МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра математического обеспечения и применения ЭВМ

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: Хеш-таблицы с открытой адресацией

Студент гр. 9382 _____ Михайлов Д.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

ЗАДАНИЕ
НА КУРСОВУЮ РАБОТУ

Студент Михайлов Д.А.

Группа 9382

Тема работы: Хеш-таблицы с открытой адресацией – вставка. Исследование (в среднем, в худшем случае)

Исходные данные: На вход программе подается один из двух заготовленных вариантов текстовых файлов для дальнейшей реализации их в качестве хеш-таблицы

Содержание пояснительной записки:
«Содержание», «Описание программы», «Тестирование», «Исследование», «Заключение», «Список использованных источников».

Предполагаемый объем пояснительной записки:
Не менее 14 страниц.

Дата выдачи задания:

Дата сдачи реферата:

Дата защиты реферата:

Студент		Михайлов Д.А.
Преподаватель		Фирсов М.А.

АННОТАЦИЯ

В ходе выполнения курсовой работы была разработана программа, позволяющая исследовать алгоритм вставки в хеш-таблицу с открытой адресацией. Программа способна создавать хеш-таблицы по заданному текстовому файлу. В результате исследования было выявлено, что данные о скорости выполнения программы, полученные в результате тестирования, сходятся с теоретическими.

SUMMARY

A program, that allows you to explore the algorithm of inserting to the hash table with open addressing was developed in the course work. The program can create hash tables with a following text file. The study showed, that information about execution speed of program, received as a result of testing, is exactly as the theoretical.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Задание

Реализация хеш-таблицы с открытой адресацией и вставкой. Провести исследование (в среднем, в худшем случае). Вариант 26.

Основные задачи

Генерация входных данных, использование их для измерения количественных характеристик структур данных, алгоритмов, действий, сравнение экспериментальных результатов с теоретическими.

Методы решения

Разработка программы велась на базе ОС Ubuntu 18.04 в среде разработки Clion. Для реализации хеш-таблицы были созданы классы Pair, динамического массива CVector, хеш-таблицы HashTable. Для работы таблицы с ключами и значениями в формате строк был создан класс StrWorker.

1. ОПИСАНИЕ ПРОГРАММЫ

1.1. Описание основных классов для реализации хеш-таблицы

Таблица 1 - Основные методы класса Pair

Метод	Назначение
`getFirst(S);`	Возвращает первый элемент пары типа S
`getSecond(T);`	Возвращает второй элемент пары типа T
`int isActive();`	Возвращает true, если в пару заносились и не удалялись какие-либо значения
`void setFirst(S);`	Устанавливает первый элемент типа S
`void setSecond(T);`	Устанавливает второй элемент типа T
`void setDeleted();`	Устанавливает флаг, помечающий пару как ранее удаленную

Динамический массив CVector для хэш-таблицы используется для хранения пар «ключ-значение», реализован с помощью шаблонного класса, что позволяет работать с любым простым типом данных. Основные методы класса приведены в табл. 2.

Таблица 2 – Основные методы класса CVector

Метод	Назначение
`Unsigned int getCapacity();`	Возвращает текущий объем выделенной памяти для динамического массива в элементах
`int resize (unsigned int nsize);`	Изменяет объем выделенной памяти, точнее выделяет память с новым объемом, копируя туда элементы массива . Принимаемое значение — размер массива

`T & operator[] (unsigned int index);`	Позволяет обращаться к элементу массива с помощью скобок [] по индексу Принимаемое значение - индекс
`CVector<T> & operator= (const CVector<T> &);`	Копирует элементы массива (присваивание) Принимаемое значение — сам массив

Для реализации хеш-таблицы был создан шаблонный класс HashTable, который хранит в себе динамический массив пар и обеспечивает выполнение операций вставки и удаления. Основные методы класса приведены в табл. 3.

