Министерство образования и науки Российской Федерации

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

Институт Информационных технологий, математики и механики

Отчёт по лабораторной работе

Таблицы

Выполнил:

студент гр. 381603-01

Кумин А.А

Проверил:

ассистент каф. МОСТ, ИИТММ

Кустикова В.Д.

Нижний Новгород

2018 г

**Содержание**

[Введение 3](#_Toc515626195)

[1 Постановка задачи 4](#_Toc515626196)

[2 Руководство пользователя 5](#_Toc515626197)

[3 Руководство программиста 6](#_Toc515626198)

[3.1 Структура программы 6](#_Toc515626199)

[3.2 Структуры данных 7](#_Toc515626200)

[3.2.1 Шаблонный класс TabRecord 7](#_Toc515626201)

[3.2.2 Шаблонный виртуальный класс Table 7](#_Toc515626202)

[3.2.3 Дочерний класс ScanTable 9](#_Toc515626203)

[3.2.4 Дочерний класс SortedTable 9](#_Toc515626204)

[3.2.5 Дочерний класс HashTable 9](#_Toc515626205)

[3.3 Описание алгоритмов 10](#_Toc515626206)

[3.3.1 Несортируемые таблицы 10](#_Toc515626207)

[3.3.2 Сортируемые таблицы 11](#_Toc515626208)

[3.3.3 Хэш-таблицы 12](#_Toc515626209)

[Заключение 14](#_Toc515626210)

[Литература 15](#_Toc515626211)

[Приложение 16](#_Toc515626212)

[Приложение 1. Программная реализация шаблонного класса TabRecord и виртуального класса Table 16](#_Toc515626213)

[Приложение 2. Программная реализация дочернего класса ScanTable 18](#_Toc515626214)

[Приложение 3. Программная реализация дочернего класса SortedTable 19](#_Toc515626215)

[Приложение 4. Программная реализация дочернего класса HashTable 21](#_Toc515626216)

[Приложение 5. Тесты для класса ScanTable 24](#_Toc515626217)

[Приложение 7. Тесты для класса SortedTable 26](#_Toc515626218)

[3.4 Приложение 8. Тесты для класса HashTable 29](#_Toc515626219)

# Введение

Таблица – способ структурирования данных. Представляет собой распределение данных по однотипным строкам и столбцам. Таблица – это удобный способ связать данные, представить их в понятном виде. Они встречаются в нашей жизни на каждом шагу, большие объемы важной информации представлены в виде таблиц, поэтому так важно уметь с ними работать и исследовать их.

Хэш-таблица – это специальная структура данных для хранения пар ключей и их значений, вычисляемых из этих ключей с помощью хэш – функции. Примечательность этих таблиц в том, что среднее время поиска, вставки и удаления меньше, чем в других способах реализации таблиц, например, сортируемых (бинарный поиск) и несортируемых.

В отчёте приводится постановка задачи, описание использующихся алгоритмов, описание программы и правила её использования, а также прилагается код программы, решающей поставленную задачу.

# Постановка задачи

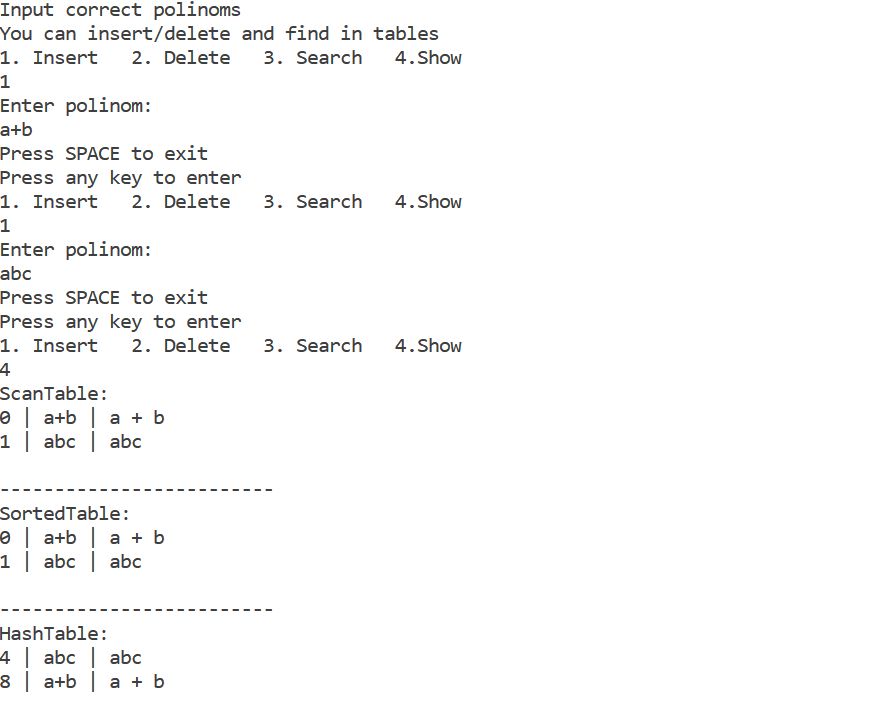
Реализовать виртуальный шаблонный класс таблиц с поддержкой 3х операций: поиск, вставка, удаление. На основе этого класса реализовать 3 дочерних класса: несортируемые таблицы, сортируемые и хэш – таблицы, с поддержкой 3х изначальных операций. Написать основную программу, предоставляющую пользователю доступ к заполнению многочленами, удалению и поиску многочленов в таблицах, а также способную выводить на экран данные этих 3х видов таблиц.

