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**

**«Таблицы»**

**Выполнила:**

студентка группы 382003-1

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

**Проверила:**

преподаватель каф. МОСТ ИИТММ

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Арисова А.Н.

Нижний Новгород  
2022

**Содержание**

[Введение 3](#_Toc103986800)

[**1** Постановка задачи 4](#_Toc103986801)

[**2** Руководство программиста 5](#_Toc103986802)

[**2.1** Описание структуры программы 5](#_Toc103986803)

[**2.2** Описание структур данных 6](#_Toc103986804)

[**2.3** Описание алгоритмов 8](#_Toc103986805)

[**2.3.1** Поиск элемента в таблице по ключу 8](#_Toc103986806)

[**2.3.2** Добавление элемента в таблицу 8](#_Toc103986807)

[**2.3.3.** Удаление элемента из таблицы по ключу 9](#_Toc103986808)

[Заключение 11](#_Toc103986809)

[Список литературы 12](#_Toc103986810)

[**Приложение** 13](#_Toc103986811)

# Введение

В лабораторной работе рассматривается вопрос разработки приложения для хранения полиномов в таблицах трёх типов (неупорядоченная таблица, упорядоченная таблица и хеш-таблица).

С целью реализации приложения в процессе выполнения данной лабораторной работы подробно описываются используемые структуры данных, а также рассматриваются и разбираются методы и алгоритмы решения основных задач (основанных на структуре данных таблица (неупорядоченная, упорядоченная, хеш-таблица)), вытекающих из вопроса взаимодействия с хранящимися данными: добавление в таблицу нового полинома с уникальным ключом, удаление полинома из таблиц по ключу, поиск по ключу в таблицах всех трех типов.

# Постановка задачи

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

Работа происходит сразу с таблицами всех типов.

Любые действия, производимые с полиномами (добавление полинома в таблицу, удаление, поиск) должны осуществляться сразу в трёх таблицах разных типов.

Пользователь может положить в таблицу свой полином, с заданием ему некоторого наименования (ключа).

Пользователь может запросить (найти в/извлечь из) таблицы полином с заданным наименованием.

В ходе выполнения программы у пользователя должна оставаться возможность производить операции с полиномами, а также размещать полученный результат в таблицу.

Во время работы программы происходит логирования количества произведенных операций.

Пример: Пользователь ввел свой полином и запросил положить его в таблицу, программа в ответ выводит на экран или в файл, какое действие произошло и сколько операций потребовалось на нее у разных типов таблиц. Аналогично с поиском и извлечением.

# Руководство программиста

## Описание структуры программы

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

* \_table\_interface – абстрактный родительский класс, определяющий основные виртуальные методы классов потомков. В данном классе содержится вспомогательный класс-наследник: cell – ячейка таблицы с полями ключ и значение для наиболее эффективной и оптимальной работы с таблицами.
* unordered\_table – шаблонный класс неупорядоченная таблица, наследуемый от абстрактного \_table\_interface, в нём переопределяются все виртуальные функции абстрактного класса согласно концепции неупорядоченной таблицы.
* ordered\_table – шаблонный класс упорядоченная таблица, наследуемый от класса unordered\_table. Поля класса остаются неизменными аналогично классу unordered\_table, но методы, определённые в родительском классе, переопределяются.
* hash\_table – шаблонный класс хеш-таблица, наследуемый от абстрактного \_table\_interface, в нём переопределяются все виртуальные функции абстрактного класса согласно концепции хеш-таблицы, также определяются public и private поля.

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

Реализация классов находится в приложении.

## Описание структур данных

***Таблица*** – это список, состоящий из конечного множества элементов, при чем каждый элемент характеризуется рядом признаков (свойств). Один из признаков, называемый ключом, позволяет отличить один элемент от другого (идентифицировать элемент). Ключ может однозначно определять элемент таблицы (ключи всех элементов различны) или неоднозначно (в таблице есть элементы с равными ключами).