Таблица 3 – Основные методы класса HashTable

Метод	Назначение
`void setSize(int size);`	Изменяет объем выделенной памяти для таблицы (динамического массива) Принимаемое значение — размер массива
`int getSize();`	Возвращает объем выделенной памяти для таблицы (количество пар)
`Pair<S,T> & operator[] (unsigned int index);`	Позволяет обращаться к элементу таблицы с помощью скобок [] по индексу Принимаемое значение — индекс
`T & operator[] (unsigned int index);`	Позволяет обращаться к элементу массива с помощью скобок [] по индексу Принимаемое значение - индекс
`unsigned int set(S x, T y, unsigned int index);`	Осуществляет вставку в таблицу по правилам открытой адресации Принимаемое значение — x, y, индекс
`Pair<S,T> get(S key, unsigned int index);`	Возвращает элемент таблицы (пару) по заданному ключу и индексу (хеш-функции) Принимаемое значение — ключ и индекс
`unsigned int remove(S key, unsigned int`	Удаляет элемент таблицы (пару) по заданному ключу и индексу (хеш-функции),

`index);`	помечая этот элемент как удаленный для реализации открытой адресации Принимаемое значение — ключ и индекс
`string print();`	Возвращает строку, в которой содержится представление хеш-таблицы в формате: <индекс> (<ключ>,<значение>)

Для демонстрации работы алгоритмов хеширования, вставки и удаления из хеш-таблицы для ключей и значений был выбран формат строк. Для функций, отвечающих за ввод и вывод, был создан класс StrWorker. Основные методы класса приведены в табл. 4.

Таблица 4 – Основные методы класса StrWorker

Метод	Назначение
`static string getFromFile (string fileName);`	Возвращает содержимое файла по указанному пути в виде строки Принимаемое значение — имя файла
`String createHashTable (string input);`	Создает хеш-таблицу, где ключи и значения представлены строками, и возвращает строку с информацией о числе итераций и элементами таблицы Принимаемое значение — ключи и значения
`string deleteElem (string input);`	Удаляет элемент, если таковой имеется, из хеш-таблицы, возвращает строку с информацией о числе итераций и элементами таблицы Принимаемое значение — ключи и значения
`static unsigned int hFunc (string key);`	Возвращает индекс элемента по ключу. Индекс формируется как целое среднее значение кодов символов в ключе Принимаемое значение — ключ

1.2. Описание алгоритма вставки в хеш-таблицу

Хеш-таблица содержит массив, элементы которого есть пары. Выполнение вставки в хеш-таблицу начинается с вычисления хеш-функции. Алгоритм вставки элемента последовательно проверяет ячейки массива, начиная с ячейки с индексом значения хеш-функции, пока не будет найдена первая свободная ячейка, в которую и будет записан новый элемент. Используется линейное пробирование, то есть ячейки хеш-таблицы последовательно просматриваются друг за другом, с единичным интервалом. Пример коллизии с разрешением открытой адресацией представлен на рис. 1.

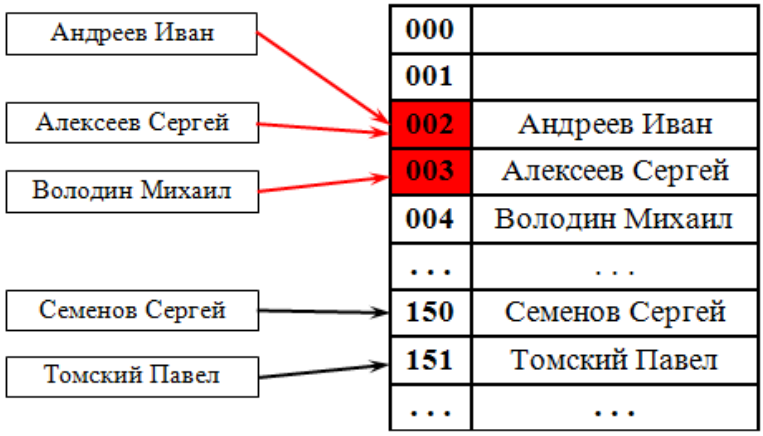


Рисунок 1 – Пример коллизии в открытой адресации

Здесь стрелки показывают значение хеш-функции для ключа. Так как у ключей «Андреев Иван» и «Алексеев Сергей» одинаковое значение хеш-функции, последний добавленный размещается на следующую ячейку. Так как ячейка массива по значению хеш-функции ключа «Володин Михаил» занята, то этот ключ также размещается на следующей ячейке.

