**Data Structures Fundamentals with Java Exam - 05 April 2020**

This document defines the examination problems for ["Data Structures – Fundamentals (Java)" course @ Software University](https://softuni.bg/trainings/2812/data-structures-fundamentals-with-java-march-2020).

Please submit your solutions (source code) of all below described problems in [Judge](https://judge.softuni.bg/Contests/2327/Data-Structures-Fundamentals-with-Java-Exam-05-Apr-2020).

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 Java 10** release doing **otherwise** will result in **compile time error**.

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 on 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 current DS implementation.

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

You have to **study** the provided **skeleton**. The code is **separated** inside **different** **packages**, for you tasks you should be writing code **mainly** **inside** **the "core" package**.

There **are** **few** **entities** **inside** the **project** you are **allowed** to **add** code to those, also you have to study the classes provided.

**All the** entities objects are stored, ordered, created and used inside the tests **by their unique Integers (IDs or Weights). You can use that for priority, search, object identification, comparison or any other operation that require to map an object to exact unique value.**

# Scheduler

You are given a skeleton with a class **ProcessScheduler** that implements the **Scheduler interface.**

Keep in mind that this class should provide **task schedulin**g which means that when we add task and we can process it **only if there are no previously added tasks**. So for example we can **add** the tasks: **"print"**, **"open\_browser"**, **"create\_folder"** etc… When we process the tasks each one is served in the order it was added so the **process** of the above tasks will be: **"print"**, **"open\_browser"** and finally **"create\_folder"**. Unless we **reschedule** some of the tasks (for that look at the **reschedule** **method description**).

The ProcessScheduler class works with **ScheduledTask** those tasks have two main properties:

* **int id –** the **unique** identification field and also can be used to for **comparison** **based** **operations** with those objects.
* **String description** – this is simply the description of the Task.

The following methods are **not** **implemented** you have to **implement** **them**:

* **void add(Task task) –** adds task after the **last** **task** currently stored and increases the count of tasks
* **Task process() –** processes the **first** stored task **in line** and **removes** it from the schedule **reducing** its size
* **Task peek() –** returns the **first** stored task **in line** **without removing** it from the schedule
* **Boolean contains(Task task) –** returns **true** or **false** based on **whether** the task is **stored** inside the schedule
* **int size() –** returns the **number** of tasks stored
* **Boolean remove(Task task) –** attempts to **remove** the task from the schedule and **returns** **true** or **false** based on whether the operation is **successful** or **not**. If there is **no such task** in the schedule **throw** **IllegalArgumentException**
* **Boolean remove(int id) –** attempts to **remove** the task **(based on Id)** from the schedule and **returns** **true** or **false** based on whether the operation is **successful** or **not**. If there is **no such task** in the schedule **throw** **IllegalArgumentException**
* **void insertBefore(int id, Task task)** – **attempts** to insert **new task** **before** the task **specified by the Id** **parameter** if there is **no such task** previously stored throw **IllegalArgumentException**. This operation also should **increase** the size as side effect
* **void insertAfter(int id, Task task)** – **attempts** to insert **new task** **after** the task **specified by the Id** **parameter** if there is **no such task** previously stored throw **IllegalArgumentException**. This operation also should **increase** the size as side effect
* **void clear()** – removes **all tasks** from the schedule
* **Task[] toArray()** – returns **all tasks** as an **array**
* **void reschedule(Task first, Task second)** – reschedules two tasks **previously** **stored** inside the schedule, this means that **simply** the two tasks are **swapped** after the operation completes. However you have to be **sure** **both tasks are stored** inside the schedule, **otherwise** throw **IllegalArgumentException**
* **List<Task> toList()** – returns all tasks as List<Tasks>
* **void reverse()** – **reverses** the order of **tasks** stored inside the schedule
* **Task find(int id)** – **finds** and **returns** the task with the corresponding Id from the schedule (**without removing it**) if there is **no such** task sored throw **IllegalArgumentException**
* **Task find(Task task)** – **finds** and **returns** the task from the schedule (**without removing it**) if there is **no such** task sored throw **IllegalArgumentException**

1. **Data Transfer System**

You are given a skeleton with a class **MessagingSystem** that implements the **DataTransferSystem interface.**

For this problem you have to **implement** **message** **storage** system that works with **TextMessages** those messages have two main properties:

* **int weight –** the **unique** identification field and also can be used to for **comparison** **based** **operations** with those objects. You can **associate** this field with something like **importance** or **priority**
* **String text** – this is simply the message information

The system is centered around the logic for message storage **based on the message's weight**. This means that the messages are stored in **hierarchal order**. There is one **main rule** when adding the new entities you have to build the hierarchy in such a way that the operations you have **are optimal**.

**Hint:** think of some data structure with **additional** **relations** between the objects stored.

The following methods **are not** **implemented** your task is to **implement them:**

* **void add(Message message)** – **adds** new message to the storage system **if the message is previously stored** instead of adding **throw** **IllegalArgumentException**. This operation should also **increase** the size
* **Message getByWeight(int weight)** – returns the message with the **corresponding** **weight** if present if not throw **IllegalArgumentException**
* **Message getLightest()** – returns the message with the **lowest** **weight** if the storage is **empty** throw **IllegalStateException**
* **Message getHeaviest()** – returns the message with the **highest weight** if the storage is **empty** throw **IllegalStateException**
* **Message deleteLightest()** – deletes and returns the message with the **lowest** **weight** if the storage is **empty** throw **IllegalStateException.** This operation should also **decrease** the size
* **Message deleteHeaviest()** – deletes and returns the message with the **highest weight** if the storage is **empty** throw **IllegalStateException.** This operation should also **decrease** the size
* **Boolean contains(Message message)** – returns **true** or **false** whether the message is **stored** or **not**
* **List<Message> getOrderedByWeight()** – returns the elements stored **ordered** **by their weight** in **increasing** **order**. If there are no elements return an **empty** List<Message>
* **List<Message> getPostOrder()** – returns the elements stored **in post order**. If there are no elements return an **empty** List<Message>
* **List<Message> getPreOrder()** – returns the elements stored **in pre order**. If there are no elements return an **empty** List<Message>
* **List<Message> getInOrder()** – returns the elements stored **in order**. If there are no elements return an **empty** List<Message>
* **int size()** – returns the **number** of elements stored

## Performance Tests

For this task you will only be required to submit the **code from the previous two problems**. Some part of the the tests will **test the performance of the first task**, the **other** one will tests the **second**. If you are having problem with this task you should **perform detailed algorithmic complexity analysis**, and try to **figure** **out** **weak** spots inside your implementation.

For this problem it is important that other operations are **implemented** **correctly** according to the specific problems: **add**, **size**, **remove**, **find** etc…

You can submit code to this problem **without full coverage** from the previous two problems, **not all test cases** will be considered only the **general** **behaviour** will be considered important, **edge** **cases** will mostly be ignored for this problem, however the performance will be measured for all operations.

"Truth is ever to be found in simplicity, and not in the multiplicity and confusion of things." – Isaac Newton