Все действия над элементами выполняются в соответствии с их ключами: по ключу элементы выбираются из таблицы и добавляются в нее.

***Неупорядоченная таблица*** – это таблица, элементы которой располагаются в порядке их поступления в таблицу.

***Упорядоченная таблица*** – это таблица, между элементами которой установлено отношение порядка. Это отношение, как правило, устанавливается на признаке ключ. Поэтому говорят, что таблица упорядочена по ключам элементов. Таблица упорядочена по возрастанию значений ключа, если ключ ( < ключ() для всех i = 1, 2 , . . . , n-1 (здесь - i-й элемент таблицы T). Таблица упорядочена по убыванию значений ключа, если ключ () > ключ () для всех i = 1, 2, . . . , n-1.

***Хеш-таблица*** — это структура данных, в которой все элементы хранятся в виде пары ключ-значение, где:

* ключ — уникальное число, которое используется для индексации значений;
* значение — данные, которые с этим ключом связаны.

*Хеширование (хеш-функция)*

В хеш-таблице обработка новых записей производится при помощи ключей. А элементы, связанные с этим ключом, сохраняются в ячейке хеш-таблицы с полученным с помощью хеш-функции индексом. Этот процесс называется хешированием.

Пусть k — ключ, а h(x) — хеш-функция.

Тогда h(k) в результате даст индекс, в котором мы будем хранить элемент, связанный с ключом k. (рис. 1)

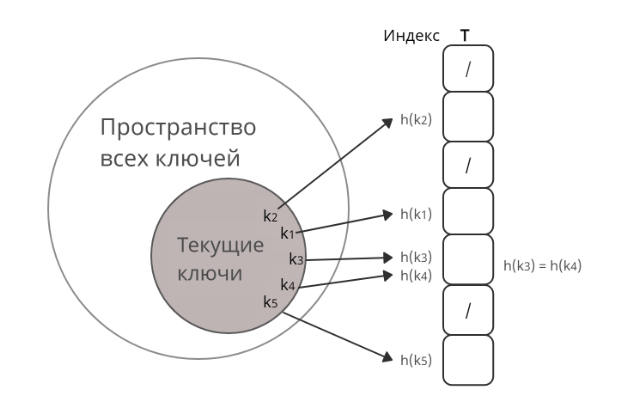


Рисунок 1

*Коллизии*

Когда хеш-функция генерирует один индекс для нескольких ключей, возникает конфликт: неизвестно, какое значение нужно сохранить в этом индексе. Это называется коллизией хеш-таблицы.

Есть несколько методов борьбы с коллизиями, один из которых – метод цепочек, реализуемый в данной лабораторной работе.

*Метод цепочек*

Суть этого метода проста: если хеш-функция выделяет один индекс сразу двум элементам, то храниться они будут в одном и том же индексе, но уже с помощью связного списка. (рис. 2)

Изображение выглядит как текст, электроника

Автоматически созданное описание

Рисунок 2

## Описание алгоритмов

### Поиск элемента в таблице по ключу

*Неупорядоченная таблица*

Поиск одного элемента в неупорядоченной таблице по заданному ключу осуществляется последовательным просмотром элементов или до нахождения искомого уникального ключа, или до конца таблицы, если искомый ключ не найден. Возвращаемым значением является указатель на найденный элемент с заданным ключом, либо признак отсутствия элемента.

Оценка сложности поиска. Минимальное время поиска, когда элемент находится в начале таблицы, О(1), максимальное время О(n), n – число элементов таблицы.

*Упорядоченная таблица*

Так как все элементы в таблице упорядочены по возрастанию ключа, то самым экономичным является бинарный поиск в упорядоченной таблице. Он состоит в последовательном делении таблицы пополам и определении, в какой из двух частей находится искомая запись. Последующему делению каждый раз подвергается часть, содержащая искомый ключ. Поскольку таблица упорядочена по возрастанию ключа, то чтобы определить, в какой части находится требуемое значение ключа, достаточно сравнить значение ключа в точке деления с искомым.