Исходные данные: 3 таблицы. Вставка многочлена, либо удаление многочлена, либо поиск многочлена.

Выходные данные: Результат вставки, удаления, поиска (вывод таблиц).

# Руководство пользователя

1. Открыть файл sample\_tables.exe.
2. Программа предложит 4 варианта: 1.Вставка, 2.Удаление, 3.Поиск, 4.Вывод.  
   Выбрать нужный вариант.
3. При выборе 1, 2 или 3 варианта программа предложит ввести полином. Ввести полином.
4. После выполнения операции программа предложит вам продолжить или выйти из нее.



1. Демонстрация работы программы

# Руководство программиста

## Структура программы

1. Программа содержит статическую библиотеку класса Polinom представленную заголовочными файлами и файлами реализации, а также тестами, описанными в отчете «Операции над полиномами».
2. table.h – содержит объявление и реализацию шаблонного виртуального класса таблиц Table, в котором объявлены методы поиска, вставки и удаления строчек таблиц, а так же перегружен вывод таблиц на экран.

Также содержит структуру шаблонного класса строчек таблицы TabRecord.

1. scan\_table.h – содержит объявление и реализацию шаблонного дочернего класса несортируемых таблиц ScanTable, в котором реализованы операции вставки, удаления и поиска.
2. sorted\_table.h – содержит объявление и реализацию шаблонного дочернего класса сортируемых таблиц SortedTable, в котором реализованы операции вставки, удаления и поиска (с помощью бинарного поиска).
3. hash\_table.h– содержит объявление и реализацию шаблонного дочернего класса сортируемых таблиц HashTable, в котором реализованы операции вставки, удаления и поиска (с помощью хэш - функции), а так же перегрузка вывода на экран.
4. main.cpp – содержит реализацию пользовательского приложения, для работы с таблицами.
5. test\_scan\_table.cpp – содержит реализацию тестов для класса ScanTable
6. test\_sorted\_table.cpp – содержит реализацию тестов для класса SortedTable.
7. test\_hash\_table.cpp – содержит реализацию тестов для класса HashTable.

## Структуры данных

### Шаблонный класс TabRecord

Поля(public):

**string key** – поле ключа записи(строки).

**type data** – поле данных записи(строки).

Методы(public):

**TabRecord()**– конструктор записи таблицы по умолчанию.

**TabRecord(const string& KEY, const type& DATA)** – конструктор записи таблицы.

**TabRecord(const TabRecord& T)**– конструктор копирования.

**TabRecord& operator=(const TabRecord& T)**– перегрузка оператора =.

### Шаблонный виртуальный класс Table

Поля(protected):

**TabRecord<type>\*\* Records** – массив указателей на записи(строчки) таблицы.

**int MaxRecords** – максимально количество записей в таблице (размер таблицы).

**int CurrRecord** – текущее кол-во записей в таблице.

**int CurrIndex** – номер текущей рассматриваемой записи таблицы.

Методы(protected):

**virtual void Realloc()** – виртуальная функция перераспределения памяти.

Методы(public):

**Table(int i = 10)** – конструктор таблицы.

**Table(const Table<type>& CopyTab)** – конструктор копирования.

**virtual ~Table()** –деструктор.

**virtual type& Search(const string& KEY) const = 0** – функция поиска записи в таблице по ключу, принимает на вход ключ, выводит ссылку на данные, не реализуется в данном классе.

**virtual void Insert(const string& KEY, const type& DATA) = 0** – функция вставки данных в таблицу, принимает на вход ключ и данные, не реализуется в данном классе.

**virtual void Delete(const string& KEY) = 0;**– функция удаления данных из таблицы по ключу, принимает на вход ключ, не реализуется в данном классе.

**virtual void reset()** – выставляет текущую запись таблицы(CurrIndex) на начало

**virtual int isended() const** - проверяет значение текущей записи, если он на последней строке таблицы или таблица пуста то выведет константу типа int, иначе 0.

**virtual type& GetCurr() const** – выводит данные текущей записи таблицы.

**virtual void SetNext()** – выставляет текущую запись таблицы на следующую.

**template<class type> friend ostream& operator<< (std::ostream& os, const Table<type>& Tab)** – перегрузка вывода таблицы на экран.

### Дочерний класс ScanTable

Поля(private):

Все поля унаследованы от класса Table.

Методы(public):

Все методы так же унаследованы от Table.

### Дочерний класс SortedTable

Поля(private):

Все поля унаследованы от класса Table.

Методы(protected):

**int BinarySearch(const string& KEY) const** – бинарный поиск по ключу, выводит номер элемента массива, в котором находится ключ и данные. Используется в поиске, вставке и удалении.

Методы(public):

Все методы так же унаследованы от Table.

### Дочерний класс HashTable

Поля:

Помимо унаследованных полей от класса Table содержит:

**int\* H** – массив, указывающий пустая ли запись или не пустая.

Методы(protected):

**int Hash(const string& KEY) const** – хэш-функция по ключу, выводит некоторое число, номер элемента массива – указателя на запись. Используется в поиске, вставке и удалении.

Методы(public):

Все методы так же унаследованы от Table.

## Описание алгоритмов

### Несортируемые таблицы

Таблицы, в которых записи располагаются в хаотичном порядке.

#### Вставка

-Сначала проверяется, полна ли таблица, если полна, то происходит перераспределение памяти с помощью функции Realloc.

-Находится номер элемента массива указателей, указывающий на последнюю запись таблицы.

-На следующее место вставляется новая строка.

-Если в таблице уже есть запись с подобным ключом или она пустая, то возникает исключение. Эта проверка реализуется обычным обходом массива указателей.

