Lab: Hash Tables Sets and Maps

This document defines the lab for ["Data Structures – Advanced (Java)" course @ Software University](https://softuni.org).

Please submit your solutions (source code) of all below described problems in [Judge](https://judge.softuni.org).

Write Java code for solving the tasks on the following pages. Code should compile under the Java 8 and above standards you can write and locally test your solution with the Java 13 standard, however, **Judge will run the submission with Java 10 JRE**. Avoid submissions with **features included after the Java 10** release doing **otherwise** will result in a **compile time error**.

Any code files that are part of the task are provided as **Skeleton**. In the beginning, import the project skeleton, do not change any of the interfaces or classes provided. You are free to add additional logic in the form of methods in both interfaces and implementations you are not allowed to delete or remove any of the code provided. Do not change the names of the files as they are part of the test logic. **Do not change the packages** or move any of the files provided inside the skeleton if you have to add a new file add it in the same package of usage.

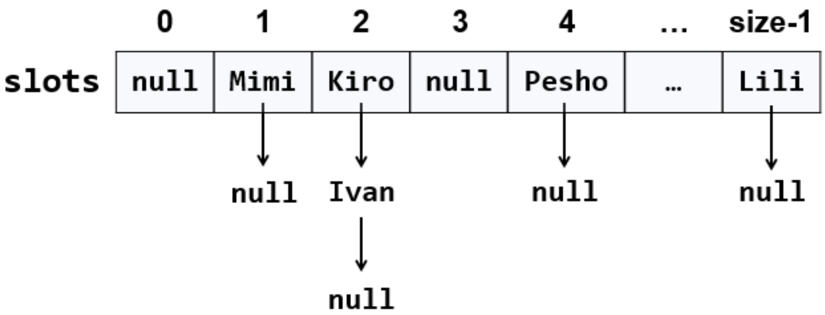
Some **tests may be provided** within the skeleton – use those for local **testing and debugging**, however, there **is no guarantee that there are no hidden tests added inside Judge**.

Please follow the exact instructions for uploading the solutions for each task. Submit as **.zip archive** the files contained inside **"...\src\main\java"** folder this should work for all tasks regardless of the current DS implementation.

In order for the solution to compile the tests **successfully** the project **must** have a **single Main.java** file containing a single **public static void main(String[] args)** method even an empty one within the **Main class**.

Some of the problems will have simple **Benchmark tests** inside the skeleton. You can try to run those with **different values** and **different implementations** in order to **observe** behavior. However, **keep** in mind that the result comes **only as numbers** and this data may be **misleading** in some situations. Also, the tests are not started from the command prompt which may **influence** the **accuracy** of the results. Those tests are only added as an **example** of **different data structures performance** on their **common** operations.

You must implement a **hash table** that uses **chaining in a linked list** as a collision resolution strategy:



The hash table will hold its **elements** (key-value pairs) in a class **`KeyValue<Key, Value>`**. The hash table will consist of **slots**, each holding a **linked list of key-value pairs**: **`LinkedList<KeyValue<Key, Value>>`**.

Problem 1. Learn about Hash Tables in Wikipedia

Before starting, get familiar with the concept of **hash table**: https://en.wikipedia.org/wiki/Hash_table. Note that there are many collision resolution strategies like chaining and open addressing. We will use one of the simplest strategies: **chaining** elements with collisions in a linked list.

The typical **operations** over a hash table are **add** or **replace**, **find** and **remove**. Additional operations are **enumerate all elements**, **enumerate all keys**, **enumerate all values** and **get count**. Let's start coding!

Problem 2. HashTable<K, V> – Project Skeleton

You are given an unfinished project holding the class **`KeyValue<Key, Value>`**, the unfinished class **`HashTable<K, V>`** and **unit tests** for its entire functionality.

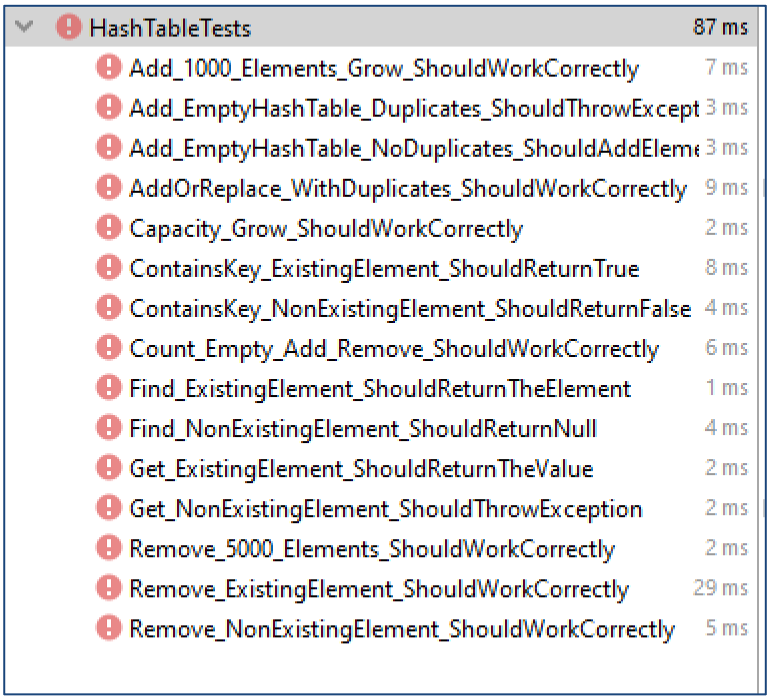
Your goal is to implement the missing functionality to finish the project.