Оценка сложности бинарного поиска для n записей в таблице имеет порядок O(.

*Хеш-таблица*

Поиск элемента в хеш-таблице по заданному ключу осуществляется определением индекса в хеш-таблице с помощью хеш-функции. Далее выполняется линейный обход связного списка, определённого методом цепочек, до момента нахождения элемента с заданным ключом или до момента прохождения всего списка, в случае если элемент с данным уникальным ключом отсутствует в таблице. Таким образом, при определении оптимальной хеш-функции и минимальном количестве коллизий средняя сложность операции поиска будет О(1).

### Добавление элемента в таблицу

*Неупорядоченная таблица*

В приведённой реализации неупорядоченной таблицы запись (ключ, элемент (полином)) просто вставляется в конец таблицы.

*Упорядоченная таблица*

В реализации, представленной в приложении к лабораторной работе, вставка в упорядоченную таблицу осуществляется по следующему несложному алгоритму:

* Осуществляется поиск позиции, на которую необходимо вставить новую запись с данным ключом.
* Если на данной позиции уже существует записанный элемент с таким же ключом, то пользователь получает сообщение об ошибке и ему предлагается повторный ввод.
* Если элемент-полином с данным ключом ещё не был добавлен, то он добавляется в конец упорядоченной таблицы и с помощью функции std::swap() осуществляется перенос элемента на нужную позицию.

*Хеш-таблица*

* С помощью заранее определённой функции поиска осуществляется проверка наличия записи с ключом, равным тому, новую запись с которым необходимо вставить. В случае нахождения полинома с таким ключом выводится сообщение об ошибке, пользователю предлагается повторный ввод.
* Если такого элемента нет в таблице, то прежде всего с помощью хэш-функции определяется индекс в хеш-таблице.
* Элемент с уникальным ключом добавляется в конец линейного связного списка.

### Удаление элемента из таблицы по ключу

*Неупорядоченная таблица*

* В приведённой реализации неупорядоченной таблицы для удаления записи в таблице по ключу осуществляется линейный обход всех записей таблицы до момента нахождения требуемого элемента. На позицию удаляемого элемента помещается последняя по индексу запись, а сам последний элемент удаляется.
* Если удаление не произошло, то выводится сообщение об отсутствии записи с данным ключом в таблице.

*Упорядоченная таблица*

* С помощью функции поиска, определяемой выше, получаем указатель на удаляемый элемент.
* Функцией std::swap() перемещаем элемент в конец таблицы.
* Удаляем последний элемент таблицы.

*Хеш-таблица*

* Определяем индекс позиции в таблице с помощью хеш-функции.
* Идём по линейному связному списку до тех пор, пока не найдём элемент с искомым индексом и запоминаем указатель на него. Если же элемент не был найден, выводим об этом сообщение пользователю.
* Удаляем с помощью полученного указателя элемент из списка.

# Заключение

В ходе данной лабораторной работы было расширено приложение, разработанное для работы с полиномами (которое поддерживает действия с десятичными и целыми коэффициентами мономов, операциями сложения, вычитания, умножения полинома на константу и перемножения двух полиномов). Благодаря наличию трёх видов таблиц: неупорядоченной, упорядоченной и хеш-таблицы, появилась возможность добавлять полиномы в таблицы, удалять их по уникальному ключу и осуществлять поиск по ключу, кроме того, для всех видов операций по работе с таблицами была организована оценка количества операций, выведенная в отдельный файл для удобства пользователя.

