МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования

"МИРЭА - Российский технологический университет"

РТУ МИРЭА

Институт информационных технологий (ИИТ)
Кафедра МОиСИТ

ОТЧЕТ
ПО ПРАКТИЧЕСКОЙ РАБОТЕ №6.1

«Быстрый доступ к данным с помощью хеш-таблиц»

по дисциплине

«Структуры и алгоритмы обработки данных»

Выполнил студент группы ИКБО-10-24 Бикташев И. И.

Практическую работу выполнил «__»_____2025 г.

«Зачтено» «__»_____2025 г.

Москва 2025

Цель работы: освоить приёмы хеширования и эффективного поиска элементов множества.

Задание: Разработать приложение, которое использует хеш-таблицу

Описание алгоритма: Хеш-таблица представляет собой ассоциативный массив пар ключ-значение. Расположение элемента в массиве определяется специально хеш-функцией. Когда хеш-функция выдает одинаковый результат для двух разных записей вызывает вторая хеш-функция, которая определяет собой шаг, с которым после будет происходить перебор по массиву пока не найдется свободное место.

Массив состоит из экземпляров отдельной структуры, которая содержит собственно запись и её доступность, если элемент массива доступен значит он не удалён.

Реализация: Одна запись представляет собой экземпляр структуры Product (листинг 1). В качестве ключа будет использовать поле code. Два продукта считаются одинаковыми, если совпадают их ключи (коды).

Листинг 1 – Структура Product

```
struct Product {
    unsigned short code;
    string name;
    unsigned short price;

    bool operator ==(const Product& other) const {
        return other.code == code;
    }
};
```

Хеш-таблица реализована через класс HashTable, массив хеш-таблицы будет состоять из экземпляров структуры Node. Первая хеш-функция принимает ключ и возвращает остаток от деления на размер массива. Вторая хеш-функция делает то же самое, но возвращает остаток от деления на размер массива – 1 и к получившемуся значению прибавляет 1 (листинг 2).

Листинг 2 - private поля класса HashTable

```
class HashTable {
    struct Node {
        Product value;
        bool state; // если значение флага state = false, значит элемент
массива был удален
        Node(const Product& value_) : value(value_), state(true) {}
```

```
    };
    const int default_size = 8; // начальный размер нашей таблицы
    const double rehash_size = 0.75; // коэффициент, при котором произойдет
увеличение таблицы

    Node** arr; // соответственно в массиве будут хранится структуры Node*
    int size; // сколько элементов у нас сейчас в массиве (без учета deleted)
    int buffer_size; // размер самого массива, сколько памяти выделено под
хранение нашей таблицы
    int size_all_non_nullptr; // сколько элементов у нас сейчас в массиве (с
учетом deleted)

    int hash1(unsigned short code) {
        return code % buffer_size;
    }
    int hash2(unsigned short code) {
        return code % (buffer_size - 1) + 1;
    }
```

Конструктор HashTable инициализирует все его основные поля и заполняет массив нулевыми указателями. Деструктор высвобождает память занятую массивом (листинг 3).

Листинг 3 - Конструктор и деструктор

```
public:
    HashTable() {
        buffer_size = default_size;
        size = 0;
        size_all_non_nullptr = 0;
        arr = new Node*[buffer_size];
        for (int i = 0; i < buffer_size; ++i)
            arr[i] = nullptr; // заполняем nullptr - то есть если значение
отсутствует, и никто раньше по этому адресу не обращался
    }
    ~HashTable() {
        for (int i = 0; i < buffer_size; ++i)
            if (arr[i])
                delete arr[i];
        delete[] arr;
    }
```

Метод Resize удваивает размер массива массива хеш-таблица, а метод Rehash удаляет все «пустые ячейки», элементы со значением поля state равному false (листинг 4).

Листинг 4 - Методы Resize и Rehash

```
    void Resize() {
        int past_buffer_size = buffer_size;
        buffer_size *= 2;
        size_all_non_nullptr = 0;
        size = 0;
        Node** arr2 = new Node*[buffer_size];

        for (int i = 0; i < buffer_size; ++i)
            arr2[i] = nullptr;
```

```
        swap(arr, arr2);
        for (int i = 0; i < past_buffer_size; ++i)
            if (arr2[i] && arr2[i]->state)
                Add(arr2[i]->value); // добавляем элементы в новый массив
        // удаление предыдущего массива
        for (int i = 0; i < past_buffer_size; ++i)
            if (arr2[i])
                delete arr2[i];
        delete[] arr2;
    }
    void Rehash() {
        size_all_non_nullptr = 0;
        size = 0;
        Node** arr2 = new Node*[buffer_size];

        for (int i = 0; i < buffer_size; ++i)
            arr2[i] = nullptr;

        swap(arr, arr2);
        for (int i = 0; i < buffer_size; ++i)
            if (arr2[i] && arr2[i]->state)
                Add(arr2[i]->value);
        // удаление предыдущего массива
        for (int i = 0; i < buffer_size; ++i)
            if (arr2[i])
                delete arr2[i];
        delete[] arr2;
    }
```

Метод Find показывает находится ли элемент с данным ключом в хеш-таблице (листинг 5).

