МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ
Ордена трудового Красного Знамени федеральное государственное бюджетное
образовательное учреждение высшего образования
«Московский технический университет связи и информатики»

Кафедра Математическая кибернетика и информационные технологии

Отчет по лабораторной работе №1

Выполнил: студент группы БПИ2401

ФИО: Орлов Даниил Дмитриевич

Проверил: Харрасов Камиль Раисович

Москва, 2025

Цель работы

Закрепить знания по объектно-ориентированному программированию в Java на примере работы с пользовательскими и стандартными коллекциями. Научиться создавать и использовать собственную хэш-таблицу, реализовывать классы и взаимодействие между ними (через конструкторы, геттеры/сеттеры, методы).

1 задание – HashTable.

```
1   import java.util.LinkedList;
2   import java.util.Objects;
```

Подключаем класс LinckedList для хранения цепочек и утилитный класс Objects.

```
3   public class HashTable<K, V> {
4       private static class Entry<K, V> {
5           K key;
6           V value;
```

Создаем публичный класс HashTable с двумя параметрами V – тип ключа и K – тип значения. Далее обьявляем вложенный приватный класс, который будет хранить одну пару ключ-значение. Обьявляем поле для ключа – K и поле для значения – V, которое будет хранить связанное значение.

```
 8          Entry(K key, V value) {
 9              this.key = key;
10              this.value = value;
11          }
```

Создаем конструктор класса Entry, который будет устанавливать поля key и value.

```
13          K getKey() { return key; }
14          V getValue() { return value; }
15          void setValue(V value) { this.value = value; }
16      }
```

Создаем геттеры для ключа и записи, которые возвращают ключ и значение из записи. Далее создаем сеттер для значения, который будет изменять его.

```
18      private LinkedList<Entry<K,V>>[] table;
19      private int capacity;
20      private int size;
```

Создаем массив списков, где поле table – это массив ссылок на LinkedList<Entry<K,V>>. Далее создаем поле capacity, которое отображает текущую емкость массива table и поле size – общее число пар ключ-значение.

```
22      @SuppressWarnings("unchecked")
23      public HashTable() {
24          this.capacity = 16;
25          this.table = new LinkedList[capacity];
26          this.size = 0;
27      }
```

Создаем аннотацию @SuppressWarnings("unchecked"), которая не будет позволять компилятору ругаться на небезопасные операции с generics. Далее создаем конструктор по умолчанию, в нем указываем начальную емкость равную 16,создаем массив ссылок на LinkedList с длиной capacity и инициализируем счетчик элементов нулем.

```
29      @SuppressWarnings("unchecked")
30      public HashTable(int initialCapacity) {
31          this.capacity = Math.max(a: 1, initialCapacity);
32          this.table = new LinkedList[capacity];
33          this.size = 0;
34      }
```

Создаем конструктор с указанием начальной емкости. Для него еще раз указываем аннотацию. Чтобы избежать нулевой емкости устанавливаем capacity равной максимуму из 1 и initialCapacity. Далее создаем массив бакетов длиной capacity и инициализируем счетчик.

```
36      private int hash(K key) {
37          if (key == null) {
38              return 0;
39          }
```

Создаем приватный метод Hash, который будет вычислять индекс бакета для заданного ключа. Так как в HashMap разрешен только один null ключ, нам

нужно сделать проверку на равенство null. Если равенство соблюдается, то мы возвращаем 0 и помещаем ключ в бакет с индексом 0.

```
40          int h = key.hashCode();
41          h = h & 0x7fffffff;
42          return h % capacity;
43      }
```

Вызываем метод hashCode() у ключа чтобы получить хэш обьекта. Так как hashCode() может быть отрицательным, а индекс должен быть только положительным, нам надо применить маску, которая убирает знак (старший бит). Далее чтобы полочить индекс в пределах массива берем остаток от деления на capacity.

```
45      public void put(K key, V value) {
46          int index = hash(key);
47          if (table[index] == null) {
48              table[index] = new LinkedList<>();
49          }
```

Создаем публичный метод put, который будет добавлять пару ключ-значение или обновлять существующую. С помощью hash вычисляем индекс бакета. Далее проверяем инициализирован ли список в этом бакете. Если там null, то создаем новый LinkedList.

```
50          for (Entry<K, V> entry : table[index]) {
51              if (Objects.equals(entry.getKey(), key)) {
52                  entry.setValue(value);
53                  return;
54              }
55          }
```

Создаем список for, который будет проходить по всем Entry в списке для данного бакета. Далее сравниваем ключи через Objectc.equals(). Если оба null - вернет true, Еслиодин null - вернет false, в остальных случаях вызывает equals. Если ключ найден – обновляем ее значение новым.

```
56          table[index].add(new Entry<>(key, value));
57          size++;
58      }
```