1.3. Описание алгоритма работы программы

Создание хеш-таблицы осуществляется в методе createHashTable, на вход которого передаются данные, представленные в текстовом формате. В цикле происходит их построчный разбор и добавление в динамический массив типа

HashTable, в ячейки которого сохраняются пары в формате [ключ, значение]. Здесь же происходит проверка корректности данных, в случае ошибки дальнейшее исполнение алгоритма прерывается. Вставка элемента в таблицу по правилам открытой адресации на каждом шаге цикла осуществляется с помощью метода HashTable::set. Его входными параметрами являются: пара [ключ, значение] и результат хэш-функции ключа, играющего роль индекса в хэш-массиве. Алгоритм вставки проходит по массиву до тех пор, пока не будет найдена первая свободная ячейка, в которую и будет записан новый элемент. Этот метод возвращает в клиентский код число итераций, потребовавшихся для операции вставки. Количество итераций зависит от возникающих коллизий, когда для различных ключей получается одно и то же хеш-значение. Хэш-функция реализована в методе hFunc, входным параметром которой является ключ в формате строки. Она возвращает индекс в виде целого числа, вычисленный как среднее значение кодов символов в ключе. Удаление элемента хеш-таблицы по правилам открытой адресации реализовано в методе HashTable::remove, на вход которого передается ключ и индекс хеш-функции. Каждая ячейка имеет булевый флаг deleted, помечающий, удален элемент в ней или нет. Таким образом, удаление элемента состоит в установке этого флага для соответствующей ячейки хеш-таблицы, а процедуры добавления - в том, чтобы алгоритм считал их свободными и сбрасывал значение флага при добавлении.

2. ТЕСТИРОВАНИЕ

1. Создание хеш-таблицы со средним количеством коллизий, состоящей из 5 элементов

```
Введите вариант входных данных
1 - среднее количество коллизий
2 - большое количество коллизий
0 - выход из программы
> 1
Число элементов > 5

49  (1,Pixies)
51  (333,Wire)
53  (55555,INXS)
97  (ab,Cure)
98  (ab,Stranglers)

------------------------------------------------------
Число элементов: 5
Коллизий: 1
Число итераций всех вставок: 6
------------------------------------------------------
```

2. Создание хеш-таблицы с большим количеством коллизий, состоящей из 5 элементов

```
Введите вариант входных данных
1 - среднее количество коллизий
2 - большое количество коллизий
0 - выход из программы
> 2
Число элементов > 5

97  (a,test)
98  (aa,test)
99  (aaa,test)
100 (aaaa,test)
101 (aaaaa,test)

------------------------------------------------------
Число элементов: 5
Коллизий: 4
Число итераций всех вставок: 15
------------------------------------------------------
```