First, let's look at the **`KeyValue<Key, Value>`** class. It holds a **key-value pair** of parameterized types **Key** and **`Value`**. To enable comparing key-value pairs, the class implements **`equals(…)`** and **`hashCode()`**. It also has **`toString()`** method to enable printing it on the console and viewing it inside the debugger. The **`KeyValue<Key, Value>`** class comes out-of-the-box with the project skeleton, so you will not need to change it:

The project comes also with **unit tests** covering the entire functionality of the hash table see the class **`HashTableTests`**

Problem 3. Run the Unit Tests to Ensure All of Them Initially Fail

Run the unit tests from the **`HashTableTests`** class. All of them should fail:



This is quite normal. We have unit tests, but the code covered by these tests is missing. Let's write it.

Problem 4. Define the Hash Table Internal Data

The first step is to define the inner **data** that holds the hash table elements:

- **`LinkedList<KeyValue<Key, Value>>[] slots`** – an array that holds the slots in the hash table
  - Each slot is either empty (**`null`**) or holds a **linked list** of elements with the same hash code
- **`int count`** – holds the number of elements in the hash table
- **`int capacity`** – holds the number of slots in the hash table
- Thus, the hash table **fill factor** can be calculated by **`Count / Capacity`**

The code might look like this:

```
public class HashTable<K, V> implements Iterable<KeyValue<K, V>> {
    private static final int INITIAL_CAPACITY = 16;
    private static final double LOAD_FACTOR = 0.80d;

    private LinkedList<KeyValue<K, V>>[] slots;
    private int count;
    private int capacity;
```

Problem 5. Implement the Hash Table Constructor

Now, let's implement the hash table **constructor**. Its purpose is to allocate the slots that will hold the hash table elements. The hash table constructor has two forms:

- Parameterless constructor – should allocate 16 slots (16 is the default initial hash table capacity)
- Constructor with parameter **`capacity`** – allocates the specified capacity in the underlying array (slots)

Problem 6. Implement the Add(key, value) Method

Now, we are ready to implement the most important method **`add(key, value)`** that inserts a new element in the hash table. It should take into account several things:

- Detect **collisions** and resolve them by **chaining** the elements in a linked list.
- Detect **duplicated keys** and throw an exception.
- **grow** the hash table if needed (resize to double capacity when the fill factor is too high).

The **`add(key, value)`** method might look like this:

```
public void add(K key, V value) {
    this.growIfNeeded();
    int slotNumber = this.findSlotNumber(key);
    if (this.slots[slotNumber] == null) {
        this.slots[slotNumber] = new LinkedList<>();
    }

    for (KeyValue<K, V> element : slots[slotNumber]) {
        if (element.getKey().equals(key)) {
            throw new IllegalArgumentException("Key already exists: " + key);
        }
    }

    KeyValue<K, V> kvp = new KeyValue<>(key, value);
    this.slots[slotNumber].addLast(kvp);
    this.count++;
}
```

How it works? First, if the hash table is full, **grow** it (resize its capacity to 2 times bigger capacity). This will be discussed later. We can leave the **`growIfNeeded()`** method empty.

Next, **find the slot** that should hold the element to be added. The slot number is calculated by the **hash value** of the key. Typically, the **hashCode()** method. We need a number in the range [0 … size-1] so we take the modulus of the hash code:

```
private int findSlotNumber(K key) {
    return Math.abs(key.hashCode()) % this.slots.length;
}
```

We take the absolute value because **hashCode()** sometimes returns negative numbers.

Once we have the slot number, it is either empty (**null**) or holds a **linked list** of elements with the same hash code as the new element. In both cases, we should have in the target slot a **linked list** holding the elements with the same hash value as the **key**.

We **check for duplicated keys** and throw an exception if the same key already exists. Then we **append the new element** at the end of the linked list in the target slot of the hash table and increase **this.count**.

Problem 7. Implement the Enumerator (Iterable<T>)

Now let's implement the enumerator: a method that passes through all elements in the hash table exactly once. The hash table holds key-value pairs (**KeyValue<Key, Value>**) elements, so we need to implement the interface **Iterable<KeyValue<Key, Value>>**.

