МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ 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**

**Кафедра математического обеспечения и суперкомпьютерных технологий**

Направление подготовки «Прикладная математика и информатика»

**ОТЧЕТ**

по учебной практике

**Организация доступа к данным по ключу**

**Выполнил:** студент группы 381603-1

Хорькин А.С.

**Проверила:** к.т.н.,доцент каф. МОСТ института ИТММ

Кустикова В.Д.

Нижний Новгород  
2018

**Содержание**

[Введение 3](#_Toc515581666)

[1 Постановка задачи 5](#_Toc515581667)

[2 Руководство пользователя 6](#_Toc515581668)

[3 Руководство программиста 7](#_Toc515581669)

[3.1 Описание структуры программы 7](#_Toc515581670)

[3.2 Описание структуры данных 7](#_Toc515581671)

[3.3 Описание программной реализация 9](#_Toc515581672)

[Заключение 11](#_Toc515581673)

[Список литературы 12](#_Toc515581674)

[Приложение 13](#_Toc515581675)

[Приложение A. Tables.h. 13](#_Toc515581676)

[Приложение B. TablesUnord.h 15](#_Toc515581677)

[Приложение C. TablesOrdered.h 16](#_Toc515581678)

[Приложение D. HashTables.h 18](#_Toc515581679)

[Приложение E. main.cpp 20](#_Toc515581680)

[Приложение F. Test\_all.cpp Тесты программы 22](#_Toc515581681)

# Введение

В настоящее время таблицы, хранящие какие-либо данные, очень широко используются в современном мире как отличное средство хранения и представления информации. Таблицы широко используются в банковской сфере, бухгалтерском и банковском учете, планировании и распределении ресурсов, обработке больших массивов информации. Они удобны для быстрого поиска информации, её обработки и сортировки по различным фильтрам и пользовательским настройкам.

Необходимо было реализовать программу, которая выполняет 3 арифметические операции: вставка, удаление, поиск элемента в различных таблицах 3х видов (hash, сортированные, несортированные), хранящих в себе ключ и значение. Необходимо обеспечить уникальность ключа в ходе работы программы и быстродействие этих трёх основных операций, для этого при необходимости таблицы могут менять свой размер.

# Постановка задачи

Формулировка задачи:

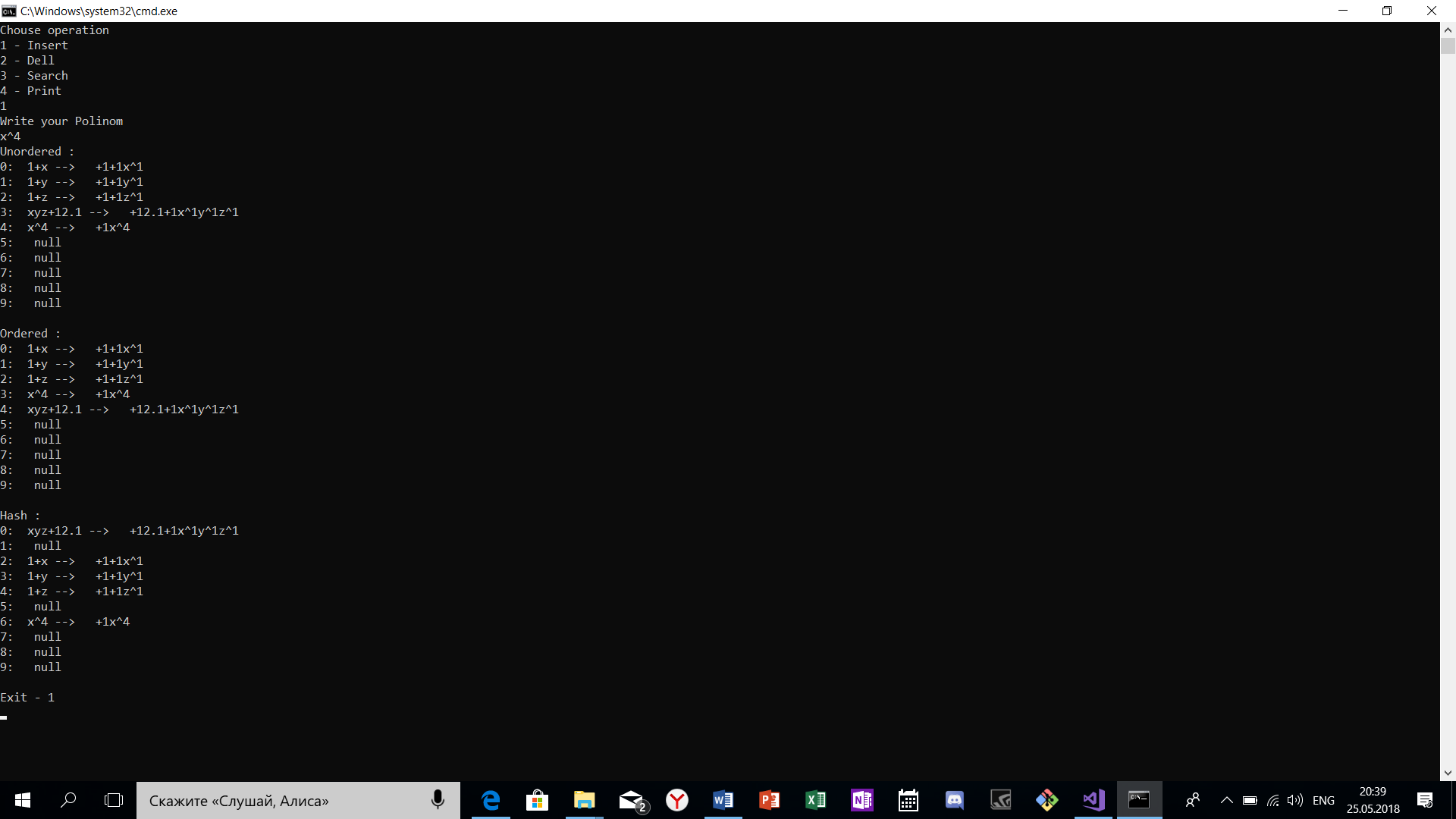
Необходимо было реализовать программу, которая выполняет 3 арифметические операции: вставка, удаление, поиск элемента в различных таблицах 3х видов (hash, сортированные, несортированные), хранящих в себе ключ - строка и значение - полином. Необходимо обеспечить уникальность ключа в ходе работы программы и быстродействие этих трёх основных операций, для этого при необходимости таблицы могут менять свой размер. В программе могут возникать коллизии, для борьбы с ними можно использовать два метода, а именно метод цепочек или открытого перемешивания. Так как все структуры, включающиеся в Polimon, включая и его, уже были описаны в отчете по лабораторной работе «Арифметические операции с полиномами», то упоминания об этом здесь не будет.

