    printf("不等式 A 为：");
    x.scan(); x.print();
    printf("不等式 B 为：");
    y.scan(); y.print();
    printf("\nA+B 为：");
    (x+y).print();
    printf("\nA-B 为：");
    (x-y).print();
    printf("\nA*B 为：");
    (x*y).print();
    pair <polynomial,polynomial> now = x/y;
    printf("\nA/B 的商多项式为：");
    now.first.print();
    printf("A/B 的余数多项式为：");
    now.second.print();
    printf("验证结果：");
    (now.first*y+now.second).print();
```

```
        printf("\n");
        for (int i=0; i<=5; ++i)
        {
                printf("A 多项式的%d 阶导数为：",i);
                x.derivative(i).print();
        }
        printf("\n");

        printf("当原函数中自变量取值为 1 时，多项式 A 取值为%.3lf\n",x.calc(1));
        printf("当原函数中自变量取值为 2 时，多项式 A 取值为%.3lf\n",x.calc(2));
        printf("当原函数中自变量取值为 3 时，多项式 A 取值为%.3lf\n",x.calc(3));
}
```