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

ОТЧЕТ

по лабораторной работе №4
по дисциплине «Алгоритмы и структуры данных»
Тема: БДП и хеш-таблицы

Студент гр. 9382 _____ Михайлов Д.А.

Преподаватель _____ Ерёменко А.А.

Санкт-Петербург
2020

Цель работы.

Ознакомиться с основными характеристиками и особенностями такой структуры данных, как хэш-таблица. Исследовать сложность таких операций, как вставка, удаление из хеш-таблицы.

Задание (вариант 26).

Необходимо реализовать хеш-таблицу с открытой адресацией и следующей функциональностью:

- Построение хеш-таблицы по заданному файлу F (типа file of *Elem*), все элементы которого различны;
- Для построенной структуры данных проверить, входит ли в неё элемент *e* типа *Elem*, и если входит, то удалить элемент *e* из структуры данных. Предусмотреть возможность повторного выполнения с другим элементом.

Описание алгоритма

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.

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», однако реализован также с помощью шаблонов. 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 метода 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 index);`	Удаляет элемент таблицы (пару) по заданному ключу и индексу (хеш-функции), 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

Оценка сложности алгоритма

Сложность алгоритма вставки в хеш-таблицу зависит от количества коллизий, возникших при ее заполнении. Если известно количество коллизий $C - const$, то число итераций не превысит $O(C)$, то есть в среднем случае сложность алгоритма $O(1)$. В худшем случае, если все элементы создают коллизию, алгоритм каждый раз будет совершать количество итераций, равное числу элементов в таблице. Таким образом, в худшем случае сложность алгоритма вставки элемента равна $O(N)$.

Сложность алгоритма удаления элемента из хеш-таблицы таким же образом зависит от количества итераций и в худшем случае составляет $O(N)$, а в среднем случае $O(1)$.

Тестирование программы

1. Незначительное количество коллизий:

```
Создана хэш-таблица:

33  (!!!,attention)
97  (ab,коллизия1_1)
98  (ba,коллизия1_2)
105 (do,делать)
106 (hello,привет)
110 (no,нет)
111 (how,как)
112 (yes,да)
116 (you,вы)
179 (тор,thor_collision)
180 (орт,orth_collision)
181 (рот,mouth_collision)
187 (очень,very)
192 (наверное,may_be)

Число элементов: 14
Коллизий: 2
Максимальное число итераций одной вставки: 3
Число итераций всех вставок: 18
```

2. Большое количество коллизий:

```
Создана хэш-таблица:

97  (a,test)
98  (aa,test)
99  (aaa,test)
100 (aaaa,test)
101 (aaaaa,test)
102 (aaaaaa,test)
103 (aaaaaaa,test)
104 (aaaaaaaa,test)
105 (aaaaaaaaa,test)
106 (aaaaaaaaaa,test)
107 (aaaaaaaaaaa,test)
108 (aaaaaaaaaaaa,test)
109 (aaaaaaaaaaaaa,test)
110 (aaaaaaaaaaaaaa,test)
111 (aaaaaaaaaaaaaaa,test)
112 (aaaaaaaaaaaaaaaa,test)
113 (aaaaaaaaaaaaaaaaa,test)
114 (aaaaaaaaaaaaaaaaaa,test)
115 (aaaaaaaaaaaaaaaaaaa,test)
116 (aaaaaaaaaaaaaaaaaaaa,test)
117 (aaaaaaaaaaaaaaaaaaaaa,test)
118 (aaaaaaaaaaaaaaaaaaaaaa,test)
119 (aaaaaaaaaaaaaaaaaaaaaaa,test)
120 (aaaaaaaaaaaaaaaaaaaaaaaa,test)
121 (aaaaaaaaaaaaaaaaaaaaaaaaa,test)

Число элементов: 25
Коллизий: 24
Максимальное число итераций одной вставки: 25
Число итераций всех вставок: 325
```

3. Некорректные данные:

```
Ошибка! Некорректное значение в строке:
"perhaps может быть"
Работа программы прервана
```

4. Пример удаления по ключу:

```
Введите ключ для удаления из таблицы или 0 для выхода: !!!
97  (ab,коллизия1_1)
98  (ba,коллизия1_2)
105 (do,делать)
106 (hello,привет)
110 (no,нет)
111 (how,как)
112 (yes,да)
116 (you,вы)
179 (тор,thor_collision)
180 (орт,orth_collision)
181 (рот,mouth_collision)
187 (очень,very)
192 (наверное,may_be)

Количество элементов: 13
Итераций удаления: 1
```

5. Пример ввода без коллизий:

```
Создана хэш-таблица:

105 (do,делать)
112 (yes,да)
116 (you,вы)
192 (наверное,may_be)

Число элементов: 4
Коллизий: 0
Максимальное число итераций одной вставки: 1
Число итераций всех вставок: 4
```

Выводы.

В ходе выполнения лабораторной работы была написана программа, реализующая хеш-таблицу для любого типа данных с открытой адресацией и возможностью удаления элементов, которая была использована в программе для работы с парами строк. Была исследована сложность алгоритмов вставки и удаления элементов из хеш-таблицы.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ.

```
#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 для демонстрации работы алгоритмов хеширования, вставки и
удаления из хеш-таблицы, ввода и вывода данных
 */
class StrWorker {
public:
    static string getFromFile(string fileName);
    string createHashTable(string input);
    string deleteElem(string input);
```

```
    bool error = false;
private:
    HashTable <string, string> hTable;
    static unsigned int hFunc(string key);
    int iter = 0;
};

/**
 * Построчное чтение входных данных из указанного файла
 */
string StrWorker::getFromFile(string fileName) {
    ifstream fin(fileName, ios::in);
    string out;
    string temp;
    while (true) {
        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
  * @return возвращает строку с информацией о числе итераций и элементами хэш-таблицы
 */
string StrWorker::createHashTable(string input) {
    hTable.setSize(SIZE);
    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 += "Размер таблицы: " + to_string(hTable.getSize());
    output += "\nЧисло элементов: ";
    output += to_string(hTable.getItemCounter());
    output += "\nКоллизий: ";
    output += to_string(maxIter - 1);
    output += "\nМаксимальное число итераций одной вставки: ";
    output += to_string(maxIter);
    output += "\nЧисло итераций всех вставок: ";
    output += to_string(sumIter);
    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 cmd;
    string data, fileName, hash, key;

    auto* worker = new StrWorker();
    cout << "Демонстрация работы с хэш-таблицами" << endl;
    cout << "**********************************" << endl;
    cout << "Загрузить данные из файла:" << endl;
     cout << "1 - с незначительным количеством коллизий\n2 - с большим
количеством коллизий\n3 - некорректные данные\n0 - выход из программы"
<< endl;
    cout << "Ваш выбор: ";
    cin >> cmd;
    cout << endl;
    switch (cmd) {
        case 1:
            fileName = "inputshort.txt";
            break;
        case 2:
            fileName = "inputlong.txt";
            break;
        case 3:
            fileName = "inputerr.txt";
            break;
        default:
            return 0;
    }
    data = worker->getFromFile(fileName);
    hash = worker->createHashTable(data);
    if (worker->error) {
        cout << hash << endl;
        cout << "Работа программы прервана" << endl;
        return 0;
    }
    cout << "Создана хэш-таблица:" << endl << endl;
    cout << hash << endl;
    while (true) {
         cout << "Введите ключ для удаления из таблицы или 0 для выхода:
";
        cin >> key;
        if (key == "0")
            break;
```

```
        hash = worker->deleteElem(key);
        cout << hash << endl;
    }
    return 0;
}
```