3. ИССЛЕДОВАНИЕ

1. Тестирование среднего количества коллизий

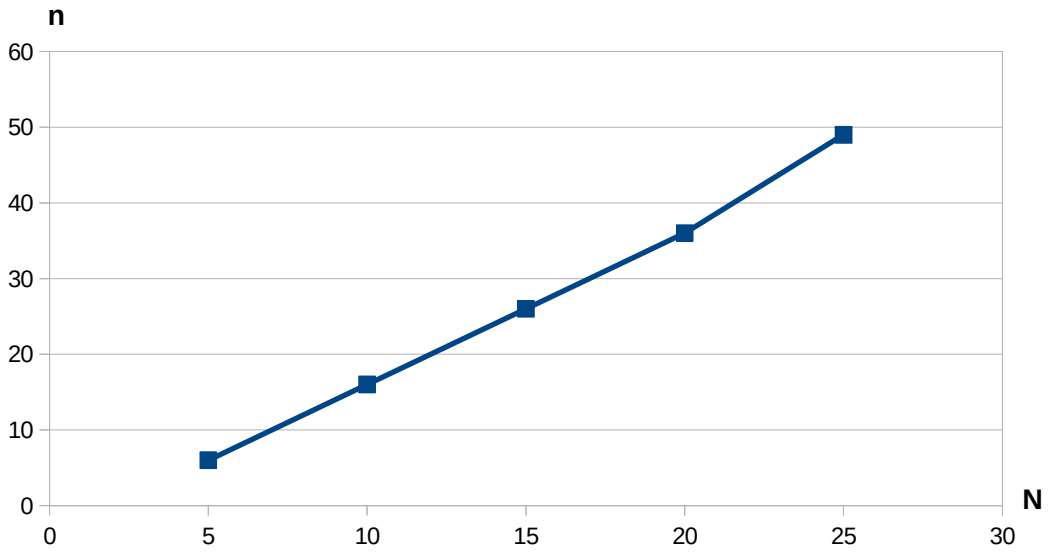
№	Элементов	Коллизий	Итераций
1	5	1	6
2	10	3	16
3	15	4	26
4	20	5	36
5	25	8	49

2. Тестирование большого количества коллизий

№	Элементов	Коллизий	Итераций
1	5	4	15
2	10	9	55
3	15	14	120
4	20	19	210
5	25	24	325

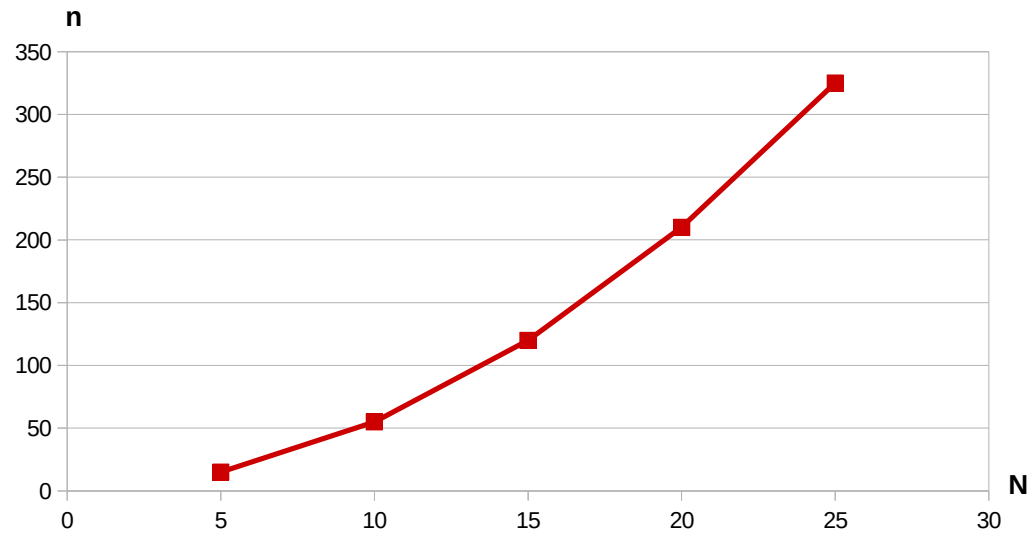
На основе полученных данных был построен график зависимости числа итераций от числа элементов хеш-таблицы n(N).

1. Среднее количество коллизий



На графике видна линейная зависимость *n* от N, что подтверждает оценку алгоритма вставки одного элемента *O*(1).

2. Большое количество коллизий



На основе полученных данных видно, что в худшем случае максимальное число итераций для вставки одного элемента совпадает с числом элементов, что соответствует сложности O(N) . Это связано с тем, что алгоритм вставки с открытой адресацией, добавляя последний элемент, проходит по всем предыдущим элементам.

