|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ | | | | | | | | | | | | |
| Федеральное государственное автономное образовательное учреждение  высшего образования | | | | | | | | | | | | |
| **Дальневосточный федеральный университет** | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
| **ИНСТИТУТ МАТЕМАТИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ** | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
| **Департамент программной инженерии и искусственного интеллекта** | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
| **О Т Ч Е Т** | | | | | | | | | | | | |
| по лабораторной работе №2.3  дисциплина «Фундаментальные структуры данных и алгоритмы» | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | |  |  | Студент гр. Б9122-09.03.04прогин | | | |
|  |  |  | | А. В. Поляков | |
|  | | | | | | |  |  | (подпись) | |  | |
|  | | | | | | | | | | | | |
|  | | | | | | |  |  | Руководитель | | |  |
|  | | | | | | |  |  | ст. преподаватель | | | |
|  | | | |  |  | |  |  |  |  | О.А. Крестникова | |
|  | | | |  |  | |  |  | (подпись) |  | (И.О. Фамилия) | |
|  |  |  |  | | |  |  |  |  | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
|  | | | | | | | | | | | | |
| г. Владивосток | | | | | | | | | | | | |
| 2024 | | | | | | | | | | | | |

**1 Неформальная постановка задачи**

Реализовать класс для работы с хеш-таблицей (ключ – ФИО и номер запроса, вид хеш-таблицы – динамическая, статус ячеек 0, 1, 2, метод разрешения коллизий – метод линейного пробирования с шагом k>1, хеш-функция – середина квадрата) (предметная область: Код направления, ФИО, название направления, номер заявки)

**Методы:** конструктор, деструктор, первичная хеш-функция, вторичная хеш-функция, добавление, удаление (при полном совпадении), поиск, печать.

**Входные данные:** 1) Текстовый файл, каждая из строк которого содержит данные из заданной предметной области. 2) Ключ поиска (ФИО и Номер заявки).

**Выходные данные:** 1) Номер строки входного файла, содержащей заданный ключ поиска. 2) Количество шагов поиска.

**2 Описание класса**

**Поля**:

initial\_size - начальный размер хеш-таблицы

current\_size - текущий размер хеш-таблицы

element\_count - количество элементов в хеш-таблице

collision\_count - количество коллизий

vector<Key> table - массив элементов типа Key

**Методы**:

Конструкторы:

HashTable() - конструктор хеш-таблицы  
~HashTable() - деструктор хеш таблицы

HashTable(int size) - конструктор хеш-таблицы с заданным размером

Хеш-функции:

int hash\_function(string fio, int request) - хеш-функция, использующая метод середины квадрата

Методы разрешения коллизий для разных функций:

void open\_addressing\_insert(const Key& new\_key, int index) - метод линейного пробирования для вставки элемента

void open\_addressing\_remove(const Key& key, int index) - метод линейного пробирования для удаления элемента

int open\_addressing\_find(const string& fio, int request, int index) - метод линейного пробирования для поиска элемента

Методы работы с хеш-таблицей:

void load\_from\_file(const string& path, int count) - загрузка элементов из файла в хеш-таблицу

void insert(const Key& new\_key) - вставка элемента в хеш-таблицу

void remove(const Key& key) - удаление элемента из хеш-таблицы

int find(const string& fio, int request) - поиск элемента в хеш-таблице

void resize(int new\_size) - изменение размера хеш-таблицы

void print() - вывод хеш-таблицы  
**Входные данные:**

int size - размер хеш-таблицы

string fio - полное имя

int request - номер заявки

int count - количество элементов для загрузки из файла

const string& path - путь к файлу с данными для загрузки

const Key& new\_key - новый элемент для вставки

const Key& key - элемент для удаления или поиска

**3 Текст программы**

#include <iostream>

#include <vector>

#include <string>

#include <fstream>

#include <iomanip>

// Динамическая хеш-таблица с открытой адресацией

// Статус ячейки: 0 - пусто, 1 - занято, 2 - было занято ранее

// Хеш-функция: метод середины квадрата

// Линейное пробирование с шагом k > 1

using namespace std;

struct Key {

    string direction\_name; // Наименование направления

    string full\_name;      // Полное имя

    string direction\_code; // Код направления

    int status;            // Статус ячейки

    int request\_number;    // Номер заявки

    // Конструкторы

    Key(string dirName, string fullName, string dirCode, int stat, int reqNumber)

        : direction\_name(dirName), full\_name(fullName), direction\_code(dirCode), status(stat), request\_number(reqNumber) {}

    Key() : status(0) {}

    // Операторы равенства

    bool operator==(const Key& other) const {

        return direction\_name == other.direction\_name && full\_name == other.full\_name &&

            request\_number == other.request\_number && direction\_code == other.direction\_code;

    }

    bool operator!=(const Key& other) const {

        return !(\*this == other);

