МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ

Ордена Трудового Красного Знамени

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Московский технический университет связи и информатики»

Кафедра «Программная инженерия»

Отчёт по Лабораторной работе №3

По дисциплине: «Информационные технологии и программирование»

«Класс Object. Работа с хэш-таблицами»

Выполнила: студентка группы БПИ2401

Алексеева Татьяна Игоревна

Проверил: Харрасов Камиль Раисович

Москва

2025

Цель работы

Изучить и практически реализовать структуры данных хэш-таблица с использованием метода цепочек, освоение основных операций с таблицей – добавления, поиска и удаления элементов. В качестве практического примера рассматривается хранение информации о студентах, где ключом является номер зачётной книжки, а значение – объект класса Student. Также работа включает освоение встроенного класса HashMap для эффективного управления данными и демонстрацию преимущества хэш-таблиц при быстром доступе к информации.

Ход работы

В первом задании нужно создать класс HashTable, который будет реализовывать хэш-таблицу с помощью метода цепочек. Также нужно реализовать методы put(key, value) – добавление пары «ключ-значение», get(key, value) – получение пары, remove – удаление пары «ключ-значение» и добавить методы size() и isEmpty(), которые возвращают количество элементов в таблице и проверяют, пуста ли она.

```
import java.util.LinkedList;

public class HashTable<K, V> {

    private static class Entry<K, V> {
        private K key;
        private V value;

        public Entry (K key, V value) {
            this.key = key;
            this.value = value;
        }

        public K getKey() {
            return key;
        }
        public V getValue() {
            return value;
        }
        public void setValue(V value) {
            this.value = value;
        }
    }
```

В данном фрагменте мы создаём обобщённый класс HashTable<K, V>, реализующий собственную хэш-таблицу, где: K обозначает тип ключа (key), а V обозначает тип значения (value). Такой подход делает класс универсальным, то есть таблица может хранить любые данные.

Внутри класса HashTable определяется вложенный статический класс Entry<K, V>. Он представляет собой ячейку (запись) таблицы и хранит пару «ключ-значение». То

есть переменные private K key и private V = value – это поля объекта Entry, где хранятся ключ и значение соответственно.

Далее идёт конструктор Entry(K key, V value), он используется для создания новой пары ключ-значение и инициализации соответствующих полей.

Прописанные после конструктора методы предоставляют доступ к полям объекта: getKey() – возвращает ключ, getValue() – возвращает значение, setValue() – позволяет изменить значение, не меняя ключ.

Второй блок работы – создание массива для хранения данных, определение конструктора и хэш-функций.

```
private static final int capacity = 10;

private LinkedList<Entry<K, V>>[] table;
private int size;

public HashTable() {
    table = new LinkedList[capacity];

    size = 0;
}

private int hash(K key) {
    return Math.abs(key.hashCode() % table.length);
}
```

Здесь мы задаём основные поля класса HashTable, конструктор для их инициализации и хэш-функцию, которая вычисляет позицию для каждого ключа таблицы.

private static final int capacity = 10 – это константа, определяющая начальный размер хэш0таблицы. Она задаёт, сколько корзин (ячеек) будет в массиве, где будут храниться элементы. static final означает, что значение константы одно для всех экземпляров класса и не может быть изменено. В нашем случае таблица состоит из 10 цепочек (индексов от 0 до 9).

private LinkedList<Entry<K, V>>[] table – это главная структура хранения данных в хэш-таблице – массив списков (цепочек). Каждый элемент массива table[i] – это связанный список (LinkedList), в котором хранятся все элементы, попавшие в одну и ту же «корзину» (при одинаковом хэш-индексе). Таким образом, если возникает коллизия (разные ключи имеют одинаковый индекс), элементы не перезаписываются, о добавляются в этот список.

Переменная size хранит текущее количество элементов в таблице. Она увеличивается при добавлении новых записей (put) и уменьшается при удалении (remove).

Далее идёт конструктор HashTable. Он создаёт массив длинной capacity. Каждый элемент массива изначально равен null. Инициализирует счётчик size = 0.

