**Российская федерация  
Ханты-Мансийский автономный округ – Югра  
Департамент образования и науки  
Сургутский государственный университет ХМАО**

Политехнический институт  
Кафедра Автоматики и компьютерных систем

**Пояснительная записка**  
к курсовому проекту   
по дисциплине «Алгоритмы и структуры данных»

Выполнил: студент группы 609-31

Гаврилов Е.Е.

Принял: старший преподаватель кафедры АиКС

Назаров Е.В.

Сургут

2025 г.

**Задание**

Реализовать класс «Хеш-таблица».

1. Провести анализ предметной области возможного применения проектируемого класса.

2. Провести анализ функциональности проектируемого класса.

3. Разработать интерфейс класса.

4. В соответствии с разработанным интерфейсом спроектировать тестовое приложение.

5. Выполнить проектирование класса, обоснованно выбирая: необходимые поля класса; методы класса, включив в обязательном порядке операции вставки, удаления, изменения, поиска отдельных элементов, входящих в класс.

6. Провести проектирование алгоритмов, лежащих в основе разрабатываемых методов.

7. Реализовать полученное проектное решение.

8. Реализовать тестовое приложение и провести тестирование разработанного и реализованного класса.

9. Провести исследование одной из операций (вставка, удаление, изменение, поиск) над элементами. Построить зависимость времени выполнения операции от числа элементов, над которыми она выполняется.

10. Оценить асимптотическую сложность реализованных алгоритмов вставки, удаления и поиска элементов класса.

**Аннотация**

В данной пояснительной записке описаны этапы написания курсового проекта по дисциплине «Алгоритмы и структуры данных» в процессе которого будет написан класс «Хеш-таблица».

В разделе «Анализ» проведено описание предметной области, обзор существующих аналогов, сформулированы требования к разрабатываемой программе.

В разделе «Проектирование» описаны структуры данных, архитектура программы, основные алгоритмы.

В разделе «Кодирование» представлена реализация алгоритмов и структур данных, приведены листинги.

В разделе «Тестирование» составлен набор тестов, проведено тестирование для выявления ошибок в программном продукте.

В разделе «Исследование» описано исследование операции поиска, изучена зависимость времени работы алгоритма от количества элементов.

В разделе «Сопровождение» описаны особенности запуска и рекомендации для пользователя.

**Содержание**

[1. Введение 6](#_Toc200917200)

[2. Анализ 7](#_Toc200917201)

[2.1 Анализ предметной области 7](#_Toc200917202)

[2.2 Обзор существующих решений 7](#_Toc200917203)

[2.2.1 Класс std::unordered\_map в стандартной библиотеке C++ 7](#_Toc200917204)

[2.2.1 Класс HashMap в стандартной библиотеке Java 8](#_Toc200917205)

[2.3 Требования к разработанной программе 8](#_Toc200917206)

[3. Проектирование 9](#_Toc200917207)

[3.1 Формальное описание задачи 9](#_Toc200917208)

[3.2 Структура классов 10](#_Toc200917209)

[3.3 Разработка структур данных 10](#_Toc200917210)

[3.4 Разработка алгоритмов, задаваемых спецификациями 10](#_Toc200917211)

[3.4.1 Инициализация хеш-таблицы 10](#_Toc200917212)

[3.4.2 Вставка, удаление, поиск, проверка на наличие элементов. 10](#_Toc200917213)

[3.4.3 Увеличение размера таблицы 10](#_Toc200917214)

[3.4.4 Прочее 11](#_Toc200917215)

[3.4.5 Тестовое приложение 11](#_Toc200917216)

[4. Кодирование 12](#_Toc200917217)

[4.1 Определение языка, вспомогательных средств и типов данных. 12](#_Toc200917218)

[4.2 Константы и перечисления 12](#_Toc200917219)

[4.3 Описание класса пары ключ-значение 12](#_Toc200917220)

[4.4 Класс-итератор 13](#_Toc200917221)

[4.5 Основной класс хеш-таблицы 14](#_Toc200917222)

[5. Тестирование 23](#_Toc200917223)

[5.1 Планирование тестирования 23](#_Toc200917224)

[5.2 Разработка тестов 23](#_Toc200917225)

[5.2.1 Вставка 23](#_Toc200917226)

[5.2.2 Удаление 23](#_Toc200917227)

[5.2.3 Поиск 23](#_Toc200917228)

[5.2.4 Итерация 23](#_Toc200917229)

[5.2.5 Копирование 23](#_Toc200917230)

[5.2.6 Отчистка 23](#_Toc200917231)

[5.2.7 Тестовая программа 23](#_Toc200917232)

[5.2.8 Тестирование ПО 24](#_Toc200917233)

[5.2.9 Вставка 24](#_Toc200917234)

[5.2.10 Удаление 24](#_Toc200917235)

[5.2.11 Поиск 24](#_Toc200917236)

[5.2.12 Итерация 24](#_Toc200917237)

[5.2.13 Копирование 24](#_Toc200917238)

[5.2.14 Отчистка 24](#_Toc200917239)

[5.2.15 Тестовая программа 24](#_Toc200917240)

[5.2.16 Работа с памятью 24](#_Toc200917241)

[5.3 Итоги тестирования 24](#_Toc200917242)

[6. Исследование 25](#_Toc200917243)

[6.1 Исследование зависимости времени работы алгоритма 25](#_Toc200917244)

[6.2 Исследование асимптотической сложности 26](#_Toc200917245)

[7. Сопровождение 27](#_Toc200917246)

[7.1 Требование к программно-аппаратной платформе 27](#_Toc200917247)

[7.2 Инструкция по установке и использованию 27](#_Toc200917248)

[7.3 Сведенья о недостатке программного продукта. 27](#_Toc200917249)

[Список используемых источников 29](#_Toc200917250)

[Приложение 1. Листинг заголовочного файла hashmap.h 30](#_Toc200917252)

[Приложение 2. Листинг файла с реализацией hashmap.tpp 33](#_Toc200917253)

[Приложение 3. Листинг тестовой программы 38](#_Toc200917254)

1. **Введение**

Хеш-таблица – структура данных, реализующая интерфейс ассоциативного массива, то есть позволяет хранить пары ключ-значение и выполнять три операции: добавление новой пары, удаление и поиск по ключу. Важное свойство хеш-таблиц состоит в том, что все три операции в среднем выполняются за время O(1). При этом не гарантируется, что время выполнения каждой из операций само по себе мало. [1,2]

1. **Анализ** 
   1. Анализ предметной области

Хеш-функция (англ. hash function от hash — «превращать в фарш», «мешанина»), или функция свёртки — функция, преобразующая массив входных данных произвольного размера в выходную битовую строку определённого (установленного) размера в соответствии с определённым алгоритмом. Преобразование, выполняемое хеш-функцией, называется хешированием. Исходные (входные) данные называются входным массивом, «ключом», «сообщением». [2]

Хеш-таблица является массивом, элементы которого есть пары (хеш-таблица с открытой адресацией) или списки пар (хеш-таблица со списками). Выполнение операции в хеш-таблице начинается с вычисления хеш-функции от ключа.   
Получающееся хеш-значение играет роль индекса в массиве. Затем выполняемая операция (добавление, удаление или поиск) перенаправляется объекту, который хранится в соответствующей ячейке массива. [1,3,4]

Число хранимых элементов, делённое на размер массива (число возможных значений хеш-функции), называется коэффициентом заполнения хеш-таблицы (load factor) и является важным параметром, от которого зависит среднее время выполнения операций.