    }

};

class HashTable {

public:

    int initial\_size;

    int current\_size;

    int element\_count;

    int collision\_count; // Для отслеживания количества попыток разрешения коллизий

    vector<Key> table;

    // Конструкторы

    HashTable() : initial\_size(0), current\_size(0), element\_count(0), collision\_count(0) {}

    HashTable(int size) {

        table.resize(size);

        current\_size = size;

        initial\_size = size;

        element\_count = 0;

        collision\_count = 0;

    }

    // Хеш-функция, использующая метод середины квадрата

    int hash\_function(string fio, int request) {

        int sum = 0;

        for (char ch : fio) {

            sum += ch;

        }

        sum += request;

        string str\_sum = to\_string(sum);

        int mod = 10;

        int count = 1;

        while (int(current\_size / mod) != 0) {

            mod \*= 10;

            count++;

        }

        int result = 0;

        size\_t str\_sum\_size = str\_sum.size();

        for (int i = 0; i < int(str\_sum\_size / count); i++) {

            result += stoi(str\_sum.substr(i \* count, count));

        }

        result \*= result;

        string result\_str = to\_string(result);

        return stoi(result\_str.substr(result\_str.size() / 2, count));

    }

    // Получение индекса с использованием линейного пробирования

    int get\_index(string fio, int request, int probe\_count, int step = 2) {

        return (hash\_function(fio, request) + probe\_count \* step) % current\_size;

    }

    // Загрузка элементов из файла в хеш-таблицу

    void load\_from\_file(const string& path, int count) {

        ifstream file;

        string buffer;

        int elements\_loaded = 0;

        file.open(path);

        if (file.is\_open()) {

            while (!file.eof() && (elements\_loaded <= count)) {

                Key new\_key;

                file >> buffer;

                new\_key.direction\_code = buffer;

                file >> buffer;

                new\_key.full\_name = buffer;

                file >> buffer;

                new\_key.full\_name.append(" ").append(buffer);

                file >> buffer;

                new\_key.full\_name.append(" ").append(buffer);

                file >> buffer;

                new\_key.direction\_name = buffer;

                file >> buffer;

                new\_key.request\_number = stoi(buffer);

                new\_key.status = 0;

                elements\_loaded++;

                insert(new\_key);

            }

        }

        file.close();

    }

    // Вставка элемента в хеш-таблицу

    void insert(const Key& new\_key) {

        if (element\_count >= 0.8 \* current\_size) {

            resize(current\_size \* 2);

        }

        int index = hash\_function(new\_key.full\_name, new\_key.request\_number) % current\_size;

        if (table[index].status == 0) {

            table[index] = new\_key;

            table[index].status = 1;

            element\_count++;

        } else {

            open\_addressing\_insert(new\_key, index);

        }

    }

    // Разрешение коллизий методом открытой адресации

    void open\_addressing\_insert(const Key& new\_key, int index) {

        int probe\_count = 0;

        while (table[index].status == 1) {

            if (table[index] == new\_key) {

                return; // Если элемент уже существует, не вставляем его

            }

            probe\_count++;

            index = get\_index(new\_key.full\_name, new\_key.request\_number, probe\_count);

            if (probe\_count >= current\_size) {

                resize(current\_size \* 2);

                insert(new\_key);

                return;

            }

        }

        // Проверка на уникальность при статусе 2

        while (table[index].status == 2) {

            int check\_index = index;

            int check\_probe\_count = probe\_count;

            while (table[check\_index].status != 0) {

                if (table[check\_index].status == 1 && table[check\_index] == new\_key) {

                    return; // Повторяющийся

                }

                check\_probe\_count++;

                check\_index = get\_index(new\_key.full\_name, new\_key.request\_number, check\_probe\_count);

                if (check\_probe\_count >= current\_size) {

                    resize(current\_size \* 2);

                    insert(new\_key);

                    return;

                }

            }

            break;

        }

        table[index] = new\_key;

        table[index].status = 1;

        element\_count++;

    }

    // Удаление элемента

    void remove(const Key& key) {

        int index = hash\_function(key.full\_name, key.request\_number) % current\_size;

        if (table[index].status == 1 && table[index] == key) {

            table[index].status = 2;

            element\_count--;

        } else {

            open\_addressing\_remove(key, index);

        }

        if (element\_count <= 0.2 \* current\_size && current\_size / 2 >= initial\_size) {

            resize(current\_size / 2);

        }

    }

    void open\_addressing\_remove(const Key& key, int index) {

        int probe\_count = 0;

        while (table[index].status != 0) {

            if (table[index] == key && table[index].status == 1) {

                table[index].status = 2;

                element\_count--;

                return;

            }

            probe\_count++;

            index = get\_index(key.full\_name, key.request\_number, probe\_count);

            if (probe\_count >= current\_size) {

                return; // Элемент не найден

            }

        }

    }

    // Поиск элемента

    int find(const string& fio, int request) {

        int probe\_count = 0;

        int index = hash\_function(fio, request) % current\_size;

        if (table[index].status == 1 && table[index].full\_name == fio && table[index].request\_number == request) {

            return probe\_count;