Метод hash(K key) – это хэш-функция, которая преобразует ключ в индекс массива. key.hashCode() – встроенный метод любого объекта Java, возвращающий числовое значение (хэш-код) для данного ключа. % table.length – берёт остаток от деления на размер таблицы, чтобы индекс не выходил за пределы массива. Math.abs() используется, чтобы получить положительное число, так как хэш-код может быть отрицательным. Таким образом, метод гарантирует, что для любого ключа вычисляется индекс от 0 до 9.

```
    public void put(K key, V value) {
        int index = hash(key);
        if (table[index] == null) {
            table[index] = new LinkedList<Entry<K, V>>();
        }
        for (Entry<K, V> entry : table[index]) {
            if (entry.getKey().equals(key)) {
                entry.setValue(value);
                return;
            }
        }
        table[index].add(new Entry<K, V> (key, value));
        size++;
    }
```

Метод put(K key, V value) отвечает за добавление пары «ключ-значение» в хэш-таблицу. Он также может обновлять значение, если ключ уже существует.

int index = hash(key) – вызываем метод hash(key), который на основе ключа вычисляет индекс в массиве table. Индекс определяет в какой «корзине» (цепочке) будет храниться пара.

Массив table содержит связанные списки для каждой корзины. Если в данной таблице ещё нет списка, то есть table[index] == null, то мы создаём новый пустой список. Это позволяет избежать NullPointerException при следующей работе с корзиной.

Далее проверяем, что ключ существует. Перебираем все элементы в цепочке по текущему индексу. Сравниваем ключи с помощью equals(), так как просто сравнить ссылку (==) нельзя. Если ключ уже существует, то объявляем значение с помощью entry.setValue(value) и выходим из метода return, чтобы не добавлять новый элемент. Таким образом, метод обновляет значение по ключу, если ключ уже существует в таблице, не создавая дубликатов.

Если же ключа в цепочке нет, создаём новый объект Entry<K, V> с заданными ключом и значением. Добавляем его в связанны список корзины table[index].add… Увеличиваем счётчик size, так как в таблицу добавлен новый элемент.

```
    public V get(K key) {
        int index = hash(key);
```

```
        if (table[index] != null) {
            for (Entry<K, V> entry : table[index]) {
                if (entry.getKey().equals(key)) {
                    return entry.getValue();
                }
            }
        }

        return null;
    }
```

Метод `get(K key)` – отвечает за получение значения по заданному ключу из хэш-таблицы.

Сначала вызывается хэш-функция, которая преобразует ключ в индекс массива table. Индекс показывает, в какой цепочке нужно искать элемент. Проверяем, существует ли список по данному индексу. Если корзина пустая, значит элемента с таким ключом точно отсутствует.

Если корзина не пустая, то перебираем все записи в цепочке по найденному индексу. Сравниваем ключи методом equals(), чтобы определить, есть ли нужный ключ. Если ключ найден, возвращаем соответствующее значение с помощью `entry.getValue()`. Если ключ не найден, метод возвращает null.

```
    public void remove(K key) {
        int index = hash(key);

        if (table[index] != null) {
            for (Entry<K, V> entry : table[index]) {
                if (entry.getKey().equals(key)) {
                    table[index].remove(entry);
                    size--;
                    return;
                }
            }
        }
    }
```

Метод `remove(K key)` отвечает за удаление пары «ключ-значение» из хэш-таблицы. Опять сначала вызывается хэш-функция. Проверяем, существует ли связанный список по данному индексу. Если цепочка пуста, значит ключ точно отсутствует, и дальнейшие действия не требуются.

Перебираем все записи в цепочке по индексу, сравниваем ключи, если ключ найден, то удаляем элемент из списка `table[index].remove(entry)`, уменьшаем счётчик и выходи из метода, чтобы не продолжать перебор.

Если цикл закончился, а ключ не был найден, метод ничего не делает.

```
public int size() {
        return size;
    }
```

```
    public boolean isEmpty() {
        return size == 0;
    }

    public void printTable() {
        for (int i = 0; i < table.length; i++) {
            System.out.print(i + ": ");
            if (table[i] != null) {
                for (Entry<K, V> entry : table[i]) {
                    System.out.print("[" + entry.getKey() + " = " + entry.getValue() +
"] ");
                }
            }
            System.out.println();
        }
    }
```

Метод size() возвращает текущее количество элементов в таблице (переменная size).

Метод isEmpty() проверяет, пуста ли таблица. Если size == 0, возвращает true, иначе возвращает false.

Метод printTable() выводит содержимое всей хэш-таблицы на экран. Перебираем все цепочки массива table по индексам, и для каждой цепочки (если она не пустая) перебираем все элементы и выводим их в формате «ключ-значение».