Ситуация, когда для различных ключей получается одно и то же хеш-значение, называется коллизией.

Существует несколько способов разрешения коллизий:

1. Метод списков. Каждая ячейка массива является связным списком пар ключ-значение, соответствующих одному и тому же хеш-значению ключа. Коллизии в таком случае приводят к тому что появляются списки длинной более одного элемента.
2. Метод открытой адресации. В массиве хранятся сами пары ключ-значение. В случае коллизии необходимо проверить ячейки массива в некотором порядке, до тех пор, пока не будет найдена необходимая ячейка.

Хеш-функция, использующаяся для хеш-таблицы, должна обладать следующими свойствами:

1. Детерминированность. Один и тот же ключ всегда должен быть преобразован в один и тот же хеш.
2. Равномерность распределения. Индексы должны быть распределены равномерно, чтобы минимизировать количество незаполненных ячеек массива и коллизий. Добиться действительно равномерного распределения на случайных данных нельзя, поэтому хеш-функция должна хотя бы стремиться к равномерному распределению.
   1. Обзор существующих решений
      1. Класс std::unordered\_map в стандартной библиотеке C++

Класс std::unordered\_map – это ассоциативный контейнер, является частью стандартной библиотеки шаблонов (STL) языка C++ (начиная со стандарта C++11). Класс реализует метод списков для разрешения коллизий. Порядок элементов не соблюдается. Дубликаты ключей запрещены. Как и прочие стандартные контейнеры, позволяет работать с любыми типами данных (как ключей, так и значений), поддерживает работу с итераторами, позволяет управлять выделяемой памятью с помощью аллокаторов. Размер таблицы меняется автоматически. [5,6]

* + 1. Класс HashMap в стандартной библиотеке Java

Класс HashMap реализует интерфейс Map на основе хеш-таблицы. Для разрешения коллизий использует метод списков, однако вместо списков использует бинарные деревья (начиная с Java8). Порядок элементов не соблюдается. Дубликаты ключей запрещены. Позволяет работать с любыми типами данных (ключей и значений). Позволяет задавать изначальную емкость таблицы и коэффициент загрузки, начиная с которого таблица будет перестроена. Как и все стандартные коллекции, поддерживает работу с итераторами.[7]

* 1. Требования к разработанной программе

Функциональные требования:

- Возможность добавлять, удалять пары ключ-значение.

- Возможность поиска значения по ключу.

- Поддержка различных типов данных.

- Время поиска в среднем должно быть O(1).

- Возможность итерироваться по парам ключ-значение.

1. **Проектирование**
   1. Формальное описание задачи

Ввиду более простой и понятной реализации, при создании таблицы будет использован метод списков.

Хеш таблица представляет собой массив, каждый элемент которого является списком (или указателем на список). В списке хранятся те пары ключ-значения, хеш-коды ключей которых совпадают.

В данной реализации структура будет немного изменена. Теперь связный список будет один. Для таблицы, содержащей n ячеек, список изначально будет состоять из n меток «начало подсписка» и 1 метки «конец списка».

Массив размера n будет содержать указатели на метки типа «начало подсписка».

Таким образом, при обращении к очередному элементу, будет вычислен его хеш-код и соответствующая позиция в массиве. Далее произойдет обращение к соответствующей метке «начало подсписка» в списке пар. После нее следует перебрать все пары ключ-значение, пока не обнаружится пара с требуемым ключом. Если в процессе перебора будет обнаружена метка «начало подсписка» или «конец списка», нужного ключа в таблице нет.

Пример такой структуры с 4 ячейками представлен на рисунке 3.1.

Сверху представлен связный список. Серыми квадратами показаны метки «начало подсписка», синим «конец списка», белыми пары ключ-значение, несущие полезную информацию.

В данном примере таблице содержится 3 пары ключ-значения. У двух из них хеш-код равен 2, у одного из них – 1.