Problem 8. Implement Find(key)

Let's implement the second most important operation after adding a key-value pair – **finding an element by key**. The **find(key)** method should either **return the element** by its key or **return null** if the key does not exist:

```
public KeyValue<K, V> find(K key) {
    int slotNumber = this.findSlotNumber(key);
    LinkedList<KeyValue<K, V>> elements = this.slots[slotNumber];
    if (elements != null) {
        for (KeyValue<K, V> element : elements) {
            if (element.getKey().equals(key)) {
                return element;
            }
        }
    }
    return null;
}
```

The above code works as follows:

1. **Finds the slot** holding the specified key (by calculating the hash code modulus the hash table size).
2. **Passes through all elements in the target slot** (in its linked list) and compare their key with the target key.

Problem 9. Implement Get(key) and ContainsKey(key) Methods

Once we have the **`find(key)`** method, it is easy to implement the methods that directly depend on it:

- **`get(key)`** – returns the element by given key or throws and exception when the key does not exist
- **`containsKey(key)`** – returns whether the key exists in the hash table

Let's start with the **`get(key)`** method:

```java
public V get(K key) {
    KeyValue<K, V> element = this.find(key);
    if (element == null) {
        throw new IllegalArgumentException();
    }
    return element.getValue();
}
```

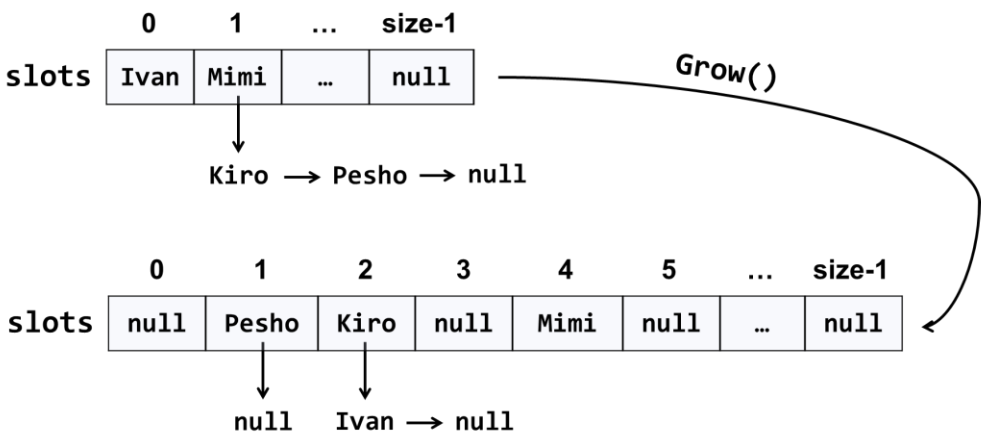
The **`containsKey(key)`** method is trivial. Implement it yourself:

Problem 10. Implement the GrowIfNeeded() and Grow() Methods

The **`growIfNeeded()`** method checks whether the hash table should grow. The hash table should grow when it has filled its capacity to **more than 75%** (load factor > 75%) and we are trying to add a new element. In this case, it first calls **`grow()`**, otherwise does nothing:

```java
private void growIfNeeded() {
    if ((double) (this.size() + 1) / this.capacity() > LOAD_FACTOR) {
        this.grow();
    }
}
```

The **`grow()`** method allocates a **new hash table** with **double capacity** and adds the old elements in the new hash table, then replaces the old hash table with the new one:



The code might look like this:

```java
private void grow() {
    HashTable<K, V> newHashTable = new HashTable<>( capacity: 2 * this.slots.length);
    for (LinkedList<KeyValue<K, V>> element : this.slots) {
        if (element != null) {
            for (KeyValue<K, V> keyValue : element) {
                newHashTable.add(keyValue.getKey(), keyValue.getValue());
            }
        }
    }

    this.slots = newHashTable.slots;
    this.count = newHashTable.count;
}
```

Problem 11. Implement AddOrReplace(key, value)

The method **addOrReplace(key, value)** is very similar to the **add(key, value)** method. The only difference is the **add(key, value)** throws and exception when the key is found to already exist in the hash table, while in the same situation, **addOrReplace(key, value)** replaces the **value** in the element holding the **key**, with the **new value** passed as argument.

Hint: copy/paste the code from **add(key, value)** and slightly modify its logic.

Implement **addOrReplace(key, value)** yourself.

Problem 12. Implement Remove(key)

The next important functionality waiting to be implemented is **removing an element by its key**. The method **remove(key)** should either:

- Successfully **remove the element** (when the key exists) from the hash table and return **true**.
- Return **false** when the key does not exist in the hash table.

The **remove(key)** method is not trivial. It should first **find the slot** that is expected to hold the key, then **traverse the linked list** from its first to its last element and **remove the element** in case the key is found and return **true**. Otherwise, it should return **false**:

Problem 13. Implement Clear()

The **clear()** method is trivial. It should reinitialize **this.slots** and **this.count**, like it was initially done in the hash table constructor. Implement it yourself.

Problem 14. Implement Keys and Values

Now implement the last piece of missing functionality: **enumerating all keys and values**. We already have an enumerator that returns all elements from the hash table. We just need to filter (select) the keys/values:

"Mort was already aware that love made you feel hot and cold and cruel and weak, but he hadn't realized that it could make you stupid." — *Terry Pratchett, Mort*