Ниже показан пример создания, заполнения хэш-таблицы и продемонстрирована работа методов.

```
public static void main(String[] arhs) {
        HashTable<String, Integer> hashTable = new HashTable<>();

        hashTable.put("apple", 5);
        hashTable.put("banana", 3);
        hashTable.put("orange", 7);
        hashTable.put("pear", 2);

        hashTable.printTable();

        System.out.println("\nКоличество элементов: " + hashTable.size());
        System.out.println("Значение по ключу 'banana': " + hashTable.get("banana"));

        hashTable.remove("orange");
        System.out.println("\nПосле удаления 'orange':");
        hashTable.printTable();
    }
}
```

Результат в терминале:

```
0: [apple = 5] [orange = 7]
1:
2:
3:
4: [pear = 2]
5:
6:
7: [banana = 3]
8:
9:

Количество элементов: 4
Значение по ключу 'banana': 3

После удаления 'orange':
0: [apple = 5]
1:
2:
3:
4: [pear = 2]
5:
6:
7: [banana = 3]
8:
9:
PS C:\Users\taale\OneDrive\Desktop\ITandP\LabWorkThree>
```

Во втором задании 1 варианта необходимо реализовать хэш-таблицу для хранения информации о студентах. Ключом будет являться номер зачётной книжки, а значением – объект класса Student, содержащий поля имя, фамилия, возраст и средний балл. Необходимо реализовать операции вставки, поиска и удаления студента по номеру зачётки.

```java
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;

class Student {
    private String firstName;
    private String lastName;
    private int age;
    private double averageGrade;

    public Student(String firtsName, String lastName, int age, double averageGrade) {
        this.firstName = firtsName;
        this.lastName = lastName;
        this.age = age;
        this.averageGrade = averageGrade;
    }

    public String getFirstName() {
        return firstName;
    }
    public String getLastName() {
        return lastName;
    }
```

```
    public int getAge() {
        return age;
    }
    public double getAverageGrade() {
        return averageGrade;
    }

    @Override
    public String toString() {
        return lastName + " " + firstName + " (Возраст: " + age + ", Средний балл: " +
averageGrade + ")";
    }
}
```

Для начала объявляем класс и его поля. Класс Student содержит информацию о студенте: firstName — имя студента, lastName — фамилия студента, age — возраст студента, averageGrade — средний балл студента. Все поля объявлены private, что обеспечивает инкапсуляцию – доступ к полям возможет только через методы класса.

Далее идёт конструктор класса, который используется для создания нового объекта Student. Параметры конструктора инициализируют поля объекта: имя, фамилию, возраст и средний балл. this.firstName = firtsName и аналогичные строки связывают значения параметров конструктора с полями объекта.

Методы доступа (геттеры) позволяют читать значения полей объекта, не предоставляя прямого доступа к самим полям.

В конце происходит переопределение метода toString(), который используется для удобства вывода информации о студенте.

```
public class StudentHashMap {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        Map<String, Student> studentMap = new HashMap<>();

        while (true) {
            System.out.println("\nМеню:");
            System.out.println("1 – Добавить студента");
            System.out.println("2 – Найти студента по номеру зачетки");
            System.out.println("3 – Удалить студента");
            System.out.println("4 – Показать всех студентов");
            System.out.println("0 – Выход");
            System.out.print("Выберите действие: ");
            int choice = scanner.nextInt();
            scanner.nextLine();

            switch (choice) {
                case 1:
                    System.out.print("Введите номер зачетки: ");
                    String id = scanner.nextLine();
```

```
                    System.out.print("Введите имя: ");
                    String name = scanner.nextLine();
                    System.out.print("Введите фамилию: ");
                    String surname = scanner.nextLine();
                    System.out.print("Введите возраст: ");
                    int age = scanner.nextInt();
                    System.out.print("Введите средний балл: ");
                    double avg = scanner.nextDouble();

                    studentMap.put(id, new Student(name, surname, age, avg));
                    System.out.println("Студент добавлен!");
                    break;

                case 2:
                    System.out.print("Введите номер зачетки: ");
                    id = scanner.nextLine();
                    Student found = studentMap.get(id);
                    if (found != null) {
                        System.out.println("Найден студент: " + found);
                    } else {
                        System.out.println("Студент не найден");
                    }
                    break;

                case 3:
                    System.out.println("Введите номер зачётки для удаления: ");
                    id = scanner.nextLine();
                    if (studentMap.remove(id) != null) {
                        System.out.println("Студент удалён");
                    } else {
                        System.out.println("Студент с таким номером не найден");
                    }
                    break;

                case 4:
                    if (studentMap.isEmpty()) {
                        System.out.println("Таблица пуста");
                    } else {
                        System.out.println("Все студенты: ");
                        for (Map.Entry<String, Student> entry : studentMap.entrySet()) {
                            System.out.println(entry.getKey() + " -> " +
entry.getValue());
                        }
                    }
                break;

                case 0:
                    System.out.println("Завершение работы программы");
                    scanner.close();
                    return;
```

```
                default:
                System.out.println("Неверный выбор, попробуйте снова");
            }
        }
    }
}
```

Класс StudentHashMap реализует консольное приложение, которое хранит студентов в хэш-таблице и позволяет выполнять операции вставки, поиска, удаления и просмотра всех студентов.