Исходные данные:

Размер таблиц, полиномы, которые будут храниться в таблице.

Требуемый результат:

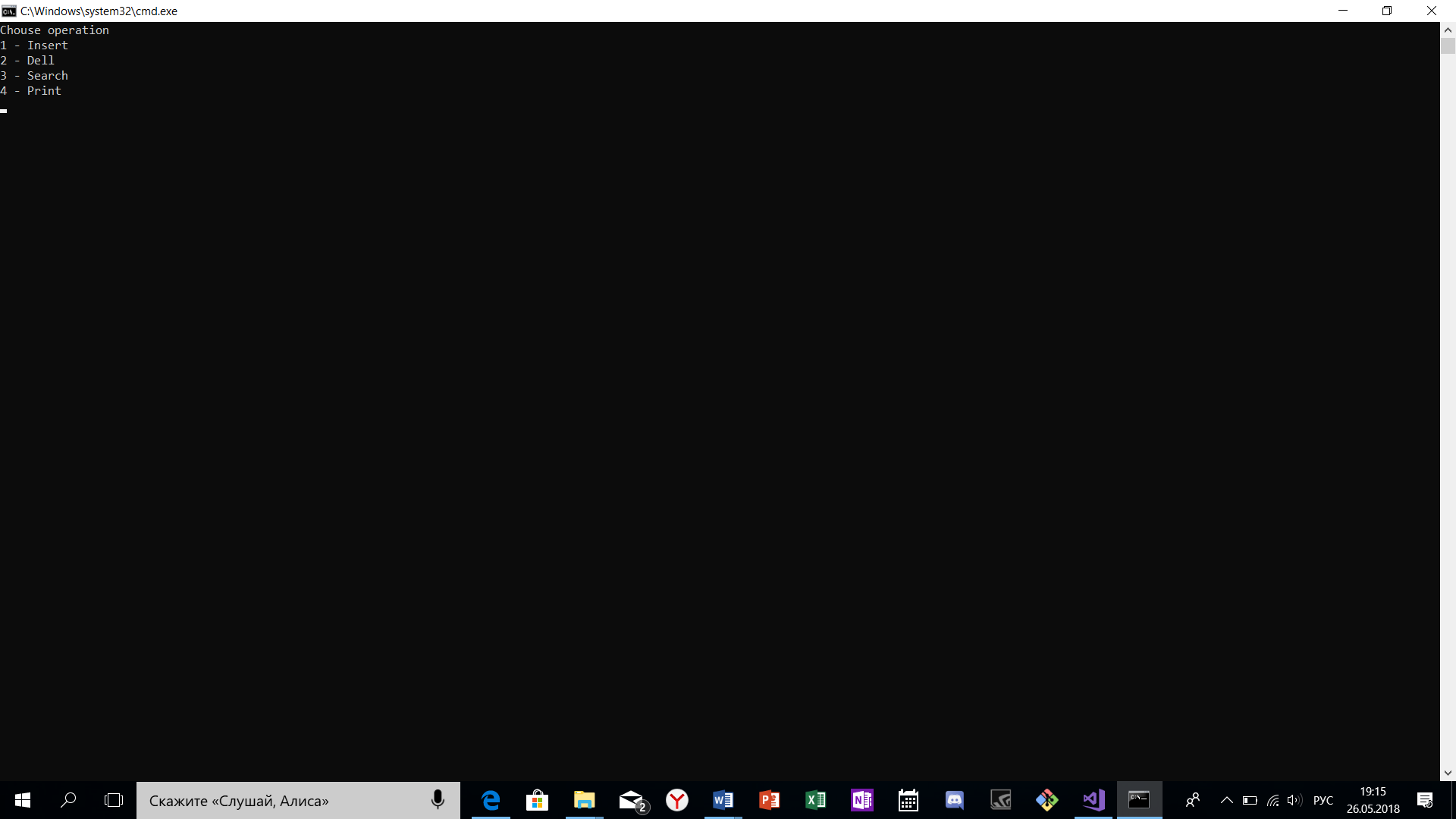
Программа правильно выполняет вставку, поиск и удаление элементов, заданных пользователем.



1. Пример работы программы.

# Руководство пользователя

Для начала работы программы необходимо открыть файл **Sample.exe.** После запуска программа предложит ввести вам размер для тамблиц, которые будут созданы автоматически. Далее вам будет предложено одно из четырёх действий: вставка, удаление, поиск, печать таблицы. Чтобы выбрать действие нужно ввести соответствующую цифру.



