МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное бюджетное образовательное учреждение

высшего образования

«УЛЬЯНОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

**Лабораторная работа № 4**

*по дисциплине «Алгоритмы и структуры данных»*

Выполнил студент

группы ПИбд-12

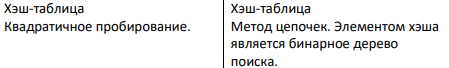
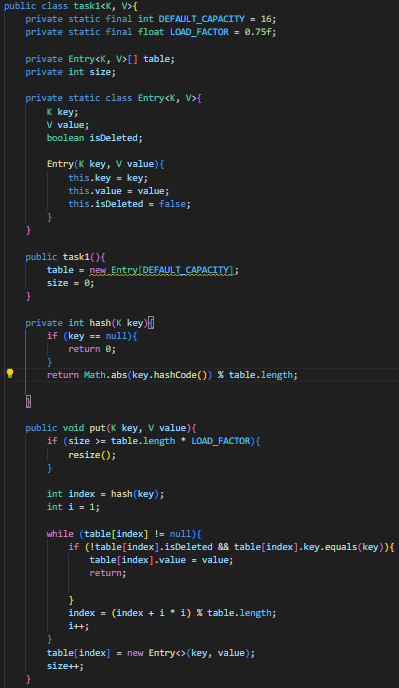
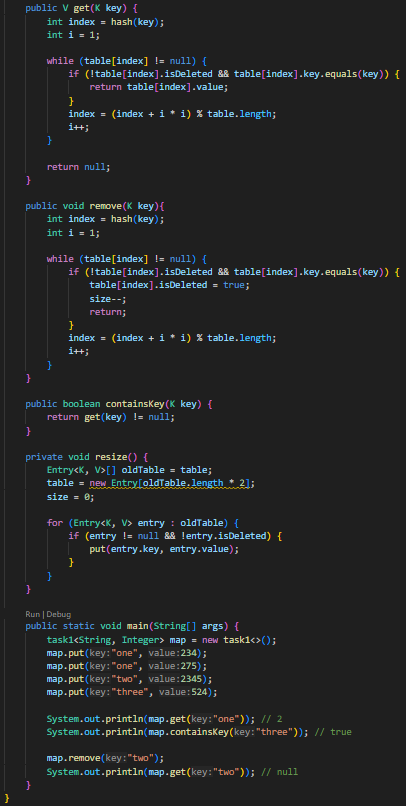
Нгуен Т. М.

Проверил доцент кафедры

«Информационные системы»

Кулешов А.В.

Ульяновск, 2025

Задачи  
  
Первое задание   
  
  
  
  
**1. Цель**

Целью данной задачи является реализация собственной структуры данных типа хеш-таблица, аналогичной HashMap из стандартной библиотеки Java. Реализация должна поддерживать операции:

* вставки элемента (put),
* получения значения по ключу (get),
* удаления элемента (remove),
* проверки наличия ключа (containsKey),  
  а также корректно обрабатывать коллизии и автоматически расширять таблицу при необходимости.

**2. Структура реализации**

**Класс Entry<K, V>**

Вложенный статический класс, представляющий отдельную запись в хеш-таблице. Содержит:

* поле key — ключ;
* поле value — значение;
* поле isDeleted — логический флаг, показывающий, был ли элемент удалён (используется для поддержки корректного пробирования).

**Основной класс task1<K, V>**

Содержит:

* массив table, представляющий хеш-таблицу;
* переменную size, отслеживающую количество активных элементов;
* константы DEFAULT\_CAPACITY и LOAD\_FACTOR, определяющие начальную ёмкость таблицы и порог заполнения для расширения;
* методы вставки, получения, удаления и проверки ключей;
* метод resize(), увеличивающий размер таблицы при достижении предельной загрузки.

**3. Основные методы**

**hash(K key)**

Вычисляет индекс в таблице по хешу ключа:

Math.abs(key.hashCode()) % table.length

**put(K key, V value)**

Добавляет новую пару ключ-значение в таблицу. В случае коллизии используется квадратичное пробирование:

index + i^2 (mod table.length)

Если ключ уже существует, значение перезаписывается. При необходимости таблица расширяется (в два раза) и выполняется повторная вставка всех существующих элементов.

**get(K key)**

Возвращает значение по заданному ключу, используя ту же схему пробирования. Пропускает удаленные элементы.