В первую очередь создаётся хэш-таблица с ключом – номер зачётной книжки, значением – объект с информацией о студенте. Scanner используется для считывания ввода пользователя из консоли.

Цикл while (true) обеспечивает постоянное отображение меню, пока пользователь не выберет выход (0). `scanner.nextInt()` считывает выбор пользователя, `scanner.nextLine()` очищает буфер ввода, чтобы корректно читать строки после чисел.

Обработка действий пользователя происходит через switch. При добавлении студента сначала считываются данные студента. Затем создаётся объект Student и добавляется в хэш-таблицу с помощью put. Если ключ уже существует, старый объект перезаписывается новым (поведение HashMap).

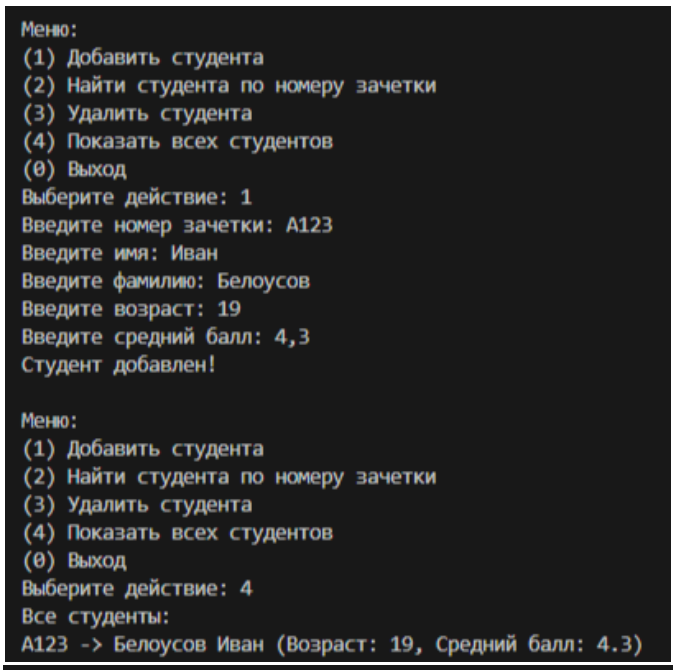
Для поиска студента по номеру зачётки сначала считывается ключ (номер зачётки), с помощью метода get ищется объект Student, если он найден, то выводится его информация, если ключа нет в таблице, то выводится сообщение, что студент не найден.

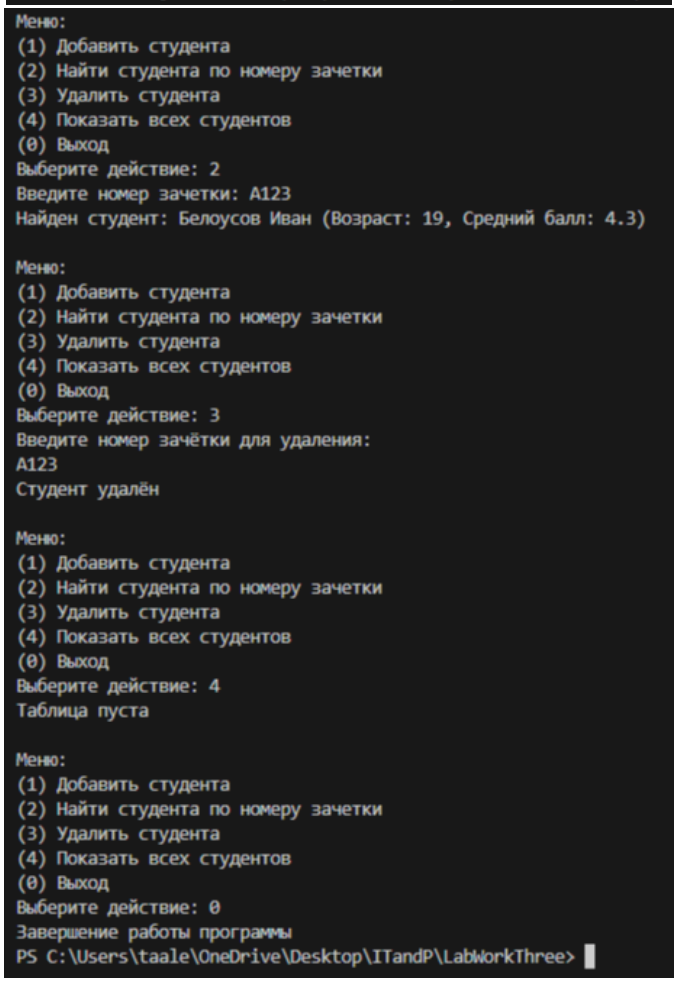
Для удаления студента вводится номер зачётки, метод remove удаляет элемент по ключу.