1. Выбор операции

После пользователю выводится результат – либо таблица, либо найденные элементы, либо сообщения об ошибке. Если пользователь хочет завершить работу программы, то необходимо ввести 1, иначе – любое число.

# Руководство программиста

## Описание структуры программы

**Библиотека Tables** состоит из:

Tables.h – заголовочный файл, который содержит шаблонный, абстрактный класс таблиц

TablesHash.h – шаблонный класс, унаследованный от абстрактного класса таблиц.

TablesOrdered.h – наследник абстрактного класса таблиц

TablecUnord.h – наследник абстрактного класса таблиц

## Описание структуры данных

Структура DataTable:

Хранит два поля: данные и ключ.

Абстрактный, шаблонный класс таблиц – Table содержит реализацию методов навигации и переупаковки, при которой данные копируются в массив большего размера.

Класс несортированных таблиц – TablesUnord

Вставка:

Пользователь вводит входные данные – полином, его строка считается ключом. Если до заполнения таблицы осталось 30 процентов происходит увеличение размера массива примерно в 1,5 раза. Старые данные копируются, чтобы не потерять их. Сначала по массиву происходит поиск совпадающих ключей, если таких нет, то вставляется в первую свободную ячейку, начиная сначала, иначе срабатывает исключение. Переменная Top увеличивается на единицу.

Поиск:

Пользователь вводит ключ. Далее массив элементов просматривается для выявления совпадений ключей. Если ключ совпал, то функция возвращается адрес на структуру DataTable, иначе вызывается исключение.

Удаление:

Пользователь вводит ключ. Происходит поиск элемента, если он найден, то происходит его удаление и обнуление адреса данной ячейки, иначе срабатывает исключение.

Класс сортированных таблиц – TablesOrdered

Вставка:

Пользователь вводит входные данные – полином, его строка считается ключом. Если до заполнения таблицы осталось 30 процентов происходит увеличение размера массива примерно в 1,5 раза. Старые данные копируются, чтобы не потерять их. Функция вставки используя алгоритм бинарного поиска находит индекс, куда необходимо вставить элемент. Если ячейка пуста, то элемент занимает своё место, иначе происходит сдвиг всех элементов массива на одну ячейку, начиная с конца. Переменная Top увеличивается на единицу. При просмотре ячейки на содержание элемента проверяется равенство ключей, если они равно, вызывается исключение.

Поиск:

Пользователь вводит ключ. Ключ передаётся в функцию бинарного поиска, которая возвращает индекс предполагаемого местонахождения элемента. Если ключи совпадают возвращается адрес DataTable, хранящийся в ячейке, иначе вызывается исключение.

Удаление:

Пользователь вводит ключ. Ключ передаётся в функцию бинарного поиска, которая возвращает индекс предполагаемого местонахождения элемента. Если ключи совпадают, то происходит удаление элемента и сдвиг всех элементов массива на одну ячейку, начиная с места удаления. Top уменьшается на единицу.

Класс хэш таблиц – HashTable

Вставка:

Пользователь вводит входные данные – полином, его строка считается ключом. Если до заполнения таблицы осталось 30 процентов происходит увеличение размера массива примерно в 1,5 раза. Старые данные копируются, чтобы не потерять их. Хэш функция, которая принимает ключ, переводит его в число, это число будет использоваться для вставки как индекс в массиве. Если ячейка свободна – элемент вставляется, переменная, которая хранит количество вставленных элементов, увеличивает своё значение на единицу, затем ячейка помечается как занятая. Если ячейка была удалена происходит тоже самое, за исключение того, что переменная не увеличивает своё значение. Если ячейка занята и ключи равны, то вызывается исключение. Иначе происходит поиск свободных ячеек с помочью функции rand (), seed которой равен индексу этой занятой ячейки, до тех пор, пока элемент, находящийся по сгенерированному индексу, занят или если ключи равно то вызывается исключение. Далее ячейка либо свободна, либо была удалена, алгоритм для этих случаев был описан выше.

Удаление:

Пользователь вводит ключ. Ключ передаётся в хэш функцию. Если по вычисленному индексу ячейка свободна, то вызывается исключение, иначе если ключи совпадают, функция возвращает адрес на DataTable, который находился по этому индексу. Если ключ не совпадает или ячейка была удалена, то происходит вычисление предполагаемого места нахождения с помощью функции rand (), seed которой равен индексу этой ячейки до тех пор, пока ячейка будет с искомым ключом, или пустая, в этом случаи вызывается исключение.

Поиск:

Аналогично удалению, только возвращается адрес, лежащий по индексу найденной ячейки.

## Описание программной реализация

Класс Table – абстрактный, шаблонный класс таблиц.

*Поля:*

DataTable <KeyType, DataType> \*\* DT – массив из указателей на DataTable

int MaxSize; – размер таблицы

virtual void Reallocate() = 0 – переупаковка

int curindex – индекс текущего элемента

int Top – индекс, или количество последнего или вставленных элемента(ов).

*Методы:*

Table(int SizeTable = MIN\_SIZE); – конструктор

virtual ~Table();– диструктор

virtual void Insert(const KeyType &KT\_T, const DataType &DT\_D) =0; – вставка

virtual void Dell(const KeyType &KT\_T) = 0; – удаление

virtual DataTable<KeyType, DataType> \* Search(const KeyType &KT\_T) const = 0 – поиск

void Reset();– переход в начало таблицы

bool IsEnded() const – проверка на конец таблицы

void GetNext()– увеличение текущего индекса

DataType GetCurData() const; – получение данных в текущей ячейке

friend std::ostream& operator<<(std::ostream& os, const Table<KeyType, DataType>&P); – оператор вывода

Класс TableUnord – класс таблиц, наследник потомок класса Table не содержит новый полей и методов.