Если ключ найден – добавляем новую запись в конец цепочки и увеличиваем счетчик общего количества элементов.

```
60      public V get(K key) {
61          int index = hash(key);
62          LinkedList<Entry<K, V>> bucket = table[index];
```

Создаем публичный метод get, который будет возвращать значение по ключу или null, если ключа нет. Сначала узнаем индекс бакета, а затем получаем ссылку на список в бакете.

```
63          if (bucket == null) return null;
64          for (Entry<K, V> entry : bucket) {
65              if (Objects.equals(entry.getKey(), key)) {
66                  return entry.getValue();
67              }
68          }
69          return null;
70      }
```

Проверяем пуст ли бакет. Если да, то возвращаем null. С помощью цикла for проходимся по всем элементам цепочки и сравниваем ключи с помощью Objects.equals. Если совпало – возвращаем связанное значение. Если мы прошли всю цепочку и ничего не нашли – возвращаем null.

```
72      public boolean remove(K key) {
73          int index = hash(key);
74          LinkedList<Entry<K, V>> bucket = table[index];
```

Создаем публичный метод remove для удаления записи по ключу. Если удаление проведено успешно – возвращаем true. Далее вычисляем индекс бакета и получаем ссылку на цепочку.

```
75          if (bucket == null) return false;
76          for (Entry<K, V> entry : bucket) {
77              if (Objects.equals(entry.getKey(), key)) {
78                  bucket.remove(entry);
79                  size--;
```

Проверяем существует ли цепочка. Если нет, то удалять нечего и возвращаем false. С помощью цикла for проходимся по записям в цепочке и проверяем

совпал ли ключ с искомым. Если да – удаляем конкретный обьект entry из списка и уменьшаем size, так как обьект был удален.

```
80                  if (bucket.isEmpty()) {
81                      table[index] = null;
82                  }
83                  return true;
84              }
85          }
86          return false;
87      }
```

Проверяем не стал ли список после удаления пустым. Если да, то очищаем слот массива и присваиваем ему null. При успешном удалении возвращаем true. Если мы прошли всю цепочку и ничего не удалили – возвращаем false.

```
89      public int size() {
90          return size;
91      }
```

Реализуем публичный метод size, который будет возвращать текущее количество элементов.

```
93      public boolean isEmpty() {
94          return size == 0;
95      }
96  }
```

Создаем публичный метод isEmpty, который будет служить для определения пуста ли таблица.

2 задание

Создадим два файла: Order.java и OrderManager.java для классов Order.java и OrderManager.java соответственно.

Файл Order.java

```java
import java.util.List;
import java.util.ArrayList;
import java.util.Objects;
```

Импортируем необходимые нам библиотеки.

```java
public class Order {
    private int orderNumber;
    private List<String> items;
    private String address;
    private double totalCost;
```

Обьявляем публичный класс Order. Создаем поля, отвечающие за номер заказа, список товаров в заказе, адрес доставки и стоимость заказа.

```java
public Order(int orderNumber, List<String> items, String address, double totalCost) {
    this.orderNumber = orderNumber;
    this.items = new ArrayList<>(items);
    this.address = address;
    this.totalCost = totalCost;
}
```

Создаем конструктор класса и передаем ему orderNumber, items, address и totalCost. Далее записываем во внутреннюю переменную класса значения, переданное в параметрах.

```java
public int getOrderNumber() { return orderNumber; }
public List<String> getItems() { return new ArrayList<>(items); }
public String getAddress() { return address; }
public double getTotalCost() { return totalCost; }
```

Создаем геттеры, с помощью которых мы будем получать значение закрытых полей.

```
23      @Override
24      public String toString() {
25          return "Order{" +
26                  "orderNumber=" + orderNumber +
27                  ", items=" + items +
28                  ", address='" + address + '\'' +
29                  ", totalCost=" + totalCost +
30                  '}';
31      }
```

Создаем метод, который будет возвращать строковое представление обьекта. С помощью @Override указываем на то, что метод переопределяет базовый метод из класса.

```
33      @Override
34      public boolean equals(Object o) {
35          if (this == o) return true;
36          if (o == null || getClass() != o.getClass()) return false;
37          Order order = (Order) o;
38          return orderNumber == order.orderNumber;
39      }
```

Создаем метод equals, который будет сравнивать два обьекта Order. С помощью this == o проверяем один и тот же ли это обьект в памяти. Если o == null или это объект другого класса - возвращаем false. Дальше приводим o к типу Order и сравниваем номера заказов. Если они одинаковые - заказы считаются равными.

```
41      @Override
42      public int hashCode() {
43          return Objects.hash(orderNumber);
44      }
45  }
```

Создаем метод hashCode(), который возвращает целое число, которое исползуется в хэш таблицах и вычисляется на основе номера заказа.