Листинг 5 - Метод Find

```
    bool Find(unsigned short code) {
        int h1 = hash1(code); // значение, отвечающее за начальную позицию
        int h2 = hash2(code); // значение, ответственное за "шаг" по таблице
        int i = 0;

        while (arr[h1] != nullptr && i < buffer_size) {
            if (arr[h1]->value.code == code && arr[h1]->state)
                return true; // такой элемент есть

            h1 = (h1 + h2) % buffer_size;
            i++; // если у нас i >=  buffer_size, значит мы уже обошли
абсолютно все ячейки, именно для этого мы считаем i, иначе мы могли бы
зациклиться.
        }

        return false;
    }
```

Метод Remove меняет состояние элемента по данному ключу на удалённое состояние (листинг 6).

Листинг 6 - Метод Remove

```
    bool Remove(unsigned short code) {
        int h1 = hash1(code);
        int h2 = hash2(code);
        int i = 0;
```

```
        while (arr[h1] != nullptr && i < buffer_size) {
            if (arr[h1]->value.code == code && arr[h1]->state) {
                arr[h1]->state = false;
                size_all_non_nullptr--;
                return true;
            }
            h1 = (h1 + h2) % buffer_size;
            i++;
        }

        return false;
    }
```

Метод Add вносит элемент в хеш-таблицу, вычисляя для него индекс с помощью первой хеш-функции. Если данный индекс уже занят, то массив перебирается с шагом равным значению второй хеш-функции и ищем свободное место. Если запись с данным ключом уже есть в массиве, то метод возвращает ложь (листинг 7).

Листинг 7 - Метод Add

```
    bool Add(const Product& value) {
        if (size + 1 > int(rehash_size * buffer_size))
            Resize();
        else if (size_all_non_nullptr > 2 * size)
            Rehash(); // происходит рехеш, так как слишком много deleted-
элементов

        int h1 = hash1(value.code);
        int h2 = hash2(value.code);
        int i = 0;
        int first_deleted = -1; // запоминаем первый подходящий (удаленный)
элемент

        while (arr[h1] != nullptr && i < buffer_size) {
            if (arr[h1]->value == value && arr[h1]->state)
                return false; // такой элемент уже есть, а значит его нельзя
вставлять повторно
            if (!arr[h1]->state && first_deleted == -1) // находим место для
нового элемента
                first_deleted = h1;
            h1 = (h1 + h2) % buffer_size;
            i++;
        }

        if (first_deleted == -1) { // если не нашлось подходящего места,
создаем новый Node
            arr[h1] = new Node(value);
            size_all_non_nullptr++; // так как мы заполнили один пробел, не
забываем записать, что это место теперь занято
        } else {
            arr[first_deleted]->value = value;
            arr[first_deleted]->state = true;
        }

        size++; // и в любом случае мы увеличили количество элементов
        return true;
```

```
    }
```

И наконец метод Print выводит информацию о хеш-таблице, ее размер количество занятый ячеек в массиве и содержание массива (листинг 8).

Листинг 8 - Метод Print

```
    void Print() {
        for (int i = 0; i < buffer_size; i++) {
            cout << i << " ";
            if (!arr[i]) {
                cout << "empty" << endl;
            } else if (!arr[i]->state) {
                cout << "deleted" << endl;
            } else {
                cout << arr[i]->value.code << " " << arr[i]->value.name
                    << " " << arr[i]->value.price << endl;
            }
        }
        cout << "Size: " << size_all_non_nullptr << endl;
        cout << "Buffer size: " << buffer_size << endl;
    }
```

Также были созданы функции fillHashTable, которая заполняет хеш-таблицу определённым количеством элементов, и add, которая добавляет в хеш-таблицу с заданным ключом и случайно сгенерированными данными (листинг

Листинг 9 - Дополнительные функции

```
void fillHashTable(HashTable* table, const int n) {
    for (int i = 0; i < n; i++) {
        Product product;
        product.code = rand() % 999;
        product.name = "Product_" + to_string(rand());
        product.price = rand() % 999;
        if (!table->Add(product)) {
            i--;
        }
    }
}

bool add(HashTable* table, unsigned short code) {
    Product product;
    product.code = code;
    product.name = "Product_" + to_string(rand());
    product.price = rand() % 999;
    return table->Add(product);
}
```

Был реализованы интерфейс и команды для взаимодействия с хеш-таблицей: add, find, remove, print, help, exit.