#### Удаление

-С помощью обычного обхода массива указателей находится нужная запись.

-Удаляется.

-На ее место вставляется запись с конца таблицы.

-Если в таблице нет записи с подобным ключом или она пустая, то возникает исключение.

#### Поиск

-С помощью обычного обхода массива указателей находится нужная запись.

-Выводятся ее данные.

-Если в таблице нет записи с подобным ключом или она пустая, то возникает исключение.

### Сортируемые таблицы

Таблицы, в которых записи располагаются в порядке возрастания ключа. Позволяют использовать быстрые методы поиска записей.

#### Бинарный поиск

-Определяется значение ключа в середине массива указателей.

-Ключ сравнивается с этим значением и если он больше, то поиск продолжается во второй половине элементов, если меньше – в первой.

-Поиск сводится к тому, что вновь определяется значение серединного ключа массива указателей и сравнивается и нужным ключом.

-Процесс продолжается до тех пор, пока не будет найден элемент с необходимым значением ключа или интервал для поиска не станет пустым.

#### Вставка

-Сначала проверяется, полна ли таблица, если полна, то происходит перераспределение памяти с помощью функции Realloc.

-С помощью бинарного поиска находятся элементы в таблице, между которыми можно вставить нужную запись, не нарушая упорядоченности записей таблицы.

-Перемещаем все элементы, ключ которых больше нужного ключа на одну запись вниз.

-На освобожденное место вставляется нужная запись.

-Если в таблице уже есть запись с подобным ключом или она пустая, то возникает исключение.

#### Удаление

-С помощью бинарного поиска по ключу находится нужная запись таблицы.

-Удаляется.

-Перемещаем все элементы, ключ которых больше нужного ключа на одну запись вверх.

-Если в таблице нет записи с подобным ключом или она пустая, то возникает исключение.

#### Поиск

-С помощью бинарного поиска по ключу находится нужная запись таблицы.

-Выводятся данные найденной записи.

-Если в таблице нет записи с подобным ключом или она пустая, то возникает исключение.

### Хэш-таблицы

Таблицы, в которых заполнение, удаление и поиск реализуются с помощью некоторого хэш-алгоритма.

#### Хэш-функция

Хэш-функция имеет следующие свойства:

-Всегда возвращает одно и то же значение для одного ключа.

-Не обязательно возвращает разные значения для разных ключей.

-Быстро вычисляет значение.

Когда для разных ключей хэш-функция вывод одинаковые значения возникает событие, называемое коллизией. Для борьбы с этим в рамках данной работы используется открытое перемешивание:

1)В случае вставки:

-Запись создается в следующей свободной ячейке таблицы.

2)В случае удаления и поиска:

-Запоминается значение хэш-функции от нужного ключа. Это номер элемента массива указателей.

-Если элемент массива по этому номеру указывает на запись с другим ключом, то последовательно идем по массиву указателей дальше и ищем элемент массива, указывающий на запись, ключ которой совпадает с нужным ключом.

-Если таких ключей нет, то вызывается исключение.

-Если запись нашлась, то удаляем ее и обозначаем пустой или выводим данные записи.

В данной работе хэш-функция реализована следующим образом:

-Каждый символ ключа переводится в десятичное число и прибавляется к некоторой переменной.

-Эта переменная и есть значение хэш-функции.

#### Вставка

-Сначала проверяется, полна ли таблица, если полна, то происходит перераспределение памяти с помощью функции Realloc.

-Запоминается значение хэш-функции от нужного ключа. Это номер элемента массива указателей.

-Если нет коллизии, то создаем нужную запись в этой ячейке таблицы и обозначаем ее непустой.

-Если в таблице уже есть запись с подобным ключом или она пустая, то возникает исключение.

#### Удаление

-Запоминается значение хэш-функции от нужного ключа. Это номер элемента массива указателей.

-Если нет коллизии, то удаляем запись и обозначаем ее пустой.

-Если таких записей нет или таблица пустая, то вызывается исключение.

#### Поиск

-Запоминается значение хэш-функции от нужного ключа. Это номер элемента массива указателей.

-Если нет коллизии, то выводим данные записи.

-Если таких записей нет или таблица пустая, то вызывается исключение.

# Заключение

В данной лабораторный работе был реализован виртуальный шаблонный класс таблиц, поддерживающий операции вставки, удаления и поиска, а также реализованы на основе этого класса 3 типа таблиц: хэш – таблицы, сортируемые и несортируемые. Реализована программа – интерфейс, предоставляющая пользователю возможность вводить, удалять, искать полиномы в данных таблицах.

# Литература

1. Лекции «Алгоритмы и структуры данных» Гергель В.П. ННГУ им. Лобачевского Н. И.