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы была написана программа для создания хеш-таблиц с открытой адресацией. Была исследована сложность алгоритмов вставки элементов. Экспериментальные данные, полученные в ходе работы, полностью совпали с теоретическими: операции вставки в среднем выполняются за время O(1), а в худшем - за O(N). Исходя из этого можно сделать вывод, что хеш-таблица обеспечивает высокую скорость работы и представляет собой эффективную структуру для хранения и обработки данных.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Bjarne Stroustrup. A Tour of C++. M.: Addison-Wesley, 2018. 217 с.
2. Макс Шлее. Qt5.10. Профессиональное программирование на C++. M.: BHV-СПб, 2018, 513 с.
3. Перевод и дополнение документации QT // CrossPlatform.RU. URL: http://doc.crossplatform.ru/
4. Хеш-таблицы // AlgoList. URL: http://algolist.ru/ds/s_has.php
5. Примеры хеш-функций // Poznayka. URL: https://poznayka.org/s97484t1.html

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ. MAIN.CPP

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>

using namespace std;

#define SIZE 256 // количество элементов хэш-таблицы по умолчанию

/**
 * Шаблонный классы для пары
 * @tparam S
 * @tparam T
 */
template <typename S, typename T>
class Pair {
public:
    Pair();
    Pair(S, T);
    Pair(Pair &);
    ~Pair();
    const Pair & operator=(const Pair &other);
    string toString();
    S getFirst();
    T getSecond();
    void setFirst(S);
    void setSecond(T);
    void setDeleted();
    int isActive();
    int wasDeleted();
private:
    S *f;
    T *s;
    bool active;
    bool deleted;
};

// Конструкторы и деструктор класса

template <typename S, typename T>
Pair<S, T>::Pair() {
    f = nullptr;
    s = nullptr;
```

```
    active = false;
    deleted = false;
}

template <typename S, typename T>
Pair<S, T>::Pair(S x, T y) {
    f = new S;  *f = x;
    s = new T;  *s = y;
    active = true;
    deleted = false;
}

template <typename S, typename T>
Pair<S, T>::Pair(Pair &other) {
    f = NULL; s = NULL;
    if (other.f != NULL)
        f = new S(*other.f);
    if (other.s != NULL)
        s = new T(*other.s);
}

template <typename S, typename T>
Pair<S, T>::~Pair() {
    if (f != nullptr)
        delete f;
    if (s != nullptr)
        delete s;
    f = nullptr;
    s = nullptr;
    deleted = false;
    active = false;
}

//

/**
 * Возвращает первый элемент пары типа S
 */
template <typename S, typename T>
S Pair<S, T>::getFirst() {
    if (f != nullptr)
        return *f;
    else
        return nullptr;
}

/**
```

```
 * Возвращает второй элемент пары типа T
 */
template <typename S, typename T>
T Pair<S, T>::getSecond() {
    if (s != NULL)
        return *s;
    else
        return NULL;
}

/**
 * Устанавливает первый элемент типа S
 */
template <typename S, typename T>
void Pair<S, T>::setFirst(S x) {
    if (f == nullptr)
        f = new S;
    *f = x;
    active = true;
    deleted = false;
}

/**
 * Устанавливает второй элемент типа T
 */
template <typename S, typename T>
void Pair<S, T>::setSecond(T y) {
    if (s == nullptr)
        s = new T;
    *s = y;
    active = true;
    deleted = false;
}

/**
 * Перегруженная функция toString для вывода значений в виде строки
 */
template <typename S, typename T>
string Pair<S, T>::toString() {
    stringstream ss;
    if (active) {
        ss << "(";
        if (f == nullptr)
            ss << "NULL";
        else
            ss << (*f);
        ss<<",";
```

```
        if (s == nullptr)
            ss << "NULL";
        else
            ss << (*s);
        ss << ")";
    } else {
        ss << "NULL";
    }
    return ss.str();
}

/**
 * Возвращает true, если в пару заносились и не удалялись какие-либо
значения
 */
template <typename S, typename T>
int Pair<S, T>::isActive() {
    return active;
}