Класс TableOrdered – класс таблиц, наследник потомок класса Table

*Новые методы:*

int BinarSearch(const KeyType &Key) const; – бинарный поиск

Класс HashTable – класс таблиц, наследник потомок класса Table

*Новые методы:*

int Hashfunc(const KeyType &k) const; – хэш функция

*Новые поля:*

int \*DM; – массив маркеров – хранит статус ячейки (удалена, занята, свободна)

# Заключение

В лабораторной работе была реализована программа, выполняющая вставку, поиск, удаления элементов, заданных пользователем. Реализован алгоритм, позволяющий разрешить проблемы при возникновении коллизий в хэш таблицах. Алгоритмы обеспечивают уникальность ключа во всех видах таблиц.

# Список литературы

1. *Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К*. Алгоритмы: построение и анализ, Москва 2013.
2. Рабочие материалы к учебному курсу «Методы программирования». *Гергель В.П*. 2002 г.

# Приложение

## Приложение A. Tables.h.

#pragma once

#include "Tables.h"

#include <string>

#include <cstdlib> // для рандомма

using namespace std;

#define Mark0 0

#define Mark1 1

#define Markminus1 -1

//0 - свободен; 1 - место занято; -1 - был удалён;

//дубликаты ключа при вставке;

template <class KeyType, class DataType>

class HashTable : public Table <KeyType, DataType>

{

public:

HashTable(int SIZE = 20) : Table<KeyType, DataType>(SIZE)

{

DM = new int[SIZE];

for (int i = 0; i < SIZE; i++)

DM[i] = Mark0;

};

~HashTable() {};

void Insert(const KeyType &KT\_T, const DataType &DT\_D); // вставка в неупорядоченный таблице

void Dell(const KeyType &KT\_T); // удаление элемента

DataTable<KeyType, DataType>\* Search(const KeyType &KT\_T) const; // поиск элемена по ключу

private:

void Reallocate(); //перераспределение, в данном случаи - перехеширование

int Hashfunc(const KeyType &k) const;

int \*DM; // DM like DataMarker

};

template<class KeyType>

int GetDigKey(const KeyType& k) { return 0; }

template<>

int GetDigKey<string>(const string &k) // сама реализация хэш функции от string

{

int g = 31; //13

int hash = 0;

for (int i = 0; i < k.length(); i++)

hash = g \* hash + k[i];

return hash;

}

template <class KeyType, class DataType> // хэш функция внутри класса (метод)

int HashTable<KeyType, DataType>:: Hashfunc(const KeyType& k) const

{

int temp = GetDigKey(k);

return (temp%(this->MaxSize));

}

template <class KeyType, class DataType>

void HashTable<KeyType, DataType>::Reallocate()

{

int i = 0;

int NextSize = (int)(this->MaxSize + 20)\*1.65;

DataTable<KeyType, DataType> \*\* NewDT = new DataTable<KeyType, DataType> \* [NextSize];

int \* DM1 = new int[NextSize];

for (int i = 0; i < this->MaxSize; i++)

DM1[i] = DM[i];

for (int i = this-> MaxSize; i < NextSize; i++)

DM1[i] = Mark0;

delete[]DM;

DM = DM1;

//

for (int i = 0; i<this->MaxSize; i++)

NewDT[i] = this->DT[i];

for (int i = this->MaxSize; i < NextSize; i++)

NewDT[i] = NULL;

delete[] this->DT;

this->DT = NewDT;

this->MaxSize = NextSize;

}

template <class KeyType, class DataType>

void HashTable<KeyType, DataType>::Insert(const KeyType &KT\_T, const DataType &DT\_D)

{

if ((double)this->curindex / (double)this->MaxSize > 0.7) // перераспределить если до заполнения осталось 30 процентов

Reallocate();

int place = Hashfunc(KT\_T);

if (DM[place] == Mark0) //ячейка свободна

{

this->DT[place] = new DataTable<KeyType, DataType>(KT\_T, DT\_D); DM[place] = Mark1; this->curindex++;

}

else if (DM[place] == Markminus1)

{

this->DT[place] = new DataTable<KeyType, DataType>(KT\_T, DT\_D); DM[place] = Mark1;

}

else

{

if ((\*(this->DT[place])).Key == KT\_T)

{

throw "duplicated key";

}

srand(place);

int laker = rand() % (this->MaxSize);

while (DM[laker] == Mark1) //пока элемент занят

{

laker = rand() % (this->MaxSize);

if ((this->DT[laker]!=NULL)&&((\*(this->DT[laker])).Key == KT\_T))

{

throw "duplicated key";

}

}

this->DT[laker] = new DataTable<KeyType, DataType>(KT\_T, DT\_D);

if (DM[laker] == 0)

this->curindex++;

DM[laker] = Mark1;

}

}

template <class KeyType, class DataType>

void HashTable<KeyType, DataType>::Dell(const KeyType &KT\_T)

{

int place = Hashfunc(KT\_T);

if (DM[place] == 0) //элемента нет

throw "element doesn't exist";

if ((this->DT[place] != NULL) && ((\*(this->DT[place])).Key == KT\_T))

{

delete this->DT[place];

this->DT[place] = NULL;

DM[place] = Markminus1;

}

else

{

srand(place);

while ((DM[place] == -1) || ((DM[place] == 1) && ((\*(this->DT[place])).Key != KT\_T))) //если элемент был удалён, или он занят, но не совпадают ключи - обновить место

place = rand() % (this->MaxSize);

if (DM[place] == 1) // совпадают ключи

{

delete this->DT[place];

this->DT[place] = NULL;