# Приложение

## Приложение 1. Программная реализация шаблонного класса TabRecord и виртуального класса Table

#pragma once

#include <string>

#include <iostream>

#define ReallocCoeff 2

using namespace std;

template <typename type>

class TabRecord

{

public:

string key;

type data;

TabRecord() { key = "empty"; }

TabRecord(const string& KEY, const type& DATA) { key = KEY; data = DATA; }

TabRecord(const TabRecord& T) { key = T.key; data = T.data; }

TabRecord& operator=(const TabRecord& T) { key = T.key; data = T.data; }

};

template <typename type>

class Table

{

protected:

TabRecord<type>\*\* Records;

int MaxRecords;

int CurrRecord;

int CurrIndex;

virtual void Realloc();

public:

Table(int i = 10);

Table(const Table<type>& CopyTab);

virtual ~Table() { delete[] Records; };

virtual type& Search(const string& KEY) const = 0;

virtual void Insert(const string& KEY, const type& DATA) = 0;

virtual void Delete(const string& KEY) = 0;

virtual void reset();

virtual int isended() const { return CurrIndex == -1 || ((CurrIndex + 1) == CurrRecord); }

virtual type& GetCurr() const;

virtual void SetNext();

template<class type> friend ostream& operator<< (std::ostream& os, const Table<type>& Tab);

};

//-----------------------------------------------------------

template<typename type>

void Table<type>::Realloc()

{

int NewMax = MaxRecords \* ReallocCoeff;

TabRecord<type>\*\* temp = new TabRecord<type>\*[NewMax];

reset();

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

temp[i] = Records[i];

delete[] Records;

Records = temp;

MaxRecords = NewMax;

};

template <typename type>

Table<type>::Table(int i = 10)

{

MaxRecords = i;

CurrIndex = -1;

CurrRecord = 0;

Records = new TabRecord<type>\*[MaxRecords];

}

template<typename type>

Table<type>::Table(const Table<type>& CopyTab)

{

MaxRecords = CopyTab.MaxRecords;

CurrRecord = CopyTab.CurrRecord;

CurrIndex = CopyTab.CurrIndex;

delete[] Records;

Records = new TabRecord<type>\*[MaxRecords];

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

Records[i] = CopyTab.Records[i];

}

template<typename type>

void Table<type>::reset()

{

if (CurrRecord)

CurrIndex = 0;

else

CurrIndex = -1;

}

template<typename type>

type& Table<type>::GetCurr() const

{

if (CurrRecord)

return Records[CurrIndex]->data;

else

throw "table is empty";

}

template<typename type>

void Table<type>::SetNext()

{

if (CurrRecord)

CurrIndex++;

else

throw "table is empty";

if (CurrIndex == CurrRecord)

reset();

}

template <typename type>

ostream& operator<< (ostream& os, const Table<type>& Tab)

{

if (Tab.CurrRecord)

for (int i = 0; i < Tab.CurrRecord; i++)

os << i << " | " << Tab.Records[i]->key << " | " << Tab.Records[i]->data << endl;

else

os << "Table is empty" << endl;

return os;

}

## Приложение 2. Программная реализация дочернего класса ScanTable

#pragma once

#include "table.h"

template<typename type>

class ScanTable :public Table<type>

{

public:

ScanTable(int i = 10) : Table(i) {};

ScanTable(const ScanTable<type>& CopyTab) : Table(CopyTab) {};

~ScanTable() { };

type& Search(const string& KEY) const override;

void Insert(const string& KEY, const type& DATA) override;

void Delete(const string& KEY) override;

};

//-----------------------------------------------------------

template<typename type>

type& ScanTable<type>::Search(const string& KEY) const

{

ScanTable<type> Temp(\*this);

Temp.reset();

if (Temp.CurrIndex > -1)

{

while (Temp.Records[Temp.CurrIndex]->key != KEY) && (Temp.CurrIndex < Temp.CurrRecord))

Temp.SetNext();

if (Temp.CurrIndex < Temp.CurrRecord)

return Temp.Records[Temp.CurrIndex]->data;

else

throw "key does not exist";

}

else

throw "Table is empty";

}

template<typename type>

void ScanTable<type>::Insert(const string& KEY, const type& DATA)

{

if (CurrRecord == MaxRecords)

Realloc();

reset();

if (CurrRecord)

{

while ((CurrIndex < CurrRecord) && (Records[CurrIndex]->key != KEY))

CurrIndex++;

if (CurrIndex == CurrRecord)

{

Records[CurrIndex] = new TabRecord<type>(KEY, DATA);

CurrRecord++;

}

else

throw "Such key already exists";

}

else

{

CurrIndex++;

Records[CurrIndex] = new TabRecord<type>(KEY, DATA);

CurrRecord++;

}

}

template<typename type>

void ScanTable<type>::Delete(const string& KEY)

{

reset();

if (CurrIndex > -1)

{

while ((CurrIndex < CurrRecord) && (Records[CurrIndex]->key != KEY))

CurrIndex++;

if (CurrRecord && (CurrIndex < CurrRecord))

{

if (CurrRecord > 1)

Records[CurrIndex] = Records[--CurrRecord];

else

CurrRecord = 0;

}

else

throw "Key does not exist";

}

else

throw "Table is empty";

}

## Приложение 3. Программная реализация дочернего класса SortedTable

#pragma once

#include "table.h"

template<typename type>

class SortedTable : public Table<type>

{

protected:

int BinarySearch(const string& KEY) const;

public:

SortedTable(int i = 10) : Table(i) {};

SortedTable(const SortedTable<type>& CopyTab) : Table(CopyTab) {};

~SortedTable() { };

type& Search(const string& KEY) const override;

void Insert(const string& KEY, const type& DATA) override;

void Delete(const string& KEY) override;

};

//-----------------------------------------------------------

template<typename type>

int SortedTable<type>::BinarySearch(const string& KEY) const

{

int i = 0, j = CurrRecord - 1;

int mid;

while (i <= j)

{

mid = (i + j) / 2;

if (KEY > Records[mid]->key)

i = mid + 1;

else

j = mid - 1;

}

return i;

}

template<typename type>

type& SortedTable<type>::Search(const string& KEY) const

{

SortedTable<type> Temp(\*this);

Temp.reset();

if (CurrIndex > -1)

{

int k = Temp.BinarySearch(KEY);

if (Temp.Records[k]->key == KEY)

return Temp.Records[k]->data;

else

throw "Key does not exist";