/**
 * Возвращает true, если пара удалена
 */
template <typename S, typename T>
int Pair<S, T>::wasDeleted() {
    return deleted;
}

/**
 * Устанавливает флаг, помечающий пару как ранее удаленную
 */
template <typename S, typename T>
void Pair<S,T>::setDeleted() {
    deleted = true;
}

/**
 * Копирует элементы пары (перегрузка оператора присваивания)
 */
template <typename S, typename T>
const Pair<S, T> & Pair<S,T>::operator=(const Pair<S, T> &other) {
    if(this != &other) {
        if(f != nullptr)
            delete f;
        if(s != nullptr)
            delete s;
        f = nullptr;
```

```
        s = nullptr;
        if (other.f != nullptr)
            f = new S(*other.f);
        if (other.s != nullptr)
            s = new T(*other.s);
        deleted = other.deleted;
        active = other.active;
    }
    return *this;
}


/**
 * Шаблонный класс динамического массива для хранения пар ключ-значение
 * @tparam T
 */
template <typename T>
class CVector {
public:
    CVector();
    CVector(unsigned int nsize);
    unsigned int getCapacity();
    int resize(unsigned int nsize);
    T & operator[](unsigned int index);
    CVector<T> & operator=(const CVector<T> &);
private:
    T *tail;
    unsigned int capacity;
};

// Конструкторы

template <typename T>
CVector<T>::CVector() {
    tail = new T[1];
    capacity = 1;
};

template <typename T>
CVector<T>::CVector(unsigned int nsize) {
    capacity = nsize;
    tail = new T[capacity];
}

//

/**
```

```
 * Возвращает текущий объем выделенной памяти для хранения динамического
массива
 */
template <typename T>
unsigned int CVector<T>::getCapacity() {
    return capacity;
}

/**
 * Перевыделяет память для хранения массива с новым заданным объемом и
сохранением всех его элементов
 */
template <typename T>
int CVector<T>::resize(unsigned int nsize) {
    if (nsize <= capacity)
        return 1;
    T *temp = new T[nsize];
    for (unsigned int i = 0; i<capacity; i++) {
        temp[i] = tail[i];
    }
    capacity = nsize;
    delete [] tail;
    tail = temp;
    return 0;
}

/**
 * Перегрузка оператора [], позволяющая обращаться к элементу массива по
цифровому индексу
 */
template <typename T>
T& CVector<T>::operator[](unsigned int index) {
    return tail[index];
}

/**
 * Копирует элементы массива (перегрузка оператора присваивания)
 */
template<class T>
CVector<T> & CVector<T>::operator=(const CVector<T> & v) {
    delete[] tail;
    capacity = v.capacity;
    tail = new T[capacity];
    for (unsigned int i = 0; i < capacity; i++)
        tail[i] = v.tail[i];
    return *this;
}
```

```
/**
  * Шаблонный класс хеш-таблицы, содержит динамический массив пар и
обеспечивает выполнение операций вставки и удаления
 * @tparam S
 * @tparam T
 */
template <typename S, typename T>
class HashTable {
public:
    HashTable();
    HashTable(unsigned int size);
    void setSize(int size);
    int getSize();
    unsigned int getItemCounter();
    Pair<S, T> & operator[](unsigned int index);
    unsigned int set(S x, T y, unsigned int index);
    Pair<S, T> get(S key, unsigned int index);
    unsigned int remove(S key, unsigned int index);
    void clear();
    string print();
private:
    CVector< Pair<S,T> > table;
    unsigned int itemCounter;
};

// Конструкторы класса

template <typename S, typename T>
HashTable<S,T>::HashTable() {
    itemCounter = 0;
    setSize(SIZE);
}

template <typename S, typename T>
HashTable<S,T>::HashTable(unsigned int size) {
    itemCounter = 0;
    setSize(size);
}

//

/**
 * Изменяет объем выделенной памяти для таблицы (динамического массива)
 */
template <typename S, typename T>
```