Файл OrderManager.java

```
1   import java.util.HashMap;
2
```

Импортируем класс HashMap, так как OrderManager использует его как внутреннее хранилище.

```
3   public class OrderManager {
4       private HashMap<Integer, Order> map;
```

Обьявляем публичный класс OrderManager. Далее обьявляем HashMap с ключом Integer и значением Order.

```
6       public OrderManager() {
7           map = new HashMap<>();
8       }
```

Создаем конструктор OrderManager, внутри него создаем пустую HashMap.

```
10      public void addOrder(Order order) {
11          map.put(order.getOrderNumber(), order);
12      }
```

Создаем публичный метод addOrder, который добавляет и обновляет Order в map. Ключом является order.getOrderNumber(), а значением order.

```
14      public Order findOrder(int orderNumber) {
15          return map.get(orderNumber);
16      }
```

Реализуем публичный метод findOrder, который возвращает Order по номеру или null, если ничего не найдено.

```
18      public boolean removeOrder(int orderNumber) {
19          return map.remove(orderNumber) != null;
20      }
21  }
```

Реализуем публичный метод removeOrder для удаления записи. Возвращает true если удаление прошло успешно.

Файл Main.java

```
1   import java.util.ArrayList;
2   import java.util.List;
```

Импортируем ArrayList и List, так как в дальнейшем будем создавать списки товаров.

```
4   public class Main {
5       public static void main(String[] args) {
```

Создаем публичный класс и метод Main.

```
7           OrderManager manager = new OrderManager();
8           List<String> items1 = new ArrayList<>();
9           items1.add("Книга");
10          items1.add("Блокнот");
11          Order o1 = new Order(1001, items1, "Улица Пушкина, дом 10", 29.99);
```

Создаем обьект OrderManager, который будет хранить заказы во внутреннем HashMap. Далее создаем список товаров items1 и добавляем туда два элемента. Создаем обьект Order, добавляем ему номер 1001, список items1, адрес и цену. Конструктор делает копию списка.

```
13          List<String> items2 = new ArrayList<>();
14          items2.add("Блокнот");
15          Order o2 = new Order(1002, items2, "Проспект Ленина, дом 5", 20.50);
```

Составляем второй список покупок.

```
17              manager.addOrder(o1);
18              manager.addOrder(o2);
```

Добавляем заказы в OrderManager внутри HashMap.

```
20          System.out.println("OrderManager — найден: " + manager.findOrder(1001));
21          System.out.println("OrderManager — найден: " + manager.findOrder(1002));
22          System.out.println("OrderManager — удаление 1002: " + manager.removeOrder(1002));
23          System.out.println("OrderManager — найден 1002: " + manager.findOrder(1002));
```

Демонстрация поиска заказов по номерам 1001 и 1002, удаление заказа 1002 и попытка поиска удаленного заказа.

```
26          HashTable<Integer, Order> table = new HashTable<>();
27          table.put(o1.getOrderNumber(), o1);
28          table.put(o2.getOrderNumber(), o2);
```

Создаем экземпляр собственной HashTable, добавляем в нее o1 и o2.

```
30          System.out.println("HashTable – найден: " + table.get(1001));
31          System.out.println("HashTable – найден: " + table.get(1002));
32          System.out.println("HashTable – удаление 1002: " + table.remove(1002));
33          System.out.println("HashTable – найден 1002: " + table.get(1002));
34          System.out.println("HashTable – размер: " + table.size());
35      }
36  }
```

Демонстрация различных операций для собственной HashTable.

Ответы на контрольные вопросы

1. Для чего нужен класс Object?
   Object — корневой (базовый) класс в Java: все классы неявно наследуют java.lang.Object. Он обеспечивает базовый набор методов, таких как equals(), hashCode(), toString(), getClass(), notify()/wait() и finalize() (устаревший). Благодаря этому любой объект в Java можно привести к типу Object и вызвать эти базовые методы.
2. Почему нужно переопределять методы equals() и hashCode()?
   Потому что дефолтная реализация equals() в Object сравнивает объекты по ссылке (т.е. this == other) — а часто нужно сравнивать по содержимому (например, два заказа с одинаковым номером должны считаться равными). Метод hashCode() используется в хэш-структурах (HashMap, HashSet) для размещения объекта в бакете. Если переопределяешь equals() (логика равенства по содержимому), нужно также переопределить hashCode(), чтобы объекты, равные по equals(), имели одинаковый hashCode() — иначе коллекции на основе хешей будут работать некорректно.
3. Какие есть правила переопределения equals() и hashCode()?
   Основные правила:
   - Если a.equals(b) возвращает true, то a.hashCode() должно быть равно b.hashCode(). (обязательное правило)
   - Если a.equals(b) возвращает false, то hashCode() может быть и одинаковым, и разным (коллизия допустима).
   - equals() должна быть рефлексивной (x.equals(x) — true), симметричной (x.equals(y) == y.equals(x)), транзитивной (если a.equals(b) и b.equals(c) → a.equals(c)), консистентной (при неизменных полях результат не меняется), и x.equals(null) должно возвращать false.