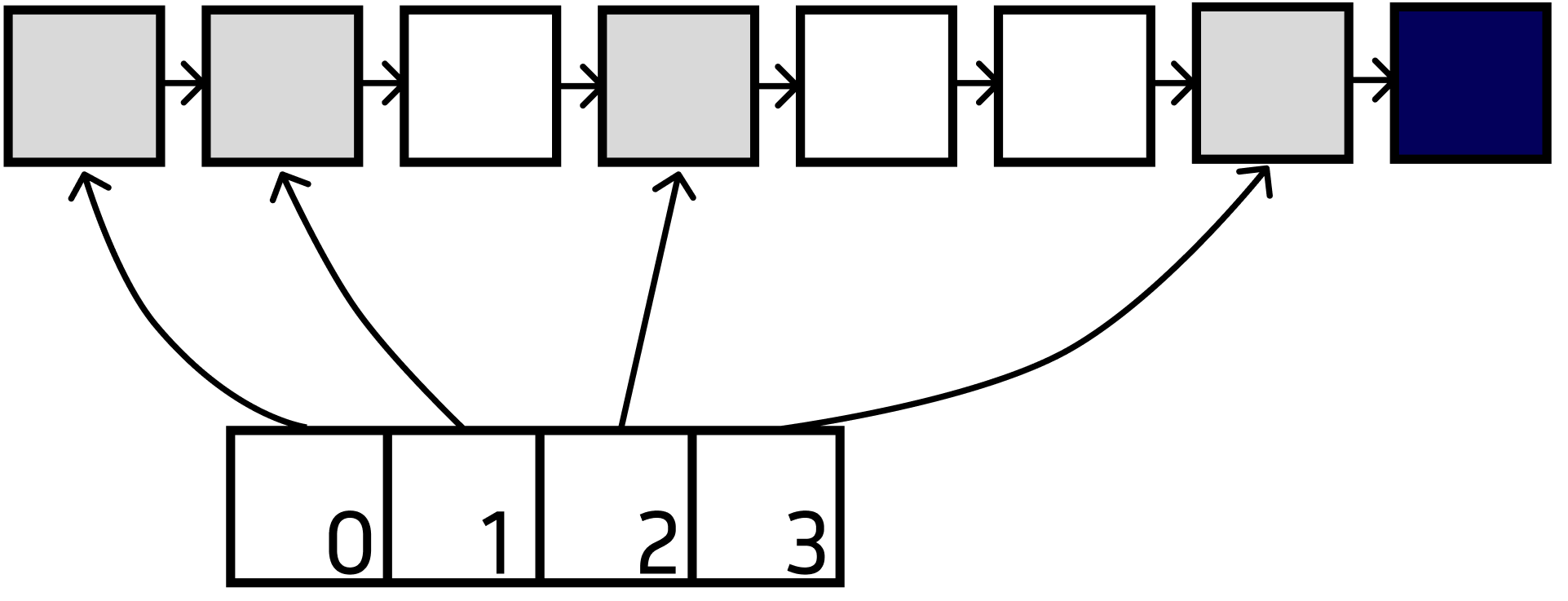


Рисунок 3.1. Структура хеш-таблицы

С возрастанием количества хранимой информации, длинна подлистов будет увеличиваться, а время поиска уменьшаться. Для сохранения малого времени, при достижении предела наполненности таблицы, ее необходимо перестроить с большим размером массива. Размер таблицы для наибольшей эффективности должен быть простым числом. При каждом увеличении таблицы, новый размер должен быть простым числом, которое не меньше чем в 2 раза больше чем предыдущий размер таблицы.

* 1. Структура классов
* Пара ключ-значение – класс, содержащий поля ключ и значения любых типов, а также тип пары. Данный класс будет использован или в качестве непосредственного хранилища данных, или в качестве метки. Такие фиктивные пары полезной информации не содержат.
* Хеш-Таблица – основной класс, хранящий набор пар ключ-значение, и содержащий все методы для работы с данными.
* Итератор – подкласс класса Хеш-Таблица для итерации по множеству пар ключ-значение.
  1. Разработка структур данных

Основные используемые структуры:

* Связный список, хранящий пары. Пары могут содержать полезные данные, либо быть помечены как «начало подсписка» или «конец списка».
* Массив указателей на начала подсписков.
  1. Разработка алгоритмов, задаваемых спецификациями
     1. Инициализация хеш-таблицы

Создается список, он заполняется метками «начало подсписка» и меткой «конец списка», создается массив размера размер листа -1, он заполняется указателями на метки.

* + 1. Вставка, удаление, поиск, проверка на наличие элементов.

Все эти действия требуют одинаковых действий по поиску элементов, поэтому объединены в один пункт.

Вычислить хеш-код ключа, перейти по указателю в массиве к узлу списка. Пока не будет обнаружен нужный ключ или начало следующего подсписка пройтись по текущему подсписку. При нахождении ключа соответственно изменить, значение, удалить, вернуть ссылку на элемент или вернуть истину. При достижении начала следующего подсписка или конца списка, соответственно, вставить новую пару, ничего не делать, вставить новую пару и инициализировать значением по умолчанию или вернуть ложь. При превышении коэффициента заполнения таблицы порогового значения, она должна быть перестроена.

* + 1. Увеличение размера таблицы

Создаются новый список и массив нового размера. Как и при инициализации, список заполняется метками, а массив – указателями на метки. Для всех пар значений в таблице высчитывается новый хеш и ими заполняется новый список. Старые массив и список удаляются.

* + 1. Прочее

Кроме того, потребуются следующие вспомогательные функции: проверки на пустоту, возвращение количества ячеек таблицы, пар, содержащихся в таблице.

* + 1. Тестовое приложение

Тестовое приложение представляет собой хеш-таблицу и интерфейс, который позволяет проводить следующие операции:

* Вставка элемента по ключу
* Удаление элемента по ключу
* Проверка наличия элемента по ключу
* Вывод количества элементов таблицы
* Вывод размера таблицы
* Вывод всех элементов таблицы
* Отчистка таблицы
* Вычисление хеш-хода для ключа с учетом размера таблицы (т.е. индекс, который будет присвоен паре с данным ключом
* Выход из программы

1. **Кодирование**
   1. Определение языка, вспомогательных средств и типов данных.

Программа реализована на языке C++ стандарта C++20 с использованием стандартных библиотек языка vector, iostream, list.

Типы данных: std::list и std:: vector из стандартной библиотеки языка.

* 1. Константы и перечисления

Для определения размера таблицы будет использован константный массив из 24 элементов size\_t size\_table.

Максимальный индекс хранится в константной переменной size\_t max\_size\_table\_index.

Для определения типа пары определено перечисление pair\_type, принимающее значение REGULAR (обычная пара), BUCKET\_BEGIN (метка начала подсписка), LIST\_END (конец списка).

Листинг 4.1 Константы и перечисления

    const size\_t size\_table[] =

        {

            5ul, 53ul, 97ul, 193ul, 389ul,

            769ul, 1543ul, 3079ul, 6151ul, 12289ul,

            24593ul, 49157ul, 98317ul, 196613ul, 393241ul,

            786433ul, 1572869ul, 3145739ul, 6291469ul, 12582917ul,

            25165843ul, 50331653ul, 100663319ul, 201326611ul, 402653189ul,

            805306457ul, 1610612741ul, 3221225473ul, 4294967291ul};

    const size\_t max\_size\_table\_index = 23;

    enum class pair\_type

    {

        REGULAR,

        BUCKET\_BEGIN,

        LIST\_END

    };

* 1. Описание класса пары ключ-значение

Шаблонный класс pair<typename key\_t, typename value\_t> содержит поля ключ и значение соответственно типов key\_t и value\_t, а так же поле перечисления типа pair\_type. Класс содержит только конструктор и методы получения данных полей ключа и значения.

Листинг 4.2 Класс-пара

template <typename key\_t, typename value\_t>

    class pair

    {

        friend class hash\_map<key\_t, value\_t>;

    public:

        pair(const key\_t &key, const value\_t &value, pair\_type type) : \_key(key), \_value(value), type(type) {}

        key\_t key() { return \_key; }

        value\_t value() { return \_value; }

        pair &operator=(const pair &p)

        {

            \_key = p.\_key;

            \_value = p.\_value;

            type = p.type;

            return \*this;

        }

    private:

        key\_t \_key;

        value\_t \_value;

        pair\_type type;

    };

* 1. Класс-итератор

Класс-итератор является оберткой над итератором списка пар стандартной библиотеки. Данная обертка позволяет пропускать незначащие пары в списке и итерироваться только по парам с полезными данными. Предполагается, что «концом» будет является пара с меткой «Конец Списка». Для итератора определены стандартные действия: инкремент, разыменование, значение по указателю, сравнение с другим итератором. Данный итератор позволит двигаться только вперед, так как хеш-таблица является неупорядоченной структурой, этого достаточно. При любом изменении таблицы итератор может стать недействительным.