```
void HashTable<S, T>::setSize(int size) {
    table.resize(size);
}

/**
 * Возвращает объем выделенной памяти для таблицы
 */
template <typename S, typename T>
int HashTable<S, T>::getSize() {
    return table.getCapacity();
}

/**
 * Возвращает количество элементов (пар) таблицы
 */
template <typename S, typename T>
unsigned int HashTable<S, T>::getItemCounter() {
    return itemCounter;
}

/**
  * Осуществляет вставку элемента в таблицу по правилам открытой адресации:
 * алгртим проходит по массиву до тех пор, пока не будет найдена первая свободная ячейка, в которую и будет записан новый элемент
 */
template <typename S, typename T>
unsigned int HashTable<S, T>::set(S x, T y, unsigned int index) {
    unsigned int counter = 1;
    if (index + 5 >= table.getCapacity())
        table.resize((index + 5) * 2);
    while (table[index].isActive()) {
        index++;
        counter++;
        if (index + 5 >= table.getCapacity())
            table.resize((index + 5) * 2);
    }
    table[index].setFirst(x);
    table[index].setSecond(y);
    itemCounter++;
    return counter;
}

/**
 * Возвращает элемент таблицы (пару) по  заданному ключу и индексу хеш-функции
 */
```

```
template <typename S, typename T>
Pair<S, T> HashTable<S, T>::get(S key, unsigned int index) {
    while (table[index].isActive()) {
        if (table[index].wasDeleted()) {
            index++;
            continue;
        }
        if (table[index].getFirst() != key)
            index++;
        else
            break;
    }
    return table[index];
}

/**
 * Удаляет элемент таблицы (пару) по заданному ключу и индексу хеш-функции, помечая этот элемент как удаленный для реализации открытой адресации
 */
template <typename S, typename T>
unsigned int HashTable<S, T>::remove(S key, unsigned int index) {
    unsigned int counter = 1;
    while (table[index].isActive()) {
        if (table[index].wasDeleted()) {
            index++;
            counter++;
            continue;
        }
        if (table[index].getFirst() != key) {
            index++;
            counter++;
        } else
            break;
    }
    if (table[index].isActive()) {
        table[index].~Pair();
        table[index].setDeleted();
        itemCounter--;
        return counter;
    } else
        return 0;
}

/**
 * Удаляет все элементы хэш-массива (очистка таблицы)
 */
```

```
template <typename S, typename T>
void HashTable<S, T>::clear() {
    for (unsigned int i=0; i<table.getCapacity(); i++) {
        table[i].~Pair();
    }
    itemCounter = 0;
}

/**
 * Возвращает строку, в которой содержится  представление хеш-таблицы в
формате: <индекс> (<ключ>,<значение>)
 * @tparam S
 * @tparam T
 * @return
 */
template <typename S, typename T>
string HashTable<S, T>::print() {
    string output;
    for (unsigned int i = 0; i < table.getCapacity(); i++) {
        if (table[i].isActive() && !table[i].wasDeleted()) {
            output += to_string(i);
            output += "\t";
            output += table[i].toString();
            output += "\n";
        }
    }
    return output;
}

/**
 * Позволяет обращаться к элементу массива по индексу
 */
template <typename S, typename T>
Pair<S, T> & HashTable<S, T>::operator[](unsigned int index) {
    return table[index];
}


/**
 * Шаблонная функция, разбивающая строку на отдельные элементы
 * @param s входная строка
 * @param delimiter символ-разделитель
 * @return массив элементов
 */
vector<string> split(const string &s, char delimiter) {
    stringstream ss(s);
    string item;
```

```
    vector<string> elements;
    while (getline(ss, item, delimiter)) {
        elements.push_back(move(item));
    }
    return elements;
}


/**
 * Класс, реализующий работу алгоритмов хеширования, вставки и удаления
из хеш-таблицы, ввода и вывода данных
 */
class StrWorker {
public:
    static string getFromFile(string, int);
    string createHashTable(string, int);
    string deleteElem(string input);
    bool error = false;
private:
    HashTable <string, string> hTable;
    static unsigned int hFunc(string key);
    int iter = 0;
};