- hashCode() должен возвращать одинаковое значение в течение жизни объекта, если поля, участвующие в вычислении хэша, не меняются.

4. Что делает метод toString()? Почему его часто переопределяют?
   toString() возвращает строковое представление объекта. В Object он по умолчанию возвращает имяКласса@хешкодВШестнадцатеричном (например Order@6d06d69c), что малоинформативно. Его часто переопределяют, чтобы выводить полезную информацию о состоянии объекта (например, список товаров, адрес, стоимость) — это упрощает отладку и логирование.

5. Что делает метод finalize()? Почему его использование считается устаревшим (deprecated)?
   finalize() — метод из Object, который JVM вызывала перед сборкой мусора объекта, чтобы дать объекту шанс освободить ресурсы. Сейчас он устарел и не рекомендуется, потому что:
   - неопределённо, когда именно (и будет ли вообще) он вызван — непредсказуемое поведение,
   - может серьёзно замедлять сборку мусора,
   - сложно гарантировать корректное освобождение ресурсов; вместо него рекомендуется использовать конструкции try-with-resources и интерфейс AutoCloseable, а также явное освобождение ресурсов.

6. Что такое коллизия?
   Коллизия — событие, когда два разных ключа дают одинаковый хэш-код (или после приведения hashCode к диапазону бакетов оба попадают в один и тот же индекс/бакет). В результате в одной цепочке/бакете оказываются несколько записей.

7. Какие есть способы разрешения коллизий?
   Основные способы:
   - Метод цепочек (chaining) — в каждом бакете хранится список (или другой контейнер) всех элементов, попавших в этот бакет. Это то, что мы реализовали в Задании 1.
   - Открытая адресация (open addressing) — все элементы хранятся в массиве; при коллизии ищется следующая свободная позиция

(линейное пробирование, квадратичное, двойное хеширование и т.д.).

- Другие гибридные/специализированные структуры (например, деревья в бакетах при большой длине цепочек, как в современных реализациях HashMap — переход на TreeNode).

8. Как хранятся данные в хэш-таблице?
   Данные — пары ключ-значение. Где именно они хранятся: ключи и связанные с ними значения находятся в записях (Entry), размещённых в бакетах массива. В простейшей реализации массив содержит ссылки на головы цепочек (LinkedList<Entry>). Индекс бакета вычисляется как hashCode(key) → нормализация → mod capacity. В некоторых реализациях внутри бакета вместо LinkedList используются другие структуры (например, сбалансированные деревья) при длинных цепочках.

9. Что происходит, если в хэш-таблицу добавить элемент с одинаковым значением ключа?
   По правилам Map (включая HashMap), если ключ уже присутствует, то при put(key, value) старое значение заменяется новым — и возвращается/перезаписывается. Количество элементов (size) при этом не увеличивается.

10. Что происходит, если в хэш-таблицу добавить элемент с таким же хэш-кодом ключа, но разными исходными значениями?
    Это коллизия: оба ключа попадут в один бакет. При методе цепочек оба будут храниться в одной цепочке; при поиске/удалении потребуется сравнение ключей по equals() (чтобы отличить их). Если equals() возвращает false, записи считаются разными и обе хранятся.

11. Как изменяется HashMap при достижении порогового значения?
    В стандартной реализации HashMap есть параметр load factor (обычно 0.75) и текущая ёмкость (число бакетов). Когда число элементов size превышает capacity * loadFactor, HashMap выполняет rehash / resize: создаёт новый массив бакетов обычно вдвое большей ёмкости и перемещает (rehash) все существующие записи в новые бакеты (т.к. индекс зависит от capacity). Это уменьшает среднюю длину цепочек и поддерживает производительность get/put близкой к O(1) амортизированно. (Важные детали: при увеличении capacity меняются индексы элементов, поэтому требуется перераспределение.)

Вывод:

В ходе выполнения лабораторной работы была реализована собственная структура данных - **хэш-таблица**, основанная на методе цепочек для разрешения коллизий.
Были изучены основные принципы работы хэш-функций, бакетов, а также механизм хранения пар «ключ–значение».

Ссылка на репозиторий: https://github.com/qpdle/ITiP.git