# 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**

**table.h**

#pragma once

#include <iostream>

#include <string>

#include <vector>

#include <initializer\_list>

#include <algorithm>

#include <cmath>

#include "polynomial.h"

#include <fstream>

#define GET\_COMPLEXITY

#ifdef GET\_COMPLEXITY

size\_t ops\_find = 0;

size\_t ops\_insert = 0;

size\_t ops\_erase = 0;

void get\_complexity\_insert() {

std::ofstream out;

out.open("log.txt", std::ios::app);

out << std::endl;

out << "Number of commands, executed the last time insert function was called:\n";

out << ops\_insert << std::endl;

out.close();

}

void get\_complexity\_find() {

std::ofstream out;

out.open("log.txt", std::ios::app);

out << std::endl;

out << "Number of commands, executed the last time the search function was called:\n";

out << ops\_find << std::endl;

out.close();

}

void get\_complexity\_erase() {

std::ofstream out;

out.open("log.txt", std::ios::app);

out << std::endl;

out << "Number of commands, executed the last time erase function was called:\n";

out << ops\_erase << std::endl;

out.close();

}

#endif

//====================== abstract class \_table\_interface and class cell =========================

template <class T>

class \_table\_interface {

public:

class cell {

public:

size\_t key;

T value;

cell() {}

cell(size\_t \_key, T \_value) :key(\_key), value(\_value){}

cell(const std::pair<size\_t, T>& p) :key(p.first), value(p.second) {}

cell(cell&& c) :key(std::move(c.key)), value(std::move(c.value)) {}

cell(const cell& c) :key(c.key), value(c.value) {}

virtual ~cell(){}

cell& operator=(const cell& c) {

key = c.key;

value = c.value;

return \*this;

}

bool operator==(const cell& c) {

return (key == c.key && value == c.value);

}

bool operator!=(const cell& c) {

return !(\*this == c);

}

};

virtual void clear() = 0;

virtual bool erase(size\_t \_key) = 0;

virtual bool insert(size\_t \_key, const T& \_val) = 0;

virtual \_table\_interface<T>::cell\* find(size\_t key) = 0;

};

template <class T>

using t\_cell = typename \_table\_interface<T>::cell;

//===================================== class unordered\_table ===================================

template <class T>

class unordered\_table: public \_table\_interface<T> {

public:

std::vector<t\_cell<T>> table;

size\_t size;

unordered\_table() : size(0) {}

unordered\_table(const std::initializer\_list<\_table\_interface<T>::cell>& l) : table(l), size(l.size()) {}

unordered\_table(const unordered\_table<T>&& t) : table(std::move(t.table)), size(std::move(t.size)) {}

unordered\_table(const unordered\_table<T>& t) : table(t.table), size(t.size) {}

virtual ~unordered\_table() {

clear();

}

bool insert(size\_t key, const T& val) override {

if (find(key) == nullptr) {

table.push\_back(t\_cell<T>(key, val));

size++;

#ifdef GET\_COMPLEXITY

ops\_insert = 1;

std::ofstream out;

out.open("log.txt", std::ios::app);

out << "\nUNORDERED TABLE";

out.close();

get\_complexity\_insert();

#endif

return true;

}

else return false;

}

bool erase(size\_t key) override {

bool deleted = false;

#ifdef GET\_COMPLEXITY

ops\_erase = 0;

#endif

for (auto& x : table) {

#ifdef GET\_COMPLEXITY

++ops\_erase;

#endif

if (x.key == key) {

// also work correct for size == 1

x = table[size - 1];

table.pop\_back();

size--;

deleted = true;

#ifdef GET\_COMPLEXITY

ops\_erase++;

std::ofstream out;

out.open("log.txt", std::ios::app);

out << "\nUNORDERED TABLE";

out.close();

get\_complexity\_erase();

#endif

return true;

}

}

if (!deleted) return false;

}

t\_cell<T>\* find(size\_t key) override {

#ifdef GET\_COMPLEXITY

ops\_find = 0;

#endif

for (auto& x : table) {

#ifdef GET\_COMPLEXITY

++ops\_find;

#endif

if (x.key == key) {

#ifdef GET\_COMPLEXITY

std::ofstream out;

out.open("log.txt", std::ios::app);

out << "\nUNORDERED TABLE";

out.close();

get\_complexity\_find();

#endif

return &x;