Листинг 4.3 Класс-итератор

struct iterator

        {

            friend class hash\_map<key\_t, value\_t>;

        private:

            std::list<pair<key\_t, value\_t>>::iterator iter;

            void skip()

            {

                while ((\*iter).type == pair\_type::BUCKET\_BEGIN)

                {

                    iter++;

                }

            }

        public:

            iterator(std::list<pair<key\_t, value\_t>>::iterator iter) : iter(iter) {}

            iterator &operator++()

            {

                ++iter;

                skip();

                return \*this;

            }

            pair<key\_t, value\_t> \*operator->()

            {

                return &(\*iter);

            }

            pair<key\_t, value\_t> &operator\*() const

            {

                return \*iter;

            }

            bool operator==(const iterator &other)

            {

                return iter == other.iter;

            }

            bool operator!=(const iterator &other)

            {

                return iter != other.iter;

            }

        };

* 1. Основной класс хеш-таблицы

Данный класс содержит приватные поля индекса в таблице размеров, текущего количества элементов, максимальный фактор наполненности таблицы, а также список элементов и вектор итераторов на элементы списка. Метод для перестройки таблицы приватный.

Листинг 4.6 Описание основного класса

template <typename key\_t, typename value\_t>

    class hash\_map

    {

    private:

        const double max\_load\_factor = 0.75;

        size\_t size\_table\_index;

        std::list<pair<key\_t, value\_t>> element\_list;

        std::vector<decltype(element\_list.begin())> iterators;

        void resize();

        size\_t \_size = 0;

    public:

        struct iterator

        {

            friend class hash\_map<key\_t, value\_t>;

        private:

            std::list<pair<key\_t, value\_t>>::iterator iter;

            void skip()

            {

                while ((\*iter).type == pair\_type::BUCKET\_BEGIN)

                {

                    iter++;

                }

            }

        public:

        hash\_map();

        void put(const key\_t &, const value\_t &);

        void remove(const key\_t &);

        bool contains(const key\_t &);

        bool empty();

        value\_t &get(const key\_t &);

        value\_t &operator[](const key\_t &);

        size\_t total\_buckets();

        size\_t size();

        void clear();

        iterator begin();

        iterator end();

        void print();

        static size\_t hash(const key\_t &, size\_t);

        hash\_map<key\_t, value\_t> &operator=(const hash\_map<key\_t, value\_t> &);

    };

Конструктор основного класса

Инициализируется вектор минимального (согласно таблице) размера, список заполняется парами типа BUCKET\_BEGIN, в вектор записываются итераторы этих пар. В конце добавляется пара типа LIST\_END.

Листинг 4.7 Конструктор hash\_map

template <typename key\_t, typename value\_t>

hash\_map<key\_t, value\_t>::hash\_map() : size\_table\_index(0)

{

    element\_list = std::list<pair<key\_t, value\_t>>();

    iterators = std::vector<decltype(element\_list.begin())>(total\_buckets() + 1);

    for (size\_t i = 0; i < total\_buckets(); i++)

    {

        element\_list.push\_back(pair(key\_t(), value\_t(), pair\_type::BUCKET\_BEGIN));

        iterators[i] = std::prev(element\_list.end());

    }

    element\_list.push\_back(pair(key\_t(), value\_t(), pair\_type::LIST\_END));

}

Хеш-функция

В качестве хеш-функции используется std::hash из стандартной библиотеки C++. Для того чтобы класс мог быть ключом в хеш-таблице, для него должен быть определена шаблонная структура std::hash с перегруженным оператором «скобки». Остаток от деления на размер является хеш-кодом для работы с таблицей.

Листинг 4.8 Метод hash

template <typename key\_t, typename value\_t>

size\_t

hash\_map<key\_t, value\_t>::hash(const key\_t &key, size\_t \_\_size)

{

    auto hash = std::hash<key\_t>();

    return hash(key) % \_\_size;

}

Проверка наличия ключа в таблице

Обход подлиста, пока не будет найден нужный элемент, либо пока не начнется следующий подлист.

Листинг 4.9 Метод проверки содержания ключа

template <typename key\_t, typename value\_t>

bool hash\_map<key\_t, value\_t>::contains(const key\_t &key)

{

    auto i = iterators[hash(key, total\_buckets())];

    for (i++; i->type == pair\_type::REGULAR; i++)

        if (i->\_key == key)

            return true;

    return false;

}

Вставка

Аналогично методу проверки наличия ключа, обход подлиста, если пара с заданным ключом найдена, значение будет изменено, в ином случае будет вставлена новая пара.

Листинг 4.10 Метод вставки

template <typename key\_t, typename value\_t>

void hash\_map<key\_t, value\_t>::put(const key\_t &key, const value\_t &val)

{

    auto i = iterators[hash(key, total\_buckets())];

    for (i++; i->type == pair\_type::REGULAR; i++)

        if (i->\_key == key)

        {

            i->\_value = val;

            return;

        }

    element\_list.insert(i, pair(key, val, pair\_type::REGULAR));

    \_size++;

    if (static\_cast<double>(size()) / total\_buckets() >= max\_load\_factor)

    {

        resize();

    }

}

Удаление работает аналогично проверки на наличие ключа.

Листинг 4.11 Метод удаления

template <typename key\_t, typename value\_t>

void hash\_map<key\_t, value\_t>::remove(const key\_t &key)

{

    auto i = iterators[hash(key, total\_buckets())];

    for (i++; i->type == pair\_type::REGULAR; i++)

        if (i->\_key == key)

        {

            element\_list.erase(i);

            \_size--;

            return;

        }

}

Поиск работает аналогично проверке на наличие ключа. Так как метод возвращает ссылку, в случае обращения к несуществующему ключу, этот элемент с ключом будет создан и инициализирован значением по умолчанию. Также перегружен оператор [] который позволит выполнить обращение по ключу.

Листинг 4.12 Метод поиска по ключу

template <typename key\_t, typename value\_t>

value\_t &hash\_map<key\_t, value\_t>::get(const key\_t &key)

{

    auto i = iterators[hash(key, total\_buckets())];

    for (i++; i->type == pair\_type::REGULAR; i++)

        if (i->\_key == key)

        {

            return (\*i).\_value;

        }

    element\_list.insert(i, pair(key, value\_t(), pair\_type::REGULAR));

    return (\*(--i)).\_value;

}

template <typename key\_t, typename value\_t>

value\_t &hash\_map<key\_t, value\_t>::operator[](const key\_t &key)

{

    return get(key);

}

Отчистка таблицы

Метод отчистки пересоздает список и вектор аналогично конструктору.