}

else

throw "Table is empty";

}

template<typename type>

void SortedTable<type>::Insert(const string& KEY, const type& DATA)

{

if (CurrRecord == MaxRecords)

Realloc();

reset();

if (CurrRecord)

{

int k = BinarySearch(KEY);

if (k != CurrRecord)

{

if (Records[k]->key != KEY)

{

for (int i = CurrRecord; i > k; i--)

Records[i] = Records[i - 1];

Records[k] = new TabRecord<type>(KEY, DATA);

CurrRecord++;

}

else

throw "Such key already exists";

}

else

{

Records[k] = new TabRecord<type>(KEY, DATA);

CurrRecord++;

}

}

else

{

CurrIndex++;

Records[CurrIndex] = new TabRecord<type>(KEY, DATA);

CurrRecord++;

}

}

template<typename type>

void SortedTable<type>::Delete(const string& KEY)

{

reset();

if (CurrRecord)

{

int k = BinarySearch(KEY);

if ((k < CurrRecord) && (Records[k]->key == KEY))

{

for (int i = k; i < CurrRecord - 1; i++)

Records[i] = Records[i + 1];

CurrRecord--;

}

else

throw "Key does not exist";

}

else

throw "Table is Empty";

}

## Приложение 4. Программная реализация дочернего класса HashTable

#pragma once

#include "table.h"

template<typename type>

class HashTable :public Table<type>

{

protected:

int\* H;

int Hash(const string& KEY) const;

void Realloc() override;

public:

HashTable(int i = 10);

HashTable(const HashTable& CopyTab);

~HashTable() { delete[] H; };

type& Search(const string& KEY) const override;

void Insert(const string& KEY, const type& DATA) override;

void Delete(const string& KEY) override;

void reset() override;

type& GetCurr() const override;

void SetNext() override;

template<class type> friend ostream& operator<< (std::ostream& os, const HashTable<type>& Tab);

};

//-----------------------------------------------------------

template<typename type>

HashTable<type>::HashTable(int i = 10) : Table(i)

{

H = new int[i];

for (int j = 0; j < MaxRecords; j++)

H[j] = 0;

};

template<typename type>

int HashTable<type>::Hash(const string& KEY) const

{

int hash = 0;

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

hash = hash + int(KEY[i]);

return hash % MaxRecords;

}

template<typename type>

void HashTable<type>::Realloc()

{

int NewMax = MaxRecords \* ReallocCoeff;

TabRecord<type>\*\* temp = new TabRecord<type>\*[NewMax];

int\* tempH = new int[NewMax];

int i = 0;

for (i; i < MaxRecords; i++)

{

temp[i] = Records[i];

tempH[i] = H[i];

}

delete[] Records;

delete[] H;

Records = temp;

H = tempH;

MaxRecords = NewMax;

}

template<typename type>

HashTable<type>::HashTable(const HashTable& CopyTab)

{

MaxRecords = CopyTab.MaxRecords;

CurrRecord = CopyTab.CurrRecord;

CurrIndex = CopyTab.CurrIndex;

delete[] Records;

delete[] H;

Records = new TabRecord<type>\*[MaxRecords];

H = new int[MaxRecords];

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

{

H[i] = CopyTab.H[i];

Records[i] = CopyTab.Records[i];

}

}

template<typename type>

type& HashTable<type>::Search(const string& KEY) const

{

HashTable<type> Temp(\*this);

Temp.reset();

if (Temp.CurrRecord)

{

Temp.CurrIndex = Temp.Hash(KEY);

int l = Temp.CurrIndex;

if (Temp.Records[Temp.CurrIndex]->key == KEY)

return Temp.Records[Temp.CurrIndex]->data;

else

{

while (Temp.H[Temp.CurrIndex] && ((Temp.CurrIndex + 1) != l) && (Temp.Records[Temp.CurrIndex]->key != KEY))

Temp.CurrIndex = (Temp.CurrIndex + 1) % Temp.MaxRecords;

if(Temp.Records[Temp.CurrIndex]->key == KEY)

return Temp.Records[Temp.CurrIndex]->data;

else

throw "Key does not exist";

}

}

else

throw "Table is empty";

}

template<typename type>

void HashTable<type>::Insert(const string& KEY, const type& DATA)

{

if (CurrRecord == MaxRecords)

Realloc();

CurrIndex = Hash(KEY);

if (!H[CurrIndex])

{

Records[CurrIndex] = new TabRecord<type>(KEY, DATA);

CurrRecord++;

H[CurrIndex] = 1;

}

else

if (Records[CurrIndex]->key != KEY)

{

int l = CurrIndex;

while (H[CurrIndex] && ((CurrIndex + 1) != l))

CurrIndex = (CurrIndex + 1) % MaxRecords;

Records[CurrIndex] = new TabRecord<type>(KEY, DATA);

CurrRecord++;

H[CurrIndex] = 1;

}

else

throw "this key exists";

}

template<typename type>

void HashTable<type>::Delete(const string& KEY)

{

reset();

if (CurrRecord)

{

CurrIndex = Hash(KEY);

int l = CurrIndex;

if (H[CurrIndex])

{

if (Records[CurrIndex]->key != KEY)

{

while (H[CurrIndex] && ((CurrIndex + 1) != l) && (Records[CurrIndex]->key != KEY))

CurrIndex = (CurrIndex + 1) % MaxRecords;

if (!H[CurrIndex] || Records[CurrIndex]->key != KEY)

throw "Key does not exist";

else

{

Records[CurrIndex] = new TabRecord<type>;