**remove(K key)**

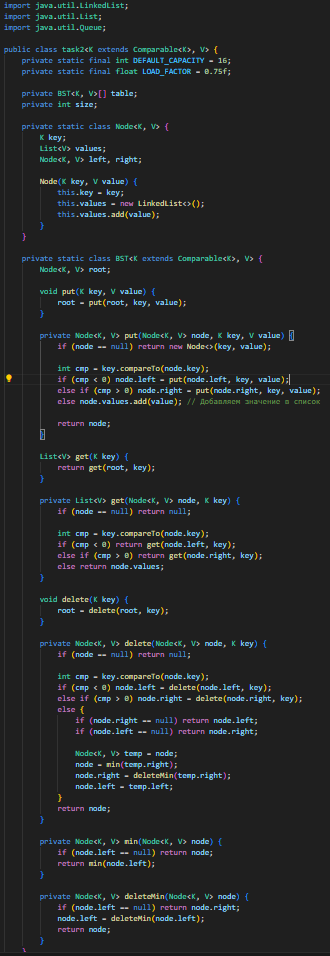
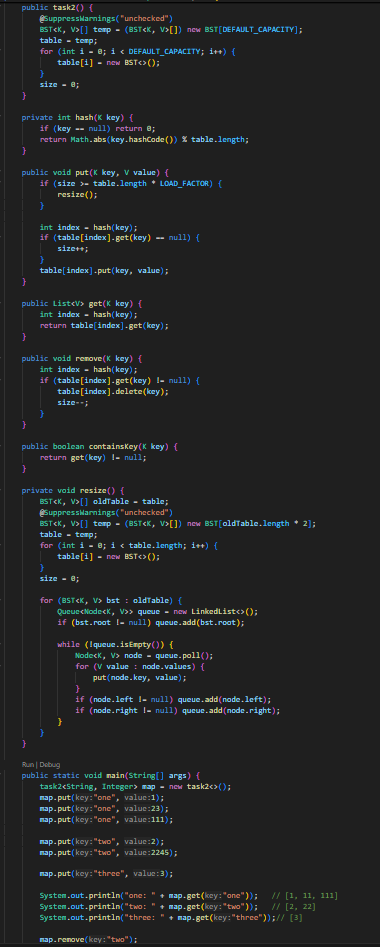
Помечает соответствующую запись как удаленную (isDeleted = true) без физического удаления, чтобы избежать нарушений логики пробирования.

**containsKey(K key)**

Возвращает true, если элемент найден методом get, иначе — false.

**resize()**

Удваивает размер массива и заново вставляет все активные элементы (isDeleted == false) в новую таблицу.

Второе задание  
  


  
  
**1. Цель**

Целью реализации является создание ассоциативного массива (Map), в котором каждая ячейка хеш-таблицы содержит бинарное дерево поиска (BST). Это повышает эффективность поиска при наличии большого количества коллизий, по сравнению со списками.

Дополнительно:

* Для одного ключа может храниться несколько значений (в List<V>).
* Поддерживаются основные операции: вставка (put), получение (get), удаление (remove) и проверка наличия ключа (containsKey).
* Таблица автоматически увеличивается при превышении пороговой загрузки.

**2. Структура реализации**

**Node<K, V>**  
Представляет узел дерева. Содержит:

* ключ key;
* список значений values, хранящих все добавленные значения для одного ключа;
* ссылки на левое и правое поддерево.

**BST<K, V>**  
Бинарное дерево поиска. Реализует:

* вставку значений (добавление в список, если ключ совпадает);
* поиск по ключу;
* удаление узла по ключу;
* нахождение минимума и удаление минимума для замены при удалении узла с двумя потомками.

**task2<K, V>**  
Основной класс хеш-таблицы, где каждая ячейка — это объект BST.  
Содержит:

* массив table, состоящий из BST;
* поле size, отслеживающее количество уникальных ключей;
* методы вставки, поиска, удаления и проверки наличия ключа;
* метод resize(), увеличивающий массив вдвое при превышении загрузки.

**3. Основные методы**

**hash(K key)**  
Вычисляет индекс хеш-таблицы:

Math.abs(key.hashCode()) % table.length;

**put(K key, V value)**  
Добавляет значение в соответствующее дерево. Если ключ новый — увеличивается size.

**get(K key)**  
Возвращает список всех значений, соответствующих ключу.

**remove(K key)**  
Удаляет узел дерева, соответствующий ключу. Если ключ найден — size уменьшается.

**containsKey(K key)**  
Проверяет, существует ли ключ в таблице (использует get()).

**resize()**  
Увеличивает размер таблицы в два раза. Все деревья из старой таблицы обходятся в ширину (BFS), и их значения вставляются в новую таблицу.