        } else {

            return open\_addressing\_find(fio, request, index);

        }

    }

    int open\_addressing\_find(const string& fio, int request, int index) {

        int probe\_count = 0;

        while (table[index].status != 0) {

            if (table[index].full\_name == fio && table[index].request\_number == request && table[index].status == 1) {

                return probe\_count;

            }

            probe\_count++;

            index = get\_index(fio, request, probe\_count);

            if (probe\_count >= current\_size) {

                return -1; // Элемент не найден

            }

        }

        return -1; // Элемент не найден

    }

    // Изменение размера хеш-таблицы

    void resize(int new\_size) {

        vector<Key> old\_table = table;

        table.clear();

        table.resize(new\_size);

        current\_size = new\_size;

        element\_count = 0;

        for (const auto& key : old\_table) {

            if (key.status == 1) {

                insert(key);

            }

        }

    }

    // Вывод хеш-таблицы

    void print() {

        int count = 0;

        for (int i = 0; i < current\_size; i++) {

            if (table[i].status != 0) {

                count++;

                cout << setw(2) << i << setw(10) << table[i].request\_number

                    << setw(35) << table[i].full\_name << setw(14) << table[i].direction\_code

                    << setw(30) << table[i].direction\_name << setw(5) << get\_index(table[i].full\_name, table[i].request\_number, 0)

                    << setw(5) << get\_index(table[i].full\_name, table[i].request\_number, 1)

                    << setw(15) << "Статус: " << table[i].status << "\n";

            }

        }

        cout << element\_count << "               " << current\_size << "\n";

    }

};

int main() {

    HashTable hashTable(13);

    hashTable.load\_from\_file("C:/Users/User/Desktop/FEFU/Second\_course/FDSA/Hash\_Table/test1.txt", 1);

    hashTable.print();

    int index;

    hashTable.print();

    while (hashTable.element\_count != 0) {

        cin >> index;

        if (index == -1) {

            break;

        }

        hashTable.remove(hashTable.table[index]);

        hashTable.print();

    }

    hashTable.print();

    while (hashTable.element\_count != 0) {

        cin >> index;

        if (index == -1) {

            break;

        }

        if (hashTable.find(hashTable.table[index].full\_name, hashTable.table[index].request\_number) == -1) {

            cout << "Student not found." << "\n";

        } else {

            cout << "Student found. Probe count: " << hashTable.find(hashTable.table[index].full\_name, hashTable.table[index].request\_number) << "\n";

        }

        hashTable.print();

    }

    hashTable.print();

    return 0;

}

**4 Тесты**

|  |  |
| --- | --- |
| **Описание тестовой ситуации** | **Тестирование** |
| Добавление при коллизии, когда дальше при разрешении встречен статус 0. | Вход:  Вставляем:  Выход: |
| Добавление при коллизии, когда дальше встречен опять статус 1. | Вход:  Вставляем:    Выход: |
| Добавление при коллизии, когда после разрешения встречен статус 2, и вставляемого элемента нет дальше в таблице. | Вход:  Вставляем:  Выход: |
| Проверка уникальности ключа (когда после встреченного статуса 2 есть элемент с таким же ключом) | Вход:  Вставляем:  Выход: |
| Расширение таблицы при добавлении, когда заполненность 80% | Вход:  Вставляем:  Выход: |
| Расширение таблицы в 2 раза при добавлении, когда количество попыток вставки больше размера таблицы. (ДЛЯ УДОБСТВА ПРОВЕРКИ ПОСТАВИЛ ШАГ 13 ПРИ РАЗРЕШЕНИИ КОЛЛИЗИИ и размер хеш-таблицы 13) | Вход:  Вставляем:  Выход: |
| Добавление – без коллизии | Вход:  Вставляем:  Выход: |
| Поиск несуществующего | Вход:  Ищем:  Выход: |
| Поиск существующего | Вход:  Ищем:  Выход: |
| Поиск при коллизии | Вход:  Ищем:  Выход: |
| Удаление существующего без коллизии | Вход:  Удаляем:  Выход: |
| Удаление при коллизии | Вход:  Удаляем:  Выход: |
| Удаление при одинаковых ключах, но разными значениями, ассоциированными с ними | Вход:  Удаляем: (2757 Lee Maxim Alexandrovich B9999 menegment) Выход: |
| Уменьшение таблицы в 2 раза при удалении, когда заполненность таблицы становится 20% | Вход: таблица, заполненная на 26%  Удаляем:  Выход: |
| Уменьшение таблицы в 2 раза при удалении, когда заполненность таблицы становится 20% не происходит, когда размер таблицы равен размеру при ее инициализации. | Вход: таблица с инициаллизированным размером 13. Заполнена на 25%  Удаляем:  Выход: таблица заполненная на 18% |