Перед выводом всех студентов идёт проверка, что таблица не пустая, с помощью метода isEmpty(). Если нет, то перебираем все пары «ключ-значение» и выводи их.

Если пользователь вводит «0», программа завершает работу.

Посмотрим работу консольного приложения:





Вывод

В результате выполнения работы была реализована собственная хэш-таблица с методом цепочек, обеспечивающая эффективное хранение и поиск пар «ключ-значение». Были продемонстрированы операции добавления, поиска и удаления элементов, а также проверка состояния таблицы. Создан класс Student с методами доступа и переопределённом toString() для удобного вывода информации. С

помощью HashMap разработано консольное приложения для управления информацией о студентах, которое подтвердило, что хэш-таблица обеспечивает быстрый доступ к данным и эффективно обрабатывает коллизии.

Ответы на контрольные вопросы

1. Класс Object является базовым классом всей иерархии классов в Java. Все остальные классы, включая пользовательские, автоматически наследуют этот класс, даже если явного указания нет. Это значит, что каждый объект в Java имеет методы, определенные в классе Object: toString(), equals(), hashCode(), getClass(), clone(), wait(), notify(), notifyAll(), finalize(). Благодаря этому все объекты в Java могут сравниваться, хэшироваться, преобразовываться в строку и использоваться полиморфно.
2. По умолчанию equals() проверяет только идентичность ссылок (==), а не содержимое объектов. Если объекты одного класса имеют одинаковые данные, но разные ссылки, логично считать их равными — для этого нужно переопределить equals(). Метод hashCode() используется в коллекциях на основе хэш-таблиц (HashMap, HashSet) для оптимизации поиска элементов. Переопределение hashCode() необходимо, чтобы объекты, которые равны по equals(), имели одинаковый хэш-код, иначе коллекции будут работать некорректно.
3. Если x.equals(y) → x.hashCode() == y.hashCode(); Если x.hashCode() == y.hashCode() → не обязательно x.equals(y). Метод equals() должен быть рефлексивным, симметричным, транзитивным, согласованным и безопасным относительно null. Переопределение hashCode() обычно делается на основе значимых полей объекта.
4. Метод toString() возвращает строковое представление объекта. По умолчанию он выводит имя класса и хеш-код, но его часто переопределяют, чтобы получать более информативное описание содержимого объекта.
5. Метод finalize() вызывается перед удалением объекта сборщиком мусора для освобождения ресурсов. Он считается устаревшим, потому что его вызов непредсказуем, может никогда не произойти и снижает производительность. Вместо него рекомендуется использовать try-with-resources и интерфейс AutoCloseable.
6. Коллизия – это ситуация, когда разные ключи имеют одинаковый хэш-код и попадают в одну и ту же ячейку (индекс) хэш-таблицы.
7. Метод цепочек – элементы с одинаковым индексом хранятся в связанном списке. Открытая адресация – при коллизии ищется следующая свободная ячейка по определённому правилу.
8. Данные хранятся в массиве ячеек. Каждый бакет содержит одну пару «ключ-значение» или связанную структуру (список, дерево), если произошло совпадение хэш-кодов. Хэш-функция преобразует ключ в индекс массива.

9. Если ключ уже существует, его значение обновляется новым, а старое заменяется. Количество элементов при этом не изменяется.
10. Возникает коллизия. Элементы с одинаковым хэш-кодом хранятся в одной ячейке (цепочке или дереве). Для различения объектов используется метод equals().
11. Когда число элементов превышает произведение ёмкости на коэффициент загрузки (load factor, по умолчанию 0.75), хэш-таблица увеличивает размер вдвое. После этого все элементы перехэшируются и распределяются по новым индексам.