H[CurrIndex] = 0;

CurrRecord--;

}

}

else

{

Records[CurrIndex] = new TabRecord<type>;

H[CurrIndex] = 0;

CurrRecord--;

}

}

else

throw "Key does not exist";

}

else

throw "Table is empty";

}

template<typename type>

void HashTable<type>::reset()

{

if (CurrRecord)

{

CurrIndex = 0;

while (H[CurrIndex] != 1)

CurrIndex++;

}

else

CurrIndex = -1;

}

template<typename type>

type& HashTable<type>::GetCurr() const

{

if (CurrRecord)

return Records[CurrIndex]->data;

else

throw "table is empty";

}

template<typename type>

void HashTable<type>::SetNext()

{

if (CurrRecord)

{

CurrIndex++;

while (H[CurrIndex] != 1)

CurrIndex = (CurrIndex + 1) % MaxRecords;

}

else

throw "table is empty";

}

template <typename type>

ostream& operator<< (ostream& os, const HashTable<type>& Tab)

{

if (Tab.CurrRecord)

{

for (int i = 0; i < Tab.MaxRecords; i++)

if (Tab.H[i])

os << i << " | " << Tab.Records[i]->key << " | " << Tab.Records[i]->data << endl;

}

else

os << "Table is empty" << endl;

return os;

}

## Приложение 5. Тесты для класса ScanTable

#include "scan\_table.h"

#include <gtest.h>

TEST(ScanTable, can\_create\_ScanTable\_table)

{

ASSERT\_NO\_THROW(ScanTable<int> T);

}

//---------------------------------------------------------

class EmptyScanTable : public testing::Test

{

protected:

ScanTable<int> Tab;

public:

EmptyScanTable() {};

~EmptyScanTable() {};

};

TEST\_F(EmptyScanTable, empty\_ScanTable\_tab\_is\_empty)

{

EXPECT\_EQ(true, Tab.isended());

}

TEST\_F(EmptyScanTable, copy\_empty\_ScanTable\_tab\_and\_correct)

{

ASSERT\_NO\_THROW(ScanTable<int> Tab1(Tab));

ScanTable<int> Tab1(Tab);

EXPECT\_EQ(true, Tab1.isended());

}

TEST\_F(EmptyScanTable, cant\_getcurr\_empty)

{

ASSERT\_ANY\_THROW(Tab.GetCurr());

}

TEST\_F(EmptyScanTable, can\_reset\_empty)

{

ASSERT\_NO\_THROW(Tab.reset());

Tab.reset();

ASSERT\_ANY\_THROW(Tab.GetCurr());

}

TEST\_F(EmptyScanTable, cant\_setnext\_empty)

{

ASSERT\_ANY\_THROW(Tab.SetNext());

}

TEST\_F(EmptyScanTable, cant\_delete\_empty)

{

ASSERT\_ANY\_THROW(Tab.Delete("p"));

}

TEST\_F(EmptyScanTable, cant\_search\_empty)

{

ASSERT\_ANY\_THROW(Tab.Search("p"));

}

TEST\_F(EmptyScanTable, can\_insert\_empty)

{

ASSERT\_NO\_THROW(Tab.Insert("first", 1));

EXPECT\_EQ(1, Tab.GetCurr());

EXPECT\_EQ(1, Tab.Search("first"));

}

//---------------------------------------------------------

class FilledScanTable : public testing::Test

{

protected:

ScanTable<int> Tab;

public:

FilledScanTable() {

Tab.Insert("1", 1);

Tab.Insert("2", 2);

Tab.Insert("3", 3);

};

~FilledScanTable() {};

};

TEST\_F(FilledScanTable, can\_reset\_filled)

{

ASSERT\_NO\_THROW(Tab.reset());

}

TEST\_F(FilledScanTable, can\_getcurr\_filled)

{

Tab.reset();

ASSERT\_NO\_THROW(Tab.GetCurr());

EXPECT\_EQ(1, Tab.GetCurr());

}

TEST\_F(FilledScanTable, can\_setnext\_filled)

{

Tab.reset();

ASSERT\_NO\_THROW(Tab.SetNext());

EXPECT\_EQ(2, Tab.GetCurr());

}

TEST\_F(FilledScanTable, setnext\_isend\_and\_correct\_table)

{

Tab.reset();

EXPECT\_EQ(1, Tab.GetCurr());

Tab.SetNext();

EXPECT\_EQ(2, Tab.GetCurr());

Tab.SetNext();

EXPECT\_EQ(3, Tab.GetCurr());

Tab.SetNext();

EXPECT\_EQ(1, Tab.GetCurr());

}

TEST\_F(FilledScanTable, can\_copy\_and\_correct)

{

ASSERT\_NO\_THROW(ScanTable<int> Tab1(Tab));

ScanTable<int> Tab1(Tab);

Tab1.reset();

EXPECT\_EQ(1, Tab1.GetCurr());

Tab1.SetNext();

EXPECT\_EQ(2, Tab1.GetCurr());

Tab1.SetNext();

EXPECT\_EQ(3, Tab1.GetCurr());

Tab1.SetNext();

EXPECT\_EQ(1, Tab1.GetCurr());

}

TEST\_F(FilledScanTable, can\_delete\_and\_correct)

{

Tab.reset();

ASSERT\_NO\_THROW(Tab.Delete("1"));

Tab.reset();

EXPECT\_EQ(3, Tab.GetCurr());

}

TEST\_F(FilledScanTable, copy\_tab\_is\_own\_memory)

{ Tab.reset();