DM[place] = Markminus1;

}

else throw "element doesn't exist";

}

}

template <class KeyType, class DataType>

DataTable<KeyType, DataType>\* HashTable<KeyType, DataType>::Search(const KeyType &KT\_T) const

{

int place = Hashfunc(KT\_T);

if (DM[place] == 0) //элемента нет

throw "element doesn't exist";

if ((this->DT[place] != NULL) && ((\*(this->DT[place])).Key)==KT\_T)

{

return this->DT[place];

}

else

{

srand(place);

while ((DM[place] == -1) || ((DM[place] == 1) && ((\*(this->DT[place])).Key != KT\_T))) //если элемент был удалён, или он занят, но не совпадают ключи - обновить место

place = rand() % (this->MaxSize);

if (DM[place] == 1) // совпадают ключи

{

return this->DT[place];

}

else

throw "element doesn't exist";

}

}

## Приложение B. TablesUnord.h

#pragma once

#include "Tables.h"

#include "math.h"

#include <string>

using namespace std;

template <class KeyType, class DataType>

class TableUnord : public Table <KeyType, DataType>

{

public:

TableUnord(int SIZE = MIN\_SIZE) : Table<KeyType, DataType>(SIZE) {};

virtual ~TableUnord() {};

virtual void Insert(const KeyType &KT\_T, const DataType &DT\_D); // вставка в неупорядоченный таблице

virtual void Dell(const KeyType &KT\_T); // удаление элемента

virtual DataTable<KeyType, DataType>\* Search(const KeyType &KT\_T) const ; // поиск элемена по ключу

//вывод

private:

virtual void Reallocate(); //перераспределение

};

template <class KeyType, class DataType>

void TableUnord <KeyType, DataType> :: Reallocate()

{

int NextSize = (int)(this->MaxSize + 20)\*1.65;

DataTable<KeyType, DataType> \*\* NewDT = new DataTable<KeyType, DataType>\*[NextSize];

for (int i=0; i<this->MaxSize; i++)

NewDT[i] = this->DT[i];

for (int i = this->MaxSize; i<NextSize; i++)

NewDT[i] = NULL;

delete[] this->DT;

this->DT = NewDT;

this->MaxSize = NextSize;

};

template <class KeyType, class DataType>

void TableUnord< KeyType, DataType> :: Insert(const KeyType &KT\_T, const DataType &DT\_D)

{

if ((double)this->Top/(double)this->MaxSize > 0.7) // перераспределить если до заполнения осталось менее 30 процентов

Reallocate();

for (int i = 0; i < this->Top; i++)

{

if ((\*(this->DT[i])).Key == KT\_T)

{

throw "duplicated key";

}

}

this->DT[this->Top++] = new DataTable<KeyType, DataType>(KT\_T, DT\_D);

};

template <class KeyType, class DataType> //прочекать на правильность

void TableUnord< KeyType, DataType> ::Dell(const KeyType &KT\_T)

{

int i = 0;

int flag = 0;

while ((i < this->Top) && (flag ==0)) {

if ((\*(this->DT[i])).Key == KT\_T) flag = 1;

i++;

}

if (flag == 1) {

i--;

delete this->DT[i];

while (i < this->Top-1) {

this->DT[i] = this->DT[i + 1];

i++;

}

this->DT[i] = NULL;

this->Top--;

}

else throw "element doesn't exist";

};

template <class KeyType, class DataType>

DataTable<KeyType, DataType>\* TableUnord< KeyType, DataType> ::Search(const KeyType &KT\_T) const

{

int i = 0;

while (i < this->Top)

{

if ((\*(this->DT[i])).Key == KT\_T)

return this->DT[i];

i++;

}

throw "element doesn't exist";};

## Приложение C. TablesOrdered.h

#pragma once

#include "Tables.h"

template <class KeyType, class DataType>

class TableOrdered: public Table<KeyType, DataType>

{

public:

TableOrdered(int SIZE = MIN\_SIZE) : Table<KeyType, DataType>(SIZE) {};

virtual ~TableOrdered() {};

virtual void Insert(const KeyType &KT\_T, const DataType &DT\_D); // вставка в упорядоченную таблицу таблице

virtual void Dell(const KeyType &KT\_T); // удаление элемента

virtual DataTable<KeyType, DataType>\* Search(const KeyType &KT\_T) const; // поиск элемена по ключу

private:

virtual void Reallocate(); //перераспределение

int BinarSearch(const KeyType &Key) const; //бинарный поиск

};

template <class KeyType, class DataType>

void TableOrdered <KeyType, DataType> ::Reallocate()

{

int NextSize = (int)(this->MaxSize + 20)\*1.65;

DataTable <KeyType, DataType> \*\* NewDT = new DataTable <KeyType, DataType>\*[NextSize];

for (int i = 0; i<this->MaxSize; i++)

NewDT[i] = this->DT[i];

for (int i = this->MaxSize; i<NextSize; i++)

NewDT[i] = NULL;

delete[] this->DT;

this->DT = NewDT;

this->MaxSize = NextSize;

}

template <class KeyType, class DataType>

int TableOrdered <KeyType, DataType> ::BinarSearch(const KeyType &Key) const //возвращает индекс (номер строки в таблице)

{

int tail = this->Top-1;

int mid = 0;

int start = 0;

while (start <= tail)

{

mid = (tail + start) / 2;

if ((\*(this->DT[mid])).Key < Key)

start = mid + 1;

else if ((\*(this->DT[mid])).Key > Key)

tail = mid - 1;

else return mid; //new disicion

}

if (start > tail)

mid = start;

return mid;