Листинг 4.13 Метод отчистки таблицы

template <typename key\_t, typename value\_t>

void hash\_map<key\_t, value\_t>::clear()

{

    size\_table\_index=0;

    element\_list = std::list<pair<key\_t, value\_t>>();

    iterators = std::vector<decltype(element\_list.begin())>(total\_buckets() + 1);

    size\_t i;

    for (i = 0; i < total\_buckets(); i++)

    {

        element\_list.push\_back(pair(key\_t(), value\_t(), pair\_type::BUCKET\_BEGIN));

    }

    element\_list.push\_back(pair(key\_t(), value\_t(), pair\_type::LIST\_END));

    i = 0;

    for (auto iter = element\_list.begin(); iter != element\_list.end(); i++, iter++)

    {

        iterators[i] = iter;

    }

    \_size=0;

}

Листинг 4.14 Переопределение оператора присвоения

template <typename key\_t, typename value\_t>

hash\_map<key\_t, value\_t> &hash\_map<key\_t, value\_t>::operator=(const hash\_map<key\_t, value\_t> &map)

{

    element\_list = std::list<pair<key\_t, value\_t>>();

    size\_table\_index = map.size\_table\_index;

    iterators = std::vector<decltype(element\_list.begin())>(map.iterators.size());

    \_size = map.\_size;

    size\_t i = 0;

    for (auto iter = map.element\_list.begin(); iter != map.element\_list.end(); iter++)

    {

        element\_list.push\_back(\*iter);

        if (element\_list.back().type == pair\_type::BUCKET\_BEGIN)

        {

            iterators[i++] = std::prev(element\_list.end());

        }

    }

    return \*this;

}

Итераторы

Аналогично контейнерам из стандартной библиотеки C++, класс hash\_map содержит методы begin и end которые возвращают итераторы соответственно на первый элемент в контейнере и первый после последнего. Данный итератор позволит передвигаться только вперед. Порядок вставки элементов не соблюдается.

Листинг 4.15 Методы, возвращающие итераторы

template <typename key\_t, typename value\_t>

hash\_map<key\_t, value\_t>::iterator hash\_map<key\_t, value\_t>::begin()

{

    auto res = iterator(element\_list.begin());

    res.skip();

    return res;

}

template <typename key\_t, typename value\_t>

hash\_map<key\_t, value\_t>::iterator hash\_map<key\_t, value\_t>::end()

{

    return iterator(std::prev(element\_list.end()));

}

Листинг 4.16 Метод увеличения размера таблицы

template <typename key\_t, typename value\_t>

void hash\_map<key\_t, value\_t>::resize()

{

    if (size\_table\_index < max\_size\_table\_index)

    {

        size\_table\_index++;

    }

    else

        return;

    auto buf\_element\_list = element\_list;

    element\_list.clear();

    iterators = std::vector<decltype(element\_list.begin())>(total\_buckets() + 1);

    for (size\_t i = 0; i < total\_buckets(); i++)

    {

        element\_list.push\_back(pair(key\_t(), value\_t(), pair\_type::BUCKET\_BEGIN));

        iterators[i] = std::prev(element\_list.end());

    }

    element\_list.push\_back(pair(key\_t(), value\_t(), pair\_type::LIST\_END));

    for (auto &j : buf\_element\_list)

    {

        if (j.type == pair\_type::REGULAR)

        {

            auto iter = iterators[hash(j.\_key, total\_buckets())];

            for (iter++; iter->type == pair\_type::REGULAR; iter++)

            {

                ;

            }

            element\_list.insert(iter, pair(j.\_key, j.\_value, pair\_type::REGULAR));

        }

    }

}

* 1. Тестовое приложение

Тестовое приложение представляет собой объект класса «хеш-таблица» и консольный интерфейс, который позволит проводить операции над хеш-таблицей (согласно п. 3.4.5)

Листинг 4.17 Тестовое приложение

#include <iostream>

#include <limits>

#include "hashmap.h"

using namespace std;

using namespace curs;

int main()

{

    int menu\_key;

    hash\_map<string, int> map;

    string key;

    int val;

    do

    {

        cout << endl;

        cout << "Хеш-Таблица" << endl

             << "Ключ: std::string" << endl

             << "Значение: int" << endl

             << "1. Вставить" << endl

             << "2. Удалить" << endl

             << "3. Проверить наличие" << endl

             << "4. Количество элементов" << endl

             << "5. Размер таблицы" << endl

             << "6. Вывести все элементы" << endl

             << "7. Отчистить" << endl

             << "8. Вычислить хеш-код" << endl

             << "0. Выход" << endl

             << ">>";

        cin >> menu\_key;

        if (cin.fail())

        {

            menu\_key = -1;

            cin.clear();

            cin.ignore(numeric\_limits<streamsize>::max(), '\n');

        }

        switch (menu\_key)

        {

        case 0:

            break;

        case 1:

            cout << "Введите ключ (std::string): ";

            cin >> key;

            cout << "Введие значение (int): ";

            cin >> val;

            if (cin.fail())

            {

                cin.clear();

                cin.ignore(numeric\_limits<streamsize>::max(), '\n');

                cout << "Значение должно быть числом" << endl;

            }

            else

            {

                map.put(key, val);

            }

            break;

        case 2:

            cout << "Введите ключ (std::string): ";

            cin >> key;

            map.remove(key);

            break;

        case 3:

            cout << "Введите ключ (std::string): ";

            cin >> key;

            if (map.contains(key))

            {

                cout << "\"" << key << "\" содержится в таблице" << endl;

            }

            else

            {

                cout << "\"" << key << "\" не содержится в таблице" << endl;

            }

            break;

        case 4:

            cout << "Количество элементов: " << map.size() << endl;

            break;

        case 5:

            cout << "Размер таблицы: " << map.total\_buckets() << endl;

            break;

        case 6:

            if (map.empty())

            {

                cout << "Таблица пуста" << endl;

            }

            for (auto i : map)

            {

                cout << "\"" << i.key() << "\": " << i.value() << endl;

            }

            break;

        case 7:

            map.clear();

            break;

        case 8:

            cout << "Введите ключ (std::string): ";

            cin >> key;

            cout << "Хеш: " << map.hash(key, map.total\_buckets()) << endl;

            break;

        default:

            cout << "Неизвестная команда" << endl;

            break;

        }

    } while (menu\_key != 0);

}

1. **Тестирование**
   1. Планирование тестирования

Тестирование должно включать в себя:

* Проверку корректной работы хеш-таблицы, выполнения всех функций
* Проверку на больших объемах данных
* Проверку корректной работы тестового приложения
* Проверка на корректную работу с памятью.
  1. Разработка тестов

Тестирование будет проводиться на таблицах с типами ключ-значение string-int. Во всех тестах ожидается корректная работа с памятью, не использование неинициализированных значений, отсутствие утечек. Проверка будет проводится консольной утилитой valgrind.

* + 1. Вставка

Вставка 10’000 пар с уникальными ключами. Ожидается что все пары будут записаны, количество элементов будет равно 10’000, размер таблицы – 24’593. Вставка 5’000 пар с ключами из первых 10’000. Ожидается, что значения будут перезаписаны, количество элементов и размер таблицы не изменится.

* + 1. Удаление

Ожидается, удаление существующих элементов удалит необходимые пары. Размер таблицы не уменьшится, количество элементов уменьшится.

* + 1. Поиск

Ожидается, что метод contains будет возвращать «истину» и «ложь» в соответствии с наличием элемента с заданным ключом. Ожидается, что метод get вернет ссылку на значение пары по ключу, которое можно будет изменить. Обращение к несуществующему элементу создаст его со значением по умолчанию.

