МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования   
**«Национальный исследовательский   
Нижегородский государственный университет им. Н.И. Лобачевского»**

**(ННГУ)**

**Институт информационных технологий, математики и механики**

**Отчет по лабораторной работе**

**«Арифметические операции с полиномами»**

**Выполнила:**

студентка группы 382003-1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Алясева Д.А.

**Проверил:**

к.т.н., доцент каф. МОСТ ИИТММ

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Мееров И.Б.

Рецензент

Нижний Новгород  
2022

**Содержание**

[**Содержание** 2](#_Toc100056167)

[Введение 3](#_Toc100056168)

[**1** Постановка задачи 4](#_Toc100056169)

[**2** Руководство программиста 5](#_Toc100056170)

[**2.1** Описание структуры программы 5](#_Toc100056171)

[**2.2** Описание структур данных 5](#_Toc100056172)

[**2.3** Описание алгоритмов 7](#_Toc100056173)

[**2.3.1.** Лексический и синтаксический анализ 7](#_Toc100056174)

[**2.3.2.** Добавление монома к полиному 8](#_Toc100056175)

[2.3.3. Слияние (сложение) двух полиномов. 8](#_Toc100056176)

[Заключение 9](#_Toc100056177)

[Список литературы 10](#_Toc100056178)

[**Приложение** 11](#_Toc100056179)

# Введение

В лабораторной работе рассматривается вопрос выполнения арифметических операций над полиномами с использованием линейной динамической структуры данных односвязный список.

Разработка приложения для работы с полиномами является практически значимой задачей, поскольку оно позволит облегчить, ускорить и оптимизировать многие математические подсчеты.

С целью реализации арифметической работы над полиномами в процессе выполнения данной лабораторной работы подробно описывается список как динамическая структура данных, а также рассматриваются и разбираются методы и алгоритмы решения основных задач (основанных на структуре данных список), вытекающих из вопроса вычислений, производимых над полиномами и мономами: лексический анализ и синтаксический анализ корректности введённых полиномов, приведение подобных слагаемых, сортировка мономов в составе полинома для более удобной работы с ним и т.д.

# Постановка задачи

В данной лабораторной работе необходимо разработать программу, выполняющую арифметические операции с полиномами, в которой должна быть реализована поддержка:

Работы с мономами трех переменных (x, y и z).

Примеры мономов: xyz, zx, x^2y, z^9, x^3z^7y^2.

Пользователь должен иметь возможность работать с полиномами, составленными из мономов от трех переменных с ограничением на степень каждой переменой от 0 до 9, степень будет храниться в свернутом виде, а именно в виде трехзначного числа (разряд сотен отвечает за степень переменной x, разряд десятков за степень переменной y, соответственно разряд единиц за степень z), при этом коэффициентами полинома могут быть как целые, так и вещественные числа. Десятичная часть числа отделяется точкой (например, 7.25, 0.43).

Операций сложения, вычитания, умножения на константу, умножения двух полиномов.

Пользователь имеет право работать с полиномами за счёт выполнения четырёх перечисленных операций (с контролем приведения подобных слагаемых, удаления монома с нулевым коэффициентом и сохранением ограничений на степень каждой переменной в составе монома). Должна быть гарантирована корректность выполняемых действий.

# Руководство программиста

## Описание структуры программы

С учетом сформулированных выше тезисов к реализации целесообразной представляется следующая структура программы:

* list - родительский класс, реализующий операции над односвязным списком. В данном классе содержится 2 вспомогательных класса-наследника: node - звено односвязного списка и iterator - указатель на звено односвязного списка, для наиболее эффективной и оптимальной работы со списком.
* polynomial – родительский класс, отвечающий непосредственно за арифметические операции над полиномами. В данном классе содержится дополнительный класс-наследник monomial, в котором описан набор инструментов для работы с мономами.

Работа класса polynomial основывается на структуре списка и представляет собой односвязный список из мономов, для каждого звена определён указатель на следующий моном.

Кроме того, есть модуль программ тестирования test\_polynomial.cpp.

Реализация классов находится в приложении.

## Описание структур данных

В данном отчёте используется такая динамическая структура данных как список. Прежде всего дадим определение структуре данных и динамической структуре данных.

Структура данных - есть модель данных в виде математической структуры

*,*

где – базисные множества, – отношения между элементами базисных множеств. [[1]](#footnote-1)

Динамическая структура есть математическая структура, которой соответствует частично-упорядоченное (по включению) базовое множество М, элементы которого являются структурами данных. При этом отношения включения индуцируются операциями преобразования структуры данных.