}

template <class KeyType, class DataType>

void TableOrdered <KeyType, DataType> ::Insert(const KeyType &KT\_T, const DataType &DT\_D)

{

int area = BinarSearch(KT\_T);

int j;

if ((this->DT[area]==NULL)||((\*(this->DT[area])).Key != KT\_T))

{

if ((double)this->Top/(double)this->MaxSize > 0.7) // перераспределить если до заполнения осталось 30 процентов

Reallocate();

for (j = this->Top; j > area; j--)

this->DT[j] = this->DT[j - 1];

this->DT[j] = new DataTable<KeyType, DataType>(KT\_T, DT\_D);

this->Top++;

}

else throw "duplicate key";

}

template <class KeyType, class DataType>

void TableOrdered <KeyType, DataType> ::Dell(const KeyType &KT\_T)

{

int area = BinarSearch(KT\_T);

if ((this->DT[area] != NULL) && ((\*(this->DT[area])).Key == KT\_T))

{

delete this->DT[area];

for (int i = area; i < this->Top; i++)

this->DT[i] = this->DT[i + 1];

this->Top -= 1;

}

else

throw "element doesn't exist";

}

template <class KeyType, class DataType>

DataTable<KeyType, DataType>\* TableOrdered <KeyType, DataType> ::Search(const KeyType &KT\_T) const

{

int area = BinarSearch(KT\_T);

if ((\*(this->DT[area])).Key == KT\_T)

return this->DT[area];

else throw "element doesn't exist";

}

## Приложение D. HashTables.h

#pragma once

#include "Tables.h"

#include <string>

#include <cstdlib> // для рандомма

using namespace std;

#define Mark0 0

#define Mark1 1

#define Markminus1 -1

//0 - свободен; 1 - место занято; -1 - был удалён;

//дубликаты ключа при вставке;

template <class KeyType, class DataType>

class HashTable : public Table <KeyType, DataType>

{

public:

HashTable(int SIZE = 20) : Table<KeyType, DataType>(SIZE)

{

DM = new int[SIZE];

for (int i = 0; i < SIZE; i++)

DM[i] = Mark0;

};

~HashTable() {};

void Insert(const KeyType &KT\_T, const DataType &DT\_D); // вставка в неупорядоченный таблице

void Dell(const KeyType &KT\_T); // удаление элемента

DataTable<KeyType, DataType>\* Search(const KeyType &KT\_T) const; // поиск элемена по ключу

private:

void Reallocate(); //перераспределение, в данном случаи - перехеширование

int Hashfunc(const KeyType &k) const;

int \*DM; // DM like DataMarker

};

template<class KeyType>

int GetDigKey(const KeyType& k) { return 0; }

template<>

int GetDigKey<string>(const string &k) // сама реализация хэш функции от string

{

int g = 31; //13

int hash = 0;

for (int i = 0; i < k.length(); i++)

hash = g \* hash + k[i];

return hash;

}

template <class KeyType, class DataType> // хэш функция внутри класса (метод)

int HashTable<KeyType, DataType>:: Hashfunc(const KeyType& k) const

{

int temp = GetDigKey(k);

return (temp%(this->MaxSize));

}

template <class KeyType, class DataType>

void HashTable<KeyType, DataType>::Reallocate()

{

int i = 0;

int NextSize = (int)(this->MaxSize + 20)\*1.65;

DataTable<KeyType, DataType> \*\* NewDT = new DataTable<KeyType, DataType> \* [NextSize];

int \* DM1 = new int[NextSize];

for (int i = 0; i < this->MaxSize; i++)

DM1[i] = DM[i];

for (int i = this-> MaxSize; i < NextSize; i++)

DM1[i] = Mark0;

delete[]DM;

DM = DM1;

//

for (int i = 0; i<this->MaxSize; i++)

NewDT[i] = this->DT[i];

for (int i = this->MaxSize; i < NextSize; i++)

NewDT[i] = NULL;

delete[] this->DT;

this->DT = NewDT;

this->MaxSize = NextSize;

}

template <class KeyType, class DataType>

void HashTable<KeyType, DataType>::Insert(const KeyType &KT\_T, const DataType &DT\_D)

{

if ((double)this->curindex / (double)this->MaxSize > 0.7) // перераспределить если до заполнения осталось 30 процентов

Reallocate();

int place = Hashfunc(KT\_T);

if (DM[place] == Mark0) //ячейка свободна

{

this->DT[place] = new DataTable<KeyType, DataType>(KT\_T, DT\_D); DM[place] = Mark1; this->curindex++;

}

else if (DM[place] == Markminus1)

{

this->DT[place] = new DataTable<KeyType, DataType>(KT\_T, DT\_D); DM[place] = Mark1;

}

else

{

if ((\*(this->DT[place])).Key == KT\_T)

{

throw "duplicated key";

}

srand(place);

int laker = rand() % (this->MaxSize);

while (DM[laker] == Mark1) //пока элемент занят

{

laker = rand() % (this->MaxSize);

if ((this->DT[laker]!=NULL)&&((\*(this->DT[laker])).Key == KT\_T))

{

throw "duplicated key";

}

}

this->DT[laker] = new DataTable<KeyType, DataType>(KT\_T, DT\_D);

if (DM[laker] == 0)

this->curindex++;

DM[laker] = Mark1;

}

}

template <class KeyType, class DataType>

void HashTable<KeyType, DataType>::Dell(const KeyType &KT\_T)

{

int place = Hashfunc(KT\_T);

if (DM[place] == 0) //элемента нет

throw "element doesn't exist";

if ((this->DT[place] != NULL) && ((\*(this->DT[place])).Key == KT\_T))