* + 1. Итерация

Ожидается что все пары ключ-значение можно обойти с помощью цикла for-each. Порядок элементов не определен. Итерирование будет проводиться без изменений таблицы, так как при изменении итераторы становятся негодными и поведение в таком случае не определено. В случае с пустой таблицей, итераторы begin и end должны совпадать.

* + 1. Копирование

Ожидается, что копирование таблицы приведет к созданию еще одной таблицы того же размера и содержания, с тем же набором сущностей. Изменение одной таблицы не повлечет за собой изменение другой.

* + 1. Отчистка

Ожидается полная отчистка таблицы. Размер отчищенной таблицы равен 5, количество элементов равно 0.

* + 1. Тестовая программа

Ожидается корректная работа всех пунктов, адекватная реакция на ввод данных неверного типа (символов вместо чисел).

* + 1. Тестирование ПО
    2. Вставка

Было вставлено 10’000 пар чисел вида {“n”; n}, где n – целое число от 0 до 9'999, а “n” – это же число, переведенная в строковый формат. Количество элементов, размер таблицы, соответствие значений своим ключам соответствуют ожидаемым.

Вставка значений с уже существующими ключами прошла успешно.

* + 1. Удаление

Удаление 5’000 пар по ключу. 4’000 из них содержались в таблице, 1’000 нет. Удаление успешно, параметры таблицы соответствуют ожидаемым.

* + 1. Поиск

Метод contains возвращает true и false соответственно, когда ключ есть и когда ключа нет в таблице. Метод put и перегруженный оператор [] позволяет получить ссылку на значение по ключу. Если ключа не было в таблице, объект с таким ключом будет создан и ссылка будет возвращена на него. Поведение соответствует ожидаемому.

* + 1. Итерация

Итерирование происходит по списку, позволяя перебрать все пары ключ-значение, но не изменять их. Поведение совпадает с ожидаемым.

* + 1. Копирование

Создан новый объект класса, содержащий копию исходной таблицы. Изменение изначальной таблицы не влияет на копию, тест пройден.

* + 1. Отчистка

Все элементы удалены, размеры таблицы соответствуют ожиданию. Тест пройден.

* + 1. Тестовая программа

Все пункты меню работают, как и ожидалось. Ошибки, такие как ввод строк в числовые поля обрабатываются адекватно.

* + 1. Работа с памятью

Во время тестирования вышеуказанных функций утечек памяти, неинициализированных значений и прочих ошибок, связанных с памятью не выявлено.

* 1. Итоги тестирования

По итогам тестирования ошибок в функциях, алгоритмах или в работе с памятью не выявлены. Поведение всех компонентов соответствует ожидаемым.

1. **Исследование**
   1. Исследование зависимости времени работы алгоритма

Будет исследована операция поиска. В качестве этой операции будет использован метод contains, который проверят наличие элемента с заданным ключом в таблице. В таблицу будут помещены случайные числа от 0 до 999'999. Ключ будет представлять собой это же число, но в формате строки. Для всех чисел из диапазона случайных чисел будет вызван метод contains, зафиксирован результат и время работы метода. Далее для всех вызовов с результатом true и для всех вызовов с результатом false будет высчитаны средние значения для каждого количества элементов. Усредненное время поиска в зависимости от количества элементов в таблице (для успешного и неуспешного поиска) представлено в таблице 1, соответствующий график на рисунке 6.1.

Таблица 1. Зависимость времени поиска успешного и неуспешного поиска от количества элементов

|  |  |  |
| --- | --- | --- |
| Количество элементов | Время успешного поиска (нс) | Время неуспешного поиска(нс) |
| 10 | 284 | 330 |
| 100 | 339 | 388 |
| 1000 | 472 | 566 |
| 10000 | 614 | 786 |
| 100000 | 797 | 993 |
| 1000000 | 1016 | 1139 |

Рисунок 6.1. Зависимость времени поиска успешного и неуспешного поиска от количества элементов

Как следует из полученных данных, увеличение количества элементов в таблице влечет за собой увеличение времени поиска, как успешного, так и неуспешного. Однако размер этого увеличения крайне мало. С увеличением количества элементов в 1’000’000 раз последовало увеличение времени на 800 нс. Это объясняется появлением в таблице коллизий, в случае которых необходимо пройтись по нескольким элементам.

* 1. Исследование асимптотической сложности

Асимптотическая сложность работы алгоритмов вставки, удаления и поиска непосредственно зависят от свойств ключей и асимптотической сложности алгоритма соответствующей хеш-фунции. Так, например, для целочисленного типа ключей сложность хеш-функции будет равна O(1), а для строчного O(n), так как значение функции зависит от каждого символа в строке.

Асимптотическая сложность алгоритмов вставки, удаления и поиска, без учета хеш-функции в лучшем случае равны O(1). Лучший случай – случай, в котором хеш-значения ключей не повторяются, количество пар в каждом из подсписков таким образом не больше 1.

Худший случай – случай, в котором все ключи имеют одинаковый хеш-код, а нужный элемент не существует, или находится в самом конце подсписка, сложность всех операций таким образом будет O(n).

Кроме того, при превышении коэффициента заполнения (при вставке нового элемента), таблица перестраивается, сложность этой перестройки O(n).

Таким образом, асимптотическая сложность алгоритмов вставки, удаления и поиска равна O(1+ f(x)) в лучшем случае и O(n+f(x)) в худшем, где f(x) – сложность алгоритма хеш-функции, для данного типа x.

1. **Сопровождение**
   1. Требование к программно-аппаратной платформе

Требований к семейству операционных систем нет. Требуемое место на диске: 11Кб для файлов с описанием и реализацией необходимых классов и 4Кб для кода тестовой программы. Компилировать следует по стандарту C++20.

* 1. Инструкция по установке и использованию

Для использования класса необходимо подключить заголовочный файл “hashmap.h”.

* 1. Сведения о недостатке программного продукта.

Несмотря на хорошую асимптотическую сложность в среднем, время работы алгоритмов в целом заметно больше возможного.

1. **Заключение**

В ходе работы был создан программный продукт, соответствующий заданию. Была разработана и реализована система классов для реализации структуры данных «Хеш-Таблица»

Достоинства:

* Позволяет хранить пары чисел ключ-значение.
* Позволяет совершать поиск в среднем за O(1).
* Позволяет итерироваться по элементам таблицы.
* Простота использования.

Недостатки:

* Из-за архитектуры использует большое количество памяти для выставления меток.
* Операция вставки может быть затратной из-за долгой перестройки таблицы.

Перспективы дальнейшего развития:

* Оптимизация дорогостоящих алгоритмов, таких как вставка.
* Оптимизация кода в целом.