Связанный список (linked list) – это структура данных, в которой объекты расположены в линейном порядке. Однако в отличие от массива, в котором этот порядок определяется индексами, порядок в связанном списке определяется указателями на каждый объект. Связанные списки обеспечивают простое и гибкое представление динамических множеств.[1, c. 268].

В данном приложении используется односвязный список, в котором для каждого звена списка определён только указатель на следующее звено.

## Описание алгоритмов

### Лексический и синтаксический анализ

Алгоритм лексической и синтаксической проверки корректности введённой строки, задающей полином, а также сортировку, входящих в его состав мономов, реализует функция void translator(std::string input), описанная в приложении:

В начале создаем объект полином с помощью конструктора по умолчанию. В этот объект мы будем добавлять мономы, полученные при лексическом разборе введённой строки. При добавлении все мономы будут уже в отсортированном порядке (это мы увидим уже при анализе функции void insert(const monomial& m)).

* Первый элемент анализируем отдельно, так как он может начинаться не со знака, а сразу с коэффициента или переменной. Соответственно если первый элемент – цифра, то запоминаем весь коэффициент текущего монома, вызывая функцию std::stod и удаляя из начала скопированной строки столько элементов, сколько занимает число, и продолжаем анализ.
* Если следующий элемент является одной из двух бинарных операций, то мы считаем, что предыдущий моном завершился и добавляем этот моном в созданный объект класса полином (если это первый моном во входной строке, то добавляем “пустой” моном, т.е. моном с нулевым коэффициентом). При этом поле степени монома после его добавления заново инициализируется нулём.

Заметим, что введенный полином не может завершаться знаком.

Если элемент после знака является одной из трёх переменных (x, y или z), значит коэффициент перед мономом либо 1, либо -1. Удаляем знак бинарной операции и переходим к следующей итерации цикла.

Если элемент после знака – число, то обрабатываем его, как уже было описано выше.

Любой другой элемент после знака бинарной операции недопустим.

* Если следующий элемент – переменная, то для начала проверяем коэффициент монома на равенство нулю. Такое могло произойти если полином начался с переменной или в том случае если после знака перед мономом стоит переменная. В обоих случаях это означает, что коэффициент равен единице.
* Далее обрабатываем степень degree переменной монома, записанную после символа ‘^’. Она должна удовлетворять неравенству -1 < degree < 10 и быть целым числом. Если такого символа после переменной не стоит, то считаем степень переменной равной единице. Удаляем из начала строки рассмотренную переменную со своей степенью и переходим к следующей итерации.

Если на начале итерации нам встречается какой-то другой элемент (не описанный выше), то считаем, что введённый полином записан некорректно.

Не забываем после выхода из цикла добавить моном в созданный объект класса полином.

### Добавление монома к полиному

Для реализации данного алгоритма в функции void insert(const monomial& m) используется проход в цикле по итератору.

Обратим внимание, что если полином пуст, то можно сделать вставку монома в начало.

Сортировать мономы будем по трехзначному числу, представляющему собой степень переменных монома:

* Итак, если степени мономов совпали, то нет необходимости добавлять новый моном. Всё что нужно сделать – это сложить коэффициенты перед мономами. Однако если после произведённого сложения коэффициент стал равен нулю, то его необходимо удалить. Если такое произошло с первым элементом полинома, то удаляем моном на нулевой позиции (по нулевому указателю), иначе удаляем моном через указатель на удаляемое звено.
* В том случае, если степень нашего добавляемого монома строго больше степени монома на данной итерации, то запоминаем указатель и переходим к следующей итерации.
* Случай, когда степень монома на текущей итерации уже оказалась больше степени нашего монома, не оставляет нам ничего другого, кроме как вставить наш моном на место до текущего.

Если после прохода по циклу мы вышли, так и не вставив моном, то добавляем его в полином на последнюю позицию.

### Слияние (сложение) двух полиномов.

Основная сложность алгоритма заключается в поддержании отсортированного состояния результирующего полинома. Функция, реализующая данный алгоритм polynomial& polynomial::operator+=(const polynomial& p).

Создаём объект класса полином и идём в цикле по мономам обоих полиномов, участвующих в слиянии, с помощью двух итераторов (соответственно для первого и второго полинома), до тех пор, пока хотя бы один из двух полиномов не будет обработан полностью.