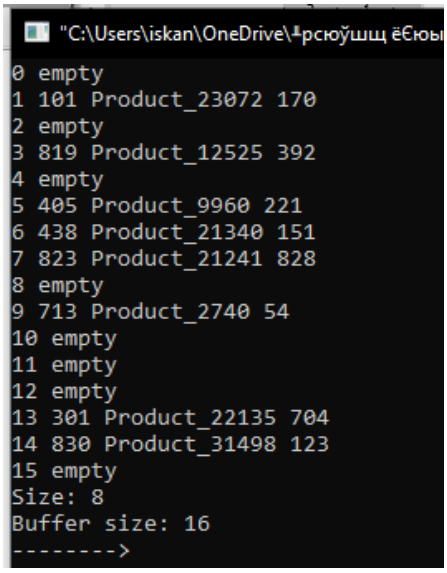

Рисунок 1 - Начальный экран программы

Команда add добавляет в таблицу случайно сгенерированные данные по ключу. Если элемент с данным ключом уже присутствует в таблице, то команда выводит 0, иначе 1 (рисунок 2)

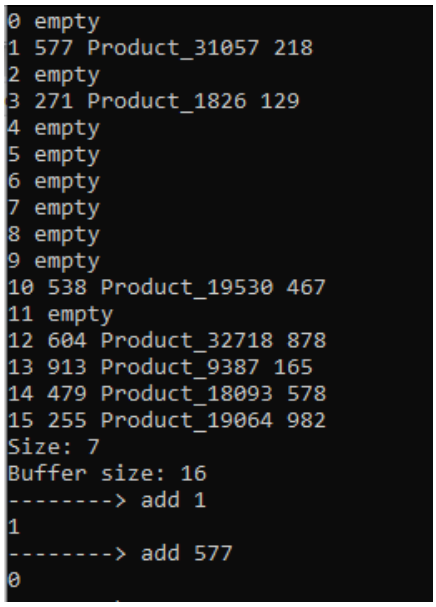

Рисунок 2 - Команда add

Команда print позволяет вывести на экран все ячейки хеш-таблицы (рисунок 3)

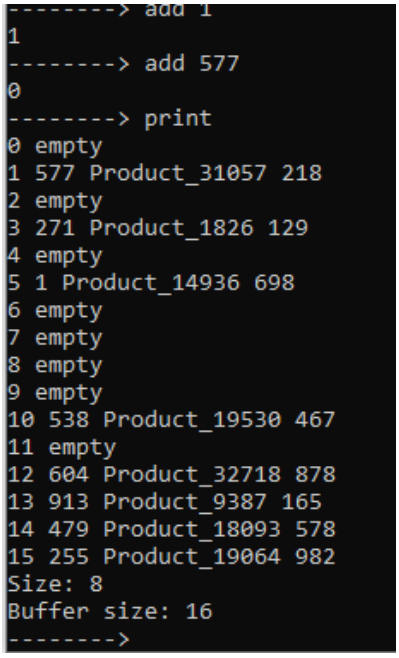

Рисунок 3 - Команда print

Команда find проверяет наличие элемента в хеш-таблице. Если данного элемента в таблице нет, то она выводит 0, иначе 1 (рисунок 4)

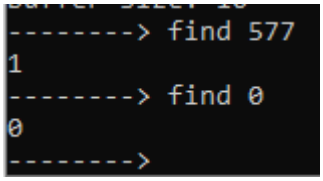

Рисунок 4 - Команда find

Команда remove удаляет элемент из хеш-таблицы. Если элемент найден и удалён, то выводит 1, иначе 0 (рисунок 5)

Рисунок 5 - Команда remove

Вывод

Поставленные задачи выполнены: реализованы функционал хеш-таблицы, методы для взаимодействия с ней, разработана программа для внесения элементов в хеш-таблицу, удаления и их поиска.

Литература

Страуструп Б. Программирование. Принципы и практика с использованием C++. 2-е изд., 2016.

Документация по языку C++ [Электронный ресурс]. URL: https://docs.microsoft.com/ru-ru/cpp/cpp/ (дата обращения 01.09.2021).

Курс: Структуры и алгоритмы обработки данных. Часть 2 [Электронный ресурс]. URL: https://online-edu.mirea.ru/course/view.php?id=4020 (дата обращения 01.09.2021).