{

delete this->DT[place];

this->DT[place] = NULL;

DM[place] = Markminus1;

}

else

{

srand(place);

while ((DM[place] == -1) || ((DM[place] == 1) && ((\*(this->DT[place])).Key != KT\_T))) //если элемент был удалён, или он занят, но не совпадают ключи - обновить место

place = rand() % (this->MaxSize);

if (DM[place] == 1) // совпадают ключи

{

delete this->DT[place];

this->DT[place] = NULL;

DM[place] = Markminus1;

}

else throw "element doesn't exist";

}

}

template <class KeyType, class DataType>

DataTable<KeyType, DataType>\* HashTable<KeyType, DataType>::Search(const KeyType &KT\_T) const

{

int place = Hashfunc(KT\_T);

if (DM[place] == 0) //элемента нет

throw "element doesn't exist";

if ((this->DT[place] != NULL) && ((\*(this->DT[place])).Key)==KT\_T)

{

return this->DT[place];

}

else

{

srand(place);

while ((DM[place] == -1) || ((DM[place] == 1) && ((\*(this->DT[place])).Key != KT\_T))) //если элемент был удалён, или он занят, но не совпадают ключи - обновить место

place = rand() % (this->MaxSize);

if (DM[place] == 1) // совпадают ключи

{

return this->DT[place];

}

else

throw "element doesn't exist";

}

}

## Приложение E. main.cpp

#include "monom.h"

#include "polinom.h"

#include <iostream>

#include <ostream>

#include "TablesUnord.h"

#include "TablesOrdered.h"

#include "TablesHash.h"

using namespace std;

int main()

{

int c = 0;

int k;

int SIZE;

cout << "Create table, please enter size of table" << endl;

cin >> SIZE;

TableUnord<string, Polinom> A(SIZE);

string a;

TableOrdered<string, Polinom> B(SIZE);

HashTable<string, Polinom> C(SIZE);

while (c != 1)

{

system("cls");

cout << "Chouse operation" << endl;

cout << "1 - Insert" << endl;

cout << "2 - Dell" << endl;

cout << "3 - Search" << endl;

cout << "4 - Print" << endl;

cin >> k;

switch (k)

{

case 1:

{

string str;

cout << "Write your Polinom" << endl;

cin >> str;

Polinom a(str);

try {A.Insert(str, a);}

catch (...) { cout << "duplicated key"<<endl; }

try {B.Insert(str, a); }

catch (...) { cout << "duplicated key" << endl; }

try { C.Insert(str, a); }

catch (...) { cout << "duplicated key" << endl; }

cout << "Unordered :" << endl;

cout << A << endl;

cout << "Ordered :" << endl;

cout << B << endl;

cout << "Hash :" << endl;

cout << C << endl;

break;

}

case 2:

{

string str;

cout << "Write your Polinom" << endl;

cin >> str;

Polinom a(str);

try { A.Dell(str);}

catch (...) { cout << "element doesn't exist" << endl; }

try { B.Dell(str); }

catch (...) { cout << "element doesn't exist" << endl; }

try { C.Dell(str); }

catch (...) { cout << "element doesn't exist" << endl; }

cout << "Unordered :" << endl;

cout << A << endl;

cout << "Ordered :" << endl;

cout << B << endl;

cout << "Hash :" << endl;

cout << C << endl;

break;

}

case 3:

{

string str;

cout << "Write your Polinom" << endl;

cin >> str;

Polinom a(str);

try { cout<<"finded: "<<A.Search(str)->Data; }

catch (...) { cout << "element doesn't exist" << endl; }

try { cout << "finded: " << B.Search(str)->Data; }

catch (...) { cout << "element doesn't exist" << endl; }

try { cout << "finded: " << C.Search(str)->Data; }

catch (...) { cout << "element doesn't exist" << endl; }

break;

}

case 4:

{

cout << "Unordered :" << endl;

cout << A << endl;

cout << "Ordered :" << endl;

cout << B << endl;

cout << "Hash :" << endl;

cout << C << endl;

break;

}

default:

{

cout << "ERROR, choose again" << endl;

break;

}

}

cout << "Exit - 1" << endl;

cin >> c;

}

return 0;

}

## Приложение F. Test\_all.cpp Тесты программы

TEST(TableUnord, can\_create) // вставка в конец

{

TableUnord<string, Polinom> A(20);

Polinom a("1+x");

ASSERT\_NO\_THROW(A.Insert("1+x", a));

}

class TableforCheck : public ::testing::Test

{

public:

HashTable <string, Polinom> HT;

TableOrdered <string, Polinom> TO;

TableUnord <string, Polinom> TU;

TableforCheck() : HT(10), TO(10), TU(10)

{

HT.Insert("1+x", Polinom("1+x"));

TO.Insert("1+x", Polinom("1+x"));

TU.Insert("1+x", Polinom("1+x"));

HT.Insert("10+xyz", Polinom("10+xyz"));

TO.Insert("10+xyz", Polinom("10+xyz"));

TU.Insert("10+xyz", Polinom("10+xyz"));

HT.Insert("x^4", Polinom("x^4"));

TO.Insert("x^4", Polinom("x^4"));

TU.Insert("x^4", Polinom("x^4"));

};

};

TEST\_F(TableforCheck, can\_get\_curr\_data\_TU)

{

TU.Reset();

Polinom p("1+x");

EXPECT\_EQ(TU.GetCurData(), p);

}

TEST\_F(TableforCheck, can\_get\_next\_TO)

{

ASSERT\_NO\_THROW(TO.GetNext());

}

TEST\_F(TableforCheck, can\_get\_next\_HT)