**Список используемых источников**

1. Хеш-таблица [Электронный ресурс]//Википедия[сайт] URL: <https://ru.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88-%D1%82%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D0%B0>
2. Хеш-таблицы [Электронный ресурс]//Хабр[сайт] URL: <https://habr.com/ru/articles/509220/>
3. Хеш-функция [Электронный ресурс]//Википедия[сайт] URL: <https://ru.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88-%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F>
4. Хеш-таблица [Электронный ресурс]//neerc.ifmo.ru[сайт] URL: <https://neerc.ifmo.ru/wiki/index.php?title=%D0%A5%D0%B5%D1%88-%D1%82%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D0%B0>
5. Unordered map [Электронный ресурс]//cppreference.com[сайт] URL: <https://en.cppreference.com/w/cpp/container/unordered_map.html>
6. Unordered map [Электронный ресурс]//learn.microsoft.ru[сайт] URL: <https://learn.microsoft.com/ru-ru/cpp/standard-library/unordered-map-class?view=msvc-170>
7. Class HashMap [Электронный ресурс]//docks.oracle.com[сайт] URL: <https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html>

Приложение 1. Листинг заголовочного файла hashmap.h

#pragma once

#include <list>

#include <iostream>

#include <vector>

#include <memory>

namespace curs

{

    const size\_t size\_table[] =

        {

            5ul, 53ul, 97ul, 193ul, 389ul,

            769ul, 1543ul, 3079ul, 6151ul, 12289ul,

            24593ul, 49157ul, 98317ul, 196613ul, 393241ul,

            786433ul, 1572869ul, 3145739ul, 6291469ul, 12582917ul,

            25165843ul, 50331653ul, 100663319ul, 201326611ul, 402653189ul,

            805306457ul, 1610612741ul, 3221225473ul, 4294967291ul};

    const size\_t max\_size\_table\_index = 23;

    enum class pair\_type

    {

        REGULAR,

        BUCKET\_BEGIN,

        LIST\_END

    };

    template <typename key\_t, typename value\_t>

    class hash\_map;

    /////////////////////////

    template <typename key\_t, typename value\_t>

    class pair

    {

        friend class hash\_map<key\_t, value\_t>;

    public:

        pair(const key\_t &key, const value\_t &value, pair\_type type) : \_key(key), \_value(value), type(type) {}

        key\_t key() { return \_key; }

        value\_t value() { return \_value; }

        pair &operator=(const pair &p)

        {

            \_key = p.\_key;

            \_value = p.\_value;

            type = p.type;

            return \*this;

        }

    private:

        key\_t \_key;

        value\_t \_value;

        pair\_type type;

    };

    /////////////////////////

    template <typename key\_t, typename value\_t>

    class hash\_map

    {

    private:

        const double max\_load\_factor = 0.75;

        size\_t size\_table\_index;

        std::list<pair<key\_t, value\_t>> element\_list;

        std::vector<decltype(element\_list.begin())> iterators;

        void resize();

        size\_t \_size = 0;

    public:

        struct iterator

        {

            friend class hash\_map<key\_t, value\_t>;

        private:

            std::list<pair<key\_t, value\_t>>::iterator iter;

            void skip()

            {

                while ((\*iter).type == pair\_type::BUCKET\_BEGIN)

                {

                    iter++;

                }

            }

        public:

            iterator(std::list<pair<key\_t, value\_t>>::iterator iter) : iter(iter) {}

            iterator &operator++()

            {

                ++iter;

                skip();

                return \*this;

            }

            pair<key\_t, value\_t> \*operator->()

            {

                return &(\*iter);

            }

            pair<key\_t, value\_t> &operator\*() const

            {

                return \*iter;

            }

            bool operator==(const iterator &other)

            {

                return iter == other.iter;

            }

            bool operator!=(const iterator &other)

            {

                return iter != other.iter;

            }

        };

        hash\_map();

        void put(const key\_t &, const value\_t &);

        void remove(const key\_t &);

        bool contains(const key\_t &);

        bool empty();

        value\_t &get(const key\_t &);

        value\_t &operator[](const key\_t &);

        size\_t total\_buckets();

        size\_t size();

        void clear();

        iterator begin();

        iterator end();

        void print();

        static size\_t hash(const key\_t &, size\_t);

        hash\_map<key\_t, value\_t> &operator=(const hash\_map<key\_t, value\_t> &);

    };

} // namespace curs

#include "hashmap.tpp"

Приложение 2. Листинг файла с реализацией hashmap.tpp

#include "hashmap.h"

using namespace curs;

template <typename key\_t, typename value\_t>

size\_t

hash\_map<key\_t, value\_t>::total\_buckets()

{

    return size\_table[size\_table\_index];

}

template <typename key\_t, typename value\_t>

hash\_map<key\_t, value\_t>::hash\_map() : size\_table\_index(0)

{

    element\_list = std::list<pair<key\_t, value\_t>>();

    iterators = std::vector<decltype(element\_list.begin())>(total\_buckets() + 1);

    for (size\_t i = 0; i < total\_buckets(); i++)

    {

        element\_list.push\_back(pair(key\_t(), value\_t(), pair\_type::BUCKET\_BEGIN));

        iterators[i] = std::prev(element\_list.end());

    }

    element\_list.push\_back(pair(key\_t(), value\_t(), pair\_type::LIST\_END));

}

template <typename key\_t, typename value\_t>

size\_t

hash\_map<key\_t, value\_t>::hash(const key\_t &key, size\_t \_\_size)

{

    auto hash = std::hash<key\_t>();

    return hash(key) % \_\_size;

}

template <typename key\_t, typename value\_t>

void hash\_map<key\_t, value\_t>::print()

{

    size\_t c = 0;

    for (auto i : element\_list)

    {

        std::cout << c << ") ";

        switch (i.type)

        {

        case pair\_type::LIST\_END:

            std::cout << "$END OF LIST";

            break;

        case pair\_type::BUCKET\_BEGIN:

            std::cout << "$BEGINING OF BUCKET";

            break;

        case pair\_type::REGULAR:

            std::cout << i.\_key << " " << i.\_value;

            break;

        default:

            std::cout << "$UNKNOWN";

            break;

        }

        std::cout << std::endl;

        if (i.type == pair\_type::LIST\_END)

            break;

        c++;

    }

}

template <typename key\_t, typename value\_t>

bool hash\_map<key\_t, value\_t>::contains(const key\_t &key)

{

    auto i = iterators[hash(key, total\_buckets())];

    for (i++; i->type == pair\_type::REGULAR; i++)

        if (i->\_key == key)

            return true;

    return false;

}

template <typename key\_t, typename value\_t>

void hash\_map<key\_t, value\_t>::put(const key\_t &key, const value\_t &val)

{

    auto i = iterators[hash(key, total\_buckets())];

    for (i++; i->type == pair\_type::REGULAR; i++)

        if (i->\_key == key)

        {

            i->\_value = val;

            return;

        }

    element\_list.insert(i, pair(key, val, pair\_type::REGULAR));

    \_size++;

    if (static\_cast<double>(size()) / total\_buckets() >= max\_load\_factor)