ScanTable<int> Tab1(Tab);

Tab.Delete("1");

EXPECT\_NE(Tab.GetCurr(), Tab1.GetCurr());

}

TEST\_F(FilledScanTable, can\_realloc)

{

for (int i = 3; i < 10; i++)

Tab.Insert(to\_string(i + 1), (i + 1));

ASSERT\_NO\_THROW(Tab.Insert("11", 11));

}

TEST\_F(FilledScanTable, can\_insert\_after\_deletion)

{

Tab.Delete("1");

Tab.Delete("2");

Tab.Delete("3");

Tab.Insert("1", 0);

EXPECT\_EQ(0, Tab.GetCurr());

}

TEST\_F(FilledScanTable, can\_search)

{

ASSERT\_NO\_THROW(Tab.Search("2"));

EXPECT\_EQ(2, Tab.Search("2"));

}

TEST\_F(FilledScanTable, can\_insret\_exect)

{

ASSERT\_ANY\_THROW(Tab.Insert("2", 2));

}

## Приложение 7. Тесты для класса SortedTable

#include "sorted\_table.h"

#include <gtest.h>

TEST(SortedTable, can\_create\_sorted\_table)

{

ASSERT\_NO\_THROW(SortedTable<int> T);

}

//---------------------------------------------------------

class EmptySortedTable : public testing::Test

{

protected:

SortedTable<int> Tab;

public:

EmptySortedTable() {};

~EmptySortedTable() {};

};

TEST\_F(EmptySortedTable, empty\_sorted\_tab\_is\_empty)

{

EXPECT\_EQ(true, Tab.isended());

}

TEST\_F(EmptySortedTable, copy\_empty\_sorted\_tab\_and\_correct)

{

ASSERT\_NO\_THROW(SortedTable<int> Tab1(Tab));

SortedTable<int> Tab1(Tab);

EXPECT\_EQ(true, Tab1.isended());

}

TEST\_F(EmptySortedTable, cant\_getcurr\_empty)

{

ASSERT\_ANY\_THROW(Tab.GetCurr());

}

TEST\_F(EmptySortedTable, can\_reset\_empty)

{

ASSERT\_NO\_THROW(Tab.reset());

Tab.reset();

ASSERT\_ANY\_THROW(Tab.GetCurr());

}

TEST\_F(EmptySortedTable, cant\_setnext\_empty)

{

ASSERT\_ANY\_THROW(Tab.SetNext());

}

TEST\_F(EmptySortedTable, cant\_delete\_empty)

{

ASSERT\_ANY\_THROW(Tab.Delete("p"));

}

TEST\_F(EmptySortedTable, cant\_search\_empty)

{

ASSERT\_ANY\_THROW(Tab.Search("p"));

}

TEST\_F(EmptySortedTable, can\_insert\_empty)

{

ASSERT\_NO\_THROW(Tab.Insert("first", 1));

EXPECT\_EQ(1, Tab.GetCurr());

EXPECT\_EQ(1, Tab.Search("first"));

}

//---------------------------------------------------------

class FilledSortedTable : public testing::Test

{

protected:

SortedTable<int> Tab;

public:

FilledSortedTable() {

Tab.Insert("3", 3);

Tab.Insert("2", 2);

Tab.Insert("1", 1);

};

~FilledSortedTable() {};

};

TEST\_F(FilledSortedTable, can\_reset\_filled)

{

ASSERT\_NO\_THROW(Tab.reset());

}

TEST\_F(FilledSortedTable, can\_getcurr\_filled)

{

Tab.reset();

ASSERT\_NO\_THROW(Tab.GetCurr());

EXPECT\_EQ(1, Tab.GetCurr());

}

TEST\_F(FilledSortedTable, can\_setnext\_filled)

{

Tab.reset();

ASSERT\_NO\_THROW(Tab.SetNext());

EXPECT\_EQ(2, Tab.GetCurr());

}

TEST\_F(FilledSortedTable, setnext\_isend\_and\_correct\_table\_insert)

{

Tab.reset();

EXPECT\_EQ(1, Tab.GetCurr());

Tab.SetNext();

EXPECT\_EQ(2, Tab.GetCurr());

Tab.SetNext();

EXPECT\_EQ(3, Tab.GetCurr());

Tab.SetNext();

EXPECT\_EQ(1, Tab.GetCurr());

}

TEST\_F(FilledSortedTable, can\_copy\_and\_correct)

{

ASSERT\_NO\_THROW(SortedTable<int> Tab1(Tab));

SortedTable<int> Tab1(Tab);

Tab1.reset();

EXPECT\_EQ(1, Tab1.GetCurr());

Tab1.SetNext();

EXPECT\_EQ(2, Tab1.GetCurr());

Tab1.SetNext();

EXPECT\_EQ(3, Tab1.GetCurr());

Tab1.SetNext();

EXPECT\_EQ(1, Tab1.GetCurr());

}

TEST\_F(FilledSortedTable, can\_delete\_and\_correct)

{

Tab.reset();

ASSERT\_NO\_THROW(Tab.Delete("1"));

Tab.reset();

EXPECT\_EQ(2, Tab.GetCurr());

}

TEST\_F(FilledSortedTable, copy\_tab\_is\_own\_memory)

{

SortedTable<int> Tab1(Tab);

Tab.reset();

Tab.Delete("1");

EXPECT\_NE(Tab.GetCurr(), Tab1.GetCurr());

}

TEST\_F(FilledSortedTable, can\_realloc)

{

for (int i = 3; i < 10; i++)

Tab.Insert(to\_string(i + 1), i);