* Если во время сравнения двух мономов оказалось, что их степени равны между собой, то делаем вывод, что мономы - подобные слагаемые. Значит необходимо сложить коэффициенты перед ними и результат проверить на равенство нулю. По результатам проверки мы либо добавляем моном в созданный вспомогательный полином, либо просто инкрементируем итераторы и переходим к следующей итерации.
* Иначе, если степень (представляемая трехзначным числом) первого монома меньше степени второго, то определённо мы можем добавить первый моном в результирующий полином, инкрементируя при этом итератор для первого полинома.
* В обратном случае, когда степень первого монома больше степени второго, мы действуем аналогично, но уже для второго полинома.
* Как только хотя бы один полином мы обработали целиком, то остаточные слагаемые второго полинома просто добавляем в конец результирующего.

# Заключение

В ходе данной лабораторной работы было разработано приложение для работы с полиномами, которое поддерживает действия с десятичными и целыми коэффициентами мономов, операциями сложения, вычитания, умножения полинома на константу и перемножения двух полиномов. Были рассмотрены стандартные алгоритмы:

* синтаксическая проверка корректности введенного полинома,
* поддержка отсортированного состояния мономов, входящих в состав полинома,
* слияние двух упорядоченных списков (операция сложения полиномов) с дальнейшим приведением подобных слагаемых итогового полинома,
* контроль зануления коэффициента монома в процессе выполнения операций.

# Список литературы

1. Кормен Т. и др. Алгоритмы. Построение и анализ:[пер. с англ.]. – Издательский дом Вильямс, 2009.
2. Шилдт, Герберт. С++: базовый курс, 3-е изд. : Пер. с англ. – СПб.: ООО «Диалектика», 2019.-624 с.
3. Барышева И.В. и др. Под руководством Гергеля В.П. Учебно-методическое пособие. Нижний Новгород: Нижегородский университет, 2017.

**Приложение**

**polynomial.h**

#pragma once

#include <iostream>

#include <string>

#include <vector>

template <class T>

class list {

public:

class node {

public:

T data;

node\* next;

node(T value = 0, node\* next\_ = NULL) : next(next\_), data(value){}

};

node\* first;

size\_t size;

class iterator {

protected:

node\* ptr;

public:

iterator() : ptr(NULL) {}

iterator(node\* ptr\_) : ptr(ptr\_) {}

iterator(const iterator& iter) : ptr(iter.ptr) {}

iterator& operator++ ()

{

ptr = ptr->next;

return \*this;

}

iterator operator++ (int)

{

iterator iter(\*this);

++(\*this);

return iter;

}

node\*& operator\*() { return ptr; }

node\* operator->() { return ptr; }

bool operator== (iterator iter) const { return ptr == iter.ptr; }

bool operator!= (iterator iter) const { return !(\*this == iter); }

};

list() : first(NULL), size(0) {}

list(const list<T>& lst) : first(NULL), size(0) {

node\* pcurr = first;

//the order of monomials in a polynomial is determined during creation,

//the insertion of a monom does not violate the order

for (iterator it\_curr = lst.begin(); it\_curr != lst.end(); ++it\_curr)

pcurr = insert(pcurr, it\_curr->data); //return pointer to current monomial

}

~list() {

while (first) {

node\* tmp = first;

first = tmp->next;

delete tmp;

size--;

}

}

void print() {

iterator iter = begin();

while (iter != end()) {

std::cout << iter->data << " ";

iter++;

}

std::cout << std::endl;

}

node\* get\_first() { return first; }

iterator search(T& value) {

iterator it = begin();

while (it != end()) {

if (it->data == value)

break;

it++;

}

return it;

}

void erase(node\* pos) {

if (pos == NULL) {

if (!size) { throw "list is empty"; }

node\* tmp = first;

first = tmp->next;

delete tmp;

}

else {

node\* p = pos->next;

pos->next = p->next;

delete p;

}

size--;

}

iterator begin() const {

iterator it(first);

return it;

}

iterator end() const {

iterator it(NULL);

return it;

}

bool empty() { return size == 0; }

size\_t get\_size() { return size; }

node\* insert(node\* pos, const T& value) {

node\* tmp;

if (pos == NULL) {

tmp = new node(value, first);

first = tmp;

}

else {

tmp = new node(value, pos->next);

pos->next = tmp;

}

size++;

return tmp;

}

};

#define EXP 9

class polynomial {

public:

class monomial {

public: //degree - three-digit number

size\_t degree; //monomial degree in the range from 0 to 9 for each //variable (x, y or z)

double multiplier; //coefficient in front of the monomial

monomial() : degree(0), multiplier(0) {}

monomial(size\_t t\_degree, double t\_mult) : degree(t\_degree), multiplier(t\_mult) {}

monomial(const monomial& t\_monom) : degree(t\_monom.degree), multiplier(t\_monom.multiplier) {}

monomial& operator\*=(const monomial& m) {

//the digit of a three-digit number corresponds to the degree of one of the variables x,y,z

if (degree / 100 + m.degree / 100 > EXP || degree / 10 % 10 + m.degree / 10 % 10 > EXP \

|| degree % 10 + m.degree % 10 > EXP)

throw "too big degree of monomial";

else {

multiplier \*= m.multiplier;

degree += m.degree;

return (\*this);