    {

        resize();

    }

}

template <typename key\_t, typename value\_t>

void hash\_map<key\_t, value\_t>::remove(const key\_t &key)

{

    auto i = iterators[hash(key, total\_buckets())];

    for (i++; i->type == pair\_type::REGULAR; i++)

        if (i->\_key == key)

        {

            element\_list.erase(i);

            \_size--;

            return;

        }

}

template <typename key\_t, typename value\_t>

value\_t &hash\_map<key\_t, value\_t>::get(const key\_t &key)

{

    auto i = iterators[hash(key, total\_buckets())];

    for (i++; i->type == pair\_type::REGULAR; i++)

        if (i->\_key == key)

        {

            return (\*i).\_value;

        }

    element\_list.insert(i, pair(key, value\_t(), pair\_type::REGULAR));

    return (\*(--i)).\_value;

}

template <typename key\_t, typename value\_t>

value\_t &hash\_map<key\_t, value\_t>::operator[](const key\_t &key)

{

    return get(key);

}

template <typename key\_t, typename value\_t>

size\_t hash\_map<key\_t, value\_t>::size()

{

    return \_size;

}

template <typename key\_t, typename value\_t>

bool hash\_map<key\_t, value\_t>::empty()

{

    return size() == 0;

}

template <typename key\_t, typename value\_t>

void hash\_map<key\_t, value\_t>::clear()

{

    size\_table\_index=0;

    element\_list = std::list<pair<key\_t, value\_t>>();

    iterators = std::vector<decltype(element\_list.begin())>(total\_buckets() + 1);

    size\_t i;

    for (i = 0; i < total\_buckets(); i++)

    {

        element\_list.push\_back(pair(key\_t(), value\_t(), pair\_type::BUCKET\_BEGIN));

    }

    element\_list.push\_back(pair(key\_t(), value\_t(), pair\_type::LIST\_END));

    i = 0;

    for (auto iter = element\_list.begin(); iter != element\_list.end(); i++, iter++)

    {

        iterators[i] = iter;

    }

    \_size=0;

}

template <typename key\_t, typename value\_t>

hash\_map<key\_t, value\_t> &hash\_map<key\_t, value\_t>::operator=(const hash\_map<key\_t, value\_t> &map)

{

    element\_list = std::list<pair<key\_t, value\_t>>();

    size\_table\_index = map.size\_table\_index;

    iterators = std::vector<decltype(element\_list.begin())>(map.iterators.size());

    \_size = map.\_size;

    size\_t i = 0;

    for (auto iter = map.element\_list.begin(); iter != map.element\_list.end(); iter++)

    {

        element\_list.push\_back(\*iter);

        if (element\_list.back().type == pair\_type::BUCKET\_BEGIN)

        {

            iterators[i++] = std::prev(element\_list.end());

        }

    }

    return \*this;

}

template <typename key\_t, typename value\_t>

hash\_map<key\_t, value\_t>::iterator hash\_map<key\_t, value\_t>::begin()

{

    auto res = iterator(element\_list.begin());

    res.skip();

    return res;

}

template <typename key\_t, typename value\_t>

hash\_map<key\_t, value\_t>::iterator hash\_map<key\_t, value\_t>::end()

{

    return iterator(std::prev(element\_list.end()));

}

template <typename key\_t, typename value\_t>

void hash\_map<key\_t, value\_t>::resize()

{

    if (size\_table\_index < max\_size\_table\_index)

    {

        size\_table\_index++;

    }

    else

        return;

    auto buf\_element\_list = element\_list;

    element\_list.clear();

    iterators = std::vector<decltype(element\_list.begin())>(total\_buckets() + 1);

    for (size\_t i = 0; i < total\_buckets(); i++)

    {

        element\_list.push\_back(pair(key\_t(), value\_t(), pair\_type::BUCKET\_BEGIN));

        iterators[i] = std::prev(element\_list.end());

    }

    element\_list.push\_back(pair(key\_t(), value\_t(), pair\_type::LIST\_END));

    for (auto &j : buf\_element\_list)

    {

        if (j.type == pair\_type::REGULAR)

        {

            auto iter = iterators[hash(j.\_key, total\_buckets())];

            for (iter++; iter->type == pair\_type::REGULAR; iter++)

            {

                ;

            }

            element\_list.insert(iter, pair(j.\_key, j.\_value, pair\_type::REGULAR));

        }

    }

}

Приложение 3. Листинг тестовой программы

#include <iostream>

#include <limits>

#include "hashmap.h"

using namespace std;

using namespace curs;

int main()

{

    int menu\_key;

    hash\_map<string, int> map;

    string key;

    int val;

    do

    {

        cout << endl;

        cout << "Хеш-Таблица" << endl

             << "Ключ: std::string" << endl

             << "Значение: int" << endl

             << "1. Вставить" << endl

             << "2. Удалить" << endl

             << "3. Проверить наличие" << endl

             << "4. Количество элементов" << endl

             << "5. Размер таблицы" << endl

             << "6. Вывести все элементы" << endl

             << "7. Отчистить" << endl

             << "8. Вычислить хеш-код" << endl

             << "0. Выход" << endl

             << ">>";

        cin >> menu\_key;

        if (cin.fail())

        {

            menu\_key = -1;

            cin.clear();

            cin.ignore(numeric\_limits<streamsize>::max(), '\n');

        }

        switch (menu\_key)

        {

        case 0:

            break;

        case 1:

            cout << "Введите ключ (std::string): ";

            cin >> key;

            cout << "Введие значение (int): ";

            cin >> val;

            if (cin.fail())

            {

                cin.clear();

                cin.ignore(numeric\_limits<streamsize>::max(), '\n');

                cout << "Значение должно быть числом" << endl;

            }

            else

            {

                map.put(key, val);

            }

            break;

        case 2:

            cout << "Введите ключ (std::string): ";

            cin >> key;

            map.remove(key);

            break;

        case 3:

            cout << "Введите ключ (std::string): ";

            cin >> key;

            if (map.contains(key))

            {

                cout << "\"" << key << "\" содержится в таблице" << endl;

            }

            else

            {

                cout << "\"" << key << "\" не содержится в таблице" << endl;

            }

            break;

        case 4:

            cout << "Количество элементов: " << map.size() << endl;

            break;

        case 5:

            cout << "Размер таблицы: " << map.total\_buckets() << endl;

            break;

        case 6:

            if (map.empty())

            {

                cout << "Таблица пуста" << endl;

            }

            for (auto i : map)

            {

                cout << "\"" << i.key() << "\": " << i.value() << endl;

            }

            break;

        case 7:

            map.clear();

            break;

        case 8:

            cout << "Введите ключ (std::string): ";

            cin >> key;

            cout << "Хеш: " << map.hash(key, map.total\_buckets()) << endl;

            break;

        default:

            cout << "Неизвестная команда" << endl;

            break;

        }

    } while (menu\_key != 0