}

}

#ifdef GET\_COMPLEXITY

std::ofstream out;

out.open("log.txt", std::ios::app);

out << "\nUNORDERED TABLE";

out.close();

get\_complexity\_find();

#endif

return nullptr;

}

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

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

void clear() { table.clear(); }

T& operator[](size\_t \_key) {

auto ptr = find(\_key);

if (ptr != nullptr)

return (\*ptr).value;

else throw "Incorrect access by key to a table cell\n";

}

};

//===================================== class ordered\_table =====================================

template <class T>

class ordered\_table : public unordered\_table<T> {

public:

std::vector<t\_cell<T>> table;

size\_t size;

ordered\_table() : size(0) {}

ordered\_table(const std::initializer\_list<t\_cell<T>>& l) :table(l), size(l.size()) {

std::sort(table.begin(), table.end(), [](const t\_cell<T>& first, const t\_cell<T>& second) {

return first.key < second.key;

});

}

ordered\_table(const unordered\_table<T>& t) :table(t.table), size(table.size()) {

std::sort(table.begin(), table.end(), [](const t\_cell<T>& first, const t\_cell<T>& second) {

return first.key < second.key;

});

}

ordered\_table(unordered\_table<T>&& t) :table(std::move(t.table)), size(std::move(table.size())) {

std::sort(table.begin(), table.end(), [](const t\_cell<T>& first, const t\_cell<T>& second) {

return first.key < second.key;

});

}

ordered\_table(const ordered\_table<T>& t) :table(t.table), size(table.size()) {}

ordered\_table(ordered\_table<T>&& t) :table(std::move(t.table)), size(std::move(table.size())) {}

virtual ~ordered\_table() {

clear();

}

void clear() { table.clear(); }

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

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

t\_cell<T>\* find(size\_t key) override {

//binary\_search

auto it = std::lower\_bound(table.begin(), table.end(), t\_cell<T>(key, T()),

[](const t\_cell<T>& first, const t\_cell<T>& second) {

return first.key < second.key;

});

#ifdef GET\_COMPLEXITY

if (table.size() == 1)

ops\_find = 1;

else if (table.size() > 1)

ops\_find = log2(table.size());

std::ofstream out;

out.open("log.txt", std::ios::app);

out << "\nORDERED TABLE";

out.close();

get\_complexity\_find();

#endif

if (it == table.end() || it->key != key)

return nullptr;

else return &(\*it);

}

bool insert(size\_t \_key, const T& \_value) override {

//binary\_search

auto it = std::lower\_bound(table.begin(), table.end(), t\_cell<T>(\_key, T()),

[](const t\_cell<T>& first, const t\_cell<T>& second) {

return first.key < second.key;

});

if (it != table.end() && it->key == \_key)

return false;

size\_t pos = it - table.begin();

table.emplace\_back(t\_cell<T>({ \_key, \_value }));

size++;

for (int i = size - 1; i > pos; --i)

std::swap(table[i], table[i - 1]);

#ifdef GET\_COMPLEXITY

//find + emplace\_back

if (table.size() == 1)

ops\_insert = 2;

else if (table.size() > 1)

ops\_insert = log2(table.size()) + 1;

//swap

ops\_insert += (size - 1 > pos) ? size - 1 - pos : 0;

std::ofstream out;

out.open("log.txt", std::ios::app);

out << "\nORDERED TABLE";

out.close();

get\_complexity\_insert();

#endif

return true;

}

bool erase(size\_t key) override {

auto it = find(key);

if (it != nullptr) {

int pos = it - &table[0];

for (int i = pos; i < size - 1; ++i)

std::swap(table[i], table[i + 1]);

table.pop\_back();

#ifdef GET\_COMPLEXITY

//find + pop\_back

ops\_erase = ops\_find + 1;

//swap

ops\_erase += (size - 1 > pos) ? size - 1 - pos : 0;

std::ofstream out;

out.open("log.txt", std::ios::app);

out << "\nORDERED TABLE";

out.close();

get\_complexity\_erase();