/**
 * Построчное чтение из файла
 * @param fileName имя файла
 * @param num количество элементов
 * @return строка с входными данными
 */
string StrWorker::getFromFile(string fileName, int num) {
    string out, temp;
    ifstream fin(fileName, ios::in);
    for (int i = 0; i < num; i++) {
        getline(fin, temp);
        out += temp;
        if (!fin.eof())
            out += "\n";
        else
            break;
    }
    fin.close();
    return out;
}

/**
 * Создает хеш-таблицу
```

```
 * @param input входные данные в формате: пары разделены переводами строк, а их элементы (ключи и значения) - пробелами
 * @param num количество элементов
 * @return возвращает строку с информацией о числе итераций и элементами хэш-таблицы
 */
string StrWorker::createHashTable(string input, int num) {
    hTable.setSize(num);
    hTable.clear();
    vector<string> pairs = split(input, '\n');
    vector<string> temp;
    string output;
    unsigned int sumIter = 0;
    unsigned int maxIter = 0;
    for (auto & pair : pairs) {
        temp = split(pair, ' ');
        if (temp.size() != 2) {
            output += "Ошибка! Некорректное значение в строке:\n\"";
            output += pair;
            output += "\"";
            error = true;
            return output;
        }
        iter = hTable.set(temp[0], temp[1], hFunc(temp[0]));
        if (iter > maxIter)
            maxIter = iter;
        sumIter += iter;
    }
                                            output          +=          "\
n----------------------------------------------------\n";
    output += "Число элементов: ";
    output += to_string(hTable.getItemCounter());
    output += "\nКоллизий: ";
    output += to_string(maxIter - 1);
      //output += "\nМаксимальное число итераций одной вставки: " +
to_string(maxIter);
    output += "\nЧисло итераций всех вставок: ";
    output += to_string(sumIter);
    output += "\n";
     output += "-----------------------------------------------------\
n";
    output = hTable.print() + output;
    return output;
}

/**
 * Удаляет элемент из хеш-таблицы, если таковой имеется
```

```
 * @param input ключ
 * @return возвращает строку с информацией о числе итераций и элементами таблицы
 */
string StrWorker::deleteElem(string input) {
    string output;
    iter = hTable.remove(input, hFunc(input));
    output += "\nКоличество элементов: ";
    output += to_string(hTable.getItemCounter());
    if (iter == 0) {
        output += "\nОшибка! Ключ \"" + input + "\" не найден";
    } else {
        output += "\nИтераций удаления: ";
        output += to_string(iter);
    }
    output += "\n";
    output = hTable.print() + output;
    return output;
}

/**
 * Возвращает индекс элемента по ключу. Индекс формируется как целое среднее значение кодов символов в ключе
 */
unsigned int StrWorker::hFunc(string key) {
    unsigned int result = 0;
    for (char i : key) {
        result += static_cast<unsigned char>(i);
    }
    result /= key.length();
    return result;
}

int main() {
    int tblType, elmNum;
    string data, fileName, hash, key;
    auto* worker = new StrWorker();
    cout << "Исследование хеш-таблиц с открытой адресацией" << endl;
    cout << "*********************************************" << endl;
    do {
         cout << "Введите вариант входных данных\n1 - среднее количество коллизий\n2 - большое количество коллизий\n0 - выход из программы\n> ";
        cin >> tblType;
        switch (tblType) {
            case 1:
                fileName = "inputmedium.txt";
                break;
```

```
            case 2:
                fileName = "inputmany.txt";
                break;
            default:
                return 0;
        }
        cout << "Число элементов > ";
        cin >> elmNum;
        cout << endl;
        data = worker->getFromFile(fileName, elmNum);
        hash = worker->createHashTable(data, elmNum);
        cout << hash << endl;
        if (worker->error) {
            cout << "Работа программы прервана" << endl;
            break;
        }
    } while (tblType == 1 || tblType == 2);
    return 0;
}
```