}

}

monomial operator\*(const monomial& m) {

monomial tmp = (\*this);

tmp \*= m;

return tmp;

}

bool operator==(const monomial& m) const {

//take into account the error of "double" type

return (abs(multiplier - m.multiplier) <= 1e-10 && degree == m.degree);

}

bool operator!=(const monomial& m) const {

return !((\*this) == m);

}

void print() const {

if (multiplier < 0) std::cout << " - ";

//do not display 1 as a coefficient, only if degree == 0 (const)

if (abs(abs(multiplier) - 1) > 1e-10 || \

abs(abs(multiplier) - 1) < 1e-10 && degree == 0) std::cout << abs(multiplier);

int deg\_x = degree / 100, deg\_y = degree / 10 % 10, deg\_z = degree % 10;

if (deg\_x > 1) std::cout << "x^" << deg\_x;

else if (deg\_x == 1) std::cout << "x";

if (deg\_y > 1) std::cout << "y^" << deg\_y;

else if (deg\_y == 1) std::cout << "y";

if (deg\_z > 1) std::cout << "z^" << deg\_z;

else if (deg\_z == 1) std::cout << "z";

}

};

public:

// create polynomial as list of monomials

list<monomial> polynom;

public:

polynomial() {}

polynomial(const polynomial& p) : polynom(p.polynom) {}

polynomial(std::string s) {

//polynomial is created by string, all monomials in the polynomial are sorted

translator(s);

//print();

}

virtual ~polynomial() {

clear();

}

void print() const {

auto it = polynom.begin();

(\*it)->data.print();

++it;

while (it != polynom.end()) {

if ((\*it)->data.multiplier > 1e-10)

std::cout << " + ";

(\*it)->data.print();

++it;

}

std::cout << '\n';

}

void insert(const monomial& m) {

if (abs(m.multiplier) < 1e-10) return;

auto it = polynom.begin();

if (it == polynom.end()) {

polynom.insert(NULL, m);

return;

}

list<polynomial::monomial>::iterator ans = NULL;

auto prev\_it = it;

while (it != polynom.end()) {

if ((\*it)->data.degree == m.degree) {

(\*it)->data.multiplier += m.multiplier;

if (abs((\*it)->data.multiplier) < 1e-10) { //if multiplier == 0, delete this monomial

if (it == prev\_it && it == polynom.begin()) //if monomial on the first position

polynom.erase(NULL);

else polynom.erase((\*prev\_it));

}

return;

}

else {

if ((\*it)->data.degree < m.degree)

ans = it;

else {

polynom.insert(\*ans, m);

return;

}

}

prev\_it = it;

++it;

}

if (it == polynom.end())

polynom.insert(\*ans, m);

}

void translator(std::string input) {

bool first = true;

polynomial::monomial\* m = new polynomial::monomial();

while (!input.empty()) {

if (first) {

if (input[0] >= '0' && input[0] <= '9') {

std::string::size\_type sz;

double num = std::stod(input, &sz);

input.erase(0, sz);

m->multiplier = num;

}

first = false;

continue;

}

// beginning of a new monomial

if (input[0] == '+' || input[0] == '-') {

//add a new monomial if we see the operator "+" or "-"

insert(\*m);

m->degree = 0;

if (input.size() == 1) {

throw "incorrect input, check your input polynomial";

return;

}

if (input[1] == 'x' || input[1] == 'y' || input[1] == 'z') { //if there is no number after the sign,

if (input[0] == '-') m->multiplier = -1; //we assume that the multiplier is equal to one

else m->multiplier = 1;

input.erase(0, 1);

continue;

}

else {

if (input[1] >= '0' && input[1] <= '9') {

std::string::size\_type sz;

double num = std::stod(input, &sz);

input.erase(0, sz);

m->multiplier = num;

}

else {

throw "incorrect input, check your input polynomial";

return;

}

}

}

else

if (input[0] == 'x' || input[0] == 'y' || input[0] == 'z') {

if (m->multiplier == 0) m->multiplier = 1;

char tmp = input[0];

int deg = 1;

if (input[1] == '^') { //get the degree

input.erase(0, 2);

std::string::size\_type sz;

deg \*= std::stoi(input, &sz);

input.erase(0, 1);

if (deg < 0 || deg > EXP) {

throw "degree is incorrect";

return;