{

ASSERT\_NO\_THROW(HT.GetNext());

}

TEST\_F(TableforCheck, can\_get\_next\_TU)

{

ASSERT\_NO\_THROW(TU.GetNext());

}

TEST\_F(TableforCheck, can\_find\_existed\_element\_HT)

{

EXPECT\_EQ(HT.Search("1+x")->Data, Polinom("1+x"));

}

TEST\_F(TableforCheck, can\_find\_existed\_element\_TU)

{

EXPECT\_EQ(TU.Search("1+x")->Data, Polinom("1+x"));

}

TEST\_F(TableforCheck, can\_find\_existed\_element\_TO)

{

EXPECT\_EQ(TO.Search("1+x")->Data, Polinom("1+x"));

}

TEST\_F(TableforCheck, can\_dell\_existed\_element\_HT)

{

HT.Dell("1+x");

ASSERT\_ANY\_THROW(HT.Search("1+x"));

}

TEST\_F(TableforCheck, can\_dell\_existed\_element\_TU)

{

TU.Dell("1+x");

ASSERT\_ANY\_THROW(TU.Search("1+x"));

}

TEST\_F(TableforCheck, can\_dell\_existed\_element\_TO)

{

TO.Dell("1+x");

ASSERT\_ANY\_THROW(TO.Search("1+x"));

}

TEST\_F(TableforCheck, can\_insert\_element\_HT)

{

HT.Insert("1+2x+y", Polinom("1+2x+y"));

EXPECT\_EQ(HT.Search("1+2x+y")->Data, Polinom("1+2x+y"));

}

TEST\_F(TableforCheck, can\_insert\_element\_TU)

{

TU.Insert("1+z", Polinom("1+z"));

EXPECT\_EQ(TU.Search("1+z")->Data, Polinom("1+z"));

}

TEST\_F(TableforCheck, can\_insert\_element\_TO)

{

TO.Insert("xyz+z^5", Polinom("xyz+z^5"));

EXPECT\_EQ(TO.Search("xyz+z^5")->Data, Polinom("xyz+z^5"));

}

TEST\_F(TableforCheck, can\_find\_existed\_element2\_HT)

{

EXPECT\_EQ(HT.Search("10+xyz")->Data, Polinom("10+xyz"));

}

TEST\_F(TableforCheck, can\_find\_existed\_element2\_TU)

{

EXPECT\_EQ(TU.Search("10+xyz")->Data, Polinom("10+xyz"));

}

TEST\_F(TableforCheck, can\_find\_existed\_element2\_TO)

{

EXPECT\_EQ(TO.Search("10+xyz")->Data, Polinom("10+xyz"));

}

TEST\_F(TableforCheck, can\_find\_existed\_element3\_HT)

{

EXPECT\_EQ(HT.Search("x^4")->Data, Polinom("x^4"));

}

TEST\_F(TableforCheck, can\_find\_existed\_element3\_TU)

{

EXPECT\_EQ(TU.Search("x^4")->Data, Polinom("x^4"));

}

TEST\_F(TableforCheck, can\_find\_existed\_element3\_TO)

{

EXPECT\_EQ(TO.Search("x^4")->Data, Polinom("x^4"));

}

TEST\_F(TableforCheck, can\_insert\_element2\_HT)

{

HT.Insert("111+z", Polinom("111+z"));

EXPECT\_EQ(HT.Search("111+z")->Data, Polinom("111+z"));

}

TEST\_F(TableforCheck, can\_insert\_element2\_TU)

{

TU.Insert("1+32.1x", Polinom("1+32.1x"));

EXPECT\_EQ(TU.Search("1+32.1x")->Data, Polinom("1+32.1x"));

}

TEST\_F(TableforCheck, can\_insert\_element2\_TO)

{

TO.Insert("xyz", Polinom("xyz"));

EXPECT\_EQ(TO.Search("xyz")->Data, Polinom("xyz"));

}

TEST\_F(TableforCheck, cannot\_dell\_element\_HT)

{

ASSERT\_ANY\_THROW(HT.Dell("12"));

}

TEST\_F(TableforCheck, cannot\_dell\_element\_TU)

{

ASSERT\_ANY\_THROW(TU.Dell("12"));

}

TEST\_F(TableforCheck, cannot\_dell\_element\_TO)

{

ASSERT\_ANY\_THROW(TO.Dell("12"));

}

TEST\_F(TableforCheck, cannot\_Inset\_ex\_element\_HT)

{

ASSERT\_ANY\_THROW(HT.Insert("1+x", Polinom("1+x")));

}

TEST\_F(TableforCheck, cannot\_Inset\_ex\_element\_TU)

{

ASSERT\_ANY\_THROW(TU.Insert("1+x", Polinom("1+x")));

}

TEST\_F(TableforCheck, cannot\_Inset\_ex\_element\_TO)

{

ASSERT\_ANY\_THROW(TO.Insert("1+x", Polinom("1+x")));

}

TEST\_F(TableforCheck, cannot\_Inset\_ex\_element2\_HT)

{

ASSERT\_ANY\_THROW(HT.Insert("x^4", Polinom("1+x")));

}

TEST\_F(TableforCheck, cannot\_Inset\_ex\_element2\_TU)

{

ASSERT\_ANY\_THROW(TU.Insert("x^4", Polinom("1+x")));

}

TEST\_F(TableforCheck, cannot\_Inset\_ex\_element2\_TO)

{

ASSERT\_ANY\_THROW(TO.Insert("x^4", Polinom("1+x")));

}

TEST\_F(TableforCheck, jast\_sotka\_test\_e\_booy)

{

ASSERT\_ANY\_THROW(TO.Dell("100"));

}