#endif

}

else return false;

size--;

return true;

}

T& operator[](size\_t \_key) {

auto ptr = find(\_key);

if (ptr != nullptr)

return (\*ptr).value;

else throw "Incorrect access by key to a table cell\n";

}

};

//======================================= class hash\_table ======================================

#define NUM\_BUCKETS 1000

#define PRIME 1009

template <class T>

class hash\_table: public \_table\_interface<T> {

public:

std::vector<List<t\_cell<T>>> table;

size\_t size;

private:

size\_t a, b;

private:

size\_t hash\_function(size\_t \_key) {

return ((a \* \_key + b) % PRIME) % NUM\_BUCKETS;

}

public:

hash\_table() :size(0) {

srand(time(NULL));

a = rand() % PRIME;

b = rand() % PRIME;

table.resize(NUM\_BUCKETS);

}

hash\_table(const hash\_table<T>& t) :table(t.table), size(t.size), a(t.a), b(t.b) {}

hash\_table(hash\_table<T>&& t) :table(std::move(t.table)), size(std::move(t.size)), a(t.a), b(t.b) {}

hash\_table(const std::initializer\_list<t\_cell<T>>& l) {

table.resize(NUM\_BUCKETS);

srand(time(NULL));

a = rand() % PRIME;

b = rand() % PRIME;

size = 0;

for (auto& val : l)

insert(val.key, val.value);

}

virtual ~hash\_table() {

clear();

}

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

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

void clear() override {

for (size\_t i = 0; i < NUM\_BUCKETS; ++i)

table[i].clear();

table.clear();

}

t\_cell<T>\* find(size\_t \_key) override {

size\_t hash = hash\_function(\_key);

#ifdef GET\_COMPLEXITY

ops\_find = 1;

#endif

for (auto it = table[hash].begin(); it != table[hash].end(); ++it) {

#ifdef GET\_COMPLEXITY

++ops\_find;

#endif

if (it->data.key == \_key) {

#ifdef GET\_COMPLEXITY

std::ofstream out;

out.open("log.txt", std::ios::app);

out << "\nHASH TABLE";

out.close();

get\_complexity\_find();

#endif

return &(it->data);

}

}

#ifdef GET\_COMPLEXITY

std::ofstream out;

out.open("log.txt", std::ios::app);

out << "\nHASH TABLE";

out.close();

get\_complexity\_find();

#endif

return nullptr;

}

bool insert(size\_t \_key, const T& \_val) override {

auto it = find(\_key);

if (it != nullptr)

return false;

size\_t hash = hash\_function(\_key);

t\_cell<T> new\_node(\_key, \_val);

table[hash].insert(0, new\_node);

size++;

#ifdef GET\_COMPLEXITY

ops\_insert = ops\_find + 1;

std::ofstream out;

out.open("log.txt", std::ios::app);

out << "\nHASH TABLE";

out.close();

get\_complexity\_insert();

#endif

return true;

}

bool erase(size\_t \_key) override {

typename List<t\_cell<T>>::iterator t = NULL;

bool f = 0;

size\_t hash = hash\_function(\_key);

#ifdef GET\_COMPLEXITY

ops\_erase = 1;

#endif

for (auto it = table[hash].begin(); it != table[hash].end(); ++it) {

#ifdef GET\_COMPLEXITY

++ops\_erase;

#endif

if (it->data.key == \_key) {

f = 1;

break;

}

else t = it->next;

}

if (f == 0) return false;

table[hash].erase(\*t);

size--;

#ifdef GET\_COMPLEXITY

++ops\_erase;

std::ofstream out;

out.open("log.txt", std::ios::app);

out << "\nHASH TABLE";

out.close();

get\_complexity\_erase();

#endif

return true;

}

T& operator[](size\_t \_key) {

auto ptr = find(\_key);

if (ptr != nullptr)

return (\*ptr).value;

else throw "Incorrect access by key to a table cell\n";

}

};