}

}

else input.erase(0, 1);

m->degree += tmp == 'x' ? deg \* 100 : tmp == 'y' ? deg \* 10 : deg;

}

else {

throw "incorrect input, check your input polynomial";

return;

}

}

insert(\*m);

}

polynomial& operator+=(const polynomial&);

polynomial operator+(const polynomial&);

polynomial& operator-=(const polynomial&);

polynomial operator-(const polynomial&);

polynomial& operator+=(const monomial&);

polynomial& operator\*=(const polynomial&);

polynomial operator\*(const polynomial&);

polynomial& operator\*=(const double&);

polynomial operator\*(const double&);

polynomial& operator=(const polynomial&);

bool operator==(const polynomial&) const;

bool operator!=(const polynomial&) const;

bool search(const monomial&);

void clear();

bool empty();

size\_t get\_count();

};

polynomial&

polynomial::operator+=(const polynomial& p)

{

polynomial ans;

auto it = polynom.begin(), it\_p = p.polynom.begin();

while (it != polynom.end() && it\_p != p.polynom.end()) {

if (it->data.degree == it\_p->data.degree) {

double mult = it->data.multiplier + it\_p->data.multiplier;

if (abs(mult) > 1e-10) {

polynomial::monomial tmp\_m(it->data.degree, mult);

ans.insert(tmp\_m);

}

++it, ++it\_p;

}

else {

if (it->data.degree < it\_p->data.degree) {

ans.insert((\*it)->data);

++it;

}

else {

ans.insert((\*it\_p)->data);

++it\_p;

}

}

}

if (it == polynom.end())

while (it\_p != p.polynom.end()) {

ans.insert((\*it\_p)->data);

++it\_p;

}

if (it\_p == p.polynom.end())

while (it != polynom.end()) {

ans.insert((\*it)->data);

++it;

}

\*this = ans;

return (\*this);

}

polynomial

polynomial::operator+(const polynomial& p)

{

polynomial tmp(\*this);

tmp += p;

return tmp;

}

polynomial&

polynomial::operator\*=(const double& num)

{

auto it = polynom.begin();

while (it != polynom.end()) {

it->data.multiplier \*= num;

++it;

}

return (\*this);

}

polynomial&

polynomial::operator-=(const polynomial& p)

{

polynomial tmp\_p = p;

tmp\_p \*= (const double)(-1);

(\*this) += tmp\_p;

return (\*this);

}

polynomial

polynomial::operator\*(const double& num)

{

polynomial tmp(\*this);

tmp \*= num;

return tmp;

}

polynomial

polynomial::operator-(const polynomial& p)

{

polynomial tmp(\*this);

tmp -= p;

return tmp;

}

polynomial&

polynomial::operator+=(const monomial& m)

{

insert(m);

return (\*this);

}

bool

polynomial::operator==(const polynomial& p) const

{

auto it = polynom.begin(), it\_p = p.polynom.begin();

while (it != polynom.end() && it\_p != p.polynom.end()) {

if (it->data != it\_p->data)

return false;

++it; ++it\_p;

}

if (it != polynom.end() || it\_p != p.polynom.end())

return false;

return true;

}

bool

polynomial::operator!=(const polynomial& p) const {

return !((\*this) == p);

}

polynomial&

polynomial::operator\*=(const polynomial& p) {

polynomial tmp;

auto it1 = polynom.begin(), it2 = p.polynom.begin();

while (it1 != polynom.end()) {

it2 = p.polynom.begin();

while (it2 != p.polynom.end()) {

tmp += (\*it1)->data \* (\*it2)->data;

++it2;

}

++it1;

}

\*this = tmp;

return \*this;

}

polynomial

polynomial::operator\*(const polynomial& p) {

polynomial tmp;

tmp = \*this;

tmp \*= p;

return tmp;

}

bool polynomial::search(const monomial& m)

{

auto it = polynom.begin();

while (it != polynom.end()) {

if ((\*it)->data == m)

return true;

else ++it;

}

return false;

}

void polynomial::clear()

{

size\_t count = polynom.size;

while (count) {

polynom.erase(NULL);

count--;

}

}

polynomial& polynomial::operator=(const polynomial& p) {

clear();

auto it = p.polynom.begin();

while (it != p.polynom.end()) {

insert((\*it)->data);

++it;

}

return \*this;

}

bool polynomial::empty() {

return polynom.size == 0;

}

size\_t polynomial::get\_count() {

return polynom.size;

}

1. Структура данных (data structure) – это способ хранения и организации данных, облегчающий доступ к этим данным и их модификацию **[1, c.30]**. [↑](#footnote-ref-1)