ASSERT\_NO\_THROW(Tab.Insert("11", 11));

}

TEST\_F(FilledSortedTable, can\_insert\_after\_deletion)

{

Tab.Delete("1");

Tab.Delete("2");

Tab.Delete("3");

Tab.Insert("1", 0);

EXPECT\_EQ(0, Tab.GetCurr());

}

TEST\_F(FilledSortedTable, can\_search)

{

ASSERT\_NO\_THROW(Tab.Search("2"));

EXPECT\_EQ(2, Tab.Search("2"));

}

TEST\_F(FilledSortedTable, can\_insret\_exect)

{

ASSERT\_ANY\_THROW(Tab.Insert("2", 2));

}

## Приложение 8. Тесты для класса HashTable

#include "hash\_table.h"

#include <gtest.h>

TEST(HashTable, can\_create\_HashTable\_table)

{

ASSERT\_NO\_THROW(HashTable<int> T);

}

//---------------------------------------------------------

class EmptyHashTable : public testing::Test

{

protected:

HashTable<int> Tab;

public:

EmptyHashTable() {};

~EmptyHashTable() {};

};

TEST\_F(EmptyHashTable, empty\_HashTable\_tab\_is\_empty)

{

EXPECT\_EQ(true, Tab.isended());

}

TEST\_F(EmptyHashTable, copy\_empty\_HashTable\_tab\_and\_correct)

{

ASSERT\_NO\_THROW(HashTable<int> Tab1(Tab));

HashTable<int> Tab1(Tab);

EXPECT\_EQ(true, Tab1.isended());

}

TEST\_F(EmptyHashTable, cant\_getcurr\_empty)

{

ASSERT\_ANY\_THROW(Tab.GetCurr());

}

TEST\_F(EmptyHashTable, can\_reset\_empty)

{

ASSERT\_NO\_THROW(Tab.reset());

Tab.reset();

ASSERT\_ANY\_THROW(Tab.GetCurr());

}

TEST\_F(EmptyHashTable, cant\_setnext\_empty)

{

ASSERT\_ANY\_THROW(Tab.SetNext());

}

TEST\_F(EmptyHashTable, cant\_delete\_empty)

{

ASSERT\_ANY\_THROW(Tab.Delete("p"));

}

TEST\_F(EmptyHashTable, cant\_search\_empty)

{

ASSERT\_ANY\_THROW(Tab.Search("p"));

}

TEST\_F(EmptyHashTable, can\_insert\_empty)

{

ASSERT\_NO\_THROW(Tab.Insert("first", 1));

EXPECT\_EQ(1, Tab.GetCurr());

EXPECT\_EQ(1, Tab.Search("first"));

}

//---------------------------------------------------------

class FilledHashTable : public testing::Test

{

protected:

HashTable<int> Tab;

public:

FilledHashTable() {

Tab.Insert("1", 1);

Tab.Insert("2", 2);

Tab.Insert("3", 3);

};

~FilledHashTable() {};

};

TEST\_F(FilledHashTable, can\_reset\_filled)

{

ASSERT\_NO\_THROW(Tab.reset());

}

TEST\_F(FilledHashTable, can\_getcurr\_filled)

{

Tab.reset();

ASSERT\_NO\_THROW(Tab.GetCurr());

//EXPECT\_EQ(1, Tab.GetCurr());

}

TEST\_F(FilledHashTable, can\_setnext\_filled)

{

Tab.reset();

ASSERT\_NO\_THROW(Tab.SetNext());

//EXPECT\_EQ(2, Tab.GetCurr());

}

TEST\_F(FilledHashTable, can\_search)

{

ASSERT\_NO\_THROW(Tab.Search("2"));

EXPECT\_EQ(2, Tab.Search("2"));

}

TEST\_F(FilledHashTable, correct\_table)

{

Tab.reset();

EXPECT\_EQ(1, Tab.Search("1"));

EXPECT\_EQ(2, Tab.Search("2"));

EXPECT\_EQ(3, Tab.Search("3"));

}

TEST\_F(FilledHashTable, can\_copy\_and\_correct)

{

ASSERT\_NO\_THROW(HashTable<int> Tab1(Tab));

HashTable<int> Tab1(Tab);

Tab1.reset();

EXPECT\_EQ(1, Tab1.Search("1"));

EXPECT\_EQ(2, Tab1.Search("2"));

EXPECT\_EQ(3, Tab1.Search("3"));

}

TEST\_F(FilledHashTable, can\_delete\_and\_correct)

{

Tab.reset();

ASSERT\_NO\_THROW(Tab.Delete("1"));

Tab.reset();

ASSERT\_ANY\_THROW(Tab.Search("1"));

}

TEST\_F(FilledHashTable, copy\_tab\_is\_own\_memory)

{

Tab.reset();

HashTable<int> Tab1(Tab);

Tab.Delete("1");

EXPECT\_EQ(1, Tab1.Search("1"));

}

TEST\_F(FilledHashTable, can\_realloc)

{

for (int i = 3; i < 10; i++)

Tab.Insert(to\_string(i + 1), (i + 1));

ASSERT\_NO\_THROW(Tab.Insert("11", 11));

}

TEST\_F(FilledHashTable, can\_insert\_after\_deletion)

{

Tab.Delete("1");

Tab.Delete("2");

Tab.Delete("3");

Tab.Insert("1", 0);

EXPECT\_EQ(0, Tab.Search("1"));

}

TEST\_F(FilledHashTable, can\_insert\_exect)

{

ASSERT\_ANY\_THROW(Tab.Insert("2", 2));

}